Lecture Notes on Algorithm Analysis and Computational Complexity (Fourth Edition)

Ian Parberry[1]
Department of Computer Sciences
University of North Texas

December 2001

[1]Author's address: Department of Computer Sciences, University of North Texas, P.O. Box 311366, Denton, TX 76203–1366, U.S.A. Electronic mail: ian@cs.unt.edu.

License Agreement

Preface

These lecture notes are almost exact copies of the overhead projector transparencies that I use in my CSCI 4450 course (Algorithm Analysis and Complexity Theory) at the University of North Texas. The material comes from

- textbooks on algorithm design and analysis,
- textbooks on other subjects,
- research monographs,
- papers in research journals and conferences, and
- my own knowledge and experience.

Be forewarned, this is *not* a textbook, and is not designed to be read like a textbook. To get the best use out of it you *must* attend my lectures.

Students entering this course are expected to be able to program in some procedural programming language such as C or C++, and to be able to deal with discrete mathematics. Some familiarity with basic data structures and algorithm analysis techniques is also assumed. For those students who are a little rusty, I have included some basic material on discrete mathematics and data structures, mainly at the start of the course, partially scattered throughout.

Why did I take the time to prepare these lecture notes? I have been teaching this course (or courses very much like it) at the undergraduate and graduate level since 1985. Every time I teach it I take the time to improve my notes and add new material. In Spring Semester 1992 I decided that it was time to start doing this electronically rather than, as I had done up until then, using handwritten and xerox copied notes that I transcribed onto the chalkboard during class.

This allows me to teach using slides, which have many advantages:

- They are readable, unlike my handwriting.
- I can spend more class time talking than writing.
- I can demonstrate more complicated examples.
- I can use more sophisticated graphics (there are 219 figures).

Students normally hate slides because they can never write down everything that is on them. I decided to avoid this problem by preparing these lecture notes directly from the same source files as the slides. That way you don't have to write as much as you would have if I had used the chalkboard, and so you can spend more time thinking and asking questions. You can also look over the material ahead of time.

To get the most out of this course, I recommend that you:

- Spend half an hour to an hour looking over the notes before each class.

- Attend class. If you think you understand the material without attending class, you are probably kidding yourself. Yes, I do expect you to understand the details, not just the principles.
- Spend an hour or two after each class reading the notes, the textbook, and any supplementary texts you can find.
- Attempt the ungraded exercises.
- Consult me or my teaching assistant if there is anything you don't understand.

The textbook is usually chosen by consensus of the faculty who are in the running to teach this course. Thus, it does not necessarily meet with my complete approval. Even if I were able to choose the text myself, there does not exist a single text that meets the needs of all students. I don't believe in following a text section by section since some texts do better jobs in certain areas than others. The text should therefore be viewed as being supplementary to the lecture notes, rather than vice-versa.

Algorithms Course Notes
Introduction

Ian Parberry*

Fall 2001

Summary

- What is "algorithm analysis"?
- What is "complexity theory"?
- What use are they?

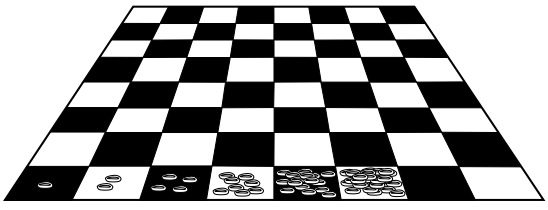
The Game of Chess

According to legend, when Grand Vizier Sissa Ben Dahir invented chess, King Shirham of India was so taken with the game that he asked him to name his reward.

The vizier asked for

- One grain of wheat on the first square of the chessboard
- Two grains of wheat on the second square
- Four grains on the third square
- Eight grains on the fourth square
- etc.

How large was his reward?



How many grains of wheat?

$$\sum_{i=0}^{63} 2^i = 2^{64} - 1 = 1.8 \times 10^{19}.$$

A bushel of wheat contains 5×10^6 grains.

Therefore he asked for 3.7×10^{12} bushels.

The price of wheat futures is around \$2.50 per bushel.

Therefore, he asked for $\$9.25 \times 10^{12} = \$92$ trillion at current prices.

The Time Travelling Investor

A time traveller invests \$1000 at 8% interest compounded annually. How much money does he/she have if he/she travels 100 years into the future? 200 years? 1000 years?

Years	*Amount*
100	$\$2.9 \times 10^6$
200	$\$4.8 \times 10^9$
300	$\$1.1 \times 10^{13}$
400	$\$2.3 \times 10^{16}$
500	$\$5.1 \times 10^{19}$
1000	$\$2.6 \times 10^{36}$

The Chinese Room

Searle (1980): Cognition cannot be the result of a formal program.

Searle's argument: a computer can compute something without really *understanding* it.

Scenario: Chinese room = person + look-up table

The Chinese room passes the Turing test, yet it has no "understanding" of Chinese.

Searle's conclusion: A symbol-processing program cannot truly understand.

Analysis of the Chinese Room

How much space would a look-up table for Chinese take?

A typical person can remember seven objects simultaneously (Miller, 1956). Any look-up table must contain queries of the form:

> "Which is the largest, a <noun>$_1$, a <noun>$_2$, a <noun>$_3$, a <noun>$_4$, a <noun>$_5$, a <noun>$_6$, or a <noun>$_7$?",

There are at least 100 commonly used nouns. Therefore there are at least $100 \cdot 99 \cdot 98 \cdot 97 \cdot 96 \cdot 95 \cdot 94 = 8 \times 10^{13}$ queries.

100 Common Nouns

aardvark	duck	lizard	sardine
ant	eagle	llama	scorpion
antelope	eel	lobster	sea lion
bear	ferret	marmoset	seahorse
beaver	finch	monkey	seal
bee	fly	mosquito	shark
beetle	fox	moth	sheep
buffalo	frog	mouse	shrimp
butterfly	gerbil	newt	skunk
cat	gibbon	octopus	slug
caterpillar	giraffe	orang-utang	snail
centipede	gnat	ostrich	snake
chicken	goat	otter	spider
chimpanzee	goose	owl	squirrel
chipmunk	gorilla	panda	starfish
cicada	guinea pig	panther	swan
cockroach	hamster	penguin	tiger
cow	horse	pig	toad
coyote	hummingbird	possum	tortoise
cricket	hyena	puma	turtle
crocodile	jaguar	rabbit	wasp
deer	jellyfish	racoon	weasel
dog	kangaroo	rat	whale
dolphin	koala	rhinocerous	wolf
donkey	lion	salamander	zebra

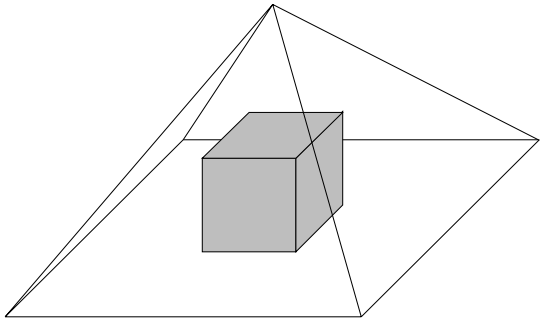
Size of the Look-up Table

The Science Citation Index:

- 215 characters per line
- 275 lines per page
- 1000 pages per inch

Our look-up table would require 1.45×10^8 inches $= 2,300$ miles of paper = a cube 200 feet on a side.

The Look-up Table and the Great Pyramid



Computerizing the Look-up Table

Use a large array of small disks. Each drive:

- Capacity 100×10^9 characters
- Volume 100 cubic inches
- Cost \$100

Therefore, 8×10^{13} queries at 100 characters per query:

- 8,000TB $= 80,000$ disk drives
- cost $\$8M$ at \$1 per GB
- volume over $55K$ cubic feet (a cube 38 feet on a side)

Extrapolating the Figures

Our queries are very simple. Suppose we use 1400 nouns (the number of concrete nouns in the Unix spell-checking dictionary), and 9 nouns per query (matches the highest human ability). The look-up table would require

- $1400^9 = 2 \times 10^{28}$ queries, 2×10^{30} bytes
- a stack of paper 10^{10} light years high [N.B. the nearest spiral galaxy (Andromeda) is 2.1×10^6 light years away, and the Universe is at most 1.5×10^{10} light years across.]
- 2×10^{19} hard drives (a cube 198 miles on a side)
- if each bit could be stored on a single hydrogen atom, 10^{31} use almost seventeen tons of hydrogen

Summary

We have seen three examples where cost increases exponentially:

- Chess: cost for an $n \times n$ chessboard grows proportionally to 2^{n^2}.
- Investor: return for n years of time travel is proportional to 1000×1.08^n (for n centuries, 1000×2200^n).
- Look-up table: cost for an n-term query is proportional to 1400^n.

Algorithm Analysis and Complexity Theory

Computational complexity theory = the study of the cost of solving interesting problems. Measure the amount of *resources* needed.

- time
- space

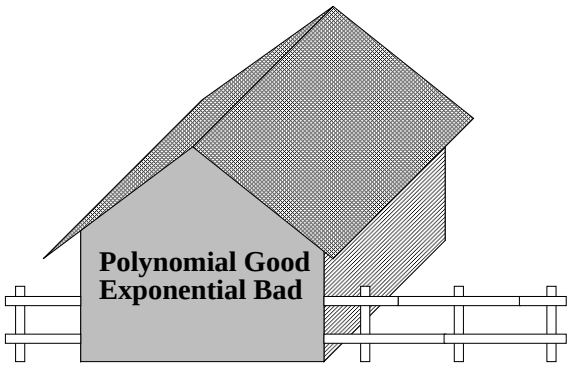
Two aspects:

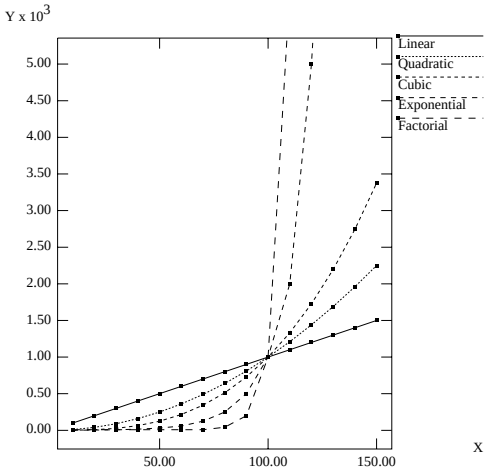
- Upper bounds: give a fast algorithm
- Lower bounds: no algorithm is faster

Algorithm analysis = analysis of resource usage of given algorithms

Exponential resource use is bad. It is best to

- Make resource usage a polynomial
- Make that polynomial as small as possible





Motivation

Why study this subject?

- Efficient algorithms lead to efficient programs.
- Efficient programs sell better.
- Efficient programs make better use of hardware.
- Programmers who write efficient programs are more marketable than those who don't!

Efficient Programs

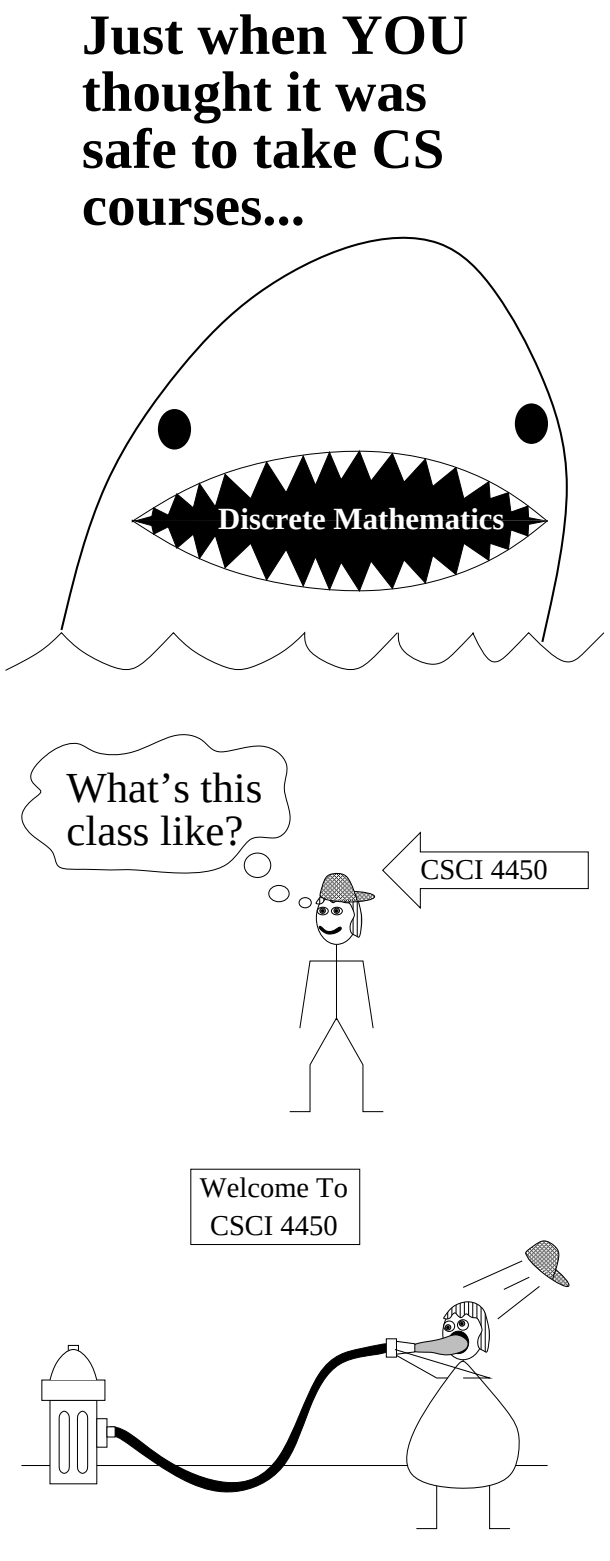
Factors influencing program efficiency

- Problem being solved
- Programming language
- Compiler
- Computer hardware
- Programmer ability
- Programmer effectiveness
- Algorithm

Objectives

What will you get from this course?

- Methods for analyzing algorithmic efficiency
- A toolbox of standard algorithmic techniques
- A toolbox of standard algorithms



Assigned Reading

CLR, Section 1.1

POA, Preface and Chapter 1.

`http://hercule.csci.unt.edu/csci4450`

Algorithms Course Notes
Mathematical Induction

Ian Parberry*

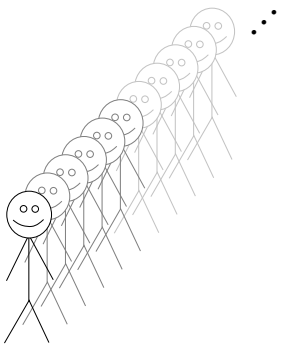
Fall 2001

Summary

Mathematical induction:

- versatile proof technique
- various forms
- application to many types of problem

Induction with People



Scenario 1:

Fact 1: The first person is Greek.

Fact 2: Pick any person in the line. If they are Greek, then the next person is Greek too.

Question: Are they all Greek?

Scenario 2:

Fact: The first person is Ukranian.

Question: Are they all Ukranian?

Scenario 3:

Fact: Pick any person in the line. If they are Nigerian, then the next person is Nigerian too.

Question: Are they all Nigerian?

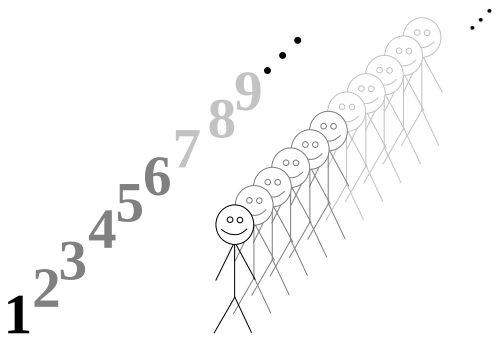
Scenario 4:

Fact 1: The first person is Indonesian.

Fact 2: Pick any person in the line. If all the people up to that point are Indonesian, then the next person is Indonesian too.

Question: Are they all Indonesian?

Mathematical Induction



To prove that a property holds for all $\mathbb{N}$, prove:

Fact 1: The property holds for 1.

Fact 2: For all $n \geq 1$, if the property holds for n, then it holds for $n+1$.

Alternatives

There are many alternative ways of doing this:

1. The property holds for 1.
2. For all $n \geq 2$, if the property holds for $n-1$, then it holds for n.

There may have to be more base cases:

1. The property holds for 1, 2, 3.
2. For all $n \geq 3$, if the property holds for n, then it holds for $n+1$.

Strong induction:

1. The property holds for 1.
2. For all $n \geq 1$, if the property holds for all $1 \leq m \leq n$, then it holds for $n+1$.

Example of Induction

An identity due to Gauss (1796, aged 9):

Claim: For all $n \in \mathbb{N}$,

$$1 + 2 + \cdots + n = n(n+1)/2.$$

First: Prove the property holds for $n = 1$.

$$1 = 1(1+1)/2$$

Second: Prove that if the property holds for n, then the property holds for $n+1$.

Let $S(n)$ denote $1 + 2 + \cdots + n$.

Assume: $S(n) = n(n+1)/2$ (the induction hypothesis).
Required to Prove:

$$S(n+1) = (n+1)(n+2)/2.$$

$$\begin{aligned} & S(n+1) \\ = \;& S(n) + (n+1) \\ = \;& n(n+1)/2 + (n+1) \quad \text{(by ind. hyp.)} \\ = \;& n^2/2 + n/2 + n + 1 \\ = \;& (n^2 + 3n + 2)/2 \\ = \;& (n+1)(n+2)/2 \end{aligned}$$

Second Example

Claim: For all $n \in \mathbb{N}$, if $1 + x > 0$, then

$$(1+x)^n \geq 1 + nx$$

First: Prove the property holds for $n = 1$.
Both sides of the equation are equal to $1 + x$.

Second: Prove that if the property holds for n, then the property holds for $n+1$.

Assume: $(1+x)^n \geq 1 + nx$.
Required to Prove:

$$(1+x)^{n+1} \geq 1 + (n+1)x.$$

$$\begin{aligned} & (1+x)^{n+1} \\ = \;& (1+x)(1+x)^n \\ \geq \;& (1+x)(1+nx) \quad \text{(by ind. hyp.)} \\ = \;& 1 + (n+1)x + nx^2 \\ \geq \;& 1 + (n+1)x \quad (\text{since } nx^2 \geq 0) \end{aligned}$$

More Complicated Example

Solve

$$S(n) = \sum_{i=1}^{n} (5i + 3)$$

This can be solved analytically, but it illustrates the technique.

Guess: $S(n) = an^2 + bn + c$ for some $a, b, c \in \mathbb{R}$.

Base: $S(1) = 8$, hence guess is true provided $a + b + c = 8$.

Inductive Step: Assume: $S(n) = an^2 + bn + c$.
Required to prove: $S(n+1) = a(n+1)^2 + b(n+1) + c$.

Now,

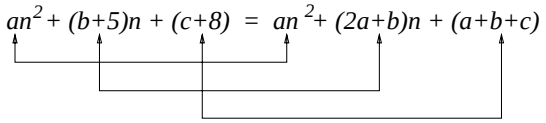
$$S(n+1) \quad = \quad S(n) + 5(n+1) + 3$$

$$\begin{aligned} &= (an^2 + bn + c) + 5(n+1) + 3 \\ &= an^2 + (b+5)n + c + 8 \end{aligned}$$

We want

$$\begin{aligned} & an^2 + (b+5)n + c + 8 \\ = \; & a(n+1)^2 + b(n+1) + c \\ = \; & an^2 + (2a+b)n + (a+b+c) \end{aligned}$$

Each pair of coefficients has to be the same.



The first coefficient tells us nothing.

The second coefficient tells us $b+5 = 2a+b$, therefore $a = 2.5$.

We know $a+b+c = 8$ (from the Base), so therefore (looking at the third coefficient), $c = 0$.

Since we now know $a = 2.5$, $c = 0$, and $a+b+c = 8$, we can deduce that $b = 5.5$.

Therefore $S(n) = 2.5n^2 + 5.5n = n(5n+11)/2$.

Complete Binary Trees

Claim: A complete binary tree with k levels has exactly $2^k - 1$ nodes.

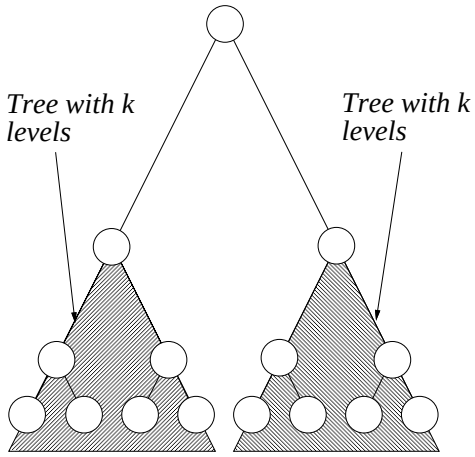
Proof: Proof by induction on number of levels. The claim is true for $k = 1$, since a complete binary tree with one level consists of a single node.

Suppose a complete binary tree with k levels has $2^k - 1$ nodes. We are required to prove that a complete binary tree with $k+1$ levels has $2^{k+1} - 1$ nodes.

A complete binary tree with $k+1$ levels consists of a root plus two trees with k levels. Therefore, by the induction hypothesis the total number of nodes is

$$1 + 2(2^k - 1) = 2^{k+1} - 1$$

as required.



Another Example

Prove that for all $n \geq 1$,

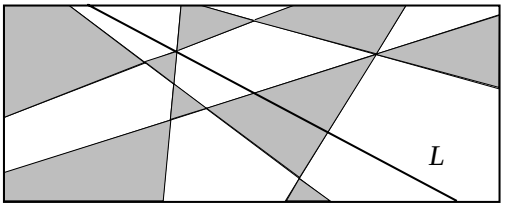
$$\sum_{i=1}^{n} 1/2^i < 1.$$

The claim is clearly true for $n = 1$. Now assume that the claim is true for n.

$$\begin{aligned} & \sum_{i=1}^{n+1} 1/2^i \\ = \; & \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^{n+1}} \\ = \; & \frac{1}{2} + \frac{1}{2}\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^n}\right) \\ = \; & \frac{1}{2} + \frac{1}{2}\sum_{i=1}^{n} 1/2^i \\ < \; & \frac{1}{2} + \frac{1}{2} \cdot 1 \quad \text{(by ind. hyp.)} \\ = \; & 1 \end{aligned}$$

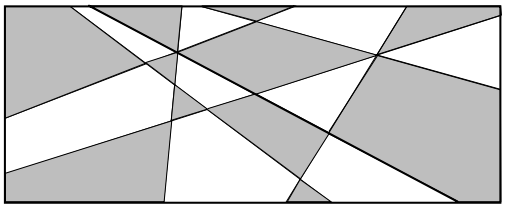
A Geometric Example

Prove that any set of regions defined by n lines in the plane can be coloured with only 2 colours so that no two regions that share an edge have the same colour.



Proof by induction on n. True for $n = 1$ (colour one side light, the other side dark). Now suppose that the hypothesis is true for n lines.

Suppose we are given $n + 1$ lines in the plane. Remove one of the lines $\mathcal{L}$, and colour the remaining regions with 2 colours (which can be done, by the induction hypothesis). Replace $\mathcal{L}$. Reverse all of the colours on one side of the line.

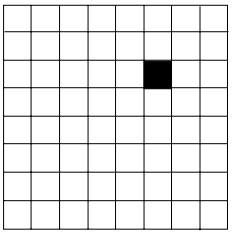


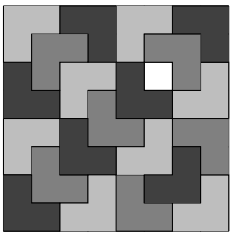
Consider two regions that have a line in common. If that line is not $\mathcal{L}$, then by the induction hypothesis, the two regions have different colours (either the same as before or reversed). If that line is $\mathcal{L}$, then the two regions formed a single region before $\mathcal{L}$ was replaced. Since we reversed colours on one side of $\mathcal{L}$ only, they now have different colours.

A Puzzle Example

A *triomino* is an L-shaped figure formed by the juxtaposition of three unit squares.

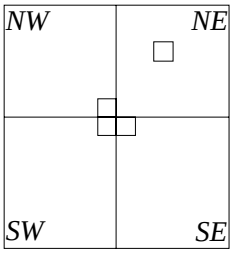
An arrangement of triominoes is a *tiling* of a shape if it covers the shape exactly without overlap. Prove by induction on $n \geq 1$ that any $2^n \times 2^n$ grid that is missing one square can be tiled with triominoes, regardless of where the missing square is.





Proof by induction on n. True for $n = 1$:

Now suppose that the hypothesis is true for n. Suppose we have a $2^{n+1} \times 2^{n+1}$ grid with one square missing.

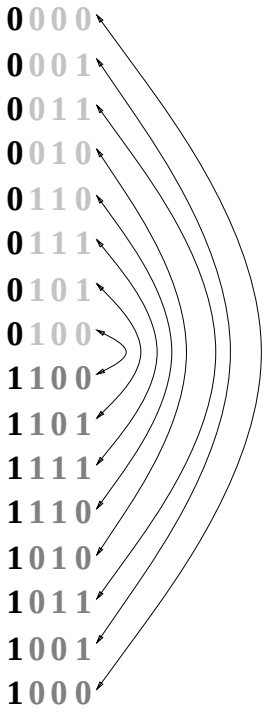


Divide the grid into four $2^n \times 2^n$ subgrids. Suppose the missing square is in the *NE* subgrid. Remove the squares closest to the center of the grid from the other three subgrids. By the induction hypothesis, all four subgrids can be tiled. The three removed squares in the *NW, SW, SE* subgrids can be tiled with a single triomino.

A Combinatorial Example

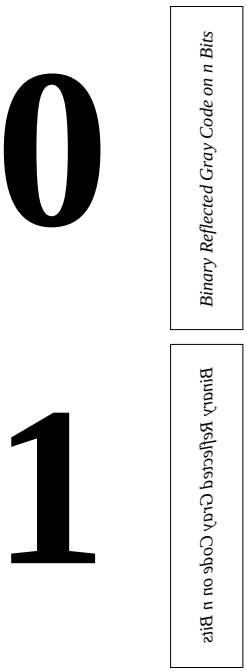
A Gray code is a sequence of 2^n n-bit binary numbers where each adjacent pair of numbers differs in exactly one bit.

$n=1$	$n=2$	$n=3$	$n=4$
0	00	000	0000
1	01	001	0001
	11	011	0011
	10	010	0010
		110	0110
		111	0111
		101	0101
		100	0100
			1100
			1101
			1111
			1110
			1010
			1011
			1001
			1000



Claim: the binary reflected Gray code on n bits is a Gray code.

Proof by induction on n. The claim is trivially true for $n = 1$. Suppose the claim is true for n bits. Suppose we construct the $n+1$ bit binary reflected Gray code as above from 2 copies of the n bit code. Now take a pair of adjacent numbers. If they are in the same half, then by the induction hypothesis they differ in exactly one bit (since they both start with the same bit). If one is in the top half and one is in the bottom half, then they only differ in the first bit.



Assigned Reading

Re-read the section in your discrete math textbook or class notes that deals with induction. Alternatively, look in the library for one of the many books on discrete mathematics.

POA, Chapter 2.

Algorithms Course Notes
Algorithm Correctness

Ian Parberry*

Fall 2001

Summary

- Confidence in algorithms from testing and correctness proof.
- Correctness of recursive algorithms proved directly by induction.
- Correctness of iterative algorithms proved using loop invariants and induction.
- Examples: Fibonacci numbers, maximum, multiplication

Correctness

How do we know that an algorithm works?

Modes of rhetoric (from ancient Greeks)

- Ethos
- Pathos
- Logos

Logical methods of checking correctness

- Testing
- Correctness proof

Testing vs. Correctness Proofs

Testing: try the algorithm on sample inputs

Correctness Proof: prove mathematically

Testing may not find obscure bugs.

Using tests alone can be dangerous.

Correctness proofs can also contain bugs: use a combination of testing and correctness proofs.

Correctness of Recursive Algorithms

To prove correctness of a recursive algorithm:

- Prove it by induction on the "size" of the problem being solved (e.g. size of array chunk, number of bits in an integer, etc.)
- Base of recursion is base of induction.
- Need to prove that recursive calls are given subproblems, that is, no infinite recursion (often trivial).
- Inductive step: assume that the recursive calls work correctly, and use this assumption to prove that the current call works correctly.

Recursive Fibonacci Numbers

Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, and for all $n \geq 2$, $F_n = F_{n-2} + F_{n-1}$.

```
    function fib(n)
        comment return F_n
1.      if n ≤ 1 then return(n)
2.      else return(fib(n − 1)+fib(n − 2))
```

Claim: For all $n \geq 0$, fib(n) returns F_n.

Base: for $n = 0$, fib(n) returns 0 as claimed. For $n = 1$, fib(n) returns 1 as claimed.

Induction: Suppose that $n \geq 2$ and for all $0 \leq m < n$, fib(m) returns F_m.

RTP fib(n) returns F_n.

What does fib(n) return?

$$\text{fib}(n-1) + \text{fib}(n-2)$$

$$
\begin{aligned}
&= F_{n-1} + F_{n-2} \quad \text{(by ind. hyp.)} \\
&= F_n.
\end{aligned}
$$

Recursive Maximum

```
function maximum(n)
  comment Return max of A[1..n].
1.  if n ≤ 1 then return(A[1]) else
2.    return(max(maximum(n − 1),A[n]))
```

Claim: For all $n \geq 1$, maximum(n) returns $\max\{A[1], A[2], \ldots, A[n]\}$. Proof by induction on $n \geq 1$.

Base: for $n = 1$, maximum(n) returns $A[1]$ as claimed.

Induction: Suppose that $n \geq 1$ and maximum(n) returns $\max\{A[1], A[2], \ldots, A[n]\}$.

RTP maximum($n+1$) returns

$$\max\{A[1], A[2], \ldots, A[n+1]\}.$$

What does maximum($n+1$) return?

$$
\begin{aligned}
& \max(\text{maximum}(n), A[n+1]) \\
= \; & \max(\max\{A[1], A[2], \ldots, A[n]\}, A[n+1]) \\
& \quad \text{(by ind. hyp.)} \\
= \; & \max\{A[1], A[2], \ldots, A[n+1]\}.
\end{aligned}
$$

Recursive Multiplication

Notation: For $x \in \mathbb{R}$, $\lfloor x \rfloor$ is the largest integer not exceeding x.

```
function multiply(y, z)
  comment return the product yz
1.  if z = 0 then return(0) else
2.  if z is odd
3.    then return(multiply(2y, ⌊z/2⌋)+y)
4.    else return(multiply(2y, ⌊z/2⌋))
```

Claim: For all $y, z \geq 0$, multiply(y, z) returns yz. Proof by induction on $z \geq 0$.

Base: for $z = 0$, multiply(y, z) returns 0 as claimed.

Induction: Suppose that for $z \geq 0$, and for all $0 \leq q \leq z$, multiply(y, q) returns yq.

RTP multiply($y, z+1$) returns $y(z+1)$.

What does multiply($y, z+1$) return?

There are two cases, depending on whether $z+1$ is odd or even.

If $z+1$ is odd, then multiply($y, z+1$) returns

$$
\begin{aligned}
& \text{multiply}(2y, \lfloor (z+1)/2 \rfloor) + y \\
= \; & 2y\lfloor (z+1)/2 \rfloor + y \quad \text{(by ind. hyp.)} \\
= \; & 2y(z/2) + y \quad \text{(since } z \text{ is even)} \\
= \; & y(z+1).
\end{aligned}
$$

If $z+1$ is even, then multiply($y, z+1$) returns

$$
\begin{aligned}
& \text{multiply}(2y, \lfloor (z+1)/2 \rfloor) \\
= \; & 2y\lfloor (z+1)/2 \rfloor \quad \text{(by ind. hyp.)} \\
= \; & 2y(z+1)/2 \quad \text{(since } z \text{ is odd)} \\
= \; & y(z+1).
\end{aligned}
$$

Correctness of Nonrecursive Algorithms

To prove correctness of an iterative algorithm:

- Analyse the algorithm one loop at a time, starting at the inner loop in case of nested loops.
- For each loop devise a *loop invariant* that remains true each time through the loop, and captures the "progress" made by the loop.
- Prove that the loop invariants hold.
- Use the loop invariants to prove that the algorithm terminates.
- Use the loop invariants to prove that the algorithm computes the correct result.

Notation

We will concentrate on one-loop algorithms.

The value in identifier x immediately after the ith iteration of the loop is denoted x_i ($i = 0$ means immediately before entering for the first time).

For example, x_6 denotes the value of identifier x after the 6th time around the loop.

Iterative Fibonacci Numbers

```
    function fib(n)
1.      comment Return F_n
2.      if n = 0 then return(0) else
3.          a := 0; b := 1; i := 2
4.          while i ≤ n do
5.              c := a + b; a := b; b := c; i := i + 1
6.      return(b)
```

Claim: fib(n) returns F_n.

Facts About the Algorithm

$$\begin{aligned} i_0 &= 2 \\ i_{j+1} &= i_j + 1 \end{aligned}$$

$$\begin{aligned} a_0 &= 0 \\ a_{j+1} &= b_j \end{aligned}$$

$$\begin{aligned} b_0 &= 1 \\ b_{j+1} &= c_{j+1} \end{aligned}$$

$$c_{j+1} = a_j + b_j$$

The Loop Invariant

For all natural numbers $j \geq 0$, $i_j = j + 2$, $a_j = F_j$, and $b_j = F_{j+1}$.

The proof is by induction on j. The base, $j = 0$, is trivial, since $i_0 = 2$, $a_0 = 0 = F_0$, and $b_0 = 1 = F_1$.

Now suppose that $j \geq 0$, $i_j = j + 2$, $a_j = F_j$ and $b_j = F_{j+1}$.

RTP $i_{j+1} = j + 3$, $a_{j+1} = F_{j+1}$ and $b_{j+1} = F_{j+2}$.

$$\begin{aligned} i_{j+1} &= i_j + 1 \\ &= (j + 2) + 1 \quad \text{(by ind. hyp.)} \\ &= j + 3 \end{aligned}$$

$$\begin{aligned} a_{j+1} &= b_j \\ &= F_{j+1} \quad \text{(by ind. hyp.)} \end{aligned}$$

$$\begin{aligned} b_{j+1} &= c_{j+1} \\ &= a_j + b_j \\ &= F_j + F_{j+1} \quad \text{(by ind. hyp.)} \\ &= F_{j+2}. \end{aligned}$$

Correctness Proof

Claim: The algorithm terminates with b containing F_n.

The claim is certainly true if $n = 0$. If $n > 0$, then we enter the while-loop.

Termination: Since $i_{j+1} = i_j + 1$, eventually i will equal $n + 1$ and the loop will terminate. Suppose this happens after t iterations. Since $i_t = n + 1$ and $i_t = t + 2$, we can conclude that $t = n - 1$.

Results: By the loop invariant, $b_t = F_{t+1} = F_n$.

Iterative Maximum

```
    function maximum(A, n)
      comment Return max of A[1..n]
1.    m := A[1]; i := 2
2.    while i ≤ n do
3.      if A[i] > m then m := A[i]
4.      i := i + 1
4.    return(m)
```

Claim: maximum(A, n) returns

$$\max\{A[1], A[2], \ldots, A[n]\}.$$

Facts About the Algorithm

$$\begin{aligned} m_0 &= A[1] \\ m_{j+1} &= \max\{m_j, A[i_j]\} \end{aligned}$$

$$\begin{aligned} i_0 &= 2 \\ i_{j+1} &= i_j + 1 \end{aligned}$$

The Loop Invariant

Claim: For all natural numbers $j \geq 0$,

$$\begin{aligned} m_j &= \max\{A[1], A[2], \ldots, A[j+1]\} \\ i_j &= j + 2 \end{aligned}$$

The proof is by induction on j. The base, $j = 0$, is trivial, since $m_0 = A[1]$ and $i_0 = 2$.

Now suppose that $j \geq 0$, $i_j = j + 2$ and

$$m_j = \max\{A[1], A[2], \ldots, A[j+1]\},$$

RTP $i_{j+1} = j + 3$ and

$$m_{j+1} = \max\{A[1], A[2], \ldots, A[j+2]\}$$

$$\begin{aligned} i_{j+1} &= i_j + 1 \\ &= (j+2) + 1 \quad \text{(by ind. hyp.)} \\ &= j + 3 \end{aligned}$$

$$\begin{aligned} & m_{j+1} \\ = \; & \max\{m_j, A[i_j]\} \\ = \; & \max\{m_j, A[j+2]\} \quad \text{(by ind. hyp.)} \\ = \; & \max\{\max\{A[1], \ldots, A[j+1]\}, A[j+2]\} \\ & \quad \text{(by ind. hyp.)} \\ = \; & \max\{A[1], A[2], \ldots, A[j+2]\}. \end{aligned}$$

Correctness Proof

Claim: The algorithm terminates with m containing the maximum value in $A[1..n]$.

Termination: Since $i_{j+1} = i_j + 1$, eventually i will equal $n+1$ and the loop will terminate. Suppose this happens after t iterations. Since $i_t = t+2$, $t = n-1$.

Results: By the loop invariant,

$$\begin{aligned} m_t &= \max\{A[1], A[2], \ldots, A[t+1]\} \\ &= \max\{A[1], A[2], \ldots, A[n]\}. \end{aligned}$$

Iterative Multiplication

```
    function multiply(y, z)
        comment Return yz, where y, z ∈ ℕ
1.      x := 0;
2.      while z > 0 do
3.          if z is odd then x := x + y;
4.          y := 2y; z := ⌊z/2⌋;
5.      return(x)
```

Claim: if $y, z \in \mathbb{N}$, then multiply(y, z) returns the value yz. That is, when line 5 is executed, $x = yz$.

A Preliminary Result

Claim: For all $n \in \mathbb{N}$,

$$2\lfloor n/2 \rfloor + (n \bmod 2) = n.$$

Case 1. n is even. Then $\lfloor n/2 \rfloor = n/2$, $n \bmod 2 = 0$, and the result follows.

Case 2. n is odd. Then $\lfloor n/2 \rfloor = (n-1)/2$, $n \bmod 2 = 1$, and the result follows.

Facts About the Algorithm

Write the changes using arithmetic instead of logic.

