A Brief Introduction to Algorithms

Paul Cull

Department of Computer Science
Oregon State University

April 6, 2007
Contents

1 INTRODUCTION ............................................. 2
  1.1 What Is An Algorithm? ................................ 2
  1.2 What Is Analysis Of Algorithms? ................. 2
    1.2.1 Proofs of correctness ........................ 4
    1.2.2 Comparing Algorithms ....................... 5
    1.2.3 Best Algorithms ............................ 6

2 TOWERS OF HANOI: AN ANALYSIS OF A PROBLEM .... 7
  2.1 The Towers of Hanoi Problem ..................... 7
  2.2 A Recursive Algorithm ........................... 7
  2.3 HANOI is correct ................................ 8
  2.4 HANOI makes the fewest possible moves .......... 11
    2.4.1 Bottleneck ................................ 13
  2.5 Time and space used .............................. 13
  2.6 Improved Algorithms ............................ 16

3 SOME INDUCTIONS ................................... 25
  3.1 What Is Induction? ............................... 25
  3.2 Inductions for easy Sums ....................... 26
  3.3 Euclid’s Algorithm ............................. 29

4 DIVIDE-AND-CONQUER ..................................... 32
  4.1 What is Divide-and-Conquer? .................... 32
  4.2 The General Form for Divide-and-Conquer Algoritms ... 34
  4.3 Divide-and-Conquer Difference Equations .... 35
  4.4 Some Divide-and-Conquer Examples ............ 36
    4.4.1 Mergesort ................................ 36
    4.4.2 Polynomial Multiplication .................. 36
    4.4.3 Integer Multiplication ...................... 38
    4.4.4 Matrix Multiplication ...................... 39
    4.4.5 Solving a Linear System of Equations ...... 40
    4.4.6 Newton’s Method .......................... 41
  4.5 Polynomial Multiplication and Fast Fourier Transform ... 42
  4.6 How Practical Are Divide-and-Conquer Algorithms? ... 46
10 Worked Examples

10.1 All Simple Roots .............................................. 105
10.2 One Multiple Root ............................................... 110
10.3 One Multiple Root, Several Simple Roots ............... 113
10.4 The Input is $\gamma_1^n p_1(n) + \gamma_2^n p_2(n)$ ........ 113

11 Further Reading

11.1 Books ............................................................. 115
11.2 Articles .......................................................... 116
Chapter 1

INTRODUCTION

1.1 What Is An Algorithm?

An algorithm is a procedure which solves a problem and is suitable for implementation as a program for a digital computer. This informal definition makes two important points. First, an algorithm solves a problem. A computer program may not solve a problem. For example, there are computer programs which never terminate. It would be very difficult to say that such a program does anything, let alone solve a problem. Second, each step of the algorithm should be well defined and be representable, at least in principle, by a program. For example, \( a = P/q \) would not be well defined if \( q \) could be 0; "find the smallest \( x \) such that \( P(x) \) is true" would not be well defined if no such \( x \) existed, or if \( x \) were allowed to range over the positive and negative integers and \( P(x) \) were true for all negative \( x \). "Add a dash of salt" would not be an acceptable instruction in a computer program unless one had created a model world and a mapping which would translate the instruction into the modification of some location in a computer's memory.

For our purposes, and for many purposes, the above informal definition of algorithm is sufficient, but we should point out that a formal definition of algorithm can be created. This has been done by Turing, Markoff, and others [Rogers, 1967]. The satisfying thing about all these formal definitions is that they define the same class of algorithms. If we have an algorithm in one of these senses, then there is an equivalent algorithm in any one of the other senses. Further, the existence of a formal definition means that if we run into difficulties with the informal definition, we could use the formal definition to unravel the difficulty.

1.2 What Is Analysis Of Algorithms?

The task of analysis of algorithms is three-fold:
1. To produce provably correct algorithms, that is, algorithms which not only solve the problem they are designed to solve, but which also can be demonstrated to solve the problem;

2. To compare algorithms for a problem with respect to various measures of resources (e.g., time and space), so that one can say when one algorithm is better than another;

3. To find, if possible, the best algorithm for a problem with respect to a particular measure of resource usage.

Each of these tasks will be examined in more detail later, but first I want to say a few words about where analysis of algorithms fits in the worlds of mathematics and computer science. Mathematics has two great branches, pure mathematics and applied mathematics. Mathematics also has two major techniques, proof and computation. It would be tempting to suggest that pure mathematics uses proof and applied mathematics uses computation, but both pure and applied mathematics use both proof and computation. The difference between the two branches is the world they work in. Pure mathematics works in an abstract world. It needs no reference to a real world. Applied mathematics, on the other hand, assumes there is a real world and then constructs an abstract world. The mapping or correspondence between entities of the real and abstract worlds form an essential part of applied mathematics. Where does computer science fit in? Computer science is the science which deals with computation and computing devices. In its theoretical branches, computer science uses the mathematical techniques of proof and computation, while in its more practical branches, computer science uses engineering techniques and experimental techniques. Analysis of algorithms, like applied mathematics, assumes a real world. This real world contains actual computers and actual computer programs. From this real world, an abstract world of abstract computers and abstract programs is constructed. This construction is usually informal and exact definitions of abstract entities are often not stated, but a number of real world limitations disappear in the abstract world. For example, real computers have a fixed finite memory size and they have an upper bound on the size of numbers which can be represented. In the abstract world, these limitations do not exist. Finite but unbounded memories are assumed to exist. No bound on the size of numbers exists.

Again like applied mathematics, analysis of algorithms uses the techniques of proof and computation to deal with the entities in its abstract world, and like applied mathematics, one must be cautious in applying results from the abstract world to the real world. There are examples of algorithms which would work very quickly if arbitrarily large numbers could be used, but implementing these algorithms on real computers results in algorithms which are much slower. As another example, there are algorithms which can be shown to be quicker than other algorithms, but only if the input is astronomically large. These examples make perfect sense in the abstract world, but have little or no relevance for the real world. Analysis of algorithms is the applied mathematics
of computer science. Whether it should be called mathematics or computer science usually depends on who is doing it. If a mathematician does analysis of algorithms, it may be called mathematics, but if a computer scientist does analysis of algorithms, it is usually called computer science.

1.2.1 Proofs of correctness

The first of the three tasks of analysis of algorithms is to produce provably correct algorithms. This task suggests that we need a methodology to both produce algorithms and to produce proofs of the correctness of these algorithms. Later we will consider two design strategies:

1. (a) to solve a problem, break it into smaller problems of the same kind;

2. (b) to solve a problem, search an answer space until you find the answer to the problem you have.

Proving correctness of algorithms based on a search strategy is usually relatively easy: Show that the answer to your problem is in the space; show that your algorithm searches the whole space, or that if your algorithm decides not to search a portion of the space, then your algorithm has established that your answer cannot lie in the portion not to be searched.

Proving correctness of algorithms based on breaking a problem into smaller problems can most readily be done by using mathematical induction. You establish that your algorithm correctly solves all small enough cases of the problem. Your algorithm will usually have a section which deals with these small cases, but particularly when the small cases may be of size 0 and no action by the algorithm is required, the section for small cases may not exist. The other section of your algorithm will deal with larger cases by breaking them into smaller cases and combining the solutions to the smaller cases to give the solution to the larger problem. You then have to show that if the smaller problems are solved correctly, then your algorithm combines these solutions correctly to yield the solution to the larger problem.

The proof procedure just outlined assumes that your algorithm is recursive, that is, to solve large problems the algorithm calls itself to solve smaller problems, but proofs by mathematical induction are not limited to recursive algorithms. Inductive proofs are also natural when dealing with iterative algorithms which are based on loops, for example, WHILE loops, FOR loops, DO loops, or REPEAT...UNTIL loops. Inductive proofs of correctness of loop programs involve the creation of a loop invariant, a truth-valued function, which is true on each iteration of the loop and which states that the loop has completed the desired computation when the loop is exited. Such a proof also requires one to show that the loop does in fact terminate. When there are several loops, the correctness of each loop must be established, and then it must be shown that the correctness of each loop implies the correctness of the whole algorithm. The most difficult part of such proofs is usually the identification of the loop invariants. Without a knowledge of how and why the algorithm was designed, such
proofs are nearly impossible. Some programming languages, like EUCLID, now allow the specification of the loop invariants and other propositions so that at least in principle it would be possible to mechanically establish the correctness of a fully annotated program.

While it is often easier to establish the correctness of a recursive algorithm, the overwhelming majority of programs are iterative rather than recursive. Why is this true? One reason is that a number of common programming languages like FORTRAN, BASIC, and COBOL do not permit recursion. Another reason is that many programmers believe that recursive programs are always slower than iterative programs. While this may have been true in languages like ALGOL and PL/1, and when recursive programs in ALGOL were compared to iterative programs in FORTRAN, the speed advantage of iterative over recursive programs has disappeared in languages like PASCAL and C.

1.2.2 Comparing Algorithms

For any solvable problem there will be an infinity of algorithms which solve the problem. How do we decide which is the best algorithm? There are a number of possible ways to compare algorithms. We will concentrate on two measures: time and space. We would like to say that one algorithm is faster, uses less time, than another algorithm if when we run the two algorithms on a computer the faster one will finish first. Unfortunately, to make this a fair test we would have to keep a number of conditions constant. For example, we would have to code the two algorithms in the same programming language, compile the two programs using the same compiler, and run the two programs under the same operating system on the same computer, and have no interference with either program while it is running. Even if we could practically satisfy all these conditions, we might be chagrined to find that algorithm A is faster under conditions C, but that algorithm B is faster under conditions D.

To avoid this unhappy situation we will only calculate time to order. We let n be some measure of the size of the problem, and give the running time as a function of n. For example: we could use the number of digits as the measure of problem size if the problem is the addition or multiplication of two integers; we could use the number of elements if the problem is to sort several elements; we could use the number of edges or the number of vertices if the problem is to determine if a graph has a certain property. We do not distinguish running times of the same order. For our purposes two functions of n, \( f(n) \) and \( g(n) \), have the same order if for some \( N \) there are two positive constants \( c_1 \) and \( c_2 \) so that \( c_1 |g(n)| \leq |f(n)| \leq c_2 |g(n)| \) for all \( n \geq N \). We symbolize this relation by \( f(n) = \Theta(g(n)) \), read \( f(n) \) is order \( g(n) \), or \( f(n) \) is Big Theta of \( g(n) \). Thus we will consider two algorithms to take the same time if their running times have the same order. In particular, we do not distinguish between algorithms whose running times are constant multiples of one another.

If we find that algorithm A has a time order which is strictly less than algorithm B, then we can be confident that for any large enough problem algorithm A will run faster than algorithm B regardless of the actual conditions. On the
other hand, if algorithms A and B have the same time order, then we will not predict which one will be faster under a given set of actual conditions.

The space used by an algorithm is the number of bits the algorithm uses to store and manipulate data. We expect the space to be an increasing function of \( n \), the size of the problem. This space measurement ignores the number of bits used to specify the algorithm, which has a fixed constant size independent of the size of the problem. Since we have chosen bits as our unit, we can be more exact about space than we can be about time. We can distinguish an algorithm which uses \( 3n \) bits from an algorithm which uses \( 2n \) bits. But we will not distinguish an algorithm which uses \( 3n + 7 \) bits from an algorithm which uses \( 3n + 1 \) bits, because we can hide a constant number of bits within the algorithm itself.

1.2.3 Best Algorithms

The third task of analysis of algorithms is to find the best algorithm with respect to a particular measure of resource usage. This involves proving “lower bounds,” that is, showing that every algorithm which solves the problem must use at least so much of the particular resource. To establish a best algorithm, one must have both a proof of a lower bound and an algorithm which uses no more than this lower bound. Here a distinction should be made between bounds for an algorithm and bounds for a problem. If one establishes an upper bound on a particular resource used by an algorithm for a problem, then one has an upper bound both for the algorithm and for the problem. If one establishes a lower bound for a problem, then one has a lower bound for all algorithms which solve the problem. But demonstrating a lower bound for one algorithm for a problem does not establish a lower bound for the problem.
Chapter 2

TOWERS OF HANOI: AN ANALYSIS OF A PROBLEM

2.1 The Towers of Hanoi Problem

In this section, we will demonstrate the three-fold task of analysis of algorithms using the Towers of Hanoi problem. We have chosen the Towers of Hanoi problem because each of the tasks can be demonstrated rather easily, and a best algorithm can be discovered. Material in this section is based on Cull and Ecklund [1985].

In the Towers of Hanoi problem, one is given three towers, usually called A, B, and C, and $n$ disks of different sizes. Initially the disks are stacked on tower A in order of size with $disk_n$, the largest, on the bottom, and $disk_1$, the smallest, on the top. The problem is to move the stack of disks to tower C, moving the disks one at a time in such a way that a larger disk is never stacked on top of a smaller disk. An extra constraint is that the sequence of moves should be as short as possible. An algorithm solves the Towers of Hanoi problem if, when the algorithm is given as input $n$ the number of disks, and the names of the towers, then the algorithm produces the shortest sequence of moves which conforms to the above rules.

2.2 A Recursive Algorithm

The road to a best algorithm starts with some algorithm which one then attempts to improve. One often uses some sort of strategy to create an algorithm. A very useful strategy is to look at the problem and see if the solution can be expressed in terms of the solutions of several problems of the same kind, but of smaller size. This strategy is usually called divide-and-conquer. If the
problem yields to the divide-and-conquer approach, one can construct a recursive algorithm which solves the problem. This construction also gives almost immediately an inductive proof that the algorithm is correct. Time and space analyses of a divide-and-conquer algorithm are often straightforward, since the algorithm directly gives difference equations for time and space usage.

While these divide-and-conquer algorithms have many nice properties, they may not use minimal time and space. They may, however, serve as a starting point for constructing more efficient algorithms.

Consideration of the Towers of Hanoi problem leads to the key observation that moving the largest disk requires that all of the other disks are out of the way. Hence the \( n - 1 \) smaller disks should be moved to tower B, but this is just another Towers of Hanoi problem with fewer disks. After the largest disk has been moved the \( n - 1 \) smaller disks can be moved from B to C; again this is a smaller Towers of Hanoi problem. These observations lead to the following recursive algorithm:

\[
\text{PROCEDURE HANOI(A,B,C,n) }
\]

\[
\begin{align*}
\text{IF } n=1 & \text{ THEN move the top disk from tower A to tower C} \\
\text{ELSE } & \text{ HANOI(A,C,B,n-1)} \\
& \text{ move the top disk from tower A to tower C} \\
& \text{ HANOI(B,A,C,n-1)}. \\
\end{align*}
\]

Is this the best algorithm for the problem? We will show that this algorithm has minimal time complexity, but does not have minimal space complexity. First, though, we prove that the algorithm correctly solves the problem, the first task of analysis of algorithms as outlined in the Introduction.

### 2.3 HANOI is correct

**Proposition 1.** The recursive algorithm HANOI correctly solves the Towers of Hanoi problem.

In the following proof, we will use the two rules for this puzzle:

1) Only one disk is moved at a time,

2) A larger disk is never placed on top of a smaller disk.

For the algorithm to be correct, it must obey these two rules as well as accomplish its task of moving \( n \) disks from one tower to another.

The proof is by *induction* (see Chapter 3) on the number of disks. To apply induction we first need an inductive hypothesis. Let us call this hypothesis \( HYP(n) \) and define it as follows:

\[
HYP(n) = ' \text{HANOI(A,B,C,n) correctly moves the } n \text{ disks } 1, 2, \ldots, n \text{ from tower A to tower C}'.
\]
Here as stated above, correctly means that rules (1) and (2) are obeyed.

The proof has TWO parts, a base case (here we will use \( n = 1 \) as the base case), and an induction step – showing that if the algorithm HANOI works correctly for \( n - 1 \) disks then it works correctly for \( n \) disks. A minor but significant point is that the induction goes backwards, that is, we consider what the algorithm will do with \( n \) disks and following the steps we find that the algorithm deals with \( n - 1 \) disks first and we can assume by \( HYP(n - 1) \) that the algorithm correctly (obeying (1) and (2)) moves \( n - 1 \) disks.

Notice further that \( HYP(n) \) is a truth-valued function – \( HYP(n) \) can only have the value TRUE or the value FALSE. \( HYP(n) \) doesn’t do anything, instead it simply observes whether or not HANOI does its job.

Now to the proof.

**BASE CASE:**
We will try to show that \( HYP(1) \) is a true. From the definition of \( HYP \),
\( HYP(1) \) says that HANOI(A, B, C, 1) correctly moves disk 1 from tower A to tower C. Now we look at the algorithm HANOI with the inputs (A, B, C, 1). The **IF** condition ‘\( n = 1 \)’ is **TRUE**, so the algorithm executes the **THEN** part and does MOVE(A,C). Assuming that MOVE(A,C) moves the top disk off tower A and puts this disk on top of tower C, and assuming that the top disk on tower A was disk 1, then HANOI(A, B, C, 1) has moved disk 1 from tower A to tower C, and has obeyed rule (1) because only one disk was moved, and has obeyed rule (2) because there is no disk smaller than disk 1. Luckily, at this point HANOI runs out of instructions and so it terminates. Putting these observations about HANOI(A, B, C, 1), we conclude that \( HYP(1) \) is a true.

**INDUCTIVE STEP:**
The base case was the easy part. Next we want to show that for each and every \( n > 1 \),
\[
HYP(n - 1) \implies HYP(n)
\]
(we read this as \( HYP(n-1) \) implies \( HYP(n) \)). This implication is a little trickier because we use the assumption that \( HYP(n - 1) \) is TRUE on two separate occasions.

Consider the algorithm HANOI with the inputs (A, B, C, n) and \( n > 1 \). The **IF** condition ‘\( n = 1 \)’ is **FALSE**, so the algorithm executes the **ELSE** part. The first statement in the **ELSE** part is HANOI(A, C, B, n-1). By our assumption that \( HYP(n - 1) \) is TRUE, this recursive call to HANOI will correctly move the disks 1, 2, \ldots, \( n - 1 \) from tower A to tower B. (Notice that our definition of \( HYP(n) \) doesn’t really refer to moving disks to tower C, rather it says that that the disks are moved to the tower whose name is the third input.)

The next instruction is MOVE(A,C) which we assumed moved the top disk from tower A to tower C. Now the top disk on tower A was disk \( n \), so disk \( n \) is moved from tower A to tower C. Clearly this obeys rule (1) since only one disk is being moved. But why is rule (2) obeyed? Well, the only disks smaller than disk \( n \) are disks 1, 2, \ldots, \( n - 1 \) and by the previous recursive call, all these disks
were moved onto tower B, so putting disk \( n \) on tower C cannot put it on top of a smaller disk.

The next instruction is the recursive call \( \text{HANOI}(B, A, C, n-1) \), and by \( HYP(n-1) \) this call correctly moves disks 1, 2, \ldots \( n-1 \) from tower B to tower C. Notice that this movement obeys rule (1) because by \( HYP(n-1) \) only one disk at a time is being moved. Further, rule (2) is obeyed because only disks 1, 2, \ldots \( n-1 \) are being moved and by \( HYP(n-1) \) none of the disks from among 1, 2, \ldots \( n-1 \) are ever placed on a larger disk from among 1, 2, \ldots \( n-1 \), and since any disks on towers A and C at the beginning of this recursive call are larger than \( n-1 \), there can be no problem in placing a disk from among 1, 2, \ldots \( n-1 \) on top of the disks that began on towers A and C.

In sum, the action of the recursive call, the move, and the second recursive call result in:
- the disks 1, 2, \ldots \( n-1 \) are moved from tower A to tower B,
- then disk \( n \) is moved to tower C,

and finally
- the disks 1, 2, \ldots \( n-1 \) are moved from tower B to tower C.

At this point the algorithm runs out of instructions and terminates, and so we conclude that if \( HYP(n-1) \) is \textbf{TRUE}, then \( HYP(n) \) is \textbf{TRUE}.

This is the \textbf{END of the Proof}.

We may want to notice that proving that this algorithm terminates (finishes, stops), was easy because of the recursive structure. As long as a recursive algorithm calls itself a finite number of times on smaller inputs, the algorithm must terminate. We should contrast this with some iterative algorithms with loops. For such algorithms it may not be obvious that the loops will in fact finish. So a separate proof of termination may be necessary.

We should also note that we had to check to make sure that rules (1) and (2) were obeyed. We had to make sure that moving disk \( n \) and then moving the \( n-1 \) smaller disks did not cause a violation of these rules.
2.4 HANOI makes the fewest possible moves

In a later section we will calculate the minimum number of moves and use this to argue a minimum “time” for any algorithm which correctly solves the Towers of Hanoi puzzle. Here, we want to argue that the algorithm HANOI does use the fewest possible moves even though we don’t know what that minimum is.

We make use of a “bottleneck” argument. Clearly, to move disk $n$ off tower $A$, all of the disks $1, 2, \ldots, n - 1$ must be off tower $A$. Further, to move disk $n$ from tower $A$ there must be a tower, other than tower $A$, which contains none of the disks $1, 2, \ldots, n - 1$ since we are not allowed to place disk $n$ on top of any of the disks $1, 2, \ldots, n - 1$. In pictures:

\[
\begin{array}{cccc}
1 & \cdots & \cdots & 1 \\
\vdots & & & \vdots \\
n & \cdots & \cdots & n - 1 \\
\hline
A & B & C & \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{OR} & & & \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & \cdots & \cdots & 1 \\
\vdots & & & \vdots \\
n & \cdots & \cdots & n - 1 \\
\hline
A & B & C & \\
\end{array}
\]

But, we notice that either of these situations (if they are a part of a minimal move solution) involve the minimal number of moves for a Towers of Hanoi problem with $n - 1$.

Clearly it takes at least 1 move to move disk $n$ to tower $C$, and just before that move we must have

\[
\begin{array}{cccc}
1 & \cdots & \cdots & 1 \\
\vdots & & & \vdots \\
n & \cdots & \cdots & n - 1 \\
\hline
A & B & C & \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{OR} & & & \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & \cdots & \cdots & 1 \\
\vdots & & & \vdots \\
n - 1 & \cdots & \cdots & n \\
\hline
A & B & C & \\
\end{array}
\]
So after disk $n$ is moved, we still have to solve a Towers of Hanoi problem with $n - 1$ disks.

Putting these possibilities together, we see that the minimal sequence of moves involves the following configurations of the puzzle

\[
\begin{array}{ccc}
  1 & 1 \\
  \cdot & \cdot \\
  \cdot & \cdot \\
  \vdots & \vdots \\
  n & n \\
  A & B & C \\
\end{array}
\]

But this is exactly the sequence of moves followed by HANOI.

Here, we are really doing an inductive proof. Our inductive hypothesis is:

\[
Min(n) = \text{HANOI}(A,B,C,n) \text{ makes the fewest possible moves}
\]
\[
\text{in moving } n \text{ disks from tower } A \text{ to tower } C
\]

Our argument so far is that if $Min(n - 1)$ is true then $Min(n)$ is true. To finish the induction, we need a base case. But it is easy to see that $Min(1)$ is true, because at least one move is needed to move one disk from tower $A$ to tower $C$ and HANOI$(A,B,C,1)$ does (I hope, clearly) use exactly one move.
### 2.4.1 Bottleneck

As we said this is a "bottleneck" argument. Where's the bottleneck? The configurations:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td></td>
<td>n−1</td>
<td></td>
<td></td>
<td>n−1</td>
<td></td>
<td>n</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>AND</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

can be considered as the bottleneck, because any minimal move solution must use these two two configurations in sequence – one after the other.

We could also say that

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td></td>
<td>n−1</td>
<td></td>
<td></td>
<td>n−1</td>
<td></td>
<td>n</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

would constitute a bottleneck since any solution must use at least one of these configurations to move disk \( n \) off of tower \( A \).

In general, we can say that a subproblem \( SP \) is a bottleneck for problem \( P \), if every solution to \( P \) must also solve \( SP \). But, this can get a little dicey. For example, if \( P \) is the problem of SORTING \( n \) items, we could let \( SP \) be the problem of SORTING the smallest \( n−1 \) items, since to SORT \( n \) items we clearly have to SORT the \( n−1 \) smallest items. The dicey part is that the \( n−1 \) smallest items may not be sorted until all \( n \) items are sorted.

### 2.5 Time and space used

We would like to calculate the running time of HANOI, but we don't know how long various operations will take. How long will it take to move a disk? How long will it take to subtract 1 from \( n \)? How long will it take to test if
$n = 1$? How long will it take to issue a procedure call? Because we only wish to calculate time to order we don’t have to answer these questions exactly, but we do have to make a distinction between operations which take a constant amount of time, independent of $n$, and operations whose running time depends on $n$.

One possibility is to assume that each operation takes constant time independent of $n$. AHU [Aho, Hopcroft, and Ullman, 1974] calls this assumption the uniform cost criterion. With this uniform cost assumption and letting $T(n)$ be the running time for $n$ disks, we have the difference equation

$$T(n) = 2T(n - 1) + c$$

because there are 2 calls to the same procedure with $n - 1$ rings and $c$ is the sum of the constant running times for the various operations. Letting $T(1)$ be the running time of the algorithm for 1 disk, we find

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

which can be verified by direct substitution. This gives

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

since

$$\frac{T(1)}{2}2^n \leq T(n) \leq \frac{(T(1) + c)2^n}{2}.$$

Another possibility is to assume that some of the operations have running times which are a function of $n$. But which function of $n$ should we use? Each of the numbers in the algorithm is between 1 and $n$, and the disks can also be represented by numbers between 1 and $n$. Since such numbers can be represented using about $\log n$ bits, it seems reasonable to assume that each operation which manipulates numbers or disks has running time which is a constant times $\log n$. AHU calls this the logarithmic cost criterion and suggests using it when the numbers used by an algorithm do not have fixed bounds. Using the logarithmic cost criterion we have the difference equation

$$T(n) = 2T(n - 1) + c\log n$$

for the running time of the algorithm. This difference equation has the solution

$$T(n) = 2^n \left[ \frac{T(1)}{2} + c \sum_{i=1}^{\log n} \frac{\log i}{2^i} \right]$$

which can be verified by substitution. Since the summation in this solution converges, as one can demonstrate by the ratio test, and assuming that the constants are positive, we have $T(n) = \Theta(2^n)$. Since both cost criteria give the same running time, we conclude:

**Proposition 2.** The algorithm HANOI has running time $\Theta(2^n)$. 

14
Although we have established the running time for a particular algorithm which solves the Towers of Hanoi problem, we have not yet established the time complexity of the problem. We need to establish a lower bound so that every algorithm which solves the problem must have running time greater than or equal to the lower bound. We establish $\Theta(2^n)$ as the lower bound in the proof of the following proposition.

**Proposition 3.** The Towers of Hanoi problem has time complexity $\Theta(2^n)$.

*Proof.* Following the proof of Proposition 1, a straightforward induction shows that the minimal number of moves needed to solve the Towers of Hanoi problem is $2^n - 1$. Since each move requires at least constant time we have established the lower bound on time complexity. An upper bound for the time complexity of the problem comes from Proposition 2. Since the upper bound and lower bound are equal to order, we have established the $\Theta(2^n)$ time complexity of the problem. \qed

Now that we know Hanoi's time complexity we would like to consider its space complexity. First we will establish a lower bound on space which follows from the lower bound on time.

**Proposition 4.** Any algorithm which solves the Towers of Hanoi problem must use at least $n + \text{constant}$ bits of storage.

*Proof.* Since the algorithm must produce $2^n - 1$ moves to solve the problem, the algorithm must be able to distinguish $2^n$ different situations. If the algorithm did not distinguish this many situations, then the algorithm would repeat a situation. But the algorithm would behave the same way the second time it reached the repeated situation as it had the first time. So the algorithm would repeat the situation again and again and again, and never halt.

The number of situations distinguished by an algorithm is equal to the number of storage situations times the number of internal situations within the algorithm. Since the algorithm has a fixed finite size, it can have only a constant number of different internal situations. The number of storage situations (states) is $2$ to the number of storage bits. Thus $c2^{BITS} \geq 2^n$, and so $BITS \geq n - \log c = n + \text{constant}$. \qed

In order to discuss the space complexity of the recursive algorithm let us now consider the data structure used. Two possible data structures are the array and the stack. An array is a set of locations indexed by a set of consecutive integers so that the information stored at a location in the array can be referenced by indicating the integer which indexes the location. For example, the information at location $l$ in the array ARRAY would be referenced by ARRAY[l]. A stack is a linearly ordered set of locations in which information can be inserted or deleted only at the beginning of the stack.

The towers could each be represented by an array with $n$ locations, and each location would need at most $\log n$ bits. So an array data structure with $\Theta(n \log n)$ bits would suffice. Alternately, each tower could be represented by a
stack. Each stack location would need log \( n \) bits, so again this is an \( \Theta(n \log n) \) bit structure. Actually a savings could be made. Since only \( n \) disks have to be represented, the stack structure needs only \( n \) locations versus the \( 3n \) locations used by the array structure. Another possible structure is an array in which the \( i^{th} \) element holds the name of the tower on which the \( i^{th} \) disk is located. This structure uses only \( \Theta(n) \) bits. Yet another possibility is to not represent the towers, but to output the moves in the form FROM \( \ldots \) TO \( \ldots \). Thus we could use no storage for the towers.

The recursive algorithm still requires space for its recursive stack. When a recursive algorithm calls itself, the parameters for this new call will take the places of the previous parameters, so these previous parameters are placed on a stack from which they can be recalled when the new call is completed. Also placed on the stack is the return address, the position in the algorithm at which execution of the old call should be resumed. All of this information, the parameters and the return address, for a single call are referred to as a stack frame. At most \( n \) stack frames will be active at any time and each frame will use a constant number of bits for the names of the towers and \( \log n \) bits for the number of disks. So the recursive algorithm will use \( \Theta(n \log n) \) bits whether or not the towers are actually represented. We summarize these considerations by the following proposition.

**Proposition 5.** The recursive algorithm HANOI correctly solves the Towers of Hanoi problem and uses \( \Theta(2^n) \) time and \( \Theta(n \log n) \) space.

The recursive algorithm uses more than minimal space. We are faced with several possibilities:

1) Minimal space is only a lower bound and is not attainable by any algorithm;

2) Minimal space can only be achieved by an algorithm which uses more than minimal time;

3) Some other algorithm attains both minimal time and minimal space.

By developing a series of iterative algorithms, we will arrive at an algorithm which uses both minimal time and minimal space.

### 2.6 Improved Algorithms

As a first step in obtaining a better algorithm, we will consider an iterative algorithm which simulates the recursive algorithm for \( n \geq 2 \). In this algorithm, RECURSIVE SIM, we have chosen to explicitly keep track of the stack counter because this will aid us in finding an algorithm using even less space.

