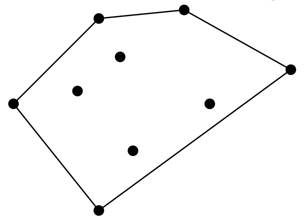
Introduction to Algorithms

Part 1: Divide and Conquer Sorting and Searching

Chapter 1: Convex Hulls: An Example

A polygon is ***convex*** if any line segment joining two points on the boundary stays within the polygon. Equivalently, if you walk around the boundary of the polygon in counterclockwise direction you always take left turns.

The ***convex hull*** of a set of points in the plane is the smallest convex polygon for which each point is either on the boundary or in the interior of the polygon. One might think of the points as being nails sticking out of a wooden board: then the convex hull is the shape formed by a tight rubber band that surrounds all the nails. A ***vertex*** is a corner of a polygon. For example, the highest, lowest, leftmost and rightmost points are all vertices of the convex hull. Some other characterizations are given in the exercises.



We discuss three algorithms: Graham Scan, Jarvis March and Divide & Conquer. We present the algorithms under the ***assumption*** that:

- *no 3 points are collinear (on a straight line)*

1.1 Graham Scan

The idea is to identify one vertex of the convex hull and sort the other points as viewed from that vertex. Then the points are scanned in order.

Let x_0 be the leftmost point (which is guaranteed to be in the convex hull) and number the remaining points by angle from x_0 going counterclockwise: $x_1, x_2, \ldots, x_{n-1}$. Let $x_n = x_0$, the chosen point. Assume that no two points have the same angle from x_0.

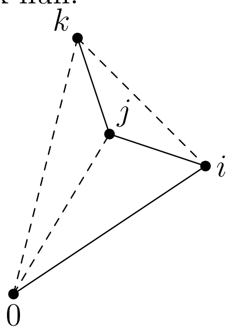
The algorithm is simple to state with a single stack:

Graham Scan

```
1. Sort points by angle from $x_0$
2. Push $x_0$ and $x_1$. Set $i$=2
3. While $i \leq n$ do:
          If $x_i$ makes left turn w.r.t. top 2 items on stack
               then { push $x_i$; $i$++ }
               else { pop and discard }
```

To prove that the algorithm works, it suffices to argue that:

- *A discarded point is not in the convex hull.* If x_j is discarded, then for some $i < j < k$ the points $x_i \to x_j \to x_k$ form a right turn. So, x_j is inside the triangle x_0, x_i, x_k and hence is not on the convex hull.



- *What remains is convex.* This is immediate as every turn is a left turn.

The running time: Each time the while loop is executed, a point is either stacked or discarded. Since a point is looked at only once, the loop is executed at most $2n$ times. There is a constant-time subroutine for checking, given three points in order, whether the angle is a left or a right turn (Exercise). This gives an $O(n)$ time algorithm, apart from the initial sort which takes time $O(n \log n)$. (Recall that the notation $O(f(n))$, pronounced “order $f(n)$”, means “asymptotically at most a constant times $f(n)$”.)

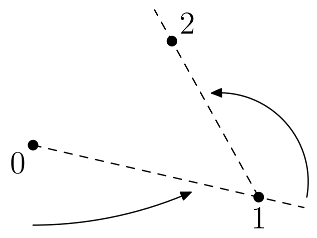
1.2 Jarvis March

This is also called the ***wrapping algorithm***. This algorithm finds the points on the convex hull ***in the order*** in which they appear. It is quick if there are only a few points on the convex hull, but slow if there are many.

Let x_0 be the leftmost point. Let x_1 be the first point counterclockwise when viewed from x_0. Then x_2 is the first point counterclockwise when viewed from x_1, and so on.

Jarvis March

```
$i = 0$
while not done do
    $x_{i+1}$ = first point counterclockwise from $x_i$
```



Finding x_{i+1} takes linear time. The while loop is executed at most n times. More specifically, the while loop is executed h times where h is the number of vertices on the convex hull. So Jarvis March takes time $O(nh)$.

The best case is $h = 3$. The worst case is $h = n$, when the points are, for example, arranged on the circumference of a circle.

1.3 Divide and Conquer

Divide and Conquer is a popular technique for algorithm design. We use it here to find the convex hull. The first step is a Divide step, the second step is a Conquer step, and the third step is a Combine step.

The idea is to:

Divide and conquer

1. Divide the n points into two halves.
2. Find convex hull of each subset.
3. Combine the two hulls into overall convex hull.

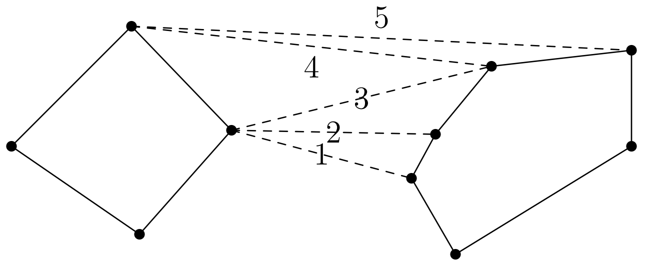
Part 2 is simply two ***recursive*** calls. The first point to notice is that, if a point is in the overall convex hull, then it is in the convex hull of any subset of points that contain it. (Use characterization in exercise.) So the task is: given two convex hulls find the convex hull of their union.

◇ ***Combining two hulls***

It helps to work with convex hulls that do not overlap. To ensure this, all the points are ***presorted*** from left to right. So we have a left and right half and hence a left and right convex hull.

Define a ***bridge*** as any line segment joining a vertex on the left and a vertex on the right that does not cross the side of either polygon. What we need are the ***upper*** and ***lower*** bridges. The following produces the upper bridge.

1. Start with any bridge. For example, a bridge is guaranteed if you join the rightmost vertex on the left to the leftmost vertex on the right.
2. Keeping the left end of the bridge fixed, see if the right end can be raised. That is, look at the next vertex on the right polygon going clockwise, and see whether that would be a (better) bridge. Otherwise, see if the left end can be raised while the right end remains fixed.
3. If made no progress in (2) (cannot raise either side), then stop else repeat (2).



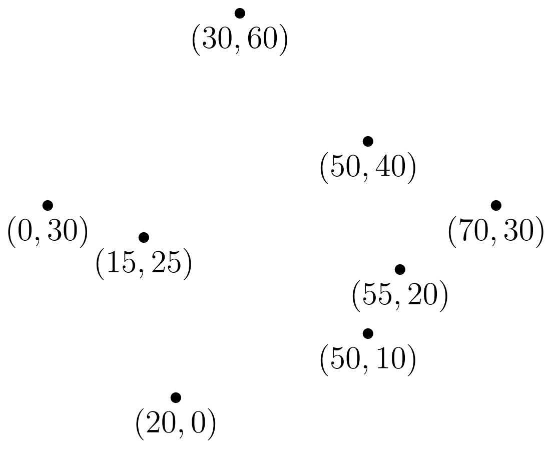
We need to be sure that one will eventually stop. Is this obvious?

Now, we need to determine the running time of the algorithm. The key is to perform step (2) in constant time. For this it is sufficient that each vertex has a pointer to the next vertex going clockwise and going counterclockwise. Hence the choice of data structure: we store each hull using a ***doubly linked circular linked list***.

It follows that the total work done in a merge is proportional to the number of vertices. And as we shall see in the next chapter, this means that the overall algorithm takes time $O(n \log n)$.

Exercises

1. Find the convex hulls for the following list of points using the three algorithms presented.



2. Give a quick calculation which tells one whether three points make a left or a right turn.

3. Discuss how one might deal with collinear points in the algorithms.

4. Show that a point D is on the convex hull if and only if there do not exist points A, B, C such that D is inside the triangle formed by A, B, C.

5. ⓔ Assume one has a fast convex hull subroutine that returns the convex hull ***in order***. Show how to use the subroutine to sort.

Chapter 2: Divide and Conquer

In this chapter we consider divide and conquer: this is essentially a special type of recursion. In divide and conquer, one:

divides the problem into pieces,
then conquers the pieces,
and re-assembles.

An example of this approach is the convex hull algorithm. We divide the problem into two pieces (left and right), conquer each piece (by finding their hulls), and re-assemble (using an efficient merge procedure).

2.1 Master Theorem for Recurrences

One useful result is the "Master Theorem" for a certain family of recurrence relations:

Consider the recurrence

$$T(n) = aT(n/b) + f(n).$$

Then

- if $f(n) \ll n^{\log_b a}$ then $T(n) = O(n^{\log_b a})$.
- if $f(n) \approx n^{\log_b a}$ then $T(n) = O(n^{\log_b a} \log n)$.
- if $f(n) \gg n^{\log_b a}$ and $\lim\limits_{n\to\infty} af(n/b)/f(n) < 1$ then $T(n) = O(f(n))$.

Example. If $T(n)$ denotes the time taken for the divide-and-conquer convex hull algorithm (ignoring the initial sort), then we obtain the recurrence

$$T(n) = 2T(n/2) + O(n).$$

This solves to $O(n \log n)$.

2.2 Multiplying Long Integers

Another example of divide-and-conquer is the problem of multiplying long integers. The following is based on the discussion by Aho, Hopcroft and Ullman.

Consider the problem of multiplying two n-bit integers X and Y. At school we learnt long multiplication: if we take two n-digit numbers then long multiplication takes $O(n^2)$ (quadratic) time. (Why?)

We can apply divide and conquer. For example, suppose we split X and Y into two halves:

$$X := \boxed{A \mid B} \qquad Y := \boxed{C \mid D}$$

Then $X = A2^{n/2} + B$ and $Y = C2^{n/2} + D$. The product of X and Y can now be written as:

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

This product requires four multiplications and three additions. To add two integers takes linear time. (Why?) So we obtain the recurrence:

$$T(n) = 4T(n/2) + O(n)$$

Alas, the solution to this is again quadratic, and in practice this algorithm is worse than normal long multiplication.

However, consider the following formula:

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

At first glance this looks more complicated. But we only need three multiplications to do this: AC, BD and $(A-B)(D-C)$. Therefore we obtain the recurrence:

$$T(n) = 3T(n/2) + cn$$

whose solution is $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$.

There is some more work needed to write this down in a program. We need to consider shifts and also deal with negative integers. Left as an exercise.

2.3 Matrix Multiplication

Consider the problem of multiplying two $n \times n$ matrices. Computing each entry in the product takes n multiplications and there are n^2 entries for a total of $O(n^3)$ work. Can one do better?

Strassen devised a better method which has the same basic flavor as the multiplication of long integers. The key idea is to save one multiplication on a small problem and then use recursion.

We look first at multiplying two 2×2 matrices. This would normally take 8 multiplications, but it turns out we can do it with 7. You can check the following. If

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \qquad \text{and} \qquad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

and one calculates the following products:

$$\begin{aligned}
m_1 &= (a_{21} + a_{22} - a_{11})(b_{22} - b_{12} + b_{11}) \\
m_2 &= a_{11}b_{11} \\
m_3 &= a_{12}b_{21} \\
m_4 &= (a_{11} - a_{21})(b_{22} - b_{12}) \\
m_5 &= (a_{21} + a_{22})(b_{12} - b_{11}) \\
m_6 &= (a_{12} - a_{21} + a_{11} - a_{22})b_{22} \\
m_7 &= a_{22}(b_{12} + b_{21} - b_{11} - b_{22})
\end{aligned}$$

then the product of A and B is given by

$$AB = \begin{pmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 + m_7 & m_1 + m_2 + m_4 + m_5 \end{pmatrix}$$

Wow! But what use is this?

What we do is to use this idea recursively. If we have a $2n \times 2n$ matrix, we can split it into four $n \times n$ submatrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad \text{and} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

and form seven products M_1 up to M_7, and the overall product AB can thus be calculated with seven multiplications of $n \times n$ matrices. Since adding matrices is clearly proportional to their size, we obtain a recurrence relation:

$$f(n) = 7f(n/2) + O(n^2)$$

whose solution is $O(n^{\log_2 7})$, an improvement on $O(n^3)$.

2.4 Modular Exponentiation

Another example of divide-and-conquer is the modular exponentiation problem. The following is based on the discussion by Brassard and Bratley.

Suppose Alice and Bob wish to establish a secret but do not initially share any common secret information. Their problem is that the only way they can communicate is by using a telephone, which is being tapped by Eve, a malevolent third party. They do not want Eve to know their secret. To simplify the problem, we assume that, although Eve can overhear conversations, she can neither add nor modify messages.

The goal is to find a ***protocol*** by which Alice and Bob can attain their ends.

The first solution to this problem was given in 1976 by Diffie and Hellman. The idea is the following.

1. Alice and Bob agree openly on some integer p with a few hundred decimal digits, and on some other integer g between 2 and $p - 1$. (The security of the secret they intend to establish is not compromised should Eve learn these two numbers.)

2. Alice and Bob choose randomly and independently of each other two positive integers A and B less than p.

3. Alice computes $a = g^A \bmod p$ (the mod function is the % of Java/C), and transmits this result to Bob. Similarly, Bob sends Alice the value $b = g^B \bmod p$.

4. Alice computes $x = b^A \bmod p$ and Bob calculates $y = a^B \bmod p$.

Now $x = y$ since both are equal to $g^{AB} \bmod p$. This value is therefore a piece of information shared by Alice and Bob. It can now be used as the key in a conventional cryptographic system.