From line 4 of the algorithm,

$$\begin{aligned} y_{j+1} &= 2y_j \\ z_{j+1} &= \lfloor z_j/2 \rfloor \end{aligned}$$

From lines 1,3 of the algorithm,

$$\begin{aligned} x_0 &= 0 \\ x_{j+1} &= x_j + y_j(z_j \bmod 2) \end{aligned}$$

The Loop Invariant

Loop invariant: a statement about the variables that remains true every time through the loop.

Claim: For all natural numbers $j \geq 0$,

$$\boxed{y_j z_j + x_j = y_0 z_0.}$$

The proof is by induction on j. The base, $j = 0$, is trivial, since then

$$\begin{aligned} y_j z_j + x_j &= y_0 z_0 + x_0 \\ &= y_0 z_0 \end{aligned}$$

Suppose that $j \geq 0$ and

$$y_j z_j + x_j = y_0 z_0.$$

We are required to prove that

$$y_{j+1} z_{j+1} + x_{j+1} = y_0 z_0.$$

By the Facts About the Algorithm

$$\begin{aligned} & y_{j+1} z_{j+1} + x_{j+1} \\ = \; & 2y_j \lfloor z_j/2 \rfloor + x_j + y_j(z_j \bmod 2) \\ = \; & y_j(2\lfloor z_j/2 \rfloor + (z_j \bmod 2)) + x_j \\ = \; & y_j z_j + x_j \quad \text{(by prelim. result)} \\ = \; & y_0 z_0 \quad \text{(by ind. hyp.)} \end{aligned}$$

Correctness Proof

Claim: The algorithm terminates with x containing the product of y and z.

Termination: on every iteration of the loop, the value of z is halved (rounding down if it is odd). Therefore there will be some time t at which $z_t = 0$. At this point the while-loop terminates.

Results: Suppose the loop terminates after t iterations, for some $t \geq 0$. By the loop invariant,

$$y_t z_t + x_t = y_0 z_0.$$

Since $z_t = 0$, we see that $x_t = y_0 z_0$. Therefore, the algorithm terminates with x containing the product of the initial values of y and z.

Assigned Reading

Problems on Algorithms: Chapter 5.

Algorithms Course Notes
Algorithm Analysis 1

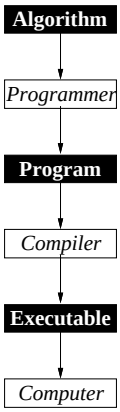
Ian Parberry*

Fall 2001

Summary

- O, Ω, Θ
- Sum and product rule for O
- Analysis of nonrecursive algorithms

Implementing Algorithms



Constant Multiples

Analyze the resource usage of an algorithm to within a constant multiple.

Why? Because other constant multiples creep in when translating from an algorithm to executable code:

- Programmer ability
- Programmer effectiveness
- Programming language
- Compiler
- Computer hardware

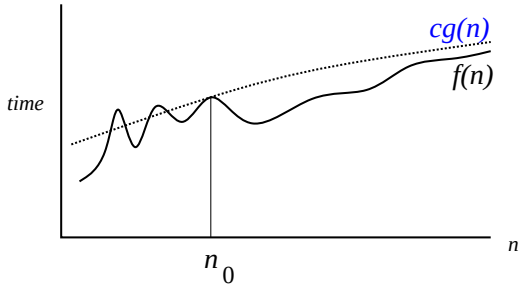
Recall: measure resource usage as a function of input size.

Big Oh

We need a notation for "within a constant multiple". Actually, we have several of them.

Informal definition: $f(n)$ is $O(g(n))$ if f grows at most as fast as g.

Formal definition: $f(n) = O(g(n))$ if there exists $c, n_0 \in \mathbb{R}^+$ such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$.



Example

Most big-Os can be proved by induction.

Example: $\log n = O(n)$.

Claim: for all $n \geq 1$, $\log n \leq n$. The proof is by induction on n. The claim is trivially true for $n = 1$, since $0 < 1$. Now suppose $n \geq 1$ and $\log n \leq n$. Then,

$$\begin{aligned} & \log(n+1) \\ \leq\ & \log 2n \\ =\ & \log n + 1 \\ \leq\ & n + 1 \quad \text{(by ind. hyp.)} \end{aligned}$$

Second Example

$2^{n+1} = O(3^n/n)$.

Claim: for all $n \geq 7$, $2^{n+1} \leq 3^n/n$. The proof is by induction on n. The claim is true for $n = 7$, since $2^{n+1} = 2^8 = 256$, and $3^n/n = 3^7/7 > 312$. Now suppose $n \geq 7$ and $2^{n+1} \leq 3^n/n$. RTP $2^{n+2} \leq 3^{n+1}/(n+1)$.

$$\begin{aligned} & 2^{n+2} \\ = \; & 2 \cdot 2^{n+1} \\ \leq \; & 2 \cdot 3^n/n \quad \text{(by ind. hyp.)} \\ \leq \; & \frac{3n}{n+1} \cdot \frac{3^n}{n} \quad \text{(see below)} \\ = \; & 3^{n+1}/(n+1). \end{aligned}$$

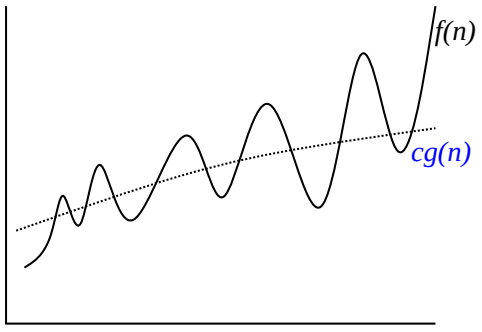
(Note that we need

$$\begin{aligned} & 3n/(n+1) \geq 2 \\ \Leftrightarrow \; & 3n \geq 2n+2 \\ \Leftrightarrow \; & n \geq 2.) \end{aligned}$$

Big Omega

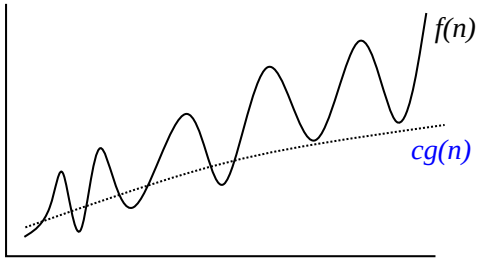
Informal definition: $f(n)$ is $\Omega(g(n))$ if f grows at least as fast as g.

Formal definition: $f(n) = \Omega(g(n))$ if there exists $c > 0$ such that there are infinitely many $n \in \mathbb{N}$ such that $f(n) \geq c \cdot g(n)$.



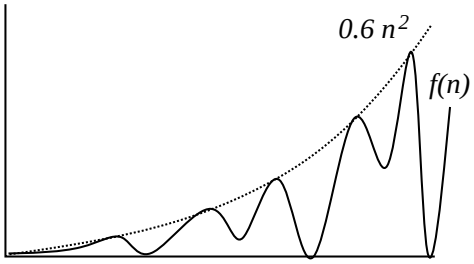
Alternative Big Omega

Some texts define Ω differently: $f(n) = \Omega'(g(n))$ if there exists $c, n_0 \in \mathbb{R}^+$ such that for all $n \geq n_0$, $f(n) \geq c \cdot g(n)$.



Is There a Difference?

If $f(n) = \Omega'(g(n))$, then $f(n) = \Omega(g(n))$, but the converse is not true. Here is an example where $f(n) = \Omega(n^2)$, but $f(n) \neq \Omega'(n^2)$.

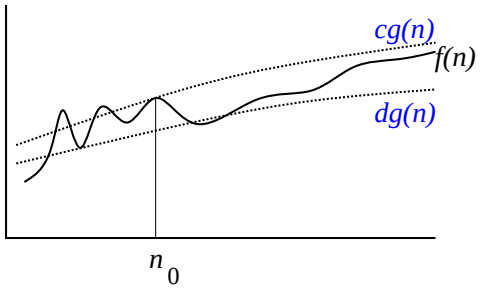


Does this come up often in practice? No.

Big Theta

Informal definition: $f(n)$ is $\Theta(g(n))$ if f is essentially the same as g, to within a constant multiple.

Formal definition: $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.



Multiple Guess

True or false?

- $3n^5 - 16n + 2 = O(n^5)$?
- $3n^5 - 16n + 2 = O(n)$?
- $3n^5 - 16n + 2 = O(n^{17})$?
- $3n^5 - 16n + 2 = \Omega(n^5)$?
- $3n^5 - 16n + 2 = \Omega(n)$?
- $3n^5 - 16n + 2 = \Omega(n^{17})$?
- $3n^5 - 16n + 2 = \Theta(n^5)$?
- $3n^5 - 16n + 2 = \Theta(n)$?
- $3n^5 - 16n + 2 = \Theta(n^{17})$?

Adding Big Ohs

Claim. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$.

Proof: Suppose for all $n \geq n_1$, $f_1(n) \leq c_1 \cdot g_1(n)$ and for all $n \geq n_2$, $f_2(n) \leq c_2 \cdot g_2(n)$.

Let $n_0 = \max\{n_1, n_2\}$ and $c_0 = \max\{c_1, c_2\}$. Then for all $n \geq n_0$,

$$\begin{aligned} f_1(n) + f_2(n) &\leq c_1 \cdot g_1(n) + c_2 \cdot g_2(n) \\ &\leq c_0(g_1(n) + g_2(n)). \end{aligned}$$

Claim. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$.

Proof: Suppose for all $n \geq n_1$, $f_1(n) \leq c_1 \cdot g_1(n)$ and for all $n \geq n_2$, $f_2(n) \leq c_2 \cdot g_2(n)$.

Let $n_0 = \max\{n_1, n_2\}$ and $c_0 = c_1 + c_2$. Then for all $n \geq n_0$,

$$\begin{aligned} f_1(n) + f_2(n) &\leq c_1 \cdot g_1(n) + c_2 \cdot g_2(n) \\ &\leq (c_1 + c_2)(\max\{g_1(n), g_2(n)\}) \\ &= c_0(\max\{g_1(n), g_2(n)\}). \end{aligned}$$

Multiplying Big Ohs

Claim. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.

Proof: Suppose for all $n \geq n_1$, $f_1(n) \leq c_1 \cdot g_1(n)$ and for all $n \geq n_2$, $f_2(n) \leq c_2 \cdot g_2(n)$.

Let $n_0 = \max\{n_1, n_2\}$ and $c_0 = c_1 \cdot c_2$. Then for all $n \geq n_0$,

$$\begin{aligned} f_1(n) \cdot f_2(n) &\leq c_1 \cdot g_1(n) \cdot c_2 \cdot g_2(n) \\ &= c_0 \cdot g_1(n) \cdot g_2(n). \end{aligned}$$

Types of Analysis

Worst case: time taken if the worst possible thing happens. $T(n)$ is the maximum time taken over all inputs of size n.

Average Case: The expected running time, given some probability distribution on the inputs (usually uniform). $T(n)$ is the average time taken over all inputs of size n.

Probabilistic: The expected running time for a random input. (Express the running time and the probability of getting it.)

Amortized: The running time for a series of executions, divided by the number of executions.

Example

Consider an algorithm that for all inputs of n bits takes time $2n$, and for one input of size n takes time n^n.

Worst case: $\Theta(n^n)$

Average Case:

$$\Theta(\frac{n^n + (2^n - 1)n}{2^n}) = \Theta(\frac{n^n}{2^n})$$

Probabilistic: $O(n)$ with probability $1 - 1/2^n$

Amortized: A sequence of m executions on *different* inputs takes amortized time

$$O(\frac{n^n + (m-1)n}{m}) = O(\frac{n^n}{m}).$$

Time Complexity

We'll do mostly worst-case analysis. How much time does it take to execute an algorithm in the worst case?

assignment	$O(1)$
procedure entry	$O(1)$
procedure exit	$O(1)$
if statement	time for test plus O(max of two branches)
loop	sum over all iterations of the time for each iteration

Put these together using sum rule and product rule. Exception — recursive algorithms.

Multiplication

```
    function multiply(y, z)
        comment Return yz, where y, z ∈ ℕ
1.      x := 0;
2.      while z > 0 do
3.          if z is odd then x := x + y;
4.          y := 2y; z := ⌊z/2⌋;
5.      return(x)
```

Suppose y and z have n bits.

- Procedure entry and exit cost $O(1)$ time
- Lines 3,4 cost $O(1)$ time each
- The while-loop on lines 2–4 costs $O(n)$ time (it is executed at most n times).
- Line 1 costs $O(1)$ time

Therefore, multiplication takes $O(n)$ time (by the sum and product rules).

Bubblesort

```
1.  procedure bubblesort(A[1..n])
2.      for i := 1 to n − 1 do
3.          for j := 1 to n − i do
4.              if A[j] > A[j + 1] then
5.                  Swap A[j] with A[j + 1]
```

- Procedure entry and exit costs $O(1)$ time
- Line 5 costs $O(1)$ time
- The if-statement on lines 4–5 costs $O(1)$ time
- The for-loop on lines 3–5 costs $O(n-i)$ time
- The for-loop on lines 2–5 costs $O(\sum_{i=1}^{n-1}(n-i))$ time.

$$O(\sum_{i=1}^{n-1}(n-i)) = O(n(n-1) - \sum_{i=1}^{n-1} i) = O(n^2)$$

Therefore, bubblesort takes time $O(n^2)$ in the worst case. Can show similarly that it takes time $\Omega(n^2)$, hence $\Theta(n^2)$.

Analysis Trick

Rather than go through the step-by-step method of analyzing algorithms,

- Identify the fundamental operation used in the algorithm, and observe that the running time is a constant multiple of the number of fundamental operations used. (Advantage: no need to grunge through line-by-line analysis.)
- Analyze the number of operations exactly. (Advantage: work with numbers instead of symbols.)

This often helps you stay focussed, and work faster.

Example

In the bubblesort example, the fundamental operation is the comparison done in line 4. The running time will be big-O of the number of comparisons.

- Line 4 uses 1 comparison
- The for-loop on lines 3–5 uses $n-i$ comparisons

- The for-loop on lines 2–5 uses $\sum_{i=1}^{n-1}(n-i)$ comparisons, and

$$\begin{aligned}\sum_{i=1}^{n-1}(n-i) &= n(n-1) - \sum_{i=1}^{n-1} i \\ &= n(n-1) - n(n-1)/2 \\ &= n(n-1)/2.\end{aligned}$$

Lies, Damn Lies, and Big-Os

(Apologies to Mark Twain.)

The multiplication algorithm takes time $O(n)$.

What does this mean? Watch out for

- Hidden assumptions: word model vs. bit model (addition takes time $O(1)$)
- Artistic lying: the multiplication algorithm takes time $O(n^2)$ is also true. (Robert Heinlein: There are 2 artistic ways of lying. One is to tell the truth, but not all of it.)
- The constant multiple: it may make the algorithm impractical.

Algorithms and Problems

Big-Os mean different things when applied to algorithms and problems.

- "Bubblesort runs in time $O(n^2)$." *But is it tight? Maybe I was too lazy to figure it out, or maybe it's unknown.*
- "Bubblesort runs in time $\Theta(n^2)$." *This is tight.*
- "The sorting problem takes time $O(n \log n)$." *There exists an algorithm that sorts in time $O(n \log n)$, but I don't know if there is a faster one.*
- "The sorting problem takes time $\Theta(n \log n)$." *There exists an algorithm that sorts in time $O(n \log n)$, and no algorithm can do any better.*

Assigned Reading

CLR, Chapter 1.2, 2.

POA, Chapter 3.

Algorithms Course Notes
Algorithm Analysis 2

Ian Parberry*

Fall 2001

Summary

Analysis of iterative (nonrecursive) algorithms.

The heap: an implementation of the priority queue

- Insertion in time $O(\log n)$
- Deletion of minimum in time $O(\log n)$

Heapsort

- Build a heap in time $O(n \log n)$.
- Dismantle a heap in time $O(n \log n)$.
- Worst case analysis — $O(n \log n)$.
- How to build a heap in time $O(n)$.

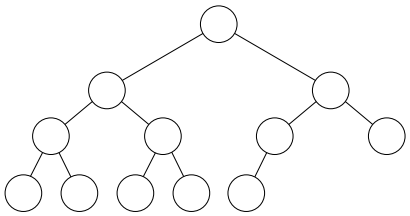
The Heap

A priority queue is a set with the operations

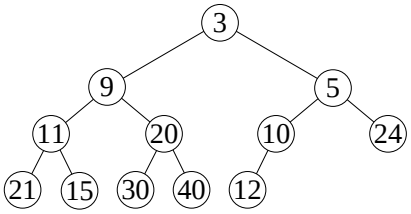
- Insert an element
- Delete and return the smallest element

A popular implementation: the heap. A heap is a binary tree with the data stored in the nodes. It has two important properties:

1. Balance. It is as much like a complete binary tree as possible. "Missing leaves", if any, are on the last level at the far right.



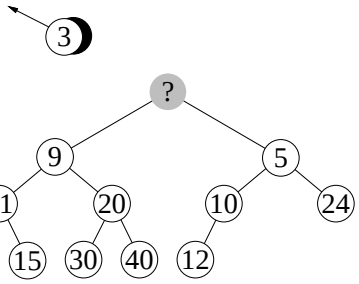
2. Structure. The value in each parent is $\leq$ the values in its children.



Note this implies that the value in each parent is $\leq$ the values in its descendants (nb. includes self).

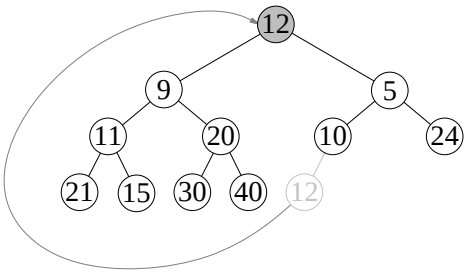
To Delete the Minimum

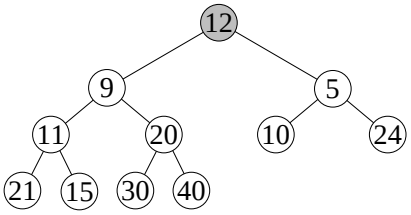
1. Remove the root and return the value in it.



But what we have is no longer a tree!

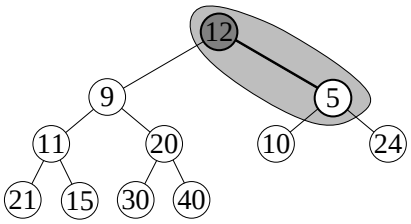
2. Replace root with last leaf.

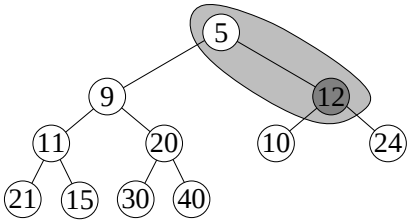


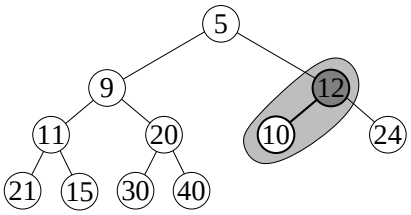


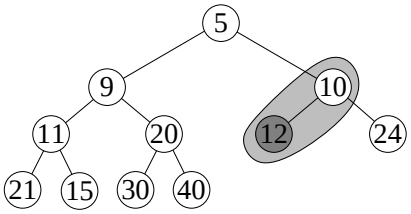
But we've violated the structure condition!

3. Repeatedly swap the new element with its smallest child until it reaches a place where it is no larger than its children.



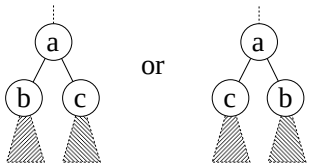




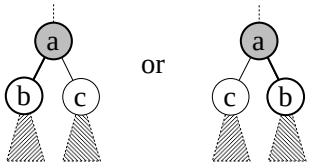


Why Does it Work?

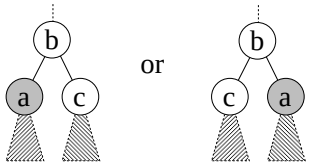
Why does swapping the new node with its smallest child work?



Suppose $b \leq c$ and a is not in the correct place. That is, either $a > b$ or $a > c$. In either case, since $b \leq c$, we know that $a > b$.



Then we get



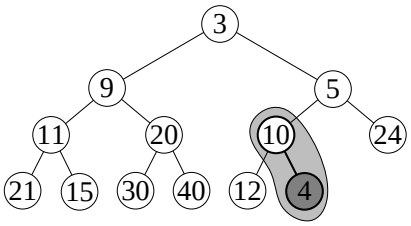
respectively.

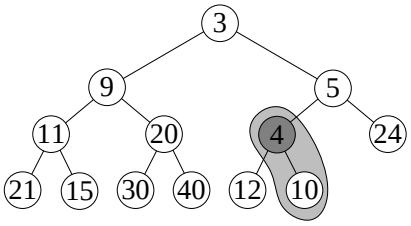
Is b smaller than its children? Yes, since $b < a$ and $b \leq c$.

Is c smaller than its children? Yes, since it was before.

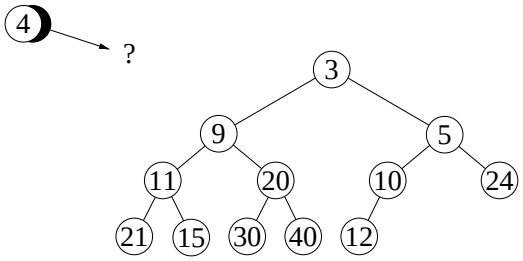
Is a smaller than its children? Not necessarily. That's why we continue to swap further down the tree.

Does the subtree of c still have the structure condition? Yes, since it is unchanged.

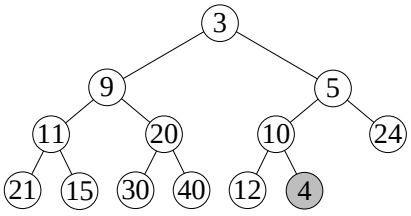




To Insert a New Element

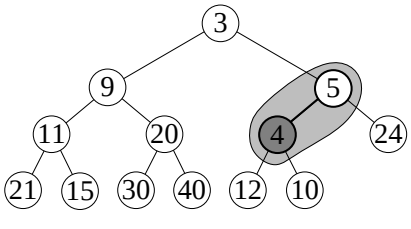


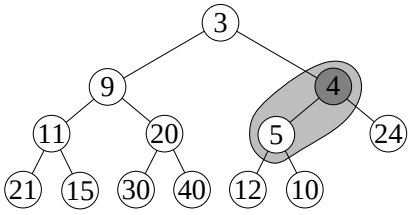
1. Put the new element in the next leaf. This preserves the balance.



But we've violated the structure condition!

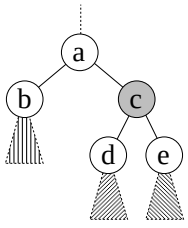
2. Repeatedly swap the new element with its parent until it reaches a place where it is no smaller than its parent.



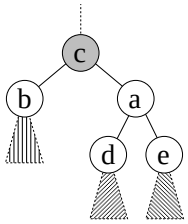


Why Does it Work?

Why does swapping the new node with its parent work?



Suppose $c < a$. Then we swap to get



Is a larger than its parent? Yes, since $a > c$.

Is b larger than its parent? Yes, since $b > a > c$.

Is c larger than its parent? Not necessarily. That's why we continue to swap

Is d larger than its parent? Yes, since d was a descendant of a in the original tree, $d > a$.

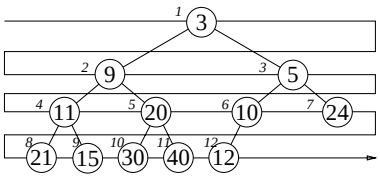
Is e larger than its parent? Yes, since e was a descendant of a in the original tree, $e > a$.

Do the subtrees of b, d, e still have the structure condition? Yes, since they are unchanged.

Implementing a Heap

An n node heap uses an array $A[1..n]$.

- The root is stored in $A[1]$
- The left child of a node in $A[i]$ is stored in node $A[2i]$.
- The right child of a node in $A[i]$ is stored in node $A[2i + 1]$.



	A
1	3
2	9
3	5
4	11
5	20
6	10
7	24
8	21
9	15
10	30
11	40
12	12

Analysis of Priority Queue Operations

Delete the Minimum:

1.	Remove root	$O(1)$
2.	Replace root	$O(1)$
3.	Swaps	$O(\ell(n))$

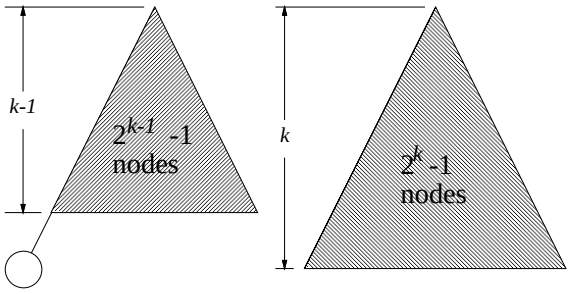
where $\ell(n)$ is the number of levels in an n-node heap.

Insert:

1.	Put in leaf	$O(1)$
2.	Swaps	$O(\ell(n))$

Analysis of $\ell(n)$

A complete binary tree with k levels has exactly $2^k - 1$ nodes (can prove by induction). Therefore, a heap with k levels has no fewer than 2^{k-1} nodes and no more than $2^k - 1$ nodes.

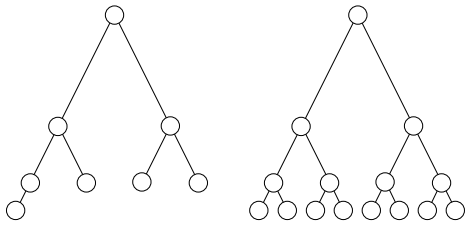


Therefore, in a heap with n nodes and k levels:

$$\begin{aligned} 2^{k-1} \leq \quad & n & \leq 2^k - 1 \\ k - 1 \leq \quad & \log n & < k \\ k - 1 \leq \quad & \lfloor \log n \rfloor & \leq k \\ k - 1 \leq \quad & \lfloor \log n \rfloor & \leq k - 1 \\ & \lfloor \log n \rfloor & = k - 1 \\ & k & = \lfloor \log n \rfloor + 1 \end{aligned}$$

Hence, number of levels is $\ell(n) = \lfloor \log n \rfloor + 1$.

Examples:



Left side: 8 nodes, $\lfloor \log 8 \rfloor + 1 = 4$ levels. Right side: 15 nodes, $\lfloor \log 15 \rfloor + 1 = 4$ levels.

So, insertion and deleting the minimum from an n-node heap requires time $O(\log n)$.

Heapsort

Algorithm:

To sort n numbers.
1. Insert n numbers into an empty heap.
2. Delete the minimum n times.
The numbers come out in ascending order.

Analysis:

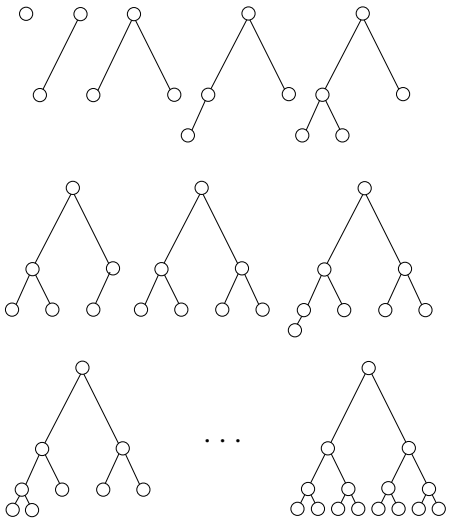
Each insertion costs time $O(\log n)$. Therefore cost of line 1 is $O(n \log n)$.

Each deletion costs time $O(\log n)$. Therefore cost of line 2 is $O(n \log n)$.

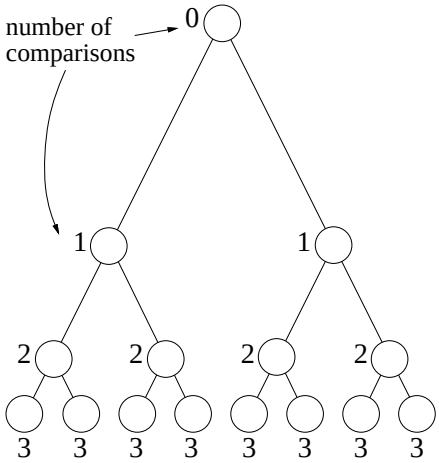
Therefore heapsort takes time $O(n \log n)$ in the worst case.

Building a Heap Top Down

Cost of building a heap proportional to number of comparisons. The above method builds from the top down.



Cost of an insertion depends on the height of the heap. There are lots of expensive nodes.



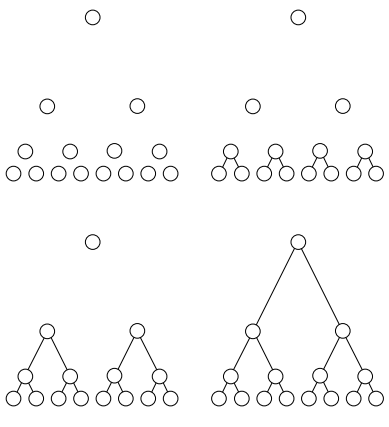
Number of comparisons (assuming a full heap):

$$\sum_{j=0}^{\ell(n)-1} j2^j = \Theta(\ell(n)2^{\ell(n)}) = \Theta(n \log n).$$

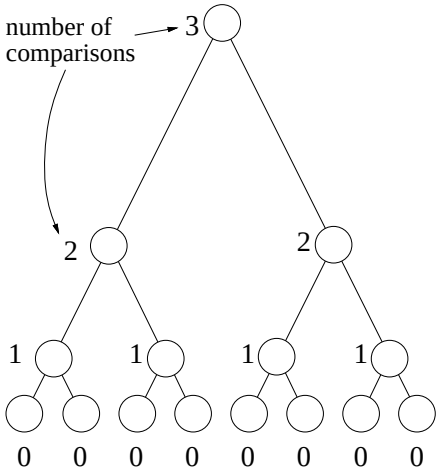
How do we know this? Can prove by induction that:

$$\begin{aligned} \sum_{j=1}^{k} j2^j &= (k-1)2^{k+1} + 2 \\ &= \Theta(k2^k) \end{aligned}$$

Building a Heap Bottom Up



Cost of an insertion depends on the height of the heap. But now there are few expensive nodes.



CLR Chapter 7.

POA Section 11.1

Number of comparisons is (assuming a full heap):

$$\sum_{i=1}^{\ell(n)} \underbrace{(i-1)}_{\text{cost}} \cdot \underbrace{2^{\ell(n)-i}}_{\text{copies}} < 2^{\ell(n)} \sum_{i=1}^{\ell(n)} i/2^i = O(n \cdot \sum_{i=1}^{\ell(n)} i/2^i).$$

What is $\sum_{i=1}^{\ell(n)} i/2^i$? It is easy to see that it is $O(1)$:

$$\begin{aligned}
\sum_{i=1}^{k} i/2^i &= \frac{1}{2^k} \sum_{i=1}^{k} i2^{k-i} \\
&= \frac{1}{2^k} \sum_{i=0}^{k-1} (k-i)2^i \\
&= \frac{k}{2^k} \sum_{i=0}^{k-1} 2^i - \frac{1}{2^k} \sum_{i=0}^{k-1} i2^i \\
&= k(2^k-1)/2^k - ((k-2)2^k+2)/2^k \\
&= k - k/2^k - k + 2 - 1/2^{k-1} \\
&= 2 - \frac{k+2}{2^k} \\
&\leq 2 \text{ for all } k \geq 1
\end{aligned}$$

Therefore building the heap takes time $O(n)$. Heapsort is still $O(n \log n)$.

Questions

Can a node be deleted in time $O(\log n)$?

Can a value be deleted in time $O(\log n)$?

Assigned Reading

Algorithms Course Notes
Algorithm Analysis 3

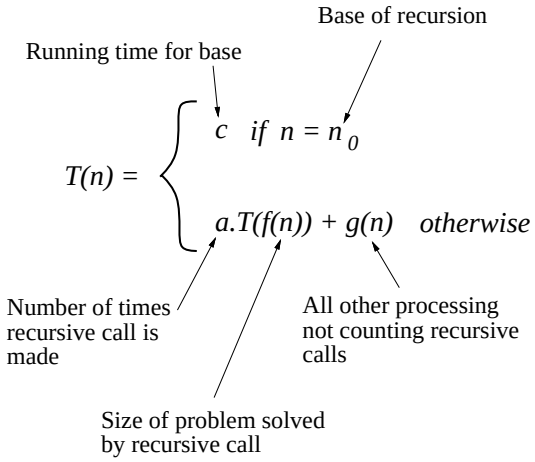
Ian Parberry*

Fall 2001

Summary

Analysis of recursive algorithms:

- recurrence relations
- how to derive them
- how to solve them



Deriving Recurrence Relations

To derive a recurrence relation for the running time of an algorithm:

- Figure out what "n", the problem size, is.
- See what value of n is used as the base of the recursion. It will usually be a single value (e.g. $n = 1$), but may be multiple values. Suppose it is n_0.
- Figure out what $T(n_0)$ is. You can usually use "some constant c", but sometimes a specific number will be needed.
- The general $T(n)$ is usually a sum of various choices of $T(m)$ (for the recursive calls), plus the sum of the other work done. Usually the recursive calls will be solving a subproblems of the same size $f(n)$, giving a term "$a \cdot T(f(n))$" in the recurrence relation.

Examples

procedure bugs(n)
  if $n = 1$ **then** do something
    else
      bugs($n - 1$);
      bugs($n - 2$);
      for $i := 1$ **to** n **do**
        something

$$T(n) = \begin{cases} \\ \\ \\ \\ \\ \end{cases}$$

```
procedure daffy(n)
  if n = 1 or n = 2 then do something
    else
      daffy(n − 1);
      for i := 1 to n do
        do something new
      daffy(n − 1);
```

$$T(n) = \left\{ \right.$$

```
procedure elmer(n)
  if n = 1 then do something
    else if n = 2 then do something else
    else
      for i := 1 to n do
        elmer(n − 1);
        do something different
```

$$T(n) = \left\{ \right.$$

```
procedure yosemite(n)
  if n = 1 then do something
    else
      for i := 1 to n − 1 do
        yosemite(i);
        do something completely different
```

$$T(n) = \left\{ \right.$$

Analysis of Multiplication

```
function multiply(y, z)
    comment return the product yz
1.  if z = 0 then return(0) else
2.  if z is odd
3.    then return(multiply(2y, ⌊z/2⌋)+y)
4.    else return(multiply(2y, ⌊z/2⌋))
```

Let $T(n)$ be the running time of multiply(y, z), where z is an n-bit natural number.

Then for some $c, d \in \mathbb{R}$,

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ T(n-1) + d & \text{otherwise} \end{cases}$$

Solving Recurrence Relations

Use repeated substitution.

Given a recurrence relation $T(n)$.

- Substitute a few times until you see a pattern
- Write a formula in terms of n and the number of substitutions i.
- Choose i so that all references to $T()$ become references to the base case.
- Solve the resulting summation

This will not always work, but works most of the time in practice.

The Multiplication Example

We know that for all $n > 1$,

$$T(n) = T(n-1) + d.$$

Therefore, for large enough n,

$$\begin{aligned} T(n) &= T(n-1) + d \\ T(n-1) &= T(n-2) + d \\ T(n-2) &= T(n-3) + d \\ &\vdots \\ T(2) &= T(1) + d \\ T(1) &= c \end{aligned}$$

Repeated Substitution

$$\begin{aligned} T(n) &= T(n-1)+d \\ &= (T(n-2)+d)+d \\ &= T(n-2)+2d \\ &= (T(n-3)+d)+2d \\ &= T(n-3)+3d \end{aligned}$$

There is a pattern developing. It looks like after i substitutions,

$$T(n) = T(n-i)+id.$$

Now choose $i = n-1$. Then

$$\begin{aligned} T(n) &= T(1)+d(n-1) \\ &= dn+c-d. \end{aligned}$$

Warning

This is not a proof. There is a gap in the logic. Where did

$$\boxed{T(n) = T(n-i)+id}$$

come from? Hand-waving!

What would make it a proof? Either

- Prove that statement by induction on i, or
- Prove the result by induction on n.

Reality Check

We claim that

$$T(n) = dn+c-d.$$

Proof by induction on n. The hypothesis is true for $n = 1$, since $d+c-d = c$.

Now suppose that the hypothesis is true for n. We are required to prove that

$$T(n+1) = dn+c.$$

Now,

$$\begin{aligned} T(n+1) &= T(n)+d \\ &= (dn+c-d)+d \quad \text{(by ind. hyp.)} \\ &= dn+c. \end{aligned}$$

Merge Sorting

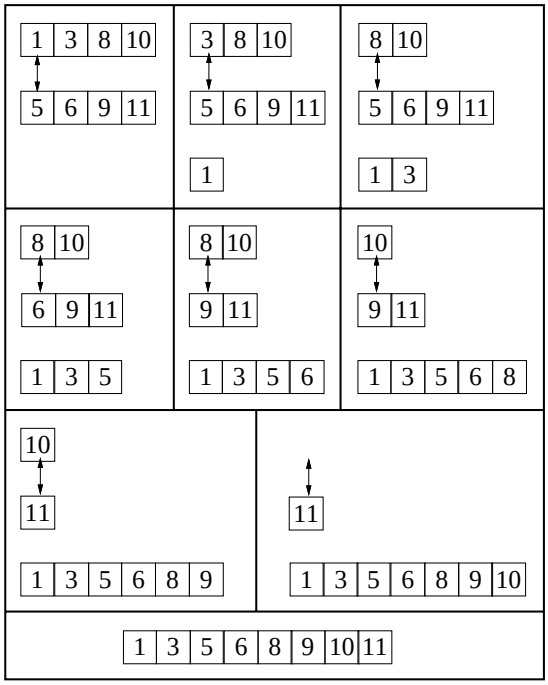
function mergesort(L, n)
  comment sorts a list L of n numbers,
    when n is a power of 2
  if $n \leq 1$ **then** return(L) **else**
    break L into 2 lists L_1, L_2 of equal size
    return(merge(mergesort($L_1, n/2$),
      mergesort($L_2, n/2$)))

Here we assume a procedure merge which can merge two sorted lists of n elements into a single sorted list in time $O(n)$.

Correctness: easy to prove by induction on n.