**PROCEDURE RECURSIVE SIM (A,B,C,n)**

\[
\begin{align*}
I &:= 1 \\
\end{align*}
\]
NUM[1] := n-1 ; PAR[1] := 1 ; PAR[0] := 1

WHILE I ≥ 1 DO
    IF NUM[I] > 1
        THEN L1[I+1] := L1[I]
            L2[I+1] := L3[I]
            L3[I+1] := L2[I]
            NUM[I+1] := NUM[I] - 1
            PAR[I+1] := 1
            I := I+1
        ELSE MOVE FROM L1[I] TO L3[I]
            WHILE PAR[I] = 2 DO
                I := I-1
            IF I ≥ 1 THEN MOVE FROM L1[I] TO L2[I]
                PAR[I] := 2
                TEMP := L1[I]
                L1[I] := L3[I]
                L3[I] := L2[I]
                L2[I] := TEMP

The names of the towers are stored in the three arrays L1, L2, L3; the number of disks in a recursive call is stored in NUM; and the value of PAR indicates whether a call is the first or second of a pair of recursive calls.

RECURSIVE SIM sets up the parameters for the call HANOI(A,C,B,n-1). When the last move for this call is made, the arrays will contain the parameters for calls with 1 through n – 2 disks, where each of these calls will have PAR=2. The arrays will still contain the parameters for the (A,C,B,n-1) call with PAR=1. The inner WHILE loop will pop each of the calls with PAR=2, leaving the array counter pointing at the (A,C,B,n-1) call. Since I will be 1 at this point, the IF condition is satisfied and the MOVE FROM L1[I] TO L2[I] accomplishes the MOVE FROM A TO C of the recursive algorithm HANOI. The following assignment statements set up the call (B,A,C,n-1) with PAR=2. So when the moves for this call are completed, all of the calls in the array will have PAR=2, and the inner WHILE loop will pop all of these calls setting I to 0. Then the IF condition will be false, so no operations are carried out, and the outer WHILE condition will be false so the algorithm will terminate.

**Proposition 6.** The RECURSIVE SIM algorithm correctly solves the Towers of Hanoi problem, and uses \( \Theta(2^n) \) time and \( \Theta(n \log n) \) space.

**Proof.** Correctness follows since this algorithm simulates the recursive algorithm which we have proved correct. The major space usage is in the arrays. Since each time I is incremented the corresponding NUM[I] is decremented and since NUM[I] never falls below 1, there are at most \( n - 1 \) locations ever used in an array. The four arrays L1, L2, L3, and PAR use only a constant amount of
space for each element, but NUM must store a number as large as \( n - 1 \) so it uses \( \Theta(\log n) \) bits for an element. Thus the arrays use \( \Theta(n \log n) \) bits.

Now we have to argue about time usage. Most of the operations deal with constant-sized operands so these operations will take constant time. The exceptional operations are incrementing, decrementing, assigning, and comparing numbers which may have \( \Theta(\log n) \) bits. A difference equation for the time is

\[
T(n) = 2T(n-1) + C \log n,
\]

where \( T(n) \) is the time to solve a problem with \( n \) disks, and \( C \log n \) is the time for manipulating the numbers with \( \Theta(\log n) \) bits. As in the proof of Proposition 2, we have \( T(n) = \Theta(2^n) \).

Notice that this algorithm does not improve on the recursive algorithm, but study of this form can lead to a saving of space. Storing the array NUM causes the use of \( \Theta(n \log n) \) space. If we did not have to store NUM, the algorithm would use only \( \Theta(n) \) space. Do we need to save NUM? NUM is used as a control variable so it seems necessary. But if we look at NUM[I] + 1 we get \( n \). When NUM[I+1] is set, it is set equal to NUM[I] - 1, but then NUM[I + 1] + I + 1 = NUM[I] + I = n. Thus the information we need about NUM is stored in I and n. So if we replace the test on NUM[I] = 1 with a test on I = n-1, we can dispense with storing NUM and improve the space complexity from \( \Theta(n \log n) \) to \( \Theta(n) \). This replacement does not increase the time complexity of any step in the algorithm, so the time complexity remains \( \Theta(2^n) \).

Our new procedure is

**PROCEDURE NEW SIM (A,B,C,n)**

\[
\begin{align*}
I &:= 1 \\
L1[I] &:= A \ ; \ L2[I] := C \ ; \ L3[I] := B \\
PAR[1] &:= 1 \ ; \ PAR[0] := 1 \\
\text{WHILE } I \geq 1 \ \text{DO} \\
&\quad \text{IF } I \neq n-1 \\
&\quad \quad \text{THEN } L1[I+1] := L1[I] \\
&\quad \quad \quad L2[I+1] := L3[I] \\
&\quad \quad \quad L3[I+1] := L2[I] \\
&\quad \quad \quad PAR[I+1] := 1 \\
&\quad \quad \quad I := I+1 \\
&\quad \quad \text{ELSE } \text{MOVE FROM } L1[I] \text{ TO } L3[I] \\
&\quad \quad \quad \text{WHILE } PAR[I] = 2 \ \text{DO} \\
&\quad \quad \quad \quad I := I-1 \\
&\quad \quad \quad \text{IF } I \geq 1 \\
&\quad \quad \quad \quad \text{THEN } \text{MOVE FROM } L1[I] \text{ TO } L2[I] \\
&\quad \quad \quad \quad \quad PAR[I] := 2 \\
&\quad \quad \quad \quad \quad TEMP := L1[I] \\
&\quad \quad \quad \quad \quad L1[I] := L3[I] \\
&\quad \quad \quad \quad \quad L3[I] := L2[I] \\
&\quad \quad \quad \quad \quad L2[I] := TEMP
\end{align*}
\]

18
From the above observation we have:

**Proposition 7.** *NEW SIM correctly solves the Towers of Hanoi problem and uses $\Theta(2^n)$ time and $\Theta(n)$ space.*

Although we have reached $\Theta(n)$ space we would like to decrease the space even further hopefully to $n + \text{constant}$ bits. If we look at the array PAR, we find that the algorithm scans PAR to find the first element not equal to 2, replaces that element by 2 and then replaces all the previous 2’s by 1’s. This is analogous to the familiar operation of adding 1 to a binary number, in which we find the first 0, replace it by a 1, and replace all the previous 1’s by 0’s. So it seems that we can replace the array PAR by a simple counter. The number of bits in the counter will, of course, depend on $n$.

So far this has not resulted in any saving of space. Will there be enough information in the counter to determine from which tower we should move a disk? The affirmative answer will enable us to achieve a minimal space algorithm. To motivate the design of our minimal space algorithm, we will examine the sequence of 31 moves needed to solve the problem with 5 disks. This sequence is shown in Table 1.
<table>
<thead>
<tr>
<th>DECIMAL</th>
<th>TOWER0</th>
<th>TOWER1</th>
<th>TOWER2</th>
<th>COUNT</th>
<th>COUNT</th>
<th>DISK</th>
<th>FROM</th>
<th>TO</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>00000</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2345</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>00001</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>345</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>00010</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>345</td>
<td>12</td>
<td>-</td>
<td>3</td>
<td>00011</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>12</td>
<td>3</td>
<td>4</td>
<td>00100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>145</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>00101</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>145</td>
<td>-</td>
<td>23</td>
<td>6</td>
<td>00110</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>-</td>
<td>123</td>
<td>7</td>
<td>00111</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>123</td>
<td>8</td>
<td>01000</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>23</td>
<td>9</td>
<td>01001</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>14</td>
<td>3</td>
<td>10</td>
<td>01010</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>4</td>
<td>3</td>
<td>11</td>
<td>01011</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>34</td>
<td>-</td>
<td>12</td>
<td>01100</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>34</td>
<td>1</td>
<td>13</td>
<td>01101</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>234</td>
<td>1</td>
<td>14</td>
<td>01110</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1234</td>
<td>-</td>
<td>15</td>
<td>01111</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>1234</td>
<td>5</td>
<td>16</td>
<td>10000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>234</td>
<td>5</td>
<td>17</td>
<td>10001</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>34</td>
<td>25</td>
<td>18</td>
<td>10010</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>34</td>
<td>125</td>
<td>19</td>
<td>10011</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>125</td>
<td>20</td>
<td>10100</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>25</td>
<td>21</td>
<td>10101</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>14</td>
<td>5</td>
<td>22</td>
<td>10110</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>4</td>
<td>5</td>
<td>23</td>
<td>10111</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>-</td>
<td>45</td>
<td>24</td>
<td>11000</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>-</td>
<td>145</td>
<td>25</td>
<td>11001</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>145</td>
<td>26</td>
<td>11010</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>45</td>
<td>27</td>
<td>11011</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>12</td>
<td>345</td>
<td>28</td>
<td>11100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>345</td>
<td>29</td>
<td>11101</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>2345</td>
<td>30</td>
<td>11110</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>12345</td>
<td>31</td>
<td>11111</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Towers of Hanoi Solution for 5 disks.
Every other move in the solution involves moving disk 1. So if we know which
tower contains disk 1 we would know from which tower to move, in alternate
moves, but we might not know which tower to move to. When we consider
the three towers to be arranged in a circle, we see from Table 1 that disk 1 always
moves in a counterclockwise direction when we have an odd number of disks.
Similarly disk 1 always moves in a clockwise direction when we have an even
number of disks. Thus by keeping track of the tower which contains disk 1 and
whether n is odd or even, we would know how to make every other move.

For the moves which do not involve disk 1, we know that the move involves
the two towers which do not contain disk 1. Looking again at Table 1, we
see that the odd numbered disks always move in the same direction as disk 1
and that the even numbered disks always move in the opposite direction. So
knowing the towers involved and whether the disk to be moved is odd or even
would allow us to decide which way to move.

Can we determine from a counter whether the disk being moved is odd or
even? If we look at the COUNT column of Table 1 we see that the position
of the rightmost 0 tells us the number of the disk to be moved. Thus a single
counter with n bits is sufficient to solve the Towers of Hanoi problem. We use
these facts to construct the algorithm which follows.

**PROCEDURE TOWERS**(n)

1. \( T := 0 \) (*TOWER NUMBER COMPUTED MODULO 3*)
2. \( \text{COUNT} := 0 \) (*COUNT HAS n BITS*)
3. \( P := \{1 \text{ if } n \text{ is even}\} \) \{-1 if n is odd\}

**WHILE TRUE DO**

1. **MOVE DISK 1 FROM T TO T+P**
2. \( T := T+P \)
3. \( \text{COUNT} := \text{COUNT} + 1 \)
4. **IF COUNT = ALL 1’s THEN RETURN**
5. **IF RIGHTMOST 0 IN COUNT IS IN EVEN POSITION**
6. **THEN MOVE DISK FROM T-P TO T+P**
7. **ELSE MOVE DISK FROM T+P TO T-P**
8. \( \text{COUNT} := \text{COUNT} + 1 \)
**ENDWHILE**

The storage used for COUNT has n bits, and we have called the rightmost
bit position 1. The positions from right to left are then odd, even, odd, even....

Remarks: We can still improve this algorithm by removing the first COUNT
:= COUNT + 1 statement and deleting the rightmost bit of COUNT. This would
also require changing the numbering of the bits in COUNT so that the rightmost
bit is bit 0. An algorithm similar to our TOWERS has been published by T. R.
Walsh [1982].

We have to show that TOWERS correctly solves the Towers of Hanoi
problem. We do this by proving that a certain sequence of moves has been ac-
complished when COUNT contains a number of the form \( 2^k - 1 \), so that when
\(k = n\), the sequence of moves for HANOI(A,B,C,\(n\)) has been completed and the procedure will terminate since COUNT contains all 1's.

**Proposition 8.** When \(\text{COUNT} = 2^k - 1\), that is \(\text{COUNT} = 00...01\ldots 1\) with \(k\) 1's, then: if \(k \neq n(\text{MOD}2)\) the correct moves for HANOI(A,C,B,k) have been completed and T contains 1 (which represents B), if \(k = n(\text{MOD}2)\) the correct moves for HANOI(A,B,C,k) have been completed and T contains 2 (which represents C).

**Proof.** If \(k = 1\) the single move T to T + P has been completed, which is A to C if \(n\) is odd, and is A to B if \(n\) is even, and T contains T + P which is 2 if \(n\) is odd and is 1 if \(n\) is even. This agrees with our claim.

Notice that COUNT can only take on the value \(2^k - 1\) immediately before the IF ... RETURN statement. Assume that the moves for either HANOI(A,B,C,k) or HANOI(A,C,B,k) have been completed. If \(k \neq n(\text{MOD}2)\), the next move will be A to C since by assumption T now contains 1; if \(k\) is odd, the move is T - P to T + P which is 1 - (1) to 1 + (1) which represents A to C, and if \(k\) is even, the move is T + P to T - P which is 1 + (-1) to 1 - (-1) which represents A to C. If \(k = n(\text{MOD}2)\), the next move will be A to B since by assumption T now contains 2; if \(k\) is odd, the move is T - P to T + P which is 2 - (-1) to 2 + (+1) which represents A to B, and if \(k\) is even, the move is T + P to T - P which is 2 + 1 to 2 - 1 which represents A to B.

Next COUNT will be incremented to 00...010...0, i.e., \(k\) trailing 0's. When \(\text{COUNT} = 2^{k+1} - 1\), the algorithm will have repeated the same sequence of moves as before since it only "sees" the rightmost information in COUNT, with the difference that T will have started with a different value. The different starting value of T will result in a cyclic permutation of the labels. If \(k \neq n(\text{MOD}2)\), then the completed moves will be

\[
\text{HANOI (A,C,B,k)}
\]

A to C

\[
\text{HANOI (B,A,C,k)}
\]

giving \(\text{HANOI (A,B,C,k+1)}\) with \(k + 1 = n(\text{MOD}2)\), and T will contain 2 (i.e., 1 + 1). If \(k = n(\text{MOD}2)\), then the completed moves will be

\[
\text{HANOI (A,B,C,k)}
\]

A to B

\[
\text{HANOI (C,A,B,k)}
\]

giving \(\text{HANOI (A,C,B,k+1)}\) with \(k + 1 \neq n(\text{MOD}2)\), and T will contain 1 (i.e., 2 + 2).

**Proposition 9.** The algorithm TOWERS uses \(\Theta(2^n)\) time and \(n + \text{constant bits of space}\).
Proof. For space usage, there are \( n \) bits in COUNT, and a constant number of bits are used for \( T \) and \( P \). For time, the initialization takes \( \Theta(n) \) and the WHILE loop is iterated \( 2^n - 1 \) times. If each iteration took a constant amount of time we would have \( \Theta(2^n) \), but the test and increment instruction on COUNT could take time \( \Theta(n) \) giving \( \Theta(n2^n) \). So we have to show that only \( \Theta(2^n) \) time is used. If the value in COUNT is even then incrementing and testing will only require looking at one bit. If the value in COUNT is odd and \( (\text{COUNT} - 1)/2 \) is even, then the algorithm only looks at 2 bits. In fact, the algorithm will look at \( k \) bits in COUNT in \( 2^{n-k} \) cases. Thus the time used will be

\[
\Theta\left(\sum_{k=1}^{n} k2^{n-k}\right) = \Theta(2^n)
\]

since \( \sum_{k=1}^{\infty} k2^{-k} \) converges.

We summarize these results in the following theorem.

**Theorem 1.** Any algorithm which solves the Towers of Hanoi problem for \( n \) disks must use at least \( \Theta(2^n) \) time and \( n + \text{constant} \) bits of storage. The algorithm TOWERS solves the problem and simultaneously uses minimum time and minimum space.

**Exercises**

**Ex 2.1.** For the following algorithm for the Towers of Hanoi problem, prove that the algorithm is correct and compute its time and space.

```plaintext
PROCEDURE HANOI ITERATIVE (A,B,C,n)
    IF n mod 2 = 0 THEN MOVE[1]:= A TO B
    ELSE MOVE[1]:= A TO C
    K:= 1
    WHILE n > 1 DO
        n:= n-1; K:= 2*K
        IF n mod 2 = 0 THEN MOVE[K]:= A TO B
            L1:= C; L2:= A; L3:=B
        ELSE MOVE[K]:= A TO C
            L1:= B; L2:= C; L3:= A
        FOR I:= 1 TO K-1 DO
            CASE MOVE[I] OF
```
A TO B : MOVE[X+1] := L1 TO L2
A TO C : MOVE[X+1] := L1 TO L3
B TO A : MOVE[X+1] := L2 TO L1
B TO C : MOVE[X+1] := L2 TO L3
C TO A : MOVE[X+1] := L3 TO L1
C TO B : MOVE[X+1] := L3 TO L2

Hints: For correctness you may want to introduce a new variable and prove a statement which says that on each iteration of the WHILE loop the new variable increases (or if you want, decreases), and that at the end of each iteration a Hanoi problem whose size depends on the new variable has been solved. You will need to give the tower names for the problem which has been solved. You will also need to specify the value of the new variable.

For space, you should know that the algorithm is storing each move in the array MOVE.

For time, you may want to consider both the uniform and the logarithmic cost measures.

Ex 2.2. (Buneman and Levy [1980]) The following is an algorithm for the Towers of Hanoi problem. Prove its correctness and compute its time and space.

MOVE SMALLEST DISK ONE TOWER CLOCKWISE

WHILE A DISK (OTHER THAN THE SMALLEST) CAN BE MOVED DO
    MOVE THAT DISK
    MOVE THE SMALLEST DISK ONE TOWER CLOCKWISE
ENDWHILE

Hints: For correctness, you should be careful since this algorithm only solves the original Towers of Hanoi problem when the number of disks is even. You will probably want to introduce a new variable and prove a statement about the configuration of the disks when the number of moves completed is a specific function of your new variable.

For time and space, the above algorithm is incomplete since it doesn’t specify the data structure used to determine if a disk can be moved. You might consider representing each tower by a stack of integers with the integers representing the disks on the tower. Alternately you might consider representing the information by an array DISK, so that DISK[t] contains the name of the tower which contains the \( t^{th} \) largest disk. You may also find it useful to show that the \( I^{th} \) disk is moved \( 2^{n-I} \) times.
Chapter 3

SOME INDUCTIONS

3.1 What Is Induction?

The most important method for proving things in computer science is induction. This method is sometimes called the method of mathematical induction to distinguish it from the philosophical or empirical method of induction. In empirical induction, we note that a certain proposition is true in individual instances and we generalize or induce or induct that the proposition is true for all instances. For example, a philosopher notices that the sun has risen for each and every morning of the philosopher’s life, and so the philosopher inducts that the proposition “the sun rises every morning” is always true. After framing this proposition, the philosopher predicts that the sun will rise the next morning, and when the sun does in fact rise, the philosopher is happy to say “I told you so!” The philosopher spends his mornings (after the sun has come up) bird watching. He has noted that he often sees swans and that every swan he has ever seen has been a white swan. He, therefore, makes the statement: “All swans are white.” As in the sun example, he predicts that the next swan he sees will be white, and he is gratified when the next swan he sees is indeed white. Of course, the philosopher is willing to admit that if he ever sees a black swan, then his inductively verified proposition will be shown to be false. But, for some reason, the philosopher still firmly believes that the sun will rise tomorrow. We will now take leave of the philosopher, and leave to your philosophy professor the problem of explaining how the two empirically verified propositions: the sun rises every morning; and, all swans are white; differ from a philosophical perspective.

Mathematical induction differs from philosophical induction in that mathematical induction is about a well-defined mathematical object, the set of natural numbers. If we can prove something by mathematical induction, there is no possibility that the proved statement might be false tomorrow or that it could be proved false by an empirical observation.

Mathematical induction deals with propositions. A proposition is a state-
ment that is either true or false. To apply mathematical induction, we need a propositional function, \( P(n) \) which takes as input a natural number \( n \) and gives as output one of the two values, true or false. Mathematical induction says that if:

1. \( P(0) \) is true,
2. and, for all \( n \) such that \( n \geq 0 \), \( P(n) \) implies \( P(n + 1) \)

then, \( P(n) \) is true for all \( n \geq 0 \). This is not an empirical observation. It is, in fact, part of the definition of natural number. In the following sections, we will give some simple examples of mathematical induction.

### 3.2 Inductions for easy Sums

A sum \( \sum_{i=0}^{n} f(n) \) may be equivalent to a closed form expression \( g(n) \). For example, \( \sum_{i=0}^{n} 1 = n + 1 \). To prove by induction, one must first decide what one is trying to prove. Here, one is given a sum and an expression, and the claim is that the sum and the expression represent the same function. That is, if I put in a value say 10 for \( n \) and evaluate the sum, I will get some number, say \( S \), and when I put 10 into the expression and evaluate it, I will get a number \( g(10) \). The claim is that \( S = g(10) \), that is, I get the same value whether I use the sum or the expression. So, to frame the inductive hypothesis we use \( P(n) = "\sum_{i=0}^{n} f(n) = g(n)". Notice that the right side, delimited by the quotation marks, is is a relational expression which claims that what is on the left side of the \( = \) (the relational operator) is equal to what is on the right side of the \( = \). To confuse things as much as possible, the two \( = \) signs serve different roles. While the \( = \) inside the quotes is a relational operator, the \( = \) after \( P(n) \) indicates a definition. So, while the \( = \) inside could have the value true or false depending on whether or not \( \sum_{i=0}^{n} f(n) \) does equal \( g(n) \), the \( = \) outside just says that \( P(n) \) is defined by the whole quoted expression. So \( P(n) \) can be true or false depending on the truth or falsity of the the quoted expression. Of course, \( P(n) \) is a function of the natural number variable \( n \), so, for example, \( P(0) \) could be true, \( P(1) \) could be false, \( P(2) \) could be false, and \( P(3) \) could be true. What we are trying to prove is that \( P(n) \) is always true, that is, for whatever value of \( n \) we choose, when we evaluate \( P(n) \) we will get true.

At this point we may want to throw up our hands because we are faced with an infinite number of tasks, that is, showing that \( P(0) \) is true, \( P(1) \) is true, \( P(2) \) is true, and so forth. The principle of mathematical induction allows us to break this infinite amount of work into a few pieces and if we can finitely show each of these pieces, we will have almost magically accomplished an infinite amount of work.

The principle of mathematical induction states:

1. if \( P(0) \) is true,
2. and, if for all \( n \geq 0 \), \( P(n) \) implies \( P(n + 1) \)
3. then, $P(n)$ is true for all $n \geq 0$.

Clearly, this form isolates an infinite amount of work in step (2). We have to figure out how to do this infinite work with a finite (and we hope small) amount of effort. The trick is that for some $P(n)$’s we can treat $n$ as an unknown variable, and use a finite amount of work to show $P(n+1)$ from $P(n)$. If we don’t have to say what value $n$ has, we will really be accomplishing an infinite amount of work with finite effort.

Let us return now to our sum examples, and let $P(n) = \sum_{i=0}^{n} 1 = n + 1$.

For the base step, we want to show that $P(0)$ is true. Of course, $P(0) = \sum_{i=0}^{0} 1 = 0 + 1$” after we have substituted 0 for $n$. Now $\sum_{i=0}^{0} 1$ evaluates to 1, and 0 + 1 also evaluates to 1. Hence, $P(0)$ has the same truth value as the expression 1 = 1, and since we know that 1 equals 1, we conclude that $P(0)$ is true.

To handle the infinite amount of work for step (2), which is called the inductive step, we write out $P(n+1)$ which is $P(n+1) = \sum_{i=0}^{n+1} 1 = (n+1) + 1$.

Now we assume that $P(n)$ is true and try to use $P(n)$ to show $P(n+1)$. $P(n)$ states that

$$\sum_{i=0}^{n} 1 = n + 1.$$ 

With a little luck (or insight) we will try to look for something in $P(n+1)$ that will simplify if we can use this relation. Consider $\sum_{i=0}^{n+1} 1$, we can break this sum apart because

$$\sum_{i=0}^{n+1} 1 = (\sum_{i=0}^{n} 1) + 1.$$ 

Now we can use $P(n)$ to replace the term in the parentheses, and so

$$\sum_{i=0}^{n+1} 1 = (n + 1) + 1.$$ 

But, miracle of miracles, this is just the relation that we had to show to show that $P(n+1)$ is true. Notice that we were able to treat $n$ as an unevaluated variable (except that we really needed $n \geq 0$ to expand the summation), so we have shown that

$$\forall n \geq 0 \ P(n) \Rightarrow P(n+1),$$

and so we can conclude by the principle of mathematical induction that $P(n)$ is true for all $n \geq 0$. Said another way, we conclude that

$$\sum_{i=0}^{n} 1 = n + 1.$$ 

Basically all summation examples work like this simple example. We start with the inductive hypothesis $P(n) = \sum_{i=0}^{n} f(i) = g(n)$. We then show the base case, $P(0)$, by demonstrating that $f(0) = g(0)$, which is what we want
because $\sum_{i=0}^{n} f(i) = f(0)$. Then, we proceed to show that $P(n) \Rightarrow P(n+1)$ by writing
\[
\sum_{i=0}^{n+1} f(i) = \left[ \sum_{i=0}^{n} f(i) \right] + f(n+1)
\]
and using $P(n)$ to replace $\sum_{i=0}^{n} f(i)$ by $g(n)$, which gives $g(n) + f(n+1)$. We are now left with showing that $g(n) + f(n+1) = g(n+1)$. This step is usually a matter of algebraic manipulation. For example, to show that $\sum_{i=0}^{n} i = n(n+1)/2$, we would use $g(n) = n(n+1)/2$, $g(n+1) = (n+1)(n+2)/2$, and $f(n) = n + 1$. Then a simple algebraic manipulation shows that
\[
g(n) + f(n+1) = \frac{n(n+1)}{2} + (n+1)
\]
\[
= \frac{(n+1)}{2}(n+2)
\]
\[
= g(n+1).
\]

**Ex 3.1.** Use induction to show that
\[
\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}.
\]

**Ex 3.2.** From the previous examples, conjecture a formula $g(n)$ for
\[
\sum_{i=0}^{n} i^3 = g(n).
\]
Use mathematical induction to show that your formula is correct.

**Ex 3.3.** Use induction to show
\[
\sum_{i=0}^{n} 2^i = 2^{n+1} - 1.
\]

**Ex 3.4.** Use induction to show
\[
\sum_{i=0}^{n} 3^i = (3^{n+1} - 1)/2.
\]

**Ex 3.5.** Use the last two exercises to conjecture a formula for $\sum_{i=0}^{n} 4^i$. Verify your formula using induction.

**Ex 3.6.** Use these examples to conjecture a formula for $\sum_{i=0}^{n} a^i$, where $a$ is any number except 1. Verify your formula by induction.
3.3 Euclid’s Algorithm

One of the oldest algorithms is Euclid’s method for computing the greatest common divisor of two non-negative integers. The greatest common divisor of $a$ and $b$ is the largest integer that divides both $a$ and $b$. [When we say that $d$ divides $a$, we mean that the quotient $a/d$ is an integer and that the remainder is 0. Said another way, $d$ divides $a$ when $a$ is an integer multiple of $d$, that is, there is an integer $n$ so that $a = nd$.] This definition does not work in the very special case when $a = b = 0$ because every integer divides 0. For this special case we define the greatest common divisor of 0 and 0 to be 0. This should not cause any difficulty because 0 is not a divisor of any other number.

Euclid’s algorithm can be written in recursive form as:

$$GCD(a, b) = \begin{cases} 
a & \text{if } b = 0 \\
GCD(b, a \mod b) & \text{if } b \neq 0.
\end{cases}$$

We need to explain the operation $a \mod b$. We assume that $a$ is an integer and that $b$ is a positive integer. We do not allow $b = 0$. We want $a \mod b$ to give us the remainder when $a$ is divided by $b$. If $a < b$, then $a \mod b = a$. If $a$ can be written as $a = nb + r$, where $0 \leq r \leq b - 1$ and $n \geq 0$, then $a \mod b = r$. The traditional division algorithm will produce the desired $r$. Notice that if $a$ is a multiple of $b$, then $r = 0$, i.e., $a \mod b = 0$.

We now use mathematical induction to show that Euclid’s algorithm does compute the greatest common divisor. For notational purposes, let $g$ be the greatest common divisor of $a$ and $b$. We want to show that $GCD(a, b)$ produces $g$ as output. As a first step in the inductive proof, we need to choose an inductive variable and an inductive hypothesis. Recall that a proposition is a statement that is either true or false. For example, $2 + 2 = 4$ is a proposition which is true, while $2 + 2 = 5$ is a proposition which is false. There are, of course, English sentences like “Do your homework.” that are not propositions because one does not assign a true or false value to them sentences. (The confusion between propositions and non-propositions forms the basis for Abbot and Costello’s famous “Who’s on first?” routine.) Since Euclid’s algorithm has two variables, $a$ and $b$, it may not be obvious which one should be chosen as the inductive variable, but the form of the algorithm having an output for $b = 0$ and a recursive call for $b \neq 0$ suggests that $b$ should be chosen as the inductive variable. We then frame the inductive hypothesis as a function of $b$. The inductive hypothesis will take a non-negative integer as input and produce an output which is in {true, false}.

For Euclid’s algorithm we can use the inductive hypothesis:

$$E(b) = \text{‘GCD}(a, b) \text{ calculates the greatest common divisor of } a \text{ and } b \text{’}$$

$$= \text{‘GCD}(a, b) = g \text{’}.$$  

(I’ve put in quote marks so that it’s easier to see the proposition.) Notice that ‘GCD$(a, b) = g$’ has a true or false value. We want to show that this proposition has the value true for all natural numbers $a$ and $b$. Now that we
have the inductive hypothesis $E(b)$, we want to show that $E(b)$ is true for small values of $b$. This is called proving the base case. In simple situations like this one, we need only consider one small value of $b$, in this case $b = 0$. So our base case is $E(0)$, and we want to show that $E(0)$ is true. If $b = 0$, what is the largest integer which divides both $a$ and $b$? Since $b = 0$, all integers divide $b$. If $a > 0$, the largest integer which divides $a$ is $a$. So the largest integer which divides both $a$ and $b$ is $a$. In the special case when $a = 0$, every integer divides $a$, but we have defined the greatest common divisor to be $0$ for this special case. Now, when $b = 0$, Euclid’s algorithm outputs $a$. In the special case, $a = 0$, this agrees with our definition of greatest common divisor. Also, when $a > 0$, $a$ is the greatest common divisor of $a$ and $b$, and Euclid’s algorithm correctly outputs $a$. Hence, $E(0)$ is true, and we have proved the base case.

Now we proceed to the inductive step. We want to show that for all $b \geq 0$, $E(b)$ is true. We do so by showing that $E(0)$ and $E(1)$ and $E(2)$ and ... and $E(b - 1)$ all being true allows us to conclude that $E(b)$ is true. We assume that $b > 0$ and look at what the algorithm does. In this case, $GCD(a, b)$ calls $GCD(b, a \mod b)$. What do we know about $a \mod b$? We know that $a \mod b$ is an integer and that $0 \leq a \mod b < b$. So $E(a \mod b)$ is somewhere among $E(0)$, $E(1)$, $E(2)$, ... , $E(b - 1)$ and we have assumed that each of these propositions is true. Hence, we may assume that $E(a \mod b)$ is true, that is, regardless of what the first argument is, $GCD$(first argument, $a \mod b$) produces the greatest common divisor of first argument and $a \mod b$. We are left with showing that the greatest common divisor of $b$ and $a \mod b$ is, in fact, the same as the greatest common divisor of $a$ and $b$. We can write $a$ as

$$a = nb + a \mod b$$

Now, if $d$ divides the left side of this equation, then $d$ also divides the right side of this equation, and vice-versa. Further, if $d$ divides $x + y$ and $d$ divides $x$, then $d$ must also divide $y$. Hence, any divisor of $a$ and $b$ must also be a divisor of $b$ and $a \mod b$. Conversely, any divisor of $b$ and $a \mod b$ must also be a divisor of $a$ and $b$. So, the set of divisors of $a$ and $b$ is the same as the set of divisors of $b$ and $a \mod b$, and the largest element in this set is both the greatest common divisor of $a$ and $b$ and is also the greatest common divisor of $b$ and $a \mod b$. Hence, we have shown that if $E(0)$, $E(1)$, $E(2)$, ... , $E(b - 1)$ are all true then $E(b)$ must also be true.

