Algorithms: An Undergraduate Course with Programming

William F. Klostermeyer
School of Computing
University of North Florida
Jacksonville, FL 32224
E-mail: klostermeyer@hotmail.com

Contents

Chapter 0

Preface

These notes are designed to be accessible: for students to read with the goal of teaching them to understand and solve algorithmic problems. Numerous programming assignments are given so that students can see the impact of algorithm design, choice of data structure, etc. on the actual running time of real programs.

Emphasis is given to both the design and analysis of algorithms and although a few proofs of correctness and other mathematical proofs are given and assigned that is not the primary emphasis of this text. Implementation details are also considered an important part of the learning experience and, accordingly, programming assignments are numerous and compilable Java code is used for all code in the book: most of these programs have exercises associated with them to encourage the students to run them on interesting inputs. Students are expected to have had one semester courses in calculus, data structures, and ideally, discrete mathematics, though a tour of the Mathematical Preliminaries chapter should provide a sufficient background for those who have not had a discrete math course. Likewise, those students who have not had a full semester of data structures, could spend a few weeks on the data structures chapter before proceeding with the rest of the text (such might be the model for a course that many schools offer called "Data Structures and Algorithms.")

It is not my goal in these notes to avoid mathematical rigor, rather, I have found that a course in algorithms is challenging for most students, as they have not yet developed a lot of experience in algorithmic problem solving. Thus I believe development of these skills is essential for many students, before they can be exposed to a hard-core course in algorithmics.

Most of the "Algorithms" books that I have read are either aimed at the researcher/practitioner (such as the wonderful book by Skiena [31]) or are too intimidating, terse, thick, or advanced for the "average" undergraduate student. These notes were specifically designed to help the student learn to understand the basic, fundamental concepts, without overwhelming them. I hope the exercises, programming assignments and the exposition meet this objective. I have evolved this material through numerous adventures in the undergraduate algorithms course at West Virginia University and the University of North Florida and have also used parts of this text (e.g., some topics from the Advanced Algorithms chapter and some from the Complexity Theory chapter) in graduate courses. But all the material, including the exercises, is designed to be highly accessible to undergraduates, even those algorithms that I refer to as "advanced." However, some material has been included for completeness sake that may not need to be covered in a first course in algorithms – the chapter on NP-completeness, for example (I typically spend only one or two days on this in a first undergraduate algorithms course) as well as the linear algebra material (again, I may spend one day at most on this). But I do think these important topics warrant some discussion and may be included for more advanced students or in a second semester undergraduate algorithms course.

0.1 Course Outlines

The course outline I have used for a one-semester undergraduate course is as follows:

1-2 Introduction and Algorithmic Thinking 1 week

9 Mathematical Preliminaries 2 weeks
9.1 Series and Sums
9.2 Graphs
9.3 Induction

3 Growth of Functions 2 weeks
3.1 Big Oh et al.
3.2 Some Example Code Segments
3.3 Recursive Algorithms
3.3.1 Recurrence relations
3.3.2 Master Theorem

4 Sorting, Selection, and Array Problems 2 weeks
4.1 Searching
4.2 Sorting
4.2.2 HeapSort
4.2.3 Linear Time Sorting
4.3 Finding the Median
4.6 Another Recursive Algorithm
4.6.1 Naive Matrix Multiplication
4.6.2 Recursive Matrix Multiplication

5 Graph Algorithms and Problems 4 weeks
5.1 Storage
5.2 Algorithmic Techniques
5.2.1 Greedy Algorithms
5.2.2 Brute-Force
5.3 Path Problems
5.3.1 Breadth-First Search
5.3.2 Dijkstra's Algorithm
5.3.4 All-Pairs Shortest Path
5.3.5 Maximum Flow
5.4 Spanning Trees
5.4.1 Union-Find Data Structures
5.4.2 Prim's Algorithm
5.4.3 Kruskal's Algorithm
5.4.4 MST with 0-1 Edge Weights

6.6 Linear Algebra Tools – An Overview 0.5 weeks
6.6.1 Solving Sets of Linear Equations
6.6.3 Linear Programming

7 Complexity – An Introduction 1 week

I then generally to spend the remaining two to three weeks on selected topics.

A lower-level (sophomore?) course called "Data Structures and Algorithms" might use an outline such as the following:

1 Introduction 0.5 weeks

9 Mathematical Preliminaries 2.5 weeks
9.1 Series and Sums
9.2 Graphs
9.3 Induction

10 Data Structures 3 weeks
10.1 Lists
10.1.1 Linked Lists, Doubly-Linked Lists, Queues, Stacks
10.1.2 Move-to-Front-Lists
10.2 Trees
10.2.1 Binary Trees
10.2.2 Heaps
10.2.2.1 Heaps as Trees
10.2.2.2 Heaps as Arrays
10.2.2.3 How many arrays are heaps?
10.2.3 Balanced Trees
10.2.4 B-trees 10.2.5 Splay Trees
2 Algorithmic Thinking 2 weeks

3 Growth of Functions 2 weeks
3.1 Big Oh et al.
3.2 Some Example Code Segments
3.3 Recursive Algorithms
3.3.1 Recurrence relations
3.3.2 Master Theorem

4 Sorting, Selection, and Array Problems 2 weeks
4.1 Searching
4.2 Sorting
4.2.2 HeapSort
4.2.3 Linear Time Sorting
4.3 Finding the Median
4.6 Another Recursive Algorithm
4.6.1 Naive Matrix Multiplication
4.6.2 Recursive Matrix Multiplication

5 Graph Algorithms and Problems 2 weeks
5.1 Storage
5.2 Algorithmic Techniques
5.2.1 Greedy Algorithms

5.2.2 Brute-Force
5.3 Path Problems
5.3.2 Dijkstra's Algorithm
5.3.4 All-Pairs Shortest Path
5.3.5 Maximum Flow
5.4 Spanning Trees
5.4.2 Prim's Algorithm
5.4.4 MST with 0-1 Edge Weights

7 Complexity – An Introduction 1 week

0.2 Java Notes

All pseudocode programs in this text have Java counterparts on my web site. All programs in this text were compiled with JDK 1.2, available from www.sun.com (Sun Microsystems). I have attempted to make the code as clear as possible, and thus it may not be as "object-oriented" as some may like. I have annotated code with "←" symbols to indicate where one might place statements such as "counter++;" as suggested in the exercises following the programs. That is, some exercises ask the student to embed a counter inside a program and print the value of the counter at the end of the program to observe the number of iterations. Again for clarity, I do not include the statements related to the counter in the code, but do indicate where these statements should be placed.

Most input/output is done with the following methods, which are not included in the code in the text, for sake of brevity and clarity, but will need to be included with any programs that use them. The methods are taken from the text "Data Structures and Algorithms in Java" by Robert Lafore.

At the beginning of the program file we use:

```
import java.io.*;                    // for I/O
import java.lang.Integer;            // for parseInt()
```

At the end of the "main" class, we place the following:

```
//-------------------------------------------------------------
   public static void putText(String s)
      {
      System.out.print(s);
      System.out.flush();
      }
//---------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   public static char getChar() throws IOException
      {
      String s = getString();
      return s.charAt(0);
      }
//--------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
```

Chapter 1

Introduction

Anyone who has ever written a program is hopefully aware of the time required for the program to execute. Some programs may run so fast you don't notice, while others may run for seconds or minutes or even more. As an example, try implementing the following (in your favorite language), which inputs an integer and determines whether or not it is prime (recall that an integer n is prime if it is evenly divisible only by 1 and n, so for example, 12 is not prime since $12 = 2*2*3$, and 17 is prime. In other words, $17 \bmod i = 0$ for all i such that $2 \leq i \leq 16$):

```
import java.io.*;                               // for I/O
class TrialDivision
{
   public static void main(String[] args)
   {
      int n;
      boolean prime;

      n=getInt();

    // test if n is prime

      prime=true;
      for (i=2; i < n; i++) {
          if (n % i == 0) prime=false;
      }
```

```
     if (prime) putText("Prime") else putText("Composite');
   }
}
```

Exercise Run this program for a large value of n, say, 32,567. How long did it take? What about for $n = 1,000,003$? How long did it take?

The clever reader will realize that we can improve the program as follows, since if n has a non-trivial divisor, it must have one whose value is at most $\sqrt{n}$:

```
n : long_integer;
prime : boolean;

input(n);

-- test if n is prime --

prime:=true;
for i in 2..sqrt(n)
   if n mod i = 0 then
      prime:=false;
end;

if prime then print("Prime") else print("Composite');
```

Exercise How long did this program take for $n = 32,567$, for $n = 1,000,003$? How long does it take for $n = 2^{32} - 1$?

Exercise Write a program to implement the sieve of Eratosthenes (dates from around 200 B.C.) to test if an integer x is prime: in the range from 2 to x, mark all numbers divisible by 2 that are greater than 2, (so you would mark 4, 6, 8, 10, ... in increasing order); then mark all numbers divisible by 3 that are greater than 3. Each subsequent iteration, mark all numbers divisible by smallest unmarked number that has not been used before. Numbers left unmarked at the end are prime. Compare the running time of this on large integers with the previous approaches.

Now we have seen several programs to solve the same problem (testing for primality), where one runs quite slowly and another runs more quickly (though, as you should observe, still fairly slow for super large values of n). [1]. The fact that factoring very large integers is computationally time-consuming is the basis for many encryption schemes such as the RSA algorithm [11]!

Algorithmic problems come in many varieties: numerical problems such as primality testing or finding the roots of a polynomial, matrix problems, algebraic problems, logic problems (is a given Boolean formula a tautology), graph problems (find the largest independent set in a graph), text problems (find the first occurrence of some pattern in a large text), game problems (is a given chess position winning for white), and many others. But for each problem, we shall be concerned with whether it is solvable, whether it is efficiently solvable, and how to design and analyze algorithms for these problems.

Every year it seems, faster (and cheaper) computers become available and the computer companies entice buyers with how fast their programs will run. However, equally important as the speed of the computer or processor is the speed of the program (or *algorithm*) itself. That is, a computer that is twice as fast as an old computer can only speed up our programs by a factor of two; whereas a faster algorithm might speed up a program by a hundred-fold. Compare the above two programs: run the slower one on a fast computer and the faster one on a slow computer for some large inputs. What can you conclude?

Our goal in this text is to understand how the choice of algorithm (or data structure) affects the speed with which a program runs and be able to effectively solve problems by providing efficient algorithmic solutions, if an efficient solution is in fact possible.

The ACM (Association for Computing Machinery, the main professional organization for computing professionals) published a definition of computer science that begins

"Computer science ... is the systematic study of algorithmic processes." (from "Computing as a Discipline" by the ACM Task Force on the Core of Computing, *Communications of the Association for Computing Machinery*, vol. 32, 1989).

[1]Faster, though more complex, algorithms for primality testing exist [27, 24, 1, 2]

We may define an "algorithm" as a finite sequence of steps (instructions) to solve a problem. A computer program is, in purest form, just a realization of an algorithm.

There are three sides to the study of algorithms:

(1) Analysis: given a program (or an algorithm), how fast does it run (or how much space does it use) and how can we improve or optimize its performance.

(2) Design: given a problem, how can we solve it efficiently (if in fact, it is solvable or tractable). A problem is "tractable" if it admits an efficient algorithm. We shall see that there are problems (color the vertices of a graph with as few colors as possible so that adjacent vertices have different colors) that, although solvable, are not tractable. By "solvable", we mean there exists an algorithm to solve every instance in finite time. The Halting Problem is an example of an unsolvable problem. In essence, the Halting Problem asks you to write a program to input the source code of an arbitrary "C" program and output whether or not the program halts on some particular input (let alone all possible inputs!).

(3) Implementation: given the pseudo-code for an algorithm, implement it in a (high-level) programming language on an actual computer. Experimentation, or simulations, are often done to compare the performance of implementations of algorithms (either different implementations of the same algorithm or implementations of different algorithms for the same problem) on realistic inputs. Memory considerations, optimizations, language issues all become factors when implementing an algorithm.

We shall be interested in the amount of resources consumed by an algorithm, especially as the size of the input gets large. By resource, we might mean space (or memory) used, number of random bits used, or, most commonly, the amount of processing time taken.

In general, we shall be concerned with the "worst-case" running time of algorithms – how much time to they require in the worst case, rather than "best case" (which is usually trivial and not very useful) or average-case (which requires some assumption about the probability distribution from which the input is drawn, and can be hard to determine).

By *running time* of an algorithm, we shall generally mean a function describing the number of steps performed by the algorithm in terms of the input length (the amount of storage space/memory needed to store the input). We usually denote the input length by n – for example, the size of an array we must process. Then by $t(n)$ we represent the maximum number of steps taken by our program over all possible inputs of length n. In this way, we will be able to estimate the actual running time (in minutes and seconds) as our inputs become larger. For example, if we can sort n integers in n^2 steps and we know that we can sort 10 integers in 1 second; then we can estimate that sorting 100 integers will take 100 seconds and sorting 1000 integers will take 10,000 seconds (which is almost three hours!). Some of the programming assignments, on the other hand, will ask you to evaluate the actual time taken by programs on certain inputs and to compare these times with the "theoretical" running times.

In some cases, a programmer may be more interested in some other resource rather than time (space is an obvious one, but there are others that may be of interest such as randomness). In this text, however, our focus will generally be on the amount of time an algorithm uses as a function of the input size.

Let us give four simple examples. Consider the following code segments.

```
Input n
Repeat
   if n is even then n:=n/2
      else n:=n+1
until n <= 1
```

```
Input n
Repeat
   if n is even then n:=n/2
      else n:=2n-2
until n <=1
```

```
Input n
Repeat
   if n is even then n:=n/2
       else n:=3n+1
until n<=1
```

```
Input n
k:=0
for i:=1 to n
  for j:=1 to n
     k:=k+1;
  end;
end;
print(k);
```

How many iterations (in terms of the input n) does each segment perform in the worst case? Try some examples ($n = 128, n = 11, n = 13$). If you cannot see a pattern emerging, write a program for each and run with several different inputs. Even if you see the pattern, write the programs and compare the actual running times of each (in minutes and seconds) for various n, say $n = 20, 100, 1000$.

In the first case, if we pair consecutive iterations, we can see that we will roughly reduce the number by half until we reach 1. Hence we can say that about $2 \log_2 n$ iterations are needed in the worst case. In the second segment, we can see that consecutive pairs of iterations will always reduce the number by at least 1. Hence at most $2n$ iterations are needed. The third case is the famous "Collatz Problem:" no one knows if it terminates on all inputs or not! The fourth program takes n^2 steps (it simply computes n^2 from the input n in a very silly way!)

The following table shows the impact the complexity (i.e. this function describing the running time) has on the actual running time of a program. Suppose we have a computer that performs 10,000 operations per second. Let us see how long it will take to run programs with varying complexities and various small input sizes.

Complexity	5	10	20	100	1000
N	< 0.1 sec	< 0.1 sec	< 0.1 sec	< 0.1 sec	0.1 sec
N^2	< 0.1 sec	< 0.1 sec	< 0.1 sec	1 secs	1.7 minutes
2^N	< 0.1 sec	< 0.1 sec	1.7 minutes	4×10^{19} centuries	3.4×10^{278}
$n!$	< 0.1 sec	6 minutes	7.7×10^7 centuries	3×10^{144} centuries	

So you may say, "but that is on a very slow computer." How much difference do you think it would make if we used a computer that was twice as fast? 1000 times faster? Will you really notice the difference between

1000 centuries and one century?

1.1 Exercises

1. For what values of n is 2^n less than n^3?

2. Search the Internet for the word “algorithm” and list a few important applications and uses of algorithms. You may wish to refine your search to say, encryption algorithms, or graph algorithms.

1.2 Programming Assignments

1. Input a positive integer n. Let $k = 0$ initially. For n steps choose a random number either 1 or -1 (with equal probability). Add this random number to k. Output the largest and smallest values of k (and largest absolute value of k) that are obtained over the course of these n steps. Repeat this *random walk* experiment for several large values of n, such as 1000; 100,000; and 1,000,000. How does the output vary for different n.

Try the experiment again, except never let k become negative. That is, if $k = 0$ and -1 is the random number, k stays at 0.

Try the experiment again, except let the probability of 1 be two-thirds and the probability of -1 be one-third.

2. Write a program to determine if a number is a *perfect number*. A number n is a perfect number if it is equal to one-half the sum of all its factors. For example, $28 = \frac{1}{2}(28 + 14 + 7 + 4 + 2 + 1)$. Observe the running time of your program for various large and small inputs.

3. Write two programs to search for a key value in an array: one using sequential search and one using binary search (those unfamiliar with binary search may find it detailed in Chapter 3 and sequential search is detailed in Chapter 4). Compare the running times of the two programs to various large and small arrays sizes.

1.3 Chapter Notes

Several interesting and sometimes amusing “war stories” (anecdotes) on how algorithm design impacted actual programmers can be found in [31].

Chapter 2

Algorithmic Thinking

In this short chapter, we will get warmed up by solving some problems. Problem solving is like swimming: listening to (or reading) lectures on the subject do not make one skilled at it, so the reader should think about the problems before proceeding to read the various solutions, then should work the suggested exercises.

Note that if a problem concerns an array $A[n]$, the program to solve the problem should initially input an integer n, then create an array A with n elements, then input n values into A, then solve the specified problem.

Problem 1. An element x in an array $A[n]$ is called a *majority* element if x occupies more than half the locations of A. Write a program to determine if an array contains a majority element.

Solution. We give a simple, albeit slow solution. We shall re-visit this problem in Section 4.4 and give a faster, linear time, solution.

```
input(A[1..n])

for i=1 to n
  count[i]=0;
end for;

for i=1 to n
   for j = 1 to n
       if A[i]=A[j] count[i]++;
   end for
end for


for i=1 to n
   if count[i] > n/2 then begin
      print(A[i], ``is a majority element.'');
      break;
   end
end for;
```

This solution requires about $n^2 + 2n$ operations, the most significant being the n^2 operations in the nested "for" loops.

Exercise: Can you improve upon this solution if we require that each element of A be an integer between 1 and n?

We can improve upon this as follows.

```
input(A[1..n]);

sort A;  -- use HeapSort (see Chapter 4)

count=1;
for i = 1 to n-1
   if A[i]=A[i+1] then count++
      else count=1;
   if count > n/2 then
       print(A[i], ``is a majority element.'');
end for;
```

This algorithm uses fewer than $2n \log n$ operations, which is a big improvement over our previous solution when n is large. The reason for this is that HeapSort can sort the array using fewer than $2n \log n$ comparisons.

As stated above, we can reduce this to something on the order of n operations using a technique from Chapter 4.

Now let us attempt to quickly determine if an array has a majority element (in the terminology of Chapter 3, in $O(n \log n)$ time) however, we may only use the "=" and "≠" comparison operators to compare elements of A So we cannot use $<$, $\leq$, $>$, or $\geq$ to compare elements of A.

Solution. The key observation is that if an array A[1..n] has a majority element, then so does either A[1..n/2] or A[n/2+1 .. n]. This observation enables us to solve the problem recursively!

```
-- We assume for simplicity that all elements of A are non-negative
-- otherwise, we need to be more careful about what we return from
-- the function, to identify the cases when the array has a majority
-- (in which case we need the actual value of that majority element
-- and the case when the array does not have a majority element

function majority(A[1..n] of integer) return integer

int m1, m2;
int count;

if n=1 then return A[1]
   else if n=2 then
               if A[1]=A[2] then return A[1]  -- has majority
                  else return -1              -- no majority
        else                                  -- n > 2
           m1 = majority(A[1..n/2]);
           m2 = majority(A[n/2+1..n]);
           if m1 != -1 then begin             -- left half has majority!
              count1 = 0;                     -- test if it is a
              for i = 1 to n                  -- majority for A
                  if A[i]=m1 then count++;
              end for;
              if count > n/2 then return m1;
           end if;
           if m2 != -1 then begin             -- right half has majority!
              count1 = 0;                     -- test if it is a
              for i = 1 to n                  -- majority for A
                  if A[i]=m2 then count++;
              end for;
              if count > n/2 then return m2;
           end if;
    return -1;
end majority;
```

```
main program:

input(A[1..n])
maj = majority(A);

if maj != -1 then print(maj, ‘‘is a majority element’’);

end main;
```

Using the analysis methods from Chapter 3, we will be able to show that this algorithm requires about $n \log n$ operations.

Exercise. Given $2n$ items with a majority element (one occurring element occurring at least $n+1$ times), find the majority element. This in the *streaming* data model in that you are allowed only one pass over the data (which means you can only read the elements one each and in sequence), and you are allowed $O(1)$ additional storage.
Problem 2. An element x in an array $A[n]$ is called a *mode* element if x occurs more frequently than any other element in A (if more than one such x occurs, the array has multiple modes). Write a program to compute the mode(s) of an array.

Problem 3. Write a $O(n)$ time algorithm to find the "slice" (i.e., contiguous sub-array) of an array whose sum is maximum over all slices. Hint: the empty slice is defined to have sum zero.

Bonus: Write an efficient algorithm to find the slice whose product is maximum. (Hint: you might first want to consider the cases when (1) the array has no negatives or (2) the array contains only integers).

Exercise: What is the running time of this slow slice algorithm?

```
input A[1..n]
max=0
for i=1 to n
   for j=i to n
       sum=0
       for k = i to j
            sum=sum+A[k]
       end
       if sum > max
          max=sum
    end
end print(max)
```

Problem 6. Write an algorithm, and analyze its running time, to decide if an array can be folded k times so that the resulting array is sorted in ascending order. Solve the problem for $k=1$ and $k=2$ and ponder how one might compute the minimum k for which the array can be folded so as to be sorted. Note: folding an array $a[1\ldots n]$ at index i produces array $a[i+1,\ldots n,1,\ldots i]$.

2.1 Programming Assignments

Chapter 3

Growth of Functions

As mentioned in Chapter 1, we shall use mathematical functions to describe the running times of algorithms. Furthermore, we shall only be interested in the approximate running time: we ignore small and constant factors and seek the “order” of the running time. That is, it is often difficult to determine precisely the number of steps that an algorithm performs (not to mention the impact that choice of language, compiler, hardware, and cpu speed might have on the exact number of “primitive” operations or assembly language instructions your program actually executes). In addition, for various inputs, your program may take a different number of steps. For example, consider **bubble sort**:

```
input array A[1..n]
i:=1; ok:=false;
while (i $\leq n$) and (not ok) loop
   ok:=true
   for j:=1 to n-i
      if A[j] > A[j+1] then
         temp:=A[j]
         A[j]:=A[j+1]
         A[j+1]:=temp
         ok:=false
   end loop
   i:=i+1
end loop
```

Of course, a simpler version of bubble sort is as follows:

```
input array A[1..n]
for i:=1 to n
   for j:=1 to n-i
      if A[j] > A[j+1] then
         temp:=A[j]
         A[j]:=A[j+1]
         A[j+1]:=temp
   end loop
end loop
```

We can see that the latter algorithm always performs about $\frac{n^2}{2}$ comparisons. By "comparison," we mean the number of times the "if" statement is executed. However, the first bubble sort will only perform about n comparisons if the initial array A is sorted. But is A is very "unsorted" to start with (for example, if A is in reverse sorted order), then the algorithm will take about $\frac{n^2}{2}$ comparisons. Try this yourself with an array with six elements.

In fact, we can use summations to count the number of comparisons done by the second bubble sort:

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

And since the average value of the terms in this summations is $\frac{n(n-1)}{2}$, we see the number of comparisons is about $\frac{n^2}{2}$.

In some cases, analysts have determined how many steps certain algorithms perform "on average." Bubble sort is known to require $\Omega(n^2)$ comparisons, on average. HeapSort and QuickSort are two such algorithms, which shall be discussed later. In these case, the "average-case" means that one assumes that all inputs of size n are equally likely. The expected number of comparisons performed by QuickSort is about $1.39n\log n$ [32] and for HeapSort it is about $2n\log n$, see for example [9], though this depends on the implementation.

In general we will be interested in the **worst-case** running time of an algorithm: what is the maximum number of steps it will perform, over all possible inputs. We shall express this running time as a function of the input size, being interested in how the running time increases as the input size increases and becomes very large. In Figure 3 we see that the slope of the curve is not too steep, perhaps reflecting an algorithm whose running time is **linear** in its input size, though of course we cannot make judgment about **all** possible input sizes from just the small sample shown in the Figure.

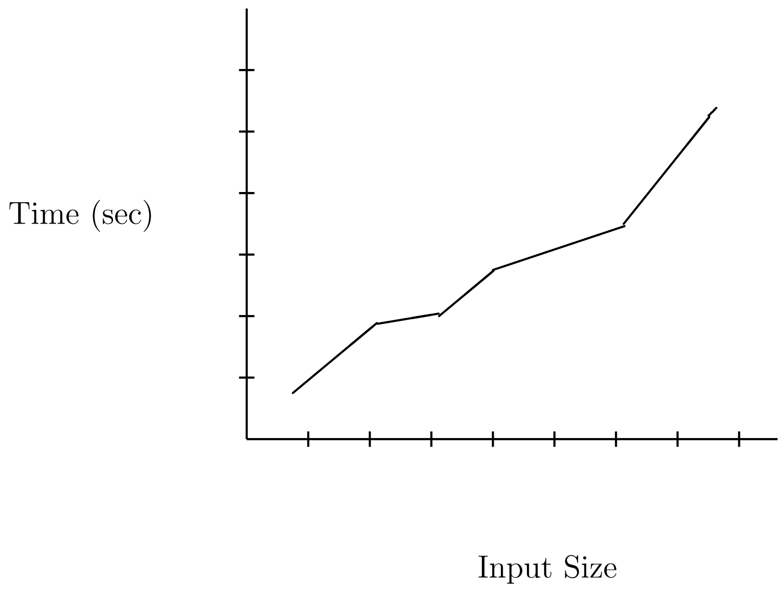


Figure 1. Function Growth

Notation: We will usually use n to denote the size of the input of an algorithm.

Question: If the input to our algorithm is an integer k, what is the size of the input to the algorithm? (Hint: How many bits are required to represent k in binary?).

3.1 Big Oh et al.

Big Oh, denoted by $O()$, describes an upper bound on a function (to within a constant factor). Remember, we use functions such as $t(n) = n^2$ to describe the running times of algorithms. We will most often use $O()$ to describe the running time of an algorithm.

Definition 1 *Let f and g be functions of n. $f(n)$ is said to be $O(g(n))$ if there exist positive constants c and k such that $f(n) < cg(n)$ for all $n \geq k$.*

We sometimes denote this as $f(n) \in O(g(n))$ or $f(n) = O(g(n))$.

In this definition, the multiplicative constant c may be quite large, for example, 5000. The definition says that for all sufficiently large n (that is, all $n \geq k$) we have that $f(n)$ is bounded from above by c times $g(n)$. Our general approach in discussing the running times of algorithms will be to ignore multiplicative factors, hence the use of big-Oh. For example, we would say that $20n$ is $O(n^2)$ even though for small n (that is, for $n \leq 20$) $20n$ is greater than n^2. See Figure 3.1 for an illustration of this.

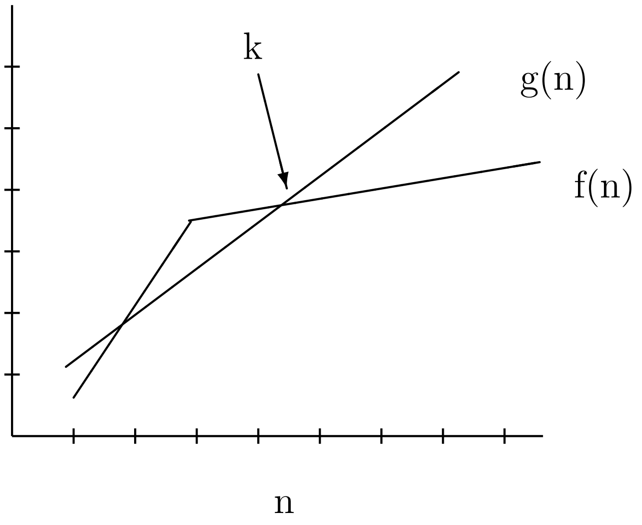


Figure 2. Function Growth

In Figure 2, we can see that for all $n \geq k$ $g(n)$ is above (is an upper bound for) $f(n)$. As another example, we can show that $\log^2 n \in O(\sqrt{n})$. Recall that $\log^2 n = (\log n)^2$. This does not seem obvious since, for example, $\log^2 64 = 36$ and $\sqrt{64} = 8$. However, in the **limit**, that is, as $n \to \infty$ one can show that $\log^2 n < O(\sqrt{n}$, for all sufficiently large n. Compute the values of these two functions for $n = 1,000,000$.

If we consider $f(n) = n$ and $g(n) = \frac{n}{2}$, we can see that $f(n) > g(n)$ for all n. However, $n \in O(\frac{n}{2})$ since we can choose $c \geq 2$ in Definition 1 to ensure that $n \leq c\frac{n}{2}$. Likewise we can observe that

$$n^2 + n + 7 \in O(n^2 - 100n - 43)$$

since $n^2 + n + 7 \leq 200(n^2 - 100n - 43)$ for all $n \geq 200$.

Another way to think about this last example, and big-Oh in general is that *low-order* additive terms do not affect the growth rate in a significant way. That is, the terms $n, 7, -100n, -43$ in $f(n)$ and $g(n)$ above are low-order, compared to the dominant term, n^2. We can see this by the fact that by setting $c = 200$, we ensure $g(n) \geq f(n)$ for all sufficiently large n. So although low-order terms to impact the actual running time of an algorithm, they are not the most significant factor in the overall running time. Thus we tend to ignore them when doing big-Oh analysis.

Some additional examples of function comparison are given below as well as provided in the exercises at the end of the chapter.

The goal is to be able to compare the growth **rates** of functions as n gets large. This will enable us to say which of two programs is "faster" for large inputs (since it is generally the case that most programs are fast for small inputs!). So if we say an algorithm takes $O(n^2)$ time, we mean that the number of steps it performs is at most $c \times n^2$ for some constant c (which never changes and which does not depend on n) and for all possible inputs of size n (for each n). We could then say that this program is "faster" than a program whose running time is $O(n^2 \log n)$ [1]

As we use big-Oh to describe an upper bound, we use Ω to describe a lower bound. The definition is as follows.

Definition 2 *$f(n)$ is $\Omega(g(n))$ if there exist positive constants c, k such that $f(n) \geq cg(n)$ for all $n \geq k$.*

In other words, if $f(n) \in O(g(n))$ then $g(n) \in \Omega(f(n))$.

But sometimes it will be more meaningful to use the following definition.

Definition 3 *$f(n) \in \Omega'(g(n)$ if $f(n) \geq g(n)$ for infinitely many n.*

[1] Assuming this second program does not in fact run in time $O(f(n))$ for some $f(n) << n^2 \log n$.

At first glance this seems identical to Definition 2. But consider the function in Figure 3.1.

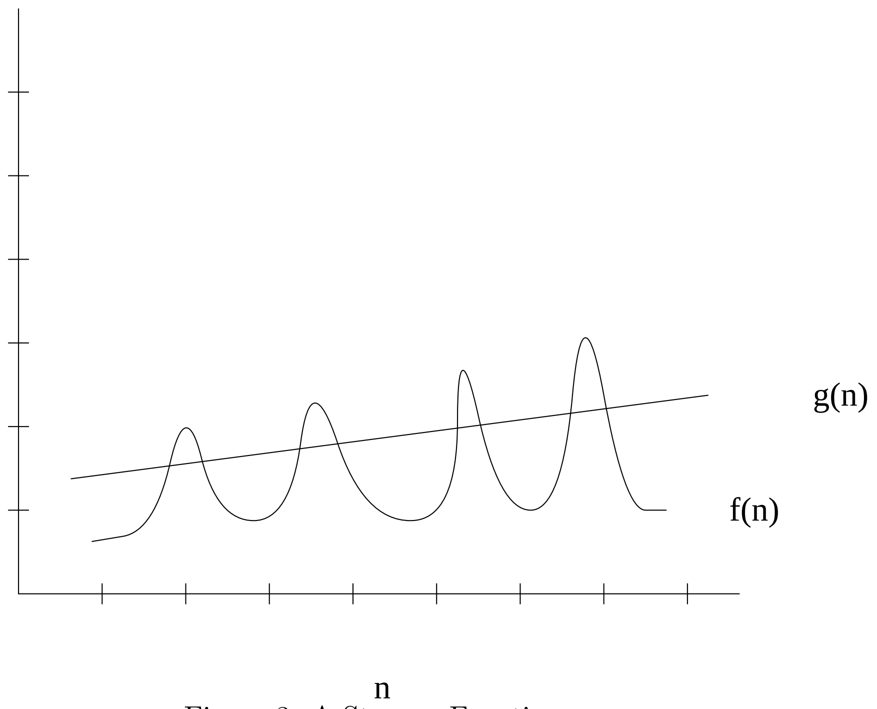


Figure 3. A Strange Function

In this function, $f(n)$ is above $g(n)$ infinitely often. In algorithmic terms, consider the following:

```
input(n)
k:=0
if n mod 100 = 0 then
   for i:=1 to n loop
      k:=k + n
   end
print k
```

If we graph the running time of this function, we would get a graph similar in shape to that in Figure 3. And we would like to convey that the worst-case number of iterations the program performs is at least n, for infinitely many n; hence we could say the number of iterations is $\Omega'(n)$ (but not $\Omega(n)$). But the number of iterations (in the worst-case) taken by this program is also $O(n)$. So you might ask what is really the difference between O and Ω'. Oftentimes, we are unable to precisely describe the number of steps taken by a program using nice everyday functions. Consider the $3n+1$ problem again:

```
Input n
Repeat
   if n is even then n:=n/2
       else n:=3n+1
until n<=1
```

We can easily see that the number of steps required by this program is $\Omega(\log n)$ as well as $\Omega'(\log n)$. Furthermore, we say that the number of steps is $\Omega'(\log(3n+1))$, since there are infinitely many inputs that are even and infinitely many that are odd. And certainly you could describe larger functions than $3n+1$ to use in Ω'. But no one knows any finite function that provides an upper-bound on the running time of this program! Other such examples exist, such as Miller's algorithm to test if an integer is prime or not. This algorithm runs in polynomial time (in the number of bits of the integer) if the Extended Riemann Hypothesis is true (which remains an unsolved problem) [24]. Thus we may not have an accurate knowledge of the big-Oh running time of this primality testing algorithm if and until the Extended Riemann Hypothesis is resolved. The resolution of this problem however, is not likely imminent. [2]

We also sometimes use Ω and Ω' to describe lower bounds on **problems**. That is, we say that the problem of sorting using comparisons (i.e. less than, greater than comparisons, such as in bubble-sort) is an $\Omega(n \log n)$ **problem**. By this we mean that **any** comparison based sorting algorithm will perform at least $n \log n$ comparisons for some inputs of length n (for all n). As another example, consider the problem of searching a sorted array for a particular element. We may sometimes get lucky and find the desired element in the first or second cell of the array we look at, but in general, any algorithm will require $\Omega(\log n)$ (and thus $\Omega'(\log n)$) steps – binary search being the algorithm that actually achieves this, taking $O(\log n)$ steps:

[2]Primality test can be solved in polynomial time using the algorithm of AgrawalKayalSaxena which was discovered in 2002 [2]. However, the Miller-Rabin test is generally used in practice as it is much faster, though it does have a small chance of error.

```
input sorted array A[1..n]
input key
lo:=1
hi:=n
found:=false
while (not found) and (lo <= hi) loop
   mid := (lo + hi)/2
   if key = A[mid] then found:=true
      else if key < A[mid] then hi:=mid - 1
           else low:=mid + 1
end
print(found)
```

Definition 4 *$f(n) \in o(g(n))$ if $f(n) \in O(g(n))$ but $g(n) \notin O(f(n))$.*

In other words, we think of $g(n)$ being **much** faster growing that $f(n)$: a "loose" upper bound,

Definition 5 *We say $f(n)$ is asymptotically smaller than $g(n)$, or $f(n) << g(n)$, if $f(n) \in o(g(n)$,*

As an example, note that $2^n << n!$ (can you prove this), even though for $1 \leq 3$ $2^n > n!$.

An important rule to remember is the following:

Any positive exponential function (i.e., one with a positive base and exponent) grows faster asymptotically than any polynomial function. And any polynomial function grows faster than any logarithmic function.

For example, $1.1^n >> n^4 >> 20 \log n >> 5$.

Finally, we say $f(n) \in \Theta(g(n))$ if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$. Equivalently, if $f(n) \in O(g(n))$ and $g(n) \in \emptyset(f(n))$. That is, Θ is a "tight", a precise bound on a function; though as usual, low-order terms can be ignored. So although $7\frac{n}{\log n}$ is $O(n)$, it is better to say that $7\frac{n}{\log n}$ is $O(\frac{n}{\log n})$.
In the following table, we compare some functions.

$f(n)$	$g(n)$	$f \in O(g)$	$f \in \Omega(g)$	$f \in \Theta(g)$	$f \in o(g)$
1. $5n$	$n \log n$	Yes	No	No	Yes
2. $n + \log n$	$2n + 100$	Yes	Yes	Yes	No
3. 2^n	3^n	Yes	No	No	Yes
4. $\frac{n}{\log n}$	$\sqrt{n}$	No	Yes	No	No
5. $n * (n \bmod 10)$	n	Yes	No	No	No
6. $\frac{1}{n}$	1	Yes	No	No	Yes

Number 2 is an example of what we might call the "rule of sums," that is, our habit to ignore low order terms. In general we say that

$$f(n) = g(n) + h(n) \text{ implies } f(n) \in \text{maximum}\{g(n), h(n)\}.$$

So for example, $n^2 + n \log n \in O(n^2)$ since $n^2 >> \log n$.

But this rule of sum only applies if there are but a constant number of terms in the summation! Hence we cannot use the fact that n is the largest term in $\sum_{i=1}^{n} i$ to deduce that the sum is $O(n)$, because the number of terms is not a constant. Of course, that sum is in fact $O(n^2)$. Whereas, $\sum_{(i=n-2)}^{n} i = O(n)$, since there are only a constant number of terms in the summations.

Likewise $f(n)+g(n)$ is $\Omega(\text{maximum}\{f(n), g(n)\})$. Here we do not require that the number of terms be a constant. So $\sum_{i=1}^{n} i = \Omega(n)$. In effect, this is our "sloppy rule of sums" that is discussed in the Mathematical Preliminaries chapter. Though it more precise to say that this sum is Ωn^2.

Observe that number 5, $f(n) = n * (n \bmod 10) \in \Omega'(n)$, since n*($n$ mod 10) is at least as large as n infinitely often – though it also has the value 0 infinitely often!

Note that if $f(n) = o(n)$, then $f(n) = O(n)$. The converse is not necessarily true, as $2n^2$ is $O(n^2)$ but not $o(n^2)$. Also note that a function $f(n)$ may be neither $O(g(n))$ nor $\Omega(g(n))$, for some $g(n)$. Consider our strange case of $f(n) = n * (n \bmod 10)$. It is neither $O(\log n)$ nor $\Omega(\log n)$. Though it is $\Omega'(\log n)$.

In most cases we can use simple algebra to see which function is asymptotically larger than the other. However, this is not always the case. To determine which of two functions is greater, recall that we are dealing with the limit as $n \to \infty$. Therefore, we say $f(n) >> g(n)$ if and only if the limit as $n \to \infty$ of $\frac{f(n)}{g(n)}$ approaches ∞. Thus it may be necessary to use de l'Hopital's rule from calculus:

$$\lim n \to \infty \frac{f(n)}{g(n)} = \lim n \to \infty \frac{f'(n)}{g'(n)}$$

where $f'(n)$ denotes the derivative of $f(n)$. But this only makes sense – we are interested in the growth rate of a function as $n \to \infty$, which is nothing more than its slope [3]. Consider the following two functions: $f(n) = \sqrt{2n}$ and $g(n) = \log_{10} n^2$. From elementary calculus, we know that $f'(n) = \frac{1}{\sqrt{2n}}$ and $g'(n) = \frac{2}{n}$. Inspection reveals that $\frac{1}{\sqrt{2n}} > \frac{2}{n}$ for all $n > 8$. Furthermore, we can see that $\frac{1}{\sqrt{2n}} >> \frac{2}{n}$, since the ratio $\frac{f'(n)}{g'(n)} = \frac{n}{2\sqrt{2n}}$, which approaches infinity as n approaches infinity. Thus we can see that in general $\log^c n \in o(\sqrt[k]{n})$ for any positive c, k.

However, simple algebra and common sense usually suffice. For example, to show $\frac{n}{\log n} >> \sqrt{n} \log^2 n$ we do the following. Recall that $\log^2 n = (\log n)^2$. Putting $\frac{n}{\log n}$ over $\sqrt{n} \log^2 n$ we get $\frac{\sqrt{n}}{\log n}$ which is clearly infinite as $n \to \infty$ since any polynomial in n is asymptotically greater than any polylogarithmic function, i.e. $\log^x n$, of n, even a polynomial with positive exponent less than one (as we just showed above). That is, $\sqrt[4]{n} >> \log^3 n$.