Analysis: Let $T(n)$ be the running time of mergesort(L, n). Then for some $c, d \in \mathbb{R}$,

$$T(n) = \begin{cases} c & \text{if } n \leq 1 \\ 2T(n/2)+dn & \text{otherwise} \end{cases}$$



$$\begin{aligned} T(n) &= 2T(n/2) + dn \\ &= 2(2T(n/4) + dn/2) + dn \\ &= 4T(n/4) + dn + dn \\ &= 4(2T(n/8) + dn/4) + dn + dn \\ &= 8T(n/8) + dn + dn + dn \end{aligned}$$

Therefore,

$$T(n) = 2^i T(n/2^i) + i \cdot dn$$

Taking $i = \log n$,

$$\begin{aligned} T(n) &= 2^{\log n} T(n/2^{\log n}) + dn \log n \\ &= dn \log n + cn \end{aligned}$$

Therefore $T(n) = O(n \log n)$.

Mergesort is better than bubblesort (in the worst case, for large enough n).

A General Theorem

Theorem: If n is a power of c, the solution to the recurrence

$$T(n) = \begin{cases} d & \text{if } n \leq 1 \\ aT(n/c) + bn & \text{otherwise} \end{cases}$$

is

$$T(n) = \begin{cases} O(n) & \text{if } a < c \\ O(n \log n) & \text{if } a = c \\ O(n^{\log_c a}) & \text{if } a > c \end{cases}$$

Examples:

- If $T(n) = 2T(n/3) + dn$, then $T(n) = O(n)$
- If $T(n) = 2T(n/2) + dn$, then $T(n) = O(n \log n)$ (mergesort)
- If $T(n) = 4T(n/2) + dn$, then $T(n) = O(n^2)$

Proof Sketch

If n is a power of c, then

$$\begin{aligned} T(n) &= a \cdot T(n/c) + bn \\ &= a(a \cdot T(n/c^2) + bn/c) + bn \\ &= a^2 \cdot T(n/c^2) + abn/c + bn \\ &= a^2(a \cdot T(n/c^3) + bn/c^2) + abn/c + bn \\ &= a^3 \cdot T(n/c^3) + a^2 bn/c^2 + abn/c + bn \\ &\vdots \\ &= a^i T(n/c^i) + bn \sum_{j=0}^{i-1} (a/c)^j \\ &= a^{\log_c n} T(1) + bn \sum_{j=0}^{\log_c n - 1} (a/c)^j \\ &= da^{\log_c n} + bn \sum_{j=0}^{\log_c n - 1} (a/c)^j \end{aligned}$$

Now,

$$a^{\log_c n} = (c^{\log_c a})^{\log_c n} = (c^{\log_c n})^{\log_c a} = n^{\log_c a}.$$

Therefore,

$$T(n) = d \cdot n^{\log_c a} + bn \sum_{j=0}^{\log_c n - 1} (a/c)^j$$

The sum is the hard part. There are three cases to consider, depending on which of a and c is biggest.

But first, we need some results on geometric progressions.

Geometric Progressions

Finite Sums: Define $S_n = \sum_{i=0}^{n} \alpha^i$. If $\alpha > 1$, then

$$\begin{aligned} \alpha \cdot S_n - S_n &= \sum_{i=0}^{n} \alpha^{i+1} - \sum_{i=0}^{n} \alpha^i \\ &= \alpha^{n+1} - 1 \end{aligned}$$

Therefore $S_n = (\alpha^{n+1} - 1)/(\alpha - 1)$.

Infinite Sums: Suppose $0 < \alpha < 1$ and let $S = \sum_{i=0}^{\infty} \alpha^i$. Then, $\alpha S = \sum_{i=1}^{\infty} \alpha^i$, and so $S - \alpha S = 1$. That is, $S = 1/(1 - \alpha)$.

Back to the Proof

Case 1: $a < c$.

$$\sum_{j=0}^{\log_c n - 1} (a/c)^j < \sum_{j=0}^{\infty} (a/c)^j = c/(c - a).$$

Therefore,

$$T(n) < d \cdot n^{\log_c a} + bcn/(c-a) = O(n).$$

(Note that since $a < c$, the first term is insignificant.)

Case 2: $a = c$. Then

$$T(n) = d \cdot n + bn \sum_{j=0}^{\log_c n - 1} 1^j = O(n \log n).$$

Case 3: $a > c$. Then $\sum_{j=0}^{\log_c n - 1} (a/c)^j$ is a geometric progression.

Hence,

$$\begin{aligned} \sum_{j=0}^{\log_c n - 1} (a/c)^j &= \frac{(a/c)^{\log_c n} - 1}{(a/c) - 1} \\ &= \frac{n^{\log_c a - 1} - 1}{(a/c) - 1} \\ &= O(n^{\log_c a - 1}) \end{aligned}$$

Therefore, $T(n) = O(n^{\log_c a})$.

Messy Details

What about when n is not a power of c?

Example: in our mergesort example, n may not be a power of 2. We can modify the algorithm easily: cut the list L into two halves of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. The recurrence relation becomes $T'(n) = c$ if $n \leq 1$, and

$$T'(n) = T'(\lfloor n/2 \rfloor) + T'(\lceil n/2 \rceil) + dn$$

otherwise.

This is much harder to analyze, but gives the same result: $T'(n) = O(n \log n)$. To see why, think of padding the input with extra numbers up to the next power of 2. You at most double the number of inputs, so the running time is

$$T(2n) = O(2n \log(2n)) = O(n \log n).$$

This is true most of the time in practice.

Examples

If $T(1) = 1$, solve the following to within a constant multiple:

- $T(n) = 2T(n/2) + 6n$
- $T(n) = 3T(n/3) + 6n - 9$
- $T(n) = 2T(n/3) + 5n$
- $T(n) = 2T(n/3) + 12n + 16$
- $T(n) = 4T(n/2) + n$
- $T(n) = 3T(n/2) + 9n$

Assigned Reading

CLR Chapter 4.

POA Chapter 4.

Re-read the section in your discrete math textbook or class notes that deals with recurrence relations. Alternatively, look in the library for one of the many books on discrete mathematics.

Algorithms Course Notes
Divide and Conquer 1

Ian Parberry*

Fall 2001

Summary

Divide and conquer and its application to

- Finding the maximum and minimum of a sequence of numbers
- Integer multiplication
- Matrix multiplication

Divide and Conquer

To solve a problem

- Divide it into smaller problems
- Solve the smaller problems
- Combine their solutions into a solution for the big problem

Example: merge sorting

- Divide the numbers into two halves
- Sort each half separately
- Merge the two sorted halves

Finding Max and Min

Problem. Find the maximum and minimum elements in an array $S[1..n]$. How many comparisons between elements of S are needed?

To find the max:

```
max:=S[1];
for i := 2 to n do
  if S[i] > max then max := S[i]
```

(The min can be found similarly).

max of n	?
min of $n-1$	?
TOTAL	?

Divide and Conquer Approach

Divide the array in half. Find the maximum and minimum in each half recursively. Return the maximum of the two maxima and the minimum of the two minima.

```
function maxmin(x, y)
  comment return max and min in S[x..y]
  if y − x ≤ 1 then
    return(max(S[x], S[y]),min(S[x], S[y]))
  else
    (max1,min1):=maxmin(x, ⌊(x + y)/2⌋)
    (max2,min2):=maxmin(⌊(x + y)/2⌋ + 1, y)
    return(max(max1,max2),min(min1,min2))
```

Correctness

The size of the problem is the number of entries in the array, $y - x + 1$.

We will prove by induction on $n = y - x + 1$ that maxmin(x, y) will return the maximum and minimum values in $S[x..y]$. The algorithm is clearly correct when $n \leq 2$. Now suppose $n > 2$, and that maxmin(x, y) will return the maximum and minimum values in $S[x..y]$ whenever $y - x + 1 < n$.

In order to apply the induction hypothesis to the first recursive call, we must prove that $\lfloor (x+y)/2 \rfloor - x + 1 < n$. There are two cases to consider, depending on whether $y - x + 1$ is even or odd.

Case 1. $y - x + 1$ is even. Then, $y - x$ is odd, and

hence $y + x$ is odd. Therefore,

$$\begin{aligned} & \lfloor \frac{x+y}{2} \rfloor - x + 1 \\ = & \frac{x+y-1}{2} - x + 1 \\ = & (y - x + 1)/2 \\ = & n/2 \\ < & n. \end{aligned}$$

Case 2. $y - x + 1$ is odd. Then $y - x$ is even, and hence $y + x$ is even. Therefore,

$$\begin{aligned} & \lfloor \frac{x+y}{2} \rfloor - x + 1 \\ = & \frac{x+y}{2} - x + 1 \\ = & (y - x + 2)/2 \\ = & (n+1)/2 \\ < & n \quad \text{(see below)}. \end{aligned}$$

(The last inequality holds since

$$(n+1)/2 < n \Leftrightarrow n > 1.)$$

To apply the ind. hyp. to the second recursive call, must prove that $y - (\lfloor (x+y)/2 \rfloor + 1) + 1 < n$. Two cases again:

Case 1. $y - x + 1$ is even.

$$\begin{aligned} & y - (\lfloor \frac{x+y}{2} \rfloor + 1) + 1 \\ = & y - \frac{x+y-1}{2} \\ = & (y - x + 1)/2 \\ = & n/2 \\ < & n. \end{aligned}$$

Case 2. $y - x + 1$ is odd.

$$\begin{aligned} & y - (\lfloor \frac{x+y}{2} \rfloor + 1) + 1 \\ = & y - \frac{x+y}{2} \\ = & (y - x + 1)/2 - 1/2 \\ = & n/2 - 1/2 \\ < & n. \end{aligned}$$

Procedure maxmin divides the array into 2 parts. By the induction hypothesis, the recursive calls correctly find the maxima and minima in these parts. Therefore, since the procedure returns the maximum of the two maxima and the minimum of the two minima, it returns the correct values.

Analysis

Let $T(n)$ be the number of comparisons made by maxmin(x, y) when $n = y - x + 1$. Suppose n is a power of 2.

What is the size of the subproblems? The first subproblem has size $\lfloor (x+y)/2 \rfloor - x + 1$. If $y - x + 1$ is a power of 2, then $y - x$ is odd, and hence $x + y$ is odd. Therefore,

$$\begin{aligned} \lfloor \frac{x+y}{2} \rfloor - x + 1 &= \frac{x+y-1}{2} - x + 1 \\ &= \frac{y-x+1}{2} = n/2. \end{aligned}$$

The second subproblem has size $y - (\lfloor (x+y)/2 \rfloor + 1) + 1$. Similarly,

$$\begin{aligned} y - (\lfloor \frac{x+y}{2} \rfloor + 1) + 1 &= y - \frac{x+y-1}{2} \\ &= \frac{y-x+1}{2} = n/2. \end{aligned}$$

So when n is a power of 2, procedure maxmin on an array chunk of size n calls itself twice on array chunks of size $n/2$. If n is a power of 2, then so is $n/2$. Therefore,

$$T(n) = \begin{cases} 1 & \text{if } n = 2 \\ 2T(n/2) + 2 & \text{otherwise} \end{cases}$$

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &= 8T(n/8) + 8 + 4 + 2 \\ &= 2^i T(n/2^i) + \sum_{j=1}^{i} 2^j \\ &= 2^{\log n - 1} T(2) + \sum_{j=1}^{\log n - 1} 2^j \end{aligned}$$

$$
\begin{aligned}
&= n/2 + (2^{\log n} - 2) \\
&= 1.5n - 2
\end{aligned}
$$

Therefore function maxmin uses only 75% as many comparisons as the naive algorithm.

Multiplication

Given positive integers y, z, compute $x = yz$. The naive multiplication algorithm:

$x := 0$;
while $z > 0$ **do**
  if $z \bmod 2 = 1$ **then** $x := x + y$;
  $y := 2y$; $z := \lfloor z/2 \rfloor$;

This can be proved correct by induction using the loop invariant $y_j z_j + x_j = y_0 z_0$.

Addition takes $O(n)$ bit operations, where n is the number of bits in y and z. The naive multiplication algorithm takes $O(n)$ n-bit additions. Therefore, the naive multiplication algorithm takes $O(n^2)$ bit operations.

Can we multiply using fewer bit operations?

Divide and Conquer Approach

Suppose n is a power of 2. Divide y and z into two halves, each with $n/2$ bits.

y	a	b
z	c	d

Then

$$
\begin{aligned}
y &= a2^{n/2} + b \\
z &= c2^{n/2} + d
\end{aligned}
$$

and so

$$
\begin{aligned}
yz &= (a2^{n/2} + b)(c2^{n/2} + d) \\
&= ac2^n + (ad + bc)2^{n/2} + bd
\end{aligned}
$$

This computes yz with 4 multiplications of $n/2$ bit numbers, and some additions and shifts. Running time given by $T(1) = c$, $T(n) = 4T(n/2) + dn$, which has solution $O(n^2)$ by the General Theorem. No gain over naive algorithm!

But $x = yz$ can also be computed as follows:

1. $u := (a + b)(c + d)$
2. $v := ac$
3. $w := bd$
4. $x := v2^n + (u - v - w)2^{n/2} + w$

Lines 2 and 3 involve a multiplication of $n/2$ bit numbers. Line 4 involves some additions and shifts. What about line 1? It has some additions and a multiplication of $(n/2 + 1)$ bit numbers. Treat the leading bits of $a + b$ and $c + d$ separately.

$a + b$	a_1	b_1
$c + d$	c_1	d_1

Then

$$
\begin{aligned}
a + b &= a_1 2^{n/2} + b_1 \\
c + d &= c_1 2^{n/2} + d_1.
\end{aligned}
$$

Therefore, the product $(a+b)(c+d)$ in line 1 can be written as

$$
\underbrace{a_1 c_1 2^n + (a_1 d_1 + b_1 c_1)2^{n/2}}_{\text{additions and shifts}} + b_1 d_1
$$

Thus to multiply n bit numbers we need

- 3 multiplications of $n/2$ bit numbers
- a constant number of additions and shifts

Therefore,

$$
T(n) = \begin{cases} c & \text{if } n = 1 \\ 3T(n/2) + dn & \text{otherwise} \end{cases}
$$

where c, d are constants.

Therefore, by our general theorem, the divide and conquer multiplication algorithm uses

$$
T(n) = O(n^{\log 3}) = O(n^{1.59})
$$

bit operations.

Matrix Multiplication

The naive matrix multiplication algorithm:

```
procedure matmultiply(X, Y, Z, n);
comment multiplies n × n matrices X := YZ
  for i := 1 to n do
    for j := 1 to n do
      X[i, j] := 0;
      for k := 1 to n do
        X[i, j] := X[i, j] + Y[i, k] * Z[k, j];
```

Assume that all integer operations take $O(1)$ time. The naive matrix multiplication algorithm then takes time $O(n^3)$. Can we do better?

Divide and Conquer Approach

Divide X, Y, Z each into four $(n/2) \times (n/2)$ matrices.

$$\begin{aligned} X &= \begin{bmatrix} I & J \\ K & L \end{bmatrix} \\ Y &= \begin{bmatrix} A & B \\ C & D \end{bmatrix} \\ Z &= \begin{bmatrix} E & F \\ G & H \end{bmatrix} \end{aligned}$$

Then

$$\begin{aligned} I &= AE + BG \\ J &= AF + BH \\ K &= CE + DG \\ L &= CF + DH \end{aligned}$$

Let $T(n)$ be the time to multiply two $n \times n$ matrices. The approach gains us nothing:

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 8T(n/2) + dn^2 & \text{otherwise} \end{cases}$$

where c, d are constants.

Therefore,

$$\begin{aligned} T(n) &= 8T(n/2) + dn^2 \\ &= 8(8T(n/4) + d(n/2)^2) + dn^2 \\ &= 8^2T(n/4) + 2dn^2 + dn^2 \\ &= 8^3T(n/8) + 4dn^2 + 2dn^2 + dn^2 \\ &= 8^iT(n/2^i) + dn^2 \sum_{j=0}^{i-1} 2^j \\ &= 8^{\log n}T(1) + dn^2 \sum_{j=0}^{\log n - 1} 2^j \\ &= cn^3 + dn^2(n-1) \\ &= O(n^3) \end{aligned}$$

Strassen's Algorithm

Compute

$$\begin{aligned} M_1 &:= (A + C)(E + F) \\ M_2 &:= (B + D)(G + H) \\ M_3 &:= (A - D)(E + H) \\ M_4 &:= A(F - H) \\ M_5 &:= (C + D)E \\ M_6 &:= (A + B)H \\ M_7 &:= D(G - E) \end{aligned}$$

Then,

$$\begin{aligned} I &:= M_2 + M_3 - M_6 - M_7 \\ J &:= M_4 + M_6 \\ K &:= M_5 + M_7 \\ L &:= M_1 - M_3 - M_4 - M_5 \end{aligned}$$

Will This Work?

$$\begin{aligned} I &:= M_2 + M_3 - M_6 - M_7 \\ &= (B + D)(G + H) + (A - D)(E + H) \\ &\quad - (A + B)H - D(G - E) \\ &= (BG + BH + DG + DH) \\ &\quad + (AE + AH - DE - DH) \\ &\quad + (-AH - BH) + (-DG + DE) \\ &= BG + AE \end{aligned}$$

$$J := M_4 + M_6$$

$$
\begin{aligned}
&= A(F-H) + (A+B)H \\
&= AF - AH + AH + BH \\
&= AF + BH
\end{aligned}
$$

$$
\begin{aligned}
K &:= M_5 + M_7 \\
&= (C+D)E + D(G-E) \\
&= CE + DE + DG - DE \\
&= CE + DG
\end{aligned}
$$

$$
\begin{aligned}
L &:= M_1 - M_3 - M_4 - M_5 \\
&= (A+C)(E+F) - (A-D)(E+H) \\
&\quad - A(F-H) - (C+D)E \\
&= AE + AF + CE + CF - AE - AH \\
&\quad + DE + DH - AF + AH - CE - DE \\
&= CF + DH
\end{aligned}
$$

Analysis of Strassen's Algorithm

$$
T(n) = \begin{cases} c & \text{if } n = 1 \\ 7T(n/2) + dn^2 & \text{otherwise} \end{cases}
$$

where c, d are constants.

$$
\begin{aligned}
T(n) &= 7T(n/2) + dn^2 \\
&= 7(7T(n/4) + d(n/2)^2) + dn^2 \\
&= 7^2 T(n/4) + 7dn^2/4 + dn^2 \\
&= 7^3 T(n/8) + 7^2 dn^2/4^2 + 7dn^2/4 + dn^2 \\
&= 7^i T(n/2^i) + dn^2 \sum_{j=0}^{i-1} (7/4)^j \\
&= 7^{\log n} T(1) + dn^2 \sum_{j=0}^{\log n - 1} (7/4)^j \\
&= cn^{\log 7} + dn^2 \frac{(7/4)^{\log n} - 1}{7/4 - 1} \\
&= cn^{\log 7} + \frac{4}{3} dn^2 (\frac{n^{\log 7}}{n^2} - 1) \\
&= O(n^{\log 7}) \\
&\approx O(n^{2.8})
\end{aligned}
$$

State of the Art

Integer multiplication: $O(n \log n \log\log n)$.

Schönhage and Strassen, "Schnelle multiplikation grosser zahlen", *Computing*, Vol. 7, pp. 281–292, 1971.

Matrix multiplication: $O(n^{2.376})$.

Coppersmith and Winograd, "Matrix multiplication via arithmetic progressions", *Journal of Symbolic Computation*, Vol. 9, pp. 251–280, 1990.

Assigned Reading

CLR Chapter 10.1, 31.2.

POA Sections 7.1–7.3

Algorithms Course Notes
Divide and Conquer 2

Ian Parberry*

Fall 2001

Summary

Quicksort

- The algorithm
- Average case analysis — $O(n \log n)$
- Worst case analysis — $O(n^2)$.

Every sorting algorithm based on comparisons and swaps must make $\Omega(n \log n)$ comparisons in the worst case.

Quicksort

Let S be a list of n distinct numbers.

```
    function quicksort(S)
1.    if |S| ≤ 1
2.      then return(S)
3.      else
4.        Choose an element a from S
5.        Let S1, S2, S3 be the elements of S
          which are respectively <, =, > a
6.        return(quicksort(S1),S2,quicksort(S3))
```

Terminology: a is called the pivot value. The operation in line 5 is called pivoting on a.

Average Case Analysis

Let $T(n)$ be the average number of comparisons used by quicksort when sorting n distinct numbers. Clearly $T(0) = T(1) = 0$.

Suppose a is the ith smallest element of S. Then

$$\begin{aligned} |S_1| &= i-1 \\ |S_2| &= 1 \\ |S_3| &= n-i \end{aligned}$$

The recursive calls need average time $T(i-1)$ and $T(n-i)$, and i can have any value from 1 to n with equal probability. Splitting S into S_1, S_2, S_3 takes $n-1$ comparisons (compare a to $n-1$ other values).

Therefore, for $n \geq 2$,

$$T(n) \leq \frac{1}{n} \sum_{i=1}^{n} (T(i-1) + T(n-i)) + n - 1$$

Now,

$$\begin{aligned} &\sum_{i=1}^{n} (T(i-1) + T(n-i)) \\ &= \sum_{i=1}^{n} T(i-1) + \sum_{i=1}^{n} T(n-i) \\ &= \sum_{i=0}^{n-1} T(i) + \sum_{i=0}^{n-1} T(i) \\ &= 2\sum_{i=2}^{n-1} T(i) \end{aligned}$$

Therefore, for $n \geq 2$,

$$T(n) \leq \frac{2}{n} \sum_{i=2}^{n-1} T(i) + n - 1$$

How can we solve this? Not by repeated substitution!

Multiply both sides of the recurrence for $T(n)$ by n. Then, for $n \geq 2$,

$$nT(n) = 2\sum_{i=2}^{n-1} T(i) + n^2 - n.$$

Hence, substituting $n-1$ for n, for $n \geq 3$,

$$(n-1)T(n-1) = 2\sum_{i=2}^{n-2} T(i) + n^2 - 3n + 2.$$

Subtracting the latter from the former,

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2(n-1),$$

for all $n \geq 3$. Hence, for all $n \geq 3$,

$$nT(n) = (n+1)T(n-1) + 2(n-1).$$

Therefore, dividing both sides by $n(n+1)$,

$$T(n)/(n+1) = T(n-1)/n + 2(n-1)/n(n+1).$$

Define $S(n) = T(n)/(n+1)$. Then, by definition, $S(0) = S(1) = 0$, and by the above, for all $n \geq 3$, $S(n) = S(n-1) + 2(n-1)/n(n+1)$. This is true even for $n = 2$, since $S(2) = T(2)/3 = 1/3$. Therefore,

$$S(n) \leq \begin{cases} 0 & \text{if } n \leq 1 \\ S(n-1) + 2/n & \text{otherwise.} \end{cases}$$

Solve by repeated substitution:

$$\begin{aligned} S(n) &\leq S(n-1) + 2/n \\ &\leq S(n-2) + 2/(n-1) + 2/n \\ &\leq S(n-3) + 2/(n-2) + 2/(n-1) + 2/n \\ &\leq S(n-i) + 2\sum_{j=n-i+1}^{n} \frac{1}{j}. \end{aligned}$$

Therefore, taking $i = n-1$,

$$S(n) \leq S(1) + 2\sum_{j=2}^{n} \frac{1}{j} = 2\sum_{j=2}^{n} \frac{1}{j} \leq 2\int_1^n \frac{1}{x}\mathrm{d}x = \ln n.$$

Therefore,

$$\begin{aligned} T(n) &= (n+1)S(n) \\ &< 2(n+1)\ln n \\ &= 2(n+1)\log n/\log e \\ &\approx 1.386(n+1)\log n. \end{aligned}$$

Worst Case Analysis

Thus quicksort uses $O(n \log n)$ comparisons on average. Quicksort is faster than mergesort by a small constant multiple in the average case, but much worse in the worst case.

How about the worst case? In the worst case, $i = 1$ and

$$T(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ T(n-1) + n - 1 & \text{otherwise} \end{cases}$$

It is easy to show that this is $\Theta(n^2)$.

Best Case Analysis

How about the best case? In the best case, $i = n/2$ and

$$T(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ 2T(n/2) + n - 1 & \text{otherwise} \end{cases}$$

for some constants c, d.

It is easy to show that this is $n \log n + O(n)$. Hence, the average case number of comparisons used by quicksort is only 39% more than the best case.

A Program

The algorithm can be implemented as a program that runs in time $O(n \log n)$. To sort an array $S[1..n]$, call quicksort$(1, n)$:

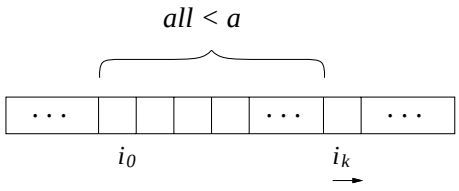
```
1.   procedure quicksort(ℓ, r)
2.     comment sort S[ℓ..r]
3.     i := ℓ; j := r;
       a := some element from S[ℓ..r];
4.     repeat
5.       while S[i] < a do i := i + 1;
6.       while S[j] > a do j := j − 1;
7.       if i ≤ j then
8.         swap S[i] and S[j];
9.         i := i + 1; j := j − 1;
10.    until i > j;
11.    if ℓ < j then quicksort(ℓ, j);
12.    if i < r then quicksort(i, r);
```

Correctness Proof

Consider the loop on line 5.

5. **while** $S[i] < a$ **do** $i := i + 1$;

Loop invariant: For $k \geq 0$, $i_k = i_0 + k$ and $S[v] < a$ for all $i_0 \leq v < i_k$.



Proof by induction on k. The invariant is vacuously true for $k = 0$.

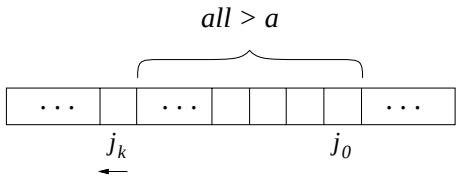
Now suppose $k > 0$. By the induction hypothesis, $i_{k-1} = i_0 + k - 1$, and $S[v] < a$ for all $i_0 \leq v < i_{k-1}$. Since we enter the while-loop on the kth iteration, it must be the case that $S[i_{k-1}] < a$. Therefore, $S[v] < a$ for all $i_0 \leq v \leq i_{k-1}$. Furthermore, i is incremented in the body of the loop, so $i_k = i_{k-1} + 1 = (i_0 + k - 1) + 1 = i_0 + k$. This makes both parts of the hypothesis true.

Conclusion: upon exiting the loop on line 5, $S[i] \geq a$ and $S[v] < a$ for all $i_0 \leq v < i$.

Consider the loop on line 6.

6. **while** $S[j] > a$ **do** $j := j - 1$;

Loop invariant: For $k \geq 0$, $j_k = j_0 - k$ and $S[v] > a$ for all $j_k < v \leq j_0$.



Proof is similar to the above.

Conclusion: upon exiting the loop on line 6, $S[j] \leq a$ and $S[v] > a$ for all $j < v \leq j_0$.

The loops on lines 5,6 will always halt.

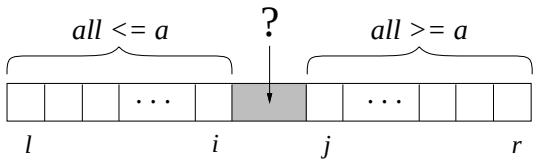
Question: Why?

Consider the repeat-loop on lines 4–10.

4. **repeat**
5. **while** $S[i] < a$ **do** $i := i + 1$;
6. **while** $S[j] > a$ **do** $j := j - 1$;
7. **if** $i \leq j$ **then**
8. swap $S[i]$ and $S[j]$;
9. $i := i + 1$; $j := j - 1$;
10. **until** $i > j$;

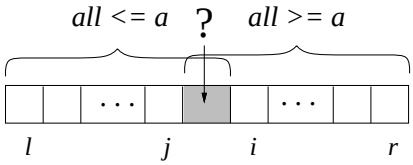
Loop invariant: after each iteration, either $i \leq j$ and

- $S[v] \leq a$ for all $\ell \leq v \leq i$, and
- $S[v] \geq a$ for all $j \leq v \leq r$.



or $i > j$ and

- $S[v] \leq a$ for all $\ell \leq v < i$, and
- $S[v] \geq a$ for all $j < v \leq r$.



After lines 5,6

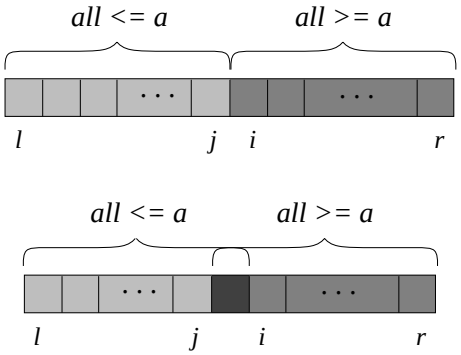
- $S[v] \leq a$ for all $\ell \leq v < i$, and
- $S[i] \geq a$
- $S[v] \geq a$ for all $j < v \leq r$.
- $S[j] \leq a$

If $i > j$, then the loop invariant holds. Otherwise, line 8 makes it hold:

- $S[v] \leq a$ for all $\ell \leq v \leq i$, and
- $S[v] \geq a$ for all $j \leq v \leq r$.

The loop terminates since i is incremented and j is decremented each time around the loop as long as $i \leq j$.

Hence we exit the repeat-loop with small values to the left, and big values to the right.



(How can each of these scenarios happen?)

Correctness proof is by induction on the size of the chunk of the array, $r-\ell+1$. It works for an array of size 1 (trace through the algorithm). In each of the two scenarios above, the two halves of the array are smaller than the original (since i and j must cross). Hence, by the induction hypothesis, the recursive call sorts each half, which in both scenarios means that the array is sorted.

What should we use for a? Candidates:

- $S[\ell]$, $S[r]$ (vanilla)
- $S[\lfloor(\ell+r)/2\rfloor]$ (fast on "nearly sorted" data)
- $S[m]$ where m is a pseudorandom value, $\ell \leq m \leq r$ (good for repeated use on similar data)

A Lower Bound

Claim: Any sorting algorithm based on comparisons and swaps must make $\Omega(n \log n)$ comparisons in the worst case.

Mergesort makes $O(n \log n)$ comparisons. So, there is no comparison-based sorting algorithm that is faster than mergesort by more than a constant multiple.

Proof of Lower Bound

A sorting algorithm permutes its inputs into sorted order.

It must perform one of $n!$ possible permutations.

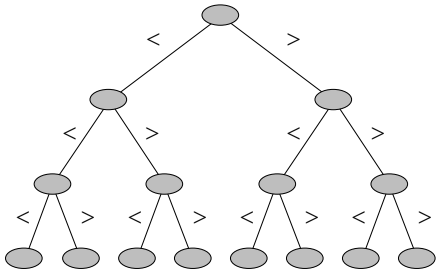
It doesn't know which until it compares its inputs.

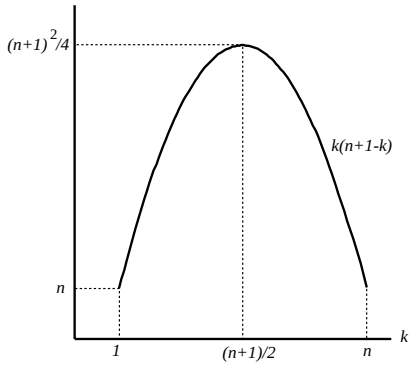
Consider a "many worlds" view of the algorithm. Initially, it knows nothing about the input.

After it makes one comparison, it can be in one of two possible worlds, depending on the result of that comparison.

After it makes 2 comparisons, each of these worlds can give rise to at most 2 more possible worlds.

Decision Tree


< >
< > < >
< > < > < > < >
< > < > < > < > < > < > < >
< > < > < > < > < > < > < > < > < > < > < > < > < >
< > < > < > < > < > < > < > < > < > < > < > < > < >
< > < > < > < > < > < >



Therefore,

$$\prod_{k=1}^{n} n \leq n!^2 \leq \prod_{k=1}^{n} \frac{(n+1)^2}{4}$$

That is,

$$n^{n/2} \leq n! \leq (n+1)^n/2^n$$

Hence

$$\log n! = \Theta(n \log n).$$

More precisely (Stirling's approximation),

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

Hence,

$$\log n! \sim n \log n + \Theta(n).$$

Conclusion

$T(n) = \Omega(n \log n)$, and hence any sorting algorithm based on comparisons and swaps must make $\Omega(n \log n)$ comparisons in the worst case.

This is called the decision tree lower bound.

Mergesort meets this lower bound. Quicksort doesn't.

It can also be shown that any sorting algorithm based on comparisons and swaps must make $\Omega(n \log n)$ comparisons on average. Both quicksort and mergesort meet this bound.

Assigned Reading

CLR Chapter 8.

POA 7.5

Algorithms Course Notes
Divide and Conquer 3

Ian Parberry*

Fall 2001

Summary

More examples of divide and conquer.

- selection in average time $O(n)$
- binary search in time $O(\log n)$
- the towers of Hanoi and the end of the Universe

Selection

Let S be an array of n distinct numbers. Find the kth smallest number in S.

function select(S, k)
  if $|S| = 1$ **then return**$(S[1])$ **else**
    Choose an element a from S
    Let S_1, S_2 be the elements of S
    which are respectively $<, > a$
    Suppose $|S_1| = j$ (a is the $(j+1)$st item)
    if $k = j + 1$ **then return**(a)
      else if $k \leq j$ **then return**(select(S_1, k))
        else return(select$(S_2, k - j - 1)$)

Let $T(n)$ be the worst case for all k of the average number of comparisons used by procedure select on an array of n numbers. Clearly $T(1) = 0$.

Analysis

Now,

$$\begin{aligned} |S_1| &= j \\ |S_2| &= n - j - 1 \end{aligned}$$

Hence the recursive call needs an average time of either $T(j)$ or $T(n-j-1)$, and j can have any value from 0 to $n-1$ with equal probability.

The average time for the recursive calls is thus at most:

$$\frac{1}{n}\sum_{j=0}^{n-1}(\text{either } T(j) \text{ or } T(n-j-1))$$

When is it $T(j)$ and when is it $T(n-j-1)$?

- $k \leq j$: recurse on S_1, time $T(j)$
- $k = j + 1$: finished
- $k > j + 1$: recurse on S_2, time $T(n-j-1)$

The average time for the recursive calls is thus at most:

$$\frac{1}{n}(\sum_{j=0}^{k-2} T(n-j-1) + \sum_{j=k}^{n-1} T(j))$$

Splitting S into S_1, S_2 takes $n-1$ comparisons (as in quicksort).

Therefore, for $n \geq 2$,

$$\begin{aligned} T(n) &\leq \frac{1}{n}(\sum_{j=0}^{k-2} T(n-j-1) + \sum_{j=k}^{n-1} T(j)) + n - 1 \\ &= \frac{1}{n}(\sum_{j=n-k+1}^{n-1} T(j) + \sum_{j=k}^{n-1} T(j)) + n - 1 \end{aligned}$$

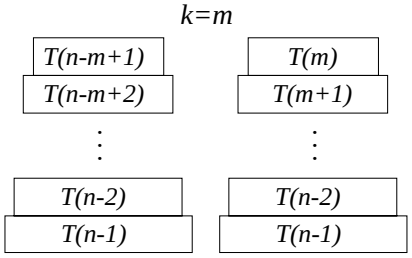
What value of k maximizes

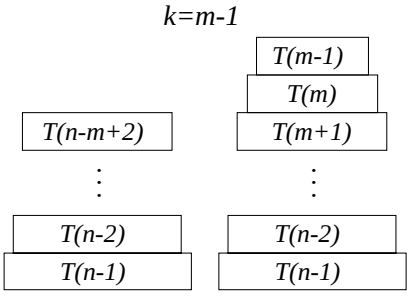
$$\sum_{j=n-k+1}^{n-1} T(j) + \sum_{j=k}^{n-1} T(j)?$$

Decrementing the value of k deletes a term from the left sum, and inserts a term into the right sum. Since T is the running time of an algorithm, it must be monotone nondecreasing.

Hence we must choose $k = n - k + 1$. Assume n is odd (the case where n is even is similar). This means we choose $k = (n+1)/2$.

Write $m=(n+1)/2$
(shorthand for this diagram only.)





Therefore,

$$T(n) \le \frac{2}{n} \sum_{j=(n+1)/2}^{n-1} T(j) + n - 1.$$

Claim that $T(n) \le 4(n-1)$. Proof by induction on n. The claim is true for $n = 1$. Now suppose that $T(j) \le 4(j-1)$ for all $j < n$. Then,

$$\begin{aligned}
& T(n) \\
\le\ & \frac{2}{n}\left(\sum_{j=(n+1)/2}^{n-1} T(j)\right) + n - 1 \\
\le\ & \frac{8}{n}\left(\sum_{j=(n+1)/2}^{n-1} (j-1)\right) + n - 1 \\
=\ & \frac{8}{n}\left(\sum_{j=(n-1)/2}^{n-2} j\right) + n - 1 \\
=\ & \frac{8}{n}\left(\sum_{j=1}^{n-2} j - \sum_{j=1}^{(n-3)/2} j\right) + n - 1 \\
=\ & \frac{8}{n}\left(\frac{(n-1)(n-2)}{2} - \frac{(n-3)(n-1)}{8}\right) \\
& + n - 1 \\
=\ & \frac{1}{n}(4n^2 - 12n + 8 - n^2 + 4n - 3) + n - 1 \\
=\ & 3n - 8 + \frac{5}{n} + n - 1 \\
\le\ & 4n - 4
\end{aligned}$$

Hence the selection algorithm runs in average time $O(n)$.

Worst case $O(n^2)$ (just like the quicksort analysis).

Comments

- There is an $O(n)$ worst-case time algorithm that uses divide and conquer (it makes a smart choice of a). The analysis is more difficult.
- How should we pick a? It could be $S[1]$ (average case significant in the long run) or a random element of S (the worst case doesn't happen with particular inputs).

