Module CS36111

Advanced Algorithms 1: Part II Complexity

David Lester

Michaelmas, 2010
Contents

1 Introduction
   1.1 Sub-Module Aims ................................................. 1
   1.2 Sub-Module Objectives ........................................... 1
   1.3 Sub-Module Assessment ........................................... 2
   1.4 Sub-Module Contents ............................................. 2
   1.5 An Example: nfib ................................................. 2
   1.6 Reading Guide .................................................... 5
   1.7 Lecture Summary .................................................. 5
   1.8 Next Lecture ...................................................... 5

2 Algorithmic Efficiency ............................................... 7
   2.1 What makes a program acceptable? ................................. 7
   2.2 Order of magnitude improvements ................................ 8
   2.3 Time analysis of recursion ....................................... 10
   2.4 Upper and lower bounds ......................................... 11
   2.5 Observations ...................................................... 11
   2.6 Asymptotic Analysis .............................................. 12
   2.7 Next Lecture ....................................................... 16

3 Recurrence Relations .................................................. 17
   3.1 Recurrence Relations for Recursive Functions ..................... 17
   3.2 Recurrence Relations for While Programs ......................... 19
   3.3 Linear Recurrence Relations ..................................... 20
   3.4 Solving Linear Recurrence Relations ................................ 21

4 Computability and Decidability ..................................... 23
   4.1 Computability ...................................................... 23
   4.2 Church-Turing Hypothesis ....................................... 26
   4.3 The Halting Problem .............................................. 27
CONTENTS

4.4 Certificates ................................................. 28
4.5 Relative Computing ............................... 29
4.6 A Lower Bound for search ............................... 29

5 NP-Completeness ............................................. 31
  5.1 P and NP .............................................. 31
  5.2 NP-Completeness ...................................... 34
  5.3 Important NP-Complete Problems ....................... 36

A Exercises ................................................. 37
  A.1 Introduction ........................................... 37
  A.2 Algorithmic Efficiency ............................... 37

B Exercises .................................................. 39
  B.1 Recurrence Relations ................................. 39
  B.2 Computability and Decidability ......................... 39

B References ............................................... 40
Lecture 1

Introduction

1.1 Sub-Module Aims

Our aim in this sub-module is to analyse programs so that we understand why they take as long to execute (or consume as much resource) as they do. With this analysis we can often significantly improve our programs' performance.

Students taking this sub-module will gain a deeper understanding of the complexity analysis of some standard programming language constructs, and of the overall structure of the resultant complexity classes.

This sub-module is a follow-on to COMP26120 “Algorithms and Imperative Programming”, given by Pete Jinks, Joshua Knowles, Milan Mihajlovic, and David Rydeheard. We’ll be using their recommended textbook [GT02], which although a bit basic for our purposes, at least gives a clue to what’s going on.

1.2 Sub-Module Objectives

At the end of the sub-module you will be able to perform the following:

1. analysis the space and time complexity of a recursive program;
2. analysis the space and time complexity of an iterative program;
3. solve simple recurrence relations;
4. classify problems as: computable, decidable, and semi-decidable;
5. classify problems into complexity classes;
6. relate complexity classes; and

7. complexity reductions on problems.

1.3 Sub-Module Assessment

There are two assessed components: 75% of the marks for this Sub-module are available by answering one question from two in the January exam; 25% of the marks are available for written answers to designated exercises.

1.4 Sub-Module Contents

1 Introduction  (This lecture!) An overview of the sub-module, and a worked example (Section 1.5).

2 Algorithmic Efficiency  The “Big-Oh” notation and its relatives. How to manipulate the notation.

3 Recurrence Relations  How to find the time- and space-complexity of a program by generating recurrence relations. How to solve simple recurrence relations.

4 Computability and Decidability  We will discuss the issue of computability; i.e. what it means for a function to be computable. We will also discuss decidability and partial decidability, and include a discussion of the halting problem.

5 Complexity Reduction  How to show that one problem is as hard or easy to solve as another problem.

6 NP problems  Problems that can be solved easily with non-deterministic choice.

1.5 An Example: fib

Let’s start with an example. Suppose that we have to implement the following function:
1.5. AN EXAMPLE: NFIB

\[
\begin{align*}
nfib(0) &= 1 \\
nfib(1) &= 1 \\
nfib(n + 2) &= nfib(n + 1) + nfib(n) + 1
\end{align*}
\]

Exercise 1.1

What does this function do?