Now, since we have shown that $E(0)$ is true, and that $E(0)$, $E(1)$, $E(2)$, ... , $E(b - 1)$ implies $E(b)$, we can conclude by the principle of mathematical induction that $E(b)$ is true for all $b \geq 0$, i.e. for all non-negative integers $b$.

**Ex 3.7.** Show that the least common multiple of $a$ and $b$ can be computed easily, if you know the greatest common divisor of $a$ and $b$.

**Ex 3.8.** What’s wrong with the following induction?
To show that all horses are the same color.
BASE: $P(0)$ is true because all the horses in a set with no horses are the same color as all the other horses in that set.
INDUCTIVE STEP: $P(n) \implies P(n + 1)$.
Consider a set of $n + 1$ horses. Remove one horse leaving $n$ horses in the set. By $P(n)$, all these $n$ horses have the same color, say brown. Next remove another horse and put the $n + 1$st back in the set. Again by $P(n)$ these $n$ horses have the same color, say brown. So all the $n + 1$ horses have the same color (brown) and $P(n + 1)$ is true.

**Ex 3.9.** A hypercube of dimension $n$ has $2^n$ vertices where each vertex is labeled by one of the $n$ bit binary vectors. Two vertices have an edge between them exactly when their vectors differ in exactly one bit position.
Use induction to show that for $n \geq 2$ the $n$ dimensional hypercube has a Hamiltonian circuit.
Chapter 4

DIVIDE-AND-CONQUER

4.1 What is Divide-and-Conquer?

The algorithm design strategy which breaks a given problem into several smaller problems of the same type is usually called the divide-and-conquer strategy. In the previous section, the recursive algorithm for the Towers of Hanoi problem is an example of a divide-and-conquer algorithm. Given a problem with \( n \) disks, this algorithm converts it into two problems with \( n - 1 \) disks. Each of the subproblems is successively broken into subproblems until problems which can be solved immediately are reached. The Towers of Hanoi algorithm continues forming problems with fewer disks until it reaches problems with 1 disk which can be solved immediately. After the subproblems are solved the divide-and-conquer algorithm then combines the solutions of the subproblems to give a solution to the original problem. In the Towers of Hanoi example, there is no explicit combining, because the necessary combination is simply to solve one subproblem after the other subproblem has been solved. This combination is handled by the ordering of the statements in the algorithm.

The recursive structure of a divide-and-conquer algorithm leads directly to an inductive proof of correctness, and also gives directly a difference equation for the running time of the algorithm.

Consider designing by divide-and-conquer an algorithm to sort the elements of an \( n \)-element array. One way to do this is to find the largest element in the array, interchange it with the last element of the array, and then sort the remaining \( n - 1 \) element array. This algorithm could be written as

\[
\text{PROCEDURE \texttt{SORT}(n)} \\
\text{IF \hspace{1em} \texttt{n > 1} \hspace{1em} THEN \texttt{LARGEST}(n)} \\
\hspace{2em} \texttt{SORT(n-1)},
\]

where \texttt{LARGEST} is an algorithm which handles the largest element. If \texttt{LARGEST} works correctly then it is easy to prove that \texttt{SORT} works correctly. Similarly if we know how many comparisons \texttt{LARGEST} used, then we could compute the
number of comparisons used by SORT from the formula

\[ S(n) = S(n - 1) + L(n), \]

where \( S(n) \) is the number of comparisons used by \( \text{SORT}(n) \), \( S(n-1) \) is the number of comparisons used by \( \text{SORT}(n-1) \), and \( L(n) \) is the number of comparisons used by \( \text{LARGEST}(n) \). It is easy to design \( \text{LARGEST}(n) \) so that it uses exactly \( n - 1 \) comparisons. This gives the difference equation \( S(n) = S(n - 1) + n - 1 \). When there is only 1 element in the array, \( \text{SORT} \) does nothing. This gives the initial condition \( S(1) = 0 \). It is easy to check that \( S(n) = n(n - 1)/2 \) satisfies both the difference equation and the initial condition.

This recursive sorting algorithm can be easily converted to an iterative algorithm because the recursive algorithm is tail-recursive, that is, the only time the algorithm calls itself is at the end of the algorithm. The corresponding iterative algorithm is

\[
\text{FOR } \text{I} := \text{n} \text{ DOWNTO } 2 \text{ DO} \\
\text{LARGEST(I)}
\]

It is still easy to write an inductive proof of the correctness of this algorithm. The number of comparisons used by this iterative algorithm can be computed by

\[
\sum_{I=n}^{2} L(I) = \sum_{I=n}^{2} (I - 1) = n(n - 1)/2.
\]

Both the recursive and iterative sorting algorithms use the same number of comparisons. They also both use space to store the original array. The recursive algorithm has the disadvantage that it uses a stack to keep track of the recursion. This stack requires some space. Further, the recursive algorithm spends some time in manipulating this stack. So in this case, the iterative algorithm would be preferred to the recursive algorithm.

In the above example, we have broken a problem of size \( n \) into a single problem of size \( n - 1 \). Instead we could try to break the problem of size \( n \) into two problems of size \( n/2 \). If we could solve the two problems of size \( n/2 \), then we would be left with the problem of combining two sorted sequences of size \( n/2 \) to form a single sorted sequence of size \( n \). Let us assume that the algorithm MERGE takes as input two sorted sequences and outputs a single sorted array which contains all the elements of the input. From the MERGE algorithm we can construct a divide-and-conquer algorithm MERGESORT:

\[
\text{PROCEDURE MERGESORT}(A, \ n) \\
\text{IF } n > 1 \text{ THEN BREAK A into two arrays A1 & A2 EACH OF SIZE n/2} \\
\text{MERGESORT(A1, n/2)} \\
\text{MERGESORT(A2, n/2)} \\
\text{MERGE(A1, A2)}
\]
As usual it is easy to construct an inductive proof of the correctness of this algorithm.

To calculate the number of comparisons used by this sort, we need to know the number of comparisons used by MERGE. Although this number will depend on which elements are actually in the two subarrays A1 and A2, it is clear that at most \( n - 1 \) comparisons are used because each comparison results in an element being merged into its proper place in the output array. So in worst case the number of comparisons used by MERGESORT is given by the difference equation

\[
C(n) = 2C(n/2) + n - 1
\]

because the algorithm with input of size \( n \) calls itself twice with input of size \( n/2 \) and MERGE uses at most \( n - 1 \) comparisons. For an initial condition we have \( C(1) = 0 \) since the algorithm does nothing when \( n = 1 \). The solution to this equation is

\[
C(n) = n\log n - n + 1
\]

where \( \log \) means logarithm to the base 2. The solution can be easily verified using induction. This solution can also be written as \( C(n) = \Theta(n\log n) \).

If the number of comparisons is the measure of resource usage, then MERGESORT is preferred to the previous sorting algorithms because MERGESORT is \( \Theta(n\log n) \) while the other sorting algorithms are \( \Theta(n^2) \).

### 4.2 The General Form for Divide-and-Conquer Algorithms

In SORT we decreased the size of the problem by 1, whereas in MERGESORT we divided the problem in half. Algorithms like SORT are sometimes called \textit{subtract-and-conquer}. Algorithms like MERGESORT are considered to be true \textit{divide-and-conquer} algorithms.

In most of our examples, a problem will be split into two subproblems of the same type that are half the original size. Here we generalize this slightly and assume that a problem of size \( n \) is split into several (say \( a \)) subproblems, each of size \( n/c \) for some constant \( c \). For many divide-and-conquer algorithms \( c = 2 \) holds, but the same analysis works for general \( c \).

In some cases the splitting process is easy and has negligible computational cost, but we want to allow for the possibility that splitting takes some time. Usually the time-consuming part of a divide-and-conquer algorithm is the combining step, when the answers to subproblems are used to compute the answer to the whole problem. Here is the general form of a divide-and-conquer algorithm:
The general form of a divide-and-conquer algorithm

<table>
<thead>
<tr>
<th>PROCEDURE</th>
<th>\text{DandC}(\text{DATA}, n, \text{SOLUTION})</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{IF}</td>
<td>\textit{n} is small</td>
</tr>
<tr>
<td>\textbf{THEN}</td>
<td>Solve by some special algorithm</td>
</tr>
<tr>
<td>\textbf{ELSE}</td>
<td>\text{SPLIT}(\text{DATA}, n) into ((D_1, n/c))</td>
</tr>
<tr>
<td>\text{DandC}(D_1, n/c, S_1)</td>
<td></td>
</tr>
<tr>
<td>\text{D}_2 := \text{f}_2(\text{DATA}, D_1, S_1)</td>
<td></td>
</tr>
<tr>
<td>\text{DandC}(D_2, n/c, S_2)</td>
<td></td>
</tr>
<tr>
<td>\text{D}_3 := \text{f}_3(\text{DATA}, D_1, S_1, D_2, S_2)</td>
<td></td>
</tr>
<tr>
<td>\vdots</td>
<td></td>
</tr>
<tr>
<td>\text{D}<em>a := \text{f}<em>a(\text{DATA}, D_1, S_1, D_2, S_2, \ldots, D</em>{a-1}, S</em>{a-1})</td>
<td></td>
</tr>
<tr>
<td>\text{DandC}(D_a, n/c, S_a)</td>
<td></td>
</tr>
<tr>
<td>\text{SOLUTION} := \text{f}_{a+1}(\text{DATA}, D_1, S_1, D_2, S_2, \ldots, D_a, S_a)</td>
<td></td>
</tr>
<tr>
<td>\text{RETURN}(\text{SOLUTION})</td>
<td></td>
</tr>
</tbody>
</table>

Often the various subproblems are solved independently, but here we've allowed for the results of previous subproblems to be processed and then fed into the next subproblem. The function \(f_{a+1}\) performs the final combination of results to produce the solution.

### 4.3 Divide-and-Conquer Difference Equations

Consider a divide-and-conquer algorithm which breaks a problem of size \(n\) into subproblems each of size \(n/c\). Assume that there are \(a\) such subproblems and that the algorithm takes \(bn^m\) time to split the original problem into subproblems and to combine the solutions of the subproblems to give the solution to the original problem. A difference equation for the time used by the algorithm is

\[
T(n) = aT(n/c) + bn^m.
\]

The solution for the time used by the algorithm is:

\[
T(n) = \begin{cases} 
\Theta(n^m) & \text{if } a < c^m \\
\Theta(n^m \log n) & \text{if } a = c^m \\
\Theta(n \log_{c}^{a}) & \text{if } a > c^m 
\end{cases}
\]

The derivation of the solution is not too complicated but it involves a number of details, so we will not give it in full. To derive the solution, you can first convert the divide-and-conquer equation to a standard linear constant coefficient difference equation by the substitution \(n = c^r\) and \(T(n) = t_r\). Use standard techniques to solve this difference equation. Determine which term in the solution will be largest, and from the assumptions that \(a\), \(b\), and \(c\) are positive and the initial condition is nonnegative, show that the coefficient of this largest term is positive.
The above solution for the time used by the algorithm also holds for the divide-and-conquer equation

\[ T(n) = aT(n/c) + P(n), \]

where \( P(n) \) is a polynomial of degree \( m \). This holds because the equation is linear. So the solutions due to the various terms in the polynomial can be linearly combined, and therefore the solution due to the highest power term in the polynomial will dominate.

Time here should be taken in a broad sense. It could mean actual time as measured by a clock, but it could also mean the number of times a particular operation is used. For example, when we considered sorting, “time” was the number of comparisons and divisions used.

4.4 Some Divide-and-Conquer Examples

4.4.1 Mergesort

For the running time of MERGESORT we have the equation:

\[ T(n) = 2T(n/2) + bn. \]

This equation holds for either the number of comparisons or for the total running time. Since \( a = 2 \), \( c = 2 \), \( m = 1 \), we have \( a = c^m \) and the solution is \( T(n) = \Theta(n \log n) \).

4.4.2 Polynomial Multiplication

A fairly common numeric problem is calculating the product of two polynomials. The input is the coefficients of the two polynomials, and the desired output is the coefficients of the product polynomial. If the array \( A[0 \ldots n-1] \) contains the coefficients of one polynomial and the array \( B[0 \ldots n-1] \) contains the coefficients of the other polynomial, we want the array \( C[0 \ldots 2n - 2] \) to contain the coefficients of the product polynomial.

<table>
<thead>
<tr>
<th>An iterative algorithm to multiply two polynomials.</th>
</tr>
</thead>
</table>
| \[
\text{FOR } I = 0 \text{ TO } n-1 \\
\text{ FOR } J = 0 \text{ TO } n-1 \\
\text{ \hspace{1cm} } C[I + J] := A[I] \ast B[J] + C[I + J] \\
\text{ ENDFOR} \\
\text{ ENDFOR}
\] |

36
The usual algorithm for this problem proceeds iteratively by multiplying each coefficient of the first polynomial by the first coefficient of the second polynomial, then multiplying each coefficient of the first polynomial by the second coefficient of the second polynomial, and adding these products to the appropriate partial coefficient of the product polynomial; this process is continued until all of the coefficients of the second polynomial have been used. If each of the polynomials have \( n \) coefficients then this algorithm will use \( \Theta(n^2) \) multiplications and \( \Theta(n^2) \) total operations.

The polynomial multiplication problem can also be solved by a divide-and-conquer algorithm which breaks each polynomial in half, multiplies 4 half-size polynomials, and then adds the half-size products in the appropriate way to give the product polynomial. More explicitly, let \( P(x) \) and \( Q(x) \) be two polynomials with \( n \) coefficients each. Write

\[
P(x) = P_0(x) + P_1(x)x^{n/2}
\]

\[
Q(x) = Q_0(x) + Q_1(x)x^{n/2}
\]

where \( P_0(x), P_1(x), Q_0(x) \) and \( Q_1(x) \) are all polynomials of degree \( n/2 - 1 \). Then

\[
P(x)Q(x) = P_0(x)Q_0(x) + (P_0(x)Q_1(x) + P_1(x)Q_0(x))x^{n/2} + P_1(x)Q_1(x)x^n.
\]

For the number of multiplications \( M(n) \), we have

\[
M(n) = 4M(n/2),
\]

and since \( a = 4, c = 2, m = 0 \). We have \( a > c^m \) and the solution is

\[
M(n) = \Theta(n^{\log_2 4}) = \Theta(n^2).
\]

For the total number of operations \( T(n) \), we have

\[
T(n) = 4T(n/2) + bn
\]

because all of the polynomial additions can be carried out with a multiple of \( n \) coefficient additions, and the multiplications by powers of \( X \) simply shift a sequence of coefficients. Since \( a = 4, c = 2, m = 1 \), we have \( a > c^m \) and \( T(n) = \Theta(n^2) \).

There seems little point to this divide-and-conquer approach since it gives us the same running time as the usual algorithm, but it suggests that all 4 of the half-size multiplications might not be necessary because we only need the sum \( (P_0(x)Q_1(x) + P_1(x)Q_0(x)) \) rather than both these products. This sum can be computed using only one multiplication if we have \( P_0(x)Q_0(x) \) and \( P_1(x)Q_1(x) \) because

\[
(P_0Q_1 + P_1Q_0) = (P_0 + P_1)(Q_0 + Q_1) - P_0Q_0 - P_1Q_1.
\]
Our new divide-and-conquer algorithm is

\[ P(x)Q(x) = P_0(x)Q_0(x)(1 - x^{n/2}) + \]
\[ (P_0(x) + P_1(x))(Q_0(x) + Q_1(x))x^{n/2} \]
\[ + P_1(x)Q_1(x)[x^n - x^{n/2}]. \]

This algorithm uses only 3 half-size multiplications so

\[ M(n) = 3M(n/2) \]

and

\[ T(n) = 3T(n/2) + bn \]

because the number of additions and subtractions is still proportional to \( n \). For \( M(n) \) we have \( a = 3, c = 2, m = 0, \) so \( a > c^m \) and \( M(n) = \Theta(n^{\log_2 3}) \). For \( T(n) \) we have \( a = 3, c = 2, m = 1, \) so \( a > c^m \) and \( T(n) = \Theta(n^{\log_2 3}) \). This divide-and-conquer algorithm will be faster than the usual \( \Theta(n^2) \) algorithm because \( \log_2 3 < 2 \).

### 4.4.3 Integer Multiplication

The 3-Half Size multiplication method for polynomials can also be applied to integer multiplication. Let \( I \) and \( J \) be \( n \)-bit numbers, so that

\[ I = I_0 + I_12^{n/2} \quad \text{and} \quad J = J_0 + J_12^{n/2}. \]

Then

\[ I \times J = m_1 + (m_3 - m_1 - m_2)2^{n/2} + m_22^n, \]

where

\[ \begin{align*}
  m_1 &= I_0 \times J_0 \\
  m_2 &= (I_0 + I_1) \times (J_0 + J_1) \\
  m_2 &= I_1 \times J_1
\end{align*} \]

These 3 multiplies are multiplications of \( n/2 \) bit numbers. So as in the polynomial multiplication algorithm, the running time satisfies \( T(n) = 3T(n/2) + bn \) and

\[ T(n) = \Theta(n^{\log_2 3}). \]

The only difference is that for the integer problem we are counting bit operations, while in the polynomial problem we are counting arithmetic operations.
Example 4.4.1. As a small example, let’s multiply 15 and 14.

\[
I = 15 = 3 + 3 \times 4, \quad I_0 = 3, \quad I_1 = 3 \\
J = 14 = 2 + 3 \times 4, \quad J_0 = 2, \quad J_1 = 3
\]

\[
m_1 = 3 \times 2 = 6 \\
m_2 = 3 \times 3 = 9 \\
m_3 = (3 + 3) \cdot (2 + 3) = 6 \times 5 = 30 \\
m_3 - m_1 - m_2 = 30 - 6 - 9
\]

\[
m_1 + (m_3 - m_1 - m_2)2^{4/2} + m_22^4 \\
= 6 + 15 \cdot 4 + 9 \cdot 16 \\
= 210 = 14 \times 15.
\]

4.4.4 Matrix Multiplication

Another problem in which divide-and-conquer leads to a faster algorithm is matrix multiplication. If \( n \times n \) matrices are broken into four \( n/2 \times n/2 \) matrices then the problem is to compute the matrix \( C \), where

\[
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix} = \begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix} \begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}
\]

The straight-forward algorithm is

\[
C_{11} = A_{11}B_{11} + A_{12}B_{21} \\
C_{12} = A_{11}B_{12} + A_{12}B_{22} \\
C_{21} = A_{21}B_{11} + A_{22}B_{21} \\
C_{22} = A_{21}B_{12} + A_{22}B_{22}
\]

The equation for the running time of this algorithm is

\[
T(n) = 8T(n/2) + bn^2,
\]

because there are 8 half-size multiplications and the additions can be done in time proportional to \( n^2 \). This equation has \( a = 8, \ c = 2, \ m = 2, \) so \( a > c^m \) and \( T(n) = \Theta(n\log_2 8) = \Theta(n^3) \)

A faster divide-and-conquer algorithm was designed by Strassen[1969]. This faster algorithm uses only 7 half-size multiplications. Strassen’s algorithm is

\[
M_1 = (A_{12} - A_{22})(B_{21} + B_{22})
\]

\[
M_2 = (A_{11} + A_{22})(B_{11} + B_{22})
\]
\[ M_5 = (A_{11} - A_{21})(B_{11} + B_{12}) \]
\[ M_4 = (A_{11} + A_{12})B_{22} \]
\[ M_5 = A_{11}(B_{12} - B_{22}) \]
\[ M_6 = A_{22}(B_{21} - B_{11}) \]
\[ M_7 = (A_{21} + A_{22})B_{11} \]
\[ C_{11} = M_1 + M_2 - M_4 + M_6 \]
\[ C_{12} = M_4 + M_6 \]
\[ C_{21} = M_6 + M_7 \]
\[ C_{22} = M_2 - M_3 + M_5 - M_7 \]

It is a simple exercise in algebra to prove this algorithm correct. The very real difficulty was discovering the algorithm in the first place. The running time of Strassen’s algorithm obeys the equation
\[ T(n) = 7T(n/2) + bn^2 \]

because there are 7 half-size multiplications and the additions and subtractions of matrices can be carried out in time proportional to \( n^2 \). Thus Strassen’s algorithm has \( \Theta(n^{\log_2 7}) \) running time.

### 4.4.5 Solving a Linear System of Equations

As the final example of this section, consider solving for \( X \), the system of linear equations \( AX = B \). A divide-and-conquer approach to this problem would attempt to break this problem into several subproblems of the same kind. By adding multiples of some \( n/2 \) of the rows of \( A \) to the other \( n/2 \) rows of \( A \), \( A \) can be reduced to the form
\[
\begin{pmatrix}
A_1 & A_2 \\
0 & A_3
\end{pmatrix}
\]
and by using the same transformation the vector \( B \) will be transformed to
\[
\begin{pmatrix}
B_1 \\
B_2
\end{pmatrix}
\]. If we also split \( X \) into
\[
\begin{pmatrix}
X_1 \\
X_2
\end{pmatrix}
\]
the solution to the original problem can be given as the solutions to

\[
\begin{align*}
(4.1) & \quad A_3X_2 = B_2 \\
(4.2) & \quad A_1X_1 = B_1 - A_2X_2
\end{align*}
\]
The equation for the running time of this algorithm is

\[ T(n) = 2T(n/2) + bn^3 \]

because we have two half-size subproblems and it takes time proportional to \( n^3 \) to reduce \( A \) to the required special form. Since \( a = 2, c = 2, m = 3 \), we have \( a < c^m \) and \( T(n) = \Theta(n^m) = \Theta(n^3) \). There is nothing that particularly recommends this algorithm – it has the same running time as the standard Gaussian elimination algorithm – but it is an example of a divide-and-conquer algorithm with \( a < c^m \). An example of a divide-and-conquer algorithm with \( a = c^m \) is MERGESORT. Examples of divide-and-conquer algorithms with \( a > c^m \) are the polynomial multiplication and matrix multiplication algorithms.

### 4.4.6 Newton’s Method

Newton’s method is often used to find the root of a function. It makes use of the idea of “tangent sliding”, that is, approximating the function by a straight line, its tangent, and then finding where this straight line crosses the axis. If a function \( f(x) \) has a root \( R \), then

\[ 0 = f(R) \approx f(x) + (R - x)f'(x) \]

or rewriting

\[ R \approx x - \frac{f(x)}{f'(x)} = N(x). \]

Under reasonable hypotheses, starting from a good initial approximation, Newton’s method will double the number of correct digits. So, if you have an approximation, say \( x \), and \( x \) agrees with \( R \) on the first 5 bits, then \( N(x) \) will agree with \( R \) on about the first 10 bits. Of course, this procedure can be iterated, so \( N(N(x)) \) will agree with \( R \) on about the first 20 bits, and so forth.

Newton’s method can also be used to calculate functions. In particular, Newton’s method can be used to calculate square root and reciprocal. To compute \( \sqrt{A} \), use the function \( f(x) = x^2 - A \) which clearly has \( \sqrt{A} \) as a root. Here, \( f'(x) = 2x \) and so

\[ N(x) = x - \frac{x^2 - A}{2x} = \frac{x^2 + A}{2x} = \frac{x}{2} + \frac{A/2}{x}. \]

So if you know how to divide, you can quickly compute square root. Let \( T(n) \) be the time to compute \( \sqrt{A} \) correct to \( n \) bits. Then the run-time equation is

\[ T(n) = T\left(\frac{n}{2}\right) + bn^2 \]

because Newton’s formula will double the number of correct bits, and the the operations involved in the formula are division by 2 which is merely a shift, division of \( A/2 \) by \( x \) which will take \( O(n^2) \) to compute \( n \) bits, and an addition which will take \( O(n) \) bit operations. So, the time in bit operations to use the
Newton formula will be bounded above by $bn^2$, at least for big enough $n$. The run-time equation can be solved by our usual method where $a = 1$, $c = 2$, and $m = 2$. Since $a < c^n$, 

$$T(n) = \Theta(n^m) = \Theta(n^2).$$

A similar procedure will allow us to divide using Newton’s method and multiplication. Since $A/B = A \cdot (1/B)$, if we can compute $1/B$, then one multiplication suffices to compute $A/B$. To compute $1/B$, we use $f(x) = \frac{1}{x} - B$, which clearly has $1/B$ as its only root. Since $f'(x) = -1/x^2$,

$$N(x) = x - \frac{1/x - B}{-1/x^2} = x(2 - Bx).$$

Let $T(n)$ be the time in bit operations to compute $1/B$ correct to $n$ bits, then

$$T(n) = T\left(\frac{n}{2}\right) + bn^2.$$

As in the case of square root, $T(\frac{n}{2})$ is the time to compute the answer correct to $n/2$ bits and $bn^2$ is an upper bound on the time to compute $N(x)$. Notice that to compute $N(x)$, the formula uses a subtraction which takes $O(n)$, and two multiplications, $B \cdot x$ and $x \cdot (2 - Bx)$, and each multiplication can use $O(n^2)$ bit operations. So again, $bn^2$ is an upper bound on the time to compute $N(x)$ correct to $n$ bits. As before, we note $a = 1$, $c = 2$, and $m = 2$, and since $a < c^n$, 

$$T(n) = \Theta(n^m) = \Theta(n^2).$$

It is worth noting that these uses of Newton’s method demonstrate that division and square root can be computed in the same time order as multiplication. Although we have only demonstrated this result for standard $\Theta(n^2)$ multiplication, notice that $a < c^n$ will hold if $m \geq 1$, so even if we could multiply $n$ bit numbers in $\Theta(n)$ time we would still have

$$T_{\text{SQRT}}(n) \leq T_{\text{DIV}}(n) \leq T_{\text{MULT}}(n).$$

The technique of the next section will allow multiplication in time only slightly worse than $\Theta(n)$, and this faster multiplication can be used to speed up both division and square root.

### 4.5 Polynomial Multiplication and Fast Fourier Transform

In practice the polynomial multiplication algorithms of the last section are not used because there is a much faster algorithm. The faster algorithm is based on the idea that there are two ways to represent a polynomial. A polynomial with $n$ coefficients may be represented by its coefficients or it may be represented by the values of the polynomial at $n$ distinct points. Either representation can be
calculated from the other representation. This can be represented schematically by:

\[
\begin{align*}
\text{coefficients} & \quad \xrightarrow{\text{evaluation}} \quad \text{values} \\
& \quad \xleftarrow{\text{interpolation}} \\
\end{align*}
\]

If we have the coefficients of two polynomials each with \( n/2 \) coefficients and we want the \( n-1 \) coefficients of the product polynomial, we could evaluate each of the input polynomials at the same \( n-1 \) points and multiply the corresponding values to obtain the values of the product polynomial at these \( n-1 \) points. To obtain the desired coefficients we could interpolate a polynomial with \( n-1 \) coefficients through these points.

The difficulty with this approach is that the standard methods for evaluation and interpolation are both \( \Theta(n^2) \), so using them would result in an \( \Theta(n^2) \) method for multiplying polynomials. On the other hand, there is nothing in the above discussion which forces us to use any particular set of points as evaluation and interpolation points. It is easy to evaluate a polynomial at certain points. For example, the value of a polynomial at 0 is simply one coefficient of the polynomial, and the value of a polynomial at 1 is simply the sum of the coefficients.

It is difficult to see how to generalize the idea of evaluation at 0, but the idea of evaluation at 1 can be generalized if we are willing to allow complex numbers. A complex number \( w \) is an \( n^{th} \) root of unity if \( w^n = 1 \). We know by the fundamental theorem of algebra that there are \( n \) complex numbers which are \( n^{th} \) roots of unity. A primitive \( n^{th} \) root of unity is a complex number \( w \) such that \( w^n = 1 \) and \( w^j \neq 1 \) for \( 1 \leq j \leq n-1 \). A primitive root is useful because its powers \( w^0, w^1, w^2, ..., w^{n-1} \) give us the \( n \) numbers which are \( n^{th} \) roots of unity. We can use as our primitive roots the complex numbers \( e^{2\pi i/n} \) which can be calculated by \( \cos(2\pi/n) + i\sin(2\pi/n) \). These numbers only need to be calculated once and stored in a table. This table will also be useful because a primitive \( (n/2)^{th} \) root of unity is \( w^2 \) which will already be in your table.

To evaluate a polynomial \( a_0 + a_1 x + \ldots + a_{n-1} x^{n-1} \) at the \( n \) roots of unity we should compute

\[
\begin{pmatrix}
w^0 & w^0 & \cdots & w^0 \\
w^0 & w^1 & \cdots & w^{n-1} \\
w^0 & w^{1+2} & \cdots & w^{(n-1)+2} \\
\vdots & \vdots & \vdots & \vdots \\
w^0 & w^{1+(n-1)} & \cdots & w^{(n-1)+(n-1)}
\end{pmatrix}
\begin{pmatrix}
a_0 \\
a_1 \\
\vdots \\
a_{n-1}
\end{pmatrix}
\]

Unfortunately if we do this by the obvious algorithm it will take time \( \Theta(n^2) \) even if we have already calculated the entries in the matrix. However, the matrix has very special structure and if we can make use of this structure we may be able to construct a faster algorithm.

To display the structure of the matrix we will permute some of the columns. Since we are planning to break things in half we will assume that \( n \) is a power of
2. Also we will number the rows and columns from 0 to \( n - 1 \), so the indices can be represented by using \( \log n \) bits. Our permutation will interchange column \( j \) with column \( \text{Rev}(j) \), where \( \text{Rev}(j) \) is a number formed by reading the bits of \( j \) in reverse order. In our permuted matrix location \( i, j \) will contain \( w^{\text{Rev}(j)} \).

In the lower left quadrant the matrix will have \( i \geq n/2, j < n/2 \), and \( \text{Rev}(j) \) will be even. So

\[
w^{i\text{Rev}(j)} = w^{(n/2+i-n/2)\text{Rev}(j)} = \left[w^{n/2}\right]^{\text{Rev}(j)} w^{(i-n/2)\text{Rev}(j)} = (-1)^{\text{Rev}(j)} w^{(i-n/2)\text{Rev}(j)} = w^{(i-n/2)\text{Rev}(j)}
\]

and the lower left quadrant will be identical to the upper left quadrant. The lower right quadrant has \( i < n/2, j \geq n/2 \), and \( \text{Rev}(j) \) is odd. So

\[
w^{i\text{Rev}(j)} = -w^{(i-n/2)\text{Rev}(j)}
\]

and the lower right quadrant is the negative of the upper right quadrant. The upper right quadrant has \( i < n/2, j \geq n/2 \) and \( \text{Rev}(j) - 1 = \text{Rev}(j-n/2) \). So

\[
w^{i\text{Rev}(j)} = w^{j\text{Rev}(j-1)} = w^{j\text{Rev}(j-n/2)}
\]

and the upper right quadrant is identical to the upper left quadrant multiplied by the diagonal matrix whose \( i^{th} \) diagonal entry is \( w^{i} \).

