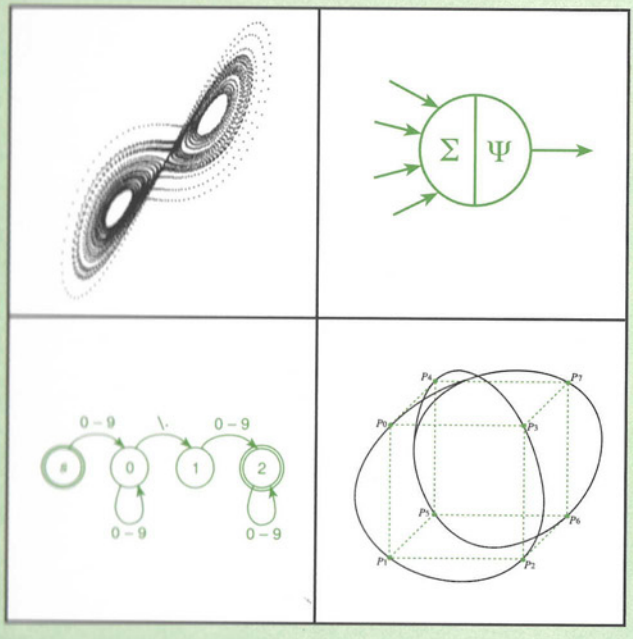
PROBLEMS & SOLUTIONS IN SCIENTIFIC COMPUTING

WITH C++ AND JAVA SIMULATIONS



Willi-Hans Steeb
Yorick Hardy
Alexandre Hardy
Ruedi Stoop

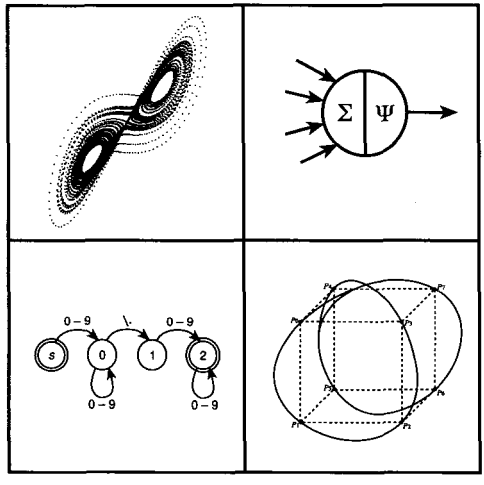
World Scientific

PROBLEMS & SOLUTIONS IN SCIENTIFIC COMPUTING

WITH C++ AND JAVA SIMULATIONS

PROBLEMS & SOLUTIONS IN SCIENTIFIC COMPUTING

WITH C++ AND JAVA SIMULATIONS



Willi-Hans Steeb

Yorick Hardy

Alexandre Hardy

Rand Afrikaans University, South Africa

Ruedi Stoop

Institute for Neuroinformatics, ETHZ, Switzerlan

World Scientific

NEW JERSEY • LONDON • SINGAPORE • BEIJING • SHANGHAI • HONG KONG • TAIPEI • CHENNAI

Published by

World Scientific Publishing Co. Pte. Ltd.

5 Toh Tuck Link, Singapore 596224

USA office: 27 Warren Street, Suite 401-402, Hackensack, NJ 07601

UK office: 57 Shelton Street, Covent Garden, London WC2H 9HE

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library.

PROBLEMS AND SOLUTIONS IN SCIENTIFIC COMPUTING WITH C++ AND JAVA SIMULATIONS

ISBN 981-256-112-9

ISBN 981-256-125-0 (pbk)

Printed in Singapore.

Preface

Scientific computing is a collection of tools, techniques and theories required to develop and solve mathematical models in science and engineering on a computer. The purpose of this book is to supply a collection of problems together with their detailed solution which will prove to be valuable to students as well as to research workers in the fields of scientific computing. The book provides the various skills and techniques needed in scientific computing. The topics range in difficulty from elementary to advanced. Almost all problems are solved in detail and most of the problems are self-contained. A number of problems contain C++ or Java code. All fields in scientific computing are covered such as matrices, numerical analysis, neural networks, genetic algorithms etc. All relevant definitions are given. Students can learn important principles and strategies required for problem solving. Chapter 1 gives a gentle introduction to problems in scientific computing. Teachers will also find this text useful as a supplement, since important concepts and techniques are developed in the problems. Basic knowledge in linear algebra, analysis, C++ and Java programming are required. We have tested the C++ programs with gcc 3.3 and Microsoft Visual Studio.NET (VC 7). The Java programs have been tested with version 1.5.0. The material was tested in our lectures given around the world.

Any useful suggestions and comments are welcome.

email addresses of the authors:

steeb_wh@yahoo.com
whs@na.rau.ac.za
yorickhardy@yahoo.com
yha@na.rau.ac.za
ah@na.rau.ac.za
ruedi@ini.phys.ethz.ch

Home pages of the author:

http://issc.rau.ac.za

Contents

Notation

$\emptyset$	empty set
$\mathbf{N}$	natural numbers
$\mathbf{Z}$	integers
$\mathbf{Q}$	rational numbers
$\mathbf{R}$	real numbers
$\mathbf{R}^+$	nonnegative real numbers
$\mathbf{C}$	complex numbers
$\mathbf{R}^n$	n-dimensional Euclidian space
$\mathbf{C}^n$	n-dimensional complex linear space
i	$\sqrt{-1}$
$\Re z$	real part of the complex number z
$\Im z$	imaginary part of the complex number z
$\mathbf{x} \in \mathbf{R}^n$	element $\mathbf{x}$ of $\mathbf{R}^n$
$A \subset B$	subset A of set B
$A \cap B$	the intersection of the sets A and B
$A \cup B$	the union of the sets A and B
$f \circ g$	composition of two mappings $(f \circ g)(x) = f(g(x))$
$\lfloor x \rfloor$	floor function $\lfloor 3.14 \rfloor = 3$
$\lceil x \rceil$	ceiling function $\lceil 3.14 \rceil = 4$
$\oplus$	XOR operation
u	dependent variable
t	independent variable (time variable)
x	independent variable (space variable)
$\mathbf{x}^T = (x_1, x_2, \ldots, x_n)$	vector of independent variables, ^{T} means transpose
$\mathbf{u}^T = (u_1, u_2, \ldots, u_n)$	vector of dependent variables, ^{T} means transpose
$\|\|\cdot\|\|$	norm
$\mathbf{x} \cdot \mathbf{y}$	scalar product (inner product)
$\mathbf{x} \times \mathbf{y}$	vector product
$\otimes$	Kronecker product, tensor product
det	determinant of a square matrix
tr	trace of a square matrix
I	unit matrix
$[\,,\,]$	commutator
δ_{jk}	Kronecker delta with $\delta_{jk} = 1$ for $j = k$ and $\delta_{jk} = 0$ for $j \neq k$
$\mathrm{sgn}(x)$	the sign of x, 1 if $x > 0$, -1 if $x < 0$, 0 if $x = 0$
λ	eigenvalue
ϵ	real parameter

Chapter 1

Quickies

Problem 1. The multiplication of the polynomials

$$f(x) = f_0 + f_1 x, \qquad g(x) = g_0 + g_1 x$$

yields the polynomial

$$h(x) = h_0 + h_1 x + h_2 x^2$$

where

$$h_0 = f_0 g_0, \quad h_1 = f_0 g_1 + f_1 g_0, \quad h_2 = f_1 g_1 .$$

This includes 4 multiplications and 1 addition to find the coefficients h_0, h_1, h_2. Is it possible to reduce the number of multiplications to 3?

Solution 1. The number of multiplications can be reduced to 3 using

$$h_0 = f_0 g_0, \qquad h_2 = f_1 g_1$$

and

$$h_1 = (f_0 + f_1)(g_0 + g_1) - h_0 - h_2 .$$

However, the number of additions is now 2 and we have 2 subtractions.

Problem 2. How would we calculate the function

$$f(x, y) = \cos(x)\sin(y) - \sin(x)\cos(y)$$

where $x, y \in \mathbf{R}$?

Solution 2. In the present form we have to calculate the cosine twice and the sine twice. Additionally we have two multiplications and one subtraction. Using the *trigonometric identity*

$$\cos(x)\sin(y) - \sin(x)\cos(y) \equiv \sin(x-y)$$

we have

$$f(x,y) = \sin(x-y)\,.$$

Thus we have reduced the number of operations considerably. We only have to calculate the difference $x - y$ and then the sine.

Problem 3. Assume we have to calculate the surface area A and the volume V of a ball with radius r, i.e.,

$$\begin{aligned} A &= 4\pi r^2 \\ V &= \frac{4}{3}\pi r^3\,. \end{aligned}$$

How can we reduce the number of multiplications used to calculate these two quantitites?

Solution 3. Obviously we can write

$$\begin{aligned} A &= 4\pi r^2 \\ V &= \frac{1}{3} A r \end{aligned}$$

to obtain 5 multiplications compared to 7 from the original formulas.

Problem 4. In a C++ program we find the following `if` condition

```
if((i/2)*2 == i) {  ...  }
```

where `i` is of data type `int`. Obviously the condition tests whether `i` is an odd or even number. How can the condition be simplified?

Solution 4. Obviously, we can write

```
if(i%2 == 0) { ... }
```

which is also faster. Another option is

```
if((i & 1) == 0) { ... }.
```

Problem 5. How would we calculate

$$\frac{\sin(x)}{x}$$

for $x \in \mathbf{R}$ and $x \ll 1$?

Solution 5. Since x is small we would not use the expression given above, since it involves the division of two small numbers. Additionally we have to calculate $\sin(x)$. Rather we expand $\sin(x)$ as a *Taylor series*. This yields

$$\frac{\sin(x)}{x} = \frac{x - x^3/3! + \cdots}{x} = 1 - \frac{x^2}{3!} + \cdots$$

For small x the term $1 - x^2/6$ provides a good enough approximation.

Problem 6. Let A be a square matrix over the real numbers. How would we calculate

$$\det(\exp(A))$$

where

$$\exp(A) := \sum_{k=0}^{\infty} \frac{A^k}{k!}\,.$$

Solution 6. To calculate $\exp(A)$ of an $n \times n$ square matrix A using the definition given above is quite time-consuming. Additionally we have to calculate the determinant of A. A better solution is to use the identity

$$\det(\exp(A)) \equiv \exp(\mathrm{tr}(A))$$

where $\mathrm{tr}(A)$ denotes the *trace*, i.e.,

$$\mathrm{tr}(A) := \sum_{j=1}^{n} a_{jj}\,.$$

After taking the trace we only have to calculate the exponent of a real number instead of a matrix.

Problem 7. (i) Find a linear map

$$f : \{0, 1\} \to \{-1, 1\}$$

such that

$$f(0) = -1, \qquad f(1) = 1\,. \tag{1}$$

(ii) Find a linear map

$$g : \{-1, 1\} \to \{0, 1\}$$

such that

$$g(-1) = 0, \qquad g(1) = 1\,. \tag{2}$$

This is obviously the inverse map of f.

Solution 7. (i) From the ansatz for a linear function $f(n) = an + b$, where a and b are determined by condition (1) we find $f(0) = b = -1$ and $f(1) = a + b = 1$. Thus,

$$f(n) = 2n - 1\,.$$

(ii) We start again from $g(m) = cm + d$, where the constants c and d are determined by the condition (2). We find $c = 1/2$ and $d = 1/2$. Thus,

$$g(m) = \frac{m+1}{2}\,.$$

Problem 8. The *standard map* is given by

$$\begin{aligned} I_{t+1} &= I_t + k\sin(\theta_t) \\ \theta_{t+1} &= \theta_t + I_t + k\sin(\theta_t) \end{aligned}$$

where $t = 0, 1, 2, \ldots$ and I_0, θ_0 are the given initial values. k is a positive constant. How would we simplify the calculation of I_t and θ_t?

Solution 8. Obviously, we can insert the first equation into the second equation. This yields

$$\begin{aligned} I_{t+1} &= I_t + k\sin(\theta_t) \\ \theta_{t+1} &= \theta_t + I_{t+1}\,. \end{aligned}$$

This saves the calculation of the sine and of an addition.

Problem 9. Given the *time-delayed logistic map*

$$x_{t+2} = rx_{t+1}(1 - x_t)$$

where $t = 0, 1, 2, \ldots$, r is a positive constant and x_0, x_1 are the given initial values. Show that it can be reformulated as a pair of first order difference equations.

Solution 9. Setting $y_t = x_{t+1}$ we find

$$\begin{aligned} y_{t+1} &= ry_t(1 - x_t) \\ x_{t+1} &= y_t\,. \end{aligned}$$

Problem 10. Let A be an $n \times n$ matrix with $\det(A) \neq 0$. Thus, the inverse A^{-1} exists. The inverse can be calculated using differentiation as follows

$$\frac{\partial}{\partial a_{ij}} \ln(\det(A)) = b_{ji}$$

where $B = A^{-1}$. Apply the formula to a 2×2 matrix to find the inverse.

Solution 10. Since

$$\det(A) = a_{11}a_{22} - a_{12}a_{21}$$

we have

$$\begin{aligned}
b_{11} &= \frac{\partial}{\partial a_{11}} \ln(a_{11}a_{22} - a_{12}a_{21}) = \frac{a_{22}}{D} \\
b_{12} &= \frac{\partial}{\partial a_{21}} \ln(a_{11}a_{22} - a_{12}a_{21}) = -\frac{a_{12}}{D} \\
b_{21} &= \frac{\partial}{\partial a_{12}} \ln(a_{11}a_{22} - a_{12}a_{21}) = -\frac{a_{21}}{D} \\
b_{22} &= \frac{\partial}{\partial a_{22}} \ln(a_{11}a_{22} - a_{12}a_{21}) = \frac{a_{11}}{D}
\end{aligned}$$

where $D = \det(A)$.

Problem 11. Calculate

$$I_n = \int_0^1 x^n e^x dx$$

for $n = 0, 1, 2, \ldots$.

Solution 11. To do numerical integration for every n is not very efficient. We try to find a recursion relation for I_n. Using *integration by parts* we find

$$I_{n+1} = \int_0^1 x^{n+1} e^x dx = \left| x^{n+1} e^x \right|_0^1 - (n+1) \int_0^1 x^n e^x dx \,.$$

Thus,

$$I_{n+1} = e - (n+1) I_n$$

with $I_0 = e - 1$. Thus we can avoid any numerical integration.

Problem 12. Which of the following two initializations to 1 of a two-dimensional array (matrix) is faster? Explain!

```
// init.cpp

#include <iostream>
using namespace std;

int main(void)
{
   // initialization 1
   int array1[128][128];
   for(int i=0;i<128;i++)
   {
   for(int j=0;j<128;j++)
   {
   array1[i][j] = 1;
   }
   }

   // initialization 2
   int array2[128][128];
   for(int k=0;k<128;k++)
   {
   for(int l=0;l<128;l++)
   {
   array2[l][k] = 1;
   }
   }
   return 0;
}
```

Solution 12. The two-dimensional array is stored in a linear (one-dimensional) array. We note that

```
array1[i][j] is equivalent to *(array1+i*128+j)
array2[l][k] is equivalent to *(array2+l*128+k).
```

The first initialization is faster since iteration over the second index involves initializing adjacent bytes. The second initialization involves iteration over the first index which are separated by at least 128 `int`. Thus, the first initialization uses primarily increment and copy operations, whereas the second uses primarily addition and copy operations. When the processor uses a memory cache the first initialization method is more efficient since each sub-array can often be stored in the cache for the initialization.

Problem 13. Consider the following sets

```
A :  0, 1, -1, 2, -2, 3, -3, ...
B :  1, 2,  3, 4,  5, 6,  7, ...
```

(i) Find a function $f : B \to A$ which sets up a 1-1 map.
(ii) Find the inverse map $g : A \to B$.
(iii) Give a Java implementation for these maps using the `BigInteger` class.

Solution 13. (i) We have

$$f(n) = \begin{cases} \frac{n}{2} & \text{if } n \text{ even} \\ -\frac{n-1}{2} & \text{if } n \text{ odd} \end{cases}.$$

(ii) We have

$$g(m) = \begin{cases} 2m & \text{if } m \text{ positive} \\ 2|m| + 1 & \text{if } m \text{ negative or zero} \end{cases}.$$

(iii) The method `int signum()` in class `BigInteger` returns the signum function of this `BigInteger`, i.e., it returns -1, 0, or 1 as the value of this `BigInteger` is negative, zero or positive.

```
// OneOneMap.java

import java.math.*;

public class OneOneMap
{
   public static BigInteger f(BigInteger n)
   {
   BigInteger TWO = new BigInteger("2");
   BigInteger rem = n.remainder(TWO);
   if(rem.equals(BigInteger.ZERO))
   { return n.divide(TWO); }
   else
   return ((n.subtract(BigInteger.ONE)).divide(TWO)).negate();
   }

   public static BigInteger g(BigInteger m)
   {
   BigInteger TWO = new BigInteger("2");
   int s = m.signum();
   if(s == 1) { return m.multiply(TWO); }
   else
   return (m.abs()).multiply(TWO).add(BigInteger.ONE);
```

```
    public static void main(String[] args)
    {
    BigInteger n1 = new BigInteger("-1001");
    BigInteger r1 = f(n1);
    System.out.println("r1 = " + r1);

    BigInteger m1 = new BigInteger("20002");
    BigInteger s1 = g(m1);
    System.out.println("s1 = " + s1);
    BigInteger m2 = new BigInteger("-20003");
    BigInteger s2 = g(m2);
    System.out.println("s2 = " + s2);
    }
}
```

Problem 14. To calculate π up to a given number of digits one can use the series expansion

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots$$

Thus for $x = 1$ we have $\arctan(1) = \pi/4$ and therefore

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

Explain why this is not a useful approach to calculate π up to a given number of digits.

Solution 14. The series converges far too slowly. A better expansion can be found by using the *addition theorem*

$$\arctan(x) + \arctan(y) \equiv \arctan\left(\frac{x+y}{1-xy}\right).$$

For $x = 1/2$ and $y = 1/3$ we obtain

$$\frac{\pi}{4} = \arctan\left(\frac{1}{2}\right) + \arctan\left(\frac{1}{3}\right).$$

This series converges much faster. We can iterate this approach to obtain even faster convergence.

Problem 15. Determine the output of the following C++ code.

```
// surprise.cpp

#include <iostream>
using namespace std;

int main(void)
{
   int max = 100;
   int min = 2;

   for(int i=min;i<=max;i++)
   {
   int j = 2;
   while(i%j++ != 0);
   if(j == i+1) cout << "i = " << i << endl;
   }
   return 0;
}
```

Solution 15. The program outputs all prime numbers between 2 and 100.

Problem 16. Given two positive numbers, say a and b. We have to test whether $\ln(a) \leq \ln(b)$. How would we perform this test?

Solution 16. If

$$\ln(a) \leq \ln(b)$$

then $a \leq b$ and vice versa. Thus it is not necessary to calculate the natural logarithm.

Problem 17. Let a and b be real numbers and $b > a$. Let $x \in [a, b]$. Consider the function $f : [a, b] \to \mathbf{R}$

$$f(x) = \frac{x - a}{b - a}.$$

What is the use of this function?

Solution 17. The function normalizes x on the unit interval $[0, 1]$. Thus $f(a) = 0$, $f(b) = 1$ and $f((a + b)/2) = 1/2$.

Problem 18. Given a set of m vectors in $\mathbf{R}^n$

$$\{ \mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{m-1} \}$$

and a vector $\mathbf{y} \in \mathbf{R}^n$. We consider the Euclidean distance, i.e.,

$$\|\mathbf{u} - \mathbf{v}\| := \sqrt{\sum_{j=0}^{n-1} (u_j - v_j)^2}, \qquad \mathbf{u}, \mathbf{v} \in \mathbf{R}^n .$$

We have to find the vector $\mathbf{x}_j$ $(j = 0, 1, \ldots, m-1)$ with the shortest distance to the vector $\mathbf{y}$, i.e., we want to find the index j. Provide an efficient computation. This problem plays a role in neural networks.

Solution 18. First we note that the minimum of a square root is the same as the minimum of a square (both are monotonically increasing functions). Thus,

$$\|\mathbf{x}_j - \mathbf{y}\|^2 = \sum_{i=0}^{n-1} (x_{ji} - y_i)^2 = \sum_{i=0}^{n-1} (x_{ji}^2 - 2x_{ji}y_i + y_i^2) .$$

Since the term

$$\sum_{i=0}^{n-1} y_i^2$$

is a constant it is not necessary to calculate it to find the vector with the shortest distance. Thus we are left to calculate

$$\sum_{i=0}^{n-1} (x_{ji}^2 - 2x_{ji}y_i)$$

for each vector $\mathbf{x}_j$. If all vectors $\mathbf{x}_j$ $(j = 0, 1, 2, \ldots, m-1)$ are normalized (say to 1), then it is also not necessary to calculate x_{ji}^2. Thus we are left with

$$-2 \sum_{i=0}^{n-1} x_{ji}y_i .$$

Obviously, we can also omit the multiplication by -2 and test for the maximum of the sum for the vectors $\{\, \mathbf{x}_j : j = 0, 1, \ldots, m-1 \,\}$.

Problem 19. The following functions

$$\sigma_k(t) = \frac{\sin(\pi n(t - k/n))}{n \sin(\pi(t - k/n))}, \qquad t \in (0, 1)$$

play a central role in *harmonic interpolation*, where n is a positive odd integer and $k = 0, 1, \ldots, n-1$. Let $n = 3$. Can the sum

$$\sum_{k=0}^{n-1} \sigma_k(t)$$

be simplified?

Solution 19. Yes, the sum can be simplified. Using the identities

$$\sin(\alpha-\beta) \equiv \sin(\alpha)\cos(\beta) - \cos(\alpha)\sin(\beta)$$

and

$$\sin(\alpha)\cos(\alpha) \equiv \frac{1}{2}\sin(2\alpha), \qquad \frac{1}{2}\cos(\alpha)\sin(2\alpha) \equiv \frac{1}{2}\sin(\alpha) + \frac{1}{2}\sin(3\alpha)$$

we find

$$\sum_{k=0}^{n-1} \sigma_k(t) = 1\,.$$

This is called a *partition of unity.* This identity does not only hold for $n = 3$ but for any n which is odd.

Problem 20. The series expansion

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots$$

converges for $x \in (-1, 1]$. Thus it allows to calculate

$$\ln 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots$$

This expansion converges very slowly. Is there a faster way to calculate $\ln 2$?

Solution 20. Consider the expansion

$$\ln(1-x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \frac{x^4}{4} - \cdots$$

which converges for $x \in [0, 1)$. Then using $x = 1/2$ we have

$$\ln\left(\frac{1}{2}\right) = -\left(\frac{1}{2} + \frac{1}{8} + \frac{1}{24} + \frac{1}{64} + \cdots\right).$$

This series converges must faster. We have $\ln(2) = -\ln(1/2)$.

We could also subtract the two series expansions and obtain

$$\begin{aligned}\ln(1+x) - \ln(1-x) &= \ln\left(\frac{1+x}{1-x}\right) \\ &= 2\left(x + \frac{x^3}{3} + \frac{x^5}{5} + \cdots\right), \qquad x \in (-1, 1)\,.\end{aligned}$$

This series converges even faster using $x = \pm 1/3$.

Problem 21. Applying Simpon's rule for the evaluation of

$$\int_1^b x^{-2} g(x) dx$$

with the smooth function g constant at large x, would result in a (finite) sum converging very slowly as b becomes large for fixed h (step size) and taking a very long time to compute. Show that changing the variable would improve the situation.

Solution 21. Using the transformation $y = x^{-1}$ with $dy = -x^{-2}dx$ and $x = b \to y = b^{-1}$ yields

$$\int_{b^{-1}}^1 g(y^{-1}) dy$$

which can then be evaluated by using Simpon's rule.

Problem 22. Consider the integral

$$\int_0^1 (1 - x^2)^{-1/2} g(x) dx$$

where g is a smooth function in the interval $[0, 1]$. We have an integrable singularity at $x = 1$ and quadrature formulas give an infinite result. Propose a transformation so that quadrature formulas can be applied.

Solution 22. Using the transformation $y(x) = (1 - x)^{1/2}$ we obtain

$$2 \int_0^1 (2 - y^2)^{-1/2} g(1 - y^2) dy$$

where quadrature formulas can be applied without problems.

Problem 23. Consider the integral

$$I = \int_0^3 \sqrt{x} \cos(x) dx \,.$$

Owing to the term $\sqrt{x}$ the integrand is not regular. Give a transformation that resolves this problem.

Solution 23. We set $x(t) = t^2$. Thus $dx(t) = 2tdt$. We find

$$I = 2 \int_0^{\sqrt{3}} t^2 \cos(t^2) dt \,.$$

Thus the integrand is now an analytical function.

Problem 24. To multiply an $i \times j$ matrix with a $j \times k$ matrix using the standard method it is necessary to do

$$i \times j \times k$$

elementary multiplications. Consider the multiplication of the four matrices A (20×2), B (2×30), C (30×12) and D (12×8). Recall that the matrix product is associative. How many multiplications do we need to do $A(B(CD))$, $(AB)(CD)$, $A((BC)D)$, $((AB)C)D$, $(A(BC))D$? Which one is the optimal order for multiplying these four matrices?

Solution 24. We find

$$\begin{aligned}
A(B(CD)) &\to 30 \cdot 12 \cdot 8 + 2 \cdot 30 \cdot 8 + 20 \cdot 2 \cdot 8 = 3680 \\
(AB)(CD) &\to 20 \cdot 2 \cdot 30 + 30 \cdot 12 \cdot 8 + 20 \cdot 30 \cdot 8 = 8880 \\
A((BC)D) &\to 2 \cdot 30 \cdot 12 + 2 \cdot 12 \cdot 8 + 20 \cdot 2 \cdot 8 = 1232 \\
((AB)C)D &\to 20 \cdot 2 \cdot 30 + 20 \cdot 30 \cdot 12 + 20 \cdot 12 \cdot 8 = 10320 \\
(A(BC))D &\to 2 \cdot 30 \cdot 12 + 20 \cdot 2 \cdot 12 + 20 \cdot 12 \cdot 8 = 3120 .
\end{aligned}$$

Thus the order $A((BC)D)$ is optimal for multiplying the four matrices A, B, C, D.

Problem 25. A semi-discrete Korteweg de-Vries equation is given by

$$\frac{du_n}{dt} = u_n(u_{n+1} - u_{n-1})$$

where $n = 0, 1, 2, \ldots$. Write the equation as a recurrence relation.

Solution 25. We find

$$u_{n+1}(t) = \frac{1}{u_n}\frac{du_n}{dt} + u_{n-1} \equiv \frac{d}{dt}\ln(u_n(t)) + u_{n-1} .$$

Thus u_{n+1} is computed from the knowledge of u_n and u_{n-1}.

Problem 26. Consider the set of two bits $\{0, 1\}$ with the operation of addition modulo 2. This can be written as the table

$\oplus$	0	1
0	0	1
1	1	0

Find a 1-1 map of this set to the set $\{+1, -1\}$ with multiplication as operation such that the algebraic structure is preserved.

Solution 26. We have the map $0 \to +1$ and $1 \to -1$ with the multiplication table

$*$	$+1$	-1
$+1$	$+1$	-1
-1	-1	$+1$

Thus, we have a finite *group* with two elements. The neutral element of the group is $+1$.

Problem 27. What is the following program doing? What is the output?

```
// eps.cpp

#include <iostream>
using namespace std;

int main(void)
{
   double eps = 1.0;
   while((1.0 + eps) > 1.0) { eps = eps/2.0; }
   eps = eps*2.0;
   cout.precision(20);
   cout << "eps = " << eps;

   return 0;
}
```

Solution 27. The program provides the *machine epsilon* for the data type `double`, i.e., the distance from 1.0 to the next largest floating number (data type `double`). We find

$$2.22044605 \cdot 10^{-16} == 2^{-52}\,.$$

(IEEE 754 64-bit conformant).

Problem 28. Write a C++ function `cumsum()` which finds the *cumulative sum* vector of a vector of numbers. For example

$$(2\ 4\ 5\ 1) \to (2\ 6\ 11\ 12)\,.$$

Use templates so that different number data types can be used.

Solution 28. We use the data types `int` and `double`.

```
// cumsum.cpp

#include <iostream>
using namespace std;

template <typename T> void sumcum(T* a,T* c,unsigned long n)
{
  T temp;
  c[0] = a[0];
  for(unsigned long j=1;j<n;j++)
  {
  c[j] = c[j-1] + a[j];
  }
}

int main(void)
{
  unsigned long n = 4;
  int* a1 = new int[n];
  a1[0] = 2; a1[1] = 4; a1[2] = 5; a1[3] = 1;
  int* c1 = new int[n];
  sumcum(a1,c1,n);
  for(unsigned long i=0;i < n;i++)
  {
  cout << "c1[" << i << "] = " << c1[i] << endl;
  }
  delete[] a1;
  delete[] c1;

  unsigned long m = 3;
  double* a2 = new double[m];
  a2[0] = 1.3; a2[1] = 2.7; a2[2] = 1.1;
  double* c2 = new double[m];
  sumcum(a2,c2,m);
  for(unsigned long j=0;j<m;j++)
  {
  cout << "c2[" << j << "] = " << c2[j] << endl;
  }
  delete[] a2;
  delete[] c2;
  return 0;
 }
```

Problem 29. Consider the following two systems of linear equations

$$1.000000x + 1.000000y = 0$$
$$1.000000x + 0.999999y = 1$$

and

$$1.000000x + 1.000000y = 0$$
$$1.000000x + 1.000001y = 1\,.$$

This means we make a change of 0.000002 (only 0.0002 percent) in the coefficient of y in the second equation. Discuss the solutions.

Solution 29. For the first system of linear equations we find

$$x = 10^6, \qquad y = -10^6$$

and for the second system of linear equations we find

$$x = -10^6, \qquad y = 10^6\,.$$

Thus we have an *ill conditioned system*, i.e., a small relative change in one of the coefficient values results in a large relative change in solution values.

Problem 30. *Kahan's summation algorithm* recovers the bits that are lost in the process of adding a small and a large number and preserves this information in the form of an accumulated correction. The following FORTRAN segment implements this summation algorithm given an array of numbers `x(j)`.

```
sum = 0.
carry = 0.
do 100 j = 1,N
  y = carry + x(j)
  t = sum + y
  carry = (sum - t) + y
100 sum = t
    sum = sum + carry
```

The algorithm works because the variable `carry` contains the information that was lost as the result of adding `x(j)` to `sum`. Write a C++ program that implements Kahan's summation algorithm and compare to direct summation. Consider the array

$$\left(1, \frac{1}{2}, \frac{1}{3}, \ldots, \frac{1}{1000}\right)\,.$$

Solution 30. We use the data type `double` for the numbers in the array.

```
// Kahan.cpp

#include <iostream>
using namespace std;

int main(void)
{
  const int n = 1000;
  double x[n];
  for(int i=0;i<n;i++)
  {
  x[i] = 1.0/((double) (i+1));
  }

  double sum1 = 0.0;
  for(int j=0;j<n;j++)
  sum1 += x[j];
  cout.precision(36);
  cout << "sum1 = " << sum1 << endl;

  double sum2 = 0.0;
  double carry = 0.0;
  for(int k=0;k<n;k++)
  {
  double y = carry + x[k];
  double t = sum2 + y;
  carry = (sum2 - t) + y;
  sum2 = t;
  }
  cout.precision(36);
  cout << "carry = " << carry << endl;
  sum2 += carry;
  cout.precision(36);
  cout << "sum2 = " << sum2 << endl;

  return 0;
}
```

Problem 31. Euler noticed that $x^2 + x + 41$ takes on prime values for $x = 0, 1, 2, \ldots, 39$. Thus we may ask whether it is possible to have a polynomial which produces only prime values. It can be shown that this is

not the case unless the polynomial is constant. Write a C++ program that checks that $x^2 + x + 41$ are prime numbers for $x = 0, 1, 2, \ldots, 39$. Extend the loop to numbers greater than 39 to see which numbers are prime and not prime beyond 39.

Solution 31. We find that for 40 and 41 the numbers are not prime, but for 42 the number is prime again. The primality testing can be improved by only considering potential factors less than $\sqrt{x}$, or applying the sieve of Eratosthenes.

```
// Euler.cpp

#include <iostream>
using namespace std;

int euler(int x) { return x*(x+1) + 41; }

int isprime(int x)
{
  if(x%2 == 0) return 0;
  for(int j=3;j<x/2;j+=2)
    if(x%j == 0) return 0;
  return 1;
}

int main(void)
{
  for(int x=0;x<=50;x++)
  {
  cout << "x = " << x << ","
       << "x^2 + x + 41 = " << euler(x);
  if(isprime(euler(x)))
    cout << " is prime";
  else
    cout << " is not prime";
  cout << endl;
  }
  return 0;
}
```

Problem 32. Given a vector $\mathbf{x} \in \mathbf{R}^n$. We want to find the *1-norm*

$$\|\mathbf{x}\| := \sum_{j=0}^{n-1} |x_j|$$

and the ∞-*norm*

$$\|\mathbf{x}\| := \max_{0 \le j < n} |x_j|$$

of this vector. Write a C++ program that calculates both norms using one `for` loop.

Solution 32. At the beginning of the iteration we set the 1-norm and the ∞-norm to `fabs(x[0])`.

```
// norms.cpp

#include <iostream>
#include <cmath>
using namespace std;

void norms(double* x,int n,double& norm1,double& norminf)
{
  norm1 = fabs(x[0]);
  norminf = norm1;
  for(int i=1;i<n;i++)
  {
  x[i] = fabs(x[i]);
  norm1 += x[i];
  if(x[i] > norminf) norminf = x[i];
  }
}

int main(void)
{
  int n = 5;
  double* x = new double[n];
  x[0] = -5.1; x[1] = 2.3; x[2] = 3.7; x[3] = 1.1; x[4] = 0.7;
  double norm1;
  double norminf;
  norms(x,n,norm1,norminf);
  cout << "norm1 = " << norm1 << endl;
  cout << "norminf = " << norminf << endl;
  delete[] x;
  return 0;
}
```

Problem 33. Given an array $\mathbf{x}$ of n numbers

$$\mathbf{x} = (x_0, x_1, x_2, \ldots, x_{n-1}) .$$

From this array form a new array **y** with $n-1$ elements as follows (first differences)

$$\mathbf{y} := (y_0 = x_1 - x_0, y_1 = x_2 - x_1, \ldots, y_{n-2} = x_{n-1} - x_{n-2}) .$$

From this array we form again a new array with $n-2$ elements (second differences)

$$\mathbf{z} := (z_0 = y_1 - y_0, z_1 = y_2 - y_1, \ldots, z_{n-3} = y_{n-2} - y_{n-3})$$

and so on, till we obtain the array with one element. Write a C++ program that finds all these arrays.

Solution 33. We store the one-dimensional arrays in a two-dimensional *jagged array A*.

```
// jaggedarray.cpp

#include <iostream>
#include <iomanip>
using namespace std;

int main(void)
{
   double x[] = { 3.0, 7.0, 11.0, 13.0, 17.0, 19.0, 23.0 };
   int n = sizeof(x)/sizeof(double); // length of array

   double** A = NULL;
   A = new double*[n];
   A[0] = new double[n];

   int i, j;
   for(j=0;j<n;j++)
   A[0][j] = x[j];

   for(i=1;i<n;i++)
   {
   A[i] = new double[n-i];
   for(j=0;j<n-i;j++)
   A[i][j] = A[i-1][j+1] - A[i-1][j];
   }

   // display output
   for(i=0;i<n;i++)
   {
```

```
    for(j=0;j<n-i;j++)
    cout << setw(4) << A[i][j] << "  ";
    cout << endl;
    }
    for(i=0;i<n;i++)
    delete[] A[i];
    delete[] A;
    return 0;
}
```

Problem 34. The *exponential function* e^x can be defined as

$$e^x := \sum_{j=0}^{\infty} \frac{x^j}{j!} \tag{1}$$

or

$$e^x := \lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n . \tag{2}$$

Thus for $x = 1$ we can calculate e. Using the second definition and a given n we can find an approximation for e. The following C++ program implements this approximation.

```
// exp.cpp

#include <iostream>
#include <cmath>
using namespace std;

int main(void)
{
  double x = 1.0;
  double n = 1e308; // 1e309 constant too big
  double d = x/n;
  cout << "d = " << d << endl;
  double e = pow(1.0+d,n);
  cout << "e = " << e;
  return 0;
}
```

What is the output of this program?

Solution 34. The output is surprisingly 1 and not 2.71828... as we expected. Explain why?

Problem 35. In holography we have to calculate the phase difference

$$\delta := z_L - z_0 + \sqrt{z_0^2 + x^2} - \sqrt{z_L^2 + x^2}$$

where $x \ll z_0$, $x \ll z_L$. Can the calculation be simplified using these conditions?

Solution 35. Since

$$\sqrt{1+a} \approx 1 + \frac{1}{2}a$$

for $|a| \ll 1$ and

$$\sqrt{z_0^2 + x^2} = z_0\sqrt{1 + x^2/z_0^2}, \qquad \sqrt{z_L^2 + x^2} = z_L\sqrt{1 + x^2/z_L^2}$$

we obtain

$$\delta = \frac{x^2}{2}\left(\frac{1}{z_0} - \frac{1}{z_L}\right).$$

Thus, we avoid the calculation of two square roots.

Problem 36. (i) Consider the vectors $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$ in $\mathbf{R}^3$ and the expression

$$\mathbf{x} \times (\mathbf{y} \times \mathbf{z}) + \mathbf{z} \times (\mathbf{x} \times \mathbf{y})$$

where $\times$ denotes the *vector product*. How can this expression be simplified?
(ii) Let A, B, C be $n \times n$ matrices over $\mathbf{C}$. Let $[A, B] := AB - BA$ be the *commutator*. How can the expression

$$[A, [B, C]] + [C, [A, B]]$$

be simplified?

Solution 36. (i) We apply the *Jacobi identity*

$$\mathbf{x} \times (\mathbf{y} \times \mathbf{z}) + \mathbf{z} \times (\mathbf{x} \times \mathbf{y}) + \mathbf{y} \times (\mathbf{z} \times \mathbf{x}) = \mathbf{0}.$$

Thus we only have to calculate $-\mathbf{y} \times (\mathbf{z} \times \mathbf{x})$.
(ii) We also apply the Jacobi identity

$$[A, [B, C]] + [C, [A, B]] + [B, [C, A]] = 0.$$

Thus, we only have to calculate $-[B, [C, A]]$.

Chapter 2

Bitwise Operations

Problem 1. What is the output of the following C++ code?

```
// twocomp.cpp

#include <iostream>
using namespace std;

int main(void)
{
  int x = 15;
  int y = ~x;   // NOT operation (one complement)
  cout << "y = " << y << endl;
  int z = ++y;  // adding 1
  cout << "z = " << z << endl;
  return 0;
}
```

Solution 1. The binary representation of 15 is

0000 0000 0000 0000 0000 0000 0000 1111

We first apply the NOT operation to x, i.e., the *one's complement*

1111 1111 1111 1111 1111 1111 1111 0000

This yields -16. Adding 1 to the least significant bit provides -15. Thus, the two operations provide the *two's complement.*

Problem 2. What is the output of the following Java code?

```
// ToLower.java

public class ToLower
{
   public static void main(String[] args)
   {
   char c = 'Z';
   int ic = (int) c;  // type conversion ASCII table
   ic = ic | 32;      // bitwise inclusive OR
   c = (char) ic;     // type conversion ASCII table
   System.out.println("c = " + c);

   char d = 'x';
   int id = (int) d;  // type conversion ASCII table
   id = id ^ 32;      // bitwise exclusive OR
   d = (char) id;     // type conversion ASCII table
   System.out.println("d = " + d);
   } // end main
}
```

where | is the bitwise inclusive OR-operation, i.e.,

```
0 | 0 = 0,   0 | 1 = 1,   1 | 0 = 1,   1 | 1 = 1
```

and ^ is the bitwise exclusive OR-operation, i.e.,

```
0 ^ 0 = 0,   0 ^ 1 = 1,   1 ^ 0 = 1,   1 ^ 1 = 0.
```

Solution 2. In the ASCII table the capital letter Z corresponds to 90 (base 10). In Java the data type char is of size 2 byte. Note that in C++ the data type char is of size 1 byte. The binary representation of 90 is (1 byte)

```
0101 1010.
```

We perform the bitwise OR-operation on the bit position 5 numbered from 0 from the right, since $2^5 = 32$ (counting from right to left starting from 0). Thus, we obtain the small z. The ASCII value for z is 122 (= 90 + 32). Thus, the first part of the program converts capital letters to small letters. In the second part of the program we convert small letter into capital letters using the bitwise XOR operation. The ASCII value for x is 120 (base 10). The binary representation of 120 is (1 byte) 11111000.

Problem 3. (i) Write down the *function table* (*truth table*) for the two's complement. The number of inputs is four bits. The number of outputs is five bits. Four outputs are for the two's complement and the fifth indicates whether there was a carry in the process.
(ii) Find the boolean functions for the outputs.

Solution 3. (i) The two-complement is constructed taking the one-complement ($0 \to 1, 1 \to 0$) and then adding 1 to the least significant bit, where $1 + 1 = 0$ carry 1. Thus we have the truth table

Inputs				Outputs				Carry
0	0	0	0	0	0	0	0	1
0	0	0	1	1	1	1	1	0
0	0	1	0	1	1	1	0	0
0	0	1	1	1	1	0	1	0
0	1	0	0	1	1	0	0	0
0	1	0	1	1	0	1	1	0
0	1	1	0	1	0	1	0	0
0	1	1	1	1	0	0	1	0
1	0	0	0	1	0	0	0	0
1	0	0	1	0	1	1	1	0
1	0	1	0	0	1	1	0	0
1	0	1	1	0	1	0	1	0
1	1	0	0	0	1	0	0	0
1	1	0	1	0	0	1	1	0
1	1	1	0	0	0	1	0	0
1	1	1	1	0	0	0	1	0

(ii) We label the least significant bit as the 0 bit (from the right). Obviously, we use

$$\overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2} \cdot \overline{I_3}$$

for the carry. For the least significant bit we use I_0. Then in increasing significance we use

$$I_0 \oplus I_1 = \overline{I_1} \oplus \overline{I_0},$$

$$\overline{I_2} \oplus \overline{I_0} \cdot \overline{I_1}$$

and

$$\overline{I_3} \oplus \overline{I_0} \cdot \overline{I_1} \cdot \overline{I_2}$$

where $\oplus$ is the XOR operation, $\cdot$ the AND operation and $^-$ is the NOT operation.

Problem 4. The truth table for the *NAND-gate* is given by

I_1	I_2	O
0	0	1
0	1	1
1	0	1
1	1	0

It is a *universal gate*, i.e., all other gates can be built from this gate. Show that the XOR-gate can be built from this gate.

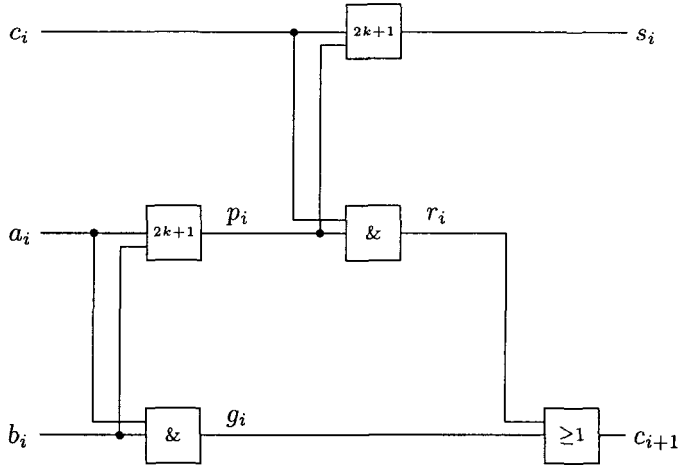
Solution 4. We need four NAND-gates to build the XOR-gate. Let a, b be the input, i.e.,

$$a, b \in \{0, 1\}.$$

Then the XOR-gate can be expressed as

```
XOR(a,b) = NAND4(NAND2(a,NAND1(a,b)),NAND3(NAND1(a,b),b)).
```

Problem 5. Consider the following circuit.



Find the truth table for s_i and c_{i+1}. What does this circuit do?

Solution 5. We have the truth table.

a_i	b_i	c_i	p_i	g_i	r_i	s_i	c_{i+1}
0	0	0	0	0	0	0	0
0	1	0	1	0	0	1	0
1	0	0	1	0	0	1	0
1	1	0	0	1	0	0	1
0	0	1	0	0	0	1	0
0	1	1	1	0	1	0	1
1	0	1	1	0	1	0	1
1	1	1	0	1	0	1	1

The circuit is a *full adder*. The output s_i is the ith bit of the sum and c_{i+1} is the carry bit.

Problem 6. Write a C++ program that counts the set bits in a given computer word. For the program use `unsigned long`. For example, if $k = 15$ the program returns 4.

Solution 6. The function `bitcount()` counts the number of bits set in an `unsigned long`. The operation `k &= (k-1)` clears the lowest order bit of `k` which is set.

```
// bitcount.cpp

#include <iostream>
using namespace std;

unsigned long bitcount(unsigned long k)
{
  unsigned long r = 0;
  while(k != 0)
  {
  r++;
  k &= (k-1);
  }
  return r;
}

int main(void)
{
  unsigned long k1 = 3;
  unsigned long r1 = bitcount(k1);
  cout << "r1 = " << r1 << endl;
  unsigned long k2 = 4;
  unsigned long r2 = bitcount(k2);
```

```
  cout << "r2 = " << r2 << endl;
  unsigned long k3 = 16;
  unsigned long r3 = bitcount(k3);
  cout << "r3 = " << r3 << endl;
  unsigned long k4 = 255;
  unsigned long r4 = bitcount(k4);
  cout << "r4 = " << r4 << endl;
  return 0;
}
```

Problem 7. What is the output of the following C++ program? Note that ^ indicates the bitwise XOR operation in C++.

```
// func.cpp

#include <iostream>
using namespace std;

void func(int& x,int& y) { x = x^y;  y = x^y;  x = x^y; }

int main(void)
{
  int x = -14, y = 17;
  func(x,y);
  cout << "x = " << x << endl; // =>
  cout << "y = " << y << endl; // =>

  x = -23; y = -45;
  func(x,y);
  cout << "x = " << x << endl; // =>
  cout << "y = " << y << endl; // =>
  return 0;
}
```

Solution 7. We pass x and y by reference. Since ^ is the bitwise XOR we swap the values of x and y. Thus the output is `x = 17, y = -14` in the first case and $x = -45$, $y = -23$ in the second.

Problem 8. There are two ways to perform binary *division*, either by repeated subtraction or using a *shift-and-subtract principle*. The latter is used in practice as it is much faster.

Division by repeated subtraction is performed by subtracting the divisor from the dividend until the result of the subtraction is negative. The resultant quotient is given by the number of subtractions required minus 1. The remainder is obtained by adding the divisor to the negative result.

The *shift-and-subtract method* of division is performed by succesively subtracting the divisor from the appropriate shifted dividend and inspecting the sign of the remainder after each subtraction. If the sign of the remainder is positive, then the value of the quotient is 1, but if the sign of the remainder is negative, then the value is 0 and the dividend is restored to its previous value by adding the divisor. The divisor is then shifted one place to the right, and the next significant bit of the dividend is included and the operation repeated until all bits in the dividend have been used. To simplify the method further, instead of adding the divisor when the subtraction yields a negative result, we can add the divisor shifted right by one position.

For example, consider the division of 90 by 9 viewed as 8 bit numbers. 90 is given by 01011010 and 9 is given by 00001001 in binary representation. Then

```
        0|1011010
-00001001
 111101111|011010          -> negative -> 0
+ 00001001
  111110000|11010          -> negative -> 0
+  00001001
   111110011|1010          -> negative -> 0
+   00001001
    111111001|010          -> negative -> 0
+    00001001
     000000100|10          -> positive -> 1
-     00001001
      111110111|0          -> negative -> 0
+      00001001
       000000000|          -> positive -> 1
-       00001001
        11110111           -> negative -> 0
+       00001001
        00000000           Remainder
```

The least significant bit is computed last. Thus the answer is 00001010. Write a C++ program which implements this algorithm.

Solution 8. The function `division()` performs the division as specified in the question.

```
// division.cpp

#include <iostream>
using namespace std;

int division(int i,int d)
{
  int q = 0;
  i -= (d << 7);
  for(int j=6;j>=0;j--)
  if(i >= 0)
    { q += (1 << (j+1)); i -= (d << j); }
    else i += (d << j);
  return q;
}

int main(void)
{
  cout << division(90,9) << endl;
  return 0;
}
```

The output is 10.

Problem 9. In the `Graphics` class of Java the method

```
public abstract void setXORMode(Color c1)
```

sets the paint mode of this `Graphics` context to alternate between this `Graphics` context's current color and the new specified color.
(i) Discuss the connection of this method with the bitwise XOR operation.
(ii) Write a Java program that uses this method and discuss the program.

Solution 9. (i) This specifies that logical pixel operations are performed in the XOR mode, which alternates pixels between the current color and a specified XOR color. When drawing operations are performed, pixels which are the current color are changed to the specified color, and vice versa.
In other words, we XOR the color c of the pixel being written with $current \oplus c1$, i.e.,

$$current = (current \oplus c1) \oplus c1,$$
$$c1 = (current \oplus c1) \oplus current.$$

(ii) The Java program is given by

```
// XOR.java

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class XOR extends Applet
{
   private Color strawberry = new Color(0xcc,0,0x66);
   private Color chocolate = new Color(0x66,0x33,0);
   private Color vanilla = new Color(0xff,0xff,0x99);
   private Color scoop = new Color(0x33,0x99,0xcc);
   private static final int BORDER = 10;

   public void init()
   {
   setBackground(chocolate);

   addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
         Graphics g = getGraphics();
         g.setColor(chocolate);
         g.setXORMode(scoop);
         Dimension dim = getSize();
         int diameter = dim.height - (2*BORDER);
         int xStart = (dim.width/2) - (diameter/2);
         int yStart = BORDER;
         g.fillOval(xStart,yStart,diameter,diameter);
         g.setPaintMode();
         }
         }
         );   // end addMouseListener
    }  // end init

    public void paint(Graphics g)
    {
    Dimension dim = getSize();
    int width = dim.width/3; int height = dim.height;
    g.setColor(strawberry);
    g.fillRect(0,0,width,height);
    g.setColor(vanilla);
    g.fillRect(dim.width-width,0,width,height);
    }  // end paint
}  // end class XOR
```

Problem 10. Consider two unsigned integers m and n. Multiplication can be done using shift operations as follows (the *Russian farmer multiplication*). At each step m is divided by 2 (shift right by 1) and n is multiplied by 2 (shift left by 1). For example, let $m = 35$ and $n = 40$

m	n
35	40
17	80
8	~~160~~
4	~~320~~
2	~~640~~
1	1280
	1400

All the numbers on the right-hand side are deleted if the number on the left-hand side is even. The remaining numbers are added. This is the result of the multiplication. Write a C++ program which implements the Russian farmer multiplication using the shift operations. Testing whether the number m is even or odd must also be done with a bitwise operation. The data type of m and n should be `unsigned long`.

Solution 10.

```
// russian.cpp

#include <iostream>
using namespace std;

unsigned long multiply(unsigned long m,unsigned long n)
{
  if(m == 0) return 0; if(n == 0) return 0;
  if(m == 1) return n; if(n == 1) return m;
  unsigned long temp = 0;
  while(m != 0)
  {
  if((m & 1) == 1)
  temp += n;
  m = m >> 1; n = n << 1;
  }
  return temp;
}

int main(void)
{
  unsigned long m = 35, n = 40;
```

```
  unsigned long result = multiply(m,n);
  cout << "result = " << result << endl;
  return 0;
}
```

Problem 11. A *checksum* can be used to determine if errors occurred in transmission of data. A simple 32-bit checksum algorithm, simply adds all data in the transmission as if it were a series of 32-bit quantities, i.e. we read 4 bytes each and add the resulting 32-bit quantities. When the checksum overflows, the overflow bit is added to the checksum.
(i) Find the checksum for the string `ABCD`.
(ii) Write C++ code to compute the checksum of some data, and provide options for comparing the checksum to a given value.
(iii) What errors can this 32-bit checksum detect?

Solution 11. (i) For the string `ABCD` we have

$$65 + 66 \cdot 256 + 67 \cdot 256^2 + 68 \cdot 256^3 = 1145258561$$

where we used the ASCII table, where `A -> 65`, `B -> 66`, `C -> 67` and `D -> 68`.
(ii) The function `checksum()` computes the checksum for an arbitrary input stream.

```
// checksum.cpp

#include <fstream>
#include <iostream>
using namespace std;

unsigned int checksum(istream &in)
{
  unsigned int sum = 0;
  unsigned int value;
  while(!(in.eof())) {
    value = 0; // for 0 padding
    in.read((char*)& value,sizeof(unsigned int));
    if(!in.eof()) {
      if(sum+value<sum) { // overflow
        sum += value;
        sum++;
        } else { sum += value; }
        }
    }
```

```
    return sum;
}

int main(int argc,char *argv[])
{
  ifstream file;
  if(argc < 2) {
     cout << "stdin: " << checksum(cin) << endl;
     } else {
        for(int i=1;i<argc;i++) {
          file.open(argv[i],ios::in | ios.binary);
          cout << argv[i] << ": " << checksum(file)
               << endl;
               file.clear();
               file.close();
               }
  }
  return 0;
}
```

(iii) If we look at the checksum it is clear that any difference in two sequences of 32-bit values that is not a multiple of 2^{32} in total will result in a different checksum. This allows us to detect all errors that do not produce a difference in the sum of precisely $2^{32} = 4294967296$. The checksum in this example is further improved by the addition of the carry bit. The addition of the carry bit will prevent multiples of 4294967296 from being detected as a no error condition. So now all errors must result in a net increase of zero in the sum if they are to go undetected, i.e., some values must decrease and others increase to get a net result of zero difference over the sum.

Problem 12. Let $\{ x_j \}_{j=0}^{N-1}$ be a sequence of N bits. A measure of the correlation between bits at distance n is given by

$$\Gamma(n) := \frac{1}{N} \sum_{j=0}^{j=N-1} x_j \oplus x_{(j+n) \bmod N}$$

where $\oplus$ is the XOR operation. Implement $\Gamma(n)$ in C++ using the standard template library with the `bitset` class. The class `bitset<size_t N>` is a class that describes objects that can store a sequence of a fixed number of bits, `N`. The constructor `bitset(unsigned long val)` constructs an object of class `bitset<N>`, initializing the first M bit values to the corresponding bits in `val`. The template parameter `N` of `bitset<N>` is of type `size_t`. Apply it to different bitstrings.

Solution 12. To implement Γ for a `bitset` of any length we need to use a template function.

```
// bitcorrelation.cpp

#include <iostream>
#include <bitset>
using namespace std;

template<size_t N> double gamma(int n,bitset<N> b)
{
  double v = 0.0;

  // make sure to use positive n
  while(n < 0) n += N;

  for(int j=0;j<N;j++) v += b[j]^b[(j+n)%N];
  v /= N;
  return v;
}

int main(void)
{
  bitset<8>  b1(string("10010010"));
  bitset<16> b2(string("0110100101101001"));

  cout << b1 << endl
       << "\tGamma(2) = " << gamma(2,b1) << endl
       << "\tGamma(4) = " << gamma(4,b1) << endl;
  cout << b2 << endl
       << "\tGamma(2) = " << gamma(2,b2) << endl
       << "\tGamma(3) = " << gamma(3,b2) << endl
       << "\tGamma(4) = " << gamma(4,b2) << endl;
  return 0;
}
```

Problem 13. Cyclic redundancy checks provide a better means of checking data for errors. The *CRC32 polynomial*

$$G(x) = x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$$

is a commonly used polynomial for error detection. This polynomial is represented as a bit string where each coefficient (1 or 0) is a binary digit in the string. To check for errors in transmitted data, we do the following:

- Compute the CRC of the data to be sent
- Send the data and CRC
- Recompute the CRC and check whether it matches the sent CRC

To compute the CRC we do the following: we let $B(x)$ be the data to be sent plus 32 0's appended (i.e., multiply by x^{32}). Now we calculate $R(x) = B(x) \bmod G(x)$, and set $T(x) = B(x) - R(x)$. This sets the low bits of T to R. We transmit T to get T'. If $T'(x) \bmod G(x) = 0$ then there was no error. We can perform the remainder calculation using a feedback shift register:

- Create a register with 32 bits, all set to zero.
- For each $G(x)$ where the coefficient is 1, build an XOR gate, so the output is xorred with the bit that is shifted into that register.

The data bits are shifted in one at a time. All the bits shift one position on, and if there is an xor gate then XOR the position with the value in the last register. Write a C++ program to calculate the CRC of a file using the CRC polynomial.

Solution 13.

```
// CRC32.cpp

#include <bitset>
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;

int poly[] = { 0,1,2,4,5,7,8,10,11,12,16,22,23,26,-1 };

const unsigned long n = 32;
bitset<n> result;

bitset<n> crc32(bitset<8> data)
{
  bool bit;
  int xorpos, pos;

  for(int i=0;i<8;i++) {
     bit = result[31];
     result <<= 1;
```

```
      result.set(0,data[i]);
      xorpos = 0;
      while(poly[xorpos]!=-1)
      {
      pos = poly[xorpos];
      result.set(pos,bit^result[pos]);
      xorpos++;
      }
      }
  return result;
}

void calc_crc(istream& in)
{
  unsigned char c;
  bitset<n> crc;
  result.reset();
    while(!in.eof()) {
      in.read((char*) &c,1);
      if(!in.eof()) {
      bitset<8> data(c);
      crc = crc32(data);
      }
    }
  cout << crc << "(" << setbase(16) << crc.to_ulong() << ")"
       << endl;
}

int main(int argc,char *argv[])
{
   ifstream fin;
   if(argc > 1) {
     for(int i=1;i<argc;i++) {
        fin.open(argv[i],ios::in|ios::binary);
        cout << argv[i] << ": ";
        calc_crc(fin);
        fin.close();
        }
        } else { calc_crc(cin); }
   return 0;
}
```

Problem 14. A *flip-flop circuit*, especially a J-K flip-flop that can memorize a single bit of information, has been of great use in memory modules of computer hardware. The next state $Q(t+1)$ of a J-K flip-flop is characterized as a function of both the present state $Q(t)$ and the present two inputs $J(t)$ and $K(t)$. The truth table of the J-K flip-flop is given by

J(t)	K(t)	Q(t)	Q(t+1)	
0	0	0	0	no change
0	0	1	1	no change
0	1	0	0	reset
0	1	1	0	reset
1	0	0	1	set
1	0	1	1	set
1	1	0	1	toggle
1	1	1	0	toggle

(i) A *sum of products* is a bitwise OR of product forms. It is also called *minterm*. Find the sum of products expression of $Q(t+1)$, where + denotes the OR operation and $^-$ denotes the NOT operation.
(ii) Simplify the minterm expression.

Solution 14. (i) We have four rows in the truth table where $Q(t+1)$ is 1. These rows are ORed together. Thus we find that the minterm expression of $Q(t+1)$ is

$$Q(t+1) = \bar{J}(t)\bar{K}(t)Q(t) + J(t)\bar{K}(t)\bar{Q}(t) + J(t)\bar{K}(t)Q(t) + J(t)K(t)\bar{Q}(t)$$

where $J(t)K(t)$ denotes the AND operation of $J(t)$ and $K(t)$.
(ii) The simplification yields

$$Q(t+1) = J(t)\bar{Q}(t) + \bar{K}(t)Q(t)\,.$$

Problem 15. Given a boolean function $f : \{0,1\}^n \to \{0,1\}$ with the operations AND, OR, XOR. Write a C++ program that calculates f for all 2^n combinations. As an example consider

$$f(x_0, x_1, x_2, x_3) = x_3 \oplus x_2 \oplus x_1 \oplus x_0$$

where $\oplus$ denotes the XOR operation. Note that the XOR operation is associative.

Solution 15. The results of the calculations of the boolean function f are stored in the array `result[]`. The different bits for each combination are stored in the array `x[]`.

```
// boolfunction.cpp

#include <iostream>
#include <cmath>
using namespace std;

int f(int* x) { return (x[0] ^ x[1] ^ x[2] ^ x[3]); }

int main(void)
{
  int n = 4;          // number of bits
  int p = pow(2,n);   // number of possible bitsrings 2^n
  int* x = new int[n];
  int* result = new int[p];

  for(int j=0;j<p;j++)
  {
  int t = j;
  for(int k=0;k<n;k++)
  {
  x[k] = t & 1;    // bitwise AND
  t = t >> 1;      // shift for integer division by 2
  }
  result[j] = f(x);
  }

  for(int l=0;l<p;l++)
  cout << "result[" << l << "] = " << result[l] << endl;

  delete[] x; delete[] result;
  return 0;
}
```

Problem 16. Depending on which computing system we use, we will have to consider the byte order in which multibyte numbers are stored, particularly when we are writing those numbers to a file. The two orders are called *Little Endian* and *Big Endian.*

Little Endian means that the low-order byte of the number is stored in memory at the lowest address, and the high-order byte at the highest address. Big Endian means that the high-order byte of the number is stored in memory at the lowest address, and the low-order byte at the highest address.

Consider a 4 byte integer where b0 is the low-order byte and b3 is the high-order byte. Let i denote the integer and bi the byte array of i. On Little Endian systems we have

```
bi[0] == b0
bi[1] == b1
bi[2] == b2
bi[3] == b3
```

and on Big Endian systems we have

```
bi[0] == b3
bi[1] == b2
bi[2] == b1
bi[3] == b0
```

Write a C++ program that converts between Little Endian and Big Endian.

Solution 16.

```
// convert.cpp

#include <iostream>
using namespace std;

long convert(long i)
{
  long temp = 0;
  temp |= (i & 0xFF000000) >> 24;
  temp |= (i & 0x00FF0000) >> 8;
  temp |= (i & 0x0000FF00) << 8;
  temp |= (i & 0x000000FF) << 24;
  return temp;
}

int main(void)
{
  long i = 293;
  cout << "(Little Endian) i = " << i << endl;;
  cout << "(Big    Endian) i = " << convert(i) << endl;
  cout << "(Little Endian) i = " << convert(convert(i));
  return 0;
}
```

Problem 17. Let X and Y be binary n-tuples, for example

```
X = "01100100",  Y = "11100111" .
```

The *Hamming distance* between X and Y, $H(Y,X)$ is the number of components in which X and Y differ. For the present example it would be 3.
The Hamming distance is a metric, i.e.,
(a) $H(X,Y) \geq 0$
(b) $H(X,Y) = 0$ if and only if $X = Y$.
(c) $H(X,Y) = H(Y,X)$
(d) $H(X,Z) \leq H(X,Y) + H(Y,Z)$

The standard template library in C++ provides a class `bitset` to handle bit operations. Write a C++ program using the `bitset` class that finds the Hamming distance between two binary n-tuples..

Solution 17. The function `hamming()` uses the function `count()` of the `bitset` class. The function `count()` counts the number of `1's` in the bitset.

```
 // hamming.cpp

#include <iostream>
#include <string>
#include <bitset>
using namespace std;

template <size_t n>
int hamming(const bitset<n>& b1,const bitset<n>& b2)
{ return (b1^b2).count(); }

int main(void)
{
  bitset<4> a(string("0110"));
  bitset<4> b(string("1010"));
  cout << "H(" << a << "," << b << ") = " << hamming(a,b);
  cout << endl;
  bitset<8> x(string("01100100"));
  bitset<8> y(string("11100111"));
  cout << "H(" << x << "," << y << ") = " << hamming(x,y);
  cout << endl;
  return 0;
}
```

Problem 18. Computers store data of all kinds as sequences of 0's and 1's, or bits (from binary digit). We might think that these bits are stored verbatim on a disk drive. However, practical considerations call for more subtle schemes, and these lead to shifts of finite type. There are two schemes to transform an arbitrary sequence of bits into one obeying constraints that control intersymbol interference and clock drift.

(i) The first method is called *frequency modulation.* This scheme controls clock drift with the insertion of a clock bit `1` between each pair of data bit (`0` or `1`). If the original data is

```
1 0 0 0 1 1 0 1
```

then the frequency modulation coded data is

```
1 1 1 0 1 0 1 0 1 1 1 1 1 0 1 1
- - - - - - - - -
```

The clock bits are underlined here. They are stored identically to data bits. The original data is recovered by ignoring the clock bits. Given the original data, write a C++ program using the `bitset` class that implements the frequency modulation and the recovering of the original data.

(ii) The modified frequency modulation scheme inserts a `0` between each pair of data bits unless both data bits are `0`, in which case it inserts a `1`. For example, if the original data are

```
1 0 0 0 1 1 0 1
```

the modified frequency modulation coded sequence is

```
0 1 0 0 1 0 1 0 0 1 0 1 0 0 0 1
- - - - - - - - -
```

Here the initial data bit is `1`, so the bit inserted to its left must be `0`. If the initial data bit were `0`, we would need to know the previous data bit before we could decide which bit to insert to its left. Given the original data write a C++ program using the `bitset` class that implements the modified frequency modulation and the recovering of the original data.

Solution 18. Note that `bitset<n>` displays right to left.

```
// modulation.cpp

#include <iostream>
#include <bitset>
using namespace std;

template <size_t n>
```

```
bitset<2*n> freqmod(const bitset<n>& b)
{
  bitset<2*n> b2;
  for(int i=0;i<n;i++)
  {
  b2[2*i] = b[i];
  b2[2*i+1] = 1;
  }
  return b2;
}

template <size_t n>
bitset<2*n> m_freqmod(const bitset<n>& b)
{
  bitset<2*n> b2;
  for(int i=0;i<n;i++)
  {
  b2[2*i] = b[i];
  if(i != (n-1) && b[i] == 0 && b[i+1] == 0) b2[2*i+1] = 1;
  else b2[2*i+1] = 0;
  }
  return b2;
}

int main(void)
{
  bitset<8> b(string("10001101"));
  cout << b << endl;
  cout << freqmod(b) << endl;
  for(int i=0;i<8;i++) cout << "- ";
  cout << endl;
  cout << b << endl;
  cout << m_freqmod(b) << endl;
  for(int j=0;j<8;j++) cout << "- ";
  cout << endl;
  return 0;
}
```

Problem 19. Write a C++ program that, given an `unsigned long`, finds the next `unsigned long` with the same number of bits set. For example, 4 is in binary 100. Then the next number would be 8 since in binary we have 1000.

Solution 19. The function `next()` in the C++ program finds the lowest order bit in `i` which is set and clears it. The next lowest order bit, which is cleared, is then set.

```
// samenoofbits.cpp

#include <iostream>
using namespace std;

unsigned long next(unsigned long i)
{
  unsigned long bit = 1, count = -1;
  if(i == 0) return 0;
  while(!(bit & i)) { bit <<= 1; } // find first one bit
  while(bit & i) { count++; bit <<= 1; } // find next zero bit
  if(!bit) { cout << "overflow in next"; exit(0); }
  i &= (~(bit - 1)); // clear lower bit
  i |= bit | ((1 << count) - 1);
  return i;
}

int main(void)
{
  unsigned long i1 = 0;
  unsigned long r1 = next(i1);
  cout << "r1 = " << r1 << endl;
  unsigned long i2 = 1;
  unsigned long r2 = next(i2);
  cout << "r2 = " << r2 << endl;
  unsigned long i3 = 3;
  unsigned long r3 = next(i3);
  cout << "r3 = " << r3 << endl;
  unsigned long i4 = 7;
  unsigned long r4 = next(i4);
  cout << "r4 = " << r4 << endl;
  return 0;
}
```

Problem 20. Let

`a = a(n-1)a(n-2)...a(1)a(0),` `b = b(n-1)b(n-2)...b(1)b(0)`

be two n-bit binary numbers. We want to compute the sum

`a + b = s = s(n)s(n-1)...s(1)s(0)` .

The standard algorithm is to add from right to left, propagating a carry-bit `c(i)` from bit to bit. In the following algorithm, `0` or `F` means false, `1` or `T` means true, and `OR`, `XOR`, and `AND` are logical operations

```
c(-1) = 0
for i=0 to n-1
  c(i) = ((a(i) XOR b(i)) AND c(i-1)) OR (a(i) AND b(i))
  s(i) = a(i) XOR b(i) XOR c(i-1)
endfor
s(n) = c(n-1).
```

The challenge is to propagate the carry bit `c(i)` from right to left more quickly. Then all the sum bits `s(i)` can be computed in a single time step.

Solution 20. Let `p(i) = (a(i) XOR b(i))` be the propagate bit and `g(i) = (a(i) AND b(i))` the generate bit. Thus we can write

```
c(i) = (p(i) AND c(i-1)) OR g(i).
```

We evaluate this recurrence using parallel prefix, where the associative operation is a 2-by-2 Boolean matrix multiplication

$$\begin{aligned}\begin{pmatrix} c(i) \\ T \end{pmatrix} &= \begin{pmatrix} p(i) \text{ AND } c(i-1) \text{ OR } g(i) \\ T \end{pmatrix} \\ &= \begin{pmatrix} p(i) & g(i) \\ F & T \end{pmatrix} \begin{pmatrix} c(i-1) \\ T \end{pmatrix} \\ &= C(i) \begin{pmatrix} c(i-1) \\ T \end{pmatrix} \\ &= C(i) * C(i-1) * \cdots * C(1) * C(0) * \begin{pmatrix} c(-1) \\ T \end{pmatrix}\end{aligned}$$

where `c(-1) = F`. The Boolean matrix multiplication is associative, because `AND` and `OR` satisfy the same associative and distributive laws as multiplication and addition. This algorithm is called *carry look-ahead.* It is used in microprocessors to perfom integer addition.

Problem 21. Consider $f : \{0,1\}^n \to \{0,1\}$. The *Walsh transform* $W_f : \{0,1\}^n \to \mathbf{Z}$ of f is given by

$$W_f(w) = \sum_{x \in \{0,1\}^n} (-1)^{f(x)+w \cdot x}$$

where

$$x = (x_0, x_1, \ldots, x_{n-1}) \qquad x_0, x_1, \ldots, x_{n-1} \in \{0,1\}$$
$$w = (w_0, w_1, \ldots, w_{n-1}) \qquad w_0, w_1, \ldots, w_{n-1} \in \{0,1\}$$

and

$$w \cdot x = w_0 x_0 \oplus w_1 x_1 \oplus \ldots \oplus w_{n-1} x_{n-1}$$

denotes the inner product of w and x. Thus, we find

$$-2^n \leq W_f(w) \leq 2^n .$$

For n even, the set of *bent functions* is the set of functions $f : \{0,1\}^n \to \{0,1\}$ such that

$$|W_f(w)| = 2^{\frac{n}{2}} \quad \forall\, w \in \{0,1\}^n.$$

Find all bent functions for $n = 2$ and $n = 4$.

Solution 21. We identify x with an integer $y(x) \in \mathbf{Z}$ according to

$$y(x) = x_0 + 2x_1 + \cdots + 2^{n-1} x_{n-1}$$

and similarly for w. Thus, we can simply iterate over the values 1, 2, 3, ..., $2^n - 1$ for w and x. The function f is represented as an element of $\{0,1\}^{2^n}$. Thus, we have

$$f \to (f_0, f_1, \ldots, f_{2^n-1}), \qquad f_0, f_1, \ldots, f_{2^n-1} \in \{0,1\}$$
$$f_{y(x)} = f(x) \qquad y(x) = 0, 1, 2 \ldots, 2^n - 1.$$

The following C++ program finds all bent functions for $n = 2$. The value for `const size_t n` should be changed to 4 to find all the bent functions for $n = 4$.

```
// bent.cpp

#include <iostream>
#include <bitset>
using namespace std;

const size_t n = 2;

long ip(bitset<n> x,bitset<n> y)
{ return (x&y).count()%2; }

long walsh(bitset<(1<<n)> f,bitset<n> w)
{
```

```
  long x = 0, sum = 0;
  for(;x<(1<<n);x++)
  {
  if((f[x]+ip(w,bitset<n>(x)))%2) sum--;
  else sum++;
  }
  return sum;
}

int main(void)
{
  long f, w, bent, wf = 0;
  for(f=0;f < (1<<(1<<n));f++)
  {
  bent = 1;
  for(w=0;w<(1<<n) && bent;w++)
  {
  wf = walsh(bitset<(1<<n)>(f),bitset<n>(w));
  if((wf != (1<<(n/2))) && (wf != -(1<<(n/2)))) bent = 0;
  }
  if(bent) cout << bitset<(1<<n)>(f) << endl;
  }
  return 0;
}
```

Problem 22. Given a binary string

```
"100001010110010101110 .... 10111"
```

of finite length n. Let A^* denote the set of all finite-length sequences (strings) over a finite alphabet A (in our case $\{0,1\}$). The quantity $S(i,j)$ denotes the substring $S(i,j) := s_i s_{i+1} \ldots s_j$. A *vocabulary* of a string S, denoted by $v(S)$, is the subset of A^* formed by all the substrings, or words, $S(i,j)$ of S. The *complexity*, in the sense of Lempel and Ziv, of a finite string is evaluated from the point of view of a simple self-delimiting learning machine which, as it scans a given n digit string $S = s_1 s_2 \ldots s_n$ from left to right, adds a new word to its memory every time it discovers a substring of consecutive digits not previously encountered. Thus, the calculation of the complexity $c(n)$ proceeds as follows. Let us assume that a given string $s_1 s_2 \ldots s_n$ has been reconstructed by the program up to the digit s_r and that s_r has been newly inserted, i.e., it was not obtained by simply copying it from $s_1 s_2 \ldots s_{r-1}$. The string up to s_r will be denoted by $R := s_1 s_2 \ldots s_r \circ$, where the $\circ$ indicates that s_r is newly inserted. In order to check whether the rest of R, i.e., $s_{r+1} s_{r+2} \ldots s_n$ can be reconstructed by simple copying or

whether one has to insert new digits, we proceed as follows: First, one takes $Q \equiv s_{r+1}$ and asks whether this term is contained in the vocabulary of the string R so that Q can simply be obtained by copying a word from R. This is equivalent to the question of whether Q is contained in the vocabulary $v(RQ\pi)$ of $RQ\pi$, where $RQ\pi$ denotes the string which is composed of R and Q (concatenation) and π means that the last digit has to be deleted. This can be generalized to situations where Q also contains two (i.e., $Q = s_{r+1}s_{r+2}$) or more elements. Let us assume that s_{r+1} can be copied from the vocabulary of R. Then we next ask whether $Q = s_{r+1}s_{r+2}$ is contained in the vocabulary of $RQ\pi$ and so on, until Q becomes so large that it can no longer be obtained by copying a word from $v(RQ\pi)$ and one has to insert a new digit. The number c of production steps to create the string S, i.e., the number of newly inserted digits (plus one if the last copy step is not followed by inserting a digit), is used as a measure of the complexity of a given string.
(i) Find the complexity of a string which contains only zeros.
(ii) Find the complexity of a string which is only composed of units of 01, i.e.,

$$01010101\ldots01\,.$$

(iii) Find the complexity of the string 0010.
(iv) Write a C++ program that finds the complexity $c(n)$ for a given binary string of length n.

Solution 22. (i) If we have a sequence which contains only zeros, we could say that it should have the smallest possible complexity of all strings (equivalent to a string consisting only of 1's). One only has to insert the first zero and can then reconstruct the whole string by copying this digit, i.e.,

$$00000\ldots \rightarrow 0 \circ 000\ldots\,.$$

Thus, the complexity of this string is $c = 2$.
(ii) Similarly, one finds for a sequence which is only composed of units 01, i.e.,

$$010101\ldots01 \rightarrow 0 \circ 1 \circ 0101\ldots01$$

the value $c = 3$.
(iii) The complexity c of the string $S = 0010$ can be determined as follows:

(1) The first digit has always to be inserted $\rightarrow 0\circ$

(2) $R = 0$, $Q = 0$, $RQ = 00$, $RQ\pi = 0$, $Q \in v(RQ\pi) \rightarrow 0 \circ 0$

(3) $R = 0$, $Q = 01$, $RQ = 001$, $RQ\pi = 00$, $Q \notin v(RQ\pi) \rightarrow 0 \circ 01\circ$

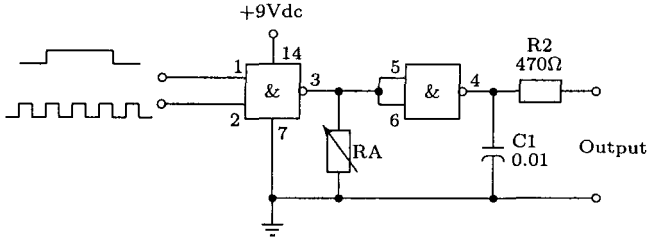
(4) $R = 001$, $Q = 0$, $RQ = 0010$, $RQ\pi = 001$, $Q \in v(RQ\pi) \rightarrow 0 \circ 01 \circ 0$.

Now c is equal to the number of parts of the string that are separated by $\circ$, i.e., $c = 3$.
(iv) A computer program in C++ for finding the complexity uses a do-while loop, a for loop and a while loop as follows:

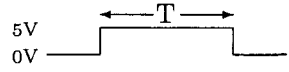
```
// Lempel.cpp

#include <iostream>
#include <string>
using namespace std;

int complexity(const string S,int n)
{
  int c = 1, m = 1;
  do
  {
  int k = 0, kmax = 1;
  for(int i=0;i<m;i++) {
    while(S[i+k] == S[m+k])
    {
    ++k;
    if(m+k >= n-1) return ++c;
    }
    if(k >= kmax) kmax = ++k;
    k = 0;
    }
    ++c;
    m += kmax;
  } while(m < n);
  return c;
}

int main(void)
{
  string S;
  cout << "enter string: ";
  cin >> S;
  int n = S.length();
  cout << "length of string is: " << n << endl;
  cout << "The complexity of the string: ";
  cout << S;
  cout << " is " << complexity(S,n) << endl;
```

```
  return 0;
}
```

Problem 23. The CMOS 4011 contains 4 NAND gates. Two of them can be used to build the following circuit



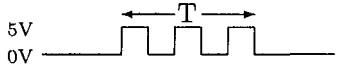
A dual monostable/astable multivibrator (for example 556) provides a monostable output pulse.



The pulse is fed into pin 1 of the NAND gate. The astable output of the multivibrator

is fed into pin 2 of the NAND gate. Discuss the circuit as a function of the gate input at 1 and the pressure sensor (resistor RA). Consider first RA=0 and RA=∞.

Solution 23. When RA=0 the voltage at the output is 0V, we have a short circuit. The second NAND gate (with inputs 5 and 6 and output 4) is used for negation. Thus, when RA=∞, we obtain the astable output of the multivibrator only when the input to pin 1 is HIGH.



Chapter 3

Number Manipulations

Problem 1. Write a C++ program to convert between unsigned integer number representations. For the internal representation, we can use an array of `int` to store the coefficients, where the first entry is the base and the last entry is -1. Thus, to represent the number 43 decimal in binary, we would use the one-dimensional array (read bitstring left to right)

$$(2, 1, 1, 0, 1, 0, 1, -1).$$

Write C++ functions of the form

```
int most_significant(int x,int base);
void tobase(int x,int base,int*& rep);
void convert_base(int* inrep,int base,int*& outrep).
```

The function `most_significant()` calculates the size of the array of coefficients for the representation. For the above example (binary representation of 43) the value returned is 5 (the coefficient of 2^5 is the last entry of the array). The function `tobase()` takes an integer and stores the representation for `x` in `rep`, `tobase()` should allocate the memory for `rep`. Explain the reason for the data type of the third parameter of `tobase()`. The function `convert_base()` is equivalent to `tobase()`, except it takes an arbitrary representation for the integer to convert. Use the program to convert 157 decimal to binary, and then from binary to ternary (base 3) representation.

Solution 1. The function `log()` calculates $\lceil \log_b n \rceil$, i.e., the smallest integer a such that $b^a \geq n$. The function `fromrep()` converts from the representation to an integer. The function `printrep()` prints out the digits of the representation.

```
// convertbase.cpp

#include <iostream>
using namespace std;

int log(int base,int n)
{
  int l = 0, b = 1;
  while(n-b>0) { l++; b *= base; }
  return l-1;
}

int most_significant(int x,int base)
{ return log(base,x); }

unsigned int fromrep(int *r)
{
  int x = 0;
  int b = 1;
  for(int i=1;r[i]!=-1;i++,b*=r[0])
   x += r[i]*b;
  return x;
}

void tobase(int x,int base,int*& rep)
{
  int len = most_significant(x,base)+2;
  rep = new int[len];
  rep[0] = base;
  for(int i=1;i<len;i++)
  { rep[i] = x%base; x /= base; }
  rep[len] = -1;
}

void convert_base(int* inrep,int base,int*& outrep)
{
  int x = fromrep(inrep);
  tobase(x,base,outrep);
}
```

```
void printrep(int* r)
{
  cout << "( ";
  for(int i=1;r[i]!=-1;i++)
  cout << r[i] << " ";
  cout << ") base " << r[0];
}

int main(void)
{
  int *bin, *tern;
  tobase(157,2,bin);
  convert_base(bin,3,tern);
  cout << "Binary representation of " << fromrep(bin)
       << " is ";
  printrep(bin); cout<<endl;
  cout << "Ternary representation of " << fromrep(tern)
       << " is ";
  printrep(tern); cout<<endl;
  delete[] bin; delete[] tern;
  return 0;
}
```

Problem 2. Any positive integer j ($j \in \mathbf{N}$) can be written uniquely as

$$j = 2^n(2s+1), \qquad n, s \in \mathbf{N}_0 .$$

For example,

$$\begin{aligned} 22 &= 2^1(2 \cdot 5 + 1) \\ 17 &= 2^0(2 \cdot 8 + 1) \\ 16 &= 2^4(2 \cdot 0 + 1) . \end{aligned}$$

Given j, write a C++ program which finds n and s which satisfy the above equation.

Solution 2. We note that j is given by the product of an even number multiplied with an odd number. Thus, if we find the largest n such that $j/2^n \in \mathbf{N}$ we can obtain s from

$$s = \frac{1}{2}\left(\frac{j}{2^n} - 1\right) .$$

The following two programs implement this method.

```
// decompose1.cpp

#include <iostream>
using namespace std;

void factor(int j)
{
  int n, s;
  for(n=0;!((j>>n)%2);n++);
  s = ((j>>n)-1)/2;
  cout << j << "=2^" << n << "(2*" << s << "+1)"
       << endl;
}

int main(void)
{
  factor(723);
  factor(800);
  return 0;
}
```

In the next program, we first test whether the number j is even or odd. If the number is odd, then obviously we have $n = 0$.

```
// decompose2.cpp

#include <iostream>
using namespace std;

void test(long j)
{
  long n, s;
  if((j%2) != 0)
  {
  n = 0;
  cout << "n = " << n << endl;
  s = (j-1)/2;
  cout << "s = " << s;
  }
  else
  {
  long counter = 1;
  while(((j/2)%2) == 0)
  {
```

```
  counter++;
  j = j/2;
  }
  n = counter;
  s = (j/2-1)/2;
  cout << "n = " << n << endl;
  cout << "s = " << s;
  } // end else
}

int main(void)
{
  long j = 22; test(j);
  cout << endl;
  j = 13; test(j);
  return 0;
}
```

Problem 3. *Sarkovskii's theorem* describes an ordering of the natural numbers according to which periodicities imply other periodicities for continuous maps on $\mathbf{R}$. The ordering is as follows

$$3 \triangleright 5 \triangleright 7 \triangleright 9 \triangleright \ldots \triangleright 2\cdot 3 \triangleright 2\cdot 5 \triangleright 2\cdot 7 \triangleright \ldots \triangleright 2^2\cdot 3 \triangleright 2^2\cdot 5 \triangleright 2^2\cdot 7 \triangleright \ldots \triangleright 2^3 \triangleright 2^2 \triangleright 2 \triangleright 1$$

In other words, all odd numbers excluding 1 come first followed by the same sequence multiplied by 2 then 2^2 and so in. Lastly, the powers of 2 in decreasing order. Write a C++ program which can determine for any two integers x and y whether $x \triangleright y$ or $y \triangleright x$.