Binary Search

Find the index of x in a sorted array A.

```
function search(A, x, ℓ, r)
  comment find x in A[ℓ..r]
  if ℓ = r then return(ℓ) else
    m := ⌊(ℓ + r)/2⌋
    if x ≤ A[m]
      then return(search(A, x, ℓ, m))
      else return(search(A, x, m + 1, r))
```

Suppose n is a power of 2. Let $T(n)$ be the worst case number of comparisons used by procedure search on an array of n numbers.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T(n/2) + 1 & \text{otherwise} \end{cases}$$

(As in the maxmin algorithm, we must argue that the array is cut in half.)

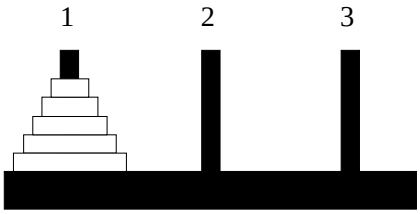
Hence,

$$T(n) \quad = \quad T(n/2) + 1$$

$$
\begin{aligned}
&= (T(n/4)+1)+1 \\
&= T(n/4)+2 \\
&= (T(n/8)+1)+2 \\
&= T(n/8)+3 \\
&= T(n/2^i)+i \\
&= T(1)+\log n \\
&= \log n
\end{aligned}
$$

Therefore, binary search on an array of size n takes time $O(\log n)$.

The Towers of Hanoi



Move all the disks from peg 1 to peg 3 using peg 2 as workspace without ever placing a disk on a smaller disk.

To move n disks from peg i to peg j using peg k as workspace

- Move $n-1$ disks from peg i to peg k using peg j as workspace.
- Move remaining disk from peg i to peg j.
- Move $n-1$ disks from peg k to peg j using peg i as workspace.

How Many Moves?

Let $T(n)$ be the number of moves it takes to move n disks from peg i to peg j.

Clearly,

$$
T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n-1)+1 & \text{otherwise} \end{cases}
$$

Hence,

$$
\begin{aligned}
T(n) &= 2T(n-1)+1 \\
&= 2(2T(n-2)+1)+1 \\
&= 4T(n-2)+2+1 \\
&= 4(2T(n-3)+1)+2+1 \\
&= 8T(n-3)+4+2+1 \\
&= 2^i T(n-i) + \sum_{j=0}^{i-1} 2^j \\
&= 2^{n-1}T(1) + \sum_{j=0}^{n-2} 2^j \\
&= 2^n - 1
\end{aligned}
$$

The End of the Universe

According to legend, there is a set of 64 gold disks on 3 diamond needles, called the Tower of Brahma. Legend reports that the Universe will end when the task is completed. (Édouard Lucas, *Récréations Mathématiques*, Vol. 3, pp 55–59, Gauthier-Villars, Paris, 1893.)

How many moves will it need? If done correctly, $T(64) = 2^{64} - 1 = 1.84 \times 10^{19}$ moves. At one move per second, that's

$$
\begin{aligned}
& 1.84 \times 10^{19} && \text{seconds} \\
= \; & 3.07 \times 10^{17} && \text{minutes} \\
= \; & 5.12 \times 10^{15} && \text{hours} \\
= \; & 2.14 \times 10^{14} && \text{days} \\
= \; & 5.85 \times 10^{11} && \text{years}
\end{aligned}
$$

Current age of Universe is $\approx 10^{10}$ years.

The Answer to Life, the Universe, and Everything

~~42~~

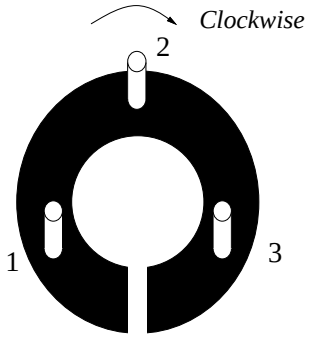
$2^{64}-1$

$= 18{,}446{,}744{,}073{,}709{,}552{,}936$

A Sneaky Algorithm

What if the monks are not good at recursion?

Imagine that the pegs are arranged in a circle. Instead of numbering the pegs, think of moving the disks one place clockwise or anticlockwise.



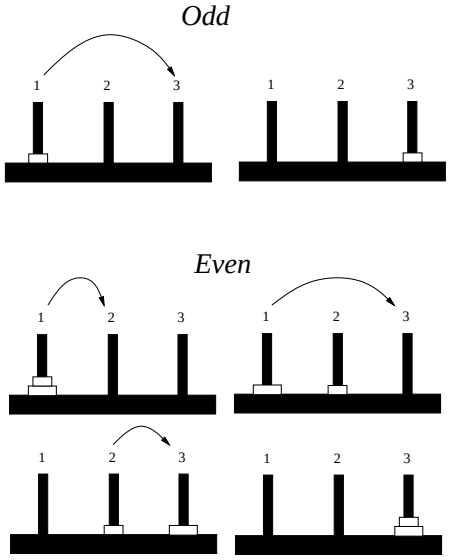
Impress Your Friends and Family

If there are an odd number of disks:

- Start by moving the smallest disk in an anticlockwise direction.
- Alternate between doing this and the only other legal move.

If there are an even number of disks, replace "anticlockwise" by "clockwise".

How do you remember whether to start clockwise or anticlockwise? Think of what happens with $n = 1$ and $n = 2$.



What happens if you use the wrong direction?

A Formal Algorithm

Let D be a direction, either *clockwise* or *anticlockwise*. Let $\overline{D}$ be the opposite direction.

To move n disks in direction D, alternate between the following two moves:

- If n is odd, move the smallest disk in direction D. If n is even, move the smallest disk in direction $\overline{D}$.
- Make the only other legal move.

This is exactly what the recursive algorithm does! Or is it?

Formal Claim

When the recursive algorithm is used to move n disks in direction D, it alternates between the following two moves:

- If n is odd, move the smallest disk in direction D. If n is even, move the smallest disk in direction $\overline{D}$.
- Make the only other legal move.

The Proof

Proof by induction on n. The claim is true when $n = 1$. Now suppose that the claim is true for n disks. Suppose we use the recursive algorithm to move $n+1$ disks in direction D. It does the following:

- Move n disks in direction $\overline{D}$.
- Move one disk in direction D.
- Move n disks in direction $\overline{D}$.

Let

- "D" denote moving the smallest disk in direction D,
- "$\overline{D}$" denote moving the smallest disk in direction $\overline{D}$
- "O" denote making the only other legal move.

Case 1. $n+1$ is odd. Then n is even, and so by the induction hypothesis, moving n disks in direction $\overline{D}$ uses

$$DODO\cdots OD$$

(NB. The number of moves is odd, so it ends with D, not O.)

Hence, moving $n+1$ disks in direction D uses

$$\underbrace{DODO\cdots OD}_{n \text{ disks}} O \underbrace{DODO\cdots OD}_{n \text{ disks}},$$

as required.

Case 2. $n+1$ is even. Then n is odd, and so by the induction hypothesis, moving n disks in direction $\overline{D}$ uses

$$\overline{D}O\overline{D}O\cdots O\overline{D}$$

(NB. The number of moves is odd, so it ends with $\overline{D}$, not O.)

Hence, moving $n+1$ disks in direction D uses

$$\underbrace{\overline{D}O\overline{D}O\cdots O\overline{D}}_{n \text{ disks}} O \underbrace{\overline{D}O\overline{D}O\cdots O\overline{D}}_{n \text{ disks}},$$

as required.

Hence by induction the claim holds for any number of disks.

Assigned Reading

CLR Chapter 10.2.

POA 7.5,7.6.

Algorithms Course Notes
Dynamic Programming 1

Ian Parberry*

Fall 2001

Summary

Dynamic programming: divide and conquer with a table.

Application to:

- Computing combinations
- Knapsack problem

Counting Combinations

To choose r things out of n, either

- Choose the first item. Then we must choose the remaining $r-1$ items from the other $n-1$ items. Or
- Don't choose the first item. Then we must choose the r items from the other $n-1$ items.

Therefore,

$$\left(\begin{array}{c} n \\ r \end{array} \right) = \left(\begin{array}{c} n-1 \\ r-1 \end{array} \right) + \left(\begin{array}{c} n-1 \\ r \end{array} \right)$$

Divide and Conquer

This gives a simple divide and conquer algorithm for finding the number of combinations of n things chosen r at a time.

function choose(n, r)
  if $r = 0$ or $n = r$ **then return**(1) **else**
    return(choose$(n-1, r-1)$ +
        choose$(n-1, r)$)

Correctness Proof: A simple induction on n.

Analysis: Let $T(n)$ be the worst case running time of choose(n, r) over all possible values of r.

Then,

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n-1) + d & \text{otherwise} \end{cases}$$

for some constants c, d.

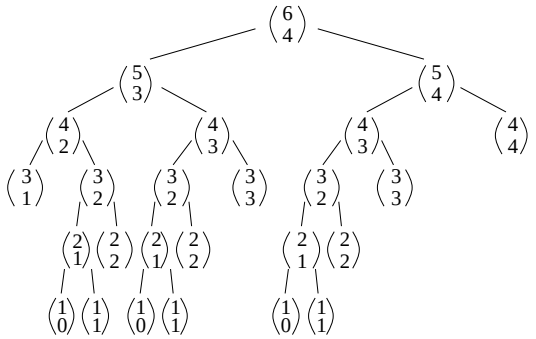
Hence,

$$\begin{aligned} T(n) &= 2T(n-1) + d \\ &= 2(2T(n-2) + d) + d \\ &= 4T(n-2) + 2d + d \\ &= 4(2T(n-3) + d) + 2 + d \\ &= 8T(n-3) + 4d + 2d + d \\ &= 2^i T(n-i) + d \sum_{j=0}^{i-1} 2^j \\ &= 2^{n-1} T(1) + d \sum_{j=0}^{n-2} 2^j \\ &= (c+d)2^{n-1} - d \end{aligned}$$

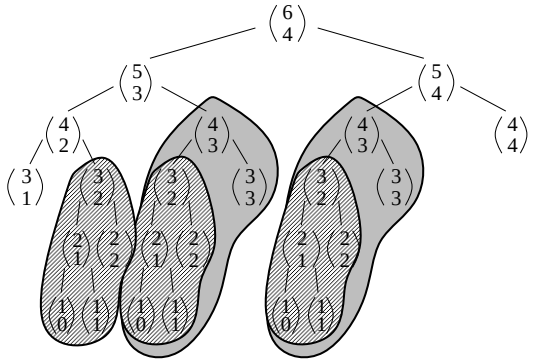
Hence, $T(n) = \Theta(2^n)$.

Example

The problem is, the algorithm solves the same subproblems over and over again!



Repeated Computation



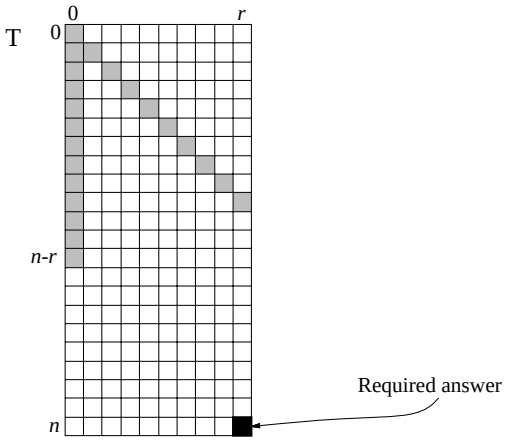
A Better Algorithm

Pascal's Triangle. Use a table T$[0..n, 0..r]$.

$$\mathrm{T}[i,j] \text{ holds } \left(\begin{array}{c} i \\ j \end{array} \right).$$

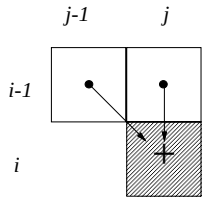
function choose(n, r)
  for $i := 0$ **to** $n - r$ **do** T$[i, 0]$:=1;
  for $i := 0$ **to** r **do** T$[i, i]$:=1;
  for $j := 1$ **to** r **do**
    for $i := j + 1$ **to** $n - r + j$ **do**
      T$[i, j]$:=T$[i - 1, j - 1]$ + T$[i - 1, j]$
  return(T$[n, r]$)

Initialization



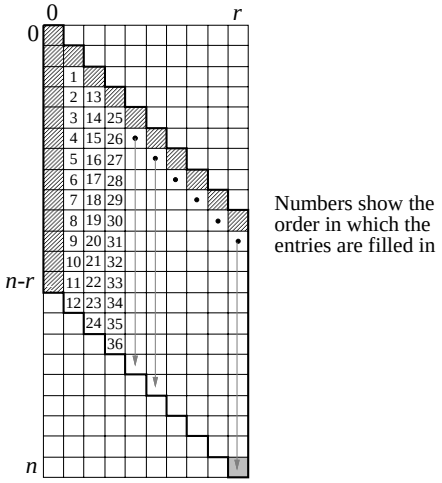
General Rule

To fill in T$[i, j]$, we need T$[i-1, j-1]$ and T$[i-1, j]$ to be already filled in.

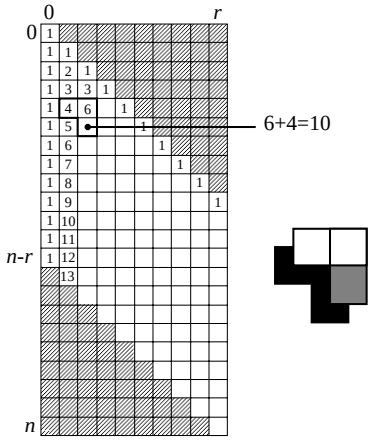


Filling in the Table

Fill in the columns from left to right. Fill in each of the columns from top to bottom.



Example



Analysis

How many table entries are filled in?

$(n-r+1)(r+1) = nr + n - r^2 + 1 \leq n(r+1) + 1$

Each entry takes time $O(1)$, so total time required is $O(n^2)$.

This is much better than $O(2^n)$.

Space: naive, $O(nr)$. Smart, $O(r)$.

Dynamic Programming

When divide and conquer generates a large number of identical subproblems, recursion is too expensive.

Instead, store solutions to subproblems in a table.

This technique is called dynamic programming.

Dynamic Programming Technique

To design a dynamic programming algorithm:

Identification:

- devise divide-and-conquer algorithm
- analyze — running time is exponential
- same subproblems solved many times

Construction:

- take part of divide-and-conquer algorithm that does the "conquer" part and replace recursive calls with table lookups
- instead of returning a value, record it in a table entry
- use base of divide-and-conquer to fill in start of table
- devise "look-up template"
- devise for-loops that fill the table using "look-up template"

Divide and Conquer

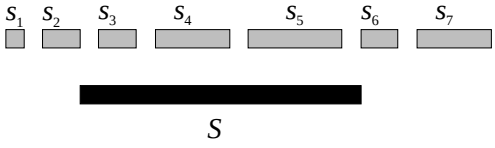
```
function choose(n,r)
  if r=0 or r=n then return(1) else
    return(choose(n-1,r-1)+choose(n-1,r))
```

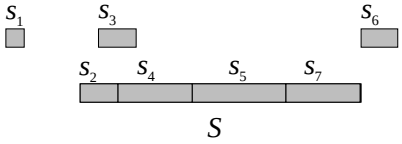
Dynamic Programming

```
function choose(n,r)
  for i:=0 to n-r do T[i,0]:=1
  for i:=0 to r do T[i,i]:=1
  for j:=1 to r do
    for i:=j+1 to n-r+j do
      T[i,j]:=T[i-1,j-1]+T[i-1,j]
  return(T[n,r])
```

The Knapsack Problem

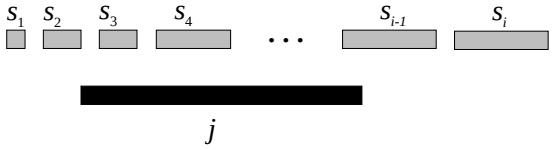
Given n items of length $s_1, s_2, \ldots, s_n$, is there a subset of these items with total length exactly S?





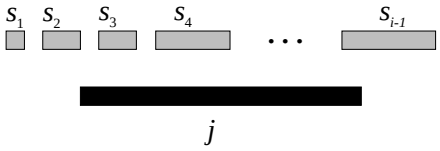
Divide and Conquer

Want knapsack(i, j) to return `true` if there is a subset of the first i items that has total length exactly j.

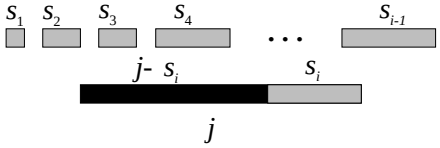


When can knapsack(i, j) return `true`? Either the ith item is used, or it is not.

- If the ith item is not used, and knapsack$(i-1, j)$ returns `true`.



- If the ith item is used, and knapsack$(i-1, j-s_i)$ returns `true`.



The Code

Call knapsack(n, S).

function knapsack(i, j)
  comment returns true if $s_1, \ldots, s_i$ can fill j
  if $i = 0$ **then return**(j=0)
  else if knapsack$(i-1, j)$ **then return**(`true`)
  else if $s_i \leq j$ **then**
    return(knapsack$(i-1, j-s_i)$)

Let $T(n)$ be the running time of knapsack(n, S).

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n-1) + d & \text{otherwise} \end{cases}$$

Hence, by standard techniques, $T(n) = \Theta(2^n)$.

Dynamic Programming

Store knapsack(i, j) in table $t[i, j]$.

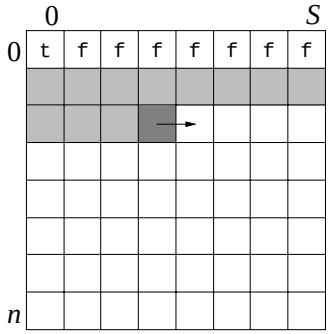
$t[i, j]$ is set to true iff either:

- $t[i-1, j]$ is true, or
- $t[i-1, j-s_i]$ makes sense and is true.

This is done with the following code:

$t[i, j] := t[i-1, j]$
if $j - s_i \geq 0$ **then**
  $t[i, j] :=$ $t[i, j]$ or $t[i-1, j-s_i]$

Filling in the Table



The Algorithm

function knapsack$(s_1, s_2, \ldots, s_n, S)$
1. $t[0, 0] :=$`true`
2. **for** $j := 1$ **to** S **do** $t[0, j] :=$`false`
3. **for** $i := 1$ **to** n **do**
4. **for** $j := 0$ **to** S **do**
5. $t[i, j] := t[i-1, j]$
6. **if** $j - s_i \geq 0$ **then**
7. $t[i, j] :=$ $t[i, j]$ or $t[i-1, j-s_i]$
8. **return**$(t[n, S])$

Analysis:

- Lines 1 and 8 cost $O(1)$.

- The for-loop on line 2 costs $O(S)$.
- Lines 5–7 cost $O(1)$.
- The for-loop on lines 4–7 costs $O(S)$.
- The for-loop on lines 3–7 costs $O(nS)$.

Therefore, the algorithm runs in time $O(nS)$. This is usable if S is small.

Example

$s_1 = 1$, $s_2 = 2$ $s_3 = 2$, $s_4 = 4$, $s_5 = 5$, $s_6 = 2$, $s_7 = 4$, $S = 15$.

$$
\begin{aligned}
t[i,j] &:= t[i-1,j] \text{ or } t[i-1,j-s_i] \\
t[3,3] &:= t[2,3] \text{ or } t[2,3-s_3] \\
t[3,3] &:= t[2,3] \text{ or } t[2,1]
\end{aligned}
$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	t	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f
1	t	t	f	f	f	f	f	f	f	f	f	f	f	f	f	f
2	t	t	t	t	f	f	f	f	f	f	f	f	f	f	f	f
3	t	t	t	t	t	t	f	f	f	f	f	f	f	f	f	f
4	t	t	t	t	t	t	t	t	t	t	f	f	f	f	f	f
5	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	f
6	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t
7	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t

Question: Can we get by with 2 rows?

Question: Can we get by with 1 row?

Assigned Reading

CLR Section 16.2.

POA Section 8.2.

$$
\begin{aligned}
t[i,j] &:= t[i-1,j] \text{ or } t[i-1,j-s_i] \\
t[3,4] &:= t[2,4] \text{ or } t[2,4-s_3] \\
t[3,4] &:= t[2,4] \text{ or } t[2,2]
\end{aligned}
$$

Algorithms Course Notes
Dynamic Programming 2

Ian Parberry*

Fall 2001

Summary

Dynamic programming applied to

- Matrix products problem

Matrix Product

Consider the problem of multiplying together n rectangular matrices.

$$M = M_1 \cdot M_2 \cdots M_n$$

where each M_i has r_{i-1} rows and r_i columns.

Suppose we use the naive matrix multiplication algorithm. Then a multiplication of a $p \times q$ matrix by a $q \times r$ matrix takes pqr operations.

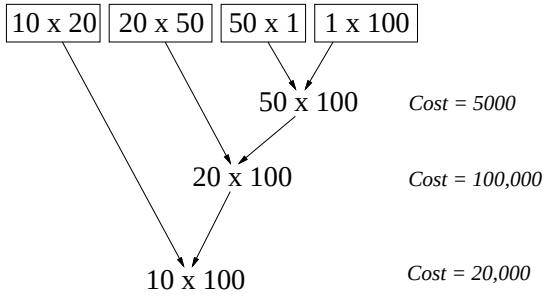
Matrix multiplication is associative, so we can evaluate the product in any order. However, some orders are cheaper than others. Find the cost of the cheapest one.

N.B. Matrix multiplication is <u>not</u> commutative.

Example

If we multiply from right to left:

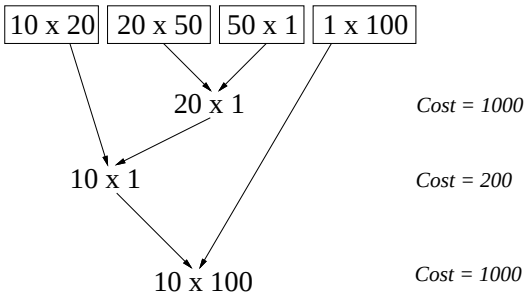
$$M = M_1 \cdot (M_2 \cdot (M_3 \cdot M_4))$$



Total cost is $5,000 + 100,000 + 20,000 = 125,000$.

However, if we begin in the middle:

$$M = (M_1 \cdot (M_2 \cdot M_3)) \cdot M_4$$



Total cost is $1000 + 200 + 1000 = 2200$.

The Naive Algorithm

One way is to try all possible ways of parenthesizing the n matrices. However, this will take exponential time since there are exponentially many ways of parenthesizing them.

Let $X(n)$ be the number of ways of parenthesizing the n matrices. It is easy to see that $X(1) = X(2) =$

1 and for $n > 2$,

$$X(n) = \sum_{k=1}^{n-1} X(k) \cdot X(n-k)$$

(just consider breaking the product into two pieces):

$$\underbrace{(M_1 \cdot M_2 \cdots M_k)}_{X(k) \text{ ways}} \cdot \underbrace{(M_{k+1} \cdots M_n)}_{X(n-k) \text{ ways}}$$

Therefore,

$$\begin{aligned} X(3) &= \sum_{k=1}^{2} X(k) \cdot X(3-k) \\ &= X(1) \cdot X(2) + X(2) \cdot X(1) \\ &= 2 \end{aligned}$$

This is easy to verify: $x(xx)$, $(xx)x$.

$$\begin{aligned} & X(4) \\ = & \sum_{k=1}^{3} X(k) \cdot X(4-k) \\ = & X(1) \cdot X(3) + X(2) \cdot X(2) + X(3) \cdot X(1) \\ = & 5 \end{aligned}$$

This is easy to verify: $x((xx)x)$, $x(x(xx))$, $(xx)(xx)$, $((xx)x)x$, $(x(xx))x$.

Claim: $X(n) \geq 2^{n-2}$. Proof by induction on n. The claim is certainly true for $n \leq 4$. Now suppose $n \geq 5$. Then

$$\begin{aligned} X(n) &= \sum_{k=1}^{n-1} X(k) \cdot X(n-k) \\ &\geq \sum_{k=1}^{n-1} 2^{k-2} 2^{n-k-2} \quad \text{(by ind. hyp.)} \\ &= \sum_{k=1}^{n-1} 2^{n-4} \\ &= (n-1) 2^{n-4} \\ &\geq 2^{n-2} \text{ (since } n \geq 5) \end{aligned}$$

Actually, it can be shown that

$$X(n) = \frac{1}{n} \left(\begin{array}{c} 2(n-1) \\ n-1 \end{array} \right).$$

Divide and Conquer

Let $\text{cost}(i, j)$ be the minimum cost of computing

$$M_i \cdot M_{i+1} \cdots M_j.$$

What is the cost of breaking the product at M_k?

$$(M_i \cdot M_{i+1} \cdots M_k) \cdot (M_{k+1} \cdots M_j).$$

It is $\text{cost}(i, k)$ plus $\text{cost}(k+1, j)$ plus the cost of multiplying an $r_{i-1} \times r_k$ matrix by an $r_k \times r_j$ matrix.

Therefore, the cost of breaking the product at M_k is

$$\text{cost}(i, k) + \text{cost}(k+1, j) + r_{i-1} r_k r_j.$$

A Divide and Conquer Algorithm

This gives a simple divide and conquer algorithm. Simply call $\text{cost}(1, n)$.

function cost(i, j)
  if $i = j$ **then return**(0) **else**
    return$(\min_{i \leq k < j}(\text{cost}(i, k) + \text{cost}(k+1, j)$
      $+ r_{i-1} r_k r_j))$

Correctness Proof: A simple induction.

Analysis: Let $T(n)$ be the worst case running time of $\text{cost}(i, j)$ when $j - i + 1 = n$.

Then, $T(0) = c$ and for $n > 0$,

$$T(n) = \sum_{m=1}^{n-1} (T(m) + T(n-m)) + dn$$

for some constants c, d.

Hence, for $n > 0$,

$$T(n) = 2 \sum_{m=1}^{n-1} T(m) + dn.$$

Therefore,

$$\begin{aligned} & T(n) - T(n-1) \\ = & (2 \sum_{m=1}^{n-1} T(m) + dn) - (2 \sum_{m=1}^{n-2} T(m) + d(n-1)) \\ = & 2T(n-1) + d \end{aligned}$$

That is,

$$T(n) = 3T(n-1) + d$$

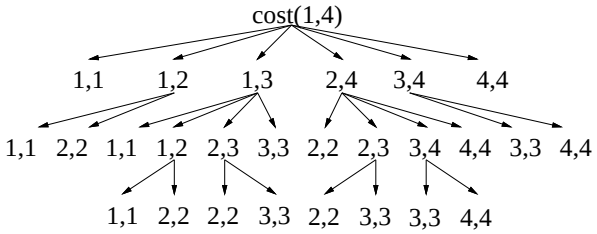
And so,

$$\begin{aligned}
T(n) &= 3T(n-1) + d \\
&= 3(3T(n-2) + d) + d \\
&= 9T(n-2) + 3d + d \\
&= 9(3T(n-3) + d) + 3d + d \\
&= 27T(n-3) + 9d + 3d + d \\
&= 3^m T(n-m) + d\sum_{\ell=0}^{m-1} 3^\ell \\
&= 3^n T(0) + d\sum_{\ell=0}^{n-1} 3^\ell \\
&= c3^n + d\frac{3^n - 1}{2} \\
&= (c + d/2)3^n - d/2
\end{aligned}$$

Hence, $T(n) = \Theta(3^n)$.

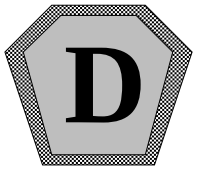
Example

The problem is, the algorithm solves the same subproblems over and over again!



Dynamic Programming

This is a job for ...



Dynamic programming!

Faster than a speeding divide and conquer!

Able to leap tall recursions in a single bound!

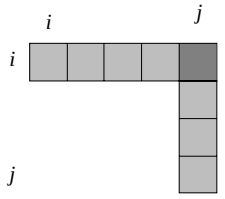
Details

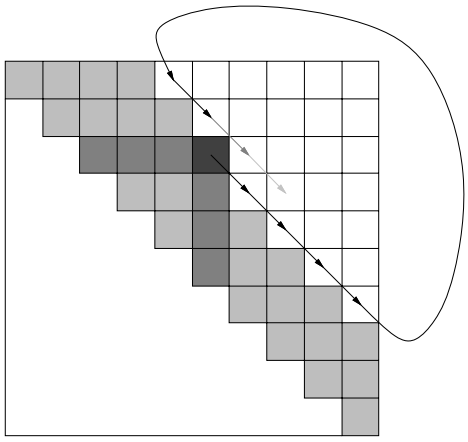
Store cost(i, j) in $m[i, j]$. Therefore, $m[i, j] = 0$ if $i = j$, and

$$\min_{i \le k < j} (m[i, k] + m[k + 1, j] + r_{i-1} r_k r_j)$$

otherwise.

When we fill in the table, we had better make sure that $m[i, k]$ and $m[k + 1, j]$ are filled in before we compute $m[i, j]$. Compute entries of $m[]$ in increasing order of difference between parameters.

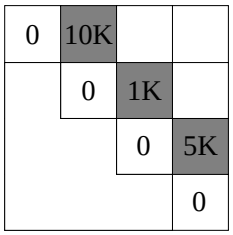




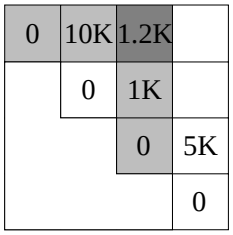
Example

$r_0 = 10$, $r_1 = 20$, $r_2 = 50$, $r_3 = 1$, $r_4 = 100$.

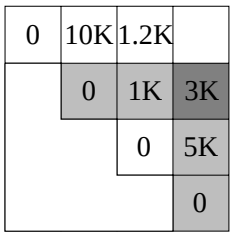
$$\begin{aligned}
m[1, 2] &= \min_{1 \le k < 2} (m[1, k] + m[k + 1, 2] + r_0 r_k r_2) \\
&= r_0 r_1 r_2 = 10000 \\
m[2, 3] &= r_1 r_2 r_3 = 1000 \\
m[3, 4] &= r_2 r_3 r_4 = 5000
\end{aligned}$$



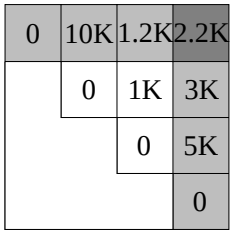
$$
\begin{aligned}
& m[1,3] \\
= \; & \min_{1 \le k < 3} (m[1,k] + m[k+1,3] + r_0 r_k r_3) \\
= \; & \min(m[1,1] + m[2,3] + r_0 r_1 r_3, \\
& \qquad m[1,2] + m[3,3] + r_0 r_2 r_3) \\
= \; & \min(0 + 1000 + 200, 10000 + 0 + 500) \\
= \; & 1200
\end{aligned}
$$



$$
\begin{aligned}
& m[2,4] \\
= \; & \min_{2 \le k < 4} (m[2,k] + m[k+1,4] + r_1 r_k r_4) \\
= \; & \min(m[2,2] + m[3,4] + r_1 r_2 r_4, \\
& \qquad m[2,3] + m[4,4] + r_1 r_3 r_4) \\
= \; & \min(0 + 5000 + 100000, 1000 + 0 + 2000) \\
= \; & 3000
\end{aligned}
$$

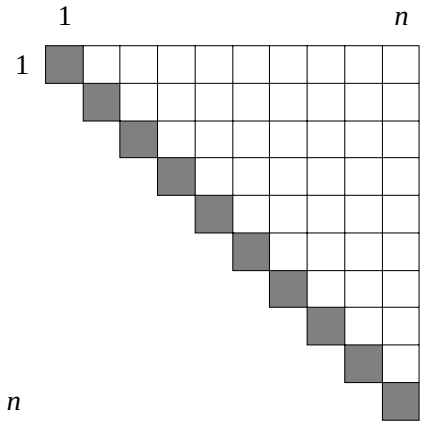


$$
\begin{aligned}
& m[1,4] \\
= \; & \min_{1 \le k < 4} (m[1,k] + m[k+1,4] + r_0 r_k r_4) \\
= \; & \min(m[1,1] + m[2,4] + r_0 r_1 r_4, \\
& \qquad m[1,2] + m[3,4] + r_0 r_2 r_4, \\
& \qquad m[1,3] + m[4,4] + r_0 r_3 r_4) \\
= \; & \min(0 + 3000 + 20000, \\
& \qquad 10000 + 5000 + 50000, \\
& \qquad 1200 + 0 + 1000) \\
= \; & 2200
\end{aligned}
$$



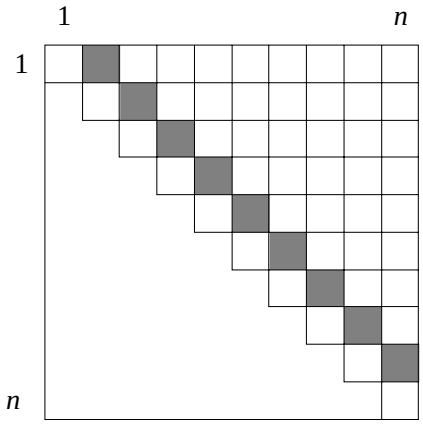
Filling in the Diagonal

for $i := 1$ **to** n **do** $m[i,i] := 0$



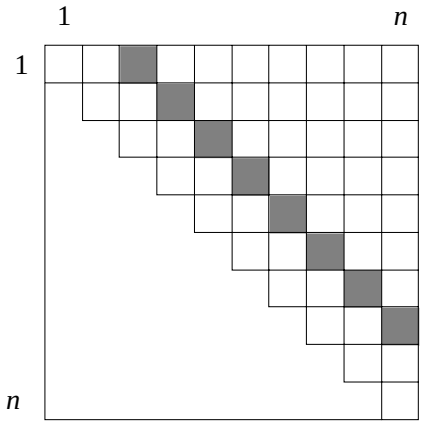
Filling in the First Superdiagonal

for $i := 1$ **to** $n-1$ **do**
  $j := i+1$
  $m[i,j] := \cdots$



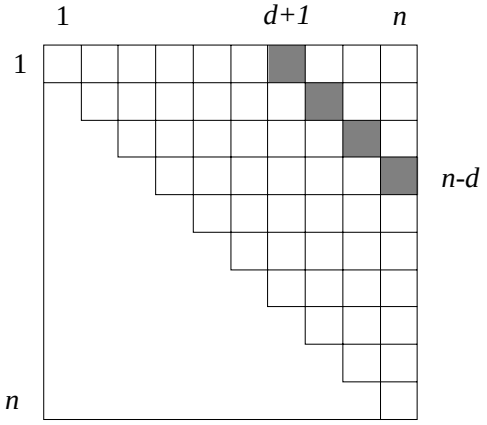
Filling in the Second Superdiagonal

for $i := 1$ **to** $n - 2$ **do**
  $j := i + 2$
  $m[i, j] := \cdots$



Filling in the dth Superdiagonal

for $i := 1$ **to** $n - d$ **do**
  $j := i + d$
  $m[i, j] := \cdots$



The Algorithm

function matrix(n)
1. **for** $i := 1$ **to** n **do** $m[i, i] := 0$
2. **for** $d := 1$ **to** $n - 1$ **do**
3. **for** $i := 1$ **to** $n - d$ **do**
4. $j := i + d$
5. $m[i, j] := \min_{i \le k < j}(m[i, k] + m[k + 1, j] + r_{i-1} r_k r_j)$
6. **return**(m$[1, n]$)

Analysis:

- Line 1 costs $O(n)$.
- Line 5 costs $O(n)$ (can be done with a single for-loop).
- Lines 4 and 6 costs $O(1)$.
- The for-loop on lines 3–5 costs $O(n^2)$.
- The for-loop on lines 2–5 costs $O(n^3)$.

Therefore the algorithm runs in time $O(n^3)$. This is a characteristic of many dynamic programming algorithms.

Other Orders

1	8	14	19	23	26	28
	2	9	15	20	24	27
		3	10	16	21	25
			4	11	17	22
				5	12	18
					6	13
						7

1	13	14	22	23	26	28
	2	12	15	21	24	27
		3	11	16	20	25
			4	10	17	19
				5	9	18
					6	8
						7

1	3	6	10	15	21	28
	2	5	9	14	20	27
		4	8	13	19	26
			7	12	18	25
				11	17	24
					16	23
						22

22	23	24	25	26	27	28
	16	17	18	19	20	21
		11	12	13	14	15
			7	8	9	10
				4	5	6
					2	3
						1

Assigned Reading

CLR Section 16.1, 16.2.

POA Section 8.1.

Algorithms Course Notes
Dynamic Programming 3

Ian Parberry*

Fall 2001

Summary

Binary search trees

- Their care and feeding
- Yet another dynamic programming example – optimal binary search trees

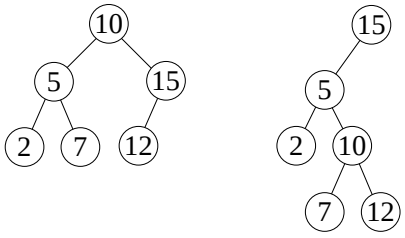
Binary Search Trees

A binary search tree (BST) is a binary tree with the data stored in the nodes.

1. The value in a node is larger than the values in its left subtree.
2. The value in a node is smaller than the values in its right subtree.

Note that this implies that the values must be distinct. Let $\ell(v)$ denote the value stored at node v.

Examples



Application

Binary search trees are useful for storing a set S of ordered elements, with operations:

1. search(x, S) return `true` iff $x \in S$.
2. min(S) return the smallest value in S.
3. delete(x, S) delete x from S.
4. insert(x, S) insert x into S.

The Search Operation

procedure search(x, v)
comment is x in BST with root v?
if $x = \ell(v)$ **then return**(`true`) **else**
  if $x < \ell(v)$ **then**
    if v has a left child w
      then return(search(x, w))
      else return(`false`)
  else if v has a right child w
      then return(search(x, w))
      else return(`false`)

Correctness: An easy induction on the number of layers in the tree.

Analysis: Let $T(d)$ be the runtime on a BST of d layers. Then $T(0) = 0$ and for $d > 0$, $T(d) \leq T(d-1) + O(1)$. Therefore, $T(d) = O(d)$.

The Min Operation

procedure min(v);
comment return smallest in BST with root v
if v has a left child w
  then return(min(w))
  else return$(\ell(v))$)

Correctness: An easy induction on the number of layers in the tree.

Analysis: Once again $O(d)$.