The upper left quadrant has \( i < n/2, j < n/2 \), and \( \text{Rev}(j) \) is even. So

\[
w^{i\text{Rev}(j)} = [w^{2}]^{i\text{Rev}(j)/2}
\]

The half-size permuted matrix will contain \( w^{2} \) raised to the \( i\text{Rev}(j) \) power because \( w^{2} \) is a principal \( 2n^{2} \) root of unity. Of course in the half-size matrix \( \text{Rev}(j) \) will have \((\log n) - 1 \) bits. In the larger matrix \( \text{Rev}(j)/2 \) simply removes the low order bit which is 0. Thus the upper left quadrant will be identical to the half-size permuted matrix.

From these considerations we have

\[
F_{2n} = \left( \begin{array}{cc} F_n & DF_n \\ F_n & -DF_n \end{array} \right)
\]

where \( D \) is a diagonal matrix whose \( k^{th} \) entry is \( w^{k} \) where \( w \) is a principal \( (2n)^{2} \) root of unity. As we have seen, the matrix can be used to evaluate a polynomial at all the \( 2n \) roots of unity, and since it turns out that this evaluation gives the coefficients of the discrete Fourier expansion of the polynomial, the matrix is called the (permuted) Fourier matrix.

The form of the above matrix suggests a divide-and-conquer algorithm to compute the discrete Fourier transform of a vector. Since the algorithm will be faster than a method based directly on the definition of Fourier transform, the algorithm is usually called the Fast Fourier Transform (FFT). Assume \( X \) is a
vector with 2n components. Let \( X_1 \) be the first \( n \) components of \( X \), and let \( X_2 \) be the last \( n \) components of \( X \). Then

\[
F_{2n}X = \begin{pmatrix} 
F_nX_1 + DF_nX_2 \\ F_nX_1 - DF_nX_2 
\end{pmatrix}
\]

Notice \( F_nX_1 \) and \( F_nX_2 \) need only be computed once each. Their values can then be combined to give \( F_{2n}X \). To obtain the nice form the columns of the original matrix had to be permuted, so to obtain the effect of the original matrix on a vector, the vector must be permuted before the permuted matrix is applied. The whole algorithm to compute the discrete Fourier transform of \( X \) is:

\[
\text{PERMUTE } X = \begin{pmatrix} 
X_1 \\ X_2 
\end{pmatrix} \text{ HALF-SIZE } F \begin{pmatrix} 
F_nX_1 \\ F_nX_2 
\end{pmatrix} \text{ COMBINE } \begin{pmatrix} 
F_nX_1 + DF_nX_2 \\ F_nX_1 - DF_nX_2 
\end{pmatrix}
\]

In this process the matrix \( F \) never really needs to exist. The \( F \)'s in the above scheme are simply recursive calls to the procedure \( F \). The nonzero elements of \( D \) are powers of \( w \). So the powers of \( w \) can be computed once, stored in an array, and used from the array when necessary.

The running time for this algorithm can be considered in two parts: the time to permute, and the time for the recursive procedure \( F \). The permutation uses the bit reversals of the number 0 through \( 2n - 1 \). Since these numbers can be represented using \( \log n + 1 \) bits, the permutation can be computed in \( \Theta(n \log n) \). The permutation can be done in \( \Theta(n) \) if the bit reversals are pre-computed. The running time for the procedure for \( F \) obeys the difference equation:

\[
T(2n) = 2T(n) + bn
\]

because there are two half-size recursive calls, and the multiplication by \( D \) and the additions and subtractions can be carried out in time proportional to \( n \). Thus

\[
T(2n) = \Theta(n \log n).
\]

To use this fast algorithm for polynomial multiplication, we need a fast way to interpolate, or what is the same, a fast way to multiply a vector by the inverse of the \( F \) matrix. For the inverse procedure we need the idea of conjugate. The conjugate of a complex number \( a + bi \) is \( a - bi \). We represent the conjugate of the complex number \( Z \) by \( Z^* \). The product \( ZZ^* = a^2 + b^2 \), which is the square of the length of the vector which represents \( Z \). For a root of unity \( w \), the product \( wuw^* = 1 \) because the length of \( w \) is 1. The conjugate of a complex matrix is the transpose of the matrix with each element replaced by its conjugate. The
The inverse of $F$ matrix is its conjugate divided by the dimension because

$$F_{2n}F_{2n}^* = \begin{pmatrix} F_n & DF_n \\ F_n^* & -DF_n \end{pmatrix} \begin{pmatrix} F_n^* & F_n^* \\ F_n^* D^* & -F_n D^* \end{pmatrix} = \begin{pmatrix} F_n F_n^* + DF_n F_n^* D^* & F_n^* - DF_n F_n^* D^* \\ F_n F_n^* D^* & F_n^* + DF_n F_n^* D^* \end{pmatrix} = \begin{pmatrix} 2nI & 0 \\ 0 & 2nI \end{pmatrix}.$$  

So $F^*X$ can be quickly computed because

$$F^*X = F^*X^{**} = (F^T X^*)^* = P(FPX^*),$$

where $F^T$ is the transpose of $F$ and $P$ is the bit-reversal permutation matrix. The last equality follows because $F^T = PP$. Finally the whole algorithm to compute the coefficients of the product polynomial is:

1. Place the $n$ coefficients of the first polynomial in the first $n$ components of the vector $X$ of dimension $2n$. The other $n$ components of $X$ will contain 0.
2. Similarly place the $n$ coefficients of the second polynomial in the vector $Y$.
3. Permute both $X$ and $Y$.
4. Use the recursive algorithm to compute both $FX$ and $FY$.
5. Componentwise multiply $FX$ and $FY$ to obtain a vector $Z$.
7. Use the recursive algorithm to compute $FZ$.
8. Conjugate and divide each component of $FZ$ by $2n$.

The resulting vector will contain the $2n - 1$ coefficients of the product polynomial as its first $2n - 1$ components. The last component should contain 0. Since this algorithm will likely be carried out using floating point arithmetic, the last component will likely not be exactly 0, but the value in this component will give us an estimate of how exact the other components are.

### 4.6 How Practical Are Divide-and-Conquer Algorithms?

After creating algorithms in the abstract world of analysis of algorithms, we should now look back to the real world of programs and ask if the divide-and-conquer algorithms, which are theoretically faster, will be practically faster, and
whether they will be practical at all. There are several reasons to be skeptical about the practicality of these algorithms. In our abstract world recursion was available at no cost. In the real world recursion may be unavailable or it may have a high cost. In the abstract world we computed time only to order, not distinguishing $10^{23}n^2$ from $2n^2$, but in the real world these functions are very different.

Let us address recursion first. The tempting thing for a theoretician is to say that all modern programming languages have recursion so there is no problem. Unfortunately large amounts of programming are done in older languages which do not support recursion. So there is still some reason for considering converting recursive algorithms to iterative algorithms. Even when recursion is available it may not be a good idea to use it. Theoretically we often assume that passing data to procedures is free. In the analysis in this section we have ignored space usage, and we have particularly ignored extra space used to maintain stacks for the recursive procedures. In the real world neither of these should be ignored. Sometimes by doing a space analysis we find that the recursive stack does not get very big, so we are justified in ignoring it. In other cases, we find that the recursive stack gets very large because we are putting copy after copy of the same data on the stack. This problem can often be overcome by having only a single global copy of the data and, instead of passing the data to the recursive procedure, passing only a pointer to the particular part of the global copy that is needed. In some cases, we find that data passed to the recursive routine may be replaced by a much simpler global structure. In the Towers of Hanoi example, the number of disks was repeatedly passed to the recursive procedure, but the equivalent information could be maintained in a global counter. There are a number of methods which will suffice to convert a recursive routine to an iterative routine, but if they are general enough to work for all recursive routines, they are too general to result in any saving in time or space. Special features of a particular recursive routine can often be exploited in producing a more efficient iterative routine. We have seen how special features can be exploited in the Towers of Hanoi example. As another example, the MERGESORT algorithm works from the top-down, splitting a big array and passing each half to the recursive routine. This routine can be made iterative by working bottom-up. Consider each element as a sorted array of size 1 and merge these 2 by 2 until you have sorted arrays of size 2. Again merge these 2 by 2 until you have sorted arrays of size 4. Continue merging until the entire array is sorted. This can be accomplished without passing any arrays to procedures, by keeping track of the size and the beginning and ending indices of the subarrays you are merging.

In summary, it is often worthwhile to convert recursive algorithms to iterative algorithms, but you should exploit the special features of the problem to create an efficient iterative algorithm.

Are faster algorithms always faster? Probably not. In our analysis we have concentrated on asymptotic time order. We expect our faster algorithms to be faster than slower algorithms for big enough inputs, but the slower algorithms may well be faster for small inputs. For example, in sorting there are $\Theta(n^2)$ algorithms which are faster than an $\Theta(n \log n)$ algorithm for $n < 20$, but for

47
$n \geq 20$ the $\Theta(n \log n)$ algorithm is faster. In this case unless we are dealing with very small data sets, the faster algorithm really is faster. On the other hand, there are $\Theta(n^{2.5})$ algorithms for matrix multiplication. These algorithms don’t become faster than the standard $\Theta(n^3)$ algorithm until one is dealing with 50,000 $\times$ 50,000 matrices. Since no one is presently trying to multiply matrices this large, the $\Theta(n^{2.5})$ algorithms are only theoretically interesting.

The algorithms we have designed may fail to be practical in other ways. The algorithms are not fool-proof, that is, the algorithms assume that they will receive the type of data they are expecting, and their behavior on unexpected data may well be rather strange. Practical programs should check the input data to make sure it is of the expected kind and issue a warning if the data is not of the expected kind. The algorithm may also not be practical if it solves the wrong problem. For example, the FFT-based polynomial multiplication algorithm is designed for dense polynomials, that is, polynomials in which all or almost all the coefficients are non-zero, but many large scale polynomial multiplication problems arise in which the polynomials are sparse, that is, have almost all the coefficients equal to zero. For such sparse problems there are algorithms which will easily outperform the FFT-based algorithm. Similarly there are special algorithms for sorting which will be slow in general but will be very fast when the input data is almost sorted.

In summary, there are practical issues which should be considered before a theoretically good algorithm is used as a practical program.

**Exercises**

**Ex 4.1.** You have a large number of coins and a pan balance. You may put any number of coins in each pan of the balance. The balance will tell you if the set of coins in one pan weighs the same as the set of coins in the other pan, or it will tell you which set is heavier. Somewhere among your coins is one coin which has a different weight from the other coins. All the coins except this odd coin have exactly the same weight. The problem is to find the odd coin.

Design a divide-and-conquer algorithm to solve this problem. You may assume that the number of coins is a power of 3. Prove that your algorithm is correct. Give and solve a difference equation for the number of times your algorithm uses the balance.

**Ex 4.2.** Design a divide-and-conquer algorithm to find the two largest elements in an array. Prove that your algorithm is correct. Calculate the number of comparisons it uses. Show by example that your algorithm uses more comparisons than necessary.

**Ex 4.3.** Two strings $C_1$ and $C_2$ commute (that is, $C_1 C_2 = C_2 C_1$) iff there is a string $w$ and two natural numbers $k_1$ and $k_2$ so that $C_1 = w^{k_1}$ and $C_2 = w^{k_2}$. Start from an inductive proof that $w$ exists and construct an algorithm to find $w$. Here $w^k$ means $ww...w$, that is $k$ copies of $w$. 48
Chapter 5

AVERAGE CASE

5.1 What is Average Case?

Up till now we have considered the running time of an algorithm to be a function of the size of the input, but what happens when there are several different inputs of the same size? An algorithm may treat all inputs of the same size in the same way, or it may handle some inputs more quickly and some inputs more slowly. The maximum of the running time over all inputs of the same size is called the worst case running time. The minimum running time over all inputs of the same size is called the best case running time. The running times averaged over all inputs of the same size is called the average case running time. The average case running time is not the same as the average of the worst case and the best case. It is often difficult to calculate the average case time because the probability associated with each of the various inputs of a particular size is unknown. For definiteness and simplicity, it is often assumed that each input with the same size is equally likely to occur. With this assumption average case can be calculated.

5.2 Some Examples of Average Case Behavior

As an example of average case behavior, consider the following algorithm:

PROCEDURE LARGETWO
    FIRST := B[1]
    SEC := B[2]
    FOR I := 2 TO n DO
        IF B[I] > FIRST
            THEN SEC := FIRST; FIRST := B[I]
        ELSE IF B[I] > SEC
            THEN SEC := B[I].
This algorithm should find the two largest elements in an array. We will consider the number of comparisons of array elements it uses to accomplish this task. In the FOR loop, for each I the algorithm makes either 1 or 2 comparisons. In best case the algorithm makes 1 comparison for each I giving a total of \( n - 1 \) comparisons. In worst case the algorithm makes 2 comparisons for each I giving a total of \( 2(n - 1) \) comparisons.

Let \( A(n) \) be the number of comparisons used on average by this algorithm. Since the algorithm proceeds in one direction across the array, we may reasonably assume that for the first \( n - 1 \) elements the algorithm will on average use \( A(n - 1) \) comparisons, that is, the same number it would use if the last element did not exist. For the last element it will use at least one comparison. It will use a second comparison exactly when \( B[n] \leq \text{FIRST} \), but since \( \text{FIRST} \) will be the largest of the first \( n - 1 \) elements, the algorithm will use a second comparison as long as \( B[n] \) is not the largest element in the array. The probability that \( B[n] \) is not the largest element is \( \frac{n-1}{n} \), if we assume that each element is equally likely to be the largest element. From these considerations we have:

\[
A(n) = A(n - 1) + 1 + \frac{n - 1}{n} = A(n - 1) + 2 - 1/n.
\]

For an initial condition we have \( A(2) = 3/2 \), since for two elements the algorithm is equally likely to use one or two comparisons. The solution to this difference equation is

\[
A(n) = 2(n - 1) - \sum_{j=1}^{n} 1/j.
\]

and for large \( n \),

\[
A(n) \to 2(n - 1) - \log n + \text{const}.
\]

So we have that the average case of this algorithm is very close to the algorithm’s worst case. Notice that if it took \( (\text{worst} + \text{best})/2 \) we would get \( 3/2(n - 1) \), which will be a severe underestimate of the average case. (As an aside, we should mention that this is a poorly set up algorithm since it assumes that the array has at least two locations. If this algorithm were used with an array containing a single element, the results would be unpredictable.)

The QUICKSORT algorithm has a more complicated average case analysis. The algorithm is:

**PROCEDURE QUICKSORT(A)**

1. **PICK AN ELEMENT \( \alpha \) of A AT RANDOM**
2. **DIVIDE A INTO A1 (THE ELEMENTS OF A WHICH ARE LESS THAN \( \alpha \))**
3. **A2 (THE ELEMENTS OF A WHICH EQUAL TO \( \alpha \))**
4. **A3 (THE ELEMENTS OF A WHICH ARE GREATER THAN \( \alpha \))**
5. **RETURN( QUICKSORT(A1) \circ A2 \circ QUICKSORT(A3) )**

50
The worst case for QUICKSORT occurs when A1 or A3 contains \( n - 1 \) elements. If we let \( W(n) \) stand for the worst case running time of QUICKSORT, we have

\[
W(n) = W(n - 1) + bn
\]

because the separation into A1, A2, A3 takes time proportional to \( n \). From this we have \( W(n) = \Theta(n^2) \). The best case for QUICKSORT occurs when A1 and A3 are each approximately \( n/2 \). If we let \( B(n) \) be the best case running time, we have approximately

\[
B(n) = 2B(n/2) + bn
\]

and \( B(n) = \Theta(n \log n) \). For the worst case to occur each recursive splitting must have one of the sets A1 or A3 empty. For the best case to occur each recursive splitting must have A1 and A3 of approximately equal size.

Will the average case be like the worst case or like the best case? We might expect nearly equal splits to occur more frequently than one-sided splits, so we could guess that average case is probably closer to best case than to worst case. If we let \( A(n) \) be the average case running time and assume that every split is equally likely, we have

\[
A(n) = \frac{1}{n} \sum_{k=1}^{n} [A(k - 1) + A(n - k) + b(n + 1)],
\]

where \( b(n + 1) \) is the time to split and combine. Then

\[
nA(n) - (n - 1)A(n - 1) = \sum_{k=1}^{n} A(k - 1) - \sum_{k=1}^{n-1} A(k - 1)
\]

\[
+ \sum_{k=1}^{n} A(n - k) - \sum_{k=1}^{n-1} A(n - 1 - k)
\]

\[
+ bn(n + 1) - bn(n - 1)
\]

\[
= 2A(n - 1) + 2bn
\]

and so \( nA(n) = (n + 1)A(n - 1) + 2bn \) and \( \frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2b}{n+1} \). Letting \( Z(n) = A(n)/(n+1) \), we have \( Z(n) = Z(n-1) + \frac{2b}{n+1} \), which has the solution \( Z(n) = c_1 + 2b \sum_{k=1}^{n} \frac{1}{k+1} \). But \( Z(n) \to c_2 + c_3 \log n \), so \( A(n) \to c_2(n + 1) + c_3(n + 1) \log n \), giving \( A(n) = \Theta(n \log n) \).

In summary, the average case behavior of an algorithm is somewhere between the worst case and the best case behavior, but it may be very close to the best case behavior or very close to the worst case behavior.

**Ex 5.1.** A number of useful tricks were used in deriving the average case behavior of QUICKSORT. For the procedure LARGETWO set up a difference
equation with a summation by assuming that each element is equally likely to be the largest element. Then employ the techniques used on QUICKSORT to obtain the simple difference equation we used for LARGETWO.

Ex 5.2. For the following procedure do an average case analysis. Is the average case nearer to worst case or to best case?

PROCEDURE BIGTWO

FIRST := B[1]
SEC := B[2]
FOR I := 2 TO n DO
  IF B[I] ≥ SEC
    THEN IF B[I] > FIRST
      THEN SEC := FIRST; FIRST := B[I]
    ELSE SEC := B[I].

Ex 5.3. Show that our discussion about the best case for QUICKSORT is correct by doing the following.

(a) Show that if \( B(n) \) is superlinear then the minimum of \( B(n - i) + B(i) \) occurs when \( i \) is about \( n/2 \).

(b) Also, show that if \( B(n) \) is linear, i.e. \( B(n) = an + b \), then \( B(n-i) + B(i) \) has the same value for all choices of \( i \).

(c) Finally, explain why \( B(n) \) must be growing at least as fast as a linear function.
Chapter 6

LOWER BOUNDS

6.1 Trivial Lower Bounds

A lower bound is a statement that every algorithm which solves a particular problem must use at least so much of a particular resource. We have already met some lower bounds in the Towers of Hanoi example. There we argued that every algorithm for the problem must take at least $\Theta(2^n)$ time because the output must contain $2^n - 1$ moves. Such a lower bound is called a trivial lower bound because it is obvious that an algorithm must take at least as much time as it takes to print its output. Although such a bound is called trivial it may not be trivial to compute the length of the output. For example, it was not immediately obvious that the output for the Towers of Hanoi must have length $2^n - 1$.

Trivial lower bounds can also depend on the input. If we can show that to compute the correct answer an algorithm must read all of its input, then the algorithm must use time which is at least as great as the length of the input. For example, if an algorithm is supposed to multiply an $n \times n$ matrix times an $n$ component vector, we have the trivial lower bound of $\Theta(n^3)$ because if the algorithm does not read some of the input, then there are vectors and matrices for which the algorithm gives the wrong answer. As another example, consider an algorithm which is supposed to find the largest element in an $n$ element array. This algorithm must take time at least $\Theta(n)$, since if it doesn't look at all the elements in the array there are arrays for which the algorithm will give an incorrect answer. There are problems in which an algorithm does not have to look at the whole input. For example, to determine if a list is empty an algorithm only has to look at the first element of the list; it does not need to look at all elements of the list.

Input and output lower bounds can vary widely. Sometimes the output bound will be larger than the input bound. Sometimes the input bound will be larger than the output bound. In the Towers of Hanoi problem the output bound is $\Theta(2^n)$, but the input bound is only $\Theta(\log n)$ because $n$ and the names
of the towers can be specified in \( \Theta(\log n) \) bits. In the variant of the Towers of Hanoi in which one is asked if a given configuration is used in moving the disks from Tower A to tower C, the output bound is \( \Theta(1) \) since the yes or no answer can be specified with a single bit, while the input bound is \( \Theta(n) \) because a \( \Theta(n) \) bits are needed to specify which tower contains which disk.

So far our lower bounds have been \textit{best case} bounds, that is, even in best case every algorithm must use at least the time specified by the lower bound. Consider the problem of determining if a list contains a particular element. The desired element could be the first element in the list, so the \textit{best case} lower bound is a \( \Theta(1) \). On the other hand, to be sure that the desired element is not in the list an algorithm must look at every element, so the \textit{worst case} lower bound is \( \Theta(n) \).

### 6.2 Lower Bounds on Specific Operations

The trivial lower bounds merely state that an algorithm must do the required input and output; they say nothing about any computation the algorithm must perform. To obtain lower bounds involving specific operations, the operations which the algorithm is allowed to use must be specified. In this section we will consider algorithms in which the only allowed operations on input elements are comparisons. This will suffice for the problems considered. For other problems, the operations of addition, subtraction, and multiplication might be allowed. We refer the interested reader to Winograd[1980].

#### 6.2.1 Input Bound

Let us consider deriving a lower bound on the number of comparisons needed to find the largest element in an array. We use an input bound argument. If some element is not compared to any other element then either the algorithm says that the uncompared element is the largest element, which will be false in some cases, or the algorithm says that some other element is largest, which will be false when the uncompared element is largest. Since elements can be compared two at a time, we have that any algorithm which correctly finds the largest using only comparisons must use at least \( n/2 \) comparisons.

#### 6.2.2 Graph Theory Bounds

We have that \( n/2 \) comparisons are needed, but we expect that \( n - 1 \) comparisons are needed because our methods for finding the largest use \( n - 1 \) comparisons. We can consider an algorithm as forming a directed graph. Each time a comparison is made, a directed edge from the larger to the smaller element is put into the graph. When the largest element has been found then for every other element there will be a directed path in the graph from the largest to the other element. If not, we could replace the element without a path by an element larger than the element that the algorithm reports to be largest. Since the algorithm is assumed
to use only comparisons, the algorithm cannot detect this replacement. So if not all these paths exist, then the algorithm will give incorrect answers. If all these paths exist, then the graph formed from the directed graph by replacing the directed edges with undirected edges will be connected. Since a connected graph has at least \( n - 1 \) edges and since each edge corresponds to a comparison, \( n - 1 \) comparisons are needed.

A directed tree argument will also produce the same lower bound. We want to construct a binary tree which has the largest element at its root. We want a binary tree because our comparisons are binary (i.e. two way). Let the original elements be the leaves of this tree. Each time two elements are compared, add two edges from the compared elements to a new node which contains the larger of the two elements. Continue in this way to construct a binary tree which has the largest element at its root. A comparison corresponds to an internal node with two children. How many such nodes are there? It is well known and Exercise 6.3 asks you to prove that:
A rooted binary tree with \( n \) leaves has at least \( n - 1 \) internal nodes with two children.
So again we have the \( n - 1 \) comparison lower bound.

### 6.2.3 Dynamical System

A different way to obtain this lower bound is to think of finding the largest as a dynamic system. As an algorithm proceeds it classifies each element as belonging to one of three sets: the set \( U \) which contains elements which have been compared to no other element; the set \( S \) of elements which have been compared to some other element and have been found to be smaller; the set \( L \) of elements which have been compared to some other element and have been found to be larger than any element they have been compared to. The states of the dynamic system will be vectors with three integer components giving the size of the sets \( U, S, \) and \( L \). When the algorithm starts the dynamic system will be in state \( (n, 0, 0) \), and when the algorithm correctly terminates the dynamic system will be in state \( (0, n - 1, 1) \). In each comparison of the algorithm, the state changes in one of the six following ways, where the sets from which the elements to be compared are indicated by the names of the sets above the arrows.

\[
\begin{align*}
(U, S, L) & \xrightarrow{U:S} (U-2, S+1, L+1) \\
(U, S, L) & \xrightarrow{S:S} (U, S, L) \\
(U, S, L) & \xrightarrow{L:L} (U, S+1, L-1) \\
(U, S, L) & \xrightarrow{U:S} (U-1, S+1, L) \text{ or } (U-1, S, L+1) \\
(U, S, L) & \xrightarrow{U:L} (U-1, S+1, L) \\
(U, S, L) & \xrightarrow{S:L} (U, S+1, L-1) \text{ or } (U, S, L)
\end{align*}
\]
Since we have a model of the action of any algorithm which finds the largest element using only comparisons, we can establish a lower bound for the algorithm by establishing a lower bound on the number of transitions in the dynamic system required to go from state \((n, 0, 0)\) to state \((0, n - 1, 1)\). Consider the middle component. It must increase from \(0\) to \(n - 1\), but this component can increase by at most 1 in any of the transitions. Hence at least \(n - 1\) transitions in the dynamic system and \(n - 1\) comparisons in the algorithm are required.

### 6.2.4 Chicken Argument

There is even an simpler argument that \(n - 1\) comparisons are needed. This argument is easy to remember because like many fables it makes use of a story about animals. In a farmyard, there are \(n\) chickens, and they need to decide who is Boss chicken. Being chickens, the only way they can decide is by having fights between two chickens. What is the fewest number of fights needed to find the Boss? Clearly, when the Boss is found, every other chicken must be known not to be Boss, and a chicken cannot be Boss if it has lost a fight. Hence, each of \(n - 1\) chickens must have lost a fight, and since each fight has only one loser, there must have been at least \(n - 1\) fights.

This chicken argument translates immediately into the find largest problem. The chickens are elements. The fights are comparisons between elements. So, by the chicken argument at least \(n - 1\) comparisons are needed to find the largest among \(n\) elements.

### 6.2.5 Bad Example

It is important to notice that this result really does depend on the type of algorithm being considered. By allowing different operations the largest element can be located using only \(\log n\) comparisons. Consider the following algorithm:

```plaintext
FUNCTION BIG(i, j)
    IF j = i+1
        THEN IF a[i] ≥ a[j]
            THEN BIG := i
            ELSE BIG := j
        ELSE IF \(\sum_{K=1}^{i} NUM[K] \geq \sum_{K=1}^{j} NUM[K]\)
            THEN BIG := BIG(i, (i+j)/2)
            ELSE BIG := BIG((i+j)/2 + 1, j)
    END FUNCTION
```

56
FOR I := 1 TO n DO
    NUM[I] := n**a[I]
    LARGEST := a[BIG(1,n)]

In this algorithm we are assuming that the elements are distinct positive integers. The ** indicates exponentiation. Let \( C(n) \) be the number of comparisons used by the algorithm with \( n \) elements; then \( C(n) = C(n/2) + 1 \) and \( C(2) = 1 \). Thus \( C(n) = \log n \). (If you also want to count the comparison between \( j \) and \( i + 1 \), there are \( 2 \log n \) comparisons.) The trick here is to make sure that the largest element is in the half-size subset with the largest sum. Consider the case in which LARGEST is in the first half. Then

\[
\sum_{K=\frac{n}{2}+1}^{n} NUM[K] \leq \frac{n}{2} \sum_{K=1}^{nLARGEST} = \frac{n^{LARGEST}}{2}
\]

\[
< n^{LARGEST} + \frac{n}{2} - 1 \leq \sum_{K=1}^{n} NUM[K]
\]

so this algorithm will work correctly. The use of exponentiation allows us to make the largest element so large that we can do a binary search through the subsets and locate the largest element quickly.

This example should indicate that lower bound proofs are sensitive to operations allowed in the algorithm. It is thus important in lower bound arguments to indicate the class of algorithms being considered.

### 6.3 Lower Bounds on Sorting

A very traditional and ubiquitous problem is sorting. Given \( n \) elements from a totally ordered set put them in order from the smallest to the largest. We have already encountered several algorithms for this problem. Some had \( \Theta(n^2) \) worst case running time. Some had \( \Theta(\log n) \) worst case running time. Some had \( \Theta(n \log n) \) average case running time. Some had \( \Theta(n \log n) \) best case running time. From these examples, one might conjecture that an \( \Theta(n \log n) \) lower bound could be established for best case and hence for average and worst cases, at least for algorithms which sort by using only comparisons. Unfortunately this conjecture is false. If the elements are already sorted, an algorithm could check in only \( n - 1 \) comparisons that the elements were in order. It is not essential that the elements actually be in order; for any particular arrangement of the elements, an algorithm can be created which tests for this arrangement using \( n - 1 \) comparisons, and if the elements are in this particular arrangement the algorithm will sort them with no additional comparisons. So any best case lower bound for sorting must be less than or equal to \( n - 1 \) comparisons. But no smaller lower bound is possible because if an array is sorted then the largest
element can be found using no comparisons, since the largest element is the last element in the array. Since we have already demonstrated a best case lower bound of $n - 1$ comparisons to find the largest element, $n - 1$ comparisons is also the best case lower bound for sorting.

It still is reasonable to believe that a lower bound of $\Theta(n \log n)$ may be possible for worst case and average case. To obtain these bounds we introduce a model of computation called the comparison tree. Each internal node of a comparison tree contains an expression like $a : b$ which means compare $a$ to $b$. From a comparison node there are two arrows to other nodes. One arrow is labeled $a \leq b$ and the other arrow is labeled $a > b$. There are also external nodes or leaves which indicate the termination of the tree. The comparison tree should capture the idea of an algorithm which uses only comparisons. The algorithm starts at the root of the tree, does the comparison there, and then follows the appropriate arrow. This process continues until a leaf is reached. The following picture shows a comparison tree for an algorithm which sorts 3 elements.

```
   a:b
     /   \
    /     \ 
   a \leq b   a > b
     |       |
    / \     / \ 
   a \leq c a \geq c   a \leq c a \geq c
     |   |     |   |
    / \ / \     / \ / \
   a \leq c b \leq c   a \geq c c \leq a
     |   |     |   |
    / \ / \     / \ / \
   a \leq c b \leq c c \geq a
```

In this example there are 6 leaves because there are 3! possible orderings of 3 elements. A comparison tree for sorting $n$ elements will have $n!$ leaves. In this example there are sets which will be sorted using only 2 comparisons, but there are also sets for which this algorithm uses 3 comparisons.

Let us define the depth of the root of a tree as depth 0, and the depth of a node in the tree as $1 + \text{depth}$ of its parent node. Then the maximum number of comparisons to sort any set using a particular comparison tree algorithm is the maximum depth of a node in the tree. In the example, there is a node (a leaf) of depth 3 and there is a set on which this algorithm uses 3 comparisons.