Solution 3. The function `decompose(i,x,y)` finds `x` and `y` such that `i = y2^x` and `y` is odd. The function `sbefore(a,b)` can then compare `a` and `b` in the Sarkovskii ordering.

```
// Sarkovskii.cpp

#include <iostream>
#include <cstdlib>
using namespace std;

void decompose(unsigned int i,unsigned int& x,unsigned int& y)
{
  x = 0;
  while(!(i&1)) { i >>= 1; x++; }
  y = i;
```

```
}

int sbefore(unsigned int x,unsigned int y)
{
  unsigned int x1,y1,x2,y2;
  decompose(x,x1,x2);
  decompose(y,y1,y2);
  if(x2==1 && y2!=1) return 0;
  if(x2!=1 && y2==1) return 1;
  if(x2==1 && y2==1) return y1 < x1;
  if(x2!=1 && y2!=1) return x1 < y1 || (x1 == y1 && x2 < y2);
}

int main(int argc,char *argv[])
{
  unsigned int x, y;
  if(argc!=3) return 1;
  x = atoi(argv[1]); y = atoi(argv[2]);
  if(sbefore(x,y)) cout << x <<" |> " << y << endl;
  if(sbefore(y,x)) cout << y <<" |> " << x << endl;
  return 0;
}
```

Problem 4. Let n be a natural number. The recursive relation used to determine the *Farey fraction* x_k/y_k is given by

$$x_{k+2} = \left\lfloor \frac{y_k + n}{y_{k+1}} \right\rfloor x_{k+1} - x_k$$
$$y_{k+2} = \left\lfloor \frac{y_k + n}{y_{k+1}} \right\rfloor y_{k+1} - y_k$$

where the initial conditions are $x_0 = 0$, $y_0 = x_1 = 1$ and $y_1 = n$. The sequence of x_k/y_k is called the *Farey sequence*. The floor of a denoted by $\lfloor a \rfloor$ is the greatest integer which is not greater than a, for example $\lfloor 5.3 \rfloor = 5$. Write a C++ program to determine the Farey sequence for given n. Determine the first 11 elements of the sequence for $n = 5$.

Solution 4.

```
// Farey.cpp

#include <iostream>
#include <cmath>
```

```
using namespace std;

int main(void)
{
  int n = 5;
  int xk = 0, yk = 1;
  int xk1 = 1, yk1 = n;
  int xk2, yk2;
  int m = 11;
  for(;m>0;m--)
  {
  xk2 = (int) floor(double(yk+n)/yk1)*xk1-xk;
  yk2 = (int) floor(double(yk+n)/yk1)*yk1-yk;
  cout << xk << "/" << yk << endl;
  xk = xk1; yk = yk1;
  xk1 = xk2; yk1 = yk2;
  }
  return 0;
}
```

Problem 5. Consider the following C program.

```
// exp1.c

#include <stdio.h>

int main(void)
{
  int N = 9009;
  int a[9009];
  int n, x = 0;

  for(n=1;n<N;n++) a[n] = 1;
  a[1] = 2;

  for(;N > 9;printf("%d",x))
  for(n=N--;--n>0;)
  {
  a[n] = x%n;
  x = 10*a[n-1]+x/n;
  }
  return 0;
}
```

(i) Describe the algorithms implemented in the program. What is calculated?
(ii) Generate the assembler code for the programs. Explain the assembler code.

Solution 5. (i) The program `exp1.c` determine the 9009 digits of

$$e = 2.71828\ldots .$$

(ii) The comments explain the programs. For the Microsoft Visual C++ compiler we use

```
cl /Fa exp1.cpp
```

to obtain the assembler code in the file `exp1.asm`. From Microsoft Visual C++ we obtain the assembler code (Intel style, line numbers and variable names automatically generated - our comments begin with ;**)

```
TITLE exp1.cpp
.386P
include listing.inc
if @Version gt 510
.model FLAT
else
_TEXT SEGMENT PARA USE32 PUBLIC 'CODE'
_TEXT ENDS
_DATA SEGMENT DWORD USE32 PUBLIC 'DATA'
_DATA ENDS
CONST SEGMENT DWORD USE32 PUBLIC 'CONST'
CONST ENDS
_BSS SEGMENT DWORD USE32 PUBLIC 'BSS'
_BSS ENDS
_TLS SEGMENT DWORD USE32 PUBLIC 'TLS'
_TLS ENDS
FLAT GROUP _DATA, CONST, _BSS
ASSUME CS: FLAT, DS: FLAT, SS: FLAT
endif
PUBLIC _main
EXTRN _printf:NEAR
EXTRN __chkstk:NEAR
_DATA SEGMENT
$SG536 DB '%d', 00H
_DATA ENDS
_TEXT SEGMENT
```

```
_N$ = -36048
_a$ = -36036
_x$ = -36040
_n$ = -36044
_main PROC NEAR
; File exp1.cpp
; Line 6
;** preserve stack **
push ebp
mov ebp, esp
;** allocate memory for a[9009],x,n,N **
mov eax, 36048 ; 00008cd0H
call __chkstk
; Line 7
;** N=9009 **
mov DWORD PTR _N$[ebp], 9009 ; 00002331H
; Line 9
;** x=0 **
mov DWORD PTR _x$[ebp], 0
; Line 11
;** n=1 **
mov DWORD PTR _n$[ebp], 1
jmp SHORT $L530
$L531:
;** n++ **
mov ecx, DWORD PTR _n$[ebp]
add ecx, 1
mov DWORD PTR _n$[ebp], ecx
$L530:
;** while n<N **
mov edx, DWORD PTR _n$[ebp]
cmp edx, DWORD PTR _N$[ebp]
jge SHORT $L532
;** a[n]=1 **
mov eax, DWORD PTR _n$[ebp]
mov DWORD PTR _a$[ebp+eax*4], 1
jmp SHORT $L531
$L532:
; Line 12
;** a[1]=2 **
mov DWORD PTR _a$[ebp+4], 2
; Line 14
jmp SHORT $L533
$L534:
```

```
;** printf("%d",x)
mov ecx, DWORD PTR _x$[ebp]
push ecx
push OFFSET FLAT:$SG536
call _printf
;** remove printf parameters from stack **
add esp, 8
$L533:
;** while N>9 **
cmp DWORD PTR _N$[ebp], 9
jle SHORT $L535
; Line 15
;** n=N N-- **
mov edx, DWORD PTR _N$[ebp]
mov DWORD PTR _n$[ebp], edx
mov eax, DWORD PTR _N$[ebp]
sub eax, 1
mov DWORD PTR _N$[ebp], eax
$L538:
;** n-- **
mov ecx, DWORD PTR _n$[ebp]
sub ecx, 1
mov DWORD PTR _n$[ebp], ecx
;while n>0
cmp DWORD PTR _n$[ebp], 0
jle SHORT $L539
; Line 17
;** eax **
mov eax, DWORD PTR _x$[ebp]
;** convert (signed) eax to edx:eax **
cdq
;** a[n]=x%n **
idiv DWORD PTR _n$[ebp]
mov eax, DWORD PTR _n$[ebp]
mov DWORD PTR _a$[ebp+eax*4], edx
; Line 18
;** ecx=a[n-1] **
mov ecx, DWORD PTR _n$[ebp]
mov ecx, DWORD PTR _a$[ebp+ecx*4-4]
;** ecx=10*a[n-1] **
imul ecx, 10 ; 0000000aH
;** eax=x/n+ecx
mov eax, DWORD PTR _x$[ebp]
cdq
```

```
idiv DWORD PTR _n$[ebp]
add ecx, eax
;** x=10*a[n-1]+x/n **
mov DWORD PTR _x$[ebp], ecx
; Line 19
jmp SHORT $L538
$L539:
jmp $L534
$L535:
; Line 20
;** restore stack and exit **
mov esp, ebp
pop ebp
ret 0
_main ENDP
_TEXT ENDS
END
```

For example, for the GNU C++ compiler we use

```
g++ -S exp1.cpp
```

to obtain the assembler code in the file `exp1.s` in `AT&T` style.

Problem 6. Given a set of N non-negative integers (all the integers are pairwise different). Let N be odd. Write a C++ program which uses an array to represent the set. Implement algorithms to find the minimum, maximum and median of the numbers. The *median* is the number such that $(N-1)/2$ of the numbers are less than the median and the other $(N-1)/2$ numbers ar larger than the median. Also determine the position in the array of the minimum, maximum and median. In each case determine the number of comparisons needed to obtain the result. Consider for example the set

$$\{3, 4, 11, 2, 8, 20, 10\}.$$

The minimum is 2, the maximum is 20 and the median is 8. The position of the median is 4 if we number the positions from 0, from left to right.

Solution 6. To find the minimum and maximum, takes $O(N)$ comparisons. To find the median takes $O(N^2)$ comparisons. For analysis of the problem, see Knuth D. E., *The Art of Computer Programming*, Volume 3, Sorting and Searching, Addison-Wesley, Reading Massachusetts 1981.

```
// median1.cpp
```

```
#include <iostream>
using namespace std;

int min(int set[],int n)
{
  int m = 0;
  for(int i=1;i<n;i++)
  if(set[i]<set[m]) m = i;
  return m;
}

int max(int set[],int n)
{
  int m = 0;
  for(int i=1;i<n;i++)
  if(set[i]>set[m]) m = i;
  return m;
}

int med(int set[],int n)
{
  int i = 0, j = 0, m = 0;
  int count = 0;
  for(;(j<n)&&(count!=n/2);j++)
  {
  count = 0; m = j;
  for(i=0;(i<n);i++)
  if(set[i]>set[m]) count++;
  }
  return m;
}

int main(void)
{
  int a[] = { 3, 4, 11, 2, 8, 20, 10 };
  cout << "minimum " << a[min(a,7)]
       << " at " << min(a,7) << endl;
  cout << "maximum " << a[max(a,7)]
       << " at " << max(a,7) << endl;
  cout << "median " << a[med(a,7)]
       << " at " << med(a,7) << endl;
  return 0;
}
```

Problem 7. The *Taylor series expansion* at $x = 0$ of $(1+x)^{1/4}$ for $x^2 \leq 1$ is given by

$$(1+x)^{1/4} = 1 + \frac{1}{4}x - \frac{1 \cdot 3}{4 \cdot 8}x^2 + \frac{1 \cdot 3 \cdot 7}{4 \cdot 8 \cdot 12}x^3 - \frac{1 \cdot 3 \cdot 7 \cdot 11}{4 \cdot 8 \cdot 12 \cdot 16}x^4 + \cdots .$$

Determine the relation between consecutive terms in the series expansion. Use the relation to implement the expansion recursively and iteratively up to the 10th term in the sum. Determine the number of multiplications performed using the formula given above explicitly and the number of multiplications when the relation is used. In each case, express the answer in terms of n, the number of terms used in the expansion.

Solution 7. We have for the $(n+1)$-th term

$$t_{n+1} = \frac{1 - 4n}{4 + 4n}xt_n$$

where $t_1 = \frac{1}{4}x$. For the explicit expansion, we add 3 multiplication operations for every term after $\frac{1}{4}x$, thus we use $1 + 3(n-2)$ multiplications for the n-th term and a total of $3n(n+1)/2 - 5(n-1)$ for n terms. When the relation is used, we have three multiplications for every term after $\frac{1}{4}x$ so that the total is $1 + 2(n-2)$ for n terms.

```
// expand.cpp

#include <iostream>
#include <cmath>
using namespace std;

double f_r(double x,int terms,double last=1.0,int p=1,int q=4)
{
  if(terms<=0) return 0.0;
  return last+f_r(x,terms-1,(last*x*p)/q,p-4,q+4);
}

double f_i(double x,int terms)
{
  double last = 1.0, sum = 1.0;
  int p = 1, q = 4;

  for(int i=1;i<terms;i++)
  {
  last *= (x*p)/q;
  sum += last;
  p -= 4; q += 4;
```

```
  }
  return sum;
}

int main(void)
{
  cout.precision(8);
  cout << pow(1.5,0.25) << endl;  // => 1.1066819
  cout.precision(8);
  cout << f_r(0.5,10) << endl;  // => 1.1066898
  cout.precision(8);
  cout << f_i(0.5,10) << endl;  // => 1.1066898
  return 0;
}
```

Problem 8. A *Diophantine equation* is an equation where only integer solutions are allowed. A linear Diophantine equation is an equation of the form

$$ax + by = c$$

where $a, b, c, x, y \in \mathbf{Z}$ and x and y are the variables. If we can find the integer solutions (x^*, y^*) of

$$ax^* + by^* = 1$$

then we can find solutions to the first equation using

$$a(cx^*) + b(cy^*) = c\,.$$

Let $r_1 = |a|$ and $r_2 = |b|$. We apply the *Euclidean algorithm.*

$$\begin{aligned} r_1 &= q_1 r_2 + r_3 \\ r_2 &= q_2 r_3 + r_4 \\ &\vdots \\ r_{n-3} &= q_{n-3} r_{n-2} + r_{n-1} \\ r_{n-2} &= q_{n-2} r_{n-1} + 1\,. \end{aligned}$$

From the last two equations we obtain

$$\begin{aligned} 1 &= r_{n-2} - q_{n-2} r_{n-1} \\ &= r_{n-2} - q_{n-2}(r_{n-3} - q_{n-3} r_{n-2}) \\ &= -q_{n-2} r_{n-3} + (1 - q_{n-2} q_{n-3}) r_{n-2} \\ &\vdots \\ &= x^* r_1 + y^* r_2 \\ &= |a| x^* + |b| y^*\,. \end{aligned}$$

In other words we use

$$cr_i + dr_{i+1} = dr_{i-1} + (c - dq_{i-1})r_i.$$

Thus, we obtain a recursion relation which we can use to find the solution. To satisfy the equation, we use $|x| = |x^*|$, $\operatorname{sgn}(x) = \operatorname{sgn}(a)$, $|y| = |y^*|$ and $\operatorname{sgn}(y) = \operatorname{sgn}(b)$. For example, we consider the equation $1027x + 712y = 1$. First we apply the Euclidean algorithm

$1027 = 712 \cdot 1 + 315$	$q_1 = 1$
$712 = 315 \cdot 2 + 82$	$q_2 = 2$
$315 = 82 \cdot 3 + 69$	$q_3 = 3$
$82 = 69 \cdot 1 + 13$	$q_4 = 1$
$69 = 13 \cdot 5 + 4$	$q_5 = 5$
$13 = 4 \cdot 3 + 1$	$q_6 = 3$

Now we apply the recursion relation

$$\begin{aligned}
1 &= r_6 - 3r_7 \\
&= -3r_5 + (1 + 3 \cdot 5)r_6 \\
&= -3r_5 + 16r_6 \\
&= 16r_4 + (-3 - 16 \cdot 1)r_5 \\
&= 16r_4 - 19r_5 \\
&= -19r_3 + (16 + 19 \cdot 3)r_4 \\
&= -19r_3 + 73r_4 \\
&= 73r_2 + (-19 - 73 \cdot 2)r_3 \\
&= 73r_2 - 165r_3 \\
&= -165r_1 + (73 + 165 \cdot 1)r_2 \\
&= -165r_1 + 238r_2\,.
\end{aligned}$$

Thus we obtain the solution $x = -165$ and $y = 238$.

Write a C++ program which solves linear Diophantine equations of the form $ax + by = 1$.

Solution 8. The function `solve_linear()` solves linear diophantine equation $ax + by = 1$ for x and y given a and b.

```
// diophantine.cpp

#include <iostream>
#include <vector>
using namespace std;
```

```
void solve_linear(int a,int b,int& x,int& y)
{
  int t, larger;
  int r1, r2, r3;
  vector<int> q;

  if(a == 0) return; if(b == 0) return;
  larger = (b>a) ? 1:0;
  r1 = (a>0) ? a:-a;
  r2 = (b>0) ? b:-b;
  if(larger) { t = r2; r2 = r1; r1 = t; }

  do
  {
  r3 = r1%r2;
  q.push_back(r1/r2);
  r1 = r2; r2 = r3;
  }
  while(r3 != 1);

  x = 1; y = -q.back();
  q.pop_back();
  while(q.size() > 0)
  {
  t = x; x = y;
  y = t - y*q.back();
  q.pop_back();
  }
  x = (a > 0) ? x:-x;
  y = (b > 0) ? y:-y;
  if(larger) { t = x; x = y; y = t; }
}

int main(void)
{
  int x, y;
  solve_linear(1027,712,x,y);
  cout << "1027*(" << x
       << ") + 712*(" << y << ") = 1" << endl;
  return 0;
}
```

Problem 9. Definition. We define the *highest common divisor* d of two positive integers a and b as the largest positive integer which divides both a and b. We write

$$d = (a, b) .$$

Definition. The *Euler totient function* is defined on the set of positive integers by $f(1) = 1$, $f(m)$ for $m > 1$ is the number of positive integers less than m and relatively prime to m. In other words we denote by $f(m)$ the number of positive integers not greater than and relatively prime to m, that is to say the number of integers n such that

$$0 < n \leq m \qquad (n, m) = 1 .$$

The number n can be equal to m only when $n = 1$. Write a Java program using the class `BigInteger` to find $f(m)$.

Definition. A function $g(m)$ is said to be multiplicative if $(m, m') = 1$ implies

$$g(mm') = g(m)g(m') .$$

Theorem. The Euler totient function $f(m)$ is multiplicative, i.e.,

$$f(mm') = f(m)f(m') .$$

Can this property be used in the program?

Solution 9. The Java class `BigInteger` includes the method

```
BigInteger gcd(BigInteger val)
```

for the greatest common divisor. Since Java does not provide operator overloading the `for` loop in the program looks quite clumsy.

```
// Totient.java

import java.math.*;

public class Totient
{
  public static void main(String[] args)
  {
  BigInteger n = new BigInteger("10");

  for(BigInteger i = new BigInteger("3");i.compareTo(n) <= 0;
     i = i.add(BigInteger.ONE))
  {
  BigInteger z = new BigInteger("1");
```

```
  BigInteger k = i;
  k = k.subtract(BigInteger.ONE);
  for(BigInteger j = new BigInteger("2");j.compareTo(k) <= 0;
     j = j.add(BigInteger.ONE))
  {
  if((j.gcd(i)).equals(BigInteger.ONE))
  z = z.add(BigInteger.ONE);
  } // end for loop j
  System.out.println("f(" + i + ") = " + z);
  } // end for loop i
  }
}
```

A C++ program using the class `Verylong` from SymbolicC++ is shorter since the operators `+`, `-`, `++` etc are overloaded.

Problem 10. *Perfect numbers* are those integers which are the sum of their positive proper divisors. For example,

```
  6 = 1 + 2 + 3
 28 = 1 + 2 + 4 + 7 + 14
496 = 1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248.
```

Write a Java program using the data type `long` that finds larger perfect numbers.

Solution 10. Larger perfect numbers are 8128, 33550336 and 8589869056.

```
// Perfect.java

class Perfect
{
   public static void main(String[] args)
   {
   long maxnumber = 100000000L;
   long halfmax;
   for(long pos=3L;pos <= maxnumber;pos++)
   {
   long half = pos/2L + 1L;
   long sum = 0L;
   for(long factor=1L;factor <= half;factor++)
   {
   if(pos%factor == 0L)     // divisible
   {
   sum += factor;           // add factor
```

```
    }
    }
    if(sum == pos)
    {
    System.out.println("" + pos + " is a perfect number");
    System.out.print("Factors are: ");
    for(long f=1L;f < pos;f++)
    {
    if(pos%f == 0L)
    {
    System.out.print(" " + f);  // print factors
    }
    }
    System.out.println();
    }
    }
    }
}
```

We can also find perfect numbers from $(2^n - 1) \cdot 2^{n-1}$, where $2^n - 1$ is a Mersenne prime.

Problem 11. (i) Consider the following arithmetic problem

	ab*c
	de
+	fg
	hi

where $a, b, c, d, e, f, g, h, i$ stand for a nonzero digit, i.e., they are elements of the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, * denotes multiplication and + addition. Each nonzero digit occurs only once in the problem. Is there a solution? If so, is the solution unique?
(ii) Write a C++ program that finds a solution (if one exists).

Solution 11. (i) We find one solution

```
ab = 17    c = 4    de = 68    fg = 25     hi = 93
```

i.e.,

```
a = 1, b = 7, c = 4, d = 6, e = 8, f = 2, g = 5, h = 9, i = 3.
```

(ii) The C++ program runs through all *permutations* of the numbers 1, 2, 3, 4, 5, 6, 7, 8, 9.

```
// arithproblem.cpp

#include <iostream>
using namespace std;

int main(void)
{
  int i, j, k, t, tau;
  unsigned long n = 9;
  int* p = NULL; p = new int[n+1];
  // starting permutation
  // identity 1, 2, ... , n -> 1, 2, ... , n
  // which is not a solution of the problem
  for(i=0; i <= n; i++)
  {
  p[i] = i;
  }
  int test = 1;
  do
  {
  i = n-1;
  while(p[i] > p[i+1]) i = i-1;
  if(i > 0) test = 1; else test = 0;
  j = n;
  while(p[j] <= p[i]) j = j-1;
  t = p[i]; p[i] = p[j]; p[j] = t; i = i+1; j = n;
  while(i < j)
  {
  t = p[i]; p[i] = p[j]; p[j] = t;
  i = i+1; j = j-1;
  }
  int ab = 10*p[1] + p[2]; int de = 10*p[4] + p[5];
  int fg = 10*p[6] + p[7]; int hj = 10*p[8] + p[9];
  int c = p[3];
  if((ab*c == de) && (de + fg == hj))
  {
  for(tau=1;tau <=n;tau++)
  cout << "p[" << tau << "]=" << p[tau] << "  ";
  cout << endl;
  }
  } while(test == 1);
  delete[] p;
  return 0;
 }
```

Problem 12. The puzzle to be solved is to map (1 to 1) each letter

`(A,B,C,D,E,F,G,H,J,K)`

in the following figure to a digit

$$(0,1,2,3,4,5,6,7,8,9)$$

such that the 3 horizontal and 3 vertical computations are correct.

EDKH	/	KF	=	AA
−		+	=	+
EDB	*	J	=	EHCG
EEJD	−	DK	=	EEAE

Write a C++ program using brute force to solve the problem. Is there a solution? Is the solution unique? How many permutations have to be checked?

Solution 12. There are $10! = 3628800$ permutations from which only one

`ABCDEFGHJK = 5793146082`

solves the puzzle. In the following C++ program the ten `for`-loops include all possible permutation and finds this solution. Thus the number of iterations is 10^{10}. Can this redundancy in the program be removed?

```
// puzzle.cpp

#include <iostream>
using namespace std;

int main(void)
{
  for(int i0=0;i0<10;i0++)
  for(int i1=0;i1<10;i1++)
  for(int i2=0;i2<10;i2++)
  for(int i3=0;i3<10;i3++)
  for(int i4=0;i4<10;i4++)
  for(int i5=0;i5<10;i5++)
  for(int i6=0;i6<10;i6++)
  for(int i7=0;i7<10;i7++)
  for(int i8=0;i8<10;i8++)
  for(int i9=0;i9<10;i9++)
  {
  int A = i0; int B = i1; int C = i2; int D = i3;
```

```
  int E = i4; int F = i5; int G = i6; int H = i7;
  int J = i8; int K = i9;
  int EDKH = 1000*E + 100*D + 10*K + H;
  int KF = 10*K + F;  int AA = 10*A + A;
  int EDB = 100*E + 10*D + B;
  int EHCG = 1000*E + 100*H + 10*C + G;
  int EEJD = 1000*E + 100*E + 10*J + D;
  int DK = 10*D + K;
  int EEAE = 1000*E + 100*E + 10*A + E;
  if(KF != 0)
  {
  if((EDKH/KF == AA) && (EDKH%KF == 0) &&
    ((EDKH-EDB) == EEJD) && ((KF+J) == DK) &&
    ((AA+EHCG) == EEAE) && ((EDB*J) == EHCG) &&
    ((EEJD-DK) == EEAE))
  {
  cout << "A solution is: " <<
  i0 << i1 << i2 << i3 << i4 << i5 << i6 << i7 << i8 << i9;
  }
  }
  } // end for
  return 0;
}
```

Problem 13. Write a C++ program that finds the integer square root of a positive integer. For example, 8 is the integer square root of 70, since $8 \cdot 8 = 64$ and $9 \cdot 9 = 81$.

Solution 13. Can the following C++ program be improved?

```
// intsqrt.cpp

#include <iostream>
using namespace std;

unsigned long int_sqrt(unsigned long n)
{
  unsigned long nn = 0;

  // counts digits in number
  int digits = 0;
  do { digits++; nn /= 10; }
     while(nn != 0);
```

```
  // estimate lower bound for root
  int d_min = digits/2;
  unsigned long root;
  if(digits%2 == 0) { root = 3; d_min -= 1; }
  else root = 1;

  for(int i=0;i<d_min;i++) root *= 10;

  // search for root
  while(root*root < n) root++;
  if(root*root != n) root--;
  return root;
}

int main(void)
{
  unsigned long n;
  cout << "Enter positive integer number: ";
  cin >> n;
  unsigned long root = int_sqrt(n);
  cout << "integer square root = " << root;
  return 0;
}
```

Problem 14. For pseudo-random number generators one quite often uses the map

$$Y_{t+1} = (AY_t + B) \bmod C$$

where $t = 0, 1, 2, \ldots$ and Y_0 is the initial value (seed). Here A is the number of cycles, B is the phase (horizontal shift), C is the number of distinct values. The sequence will repeat after at most C steps. Let

$$A = 1366, \quad B = 150889, \quad C = 714025\,.$$

Give a Java implementation of this map. Use the data type `long`.

Solution 14.

```
// MyRandom.java

public class MyRandom
{
   public static void main(String[] args)
```

```
   {
   long A = 1366, B = 150889, C = 714025;
   long Y0 = 15; // seed
   long R = 15;
   long Y1;
   long count = 0;
   do
   {
   Y1 = (A*Y0 + B)%C;
   Y0 = Y1;
   System.out.println(Y0);
   count++;
   } while(Y0 != R);
   System.out.println("Y1 = " + Y1);
   System.out.println("count = " + count);
   }
}
```

Problem 15. Given a floating point number within the range $[0, 1]$. Write a C++ program that finds the binary representation of this number.

Solution 15. Any fraction x can be written as

$$x = a_1 \cdot 2^{-1} + a_2 \cdot 2^{-2} + \cdots$$

where $a_j \in \{ 0, 1 \}$. The character string

$$0.a_1 a_2 \ldots$$

is then x in binary form. Since

$$2x = a_1 + a_2 \cdot 2^{-1} + a_3 \cdot 2^{-2} + \cdots$$

one immediately has (x is decimal form) $\text{int}(2x) = a_1$ and

$$\text{frac}(2x) = a_2 \cdot 2^{-1} + a_3 \cdot 2^{-2} + \cdots .$$

It is obvious that these steps can be iterated to find a_j. What is the problem with the following code for large n? For example, if we enter $x = 0.4$ and $n = 100$. Discuss rounding errors.

```
// binfrac.cpp

#include <iostream>
#include <cmath>
```

```
using namespace std;

void binfrac(double x,int n)
{
  for(int j=0;j<n;j++)
  {
  double fraction;
  int integer;
  x = 2.0*x;
  fraction = fmod(x,1.0);
  integer = x - fraction;
  x = fraction;
  cout << integer;
  }
}

int main(void)
{
  double x;
  cout << "enter decimal fraction: "; cin >> x;
  int n;
  cout << "enter number of binary digits required: ";
  cin >> n;
  cout << endl;
  cout << "binary representation of " << x << " is 0.";
  binfrac(x,n);
  return 0;
}
```

Problem 16. An algorithm based on the *public key encryption scheme* was introduced by Rivest, Shamir and Adleman. The public encryption key is a pair (e, n). The private key is a pair (d, n), where e, d, and n are positive integers. Each message m is represented as an integer between 0 and $n-1$. A long message is broken into a series of smaller messages, each of which can be represented as such an integer. The functions E and D are defined as

$$E(m) = m^e \bmod n = C$$
$$D(C) = C^d \bmod n\,.$$

The integer n is computed as the product of two large (100 digits or more) randomly chosen prime numbers p and q with $n = p \cdot q$. The integer d

is chosen to be a large, randomly chosen integer relatively prime to the product $(p-1)\cdot(q-1)$. That is, d satisfies

$$\text{greatest common divisor}\,[d,(p-1)\cdot(q-1)]=1\,.$$

Finally, the integer e is computed from p, q, and d to be the multiplicative inverse of d modulo $(p-1)\cdot(q-1)$. That is, e satisfies

$$e\cdot d \bmod (p-1)\cdot(q-1)=1\,.$$

Although n is known, p and q are not. This condition is due to the fact that it is very difficult to factor n. Consequently, the integers d and e cannot be easily guessed. Apply the algorithm to the numbers $p=5$, $q=7$ and $m=3$.

Solution 16. From $p=5$ and $q=7$ we find $n=35$ and $(p-1)\cdot(q-1)=24$. Since 11 is relatively prime to 24, we can choose $d=11$, and since

$$11\cdot 11 \bmod 24 = 121 \bmod 24 = 1$$

we have $e=11$. Suppose now that $m=3$. Then

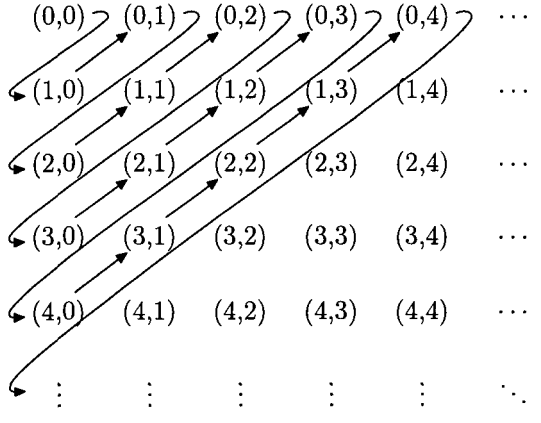
$$C=m^e \bmod n = 3^{11} \bmod 35 = 12$$

and

$$C^d \bmod n = 12^{11} \bmod 35 = 3 = m\,.$$

Thus, if we encode m using e, we can decode m using d.

Problem 17. Let $\mathbf{N}_0$ be the natural numbers including 0. A *pairing function* is a computable bijection from $\mathbf{N}_0^2$ to $\mathbf{N}_0$ which provides an effective enumeration of pairs of integers. Provide such a map.



Solution 17. The map given by

$$f(i,j) = \frac{1}{2}(i+j)(i+j+1) + i$$

for all $i, j \in \mathbf{N}_0$ is a computable one-to-one correspondence from $\mathbf{N}_0^2$ onto $\mathbf{N}_0$. We have

$$f(0,0) = 0, \quad f(0,1) = 1, \quad f(1,0) = 2, \quad f(0,2) = 3, \quad f(1,1) = 4$$

etc. We obtain f by adaptation of *Cantor's enumeration* for rational numbers. Given $f(i,j)$. How can we find i and j?

Problem 18. Multiplication of large positive integers can be done as follows. Using the *divide-and-conquer algorithm* we split an n-digit integer into two integers of approximately $n/2$ digits. For example

$$567832 = 567 \cdot 10^3 + 832$$

$$9423723 = 9423 \cdot 10^3 + 723\,.$$

In general, if n is the number of digits in the integer u, we split the integer into two integers, one with $\lceil n/2 \rceil$ digits and the other with $\lfloor n/2 \rfloor$ digits, as follows

$$u = x \cdot 10^m + y$$

where x has $\lceil n/2 \rceil$ digits and y has $\lfloor n/2 \rfloor$ digits. With this representation, the exponent m of 10 is given by

$$m = \left\lfloor \frac{n}{2} \right\rfloor .$$

If we have two n-digit integers,

$$u = x \cdot 10^m + y$$
$$v = w \cdot 10^m + z$$

their product is given by

$$\begin{aligned} uv &= (x \cdot 10^m + y)(w \cdot 10^m + z) \\ &= xw \cdot 10^{2m} + (xz + wy) \cdot 10^m + yz\,. \end{aligned}$$

The method uses integers with about the same number of digits. However, it can also be applied when this is not the case. We simply use $m = \lfloor n/2 \rfloor$ to split both of them, where n is the number of digits in the larger integer. Write a Java application that implements this algorithm to multiply to positive integers of data type `long`.

Solution 18. The method `noofdigits()` finds the number of digits in the number `m`.

```
// LargeMulti.java

public class LargeMulti
{
  public static long product(long u,long v)
  {
  long x, y, w, z, p;
  long t = 4; // threshold
  long a = noofdigits(u);
  long b = noofdigits(v);
  long n = Math.max(a,b);
  if((u == 0L) || (v == 0L)) { return  0; }
   else
    if(n <= t) { p = u*v; return p; }
    else
    {
    long m = n/2L;
    x = u/((long) Math.pow(10,m));
    y = u%((long) Math.pow(10,m));
    w = v/((long) Math.pow(10,m));
    z = v%((long) Math.pow(10,m));
    p = product(x,w)*((long) Math.pow(10,2*m))
        +(product(x,z)+product(w,y))*((long) Math.pow(10,m))
        +product(y,z);
    return p;
    }
  }

  public static long noofdigits(long m)
  {
  Long I = new Long(m);
  String s = I.toString();
  long length = s.length();
  return length;
  }

  public static void main(String[] args)
  {
  long u = 98123456L, v = 1012341234567L;
  long n1 = noofdigits(u);
  long n2 = noofdigits(v);
  System.out.println("result1 = " + n1);
  System.out.println("result2 = " + n2);
  long s = product(u,v);
```

```
  System.out.println("s = " + s);
  long t = u*v;
  System.out.println("t = " + t);
  }
}
```

Problem 19. Consider the bijective *spiral map*

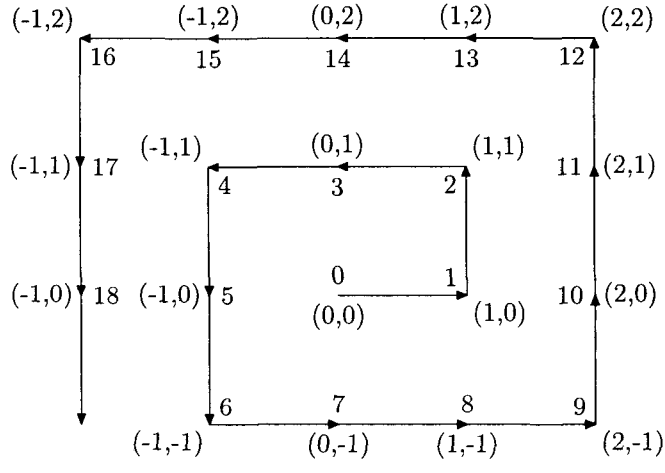
$$f : \mathbf{Z} \times \mathbf{Z} \to \mathbf{N}_0$$

displayed in the following figure. We have

$$f(0,0) = 0, \quad f(1,0) = 1, \quad f(1,1) = 2, \quad f(0,1) = 3,$$

$$f(-1,1) = 4, \quad f(-1,0) = 5, \quad f(-1,-1) = 6, \quad f(0,-1) = 7, \ldots$$

Write a C++ program that given $(m,n) \in \mathbf{Z} \times \mathbf{Z}$ finds $f(m,n)$. Implement also the inverse transformation.



Solution 19. The following program implements the spiral map and its inverse. The spiral map is determined by following the spiral in reverse until the point $(0,0)$ is reached.

```
// spiralmap.cpp

#include <iostream>
using namespace std;

int main(void)
```

```
{
  int mi, ni, m, n, count = 0;
  int f1, f;
  cout << "Enter m, n: "; cin >> mi >> ni;
  m = mi; n = ni;

  while((m!=0) || (n!=0))
  {
  if((n>m) && (n>=-m))          ++m;
  else if((n>=m) && (n < -m))   ++n;
  else if((n<=m) && (n > -m+1)) --n;
  else                          --m;
  count++;
  }
  cout << "f(" << mi << "," << ni << ") = " << count << endl;

  // inverse map
  cout << "Enter the number: ";
  cin >> f;
  f1 = f;
  m = n = 0;
  if(f-- != 0) m++;

  while(f > 0)
  {
  if((n>=m) && (n>-m))        --m;
  else if((n>m) && (n<=-m))   --n;
  else if((n<m) && (n>=-m+1)) ++n;
  else                        ++m;
  f--;
  }
  cout << endl;
  cout << "f^-1(" << f1 << ") = ("
       << m << "," << n << ")" << endl;
  return 0;
}
```

Problem 20. The naive approach to multiplying two integers X and Y represented as binary numbers is $O(n^2)$, where the integers have n bits. We can write a faster multiplication algorithm if we split the integers in two,

$$X = A2^{n/2} + B$$
$$Y = C2^{n/2} + D\,.$$

The multiplication can now be written as

$$XY = AC2^n + (AD + BC)2^{n/2} + BD\,.$$

The number of multiplies, additions and shifts still take $O(n)$ steps. If we rewrite the multiplication as

$$XY = AC2^n + [(A - B)(D - C) + AC + BD]2^{n/2} + BD\,,$$

then we replace the two multiplications AD, BC by one and we can reuse the results AC and BD. This algorithm can be applied recursively resulting in an algorithm that is $O(n^{1.5})$. Write a C++ program to multiply two large integers using the bitset class. Use a two's-complement representation for the integers. If we represent a polynomial P in x as a list of the coefficients of the terms in P, then the same technique can be applied to speed up multiplication of polynomials. Instead of the shift operation 2^n we have multiplication by x^n. This is known as the *Karatsuba-Ofman algorithm.*

Solution 20.

```
// fastmult.cpp

#include <iostream>
#include <bitset>
using namespace std;

template<size_t T> int sign(bitset<T> X)
{
  if(X.test(T-1)) return -1;
    else return 1;
}

template<size_t T> bitset<T> neg(bitset<T> X)
{
   bitset<T> one(1);
   for(int i=0;i<T;i++) { X.flip(i); }
   X = X + one;
   return X;
}

template<size_t T> bitset<T> abs(bitset<T> X)
{
   if(sign(X) < 0) { return neg(X); }
     else { return X; }
}
```

```
template<size_t T> double value(bitset<T> x)
{
  int s = sign(x);
  x = abs(x);
  double val = 0.0;
    for(int i=T-1;i>=0;i--)
    {
    val *= 2.0;
    if(x[i]) { val += 1.0; }
    }
    if(s < 0) val = -val;
  return val;
}


template<size_t T>
bitset<T> operator + (bitset<T> X,bitset<T> Y)
{
  int bit, carry = 0;
  bitset<T> answer;
  for(int i=0;i<T;i++)
  {
  bit = ((X.test(i))?1:0)+((Y.test(i))?1:0)+carry;
    if(bit & 1) { answer.set(i); }
      else { answer.reset(i); }
         carry = bit >> 1;
  }
  return answer;
}

template<size_t T>
bitset<T> operator - (bitset<T> X,bitset<T> Y)
{ return X + neg(Y); }

template<size_t T>
bitset<T> mult(bitset<T> X,bitset<T> Y,int n)
{
   bitset<T> A, B, C, D, AC, BD, A_BD_C, answer;
   int i;
   int s = sign(X)*sign(Y);
   X = abs(X); Y = abs(Y);
   if(n==1)
```

```
    {
    if((X.test(0))&&(Y.test(0))) { answer.set(0); }
    } else { A = X >> (n/2); B = X;
    for(i=T-1;i>=n/2;i--) B.reset(i);
    C = Y >> (n/2);  D = Y;
    for(i=T-1;i>=n/2;i--) D.reset(i);
      AC = mult(A,C,n/2);
      A_BD_C = mult(A-B,D-C,n/2);
      BD = mult(B,D,n/2);
      answer = ((AC<<n) + ((AC+A_BD_C+BD)<<(n/2)) + BD);
      }
      if(s < 0) answer = neg(answer);
      return answer;
}

int main(void)
{
  bitset<128> X(2000000000);
  bitset<128> Y(300000000);
  bitset<128> Z;
  X = neg(X);
  Z = mult(X,Y,128);
  cout << value(X) << "*" << value(Y) << "="
       << value(Z) << endl;
  return 0;
}
```

Problem 21. The *Zeckendorf representation* $(z_2, z_3, \ldots, z_{n-1}) \in \{0,1\}^{n-2}$ of a non-negative integer z is given by

$$z = \sum_{i=2}^{n-1} z_i F_i$$

where $F_1 = F_2 = 1$ and $F_{i+2} = F_{i+1} + F_i$ are the *Fibonacci numbers* , and F_n is the smallest Fibonacci number such that $F_n > z$. The representation is obtained by repeatedly subtracting the largest Fibonacci number F_j with $z > F_j$, and writing a 1 in this place ($z_j = 1$). This representation is unique for any given z. Furthermore, if $z_j \neq 0$ then $z_{j+1} = 0$, i.e., the representation can never have two consecutive 1s. Suppose two consecutive 1s appear in the representation, say at position k and $k+1$, then from $F_{k+2} = F_{k+1} + F_k$, F_{k+1} could not have been the largest Fibonacci smaller than z, and consequently two consecutive 1s cannot appear in the representation. For example, we find the Zeckendorf representation of 19 is 100101

where the first digit is z_2, the second z_3 etc. The first Fibonacci number which is greater than 19 is 21.

j	F_j	z	z_j
8	21	19	
7	13	19	1
6	8	6	0
5	5	6	1
4	3	1	0
3	2	1	0
2	1	1	1

The Zeckendorf representation leads to *Zeckendorf arithmetic*, i.e., the arithmetic based on the Zeckendorf representation of numbers. To add two numbers x and y in the Zeckendorf representation

$$x = \sum_{j=2}^{n-1} x_j F_j, \qquad y = \sum_{j=2}^{n-1} y_j F_j$$

we use $(x+y)_j = x_j + y_j$ whenever $x_j + y_j < 2$. However, for $x_j = y_j = 1$ we have to use the following property of the Fibonacci numbers

$$2F_j = \begin{cases} F_3 & j = 2 \\ F_2 + F_4 & j = 3 \\ F_{j-2} + F_{j+1} & \text{otherwise} \end{cases}.$$

This follows from the fact that

$$F_j + F_j = F_j + (F_{j+1} - F_{j-1}) = F_j + (F_{j+1} - (F_j - F_{j-2}))$$

for $j \geq 3$. After addition, the result must be normalized to ensure that no consecutive 1s appear for a valid Zeckendorf representation. Thus, it is necessary to replace 110 with 001. If the replacement is done from the right side of the representation we can ensure that a zero always follows the rightmost pair of 1s. Consequently we have the following algorithm to add two Zeckendorf representations x and y

add(x, y)

1. Define *sum* such that $sum_j = x_j \oplus y_j$ (i.e. $sum_j = x_j + y_j \bmod 2$). Define b such that $b_j = x_j \cdot y_j$.

2. Set $sum :=$ normalize(sum).

3. For $j = 2, 3, \ldots$
   If $j = 2$ set $c :=$ 01, if $j = 3$ set $c :=$ 101, if $j = 4$ set $c :=$ 1001.
   If $j > 4$ set $c :=$ 0c.
   If $b_j = 1$ then $sum :=$ add(sum, c).

4. The result is stored in *sum*.

For multiplication we can use a technique similar to the russian farmer multiplication. We use the property that $F_1 x = F_2 x = x$ and

$$F_{j+2}x = (F_{j+1} + F_j)x = (F_{j+1}x) + (F_j x).$$

multiply(x, y)

1. Set $a1 := y$.
   Set $a2 := y$.
   Set $sum := 0$.

2. For $j = 2, 3, \ldots$
   If $x_j = 1$ then $sum := \text{add}(sum, a2)$.
   Set $a0 := a1$ and $a1 := a2$.
   Set $a2 := \text{add}(a0, a1)$.

3. The result is stored in *sum*.

Give a C++ implementation of the Zeckendorf arithmetic.

Solution 21. The C++ program implements the Zeckendorf arithmetic. The function `itoz()` converts an integer to the Zeckendorf representation while `ztoi()` converts the Zeckendorf representation back to an integer. The function `znormalize(vector<bool>& b)` will normalize `b` for the Zeckendorf representation. The two functions `zadd()` and `zmul()` perform addition and multiplication in the Zeckendorf representation, respectively.

```
// zeckendorf.cpp

#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

// overloaded operator to output vector<bool> for the
// Zeckendorf representation
ostream& operator << (ostream& o,const vector<bool>& b)
{ for(int i=0;i<b.size();i++) o << b[i]; return o; }

// Zeckendorf representation to integer
int ztoi(const vector<bool>& b)
{
  int f0, f1 = 1, f2 = 2, i, z = 0;
  for(i=0;i<b.size();i++)
```

```
  {
  if(b[i]) z += f1;
  f0 = f1; f1 = f2; f2 = f1 + f0;
  }
  return z;
}

// integer to Zeckendorf representation
vector<bool> itoz(int z)
{
  int f0 = 1, f1 = 1, f2 = 2;
  vector<bool> b;
  while(z >= f2) { f0 = f1; f1 = f2; f2 = f1 + f0; }
  while(f1 != f2)
  {
  if(z >= f1) { z -= f1; b.insert(b.begin(),1,1); }
  else                   b.insert(b.begin(),1,0);
  f0 = f2 - f1; f2 = f1; f1 = f0;
  }
  return b;
}

// normalize a vector<bool> for the Zeckendorf representation
// i.e. remove all consecutive 1's
void znormalize(vector<bool>& b)
{
  for(int i=b.size()-1; i>=1; i--)
  {
  if((b[i] == 1) && (b[i-1] == 1))
  {
  b[i] = b[i-1] = 0;
  if(i == b.size()-1) b.push_back(1); else b[i+1] = 1;
  if(i+3 >= b.size()) i = b.size(); else i += 3;
  }
  }
  if(b.size() == 0) b.push_back(0);
}

// addition for Zeckendorf representations
vector<bool> zadd(const vector<bool>& x,const vector<bool>& y)
{
  int i;
  bool b1, b2;
  vector<bool> sum, c2(2), c3(3), c4(4);
```

```
  c2[0] = 0; c2[1] = 1;
  c3[0] = 1; c3[1] = 0; c3[2] = 1;
  c4[0] = 1; c4[1] = 0; c4[2] = 0; c4[3] = 1;
  for(i=0;(i<x.size()) || (i<y.size());i++)
  {
  if(i<x.size()) b1 = x[i]; else b1 = 0;
  if(i<y.size()) b2 = y[i]; else b2 = 0;
  if(b1 ^ b2) sum.push_back(1); else sum.push_back(0);
  }
  znormalize(sum);
  for(i=0;(i<x.size()) && (i<y.size());i++)
  {
  if(x[i] & y[i])
  {
  if(i==0)      sum = zadd(sum,c2); // 2*F_2
  else if(i==1) sum = zadd(sum,c3); // 2*F_3
  else          sum = zadd(sum,c4); // 2*F_j, j>3
  }
  if(i>1) c4.insert(c4.begin(),1,0);
  }
  return sum;
}

// multiplication for Zeckendorf representations
vector<bool> zmul(const vector<bool>& x,const vector<bool>& y)
{
  vector<bool> a0, a1(y), a2(y), sum;
  for(int i=0;i<x.size();i++)
  {
  if(x[i] == 1) sum = zadd(sum,a2);
  a0 = a1; a1 = a2; a2 = zadd(a0,a1);
  }
  return sum;
}

int main(void)
{
  int i, j;
  cout << "integer -> Zeckendorf : " << endl;
  for(i=0;i<22;i++) cout << i << " -> " << itoz(i) <<endl;
  cout << endl;
  cout << "Zeckendorf addition : " << endl;
  for(i=0;i<22;i++)
  {
```

```
  for(j=0;j<22;j++)
  cout << setw(2) << ztoi(zadd(itoz(i),itoz(j))) << " ";
  cout << endl;
  }
  cout << endl;
  cout << "Zeckendorf multiplication : " << endl;
  for(i=0;i<11;i++)
  {
  for(j=0;j<11;j++)
  cout << setw(2) << ztoi(zmul(itoz(i),itoz(j))) << " ";
  cout << endl;
  }
  cout << endl;
  return 0;
}
```

Problem 22. Determine the coefficient of $x_1x_2^3x_4x_5^5$ in the polynomial $(x_1 + x_2 + x_3 + x_4 + x_5)^{10}$.

Solution 22. To multiply the term $(x_1 + x_2 + x_3 + x_4 + x_5)$ ten times with itself and then collect the terms would be quite cumbersome. Using the *multinomial theorem* , the coefficient of $x_1x_2^3x_4x_5^5$ in $(x_1+x_2+x_3+x_4+x_5)^{10}$ is given by

$$\frac{10!}{1!3!0!1!5!} = 5040\,.$$

Chapter 4

Combinatorical Problems

Problem 1. Let A and B be finite sets. Let $|A|$ and $|B|$ be the *cardinality* (i.e., the number of elements in the set) of A and B, respectively. Let

$$f : A \rightarrow B .$$

What is the number of possible functions f?

Solution 1. The number of possible functions is

$$|B|^{|A|} .$$

If $|B| = 2$ we have $2^{|A|}$. For example, if $|A| = |B| = 2$, we have four functions, namely $f_1(0) = 0$, $f_1(1) = 0$, $f_2(0) = 0$, $f_2(1) = 1$, $f_3(0) = 1$, $f_3(0) = 0$, and $f_4(0) = 1$, $f_4(1) = 1$.

Problem 2. Given a set of seven elements

$$\{ x_1, x_2, x_3, x_4, x_5, x_6, x_7 \} .$$

Is it possible to find 7 subsets each having each $k = 3$ elements and such that any two of these subsets have $\lambda = 1$ element in common?

Solution 2. We apply *binary matrices* whose entries are restricted to the set $\{ 0, 1 \}$. The first row of the 7×7 matrix is

$$(1110000)$$

which refers to the set $\{x_1, x_2, x_3\}$. Taking into account the conditions the next row is

$$(1001100)$$

which refers to the set $\{x_1, x_4, x_5\}$. Obviously the next row is

$$(1000011)$$

with corresponds to the set $\{x_1, x_6, x_7\}$. Next we have

$$(0101010)$$

with the set $\{x_2, x_4, x_6\}$. Proceeding on we find the rows

$$(0100101) \Leftrightarrow \{x_2, x_5, x_7\}$$

$$(0011001) \Leftrightarrow \{x_3, x_4, x_7\}$$

$$(0010110) \Leftrightarrow \{x_3, x_5, x_6\}.$$

Thus we obtain the binary matrix

$$B = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

and consequently the problem admits a solution.

Problem 3. Let A be a finite set. Then the number of distinct elements of A is called the *cardinality* and is denoted by $|A|$ or $\text{card}(A)$. For any finite set A, $|2^A| = 2^{|A|}$; that is the cardinality of the power set of A is 2 raised to a power equal to the cardinality of A. The *power set* of A is the set of all subsets of A including the empty set and the set itself. Prove this statement by induction on the cardinality of A.

Solution 3. The basis step is: let A be a set of cardinality $n = 0$. Then $A = \emptyset$, and $2^{|A|} = 2^0 = 1$. On the other hand, $2^A = \{\emptyset\}$, and $|2^A| = |\{\emptyset\}| = 1$.
The induction hypothesis: let $n > 0$, and suppose that $|2^A| = 2^{|A|}$ provided that $|A| \leq n$.
Induction step: let A be such that $|A| = n + 1$. Since $n > 0$, the set A contains at least one element a. Let $B = A \setminus \{a\}$; then $|B| = n$. By the induction hypothesis, $|2^B| = 2^{|B|} = 2^n$. Now the power set of A can be

divided into two parts, those sets containing the element a and those sets not containing a. The later part is just the set 2^B, and the former part is obtained by introducing a into each member of the set 2^B. Thus

$$2^A = 2^B \cup \{ C \cup \{ a \} : C \in 2^B \}.$$

This division in fact partitions 2^A into two disjoint equinumerous parts, so the cardinality of the whole is twice $2^{|B|}$, which, by the induction hypothesis, is $2 \cdot 2^n = 2^{n+1}$, as was to be shown.

Problem 4. (i) Given a 3×3 two-dimensional array (matrix). Arrange the numbers

```
1 2 3 4 5 6 7 8 9
```

so that all lines (rows and columns) including the diagonals add up to the same total. This is known as a *magic square.*
(ii) Write a C++ program that uses the brute force method by checking all possible permutations of the numbers, i.e., we have

$$9! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot 8 \cdot 9 = 362880.$$

Solution 4. (i) We find eight solutions

```
2 7 6     2 9 4     4 3 8     4 9 2
9 5 1     7 5 3     9 5 1     3 5 7
4 3 8     6 1 8     2 7 6     8 1 6

6 1 8     6 7 2     8 1 6     8 3 4
7 5 3     1 5 9     3 5 7     1 5 9
2 9 4     8 3 4     4 9 2     6 7 2
```

Obviously, four of the solutions are the transpose of the matrix of the other four.
(ii) The function `factorial()` finds the factorial of the number `i`. The function `test()` checks whether the rows, columns and diagonals add up the same number. The C++ implementation is given by

```
// teaser.cpp

#include <iostream>
#include <iomanip>
using namespace std;

const int N = 3;
```

```
const int N2 = 9;

double factorial(int i)
{
  double f = i;
  while(i > 1) f *= (--i);
  return f;
}

int test(int array[N][N])
{
  int sum, sumcol, sumrow, sumdiag1, sumdiag2;
  int i, j;
  for(i=sum=0;i<N;i++)
  sum += array[0][i];
  sumdiag1 = sumdiag2 = 0;
  for(i=0;i<N;i++)
  {
  sumcol = sumrow = 0;
  for(j=0;j<N;j++)
  {
  sumrow += array[i][j];
  sumcol += array[j][i];
  }
  if(sumrow != sum) return 0;
  if(sumcol != sum) return 0;
  sumdiag1 += array[i][i];
  sumdiag2 += array[i][N-1-i];
  }
  if(sumdiag1 != sum) return 0;
  if(sumdiag2 != sum) return 0;
  return 1;
} // end test

void print(int array[N][N],int width)
{
  for(int i=0;i<N;i++)
  {
  for(int j=0;j<N;j++)
  cout << setw(width) << array[i][j];
  cout << endl;
  }
  cout << endl;
} // end print
```

```
void generate(int numbers[N],int width)
{
  static double done = 0;
  static double total = factorial(N2);
  static int solutions = 0;
  static int used[N2];
  static int array[N][N];
  static int d = 0;
  if(d == 0)
  for(int j=0;j<N2;j++) used[j] = 0;

  if((d == N2) && (test(array)))
  {
  cerr << endl << (++solutions) << " solutions found" << endl;
  print(array,width);
  }

  if(d == N2)
  {
  cerr << (++done)/total << "          \r";
  return;
  }

  for(int i=0;i<N2;i++)
  {
  if(!used[i])
  {
  array[d/N][d%N] = numbers[i];
  d++;
  used[i] = 1;
  generate(numbers,width);
  used[i] = 0;
  d--;
  }
  }
} // end generate

int main(void)
{
  int numbers[N2] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  generate(numbers,N);
  return 0;
}
```

Problem 5. Given a finite set with n elements the number of *subsets* is given by 2^n which includes the set itself and the empty set. Write a Java program that finds all subsets of a finite set of strings.

Solution 5. We use recursion.

```
// Subsets.java

public class Subsets
{
  String[] set = null;
  int setLength = 0, increment = 0;

  public Subsets(String[] set)
  {
  this.set = set;
  setLength = set.length;
  // empty set
  System.out.println("" + increment + ":\t");
  increment++;

  // starting for all positions in set
  for(int i=0;i<setLength;i++) { subSet("",set,i); }
  } // end constructor

  public void subSet(String preString,String[] set,int pos)
  {
  String newString = "" + preString + " " + set[pos];
  System.out.println("" + increment + ":\t" + newString);
  increment++;

  for(int i=pos+1;i<setLength;i++) { subSet(newString,set,i); }
  } // end Subsets(String[] set)

  public static void main(String[] args)
  {
  String[] mySet = { "X", "Y", "Z", "A" };
  Subsets set = new Subsets(mySet);
  }
}
```

Problem 6. Assume that N people (counting $1, 2, ..., N$) have decided to elect a leader by arranging themselves in a circle and eliminating every

Mth person around the circle, closing ranks as a person drops out. Write a Java program using a circular linked list that finds the remaining person given N and M. The identity of the elected leader is a function of N and M. This function is called the *Josephus function.*

Solution 6. For example, if we have $N = 9$ and $M = 5$ the people are eliminated in the order

$$5 \quad 1 \quad 7 \quad 4 \quad 3 \quad 6 \quad 9 \quad 2$$

and 8 is the leader chosen.

```
// Josephus.java

class Josephus
{
  static class Node
  {
  int val;
  Node next;
  Node(int v) { val = v; }
  }

  public static void main(String[] args)
  {
  int N = Integer.parseInt(args[0]);
  int M = Integer.parseInt(args[1]);
  Node t = new Node(1);
  Node x = t;
  for(int i=2;i<=N;i++)
  {
  x = (x.next = new Node(i));
  }
  x.next = t;
  while(x != x.next)
  {
  for(int i=1;i<M;i++) { x = x.next; }
  x.next = x.next.next;
  }
  System.out.println("survivor is: " + x.val);
  }
}
```

Problem 7. A person starting at $(0,0)$ moves on a rectangular grid. At each grid point (m,n) $m,n \in \mathbf{Z}$ ($\mathbf{Z}$ denotes the set of all integers) he/she

can move to the right (r), to the left (l), up (u) or down (d) one step. The person notes his path, for example,

$$rullddrrruuullll\,.$$

Given this string write a C++ program that finds the position $(m, n) \in \mathbf{Z} \times \mathbf{Z}$ of the person.

Solution 7. We use the `string` class of C++. We count the number of r's, l's, u's and d's in the string to find the position.

```
// position.cpp

#include <iostream>
#include <string>
using namespace std;

void position(string s,int& m,int& n)
{
  m = n = 0;
  for(int i=0;i<s.length();i++)
  {
  switch(s[i])
  {
  case 'r': m++; break;
  case 'u': n++; break;
  case 'l': m--; break;
  case 'd': n--; break;
  }
  } // end for
}

int main(void)
{
   int m, n;
   position("rullddrrruuullll",m,n);
   cout << "(" << m << "," << n << ")" << endl;
   position("lluurrrrddll",m,n);
   cout << "(" << m << "," << n << ")" << endl;
   position("",m,n);
   cout << "(" << m << "," << n << ")" << endl;
   return 0;
}
```

Problem 8. Let

$$a_0,\ a_1,\ \ldots,\ a_{n-1}, \qquad b_0,\ b_1,\ \ldots,\ b_{m-1}$$

be two strictly increasing sequences of integers. Write a C++ program that finds the number of integers common to both sets; that is, the *cardinality* of the set

$$\{\, a_0,\ a_1,\ \ldots,\ a_{n-1} \,\} \cap \{\, b_0,\ b_1,\ \ldots,\ b_{m-1} \,\}$$

where $\cap$ denotes the *intersection* of the two sets.

Solution 8. We use a `do-while` loop to solve the problem. Obviously the output is `count = 3`.

```
// cardinality.cpp

#include <iostream>
using namespace std;

int main(void)
{
   int a[] = { 2, 3, 5, 9 };
   int n = 4;
   int b[] = { 3, 4, 5, 8, 9, 10 };
   int m = 6;
   int i = 0, j = 0;
   int count = 0;

   do
   {
   if(a[i] == b[j]) { i++; j++; count++; }
   else
   { if(a[i] < b[j]) i++; if(a[i] > b[j]) j++; }
   }
   while((i < n) && (j < m));
   cout << "count = " << count;
   return 0;
}
```

Problem 9. Let n be a positive integer. Let $f(n)$ be the number of ways in which one can cover a 3-by-n rectangle $ABCD$ with *dominoes* (rectangles with side length 1 and 2). If n is odd we find $f(n) = 0$ and if n is even we find

$$f(2n) = \frac{1}{2\sqrt{3}}((\sqrt{3}+1)(2+\sqrt{3})^n + (\sqrt{3}-1)(2-\sqrt{3})^n)\,. \qquad (1)$$

(i) Could a C++ implementation of (1) lead to rounding error?
(ii) Derive a recurrence relation. Implement the recurrence in the C++ program.

Solution 9. (i) The C++ implementation is given by

```
// dominoes.cpp

#include <iostream>
#include <cmath>
using namespace std;

unsigned long f(unsigned long n)
{
   if(n%2) return 0;
   if(n == 2) return 3;
   if(n == 4) return 11;
   return 4*f(n-2) - f(n-4);
}

int main(void)
{
   double t = sqrt(3.0);
   int m = 3;
   double n_ways = ((t+1)*pow(t+2,m) + (t-1)*pow(2-t,m))/(2*t);
   cout << "n_ways = " << n_ways << endl;

   for(unsigned long k=1;k<=10;k++)
   {
   cout << "f(" << k << ") = " << f(k) << endl;
   }
   return 0;
}
```

It does not lead to rounding errors.
(ii) The linear recurrence is given by

$$f(2n+2) = 4f(2n) - f(2n-2)$$

where $n = 1, 2, \ldots$ and the "initial values" $f(2) = 3$ and $f(4) = 11$. The solution of this recurrence with the initial values yields (1).

Problem 10. The number of ways of selecting k objects from n distinct objects is given by

$$\binom{n}{k} := \frac{n!}{k!(n-k)!}.$$

We have

$$\binom{n}{0} = \binom{n}{n} = 1\,.$$

Using this base case, and the relation

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

implement a recursive function to calculate $\binom{n}{k}$. Use a C++ template function, applying the function using the basic data type `unsigned long` and the abstract data type `Verylong` of SymbolicC++. Alternatively, apply the function using the data types `long` and `BigInteger` in Java. Also give an iterative implementation. Compare the execution times (time complexity) of the two implementations. Explain any differences in the execution times.

Solution 10. The recursive method uses only addition which is faster than the multiplication which is used by the iterative method. On the other hand, the recursive method uses substantially more additions than the multiplications of the iterative method as well as function calls which have relatively high overhead. Thus, the recursive method is simpler but slower.

```
// comb.cpp

#include <iostream>
#include <ctime>
#include "Verylong.h"
using namespace std;

template <class T> T comb_r(T n,T k)
{
  if(k==T(0)||k==n) return T(1);
  return comb_r(n-T(1),k)+comb_r(n-T(1),k-T(1));
}

template <class T> T fact(T n)
{
  T f(1);
  if(n<=T(0)) return T(1);
  for(T i(1);i<=n;i++) f *= i;
  return f;
}

template <class T> T comb_i(T n,T k)
```

```
{ return fact(n)/(fact(k)*fact(n-k)); }

int main(void)
{
  time_t start;
  start = time(NULL);
  cout << "comb_r(30,7) = " << comb_r(30,7);
  cout << " - time " << time(NULL)-start << "s" << endl;
  start = time(NULL);
  cout << "Verylong comb_r(30,7) = "
       << comb_r(Verylong(30),Verylong(7));
  cout << " - time " << time(NULL)-start << "s" << endl;
  return 0;
}
```

A typical output is

```
comb_r(30,7) = 2035800 - time 0s
Verylong comb_r(30,7) = 2035800 - time 22s
```

Problem 11. The *compositional complexity* K for *nucleic acid sequences* and *protein sequences* of length L is given by

$$K := \frac{1}{L} \log_N \left(\frac{L!}{\prod_{all\, j} n_j!} \right)$$

where N is 4 for nucleic acid sequences and 20 for protein sequences, and n_j are the numbers of each residue in the sequence. K will vary from 0 for very low complexity to 1 for high complexity.
(i) Find K for the nucleic acid sequence $GGGG$.
(ii) Find K for the nucleic acid sequence $CTGA$.
(iii) Given a nucleic acid sequence write a Java program that finds K.

Solution 11. (i) We have $N = 4$, $L = 4$ and $n_G = 4$, $n_C = 0$, $n_T = 0$, $n_A = 0$. Thus

$$\prod_{all\, j} n_j! = 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0! \cdot 0! \cdot 0! = 24 .$$

Therefore

$$K = \frac{1}{4} \log_4 \left(\frac{24}{24} \right) = 0 .$$

(ii) For the sequence $CTGA$ we have $N = 4$, $L = 4$, and $n_G = 1$, $n_C = 1$, $n_T = 1$, $n_A = 1$. Thus

$$\prod_{all\, j} n_j! = 1! \cdot 1! \cdot 1! \cdot 1! = 1 .$$

Therefore

$$K = \frac{1}{4}\log_4\left(\frac{24}{1}\right) = 0.573\,.$$

(iii) The following Java program calculates K for $N = 4$. The method `byte[] getBytes()` in the class `String` converts this `String` into bytes according to the platforms default character encoding, storing the result into a new byte array.

```
// Complexity.java

import java.lang.*;

public class Complexity
{
  public static double K(String s)
  {
  int i, j, L, N = 4;
  double k = 1.0;
  int[] n = new int[N];
  L = s.length();  // length of string
  for(i=n[0]=0;i<L;i++) if(s.getBytes()[i] == 'G') n[0]++;
  for(i=n[1]=0;i<L;i++) if(s.getBytes()[i] == 'C') n[1]++;
  for(i=n[2]=0;i<L;i++) if(s.getBytes()[i] == 'T') n[2]++;
  for(i=n[3]=0;i<L;i++) if(s.getBytes()[i] == 'A') n[3]++;
  for(i=2;i<=L;i++) k *= i;
  for(i=0;i<N;i++)
  for(j=2;j<=n[i];j++) k /= j;
  k = Math.log(k)/Math.log(N);
  k /= L;
  return k;
  }

  public static void main(String argv[])
  {
  System.out.print("Complexity of GGGG K=");
  System.out.println(K("GGGG"));
  System.out.print("Complexity of CTGA K=");
  System.out.println(K("CTGA"));
  }
}
```

Problem 12. An ancestral binary tree has one original ancestor (root) and each person (node) in the tree has 0 or two offspring (a first and

second child denoted by the left and right branch, respectively). Find a recurrence relation and associated generating function for the number of different ancestral trees with n people.

Solution 12. Any valid binary tree has an odd number of nodes. Let c_n be the number of binary trees with $2n+1$ nodes. The first few terms are $c_0 = 1$, $c_1 = 1$, $c_2 = 2$, $c_3 = 5$, $c_4 = 14$. We obtain a recurrence relation for c_n by partitioning the trees according to the number of nodes on their left and right branches from the root node. If there are $2j+1$ nodes on the left and $2k+1$ nodes on the right, then the total number of such arrangements is $c_j c_k$ since we can attach any valid binary tree to each child of the root. Thus

$$c_n = \sum_{j=0}^{n-1} c_j c_{n-j-1} \,.$$

Let $C(x) = \sum_{n=0}^{\infty} c_n x^n$ be the generating function. Then

$$C^2(x) = c_0^2 + (c_1 c_0 + c_0 c_1)x + \cdots + \left(\sum_{j=0}^{n-1} c_j c_{n-j-1} \right) x^{n-1} + \cdots \,.$$

Using the recurrence, we have $xC^2(x) = C(x) - c_0 = C(x) - 1$. Solving this quadratic equation and taking into account that the coefficients of $C(x)$ are non-negative we find

$$C(x) = \frac{1 - \sqrt{1-4x}}{2x} \,.$$

Using the expansion

$$(1-4x)^{1/2} = 1 - 2\sum_{n=1}^{\infty} \frac{1}{n} \binom{2n-2}{n-1} x^n$$

we arrive at

$$C(x) = \sum_{n=0}^{\infty} \frac{1}{n+1} \binom{2n}{n} x^n \,.$$

Chapter 5

Matrix Calculus

Problem 1. Everyone knows that multiplying two arbitrary $n \times n$ matrices requires n^3 multiplications, at least if we do it using the standard formula. Let A and B be 2×2 matrices. Then $C = AB$ is given by

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} \\ c_{22} &= a_{21}b_{12} + a_{22}b_{22} \end{aligned}$$

which gives 8 multiplications and 4 additions. Strassen discovered a surprising base algorithm to compute the product of 2×2 matrices with only seven multiplications. Find out how the number of multiplications for 2×2 matrices can be reduced to 7.

Solution 1. We have the seven products

$$\begin{aligned} p_1 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\ p_2 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ p_3 &= (a_{11} - a_{21})(b_{11} + b_{12}) \\ p_4 &= (a_{11} + a_{12})b_{22} \\ p_5 &= a_{11}(b_{12} - b_{22}) \\ p_6 &= a_{22}(b_{21} - b_{11}) \\ p_7 &= (a_{21} + a_{22})b_{11}\,. \end{aligned}$$

We obtain the entries of the product matrix C by forming the sums

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} = p_1 + p_2 - p_4 + p_6 \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} = p_4 + p_5 \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} = p_6 + p_7 \\ c_{22} &= a_{21}b_{12} + a_{22}b_{22} = p_2 - p_3 + p_5 - p_7 \,. \end{aligned}$$

This result is of no practical importance for matrices of real or complex numbers because calculations are usually carried out by computers multiplying almost as fast as adding. Therefore, it makes no sense to replace 8 multiplications and 4 additions by 7 multiplications and 18 additions.

Problem 2. The *Strassen's algorithm* described in problem 1 can be recursively extended to larger matrices via a Kronecker product construction. The construction is very simple: large matrices are broken down recursively by partitioning the matrices into quarts, sixteenths, etc. Apply Strassen's algorithm to the matrices

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{pmatrix}, \quad B = \begin{pmatrix} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{pmatrix}.$$

Solution 2. We have the submatrices

$$A_{11} = \begin{pmatrix} 1 & 2 \\ 5 & 6 \end{pmatrix}, \quad A_{12} = \begin{pmatrix} 3 & 4 \\ 7 & 8 \end{pmatrix}, \quad A_{21} = \begin{pmatrix} 9 & 1 \\ 4 & 5 \end{pmatrix}, \quad A_{22} = \begin{pmatrix} 2 & 3 \\ 6 & 7 \end{pmatrix}$$

$$B_{11} = \begin{pmatrix} 8 & 9 \\ 3 & 4 \end{pmatrix}, \quad B_{12} = \begin{pmatrix} 1 & 2 \\ 5 & 6 \end{pmatrix}, \quad B_{21} = \begin{pmatrix} 7 & 8 \\ 2 & 3 \end{pmatrix}, \quad B_{22} = \begin{pmatrix} 9 & 1 \\ 4 & 5 \end{pmatrix}.$$

Then

$$P_1 = (A_{12} - A_{22})(B_{21} + B_{22}) = \begin{pmatrix} 22 & 17 \\ 22 & 17 \end{pmatrix}$$

$$P_2 = (A_{11} + A_{22})(B_{11} + B_{22}) = \begin{pmatrix} 86 & 75 \\ 278 & 227 \end{pmatrix}$$

$$P_3 = (A_{11} - A_{21})(B_{11} + B_{12}) = \begin{pmatrix} -64 & -78 \\ 17 & 21 \end{pmatrix}$$

$$P_4 = (A_{11} + A_{12})B_{22} = \begin{pmatrix} 60 & 34 \\ 164 & 82 \end{pmatrix}$$

$$P_5 = A_{11}(B_{12} - B_{22}) = \begin{pmatrix} -6 & 3 \\ -34 & 11 \end{pmatrix}$$

$$P_6 = A_{22}(B_{21} - B_{11}) = \begin{pmatrix} -5 & -5 \\ -13 & -13 \end{pmatrix}$$

$$P_7 = (A_{21} + A_{22})B_{11} = \begin{pmatrix} 100 & 115 \\ 116 & 138 \end{pmatrix}.$$

Thus

$$C_{11} = P_1 + P_2 - P_4 + P_6 = \begin{pmatrix} 43 & 53 \\ 123 & 149 \end{pmatrix}$$

$$C_{12} = P_4 + P_5 = \begin{pmatrix} 54 & 37 \\ 130 & 93 \end{pmatrix}$$

$$C_{21} = P_6 + P_7 = \begin{pmatrix} 95 & 110 \\ 103 & 125 \end{pmatrix}$$

$$C_{22} = P_2 - P_3 + P_5 - P_7 = \begin{pmatrix} 44 & 41 \\ 111 & 79 \end{pmatrix}.$$

Problem 3. Given a sequence $r(0)$, $r(1), \ldots, r(p)$. Using this sequence we can form the $p \times p$ matrix

$$R := \begin{pmatrix} r(0) & r(1) & r(2) & \ldots & r(p-1) \\ r(1) & r(0) & r(1) & \ldots & r(p-2) \\ r(2) & r(1) & r(0) & \ldots & r(p-3) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ r(p-1) & r(p-2) & r(p-3) & \ldots & r(0) \end{pmatrix}.$$

The matrix is symmetric and all the elements along the diagonal are equal. We assume that $r(0) \neq 0$ and R is invertible. Let

$$A := \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_p \end{pmatrix}, \qquad P := \begin{pmatrix} r(1) \\ r(2) \\ \vdots \\ r(p) \end{pmatrix}.$$

Then the linear equation $RA = P$ can be solved using *Durbin's algorithm* which is a recursive procedure. The algorithm is as follows:

$$E(0) := r(0), \qquad k_1 = r(1)/E(0)$$

$$k_i = \frac{1}{E(i-1)} \left[r(i) - \sum_{j=1}^{i-1} a_j(i-1) r(i-j) \right], \quad i = 2, 3, \ldots, p$$

$$a_i(i) = k_i, \quad i = 1, 2, \ldots, p$$

$$a_j(i) = a_j(i-1) - k_i a_{i-j}(i-1), \quad \begin{cases} i = 2, 3, \ldots, p \\ j = 1, 2, \ldots, i-1 \end{cases}$$

$$E(i) = (1 - k_i^2) E(i-1), \quad i = 1, 2, \ldots, p.$$

After solving these equations recursively, the solution of the linear equation is given by

$$a_j = a_j(p), \quad j = 1, 2, \ldots, p\,.$$

Write a C++ program that implements this algorithm. Test your program with the sequence

$$r(0) = 1, \quad r(1) = 0.5, \quad r(2) = 0.2\,.$$

This problem plays a role in *linear predictive coding.*

Solution 3.

```
// durbin.cpp

#include <iostream>
using namespace std;

double E(int,double*);
double k(int,double*);
double a(int,int,double*);

double E(int i,double* r)
{
  if(i==0) return r[0];
  double ki = k(i,r);
  return (1-ki*ki)*E(i-1,r);
}

double k(int i,double* r)
{
  double ki = r[i];
  for(int j=1;j<i;j++)
  ki -= a(j,i-1,r)*r[i-j];
  ki /= E(i-1,r);
  return ki;
}

double a(int j,int i,double* r)
{
  if(i==j) return k(i,r);
  return a(j,i-1,r)-k(i,r)*a(i-j,i-1,r);
}

int main(void)
```

```
{
  int p = 2;
  double *r = new double[p+1];
  r[0]=1.0; r[1]=0.5; r[2]=0.2;

  for(int i=1;i<=p;i++)
  cout<< "a(" << i << ") = " << a(i,p,r) << endl;
  delete[] r;
  return 0;
}
```

The output is

```
a(1) = 0.533333
a(2) = -0.0666667
```

Problem 4. Give the matrices to perform any one of the following operations

- Rotation around the z-axis
- Scaling
- Shearing of the x by the z-coordinate
- Translation

in any order using only matrix multiplications. Use the homogeneous form of a point: $(x, y, z, 1)^T$ and 4×4 matrices to solve the problem.

Solution 4. The *rotation matrix* for rotation around the z-axis by an angle θ is given by

$$R_z = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Similar matrices are available for the other axes.

Scaling is achieved by

$$S = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Shearing the x-coordinate by the z-coordinate is given by

$$H_{xz} = \begin{pmatrix} 1 & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Finally *translation* is represented by

$$T = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Now any sequence of operations can be combined by multiplying the corresponding matrices together. This matrix can then be applied to a point to transform the point.

Problem 5. Let A be an arbitrary $m \times n$ matrix over $\mathbf{R}$, i.e., $A \in \mathbf{R}^{m \times n}$. Then A can be written as

$$A = U \Sigma V^T$$

where U is an $m \times m$ orthogonal matrix, V is an $n \times n$ orthogonal matrix, Σ is an $m \times n$ diagonal matrix with nonnegative entries and ^{T} denotes the transpose. This is called the *singular value decomposition.* An algorithm to find the singular value decomposition is given as follows.

1) Find the eigenvalues λ_j $(j = 1, 2, \ldots, n)$ of the $n \times n$ matrix $A^T A$. Arrange the eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ in descending order.

2) Find the number of nonzero eigenvalues of the matrix $A^T A$. We call this number r.

3) Find then orthonormal eigenvectors $\mathbf{v}_j$ $(j = 1, 2, \ldots, n)$ of the matrix $A^T A$ corresponding to the obtained eigenvalues, and arrange them in the same order to form the column-vectors of the $n \times n$ matrix V.

4) Form an $m \times n$ diagonal matrix Σ placing on the leading diagonal of it the square root $\sigma_j := \sqrt{\lambda_j}$ of $p = \min(m, n)$ first eigenvalues of the matrix $A^T A$ found in 1) in descending order.

5) Find the first r column vectors of the $m \times m$ matrix U

$$\mathbf{u}_j = \frac{1}{\sigma_j} A \mathbf{v}_j, \quad j = 1, 2, \ldots, r.$$

6) Add to the matrix U the rest of $m - r$ vectors using the *Gram-Schmidt orthogonalization process.*

Apply the algorithm to the matrix

$$A = \begin{pmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

Solution 5. We find

$$A^T A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}.$$

The eigenvalues are (arranged in descending order) $\lambda_1 = 3$ and $\lambda_2 = 1$.

2) The number of nonzero eigenvalues are $r = 2$.

3) The orthonormal eigenvectors of the matrix $A^T A$ corresponding to the eigenvalues λ_1 and λ_2 are given by

$$\mathbf{v}_1 = \begin{pmatrix} \sqrt{2}/2 \\ \sqrt{2}/2 \end{pmatrix}, \quad \mathbf{v}_2 = \begin{pmatrix} \sqrt{2}/2 \\ -\sqrt{2}/2 \end{pmatrix}.$$

Thus we obtain the 2×2 matrix V (V^T follows by taking the transpose)

$$V = (\mathbf{v}_1 \ \mathbf{v}_2) = \begin{pmatrix} \sqrt{2}/2 & \sqrt{2}/2 \\ \sqrt{2}/2 & -\sqrt{2}/2 \end{pmatrix}.$$

4) From the eigenvalues we find the singular matrix

$$\Sigma = \begin{pmatrix} \sqrt{3} & 0 \\ 0 & \sqrt{1} \\ 0 & 0 \end{pmatrix}$$

on the leading diagonal of which are the square roots of the eigenvalues of the matrix $A^T A$ (in descending order) and the rest of the entries of the matrix Σ are zeros.

5) Next we find two column vectors of the 3×3 matrix U using the equation given above

$$\mathbf{u}_1 = \frac{1}{\sigma_1} A\mathbf{v}_1 = \begin{pmatrix} \sqrt{6}/3 \\ \sqrt{6}/6 \\ \sqrt{6}/6 \end{pmatrix}, \quad \mathbf{u}_2 = \frac{1}{\sigma_2} A\mathbf{v}_2 = \begin{pmatrix} 0 \\ -\sqrt{2}/2 \\ \sqrt{2}/2 \end{pmatrix}.$$

6) To find the vector $\mathbf{u}_3$ we apply the Gram-Schmidt process. The vector $\mathbf{u}_3$ is perpendicular to $\mathbf{u}_1$ and $\mathbf{u}_2$. We have

$$\mathbf{u}_3 = \mathbf{e}_1 - (\mathbf{u}_1^T\mathbf{e}_1)\mathbf{u}_1 - (\mathbf{u}_2^T\mathbf{e}_1)\mathbf{u}_2 = (1/3 \ \ -1/3 \ \ -1/3)^T .$$

Normalizing the vector we obtain

$$\mathbf{u}_3 = \begin{pmatrix} \sqrt{3}/3 \\ -\sqrt{3}/3 \\ -\sqrt{3}/3 \end{pmatrix} .$$

It follows that

$$U = (\mathbf{u}_1 \ \mathbf{u}_2 \ \mathbf{u}_3) = \begin{pmatrix} \sqrt{6}/3 & 0 & \sqrt{3}/3 \\ \sqrt{6}/6 & \sqrt{2}/2 & -\sqrt{3}/3 \\ \sqrt{6}/6 & -\sqrt{2}/2 & -\sqrt{3}/3 \end{pmatrix} .$$

Thus we have found the singular value decomposition of the matrix A.

Remark. We have

$$A\mathbf{v}_j = \sigma_j\mathbf{u}_j, \qquad A^T\mathbf{u}_j = \sigma_j\mathbf{v}_j$$

and therefore

$$A^TA\mathbf{v}_j = \sigma_j^2\mathbf{v}_j, \qquad AA^T\mathbf{u}_j = \sigma_j^2\mathbf{u}_j .$$

Problem 6. Let A be an $n \times n$ matrix. Assume that the inverse matrix of A exists. The inverse matrix can be calculated as follows (*Csanky's algorithm*). Let

$$p(x) = \det(xI_n - A) \tag{1}$$

where I_n is the $n \times n$ unit matrix. The roots are, by definition, the eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ of A. We write

$$p(x) = x^n + c_1x^{n-1} + \cdots + c_{n-1}x + c_n \tag{2}$$

where

$$c_n = (-1)^n \det A .$$

Since A is nonsingular we have $c_n \neq 0$ and vice versa. The *Caley-Hamilton theorem* states that

$$p(A) = A^n + c_1A^{n-1} + \cdots + c_{n-1}A + c_nI_n = 0_n . \tag{3}$$

Multiplying this equation with A^{-1} we obtain

$$A^{-1} = \frac{1}{-c_n}(A^{n-1} + c_1A^{n-2} + \cdots + c_{n-1}I_n) . \tag{4}$$

If we have the coefficients c_j we can calculate the inverse matrix A. Let

$$s_k := \sum_{j=1}^{n} \lambda_j^k .$$

Then the s_j and c_j satisfy the following $n \times n$ lower triangular system of linear equations

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ s_1 & 2 & 0 & \dots & 0 \\ s_2 & s_1 & 3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ s_{n-1} & s_{n-2} & \dots & s_1 & n \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} -s_1 \\ -s_2 \\ -s_3 \\ \vdots \\ -s_n \end{pmatrix} .$$

Since

$$\operatorname{tr}(A^k) = \lambda_1^k + \lambda_2^k + \cdots + \lambda_n^k = s_k$$

we find s_k for $k = 1, 2, \dots, n$. Thus we can solve the linear equation for c_j. Finally using (4) we obtain the inverse matrix of A. Apply Csanky's algorithm to the 4×4 matrix

$$U = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} .$$

Solution 6. Since

$$U^2 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}, \quad U^3 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

and $U^4 = I_4$ we find

$$\operatorname{tr}U = 0 = s_1, \quad \operatorname{tr}U^2 = 0 = s_2, \quad \operatorname{tr}U^3 = 0 = s_3, \quad \operatorname{tr}U^4 = 4 = s_4 .$$

Thus we obtain the linear equation

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ -4 \end{pmatrix}$$

with the solution

$$c_1 = 0, \quad c_2 = 0, \quad c_3 = 0 \quad c_4 = -1 .$$

Thus the inverse matrix is given by

$$U^{-1} = U^3 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Problem 7. A distance between two $m \times n$ matrices A and B over the real numbers can be defined as

$$\|A - B\| := \sqrt{\mathrm{tr}((A-B)(A-B)^T)}$$

where ^{T} denotes the transpose. Find an efficient way to calculate the distance. The expression

$$\|A\| := \sqrt{\mathrm{tr}(AA^T)}$$

is called the *Hilbert-Schmidt norm* of A.