The Delete Operation

To delete x from the BST:

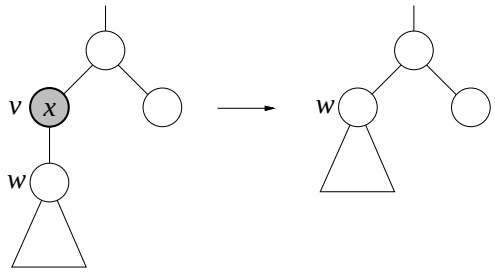
Procedure search can be modified to return the node v that has value x (instead of a Boolean value). Once this has been found, there are four cases.

1. v is a leaf.

Then just remove v from the tree.

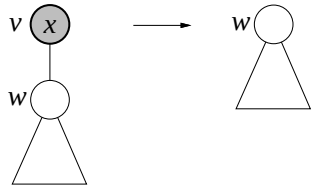
2. v has exactly one child w, and v is not the root.

Then make the parent of v the parent of w.



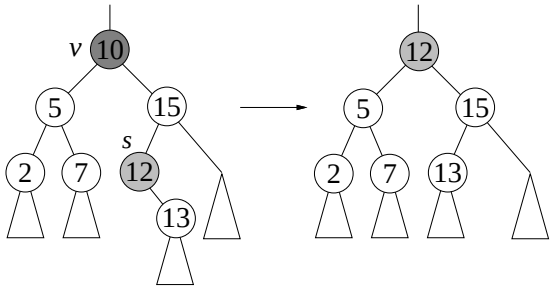
3. v has exactly one child w, and v is the root.

Then make w the new root.



4. v has 2 children.

This is the interesting case. First, find the smallest element s in the right subtree of v (using procedure min). Delete s (note that this uses case 1 or 2 above). Replace the value in node v with s.



Note: could have used largest in left subtree for s.

Correctness: Obvious for cases 1, 2, 3. In case 4, s is larger than all of the values in the left subtree of v, and smaller than the other values in the right subtree of v. Therefore s can replace the value in v.

Analysis: Running time $O(d)$ — dominated by search for node v.

The Insert Operation

procedure insert(x, v);
comment insert x in BST with root v
if v is the empty tree
  then create a root node v with $\ell(v) = x$
else
  if $x < \ell(v)$ **then**
    if v has a left child w
      then insert(x, w)
      else
        create a new left child w of v
        $\ell(w) := x$
  else if $x > \ell(v)$ **then**
    if v has a right child w
      then insert(x, w)
      else
        create a new right child w of v
        $\ell(w) := x$

Correctness: Once again, an easy induction.

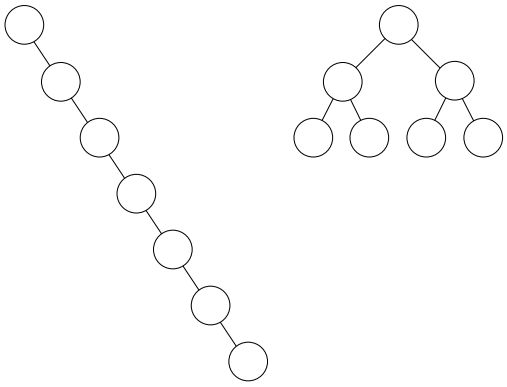
Analysis: Once again $O(d)$.

Analysis

All operations run in time $O(d)$.

But how big is d?

The worst case for an n node BST is $O(n)$ and $\Omega(\log n)$.



The average case is $O(\log n)$.

Average Case Analysis

What is the average case running time for n insertions into an empty BST? Suppose we insert $x_1, \ldots, x_n$, where $x_1 < x_2 < \cdots < x_n$ (not necessarily inserted in that order). Then

- Run time is proportional to number of comparisons.
- Measure number of comparisons, $T(n)$.
- The root is equally likely to be x_j for $1 \leq j \leq n$ (whichever is inserted first).

Suppose the root is x_j. List the values to be inserted in ascending order.

- $x_1, \ldots, x_{j-1}$: these go to the left of the root; needs $j-1$ comparisons to root, $T(j-1)$ comparisons to each other.
- x_j: this is the root; needs no comparisons
- $x_{j+1}, \ldots, x_n$: these go to the right of the root; needs $n-j$ comparisons to root, $T(n-j)$ comparisons to each other.

Therefore, when the root is x_j the total number of comparisons is

$$T(j-1) + T(n-j) + n - 1$$

Therefore, $T(0) = 0$, and for $n > 0$,

$$\begin{aligned} T(n) &= \frac{1}{n}\sum_{j=1}^{n}(n-1+T(j-1)+T(n-j)) \\ &= n-1+\frac{1}{n}(\sum_{j=1}^{n}T(j-1)+\sum_{j=1}^{n}T(n-j)) \\ &= n-1+\frac{2}{n}\sum_{j=0}^{n-1}T(j) \end{aligned}$$

This is just like the quicksort analysis! It can be shown similarly that $T(n) \leq kn\log n$ where $k = \log_e 4 \approx 1.39$.

Therefore, each insert operation takes $O(\log n)$ time on average.

Optimal Binary Search Trees

Given:

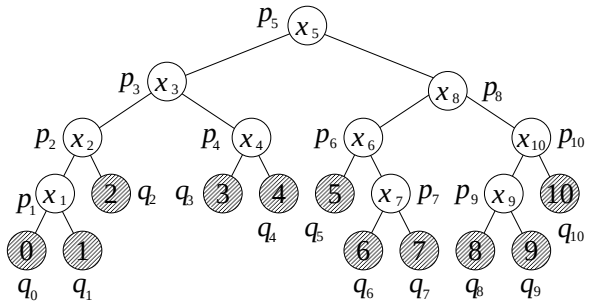
- $S = \{x_1, \ldots x_n\}$, $x_i < x_{i+1}$ for $1 \leq i < n$.
- For all $1 \leq i \leq n$, the probability p_i that we will be asked search(x_i, S).
- For all $0 \leq i \leq n$, the probability q_i that we will be asked search(x, S) for some $x_i < x < x_{i+1}$ (where $x_0 = -\infty$, $x_{n+1} = \infty$).

The problem: construct a BST that has the minimum number of expected comparisons.

Fictitious Nodes

Add fictitious nodes labelled $0, 1, \ldots, n$ to the BST. Node i is where we would fall off the tree on a query search(x, S) where $x_i < x < x_{i+1}$.

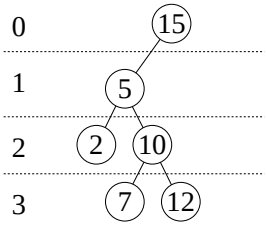
Example:



Depth of a Node

Definition: The depth of a node v is

- 0 if v is the root
- $d+1$ if v's parent has depth d



Cost of a Node

Let depth(x_i) be the depth of the node v with $\ell(v) = x_i$.

Number of comparisons for search(x_i, S) is

$$\text{depth}(x_i) + 1.$$

This happens with probability p_i.

Number of comparisons for search(x, S) for some $x_i < x < x_{i+1}$ is

$$\text{depth}(i).$$

This happens with probability q_i.

Cost of a BST

The expected number of comparisons is therefore

$$\underbrace{\sum_{h=1}^{n} p_h(\text{depth}(x_h)+1)}_{\text{real}} + \underbrace{\sum_{h=0}^{n} q_h \text{depth}(h)}_{\text{fictitious}}.$$

Given the probabilities, we want to find the BST that minimizes this value.

Call it the cost of the BST.

Weight of a BST

Let $T_{i,j}$ be the min cost BST for $x_{i+1}, \ldots, x_j$, which has fictitious nodes $i, \ldots, j$. We are interested in $T_{0,n}$.

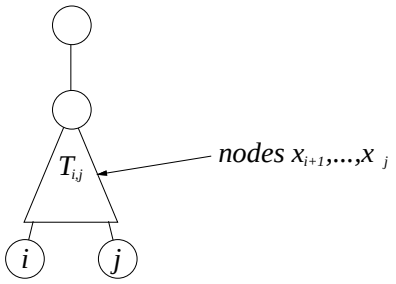
Let $c_{i,j}$ be the cost of $T_{i,j}$.

Let

$$w_{i,j} = \sum_{h=i+1}^{j} p_h + \sum_{h=i}^{j} q_h.$$

What is $w_{i,j}$? Call it the weight of $T_{i,j}$. Increasing the depths by 1 by making $T_{i,j}$ the child of some node increases the cost of $T_{i,j}$ by $w_{i,j}$.

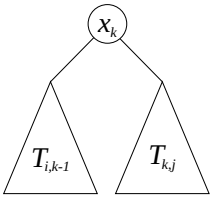
$$c_{i,j} = \sum_{h=i+1}^{j} p_h(\text{depth}(x_h)+1) + \sum_{h=i}^{j} q_h \text{depth}(h).$$



Constructing $T_{i,j}$

To find the best tree $T_{i,j}$:

Choose a root x_k. Construct $T_{i,k-1}$ and $T_{k,j}$.



Computing $c_{i,j}$

$$\begin{aligned} c_{i,j} &= (c_{i,k-1} + w_{i,k-1}) + (c_{k,j} + w_{k,j}) + p_k \\ &= c_{i,k-1} + c_{k,j} + (w_{i,k-1} + w_{k,j} + p_k) \\ &= c_{i,k-1} + c_{k,j} + \sum_{h=i+1}^{k-1} p_h + \sum_{h=i}^{k-1} q_h + \end{aligned}$$

$$\begin{aligned} & \sum_{h=k+1}^{j} p_h + \sum_{h=k}^{j} q_h + p_k \\ = \; & c_{i,k-1} + c_{k,j} + \sum_{h=i+1}^{j} p_h + \sum_{h=i}^{j} q_h \\ = \; & c_{i,k-1} + c_{k,j} + w_{i,j} \end{aligned}$$

Which x_k do we choose to be the root?

Pick k in the range $i+1 \leq k \leq j$ such that

$$c_{i,j} = c_{i,k-1} + c_{k,j} + w_{i,j}$$

is smallest.

Loose ends:

- if $k = i+1$, there is no left subtree
- if $k = j$, there is no right subtree
- $T_{i,i}$ is the empty tree, with $w_{i,i} = q_i$, and $c_{i,i} = 0$.

Dynamic Programming

To compute the cost of the minimum cost BST, store $c_{i,j}$ in a table $c[i,j]$, and store $w_{i,j}$ in a table $w[i,j]$.

for $i := 0$ **to** n **do**
  $w[i,i] := q_i$
  $c[i,i] := 0$
for $\ell := 1$ **to** n **do**
  for $i := 0$ **to** $n - \ell$ **do**
    $j := i + \ell$
    $w[i,j] := w[i,j-1] + p_j + q_j$
    $c[i,j] := \min_{i<k\leq j}(c[i,k-1] + c[k,j] + w[i,j])$

Correctness: Similar to earlier examples.

Analysis: $O(n^3)$.

Assigned Reading

CLR, Chapter 13.

POA Section 8.3.

Algorithms Course Notes
Dynamic Programming 4

Ian Parberry*

Fall 2001

Summary

Dynamic programming applied to

- All pairs shortest path problem (Floyd's Algorithm)
- Transitive closure (Warshall's Algorithm)

Constructing solutions using dynamic programming

- All pairs shortest path problem
- Matrix product

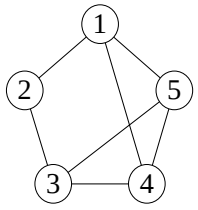
Graphs

A graph is an ordered pair $G = (V, E)$ where

V is a finite set of vertices
$E \subseteq V \times V$ is a set of edges

For example,

$$\begin{aligned} V &= \{1,2,3,4,5\} \\ E &= \{(1,2),(1,4),(1,5),(2,3), \\ &\quad (3,4),(3,5),(4,5)\} \end{aligned}$$

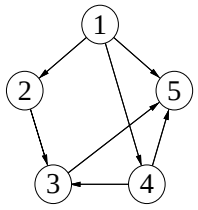


Directed Graphs

A directed graph is a graph with directions on the edges.

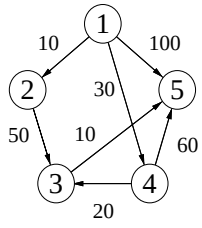
For example,

$$\begin{aligned} V &= \{1,2,3,4,5\} \\ E &= \{(1,2),(1,4),(1,5),(2,3), \\ &\quad (4,3),(3,5),(4,5)\} \end{aligned}$$



Labelled, Directed Graphs

A labelled directed graph is a directed graph with positive costs on the edges.



Applications: cities and distances by road.

Conventions

n is the number of vertices
e is the number of edges

Questions:

What is the maximum number of edges in an undirected graph of n vertices?

What is the maximum number of edges in a directed graph of n vertices?

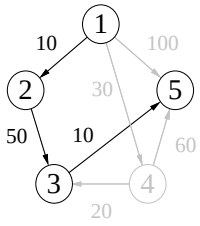
Paths in Graphs

A path in a graph $G = (V, E)$ is a sequence of edges

$$(v_1, v_2), (v_2, v_3), \ldots, (v_n, v_{n+1}) \in E$$

- The length of a path is the number of edges.
- The cost of a path is the sum of the costs of the edges.

For example, $(1, 2), (2, 3), (3, 5)$. Length 3. Cost 70.



All Pairs Shortest Paths

Given a labelled, directed graph $G = (V, E)$, find for each pair of vertices $v, w \in V$ the cost of the shortest (i.e. least cost) path from v to w.

Define A_k to be an $n \times n$ matrix with $A_k[i, j]$ the cost of the shortest path from i to j with internal vertices numbered $\leq k$.

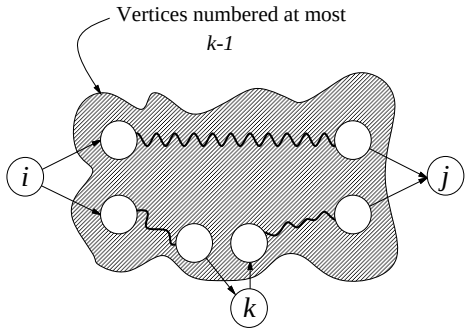
$A_0[i, j]$ equals

- If $i \neq j$ and $(i, j) \in E$, then the cost of the edge from i to j.
- If $i \neq j$ and $(i, j) \notin E$, then ∞.
- If $i = j$, then 0.

Computing A_k

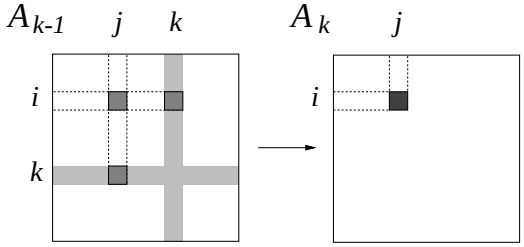
Consider the shortest path from i to j with internal vertices $1..k$. Either:

- It does not go through k, in which place it costs $A_{k-1}[i, j]$.
- It goes through k, in which case it goes through k only once, so it costs $A_{k-1}[i, k] + A_{k-1}[k, j]$. (Only true for positive costs.)



Hence,

$$A_k[i, j] = \min\{A_{k-1}[i, j], A_{k-1}[i, k] + A_{k-1}[k, j]\}$$



All entries in A_k depend upon row k and column k of A_{k-1}.

The entries in row k and column k of A_k are the same as those in A_{k-1}.

Row k:

$$\begin{aligned} A_k[k, j] &= \min\{A_{k-1}[k, j], \\ &\quad A_{k-1}[k, k] + A_{k-1}[k, j]\} \\ &= \min\{A_{k-1}[k, j], 0 + A_{k-1}[k, j]\} \\ &= A_{k-1}[k, j] \end{aligned}$$

Column k:

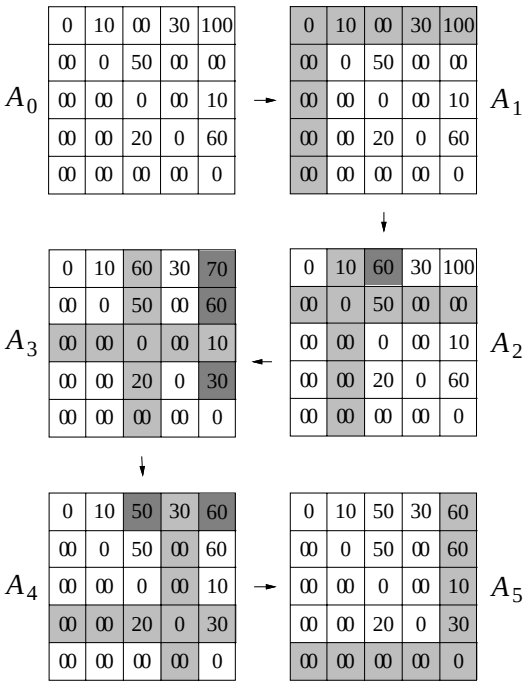
$$\begin{aligned} A_k[i, k] &= \min\{A_{k-1}[i, k], \\ &\quad A_{k-1}[i, k] + A_{k-1}[k, k]\} \\ &= \min\{A_{k-1}[i, k], A_{k-1}[i, k] + 0\} \\ &= A_{k-1}[i, k] \end{aligned}$$

Therefore, we can use the same array.

Floyd's Algorithm

```
for i := 1 to n do
  for j := 1 to n do
    if (i, j) ∈ E
      then A[i, j] := cost of (i, j)
      else A[i, j] := ∞
  A[i, i] := 0
for k := 1 to n do
  for i := 1 to n do
    for j := 1 to n do
      if A[i, k] + A[k, j] < A[i, j] then
        A[i, j] := A[i, k] + A[k, j]
```

Running time: $O(n^3)$.



Storing the Shortest Path

```
for i := 1 to n do
  for j := 1 to n do
  [P[i, j] := 0]
    if (i, j) ∈ E
      then A[i, j] := cost of (i, j)
      else A[i, j] := ∞
  A[i, i] := 0
for k := 1 to n do
  for i := 1 to n do
    for j := 1 to n do
      if A[i, k] + A[k, j] < A[i, j] then
        A[i, j] := A[i, k] + A[k, j]
        [P[i, j] := k]
```

Running time: Still $O(n^3)$.

Note: On termination, $P[i, j]$ contains a vertex on the shortest path from i to j.

Computing the Shortest Path

```
for i := 1 to n do
  for j := 1 to n do
    if A[i, j] < ∞ then
      print(i); shortest(i, j); print(j)

procedure shortest(i, j)
  k := P[i, j]
  if k > 0 then
    shortest(i, k); print(k); shortest(k, j)
```

Claim: Calling procedure shortest(i, j) prints the internal nodes on the shortest path from i to j.

Proof: A simple induction on the length of the path.

Warshall's Algorithm

Transitive closure: Given a directed graph $G = (V, E)$, find for each pair of vertices $v, w \in V$ whether there is a path from v to w.

Solution: make the cost of all edges 1, and run Floyd's algorithm. If on termination $A[i, j] \neq \infty$, then there is a path from i to j.

A cleaner solution: use Boolean values instead.

```
for i := 1 to n do
  for j := 1 to n do
    A[i,j] := (i,j) ∈ E
  A[i,i] := true
for k := 1 to n do
  for i := 1 to n do
    for j := 1 to n do
        A[i,j] := A[i,j] or (A[i,k] and A[k,j])
```

Finding Solutions Using Dynamic Programming

We have seen some examples of finding the cost of the "best" solution to some interesting problems:

- cheapest order of multiplying n rectangular matrices
- min cost binary search tree
- all pairs shortest path

We have also seen some examples of finding whether solutions exist to some interesting problems:

- the knapsack problem
- transitive closure

What about actually finding the solutions?

We saw how to do it with the all pairs shortest path problem.

The principle is the same every time:

- In the inner loop there is some kind of "decision" taken.
- Record the result of this decision in another table.
- Write a divide-and-conquer algorithm to construct the solution from the information in the table.

Matrix Product

As another example, consider the matrix product algorithm.

```
for i := 1 to n do m[i,i] := 0
for d := 1 to n − 1 do
  for i := 1 to n − d do
    j := i + d
    m[i,j] := min_{i≤k<j}(m[i,k] + m[k+1,j]
                          +r_{i−1} r_k r_j)
```

In more detail:

```
for i := 1 to n do m[i,i] := 0
for d := 1 to n − 1 do
  for i := 1 to n − d do
    j := i + d
    m[i,j] := m[i,i] + m[i+1,j] + r_{i−1} r_i r_j
    for k := i + 1 to j − 1 do
      if m[i,k] + m[k+1,j] + r_{i−1} r_k r_j < m[i,j]
        then
          m[i,j] := m[i,k] + m[k+1,j] + r_{i−1} r_k r_j
```

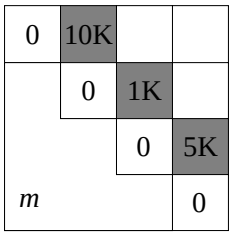
Add the extra lines:

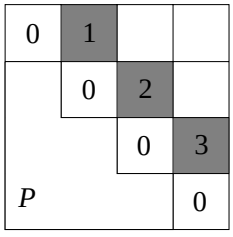
```
for i := 1 to n do m[i,i] := 0
for d := 1 to n − 1 do
  for i := 1 to n − d do
    j := i + d
    m[i,j] := m[i,i] + m[i+1,j] + r_{i−1} r_i r_j
    [P[i,j] := i]
    for k := i + 1 to j − 1 do
      if m[i,k] + m[k+1,j] + r_{i−1} r_k r_j < m[i,j]
        then
          m[i,j] := m[i,k] + m[k+1,j] + r_{i−1} r_k r_j
          [P[i,j] := k]
```

Example

$r_0 = 10$, $r_1 = 20$, $r_2 = 50$, $r_3 = 1$, $r_4 = 100$.

$$
\begin{aligned}
m[1,2] &= \min_{1 \le k < 2}(m[1,k] + m[k+1,2] + r_0 r_k r_2) \\
&= r_0 r_1 r_2 = 10000 \\
m[2,3] &= r_1 r_2 r_3 = 1000 \\
m[3,4] &= r_2 r_3 r_4 = 5000
\end{aligned}
$$





$m[1,3]$

$$
\begin{aligned}
&= \min_{1\le k<3}(m[1,k]+m[k+1,3]+r_0r_kr_3)\\
&= \min(m[1,1]+m[2,3]+r_0r_1r_3,\\
&\qquad m[1,2]+m[3,3]+r_0r_2r_3)\\
&= \min(0+1000+200, 10000+0+500)\\
&= 1200
\end{aligned}
$$

0	10K	1.2K	
	0	1K	
		0	5K
m			0

0	1	1	
	0	2	
		0	3
P			0

$$
\begin{aligned}
&m[2,4]\\
&= \min_{2\le k<4}(m[2,k]+m[k+1,4]+r_1r_kr_4)\\
&= \min(m[2,2]+m[3,4]+r_1r_2r_4,\\
&\qquad m[2,3]+m[4,4]+r_1r_3r_4)\\
&= \min(0+5000+100000, 1000+0+2000)\\
&= 3000
\end{aligned}
$$

0	10K	1.2K	
	0	1K	3K
		0	5K
m			0

0	1	1	
	0	2	3
		0	3
P			0

$$
\begin{aligned}
&m[1,4]\\
&= \min_{1\le k<4}(m[1,k]+m[k+1,4]+r_0r_kr_4)\\
&= \min(m[1,1]+m[2,4]+r_0r_1r_4,\\
&\qquad m[1,2]+m[3,4]+r_0r_2r_4,\\
&\qquad m[1,3]+m[4,4]+r_0r_3r_4)\\
&= \min(0+3000+20000,\\
&\qquad 1000+5000+50000,\\
&\qquad 1200+0+1000)\\
&= 2200
\end{aligned}
$$

0	10K	1.2K	2.2K
	0	1K	3K
		0	5K
m			0

0	1	1	3
	0	2	3
		0	3
P			0

Performing the Product

Suppose we have a function matmultiply that multiplies two rectangular matrices and returns the result.

```
function product(i, j)
  comment returns M_i · M_{i+1} ··· M_j
  if i = j then return(M_i) else
    k := P[i, j]
    return(matmultiply(product(i, k),
        product(k + 1, j)))
```

Main body: product$(1, n)$.

Parenthesizing the Product

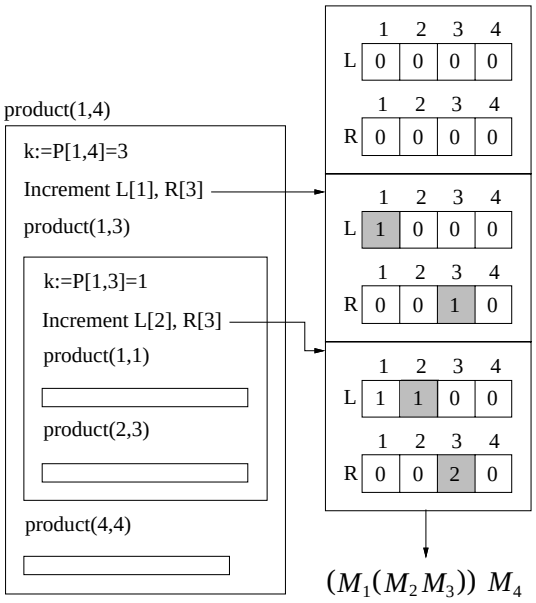
Suppose we want to output the parenthesization instead of performing it. Use arrays L, R:

- $L[i]$ stores the number of left parentheses in front of matrix M_i
- $R[i]$ stores the number of right parentheses behind matrix M_i.

```
for i := 1 to n do
  L[i] := 0; R[i] := 0
product(1, n)
for i := 1 to n do
  for j := 1 to L[i] do print("(")
  print("M_i")
  for j := 1 to R[i] do print(")")
```

```
procedure product(i, j)
  if i < j − 1 then
    k := P[i, j]
    if k > i then
      increment L[i] and R[k]
    if k + 1 < j then
      increment L[k + 1] and R[j]
    product(i,k)
    product(k+1,j)
```

Example



Assigned Reading

CLR Section 16.1, 26.2.

POA Section 8.4, 8.6.

Algorithms Course Notes
Greedy Algorithms 1

Ian Parberry*

Fall 2001

Summary

Greedy algorithms for

- Optimal tape storage
- Continuous knapsack problem

Greedy Algorithms

- Start with a solution to a small subproblem
- Build up to a solution to the whole problem
- Make choices that look good in the short term

Disadvantage: Greedy algorithms don't always work. (Short term solutions can be disastrous in the long term.) Hard to prove correct.

Advantage: Greedy algorithms work fast when they work. Simple algorithms, easy to implement.

Optimal Tape Storage

Given n files of length

$$m_1, m_2, \ldots, m_n$$

find which order is the best to store them on a tape, assuming

- Each retrieval starts with the tape rewound.
- Each retrieval takes time equal to the length of the preceding files in the tape plus the length of the retrieved file.
- All files are to be retrieved.

Example

$n = 3$; $m_1 = 5$, $m_2 = 10$, $m_3 = 3$.

There are $3! = 6$ possible orders.

1,2,3:	(5+10+3)+(5+10)+5 =38
1,3,2:	(5+3+10)+(5+3) +5 =31
2,1,3:	(10+5+3)+(10+5)+10=43
2,3,1:	(10+3+5)+(10+3)+10=41
3,1,2:	(3+5+10)+(3+5) +3 =29
3,2,1:	(3+10+5)+(3+10)+3 =34

The best order is $3, 1, 2$.

The Greedy Solution

make tape empty
for $i := 1$ **to** n **do**
  grab the next shortest file
  put it next on tape

The algorithm takes the best short term choice without checking to see whether it is the best long term decision.

Is this wise?

Correctness

Suppose we have files $f_1, f_2, \ldots, f_n$, of lengths $m_1, m_2, \ldots, m_n$ respectively. Let $i_1, i_2, \ldots, i_n$ be a permutation of $1, 2, \ldots, n$.

Suppose we store the files in order

$$f_{i_1}, f_{i_2}, \ldots, f_{i_n}.$$

What does this cost us?

To retrieve the kth file on the tape, f_{i_k}, costs

$$\sum_{j=1}^{k} m_{i_j}$$

Therefore, the cost of retrieving them all is

$$\sum_{k=1}^{n}\sum_{j=1}^{k} m_{i_j} = \sum_{k=1}^{n}(n-k+1)m_{i_k}$$

To see this:

$$\begin{array}{ll} f_{i_1}: & m_{i_1} \\ f_{i_2}: & m_{i_1}+m_{i_2} \\ f_{i_3}: & m_{i_1}+m_{i_2}+m_{i_3} \\ & \vdots \\ f_{i_{n-1}}: & m_{i_1}+m_{i_2}+m_{i_3}+\ldots+m_{i_{n-1}} \\ f_{i_n}: & m_{i_1}+m_{i_2}+m_{i_3}+\ldots+m_{i_{n-1}}+m_{i_n} \end{array}$$

Total is

$$nm_{i_1} + (n-1)m_{i_2} + (n-2)m_{i_3} + \ldots + 2m_{i_{n-1}} + m_{i_n} = \sum_{k=1}^{n}(n-k+1)m_{i_k}$$

The greedy algorithm picks files f_i in nondecreasing order of their size m_i. It remains to prove that this is the minimum cost permutation.

Claim: Any permutation in nondecreasing order of m_i's has minimum cost.

Proof: Let $\Pi = (i_1, i_2, \ldots, i_n)$ be a permutation of $1, 2, \ldots, n$ that is not in nondecreasing order of m_i's. We will prove that it cannot have minimum cost.

Since $m_{i_1}, m_{i_2}, \ldots, m_{i_n}$ is not in nondecreasing order, there must exist $1 \leq j < n$ such that $m_{i_j} > m_{i_{j+1}}$.

Let Π' be permutation Π with i_j and i_{j+1} swapped.

The cost of permutation Π is

$$\begin{aligned} C(\Pi) &= \sum_{k=1}^{n}(n-k+1)m_{i_k} \\ &= \sum_{k=1}^{j-1}(n-k+1)m_{i_k} + \\ &\quad (n-j+1)m_{i_j} + (n-j)m_{i_{j+1}} + \\ &\quad \sum_{k=j+2}^{n}(n-k+1)m_{i_k} \end{aligned}$$

The cost of permutation Π' is

$$\begin{aligned} C(\Pi') = &\sum_{k=1}^{j-1}(n-k+1)m_{i_k} + \\ &(n-j+1)m_{i_{j+1}} + (n-j)m_{i_j} + \\ &\sum_{k=j+2}^{n}(n-k+1)m_{i_k} \end{aligned}$$

Hence,

$$\begin{aligned} C(\Pi) - C(\Pi') &= (n-j+1)(m_{i_j} - m_{i_{j+1}}) \\ &\quad + (n-j)(m_{i_{j+1}} - m_{i_j}) \\ &= m_{i_j} - m_{i_{j+1}} \\ &> 0 \ \text{(by definition of } j) \end{aligned}$$

Therefore, $C(\Pi') < C(\Pi)$, and so Π cannot be a permutation of minimum cost. This is true of any Π that is not in nondecreasing order of m_i's.

Therefore the minimum cost permutation must be in nondecreasing order of m_i's.

Analysis

$O(n \log n)$ for sorting

$O(n)$ for the rest

The Continuous Knapsack Problem

This is similar to the knapsack problem met earlier.

- given n objects $A_1, A_2, \ldots, A_n$
- given a knapsack of length S
- A_i has length s_i
- A_i has weight w_i
- an x_i-fraction of A_i, where $0 \leq x_i \leq 1$ has length $x_i s_i$ and weight $x_i w_i$

Fill the knapsack as full as possible using fractional parts of the objects, so that the weight is minimized.

Example

$S = 20$.

$s_1 = 18$, $s_2 = 10$, $s_3 = 15$.

$w_1 = 25$, $w_2 = 15$, $w_3 = 24$.

(x_1, x_2, x_3)	$\sum_{i=1}^{3} x_i s_i$	$\sum_{i=1}^{3} x_i w_i$
(1,1/5,0)	20	28
(1,0,2/15)	20	28.2
(0,1,2/3)	20	31
(0,1/2,1)	20	31.5
	$\vdots$	

The first one is the best so far. But is it the best overall?

The Greedy Solution

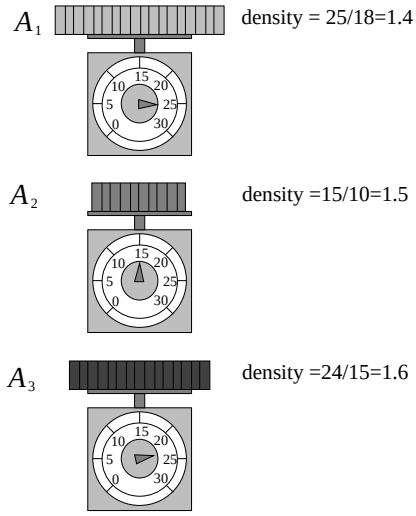
Define the density of object A_i to be w_i/s_i. Use as much of low density objects as possible. That is, process each in increasing order of density. If the whole thing fits, use all of it. If not, fill the remaining space with a fraction of the current object, and discard the rest.

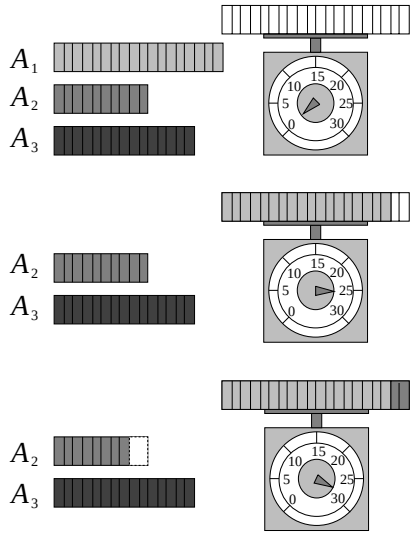
First, sort the objects in nondecreasing density, so that $w_i/s_i \leq w_{i+1}/s_{i+1}$ for $1 \leq i < n$.

Then, do the following:

```
s := S; i := 1
while s_i ≤ s do
  x_i := 1
  s := s − s_i
  i := i + 1
x_i := s/s_i
for j := i + 1 to n do
  x_j := 0
```

Example





Correctness

Claim: The greedy algorithm gives a solution of minimal weight.

(Note: a solution of minimal weight. There may be many solutions of minimal weight.)

Proof: Let $X = (x_1, x_2, \ldots, x_n)$ be the solution generated by the greedy algorithm.

If all the x_i are 1, then the solution is clearly optimal (it is the only solution).

Otherwise, let j be the smallest number such that

$x_j \neq 1$. From the algorithm,

$$\begin{aligned} x_i &= 1 \quad \text{for} \quad 1 \leq i < j \\ 0 \leq x_j &< 1 \\ x_i &= 0 \quad \text{for} \quad j < i \leq n \end{aligned}$$

Therefore,

$$\sum_{i=1}^{j} x_i s_i = S.$$

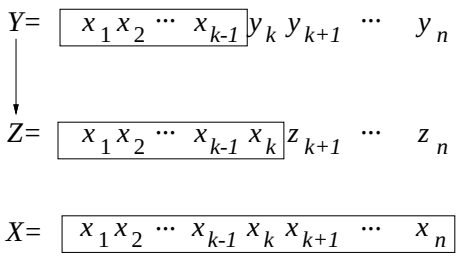
Let $Y = (y_1, y_2, \ldots, y_n)$ be a solution of minimal weight. We will prove that X must have the same weight as Y, and hence has minimal weight.

If $X = Y$, we are done. Otherwise, let k be the least number such that $x_k \neq y_k$.

Proof Stategy

Transform Y into X, maintaining weight.

We will see how to transform Y into Z, which looks "more like" X.



Digression

It must be the case that $y_k < x_k$. To see this, consider the three possible cases:

Case 1: $k < j$. Then $x_k = 1$. Therefore, since $x_k \neq y_k$, y_k must be smaller than x_k.

Case 2: $k = j$. By the definition of k, $x_k \neq y_k$. If $y_k > x_k$,

$$\begin{aligned} S &= \sum_{i=1}^{n} y_i s_i \\ &= \sum_{i=1}^{k-1} y_i s_i + y_k s_k + \sum_{i=k+1}^{n} y_i s_i \\ &= \sum_{i=1}^{k-1} x_i s_i + y_k s_k + \sum_{i=k+1}^{n} y_i s_i \\ &= \sum_{i=1}^{k} x_i s_i + (y_k - x_k) s_k + \sum_{i=k+1}^{n} y_i s_i \\ &= \sum_{i=1}^{j} x_i s_i + (y_k - x_k) s_k + \sum_{i=k+1}^{n} y_i s_i \\ &= S + (y_k - x_k) s_k + \sum_{i=k+1}^{n} y_i s_i \\ &> S \end{aligned}$$

which contradicts the fact that Y is a solution. Therefore, $y_k < x_k$.

Case 3: $k > j$. Then $x_k = 0$ and $y_k > 0$, and so

$$\begin{aligned} S &= \sum_{i=1}^{n} y_i s_i \\ &= \sum_{i=1}^{j} y_i s_i + \sum_{i=j+1}^{n} y_i s_i \\ &= \sum_{i=1}^{j} x_i s_i + \sum_{i=j+1}^{n} y_i s_i \\ &= S + \sum_{i=j+1}^{n} y_i s_i \\ &> S \end{aligned}$$

This is not possible, hence Case 3 can never happen.

In the other 2 cases, $y_k < x_k$ as claimed.

Back to the Proof

Now suppose we increase y_k to x_k, and decrease as many of $y_{k+1}, \ldots, y_n$ as necessary to make the total length remain at S. Call this new solution $Z = (z_1, z_2, \ldots, z_n)$.