3.2 Some Example Code Segments

We'll now give some programs and analyze the running time using big-Oh. Remember, what we are interested in is the running time of the program in terms of its input **length**, or **size**.

```
function foo(input: a: array[1..n] of integer; output: b: integer);

i : integer;

i:=1
while i < n/2 loop

if a[i] < a[2i] then
   i:=3i/2
end if;

i:=i+1;
end while;
```

[3]We may iterate de l'Hopital's rule if necessary to the second derivative, and so on

```
return a[i];

end foo;
```

Since we are concerned with the **worst-case** running time, we pessimistically assume that the "i:=3i/2" line is never executed (that is, the "if" test is never satisfied) since that would only speed us towards our goal of i getting to be at least as large as $\frac{n}{2}$. The only other place i is chanced is the "i:=i+1" line and since $i = 1$ initially, we can see that the number of iterations is roughly $\frac{n}{2}$ in the worst-case, which is $O(n)$, linear in the size of the input.

```
function bar(input: a: array[1..n] of integer; output: b: integer);

i : integer;

i:=3;
a[1]:=1;
a[2]:=1;
while i < n loop

if a[i] < 1 then
    i:=i-1;
    a[i]:=1;
else
   i:=i * 2;
end if;

end while;

return i;

end bar;
```

Again, we are interested in the worst-case running time. So we want the "i:=i-1" step to be performed as many times as possible. But note that once that step is performed, we prevent it from being performed during the next iteration of the "while" loop, due to the line "a[i]:=1". Thus in the worst-case we decrement i during every other iteration and double i during the other iterations. Thus the running time is $O(\log n)$, because if we *pair* consecutive iterations of the "while", the net effect is that "i:=2i-2" in the worst case. Thus after $2\log n$ iterations (i.e. $\log n$ pairs of iterations) we have that $i \geq 2^{\log n-1} + 1$[4]. Clearly, after $O(1)$ more iterations it will be the case that $i \geq n$ and thus the function will terminate.

Note that it somewhat strange to have a program whose running time is less than $O(n)$, that is, sub-linear. In such a program, we did not even read all the input! One could argue that in the above function "bar," the running time depends upon the language implementation. That is, how much time does computer actually spend passing the parameter "a: array[1..n]." If the array was passed by reference (i.e. only an address was passed) we can safely say that the array was not actually read by "bar" at run-time and that the running time is truly $O(\log n)$ (though we can expect the the array was read somewhere in the **main** program. In general though, one must approach sub-linear running times with a bit of caution, to ensure that one does not actually *read* the entire input.

```
p : integer;
k : integer :=0;

input p;
for i:=1 to p loop
     if p mod i = 1 then
        print(i)
        k:=k+i;
     end if;
end loop
```

It is easy to see that this program takes $O(p)$ steps, however we would not say the running time is $O(n)$. Remember, we generally use n to also denote the size of the input. In this case, the size of the input is $O(\log p)$. Since we store the integer p as a binary number, its length is on the order of $\log p$ bits. If p is small, we may use a built-in data type such as "int" or "longint" or "double" depending on the language. But if p is very large, say 10^{45634}, we would probably need a user-defined type to accommodate this – and again we would require $O(\log p)$ bits to represent n. Since the number of iterations of the **for** loop is $O(p)$ and $2^{\log p} = p$, we have that the

[4]Exercise 5 asks you to prove this

execution time is actually exponential in the input length: setting $n = \log p$, we have the running time is $O(2^n)$.

3.3 Recursive Algorithms

A *palindrome* is a string that reads the same forwards and backwards, such as "bob." The following recursive program determines whether or not the input string is a palindrome.

```
function palindrome(s : string[1..n]) : return boolean;

if s[1] != s[n] then return FALSE
   else if n-1 > 0 then return(palindrome(s[2..n-1])
        else return TRUE
end palindrome;

main:

input(s);
print(palindrome(s));
end;
```

What is the running time of the palindrome program? If we let $T(n)$ denote its running time on an input string of length n, we can express $T(n) = T(n-2) + O(1)$, since there is a recursive call on a string of length $n-2$ and the remaining work (including passing the parameter – which can be done efficiently using pointers to the first and last characters of the string) can be done in $O(1)$ time. Intuitively, we can see that the recursion will take about $\frac{n}{2}$ steps before it "bottoms out" (that is, until the string passed as parameter reaches length 1 or 2) and during each one of these steps, we spend $O(1)$ time; hence the running time is $O(n)$.

Suppose we have a similar expression: $T(n) = T(n-1) + n$. We can express this in another, more familiar form: $\sum_{i=1}^{n} i$. Thus this would have running time $O(n^2)$. $T(n) = T(n-1) + n$ expresses the running time of Selection Sort, which works as follows. Scan the array to find the smallest element, say in position i. Then swap element 1 with that in position i and recursively sort elements 2 through n.

Unlike the palindrome program, most interesting recursive programs involve more than just one recursive call. The famous Fibonacci sequence is the integer sequence 1, 1, 2, 3, 5, 8, 13, ... Suppose we compute F_n, the n^{th}

Fibonacci number recursively using the definition $F_n = F_{n-1} + F_{n-2}$ with appropriate base cases: $F_1 = 1, F_2 = 1$.

Suppose we write a program to compute the n^{th} Fibonacci number recursively (which you will do in programming assignment 4). How long will it take to run? Computing F_n in this manner involves computing F_{n-1} and F_{n-2} as distinct subproblems:

```
function fib(n: integer) return integer;

if (n=1) or (n=2) then return 1
   else return(fib(n-1) + fib(n-2))

end fib;

main:

input(n);
print(fib(n));

end;
```

We let $T(n)$ denote the time taken to compute the solution on input n. We can then write $T(n) = T(n-1) + T(n-2)$, and we let $T(1) = 1$ and $T(2) = 1$ (since the time to compute F_1 or F_2 is $O(1)$ steps).

We can try to solve this by hand, to see if a pattern emerges:

n	$T(n)$
1	1
2	1
3	2
4	3
5	5
6	8

Aha! It's just the Fibonacci numbers, and this should come as no surprise. But this does not tell us our ultimate goal, which is a big-Oh or big-Omega description of the running time of our program. We do this as follows.

Let $\phi = \frac{1+\sqrt{5}}{2}$. Using some algebra, we can see that ϕ is the positive root of the polynomial $x^2 - x - 1$; that is, $\phi^2 - \phi - 1 = 0$.

Theorem 6 $T(n) \geq \phi^n$.

Proof: By induction on n.

Base Case: $n = 0$. This is trivial, since $\phi^0 = 1$.

Inductive Hypothesis: Assume $T(n) \geq \phi^n$ and $T(n+1) \leq \phi^{n+1}$.

Inductive Step: We show that $T(n+2) \geq \phi^{n+2}$. By definition, $T(n+2) = T(n+1) + T(n)$. By the inductive hypothesis, $T(n+2) \geq \phi^{n+1} + \phi^n$. Factoring terms yields $T(n+2) \geq \phi^n(\phi + 1)$. Since ϕ is a root of $x^2 - x - 1$, it is true that $\phi^2 = \phi + 1$. Thus $\phi^n(\phi + 1) = \phi^n\phi^2$ and we conclude that $T(n+2) \geq \phi^{n+2}$. $\square$

Thus the algorithm runs in time $\Omega(1.618^n)$. Another approach is to observe that $T(n) = T(n-1) + T(n-2) < 2T(n-1) + 1$ (because $T(n-2) < T(n-1)$, since our functions are increasing functions) and proceed by induction, as exercise 6 suggests. But this bound of $O(2^n)$ is not as precise as that in Theorem 6.

We can see that this program is very slow – exponential running time in fact; programming assignment 4 asks you to compare this with a more efficient algorithm.

The form $T(n) = T(n-1) + T(n-2)$ is called a *recurrence relation*, since it describes the running time of a recursive program in terms of the recursion itself. If we wanted to be more exact, we could observe that for our program $T(n) = T(n-1) + T(n-2) + 1$ would be a more descriptive recurrence, since the "1" expresses the time for the "+" operation (we could use $O(1)$ if we chose, but the effects on the magnitude of the solution to the recurrence will be the same).

As we add terms to the recurrence, it is obviously going to increase the solution. In this case, one can prove (exercise 7) that

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

has the solution $T(n) = 2F_n - 1$, which is almost twice that of our original recurrence! To see this, look at how many times the "1" is added as the recurrence *unfolds*. For example, let us compute $T(5)$:

$$\begin{aligned}
T(5) &= T(4) + T(3) + 1 \\
&= (T(3) + T(2) + 1) + (T(2) + T(1) + 1) + 1 \\
&= ((T(2) + T(1) + 1)) + 1 + 1) + (1 + 1 + 1) + 1 \\
&= 9
\end{aligned}$$

Here is another example of a recursive program: a recursive version of binary search.

```
import java.io.*;                    // for I/O

class recbinsearch

// Note that this version does not have the binary search
// method tied to an object.

   {
   public static void main(String[] args)
      {
      int i;
      int maxSize = 100;             // array size
      int [] arr = new int[maxSize];   // create the array

      for (i=0; i < 100; i++)
             arr[i]=i;

      int searchKey = 55;            // search for item

      if(binsearch(arr, 0, arr.length-1, searchKey))
         System.out.println("Found " + searchKey);
      else
         System.out.println("Can't find " + searchKey);

      }  // end main()

    public static boolean binsearch(int [] a, int lo, int hi, int key)

// Assumes a[] is sorted
// Could re-write using Comparable rather than int!!!
// Could also make a[] global and not pass as parameter, to save time

      {
       if (lo == hi)
            return (a[lo] == key);
       else {
           int mid = (lo + hi)/2;
```

```
          if (a[mid] < key) {
             lo = mid + 1;
             return(binsearch(a, lo, hi, key));
           }
           else if (a[mid] > key) {
             hi = mid -1;
             return(binsearch(a, lo, hi, key));
           }
           else return(true);
      }
     } // end binsearch

   }  // end
```

We can express the running time of the recursive binary search (assuming negligible cost for passing the array as a parameter) as

$$T(n) = T(n/2) + O(1), T(1) = O(1)$$

That is, the depth of the recursion is at most $\log n$ and the cost incurred at each level is $O(1)$, hence the total running time is $O(\log n)$.

3.3.1 Recurrence relations

Many recursive programs use the recursion as a “divide-and-conquer” strategy to solve the problem. For example, we divide the problem into two equally sized subproblems, solve each of those independently (via recursion), and then combine the solutions to the small problems into a solution for the original problem.

A classic example of this is the MergeSort algorithm, which inputs an array of n items and sorts them in ascending order. The idea is to divide the list in half, recursively sort each half, and then “merge” the two lists together to form a sorted list. Let us first give the merging step, which inputs two sorted arrays and outputs a single sorted array containing all elements from the two input arrays.

```
function merge(A: array[1..p], B: array[1..q]) return array[1..p+q]

C: array[1..p+q];

i:=1; j:=1; k:=1;
while (i <= p) and (j <= q) loop
    if A[i] < B[j] then
       C[k]:=A[i]
       i:=i+1
    else
       C[k]:=B[j]
       j:=j+1
    end if;
    k:=k+1
end while

if i <= p then
   for m:=i to p loop
       C[k]:=A[m];
       k:=k+1;
   end for
else
   for m:=j to q loop
       C[k]:=B[m];
       k:=k+1;
   end for
end if

return C;

end merge;
```

Function merge simply compares the “front” elements of A and B and inserts the smaller one into the next position of C. It is easy to see that the running time is $O(p+q)$. Now we can write MergeSort.

```
function MergeSort(A: array[1..n]) return array[1..n]

if n=1 then return A      /* Since a one element array is sorted */
   else return(merge(MergeSort(A[1..n/2]), MergeSort(A[n/2+1 .. n])

end MergeSort;
```

Two recursive calls are made in each invocation of MergeSort – each on a list half the size of the parameter that was passed. The recursion continues deeper and deeper until the parameter is of size one and then the merging steps will begin combining sorted arrays together. If we divide the size of the list in half each time, the depth of the recursion will be $O(\log n)$. Like a binary tree, we begin with a list of length n, divide that into two lists of length $\frac{n}{2}$, divide those into a total of four lists of length $\frac{n}{4}$ and so on, until we have n "leaves" of length one each.

The running time may be expressed as the recurrence:

$$T(n) = 2T(\frac{n}{2}) + O(n), \quad T(1) = 1$$

since we make two recursive calls (in the initial call to MergeSort) on lists of length $\frac{n}{2}$ each and spend $O(n)$ time merging together the two sorted lists of length $\frac{n}{2}$ each. To solve this recurrence, we give two approaches.

The first method, "guess and prove," requires some good guessing – good guesses often come from unfolding the recurrence and seeing if an obvious pattern emerges. The second method will not require any guessing. For example: let us unfold for $n = 8$ (and assume for simplicity that $O(n) = n$).

$$\begin{aligned} T(8) &= T(4) + T(4) + 8 \\ &= (T(2) + T(2) + 4) + (T(2) + T(2) + 4) + 8 \\ &= ((T(1) + T(1) + 2) + (T(1) + T(1) + 2) + 4) + ((T(1) + T(1) + 2) + (T(1) + T(1) + 2) + 4) + \\ &= 4 + 4 + 4 + 4 + +4 + 48 \\ &= 32 \end{aligned}$$

We might guess (if we worked a few more examples) that $32 = (8 + 1)\log 8 = 8 * 3$ and that $O(n \log n)$ might be a good guess. So ...

Another way of seeing this cost as 32 is by viewing the recursion as a tree. The root of the tree has cost 8 (cost of merging two arrays of length 4 each). The next level of the tree has two nodes of cost 4 (these of the children of the root). These costs come from each node representing the cost of merging two arrays of size two each. When one completes the tree, one sees that the sum of costs on each level is 8 (more generally, the sum of the costs on each level is n) and that there are $O(\log n)$ levels, so the total running time is $O(n \log n)$.

Theorem 7 *MergeSort runs in $O(n \log n)$ time.*

Proof: We guessed that $O(n \log n)$ would be a solution to the recurrence. To prove that it is correct, we proceed by induction on n, showing that

$$T(n) = 2T(\frac{n}{2}) + n \leq cn \log n;\ T(1) = 1.$$

where c is a constants.
Base Case: $n = 1$. $T(1) = 1 \leq c$, by choosing $c \geq 1$.

Inductive Hypothesis: Assume $T(m) \leq cm \log m$, for all $m < n$.

Inductive Step: Consider $T(n)$.

$$\begin{aligned} T(n) &\leq 2T(\tfrac{n}{2}) + n \\ &\leq 2c\tfrac{n}{2} \log \tfrac{n}{2} + n \ \text{(by induction)} \\ &= cn \log n + n - 2n \ \text{(simplifying)} \\ &= cn \log n + n - 2n \\ &\leq cn \log n \end{aligned}$$

□

Note: we have ignored any problems that might occur when n is odd (or if we encounter an odd n at any time during the recursion), but this turns out to be insignificant and we shall overlook the slightly messy details.

We could also have made the $+n$ term a bit more realistic by making it dn, as in $T(n) = 2T(\frac{n}{2}) + dn$ for some constant d, but the same proof we gave can be easily adjusted to deal with this. The careful reader may wish to re-write the proof above to handle this situation.

Let's do another example using induction.

We want to prove that the recurrence $T(n) = 2T(n-1) + n, T(1) = 1$ is equal to $2^{n+1} - n - 2$. (These are the so-called Eulerian numbers, though indexed differently).

Base Case: $n = 1$. Both sides are equal to 2.

Assume $T(n) = 2^{n+1} - n - 2$.

Show $T(n+1) = 2^{n+2} - (n+1) - 2$.

Then $T(n+1) = 2T(n) + n + 1$ (by definition of the recurrence).

$= 2(2^{n+1} - n - 2) + n + 1$ (invoking the inductive hypothesis)

$= 2^{n+2} - 2n - 4 + n + 1$

$= 2^{n+2} - (n + 1) - 2$, as desired.

3.3.2 Master Theorem

We can now state a general method for solving lots of common recurrences – particularly those that describe divide-and-conquer recursive algorithms. This is the so-called *master theorem.* We state two versions, the first being a simpler and less general version before stating the full master theorem.

The simple version is from [3].

Theorem 8 *Let a, b, c be non-negative constants. The recurrence*

$$T(n) = \begin{cases} c & \text{for } n = 1 \\ aT(\frac{n}{b}) + cn & \text{for } n > 1 \end{cases}$$

Then

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

Consider $T(n) = 3T(\frac{n}{4}) + n$, where $T(1) = 1$. Since $3 < 4$, we have that the solution to this recurrence is $O(n)$. We can see that if $T(n) = 5T(\frac{n}{2}) + n$ and $T(1) = 1$, then the solution is $O(n^{log_2 5}) = O(n^{2.33})$. Whereas for MergeSort we saw that $T(n) = 2T(\frac{n}{2}) + n$ and $T(1) = 1$ implies that $T(n) = O(n \log n)$. The difference between these two recurrences is that in the first case, we divide the problem into five problems of size $\frac{n}{2}$ each and solve each (recursively) whereas in the latter we divide the problem into only two problems of size $\frac{n}{2}$ each. So it should not be surprising that the running time is slower for the first recurrence.

We now state the full version of the master theorem [11].

Theorem 9 *Let a, b, c be non-negative constants and $f(n)$ be a function. The recurrence*

$$T(n) = \begin{cases} c & \text{for } n = 1 \\ aT(\frac{n}{b}) + f(n) & \text{for } n > 1 \end{cases}$$

Then
Case 1. If $f(n) = O(n^{log_b a - \epsilon})$ for some constant $\epsilon > 0$ then $T(n) = \Theta(n^{log_b a})$.

Case 2. If $f(n) = \Theta(n^{log_b a})$ then $T(n) = \Theta(n^{log_b a} \log n)$.

Case 3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and if $af(\frac{n}{b}) \leq df(n)$ for some constant $d < 1$ and for all sufficiently large n then $T(n) = \Theta(f(n))$.

Note that saying $T(n) = \Theta(g(n))$ implies $T(n) = O(g(n))$.

Let us do a few examples, additional problems can be found in the exercises. We assume $T(1) = 1$ in all examples. Suppose

$$T(n) = 3T(\frac{n}{2}) + 2n \ .$$

In this case $f(n) = 2n, a = 3, \text{and} b = 2$. $f(n) \in O(n) \in O(n^{\log_2 3 - \epsilon})$ for $\epsilon = 0.1$. That is, since $n^{\log_2 3} \approx n^{1.58}$, then $n \in O(n^{1.48})$. Therefore Case 1 applies and $T(n) = \Theta(n^{\log_2 3})$. So we see that the "divide-and-conquer" step is the dominant step here, rather than the "combining" or "reconstitution" step!

Suppose

$$T(n) = 2T(\frac{n}{2}) + n^2$$

This recurrence looks similar to that for MergeSort, except that $f(n) = n^2$. We can see that Case 1 will not apply, since $f(n) = n^2 >> n^{\log_2 2} = n$. Likewise, Case 2 cannot apply (note that Case 2 **does** apply for the MergeSort recurrence!). Case 3 does apply to this recurrence, since $n^2 \in \Omega(n^{1.1})$ (choosing $\epsilon = 0.1$) and because

$$2(\frac{n}{2})^2 \leq 0.75n^2, \text{for all positive } n.$$

Thus we conclude that $T(n) = \Theta(n^2)$. Note that here we chose $c = 0.75$ to satisfy the additional condition.

Suppose

$$T(n) = 9T(\frac{n}{3}) + n^2 \ .$$

Case 1 does not apply because $f(n) = n^2 \notin O(n^{\log_3 9-\epsilon})$ for any $\epsilon > 0$. For example, if we choose $\epsilon = 0.01$, we have that $n^2 \notin O(n^{\log_3 9-0.01}) = O(n^{1.99})$. But we see that $f(n) = n^2 = O(n^{\log_3 9}) = O(n^2)$ and so Case 2 applies. Thus $T(n) = O(n^2 \log n)$.

We have already seen recurrences such as $T(n) = T(n-c) + f(n)$ for some c (c may be a constant or a function of n) to which the master theorem obviously cannot apply. One way to get a "loose" bound on recurrences of this form is to determine the "depth" of the recursion and multiply that by $f(n)$. So for example, we can bound $T(n) = T(n-1) + n$ by $O(n^2)$ since the depth of the recursion is $O(n)$. This happens to be a precise bound, i.e. $T(n) = \Theta(n^2)$, but this may not always be the case.

Other recurrences such as $T(n) = 2T(n-1) + 1$ cannot be bounded so easily and need to be attacked in other ways (see Exercise 6). Yet many recurrences, even those of the proper divide-and-conquer format, i.e. $T(n) = aT(\frac{n}{b}) + f(n)$, cannot be solved using the master theorem. For example, suppose we have a divide-and-conquer type recurrence

$$T(n) = 4T(\frac{n}{2}) + n^2 \log n \ .$$

then $f(n) = n^2 \log n$ and $a = 4, b = 2$, so $n^{\log_b a} = n^2$. Since $n^2 \log n >> n^2$, Case 1 and Case 2 cannot apply. Checking case 3, we test whether $n^2 \log n = \Omega(n^{2+\epsilon})$ for some constant $\epsilon > 0$. For example, is $n^2 \log n = \Omega(n^{2.001})$. The answer is no, because $n^{2+\epsilon} >> n^2 \log n$ for any positive epsilon: any polynomial function with positive exponent dominates any polylogarithmic function. Thus the master theorem does not apply. How might we solve such a recurrence? We could use more advanced methods such as discussed in [33] or settle for a "loose bound." That is, we can see that

$$T(n) = 4T(\frac{n}{2}) + n^2 \log n << 4T'(\frac{n}{2}) + n^{2.1}$$

Using the master theorem, Case 3 applies, since $f(n) = n^{2.1} \in \Omega(n^2)$ and

$$4(\frac{n}{4})^2 \leq 0.75n^{2.1}, \text{for all positive } n.$$

Hence we may conclude $T'(n) = \Theta(n^{2.1})$ and $T(n) = O(n^{2.1})$

Consider the following program for finding the maximum element in an array. [5]

[5]The reader may wish to implement this and compare its running time with the iterative method that uses a "for" loop.

```
function maximum(a : array[1..n]) return integer

if n=1 then return a[1] else
   m1 := maximum(a[1../n2]);       // find maximum of first half
   m2 := maximum(a[n/2 + 1 .. n]); // find maximum of second half
   if m1 < m2 then return m2
      else return m1;             // return larger of two maximums
   end if;
end if;

end maximum;

main:

  input(A[1..n]);
  max:=maximum(A);

end main;
```

The running time of this maximum-finding algorithm is described by

$$T(n) = 2T(\frac{n}{2}) + 1$$

since we divide the array in half and recursively process each half. The time to compare the two maximums returned from the recursive calls is $O(1)$. Since $f(n) = O(1) \in O(n^{log_2 2-\epsilon})$ for $\epsilon = 0.5$ (i.e., $O(1) \in O(\sqrt{n})$) Case 1 of the master theorem applies and we conclude the running time is $O(n^{log_2 2}) = O(n)$.

We shall see more recursive programs in subsequent chapters and use the Master Theorem to analyze many of them.

The Master Theorem cannot be used to give precise bounds on all divide-and-conquer type recurrences. Consider $T(n) = 2T(n/2) + \frac{n}{\log n}$. None of the three cases apply, as $\frac{n}{\log n}$ is not polynomially smaller than n, though it is smaller. We can however, say that in this case $T(n) < 2T(n/2) + n$ and conclude that the recurrence is $O(n \log n)$.

3.4 Exercises

1. Find the smallest positive n such that $\sqrt[4]{n} > \log^3 n$. (Hint: use trial and error).

2. Is $2^{n+1} \in O(2^n)$? Is $n * (n \bmod 10) \in O(n)$?

3. Rank the following functions from slowest to fastest growing:

$$2^{\log n} \quad n^n \quad n^3 - 20n^2 \quad \log\log^2 n \quad n^{\log 5} \quad n\log n \quad 2^{2n} \quad 75 \quad \sqrt{n^3}$$

Hint: What is simpler way to express $2^{\log n}$?

4. Complete the following table:

$f(n)$	$g(n)$	$f(n) \in O(g(n))$	$f \in \Omega(g)$	$f \in \Theta(g)$	$f \in o(g)$
$\frac{n^2}{100} + n\log n$	$\frac{n^2}{\log n}$				
$\log\log n$	$\log n - 7$				
$\frac{1}{n}$	25				
$n + n^3$	$n^2\log n - n$				
$\frac{n}{\sqrt{n}} + \log n$	$5n + 10$				

5. Prove the following outputs the value $2^k + 2$, by using induction on k.

```
input(k);
i:=3;
for j:= 1 to k loop
    i:=2i - 2;
end loop;
print(k);
```

What is output if we replace the line "i:=2i-2" with "i:=2i" ?

6. Let $T(n) = T(n-1) + T(n-2)$ and $T'(n) = 2T'(n-1) + 1$, where $T(1) = 1, T(2) = 1, T'(1) = 1$. Note that $T(n) < T'(n)$. Prove by induction on n that $T'(n) = 2^n - 1$.[6]

7. Prove that the recurrence

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

has the solution $T(n) = 2F_n - 1$.

[6] $T'(n)$ is the recurrence for the algorithm for the famous "Towers of Hanoi" problem [25].

8. Solve the following recurrences (give a big-Oh solution). Assume $T(1) = 1$ in all cases.

a) $T(n) = T(n-1) + \log n$

b) $T(n) = T(\frac{n}{3}) + n$

c) $T(n) = T(n-2) + n$

d) $T(n) = T(\frac{3n}{4}) + 1$. (Solve this using the Master theorem and try solving it by analyzing the depth of the recursion).

e) $T(n) = T(n) + n^2$

f) $T(n) = 2T(n-1) + 1$. Prove this is $O(2^n)$. (this recurrence describes the running time for an algorithm to solve the famous Towers of Hanoi problem).

Bonus $T(n) = T(n-1) + T(\frac{n}{2}) + n$.

Bonus $T(n) = 2T(n-1) + n$. [try unfolding]

Bonus $T(n) = 2T(n-1) + n^2$

[11] Bonus $T(n) = T(\sqrt{n}) + 1$.

9. Solve the following recurrences using the Master Theorem (give a big-Oh or Theta solution). If none of the cases apply, provide a "loose" bound by substituting another function for $f(n)$. Assume $T(1) = 1$ in all cases.

a) $T(n) = T(\frac{n}{3}) + \log n$

b) $T(n) = 6T(\frac{n}{3}) + n \log n$

c) $T(n) = 8T(\frac{3n}{2}) + n^2$

d) $T(n) = 3T(\frac{n}{2}) + n^2$

10. Analyze the running times of the following "programs" (first find a recurrence describing the running time of the program then solve the recurrence.)

a)

```
function foo(input: a: array[1..n] of integer) return integer;

if n=1 then return a[1];
min:=a[1];
for i:=1 to n loop
  if a[i] < min then
     min:=a[i];
     swap(a[1], a[i]);
  end if;
end loop;

return foo(a[2..n]);
end;

main:

  input(a[1..n]);
  x:=foo(a);

end;
```

What is output by this program?

b)

```
function foo(input: a: array[1..n] of integer) return array[1..n] of integer;

if n=1 then return a[1];
a:=MergeSort(a);

a:=concatenate(foo(a[n/2+1..n]), foo(a[1..n/2]));

-- Assume concatenation of two arrays takes O(1) time

end foo;

main:

  input(a[1..n]);
  a:=foo(a);

end;
```

c) [Bonus]

```
function foo(input: a: array[1..n] of integer) integer;

if n=1 then return a[1];
a:=MergeSort(a);

for i:=1 to n loop
  x:=x + foo(a[1..i])
end for;

end foo;

main:

  input(a[1..n]);
  x:=foo(a);

end;
```

Implement this program and see what happens!

11. Show $\sum_{i=1}^{\log n} \log i$ is $\Omega(\log n)$. Is it $\Omega(n)$?

12. Do there exist $f(n), g(n)$ such that $f(n) \notin O(g(n))$ but $\log f(n) \in O(\log g(n))$?

13. Consider the recurrence $T(n) = nT(n/2)+1$. Draw a recursion tree and show that this sum is greater than or equal to $\prod_{i=0}^{i=\log n} \frac{n}{2^i}$. Then show that this product is equal to $n^{\log n}/2^{(\log n)(\log n+1)/2}$ and then show this is equal to $n^{\frac{\log n}{2}} n^{\frac{1}{2}}$.

14. Solve $T(1) = 1, T(n) = 2^n T(n-1)$.

15. Use induction to show $T(1) = 1, T(n) = 2^n T(n-1) + 1 \le 2^{(n+1)^2/2}$.

16. Solve the recurrence $T(n) = 2T(n-1) + n$.

17. Describe an algorithm that inputs two n-digit numbers and uses the algorithm you learned in grammar school to compute their product. What is the running time of this algorithm?

Bonus: Show that a solution for the recurrence: $T(n) = T(\sqrt{n}) + 1$, is $\Theta(\log \log n)$. Hint: Show that $\log \log n$ is an upper bound on the depth of the recursion [36].

Bonus: Euclid's algorithm for finding the greatest common divisor (gcd) of two positive integers is one of the oldest numerical algorithms. It is naturally recursive in nature, very fast, and based on the following idea. If $a > b$, then $\gcd(a, b)$=$\gcd(b, a \bmod b)$. Write pseudo-code for this algorithm (you need to determine the termination condition for the recursion) and analyze the running time.

3.5 Programming Assignments

1. Input an integer x and an integer k, both positive. Write a program to compute x^k. You should be able to do this easily in $O(k)$ steps – except for the fact that you will likely get an "overflow" error if x and k are very big at all (Try it and see!). Compute x^k modulo p in $O(k)$ steps (by modulo p, we mean the remainder when x^k is divided by p). Can you avoid overflow errors in this case?

Bonus: Compute x^k modulo p in $O(\log k)$ steps.

2. Write a program to compute F_n, the n^{th} *Fibonacci number* in two different ways. First compute F_n recursively using the definition $F_n = F_{n-1} + F_{n-2}$ with appropriate base cases. Then compute F_n iteratively, using a single "for" loop. Compare the running times of the two programs for $n = 10, 20, 50, 100$. [7]
3. Implement the MergeSort and BubbleSort programs discussed in this chapter. Compare their running times on 100 input arrays of size 100; 100 arrays of size 1000; and 100 arrays of size 10,000. Randomly generate the numbers in each input array.

[7] You can also compute F_n using the program from programming assignment 1 and the fact that $F_n = \frac{\phi^n}{\sqrt{5}}$ rounded to the nearest integer, where $\phi = \frac{1+\sqrt{5}}{2}$.

Chapter 4

Sorting, Selection, and Array Problems

4.1 Searching

Let A be an array with n elements and *key* a value we wish to search for. That is, we want to know whether or not any element of A is equal to *key*. The simplest way to perform this this is to sequentially compare each of the n elements of A to *key*. This takes $O(n)$ steps in the worst case – the worst case being when $key = A[n]$ or if *key* is not in the array. Of course, if *key* is in the array, we would expect, on average, that $\frac{n}{2}$ comparisons would suffice. Note that sequential search does not require that A be sorted in any order. If A is sorted, we can achieve much faster search results, as described below.

In the previous chapter we showed the classic iterative binary search procedure. Given an array of length n and a key value; the binary search procedure makes at most $O(\log n)$ comparisons to either find the key or determine that the key is not in the array. Let us now recall the recursive version of binary search:

```
function binary_search(a : array[1..n], key) return boolean

-- a is a sorted array, sorted in ascending order

-- returns true if the key is found

if n=1 then return(a[1] = key)
   else
      if key < a[n/2] then return(binary_search(a[n/2+1..n], key)
         else if key > a[n/2] then return(binary_search(a[1..n/2-1], key)
                 else return true;
              end if;
      end if;
end if;

end binary_search;

main:

  input(A[1..n]);
  input(key);
  found:=binary_search(A, key);

end main;
```

We can now analyze this using recurrence relations. It is easy to see from the code that the recurrence $T(n) = T(\frac{n}{2}) + O(1), T(1) = 1$ describes its running time. Using induction we see that the running time is $O(\log n)$, in the worst case:

Proof: The base case is trivial. Assume for the inductive hypothesis that $T(\frac{n}{2}) \leq c \log \frac{n}{2}$ for some constant c. Now substituting we have that

$$T(n) = T(\frac{n}{2}) + O(1) \leq c \log \frac{n}{2} + d$$

where $d = O(1)$. Simplifying, we get

$$T(n) \leq c \log n - c \log 2 + d$$

$$T(n) =\leq c \log n - c + d$$

which implies

$$T(n) = \leq c \log n$$

since we can choose $c \geq d$. $\square$

Note that we could have used the Master Theorem to arrive at the same conclusion.

Suppose we wish to search an array with n elements for both the maximum and minimum elements. Obviously, two sequential searches can accomplish this with no more than $2n-2$ comparisons (since, for example, the maximum can be found with $n-1$ comparisons by initializing a variable to the first element in the array and then doing comparisons with elements 2 through n).

However, we can accomplish this task with only $\frac{3n}{2}$ comparisons as follows. Rather than considering elements from the array one at a time, we consider then two at a time. We first find the smaller of the two elements and compare that with *min* (the current candidate for the minimum element from the array) and then compare the larger of the two with *max* (the current candidate for the minimum element from the array).

The algorithm in detail:

```
procedure max_and_min

a : array[1..n] of integer;
max : integer;
min : integer;
m1, m2 : integer;

max:=a[1];
min:=a[1];


for i:=2 to n-1 step 2 do begin  -- i increases by 2 each iteration

     if a[i] <= a[i+1] then
        m1:=a[i];
        m2:=a[i+1];
     else
        m1:=a[i+1];
        m2:=a[i];
     end if;

     if m1 < min then
        min:=m1;
     end if;

     if m2 > max then
        max:=m2;
     end if;
end for;

-- make sure we check the last element if we did not do so already

if n mod 2 = 0 then
   if a[n] > max then
      max:= a[n];
   else
      if a[n] < min then
         min := a[n];
   end if;
end if;
```

The "for" loop iterates no more than $\frac{n}{2}$ times and each iteration contains three comparisons, hence the total number of comparisons is at most $\frac{3n}{2}$.

4.2 Sorting

We have already seen MergeSort and BubbleSort; the former requiring $O(n \log n)$ steps in the worst case and the latter $O(n^2)$ steps in the worst case to sort an array with n elements.

4.2.1 QuickSort

QuickSort is one of the most widely used sorting algorithms in practice, for example the UNIX program **qsort** is an implementation of QuickSort. Though a worst-case $O(n^2)$ time algorithm, as we mentioned above, QuickSort runs on average in $O(n \log n)$ steps. That is, if we average the number of steps taken over all possible inputs (permutations) of length n, the average number of comparisons is about $1.39 n \log n$, which compares favorably to other sorting routines; the instances which take $\Theta(n^2)$ steps are rare.

Though the number of comparisons taken by MergeSort is about $n \log n$ (slightly less than QuickSort), the actual number of operations in practice often shows QuickSort to be a bit faster (see, e.g., [32]). MergeSort does have the advantage of being stable (i.e. elements with equal keys keep their relative ordering in the final sorted list), but does require a bit more memory than QuickSort.

QuickSort is somewhat similar in spirit to MergeSort: it is a recursive algorithm. However, we will not incur the expense of merging. Rather, the idea is to partition the array into two parts: one containing all the "small" elements and one containing all the "large" elements and then recursively sort each part. This partitioning, or "shuffling" is done by choosing a *pivot* element (typically at random to approximate the median). We then move all the elements less than the pivot to the left part of the array and all the elements greater than the pivot to the right part of the array. This is accomplished by maintaining two pointers, one that starts at the left end of the array (which moves to the right) and one at the right end (which moves to the left). When the left pointer finds an element greater than the pivot, it stops. When the right pointer finds an element less than the pivot, it stops. The two elements are then swapped, and the process continues until the two pointers meet. This is the method used below, slight variations exist in different QuickSort implementations.

Since all the elements in the small part of less than each element in the large part, no merging is necessary. Ideally, we would like to be able to partition the array quickly and into equal sized parts: otherwise the recursion is not "balanced" and execution will be slowed – in the worst case leading to a running time described by the recurrence $T(n) = T(n-1) + O(n)$, which of course is $O(n^2)$. The "shuffle" procedure in the code below does the actual task of moving the small elements of the array to the left side of the array and the large elements of the array to the right side of the array.

There are several methods available to choose a *pivot* element, the element such that all the values less than or equal to the pivot are placed in the "small" part and all the elements greater than the pivot are placed in the "large" part. We now give the basic QuickSort algorithm, which uses a simple method of choosing a pivot: choose an random element of the array.

```
a : array[1..n] of integer;  -- other data types are possible

function quicksort(b : array[1..m]) return array[1..m]

-- For efficient implementation, the array b should be passed either
-- by reference or by passing the two indices corresponding to
-- the "endpoints" of the array

pivot : integer;
k : integer;

begin

   k:=random(1, m);     -- choose an element of b at random
   pivot:=b[k];         -- k is index of pivot

   -- Shuffle: re-arranges b so that all elements less than or equal to pivot
   -- precede elements larger than pivot.
   -- Let pivot be in position k after this re-arranging

   shuffle(b, pivot, k);  -- k will be changed when this returns

   quicksort(b[1..k-1]);  -- Note that b[k] is in its proper place already
   quicksort(b[k+1..m]);

   return b;

end;
```

```
procedure shuffle(b : array[1..m], pivot, k)

-- b and k are call-by-reference parameters

lo, hi, temp : integer;

begin

    lo:=0;
    hi:=m;

    temp := b[1];   -- Move pivot to front, out of the way
    b[1]:=pivot;
    b[k]:=temp;

    while lo < hi loop
       repeat
          hi:=hi-1;            -- find element in upper part of array less than pivot
       until b[hi] <= pivot;

       repeat
          lo:=lo + 1;          -- find element in lower part of array greater than pivot
       until b[lo] >= pivot


       if lo < hi then begin   -- found two misplaced elements; swap them

          temp:=b[lo]
          b[lo]:=b[hi];
          b[hi]:=temp

       end if;

    end loop;

    if pivot=b[lo] then k:=lo else k:=hi; --- locate where the pivot is at the end of shu
    b[k] := pivot;                        --- Move pivot to "middle"

end;
```

```
main:

   input a;
   a:=quicksort(a);
```

Exercise 1 of this chapter asks you to execute shuffle by hand to see that this procedure in fact re-arranges the array properly into "small" and "large" parts. The basic idea of shuffle is that we use the "lo" and "hi" pointers to find two elements that are each in the "wrong" part of the array (relative to the pivot) and then swap their positions. Once "lo" exceeds "hi", each element must be in the proper part of the array.

Of course, choosing a random element of the array does not guarantee that the array is partitioned into two parts of nearly equal size. Another method of choosing the pivot is to choose three elements at random from the array and let the median of these three elements be the pivot. It is likely that this choice of pivot will result in a more balanced partitioning of the array (of course, it is still not guaranteed to do so), though at the expense of a few additional operations to find this median-of-three. The key to the good performance of QuickSort in practice is that it makes a reasonably balanced partitioning "most" of the time and that choosing a pivot and doing the partitioning can be done efficiently.

Further details on QuickSort, especially some valuable implementation details and analysis, can be found in [32].

4.2.2 Heaps and Priority Queues

A priority queue is a data structure that supports insertion and delete-maximum operations (the latter removes the maximum element from the queue). Sometimes we also want to allow an update operation that changes the value of an element in the queue or a delete operation.

We can use an unordered linked list or array and do insertion in $O(1)$ time and delete-maximum in $O(n)$ time. Using a sorted linked list or array, we can do insertion in $O(n)$ time and delete-maximum in $O(1)$ time.

A heap is a binary tree such that each node is greater than or equal to its children. (A min-heap would be just the opposite). It is also a balanced binary tree, so the leaves are all on the last one or two levels of the tree. As such, the height of a heap on n nodes is $O(\log n)$. A heap can be built in $O(n)$ time and insertion, delete-maximum and update take $O(\log n)$ time.

Here is some code to implement a heap priority queue.

```
Let A be heap with n nodes stored in Array A[1..n] left child of
A[i] stored at A[2i] right child of A[i] stored at A[2i+1]. This is
a min heap, with the minimum value at A[1]

Heapify(A, i)  // utility routine to percolate down from index i

{

  left = 2i; r = 2i+1;       // left and right children

// find smallest child, if any less than A[i]

  if left <= n and A[l] < A[i] then
     min = left             // save index of smaller child
     else min = i;
  if r <=n and A[r] < A[min] then
     min = r;           // save index of smaller child

// swap and percolate, if necessary

  if min != i then {
      swap(A[i], A[min])   // exchange values at two indices
      Heapify(A, min)
     }

}

Make_heap(A)    // create a heap from unordered array

{

// Work from the leaf level up. At each node, push node i down using
// Heapify to the correct location. After processing node i,
// subtree rooted at i will be a heap

for i=n/2 downto 1 {
   Heapify(A, i);

} // for
```

```
}

Insert(A, key) // insert new node with key value = key

// Parent of node i is at node i/2 ... assume i/2 = floor(i/2)

// Insert node at end of array, i.e., after last existing leaf
// then percolate it UP the heap

{

   n++;
   i = n;
   while i > 1 and A[i/2] > key {     // percolate up
      A[i] = A[i/2];
      i = i/2;
   }
   A[i] = key;

}

Delete_root(A)   // remove node with minimum value

{

if n < 1 then print("error") else{
   min = A[1];
   A[1] = A[n];    // replace root with last element in heap
   n--;            // reduce heap size
   Heapify(A, 1);  // percolate new root downwards
   return min;
} // else }


Update(i, new_key)  // decrease value of key at index i

{

A[i] = new_key;   // assuming this is less than old key value while
i > 1 and A[i/2] > new_key  // percolate up
```

```
  swap(A[i], A[i/2]);
  i=i/2;
}

}
```

Since the height of a heap is $O(\log n)$, insert and deletemax operations can be done in $O(\log n)$ time. It is trivial to see that a heap can be built in $O(n \log n)$ time (consider n insert operations, each taking $O(\log n)$ time), but a more sophisticated analysis of the Makeheap() procedure above shows one can be built in $O(n)$ time.