Exercise 1.2

How can we write this in “C”?

Let’s try out our simple implementation. After all it might be fast enough. This is a key part to generating reliable and maintainable code: always try the simple thing first.

Exercise 1.3

How fast does this program go? (You should investigate the use of `/usr/bin/time`).

Suppose that we are unhappy with the speed; will compiler optimization help?

Exercise 1.4

How fast does the “optimized” program go? (You should investigate the use of flags with `gcc`, particularly `-O4`. Also check the generated code with `objdump -d`).

The problem with aggressive optimization is that we may have changed the meaning of our program. This is particularly true of code involving floating point numbers where we would most like help. In short: “Code generated by optimizing compilers is less reliable.” You have been warned!

Can we make this program even faster? Without resorting to unsafe optimization?

We can utilize a technique called dynamic programming. Using this technique, we build up the desired result from simpler, smaller, results. This contrasts to the divide-and-conquer (divide a big problem into smaller problems) approach we have previously taken.
Consider the problem of evaluating \( \text{nfib}(15) \).

\[
\text{nfib}(15) = \text{nfib}(14) + \text{nfib}(13) + 1 \\
= (\text{nfib}(13) + \text{nfib}(12) + 1) + (\text{nfib}(13)) + 1
\]

The evaluation of \( \text{nfib}(13) \) will take place twice. The situation is much, much worse for \( \text{nfib}(0) \), which is called about 1,000 times.

In the dynamic programming style, we instead construct a table of intermediate results – from smaller to larger – and then get our overall result that way. Functional Programmers call this memo-ization.

**Exercise 1.5**

Rewrite the program in the dynamic programming style in “C”?

**Exercise 1.6**

How fast does this alternative program go?

We may be feeling a bit smug with ourselves at this point, but why is the dynamic program so much better? How much better is “so much better” anyway?

**Divide-and-conquer Time-complexity**  This is a bit tricky to analyze. Still, let’s count the number of returns from subroutine we do. For 0 and 1 we return immediately, so the answer is 1. For the recursive case, we must count the number of returns for \( \text{nfib}(n+1) \) and \( \text{nfib}(n) \), add them together and then return. This gives the following recurrence relation \( T(n) \) for the number of returns executed by \( \text{nfib}(n) \):

\[
\begin{align*}
T(0) & = 1 \\
T(1) & = 1 \\
T(n+2) & = T(n+1) + T(n) + 1
\end{align*}
\]

Notice that \( T \) is actually defined the same way as \( \text{nfib}! \).

In what follows it can be useful to know that the *golden ratio* \( \phi = \frac{1}{2}(1 + \sqrt{5}) \), a related constant is \( \hat{\phi} = \frac{1}{2}(1 - \sqrt{5}) \).

**Exercise 1.7**

How fast does \( T(n) \) grow?
Dynamic Programming Time-complexity  Clearly, the time taken to run this program depends on how many values of the table we fill in. To run \texttt{nfib} \((n)\), we will need to fill in \(n + 1\) table entries (don’t forget 0). We will say that this program is \textit{linear time} (see Lecture 2).

\textbf{Constant time \texttt{nfib}?}  Can we do any better than linear time? How can we use Exercise A.1.

1.6 Reading Guide

It is recommended that students investigare the following textbooks:

- Goodrich and Tamassia [GT02]. Covers most of the material required, but can be a bit superficial in places.

- David Harel [Har87]. Out of print (but second hand copies are available; and of course there are also libraries on campus). An excellent primer; again short on specific details.

- Donald Knuth [Knu79]. Everyone who’s any pretence to being a computer scientist has this book on their shelves. Now very old fashioned. Excellent for detail; which is probably why it looks so old fashioned.

- Nigel Cutland [Cut80]. Specifically for the computability in Lecture 4.

1.7 Lecture Summary

The Lecture looked at improvements to a simple program.