Therefore,

1. $(z_k - y_k)s_k > 0$
2. $\sum_{i=k+1}^{n} (z_i - y_i)s_i < 0$

3. $(z_k - y_k)s_k + \sum_{i=k+1}^{n}(z_i - y_i)s_i = 0$

Then,

$$\begin{aligned}
& \sum_{i=1}^{n} z_i w_i \\
= & \sum_{i=1}^{k-1} z_i w_i + z_k w_k + \sum_{i=k+1}^{n} z_i w_i \\
= & \sum_{i=1}^{k-1} y_i w_i + z_k w_k + \sum_{i=k+1}^{n} z_i w_i \\
= & \sum_{i=1}^{n} y_i w_i - y_k w_k - \sum_{i=k+1}^{n} y_i w_i + \\
& z_k w_k + \sum_{i=k+1}^{n} z_i w_i \\
= & \sum_{i=1}^{n} y_i w_i + (z_k - y_k) w_k + \sum_{i=k+1}^{n} (z_i - y_i) w_i \\
= & \sum_{i=1}^{n} y_i w_i + (z_k - y_k) s_k w_k / s_k + \\
& \sum_{i=k+1}^{n} (z_i - y_i) s_i w_i / s_i
\end{aligned}$$

$$\begin{aligned}
\leq & \sum_{i=1}^{n} y_i w_i + (z_k - y_k) s_k w_k / s_k + \\
& \sum_{i=k+1}^{n} (z_i - y_i) s_i w_k / s_k \text{ (by (2) \& density)} \\
= & \sum_{i=1}^{n} y_i w_i + \\
& \left((z_k - y_k) s_k + \sum_{i=k+1}^{n} (z_i - y_i) s_i \right) w_k / s_k \\
= & \sum_{i=1}^{n} y_i w_i \text{ (by (3))}
\end{aligned}$$

Now, since Y is a minimal weight solution,

$$\sum_{i=1}^{n} z_i w_i \not< \sum_{i=1}^{n} y_i w_i.$$

Hence Y and Z have the same weight. But Z looks "more like" X — the first k entries of Z are the same as X.

Repeating this procedure transforms Y into X and maintains the same weight. Therefore X has minimal weight.

Analysis

$O(n \log n)$ for sorting

$O(n)$ for the rest

Assigned Reading

CLR Section 17.2.

POA Section 9.1.

Algorithms Course Notes
Greedy Algorithms 2

Ian Parberry*

Fall 2001

Summary

A greedy algorithm for

- The single source shortest path problem (Dijkstra's algorithm)

Single Source Shortest Paths

Given a labelled, directed graph $G = (V, E)$ and a distinguished vertex $s \in V$, find for each vertex $w \in V$ a shortest (i.e. least cost) path from s to w.

More precisely, construct a data structure that allows us to compute the vertices on a shortest path from s to w for any $w \in V$ in time linear in the length of that path.

Vertex s is called the source.

Let $\delta(v, w)$ denote the cost of the shortest path from v to w.

Let $C[v, w]$ be the cost of the edge from v to w (∞ if it doesn't exist, 0 if $v = w$).

Dijkstra's Algorithm: Overview

Maintain a set S of vertices whose minimum distance from the source is known.

- Initially, $S = \{s\}$.
- Keep adding vertices to S.
- Eventually, $S = V$.

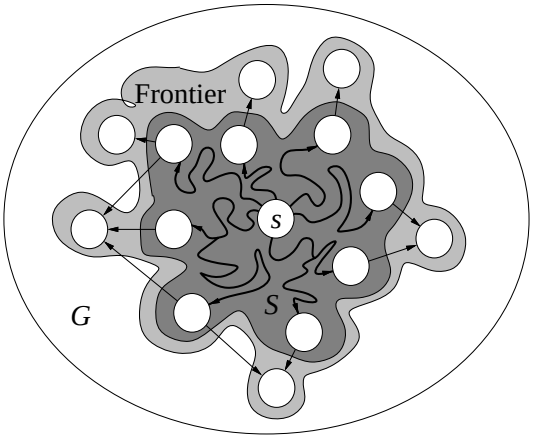
A internal path is a path from s with all internal vertices in S.

Maintain an array D with:

- If $w \notin S$, then $D[w]$ is the cost of the shortest internal path to w.
- If $w \in S$, then $D[w]$ is the cost of the shortest path from s to w, $\delta(s, w)$.

The Frontier

A vertex w is on the frontier if $w \notin S$, and there is a vertex $u \in S$ such that $(u, w) \in E$.



All internal paths end at vertices in S or the frontier.

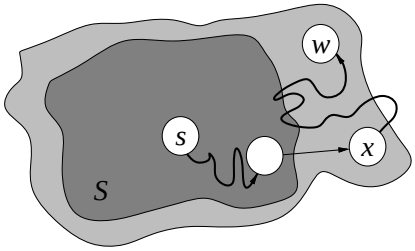
Adding a Vertex to S

Which vertex do we add to S? The vertex $w \in V - S$ with smallest D value. Since the D value of vertices not on the frontier or in S is infinite, w will be on the frontier.

Claim: $D[w] = \delta(s, w)$.

That is, we are claiming that the shortest internal path from s to w is a shortest path from s to w.

Consider the shortest path from s to w. Let x be the first frontier vertex it meets.



Then

$$\begin{aligned}\delta(s,w) &= \delta(s,x)+\delta(x,w)\\ &= D[x]+\delta(x,w)\\ &\geq D[w]+\delta(x,w) \text{ (By defn. of } w)\\ &\geq D[w]\end{aligned}$$

But by the definition of δ, $\delta(s,w) \leq D[w]$.

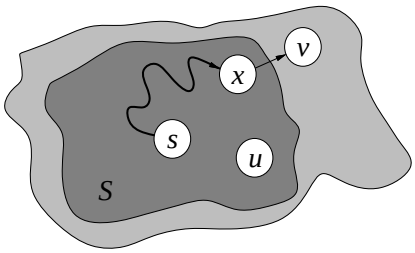
Hence, $D[w] = \delta(s,w)$ as claimed.

Observation

Claim: If u is placed into S before v, then $\delta(s,u) \leq \delta(s,v)$.

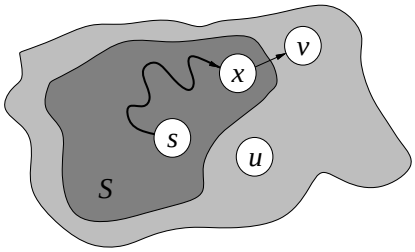
Proof: Proof by induction on the number of vertices already in S when v is put in. The hypothesis is certainly true when $|S| = 1$. Now suppose it is true when $|S| < n$, and consider what happens when $|S| = n$.

Consider what happens at the time that v is put into S. Let x be the last internal vertex on the shortest path to v.



Case 1: x was put into S before u.

Go back to the time that u was put into S.



Since x was already in S, v was on the frontier. But since u was chosen to be put into S before v, it must have been the case that $D[u] \leq D[v]$. Since u was chosen to be put into S, $D[u] = \delta(s,u)$. By the definition of x, and because x was put into S before u, $D[v] = \delta(s,v)$. Therefore, $\delta(s,u) = D[u] \leq D[v] = \delta(s,v)$.

Case 2: x was put into S after u.

Since x was put into S before v, at the time x was put into S, $|S| < n$. Therefore, by the induction hypothesis, $\delta(s,u) \leq \delta(s,x)$. Therefore, by the definition of x, $\delta(s,u) \leq \delta(s,x) \leq \delta(s,v)$.

Updating the Cost Array

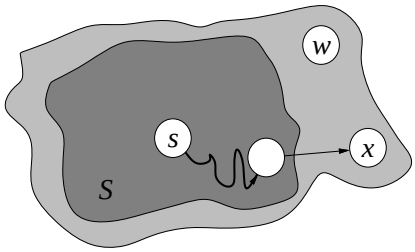
The D value of frontier vertices may change, since there may be new internal paths containing w.

There are 2 ways that this can happen. Either,

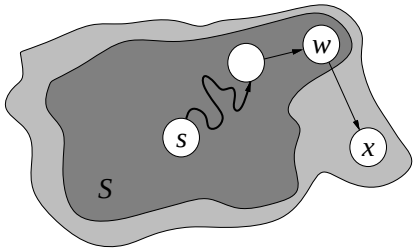
- w is the last internal vertex on a new internal path, or
- w is some other internal vertex on a new internal path

If w is the last internal vertex:

Old internal path of cost $D[x]$

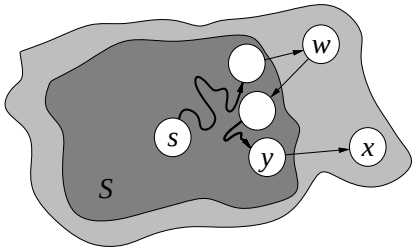


New internal path of cost $D[w]+C[w,x]$

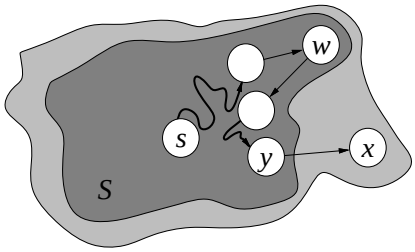


If w is not the last internal vertex:

Was a non-internal path



Becomes an internal path

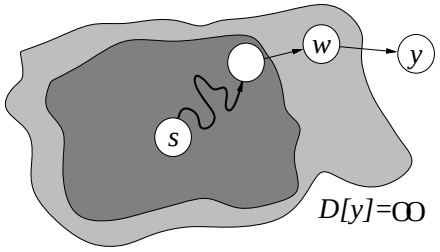


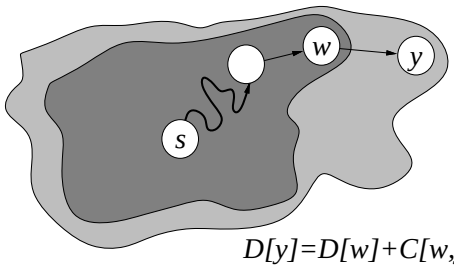
This cannot be shorter than any other path from s to x.

Vertex y was put into S before w. Therefore, by the earlier Observation, $\delta(s, y) \leq \delta(s, w)$.

That is, it is cheaper to go from s to y directly than it is to go through w. So, this type of path can be ignored.

The D value of vertices outside the frontier may change





The D value of vertices in S do not change (they are already the shortest).

Therefore, for every vertex $v \in V$, when w is moved from the frontier to S,

$$D[v] := \min\{D[v], D[w] + C[w, v]\}$$

Dijkstra's Algorithm

1. $S := \{s\}$
2. **for** each $v \in V$ **do**
3. $\quad D[v] := C[s, v]$
4. **for** $i := 1$ **to** $n - 1$ **do**
5. $\quad$ choose $w \in V - S$ with smallest $D[w]$
6. $\quad S := S \cup \{w\}$
7. $\quad$ **for** each vertex $v \in V$ **do**
8. $\quad\quad D[v] := \min\{D[v], D[w] + C[w, v]\}$

First Implementation

Store S as a bit vector.

- Line 1: initialize the bit vector, $O(n)$ time.
- Lines 2–3: n iterations of $O(1)$ time per iteration, total of $O(n)$ time
- Line 4: $n - 1$ iterations
- Line 5: a linear search, $O(n)$ time.
- Line 6: $O(1)$ time

- Lines 7–8: n iterations of $O(1)$ time per iteration, total of $O(n)$ time
- Lines 4–8: $n-1$ iterations, $O(n)$ time per iteration

Total time $O(n^2)$.

Second Implementation

Instead of storing S, store $S' = V - S$. Implement S' as a heap, indexed on D value.

Line 8 only needs to be done for those v such that $(w, v) \in E$. Provide a list for each $w \in V$ of those vertices v such that $(w, v) \in E$ (an adjacency list).

```
1–3.  makenull(S')
      for each v ∈ V except s do
        D[v] := C[s, v]
        insert(v, S')
4.    for i := 1 to n − 1 do
5–6.    w := deletemin(S')
7.      for each v such that (w, v) ∈ E do
8.        D[v] := min{D[v], D[w] + C[w, v]}
          move v up the heap
```

Analysis

- Line 8: $O(\log n)$ to adjust the heap
- Lines 7–8: $O(n \log n)$
- Lines 5–6: $O(\log n)$
- Lines 4–8: $O(n^2 \log n)$
- Lines 1–3: $O(n \log n)$

Total: $O(n^2 \log n)$

But, line 8 is executed exactly once for each edge: each edge $(u, v) \in E$ is used once when u is placed in S. Therefore, the true complexity is $O(e \log n)$:

- Lines 1–3: $O(n \log n)$
- Lines 4–6: $O(n \log n)$
- Lines 4,8: $O(e \log n)$

Dense and Sparse Graphs

Which is best? They match when $e \log n = \Theta(n^2)$, that is, $e = \Theta(n^2 / \log n)$.

- use first implementation on dense graphs (e is larger than $n^2 / \log n$, technically, $e = \omega(n^2 / \log n)$)
- use second implementation on sparse graphs (e is smaller than $n^2 / \log n$, technically, $e = o(n^2 / \log n)$)

The second implementation can be improved by implementing the priority queue using *Fibonacci heaps*. Run time is $O(n \log n + e)$.

Computing the Shortest Paths

We have only computed the costs of the shortest paths. What about constructing them?

Keep an array P, with $P[v]$ the predecessor of v in the shortest internal path.

Every time $D[v]$ is modified in line 8, set $P[v]$ to w.

```
1.  S := {s}
2.  for each v ∈ V do
3.    D[v] := C[s, v]
      if (s, v) ∈ E then P[v] := s else P[v] := 0
4.  for i := 1 to n − 1 do
5.    choose w ∈ V − S with smallest D[w]
6.    S := S ∪ {w}
7.    for each vertex v ∈ V − S do
8.      if D[w] + C[w, v] < D[v] then
          D[v] := D[w] + C[w, v]
          P[v] := w
```

Reconstructing the Shortest Paths

For each vertex v, we know that $P[v]$ is its predecessor on the shortest path from s to v. Therefore, the shortest path from s to v is:

- The shortest path from s to $P[v]$, and
- the edge $(P[v], v)$.

To print the list of vertices on the shortest path from s to v, use divide-and-conquer:

```
procedure path(s, v)
  if v ≠ s then path(s, P[v])
  print(v)
```

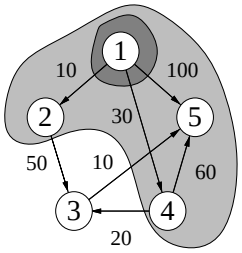
To use, do the following:

if $P[v] \neq 0$ **then** path(s, v)

Correctness: Proof by induction.

Analysis: linear in the length of the path; proof by induction.

Example

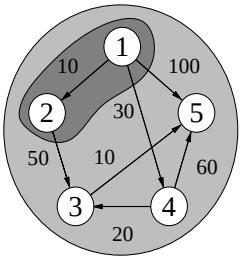


Dark shading: S
Light shading: frontier

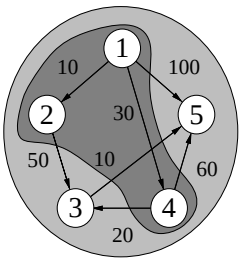
Source is vertex 1.

	2	3	4	5
D	10	∞	30	100
P	1	0	1	1

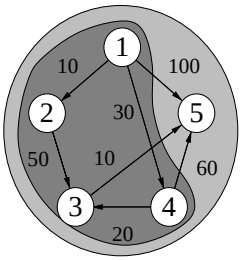
Dark shading: S
Light shading: entries that have changed



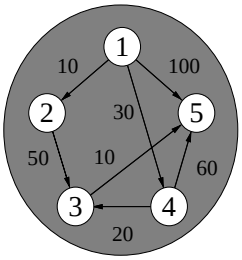
	2	3	4	5
D	10	60	30	100
P	1	2	1	1



	2	3	4	5
D	10	50	30	90
P	1	4	1	4



	2	3	4	5
D	10	50	30	60
P	1	4	1	3



	2	3	4	5
D	10	50	30	60
P	1	4	1	3

To reconstruct the shortest paths from the P array:

$P[2]$	$P[3]$	$P[4]$	$P[5]$
1	4	1	3

Vertex	Path	Cost
2:	1 2	10
3:	1 4 3	20 + 30 = 50
4:	1 4	30
5:	1 4 3 5	10 + 20 + 30 = 60

The costs agree with the D array.

The same example was used for the all pairs shortest path problem.

Assigned Reading

CLR Section 25.1, 25.2.

Algorithms Course Notes
Greedy Algorithms 3

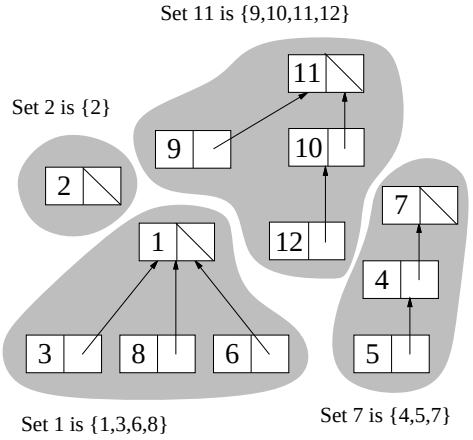
Ian Parberry*

Fall 2001

Summary

Greedy algorithms for

- The union-find problem
- Min cost spanning trees (Prim's algorithm and Kruskal's algorithm)

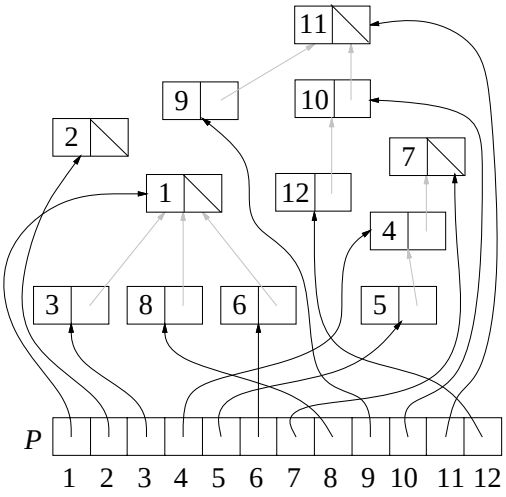


The Union-Find Problem

Given a set $\{1, 2, \ldots, n\}$ initially partitioned into n disjoint subsets, one member per subset, we want to perform the following operations:

- find(x): return the name of the subset that x is in
- union(x, y): combine the two subsets that x and y are in

What do we use for the "name" of a subset? Use one of its members.



A Data Structure for Union-Find

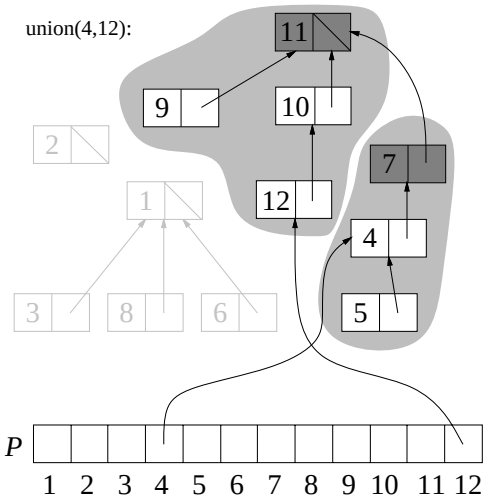
Use a tree:

- implement with pointers and records
- the set elements are stored in the nodes
- each child has a pointer to its parent
- there is an array $P[1..n]$ with $P[i]$ pointing to the node containing i

Implementing the Operations

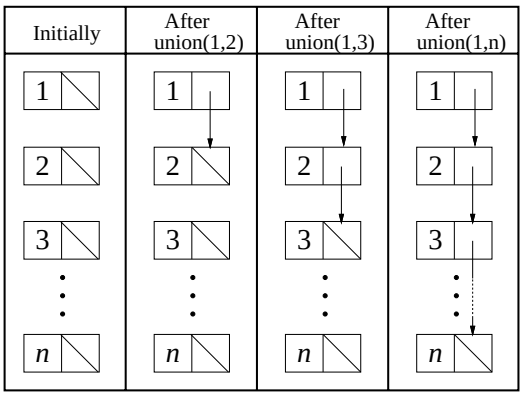
- find(x): Follow the chain of pointers starting at $P[x]$ to the root of the tree containing x. Return the set element stored there.
- union(x, y): Follow the chains of pointers starting at $P[x]$ and $P[y]$ to the roots of the trees containing x and y, respectively. Make the root of one tree point to the root of the other.

Example



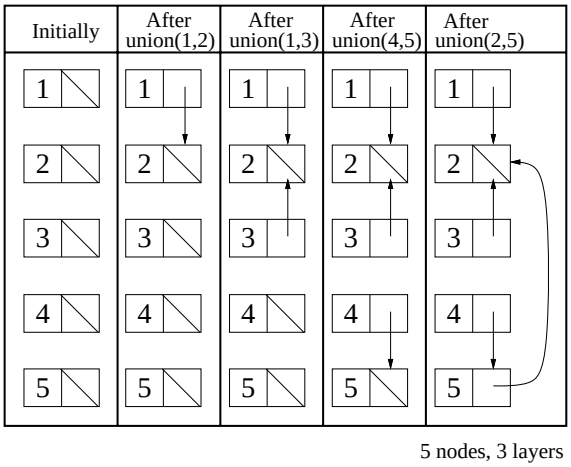
Analysis

Running time proportional to number of layers in the tree. But this may be $\Omega(n)$ for an n-node tree.



Improvement

Make the root of the smaller tree point to the root of the bigger tree.



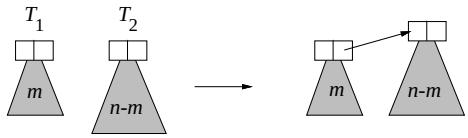
More Analysis

Claim: If this algorithm is used, every tree with n nodes has at most $\lfloor \log n \rfloor + 1$ layers.

Proof: Proof by induction on n, the number of nodes in the tree. The claim is clearly true for a tree with 1 node. Now suppose the claim is true for trees with less than n nodes, and consider a tree T with n nodes.

T was made by unioning together two trees with less than n nodes each.

Suppose the two trees have m and $n-m$ nodes each, where $m \leq n/2$.



Then, by the induction hypothesis, T has

$$\begin{aligned} & \max\{\text{depth}(T_1)+1, \text{depth}(T_2)\} \\ = \; & \max\{\lfloor \log m \rfloor + 2, \lfloor \log(n-m) \rfloor + 1\} \\ \leq \; & \max\{\lfloor \log n/2 \rfloor + 2, \lfloor \log n \rfloor + 1\} \\ = \; & \lfloor \log n \rfloor + 1 \end{aligned}$$

layers.

Implementation detail: store count of nodes in root of each tree. Update during union in $O(1)$ extra time.

Conclusion: union and find operations take time $O(\log n)$.

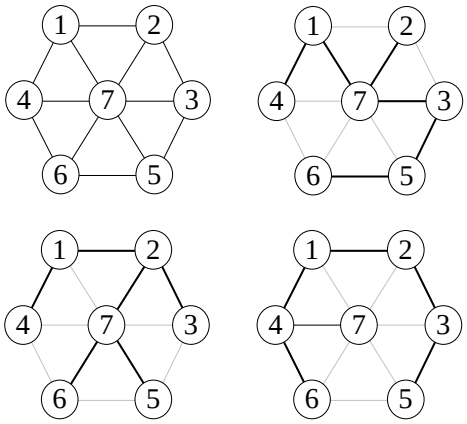
It is possible to do better using *path compression*. When traversing a path from leaf to root, make all nodes point to root. Gives better amortized performance.

$O(\log n)$ is good enough for our purposes.

Spanning Trees

A graph $S = (V, T)$ is a spanning tree of an undirected graph $G = (V, E)$ if:

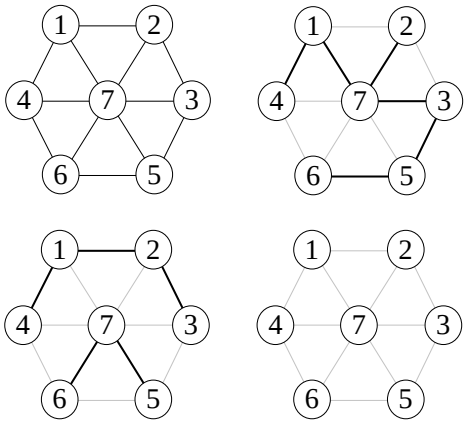
- S is a tree, that is, S is connected and has no cycles
- $T \subseteq E$



Spanning Forests

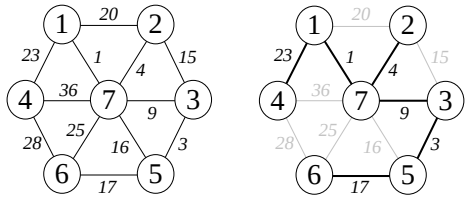
A graph $S = (V, T)$ is a spanning forest of an undirected graph $G = (V, E)$ if:

- S is a forest, that is, S has no cycles
- $T \subseteq E$



Min Cost Spanning Trees

Given a labelled, undirected, connected graph G, find a spanning tree for G of minimum cost.



Cost = 23+1+4+9+3+17=57

The Muddy City Problem

The residents of Muddy City are too cheap to pave their streets. (After all, who likes to pay taxes?) However, after several years of record rainfall they are tired of getting muddy feet. They are still too miserly to pave all of the streets, so they want to pave only enough streets to ensure that they can travel from every intersection to every other intersection on a paved route, and they want to spend as little money as possible doing it. (The residents of Muddy City don't mind walking a long way to save money.)

Solution: Create a graph with a node for each intersection, and an edge for each street. Each edge is labelled with the cost of paving the corresponding street. The cheapest solution is a min-cost spanning tree for the graph.

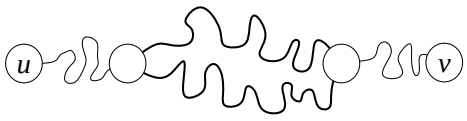
An Observation About Trees

Claim: Let $S = (V, T)$ be a tree. Then:

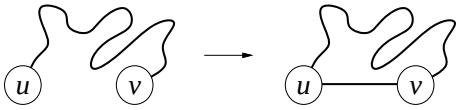
1. For every $u, v \in V$, the path between u and v in S is unique.
2. If any edge $(u, v) \notin T$ is added to S, then a unique cycle results.

Proof:

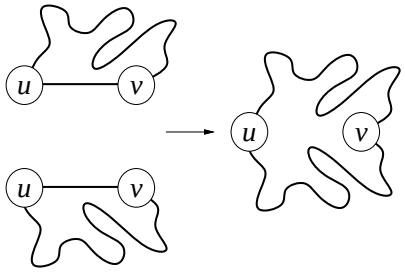
1. If there were more than one path between u and v, then there would be a cycle, which contradicts the definition of a tree.



2. Consider adding an edge $(u, v) \notin T$ to T. There must already be a path from u to v (since a tree is connected). Therefore, adding an edge from u to v creates a cycle.



This cycle must be unique, since if adding (u, v) creates two cycles, then there must have been a cycle before, which contradicts the definition of a tree.



Key Fact

Here is the principle behind MCST algorithms:

Claim: Let $G = (V, E)$ be a labelled, undirected graph, and $S = (V, T)$ a spanning forest for G.

Suppose S is comprised of trees

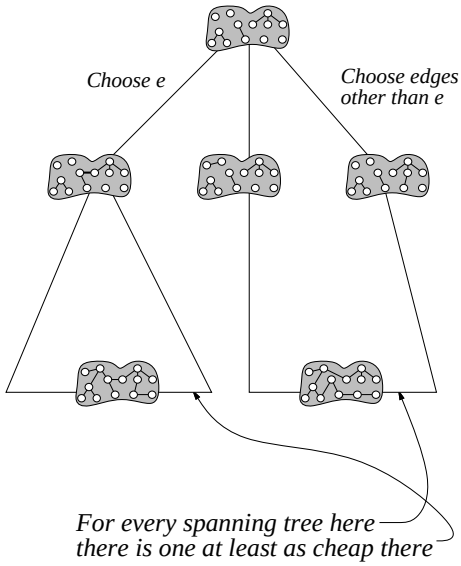
$$(V_1, T_1), (V_2, T_2), \ldots, (V_k, T_k)$$

for some $k \leq n$.

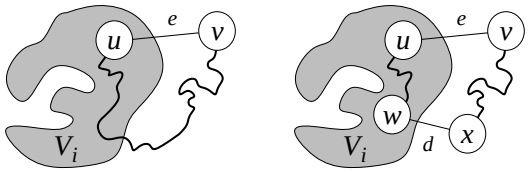
Let $1 \leq i \leq k$. Suppose $e = (u, v)$ is the cheapest edge leaving V_i (that is, an edge of lowest cost in $E - T$ such that $u \in V_i$ and $v \notin V_i$).

Consider all of the spanning trees that contain T. There is a spanning tree that includes $T \cup \{e\}$ that is as cheap as any of them.

Decision Tree



Proof: Suppose there is a spanning tree that includes T but does not include e. Adding e to this spanning tree introduces a cycle.



Therefore, there must be an edge $d = (w, x)$ such that $w \in V_i$ and $x \notin V_i$.

Let $c(e)$ denote the cost of e and $c(d)$ the cost of d. By hypothesis, $c(e) \leq c(d)$.

There is a spanning tree that includes $T \cup \{e\}$ that is at least as cheap as this tree. Simply add edge e and delete edge d.

- Is the new spanning tree really a spanning tree? Yes, because adding e introduces a unique new cycle, and deleting d removes it.
- Is it more expensive? No, because $c(e) \leq c(d)$.

Corollary

Claim: Let $G = (V, E)$ be a labelled, undirected graph, and $S = (V, T)$ a spanning forest for G.

Suppose S is comprised of trees

$$(V_1, T_1), (V_2, T_2), \ldots, (V_k, T_k)$$

for some $k \leq n$.

Let $1 \leq i \leq k$. Suppose $e = (u, v)$ is an edge of lowest cost in $E - T$ such that $u \in V_i$ and $v \notin V_i$.

If S can be expanded to a min cost spanning tree, then so can $S' = (V, T \cup \{e\})$.

Proof: Suppose S can be expanded to a min cost spanning tree M. By the previous result, $S' = (V, T \cup \{e\})$ can be expanded to a spanning tree that is at least as cheap as M, that is, a min cost spanning tree.

General Algorithm

1. $F :=$ set of single-vertex trees
2. **while** there is > 1 tree in F **do**
3. Pick an arbitrary tree $S_i = (V_i, T_i)$ from F
4. $e :=$ min cost edge leaving V_i
5. Join two trees with e

1. $F := \emptyset$
   for $j := 1$ **to** n **do**
   $V_j := \{j\}$
   $F := F \cup \{(V_j, \emptyset)\}$
2. **while** there is > 1 tree in F **do**
3. Pick an arbitrary tree $S_i = (V_i, T_i)$ from F
4. Let $e = (u, v)$ be a min cost edge, where $u \in V_i$, $v \in V_j$, $j \neq i$
5. $V_i := V_i \cup V_j$; $T_i := T_i \cup T_j \cup \{e\}$
   Remove tree (V_j, T_j) from forest F

Many ways of implementing this.

- Prim's algorithm
- Kruskal's algorithm

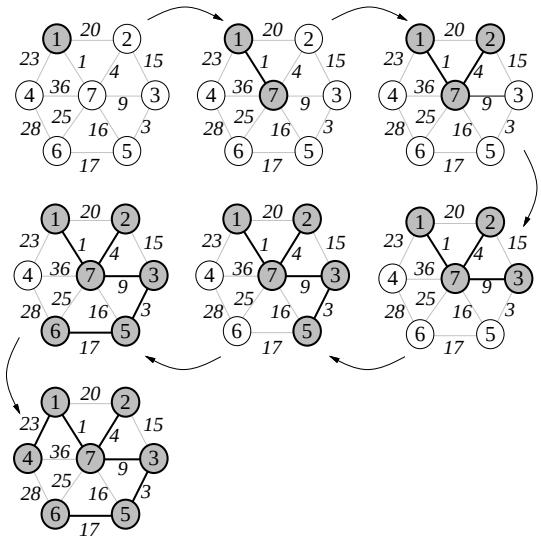
Prim's Algorithm: Overview

Take $i = 1$ in every iteration of the algorithm.

We need to keep track of:

- the set of edges in the spanning forest, T_1
- the set of vertices V_1
- the edges that lead out of the tree (V_1, T_1), in a data structure that enables us to choose the one of smallest cost

Example



Prim's Algorithm

Q is a priority queue of edges, ordered on cost.

1. $T_1 := \emptyset$; $V_1 := \{1\}$; $Q :=$ empty
2. **for** each $w \in V$ such that $(1, w) \in E$ **do**
3. insert$(Q, (1, w))$
4. **while** $|V_1| < n$ **do**
5. $e :=$deletemin(Q)
6. Suppose $e = (u, v)$, where $u \in V_1$
7. **if** $v \notin V_1$ **then**
8. $T_1 := T_1 \cup \{e\}$
9. $V_1 := V_1 \cup \{v\}$
10. **for** each $w \in V$ such that $(v, w) \in E$ **do**
11. insert$(Q, (v, w))$

Data Structures

For

- E: adjacency list
- V_1: linked list
- T_1: linked list
- Q: heap

Analysis

- Line 1: $O(1)$
- Line 3: $O(\log e)$

- Lines 2–3: $O(n \log e)$
- Line 5: $O(\log e)$
- Line 6: $O(1)$
- Line 7: $O(1)$
- Line 8: $O(1)$
- Line 9: $O(1)$
- Lines 4–9: $O(e \log e)$
- Line 11: $O(\log e)$
- Lines 4,10–11: $O(e \log e)$

Total: $O(e \log e) = O(e \log n)$

Kruskal's Algorithm: Overview

At every iteration of the algorithm, take $e = (u, v)$ to be the unused edge of smallest cost, and V_i and V_j to be the vertex sets of the forest such that $u \in V_i$ and $v \in V_j$.

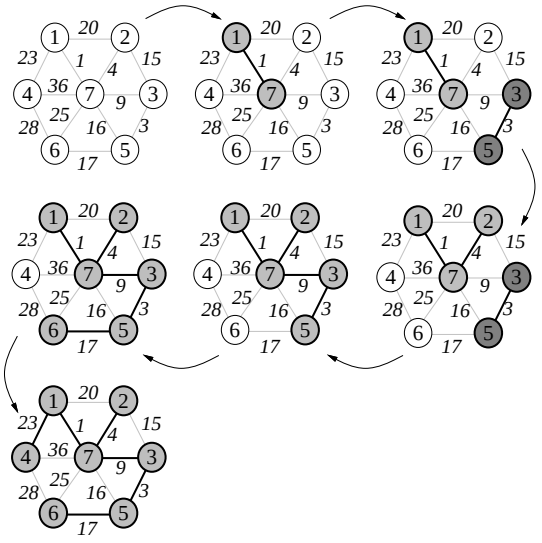
We need to keep track of

- the vertices of spanning forest

$$F = \{V_1, V_2, \ldots, V_k\}$$

  (note: F is a partition of V)
- the set of edges in the spanning forest, T
- the unused edges, in a data structure that enables us to choose the one of smallest cost

Example



Kruskal's Algorithm

T is the set of edges in the forest.

F is the set of vertex-sets in the forest.

1. $T := \emptyset$; $F := \emptyset$
2. **for** each vertex $v \in V$ **do** $F := F \cup \{\{v\}\}$
3. Sort edges in E in ascending order of cost
4. **while** $|F| > 1$ **do**
5. $\quad (u, v) :=$ the next edge in sorted order
6. $\quad$ **if** find$(u) \neq$ find(v) **then**
7. $\quad\quad$ union(u, v)
8. $\quad\quad T := T \cup \{(u, v)\}$

Lines 6 and 7 use union-find on F.

On termination, T contains the edges in the min cost spanning tree for $G = (V, E)$.

Data Structures

For

- E: adjacency list
- F: union-find data structure
- T: linked list

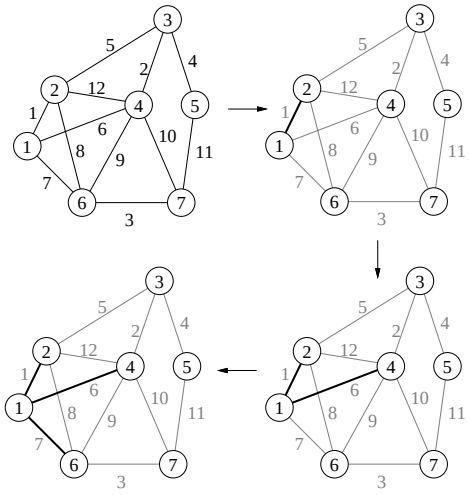
Analysis

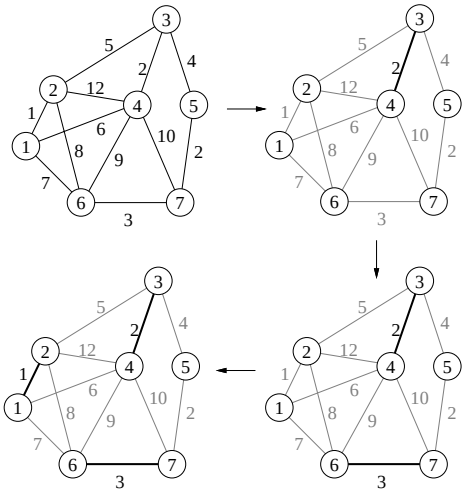
- Line 1: $O(1)$
- Line 2: $O(n)$
- Line 3: $O(e \log e) = O(e \log n)$
- Line 5: $O(1)$
- Line 6: $O(\log n)$
- Line 7: $O(\log n)$
- Line 8: $O(1)$
- Lines 4–8: $O(e \log n)$

Total: $O(e \log n)$

Questions

Would Prim's algorithm do this?

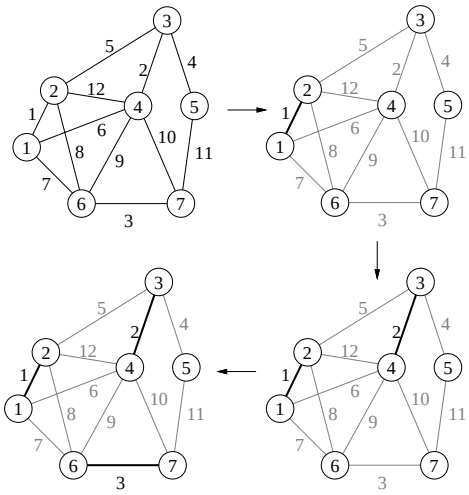




Assigned Reading

CLR Chapter 24.

Would Kruskal's algorithm do this?



Could a min-cost spanning tree algorithm do this?

Algorithms Course Notes
Backtracking 1

Ian Parberry*

Fall 2001

Summary

Backtracking: exhaustive search using divide-and-conquer.

General principles:

- backtracking with binary strings
- backtracking with k-ary strings
- pruning
- how to write a backtracking algorithm

Application to

- the knapsack problem
- the Hamiltonian cycle problem
- the travelling salesperson problem

Exhaustive Search

Sometimes the best algorithm for a problem is to try all possibilities.