Numerous other implementations of priority queues are known, such as Fibonacci heaps, with different performance characteristics. Other priority queues with useful properties include binomial heaps, leftist heaps, pairing heaps, thin heaps, and Brodal queues.
BONUS: summarize the performance of all the priority queues mentioned above.

4.2.3 HeapSort

Heapsort is an algorithm that runs in $O(n \log n)$ in the worst-case, based on the heap data structure. A heap is a binary tree such that each node's value is greater than or equal to the value of its children. Heapsort is sometimes faster in practice than MergeSort, since it does not incur the overhead associated with recursion. MergeSort, however, often uses the cache more efficiently than heapsort. Heapsort requires $O(1)$ extra space, in addition to the array to be sorted, whereas MergeSort requires $O(n)$ extra space.

Various analyses of heapsort have been done. Assuming we use Floyd's $O(n)$ algorithm to build a heap (which takes $2n$ comparisons in the worst-case and about $1.88n$ on average [9], the heapsort algorithm takes about $2n \log n$ comparisons in the worst case (minus some low order terms) and about the same on average [29]. Slightly better ways to build a heap are known [9].

The variant known as bottom-up heapsort due to McDiarmid and Reed requires at most $1.5n \log n - 0.4n$ comparisons, cf. [13]. A heapsort variant due to Carlsson uses only $n \log n + n \log \log n - 0.82n$ comparisons in the worst case and another variant of his uses $n \log n + 0.67n + O(\log n)$ comparisons on average [9]. Dutton introduced the weak-heap sort (which relaxes the requirement of the data structure being in heap order) and creates a sorting algorithm using $(n-1)\log n + 0.086013n$ comparisons in the worst case [13]. The weak-heap data structure does require n additional bits of data to store n keys.

Implementations of the basic heapsort algorithm tend to be about twice as slow as quicksort in practice [29]. MergeSort generally is more efficient in terms of cache performance than heapsort (unless the cache is very small or very large), though MergeSort requires more memory than heapsort, but MergeSort is stable. Whether MergeSort or heapsort is better in practice would depend on a number of application-specific factors such as this. Some variations of weak-heap sort seem competitive with quicksort in practice.

The basic heapsort algorithm is as follows and was adapted from http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/heap/heapen.htm

```
public class HeapSortAlg {
    int[] a; // keep it quasi-global to avoid passing too many parameters
    int n;

    public static void sort(int[] array_to_be_sorted)
    {
        a=array_to_be_sorted;
        n=a.length;
        heapsort();
    }

    private static void heapsort()
    {
        buildheap();
        while (n>1)
        {
            n=n-1;
            swap (0, n);  // put max element at end
            siftdown (0);
        }
    }

    private static void buildheap()
    {
        for (int v=n/2-1; v>=0; v--) // go up the tree
            siftdown (v);            // sift down
    }

    private static void siftdown(int v) // this is called Heapify in
    // the heap program above, various texts use each name
    {
        int w=2*v+1;    // first child of v...array starts at 0
        while (w<n)
        {
            if (w+1<n)    // is there a second child of v
                if (a[w+1]>a[w]) w++;  //identify larger child

// In the sorting phase, Floyd conceived of an improvement to the
//following. Don't do this next comparison, since a[v] is likely
//smaller than a[w]. Then once you get to the bottom level, sift a[v]
//back up if necessary. This avoids some comparisons on average

            if (a[v]>=a[w]) return; // compare with larger child (*)
               else
               swap(v, w);  // heap violation, sift down
               v=w;
               w=2*v+1;    // find child of new node of w

        }
    }

    private static void swap(int i, int j)
    {
        int t=a[i];
        a[i]=a[j];
        a[j]=t;
    }

}    // end class HeapSortAlg
```

Let us explain Floyd's improvement to the heapsort algorithm that reduces the number of comparisons to a shade less than $n \log n$ and is as follows. During the sorting phase, when an element is re-inserted at the root of the heap, rather than sifting the element down to its correct place (which requires the comparison at step (*) above, simply sift the node down to the leaf level. Then sift it up to its correct position in the heap.

We note that heapsort is not a stable sorting algorithm, but it does require less memory than MergeSort.

4.2.4 Linear Time Sorting

In certain special circumstances, we are able to sort an array in linear time. Let us discuss two examples of this.

Suppose we wish to sort an array of n integers, each of which is in the range $1 \ldots c$, where c is a **constant**. That is, c is fixed and independent of n. For example, we may have $c = 100$.

Since c is fixed (and "small") and since the array to be sorted contains elements of a discrete type (such as integers) we can declare an auxiliary array, *count*, with index range $1 \ldots c$ to count how many elements of the array are of valued i, $1 \leq i \leq c$:

```
c : constant;
a : array[1..n] of integer range 1..c;
count : array[1..c];
i, j : integer;

-- now we sort a using counting sort

for i:=1 to c do begin
   count[i]:=0;          (*)
end;

for i:=1 to n do begin
   count[a[i]]:=count[a[i]]+1;
end;

for i:=1 to c do begin
   for j:=1 to count[i] do begin
       print(i);         (**)          -- print value i out count[i] times
   end;
end;
```

Note that if c was not a fixed constant, then steps (*) and (**) would require time dependent on n. But since c is a constant, we can see that the number of steps taken by the algorithm is at most $2c + 2n = O(n)$, since step (**) is executed n times. There may be some i's such that $count[i] = 0$, so we must allow that the cost of the final "for" loop is as much as $n + c$.

A *stable* counting sort is described in [11]. A sorting algorithm is stable if records with equal valued keys appear in the sorted list in the same order as in the original list. This is useful when, for example, we first sort an array of records on a secondary key (such as age) and then sort on the primary key (such as name). In this example then, the people named "John Smith" would appear in the sorted list in increasing order of age.

Radix sort is designed to sort n numbers, each with d digits (left padded with zeroes if necessary) or equivalently, strings having at most d characters each. The idea is to make d passes over the data, examining the i^{th} least significant digit on pass i. For example, suppose we have the numbers 26, 31, 46, and 17. We have maintain 10 lists (since there are 10 possible values for a digit). We insert items into these lists (imagine each list is a queue or a linked list) based on the i^{th} least significant digit of each. So after pass 1, we have 26 and 46 in the “6” list, 31 in the “1” list and 17 in the “7” list. Re-arrange these by outputting the items in the lists (processing the lists from 0 to 9) and we have the list 31, 26, 46, 17. Now clear the lists and pass over the second least significant digit and we put 31 into the “3” list, 26 into the “2” list, 46 into the “4” list and 17 into the “1” list. Then we output 17, 26, 31, 46.

As a final note on sorting, Fredman and Willard discovered a fairly general-purpose sorting algorithm that runs asymptotically faster than $O(n \log n)$ (though they make some additional assumptions about the size of the numbers being sorted) in “Surpassing the Information Theoretic Bound with Fusion Trees *Journal of Computer and System Sciences*, 47(3), pp. 424-436, December 1993.

Exercises

Problem 1. (a) Given a set of n numbers and a number x, find two distinct elements from the set whose sum is x. Solve in $O(n \log n)$ time.
(b) Given a set of n numbers and a number x, find three distinct elements from the set whose sum is x. Solve in $O(n \log^2 n)$ time, bonus points if you can solve in $O(n^2)$ time.
(c) Given a set of n numbers and a number x, find four distinct elements from the set whose sum is x. Solve in $O(n \log^2 n)$ time.

Problem 2. Given two arrays of length n, each containing integers between 1 and $2n$, determine if they have a non-empty intersection in $O(n)$ time.

4.3 Finding the Median

How do we find the k^{th} largest (or smallest) element in an array? We could find the largest, then the second largest, and so on. This is fine if k is small, but $O(n^2)$ if k approaches n. We could sort the array and choose the k^{th} element, taking $On \log n$ time.

We present an $O(n)$ algorithm that is similar to QuickSort. More specifically, we describe a linear-time algorithm for finding the k^{th} largest element of an array for any k. We describe a solution devised by Blum, Floyd, Pratt, Rivest, and Tarjan. The algorithm is like quicksort: partition around a pivot and proceed recursively. For the pivot, we'll use something called the median-of-medians.

Choosing pivot:

(1) Arrange elements in groups of five.
(2)Call each group S[i], with i ranging from 1 to n/5.
(3) Find the median of each group (e.g., by sorting each group of five). This can be done with as few as 6 comparisons per group, so O(n) time total (it is linear time because 6 is a constant).
(4)Find the median of the just found medians; this step takes time T(n/5).
(5)Let M be this median of medians. Use M to partition the input and call the algorithm recursively on one of the partitions.

It is not hard to see that we discard at least 25 percent of the items from call to call: for example, we throw away the smallest 3 items from each group whose median is less than M and the largest 3 items from each group whose median is greater than M.

```
select(L,k)
{

if (L has 50 or fewer elements) {
    sort L
    return element in the kth position
}

partition L into subsets n/5 groups S[i] of five elements each

for (i = 1 to n/5) do
    x[i] = median(S[i]) // sorting S[i] is one way to do this

M = select({x[i]}, n/10)  // n/10 is half of n/5

partition L into three parts: L1<M, L2=M, L3>M

if (k <= length(L1))
```

```
    return select(L1,k)
else if (k > length(L1)+length(L2))
       return select(L3,k-length(L1)-length(L2))
else return M

}
```

Then clearly $T(n) \leq O(n) + T(n/5) + T(3n/4)$. The $O(n)$ terms from choosing the medians of each group and partitioning the array around M. One can prove by induction that this is $O(n)$, mainly because $0.2+0.75 < 1$.

A randomized version of this chooses M randomly, rather than via recursive call: this will be fast in practice, though with the chance of running slowly if you are unlucky.

Recall Problem 1 from Chapter 2, where you were asked to determine if an array contained a majority element. Now we can solve Problem 1 from Chapter 2 in $O(n)$ time. First find the median of the array, and then scan the array testing how often the median occurs in that array. If it occurs more than half the time it is a majority element, else the array contains no majority element. In other words, if a majority elements exists, it must equal the median.

4.4 More Array Problems

This problem is from [11]. Suppose an array A[1..n] with n elements contains each number from 0 to n, except for one number. How do we find the missing number? We can use a variation on counting sort to do it in $O(n)$ time. However, suppose we can only access one bit at a time. That is, we can only ask "is the j^{th} bit of $A[i]$ equal to 1? I claim we can still solve it in $O(n)$ time (thereby not even reading all $n \log n$ bits of the input). As a hint, the solution can be described by the recurrence $T(n) = T(n/2) + O(n)$.

4.5 String Matching

An important problem in text processing is that of taking a pattern of text $y = b_1 b_2 \ldots b_l$ and locating the first occurrence of y in a text string (or file) $x = a_1a_2 \ldots a_n$. The naive algorithm–simply seeing, for all i, if the text string y begins at x_i, requires $O(nl)$ time. This is an example of a brute force or exhaustive search approach in which all possibilities are tested, albeit one that runs in polynomial time.

```
input x  (length n)
input y  (length l)
match = -1;  // set to start
             index of matching substring if we find it found = true

for i=1 to n-l+1     \\try all start positions in x
     found = true;
     for j=1 to l        \\ try to match y with x[i...i+l-1]
          if x[i+j-1] != y[j]  then {
             found = false; break;
           }
     end
     if found then {
        match=i; break;
     }
end;
print(match);
```

Nest we show an $O(l+n)$ time algorithm, due to Knuth-Morris-Pratt.

We begin by computing a *failure function* f, for the pattern y. This function (actually a simple table) will essentially tell us how far we have to "back up" in the pattern when we have a mismatch. Specifically, $f(j)$ is the largest s less than j such that $b_1b_2\ldots b_s$ is a suffix of $b_1b_2\ldots b_j$. That is, $f(j)$ is the largest $s<j$ such that $b_1b_2\ldots b_s = b_{j-s+1}b_{j-s+2}\ldots b_j$. If there is no such s, then $f(j)=0$.

For example suppose y=aabbaab. Length(y)=7 and

```
i      0  1   2   3   4   5   6   7
f(i)   0  0   1   0   0   1   2   3
```

For instance, f(6)=2 since the first 2 characters of y are the same as characters 5 and 6. That is, as is the longest proper prefix of aabbaa that is a suffix of aabbaa.

We use the failure function as follows to find y in x. We scan x, comparing the characters of x with those of the pattern y. When a match occurs, we look at the next character of x and the next character of y. When a mismatch occurs, rather than back up in x, which would lead to a non-linear time algorithm, we simply use the failure function to determine which character of the pattern to compare with the next character of the text.

For example, suppose $y = aabbaab$ and $x = abaabaabbaab$, the algorithm described would behave as follows. Since the pattern has 7 characters, there are eight states of the algorithm, corresponding to the number of matched characters in the pattern, 0-7.

```
Input:   a b a a b a a b b a a b
State   0 1 0 1 2 3 1 2 3 4 5 6 7
```

Initially we are in state 0, no matched characters. After reading the first characters of x and y, we enter state 1 (1 matched character). On reading the second character of x and y, there is a mismatch. Thus we return to state 0, since $f(1)=0$. We next compare the second character of x to the first character of y (on a mis-match, we stay on the same character of x for one comparison). We then have another mis-match (2nd of x which is b which does not equal the first of y which is an a). By definition, $f(0)=0$. We then look at the first character of y and the third of x, which is a match and thus move to state 1 and so on.

This algorithm is clearly $O(N)$, since we evaluate each character of x at most two times, we are always moving forward in the text. As another example: suppose $y = aabbaab$ and $x = aabaabbaab$. We start with

aabbaab
aabbaaaaabbaab

and have a mismatch on the seventh character. So then we can make the following alignment of the strings:

(align beginning aa of pattern with characters 5, 6 of text)

That has a mismatch on the third character of y (still the seventh character of x), so we make the following alignment:

(align beginning aa of pattern with characters 8, 9 of text)

We now look at the computation of the failure function, which will take $O(l)$ time.
Of course, f(0)=0 and f(1)=0.
The rest of the table is built incrementally.

```
for j:=2 to l do begin
          i:=f(j-1);
          while (bj != bi+1) and (i > 0) do
              i:=f(i);
          if (bj != bi+1 ) and (i=0) then f(j):=0
              else f(j):=i+1;
end;
```

We now show and example how the above is built.
For example suppose $y = aabbaab$. Length(y)=7 and

i	0	1	2	3	4	5	6	7
f(i)	0	0	1	0	0	1	2	3

To see that this is $O(l)$, only the while loop needs to be considered, as each other statement in the for loop is $O(1)$ time. The cost of the while loop is proportional to the number of times i is decremented by the $i := f(i)$ line inside the while loop. The only way i is incremented is by the $f(j) := i + 1$

statement and the corresponding $j{:=}j+1$ (in the for loop) and $i{:=}f(j-1)$ statements. Since $i=0$ initially and since is incremented at most $l-1$ times, we can conclude that the total number of iterations of the while loop (over all iterations of the for loop) is $O(l)$.

A different string matching algorithm is the Boyer-Moore algorithm. Though it has an $O(l+n)$ running time also, it is often faster in practice than Knuth-Morris-Pratt, especially for large alphabets and relatively long pattern strings. However, it is not as well suited for small alphabets (such as DNA) and Knuth-Morris-Pratt. Boyer-Moore scans the pattern string from right to left, comparing character by character with the text. So suppose our pattern is abc and text string is runspotrunabc. Then we begin by comparing 'c' with 'n'. Since it is a mismatch, a pre-computed table tells us that none of the characters in the pattern are are 'c', so we can skip over some of the text (the amount we skip over is equal to the length of the pattern) and now try to match abc with spo.

On the the other hand, if our pattern is abc and out text is robthebank, we would try to match the c with the b and fail. But then since there is a 'b' in our pattern, would would next compare abc with obt. That would fail (c $\neq$ t). The Boyer-Moore algorithm uses a second table in addition to that we described above to determine how far forward we can move the pattern based on suffixes. The algorithm shifts the pattern ahead the larger of the amounts given by the two tables for a given mismatch. Details can be found in the textbook.

4.6 Another Recursive Algorithm

4.6.1 Naive Matrix Multiplication

Let A, B be $n \times n$ matrices. Product[i, j] is computed as follows.

```
Sum = 0;
For a = 1 to n
    Sum = Sum + A[i, a] * B[a, j]
end
Product[i, j] = Sum
```

Since there are $O(n^2)$ entries in Product, the running time is $O(n^3)$.

4.6.2 Recursive Matrix Multiplication

We now discuss Strassen's Matrix Multiplication algorithm. Let A, B be 2×2 matrices.

Observe that we can add or subtract two $n \times n$ matrices in $O(n^2)$ time.

Then define:

M1 = (A11 + A22)*(B11 + B22) ;
M2 = (A21+A22)*B11;
M3 = A11*(B12-B22);
M4 = A22*(B21 - B11);
M5 = (A11 + A12)*B22;
M6 = (A21 - A11)*(B11+B12);
M7 = (A12-A22)*(B21 + B22).

Now the entries in the product matrix C=A*B can be computed as

C11 = M1 + M4 - M5 + M7; C12 = M3 + M5;
C21 = M2 + M4; C22 = M1 - M2 + M3 + M6.

Note that it takes 7 multiplications and 18 additions to compute the $M's$ and the C's. In the simplest case, when $n = 2$, this gives a way to multiply two 2×2 matrices together using only 7 multiplication operations, instead of 8 multiplications using the naive algorithm.

We can use the same idea for larger matrices A and B (so when $n = 2^k, k > 1$). Partition both A and B into four $n/2 \times n/2$ matrices and recursively use the procedure described above. This leads to a recurrence of the form $T(n) = 7T(n/2) + 18n$, which is $O(n^{2.81})$.

If the initial matrix M is such that n is not a power of 2, embed into the upper left corner of an $n' \times n'$ matrix where $2n > n' > n$ and n' is a power of 2; fill the rest of this array with 0's.

Asymptotically faster, albeit more complex, recursive matrix multiplication algorithms have been discovered since Strassen's, though they are likely not faster in most practical situations. In fact, Strassen's algorithm is faster than the naive method only when n gets fairly large (perhaps $n > 100$ or $n > 1000$, depending on hardware).

4.7 Exercises

1. Execute procedure shuffle from QuickSort by hand on the following array $< 7, 3, 1, 2, 5, 8, 11, 14, 0, -3, 16, 19, 23, 6 >$ with pivot element 8.

2. In terms of a perfectly balanced partitioning in the execution of QuickSort, we would like to choose the median of the array as the pivot element. Suppose we used the linear-time median finding algorithm to do this. Write a recurrence relation to describe the running time of the resulting QuickSort algorithm and solve the recurrence. Would it be a good idea to use the method of choosing the pivot in practice?

3. Devise an example where QuickSort takes $\Omega(n^2)$ steps to sort an array with n elements.

4. A building has n floors. The objective is to find at minimum cost the lowest floor for which a pumpkin breaks when it hits the ground. Cost is the number of drops made. Implicit is the assumption that once you've broken all your pumpkins your algorithm terminates. Also, all pumpkins behave the same. Describe an algorithm to find the solution with minimum cost for each of the following:

Problem a: You have exactly one pumpkin. Explain why binary search doesn't work.

Problem b: You have exactly two pumpkins.

Problem c: You have exactly three pumpkins.

5. Give an $O(n \log n)$ time algorithm to take an array with n elements and determine if it contains two elements whose sum is x.
Repeat the problem for three elements, rather than two.

6. Write an algorithm that takes a queue of n integers and using only one stack (and possibly some primitive variables such as integers (but no arrays or other data structures)), re-arranges the elements in the queue so they are in ascending order. Analyze the running time of your algorithm.

7. Let A, B be two arrays of n integers. Let C be the array $\{a + b | a \in A, b \in B\}$. Note that C has $O(n^2)$ elements. Write an $O(n \log\ n)$ algorithm to print the n largest elements in C. [Interesting Fact: the problem cam ac-

tually be solved in $O(n)$ time, see Frederickson and Johnson, SIAM Journal on Computing, 1984]

8. Show how you can simulate n operations on a queue by using two stacks. What is the running time of each operation?

9. Two questions about subsequences in an array A. In this question, a subsequence of A is a set of elements $A[i_1], A[i_2], \dots A[i_k]$ where $i_1 < i_2 < \dots < i_k$. For example, $A[1]A[3]A[7]A[17]$ is a subsequence of 4 elements.

(a) Given a permutation of $1 \dots n$ stored in an array $A[1..n]$. Find the longest *run* of elements in A. A run is a subsequence of consecutive integers. For example, if $A = 7, 2, 4, 3, 5, 6, 1$, the longest run is $4, 5, 6$. Give an algorithm that runs in $O(n)$ time.
Hint: Take advantage of the fact that the elements of A are between 1 and n.

9(b). Now suppose A contains n positive integers. Find the longest subsequence of elements such that each element less than the next element. For example, of $A = 20, 3, 7, 12, 8, 10, 15, 9, 21$, the longest such subsequence is $3, 7, 8, 10, 15, 21$. Your algorithm should run in $O(n^2)$ time or better.

10. Given an array $A[1 \dots n]$ of any type values. Print out the values in the order they appear in A, removing any duplicates. Can you find an algorithm that runs in $o(n^2)$ time?

```
Example:
      A:    7 3 4 6 7 3 5 6 8
output      7 3 4 6 5 8
```

11. Give a fast algorithm to find the mode (most frequently occurring element) in an array.

12. Given an array $A[1..n]$, give an $O(n)$ time algorithm to find the substring of A (a substring of consecutive elements) whose sum is maximum.

```
Example:
      A:    1 4 6 8 -23 4 6 7
      answer: A[1..4] whose sum is 19
```

13. Given $A[1..n]$, determine if A has a "majority" element – a value that occurs in at least $\lfloor \frac{n}{2} \rfloor + 1$ of the positions of A (A may contain values of any type)
a) Give an O(n log n) method
b) Give an O(n) method
c) Same as a), but you may not perform greater than, less than, greater than or equal to, or less than or equal to comparisons. You can only use "=" and "!=". How fast can you solve the problem?
Hint: O(n log n) – Assume n is a power of 2.

14. Input two separate lists of n pairs each. Each pair is of the form (a, b) where each a, b is an integer between 1 and n. No pair appears more than once in a list. In linear time, output the pairs that occur in both lists.

15. An *inversion* is a pair of elements in an array A, say $A[i]$ and $A[j]$, such that $i < j$ and $A[i] > A[j]$. What is the is maximum number of inversions that an array with n elements can have? What is the relationship between a "swap" in the execution of Bubble Sort and an inversion?

4.8 Programming Assignments

1. Run the UNIX **qsort** program on some large arrays and report the running time. (Read the man page on **qsort** to see how to use it in a C program).

2. Compare the results of program 1 with MergeSort, HeapSort, BubbleSort, and CountingSort programs on a number of large arrays.

3. Implement Strassen's matrix multiplication algorithm and compare its performance with the naive $O(n^3)$ algorithms for various sizes of matrices.

Chapter 5

Graph Algorithms and Problems

5.1 Storage

A graph will be denoted by $G = (V, E)$ where V is the vertex set (usually $|V| = n$) and E is the edge set (usually $|E| = m$). In this class we will consider undirected and directed graphs (digraphs) and weighted and unweighted graphs (weights will usually be edge weights). Graphs are commonly used to model many structures such as maps, computer networks and relationships (e.g. in a database or a family tree or any type of flow or organizational chart). A graph is *simple* if it has no multi-edges. We assume simple graphs unless otherwise noted. A *self-loop* is an edge from a vertex to itself.

1) Store graph as a multi-pointer structure, but this can be hard to implement and cumbersome to use.

2) An *Adjacency Matrix* is an $n \times n$ matrix. The matrix is a $0, 1$ matrix when G is unweighted, symmetric when G is undirected. Matrix entry (i, j) is 0 if the is no edge from v_i to v_j and otherwise (i, j) is the weight of the edge. An example follows in Figures 5.1 and 5.2.

If G is *sparse* (i.e. has $o(V^2)$ edges), an adjacency matrix "wastes" a lot of space. Adjacency matrices are easy to implement and enable easy use of matrix multiplication and related operations that are often useful in graph algorithms.

If the graph has edge weights, the edge weights can be used in placed of

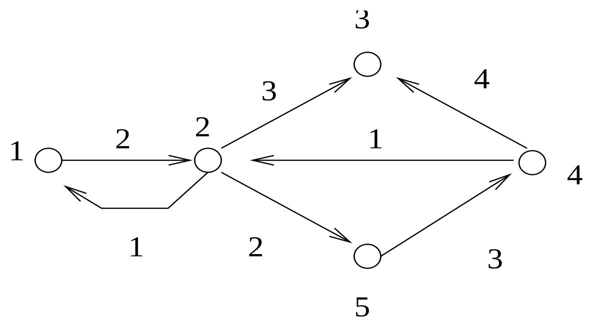


Figure 5.1: Graph G

$$A = \begin{bmatrix} 0 & 2 & 0 & 0 & 0 \\ 1 & 0 & 3 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

Figure 5.2: Adjacency Matrix for G

the 0-1 entries and "null" used as an entry when there is no edge.

3) Adjacency List. A linked list is maintained for each vertex v that contains all vertices u adjacent to v ($v \to u$ edges in the case of digraphs). It costs an extra pointer for each edge, but if G is sparse, this will be require less space than a matrix. Some algorithms run more efficiently with adjacency lists, as we shall see later. Figure 5.3 is the adjacency list for the digraph in Figure 5.1. If G is weighted, an extra field can be added to each node in the linked list.

This requires $O(n+m)$ space but can require $O(n)$ time to determine of two vertices are adjacent, compared with $O(1)$ time in a matrix. Also note that the headers (the left hand box in Figure 5.3) are stored in an array for easy indexing.

5.2 Algorithmic Techniques

5.2.1 Greedy Algorithms

Greedy algorithms are typically fast, iterative algorithms in which a locally optimal decision, or choice, is made at each iteration (and this choice is

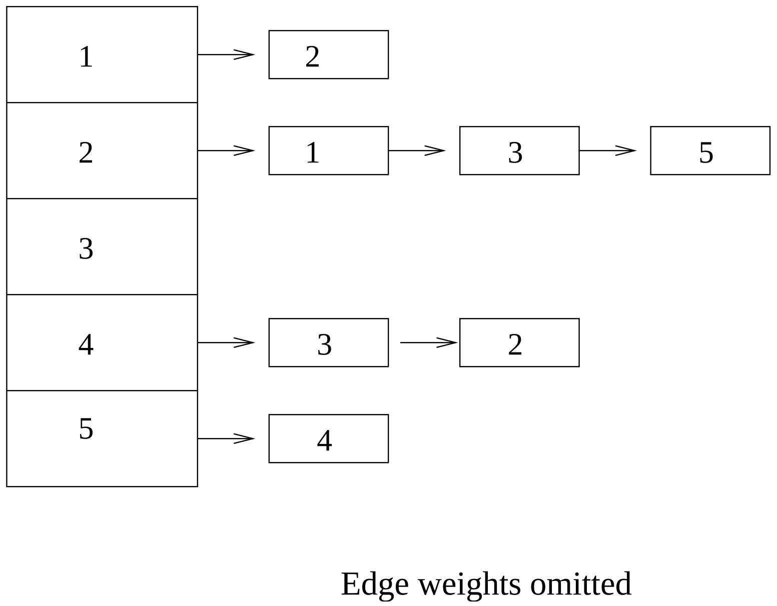


Figure 5.3: Adjacency List for G

never re-evaluated). For some problems, greedy algorithms can lead to fast algorithms that always yield optimal solutions to the problem (see Dijsktra's algorithm below for an example). For others, greedy algorithms to not always yield optimal solutions (see the section on NP-completeness or the greedy dominating set algorithm discussed below). Yet in such cases, they may still be useful due to their speed.

5.2.2 Brute-Force

Brute-force, also known as exhaustive search, is a technique whereby all possibilities are examined in turn. For some problems this technique may prove reasonably efficient and for some problems it may not (e.g., when there are exponentially many possibilities to consider). Trial division for primality testing is an example of this that we have already seen.

Exercises:

1. Devise a brute-force algorithm to determine if a graph is 2-connected. Analyze its running time. [A graph G is 2-connected if $G - v$ is connected for all vertices v].

5.3 Path Problems

5.3.1 Breadth-First Search

A breadth-first search, or BFS, is an exploration, or traversal, of a graph in which all the vertices are visited (assuming the graph is connected). A queue is used to control the order in which vertices are visited and the neighbors of the current vertex are placed on the queue. This algorithm can be used to find shortest paths in graphs with all edge weights equal and positive.

```
BFS(u, v):   //find shortest uv path in (unweighted) graph

Q = empty queue;
color all vertices white;

enqueue <u, 0> onto Q;
color u black;
found = false;

while (Q not empty) and (not found)
 <x, length> = de-queue(Q);
 label x with length
 if x=v then found=true
 else
    for each neighbor w of x
        if w = white then begin
           enqueue <w, length+1> onto Q;
           color w black
        end
    end;
end;

if found then print(length[v]);

end BFS;
```

Runs in $O(|V| + |E|)$ time.

5.3.2 Dijkstra's Algorithm

Consider the problem in a directed, weighted graph of finding the shortest paths (routes) from a single source vertex to all other vertices. The algorithm we shall discuss is a greedy-type algorithm called Dijkstra's algorithm.

Dijkstra's algorithm has the constraint that all edge weights are non-negative–but will work under the slightly more general circumstances that there are no negative-weight cycles (Why?...how can we find a negative weight cycle?).

We note that the correctness of the algorithm can be proven (EXERCISE) by induction on the number of vertices whose shortest paths have been determined thus far.

Dijkstra's algorithm works by maintaining two set of vertices:

S: those whose shortest path from the source has already been determined and

V-S: those whose shortest path is thus far unknown. A vertex in V-S is added to S by selecting the vertex which will have the minimum path length–clearly this vertex is a neighbor of at least one vertex in S.

```
1. for each v in V
        dist[v]=infinity
        pred[v]=nil     -- keep track of predecessor in shortest path
2. dist[source]=0
3. S:={}
4. construct Q -- priority queue based on dist values of vertices in V-S

5. while Q != {} do begin
6.      u:=extract_min(Q);
7.      S:=S + {u}
8.      for each v adjacent to u do
9.              if dist[v] > dist[u] + length(u, v) then   -- found shorter
                                                           -- path to v
10.                       dist[v]:=dist[u] + length(u, v)
11.                       pred[v]:=u
12. end
```

Analysis: Naive: Suppose we implement the priority queue as a linear (unsorted) array. Then each extract min takes O(V) time and this is done

$|V|$ times. In addition, each edge is examined exactly once at line 8 for O(E) time. Thus we have $O(V^2 + E) = O(V^2)$.

Implementing the priority queue using a heap, we can achieve a running time of O((V+E)logV)), which is O(E log V) if all nodes are reachable from the source (connected to the source). This is because building the heap takes O(V) time. Each subsequent extract min takes O(logV) time, of which there are O(V). Each iteration of the for loop that updates the path lengths takes O(logV) time to heapify, and there are O(E) of those.

In practice, elements with infinite distance values should be inserted onto the heap as they are encountered for the first time, rather than as part of the initialization phase. This will reduce the size of the heap for many of the computations performed.

If the graph is "sparse", $(E < O(n^2/\log n))$ this is faster than the naive priority queue method.

The problem with the heap implementation is the update phase. However, using *Fibonacci heaps* [15] we can improve this phase and achieve a running time of $O(V \log V + E)$ time. Experimental results typically find that using Fibonacci Heaps is somewhat slower in practice than heaps or other data structures (e.g., buckets).

Notes– 1. Single source shortest paths can be found in DAGs in $O(V+E)$ time by using topological sort (see exercise 4.7).

2. By using a powerful technique called *scaling*, Gabow developed an $O(ElogW)$ algorithm for single-source shortest paths where W is the maximum edge length. The scaling technique basically considers edge weights bit by bit, iteratively refining the solution until the optimal solution is found.

3. A linear time algorithm for planar graphs was found by Henziner et al. "Faster Shortest Path Algorithms for Planar graphs," *Journal Computer and System Sci.*, vol. 53, pp 2-23, 1997 while a linear-time algorithm for arbitrary graphs (with positive integer edge weight constraint) was found by M. Thorup, "Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time," *Journal of the ACM* vol. 46 (1999), pp. 362-394, though the running time does depend on the fact that the analysis is done using the **RAM** model of computation, which is slightly less general than our usual means of analysis.

5.3.3 Consistent Paths

Let G be an undirected, weighted graph. We wish to design an efficient algorithm to determine if there is a uv path (u and v are given) such that all edge weights on this path are equal.

We use adjacency lists. Our first approach might be to teach each possible edge weight, w, for a path from u to v such that all edge weights on the path are equal to w. But this might take $O(V(V+E))$, time if we do each test using BFS (since we might waste a lot of time scanning through adjacency lists looking for the desired edge weight). A faster algorithm is as follows.

Begin by sorting the edge weights in ascending order (to each edge weight in this list of edge weights we maintain a field indicating the endvertices of the edge). Delete all the edges from G. Now add in all edges of the minimum edge weight and test for a uv path using BFS. If one is found, we can stop. Otherwise, delete those edges (yielding a graph with no edges) and try the edges of the second smallest weight. Continue until a "consistent" path is found or all edges are exhausted. This method takes $O(E\log E+(V+E)) = O(E\log E)$ time (assuming $E \geq V-1$). They key to this analysis is noting that in our repeated BFS tests, we never visit a vertex who does not have incident edges of the desired weight. One implementation detail we need to include is that the BFS always starts at u and if a "dead end" is met (i.e. the queue of vertices becomes empty) we do not initiate a traversal from another "start vertex."
EXERCISE: Consider the problem of finding the uv path whose average edge weight is minimum? How long might such a path be?

5.3.4 All-Pairs Shortest Path

Our first problem in this section is computing the all-pairs shortest path matrix for a graph. That is, we want to know the length of the shortest path between each pair of vertices. If the graph is undirected, of course this matrix will be symmetric. We could run Dijkstra's algorithm $|V|$ times (using adjacency lists). This gives an $O(|V|^2 + |V||E|\log|V|)$ running time using heaps and $O(|V||E| + |V|^2\log|V|)$ running time using Fibonacci heaps. Note that to do so, we must transfer the output from the adjacency list into a row of the all-pairs shortest-path matrix after each of the $|V|$ iterations of Dijkstra's algorithm. If we used Dijkstra's algorithm with an adjacency matrix. the time to compute the all-pairs shortest path matrix

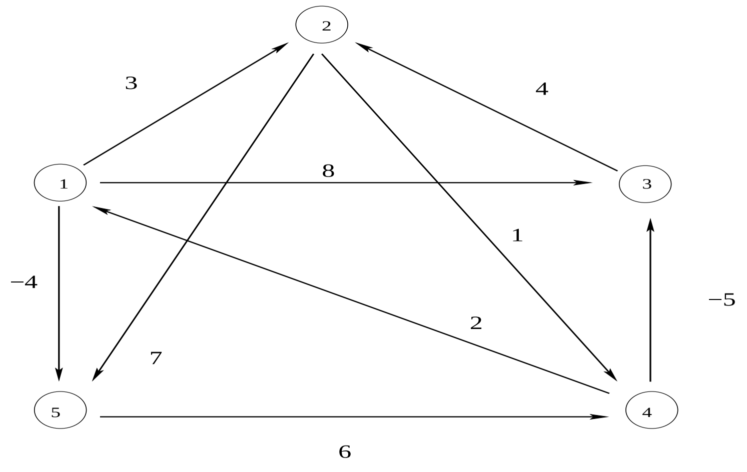


Figure 5.4: APSP Digraph

would be $O(|V|^3 + |V||E| \log |V|)$ using heaps and $O(|V|^3)$ using Fibonacci heaps. If negative weight edges exist, run Bellman-Ford $|V|$ times (yielding an $O(|V|^2|E|)$ running time). The dynamic programming approach will take $O(|V|^3)$ time (and have small hidden constants!).

We shall assume an adjacency matrix representation of the graph and use zero for the diagonal entries, integers for edge weights, and infinity for the "weights" of non-existent edges.

Figure 5.4 contains a digraph we shall use in our example (again, it is from [11]).

We give two algorithms, the second a refinement of the first. Both are "dynamic programming" but they use different criteria.

Let D^0 be the initial adjacency matrix. Both algorithms will construct additional matrices, with the final matrix holding the solution. Let $n = |V|$. Figure 5.5 is D^0 (we will use it in both algorithms), which is just the adjacency matrix of Figure 5.4.

Algorithm 1. Idea: D^m_{ij} (that entry in the table) is the shortest path between v_i and v_j having "at most" m edges (I shall explain below why I use quotes here). We build up paths having more and more edges as we move down the table.

Here is a naive version:

$$A = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

Figure 5.5: D^0, Adjacency Matrix for Figure 5.4

```
input $D^1 = D^0$
for m:=2 to n-1 do begin      /* since path can be up to length n-1 */
    for i:=1 to n do begin      /* $n^2$ entries in D */
        for j:=1 to n do begin
            $D^m_{ij}$ := MIN ($D^{m-1}_{ij}$, $D^{m-1}_{ip} + D^1_{pj}$, $D^1_{ip} + D^{m-1}_{pj}$),
                    over all $p$, $1 \le p \le n$
        end;
    end;
end;
```

Note that since $D^x_{ij} \le D^y_{ij}$ when $x \ge y$, we can use the following version of the algorithm instead, which avoids having to keep a copy of D^1 in memory.

```
input $D^1 = D^0$
for m:=2 to n-1 do begin      /* since path can be up to length n-1 */
    for i:=1 to n do begin      /* $n^2$ entries in D */
        for j:=1 to n do begin
            $D^m_{ij}$ := MIN ($D^{m-1}_{ij}$, $D^{m-1}_{ip} + D^{m-1}_{pj}$), over all $p$, $1 \le p \le n$
        end;
    end;
end;
```

At the MIN step, we search for a better path with m edges that passes through each other vertex possible intermediate vertex p. Running time is $O(n^4)$. The execution on the input of Figure 5.4 can be seen in Figures 5.6-5.8.

To see how this algorithm works, consider $D^4_{3,5} = 3$. We computed this when we saw that $D^3_{3,1} + D^3_{1,5} = 3 < 11 = D^3_{3,5}$. Also note that since all the entries in D^4 are finite, G is strongly connected.

Note that in the "MIN" step, we may build a path that is of length

$$A = \begin{bmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{bmatrix}$$

Figure 5.6: D^2

$$A = \begin{bmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

Figure 5.7: D^3

$$A = \begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

Figure 5.8: D^4

$$A = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

Figure 5.9: D^0, Adjacency Matrix for Figure 8

as much as $m - 1 + m - 1 > m$. As mentioned earlier, we could compute all the values $D^m_{ij} = MIN\{D^{m-1}_{ij}, D^{m-1}_{ik} + D^1_{kj}, D^{m-2}_{ik} + D^2_{kj}, \ldots\}$ for all $1 \le k \le n$, but this is unnecessary (and requires us to store all previous matrices at all times), since we know that $D^{m-1}_{ik} \le D^q_{ik}$ for all $1 \le q \le m-1$.

Algorithm 2. Floyd-Warshall.
Idea: D^k_{ij} is the shortest path between v_i and v_j which passes through no vertex numbered higher than k. By "through," we mean the path enters and leaves the vertex. So, for example, D^k_{ij} makes sense even when $j > k$.

```
input $D^0$
for k:=1 to n do begin    /* paths passing thru verts numbered $\le n$ */
    for i:=1 to n do begin    /* $n^2$ entries in D */
        for j:=1 to n do begin
            $D^k_{ij}$ := MIN ($D^{k-1}_{ij}$, $D^{k-1}_{ik} + D^{k-1}_{kj}$)
        end;
    end;
end;
```

This runs in time $O(n^3)$. Now let us show, in Figures 5.9-5.14, the matrices generated by Floyd-Warshall. In this example, consider $D^1_{4,2}$. The matrix indices indicates we are looking for the shortest path from v_4 to v_2 that passes through no vertex numbered higher than 1. Since there is no edge from v_4 to v_2, $D^0_{4,2} = \infty$. But since there is an edge from v_4 to v_1 (of weight 2, as shown in $D^0_{4,1}$) and from v_1 to v_2 (of weight 3, as shown in $D^0_{1,2}$) we set $D^1_{4,2}$ to 5 (so we might say that v_1 is an intermediate vertex on this path – i.e. we do this computation when $k = 1$ in the algorithm).

EXERCISE: Can these algorithms be easily modified to find the all-pairs **longest** paths (such paths must still be simple paths)? Justify your answer.

$$A = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

Figure 5.10: D^1

$$A = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

Figure 5.11: D^2

$$A = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

Figure 5.12: D^3

$$A = \begin{bmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

Figure 5.13: D^4

$$A = \begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

Figure 5.14: D^5

5.3.5 Maximum Flow

The Maximum flow problem (Max-Flow) is one of the most studied problems in operations research and combinatorial optimization. We shall discuss the most basic algorithms for this problem. We do note that faster algorithms have been devised since this one.

The Max-flow problem is a network problem–a network is a weighted, directed graph in which the weight of an edge represents its *capacity*. The capacity of an edge (u, v) is denoted by $c(u, v)$. Two vertices in the graph are distinguished, the source s and the sink t. We may think of the edges as pipes with a certain diameter (and hence a capacity for a certain amount of liquid per second to flow through them). The objective is to determine how to send the maximum amount of liquid through the pipe network, i.e., find the maximum flow per unit time from the source to the sink.