At the end of the exchange, Eve has the values of p, g, a, and b only. One way for her to deduce x would be to find an integer A' such that $a = g^{A'} \bmod p$, and then to proceed like Alice to calculate $x' = b^{A'} \bmod p$. Under some simple algebraic conditions, such A' is necessarily equal to A, and the secret is correctly computed by Eve in this case.

Calculating A' from p, g and a is called the problem of the ***discrete logarithm***. There exists an obvious algorithm to solve it. (If the logarithm does not exist, the algorithm returns the value p. For instance, there is no integer A such that $3 = 2^A \bmod 7$.)

```
dlog (base $g$,answer $a$,modulus $p$)
    $A \leftarrow 0$; $x \leftarrow 1$
    repeat
        $A \leftarrow A + 1$
        $x \leftarrow xg$
    until ($a = x$ mod $p$) or ($A = p$)
    return $A$
```

This algorithm takes an unacceptable amount of time, since it makes $p/2$ trips round the loop on the average. If each trip round the loop takes 1 microsecond, this average time is more than the age of Earth even if p only has 24 decimal digits. Although there are more efficient algorithms for calculating discrete logarithms, none is able to solve a randomly chosen instance in a reasonable amount of time when p is a prime with several hundred decimal digits. Furthermore, there is no known way of recovering x from p, g, a, and b that does not involve calculating a discrete logarithm. For the time being, it seems therefore that this method of providing Alice and Bob with a shared secret is sound, although no-one has yet been able to prove this.

But is this a joke? If Eve needs to be able to calculate discrete logarithms efficiently to discover the secret shared by Alice and Bob, it is equally true that Alice and Bob must be able to calculate efficiently exponentiations of the form $a = g^A \bmod p$. The obvious algorithm for this is no more subtle or efficient than the one for discrete logarithms.

The fact that

$$xyz \bmod p = ((xy \bmod p) \times z) \bmod p$$

for every x, y, z, and p allows us to avoid accumulation of extremely large integers in the loop. (The same improvement can be made in dlog, which is necessary if we hope to execute each trip round the loop in 1 microsecond.)

Happily for Alice and Bob, there is a more efficient algorithm for computing the exponentiation. An example will reveal the basic idea.

$$x^{25} = (((x^2x)^2)^2)^2x$$

Thus x^{25} can be obtained with just two multiplications and four squarings. This formula arises because $x^{25} = x^{24}x$, $x^{24} = (x^{12})^2$, and so on.

This idea can be generalized to obtain a divide-and-conquer algorithm.

```
dexpo ($g$,$A$,$p$)      % calculates $a = g^A$ mod $p$
    if $A = 0$ then return 1
    if $A$ odd then {
        $a \leftarrow$ dexpo($g$, $A - 1$, $p$)
        return($ag$ mod $p$)
        }
    else {
        $a \leftarrow$ dexpo ($g$, $A/2$, $p$)
        return($a^2$ mod $p$)
        }
```

This requires $O(\log g)$ multiplications.

2.5 Good Algorithms

Recursion is often easy to think of but does not always work well. Consider the problem of calculating the nth Fibonacci number, which is defined by

$$a_n = a_{n-1} + a_{n-2}$$

with initial values $a_1 = 1$ and $a_2 = 1$. Recursion is terrible! Rather do iteration: calculate a_3 then a_4 etc.

The key to good performance in divide and conquer is to partition the problem as evenly as possible, and to save something over the naïve implementation.

Exercises

1. Illustrate the multiplication of 1234 and 5678 using the algorithm described above.

2. Create a class for long integers. Then provide the operations shift, addition and multiplication. Then calculate the square of the 50th Fibonacci number.

3. Use Strassen's algorithm to multiply

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \qquad \text{and} \qquad \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

4. Consider the recurrence:

$$f(n) = 2f(n/2) + cn \qquad f(1) = 0$$

   Prove that this has the solution $f(n) = cn \log_2 n$ for n a power of 2.

5. Calculate $7^{2004} \bmod 13$

6. Consider the following program:

```
function Fibonacci(n)
  if n<2 then return n
  else return Fibonacci(n-1) + Fibonacci(n-2)
```

   Analyze the time used for this algorithm.

7. ⓔ a) Give a fast algorithm to computer F^n where F is a 2-by-2 matrix and n a positive integer. It should run in time $O(\log n)$.

   b) Consider the matrix

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

   Show that

$$F^n = \begin{pmatrix} a_{n-1} & a_n \\ a_n & a_{n+1} \end{pmatrix}$$

   where a_n denotes the nth Fibonacci number.

   c) Hence give a fast algorithm to determine the nth Fibonacci number.

8. ⓔ Consider an n-bit binary counter. This is used to count from 0 up to $2^n - 1$. Each time the counter is incremented, the natural add-one algorithm is follows: one works in from the right-most bit changing bits until there is no carry. Determine the ***average*** number of bit operations needed per increment.

Chapter 3: Sorting Revisited

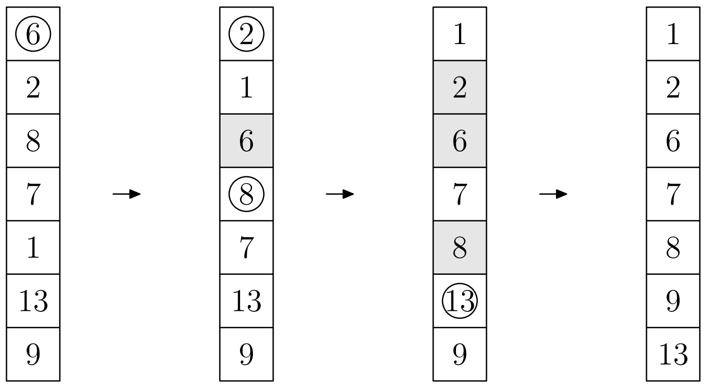
In this chapter we revisit the sorts Quicksort and Merge Sort which you have probably seen before.

3.1 Quicksort

Quicksort is a procedure for sorting that was invented by Hoare in 1962. It uses divide and conquer. Say one starts with n distinct numbers in a list. Call the list A. Then Quicksort does:

```
Quicksort (A:valuelist)

1: Choose a Key (often just the first element in the list).
2: Split the list into two sublists $A_<$ and $A_>$ (called buckets). The bucket $A_<$ contains
   those elements smaller than the Key and the bucket $A_>$ contains those elements
   larger than the Key.
3: Use Quicksort to sort both buckets recursively.
```



There remain questions about:

1) Implementation
2) Speed
3) Storage required
4) Is this the best we can do?
5) Problems with the method.

The beauty of Quicksort lies in the storage requirement: the sorting takes place "in situ" and very little extra memory is required.

3.2 How Fast is Quicksort?

To analyze the speed, we focus on the number of comparisons between data items. We count only the comparisons. We will come back to whether this is valid or not. But even this counting is hard to do.

◇ ***Worst case***

What is the worst case scenario? A very uneven split. For example, our Key might be the minimum value. Then we compare it with every element in the list only to find that the bucket $A_<$ is empty. Then when we sort $A_>$ we might again be unlucky and have the minimum value of that bucket as the Key. In fact, if the list was already sorted we would end up comparing every element with every other element for a total of $\binom{n}{2}$ comparisons.

We can analyze the ***worst case*** (when the list is in fact already sorted) another way: The first step breaks the list up into $A_<$ which is empty and $A_>$ which contains $n-1$ items. This takes $n-1$ comparisons. Then $A_>$ is split using $n-2$ comparisons, and leaves a basket of $n-2$ items. So number of comparisons is:

$$(n-1)+(n-2)+(n-3)+\ldots+2+1 = n(n-1)/2 \approx n^2/2.$$

◇ ***Best case***

The best case is when the list is split evenly each time. (Why?) In this case the size of the largest bucket goes down by a factor of 2 each time.

At the top level we use $n-1$ comparisons and then have to sort the buckets $A_<$ and $A_>$ which have approximately $(n-1)/2$ elements each. To make the arithmetic simpler, let's say that we use n comparisons and end up with two buckets of size $n/2$.

Let $f(n)$ denote the number of comparisons needed by Quicksort in the best case. We then have the recurrence relation:

$$f(n) = n + 2f(n/2)$$

with the boundary condition that $f(1) = 0$. One can then check that the solution to this, at least in the case that n is a power of 2, is

$$f(n) = n \log_2 n$$

◇ ***Average case***

Of course, we are actually interested in what happens in real life. Fortunately, the typical behavior of Quicksort is much more like $n \log_2 n$ than n^2. We do not explore this here—but see Exercise 6.

3.3 Binary Sort

An iterative method for sorting is also available. At each stage one inserts the new element in the already sorted list.

Finding the place for the new element is known as ***binary search***. Recall that one compares the new element with the middle element of the list. If the new element is smaller than the middle element, one recursively searches in the top half of the list. If the new element is larger than the middle element, one searches in the bottom half.

```
BinarySort (A:valuelist)

For each element of A in turn:
    Insert the new element into current sorted list using binary search.
```

A binary search takes $\log_2 m$ comparisons (rounded up) where m is the number of elements in the list. In binary sort, at each stage one does a binary search to find the place where the new element goes. So each stage takes $\log_2 m$ comparisons where m is the number of elements in the list. This provides a guarantee of at most $n \log_2 n$ comparisons.

But this sort is seldom used. Why?

3.4 Merge Sort

Another example of recursion is the following.

```
MergeSort (A:valuelist)

    1: Arbitrarily split the list into two halves.

    2: Use MergeSort to sort each half.

    3: Merge the two sorted halves.
```

One divides the list into two pieces just by slicing in the middle. Then one sorts each piece using recursion. Finally one is left with two sorted lists. And must now combine them. The process of combining is known as ***merging***.

How quickly can one merge? Well, think of the two sorted lists as stacks of exam papers sitting on the desk with the worst grade on top of each pile. The worst grade in the entire list is either the worst grade in the first pile or the worst grade in the second pile. So compare the two top elements and set the worst aside. The second-worst grade is now found by comparing the top grade on both piles and setting it aside. Etc.

Why does this work?

How long does merging take? Answer: One comparison for every element placed in the sorted pile. So, roughly n comparisons where n is the total number of elements in the combined list. (It could take less. When?)

Merge Sort therefore obeys the following recurrence relation

$$M(n) = n + 2M(n/2).$$

(Or rather the number of comparisons is like this.)

We've seen before that the solution to this equation is $n \log_2 n$. Thus Merge Sort is an $n \log n$ algorithm in the worst case.

What is the drawback of this method?

3.5 Optimality of Sorting

So we have sorting algorithms that take time proportional to $n \log_2 n$. Is this the best we can do? Yes, in some sense, as we will show later.

Exercises

1. Illustrate the behavior of Quicksort, Binary Sort and Merge Sort on the following data: 2, 4, 19, 8, 9, 17, 5, 3, 7, 11, 13, 16
2. Discuss the advantages and disadvantages of the three sorts introduced here.
3. In your favorite programming language, code up the Quicksort algorithm. Test it on random lists of length 10, 100, 1000 and 10 000, and comment on the results.
4. Suppose we have a list of n numbers. The list is guaranteed to have a number which appears more than $n/2$ times on it. Devise a good algorithm to find the Majority element.
5. ⓔ Repeat the previous question but now assume one can only test two numbers for equality or inequality.
6. ⓔ Let $q(n)$ be the average number of data-comparisons required for Quicksort on a randomly generated list.

   a) Explain why $q(1) = 0$, $q(2) = 1$ and $q(3) = 2\frac{2}{3}$

b) Explain why the following recurrence holds:

$$q(n) = n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} q(i)$$

c) Show that the solution to the above recurrence is:

$$q(n) = 2(n+1) \sum_{k=1}^{n} \frac{1}{k} - 4n$$

d) Use a calculator, computer or mathematical analysis to look at the asymptotics of $q(n)$.

7. a) Suppose we have a collection of records with a 1-bit key K. Devise an efficient algorithm to separate the records with $K = 0$ from those with $K = 1$.

   b) What about a 2-bit key?

   c) What has all this to do with sorting? Discuss.

Chapter 4: Parallel Sorting

We consider parallel algorithms where the processors have limited capabilities and where there is a definite topology: a processor can communicate with only a few processors. Also there is a global clock (SIMD). We consider the sorting problem where each processor maintains one data value, and there is a specific target order which must be established. At each step, a processor can do a bit of computation, and swap data value with one of its neighbors.

This chapter is based on the discussion by Leighton.

4.1 Odd-Even Transposition Sort on a Linear Array

A ***linear array*** has the processors in a row: each processor can communicate with only the one before it and the one after it. We give a linear array sorting algorithm that takes precisely N steps, where N is the number of processors and the number of data values. $N - 1$ is a lower bound, because one might need to get a data value from the last cell to the first cell.

Odd-even transposition sort.
At odd steps, compare contents of cell 1 and 2, cells 3 and 4, etc, and swap values if necessary so that the smaller value ends up in the leftmost cell. At even steps do the same but for the pairs 2 and 3, 4 and 5, etc.

EXAMPLE. 46278135.
46-27-81-35 → 4-62-71-83-5 → 42-61-73-85 → 2-41-63-75-8 → 21-43-65-78 → 1-23-45-67-8

To prove that this works, focus on the smallest element. It is clear that after a while the smallest element ends up in the leftmost cell. At that stage we can look at the second smallest element and so on. With a bit more work one can show that the whole process runs in N steps.

4.2 Odd-Even Merge Sort

This sort is a famous parallel algorithm. It can be implemented on the topology called a hypercube, but we will consider only the PRAM version. PRAM is ***parallel random access machine***: in this model there is global memory accessible to all processors. In particular all the data is accessible to all processors (though each can only look at one piece in one clock-step). The algorithm runs in $O(\log^2 N)$ steps.