This is always slow, but there are standard tools that we can use to help: algorithms for generating basic objects such as

- 2^n binary strings of length n
- k^n k-ary strings of length n
- $n!$ permutations
- $n!/r!(n-r)!$ combinations of n things chosen r at a time

Backtracking speeds the exhaustive search by pruning.

Bit Strings

Problem: Generate all the strings of n bits.

Solution: Use divide-and-conquer. Keep current binary string in an array $A[1..n]$. Aim is to call process(A) once with $A[1..n]$ containing each binary string. Call procedure binary(n):

procedure binary(m)
comment process all binary strings of length m
  if $m = 0$ **then** process(A) **else**
    $A[m] := 0$; binary($m-1$)
    $A[m] := 1$; binary($m-1$)

Correctness Proof

Claim: For all $m \geq 1$, binary(m) calls process(A) once with $A[1..m]$ containing every string of m bits.

Proof: The proof is by induction on m. The claim is certainly true for $m = 0$.

Now suppose that binary($m-1$) calls process(A) once with $A[1..m-1]$ containing every string of $m-1$ bits.

First, binary(m)

- sets $A[m]$ to 0, and
- calls binary($m-1$).

By the induction hypothesis, this calls process(A) once with $A[1..m]$ containing every string of m bits ending in 0.

Then, binary(m)

- sets $A[m]$ to 1, and
- calls binary($m-1$).

By the induction hypothesis, this calls process(A) once with $A[1..m]$ containing every string of m bits ending in 1.

Hence, binary(m) calls process(A) once with $A[1..m]$ containing every string of m bits.

Analysis

Let $T(n)$ be the running time of binary(n). Assume procedure process takes time $O(1)$. Then,

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n-1) + d & \text{otherwise} \end{cases}$$

Therefore, using repeated substitution,

$$T(n) = (c+d)2^{n-1} - d.$$

Hence, $T(n) = O(2^n)$, which means that the algorithm for generating bit-strings is optimal.

k-ary Strings

Problem: Generate all the strings of n numbers drawn from $0..k-1$.

Solution: Keep current k-ary string in an array $A[1..n]$. Aim is to call process(A) once with $A[1..n]$ containing each k-ary string. Call procedure string(n):

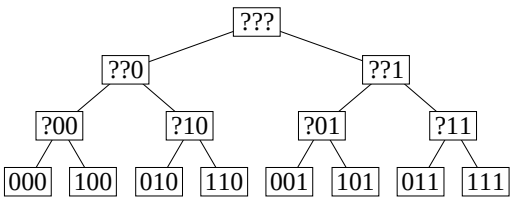
```
procedure string(m)
comment process all k-ary strings of length m
  if m = 0 then process(A) else
    for j := 0 to k − 1 do
      A[m] := j; string(m − 1)
```

Correctness proof: similar to binary case.

Analysis: similar to binary case, algorithm optimal.

Backtracking Principles

Backtracking imposes a tree structure on the solution space. e.g. binary strings of length 3:



Backtracking does a preorder traversal on this tree, processing the leaves.

Save time by pruning: skip over internal nodes that have no useful leaves.

k-ary strings with Pruning

Procedure string fills the array A from right to left, string(m, k) can prune away choices for $A[m]$ that are incompatible with $A[m+1..n]$.

```
procedure string(m)
comment process all k-ary strings of length m
  if m = 0 then process(A) else
    for j := 0 to k − 1 do
      if j is a valid choice for A[m] then
        A[m] := j; string(m − 1)
```

Devising a Backtracking Algorithm

Steps in devising a backtracker:

- Choose basic object, e.g. strings, permutations, combinations. (In this lecture it will always be strings.)
- Start with the divide-and-conquer code for generating the basic objects.
- Place code to test for desired property at the base of recursion.
- Place code for pruning before each recursive call.

General form of generation algorithm:

```
procedure generate(m)
  if m = 0 then process(A) else
    for each of a bunch of numbers j do
      A[m] := j; generate(m − 1)
```

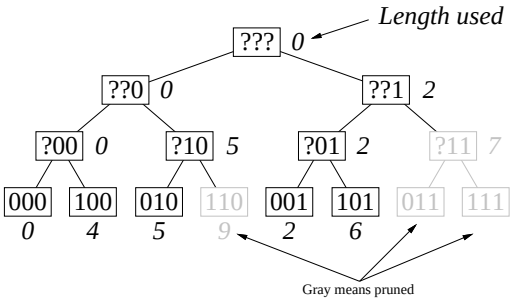
General form of backtracking algorithm:

```
procedure generate(m)
  if m = 0 then process(A) else
    for each of a bunch of numbers j do
      if j consistent with A[m + 1..n]
        then A[m] := j; generate(m − 1)
```

The Knapsack Problem

To solve the knapsack problem: given n rods, use a bit array $A[1..n]$. Set $A[i]$ to 1 if rod i is to be used. Exhaustively search through all binary strings $A[1..n]$ testing for a fit.

$n = 3$, $s_1 = 4$, $s_2 = 5$, $s_3 = 2$, $L = 6$.



The Algorithm

Use the binary string algorithm. Pruning: let ℓ be length remaining,

- $A[m] = 0$ is always legal
- $A[m] = 1$ is illegal if $s_m > \ell$; prune here

To print all solutions to knapsack problem with n rods of length $s_1, \ldots, s_n$, and knapsack of length L, call knapsack(n, L).

```
procedure knapsack(m, ℓ)
comment solve knapsack problem with m
    rods, knapsack size ℓ
 if m = 0 then
    if ℓ = 0 then print(A)
 else
   A[m] := 0; knapsack(m − 1, ℓ)
   if s_m ≤ ℓ then
     A[m] := 1; knapsack(m − 1, ℓ − s_m)
```

Analysis: time $O(2^n)$.

Generalized Strings

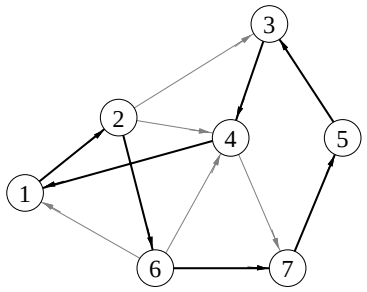
Note that k can be different for each digit of the string, e.g. keep an array $D[1..n]$ with $A[m]$ taking on values $0..D[m] - 1$.

```
procedure string(m)
comment process all strings of length m
 if m = 0 then process(A) else
   for j := 0 to D[m] − 1 do
     A[m] := j; string(m − 1)
```

Application: in a graph with n nodes, $D[m]$ could be the degree of node m.

Hamiltonian Cycles

A *Hamiltonian cycle* in a directed graph is a cycle that passes through every vertex exactly once.



Problem: Given a directed graph $G = (V, E)$, find a Hamiltonian cycle.

Store the graph as an adjacency list (for each vertex $v \in \{1, 2, \ldots, n\}$, store a list of the vertices w such that $(v, w) \in E$).

Store a Hamiltonian cycle as $A[1..n]$, where the cycle is

$$A[n] \to A[n-1] \to \cdots \to A[2] \to A[1] \to A[n]$$

Adjacency list:

N	1	2	3
1	2		
2	3	4	6
3	4		
4	1	7	
5	3		
6	1	4	7
7	5		

D	
1	1
2	3
3	1
4	2
5	1
6	3
7	1

Hamiltonian cycle:

	1	2	3	4	5	6	7
A	6	2	1	4	3	5	7

The Algorithm

Use the generalized string algorithm. Pruning: Keep a Boolean array $U[1..n]$, where $U[m]$ is `true` iff vertex m is unused. On entering m,

- $U[m] =$ `true` is legal
- $U[m] =$ `false` is illegal; prune here

To solve the Hamiltonian cycle problem, set up arrays N (neighbours) and D (degree) for the input graph, and execute the following. Assume that $n > 1$.

```
 1.  for i := 1 to n − 1 do U[i] := true
 2.  U[n] := false; A[n] := n
 3.  hamilton(n − 1)

     procedure hamilton(m)
     comment complete Hamiltonian cycle
         from A[m + 1] with m unused nodes
 4.    if m = 0 then process(A) else
 5.      for j := 1 to D[A[m + 1]] do
 6.        w := N[A[m + 1], j]
 7.        if U[w] then
 8.          U[w]:=false
 9.          A[m] := w; hamilton(m − 1)
10.          U[w]:=true

     procedure process(A)
     comment check that the cycle is closed
11.    ok:=false
12.    for j := 1 to D[A[1]] do
13.      if N[A[1], j] = A[n] then ok:=true
14.    if ok then print(A)
```

Notes on the algorithm:

- From line 2, $A[n] = n$.
- Procedure process(A) closes the cycle. At the point where process(A) is called, $A[1..n]$ contains a Hamiltonian path from vertex $A[n]$ to vertex $A[1]$. It remains to verify the edge from vertex $A[1]$ to vertex $A[n]$, and print A if successful.
- Line 7 does the pruning, based on whether the new node has been used (marked in line 8).
- Line 10 marks a node as unused when we return from visiting it.

Analysis

How long does it take procedure hamilton to find <u>one</u> Hamiltonian cycle? Assume that there are none.

Let $T(n)$ be the running time of hamilton(v, n). Suppose the graph has maximum out-degree d. Then, for some $b, c \in \mathbb{N}$,

$$T(n) \leq \begin{cases} bd & \text{if } n = 0 \\ dT(n-1) + c & \text{otherwise} \end{cases}$$

Hence (by repeated substitution), $T(n) = O(d^n)$.

Therefore, the total running time on an n-vertex graph is $O(d^n)$ if there are no Hamiltonian cycles. The running time will be $O(d^n + d) = O(d^n)$ if one is found.

Travelling Salesperson

Optimization problem (TSP): Given a directed labelled graph $G = (V, E)$, find a Hamiltonian cycle of minimum cost (cost is sum of labels on edges).

Bounded optimization problem (BTSP): Given a directed labelled graph $G = (V, E)$ and $B \in \mathbb{N}$, find a Hamiltonian cycle of cost $\leq B$.

It is enough to solve the bounded problem. Then the full problem can be solved using binary search.

Suppose we have a procedure BTSP(x) that returns a Hamiltonian cycle of length at most x if one exists. To solve the optimization problem, call TSP$(1, B)$, where B is the sum of all the edge costs (the cheapest Hamiltonian cycle costs less that this, if one exists).

```
function TSP(ℓ, r)
  comment return min cost Ham. cycle
    of cost ≤ r and ≥ ℓ
  if ℓ = r then return (BTSP(ℓ)) else
    m := ⌊(ℓ + r)/2⌋
    if BTSP(m) succeeds
      then return(TSP(ℓ, m))
      else return(TSP(m + 1, r))
```

(NB. should first call BTSP(B) to make sure a Hamiltonian cycle exists.)

Procedure TSP is called $O(\log B)$ times (analysis similar to that of binary search). Suppose

- the graph has n edges
- the graph has labels at most b (so $B \leq bn$)
- procedure BTSP runs in time $O(T(n))$.

Then, procedure TSP runs in time

$$(\log n + \log b)T(n).$$

Backtracking for BTSP

Let $C[v, w]$ be cost of edge $(v, w) \in E$, $C[v, w] = \infty$ if $(v, w) \notin E$.

```
1.  for i := 1 to n − 1 do U[i] := true
2.  U[n] := false; A[n] := n
3.  hamilton(n − 1, B)

    procedure hamilton(m, b)
    comment complete Hamiltonian cycle from
        A[m + 1] with m unused nodes, cost ≤ b
4.    if m = 0 then process(A, b) else
5.      for j := 1 to D[A[m + 1]] do
6.        w := N[A[m + 1], j]
7.        if U[w] and C[A[m + 1], w] ≤ b then
8.          U[w]:=false
9.          A[m] := w
            hamilton(m − 1, b − C[v, w])
10.         U[w]:=true

    procedure process(A, b)
    comment check that the cycle is closed
11.   if C[A[1], n] ≤ b then print(A)
```

Notes on the algorithm:

- Extra pruning of expensive tours is done in line 7.
- Procedure process (line 11) is made easier using adjacency matrix.

Analysis: time $O(d^n)$, as for Hamiltonian cycles.

Algorithms Course Notes
Backtracking 2

Ian Parberry*

Fall 2001

Summary

Backtracking with permutations.

Application to

- the Hamiltonian cycle problem
- the peaceful queens problem

Permutations

Problem: Generate all permutations of n distinct numbers.

Solution:

- Backtrack through all n-ary strings of length n, pruning out non-permutations.
- Keep a Boolean array $U[1..n]$, where $U[m]$ is `true` iff m is unused.
- Keep current permutation in an array $A[1..n]$.
- Aim is to call process(A) once with $A[1..n]$ containing each permutation.

Call procedure permute(n):

```
    procedure permute(m)
    comment process all perms of length m
1.    if m = 0 then process(A) else
2.      for j := 1 to n do
3.        if U[j] then
4.          U[j]:=false
5.          A[m] := j; permute(m − 1)
6.          U[j]:=true
```

Note: this is what we did in the Hamiltonian cycle algorithm. A Hamiltonian cycle is a permutation of the nodes of the graph.

Analysis

Clearly, the algorithm runs in time $O(n^n)$. But this analysis is not tight: it does not take into account pruning.

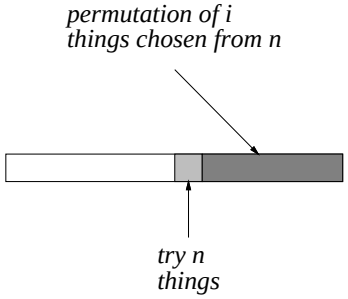
How many times is line 3 executed? Running time will be big-O of this.

What is the situation when line 3 is executed? The algorithm has, for some $0 \leq i < n$,

- filled in i places at the top of A with part of a permutation, and
- tries n candidates for the next place in line 2.

What do the top i places in a permutation look like?

- i things chosen from n without duplicates
- permuted in some way



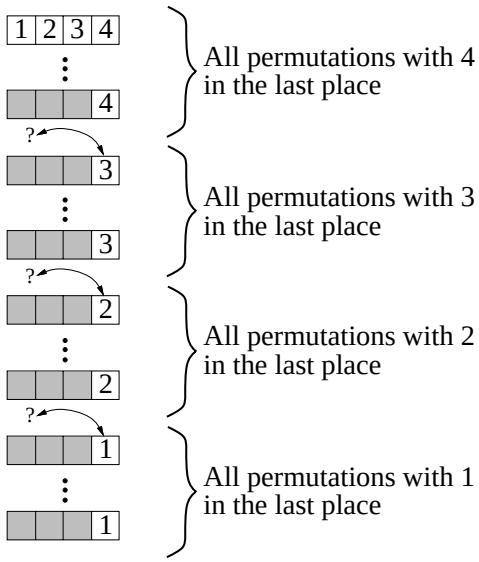
How many ways are there of filling in part of a permutation?

$$\binom{n}{i} i!$$

Therefore, number of times line 3 is executed is

$$n \sum_{i=0}^{n-1} \binom{n}{i} i!$$

$$
\begin{aligned}
&= n \sum_{i=0}^{n-1} \frac{n!}{(n-i)!} \\
&= n \cdot n! \sum_{i=1}^{n} \frac{1}{i!} \\
&\leq (n+1)!(e-1) \\
&\leq 1.718(n+1)!
\end{aligned}
$$



Analysis of Permutation Algorithm

Therefore, procedure permute runs in time $O((n+1)!)$.

This is an improvement over $O(n^n)$. Recall that

$$n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n.$$

```
procedure permute(m)
comment process all perms in A[1..m]
1.  if m = 1 then process(A) else
2.    permute(m − 1)
3.    for i := 1 to m − 1 do
4.      swap A[m] with A[j] for some 1 ≤ j < m
5.      permute(m − 1)
```

Faster Permutations

The permutation generation algorithm is not optimal: off by a factor of n

- generates $n!$ permutations
- takes time $O((n+1)!)$.

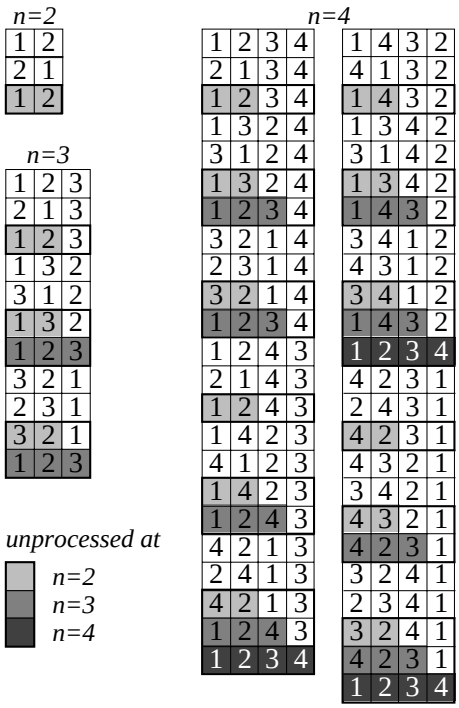
Problem: How to make sure that the values swapped into $A[m]$ in line 4 are distinct?

Solution: Make sure that A is reset after every recursive call, and swap $A[m]$ with $A[i]$.

A better algorithm: use divide and conquer to devise a faster backtracking algorithm that generates only permutations.

- Initially set $A[i] = i$ for all $1 \leq i \leq n$.
- Instead of setting $A[n]$ to $1..n$ in turn, swap values from $A[1..n-1]$ into it.

```
procedure permute(m)
1.  if m = 1 then process else
2.    permute(m − 1)
3.    for i := m − 1 downto 1 do
4.      swap A[m] with A[i]
5.      permute(m − 1)
6.      swap A[m] with A[i]
```



Analysis

Let $T(n)$ be the number of swaps made by permute(n). Then $T(1) = 0$, and for $n > 2$, $T(n) = nT(n-1) + 2(n-1)$.

Claim: For all $n \geq 1$, $T(n) \leq 2n! - 2$.

Proof: Proof by induction on n. The claim is certainly true for $n = 1$. Now suppose the hypothesis is true for n. Then,

$$\begin{aligned} T(n+1) &= (n+1)T(n) + 2n \\ &\leq (n+1)(2n! - 2) + 2n \\ &= 2(n+1)! - 2. \end{aligned}$$

Hence, procedure permute runs in time $O(n!)$, which is optimal.

Hamiltonian Cycles on Dense Graphs

Use adjacency matrix: $M[i,j]$ `true` iff $(i,j) \in E$.

```
1.  for i := 1 to n do
2.    A[i] := i
3.  hamilton(n − 1)

    procedure hamilton(m)
    comment find Hamiltonian cycle from A[m]
       with m unused nodes
4.   if m = 0 then process(A) else
5.     for j := m downto 1 do
6.       if M[A[m + 1], A[j]] then
7.         swap A[m] with A[j]
8.         hamilton(m − 1)
9.         swap A[m] with A[j]

    procedure process(A)
    comment check that the cycle is closed
10.   if M[A[1], n] then print(A)
```

Analysis

Therefore, the Hamiltonian circuit algorithms run in time

- $O(d^n)$ (using generalized strings)
- $O((n-1)!)$ (using permutations).

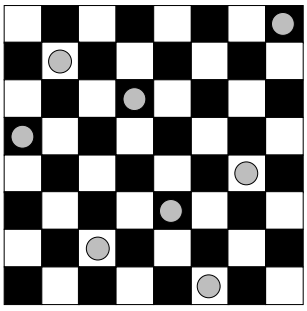
Which is the smaller of the two bounds? It depends on d. The string algorithm is better if $d \leq n/e$ ($e = 2.7183\ldots$).

Notes on algorithm:

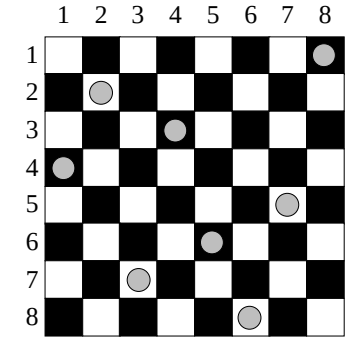
- Note upper limit in for-loop on line 5 is m, not $m - 1$: why?
- Can also be applied to travelling salesperson.

The Peaceful Queens Problem

How many ways are there to place n queens on an $n \times n$ chessboard so that no queen threatens any other.



Number the rows and columns $1..n$. Use an array $A[1..n]$, with $A[i]$ containing the row number of the queen in column i.



A	4	2	7	3	6	8	5	1

This is a permutation!

An Algorithm

procedure queen(m)
1. **if** $m = 0$ **then** process **else**
2. **for** $i := m$ **downto** 1 **do**
3. **if** OK to put queen in $(A[i], m)$ **then**
4. swap $A[m]$ with $A[i]$
5. queen($m - 1$)
6. swap $A[m]$ with $A[i]$

How do we determine if it is OK to put a queen in position (i, j)? It is OK if there is no queen in the same diagonal or backdiagonal.

Consider the following array: entry (i, j) contains $i + j$.

	1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8	9
2	3	4	5	6	7	8	9	10
3	4	5	6	7	8	9	10	11
4	5	6	7	8	9	10	11	12
5	6	7	8	9	10	11	12	13
6	7	8	9	10	11	12	13	14
7	8	9	10	11	12	13	14	15
8	9	10	11	12	13	14	15	16

Note that:

- Entries in the same back-diagonal are identical.
- Entries are in the range $2..2n$.

Consider the following array: entry (i, j) contains $i - j$.

	1	2	3	4	5	6	7	8
1	0	-1	-2	-3	-4	-5	-6	-7
2	1	0	-1	-2	-3	-4	-5	-6
3	2	1	0	-1	-2	-3	-4	-5
4	3	2	1	0	-1	-2	-3	-4
5	4	3	2	1	0	-1	-2	-3
6	5	4	3	2	1	0	-1	-2
7	6	5	4	3	2	1	0	-1
8	7	6	5	4	3	2	1	0

Note that:

- Entries in the same diagonal are identical.
- Entries are in the range $-n + 1..n - 1$.

Keep arrays $b[2..2n]$ and $d[-n+1..n-1]$ (initialized to `true`) such that:

- $b[i]$ is `false` if back-diagonal i is occupied by a queen, $2 \leq i \leq 2n$.
- $d[i]$ is `false` if diagonal i is occupied by a queen, $-n + 1 \leq i \leq n - 1$.

procedure queen(m)
1. **if** $m = 0$ **then** process **else**
2. **for** $i := m$ **downto** 1 **do**
3. **if** $b[A[i] + m]$ and $d[A[i] - m]$ **then**
4. swap $A[m]$ with $A[i]$
5. $b[A[m] + m] :=$ `false`
6. $d[A[m] - m] :=$ `false`
7. queen($m - 1$)
8. $b[A[m] + m] :=$ `true`
9. $d[A[m] - m] :=$ `true`
10. swap $A[m]$ with $A[i]$

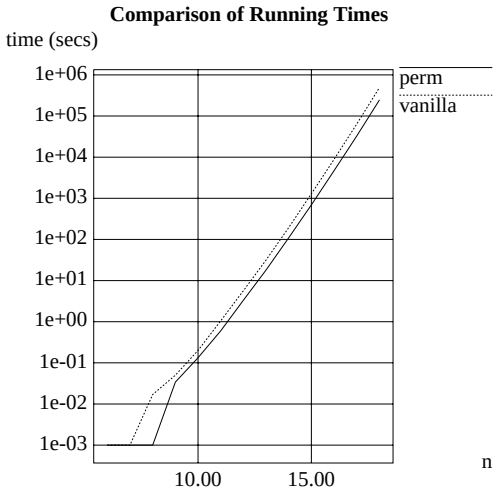
Experimental Results

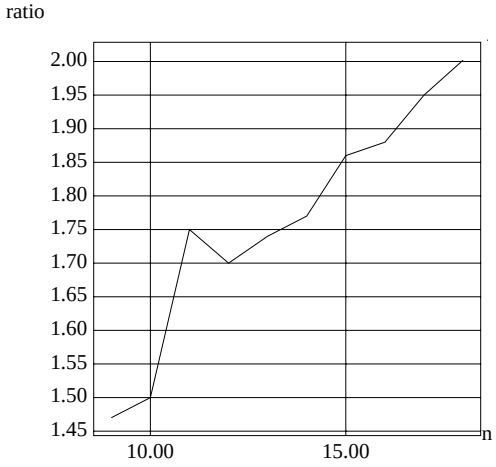
How well does the algorithm work in practice?

- theoretical running time: $O(n!)$
- implemented in Berkeley Unix Pascal on Sun Sparc 2
- practical for $n \leq 18$
- possible to reduce run-time by factor of 2 (not implemented)

- improvement over naive backtracking (theoretically $O(n \cdot n!)$) observed to be more than a constant multiple

n	**count**	**time**
6	4	< 0.001 seconds
7	40	< 0.001 seconds
8	92	< 0.001 seconds
9	352	0.034 seconds
10	724	0.133 seconds
11	2,680	0.6 seconds
12	14,200	3.3 seconds
13	73,712	18.0 seconds
14	365,596	1.8 minutes
15	2,279,184	11.6 minutes
16	14,772,512	1.3 hours
17	95,815,104	9.1 hours
18	666,090,624	2.8 days





Algorithms Course Notes
Backtracking 3

Ian Parberry*

Fall 2001

Summary

Backtracking with combinations.

- principles
- analysis

Combinations

Problem: Generate all the combinations of n things chosen r at a time.

Solution: Keep the current combination in an array $A[1..r]$. Call procedure choose(n, r).

```
procedure choose(m, q)
comment choose q elements out of 1..m
  if q = 0 then process(A)
    else for i := q to m do
      A[q] := i
      choose(i − 1, q − 1)
```

Correctness

Proved in three parts:

1. Only combinations are processed.
2. The right number of combinations are processed.
3. No combination is processed twice.

Claim 1. If $0 \leq r \leq n$, then in combinations processed by choose(n, r), $A[1..r]$ contains a combination of r things chosen from $1..n$.

Proof: Proof by induction on r. The hypothesis is vacuously true for $r = 0$.

Now suppose that $r > 0$ and for all $i \geq r - 1$, in combinations processed by choose$(i, r - 1)$, $A[1..r - 1]$ contains a combination of $r-1$ things chosen from $1..i$.

Now, choose(n, r) calls choose$(i - 1, r - 1)$ with i running from r to n. Therefore, by the induction hypothesis and since $A[r]$ is set to i, in combinations processed by choose(n, r), $A[1..r]$ contains a combination of $r - 1$ things chosen from $1..i - 1$, followed by the value i.

That is, $A[1..r]$ contains a combination of r things chosen from $1..n$.

Claim 2. A call to choose(n, r) processes exactly

$$\left(\begin{array}{c} n \\ r \end{array} \right)$$

combinations.

Proof: Proof by induction on r. The hypothesis is certainly true for $r = 0$.

Now suppose that $r > 0$ and a call to choose$(i,r-1)$ generates exactly

$$\left(\begin{array}{c} i \\ r-1 \end{array} \right)$$

combinations, for all $r - 1 \leq i \leq n$.

Now, choose(n, r) calls choose$(i - 1, r - 1)$ with i running from r to n. Therefore, by the induction hypothesis, the number of combinations generated is

$$\begin{aligned} \sum_{i=r}^{n} \left(\begin{array}{c} i-1 \\ r-1 \end{array} \right) &= \sum_{i=r-1}^{n-1} \left(\begin{array}{c} i \\ r-1 \end{array} \right) \\ &= \left(\begin{array}{c} n \\ r \end{array} \right). \end{aligned}$$

The first step follows by re-indexing, and the second step follows by Identity 2:

Identity 1

Claim:

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

Proof:

$$\begin{aligned}
& \binom{n}{r} + \binom{n}{r-1} \\
= \; & \frac{n!}{r!(n-r)!} + \frac{n!}{(r-1)!(n-r+1)!} \\
= \; & \frac{(n+1-r)n! + rn!}{r!(n+1-r)!} \\
= \; & \frac{(n+1)n!}{r!(n+1-r)!} \\
= \; & \binom{n+1}{r}.
\end{aligned}$$

Identity 2

Claim: For all $n \geq 0$ and $1 \leq r \leq n$,

$$\sum_{i=r}^{n} \binom{i}{r} = \binom{n+1}{r+1}.$$

Proof: Proof by induction on n. Suppose $r \geq 1$. The hypothesis is certainly true for $n = r$, in which case both sides of the equation are equal to 1.

Now suppose $n > r$ and that

$$\sum_{i=r}^{n} \binom{i}{r} = \binom{n+1}{r+1}.$$

We are required to prove that

$$\sum_{i=r}^{n+1} \binom{i}{r} = \binom{n+2}{r+1}.$$

Now, by the induction hypothesis,

$$\begin{aligned}
& \sum_{i=r}^{n+1} \binom{i}{r} \\
= \; & \sum_{i=r}^{n} \binom{i}{r} + \binom{n+1}{r} \\
= \; & \binom{n+1}{r+1} + \binom{n+1}{r} \\
= \; & \binom{n+2}{r+1} \quad \text{(By Identity 1).}
\end{aligned}$$

Back to the Correctness Proof

Claim 3. A call to choose(n, r) processes combinations of r things chosen from $1..n$ without repeating a combination.

Proof: The proof is by induction on r, and is similar to the above.

Now we can complete the correctness proof: By Claim 1, a call to procedure choose(n, r) generates combinations of r things from $1..n$ (since r things at a time are processed from the array A. By Claim 3, it never duplicates a combination. By Claim 2, it generates exactly the right number of combinations. Therefore, it generates exactly the combinations of r things chosen from $1..n$.

Analysis

Let $T(n, r)$ be the number of assignments to A made in choose(n, r).

Claim: For all $n \geq 1$ and $0 \leq r \leq n$,

$$T(n, r) \leq r \binom{n}{r}.$$

Proof: A call to choose(n, r) makes the following recursive calls:

```
for i := r to n do
  A[r] := i
  choose(i − 1, r − 1)
```

The costs are the following:

Call	*Cost*
choose($r-1$,$r-1$)	$T(r-1,r-1)+1$
choose(r,$r-1$)	$T(r,r-1)+1$
$\vdots$	
choose($n-1$,$r-1$)	$T(n-1,r-1)+1$

Therefore, summing these costs:

$$T(n,r) = \sum_{i=r-1}^{n-1} T(i,r-1) + (n-r+1)$$

We will prove by induction on r that

$$T(n,r) \leq r \left(\begin{array}{c} n \\ r \end{array} \right).$$

The claim is certainly true for $r = 0$. Now suppose that $r > 0$ and for all $i < n$,

$$T(i,r-1) \leq (r-1) \left(\begin{array}{c} i \\ r-1 \end{array} \right).$$

Then by the induction hypothesis and Identity 2,

$$\begin{aligned}
& T(n,r) \\
= & \sum_{i=r-1}^{n-1} T(i,r-1) + (n-r+1) \\
\leq & \sum_{i=r-1}^{n-1} ((r-1) \left(\begin{array}{c} i \\ r-1 \end{array} \right)) + (n-r+1) \\
= & (r-1) \sum_{i=r-1}^{n-1} \left(\begin{array}{c} i \\ r-1 \end{array} \right) + (n-r+1) \\
= & (r-1) \left(\begin{array}{c} n \\ r \end{array} \right) + (n-r+1) \\
\leq & r \left(\begin{array}{c} n \\ r \end{array} \right)
\end{aligned}$$

This last step follows since, for $1 \leq r \leq n$,

$$\left(\begin{array}{c} n \\ r \end{array} \right) \geq n-r+1$$

This can be proved by induction on r. It is certainly true for $r = 1$, since both sides of the inequality are equal to n in this case.

Now suppose that $r > 1$.

$$\begin{aligned}
\left(\begin{array}{c} n \\ r \end{array} \right) &= \frac{n!}{r!(n-r)!} \\
&= \frac{n(n-1)!}{r(r-1)!((n-1)-(r-1))!} \\
&= \frac{n}{r} \frac{(n-1)!}{(r-1)!((n-1)-(r-1))!} \\
&= \frac{n}{r} \left(\begin{array}{c} n-1 \\ r-1 \end{array} \right) \\
&\geq \frac{n}{r} ((n-1)-(r-1)+1) \quad \text{(by hyp.)} \\
&\geq n-r+1 \quad \text{(since } r \leq n\text{)}
\end{aligned}$$

Optimality

Therefore, the combination generation algorithm runs in time $O(r \left(\begin{array}{c} n \\ r \end{array} \right))$ (since the amount of extra computation done for each assignment is a constant).

Therefore, the combination generation algorithm seems to have running time that is <u>not</u> optimal to within a constant multiple (since there are $\left(\begin{array}{c} n \\ r \end{array} \right)$ combinations).

Unfortunately, it often seems to require this much time. In the following table, A denotes the number of assignments to array A, C denotes the number of combinations, and *Ratio* denotes A/C.

Experiments

n	r	A	C	*Ratio*
20	1	20	20	1.00
20	2	209	190	1.10
20	3	1329	1140	1.17
20	4	5984	4845	1.24
20	5	20348	15504	1.31
20	6	54263	38760	1.40
20	7	116279	77520	1.50
20	8	203489	125970	1.62
20	9	293929	167960	1.75
20	10	352715	184756	1.91
20	11	352715	167960	2.10
20	12	293929	125970	2.33
20	13	203489	77520	2.62
20	14	116279	38760	3.00
20	15	54263	15504	3.50
20	16	20348	4845	4.20
20	17	5984	1140	5.25
20	18	1329	190	6.99
20	19	209	20	10.45
20	20	20	1	20.00

Optimality for $r \leq n/2$

It looks like $T(n) < 2\binom{n}{r}$ provided $r \leq n/2$. But is this true, or is it just something that happens with small numbers?

Claim: If $1 \leq r \leq n/2$, then

$$T(n,r) \leq 2\binom{n}{r} - r.$$

Observation: With induction, you often have to prove something stronger than you want.

Proof: First, we will prove the claim for $r = 1, 2$. Then we will prove it for $3 \leq r \leq n/2$ by induction on n.

The Case $r = 1$

$T(n,1) = n$ (by observation), and

$$2\binom{n}{1} - 1 = 2n - 1 \geq n.$$

Hence,

$$T(n,1) \leq 2\binom{n}{1} - 1,$$

as required.

The Case $r = 2$

Since

$$\begin{aligned} T(n,2) &= \sum_{i=1}^{n-1} T(i,1) + (n-1) \\ &= \sum_{i=1}^{n-1} i + (n-1) \\ &= n(n-1)/2 + (n-1) \\ &= (n+2)(n-1)/2, \end{aligned}$$

and

$$2\binom{n}{2} - 2 = n(n-1) - 2,$$

we require that

$$\begin{aligned} & (n+2)(n-1)/2 \leq n(n-1) - 2 \\ \Leftrightarrow\ & 2n(n-1) - (n+2)(n-1) \geq 4 \\ \Leftrightarrow\ & (n-2)(n-1) \geq 4 \\ \Leftrightarrow\ & n^2 - 3n - 2 \geq 0. \end{aligned}$$

This is true since $n \geq 4$ (recall, $r \leq n/2$), since the largest root of $n^2 - 3n - 2 \geq 0$ is $(3+\sqrt{17})/2 < 3.562$.

The Case $3 \leq r \leq n/2$

Claim: If $3 \leq r \leq n/2$, then

$$T(n,r) \leq 2\binom{n}{r} - r.$$

Proof by induction on n. The base case is $n = 6$. The only choice for r is 3. Experiments show that the algorithm uses 34 assignments to produce 20 combinations. Since $2 \cdot 20 - 2 = 38 > 34$, the hypothesis holds for the base case.

Now suppose $n \geq 6$, which implies that $r \geq 3$. By the induction hypothesis and the case $r = 1, 2$,

$$\begin{aligned} & T(n,r) \\ = & \sum_{i=r-1}^{n-1} T(i, r-1) + (n-r+1) \\ \leq & \sum_{i=r-1}^{n-1} \left(2\binom{i}{r-1} - (r-1)\right) + (n-r+1) \end{aligned}$$

$$
\begin{aligned}
&= 2\sum_{i=r-1}^{n-1}\binom{i}{r-1} - (n-r+1)(r-2) \\
&= 2\binom{n}{r} - (n-r+1)(r-2) \\
&\leq 2\binom{n}{r} - (n-\frac{n}{2}+1)(3-2) \\
&< 2\binom{n}{r} - r.
\end{aligned}
$$

Optimality Revisited

Hence, our algorithm is optimal for $r \leq n/2$.

Sometimes this is just as useful: if $r > n/2$, generate the items not chosen, instead of the items chosen.

Algorithms Course Notes
Backtracking 4

Ian Parberry*

Fall 2001

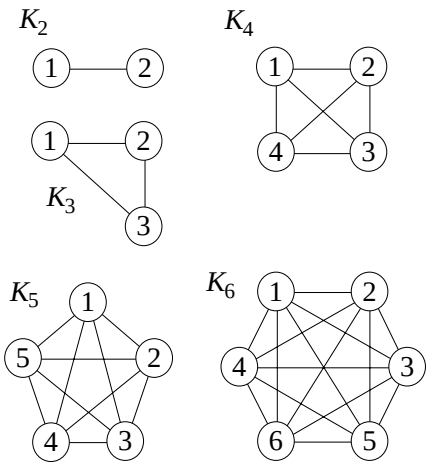
Summary

More on backtracking with combinations. Application to:

- the clique problem
- the independent set problem
- Ramsey numbers

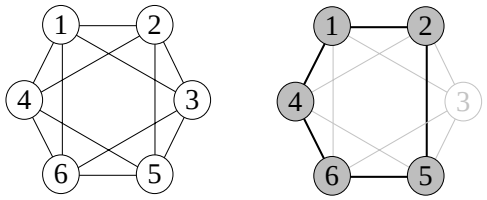
Complete Graphs

The complete graph on n vertices is $K_n = (V, E)$ where $V = \{1, 2, \ldots, n\}$, $E = V \times V$.



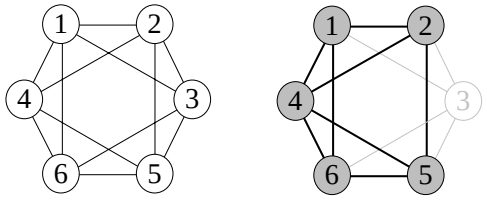
Subgraphs

A subgraph of a graph $G = (V, E)$ is a graph $B = (U, F)$ such that $U \subseteq V$ and $F \subseteq E$.