Solution 7. Since we take the trace we only have to calculate the diagonal elements of the matrix

$$(A-B)(A-B)^T.$$

Problem 8. (i) To test programs for the eigenvalue problem of real $n \times n$ symmetric matrices it is useful to have a set of easily generated matrices whose eigenvalues are known. Consider the following, nine real symmetric matrices and discuss the problem with the eigenvalue calculations.
(ii) What other tests on the eigenvalues can be performed to test the accuracy?

(1) The Hilbert segment of order n matrix is given by

$$a_{ij} = \frac{1}{i+j-1}.$$

For $n = 4$ we have

$$A = \begin{pmatrix} 1 & 1/2 & 1/3 & 1/4 \\ 1/2 & 1/3 & 1/4 & 1/5 \\ 1/3 & 1/4 & 1/5 & 1/6 \\ 1/4 & 1/5 & 1/6 & 1/7 \end{pmatrix}.$$

(2) The Ding Dong matrix

$$a_{ij} = \frac{1}{2}\frac{1}{n-i-j+3/2}.$$

For $n = 4$ we have

$$A = \begin{pmatrix} 1/7 & 1/5 & 1/3 & 1 \\ 1/5 & 1/3 & 1 & -1 \\ 1/3 & 1 & -1 & -1/3 \\ 1 & -1 & -1/3 & -1/5 \end{pmatrix}.$$

(3) The Moler matrix

$$A_{ii} = i$$

and

$$A_{ij} = \min(i, j) - 2, \qquad i \neq j\,.$$

For $n = 4$ we have

$$A = \begin{pmatrix} 1 & -1 & -1 & -1 \\ -1 & 2 & 0 & 0 \\ -1 & 0 & 3 & 1 \\ -1 & 0 & 1 & 4 \end{pmatrix}.$$

(4) The Frank matrix

$$A_{ij} = \min(i, j)\,.$$

For $n = 4$ we have

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 \\ 1 & 2 & 3 & 3 \\ 1 & 2 & 3 & 4 \end{pmatrix}.$$

(5) The bordered matrix

$$\begin{aligned} &A_{ii} = 1 \\ &A_{in} = A_{ni} = 2^{1-i} \quad \text{for} \quad i \neq n \\ &A_{ij} = 0 \quad \text{otherwise}\,. \end{aligned}$$

For $n = 4$ we have

$$A = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1/2 \\ 0 & 0 & 1 & 1/4 \\ 1 & 1/2 & 1/4 & 1 \end{pmatrix}.$$

(6) The diagonal matrix

$$A_{ii} = i$$

and

$$A_{ij} = 0 \qquad \text{for } i \neq j\,.$$

(7) The Wilkinson $W+$ matrix (n odd)

$$\begin{aligned} A_{ii} &= \lfloor n/2 \rfloor + 1 - \min(i, n-i+1) \quad \text{for} \quad i = 1, 2, \ldots, n \\ A_{i,i+1} &= A_{i+1,i} = 1 \quad \text{for} \quad i = 1, 2, \ldots, n-1 \\ A_{ij} &= 0 \quad \text{for} \quad |j-i| > 1 \end{aligned}$$

where $\lfloor b \rfloor$ is the largest integer less than or equal to b. For $n = 5$ we have

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 2 \end{pmatrix}.$$

(8) The Wilkinson $W-$ matrix (n odd)

$$\begin{aligned} A_{ii} &= \lfloor n/2 \rfloor + 1 - i \quad \text{for} \quad i = 1, 2, \ldots, n \\ A_{i,i+1} &= A_{i+1,i} = 1 \quad \text{for} \quad i = 1, 2, \ldots, n-1 \\ A_{ij} &= 0 \qquad \text{for} \quad |j-i| > 1\,. \end{aligned}$$

For $n = 5$ we have

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & -1 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{pmatrix}.$$

(9) Matrix with all ones

$$a_{ij} = 1 \quad \text{for all} \quad i, j\,.$$

Solution 8. Since the matrices are real and symmetric all the eigenvalues are real.

(1) This matrix is notorious for its logarithmically distributed eigenvalues. While it can be shown in theory to be positive definite, in practice it is so ill conditioned that most eigenvalue algorithms fail. The eigenvalues for the 4×4 case are: `0.0001`, `0.0067`, `0.1691`, `1.5002`.

(2) The Ding Dong matrix has few trailing zeros in any elements, so is always represented inexactly in the machine. However, it is very stable under inversion by elimination methods. Its eigenvalues have the property of clustering near $\pm\pi/2$. For the case $n = 4$ we find `-1.5707`, `-1.4811`,

0.7608, 1.5672.

(3) This matrix has a very simple Choleski decomposition. Thus it is positive definite. Nevertheless, it has one small eigenvalue and often upsets elimination methods for solving linear equation systems. The eigenvalues for the case $n = 4$ are 2.2974, 2.5477, 0.0334, 5.1215.

(4) This is a well behaved matrix. The eigenvalues for the case $n = 4$ are 0.2831, 0.4260, 1.0000, 8.2909.

(5) This matrix has $(n - 2)$ eigenvalues at 1. The eigenvalues for the case $n = 4$ are 1.0000, 1.0000, 2.1456, -0.1456.

(6) This matrix has the eigenvalues $1, 2, \ldots, n$. However one finds programs that fail to run correctly for this case.

(7) The tridiagonal matrix has several pairs of close eigenvalues despite the fact that no superdiagonal element is small. The separation between the two largest eigenvalues is of the order of $(n!)^{-2}$ so that the power method will be unable to separate them unless n is very small.

(8) For odd order this matrix has eigenvalues which are pairs of equal magnitude but opposite sign.

(9) This matrix is singular. The rank is one. Thus only one eigenvalue, namely n is nonzero, i.e., $(n - 1)$ eigenvalues are zero.

(ii) The *trace* of an $n \times n$ matrix is the sum of the eigenvalues, i.e.,

$$\mathrm{tr}(A) = \sum_{j=1}^{n} \lambda_j \,.$$

The *determinant* of an $n \times n$ matrix is the product of the eigenvalues

$$\det(A) = \prod_{j=1}^{n} \lambda_j \,.$$

Problem 9. To find the eigenvalues and eigenvectors of a symmetrical matrix over $\mathbf{R}$ one often uses the *Jacobi method.* It consists of a sequence of orthogonal similarity transformations of the form

$$A \to Q_1^{-1} A Q_1 \to Q_2^{-1} Q_1^{-1} A Q_1 Q_2 \to \ldots$$

Let $1 \leq p < r \leq n$. The basic Jacobi rotation matrix Q is a matrix with

$$q_{pp} = q_{rr} = \cos(\theta), \quad q_{ii} = 1 \text{ if } i \neq p, r$$
$$q_{pr} = -q_{rp} = -\sin(\theta), \quad q_{ip} = q_{pi} = q_{ir} = q_{ri} = 0 \text{ if } i \neq p, r$$
$$q_{ij} = 0 \text{ if } i \neq p, r \text{ and } j \neq p, r\,.$$

The idea of the Jacobi method is to try to zero the off-diagonal elements by a series of plane rotations. Write a C++ program that implements this method.

Solution 9. In the program we consider the case $n = 4$. However, it can easily be modified to other orders.

```
// eigenvalues.cpp

#include <iostream>
#include <cmath>
using namespace std;

int main(void)
{
  const int M = 4, np = 4, n = 4;
  int j, iq, ip, i, nrot;
  double tresh, theta, tau, t, sm, s, h, g, c;

  double A[M][M], v[M][M], d[M], b[M], z[M];
  A[0][0] = 1.0; A[0][1] = 1.0/2.0;
  A[0][2] = 1.0/3.0; A[0][3] = 1.0/4.0;
  A[1][0] = 1.0/2.0; A[1][1] = 1.0/3.0;
  A[1][2] = 1.0/4.0; A[1][3] = 1.0/5.0;
  A[2][0] = 1.0/3.0; A[2][1] = 1.0/4.0;
  A[2][2] = 1.0/5.0; A[2][3] = 1.0/6.0;
  A[3][0] = 1.0/4.0; A[3][1] = 1.0/5.0;
  A[3][2] = 1.0/6.0; A[3][3] = 1.0/7.0;

  for(ip=0;ip<n;ip++)
  {
  for(iq=0;iq<n;iq++) { v[ip][iq] = 0.0; }
  v[ip][ip] = 1.0;
  }

  for(ip=0;ip<n;ip++)
  {
  b[ip] = A[ip][ip];
```

```
d[ip] = b[ip];
z[ip] = 0.0;
}
nrot = 0;

for(i=0;i<100;i++)
{
sm = 0.0;
for(ip=0;ip<(n-1);ip++)
{
for(iq=ip+1;iq<n;iq++)
{ sm += fabs(A[ip][iq]); }
}
if(sm == 0.0) goto LAB;

if(i < 6) tresh = 0.2*sm/pow(n,2.0);
else tresh = 0.0;
for(ip=0;ip<(n-1);ip++)
{
for(iq=ip+1;iq<n;iq++)
{
g = 100.0*fabs(A[ip][iq]);
if((i > 6) && ((fabs(d[ip]) + g) == fabs(d[ip]))
  && ((fabs(d[iq])+g) == fabs(d[iq])))
A[ip][iq] = 0.0;
else if(fabs(A[ip][iq]) > tresh)
{
h = d[iq] - d[ip];
if((fabs(h) + g) == fabs(h)) { t = A[ip][iq]/h; }
else
{
theta = 0.5*h/A[ip][iq];
t = 1.0/(fabs(theta) + sqrt(1.0 + pow(theta,2.0)));
if(theta < 0.0) t = - t;
}
c = 1.0/sqrt(1.0 + pow(t,2.0));
s = t*c;
tau = s/(1.0 + c);
h = t*A[ip][iq];
z[ip] = z[ip] - h; z[iq] = z[iq] + h;
d[ip] = d[ip] - h; d[iq] = d[iq] + h;
A[ip][iq] = 0.0;

for(j=0;j<ip-1;j++)
```

```
  {
  g = A[j][ip]; h = A[j][iq];
  A[j][ip] = g - s*(h+g*tau); A[j][iq] = h + s*(g-h*tau);
  }

  for(j=ip+1;j<iq-1;j++)
  {
  g = A[ip][j]; h = A[j][iq];
  A[ip][j] = g - s*(h+g*tau); A[j][iq] = h + s*(g-h*tau);
  }
  for(j=iq+1;j<n;j++)
  {
  g = A[ip][j]; h = A[iq][j];
  A[ip][j] = g - s*(h+g*tau); A[iq][j] = h + s*(g-h*tau);
  }
  for(j=0;j<n;j++)
  {
  g = v[j][ip]; h = v[j][iq];
  v[j][ip] = g - s*(h+g*tau); v[j][iq] = h + s*(g-h*tau);
  }
  nrot++;
  }
  }
  }
  for(ip=0;ip<n;ip++)
  {
  b[ip] = b[ip] + z[ip];
  d[ip] = b[ip];
  z[ip] = 0.0;
  }
  }
  LAB:
  for(i=0;i<np;i++)
  cout << "i = " << i << " d[" << i << "] = " << d[i]
       << endl;
  return 0;
}
```

Problem 10. Given an $n \times n$ positive semi-definite matrix A. We want to find the eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ of this matrix. Can the search of the eigenvalues be restricted to a certain range?

Solution 10. A positive-semidefinite matrix is hermitian. Thus the eigenvalues are real. Since A is positive-semidefinite we have $\lambda_{min} \geq 0$. Since

$$\mathrm{tr}A = \sum_{j=1}^{n} \lambda_j$$

we have $\lambda_{max} \leq \mathrm{tr}A$. Thus the eigenvalues are restricted to the interval $[0, \mathrm{tr}A]$.

Problem 11. Let A be an $n \times n$ matrix over $\mathbf{R}$. Then a given vector norm indices a matrix norm through the definition

$$\|A\| := \max_{\|\mathbf{x}\|=1} \|A\mathbf{x}\|$$

where $\mathbf{x} \in \mathbf{R}^n$. We consider the Euclidean norm for the vector norm in the following. Thus $\|A\|$ can be calculated using the *Lagrange multiplier method* with the constraint $\|\mathbf{x}\| = 1$. The matrix norm given above of the matrix A is the square root of the principle component for the matrix $A^T A$, where ^{T} denotes transpose. Thus it is equivalent to the spectral norm

$$\|A\| := \lambda_{max}^{1/2}$$

where λ_{max} is the largest eigenvalue of $A^T A$.
(i) Which of the two methods is faster in calculating the norm of A?
(ii) Apply the two methods to the matrix

$$A = \begin{pmatrix} 1 & 1 \\ 3 & 3 \end{pmatrix}.$$

Solution 11. (i) In the first method we have to take into account the constraint $\|\mathbf{x}\| = 1$. We can use the Lagrange multiplier method for the function

$$f(\mathbf{x}) = \mathbf{x}^T A^T A\mathbf{x} + \mu\|\mathbf{x}\|^2$$

where μ is the Lagrange multiplier. After differentiation of the function with respect to $x_1, x_2, \ldots, x_n$ and setting the results equal to zero we have to solve a system of $n+1$ equations with respect to $x_1, x_2, \ldots, x_n$ and μ, where the $n+1$ equation is $\|\mathbf{x}\|^2 = 1$. In the second method we have to calculate the transpose of A. Then we have to do matrix multiplication. Finally for the matrix $A^T A$ we have to find the largest eigenvalue. Thus, *it seems that the second method is faster to calculate the norm.*
(ii) We apply the Lagrange multiplier method. Since

$$\|A\mathbf{x}\| = (A\mathbf{x})^T A\mathbf{x} = \mathbf{x}^T A^T A\mathbf{x}$$

we find

$$\|A\mathbf{x}\|^2 = 10x_1^2 + 20x_1x_2 + 10x_2^2 .$$

From the constraint $\|\mathbf{x}\| = 1$ we obtain

$$x_1^2 + x_2^2 = 1 .$$

Differentiating the function

$$f(\mathbf{x}) = 10x_1^2 + 20x_1x_2 + 10x_2^2 + \mu(x_1^2 + x_2^2)$$

with respect to x_1 and x_2 where μ is the Lagrange multiplier and setting the derivative equal to zero yields

$$\frac{\partial f}{\partial x_1} = 20x_1 + 20x_2 + 2\mu x_1 = 0$$

$$\frac{\partial f}{\partial x_2} = 20x_1 + 20x_2 + 2\mu x_2 = 0 .$$

Thus, we find that $x_1 = x_2$ and

$$(x_1, x_2) = (1/\sqrt{2}, 1/\sqrt{2}), \qquad (x_1, x_2) = (-1/\sqrt{2}, -1/\sqrt{2}) .$$

It follows that $\|A\|^2 = 20$. For the second method we have

$$A^T A = \begin{pmatrix} 10 & 10 \\ 10 & 10 \end{pmatrix}$$

with the eigenvalues 20 and 0. Thus 20 is the largest eigenvalue and therefore $\|A\| = \sqrt{20}$.

Problem 12. If C is an $n \times m$ matrix and D is an $m \times n$ matrix then

$$\det(I_n - CD) = \det(I_m - DC)$$

where I_n is the $n \times n$ identity matrix. Prove the identity.

Solution 12. Let

$$X = \begin{pmatrix} I_n & C \\ D & I_m \end{pmatrix}, \qquad Y = \begin{pmatrix} I_n & 0 \\ -D & I_m \end{pmatrix} .$$

Then

$$\det(XY) = \det \begin{pmatrix} I_n - CD & C \\ 0 & I_m \end{pmatrix} = \det(I_n - CD)$$

and

$$\det(YX) = \det \begin{pmatrix} I_n & C \\ 0 & I_m - DC \end{pmatrix} = \det(I_m - DC) .$$

The result follows from $\det(XY) = \det(YX)$. Which of the two sides is faster to calculate?

Problem 13. Let C be an $n \times m$ matrix and D be an $m \times n$ matrix. Show that the $n \times n$ matrix CD and the $m \times m$ DC have the same non-zero eigenvalues.

Solution 13. The characteristic polynomial of CD is

$$p_n(\lambda) = \det(\lambda I_n - CD) = \lambda^n \det(I_n - (\lambda^{-1}C)D)$$

and the characteristic polynomial of DC is

$$p_m(\lambda) = \det(\lambda I_m - DC) = \lambda^m \det(I_m - D(\lambda^{-1}C)) .$$

Applying that $\det(I_n - CD) = \det(I_m - DC)$ from problem 12 we find that

$$p_m(\lambda) = \lambda^{m-n} p_n(\lambda)$$

and the characteristic polynomial of CD and DC have the same non-zero roots.

Problem 14. A *magic square* is an $n \times n$ matrix of the integers from 1 to n^2 such that the sum of each row and column and the two major diagonals is the same. For example

$$\begin{pmatrix} 15 & 8 & 1 & 24 & 17 \\ 16 & 14 & 7 & 5 & 23 \\ 22 & 20 & 13 & 6 & 4 \\ 3 & 21 & 19 & 12 & 10 \\ 9 & 2 & 25 & 18 & 11 \end{pmatrix} .$$

In this example, the common sum is 65. Coxeter has given the following rule for generating a magic square when n is odd. Put a one in the middle of the top row. Go up and left assigning numbers in increasing order to empty boxes. If our move causes us to jump off the square (that is, we go beyond the square's boundaries), figure out where we would be if we landed on a box on the opposite side of the square. Continue with this box. If a box is occupied go down, instead of up and left and continue. Write a C++ program that implements this rule.

Solution 14.

```
// Coxeter.cpp

#include <iostream>
```

```
using namespace std;

const int maxsize = 15;

int main(void)
{
  static int square[maxsize][maxsize];
  int i, j, row, column;
  int count, size;
  cout << "enter the size of the square: ";
  cin >> size;

  // check for input errors
  if((size < 1) || (size > maxsize + 1))
  {
  cout << "size is out of range";
  exit(0);
  }
  if((size%2) == 0)
  {
  cout << "size is even";
  exit(1);
  }

  for(i=0;i<size;i++)
    for(j=0;j<size;j++)
     square[i][j] = 0;

  square[0][(size-1)/2] = 1; // middle of first row

  i = 0;
  j = (size-1)/2;
  for(count=2;count<=size*size;count++)
  {
  row = (i-1 < 0) ? (size-1) : (i-1);    // up
  column = (j-1 < 0) ? (size-1) : (j-1); // left
  if(square[row][column])  // down
    i = (++i)%size;
    else { i = row; j = column; }
    square[i][j] = count;
  }

  // output of magic square
  cout << "magic square of size: " << size << endl;
```

```
  for(i=0;i<size;i++)
  {
    for(j=0;j<size;j++)
    cout << square[i][j] << " ";
    cout << endl;
  }
  return 0;
}
```

Problem 15. Consider the $n \times n$ nonsingular tridiagonal matrix A given by

$$A = \begin{pmatrix} b_0 & c_0 & 0 & 0 & \dots & 0 \\ a_1 & b_1 & c_1 & 0 & \dots & 0 \\ 0 & a_2 & b_3 & c_2 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & & \\ 0 & 0 & & a_{n-2} & b_{n-2} & c_{n-2} \\ 0 & 0 & \dots & 0 & a_{n-1} & b_{n-1} \end{pmatrix}.$$

We seek the *LDU-factorization* of the matrix A - that is, the determination of a unit lower triangular matrix L, a diagonal matrix D, and a unit upper triangular matrix U such that $A = LDU$. It is easy to see (check it) that the matrices L, D and U can be specified as follows

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ \ell_1 & 1 & 0 & 0 & \dots & 0 \\ 0 & \ell_2 & 1 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & & \\ 0 & 0 & \dots & \ell_{n-2} & 1 & 0 \\ 0 & 0 & \dots & 0 & \ell_{n-1} & 1 \end{pmatrix}$$

$$U = \begin{pmatrix} 1 & u_0 & 0 & 0 & \dots & 0 \\ 0 & 1 & u_1 & 0 & \dots & 0 \\ 0 & 0 & 1 & u_2 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & & \\ 0 & 0 & & 0 & 1 & u_{n-2} \\ 0 & 0 & \dots & 0 & 0 & 1 \end{pmatrix}$$

and

$$D = \operatorname{diag}(d_0, d_1, \dots, d_{n-1}).$$

From $A = LDU$ we find

$$\begin{aligned} d_0 &= b_0 \\ d_j &= b_j - a_j c_{j-1}/d_{j-1}, \quad j = 1, 2, \dots, n-1 \\ \ell_j &= a_j/d_{j-1}, \quad j = 1, 2, \dots, n-1 \\ u_j &= c_j/d_j, \quad j = 0, 1, \dots, n-2. \end{aligned}$$

(i) Apply the algorithm to find L, D, and U to the matrix

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 3 \end{pmatrix}.$$

(ii) Use the matrices L, D and U to solve the linear equation

$$A\mathbf{x} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

Solution 15. (i) Applying the recursion relation we find

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}, \quad D = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{pmatrix}.$$

(ii) Since

$$\mathbf{x} = L^{-1}D^{-1}U^{-1}\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

and

$$L^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 1 & -1 & 1 \end{pmatrix}$$

$$U^{-1} = \begin{pmatrix} 1 & -1 & 1 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix}$$

$$D^{-1} = \text{diag}(1\ 1\ 1/2)$$

we obtain

$$\mathbf{x} = \begin{pmatrix} 1 \\ -1 \\ 3/2 \end{pmatrix}.$$

Problem 16. A *Hadamard matrix* is an $m \times m$ matrix H whose entries are ± 1 so that

$$HH^T = mI_m$$

where ^{T} denotes the transpose and I_m is the identity matrix of order m. This means that the rows of H are orthogonal - that is, if Q_j is the jth row

of H, then for $j \neq k$ we have $Q_j Q_k^T = 0$. We consider the case $m = 2^n$, where n is a postive integer. A class of such matrices can be constructed as follows. The registers, C and R, contain the binary representation of the row number r ($r = 0, 1, \ldots, 2^n - 1$) and column number c ($c = 0, 1, \ldots, 2^n - 1$), repectively. Let r_j and c_j be the digits of the binary registers C and R, respectively. We define

$$H(c,r) := (-1)^{s(c,r)}, \qquad s(c,r) := \left(\sum_{j=0}^{n-1} r_j \cdot c_j \right) \bmod 2$$

where $\cdot$ defines the bitwise AND, i.e.,

$$0 \cdot 0 = 0, \quad 0 \cdot 1 = 0, \quad 1 \cdot 0 = 0, \quad 1 \cdot 1 = 1 .$$

Each matrix element is thus defined as the parity of the bitwise AND between a register r, containing its row number, and a register c, containing the column number. Find the matrix H for $n = 3$.

Solution 16. Since $n = 3$ we have an 8×8 matrix. For example, let $c = 0$ and $r = 0$. Then we have bitstrings `"000"` and `"000"`. Thus $s(0,0) = 0$ and $(-1)^0 = +1$. For example, let $r = 5$ and $c = 6$. Then we have the bitstrings `"101"` and `"110"`. Thus

$$s(6,5) = (1 \cdot 0 + 0 \cdot 1 + 1 \cdot 1) \bmod 2 = 1$$

and therefore $(-1)^1 = -1$. Performing this calculation for $r = 0, 1, \ldots, 7$ and $c = 0, 1, \ldots, 7$ we find the matrix

$$H = \begin{pmatrix} +1 & +1 & +1 & +1 & +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & -1 & +1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 & +1 & +1 & -1 & -1 \\ +1 & -1 & -1 & +1 & +1 & -1 & -1 & +1 \\ +1 & +1 & +1 & +1 & -1 & -1 & -1 & -1 \\ +1 & -1 & +1 & -1 & -1 & +1 & -1 & +1 \\ +1 & +1 & -1 & -1 & -1 & -1 & +1 & +1 \\ +1 & -1 & -1 & +1 & -1 & +1 & +1 & -1 \end{pmatrix} .$$

Obviously, the matrix is symmetrical.

Problem 17. An $n \times n$ permutation matrix P is a matrix with precisely one entry whose value is 1 in each column and row, and all of whose other entries are 0. The rows of P are a permutation of the rows of the identity matrix. Describe efficient storages for permutation matrices.

Solution 17. Of course we would not store an $n \times n$ permutation matrix as an $n \times n$ matrix. Since it describes permutations, we could store just

the permutation. For example, for the 4×4 permutation matrix

$$P = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

would be stored as (3 2 4 1) since

$$(1 \quad 2 \quad 3 \quad 4)\,P = (3 \quad 2 \quad 4 \quad 1)$$

and P is uniquely determined by the permutation. We could also store the matrix as a linked list.

Problem 18. A *sparse matrix* is one with a large number of zero entries, enough to make it worthwhile modifying algorithms for these matrices to exploit the presence of these zeros, by not storing them, and by not doing arithmetic with them. Describe different ways to store sparse matrices.

Solution 18. The simplest way is as a list of triples (i, j, a_{ij}), one for each nonzero a_{ij} in the matrix A. The problem with this simple representation is that it is hard to access just one row or one column of A, as required by Gaussian elimination, without inefficiently searching through the whole list. We can however organize this list as a linked list, with entries

$$(i, j, a_{ij}, next_i, last_i, next_j, last_j)$$

where $next_i$ points to the next nonzero entry in column j below i, $last_i$ points to the previous nonzero entry in column j above i, and $next_j$ and $last_j$ similarly point to the neighboring nonzero entries in the same row. This scheme has two drawbacks. First it requires as much as 7 times as much storage as needed just for the nonzero entries themselves. Second, there is no memory locality, i.e., operating on a single row of m entries requires $2m$ memory accesses (to get the data and the $next_i$ pointers) which may or may not be in nearby memory locations.

Another storage scheme is the *column compressed storage*. This scheme stores the nonzero entries by column in one contiguous array called values. In addition, there is an equally long array *row_ind* of the row indices of each value. In other words, $values(i)$ lies in row $row\_ind(i)$. Finally, there is a (generally much shorter) array called $col\_ptr(j)$, indicating where in the values and *row_ind* arrays each column begins: column j starts at index $col\_ptr(j)$ and ends at $col\_ptr(j+1) - 1$. For example, consider the matrix

$$A = \begin{pmatrix} 1 & 0 & 2 & 0 & 4 \\ 0 & 4 & 6 & 0 & 0 \\ 5 & 3 & 0 & 0 & 9 \end{pmatrix}.$$

Then the storage would be

$$\begin{aligned} values &= (1\ 5\ 4\ 3\ 2\ 6\ 4\ 9) \\ row\_ind &= (1\ 3\ 2\ 3\ 1\ 2\ 1\ 3) \\ col\_ptr &= (1\ 3\ 5\ 7\ 7\ 9)\,. \end{aligned}$$

This storage scheme allows fast access to columns but not rows of A, and requires about twice as much storage as for the nonzero entries alone. Since it is not convenient to access rows with this scheme, algorithms using this scheme should be column oriented.

Problem 19. Consider the matrices

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \quad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}.$$

(i) Write down the matrix multiplication table for these matrices.
(ii) We assign bitstrings to the matrices

$$I \to 00, \quad Z \to 01, \quad X \to 10, \quad Y \to 11$$

where the left bit is the pattern bit and the right bit is the sign bit. The operation on these bit strings is carried out by taking the bitwise XOR of the two bit strings, up to a minus sign, and that the sign is negative iff the sign bit of the left operand and the pattern bit of the right operand are both 1. What is the use of this map and the operation described?

Solution 19. (i) Matrix multiplication yields

$*$	I	Z	X	Y
I	I	Z	X	Y
Z	Z	I	$-Y$	$-X$
X	X	Y	I	Z
Y	Y	X	$-Z$	$-I$

(ii) We find

```
00 XOR 00 -> 00 no minus sign
00 XOR 01 -> 01 no minus sign
00 XOR 10 -> 10 no minus sign
00 XOR 11 -> 11 no minus sign
01 XOR 00 -> 01 no minus sign
01 XOR 01 -> 00 no minus sign
01 XOR 10 -> 11 minus sign
01 XOR 11 -> 10 minus sign
```

```
10 XOR 00 -> 10 no minus sign
10 XOR 01 -> 11 no minus sign
10 XOR 10 -> 00 no minus sign
10 XOR 11 -> 01 no minus sign
11 XOR 00 -> 11 no minus sign
11 XOR 01 -> 10 no minus sign
11 XOR 10 -> 01 minus sign
11 XOR 11 -> 00 minus sign
```

The ability to carry out multiplication of basis elements I, X, Y, Z using only simple bit operations is what makes the bit string representation so useful.

Problem 20. To calculate the eigenvalues of the 4×4 matrix

$$A = \begin{pmatrix} 0 & 1 & 0 & 2 \\ 1 & 0 & 2 & 0 \\ 0 & 2 & 0 & 1 \\ 2 & 0 & 1 & 0 \end{pmatrix}$$

one has to solve the quartic equation given by $\det(\lambda I - A) = 0$. Can we avoid solving this quartic equation?

Solution 20. This matrix can be written as the *Kronecker product* of two 2×2 matrices, i.e.,

$$A = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

We only have to find the eigenvalues of the two 2×2 matrices. Then all the possible products of these eigenvalues are the eigenvalues of A. For the first 2×2 matrix, we find the eigenvalues $3, -1$. For the second 2×2 matrix, we find the eigenvalues $1, -1$. Thus, the eigenvalues of the 4×4 matrix A are $3, -3, -1, 1$.

Problem 21. Consider the *geometric series*

$$\sum_{n=0}^{\infty} \frac{1}{12^n} \begin{pmatrix} 7 & 1 \\ 1 & 7 \end{pmatrix}^n.$$

(i) Show that the series converges.
(ii) Describe two methods to calculate the sum. Which of the two methods is more efficient?

Solution 21. (i) Let

$$A = \frac{1}{12} \begin{pmatrix} 7 & 1 \\ 1 & 7 \end{pmatrix}.$$

For the norm we find $\|A\| < 1$. Thus the series converges.
(ii) Since we have a geometric series we have

$$\sum_{n=0}^{\infty} \frac{1}{12^n} \begin{pmatrix} 7 & 1 \\ 1 & 7 \end{pmatrix}^n = \left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \frac{1}{12} \begin{pmatrix} 7 & 1 \\ 1 & 7 \end{pmatrix} \right)^{-1}$$
$$= \left(\frac{1}{12} \begin{pmatrix} 5 & -1 \\ -1 & 5 \end{pmatrix} \right)^{-1}$$
$$= \frac{1}{2} \begin{pmatrix} 5 & 1 \\ 1 & 5 \end{pmatrix}.$$

Using diagonalization of the matrix A

$$A = \frac{1}{12} \begin{pmatrix} 7 & 1 \\ 1 & 7 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 2/3 & 0 \\ 0 & 1/2 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}^{-1}$$

we find

$$\sum_{n=0}^{\infty} \frac{1}{12^n} \begin{pmatrix} 7 & 1 \\ 1 & 7 \end{pmatrix}^n = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1/(1-2/3) & 0 \\ 0 & 1/(1-1/2) \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}^{-1}$$
$$= \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix} \frac{1}{2} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$
$$= \frac{1}{2} \begin{pmatrix} 5 & 1 \\ 1 & 5 \end{pmatrix}.$$

In the first method, we do not need the eigenvalues of A, however, we calculate the inverse of a matrix. For the second method, we calculate the eigenvalues but not the inverse.

Problem 22. Let A be an $n \times n$ symmetric matrix over $\mathbf{R}$. We assume that the real eigenvalues of A are $\lambda_1, \lambda_2, \ldots, \lambda_n$, where

$$|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|.$$

Let $\mathbf{v} \in \mathbf{R}^n$. Then we can write

$$\mathbf{v} = c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_n\mathbf{v}_n$$

where $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ are the eigenvectors of A and $c_1, c_2, \ldots, c_n \in \mathbf{R}$ with $c_1 \neq 0$.
(i) Find $A^p\mathbf{v}$, where $p = 1, 2, \ldots$.
(ii) Show that this provides a way to calculate λ_1 and $\mathbf{v}_1$.
(iii) Give a C++ implementation for the 3×3 matrix

$$A = \begin{pmatrix} 3 & 2 & 1 \\ 2 & 2 & 5 \\ 1 & 5 & 8 \end{pmatrix}.$$

Solution 22. (i) Since $A\mathbf{v}_j = \lambda_j \mathbf{v}_j$ we have

$$\begin{aligned} A\mathbf{v} &= c_1 A\mathbf{v}_1 + c_2 A\mathbf{v}_2 + \cdots + c_n A\mathbf{v}_n \\ &= \lambda_1 \left(c_1 \mathbf{v}_1 + c_2 \frac{\lambda_2}{\lambda_1} \mathbf{v}_2 + \cdots + c_n \frac{\lambda_n}{\lambda_1} \mathbf{v}_n \right) . \end{aligned}$$

Thus

$$A^p \mathbf{v} = \lambda_1^p \left(c_1 \mathbf{v}_1 + c_2 \left(\frac{\lambda_2}{\lambda_1} \right)^p \mathbf{v}_2 + \cdots + c_n \left(\frac{\lambda_n}{\lambda_1} \right)^p \mathbf{v}_n \right) .$$

For large values of p, the vector in the bracket will converge to $c_1 \mathbf{v}_1$, that is, the eigenvector of λ_1. The eigenvalue λ_1 is obtained from

$$\lambda_1 = \lim_{p \to \infty} \frac{(A^{p+1}\mathbf{v})_r}{(A^p \mathbf{v})_r}$$

where r $(r = 1, 2, \ldots, n)$ is the component of the vector with the largest absolute value. The rate of convergence is determined by the quotient. Convergence is faster the smaller $|\lambda_2/\lambda_1|$ is.
(iii) An algorithm can be formulated as follows. Given a vector $\mathbf{y}_k$, we form two other vectors

$$\mathbf{z}_{k+1} = A\mathbf{y}_k, \quad \alpha_{k+1} = \max_{1 \le r \le n} |(\mathbf{z}_{k+1})_r|, \quad \mathbf{y}_{k+1} = \frac{\mathbf{z}_{k+1}}{\alpha_{k+1}}$$

where $k = 0, 1, 2, \ldots$. For the initial vector $\mathbf{y}_0$ we choose

$$\mathbf{y}_0 = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$$

i.e., all components are equal to 1.

```
// largesteig.cpp

#include <iostream>
#include <cmath>
using namespace std;

double largest(double** A,int iter,int n)
{
  int i, j, k;
  double* y = new double[n];
  for(i=0;i<n;i++) y[i] = 1.0;
  double* z = new double[n];
```

```
  double largest;

  for(i=1;i<iter;i++)
  {
  for(j=0;j<n;j++)
  {
  z[j] = 0.0;
  for(k=0;k<n;k++)  z[j] += A[j][k]*y[k];
  }
  largest = z[0];
  for(j=1;j<n;j++)
   if(fabs(z[j]) > fabs(largest)) largest = z[j];
  for(j=0;j<n;j++)  y[j] = z[j]/largest;
  }
  delete[] y; delete[] z;
  return largest;
}

int main(void)
{
  int n = 3;
  double** A = NULL;
  A = new double*[n];
  for(int i=0;i<n;i++)
  A[i] = new double[n];
  A[0][0] = 3.0; A[0][1] = 2.0; A[0][2] = 1.0;
  A[1][0] = 2.0; A[1][1] = 2.0; A[1][2] = 5.0;
  A[2][0] = 1.0; A[2][1] = 5.0; A[2][2] = 8.0;
  cout << "The largest eigenvalue is " << largest(A,10,n);
  for(int j=0;j<n;j++) delete[] A[j];
  delete[] A;
  return 0;
}
```

Problem 23. With the *Leverrier method* the characteristic polynomial and the inverse (if it exists) of an $n \times n$ matrix A can be calculated as follows. We form a sequence of $n \times n$ matrices $B_1, B_2, \ldots, B_n$ from which we calculate the p_k values of the characteristic polynomial

$$(-1)^n(\lambda^n - p_1\lambda^{n-1} - p_2\lambda^{n-2} - \cdots - p_n) = 0$$

where the $(-1)^n$ is used to give the terms of the polynomial the same signs that they would have if the polynomial were generated by expanding the

determinant of $A - \lambda I_n$. We have

$$\begin{aligned}
&B_1 = A \qquad && p_1 = \mathrm{tr}B_1 \\
&B_2 = A(B_1 - p_1 I_n) \qquad && p_2 = \frac{1}{2}\mathrm{tr}B_2 \\
&B_3 = A(B_2 - p_2 I_n) \qquad && p_3 = \frac{1}{3}\mathrm{tr}B_3 \\
&\quad \vdots \qquad\qquad && \quad \vdots \\
&B_n = A(B_{n-1} - p_{n-1} I) \qquad && p_n = \frac{1}{n}\mathrm{tr}B_n \,.
\end{aligned}$$

The inverse (if it exists) is given by

$$A^{-1} = \frac{1}{p_n}(B_{n-1} - p_{n-1} I_n) \,.$$

The condition that the inverse exists is $p_n \neq 0$. Apply the Leverrier method to the matrix

$$A = \begin{pmatrix} 3 & 2 & 4 \\ 2 & 0 & 2 \\ 4 & 2 & 3 \end{pmatrix}$$

and determine whether the inverse exists.

Solution 23. We have $n = 3$. We obtain

$$B_1 = A, \qquad p_1 = \mathrm{tr}B_1 = 6 \,.$$

Thus

$$B_2 = A(B_1 - p_1 I_3) = \begin{pmatrix} 11 & 2 & 4 \\ 2 & 8 & 2 \\ 4 & 2 & 11 \end{pmatrix}$$

and

$$p_2 = \frac{1}{2}\mathrm{tr}B_2 = 15 \,.$$

Finally

$$B_3 = A(B_2 - p_2 I_3) = \begin{pmatrix} 8 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 0 & 8 \end{pmatrix}$$

with

$$p_3 = \frac{1}{3}\mathrm{tr}B_3 = 8 \,.$$

Thus we obtain the characteristic polynomial

$$(-1)^3(\lambda^3 - 6\lambda^2 - 15\lambda - 8) = 0 \,.$$

Since $p_3 \neq 0$ the inverse of A exists, where p_3 is the determinant of A. We find

$$A^{-1} = \frac{1}{p_3}(B_2 - p_2 I_3) = \frac{1}{8}\begin{pmatrix} -4 & 2 & 4 \\ 2 & -7 & 2 \\ 4 & 2 & -4 \end{pmatrix}.$$

Problem 24. Let A be an $n \times n$ hermitian matrix. Then we have the so-called *Stratonovich-Hubbard identity*

$$\exp(A^2) \equiv \int_{-\infty}^{+\infty} \exp(-\pi x^2 I_n - 2Ax\sqrt{\pi})dx$$

where I_n is the $n \times n$ identity matrix. Would it be faster to calculate the right-hand side instead of the left-hand side?

Solution 24. Since $[I_n, A] = 0$, where $[\,,\,]$ denotes the commutator we can write

$$\exp(-\pi x^2 I_n - 2Ax\sqrt{\pi}) = \exp(-\pi x^2 I_n)\exp(-2Ax\sqrt{\pi}).$$

The first matrix on the right-hand side is a diagonal matrix.

Problem 25. The *Householder method* is a method used for reducing an $n \times n$ matrix A to an upper Hessenberg matrix or into tridiagonal form if the matrix A is symmetrical (over $\mathbf{R}$). For a nonzero vector $\mathbf{v} \in \mathbf{R}^n$, we define a *Householder transform* as an $n \times n$ matrix

$$P = I_n - \frac{2}{\|\mathbf{v}\|^2}\mathbf{v}\mathbf{v}^T$$

then P is symmetrical, orthogonal and $P^{-1} = P$. We form a sequence of matrices $A^{(1)} = A$, $A^{(2)} = P^{(1)}AP^{(1)}$, so that after $n-1$ steps the final matrix is nearly Hessenberg since the roundoff error may result in some slightly nonzero entries below the subdiagonal.

For a given $\mathbf{x} \in \mathbf{R}^m$ and $\mathbf{w} = (1, 0, \ldots, 0)^T \in \mathbf{R}^m$, we set

$$\sigma := \|\mathbf{x}\|, \quad \mathbf{v} = \mathbf{x} + \sigma\mathbf{w}.$$

Then we have with $P = I_m - 2\mathbf{v}\mathbf{v}^T/\|\mathbf{v}\|^2$, where $\mathbf{v} \in \mathbf{R}^m$ that

$$P\mathbf{x} = -\sigma\mathbf{w} = (-\sigma, 0, \ldots, 0)^T.$$

Let

$$\mathbf{x} = \begin{pmatrix} a_{21} \\ a_{31} \\ \vdots \\ a_{n1} \end{pmatrix}, \quad \mathbf{w} = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

with $\sigma = \sqrt{\sum_{k=2}^{n} a_{k1}^2}$ and $A = (a_{ij})$. We set

$$P^{(1)} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & & & \\ \vdots & & P & \\ 0 & & & \end{pmatrix}$$

i.e., $P^{(1)} = I_n - (2/\|\mathbf{v}\|^2)\mathbf{v}\mathbf{v}^T$ with $\mathbf{v} = (0, a_{21} + \sigma, a_{31}, \dots, a_{n-1})^T$. Thus we have the matrix

$$P^{(1)}AP^{(1)} = \begin{pmatrix} a_{11} & \dots \\ -\sigma & \\ 0 & \\ \vdots & (*) \\ 0 & \end{pmatrix}.$$

The next step is to apply this step to $(*)$ and so on. Write a C++ program that implements the Householder method.

Solution 25. We consider the nondiagonal matrix

$$A = \begin{pmatrix} 5.3 & 2.3 & 4.6 & 2.7 & 1.6 & 2.2 \\ 2.4 & 7.8 & 5.7 & 8.4 & 3.4 & 4.2 \\ 3.4 & 5.6 & 2.4 & 1.7 & 7.4 & 3.9 \\ 8.3 & 7.5 & 9.2 & 6.1 & 5.2 & 7.9 \\ 4.3 & 5.9 & 7.2 & 2.6 & 4.9 & 0.8 \\ 0.9 & 2.7 & 4.9 & 4.8 & 6.7 & 4.8 \end{pmatrix}.$$

```
// Hessenberg.cpp

#include <iostream>
#include <cmath>
using namespace std;

int main(void)
{
  const int n = 6;
  double A[n][n];
  A[0][0] = 5.3; A[0][1] = 2.3; A[0][2] = 4.6;
  A[0][3] = 2.7; A[0][4] = 1.6; A[0][5] = 2.2;
  A[1][0] = 2.4; A[1][1] = 7.8; A[1][2] = 5.7;
  A[1][3] = 8.4; A[1][4] = 3.4; A[1][5] = 4.2;
  A[2][0] = 3.4; A[2][1] = 5.6; A[2][2] = 2.4;
  A[2][3] = 1.7; A[2][4] = 7.4; A[2][5] = 3.9;
  A[3][0] = 8.3; A[3][1] = 7.5; A[3][2] = 9.2;
```

```
A[3][3] = 6.1; A[3][4] = 5.2; A[3][5] = 7.9;
A[4][0] = 4.3; A[4][1] = 5.9; A[4][2] = 7.2;
A[4][3] = 2.6; A[4][4] = 4.9; A[4][5] = 0.8;
A[5][0] = 0.9; A[5][1] = 2.7; A[5][2] = 4.9;
A[5][3] = 4.8; A[5][4] = 6.7; A[5][5] = 4.8;

double T[n][n];
int ir, i, j, k;
double s, w, ssr, h, uau, b23;
double u[n];

for(ir=0;ir<(n-2);ir++)
{
s = 0.0;
for(i=0;i<n;i++)
{
u[i] = 0.0;
if(i > ir+1) u[i] = A[i][ir];
if(i > ir) s += A[i][ir]*A[i][ir];
} // end for i
w = 1.0;
if(A[ir+1][ir] < 0.0) w = -1.0;
ssr = sqrt(s);
h = s + fabs(A[ir+1][ir])*ssr;
u[ir+1] = A[ir+1][ir] + ssr*w;
uau = 0.0;
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
uau += u[i]*A[i][j]*u[j];
if((i <= ir) && (j <= ir)) T[i][j] = A[i][j];
else if((j == ir) && (i >= ir+2)) T[i][j] = 0.0;
else
{
b23 = 0.0;
for(k=0;k<n;k++)
{
b23 += - (u[i]*A[k][j] + A[i][k]*u[j])*u[k];
}
T[i][j] = A[i][j] + b23/h;
}
}
}
```

```
  uau = uau/(h*h);
  for(i=0;i<n;i++)
  for(j=0;j<n;j++)
  A[i][j] = T[i][j] + uau*u[i]*u[j];
  } // end for ir

  // display the result
  for(i=0;i<n;i++)
  for(j=0;j<n;j++)
  cout << "A[" << i << "][" << j << "] = " << A[i][j] << endl;
  return 0;
} // end main
```

Problem 26. Consider the system of linear equations

$$\begin{pmatrix} B_0 & C_0 & & & & \\ A_0 & B_1 & C_1 & & & \\ & A_1 & B_2 & C_2 & & \\ & & & \ddots & & \\ & & & & B_{n-2} & C_{n-2} \\ & & & & A_{n-2} & B_{n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} D_0 \\ D_1 \\ D_2 \\ \vdots \\ D_{n-1} \end{pmatrix}$$

where the matrix on the left-hand side is an $n \times n$ *tridiagonal matrix*. We assume that the matrix is invertible and that $B_j \neq 0$ for $j = 0, 1, \ldots, n-1$. This type of equation appears when we consider discretization of boundary value problems of differential equations. An algorithm to find $x_0, x_1, \ldots, x_{n-1}$ given A_j, C_j $(j = 0, 1, \ldots, n-2)$, B_k $(k = 0, 1, \ldots, n-1)$ and D_k $(k = 0, 1, \ldots, n-1)$ is as follows:

(1) Calculate in increasing order of $i = 1$ until $n - 1$

$$\begin{aligned} t &\leftarrow A_{i-1}/B_{i-1} \\ B_i &\leftarrow B_i - tC_{i-1} \\ D_i &\leftarrow D_i - tD_{i-1} \end{aligned}$$

(2) Calculate

$$D_{n-1} \leftarrow D_{n-1}/B_{n-1}$$

(3) Calculate in descreasing order of i

$$D_i \leftarrow (D_i - C_i D_{i+1})/B_i, \qquad i = n-2, \ldots, 0\,.$$

The result $(x_0, x_1, \ldots, x_{n-1})$ is stored in $(D_0, D_1, \ldots, D_{n-1})$. Write a C++ program that implements this algorithm. Apply it to the case $n = 3$ and $A_0 = 1.0$, $A_1 = 1.0$, $B_0 = 1.0$, $B_1 = 2.0$, $B_2 = 3.0$, $C_0 = 1.0$, $C_1 = 1.0$ and $D_0 = 1.0$, $D_1 = 1.0$ and $D_2 = 1.0$.

Solution 26. The function `trdg()` implements the algorithm.

```
// tridiagonal.cpp

#include <iostream>
using namespace std;

void trdg(double* A,double* B,double* C,double* D,int n)
{
  int i;
  double t;
  for(i=1;i<n;i++)
  {
  t = A[i-1]/B[i-1];
  B[i] += -t*C[i-1];
  D[i] += -t*D[i-1];
  }
  D[n-1] = D[n-1]/B[n-1];
  for(i=n-2;i>=0;i--)
  {
  D[i] = (D[i] - C[i]*D[i+1])/B[i];
  }
}

int main(void)
{
  int n = 3;
  double* A = new double[n-1];
  A[0] = 1.0; A[1] = 1.0;
  double* B = new double[n];
  B[0] = 1.0; B[1] = 2.0; B[2] = 3.0;
  double* C = new double[n-1];
  C[0] = 1.0; C[1] = 1.0;
  double* D = new double[n];
  D[0] = 1.0; D[1] = 1.0; D[2] = 1.0;
  trdg(A,B,C,D,n);
  for(int i=0;i<n;i++)
  {
  cout << "D[" << i << "] = " << D[i] << endl;
  }
  delete[] A; delete[] B;
  delete[] C; delete[] D;
  return 0;
}
```

Problem 27. Let A be an $n \times n$ symmetric positive-definite matrix over $\mathbf{R}$. Let λ_{max} and λ_{min} be the largest and smallest eigenvalues of A, respectively. Since A is positive-definite the eigenvalues are real and positive. For all θ with $0 < \theta < 1$ the iteration

$$B_{k+1} = B_k + c(A - B_k^2), \qquad B_0 = 2cA$$

with

$$c = \frac{\theta}{2\sqrt{\lambda_{max}}}$$

converges to $\sqrt{A}$. The rate of convergence of the iteration is given by the rate of convergence to zero of the sequence

$$x_k = (1 - \theta\sqrt{\lambda_{min}/\lambda_{max}})^k .$$

As $\|A\| = \alpha\lambda_{max}$ with $\alpha \geq 1$ we set

$$c_1 = \frac{\theta_1}{2\sqrt{\|A\|}} .$$

Choose the norm

$$\|A\| := \max_{0 \leq j < n} \sum_{k=0}^{n-1} |a_{jk}| .$$

Then the iteration with $c = c_1$ converges for all θ_1 with

$$0 < \theta_1 < \sqrt{\alpha} = \sqrt{\|A\|/\lambda_{max}}$$

and therefore in any case for θ_1 with $0 < \theta_1 < 1$. Write a C++ program that implements this iteration. Apply it to the 3×3 matrix

$$A = \begin{pmatrix} 5 & 2 & 1 \\ 2 & 5 & 4 \\ 1 & 4 & 5 \end{pmatrix} .$$

Solution 27. We set $\theta_1 = 0.5$ in the program. Furthermore, `eps = 0.001` is an accuracy parameter.

```
// sqrtmatrix.cpp

#include <iostream>
#include <cmath>
using namespace std;

int main(void)
```

```
{
  const int n = 3;
  double A[n][n];
  A[0][0] = 5.0; A[0][1] = 2.0; A[0][2] = 1.0;
  A[1][0] = 2.0; A[1][1] = 5.0; A[1][2] = 4.0;
  A[2][0] = 1.0; A[2][1] = 4.0; A[2][2] = 5.0;
  double B[n][n];
  double BB[n];
  int i, j, k;
  double delta, r, s;
  double c = 0.0;
  double theta = 0.5;
  double eps = 0.001;

  for(i=0;i<n;i++)
  {
  s = 0.0;
  for(j=0;j<n;j++) { s += fabs(A[i][j]); }
  if(c < s) c = s;
  }
  c = 0.5*theta/sqrt(c);
  // initializing B
  for(i=0;i<n;i++)
  for(j=0;j<n;j++)
  {
  B[i][j] = B[j][i] = 2.0*c*A[i][j];
  }
  // start of iteration
  do
  {
  delta = 0.0;
  for(i=0;i<n;i++)
  {
  for(j=i;j<n;j++)
  {
  s = 0.0;
  for(k=0;k<n;k++) s += - B[i][k]*B[k][j];
  BB[j] = B[i][j] + c*(A[i][j] + s);
  }
  for(j=i;j<n;j++)
  {
  s = fabs(B[i][j] - BB[j]);
  if(s > delta) delta = s;
  B[i][j] = BB[j];
```

```
  }
  }
  for(i=0;i<n-1;i++)
  for(j=i+1;j<n;j++)
  B[j][i] = B[i][j];
  } while(delta > eps);

  // display the result
  for(i=0;i<n;i++)
  for(j=0;j<n;j++)
  cout << "B[" << i << "][" << j << "] = " << B[i][j] << endl;
  return 0;
}
```

Compare this method with the following alternative method using a binomial series, i.e.,

$$C := I_n - \frac{\theta}{\|A\|} A \,.$$

Thus

$$A^{1/2} = \left(\frac{\|A\|}{\theta}\right)^{1/2} \left(I_n - \frac{1}{2}C - \frac{1}{8}C^2 - \frac{1}{16}C^3 - \cdots\right) .$$

For convergence we have

$$\theta < \frac{2\|A\|}{\lambda_{max}} \,.$$

Problem 28. Let $f : \mathbf{R}^n \to \mathbf{R}$ be a quadratic functional

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{x}^T\mathbf{b}$$

with $A \in \mathbf{R}^{n\times n}$ a given symmetric positive-definite matrix. Find the minimum $\mathbf{x}^*$ of f over $\mathbf{R}^n$.

Solution 28. The minimum $\mathbf{x}^*$ of f over $\mathbf{R}^n$ is unique and occurs when the gradient of $f(\mathbf{x})$ vanishes, i.e.,

$$\nabla f(\mathbf{x}^*) = A\mathbf{x}^* - \mathbf{b} = \mathbf{0} \,.$$

This quadratic minimization problem is thus equivalent to solving the system of linear equations $A\mathbf{x} = \mathbf{b}$.

Problem 29. Let A be an $n \times n$ matrix over $\mathbf{R}$. Using recursion write a C++ program that finds the *determinant* of the matrix A.

Solution 29. If we use dynamically allocated arrays we would write a recursive function for the cofactor expansion. Here we use a template function for recursion, i.e., the recursion takes place at compile time.

```
// determinant.cpp

#include <iostream>
using namespace std;

template <const int n>
double det(double A[n][n]);

// determinant of a 1x1 matrix is the only entry of the matrix
template <>
double det<1>(double A[1][1])
{ return A[0][0]; }

// determinant using cofactor expansion
template <const int n>
double det(double A[n][n])
{
  int i, j, k, l, m = 1;
  double sum = 0.0;

  // since n is a const int we can declare the matrix C
  double C[n-1][n-1];

  // iterate through each of the elements in the first row
  // which will be used for the cofactor expansion
  for(i=0;i<n;i++)
  {
  // iterate through the rows of A for the cofactor
  // omitting row 0
  for(j=1;j<n;j++)
  {
  // iterate through the columns of A for the cofactor
  // omitting column i
  for(l=k=0;k<n;k++,l++)
  {
  if(k == i) k++;
  if(k < n) C[j-1][l] = A[j][k];
  }
  }
  // m is the sign in the cofactor expansion which alternates
```

```
  // between -1 and 1
  sum += m*A[0][i]*det<n-1>(C);
  m = -m;
  }
  return sum;
}

int main(void)
{
  double A[1][1] = { { 3.0 } };
  double B[2][2] = { { 1.0,  3.0 },
                     { 7.0, 11.0 } };
  double C[3][3] = { { 1.0, 2.0, 3.0 },
                     { 4.0, 5.0, 6.0 },
                     { 7.0, 8.0, 9.0 } };
  double D[5][5] = { { 1.0, 0.0, 0.0, 0.0, 1.0 },
                     { 1.0, 0.0, 0.0, 0.0,-1.0 },
                     { 0.0, 1.0, 0.0, 1.0, 0.0 },
                     { 0.0, 1.0, 0.0,-1.0, 0.0 },
                     { 0.0, 0.0, 1.0, 0.0, 0.0 } };
  cout << "det(A) = " << det<1>(A) << endl;
  cout << "det(B) = " << det<2>(B) << endl;
  cout << "det(C) = " << det<3>(C) << endl;
  cout << "det(D) = " << det<5>(D) << endl;
  return 0;
}
```

Problem 30. Let A be an $n \times n$ matrix over $\mathbf{C}$. Let $t \in \mathbf{R}$. We define

$$e^{tA} := \sum_{k=0}^{\infty} \frac{A^k t^k}{k!} .$$

If we know the eigenvalues of the matrix A we can calculate e^{tA} as follows (*Putzer algorithm*). Let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the eigenvalues of A, where eigenvalues may be multiple. We set

$$\begin{aligned} B_1 &= I_n \\ B_2 &= (A - \lambda_1 I_n) B_1 \\ B_3 &= (A - \lambda_2 I_n) B_2 \\ &\vdots \\ B_n &= (A - \lambda_n I_n) B_{n-1} . \end{aligned}$$

With this notation, we can write the *Cayley-Hamilton theorem* as

$$B_{n+1} = (A - \lambda_n I_n)B_n = 0\,.$$

Let $y_1(t), y_2(t), \ldots, y_n(t)$ be the solutions of the following system of linear differential equations with constant coefficients

$$\begin{aligned}
\frac{dy_1}{dt} &= \lambda_1 y_1 \\
\frac{dy_2}{dt} &= \lambda_2 y_2 + y_1 \\
&\vdots \\
\frac{dy_n}{dt} &= \lambda_n y_n + y_{n-1}\,.
\end{aligned}$$

With the notation above, we have

$$e^{tA} = y_1(t)B_1 + y_2(t)B_2 + \cdots + y_n(t)B_n\,.$$

Consider a 3×3 matrix with a triple eigenvalue λ. Simplify the calculation of e^{tA}.

Solution 30. We have

$$B_1 = I_3, \quad B_2 = A - \lambda I_3, \quad B_3 = A^2 - 2\lambda A + \lambda^2 I_3$$

and

$$y_1(t) = e^{\lambda t}, \quad y_2(t) = te^{\lambda t}, \quad y_3(t) = \frac{t^2}{2}e^{\lambda t}\,.$$

Thus

$$e^{tA} = e^{\lambda t}I_3 + te^{\lambda t}(A - \lambda I_3) + \frac{1}{2}t^2 e^{\lambda t}(A - \lambda I_3)^2\,.$$

Problem 31. Let

$$p(x) = c_0 + c_1 x + \cdots + c_{n-1}x^{n-1}$$

in which $c_{n-1} \neq 0$, be a polynomial of degree $n-1$ in the unknown x. Let A_p be the *companion matrix*

$$A_p := \begin{pmatrix}
0 & 1 & 0 & \ldots & 0 & 0 \\
0 & 0 & 1 & \ldots & 0 & 0 \\
0 & 0 & 0 & & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & \ldots & 0 & 1 \\
-c_0 & -c_1 & -c_2 & \ldots & -c_{n-2} & -c_{n-1}
\end{pmatrix}.$$

Calculate $\det(A_p - xI_n)$ and discuss.

Solution 31. Obviously we find

$$\det(A_p - xI_n) = (-1)^n p(x)\,.$$

Problem 32. We consider 3×3 matrices over $\mathbf{R}$. An *orthogonal matrix* Q ($Q^T = Q^{-1}$) such that $\det(Q) = 1$ will be called a *rotation matrix*. Let $1 \le p < r \le 3$ and ϕ be a real number. An orthogonal 3×3 matrix $Q_{pr}(\phi) = (q_{ij})_{1\le i,j\le 3}$ given by

$$\begin{aligned} q_{pp} &= q_{rr} = \cos\phi \\ q_{ii} &= 1 \text{ if } i \neq p, r \\ q_{pr} &= -q_{rp} = -\sin\phi \\ q_{ip} &= q_{pi} = q_{ir} = q_{ri} = 0 \text{ if } i \neq p, r \\ q_{ij} &= 0 \text{ if } i \neq p, r \text{ and } j \neq p, r \end{aligned}$$

will be called a plane rotation through ϕ in the plane span $(\mathbf{e}_p, \mathbf{e}_r)$. Let $Q = (q_{ij})_{1\le i,j\le 3}$ be a rotation matrix. Show that there exist angles ϕ in $[0, \pi)$ and θ, ψ in $(-\pi, \pi]$ called the *Euler angles* of Q such that

$$Q = Q_{12}(\phi)Q_{23}(\theta)Q_{12}(\psi)\,. \tag{1}$$

Solution 32. The algebraic proof of (1) is simply the QR factorization of our rotation matrix Q. We set $Q_1 = Q$, $Q_2 = Q_{12}(-\phi)Q_1$, $Q_3 = Q_{23}(-\theta)Q_2$, $Q_4 = Q_{12}(-\psi)Q_3$, where

$$Q_k := (q_{ij}^{(k)})_{1\le i,j\le 3}\,.$$

We then choose ϕ, θ, ψ to be the numbers that subsequently annihilate $q_{13}^{(2)}$, $q_{23}^{(3)}$, $q_{12}^{(4)}$, that is such that

$$\cot\phi = -\frac{q_{23}^{(1)}}{q_{13}^{(1)}}, \qquad \cot\theta = -\frac{q_{33}^{(2)}}{q_{23}^{(2)}}, \qquad \cot\psi = -\frac{q_{22}^{(3)}}{q_{12}^{(3)}}\,.$$

Q_4 is a rotation lower triangular matrix and hence the 3×3 identity matrix. Since $Q_{pr}(-\psi) = Q_{pr}(\psi)^{-1}$ we obtain (1). The rotation matrix is

$$Q = \begin{pmatrix} \cos\phi\cos\psi - \sin\phi\cos\theta\sin\psi & -\cos\phi\sin\psi - \sin\phi\cos\theta\cos\psi & \sin\phi\sin\theta \\ \sin\phi\cos\psi + \cos\phi\cos\theta\sin\psi & -\sin\phi\sin\psi + \cos\phi\cos\theta\cos\psi & -\cos\phi\sin\theta \\ \sin\theta\sin\psi & \sin\theta\cos\psi & \cos\theta \end{pmatrix}.$$

Problem 33. To allocate memory in C++ for an n-dimensional array, we use the `new` operator n-times. How would we implement this using recursion?

Solution 33. We allocate our top-level array and then use recursion to allocate $n-1$ dimensional arrays. The following C++ program shows how to achieve this.

```
// arrays.cpp

#include <iostream>
#include <cassert>
using namespace std;

template<class T> class Array
{
   public:
     Array<T>(int n=1);
     ~Array<T>();
     // for a larger than one dimensional array
     Array<T>& operator[](int);
     T& operator()(int);
     void resize(int);
   protected:
     Array<T>* subarrays;
     T* data;
     T single;
     int size, dim;
     void redim(int);
};

template<class T> Array<T>::Array(int n)
{
   assert(n > 0);
   size = 0; dim = n;
}

template<class T> Array<T>::~Array()
{
   if(size)
   {
   if(dim == 1) delete[] data;
   else delete[] subarrays;
   }
```

```
}

template<class T> Array<T>& Array<T>::operator[](int i)
{
   assert((i >= 0) && (i < size));
   assert(dim > 1);
   return subarrays[i];
}

template<class T> T& Array<T>::operator()(int i)
{
   assert((i >= 0) && (i < size));
   assert(dim == 1);
   return data[i];
}

template<class T> void Array<T>::resize(int i)
{
  assert(i >= 0);
  if(i != size)
  {
  if(size)
  {
  if(dim == 1) delete[] data;
  else delete[] subarrays;
  }
  size = i;
  if(size > 0)
  {
  if(dim == 1) data = new T[size];
  else
  {
  subarrays = new Array<T>[size];
  for(int j=0;j<size;j++)
  subarrays[j].redim(dim-1);
  }
  }
  }
}

template<class T> void Array<T>::redim(int n)
{ dim = n; }

int main(void)
```

```
{
  Array<int> a1(1);
  Array<int> a2(2);
  Array<int> a3(3);
  a1.resize(3);
  a1(0) = 1; a1(1) = 2; a1(2) = 3;
  int sum1 = 0;
  for(int j=0;j < 3;j++) sum1 += a1(j);
  cout << " sum1 = " << sum1;

  a2.resize(2);
  a2[0].resize(2); a2[1].resize(2);
  a2[0](0) = 1; a2[0](1) = 2; a2[1](0) = 3; a2[1](1) = 4;
  int sum2 = 0;
  for(int k=0;k<2;k++)
  for(int m=0;m<2;m++)
  sum2 += a2[k](m);
  cout << " sum2 = " << sum2;

  a3.resize(2);
  a3[0].resize(2); a3[1].resize(2);
  a3[0][0].resize(2); a3[0][1].resize(2);
  a3[1][0].resize(2); a3[1][1].resize(2);
  a3[0][0](0) = 1; a3[0][0](1) = 2;
  a3[0][1](0) = 3; a3[0][1](1) = 4;
  a3[1][0](0) = 5; a3[1][0](1) = 6;
  a3[1][1](0) = 7; a3[1][1](1) = 8;
  int sum3 = 0;
  for(int n=0;n<2;n++)
  for(int p=0;p<2;p++)
  for(int q=0;q<2;q++)
  sum3 += a3[n][p](q);
  cout << " sum3 = " << sum3;
  return 0;
}
```

Problem 34. Using the `map` class of the standard template library and the `string` class, write a C++ program for matrices whose subscripts are strings. For example

```
Matrix M;
M["Manchester"]["Leeds"] = 44;
```

Solution 34.

```
// MatrixStrings.cpp

#include <iostream>
#include <iomanip>
#include <string>
#include <map>
using namespace std;

typedef map<string,double> Row;
typedef map<string,Row> Matrix;

int main(void)
{
  Matrix M;
  M["Manchester"]["Leeds"] = 44;
  M["Leeds"]["Manchester"] = 44;
  M["Manchester"]["Liverpool"] = 35;
  M["Liverpool"]["Manchester"] = 35;
  M["Liverpool"]["Leeds"] = 72;
  M["Leeds"]["Liverpool"] = 72;

  Matrix::iterator i;
  Row::iterator j;
  for(i=M.begin();i!=M.end();i++)
    for(j=i->second.begin();j!=i->second.end();j++)
    {
    int n = i->first.length() + j->first.length() + 3;
    cout << "(" << i->first << "," << j->first << ")"
         << setw(30-n) << j->second << endl;
    }
  return 0;
}
```

Chapter 6

Recursion

Problem 1. The *Fibonacci numbers* are defined by the recurrence relation

$$f(n+2) = f(n+1) + f(n)$$

where $n = 0, 1, 2, \ldots$ and $f(0) = 0$, $f(1) = 1$. We find

$$f(2) = 1, \quad f(3) = 2, \quad f(4) = 3, \quad f(5) = 5, \quad f(6) = 8, \ldots$$

Thus, we can apply recursion to obtain the Fibonacci numbers. The *golden mean number* g is defined as

$$g := \lim_{n \to \infty} \frac{f(n+1)}{f(n)} .$$

Write a Java program using recursion that finds the Fibonacci sequence using the data type `long`. Find also an approximation of the golden mean number g.

Solution 1. Extend the following program using the Java class `BigInteger`.

```
// Fibonacci.java

public class Fibonacci
{
  public static long f(long n)
  {
  if(n == 0) return 0;
  if(n == 1) return 1;
```

```
  return f(n-1) + f(n-2);
  }

  public static void main(String[] args)
  {
  long n = 10;
  long result = f(n);
  System.out.println("result = " + result);
  double gapprox = ((double) f(n))/((double) f(n-1));
  System.out.println("gapprox = " + gapprox);
  }
}
```

Problem 2. A *palindrome* is a string that equals itself when reversed, for example, `"racecar"`, `"abba"`, `"12321"`. Write a recursive method in Java that returns true if a given string is a palindrome. What is/are the base case(s)? What is the recursive case?

Solution 2. The recursive definition of a palindrome is: An empty string is a palindrome. A single character string is a palindrome. A string of length > 1 is a palindrome if and only if its first and last characters are the same, and the substring obtained by removing the first and last characters is also a palindrome.

```
// Palindrome.java

public class Palindrome
{
  private String pal;

  public Palindrome(String initPal)
  { pal = initPal.toUpperCase(); }

  public boolean isPalindrome()
  {
  if(pal.length() <= 1)
  { return true; }  // base case

  // get the first and last characters of the String
  char first = pal.charAt(0);
  char last = pal.charAt(pal.length()-1);

  if(Character.isLetter(first) && Character.isLetter(last))
```

```
  {
  if(first != last) { return false; }
  else
  {
  Palindrome sub =
    new Palindrome(pal.substring(1,pal.length()-1));
  return sub.isPalindrome(); // recursive case
  }
  }
  else if(!Character.isLetter(first))
  {
  Palindrome sub = new Palindrome(pal.substring(1));
  return sub.isPalindrome();
  }
  else
  {
  Palindrome sub =
    new Palindrome(pal.substring(0,pal.length()-1));
  return sub.isPalindrome();
  }
  }

  public static void main(String[] args)
  {
  Palindrome p1 = new Palindrome("dad");
  System.out.println(p1.isPalindrome());
  Palindrome p2 = new Palindrome("racecar");
  System.out.println(p2.isPalindrome());
  Palindrome p3 = new Palindrome("nop");
  System.out.println(p3.isPalindrome());
  Palindrome p4 = new Palindrome("Madam, I'm Adam.");
  System.out.println(p4.isPalindrome());
  }
}
```

Problem 3. Square a positive integer value by only using adds, subtracts, multiply by 2 (left shift) and recursion.

Solution 3. We have the identity

$$\begin{aligned} x^2 &\equiv (x-1+1)^2 \\ &\equiv (x-1)^2 + 2(x-1) + 1 \\ &\equiv (x-1)^2 + 2x - 1\,. \end{aligned}$$

Thus we can calculate x^2 if we can calculate $(x-1)^2$. Naturally, 0^2 is 0. The recursive implementation follows trivially.

Problem 4. Let $n \in \mathbf{N}_0$. The *Hofstadter function* is defined as follows

$$g(n) = \begin{cases} 0 & \text{if } n = 0 \\ n - g(g(n-1)) & \text{if } n \geq 1 \end{cases}$$

Implement this function in C++ using recursion.

Solution 4. We use recursion to implement this function. The function `g()` is called twice for every evaluation of `g(n)`, first with the parameter `n-1` and then with the parameter `g(n-1)`.

```
// Hofstadter.cpp

#include <iostream>
using namespace std;

unsigned long g(unsigned long n)
{
  if(n == 0) return 0;
  else return (n - g(g(n-1)));
}

int main(void)
{
  unsigned long n = 7;
  unsigned long r1 = g(n);
  cout << "r1 = " << r1 << endl;
  n = 10;
  unsigned long r2 = g(n);
  cout << "r2 = " << r2 << endl;
  n = 100;
  unsigned long r3 = g(n);
  cout << "r3 = " << r3 << endl;
  return 0;
}
```

Problem 5. The following algorithm finds the k-th smallest element of a sequence S:

select(k, S)

1. If $|S| = 1$ then the required element is the only element in S.
2. Choose an arbitrary (random) element a from S.
3. Let $S_1 := \{s \in S \mid s < a\}$.
   Let $S_2 := \{s \in S \mid s = a\}$.
   Let $S_3 := \{s \in S \mid s > a\}$.
4. If $|S_1| \geq k$ the required element is given by select(k, S_1).
5. If $|S_1| + |S_2| \geq k$ then the required element is a.
6. The required element is given by select$(k - |S_1| - |S_2|, S_3)$.

Give a C++ implementation that finds the *median*, i.e., the $|S|/2$ smallest element of S.

Solution 5. We use the `vector` template class to denote the sequences. The function `select()` implements the selection algorithm, while the function `median()` uses `select()` to find the median element of the vector.

```
// median.cpp

#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
using namespace std;

template <class T> T select(int k,vector<T> s)
{
  int j;
  vector<T> s1, s2, s3;
  if(s.size() == 1) return s[0];
  j = rand()%s.size();
  for(int i=0;i<s.size();i++)
  if(s[i] < s[j]) s1.push_back(s[i]);
  else if(s[i] == s[j]) s2.push_back(s[i]);
  else s3.push_back(s[i]);
  if(s1.size() >= k) return select(k,s1);
  else if(s1.size()+s2.size() >= k) return s[j];
  return select(k-s1.size()-s2.size(),s3);
}

template <class T> T median(vector<T> s)
{ return select(s.size()/2,s); }
```

```
int main(void)
{
  vector<int> v1(7);
  v1[0] = 3; v1[1] = 1; v1[2] = 5; v1[3] = 3; v1[4] = 7;
  v1[5] = 2; v1[6] = 37;
  cout << "v1 = (";
  for(int i=0;i<v1.size();i++) cout << v1[i] << " ";
  cout << ")" << endl;
  cout << "median(v1) = " << median(v1) << endl;

  vector<double> v2(6);
  v2[0] = 3.0; v2[1] = 21.9; v2[2] = 10.1; v2[3] = 0.5;
  v2[4] = 0.7; v2[5] = 1.0;
  cout << "v2 = (";
  for(int j=0;j<v2.size();j++) cout << v2[j] << " ";
  cout << ")" << endl;
  cout << "median(v2) = " << median(v2) << endl;
  return 0;
}
```

Problem 6. Write a C++ program to sort an array of integers using the *quicksort algorithm*:

- Split the array into two partitions, all those less than the first element, and all those greater than the first element.
- Sort the two partitions.

Solution 6. Partition around the first element of the array, any element could have been used. `p` is the index of the element around which the partition is made and `pe` (declared below) points to the element after the second partition.

```
// qsort.cpp

#include <iostream>
#include <string>
using namespace std;

// general definition of ordering R(t1,t2)
// returns > 0 if t2 R t1, <= 0 otherwise
template <class T>
```

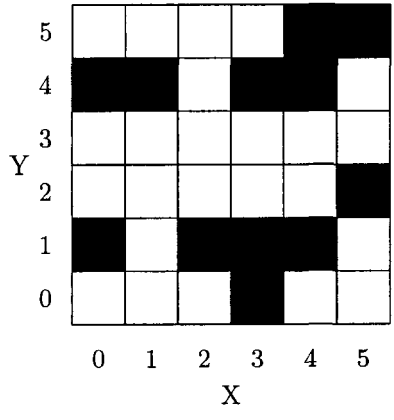
```
void partition(T *array,int n,int (*R)(T,T),int &p)
{
  int i = n-1, pe = 1;
  T t1, t2;
  p = 0;
  while(i > 0)
  {
  if(R(array[p],array[pe]) > 0)
  {
  t1 = array[p]; t2 = array[p+1];
  array[p++] = array[pe]; // put element in first partition
  array[p] = t1;  // move element around which partition
                  // is made, one element right
  if(pe-p > 0)    // if the second partition is not empty
  array[pe] = t2; // move second partition one element right
  }
  pe++;
  i--;
  }
}

template <class T>
void qsort(T *array,int n,int (*R)(T,T))
{
  int pelement;
  if(n <= 1) return;
  partition(array,n,R,pelement);
  qsort(array,pelement,R);
  qsort(array+pelement+1,n-pelement-1,R);
}

int less_int(int n1,int n2) { return (n1 > n2); }
int less_string(string n1,string n2) { return (n1 > n2); }

int main(void)
{
  int test1[9] = { 1,5,3,7,2,9,4,6,8 };
  qsort<int>(test1,9,less_int);
  for(int i=0;i<9;i++) cout << test1[i] << "  ";
  cout << endl;

  string test2[4] = { "orange","grape","apple","banana" };
  qsort<string>(test2,4,less_string);
  for(int j=0;j<4;j++) cout << test2[j] << "  ";
```

```
  cout << endl;
  return 0;
}
```

The output of the program is

```
1  2  3  4  5  6  7  8  9
apple  banana  grape  orange
```

Problem 7. We have a two-dimensional grid of cells, each of which may be empty or filled. If the cell is empty we associate it with a 0 and if the cell is filled we associate it with a 1. Any group of cells which are connected horizontally, vertically, or diagonally constitutes a *blob*. We may imagine that the cells have been created by scanning a microscope slide of a bacterial culture, and that the purpose is to estimate the degree of infection. The figure shows an example. Write a Java program using recursion that accepts the coordinates of a cell, and returns the number of cells in the blob that contains the cell. The result is zero if the designated cell is empty.



Solution 7. We using eight recursive calls since there are eight neighbors of a given cell. This includes the diagonal cells.

```
// Blob.java

public class Blob
{
  public static int cc(int x,int y,int[][] grid,
                       int maxX,int maxY)
  {
  int sum = 0;
```

```
  if((x < 0) || (x >= maxX) || (y < 0) || (y >= maxY))
  return 0;
  if(grid[x][y] == 0) return 0;
  else
  {
  grid[x][y] = 0;
  sum = 1 + cc(x-1,y-1,grid,maxX,maxY)
          + cc(x-1,y,grid,maxX,maxY)
          + cc(x-1,y+1,grid,maxX,maxY)
          + cc(x,y+1,grid,maxX,maxY)
          + cc(x+1,y+1,grid,maxX,maxY)
          + cc(x+1,y,grid,maxX,maxY)
          + cc(x+1,y-1,grid,maxX,maxY)
          + cc(x,y-1,grid,maxX,maxY);
  }
  return sum;
  }

  public static void main(String[] args)
  {
  int maxX = 6;
  int maxY = 6;
  int[][] grid = new int[maxX][maxY];
  for(int i=0;i<maxX;i++)
   for(int j=0;j<maxY;j++)
    grid[i][j] = 0;

  grid[0][3] = 1;
  grid[1][0] = grid[1][2] = grid[1][3] = grid[1][4] = 1;
  grid[2][5] = 1;
  grid[4][0] = grid[4][1] = grid[4][3] = grid[4][4] = 1;
  grid[5][4] = grid[5][5] = 1;
  int result1 = cc(1,3,grid,maxX,maxY);
  System.out.println("result1 = " + result1);
  int result2 = cc(1,0,grid,maxX,maxY);
  System.out.println("result2 = " + result2);
  int result3 = cc(5,5,grid,maxX,maxY);
  System.out.println("result3 = " + result3);
  int result4 = cc(2,3,grid,maxX,maxY);
  System.out.println("result4 = " + result4);
  }
}
```

Problem 8. The *Jacobi elliptic functions* can be defined as the inverse of the *elliptic integral of first kind.* Thus, if we write

$$x(\phi, k) = \int_0^{\phi} \frac{ds}{\sqrt{1 - k^2 \sin^2 s}} \tag{1}$$

where $k \in [0, 1]$ we then define the following functions

$$\mathrm{sn}(x, k) := \sin(\phi), \qquad \mathrm{cn}(x, k) := \cos(\phi), \qquad \mathrm{dn}(x, k) := \sqrt{1 - k^2 \sin^2 \phi}\,. \tag{2}$$

For $k = 0$ we obtain

$$\mathrm{sn}(x, 0) \equiv \sin(x), \qquad \mathrm{cn}(x, 0) \equiv \cos(x), \qquad \mathrm{dn}(x, 0) \equiv 1 \tag{3}$$

and for $k = 1$ we have

$$\mathrm{sn}(x, 1) \equiv \tanh(x), \qquad \mathrm{cn}(x, 1) \equiv \mathrm{dn}(x, 1) \equiv \frac{2}{e^x + e^{-x}}\,. \tag{4}$$

We have the following identities

$$\begin{aligned}
\mathrm{sn}(x, k) &\equiv \frac{2\mathrm{sn}(x/2, k)\mathrm{cn}(x/2, k)\mathrm{dn}(x/2, k)}{1 - k^2\mathrm{sn}^4(x/2, k)} \\
\mathrm{cn}(x, k) &\equiv \frac{1 - 2\mathrm{sn}^2(x/2, k) + k^2\mathrm{sn}^4(x/2, k)}{1 - k^2\mathrm{sn}^4(x/2, k)} \\
\mathrm{dn}(x, k) &\equiv \frac{1 - 2k^2\mathrm{sn}^2(x/2, k) + k^2\mathrm{sn}^4(x/2, k)}{1 - k^2\mathrm{sn}^4(x/2, k)}\,.
\end{aligned} \tag{5}$$

The expansions of the Jacobi elliptic functions in powers of x up to order 3 are given by

$$\begin{aligned}
\mathrm{sn}(x, k) &= x - (1 + k^2)\frac{x^3}{3!} + \cdots \\
\mathrm{cn}(x, k) &= 1 - \frac{x^2}{2!} + \cdots \\
\mathrm{dn}(x, k) &= 1 - k^2\frac{x^2}{2!} + \cdots
\end{aligned} \tag{6}$$

For x sufficiently small this will be a good approximation.
(i) Use the identities (5) and the expansions (6) to implement the Jacobi elliptic functions using one recursive call.
(ii) Write a C++ program.

Solution 8. (i) The recursive call in `scdn()` uses half of the provided parameter x, in other words the absolute value of the parameter passed in

the recursive call is always smaller (by $\frac{1}{2}$). This guarantees that for fixed $\epsilon > 0$ the parameter x will satisfy $x < \epsilon$ after a finite number of recursive calls. At this point a result is returned immediately using the polynomial approximation (6). This ensures that the algorithm will complete successfully. The recursive call is possible due to the identities for the sn, cn and dn functions given in (5). Since the identities depend on all three functions sn, cn and dn we can calculate all three at each step instead of repeating calculations for each of sn, cn and dn. The denominator of all three identities is the same. All three Jacobi elliptic functions are found with one function call. The cases $k = 0$ and $k = 1$ include the sine, cosine, tanh and sech functions. For these special cases faster routines are available. Elliptic functions belong to the class of doubly periodic functions in which $2K$ plays a similar role to π in the theory of circular functions, where $K = F(1, k)$ is the complete elliptic integral of first kind. We have the identities

$$\mathrm{sn}(x \pm 2K, k) \equiv -\mathrm{sn}(x, k) \tag{7a}$$

$$\mathrm{cn}(x \pm 2K, k) \equiv -\mathrm{cn}(x, k) \tag{7b}$$

$$\mathrm{dn}(x \pm 2K, k) \equiv \mathrm{dn}(x, k)\,. \tag{7c}$$

To reduce the argument of the Jacobi elliptic functions we can also apply these identities.
(ii) The recursion method described above can be implemented using C++ as follows.

```
// jacobi.cpp

#include <iostream>
#include <cmath>
using namespace std;

// forward declaration
void scdn(double,double,double,double&,double&,double&);

int main(void)
{
  double x, k, k2, eps;
  x = 3.0;
  cout << "x = " << x << endl;
  eps = 0.01;
  double res1, res2, res3;

  // sin,cos,1 of x
  k = 0.0;
  k2 = k*k;
```

```
  scdn(x,k2,eps,res1,res2,res3);
  cout << "sin(x) = " << res1 << endl;
  cout << "cos(x) = " << res2 << endl;
  cout << "1(x) = " << res3 << endl;

  // tanh,sech,sech of x
  k = 1.0;
  k2 = k*k;
  scdn(x,k2,eps,res1,res2,res3);
  cout << "tanh(x) = " << res1 << endl;
  cout << "sech(x) = " << res2 << endl;
  cout << "sech(x) = " << res3 << endl;
  return 0;
}

void scdn(double x,double k2,double eps,double &s,double &c,
          double &d)
{
  if(fabs(x) < eps)
  {
  double x2 = x*x/2.0;
  s = x*(1.0 - (1.0 + k2)*x2/3.0);
  c = 1.0 - x2; d = 1.0 - k2*x2;
  }
  else
  {
  double sh,ch,dh;
  scdn(x/2.0,k2,eps,sh,ch,dh);  // recursive call
  double sh2 = sh*sh;
  double sh4 = k2*sh2*sh2;
  double denom = 1.0 - sh4;
  s = 2.0*sh*ch*dh/denom;
  c = (1.0 - 2.0*sh2+sh4)/denom;
  d = (1.0 - 2.0*k2*sh2+sh4)/denom;
  }
}
```

Problem 9. Let n be a given positive integer > 1. Let $i = 0, 1, \ldots, n-2$ and $j = 1, 2, \ldots, n-1$ with $i < j$. Consider the map

$$f(0,1) = 0, \quad f(0,2) = 1, \quad f(0,3) = 2, \ \ldots, \ f(0,n-1) = n-2$$

$$f(1,2) = n-1, \quad f(1,3) = n, \quad f(1,4) = n+1, \ \ldots, \ f(1,n-1) = 2n-4$$

$$f(2,3) = 2n-3, \ \ldots, \ f(n-2,n-1) = n(n-1)/2\,.$$

Write a recursion using C++ that implements this function.