The flow in a network $G = (V, E)$ is made up of the flows from the edges. Let $f(u, v)$ denote the flow through edge (u, v). The flow must satisfy certain obvious constraints:

Capacity: $\forall u, v \in V, \;\; f(u, v) \leq c(u, v)$
Skew Symmetry: $\forall u, v \in V \;\; f(u, v) = -f(v, u)$
(flow from v to u is opposite of that of u to v.
Conservation: $\forall u \in V - \{s, t\} \quad \sum_{v \in V} f(u, v) = 0$ (flow into a node into node equals flow out of a node)

Thus the total net flow can be seen to be $\sum_{v \in V - \{s\}} f(s, v)$ (equals the flow from the source). Likewise the net flow is also equal to the flow into the sink.

Note – we may also wish to consider a flow problem with multiple sources and/or sinks. These problems are easily solved by modifying the network to "simulate" a single source/single sink problem.

Notation: $f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y)$

This represents the flow through a subgraph (i.e., treat these nodes as one super-node). The same basic constraints hold on a super-node as on a simple node.

The basic Ford-Fulkerson method.

The idea behind the Ford-Fulkerson algorithm is simple: given some initial flow, compute the residual network (i.e., determine the excess capacity for each edge) and then augment the flow, i.e. improve the initial flow by utilizing the excess capacity.

Given some flow through an edge, the residual capacity of the edge (u, v) is $cf(u, v) = c(u, v) - f(u, v)$. Note that the residual capacity of an edge (u, v) may be defined even if the current flow is directed from v to u: for example, if we have 2 units flowing from v to u (and the capacity of that edge is two), the residual flow from u to v is two units.

The residual network induced by a flow f is $Gf = (V, Ef)$ where $Ef = \{(u, v) \in V \times V : cf(u, v) > 0\}$.

That is, the residual network indicates how much extra flow we can push along each edge (in either direction, if allowable).

By finding a flow from s to t in Gf, we may augment the initial flow in G. A simple path from s to t in the residual network is called an *augmenting path*. The capacity of the augmenting path is easily seen to be the minimum of the capacities of the edges in the path–this is the residual capacity. If this value is f', we may augment the initial flow by f' units. Thus by adding this residual flow to our initial flow f, the new flow is $f + f'$.

The simple Ford-Fulkerson algorithm repeats this procedure of finding augmenting paths and residual flows until no more augmenting path exists. Analysis of Ford-Fulkerson:

Unfortunately, the worst-case bound on this algorithm is not good: $O(Ef^*)$ where f^* is the value of max-flow.

However, by using BFS to find a shortest augmenting path in the residual network (that is, we find an augmenting path with the minimum number of edges), it is guaranteed that the number of iterations is $O(VE)$, ensuring that the running time of the algorithm is $O(VE^2)$, since augmenting paths and residual flows can be found in $O(E)$ time (assuming without loss of generality that $E \geq V - 1$).

Min Cut

The dual problem of finding a maximum flow is that of finding the minimum cost cut in a network. A *cut* of a network is a set of edges whose deletion partitions the network into two (or more) distinct components such that there is no path between vertices in different components, the source in one and the sink in the other. The actual cost of "cut" set are those edges between the two partitions. Thus we want to cut the flow from the source to the sink by disabling the minimum total weight of edges.

A famous theorem, the Max-flow/Min-Cut theorem shows the relationship between these two properties:

Theorem 10 *If f is a flow from s to t with total flow value f^* then the following are equivalent:*
1. f is a maximum flow in G
2. The residual network Gf contains no augmenting paths
3. f^*=c(S, T) for some cut (S, T) of G.

That is, the sum of the capacities of the edges in a minimum-weight cut equals the value of the maximum flow.

Bipartite Matching

Another problem related to that of a maximum flow us that of finding a bipartite matching. We may think of the bipartite graph $G = (V, E)$, $V = L \cup R$ as follows. Let L be a set of boys and R a set of girls. We want to marry the boys to the girls so that the maximum number of marriages occur with the constraint that a couple may marry if and only if there is a "compatibility" edge between them indicating they are willing to marry.

Modify G by adding a new node s and connecting s to each vertex in L. Add a new vertex t and connect it to each vertex in R. Now let each edge in the graph have capacity one. Now it is clear that a maximum flow from s to t can use each vertex in L only once (i.e., at most one unit flow can flow into and out of such a vertex) and each vertex in R at most once Hence the edges between L and R used in the maximum flow must be independent, that is, have no vertices common. And these edges therefore constitute a maximum matching. So we can use the Ford-Fulkerson algorithm using BFS to find the maximum bipartite matching in $O(VE)$ time.

In general graphs, not necessarily bipartite, the maximum matching problem can be solved in $O(VE)$ time, due to Edmonds. This problem asks us to find the maximum size set of edges such that no two share a vertex. Edmonds' algorithm for this (from the 1960's) is quite-complicated and clever.

5.4 Stable Marriage Problem

coming soon...

5.5 Spanning Trees

A spanning tree of a graph G is a subgraph T of G such that T is a tree and contains all the vertices of G. If G is a weighted graphs (weights on the edges), a minimum spanning tree of G is the spanning tree T of G such that the sum of the edge weights in T is as small as possible.

Note that if all the edge weights of G are equal, then any spanning tree is a minimum spanning tree and we could us BFS to find a spanning tree.

5.5.1 Union-Find Data Structures

This is an abstract data type with only two operations performed on sets (or lists) of elements. We can FIND(u), which returns the name of the set to which u belongs (one of the elements of a set, called the set representative, is used as the name of the set). We can UNION(u, v) which combines the set u is in with the set v is in. Note that if we do enough UNION operations, all the elements will eventually belong to the same set. We describe two implementations.

Algorithm 1. Have n elements and do m UNION-FINDS in $O(m + n \log\ n)$ time. We use a linked list for each set. Each element has four data fields: one of a pointer to its set representative, one for a pointer to the last element in the set (only the set representative needs to have a value in this field), one containing the size of the set (only the set representative needs to have a value in this field), and one pointer to the next element in the set. A find obviously takes $O(1)$ time. A union is done by first finding the set representatives of the two elements we want to union, then appending the smaller list to the end of the larger. We must update the set representative pointer in each element of the smaller list. Since we at least double the size of the smaller set, the total number of set representative updates that can be done over the life of the algorithm is $O(n \log\ n)$.

Here is an example with sets $\{a, b, c\}, \{e, f\}, \{d\}$.

a	b	c	d	e	f
a	a	a	d	e	e
c	null	null	d	f	null
3	null	null	1	2	null
b	c	null	null	f	null

Algorithm 2. Each set is a rooted tree. A node has one pointer, to its parent. The root is the set representative. A FIND is done by traversing path from node to root. While doing this, we store the nodes visited on a stack (or queue) and later make each of these nodes a child of the root. This is called *path compression.* Hence future FIND operations on these nodes will be

fast. To do a UNION(u, v), we first FIND(U) and FIND(V) (and do path compression) and then make the smaller tree a child of the larger tree. The analysis is complex, but we can do n elements and do m UNION-FINDS in $O(m + n\alpha(m, n))$ time, where α is the inverse Ackermann function, which grows very, very slowly.

5.5.2 Prim's Algorithm

A greedy algorithm for computing a minimum spanning tree of a weighted graph.

```
1. for each v in V
         dist[v]=infinity
         parent[v]=nil     -- keep track of parent in spanning tree
2. dist[source]=0
3. S:={}
4. construct Q -- priority queue based on dist values of vertices in V-S

5. while Q != {} do begin
6.       u:=extract_min(Q);
7.       S:=S + {u}
8.       for each v adjacent to u such that v is not in S do
9.              if dist[v] > length(u, v) then             -- found shorter
                                                           -- edge to v
10.                        dist[v]:=length(u, v)
11.                        parent[v]:=u
12. end
```

Using heaps, the running time is easily seen to be $O(E \log\ V)$, since we do at most E heap update operations. If we use Fibonacci heaps, we get an $O(V \log\ V + E)$ running time.

5.5.3 Kruskal's Algorithm

Kruskal's algorithm for MST uses an alternate greedy strategy.

```
1. T := {}
2. let each vertex be a singleton set in the UNION-FIND data structure
3. Sort edges in increasing order of weight
4. for each edge uv (considered in increasing order of weight)
5.      if FIND(v) != FIND(u)
6.              add uv to tree
7.              UNION(u, v)
```

The running time is $O(E \log\ E) = O(E \log\ V)$, since the UNION-FIND operations take only $O(E\alpha(E, V))$ time. Note that if the edge weights can be sorted quickly, e.g., with counting sort, then this is a very good algorithm.

Asymptotically faster algorithms for the MST problem exist, due to Chazelle (using a different approach and data structure than those mentioned above) J. ACM 47(6):1028-1047, 2000.) and another due to Pettie and Ramachandran (J. ACM 49(1):16-34, 2002.). Chazelle's algorithm runs in $O(E\alpha(E, V))$ time. The exact running time of the Pettie/Ramachandran algorithm is unknown, thought it has been proven be optimal! In other words, it is the fastest MST algorithm possible, though the exact running time has yet to be determined.

5.5.4 MST with 0-1 Edge Weights

We describe two algorithms for the MST in a graph G in which all weights are equal to 0 or 1.

A) Kruskal with counting sort. Running time of $O(E\alpha(E, V))$.

B) $O(V + E)$ time: (use adjacency list)

Do BFS using only weight 0 edges. This forms a spanning forest F of G (Some vertices may be isolated).

Form a graph G' from G and F by coalescing each tree of F into a single vertex, Two vertices u', v' of G' are adjacent if there is an edge from a vertex in tree u' to a vertex in tree v'.

[to see how to do this in $O(V + E)$ time, see below]

Use BFS to find a spanning tree T' of G' (all weights will be weight 1). Find the edges in G corresponding to the edges of T'. Adding these edges to F forms a MST of G.

To form G':

Form one vertex for each tree of F. Label each vertex of G with tree of F that it is in (takes O(V+E) time). For each edge of G, add corresponding edge to G' (may contain redundant edges, but this still only takes O(V+E) time and is easier than trying to ensure G' is a simple graph). We can attach to each edge of G' a field describing the endvertices of the corresponding edge of G.

5.6 Exercises

1. Describe (at a high level) an algorithm to find the largest and second largest elements in an array using at most $n + \log n$ **comparisons**. That is, we only care about the number of comparisons between values in the array. Hint: Assume n is a power of 2 and think about having a "tournament" to determine the largest element.

2. Let $G = (V, E)$ be a digraph. Our goal is to minimize the maximum outdegree over all vertices in the graph by orienting edges – that is we can reverse the direction of individual edges in order to achieve this goal. We want a polynomial time algorithm to do this. Hint: "transfer" an outgoing arc from a vertex with large outdegree to a vertex of small outdegree by finding a path between the two vertices. Repeat this process.

3. (Bonus) Study the minimum spanning algorithm which runs in $O(E\alpha(E, V))$ time and is due to Chazelle. It is based on the soft-heap priority queue, which is a heap-like structure.

4. Modify Dijkstra's algorithm to find a path between u and v that minimizes the maximum edges weight on the path (this type of problem is known as a "bottleneck" problem).

5. Determine if graph G contains a uv path, all of whose edge weights are equal.

6. Show (by an example) that Dijkstra's algorithm (when you change MIN to MAX and $<$ to $>$) fails to compute longest paths.

7. An independent set in a graph is a subset of the vertices, no two of which are adjacent. The maximum independent set is the largest such subset. A Maximal Independent Set of a graph $G = (V, E)$ is an independent set M such that the addition to M of any vertex in $V - M$ creates a non-independent set. Write an efficient algorithm to find a maximal independent set in a graph and analyze its running time. Demonstrate a graph on which your algorithm's maximal independent set is not the maximum independent set.

8. Give a polynomial time algorithm the following: "Does G have a dominating set of size at most 3?" [A dominating set D is a subset of vertices such that every vertex in G is either in D or has at least one neighbor in D].

9. Let G be a weighted graph. Find a second shortest path from vertex u to vertex v (note that 2nd shortest path may possibly have same weight as shortest path).

5.7 Programming Assignments

1. Implement both the iterative and recursive binary search routines and compare their running times on arrays of various lengths (with randomly generated elements in the arrays and randomly generated key values).

2. Implement both Prim's and Kruskal's algorithms for MST and compare their performance on different graphs.

Chapter 6

Advanced Algorithms and Problems

6.1 Depth-first Search

Depth-first Search, or DFS for short, is a method of graph traversal and exploration. It works for directed and undirected graphs and is the building block of many graph algorithms. We shall assume an adjacency list representation of the graph.

Recall that BFS explores a graph by visiting all the neighbors of a vertex before visiting any of their neighbors. DFS, on the other hand, is a recursive procedure in which we go "deeper and deeper" into the graph, backtracking when we come across a vertex that we have already visited. The algorithm is as follows. Both the algorithm and Figure 3 are from [11].

```
DFS(G=(V, E)) – "main" routine
1. for each $u \in V$ do
    2. color[$u$]:=white
    3. pred[$u$]:=NIL
    4. time:=0
5. for each $u \in V$ do
    if color[$u$]=white then DFS-VISIT($u$)
```

```
DFS-VISIT($u$)
1. color[$u$]=gray
2. d[$u$]:=time
3. time:=time+1
```

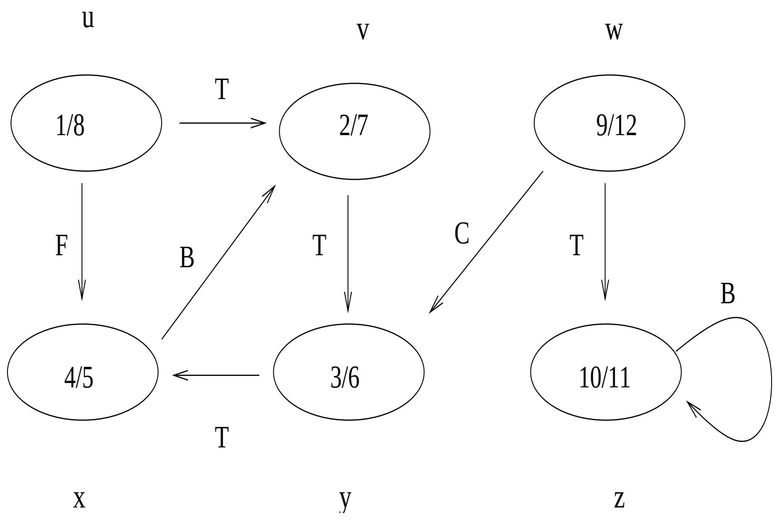


Figure 6.1: Digraph for DFS

```
4. for each $v \in Adjacency_list[u]$ do
    5. if color[$v$]=white then
        6. pred[$v$]:=$u$
        7. DFS-VISIT($v$)
8. color[$u$]:=black
9. f[$u$]:=time
10. time:=time+1
```

DFS starts from some arbitrary vertex (execution obviously will be different for different start vertices, though not in any significant way). All vertices are initially white. A gray vertex is "active" – it has been visited but the recursion that started at it has not terminated. A vertex is black when that recursion terminates. d[u] is the time a vertex is discovered (visited for the first time) and f[u] is the time when the recursion started at u terminates. pred[u] gives the vertex from which we discovered u.

Consider the digraph in Figure 4 (DFS is more interesting in digraphs).

The edges in Figure 4 are labelled according to the type of edge they are with respect to the DFS (note that if a different start vertex were chosen, or the adjacency lists re-ordered, this labelling may be different). Tree edges form the spanning forest of the graph (tree edges are formed when a vertex is discovered); back edges (those edges connecting a vertex to an ancestor of it, which has previously been discovered) in the DFS tree; forward edges (non-tree edges connecting a vertex to a descendent in a DFS tree); cross edges (all other edges: edges between different trees in a DFS forest, or edges in a DFS tree between vertices that are not ancestor/descendents of one another). The vertex labels in Figure 3 are discovery/finish times.

A DFS on the digraph of Figure 4 with start vertex u proceeds as follows (see also [11] page 479). Vertex u is the start vertex and is labelled with time 1. Vertex v is discovered (hence $u \to v$ is a tree edge), y is discovered, x is discovered, $x \to v$ is seen to be a back edge since x is a descendent of the gray vertex v. Vertex x is colored black since its adjacency list is exhausted, and likewise for y at the next step, and v the step after that. Next $u \to x$ is a forward edge, since x is black and is a descendent of u. Vertex u is then colored black since its adjacency list is exhausted. But there are still undiscovered vertices, and the main routine initiates a call to DFS-VISIT with w at time 9. $w \to y$ is a cross edge, since y is black and in a completed tree. Vertex z is then found, its self-loop can be labelled a back-edge, z is then blackened, and w is blackened on the next step.

The running time for any DFS is $\Theta(|V|+|E|)$ since each vertex is visited at least once and each edge is explored. Note that in an undirected graph, each edge may be traversed twice (once in each direction), but this is only a constant factor in the running time.

Question: What is the running time of DFS if we use an $n \times n$ adjacency matrix? Argue that your answer is correct.

Exercise: Run a DFS on Figure 3 using each of the other 5 possible start vertices.

Question: What types of edges are possible in a DFS of an undirected graph?

Exercise: Draw an undirected graph and run a DFS on it by hand, labelling the vertices with start/finish times.

Question: What type of edges indicate a cycle in a digraph?

6.2 Strongly Connected Components

Another building block of graph algorithms is finding the strongly connected components of a digraph. A strong component is a maximal set of vertices such that all vertices in the component have directed paths to one another. A graph with one strong component is called strongly connected.

Exercise: How many strong components does the graph in Figure 1 have?

You could find the strong components by running $O(n)$ DFS's, one from each vertex, in order to determine which vertices are mutually reachable (i.e. have a path in each direction between one another). But this is fairly slow and we can do better using only two DFS's, though in a seemingly strange way!

First we need to define G^T, the *transpose* of a digraph G. G^T is obtained from G by reversing the direction of all the edges of G. Note that this can be done in linear time (if we use adjacency lists).

Exercise Prove that G and G^T have the same strongly connected components.

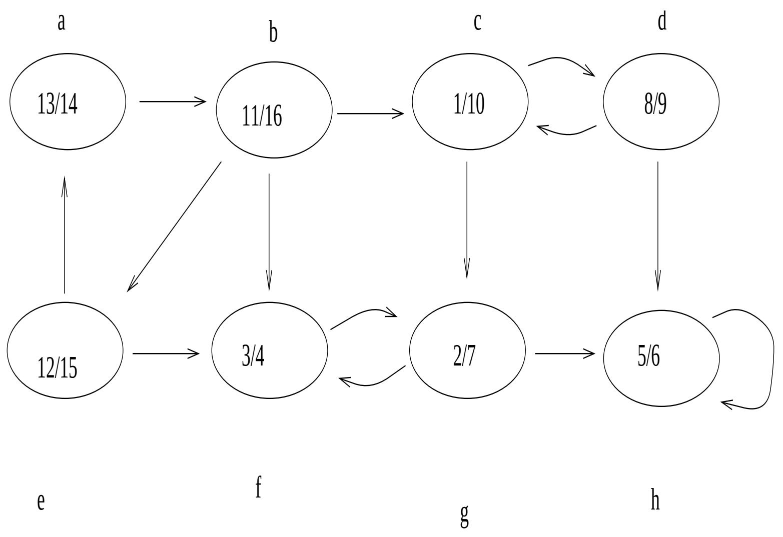


Figure 6.2: Digraph for SCC

Here is the algorithm (and Figures) from [11]

SCC
1. call DFS(G) to compute the finishing time $f(u)$ for each vertex
2. compute G^T
3. call DFS(G^T), but in the main loop of DFS, consider the vertices in decreasing order of finishing time from step 1. These can be stored in a stack during step 1 as vertices "finish."
4. output each DFS tree from step 3 as a separate strong component.

This algorithm is easily seen to take only $O(|V|+|E|)$ time. An example is in order. Consider the digraph in Figure 5, which has been labelled with start and finish times from a DFS started at c.

G^T is given in Figure 6 and the *component graph* in Figure 7. In the component graph, each strong component of the original digraph is a single vertex, with edges directed from component a to b if there is a vertex in component a with an edge to a vertex in component b.

Exercise: Prove that a component graph is acyclic.

Step 3 then will process vertices in Figure 6 in the order b, c, g, h (since there turn out to be four components), since for example, b has the last finishing time.

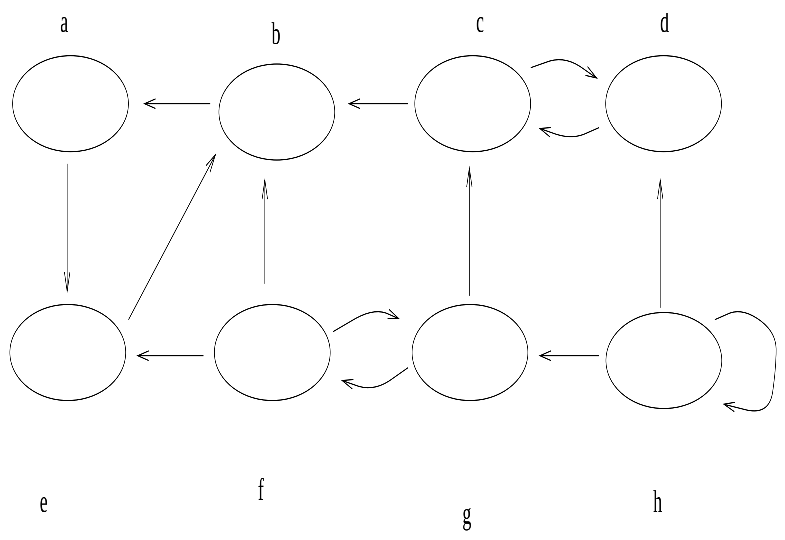


Figure 6.3: G^T of Figure 5

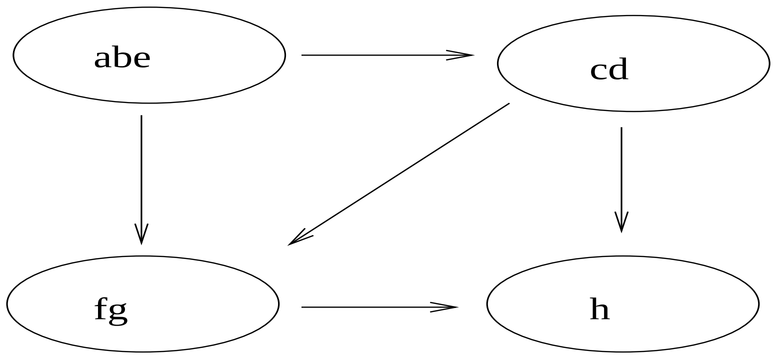


Figure 6.4: Component Graph of Figure 5

Now let's see why it works. Note that if u has a path to v in G then v has a path to u in G^T. To prove this, simply look at a uv path in G: all edges on the path are reversed in G^T.

Theorem 11 *If u and v are output in the same tree in step 3, then u and v are in the same SCC of G.*

Proof: By assumption, u and v are output as being in the same DFS tree of G^T. Let us assume v has a later finishing time than u at step 1. Thus v will be processed prior to u at step 3. First assume that v is an ancestor of u in the DFS tree of G^T (that is, there is a path from v to u in G^T which implies there is a path from u to v in G).

Suppose by way of contradiction that there is no path from v to u in G. There are two cases.

Case 1) Suppose DFS of G initiated at v prior to u (v was gray before u was gray at step 1). Since there is no path from v to u in G, it must be that u finished later than v, a contradiction.

Case 2) Suppose DFS of G initiated at u prior to v (u was gray before v was gray at step 1). Since we know there is a path from u to v, we know u had a later finishing time than v, again a contradiction.

On the other hand, suppose v is not an ancestor of u in the DFS tree of G^T. If u and v are "cousins" in the DFS tree of G^T (neither is a ancestor of the other) we need only consider a vertex w that is a common ancestor of both u and v. Vertex w in this case must have an earlier starting time in the DFS of G^T (since there is a path in the DFS tree of G^T from w to each of u, v and thus a later finishing time than either u, v in the DFS of G. Using the same logic as above, we can show that w, u and w, v belong to the same SCC of G, and thus u, v belong to the same SCC of G.

If u is an ancestor of v in the DFS tree of G^T then there obviously is a vu path in G. Then it must be the case that some vertex x has a later finishing time than v (and thus later than u) and during the DFS of G^T, and x is an ancestor of both u and v in the DFS tree of G^T. Then using the same logic as above, we can show x, u are in the same SCC of G, and x, v are in the same SCC of G, thus u, v are in the same SCC of G.

The remainder of the proof of correctness, namely that if u and v are in the same SCC of G that they are output in the same tree of G^T, is trivial (though the student should complete it as an exercise). □

6.3 Graph Coloring

6.3.1 Basics

A *coloring* of a graph is a labelling of the vertices with colors (equivalently, positive integers), so that each pair of two adjacent vertices receives a different color. The objective is to colore all the vertices in graph G using a few colors as possible.

Graph coloring is arguably the most famous and most important problem in graph theory. It has many applications in areas such as scheduling.

Exercise: Devise and analyze an algorithm to determine if a graph G can be colored with at most 2 colors.

6.3.2 Planar Graph Coloring

An undirected graph is *planar* if it can be drawn in the plane with no two edges crossing. Planar graphs are important in the construction of circuits, where a crossed wire would cause a short-circuit. Determining whether a graph is planar or not (from the internal representation of it) can be done in $O(n)$ time, due to a very sophisticated algorithm [19]. A clever, though still rather complex, linear-time planarity test based on depth-first search is given in [30].

It is also known that all planar graphs can be colored (colors assigned to the vertices of the graph so that no two adjacent vertices have the same color) with at most four colors and an efficient algorithm exists to 4-color a planar graph. But these are quite complex. The original (computer-aided) proof of this is due to Appel and Haken from 1976. Recently, Robertson, Thomas, and Sanders (1997, see http://www.math.gatech.edu/ thomas/FC/fourcolor.htm)l simplified the proof and algorithm somewhat, though the proof still involves over 500 cases. As an interesting note, the problem of determining the minimum number of colors needed to color a specific planar graph (is it 1, 2, 3, or 4?) is difficult (i.e., NP-complete)!

We shall give a simple recursive algorithm to 5-color a planar graph. But first some definitions and results are needed. A *face* in a planar representation of a graph is a region bounded by edges and vertices, containing no edges/vertices in its interior; or the outside of the graph. So a triangle has two faces and a cycle with exactly one chord has three faces. Let F be the number of faces of a planar graph.

Theorem 12 $F + V = E + 2$*, for a connected, planar graph.*

Proof: By induction on the number of edges. The base case, $E = 1$ is easily verified (and the reader should do so!). Assume the theorem is true for all graphs with m edges. Consider a graph with $m + 1$ edges and let us delete an edge e such that this deletion does not disconnect the graph (if there is no such edge, then the graph is a tree and it is easy to see the theorem holds for trees, since they only have one face). By the inductive hypothesis, the formula holds for $G - e$. Now return the deleted edge to the graph as a "new edges." If this new edge needs has one new end-vertex, then V and E both increase by one and F does not change, so the equation is still balanced. If the new edge is between existing vertices, then an old face is split in two, so F and E increase by one. □

Using this, we prove the following.

Theorem 13 *In a planar graph with at least three vertices, $E \leq 3V - 6$.*

Proof: Let F_i denote the number of edges bounding the i^{th} face of (a planar representation of) G. Obviously, $F_i \geq 3$, since $V \geq 3$. Thus

$$\sum_{i=1}^{F} F_i \geq 3F$$

Since each edge borders exactly two faces, we have

$$\sum_{i=1}^{F} F_i = 2E$$

.

Thus $2E \geq 3F$, and using the previous theorem we get, $2E \geq 3(E - V + 2)$, and simplifying this gives $2E \geq 3E - 3v + 6$, which implies $3V - 6 \geq E$. □.

Corollary 14 *Every planar graph has a vertex of degree at most 5.*

Also note that each subgraph of a planar graph is planar. We can now present the algorithm (in the form of an inductive proof that planar graphs can be five colored).

Theorem 15 *Every planar graph is 5-colorable.*

Proof: By induction on $|V|$. The base case is trivial. Let $v \in V$ be a vertex of degree at most 5. By the inductive hypothesis, $G - v$ (the graph with v and all its incident edges deleted) is 5-colorable. Assume $G - v$ is 5-colored (recursively!). Now let us add v back to the graph (with its edges). If the neighbors of v use only 4 different colors, we may safely color v with color 5. So assume the neighbors of v (of which there are at most 5) are colored with 5 different colors. Assume the neighbors of v, $v_1, v_2, \ldots, v_5$ are arranged in a clockwise order around v (see Figure 25). Define $H(i.j)$, $i, j = 1, \ldots 5$ to be the subgraph induced in $G - v$ by the vertices of $G - v$ colored i and j. There are two cases.

Case 1. Suppose v_1 and v_3 lie in different (connected) components of $H(1, 3)$ (see Figure 26). Let $v_i (i = 1, 3)$ lie in component $H_i(1, 3)$ of $H(1, 3)$. If we invert (toggle) the colors in $H_3(1, 3)$ (changing every color 1 vertex to color 3 and every color 3 to color 1) and leave the colors in $H_1(1, 3)$ unchanged, we obtain a valid coloring (since there is no path in $H(1, 3)$ between v_1 and v_3) in which both v_1 and v_3 are color 1, so we may color v with color 3.

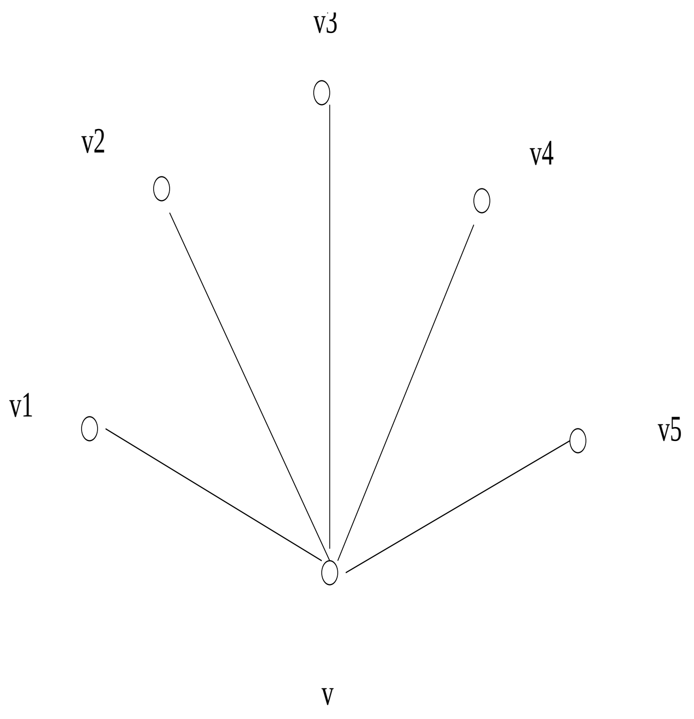
Case 2. Suppose v_1 and v_3 lie in the same (connected) component of $H(1, 3)$. Then there is a path from v_1 to v_3 in $H(1, 3)$. Together with edges vv_1 and v_3v this path forms a cycle, C, in G. Because of the clockwise arrangement of the neighbors of v, one of v_2, v_4 must like inside this cycle and one must lie outside the cycle, else G is not planar (see Figures 27 - 29 for an example): thus v_2 and v_4 cannot lie in the same component of $H(2, 4)$, since any path in $H(2, 4)$ from v_2 to v_4 would have to either pass through a vertex in $C - v$ (which cannot be since the vertices of $C - v$ have colors 1 and 3) or cross an edge on the cycle (which violates the planarity of G). Thus we may apply Case 1 to $H(2, 4)$. $\square$

Exercise: Analyze the running time of this 5-coloring algorithm.

6.4 Analysis of a Greedy Dominating Set Algorithm

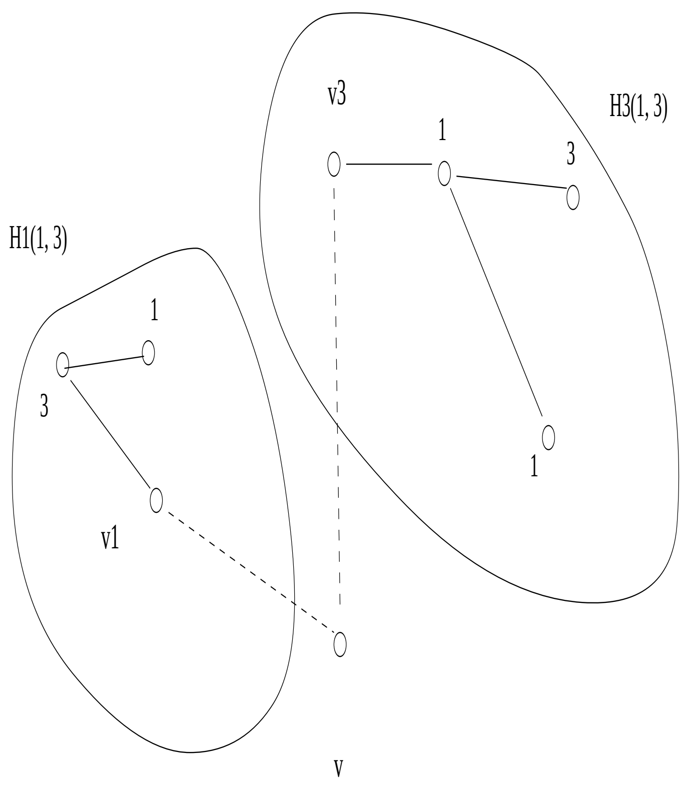
6.4.1 Introduction

One of the basic tenants in the design and analysis of algorithms is that various implementations of the same high-level algorithm can have different performance characteristics. In this section we present six implementations of a greedy dominating set algorithm. The "big Oh" running time of each is analyzed. The implementations and analysis illustrate many of the important techniques in the design and analysis of algorithms. This material fist appeared in [20].



vi has color i in 5-coloring of G-v

Figure 6.5: Five-Coloring a Planar Graph



vi has color i in 5-coloring of G-v

Figure 6.6: Five-Coloring a Planar Graph

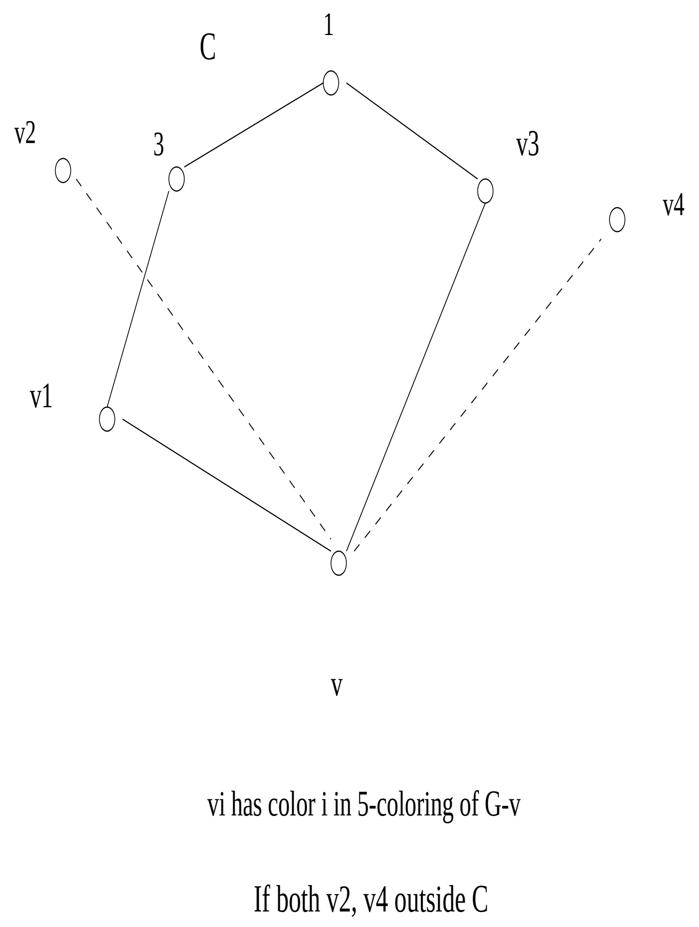


Figure 6.7: Five-Coloring a Planar Graph

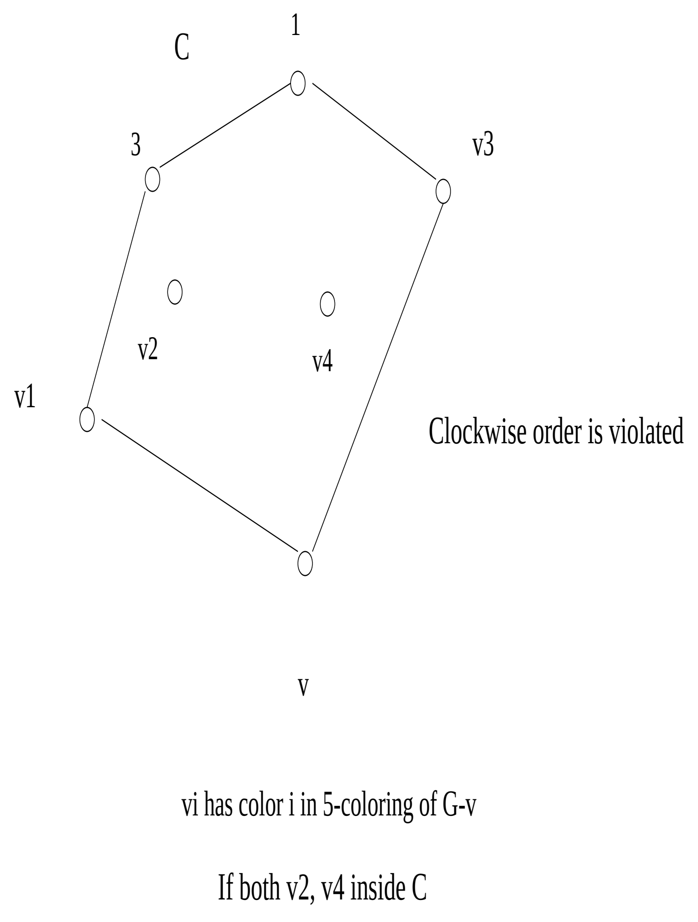

C
1
3
v3
v2
v4
v1
Clockwise order is violated
v
vi has color i in 5-coloring of G-v
If both v2, v4 inside C

Figure 6.8: Five-Coloring a Planar Graph

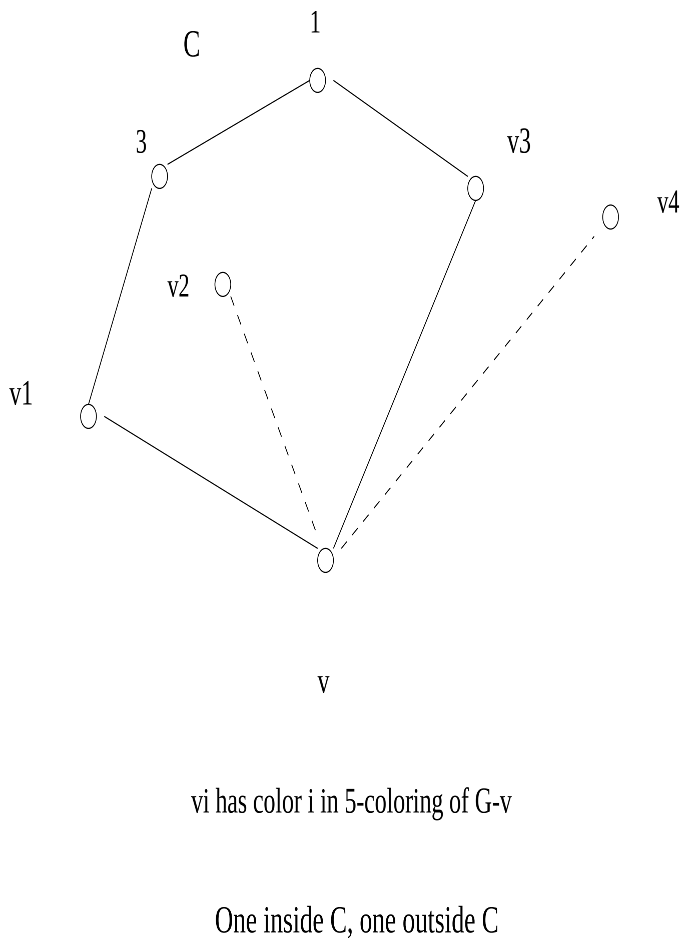

C
1
3
v3
v4
v2
v1
v
vi has color i in 5-coloring of G-v
One inside C, one outside C

Figure 6.9: Five-Coloring a Planar Graph

Recall that a *dominating set* of a graph $G = (V, E)$ is a set $D \subseteq V$ such that each vertex in V is either a member of D or adjacent to at least one member of D. The objective of the dominating set problem is to find a minimum cardinality dominating set. However, this is a well-known $NP-complete$ problem [17], which means it is unlikely that an efficient (i.e. polynomial time) algorithm can always succeed in finding a minimum cardinality dominating set. Thus, we would like to be able to devise a polynomial time algorithm that guarantees a dominating set that is always "close" to the minimum size. A generally accepted definition of "close" is that the ratio of the size of this approximate dominating set to the size of a minimum sized dominating set is at most some fixed constant, for all input graphs. The PCP Theorem [Arora, Sudan, et al., 1992] implies that hat no such polynomial-time *approximation algorithm* can exist for the dominating set problem unless $P = NP$, which is thought highly unlikely. Thus, in practice, we often resort to "heuristic" algorithms for problems such as dominating set. Heuristics often yield good solutions in practice, though pathologically bad solutions can also result.