To obtain a worst case lower bound, we will consider the depth of comparison trees for sorting. Since there are only two arrows from each node, the number of nodes can only double when the depth is increased by 1. So there are at most $2^D$ nodes at depth $D$, and summing the nodes at every depth there are at most $2^{D+1}$ nodes in a tree with maximum depth $D$. Since there are $n!$ possible
orderings of $n$ things, a comparison tree for sorting must have at least $n!$ leaves. From these two facts we have

$$2^{D+1} \geq n!$$

Taking logs and using Stirling’s approximation that

$$n! \sim n^n e^{-n \sqrt{2\pi n}}$$

we have

$$D \geq \Theta(n \log n)$$

and hence $\Theta(n \log n)$ is a worst case lower bound for the number of comparisons used by an algorithm which sorts using only comparisons. The comparison tree model can also be used to produce an average case lower bound. The average number of comparisons will be the average depth of a leaf of the comparison tree. By average here we mean that we assume that each ordering of the $n$ elements is equally likely and hence each ordering has probability $1/n!$. Now the average depth of a leaf will be at least as great as the average depth of a node. To minimize the average depth of a node, we will consider the full tree with total depth $L$ where $\log n! - 1 < L \leq \log n!$. By a full tree we mean a tree which has exactly $2^i$ nodes at depth $i$. This full tree will give us a lower bound because any comparison tree for sorting has at least $\log n!$ nodes and we are making their depths as small as possible. The average depth of a node in our full tree is at least

$$\frac{1}{n!} \sum_{i=0}^{L} i 2^i = \frac{2}{n!} ((L - 1)2^L + 1) \geq \frac{2}{n!} ((\log n! - 2) \frac{n!}{2} + 1)$$

Again using Stirling’s approximation this is asymptotic to $\Theta(n \log n)$, and $\Theta(n \log n)$ is an average case lower bound for sorting. We summarize these results in the following proposition.

**Proposition 10.** Any algorithm which sorts using only comparisons must use at least

1. $\Theta(n)$ comparisons in best case
2. $\Theta(n \log n)$ comparisons in average case
3. $\Theta(n \log n)$ comparisons in worst case.

We close this section with two reminders. One. average case here assumes that all $n!$ orderings are equally likely. If there are significantly fewer orderings which are very likely to occur, then the above average case bound does not hold. For example, if the inputs are almost-sorted then there are $\Theta(n)$ average case algorithms. Two, the bounds in the proposition apply for algorithms which sort using only comparisons. If different types of operations are allowed, then these bounds may not hold.
6.4 Information Theoretic Bounds

Information theory is useful in many areas of computer science including analysis of algorithms. The basic idea of information theory is counting which leads to the basic result:

A string of \( n \) bits allows one to specify \( 2^n \) different objects. Conversely, \( m \) distinct objects can be specified using \( \lceil \log m \rceil \) bits.

This statement leads immediately to a lower bound on sorting. Since there are \( n! \) different sorted orders at least \( \log(n!) \) bits are need to specify which order you have. But, each binary comparison only give one bit of information, and so at least \( \log(n!) \) binary comparisons are needed. As we saw above, from Stirling's approximation, \( \log(n!) = \Theta(n \log n) \), and so at least \( \Theta(n \log n) \) binary comparisons are needed for sorting.

This same counting technique also gives a lower bound for the coin problem (Ex 4.1). Since there are \( n \) coins and all you have to do is pick out one of them (the bad coin), you immediately know that \( \log(n) \) bits are required. But in this problem, you are using 3-way comparisons, instead of the more usual binary comparisons. Hence, you want to count trits (base 3 digits or ternary \{base 3\} bits). So \( \log_3(n) \) trits are required, and since you get at most 1 trit per weighing, at least \( \log_3(n) \) weighings are required. { Since you don't know whether the bad coin is heavier or lighter, you may not get a full trit of information for each weighing.}

The marvelous thing about information theory is that extends the idea of counting to weighted counting. For example, if there were two objects in a set, but only one of them would ever appear, how many bits would you need to specify which one appeared? Specifically, if we had two events \( A \) and \( B \), and we are sure that event \( A \) will occur, then we don't need any more bits to confidently predict that \( A \) will occur. On the other hand, if \( A \) and \( B \) are equally likely, then we'll need one bit to specify which one occurred. What about the other cases, neither when the events are equally likely, nor when one of the events is sure. Here, information theory tells us to compute the entropy and says that for events \( A_1, \ldots, A_n \) which occur with respective probabilities \( p_1, \ldots, p_n \), then

\[
\text{Entropy} = - \sum p_i \log p_i.
\]

This entropy is the amount of information we get when we find which one of the \( A_i \) events occurred, or the extra information we would need to predict which of these events will surely occur.

Let's apply the entropy formula to the average case analysis of the sorting problem. As we have mentioned, we almost never know the actual probability distribution over the set of possible inputs, so, we often assume that all inputs are equally likely. For sorting, there are \( n! \) different permutations of \( n \) distinct things, and thinking uniformly, we would assume that each of these permutations has the same probability, i.e. \( p_i = 1/n! \). Applying the entropy formula,
we get

\[- \sum p_i \log p_i = - \sum_{i=1}^{n} \frac{1}{n!} \log \left( \frac{1}{n!} \right) \]

\[= \frac{1}{n!} \log n! \sum_{i=1}^{n} \frac{1}{n!} \quad \text{since each term is independent of the summation index} \]

\[= \frac{1}{n!} \log n! \cdot n! = \log n! \]

\[= \Theta(n \log n) \quad \text{by Stirling’s formula.} \]

So, we can see that “on average”, sorting is not easier than “worst case”.

(Actually this only says that the lower bounds calculated by this method are the same. It would be plausible that the real lower bound for worst case might be much higher. But, since we have MERGESORT which has \( \Theta(n \log n) \) run time (even in worst case), we know that the worst case lower bound is “tight”, i.e. the running time of some algorithm matches the lower bound we can prove. Since the average case lower bound is the same as the worst case lower bound (which is tight), we know that the average case lower bound is also tight!)

Notice how simple this information theory proof is. It is certainly less complicated than setting up a tree model, and having to calculate the average depth of leaves in the tree.

We can say a little bit about sorting assuming other than the uniform probability distribution. Obviously if we know what the sorted order is, we can report this sorted order without making any comparisons. If there is one sorted order which is overwhelming likely, we can always report this extremely probable order. Of course, in this case we will occasionally make a mistake, but the probability of a mistake will be near 0. The entropy of such a probability distribution is also quite small. For example, if \( A_1 \) has probability \( p = 1 - 1/n! \) and \( A_2, \ldots, A_n \) each have probability

\[\frac{1-p}{n!-1} = \frac{1}{n!(n!-1)} \]

then the entropy is

\[- \sum p_i \log p_i = - \left( 1 - \frac{1}{n!} \right) \log \left( 1 - \frac{1}{n!} \right) + \frac{\log(n!(n!-1))}{n!} \]

\[- \sum_{i=2}^{n} \frac{1}{n!(n!-1)} \log \left( \frac{1}{n!(n!-1)} \right) \]

\[= - \left( 1 - \frac{1}{n!} \right) \log \left( 1 - \frac{1}{n!} \right) + \frac{\log(n!(n!-1))}{n!} \]

\[\approx \frac{1}{n!} + \frac{\log(n!(n!-1))}{n!} \quad \text{by the Taylor series for } (1 - x) \log(1 - x) \]

\[\rightarrow 0 \quad \text{(as } n \rightarrow \infty). \]

61
Even if one ordering is not overwhelmingly likely, we may be able to show that a distribution has low entropy. For example, if the orderings \( A_1, \ldots, A_n \) have respective probabilities \( e^{-\alpha}, e^{-2\alpha}, \ldots, e^{-n\alpha} \), i.e. the orderings have an exponential distribution, then the entropy will be small. Specifically,

\[
- \sum p_i \log p_i = - \sum e^{-\alpha i} \log (e^{-\alpha i}) = (\log_2 e) \alpha \sum i e^{-\alpha i},
\]

and since \( \sum_{i=0}^{\infty} i e^{-\alpha i} \) is a convergent summation, the entropy is a constant, i.e. entropy = \( \Theta(1) \).

(I skipped putting in a normalization factor to assure that the above distribution is actually a probability distribution, but the inclusion of such a factor does not change the result of \( \Theta(1) \) entropy.)

So for this distribution, the lower bound on the number of needed comparisons is a constant and does not grow as \( n \), the number of items, increases. Hence, we expect that by making only a small number of comparisons, we could determine a sorted order and our reported sorted order will be the correct ordering with high probability. That is, we may report the wrong sorted order, but the probability of this reported order being incorrect will be very small (very close to 0).

We should point out that information theory does not always give the best possible lower bounds. For example, as Exercise 6.4 asks you to show, information theory only gives an \( \Omega(\log n) \) lower bound on comparisons needed to find the largest among \( n \) items.

**Exercises**

**Ex 6.1.** Prove a worst case lower bound of \( \frac{3n}{2} - 2 \) comparisons for any algorithm which finds the largest and smallest elements in an array by using only comparisons. Devise a divide-and-conquer algorithm for this problem. Show that your algorithm uses \( \frac{3n}{2} - 2 \) comparisons.

**Ex 6.2.** Create an algorithm which finds the largest and second largest elements in an array and uses only \( n + \log n - 2 \) comparisons. (HINT: Keep track of the elements which are potentially the largest, and for each such element keep track of those elements which have *lost* only to this element.)

**Ex 6.3.** Prove: A rooted binary tree with \( n \) leaves has at least \( n - 1 \) internal nodes with two children.

(HINT: You may want to induct on both the number of leaves and on the total number of nodes.)

**Ex 6.4.** Show that the information theory lower bound for finding the largest among \( n \) items is \( \Omega(\log n) \) comparisons.
Chapter 7

EXHAUSTIVE SEARCH

7.1 Straightforward Exhaustive Search

The algorithms which we have considered so far have been based on the divide-and-conquer strategy, that is, try to break the problem into several smaller problems of the same kind. An exhaustive search is a different strategy for designing algorithms. This strategy is based on the idea of trying all possible answers. Either the solution is located or no solution exists.

A simple problem for which this strategy is reasonable is: given a list, and a value, is there an element of the list which contains the given value? An exhaustive search algorithm would look at each element in turn until either one with the given value is found or until all of the elements have been viewed. If there are \( n \) elements in the list, then this algorithm will take \( \Theta(1) \) in best case and \( \Theta(n) \) in worst case, and these are the best possible running times for any algorithm for this problem.

There are also problems for which straightforward exhaustive search gives very poor algorithms. Consider sorting. A straightforward exhaustive search would try each possible ordering until the sorted ordering is found. Unfortunately, there are \( n! \) possible orderings so in worst case (when the last ordering is the correct one) this algorithm will take at least \( \Theta(n!) \), which is much, much greater than the \( \Theta(n \log n) \) taken by algorithms we have already found.

Satisfiability is another problem to which exhaustive search can be applied. The satisfiability problem is: given a Boolean expression, is there an assignment of true and false to the variables which makes the expression true? If there are \( n \) variables, there are \( 2^n \) possible true/false assignments for the variables, and since a Boolean expression of length \( L \) can be evaluated in \( \Theta(L) \) time, the exhaustive search algorithm will take \( \Theta(L2^n) \) time in worst case. Unlike sorting, it is not clear that there are faster algorithms for this problem. Also, unlike the find value problem, it is not clear that this is the best running time possible for this problem.
7.2 Backtracking

While straightforward exhaustive search may be useful in some problems, there is usually additional information available which will allow an algorithm to eliminate some of the possibilities. One way to make use of this additional information is called backtracking. In a backtrack algorithm the search is broken into stages, and at each stage only the possibilities which can still lead to a solution are considered. The name backtracking comes from a method for finding a way through a maze. At a choice-point in the maze, pick an arbitrary path which has not yet been used. Continue making such choices until you reach a dead-end or a choice-point at which all paths have already been used. Then go back on the path you have come down to the previous choice-point and continue the method. This last step, going back on a path, is called backtracking, and this name is also given to the general method. So, in the general backtracking method

- the search is broken into stages;
- at each stage a choice is made;
- when the algorithm determines that none of the choices at a stage can lead to a solution, then the algorithm backtracks by returning to the previous stage and taking one of the unused choices.

A problem to which backtracking can be directly applied is Hamiltonian path: *given a graph, find a sequence of vertices which contains each vertex exactly once and such that adjacent vertices in the sequence are connected by an edge in the graph.* At each stage in the backtrack algorithm an unused vertex sharing an edge with the current vertex is chosen. If it is impossible to choose such a vertex, the algorithm backtracks to the previous vertex and tries to choose a vertex other than the one that led to the dead-end. If the graph contains a Hamiltonian path, this algorithm will eventually find it, because the algorithm generates all sequences of vertices with adjacent vertices connected by an edge and no repeated vertices. A straightforward exhaustive search for this problem would generate a permutation of the vertices and then test to see if the vertices adjacent in the permutation were also adjacent in the graph. In worst case all permutations would be generated and the exhaustive search algorithm would take at least $\Theta(n!)$ time. The backtrack algorithm in worst case would generate at most $\prod_{i=1}^{n} d_i$ paths, where $d_i$ is the degree of vertex $i$. So in worst case we expect the backtrack algorithm to be faster than the exhaustive search algorithm. The backtrack algorithm also has the advantage that it uses information about the graph in generating a path. If the backtrack algorithm uses more information about the graph in picking the next vertex, we can reasonably expect the backtrack algorithm to work much more quickly in average case.

We will next consider backtracking and exhaustive search for the satisfiability problem. Consider the Booleean expression:

$$X_1 \cdot (X_1 \lor X_2 \lor X_3) \cdot (\overline{X_1} \lor X_2).$$
Since this expression has 3 variables, there are 8 possible truth assignments, which we show in the following table:

<table>
<thead>
<tr>
<th>$X_1$</th>
<th>$X_2$</th>
<th>$X_3$</th>
<th>EXPRESSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

An exhaustive search algorithm would generate 7 out of 8 of these assignments before it found that the expression was satisfiable. A backtrack algorithm could proceed according to the following tree:

```
X_1=F  X_1=T
  /   \
 /     \
F    X_2
    /   |
   /     |
  X_2=T T
```

It picks the first variable $X_1$ and assigns it $F$, then $X_1 = F$ is substituted in the expression yielding the expression $F$. Since $F$ is not satisfiable, the algorithm backtracks and tries the assignment $X_1 = T$. When this is substituted into the expression, it yields the expression $X_2$. A reasonably clever algorithm would notice that this expression can be satisfied by the assignment $X_2 = T$, and hence that the whole expression is satisfiable. A less clever algorithm might try $X_2 = F$, and have to backtrack once more before finding a satisfying assignment. In either case, it seems that the backtrack algorithm will be better than the exhaustive search algorithm.

The backtrack algorithm should simplify the expression at each stage. While this might seem complicated in general, it is very easy when the expression is in clause form. In clause form a Boolean expression is a set of clauses AND-ed together, and within each clause are a set of literals (that is, complemented and uncomplemented variables) OR-ed together. When a variable $X$ is assigned the value $T$, all the clauses containing the literal $X$ become true and can be removed from the expression, while all the clauses containing $\overline{X}$ are simplified by the removal of $\overline{X}$. Similarly, if $X$ is assigned the value $F$, then each clause containing $\overline{X}$ can be removed from the expression and each clause containing $X$ can be simplified by the removal of $X$. One special case should be noted: if a clause containing a single variable has that variable removed, then the whole expression becomes false.

How should the backtrack algorithm choose which variable to assign at each stage? An algorithm could try assigning $X_1$, then $X_2$, and so forth, but this
method does not take advantage of the structure of the expression. If an expression contains a clause with a single literal, then that literal must be set to true for the entire expression to be true. If an expression contains a variable \( X \) which only appears in the form \( X \) and never in the form \( \overline{X} \), then \( X \) can be assigned true and all the clauses containing \( X \) can be removed without affecting the satisfiability of the expression. These two rules can be used to simplify the expression. There is one complication: if the expression contains a clause with the single literal \( X \) and another clause with a single literal \( \overline{X} \), then the expression is not satisfiable.

After the expression has been simplified as much as possible by the above rules, a method for choosing which assignment to make next is still needed. A method which seems reasonable is to pick the literal which appears in the most clauses and to make this literal true. This method is not guaranteed to find a satisfying assignment, but it will simplify the expression. Such a method which is reasonable but not guaranteed to lead to a solution is called a heuristic. A backtrack algorithm will still have to backtrack when the heuristic does not lead to a solution, but a well-chosen heuristic can greatly reduce the number of times the algorithm has to backtrack.

### 7.3 Knight's Tour

We have already mentioned the Hamiltonian path problem. A specialization of this problem is the knight's tour problem. In the knight's tour problem, the vertices of the graph are the squares of an \( n \times m \) chessboard. Two squares are connected by an edge iff a knight can legally move from one square to the other. The knight can move two squares in one direction and one square in an orthogonal direction. The knight's tour problem is to find a Hamiltonian path in this graph, or equivalently move the knight legally around the chessboard so that it visits each square exactly once.

If \( n \cdot m \) is odd then from some starting squares there is no knight's tour. The traditional chessboard has squares of two colors, so that orthogonally adjacent squares are of different colors. If \( n \cdot m \) is odd, then there are more squares of one color. Since the knight always moves from one color to the other, a tour starting from a square of the less numerous color is impossible.

To find a knight's tour one could use the backtrack algorithm for Hamiltonian path, but as the last section suggests, some sort of heuristic should be used to choose the next square. One plausible idea is based on the degree of a square. The degree of a square is the number of unused squares the knight could go to if the knight were on this square. If the square has large degree then there are many ways into and out of the square, and it is not too important to deal with this square immediately. If a square has degree 2, then there is one way in and one way out of the square, and this square should probably be used as soon as possible. So the heuristic is: next visit a possible square of lowest degree.

Even without this heuristic a backtrack algorithm will find a knight's tour if one exists. With this heuristic, the backtrack algorithm may work faster. As
well as picking the next square the degree can also be used in deciding when to backtrack. Clearly a backtrack is called for when the tour reaches a dead end, a square from which no unused square can be reached in one step. Just before the move into this dead end, the dead end square had degree 1. A square of degree 1 should be entered only if it is the last square on the board. So if the lowest degree among possible next squares in 1 and there is more than one unused square, then the algorithm should backtrack. Further, if there are ever two unused squares of degree 1, then the algorithm should backtrack since both of these squares cannot be the last square.

Initially squares near the center of the board will have degree 8, squares nearer an edge of board will have degree less than 8, and the corner squares will have degree 2. As a partial tour is constructed the degrees of the squares will decrease.

How well will this heuristic work? Is backtracking ever needed? We will answer these questions in a moment, but first we want to consider a special case. In the special case \( n = m = 4k + 1 \), for \( k \) a positive integer, and the tour starts in one of the corner squares. For this special case a knight’s tour can be found using a simple rule with no backtracking. The simple rule is: choose as the next square the possible square nearest the edge of the board. The following picture shows a \( 5 \times 5 \) board with each square containing a number indicating the order in which the knight visits the squares.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>14</th>
<th>9</th>
<th>20</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>19</td>
<td>2</td>
<td>15</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>8</td>
<td>25</td>
<td>4</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>23</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>17</td>
<td>22</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Notice that the knight starts in a corner and finishes the tour at the center of the board. It is clear that the simple rule needs some amplification. How did the knight at the square numbered 2 decide to go to the square numbered 3 rather than the square numbered 21 or the square numbered 13, and how did the knight at square 8 decide to go to square 9 rather than square 17? The extra tie-breaking rules are: choose the square with lowest initial degree, choose the first square in clockwise order. With these rules one can prove inductively that a knight’s tour will be found in this special case. The induction considers a \( 4(k + 1) + 1 \times 4(k + 1) + 1 \) board as a \( 4k + 1 \times 4k + 1 \) board surrounded by a
two square border.

From this special case, we can see that in the general case we will also need tie-breaking rules to determine which square should be picked when two possible next squares have the same degree. From the special case, we could try the rule which says: *choose the first such square in clockwise order*. Unfortunately, it can be shown that for some starting squares this tie-breaking rule does not give a knight’s tour.

Another possible tie-breaking rule is to “look-ahead”; choose the next possible square of lowest degree which has the lowest degree following square. But again this tie-breaking rule may fail. Further look-ahead would be possible but it would take considerably more computation.

In spite of the fact that the heuristic even with tie-breaking rules does not work in all cases, it still works in a large majority of cases. Some empirical studies show that a backtracking algorithm using the heuristic uses no backtracks in most cases, one backtrack in a few cases, and 2 backtracks in rare cases, and never uses more than 2 backtracks. A proof of any of these empirical results would be interesting, but it is probably very difficult.

When heuristics work so well on a problem one should consider whether there is a reasonable non-exhaustive algorithm for the problem. Cull & DeCurtins[1978] proved that if \(\min(n, m) \geq 5\), then there is a knight’s tour from at least one starting square on an \(n \times m\) board, and if \(n \cdot m\) is even there is a knight’s tour starting from any square on the board. Their proof technique is constructive and takes \(\Theta(n \cdot m)\) time to construct the knight’s tour. Their paper does not contain a proof that if \(n \cdot m\) is odd then there is a knight’s tour starting from every square of the more numerous color.

Could knight’s tour still be difficult? Yes, if we allow boards with holes. Consider the 9 \(\times\) 9 chessboard. We could remove the center 5 \(\times\) 5 board and leave a 2 square wide border with a large hole in the middle. In spite of this hole, it is easy to start in a corner square and to move the knight so that it visits every square in the border exactly once. On the other hand, if we took the 5 \(\times\) 5 chessboard and removed the center 3 \(\times\) 3 board, the remaining board would not have a knight’s tour. For these two simple the existence of a knight’s tour is still an easy question, but this question becomes more difficult on general board with holes. To make “boards with holes”, we allow any set of squares to be removed from the chessboard. The removed squares do not need to be contiguous or to form one hole. Recently McGown, Leininger, and Cull[2004] showed that the existence of a knight’s tour on a chessboard with holes is an \(NP\)-complete problem. For an explanation of \(NP\)-complete, see the next chapter.

### 7.4 The n-Queens Problem

Backtracking is an effective method for the n-Queens problem. Can \(n\) queens be placed on an \(n \times n\) board so that no queen can take another queen: that is, can \(n\) objects be placed on an \(n \times n\) board so that no two objects are in the same row, the same column, or the same diagonal? The answer to this questions is
“no” when \( n = 2 \) or 3, and “yes” otherwise. The problem becomes more difficult when one is asked for all the different ways to place \( n \) non-taking queens on an \( n \times n \) board. Two ways are different if one cannot be obtained from the other by rotating and/or reflecting the board.

Since one queen will be placed in each row, we will take placing a queen in a row as a stage in a backtracking algorithm. The algorithm starts by placing a queen in column 1 of row 1, then it successively places queens in the lowest numbered allowed column of each successive row, until either every row has a queen or there is no allowed place for a queen in the present row. If there is no allowed place to put a queen in the present row, then the algorithm backtracks and tries to place a queen in the next allowed place in the previous row. If a queen has been placed in each row then this solution is output and the algorithm backtracks. The algorithm terminates when it has backtracked to row 1 and the queen in row 1 is in column \( n \). This algorithm can be written as either a recursive or as an iterative program. Since the algorithm is tail-recursive, we suggest writing it as an iterative algorithm. We leave the actual program as an exercise and refer the reader to Wirth[1976].

The algorithm we have outlined will produce all allowed placements, not all different allowed placements. We still need a method which determines if two placements are different. Two placements are the same if one is the rotation-reflection of the other. The rotation by \( 90^\circ \) and the reflection generate the 8-element group of the symmetries of a square. By applying each element of this group to a placement, one will obtain the 8 placements which are equivalent. If one had a table of the different placements found so far, then one could look in the table and see if any of these 8 equivalent placements had already been found. But there is an easier way. Since a placement will consist of \( n \) numbers indicating the columns in which each of the \( n \) queens are placed, and each column number will be between 1 and \( n \), a solution can be viewed as a single number in base \( n+1 \). The algorithm is set up so that it generates the placements in numeric order. Thus to see if a placement is different, apply to it each of the \( n \) non-identity elements of the group and determine if any of these equivalent placements gives a lower number. If one of these equivalent placements gives a lower number, then this placement is equivalent to an already generated placement. If each of the \( n \) equivalent placements gives a number greater than or equal to the present placement, then the present placement is different from all previous placements.

How fast is this backtrack algorithm for \( n \)-queens? A straightforward exhaustive search would generate all permutations of \( n \) things and hence take about \( n! \) time. Empirical studies suggest that the backtrack algorithm runs in time proportional to \((n/2)^{n/2}\). This is much much smaller than \( n! \). A proof that this algorithm runs in the time suggested by the empirical studies would be very interesting.

**Exercises**

*Ex 7.1.* Use the heuristics of section 6.3 to construct a knight’s tour of a 6 x 6 board.

*Ex 7.2.* Pick starting squares and show that the heuristics of section 6.3 do not
find a knight’s tour of a 5 x 5 board even when such a tour is possible.

**Ex 7.3.** Determine all the different placements of 8 non-taking queens on an 8 x 8 board.
Chapter 8

HARD PROBLEMS

8.1 Classification of Algorithms and Problems

We have encountered various algorithms, particularly of the divide-and-conquer type, which have running times like $\Theta(n)$, $\Theta(n \log n)$, and $\Theta(n^2)$, where $n$ is some measure of the size of the input. We have also encountered algorithms, particularly of the exhaustive search type, which have running times like $\Theta(2^n)$ and $\Theta(n!)$. For algorithms of the first type, doubling the size of the input increases the running time by a constant factor, while for algorithms of the second type, doubling $n$ increases the running time by a factor proportional to the running time. If we double the speed of the computer we are using, then the largest input size which our computer can solve in a given time will increase by a constant factor if we have an algorithm of the first type; for an algorithm of the second type, the input size our computer can solve will only increase by an additive constant, at best. These considerations led Edmonds to propose that algorithms of the first type are computationally reasonable, while algorithms of the second type are computationally unreasonable. More specifically, he suggested the definitions:

- **reasonable algorithm**: an algorithm whose running time is bounded by a polynomial in the size of the input.

- **unreasonable algorithm**: an algorithm whose running time cannot be bounded by any polynomial in the size of the input.

This suggests that we should try to replace unreasonable algorithms by reasonable algorithms. Unfortunately, this goal is not always attainable. For the Towers of Hanoi problem, any algorithm must have running time at least $\Theta(2^n)$. Since some problems do not have reasonable algorithms, we should classify problems as well as algorithms. Corresponding to Edmonds definition for algorithms, Cook and Karp suggested the following definition for problems:

- **easy problem**: a problem which has a polynomial time bounded algorithm.
• **hard problem**: a problem for which there is no polynomial time bounded algorithm.

An easy problem may have unreasonable algorithms. For example, we have seen an \( \Theta(n!) \) exhaustive search algorithm for sorting, but sorting is an easy problem because we also have an \( \Theta(n \log n) \) sorting algorithm. A hard problem, on the other hand, can never have a reasonable algorithm.

Some problems like Towers of Hanoi are hard for the trivial reason that their output is too large. To avoid such output-bound problems, Cook suggested considering only yes/no problems; that is, problems whose output is limited to be either yes or no. One such yes/no problem is the variant of the Towers of Hanoi problem in which the input is a configuration and the question is: is this configuration used in moving the disks from tower A to tower C? Furthermore, this variant is an easy problem. Usual easy problems can be transformed into easy yes/no problems by giving as the instance of the yes/no problem both the input and the output of the usual problem and asking if the output is correct. For example, sorting can be converted into a yes/no problem when we give both the input and the output and ask if the output is the input in sorted order. As another example, matrix multiplication can be converted into a yes/no problem in which we give three matrices \( A, B, \) and \( C \) and ask if \( C = AB \). There are also other ways to transform usual problems into yes/no problems. Examples are the above variant of the Towers of Hanoi, and the variant of sorting in which we ask if the input is in sorted order.

To avoid problems which are hard only because of the length of their output, Cook defined:

• \( P = \) the class of yes/no problems which have polynomial time algorithms.

(Some authors call this class \( PTIME \).) For many problems, it is easy to show that they are in \( P \). One gives an algorithm for the problem and demonstrates a polynomial upper bound on its running time. It may be quite difficult to show that a problem is not in \( P \), but it is clear that there are problems which are not in \( P \). For example, the halting problem, which asks if an algorithm ever terminates when given a particular input, is not in \( P \) because the halting problem has no algorithm, and therefore certainly no polynomial time algorithm.

Classification of problems would not be very useful if we only could say that some problems have polynomial time algorithms and some problems have no algorithms. We would like a finer classification, particularly one that helps classify problems which arise in practice. In the next section we introduce some machinery which is needed for a finer classification.

### 8.2 Nondeterministic Algorithms

Up to this point we have discussed only deterministic algorithms. In a deterministic algorithm there are no choices; the result of an instruction determines which instruction will be executed next. In a nondeterministic algorithm choices
are allowed; any one of a set of instructions may be executed next. Nondeterministic algorithms can be viewed as "magic": if there is a correct choice, the magic forces the nondeterministic algorithm to make this correct choice. A less magic view is that if there are several possibilities the nondeterministic algorithm does all of them in parallel. Since the number of possibilities multiply at each choice-point, there may be arbitrarily many possibilities being executed at once. Therefore, a nondeterministic algorithm gives us unbounded parallelism.

This view of nondeterminism as unbounded parallelism makes clear that nondeterminism does not take us out of the realm of things which can be computed deterministically, because we could build a deterministic algorithm which simulates the nondeterministic algorithm by keeping track of all the possibilities. However, a nondeterministic algorithm may be faster than any deterministic algorithm for the same problem. We define the time taken by a nondeterministic algorithm as the fewest instructions the nondeterministic algorithm needs to execute to reach an answer. (This definition is not precise since what an instruction means is undefined. This could be made precise by choosing a model of computation like the Turing machine in which an instruction has a well-defined meaning, but this imprecise definition should suffice for our purposes.) With this definition of time, a nondeterministic algorithm could sort in \( \Theta(n) \) time because it always makes the right choice, whereas a deterministic algorithm would take at least \( \Theta(n \log n) \) time in some cases. In some sense we are comparing the best case of the nondeterministic algorithm with the worst case of the deterministic algorithm, so it is not surprising that the nondeterministic algorithm is faster.

For yes/no problems we give nondeterministic algorithms even more of an edge. We divide the inputs into yes-instances and no-instances. The yes-instances eventually lead to yes answers. The no-instances always lead to no answers. We assume that our nondeterministic algorithms cannot lead to yes as a result of some choices, and to a no as a result of some other choices. For a yes-instance, the running time of a nondeterministic algorithm is the fewest instructions the algorithm needs to execute to reach a yes answer. The running time of the nondeterministic algorithm is the maximum over all yes-instances of the running time of the algorithm for the yes-instances. We ignore what the nondeterministic algorithm does in no-instances.