Solution 9.

```
// map.cpp

#include <iostream>
using namespace std;

int f(int i,int j,int n)
{
  if((i==0) && (j==1)) return 0;
  else if((j > (i+1)) && (i >= 0)) return (f(i,j-1,n)+1);
  else if((i > 0) && (j == (i+1))) return (f(i-1,j,n)+n-i-1);
}

int main(void)
{
  cout << f(0,1,4) << endl; // => 0
  cout << f(0,3,4) << endl; // => 2
  cout << f(1,3,4) << endl; // => 4
  cout << f(2,3,4) << endl; // => 5
  cout << f(3,4,5) << endl; // => 9
  cout << f(1,4,5) << endl; // => 6
  cout << f(10,11,12) << endl; // => 65
  return 0;
}
```

Problem 10. Let $(a_0, a_1, \ldots, a_n)$ be an array representing the coefficients of the polynomial

$$p(x) = a_0 x^n + a_1 x^{n-1} + \cdots + a_{n-1} x + a_n .$$

We wish to compute $p(x_0)$, where $x_0 \in \mathbf{R}$. *Horner's algorithm* is a method for computing $p(x_0)$ using n multiplications and n additions. This method is based on rewriting the expression of $p(x_0)$ as follows

$$p(x_0) = (\cdots((a_0 x_0 + a_1)x_0 + a_2)x_0 + \cdots + a_{n-1})x_0 + a_n .$$

Find a recursion relation for this expression.

Solution 10. We can obtain $p(x_0)$ by computing the innermost term $y_1 = a_0 x_0 + a_1$, then the next innermost term $y_2 = y_1 x_0 + a_2$, and so on, until we obtain

$$p(x_0) = y_n = y_{n-1} x_0 + a_n .$$

This method can be expressed by the following linear recurrence

$$y_0 = a_0$$
$$y_j = y_{j-1}x_0 + a_j, \qquad j = 1, 2, \ldots, n\,.$$

This recurrence is a first-order linear recurrence since y_j depends only on y_{j-1}.

Problem 11. Let n be a positive integer. The *Walsh-Hadamard transform* of a signal (row vector) $\mathbf{x}$, of size $N = 2^n$, is the matrix-vector product $W_N\mathbf{x}^T$, where

$$W_N = \bigotimes_{j=1}^{n} W_2 \equiv W_2 \otimes W_2 \otimes \cdots \otimes W_2$$

with

$$W_2 := \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

and $\otimes$ denotes the *Kronecker product*. Let A be an $m \times n$ matrix and let B be a $p \times q$ matrix. Then the Kronecker product of A and B is that $(mp) \times (nq)$ matrix defined by

$$A \otimes B := \begin{pmatrix} a_{11}B & a_{12}B & \ldots & a_{1n}B \\ a_{21}B & a_{22}B & \ldots & a_{2n}B \\ \vdots & & & \\ a_{m1}B & a_{m2}B & \ldots & a_{mn}B \end{pmatrix}.$$

(i) Give a recursive algorithm to calculate W_N.
(ii) Give an iterative algorithm to calculate W_N.

Solution 11. (i) A recursive algorithm for calculating W_N is obtained from the factorization

$$W_N = (W_2 \otimes I_{2^{n-1}})(I_2 \otimes W_{2^{n-1}})$$

where I_2 is the 2×2 unit matrix and $I_{2^{n-1}}$ is the $2^{n-1} \times 2^{n-1}$ unit matrix. This equation corresponds to the divide and conquer step in a recursive fast Fourier transform.
(ii) An iterative algorithm for computing W_N is obtained from the factorization

$$W_N = \prod_{j=1}^{n} (I_{2^{j-1}} \otimes W_2 \otimes I_{2^{n-j}})$$

which corresponds to an iterative fast Fourier transform.

More generally, let $n = n_1 + n_2 + \cdots + n_t$, then

$$W_N = \prod_{j=1}^{N} (I_{2^{n_1+\cdots+n_{j-1}}} \otimes W_{2^{n_j}} \otimes I_{2^{n_{j+1}+\cdots+n_t}}) .$$

This factorization encompasses both the iterative and recursive algorithm and provides a mechanism for exploring different breakdown strategies and combinations of recursion and iteration.

Problem 12. Given an $m \times n$ matrix A with 0's and 1's, for example

$$A = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{pmatrix}$$

where $a_{0,0} = 1$ and $a_{m-1,n-1} = 1$. The matrix represents a *maze* for the 1's and 0's. The aim is to move from position $(0, 0)$ (top left corner) to the position $(m - 1, n - 1)$ (bottom right corner) following the path's of 1's. We can only move to the right, left, down or up, but not diagonal. Write a Java program using recursion that finds out whether or not there is a solution and if so provides a solution.

Solution 12. If a path exists, then it is displayed in the output by 5's.

```
// Maze.java

public class Maze
{
  public static int[][] A = {{1,1,1,0,0,0,1,0},
                             {0,1,1,0,1,1,1,0},
                             {0,0,1,1,0,1,0,0},
                             {1,1,0,1,1,1,1,0},
                             {0,1,1,0,0,0,1,1},
                             {1,1,0,1,1,0,1,1},
                             {1,0,1,0,1,0,1,0},
                             {1,0,0,1,0,0,1,1}};

  public static void display()
  {
  for(int r=0;r<A.length;r++)
```

```
{
for(int c=0;c<A[r].length;c++)
System.out.print(A[r][c]);
System.out.println();
}
System.out.println();
} // end display()

public static boolean solve(int r,int c)
{
boolean done = false;
if(valid(r,c) != false)
{
A[r][c] = 3; // cell has been tried
if(r == A.length-1 && c == A[0].length-1)
  done = true;  // maze is solved
else
{
done = solve(r+1,c); // down
if(done == false) done = solve(r,c+1); // right
if(done == false) done = solve(r-1,c); // up
if(done == false) done = solve(r,c-1); // left
}
if(done != false) A[r][c] = 5;
}
return done;
} // end solve()

public static boolean valid(int r,int c)
{
boolean result = false;
if(r >= 0 && r < A.length && c >= 0 && c < A[0].length)
if(A[r][c] == 1) result = true;
return result;
}

public static void main(String[] args)
{
Maze lab = new Maze();
lab.display();
if(lab.solve(0,0) != false)
  System.out.println("Maze solved");
else System.out.println("no solution");
lab.display();
```

```
  }
}
```

Problem 13. Given an array of sorted elements of data type T. The key value and the lower and upper index bounds are parameters. Write a C++ program using recursion for a *binary search* of the array. If the key is not in the array return -1. Implement the C++ code using templates so that different data types can be applied, for example the data types `double` and `string`.

Solution 13. We use the `string` class from C++.

```
// binarysearch.cpp

#include <iostream>
#include <string>
using namespace std;

template <class T>
int binarysearch(T* arr,int low,int high,T key)
{
  int mid;
  T midvalue;
  // stopping condition: key not found
  if(low > high) return -1;
  // compare against arrays midpoint and subdivide
  // if a match does not occur,
  // apply binary search to the appropiate sublist
  else
  {
  mid = (low + high)/2;
  midvalue = arr[mid];
  if(key == midvalue)
    return mid; // key found at index mid

  // look left if key < midvalue; otherwise look right
  else if(key < midvalue) // recursive step
    return binarysearch(arr,low,mid-1,key);
    else return binarysearch(arr,mid+1,high,key);
  }
}

int main(void)
```

```
{
  int n = 4;
  double* arr = new double[n];
  arr[0] = 1.1; arr[1] = 3.4; arr[2] = 4.7; arr[3] = 5.9;
  int result = binarysearch(arr,0,n-1,3.5);
  cout << "result = " << result << endl;
  delete[] arr;
  string sarr[4] = { "abba", "susi", "willi", "xena" };
  string s = "willi";
  result = binarysearch(sarr,0,3,s);
  cout << "result = " << result;
  return 0;
}
```

Problem 14. We have 50 "gold" coins all of the same weight, except for one which is fake and weighs less. In other words, we have 49 gold coins of equal weight and one fake coin of less weight. We also have a balance scale. Any number of coins can be put on each side of the scale at the same time. The scale indicates if the two sides weigh the same, or which side is lighter if this is not the case. Find an algorithm to locate the fake coin using the balance scale as few times as possible. If there are $2n$ gold coins one of which is fake, how many times would the algorithm use the balance scale?

Solution 14. Since we know exactly one coin weighs less than the others we apply the following strategy. The strategy requires $O(\log n)$ iterations to find the coin (more accurately it requires $O(\log_3 2n)$ iterations).

1. Divide the coins into 3 piles where 2 of the piles have the same number of coins and the last pile has at most 2 coins less than the other two.
2. Weigh two of the piles with an equal number of coins.
3. If the two piles are equal in weight, keep the third pile.
4. Otherwise keep the lighter of the two piles.
5. Repeat with the remaining pile until only one coin remains.

Chapter 7

Finite State Machines

Problem 1. A finite state machine is defined as

$$M = \{ S,\ I,\ O,\ f_S,\ f_0 \}$$

where S is a finite set of states, I is a finite set of input symbols (the input alphabet), O is a finite set of output symbols (the output alphabet), and

$$f_S : S \times I \rightarrow S, \qquad f_O : S \rightarrow O\,.$$

The function f_S is the next-state function. It maps a (state,input) pair to a state. The state at clock pulse t_{j+1}, state (t_{j+1}), is obtained by applying the next-state function to the state at time t_j, and the input at t_j. The function f_O is the output function. When f_O is applied to a state at time t_j, we obtain the output at time t_j. Consider

$$S = \{ s_0,\ s_1,\ s_2 \}, \qquad I = \{ 0,\ 1 \}\,.$$

Let

$$\begin{aligned} f_S(s_0, 0) &= s_1, & f_S(s_0, 1) &= s_0 \\ f_S(s_1, 0) &= s_2, & f_S(s_1, 1) &= s_1 \\ f_S(s_2, 0) &= s_2, & f_S(s_2, 1) &= s_0 \end{aligned}$$

and

$$f_O(s_0) = 0, \quad f_O(s_1) = 1, \quad f_O(s_2) = 1\,.$$

Write a C++ program to find the states and the output. Consider an input sequence 01101.

Solution 1. We implement the input "01101" as a string. The function fS() implements f_S and the function fO() implements f_O.

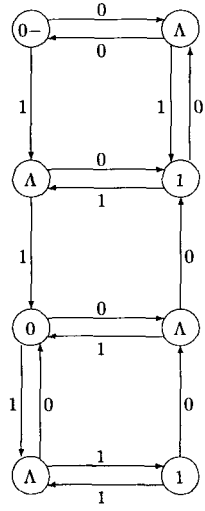
```
// fsm.cpp

#include <iostream>
#include <string>
using namespace std;

string fS(string s,char c)
{
  if((s == "s0") && (c == '0')) return "s1";
  if((s == "s0") && (c == '1')) return "s0";
  if((s == "s1") && (c == '0')) return "s2";
  if((s == "s1") && (c == '1')) return "s1";
  if((s == "s2") && (c == '0')) return "s2";
  if((s == "s2") && (c == '1')) return "s0";
}

int fO(string s)
{
  if(s == "s0") return 0;
  if(s == "s1") return 1;
  if(s == "s2") return 1;
}

int main(void)
{
  string arr = "01101";
  string initial = "s0";
  int output = fO(initial);
  cout << output << endl;
  string temp;
  for(int j=0;j<arr.length();j++)
  {
  temp = fS(initial,arr[j]);
  cout << temp << "  ";
  output = fO(temp);
  cout << output << endl;
  initial = temp;
  }
  return 0;
}
```

Problem 2. A *Moore machine* is a finite state machine that produces an output for each state. The output depends only on the present state. Design a Moore machine which adds two binary numbers of the same length. Give the design in the form of a state diagram. The state diagram is a directed graph where each vertex denotes a state (a vertex is drawn as a circle with the symbol to output on entry to the state, and a - to indicate the initial state) and edges denote transitions between states and are labelled with the input symbol which caused the transition.

Solution 2. We use $\Sigma := \{0, 1\}$ and $\Gamma := \{0, 1, \Lambda\}$, where Σ is an alphabet of possible input symbols and Γ is an alphabet of possible output symbols. Λ is the (null or empty) symbol indicating no output, i.e., we simply ignore this symbol. For the input string we use (where we read from left to right)

$$a_0 b_0 a_1 b_1 \cdots a_n b_n$$

where $a_n a_{n-1} \cdots a_0$ and $b_n b_{n-1} \cdots b_0$ are the binary numbers to be added. The initial state starts with a sum of zero and carry of zero. The upper 4 vertices (states) implement addition when the carry is zero and the lower 4 vertices implement addition when the carry is 1. The upper half is responsible for adding without carry while the lower half is responsible for adding with carry.



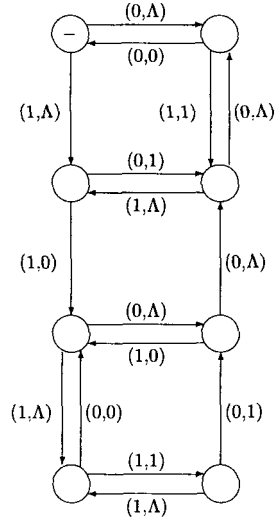
Problem 3. A *Mealy machine* is a finite state machine which produces an output for each transition. The output depends on the present state and the present value of the inputs. Design a Mealy machine which adds two

binary numbers of the same length. Give the design in the form of a state diagram. The state diagram is a directed graph where each vertex denotes a state (a vertex is drawn as a circle with an optional - to indicate the initial state) and edges denote transitions between states and are labelled with the input symbol which caused the transition and the symbol to output for the transition. Describe how to create an equivalent Mealy machine from a Moore machine.

Solution 3. We use $\Sigma = \{0, 1\}$ and $\Gamma = \{0, 1, \Lambda\}$. Λ is the symbol indicating no output, i.e., we simply ignore this symbol. For the input string we use (where we read from left to right)

$$a_0 b_0 a_1 b_1 \cdots a_n b_n$$

where $a_n a_{n-1} \cdots a_0$ and $b_n b_{n-1} \cdots b_0$ are the binary numbers to be added. The initial state starts with a sum of zero and carry zero. The upper 4 vertices (states) implement addition when the carry is zero and the lower 4 vertices implement addition when the carry is 1. The upper half is responsible for adding without carry while the lower half is responsible for adding with carry.

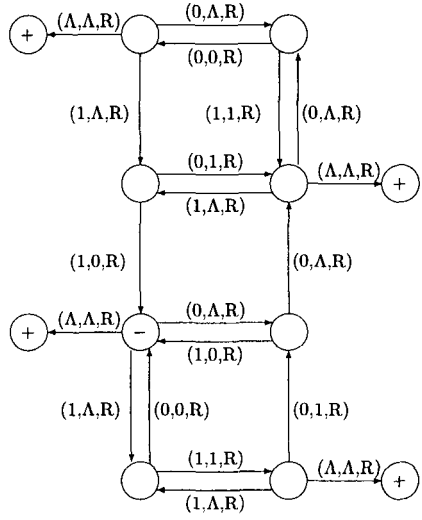


Simply move the target symbol onto all incoming edges, i.e., if a transition to a state with symbol o on reading i then label the transition (i, o).

Problem 4. Let S be a finite set of states and Γ a finite set of tape symbols (the tape alphabet) including the special symbol Λ to denote a blank cell. A *Turing machine* is a set of quintuples of the form (s, i, i', s', d),

where $s, s' \in S$, $i, i' \in \Gamma$, and $d \in \{ R, L \}$ (R for right, L for left) and where no two quintuples begin with the same s and i. Design a Turing machine which subtracts a two's complement binary number from another of the same length. Determine the relevant alphabets and a suitable formatting for the input.

Solution 4. We use $\Gamma := \{0, 1, \Lambda\}$. Λ is the symbol indicating no output, i.e., we simply ignore this symbol. For the input string we use (where we read from left to right) $a_0 b_0 a_1 b_1 \cdots a_n b_n$, where $a_n a_{n-1} \cdots a_0$ and $b_n b_{n-1} \cdots b_0$ are the binary numbers to be added. The upper half is responsible for adding without carry while the lower half is responsible for adding with carry. We negate $b_n b_{n-1} \cdots b_0$ by swapping the meaning of 0 and 1 and by starting in the add with carry portion of the machine (i.e., we negate $b_n b_{n-1} \cdots b_0$ and increment to obtain the two's complement) while adding the two numbers.



Problem 5. Design a *Mealy machine* to represent a vending machine which provides a carbonated drink for R 2,50. The machine accepts 50c, R1 and R2 coins. The symbol (number) output by the machine is interpreted as change (in cents) for the customer. The symbol "C" output by the machine is interpreted as the output of the carbonated drink. We allow null output, i.e., output which does not actually appear (empty string). Thus the output "symbols" we could expect are the strings

"", "C", "C50", "C100", "C150" .

Write a C++ and Java program to simulate the machine we have designed.

Solution 5. We use the states to indicate how many 50c the vending machine has received. In other words $S0$ represents 0c, $S50$ represents 50c and so on. The following transition table describes the machine.

State	Input	Output	Next State
$S0$–	50c		$S50$
$S0$–	R1		$S100$
$S0$–	R2		$S200$
$S50$	50c		$S100$
$S50$	R1		$S150$
$S50$	R2	C	$S0$
$S100$	50c		$S150$
$S100$	R1		$S200$
$S100$	R2	C50	$S0$
$S150$	50c		$S200$
$S150$	R1	C	$S0$
$S150$	R2	C100	$S0$
$S200$	50c	C	$S0$
$S200$	R1	C50	$S0$
$S200$	R2	C150	$S0$

The following C++ program implements this finite state machine.

```
//  machine.cpp

#include <iostream>
using namespace std;

struct transition { char *state, *input, *output, *nextstate; };

transition transitions[] = {
  {"S0"   ,"50c" ,""     ,"S50" },{"S0"   ,"R1"  ,""     ,"S100"},
  {"S0"   ,"R2"  ,""     ,"S200"},{"S50"  ,"50c" ,""     ,"S100"},
  {"S50"  ,"R1"  ,""     ,"S150"},{"S50"  ,"R2"  ,"C"    ,"S0"  },
  {"S100" ,"50c" ,""     ,"S150"},{"S100" ,"R1"  ,""     ,"S200"},
  {"S100" ,"R2"  ,"C50"  ,"S0"  },{"S150" ,"50c" ,""     ,"S200"},
  {"S150" ,"R1"  ,"C"    ,"S0"  },{"S150" ,"R2"  ,"C100","S0"  },
  {"S200" ,"50c" ,"C"    ,"S0"  },{"S200" ,"R1"  ,"C50" ,"S0"  },
  {"S200" ,"R2"  ,"C150","S0"  },{ NULL, NULL, NULL, NULL }};
```

```
char *output[][2] = {
 {"",""},
 {"C","Machine produces a can\n"},
 {"C50","Machine produces a can and 50 cents change\n"},
 {"C100","Machine produces a can and R1 change\n"},
 {"C150","Machine produces a can and R1,50 change\n"},
 {NULL, NULL}};

int main(void)
{
  char *state = transitions[0].state;
  char input[32];
  int i, j;
  cout << "Input a 50c, R1 or R2 coin in the machine"
       << endl;
  cin.getline(input,32,'\n');
  for(i=0;transitions[i].state!=NULL;)
  {
  if(strcmp(state,transitions[i].state)
    || strcmp(input,transitions[i].input)) i++;
  else
  {
  for(j=0;output[j][0]!=NULL;j++)
   if(strcmp(transitions[i].output,output[j][0])==0)
    cout << output[j][1];
  state = transitions[i].nextstate;
  i = 0;
  cout << "Input a 50c, R1 or R2 coin in the machine"
       << endl;
  cin.getline(input,32,'\n');
  }
  }
  cout << "Machine fails invalid coin inserted"
       << endl;
  return 0;
}
```

The following Java program implements this finite state machine

```
// Machine.java

import java.io.*;
```

```
public class Machine
{
   static transition trans[] =
     { new transition("S0","50c","","S50"),
       new transition("S0","R1","","S100"),
       new transition("S0","R2","","S200"),
       new transition("S50","50c","","S100"),
       new transition("S50","R1","","S150"),
       new transition("S50","R2","C","S0"),
       new transition("S100","50c","","S150"),
       new transition("S100","R1","","S200"),
       new transition("S100","R2","C50","S0"),
       new transition("S150","50c","","S200"),
       new transition("S150","R1","C","S0"),
       new transition("S150","R2","C100","S0"),
       new transition("S200","50c","C","S0"),
       new transition("S200","R1","C50","S0"),
       new transition("S200","R2","C150","S0"),
       new transition(null,null,null,null)};

   static String output[][] =
     { {"",""},
     {"C","Machine produces a can\n"},
     {"C50","Machine produces a can and 50 cents change\n"},
     {"C100","Machine produces a can and R1 change\n"},
     {"C150","Machine produces a can and R1.50 change\n"},
     {null,null} };

public static void main(String args[]) throws IOException
{
  String state = trans[0].state;
  String input = "";
  int i = 0, j = 0;
  BufferedReader kbd =
       new BufferedReader(new InputStreamReader(System.in));
  System.out.println("Input a 50c, R1 or R2 coin");
  input = kbd.readLine();

  for(i=0;i<trans.length;)
  {
  if(!((state.equals(trans[i].state)) &&
       (input.equals(trans[i].input))))
  { i++; }
```

```
  else
  {
  for(j=0;output[j][0] != null;j++)
  {
  if(trans[i].output.equals(output[j][0]))
  {
  System.out.println(output[j][1]);
  }
  state = trans[i].nextstate;
  }
  i = 0;
  System.out.println("Input a 50c, R1 or R2 coin");
  input = kbd.readLine();
  }
  }
  System.out.println("Machine fails invalid coin inserted");
  }
}

class transition
{
  String state, input, output, nextstate;
  public transition(String st,String in,String out,String ne)
  {
  state = st; input = in; output = out; nextstate = ne;
  }
}
```

Problem 6. A *regular expression* is a sequence of symbols that matches words (or strings). A regular expression may match more than one word. Thus regular expressions can describe a language, i.e., words that can be recognized. The rules for matching regular expressions are as follows.

> **Symbol:** For each symbol a in the alphabet, the regular expression a matches a.
>
> **Alternation:** Let M and N be regular expressions. $M|N$ is a regular expression that matches a string if M matches the string or N matches the string. Here | is the special symbol denoting alternation.
>
> **Concatenation:** Let M and N be regular expressions. Then $M \cdot N$ is a regular expression that matches a string $\alpha\beta$ if M matches α and N matches β for some strings α, β. Here $\cdot$ is the special symbol denoting concatenation.
>
> **Epsilon:** The regular expression ϵ matches the empty string.

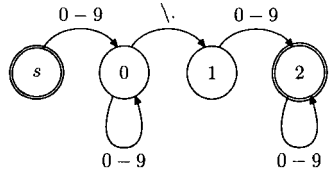
Repetition: The Kleene closure of a regular expression M, M^*, is a regular expression which matches a concatenation of 0 or more strings, where each string is matched by M.

Kleene closure binds tighter than concatentation, and concatenation binds tighter than alternation. Parenthesis may be used to indicate the order in which the operators should be applied. We use the following shorthand to simplify regular expressions:

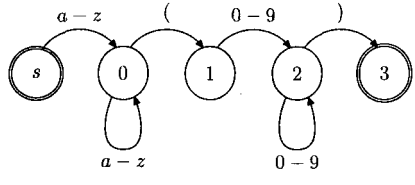
- $\mathbf{a}^+ := \mathbf{aa}^*$
- $\mathbf{a?} := (\mathbf{a}|\epsilon)$
- $[a_1 a_2 a_3 \ldots a_n] := a_1|a_2|\cdots|a_n$
- **[a-z]:=[abc...z]**, **[0-9]:=[0123456789]**
- . matches any symbol.
- $\backslash\alpha$ matches α.

Regular expressions can be transformed into deterministic finite automata (and automatically). Write regular expressions and give the corresponding finite automata for the following:
(i) Positive real numbers of the form $a.b$ where a and b are positive integers. Examples include `1.0 000.000 090.050`. The integer must have at least one digit, and may have preceding zeros.
(ii) Functions of the form $f(a)$ where a is an integer, and f is a series of alphanumerical characters. f must consist of at least one character, and a is of the same form as in (i).

Solution 6. (i) We have `[0-9]+\.[0-9]+`



(ii) We have `[a-z]+\([0-9]+\)`



Chapter 8

Lists, Trees and Queues

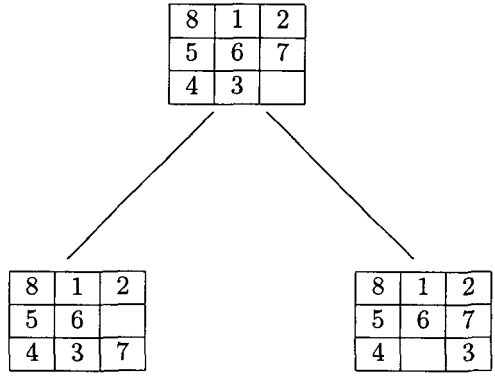
Problem 1. Consider the eight puzzle depicted in the figure below. This is a familiar game with eight tiles arranged in a 3-by-3 configuration with one open square. Tiles can be moved up, down, left or right into the open square, creating a new open square in the space that is vacated. The problem is to find a sequence of moves that convert the initial scrambled configuration of tiles into a goal configuration in which the tiles are arranged in numerical order. The brute force method would be to use trees to obtain all possible configurations.

6	1	2
5	8	7
4	3	

initial configuration

1	2	3
8		4
7	6	5

goal configuration



Write a C++ program which finds the solution by expanding the tree.

Solution 1. We provide two possible solutions.

```
// tile1.cpp

#include <iostream>
using namespace std;

char goal[3][3] = {{'1','2','3'},{'8',' ','4'},{'7','6','5'}};

struct treenode
{
  char b[3][3];
  int value; // integer value representing puzzle configuration
  struct treenode *left, *right;
} *treeroot = NULL;

int puzzlevalue(char b[3][3])
{
  int k = 1;
  int value = 0;
  for(int i=0;i<3;i++)
  for(int j=0;j<3;j++)
  {
  if(b[i][j]!=' ') value += k*(b[i][j]-'0');
```

```
  k *= 10;
  }
  return value;
}

void newnode(struct treenode *&t,char b[3][3],int value)
{
  t = new struct treenode;
  t -> left = t -> right = NULL;
  for(int i=0;i<3;i++)
  for(int j=0;j<3;j++) t -> b[i][j] = b[i][j];
  t -> value = value;
}

int addnode(struct treenode *&t,char b[3][3],int value)
{
  if(t == NULL) { newnode(t,b,value); return 1; }
  if(value < t->value) return addnode(t->left,b,value);
  if(value > t->value) return addnode(t->right,b,value);
  return 0;
}

void swap(char &a,char &b) { a^=b; b^=a; a^=b; }

void print(char b[3][3])
{
  for(int i=0;i<3;i++)
  {
  for(int j=0;j<3;j++) cout << b[i][j] << " ";
  cout << endl;
  }
  cout << endl;
}

int eight(char b[3][3],int x,int y)
{
  int i,j, g = 0;
  for(i=0;i<3;i++)
  for(j=0;j<3;j++)
  if(b[i][j]==goal[i][j]) g++;
  if(g==9) { print(b); return 1; }
  if(x > 0)
  {
  swap(b[x][y],b[x-1][y]); g=0;
```

```
  if(addnode(treeroot,b,puzzlevalue(b))) g = eight(b,x-1,y);
  swap(b[x][y],b[x-1][y]);
  if(g) { print(b); return 1; }
  }
  if(y > 0)
  {
  swap(b[x][y],b[x][y-1]); g=0;
  if(addnode(treeroot,b,puzzlevalue(b))) g = eight(b,x,y-1);
  swap(b[x][y],b[x][y-1]);
  if(g) { print(b); return 1; }
  }
  if(x<2)
  {
  swap(b[x][y],b[x+1][y]); g = 0;
  if(addnode(treeroot,b,puzzlevalue(b))) g = eight(b,x+1,y);
  swap(b[x][y],b[x+1][y]);
  if(g) { print(b); return 1; }
  }
  if(y < 2)
  {
  swap(b[x][y],b[x][y+1]); g = 0;
  if(addnode(treeroot,b,puzzlevalue(b))) g = eight(b,x,y+1);
  swap(b[x][y],b[x][y+1]);
  if(g) { print(b); return 1; }
  }
  return 0;
}

int main(void)
{
  char b[3][3] = {{'6','1','2'},{'5','8','7'},{'4','3',' '}};

  if(eight(b,2,2)) cout << "The puzzle is solved" << endl;
  else cout << "The puzzle could not be solved" << endl;
  return 0;
}
```

A shorter solution is given by

```
// tile2.cpp

#include <iostream>
using namespace std;
```

```
void displaytiles(int level,int* tiles)
{
  cout << "\nLevel " << level << endl;
  for(int i=0;i<9;i++)
  {
  if(i%3 == 0) cout << endl;
  cout << tiles[i] << " ";
  }
  cout << endl;
}

void swap(int *i1,int *i2)
{ int temp = *i1; *i1 = *i2; *i2 = temp; }

void nextnode(int *tiles,int open,int level,int max)
{
  int row = open/3;  int col = open%3;
  int *o = tiles + open;
  displaytiles(level,tiles);
  if(level >= max) return;
  level++;
  if(row > 0)
  {
  swap(o,o-3);
  nextnode(tiles,open-3,level,max);
  swap(o,o-3);
  }
  if(row < 2)
  {
  swap(o,o+3);
  nextnode(tiles,open+3,level,max);
  swap(o,o+3);
  }
  if(col > 0)
  {
  swap(o,o-1);
  nextnode(tiles,open-1,level,max);
  swap(o,o-1);
  }
  if(col < 2)
  {
  swap(o,o+1);
  nextnode(tiles,open+1,level,max);
  swap(o,o+1);
```

```
  }
}

int main(void)
{
  int tiles[9];
  int open, maxlevel;
  cout << "\nEnter a pattern: \n";
  for(int i=0;i<9;i++)
  {
  cin >> tiles[i];
  if(tiles[i]==0) open = i;
  }
  cout << "\n\nHow many levels? \n";
  cin >> maxlevel;
  nextnode(tiles,open,0,maxlevel);
  return 0;
}
```

Problem 2. An *intrusive linked list* is a linked list where the data is aware of the list structure. Thus the data in the intrusive linked list are actually the linked list nodes. Implement a heterogeneous intrusive linked list which can accept nodes of any type for insertion into the list. Thus it is necessary to define a node. Consequently, we have a more general implementation than that of a homogeneous intrusive linked list where all the nodes are of the same type.

Solution 2. The class `List` must be a `friend` to use the functions `insertafter()` and `insertbefore()`.

```
// intrusive.cpp

#include <iostream>
#include <string>
#include <cassert>
using namespace std;

// Node in a circular linked list
class Node
{
  public:
     Node(void);
     Node *next(void);
```

```
    Node *previous(void);
    int operator == (Node&);
    int operator != (Node&);
    virtual void display(ostream&);
  protected:
    void insertafter(Node&);
    void insertbefore(Node&);
    void remove(void);
  private:
    Node *nextnode;
    Node *prevnode;
    // forward declaration of class List
    friend class List;
};

Node::Node(void) { nextnode = prevnode = this; }

Node *Node::next(void)
{ assert(nextnode != NULL); return nextnode; }

Node *Node::previous(void)
{ assert(prevnode != NULL); return prevnode; }

void Node::insertafter(Node &n)
{
  assert(nextnode != NULL);
  assert(prevnode != NULL);
  n.nextnode = nextnode;
  n.prevnode = this;
  nextnode = nextnode -> prevnode = &n;
}

void Node::insertbefore(Node &n)
{
  assert(nextnode != NULL);
  assert(prevnode != NULL);
  n.nextnode = this;
  n.prevnode = prevnode;
  prevnode = prevnode -> nextnode = &n;
}

void Node::remove()
{
  assert(nextnode != NULL && nextnode != this);
```

```
  assert(prevnode != NULL && prevnode != this);
  prevnode->nextnode = nextnode;
  nextnode->prevnode = prevnode;
}

int Node::operator == (Node &n)
{ return &n == this; }

int Node::operator != (Node &n)
{ return &n != this; }

void Node::display(ostream &out)
{ out << "(Node " << this << ")"; }

// define a list node for storing arbitrary data
template <class T>
class DataNode : public Node
{
 public:
    DataNode(T);
    virtual void display(ostream&);
 private:
    T data;
};

template <class T>
DataNode<T>::DataNode(T t) : data(t) {}

template <class T>
void DataNode<T>::display(ostream &out) { cout << data; }

// define a list class to denote the first node
// in the circular linked list
class List : public Node
{
  public:
    void insertafter(Node&,Node&);
    void insertbefore(Node&,Node &);
    void remove(Node&);
    virtual void display(ostream&);
};

void List::insertafter(Node& n1,Node& n2)
{
```

```
  Node *n = next();
  while(*n != *this)
  {
  if(*n == n2)
  {
  cerr << "Cannot duplicate a list node, "
       << "nodes should be unique." << endl;
  assert(*n != n2);
  }
  n = n -> next();
  }
  n1.insertafter(n2);
}

void List::insertbefore(Node &n1,Node &n2)
{
  Node *n = next();
  while(*n != *this)
  {
  if(*n == n2)
  {
  cerr << "Cannot duplicate a list node, "
       << "nodes should be unique." << endl;
  assert(*n != n2);
  }
  n = n -> next();
  }
  n1.insertbefore(n2);
}

void List::remove(Node &n) { n.remove(); }

void List::display(ostream &out)
{
  Node *n = next();
  cout << "List-|" << endl;
  while(*n != *this)
  {
  cout << "     |-> ";
  n -> display(out);
  cout << endl;
  n = n -> next();
  }
}
```

```
ostream& operator << (ostream& out,List& l)
{ l.display(out); return out; }

int main(void)
{
  DataNode<int> n1(1), n2(2), n3(3), n4(4), n5(5);
  DataNode<double> pi(3.14159265), e(2.7182818);
  DataNode<string> scientific("scientific");
  DataNode<string> computing("computing");
  List list;
  list.insertafter(list,n1);
  cout << list;
  list.insertafter(n1,n2);
  list.insertafter(n2,n3);
  list.insertafter(n3,n4);
  list.insertafter(n4,n5);
  cout << list;
  list.insertafter(n2,pi);
  list.insertbefore(n4,e);
  cout << list;
  list.insertafter(pi,scientific);
  list.insertbefore(e,computing);
  cout << list;
  list.remove(n3);
  list.remove(e);
  list.remove(n1);
  cout << list;
  return 0;
}
```

Problem 3. Consider the following problem. A deranged king has decided to choose his successor in a most brutal way. He directs his m knights to sit at his round table. Then he numbers them $1, 2, 3, ..., m$ clockwise. The king then chooses a cycle number, k, and starting with the first knight, counts off $1, 2, 3, ..., k$. The k-th knight is carried off and summarily executed, and the count-off by k continues with the next surviving knight. The process goes on until there is but one survivor. He is the heir to the throne. For example, let $m = 7$ (the number of knights) and $k = 3$. Then the survivor would be knight number 4. This problem can be formulated as a singly linked circular list. Write a C++ program to solve the problem.

Solution 3.

```
// king.cpp

#include <iostream>
using namespace std;

class Knight
{
  public:
    int number;
    Knight* next;
};

int S(int m,int k)
{
  Knight* first = NULL;
  Knight* current = NULL;
  Knight* ptr = NULL;

  for(int i=0;i<m;i++)
  {
  ptr = new Knight;
  if(first == NULL)
  {
  first = ptr; current = ptr;
  current -> number = i + 1;
  }
  else {
  current -> next = ptr;
  current = ptr;
  current -> number = i + 1;
  }
  }

  current -> next = first;
  current = first;
  Knight* prevKnight;
  prevKnight = NULL;

  while(current -> next != current)
  {
  for(int cnt = 0;cnt<k-1;cnt++)
  {
  prevKnight = current;
  current = current -> next;
```

```
  }
  prevKnight -> next = current -> next;
  delete current;
  current = prevKnight -> next;
  }
  return current -> number;
}

int main(void)
{
  int m, k, t;
  cout << "enter number of knights: ";
  cin >> m;
  cout << endl << "enter cycle number: ";
  cin >> k;
  cout << endl << endl;
  cout << "the survivor is knight number: " << S(m,k);
  cout << endl;
  return 0;
}
```

Problem 4. The generalized linked list (i.e., a list which also allows lists as elements) is the basic abstract data type of some languages. The best known example is the language LISP, which stands for LIST PROCESSING. For example

```
(A  ((B  C)  D)  E  (F  G))
```

has four items: the first item is `A`; the second is a sublist made up of the sublist `(B C)` and `D`; the third is `E`; and the fourth is the sublist `(F G)`. The generalized list has proved to be such a useful data structure that it has been made the basis of a powerful language called LISP. Build a storage structure in C and C++ to hold a linked list with sublists. At each link we have two pointers to `Node`.

Solution 4. Some of these nodes look quite conventional: the left part has some literal value, `A`, `B`, `C`, and so forth (called an *atom* in LISP), and the right part contains a pointer. However, other nodes consists of two pointers. The center field of the node is either a character or a nodepointer (`clink` is short for center link). The function we need to make generalized lists is `cons()`, which constructs a list by adding a new first element to (a copy of) an old one. A list can be started by using `cons()` to prefix an atom to `NULL`, the empty list. The function `cons()` needs two arguments:

the first item to be added to the list, and a pointer to that list. Since the first argument (the new first item) can be either an atom or a list, we do not know what type to put in the function heading. Thus we need two `cons()` functions, the one is used when the first node is an atom, and the other one when it is a pointer to a list. Besides `class Node`, the program also introduces `class List`. The functions `cons()`, `car()`, `cdr()` and `copy()` are member functions of `class List`.

```
// lisp.cpp

#include <iostream>
using namespace std;

class Node
{
  public:
    Node();
    Node(const Node&);
    ~Node();
    int atom;
    char info;
    Node* next;
    Node* clink;
};

class List
{
  private:
    Node* head;
  public:
    List();                    // Constructor
    List(const List&);         // Copy constructor
    ~List();                   // Destructor
    List& operator = (const List&); // overloading =
    Node* copy(Node*);
    List cons(char,List&);
    List cons(List&,List&);
    List car(List&);
    List cdr(List&);
    void printNode(Node*);
    void printList();
    void release(Node*);
};
```

```
Node::Node() { atom = 1; info = '\0'; next = clink = NULL; }

Node::~Node() {  }

Node::Node(const Node& nd)
{
  atom = nd.atom;
  info = nd.info;
  next = nd.next;
  clink = nd.clink;
}

List::List() { head = NULL; }

List::List(const List& larg)
{
  if(larg.head == NULL) head = NULL;
  else head = copy(larg.head);
}

List::~List()
{
  if(head) { release(head); }
  delete head;
}

List& List::operator = (const List& larg)
{
  if(larg.head == NULL) head = NULL;
  else head = copy(larg.head);
  return *this;
}

Node* List::copy(Node* narg)
{
  Node* res = new Node;
  if(narg != NULL)
  {
  res->atom = narg->atom;
  if(res->atom)
  {
  res->info = narg->info;
  res->clink = NULL;
  }
```

```
  else res->clink = copy(narg -> clink);
  res->next = copy(narg -> next);
  }
  else res = NULL;
  return res;
}

List List::cons(char newatom,List& oldlist)
{
  List res(oldlist);
  Node* newNode;
  newNode = new Node;
  newNode -> atom = 1;
  newNode -> info = newatom;
  newNode -> clink = NULL;
  newNode -> next = res.head;
  res.head = newNode;
  return res;
}

List List::cons(List& newList,List& oldList)
{
  List res(oldList);
  Node* newNode;
  newNode = new Node;
  newNode -> atom  = 0;
  newNode -> clink = copy(newList.head);
  newNode -> next  = res.head;
  res.head = newNode;
  return res;
}

List List::car(List& oldList)
{
  List res(oldList);
  List empty;
  List carList;
  if(res.head -> clink == NULL)
  carList=cons(res.head->info,empty);
  else carList.head = copy(res.head->clink);
  return carList;
}

List List::cdr(List& oldList)
```

```
{
  List res(oldList);
  res.head = res.head->next;
  return res;
}

void List::printNode(Node* narg)
{
  Node* tmp;
  tmp = copy(narg);
  cout << "(";
  while(tmp != NULL)
  {
  if(tmp->atom) cout << tmp->info << " ";
  else printNode(tmp->clink);
  tmp = tmp->next;
  }
  cout << ")";
}

void List::printList() { printNode(head); }

void List::release(Node* ptr)
{
  Node* tmp;
  {
  if((tmp = ptr -> next) != NULL)
  {
  release(tmp); delete tmp;
  tmp = NULL;
  }
  if((tmp = ptr -> clink) != NULL)
  {
  release(tmp); delete tmp;
  tmp = NULL;
  }
  }
}

int main(void)
{
  List empty;
  List L;
  L = L.cons('G',empty);
```

```
  cout << "L = ";
  L.printList();
  cout << endl;
  List M;
  M = M.cons('E',M.cons(L,empty));
  cout << "M = ";
  M.printList();
  cout << endl;
  List N;
  N = N.cons(N.cons('B',N.cons('C',empty)),N.cons('D',empty));
  cout << "N = ";
  N.printList();
  cout << endl;
  List Z;
  Z = Z.cons('1',Z.cons(Z.cons('2',Z.cons('7', empty)),
      Z.cons('5',empty)));
  cout << "Z = ";
  Z.printList();
  cout << endl;
  List Zcar = Z.car(Z);
  cout << "car (Z) = ";
  Z.printList();
  return 0;
}
```

Problem 5. Consider the *producer-consumer problem.* A number of *producers* create objects which are used by consumers. All the objects are stored in the same place until they are needed. If the storage place is full, a producer must wait until there is place. Similarly, if a consumer requires an object and the storage place is empty, the consumer must wait until a producer places an object in the storage place. A data structure for this task is a *queue.* A queue is a first in first out structure. Data can only be added to the back of queue, and removed only from the front of the queue.

(i) Implement a queue data type for Java.
(ii) Implement a producer and consumer using the `Thread` class of Java and the `Queue` class.. Use `synchronized` methods and the methods

```
public final void wait() throws InterruptedException
public final void notify()
public final void notifyAll()
```

to ensure that the queue works correctly in a multi-threaded environment. The method `wait()` causes a thread to wait on the associated object. Simi-

larly the methods `notify()` and `notifyAll()` notify a thread or all threads respectively, which are waiting on the associated object, to resume execution.

Solution 5. (i) The Queue class is given by

```
// Queue.java

import java.lang.Exception;

class QueueElement
{
  private Object o;
  private QueueElement n;
  private boolean nvalid=false;
  public QueueElement(Object ob) { o=ob; }
  public QueueElement next() throws Exception
  {
  if(nvalid) return n;
  else throw(new Exception("No next element"));
  }
  public void setnext(QueueElement nx) { n=nx; nvalid=true; }
  public void unsetnext() { nvalid=false; }
  public Object data() { return o; }
}

public class Queue
{
  private QueueElement front;
  private QueueElement back;
  private int size=0;
  private int maxsize=0;
  public Queue(int n) {maxsize=n;}
  public synchronized void add(Object o) throws Exception
  {
  while((size>=maxsize) && (maxsize>0)) this.wait();
  if(size==0) { front=back=new QueueElement(o); size++; }
  else
  {
  back.setnext(new QueueElement(o));
  back=back.next();
  size++;
  }
  this.notify();
```

```
  }

  public synchronized Object remove() throws Exception
  {
  while(size==0) this.wait();
  if(size==0) throw(new Exception("Queue is empty."));
  Object r=front.data();
  if(size>1) front=front.next();
  size--;
  this.notify();
  return r;
  }

  public int Size() { return size; }
}
```

(ii) The Producer and Consumer classes are given by

```
// Producer.java

import java.lang.Exception;

public class Producer extends Thread
{
  String s;
  Queue q;

  public Producer(String s,Queue q) { this.s=s; this.q=q; }

  public void run()
  {
  for(int i=0;i<10;i++)
  {
  String p=s+" "+Integer.toString(i);
  try { q.add(p); }
  catch(Exception e) { System.out.println(e.getMessage()); }
  }
  }
}

// Consumer.java

import java.lang.Exception;

public class Consumer extends Thread
```

```
{
  String s;
  Queue q;

  public Consumer(String s,Queue q) { this.s=s; this.q=q; }

  public void run()
  {
  while(true)
  {
  try
  {
  String c = s + " " + (String) q.remove();
  System.out.println(c);
  } catch(Exception e) { System.out.println(e.getMessage()); }
  }
  }
}
```

The program `PCMain.java` uses these classes.

```
// PCMain.java

import java.lang.Exception;

public class PCMain
{
  public static void main(String[] args)
  {
  Queue q = new Queue(3);
  Producer p1 = new Producer("p1",q);
  Producer p2 = new Producer("p2",q);
  Producer p3 = new Producer("p3",q);
  Producer p4 = new Producer("p4",q);
  Producer p5 = new Producer("p5",q);
  Consumer c1 = new Consumer("c1",q);
  Consumer c2 = new Consumer("c2",q);
  p1.start(); p2.start(); p3.start();
  p4.start(); p5.start();
  c1.start(); c2.start();
  }
}
```

Problem 6. The standard template library of C++ does not provide any template with `Tree` in their name. However, some of its containers — the `set<T>`. `map<T>`, `multiset<T>`, and `multimap<T>` templates — are generally built using a special kind of self-balancing binary search tree called a red-black tree. A self-balancing binary search tree ensures that the tree is always as balanced as possible, so that searches take $O(\log_2 n)$ time. A *sparse matrix* is a matrix with a large number of zero entries, which makes it worthwhile for storage to exploit the presence of these zeros by not storing them. Consider the following data structure for an $n \times n$ sparse matrix M with `double` components `map<pair<int,int>,double> M;`. Implement the matrix

$$M = \begin{pmatrix} 2.2 & 0 & 1.7 \\ 0 & 0 & 0 \\ 0 & 2.3 & 0 \end{pmatrix}$$

using this data structure.

Solution 6.

```
// MapPair.cpp

#include <iostream>
#include <map>
using namespace std;

int main(void)
{
  pair<int,int> i00(0,0);
  pair<pair<int,int>,double> v00(i00,2.2);
  map<pair<int,int>,double> M;
  M.insert(v00);
  cout << M[i00] << endl;
  pair<int,int> i02(0,2);
  pair<pair<int,int>,double> v02(i02,1.7);
  M.insert(v02);
  cout << M[i02] << endl;
  pair<int,int> i21(2,1);
  pair<pair<int,int>,double> v21(i21,2.3);
  M.insert(v21);
  cout << M[i21] << endl;
  double r = M[i00]*M[i21];
  cout << "r = " << r << endl;
  return 0;
}
```

Problem 7. The standard least recently used algorithm (*LRU algorithm*) keeps in the cache the data that has been used most recently, on the assumption that future accesses will mirror past accesses. Write a C++ program that implements the LRU algorithm using the `list` class of the standard template library and the `string` class.

Solution 7. We keep `m = 3` strings in the cache.

```
// LRUAlg.cpp

#include <iostream>
#include <string>
#include <list>
#include <algorithm>
using namespace std;

template <typename T> list<T>& LRU(const T& d)
{
  static int m = 3;
  static list<T> li;
  list<T>::iterator ptr = find(li.begin(),li.end(),d);
  if(ptr != li.end()) li.erase(ptr);
    else if(li.size() == m) li.pop_back();
  li.push_front(d);
  return li;
}

int main(void)
{
  while(true) {
   string d;
   cin >> d;
   list<string>& li = LRU(d);
   list<string>::iterator itr;
   for(itr=li.begin();itr!=li.end();itr++)
   cout << *itr << " ";
   cout << endl;
  }
  return 0;
}
```

Chapter 9

Numerical Techniques

Problem 1. A polynomial of the form

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

can be evaluated efficiently using *Horner's rule*. This rule avoids explicit computation of the x^i terms as follows

$$p(x) = a_0 + x(a_1 + x(a_2 + x(\ldots(a_{n-1} + xa_n)\ldots))) .$$

Complete the definition of the recursive method `horner()` in a Java program with header

```
public static double horner(double[] a,double x,int i)
```

where the first parameter is an array of coefficients of the polynomial, the second is the value of x at which the polynomial is to be evaluated and `int i` is a marker we use to keep track of the recursion.

Solution 1. The command `a.length` finds the length of the array `a`.

```
// Horner.java

import java.io.*;

public class Horner
{
  public static double horner(double[] a,double x,int i)
```

```
  {
  final int m = a.length-1;
  if(i == m) return a[m];
  else return a[i] + x*horner(a,x,i+1);
  }

  public static void main(String[] args)
  {
  int d = 3;
  double[] a = new double[d];
  a[0] = 2.0; a[1] = 3.0; a[2] = 5.0;
  double x = 1.5;
  int i = 0;
  double result = horner(a,x,i);
  System.out.println("result = " + result);
  }
}
```

Problem 2. Suppose we have n measured values that are Cartesian coordinates of the form (x_j, y_j) $(j = 0, 1, \ldots, n-1)$ to which we would like to fit a straight line of the form

$$y(x) = a + bx$$

where the coefficients a and b are parameters. The values of these parameters are to be determined by the *least-square method* that minimizes the sum of the squares of the difference between the computed values and the measured values; i.e., it minimizes the function

$$f(a,b) = \sum_{j=0}^{n-1} (y(x_j) - y_j)^2 \equiv \sum_{j=0}^{n-1} (a + bx_j - y_j)^2$$

with respect to the coefficients a and b.
(i) Find the equations for a and b.
(ii) Give a C++ implementation. Recall that both the equations for a and b include the summation over x_j and y_j.

Solution 2. (i) Differentiating the function f with respect to a and b and setting the derivative equal to zero yields

$$\frac{\partial f}{\partial a} = 0 \Rightarrow \sum_{j=0}^{n-1} (a + bx_j - y_j) = 0 \tag{1}$$

and

$$\frac{\partial f}{\partial b} = 0 \Rightarrow \sum_{j=0}^{n-1} x_j(a + bx_j - y_j) = 0\,. \tag{2}$$

Thus we obtain from (1)

$$an + b\sum_{j=0}^{n-1} x_j = \sum_{j=0}^{n-1} y_j\,. \tag{3}$$

We obtain from (2)

$$a\sum_{j=0}^{n-1} x_j + b\sum_{j=0}^{n-1} x_j^2 = \sum_{j=0}^{n-1} x_j y_j\,. \tag{4}$$

Solving (3) and (4) with respect to a and b yields

$$a = \frac{\sum_{j=0}^{n-1} y_j - b\sum_{j=0}^{n-1} x_j}{n}, \quad b = \frac{n\sum_{j=0}^{n-1} x_j y_j - \sum_{j=0}^{n-1} x_j \sum_{j=0}^{n-1} y_j}{n\sum_{j=0}^{n-1} x_j^2 - (\sum_{j=0}^{n-1} x_j)^2}\,.$$

The line of best fit $y = a + bx$ always passes through the *center of gravity* of the data points, since

$$\bar{x} := \frac{1}{n}\sum_{j=0}^{n-1} x_j, \qquad \bar{y} := \frac{1}{n}\sum_{j=0}^{n-1} y_j$$

and $a = \bar{y} - b\bar{x}$. The root mean square error is an approximation of the standard deviation of the fit defined by

$$\sigma := \left(\frac{1}{n}\sum_{j=0}^{n-1}(y_j^* - a - bx_j)^2\right)^{1/2}$$

where y_j^* is the true value at x_j of the function underlying the experimental data.

(ii) The C++ program is given by

```
// leastsquare.cpp

#include <iostream>
using namespace std;

void leastsquare(double* x,double* y,int n,double* a,double* b)
{
  double sum_x = 0.0, sum_y = 0.0, sum_xy = 0.0, sum_x2 = 0.0;
```

```
  for(int i=0;i<n;i++)
  {
  sum_x += x[i];
  sum_y += y[i];
  sum_xy += x[i]*y[i];
  sum_x2 += x[i]*x[i];
  }
  *b = (n*sum_xy - sum_x*sum_y)/(n*sum_x2 - sum_x*sum_x);
  *a = (sum_y - (*b)*sum_x)/n;
}

int main(void)
{
  int n = 7;
  double* x = new double[n];
  double* y = new double[n];
  x[0] = 2.0; y[0] = 0.7;
  x[1] = 3.0; y[1] = 0.8;
  x[2] = 4.0; y[2] = 1.1;
  x[3] = 5.0; y[3] = 1.2;
  x[4] = 6.0; y[4] = 1.6;
  x[5] = 7.0; y[5] = 1.7;
  x[6] = 8.0; y[6] = 2.0;
  double a = 0.0;
  double b = 0.0;
  leastsquare(x,y,n,&a,&b);
  cout << "a = " << a << "    " << "b = " << b;
  delete[] x; delete[] y;
  return 0;
}
```

Problem 3. The *bisection method* is defined as follows. Let $a < b$. Let f be a smooth function in the interval $[a, b]$. Assume that $f(a)$ and $f(b)$ have opposite signs. The function f must cross the x-axis at some point within the interval $[a, b]$. We assume that the function f crosses the x-axis only once. Thus the equation

$$f(x) = 0$$

has a root in the interval $[a, b]$. This root can be found by repeatedly halving the interval. Denoting the mid-point of $[a, b]$ by m the root will be in $[a, m]$ if $f(a)$ and $f(m)$ have opposite signs. Otherwise, $f(m)$ and $f(b)$ will have opposite signs and the root will lie in $[m, b]$. Thus, the half-interval containing the root can be determined by simply comparing the sign of $f(m)$ with the sign of $f(a)$ or $f(b)$. The process can then be

repeated for the smaller interval containing the root. In this way a sequence of mid-points is generated that converges to the root. Convergence is not as rapid as in some other methods. Give a C++ implementation of the bisection method. Consider the function

$$f(x) = \cos(x) - \frac{x}{4}$$

and the interval $[0.5, 2.0]$.

Solution 3. We have

$$f(0.5) = 0.75258\ldots > 0 \quad \text{and} \quad f(2.0) = -0.5411468\ldots < 0\,.$$

Thus the bisection method can be applied. The function f crosses the x-axis only once.

```
// bisection.cpp

#include <iostream>
#include <cmath>
#include <cassert>
using namespace std;

// bisection method to find the root within accuracy
// eps for f(x) between a and b where f(a)*f(b)<=0

double bisec(double (*f)(double),double a,double b,double eps)
{
  double m, fm, fa = f(a), fb = f(b);

  assert(fa*fb <= 0);
  while(fabs(a-b) > eps)
  {
  m = (a+b)/2.0;
  fm = f(m);
  if(fa*fm < 0.0) { b = m; fb = fm; }
  else            { a = m; fa = fm; }
  }
  return a;
}

double f(double x) { return cos(x)-x/4.0; }

int main(void)
{
```

```
  cout << "m = " << bisec(f,0.5,2.0,0.000005) << endl;
  return 0;
}
```

Problem 4. The *secant method* is defined as follows. Let f be a differentiable function in an interval $[a, b]$. Assume that f has exactly one root α in this interval, i.e.,

$$f(\alpha) = 0\,.$$

At each step we obtain a new approximation to the root of our equation by approximating the function with a straight line. In this case, we use the straight line (the secant) that passes through the curve at the points corresponding to the two most recent approximations to the root. The equation of the line passing through the two points

$$(x_{i-1}, f(x_{i-1})) \quad \text{and} \quad (x_i, f(x_i))$$

is

$$y = f(x_i) + \frac{f(x_{i-1}) - f(x_i)}{x_{i-1} - x_i}(x - x_i)$$

and this crosses the x-axis at the point

$$x_{i+1} = x_i - f(x_i)\frac{x_{i-1} - x_i}{f(x_{i-1}) - f(x_i)}\,.$$

This process is very similar to that used in the Newton-Raphson method. The gradient of the line passing through two points on the curve is used in place of the gradient of the tangent at x_i. If the function f is difficult to differentiate, we can use the secant method as an alternative to the Newton-Raphson method. Give a C++ implementation of the secant method. Apply it to $f(x) = 0$, where

$$f(x) = \cos(x) - \frac{x}{4}$$

and $a = 1.4$, $b = 1.5$.

Solution 4.

```
// secant.cpp

#include <iostream>
#include <cmath>
#include <cassert>
using namespace std;
```

```
// secant method to find the root within accuracy p for f(x)
// where x1 and x2 are starting points

double secant(double (*f)(double),double x1,double x2,double p)
{
  double x, fx1 = f(x1), fx2 = f(x2);

  while(fabs(x1-x2) > p)
  {
  x  = x2 - fx2*(x1-x2)/(fx1-fx2);
  x1 = x2; fx1 = fx2;
  x2 = x;  fx2 = f(x2);
  }
  return x1;
}

double f(double x) { return cos(x)-x/4.0; }

int main(void)
{
  cout << "x = " << secant(f,1.5,1.4,0.000005) << endl;
  return 0;
}
```

Problem 5. Consider the equation

$$f(x) = 0 \tag{1}$$

where it is assumed that f is at least twice differentiable. Let I be some interval containing a root of f. The *Newton-Raphson method* can be derived by taking the tangent line to the curve $y = f(x)$ at the point $(x_n, f(x_n))$ corresponding to the current estimate, x_n, of the root. The intersection of this line with the x-axis gives the next estimate to the root, x_{n+1}. The gradient of the curve $y = f(x)$ at the point $(x_n, f(x_n))$ is $f'(x_n)$. The tangent line at this point has the form $y = f'(x_n)x + b$. Since it passes through $(x_n, f(x_n))$ we see that $b = f(x_n) - x_n f'(x_n)$. Therefore the tangent line is $y = f'(x_n)x + f(x_n) - x_n f'(x_n)$. To determine where this line cuts the x-axis, we set $y = 0$. Taking this point of intersection as the next estimate, x_{n+1}, to the root we have $0 = f'(x_n)x_{n+1} + f(x_n) = -x_n f'(x_n)$. It follows that

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}\,. \tag{2}$$

This is the Newton-Raphson method. This method has the form "next estimate = current estimate + correction term". The correction term is $-f(x_n)/f'(x_n)$ and this must be small when x_n is close to the root if convergence is to be achieved. This will depend on the behavior of $f'(x)$ near the root and, in particular, difficulty will be encountered when $f'(x)$ and $f(x)$ have roots close together. Since the method is of the form $x_{n+1} = g(x_n)$ with $g(x) = x - f(x)/f'(x)$ the order of the method can be examined. Differentiating g leads to $g'(x) = (f(x)f''(x))/(f'(x))^2$. For convergence we require that

$$\left| \frac{f(x)f''(x)}{(f'(x))^2} \right| < 1 \tag{3}$$

for all x in some interval I containing the root. Since $f(a) = 0$, the above condition is satisfied at the root $x = a$ provided that $f'(a) \neq 0$. Then, provided that $g(x)$ is continuous, an interval I must exist in the neighbourhood of the root and over which (3) is satisfied. Difficulty is sometimes encountered when the interval I is small because the initial guess must be taken from the interval. This usually arises when $f(x)$ and $f'(x)$ have roots close together since the correction term is inversely proportional to $f'(x)$. Give a C++ implementation of the Newton-Raphson method in one dimension. Apply it to $f(x) = 0$, where

$$f(x) = \ln(x)\,.$$

Solution 5. Obviously we must have $x_0 > 0$. For example, the initial values $x_0 = 0.5$ and $x_0 = 2.0$ approaches the solution 1. For $x_0 = 3.0$ the program gives as root -0.295837. Explain why!

```
// onenewton.cpp

#include <iostream>
#include <cmath>
using namespace std;

// one dimensional Newton's method to find roots of f
// with derivative df for initial value x, and accuracy eps

double newton(double (*f)(double),double (*df)(double),
              double x,double eps)
{
  double x0, x1 = x;
  do
  {
  x0 = x1;
```

```
  x1 = x0 - f(x0)/df(x0); }
  while(fabs(x1-x0) > eps);
  return x0;
}

double f(double x) { return log(x); }

double df(double x) { return 1.0/x; }

int main(void)
{
  double x;
  cout << "Enter the initial value (greater than 0): ";
  cin  >> x;
  cout << "Approximation = " << newton(f,df,x,0.00001) << endl;
  return 0;
}
```

Problem 6. *Halley's method* is a third order method for finding roots of a function f, i.e., we want to find a x^* such that $f(x^*) = 0$. There are two method's called Halley's method. One is called the irrational method and a simpler one is called the rational method. We consider the rational method. Taylor expansion of $f(x)$ around x_n gives

$$f(x) = f(x_n) + f'(x_n)(x - x_n) + \frac{1}{2}f''(x_n)(x - x_n)^2 + \cdots$$

Taking into account up to second order derivatives and $f(x) = 0$ we obtain

$$f(x_n) + f'(x_n)(x - x_n) + \frac{1}{2}f''(x)(x - x_n)^2 = 0\,.$$

Thus

$$x = x_n - \frac{f(x_n)}{f'(x_n) + \frac{1}{2}f''(x_n)(x - x_n)}\,.$$

Using the result from Newton's second order method

$$x = x_n - \frac{f(x_n)}{f'(x_n)}$$

we obtain with $x = x_{n+1}$ Halley's rational iteration formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}\left(1 - \frac{f(x_n)f''(x_n)}{2(f'(x_n))^2}\right)^{-1}$$

which can be written as

$$x_{n+1} = x_n - u_n \left(1 - \frac{u_n v_n}{2}\right)^{-1}$$

where

$$u_n = \frac{f(x_n)}{f'(x_n)}, \qquad v_n = \frac{f''(x_n)}{f'(x_n)}.$$

Write a C++ implemenation of Halley's rational method for the equation

$$\sin(2x) = \sinh(x).$$

Use the initial values 1.0 and 2.0.

Solution 6. For the initial value 1.0 we obtain the root 0.870998, whereas for the initial value 2.0 we obtain the root 0.0.

```
// Halley.cpp

#include <iostream>
#include <cmath>
using namespace std;

double f(double x) { return sin(2.0*x) - sinh(x); }
double fd(double x) { return 2.0*cos(2.0*x) - cosh(x); }
double fdd(double x) { return -4.0*sin(2.0*x) - sinh(x); }

int main(void)
{
  double eps = 0.00001;
  double x0, x1;
  x0 = 1.0;

  do
  {
  x1 = x0;
  double u = f(x1)/fd(x1);
  double v = fdd(x1)/fd(x1);
  double b = 1.0 - u*v/2.0;
  x0 = x1 - u/b;
  }
  while(fabs(x1-x0) > eps);
  cout << "x0 = " << x0;
  return 0;
}
```

Problem 7. We consider the system of (nonlinear) equations

$$f_1(x_1, x_2, \ldots, x_n) = 0, \quad \ldots \quad , f_n(x_1, x_2, \ldots, x_n) = 0$$

where we assume that the functions f_j are twice differentiable. We denote by $\mathbf{x}$ the vector $(x_1, x_2, \ldots, x_n)$ and by $\mathbf{f}$ the vector with components $(f_1, f_2, \ldots, f_n)$. Then the system can be written as one vector equation $\mathbf{f}(\mathbf{x}) = \mathbf{0}$. If we take the gradients of the components, we obtain an $n \times n$ function matrix called the *Jacobian matrix*. It is defined as follows

$$\mathbf{J}(\mathbf{f}(\mathbf{x})) = \left(\frac{\partial f_j}{\partial x_k}\right) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & & & \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}.$$

The Newton method in higher dimensions works as follows. Suppose that the vector $\mathbf{y}$ is the exact solution and that our present approximation $\mathbf{x}$ can be written as $\mathbf{x} = \mathbf{y} + \mathbf{h}$. We compute $f_j(x_1, \ldots, x_n)$ and call these values g_j. Thus $f(\mathbf{y} + \mathbf{h}) = \mathbf{g}$. Expanding this in a Taylor series, we find

$$f_j(y_1, y_2, \ldots, y_n) + \sum_{k=1}^{n} h_k \frac{\partial f_j}{\partial y_k} + (\text{higher order terms}) = g_j\,.$$

The first terms are zero by definition, and neglecting higher order terms we obtain with $J_{jk} = (\partial f_j / \partial x_k)_{\mathbf{x}=\mathbf{y}}$

$$\mathbf{J}\mathbf{h} = \mathbf{g} \Leftrightarrow \mathbf{h} = \mathbf{J}^{-1}\mathbf{g}$$

where we suppose that the matrix $\mathbf{J}$ is nonsingular. The vector $\mathbf{h}$ found in this way in general does not give us an exact solution. We arrive at the iteration formula

$$\mathbf{x}^{(m+1)} = \mathbf{x}^{(m)} - \mathbf{J}^{-1}\mathbf{f}(\mathbf{x}^{(m)})\,.$$

The matrix $\mathbf{J}$ changes from step to step. This suggest a simplification to reduce computational efforts replace $\mathbf{J}(\mathbf{x}^{(m)})$ by $\mathbf{J}(\mathbf{x}^{(0)})$. Write a C++ program that finds the roots for the system of equations

$$f_1(\mathbf{x}) = x_1^2 + x_2^2 - 1, \qquad f_2(\mathbf{x}) = x_1 - x_2\,.$$

Solution 7. Since

$$\frac{\partial f_1}{\partial x_1} = 2x_1, \quad \frac{\partial f_1}{\partial x_2} = 2x_2, \quad \frac{\partial f_2}{\partial x_1} = 1, \quad \frac{\partial f_2}{\partial x_2} = -1$$

we obtain

$$J = \begin{pmatrix} 2x_1 & 2x_2 \\ 1 & -1 \end{pmatrix}, \qquad J^{-1} = \frac{1}{2(x_1 + x_2)} \begin{pmatrix} 1 & 2x_2 \\ 1 & -2x_1 \end{pmatrix}.$$

Thus

$$J^{-1}\mathbf{f} = \frac{1}{2(x_1 + x_2)} \begin{pmatrix} x_1^2 - x_2^2 + 2x_1x_2 - 1 \\ -x_1^2 + x_2^2 + 2x_1x_2 - 1 \end{pmatrix}.$$

The C++ program is

```
// twonewton.cpp

#include <iostream>
#include <cmath>
using namespace std;

void JIf(double& x,double& y)
{
  double t1 = 2.0*(x + y);
  double t2 = x*x - y*y;
  double t3 = 2.0*x*y - 1.0;
  x = x - (t2 + t3)/t1;
  y = y - (-t2 + t3)/t1;
}

int main(void)
{
  double eps = 0.001;
  double x1 = 0.9, y1 = 0.6;
  double x0, y0;

  do
  {
  x0 = x1; y0 = y1;
  JIf(x0,y0);
  x1 = x0; y1 = y0;
  } while(fabs(x1*x1 + y1*y1 - 1.0) > eps);

  cout << "x1 = " << x1 << endl;
  cout << "y1 = " << y1 << endl;
  return 0;
}
```

Problem 8. We want to minimize a smooth function $f : \mathbf{R}^n \to \mathbf{R}$. It is assumed that the *gradient vector* $\nabla f = (\partial f/\partial x_1, \ldots, \partial f/\partial x_n)^T$ and the $n \times n$ *Hessian matrix* $\nabla^2 f(\mathbf{x}) = (\partial^2 f/\partial x_j \partial x_k)$ of the function f exists and can be evaluated analytically. Most of the discrete time iterative methods for this problem involves generating a sequence of search points $\mathbf{x}_k$ via the

iteration procedure

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \eta_k \mathbf{d}_k, \qquad \mathbf{x}_0 = \mathbf{x}(0), \quad k = 0, 1, 2, \ldots$$

where the scalar η_k determines the length of the step to be taken in the direction of the vector $\mathbf{d}_k$ (sometimes called the *learning rate*). In numerical optimization, different techniques for the computation of the parameter η_k and the direction vector $\mathbf{d}_k$ are known. There are four basic methods.

1) For the steepest-descent *gradient method* we define the direction as

$$\mathbf{d}_k := -\nabla f(\mathbf{x}_k) .$$

2) For Newton's method the search direction is determined from the formula

$$\mathbf{d}_k = -(\nabla^2 f(\mathbf{x}_k))^{-1} \nabla f(\mathbf{x}_k) .$$

3) The calculation of the inverse Hessian matrix can be complicated. Thus, we try to approximate it by some $n \times n$ symmetric definite matrix H_k, i.e.,

$$H_k \approx (\nabla^2 f(\mathbf{x}_k))^{-1} .$$

Thus, we obtain the quasi-Newton (variable metric) methods in which the search direction is determined as

$$\mathbf{d}_k = -H_k \nabla f(\mathbf{x}_k) .$$

4) The *conjugate gradient method* computes the actual search direction $\mathbf{d}_k$ as a linear combination of the current gradient vector and the previous search directions. In the simplest form, the search direction is calculated as

$$\mathbf{d}_k = -\nabla f(\mathbf{x}_k) + \beta_k \mathbf{d}_{k-1}, \qquad k = 1, 2, \ldots$$

with $\mathbf{d}_0 = -\nabla f(\mathbf{x}_0)$ and β_k is a scalar parameter that ensures the sequence of vector $\mathbf{d}_k$ satisfies a mutual conjugacy condition.

Consider the function $f : \mathbf{R} \to \mathbf{R}$

$$f(x) = x^4 - x^2 .$$

Thus from

$$\frac{df}{dx} = 0 \Rightarrow 4x^{*3} - 2x^* = 0 \Rightarrow x^*(2x^{*2} - 1) = 0$$

we find the critical points

$$x^* = 0 \Rightarrow f(x^* = 0) = 0 \quad \text{local maximum}$$

and

$$x^* = \pm\frac{1}{\sqrt{2}} \Rightarrow f(x^* = \pm\frac{1}{\sqrt{2}}) = -\frac{1}{4} \quad \text{global minima}$$

For the steepest-descent (gradient) method the iteration is given by

$$x_{k+1} = x_k + \eta_k \left(-\frac{df}{dx}(x = x_k) \right), \qquad k = 0, 1, \ldots .$$

Thus

$$x_{k+1} = x_k - \eta_k(4x_k^3 - 2x_k) = x_k - 2\eta_k x_k(2x_k^2 - 1) .$$

Select the initial value $x_0 = 4.0$.
(i) Iterate for η fixed ($\eta = 0.5, 0.1, 0.01$).
(ii) Iterate for $\eta_{k+1} = \eta_k/1.1$ with $\eta_0 = 0.05, 0.02$.
(iii) Write a C++ program for the case $\eta_{k+1} = \eta_k/1.01$ and $\eta_0 = 0.02$.

Solution 8. (i) For $\eta = 0.5$ we find

```
k    x_k
0    4.0
1    -120.0
2    3.45576E+006
.    ......
```

Thus we have an escaping solution, i.e., the step length $\eta = 0.5$ is to large. For $\eta = 0.1$ we also find an escaping solution. Also this step length is to large. For $\eta = 0.01$ the iteration approaches the solution $x^* = 1/\sqrt{2}$.
(ii) For $\eta = 0.05$ we also find an escaping solution. For $\eta = 0.02$ the iteration does not approach the solution $x^* = 1/\sqrt{2}$ since the step size (learning rate) η decreases too fast. We can avoid this problem when we select $\eta_{k+1} = \eta_k/1.01$ instead of $\eta_{k+1} = \eta_k/1.1$. Then we find the solution $x^* = -1/\sqrt{2}$.
(iii) We have

```
// gradient.cpp

#include <iostream>
using namespace std;

int main(void)
{
  int n = 400;
  double eta = 0.2; // initial value eta
  double x = 4.0;   // initial value x
```

```
  for(int k=0;k<n;k++)
  {
  x = x - 2.0*eta*x*(2.0*x*x - 1.0);
  eta = eta/1.01;
  cout << "k = " << k+1 << "  x = " << x
       << "  eta = " << eta << endl;
  }
  return 0;
}
```

Problem 9. Consider the eigenvalue equation

$$\left(-\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + V(x)\right) u(x) = Eu(x)$$

in the Hilbert space $L_2[0,1]$, where V (potential) and u are real-valued functions and u is normalized, i.e.,

$$\int_0^1 u^2(x)dx = 1\,.$$

We consider the case $V(x) = 0$ and $u(0) = u(1) = 0$. Thus, the lowest eigenvalue is positive and not degenerate. This choice corresponds to a free particle in hard-walled box of unit length. Introducing

$$v(x) := \frac{2mV(x)}{\hbar^2}, \qquad e := \frac{2mE}{\hbar^2}$$

we have

$$\left(-\frac{d^2}{dx^2} + v(x)\right) u(x) = eu(x)\,.$$

The *energy functional* is given by

$$e = \int_0^1 \left(\left(\frac{du}{dx}\right)^2 + v(x)u^2(x)\right)\,.$$

To find numerically the lowest eigenvalue we discretize, i.e.,

$$e = \sum_{j=1}^{N-1} h\left(\frac{(u_j - u_{j-1})^2}{h^2} + v_j u_j^2\right)$$

and

$$\sum_{j=1}^{N-1} hu_j^2 = 1$$

where h is the step length and $u_0 = u_N = 0$. An approximation to the second derivative

$$\hat{H}u = -\frac{d^2}{dx^2}u$$

accurate to order h^2 is given by

$$\hat{H}u \approx \frac{u_{j+1} - 2u_j + u_{j-1}}{h^2}\,.$$

To find the lowest eigenvalue we consider

$$\frac{du}{d\epsilon} = -\hat{H}u$$

where ϵ is a real parameter. The simplest solution (iteration) is given by

$$u^{(n+1)} \sim (1 - \hat{H}\Delta\epsilon)u^{(n)}, \qquad n = 0, 1, 2, \ldots$$

where $\sim$ indicates that $u^{(n+1)}$ has to be normalized. For the initial function $u^{(0)}$ we have to choose a function that is not orthogonal to the exact eigenfunction. Choose

$$u(x) = x(1 - x)\,.$$

Note that this function is not normalized. Write C++ code that implements this algorithm. Discuss.

Solution 9. The function `norm()` normalizes the function u.

```
// lowesteigenvalue.cpp

#include <iostream>
#include <cmath>
using namespace std;

void norm(double* u,int nsteps)
{
  double norm = 0.0;
  int ix;
  for(ix=0;ix<nsteps;ix++)
  {
  norm += pow(u[ix],2.0);
  }
  norm = sqrt((nsteps-1)/norm);
  for(ix=0; ix < nsteps; ix++)
  { u[ix] *= norm; }
}
```

```
int main(void)
{
  int nsteps = 21;                  // lattice parameter
  double* u = new double[nsteps]; // solution array
  double h = 1.0/(nsteps - 1.0);
  double dt = 0.0005;
  double dth = dt/(h*h);

  int ix;
  for(ix=0;ix<nsteps;ix++)
  {
  double x = ix*h;
  u[ix] = x*(1.0 - x);     // initial function for u
  }
  norm(u,nsteps);          // normalize initial u

  for(int iter=1;iter<100;iter++)
  {
  double pold = 0.0; // u[0] = 0
  double pnew;
  for(ix=1;ix<(nsteps-1);ix++)
  {
  // (1 - H*dth)u
  pnew = u[ix] + dth*(pold + u[ix+1] - 2.0*u[ix]);
  pold = u[ix];
  u[ix] = pnew;
  norm(u,nsteps);  // normalize
  }
  // calculation of energy e
  double e = 0.0;
  for(ix=1;ix<nsteps;ix++)
  {
  double temp = u[ix] - u[ix-1];
  e += temp*temp;
  }
  e = e/h;
  cout << " energy = " << e << endl;
  }
  return 0;
}
```

The result is 9.85. This agrees well with the exact eigenvalue

$$e_n = n\pi^2, \qquad n = 1, 2, \ldots$$

For $n = 1$ (ground state) we have $e_1 = \pi^2 = 9.869\ldots$. The exact normalized eigenfunctions are given by

$$u_n(x) = 2^{1/2} \sin(n\pi x) .$$

Problem 10. The p-norm of a vector $(x_0, x_1, \ldots, x_{n-1})^T \in \mathbf{R}^n$ is defined as

$$\|\mathbf{x}\|_p := \left(\sum_{j=0}^{n-1} |x_j|^p \right)^{1/p}$$

where $p \in \mathbf{R}^+$.
(i) Find the norm for $p \to \infty$.
(ii) Discuss the case $p = 1$ and $p = 2$.

Solution 10. (i) Obviously we find

$$\|\mathbf{x}\|_\infty = \max_{0 \le j \le n-1} |x_j| .$$

(ii) For $p = 1$ we have

$$\|\mathbf{x}\|_1 = \sum_{j=0}^{n-1} |x_j|$$

and for $p = 2$ we have the *Euclidean norm*

$$\|\mathbf{x}\|_2 = \left(\sum_{j=0}^{n-1} x_j^2 \right)^{1/2} .$$

Problem 11. Let $p \ge 1$. Let $\mathbf{x}$ and $\mathbf{y}$ be two n-dimensional vectors over the real numbers. A metric (distance) between $\mathbf{x}$ and $\mathbf{y}$ is defined as

$$\|\mathbf{x} - \mathbf{y}\|_p := \left(\sum_{j=0}^{n-1} |x_j - y_j|^p \right)^{1/p} .$$

Write a C++ program with the function

```
double metric(double* v1,double* v2,double p,double n)
```

which calculates the metric for two given vectors $\mathbf{v}_1$ and $\mathbf{v}_2$. Apply the program to

$$\mathbf{v}_1 = (1.0, 2.0, -3.0, 2.1), \qquad \mathbf{v}_2 = (4.0, 3.0, 1.0, -7.0)$$

and $p = 1$ and $p = 2$. For $p \to \infty$ we obtain the metric (max metric)

$$\|\mathbf{x} - \mathbf{y}\|_\infty := \max_{0 \le j \le n-1} |x_j - y_j| .$$

Give a C++ implementation with the function

`double maxmetric(double* v1,double* v2,double n).`

Solution 11.

```
// pdistance.cpp

#include <iostream>
#include <cmath>
using namespace std;

double pmetric(double* v1,double* v2,double p,double n)
{
  double s = 0.0;
  for(int i=0;i<n;i++) { s += pow(fabs(v1[i] - v2[i]),p); }
  s = pow(s,1.0/p);
  return s;
}

double maxmetric(double* v1,double* v2,double n)
{
  double t = fabs(v1[0] - v2[0]);
  for(int i=1;i<n;i++)
  {
  double s = fabs(v1[i] - v2[i]);
  if(s > t) t = s;
  }
  return t;
}

int main(void)
{
  int n = 4;
  double* v1 = new double[n];
  v1[0] = 1.0; v1[1] = 2.0; v1[2] = -3.0; v1[3] = 2.1;
  double* v2 = new double[n];
  v2[0] = 4.0; v2[1] = 3.0; v2[2] = 1.0; v2[3] = -7.0;
  double p = 10.0;
  double distance1 = pmetric(v1,v2,p,n);
```

```
  cout << "distance1 = " << distance1 << endl;

  double distance2 = maxmetric(v1,v2,n);
  cout << "distance2 = " << distance2 << endl;

  delete[] v1; delete[] v2;
  return 0;
}
```

Problem 12. Write a C++ program that computes $\sin(x)$ to at least 5 decimal places for all x using the Taylor expansion series of sine

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots .$$

Use Horner's method and reduction to the domain $0 \leq x \leq \pi/2$.

Solution 12.

```
// sine.cpp

#include <iostream>
#include <cmath>
using namespace std;

double sine(double x)
{
  const double PI = 3.1415926535897932;
  const double TWOPI = 6.2831853071795865;
  const double PIHALF = 1.5707963267948966;
  const int m = 9;
  double s, x2, n, sign;

  // reduce to 0 <= x < 2*PI using sin(x-2*n*PI) = sin(x)
  n = floor(x/TWOPI);
  x = x - TWOPI*n;
  // reduce to 0 <= x <= PI using sin(2*PI-x) = -sin(x)
  if(x > PI) { x = TWOPI - x; sign = -1.0; }
  else sign = 1.0;
  // reduce to 0 <= x <= PI/2 using sin(PI-x) = sin(x)
  if(x > PIHALF) x = PI - x;
  // compute x - x^3/3! + x^5/5! - x^7/7! + x^9/9!
  s = 1.0;
  x2 = x*x;
```

```
  for(int i=m;i>1;i=i-2) { s = 1.0 - (x2/(i*(i-1)))*s; }
  return sign*s*x;
}

int main(void)
{
  double x = 1.0;
  double result1 = sine(x);
  cout << "result1 = " << result1 << endl;
  x = 10.0;
  double result2 = sine(x);
  cout << "result2 = " << result2 << endl;
  x = 100.0;
  double result3 = sin(x);
  cout << "result3 = " << result3 << endl;
  return 0;
}
```

Problem 13. Determine the *Hausdorff distance* $H(A, B)$ between the set of vectors in $\mathbf{R}^3$

$$A := \{ (3,1,4)^T , (1,5,9)^T \}$$

and

$$B := \{ (2,7,1)^T , (8,2,8)^T , (1,8,2)^T \}$$

where

$$\begin{aligned} H(A,B) &:= \max \{ h(A,B) , h(B,A) \} \\ h(A,B) &:= \max_{\mathbf{x} \in A} \left\{ \min_{\mathbf{y} \in B} \{ \|\mathbf{x} - \mathbf{y}\| \right\} \\ h(B,A) &:= \max_{\mathbf{y} \in B} \left\{ \min_{\mathbf{x} \in A} \{ \|\mathbf{y} - \mathbf{x}\| \right\} . \end{aligned}$$

Use the Euclidean distance for $\| \cdot \|$.