Merge Sort.
Given list of length N, split into N lists of length 1. Then in parallel merge pairs of length-1 lists to form $N/2$ sorted lists of length 2. These lists are then merged into $N/4$ sorted lists of length 4, and so on. At the end, we merge two sorted lists of length $N/2$ into the final sorted list.

But the merge step we described in Merge Sort looks inherently sequential. The new idea is as follows:

Odd-even merge.
To merge two sorted lists A and B, split A into two sublists A_{even} and A_{odd} containing the even-numbered entries and the odd-numbered entries respectively; similarly split B into B_{even} and B_{odd}. Since A and B are sorted, the four sublists are sorted. Then use recursion to merge A_{even} and B_{odd} to form C, and merge A_{odd} and B_{even} to form D. Finally, merge C and D.

At first glance, this appear ridiculous: we have made no progress. But it turns out that merging C and D is trivial. The fact is, the first two elements of the sorted list are the top elements in C and D in some order, the third and forth elements in the sorted list are the second elements of C and D in some order, and so on. (See below.)

Example. To merge A=1467 and B=2358, C is merge of 47 and 25, D is merge of 16 and 38, so C is 2457 and D is 1368.

4.3 Proof and Analysis of Odd-Even Merge Sort

The proof is made simpler by considering only small and big elements. For any value T (threshold), we define a data element to be ***T-small*** if it less than T and ***T-big*** if it is T or greater. Then:

> Observation. A list is sorted if and only if for all choices of T all T-small elements come before all T-big elements.

Now, notice that A_{even} and A_{odd} have almost the same number of T-small elements, as do B_{even} and B_{odd}. So C and D have almost the same number of T-small elements. Indeed, the pairing of A_{even} with B_{odd} ensures that if A_{even} and A_{odd} have different amounts of small elements, and B_{odd} and B_{even} different amounts too, then C and D have the same number of smalls. Thus in the final list, all the smalls will occur before all the bigs.

It remains to determine how long this process takes. The final merge of C and D takes $O(1)$ time, and the two submerges occur in parallel. So the overall merge takes $O(\log N)$ steps. There are $O(\log N)$ merge-phases, and so the overall algorithm runs in time $O(\log^2 N)$. Provided there are $O(N)$ processors.

Exercises

1. For the supplied data, illustrate the stages of the given sort and count the actual number of steps. (i) odd-even transposition sort (ii) odd-even merge sort.

   9 3 6 1 7 8 2 4

2. An ***oblivious*** sorting algorithm is one which consists entirely of a prescribed sequence of compare-and-swap operations: each step is to compare two elements and swap them if necessary so that they are in a particular order.

   (a) Explain why the odd-even transposition sort is oblivious, but neither Quicksort nor Merge Sort is oblivious.

   (b) ⓔ Prove that if an oblivious algorithm sorts a list consisting entirely of 0s and 1s, then it sorts a general list.

Chapter 5: Finding the Median

The ***median*** of a collection is the middle value (in sorted order). For example, the collection $\{3, 1, 6, 2, 8, 4, 9\}$ has a median of 4: There are three values smaller than 4 and three values larger. The answer to a median question when there is an even number of values is the smaller of the two middle values. (There are other possible definitions but this is convenient.)

We want an algorithm for finding the median of a collection of n values. One possibility would be to sort the list. But one hopes for a faster method. One can try recursion or iteration, but at first glance they don't seem to work.

Blum, Floyd, Pratt, Rivest and Tarjan provided a linear-time algorithm: it uses $O(n)$ comparisons. It is a bit hairy at places.

5.1 A Generalization

One approach to the median problem is to define a generalization: the ***rank selection*** problem. The input is a collection of n numbers and a rank k. The output must be the value that is the kth smallest value. For example, for the above collection, 4 has rank 4 and 8 has rank 6. The minimum has rank 1.

Thus the median of a collection with n values has rank $\lceil n/2 \rceil$ (round up): it's $n/2$ if n is even and $(n+1)/2$ if n is odd.

5.2 Using Median for Rank Selection

Okay. We do not know how to find the median. But I now argue that if (and that's a big if) we have an algorithm for median, then we can derive an algorithm for rank-selection:

> *We use the median algorithm to break the list in half. And keep only the relevant half.*

How much work does this take?

```
RankSelect (A:valuelist, k:integer)

  R1: Calculate the median $m$ of the list $A$.

  R2: Determine the set $A_<$ of elements smaller than the median.
      Determine the set $A_>$ of elements larger than the median.

  R3: If $k < \lceil n/2 \rceil$ then return RankSelect($A_<$, $k$)
      If $k > \lceil n/2 \rceil$ then return RankSelect($A_>$, $k - \lceil n/2 \rceil$)
      Else return $m$
```

Suppose that the median procedure takes cn comparisons where c is some constant. (That is, suppose we have a linear-time median algorithm.) The number of comparisons for Step R1 the first time is cn. The number for Step R2 the first time is n. The total for the first two steps is therefore $n(1 + c)$ comparisons.

The second time we have a list half the size. So our work is half that of the original. The number of comparisons for Step R1 the second time is $cn/2$. The number for Step R2 the second time is $n/2$. The total for the first two steps is $n(1 + c)/2$ comparisons.

Hence the total number of comparisons is

$$n(1+c) + \frac{n(1+c)}{2} + \frac{n(1+c)}{4} + \frac{n(1+c)}{8} + \frac{n(1+c)}{16} + \frac{n(1+c)}{32} + \dots$$

If this sum continues forever it would add up to

$$2n(1 + c).$$

So the total work is at most $2(1+c)n$. This is linear! (The beauty of the geometric series.)

5.3 The Pseudo-Median

We still don't have a median algorithm. There are two ideas needed:
(1) that the above algorithm is still efficient if one uses only an ***approximation*** to the median; and
(2) one can actually find an approximation to the median.

We define a ***pseudo-median*** as any value such that at least 30% of the values lie above the pseudo-median and at least 30% of the values lie below the pseudo-median. (The value 30% is chosen to make the presentation easier.)

Suppose we have an algorithm that finds a pseudo-median. Then we can use it for rank selection in exactly the same way as above. (And hence we can find the median!)

```
RankSelect (A:valuelist, k:integer)

R1: Calculate a pseudo-median $m$ of the list $A$.

R2: Determine the set $A_<$ of elements smaller than $m$.
    Determine the set $A_>$ of elements larger than $m$.
    Calculate the rank $r$ of $m$ in $A$.

R3: If $k < r$ then return RankSelect($A_<$, $k$)
    If $k > r$ then return RankSelect($A_>$, $k - r$)
    Else return $m$
```

Hence we have a function RankSelect which uses as a subroutine the function Pseudo-Median.

5.4 Finding a Pseudo-Median

How do we find a pseudo-median? What we do is:

*take a **sample** of the data and find the median of the sample.*

This algorithm is bizarre but beautiful: we now have a function Pseudo-Median which uses as a subroutine the function RankSelect.

```
Pseudo-Median (B:valuelist)

P1: Break up the list B into quintets (groups of 5).

P2: Calculate the median in each quintet (by brute-force). Call the result the representative of the quintet. There are n/5 representatives.

P3: Return the median of the set of representatives:
    RankSelect(reps,n/10)
```

Does this produce a pseudo-median?

Well, consider the resultant value m_r which is the median of the representatives. This value is bigger than $n/10$ representatives. And for each representative that m_r is bigger than, m_r is bigger than two other members of the quintet. Thus there are at least $3n/10$ values which m_r is guaranteed to be bigger than. And similarly there are $3n/10$ values which m_r is guaranteed to be smaller than. Hence m_r is guaranteed to be a pseudo-median.

Note that the intricate recursion: RankSelect calls PseudoMedian which calls RankSelect and so on. The reason why this works is that the problem size is getting smaller at each stage. (When implementing this algorithm one must be very careful not to get an infinite loop...)

5.5 Analysis

The calculation of the median in a quintet takes at most 10 comparisons. (Since sorting the quintet takes this many comparisons!) So the total number of comparisons for Step P2 is $2n$.

Now, let us return to the RankSelect algorithm modified to use PseudoMedian. Let $g(n)$ be the number of comparisons needed by RankSelect on a list of size n in the worst case. Then the number of comparisons for the three steps are:

Step R1: $2n + g(n/5)$
Step R2: n
Step R3: $g(7n/10)$

This gives us the recurrence:

$$g(n) = g(n/5) + g(7n/10) + 3n$$

which has solution $g(n) = 30n$. (Plug it in and check!)

Exercises

1. Implement the median-finding algorithm.
2. Run your program on random lists of length 10, 100, 1000 and 10 000. Comment on the results.
3. Why quintets? Use either theory or a modification of your program to compare triplets with quintets.

Chapter 6: Books

I have made much use of:

• *Data Structures and Algorithms*,
A. Aho, J. Hopcroft and J. Ullman

• *Fundamentals of Algorithmics*,
G. Brassard and P. Bratley

• *Introduction to Algorithms*,
T. Cormen, C. Leiserson and R. Rivest

• *Introduction to Parellel Algorithms and Architectures: Arrays, Trees, Hypercubes*,
F.T. Leighton

• *Data Structures and Algorithm Analysis,*
M.A. Weiss

Introduction to Algorithms

Part 2: Greedy Algorithms Dynamic Programming Graph Algorithms

Chapter 1: Greedy Algorithms and Spanning Trees

In a greedy algorithm, the optimal solution is built up one piece at a time. At each stage the best feasible candidate is chosen as the next piece of the solution. There is no back-tracking.

These notes are based on the discussion in Brassard and Bratley.

1.1 The Generic Greedy Algorithm

The elements of a greedy algorithm are:

1. A set C of candidates
2. A set S of selected items
3. A ***solution check***: does the set S provide a solution to the problem (ignoring questions of optimality)?
4. A ***feasibility check***: can the set S be extended to a solution to the problem?
5. A ***select*** function which evaluates the items in C
6. An ***objective function***

Example. How do you make change in South Africa with the minimum number of coins? (The coins are 1c, 2c, 5c, 10c, 20c, 50c.) Answer: Repeatedly add the largest coin that doesn't go over.

The set C of candidates is the infinite collection of coins $\{1, 2, 5, 10, 20, 50\}$. The feasibility check is that the next coin must not bring the total to more than that which is required. The select function is the value of the coin. The solution check is whether the selected coins reach the desired target.

However, a greedy algorithm does not work for every monetary system. Give an example!

In general, one can describe the greedy algorithm as follows:

```
Greedy(C:set)
    S := [ ]
    while not Solution(S) and C nonempty do {
        x := element of C that maximizes Select(x)
        C := C − [x]
        if Feasible(S +[x]) then S += [x]
        }
    if Solution(S) then return S
                else return "no solution"
```

There are greedy algorithms for many problems. Unfortunately most of those do not work! It is not simply a matter of devising the algorithm—one must prove that the algorithm does in fact work.

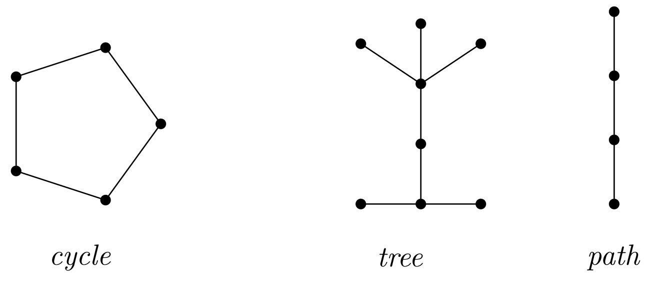
One useful concept for proving the correctness of greedy algorithms is the definition of a ***promising*** set. This is a set that can be extended to an optimal solution. It follows that:

LEMMA. If S is promising at every step of the Greedy procedure and the procedure returns a solution, then the solution is optimal.

1.2 Graphs and Minimum Spanning Trees

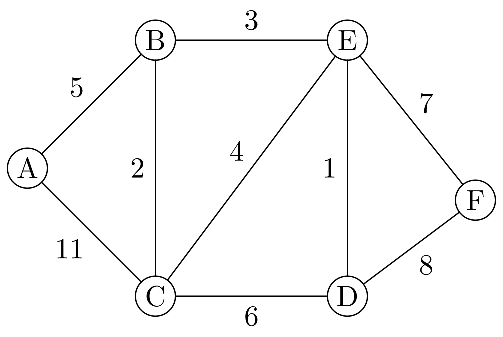
A ***graph*** is a collection of nodes some pairs of which are joined by edges. In a ***weighted graph*** each edge is labeled with a weight. In an unweighted graph each edge is assumed to have unit weight. The number of nodes is n and the number of edges is a.

A ***walk*** in a graph is a sequence of edges where the end of one edge is the start of the next. A ***cycle*** in a graph is a walk of distinct edges that takes you back to where you started without any repeated intermediate node. A graph without a cycle is called a ***forest***. A graph is ***connected*** if there is a walk between every pair of nodes. A ***tree*** is a connected forest.



A ***subgraph*** of a given graph is a graph which contains some of the edges and some of the nodes of the given graph. A subgraph is a ***spanning*** subgraph if it contains all the nodes of the original graph. A ***spanning tree*** is a spanning subgraph that is a tree. The ***weight*** of a subgraph is the sum of the weights of the edges. The ***minimum spanning***

tree is the spanning tree with the minimum weight. For the following picture, a spanning tree would have 5 edges; for example, the edges BC, AB, BE, EF and DF. But this is not optimal.