As an example of nondeterministic time, consider the yes/no problem: given a set of \( n \) numbers each containing \( \log n \) bits, are there two identical numbers in the set? In a yes-instance, a nondeterministic algorithm could guess which two numbers were identical and then check the bits of the two numbers, so the running time for this nondeterministic algorithm is \( \Theta(\log n) \). On the other hand, even in a yes-instance, a deterministic algorithm would have to look at almost all the bits in worst case, so the running time for any deterministic algorithm is at least \( \Theta(n \log n) \).

The definition of nondeterministic time may seem strange, but it does measure an interesting quantity. If we consider a yes/no problem, the yes-instances are all those objects which have a particular property. The nondeterministic time is the length of a proof that an object has a certain property. We ignore
no-instances because we are not interested in the lengths of proofs that an object does not have the property.

8.3 NP and Reducibility

Now that we have a definition of the time used by a nondeterministic algorithm, we can, in analogy with the class \( P \), define

- \( NP \) = the class of yes/no problems which have polynomial time nondeterministic algorithms.

It is immediate from this definition that

\[ P \subseteq NP \]

because every problem in \( P \) has a deterministic polynomial time algorithm and we can consider a deterministic algorithm as a nondeterministic algorithm which has exactly one choice at each step. It is not clear, however, whether \( P \) is properly contained in \( NP \) or whether \( P \) is equal to \( NP \).

Are there problems in \( NP \) which may not be in \( P \)? Consider the yes/no version of satisfiability: given a Boolean expression, is there an assignment of true and false to the variables which makes the expression true? This problem is in \( NP \) because if there is a satisfying assignment we could guess the assignment, and in time proportional to the length of the expression we could evaluate the expression and show that the expression is true. We discussed exhaustive algorithms for satisfiability because these seem to be the fastest deterministic algorithms for satisfiability. No polynomial time deterministic algorithm for satisfiability is known. Similarly, the problem: given a graph does it contain a Hamiltonian path?, seems to be in \( NP \) but not in \( P \). Hamiltonian path is in \( NP \) because we can guess the Hamiltonian path and quickly (i.e., in polynomial time) check to see if the guessed path really is a Hamiltonian path. Hamiltonian path does not seem to be in \( P \) because exhaustive search algorithms seem to be the fastest deterministic algorithms for this problem and these search algorithms have worst case running times which are at least \( \Theta(2^n) \), and hence their running times cannot be bounded by any polynomial.

Another problem which is in \( NP \) but may not be in \( P \) is composite number: given a positive integer \( n \), is \( n \) the product of two positive integers which are both greater than 1? Clearly this is in \( NP \) because we could guess the two factors, multiply them, and show that their product is \( n \). Why isn’t this problem clearly in \( P \)? Everyone knows the algorithm which has running time at most \( \Theta(n^2) \) and either finds the factors or reports that there are no factors. This well-known algorithm simply tries to divide \( n \) by 2 and by each odd number from 3 to \( n - 1 \). While this algorithm is correct, its running time is not bounded by a polynomial in the size of the input. Since \( n \) can be represented in binary, or in some other base, the size of the input is only \( \log n \) bits, and \( n \) cannot be bounded by any polynomial in \( \log n \). This example points out that we have been
too loose about the meaning of size of input. The official definition says that the size of the input is the number of bits used to represent the input. According to the official definition, the size of the input for this problem is \( \log n \) if \( n \) is represented in binary. But if \( n \) were represented in unary then the size of the input would be \( n \). So the representation of the input can affect the classification of the problem.

There are a great variety of problems which are known to be in \( NP \), but are not known to be in \( P \). Some of these problems may not seem to be in \( NP \) because they are optimization problems rather than yes/no problems. An example of this kind of optimization problem is the traveling salesman problem: given a set of cities and distances between them, what is the length of the shortest circuit which visits each city exactly once and returns to the starting city? This optimization problem can be changed into a yes/no problem by giving an integer \( B \) as part of the input. The yes/no question becomes: is there a circuit which visits each city exactly once and returns to the starting city and has length at most \( B \)? At first glance the optimization problem seems harder than the yes/no problem, because we can solve the yes/no problem by solving the optimization problem, and solving the yes/no problem does not give a solution to the optimization problem. But we can use the yes/no problem to solve the optimization problem.

- Set \( B \) to \( n \) times the largest distance; then the answer to the yes/no problem is yes.
- Set \( STEP \) to \( B/2 \).
- Now set \( B \) to \( B - STEP \), and \( STEP \) to \( STEP/2 \).
- Now do the yes/no problem with this new \( B \) and this new \( STEP \).
- If the answer is yes, set \( B \) to \( B - STEP \) and \( STEP \) to \( STEP/2 \).
- If the answer is no, set \( B \) to \( B + STEP \) and \( STEP \) to \( STEP/2 \).
- Continue this process until \( STEP = 0 \).

The last value of \( B \) will be the solution to the optimization problem. How long will this take? Since \( STEP \) is halved at each call to the yes/no procedure, the number of calls will be the log of the initial value of \( STEP \). But \( STEP \) is initially \( n \) times the largest distance, so the number of calls is \( \log n + \log(\text{largest distance}) \) which is less than the size of the input. Thus the optimization problem can be solved by solving the yes/no problem a number of times which is less than the length of the input.

This example suggests the idea that two problems are equally hard if both of them can be solved in polynomial time if either one of them can be solved in polynomial time. This equivalence relation on problems also suggests a partial ordering of problems. A problem \( A \) is no harder than problem \( B \) if a polynomial time deterministic algorithm for \( B \) can be used to construct a polynomial time deterministic algorithm for \( A \). We symbolize this relation by \( A \leq B \). If \( A \) is the
yes/no traveling salesman problem, and \( B \) is the traveling salesman optimization problem, then we have both \( A \leq B \) and \( B \leq A \). If \( A \) and \( B \) are any two problems in \( P \) then we have both \( A \leq B \) and \( B \leq A \), because we could take the polynomial time deterministic algorithm for one problem and make it a subroutine of the polynomial deterministic algorithm for the other problem and never call the subroutine. While in these examples, the relation \( \leq \) works both ways, there are cases in which \( \leq \) only works one way. For example, let \( A \) be any problem in \( P \), and let \( HALT \) be the halting problem; then \( A \leq HALT \), but \( HALT \not\leq A \), because we don’t need an algorithm for \( HALT \) to construct a polynomial time algorithm for \( A \), and the polynomial time algorithm for \( A \) cannot help in constructing any algorithm for \( HALT \), let alone a polynomial time algorithm for \( HALT \).

This relation \( A \leq B \) which we are calling \( A \) is no harder than \( B \), is usually called polynomial time reducibility, and is read \( A \) is polynomial time reducible to \( B \). There are many other definitions of reducibility in the literature. We refer the interested reader to Garey and Johnson[1979] and Hartley Rogers[1967].

The notion of a partial ordering on problems should aid us in our task of classifying problems. In particular, it may aid us in saying that two problems in \( NP \) are equally hard. Further, it suggests the question: is there a hardest problem in \( NP \)? We consider this question in the next section.

### 8.4 NP-Complete Problems

In this section, we will consider the relation \( \leq \) defined in the last section, and answer the question: is there a hardest problem in \( NP \)? Since we have a partial order \( \leq \), we might think of two very standard instances of partial orders: the partial (and total) ordering of the integers, which has no maximal element; and the partial ordering of subsets, which has a maximal element. From these examples, we see that our question cannot be answered on the basis that we have a partial order. If we consider \( \leq \) applied to problems, is there a hardest problem? The answer is no, because the Cantor diagonal proof always allows us to create harder problems. On the other hand, one may recall that the halting problem is the hardest recursively enumerable (RE) problem. So on the analogy with \( RE \), there may be a hardest problem in \( NP \). Cook[1971] proved that there is a hardest problem in \( NP \). A problem which is the hardest problem in \( NP \) is called an \( NP \)-complete problem. Cook proved the more specific result:

**Theorem 2 (Cook’s Theorem).** Satisfiability is \( NP \)-complete.

The proof of this theorem requires an exact definition of nondeterministic algorithm, and since we have avoided exact definitions, we will only be able to give a sketch of the proof. We refer the interested reader to Garey and Johnson[1979] for the details. The basic idea of the proof is to take any instance of a problem in \( NP \) which consists of a nondeterministic algorithm, a polynomial that gives the bound on the nondeterministic running time, and an input for the algorithm, and to show how to construct a Boolean expression which is
satisfiable iff the nondeterministic algorithm reaches a yes answer within the
number of steps specified by the polynomial applied to the size of the input.
The construction proceeds by creating clauses which can be interpreted to mean
that at step 0 the algorithm is in its proper initial state. Then for each step, a set
of clauses are constructed which can be interpreted to mean that the state of the
algorithm and the contents of the memory are well-defined at this step. Further,
for each step a set of clauses are constructed which can be interpreted to mean
that the state of the algorithm and the contents of memory at this step follow
from the state and contents at the previous step by an allowed instruction of
the algorithm. Finally, some clauses are constructed which can be interpreted
to mean that the algorithm has reached a yes answer. The polynomial time
bound is used to show that the length of this Boolean expression is bounded by
a polynomial in the length of the input.

An interesting consequence of the proof is that satisfiability of Boolean expres-
sions in clause form is NP-complete. This result can be refined to show
that satisfiability in clause form with exactly 3 literals per each clause is also
NP-complete.

After Cook’s result, Karp[1972] quickly showed that a few dozen other stan-
dard problems are NP-complete. Garey and Johnson’s book contains several
hundred NP-complete problems. Johnson also writes a column for the Journal
of Algorithms which contains even more information on NP-complete problems.

Why is this business of NP-complete problems so interesting? The NP-
complete problems are the hardest problems in NP in the sense that for any
problem A in NP, \( A \leq \text{NP-complete} \). So if there were a polynomial time deter-
ministic algorithm for any NP-complete problem, there would be a polynomial
time deterministic problem for any problem in NP; that is, \( P \) and \( NP \) would
be the same class. Conversely, if \( P \neq NP \), then there is no point to looking for
a polynomial time deterministic algorithm for an NP-complete problem. Sim-
ply knowing that a problem is in NP without knowing that it is NP-complete
leaves open the question of whether or not the problem has a polynomial time
bounded algorithm even on the supposition that \( P \neq NP \). The fact that a
number of NP-complete problems have been well known problems for several
hundred years and no one has managed to find a reasonable algorithm for any
one of them suggests to most people that \( P \neq NP \) and that the NP-complete
problems really are hard.

One of the virtues of Cook’s theorem is that to show that an NP problem \( A \)
is NP-complete you only have to show that satisfiability \( \leq A \).

As a catalog of NP-complete problems is built, the task of showing that an
NP problem \( A \) is NP-complete gets easier because you only have to pick some
NP-complete problem \( B \) and show that \( B \leq A \).

We have already mentioned the traveling salesman problem (TSP). This
problem is NP-complete. Let us show that if Hamiltonian circuit is NP-
complete then TSP is NP-complete. The Hamiltonian circuit problem (HC)
is: given a graph, is there a circuit which contains each vertex exactly once and
uses only edges in the graph? Given an instance of HC, we create an instance
of TSP by letting each vertex from HC become a city for TSP, and defining the
distance between cities by \(d(i, j) = 1\) if there is an edge in HC between vertices \(i\) and \(j\), and \(d(i, j) = 2\) if there is no such edge. Now if there were \(n\) vertices in HC, we use \(B = n\) as our bound for TSP. If the answer to TSP is \textit{yes}, then there is a circuit of length \(n\), but this means that the circuit can only contain edges of distance 1 and hence this circuit is also a Hamiltonian circuit of the original graph. Conversely, if there is a Hamiltonian circuit, then there is a TSP circuit of length \(n\). To complete our proof. We must make sure that this transformation from HC to TSP can be accomplished in polynomial time in the size of the instance of HC. Since HC has \(n\) vertices and TSP can be specified by giving the \(n(n-1)/2\) distances between the \(n\) cities, we only have to check for each of the \(n(n-1)/2\) distances whether or not it corresponds to an edge in the original graph. Even with a very simple algorithm this can be done in at worst \(\Theta(n^4)\), which is bounded by a polynomial in the size of the HC instance.

This is a very simple example of proving that one problem in \(NP\) is \(NP\)-complete by reducing a known \(NP\)-complete problem to the problem. We refer the reader to Garey and Johnson[1979] for more complicated examples.

### 8.5 Dealing with Hard Problems

In the theoretical world there are hard problems. Some of these hard problems are \(NP\)-complete problems and there are other problems which are harder than \(NP\)-complete problems. How can these hard problems be handled in the real world?

The simplest way to handle hard problems is to ignore them. Many practical programmers do not know what hard problems are. Their programming involves tasks like billing and payroll which are theoretically trivial, but practically quite important. Ignoring hard problems may be a reasonable strategy for these programmers.

Another way to handle hard problems is to avoid them. To avoid hard problems, you have to know what they are. One of the major virtues of lists of \(NP\)-complete problems is that they help the programmer to identify hard problems and to point out that no reasonable algorithms for these problems are known. It is often unfortunately the case that a programmer is approached with a request for a program and the requester has tried to remove all the specific information about the problem and generalize the problem as much as possible. Over-generalization can make a problem very hard. If the programmer can get the specific information, she may be able to design a reasonable algorithm for the real problem and avoid the hard generalization.

Sometimes real problems are really hard but not too big. For example, many real scheduling problems turn out to be traveling salesman problems with 30 to 50 cities. For these situations an exhaustive algorithm may still solve the problem in reasonable time. The programmer should still try to tune the algorithm to take advantage of any special structure in the problem, and to take advantage of the instructions of the actual computer which will be used. Exhaustive search is a way to handle some \(NP\)-complete problems when the
size of the input is not too large.

Heuristics are another way to deal with hard problems. We have already mentioned heuristics in discussing backtrack algorithms. A heuristic is a method to solve a problem which doesn’t always work. To be useful a heuristic should work quickly when it does work. The use of heuristics is based on the not unreasonable belief that the real world is usually not as complicated as the worst case in the theoretical world. This belief is supported by the observation that creatures which seem to have less computing power than computers can make a reasonable living in the real world. Artificial intelligence has been using heuristics for years to solve problems like satisfiability. These heuristics seem to be very effective on the instances of satisfiability which arise in artificial intelligence contexts. Heuristics are also widely used in the design of operating systems. Occasional failures in these heuristics lead to software crashes. Since we usually see only a couple of such crashes per year these heuristics seem to be very effective.

Sometimes the behavior of heuristics can be quantified so that we can talk about the probability of the heuristic being correct. For example, a heuristic to find the largest element in an $n$ element array is to find the largest element among the first $n−1$ elements. If the elements of the array are in random order, then this heuristic fails with probability $1/n$, and as $n → ∞$ the probability that this heuristic gives the correct answer goes to 1. Such probabilistic algorithms are now being used for a wide variety of hard problems. For example, large primes are needed for cryptographic purposes. While it seems to be hard to discover large primes, there are tests which are used so that if a number passes all the tests, then the number is probably a prime.

Many hard problems can be stated as optimization problems: find the smallest or largest something which has a particular property. While actually finding the optimum may be difficult, it may be much easier to find something which is close to optimum. For example, in designing a computer circuit one would like the circuit with the fewest gates which carries out a particular computation. This optimization problem is hard. But from a practical point of view, no great disaster would occur if you designed a circuit with 10% more gates than the optimum circuit. For various hard problems, approximation algorithms have been produced which produce answers close to the optimum answer. We will consider an example of approximation in the next section.

8.6 Approximation Algorithms

Let us consider the traveling salesman optimization problem: given a set of cities and distances between them, find the shortest circuit which contains each city exactly once. This problem arises in many real scheduling situations. We will try to approximate the shortest circuit.

To make an approximation possible, we will assume that the distances behave like real distances, that is, the distances obey the triangle inequality $d(i, j) ≤ d(i, k) + d(k, j)$, so that the distance from city $i$ to city $j$ is no longer than the
distance from city $i$ to city $k$ plus the distance from city $k$ to city $j$.

A simpler task than finding the minimum circuit is finding the minimum spanning tree. The minimum spanning tree is a set of links which connects all the cities and has the smallest sum of distances. In the minimum spanning tree, the cities are not all directly connected; several links may have to be traversed to get from city $i$ to city $j$. There is a reasonable algorithm for the minimum spanning tree because the shortest link is always in this tree. So one can proceed to find this tree by putting in the shortest link and continuing to add the shortest link which does not complete a cycle.

A circuit can be constructed from the minimum spanning tree by starting at some city and traversing the links of the tree to visit every other city and returning to the starting city. This circuit is twice as long as the sum of the distances of the links in the minimum spanning tree. But this circuit may visit some cities more than once. To "clean up" this circuit, we use this circuit while no city is repeated and, if city $j$ is the first repeated city and if city $i$ is the city before city $j$ and if city $k$ is the next city after city $j$ which has not yet been visited, we connect city $i$ to city $k$. We continue to use this procedure to produce a circuit in which each city is visited exactly once. From the triangle inequality we have that the length of this new circuit is at most as long as the circuit with cities repeated, and hence that this new circuit is no longer than twice the length of the minimum spanning tree.

The optimum circuit must be at least as long as the minimum spanning tree because the optimum circuit connects every city. Thus we have

$$OPT \leq ALG \leq 2OPT$$

where $OPT$ is the length of the optimum circuit and $ALG$ is the length of the circuit produced by the approximation algorithm.

It would be pleasant if all hard optimization problems had approximation algorithms. Unfortunately this is not the case. We really needed the triangle inequality to produce an approximation for the traveling salesman problem. Consider an instance of Hamiltonian circuit with $n$ vertices. We can convert this to an instance of traveling salesman without triangle inequality by assigning distance 1 to all the edges which are in the original graph, and assigning distance $n+2$ to all the edges which were not in the original graph. Now if there were a Hamiltonian circuit in the original graph, then there would be a traveling salesman circuit of length $n$. If we could approximate this traveling salesman problem within a factor of 2, the traveling salesman circuit would have length at most $2n$ exactly when the original graph had a Hamiltonian circuit, because if the traveling salesman circuit used even one of the edges not in the original graph, it would have length at least $2n + 1$. So approximating the traveling salesman problem without triangle inequality is as hard as Hamiltonian circuit.

This example can be generalized to show that no approximation within a factor of $f(n)$ is possible by assigning each edge not in the graph a distance greater than $f(n) - (n-1)$. To make sure the transformation can be carried out in polynomial time in the size of the instance of Hamiltonian circuit, $f(n)$ must be
bounded by $2^{o(n)}$ where $p(n)$ is a polynomial. Thus no reasonable approximation to the traveling salesman problem is possible unless the Hamiltonian circuit problem can be solved quickly; that is, unless $P = NP$.

### 8.7 The World of $NP$

The fact that various $NP$-complete problems have resisted attempts to find reasonable algorithms for them suggests to many people that $P \neq NP$. Even if we accept this belief, there are still other open questions about classes of problems associated with $NP$. The class $NP$ is defined in terms of the *yes*-instances of its problems. A class could also be defined in terms of *no*-instances. In correspondence with $NP$, we define the class $coNP$ as problems whose complements are in $NP$; that is, for each problem in $coNP$ there is a non-deterministic algorithm which has polynomial bounded running time for the *no* instances of the problem.

The notion of $\leq$ we have used before will lump $NP$ and $coNP$ together. To tell them apart we will need a finer notion called *polynomial time many-one reducibility*, and as is traditional in computer science, we will “overload” the symbol $\leq$ to stand for this new partial ordering. This new ordering will be defined on sets, but we will also use the ordering for problems where we ambiguously use the same name for the problem and for the set of *yes*-instances of the problem. For example, we will use $SAT$ to mean the satisfiability problem, and we will also use $SAT$ to mean the set of Boolean formulas which can be satisfied.

With these preliminaries, we define:

$$A \leq B$$

if there exists a polynomial time computable function $f(x)$, so that for every input instance $x$,

$$x \in A \iff f(x) \in B.$$ 

With this ordering it is still true that all (nontrivial) sets in $P$ are mutually equivalent, i.e.

$$\forall A, B \in P \quad A \leq B \quad \text{and} \quad B \leq A.$$ 

On the other hand, sets in $NP$ and $coNP$ do not have to be related. For example, let $SAT$ mean all the Boolean expressions that are true for at least one setting of the variables, and let $coSAT$ mean all the Boolean expressions that are false for all settings of the variables. We have already seen that $SAT \in NP$. Now, notice that $coSAT \in coNP$ because if the answer to the $coSAT$ question is *no*, then there is a setting which makes the expression true (that is, not false) and a non-deterministic algorithm could quickly guess the setting and verify that the setting makes the expression true. While $SAT$ and $coSAT$ seem to be very
closely related, this notion of polynomial time many-one reducibility (\(\leq\)) does not seem to see them as related. Notice that for an expression \(E\), we have

\[
E \in \text{coSAT} \quad \text{iff} \quad E \not\in \text{SAT}.
\]

but our notion of \(\leq\) does not allow \(\not\leq\), it requires that

\[
E \in \text{coSAT} \quad \text{iff} \quad f(E) \in \text{SAT}.
\]

While it is possible to construct this \(f(E)\), the only ways I know would give, at least for some \(E'\)’s, an \(f(E)\) whose length is exponentially larger than the length of \(E\). So, we expect that this notion of \(\leq\) will be able to distinguish \(NP\) from \(\text{coNP}\).

The above definition of \(\text{coNP}\) suggests the questions:

- Does \(NP = \text{coNP}\)?
- Does \(P = NP \cap \text{coNP}\)?

Unfortunately, these questions are unsolved. While the above questions are unsolved, many people believe that the answer to each of these questions is no. The basis for this belief is the analogy between \(NP\) and \(RE\). For \(RE\) the following diagram can be shown to be valid:

If the analogy between \(NP\) and \(RE\) is valid, the world of \(NP\) should look like:
A minor difference in the two diagrams is that $RE \cap coRE$ has the name RECURRENT, but $NP \cap coNP$ has not been assigned a name. If this diagram is correct then there are several types of hard problems which are not $NP$-complete. In particular, there may be problems which are in $NP \cap coNP$ but are not in $P$.

Until recently Composite-Number was a candidate problem for this status. We know that Composite-Number is in $NP$ because we can show that a number is composite by guessing a factor and then proving that it is a factor by dividing which takes polynomial time in the number of digits. The complement of Composite-Number is Primes. While it is not immediately obvious, for every prime number there is a proof that the number is prime and the length of the proof is bounded by a polynomial in the log of the number. So the complement of composite is also in $NP$, and composite is in $NP \cap coNP$. But everyone (including the National Security Agency) assumes that composite and prime do not have reasonable algorithms. Unfortunately, proving that composite/prime is not in $P$ is probably very difficult since this would imply $P \neq NP$.

The status of Primes has recently been clarified by Agrawal et al. (2002). (Interestingly, this result was discovered by one professor working with two undergraduates.) They showed that there is a polynomial time algorithm which determines whether or not a number is a prime. At the moment, their algorithm is not considered fast enough to be a practical algorithm.

How does $PRIMES \in P$ effect cryptographic schemes? As you may know several cryptographic schemes, like RSA, depend on the difficulty of factoring
for their security. The new algorithm does not (at present) effect RSA’s security because the algorithm can show that a number is composite without giving a factor. Even if we know that \( q \) has a factor, we don’t know how to find the factor quickly. Hence, RSA is still secure. (But many people expect that a fast composite algorithm which produces a factor will be found soon. The paranoids assume that NSA has such a factoring algorithm but wants to keep it secret so only they can break RSA coded messages.)

We are now without a reasonable candidate for a problem in \( NP \cap coNP \) but not in \( P \). Each of the problems that were known to be in \( NP \cap coNP \) have each been shown to be in \( P \). For example, Linear-Inequalities was known to be in \( NP \cap coNP \) and it has now been shown to be in \( P \).

**LINEAR INEQUALITIES:**

**INPUT:** A set of inequalities of the form

\[
a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \leq b_i
\]

for \( i = 1 \) through \( m \).

**QUESTION:** Is there a set of rational numbers \( x_1, x_2, \ldots, x_n \) which simultaneously satisfy all \( m \) inequalities?

We conclude by mentioning that our diagram of the world of \( NP \) may be incorrect, but there is some reasonable circumstantial evidence to support it.

## 8.8 Exercises

**Ex 8.1.** Show that if Hamiltonian circuit is \( NP \)-complete then Hamiltonian path is \( NP \)-complete.

**Ex 8.2.** Show that if Hamiltonian path is \( NP \)-complete then Hamiltonian circuit is \( NP \)-complete.

**Ex 8.3.** For the following graph use the minimum spanning tree method to construct a short traveling salesman circuit. Assume that any missing edges have the shortest distance consistent with the triangle inequality. How close to the minimum circuit is your constructed circuit?
Ex 8.4. Show that the following variant of the Towers of Hanoi is in the class $P$. INPUT: A configuration, $CON$, of the Towers of Hanoi puzzle with $n$ disks. QUESTION: Does $CON$ occur in the sequence of configurations used to move $n$ disks from tower $A$ to tower $C$ using the minimal number of moves?

Ex 8.5. Show that the following Graph Isomorphism problem is in the class $NP$. INPUT: Two graphs $G_1$ and $G_2$. QUESTION: Can the vertices of $G_1$ be relabeled so that the relabeled $G_1$ is identical to $G_2$.
Chapter 9

DIFFERENCE EQUATIONS

9.1 First Order Equations

A first order linear constant coefficient difference equation has the form

\[ X_n = \lambda X_{n-1} + f(n) \]

where \( \lambda \) is a constant and \( f(n) \) is a function of \( n \). We would like to find the solution \( X_n \) written as a function of \( n \). If we knew the value, \( X_0 \), of \( X_n \) when \( n = 0 \), we could calculate

\[ X_1 = \lambda X_0 + f(1) \]

then we could compute

\[ X_2 = \lambda X_1 + f(2) = \lambda^2 X_0 + \lambda f(1) + f(2) \]

and

\[ X_3 = \lambda X_2 + f(3) = \lambda^3 X_0 + \lambda^2 f(1) + \lambda f(2) + f(3). \]

We can summarize these computations by

\[ X_n = \lambda^n X_0 + \sum_{i=1}^{n} f(i) \lambda^{n-i} \]

which we will call (ANS). This formula is correct when \( n = 1 \) for then it says

\[ X_1 = \lambda X_0 + \sum_{i=1}^{1} f(i) \lambda^{1-i} = \lambda X_0 + f(1) \]

(since \( \lambda^1 = \lambda \)). Assuming the formula (ANS) is correct for \( n - 1 \) we get
\[ X_n = \lambda \lambda^{n-1} X_0 + \sum_{i=1}^{n-1} f(i) \lambda^{n-1-i} + f(n) \]

verifying by induction that (ANS) is correct.

We now have a solution to the initial value problem for a first order linear constant coefficient difference equation. The initial value problem is: Given \( X_0 \) and a difference equation find \( X_n \) as a function of \( n \). We will have to show that initial value problems for difference equations of this type have only one solution. Then we will be sure that (ANS) gives us the solution not just one of several possible solutions.

### 9.2 Uniqueness Theorem

**Theorem 3.** An initial value problem for a difference equation of the form

\[ X_n = f(X_{n-1}, X_{n-2}, \ldots, X_{n-k}, n) \]

with initial conditions

\[ X_1 = \alpha_1, X_2 = \alpha_2, \ldots, X_k = \alpha_k \]

has a unique solution.

**Proof.** From the formula and the initial values we can compute \( X_{k+1} \) by

\[ X_{k+1} = f(X_k, X_{k-1}, \ldots, X_1, k + l) = f(\alpha_k, \alpha_{k-1}, \ldots, \alpha_1, k + l) \]

So given any \( k \) consecutive values \( X_{n-1}, X_{n-2}, \ldots, X_{n-k} \), we can uniquely compute \( X_n \). Thus by induction, we have that the initial value problem has a unique solution. \( \square \)

REMARK: The one essential ingredient in this proof is the assumption that \( f \) is a function, that is, given an input value (or vector of values) \( f \) returns exactly one value. More general recurrence relations are possible and it may not be possible to put them in the above form and satisfy the assumption that \( f \) is a function.

REMARK: You may notice that the solution of an initial value problem for a linear constant coefficient difference equation is analogous to the solution of an initial value problem for a linear constant coefficient differential equation, with the sum playing the same role for the difference equation as the integral does for the differential equation. But the theory for the difference equation is much simpler. The proof of the Uniqueness Theorem is trivial and no assumptions, other than being defined, have to be placed on the function \( f \) to assure a unique solution.
9.3 \( k^{th} \) Order Equations

If we consider a \( k^{th} \) order linear constant coefficient difference equation, we can write it in the form

\[
X_n + c_1 X_{n-1} + c_2 X_{n-2} + \cdots + c_k X_{n-k} = g(n)
\]

or in the more compact form

\[
L[X] = g(n)
\]

where \( L[X] = X_n + c_1 X_{n-1} + c_2 X_{n-2} + \cdots + c_k X_{n-k} \). We can consider \( L \) to be a linear operator (analogous to a matrix) which operates on functions defined on the natural numbers. These functions, also called sequences, are analogous to vectors on which a matrix would operate.

Recall that when we work with vectors, we define the operation of addition of vectors to be the componentwise sum of the two vectors, i.e., \((X + Y)_i = X_i + Y_i\) where \((X + Y)_i\) is the \( i^{th} \) component of the sum vector and \( X_i \) and \( Y_i \) are respectively the \( i^{th} \) components of the vectors \( X \) and \( Y \). This operation of vector addition can be extended to sequences where we view the sequences as infinite dimensional vectors with a component for each natural number.

As well as vector addition, we also have the operation of scalar multiplication when we are dealing with vectors. That is, we consider the components of the vector \( V \) to be numbers from some field (like the real or complex numbers), and if we take an element, \( a \), called a scalar from the same field, then we can define the scalar product \( a V \) to be the vector whose \( i^{th} \) component is \( a \) times the \( i^{th} \) component of \( V \).

Of course the multiplication of each component by the scalar is carried out in the field we are using. Again this operation of scalar multiplication can be extended to sequences by considering the sequences as infinite dimensional vectors.

Now that we have defined vector addition and scalar multiplication, the formula \( aX + bY \) is well defined where \( X \) and \( Y \) are sequences and \( a \) and \( b \) are scalars. We can now define what we mean by saying that \( L \) is a linear operator. \( L \) is a linear operator iff

\[
L[aX + bY] = aL[X] + bL[Y]
\]

for all sequences \( X \) and \( Y \) and all scalars \( a \) and \( b \).

Returning to the operator \( L \) defined for our linear difference equation we find that if \( X = aY + bZ \) then

\[
L[X] = L[aY + bZ] = (aY_n + bZ_n) + c_1(aY_{n-1} + bZ_{n-1}) + \cdots + c_k(aY_{n-k} + bZ_{n-k})
\]

rearranging the right hand side of the equation using the usual rules for addition and multiplication we find that

\[
L[aY + bZ] = a(Y_n + c_1 Y_{n-1} + \cdots + c_k Y_{n-k}) + b(Z_n + c_1 Z_{n-1} + \cdots + c_k Z_{n-k})
\]

\[
= aL[Y] + bL[Z]
\]
and thus \( L \) is, in fact, a linear operator on the space of sequences.