Solution 13. We set

$$A := \{ \mathbf{a}_1 = (3,1,4)^T , \mathbf{a}_2 = (1,5,9)^T \}$$

and

$$B := \{ \mathbf{b}_1 = (2,7,1)^T , \mathbf{b}_2 = (8,2,8)^T , \mathbf{b}_3 = (1,8,2)^T \}.$$

We tabulate the Euclidean distances

	$\mathbf{a}_1$	$\mathbf{a}_2$	Minimum
$\mathbf{b}_1$	$\sqrt{46}$	$\sqrt{69}$	$\sqrt{46}$
$\mathbf{b}_2$	$\sqrt{42}$	$\sqrt{59}$	$\sqrt{42}$
$\mathbf{b}_3$	$\sqrt{57}$	$\sqrt{58}$	$\sqrt{57}$
Minimum	$\sqrt{42}$	$\sqrt{58}$	

Thus

$$h(A,B) = \max\{\sqrt{42}, \sqrt{58}\} = \sqrt{58}$$
$$h(B,A) = \max\{\sqrt{46}, \sqrt{42}, \sqrt{57}\} = \sqrt{57}$$

and therefore

$$H(A,B) = \max\{\sqrt{58}, \sqrt{57}\} = \sqrt{58}\,.$$

Problem 14. Assume that a function f has a power series expansion

$$f(x) = \sum_{j=0}^{\infty} c_j x^j\,.$$

The (N,M) *Padé approximant* is given by

$$[N,M](x) := \frac{\det \begin{pmatrix} c_{M-N+1} & c_{M-N+2} & \cdots & c_{M+1} \\ \vdots & \vdots & & \vdots \\ c_M & c_{M+1} & \cdots & c_{M+N} \\ \sum_{j=N}^{M} c_{j-N} x^j & \sum_{j=N-1}^{M} c_{j-N+1} x^j & \cdots & \sum_{j=0}^{M} c_j x^j \end{pmatrix}}{\det \begin{pmatrix} c_{M-N+1} & c_{M-N+2} & \cdots & c_{M+1} \\ \vdots & \vdots & & \vdots \\ c_M & c_{M+1} & \cdots & c_{M+N} \\ x^N & x^{N-1} & \cdots & 1 \end{pmatrix}}\,.$$

(i) Calculate $[1,1](x)$ for

$$f_1(x) = \frac{1}{1+x}\,.$$

(ii) Calculate $[1,1](x)$ for

$$f_2(x) = \sqrt{1+x}\,.$$

(iii) Calculate $[2,2](x)$ for

$$f_3(x) = e^x\,.$$

Solution 14. (i) We have

$$f_1(x) = 1 - x + x^2 - x^3 + \cdots\,.$$

Thus $c_0 = 1$, $c_1 = -1$, $c_2 = 1$ and therefore

$$[1,1](x) = \frac{\det\begin{pmatrix} c_1 & c_2 \\ c_0 x & c_0 + c_1 x \end{pmatrix}}{\det\begin{pmatrix} c_1 & c_2 \\ x & 1 \end{pmatrix}} = \frac{\det\begin{pmatrix} -1 & 1 \\ x & 1-x \end{pmatrix}}{\det\begin{pmatrix} -1 & 1 \\ x & 1 \end{pmatrix}} = \frac{1}{1+x}.$$

(ii) We have

$$\sqrt{1+x} = 1 + \frac{1}{2}x - \frac{1}{8}x^2 + \cdots.$$

Thus $c_0 = 1$, $c_1 = 1/2$, $c_2 = -1/8$ and therefore

$$[1,1](x) = \frac{\det\begin{pmatrix} c_1 & c_2 \\ c_0 x & c_0 + c_1 x \end{pmatrix}}{\det\begin{pmatrix} c_1 & c_2 \\ x & 1 \end{pmatrix}} = \frac{\det\begin{pmatrix} 1/2 & -1/8 \\ x & 1 + x/2 \end{pmatrix}}{\det\begin{pmatrix} 1/2 & -1/8 \\ x & 1 \end{pmatrix}} = \frac{1 + 3x/4}{1 + x/4}.$$

(iii) We have the expansion

$$e^x = 1 + x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + \frac{1}{4!}x^4 + \cdots.$$

Thus

$$c_0 = 1, \quad c_1 = 1, \quad c_2 = \frac{1}{2}, \quad c_3 = \frac{1}{6}, \quad c_4 = \frac{1}{24}$$

and therefore

$$\begin{aligned}
[2,2](x) &= \frac{\det\begin{pmatrix} c_1 & c_2 & c_3 \\ c_2 & c_3 & c_4 \\ c_0 x^2 & c_0 x + c_1 x^2 & c_0 + c_1 x + c_2 x^2 \end{pmatrix}}{\det\begin{pmatrix} c_1 & c_2 & c_3 \\ c_2 & c_3 & c_4 \\ x^2 & x & 1 \end{pmatrix}} \\
&= \frac{\det\begin{pmatrix} 1 & 1/2 & 1/6 \\ 1/2 & 1/6 & 1/24 \\ x^2 & x + x^2 & 1 + x + x^2/2 \end{pmatrix}}{\det\begin{pmatrix} 1 & 1/2 & 1/6 \\ 1/2 & 1/6 & 1/24 \\ x^2 & x & 1 \end{pmatrix}} \\
&= \frac{12 + 6x + x^2}{12 - 6x + x^2}.
\end{aligned}$$

For example

$$[2,2](1) = \frac{19}{7} \approx 2.7142857\ldots.$$

Problem 15. Let f be a real-valued function defined at any point x in the interval (a, b). The derivative of f at $x = x_0$, $x_0 \in (a, b)$ is defined as

$$f'(x) := \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{1a}$$

if this limit exists or as

$$f'(x) := \lim_{h \to 0} \frac{f(x) - f(x-h)}{h} \tag{1b}$$

if this limit exists. From (1a) and (1b) we obtain

$$f'(x) := \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}. \tag{2}$$

For (1a) and (1b) the error is of order h, whereas for (2) the error is of order h^2. Another definition would be

$$f'(x) := \lim_{h \to 0} \frac{-11f(x) + 18f(x+h) - 9f(x+2h) + 2f(x+3h)}{6h}. \tag{3}$$

Equations (1a), (1b), (2) and (3), respectively can be used as an approximation for the derivative with h small. Write a C++ program that implements these approximations for (1a), (1b) and (2) for $h = 0.1, 0.01, 0.001, 0.0001$.

Solution 15. Difficulties can arise as h becomes smaller. In theory, the smaller the value of h is, the more accurate the approximation to $f'(x)$. However, in a program the values obtained may be unreliable because of rounding errors that take place during the evaluation of the differences and the division.

```
// derivative.cpp

#include <iostream>
#include <cmath>
using namespace std;

double f(double x) { return x*x*x; }

// numerical derivative of f at x method 1a
double df1a(double (*f)(double),double x,double h)
{ return (f(x+h) - f(x))/h; }

// numerical derivative of f at x method 1b
double df1b(double (*f)(double),double x,double h)
{ return (f(x) - f(x-h))/h; }
```

```
// numerical derivative of f at x second method
double df2(double (*f)(double),double x,double h)
{ return (f(x+h) - f(x-h))/(2.0*h); }

int main(void)
{
  double x = 2.0, h = 0.1;
  for(int count=1;count<5;count++)
  {
  cout << "h = " << h << endl;
  cout << "Method 1a: df/dx(x=" << x << ") = " << df1a(f,x,h)
       << endl;
  cout << "Method 1b: df/dx(x=" << x << ") = " << df1b(f,x,h)
       << endl;
  cout << "Method 2:  df/dx(x=" << x << ") = " << df2(f,x,h)
       << endl;
  h /= 10.0;
  }
  return 0;
}
```

Problem 16. Let f be a smooth function. Consider the integral

$$I(x_n, x_0) = \int_{x_0}^{x_n} f(x)dx\,.$$

To find an approximation of this integral we can use the *trapezium rule.* The name of this method comes from a simple geometrical interpretation; the area between the curve, the x-axis, and two ordinates is approximated by the area of a trapezoid

$$\int_{x_0}^{x_n} f(x)dx \approx \frac{x_n - x_0}{2}(f(x_n) + f(x_0))\,.$$

This approximation is repeated over several intervals of length h, with $x_k = x_0 + kh$, $k = 0, 1, 2, \ldots, n$. We obtain

$$\int_{x_0}^{x_n} f(x)dx \approx \frac{h}{2}(f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(x_n))\,.$$

We find the truncation error by considering one interval

$$R(h) = \frac{h}{2}\left(f\left(-\frac{h}{2}\right) + f\left(\frac{h}{2}\right)\right) - \int_{-h/2}^{h/2} f(x)dx\,.$$

From dR/dh, d^2R/dh^2 and d^3R/dh^3 we find that $dR(h=0)/dh = 0$, $d^2R(h=0)/dh^2 = 0$ and

$$\frac{d^3R(h=0)}{dh^3} = \frac{1}{2}\frac{d^2f(h=0)}{dh^2}\,.$$

Thus the truncation error is essentially $(h^3/12)d^2f(h=0)/dh^2$. Write a C++ program that implements the trapezium rule.

Solution 16.

```
// trapez.cpp

#include <iostream>
#include <cmath>
using namespace std;

// integrate f on (a,b) using the trapezium rule
// using n steps
double trapezium(double (*f)(double),double a,double b,int n)
{
  double h = (b - a)/n;
  double sum = (f(a) + f(b))/2.0;
  for(int i=1;i<n;i++) sum += f(a+i*h);
  return h*sum;
}

double f(double x) { return exp(-x); }

int main(void)
{
  cout << "The integral of exp(-x) on (0,1) is "
       << trapezium(f,0.0,1.0,100) << endl;
  return 0;
}
```

Problem 17. For using the *Simpson rule* in integrating a smooth function f we divide the interval $[x_0, x_n]$ into sub-intervals $[x_j, x_{j+2}]$ for $j = 0, 2, \ldots, n-2$ and express the integral in the form

$$\int_{x_0}^{x_n} f(x)dx = \sum_{j=0\,(2)\,n-2} \int_{x_j}^{x_{j+2}} f(x)dx\,. \tag{1}$$

On a sub-interval $[x_j, x_{j+2}]$ the integrand f is approximated by the quadratic passing through $(x_j, f(x_j))$, $(x_{j+1}, f(x_{j+1}))$ and $(x_{j+2}, f(x_{j+2}))$. Taking

$x = x_j + rh$, the first three terms of the Gregory-Newton forward formula give

$$f(x) \approx f_j + r\Delta f_j + \frac{r(r-1)}{2}\Delta^2 f_j, \quad x_j \le x \le x_{j+2}, \quad 0 \le r \le 2. \quad (2)$$

Thus the contribution to (1) over $[x_j, x_{j+2}]$ is

$$\begin{aligned}\int_{x_j}^{x_{j+2}} f(x)dx &= \int_0^2 f(x_j + rh)hdr \\ &= \int_0^2 \left(f_j + r\Delta f_j + \frac{(r^2-r)}{2}\Delta^2 f_j\right) hdr + \epsilon_j\end{aligned}$$

where ϵ_j is the local truncation error for the integration formula on $[x_j, x_{j+2}]$. We find that this local truncation error is given by $\epsilon_j = -h^5 f^{(iv)}(\xi_j)/90$, where ξ_j lies in the interval $[x_j, x_{j+2}]$. Thus we find

$$\int_{x_0}^{x_n} f(x)dx = \frac{h}{3}(f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \cdots + 2f_{n-2} + 4f_{n-1} + f_n) + \epsilon_G$$

where $f_j = f(x_j)$ and ϵ_G is the global error. ϵ_G is given by

$$\epsilon_G = -\frac{h^5}{90}(f^{(iv)}(\xi_0) + f^{(iv)}(\xi_2) + \cdots + f^{(iv)}(\xi_{n-2})).$$

Write a C++ program that implements the Simpson rule.

Solution 17.

```
// simpson.cpp

#include <iostream>
#include <cmath>
using namespace std;

// integrate f on (a,b) using the Simpson rule with n steps
double simpson(double (*f)(double),double a,double b,int n)
{
  double h = (b-a)/(2.0*n);
  double sum = f(a) + f(b) + 4.0*f(a+h);
  for(int i=1;i<n;i++)
  sum += 2.0*f(a+2.0*i*h) + 4.0*f(a+h+2.0*i*h);
  return h*sum/3.0;
}
```

```
double f(double x) { return exp(-x); }

int main(void)
{
  cout << "The integral of exp(-x) on (0,1) is "
       << simpson(f,0.0,1.0,100) << endl;
  return 0;
}
```

Problem 18. The *Romberg integration* is a method of obtaining high-order numerical approximations to integrals from the trapezium rule (or Simpson's rule) approximation. To obtain a higher order method based on the trapezium rule, we examine the correction or error term associated with the trapezium rule. The correction term associated with the composite form of the trapezium rule can be expressed in the form

$$C = a_2 h^2 + a_4 h^4 + a_6 h^6 + \cdots \tag{1}$$

where h is the step length and a_2, a_4, a_6, ... are constants. The correction term is an even function of h, i.e., $C(h) = C(-h)$. Since the correction of the trapezium rule is simply the difference between the exact integral and the trapezium rule approximation we have

$$C = \int_{x_0}^{x_n} f(x)dx - h\left(\frac{f_0}{2} + f_1 + f_2 + \cdots + f_{n-1} + \frac{f_n}{2}\right) \tag{2}$$

where $f_j = f(x_j)$. Denoting the exact integral by I and the trapezium rule approximation with the step length h by $T(h)$, from (1) and (2) we may write

$$I = T(h) + a_2 h^2 + a_4 h^4 + a_6 h^6 + \cdots$$

If this is used with step length h and $h/2$ we have $I = T(h) + a_2 h^2 + a_4 h^4 + \cdots$ and

$$I = T(h/2) + a_2 \frac{h^2}{4} + a_4 \frac{h^4}{16} + \cdots . \tag{3}$$

Eliminating the leading contribution to the correction term between these two equations gives

$$I = \frac{4T(h/2) - T(h)}{3} + b_4 h^4 + \cdots \tag{4}$$

where b_4 is a constant. Whereas $T(h)$ and $T(h/2)$ have errors of order h^2, the first term on the right-hand side of (4)

$$T(h, h/2) = \frac{2^2 T(h/2) - T(h)}{2^2 - 1}$$

gives an approximation to I with error of order h^4. Write a C++ program that implements the Romberg integration.

Solution 18. In the program p denotes the accuracy.

```
// romberg.cpp

#include <iostream>
#include <cmath>
#include <vector>
using namespace std;

// integrate f on (a,b) using the Romberg integration
// using n steps
double romberg(double (*f)(double),double a,double b,double p)
{
  int i, j, k, n = 4;
  double sum, h, factor, help, integral;
  vector<vector<double> > trap;

  trap.resize(1);
  trap[0].resize(1);
  h = (b-a)/n;
  // trapezium rule for T(h)
  sum = (f(a) + f(b))/2.0;
  for(i=0;i<n;i++) sum += f(a+i*h);
  trap[0][0] = h*sum;
  j = 0;
  do
  {
  j++;
  trap.resize(j+1);
  trap[0].resize(j+1);
  // trapezium rule for T(h/2) ...
  for(k=0;k<n;k++) sum += f(a+k*h+h/2.0);
  n *= 2; h /= 2.0;
  // T(h), T(h/2), T(h/4), T(h/8), ... , T(h/(2^j)), ...
  trap[0][j] = h*sum;

  factor = 1.0;
  for(i=0;i<j;i++)
  {
  trap[i+1].resize(j+1);
  // T(h,h/2),      T(h/2,h/4),      T(h/4,h/8), ...
```

```
  // T(h,h/2,h/4),  T(h/2,h/4,h/8), ...
  factor *= 4.0;
  trap[i+1][j] = (factor*trap[i][j]-trap[i][j-1])/(factor-1.0);
  }
  // repeat until the change in the approximation
  // is less than p
  } while(fabs(trap[j][j]-trap[j-1][j]) > p);
  return trap[j][j];
}

double f(double x) { return x*x; }

int main(void)
{
  cout << "The integral of x^2 on (0,1) is "
       << romberg(f,0.0,1.0,0.0001) << endl;
  return 0;
}
```

Problem 19. *Double numerical integration* is the application twice of a numerical integration method for single integration, once for the y direction and another for the x direction. Any numerical integration method for single integration can be applied to double integration. Write a C++ program that applies Simpson's 1/3 rule to find the double integral

$$I = \int_{a=1}^{b=3} \left(\int_{y=\ln(x)}^{y=3+\exp(x/5)} \sin(x+y) dy \right) dx .$$

Solution 19. To do the numerical integration we apply two nested `for` loops for the y and x integration.

```
// doubleintegration.cpp

#include <iostream>
#include <cmath>
using namespace std;

double function(double x,double y) { return sin(x+y); }

double lower_curve(double x) { return log(x); }

double upper_curve(double x) { return (3.0 + exp(x/5.0)); }
```

```
int main(void)
{
  int k, m, n;
  double a, b, c, d, f, hx, hy, s, t, w, x, y;
  cout << "double integration using the Simpson rule" << endl;
  cout << "number of intervals must be even" << endl;
  cout << "number of intervals in x-direction: m = "; cin >> m;
  cout << "number of intervals in y-direction: n = "; cin >> n;
  cout << "lower boundary of x: a = "; cin >> a;
  cout << "upper boundary of x: b = "; cin >> b;
  hx = (b - a)/m;
  t = 0.0;
  for(int i=0;i<=m;i++)
  {
  x = a + i*hx;
  c = lower_curve(x); // finds the lower limit of y value
  d = upper_curve(x); // finds the upper limit of x value
  hy = (d - c)/n;     // interval size in y-direction
  s = 0.0;
  for(int j=0;j<=n;j++)
  {
  y = c + j*hy;       // y value of grid points
  f = function(x,y);
  w = 4.0;
  if(j%2 == 0) w = 2.0;
  if((j==0) || (j == n)) w = 1.0;
  s += w*f;           // integration in y-direction
  } // end for j
  s *= hy/3.0;
  w = 4.0;
  if(i%2 == 0) w = 2.0;
  if((i == 0) || (i == m)) w = 1.0;
  t += w*s;          // integration in x-direction
  } // end for i
  t *= hx/3.0;
  cout << "integral = " << t << endl;
  return 0;
}
```

Problem 20. Given a set of data

$$\{ x_j , y_j = f(x_j) \}$$

where $j = 0, 1, \ldots, N-1$. Suppose that the y_j can be complex and that the node points are equally spaced in the interval $[0, 2\pi]$, i.e., $x_j = 2\pi j/N$ for $j = 0, 1, \ldots, N$.
(i) Find

$$p(x) = \sum_{k=0}^{N-1} c_k e^{ikx} = c_0 + c_1 e^{ix} + c_2 e^{2ix} + \cdots + c_{N-1} e^{(N-1)ix} \qquad (1)$$

such that

$$p(x_j) = f(x_j)\,.$$

This is called *trigonometric interpolation.*
(ii) Apply it to the data set ($N = 4$)

$$\begin{aligned}(x_0, y_0) &= (0, 0)\\ (x_1, y_1) &= \left(\frac{\pi}{2}, \frac{\sqrt{3}}{2}\pi\right)\\ (x_2, y_2) &= (\pi, \pi)\\ (x_3, y_3) &= \left(\frac{3}{2}\pi, \frac{\sqrt{3}}{2}\pi\right)\,.\end{aligned}$$

Solution 20. Since

$$e^{i(x+2\pi)} = e^{ix} e^{2i\pi} = e^{ix}$$

the interpolation is periodic, i.e., $p(x + 2\pi) = p(x)$. Let

$$w^j = e^{ix_j}$$

where

$$w := e^{2\pi i/N}\,.$$

Thus $w^N = 1$ so that w is the Nth root of unity. We obtain from (1)

$$f(x_j) = p(x_j) = c_0 + c_1 w^j + c_2 w^{2j} + \cdots + c_{N-1} w^{(N-1)j}\,.$$

This set of equations for $j = 0, 1, \ldots, N-1$ is a system of N equations in N unknowns and can be written in matrix form $A\mathbf{c} = \mathbf{y}$

$$\begin{pmatrix} 1 & 1 & 1 & \ldots & 1\\ 1 & w & w^2 & \ldots & w^{N-1}\\ 1 & w^2 & w^4 & \ldots & w^{2(N-1)}\\ \vdots & & & & \vdots\\ 1 & w^{N-1} & w^{2(N-1)} & \ldots & w^{(N-1)^2} \end{pmatrix} \begin{pmatrix} c_0\\ c_1\\ c_2\\ \vdots\\ c_{N-1} \end{pmatrix} = \begin{pmatrix} y_0\\ y_1\\ y_2\\ \vdots\\ y_{N-1} \end{pmatrix}\,.$$

This is the *discrete Fourier transform.* The matrix A on the left-hand side has an inverse with

$$(A^{-1})_{jk} = \frac{1}{N} w^{-jk} .$$

Thus $\mathbf{c} = A^{-1}\mathbf{y}$. It follows that

$$c_k = \frac{1}{N} \sum_{m=0}^{N-1} w^{-km} y_m .$$

Neglecting sums, to compute each coefficient c_k we perform N multiplications. This means that discrete Fourier transforms require N^2 operations. Hence, for the calculation we should use the fast Fourier transform.
(ii) We find

$$c_0 = \frac{\pi}{4}(1+\sqrt{3}), \quad c_1 = -\frac{\pi}{4}, \quad c_2 = \frac{\pi}{4}(1-\sqrt{3}), \quad c_3 = -\frac{\pi}{4} .$$

Thus

$$p(x) = \frac{\pi}{4}(1+\sqrt{3} - e^{ix} + (1-\sqrt{3})e^{2ix} - e^{3ix}) .$$

The real part is

$$p_r(x) = \frac{\pi}{4}(1+\sqrt{3} - \cos(x) + (1-\sqrt{3})\cos(2x) - \cos(3x)) .$$

Problem 21. Let $b > a$ and consider the integral

$$\int_a^b f(x)dx .$$

Let

$$t = \frac{1}{a-b}(a+b-2x) . \tag{1}$$

Express the integral with this variable.

Solution 21. We have $a \le x \le b$. Using the transformation we obtain $-1 \le t \le 1$. Since

$$dx = \frac{b-a}{2} dt$$

and solving (1) for x

$$x = \frac{1}{2}(a+b) + \frac{1}{2}(b-a)t$$

yields

$$\int_a^b f(x)dx = \frac{b-a}{2} \int_{-1}^{+1} f\left(\frac{1}{2}(a+b) + \frac{1}{2}(b-a)t\right) dt .$$

Problem 22. Linear interpolation and quadratic interpolation are special cases of *polynomial interpolation* where we approximate an unkown function f for which we know values at $n+1$ points by the unique polynomial of degree n that passes through these points. We construct one form of the equation for the polynomial passing through the points

$$(x_0, f(x_0)), \quad (x_1, f(x_1)), \ldots, (x_n, f(x_n)) .$$

We define the following functions

$$\begin{aligned}
L_0(x) &:= (x - x_1)(x - x_2) \cdots (x - x_n) \\
L_1(x) &:= (x - x_0)(x - x_2) \cdots (x - x_n) \\
&\vdots \\
L_i(x) &:= (x - x_0)(x - x_1) \cdots (x - x_{i-1})(x - x_{i+1}) \cdots (x - x_n) \\
&\vdots \\
L_n(x) &:= (x - x_0)(x - x_1) \cdots (x - x_{n-1}) .
\end{aligned}$$

Each of these functions is a polynomial of degree n. Note that $L_i(x_j) = 0$ for $i \neq j$ and $L_i(x_i) \neq 0$. We can use these functions to construct a polynomial of degree n that passes through our $n+1$ points as follows

$$p(x) = \frac{L_0(x)}{L_0(x_0)} f(x_0) + \frac{L_1(x)}{L_1(x_1)} f(x_1) + \cdots + \frac{L_n(x)}{L_n(x_n)} f(x_n) .$$

Obviously $p(x_i) = f(x_i)$ for $i = 1, 2, \ldots, n$. Write a C++ program that implements this interpolation. The data are $(0, 1.0)$, $(0.5, 1.2)$, $(1.0, 1.4)$, $(1.5, 1.7)$, $(2.0, 2.2)$.

Solution 22. We are using two `for`-loops and an `if`-condition to solve the problem. For the products in L_i we initialize `termi` to 1.

```
// interpolation.cpp

#include <iostream>
using namespace std;

int main(void)
{
   int n = 5; // number of points
   double* x = new double[n];
   x[0] = 0.0; x[1] = 0.5; x[2] = 1.0; x[3] = 1.5; x[4] = 2.0;
   double* fx = new double[n];
```

```
    fx[0] = 1.0; fx[1] = 1.2; fx[2] = 1.4;
    fx[3] = 1.7; fx[4] = 2.2;

    double xvalue = 1.25;
    double p = 0.0;
    double termi;

    for(int i=0;i<n;i++)
    {
    termi = 1.0;
    for(int j=0;j<n;j++)
    {
    if(j != i) termi = termi*(xvalue - x[j])/(x[i] - x[j]);
    } // end for j
    p += termi*fx[i];
    } // end for i
    cout << "p = " << p;
    delete[] x;  delete[] fx;
    return 0;
}
```

Problem 23. Let $A = (a_{ik})$ be an $n \times n$ matrix, and let $\mathbf{x}$ and $\mathbf{b}$ be n-vectors with elements in $\mathbf{C}$ such that

$$A\mathbf{x} = \mathbf{b}. \tag{1}$$

Assume that

$$a_{jj} \neq 0, \quad j = 1, 2, \ldots, n \tag{2}$$

and define

$$D := \operatorname{diag}(a_{jj}). \tag{3}$$

Then (1) can be written as

$$\mathbf{x} = B\mathbf{x} + \mathbf{c} \tag{4}$$

where

$$B = -D^{-1}(A - D)$$
$$\mathbf{c} = D^{-1}\mathbf{b}.$$

The *Jacobi method* for the solution of (1) respectively (4) is given by

$$\mathbf{x}^{(k+1)} = B\mathbf{x}^{(k)} + \mathbf{c}, \qquad k = 0, 1, 2, \ldots$$

where $\mathbf{x}^{(0)}$ is any initial vector. This method is convergent for each initial vector $\mathbf{x}^{(0)}$ iff

$$B^k \to 0 \quad \text{for} \quad k \to \infty$$

which in turn is equivalent to the condition that the *spectral radius* $\rho(B)$ satisfies

$$\rho(B) := \max_{j=1,2,\ldots,n} |\lambda_j(B)| < 1$$

where $\lambda_j(B)$, $j = 1, 2, \ldots, n$ are the eigenvalues of B. Write a C++ program for this method and apply it to the system of linear equations

$$\begin{aligned} 5x + y + z &= 10 \\ x + 6y - 2z &= 7 \\ x - 2y + 7z &= 16\,. \end{aligned}$$

Solution 23. Obviously, the system satisfies the condition given above. The C++ implementation is

```
// Jacobi.cpp

#include <iostream>
using namespace std;

int main(void)
{
  int n = 3;
  double** A = NULL;
  A = new double*[n];
  for(int i=0;i<n;i++) A[i] = new double[n];

  A[0][0] = 5.0; A[0][1] = 1.0; A[0][2] = 1.0;
  A[1][0] = 1.0; A[1][1] = 6.0; A[1][2] = -2.0;
  A[2][0] = 1.0; A[2][1] = -3.0; A[2][2] = 7.0;

  double* b = new double[n];
  b[0] = 10.0; b[1] = 7.0; b[2] = 16.0;
  double x = 0.0, y = 0.0, z = 0.0;
  double x1, y1, z1;

  int T = 100; // number of iterations
  for(int j=0;j<T;j++)
  {
  x1 = x; y1 = y; z1 = z;
  x = (b[0] - A[0][1]*y1 - A[0][2]*z1)/A[0][0];
```

```
  y = (b[1] - A[1][0]*x1 - A[1][2]*z1)/A[1][1];
  z = (b[2] - A[2][0]*x1 - A[2][1]*y1)/A[2][2];
  }
  cout << "x = " << x << endl;
  cout << "y = " << y << endl;
  cout << "z = " << z << endl;

  for(int k=0;k<n;k++) delete[] A[k];
  delete[] A;
  return 0;
}
```

Problem 24. A finite *continued fraction* is an expression of the form

$$f_n(x) = \cfrac{a_0}{b_0 + x + \cfrac{a_1}{b_1 + x + \cfrac{a_2}{\ldots + \cfrac{\cdots}{b_{n-1} + x + \cfrac{a_n}{b_n + x}}}}}$$

A convenient notation for this expression is the following

$$\frac{a_0}{b_0 + x+}\,\frac{a_1}{b_2 + x+}\cdots\frac{a_n}{b_n + x}\,.$$

The following recursion formulas are valid

$$f_n(x) = \frac{a_0}{r_0}, \quad r_n := x + b_n, \quad r_k := x + b_k + \frac{a_{k+1}}{r_{k+1}}$$

where $k = n-1, n-2, \ldots, 1, 0$. The formula is proved by induction. Continued fractions containing a variable x are often useful for representation of functions. Such expressions often have much better convergence properties than, for example, the power-series expansion. An example is

$$\arctan(x) = \frac{x}{1+}\,\frac{x^2}{3+}\,\frac{4x^2}{5+}\,\frac{9x^2}{7+}\,\frac{16x^2}{9+}\,\frac{25x^2}{11+}\cdots$$

Write a C++ program that implements this recursion and apply it to $x = 0$,

$$a_0 = a_1 = \cdots = a_n = 1, \qquad b_0 = b_1 = \cdots = b_n = 1$$

to find an approximation of the golden mean number $(\sqrt{5} - 1)/2$.

Solution 24.

```
// contfrac.cpp

#include <iostream>
using namespace std;

double fn(double* a,double* b,double x,int n)
{
  double zr;
  zr = x + b[n];
  for(int k=n-1;k>=0;k--) { zr = x + b[k] + a[k+1]/zr; }
  return a[0]/zr;
}

int main(void)
{
  int n = 4;
  double* a = new double[n+1];
  double* b = new double[n+1];
  a[0] = 1.0; a[1] = 1.0; a[2] = 1.0;
  a[3] = 1.0; a[4] = 1.0;
  b[0] = 1.0; b[1] = 1.0; b[2] = 1.0;
  b[3] = 1.0; b[4] = 1.0;
  double x = 0.0;
  double result = fn(a,b,x,n);
  cout << "result = " << result << endl;
  delete[] a; delete[] b;
  return 0;
}
```

Problem 25. The *complete elliptic integral* of the first kind is

$$K(m) = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1 - \sin^2\alpha \sin^2\theta}} = \int_0^1 \frac{dt}{\sqrt{(1-t^2)(1-mt^2)}}$$

where $m \in [0,1)$, $k = \sin\alpha$ and $k^2 = m$. The elliptic integral can be computed using the *method of the arithmetic geometric mean.* We start with the initial values

$$a_0 = 1, \quad b_0 = \sqrt{1-m}, \quad c_0 = \sqrt{m}.$$

We compute successive iterations of a_j, b_j and c_j with

$$a_j = \frac{1}{2}(a_{j-1} + b_{j-1}), \quad b_j = (a_{j-1}b_{j-1})^{1/2}, \quad c_j = \frac{1}{2}(a_{j-1} - b_{j-1})$$

stopping at iteration n, when $c_n \approx 0$ within the tolerance specified by ϵ. The complete elliptic integral of first kind is then

$$K(\alpha) = \frac{\pi}{2a_n}.$$

Write a C++ program that implements this algorithm.

Solution 25. In the program we set $m = 1/2$. Thus $k = 1/\sqrt{2}$ and from $\sin\alpha = k$ we find $\alpha = \pi/4$.

```
// elliptic.cpp

#include <iostream>
#include <cmath>
using namespace std;

int main(void)
{
  double pi = 3.14159265;
  double m = 0.5;
  double a0 = 1.0, b0 = sqrt(1.0-m), c0 = sqrt(m);
  double eps = 0.0001;
  double a1, b1, c1;

  while(fabs(c0) > eps)
  {
  a1 = (a0 + b0)/2.0;
  b1 = sqrt(a0*b0);
  c1 = (a0 - b0)/2.0;
  a0 = a1;  b0 = b1;  c0 = c1;
  }
  double Kalpha = pi/(2.0*a1);
  cout << "Kalpha = " << Kalpha;
  return 0;
}
```

Problem 26. Let S be a closed set on a complete metric space X. A *contracting mapping* is a mapping $f : S \to S$ such that

$$d(f(x), f(y)) \leq kd(x, y), \quad 0 \leq k < 1, \quad d \text{ is the distance in } X.$$

One also says that "f is *lipschitzian* of order $k < 1$".

Contracting mapping theorem. A contracting mapping f has strictly one *fixed point*; i.e., there is one and only one point x^* such that $x^* = f(x^*)$. The proof is by successive iteration. Let $x^* \in S$, then

$$f(x_0) \in S, \ldots, f^{(n)}(x_0) = f(f^{(n-1)}(x_0)) \in S$$

and

$$d(f^{(n)}(x_0), f^{(n-1)}(x_0)) \leq k^{n-1} d(f(x_0), x_0).$$

Since $k < 1$ the sequence $f^{(n)}(x_0)$ is a *Cauchy sequence* and it tends to a limit $x^* \in S$ when n tends to infinity

$$x^* = \lim_{n \to \infty} f^{(n)}(x_0) = \lim_{n \to \infty} f(f^{(n-1)}(x_0)) = f(x^*).$$

The uniqueness of x^* results from the defining property of contracting mappings. Assume that there is another point y^* such that $y^* = f(y^*)$, then

$$d(f(y^*), f(x^*)) = d(y^*, x^*)\,.$$

On the other hand, $d(f(y^*), f(x^*)) \leq k d(y^*, x^*)$, where $k < 1$. Hence, $d(y^*, x^*) = 0$ and $y^* = x^*$.

Consider the map $f : (0, 1] \to (0, 1]$

$$f(x) = \frac{5}{2} x(1 - x).$$

Obviously, the map f has a stable fixed point $x^* = 3/5$. Write a C++ program that applies the contracting mapping theorem with the initial value $x_0 = 9/10$.

Solution 26. If the relative difference is less than 10^{-5}, we claim that a fixed point has been obtained. The numerical value of the mapping converges to 0.6.

```
// fixedpoint.cpp

#include <iostream>
#include <cmath>
using namespace std;

double f(double x) { return 5.0*x*(1.0 - x)/2.0; }

int main(void)
{
  double t, x = 0.9;
```

```
  double eps = 0.00001;

  do { t = x; x = f(x); }
    while(fabs(t - x) > eps);
  cout << "x = " << x;
  return 0;
}
```

Problem 27. We consider the Hilbert space $L_2[-1/2, 1/2]$. Expand the step function

$$f(x) := \begin{cases} -1 & x \in [-1/2, 0] \\ 1 & x \in (0, 1/2] \end{cases}$$

with respect to the Fourier basis

$$\{ \phi_k(x) := \exp(2\pi i k x) \ : \ k \in \mathbf{Z} \} .$$

In other words, calculate the expansion coefficients $\langle f, \phi_k \rangle$ where

$$\langle f(x), g(x) \rangle := \int_{-1/2}^{1/2} f(x) \overline{g(x)} dx.$$

Thus, find the *Fourier expansion*

$$f(x) = \sum_{k \in \mathbf{Z}} \langle f(x), \phi_k(x) \rangle \phi_k(x) .$$

Solution 27. For $k = 0$ we have

$$\langle f(x), \phi_0(x) \rangle = - \int_{-\frac{1}{2}}^{0} dx + \int_{0}^{\frac{1}{2}} dx = 0 .$$

For $k \neq 0$ we have

$$\begin{aligned} \langle f(x), \phi_k(x) \rangle &= - \int_{-\frac{1}{2}}^{0} e^{-2\pi i k x} dx + \int_{0}^{\frac{1}{2}} e^{-2\pi i k x} dx \\ &= \frac{1}{2\pi i k x} \left(\left[e^{-2\pi i k x} \right]_{-\frac{1}{2}}^{0} - \left[e^{-2\pi i k x} \right]_{0}^{\frac{1}{2}} \right) \\ &= \frac{1}{\pi i k} (1 - (-1)^k) . \end{aligned}$$

Thus

$$f(x) = \sum_{k \in \mathbf{Z}, k \neq 0} \frac{1}{\pi i k} (1 - (-1)^k) e^{2\pi i k x}$$

$$= \sum_{k\in\mathbf{Z},k\neq 0} \frac{1}{\pi i k}(1-(-1)^k)(\cos(2\pi kx)+i\sin(2\pi kx))$$
$$= \sum_{k=1,k\ odd}^{\infty} \frac{4}{\pi k}\sin(2\pi kx)\,.$$

Problem 28. Consider the function $f \in L_2[0,1]$

$$f(x) = \begin{cases} x & \text{for } 0 \le x \le 1/2 \\ 1-x & \text{for } 1/2 < x < 1 \end{cases}$$

An orthonormal basis in the Hilbert space $L_2[0,1]$ is given by

$$B := \left\{ 1,\ \sqrt{2}\cos(\pi nx) \ :\ n = 1,2,\dots \right\}.$$

(i) Find the *Fourier expansion* of f with respect to this orthonormal basis.
(ii) From this expansion show that we can calculate

$$\frac{\pi^2}{8} = \sum_{k=0}^{\infty} \frac{1}{(2k+1)^2}\,. \tag{1}$$

Solution 28. (i) The Fourier expansion is given by

$$f(x) = \sum_{n=0}^{\infty} \langle f(x), \phi_n(x)\rangle \phi_n(x)$$

where $\phi_0(x) = 1$ and $\phi_n(x) = \sqrt{2}\cos(\pi nx)$ with $n = 1,2,\dots$. For the scalar products we have

$$\langle f(x), 1\rangle = \int_0^1 f(x)dx = \int_0^{1/2} x dx + \int_{1/2}^1 (1-x)dx = \frac{1}{4}$$

and

$$\langle f(x), \phi_n(x)\rangle = \sqrt{2}\int_0^{1/2} x\cos(\pi nx)dx + \sqrt{2}\int_{1/2}^1 (\cos(n\pi x) - x\cos(n\pi x)dx$$
$$= \frac{\sqrt{2}}{\pi^2 n^2}\left(2\cos\left(\frac{n\pi}{2}\right) - 1 - (-1)^n\right).$$

Thus, for n odd we have $\langle f(x), \phi_n(x)\rangle = 0$. For $n = 2j$ even ($j \in \mathbf{N}$) we have

$$\langle f(x), \phi_{2j}(x)\rangle = \frac{\sqrt{2}}{4\pi^2 j^2}(2\cos(j\pi) - 2)\,.$$

For j, even $\langle f(x), \phi_{2j}(x)\rangle = 0$. For $j = 2k+1$ odd ($k \in \mathbf{N}_0$) we have

$$\langle f(x), \phi_{4k+2}(x)\rangle = -\frac{\sqrt{2}}{\pi^2(2k+1)^2}\,.$$

Thus, the Fourier epansion yields

$$f(x) = \frac{1}{4} - \frac{2}{\pi^2}\sum_{k=0}^{\infty}\frac{1}{(2k+1)^2}\cos(\pi(4k+2)x)\,.$$

(ii) For $x = 0$ we have $f(0) = 0$ and $\cos(0) = 1$. Thus

$$f(0) = 0 = \frac{1}{4} - \frac{2}{\pi^2}\sum_{k=0}^{\infty}\frac{1}{(2k+1)^2}\,.$$

Consequently we obtain (1).

Problem 29. Let $\mathcal{H}$ be an arbitrary Hilbert space. Let $B := \{\,\phi_k \;:\; k \in I\,\}$ be an orthonormal basis in the Hilbert space $\mathcal{H}$, where I is the countable index set. Then for all $f, g \in \mathcal{H}$ we have

$$\langle f, g\rangle = \sum_{k\in I}\overline{\langle f, \phi_k\rangle}\langle g, \phi_k\rangle$$

where $\langle f, g\rangle$ denotes the scalar product in the Hilbert space. The equation is called *Parseval's equality* or *Parseval's relation.* Consider the Hilbert space $\mathcal{H} = L_2[-\pi, \pi]$ with the orthonormal basis

$$\left\{\,\phi_k(x) = \frac{1}{\sqrt{2\pi}}e^{ikx} \;:\; k \in \mathbf{Z}\,\right\}.$$

Let

$$f(x) = e^{icx}, \quad c \in \mathbf{R} \text{ and } c \notin \mathbf{Z}\,.$$

Use f and Parseval's equality to show that

$$\sum_{k=-\infty}^{\infty}\frac{1}{(c-k)^2} = \frac{\pi^2}{\sin^2(c\pi)}\,. \tag{1}$$

Solution 29. If $g = f$, Parseval's equality reduces to

$$\langle f, f\rangle = \|f\|^2 = \sum_{k\in I}\overline{\langle f, \phi_k\rangle}\langle f, \phi_k\rangle\,.$$

For the scalar product we have

$$\langle f, \phi_k \rangle = \int_{-\pi}^{\pi} f(x)\overline{\phi_k(x)}dx = \frac{1}{\sqrt{2\pi}} \int_{-\pi}^{\pi} e^{i(c-k)x}dx \,.$$

Integration yields

$$\langle f, \phi_k \rangle = \frac{2}{\sqrt{2\pi}} \frac{(-1)^k \sin(c\pi)}{c-k} \,.$$

Since

$$\overline{\langle f, \phi_k \rangle} = \langle f, \phi_k \rangle$$

and

$$\|f\|^2 = \langle f, f \rangle = \int_{-\pi}^{\pi} dx = 2\pi$$

we obtain

$$2\pi = \frac{2}{\pi} \sum_{k=-\infty}^{\infty} \frac{\sin^2(c\pi)}{(c-k)^2} \,.$$

Thus (1) follows.

Chapter 10

Random Numbers and Monte Carlo Techniques

Problem 1. ANSI C and ANSI C++ include an integer *random number* generator. The function `srand()` initializes the random number generator (seed). The function `rand()` uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo random numbers in the range from `0` to `RAND_MAX`. The symbolic constant `RAND_MAX` is defined in `stdlib.h` and its value is platform dependent. Write a C++ program that generates random numbers in range [0,1] (data type `double`). Add code that generates 10 random number in the range `0..99` and a random binary string of length 32.

Solution 1.

```
// random.cpp

#include <iostream>
#include <cstdlib>  // for srand(), rand()
#include <ctime>    // for time()
using namespace std;

int main(void)
{
  srand((unsigned long) time(NULL)); // seed
  double x = (double) rand();
```

```
  double m = (double) RAND_MAX;
  cout << "m = " << m << endl;
  double r1 = x/m;
  cout << "r1 = " << r1 << endl;

  double y = (double) rand();
  double r2 = y/m;
  cout << "r2 = " << r2 << endl;

  cout << "ten random numbers from 0 to 99 " << endl;
  for(int i=0;i<10;i++) { cout << rand()%100 << endl; }

  cout << "random binary string " << endl;
  int* bstr = new int[32];
  for(int j=0;j<32;j++) { bstr[j] = rand()%2; }
  for(int k=0;k<32;k++) { cout << bstr[k]; }
  delete[] bstr;
  return 0;
}
```

Problem 2. Write a C++ program that provides a seed (`unsigned long`) for the function `srand()` and finds out after how many steps the first random number repeats.

Solution 2.

```
// randrepeat.cpp

#include <iostream>
#include <cstdlib>
using namespace std;

int main(void)
{
  unsigned long seed;
  cout << "Enter seed for rand() : ";
  cin >> seed;
  srand(seed);
  unsigned long first = rand();
  unsigned long i = 0;
  while(rand() != first) ++i;
  cout << "Random number " << first << " repeats after "
       << i << " steps" << endl;
```

```
  return 0;
}
```

Problem 3. The generation of uniform sampling points on the sphere $x^2 + y^2 + z^2 = R^2$ can be done as follows. Choose a random number s uniformly distributed in the interval $[-1, 1]$ and a random number ϕ uniformly distributed in $[0, 2\pi]$. The x, y, and z coordinates of the point on the sphere are given by

$$x = R\sqrt{1 - s^2}\cos\phi$$
$$y = R\sqrt{1 - s^2}\sin\phi$$
$$z = Rs\,.$$

The principle underlying this algorithm is that for a sphere of radius R, the area of a zone of width h is $2[0, 2\pi]Rh$, regardless of where the sphere is sliced. Therefore, the z-coordinates of random points on the sphere are distributed uniformly. Points obtained by using this algorithm are then uniformly distributed on the surface of the sphere. Write a Java program that generates these points.

Solution 3. The method `double random()` in the `Math` class of Java provides a "random number" of data type `double` in the interval $[0, 1]$ uniformly distributed. To obtain random numbers for the interval $[-1, 1]$ we have to scale by 2 and translate by -1 (i.e., $2r-1$) and to obtain random numbers in the interval $[0, 2\pi]$ we have to scale by $2\pi r$, where $r \in [0, 1]$.

```
// SphereRandom.java

class SphereRandom
{
  public static void main(String[] args)
  {
  double R = 1.0;
  double r1 = Math.random();
  double s = 2.0*r1-1.0;
  double r2 = Math.random();
  double phi = 2.0*Math.PI*r2;
  double x = R*Math.sqrt(1.0-s*s)*Math.cos(phi);
  double y = R*Math.sqrt(1.0-s*s)*Math.sin(phi);
  double z = R*s;

  System.out.println("x = " + x);
  System.out.println("y = " + y);
```

```
  System.out.println("z = " + z);
  double square = x*x + y*y + z*z;
  System.out.println("square = " + square);
  }
}
```

Problem 4. A very simple technique to generate pseudo random numbers, is to use a generalized feedback shift register (*shift register algorithm*). A generalized feedback shift register produces random bits from a register of p bits. Everytime a new random bit is needed, the register is shifted to produce output and certain bits are XORed with other bits. For example

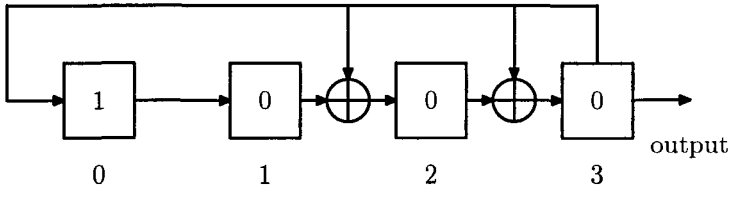
$$a_n = a_{n-103} \text{ XOR } a_{n-250}$$

for a random number generator. Here XOR denotes the bitwise exclusive OR operation. We can simplify the algorithm if we only allow the output to be XORed with other bits. Thus, we have a set of p bits a_k, $k = 0, 1, \ldots, p$ generated in advance somehow, where a_p is the last bit. Now we perform the shift operation

$$a_k := a_{k-1}, \qquad a_0 := 0$$

for $k = 1, 2, \ldots, p$. The output bit is $a_{out} := a_{p+1}$. After the shift operation has been performed, we XOR certain positions with the output. Let these q positions be $x_1, x_2, \ldots, x_q$. Then

$$a_{x_q} := a_{x_q} \oplus a_{out} .$$



For example, let $p = 3$, $x_1 = 2$ and $x_2 = 3$. If we run the feedback shift register algorithm, we get:

Round	Cell 0	Cell 1	Cell 2	Cell 3
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1
5	1	0	1	1
6	1	1	1	0
7	0	1	1	1

Write a C++ program for a feedback shift register with $p = 19$ and $x_1 = 0$, $x_2 = 2$, $x_3 = 4$, $x_4 = 6$, $x_5 = 7$, $x_6 = 14$, $x_7 = 17$ and $x_8 = 19$. Use the feedback shift register to generate 100 16-bit pseudo random numbers.

Solution 4. The function `fsr_bit()` obtains the next bit from the feedback shift register. The function `rnd()` uses the bits from the feedback register to construct a random integer.

```
// gfsr.cpp

#include <iostream>
#include <bitset>
using namespace std;

bitset<20> a;
int x[] = { 0,2,4,6,7,14,17,19 };
const int NXOR = 8;

int fsr_bit(void)
{
  int i;
  int output = a[19];
  // shift
  for(i=19;i>0;i--) a[i] = a[i-1];
  a[0] = 0;
  // feedback
  if(output == 1) {
    for(i=0;i<NXOR;i++) { a.flip(x[i]); }
  }
  return output;
}

int rnd(void)
{
  int r = 0;
  for(int i=0;i<16;i++) { r *= 2; r += fsr_bit(); }
  return r;
}

int main(void)
{
  a.reset(); a.set(19);
  cout << "Generating 100 pseudo-random numbers:" << endl;
  for(int i=0;i<100;i++)
```

```
  { cout << rnd() << endl; }
  return 0;
}
```

Problem 5. Random numbers with a homogeneous distribution are fairly easy to create, but for many applications random numbers with a *Gaussian distribution* are necessary. Convert a random number sequence with homogeneous distribution into a sequence with a Gaussian distribution.

Solution 5. Given a real valued random number sequence $\{ r_j \}$ with a normalized, uniform probability distribution in the interval $[0, 1]$

$$p(r)dr = \begin{cases} dr & \text{for} \quad 0 \leq r \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

We create a new random number sequence $\{ s_j \}$ by mapping a strictly monotonous function $f(r)$ onto the sequence $\{ r_j \}$. The probability distribution of the new sequence $\{ s_j \} = \{ f(r_j) \}$ is then given by

$$p(s)ds = p(r) \left| \frac{\partial r}{\partial s} \right| ds \,. \tag{1}$$

To find the function $f(r)$ for a desired probability distribution $p(s)$ one integrates both sides of (1) over the interval $s \in [f(0), f(1)]$ or $s \in [f(1), f(0)]$ depending on $f(r)$ being a montonously increasing (i.e., $\partial f(r)/\partial r > 0$) or decreasing (i.e., $\partial f(r)/\partial r < 0$) function. Assume that $\partial f(r)/\partial r > 0$. We obtain

$$\begin{aligned} \int_{f(0)}^{s} d\tilde{s} p(\tilde{s}) &= \int_{f(0)}^{s} d\tilde{s}\, p(r) \frac{\partial r}{\partial \tilde{s}} \\ &= \int_{f(0)}^{s} d\tilde{s}\, \frac{\partial f^{(-1)}(\tilde{s})}{\partial \tilde{s}} \\ &= f^{(-1)}(\tilde{s}) \Big|_{f(0)}^{s} \,. \end{aligned}$$

Consequently

$$f^{(-1)}(s) = \int_{f(0)}^{s} d\tilde{s}\, p(\tilde{s}) \,. \tag{2}$$

The inverse of (2) renders $f(r)$. The method described fails if one cannot find a closed or at least numerically feasible form for $f(r)$. This is the case for the Gaussian distribution. However we can resort to a similar, two-dimensional approach. In the two-dimensional case we have

$$p(s_1, s_2) ds_1 ds_2 = p(r_1, r_2) \left| \frac{\partial(r_1, r_2)}{\partial(s_1, s_2)} \right| ds_1 ds_2$$

where $|\partial(\cdots)/\partial(\cdots)|$ denotes the Jacobian determinant. We obtain Gaussian-distributed random numbers as follows. One first generates two random numbers r_1 and r_2 uniformly distributed in the interval $[0,1]$. The functions

$$s_1 = \sqrt{-2\ln r_1}\sin(2\pi r_2)$$
$$s_2 = \sqrt{-2\ln r_1}\cos(2\pi r_2)$$

render then two Gaussian-distributed numbers s_1 and s_2. To verify this, we note that the inverse is given by

$$r_1 = \exp\left(-\frac{1}{2}(s_1^2+s_2^2)\right)$$
$$r_2 = \frac{1}{2\pi}\arctan\left(\frac{s_1}{s_2}\right).$$

We find

$$\begin{aligned} p(s_1,s_2)ds_1ds_2 &= p(r_1,r_2)\begin{vmatrix}\partial r_1/\partial s_1 & \partial r_1/\partial s_2\\ \partial r_2/\partial s_1 & \partial r_2/\partial s_2\end{vmatrix} \\ &= \frac{1}{2\pi}\left(\frac{s_1^2}{s_1^2+s_2^2}+\frac{s_2^2}{s_1^2+s_2^2}\right)\exp\left(-\frac{1}{2}(s_1^2+s_2^2)\right)ds_1ds_2 \\ &= \left(\frac{1}{\sqrt{2\pi}}e^{-s_1^2/2}ds_1\right)\left(\frac{1}{\sqrt{2\pi}}e^{-s_2^2/2}ds_2\right). \end{aligned}$$

This shows that s_1 and s_2 are independent Gaussian distributed numbers. Thus one can employ the equations to produce Gaussian random numbers two at a time.

Problem 6. To evaluate an integral

$$\int_\Omega d\mathbf{x} f(\mathbf{x})$$

with the *Monte Carlo method* one samples the function f at M homogeneously distributed random points $\mathbf{r}_k$ in the integration domain Ω. The average of the function values of these random points multiplied by the volume $|\Omega|$ of the integration domain Ω can be taken as an estimate for the integral

$$\int_\Omega d\mathbf{x} f(\mathbf{x}) = \frac{|\Omega|}{M}\sum_{k=0}^{M-1} f(\mathbf{r}_k) + O\left(\frac{1}{\sqrt{M}}\right).$$

The more function values $f(\mathbf{r}_k)$ are taken into account the more accurate the Monte Carlo method becomes. The average $\langle f\rangle$ exhibits a statistical error proportional to $1/\sqrt{M}$. Thus, the error of the numerical integration

result is the order $O(1/\sqrt{M})$. Compare the Monte Carlo method to the *trapezoidal rule for integration.*

Solution 6. The trapezoidal rule approximates a one-dimensional function f linearly. An approximation over intervals of length h between sampling points is thus correct up to order of $f''(x)h^2$. In one dimension, the length of the integration step h is given by the number of sampling points M and the length of the integration domain Ω according to $h = |\Omega|/(M+1)$. Hence, the trapezoidal rule is an approximation up to the order of $O(1/M^2)$. Obviously, systematic numerical integration techniques are superior to the Monte Carlo method. However, the rating is different for integrals on higher dimensional domains. Consider an integration domain in n-dimensions. A systematic sampling would be done over an n-dimensional grid. A total of M sampling points would result in $\sqrt[n]{M}$ sampling points in each grid dimension. The trapezoidal rule would, thus, be correct up to the order of $O(M^{-2/n})$. A random sampling, however, is not effected by dimensional properties of the sampling domain. The Monte Carlo precision remains in the order of $O(M^{-1/2})$. Hence, we see that the Monte Carlo method becomes feasible for $(n = 4)$ dimensions and that it is superior for high-dimensional integration domains Ω.

Problem 7. Let $f : [0,1] \to [0,1]$ be a continuous function. We consider

$$I = \int_0^1 f(x)dx\,.$$

We choose N number pairs (x_j, y_j) with uniform distribution and define z_j by

$$z_j := \begin{cases} 0 \text{ if } y_j > f(x_j) \\ 1 \text{ if } y_j \leq f(x_j) \end{cases}.$$

Putting $n := \sum_{j=1}^{N} z_j$, we have $n/N \approx I$. More precisely, we have

$$I = \frac{n}{N} + O(N^{-1/2})\,.$$

The accuracy is not very good in one dimension. The traditional formulas, such as Simpson's formula, are much better. In higher dimensions, the Monte Carlo is favourable, at least if the number of dimensions is $n \geq 6$. It is more natural to consider the integral as the mean value of (ξ) where ξ is uniform, and then estimate the mean value from

$$I \approx \frac{1}{N} \sum_{j=1}^{N} f(\xi_j)\,.$$

Write a C++ implementation for Monte Carlo integration. Evaluate

$$\int_0^1 \sin(x)dx = 0.459697694132\,.$$

Solution 7. We are using the "random number generator" for the interval $[0,1]$

$$f(x) = (\pi + x)^5 \mod 1\,.$$

```
// MonteCarlo.cpp

#include <iostream>
#include <cmath>
using namespace std;

void randgen(double* x)
{
  double PI = 3.1415926535;
  *x = fmod((*x+PI)*(*x+PI)*(*x+PI)*(*x+PI)*(*x+PI),1.0);
}

double sum(double (*f)(double),int n)
{
  double u = 0.618;
  double result = 0.0;
  for(int i=0;i<n;i++) { randgen(&u); result += f(u); }
  return result/n;
}

double f(double x) { return sin(x); }

int main(void)
{
  double value;
  int n = 10000;
  value = sum(f,n);
  cout << "value = " << value;
  return 0;
}
```

Problem 8. A popular approach for generating a pseudorandom bit sequence is given by the *Blum, Blum, Shub generator* or *BBS generator.*

Let p and q be two large prime numbers such that

$$p \bmod 4 = q \bmod 4 = 3\,.$$

Setting $n = pq$ and selecting s relatively prime to n, the pseudo random bit sequence $B_1B_2B_3\ldots$ is obtained from

1. Set $X_0 := s^2 \bmod n$.
2. For $i = 1, 2, \ldots$
   $X_i := X_{i-1}^2 \bmod n$
   $B_i := X_i \bmod 2$

The BBS generator is cryptographically secure in the sense that it passes the next-bit test, i.e., there is no polynomial time algorithm that (given the previous bits of the sequence) can predict the next bit of the sequence with probability greater than $\frac{1}{2}$. Give a C++ implementation of the BBS generator using $p = 383$, $q = 503$ and $s = 101355$.

Solution 8. We use the `Verylong` class of SymbolicC++ to support integers of larger magnitude than supported by the C++ implementation. The function `bbsg()` is a template function which returns a `bitset` with bits generated using the BBS generator. Since the display of a bitset starts with the most significant bit, we fill the `bitset` in reverse order.

```
// bbs.cpp

#include <iostream>
#include <bitset>
#include "verylong.h"
using namespace std;

template <const size_t l>
bitset<l> bbsg(void)
{
  Verylong n = 192649;
  Verylong s = 101355;
  Verylong X = s*s%n;
  Verylong two = 2;
  bitset<l> b;

  for(int i=l-1;i>0;i--)
  { X = X*X%n; b.set(i,int(X%two)); }
  return b;
}
```

```
int main(void)
{
  cout << bbsg<20>() << endl;
  return 0;
}
```

Problem 9. Write a C++ program which implements the multidimensional Monte Carlo quadrature to integrate

$$\int_{a_0}^{b_0}\int_{a_1}^{b_1}\cdots\int_{a_{n-1}}^{b_{n-1}} v(x_0,x_1,\ldots,x_{n-1})dx_{n-1}dx_{n-2}\ldots dx_0\,.$$

Let `n` be the dimension of the domain on which the function `v` is to be integrated. Let `a` and `b` are the lower and upper bounds, respectively for the integrals over each of the dimensions. Let `esq` be the square of the error tolerance and `m+n` is the number of samples used for the Monte Carlo integration. Denote by `Vab` the volume being integrated.

Solution 9. In the following C++ implementation we use `rn()` to generate random numbers on the interval $[0,1]$. The function `quadmc()` implements the multidimensional Monte Carlo quadrature. As an alternative for `rn()` we can use the sequence $x_{n+1} = (x_n + \pi)^5 \bmod 1$.

```
double rn(void)
{
  static double x = 0.1;
  static const double pi = 3.14159265358979323846;
  x = (x + pi);
  x = x*x*x*x*x;
  x = x - floor(x);
  return x;
}
```

The implementation using the C++ random number generator `rand()` is given below.

```
// quadmc.cpp

#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>   // for fabs()
#include <cstdlib> // for srand()
#include <ctime>   // for time()
```

```
using namespace std;

double rn(void)
{
  static int init = 1;
  if(init)
  {
  srand(time(NULL));
  init = 0;
  }
  return double(rand())/RAND_MAX;
}

double quadmc(int n,vector<double> &a,vector<double> &b,
       double (*v)(const vector<double>&),double esq,int m)
{
  if(m < n) m = n;
  int l, itemp, ir, k, i;
  double vbar, ssq, gm, vi, vip, vsqbar, Vab = 1.0;
  vector<int> it(m);
  vector<double> x(n);

  for(i=0;i<n;i++) Vab *= fabs(b[i]-a[i]);
  for(i=0;i<m;i++) it[i] = i;
  l = 0;
  vsqbar = vbar = gm = 0;
  for(i=0;i<m;i++)
  {
  for(k=0;k<n;k++)
   x[k] = a[k] + rn()*(b[k]-a[k]);
  vi = v(x); vbar += vi; vsqbar += vi*vi;
  if(i < n)
  {
  if(x[i] < (b[i] + a[i])/2)
  x[i] += fabs(b[i]-a[i])/2;
  else x[i] -= fabs(b[i]-a[i])/2;
  vip = v(x); vbar += vip; vsqbar += vip*vip;
  if(gm < fabs(vip - vi)) { l = i; gm = fabs(vip-vi); }
  }
  }
  vbar /= (m + n);
  vsqbar /= (m + n);
  ssq = Vab*Vab*(vsqbar-vbar*vbar)/(m+n-1);
```

```
  if(ssq <= 2*esq) return vbar*Vab;

  double temp, cl, al, bl;
  m = int(m/sqrt(2.0));
  if(m < ssq/esq) m = int(ssq/esq);
  esq = esq*ssq/(ssq-esq);
  al = a[l]; bl = b[l];
  b[l] = cl = (bl + al)/2;
  temp = quadmc(n,a,b,v,esq/2,m);
  b[l] = bl; a[l] = cl;
  temp = quadmc(n,a,b,v,esq/2,m) + temp;
  a[l] = al;
  return (temp*ssq + esq*vbar*Vab)/(ssq+esq);
}

double sin(const vector<double> &x)
{
  if(x[0] > 0.0 && x[0] <= 1.0) return sin(x[0]);
  return 0.0;
}

double unit_circle(const vector<double> &x)
{
  if(x[0]*x[0]+x[1]*x[1] < 1.0) return 1.0;
  return 0.0;
}

double unit_cube(const vector<double> &x)
{
  for(int i=0;i<3;i++)
   if(x[i] < 0.0 || x[i] > 1.0) return 0.0;
  return 1.0;
}

double unit_sphere(const vector<double> &x)
{
  if(x[0]*x[0]+x[1]*x[1]+x[2]*x[2] < 1.0) return 1.0;
  return 0.0;
}

double f(const vector<double> &x)
{
  if(x[0]*x[0] <= x[1] && x[1] <= sqrt(x[0])) return 1.0;
  if(x[0]*x[0] >= x[1] && x[1] >= sqrt(x[0])) return 1.0;
```

```
  return 0.0;
}

int main(void)
{
  vector<double> a(3);
  vector<double> b(3);
  a[0] = 0.0; b[0] = 1.0;
  cout << "Int(sin(x),0,1) = "
       << setprecision(4)    // accuracy sqrt(1e-6) = 1e-3
       << quadmc(1,a,b,sin,1e-6,10000) << endl;
  a[0] = a[1] = -1.0; b[0] = b[1] =  1.0;
  cout << "Area of the unit circle = "
       << setprecision(4)    // accuracy sqrt(1e-6) = 1e-3
       << quadmc(2,a,b,unit_circle,1e-6,10000) << endl;
  a[0] = a[1] = a[2] = 0.0; b[0] = b[1] = b[2] = 1.0;
  cout << "Volume of the unit cube = "
       << setprecision(4)    // accuracy sqrt(1e-6) = 1e-3
       << quadmc(3,a,b,unit_cube,1e-6,10000) << endl;
  a[0] = a[1] = a[2] = -1.0; b[0] = b[1] = b[2] =  1.0;
  cout << "Volume of the unit sphere = "
       << setprecision(4)    // accuracy sqrt(1e-6) = 1e-3
       << quadmc(3,a,b,unit_sphere,1e-6,10000) << endl;
  a[0] = a[1] = 0.0; b[0] = b[1] = 1.0;
  cout << "Area between sqrt(x) and x^2 = "
       << setprecision(4)    // accuracy sqrt(1e-6) = 1e-3
       << quadmc(2,a,b,f,1e-6,10000) << endl;
  return 0;
}
```

Problem 10. *Annealing* is the process of cooling a molten substance with the objective of condensing matter into a crystaline solid. Annealing can be regarded as an optimization process. The configuration of the system during annealing is defined by the set of atomic positions r_i. A configuration of the system is weighted by its Boltzmann probability factor,

$$e^{-E(r_i)/kT}$$

where $E(r_i)$ is the energy of the configuration, k is the Boltzmann constant, and T is the temperature. When a substance is subjected to annealing, it is maintained at each temperature for a time long enough to reach thermal equilibrium. The iterative improvement technique for combinatorial optimization has been compared to rapid quenching of molten metals. Rapid cooling results in metastable system states; in metallurgy,

a glassy substance rather than a crystalline solid is obtained as a result of rapid cooling. The analogy between iterative improvement and rapid cooling of metals stems from the fact that iterative improvement and rapid cooling of metals accepts only those system configurations which decrease the cost function. In an annealing process, a new system configuration that does not improve the cost function is accepted based on the Boltzmann probability factor of the configuration. This criterion for accepting a new system state is called the *Metropolis criterion.* The process of allowing a fluid to attain thermal equilibrium at a temperature is also known as the Metropolis process. Simulated annealing essentially consists of repeating the Metropolis procedure for different temperatures. The temperature is gradually decreased at each iteration of the simulated annealing algorithm. If the initial temperature is too low, the process gets quenched very soon and only a local optimum is found. If the initial temperature is too high, the process is very slow. Only a single solution is used for the search and this increases the chance of the solution becoming stuck at a local optimum. The changing of the temperature is based on an external procedure which is unrelated to the current quality of the solution, that is, the rate of change of temperature is independent of the solution quality. The annealing mechanism can also be coupled with the quality of the current solution by making the rate of change of temperature sensitive to the solution quality. Write a C++ program which applies simulated annealing to find the minimum of the function

$$f(x) = x^2 \exp(-x/15) \sin x.$$

Solution 10. We consider the x range $[0, 100]$.

```
// annealing.cpp

#include <iostream>
#include <cmath>
#include <cstdlib>
#include <ctime>
using namespace std;

double f(double &x) { return sin(x)*x*x*exp(-x/15.0); }

int accept(double &Ecurrent,double &Enew,double &T,double &s)
{
  double dE = Enew - Ecurrent;
  double k = 1380662e-23;
  if(dE < 0.0) return 1;
  if(s < exp(-dE/(k*T))) return 1;
```

```
  else return 0;
}

int main(void)
{
  cout << "Finding the minimum via simulated annealing:"
       << endl;
  double xlow = 0.0, xhigh = 100.0;
  double T, Tmax = 500.0, Tmin = 1.0, Tstep = 0.1;

  srand(time(NULL));
  double s = rand()/double(RAND_MAX);
  double xcurrent = s*(xhigh - xlow);
  double Ecurrent = f(xcurrent);

  for(T=Tmax;T>Tmin;T-=Tstep)
  {
  s = rand()/double(RAND_MAX);
  double xnew = s*(xhigh - xlow);
  double Enew  = f(xnew);
  if(accept(Ecurrent,Enew,T,s))
  {
  xcurrent = xnew; Ecurrent = Enew;
  }
  }
  cout << "The minimum found is " << Ecurrent << " at x = "
       << xcurrent << endl;
  return 0;
}
```

Two typical outputs are given below.

```
Finding the minimum via simulated annealing:
The minimum found is -121.796 at x = 29.8397

Finding the minimum via simulated annealing:
The minimum found is -121.749 at x = 29.874
```

The global minimum of f is found as one of the solutions to the transcendental equation

$$\tan(x^*) = \frac{15x^*}{x^* - 30}$$

in the interval $[0, 100]$ with $x^* \approx \frac{19\pi}{2}$.

Problem 11. The *Ising model* is a simple model for magnetism. The classical approximation to an atomic or electronic magnetic moment is provided by an Ising spin which can take two values

$$s_j = \begin{cases} +1 & \text{spin up} \\ -1 & \text{spin down} \end{cases} .$$

A two-dimensional magnet can be modeled by a set of N spins located on a fixed two-dimensional lattice of sites. We consider a square lattice. Assume that we have a square lattice with N_x spins in the x-direction and N_y spins in the y direction such that $N = N_x N_y$. We assume that $N_x = N_y$. The interaction energy (Hamiltonian) is given by

$$E = -J \sum_{\langle ij \rangle} s_i s_j - H \sum_i s_i .$$

The first sum is over nearest neighbor pairs of spins. There are four nearest neighbors of a given lattice site. We assume periodic boundary conditions for the x and y direction. If the dimensionsless coupling constant is $J > 0$ the system is ferromagnetic, i.e., the energy is minimized if the spins point in the same direction $s_i s_j = +1$. If $J < 0$ the system is antiferromagnetic. H represents an external magnetic field which couples to the magnetization $M := \sum_i s_i$. The spins prefer to line up with the magnetic field. The average magnetization and average energy at some fixed temperature kT is given by

$$\langle M \rangle = \frac{\sum_{configs} M e^{-E/kT}}{\sum_{configs} e^{-E/kT}}, \qquad \langle E \rangle = \frac{\sum_{configs} E e^{-E/kT}}{\sum_{configs} e^{-E/kT}} .$$

To simulate the Ising model at constant temperature in the canonical ensemble we use the *Metropolis algorithm*. The algorithm is as follows.

1. Create an initial state: a random initial configuration of of spins. Calculate the energy E.
2. Make a random trial change in the state: choose a spin at random and flip it, $s_i \rightarrow -s_i$.
3. Compute $\Delta E = E_{trial} - E_{old}$, the change in the energy of the system owing to the trial change.
4. If $\Delta E \leq 0$, accept the new state and go to step 8.
5. If $\Delta E > 0$, compute the Boltzmann weighting factor $w = \exp(-\Delta E / k_B T)$.
6. Generate a random number r in the unit interval $[0, 1]$.
7. If $r \leq w$, accept the new state, otherwise retain the previous state.
8. Determine the values of the physical quantities, i.e., $\langle M \rangle$ and $\langle E \rangle$.
9. Repeat steps (2) through (8) to obtain a sufficient number of states.
10. Periodically compute averages over states.

(i) What are the number of configurations for the Ising model with N lattice sites. What is the total number of nearest neighbor pairs for an $N_x \times N_x$ lattice with periodic boundary conditions?
(ii) Write a C++ program that implements the Metropolis algorithm.

Solution 11. (i) The total number of configurations is 2^N. If for example $N_x = N_y = 20$. Then $N = 400$ and we find

$$2^N = 2^{400} = 2.58 \cdot 10^{120}.$$

The total number of nearest neighbor pairs is $2N_xN_x$.
(ii) The Boltzmann factors are pre-computed for the given fixed T and H.

```
// Metropolis.cpp

#include <iostream>
#include <cmath>
#include <cstdlib>
#include <ctime>
using namespace std;

double J = 1.0;  // couping constant ferromagentic
int Nx = 60;     // size in x direction
int Ny = 60;     // size in y direction
int N = Nx*Ny;   // number of lattice sites
double T = 0.1;  // temperature
double H = 0.0;  // magnetic field
int** s;         // spins for two-dimensional lattice
double w[17][3]; // Boltzmann factors
int steps = 0;

double rand01()  // random number in [0,1]
{
  double x = (double) rand();
  double y = (double) RAND_MAX;
  return x/y;
}

void factors()
{
  for(int i=-8;i<=8;i+=4)
  {
  w[i+8][0] = exp(-(i*J+2.0*H)/T);
  w[i+8][2] = exp(-(i*J-2.0*H)/T);
```

```
  }
}

void initialize() // allocating memory
{
  s = new int* [Nx];
  for(int k=0;k<Nx;k++) s[k] = new int[Ny];
  for(int i=0;i<Nx;i++)
    for(int j=0;j<Ny;j++)  // initial random configuration
      s[i][j] = -1 + 2*(rand()%2);
  factors();
}

bool metropolisStep()
{
  int i = rand()%Nx; int j = rand()%Ny; // random spin
  int iPrev = i == 0 ? Nx-1 : i-1;
  int iNext = i == Nx-1 ? 0 : i+1;
  int jPrev = j == 0 ? Ny-1 : j-1;
  int jNext = j == Ny-1 ? 0 : j+1;

  int sumNeighbors =
     s[iPrev][j] + s[iNext][j] + s[i][jPrev] + s[i][jNext];
  int delta_ss = 2*s[i][j]*sumNeighbors;

  double ratio = w[delta_ss+8][1+s[i][j]];
  if(rand01() < ratio)
  {
  s[i][j] = -s[i][j];
  return true;
  }
  else return false;
}

void oneMonteCarloStepPerSpin()
{
  int accepts = 0;
  for(int i=0;i<N;i++)
  if(metropolisStep()) accepts++;
  ++steps;
}

double magnetizationPerSpin()
{
```

```
  int sSum = 0;
  for(int i=0;i<Nx;i++)
     for(int j=0;j<Ny;j++)
       sSum += s[i][j];
  return sSum/((double) N);
}

double energyPerSpin()
{
  int sSum = 0, ssSum = 0;
  for(int i=0;i<Nx;i++)
    for(int j=0;j<Ny;j++)
    {
    sSum += s[i][j];
    int iNext = i == Nx-1 ? 0 : i+1;
    int jNext = j == Ny-1 ? 0 : j+1;
    ssSum += s[i][j]*(s[iNext][j] + s[i][jNext]);
    }
  return -(J*ssSum + H*sSum)/((double) N);
}

int main(void)
{
  srand((unsigned) time(NULL));
  int noSteps;
  cout << "Enter number of steps: ";
  cin >> noSteps;
  initialize();
  int thermSteps = (int) (0.2*noSteps);
  for(int s=0;s<thermSteps;s++) oneMonteCarloStepPerSpin();

  double m = 0.0, e = 0.0;
  for(int k=0;k<noSteps;k++) { oneMonteCarloStepPerSpin(); }
  m = magnetizationPerSpin();  cout << "m = " << m << endl;
  e = energyPerSpin();  cout << "e = " << e << endl;
  return 0;
}
```

Chapter 11

Ordinary Differential Equations

Problem 1. Consider the Cauchy problem for the *logistic differential equation*

$$\frac{du}{dt} = S(u), \qquad S(u) := u(1-u)$$

where $u(0) = u_0$ with $u_0 > 0$. Its solution is monotone, has a stable fixed point (steady-state solution) $u^* = 1$ for every positive initial data u_0. *Fixed points* of a differential equation $du/dt = S(u)$ are the solutions of the equation $S(u^*) = 0$.
(i) Apply the *explicit Euler method*

$$u_{n+1} = u_n + \tau S(u_n)$$

to the differential equation, where $n = 0, 1, \ldots$.
(ii) Apply the *Leap-frog method*

$$u_{n+1} = u_{n-1} + 2\tau S(u_n)$$

to the differential equation.
(iii) Apply the *Adams-Bashforth method*

$$u_{n+1} = u_n + \frac{\tau}{2}\left(3S(u_n) - S(u_{n-1})\right)$$

to the differential equation.

(iv) Apply the *modified explicit Euler method*

$$u_{n+1} = u_n + \tau S\left(u_n + \frac{\tau}{2} S(u_n)\right)$$

to the differential equation.
(v) Apply the improved Euler method

$$u_{n+1} = u_n + \frac{\tau}{2}\left(S(u_n) + S(u_n + \tau S(u_n))\right)$$

to the differential equation.
(vi) Apply the fourth order accurate *Runge-Kutta method*

$$u_{n+1} = u_n + \frac{\tau}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

$$\begin{aligned} k_1 &= S(u_n) \\ k_2 &= S\left(u_n + \frac{\tau}{2} k_1\right) \\ k_3 &= S\left(u_n + \frac{\tau}{2} k_2\right) \\ k_4 &= S(u_n + \tau k_3) \end{aligned}$$

to the differential equation.

Solution 1. (i) The explicit Euler method gives a stable asymptote $u^* = 1$ only if $0 < u_0 < 1/\tau$ and for $\tau < 2$.
(ii) The Leap-frog method gives chaotic solutions for all $\tau > 0$, although it is symplectic area-preserving.
(iii) The Adams-Bashforth method gives the stable steady-state solution $u^* = 1$ only when $0 < \tau < 1$.
(iv) The modified explicit Euler method gives the correct steady state solution only for parameter τ from a limited region.
(v) The improved Euler method gives the correct steady state solution only for parameter τ from a limited region.
(vi) The fourth order accuarte Runge-Kutta method gives the correct steady state solution only for parameter τ from a limited region.

Problem 2. Consider a one particle system with canonical space variable (q, p) and Hamilton function

$$H(p, q) = \frac{p^2}{2m} + V(q)\,.$$

The *Hamilton's equations of motion* are given by

$$\frac{dq}{dt} = \frac{\partial H}{\partial p}, \qquad \frac{dp}{dt} = -\frac{\partial H}{\partial q}\,.$$

A first attempt at time discretization of Hamilton's equations of motion with step size τ might be the simplest finite differencing scheme

$$q_{n+1} = q_n + \tau \frac{\partial H(q_n, p_n)}{\partial p_n}$$
$$p_{n+1} = p_n - \tau \frac{\partial H(q_n, p_n)}{\partial q_n} .$$

What is the problem with this discretization?

Solution 2. This discretization does not preserve the *Poisson bracket*

$$\{f, g\} := \frac{\partial f}{\partial p} \cdot \frac{\partial g}{\partial q} - \frac{\partial f}{\partial q} \cdot \frac{\partial g}{\partial p}$$

under evolution. The order τ^2 term does not vanish unless $V(q) = 0$. To rectify this it is necessary to introduce higher order terms in τ in the defining equations of the discretization.

Problem 3. Consider the autonomous second-order ordinary differential equation

$$\frac{d^2x}{dt^2} + g(x)\frac{dx}{dt} + f(x) = 0$$

where g and f are analytic functions. The equation can be written as a system of first-order differential equations

$$\frac{dx_1}{dt} = x_2 = V_1(x_1, x_2)$$
$$\frac{dx_2}{dt} = -g(x_1)x_2 - f(x_1) = V_2(x_1, x_2) .$$

In *Lie series technique* to solve this system we consider the vector field

$$V = V_1(x_1, x_2)\frac{\partial}{\partial x_1} + V_2(x_1, x_2)\frac{\partial}{\partial x_2}$$
$$= x_2\frac{\partial}{\partial x_1} + (-g(x_1)x_2 - f(x_1))\frac{\partial}{\partial x_2}$$

given by this system of differential equations. Then for sufficiently small t the solution of the initial value problem is given by the Lie series

$$x_1(t) = e^{tV} x_1|_{x_1 = x_1(0), x_2 = x_2(0)}, \qquad x_2(t) = e^{tV} x_2|_{x_1 = x_1(0), x_2 = x_2(0)} .$$

In the standard approximation e^{tV} is expanded up to a certain order in t. Here we consider the approximation

$$e^{tV} \equiv e^{t(V_1 + V_2)} \approx e^{tV_1} e^{tV_2}$$

where $V_1 := V_1(x_1, x_2)\partial/\partial x_1$ and $V_2 := V_2(x_1, x_2)\partial/\partial x_2$. This is an approximation since

$$[tV_1, tV_2] \neq 0\,.$$

However for sufficiently small t it provides a good enough approximation.
(i) Find $x_1(t)$ and $x_2(t)$ with this approximation.
(ii) Consider the special case $g(x) = \epsilon(x^2 - 1)$, $f(x) = x$. This is the *van der Pol equation* with $\epsilon = 10$.
(iii) Write a Java program that implements this approximation.

Solution 3. (i) We have

$$e^{tV_2}x_1 = e^{t(-g(x_1)x_2 - f(x_2))\partial/\partial x_2}x_1 = x_1$$
$$e^{tV_1}e^{tV_2}x_1 = e^{tV_1}x_1 = e^{tx_2\partial/\partial x_1}x_1 = x_1 + tx_2$$
$$e^{tV_2}x_2 = e^{t(-g(x_1)x_2 - f(x_1))\partial/\partial x_2}x_2 = x_2 e^{-tg(x_1)} + f(x_1)\frac{e^{-tg(x_1)} - 1}{g(x_1)}$$
$$e^{tV_1}e^{tV_2}x_2 = x_2 e^{-tg(x_1+tx_2)} + f(x_1 + tx_2)\frac{e^{-tg(x_1+tx_2)} - 1}{g(x_1 + tx_2)}$$

Here we used that if $h_1(\mathbf{x})$ and $h_2(\mathbf{x})$ are analytic functions then we have

$$e^{tV}h_1(\mathbf{x})h_2(\mathbf{x}) = h_1(e^{tV}\mathbf{x})h_2(e^{tV}\mathbf{x})\,.$$

Obviously the operator e^{tV} is also linear. We can also write

$$e^{tV_1}e^{tV_2}x_2 = x_2 e^{-tg(z)} - tf(z)\left(1 - \frac{t}{2!}g(z) + \frac{t^2}{3!}g^2(z) - \cdots\right)$$

where $z = x_1 + tx_2$. Thus we obtain the map

$$x_1(t) = x_1(0) + tx_2(0)$$
$$x_2(t) = x_2(0)e^{-tg(x_1(t))} + f(x_1(t))\frac{(e^{-tg(x_1(t))} - 1)}{g(x_1(t))}$$

or

$$x_1(t) = x_1(0) + tx_2(0)$$
$$x_2(t) = x_2(0)e^{-tg(x_1(t))} - tf(x_1(t))\left(1 - \frac{t}{2!}g(x_1(t)) + \frac{t^2}{3!}g^2(x_1(t)) - \cdots\right)$$

(ii) For $f(x) = x$ and $g(x) = \epsilon(x^2 - 1)$ we have

$$x_1(t) = x_1(0) + tx_2(0)$$
$$x_2(t) = x_2(0)e^{-t\epsilon(x_1^2(t)-1)} + x_1(t)\frac{(e^{-t\epsilon(x_1^2(t)-1)} - 1)}{\epsilon(x_1^2(t) - 1)}\,.$$

(iii) The program contains an `if` condition so that we can filter the transients.

```
// Lie.java

import java.awt.*;
import java.awt.event.*;
import java.awt.Graphics;

public class Lie extends Frame
{
   public Lie()
   {
   setSize(600,500);
   addWindowListener(new WindowAdapter()
   { public void windowClosing(WindowEvent event)
   { System.exit(0); }});
   }

   public void paint(Graphics g)
   {
   g.drawRect(40,50,500,440);
   double eps = 1.0;
   double x1, x2, x11, x22;
   double tau, count;
   tau = 0.005; count = 0.0;
   x11 = 0.5; x22 = 0.5; // initial conditions

   while(count < 1000.0)
   {
   x1 = x11; x2 = x22;
   x11 = x1 + tau*x2;
   double temp = Math.exp(-tau*eps*(x11*x11 - 1.0));
   x22 = x2*temp + x11*(temp - 1.0)/(eps*(x11*x11 - 1.0));


   if(count > 1.0)  // for transients to be decayed
   {
   int m = (int) (30.0*x1 + 250);
   int n = (int) (30.0*x2 + 250);
   int m1 = (int) (30.0*x11 + 250);
   int n1 = (int) (30.0*x22 + 250);
   g.drawLine(m,n,m1,n1);
   }
```

```
    count += tau;
    } // end while
    }

    public static void main(String[] args)
    {
    Frame f = new Lie();
    f.setVisible(true);
    }
}
```

Problem 4. We extend the differential equation described in the previous problem to

$$\frac{d^2x}{dt^2} + g(x)\frac{dx}{dt} + f(x) = h(t)$$

where $h(t) = k\cos(\omega t)$ is an external force. We can write this equation as an autonomous system of first-order differential equations

$$\begin{aligned}
\frac{dx_1}{dt} &= x_2 = V_1(x_1, x_2, x_3) \\
\frac{dx_2}{dt} &= -g(x_1)x_2 - f(x_1) + k\cos(x_3) = V_2(x_1, x_2, x_3) \\
\frac{dx_3}{dt} &= \omega = V_3(x_1, x_2, x_3)\,.
\end{aligned}$$

Thus we have the vector field

$$\begin{aligned}
V &= V_1(x_1, x_2, x_3)\frac{\partial}{\partial x_1} + V_2(x_1, x_2, x_3)\frac{\partial}{\partial x_2} + V_3(x_1, x_2, x_3)\frac{\partial}{\partial x_3} \\
&= x_2\frac{\partial}{\partial x_1} + (-g(x_1)x_2 - f(x_1))\frac{\partial}{\partial x_2} + \omega\frac{\partial}{\partial x_3}
\end{aligned}$$

given by this system of differential equations. Apply the method described in the previous problem to this system.