Trees have some useful properties:

LEMMA. (a) If a tree has n nodes then it has $n-1$ edges.
(b) If an edge is added to a tree, a unique cycle is created.

The proof is left as an exercise.

1.3 Primm's Minimum Spanning Tree

This is a greedy algorithm. One starts at an arbitrary node and maintains a tree throughout. At each step, we add one node to the tree.

Primm
– Candidates = edges
– Feasibility Test = no cycles
– Select Function = weight of edge if incident with current tree, else ∞.
(Note we minimize Select(x))

EXAMPLE. For the graph in the previous picture, suppose we started at node A. Then we would add edges to the tree in the order: AB, BC, BE, DE, EF.

We need to discuss (1) validity (2) running time.

◇ ***Validity***

A collection of edges is therefore ***promising*** if it can be completed to a minimum spanning tree. An edge is said to ***extend*** a set B of nodes if precisely one end of the edge is in B.

LEMMA. Let G be a weighted graph and let B be a subset of the nodes. Let P be a promising set of edges such that no edge in P extends B. Let e be any edge of largest weight that extends B. Then $P \cup \{e\}$ is promising.

PROOF. Let U be a minimum spanning tree that contains P. (U exists since P is promising.) If U contains e then we are done. So suppose that U does not contain e. Then adding e to U creates a cycle. There must be at least one other edge of this cycle that extends B. Call this edge f. Note that e has weight at most that of f (look at the definition of e), and that f is not in P (look at the definition of P).

Now let U' be the graph obtained from U by deleting f and adding e. The subgraph U' is a spanning tree. (Why?) And its weight is at most that of U. Since U was minimum, this means that U' is a minimum spanning tree. Since U' contains P and e, it follows that $P \cup \{e\}$ is promising. $\diamond$

If we let B be the set of nodes inside our tree, then it is a direct consequence of the lemma that at each stage of Primm's algorithm the tree constructed so far is promising. When the tree reaches full size, it must therefore be optimal.

$\diamond$ ***Running time***

By a bit of thought, this algorithm can be implemented in time $O(n^2)$. One keeps track of

- *for each node outside B the smallest weight edge from B to it*

This is stored in an array called minDist. When a node is added to B, one updates this array.

The code is as follows, assuming one starts at node 1:

```
Primm(G:graph)
        Tree ← []
        B ← [1]
        for i=2 to $n$ do {
            nearest[i] ← 1
            minDist[i] ← weight[1,i]
            }
    while B not full do {
        min ← $\infty$
        for all j not in B do
            if minDist[j] < min then {
                min ← minDist[j]
                newBie ← j
                }
        Tree ← Tree + Edge(newBie,nearest[newBie])
        B += newBie
        for all j not in B do
            if weight[newBie,j] < minDist[j] then {
                minDist[j] ← weight[newBie,j]
                nearest[j] ← newBie
                }
    }
    return Tree
```

1.4 Kruskal's Minimum Spanning Tree

A greedy algorithm. This time a forest is maintained throughout.

Kruskal

– Candidates = edges
– Feasibility Test = no cycles
– Select Function = weight of edge
(Note we minimize Select(x))

The validity is left as an exercise. Use the above lemma.

Example. For the graph on page 4, the algorithm would proceed as follows. DE, BC, BE, CE not taken, AB, CD not taken, EF.

◇ ***Running time***

One obvious idea is to pre-sort the edges.

Each time we consider an edge, we have to check whether the edge is feasible or not. That is, would adding it create a cycle. It seems reasonable to keep track of the different components of the forest. Then, each time we consider an edge, we check whether the two ends are in the same component or not. If they are we discard the edge. If the two ends are in separate components, then we merge the two components.

We need a data structure. Simplest idea: number each node and use an array Comp where, for each node, the entry in Comp gives the smallest number of a node in the same component. Testing whether the two ends of an edge are in different components involves comparing two array entries: $O(1)$ time. Total time spent querying: $O(a)$. This is insignificant compared to the $O(a \log a)$ needed for sorting the edges.

However, merging two components takes $O(n)$ time (in the worst case). Fortunately, we only have to do a merge $n-1$ times. So total time is $O(n^2)$.

The running time of Kruskal's algorithm, as presented, is

$$\max\{O(n^2), O(a \log a)\}$$

which is no improvement over Primm. But actually, we can use a better data structure and bring it down.

1.5 Disjoint Set Structure

So, in implementing Kruskal's algorithm we need to keep track of the components of our growing forest. This requires a data structure which represents an arbitrary collection of disjoint sets. It should allow two operations:
- FIND tells one what set a value is in,
- MERGE combines two sets.

Above, we had an array implementation which took time $O(1)$ for FIND but $O(n)$ for MERGE.

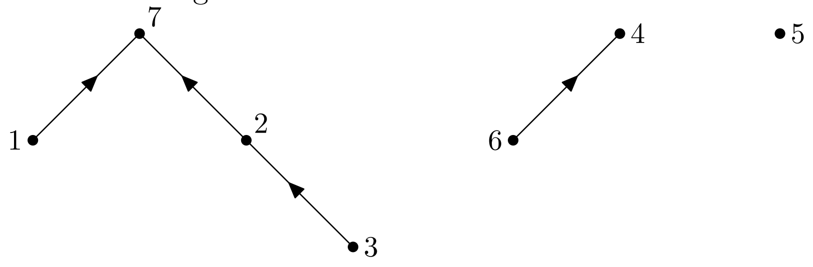
Try the following. Again store values in an array A. But this time each set is stored as a rooted sub-tree according to the following scheme:

> *If $A[i] = i$, then i is the label of the set and the root of some sub-tree.*
> *If $A[i] \neq i$, then $A[i]$ is the parent of i in some sub-tree.*

For example, if the components are $\{1, 2, 3, 7\}$, $\{4, 6\}$, $\{5\}$, then the array might be

	1	2	3	4	5	6	7
A	7	7	2	4	5	4	7

which represents the following sub-trees.



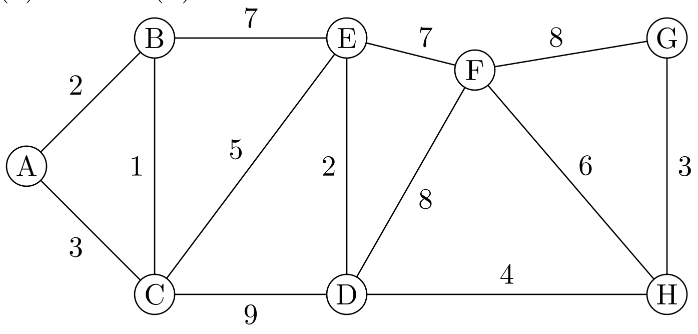
To determine the label of the set of a node, one follows the pointers up to the root. To combine two sets, one changes the root of one to point to the root of the other.

These operations take time proportional to the depth of the sub-tree. And so we haven't made any progress, yet. But there is a simple fix: When merging, make the root of the shorter one point to the root of the taller one. Recall that the ***depth*** of a rooted tree is the maximum number of edges from the root to another node. It can be shown that the depth after a series of k merges is at most $\log k$. (See exercise.) This means that both operations run in time $O(\log n)$. Note that one can keep track of the depth of a sub-tree. Further improvements are possible.

Applied to Kruskal, this gives an $O(a \log a)$ algorithm for finding a minimum spanning tree, since the sorting of the edges is now the most time-consuming part.

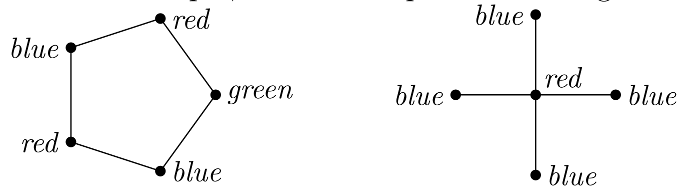
Exercises

1. Prove that the greedy algorithm works for U.S. coinage. Concoct an example monetary system where it doesn't work.
2. In what order are the edges of a minimum spanning tree chosen in the following graph, using (a) Primm (b) Kruskal?



3. Write an essay on spanning tree algorithms, explaining why they work and how to implement them. Include at least one algorithm not described here.

4. ⓔ In a ***coloring*** of a graph, one assigns colors to the nodes such that any two nodes connected by an edge receive different colors. An optimal coloring is one which uses the fewest colors. For example, here is an optimal coloring of these two graphs.



Dr I.B. Greedy proposes the following algorithm for finding an optimal coloring in a graph where the nodes are numbered 1 up to n: Use as colors the positive integers and color the nodes in increasing order, each time choosing the smallest unused color of the neighbors colored so far.

(a) How long does this algorithm take? Justify.

(b) Does the algorithm work? Justify.

5. Suppose in Kruskal's algorithm we use the rooted-tree disjoint-set data structure for keeping track of components. If the nodes are A,B,C,D,E,F,G,H and the edges that Kruskal adds are in order AB, AC, DE, EF, AG, DH, GH, what does the final data structure look like?

6. In the rooted-tree disjoint-set data structure, show that a tree of depth d has at least 2^d nodes.

Chapter 2: Dynamic Programming

Often there is no way to divide a problem into a ***small*** number of subproblems whose solution can be combined to solve the original problem. In such cases we may attempt to divide the problem into many subproblems, then divide each subproblem into smaller subproblems and so on. If this is all we do, we will most likely end up with an exponential-time algorithm.

Often, however, there is actually a limited number of possible subproblems, and we end up solving a particular subproblem several times. If instead we keep track of the solution to each subproblem that is solved, and simply look up the answer when needed, we would obtain a much faster algorithm.

In practice it is often simplest to create a ***table*** of the solutions to all possible subproblems that may arise. We fill the table not paying too much attention to whether or not a particular subproblem is actually needed in the overall solution. Rather, we fill the table in a particular order.

2.1 Longest Increasing Subsequence

Consider the problem of: Given a sequence of n values, find the ***longest increasing subsequence***. By subsequence we mean that the values must occur in the order of the sequence, but they ***need not be consecutive***.

For example, consider 3,4,1,8,6,5. In this case, 4,8,5 is a subsequence, but 1,3,4 is not. The subsequence 4,8 is increasing, the subsequence 3,4,1 is not. Out of all the subsequences, we want to find the longest one that is increasing. In this case this is 3,4,5 or 3,4,8 or 3,4,6.

Now we want an efficient algorithm for this. One idea is to try all subsequences. But there are too many. Divide-and-conquer does not appear to work either.

Instead we need an idea. And the idea is this: if we know the longest increasing subsequence that ends at 3, and the longest one that ends at 4, and ... the longest one that ends at 6, then we can determine the longest that ends at 5.

How? Well, any increasing subsequence that ends at 5 has a penultimate element that is smaller than 5. And to get the longest increasing subsequence that ends at 5 with, for example, penultimate 4, one takes the longest increasing subsequence that ends with 4 and appends 5.

Books on dynamic programming talk about a ***principle of optimal substructure***: paraphrased, this is that

- *"a portion of the optimal solution is itself an optimal solution to a portion of the problem".*

For example, in our case we are interested in the longest increasing sequence; call it $\mathcal{L}$. If the last element of $\mathcal{L}$ is $A[m]$, then the rest of $\mathcal{L}$ is the longest increasing subsequence of the sequence consisting only of those elements lying before $A[m]$ and smaller than it.

Suppose the input is array A. Let $f(m)$ denote the longest increasing subsequence that ends at $A[m]$. In our example:

A	3	4	1	8	6	5
f	1	2	1	3	3	3

In general, to compute $f(m)$: go through all the $i < m$, look at each i such that $A[i] < A[m]$, determine the maximum $f(i)$, and then add 1. In other words:

$$f(m) = 1 + \max_{i<m}\{\, f(i) : A[i] < A[m] \,\}$$

Efficiency. The calculation of a particular $f(m)$ takes $O(m)$ steps. The algorithm calculates $f(1)$, then $f(2)$, then $f(3)$ etc. Thus, the total work is quadratic—it's on the order of $1 + 2 + \cdots + n$.

Dynamic programming has a similar flavor to divide-and-conquer. But dynamic programming is a bottom-up approach: smaller problems are solved and then combined to solve the original problem. The efficiency comes from storing the intermediate results so that they do not have to be recomputed each time.

2.2 Largest Common Subsequence

The largest increasing subsequence problem is a special case of the ***largest common subsequence problem***. In this problem, one is given two strings or arrays and must find the longest subsequence that appears in both of them.

(Explain why the longest increasing subsequence problem is a special case of the longest common subsequence problem.)

The approach is similar to above. One does organized iteration. Suppose the input is two arrays A and B. Then define

$g(m, n)$ to be the longest common subsequence that ends with $A[m]$ and $B[n]$.

Obviously this is 0 unless $A[m] = B[n]$.

To compute $g(m, n)$ from previous information, we again look at the penultimate value in the optimal subsequence. Say the penultimate is in position i in A and in position j

in B (with of course $A[i] = B[j]$). Then the portion up to the penultimate is the longest common subsequence that ends with $A[i]$ and $B[j]$.