Note: The PCP Theorem [Arora, Sudan, et al., 1992] implies that no polynomial time approximation algorithm for Dominating Set have have a performance ratio that is less that a function of n (so, in particular, no constant performance ratio is possible) unless $P = NP$.

6.4.2 The Heuristic

The greedy method is useful in designing algorithms. Greedy algorithms are iterative and are characterized by a locally optimal, or "greedy", choice at each iteration. For many problems, such as shortest path and minimum spanning tree, greedy methods can yield the optimal solution. A simple greedy algorithm for the dominating set problem does the following at each iteration: add the largest degree vertex in the graph to the dominating set and remove it and its neighbors. A formal statement of the heuristic is as follows. Note however, that no mention is made of implementation details such as data structures. The algorithm is formally stated below.

```
input G=(V, E);
D:=empty set;
while V not empty do
    let v be vertex in V of max. degree
    D:=D + {v}
    V:=V - {v} - {w | w adjacent to v}
    E:= E - {(w, u) | w adjacent to v}
end;
```

In other words, the algorithm chooses at each step the vertex that has the largest degree– which thus ”dominates” the most other vertices. This vertex and its neighbors are then removed from the graph and the process continues until no more vertices remain.

6.4.3 The Naive Implementation

A first attempt at implementing many algorithms is the naive one. In this case, we begin by computing the vertex degrees and storing that information with each vertex. In terms of data structures, an array or list could be used to store the vertex degrees. Each step chooses the largest degree vertex by scanning the entire list of vertices and proceeds to remove it and its neighbors from the graph. The details of the algorithm are described below. Following the algorithm, we shall discuss the running time of each step. We use V to denote the number of vertices and E the number of edges.

Algorithm 1. Naive Solution

```
1. Compute Vertex degrees
2. repeat
3.   remove vertex with max degree
4.   for each v adjacent to max do
5.     remove v from G
6.     for each w adjacent to v do
7.       degree(w) = degree(w) - 1
8.       remove edge (v, w)
       end for
     end for
   until G is empty
```

We can see that line 1 takes $O(E)$ time; line 3 takes $O(V)$ time per iteration; line 7 takes $O(E)$ time in total (i.e. over the life of the algorithm); and line 8 takes $O(E)$ time in total. Summing the costs shows the running time to be $O(V^2)$ in the worst case.

6.4.4 Choosing a Data Structure

The naive implementation does not really use any data structures other than those that come with the graph and an array to store the vertex degrees. Choosing an appropriate data structure is a key step in designing efficient algorithms In this section, we use a heap to store the vertices according to vertex degree–with the maximum vertex degree being at the root. A heap is a type of priority queue, a generic type of data structure that is used to maintain a set of items by priority so that the item with highest priority can be easily accessed. When the maximum degree vertex and its neighbors are removed from the graph, they must also be removed from the heap, at a cost of $O(\log V)$ per deletion, in the worst case. Furthermore, the degrees of the vertices adjacent to the maximum's neighbors must be updated), which affects the heap. A simple analysis indicates that roughly one update (called a *decrease_key*) may need to be performed for each edge in the graph. This indicates a total running time of $O(V \log V + E \log V)$. The algorithm is formally stated below.

Algorithm 2. Using Heaps

```
1. Compute all vertex degrees
2. Build Heap based on vertex degrees

/* keep a pointer from each vertex to associated node in heap */

3. repeat
4.   remove max element from heap
5.   heapify;
6.   for each v adjacent to max do
7.     remove v from heap
8.     remove v from G
9.     for each w adjacent to v do
10.      degree(w) = degree(w) - 1
11.      update heap accordingly
12.      remove edge (v, w)
13.    end for
14.  end for
15. remove max from G
   until G is empty
```

Step 5 of course take $O(\log V)$ time each time that step is performed.

Step 7 requires $O(V \log V)$ time in total and Step 11 can be seen to take $O(E \log V)$ time over the course of the algorithm.

6.4.5 Combining Two Algorithms

The careful reader will notice that for sparse graphs, $O(V \log V + E \log V)$ is faster than $O(V^2)$. But when $V^2/\log V \in o(E)$, i.e. when the graph is dense, the naive algorithm is actually faster than the heap algorithm. We can have the best of both algorithms by simply counting the number of edges and making the economical choice: if $E < V^2/\log V$, we run the heap algorithm, otherwise we run the naive algorithm. This "combination" algorithm has a running time of $O(MIN(V \log V + E \log V, V^2))$. Combining two distinct algorithms that behave differently is a powerful design technique.

High Powered Data Structures

The most costly part of the heap algorithm is the fact that $O(E)$ decrease_key operations had to be performed in the worst case, each costing $O(\log V)$. The advanced student of algorithms will be aware that a data structure called *Fibonacci heaps* (originally designed by Fredman and Tarjan) is able to perform the same operations as ordinary binary heaps, with the added bonus that decrease_key operations take $O(1)$ unit time each. Technically, the decrease_keys operations take $O(1)$ unit time on average over a sequence of heap operations. This form of analysis is known as *amortized analysis*. Amortized analysis essentially averages the cost of operations over a long sequence of operations. This method allows us to offset a costly individual operation with an inexpensive operation somewhere else in the sequence. For example, suppose we perform one decrease_key of cost $\log V$ and $\log V - 1$ decrease_key operations of cost one each. Typical "worst-case" analysis may lead us to conclude that the sequence runs in time $O(\log^2 V)$. Whereas amortized analysis will charge one additional unit of cost to each of the $\log V - 1$ inexpensive operations and deduct a total of $\log V - 1$ units of cost from the amount charged to the expensive operation. Thus, in our accounting, each operation costs $O(1)$, leading us to conclude that the sequence has a running time of $O(\log V)$. Chapter 18 of [11] contains a very readable introduction to amortized analysis. Replacing heaps with Fibonacci heaps as our data structure gives a running time of $O(V \log V + E)$. However, Fibonacci heaps are quite complex and tend to hide a large constant in the $O()$ notation, and as a result are generally not very practical [11].

6.4.6 Combinatorial Tricks

The impracticality of Fibonacci heaps leads us to attempt to design a more clever algorithm using binary heaps. Two design and analysis techniques come into play. First, a careful observation of the heap algorithm's behavior reveals that a vertex may have a decrease_key operation applied to it more than once on a given iteration. This seems quite wasteful and is a viable place where we could optimize our algorithm. This is done by keeping a *count* variable for each vertex that counts the number of its neighbors that are deleted during a given iteration. A single decrease_key operation will be applied to a vertex (if necessary) which decreases its degree by the appropriate amount. The code follows, annotated with running times per operation. The input is assumed to come in the form of an adjacency list, G.

```
0. G':=G;    /* create a copy of G */
1. compute vertex degrees
2. build heap
3. build adjacency list for G'
      with the following fields:

      type list_element is
        count   : integer :=0;
        deleted : boolean:=false;
        list    : ptr to linked list
                  of vertices;
       end;

       adj_list : array[1..V]
                  of list_element;

       Q : queue;

   /* Q stores vertices deleted during a
      given iteration of line 4  */

4. while G not empty do
5.   remove maximum element from heap;
            also remove it from G;
```

```
6.   for each v adjacent to max in G do
7.     adj_list[v].deleted := true;
8.     remove v from heap;
9.     remove v and its
          incident edges from G;
10.    add v to Q;
11.  end for;

12.  for each v in Q do
13.    for each w in adj_list[v].list do
14.      if not adj_list[w].deleted then
15.         adj_list[w].count++;
16.    end for;
17.  end for;

18.  for each v in Q do
19.   for each w in adj_list[v].list do
20.     if (not adj_list[w].deleted) and
(*)        (adj_list[w].count > 0) then
21.         decrease_key (w,
                adj_list[w].count);
22.         adj_list[w].count:=0;
23.   end for;
24.   remove v from Q;
     end for;
   end;
```

In terms of the individual steps, line 4 iterates $O(V)$ times; line 5 requires $O(\log V)$ time for each remove; lines 6-7 require $O(V)$ time in total; line 8 takes $O(\log V)$ per remove; line 9 takes $O(E)$ time in total and line 10 takes $O(1)$ time for each "add." Likewise, lines 12-17 require $O(V + E)$ time in total. Line 18 iterates $O(V)$ times and line 19 iterates $O(E)$ times in total. Finally, line (*) requires $O(\log V)$ time per decrease_key.

It can be shown that this algorithm runs in $O(V \log V + E)$ time. The idea is to use combinatorial analysis to show that the optimization trick employed actually sped up the algorithm. In particular, we claim that only $O(V + E/\log V)$ in total of "then" statements indicated by (*) above, are satisfied. This implies only $E/\log V$ decrease_key operations are performed, thereby giving the desired running time. The proof of this claim is non-trivial and the interested reader is referred to [20]. Students with an interest in graph theory may wish to find their own proof of this claim.

6.4.7 Elegant Data Structure

Perhaps the best, and most elusive, algorithm design technique of all is sheer cleverness. This was evidenced when the second author conceived the following approach to the problem in an analysis of algorithms course taught by the first author. The data structure designed elegantly suits the problem so well that proving the $O(V+E)$ running time becomes trivial. The idea is to initially sort the vertices by degree. Since the maximum degree of a vertex is bounded by the number of vertices, the sorting can be done quickly as follows. All vertices having the same vertex degree will be linked together in a linked list. The first vertex in the linked list of vertices having degree k can be accessed by indexing into an array, i.e. $A[k]$. That is, the array element $A[k]$ contains a pointer to the list of degree k vertices. Furthermore, each vertex in G has a pointer to its associated vertex in the linked list. Computing the vertex degrees initially takes $O(V+E)$ time. Sorting the degrees can be done in $O(V)$ time by placing each vertex into the appropriate linked list. This can be done in $O(1)$ time by placing the vertex at the head of the $A[k]$ linked list.

At each step of the greedy algorithm, a maximum degree vertex is chosen and removed from the graph (and the vertex degree data structure) along with its neighbors. A vertex of degree k whose neighbor is deleted from the graph can be moved in $O(1)$ time from linked list $A[k]$ to linked list $A[k-1]$. In this manner the "greedy" steps of the algorithm can be performed in $O(V+E)$ time, since only $O(1)$ work is done for each vertex and each edge in G. This running time is optimal, since any dominating set algorithm must examine every vertex and edge of the graph.

```
1. Compute Vertex degrees
2. Sort vertices by vertex degree
3. Build the following data structure:

      A[V-1..1] : array of pointers
                to nodes in graph
      A[i] has a (doubly) linked list of all
                nodes with degree i
      A[i] points to the first
                element in that list
```

```
     Keep index to the first non-empty
        A[i], call this "head"

     Keep a pointer from each node in G
          to its associated node in A.

4.  While A contains a pointer
          to at least one node do
5.    remove first element from A[head];
              call this max;
6.    for each neighbor v of max do
7.      remove v from A;
8.      for each neighbor w of v
9.         let w be in list A[j];
10.        move w to list A[j-1];
11.     end for;
12.   end for ;
13.   while A[head] empty do
14.       head:=head-1;
    end while from line 4;
```

In this final algorithm, steps 1-2 take $O(V+E)$ time. The "for" loop at line 6 iterates $O(V)$ time in total and the "removed" at line 7 takes $O(1)$ time per "remove." The "for" loop at line 8 iterates $O(E)$ times in total and steps 9-10 require $O(1)$ time per iteration. The "while" loop on line 13 takes $O(V)$ time.

6.5 More Dynamic Programming Examples

6.5.1 Matrix Chain Multiplication

Recall that the product of two matrices $C = A \times B$ is equal to

$$C_{ij} = \sum_{k=1}^{columns(A)} A_{ik} \cdot B_{kj}$$

Assuming A, B, C are $n \times n$ matrices, by the obvious method we can compute the n^2 elements of C in $O(n^3)$ time. More complex algorithms (with fairly large hidden constants) that take $O(n^{2.81})$ time and even as fast as $O(n^{2.37})$ also exist.

Consider the multiplication of n matrices $M_1 \times M_2 \times \ldots \times M_n$. Each M_i is a $p_i \times q_i$ matrix. Recall that we can multiply two matrices A, which is $p \times q$, and B, which is $y \times z$, if and only if $q = y$ or $z = p$ That is, $A \times B$ is allowed only if they are $p \times q$ and $q \times z$ matrices and $B \times A$ is allowed only if B is $y \times p$ and A is $p \times q$. In words, the number of columns of the first matrix must equal the number of rows of the second matrix.

Exercise: Write an $O(n^3)$ algorithms to multiply two $n \times n$ matrices.

The cost, in terms of total number of operations, to multiply a $p \times q$ matrix times a $q \times r$ matrix (using the simple $O(n^3)$ algorithm) is $p \cdot q \cdot r$ (do you see why?). Thus the q term only appears once in the cost (rather than say $p \cdot q \cdot q \cdot r$). Since matrix multiplication is associative (but not commutative) we can multiply the n matrices M_i is several possible orders. For example (data taken from [3])

```
    M1        *        M2        *        M3        *        M4
 [10 X 20]         [20 X 50]           [50 X 1]         [1 X 100]
```

Can be done in the order $M1 \times (M2 \times (M3 \times M4))$ using 125,000 operations or in order $(M1 \times (M2 \times M3)) \times M4$ requiring 2,200 operations!

We could evaluate all possible ordering of the matrices and choose the one that is best, but there are an exponential number of possible such orderings; we will do it in $O(n^3)$ time as follows.

In the following table we compute the optimal solution for the example above. The i, j^{th} entry (row i, column j) in the table represents the optimal solution to the matrix multiplication problem for a subsequence of the matrices–beginning with matrix Mi and proceeding through Mj. Thus when we reach table entry (1,n) – we have the optimal solution.

This tabular method is the key to dynamic programming–building optimal solutions and deciding which optimal sub-solution(s) make an optimal solution to the next-sized sub-problem.

```
$m_{i,i}$ := 0, for all $i$
for l:=1 to n do begin
    for i:=1 to n-l do begin
```

```
            j:=i+l;
            $m_{ij} := MIN_{i \le k < j}\ (m_{ik} + m_{k+1,j} + (r_{i-1} \cdot r_k \cdot r_j))$
        end;
    end;
end;
```

Where Mi has r_{i-1} rows and r_i columns.

So in our example, $r_0 = 10, r_1 = 20, r_2 = 50, r_3 = 1, r_4 = 100$. The computation table is shown below.

$m_{1,1} = 0$	$m_{2,2} = 0$	$m_{3,3} = 0$	$m_{4,4} = 0$
$m_{1,2} = 10,000$	$m_{2,3} = 1000$	$m_{3,4} = 5000$	-
$m_{1,3} = 1200$	$m_{2,4} = 3000$	-	-
$m_{1,4} = 2200$	-	-	-

$m_{i,j}$ is the minimum cost to multiply matrices Mi through Mj.

Let's see where these entries come from! Consider $m_{1,4}$ (looking for a product subsequence of matrices numbered 1 through 4). According to the algorithm (the MIN step) this entry is the minimum of

(M1 * M2 * M3) * M4 = $m_{1,3} + r_{4,4} + r_0 \cdot r_3 \cdot r_4$ = 1200 + 0 + 1000
(M1 * M2) * (M3 * M4) = $m_{1,2} + m_{3,4} + r_0 \cdot r_2 \cdot r_4$ = 10,000 + 5,000 + 50,000 = 65,000
M1 * (M2 * M3 * M4) = $m_{1,1} + m_{2,4} + r_0 \cdot r_1 \cdot r_4$ = 0 + 3000 + 20,000 = 23,000

Thus the optimal is 2200, which recursively, we can see is the order (M1 * M2 * M3) * M4 = (M1 * (M2 * M3)) * M4. Why is this so – why did we multiply (M1 * M2 * M3) in this order?

6.5.2 DNA Sequence Alignment

A DNA sequence is a string a letters, each letter being either an A, T, C, G. Given two DNA sequences, of possibly different lengths, we want to *align* them in a way that best expresses their similarities. For example, if one string is ATTACG and another is TTAG, we may select the alignment

```
AATTACG
--TTAG-
```

A "-" indicates a space in the alignment. Note that there is a mismatched character in this alignment, the C and G. Here is another possible alignment of these strings:

```
AATTACG
--TTA-G
```

Note that this alignment has inserted a space inside one of the strings, which is allowable. We need to define the *score* of an aligment, which represent how good the aligment is. For example, we would like to express that the following alignment is probably not very good:

```
AATTACG
TTA-G--
```

Let the strings be s and t. One possible scoring scheme is to give $p(i, j)$ points if $s[i]$ is aligned with $t[j]$ and the $s[i] = t[j]$, g points if $s[i]$ is aligned with a space inserted into position $t[j]$ and $-p(i, j)$ points if $s[i]$ is aligned with $t[j]$ and the $s[i] \neq t[j]$. We do not allow two spaces to be aligned together. For instance, if $p(i, j) = 1, g = 2$, then the three alignments above have scores of -4, -2, and -10, so we would say the second alignment is the best of these three. Our goal is to compute the best alignment. We do so using dynamic programming.

The algorithm will compute optimal ways to align prefixes of s and t. Our description of the algorithm follows that of [34]. To align $s[1..i]$ with $t[1..j]$ there are three possibilities: align $s[1..i]$ with $t[1..j-1]$ and match a space with $t[j]$, align $s[1..i-1]$ with $t[1..j]$ and match a space with $s[i]$, or align $s[1..i-1]$ with $t[1..j-1]$ and (mis)match $s[i]$ and $t[j]$.

The algorithm is as follows [34]:

```
Input: s and t
m = |s|
n = |t|
for i = 0 to m
    A[i, 0] = i * g
for j = 0 to n
    A[0, j] = j * g

for i = 1 to m
    for j = 1 to n
```

```
        A[i, j] = MAX {A[i-1, j] + g,
                       A[i-1, j-1] + p(i, j) if s[i]=t[j],
                       A[i-1, j-1] - p(i, j) if s[i]!=t[j],
                       A[i, j-1] + g}

    end for
end for

Output(A[m ,n])
```

If $s = AAAC$ and $t = AGC$, then A would be as follows:

```
0   -2   -4   -6
-2   1   -1   -3
-4  -1    0   -2
-6  -3   -2   -1
-8  -5   -4   -1
```

So the optimal alignment of these two strings has score -1. Note that this comes from optimally alignment AAA with AG (scorfe -2) and then matching C with C (score +1)

6.5.3 A Decision Problem example of Dynamic Programming

The partition problem. Given a (multi-)set A of integers $\{a_1, a_2, \ldots, a_n\}$, determine if A can be partitioned into two distinct subsets A_1 and A_2 (with $|A_1| + |A_2| = |A|$) such that the sum of the elements in A_1 equals the sum of the elements in A_2. This is the electoral college "tie" question: is it possible for a presidential election (the number of electoral votes) between two candidates to end in a tie?

In general, it takes exponential time to solve this problem (as we shall see) but under some special circumstances, we can use dynamic programming to optimally solve the problem in polynomial time. The special circumstances are that we can place a fixed upper bound on the size of the largest a_i, such as, for example n^x, for some constant x.

Key Point: The number of bits needed to store an integer (what we usually call the size of the input) is logarithmic in the magnitude of the integer – and we measure the running time of algorithms in terms of the input size. So if the size of the input is $\log n$ and the computation time is n steps – then the running time is exponential in the size of the input.

We solve the by filling in a table T with entries $T[i, j]$, for $1 \leq i \leq n$ and $1 \leq j \leq \frac{1}{2} \sum_{k=1}^{n} a_k$. Note that the solution is guaranteed to be "no" (no partition exists) if the sum of the a_k's is odd. T is filled in ROW by ROW from left to right (and from top to bottom). Each $T[i, j]$ is "True" if there is a subset of the first 1..i elements that sum to j. Thus if $T[n, \frac{a_k}{2}]$ is True (or in fact any entry in the rightmost column), the solution in "True." The dynamic programming "engine" is the following statement:

$T[i,j]$:= true if and only if either

i) $a_i = j$

ii) $T[i-1,j] =$ True

$iii)T[i-1,j-a_i] =$ True

Consider the following T with $a_1 = 1, a_2 = 9, a_3 = 5, a_4 = 3, a_5 = 8$. The rows are indexed by i and the columns by j. The example is from [17].

0	1	2	3	4	5	6	7	8	9	10	11	12	13
T	T	F	F	F	F	F	F	F	F	F	F	F	F
T	T	F	F	F	F	F	F	F	T	T	F	F	F
T	T	F	F	F	T	T	F	F	T	T	F	F	F
T	T	F	T	T	T	T	F	T	T	T	F	T	T
T	T	F	T	T	T	T	F	T	T	T	T	T	T

In this example, the sum of all the elements in 26 and the answer is "Yes" (if we let $A_1 = \{a_1, a_2, a_4\}$). To see how the table works, observe that $T[4, 13] =$ True because $a_4 = 3$ and $T[3, 10] =$ True.

Question: What is the running time of this algorithm?

Hint: Do you see why the running time is exponential in the input size? Of course, in practice, we may be fortunate and all of the integers in the input are small (say, 1000 or less) in which case this algorithm is fast and practical. But if there are large integers in the input (say, on the order of 10^{10}), the algorithm will be **VERY** slow.

6.5.4 Knapsack

In the Knapsack problem, we are given a list of n items, each item has a size and a profit. The goal is to find a subset of the items whose cumulative size is at most C and which maximizes the sum of the profits of the items. Here is a dynamic programming solution.

```
T[i, j] = TRUE iff there is a subset of first i items whose profit
is j

X[i, j] = smallest cumulative size of subset of first i items
whose profit is j (infinity if no such subset)

// pi is profit of ith item, si its size
// P is the max of the pi and S is the max of the si

for i = 1 to n    // initialize first column
    T[i, 0] = 0
    X[i, 0] = 0
end

for i = 1 to n
   for j = 1 to n
        X[i, j] = infinity
   end
end

for all j != p1
   T[1, j] = False
end

T[1, p1] = TRUE

X[1, p1] = s1

for i = 2 to n
   for j = 1 to nP
       if T[i-1, j] = TRUE then
           T[i, j] = TRUE
           X[i, j] = X[i-1, j]
       end if
       if X[i-1, j - pi] + si <= C then
           T[i,  j] = TRUE
           X[i, j] = min{X[i, j], X[i-1, j-pi] + si
       end if
   end
end

output largest j such that T[n, j] = TRUE
```

Run time is $O(n^2P)$ (Run time increases by a factor of $n \log SC$ if we modify X to actually store the sets of items themselves that achieve the given profit, where S is maxsi. Of course, the run time is dependent on the magnitude of the profit and size integers, so it may not be polynomial time.

6.5.5 Offline 2-server algorithm

Give a dynamic programming algorithm to compute the optimal schedule for the two-vehicle routing problem in a graph, given that the requests must be served in a specific order r1, r2, ..., rn.

Given graph $G = (V, E)$ and sequence of requests: $\{r_1, \ldots, r_n\}$. Suppose the servers are initially at v_1 and v_2
Let V denote the number of vertices and n the number of requests.

1. Compute all-pairs shortest path: $O(V^3)$
2. Set up a 2-D table T[0..n, (1,1)..(V, V)], i.e. V^2 columns, one for each possible configuration of the 2 servers. So a configuration is just a pair (s_1, s_2) where s_1 is the location of server 1 and s_2 is the location of server 2.

– We number these configurations from 1 to V^2 in the obvious manner

3. let T[0, (v_1, v_2)] = 0 and T[0, (i, j)] = ∞, for (i, j) $\neq (v_1, v_2)$

– T[i, j] will be the minimum cost of serving the first i requests and finishing (finishing the first i requests) in configuration j

```
4. for i:=1 to n
   for j:=1 to $V^2$ do
      T[i, j]:=min{T[i-1, k] + cost(k, j)} over all configurations k
      where cost(k, j) is cost of moving from
         configuration k to configuration j
      That is, if k=(u, v) and j=(w, x) then
      cost(k, j) = distance(u, w) + distance(v, x)
      and distance is from the APSP matrix computed in step 1.
      Furthermore, if j=(u, v) then at least one of u, v must equal $r_i$ else
T[i, j] = $\infty$.
      That is, we can safely ignore candidate configurations that do not
include the requested vertex.
   end;
end;
```

The minimum cost to serve the sequence is then the minimum value on row n. (If we actually want to know **what** that sequence is, we can store some additional information in the table (i.e. where each optimal value in each cell came from, that is the sequence of server movements, i.e. which server served all the preceding requests).

A simple analysis gives a running time of step 4 is $O(n * V^2 * V^2)$ which is $O(nV^4)$. Note that this algorithm works for the k-server problem in general, though the running time is exponential in k. Faster algorithms exist based on maximum flow techniques [10], for example Chrobak et al. give one which runs in time $O(nV^2)$ – note that this is independent of k.

What might happen if instead of the request sequence $r_1, \ldots, r_n$ being given as input, it is revealed to us one request at a time? That is, we are given one request, must serve it (by sending one of our servers), and then the next request is revealed. This would be called an *online* problem, since the input is revealed in an online fashion.

The performance of online algorithms is typically measured by their *competitive ratio*, i.e. their cost divided by the cost of an optimal offline algorithm (one that knows the request sequence in advance, such as our homework problem), over request sequences of arbitrary length.

One obvious algorithm for the online 2-server problem is the greedy approach: always send the server that is closest to the request. As an exercise, can you show the greedy approach can perform very poorly (relative to a more clever strategy) for the online 2-server problem when the graph is a path [for example: o - o - o - o - o] with n vertices. One "optimal" online 2-server algorithm for the path on n vertices does the following: the closer server always serves the request, but if the two servers are on opposite sides of the requested vertex, the farther server moves d steps closer to the requested vertex, where d is the distance between the closer server and the requested vertex. Note that this is optimal in the sense that the total distance traveled by the two servers is never more than twice the distance traveled by the serves of an optimal offline algorithm. A formal analysis of this algorithm is in the section on on-line algorithms.

There exist several 2-competitive algorithms for the 2-server problem (in any graph or metric space) and a $2k-1$-competitive algorithm for the k-server problem. It is known that k is a lower bound on the competitive ratio for the online k-server problem and it is believed that the $2k-1$-competitive algorithm (which is quite simple and is based in part on our dynamic programming solution above) is in fact k-competitive. A classic application of the k-server problem (in this case on the complete graph with n vertices and (say) $n-1$ servers is the paging problem.

Online problems will be considered in more detail in a later chapter.

6.5.6 Trees

Suppose we want to find the maximum-sized independent set S in a tree T (an independent set is a set of vertices such that no two of them are adjacent). One way to do it is as follows.

Start with S empty. Add all leaves to S (unless T is a tree with one edge, in which case add just one of the leaves to S). Delete all vertices added to S as well as all vertices adjacent to those added to S. The results graph is a forest. Recurse on each tree in the forest.

Exercise: Prove the above algorithm is correct. Analyze its running time (hint: it is polynomial time).

However, suppose we add weights to each vertex of T and wish to find the independent set S such that the sum of the weights in S is as large as possible. Then the above algorithm is useless. A kind of dynamic programming algorithm can solve the problem. Working from the leaf level up on larger and larger subtrees until we get to the root, we can solve this problem by storing some information about the optimal solutions in the subtrees rooted at each vertex. The details are left as an important exercise.

6.6 Linear Algebra Tools – An Overview

6.6.1 Solving Sets of Linear Equations

6.6.2 More on Matrices

6.6.3 Linear Programming

6.6.4 Integer Linear Programs

6.7 Exercises

1. The following problem is from [8]:

Let L be a set of strings of characters and let Q be a program that runs in polynomial time that decides whether or not a given string is in L or not. Write a polynomial time algorithm (using dynamic programming) to decide if a given string is in L^*, where L^* consists of strings that are formed by concatenating one or more strings from L.

Note: Be aware that the empty string, ϵ, may or may not be a member of L.

2a. Give a dynamic programming algorithm to find a smallest dominating set in a tree. A dominating set is a subset of vertices, D, such that every vertex is either in D or adjacent to at least one member of D.

b. Modify your solution to a) in the case that vertices are weighted and we are searching for a minimum weight dominating set (the sum of all the weights of vertices in the dominating set is minimum).

3. A graph is said to be *2-connected* (or *bi-connected* if any single vertex (and its incident edges) can be deleted and the remaining graph is still connected. A connected graph that is not 2-connected contains at least one *cut-vertex*, whose deletion disconnects the graph.

a) Use brute force method to determine in polynomial time whether or not a graph is 2-connected. Hint: test if each vertex is a cut-vertex. Analyze the running time of your algorithm.

b) Use depth-first search to derive a linear-time algorithm to determine if a graph is 2-connected. Hint: During the DFS, at each vertex v, test whether any descendent of v has a back edge to an ancestor of v. Also note under what conditions the root of the DFS tree is a cut-vertex.

4. Use BFS as the basis of an algorithm to determine if a graph can be colored with 2 colors.

5. Given a context-free grammar in Chomsky Normal Form and a string x, give a dynamic programming algorithm to determine if x can be generated by the grammar.

6. We wish to determine if there is a simple uv path in an undirected graph that passes through vertex x. Argue that the following does not work: find a ux path; find a xv path; paste the two paths together.

Bonus: Find a polynomial time algorithm to solve the problem.

7. Find a linear-time algorithm for single-source shortest in a directed acyclic graph. Hint: First do a *topological sort* of the vertices. A topological sort of a DAG on n vertices labels the vertices from 1 to n so that all arcs are directed from a lower number vertex to a higher numbered vertex. Your topological sort should be based on DFS.

8. Describe and analyze (be as brief as possible) and algorithm to find

a path P between two vertices, u and v, with the following characteristic: the weight of the ”heaviest” edge on P is less than that of all other paths $Q \neq P$. That is, we want the path which minimizes the maximum edge weight on the path.

6.8 Programming Assignments

Chapter 7

Complexity: An Introduction

7.1 Introduction

7.1.1 Easy Problems, Efficient Algorithms

We say an algorithm is *efficient* if it runs in polynomial time. Of course, if a linear running-time algorithm for a problem is available, we'd hardly call an $O(n^{40})$ running-time algorithm for the same problem "efficient", except in this technical sense. We've considered some problems now for which we found no efficient algorithms:

* all pairs longest paths

* coloring vertices of a graph with minimum number of colors

* partition

The reasons why it can prove difficult to find efficient algorithms for solving particular problems, such as the above, will be a main topic for the rest of the semester. Another main objective in the remainder of the course is to learn to recognize intractable problems when we come across them.

We should be careful to distinguish algorithms from problems. A problem is easy if it can be solved by an efficient algorithms (more on algorithms as solutions to problems below). Here is an easy problem:

Sorting. We know many efficient algorithms that solve this problem. But consider the following algorithm for sorting:

<u>Stupid Sort</u>

```
begin
   input A
   for i in 2..length of A loop

       form the set, Pi,  of permutations of A
       for each alpha in Pi
           test if alpha is sorted

           exit loop if alpha is sorted

       end loop;
   end loop;

   for i in 1..length of A loop

       A(i) := alpha(i);

   end loop;
end Stupid Sort;
```

This algorithm for the Sorting problem is TERRIBLE. In a worst case it will consider all possible permutations of A before finding a sorted one and finishing running-time $O(|A|!)$. Nevertheless the sorting problem is easy. So the difficulty of a *problem* can't be judged by the (in)efficiency of some particular *algorithm* designed to solve the problem.

Another example, a famous one, of an inefficient algorithm for an easy problem:

In the program for the Internet Worm (1987) the programmer used a sequential search to search a sorted array, rather than a binary search, thus requiring so much computation time as to bring the Internet to a halt.

A problem is said to be "hard" or "intractable" if it is not easy, i.e. just in case it is not solvable by any efficient algorithm. We are now going to try to characterize problems more generally, to distinguish the easy from the hard.

7.1.2 Problems, Instances and Solutions

Generally, a problem is a question with some parameters and constraints. An "instance" of a problem is obtained by specifying particular values for the parameters of the problem (i.e. by specifying "input"). A solution to an instance of a problem is a correct answer to that instance. The following example is a very famous problem.

The Travelling Salesman's Problem (TSP).

$C = \{c_1, ..., c_m\}$ is a set of "cities" (vertices) and d is a function such that for all i and j, $1 \leq i,j \leq m$, $d(c_i, c_j)$ is a positive integer, find an ordering

$$< c_{\pi(1)}, c_{\pi(2)}, c_{\pi(3)}, ..., c_{\pi(m)} >$$

that minimizes

$$d(c_{\pi(1)}, c_{\pi(m)}) + \sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)})$$

TSP asks us to find, for any given complete weighted graph, a tour of minimum length in that graph (a tour is a cycle that visits every vertex exactly once apart from the starting/end point). Intuitively, think of the set C as a set of cities, with the function, d, giving the distance between any pair of cities in C. What we want is to find a way to visit every city other than the starting point exactly once, minimizing the distance we've travelled along the way. The Figure illustrates a particular instance of TSP:

An optimal solution to this instance is easy: we can just go around the outside edges. But in general, for any given complete weighted graph, $G = (V,E)$, there are $O(V!)$ possible tours. And there are an infinite number of possible instances of this problem, since there are an infinite number of complete weighted graphs. (In general, the only interesting problems are ones such as this, with an infinite number of instances; otherwise, solutions could be produced by brute-force and hard-coding). We can ask whether there is an efficient algorithm which provides a solution for any given instance of TSP. That's what is meant when we ask whether TSP itself is easy or hard.

In general, an algorithm *solves* (or is a solution to) a problem if it takes any instance of the problem as input and outputs a correct solution to that

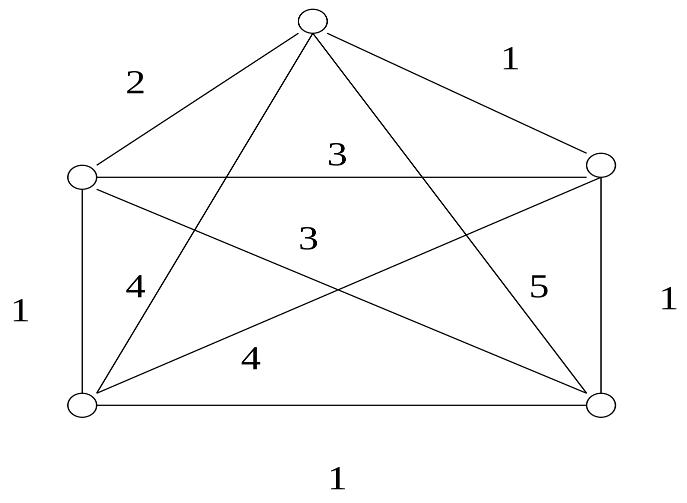


Figure 7.1: TSP Example

instance.

Two Sorts of Problems

Decision Problems are problems that call for a "yes/no" answer. Examples:

* Partition problem: Is there a way to partition a set S of integers into two non-empty subsets so that the sum of elements in one subset equals the sum of elements in the other?
* Hamiltonian Cycle: Is there a simple cycle that visits all vertices in graph G?
* Primality: Is n a prime number?

In contrast, there are what are called "optimization problems."
Optimization Problems: problems that seek a minimum or maximum of some function.
* Travelling salesman
* Minimum Spanning Tree
* Maximum Flow problem

For the time being, we confine our attention to decision problems. This may seem limiting, but actually any optimization problem can be turned into a decision problem by adding a "dummy" parameter. For example, with TSP, we can ask:

Does G have a tour of weight at most k? Where G is a complete weighted graph and k a positive real number.

Here, k is the dummy parameter. For any choice of k we have converted the optimization problem into a decision problem.

The graph coloring problem affords another example of how an optimization problem can be turned into a decision problem by employing a dummy parameter in the statement of the problem:

Can graph G be colored with k colors?

It turns out that this question is easy if $k = 2$ (It's equivalent to the decision problem: "Is G bipartite?", which we can answer in linear time using a simple graph traversal, such as a modified depth-first search), and it's easy as well for $k \geq n$. But for many choices of k strictly between 2 and n, the problem is hard (we note that graph coloring remains easy solved in cases such as $k = n - 1$ or when $k \geq \Delta(G) - 1$, where $\Delta(G)$ is the maximum vertex degree in G). This illustrates the fact that a problem can be subdivided into cases– classes of instances–some of which are hard, while others are easy.

Recall the dominating set problem. A *dominating set* of a graph $G = (V, E)$ is a set $D \subseteq V$ such that each vertex in V is either a member of D or adjacent to at least one member of D. The objective of the dominating set problem is to find a minimum cardinality dominating set. In terms of a decision problem:

Does graph G have a dominating set with at most k vertices?

This problem is easy for any fixed k, such as $k = 1$ or $k = 2$. To see this, note that if G has n vertices there are just $\left(\begin{array}{c} n \\ 2 \end{array} \right)$ different pairs of vertices. Using brute force, we can test whether or not each pair is a dominating set in polynomial time. But then note that such an approach will not run in polynomial time when, for example, $k = \frac{n}{10}$.

7.2 The Failure of Greedy Algorithms

Consider the following greedy algorithm for TSP: from the current city, you next visit the closest city that you have not yet visited. Though a logical approach, this algorithm does not always yield the optimal solution, and in fact it may produce a solution that is much longer than the optimal.

We can color a graph with the following algorithm. Let the colors be represented by integers 1, 2, ... Iteratively color the vertices, at each step, color vertex i with the smallest color that has not yet been assigned to one of its neighbors. This greedy algorithm colors the graph properly, but not necessarily with the minimum number of colors.

Recall the dominating set heuristic consider earlier: we iteratively chose the vertex with maximum degree (i.e., the vertex that dominates the most vertices) and deleted it and its neighbors. That algorithm does not always yield the optimal solution, even if the input graph is a tree! Nor does the modified algorithm that iteratively chooses the vertex in the graph that dominates the most vertices that have yet to have a neighbor in the dominating set. [Note that the second algorithm does not delete vertices after each iteration, rather vertices are simply marked as to whether or not they have been "dominated" by either being chose to be in the dominating set or by having a neighbor in the dominating set].

Exercise: Find a graph where the greedy coloring algorithm fails to use the minimum number of colors. Find a graph where either greedy dominating set algorithm fails to find a smallest dominating set.

Exercise: Prove that, for every graph G, there exists a permutation of the vertex set of G, such that when the vertices are considered by the greedy coloring algorithm in that order, an optimal coloring is produced. (Hint: Consider an optimal coloring and order/permute the vertices from lowest color to highest).

7.3 A Classification of Problems

"P" denotes the class of decision problems that can be solved (decided) in polynomial time. That is, for any problem in P, there is an algorithm for that problem whose worst-case running time is $O(n^c)$ where n is the input size and c is some constant.

"NP" denotes the class of problems whose solutions can be verified in polynomial time. ("NP" stands for "Non-deterministic Polynomial time"). The graph coloring problem, put as a Decision problem with $k = 4$ is an example of an NP problem:

"Can G be colored with 4 colors?"

This problem is in NP because any candidate solution to an instance of the problem can be verified in linear time. Here a solution is a particular coloring of the graph. We can think of verification being handled by a procedure:

```
program check-coloring($G$, $C$, $k$)

test if $G$ is $k$-colored by candidate coloring $C$

end;
```

Then if we could invoke a non-deterministic, omniscient "guesser", we could have the following polynomial-time algorithm to actually k-color a graph:

```
program color($G$, $k$)

1. Guess a coloring, $C$;

2. Test if $C$ is a $k$-coloring of $G$;

end;
```

If a k-coloring exists, our guesser would always guess one at step 1. and the verifying at step 2. would be what takes polynomial time. If there's no coloring, no guess would work. There are $O(4^n)$ different colorings. You could also employ 4^n different machines, each checking its own coloring, to check all the possible colorings.
The Partition Problem is another another example of a problem in the class NP. For any particular partition of a set of integers S into two non-empty subsets, it takes only linear time to determine whether the sum of the elements of one subset equals the sum of the elements of the other. However, in such a case, where n is the size of S, there are 2^n possible partitions of S.