Solution 4. We have

$$\left[V_1(x_1, x_2, x_3)\frac{\partial}{\partial x_1}, \omega\frac{\partial}{\partial x_3}\right] = 0$$

and

$$\left[V_2(x_1, x_2, x_3)\frac{\partial}{\partial x_2}, \omega\frac{\partial}{\partial x_3}\right] = \omega k\sin(x_3)\frac{\partial}{\partial x_2}\,.$$

We use the approximation

$$e^{tV} \approx e^{tV_3}e^{tV_1}e^{tV_2}\,.$$

We find

$$
\begin{aligned}
e^{tV_2}x_1 &= e^{t(-g(x_1)x_2 - f(x_2) + k\cos(x_3))\partial/\partial x_2}x_1 = x_1 \\
e^{tV_1}e^{tV_2}x_1 &= e^{tV_1}x_1 = e^{tx_2\partial/\partial x_1}x_1 = x_1 + tx_2 \\
e^{tV_3}e^{tV_1}e^{tV_2}x_1 &= x_1 + tx_2 \\
e^{tV_2}x_2 &= e^{t(-g(x_1)x_2 - f(x_1) + k\cos(x_3))\partial/\partial x_2}x_2 \\
&= x_2 e^{-tg(x_1)} + f(x_1)\frac{e^{-tg(x_1)} - 1}{g(x_1)} + k\cos(x_3)\frac{e^{tg(x_1)} - 1}{g(x_1)} \\
e^{tV_1}e^{tV_2}x_2 &= x_2 e^{-tg(x_1+tx_2)} + f(x_1 + tx_2)\frac{e^{-tg(x_1+tx_2)} - 1}{g(x_1 + tx_2)} \\
&\quad + k\cos(x_3)\frac{e^{tg(x_1+tx_2)} - 1}{g(x_1 + tx_2)} \\
e^{tV_3}e^{tV_1}e^{tV_2}x_2 &= x_2 e^{-tg(x_1+tx_2)} + f(x_1 + tx_2)\frac{e^{-tg(x_1+tx_2)} - 1}{g(x_1 + tx_2)} \\
&\quad + k\cos(x_3 + \omega t)\frac{e^{tg(x_1+tx_2)} - 1}{g(x_1 + tx_2)}\,.
\end{aligned}
$$

Thus we obtain the map

$$
\begin{aligned}
x_3(t) &= x_3(0) + \omega t \\
x_1(t) &= x_1(0) + tx_2(0) \\
x_2(t) &= x_2(0)e^{-tg(x_1(t))} + f(x_1(t))\frac{(e^{-tg(x_1(t))} - 1)}{g(x_1(t))} \\
&\quad + k\cos(x_3(t))\frac{(e^{tg(x_1(t))} - 1)}{g(x_1(t))}\,.
\end{aligned}
$$

For $f(x) = x$ and $g(x) = \epsilon(x^2 - 1)$ we have

$$
\begin{aligned}
x_3(t) &= x_3(0) + \omega t \\
x_1(t) &= x_1(0) + tx_2(0) \\
x_2(t) &= x_2(0)e^{-t\epsilon(x_1^2(t)-1)} + x_1(t)\frac{(e^{-t\epsilon(x_1^2(t)-1)} - 1)}{\epsilon(x_1^2(t) - 1)} \\
&\quad + k\cos(x_3(t))\frac{e^{t\epsilon(x_1^2(t)-1)} - 1}{\epsilon(x_1^2(t) - 1)}\,.
\end{aligned}
$$

Problem 5. The *Verlet algorithm* is a method for integrating second order ordinary differential equations

$$\frac{d^2\mathbf{x}}{dt^2} = \mathbf{F}(\mathbf{x}(t), t)\,.$$

The Verlet algorithm is used in *molecular dynamics simulations.* It has a fixed time discretization interval h and it needs only one evaluation of the force $\mathbf{F}$ per step. The algorithm is derived by adding the *Taylor expansions* for the coordinates $\mathbf{x}$ at $t = \pm h$ about t

$$\mathbf{x}(h) = \mathbf{x}(0) + h\frac{d\mathbf{x}(0)}{dt} + \frac{h^2}{2}\mathbf{F}(\mathbf{x}(0),0) + \frac{h^3}{6}\frac{d^3\mathbf{x}(0)}{dt^3} + O(h^4)$$
$$\mathbf{x}(-h) = \mathbf{x}(0) - h\frac{d\mathbf{x}(0)}{dt} + \frac{h^2}{2}\mathbf{F}(\mathbf{x}(0),0) - \frac{h^3}{6}\frac{d^3\mathbf{x}(0)}{dt^3} + O(h^4)$$

leading to

$$\mathbf{x}(h) = 2\mathbf{x}(0) - \mathbf{x}(-h) + h^2\mathbf{F}(\mathbf{x}(0),0) + O(h^4)\,.$$

Knowing the values of $\mathbf{x}$ at time 0 and $-h$, this algorithm predicts the value of $\mathbf{x}(h)$. Thus we need the last two values of $\mathbf{x}$ to produce the next one. If we only have the initial position $\mathbf{x}(0)$ and initial velocity $\mathbf{v}(0)$ at our disposal, we approximate $\mathbf{x}(h)$ by

$$\mathbf{x}(h) \approx \mathbf{x}(0) + h\mathbf{v}(0) + \frac{h^2}{2}\mathbf{F}(\mathbf{x}(0),0)$$

i.e., we set

$$\mathbf{v}(0) = \frac{\mathbf{x}(0) - \mathbf{x}(-h)}{h}\,.$$

Discuss the Verlet method for the one-dimensional *harmonic oscillator*

$$\frac{d^2x}{dt^2} + \Omega^2 x = 0$$

where Ω is a constant frequency.

Solution 5. For the one-dimensional harmonic oscillator we find

$$x(t+h) = 2x(t) - x(t-h) - h^2\Omega^2 x(t)\,.$$

The analytic solution to this difference equation can be written in the form

$$x(t) = \exp(i\omega t)$$

with ω satisfying the condition

$$2 - 2\cos(\omega h) = h^2\Omega^2\,.$$

If $h^2\Omega^2 \equiv (h\Omega)^2 > 4$, the frequency ω becomes imaginary, and the analytical solution becomes unstable. Thus it is useful to introduce a dimensionless time τ via

$$\widetilde{x}(\tau(t)) = x(t)$$
$$\tau(t) = \Omega t\,.$$

Then the linear differential equation for the harmonic oscillator takes the form

$$\frac{d^2\tilde{x}}{d\tau^2} + \tilde{x} = 0\,.$$

Problem 6. Runge-Kutta methods were developed to avoid the computation of high order derivatives which the Taylor method may involve. In place of these derivatives extra values of the given function $\mathbf{g}(\mathbf{u}, t)$ are used in a way which essentially duplicates the accuracy of a Taylor polynomial. Suppose that the initial-value problem

$$\frac{d\mathbf{u}}{dt} = \mathbf{g}(\mathbf{u}, t), \qquad \mathbf{u}(t = 0) = \mathbf{u}_0$$

for an autonomous system of ordinary differential equations of first order is to be integrated, where $\mathbf{u}(t_0) = \mathbf{u}_0$. We present the formulas on which a *Runge-Kutta-Fehlberg method* of order five is based. A typical integration step approximates $\mathbf{u}$ at $t = t_0 + h$, where h is the step length. The formulas are

$$\mathbf{u}(t) = \mathbf{u}_0 + h\sum_{k=0}^{5} c_k \mathbf{g}^{(k)}, \qquad t = t_0 + h$$

with

$$\mathbf{g}^{(0)} = \mathbf{g}(\mathbf{u}_0), \qquad \mathbf{g}^{(k)} = \mathbf{g}(\mathbf{u}_0 + h\sum_{j=0}^{k-1} b_{kj}\mathbf{g}^{(j)}).$$

Thus $\mathbf{u}(t)$ approximates the exact solution. The coefficients c_0, c_1, ..., c_5 are

$$c_0 = \frac{16}{135}, \quad c_1 = 0, \quad c_2 = \frac{6656}{12825},$$
$$c_3 = \frac{28561}{56430}, \quad c_4 = -\frac{9}{50}, \quad c_5 = \frac{2}{55}\,.$$

The coefficients for b_{jk} $(j = 0, 1, \ldots, 5,\ k = 0, 1, \ldots, 4)$ are

$$b_{00} = b_{01} = b_{02} = b_{03} = b_{04} = 0$$

$$b_{10} = \frac{1}{4}, \quad b_{11} = b_{12} = b_{13} = b_{14} = 0$$

$$b_{20} = \frac{3}{32}, \quad b_{21} = \frac{9}{32}, \quad b_{22} = b_{23} = b_{24} = 0$$

$$b_{30} = \frac{1932}{2197}, \quad b_{31} = \frac{-7200}{2197}, \quad b_{32} = \frac{7296}{2197}, \quad b_{33} = b_{34} = 0$$

$$b_{40} = \frac{439}{216}, \quad b_{41} = -8, \quad b_{42} = \frac{3680}{513}, \quad b_{43} = -\frac{845}{4104}, \quad b_{44} = 0$$

$$b_{50} = -\frac{8}{27}, \quad b_{51} = 2, \quad b_{52} = -\frac{3544}{2565}, \quad b_{53} = \frac{1859}{4104}, \quad b_{54} = -\frac{11}{40}.$$

Apply the method to the *Lorenz model*

$$\begin{aligned}
\frac{du_1}{dt} &= \sigma(u_2 - u_1) \\
\frac{du_2}{dt} &= -u_1 u_3 + r u_1 - u_2 \\
\frac{du_3}{dt} &= u_1 u_2 - b u_3 .
\end{aligned}$$

Write a Java program that evaluates the phase portrait $(u_1(t), u_2(t))$ of the Lorenz model.

Solution 6. In the program `LorenzPhase.java`, we find the phase portrait $(u_1(t), u_2(t))$ of the Lorenz model for the initial values

$$u_1(0) = 0.8, \qquad u_2(0) = 0.8, \qquad u_3(0) = 0.8$$

and the parameter values $\sigma = 16$, $r = 40$ and $b = 4$.

```
// LorenzPhase.java

import java.awt.*;
import java.awt.event.*;
import java.awt.Graphics;

public class LorenzPhase extends Frame
{
  public LorenzPhase()
  {
  setSize(600,500);
  addWindowListener(new WindowAdapter()
  { public void windowClosing(WindowEvent event)
  { System.exit(0); }});
  }

  public void fsystem(double h,double t,double u[],double hf[])
  {
  double sigma = 16.0; double r = 40.0; double b = 4.0;
  hf[0] = h*sigma*(u[1] - u[0]);
  hf[1] = h*(-u[0]*u[2] + r*u[0] - u[1]);
  hf[2] = h*(u[0]*u[1] - b*u[2]);
  }
```

```
public void map(double u[],int steps,double h,double t,int N)
{
double uk[] = new double[N];
double tk;
double a[] = { 0.0,1.0/4.0,3.0/8.0,12.0/13.0,1.0,1.0/2.0 };
double c[] = { 16.0/135.0,0.0,6656.0/12825.0,28561.0/56430.0,
               -9.0/50.0,2.0/55.0 };
double b[][] = new double[6][5];
b[0][0] = b[0][1]= b[0][2] = b[0][3] = b[0][4] = 0.0;

b[1][0] = 1.0/4.0; b[1][1] = 0.0; b[1][2] = 0.0;
b[1][3] = 0.0; b[1][4] = 0.0;

b[2][0] = 3.0/32.0; b[2][1] = 9.0/32.0;
b[2][2] = 0.0; b[2][3] = 0.0; b[2][4] = 0.0;

b[3][0] = 1932.0/2197.0; b[3][1] = -7200.0/2197.0;
b[3][2] = 7296.0/2197.0; b[3][3] = b[3][4] = 0.0;

b[4][0] = 439.0/216.0; b[4][1] = -8.0;
b[4][2] = 3680.0/513.0; b[4][3] = -845.0/4104.0;
b[4][4] = 0.0;

b[5][0] = -8.0/27.0; b[5][1] = 2.0;
b[5][2] = -3544.0/2565.0; b[5][3] = 1859.0/4104.0;
b[5][4] = -11.0/40.0;

double f[][] = new double[6][N];
int i, j, l, k;

for(i=0;i<steps;i++)
{
fsystem(h,t,u,f[0]);

for(k=1;k<=5;k++)
{
tk = t + a[k]*h;
for(l=0;l<N;l++)
{
uk[l] = u[l];
for(j=0; j<=k-1; j++)
uk[l] += b[k][j]*f[j][l];
}
fsystem(h,tk,uk,f[k]);
```

```
  }
  for(l=0;l<N;l++)
  for(k=0;k<6;k++)
  u[l] += c[k]*f[k][l];
  }
  }

  public void paint(Graphics g)
  {
  g.drawLine(10,10,10,400);
  g.drawLine(10,200,630,200);
  g.drawRect(10,10,630,400);
  int steps = 1;
  int N = 3;
  double h = 0.005;
  double t = 0.0;
  double u[] = { 0.8, 0.8, 0.8 };  // initial conditions

  // wait for transients to decay
  for(int i=0;i<1000;i++)
  {
  t += h;
  map(u,steps,h,t,N);
  }
  t = 0.0;
  for(int i=0;i<4800;i++)
  {
  t += h;
  map(u,steps,h,t,N);
  int m = (int) (5.0*u[0] + 300);
  int n = (int) (5.0*u[1] + 200);
  g.drawLine(m,n,m,n);
  }
  }

  public static void main(String[] args)
  {
  Frame f = new LorenzPhase();
  f.setVisible(true);
  }
}
```

Chapter 12

Partial Differential Equations

Problem 1. Consider the partial differential equation (one-dimensional *wave equation*)

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \tag{1}$$

with the initial conditions

$$u(x,0) = f(x), \quad 0 \le x \le \ell \tag{2a}$$

$$\frac{\partial u(x,0)}{\partial t} = g(x), \quad 0 \le x \le \ell \tag{2b}$$

and the boundary conditions

$$u(0,t) = L(t), \quad t \ge 0 \tag{3a}$$

$$u(\ell,t) = R(t), \quad t \ge 0\,. \tag{3b}$$

The function u of x and t gives the amplitude of the string at position x and t. Discretize the partial differential equation and the initial and boundary conditions and formulate the discretized version as an initial value problem. For the discretization use

$$\frac{df}{dx} \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$$
$$\frac{d^2 f}{dx^2} \approx \frac{f(x+\Delta x) - 2f(x) + f(x-\Delta x)}{(\Delta x)^2}\,.$$

Solution 1. Using the notation

$$u_{m,n} \equiv u(m\Delta x, n\Delta t) \equiv u(x_m, t_n)$$

we find using the discretization given above

$$\frac{\partial^2 u(x_m,t_n)}{\partial t^2} \approx \frac{u(x_m,t_n+\Delta t)-2u(x_m,t_n)+u(x_m,t_n-\Delta t)}{(\Delta t)^2}$$
$$= \frac{u_{m,n+1}-2u_{m,n}+u_{m,n-1}}{(\Delta t)^2}$$

and

$$\frac{\partial^2 u(x_m,t_n)}{\partial x^2} \approx \frac{u(x_m+\Delta x,t_n)-2u(x_m,t_n)+u(x_m-\Delta x,t_n)}{(\Delta x)^2}$$
$$= \frac{u_{m+1,n}-2u_{m,n}+u_{m-1,n}}{(\Delta x)^2}\,.$$

Substituting these expressions into the wave equation (1) yields

$$\frac{u_{m,n+1}-2u_{m,n}+u_{m,n-1}}{(\Delta t)^2} = c^2\frac{u_{m+1,n}-2u_{m,n}+u_{m-1,n}}{(\Delta x)^2}\,. \tag{4}$$

We set

$$\epsilon := \frac{c^2(\Delta t)^2}{(\Delta x)^2}\,.$$

Writing (4) as a recursion with respect to n (time) we have

$$u_{m,n+1} = \epsilon(u_{m+1,n}+u_{m-1,n}) + 2(1-\epsilon)u_{m,n} - u_{m,n-1}\,. \tag{5}$$

To generate approximate values of $u(x,t)$ at time t_{n+1} we must know approximate values at times t_n and t_{n-1}. Suppose that $u_{m,0}$ and $u_{m,1}$ are know for all $0 \le m \le M$. Then (5) with $n=1$

$$u_{m,2} = \epsilon(u_{m+1,1}+u_{m-1,1}) + 2(1-\epsilon)u_{m,1} - u_{m,0}$$

determines $u_{m,2}$ for all $1 \le m \le M-1$. When $1 \le m \le M-1$, the subscripts $m+1$ and $m-1$, which appear on the right-hand side of (5), are both between 0 and M. The two boundary conditions determine $u_{0,2}$ and $u_{M,2}$, respectively. At this point $u_{m,n}$ is known for every $0 \le m \le M$ and $n \le 2$. Then (5) with $n=2$ yields $u_{m,3}$ for all $1 \le m \le M-1$. Again the boundary conditions determine $u_{0,3}$ and $u_{M,3}$ and so on. To generate approximate values of $u(x,t)$ (using (5)) at time $t_1 = \Delta t$ it is necessary to know approximate values at times $t_0 = 0$ and $t_{-1} = -\Delta t$. The first initial condition (2a) provides

$$u_{m,0} = u(m\Delta x, 0) = f(m\Delta x) \tag{6}$$

and the second speed initial condition (2b) provides us indirectly and approximately $u_{m,-1}$. The simplest approximation would be

$$g(x) = \frac{\partial u(x,0)}{\partial t} = \lim_{h\to 0} \frac{u(x,0) - u(x,-h)}{h} \approx \frac{u(x,0) - u(x,-\Delta t)}{\Delta t}.$$

Thus

$$u_{m,-1} = u(m\Delta x, -\Delta t) \approx u(m\Delta x, 0) - g(m\Delta x)\Delta t = u_{m,0} - g(m\Delta x)\Delta t.$$

We obtain a more accurate approximation by using

$$g(x) = \frac{\partial u(x,0)}{\partial t} = \lim_{h\to 0} \frac{u(x,h) - u(x,-h)}{2h} \approx \frac{u(x,\Delta t) - u(x,-\Delta t)}{2\Delta t}.$$

Thus

$$\begin{aligned} u_{m,-1} &= u(m\Delta x, -\Delta t) \\ &\approx u(m\Delta x, \Delta t) - 2g(m\Delta x)\Delta t \\ &= u_{m,1} - 2g(m\Delta x)\Delta t. \end{aligned}$$

Substituting this equation into the difference equation (4) with $n = 0$ yields

$$\begin{aligned} u_{m,1} &= \epsilon(u_{m+1,0} + u_{m-1,0}) + 2(1-\epsilon)u_{m,0} - u_{m,-1} \\ &= \epsilon(u_{m+1,0} + u_{m-1,0}) + 2(1-\epsilon)u_{m,0} - u_{m,1} + 2g(m\Delta x)\Delta t. \end{aligned}$$

Thus

$$u_{m,1} = \frac{1}{2}\epsilon(u_{m+1,0} + u_{m-1,0}) + (1-\epsilon)u_{m,0} + g(m\Delta x)\Delta t. \qquad (7)$$

Equations (6) and (7) give $u_{m,n}$ for all $n = 0, 1$ and $1 \le m \le M-1$. Thus we have the algorithm:

Set for all $1 \le m \le M-1$

$$u_{m,0} = f(m\Delta x)$$

and

$$u_{m,1} = \frac{1}{2}\epsilon(u_{m+1,0} + u_{m-1,0}) + (1-\epsilon)u_{m,0} + g(m\Delta x)\Delta t.$$

Then for each iteration step $n = 1, 2, \ldots$, we set

$$u_{m,n+1} = \epsilon(u_{m+1,n} + u_{m-1,n}) + 2(1-\epsilon)u_{m,n} - u_{m,n-1}$$

for all $1 \le m \le M-1$. Whenever $u_{0,k}$ or $u_{M,k}$ is encountered on the right-hand sides of these formulas we apply

$$u_{0,k} = L(k\Delta t), \qquad u_{M,k} = R(k\Delta t).$$

How do we choose ϵ?

Problem 2. *Maxwell's equations* in free space are given by

$$\nabla \times \mathbf{B} = \epsilon_0 \mu_0 \frac{\partial \mathbf{E}}{\partial t}$$
$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$$
$$\nabla \cdot \mathbf{E} = 0$$
$$\nabla \cdot \mathbf{B} = 0$$

where $\mathbf{B}$ is the magnetic induction, $\mathbf{E}$ is the electric field intensity and

$$\nabla \times \mathbf{B} = \begin{pmatrix} \dfrac{\partial B_3}{\partial y} - \dfrac{\partial B_2}{\partial z} \\ \dfrac{\partial B_1}{\partial z} - \dfrac{\partial B_3}{\partial x} \\ \dfrac{\partial B_2}{\partial x} - \dfrac{\partial B_1}{\partial y} \end{pmatrix}.$$

We assume that $B_1 = B_3 = 0$, $E_2 = E_3 = 0$, B_2 only depends on z and t and E_1 only depends on z and t.
(i) Find Maxwell's equations for this special case.
(ii) Thus we choose the z-direction for direction of propagation of the electromagnetic field, the x-direction for the polarization of the electric field intensity. Then it follows that the magnetic induction is y-polarized. Discretize these equations using the *central difference approximation* for the derivative of time and space

$$\frac{\partial V}{\partial t} \to \frac{\Delta V}{\Delta t} \equiv \frac{V(n+1/2) - V(n-1/2)}{\Delta t}$$
$$\frac{\partial V}{\partial z} \to \frac{\Delta V}{\Delta z} \equiv \frac{V(k+1/2) - V(k-1/2)}{\Delta z}$$

where the temporal steps are indexed by the integer n and related to the continuous time by $t = n\Delta t$, and the spatial steps are indexed by the integer k and related to continuous space by $z = k\Delta z$. Discuss the accuracy.

Solution 2. (i) We find

$$\frac{\partial E_1}{\partial t} = -c^2 \frac{\partial B_2}{\partial z}$$
$$\frac{\partial B_2}{\partial t} = -\frac{\partial E_1}{\partial z}$$

where $\epsilon_0 \mu_0 = 1/c^2$.

(ii) The discretization yields

$$\frac{E_1(n+1/2,k)-E_1(n-1/2),k}{\Delta t} = -c^2\frac{B_2(n,k+1/2)-B_2(n,k-1/2)}{\Delta z}$$
$$\frac{B_2(n+1/2,k)-B_2(n-1/2,k)}{\Delta t} = -\frac{E_1(n,k+1/2)-E_1(n,k-1/2)}{\Delta z}.$$

In the method of finite difference time domain simulation we need only to consider the two curl equations, because the divergence conditions can be satisfied implicitly by interleaving the electric field intensity and magnetic induction components in space. A consequence of the spatial interleaving is that the fields must also be interleaved in time, known as leapfrog, since the temporal response of one field is proportional to the spatial variation of the other at the previous time step. Thus we obtain for the second equation with $n \to n+1/2$ and $k \to k+1/2$

$$\frac{B_2(n+1,k+1/2)-B_2(n,k+1/2)}{\Delta t} = -\frac{E_1(n+1/2,k+1)-E_1(n+1/2,k)}{\Delta z}.$$

To yield accurate results, the grid spacing δ in the finite difference simulation must be less than the wavelength λ of the electromagnetic wave, usually less then $\lambda/10$. The stability condition relating the spatial and temporal step size is

$$v_{max}\Delta t = \left(\frac{1}{(\Delta x)^2}+\frac{1}{(\Delta y)^2}+\frac{1}{(\Delta z)^2}\right)^{-1/2}$$

where v_{max} is the maximum velocity of the wave.

Problem 3. When doing numerical calculations for partial differential equations we must establish an upper bound on the growth of the solution. We consider one space dimension. In the continuous case the energy method is often used with the usual scalar product in the Hilbert space $L_2[a,b]$

$$\langle u,v\rangle := \int_a^b u(x)v(x)dx\,.$$

This implies a norm $\|u\| := \langle u,u\rangle^{1/2}$. Consider the simplest hyperbolic partial differential equation

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial x}.$$

(i) Show that the energy growth in time is governed by the boundary values a and b, i.e., we have to calculate

$$\frac{1}{2}\frac{d\|u\|^2}{dt}.$$

(ii) Extend to the discrete case.

Solution 3. (i) We find

$$\frac{1}{2}\frac{d\|u\|^2}{dt} = \langle u, u_t\rangle = \langle u, u_x\rangle = \frac{1}{2}u^2|_a^b$$

where we used the notation $u_t = \partial u/\partial t$, $u_x = \partial u/\partial x$ and *integration by parts*

$$\int_a^b \frac{\partial u}{\partial x}v(x)dx = -\int_a^b u\frac{\partial v}{\partial x}dx + u(x)v(x)|_a^b\,.$$

(ii) In the discrete case we have to find an operator D and a scalar product with a matrix H that approximates the derivative d/dx and the integral $\int_a^b$ with the same properties as the continuous case. Thus integration by parts is replaced by

$$\langle u, Dv\rangle_h = u_n v_n - u_0 v_0 - \langle Du, v\rangle_h$$

where $\langle u, v\rangle_h := hu^T Hv$ and ^{T} means transpose. An example of such a matrix operator D (tridiagonal matrix) and scalar product matrix H is

$$D = \frac{1}{h}\begin{pmatrix} -1 & 1 & & & \\ -0.5 & 0 & 0.5 & & \\ & \ddots & \ddots & \ddots & \\ & & -0.5 & 0 & 0.5 \\ & & & -1 & 1 \end{pmatrix}$$

and the diagonal matrix $H = \mathrm{diag}(0.5,\ 1,\ \ldots\ 1,\ 0.5)$.

Problem 4. The *inviscid Burgers equation* is given by

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0\,.$$

(i) Show that the partial differential equation can be written as a scalar *conservation law*

$$\frac{\partial u}{\partial t} + \frac{\partial(f(u))}{\partial x} = 0\,.$$

(ii) Discuss the solution.
(iii) Use the truncated Taylor expansion

$$\left.\frac{\partial u}{\partial x}\right|_i \approx \frac{u_i - u_{i-1}}{\Delta x} + O(\Delta x)$$

$$\left.\frac{\partial u}{\partial t}\right|_n \approx \frac{u_{n+1} - u_n}{\Delta t} + O(\Delta t)$$

to approximate the inviscid Burgers equation. Disucss the problem with this approximation.

Solution 4. (i) We can write the equation as conservation law

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\left(\frac{u^2}{2}\right) = 0\,.$$

(ii) We have a hyperbolic differential equation. The information propagates along characteristic curves. u is constant on the characteristic curve and u is the slope of the characteristic curve and where the characteristics cross we have shock formation (weak solution).
(iii) The approximation

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0 \Rightarrow \frac{u_{i,n+1} - u_{i,n}}{\Delta t} + u_{i,n}\frac{u_{i,n} - u_{i-1,n}}{\Delta x} = 0$$

converges to an incorrect solution. The reason is that the Taylor expansion is not valid at the shock.

Problem 5. In many practical situations a hyperbolic partial differential equation appears in the form of a conservation law

$$\frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x} = 0$$

which can be written as

$$\frac{\partial u}{\partial t} + g(u)\frac{\partial u}{\partial x} = 0$$

where $g(u) = \partial f/\partial u$. If $f(u) = u^2/2$ we obtain the inviscid Burgers equation. For the *Lax-Wendroff method* we look at the Taylor expansion of the partial differential equation with respect to t. Starting from

$$\frac{\partial u}{\partial t} = -\frac{\partial}{\partial x} f(u)$$

we obtain

$$\frac{\partial^2 u}{\partial t^2} = -\frac{\partial}{\partial t}\frac{\partial}{\partial x} f(u) = -\frac{\partial}{\partial x}\frac{\partial}{\partial t} f(u) = -\frac{\partial}{\partial x}\left(g(u)\frac{\partial u}{\partial t}\right).$$

Thus

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(g(u)\frac{\partial f(u)}{\partial x}\right).$$

Replacing the x-derivative by central differences we obtain

$$U_j^{n+1} = U_j^n - \frac{\Delta t}{\Delta x}\Delta_{0,x} f(U_j^n) + \frac{1}{2}\left(\frac{\Delta t}{\Delta x}\right)^2 \delta_x\left(g(U_j^n)\delta_x f(U_j^n)\right)$$

where

$$U_j^n \approx u(x_j, t_n)$$

and we use the definitions

$$\begin{aligned}
\delta_x v(x,t) &:= v(x + \frac{1}{2}\Delta x, t) - v(x - \frac{1}{2}\Delta x, t) \quad \text{central difference} \\
\Delta_{0x} v(x,t) &:= \frac{1}{2}(v(x + \Delta x, t) - v(x - \Delta x, t)) \\
\Delta_{+x} v(x,t) &:= v(x + \Delta x, t) - v(x,t) \quad \text{forward difference} \\
\Delta_{-x} v(x,t) &:= v(x,t) - v(x - \Delta x, t) \quad \text{backward difference.}
\end{aligned}$$

If we expand the last term we find that it contains the values of $g(U_{j-1/2}^n)$ and $g(U_{j+1/2}^n)$. In evaluating this, one sets

$$U_{j\pm 1/2}^n := \frac{1}{2}(U_j^n + U_{j\pm 1}^n)\,.$$

Thus the scheme becomes

$$\begin{aligned}
U_j^{n+1} = U_j^n - \frac{1}{2}\frac{\Delta t}{\Delta x}\Bigg(&\left(1 - g(U_{j+1/2}^n)\frac{\Delta t}{\Delta x}\right)\Delta_{+x} f(U_j^n) \\
&+ \left(1 + g(U_{j-1/2}^n)\frac{\Delta t}{\Delta x}\right)\Delta_{-x} f(U_j^n)\Bigg)\,.
\end{aligned}$$

Write a C++ program that implements the Lax-Wendroff method for the Burgers equation

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}$$

where ν is the kinematic viscosity.

Solution 5.

```
// LaxWendroff.cpp

#include <iostream>
#include <cmath>      // for fabs()
using namespace std;

int main(void)
{
  double L = 1.0;    // size of periodic region
  int N = 200;       // number of grid points
  double h;          // lattice spacing
  double t;          // time step length
```

```
double nu = 0.0;  // kinematic viscosity

// allocating memory
double* u = new double[N];    // the solution
double* uNew = new double[N]; // for updating
double* F = new double[N];    // for flow

h = L/N;                      // grid length
double uMax = 0.0;
int i, j, k;
// initial distribution
for(i=0;i<N;i++)
{
double x = i*h;
u[i] = x;
if(fabs(u[i]) > uMax) uMax = fabs(u[i]);
}

t = h/uMax;
cout << "t = " << t << endl;
int steps = 50000;
int count = 0;

while(count < steps)
{
for(j=0;j<N;j++)
F[j] = u[j]*u[j]/2.0;
for(j=0;j<N;j++)
{
int jMinus1 = j > 0 ? j-1 : N-1;
int jPlus1 = j < N-1 ? j+1 : 0;
int jPlus2 = jPlus1 < N-1 ? jPlus1+1 : 0;
uNew[j] =
  (u[j] + u[jPlus1])/2.0 -
  (t/(2.0*h))*(F[jPlus1]-F[j]) +
  (nu*t/(2.0*h*h))*((u[jPlus1]+u[jMinus1]-2.0*u[j])/2.0
  + (u[jPlus2]+u[j]-2.0*u[jPlus1])/2.0);
} // end j loop
for(j=0;j<N;j++) F[j] = uNew[j]*uNew[j]/2.0;
for(j=0;j<N;j++)
{
int jMinus1 = j > 0 ? j-1 : N-1;
int jPlus1 = j < N-1 ? j+1 : 0;
uNew[j] = u[j] - (t/h)*(F[j]-F[jMinus1]) +
```

```
            (nu*t/(h*h))*(u[jPlus1]+u[jMinus1]-2.0*u[j]);
  }
  for(i=0;i<N;i++) u[i] = uNew[i];
  count++;
  } // end while

  // solution
  for(i=0;i<N;i++)
  {
  cout << "u[" << i << "] = " << u[i] << endl;
  }
  delete[] u; delete[] uNew; delete[] F;
  return 0;
}
```

Problem 6. Consider the one-dimensional nonlinear Schrödinger equation

$$i\frac{\partial u}{\partial t} + \frac{\partial^2 u}{\partial x^2} + q|u|^2 u = 0 \tag{1}$$

together with the periodic boundary condition $u(x,t) = u(x+L,t)$. Here $u(x,t)$ is a complex valued function and q is a real parameter. Give two different discretizations for the space variable x.

Solution 6. Since the second order derivative of a function f can be defined as

$$\frac{d^2 f}{dx^2} = \lim_{h\to 0} \frac{f(x-h) - 2f(x) + f(x+h)}{h^2}$$

we can approximate the partial differential equation (1) as

$$i\frac{d}{dt}U_j + \frac{U_{j-1} - 2U_j + U_{j+1}}{h^2} + q|U_j|^2 U_j = 0$$

where $U_{j+N} = U_j$, $h = L/N$, i.e., N denotes the number of grid points and $x_j = jh$ with $U_j(t)$ denoting the numerical approximation of $u(x_j,t)$. Another approximation would be by modifying the last term $q|u|^2u$ to

$$\frac{1}{2}q|U_j|^2(U_{j-1} + U_{j+1})$$

and leaving the approximation for the first two terms the same.

Chapter 13

Wavelets

Problem 1. In many applications, given a signal in the time domain $f(t)$, one is interested in its frequency content locally in time, the standard Fourier transformation cannot provide this time localization. The *wavelet transform*

$$W_{a,b}(f) = |a|^{-1/2} \int_{-\infty}^{\infty} f(t)\psi\left(\frac{t-b}{a}\right) dt$$

provides a time localization. One assumes that

$$\int_{-\infty}^{\infty} \psi(t)dt = 0\,.$$

The functions

$$\psi_{a,b}(s) = |a|^{-1/2}\psi\left(\frac{s-b}{a}\right)$$

are called *wavelets* and the function $\psi(s)$ is called *mother wavelet.* As a changes, the $\psi_{a,b}$ cover different frequency ranges, changing the parameter as well allows us to move the time location center (which is in $s = b$). Thus all the wavelets are translated and dilated versions of one function the mother wavelet. Examples of mother wavelets are

$$\psi(t) = (1-t^2)\exp(-t^2/2) \qquad \textit{Mexican hat wavelet}$$

$$\psi(t) = \begin{cases} 1 & \text{for} \quad 0 \le t \le 1/2 \\ -1 & \text{for} \quad 1/2 < t < 1 \\ 0 & \text{otherwise} \end{cases} \qquad \textit{Haar wavelet}.$$

Discuss the localization of these wavelets in the time domain and frequency domain.

Solution 1. The Haar wavelet is very well localized in the time-domain but badly localized in frequency domain due to the appearance of side-bands in the Fourier spectrum. In contrast, the frequency localization of the Mexican hat wavelet is much better but at the cost of the time localization (*uncertainty principle*).

Problem 2. The Dirichlet-Gabor distributed approximating functional wavelet is given by

$$\phi(x) = C_{M,\sigma} \exp\left(\frac{-x^2}{2\sigma^2}\right)\left(\frac{1}{2} + \sum_{k=1}^{M} \cos(kx)\right).$$

(i) Can this expression be simplified?
(ii) Let $\hat{\phi}$ be the Fourier transform of ϕ. The constant $C_{M,\sigma}$ is determined by $\hat{\phi}(0) = 1$. Find $C_{M,\sigma}$.

Solution 2. (i) Since

$$\frac{1}{2} + \sum_{k=1}^{M} \cos(kx) = \frac{\sin(M + \frac{1}{2})x}{2\sin(x/2)}$$

the function ϕ takes the form

$$\phi(x) = C_{M,\sigma} \exp\left(\frac{-x^2}{2\sigma^2}\right) \frac{\sin(M + \frac{1}{2})x}{2\sin(x/2)}.$$

(ii) Since

$$\hat{\phi}(k) = \int_{-\infty}^{\infty} \phi(x) e^{ikx} dx$$

we have

$$\hat{\phi}(0) = \int_{-\infty}^{\infty} \phi(x) dx = 1.$$

Thus we find

$$C_{M,\sigma}\sqrt{2\pi}\sigma\left(\frac{1}{2} + \sum_{k=1}^{M} \exp\left(-\frac{\sigma^2 k^2}{2}\right)\right) = 1.$$

Problem 3. Consider the Hilbert space $L_2(\mathbf{R})$. Assume that the function $\phi \in L_2(\mathbf{R})$ satisfies

$$\int_{-\infty}^{\infty} \phi(t)\overline{\phi(t-k)} dt = \delta_{0,k}$$

i.e., the integral equals 1 for $k = 0$ and vanishes for $k = \pm 1, \pm 2, \ldots$. Show that for any fixed integer $j \in \mathbf{Z}$ the functions

$$\phi_{jk}(t) := 2^{j/2}\phi(2^j t - k), \qquad k \in \mathbf{Z}$$

form an orthonormal set.

Solution 3. We have

$$\int_{-\infty}^{\infty} \phi_{jk}(t)\overline{\phi_{j\ell}(t)}dt = \int_{-\infty}^{\infty} 2^{j/2}\phi(2^j t - k)\overline{2^{j/2}\phi(2^j t - \ell)}dt\,.$$

Let $x := 2^j t - k$. Then $dx = 2^j dt$ and

$$\int_{-\infty}^{\infty} \phi_{jk}(t)\overline{\phi_{j\ell}(t)}dt = \int_{-\infty}^{\infty} \phi(x)\overline{\phi(x + k - \ell)}dx = \delta_{0,\ell-k}\,.$$

When $\ell = k$, then

$$\int_{-\infty}^{\infty} \phi_{jk}(x)\overline{\phi_{j\ell}(x)}dx = 1$$

otherwise

$$\int_{-\infty}^{\infty} \phi_{jk}(x)\overline{\phi_{j\ell}(x)}dx = 0\,.$$

Thus the set forms an orthonormal set.

Problem 4. Consider the Hilbert space $L_2[0,1]$. Let ϕ be the *Haar scaling function*

$$\phi(t) := \begin{cases} 1 & \text{if} \quad 0 \le t < 1 \\ 0 & \text{otherwise} \end{cases}\,.$$

Let n be a positive integer and

$$g_k(t) := \sqrt{n}\phi(nt - k), \qquad k = 0, 1, 2, \ldots, n-1\,.$$

(i) Show that $\{\, g_0, g_1, \ldots, g_{n-1} \,\}$ is an orthonormal set in the Hilbert space $L_2[0,1]$.
(ii) Let f be a continuous function on $[0,1]$ and form the projection f_n on the subspace $S_n[0,1]$ of $L_2[0,1]$ spanned by $\{\, g_0, g_1, \ldots, g_{n-1} \,\}$

$$f_n(t) = \sum_{k=0}^{n-1} \langle f, g_k \rangle g_k(t)$$

where the scalar product is defined by

$$\langle f, g_k \rangle := \int_0^1 f(t)g_k(t)dt\,.$$

Show that $f_n(t) \to f(t)$ pointwise in t as $n \to \infty$.

Solution 4. (i) For $k \neq \ell$ we have $g_k(t)g_\ell(t) = 0$ for all $t \in (0,1)$. Thus for $k \neq \ell$ we have

$$\int_0^1 g_k(t)g_\ell(t)dt = 0\,.$$

For $k = \ell$ we have

$$\int_0^1 g_k^2(t)dt = n\int_{k/n}^{(k+1)/n} dt = 1\,.$$

(ii) We have

$$f_n(t) = \sum_{k=0}^{n-1} n\left(\int_{k/n}^{(k+1)/n} f(\tau)d\tau\right)\phi(nt-k)\,.$$

Hence

$$f_n(t) = n\int_{k/n}^{(k+1)/n} f(\tau)d\tau \quad \text{for} \quad \frac{k}{n} \leq t < \frac{k+1}{n}\,.$$

By the *mean value theorem of calculus*, there is a point $t_{n,k}$ in the interval $\left(\frac{k}{n}, \frac{k+1}{n}\right)$ such that the integral expression on the right-hand side is equal to $f(t_{n,k})$. Thus

$$f_n(t) = f(t_{n,k}) \quad \text{for} \quad \frac{k}{n} \leq t < \frac{k+1}{n}\,.$$

Since f is continuous on the closed bounded set $[0,1]$, it is uniformly continuous on this set. Thus for any $\epsilon > 0$ there is a $\delta(\epsilon) > 0$ such that $|f(t) - f(t')| < \epsilon$ whenever $|t - t'| < \delta(\epsilon)$. Now given $t \in [0,1)$ and $\epsilon > 0$, choose $n > 1/\delta(\epsilon)$. Then

$$|f(t) - f_n(t)| = |f(t) - f(t_{n,k})| < \epsilon$$

since

$$|t - t_{n,k}| < \frac{1}{n} < \delta(\epsilon)$$

so that $f_n(t) \to f(t)$, uniformly in t as $n \to \infty$.

Problem 5. We consider the Hilbert space $L_2(\mathbf{R})$. Expand the step function

$$f(x) := \begin{cases} -1 & x \in [-1/2, 0] \\ 1 & x \in (0, 1/2] \end{cases}$$

with respect to the *Haar basis*

$$\{\ \psi_{j,k}(x) := 2^{j/2}\psi(2^j x - k)\ :\ j,k \in \mathbf{Z}\ \}$$

and

$$\psi(x) = \begin{cases} -1 & x \in [0, 1/2] \\ 1 & x \in (1/2, 1] \\ 0 & \text{otherwise} \end{cases}.$$

Calculate the expansion coefficients $\langle f(x), \psi_{j,k}(x) \rangle$ where the scalar product is defined by

$$\langle f(x), g(x) \rangle := \int_{-\infty}^{\infty} f(x)\overline{g(x)}dx.$$

Thus we determine

$$f(x) = \sum_{j,k \in \mathbf{Z}} \langle f(x), \psi_{jk}(x) \rangle \psi_{j,k}(x).$$

Solution 5. We have

$$\psi_{j,k}(x) = \begin{cases} -2^{j/2} & x \in \left[k2^{-j}, (k+\frac{1}{2})2^{-j}\right] \\ 2^{j/2} & x \in \left((k+\frac{1}{2})2^{-j}, (k+1)2^{-j}\right] \\ 0 & \text{otherwise} \end{cases}.$$

It is easy to see that

$$f(x) = -\psi_{0,0}(x) - \psi_{0,-1}(x)$$

for $x \in (-\frac{1}{2}, \frac{1}{2})$. To evaluate the integrals we are only interested in the intervals $\left[k2^{-j}, (k+\frac{1}{2})2^{-j}\right]$ and $\left((k+\frac{1}{2})2^{-j}, (k+1)2^{-j}\right]$ which have a non-empty intersection with $[-1/2, 1/2]$. Furthermore, if the union of the intervals is completely contained in $[-1/2, 1/2]$ the corresponding coefficient in the expansion will be zero. Thus we need only consider the intervals of length $\geq \frac{1}{2}$ which corresponds to $j < 0$ which have 0 as one of the boundaries ($k \in \{0, -1\}$). We find

$$\langle f, \psi_{j,k} \rangle = \begin{cases} -2^{j/2-1} & j < 0 \quad k \in \{0, -1\} \\ 0 & \text{otherwise} \end{cases}.$$

Thus we find

$$f(x) = \sum_{j<1} (-2^{j/2-1})(\psi_{j,0}(x) + \psi_{j,-1}(x))$$

where $x \in \mathbf{R}$. We note that

$$\sum_{j<1} 2^{j-1} = \frac{1}{2} \sum_{j=0}^{\infty} 2^{-j} = 1.$$

Problem 6. We consider the *discrete wavelet transform* of an orthogonal function which can be applied to a finite group of data. The input signal is assumed to be a set of discrete-time samples. The transform is a convolution. The wavelet basis is a set of functions which are defined by a recursive difference equation for the *scaling function* ϕ

$$\phi(x) = \sum_{k=0}^{M-1} c_k \phi(2x - k)$$

where the range of the summation is determined by the specified number of nonzero coefficients M. Here k is the translation parameter. The number of coefficients is not arbitrary and is determined by constraints of orthogonality and normalization. Owing to the periodic boundary condition we have $c_k :\equiv c_{k+nM}$, where $n \in \mathbf{N}$. Periodic wavelets are only one possibility to deal with signals on an interval. Generally, the area under the scaling function over all space should be unity, i.e.,

$$\int_{\mathbf{R}} \phi(x) dx = 1 .$$

It follows that

$$\sum_{k=0}^{M-1} c_k = 2 .$$

In the Hilbert space $L_2(\mathbf{R})$, the scaling function ϕ is orthogonal to its translations; i.e.,

$$\int_{\mathbf{R}} \phi(x)\phi(x - k) dx = 0, \qquad k \neq 0 .$$

What is desired is a function ψ which is also orthogonal to its dilations, or scales, i.e.,

$$\int_{\mathbf{R}} \psi(x)\psi(2x - k) dx = 0 .$$

Such a function ψ does exist and is given by (the so-called *associated wavelet function*)

$$\psi(x) = \sum_{k=1} (-1)^k c_{1-k} \phi(2x - k)$$

which is dependent on the solution of ϕ. The following equation follows from the orthonormality of scaling functions

$$\sum_k c_k c_{k-2m} = 2\delta_{0m}$$

which means that the above sum is zero for all m not equal to zero, and that the sum of the squares of all coefficients is two. Another equation which can be derived from $\psi(x) \perp \phi(x-m)$ is

$$\sum_k (-1)^k c_{1-k} c_{k-2m} = 0 .$$

A way to solve for ϕ is to construct a matrix of coefficient values. This is a square $M \times M$ matrix where M is the number of nonzero coefficients. The matrix is designated L with entries $L_{ij} = c_{2i-j}$. This matrix has an eigenvalue equal to 1, and its corresponding (normalized) eigenvector contains, as its components, the value of the function ϕ at integer values of x. Once these values are known, all other values of the function ϕ can be generated by applying the recursion equation to get values at half-integer x, quarter-integer x, and so on down to the desired dilation. This determines the accuracy of the function approximation. Give an example for the scaling function ϕ and the associated wavelet function ψ.

Solution 6. An example for ψ is the *Haar function*

$$\psi(x) := \begin{cases} 1 & 0 \le x < \frac{1}{2} \\ -1 & \frac{1}{2} \le x < 1 \\ 0 & \text{otherwise} \end{cases}$$

and the scaling function ϕ is given by

$$\phi(x) := \begin{cases} 1 & 0 \le x < 1 \\ 0 & \text{otherwise} \end{cases} .$$

The functions

$$\psi_{m,n}(x) := 2^{\frac{m}{2}} \psi(2^m x - n), \quad m, n \in \mathbf{Z}$$

form a basis in the Hilbert space $L_2(\mathbf{R})$. If we restrict m to $m = 0, 1, 2, \ldots$ and $n = 0, 1, 2, \ldots, 2^m - 1$ we obtain a basis in the Hilbert space $L_2[0, 1]$.

Problem 7. This class of wavelet functions is constrained, by definition, to be zero outside of a small interval. This is what makes the wavelet transform able to operate on a finite set of data, a property which is formally called *compact support*. The recursion relation ensures that a scaling function ϕ is non-differentiable everywhere. Of course this is not valid for Haar wavelets. The following table lists coefficients for two wavelet transforms. The *pyramid algorithm* operates on a finite set on N input data

$$f_0, f_1, \ldots, f_{N-1}$$

Wavelet	c_0	c_1	c_2	c_3
Haar	1.0	1.0		
Daubechies-4	$(1+\sqrt{3})/4$	$(3+\sqrt{3})/4$	$(3-\sqrt{3})/4$	$(1-\sqrt{3})/4$

Table 13.1: Coefficients for Two Wavelet Functions

where N is a power of two; this value will be referred to as the input block size. These data are passed through two convolution functions, each of which creates an output stream that is half the length of the original input. These convolution functions are filters, one half of the output is produced by the "low-pass" filter

$$a_i = \frac{1}{2}\sum_{j=0}^{N-1} c_{2i-j+1} f_j, \qquad i = 0, 1, \ldots, \frac{N}{2} - 1$$

and the other half is produced by the "high-pass" filter function

$$b_i = \frac{1}{2}\sum_{j=0}^{N-1} (-1)^j c_{j-2i} f_j, \qquad i = 0, 1, \ldots, \frac{N}{2} - 1$$

where N is the input block size, c_j are the coefficients, f is the input function, and a and b are the output functions. In the case of the lattice filter, the low- and high-pass outputs are usually referred to as the odd and even outputs, respectively. In many situations, the odd or low-pass output contains most of the information content of the original input signal. The even, or high-pass output contains the difference between the true input and the value of the reconstructed input if it were to be reconstructed from only the information given in the odd output. In general, higher order wavelets (i.e. those with more nonzero coefficients) tend to put more information into the odd output, and less into the even output. If the average amplitude of the even output is low enough, then the even half of the signal may be discarded without greatly affecting the quality of the reconstructed signal. The Haar wavelet represents a simple interpolation scheme. After passing these data through the filter functions, the output of the low-pass filter consists of the average of every two samples, and the output of the high-pass filter consists of the difference of every two samples. The high-pass filter contains less information than the low pass output. If the signal is reconstructed by an inverse low-pass filter of the form

$$f_j^L = \sum_{i=0}^{N/2-1} c_{2i-j+1} a_i, \qquad j = 0, 1, \ldots, N-1$$

then the result is a duplication of each entry from the low-pass filter output. This is a wavelet reconstruction with $2\times$ data compression. Since the

perfect reconstruction is a sum of the inverse low-pass and inverse high-pass filters, the output of the inverse high-pass filter can be calculated. This is the result of the inverse high-pass filter function

$$f_j^H = \sum_{i=0}^{N/2-1} (-1)^j c_{j-1-2i} b_i, \qquad j = 0, 1, \ldots, N-1.$$

The perfectly reconstructed signal is $f = f^L + f^H$, where each f is the vector with elements f_j. Using other coefficients and other orders of wavelets yields similar results, except that the outputs are not exactly averages and differences, as in the case using the Haar coefficients. Write a C++ program that implements the pyramid algorithm.

Solution 7. The C++ program implements the Haar wavelet transform. We first find the coefficients `a[i]` and `b[i]`. Then we obtain `fL[i]` and `fH[i]`.

```
// wavelet.cpp

#include <iostream>
#include <cmath>
using namespace std;

int main(void)
{
  const double pi = 3.14159;
  int N = 16;  // N must be a power of 2
  double* f = new double[N];
  int k;
  for(k=0;k<N;k++) f[k] = sin(2.0*pi*(k+1)/N); // input signal
  // coefficients Haar wavelet
  double* c = new double[N];
  for(k=0;k<N;k++) c[k] = 0.0;
  c[0] = 1.0; c[1] = 1.0;
  // array a
  double* a = new double[N/2];
  for(k=0;k<N/2;k++) a[k] = 0.0;
  // array b
  double* b = new double[N/2];
  for(k=0;k<N/2;k++) b[k] = 0.0;
  int i, j;
  for(i=0;i<N/2;i++) {
    for(j=0;j<N;j++)  {
     if(2*i-j+1 < 0) a[i] += c[2*i-j+1+N]*f[j];
```

```
    else a[i] += c[2*i-j+1]*f[j];
  }
  a[i] = 0.5*a[i];
  }
  for(i=0;i<N/2;i++) {
    for(j=0;j<N;j++)  {
     if(j-2*i < 0) b[i] += pow(-1.0,j)*c[j-2*i+N]*f[j];
     else b[i] += pow(-1.0,j)*c[j-2*i]*f[j];
  }
  b[i] = 0.5*b[i];
  }
  for(k=0;k<N/2;k++) cout << "a[" << k << "]=" << a[k] << endl;
  for(k=0;k<N/2;k++) cout << "b[" << k << "]=" << b[k] << endl;
  // inverse
  double* fL = new double[N]; double* fH = new double[N];
  for(j=0;j<N;j++) fL[j] = 0.0;
  for(j=0;j<N;j++) fH[j] = 0.0;
  for(j=0;j<N;j++) {
    for(i=0;i<N/2;i++) {
     if(2*i-j+1 < 0) fL[j] += c[2*i-j+1+N]*a[i];
     else fL[j] += c[2*i-j+1]*a[i];
  }
  }

  for(k=0;k<N;k++)
  cout << "fL[" << k << "] = " << fL[k] << endl;
  for(j=0;j<N;j++) {
    for(i=0;i<N/2;i++) {
    if(j-1-2*i < 0) fH[j] += pow(-1.0,j)*c[j-1-2*i+N]*b[i];
    else fH[j] += pow(-1.0,j)*c[j-1-2*i]*b[i];
  }
  }
  for(k=0;k<N;k++)
  cout << "fH[" << k << "] = " << fH[k] << endl;
  // input signal reconstructed
  double* g = new double[N];
  for(k=0;k<N;k++) g[k] = fL[k] + fH[k];
  for(k=0;k<N;k++)
  cout << "g[" << k << "] = " << g[k] << endl;
  delete[] f; delete[] c; delete[] a; delete[] b;
  delete[] fL; delete[] fH; delete[] g;
  return 0;
}
```

Chapter 14

Graphs

Problem 1. A simple *graph* $G = (V, E)$ consists of V, a nonempty set of vertices (nodes), and E, a set of unordered pairs of distinct elements of V called edges (arcs). This definition of a graph does not permit multiple edges connecting a pair of vertices. It also does not permit a loop, an edge that connects a vertex to itself. Objects with such edges are called *multigraphs.* We consider multigraphs without loops. The degree of a vertex v is the number of edges incident with it. The degree of v is denoted with $d(v)$. Suppose a graph has m nodes, numbered $n_1, n_2, \ldots, n_m$. We can form an $m \times m$ matrix where entry i, j is the number of edges (arcs) between vertices n_i and n_j. This matrix is called the *adjacency matrix* A of the graph. Thus $a_{ij} = p$ where there are p edges between n_i and n_j. A path in a multigraph is a sequence of alternating vertices and edges, such that all edges are distinct. A path can cross a vertex more than once, but can only cross an edge once. An *Euler path* in a graph is a path such that every edge is traversed exactly once. An Euler path exists in a connected graph if and only if $d(v)$ is even for every vertex v of G or two vertices have $d(v)$ odd and all others are even. In the first case the path can begin at any vertex and will end there. In the second case the path must begin at one of the vertex with $d(v)$ odd and will end at the other vertex with $d(v)$ odd. Thus if a graph has more than two vertices of $d(v)$ even, then it cannot have an Euler path.

Given the adjacency matrix

$$A = \begin{pmatrix} 0 & 2 & 1 & 0 & 0 \\ 2 & 0 & 2 & 0 & 0 \\ 1 & 2 & 0 & 2 & 2 \\ 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 2 & 0 \end{pmatrix}$$

write a C++ program that tests for these three cases.

Solution 1. The matrix A is symmetric.

```
// EulerPath.cpp

#include <iostream>
using namespace std;

int main(void)
{
  const int n = 5;
  int A[n][n];
  A[0][0]=0; A[0][1]=2; A[0][2]=1; A[0][3]=0; A[0][4]=0;
  A[1][0]=2; A[1][1]=0; A[1][2]=2; A[1][3]=0; A[1][4]=0;
  A[2][0]=1; A[2][1]=2; A[2][2]=0; A[2][3]=2; A[2][4]=2;
  A[3][0]=0; A[3][1]=0; A[3][2]=2; A[3][3]=0; A[3][4]=2;
  A[4][0]=0; A[4][1]=0; A[4][2]=2; A[4][3]=2; A[4][4]=0;

  int odd = 0, i = 0;

  while(odd <= 2 && i < n)
  {
  int d = 0;
  for(int j=0;j<n;j++) { d += A[i][j]; }
  if(d%2 == 1) odd++;
  i++;
  } // end while
  if(odd > 2) cout << "no Euler path exists";
  if(odd == 0)
  cout << "Euler path exists. Path ends at the starting vertex";
  if(odd == 2 || odd == 1)
  cout << "Euler path exists << endl;
  cout << "Path does not end at the starting vertex";
  return 0;
}
```

Problem 2. Given a network, the all-to-all shortest path problem is to find the total cost of going from every node to every other node in the network. For some pairs of nodes, this can be found by inspecting the network. A large network may consist of hundreds of arcs so that finding the shortest path between various nodes may be complicated. A method to solve this problem is the *Floyd-Warshall algorithm.* It uses a two-dimensional array to represent the cost of travelling from every node to every other node. Each row of the two-dimensional array corresponds to an originating node, and each column corresponds to a destination node.

The Floyd-Warshall algorithm begins by initializing the entries in the cost array with the costs of the arcs in the network. Let n be the number of nodes. The initialization is as follows:

1) Initialize the diagonal elements of the two-dimensional array C to zero, i.e.,

$$c_{0,0} = c_{1,1} = \cdots = c_{n-1,n-1} = 0\,.$$

This reflects the fact that travel from one node to itself is costless.

2) Each arc in the network connects some node i and some node j. For each arc, initialize the value c_{ij} in the cost matrix C to the cost of that arc.

3) All other entries in the array should be set to some large value that exceeds the total cost of any final path in the network (for example we set it to be 20000).

The Floyd-Warshall algorithm then examines every pair of nodes ij $(i \neq j)$, and for each node pair, tests whether the current cost of going from i to j can be reduced by using node k as an intermediate node. This corresponds to finding the following minimum

$$\min(c_{i,j}, c_{i,k} + c_{k,j})\,.$$

Once this is done for all i, j pairs, the updated matrix C now holds the cost of the shortest paths that use only node k as a possible intermediate node. This should be done for all $k \neq i$, $k \neq j$.

As an example consider the matrix

$$C = \begin{pmatrix} 0 & 5 & 3 & 20000 & 1 \\ 2 & 0 & 20000 & 2 & 1 \\ 6 & 20000 & 0 & 2 & 20000 \\ 20000 & 1 & 1 & 0 & 8 \\ 1 & 3 & 20000 & 1 & 0 \end{pmatrix}.$$

Solution 2. The resulting matrix is given by

$$C_F = \begin{pmatrix} 0 & 3 & 3 & 2 & 1 \\ 2 & 0 & 3 & 2 & 1 \\ 5 & 3 & 0 & 2 & 3 \\ 3 & 1 & 1 & 0 & 2 \\ 1 & 2 & 2 & 1 & 0 \end{pmatrix}.$$

```
// floyd.cpp

#include <iostream>
#include <cstdio>
#include <climits>
using namespace std;

void floyd_warshall(int**,int);
int minint(int,int);
void show_C(int**,int);

int main(void)
{
  int i, j, nodes;
  cout << "number of nodes: ";
  cin >> nodes;
  int** C;
  C = new int *[nodes];
  for(i=0;i<nodes;i++) { C[i] = new int [nodes]; }

  for(i=0;i<nodes;i++)
   for(j=0;j<nodes;j++)
    C[i][j] = 0;

  for(i=0;i<nodes;i++)
  for(j=0;j<nodes;j++)
  if(i!=j)
  {
  cout << "C[" << i << "][" << j << "] = ";
  cin >> C[i][j];
  }
  floyd_warshall(C,nodes);
  show_C(C,nodes);
  return 0;
}

void floyd_warshall(int **m,int nodes)
```

```
{
  for(int k=0;k<nodes;k++)
  for(int i=0;i<nodes;i++)
  for(int j=0;j<nodes;j++)
  if(i!=j) m[i][j] = minint(m[i][j], m[i][k]+m[k][j]);
}

int minint(int x,int y)
{
  if(x < y) return x;
  return y;
}

void show_C(int **m,int nodes)
{
  cout << "Cost matrix is: " << endl;
  for(int i=0;i<nodes;i++)
  {
  for(int j=0;j<nodes;j++) cout << m[i][j] << " ";
  cout << endl;
  }
}
```

Problem 3. A *digraph* is a graph where each edge (arc) only connects vertices in one direction, i.e., a directed graph. Consider a digraph D in which each arc has a given (positive) traversal cost, i.e., for each $arc(i,j)$ in D let W_{ij} be the cost of travelling from point i to point j along $arc(i,j)$, where $i,j = 0,1,\ldots,n-1$. Thus $W = (W_{ij})$ is an $n \times n$ matrix. For example let

$$W_{jk} = \begin{pmatrix} 0 & 28 & 2 & \infty & 1 & \infty & \infty & \infty \\ \infty & 0 & \infty & 9 & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty & 24 & \infty & 27 \\ \infty & \infty & \infty & 0 & \infty & \infty & 8 & 7 \\ \infty & 8 & \infty & \infty & 0 & 26 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 0 & 8 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 0 & 7 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 \end{pmatrix}.$$

The cost of a path in a digraph is defined to be the sum of the costs of the arcs of the path. In the *shortest path problem*, one finds the path of least cost which joins one given point to another given point. The cost W_{ij} of a point pair (edge) i,j for which there is no $arc(i,j)$ is set equal to a prohibitively large number. The *Dijkstra method* is as follows. Given

and a source-sink pair of points. We want to find the shortest path from the source to the sink. At each iteration the Dijkstra method identifies a new point (vertex) which is the closest to the source among all those points which are currently not yet identified. The length of the path from the source to this point is calculated and associated with the point. The method builds up a series of shortest paths from the source to successive points until the sink is included in this set. Then the problem is solved. To find the arcs making up the shortest path one uses a backtracking process. The method can be continued until all points have been identified if we want to find shortest paths from the origin to all other points.
(i) Use Dijkstra's method to solve the shortest path problem for the digraph given above, where the source point is p_0.
(ii) Write a C++ program that implements the Dijkstra method.

Solution 3. (i) We partition the set of points (vertexes)

$$\{ p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7 \}$$

into the sets A and B: A containing the origin p_0 and B containing all other points. Some of the points are labèled, a label of $d[i]$ for point p_i representing the shortest distance from the source to point p_i. First the origin is assigned a label $d[0] = 0$. Next we find the point in B which is closest to the origin, i.e.,

$$\min_{i \in A, j \in B} \{ d[i] + W_{ij} \} .$$

We find

$$d[0] + W_{04} = 0 + 1 = 1 .$$

Then the point j in B found is removed from B and placed in A. Thus

$$A = \{ p_0, p_4 \}, \qquad B = \{ p_1, p_2, p_3, p_5, p_6, p_7 \}$$

and $d[4] = 1$. This sequence of steps is repeated until the sink (p_7) is transferred from B to A. We find

$$\min_{i \in A, j \in B} (d[i] + W_{ij}) = d[0] + W_{02} = 0 + 2 = 2$$

Thus

$$A = \{ p_0, p_4, p_2, \}, \quad B = \{ p_1, p_3, p_5, p_6, p_7 \}$$

and $d[2] = 2$. Next we find

$$\min_{i \in A, j \in B} (d[i] + W_{ij}) = d[4] + W_{41} = 1 + 8 = 9 .$$

Thus

$$A = \{ p_0, p_4, p_2, p_1 \}, \quad B = \{ p_1, p_3, p_5, p_6, p_7 \}$$

and $d[1] = 9$. Next we find

$$\min_{i \in A, j \in B}(d[i] + W_{ij}) = d[2] + W_{13} = 9 + 9 = 18\,.$$

Thus

$$A = \{\, p_0, p_4, p_1, p_2 \,\}, \quad B = \{\, p_5, p_6, p_7 \,\}$$

and $d[3] = 18$. Finally we find

$$\min_{i \in A, j \in B}(d[i] + W_{ij}) = d[3] + W_{37} = 18 + 7 = 25\,.$$

Thus

$$A = \{\, p_0, p_4, p_2, p_1, p_3, p_7 \,\}, \quad B = \{\, p_5, p_6 \,\}$$

and $d[7] = 25$. To find the sequence arcs for the shortest path from source to sink we backtrack through the digraph as follows. One forms a list of values of the form

$$d[j] - W_{ij} - d[i]$$

where p_j is the sink and p_i are labeled points connected directly to p_j. Thus

$$d[7] - W_{27} - d[2] = 25 - 27 - 2 \neq 0\,.$$

Hence $arc(p_2, p_7)$ is not on the shortest path. However

$$d[7] - W_{37} - d[3] = 25 - 7 - 18 = 0\,.$$

Thus $arc(p_3, p_7)$ is in the shortest path. Next we replace p_7 by p_3, the point just found to be on the shortest path. A new list of values is found

$$d[3] - W_{13} - d[1] = 18 - 9 - 9 = 0$$

and so $arc(p_1, p_3)$ is on the shortest path. Next we have

$$d[1] - W_{41} - d[4] = 9 - 8 - 1 = 0$$

and

$$d[4] - W_{04} - d[0] = 1 - 1 - 0 = 0\,.$$

Hence $arc(p_4, p_1)$ and $arc(p_0, p_4)$ are also in the shortest path. Thus we find that the shortest path is

$$(p_0, p_4),\ (p_4, p_1),\ (p_1, p_3),\ (p_3, p_7)$$

with a length of $d(7)$ which is 25.

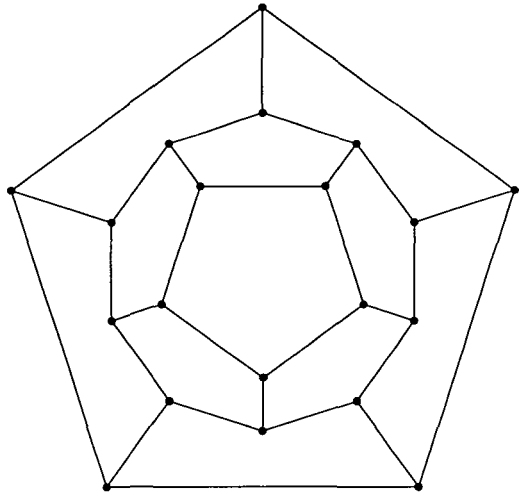
(ii) The function `find()` tests whether an element `x` is in the array `A[n]`.

```
// Dijkstra.cpp

#include <iostream>
using namespace std;

const unsigned long n = 8;

unsigned long find(unsigned long A[n],
                   unsigned long n,unsigned long x)
{
  for(unsigned long k=0;k<n;k++)
  {
  if(x == A[k]) return 1;
  }
  return 0;
}

int main(void)
{
  unsigned long inf = 100000;
  unsigned long i, j, v;
  unsigned long mindist;
  unsigned long A[n];
  unsigned long W[n][n];
  unsigned long dist[n];
  unsigned long path[n];

  for(i=0;i<n;i++) { A[i] = 0; }

  for(i=0;i<n;i++)
   for(j=0;j<n;j++)
     W[i][j] = inf;

  for(i=0;i<n;i++) { W[i][i] = 0; }

  W[0][1] = 28;  W[0][2] = 2;  W[0][4] = 1;
  W[1][3] = 9;
  W[2][5] = 24;  W[2][7] = 27;
  W[3][6] = 8;   W[3][7] = 7;
  W[4][1] = 8;   W[4][5] = 26;
  W[5][6] = 8;
  W[6][7] = 7;

  for(j=0;j<n;j++)
```

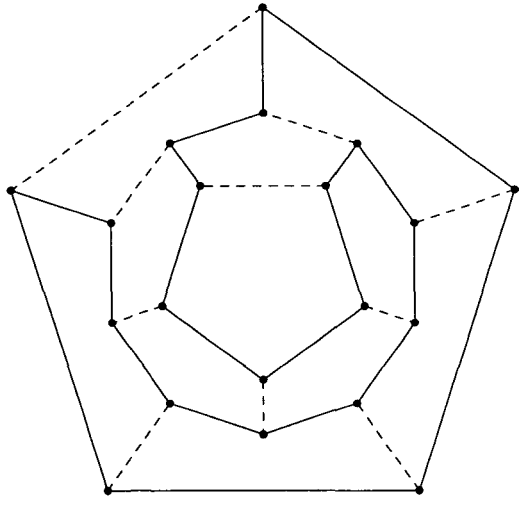
```
  {
  dist[j] = W[0][j];
  if(W[0][j] == inf) { path[j] = 0; }
   else path[j] = 1;
  }

  for(i=0;i<(n-1);i++)
  {
  mindist = inf;
  for(j=1;j<n;j++)
  {
  if((find(A,n,j) == 0) && (dist[j] < mindist))
  {
  mindist = dist[j];
  v = j;
  }
  }
  A[i] = v;

  for(j=1;j<n;j++)
  {
  if((find(A,n,j) == 0) && (dist[v] + W[v][j] < dist[j]))
  {
  dist[j] = dist[v] + W[v][j];
  path[j] = v;
  }
  }
  }

  // output
  for(i=0;i<n;i++)
  cout << "dist[" << i << "] = " << dist[i] << endl;
  for(i=0;i<n;i++)
  cout << "path[" << i << "] = " << path[i] << endl;
  for(i=0;i<n;i++)
  cout << "A[" << i << "] = " << A[i] << endl;
  return 0;
}
```

Problem 4. A *Hamilton path* of a graph (or digraph) is a path that contains all vertices exactly once. If the last vertex in a Hamilton path has the first vertex as a neighbor, then the path can be turned into a Hamilton circuit by joining the last vertex in the path to the first vertex, i.e., a

i.e., a Hamilton path that is also a cycle is a Hamilton circuit. Show that the following graph has a Hamilton circuit.



Solution 4. The solid line gives the Hamilton circuit.



Chapter 15

Neural Networks

Problem 1. Let P and N be two finite sets of points in the Euclidean space $\mathbf{R}^n$ which we want to separate linearly. A weight vector is sought so that the points in P belong to its associated positive half-space and the points in N to the negative half-space. The error of a perceptron with weight vector $\mathbf{w}$ is the number of incorrectly classified points. The learning algorithm must minimize this error function $E(\mathbf{w})$. Now we introduce the *perceptron learning algorithm.* The training set consists of two sets, P and N, in n-dimensional extended input space. We look for a vector $\mathbf{w}$ capable of absolutely separating both sets, so that all vectors in P belong to the open positive half-space and all vectors in N to the open negative half-space of the linear separation.

Algorithm. Perceptron learning

start:	The weight vector $\mathbf{w}(t=0)$ is generated randomly
test:	A vector $\mathbf{x} \in P \cup N$ is selected randomly,
	if $\mathbf{x} \in P$ and $\mathbf{w}(t)^T\mathbf{x} > 0$ goto *test*,
	if $\mathbf{x} \in P$ and $\mathbf{w}(t)^T\mathbf{x} \leq 0$ goto *add*,
	if $\mathbf{x} \in N$ and $\mathbf{w}(t)^T\mathbf{x} < 0$ goto *test*,
	if $\mathbf{x} \in N$ and $\mathbf{w}(t)^T\mathbf{x} \geq 0$ goto *substract*,
add:	set $\mathbf{w}(t+1) = \mathbf{w}(t) + \mathbf{x}$ and $t := t+1$, goto *test*
subtract:	set $\mathbf{w}(t+1) = \mathbf{w}(t) - \mathbf{x}$ and $t := t+1$ goto *test*

This algorithm makes a correction to the weight vector whenever one of the selected vectors in P or N has not been classified correctly. The perceptron convergence theorem guarantees that if the two sets P and N are linearly separable the vector $\mathbf{w}$ is updated only a finite number of times. The routine can be stopped when all vectors are classified correctly.

Consider the sets in the extended space

$$P = \{\,(1.0, 2.0, 2.0),\ (1.0, 1.5, 1.5)\,\}$$

and

$$N = \{\,(1.0, 0.0, 1.0),\ (1.0, 1.0, 0.0),\ (1.0, 0.0, 0.0)\,\}.$$

Thus in $\mathbf{R}^2$ we consider the two sets of points

$$\{\,(2.0, 2.0),\ (1.5, 1.5)\,\}, \qquad \{\,(0.0, 1.0),\ (1.0, 0.0),\ (0.0, 0.0)\,\}.$$

Solution 1. Depending on the random selection of the vector the perceptron learning algorithm could look like

w^T	x^T	$\mathbf{w}^T\mathbf{x}$	set	action
(0 0 0)	(1.0 2.0 2.0)	0.0	P	add
(1 2 2)	(1.0 1.5 1.5)	7.0	P	test
(1 2 2)	(1.0 0.0 1.0)	3.0	N	subtract
(0 2 1)	(1.0 1.0 0.0)	2.0	N	subtract
(-1 1 1)	(1.0 0.0 0.0)	-1.0	N	test
(-1 1 1)	(1.0 2.0 2.0)	3.0	P	test
(-1 1 1)	(1.0 1.5 1.5)	2.0	P	test
(-1 1 1)	(1.0 0.0 1.0)	0.0	N	subtract
(-2 1 0)	(1.0 1.0 0.0)	-1.0	N	test
(-2 1 0)	(1.0 0.0 0.0)	-2.0	N	test
(-2 1 0)	(1.0 2.0 2.0)	0.0	P	add
(-1 3 2)	(1.0 1.5 1.5)	6.5	P	test
(-1 3 2)	(1.0 0.0 1.0)	1.0	N	subtract
(-2 3 1)	(1.0 1.0 0.0)	1.0	N	subtract
(-3 2 1)	(1.0 0.0 0.0)	-3.0	N	test
(-3 2 1)	(1.0 2.0 2.0)	3.0	P	test
(-3 2 1)	(1.0 1.5 1.5)	1.5	P	test
(-3 2 1)	(1.0 0.0 1.0)	-2.0	N	test
(-3 2 1)	(1.0 1.0 0.0)	-1.0	N	test
(-3 2 1)	(1.0 0.0 0.0)	-3.0	N	test

The following C++ program `classify.cpp` implements the algorithm.

```
// classify.cpp

#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
```

```
void classify(double **P,double **N,int p,
              int n,double *w,int d)
{
  int i, j, k = 0, classified = 0;
  double *x, sum;
  srand(time(NULL));
  for(i=0;i<d;i++) w[i] = double(rand())/RAND_MAX;
  while(!classified)
  {
  i = rand()%(p+n-1);
  if(i<p) x = P[i]; else x = N[i-p];
  for(j=0,sum=0.0;j<d;j++) sum += w[j]*x[j];
  if((i<p) && (sum<=0.0))
   for(j=0;j < d;j++) w[j] += x[j];
  if((i>=p) && (sum>=0.0))
   for(j=0;j < d;j++) w[j] -= x[j];
  k++;
  classified = 1;
  // check if the vectors are classified,
  // we expect a minimum of 2 iterations through
  // the p+n elements of the training
  if((k%(2*p+2*n)) == 0)
  {
  for(i=0;(i<p) && classified;i++)
  {
  sum = 0.0;
  for(j=0,sum=0.0;j<d;j++) sum += w[j]*P[i][j];
  if(sum <= 0.0) classified = 0;
  }
  for(i=0;(i<n) && classified;i++)
  {
  sum = 0.0;
  for(j=0,sum=0.0;j<d;j++) sum += w[j]*N[i][j];
  if(sum >= 0.0) classified = 0;
  }
  }
  else classified = 0;
  }
}

int main(void)
{
  double **P = new double*[2];
```

```
  P[0] = new double[3]; P[1] = new double[3];
  P[0][0] = 1.0; P[0][1] = 2.0; P[0][2] = 2.0;
  P[1][0] = 1.0; P[1][1] = 1.5; P[1][2] = 1.5;
  double **N = new double*[3];
  N[0] = new double[3];
  N[1] = new double[3];
  N[2] = new double[3];
  N[0][0] = 1.0; N[0][1] = 0.0; N[0][2] = 1.0;
  N[1][0] = 1.0; N[1][1] = 1.0; N[1][2] = 0.0;
  N[2][0] = 1.0; N[2][1] = 0.0; N[2][2] = 0.0;
  double *w = new double[3];
  classify(P,N,2,3,w,3);
  cout << "w = ( " << w[0] << " , " << w[1]
       << " , " << w[2] << " ) " << endl;

  delete[] P[0]; delete[] P[1];
  delete[] N[0]; delete[] N[1]; delete[] N[2];
  delete[] P; delete[] N;
  delete w;
  return 0;
}
```

Problem 2. For the back-propagation algorithm in neural networks we need the derivative of the functions

$$f_\lambda(x) = \frac{1}{1+e^{-\lambda x}}, \quad \lambda > 0 \tag{1}$$

and

$$g_\lambda(x) = \tanh(\lambda x), \quad \lambda > 0\,. \tag{2}$$

Find the ordinary differential equations for f_λ and g_λ.

Solution 2. The derivative of f_λ is given by

$$\frac{df_\lambda}{dx} = \frac{\lambda e^{-\lambda x}}{(1+e^{-\lambda x})^2}\,.$$

Using (1) we obtain

$$\frac{df_\lambda}{dx} = \lambda f_\lambda(1-f_\lambda)\,.$$

Thus the derivative can be replaced by $\lambda f_\lambda(1-f_\lambda)$. The derivative of g_λ is given by

$$\frac{dg_\lambda}{dx} = \lambda \operatorname{sech}^2(\lambda x) \equiv \lambda \frac{1}{\cosh^2(\lambda x)}\,.$$

Since $\operatorname{sech}^2(\alpha) + \tanh^2(\alpha) \equiv 1$ and using (2)

$$\tanh^2(\lambda x) = g_\lambda^2(x)$$

yields

$$\frac{dg_\lambda}{dx} = \lambda(1 - g_\lambda^2)\,.$$

Thus the derivative can be replaced by $\lambda(1 - g_\lambda^2)$.

Problem 3. In the case of least squares applied to supervised learning with a linear model the function to be minimized is the sum-squared-error

$$S := \sum_{i=0}^{p-1} (\hat{y}_i - f(\mathbf{x}_i))^2$$

where

$$f(\mathbf{x}) = \sum_{j=0}^{m-1} w_j h_j(\mathbf{x})$$

and the free variables are the weights $\{ w_j \}$ for $j = 0, 1, \ldots, m-1$. The given training set, in which there are p pairs (indexed by j running from 0 up to $p-1$), is represented by

$$T := \{ (\mathbf{x}_j, \hat{y}_j) \}_{j=0}^{j=p-1}\,.$$

(i) Find the weights $\hat{w}_j$ from the given training set and the given h_j.
(ii) Apply the formula to the training set

$$\{ (1.0, 1.1), \quad (2.0, 1.8), \quad (3.0, 3.1) \}$$

i.e., $p = 3$, $x_0 = 1.0$, $y_0 = 1.1$, $x_1 = 2.0$, $y_1 = 1.8$, $x_2 = 3.0$, $y_2 = 3.1$. Assume that

$$h_0(x) = 1, \quad h_1(x) = x, \quad h_2(x) = x^2$$

i.e., $m = 3$.

Solution 3. (i) Differentiating S with respect to w_j yields

$$\frac{\partial S}{\partial w_j} = 2 \sum_{i=0}^{p-1} (f(\mathbf{x}_i) - \hat{y}_i) \frac{\partial f}{\partial w_j}(\mathbf{x}_i)\,.$$

Since

$$\frac{\partial f}{\partial w_j}(\mathbf{x}_i) = h_j(\mathbf{x}_i)$$

we find

$$\sum_{i=0}^{p-1} f(\mathbf{x}_i)h_j(\mathbf{x}_i) = \sum_{i=0}^{p-1} \hat{y}_i h_j(\mathbf{x}_i)$$

for $j = 0, 1, \ldots, m-1$. This equation can be written as

$$\mathbf{h}_j^T \mathbf{f} = \mathbf{h}_j^T \hat{\mathbf{y}}$$

where

$$\mathbf{h}_j = \begin{pmatrix} h_j(\mathbf{x}_0) \\ h_j(\mathbf{x}_1) \\ \vdots \\ h_j(\mathbf{x}_{p-1}) \end{pmatrix}, \quad \mathbf{f} = \begin{pmatrix} f(\mathbf{x}_0) \\ f(\mathbf{x}_1) \\ \vdots \\ f(\mathbf{x}_{p-1}) \end{pmatrix}, \quad \hat{\mathbf{y}} = \begin{pmatrix} \hat{y}_0 \\ \hat{y}_1 \\ \vdots \\ \hat{y}_{p-1} \end{pmatrix}, \quad \hat{\mathbf{w}} = \begin{pmatrix} \hat{w}_0 \\ \hat{w}_1 \\ \vdots \\ \hat{w}_{m-1} \end{pmatrix}.$$

We introduce the $p \times m$ matrix (so-called *design matrix*)

$$H = \begin{pmatrix} h_0(\mathbf{x}_0) & h_1(\mathbf{x}_0) & \ldots & h_{m-1}(\mathbf{x}_0) \\ h_0(\mathbf{x}_1) & h_1(\mathbf{x}_1) & \ldots & h_{m-1}(\mathbf{x}_1) \\ \vdots & \vdots & \ddots & \vdots \\ h_0(\mathbf{x}_{p-1}) & h_1(\mathbf{x}_{p-1}) & \ldots & h_{m-1}(\mathbf{x}_{p-1}) \end{pmatrix}.$$

Since

$$\mathbf{f} = H\hat{\mathbf{w}}$$

we finally arrive at

$$\hat{\mathbf{w}} = (H^T H)^{-1} H^T \hat{\mathbf{y}}\,.$$

(ii) For the given case $h_0(x) = 1$, $h_1(x) = x$ and $h_2(x) = x^2$ we find the 3×3 matrix

$$H = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{pmatrix}.$$

Thus

$$H^T = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 4 & 9 \end{pmatrix}$$

and

$$H^T H = \begin{pmatrix} 3 & 6 & 14 \\ 6 & 14 & 36 \\ 14 & 36 & 98 \end{pmatrix}, \qquad (H^T H)^{-1} = \begin{pmatrix} 19 & -21 & 5 \\ -21 & 24.5 & -6 \\ 5 & -6 & 1.5 \end{pmatrix}.$$

Finally

$$\begin{pmatrix} \hat{w}_0 \\ \hat{w}_1 \\ \hat{w}_2 \end{pmatrix} = \begin{pmatrix} 19 & -21 & 5 \\ -21 & 24.5 & -6 \\ 5 & -6 & 1.5 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 4 & 9 \end{pmatrix} \begin{pmatrix} 1.1 \\ 1.8 \\ 3.1 \end{pmatrix} = \begin{pmatrix} 1.0 \\ -0.2 \\ 0.3 \end{pmatrix}.$$

Problem 4. Consider the following 14 capital letters

$$A\ E\ F\ H\ I\ K\ L\ M\ N\ T\ V\ X\ Y\ Z.$$

Each letter can be described by a 7×5 matrix with entries 0 or 1. For example A is represented by

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

and E is represented by

$$E = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

Write a C++ program that finds the Hamming distance of an arbitrary input of a 7×5 matrix with 0's and 1' to the letters A and E. What is the Hamming distance of A and E?