So we obtain the recursive formula:

$$g(m,n) = \begin{cases} 1 + \max\limits_{i<m,\, j<n} g(i,j) & \text{if } A[m] = B[n], \\ 0 & \text{otherwise} \end{cases}$$

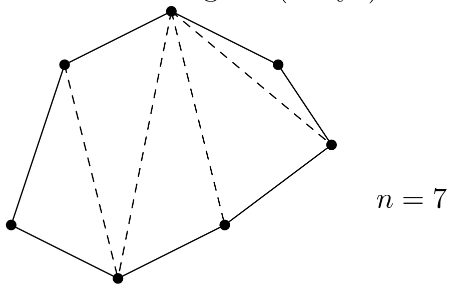
A simple implementation yields an $O(n^3)$ algorithm. Note that rather than doing recursion, one works one's way systematically through the table of $g(m,n)$ starting with $g(1,1)$, then $g(2,1)$, then $g(3,1)$, and so on.

2.3 The Triangulation Problem

The following is based on the presentation of Cormen, Leiserson and Rivest.

A ***polygon*** is a closed figure drawn in the plane which consists of a series of line segments, where two consecutive line segments join at a vertex. It is called a ***convex polygon*** if no line segment joining a pair of nonconsecutive vertices intersects the polygon.

To form a ***triangulation*** of the polygon, one adds line segments inside the polygon such that each interior region is a triangle. If the polygon has n vertices, $n-3$ line segments will be added and there will be $n-2$ triangles. (Why?)

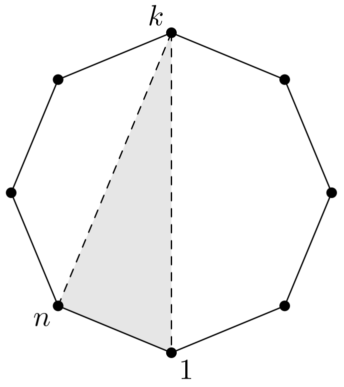


Now, suppose that associated with any possible triangle there is a ***weight*** function. For example, one might care about the perimeter of the triangle. Then the ***weight*** of a triangulation is the sum of the weights of the triangles. An ***optimal triangulation*** is one of minimum weight.

(This problem comes up in graphics. In particular it is useful to find the optimal triangulation where the weight function is the perimeter of the triangle.)

There is a recursive nature to the optimal triangulation. But one still has to be careful to ensure that the number of subproblems does not become exponential.

Assume that the vertices of the polygon are labeled v_1, v_2, ..., v_n where v_n is adjacent to v_1. Now, the side v_1v_n must be the side of some triangle: say v_k is the other vertex of the triangle in the optimal triangulation. This triangle splits the polygon into two smaller polygons: one with vertices $\{v_1, v_2, \ldots, v_k\}$ and one with vertices $\{v_k, v_{k+1}, \ldots, v_n\}$. Furthermore, the optimal triangulation of the original problem includes optimal triangulations of these two smaller polygons.



Now, if we apply the recursion to the smaller polygons using the nonoriginal segment as the base of the triangle, we get two smaller problems, and the boundaries of these polygons again only have one nonoriginal segment.

Let us define $t[i, j]$ for $1 \leq i < j \leq n$ as the weight of an optimal triangulation of the polygon with vertices $\{v_i, v_{i+1}, \ldots, v_j\}$. If $i = j - 1$ then the polygon is degenerate (has only two vertices) and the optimal weight is define to be 0.

When $i < j - 1$, we have a polygon with at least three vertices. We need to minimize over all vertices v_k, for $k \in \{i+1, i+2, \ldots, j-1\}$, the weight of the triangle $v_i v_k v_j$ added to the weights of the optimal triangulations of the two smaller polygons with vertices $\{v_i, v_{i+1}, \ldots, v_k\}$ and $\{v_k, v_{k+1}, \ldots, v_j\}$.

Thus we obtain the formula:

$$t[i, j] = \begin{cases} 0 & \text{if } i = j - 1 \\ \min\limits_{i<k<j} t[i, k] + t[k, j] + w(\triangle v_i v_k v_j) & \text{if } i < j - 1 \end{cases}$$

The calculation of the time needed is left to the reader.

Exercises

1. Illustrate the behavior of:

   a) The longest increasing subsequence algorithm on the list:
   2, 11, 3, 10, 8, 6, 7, 9, 1, 4, 5

   b) The longest common subsequence algorithm on the two lists:
   1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 and 2, 11, 3, 10, 8, 6, 7, 9, 1, 4, 5

2. Code up the longest common subsequence algorithm. Your algorithm should return one of the longest common subsequences, not just the length.

3. John Doe wants an algorithm to find a triangulation where the sum of the lengths of the additional line segments is as small as possible. Can you help?

   Jane Doe wants an algorithm to find an optimal triangulation where the weight function is the area of the triangle. Can you help?

4. ⓔ *(From Cormen et al.)* Consider the problem of neatly printing a paragraph on a printer. The input text is a sequence of n words of lengths $l_1, l_2, \ldots, l_n$, measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. Our criterion of "neatness" is as follows. If a given line contains words i through j and we leave exactly one space between words, the number of extra characters at the end of the line is $M - j + i - \sum_{k=i}^{j} l_k$. The ***penalty*** for that line is the cube of the number of extra spaces. We wish to minimize the sum, over all lines except the last, of the penalties. Give a dynamic-programming algorithm to print a paragraph of n words neatly on a printer. Analyze the running time and storage requirements of your algorithm.

Chapter 3: Shortest Paths

The following question comes up often. What is the quickest way to get from A to B? This is known as the shortest-path problem. The underlying structure is a graph. The graph need not be explicitly precalculated. It could be the state graph of a finite automaton, the search graph of an AI problem, or the position graph of a game.

3.1 Dijkstra's Algorithm

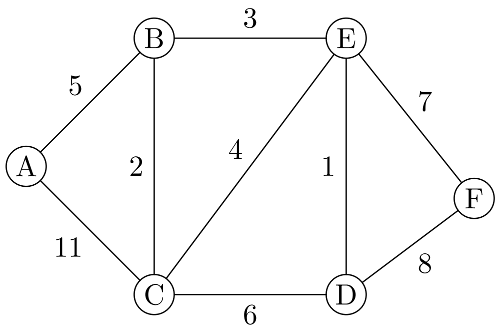
If we are just interested in finding the shortest path from one node to another node in a graph, then the famous algorithm is due to Dijkstra. It essentially finds a breadth-first search tree.

We grow the tree one node at a time. We define the auxiliary function $\mathit{currDis}(v)$ for a node v as the length of the shortest path to v subject to the restriction that the penultimate node is in the current tree. At each stage we add to the current tree that node which has the smallest value of $\mathit{currDis}(v)$. We then update the value of $\mathit{currDis}(v)$ for the remaining nodes.

The following produces the distance from node a to all other nodes.

```
ShortestPath (G:graph, a:node)
        for all nodes v do currDis(v) ← infinity
        currDis(a) ← 0
        remainder ← [ all nodes ]
    while remainder nonempty do {
        let w be node with minimum value of currDis
        remainder −= [w]
        for all nodes v in remainder do
            currDis (v) ← min ( currDis(v), currDis(w)+length(w,v) )
    }
```

Example. For the following graph, the table gives the value of currDis at each stage.



A	B	C	D	E	F
⓪	∞	∞	∞	∞	∞
	⑤	11	∞	∞	∞
		⑦	∞	8	∞
			13	⑧	∞
			⑨		15
					⑮

Complexity: quadratic. We go through the while loop about n times and the update loop takes $O(n)$ work.

3.2 The All Pairs Shortest Path Problem

Suppose we wanted instead to calculate the shortest path between every pair of nodes. One idea would be to run Dijkstra with every node as source node.

Another algorithm is the following dynamic programming algorithm known as Floyd–Warshall. Suppose the nodes are ordered 1 up to n. Then we define

> $d_m(u, v)$ *as the length of the shortest path between* u *and* v *that uses only the nodes numbered* 1 *up to* m *as intermediates.*

The answer we want is $d_n(u, v)$ for all u and v. (Why?)

There is a formula for d_m in terms of d_{m-1}. Consider the shortest u to v path that uses only nodes labeled up to m—call it P. There are two possibilities. Either the path P uses node m or it doesn't. If it doesn't, then P is the shortest u to v path that uses only nodes up to $m-1$. If it does use m, then the segment of P from u to m is the shortest path from u to m using only nodes up to $m-1$, and the segment of P from m to v is the shortest path from m to v using only nodes up to $m-1$.

Hence we obtain:

$$d_m(u, v) = \min \begin{cases} d_{m-1}(u, v) \\ d_{m-1}(u, m) + d_{m-1}(m, v) \end{cases}$$

The resultant program iterates m from $m = 0$ to $m = n - 1$. Each time there are $O(n^2)$ values of $d_m(u, v)$ to be calculated, and each calculation takes $O(1)$ time. Hence, we have an $O(n^3)$ algorithm. (Same as Dijkstra but runs faster.) Note the storage requirements.

Exercises

1. Implement Floyd–Warshall using your favorite programming language.

The program should take the input graph in the form of a text file provided by the user. The first line is the number of nodes. Each remaining line is an edge: three integers in order provide the number of each node and then the weight. The end of the input is signified by the triple `0 0 0`.

2. Suggest ways in which the efficiency of Floyd–Warshall might be improved.

3. Illustrate the steps of Dijkstra, using node A as source, on the graph in Exercise 2 of Chapter 1.

4. ⓔ Sometimes graphs have edges with negative weights. Does the concept of distance still make sense? Do the above algorithms still work? Where might one find such graphs? Discuss carefully.

Chapter 4: Maximum Matchings and Trees

We consider the maximum matching problem. We start with the case of a tree. There are some nice algorithms on trees: both greedy algorithms and recursion can be successful.

4.1 Traversal of Rooted Trees

A ***rooted tree*** is a tree with one node designated the root. For a rooted tree, we can talk of ***parents***, ***children*** and ***descendants***. The ***subtree rooted*** at node p consists of the node p and all its descendants.

For a binary tree, we can visit the nodes of the tree in a specific order.

- In a ***preorder*** traversal, a node is visited before either of its children
- in a ***inorder*** traversal, a node is visited after visiting its left child and before its right child
- in an ***postorder*** traversal, a node is visited after both its children.

Assume the tree is stored with pointers from parent to children. Then a preorder, inorder or postorder traversal can be performed in linear time with the obvious recursive algorithm. For example:

```
Inorder(root)
    if root=nil exit
    Inorder( LeftChild(root) )
    visit(root)
    Inorder( RightChild(root) )
```

How does one justify the claim that takes linear time? (For you.) There is a natural generalization to nonbinary rooted trees of preorder and postorder traversals, and the recursive algorithm again takes linear time.

4.2 Matchings

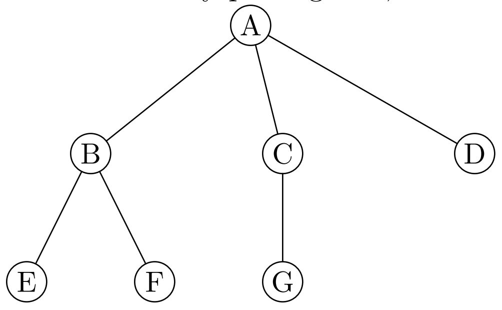
A ***matching*** in a graph is a collection of edges such that no pair of edges share a common node. A matching with the most edges is known as a ***maximum matching***.

This question comes up in scheduling problems. Suppose we have a school and we list all the classes occurring at a particular time as one set of nodes, and all the venues available at that time as another set of nodes, and join the two nodes if the venue is suitable for the class. The result is a graph. If there is a matching which uses all the classes, then the schedule for that time is possible.

An algorithm for maximum matching in trees is the following. A ***leaf-edge*** is an edge whose one end has no other neighbor. The (greedy) algorithm is to repeatedly take any leaf-edge.

```
TreeMatch(F:forest)
    M ← [ ]
    while F nonempty do {
        select any leaf-edge e
        M ← M + [e]
        F ← F − both ends of e
        }
```

A maximum matching can be obtained by pairing BE, then CG, then AD.

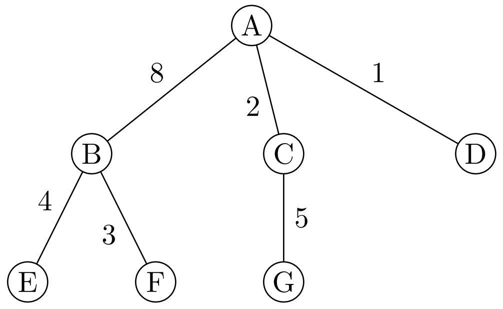


Why works? Well, assume e is a leaf edge and consider any maximum matching N. Suppose N does not contain e. Then if we add e to N, only one node now has two edges incident with it. So we can delete one of the edges of N and attain a maximum matching containing e. And so on. Hence at each stage M is promising.

One can implement this algorithm using a postorder traversal: every time one visits a node, if there is a child available then one pairs the node with one of its children.

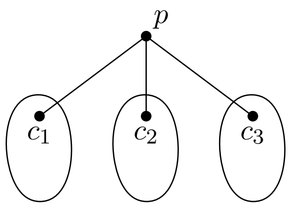
4.3 Weighted Matching in Trees

If the edges have weights, then the weight of the matching is the sum of the weights. The greedy algorithm does not work here. Consider for example the following tree: if we use the greedy algorithm we would start with the edge BE, but this is wrong.



There is a recursive algorithm, though it is a bit mysterious. We will work from the leaves up to the root. For a node p we define the subtree rooted at p, denoted T_p, as the tree consisting of p and all its descendants. At each level we calculate for the subtree T_p rooted at p two quantities:

- the maximum weight matching of T_p, denoted $m(T_p)$;
- the maximum weight matching over all matchings such that the root is not in the matching, denoted $mnr(T_p)$.