Induced Subgraphs

An induced subgraph of a graph $G = (V, E)$ is a graph $B = (U, F)$ such that $U \subseteq V$ and $F = (U \times U) \cap E$.



The Clique Problem

A clique is an induced subgraph that is complete. The size of a clique is the number of vertices.

The clique problem:

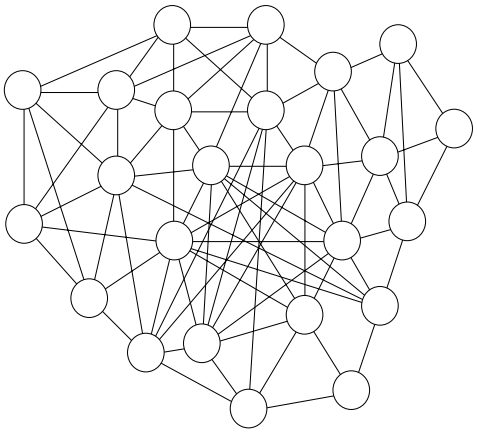
Input: A graph $G = (V, E)$, and an integer r.

Output: A clique of size r in G, if it exists.

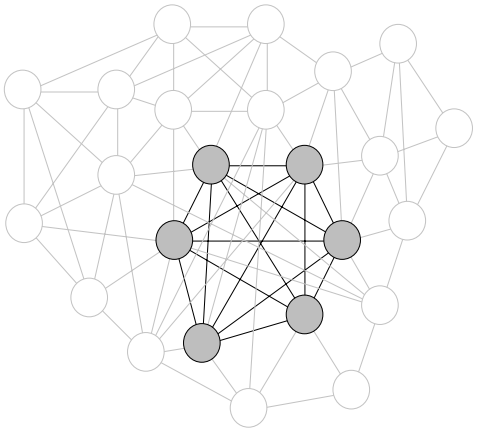
Assumption: given $u, v \in V$, we can check whether $(u, v) \in E$ in $O(1)$ time (use adjacency matrix).

Example

Does this graph have a clique of size 6?



Yes!



A Backtracking Algorithm

Use backtracking on a combination A of r vertices chosen from V. Assume a procedure process(A) that checks to see whether the vertices listed in A form a clique. A represents a set of vertices in a potential clique. Call clique(n, r).

Backtracking without pruning:

```
procedure clique(m, q)
  if q = 0 then process(A) else
    for i := q to m do
      A[q] := i
      clique(i − 1, q − 1)
```

Backtracking with pruning (line 3):

```
     procedure clique(m, q)
1.     if q = 0 then print(A) else
2.       for i := q to m do
3.         if (A[q], A[j]) ∈ E for all q < j ≤ r
             then
4.           A[q] := i
5.           clique(i − 1, q − 1)
```

Analysis

Line 3 takes time $O(r)$. Therefore, the algorithm takes time

- $O(r \binom{n}{r})$ if $r \leq n/2$, and
- $O(r^2 \binom{n}{r})$ otherwise.

The Independent Set Problem

An independent set is an induced subgraph that has no edges. The size of an independent set is the number of vertices.

The independent set problem:

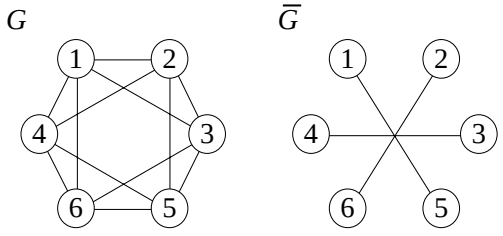
Input: A graph $G = (V, E)$, and an integer r.

Output: Does G have an independent set of size r?

Assumption: given $u, v \in V$, we can check whether $(u, v) \in E$ in $O(1)$ time (use adjacency matrix).

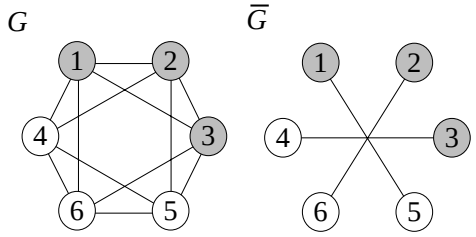
Complement Graphs

The complement of a graph $G = (V, E)$ is a graph $\overline{G} = (V, \overline{E})$, where $\overline{E} = (V \times V) - E$.



Cliques and Independent Sets

A clique in G is an independent set in $\overline{G}$.



Therefore, the independent set problem can be solved with the clique algorithm, in the same running time: Change the **if** statement in line 3 from "**if** $(A[q], A[j]) \in E$" to "**if** $(A[i], A[j]) \notin E$".

Ramsey Numbers

Ramsey's Theorem: For all $i, j \in \mathbb{N}$, there exists a value $R(i, j) \in \mathbb{N}$ such that every graph G with $R(i, j)$ vertices either has a clique of size i or an independent set of size j.

So, $R(i, j)$ is the smallest number n such that every graph on n vertices has either a clique of size i or an independent set of size j.

i	j	$R(i,j)$
3	3	6
3	4	9
3	5	14
3	6	18
3	7	23
3	8	28?
3	9	36
4	4	18

$R(3, 8)$ is either 28 or 29, rumored to be 28.

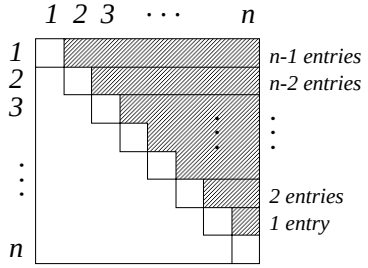
Finding Ramsey Numbers

If the following prints anything, then $R(i, j) > n$. Run it for $n = 1, 2, 3, \ldots$ until the first time it doesn't print anything. That value of n will be $R(i, j)$.

for each graph G with n vertices **do**
  if (G doesn't have a clique of size i) and
    (G doesn't have an indept. set of size j)
  then print(G)

How do we implement the for-loop?

Backtrack through all binary strings A representing the upper triangle of the incidence matrix of G.

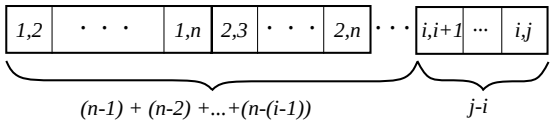


How many entries does A have?

$$\sum_{i=1}^{n-1} i = n(n-1)/2$$

To test if $(i, j) \in E$, where $i < j$, we need to consult the (i, j) entry of the incidence matrix. This is stored in

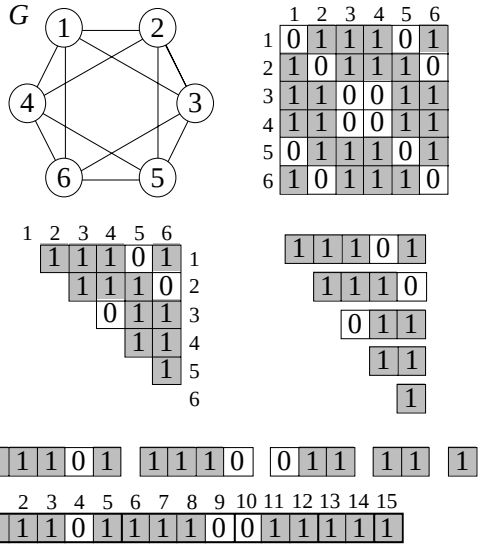
$$A[n(i-1) - i(i+1)/2 + j].$$



Address is:

$$\begin{aligned} &= \sum_{k=1}^{i-1} (n-k) + (j-i) \\ &= n(i-1) - i(i-1)/2 + j - i \\ &= n(i-1) - i(i+1)/2 + j. \end{aligned}$$

Example



Further Reading

R. L. Graham and J. H. Spencer, "Ramsey Theory", *Scientific American*, Vol. 263, No. 1, July 1990.

R. L. Graham, B. L. Rothschild, and J. H. Spencer, *Ramsey Theory*, John Wiley & Sons, 1990.

Algorithms Course Notes
NP Completeness 1

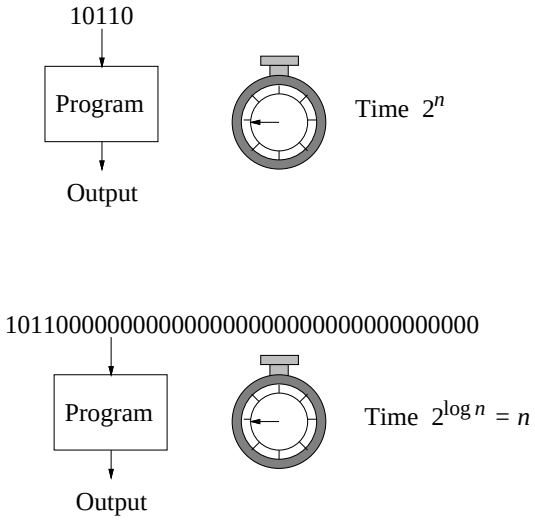
Ian Parberry*

Fall 2001

Summary

An introduction to the theory of *NP* completeness:

- Polynomial time computation
- Standard encoding schemes
- The classes P and NP.
- Polynomial time reductions
- NP completeness
- The satisfiability problem
- Proving new NP completeness results



Polynomial Time

Encode all inputs in binary. Measure running time as a function of n, the number of bits in the input.

Assume

- Each instruction takes $O(1)$ time
- The word size is fixed
- There is as much memory as we need

A program runs in polynomial time if there are constants $c, d \in \mathbb{N}$ such that on every input of size n, the program halts within time $d \cdot n^c$.

Polynomial time is not well-defined. Padding the input to 2^n bits (with extra zeros) makes an exponential time algorithm run in linear time.

But this doesn't tell us how to compute faster.

Standard Encoding Scheme

We must insist that inputs are encoded in binary as tersely as possible.

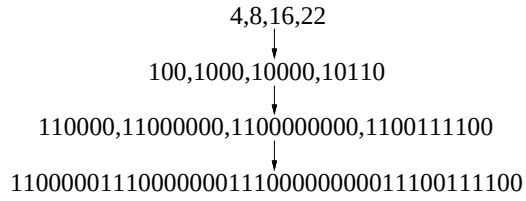
Padding by a polynomial amount is acceptable (since a polynomial of a polynomial is a polynomial), but padding by an exponential amount is not acceptable.

Insist that encoding be no worse than a polynomial amount larger than the *standard encoding scheme*. The standard encoding scheme contains a list each mathematical object and a terse way of encoding it:

- Integers: Store in binary using two's complement.

  154 → 010011010
  -97 → 10011111

- Lists: Duplicate each bit of each list element, and separate them using 01.



- Sets: Store as a list of set elements.
- Graphs: Number the vertices consecutively, and store as an adjacency matrix.

Standard Measures of Input Size

There are some shorthand sizes that we can easily remember. These are no worse than a polynomial of the size of the standard encoding.

- Integers: $\log_2$ of the absolute value of the integer.
- Lists: number of items in the list times size of each item. If it is a list of n integers, and each integer has less than n bits, then n will suffice.
- Sets: size of the list of elements.
- Graphs: Number of vertices or number of edges.

Recognizing Valid Encodings

Not all bit strings of size n encode an object. Some are simply nonsense. For example, all lists use an even number of bits. An algorithm for a problem whose input is a list must deal with the fact that some of the bit strings that it gets as inputs do not encode lists.

For an encoding scheme to be valid, there must be a polynomial time algorithm that can decide whether a given inputs string is a valid encoding of the type of object we are interested in.

Examples

Polynomial time algorithms

- Sorting
- Matrix multiplication
- Optimal binary search trees
- All pairs shortest paths
- Transitive closure
- Single source shortest paths
- Min cost spanning trees

Exponential Time

A program runs in exponential time if there are constants $c, d \in \mathbb{N}$ such that on every input of size n, the program halts within time $d \cdot 2^{n^c}$.

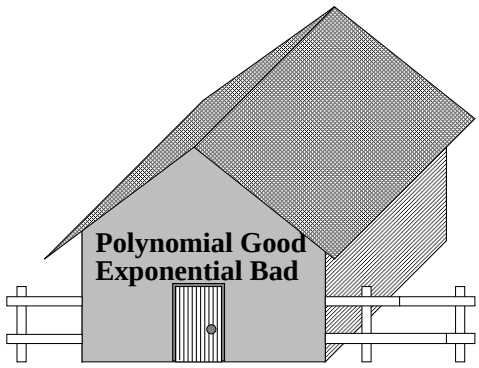
Once again we insist on using the standard encoding scheme.

Example: $n! \leq n^n = 2^{n \log n}$ is counted as exponential time

Exponential time algorithms:

- The knapsack problem
- The clique problem
- The independent set problem
- Ramsey numbers
- The Hamiltonian cycle problem

How do we know there are not polynomial time algorithms for these problems?



Complexity Theory

We will focus on *decision problems*, that is, problems that have a Boolean answer (such as the knapsack, clique and independent set problems).

Define P to be the set of decision problems that can be solved in polynomial time.

If x is a bit string, let $|x|$ denote the number of bits in x.

Define NP to be the set of decision problems of the following form, where $R \in P$, $c \in \mathbb{N}$:

"Given x, does there exist y with $|y| \leq |x|^c$ such that $(x, y) \in R$."

That is, *NP* is the set of existential questions that can be *verified* in polynomial time.

Problems and Languages

The *language* corresponding to a problem is the set of input strings for which the answer to the problem is affirmative.

For example, the language corresponding to the clique problem is the set of inputs strings that encode a graph G and an integer r such that G has a clique of size r.

We will use capital letters such as A or *CLIQUE* to denote the language corresponding to a problem.

For example, $x \in CLIQUE$ means that x encodes a graph G and an integer r such that G has a clique of size r.

Example

The clique problem:

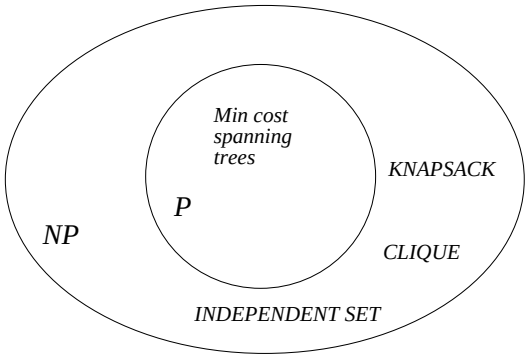
- x is a pair consisting of a graph G and an integer r,
- y is a subset of the vertices in G; note y has size smaller than x,
- R is the set of (x, y) such that y forms a clique in G of size r. This can easily be checked in polynomial time.

Therefore the clique problem is a member of *NP*. The knapsack problem and the independent set problem are members of *NP* too. The problem of finding Ramsey numbers doesn't seem to be in *NP*.

P versus NP

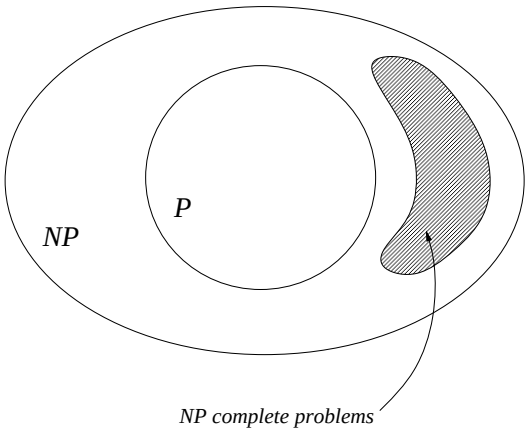
Clearly $P \subseteq NP$.

To see this, suppose that $A \in P$. Then A can be rewritten as "Given x, does there exist y with size zero such that $x \in A$."



It is not known whether $P = NP$.

But it is known that there are problems in *NP* with the property that if they are members of P then $P = NP$. That is, if anything in *NP* is outside of P, then they are too. They are called *NP complete* problems.



Every problem in *NP* can be solved in exponential time. Simply use exhaustive search to try all of the bit strings y.

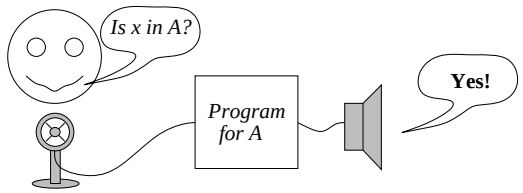
Also known:

- If $P \neq NP$ then there are problems in *NP* that are neither in P nor *NP* complete.
- Hundreds of *NP* complete problems.

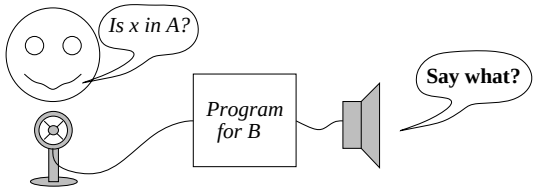
Reductions

A problem A is *reducible* to B if an algorithm for B can be used to solve A. More specifically, if there is an algorithm that maps every instance x of A into an instance $f(x)$ of B such that $x \in A$ iff $f(x) \in B$.

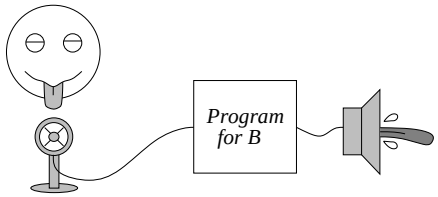
What we want: a program that we can ask questions about A.



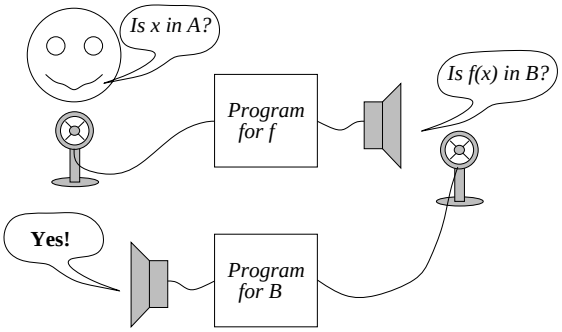
Instead, all we have is a program for B.



This is not much use for answering questions about A.



But, if A is reducible to B, we can use f to translate the question about A into a question about B that has the same answer.



Polynomial Time Reductions

A problem A is *polynomial time reducible* to B, written $A \leq_p B$ if there is a polynomial time algorithm that maps every instance x of A into an instance $f(x)$ of B such that $x \in A$ iff $f(x) \in B$. Note that the size of $f(x)$ can be no greater than a polynomial of the size if x.

Observation 1.

Claim: If $A \leq_p B$ and $B \in P$, then $A \in P$.

Proof: Suppose there is a function f that can be computed in time $O(n^c)$ such that $x \in A$ iff $f(x) \in B$. Suppose also that B can be recognized in time $O(n^d)$.

Given x such that $|x| = n$, first compute $f(x)$ using this program. Clearly, $|f(x)| = O(n^c)$. Then run the polynomial time algorithm for B on input $f(x)$. The compete process recognizes A and takes time

$$O(|f(x)|^d) = O((n^c)^d) = O(n^{cd}).$$

Therefore, $A \in P$.

Proving *NP* Completeness

Polynomial time reductions are used for proving *NP* completeness results.

Claim: If $B \in NP$ and for all $A \in NP$, $A \leq_p B$, then B is *NP* complete.

Proof: It is enough to prove that if $B \in P$ then $P = NP$.

Suppose $B \in P$. Let $A \in NP$. Then, since by hypothesis $A \leq_p B$, by Observation 1 $A \in P$. Therefore $P = NP$.

The Satisfiability Problem

A *variable* is a member of $\{x_1, x_2, \ldots\}$.

A *literal* is either a variable x_i or its complement $\overline{x}_i$.

A *clause* is a disjunction of literals

$$C = x_{i_1} \vee \overline{x}_{i_2} \vee \cdots \vee x_{i_k}.$$

A *Boolean formula* is a conjunction of clauses

$$C_1 \wedge C_2 \wedge \cdots \wedge C_m.$$

Satisfiability (SAT)
INSTANCE: A Boolean formula F.
QUESTION: Is there a truth assignment to the variables of F that makes F true?

Example

$$(x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee \overline{x}_2)$$

This formula is satisfiable: simply take $x_1 = 1$, $x_2 = 0$, $x_3 = 0$.

$$\begin{aligned} & (1 \vee 0 \vee 1) \wedge (0 \vee 1 \vee 1) \wedge \\ & (1 \vee 1) \wedge (0 \vee 1) \\ = \; & 1 \wedge 1 \wedge 1 \wedge 1 \\ = \; & 1 \end{aligned}$$

But the following is not satisfiable (try all 8 truth assignments)

$$(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee \overline{x}_3) \wedge (\overline{x}_2 \vee \overline{x}_3)$$

Cook's Theorem

SAT is *NP* complete.

This was published in 1971. It is proved by showing that every problem in *NP* is polynomial time reducible to SAT. The proof is long and tedious. The key technique is the construction of a Boolean formula that simulates a polynomial time algorithm. Given a problem in *NP* with polynomial time verification problem A, a Boolean formula F is constructed in polynomial time such that

- F simulates the action of a polynomial time algorithm for A
- y is encoded in any assignment to F
- the formula is satisfiable iff $(x, y) \in A$.

Observation 2

Claim: If $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$ (transitivity).

Proof: Almost identical to the Observation 1.

Claim: If $B \leq_p C$, $C \in NP$, and B is *NP* complete, then C is *NP* complete.

Proof: Suppose B is *NP* complete. Then for every problem $A \in NP$, $A \leq_p B$. Therefore, by transitivity, for every problem $A \in NP$, $A \leq_p C$. Since $C \in NP$, this shows that C is *NP* complete.

New *NP* Complete Problems from Old

Therefore, to prove that a new problem C is *NP* complete:

1. Show that $C \in NP$.
2. Find an *NP* complete problem B and prove that $B \leq_p C$.
   (a) Describe the transformation f.
   (b) Show that f can be computed in polynomial time.
   (c) Show that $x \in B$ iff $f(x) \in C$.
      i. Show that if $x \in B$, then $f(x) \in C$.
      ii. Show that if $f(x) \in C$, then $x \in B$, or equivalently, if $x \notin B$, then $f(x) \notin C$.

This technique was used by Karp in 1972 to prove many *NP* completeness results. Since then, hundreds more have been proved.

In the Soviet Union at about the same time, there was a Russian Cook (although the proof of the *NP* completeness of SAT left much to be desired), but no Russian Karp.

The standard text on *NP* completeness:
M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.

What Does it All Mean?

Scientists have been working since the early 1970's to either prove or disprove $P \neq NP$. The consensus of opinion is that $P \neq NP$.

This open problem is rapidly gaining popularity as one of the leading mathematical open problems today (ranking with Fermat's last theorem and the Reimann hypothesis). There are several incorrect proofs published by crackpots every year.

It has been proved that *CLIQUE*, *INDEPENDENT SET*, and *KNAPSACK* are *NP* complete. Therefore, it is not worthwhile wasting your employer's time looking for a polynomial time algorithm for any of them.

Assigned Reading

CLR, Section 36.1–36.3

Algorithms Course Notes
NP Completeness 2

Ian Parberry*

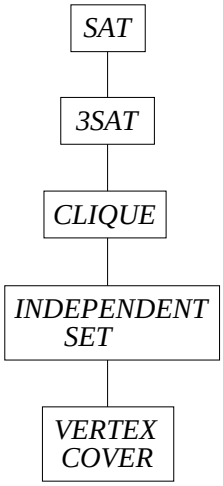
Fall 2001

Summary

The following problems are *NP* complete:

- *3SAT*
- *CLIQUE*
- *INDEPENDENT SET*
- *VERTEX COVER*

Reductions



3SAT

3SAT is the satisfiability problem with at most 3 literals per clause. For example,

$$(x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee \overline{x}_2)$$

3SAT is a subproblem of *SAT*. Does that mean that *3SAT* is automatically *NP* complete?

No:

- Some subproblems of *NP* complete problems are *NP* complete.
- Some subproblems of *NP* complete problems are in *P*.

Claim: *3SAT* is *NP* complete.

Proof: Clearly *3SAT* is in *NP*: given a Boolean formula with n operations, it can be evaluated on a truth assignment in $O(n)$ time using standard expression evaluation techniques.

It is sufficient to show that $SAT \leq_p 3SAT$. Transform an instance of *SAT* to an instance of *3SAT* as follows. Replace every clause

$$(\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k)$$

where $k > 3$, with clauses

- $(\ell_1 \vee \ell_2 \vee y_1)$
- $(\ell_{i+1} \vee \overline{y}_{i-1} \vee y_i)$ for $2 \leq i \leq k-3$
- $(\ell_{k-1} \vee \ell_k \vee \overline{y}_{k-3})$

for some new variables $y_1, y_2, \ldots, y_{k-3}$ different for each clause.

Example

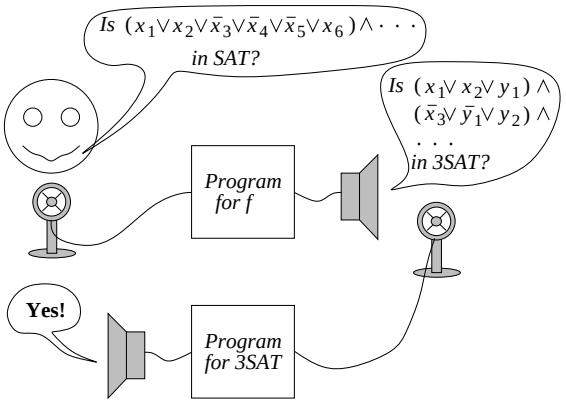
Instance of *SAT*:

$$(x_1 \vee x_2 \vee \overline{x}_3 \vee \overline{x}_4 \vee \overline{x}_5 \vee x_6) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3 \vee x_5) \wedge (x_1 \vee \overline{x}_2 \vee x_6) \wedge (\overline{x}_1 \vee \overline{x}_5)$$

Corresponding instance of *3SAT*:

$$(x_1 \vee x_2 \vee y_1) \wedge (\overline{x}_3 \vee \overline{y}_1 \vee y_2) \wedge$$

$$(\overline{x}_4 \vee \overline{y}_2 \vee y_3) \wedge (\overline{x}_5 \vee x_6 \vee \overline{y}_3) \wedge$$
$$(\overline{x}_1 \vee \overline{x}_2 \vee z_1) \wedge (\overline{x}_3 \vee x_5 \vee \overline{z}_1) \wedge$$
$$(x_1 \vee \overline{x}_2 \vee x_6) \wedge (\overline{x}_1 \vee \overline{x}_5)$$



Back to the Proof

Clearly, this transformation can be computed in polynomial time.

Does this reduction preserve satisfiability?

Suppose the original Boolean formula is satisfiable. Then there is a truth assignment that satisfies all of the clauses. Therefore, for each clause

$$(\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k)$$

there will be some ℓ_i with $1 \leq i \leq k$ that is assigned truth value 1. Then in the new instance of $3SAT$, set each

- ℓ_j for $1 \leq j \leq k$ to the same truth value
- $y_j = 1$ for $1 \leq j \leq i-2$
- $y_j = 0$ for $i-2 < j \leq k-3$.

Every clause in the new Boolean formula is satisfied.

Clause	*Made true by*
$(\ell_1 \vee \ell_2 \vee y_1) \wedge$	y_1
$(\ell_3 \vee \overline{y}_1 \vee y_2) \wedge$	y_2
$\vdots$	
$(\ell_{i-1} \vee \overline{y}_{i-3} \vee y_{i-2}) \wedge$	y_{i-2}
$(\ell_i \vee \overline{y}_{i-2} \vee y_{i-1}) \wedge$	ℓ_i
$(\ell_{i+1} \vee \overline{y}_{i-1} \vee y_i) \wedge$	$\overline{y}_{i-1}$
$\vdots$	
$(\ell_{k-2} \vee \overline{y}_{k-4} \vee y_{k-3}) \wedge$	$\overline{y}_{k-4}$
$(\ell_{k-1} \vee \ell_k \vee \overline{y}_{k-3})$	$\overline{y}_{k-3}$

Therefore, if the original Boolean formula is satisfiable, then the new Boolean formula is satisfiable.

Conversely, suppose the new Boolean formula is satisfiable.

If $y_1 = 0$, then since there is a clause

$$(\ell_1 \vee \ell_2 \vee y_1)$$

in the new formula, and the new formula is satisfiable, it must be the case that one of $\ell_1, \ell_2 = 1$. Hence, the original clause is satisfied.

If $y_{k-3} = 1$, then since there is a clause

$$(\ell_{k-1} \vee \ell_k \vee \overline{y}_{k-3})$$

in the new formula, and the new formula is satisfiable, it must be the case that one of $\ell_{k-1}, \ell_k = 1$. Hence, the original clause is satisfied.

Otherwise, $y_1 = 1$ and $y_{k-3} = 0$, which means there must be some i with $1 \leq i \leq k-4$ such that $y_i = 1$ and $y_{i+1} = 0$. Therefore, since there is a clause

$$(\ell_{i+2} \vee \overline{y}_i \vee y_{i+1})$$

in the new formula, and the new formula is satisfiable, it must be the case that $\ell_{i+2} = 1$. Hence, the original clause is satisfied.

This is true for all of the original clauses. Therefore, if the new Boolean formula is satisfiable, then the old Boolean formula is satisfiable.

This completes the proof that *3SAT* is *NP* complete.

Clique

Recall the *CLIQUE* problem again:

CLIQUE
INSTANCE: A graph $G = (V, E)$, and an integer r.
QUESTION: Does G have a clique of size r?

Claim: *CLIQUE* is *NP* complete.

Proof: Clearly *CLIQUE* is in *NP*: given a graph G, an integer r, and a set V' of at least r vertices, it is easy to see whether $V' \subseteq V$ forms a clique in G using $O(n^2)$ edge-queries.

It is sufficient to show that *3SAT* $\leq_p$ *CLIQUE*. Transform an instance of *3SAT* into an instance of *CLIQUE* as follows. Suppose we have a Boolean formula F consisting of r clauses:

$$F = C_1 \wedge C_2 \wedge \cdots \wedge C_r$$

where each

$$C_i = (\ell_{i,1} \vee \ell_{i,2} \vee \ell_{i,3})$$

Construct a graph $G = (V, E)$ as follows.

$$V = \{(i,1),(i,2),(i,3) \text{ such that } 1 \leq i \leq r\}.$$

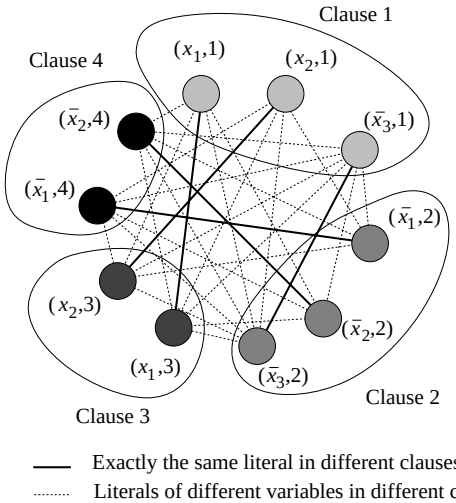
The set of edges E is constructed as follows: $((g,h),(i,j)) \in E$ iff $g \neq i$ and either:

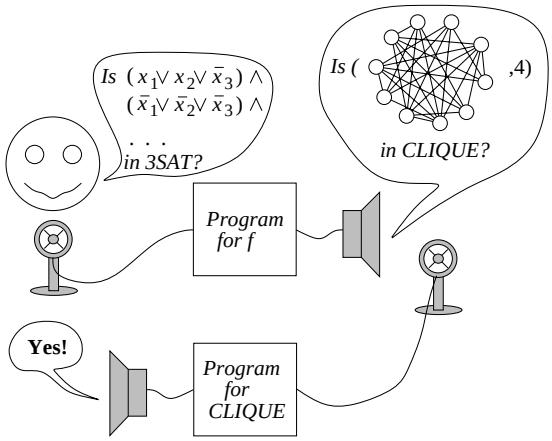
- $\ell_{g,h} = \ell_{i,j}$, or
- $\ell_{g,h}$ and $\ell_{i,j}$ are literals of different variables.

Clearly, this transformation can be carried out in polynomial time.

Example

$$(x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge$$
$$(x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee \overline{x}_2)$$





Back to the Proof

Observation: there is an edge between (g,h) and (i,j) iff literals $\ell_{g,h}$ and $\ell_{i,j}$ are in different clauses and can be set to the same truth value.

Claim that F is satisfiable iff G has a clique of size r.

Suppose that F is satisfiable. Then there exists an assignment that satisfies every clause. Suppose that for all $1 \leq i \leq r$, the true literal in C_i is ℓ_{i,j_i}, for some $1 \leq j_i \leq 3$. Since these r literals are assigned the same truth value, by the above observation, vertices (i, j_i) must form a clique of size r.

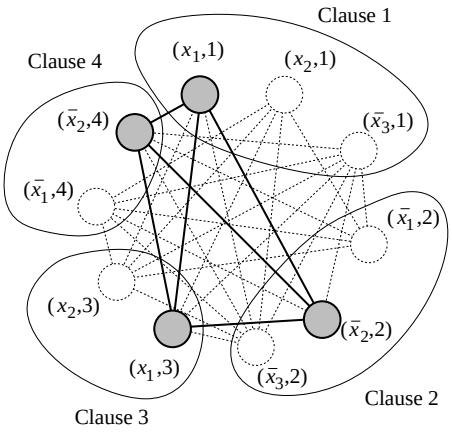
Conversely, suppose that G has a clique of size r. Each vertex in the clique must correspond to a literal in a different clause (since no edges go between vertices representing literals in different clauses). Since there are r of them, each clause must have exactly one literal in the clique. By the observation, all of these literals can be assigned the same truth value. Setting the variables to make these literals true will satisfy all clauses, and hence satisfy the formula F.

Therefore, G has a clique of size r iff F is satisfiable.

This completes the proof that *CLIQUE* is *NP* complete.

Example

$$(x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge$$
$$(x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee \overline{x}_2)$$



Independent Set

Recall the *INDEPENDENT SET* problem again:

INDEPENDENT SET

INSTANCE: A graph $G = (V, E)$, and an integer r.

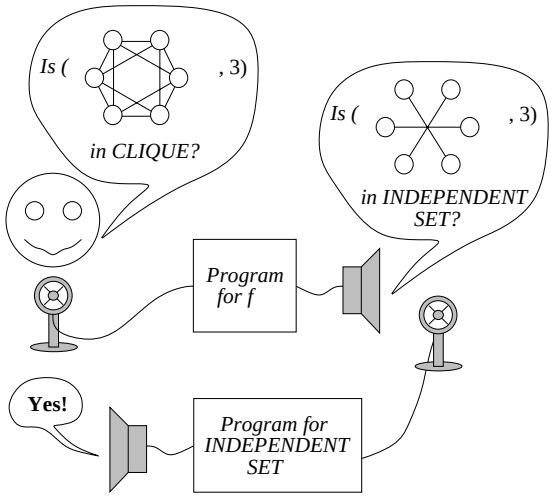
QUESTION: Does G have an independent set of size r?

Claim: *INDEPENDENT SET* is *NP* complete.

Proof: Clearly *INDEPENDENT SET* is in *NP*: given a graph G, an integer r, and a set V' of at least r vertices, it is easy to see whether $V' \subseteq V$ forms an independent set in G using $O(n^2)$ edge-queries.

It is sufficient to show that *CLIQUE* $\leq_p$ *INDEPENDENT SET*.

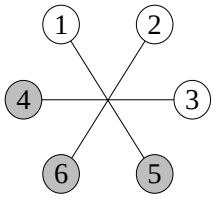
Suppose we are given a graph $G = (V, E)$. Since G has a clique of size r iff $\overline{G}$ has an independent set of size r, and the complement graph $\overline{G}$ can be constructed in polynomial time, it is obvious that *INDEPENDENT SET* is *NP* complete.



Vertex Cover

A *vertex cover* for a graph $G = (V, E)$ is a set $V' \subseteq V$ such that for all edges $(u, v) \in E$, either $u \in V'$ or $v \in V'$.

Example:



VERTEX COVER

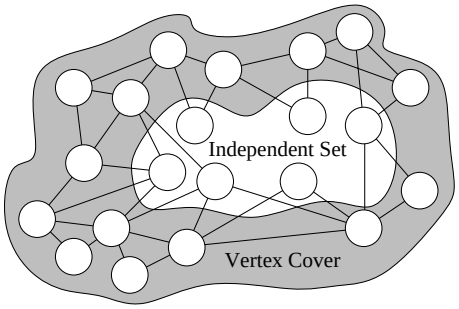
INSTANCE: A graph $G = (V, E)$, and an integer r.

QUESTION: Does G have a vertex cover of size r?

Claim: *VERTEX COVER* is *NP* complete.

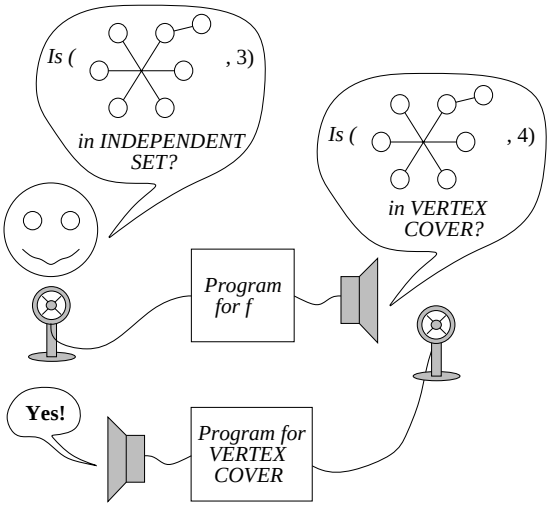
Proof: Clearly *VERTEX COVER* is in *NP*: it is easy to see whether $V' \subseteq V$ forms a vertex cover in G using $O(n^2)$ edge-queries.

It is sufficient to show that *INDEPENDENT SET* $\leq_p$ *VERTEX COVER*. Claim that V' is an independent set iff $V - V'$ is a vertex cover.



Suppose V' is an independent set in G. Then, there is no edge between vertices in V'. That is, every edge in G has at least one endpoint in $V - V'$. Therefore, $V - V'$ is a vertex cover.

Conversely, suppose $V - V'$ is a vertex cover in G. Then, every edge in G has at least one endpoint in $V - V'$. That is, there is no edge between vertices in V'. Therefore, V' is a vertex cover.



Assigned Reading

CLR, Sections 36.4–36.5