To understand what we are doing consider a specific operator:

\[
L_1[X] = X_n - X_{n-1}.
\]

If \( X_n = n \) then \( L_1[X] = n - (n - 1) = 1 \). So, we see that the effect of this operator on the function \( n \) is the production of the function which is constantly 1. We also find that \( L_1[1] = 1 - 1 = 0 \) and since \( L_1 \) is a linear operator

\[
L_1[a_1 + bX] = aL_1[1] + bL_1[X] = bL_1[X].
\]

We want to be able to solve the \( k^{th} \) order linear constant coefficient difference equation \( L[x] = g(n) \). Notice that the Uniqueness Theorem we have proved assures us that the initial value problem for these equations has a unique solution.

Let us first consider the \textit{Homogeneous} Equation \( L[X] = 0 \) with initial conditions

\[
X_0 = \alpha_0, \ X_1 = \alpha_1, \ldots, \ X_{k-1} = \alpha_{k-1}.
\]

If we take

\[
X_n + c_1 X_{n-1} + c_2 X_{n-2} + \cdots + c_k X_{n-k} = 0
\]

and replace \( X_i \) by \( \lambda^{i+k-n} \) we get the \textit{characteristic equation} for \( L \):

\[
\lambda^k + c_1 \lambda^{k-1} + \cdots + c_k = 0.
\]

Let \( \lambda_i \) be the \( i^{th} \) root of the characteristic equation, then

\[
X_n = \sum_{i=1}^{k} a_i \lambda_i^n
\]

is a solution of the Homogeneous Equation since

\[
L[\sum_{i=1}^{k} a_i \lambda_i^n] = \sum_{i=1}^{k} a_i L[\lambda_i^n] = \sum_{i=1}^{k} a_i \cdot 0 = 0
\]

because \( L[\lambda_i^n] = 0 \) from the fact that \( \lambda_i \) is a solution to the characteristic equation. Notice that \( \sum_{i=1}^{k} a_i \lambda_i^n \) is a solution for any values we assign to the \( a_i \)'s.

To find \textit{the} values for the \( a_i \)'s we must look at the initial values. Because

\[
X_0 = \sum_{i=1}^{k} a_i \lambda_i^0 = \alpha_0
\]

\[
X_1 = \sum_{i=1}^{k} a_i \lambda_i^1 = \alpha_1
\]
\[
X_{k-1} = \sum_{i=1}^{k} a_i \lambda_i^{k-1} = \alpha_{k-1}
\]

we have a system of \( k \) linear equations in \( k \) unknowns, which we can write in matrix form as

\[
\begin{pmatrix}
1 & 1 & \cdots & 1 \\
\lambda_1 & \lambda_2 & \cdots & \lambda_k \\
\lambda_1^2 & \lambda_2^2 & \cdots & \lambda_k^2 \\
\vdots & \vdots & \ddots & \vdots \\
\lambda_1^{k-1} & \lambda_2^{k-1} & \cdots & \lambda_k^{k-1}
\end{pmatrix}
\begin{pmatrix}
a_1 \\
a_2 \\
\vdots \\
a_k
\end{pmatrix}
= \begin{pmatrix}
\alpha_0 \\
\alpha_1 \\
\vdots \\
\alpha_{k-1}
\end{pmatrix}.
\]

If the \( \lambda \)'s are distinct then the matrix, which we’ll call \( \Lambda \), is non-singular. If the matrix were singular there would be a vector \((b_0, b_1, \ldots, b_{k-1})\) such that \(b\Lambda = 0\). But this would mean that the \((k-1)^{st}\) degree polynomial \(b_0 + b_1 \lambda + \ldots + b_{k-1} \lambda^{k-1}\) would have \(k\) distinct roots, which is impossible. So if the \( \lambda \)'s are distinct we can uniquely solve this system of equations and calculate the \( a_i \)'s.

If two \( \lambda \)'s are identical then two columns of the matrix are identical, the matrix is singular, and the \( \alpha \)'s do not uniquely specify the \( a_i \)'s.

In general the solution is

\[
X_n = \sum_{i=1}^{k} a_i D^{m_i} [\lambda_i^n]
\]

where \( m_i \) is the number of roots with indices less than \( i \) which have the same value as \( \lambda_i \). \( D^{m_i} [\lambda_i^n] \) is the \( m_i^{th} \) derivative, that is, \( n(n-1)(n-2)\ldots(n-1-m_i)\lambda_i^{n-m_i} \).

Using this general solution the matrix for computing the \( a_i \)'s from the \( \alpha_i \)'s is nonsingular. Otherwise there would be a vector \( b \) such that \( b\Lambda = 0 \), but this means there would be a \((k-1)^{st}\) degree polynomial with \( k \) roots counting multiplicity, which is impossible.

Returning to the Nonhomogeneous Equation \( L[x] = g(n) \), we notice that the Uniqueness Theorem holds. So we know that the initial value problem for a Nonhomogeneous Equation has a unique solution.

One method for solving these problems is to find a particular solution \( v \), such that \( L[v] = g(n) \), but \( v \) is not required to satisfy the initial conditions. Let \( h \) be any solution of the Homogeneous Equation, that is, \( L[h] = 0 \). Then \( L[v + h] = L[v] + 0 = g(n) \).

Thus to solve the initial value problem, we pick that \( h \) which causes \( v + h \) to satisfy the initial conditions. We can view \( h \) as the general solution of the Homogeneous Equation. So \( h \) will contain \( k \) unspecified coefficients \( a_1, a_2, \ldots, a_k \). We then solve a system of linear equations for the \( a \)'s. This system of equations is specified by the requirement that \( v + h \) satisfy the initial conditions.

The method we have just described is not an algorithm, since it does not tell us how to find the particular solution \( v \). In general, there is not an algorithm
which will always give us a particular solution in closed form. (There is an algorithm which gives us the particular solution as a summation). But for a very common class of Difference Equations there is an easy method to obtain a particular solution, as the following theorem states.

**Theorem 4.** Consider the Difference Equation \( L[x] = p(n)\lambda^n \) where \( p(n) \) is a polynomial of degree \( r \) and \( \lambda \) is a root of multiplicity \( m \) of the characteristic polynomial of \( L \). This equation has a particular solution of the form \( q(n)\lambda^n \) where \( q(n) \) is a polynomial of degree \( r + m \).

**Remark:** If \( \lambda \) is not a root of the characteristic polynomial of \( L \), then we consider \( \lambda \) to have multiplicity 0. Thus, if \( \lambda \) is not a root of the characteristic polynomial then the Difference Equation has a particular solution of the form \( q(n)\lambda^n \) where \( q(n) \) has the same degree as \( p(n) \).

**Proof.** Consider \( L[D^j\lambda^n] \) where \( D^j \) is the \( j^{th} \) derivative with respect to \( \lambda \). \( L[D^j\lambda^n] = D^jL[\lambda^n] \), since the derivative operator can be moved across sums and constants with respect to \( \lambda \). \( L[\lambda^n] = \lambda^{n-k}ch(\lambda) \), where \( ch(\lambda) \) is the characteristic polynomial of \( L \) and \( k \) is the order of the Difference Equation.

\[
L[D^j\lambda^n] = D^j[\lambda^{n-k}ch(\lambda)] = \begin{cases} 0 & \text{if } m > j \\ q_{j-m}(n)\lambda^{n-k} & \text{if } j \geq m \end{cases}
\]

Since \( ch(\lambda) \) and its first \( m-1 \) derivatives will be 0 if \( \lambda \) has multiplicity \( m \), and since the \( r^{th} \) derivative of \( \lambda^{n-k} \) will give a polynomial of degree \( r \) in \( n \) times \( \lambda^{n-k} \). In the above \( q_{j-m}(n) \) is a polynomial of degree \( j-m \).

As a particular solution of the Difference Equation we consider \( \sum_{j=1}^{k} a_j D^j[\lambda^n] \). From above

\[
L \left[ \sum_{j=1}^{k} a_j D^j[\lambda^n] \right] = \sum_{j=1}^{k} a_j L \left[ D^j[\lambda^n] \right] = \sum_{j=1}^{k} a_j q_{j-m}(n)\lambda^{n-k}
\]

For this to be equal to \( p(n)\lambda^n \) we need

\[
\sum_{j=m}^{J} a_j q_{j-m}(n) = p(n)\lambda^k
\]

Since the right hand side is a polynomial of degree \( r \) the left hand side must have degree \( r \), so \( J - m = r \) and \( J = r + m \).

Finally we notice that \( \sum a_j D^j[\lambda^n] \) can be written in the form \( q(n)\lambda^n \) where \( q(n) \) is a polynomial of degree \( r + m \).

To use this theorem to find a particular solution of \( L[x] = p(n)\lambda^n \) we determine \( r \) the degree of \( p(n) \) and determine \( m \) the multiplicity of \( \lambda \). Then we take a polynomial \( q(n) \) of degree \( r + m \) with unknown coefficients, and substitute \( q(n)\lambda^n \) for \( X \) in the Difference Equation. This will give a system of Linear Equations one for each power of \( n \). We then solve the system to determine the unknown coefficients of \( q(n) \).
If we want to find a particular solution $v$ for the difference equation

$$L[x] = f_1(n) + f_2(n)$$

and we can find $y$ and $z$ such that $L[y] = f_1$, and $L[z] = f_2$ then we can use $v = y + z$ since

$$L[v] = L[y + z] = L[y] + L[z] = f_1 + f_2.$$

**Algorithm for solving linear constant coefficient difference equations**

**INPUT:**

$$x_n + c_1 x_{n-1} + \cdots + c_k x_{n-k} = g(n)$$

$$x_0, x_1, \ldots, x_{k-1}$$

1. Find the roots of the polynomial

$$\lambda^k + c_1 \lambda^{k-1} + \cdots + c_k$$

Call these roots $\lambda_1, \lambda_2, \ldots, \lambda_k$.

2. Find a particular solution $v$ so that

$$v_n + c_1 v_{n-1} + \cdots + c_k v_{n-k} = g(n)$$

$v$ does not have to satisfy the initial conditions. The last theorem in the handout may help in finding $v$.

3. The general solution is

$$x_n = v_n + \sum_{i=1}^{k} a_i D^{m_i} \lambda_i^n$$

where $m_i$ is the number of roots with indices less than $i$ which have the same value as $\lambda_i$. $D^{m_i} \lambda_i^n$ is the $m_i$th derivative, that is

$$n(n-1) \cdots (n+1-m_i) \lambda_i^{n-m_i}$$

4. Find $a_1, a_2, \ldots, a_k$ by solving the set of linear algebraic equations:

$$x_0 - v_0 = \sum_{i=1}^{k} a_i D^{m_i} [\lambda_i^0]$$

$$x_1 - v_1 = \sum_{i=1}^{k} a_i D^{m_i} [\lambda_i^1]$$

$$\vdots$$

$$x_{k-1} - v_{k-1} = \sum_{i=1}^{k} a_i D^{m_i} [\lambda_i^{k-1}]$$

92
Exercises

Ex 9.1. Use the methods described above to solve the following initial value problems.

1. \( x_n = 2x_{n-1} \quad x_0 = 1 \)
2. \( x_n = 2x_{n-1} - 1 \quad x_0 = 0 \)
3. \( x_n = 2x_{n-1} - 1 \quad x_0 = 1 \)
4. \( x_n = 2x_{n-1} - 1 \quad x_0 = 7 \)
5. \( f_n = f_{n-1} + f_{n-2} \quad f_1 = f_2 = 1 \)
6. \( f_n = f_{n-1} + f_{n-2} - 1 \quad f_1 = f_2 = 0 \)
7. \( f_n = f_{n-1} + f_{n-2} - 1 \quad f_1 = f_2 = 1 \)
8. \( f_n = f_{n-1} + f_{n-2} + 1 \quad f_1 = f_2 = 0 \)
9. \( x_n = 2x_{n-1} + n \quad x_0 = 0 \)
10. \( x_n = 2x_{n-1} + n \quad x_0 = -1 \)
11. \( x_n = 2x_{n-1} - n \quad x_0 = 0 \)
12. \( x_n = 2x_{n-1} + 2^n \quad x_1 = 2 \)
13. \( x_n = 2x_{n-1} + 2^n \quad x_0 = 2 \)
14. \( f_{n+1} = f_n + f_{n-1} + 4n - n^2 \quad f_1 = 1 \quad f_2 = 4 \)
15. \( x_n = 2x_{n-1} + n^2 \quad x_1 = 2 \)

16. An AVL tree is a binary tree in which at each vertex the heights of the right and left subtrees of that vertex differ by at most 1. If the AVL tree is of height \( h \) what is the maximum possible number of vertices in the tree? What is the minimum number of vertices? Derive difference equations for the maximum and the minimum number of vertices in an AVL tree. Find appropriate initial conditions for these difference equations. Solve both these initial value problems and thus obtain bounds on the number of vertices in an AVL tree of height \( h \).

9.4 NonNegative Difference Equations

In this section we will consider nonnegative difference equations, that is, equations which can be written in the form

\[ x_n = c_1 x_{n-1} + c_2 x_{n-2} + \cdots + c_k x_{n-k} + g(n) \]

where \( c_i \geq 0 \) for each \( i \) and \( g(n) \geq 0 \). We will discover some facts that make these equations easier to solve than the equations we have dealt with above. In particular, we will give an easy method to determine the asymptotic solution of these nonnegative equations.
9.4.1 Homogeneous Nonnegative Difference Equations

First, we will consider the special case when $g(n) = 0$. As usual, we call these equations homogeneous. (The term, homogeneous, is used because we could replace each occurrence of $x$ by $ax$ and then divide the equation by $a$ to obtain the original equation.) A homogeneous nonnegative difference equation can be written as

$$x_n = c_1 x_{n-1} + c_2 x_{n-2} + \cdots + c_k x_{n-k}$$

where $c_i \geq 0$ for each $i$. We will also assume that our equations are aperiodic in the sense that

$$\gcd\{i \mid c_i > 0\} = 1.$$  

This assumption is usually satisfied by the equations that arise in practice, and the assumption allows the results to be written more simply.

**Theorem 5.** An aperiodic nonnegative difference equation has a solution of the form

$$x_n = \lambda_0^n$$

where $\lambda_0$ is the unique positive real root of $\lambda^k - c_1 \lambda^{k-1} - \cdots - c_k$. Further, for any other root $\lambda_1$ of this polynomial, $\lambda_0 > |\lambda_1|$.

**Theorem 6.** If $x_0, x_1, \ldots, x_{k-1}$ are the initial values for an aperiodic nonnegative difference equation, then the solution is

$$x_n = \Theta(\lambda_0^n)$$

if $x_i \geq 0$ for each $i \in \{0, 1, \ldots, k-1\}$ and for at least one such $i$, $x_i > 0$.

For example, we can apply these theorems to the Fibonacci numbers which have the difference equation

$$f_n = f_{n-1} + f_{n-2}$$

and the initial conditions $f_0 = 0, f_1 = 1$. Here, the coefficients are $c_1 = c_2 = 1$, and the initial conditions have $f_0 \geq 0$ and $f_1 > 0$. So, the theorems say that $f_n = \Theta(\lambda_0^n)$ where $\lambda_0$ is the unique positive real root of $\lambda^2 - \lambda - 1$. For this example, it is easy to calculate that $\lambda_0 = (1 + \sqrt{5}) / 2$, to verify that $\lambda_1 = (1 - \sqrt{5}) / 2$, and to see that $\lambda_0 > |\lambda_1|$. Notice that $f_n$ is always an integer, but $\lambda_0^n$ is an irrational number.

Another simple example is,

$$x_n = x_{n-1} + 2x_{n-2}$$

with $x_0 = 1$, $x_1 = 2$. The solution is $x_n = 2^n$ which is clearly $\Theta(2^n)$.

**Ex 9.2.** Consider this equation $x_n = x_{n-1} + 2x_{n-2}$ with initial conditions $x_0 = 0, x_1 = 15$ and show that the solution is still $\Theta(2^n)$.

**Ex 9.3.** Consider the same equation, $x_n = x_{n-1} + 2x_{n-2}$, with initial conditions $x_0 = -4, x_1 = 7$. Show that $x_n > 0$ if $n \geq 3$ and that $x_n = \Theta(2^n)$ still holds.

**Ex 9.4.** On the other hand, consider the equation $x_n = x_{n-1} + 2x_{n-2}$ with initial conditions, $x_0 = -1, x_1 = 1$, and show that $x_n \neq \Theta(2^n)$. 

94
9.4.2 Equations with Forcing Functions

Next we want to consider nonnegative difference equations with nonnegative forcing functions, that is

\[ x_n = \sum_{i=1}^{k} c_i x_{n-i} + g(n). \]

If we assume that \( g(n) > 0 \) or \( x_i \geq 0 \) for each \( i \in \{0, 1, \ldots, k-1\} \), then it is easy to see that \( x_n > 0 \) for all \( n \geq k \). (This assumption will make the proofs simpler.)

There are three situations we want to consider:

- \( g(n) \) is much less than \( \lambda_0^n \)
- \( g(n) \) is \( \lambda_0^n \) times a polynomial in \( n \)
- \( g(n) \) is much greater than \( \lambda_0^n \).

We will show that the solutions will behave almost as one would expect. So if \( g(n) \) is much less than \( \lambda_0^n \), \( x_n = \Theta(\lambda_0^n) \), and if \( g(n) \) is much greater than \( \lambda_0^n \), then \( x_n = \Theta(g(n)) \). The behavior in the remaining case may be a little unexpected. If \( g(n) = n^d \lambda_0^n \), then \( x_n = \Theta(n^{d+1} \lambda_0^n) \), that is, the response to forcing by a polynomial times \( \lambda_0^n \) is a polynomial of one higher degree times \( \lambda_0^n \). In the most common situation this is, \( g(n) = \lambda_0^n \) and \( x_n = \Theta(n \lambda_0^n) \). Notice that in this situation you don’t see the polynomial, \( g(n) = 1 \cdot \lambda_0^n \) because the polynomial 1 is invisible, that is, not explicitly written down. Of course, 1 is a polynomial of degree 0, and so the response will involve a polynomial of degree 1. Here again there is a problem of invisibility because \( n = n^{0+1} = n^1 \), and we usually don’t write the exponent 1.

To prove these results we will make use of the following lemma about nonnegative polynomials.

**Lemma 1.** Let \( p(\lambda) = \lambda^k - \sum_{i=1}^{k} c_i \lambda^{k-i} \) be the characteristic polynomial for a nonnegative difference equation. Then \( p(\lambda) \) has a single positive real root \( \lambda_0 \), and \( p(\lambda) < 0 \) for all \( \lambda \in (0, \lambda_0) \), and \( p(\lambda) > 0 \) for all \( \lambda > \lambda_0 \).

The next three lemmas will consider the various growth rates for \( g(n) \) and find the asymptotic order for the solution of a nonnegative difference equation with \( g(n) \) as the forcing function.

**Lemma 2.** The nonnegative difference equation

\[ x_n = c_1 x_{n-1} + c_2 x_{n-2} + \cdots + c_k x_{n-k} + g(n) \]

with \( g(n) = O(\lambda_1^n) \) for some \( \lambda_1 \) with \( 0 < \lambda_1 < \lambda_0 \) has the solution \( x_n = \Theta(\lambda_1^n) \).
Proof. Clearly, $x_n \geq \sum_{i=1}^{k} c_i x_{n-i}$ since $g(n) \geq 0$. Since $x_n > 0$ for all $n \geq k$, one can find a $b > 0$, so that $x_n > b \lambda_0^n$ holds for $n \in \{k, \ldots, 2k - 1\}$. Then

$$x_n \geq \sum_{i=1}^{k} c_i x_{n-i} \geq \sum_{i=1}^{k} c_i b \lambda_0^{n-i} \geq b \lambda_0^{n-k} \sum_{i=1}^{k} c_i \lambda_0^{k-i}$$

$$\geq b \lambda_0^{n-k} \lambda_0^k \geq b \lambda_0^n,$$

and by induction, $x_n \geq b \lambda_0^n$ for all $n \geq k$.

For the upper bound,

$$x_n \leq \sum_{i=1}^{k} c_i x_{n-i} + g(n) \leq \sum_{i=1}^{k} c_i x_{n-i} + \beta \lambda_1^n$$

for all big enough $n$. Now assume that

$$x_{n-i} \leq \alpha \lambda_0^{n-i} - \gamma \lambda_1^{n-i}$$

for big enough $n$ and $i$ in $\{1, \ldots, k\}$, then

$$x_n \leq \sum_{i=1}^{k} c_i (\alpha \lambda_0^{n-i} - \gamma \lambda_1^{n-i}) + \beta \lambda_1^n$$

$$\leq \alpha \lambda_0^n - \gamma \sum_{i=1}^{k} c_i \lambda_1^{n-i} + \beta \lambda_1^n$$

$$\leq \alpha \lambda_0^n + \gamma \lambda_1^{n-k} p(\lambda_1) + (\beta - \gamma) \lambda_1^n.$$

So we can choose $\gamma > \beta$ which gives $\beta - \gamma < 0$ and by the lemma on polynomials $p(\lambda_1) < 0$, and thus $x_n \leq \alpha \lambda_0^n$. Putting the upper and lower bounds together gives $x_n \leq \Theta(\lambda_0^n)$.

\[\square\]

Lemma 3. The nonnegative difference equation

$$x_n = c_1 x_{n-1} + c_2 x_{n-2} + \cdots + c_k x_{n-k} + g(n)$$

with $g(n) = \Theta(\lambda_2^n)$ for some $\lambda_2$ with $0 < \lambda_0 < \lambda_2$ has the solution $x_n = \Theta(g(n))$.

Proof. This proof is left as an exercise. \[\square\]

Lemma 4. The nonnegative difference equation

$$x_n = c_1 x_{n-1} + c_2 x_{n-2} + \cdots + c_k x_{n-k} + g(n)$$

with $g(n) = n^d \lambda_0^n$ has the solution $x_n = \Theta(n^{d+1} \lambda_0^n)$.

Proof. To simplify the proof, divide the difference equation by $\lambda_0^n$ to get

$$x_n = \sum_{i=1}^{k} c_i x_{n-i} + n^d$$

$$\lambda_0^n$$

\[96\]
Next define $y_n = x_n / \lambda_0^n$ and $b_i = \alpha i / \lambda_0^i$ to obtain

$$y_n = \sum_{i=1}^{k} b_i y_{n-i} + n^d$$

with the condition that $\sum_{i=1}^{k} b_i = 1$.

For the upper bound, assume that you can find $\alpha$ so that $y_{n-i} \leq \alpha(n - i)^{d+1}$. Obviously, if $n$ is big enough $\alpha$ can be chosen so that this bound holds for $k$ consecutive $y_{n-i}$'s. Then

$$y_n \leq \sum_{i=1}^{k} b_i \alpha(n - i)^{d+1} + n^d$$

$$\leq \alpha(n - 1)^{d+1} \sum_{i=1}^{k} b_i + n^d$$

$$\leq \alpha(n - 1)^{d+1} + n^d$$

because $n - i \leq n - 1$ and $\sum_{i=1}^{k} b_i = 1$. We're left with showing that

$$\alpha(n - 1)^{d+1} + n^d \leq \alpha n^{d+1}$$

or equivalently

$$\alpha(n - 1)^{d+1} \leq \alpha n^{d+1} - n^d.$$  

Factoring the right hand side leaves

$$\alpha(n - 1)^{d+1} \leq n^d(\alpha n - 1)$$

but clearly $(n - 1)^d < (n)^d$ and we only need $\alpha(n - 1) \leq (an - 1)$ and this relation holds for all $\alpha \geq 1$. Notice that for the base of this induction, we had to choose $\alpha$ large enough to give the bound $y_{n-i} \leq \alpha(n - i)^{d+1}$ and now we find that we also need $\alpha \geq 1$. So we can certainly pick $\alpha$ big enough to satisfy both of these conditions.

For the lower bound, assume that we can find a $\beta > 0$ so that $y_{n-i} \geq \beta(n - 1)^{d+1}$ holds for $k$ consecutive $y_{n-i}$'s. Then

$$y_n \geq \sum_{i=1}^{k} b_i \beta(n - i)^{d+1} + n^d$$

$$\geq \beta(n - k)^{d+1} \sum_{i=1}^{k} b_i + n^d$$

$$\geq \beta(n - k)^{d+1} + n^d$$

because $n - i \geq n - k$ and $\sum_{i=1}^{k} b_i = 1$. We are left with showing that

$$\beta(n - k)^{d+1} + n^d \geq \beta(n)^{d+1}$$
or equivalently
\[ n^d \geq \beta[(n)^{d+1} - (n-k)^{d+1}]. \]
The trick here is to rewrite \( n^{d+1} \) as \([ (n-k) + k ]^{d+1} \) and use the binomial expansion. So
\[
n^{d+1} - (n-k)^{d+1} = [(n-k) + k]^{d+1} - (n-k)^{d+1} \\
= \sum_{i=0}^{d+1} (n-k)^{d+1-i} k^i \binom{d+1}{i} - (n-k)^{d+1} \\
= \sum_{i=1}^{d+1} (n-k)^{d+1-i} k^i \binom{d+1}{i} = (n-k)^d \sum_{i=1}^{d+1} (n-k)^{1-i} k^i \binom{d+1}{i} \\
\leq (n-k)^d \sum_{i=0}^{d+1} k^i \binom{d+1}{i} \leq (n-k)^d (k+1)^{d+1}. \]
In this calculation we have changed the limits of the summation at a couple of places. We have also used the fact that \((n-k)^{1-i} \leq 1\) for \(i \geq 1\) and \(n \geq k\).

Now \( n^d \geq \beta(n-k)^d(k+1)^{d+1} \geq \beta[(n)^{d+1} - (n-k)^{d+1}] \) by choosing \( \beta \) small enough. Notice that to start this induction we only had to choose \( \beta \) small enough to give a lower bound on \( y_{n-1} \). So our choices of \( \beta \) are consistent.

Putting the upper and lower bounds together, we have \( y_n = \Theta(n^{d+1}) \), and retranslating this result to \( x_n \), we have \( x_n = \Theta(n^{d+1} \lambda_0^n) \).

Because our difference equations are linear we have the familiar superposition principle, that is, if \( x_n \) is a solution to
\[ x_n = \sum_{i=1}^{k} c_i x_{n-i} + g(n) \]
and \( y_n \) is a solution to
\[ y_n = \sum_{i=1}^{k} c_i y_{n-i} + f(n) \]
then \( z_n \) is a solution to
\[ z_n = \sum_{i=1}^{k} c_i z_{n-i} + g(n) + f(n). \]
Using the superposition principle we can restate the previous lemma as:

**Lemma 5.** The nonnegative difference equation
\[ x_n = c_1 x_{n-1} + c_2 x_{n-2} + \cdots + c_k x_{n-k} + h(n) \lambda_0^n \]
where \( h(n) \) is a polynomial of degree \( d \) in \( n \), and \( \lambda_0 \) is the positive real root of \( \lambda^k - \sum_{i=1}^{k} c_i \lambda^{k-i} \), has the solution \( x_n = \Theta(n^{d+1} \lambda_0^n) \).
The special case when the polynomial has degree zero arises so often in practice, that we restate this special case.

**Lemma 6.** The nonnegative difference equation

\[ x_n = c_1 x_{n-1} + c_2 x_{n-2} + \cdots + c_k x_{n-k} + b \lambda_0^n \]

where \( \lambda_0 \) is the positive real root of \( \lambda^k - \sum_{i=1}^k c_i \lambda^{k-i} \), has the solution \( x_n = \Theta(n \lambda_0^n) \).

### 9.5 Homogeneous: Periodic and Primitive

*Homogeneous* nonnegative recurrences, which can be written in the form:

\[ \text{(HNN)} \quad s_n = c_1 s_{n-1} + c_2 s_{n-2} + \cdots + c_k s_{n-k} \quad \text{where all } c_i \geq 0 \text{ and } c_k \neq 0. \]

Since the characteristic polynomial of this equation is a nonnegative polynomial, there is a dominant eigenvalue which we call \( \lambda_0 \). Let \( g = \gcd\{i|c_i > 0\} \). If \( g = 1 \), the equation is *primitive* and for all other eigenvalues \( \lambda_j, \lambda_0 > |\lambda_j| \). If \( g > 1 \), the equation is *periodic* and has \( g \) eigenvalues \( \lambda_0, \lambda_1, \ldots, \lambda_{g-1} \), all with absolute value equal to \( |\lambda_0| \).

An example of a simple *periodic* nonnegative difference equation is

\[ s_n = 4s_{n-2} \]

which has periodic characteristic polynomial \( ch(x) = x^2 - 4 \) and dominant root \( \lambda_0 = 2 \). With initial conditions \( s_0 = 1 = s_1 \) the solution is

\[ s_n = 4^{n/2}, \]

and \( s_n = \Theta(2^n) = \Theta(\lambda_0^n) \). When the initial conditions are \( s_0 = 1, s_1 = 0 \) the solution is

\[ s_n = \begin{cases} 4^{n/2} & \text{when } n \text{ is even} \\ 0 & \text{when } n \text{ is odd} \end{cases} \]

and \( s_n = O(2^n) \) but \( s_n \neq \Theta(\lambda_0^n) \) since every odd position in the sequence is 0.

The next theorem and its corollary elucidate the difference between periodic and primitive equations.

**Theorem 7.** Let \( \lambda_0 \) be the dominant eigenvalue of the homogeneous nonnegative recurrence (HNN) with nonnegative initial values \( s_0, \ldots, s_{k-1} \). If \( \langle s_n \rangle \) is the solution then \( s_n = O(\lambda_0^n) \). If in addition there are \( k \) consecutive elements in \( \langle s_n \rangle \) which are positive, then \( s_n = \Theta(\lambda_0^n) \).

For example, the Fibonacci recurrence is a homogeneous nonnegative recurrence in which all terms after the first are positive. Therefore this result says that \( f_n = \Theta(\lambda_0^n) \) where \( \lambda_0 = \frac{1 + \sqrt{5}}{2} \) is the unique positive eigenvalue. Another simple example is

\[ s_n = s_{n-1} + 2s_{n-2} \quad \text{with } s_0 = 1, s_1 = 2. \]

Since \( ch(x) = x^2 - x - 2 = (x + 1)(x - 2) \) and the first two conditions are positive, the solution is \( \Theta(2^n) \).
Corollary 1. Let \( \langle s_n \rangle \) be the solution to a homogeneous nonnegative recurrence with nonnegative initial conditions where not all initial conditions are zero. If the characteristic polynomial is primitive then \( \langle s_n \rangle \) is eventually positive and \( s_n = \Theta(\lambda_0^n) \).