The maximum weight matching in the subtree T_p is easy to calculate if the root is specifically excluded. For, one takes the best matching in each of the subtrees rooted at each of the children and simply combines them:

$$mnr(T_p) = \sum_{\text{children } c} m(T_c)$$

If the root is allowed to be in the matching, then apart from the above situation we also have the possibility that there is an edge in the optimal matching connecting p to one of its children. Say that child is c. Then in the subtree rooted at c we must take an optimal matching which does not include c, while in the other subtrees we can take any matching. This gives the following formula:

$$m(T_p) = \max \begin{cases} mnr(T_p) \\ \max\limits_{\text{children } c} w(p-c) + mnr(T_c) + \sum\limits_{d \neq c} m(T_d) \end{cases}$$

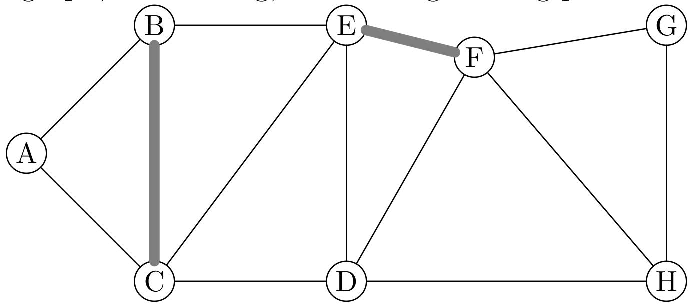
So our algorithm is: visit the nodes with a postorder traversal. At each node, the values for children are already calculated, and so we can calculate the values for the parent by the above recursive formulas. The result is a linear-time algorithm. (That this is true for non-binary trees requires some work to show.)

4.4 Maximum Matching in General Graphs

Let us return to the maximum matching problem without weights but now in a general graph.

Let M be a matching. A path is called ***alternating*** (with respect to M) if the first edge of the path is not in M and thereafter the edges alternate being in M and out of M. A path is called ***augmenting*** if it is alternating and neither end-node is incident with an edge of M.

Example. Here's a graph, a matching, and an augmenting path is $ACBEFG$.



We also need the concept of the ***symmetric difference*** of two sets:

$$E \triangle F = (E - F) \cup (F - E).$$

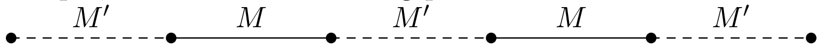
That is, the symmetric difference consists of those edges in precisely one of the two sets.

Before reading further, explain to yourself what the symmetric difference of two matchings can look like. The following result is due to Berge.

Lemma. A matching is maximum if and only if there exists no augmenting path.

Proof. We show: (1) if there is an augmenting path, then we can increase the matching; and (2) if the matching is not largest, then there is an augmenting path.

(1) Suppose M is a matching and there is an augmenting path P. Then let $M' = M \triangle P$. In other words, delete from M those edges lying in P and add back in those edges in P which didn't lie in M. The result is a matching. The matching M' has one more edge than M. For example, consider the following picture:



(2) Suppose M is a matching and there is a larger matching N. Consider the graph $H = M \triangle N$. Each node of H is incident with at most two edges, and if it is incident with two edges then one edge is from M and one is from M. This means that H consists of cycles and paths. Furthermore, a cycle in H must have an even number of edges: they alternate between M and N.

Now, since there are more edges in N than in M, in H there must be more edges of N than of M. In an even cycle the edges of M and N alternate so they are equinumerous.

So there must be a path P in H in which N is in the majority. But think about the path P: it must be augmenting with respect to M! ◇

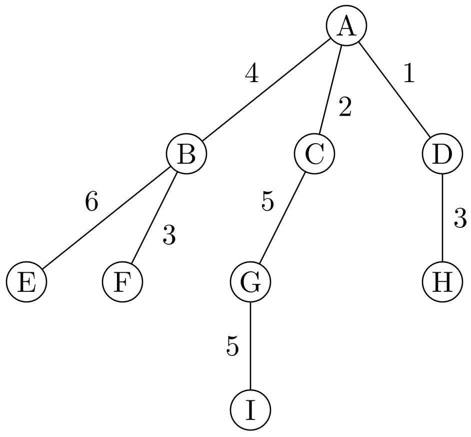
This lemma gives an iterative algorithm for finding a maximum matching in a graph.

```
**MaximumMatching** (G:graph)
    M ← [ ]
    repeat
        find augmenting path P
        if found then M ← M △ P
    until path not found
```

There's only the "small" problem of finding an augmenting path! We leave that to a later course.

Exercises

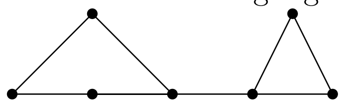
1. Illustrate the workings of the maximum matching algorithm on the following weighted tree.



2. A graph is ***bipartite*** if the nodes can be colored with two colors such that adjacent nodes receive different colors. Show that if a graph is bipartite then it cannot contain a cycle with an odd number of node. (The converse is also true.)

3. A tree is always bipartite. Construct a linear-time algorithm for finding a 2-coloring of a tree.

4. Illustrate the steps of the maximum matching algorithm on the following graph.



5. A set of nodes is said to be ***dominating*** if every node in the graph is either in the set or adjacent to a member of the set. The ***domination number*** of a graph is the minimum size of a dominating set. What is the domination number of a path on n nodes?

6. Describe a linear-time algorithm for finding the domination number of a binary tree.

Chapter 5: Revision Problems

1. A thief robbing a jeweler finds only some containers of metallic powders. Each powder has a value per gram. The thief wants to take away as valuable load as possible. Unfortunately her knapsack can hold only 20 kg. Devise a algorithm to determine the best load, quickly.

2. The Coinage system of Octavia has only coins which are a power of 8, that is, 1, 8, 64, 512 etc. Devise a simple algorithm for determining the minimum number of coins to make up a given value.

3. Given a graph, a ***happy*** coloring is one which colors the nodes with two colors red and blue such that at least half of the edges have different colored ends. Find an efficient algorithm for finding a happy coloring.

4. You are given a sorted array of n integers (which can be positive or negative) $x_1, x_2, \ldots, x_n$. Give an algorithm which finds an index i such that $x_i = i$ provided such an index exists. Your algorithm should take time $O(\log n)$ in the worst case.

5. You are given a list of n integers and must find the m smallest elements, where m is much smaller than n. Would you:
   (a) sort the list
   (b) use RankSelect m times
   (c) use some other method?

   Justify your answer.

6. When one wants to compute the product of n matrices, since matrix multiplication is associative we can put brackets wherever we like. For example, to compute $ABCD$, there are five possibilities: $(AB)(CD)$, $((AB)C)D$, $(A(BC))D$, $A((BC)D)$, $A(B(CD))$. Give an algorithm to determine how many possibilities there are for n matrices.

7. Given a collection of n line segments on the straight line $[x_i, y_i]$, devise a fast algorithm to count how many pairs of line segments intersect.

8. The obvious way to compute the maximum and minimum elements in an n-element set takes $2n - 3$ comparisons. Give an algorithm which takes fewer comparisons.

9. Design a fast algorithm that takes a sequence of numbers and an integer k and counts the number of increasing subsequences of length k.

Chapter 6: Books

I have made much use of:

• *Data Structures and Algorithms*,
A. Aho, J. Hopcroft and J. Ullman

• *Fundamentals of Algorithmics*,
G. Brassard and P. Bratley

• *Introduction to Algorithms*,
T. Cormen, C. Leiserson and R. Rivest

• *Introduction to Parellel Algorithms and Architectures: Arrays, Trees, Hypercubes*,
F.T. Leighton

• *Data Structures and Algorithm Analysis,*
M.A. Weiss

Introduction to Algorithms

Part 3: P, NP
Hard Problems

Chapter 1: Polynomial Time: P and NP

The field of ***complexity theory*** deals with how fast can one solve a certain type of problem. Or more generally how much resources does it take: time, memory-space, number of processors etc. We assume that the problem is solvable.

1.1 Computations, Decisions and Languages

The most common resource is time: number of steps. This is generally computed in terms of n, the length of the input string. We will use an informal model of a computer and an algorithm. All the definitions can be made precise by using a model of a computer such as a Turing machine.

While we are interested in the difficulty of a computation, we will focus our hardness results on the difficulty of yes–no questions. These results immediately generalize to questions about general computations. It is also possible to state definitions in terms of ***languages***, where a language is defined as a set of strings: the language associated with a question is the set of all strings representing questions for which the answer is Yes.

1.2 The Class P

The collection of all problems that can be solved in polynomial time is called P. That is, a decision question is in P if there exists an exponent k and an algorithm for the question that runs in time $O(n^k)$ where n is the length of the input.

P roughly captures the class of practically solvable problems. Or at least that is the conventional wisdom. Something that runs in time 2^n requires double the time if one adds one character to the input. Something that runs in polynomial time does not suffer from this problem.

P is robust in the sense that any two reasonable (deterministic) models of computation give rise to the same definition of P.

◇ ***True Boolean Formulas***

A ***boolean formula*** consists of variables and negated variables (known collectively as ***literals***), and the operations "and" and "or". We use $\vee$ for "or", $\wedge$ for "and", and $\bar{x}$ for "not x". For example

$$x \wedge (x \vee y \vee \bar{z}) \wedge (\bar{x} \vee \bar{y})$$

An ***assignment*** is a setting of each of the variables to either TRUE or FALSE. For example if x and y are TRUE and z is FALSE, then the above boolean formula is FALSE.

> TRUEBF
> Input: ϕ a boolean formula, assignment ψ
> Question: Does ψ make ϕ TRUE?

Given a boolean function and an assignment of the variables, we can determine whether the formula is TRUE or not in linear time. (A formula can be evaluated like a normal arithmetic expression using a single stack.) So it follows that the TRUEBF problem is in P.

◇ ***Paths***

Recall that a ***path*** is a sequence of edges leading from one node to another.

> PATH
> Input: graph G, nodes a and b
> Question: Is there a path from a to b in G?

This problem is in P. To see if there is a path from node a to node b, one might determine all the nodes reachable from a by doing for instance a breadth-first search or Dijkstra's algorithm. Note that the actual running time depends on the representation of the graph.

1.3 Nondeterminism and NP

What happens if we allow our machines to receive advice? We define a nondeterministic algorithm as one which receives two input: the regular input and a ***hint***. The input is considered to be a Yes-instance iff there is some hint that causes the algorithm to say yes. So, the input is a No-instance only if every hint causes the algorithm to say no. A hint that causes the algorithm to say yes is called a ***certificate***.

The collection of all problems that can be solved in polynomial time using nondeterminism is called NP. That is, a decision question is in NP if there exists an exponent k and an nondeterministic algorithm for the question that for all hints runs in time $O(n^k)$ where n is the length of the input.

◇ ***Satisfiability***

In dealing with boolean formulas, a ***clause*** is the "or" of a collection of literals. A formula is said to be in ***conjunctive form*** if it is the "and" of several clauses.

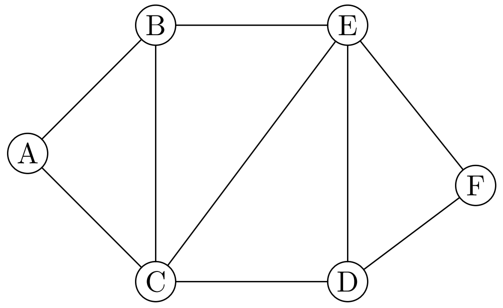
> SAT
> Input: ϕ a boolean formula in conjunctive form
> Question: Is there an assignment that makes ϕ TRUE?

The SAT problem is in NP. The certificate is the assignment of the variables. However, the number of assignments in exponential, so there does not exist an obvious polynomial-time algorithm.

◇ *Hamiltonian path*

A ***hamiltonian path*** from node a to node b is a path that visits every node in the graph exactly once.

Example. In the graph drawn, there is a path from every node to every other node. There is a hamiltonian path from A to F but not one from C to E.



> `HAMPATH`
> Input: graph G, nodes a and b
> Question: Is there a hamiltonian path from a to b in G?

It is far from obvious how to decide `HAMPATH` in polynomial time. (Note that trying all possible paths does not work since there are exponentially many.) But there is a fast nondeterministic program for it: the certificate is the actual path.

The time taken by the nondeterministic algorithm is at most quadratic in the number of nodes of the graph. And so `HAMPATH` is in NP. On the other hand, its complement does not appear to be so. No simple general certificate of the nonexistence of a hamiltonian path springs to mind.

◇ *What is Nondeterminism?*

Thus NP can be viewed as those problems that are ***checkable*** in polynomial time: the certificate provides a proof of a Yes-answer. So we have this Prover/Verifier situation. This problem can be decided by a Verifier in polynomial time if he is given a hint from an omniscient Prover. Note that the correct answer is made: if there is a hamiltonian path then the correct hint will be verified and the Verifier will say yes; and if there is no hamiltonian path then there is no way the Verifier can be conned into saying yes.

Nondeterminism is more usually defined as allowing a program more than one possible next move and the program accepts the input iff there is some possible sequence of computations that lead to an accept state. This notion is equivalent to the one defined as above: the certificate is a record of the correct next moves.

1.4 P versus NP

It would seem that P and NP might be different sets. In fact, probably the most important unsolved problems in Mathematics and Computer Science today is:

Conjecture. P $\neq$ NP

If the conjecture is true, then many problems for which we would like efficient algorithms do not have them. Which would be sad. If the conjecture is false, then much of cryptography is under threat. Which would be sad.