Solution 4. The Hamming distance between E and A is 14.

```
// Hamming.cpp

#include <iostream>
using namespace std;

int hamming(int M1[7][5],int M2[7][5],int rows,int columns)
{
  int d = 0;
  for(int i=0;i<rows;i++)
  for(int j=0;j<columns;j++)
  {
  if(M1[i][j] != M2[i][j]) d++;
  }
  return d;
}
```

```
int main(void)
{
  const int rows = 7;
  const int columns = 5;
  int A[rows][columns];
  A[0][0]=0; A[0][1]=0; A[0][2]=1; A[0][3]=0; A[0][4]=0;
  A[1][0]=0; A[1][1]=1; A[1][2]=0; A[1][3]=1; A[1][4]=0;
  A[2][0]=1; A[2][1]=0; A[2][2]=0; A[2][3]=0; A[2][4]=1;
  A[3][0]=1; A[3][1]=1; A[3][2]=1; A[3][3]=1; A[3][4]=1;
  A[4][0]=1; A[4][1]=0; A[4][2]=0; A[4][3]=0; A[4][4]=1;
  A[5][0]=1; A[5][1]=0; A[5][2]=0; A[5][3]=0; A[5][4]=1;
  A[6][0]=1; A[6][1]=0; A[6][2]=0; A[6][3]=0; A[6][4]=1;

  int E[rows][columns];
  E[0][0]=1; E[0][1]=1; E[0][2]=1; E[0][3]=1; E[0][4]=1;
  E[1][0]=1; E[1][1]=0; E[1][2]=0; E[1][3]=0; E[1][4]=0;
  E[2][0]=1; E[2][1]=0; E[2][2]=0; E[2][3]=0; E[2][4]=0;
  E[3][0]=1; E[3][1]=1; E[3][2]=1; E[3][3]=1; E[3][4]=0;
  E[4][0]=1; E[4][1]=0; E[4][2]=0; E[4][3]=0; E[4][4]=0;
  E[5][0]=1; E[5][1]=0; E[5][2]=0; E[5][3]=0; E[5][4]=0;
  E[6][0]=1; E[6][1]=1; E[6][2]=1; E[6][3]=1; E[6][4]=1;

  int In[rows][columns];
  In[0][0]=1; In[0][1]=0; In[0][2]=1; In[0][3]=1; In[0][4]=1;
  In[1][0]=1; In[1][1]=1; In[1][2]=0; In[1][3]=0; In[1][4]=0;
  In[2][0]=1; In[2][1]=0; In[2][2]=0; In[2][3]=0; In[2][4]=0;
  In[3][0]=1; In[3][1]=1; In[3][2]=1; In[3][3]=1; In[3][4]=1;
  In[4][0]=1; In[4][1]=0; In[4][2]=0; In[4][3]=0; In[4][4]=0;
  In[5][0]=1; In[5][1]=0; In[5][2]=0; In[5][3]=0; In[5][4]=0;
  In[6][0]=1; In[6][1]=1; In[6][2]=1; In[6][3]=1; In[6][4]=0;

  int distAIn = hamming(A,In,rows,columns);
  cout << "distAIn = " << distAIn << endl;
  int distEIn = hamming(E,In,rows,columns);
  cout << "distEIn = " << distEIn << endl;
  int distAE = hamming(A,E,rows,columns);
  cout << "distAE = " << distAE;
  return 0;
}
```

Problem 5. Let the training set of two separate classes be represented by the set of vectors

$$(\mathbf{v}_0, y_0), \quad (\mathbf{v}_1, y_1), \ \ldots \ , (\mathbf{v}_{n-1}, y_{n-1})$$

where $\mathbf{v}_j$ $(j = 0, 1, \ldots, n-1)$ is a vector in the m-dimensional real Hilbert space $\mathbf{R}^m$ and $y_j \in \{-1, +1\}$ indicates the class label. Given a weight vector $\mathbf{w}$ and a bias b, it is assumed that these two classes can be separated by two margins parallel to the hyperplane

$$\mathbf{w}^T\mathbf{v}_j + b \geq 1, \quad \text{for} \quad y_j = +1 \tag{1}$$

$$\mathbf{w}^T\mathbf{v}_j + b \leq -1, \quad \text{for} \quad y_j = -1 \tag{2}$$

for $j = 0, 1, \ldots, n-1$ and $\mathbf{w} = (w_0, w_1, \ldots, w_{m-1})^T$ is a column vector of m-elements. Inequalities (1) and (2) can be combined into a single inequality

$$y_j(\mathbf{w}^T\mathbf{v}_j + b) \geq 1 \quad \text{for } j = 0, 1, \ldots, n-1\,. \tag{3}$$

There exist a number of separate hyperplanes for an identical group of training data. The objective of the *support vector machine* is to determine the optimal weight $\mathbf{w}^*$ and the optimal bias b^* such that the corresponding hyperplane separates the positive and negative training data with maximum margin and it produces the best generation performance. This hyperplane is called an optimal separating hyperplane. The equation for an arbitrary hyperplane is given by

$$\mathbf{w}^T\mathbf{x} + b = 0 \tag{4}$$

and the distance between the two corresponding margins is

$$\gamma(\mathbf{w}, b) = \min_{\{\mathbf{v}\,|\,y=+1\}} \frac{\mathbf{w}^T\mathbf{v}}{\|\mathbf{w}\|} - \max_{\{\mathbf{v}\,|\,y=-1\}} \frac{\mathbf{w}^T\mathbf{v}}{\|\mathbf{w}\|}\,. \tag{5}$$

The optimal separating hyperplane can be obtained by maximizing the above distance or minimizing the norm of $\|\mathbf{w}\|$ under the inequality constraint (3), and

$$\gamma_{max} = \gamma(\mathbf{w}^*, b^*) = \frac{2}{\|\mathbf{w}\|}\,. \tag{6}$$

The saddle point of the Lagrange function

$$L_P(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\mathbf{w}^T\mathbf{w} - \sum_{j=0}^{n-1} \alpha_j(y_j(\mathbf{w}^T\mathbf{v}_j + b) - 1) \tag{7}$$

gives solutions to the minimization problem, where $\alpha_j \geq 0$ are Lagrange multiplier. The solution of this quadratic programming optimization problem requires that the gradient of $L_P(\mathbf{w}, b, \boldsymbol{\alpha})$ with respect to $\mathbf{w}$ and b vanishes, i.e.,

$$\left.\frac{\partial L_P}{\partial \mathbf{w}}\right|_{\mathbf{w}=\mathbf{w}^*} = \mathbf{0}, \qquad \left.\frac{\partial L_P}{\partial b}\right|_{b=b^*} = 0\,.$$

We obtain

$$\mathbf{w}^* = \sum_{j=0}^{n-1} \alpha_j y_j \mathbf{v}_j \tag{8}$$

and

$$\sum_{j=0}^{n-1} \alpha_j y_j = 0\,. \tag{9}$$

Inserting (8) and (9) into (7) yields

$$L_D(\boldsymbol{\alpha}) = \sum_{i=0}^{n-1} \alpha_i - \frac{1}{2} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \alpha_i \alpha_j y_i y_j \mathbf{v}_i^T \mathbf{v}_j \tag{10}$$

under the constraints

$$\sum_{j=0}^{n-1} \alpha_j y_j = 0 \tag{11}$$

and

$$\alpha_j \geq 0, \qquad j = 0, 1, \ldots, n-1\,.$$

The function $L_D(\boldsymbol{\alpha})$ has be maximized. Note that L_P and L_D arise from the same objective function but with different constraints; and the solution is found by minimizing L_P or by maximizing L_D. The points located on the two optimal margins will have nonzero coefficients α_j among the solutions of $\max L_D(\boldsymbol{\alpha})$ and the constraints. These vectors with nonzero coefficients α_j are called support vectors. The bias can be calculated as follows

$$b^* = -\frac{1}{2} \left(\min_{\{\mathbf{v}_j \,|\, y_j = +1\}} \mathbf{w}^{*T} \mathbf{v}_j + \max_{\{\mathbf{v}_j \,|\, y_j = -1\}} \mathbf{w}^{*T} \mathbf{v}_j \right).$$

After determination of the support vectors and bias, the decision function that separates the two classes can be written as

$$f(\mathbf{x}) = \operatorname{sgn} \left(\sum_{j=0}^{n-1} \alpha_j y_j \mathbf{v}_j^T \mathbf{x} + b^* \right).$$

Apply this classification technique to the data set (AND gate)

j	Training set $\mathbf{v}_j$	Target y_j
0	(0,0)	1
1	(0,1)	1
2	(1,0)	1
3	(1,1)	−1

Solution 5. For the present data set we find

$$L_D(\boldsymbol{\alpha}) = \sum_{j=0}^{3} \alpha_j - \frac{1}{2}\alpha_1^2 + \alpha_1\alpha_3 - \frac{1}{2}\alpha_2^2 + \alpha_2\alpha_3 - \alpha_3^2$$

since for the scalar products we have

$$\mathbf{v}_0^T\mathbf{v}_j = 0, \qquad j = 0,1,2,3$$

and

$$\mathbf{v}_1^T\mathbf{v}_1 = 1,\ \mathbf{v}_1^T\mathbf{v}_2 = 0,\ \mathbf{v}_1^T\mathbf{v}_3 = 1,\ \mathbf{v}_2^T\mathbf{v}_2 = 1,\ \mathbf{v}_2^T\mathbf{v}_3 = 1,\ \mathbf{v}_3^T\mathbf{v}_3 = 2\,.$$

The constraints are

$$\alpha_0 \geq 0, \quad \alpha_1 \geq 0, \quad \alpha_2 \geq 0, \quad \alpha_3 \geq 0$$

and

$$\alpha_0 + \alpha_1 + \alpha_2 - \alpha_3 = 0\,.$$

To apply the *Kuhn-Tucker conditions* (which is formulated for a minimum) we have to change $L_D(\boldsymbol{\alpha})$ to $-L_D(\boldsymbol{\alpha})$. Thus we have the Lagrangian

$$\begin{aligned}\widetilde{L}(\boldsymbol{\alpha}) = &-\sum_{j=0}^{3} \alpha_j + \frac{1}{2}\alpha_1^2 - \alpha_1\alpha_3 + \frac{1}{2}\alpha_2^2 - \alpha_2\alpha_3 + \alpha_3^2 \\ &-\mu(\alpha_0 + \alpha_1 + \alpha_2 - \alpha_3) - \lambda_0\alpha_0 - \lambda_1\alpha_1 - \lambda_2\alpha_2 - \lambda_3\alpha_3\,.\end{aligned}$$

Thus we have to solve the sytem of equations

$$\begin{aligned}\frac{\partial\widetilde{L}}{\partial\alpha_0} &= 0 \rightarrow -1 - \mu - \lambda_0 = 0 \\ \frac{\partial\widetilde{L}}{\partial\alpha_1} &= 0 \rightarrow -1 + \alpha_1 - \alpha_3 - \mu - \lambda_1 = 0 \\ \frac{\partial\widetilde{L}}{\partial\alpha_2} &= 0 \rightarrow -1 + \alpha_2 - \alpha_3 - \mu - \lambda_2 = 0 \\ \frac{\partial\widetilde{L}}{\partial\alpha_3} &= 0 \rightarrow -1 - \alpha_1 - \alpha_2 + 2\alpha_3 + \mu - \lambda_3 = 0\end{aligned}$$

together with

$$\lambda_0\alpha_0 = 0, \quad \lambda_1\alpha_1 = 0, \quad \lambda_2\alpha_2 = 0, \quad \lambda_3\alpha_3 = 0$$

$$\lambda_0 \geq 0, \quad \lambda_1 \geq 0, \quad \lambda_2 \geq 0, \quad \lambda_3 \geq 0$$

and the constraints $\alpha_0 + \alpha_1 + \alpha_2 - \alpha_3 = 0$. We find the solution

$$\alpha_0 = 0, \quad \alpha_1 = 2, \quad \alpha_2 = 2, \quad \alpha_3 = 4$$

$$\lambda_0 = 2, \quad \lambda_1 = 0, \quad \lambda_2 = 0, \quad \lambda_3 = 0, \quad \mu = -3\,.$$

Thus

$$\mathbf{w}^* = \sum_{j=0}^{3} \alpha_j y_j \mathbf{v}_j = (-2, -2)\,.$$

For b^* we obtain $b^* = 3$. The decision function that separates the two classes is given by

$$\begin{aligned} f(\mathbf{x}) &= \operatorname{sgn}(\alpha_1 y_1 \mathbf{v}_1^T \mathbf{x} + \alpha_2 y_2 \mathbf{v}_2^T \mathbf{x} + \alpha_3 y_3 \mathbf{v}_3^T \mathbf{x} + b^*) \\ &= \operatorname{sgn}(2x_2 + 2x_1 - 4(x_1 + x_2) + b^*) \\ &= \operatorname{sgn}(-2x_1 - 2x_2 + 3) \\ &= \operatorname{sgn}\left(-x_1 - x_2 + \frac{3}{2}\right)\,. \end{aligned}$$

Thus

$$x_1 + x_2 = \frac{3}{2}\,.$$

This solution can also be seen on inspection of the data set.

Problem 6. In the previous problem we have considered a data set which can be separated by a hyperplane. For nonlinear decision boundaries we can extend the method as follows. The datapoints, $\mathbf{v}_j$ only appear inside a scalar product. We map the datapoints into an alternative higher dimensional space, called *feature space*, through

$$\mathbf{v}_i^T \mathbf{v}_j \to \langle \phi(\mathbf{v}_i), \phi(\mathbf{v}_j) \rangle$$

where $\langle\,,\,\rangle$ denotes the scalar product in the feature space. The map $\phi(\mathbf{v}_i)$ does not need to be known since it is implicitly defined by the choice of the positive definite kernel

$$K(\mathbf{v}_i, \mathbf{v}_j) = \langle \phi(\mathbf{v}_i), \phi(\mathbf{v}_j) \rangle\,.$$

It is assumed that $K(\mathbf{v}_i, \mathbf{v}_j) = K(\mathbf{v}_j, \mathbf{v}_i)$. Examples are the radial base function kernel

$$K(\mathbf{v}_i, \mathbf{v}_j) = \exp(-\|\mathbf{v}_i - \mathbf{v}_j\|^2/(2\sigma^2))$$

and the polynomial kernel

$$K(\mathbf{v}_i, \mathbf{v}_j) = (1 + \mathbf{v}_i^T \mathbf{v}_j)^d\,.$$

For binary classification with a given choice of kernel, the learning task therefore involves maximization of the Lagrangian

$$L_D(\boldsymbol{\alpha}) = \sum_{i=0}^{n-1} \alpha_i - \frac{1}{2} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \alpha_i \alpha_j y_i y_j K(\mathbf{v}_i, \mathbf{v}_j)$$

subject to the constraints

$$\sum_{i=0}^{n-1} \alpha_i y_i = 0, \qquad \alpha_i \geq 0, \quad i = 0, 1, \ldots, n-1 .$$

After the optimal values α_i^* have been found the *decision function* is given by

$$f(\mathbf{x}) = \text{sign}\left(\sum_{i=0}^{n-1} \alpha_i^* y_i K(\mathbf{x}, \mathbf{v}_i) + b \right) .$$

The bias b is found from the primal constraints

$$b = -\frac{1}{2}\left(\max_{\{i:y_i=-1\}} \left(\sum_{j=0}^{n-1} \alpha_j y_j K(\mathbf{v}_i, \mathbf{v}_j) \right) + \min_{\{i:y_i=+1\}} \left(\sum_{j=0}^{n-1} \alpha_j y_j K(\mathbf{v}_i, \mathbf{v}_j) \right) \right) .$$

(i) Find the Kuhn-Tucker conditions.
(ii) Let $\mathbf{v} = (v_1, v_2)^T$ and a feature map that maps

$$\mathbf{v} \to \phi(\mathbf{v}) = (v_1^2, v_2^2, \sqrt{2}v_1 v_2, \sqrt{2}v_1, \sqrt{2}v_2, 1)^T .$$

Find the kernel function $K(\mathbf{v}_i, \mathbf{v}_j)$.
(iii) Find the solutions of the Kuhn–Tucker conditions from (i) for the kernel given in (ii) and the data set

j	Training set $\mathbf{v}_j$	Target y_j
0	$(-1, -1)$	-1
1	$(-1, +1)$	$+1$
2	$(+1, -1)$	$+1$
3	$(+1, +1)$	-1

(iv) Derive a relation between the learning rate η and the kernel using

$$L(\boldsymbol{\alpha}) = \sum_{j=0}^{n-1} \alpha_j - \frac{1}{2} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \alpha_i \alpha_j y_i y_j K(\mathbf{v}_i, \mathbf{v}_j) - \mu \sum_{j=0}^{n-1} \alpha_j y_j .$$

and choose the gradient ascent algorithm

$$\delta\alpha_k = \eta \frac{\partial L}{\partial \alpha_k} = \eta \left(1 - y_k \sum_{j=0}^{n-1} \alpha_j y_j K(\mathbf{v}_j, \mathbf{v}_k) - \mu y_k \right) .$$

(v) The *Kernel-Adatron algorithm* is given by

1. Initialize $\alpha_0 = \alpha_1 = \cdots = \alpha_{n-1} = 1$, $\theta = 0$.

2. For $i = 0, 1, \ldots, n-1$ calculate

$$z_i = \sum_{j=0}^{n-1} \alpha_j y_j K(\mathbf{v}_i, \mathbf{v}_j)$$

3. Calculate $\gamma_i = y_i(z_i - \theta)$.

4. Let $\delta\alpha_i := \eta(1 - \gamma_i)$ be the proposed change to α_i.

   (a) if $\alpha_i + \delta\alpha_i \leq 0$ then $\alpha_i = 0$.

   (b) if $\alpha_i + \delta > 0$ then $\alpha_i = \alpha_i + \delta\alpha_i$.

5. Calculate the new threshold

$$\theta := \frac{1}{2}(\min_i(z_i^+) + \max_i(z_i^-))$$

   where z_i^+ are those patterns i with class label $+1$ and z_i^- those with class label -1.

6. If a maximum number of presentations of the pattern set has been exceeded or the margin

$$m := \frac{1}{2}(\min_i(z_i^+) - \max_i(z_i^-))$$

   has approached 1 then stop, otherwise return to step 2.

Solution 6. (i) For the Kuhn–Tucker conditions (which are formulated for a minimum) we have to change L_D to $-L_D$. Thus taking into account the constraints we have the Lagrangian

$$\widetilde{L}(\boldsymbol{\alpha}) = -\sum_{j=0}^{n-1} \alpha_j + \frac{1}{2}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} \alpha_i\alpha_j y_i y_j K(\mathbf{v}_i, \mathbf{v}_j) - \mu\sum_{j=0}^{n-1} \alpha_j y_j - \sum_{j=0}^{n-1} \lambda_j\alpha_j \,.$$

From $\partial L/\partial\alpha_k = 0$ we find

$$-1 + y_k \sum_{j=0}^{n-1} \alpha_j y_j K(\mathbf{v}_k, \mathbf{v}_j) - \mu y_k - \lambda_k = 0$$

for $k = 0, 1, \ldots, n-1$. The other Kuhn–Tucker conditions are

$$\begin{aligned} \sum_{j=0}^{n-1} \alpha_j y_j &= 0 \\ \alpha_j &\geq 0, \quad j = 0, 1, \ldots, n-1 \\ \lambda_j\alpha_j &= 0, \quad j = 0, 1, \ldots, n-1 \\ \lambda_j &\geq 0, \quad j = 0, 1, \ldots, n-1 \,. \end{aligned}$$

Note that there is no condition on the Lagrange multiplier μ.
(ii) The kernel function for the feature space is

$$K(\mathbf{v}_i, \mathbf{v}_j) = (v_{i1}v_{j1} + v_{i2}v_{j2} + 1)^2 = (1 + (\mathbf{v}_i^T \mathbf{v}_j))^2 .$$

(iii) Inserting the data set into the kernel we obtain a (positive definite) matrix with $K(\mathbf{v}_i, \mathbf{v}_i) = 9$ and $K(\mathbf{v}_i, \mathbf{v}_j) = 1$ for $i \neq j$ with $i, j = 0, 1, 2, 3$. The solution of the Kuhn–Tucker conditions is given by

$$\alpha_0 = \alpha_1 = \alpha_2 = \alpha_3 = \frac{1}{8}$$

and $\lambda_0 = \lambda_1 = \lambda_2 = \lambda_3 = \mu = 0$. Furthermore, we have $b = 0$. Thus

$$f(\mathbf{x}) = \operatorname{sgn}(-x_1 x_2) .$$

(iv) We obtain

$$\begin{aligned}
\Delta L_k &:= L(\alpha_0, \ldots, \alpha_k + \delta\alpha_k, \ldots, \alpha_{n-1}) - L(\alpha_0, \ldots, \alpha_k, \ldots, \alpha_{n-1}) \\
&= \delta\alpha_k \left(1 - y_k \sum_{j=0}^{n-1} \alpha_j y_j K(\mathbf{v}_j, \mathbf{v}_k) - \mu y_k \right) - \frac{1}{2}(\delta\alpha_k)^2 K(\mathbf{v}_k, \mathbf{v_k}) \\
&= \left(\frac{1}{\eta} - \frac{K(\mathbf{v}_k, \mathbf{v}_k)}{2} \right) (\delta\alpha_k)^2 .
\end{aligned}$$

Given that $\Delta L_k > 0$ we find $0 < \eta K(\mathbf{v}_k, \mathbf{v}_k) < 2$ and thus

$$0 < \eta < \frac{2}{K(\mathbf{v}_k, \mathbf{v}_k)} .$$

When L reaches a maximum value and the we have a stable solution, $\delta\alpha_k = 0$. Thus it follows that

$$1 - y_k \sum_{j=0}^{n-1} \alpha_j y_j K(\mathbf{v}_j, \mathbf{v}_k) - \mu y_k = y_k (y_k - \sum_{j=0}^{n-1} \alpha_j y_j K(\mathbf{v}_j, \mathbf{v}_k) - \mu) = 0$$

where we used that $y_k^2 = +1$.
(v) The C++ program could be improved by calculating the kernel beforehand.

```
// kerneladatron.cpp

#include <iostream>
#include <cmath>
#include <vector>
using namespace std;
```

```
double K(double vi[2], double vj[2]) // kernel
{
  double k1=1+vi[0]*vj[0]+vi[1]*vj[1];
  return k1*k1;
}

int main(void)
{
  const int m = 4;
  double v[m][2] = { {-1,-1}, {-1,+1}, {+1,-1}, {+1,+1} };
  double y[m]    = {      +1,      -1,      -1,      +1 };
  double alpha[m];
  double eta=0.01, eps=0.00001, margin=0.0, theta=0.0;
  double min, max;
  int i, j, mininit, maxinit;
  for(i=0;i<m;i++) alpha[i]=1.0;
  while(fabs(margin-1.0)>eps)
  {
  mininit=maxinit=1;
  for(i=0;i<m;i++)
  {
   double z = 0.0;
   for(j=0;j<m;j++) z += alpha[j]*y[j]*K(v[i],v[j]);
   double delta=eta*(1.0-y[i]*(z-theta));
   if(alpha[i]+delta<=0.0) alpha[i] = 0.0;
   else                    alpha[i] += delta;
   if((mininit || z<min) && y[i]>0) { min=z; mininit=0; }
   if((maxinit || z>max) && y[i]<0) { max=z; maxinit=0; }
  }
  margin=(min-max)/2.0; theta=(min+max)/2.0;
  }
  for(i=0;i<m;i++)
  cout << "alpha[" << i << "] = " << alpha[i] << endl;
  cout << "theta = " << theta << endl;
  return 0;
}
```

Chapter 16

Genetic Algorithms

Problem 1. Consider the function

$$f(x_1, x_2) = \frac{x_1^2 + x_2^2}{x_1^2 + \epsilon} + \frac{x_1^2 + x_2^2}{x_2^2 + \epsilon}$$

where ϵ is a small positive constant. What is the problem finding the mimimum of this function using genetic algorithms? Compare with the gradient descent method.

Solution 1. The minimum of this function is at $(0, 0)$. Any gradient method can find it immediately. It is not possible to approach the point $(0, 0)$ following the axes. This means that only correlated mutations are favorable, so that the origin is reached through the diagonal valleys of the function. Functions which "hide" the optimum from genetic algorithms have been called *deceptive functions.*

Problem 2. Consider the polynomial

$$p(x) = x^4 - 7x^3 + 8x^2 + 2x - 1\,.$$

The *zeros* are given by the solution of the equation

$$p(x^*) = 0\,.$$

(i) Apply genetic algorithms to search for zeros in the range $[0, 8]$. What are fitness functions for this problem?

(ii) Write a C++ program that implements the algorithm. Use mutation and crossover as genetic operations.

Solution 2. (i) As fitness function we can use

$$f(x) = -p(x) \cdot p(x)$$

which we have to maximize, i.e., the zeros of the polynomial p are found where f takes a global maximum. Obviously the global maximum of f is 0. Another possible fitness function would be $f(x) = |p(x)|$. This fitness function has to be minimized.
(ii) For faster calculation of f we use Horner's scheme. The elements in the farm is 24 and the length of the bitstring is 20. The number of iterations is 1000. The output provides the two zeros 5.6385 and 1.21558. The two other roots of p have an imaginary part.

```
// genetic.cpp

#include <iostream>
#include <cstdlib>   // srand(), rand()
#include <ctime>     // time()
#include <cmath>     // pow
using namespace std;

// fitness function where maximum to be found
double f(double x)
{
  double temp = -1.0 + x*(2.0 + x*(8.0 + x*(-7.0 + x)));
  return -temp*temp;
}

// fitness function value for individual
double f_value(double (*func)(double),int* arr,int& N,
       double a,double b)
{
  double res;
  double m = 0.0;
  for(int j=0;j<N;j++)
  {
  double k = j;
  m += arr[N-j-1]*pow(2.0,k);
  }
  double x = a + m*(b-a)/(pow(2.0,N)-1.0);
  res = func(x);
  return res;
```

```
}

// x_value at global maximum
double x_value(int* arr,int& N,double a,double b)
{
  double m = 0.0;
  for(int j=0;j<N;j++)
  {
  double k = j;
  m += arr[N-j-1]*pow(2.0,k);
  }
  double x = a + m*(b-a)/(pow(2.0,N)-1.0);
  return x;
}

// setup the population (farm)
void setup(int** farm,int M,int N)
{
  srand((unsigned) time(NULL));
  for(int j=0;j<M;j++)
    for(int k=0;k<N;k++)
      farm[j][k] = rand()%2;
}

// cross two individuals
void crossings(int** farm,int& M,int& N,double& a,double& b)
{
  int K = 2;
  int** temp = new int* [K];
  for(int i=0;i<K;i++) temp[i] = new int[N];

  double res[4];
  int r1 = rand()%M;  int r2 = rand()%M;
  // random returns a value between
  // 0 and one less than its parameter
  while(r2 == r1) r2 = rand()%M;

  res[0] = f_value(f,farm[r1],N,a,b);
  res[1] = f_value(f,farm[r2],N,a,b);
  for(int j=0;j<N;j++)
  {
  temp[0][j] = farm[r1][j]; temp[1][j] = farm[r2][j];
  }
  int r3 = rand()%(N-2) + 1;
```

```
  for(j=r3;j<N;j++)
  {
  temp[0][j] = farm[r2][j]; temp[1][j] = farm[r1][j];
  }
  res[2] = f_value(f,temp[0],N,a,b);
  res[3] = f_value(f,temp[1],N,a,b);

  if(res[2] > res[0])
  {
  for(j=0;j<N;j++)
  farm[r1][j] = temp[0][j];
  res[0] = res[2];
  }

  if(res[3] > res[1])
  {
  for(j=0;j<N;j++)
  farm[r2][j] = temp[1][j];
  res[1] = res[3];
  }
  for(j=0;j<K;j++) delete[] temp[j];
  delete[] temp;
}

// mutate an individual
void mutate(int** farm,int& M,int& N,double& a,double& b)
{
  double res[2];
  int r4 = rand()%N; int r1 = rand()%M;
  res[0] = f_value(f,farm[r1],N,a,b);
  int v1 = farm[r1][r4];
  if(v1 == 0) farm[r1][r4] = 1;
  if(v1 == 1) farm[r1][r4] = 0;
  double a1 = f_value(f,farm[r1],N,a,b);
  if(a1 < res[0]) farm[r1][r4] = v1;
  int r5 = rand()%N; int r2 = rand()%M;
  res[1] = f_value(f,farm[r2],N,a,b);
  int v2 = farm[r2][r5];
  if(v2 == 0) farm[r2][r5] = 1;
  if(v2 == 1) farm[r2][r5] = 0;
  double a2 = f_value(f,farm[r2],N,a,b);
  if(a2 < res[1]) farm[r2][r5] = v2;
}
```

```
int main(void)
{
  int M = 24;     // population (farm) has 24 individuals
  int N = 20;     // length of binary string
  int** farm = NULL;   // allocate memory for population
  farm = new int* [M];
  for(int i=0;i<M;i++) farm[i] = new int[N];
  setup(farm,M,N);
  double a = 0.0;  double b = 8.0;  // interval [a,b]
  for(int k=0;k<1000;k++)
  {
  crossings(farm,M,N,a,b);
  mutate(farm,M,N,a,b);
  }  // end for loop
  int j;
  for(j=0;j<M;j++)
  cout << "fitness f_value[" << j << "] = "
       << f_value(f,farm[j],N,a,b)
       << "  " << "x_value[" << j << "] = "
       << x_value(farm[j],N,a,b) << endl;
  for(j=0;j<M;j++) delete[] farm[j];
  delete[] farm;
  return 0;
}
```

Problem 3. Consider the nonlinear second-order ordinary differential equation

$$(u-x)\frac{d^2u}{dx^2} + \sin^2(x) = 0$$

with the boundary value problem

$$u(0) = 0, \qquad u(1) = 1 + \sin(1), \qquad x \in [0,1]\,.$$

As an ansatz for the solution we use the polynomial

$$u(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + c_4 x^4\,.$$

(i) Why can we set $c_0 = 0$? Use the second boundary condition to eliminate one more coefficient.
(ii) Explain why we can define a fitness function as

$$f(c_2, c_3, c_4) = -\sum_{j=0}^{1/h} \left((u(j\cdot h) - j\cdot h)\frac{d^2u(j\cdot h)}{dx^2} + \sin^2(j\cdot h) \right)^2$$

where h is the step length (for example $h = 0.1$).

Solution 3. (i) Since $u(0) = 0$ we find $c_0 = 0$. From the second boundary condition we find

$$u(1) = 1 + \sin(1) = c_1 + c_2 + c_3 + c_4 .$$

Thus

$$c_1 = 1 + \sin(1) - c_2 - c_3 - c_4 .$$

(ii) For the implementation of the second order derivative, we first differentiate u, i.e.,

$$\frac{d^2u}{dx^2} = 2c_2 + 6c_3x + 12c_4x^2$$

and then replace x by $j \cdot h$. Inserting $u(j \cdot h)$ for $u(x)$, $j \cdot h$ for x and d^2u/dx^2 into the differential equation yields the fintness function f.

Problem 4. Consider the symmetric 10×10 matrix

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \end{pmatrix}$$

and the linear equation

$$A\mathbf{x} = \mathbf{r}$$

where

$$\mathbf{r} = (1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1)^T .$$

Here ^{T} denotes transpose. Find a fitness function so that genetic algorithms can be used to solve this linear equation.

Solution 4. A possible fitness function is

$$f(\mathbf{x}) = -\left(\frac{1}{10} \sum_{i=0}^{9} \left| \sum_{j=0}^{9} a_{ij}x_j - r_i \right| \right)$$

where a_{ij} are the matrix elements of A. This function has to be maximized.

Problem 5. Let A be a given $m \times m$ symmetric positive-semidefinite matrix over $\mathbf{R}$. Let $\mathbf{b} \in \mathbf{R}^m$, where $\mathbf{b}$ is a given column vector. Consider the quadratic functional

$$E(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{x}^T\mathbf{b}$$

where ^{T} denotes transpose. The minimum $\mathbf{x}^*$ of $E(\mathbf{x})$ over $\mathbf{R}^m$ is unique and occurs where the gradient of $E(\mathbf{x})$ vanishes, i.e.,

$$\nabla E(\mathbf{x} = \mathbf{x}^*) = A\mathbf{x} - \mathbf{b} = \mathbf{0}\,.$$

The quadratic minimization problem is thus equivalent to solving the system of linear equations $A\mathbf{x} = \mathbf{b}$. Let

$$A = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Use E as a fitness function to solve the system of linear equations applying genetic algorithms.

Solution 5. We have to find the minima of the function

$$\begin{aligned} E(\mathbf{x}) &= \frac{1}{2}(x_1, x_2)\begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} - (x_1, x_2)\begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ &= x_1^2 + x_1x_2 + \frac{1}{2}x_2^2 - x_1 - x_2\,. \end{aligned}$$

The solution is $x_1 = 0$, $x_2 = 1$.

Problem 6. A map is called n-colorable if each region of the map can be assigned a color from n different colors such that no two adjacent regions have the same color. The four color conjecture is that every map is 4-colorable. In 1976 Appel and Haken proved the four color conjecture with extensive use of computer calculations. We can describe the m regions of a map using a $m \times m$ *adjacency matrix* A where $A_{ij} = 1$ if region i is adjacent to region j and $A_{ij} = 0$ otherwise. We set $A_{ii} = 0$. For the fitness function we can determine the number of adjacent regions which have the same color. The lower the number, the fitter the individual. Individuals are represented as strings of characters, where each character represents the color for the region corresponding to the characters position in the string. Write a Java program that uses genetic algorithm to find a solution of the four color problem given the adjacency matrix.

Solution 6. The data member `population` is the number of individuals in the population, and `mu` is the probability that an individual is mutated.

The method `fitness()` evaluates the fitness of a string using the adjacency matrix to determine when adjacent regions have the same color. If the fitness is equal to 0 we have found a solution. The adjacency matrix can be modified to solve for any map. The method `mutate()` determines for each individual in the population whether the individual is mutated, and mutates a component of the individual by randomly changing the color. The method `crossing()` performs the crossing operation. The genetic algorithm is implemented in the method `GA()`. The arguments are an adjacency matrix, a string specifying which colors to use and the number of regions on the map. It returns a string specifying a solution to the problem. One such solution is `YBRBYGYRYB`, where `R` stands for red, `G` for green, `B` for blue and `Y` for yellow.

```
// FourColor.java

public class FourColor
{
  static int population = 1000;
  static double mu = 0.01;

  public static void main(String[] args)
  {
  int[][] adjM = {{0,1,0,1,0,0,0,0,0,0},
                  {1,0,1,0,0,1,0,0,0,0},
                  {0,1,0,0,0,0,1,0,0,0},
                  {1,0,0,0,1,1,0,0,0,0},
                  {0,0,0,1,0,1,0,1,0,0},
                  {0,1,0,1,1,0,1,0,1,1},
                  {0,0,1,0,0,1,0,0,0,1},
                  {0,0,0,0,1,0,0,0,1,0},
                  {0,0,0,0,0,1,0,1,0,1},
                  {0,0,0,0,0,1,1,0,1,0}};

  System.out.println(GA(adjM,"RGBY",10));
  }

  static int fitness(int[][] adjM,String s,int N)
  {
  int count = 0;
  for(int i=0;i<N-1;i++)
  {
  for(int j=i+1;j<N;j++)
  {
  if((s.charAt(i) == s.charAt(j)) && (adjM[i][j] == 1))
```

```
count++;
}
}
return count;
}

static void mutate(String[] p,String colors)
{
int j;
for(int i=0;i<p.length;i++)
{
if(Math.random()<mu)
{
int pos=(int)(Math.random()*(p[i].length()-1));
int mut=(int)(Math.random()*(colors.length()-2));
char[] ca1=p[i].toCharArray();
char[] ca2=colors.toCharArray();
for(j=0;ca1[pos]!=ca2[j];j++) {};
ca1[pos]=ca2[(j+mut)%colors.length()];
p[i]=new String(ca1);
}
}
}

static void crossing(String[] p,int[][] adjM)
{
int p1 = (int)(Math.random()*(p.length-1));
int p2 = p1;
int c1 = (int)(Math.random()*(p[0].length()-1));
int c2 = c1;
while(p2==p1) p2 = (int)(Math.random()*(p.length-1));
while(c2==c1) c2 = (int)(Math.random()*(p[0].length()-1));
if(c2<c1) { int temp = c2; c2 = c1; c1 = temp;}
String[] temp = new String[4];
temp[0]=p[p1]; temp[1]=p[p2];
temp[2]=p[p1].substring(0,c1)+p[p2].substring(c1+1,c2)
       +p[p1].substring(c2+1,p[p1].length()-1);
temp[3]=p[p2].substring(0,c1)+p[p1].substring(c1+1,c2)
       +p[p2].substring(c2+1,p[p2].length()-1);
int i, f;
for(i=0,f=0;i<4;i++)
{
if(fitness(adjM,temp[i],temp[i].length())
  >fitness(adjM,temp[f],temp[f].length()))
```

```
  f = i;
  }
  { String tmp=temp[f]; temp[f]=temp[0]; temp[0]=tmp; }
  for(i=1,f=1;i<4;i++)
  {
  if(fitness(adjM,temp[i],temp[i].length())
    >fitness(adjM,temp[f],temp[f].length()))
  f = i;
  }
  { String tmp=temp[f]; temp[f]=temp[1]; temp[1]=tmp; }
  p[p1] = temp[2]; p[p2] = temp[3];
  }

  static String GA(int[][] adjM,String colors,int N)
  {
  int maxfitness, mfi = 0;
  String[] p = new String[population];
  char[] temp = new char[N];
  for(int i=0;i<population;i++)
  {
  for(int j=0;j<N;j++)
  {
  temp[j] =
    colors.charAt((int)((Math.random()*colors.length())));
  }
  p[i] = new String(temp);
  }
  maxfitness=fitness(adjM,p[0],p[0].length());
  while(maxfitness!=0)
  {
  mutate(p,colors); crossing(p,adjM);
  for(int i=0;i<p.length;i++)
  {
  if(fitness(adjM,p[i],p[i].length())<maxfitness)
  {
  maxfitness=fitness(adjM,p[i],p[i].length());
  mfi = i;
  }
  }
  }
  return p[mfi];
  }
}
```

Chapter 17

Optimization

Problem 1. Calculate the shortest distance (Euclidean) between the curves

$$x^2 + (y-5)^2 = 1$$

and

$$y = x^2.$$

Apply the Lagrange multiplier method.

Solution 1. The square of the distance between two points (x_1, y_1) and (x_2, y_2) on the curves is given by

$$d^2 = (x_1 - x_2)^2 + (y_1 - y_2)^2 .$$

We have two constraints. This means we have two Lagrange multipliers. Thus, we define

$$F(x_1, y_1, x_2, y_2) := (x_1 - x_2)^2 + (y_1 - y_2)^2 + \lambda_1(x_1^2 + (y_1 - 5)^2 - 1) + \lambda_2(y_2 - x_2^2).$$

Thus, we obtain the equations

$$\begin{aligned}
\frac{\partial F}{\partial x_1} &= 2(x_1 - x_2) + 2\lambda_1 x_1 = 0 \\
\frac{\partial F}{\partial x_2} &= -2(x_1 - x_2) - 2\lambda_2 x_2 = 0 \\
\frac{\partial F}{\partial y_1} &= 2(y_1 - y_2) + 2\lambda_1(y_1 - 5) = 0 \\
\frac{\partial F}{\partial y_2} &= -2(y_1 - y_2) + \lambda_2 = 0 .
\end{aligned}$$

Adding the first two equations and the last two equations yields

$$\lambda_1 x_1 = \lambda_2 x_2$$
$$2\lambda_1(y_1 - 5) = -\lambda_2 .$$

Of course we still have

$$(1 + \lambda_1)x_1 = x_2$$
$$(1 + \lambda_1)(y_1 - 5) = y_2 - 5$$

from the first set of equations. If $\lambda_1 = 0$, we have $\lambda_2 = 0$ so that $x_1 = x_2$ and $y_1 = y_2$. Thus, we obtain

$$x_1^2 + (x_1^2 - 5)^2 = 1$$
$$x_1^4 - 9x_1^2 + 24 = 0$$
$$x_1^2 = \frac{9 \pm \sqrt{-15}}{2} .$$

Thus, $\lambda_1 = 0$ does not give a valid solution. Suppose $\lambda_2 = 0$. Once again $x_1 = x_2$ and $y_1 = y_2$, thus $\lambda_2 = 0$ does not give a valid solution. Now, suppose $x_1 = 0$, thus $x_2 = y_2 = 0$ and $y_1 = 6$ or $y_1 = 4$. Lastly, suppose $x_1 \neq 0$. Thus

$$\frac{x_2}{x_1} = \frac{\lambda_1}{\lambda_2} = (1 + \lambda_1)$$
$$\frac{\lambda_1}{\lambda_2}(y_1 - 5) = (1 + \lambda_1)(y_1 - 5) = -\frac{1}{2}$$
$$(1 + \lambda_1)(y_1 - 5) = y_2 - 5$$
$$y_2 = \frac{9}{2} .$$

From $y_2 = x_2^2$ we obtain

$$x_2 = \pm \frac{3}{\sqrt{2}} .$$

Furthermore, we have

$$\frac{1}{1 + \lambda_1} = \frac{x_1}{x_2} = \frac{y_1 - 5}{y_2 - 5} .$$

From $x_1^2 + (y_1 - 5)^2 = 1$ we obtain

$$\frac{2}{9}x_1^2 = 4(y_1 - 5)^2$$
$$x_1 = \pm \frac{6}{\sqrt{38}}$$
$$y_1 = \pm \sqrt{\frac{2}{38}} + 5 .$$

We tabulate the solutions

x_1	y_1	x_2	y_2	Value
0	4	0	0	4
0	6	0	0	6
$\frac{6}{\sqrt{38}}$	$\sqrt{\frac{2}{38}}+5$	$-\frac{3}{\sqrt{2}}$	$\frac{9}{2}$	3.179449
$\frac{6}{\sqrt{38}}$	$-\sqrt{\frac{2}{38}}+5$	$\frac{3}{\sqrt{2}}$	$\frac{9}{2}$	1.179449
$-\frac{6}{\sqrt{38}}$	$\sqrt{\frac{2}{38}}+5$	$\frac{3}{\sqrt{2}}$	$\frac{9}{2}$	3.179449
$-\frac{6}{\sqrt{38}}$	$-\sqrt{\frac{2}{38}}+5$	$-\frac{3}{\sqrt{2}}$	$\frac{9}{2}$	1.179449

The minimum distance is approximately 1.179449.

Problem 2. A firm uses two inputs to produce one output. Its production function is

$$f(x_1, x_2) = x_1^a x_2^b, \qquad a, b > 1\,.$$

The price of the output is p, and the prices of the inputs are w_1 and w_2. The firm is constrained by a law that says it must use exactly the same number of units of both inputs. Use the Lagrange multiplier method to maximize the function

$$g(x_1, x_2) = pf(x_1, x_2) - w_1 x_1 - w_2 x_2$$

subject to $x_2 - x_1 = 0$.

Solution 2. The Lagrange function is given by

$$L(x_1, x_2) = px_1^a x_2^b - w_1 x_1 - w_2 x_2 - \lambda(x_2 - x_1)\,.$$

Thus from

$$\frac{\partial L}{\partial x_1} = 0, \qquad \frac{\partial L}{\partial x_2} = 0$$

we find

$$apx_1^{a-1}x_2^b - w_1 + \lambda = 0, \qquad bpx_1^a x_2^{b-1} - w_2 - \lambda = 0\,.$$

Furthermore, we have the constraint $x_2 = x_1$. These three equations have a single solution

$$x_1^* = x_2^* = \left(\frac{w_1 + w_2}{p(a+b)}\right)^{1/(a+b-1)}$$

and

$$\lambda^* = \frac{bw_1 - aw_2}{a+b}\,.$$

Thus

$$g(x_1^*, x_2^*) = p(x_1^*)^{a+b} - x_1^*(w_1 + w_2)\,.$$

Problem 3. A household has the utility function

$$f(x_1, x_2) = x_1^{\alpha} x_2^{1-\alpha}$$

where $\alpha > 0$ and faces the budget constraint

$$p_1 x_1 + p_2 x_2 \leq m$$

with $m > 0$. Maximize the household's utility using *Kuhn–Tucker conditions* for the demand functions $x_1(p_1, p_2, m)$ and $x_2(p_1, p_2, m)$. Evaluate the demand functions for $(p_1, p_2, m) = (1, 0.5, 10)$ and $\alpha = 0.5$. Find also $f(x_1, x_2)$ for these values.

Solution 3. The Lagrangian is

$$L(x_1, x_2, \lambda) = x_1^{\alpha} x_2^{1-\alpha} + \lambda(m - p_1 x_1 - p_2 x_2)$$

with the Kuhn–Tucker conditions

$$\begin{aligned}
\frac{\partial L}{\partial x_1} &= \frac{\alpha}{x_1} x_1^{\alpha} x_2^{1-\alpha} - \lambda p_1 = 0 \\
\frac{\partial L}{\partial x_2} &= \frac{1-\alpha}{x_2} x_1^{\alpha} x_2^{1-\alpha} - \lambda p_2 = 0 \\
\lambda(m - p_1 x_1 - p_2 x_2) &= 0 \\
\lambda &\geq 0 \\
m - p_1 x_1 - p_2 x_2 &\geq 0\,.
\end{aligned}$$

If $\lambda = 0$ the problem has no solution, since

$$\frac{\alpha}{x_1} x_1^{\alpha} x_2^{1-\alpha} = 0$$

only if $x_2 = 0$, but then

$$\frac{1-\alpha}{x_2} x_1^{\alpha} x_2^{1-\alpha} = \infty\,.$$

Thus $\lambda > 0$. Therefore, we have to solve the system of nonlinear equations

$$\begin{aligned}
\frac{\alpha}{x_1} x_1^{\alpha} x_2^{1-\alpha} - \lambda p_1 &= 0 \\
\frac{1-\alpha}{x_2} x_1^{\alpha} x_2^{1-\alpha} - \lambda p_2 &= 0 \\
m - p_1 x_1 - p_2 x_2 &= 0
\end{aligned}$$

where (m, p_1, p_2, α) are given and the unkowns are (x_1, x_2, λ). The solution is

$$x_1 = \alpha \frac{m}{p_1}$$
$$x_2 = (1-\alpha)\frac{m}{p_2}\,.$$

At the given vector of prices, income, and α, we obtain $x_1 = 5$, $x_2 = 10$ and the utility is

$$f(x_1 = 5, x_2 = 10) = 5^{1/2}10^{1/2}\,.$$

Problem 4. Given the *Lagrange function*

$$L(x, \dot{x}) = \frac{1}{2}\sum_{j=1}^{3}(\dot{x}_j^2 - \omega_j^2 x_j^2) - \frac{1}{2}\lambda(t)\left(\sum_{j=1}^{3} x_j^2 - 1\right) \tag{1}$$

where λ is the time-dependent Lagrange multiplier. The system describes an $n = 3$ dimensional *harmonic oscillator* constrainted to a unit $n - 1 = 2$ sphere, i.e.,

$$\sum_{j=1}^{3} x_j^2 = 1\,. \tag{2}$$

Find the equations of motion using the *Euler-Lagrange equations*

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{x}_j} - \frac{\partial L}{\partial x_j} = 0, \qquad j = 1, 2, 3\,. \tag{3}$$

Solution 4. From (3) and (1) we find

$$\ddot{x}_j + \omega_j^2 x_j + \lambda(t) x_j = 0, \qquad j = 1, 2, 3\,. \tag{4}$$

To eliminate the Lagrange multiplier λ we proceed as follows. From (2) we obtain by differentiating with respect to t

$$\sum_{j=1}^{3} \dot{x}_j x_j = 0$$

and differentiating twice with respect to t yields

$$\sum_{j=1}^{3}(\ddot{x}_j x_j + \dot{x}_j^2) = 0\,. \tag{5}$$

From (4) we obtain $\ddot{x}_j x_j + \omega_j^2 x_j^2 + \lambda(t) x_j^2 = 0$. Summation yields

$$\sum_{j=1}^{3} \ddot{x}_j x_j + \sum_{j=1}^{3} \omega_j^2 x_j^2 + \lambda(t) = 0$$

where we used (2). Thus

$$\lambda(t) = -\sum_{j=1}^{3} \ddot{x}_j x_j - \sum_{j=1}^{3} \omega_j^2 x_j^2 = \sum_{j=1}^{3} \dot{x}_j^2 - \sum_{j=1}^{3} \omega_j^2 x_j^2$$

where we used (5). Inserting λ into (4) yields the equations of motion

$$\ddot{x}_j + \omega_j^2 x_j + x_j \sum_{j=1}^{3} (\dot{x}_j^2 - \omega_j^2 x_j^2) = 0\,.$$

Problem 5. Consider the constraint nonlinear programming problem with inequality and equality constraints: minimize the scalar function f subject to the inequality constraints $g_k(\mathbf{x}) \geq 0$, $k = 1, 2, \ldots, K$ and the equality constraints $h_m(\mathbf{x}) = 0$, $m = 1, 2, \ldots, M$. For this problem we can construct the *Lagrange function*

$$L(\mathbf{x}, \lambda, \mu) = f(\mathbf{x}) - \sum_{k=1}^{K} \lambda_k g_k(\mathbf{x}) - \sum_{m=1}^{M} \mu_m h_m(\mathbf{x})$$

where λ_k and μ_m are the Lagrange multipliers. If the problem has a solution

$$\mathbf{x}^* = (x_1^*, x_2^*, \ldots, x_n^*)$$

i.e., $\min_{\mathbf{x}} f(\mathbf{x}) = f(\mathbf{x}^*)$ and all constraints are satisfied, then the following conditions (called *Kuhn-Tucker conditions*) hold

$$\nabla f(\mathbf{x}^*) - \sum_{k=1}^{K} \lambda_k^* \nabla g_k(\mathbf{x}^*) - \sum_{m=1}^{M} \mu_m^* \nabla h_m(\mathbf{x}^*) = \mathbf{0}$$

and

$$\begin{aligned} g_k(\mathbf{x}^*) &\geq 0 \quad k = 1, 2, \ldots, K \\ h_m(\mathbf{x}^*) &= 0 \quad m = 1, 2, \ldots, M \\ \lambda_k^* g_k(\mathbf{x}^*) &= 0 \quad k = 1, 2, \ldots, K \\ \lambda_k^* &\geq 0 \quad k = 1, 2, \ldots, K\,. \end{aligned}$$

In convex programming problems, the Kuhn-Tucker conditions are necessary and sufficient for a global minimum. Find the minimum of the function

$$f(\mathbf{x}) = (x_1 - 2)^2 + (x_2 - 1)^2$$

under the constraints

$$\begin{aligned} g_1(\mathbf{x}) &= x_2 - x_1^2 \geq 0 \\ g_2(\mathbf{x}) &= 2 - x_1 - x_2 \geq 0 \\ g_3(\mathbf{x}) &= x_1 \geq 0\,. \end{aligned}$$

Solution 5. The Lagrange function is

$$L(\mathbf{x}, \lambda) = (x_1 - 2)^2 + (x_2 - 1)^2 - \lambda_1(x_2 - x_1^2) - \lambda_2(2 - x_1 - x_2) - \lambda_3 x_1\,.$$

Thus we find the Kuhn-Tucker conditions

$$\begin{aligned} 2(x_1 - 2) + 2\lambda_1 x_1 + \lambda_2 - \lambda_3 &= 0 \\ 2(x_2 - 1) - \lambda_1 + \lambda_2 &= 0 \\ x_2 - x_1^2 &\geq 0 \\ 2 - x_1 - x_2 &\geq 0 \\ x_1 &\geq 0 \\ \lambda_1(x_2 - x_1^2) &= 0 \\ \lambda_2(2 - x_1 - x_2) &= 0 \\ \lambda_3 x_1 &= 0 \end{aligned}$$

and

$$\lambda_1 \geq 0, \quad \lambda_2 \geq 0, \quad \lambda_3 \geq 0\,.$$

These equations and inequalities can be solved starting from $\lambda_3 x_1 = 0$ with the cases $\lambda_3^* = 0$ or $x_1^* = 0$. We find

$$x_1^* = 1,\ x_2^* = 1,\ \lambda_1^* = \frac{2}{3},\ \lambda_2^* = \frac{2}{3},\ \lambda_3^* = 0$$

for which $f(x_1^*, x_2^*) = 1$.

Problem 6. A norm of an $n \times n$ matrix over $\mathbf{R}$ is given by

$$\|A\| := \max_{\|\mathbf{x}\|=1} \|A\mathbf{x}\|$$

where $\mathbf{x} \in \mathbf{R}^n$ and $\|A\mathbf{x}\|$ denotes the Euclidean norm. How can the constraint

$$\|\mathbf{x}\| = 1 \Leftrightarrow x_1^2 + x_2^2 + \cdots + x_n^2 = 1$$

be eliminated?

Solution 6. We can introduce n-dimensional *spherical coordinates*

$$\begin{aligned}
x_1 &= r\cos\theta_1 \\
x_2 &= r\sin\theta_1\cos\theta_2 \\
x_3 &= r\sin\theta_1\sin\theta_2\cos\theta_3 \\
x_4 &= r\sin\theta_1\sin\theta_2\sin\theta_3\cos\theta_4 \\
&\vdots = \vdots \\
x_{n-1} &= r\sin\theta_1\sin\theta_2\sin\theta_3\cdots\sin\theta_{n-2}\cos\theta_{n-1} \\
x_n &= r\sin\theta_1\sin\theta_2\sin\theta_3\cdots\sin\theta_{n-2}\sin\theta_{n-1}
\end{aligned}$$

or

$$x_k = r\cos\theta_k \prod_{\ell=1}^{k-1}\sin\theta_\ell, \quad \text{for } k = 1, 2, \ldots, n-1$$

$$x_n = r\prod_{\ell=1}^{n-1}\sin\theta_\ell$$

with $r = 1$, $-\pi \le \theta_1 \le \pi$, and $0 \le \theta_j \le \pi$, $2 \le j \le n-1$. The inverse transform is

$$\theta_k = \arccos\left(\frac{x_k}{\sqrt{r^2 - \sum_{j=1}^{k-1} x_j^2}}\right), \quad \text{for } k = 1, 2, \ldots, n-2$$

$$\theta_{n-1} = \arctan\left(\frac{x_n}{x_{n-1}}\right).$$

The Jacobian J of this transform is

$$J = r^{n-1}\prod_{p=1}^{n-1}(\sin\theta_p)^{n-p-1}.$$

Problem 7. Consider the traveling salesman problem with six cities A, B, C, D, E, F, and their (x, y)-coordinates, i.e.,

$$\begin{aligned}
A &= (0,0), \quad B = (4,3), \quad C = (1,7) \\
D &= (15,7), \quad E = (15,4), \quad F = (18,0).
\end{aligned}$$

To find a solution of the traveling salesman problem the *greedy algorithm* can be applied as follows:

1) The shortest connection (Euclidean distance) between all the cities is $D-E$ with $d=3$. Thus we add this connection to the path.
2) The next shortest distances are the connection $B-C$, $A-B$ and $E-F$. They all have the same distance, namely $d=5$. Since they lead to no contradictions of the problem, we add these connections to the path.
3) The next shortest connection is $A-C$, but it leads to a loop, and thus cannot be added to the path.
4) The next shortest connection $D-F$ also leads to a loop, and thus cannot be added to the path.
5) The next shortest connection is $B-E$, but this line also contradicts the problem.
6) The next shortest connection $C-D$ can be added to the path.
7) To complete the path for a solution of the traveling salesman problem, we have to add $F-A$. Then the path is

$$A-B-C-D-E-F-A\,.$$

Is this the path with the shortest distance?

Solution 7. The greedy algorithm normally leads only to a local minimum. This is the case here. The path with the shortest distance is

$$A-C-D-E-F-B-A\,.$$

Problem 8. In the *knapsack problem*, we have a knapsack with a certain capacity (c) that we wish to pack with a number of items. Each item i has a value v_i and a weight w_i. The task is to pack the knapsack in such a way that the maximum value is obtained. If we can cut the items up, then the problem can be solved with a *greedy algorithm*:

- Calculate the value-to-weight ratio of every item.
- Repeatedly pack in the item with the heighest value-to-weight ratio until there is not enough space left for the object.
- Cut the highest value-to-weight ratio item to get a fractional amount that just fills the knapsack and put it in.

Usually we cannot have a fractional amount of an item, in which case the fraction of the item i, a_i, that we want to pack in must be either 0 or 1. This is known as the *Zero-One Knapsack Problem.* In this case, we can try to solve the problem using brute force: try every possible combination and see which is best. This technique is $O(2^n)$. A faster approach uses dynamic

programming. We consider packing all knapsacks with capacities 0 to c. We want to maximize

$$f_n(c) = \sum_{i=1}^{n} a_i v_i$$

subject to the constraint

$$\sum_{i=1}^{n} a_i w_i \leq c$$

where n is the number of items, c is the capacity of the knapsack and $a_i \in 0, 1$. We will try to pack each item i into knapsacks of size $0 \ldots c$. If we pack item i into a knapsack of size j, then the remaining capacity is $j - w_i$. We try to find the optimal packing of a knapsack of capacity $j - w_i$ with the $i - 1$ items before this item and then add item i. If the value of this packing is better than the current value for a knapsack of size j, then we use this packing. Since we are using integers, we can represent $f_n(c)$ as a two-dimensional array indexed by n and c. We have the properties

$$f_i(0) = 0, \quad i = 0, 1, ..., n$$
$$f_0(j) = 0, \quad j = 0, 1, ..., c.$$

To determine which items we take, we compare $f_i(c)$ and $f_{i-1}(c)$. If they are the same, we pack the item in the knapsack. If not, we leave it behind. If we take the item, then for the next item we must compare $f_{i-1}(c - w_i)$ and $f_{i-2}(c - w_i)$ to take take into account the fact that item i is taken. This algorithm is $O(nc)$. Write a C++ program to solve the Zero-One Knapsack Problem using this algorithm.

Solution 8.

```
// knapsack.cpp

#include <iostream>
#include <string>
using namespace std;

void knapsack(int c,int n,int *w,int *v,int *taken)
{
  int i, j, p;
  // allocate memory
  int **f = new int*[n+1];
  for(i=0;i<=n;i++) { f[i] = new int[c+1]; }
  // initial conditions
  for(j=0;j<=c;j++) { f[0][j] = 0; }
  for(i=0;i<=n;i++) { f[i][0] = 0; }
```

```
  for(i=1;i<=n;i++) { // for every item
    for(j=1;j<=c;j++) { // for every knapsack size
      f[i][j] = f[i-1][j]; // assume we don't take the item
       if(j-w[i-1]>=0) {
          // we have space for it
          // find the new value if we take it.
          p = f[i-1][j-w[i-1]]+v[i-1];
          if(p>f[i-1][j]) { f[i][j]=p; }
      }
    }
  }

  // search from the end, to see which items are taken
  i = n; j = c;
  while(i > 0) {
    if(f[i][j]!=f[i-1][j]) {
      j -= w[i-1];
      taken[i-1] = 1;
      } else { taken[i-1]=0; }
      i--;
  }
  for(i=0;i<=n;i++) { delete[] f[i]; }
  delete[] f;
}

int main(void)
{
  int n = 6, c = 20, weight = 0;
  string* items = new string[n];
  int* w = new int[n];  int* v = new int[n];
  int* taken = new int[n];
  w[0] = 3; v[0] = 5; items[0] = "torch";
  w[1] = 1; v[1] = 6; items[1] = "knife";
  w[2] = 5; v[2] = 5; items[2] = "bread";
  w[3] = 5; v[3] = 2; items[3] = "camera";
  w[4] = 10; v[4] = 15; items[4] = "clothes";
  w[5] = 4; v[5] = 1; items[5] = "books";
  knapsack(c,n,w,v,taken);
  cout << "Taken:" << endl;
    for(int i=0;i<n;i++) {
      if(taken[i]==1) {
        weight += w[i];
        cout << "   " << items[i] << endl;
```

```
      }
    }
  cout << "Total weight: " << weight << endl;
  delete[] items;
  delete[] w;
  delete[] v;
  delete[] taken;
  return 0;
}
```

Problem 9. The following table describes jobs (or tasks) according to their starting time and time of completion.

Job	Start	End
1	6	10
2	1	5
3	1	6
4	9	12
5	5	7
6	6	14
7	3	7
8	10	14
9	13	16

Obviously, there may be some conflicts, for example job 1 and job 6 would need to be done simultaneously. The scheduling problem requires that we find the subset of maximum size of jobs which do not conflict. For example doing jobs 3, 1, 9 in that order is a non-optimal solution. A *greedy algorithm* is an algorithm that optimizes the choice at each iteration without regard to previous choices. In the case of our scheduling problem, a greedy algorithm chooses the job which finishes in the least amount of time in order to fit as many jobs as possible. Obviously, this cannot in general lead to an optimal solution. Write a C++ program which implements a greedy algorithm to solve the scheduling problem. In order for the algorithm to be efficient, it is necessary to sort the jobs according to their times first.

Solution 9. Partition around the first element of the array any element could have been used. The variable `p` is the index of the element around which the partition is made `pe` (declared below) points to the element after the second partition

```
// jobs.cpp

#include <iostream>
```

```
using namespace std;

struct job { int number, start, end; };

// general definition of ordering R(t1,t2)
// returns > 0 if t2 R t1, <= 0 otherwise
template <class T>
void partition(T *array,int n,int (*R)(T,T),int &p)
{
  int i = n-1, pe = 1;
  T temp1, temp2;
  p = 0;
  while(i > 0)
  {
  if(R(array[p],array[pe]) > 0)
  {
  temp1 = array[p]; temp2 = array[p+1];
  array[p++] = array[pe];// put element in first partition
  array[p] = temp1;  // move element around which partition
                     // is made, one element right
  if(pe-p > 0)       // if the second partition is not empty
  array[pe] = temp2; // move second partition one
  }                  // element right
  pe++;
  i--;
  }
}

template <class T>
void qsort(T *array,int n,int (*R)(T,T))
{
  int pelement;
  if(n <= 1) return;
  partition(array,n,R,pelement);
  qsort(array,pelement,R);
  qsort(array+pelement+1,n-pelement-1,R);
}

int timeorder(struct job a,struct job b)
{
  if(a.start-b.start != 0) return a.start > b.start;
  return a.end > b.end;
}
```

```
void greedy(struct job *j,int n)
{
  int k = 0, i = 0, m;
  qsort(j,n,timeorder);
  m = j[0].start;
  while(i<n)
  {
  if(j[i].start>=m)
  {
  k++;
  m = j[i].end;
  cout << j[i].number << " " << j[i].start << " " << j[i].end
       << endl;
  }
  i++;
  }
}

int main(void)
{
  struct job jobs[] = {{ 1,  6, 10 },{ 2,  1,  5 },
                       { 3,  1,  6 },{ 4,  9, 12 },
                       { 5,  5,  7 },{ 6,  6, 14 },
                       { 7,  3,  7 },{ 8, 10, 14 },
                       { 9, 13, 16 }};
  greedy(jobs,9);
  return 0;
}
```

Problem 10. Find the shortest distance between two coordinates on the earth's surface using the *latitude* and *longitude*. Assume that the earth is a perfect sphere with radius = 6371.0 km.

```
             Latitude  Longitude
Example 1:
 Egoli:        26 10 S  28  2 E
 Tombouctou:   16 50 N   3  0 W

Example 2:
 Leningrad:    59 55 N  30 20 E
 San Fransico: 37 47 N 122 30 W
```

```
Example 3:
 Tecka          43 30 S   71   0 W
 Petropavlosk   52 50 N  158  50 E
```

Solution 10. We use *spherical coordinates*

$$x(r,\alpha,\beta) = r\cos(\beta)\sin(\alpha)$$
$$y(r,\alpha,\beta) = r\cos(\beta)\cos(\alpha)$$
$$z(r,\alpha,\beta) = r\sin(\beta)$$

where $-\pi \leq \alpha \leq \pi$ and $-\pi/2 \leq \beta \leq \pi/2$. Thus α describes the *longitude* and β describes the *latitude*. Given (α_1, β_1) and (α_2, β_2) we consider the vectors

$$\mathbf{p}_1 = r(\cos(\beta_1)\sin(\alpha_1), \cos(\beta_1)\cos(\alpha_1), \sin(\beta_1))$$
$$\mathbf{p}_2 = r(\cos(\beta_2)\sin(\alpha_2), \cos(\beta_2)\cos(\alpha_2), \sin(\beta_2))\,.$$

Let θ be the angle between $\mathbf{p}_1$ and $\mathbf{p}_2$. The *scalar product* provides

$$\mathbf{p}_1 \cdot \mathbf{p}_2 = \|\mathbf{p}_1\| \cdot \|\mathbf{p}_2\| \cos(\theta) = r^2 \cos(\theta)\,.$$

Since $\cos(\theta) = (\mathbf{p}_1 \cdot \mathbf{p}_2)/(r^2)$ we have

$$\cos(\theta) = \cos(\beta_1)\cos(\beta_2)\cos(\alpha_1 - \alpha_2) + \sin(\beta_1)\sin(\beta_2)$$

where we used $\sin(\alpha_1)\sin(\alpha_2) + \cos(\alpha_1)\cos(\alpha_2) \equiv \cos(\alpha_1 - \alpha_2)$. The distance d is given by $d = r\theta$. This leads to the C++ implementation

```
// distance.cpp

#include <iostream>
#include <cmath>
using namespace std;

int main(void)
{
  double r = 6371.0; // radius of the earth
  double M_PI = 3.14159;
  double alpha1, alpha2, beta1, beta2, temp, theta, distance;
  char dir, source[16], dest[16];
  cout << "enter the name of the source: ";
  cin >> source;
  cout << "enter the latitude of " << source << ": \ndegrees:";
  cin >> beta1;
```

```
  cout << "minutes: ";
  cin >> temp;
  beta1 += temp/60.0;
  cout << "N/S: ";
  cin >> dir;
  if(dir == 'S' || dir == 's') beta1 = -beta1;
  cout << "enter the longitude of " << source << ": \ndegrees:";
  cin >> alpha1;
  cout << "minutes: ";
  cin >> temp;
  alpha1 += temp/60.0;
  cout << "W/E: ";
  cin >> dir;
  if(dir == 'E' || dir == 'e') alpha1 = -alpha1;
  cout << "enter the name of the destination: ";
  cin >> dest;
  cout << "enter the latitude of " << dest << ": \ndegrees:";
  cin >> beta2;
  cout << "minutes: ";
  cin >> temp;
  beta2 += temp/60.0;
  cout << "N/S: ";
  cin >> dir;
  if(dir == 'S' || dir == 's') beta2 = -beta2;
  cout << "enter the longitude of " << dest << ": \ndegrees:";
  cin >> alpha2;
  cout << "minutes: ";
  cin >> temp;
  alpha2 += temp/60.0;
  cout << "W/E: ";
  cin >> dir;
  if(dir == 'E' || dir == 'e') alpha2 = -alpha2;
  alpha1 *= M_PI/180.0; alpha2 *= M_PI/180.0;
  beta1 *= M_PI/180.0; beta2 *= M_PI/180.0;
  temp = cos(beta1)*cos(beta2)*cos(alpha1-alpha2)
         + sin(beta1)*sin(beta2);
  theta = acos(temp);
  distance = r*theta;
  cout << "The distance between " << source << " and " << dest
       << " is " << distance << " km.\n";
  return 0;
}
```

Chapter 18

File and String Manipulations

Problem 1. *Intel hex record* files are printable files consisting of any number of Intel hex records. Each record is represented by exactly one line of the format

```
:CCAAAARRDD...DZZ
```

The leading colon indicates that the line is an Intel hex record. The characters following the colon are hexadecimal (base 16) digits, i.e., `0` to `9` and `A` to `F`. The first two digits represented by `CC` determine the number of data bytes (represented by two hexadecimal digits each). The next four digits, `AAAA`, is an address. The two digits `RR` indicate the record type:

- `00` for a data record
- `01` for an end of file record
- `02` for an extended segment address record
- `03` for a start segment address record
- `04` for an extended linear address record
- `05` for a start linear address record.

This is followed by the data bytes. Lastly two hexadecimal digits `ZZ` give the *checksum*, which is the two's complement of the sum of all the previous

bytes (modulo 256).

For example, the record

```
:020000021000EC
```

has 2 data bytes, and is a data record, with address 0. The checksum EC in base 10 is

```
EC = 14*16+12 = 236
```

The sum of the preceding bytes 02 00 00 02 10 00 is

$$2 + 0 + 0 + 2 + 16 + 0 = 20$$

in decimal. In binary 20 is represented as

```
00010100
```

The 8 bit one's complement of this bitstring is

```
11101011
```

Thus the 8 bit two's complement is

```
11101100
```

This gives the result

$$2^7 + 2^6 + 2^5 + 2^3 + 2^2 = 128 + 64 + 32 + 8 + 4 = 236.$$

Write a program to read an Intel hex record file, which uses the checksum of each record to determine if the record is correct. Use the program with the following file

```
:100000008316FF3085000030860083120230092OFD
:0C0010008600820741344234433444343FB
:00000001FF
```

Solution 1. The following C++ program checks the validity of the records. The function `getbyte()` is used to obtain an integer from a character stream of hexadecimal numbers. Adding the checksum to the data bytes yields 0.

```
// intelhex.cpp

#include <iostream>
#include <fstream>
#include <ctype.h>
using namespace std;

char *record_type[] = { "DATA","EOF","EXTENDED SEGMENT",
                        "START SEGMENT","EXTENDED LINEAR",
                        "START LINEAR" };

int getbyte(istream& i)
{
  int bytehigh = 0, bytelow = 0;
  bytehigh = i.get();
  if(!isxdigit(bytehigh)) { i.unget(); return -1; }
  if(!isdigit(bytehigh)) bytehigh=10+toupper(bytehigh)-'A';
  else bytehigh -= '0';
  bytelow = i.get();
  if(!isxdigit(bytelow)) { i.unget(); return -1; }
  if(!isdigit(bytelow)) bytelow = 10+toupper(bytelow)-'A';
  else bytelow -= '0';
  return ((bytehigh << 4) + bytelow);
}

int main(int argc,char *argv[])
{
  int i, c, extra, valid, line = 0;
  int databytes, address, type;
  char checksum;
  if(argc!=2)
  { cout << "Usage: " << argv[0] << " file"
         << endl; return 1; }
  ifstream f(argv[1]);
  if(f.fail())
  { cout << "Failed to open " << argv[1]
         << endl; return 1; }
  while(!f.eof())
  {
  extra = 0; valid = 0;
  if(f.get()==':')
  {
   cout << "Record on line " << line <<" ";
   if((databytes=getbyte(f)) >= 0)
```

```
    if((c=getbyte(f)) >= 0) if(address=c,(c=getbyte(f))>=0)
    if((type=getbyte(f)) >= 0)
    {
   checksum = databytes+address+c+type;
   address = (address << 8)+c;
   for(i=0;i<=databytes;i++)
     if((c=getbyte(f)) >= 0) checksum += c;
     else break;
    if(i>databytes) valid = 1;
   }
   if(!valid) cout << "invalid format" << endl;
   else
   cout << "of type " << record_type[type] << " with "
        << databytes << " databytes, "
        << "at address " << address
        << " - checksum " << ((checksum)?"invalid.":"valid.")
        << endl;
   } else f.unget();
   while(((c=f.get())!='\n') && !f.eof()) extra += !isspace(c);
   if(extra)
     cout << "Ignoring data on line " << line
          << " which is not part of a record." << endl;
   line++;
   }
   return 0;
}
```

For the example input file, the output is

```
Record on line 0 of type DATA with 16 databytes,
at address 0 - checksum valid.
Record on line 1 of type DATA with 12 databytes,
at address 16 - checksum valid.
Record on line 2 of type EOF with 0 databytes, at
address 0 - checksum valid.
```

Problem 2. The *LZW-algorithm* is a general purpose compression scheme developed by Lempel-Ziv and Welch. The algorithm uses a table of 4096 entries to store frequently used patterns of information. Initially, the first 256 entries are set to the values a single byte can take on (i.e., each entry i has the value i for $i = 0, 1, \ldots, 255$). Each of the first 512 entries of the table are represented using 9 bits. When the table is filled up to exceed 512 entries, then 10 bits are used for each index. Similarly 11 and 12 bits are used for each index when the table exceeds 1024 and 2048 entries

respectively. The first 258 table entries are fixed. The entry 256 indicates a *clear code* which indicates all entries above 257 should be cleared. The entry 257 indicates end of information, i.e., the current block of data is completely encoded. The way the table is filled up is as follows

1. P=""
2. read C (a character)
3. if P+C (string concatenation) is a string in the table
   P=P+C
   goto 2
4. write the index in the table where P was found
   add P+C to the table
   P=""
   goto 3

When the table is full (all 4096 entries used) the table must first be cleared before starting a new table. Thus the index 256 must be written to the output. Write a program which encodes data using the algorithm. Write a C++ program which decodes the encoded data.

Solution 2. We first write a `Bitstream` class which allows us to read or write an arbitrary number of bits to or from a file. We achieve this by buffering to fill up complete bytes before returning after reading or writing.

```
// bitstream.h

#ifndef BITSTREAM_HEADER
#define BITSTREAM_HEADER

#include <iostream>
using namespace std;

const int BUFFERSIZE=1024;

class OBitStream
{
  private:
   unsigned char b[BUFFERSIZE];
   unsigned int p, q;
   ostream& stream;
  public:
   OBitStream(ostream &o): stream(o) { p=q=0; b[0]=0; }
```

```
    void write(unsigned char* bits,unsigned int number);
    void flush(void);
    ~OBitStream()
    { if(q) p++; stream.write((char*)b,p*sizeof(unsigned char));
};

// assume that any unused bits in the array bits
// are set to zero
void OBitStream::write(unsigned char* bits,unsigned int number)
{
  int j = (8*sizeof(unsigned char));
  int m = number/j; int n = number%j;
  for(int i=0;i<m;i++)
  {
  b[p] |= (bits[i]<<q);
  b[++p] = (bits[i]>>(j-q));
  if(p>=BUFFERSIZE-2) flush();
  }
  if(n)
  {
  b[p] |= (bits[m] << q);
  if(n >= j-q) b[++p] = (bits[m] >> (j-q));
  if(p >= BUFFERSIZE-2) flush();
  q = (q+n)%j;
  }
}

void OBitStream::flush(void)
{
  if(p > 0) stream.write((char*)b,p*sizeof(unsigned char));
  b[0] = b[p]; p = 0;
}

class IBitStream
{
  private:
    unsigned char b;
    unsigned int q;
    static unsigned int masks[];
    istream& stream;
  public:
    IBitStream(istream &i): stream(i) { q=0; b=0;}
    void read(unsigned char* bits,unsigned int number);
    int eof(void) { return stream.eof(); }
```

```
    ~IBitStream() { }
};

unsigned int IBitStream::masks[] = { 0,1,3,7,15,31,63,
                                     127,255,511,1023,2047 };

// bits is an array of size at least number/(sizeof(char)) + 1
void IBitStream::read(unsigned char* bits,unsigned int number)
{
  int j = (8*sizeof(unsigned char));
  int m = number/j; int n = number%j;
  bits[0] = b;
  memset(bits+1,0,m);
  stream.read((char*)(bits+1),m*sizeof(unsigned char));
  for(int i=0;i<m;i++)
  {
  bits[i] |= (bits[i+1]<<q); bits[i+1] >>= (j-q);
  }
  if(n<=q) { b=bits[m]; bits[m] &= masks[n]; q-=n; b>>=n; }
  else
  {
  b=0;
  stream.read((char*)&b,sizeof(unsigned char));
  bits[m] |= (b<<q); bits[m] &= masks[n];
  b >>= n-q; q = j-(n-q);
  }
}
#endif
```

Now we can write a class to write data compressed from a buffer and uncompress data from a file into a buffer. These classes can be applied for any binary data, including ASCII data.

```
// lzw.h

#ifndef LZWHEADER
#define LZWHEADER

#include <cstring>
#include <limits>
#include "bitstream.h"

const int MAXLEN=1024;
```

```
unsigned char *uint_to_bytes(unsigned int u)
{
 static const int n=sizeof(unsigned int);
 static const int m=numeric_limits<unsigned char>::max()+1;
 static unsigned char b[n];
 for(int i=0;i<n;i++,u/=m) b[i]=u%m;
 return b;
}

unsigned int bytes_to_uint(unsigned char *b)
{
 static const int n=sizeof(unsigned int);
 static const int m=numeric_limits<unsigned char>::max()+1;
 unsigned int u=0;
 for(int i=n-1;i>=0;i--) u = u*m+b[i];
 return u;
}

class LZWOStream
{
  private:
    OBitStream *bs;
    unsigned char *table[4096];
    int lengths[4096];
    unsigned int size, bits, bestmatch;
    int p;
    unsigned char P[MAXLEN];
  public:
    LZWOStream(ostream& s);
    void write(unsigned char*,unsigned int);
    ~LZWOStream();
};

LZWOStream::LZWOStream(ostream& s)
{
  int i;
  bs = new OBitStream(s);
  for(i=0;i<256;i++)
  { table[i]=new unsigned char[1]; table[i][0]=i; lengths[i]=1;}
  lengths[i++] = 0; lengths[i++] = 0;
  size = 258; bits = 9; p = 0;
}

void LZWOStream::write(unsigned char *b,unsigned int n)
```

```
{
  int j;
  if((n>0)&&(p==0)) { P[p++]=*(b++); n--; }
  while(n)
  {
  for(j=0;j<size;j++)
   if(lengths[j]==p && memcmp(P,table[j],p)==0) break;
  if(j!=size) { bestmatch=j; P[p++]=*(b++); n--; }
  else
  {
  bs -> write(uint_to_bytes(bestmatch),bits);
  if(size==4096)
  {
  j = 256;
  bs -> write(uint_to_bytes(j),bits);
  for(j=258;j<size;j++) { delete[] table[j]; lengths[j]=0; }
  size = 258; bits = 9;
  P[0] = P[p-1]; p = 1;
  }
  else
  {
  table[size] = new unsigned char[p];
  lengths[size]=p;
  memcpy(table[size++],P,p);
  P[0] = P[p-1]; p = 1;
  if(size>512) bits = 10; if(size>1024) bits = 11;
  if(size>2048) bits = 12;
  }
  }
 }
}

LZWOStream::~LZWOStream()
{
  if(p>0) {
  bs -> write(uint_to_bytes(bestmatch),bits);
  p -= lengths[bestmatch];
  }
  for(int j=0;j<size;j++) delete[] table[j];
  bestmatch = 257;
  bs -> write(uint_to_bytes(bestmatch),bits);
  delete bs;
}
```

```
class LZWIStream
{
  private:
    IBitStream* bs;
    unsigned char *table[4096];
    int lengths[4096];
    unsigned int size, bits;
    int iseof;
  public:
    LZWIStream(istream &s);
    int read(unsigned char*,unsigned int);
    int eof();
    ~LZWIStream();
};

int LZWIStream::eof() { return iseof; }

LZWIStream::LZWIStream(istream& s)
{
  int i;
  bs = new IBitStream(s);
  for(i=0;i<256;i++)
  { table[i]=new unsigned char[1]; table[i][0]=i; lengths[i]=1;}
  lengths[i++] = 0; lengths[i++] = 0;
  size = 258; bits = 9; iseof = 0;
}

int LZWIStream::read(unsigned char* b,unsigned int n)
{
  int j, i = 0;
  unsigned char index_bytes[sizeof(unsigned int)];
  unsigned int index = 0;
  if(n>MAXLEN) n-=MAXLEN;
  while((n>i) && !bs->eof() && !iseof)
  {
  memset(index_bytes,0,sizeof(unsigned int));
  bs -> read(index_bytes,bits);
  index = bytes_to_uint(index_bytes);
  if(bs->eof()) { iseof=1; break; }
  if(index==256)
  {
  for(j=258;j<size;j++) { delete[] table[j]; lengths[j]=0; }
  size=258; bits=9;
  }
```

```
  else if(index == 257) { iseof=1; return i; }
  else if(index >= size) return -1;
  else
  {
  if(size > 258)
  {
  unsigned char *entry=new unsigned char[lengths[size-1]+1];
  memcpy(entry,table[size-1],lengths[size-1]);
  entry[lengths[size-1]++]=table[index][0];
  delete[] table[size-1]; table[size-1]=entry;
  }
  if(size < 4096)
  {
  table[size] = new unsigned char[lengths[index]];
  memcpy(table[size],table[index],lengths[index]);
  lengths[size++]=lengths[index];
  }
  memcpy(b+i,table[index],lengths[index]);
  i += lengths[index];
  if(size>512) bits = 10; if(size>1024) bits = 11;
  if(size>2048) bits = 12;
  }
  }
  return i;
}

LZWIStream::~LZWIStream()
{
  delete bs;
  for(int j=0;j<size;j++) delete[] table[j];
}
#endif
```

Finally we use the classes in programs to compress and decompress files.

```
// compress.cpp

#include <fstream>
#include "lzw.h"
using namespace std;

int main(int argc,char *argv[])
{
  if(argc!=3)
```

```
  {
  cout << "Use:" << argv[0] << " filename compressed_filename"
       << endl;
  return 1;
  }
  ofstream out(argv[2],ios::binary);
  ifstream in(argv[1],ios::binary);
  LZWOStream lzw(out);
  unsigned char b[4096];
  while(!in.fail() && !in.eof())
  {
  int n=0;
  while(!in.eof() && n<4096) in.read((char*)b+(n++),1);
  if(n>0) lzw.write(b,n);
  }
  return 0;
}

// decompress.cpp

#include <fstream>
#include "lzw.h"
using namespace std;

int main(int argc,char *argv[])
{
  if(argc!=3)
  {
  cout << "Use:" << argv[0] << " compressed_filename filename"
       << endl;
  return 1;
  }
  ofstream out(argv[2],ios::binary);
  ifstream in(argv[1],ios::binary);
  LZWIStream lzw(in);
  unsigned char b[4096];
  while(!lzw.eof())
  {
  int n = lzw.read(b,4096);
  if(n>0) out.write((char*)b,n);
  }
  return 0;
}
```

Problem 3. The major problem with the brute-force search of a string in a text is that characters in the text may be re-examined multiple times and this can lead to poor performance in some cases. The algorithm of Knuth, Morris and Pratt provides a way to alleviate the repeated accesses to the text and, as a result, it gives us a guaranteed linear time searching algorithm. The key aspect of the *Knuth-Morris-Pratt algorithm* is that a failed attempt to find a match yields useful information to be used on the next attempt. Specifically, if a mismatch is detected when considering the characters `pat[j]` and `text[k]`, we do not need to start the next attempt at `text[k-j+1]` as we know the characters

`text[k-j], text[k-j+1], ... , text[k-1]`

are identical to the prefix of the pattern,

`pat[0], pat[1], ... , pat[j-1]` .

Thus, we can access the text characters sequentially and alleviate the need to back-up the text. The KMP algorithm is essentially the brute-force algorithm with a more intelligent re-initialization of pointers when a mismatch is detected. Write a C++ program for the KMP-algorithm.