We should note that a periodic equation whose index of imprimitivity is \( g > 1 \) (such as \( s_n = 4s_{n-2} \), the example at the beginning of this section with \( g = 2 \)) can be expressed as a system of \( g \) primitive equations because

\[
s_n = c_g s_{n-g} + c_{2g} s_{n-2g} + \ldots + c_{rg} s_{n-rg}
\]

can be written as the primitive system

\[
i_n^{(0)} = c_g i_{n-1}^{(0)} + c_{2g} i_{n-2}^{(0)} + \ldots + c_{rg} i_{n-r}^{(0)}
\]

\[
i_n^{(1)} = c_g i_{n-1}^{(1)} + c_{2g} i_{n-2}^{(1)} + \ldots + c_{rg} i_{n-r}^{(1)}
\]

\[
\vdots
\]

\[
i_n^{(g-1)} = c_g i_{n-1}^{(g-1)} + c_{2g} i_{n-2}^{(g-1)} + \ldots + c_{rg} i_{n-r}^{(g-1)}
\]

where the initial conditions for each \( \langle i_n^{(j)} \rangle \) are the initial conditions of \( \langle s_n \rangle \) whose subscripts equal \( j \) (mod \( g \)). If all initial conditions for a particular \( j \) are zero, the sequence \( \langle i_n^{(j)} \rangle \) is the zero sequence. Otherwise, at least one initial condition for \( \langle i_n^{(j)} \rangle \) is positive and by the corollary \( i_n^{(j)} = \Theta(\lambda_0^n) \). Translating this back to the original sequence \( \langle s_n \rangle \), we see that for a fixed \( j \) the subsequence whose subscripts satisfy the arithmetic progression \( n \equiv j \pmod{g} \) can either be the zero sequence or it will grow like \( \lambda_0^n \). In some sense we can therefore regard \( \langle s_n \rangle \) as a periodic sequence with period \( g \), or under some special circumstances the period of \( \langle s_n \rangle \) may be a divisor of \( g \).

9.6 Nonhomogeneous Nonnegative Equations

In this section we look at nonhomogeneous nonnegative equations which have one of three types of non-zero inputs.

Theorem 8. Consider a nonnegative difference equation

\[
s_n = c_1 s_{n-1} + c_2 s_{n-2} + \ldots + c_k s_{n-k} + g(n)
\]

where \( g \) is a function such that the value of \( g(n) \) is nonnegative for each natural number \( n \). Let \( \lambda_0 \) be the dominant eigenvalue of the recurrence. If \( s_n > 0 \) for all sufficiently large \( n \), then

- \( s_n = \Theta(\lambda_0^n) \) when \( g(n) = O(\lambda_0^n) \) for some positive \( \lambda_1 < \lambda_0 \);
- \( s_n = \Theta(g(n)) \) when \( g(n) = \Theta(\lambda_0^n) \) for some \( \lambda_2 > \lambda_0 \);
• $s_n = \Theta(n^{d+1}\lambda^n_0)$ when $g(n) = f(n)\lambda^n_0$ for a polynomial $f$ with $\deg(f) = d$.

These three types of nonnegative forcing functions respectively correspond to the situations when $g(n)$ is much less than $\lambda^n_0$, much greater than $\lambda^n_0$, and equal to $\lambda^n_0$ times a polynomial in $n$. For the last two types of forcing functions the hypothesis requiring $s_n$ to be eventually positive is superfluous. In the first type, the required eventual positivity follows once there are $k$ consecutive positive terms, and that can result from positive initial conditions or positive $g(n)$ or some combination of these forms of positiveness.

Notice that the solutions asymptotically behave about as one would expect, except that the behavior in the third case may be a little unexpected. In that case, the response to forcing by a polynomial times $\lambda^n_0$ results in a polynomial of one higher degree times $\lambda^n_0$. The most common occurrence of this is with $g(n) = \lambda^n_0$, and then $s_n = \Theta(n\lambda^n_0)$.

The following special case arises often in practice.

**Corollary 2.** For any positive constant $b$, any nonnegative difference equation of the form

$$s_n = c_1 s_{n-1} + c_2 s_{n-2} + \cdots + c_k s_{n-k} + b \lambda^n_0$$

has solution $s_n = \Theta(n\lambda^n_0)$ where $\lambda_0$ is the dominant eigenvalue.

There are many variations of the results in Theorem 8. Because these variations are proved in essentially the same way as the results in the theorem, we simply state them here and leave their proofs as exercises.

- If $g(n) = O(\lambda^n_0)$ for $\lambda_1 < \lambda_0$ then $s_n = O(\lambda^n_0)$.
  - (a) If the difference equation is primitive and $g(n) = O(\lambda^n_1)$ for $\lambda_1 < \lambda_0$ then $s_n = \Theta(\lambda^n_1)$.
  - (b) If $s_n > 0$ for $k$ consecutive values of $n$ and $g(n) = O(\lambda^n_1)$ for $\lambda_1 < \lambda_0$ then $s_n = \Theta(\lambda^n_1)$.

- Let $G(n) = \max\{g(n), \lambda_2 g(n-1), \ldots, \lambda_2^{n-1} g(1)\}$. If $g(n) = \Omega(\lambda^n_2)$ for $\lambda_2 > \lambda_0$ then
  - (a) $s_n = \Omega(\lambda^n_2)$.
  - (b) $s_n = \Omega(g(n))$.
  - (c) $s_n = O(G(n))$.
  - (d) If $G(n) = O(g(n))$ then $s_n = \Theta(g(n))$.
  - (e) If $g(n) = \Omega(G(n))$ then $s_n = \Theta(G(n))$.

- If $g(n) = h(n)\lambda^n_0$ for some nonnegative $h(n)$ then $s_n = O(n g(n))$ and $s_n = \Omega(g(n))$.

- If $g(n) = h(n)\lambda^n_0$ for some nonnegative $h(n)$ which is nondecreasing then $s_n = \Theta(\sum_{j=1}^n \lambda_0^{n-j} g(j))$.
9.7 Summary of Results

Here we briefly re-cap our results on difference equations in an easy to look up form.

9.7.1 Homogeneous Equations

Asymptotic Behavior for Homogeneous Equations

1. Dominant Eigenvalue – Multiplicity 1

If $\lambda_1$ is larger than every other eigenvalue and $\lambda_1$ is a simple root of $ch(\lambda)$ then

$$\lim_{n \to \infty} \frac{x_n}{\lambda_1^n} = \text{constant.}$$

The constant may be 0. In other words,

$$|x_n| = O(|\lambda_1|^n).$$

2. Dominant Eigenvalue – Multiplicity $d + 1$

If $\lambda_1$ is larger than every other eigenvalue and $\lambda_1$ has multiplicity $d + 1$ as a root of $ch(\lambda)$ then

$$\lim_{n \to \infty} \frac{x_n}{n^d \lambda_1^n} = \text{constant.}$$

The constant may be 0. In other words

$$|x_n| = O(n^d |\lambda_1|^n).$$

3. Several Dominant Eigenvalues – Multiplicity 1

If there are several simple eigenvalues, $\lambda_1, \lambda_2, \ldots, \lambda_j$ all with largest magnitude then

$$|x_n| = O(|\lambda_1|^n).$$

(Unlike in (1) and (2) limits may not exist.)

4. Several Dominant Eigenvalues – Several Multiplicities

If there are several eigenvalues, $\lambda_1, \lambda_2, \ldots, \lambda_j$ all with largest magnitude,
and $d = \max\{d_1, d_2, \ldots, d_j\}$ where $d_i + 1$ is the multiplicity of $\lambda_i$ as a root of $ch(\lambda)$,

$$|x_n| = \mathcal{O}(n^d |\lambda_1|^n).$$

(Again, limits may not exist.)

In each of these cases, we see that the $k$-dimensional solutions asymptotically become 1-dimensional. So, some degree of long-range prediction of upper bounds for solutions of linear difference equations is possible. If a few extra conditions are satisfied, then the limits can be shown to be non-zero and the $\mathcal{O}$'s can be replaced by $\Theta$'s, that is, the long range predictions become exact orders of growth rather than just upper bounds.
9.7.2 Nonnegative Difference Equations

1. Nonnegative Problem

(a) Difference Equation

\[ x_n = c_1 x_{n-1} + c_2 x_{n-2} + \ldots + c_k x_{n-k} + g(n). \]

with \( c_1 \geq 0, c_2 \geq 0, \ldots, c_{k-1} \geq 0, \text{ and } c_k > 0, \text{ and } g(n) \geq 0. \)

(b) Positive Initial Conditions

\[ x_1 > 0, \ x_2 > 0, \ldots \ x_k > 0. \]

(c) \( \lambda_0 \) is the unique positive real root of

\[ \lambda^k - c_1 \lambda^{k-1} - c_2 \lambda^{k-2} - \ldots - c_k = 0. \]

2. Homogeneous \((g(n) \equiv 0)\)

(a) then \( x_n = \Theta(\lambda_0^n). \)

3. Non-Homogeneous \((g(n) > 0)\)

(a) If \( g(n) = O(\gamma^n) \) with \( \gamma < \lambda_0 \)

then \( x_n = \Theta(\lambda_0^n). \)

(b) i. If \( g(n) = \lambda_0^n \) then \( x_n = \Theta(n \lambda_0^n). \)

ii. If \( g(n) = n^d \lambda_0^n \) then \( x_n = \Theta(n^{d+1} \lambda_0^n). \)

(c) If \( g(n) = \Theta(\gamma^n) \) with \( \gamma > \lambda_0 \)

then \( x_n = \Theta(\gamma^n). \)
Chapter 10

Worked Examples

Here we consider $k^{th}$-order linear recurrences, that is, equations of the form

$$(L) \quad s_n - c_1 s_{n-1} - c_2 s_{n-2} - \cdots - c_k s_{n-k} = \psi(n) \quad \text{for } n \geq k,$$

where $c_i$ are complex scalars with $c_k \neq 0$, and $\psi$ is a complex-valued function. We can find a nice solution for the special cases in which the forcing function has the form

$$\psi(n) = \lambda^n p(n),$$

where $\lambda$ is a fixed scalar and $p$ is a polynomial with complex coefficients. The initial value problems we consider here have this form. Throughout, $\lambda_1, \ldots, \lambda_t$ are the different eigenvalues of the recurrence.

10.1 All Simple Roots

All examples in this section are second-order linear equations whose characteristic polynomial is

$$ch(x) = x^2 - x - 6 = (x - 3)(x + 2),$$

which has the simple roots $\lambda_1 = 3$, $\lambda_2 = -2$, and the general solution of the homogeneous system has the form $s_n = a_1 3^n + a_2 (-2)^n$ for some $a_1, a_2 \in \mathbb{C}$, and for any $\psi$ the equation

$$s_n = s_{n-1} + 6s_{n-2} + \psi(n)$$

has the general solution

$$s_n = a_1 3^n + a_2 (-2)^n + v_n,$$

for some particular solution $v_n$. The constants $a_1, a_2$ are determined by the initial conditions.
To find a particular solution to the nonhomogeneous equation

\[ s_n = s_{n-1} + 6s_{n-2} + \lambda^n p(n), \]

where \( p \) is a non-zero polynomial and \( \lambda \) is a scalar, we use the following.

---

**Solving (L) when there are no multiple roots**

When \( \text{ch}(x) \) has no multiple roots, then \( \langle s_n \rangle \) is a solution to the equation

\[ s_n - \alpha_1 s_{n-1} - \ldots - \alpha_k s_{n-k} = \lambda^n p(n), \]

if and only if

\[ s_n = \sum_{i=1}^{k} a_i \lambda_i^n + \lambda^n n^\delta q(n), \]

where \( a_1, \ldots, a_k \in \mathbb{C}, q(x) \) is a polynomial with \( \text{deg}(q) = \text{deg}(p) \) and

\[ \delta = \begin{cases} 0 & \text{if } \lambda \notin \{\lambda_1, \ldots, \lambda_k\} \\ 1 & \text{if } \lambda \in \{\lambda_1, \ldots, \lambda_k\} \end{cases}. \]

A particular solution is \( \lambda^n n^\delta q(n) \).

---

Let’s first analyze the homogeneous initial value problem.

**Example 10.1.1.** For any initial value problem with equation

\[ s_n = s_{n-1} + 6s_{n-2} \]

we have \( s_n = a_1 3^n + a_2 (-2)^n \), and the initial conditions \( s_0, s_1 \) give

\[ s_0 = a_1 + a_2 \quad \text{and} \quad s_1 = 3a_1 - 2a_2. \]

Multiplying the first equation by 3 and subtracting that from the second equation, we obtain

\[ s_1 - 3s_0 = -5a_2 \quad \text{and} \quad a_2 = \frac{3s_0 - s_1}{5}. \]

Inserting this value for \( a_2 \) into the first equation of (10.4), we have

\[ a_1 = s_0 - \frac{3s_0 - s_1}{5} = \frac{2s_0 + s_1}{5}, \]

and the solution to the homogeneous initial value problem is

\[ s_n = \frac{2s_0 + s_1}{5} 3^n + \frac{3s_0 - s_1}{5} (-2)^n. \]
Example 10.1.2. Consider the second-order equation

\[ s_n = s_{n-1} + 6s_{n-2} + 2^n, \]

which has \( \text{deg}(p) = 0 \) and \( \lambda = 2 \notin \{3, -2\} \), and so \( \text{deg}(q) = 0 \) and \( \delta = 0 \). The above formula gives the particular solution \( v_n = c2^n \), and the recurrence can be used to solve for \( c \):

\[ v_{n-1} + 6v_{n-2} + 2^n - v_n = c2^{n-1} + 6c2^{n-2} + 2^n - c2^n = 2^{n-1}(c + 3c + 2 - 2c). \]

Therefore, \( 2c + 2 = 0 \), which gives the particular solution \( v_n = -2^n \), and

\[ (10.5) \quad s_n = a_13^n + a_2(-2)^n - 2^n \text{ for some } a_1, a_2 \in \mathbb{C}. \]

We can write this in terms of the initial values \( s_0, s_1 \), namely,

\[ (10.6) \quad s_0 = a_1 + a_2 - 1 \text{ and } s_1 = 3a_1 - 2a_2 - 2, \]

which can be solved for \( a_1, a_2 \) as in the previous example. Multiplying the first equation by 2 and adding it to the second, we have

\[ 2s_0 + s_1 = 5a_1 - 4 \text{ and } a_1 = \frac{2s_0 + s_1 + 4}{5}. \]

Substituting this value of \( a_1 \) into the first equation of (10.6), we obtain

\[ a_2 = s_0 - a_1 + 1 = s_0 - \frac{2s_0 + s_1 + 4}{5} + 1 = \frac{3s_0 - s_1 + 1}{5}, \]

and (10.5) becomes

\[ s_n = \frac{2s_0 + s_1 + 4}{5}3^n + \frac{3s_0 - s_1 + 1}{5}(-2)^n - 2^n. \]

To satisfy ourselves that this has been done correctly, let us use the last expression to verify that

\[ s_2 = \frac{2s_0 + s_1 + 4}{5}9 + \frac{3s_0 - s_1 + 1}{5}4 - 4 = s_1 + 6s_0 + 4, \]

which is consistent with the recurrence.

Example 10.1.3. For the second-order equation

\[ (10.7) \quad s_n = s_{n-1} + 6s_{n-2} + n2^n, \]

let us first check that there is no constant \( c \) for which \( w_n = c2^n \) is a particular solution. We have

\[ w_{n-1} + 6w_{n-2} + n2^n - w_n = c2^{n-1} + 6c2^{n-2} + n2^n - c2^n \]

\[ = 2^{n-1}(c + 3c + 2n - 2c) = 2^n(c + n), \]

107
which cannot be zero for all \( n \) when \( c \) is a constant. Rather, since \( \text{deg}(p) = 1 \) and \( \lambda = 2 \notin \{3, -2\} \), the formula says that there exists a particular solution of the form \( v_n = (an + b)2^n \) for some constants \( a, b \). The recurrence gives

\[
v_{n-1} + 6v_{n-2} + n2^n - v_n
= (an - a + b)2^{n-1} + 6(an - 2a + b)2^{n-2} + n2^n - (an + b)2^n
= 2^{n-1}((an - a + b) + 3(an - 2a + b) + n - 2(an + b))
= 2^{n-1}((2a + 2)n + (-7a + 2b)),
\]

which must equal zero for all \( n \); that is, \( 2a + 2 = 0 \) and \( -7a + 2b = 0 \). This gives \( a = -1, 2b = 7a = -7 \), and \( v_n = -(2n + 7)2^{n-1} \) is a particular solution of (10.7), and the general solution has the form

\[
s_n = a_13^n + a_2(-2)^n - (2n + 7)2^{n-1}, \text{ where } a_1, a_2 \text{ are constants in } \mathbb{C}.
\]

For initial values \( s_0, s_1 \), we obtain

\[
s_0 = a_1 + a_2 - 7/2 \text{ and } s_1 = 3a_1 - 2a_2 - 9.
\]

Simultaneously solving this system of equations yields

\[
a_1 = \frac{2s_0 + s_1 + 16}{5} \quad \text{and} \quad a_2 = \frac{6s_0 - 2s_1 + 3}{10},
\]

which gives

\[
s_n = \frac{2s_0 + s_1 + 16}{5}3^n - \frac{6s_0 - 2s_1 + 3}{5}(2)^{n-1} - (2n + 7)2^{n-1}.
\]

Verifying this calculation for \( n = 2 \) shows that

\[
s_2 = \frac{18s_0 + 9s_1 + 144}{5} + \frac{12s_0 - 4s_1 + 6}{5} - 22 = s_1 + 6s_0 + 8,
\]

as required.

**Example 10.1.4.** For

\[
s_n = s_{n-1} + 6s_{n-2} + 3^n,
\]

\( \text{deg}(p) = 0 \) and \( \lambda = \lambda_1 \), which means that \( \delta = 1 \), and the formula gives \( v_n = 3^n b_n \) for some \( b \in \mathbb{C} \). (Note that \( w_n = c3^n \) is a solution to the homogeneous equation and so cannot be a particular solution here. Also, \( v_n = 3^n(bn + a) \) could be used here, but the constant term can be absorbed into the earlier coefficient of \( 3^n \).) To find \( b \), use

\[
v_{n-1} + 6v_{n-2} + 3^n - v_n = 3^{n-1}b(n - 1 + 2n - 4 - 3n) + 3^n = 3^{n-1}(-5b + 3),
\]

and from this, \( b = 3/5 \) and \( v_n = n3^{n+1}/5 \). Therefore,

\[
s_n = \frac{a_13^n + a_2(-2)^n + n3^{n+1}}{5},
\]

108
and \( s_0 = a_1 + a_2, \ s_1 = 3a_1 - 2a_2 + 9/5 \) gives

\[
a_1 = \frac{2s_0 + s_1 - 9/5}{5} \quad \text{and} \quad a_2 = \frac{3s_0 - s_1 + 9/5}{5}.
\]

Hence,

\[
s_n = \frac{1}{5} \left( (2s_0 + s_1 - 9/5 + 3n)3^n + (3s_0 - s_1 + 9/5)(-2)^n \right).
\]

Example 10.1.5. For

\[ s_n = s_{n-1} + 6s_{n-2} + n(-2)^n, \]

\(\deg(p) = 1 \) and \( \lambda = \lambda_2 \), from which we obtain \( \delta = 1 \) and \( v_n = (-2)^n q(n) \), where \( q(n) = cn^2 + bn \) for some \( b, c \in \mathbb{C} \). Then

\[
v_{n-1} + 6v_{n-2} + n(-2)^n - v_n
\]

\[ = (-2)^{n-1} \left( q(n-1) - 3q(n-2) - 2n + 2q(n) \right) \]

equals zero for all integers \( n \geq 2 \). In particular, from \( n = 2 \) and \( n = 3 \) we obtain

\[
q(1) - 3q(0) + 2q(2) = 4,
\]

\[
q(2) - 3q(1) + 2q(3) = 6,
\]

which give \( 5b + 9c = 4 \) and \( 5b + 19c = 6 \), and \( c = 1/5, \ b = 11/25 \). Therefore, \( q(n) = \frac{5n^2 + 11n}{25} \) and

\[
s_n = a_1 3^n + \left( a_2 + \frac{5n^2 + 11n}{25} \right)(-2)^n.
\]

Substituting for \( n = 0 \) and \( n = 1 \), we have

\[
s_0 = a_1 + a_2 \quad \text{and} \quad s_1 = 3a_1 - 2a_2 - 32/25,
\]

and from this we obtain

\[
a_1 = \frac{2s_0 + s_1 + 32/25}{5} \quad \text{and} \quad a_2 = \frac{3s_0 - s_1 - 32/25}{5}.
\]
10.2 One Multiple Root

Solving (L) when there is a single eigenvalue

When there is a single eigenvalue, \( \lambda_1 \), it has multiplicity \( k \), and the rule for particular solutions simplifies to

\[ v_n = \lambda^n n^k q(n) \]

is a particular solution

where \( q \) is a polynomial with \( \deg(q) = \deg(p) \) and

\[
\delta = \begin{cases} 
0 & \text{if } \lambda \neq \lambda_1, \\
1 & \text{if } \lambda = \lambda_1.
\end{cases}
\]

If \( \lambda \neq \lambda_1 \), the solution is

\[ s_n = \lambda^n q_1(n) + \lambda^n q(n) \]

where \( \deg(q) = \deg(p) \), \( \deg(q_1) \leq k \) and the coefficients of \( q_1 \)
are determined from the initial conditions.

If \( \lambda = \lambda_1 \), the solution is

\[ s_n = \lambda^n q_1(n) + \lambda^n n^k q(n) = \lambda^n Q(n) \]

and \( \deg(Q) = k + \deg(p) \).

The five examples in this section are second-order equations, and each has the characteristic polynomial

\[ ch(x) = x^2 - 4x + 4 = (x - 2)^2. \]

This has the double root \( \lambda_1 = 2 \), which means that the general solution is

\[ s_n = (a_1 n + a_2) 2^n + v_n, \]

where \( v_n \) is a particular solution. As in the last section, \( v_n \) depends on \( \psi \), and \( a_1, a_2 \) can be calculated from the initial conditions.

Example 10.2.1. For any initial value problem with recurrence

\[ s_n = 4s_{n-1} - 4s_{n-2}, \]

\[ s_n = (a_1 n + a_2) 2^n, \]

where the constants \( a_1 = (s_1 - 2s_0)/2 \) and \( a_2 = s_0 \) can be computed from the initial conditions \( s_0, s_1 \), and the solution is

\[ s_n = ((s_1 - 2s_0) n + 2s_0) 2^{n-1}. \]
Example 10.2.2. Consider the second-order equation
\[ s_n = 4s_{n-1} - 4s_{n-2} + 3^n, \]
where \( \deg(p) = 0 \) and \( \lambda = 3 \) is not an eigenvalue of the recurrence. From the formula, a particular solution is \( v_n = c3^n \) for some constant \( c \), and
\[ 4v_{n-1} - 4v_{n-2} + 3^n - v_n = 3^{n-2}(12c - 4c + 9 - 9c) = 3^{n-2}(9 - c), \]
which equals zero for \( c = 9 \). This gives \( v_n = 3^n+2 \) and general solution\( s_n = (a_1n + a_2)2^n + 3^{n+2}, \) for some \( a_1, a_2 \in \mathbb{C}. \)

For the initial conditions \( s_0, s_1 \), computation gives
\[ s_n = \left( \frac{s_1 - 2s_0 - 9}{2} + \frac{s_0 - 9}{2} \right)2^n + 3^{n+2}. \]

Example 10.2.3. The second-order equation
\[ s_n = 4s_{n-1} - 4s_{n-2} + 3^n n \]
has \( \deg(p) = 1 \), and \( \lambda = 3 \) is again not an eigenvalue. Therefore, there are constants \( a, b \) such that \( v_n = (an + b)3^n \) is a particular solution. Also,
\[ 4v_{n-1} - 4v_{n-2} + n3^n - v_n \]
\[ = 3^{n-2}(12(an - a + b) - 4(an - 2a + b) + 9n - 9(an + b)) \]
\[ = 3^{n-2}((9 - a)n - (4a + b)), \]
which equals zero for all integers \( n \geq 2 \) when \( a = 9 \) and \( b = -4a = -36. \)
Therefore, \( v_n = (9n - 36)3^n = (n - 4)3^n+2 \) is a particular solution, and the general solution has the form
\[ s_n = (a_1n + a_2)2^n + (n - 4)3^{n+2}. \]

For initial values \( s_0, s_1 \) we obtain
\[ s_n = \left( \frac{s_1 - 2s_0 + 9}{2} + \frac{s_0 + 36}{2} \right)2^n + (n - 4)3^{n+2}. \]

Example 10.2.4. The second-order equation
\[ s_n = 4s_{n-1} - 4s_{n-2} + 2^n \]
has \( \deg(p) = 0 \), and \( \lambda = 2 \) is an eigenvalue. Since its multiplicity is 2, \( \delta = 2 \) holds in the formula, and \( v_n = an^22^n \) is a particular solution for some \( a \in \mathbb{C}. \)
Then
\[ 4v_{n-1} - 4v_{n-2} + 2^n - v_n \]
\[ = 2^n(2a(n - 1)^2 - a(n - 2)^2 + 1 - an^2) \]
\[ = 2^n(-2a + 1), \]
implying $a = \frac{1}{2}$, which means that $v_n = n^22^{n-1}$ is a particular solution. The general solution therefore has the form

$$s_n = (n^2 + a_1n + a_2)2^{n-1},$$

and from the initial values $s_0, s_1$ we obtain

$$s_n = (n^2 + (s_1 - 1 - 2s_0)n + 2s_0)2^{n-1}.$$

**Example 10.2.5.** The second–order equation

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

has $\text{deg}(p) = 1$, and $\lambda = 2$ is again the double eigenvalue of the recurrence, which means that $v_n = 2^n q(n)$ is again the particular solution for some $q(n) = an^3 + bn^2$. Then

$$4v_{n-1} - 4v_{n-2} + n2^n - v_n = 2^n[2q(n-1) - q(n-2) + n - q(n)],$$

which must equal zero for all values of $n \geq 2$. From

$$2q(n-1) - q(n-2) + n - q(n) = 0 \text{ for } n = 2, 3$$

we obtain $a = 1/6$ and $b = 3a = 1/2$. Therefore, $v_n = \left(\frac{1}{3}n^3 + n^2\right)2^{n-1}$ is a particular solution, and the general solution has the form

$$s_n = \left(\frac{n^3}{3} + n^2 + a_1n + a_2\right)2^{n-1}.$$

Initial values $s_0, s_1$ give

$$s_n = \left(\frac{n^3}{3} + n^2 + (s_1 - 2s_0 - \frac{4}{3})n + 2s_0\right)2^{n-1}.$$

Since from (10.8) the solution is $2^n Q(n)$ with $Q(n)$ a polynomial of degree 3, the same solution could have been obtained by solving a system of four linear equations to find the coefficients of $Q(n)$. To obtain these coefficients as functions of only $s_0$ and $s_1$, the recurrence (10.2.5) could be used twice to give $s_2$ and $s_3$ in terms of $s_0$ and $s_1$. 

112
10.3 One Multiple Root, Several Simple Roots

Solving (L) when there is a multiple eigenvalue
When $m_1 \geq 2$ and $m_2 = \cdots = m_k = 1$ are the respective multiplicities of the distinct eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_k$ of the recurrence (L), then $v_n = \lambda^n q(n)$ is a particular solution where $q$ is a polynomial with $\deg(q) = \deg(p)$ and

$$\delta = \begin{cases} 
0 & \text{if } \lambda \notin \{\lambda_1, \ldots, \lambda_k\} \\
1 & \text{if } \lambda = \lambda_1 \\
m_1 & \text{if } \lambda = \lambda_1 \\
1 & \text{if } \lambda \in \{\lambda_2, \ldots, \lambda_k\}
\end{cases}$$

Since each of the last five examples had only one (double) root, The last alternative of (10.9) did not occur. The next example illustrates this case.

Example 10.3.1. The second–order equation

$$s_n = 4s_{n-1} - 5s_{n-2} + 2s_{n-3} + 2^n$$

has $\deg(p) = 0$ and $\lambda = 2$. Since the characteristic polynomial factors as

$$ch(x) = x^3 - 4x^2 + 5x - 2 = (x - 1)^2(x - 2),$$

$\lambda = 2$ is a simple root of $ch(x)$ and $\delta = 1$, which gives the particular solution $v_n = an^2$. Then

$$4v_{n-1} - 5v_{n-2} + 2v_{n-3} + 2^n - v_n$$

$$= 2^{n-2}(8a(n - 1) - 5a(n - 2) + a(n - 3) + 4 - 4an) = 2^{n-2}(4 - a),$$

and $v_n = 4n^2 = n2^{n+2}$ is a particular solution, and the general solution is

$$s_n = (a_1 n + a_2) + a_3 2^n + n^2 = (a_1 n + a_2) + (a_3 + 4n) 2^n.$$

For initial values $s_0, s_1, s_2$ we obtain

$$s_0 = a_2 + a_3$$

and

$$s_1 = a_1 + a_2 + 2a_3 + 8; \quad s_2 = 2a_1 + a_2 + 4a_3 + 32,$$

which implies

$$s_n = (-2s_0 + 3s_1 - s_2 + 8)n + (2s_1 - s_2 + 16) + (s_0 - 2s_1 + s_2 - 16)2^n + n2^{n+2}.$$

10.4 The Input is $\gamma_1^n p_1(n) + \gamma_2^n p_2(n)$

For this form of input, one can find a particular solution by taking the sum of particular solutions to two related equations. Of course, there is nothing special about 2, so if the input is a sum of $j$ terms, one can find a particular solution by taking the sum of particular solutions to $j$ related equations.
Example 10.4.1. For the second-order equation

\begin{equation}
  s_n = 4s_{n-1} - 4s_{n-2} + 3^n n + 2^n,
\end{equation}

we break this equation into two equations, one for the input \(3^n n\) and one the input \(2^n\)

\begin{align*}
  (10.11) & \quad v_n = 4v_{n-1} - 4v_{n-2} + 3^n n, \\
  (10.12) & \quad w_n = ws_{n-1} - 4w_{n-2} + 2^n.
\end{align*}

From Example 10.2.2, we have that

\[ v_n = (n - 4) 3^{n+2} \]

is a particular solution to (10.11), and from Example 10.3.1,

\[ w_n = n^2 2^{n-1} \]

is a particular solution to (10.12). Combining these we have that

\[ (n - 4) 3^{n+2} + n^2 2^{n-1} \]

is a particular solution to (10.10). The general solution to (10.10) is then

\[ s_n = (\alpha_1 n + \alpha_2) 2^n + (n - 4) 3^{n+2} + n^2 2^{n-1}, \]

where \(\alpha_1\) and \(\alpha_2\) depend on the initial conditions \(s_0\) and \(s_1\). Solving for \(\alpha_1\) and \(\alpha_2\), we find that

\[ s_n = \left(\frac{s_1}{2} - s_0 + 4\right) n + s_0 + 36 \right) 2^n + (n - 4) 3^{n+2} + n^2 2^{n-1}. \]
Chapter 11

Further Reading

11.1 Books


11.2 Articles

• M. Agrawal. N. Kayal, and N. Saxena. 2002. PRIMES is in P. www.cse.iitk.ac.in/primality.pdf