P is a subset of NP. Any instance of a problem that can be solved in polynomial can be verified in polynomial time. This suggests the following diagram:

We note that there exist very hard problems beyond NP (e.g. PSPACE: class of problems decidable in polynomial space, EXPTIME and EXPSPACE

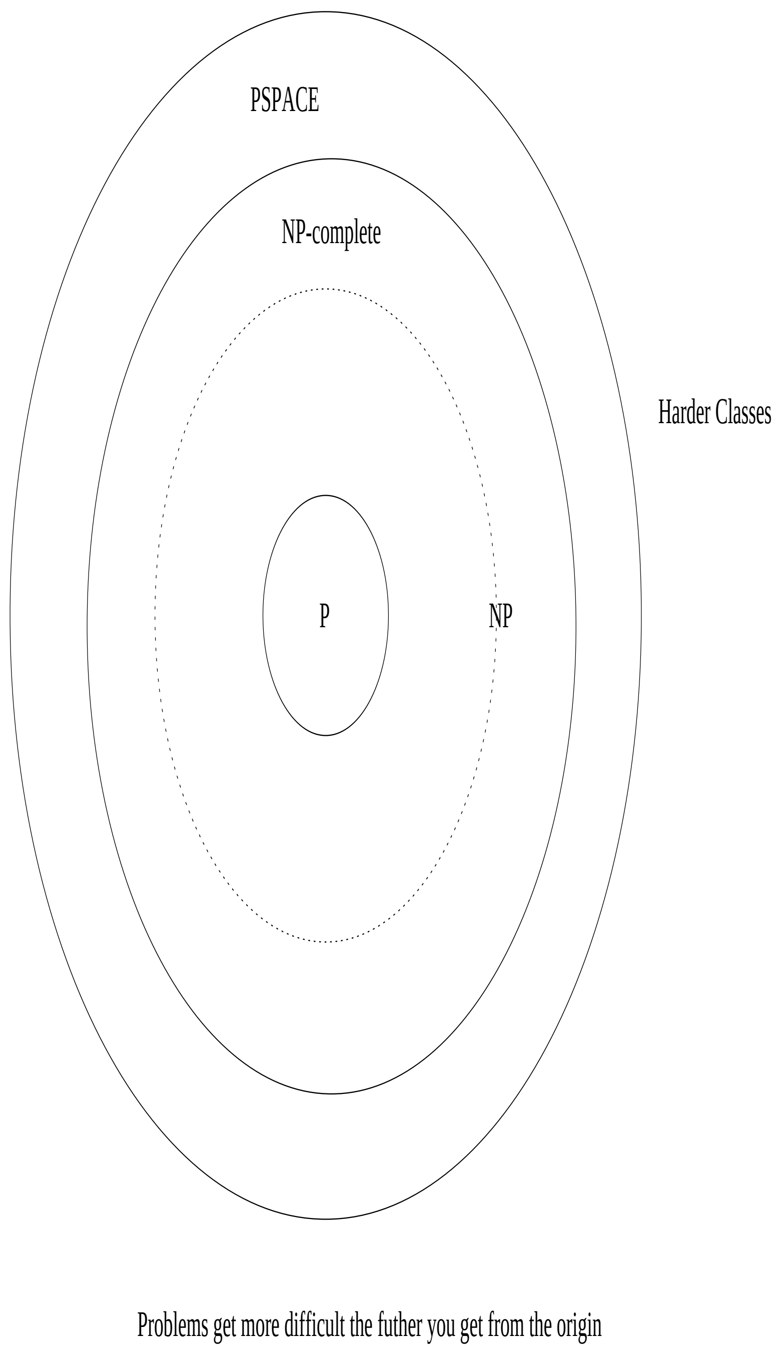

PSPACE
NP-complete
Harder Classes
P
NP
Problems get more difficult the futher you get from the origin

Figure 7.2: Complexity Classes

(exponential time and exponential space) as well as undecidable problems such as the Halting Problem for which no algorithms exist at all).

We will want to place problems in this diagram, to figure out where they "live" in it.

The most important open question in computer science is

$$P = NP?$$

(a.k.a the "P vs NP" problem). It is strongly believed that P and NP are distinct, i.e., P is a proper subset of NP, but no proof has been found as yet.

7.4 NP-completeness and Complexity Classes

7.4.1 Reductions

Let $G = (V, E)$. A *dominating set* of G is a subset $D \subseteq V$, such that every $v \in V$ is either in D or adjacent to at least one member of D. We are interested in finding a smallest dominating set in a graph (i.e. it is usually a minimization problem).

As a decision problem, we could ask:

"Does G have a dominating set of size less than or equal to k?"

Given G and k, if the answer is "yes" then there is a witness/proof to confirm this: the dominating set itself. Consider the"non-dominating set" problem:

"Does G have no dominating set of size less than or equal to k?"

In which case one seemingly cannot exhibit a concise "proof" that G has no small dominating set – the best we can do is enumerate all 2^n candidate solutions and show that none of them are in fact solutions. Interestingly, there do exist short proofs that a number is prime (though these are complex), whereas a short proof that a number is not prime simply consists of finding two factors of that number.

We are interested in the "hardest" problems in NP, which will term the NP-complete problems. Suppose π is the hardest problem in NP and we could solve π in polynomial time. Then we should be able to solve all problems in NP in polynomial time.

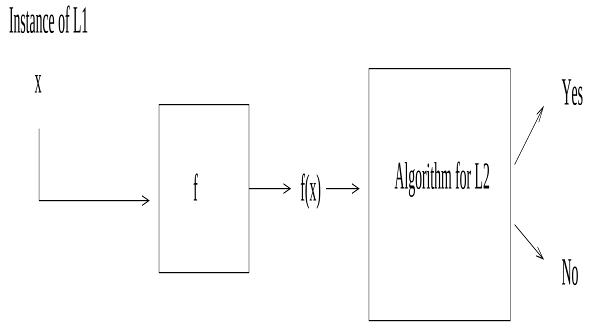


Figure 7.3: Reduction Process

A *Language*: the set of instances having “yes” answers for some decision problem π.

An an example, the (infinite) set L of all 3-colorable graphs. So the decision problem which asks if G is 3-colorable is equivalent to the question is G an element of L.

A *polynomial time transformation (reduction)* from L1 to L2 is a function f (computable in polynomial time) such that, for all x, x is an element of L1 if and only if $f(x)$ is an element of L2. If such an f exists, we say L1 is no harder that L2 and denote this as L1 $\leq_m$ L2.

Suppose we want to solve L1.

Suppose the running time of the algorithm for L2 is $O(n^3)$, and suppose f runs in $O(n^2)$ time. What is the running time for my new algorithm for L1 that is illustrated in Figure 7.3? Note that The output size of f is at most $O(n^2)$ (if the input size to L1 and thus f is $O(n)$). Therefore, the input to L2 is of size $O(n^2)$, so that the running time is $O(n^6)$.

Thus we have the following lemma.

Lemma 16 *If L2 $\in P$ and L1 $\leq_m$ L2 then L1 $\in P$.*
If L1 $\notin P$ and L1 $\leq_m$ L2, then L2 $\notin P$.

Let us give an example. Hamiltonian cycle (HC) is reducible to the traveling salesman problem (TSP).

Instance: $G = (V, E)$, such that $|v| = n$.
Question: Does G contain a simple cycle containing every vertex?

Recall that instances of TSP are complete, weighted graphs.

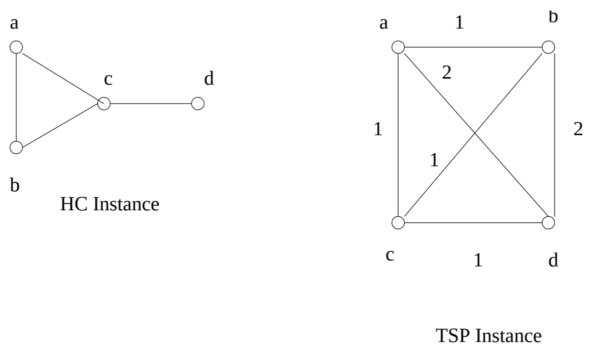


Figure 7.4: Reduction from HC to TSP

We construct f that maps HC to TSP: Given G, an instance of HC, construct a TSP instance (G' a.k.a. $f(G)$) with n vertices and distance(c_i, c_j)=1 if (c_i, c_j) is an element of E, distance (c_i, c_i)=0 and distance (c_i, c_j)=2 if (c_i, c_j) is NOT an element of E.

We ask if G' has a TSP tour (simple cycle with n vertices) of weight at most n. The answer is obviously "yes" if and only if G has a Hamiltonian cycle.

Two problems L1 and L2 are *polynomially equivalent* (equally hard) if L1 $\leq_m$ L2 and L2 $\leq_m$ L1. Note that $\leq_m$ is a relation and induces a partial order on the resulting equivalence classes:

P is the "base" of this partial order, since for all L1, L2 which are elements of P, it is easy to see that L1 and L2 are polynomially equivalent (a reduction f between them can simply solve the initial problem in polynomial time).

Also note that $\leq_m$ is transitive. Therefore, if L1 $\leq_m$ L2 and L2 $\leq_m$ L3, then L1 $\leq_m$ L3. This is like a composition of two functions (and in fact the f that reduces L1 to L3 is just the composition of the reduction that takes L1 to L2 and the reduction that takes L2 to L3).

7.4.2 Formal Definition of NP-Completeness and Examples of NP-Complete Problems

Formally, a language L (or a decision problem) is NP-Complete if (see Fig. 7.2)
1. $L \in NP$
2. $\forall L' \in NP \; L' \leq_m L$
If 2 holds, but not necessarily 1, we say L is NP-hard.

To show some problem π is NP-complete (i.e., difficult, if not impossible, to solve efficiently), we do the following:
1. Show $\pi \in NP$ (usually it's easy to do)
2. Show $\pi' \leq_m \pi$ for all $\pi' \in NP - complete$

Note that:

- All NP-complete problems are polynomially equivalent.
- P is an equivalence class under $\leq_m$.
- NP *is not* an equivalence class unless P=NP.

Recall that $\leq_m$ is transitive, so it suffices to show that

$$\pi' \leq_m \pi$$

for *any* single NP-complete problem π'. Depending on choice of π', the reduction of π' to π may be either hard or easy.
Cook's Theorem (*1971, also due to Levin*): SAT is NP-complete.

SAT stands for Satisfiability Problem.
Instance: A set U of boolean variables and a collection of clauses C over U.
Question: Is there a satisfying truth assignment for C?
Let's detail this. Let

$$U = \{u_1, u_2, ...u_m\}$$

If u_i is a variable, then u_i and $\bar{u}_i$ are literals. Clause is a subset of literals, e.g. $(u_1, \bar{u}_7, u_3)$ where all literals in the subset are connected by OR.
Example:

$$(a \vee b) \wedge (\bar{a} \vee b \vee \bar{d} \vee c) \wedge (\bar{b} \vee d \vee \bar{c}) \wedge (b \vee \bar{c})$$

We have 4 clauses. Does there exist truth assignment such that the formula is True? Obviously, the problem is in NP. In fact (omitting the proof)

$$\pi' \leq_m SAT \; \forall \pi' \in NP.$$

7.4.3 Some More NP-complete Problems

- **3-SAT:** a version of SAT in which each clause contains exactly 3 literals.
  Exercise: Prove 2-SAT is in P. Notice the jump in difficulty when going from 2-SAT to 3-SAT.

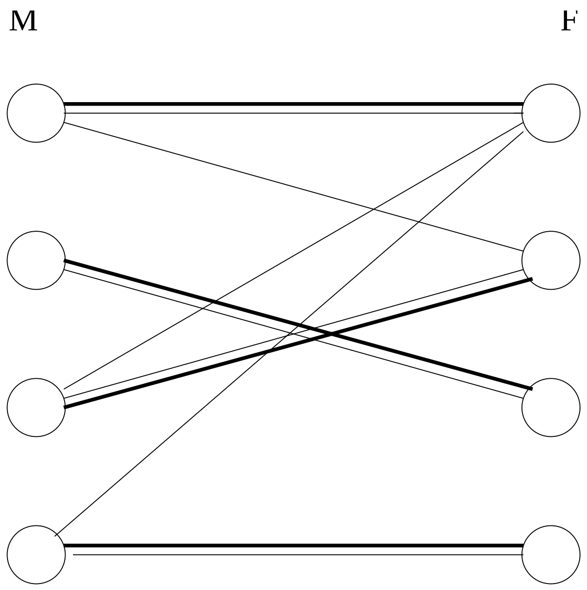


Figure 7.5: 2-dimensional Matching Problem

- **3DM** (3-dimensional matching):
  Instance: Set
  $$M \subset W \times X \times Y$$
  where W, X, Y are disjoint and $|W| = |X| = |Y| = q$
  Question: Does M have a matching subset $M' \subset M$ such that $|M'| = q$ and no 2 elements of M' agree in any coordinate?

  First look at 2DM problem $\in P$ (see Fig. 7.5). Edges connect people willing to be married. Is it possible to marry them avoiding bigamy?

  Now look at 3DM (see Fig. 7.6). Can we 'marry' males, females and dogs so they are happy?

- **Vertex Cover:** Does undirected graph $G = (V, E)$ contain a vertex cover of size *at most* k? A *vertex cover* is a subset of vertices such that every edge contains at least one endpoint in the cover. An example is shown on Fig. 7.7.

  In this example vertex cover is also a dominating set, *but* there is a dominating set in this graph (consisting of circled vertices) that is *not* a vertex cover. Also, if we add a vertex not connected to the rest of the graph, the black vertices still form a vertex cover, but *not* a dominating set!

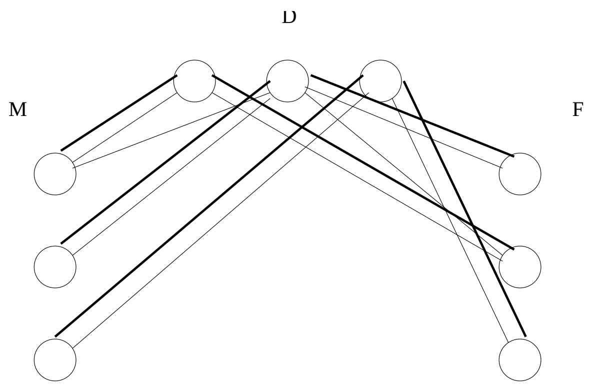


Figure 7.6: 3-dimensional Matching Problem

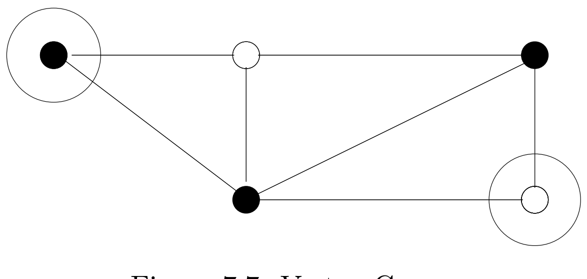


Figure 7.7: Vertex Cover

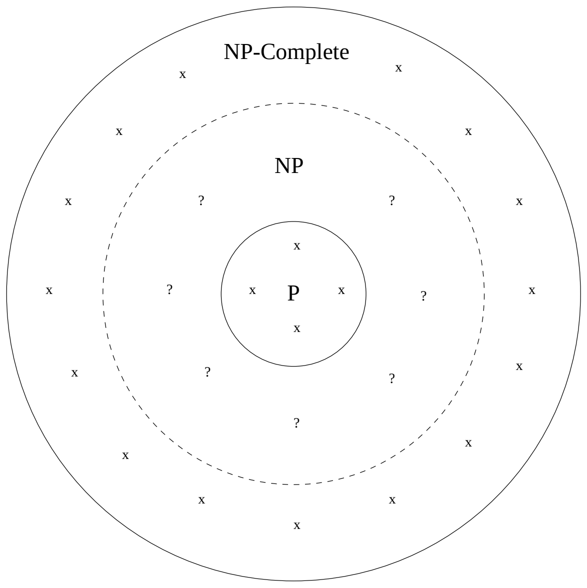


Figure 7.8: Are there problems in NP that are not in P and not in NP-complete?

- **Clique:** Does graph G contain a complete subgraph of size *at least* k?
- **Hamiltonian Cycle:** Does G (directed or undirected) have a simple cycle of length n?
- **Hamiltonian Path:** Does G (directed or undirected) have a simple path of length $n-1$?
- **Partition Problem**.
- **TSP** (Traveling Salesman Problem).
- **Graph Coloring** (Can G be colored with at most k colors?)

7.4.4 Remarks on Relationship Between Problem Classes

Now we know problems that 'live' in areas marked with 'x' on Figure 7.8. But who lives in the '?'-marked area?

If $P \neq NP$, then there *do* exist problems not in P that are *not* in NP-complete, but *are* in NP (in fact, an infinite number of distinct equivalence classes between P and NP-complete would exist [22]).

Here are some 'suspects' for this class of problems:

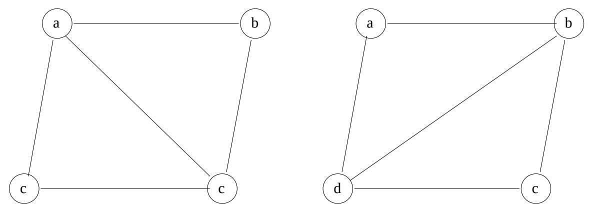


Figure 7.9: Example of Isomorphic Graphs

- **Primality**: Is a given integer N prime? (It has been know since the 1970's that this problem lies in $NP \cap$ co-NP and so is not NP-complete unless $P = NP$. An example of a co-NP-complete problem is "Does boolean formula S have no satisfying assignments?")

  However, in 2002, Manindra Agarwal stunned the computer science community by announcing (with two of his students) a beautiful, short, and elegant deterministic polynomial-time algorithm that determines whether or not an integer is prime [2]. This resolves a problem that had stumped science for thousands of years! Thus Primality belongs to the class P, and is no longer a suspect to lie between P and NP-complete!

  On the other hand, the precise complexity of the related problem of FACTORING an integer, remains unresolved.

- **Graph Isomorphism**: Are G_1 and G_2 isomorphic? See Fig. 7.9.

Graph Isomorphism seems very hard, but is not NP-complete unless the polynomial time hierarchy collapses to a very low level (which is thought highly unlikely [5]).

In conclusion, we also note that problems are very sensitive to parameters and specific graphs. For instance, "Does G have a vertex cover of size at most k?" is easy when k is small (i.e. a constant). In such cases we can solve the problem in polynomial time by enumerating all possible solutions of size k or less (but this is not feasible if k is, say, $\frac{n}{\log n}$).

7.4.5 3-SAT is NP-complete

Consider the 3-SAT problem

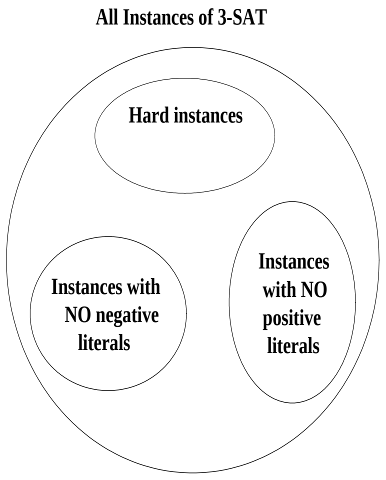

All Instances of 3-SAT
Hard instances
Instances with NO negative literals
Instances with NO positive literals

Figure 7.10: Categorization of 3-SAT

$(a \vee b \vee \vee c) \wedge (\overline{a} \vee b \vee \overline{d}) \wedge (\overline{b} \vee \overline{c} \vee d) \wedge (\overline{a} \vee b \vee \overline{c})$

Once can see that this instance is satisfiable: if a is True, b is True, and c is False.

Suppose that the instance is

$(a \vee b \vee c) \wedge (b \vee c \vee d) \wedge (a \vee b \vee e) \wedge (a \vee b \vee d)$

in which there are no negated literals. It is clearly satisfiable in linear time. Similarly a set with no positive literals also can be decided to be satisfiable or not in linear time. The hard instances are that have mixed negative and positive literals, are not satisfiable in polynomial time. The categorization of the world of 3-SAT is shown in 7.10.

Even though some hard instances picked at random might yield quick solution by some (lucky) algorithm, in general this category contains an infinite set of problems that will force any algorithm to use super-polynomial time (assuming $P \neq NP$).

(1) 3-SAT $\in$ NP

It is easy to guess a truth assignment and verify in polynomial time whether or not it satisfies the instance.

(2) 3-SAT $\in$ NP-hard

We can show that a polynomial time reduction is possible from any SAT instance to a 3-SAT instance

i.e. SAT $\leq_m$ 3-SAT

Let E be an arbitrary instance of instance of SAT:

The strategy for the reduction is the following

• Replace each clause C $\in$ E which does not have 3 literals with "a few" clauses with 3 literals.

Let C = $(x_1, \vee x_2 \vee \ldots x_k)$ where $k \geq 4$, or there are at least 4 literals in C.

Now introduce new variables $y_1, y_2, \ldots y_{k-3}$ to transform C in to group of clauses with 3 literals as shown below

C' = $(x_1, \vee x_2 \vee y_1) \wedge (x_3 \vee \overline{y_1} \vee y_2) \wedge (x_4 \vee \overline{y_2} \vee y_3) \ldots (x_k - 1 \vee x_k \vee \overline{y_k - 3})$

The expansion of C to C' is done in polynomial time since the total number of dummy variables used is less than k and each dummy variable appears in at most two clauses.

Example:

C = $(x_1 \vee \overline{x_2} \vee x_3 \vee x_4 \vee \overline{x_5})$

C' = $(x_1 \vee \overline{x_2} \vee y_1) \wedge (x_3 \vee \overline{y_1} \vee y_2) \wedge (x_4 \vee \overline{x_5} \vee \overline{y_2})$

First and last clauses have two original literals (x's) and all other clauses have only one original and two dummies. No new dummy variable is used in the last clause, the one used is the negation of the same dummy from the previous clause (y_2 in C' above).

Claim: C' is satisfiable if and only if C is satisfiable.

• If C is satisfiable then some x_i is true (or x_i is false and it appears in C in negated form)

• In this case we can set all the y_i's in C' such that all clauses in C' are satisfied. As an example let us suppose x_3 = True. Then setting y_1 = True and y_2 = False satisfies C'.

In general, if x_i = True, we set $_1, y_2, \ldots y_{i-2}$ to True and the remaining y's to False. So, all the y's preceding x_i are set to be True and all the subsequent are set to be False.

-If C' is satisfiable then we claim at least one of the x_i's must be True.

Suppose all the x_i's are "false" (non-negated x_i's are false, negated x_i's are true, i.e. C has no witnesses). Now we have to satisfy C' using only the y's. in our example, We must have y_1 = True, y_2 = True and so on up to $y_k - 3$ = True. But y_{k-3} must be false to satisfy the last clause. Hence a contradiction.

Now consider clauses with less than 3 literals

C = $(x_1 \vee x_2)$
C' = $(x_1 \vee x_2 \vee z) \wedge (x_1 \vee x_2 \vee \overline{z})$
C = (x_1)
C' = $(x_1 \vee y \vee z) \wedge (x_1 \vee \overline{y} \vee z) \wedge (x1 \vee y \vee \overline{z}) \wedge (x_1 \vee \overline{y} \vee \overline{z})$

So x_1 has to be true to for C' satisfiable in the latter instance since all other combinations of $\overline{y}$ and z are present in C'.

Key Point 3-SAT formula f(E) is satisfiable if and only if the original SAT E instance is satisfiable. The one-to-one mapping of the satisfiable instances from SAT to satisfiable instances of 3-SAT and the non-satisfiable instances from SAT to non-satisfiable instances of 3-SAT by the polynomial function f(E) is shown in 7.11

Also note that we could reduce a SAT instance (or a 3-SAT instance) of length n to an instance of 2-SAT (and recall that 2-SAT is solvable in polynomial time), but the resulting formula could be of exponential length in n, so it would not be a polynomial time reduction.

It is also interesting to consider optimization versions of 3-SAT and 2-SAT in which which wish to satisfy as many clauses as possible. The related decision problems ("Can we satisfy at least k clauses in the 3-SAT (2-SAT) instance?") are NP-complete for both 3-SAT and 2-SAT! So it is only the special case when k is equal to the total number of clauses in which we can solve MAX 2-SAT in polynomial time! [Although we can also solve MAX 2-SAT in other cases in polynomial time also, when $k \leq 1$, for example].

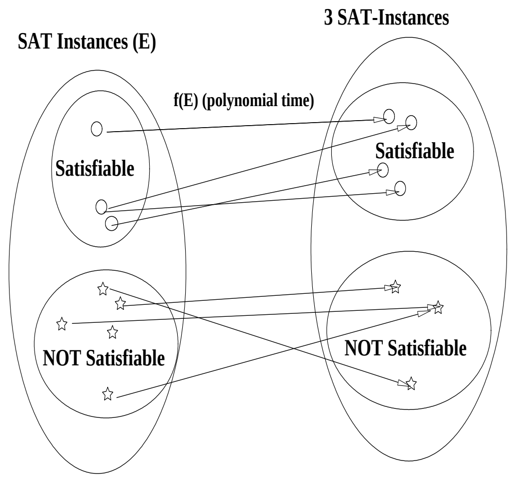


Figure 7.11: One-to-One mapping of SAT to 3-SAT

7.4.6 Proving NP-completeness by Restriction

Sometimes we can show that a problem we are faced with is NP-complete by means of a short-cut method called *restriction*. [Rather than doing an involved reduction].

Suppose we have a problem T and suppose by fixing/restricting certain parameters of the problem T , we get a known NP-Complete problem Q. Then T is clearly NP-hard. That is, a subset of the possible instances of T comprise all possible instances of Q. See Figure 7.12.

Restriction is best illustrated by the following examples:

Hitting Set
INSTANCE: A collection C of subsets of a set S, positive integer k.
QUESTION: Does S contain a "hitting set" for C of size k or less.
A hitting set is a subset S' of S , $|S'| \leq k$ such that S' contains at least one element from each subset in C.

For example:
S = { 1,2,3,4,5}
C = { {1,2,3}, {1,5}, {3,2,4} }
$k = 2$. Then S' = {1,2} is a Hitting set for C.

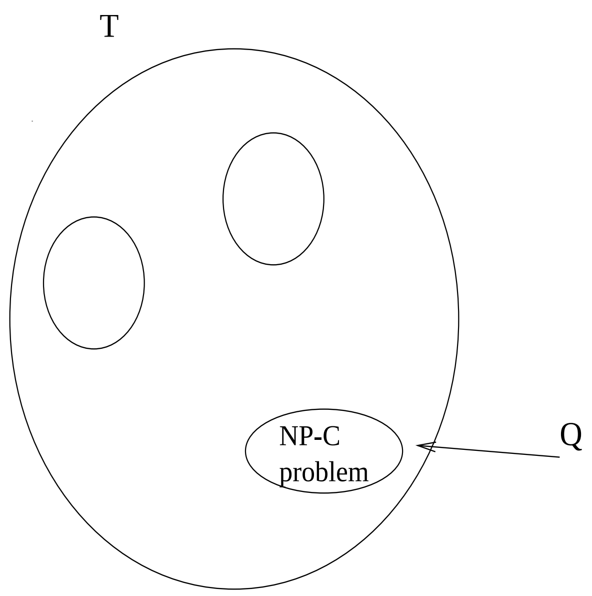


Figure 7.12:

To show Hitting Set (HS) is NP-Complete, we restrict HS instances so that each subset in C has exactly 2 elements. This is now just the vertex cover problem. In other words, each subset in C is an edge and each element of S is a vertex. So edges in C must be covered by vertices in the HS. Therefore HS is NP-Complete (since we can easily see that HS is in NP).

Subgraph Isomorphism
INSTANCE: Two graphs G and H.
QUESTION: Does G contain a subgraph isomorphic to H?

Restrict H to be a CLIQUE (complete graph) with k vertices. And we are now simply asking if G has a clique of size at least k. Note that this is seemingly harder than Graph Isomorphism because G has a lot of subgraphs that you have to compare with H (since we may assume H is somewhat smaller than G). Whereas in the Graph Isomorphism problem, the two given graphs are of the same size and order.

Bounded degree spanning tree
INSTANCE: G = (V,E), integer k.
QUESTION: Does G contain a spanning tree T such that every vertex in T

has degree at most k.

In general this is finding a minimum spanning tree that minimizes the maximum degree of any node in the spanning tree. If we restrict $k = 2$ then this is just the Hamiltonian path problem.

Multiprocessor Scheduling
INSTANCE: Finite set A of jobs. Each job has a positive integer length $l(a)$ and there are a finite number, m, of independent identical processors and some deadline $d \geq 1$.
QUESTION: Can we schedule the jobs so that every job is finished by the deadline.

Example:
A = {1,2,4,5,9,2} (these are the lengths of the jobs, $l(a)$)
3 processors, $d = 9$.

Solution: Yes we can meet the deadline: Processor 1 runs jobs of length 1, 2, 2; processor 2 runs jobs of length 5 and 4; and processor 3 runs a job of length 9.

Claim: If we restrict $m = 2$ and set d equal to the sum of the length of all the jobs divided by 2, then this is the partition problem.

To see this in a more general setting, consider the following question. What happens if $d < \frac{\sum_{a \in A} l(a)}{m}$? Then it is not possible to meet the deadline. Why is that true?

7.4.7 Some Reductions

We illustrate some standard reduction techniques. The key is to first choose a known NP-hard problem to reduce to your target problem and then to understand the "characters" in each problem and their relationships to one another (e.g. in SAT, the characters are clauses and literals; literals satisfy clauses and must be assigned values in such a way that both a literal and its negation are not both true).

Vertex Cover

Instance: Undirected graph $G = (V, E)$, integer k
Question: Does G have a vertex cover of size at most k? That is a set $C \subseteq V$

such that each $e \in E$ is incident to at least one member of C.

Now it can be clearly seen that Vertex Cover is in NP
Given some subset of vertices for a graph G , test to see if this size is at most k and then check if this vertex set satisfies the criteria of covering every edge.

We show 3-SAT can be reduced to Vertex Cover in polynomial time (i.e. Vertex Cover is at least as hard as 3-SAT).

Let our 3-SAT instance have n variables. Let
$U = \{u_1, u_2, u_3, \dots, u_n\}$

be the variables of a Formula F. There are m clauses in F and each clauses has 3 literals.

C is the set of Clauses

$$C = \{c_1, c_2, c_3, \dots, c_m\}$$

We are going to construct a graph $G = (V, E)$ and integer k such that G has a Vertex Cover of size at most k, if and only if F is satisfiable.

For each variable u_i contained in U
we are going to have 2 vertices joined by an edge;
let's call them u_i and $\overline{u_i}$

For each clause c_j in C We create a "triangle" subgraph (a complete graph on three vertices $a_1[j], a_2[j], a_3[j]$).

Then for a clause $c_j = (x_j \vee y_j \vee z_j)$ we put edges ($a_1[j], x_j), (a_2[j], y_j), (a_3[j], z_j)$ in the graph. Thus we connect the i^{th} clause vertex in clause j to the literal vertex corresponding to the i^{th} literal in clause j of F.

We are going to let $k = n + 2m$, where n = number of variables and m = number of clauses.

Illustrating reduction with an example.

We are going to have $U = \{u_1, u_2, u_3, u_4\}$ and $C = (u_1 \vee u_3 \vee u_4) \wedge (\overline{u_1} \vee u_2 \vee \overline{u_4})$

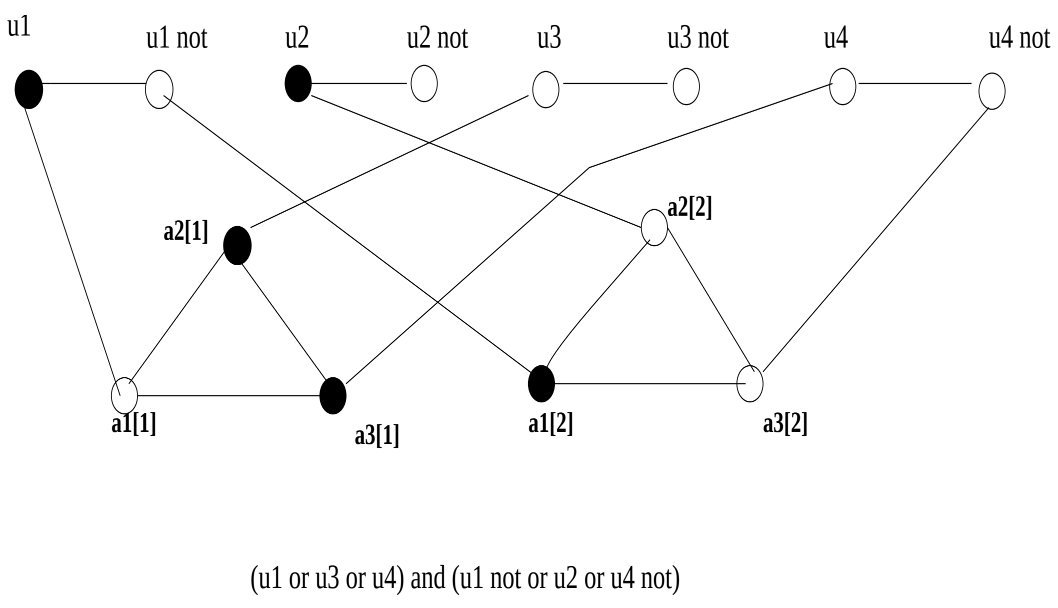


Figure 7.13: 3-SAT reduction to Vertex Cover

See the Figure 7.13 for the construction.

Any Vertex Cover of this graph should have one of each of the following pairs of vertices:
u_1 and $\overline{u_1}$,
u_2 and $\overline{u_2}$,
u_3 and $\overline{u_3}$,
u_4 and $\overline{u_4}$.

Any Vertex Cover should have at least two of the vertices in each triangle. That is, at least 2 of $a_1[1]$, $a_2[1]$, $a_3[1]$, and at least 2 of $a_1[2]$, $a_2[2]$, $a_3[2]$.

Therefore we need at least n vertices in the literals to take care of the Vertex Cover (to cover all literal edges) and we need at least $2m$ vertices from the triangles to take care of Vertex Cover (to cover all the triangle edges edges). Of course, there are additional edges between the literals and triangles!

We now show F is satisfiable if and only if G has a Vertex Cover of size less than or equal to k.

$\leftarrow$. Suppose $V' \subseteq V$ is a Vertex Cover of G with $|V'|$ less than or equal to

k. Since V' must contain at least one vertex from each edge corresponding to a pair of literals and two vertices from each triangle corresponding to a clause, we must have $|V'| \geq n + 2m = k$.

Thus by setting u_i = True , if u_i belongs to V' and u_i = False, if $\overline{u_i}$ belongs to V' we have a truth assignment.

Now we have to show that this truth assignment is satisfying truth assignment. This truth assignment satisfies F because one of the the inter-component edges (between literals and triangles) is covered by a u_i or a $\overline{u_i}$ for each clause, and two by $a_i[j]$ vertices. And this u_i or $\overline{u_i}$ vertex is a witness for the clause corresponding to $a_i[j]$. that is every clause has a witness. By this we mean what is shown in Figure 7.14.

The Vertex Cover V' includes one vertex for each pair of literal vertices (the literal whose value is true – u_i if it is assigned true and $\overline{u_i}$ if u_i is assigned false) and two vertices from each triangle (so as to cover the inter-component edges (edges between literals and triangles) as shown in Figure 7.14). So consider the vertex cover in our example consists of the shaded vertices (and we see that u_1 and u_2 are assigned true).
$\rightarrow$. Since F is satisfiable , each clause has a witness. Construct a vertex cover, V', as before (if u_i is assigned true then place u_i in the cover, else place $\overline{u_i}$ in the cover. Finally, place two vertices from each triangle in the cover in such a way as to cover the inter-component edges not covered by the witness for the clause. Hence $|V'|$ is less than or equal to k .

Clique

Instance: Undirected graph $G = (V, E)$, integer k
Question: Does G have a clique of size at least k? That is a complete subgraph with k or more vertices.

It is easy to see Clique is in NP. Given a solution , it is easy to verify that the solution in polynomial time. We now show SAT is reducible to Clique.

Let $F = F_1 \wedge F_2 \wedge F_3 \ldots \wedge F_m$ (m clauses).

Let $k = m$. Suppose $F_i = (x \vee y \vee z \vee w)$. It will be a different flavor of reduction from that of the Vertex Cover. The central character here is the Clauses.

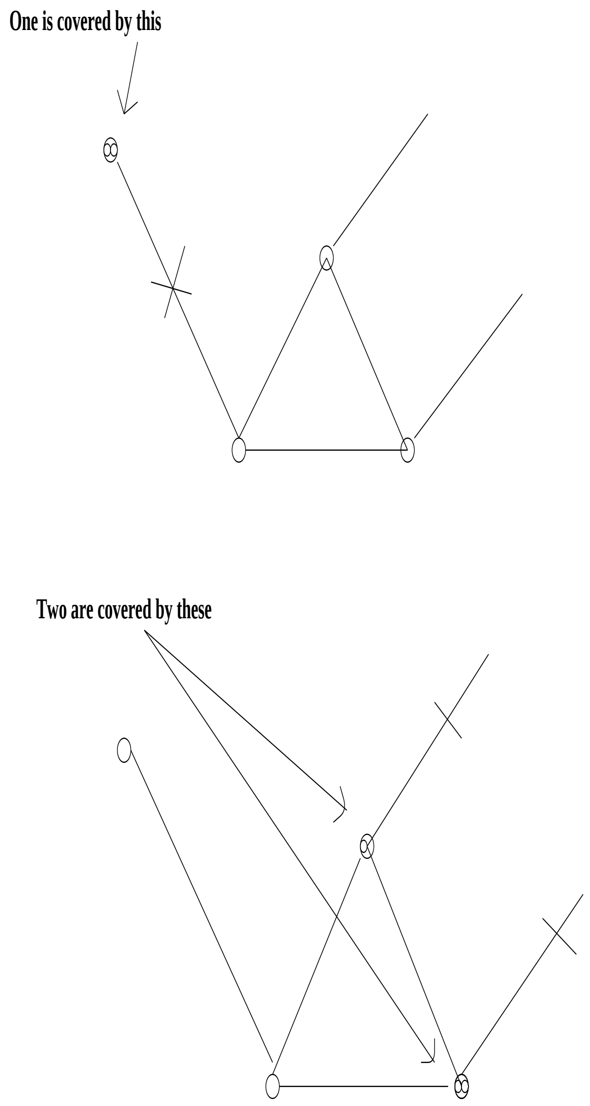


Figure 7.14: Covering Edges

We associate four vertices with this clause F_i –one for each literal in the clause. Each one of these clauses will have a column of vertices having a number of vertices equal to the number of literal in the clause.

To connect the vertices: vertices from same column are not connected (so we get an m-partite graph). Vertices from different columns are connected unless they represent the same variable in the negated form.

Here is an example.

$F = (x \vee y \vee \overline{z}) \wedge (\overline{x} \vee \overline{y} \vee z) \wedge (y \vee \overline{z})$.

We have 3 Clauses. So our $k = 3$ in this case. This example is obviously satisfiable: set $y =$ True, $z =$ True.

This reduction (building this graph) can obviously be done in Polynomial time in $|F|$. An example is shown in Figure 7.15. The columns are contained in dashed lines and a clique is shaded.

Now we claim G has a Clique of Size $k = m$, if and only if F is satisfiable

$\leftarrow$ Suppose F is satisfiable that means (each clause has a witness)

So in our example we can have $x =$ True, $y =$ True, $z =$ True.

Each Clause has a witness which implies each column has a TRUE vertex which implies each column will contain one vertex in a clique if size m.

Since these vertices must have edge between them (thee are in different columns and cannot represent a literal and its negation) and thus form a Clique of size k.

$\rightarrow$ Suppose there is a Clique of size $k = m$, obviously each edge from the clique must come from a different column. Assign values in the truth assignment so that the literal corresponding to these vertices in the clique simplify to the value True (and this is necessarily a legal truth assignment) and we have a satisfying truth assignment. Therefore Clique is NP-Complete.

Exercises: Prove Independent Set is NP-complete. Prove Clique if NP-complete using a reduction from Vertex Cover.

Dominating Set

Instance: Undirected graph $G = (V, E)$, integer k
Question: Does G have a dominating set of size at most k? That is a set

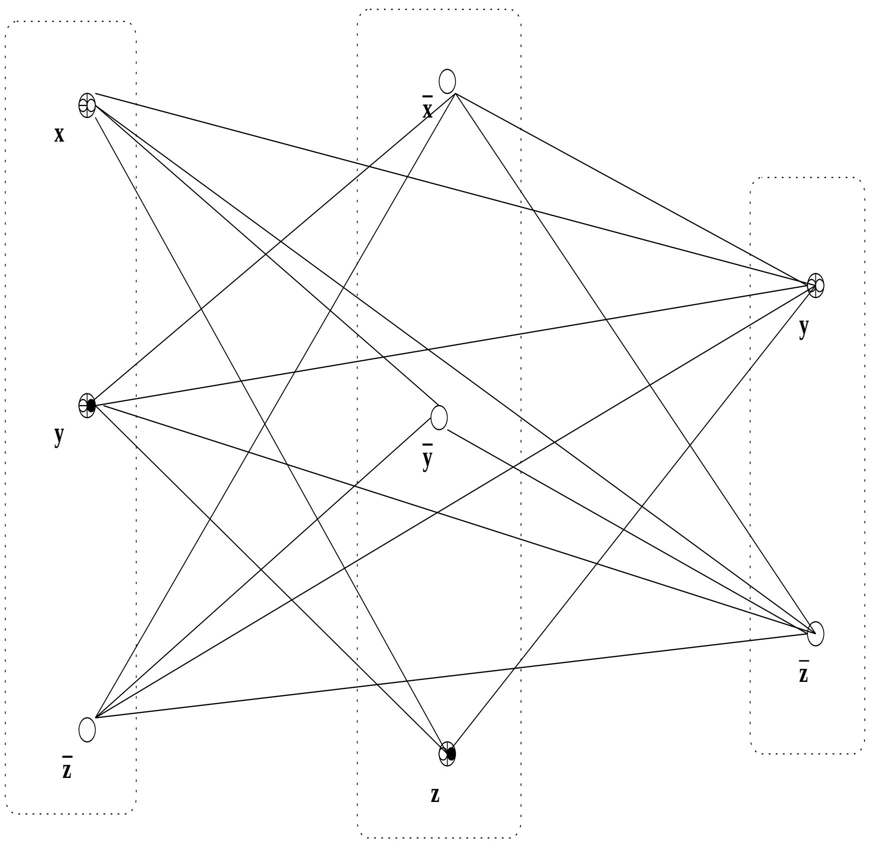


Figure 7.15: SAT Reduction to Clique

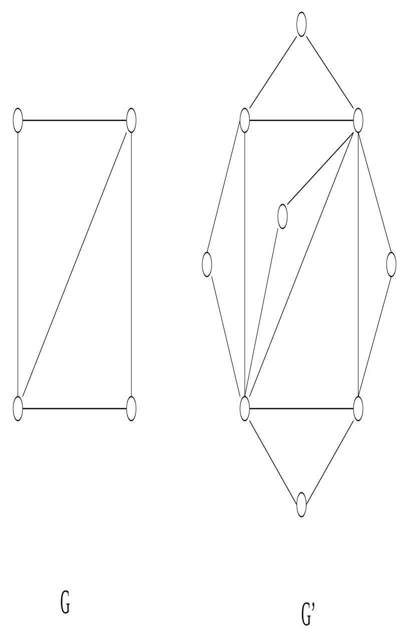


Figure 7.16:

$D \subseteq V$ such that each $v \in V$ is either a member of D or adjacent to at least one member of D.