Solution 3.

```
// kmp.cpp

#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int prefix(int from,int to,string word)
{
  for(int i=0;from<=to;i++,from++)
  { if(word[from] != word[i]) return 0; }
  return 1;
}

void initnext(string word,int*& next)
{
  int wordlength = word.length();
  next = new int[wordlength];
  for(int i=0; i<wordlength;i++)
  {
  if(word[i] == word[0]) { next[i] = 0; }
  else
```

```
  {
  int found = 0;
  for(int j=0;(j<i && !(found));j++)
  {
  if(prefix(j,(i-1),word) && word[(i-j)] != word[i])
  {
  next[i] = (i-j)+1;
  found = 1;
  } // if
  } // for
  if(!(found)) next[i] = 1;
  } // ifelse
  } // for
} // initnext

int search(string word,ifstream& text,int& pos,int begin=1)
{
  int* next;
  int j, k, wordlength;
  char ch;
  wordlength = word.length();
  initnext(word,next);
  text.seekg((begin-1),ios::beg);
  text.get(ch);
  for(j=0;!text.eof();)
  {
  if(wordlength == j)
  {
  int temp = text.tellg();
  pos = temp - wordlength;
  return 1;
  } // if
  if(ch == word[j]) { text.get(ch); j++; }
  else
  {
  if(next[j] == 0) { j = 0; text.get(ch); }
  else { j = (next[j]-1); } // ifelse
  } // ifelse
  } // for
  return 0;
} // search

int main(int argc,char* argv[])
{
```

```
  int pos;
  if(argc != 3)
  {
  cout << "usage: kmp <string> <file_name> \n";
  exit(1);
  }
  ifstream file(argv[2],ios::in);
  if(search(argv[1],file,pos))
  {
  cout << "\nThe string \"" << argv[1]
       << "\" occurs the first time at position "
       << pos << "\n\n";
  }
  else
  {
  cout << "\nThe string \"" << argv[1]
       << "\" does not occur in " << argv[2] << "\n\n";
  }
  return 0;
}
```

Another implementation of the KMP search is

```
// kmp1.cpp

#include <iostream>
using namespace std;

int kmp(char* needle,char* haystack)
{
  // support a maximum of 256 character needles
  int skiplist[256];
  int i, j, n, m, skip;
  skiplist[0] = 0; skiplist[1] = 1;
    for(m=2;m<=strlen(needle);m++) {
      for(i=m-1;i>0;i--) {
        for(j=0;j<i;j++) {
          if(needle[j]!=needle[m-i+j]) break;
          }
        if(j==i) { break; }
        }
    skiplist[m] = m-i;
    }
    m = strlen(needle); n = strlen(haystack);
    i = 0; j = 0;
```

```
    while(i<=n-m) {
      while(j<m) {
        if(haystack[i+j] != needle[j]) {
          if(j==0) {
            i++;
              if(i>n-m) return -1;
               } else {
               skip = skiplist[j];
               i += skip; j -= skip;
         }
      } else { j++; }
  }
  return i;
  }
  return -1;
}

int main(void)
{
  char* needle = "ring";
  char* haystack = "Look for a substring";
  int location = kmp(needle,haystack);
  cout << "Found " << endl << needle << endl << " in " << endl
       << haystack << endl << " at position "
       << location << endl;
  return 0;
}
```

Problem 4. A common task in string manipulation is to search a string (or file) for the occurrences of a particular substring (or piece of text). What is needed is a function which could take a string such as

```
Isn't it funny how bees fly
```

and a substring such as `funny` and tell us that the substring occurs in the string at character position 9. The first character position is numbered zero. The straightforward approach is to scan the string for an `f` and when one is found the characters after the `f` are checked to see whether they match. However, this is not the best option. It involves checking every character of the string, which can be very slow. A better approach is given by the *Boyer-Moore algorithm.* In their method a table is built from the substring indicating the position of the last occurrence of each character within it. For the string `funny` this would hold the information that

`f at 0, u at 1, n at 3, y at 4.`

The algorithm then involves searching in the string for the last character of the substring as follows: the substring contains five characters, numbered zero to four. Obviously, we must not check the string before character number 4, which is a `t`. The table is checked and reveals that there is no blank in the substring and therefore we can skip five characters to the `f`. This is in the table and is at position zero in the substring. Thus, four characters can be skipped to the `y`. This is the character being searched for and so a comparison is made at this point between the whole substring and the appropriate part of the string. This operation finds a match. Write a C++ program that implements this algorithm.

Solution 4.

```
// boyer.cpp

#include <iostream>
#include <fstream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <climits>
using namespace std;

const int charsetsize = 128;

int substringposition(char *str,int strleng,char *substr,
int substrleng,int contentlist[])
{
  int result, index, position, firstchar;
  result = -1;
  char finalchar, currentchar;
  finalchar = substr[substrleng-1];
  char *stringptr, *substrptr;
  if(strleng >= substrleng)
  {
  if(substrleng == 1)
  {
  for(index = 0;index < strleng && result == -1;++index)
  if(finalchar == str[index])
  result = index;
  }
  else
  {
```

```
  index = substrleng-1;
  do
  {
  if((position = contentlist[currentchar = str[index]])==-1)
  index += substrleng;
  else
  if(currentchar == finalchar)
  {
  firstchar = index - substrleng + 1;
  if(strncmp(&str[firstchar],substr,substrleng) == 0)
  result = firstchar;
  if(result == -1) ++index;
  }
  else index += (substrleng - position - 1);
  }
  while(index < strleng && result == -1);
  }
  }
  return result;
}

void makecontentlist(char *substr,int contentlist[])
{
  for(int i=0;i<= charsetsize-1;++i)
  contentlist[i] = -1;
  for(int j=0;*substr != '\0';++j,++substr)
  contentlist[*substr] = j;
}

int main(void)
{
  const int lineleng = 132;
  const int nameleng = 12;
  int line = 1;
  int searchstringleng;
  int contentlist[charsetsize];
  char currentline[lineleng], filename[nameleng],
  searchstring[lineleng];
  cout << "enter name of file to be searched: ";
  cin >> filename;
  FILE* InStream = fopen(filename,"r");
  if(!InStream)
  {
  cout << " *** Error *** : cannot open " << filename << endl;
```

```
  exit(1);
  }
  cout << "enter the substring to be searched: ";
  cin >> searchstring; // gets(searchstring);
  makecontentlist(searchstring,contentlist);
  searchstringleng = strlen(searchstring);
  while(fgets(currentline,lineleng,InStream) != NULL)
  {
  if(substringposition(currentline,strlen(currentline),
  searchstring,searchstringleng,contentlist) != -1)
  printf("line: %d text: %s",line,currentline);
  ++line;
  }
  return 0;
}
```

Problem 5. If two streams are opened, one as a binary stream and the other as a text stream, and the same information `hello world\n` is written to both, the contents of the stream may differ. The C++ code shows two streams of different types and the hexadecimal values of the resulting files. Discuss why the data is stored differently for each file.

```
// textbinary.cpp

#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;

int main(void)
{
  char lineBin[15], lineTxt[15];
  ofstream ofile1("script.bin", ios::out|ios::binary);
  ofile1 << "hello world\n";
  ofile1.close();
  ofstream ofile2("script.txt");
  ofile2 << "hello world\n";
  ofile2.close();
  ifstream ifile1("script.bin",ios::in|ios::binary);
  // opening the text file as binary to supress
  // the conversion of internal data
  ifstream ifile2("script.txt",ios::in|ios::binary);
  ifile1.getline(lineBin,15); ifile2.getline(lineTxt,15);
  ifile1.close(); ifile2.close();
```

```
  cout << setbase(16);
  cout << "Hex value of binary file = ";
  for(int i=0;lineBin[i];i++)
    cout << setw(2) << setfill('0') << (int)lineBin[i];
  cout<<endl;
  cout << "Hex value of text file = ";
  for(int j=0;lineTxt[j];j++)
    cout << setw(2) << setfill('0') << (int)lineTxt[j];
    cout << endl;
  return 0;
}
```

Solution 5. First we note that the function `getline()` discards the new-line character (line-feed) `\0a`. Under Windows the output is

```
Hex value of binary file = 68656c6c6f20776f726c64
Hex value of text file   = 68656c6c6f20776f726c640d
```

In the text stream `script.txt`, we have a carriage-return character `\0d` at the end of the file. Under Linux we find that the binary file and text file are the same, namely

```
68656c6c6f20776f726c64
```

Problem 6. The *DNA molecule* is a two stranded molecule. Each strand is a polynucleotide composed of A (adenosine), T (thymidine), C (cytidine) and G (guanosine) residues polymerized by dehydration synthesis in linear chains with specific sequences. Each strand has polarity, such that the 5′-hydoxyl (or 5′-phospho) group of the first nucleotide begins the strand and the 3′-hydroxyl group of the final nucleotide ends the strand accordingly. We say that this strand runs 5′ to 3′. The two strands of DNA run antiparallel such that one strand runs `5' -> 3'` while the other runs `3' -> 5'`. At each nucleotide residue along the double-stranded DNA molecule, the nucleotide are complementary. That is, A froms two hydrogen-bonds with T; C forms three hydrogen bonds with G. One strand of DNA holds the information that codes for various genes. This strand is called the template strand or antisense strand (containing anticodons). The other, and complementary, strand is called the coding strand or sense strand (containing codons). Since mRNA is made from the template strand, it has the same information as the coding strand. Each group of three consecutive nucleotides is called a *codon*. Each triplet (codon) encodes exactly one amino acid or serves as an indicator for starting (methionine `MET`) and stopping the synthesis. The mapping of codons to amino acids is called the *genetic code* and is often presented in the following two-dimensional form.

```
     T              C              A              G
   phenylalanine  serine         tyrosine       cysteine     T
   phenylalanine  serine         tyrosine       cysteine     C
T  leucine        serine         period         period       A
   leucine        serine         period         tryptophan   G
   ------------------------------------------------------------

   leucine        proline        histidine      arginine     T
   leucine        proline        histidine      arginine     C
C  leucine        proline        glutamine      arginine     A
   leucine        proline        glutamine      arginine     G
   ------------------------------------------------------------

   isoleucine     threonine      asparagine     serine       T
   isoleucine     threonine      asparagine     serine       C
A  isoleucine     threonine      lysine         arginine     A
   methionine     threonine      lysine         arginine     G
   ------------------------------------------------------------
   valine         alanine        aspartic acid  glycine      T
   valine         alanine        aspartic acid  glycine      C
G  valine         alanine        glutamic acid  glycine      A
   valine         alanine        glutamic acid  glycine      G
   ------------------------------------------------------------
```

For example, `TCG` is the codon for serine, `CAA` is the codon for glutamine and `GTC` is the codon for valine. The codons `TAA`, `TAG` and `TGA` do not code for an amino acid but serve as `halt` instructions. The table lists `period` for those codons instead of the name for an amino acid. The genetic code is more naturally presented in three-dimensional form. An example of two complementary strands of DNA would be

```
(5' -> 3') ATGGAATTCTCGCTA (coding, sense strand)
(3' -> 5') TACCTTAAGAGCGAG (template, antisense strand)

(5' -> 3') AUGGAAUUCUCGCUC (mRNA made from template strand).
```

The code for the mRNA would be identical to the sense strand but the mRNA contains U (uridine) rather than T. Use the class `TreeMap` of Java and the methods

```
Object put(Object key,Object value)
Object get(Object key)
```

to implement this map. Thus, for example the reference `get("GTC")` should fetch valine. Test the program by reading DNA sequences and then listing

the sequence of amino acids that they encode. Stop encoding when we come to a codon that codes for period.

Solution 6. We introduce the following abbreviation for the amino acids:

Alanine (ala), Arginine (arg), Asparagine (asn), Aspartic acid (asp), Cysteine (cys), Glutamine (gln), Glutamic acid (glu), Glycine (gly), Histidine (his), Isoleucine (ile), Leucine (leu), Lysine (lys), Methionine (met), Phenylalanine (phe), Proline (pro), Serine (ser), Threonine (thr), Tryptophan (trp), Tyrosine (tyr), Valine (val).

The Java program is as follows.

```
// GCode.java

import java.util.*;
import java.io.*;

public class GCode
{
  public static void main(String[] args)
  {
  // codon (key)
  String[] c = new String[64];
  c[0] = new String("AAA"); c[1] = new String("AAC");
  c[2] = new String("AAG"); c[3] = new String("AAT");
  c[4] = new String("ACA"); c[5] = new String("ACC");
  c[6] = new String("ACG"); c[7] = new String("ACT");
  c[8] = new String("AGA"); c[9] = new String("AGC");
  c[10] = new String("AGG"); c[11] = new String("AGT");
  c[12] = new String("ATA"); c[13] = new String("ATC");
  c[14] = new String("ATG"); c[15] = new String("ATT");
  c[16] = new String("CAA"); c[17] = new String("CAC");
  c[18] = new String("CAG"); c[19] = new String("CAT");
  c[20] = new String("CCA"); c[21] = new String("CCC");
  c[22] = new String("CCG"); c[23] = new String("CCT");
  c[24] = new String("CGA"); c[25] = new String("CGC");
  c[26] = new String("CGG"); c[27] = new String("CGT");
  c[28] = new String("CTA"); c[29] = new String("CTC");
  c[30] = new String("CTG"); c[31] = new String("CTT");
  c[32] = new String("GAA"); c[33] = new String("GAC");
  c[34] = new String("GAG"); c[35] = new String("GAT");
  c[36] = new String("GCA"); c[37] = new String("GCC");
  c[38] = new String("GCG"); c[39] = new String("GCT");
```

```
c[40] = new String("GGA"); c[41] = new String("GGC");
c[42] = new String("GGG"); c[43] = new String("GGT");
c[44] = new String("GTA"); c[45] = new String("GTC");
c[46] = new String("GTG"); c[47] = new String("GTT");
c[48] = new String("TAA"); c[49] = new String("TAC");
c[50] = new String("TAG"); c[51] = new String("TAT");
c[52] = new String("TCA"); c[53] = new String("TCC");
c[54] = new String("TCG"); c[55] = new String("TCT");
c[56] = new String("TGA"); c[57] = new String("TGC");
c[58] = new String("TGG"); c[59] = new String("TGT");
c[60] = new String("TTA"); c[61] = new String("TTC");
c[62] = new String("TTG"); c[63] = new String("TTT");

// amino acid (value)
String[] a = new String[64];
a[0] = new String("lys");  a[1] = new String("asn");
a[2] = new String("lys");  a[3] = new String("asn");
a[4] = new String("thr");  a[5] = new String("thr");
a[6] = new String("thr");  a[7] = new String("thr");
a[8] = new String("arg");  a[9] = new String("ser");
a[10] = new String("arg");  a[11] = new String("ser");
a[12] = new String("ile");  a[13] = new String("ile");
a[14] = new String("met");  a[15] = new String("ile");
a[16] = new String("gln");  a[17] = new String("his");
a[18] = new String("gln");  a[19] = new String("his");
a[20] = new String("pro");  a[21] = new String("pro");
a[22] = new String("pro");  a[23] = new String("pro");
a[24] = new String("arg");  a[25] = new String("arg");
a[26] = new String("arg");  a[27] = new String("arg");
a[28] = new String("leu");  a[29] = new String("leu");
a[30] = new String("leu");  a[31] = new String("leu");
a[32] = new String("glu");  a[33] = new String("asp");
a[34] = new String("glu");  a[35] = new String("asp");
a[36] = new String("ala");  a[37] = new String("ala");
a[38] = new String("ala");  a[39] = new String("ala");
a[40] = new String("gly");  a[41] = new String("gly");
a[42] = new String("gly");  a[43] = new String("gly");
a[44] = new String("val");  a[45] = new String("val");
a[46] = new String("val");  a[47] = new String("val");
a[48] = new String(".");    a[49] = new String("tyr");
a[50] = new String(".");    a[51] = new String("tyr");
a[52] = new String("ser");  a[53] = new String("ser");
a[54] = new String("ser");  a[55] = new String("ser");
a[56] = new String(".");    a[57] = new String("cys");
```

```
  a[58] = new String("trp");  a[59] = new String("cys");
  a[60] = new String("leu");  a[61] = new String("phe");
  a[62] = new String("leu");  a[63] = new String("phe");

  TreeMap map = new TreeMap();
  for(int i=0;i < 64;i++) { map.put(c[i],a[i]); }
  File f = new File("DNA.txt");
  long leng = f.length();
  System.out.println("leng = " + leng);
  int i = 0;
  try
  {
  DataInputStream in = new DataInputStream(
                       new BufferedInputStream(
                        new FileInputStream("DNA.txt")));
  while(i < leng)
  {
  char c1 = (char) in.readByte();
  char c2 = (char) in.readByte();
  char c3 = (char) in.readByte();
  char data[] = { c1, c2, c3 };
  String dat = new String(data);
  Object ob1 = map.get(dat);
  String s1 = ob1.toString();
  System.out.println(dat + " -> " + s1);
  if(s1.equals("."))
  { System.out.println("synthesis stopped"); System.exit(0); }
  i += 3;
  } // end while
  in.close();
  } // end try block
  catch(IOException e)
  {
  System.err.println("IOException");
  }
  }
}
```

An input file could be ATGGAATTCTCGCTCTAGGTC.

Problem 7. Use the standard template library `vector` class and the `sort()` function in class `algorithm` to write a C++ program that reads a sequence of strings from an input file and writes them to an output file in sorted order, one per line.

Solution 7. The `sort()` function in class `algorithm` of the standard template library takes two iterators as arguments, one indicating the beginning of the data and the second refers to the position after the end of the data. The data between these iterators will be sorted provided the comparision operator < is defined.

```
// sorting.cpp

#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

int main(void)
{
  string name, next;
  ifstream input;
  ofstream output;
  cout << "input file name? "; cin >> name;
  input.open(name.c_str());
  cout << "output file name? "; cin >> name;
  output.open(name.c_str());
  vector<string> values;
   while(input >> next) values.push_back(next);
  vector<string>::iterator i = values.begin();
   while(i != values.end()) { cout << *i << endl; i++; }
  sort(values.begin(),values.end());
   for(int j=0;j < values.size();j++)
     output << values[j] << endl;
  return 0;
}
```

Problem 8. The *Baeza-Yates-Gonnet algorithm* (BYG algorithm) for searching for a pattern in a string runs 40-50 percent faster than KMP. The technique works as follows:

- Build a table of bitmasks for each letter of the alphabet, the mask has the length of the search pattern. Each bit is 1 (if that letter is not in that position) or 0 (if that letter is in that position).

- Create a working bitmask, initialized to all 1's.

- For each letter in the search string, shift the working mask 1 bit to the left, and perform the logical or with the bitmask for the letter in that position of the string.
- Let the length of the search pattern be x. If the bit in position x of the working bitmask is 0, then the string matches.
- If we reach the end of the string without finding a match, then there is no match.

Write a C++ program to match strings using the BYG algorithm.

Solution 8. We are using the `bitset` class.

```
// byg.cpp

#include <iostream>
#include <string>
#include <bitset>
using namespace std;

// bitmasks for the 256 ascii characters
const int MASKSIZE = 32;
bitset<MASKSIZE> masks[256];
string pattern;

void prepare_masks(string s)
{
  for(int i=0;i<256;i++) {
    masks[i].reset();
     for(int j=0;j<s.length();j++) {
       if(((unsigned char)s[j])==i) masks[i].set(j);
     }
  // invert the mask
  masks[i].flip();
  }
  pattern = s;
}

int match(string s)
{
  int i = 0;
  while(i<s.length()) {
    bitset<MASKSIZE> work;
      work.flip();
```

```
        while(i<s.length()) {
          work <<= 1;
            work|=masks[(unsigned char)s[i]];
              if(!(work.test(pattern.length()-1)))
              { return i+1-pattern.length(); }
           i++;
        }
  }
  return -1;
}

int main(void)
{
  string needle, haystack;
  int pos;
  cout << "Enter string to search in: ";
  getline(cin,haystack);
  cout << "Enter search string: "; cin >> needle;
  prepare_masks(needle);
  if((pos=match(haystack))>0)
  {
  cout << "Match found at " << pos << " starting at 0" << endl;
  }
  else { cout << "No match." << endl; }
  return 0;
}
```

Problem 9. Write a C++ program to evaluate *arithmetic expressions.* The program will have to decompose a string into a sequence of tokens, where each token represents an operator, function or value. The sequence of tokens are evaluated using the standard precedence rules for arithmetic, i.e., first parenthesis, functions, multiplication, division, modulus, and finally addition and subtraction.

Solution 9. The function `get_tokens()` separates a string into tokens according to `separator` (an array of symbols used to separate tokens and are tokens themselves) and `ignore` (an array of symbols used to separate tokens but are discarded). The function `evaluate_tokens()` evaluates the list of tokens in the order assignment, parenthesis, functions and finally standard arithmetic operators according to precedence.

```
// eval.cpp
```

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <map>
#include <cstdlib>
#include <cmath>
using namespace std;

double error(string s)
{ cerr << "Error: " << s << ", using 0." << endl; return 0.0; }

class token
{
  private:
   int is_value;
   double v;
   string t;
   static map<string,double> values;
  public:
   token() : is_value(0), v(0.0), t("") {};
   token(double d) : is_value(1), v(d),   t("") {};
   token(string s) : is_value(0), v(0.0), t(s)  {};
   double value();
   string name() { return t; }
   int isvalue() { return is_value; }
   double set(double d) { return values[t]=d; }
   int operator==(string s) { return (!is_value) && (t == s); }
   friend ostream& operator << (ostream&,token);
};

map<string,double> token::values;

double token::value()
{
  if(is_value) return v;
  char *end; double val=strtod(t.c_str(),&end);
  if(*end == '\0') return val;
  if(values.count(t)>0) return values[t];
  return error(t+" has no value assigned");
}

ostream& operator << (ostream& o,token t)
{ if(t.is_value) o << t.v; else o << t.t; return o;}
```

```
vector<token>
get_tokens(string s,string separator[],string ignore[])
{
  int i = 0, j, istoken = 0;
  vector<token> v;
  string value = "";
  while(i<s.length())
  {
  istoken = 0;
  for(j=0;ignore[j] != "" && i<s.length();j++)
   if(s.substr(i,ignore[j].length()) == ignore[j])
    i += ignore[j].length(), j = -1, istoken = 1;
  for(j=0;separator[j] != "" && !istoken;j++)
   if(s.substr(i,separator[j].length()) == separator[j])
   {
   if(value != "") { v.push_back(token(value)); value = ""; }
   v.push_back(token(separator[j]));
   i += separator[j].length();
   istoken = 1;
   }
  if(!istoken) value += s[i++];
  else if(value!="") { v.push_back(token(value)); value = ""; }
  }
  if(value != "") v.push_back(token(value));
  return v;
}

double fpow(double x,double y) { return pow(x,y); }
double fmul(double x,double y) { return x*y; }
double fdiv(double x,double y) { return x/y; }
double fadd(double x,double y) { return x+y; }
double fsub(double x,double y) { return x-y; }
double evaluate(token t);

double evaluate_tokens(vector<token> v)
{
  vector<token> v2, v3;
  int parenthesis, i, j, k;
  // unary function names and their implementation
  string fnames[] = { "cos", "sin", "tan", "exp",
                      "ln", "acos", "asin", "atan", "" };
  double (*fimpl[])(double) =  { cos, sin, tan, exp,
                                 log, acos, asin, atan };
```

```
// default left operands for binary operators
double initleft[] = { 1.0, 1.0, 0.0 };
// binary operators and their implementation
string opnames[][4] = { { "^", "" },
                        { "*", "/", "%", "" },
                        { "+", "-", "" } };
double (*opimpl[][3])(double,double) =
   { { fpow }, { fmul, fdiv, fmod }, { fadd, fsub } };

// check for the assignment statement
if(v.size()>2 && v[1] == "=") {
  for(j=2;j<v.size();j++) v2.push_back(v[j]);
  return v[0].set(evaluate_tokens(v2));
}
// evaluate parenthesis first
for(j=0;j<v.size();j++)
{
if(v[j] == ")") return error("unbalanced parenthesis");
else if(v[j] == "(")
{
for(parenthesis=1,j++;parenthesis && j<v.size();j++)
{
if(v[j] == "(") parenthesis++;
if(v[j] == ")") parenthesis--;
if(parenthesis) v3.push_back(v[j]);
}
if(parenthesis) return error("unbalanced parenthesis");
v2.push_back(token(evaluate_tokens(v3)));
v3.clear(); j--;
}
else v2.push_back(v[j]);
}
// evaluate functions
for(j=0,v.clear();j<v2.size();j++)
{
for(i=0;fnames[i]!="";i++)
if(v2[j] == fnames[i])
{
if(j+1<v2.size())
  v.push_back(token(fimpl[i](evaluate(v2[++j]))));
  else return error(fnames[i]+" without argument");
  break;
}
if(fnames[i]=="") v.push_back(v2[j]);
```

```
  }
  // evaluate operators in order of precedence
  for(k=0,v2.clear();k<3;k++,v = v2,v2.clear())
  {
  token left(initleft[k]);
  for(j=0;j<v.size();j++)
  {
  for(i=0;opnames[k][i]!="";i++)
   if(v[j] == opnames[k][i])
   {
   if(v2.size()) v2.pop_back();
   if(j+1<v.size())
   v2.push_back(token(opimpl[k][i](evaluate(left),
                                   evaluate(v[++j]))));
   else return error(opnames[k][i]+" without second argument");
   break;
   }
   if(opnames[k][i]=="") v2.push_back(v[j]);
   left = v2.back();
  }
  }
  // check that evaluation gave a single numerical result
  if(v.size() != 1)
  {
  for(j=0;j<v.size();j++)
   cerr << "token " << j+1 << " : " << v[j] << endl;
  return error("could not evaluate expression");
  }
  return v[0].value();
}

double evaluate(token t)
{ vector<token> v; v.push_back(t); return evaluate_tokens(v); }

double evaluateformula(istream &s)
{
  char c;
  string expression;
  static string ws[] = { " ", "\t", "\n", "\r", "" };
  static string separator[] = { "=", "+", "-", "*", "/",
                                "^", "(", ")", "" };
  do if((c = s.get()) != ';' && !s.eof()) expression += c;
  while (c != ';' && !s.eof());
  if(c != ';') return error("formula not terminated");
```

```
  vector<token> v = get_tokens(expression,separator,ws);
  return evaluate_tokens(v);
}

int main(void)
{
  while(!cin.eof())
  cout << " -> " << evaluateformula(cin) << endl;
  return 0;
}
```

Problem 10. The *Thue-Morse sequence* can be generated from the substitution map $0 \to 01$, $1 \to 10$. Starting from 0 the Thue-Morse sequence is generated as follows

$$0 \to 01 \to 0110 \to 01101001 \to \cdots.$$

Give a C++ implementation which generates the Thue-Morse sequence.

Solution 10. The function `thuemorse()` generates the Thue-Morse sequence by recursively applying the substitution map.

```
// thuemorse.cpp

#include <iostream>
#include <string>
using namespace std;

string thuemorse(int n)
{
  if(n==0) return string("0");
  string tm = thuemorse(n-1);
  string tm2;
  for(int i=0;i<tm.length();i++)
    if(tm[i]=='0') tm2 += "01"; else tm2 += "10";
  return tm2;
}

int main(void)
{
  for(int i=0;i<7;i++) cout << thuemorse(i) << endl;
  return 0;
}
```

Chapter 19

Computer Graphics

Problem 1. A *Lindenmayer system* (A, R, s) is defined as follows. Consider a finite set A of characters (the alphabet), a map $R : A \to A^*$ and a non-empty starting word s (initial string, axiom), an element of A^*. A^* are the words with characters from A. For each $a \in A$ the pair $(a, R(a))$ is called a rule and is written as

$$a \to b_1 b_2 \cdots b_n$$

where $R(a) = b_1 b_2 \cdots b_n \in A^*$. a is the left hand side and $b_1 b_2 \cdots b_n$ the right hand side of the rule. A Lindenmayer system describes a language L, a subset of A^*. The language is defined as follows:

- s is an element of L.
- Let w be an element of L and let $\sim w$ be the word where each character a of w has been replaced by $R(a)$. Then $\sim w$ is in L.

From the starting word $s = s_0$, the word s_1 is created by replacing all characters by their rule image (the right hand side of the corresponding rule). From s_1 the word s_2 is created, from that s_3 and so on. Call s_i the i-th generation of the starting word s. The interpretation of the words s_i of the language L can be done using a turtle. It visualizes the words of the language. A turtle is a drawing device which understands a few simple commands. Given a word of the language L each character of the word is interpreted as a command for the turtle. The word turns into a picture with the help of the turtle. A turtle has a position in the plane, a forward direction and a color. It understands the following commands: Move

Move forward a given number of units and draw a line, move forward a given number of units without drawing, turn left a given number of degrees, turn right a given number of degrees and change our color to a given color. Further, a turtle may remember its current state (position, direction and color) by pushing it onto a stack and change its state to a former one by popping it off from the stack. For each character of the alphabet A, one of these turtle commands may be defined. A character may also correspond to no command at all, causing the turtle to do nothing. A turtle command may be one of the identifiers `Move`, `Line`, `Left`, `Right`, `Push` or `Pop`. These commands cause the turtle to move without drawing, to draw a line, to turn left or right and to push or pop its current state. A color value must be a list of three numbers [r,g,b] defining new red-, green- and blue color values of the turtle. The following default rules for the turtle commands exist:

```
"F" = Line = move foreward one unit and draw a line
"f" = Move = move foreward one unit
"+" = Left = change forward direction by a left rotation
      of deg degrees
"-" = Right = change forward direction by a right rotation
      of deg degrees
"[" = Push = push current state to stack
"]" = Pop = pop current state from stack
```

These rules are used if no other rules for the turtle commands are defined. It is also necessary to specify which generation of the starting word of the system is to be plotted. Write a Java applet that implements the turtle.

Solution 1. We restrict the stack to one position. We use the the start `F` and the rule `F -> F[+F]F[-F]F`. The angle is $30° = \pi/6$.

```
// Lindenmayer.java

import java.awt.*;
import java.applet.*;

public class Lindenmayer extends Applet
{
  Point a, b;
  int lengthF = 3;   // step length
  double direction;
  double rotation = 30.0;  // rotation in grad
  Graphics g;
  Graphics2D g2;
```

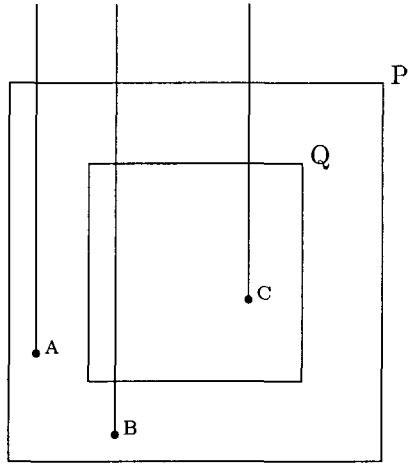
```
public void init()
{
setBackground(new Color(255,255,255));
}

public void paint(Graphics g)
{
g2 = (Graphics2D) g; // Anti-Aliasing
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                    RenderingHints.VALUE_ANTIALIAS_ON);
g2.setColor(new Color(110,170,60)); // Color
a = new Point(115,495); // starting point
direction = -80;        // starting direction
turtle(g2,"F",6);
}

public void turtle(Graphics g2,String instruc,int depth)
{
if(depth==0) return;
depth -= 1;
Point aMark = new Point(0,0);
double directionMark = 0;
char c;
for(int i=0;i<instruc.length();i++)
{
c = instruc.charAt(i);   // step forward
if(c=='F')
{
// iteration
turtle(g2,"F[+F]F[-F]F",depth);
// draw
if(depth==0)
{
double rad = 2.0*Math.PI*direction/360.0; // grad -> rad
int p = (int) (lengthF*Math.cos(rad));
int q = (int) (lengthF*Math.sin(rad));
b = new Point(a.x+p,a.y+q);
g2.drawLine(a.x,a.y,b.x,b.y);
a = b; // new starting point
}
}
// rotation left
else if(c=='+') direction += rotation;
```

```
  // rotation right
  else if(c=='-') direction -= rotation;
  // store position and direction
  else if(c=='[') { aMark = a; directionMark = direction; }
  else if(c==']') { a = aMark; direction = directionMark; }
  }
  }
}
```

Problem 2. Suppose we have a *polygon* in two dimensions. The polygon can be represented by a sequence of points

$$(x_0, y_0) \to (x_1, y_1) \to \cdots \to (x_{n-1}, y_{n-1}) \to (x_0, y_0)\,.$$

For a polygon it is not allowed that the lines connecting these points intersect. Given a point (x, y) how can we determine if the point is inside the polygon? A simple way is to draw a straight line from the point (x, y) outwards in any direction continuing to infinity. If the line intersects the polygon an odd number of times, the point is inside the polygon. For example



using a vertical line results in a simple description of an algorithm to determine whether a point is in a polygon. Thus, point A is in P but not in Q, point B is in P but not in Q and point C is in both P and Q. We could interpret Q as a "hole" in P in which case we would say C is not in P.

A *bounding box* of an edge is the rectangle determined by the endpoints of the edge.

The algorithm is as follows

1. $crossings := 0$
2. $edge :=$ first edge in polygon
3. $A :=$ point to test
4. If both endpoints of *edge* are left of A goto 9.
5. If both endpoints of *edge* are right of A goto 9.
6. If both endpoints of *edge* are below A goto 9.
7. If both endpoints of *edge* are above A and there is an endpoint on either side of A, $crossings := crossings + 1$.
8. If A is contained within the bounding box of *edge* let $(x_L, y_L) \rightarrow (x_R, y_R) := edge$, where (x_L, y_L) is the leftmost point of *edge*.

$$(x_A, y_A) := A$$

$$y_C := y_L + \frac{y_R - y_L}{x_R - x_L}(x_A - x_L).$$

If $y_C > y_A$ then

$$crossings := crossings + 1.$$

9. If edges from the polygon remain to be tested $edge :=$ next edge of polygon and goto 4.
10. If *crossings* is odd the point is in the polygon, otherwise, the point is outside the polygon.

Write a C++ program which determines whether a given point is in a given two-dimensional polygon.

Solution 2.

```
// inpolygon.cpp

#include <iostream>
using namespace std;

bool inside(double* xp,double* yp,int size,double x,double y)
{
```

```
  int j = 0;
  bool result = false;
  for(int i=0;i<size;i++)
  {
  j++;
  if(j == size) j = 0;
  if(xp[i] < x && xp[j] >= x || xp[j] < x && xp[i] >= x)
  {
  if(yp[i]+(yp[j]-yp[i])*(x-xp[i])/(xp[j]-xp[i]) < y)
  { result = !result; }
  }
  } // end for i
  return result;
}

int main(void)
{
  int size = 4;
  double* xp = new double[size];
  double* yp = new double[size];
  xp[0] = 1.0; xp[1] = -1.0; xp[2] = -1.0; xp[3] = 1.0;
  yp[0] = 1.0; yp[1] = 1.0; yp[2] = -1.0; yp[3] = -1.0;
  double x = 0.0; double y = 0.0;
  bool result = inside(xp,yp,size,x,y);
  cout << "result = " << result << endl;
  x = -0.8; y = 0.75;
  result = inside(xp,yp,size,x,y);
  cout << "result = " << result << endl;
  x = -1.1; y = 0.8;
  result = inside(xp,yp,size,x,y);
  cout << "result = " << result << endl;
  delete[] xp; delete[] yp;
  return 0;
}
```

Problem 3. Around 1890, Peano and Hilbert discovered curves that converge towards a function mapping the unit interval to the unit square. The *Hilbert curve* is constructed as follows. The construction can be defined recursively. The starting curve is the curve (three line segment generator) on the left-hand side of figure 19.1. Each curve H_j consists of four half-sized copies of H_{j-1} with a different orientation. The Hilbert curve is the limit of this construction process, i.e., H_∞. Thus, we can implement the methods `A()`, `B()`, `C()` and `D()` to draw the four copies for each step in

the construction of the Hilbert curve using recursion. Lines are drawn to connect the four copies. For example, the first three steps in constructing the Hilbert curve are given below. Write a Java program that generates the first n-steps in the construction of the Hilbert curve.

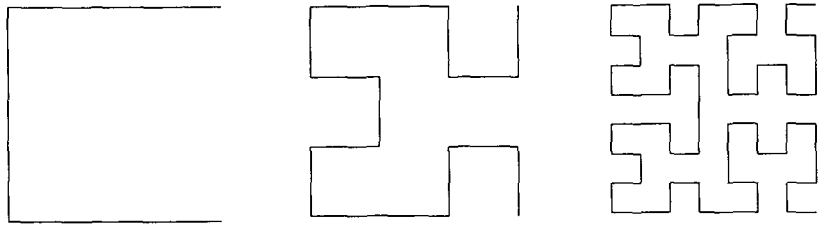


Figure 19.1: First 3 Steps in the Construction of the Hilbert Curve

Solution 3.

```
// Hilbert.java

import java.awt.*;
import java.awt.event.*;

public class Hilbert extends Frame implements ActionListener
{
  public Hilbert()
  {
  addWindowListener(new WindowAdapter()
    { public void windowClosing(WindowEvent event)
      { System.exit(0); }
      });
  drawButton.addActionListener(this);
  setTitle("Hilbert");
  Panel parameterPanel = new Panel();
  parameterPanel.setLayout(new GridLayout(2,1));
  Panel nStepsPanel = new Panel();
  nStepsPanel.add(new Label("no of steps = "));
  nStepsPanel.add(nStepsField);
  Panel buttonPanel = new Panel();
  buttonPanel.add(drawButton);
  parameterPanel.add(nStepsPanel);
  parameterPanel.add(buttonPanel);
```

```
  add("North",parameterPanel);
  add("Center",curve);
  setSize(400,400); setVisible(true);
  } // end class Hilbert

  public void windowClosing(WindowEvent event)
  { System.exit(0); }

  public static void main(String[] args) { new Hilbert(); }

  public void actionPerformed(ActionEvent action)
  {
  if(action.getSource() == drawButton)
  curve.setSteps(Integer.parseInt(nStepsField.getText()));
  System.out.println(Integer.parseInt(nStepsField.getText()));
  }

  TextField nStepsField = new TextField("5",5);
  Button drawButton = new Button("Draw");
  HilbertCurve curve = new HilbertCurve();
}

class HilbertCurve extends Canvas
{
  private int x, y, h, n, len;
  public HilbertCurve() { n = 5; }

  public void A() { if(n > 0)
  {
  Graphics g = getGraphics(); n--;
  D(); g.drawLine(x,y,x-h,y); x-=h;
  A(); g.drawLine(x,y,x,y-h); y-=h;
  A(); g.drawLine(x,y,x+h,y); x+=h;
  B(); n++; }
  }

  public void B() { if(n > 0)
  {
  Graphics g = getGraphics(); n--;
  C(); g.drawLine(x,y,x,y+h); y+=h;
  B(); g.drawLine(x,y,x+h,y); x+=h;
  B(); g.drawLine(x,y,x,y-h); y-=h;
  A(); n++; }
  }
```

```
  public void C() { if(n > 0)
  {
  Graphics g = getGraphics(); n--;
  B(); g.drawLine(x,y,x+h,y); x+=h;
  C(); g.drawLine(x,y,x,y+h); y+=h;
  C(); g.drawLine(x,y,x-h,y); x-=h;
  D(); n++; }
  }

  public void D() { if(n > 0)
  {
  Graphics g = getGraphics(); n--;
  A(); g.drawLine(x,y,x,y-h); y-=h;
  D(); g.drawLine(x,y,x-h,y); x-=h;
  D(); g.drawLine(x,y,x,y+h); y+=h;
  C(); n++; }
  } // end void D()

public void paint(Graphics g)
{
  Dimension size = getSize();
  h = 4*Math.min(size.width,size.height)/5;
  x = size.width/2 + h/2; y = size.height/2 + h/2;
  for(int i=len=1;i<n;i++) len = 2*len+1;
  h/=len; A();
}

public void setSteps(int nSteps) { n = nSteps; repaint(); }
} // end class HilbertCurve
```

Problem 4. A *Bézier curve* $\mathbf{B}(t)$ of degree n is defined by a set of control points $\mathbf{p}_i$ for $i = 0, 1, .., n$ as follows

$$\mathbf{B}(t) = \sum_{i=0}^{n} \mathbf{p}_i B_{i,n}(t)$$

where $B_{i,n}$ are the *Bernstein polynomials*

$$B_{i,n}(t) := \binom{n}{i} t^i (1-t)^{n-i}.$$

The sum of the Bernstein polynomials is a partition of unity

$$\sum_{i=0}^{n} B_{i,n}(t) = 1\,.$$

We have $B_{i,n}(t) \geq 0$ for every $0 \leq t \leq 1$. These properties imply that Bézier curves are affine invariant, and that the curve lies entirely in the convex hull of the control points defining the curve. We find that $\mathbf{B}(0) = \mathbf{p}_0$ and that $\mathbf{B}(1) = \mathbf{p}_n$. Thus, the Bézier curve interpolates the end points. If we calculate the derivative with respect to t at the endpoints, we find that

$$\frac{d\mathbf{B}}{dt}(0) = n(\mathbf{p}_1 - \mathbf{p}_0), \qquad \frac{d\mathbf{B}}{dt}(1) = n(\mathbf{p}_n - \mathbf{p}_{n-1}) .$$

Thus, the tangents at the end points of the curve are easily computed. This allows us to construct piecewise smooth C^1 or G^1 curves out of Bézier curves by setting $n_p(\mathbf{p}_1 - \mathbf{p}_0) = cn_q(\mathbf{q}_n - \mathbf{q}_{n-1})$ for a curve

$$\mathbf{Q}(t) = \sum_{i=0}^{n_q} \mathbf{q}_i B_{i,n}(t)$$

followed by a curve

$$\mathbf{P}(t) = \sum_{i=0}^{n_p} \mathbf{p}_i B_{i,n}(t) .$$

Write a Java Applet to display Bézier curves.

Solution 4. The class `Vertex` describes a point in the plane $\mathbf{R}^2$.

```
// Bezier.java

import java.awt.event.*;
import java.awt.*;
import java.applet.*;
import java.awt.Graphics;
import java.util.Vector;

class Vertex {
  public double x, y;
  public boolean selected;
  public Vertex(double x,double y) {
    this.x = x; this.y = y;
    selected = false;
    }
} // end class Vertex

class BezierPanel extends Panel implements MouseListener,
       MouseMotionListener {
  private int n = 0;
  private double delta = 0.0001;
```

```
  private Vector vertices = new Vector();
  private boolean drawcurve;
  private double a[], b[], c[], d;
  private double cost, sint;

  public BezierPanel() {
    n = 0;
      addMouseListener(this);
      addMouseMotionListener(this);
      drawcurve = true;
}

public void mouseMoved(MouseEvent e) { drawcurve = true; }
public void mouseDragged(MouseEvent e)
{
  Vertex p;
  e.consume();
  for(int k=0;k<n;k++) {
  p = (Vertex)vertices.elementAt(k);
  if(p.selected) { p.x=e.getX(); p.y=e.getY(); }
  }
  repaint();
}

public void mousePressed(MouseEvent e)
{
  Vertex p;
  e.consume();
  for(int k=0;k<n;k++) {
  p = (Vertex)vertices.elementAt(k);
  if((Math.abs(e.getX()-p.x)<3)&&(Math.abs(e.getY()-p.y)<3))
  p.selected = true; else p.selected = false; }
  drawcurve = false;
  repaint();
}

public void mouseReleased(MouseEvent e)
{ drawcurve = true; repaint(); }

public void mouseEntered(MouseEvent e) { }
public void mouseExited(MouseEvent e) { }
public void mouseClicked(MouseEvent e) {
  e.consume();
   if(e.getButton()==MouseEvent.BUTTON3) {
```

```
      n = 0;
      vertices.clear();
      } else { n++;
      vertices.add(new Vertex(e.getX(),e.getY()));
      }
      drawcurve = true;
      repaint();
}

public int comb(int t,int b) {
  int r = 1;
  for(int i=b+1;i<=t;i++) r*=i;
  for(int i=2;i<=(t-b);i++) r/=i;
  return r;
}

public double Bernstein(int i,int n,double t)
{ return comb(n,i)*Math.pow(t,i)*Math.pow(1.0-t,n-i); }

public void paint(Graphics g)
{
  double x, y, oldx, oldy, t = 0.0;
  int i, j, count;
  double num, den;
  Vertex p;
  for(i=0;i<n;i++) {
    p = (Vertex) vertices.elementAt(i);
    x = p.x; y = p.y;
    if(p.selected) g.setColor(Color.red);
    else g.setColor(Color.green);
    g.fillRect((int)x-3,(int)y-3,6,6);
  }

  g.setColor(Color.red);
  for(i=1;i<n;i++) {
  p = (Vertex) vertices.elementAt(i);
  x = p.x; y = p.y;
  p = (Vertex) vertices.elementAt(i-1);
  oldx = p.x; oldy = p.y;
  g.drawLine((int)oldx,(int)oldy,(int)x,(int)y);
  }
  if(!drawcurve) return;
  g.setColor(Color.black);
  if(n<3) return;
```

```
  oldx = oldy = 0.0;
  for(i=0;i<n;i++) {
   p = (Vertex) vertices.elementAt(i);
    oldx += Bernstein(i,n-1,t)*p.x;
    oldy += Bernstein(i,n-1,t)*p.y;
  }
  for(t=0.0;t<=1.0;t+=delta) {
  x = y = 0.0;
   for(i=0;i<n;i++) {
    p = (Vertex)vertices.elementAt(i);
    x += Bernstein(i,n-1,t)*p.x; y += Bernstein(i,n-1,t)*p.y;
    }
    g.drawLine((int)oldx,(int)oldy,(int)x,(int)y);
    oldx = x; oldy = y;
    }
  }
}

public class Bezier extends java.applet.Applet
       implements WindowListener {
  BezierPanel panel;
  static Frame f;
  public void windowActivated(WindowEvent e) {};
  public void windowClosed(WindowEvent e) {};
  public void windowClosing(WindowEvent e) {
  remove(panel);
  panel = null;
  f.dispose(); f = null;
};

public void windowDeactivated(WindowEvent e) {};
public void windowDeiconified(WindowEvent e) {};
public void windowIconified(WindowEvent e) {};
public void windowOpened(WindowEvent e) {};

public void init() {
    setLayout(new BorderLayout());
    panel = new BezierPanel();
    add("Center",panel);
}

public void destroy() { remove(panel); panel = null; }

public static void main(String args[]) {
```

```
  f = new Frame("Bezier Curves");
  Bezier h = new Bezier();
  h.init(); h.start();
  f.add("Center",h);
  f.setSize(300,300);
  f.show();
  f.addWindowListener(h);
}

public String getAppletInfo()
{ return "Draws a Bezier curve between points"; }
}
```

Problem 5. The $n \times n$ primary permutation matrix U is given by

$$U = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ 1 & 0 & 0 & \dots & 0 \end{pmatrix} . \tag{1}$$

The eigenvalues of U are given by $\lambda^0 = 1$, λ^1, λ^2, ..., λ^{n-1} with $\lambda := \exp(2\pi i/n)$. The spectral decomposition of U is

$$U = \sum_{j=0}^{n-1} \lambda^j P_j . \tag{2}$$

The projection matrix P_j can be expressed using U^k. Since $P_j P_k = 0$ for $j \neq k$ and $P_j^2 = P_j$ we find

$$U^k = \sum_{j=0}^{n-1} \lambda^{jk} P_j , \qquad k = 1, 2, \dots, n . \tag{3}$$

Thus, we calculated the Fourier transform of the projection matrices. For $k = n$ we have $U^n = I_n$ and the completeness relation

$$I_n = \sum_{j=0}^{n-1} P_j .$$

The inverse of the matrix (λ^{jk}) is given by $\frac{1}{n}(\lambda^{-jk})$. Thus

$$P_j = \frac{1}{n} \sum_{k=0}^{n-1} \lambda^{-jk} U^k , \qquad j = 0, 1, \dots, n-1 . \tag{4}$$

We apply the permutations in such a way that we can continuously interpolate between the points of a polygon. We embed the matrices U^k into a real Lie group. Thus, we have to consider the cases $n = 2m + 1$ and $n = 2m$, where m is a positive integer. In the second case we have $\lambda^0 = 1$ and $\lambda^m = -1$. Furthermore, we define $P_{-j} := P_{n-j}$. The projection matrices are therefore real. We consider first the case $n = 2m + 1$. We obtain by replacing k by nt, $t \in [0, 1]$

$$\widetilde{U}(t) = P_0 + \sum_{j=1}^{m} (e^{2\pi ijt} P_j + e^{-2\pi ijt} P_{-j}), \qquad t \in \mathbf{R}.$$

The unitary matrices $\widetilde{U}(t)$ are 1-periodic, i.e., $\widetilde{U}(t+1) = \widetilde{U}(t)$. Thus $\widetilde{U}(t+1) = \widetilde{U}(t)$ and $\widetilde{U}(k/n) = U^k$ for $k = 0, 1, \ldots, n-1$. Owing to (4), we can write $\widetilde{U}(t)$ as

$$\widetilde{U}(t) = \frac{1}{n} \sum_{k=0}^{n-1} \left(1 + \sum_{j=1}^{m} \left(e^{2\pi ij(t-k/n)} + e^{-2\pi ij(t-k/n)} \right) \right) U^k .$$

We define

$$\sigma_k(t) \equiv \sigma(t - k/n) := \frac{1}{n} \left(1 + \sum_{j=1}^{m} \left(e^{2\pi ij(t-k/n)} + e^{-2\pi ij(t-k/n)} \right) \right) .$$

Using $\exp(i\alpha) \equiv \cos(\alpha) + i \sin(\alpha)$,

$$1 + 2 \sum_{j=1}^{m} \cos(j\alpha) \equiv \frac{\sin((m + 1/2)\alpha)}{\sin(\alpha/2)}$$

and with $n = 2m + 1$ we find

$$\sigma_k(t) = \frac{\sin(\pi n(t - k/n))}{n \sin(\pi(t - k/n))}, \quad k = 0, 1, \ldots, n-1 .$$

Thus, we can write

$$\widetilde{U}(t) = \sum_{k=0}^{n-1} \sigma_k(t) U^k .$$

We have the properties

$$\sum_{k=0}^{n-1} \sigma_k(t) = 1, \qquad \sum_{k=0}^{n-1} \sigma_k^2(t) = 1 .$$

Thus, the functions $\sigma_k(t)$ and $\sigma_k^2(t)$ provide a partition of unity, and the harmonic interpolation is affine invariant. Now let $\mathbf{X} = (X_0, X_1, \ldots, X_{n-1})^T$ be the vector which describes the polygon. Then

$$\mathbf{X}(t) = \widetilde{U}(t)\mathbf{X} = \sum_{k=0}^{n-1} \sigma(t - k/n) U^k \mathbf{X}\,.$$

The curve which describes our closed smooth curve is given by

$$X_\ell(t) = \sum_{k=0}^{n-1} \sigma(t-k/n) X_{k+\ell} = \sum_{k=0}^{n-1} \sigma(t+\ell/n-(k+\ell)/n) X_{k+\ell} = X_0(t+\ell/n)$$

where $(k + \ell)$ is calculated $\bmod n$. Hence, for all ℓ it represents the same curve and we can write

$$X(t) = \sum_{k=0}^{n-1} \sigma(t - k/n) X_k, \qquad t \in [0,1]\,. \tag{5}$$

Thus, this curve interpolates the points of the given polygon smoothly. We consider now the case $n = 2m$. For this case we can write

$$U^k = P_0 + (-1)^k P_m + \sum_{j=1}^{m-1} (\lambda^{jk} P_j + \lambda^{-jk} P_{-j})\,.$$

If we replace the discrete variable k by the real variable t, we find the factor $(-1)^t \equiv e^{\pi i t}$. The other terms are real. Thus, the Lie group we would find is not real and therefore cannot be used directly for the harmonic interpolation in computer graphics. Since the function $U(t)$ will be continuous in the complex domain and go through the points of the polygon, the real part will also be continuous and go through the points of the polygon. A similar calculation as described for the case $n = 2m+1$ given above yields ($t \in [0,1]$)

$$\widetilde{U}(t) = \frac{1}{n} \sum_{k=0}^{n-1} \cos(\pi(nt - k)) U^k + \frac{i}{n} \sum_{k=0}^{n-1} \sin(\pi(nt - k)) U^k + \sum_{k=0}^{n-1} \xi_k(t) U^k$$

where

$$\xi(t) := \frac{\sin(\pi t(n-1))}{n \sin(\pi t)}, \qquad \xi_k(t) := \xi\left(t - \frac{k}{n}\right)\,.$$

Thus, our smooth curve in the even case which goes through all the points of the polygon is

$$X(t) = \sum_{k=0}^{n-1} \left(\frac{1}{n} \cos(\pi(nt - k)) + \xi_k(t) \right) X_k\,. \tag{6}$$

Write a Java applet and application to draw the harmonic interpolation of a polygon.

Solution 5. The following Java applet and application implements the harmonic interpolation. Click with the left mouse button to add points. The Java program will draw the curve resulting from harmonic interpolation. Points can also be dragged to new positions. Click with the right mouse button to remove all points.

```
// HarmonicInterpolation.java

import java.awt.event.*;
import java.awt.*;
import java.applet.*;
import java.awt.Graphics;
import java.util.Vector;

class Vertex {
  public double x, y;
  public boolean selected;
  public Vertex(double x,double y) {
     this.x = x; this.y = y;
     selected = false;
  }
}

class HarmonicPanel extends Panel implements MouseListener,
       MouseMotionListener {
  private int n = 0;
  private double delta = 0.0001;
  private Vector vertices = new Vector();
  private boolean drawcurve;
  private double a[], b[], c[], d;
  private double cost, sint;

  public HarmonicPanel() {
    n=0;
    addMouseListener(this);
    addMouseMotionListener(this);
    drawcurve=true;
  }

  public void mouseMoved(MouseEvent e) { drawcurve=true; }
  public void mouseDragged(MouseEvent e) {
```

```
    Vertex p;
    e.consume();
    for(int k=0;k<n;k++) {
       p = (Vertex)vertices.elementAt(k);
       if(p.selected) { p.x=e.getX(); p.y=e.getY(); }
       }
   repaint();
  }

public void mousePressed(MouseEvent e) {
  Vertex p;
  e.consume();
  for(int k=0;k<n;k++) {
   p = (Vertex) vertices.elementAt(k);
   if((Math.abs(e.getX()-p.x)<3) &&
      (Math.abs(e.getY()-p.y)<3))
   p.selected = true;
   else p.selected = false;
   }
   drawcurve = false;
   repaint();
}

public void mouseReleased(MouseEvent e)
{ drawcurve = true; repaint(); }

public void mouseEntered(MouseEvent e) { }
public void mouseExited(MouseEvent e) { }
public void mouseClicked(MouseEvent e) {
  e.consume();
  if(e.getButton()==MouseEvent.BUTTON3) {
  n = 0;
  vertices.clear();
  } else {
  n++;
  vertices.add(new Vertex(e.getX(),e.getY()));
  }
  drawcurve = true;
  repaint();
}

private double sigma_helper(int n,double t) {
  if(n==1) return 1;
  return Math.cos(Math.PI*(n-1)*t)+
```

```
        Math.cos(Math.PI*t)*sigma_helper(n-1,t);
}

private double sigma(double t) {
  double result;
  double test = Math.abs(t)-(int)(Math.abs(t));
  double dist1 = test;
  double dist2 = Math.abs(1.0-test);
  if((dist1<1e-7)||(dist2<1e-7)) {
  result=sigma_helper(n, t)/n;
  } else {
  result = Math.sin(Math.PI*n*(t))/(n*Math.sin(Math.PI*(t)));
  }
  if(n%2==0) return result *= Math.cos(Math.PI*t);
  return result;
}

private double sigma_k(int k,double t)
{ return sigma(t-(double)k/n); }

public void paint(Graphics g) {
  double x, y, oldx, oldy, num, den;
  int k, j, count;
  double t = 0.0;
  Vertex p;

  for(k=0;k<n;k++) {
    p = (Vertex) vertices.elementAt(k);
    x = p.x; y = p.y;
    if(p.selected) g.setColor(Color.red);
    else g.setColor(Color.green);
    g.fillRect((int)x-3,(int)y-3,6,6);
  }

  g.setColor(Color.red);
  for(k=1;k<n;k++) {
   p = (Vertex) vertices.elementAt(k);
   x = p.x; y = p.y;
   p = (Vertex)vertices.elementAt(k-1);
   oldx = p.x; oldy = p.y;
   g.drawLine((int)oldx,(int)oldy,(int)x,(int)y);
   }

   g.setColor(Color.blue);
```

```
    if(n>2) {
     p = (Vertex) vertices.elementAt(0);
     x = p.x; y = p.y;
     p = (Vertex) vertices.elementAt(n-1);
     oldx = p.x; oldy = p.y;
     g.drawLine((int)oldx,(int)oldy,(int)x,(int)y);
    }

    if(!drawcurve) return;
    g.setColor(Color.black);
    if(n<3) return;
    oldx = oldy =0.0;
    for(k=0;k<n;k++) {
    p = (Vertex) vertices.elementAt(k);
    oldx += sigma_k(k,t)*p.x; oldy += sigma_k(k,t)*p.y;
    }

    for(t=0.0;t<=1.0;t+=delta) {
      x = y = 0.0;
        for(k=0;k<n;k++) {
          p = (Vertex) vertices.elementAt(k);
          x += sigma_k(k,t)*p.x; y += sigma_k(k,t)*p.y;
    }

    g.drawLine((int)oldx,(int)oldy,(int)x,(int)y);
    oldx = x; oldy = y;
    }
    }
}

public class HarmonicInterpolation extends java.applet.Applet
            implements WindowListener {
    HarmonicPanel panel;
    static Frame f;
    public void windowActivated(WindowEvent e) {};
    public void windowClosed(WindowEvent e) {};
    public void windowClosing(WindowEvent e) {
    remove(panel); panel = null; f.dispose(); f = null;
    };

    public void windowDeactivated(WindowEvent e) {};
    public void windowDeiconified(WindowEvent e) {};
    public void windowIconified(WindowEvent e) {};
    public void windowOpened(WindowEvent e) {};
```

```
    public void init() {
          setLayout(new BorderLayout());
          panel=new HarmonicPanel();
          add("Center",panel);
    }

    public void destroy()
    { remove(panel); panel = null; }

    public static void main(String args[]) {
        f = new Frame("Harmonic Interpolation");
   HarmonicInterpolation h = new HarmonicInterpolation();
   h.init(); h.start();
   f.add("Center",h);
   f.setSize(300,300);
   f.show();
   f.addWindowListener(h);
   }

   public String getAppletInfo() {
       return "Draws a curve using harmonic interpolation";
   }
}
```

Problem 6. Let P be a non-empty finite set of planar points. A planar *Voronoi diagram* of the set P is a partition of the plane into such regions that for any element of P, a region corresponding to a unique point p contains all those points of the plane that are closer to p than to any other node of P. A unique region

$$\mathrm{vor}(p) := \{\, q \in \mathbf{R}^2 : d(p,q) < d(q,m) \text{ for all } m \in P,\ m \neq p \,\}$$

assigned to a point $p \in \mathbf{R}^2$ is called a *Voronoi cell.* A boundary of the Voronoi cell of a point p is built of segments of bisectors separating the point p and its geographically closest neighbors from the given set P. A union of all boundaries of the Voronoi cells comprises the planar Voronoi diagram

$$VD(P) := \bigcup_{p \in P} \partial \mathrm{vor}(p).$$

Voronoi diagrams are applied in computer science, statistics, geography and economics. The problem becomes simpler if we consider the mapping $*$:

$$*(x,y) = (x, y + d(q))$$

where

$$d(q) := \min_{p \in P} d(p, q)$$

and $q = (x, y)$. This mapping ensures that a region in $V*$ is encountered for the first time at a site and disappears at the intersection between two edges. This allows us to sweep a horizontal line vertically, and when we encounter an event (a site, or intersection) to modify our description of the current regions that applies for this horizontal line. Each site will be at the lowest point of the region it describes. Once we have calculated $V*$, it is trivial to construct $VD(P)$ and this can be done simultaneously. Q is a priority queue of points in the plane, ordered lexicographically. Each point is labeled as a site or intersection. L is a sequence of regions and boundaries. Note that if p is the intersection of two bisectors B_1 and B_2, then $p*$ is the intersection of B_1^* and B_2^*. R_p is $\mathrm{vor}(p)$. R_p^* is R_p under the mapping $*$. B_{pq} is the perpendicular bisector of p and q. C_{pq}^- is the part of B_{pq} to the left of p and C_{pq}^+ is the part of B_{pq} to the right of p as appropriate. The pseudocode is as follows (*Fortune's algorithm*):

1. initialize Q with all sites
2. $p \leftarrow$ extract_min(Q)
3. $L \leftarrow$ the list containing R_p^*
4. **while** Q is not empty **begin**
5. $p \leftarrow$ extract_min(Q)
6. **case**
7. p is a site:
8. Find an occurrence of a region R_q^* on L containing p.
9. Create bisector B_{pq}^*.
10. Update list L so that it contains ..., R_q^*, C_{pq}^-, R_p^*, C_{pq}^+, R_q^*, ... in place of R_q^*.
11. Insert intersections between C_{pq}^-, C_{pq}^+ with neighboring boundaries into Q.
12. p is an intersection:
13. Let p be the intersection of boundaries C_{qr} and C_{rs}.
14. Create the bisector B_{qs}^*.
15. Update list L so that it contains $C_{qs} = C_{qs}^-$ or C_{qs}^+ as appropriate, instead of C_{qr}, R_r^*, C_{rs}.

16. Delete from Q any intersection between C_{qr}, C_{rs} and their neighbors.

17. Insert any intersections between C_{qs} and its neighbors into Q.

18. Mark p as a vertex and as an endpoint of B^*_{qr}, B^*_{rs}, and B^*_{qs}.

19. **end**

Write a C++ program that finds the planar Voronoi diagram for a given set of points. The points should be read in from a file. Use metapost as the output format and for rendering the Voronoi diagrams.

Solution 6. The class `point` describes a point in $\mathbf{R}^2$.

```
// fortune.cpp

#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>
#include <cassert>
using namespace std;

const int SITE = 0, INTERSECT = 1, REGION = 0, EDGE = 1;

class point {
 public:
  double x, y;
  point() { x = y = 0.0; }
  point(double x,double y) { this->x = x; this->y = y; }
  point operator - (const point &p2) const {
     return point(x-p2.x,y-p2.y);
  }
  point operator + (const point &p2) const {
     return point(x+p2.x,y+p2.y);
  }
  // dot product
  double operator * (const point &p2) const {
    return x*p2.x+y*p2.y;
  }
  point operator * (double t) const {
     return point(x*t,y*t);
  }
  point operator / (double t) const {
```

```
    return point(x/t,y/t);
  }
};

point operator * (double t,const point& p1)
{ return p1*t; }

ostream& operator << (ostream& out,point p) {
  // get rid of scientific notation for metapost
  p.x = (int) (p.x*1e6)/1e6; p.y = (int) (p.y*1e6)/1e6;
  out << "(" << p.x << "u, " << p.y << "v)";
  return out;
}
point zero(0,0);

double dist(const point& p1,const point& p2)
{ return sqrt((p1-p2)*(p1-p2)); }

point norm(const point &p)
{ point q=p/dist(p,zero); return q; }

struct edge;
struct site_info;

typedef struct intersection {
   struct edge *e1, *e2;
   struct site_info *q, *s, *r;
} intersection;

typedef struct site_info {
  int type, name;
  point site;
  intersection intersect;
} site_info;

typedef struct edge {
  site_info *o1, *o2;
  point p1, p2;
  double t1, t2;
  struct edge *parent;
  bool marked[2];
  int count;
} edge;
```

```
typedef struct region {
  int type;
  union {
     site_info *site;
     edge *bisector;
  } data;
} region;

vector<site_info*> read_points(char *file) {
  vector<site_info*> S;
  ifstream fin(file);
  site_info *p; int i=0;
  while((!fin.eof())&&(!fin.fail())) {
    p = new site_info; p -> type=SITE;
    fin >> p->site.x>>p->site.y;
    p -> name=i++;
    if(!fin.fail()) { S.push_back(p); }
    else { delete p; }
  }
  fin.close();
  return S;
}

double ystar(site_info *p) {
  double y = p->site.y;
  if(p->type==INTERSECT) {
    y += dist(p->site,p->intersect.e1->o1->site);
  }
  return y;
}

void begin_metapost(const vector<site_info*> &S) {
  cout << "u=15cm;" << endl; cout << "v=15cm;" << endl;
  cout << "beginfig(1);" << endl;
  cout << "\tpickup pencircle scaled 4pt;" << endl;
  for(int i=0;i<S.size();i++) {
     cout << "\tdraw (" << S[i]->site.x << "u, "
          << S[i]->site.y << "v);" << endl;
  }
  cout << "\tpickup pencircle scaled 0.5pt;" << endl;
}

void end_metapost()
{ cout << "endfig;" << endl; cout << "end;" << endl; }
```

```
bool between(double t1,double t2,double t) {
   if(t1<t2) return (t1<=t)&&(t<=t2);
   else return (t2<=t)&&(t<=t1);
}

double intersect_star(edge *e,double y) {
  double x;
  double dx = e->p2.x-e->p1.x; double dy = e->p2.y-e->p1.y;
  double dpoy = (e->p1.y-e->o1->site.y);
  double dpox = (e->p1.x-e->o1->site.x);
  double a = dx*dx;
  double b = dx*dpox+dy*(y-e->o1->site.y);
  double c=dpox*dpox+dpoy*dpoy-(y-e->p1.y)*(y-e->p1.y);
  // should always intersect
  assert(b*b-a*c>=0);
  if(b*b-a*c<0.0) { return -1e15; }
  double t1 = (-b+sqrt(b*b-a*c))/a;
  double t2 = (-b-sqrt(b*b-a*c))/a;
  if(between(e->t1,e->t2,t1))
     x=e->p1.x+t1*dx;
  else x=e->p1.x+t2*dx;
  return x;
}

double gettx(edge *e,double x) {
  double t = (x-e->p1.x)/(e->p2.x-e->p1.x);
  return t;
}

double getty(edge *e,double y) {
  double t = (y-e->p1.y)/(e->p2.y-e->p1.y);
  return t;
}

double gett(edge *e,const point &p) {
  if(e->p2.y-e->p1.y>e->p2.x-e->p1.x)
    return getty(e,p.y);
  else return gettx(e,p.x);
}

double gety(edge *e,double t) {
  point q = e->p1+t*(e->p2-e->p1);
  q.y += dist(q,e->o1->site);
```

```
  return q.y;
}

point getpoint(edge *e,double t) {
  point q = e->p1+t*(e->p2-e->p1);
  return q;
}

site_info *intersect(edge *e1,edge *e2) {
  double t, tn;
  site_info *s;
  point d1 = e1->p2-e1->p1; point d2 = e2->p2-e2->p1;
  point &o1=e1->p1, &o2=e2->p1;
  point d1p=e1->o2->site-e1->o1->site;
  point d2p=e2->o2->site-e2->o1->site;
  // check for parallel
  if(fabs(d2p*d1)<1e-15) return NULL;
  t = ((o2-o1)*d2p)/(d1*d2p);
  tn = ((o1-o2)*d1p)/(d2*d1p);
  if(!between(e1->t1,e1->t2,t)) return NULL;
  if(!between(e2->t1,e2->t2,tn)) return NULL;
  s = new site_info;
  s -> type=INTERSECT;
  s -> site=o1+t*d1;
  s -> intersect.e1=e1; s -> intersect.e2=e2;
  if(e1->o2==e2->o1) {
    s->intersect.q=e1->o1;
    s->intersect.r=e1->o2;
    s->intersect.s=e2->o2;
  } else if(e1->o1==e2->o1) {
    s->intersect.q=e1->o2;
    s->intersect.r=e1->o1;
    s->intersect.s=e2->o2;
  } else if(e1->o1==e2->o2) {
    s->intersect.q=e1->o2;
    s->intersect.r=e1->o1;
    s->intersect.s=e2->o1;
  } else
  if(e1->o2==e2->o2) {
    s->intersect.q=e1->o1;
    s->intersect.r=e1->o2;
    s->intersect.s=e2->o1;
  } else {
    cout << "No common vertex! e1=";
```

```
    cout << e1->o1->name << "E" << e1->o2->name;
    cout << " e2=";
    cout << e2->o1->name << "E" << e2->o2->name << endl;
  }
  return s;
}

void replace(vector<region> &L,int k,region Rq,edge *Bpq,
             region Rp)
{
  int i=0;
  vector<region>::iterator iter;
  region r;
  iter = L.begin();
  while(i<k) { iter++; i++; }
  iter = L.erase(iter); iter = L.insert(iter,Rp);
  r.type = EDGE;
  r.data.bisector = Bpq;
  iter = L.insert(iter,r); iter = L.insert(iter,Rq);
}

void replace(vector<region> &L,int k,region Rq1,edge *Cpq_m,
             region Rp,edge *Cpq_p,region Rq2)
{
  int i=0;
  region r;
  vector<region>::iterator iter;
  iter = L.begin();
  while(i<k) { iter++; i++; }
  iter = L.erase(iter); iter = L.insert(iter,Rq2);
  r.type = EDGE;
   r.data.bisector = Cpq_p;
   iter = L.insert(iter,r); iter = L.insert(iter,Rp);
   r.type = EDGE;
   r.data.bisector = Cpq_m;
   iter = L.insert(iter,r);
   iter = L.insert(iter,Rq1);
}

void replace(vector<region> &L,int k,int l,edge *Cqs)
{
  int i=0;
  vector<region>::iterator iter;
  region r;
```

```
  r.type = EDGE;
  r.data.bisector = Cqs;
  iter = L.begin();
  while(i<k) { iter++; i++; }
  for(;i<=l;i++) { iter=L.erase(iter); }
  L.insert(iter,r);
}

int find_region(const vector<region> &L,site_info *p)
{
  for(int i=0;i<L.size(); i++) {
    if(L[i].type==EDGE) {
      double x = intersect_star(L[i].data.bisector,p->site.y);
      if(x>p->site.x) { return i-1; }
    }
  }
  return L.size()-1;
}

void remove(vector<site_info*> &Q,site_info *m) {
  vector<site_info*>::iterator iter;
  iter = Q.begin();
  while(*iter!=m) { iter++; }
  Q.erase(iter);
}

void remove_intersections(vector<site_info*> &Q,edge *e) {
  vector<site_info*>::iterator iter;
  iter = Q.begin();
  while(iter!=Q.end()) {
    if((*iter)->type==INTERSECT) {
      if((*iter)->intersect.e1 == e) {
        delete *iter;
        iter = Q.erase(iter);
        } else if((*iter)->intersect.e2==e) {
        delete *iter;
        iter=Q.erase(iter);
        } else iter++;
      } else
      iter++;
  }
}

site_info *extract_min(vector<site_info*> &Q) {
```

```
  int min = 0;
  site_info *m;
  for(int i=0;i<Q.size();i++) {
    if(ystar(Q[i])<ystar(Q[min])) min = i;
     else if(ystar(Q[i])==ystar(Q[min]))
     if(Q[i]->site.x<Q[min]->site.x) min = i;
  }
  m = Q[min];
  remove(Q,m);
  return m;
}

edge *bisector(site_info *v1,site_info *v2)
{
  edge *e; point perp;
  perp.x=v1->site.y-v2->site.y;
  perp.y=v2->site.x-v1->site.x;
  if(perp.x<0.0) { perp.x=-perp.x; perp.y=-perp.y; }
  e = new edge;
  e->parent = NULL; e->count = 0;
  e->marked[0] = false; e->marked[1] = false;
  e->o1=v1; e->o2=v2;
  e->p1=(v1->site+v2->site)/2.0-perp;
  e->p2=(v1->site+v2->site)/2.0+perp;
  e->t1=-1e7; e->t2=1e7;
  return e;
}

int get_index(const vector<region> &L,edge *e)
{
  for(int i=0;i<L.size();i++)
  if((L[i].type==EDGE)&&(L[i].data.bisector==e))
  return i;
  return -1;
}

void markend(edge *e,site_info *p)
{
  edge *ep = e;
  if(e->parent!=NULL) ep = e->parent;
  double t = gett(e,p->site);
  if(gety(e,e->t1)>gety(e,t)) {
    ep->t1=t;
    ep->marked[0]=true;
```

```
   } else { ep->t2=t; ep->marked[1]=true; }
}

void markbeg(edge *e,site_info *p)
{
  edge *ep = e;
  if(e->parent!=NULL) ep = e->parent;
  double t = gett(e,p->site);
  if(gety(e,e->t1)>gety(e, t)) {
    ep->t2 = t; ep->marked[1] = true;
   } else { ep->t1=t; ep->marked[0]=true; }
}

void clip(edge *e)
{
  point p = getpoint(e,e->t1);
  if(p.x<0.0) {e->t1=gettx(e,0.0); e->marked[0]=false;}
  if(p.x>1.0) {e->t1=gettx(e,1.0); e->marked[0]=false;}
  p=getpoint(e,e->t1);
  if(p.y<0.0) {e->t1=getty(e,0.0); e->marked[0]=false;}
  if(p.y>1.0) {e->t1=getty(e,1.0); e->marked[0]=false;}
  p=getpoint(e,e->t2);
  if(p.x<0.0) {e->t2=gettx(e,0.0); e->marked[1]=false;}
  if(p.x>1.0) {e->t2=gettx(e,1.0); e->marked[1]=false;}
  p=getpoint(e,e->t2);
  if(p.y<0.0) {e->t2=getty(e,0.0); e->marked[1]=false;}
  if(p.y>1.0) {e->t2=getty(e,1.0); e->marked[1]=false;}
  e->count=0;
  if(e->marked[0]) e->count++;
  if(e->marked[1]) e->count++;
}

void metapost_edges(const vector<edge *> &edges) {
  for(int i=0;i<edges.size();i++) {
  if(edges[i]->parent==NULL) {
   clip(edges[i]);
   if(edges[i]->t1==edges[i]->t2) continue;
   if(edges[i]->count==2)
   cout << "\tdraw "
        << getpoint(edges[i],edges[i]->t1) << "--"
        << getpoint(edges[i],edges[i]->t2) << ";" << endl;
   if(edges[i]->count==1)
   if(edges[i]->marked[0])
   cout << "\tdrawarrow "
```

```
        << getpoint(edges[i],edges[i]->t1) << "--"
        << getpoint(edges[i],edges[i]->t2) << ";" << endl;
    else
    cout << "\tdrawarrow "
        << getpoint(edges[i],edges[i]->t2) << "--"
        << getpoint(edges[i],edges[i]->t1) << ";" << endl;
    if(edges[i]->count==0) {
    cout << "\tdrawarrow "
        << getpoint(edges[i],0.5*(edges[i]->t1+edges[i]->t2))
        << "--" << getpoint(edges[i],edges[i]->t2) << ";"
        << endl;
    cout << "\tdrawarrow "
        << getpoint(edges[i],0.5*(edges[i]->t1+edges[i]->t2))
        << "--" << getpoint(edges[i],edges[i]->t1) << ";"
        << endl;
    }
    }
  }
}

void split(edge *Bpq,edge *&Cpq_m,edge *&Cpq_p,site_info *p)
{
  double t = gettx(Bpq,p->site.x);
  Cpq_m = new edge; *Cpq_m = *Bpq; Cpq_m -> count = 0;
  Cpq_p = new edge; *Cpq_p = *Bpq; Cpq_p -> count = 0;
  Cpq_m -> parent = Bpq; Cpq_p -> parent = Bpq;
  Cpq_m -> t2 = t; Cpq_p -> t1 = t;
}

int main(int argc,char *argv[]) {
  vector<site_info*> S, Q;
  vector<region> L;
  vector<edge*> edges;
  site_info *p, *s, *q;
  region r, Rp, Rq;
  edge *Bpq, *Bqs;
  edge *Cpq_p, *Cpq_m, *Cqr, *Cqs, *Crs, *Cqs_p, *Cqs_m;
  double t; int i, j;
  if(argc!=2) {
    cout << "Usage: " << argv[0] << " file.pts " << endl;
    return 0;
  }

  S = read_points(argv[1]);
```

```
begin_metapost(S);
Q = S;
p = extract_min(Q);
r.type = REGION; r.data.site=p;
L.push_back(r);
while(Q.size()>0) {
 p = extract_min(Q);
 switch(p->type) {
  case SITE:
  i = find_region(L,p);
  Rq = L[i]; q = Rq.data.site;
  Bpq = bisector(p,q);
  edges.push_back(Bpq);
  Rp.type=REGION; Rp.data.site=p;
  if(fabs(q->site.y-p->site.y)<1e-15) {
   Cpq_m = Bpq; Cpq_p = Bpq;
   } else {
   split(Bpq,Cpq_m,Cpq_p,p);
   edges.push_back(Cpq_p);
   edges.push_back(Cpq_m);
  }
  if(i-1>=0) {
  s = intersect(L[i-1].data.bisector,Cpq_m);
  if(s!=NULL) Q.push_back(s);
  }
  if(i+1<L.size()) {
  s = intersect(L[i+1].data.bisector,Cpq_p);
  if(s!=NULL) Q.push_back(s);
  }
  if(fabs(q->site.y-p->site.y)<1e-15) {
  if(p->site.x<q->site.x) replace(L,i,Rp,Bpq,Rq);
  else replace(L,i,Rq,Bpq,Rp);
  } else { replace(L,i,Rq,Cpq_m,Rp,Cpq_p,Rq); }
  break;
  case INTERSECT:
   q = p->intersect.q; s=p->intersect.s;
   Cqr = p->intersect.e1; Crs = p->intersect.e2;
   Bqs = bisector(q,s);
   edges.push_back(Bqs);
   i = get_index(L,Cqr); j = get_index(L,Crs);
   if(j<i) { int k=i; i=j; j=k; }
   if((i==-1)||(j==-1)) {
   cout << "% i=" <<i<<" j=" << j<<": Internal error"
        << endl;
```

```
      break;
      }
      site_info *h;
      if(q->site.y>s->site.y) h = q; else h=s;
      split(Bqs,Cqs_m,Cqs_p,h);
      if(fabs(q->site.y-s->site.y)<1e-15) { Cqs = Cqs_p; }
      else {
      if(p->site.x>h->site.x) Cqs=Cqs_p;
      else Cqs=Cqs_m;
      }
      if(Cqs==Cqs_m) delete Cqs_p; else delete Cqs_m;
      edges.push_back(Cqs);
      markend(Cqr,p); markend(Crs,p); markbeg(Cqs,p);
      if(j-i!=2) {
      cout << "%Internal error (>2) i=" << i << " j="<<j<<endl;
      break;
      }
      remove_intersections(Q,Cqr); remove_intersections(Q,Crs);
      replace(L,i,j,Cqs);
      if(i-2>=0) {
      s = intersect(L[i-2].data.bisector,Cqs);
      if(s!=NULL) Q.push_back(s);
      }
      if(i+2<L.size()) {
      s = intersect(L[i+2].data.bisector,Cqs);
      if(s!=NULL) Q.push_back(s);
      }
      // removed by remove_intersections;
      break;
      }
   }
   metapost_edges(edges);
   end_metapost();
   for(i=0;i<S.size();i++) { delete S[i]; }
   for(i=0;i<edges.size();i++) { delete edges[i]; }
   return 0;
}
```

Bibliography

Akenine-Möller T. and Haines E.
Real-Time Rendering, second edition
AK Peters, Natick, Massachusetts (2002)

Axelsson O.
Iterative Solution Methods
Cambridge University Press, Cambridge (1994)

Bäck T.
Evolutionary Algorithms in Theory and Practice
Oxford University Press, Oxford (1996)

Baase S.
Computer Algorithms, second edition
Addison-Wesley, Reading, Massachusetts (1988)

Bertsekas D. P.
Nonlinear Programming, second edition
Athena Scientific, Belmont, Massachusetts (1999)

Cohen D. I. A.
Introduction to Computer Theory, second edition
J. Wiley, New York (1990)

Duchateau P. and Zachmann D. W.
Partial Differential Equations, Schaum's Outline Series in Mathematics
McGraw-Hill, New York (1986)

Fortune S. J.
A Sweepline Algorithm for Voronoi Diagrams,
Algorithmica, **2**, 153–174 (1987)

Gallaher L. J.
A Multidimensional Monte Carlo Quadrature with Adaptive Stratified Sampling,
CACM, Algorithm, 440 (1971)

Gelbaum B.
Problems in Analysis
Springer-Verlag, New York (1982)

Gestlong J. L.
Mathematical Structures for Computer Science, second edition
W. H. Freeman, New York (1987)

Hairer E., Nørsett S. P. and Wanner G.
Solving Ordinary Differential Equations I, second revised edition
Springer-Verlag, Berlin (1991)

Hardy Y. and Steeb W.-H.
Classical and Quantum Computing: With C++ and Java Simulations
Birkhäuser-Verlag (2002)

JáJá J.
An Introduction to Parallel Algorithms,
Addison-Wesley Publishing, Reading (1992)

James M. L., Smith G. M. and Wolford J. C.
Applied Numerical Methods for Digital Computation, third edition
Harper and Row Publishers, New York (1985)

Koonin S. E. and Meredith D. C.
Computational Physics,
Addison-Wesley, Redwood City (1990)

Krzyz J. G.
Problems in Complex Variables Theory
Elsevier, New York (1971)

Larson L. C.
Problem Solving Through Problems
Springer-Verlag, New York (1983)

Lipschutz S.
Discrete Mathematics, Schaum's Outline Series in Mathematics
McGraw-Hill, New York (1976)

McCracken D. D.
A Second Course in Computer Science with Pascal
J. Wiley, New York (1987)

Morton K. W. and Mayers D. F.
Numerical Solution of Partial Differential Equations
Cambridge University Press, Cambridge 1994

Nakamura S.
Applied Numerical Methods in C
Prentice-Hall, Englewood Cliffs (1993)

Neapolitan R. and Naimipour K.
Foundations of Algorithms
D. C. Heath and Company, Lexington (1996)

Osyczka A.
Evolutionary Algorithms for Single and Multicriteria Design Optimization
Physica-Verlag, Heidelberg (2002)

Pearson T. W.
Introduction to Algorithms in PASCAL
John Wiley, New York (1995)

Saad Y.
Iterative Methods for Sparse Linear Systems
PWS Publishing Company, Boston (1996)

Séroul R.
Programming for Mathematicians
Springer Verlag, Berlin (1991)

Spiegel M. R.
Advanced Calculus, Schaum's Outline Series
McGraw Hill, New York (1974)

Spiegel M. R.
Finite Differences and Difference Equations, Schaum's Outline Series
McGraw Hill, New York (1971)

Steeb W.-H.
The Nonlinear Workbook: Chaos, Fractals, Cellular Automata, Neural Networks, Genetic Algorithms, Gene Expression Programming, Wavelets, Fuzzy Logic with C++, Java and SymbolicC++ Programs, second edition
World Scientific, Singapore (2002)

Thijssen J. M.
Computational Physics
Cambridge University Press, Cambridge (1999)

Tomescu I.
Problems in Combinatorics and Graph Theory
J. Wiley, New York (1985)

Van de Velde E. F.
Concurrent Scientific Computing
Springer-Verlag, New York (1994)

Wiener R. S. and Pinson L. J.
An Introduction to Object-Oriented Programming and C++
Addison-Wesley, Reading (1988)

Index