Exercises

1. Define and compare the concepts of: polynomial, logarithmic, linear, quadratic, exponential, superexponential, superlinear.

2. Consider a list of implications involving literals: an ***implication*** $a \to b$ means that if a is TRUE then b is TRUE. Give a polynomial-time algorithm to determine if a set of implications are satisfiable.

3. One is given a boolean function GoodTest which takes as input a string (and runs in precisely 1 second irrespective of the length of the string). You must devise a polynomial-time algorithm for the following problem:

    > `GoodDivision`
    > Input: A string of length n
    > Question: Can the string be divided into pieces such that each piece is good (returns true from GoodTest)?

Chapter 2: NP-Completeness

While we cannot determine whether P = NP or not, we can, however, identify problems that are the hardest in NP. These are called the NP-complete problems. They have the property that if there is a polynomial-time algorithm for any one of them then there is a polynomial-time algorithm for every problem in NP.

2.1 Reductions

For decision problems A and B, A is said to be ***polynomial-time reducible*** to B (written $A \leq_p B$) if there is a polynomial-time computable function f such that

$$q \text{ is a Yes-instance of } A \iff f(q) \text{ is a Yes-instance of } B$$

That is, f translates questions about A into questions about B while preserving the answer to the question.

The key result:

LEMMA. a) If $A \leq_p B$ and B in P, then A in P.
b) If $A \leq_p B$ and A not in P, then B not in P.

PROOF. Suppose the reduction from A to B is given by the function f which is computable in $O(n^k)$ time. And suppose we can decide questions about B in $O(n^\ell)$ time. Then we build a polynomial-time decider for A as follows. It takes the input q, computes $f(q)$ and then sees whether $f(q)$ is a Yes-instance of B or not. Does the program run in polynomial-time? Yes. If q has length n then the length of $f(q)$ is at most $O(n^k)$ (since a program can only write one symbol each step). Then the test about B takes $O(n^{k\ell})$ time. And that's polynomial. $\diamond$

2.2 NP-completeness

We need a definition:

A decision problem S is defined to be NP*-complete if*
a) It is in NP*; and*
b) For all A in NP *it holds that $A \leq_P S$.*

Note that this means that:

- If S in NP-complete and S in P, then P=NP.
- If S is NP-complete and T in NP and $S \leq_p T$, then T is NP-complete.

2.3 Examples

There are tomes of NP-complete problems. The standard method to proving NP-completeness is to take a problem that is known to be NP-complete and reduce it to your problem. What started the whole process going was Cook's original result:

THEOREM. `SAT` is NP-complete.

We omit the proof. Some more examples:

- The `3SAT` problem is NP-complete:

  > `3SAT`
  > Input: ϕ a boolean formula in conjunctive form with three literals per clause
  > Question: Is there a satisfying assignment?

- `HAMPATH` is NP-complete.
- Domination. A set of nodes in a graph is a ***dominating set*** if every other node is adjacent to at least one of these nodes. For example, in the graph on page 4, $\{A, D\}$ is a dominating set.

  The `DOMINATION` problem is NP-complete:

  > `DOMINATION`
  > Input: Graph G and integer k
  > Question: Does there exist a dominating set of G of at most k nodes?

2.4 Primes, Composites and Factorization

An old problem is to find a fast algorithm which determines whether a number is prime or not, and if composite, determines a factorization. Now, it is important to realize that a number m is input into a program in binary. Thus the length of the input is $\log_2 m$. That is, we want algorithms that run in time polynomial in the number of bits, not in time proportional to the number itself.

So define `PRIME` as determining whether a binary input is prime and `COMPOSITE` as determining whether it is composite. It is clear that `COMPOSITE` is in NP: simply guess a split into two factors and then verify it by multiplying the two factors. It is not immediately clear what a certificate for primeness looks like, but elementary number theory provides an answer, and so it has been known for many years that `PRIME` is in NP.

The question of compositeness was conjectured by some to be NP-complete. However, in 2002 it was shown that `PRIME` is in P, and hence so is `COMPOSITE`. Nevertheless, there is still no polynomial-time algorithm for determining the factorization of a composite number. Which is a good thing in some respects, since RSA public-key cryptography assumes that factorization is hard.

2.5 Proving NP-completeness by Reduction

◇ *3SAT*

We show `3SAT` is NP-complete. Since `3SAT` is a "restriction" of `SAT`, it must be in NP since `SAT` is.

We now show how to reduce `SAT` to `3SAT`. Our task is to describe a polynomial-time algorithm which takes as input a boolean formula ϕ in conjunctive form, and produces a boolean formula ψ in conjunctive form with 3 literals per clause such that ϕ is satisfiable iff ψ is.

The idea is to take each clause C of ϕ and replace it by a family D of clauses. For example, suppose there was a clause $C = a \vee b \vee c \vee d \vee e$. What does this mean? This means that in a satisfying assignment of ϕ at least one of the five literals must be TRUE. In turns out that one can simulate this by

$$D = (a \vee b \vee x) \wedge (\bar{x} \vee c \vee y) \wedge (\bar{y} \vee d \vee e)$$

where x and y are totally new variables. We need to show two things: (1) If all the literals in C are FALSE, then D is FALSE; and (2) If one of the literals in C is TRUE, then there exist settings of x and y such that D is TRUE. (Why?)

To prove (1), observe that if all of a through e are FALSE and D is TRUE, then by first clause of D it holds that x is TRUE. Then by the second clause it holds that y is TRUE. But then the third clause is FALSE. (2) is proved similarly. It is left to the reader (DO IT!) to explain how to deal with clauses C of other sizes.

So this construction yields a boolean formula ψ in the right form. If ϕ is satisfiable, then there is an assignment where each clause is TRUE. This can be extended to assignment where ϕ is TRUE. On the other hand, if an assignment evaluates ϕ to FALSE, then one of the clauses must be false and one of the corresponding family of clauses in ψ must be false. The last thing to check is that the process of conversion can in fact be encoded as a polynomial-time algorithm. Which it can.

◇ *Clique*

A ***clique*** in a graph is a set of nodes each pair of which is joined by an edge.

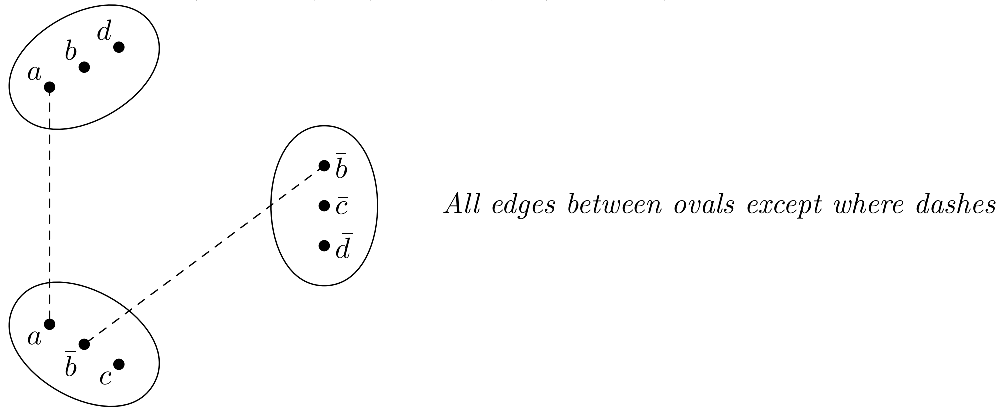
> `CLIQUE`
> Input: Graph G and integer k
> Question: Does there exist a clique with k nodes?

We show that `CLIQUE` is NP-complete. First we must check that the problem is in NP. The program guesses k nodes and checks they form a clique.

We now show how to reduce `3SAT` to `CLIQUE`. That is, we describe a procedure which takes a boolean formula ϕ and produces a graph G_ϕ and integer m such that ϕ is satisfiable iff G_ϕ has a clique with m nodes.

Consider the input ϕ: suppose it has c clauses. For each clause and each literal in that clause, create a node (so we have $3c$ nodes). Then join two nodes if they are from different clauses unless they correspond to opposite literals (that is, do not join a to $\bar{a}$ etc.). The result is a graph G_ϕ.

This is the graph for $(a \vee \bar{b} \vee c) \wedge (a \vee b \vee d) \wedge (\bar{b} \vee \bar{c} \vee \bar{d})$:



Claim: $\langle \phi \rangle \mapsto \langle G_\phi, c \rangle$ is the desired reduction.

First of all, the graph can be constructed in polynomial-time. Then observe that a clique of size c would have to contain exactly one node from each clause (since it cannot contain two nodes from the same clause). Then note that the clique cannot contain both a and $\bar{a}$. Thus, if we set all the literals in the clique to be true, we obtain a satisfying assignment. On the other hand, if we take a satisfying assignment, then a clique with c nodes can be obtained by taking one true literal from each clause.

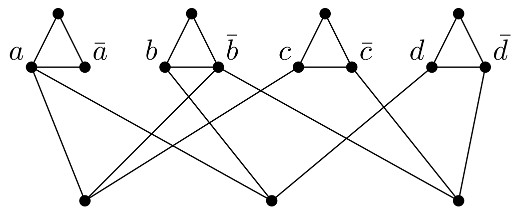
◇ *Domination*

We show `DOMINATION` is NP-complete. First we must check that the problem is in NP. But that is not hard. The program guesses k nodes and then checks whether they form a dominating set or not.

We now show how to reduce `3SAT` to `DOMINATION`. That is, we describe a procedure which takes a boolean formula ϕ and produces a graph G_ϕ and an integer m such that ϕ is satisfiable iff there is a dominating set of G_ϕ of m nodes.

Suppose input ϕ in conjunctive form has c clauses and a total of m variables. For each clause, create a node. For each variable v, create a triangle with one node labeled v and one labeled $\bar{v}$. Then for each clause, join the clause-node to (the nodes corresponding to) the three literals that are in that clause. The result is a graph G_ϕ.

This is the graph for $(a \vee \bar{b} \vee c) \wedge (a \vee b \vee d) \wedge (\bar{b} \vee \bar{c} \vee \bar{d})$:



Claim: $\langle \phi \rangle \mapsto \langle G_\phi, m \rangle$ is the desired reduction.

If ϕ has a satisfying assignment, then let D be the set of the m nodes corresponding to the TRUE literals in the assignment. Then each triangle is dominated (there's one node of D in each triangle). And each clause-node is dominated, since one of the literals in that clause must be TRUE.

Conversely, suppose G_ϕ has a dominating set of size m. Then D must consist of one node from each triangle, since all the unlabeled nodes of the triangles must be dominated. Now to be dominating every clause must be connected to one literal in D. So if we set all the literals (corresponding to nodes) in D to TRUE, we have a satisfying assignment.

◇ ***Independence***

An independent set in a graph is a set of nodes no pair of which is joined by an edge.

> `INDEPENDENCE`
> Input: Graph G and integer k
> Question: Does there exist an independent set with k nodes?

This problem is NP-complete. We simply reduce from `CLIQUE` by taking the complement of the graph: change every edge into a non-edge and vice versa.

◇ ***Independence with Maximum Degree 3***

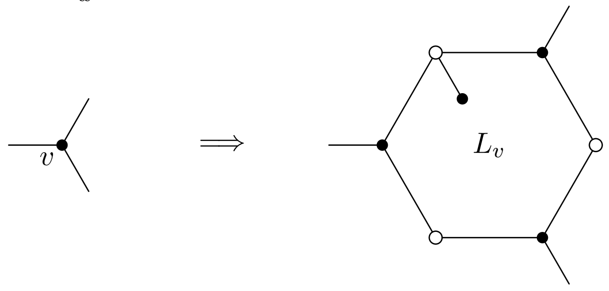
The degree of a node is the number of edges incident with it. We now show that the `INDEPENDENCE` problem remains hard even if the graph is guaranteed to have no node with degree more than 3.

> `IndepMaxDeg3`
> Input: Graph G with maximum degree 3 and integer k
> Question: Does there exist a dominating set of G of at most k nodes?

We now show how to reduce `INDEPENDENCE` to `IndepMaxDeg3`. The input is a graph G and integer k. We produce a new graph H and integer l.

The construction is to replace each node v in G with a larger subgraph, call it L_v. In particular, if v has degree d, then L_v has $2d+1$ nodes: it consists of a cycle of length $2d$

and one leaf adjacent to one node in the cycle. The nodes of the cycle are colored black and white alternately such that the leaf is adjacent to a white node; the leaf is colored black. For each original edge in G, say joining vw, there is an edge joining a black node of L_v and a black node of L_w.



If D denotes the sum of the degrees in the original graph, the claim is that the reduction is:

$$\langle G, k \rangle \mapsto \langle H, D + k \rangle$$

We need to argue that G has an independent set with k nodes iff H has an independent set with $D + k$ nodes. For, given any independent set in the graph outside L_v, one can always add the d white nodes of L_v. The only way to take $d + 1$ nodes in L_v is to take the black nodes, but then one cannot take a black node in any L_w if v and w were adjacent in G.

Exercises

1. Define SPATH as the problem of given a graph G, nodes a and b and integer k, is there a path from a to b of length at most k. Define LPATH similarly except that there is a path of length at least k (and no repeated node). Show that SPATH is in P. Show that LPATH is NP-complete. (You may assume the NP-completeness of HAMPATH.)