It is easy to see than Dominating Set is in NP, since we can guess a set D' of vertices, verify that $|D'| \leq k$ or not and check in polynomial time whether or not every vertex in $V - D'$ is adjacent to a member of D'.

We do a reduction from Vertex Cover, which we know to be NP-complete. Let $G = (V, E), k$ be an arbitrary instance of Vertex Cover. Construct an instance $G' = (V', E'), k$ of Dominating Set as follows. Take G and remove any isolated vertices. For each edge $(u, v) \in E$ and a new vertex uv and two new edges (u, uv) and (v, uv) as shown in Figure 7.16. This reduction can easily be done in $O(V + E)$ time.

We claim G' has a dominating set of size k if and only of G has a vertex cover of size k.

$\rightarrow$. Let D be a dominating set of G' of size at most k. If D contains any uv type vertices, they can be replaced by either u or v with no harm to or increase in the size of the dominating set. So we can assume that D contains only u type vertices (those that correspond to vertices from G). But since D dominates all the uv type vertices also, it must contain at least one vertex from each (u, uv) type edge, hence D will cover all the (u, v) type edges (since each (u, uv) type edge corresponds to a (u, v) type edge). Hence D is a vertex cover of size at most k for G.

$\leftarrow$ If C is a vertex cover of size at most k, then each edge is covered by a vertex in C. This implies that each vertex in V is dominated by some vertex in C. But it also implies that C will dominate all the uv type vertices in G', since $(u, v) \in E$ and thus at least one of $u, v \in C$.

Thus VC $\leq_m$ Dominating Set. Note that our claim is equivalent to saying if G' has a dominating set of size at most k then G has a vertex cover of size at most k and if G' does not have such a dominating set, then G does not have such a vertex cover.

7.5 Approximation Algorithms

7.5.1 Definitions

Given that we cannot hope to always optimally solve NP-complete problems efficiently (unless $P = NP$), we resolve ourselves to four possibilities:

1) Hope we are given "easy" instances of the problem (such as 2-SAT, dominating set in a tree, vertex cover in a grid, etc.). It is an active research area to find the "threshold" at which particular problems become NP-complete. For example, Vertex Cover remains NP-complete in graphs having maximum degree three. Certain classes of graphs such as trees, series-parallel, bounded-treewidth, grids are so structured that many NP-complete problems become easy when restricted to those cases.

2) Run an exponential time algorithm and hope we stumble upon the correct solution quickly.

3) Hope one of the parameters is fixed in such a way as to allow an efficient solution (for example, 2-coloring a graph). A rigorous study of this concept in which efficient is carefully defined (so as to rule out strategies that run in time exponential in the parameter – such as with determining if a graph has a dominating set of size at most 20 by brute force, i.e. looking at all vertex sets of size at most 20) was initiated in [12].

4) Settle for a near-optimal solution in polynomial time. Of course, polynomial time heuristics may oftentimes yield the optimal solution, but will also yield non-optimal solutions to many or most instances.

The latter idea is what we will focus on. An *approximation algorithm* is a polynomial time algorithm A for an NP-hard problem whose goal is to produce a "near-optimal" solution for a problem Π. We shall denote the value of the solution produced by A on an instance I as $A(I)$ and the optimal solution as $OPT(I)$. Let us consider various notions of "near-optimal," that is, we wish to provide some guarantee as to how close our solution will be to the optimal (in the worst case).

Absolute Approximation Algorithms

We will use the term *instance* to mean an input to a specific problem. A polynomial time algorithm A for problem Π has an *absolute performance guarantee* if for all instances I if Π we have that $|A(I) - OPT(I)| \leq c$ for some **constant** c. That is, the value of the solution returned by A is guaranteed to be within an additive constant of the optimal solution, for all possible inputs. This is the strongest type of performance guarantee we can have.

Not surprisingly, very few NP-complete problems are known to have approximation algorithms with absolute performance guarantees. Let us see three examples.

Planar Graph Coloring It is known that planar graphs can be colored with at most four colors; and this can be done in polynomial time [28]. However it is NP-complete to decide if a planar graph can be colored with at most three colors (and obviously we can decide in linear time if a graph can be colored with two colors by using a modified breadth-first search). Hence we have an algorithm to color a planar graph with either the optimal number of colors or one more than the optimal, an absolute performance guarantee of one.

Edge Coloring We wish to color the edges of a graph so that every pair of edges incident upon a common vertex has a different color. To decide if k colors suffice to edge color a graph is an NP-complete problem. However, it is known by Vizing's theorem that the number of colors needed is either δ or $\delta + 1$ where δ is the maximum degree of any vertex in the graph. This theorem leads to a polynomial time algorithm to edge color a graph with at most $\delta + 1$ colors (though the edge coloring number of such a graph might be δ or $\delta + 1$. Hence we have an absolute performance guarantee of one.

Note that in the last two problems, it was easy to get an absolute performance guarantee because we had a concrete upper bound on the value of the optimal solution. This is not the case with the next problem.

Bounded Degree Spanning Tree We wish to find a spanning tree T of a graph G such that the maximum degree of any vertex in T is as small as possible. This is easily seen to be NP-complete by restriction to Hamiltonian Path. It was shown in [16] that there is an approximation algorithm with absolute performance guarantee of one for this problem.

For many NP-complete problems, it is easy to prove that finding a solution that is within an additive constant of the optimal is as hard as finding an optimal solution. Here is how a typical proof goes.

Suppose we had an approximation algorithm A for Vertex Cover with absolute performance guarantee c. That is, this polynomial time algorithm allegedly find a vertex cover using at most c more vertices than the optimal. We show how to use this polynomial time algorithm A to determine the optimal solution for a graph G. We wish to determine if G has a vertex cover of size at most k.

Make $c + 1$ disjoint copies of G, call this (disconnected) graph G'. Obviously, G has a vertex cover of size at most k if and only if G' has a vertex cover of size at most $k(c + 1) = kc + k$. Run A on G'. Suppose G has a vertex cover of size k. Then A returns a vertex cover C of size at most $kc+k+c < (c+1)(k+1)$. Thus is must be that C contains at most k vertices from one of the copies of G that comprise G'. Hence we can determine that G has a vertex cover of size at most k in polynomial time (since the size of G' is polynomial in the size of G). Now suppose G does not contain a vertex cover of size k. Then A returns a vertex cover for G' of size at least $(k + 1)(c + 1) = kc + k + c + 1 > kc + k + c$ and we can quickly conclude that G must not have a vertex cover of size at most k.

See additional examples of the proof technique in [17].

Exercise: Construct such a proof for Clique showing that there cannot exist a polynomial time algorithm with absolute performance guarantee. (Hint: Make $c + 1$ copies of the graph; but disjoint copies won't suffice).

Relative Performance Guarantees

Since so few problems have absolute performance guarantees, we shall examine a weaker guarantee. Our objective being to classify the NP-complete problems with regard to how hard they are to approximate (find those that have "good" approximation algorithms and those that do not). By good we shall mean that that have a constant relative performance guarantee.

We say A has a *relative performance guarantee* (a.k.a *performance ratio*) $f(n)$ if, for all inputs I of a minimization problem Π of size n we have that $\frac{A(I)}{OPT(I)} \leq f(n)$. If Π is a maximization problem, we consider the reciprocal of this ratio. If $f(n)$ is a constant c, that is if $\frac{A(I)}{OPT(I)} \leq c$ for all instances I, we say c is a constant performance ratio. A constant performance ratio (e.g. 2) is usually thought of as good.

In addition there are a few problems (such as Knapsack) that allow a certain type of approximation algorithm called an *approximation scheme*. These are approximation algorithms that have performance ratio of $1 + \epsilon$ for any $\epsilon > 0$ that the user desires! The running time of approximation schemes generally depends on ϵ, so there is a time/performance trade-off. If the running time is polynomial in $\frac{1}{\epsilon}$ (which is the most desirable case) we say the algorithm is a *fully polynomial time approximation scheme*. We note that it has been proved (via the PCP theorem, see below) that no MAXSNP-hard problem can have an approximation scheme unless $P = NP$. MAXSNP is a wide class of problems such as Vertex Cover, MAX-2SAT, MAX-CUT, etc., see for example [26].

But we will also see that there are problems that cannot have a constant performance ratio unless $P = NP$ (in which case they could be optimally solved in polynomial time). Many of such "hardness"of approximation proofs are based on the PCP theorem.

7.5.2 Vertex Cover

Recall that a vertex cover is a subset of the vertices that "covers" every edge in the graph. An obvious approach to an approximation algorithm is a greedy one: choose the vertex of maximum degree, delete that vertex and the edges it covers, and repeat until all edges have been covered. It is easy to verify that this method will produce the optimal vertex cover for lots of graphs. But not all! In fact, the performance ratio of this algorithm is known to be $O(\log n)$ (where the graph has n vertices) even in bipartite graphs! See the graph in Figure 7.17 as an example. What are the sizes of the optimal and approximate vertex covers in this example? Note that in

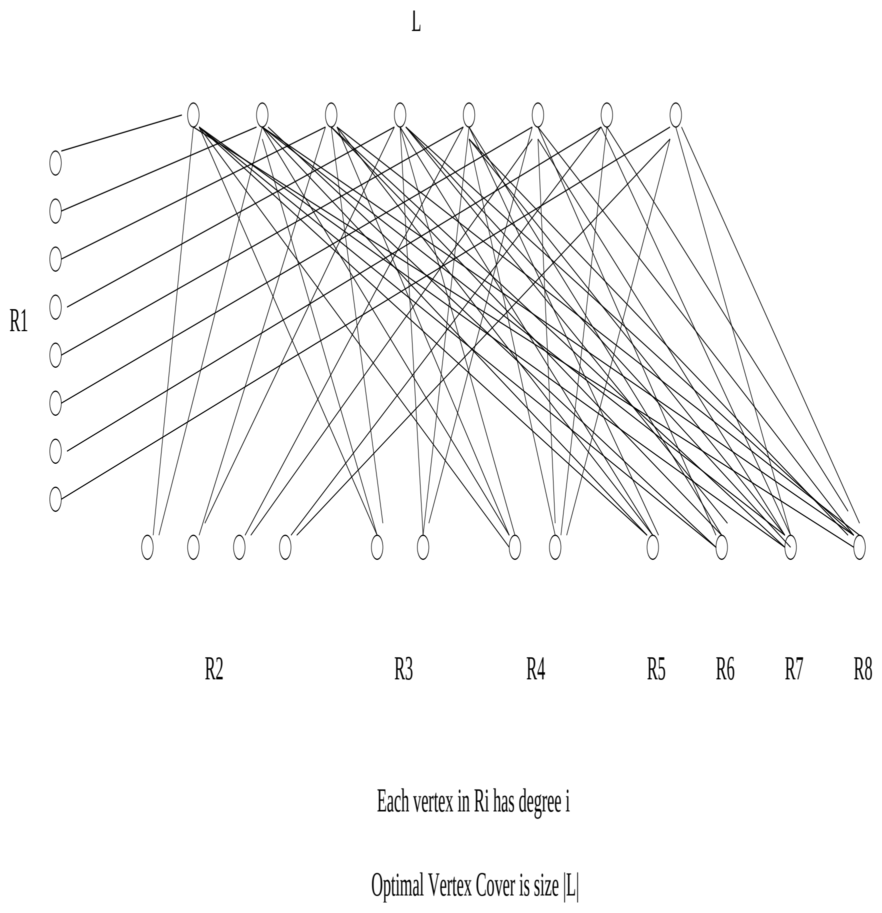

L
R1
R2
R3
R4
R5
R6
R7
R8
Each vertex in Ri has degree i
Optimal Vertex Cover is size |L|

Figure 7.17: Greedy is Bad for Vertex Cover

general $|R_i| = \lfloor \frac{r}{i} \rfloor$ and in our example $r = 8$.

We are now ready to illustrate some approximation algorithms that do have constant performance ratios.

Unweighted Vertex Cover

Recall that a *matching* in an undirected graph $G = (V, E)$ is a set of independent edges. One of the most famous algorithms is Edmonds' algorithm for finding a **maximum** sized matching in a graph in polynomial time. A **maximal** matching can be found quite easily by greedy means in linear time (a matching M is maximal if $M+e$ is not a matching for some edge $e \in E-M$.)

Exercise: Write a linear time algorithm to find a maximal matching in a graph.

Let C be the set of vertices obtained by taking both endvertices from each edge in a maximal matching M. We first claim that C is vertex cover. Suppose otherwise, that C is not a vertex cover. Then there is some edge $e = (u, v) \in E$ such that neither u nor v are in C. But then C could not be a maximal matching as we could have added e to C and still had a set of independent edges.

We next show that any vertex cover (including a minimum sized one) must have at least $|M|$ vertices. Consider edges $(u, v), (y, z) \in M$, which of course share no endpoints. Any vertex cover X must cover these edges. But X cannot cover these two edges with a single vertex, since the edges share no common endpoint. Hence each edge in M must be covered by a different vertex in X, hence $|X| \geq |M|$.

Thus we have that $|C| \leq 2 \cdot |OPT|$, where OPT is the size of the smallest vertex cover, i.e. the performance ratio of this algorithm is two. Although the best known lower bound on the performance ratio for vertex cover is about 1.04 [4], there are no algorithms known that have a better performance ratio than two. Finding one (or proving it is not possible) remains one of the holy grails of this field.

Weighted Vertex Cover

Let us generalize the vertex cover problem as follows. Each vertex is assigned a positive weight. We are seeking the vertex cover of minimum total weight (the sum of the vertex weights in the cover is as small as possible). This is obviously at least as hard as the usual vertex cover problem, but is still in NP. However, our previous approximation algorithm does not provide us with a good performance guarantee in this case.

Consider the following integer linear programming formulation of the minimum weight vertex cover problem in a graph $G = (V, E)$, where $w(v)$ denotes the weight of vertex v:

$$minimize \sum_{v \in V} w(v)x(v) \tag{7.1}$$

subject to

$$x(u) + x(v) \geq 1 \quad \forall (u, v) \in E \tag{7.2}$$

$$x(v) \in \{0, 1\} \quad \forall v \in V \tag{7.3}$$

$x(v)$ is the characteristic function of the vertex cover, C. If $v \in C$ then $x(v) = 1$, otherwise $x(v) = 0$. So constraint (2) ensures that each edge is covered.

Unfortunately, integer linear programming (finding the optimal solution to (1)) is NP-complete. But, perhaps surprisingly, Linear Programming is solvable in polynomial time (albeit by complicated methods) – since in such cases we do not have constraints like (3) which require the solutions to be integral. So we can "relax" the integer linear program above by replacing (3) with the following

$$x(v) \geq 0 \quad \forall v \in V \tag{7.4}$$

Now solve this linear program optimally in polynomial time and let x^* denote the optimal solution (consisting of values assigned to each vertex). We round x^* to an integer solution x_i (since it makes no sense to have half a vertex in a vertex cover!) as follows: for each $v \in V$ set $x_i(v) = 1$ if $x^*(v) \geq 0.5$ and $x_i(v) = 0$ otherwise.

Observe that x_i is indeed a vertex cover for G. Constraint (2) for each edge (u, v) ensures that at least one of $x^*(v)$ and $x^*(v)$ is at least 0.5, hence at least one of them will be rounded up and included in the cover. Furthermore, the rounding has the property that $x_i(v) \leq 2x^*(v)$ for each $v \in V$ so we obtain the fact that the weight of the vertex cover corresponding to x_i is at most twice the weight of the "cover" corresponding to x^* (which wasn't really a cover, since it had fractional vertices!). But it is clearly the case that any vertex cover must have weight at least that of x^*, so we have that the weight of the approximate cover is at most twice that of the optimal.

Note how in both approximation algorithms for Vertex Cover, we found some way of giving a lower bound of the value of the optimal solution, whether os was the cardinality of a maximal matching or the value of an optimal (though unreal) "fractional" cover. Then once we can give an upper bound of the value of our approximate solution, we can derive a performance ratio.

This technique of using linear programming as a basis for approximation algorithms is often a fruitful one. It was generalized by Goemans and Williamson [18], who devised a method called semi-definite programming that yields, for example, an algorithm for MAX-CUT with performance ratio 1.14 and one with performance ratio 1.07 for MAX-2SAT.

7.5.3 Traveling Salesman's Problem

We show how to approximate TSP (find the shortest Hamiltonian cycle in a complete weight graph) in the case the distances satisfy the triangle inequality: for all vertices u, v, w, distance$(u, w) \leq$ distance(u, v) + distance(v, w). Which is another way of saying the shortest distance between two points is a straight line!

Euler Tours

Recall than an *Euler tour* of a graph is a closed walk that traverses each edge of the graph exactly once (some vertices may be visited more than once, and of course we finish the walk where we start). It is well-known that G has an Euler Tour if and only if it is connected and each vertex has even degree.

If every vertex has even degree, we can find an Euler tour in polynomial time as follows. Start at a vertex v. Go to a neighbor u of v (and delete edge uv) provided deleting edge uv does not disconnect G into non-trivial components. That is, if we delete an edge a leave a vertex isolated, that is ok, since we have used up all that vertice's edges. But if we disconnect the graph so that more than two components are non-trivial (a trivial component is an isolated vertex) we choose another neighbor of v with which to proceed.

It is easy to see that we can implement this algorithm in polynomial time. An example can be seen in Figure 7.18.

Exercise: Analyze this algorithm.

To prove that it works, let G be a graph with every vertex having even degree. Note that G cannot have any bridges (edges whose deletion disconnect the graph) since G has an Euler tour. That is, if there were a bride, once crossed, there would be no way to get back where we started.

Suppose the algorithm fails to find an Euler tour in G. That is, at some point the algorithm halts because the only edges incident to v are bridges between nontrivial components in the current graph $G' = G$ minus the set of edges that have been deleted thus far. Consider the edges deleted so far. Let C be those edges deleted so far that form closed walks and cycles. Let $G'' = G - C$. Clearly G'' has every vertex with even degree, but is has a bridge incident to v and thus cannot have an Euler tour. This is a contradiction, proving our algorithm is correct.

A First Approximation Algorithm

Let $G = (V, E)$ be the weighted graph that is our instance of TSP; G satisfies the triangle inequality. The algorithm is as follows:

1) Find a Minimum Spanning Tree, T, of G.
2) Replace each edge in T with a pair of edges, thus forming a graph T.
3) Observe that T' must have an Euler tour.
4) Find a Euler tour of T'.
5) Short-cut the Euler tour, thereby forming a TSP tour.

Let us explain step 5. Suppose the Euler tour is $a, b, c, d, a, f, \ldots$. That is, we have repeated vertices. A TSP tour must be a simple cycle (no repeated vertices except for the start/finish). We short cut this Euler tour by modifying it to be $a, b, s, d, f, \ldots$. That is, we jump from d over repeated vertex a to the next vertex we have not visited before. By the triangle inequality, we will have the the length of the TSP tour thus obtained is no greater than that of the Euler tour. Also note that the length of the Euler tour will be at most twice that of the weight of a minimum spanning tree for G. Finally, observe that any TSP tour must have a weight at least as large as that of a minimum spanning tree, since a Hamiltonian Path is a subgraph of the TSP tour a a Hamiltonian Path is a spanning tree of G. Thus the performance ratio of this algorithm is two.

A Better Approximation Algorithm

We can improve upon the previous algorithm as follows. Steps 1, 3, 4, and 5 are the same. The only difference is Step 2, the way in which we obtain an Eulerian graph T'. Consider T, a minimum spanning tree of G. Now just focus on the vertices in T having odd degree. There must be an even number of them (since any graph has an even number of odd degree vertices).

Let H be the subgraph of G induced by these odd degree vertices from T. Now find a minimum weight matching, M, of H (which will include all the vertices of H, since H is a complete graph on an even number of vertices). Add the edges in this matching to T in order to form an Eulerian graph. We then complete the algorithm as before.

An example is shown in Figure 7.18.

We now claim that the TSP tour thus obtained is of weight at most 1.5 times that of the optimal TSP tour. To prove this we must only show that the sum of the weights of the edges in M is at most 0.5 times that of an

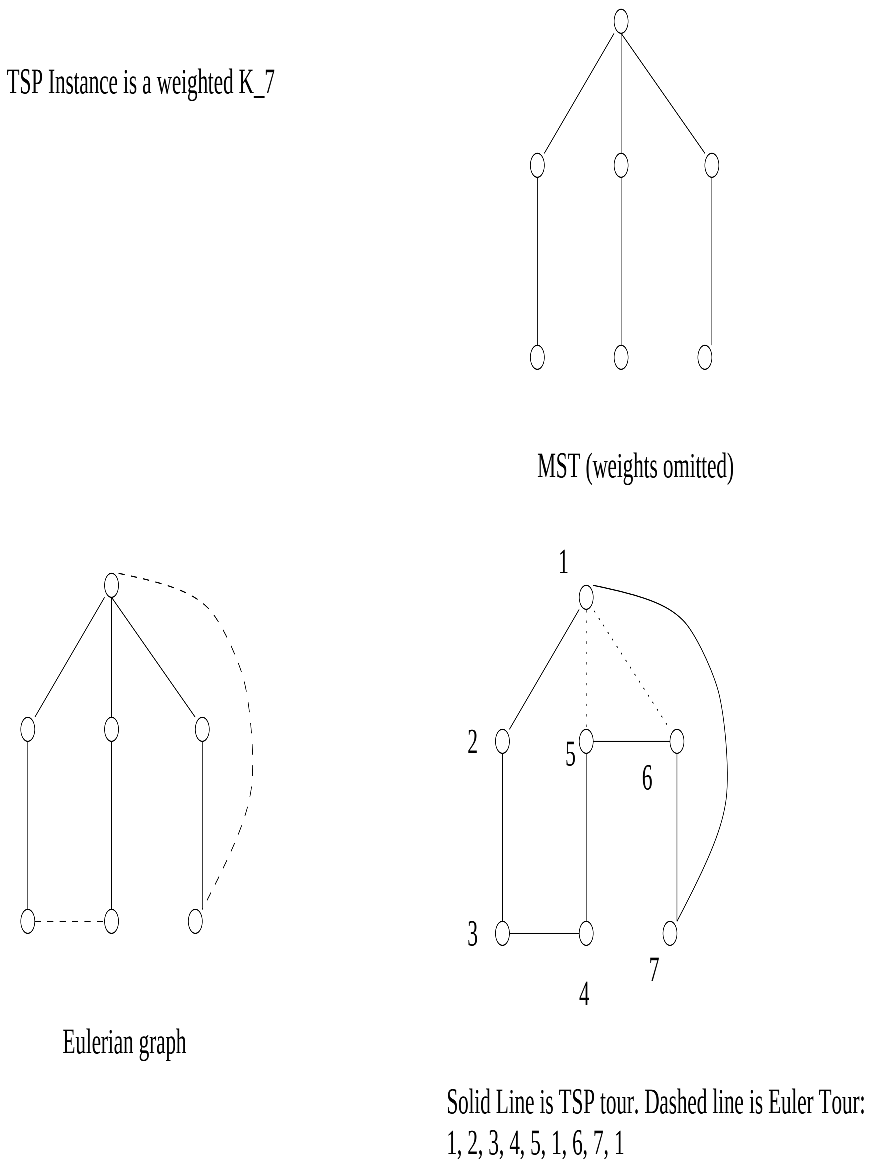

TSP Instance is a weighted K_7
MST (weights omitted)
Eulerian graph
1
2
5
6
3
4
7
Solid Line is TSP tour. Dashed line is Euler Tour:
1, 2, 3, 4, 5, 1, 6, 7, 1

Figure 7.18: Approximating TSP

optimal TSP tour of G.

Let $W = v_1, v_2, \ldots, v_n$ be an optimal TSP tour of G and let $O \subseteq W$ be the odd degree vertices from T, say $O = \{v_i, v_j, v_k, \ldots, v_z\}$. A minimum weight matching, M, of O contains at most $\frac{n}{2}$ edges. To see that the weight of M as at most $\frac{weight(W)}{2}$, note that W has n edges. Now traverse W in a clockwise fashion, going from one vertex in O to the next vertex in O on W by skipping over vertices not in O via short-cuts (remember, the triangle inequality is at work here!). Pair off the vertices in O as you go, forming a matching – the first two vertices in O encountered on W get matched, then the next two and so on. Now traverse W in a counter-clockwise direction, again matching the first two vertices in O, the second pair, and so on, again forming a matching of the vertices in O. Now observe that these two matchings are disjoint (share no edges in common), since there are an even number of vertices number of vertices in O. So for example, in the first matching, v_i is matched with v_j, whereas in the second matching, v_j will be matched with v_k and v_i with v_z. By the triangle inequality, the sum of the edge weights in the union of these two matchings is at most the weight of W, hence one of the matching has weight at most $\frac{weight(W)}{2}$. This proves the performance ratio is at most 1.5.

This algorithm, due to Christofodes (1976) is the best known approximation algorithm for metric TSP. As with vertex cover, it would be a major accomplishment to improve upon this algorithm. In 1999 Arora gave an approximation scheme for the Euclidean TSP (a special case of metric TSP where distances correspond to actual distances in 2-dimensions) (*S. Arora, Journal of the ACM, vol. 45, no. 5, pp. 753-782, 1999*). Like vertex cover, the analysis was simple because we could easily find a lower bound on the value of the optimal solution and an upper bound on our approximate solution.

7.5.4 Multi-processor Scheduling

The multi-processing problem consists of n jobs, $J_1, \ldots, J_n$, each having a runtime $p_1, \ldots, p_n$ which are to be scheduled on m identical machines. The goal is to minimize the time at which the last job finishes. We saw before that this is NP-complete even when $m = 2$ (reduction from Partition).

The following algorithm, known as the List Scheduling algorithm, was devised by Graham in 1966 and is one of the earliest approximation algorithms. The algorithm considers the jobs one by one and assigns the current job to the least loaded processor (the *load* on a processor is the sum of the job lengths assigned to it so far).

We show that List Scheduling has a performance ratio of $2 - \frac{1}{m}$.

Assume that after all the jobs have been scheduled that machine M_1 has the highest load (= latest finishing time). Let L denote the total load on M_1 and let J_j be the last job assigned to M_1. Note that every machine (after all jobs have been scheduled) has load at least $L - p_j$, else J_j would not have been assigned to M_1 – M_1 had load exactly $L - p_j$ when the J_j was assigned to it. Hence, summing up all the job lengths, we see that

$$\sum_{i=1}^{n} p_i \geq m(L - p_j) + p_j$$

And we also know that

$$OPT(I) \geq \frac{\sum_{i=1}^{n} p_i}{m}$$

Since some machine must have a load at least equal to the average of the machine loads. Combining these two equations yields

$$OPT(I) \geq \frac{m(L - p_j) + p_j}{m}$$

and simplifying, we have that

$$OPT(I) \geq (L - p_j) + \frac{p_j}{m}$$

$$OPT(I) \geq L - (1 - \frac{1}{m})p_j$$

.

Now dividing both sides by $OPT(I)$ and re-arranging shows

$$1 + \frac{(1 - \frac{1}{m})p_j}{OPT(I)} \geq \frac{L}{OPT(I)}$$

and since $OPT(I) \geq p_j$ (some machine must run that job!) we get

$$1 + 1 - \frac{1}{m} \geq \frac{L}{OPT(I)}$$

which means

$$2 - \frac{1}{m} \geq \frac{L}{OPT(I)}$$

Since L is the length of the approximate schedule, we have proved our performance bound.

To see that this analysis is optimal, we show that there exists an instance such that List Scheduling does this badly.

Let $n = m(m-1)+1$ and let the first n jobs have run time 1 and the last job has run time m. The optimal schedule is to run m jobs each (of length 1) on the first $m-1$ machines and the big job all to itself on one machine for a run time of m. But list scheduling assigns $m-1$ length 1 jobs to each machine and the long job to some machine, hence it requires $2m-1$ units time to finish.

A better performance ratio can be obtained by scheduling the c largest jobs optimally (for some constant c – using brute force) and then using list scheduling on what remains (but the jobs are sorted in decreasing order of run time first). This strategy can be shown to have a performance ratio of $1+\epsilon$ for any ϵ you like – though the running time is exponential in $\frac{1}{e}$. This is known as an approximation scheme (but is not a fully polynomial one – these would run in time polynomial in $\frac{1}{e}$. Such schemes exist for Knapsack, for example). But even for $c = 10$, you can see a reasonable benefit.

7.5.5 Bin-Packing

Bin Packing is a similar problem to scheduling. We have n items to pack into as few unit sized bins as possible. Each item has a size, $0 < s_i \leq 1$. The First Fit algorithm is just like List Scheduling: we consider the items one by one, packing each into the lowest indexed bin into which it will fit.

We show that First Fit has a performance ratio of 2. The optimal clearly must use at least $\lceil \sum s_i \rceil$ bins. We will show that First Fit uses no more than $\lceil 2 \sum s_i \rceil$ bins.

First note that when First Fit is done, there is at most one bin that is half-empty (or more than half-empty). If this were not the case, First First would have combined those two bins into one.

If all the bins are at least half full, then obviously First Fit has used at most $\lceil 2 \sum s_i \rceil$ bins. So suppose one bin, B_q, is less than half full, i.e. $B_q = 0.5 - \epsilon, \epsilon > 0$.

Let us assume without loss of generality that q is the highest numbered bin. Then for all $B_i \neq B_q$, $B_i > 0.5 + \epsilon$, else $B_i + B_q \leq 1$ and First Fit would have used only one bin. So if $q \leq 2$, First First must be optimal and our performance guarantee is achieved.

So assume $q > 2$. Then

$$(q-1)(0.5+\epsilon)+0.5-\epsilon = 0.5q-0.5-\epsilon+q\epsilon+0.5-\epsilon$$

$$= 0.5q-2\epsilon+q\epsilon$$

which from our comments above about the B_i's is less than or equal to $\lceil \sum s_i \rceil$ which we know is a lower bound on $OPT(I)$. That is

$$= 0.5q-2\epsilon+q\epsilon \leq OPT(I)$$

We claim that $FirstFit(I) = q \leq 2OPT(I)$. First notice that $2\lceil \sum s_i \rceil \geq \lceil 2\sum s_i \rceil$.

So we want to show that

$$q < \lceil 2(0.5q-2\epsilon+q\epsilon) \rceil$$

where q is the number of bins used by First Fit. Where the right hand side is twice a lower bound on the OPT(I). Simplifying, we want to show that

$$q < \lceil q-4\epsilon+2q\epsilon \rceil$$

Which is true if we can show

$$q < q-4\epsilon+2q\epsilon$$

And re-arranging we get

$$0 < 2q\epsilon-4\epsilon$$

$$0 < q\epsilon-2\epsilon$$

which is true because we assumed $q > 2$. Hence the proof.

We note that if we initially sort the items in decreasing order of size, First Fit has a performance ratio of 1.5 (and the proof of this is quite involved).

7.5.6 Max-SAT and Max-Cut: Randomized Algorithms

7.5.7 Approximation Schemes

Knapsack

A Scheduling Problem

7.5.8 Impossibility of Approximation

We now see that certain NP-complete problems cannot be effectively approximated, unless $P = NP$. Thus we may sub-classify the NP-complete problems as to how hard they are to approximate (absolute guarantees, approximation schemes, relative guarantees, no guarantees). So although we initially stated that all the NP-complete problems "were the same," (which they are, in terms of being polynomially equivalent), they are not when it comes to approximability.

TSP

Previously, we saw that TSP could be effectively approximated if the distances satisfied the triangle inequality. Now we show that the general TSP problem is hard to approximate. Suppose we had an approximation algorithm for TSP with performance ratio c (c is some constant). Let G be an instance of the NP-complete Hamiltonian cycle problem with n vertices. Construct an instance of TSP, G' (a complete, weighted graph on n vertices) by setting weight(u, v)=1 if (u, v) is an edge in G and weight(u, v)=cn otherwise. Now we input this graph to our approximation algorithm for TSP. If G has a Hamiltonian cycle, then G' will have a TSP tour of weight n. Otherwise, the shortest TSP tour in G' must be of weight greater than cn (since such a tour must contain at least three vertices and one edge of weight cn). Thus, because of the performance ratio of our algorithm, if G has a Hamiltonian cycle, we will find a TSP tour of weight exactly n (since we are guaranteed to find a tour of weight at most c times the optimal). And if G does not have a Hamiltonian cycle, we will find a TSP tour of weight greater than cn. Thus we could decide the Hamiltonian Cycle problem in polynomial time. So we see that TSP is in fact NP-hard to approximate.

PCP Theorem and Clique

For many years there were very few impossibility results for NP-complete problems, except for TSP and a few others, even though many such problems seemed hard to approximated (the best known approximation algorithms had performance ratios of $O(\log n)$ or worse). Then in 1992 came the following theorem, known as the *PCP Theorem*, which is probably the most spectacular result in the field of computer science.

Theorem 17 (Arora, Lund, Motwani, Sudan, Szegedy et al.) $NP = PCP(\log n, 1)$

The main reference for this is "Probabilistic Checking of Proofs and Hardness of Approximation Algorithms," by Sanjeev Arora, Ph.D. dissertation, U.C. Berkeley Computer Science Division, 1994. A useful survey with applications on how to prove non-approximability results can be found in [4].

Let us explain the statement of the theorem. A language (set) L is in $PCP(r(n), q(n))$ if there is a verifier M that probabilistically checks membership proofs for L on inputs of length n using at most $O(r(n))$ random bits and $O(q(n))$ queries to the "proof."

A "proof" Q is a finite bit string that purports to demonstrate that input x is in L (for example that at certain graph is in the set of 3-colorable graphs). The verifier then tests whether Q is in fact a legitimate proof of this by reading $O(q(n))$ bits of Q. If there is an polynomial time verifier M such that for every input $x \in L$ there is a proof Q that causes M to accept x (regardless of the string of random bits M uses) **and** if for any $x \notin L$ every alleged proof Q is rejected with probability at least $\frac{3}{4}$ then we say that $L \in PCP(r(n), q(n))$.

Thus the PCP theorem says that (specially constructed) proofs of membership for all languages in NP exist (which of course need to be only polynomial in size) and can be (probabilistically) tested in polynomial time by examining only a constant number of bits from the proof.

What does this have to do with approximation algorithms?

Theorem 18 *There is no approximation algorithm for Clique with a constant performance ratio unless $P = NP$.*

Proof (sketch): Suppose we have a $PCP(\log n, 1)$ proof system for a 3-SAT formula F. We build a graph G as follows. Let r be the actual number of random bits and b the number of queries. Let $< x, y >$ be a vertex where the verifier would declare the proof valid should the random bits generated by x and the answers to queries be the bit string y. An edge $(< x, y >, < a, b >)$ exists if $< x, y >$ and $< a, b >)$ are consistent. That is if for each common address in x (the random bit strings x and y are interpreted as being partitioned into addresses of length $\log p(n)$ for the proof of length $p(n)$) and y, a and b contain the same answer to the query. Since at most $\log n$ random bits are generated, G has $p(n)$ vertices. It is easy to see that G has a clique of size 2^r if F is in fact satisfiable, since the verifier must declare the proof to be true no matter what its random string is. Thus for each of the 2^r choices of random string x there is a vertex corresponding to this bit string and all 2^r such vertices must be consistent in order to meet the definition of a PCP. On the other hand, suppose F is not satisfiable and we have a (unbeknownst to us) bogus proof of this. Then we claim G has a clique of size at most $\frac{2^r}{4}$. Suppose there exists a clique V' in G of size greater than or equal to $\frac{2^r}{4}$. Construct a "proof" for F as follows. For each vertex $v = < x, y > \in V'$, assign to the bits in the proof at $c * |V'|$ addresses given by x the bits in y (each x corresponds to $c = O(1)$ distinct addresses). Since all vertices in V' are consistent, this will assign each of these addresses exactly one value. To each of the other $2^r - c * \frac{2^r}{4}$ addresses, assign arbitrary values. But now we have a proof for F which the verifier will accept with probability at least $\frac{2^r}{4}/2^r \geq \frac{1}{4}$ which is a contrary to the definition of a PCP.

If there were a polynomial-time approximation algorithm A for Clique with a performance ratio of two, we could run A on G and determine with certainty whether of not G has a clique of size 2^r or not: since if G does, A would return a clique of size at least $\frac{2^r}{2}$, which is only the case if G has a clique of size at least 2^r, by construction.

Thus it is NP-hard to approximate Clique to within a factor 2. This result has been strengthened to there being no approximation algorithm for Clique with performance ratio less than $n^{\frac{1}{3}}$ unless $P = NP$. $\square$

Using related techniques called "gap-preserving reductions" one can show a host of other problems (Dominating Set, Independent Set, Graph Coloring) are NP-hard to approximate to within a constant factor. The PCP theorem also implies the MAX-SNP-hard problems (such as MAX-SAT, Vertex Cover) do not have approximation schemes unless $P = NP$, i.e. there is a lower bound (greater than one) on the performance ratio of any approximation algorithm for each of these problems.

7.6 Exercises

7.7 Programming Assignments

Chapter 8

On-line Algorithms

8.1 Simple Amortization Arguments

8.2 On-line Algorithms

The multi-processor scheduling approximation algorithm we saw, the List Scheduling algorithm, had the property that a process was scheduled without knowledge of what the remaining, unscheduled processes look like. An algorithm that behaves this way, i.e. processes each piece of the input as it arrives, without waiting to see future inputs, is called an *online algorithm.*

Consider Louie De Palma, dispatcher for the Sunshine Cab company. Customers call the dispatcher and want a taxi sent to their location. Louie must decide which of his available taxis to send, with the overall goal of minimizing the distance traveled by his fleet. When a call arrives, Louie does not know what future calls may be and he must dispatch one taxi to pick up the customer immediately.

Let us formalize this as the k-server problem [23]. Let G be a weighted graph on n vertices (can be directed or undirected, let us assume undirected). We have $k < n$ mobile servers available. Requests arrive one by one (a sequence σ of requests must be processed). A request is simply a vertex v. If no server is at v one must be sent to v immediately, without any knowledge of subsequent requests. The cost incurred is the total distance travelled by the servers over the entire sequence of requests.

We measure the performance of an on-line algorithm for the k-server problem (and many other on-line problems) by means of its *competitive ratio*, a concept introduced by Sleator and Tarjan [35]. On-line algorithm A is said to be c-competitive for problem Π if the cost incurred by A is at most

c times the cost incurred by any other algorithm B for Π for any request sequence σ, even if B knows σ in advance. Hence we are comparing the cost incurred by A with that of an optimal offline algorithm B and doing so over all possible request sequences. Analysis of on-line algorithms is typically done in an amortized fashion (costs are averaged over the entire request sequence so that high costs for single requests may be offset by low costs for other requests) and worst-case sequences are constructed using adversarial arguments.

It is not difficult to see that the k-server problem is a generalization of the paging problem (consider the k-server problem on an complete, unweighted graph: in this case the servers are "marking" the pages stored in fast memory, whereas the vertex set if the universe of all possible pages).

It is not hard to prove that k is a lower bound on the competitive ratio for the k-server problem, even in the case of complete, unweighted graphs (though it is a bit more difficult to prove that this lower bound applies to any metric space with at least $k+1$ points). Many algorithms are know to be 2-competitive in general (see for example [23]). In [21], the simple work-function algorithm is shown to be $2k-1$ competitive in general, and it is conjectured that the work-function algorithm is in fact k-competitive, which would be optimal, given the lower bound of k on the competitive ratio for the k-server problem. Sleator and Tarjan [35] discuss the competitiveness of some familiar algorithms for paging such as LRU. Load balancing in parallel and distributed computing systems and other scheduling problems are obvious application areas for on-line algorithms.

Another well-known on-line problem is the dictionary problem for data structures in which a data structure maintains a set of keys and searches, inserts, and deletes are performed. The goal is to minimize the time spent traversing the data structure. The move-to-front-list is shown below to be 2-competitive for list (one-dimensional) data structures [35] and the splay tree is known to be $O(\log n)$ competitive for binary trees (of course, a static height-balanced binary tree is also $O(\log n)$ competitive, though not as effective for certain request sequences as a splay tree [6]).

In an amortized analysis, the time required to perform a sequence of operations is averaged over all the operations. A total time to perform n operations of $T(n)$ is shown–yielding an average of $\frac{T(n)}{n}$ time per operation, although some individual operations may require more or less than the average time. Amortized analysis differs from average-case analysis in that an amortized bound is a guaranteed worst-case bound. In average-case analysis, some probability distribution of the inputs is typically assumed (for example, we might assume that the searches are drawn uniformly and randomly with equal probability) and use probabilistic methods to determine how much time we expect to use, on average.

If we flip a fair coin 100 times, we expect to get 50 heads and 50 tails, but more often than not we will get some other combination of heads/tails. We may think of an amortized bound as being a coin that has a memory–so that if we flip the coin a sufficient number of times, it will somehow "adjust" its outcomes so that the number of heads equals the number of tails.

An outstanding and thorough text on on-line algorithm is [7].

8.2.1 Move-to-front List

We now study the dictionary problem: use some data structure to maintain a set of keys under an intermixed sequence of inserts, accesses, and deletes. In real-world applications we usually do not know two crucial pieces of information:

1) the sequence of operations (what order the keys are searched)

2) the overall probability of access to a particular key

If we do know 2) and these probabilities are fixed, we can construct on optimal search tree by using dynamic programming techniques. However, in the real-world, operations also often exhibit locality of reference. That is, some keys are accessed frequently for some finite period of time, followed by period of lighter access. In other words, the access probabilities are dynamic. The set of keys that are currently being accessed (frequently) is called the working set. A static data structure such as a balanced tree does not notice or reflect changing access patterns.

To speed up access of frequently accessed keys, these keys need to be near the front of the list, or near the root of the tree. However, we don't really know what keys are in the working set, as it is constantly changing!! Data structures that modify their structure to reflect the access pattern are called *self-adjusting data structures.*

We consider the simplest of these structures: the *move-to-front list.* The move-to-front (MTF) list is a linked list. The keys are not ordered (sorted) by key value. After accessing or inserting a key, it is moved to the front of the list.

Other variants on this theme move an accessed record one place up in the list (but this can result in two records continuously swapping places and never moving forward) and moving halfway toward the front of the list. The overall goal of the move-to-front rule is to have the working set congregate near the front of the list.

One important use of amortized analysis is called *competitive analysis*. We shall examine how optimal this MTF strategy is. Competitive analysis is used to analyze on-line algorithms: that is, algorithms that must make decisions without knowledge of future requests.

Recall that we said an algorithm is c-competitive if $C_A(\sigma) \leq c{*}C_B(\sigma)+a$ for some constant a for any and every algorithm B; where $C_A(\sigma)$ is the cost of our algorithm,A, on a sequence of requests σ and $C_B(\sigma)$ is the cost incurred by B on the same sequence of requests σ. By cost we may mean comparisons made, in the case of list access strategies.

To avoid the arduous(!) task of comparing our algorithm with every other one, we simply compare it with the optimal one! Of course, since we don't know the optimal one, we loosen the rules and compare our algorithm with the optimal algorithm that knows the future (i.e. knows σ in advance) since the optimal on-line algorithm is certainly no better than the optimal off-line algorithm. The optimal offline algorithm is said to be controlled by an *adversary*, who is allowed to **construct** σ. The adversary constructs σ in order to maximize the ratio of C_A to C_B. We may think of the adversary as knowing how algorithm A will behave in the future and we will refer to the optimal, offline algorithm B as the adversary's algorithm.

We now prove that MTF is 2-competitive, that is, MTF makes no more than twice as many comparisons as any omnipotent algorithm. This result is due to Sleator and Tarjan [35].

We use the concept of a *potential function*. Recall from physics that an object hanging above the ground has potential energy. A potential function captures the goodness of the configuration of our algorithm (in this case, a low potential value is good from our algorithms perspective and conversely, a large potential value is good from the adversary's point of view). For instance, we may be willing to pay a lot for one move now if the result is an improved potential (which will result in future moves costing less). By configuration, we mean the order of the keys in our list.

In general, a potential function is a non-negative function used to capture the goodness of an online algorithm's configuration relative to the adversary's.

Specifically, a dictionary operation takes time t_i for the i^{th} operation and changes the potential from Φ to Φ', then the sum $t_i+\Phi'-\Phi$ is the *amortized cost* of the i^{th} operation. Thus a large t_i (high actual cost for operation) is balanced by a large decrease in potential (good for us!!), meaning we will spend less in the future. Thus we are reducing the cost we just paid,

t_i, to a_i, because we know that in the future we will spend less for some operation(s). Likewise, we will pay an inexpensive actual cost with some extra cost in order to pay for future expensive operations. This is what we mean by amortize.

Then for all operations in σ (which is possibly an infinite sequence) we have

$$\sum t_i = \Phi_0 - \Phi_f + \sum a_i$$

where Φ_0 is the initial value of the potential function (often taken to be zero in many applications), Φ_f is the final value of the potential function and a_i is the amortized cost of the i^{th} operation. Note that the Φ terms have dropped out of the summation because they form a telescoping summation, hence only the first and last Φ terms have cancelled out!

Re-arranging terms, we have that

$$a_i = t_i + \Phi' - \Phi$$

which we may write as

$$a_i = t_i + \Delta(\Phi)$$

where $\Delta(\Phi)$ means the change in potential during the i^{th} operation.

To show the MTF list is 2-competitive we define the potential to be the number of *inversions* in the MTF list as compared to the adversary's list. An inversion is any pair of keys i, j such that i occurs before j in one list and after j in the other list. Thus if the MTF list were the same order as the optimal list, there will be no inversions. So the number of inversions measures how far our list is from the optimal list. We note the in general, potential functions must be non-negative functions, and ours is in this case as well.

Now consider an access of key i on the i^{th} operation. Let k be i's position in MTF's list and let x_i be the number of items that precede i in MTF's list but follow i in the adversary's list.

```
Adv       1  2  3 ...... i .....
MTF       1  2  3 ............ i ....
          |____________________|
                     k
```

Then the number of (common) keys preceding i in BOTH lists (each list) is $k-1-x_i$.

Moving i to the front of the MTF list creates $k-1-x_i$ additional inversions and destroys x_i inversions (since i now precedes these x_i values). Therefore the amortized time for the i^{th} operation, a_i is

$$a_i = t_i + \Phi' - \Phi =$$

$$k + \Delta(\Phi)$$

Since x_i inversions are created and $(k-1-x_i$ inversions destroyed, we have

$$k + (k-1-x_i) - x_i = 2(k-x_i) - 1.$$

But $(k-x_i) \leq i$, since of the $k-1$ items preceding i in the MTF list, only $i-1$ of them precede i in the adversary's list. Hence the amortized time to access i is at most $2i-1$, proving our claim.

```
Adv       1  2  3 ..... i .....
          |_____________|
                 i
MTF       1  2  3 ...........i ....
                        |___|
                         x_i
          |_________________|
                 k-1
```

8.2.2 Paging and the k-Server Problem

Paging

Recall the paging problem from operating systems: n pages of **fast** memory are available to a computer system, e.g. cache memory. Memory accesses to these memory locations are considerably faster than accesses to RAM or disk, hence we would like to store those pages that we will be accessing often in the near future in this fast memory (a single page may store a few bytes or a few thousand bytes, depending on the system and particular type of fast access memory we are dealing with). The paging problem then is to decide which page to eject from the n fast pages when a page is requested that is not currently stored in the fast page set. As a rule, we must always store the most recently accessed page amongst the n fast pages. The objective is to minimize the number of page faults, i.e., the number of requests which require us to retrieve a page from slow memory and eject a page from fast memory. Having few page faults then equates with fast performance of our computer system.

Given that we do not know the future sequence of page requests, this is an on-line problem. Several well-known strategies exist for the paging problem:

(1) LRU: eject the page, from amongst the n pages in fast memory, that has not requested for the longest period of time (Least Recently Used).
(2) FIFO: eject the page that was brought into fast memory the longest time ago (First-in First-out).
(3) LFU: eject the page the page that has been used least frequently from among all the pages in fast memory.

The optimal strategy, MIN, requires knowledge of the future requests: eject the page whose next request is furthest into the future. We prove that LRU is n-competitive, i.e., it makes at most n times as many page faults as MIN, given that each has n pages of fast memory.

Note that an easy way to implement LRU is to maintain the list of pages in fast memory in a move-to-front list. Each time a page is accessed, move that page to the front of the list. When a page fault occurs, evict page that is at the end of the list. The page brought into memory is inserted at the front of the list.

We assume that both LRU and MIN begin with no pages in fast memory. Let S be the sequence of page requests. Divide S into subsequences $S_1, S2, \dots$ so that each S_i contains precisely n page faults by LRU. If S is finite and the last subsequence contains fewer than n faults, then these page faults can be lumped into the additive constant in the definition of c-competitive. In subsequence S_1, LRU faults n times and MIN faults n times, since both begin with no pages in their fast memories.

Consider subsequence S_i, where $i > 1$. Let p be the last page requested in the previous subsequence. So at the beginning of S_i, both LRU and MIN have page p in their fast memories.

Lemma 19 *There are requests to at least n different pages other than p during S_i.*

Proof: S_i ends when LRU faults n times during S_i. Consider a page q that is in LRU's fast memory at the beginning of S_i or that is brought into LRU's fast memory during S_i. It may be that q is evicted at some point during S_i. However, we claim that q cannot be evicted twice during S_i. If true, then it is easy to see that the Lemma must also be true (think about the extreme situation when none of the pages that are in LRU's fast memory at the beginning of S_i are accessed during S_i. That is, each request during S_i causes LRU to fault). To see the claim is true, suppose to the contrary that a page q is evicted twice during S_i. That means q was evicted, then brought into main memory later in S_i when q was again requested. But after this request, q is inserted at the front of the move-to-front list we use to implement LRU. Page q will be evicted a second time only when it is at the end of the move-to-front list. A page can only move one position towards the end of the list per request during S_i. And a page moves one position only if a page x that follows it is requested (in which case x will then precede q in the list) or a page from external memory is brought into fast memory. Hence n pages other than q must be accessed before q works its way to the end of the list. The Lemma then follows. □

Thus we know that during S_i, n pages other than p are accessed. Now consider MIN. MIN has n pages of fast memory, thus cannot keep both p and n pages other than p in its fast memory at one time during S_i. Therefore, MIN makes at least one page faulty during S_i.

Since LRU makes n page faults during each S_i and MIN makes at least one page fault during each S_i, the ratio of page faults made by LRU during S to the number of page faults made by MIN over S is at most n.

We now show that no on-line algorithm can have competitive ratio less than n. We define a sequence S of page requests in which on-line algorithm A makes n page faults and MIN makes only 1 fault, and this sequence can be repeated indefinitely. Assume that A and MIN initially have the same set of n pages.

(1) The first request is to a page in neither A nor MIN's fast memory, hence each fault once.

(2) Let X be the set of pages either in MIN's fast memory initially or brought in on the first request, so $|X| = n + 1$.

(3) Each of the next $n - 1$ requests is made to a page in X which is not currently in A's fast memory.

Hence during this sequence, A faults n times and MIN faults only once. At the end of this sequence, both A and MIN have the same set of n pages in fast memory and thus the process can be repeated.

Example. Suppose $n = 3$ and let A and MIN both hold pages 1, 2, 3 initially. On request for page 4, supposed A ejects 3, in which case MIN chooses to eject 1. The next request is for 3, and A ejects 2. The next request is for 2 and A ejects 1. At this point both A and MIN hold pages 2, 3, 4 and A has made 3 faults versus 1 for MIN.

Thus LRU is n-competitive and an "optimal" on-line paging algorithm. It turns out that FIFO is also n-competitive (which can be proved using a similar argument as above), though in practice, LRU performs much better that FIFO.

Note that this lower bound also gives a lower bound of k on the k-server problem (since the paging problem is simply the special case of the k-server problem when it is played on a complete graph: the vertices are pages and the servers locations' represent pages that are in fast memory).

k-Server Problem

Recall that we gave a dynamic programming algorithm previously to solve the offline 2-server problem. The k-server problem is as follows:

Given a graph $G = (V, E)$ and a (possibly infinite) sequence of requests: $\{r_1, \ldots, r_n\}$ [1] Suppose the servers are initially at $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$
Let V denote the number of vertices and n the number of requests.

Let us consider the 2-server problem on a path of length two, a graph having three vertices, v_1, v_2, v_3 and edges v_1v_2, v_2v_3.

We employ the Harmonic algorithm. It is a randomized algorithm, and we use the *oblivious* adversary model, which means the adversary knows our strategy, but not the outcome of our coin flips (and thus not the actual moves made by our algorithm, as opposed to an *adaptive adversary*, who can view the outcome of our coin flips. Obviously, we are more likely to have a good competitive ration when playing against the weaker opponent: the oblivious adversary. We shall describe the algorithm as we go.

[1]Actually, the k-server problem is defined for any metric space, not just graphs, but we limit our discussion to graphs.

Let Harmonic's two servers be at vertices x and y. Our potential function is $\Phi = 3M_{min} + dist(x, y)$, where M_{min} is the weight of the minimum bipartite matching between our server and the adversary's servers, a, b. In other words, M_{min} simply captures how closely aligned our servers are to the adversary's.

We prove that Harmonic is 3-competitive (there do exist 2-competitive algorithms for this problem). Again, the basic c-competitiveness argument has two phases:

(1) Show that when the adversary moves, that Φ does not increase by more than c times the cost incurred by the adversary.

(2) Show that when the on-line algorithm moves, that Φ decreases by at least the cost incurred by the online algorithm.

For our Harmonic algorithm, we consider two cases.

Case 1). Suppose $x = v_1, y = v_3$. Then the only request of interest is at v_2, else the request can be served at cost zero.

When the adversary serves this request, if it does not move a server, then it costs zero and there is no change in the value of Φ. If the adversary does move, then it moves at cost one or two and the value of M_{min} can increase by at most one. Hence $\Delta(\Phi) \leq 3 * cost_{ad}$, where $cost_{ad}$ is the cost incurred by the adversary. To see how the adversary can move at a cost of two, imagine that a, b are located at v_1, v_2 and a moves to v_3. But in this case, M_{min} cannot increase by two because of the location of x, y. This is the first phase of the competitive analysis, showing that, when the adversary moves, the change in potential is bounded by the multiplicative factor of the competitive ration.

Harmonic serves a request at v_2 by sending x with probability 1/2 and y with probability 1/2. The cost incurred by Harmonic is 1. The expected change to $dist(x, y)$ is 1, i.e., our two servers are guaranteed to get closer together. The expected change to M_{min} is as follows. It must be that the adversary's servers occupy either v_1, v_2 or v_2, v_3, since the adversary just served the request at v_2. (The argument is similar if the adversary would for some reason choose to have both its servers at the same vertex, though one can argue that this will never happen). In the former case, if x serves the request, then M_{min} increases by 1, from 1 to 2; and in the latter case M_{min} decreases by 1 from 1 to 0. Each of these events occurs with equal probability, hence the expected change to M_{min} is zero. Likewise if a, b are located at v_2, v_3. Thus the expected change in potential is a decrease by 1, which pays for the expected cost incurred, which is 1.

Case 2) Suppose $x = v_1, y = v_2$ (the argument is similar when $x = v_2, y = v_3$). Then the only case we need to consider is when the request is at v_3.

When the adversary serves this request, if it does not move a server, then it costs zero and there is no change in the value of Φ. If the adversary does move, then it moves at cost one or two and the value of M_{min} can increase by at most two. If the adversary moves one unit, then M_{min} can increase by at most one, and if the adversary incurs a cost of two, then M_{min} can increase by at most two. Hence $\Delta(\Phi) \leq 3 * cost_{ad}$, where $cost_{ad}$ is the cost incurred by the adversary.

Harmonic serves the request with x with probability 1/3 and with y with probability 2/3. Hence the expected cost is $1 * (2/3) + 2 * (1/3) = 4/3$.

EXERCISE: Show that if the on-line algorithm always serves the request with the closest server, then the algorithm is not c-competitive, for any constant c.

The expected change in $dist(x, y)$ is $0 * (1/3) + 1 * (2/3) = 2/3$, the former term being if we serve the request with x and the latter if we serve the request with y. Note that this expected change is an **increase** in value.

The expected change in M_{min} is as follows. There are two cases, since we know that one of the adversary's servers must be at v_3.

(i) If $a = v_1, b = v_3$. Then prior to Harmonic's move, $M_{min} = 1$, since x and a are aligned and y and b are not. If x serves the request, then there is no change in M_{min} (since now we have that x and b are aligned and y and a are not). If y serves the request, then M_{min} becomes zero, a reduction of one. Thus the expected change in potential is $3*((1/3*0)+(2/3*-1)) = -2$.

(ii) If $a = v_2, b = v_3$. Then prior to Harmonic's move, $M_{min} = 2$, since y and a are aligned and x and b are distance two from one another. If x serves the request, then both x and y are aligned with a and b so the reduction in M_{min} is 2. If y serves the request, then the reduction in M_{min} is 1, since y and b will be aligned and x and a will be distance one from one another. Hence the expected change in potential is $3 * ((1/3 * -2) + (2/3 * -1)) = -12/3 = -4$

.

In either case (i) or (ii), the expected decrease in potential is at least -2, which is enough to cover the expected cost, 4/3, plus the expected increase in potential due to the change in $dist(x, y)$ of 2/3. Thus Harmonic is 3-competitive.

We describe a 2-competitive algorithm called Double-Coverage for 2 servers on the real number line. The algorithm can be adapted to be used on a graph that is a path by treating vertices as integers on the real number line and having servers store in memory where their "real" location is. Servers physically move to a vertex when their stored real number location is an integer

Let the requested point be p. Let s_1, s_2 be the two servers and let s_1 be the closest server to p. Serve the request in a greedy fashion with s_1. However, if p lies in the interval between s_1 and s_2, we also move s_2 $d = p - s_1$ units towards p.

Use the following potential function, Φ, to prove the 2-competitiveness. Let x_1, x_2 denote the locations of the adversary's two servers. Let $\Phi = \alpha + \beta$ where

$$\alpha = 2 * \sum_{i=1}^{2} |x_i - s_i|$$

and

$$\beta = s_2 - s_1$$

where we re-label the servers as necessary so that $s_1 \leq s_2$ and $x_1 \leq x_2$.

Assume that initially $s_1 = x_1$ and $s_2 = x_2$. Note that when the adversary moves, only α can change, and if the adversary incurs a cost of d, then α changes by at most $2d$.

We consider one case of when Double-Coverage moves, to illustrate the technique, the other case is left as a challenging exercise.

Suppose s_1 and s_2 both lie on the same side of the requested point p (the alternative is that the requested point lies in the interval between s_1 and s_2). In this case, Double-Coverage moves only one server, say s_1, and does so at a cost of d. Recall that Double-Coverage will move the closer serve to p, so its servers will be farther apart after serving this request. Use the following diagram as a guide.

```
------- p ----- s1 ------ s2 ------- z  --------
          d         d2          d3
```

Let us first suppose that in so doing, s_1 moves d units closer to x_1 (where x_1 is the adversary's server that is "matched" with s_1 in the α matching). Of course, x_1 now resides at p. Then α decreases by $2d$ and β increases by d, for a net decrease of d, which suffices to pay for the cost incurred.

Now suppose that s_1 moves d units to point p and towards x_2, the server matched previously with s_2. β increases by d as above. Consider where x_1, the server matched with s_1 in α may be located (using the diagram above for reference). If x_1 lies at or to the left of p, then α still decreases by $2d$, as above. If x_1 lies to the right of s_2 (at point z) then α decreases from $2(d + 2d_2 + d_3$ to $d_3)$ to $2(d_3)$, which is a decrease of at least $2d$. If x_1 lies between s_1 and s_2 (at a distance of d_4 from s_1 and a distance of d_5 from s_2), then α decreases from $2(d_4 + d_4 + d_5 + d)$ to $2(d_5)$, which is a decrease of at least $2d$. Finally, if x_1 lies between s_1 and p (at a distance of d_6 from s_1 and a distance of d_7 from p), then α decreases from $2(d_6 + d + d_2)$ to $2(d_2 + d_6)$, which is a decrease $2d$.

Thus the α part of the potential always decreases by at least $2d$, which is sufficient to cover the cost of moving plus the increase in β.

EXERCISE (bonus): Analyze the second case, where p lies in the interval between s_1, s_2, to show that Double Coverage is 2-competitive. Remember that the adversary must have a server on p. Note that in this case, β will cause the potential to decrease by $2d$.

8.2.3 Examples of More On-line Problems

Two other famous on-line problems are the Union-Find problem [37] which is the basis for Kruskal's Minimum Spanning Tree Algorithm and that of finding efficient priority queues for certain applications, such as for Prim's MST algorithm and for Dijkstra's algorithm. Fibonacci heaps, introduced in [15] are among those data structures that theoretically provide the best performance for these algorithms.

Lots of of network problems (e.g., routing packets in networks, file and process migration) and numerous scheduling problems are important on-line problems.

8.3 Dynamic Algorithms

8.3.1 Minimum Spanning Tree

8.3.2 Connected Components

8.4 Exercises

8.5 Programming Assignments

Chapter 9

Mathematical Preliminaries

9.1 Functions

A function f is a mapping from A (domain) to B (co-domain) such that $f(a) = b$, that is, each element in the domain is mapped to exactly one element of the co-domain.

9.1.1 Basic definitions

A function $f(n)$ is monotonically increasing if $m \geq n \Rightarrow f(m) \geq f(n)$.

A function $f(n)$ is monotonically decreasing if $m \geq n \Rightarrow f(m) \leq f(n)$.
A function $f(n)$ is strictly increasing if $m < n \Rightarrow f(m) < f(n)$.
A function $f(n)$ is strictly decreasing if $m > n \Rightarrow f(m) < f(n)$.
Floor: the greatest integer less than or equal to a value. $\lfloor \frac{7}{3} \rfloor = 2$.
Ceiling: the least integer greater than or equal to a value. $\lceil \frac{5}{3} \rceil = 2$.

9.1.2 Special Functions

Recall that $a^0 = 1, a^1 = a, a^{-1} = \frac{1}{a}$.
Also, $(am)^n = a^n m^n, a^m a^n = a^{m+n}$.

The latter functions are exponential functions. Any positive exponential function (i.e., one with a positive base and exponent) grows faster asymptotically than any polynomial function. For example, $2^{n^{\frac{1}{2}}}$ dominates $n^{23} + 5n^2$. Likewise, $n^{\frac{1}{2}}$ dominates $37 \log^4 n$, even though the latter function is larger for “small” values of n [Note: $\log^k n = (\log n)^k$].

$\log_2 n$ will be denote by $\log n$ as it is so commonly used. $\ln n$ will refer to $\log_e n$ and $\log\log n$ will refer to $\log(\log n)$.

Recall from algebra the following definitions and properties of logarithms:

$\log_c(ab) = \log_c a + \log_c b$, $\log_b a^n = n \log_b a$, $\log_b a = \frac{\log_c a}{\log_c b}$,
$\log_b(1/a) = -\log_b a$, $\log_b a = \frac{1}{\log_a b}$.

A nice problem is to show that $\log_{10} n = \Theta(\log_2 n)$. In other words, the base of the log is not so important to us. Obviously, $\log_{10} n \leq \log_2 n$. For the other direction, note that $2^{3.322}$ is slightly greater than 10. So $\log_{10} n^{3.322} > \log_2 n$, which implies $3.322 \log_{10} n > \log_2 n)$.

Recall that $2^{\log n} = n$ and $2^{\frac{\log n}{2}} = \sqrt{n}$.

Furthermore, $\log^* n$ (the iterated log) is used to denote the smallest value i such that $\log\log\log\log n \leq 1$, where the log operation is done i times.

Examples:

$\log 64 = 6$ since $2 * 2 * 2 * 2 * 2 * 2 = 64$

$\log^* 16 = 3$ since $\log 16 = 4, \log 4 = 2$, and $\log 2 = 1$.

Other definitions:

$n! = n * (n-1)!$, where $0! = 1$ (factorial), note that $n! = O(n^n)$
$F_0 = 0, F_1 = 1$, and $F_i = F_{i-1} + F_{i-2}$ defines the Fibonacci sequence with f_i the i^{th} Fibonacci number

Ackermann's function: a rapidly-growing function of two parameters:

$A(1, j) = 2^j$, for $j \geq 1$

$A(i, 1) = A(i-1, 2)$, for $i \geq 2$

$A(i, j) = A(i-1, A(i, j-1))$, for $i, j \geq 2$

A simplified variation of Ackermann's function is represented by:

n	F(n)
0	1
1	2
2	4
3	16
4	65,536
5	2^{65536}

This function is also super-exponential and observe that $F^{-1}(n)$ is almost constant. That is, for $n \leq 2^{65536}$, $F^{-1}(n) \leq 5$.

Stirling's approximation: $n!$ is approximately equal to $(\frac{n}{e})^n\sqrt{2n\pi}$.

9.2 Series and Sums

If an algorithm is composed of several parts, its complexity is the sum of the parts. These parts may include loops whose number of iterations depends on the input. Therefore we need techniques for summing expressions. The sum $a_1 + a_2 + \ldots + a_{n-1} + a_n$ is written as

$$\sum_{i=1}^{n} a_i$$

This means we run the variable, or index, i from 1 to n.
We can manipulate summations in the obvious ways:
Some common summations.
The arithmetic series:

$$\sum_{i=1}^{n} a_i = \frac{n(n+1)}{2} = O(n^2)$$

The geometric series:

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$$

When $|a| < 1$ we have that

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a}$$

The harmonic series:

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = \ln n + O(1) = O(\ln n) = O(\log n)$$

Let's consider the following.

$$\sum_{i=1}^{n} \log i$$

Note that

$$\sum_{i=n/2}^{n} \log i \geq \frac{n}{2} \log \frac{n}{2}$$

$$= \frac{n}{2} \log n - \frac{n}{2} \log 2 = \frac{n}{2} \log n - n = \theta(n \log n)$$

Certain series, called telescoping series, are particularly easy to manipulate. In general, these are of the form

$$\sum_{k=1}^{n} (a_k - a_{k-1}) = a_n - a_0$$

As an example:

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

To see why, note that the series is equal to (1 - 1/2) + (1/2 - 1/3) + (1/3 - 1/4) ... (1/(n-1) - 1/n), which is the mirror image of the general form shown above.

As a simple example of another evaluation technique:

$$F(n) = \sum_{i=0}^{n} 2^i = 1 + 2 + \ldots + 2^n$$

We can compute this sum by multiplying the equation by 2, giving us

$$2F(n) = 2 + 4 + 8 + ... + 2^n + 2^{n+1}.$$

We can now use our telescoping trick by

$$2F(n) - F(n) = 2^{n+1} - 1$$

which implies that $F(n) = 2^{n+1} - 1$.

Another simple method is the “sloppy rule of sums.”

$$\sum_{i=0}^{n} f(i,n) = O(n * \max\{f(i,n)\})$$

This tells us that

$$\sum_{i=0}^{n} n - i = O(n^2)$$

Splitting the terms. Sometimes we can split a sum into two or more parts that are simpler to analyze.

Consider

$$\sum_{k=0}^{\infty} k^2/2^k$$

Observe that the ratio between two consecutive terms is

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

which is at most 8/9 for $k \geq 3$.

Thus we split the sum into two parts:

$$\sum_{k=0}^{2} \frac{k^2}{2^k} + \sum_{k=3}^{\infty} \frac{k^2}{2^k}$$

By our observation above, letting the terms in the second summation be denoted as $a_i, 0 \leq i \leq \infty$, we have that $a_k \leq a_0 * (\frac{8}{9})^k$ with $a_o = \frac{9}{8}$

we can re-write the sum then as

$$\sum_{k=0}^{2} \frac{k^2}{2^k} + \sum_{k=3}^{\infty} \frac{9}{8}(\frac{8}{9})^k$$

which can be bounded from above by

$$O(1) + \sum_{k=0}^{\infty} \frac{9}{8} (\frac{8}{9})^k =$$

$$O(1) + \frac{9}{8} \sum_{k=0}^{\infty} (\frac{8}{9})^k$$

Since the second term is a geometric series with $x < 1$, it too is $O(1)$. Hence the sum is $O(1)$.

Let's do a similar one. Consider

$$\sum_{k=1}^{\infty} \frac{k}{3^k}$$

The ratio of two consecutive terms is

$$\frac{(k+1)/3^{k+1}}{k/3^k} = \frac{1}{3}\frac{k+1}{k} \leq \frac{2}{3}$$

Hence

$$\leq \sum_{k=1}^{\infty} \leq \sum_{k=1}^{\infty} \frac{k}{3^k} \sum_{k=1}^{\infty} \frac{1}{3}(\frac{2}{3})^k \leq \frac{1}{3} \sum_{k=0}^{\infty} (\frac{2}{3})^k \leq$$

The latter sum is geometric, and has the value $1/3 * 1/(1 - 2/3)$ which is equal to 1.

Consider the harmonic number

$$H_n = \sum_{k=1}^{n} \frac{1}{k}$$

If we split the sum into $\log n$ pieces, we can easily bound it by $O(\log n)$. This is done by using a double summation.

$$\sum_{k=1}^{n} \frac{1}{n} \leq \sum_{i=0}^{\lceil \log n \rceil} \sum_{j=0}^{2^i - 1} \frac{1}{2^i + j} \leq$$

$$\sum_{i=0}^{\lceil \log n \rceil} \sum_{j=0}^{2^i - 1} \frac{1}{2^i} \leq \sum_{i=0}^{\lceil \log n \rceil} 2 = O(\log n)$$

The first term in the double summation breaks in into $\log n$ pieces. The second term is the 1/k – where $k = 2^i + j$. We can then sweep the j term under the rug. What remains is simply $1/2^i * c * 2^i$, for $c < 1$, which is no more than 2. That is, each term in the summation is at most 2. Hence the $O(\log n)$ bound.

We can bound the harmonic number another way, using an integral.
It can be shown that

$$\int_{m-1}^{n} f(x)dx \le \sum_{k=m}^{n} f(k) \le \int_{m}^{n+1} f(x)dx$$

In other words, we are bounding a discrete sum by a kind of continuous sum.

Thus

$$\sum_{k=1}^{n} \frac{1}{k} \le 1 + \sum_{k=2}^{n} \frac{1}{k} \le (\int_{1}^{n} \frac{dx}{x}) + 1 \le \ln n + 1$$

thus

9.3 Induction and Proofs

Mathematical induction is a powerful proof technique that is used on discrete (countable), infinite sets, such as the integers. To prove an assertion we first show that it holds for a base case (such as $n = 0, n = 1$, or whatever is appropriate in the given situation). We then assume (the inductive hypothesis) that the assertion holds for some value k greater than the base case value. We then show (the inductive step) that the assertions holds for the value $k + 1$. If so, then the assertion holds for every value greater than or equal to the base value. [Likewise, we may perform the induction from $k - 1$ to k, rather than from k to $k + 1$].

Example: Assertion:

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

Base case. $n = 1$. The sum has only one term, 1, and thus the value of the summation is 1. And we have that $1 * (1 + 1)/2 = 1 * 2/2 = 1$.

Inductive hypothesis: Assume it holds for n.

Inductive step: Show that it holds for $n+1$. That is, we want to show that $\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$.

The idea is to find some way to use the inductive hypothesis to our advantage. In this case, we do this by taking the sum above and separating out the last term:

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

By inductive hypothesis, we have that

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

which is equal to

$$(n^2+n)/2 + (2n+2)/2 = (n^2+3n+2)/2 = (n+1)(n+2)/2$$

which is what we were trying to prove.

A sometimes useful variation on induction is strong induction. In strong induction, we do not move from k to $k+1$, rather our inductive hypothesis assumes that the assertion holds all $m < k+1$. We then proceed to prove the inductive hypothesis.

Assertion: any integer $r \geq 2$ can be expressed as a finite product of primes (and 1).

Base Case: $r = 2$. 1*2=2.

Inductive hypothesis: Assume the assertion true for all $k < n$.
Inductive Step: Show for n. There are two cases.

1) If n is prime we are done, as $n * 1 = n$.

2) If n is not prime then it must be that $n = pq$ for some $p, q < n$. We now simply invoke the inductive hypothesis on each of p and q, yielding the

desired product for n.

Note that induction from $n-1$ to n would not work in this case! In fact, one can also prove that this product of primes is unique (this is the "fundamental theorem of arithmetic"). Proving uniqueness is a bit more difficult and most proofs rely on Euclid's GCD algorithm. The interested reader may wish to research the details.

As we saw above, induction gives us a nice way to evaluate certain summations.

Show that

$$\sum_{k=0}^{n} 3^k = O(3^n) \leq c * 3^n$$

Base case. $n = 0$. The sum is $3^0 = 1$, which holds so long as $c \geq 1$.

Assume it holds for n.

Prove it holds for $n+1$.

$$\sum_{k=0}^{n+1} 3^k = (\sum_{k=0}^{n} 3^k) + 3^{n+1} \leq c3^n + 3^{n+1} = (\frac{1}{3} + \frac{1}{c})c3^{n+1} \leq c3^{n+1}$$

This holds as long as $1/3 + 1/c \leq 1$ (or $c \geq 3/2$, which is ok given our base case), otherwise we are changing our constant in midstream, which is not what a constant does!!

Proof by contradiction: This is another valuable proof technique, useful in proving the correctness of an algorithm or other properties. In this technique, we choose some part of an assertion, suppose that it is not true, and derive a contradiction (i.e., something which is the complement/negation of one of our assumptions or a clear violations of our mathematical system, e.g. 0=1).

Let n be an integer, with n^2 even. Then n is even.
Proof: Suppose, by way of contradiction, that n^2 is even and n is odd. Thus there is a k such that $2k+1 = n$. From which we have

$n^2 = (2k+1)^2$
$= 4k^2 + 4k + 1$
$= 2(2k^2 + 2k) + 1$, which is clearly odd, contradicting our assumption that n is odd. Hence we conclude that n is even.

There are an infinite number of prime numbers.
Proof. Suppose otherwise, that are are only finitely many primes. Then there must be a largest prime number, say n. Consider the number $N = n! + 1 = n*(n-1)*(n-2)\ldots*3*2+1$. Then $N > n$ and so N is not prime. We know N has a unique prime factorization, as all integers do. Let p be a prime factor of N. Since n is the largest prime number, $p \leq n$. However, when we divide N by all the numbers from 2 to n, we get a remainder of 1 each time, i.e., $N \bmod i = 1, 2 \leq i \leq n$. Therefore no number between 2 and n divides N. Thus p must be greater than n, contradicting our assumption that n was the largest prime.

9.4 Graphs

Let $G = (V, E)$ be a graph with vertex set V and edge set E. Usually, we let $|V| = n, |E| = m$.

An outerplanar graph is one that can be drawn in the plane with all vertices bordering the exterior face.

A (vertex) coloring of a graph is an assignment of integers (colors) to the vertices so that any two adjacent vertices have different colors.
Theorem. Any outerplanar graph can be colored using at most three colors.

Proof. By induction on n, the number of vertices.

Base Case: $n = 1$. Then only one vertex, so one color suffices.

Inductive Hypothesis. Assume true for outerplanar graphs with n vertices.

Inductive Step. Consider an outerplanar graph with $n+1$ vertices. There are two cases:

(1) Suppose there is a cut-vertex v, a vertex whose deletion disconnects the graph into components $G_1, \ldots, G_k, k > 1$. Let G'_i denote G_i with vertex v added back, i.e., the subgraph of G induced by $V(G_i) \cup \{v\}$. By the induc-

tive hypothesis, each of these G'_i are 3-colorable, and we may assume that v receives color 1 in each of these (else the colorings may be easily modified so this is the case). These colorings then induce a coloring of G.

(2) Suppose G has no cut-vertex. Then G contains a Hamiltonian cycle C (Exercise: prove that a 2-connected outerplanar graph contains a Hamiltonian cycle). If $C = E(G)$, then G is 3-colorable, since any cycle can be colored using at most 3 colors. Else there is a chord uv on C. Cut G along uv into two graphs, each containing uv. Inductively 3-color each graph with u and v receiving colors 1 and 2, respectively, in each of the colorings. These two colorings form a 3-coloring of G. □
Exercise: Prove the above by induction on the number of faces in the graph, using 1 as the base case.

9.5 Exercises

9.6 Programming Assignments

Chapter 10

Data structures

10.1 Lists

10.2 Trees and Heaps

10.3 Programming Assignments

Bibliography

[1] L. Adleman, C. Pomerance, and R. Rumely (1983), "On distinguishing prime numbers from composite numbers," *Annals of Mathematics* vol. 117, pp. 173-206

[2] M. Agarwal, N. Saxena, and N. Kayal (2002) "Primes in P," manuscript, CSE Department, Indian Institute of Technology, Kanpur, http://www.cse.iitk.ac.in/news/primality.html

[3] A. Aho, J. Hopcroft, and J. Ullman (1974), **The Design and Analysis of Computer Algorithms**, Addison-Wesley, Reading, Mass.

[4] S. Arora and C. Lund (1997), Hardness of Approximation, in Approximation Algorithms for NP-Hard Problems, D. Hochbaum editor, PWS Publishing

[5] J. Balcazar, J. Diaz, and J. Gabarro (1990), Structural Complexity II, Springer-Verlag

[6] J. Bell and G. Gupta (1993), "An evaluation of self-adjusting binary search tree techniques," *Software – Practice and Experience* vol. 23, pp. 369-382

[7] A. Borodin and R. El-Yaniv (1998) **Online Computation and Competitive Analysis**, Cambridge Univ. Press

[8] D. Bovet and P. Crescenzi (1994), **Introduction to the Theory of Complexity**, Prentice Hall, Englewood Cliffs, NJ

[9] S. Carlsson (1987), "Average-Case Results on HeapSort," *BIT*, vol. 27, pp. 2-17

[10] M. Chrobak, H. Karloff, T.Payne, and S. Vishwanathan (1991), "New Results on Server Problems," *SIAM Discrete Math.*, vol. 4, pp. 172-181

[11] T. Cormen, C. Leiserson, and R. Rivest (1990), **Introduction to Algorithms**, MIT Press, Cambridge, Mass.

[12] R. Downey and M. Fellows (1995), Fixed-Parameter Tractability and Completeness I: Basic Results, *SIAM J. Comput.*, vol. 24, pp. 873-921

[13] R. Dutton (1993), "Weak-heap sort," *BIT*, vol. 33, pp. 372-381

[14] R. Dutton and W. Klostermeyer (1999), "A Faster Algorithm for Least Deviant Path," *J. Comb. Math. and Comb. Comput.*, to appear [Note: the "slow" algorithm mentioned in the introduction of this paper actually runs in $O(|E|^{2.586})$ time, not $O(|E|^{1.793})$ time.]

[15] M. Fredman and R. Tarjan (1987), "Fibonacci Heaps and their uses in Improved Network Optimization," *Journal of the ACM* vol. 34, pp. 596-615

[16] M. Furer and B. Raghavachari (1992), Approximating the Minimum Degree Spanning Tree to within One from the Optimal Degree, *Proc. Third ACM-SIAM Symp. on Disc. Algorithms*, pp. 317-324

[17] M. Garey and D. Johnson (1979), Computers and Intractability, W. H Freeman

[18] M. Goemans and D. Williamson (1995), Improved Approximation Algorithms for Maximum Cut and Satisfiability Using Semidefinite Programming, *J. ACM*, vol. 42, pp. 1115-1145

[19] J. Hopcroft and R. Tarjan (1974), Efficient Planarity Testing, *J. ACM*, vol. 21, no. 4, pp. 549-568

[20] W. F. Klostermeyer and M. Muslea (1996), "Techniques for Algorithm Design and Analysis: Case Study of a Greedy Algorithm," *Informatica*, vol. 20, no. 2, pp. 185-190

[21] E. Koutsoupias and C. Papadimitriou (1995), "On the k-server Conjecture," *J. ACM*, vol. 42, pp. 971-983

[22] R. Ladner (1975), On the Structure of Polynomial-time Reducibility, *J. ACM*, vol. 22, pp. 155-171

[23] M. Manasse, L. McGeoch, and D. Sleator (1990), "Competitive Algorithms for Server Problems," *J. Algorithms*, vol. 11, pp. 208-230

[24] G. Miller (1976), "Reimann's Hypothesis and Test for Primality," *J. Comput. System Sci.*, vol. 13, pp. 300-317

[25] B. Moret and H. Shapiro (1991), **Algorithms from P to NP**, Benjamin Cummings, Redwood City, CA

[26] C. Papadimitriou and M. Yannakakis (1991), Optimization, Approximation, and Complexity Classes, *J. Computer and System Sciences*, vol. 43, pp. 425-440

[27] M. Rabin (1980), Probabilistic algorithm for testing primality, *Journal of Number Theory*, vol. 12, pp. 128-138

[28] N. Robertson, D. Sanders, P. Seymour, and R. Thomas (1997), The Four-Color Theorem, *J. Combinatorial Theory (B)*, vol. 70, pp. 2-44

[29] R. Schaffer and R. Sedgewick (1993), "The Analysis of Heapsort," *J. Alg.*, vol. 15, pp. 76-100

[30] Shih Wei-Kuan and Hsu Wen-Lian (1999), "A new planarity test," *Theoretical Computer Science* vol. 223, no. 1-2, pp. 179-191

[31] S. Skiena (1998), **The Algorithm Design Manual**, Springer-Verlag, New York

[32] R. Sedgewick (1998), **Algorithms in C, Parts 1-4** (Third Edition), Addison-Wesley, Reading, Mass.

[33] R. Sedgewick and P. Flajolet (1996), **An Introduction to the Analysis of Algorithms**, Addison-Wesley, Reading, Mass.

[34] J. Setubal and J. Meidanis (1997), **Introduction to Computational Molecular Biology**, PWS publishing, Boston, Mass.

[35] D. Sleator and R. Tarjan (1985) "Amortized Efficiency of List Update and Paging Rules," *Communications of the ACM*, vol. 28, pp. 202-208

[36] J. Sussman (1991), **Instructor's Manual to Accompany Introduction to Algorithms**, MIT Press, Cambridge, Mass.

[37] R. Tarjan (1975), Efficiency of a good but not Linear Set Union Algorithm, *J. ACM*, vol. 22, pp. 215-225