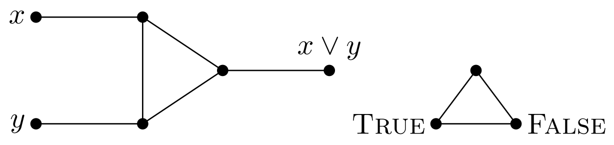
2. Show that the independence number of a graph with maximum degree 2 can be computed in polynomial time.

3. Show that if P=NP then there is a polynomial-time algorithm which on input a graph finds a hamiltonian path if one exists. (Note: this is not immediate.)

4. Show that if P=NP then there is a polynomial-time algorithm which on input ϕ finds a satisfying assignment if one exists.

5. Show that the DOMINATION problem is still hard in bipartite graphs. (Hint: mimic the reduction from 3SAT but change the way a and $\bar{a}$ are connected.)

6. ⓔ Show that deciding if a graph has a 3-coloring of the nodes is NP-complete. *Hint:* You might want to reduce from 3SAT using the gadgets pictured below.


x
y
$x \vee y$
True
False

Chapter 3: Dealing with Hard Problems

When the problem is known to be NP-complete (or worse), what you gonna do? The best one can hope for is an algorithm that is guaranteed to be close. Alternatively, one can ask for an algorithm that is close most of the time, or maybe is correct but only its average running time is fast.

3.1 Approximation Algorithms

These are algorithms which run fast and are guaranteed to be close to the correct answer. A ***c-approximation algorithm*** is guaranteed to be within a factor c of the exact solution.

For example, we saw earlier that there is a polynomial-time algorithm for maximum matching in a graph. But there is a simple 2-approximation algorithm: start with any edge and keep on adding edges to the matching until you cannot proceed. This is called a ***maximal*** matching.

Lemma. Every maximal matching M has size at least half the maximum matching number

Proof. Let M^* be a maximum matching. Since M is maximal, none of the edges of M^* can be added to M. Thus for each edge of M^*, one of its ends must already be used in M. Thus the total number of nodes in M must be at least the number of edges in M^*: but every edge has two nodes, whence the result. ◇

The result is accurate. For example, if one has a path with three edges, then the maximum matching number is 2, but one can get a maximal matching with just the central edge.

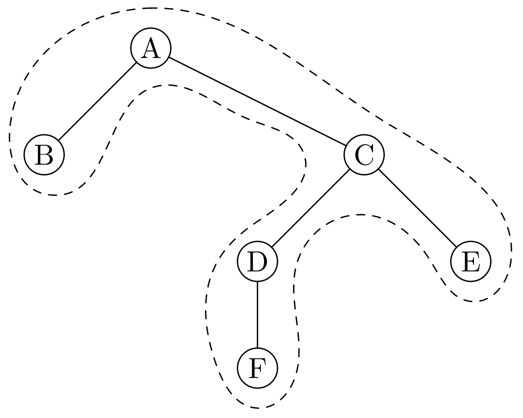
◇ ***Traveling Salesman Problem***

Recall that a hamiltonian cycle is a cycle that visits all nodes exactly once and ends up where it started. In the traveling salesman problem `TSP`, one has the complete graph and each edge has a weight, and the task is to find the hamiltonian cycle with minimum weight. Since the `HAMPATH` problem is NP-complete, it is easy to show/believe that `TSP` is NP-complete too.

Now suppose the graph obeys the ***triangle inequality***: *for all nodes a, b, c, the weight of the edge (a, b) is at most the sum of the weights of the edges (a, c) and (b, c).* Such would be true if the salesman were on a flat plane, and the weights represented actual distances.

There is a simple 2-approximation algorithm for this version. Start with a minimum spanning tree T. Since a tour is a connected spanning subgraph, every tour has weight at least that of T. Now, if one uses every edge of T twice, then one can visit every node at least once and end up where one started.

To turn this into a tour we simply skip repeated nodes. For example, in the picture the traversal of T goes $ABACDFDCECA$ and we short-cut this to $ABCDFEA$. We rely on the triangle inequality for the shortcut to be no worse than the original path.



◇ ***Coloring 3-colorable graphs***

In an earlier exercise you were asked to show that deciding whether a graph is 3-colorable is NP-complete. But consider the following problem: suppose you know that the graph is 3-colorable (or at least your algorithm need only work for 3-colorable graphs). How easy is it to find a 3-coloring?

Well it seems to be very difficult. For a long time the best polynomial-time algorithm known used $O(\sqrt{n})$ colors to color a 3-colorable graph. This algorithm rested on two facts:

1. One can easily 2-color a bipartite (meaning 2-colorable) graph.

2. If a graph has maximum degree Δ, one can easily produce a coloring with at most $\Delta + 1$ colors.

The only fact about 3-colorable graphs that we need is that in a 3-coloring, the neighbors of a node are colored with two colors; that is, the neighborhood of a node is bipartite.

Proving this and putting this all together is left as an interesting exercise.

◇ NP***-completeness***

It can be shown that finding a c-approximation of an independent set, for any constant c, is NP-complete. Indeed, Arora et al. showed that there are problems where for sufficiently small ε, coming within n^ε of the correct value is NP-complete.

3.2 Randomized Algorithms

Sometimes randomness can be the key to obtaining a fast algorithm. There are two basic types of randomized algorithms:

1. A ***Monte Carlo*** algorithm use randomness and the answer is guaranteed to be correct most of the time.
2. A ***Las Vegas*** algorithm use randomness, the answer is guaranteed to be correct, but the running time is only an average.

We give here only a simple example. Suppose you have two copies of the same long binary file at two different locations: call them F_1 and F_2. Then a hacker has a go at F_1. You would like to check whether the file is corrupted. Obviously this can be achieved by sending the one file to the other location. But we would like to minimize the number of bits that are communicated. This can be done with random check-bits.

Assuming that you have the same random number generator at both ends, you can do the following. Compute a random subset S of the bits by tossing a coin for each bit. Then compute the XOR of the bits of F_i corresponding to S: call the result c_i. Obviously, if F_1 is uncorrupted then $c_1 = c_2$.

Lemma. If $F_1 \neq F_2$ then $c_1 \neq c_2$ 50% of the time.

Proof. Consider the last bit in which the files differ: call it b. Let u_1 and u_2 be the calculations so far. If $u_1 = u_2$, then $c_1 \neq c_2$ iff S includes b: which is a 50-50 event. But if $u_1 \neq u_2$, then $c_1 \neq c_2$ iff S does not include b, again a 50-50 event. $\diamond$

Thus, if $F_1 \neq F_2$ there is a 50-50 chance that this random check-bit is different. Thus we have a Monte Carlo algorithm. If you perform this computation 100 times, and each check bit matches, then there is only a 1 in 2^{100} chance that the files are different. This is about the same odds as winning the lottery, four weeks in a row!

Exercises

1. Give a fast algorithm to 2-color a bipartite (meaning 2-colorable) graph.
2. Give a fast algorithm to color the nodes of a graph with at most $\Delta + 1$ colors, where Δ denotes the maximum degree.
3. Hence show how to color a 3-colorable graph with $O(\sqrt{n})$ colors.
4. ⓔ The k-center problem is to place k facilities (e.g. fire-stations), such that the farthest that anyone has to travel to the nearest fire-station is as small as possible. Use a greedy approach to produce a 2-approximation.

5. Explain how to use an unbiased coin to choose among 3 people. How many tosses does your algorithm take?

6. Suppose you have two n-bit files at different locations and they are known to differ in exactly one bit. Give an algorithm to determine which bit they differ in that uses $O(\log n)$ communication.

7. Use dynamic programming to obtain an algorithm for TSP that runs in time $O(2^n)$.

Chapter 4: Lower Bounds

In Algorithms, we are able to prove lower bounds only for very simple problems. We look here at some problems related to sorting.

4.1 Information Theory

Suppose you have to guess a number between 1 and 100 by asking yes–no queries. You might start by asking "Is it 1?" Then "Is it 2?" and so on. But one can do better by asking first "Is it larger than 50?" and if the answer is yes then asking "Is it larger than 75?" and so on. In other words, we can perform a binary search. In the worst case the binary search will take 7 queries. (Try it!) The question is: Is 7 the best we can hope for?

The answer to that question is yes. And the argument is as follows. The ***problem*** is:

> Determine a number between 1 and 100.

There are 100 possible ***solutions***—call the set S. Each time one asks a yes–no question. This splits the set up into sets S_Y and S_N. If the person answers yes, then we are left with the set S_Y. If she answers no, then we are left with the set S_N. One of these sets must have at least 50 elements. So in the worst case we are left with at least 50 elements. In other words:

> *In the worst case the number of solutions comes down by a factor of at most 2 for each query.*

We are done when there is only one possible solution remaining. So if we define $\boldsymbol{I}$ as the number of solutions to the problem then

> *The number of queries needed, in the worst case, is at least* $\log_2 I$

In the example above, the lower bound is $\log_2 100$ which, rounded up, is 7. Note that:

- The lower bound does not assume a particular algorithm, but rather deals with the best one could hope for in any algorithm. It depends only on the number of solutions.
- The lower bound is also valid for the average case behavior of any algorithm. The idea is that one query will on average decrease the number of solutions by a factor of at most 2.

The information-theory lower bound is the only general procedure out there for providing estimates of the time required by the best possible algorithm. In applying information theory, we must determine or estimate I; this is not always so simple.

4.2 Examples

Sorting.
Here the problem is: given a list of numbers, output them in sorted order. The solution is the sorted list. For a list of n numbers there are $I = n!$ solutions. So a lower bound on sorting is given by

$$\log_2(n!)$$

Since there is an approximation

$$n! \approx \left(\frac{n}{e}\right)^n$$

$\log_2(n!)$ is about $n \log_2 n - O(n)$.

This means that any comparison-based sorting algorithm must take at least this many comparisons (in the worst case). And hence the best we can hope for time-wise for a sorting algorithm is $O(n \log n)$. Merge sort comes close to this value.

Maximum.
Here the problem is: given a list of numbers, output the maximum. For a list of n elements there are n answers. So a lower bound on maximum-finding is given by

$$\log_2 n$$

However, there is no algorithm for finding the maximum that runs this fast. (This problem is looked at again in the next section.)

Set-Maxima.
Suppose one has as input the values of A, B and C and one must output the maximum in each of the three sets $\{A, B\}$, $\{A, C\}$ and $\{B, C\}$. Naïvely there are 2 possible answers for the maximum in each set, so one might think that $I = 2^3$. However, this is not correct. It cannot happen that the maximum in $\{A, B\}$ is A, the maximum in $\{B, C\}$ is B and the maximum in $\{C, A\}$ is C. (Why not?) In fact $I = 6$ (or 3!) as there is one possible output answer for each ordering of the set $\{A, B, C\}$.

4.3 Adversarial Arguments

We want to know about the worst-case behavior of a comparison-based algorithm. Consider, for example, computing the maximum of n elements using comparisons: this takes at least $n - 1$ comparisons. The idea is that if one wants a winner out of n players, each player except for the winner must lose at least one game, hence there must be $n-1$ games.

This type of argument can be generalized to what is called an ***adversarial argument***: we assume that we are asking a question of an adversary who is making up the data as he goes along. Of course, he is not allowed to invalidate the answers to previous questions.

Consider the problem of computing the maximum and minimum of n elements simultaneously. For an adversarial argument, we must describe a strategy for the adversary. He will create two buckets: L and H for low and high. Initially, both buckets are empty. Each value in L will be less than each value in H.

For each question (which is of the form "Is $A[i]$ bigger than $A[j]$"), the adversary does the following.

1. If neither element has been assigned to a bucket, he arbitrarily puts one in L, one in H.
2. If only one element has been assigned to a bucket, then he puts the other element in the other bucket.
3. Otherwise the adversary does nothing.

Then to answer the question, the adversary chooses any answer that is compatible with previous information and respects the property that all of H is bigger than all of L.

Obviously, the user must ask a question about each element (okay, we should assume $n \geq 2$). So every element will eventually be assigned to a bucket. Furthermore, the user has to determine the maximum in H and the minimum in L. Each question provides information about only one thing: (a) the assignment to buckets, (b) the maximum in H, or (c) the minimum in L.

Assigning to buckets takes at least $n/2$ comparisons. Determining the maximum in H and the minimum in L (which are disjoint) together takes $(|H|-1)+(|L|-1) = n-2$ comparisons. Thus, determining the maximum and minimum in the set takes at least $3n/2 - 2$ comparisons.

Exercises

1. Sorting takes $O(n \log n)$ comparisons. How long does it take to ***check*** whether a list is sorted or not?
2. ⓔ Consider a set-maxima problem where one is given a list of n numbers and n subsets from this list, and one must find the maximum in each subset. Find an information-theoretic lower bound on the number of comparisons needed.
3. ⓔ Use an adversarial argument to give a $3n/2 - O(1)$ lower bound on the number of comparisons needed to find the median of a list of numbers.

Chapter 5: Books

I have made much use of:

• *Data Structures and Algorithms*,
A. Aho, J. Hopcroft and J. Ullman

• *Fundamentals of Algorithmics*,
G. Brassard and P. Bratley

• *Introduction to Algorithms*,
T. Cormen, C. Leiserson and R. Rivest

• *Introduction to Parellel Algorithms and Architectures: Arrays, Trees, Hypercubes*,
F.T. Leighton

• *Data Structures and Algorithm Analysis,*
M.A. Weiss

• *Introduction to Computer Theory,*
D.I.A. Cohen

• *Introduction to Discrete Mathematics,*
S.C. Althoen and R.J. Bumcrot

• *Mathematical Theory of Computation,*
Z. Manna

• *Introduction to the Theory of Computation,*
M. Sipser