1.8 Next Lecture

In the next lecture we arm ourselves with the tools needed to analyze the complexity of programs.
Lecture 2

Algorithmic Efficiency

In this lecture we will be looking at the measures of efficiency that we may apply to the programs we write.

At the end of the lecture you should understand the terms: time and space complexities, upper and lower bounds, closed problems and algorithmic gaps.

The material of this lecture can be found in [Har87, Chapter 6].

2.1 What makes a program acceptable?

We'll begin with an analogy; in civil engineering what are the reasons for selecting one bridge design over another? The answer is that it must meet its specifications; one of which is undoubtedly the cost. Since in algorithm design, man-power costs, capital costs, and other related costs are not relevant, we are left with two measures of the cost: materials and time. In computer science we refer to these costs as the complexity measures of space and time. The space complexity of an algorithm is the number of variables and the number and size of the data structures it uses. The time complexity of an algorithm is the number of elementary steps performed by a processor in an execution of the program.

\(^1\)The word algorithm derives from the Persian mathematician Muhammad ibn Mûsâ al-Khwârizmî (c.780 – c.850). Although the original Arabic text is lost, the Latin translation of his work, which popularised the Hindu positional number system in Western Europe, has the title: Algorithmi de numero Indorum. His other blockbuster, Kitab al-jabr w’al-muqabala, of which an Arabic text is extant, gives us the word describing the other major strand of this course: Algebra.
2.1.1 Time and space complexities in practice

Both the space and time complexities of an algorithm are likely to change when the program is run with different input. It should be clear that the \texttt{sum} function will take longer to run with a longer list as its input. It is probably worth convincing yourself of this fact by doing exercise 2.1.

Exercise 2.1

Write a ‘C’ function that performs vector addition on two vectors with a fixed (and equal) size.

Run your program with vectors of size $2^{10}$, and $2^{20}$; How long will the algorithm take to run with a vector of length $n$?

The fact that the time taken depends on the length of the input doesn’t mean that we can’t give an exact formulation of the time complexity. We quantify the input (in this case by measuring the size of the vector input) and determine the time taken for vectors of different lengths.

In this course we will concentrate on time complexity, but the reader should be aware that similar concerns arise in determining the space complexity of an algorithm.

2.1.2 Computers can help

There are many ways standard ways to improve the running time of an algorithm; some are incorporated into compilers as optimizations\textsuperscript{2}.

The use of an aggressive optimization strategy can sometimes result in the running time of the algorithm being reduced by a significant amount. The gain, whilst welcome, has been made at the expense of some thought from the programmer in transforming the algorithm. Furthermore, we have lost some of the modularity gained by using higher-order functions.

2.2 Order of magnitude improvements

Whilst the gain of the previous example was impressive, we can often do even better. Recall, from Lecture 1, that we proved that there are better (and

\textsuperscript{2}This is something of a misnomer, as it implies that the programs are “optimal”, which as we shall see, they aren’t. A better term might be \textit{improving compilers}, which has a useful double meaning for compilers with endemic software “maintenance” problems. It is, however, unlikely that we can buck forty years of misuse of the term “optimizing compilers”.

worse) ways to program the \texttt{nfib} example. Let’s investigate their relative performance.

**Exercise 2.2**

For which values of \( n \) will the recursive version of \texttt{nfib} out-perform the dynamic programming version?

One way to simplify discussion of algorithmic performance is to use asymptotic analysis. The idea of this method is to permit discussion of the relative performance of algorithms as the measure of the input \( N \) grows ever larger. We say that the running time of dynamic programming version of \texttt{nfib} is \( O(n) \), whilst that of the divide-and-conquer version is \( O(\phi^n) \).

Notice that we didn’t mention “execution steps” or quantify the measure of the units of time we are using. Interestingly enough this information isn’t necessary. Another point of note about these two examples is that we could accurately characterize the running time independently off the exact input given to the two algorithms. In general this is not the case.

Consider the problem of finding someone’s telephone number in an address book that is totally jumbled – like my address book. Here’s an informal algorithm to search the list of pairs of names and numbers: Starting at the beginning, work through to the end of the list comparing names; if a match is found, then return the associated number.

On average, how long does it take me to find someone’s telephone number? What is the worst time it will take me to find a number?

**2.2.1 There is a better way**

If, unlike me, you keep an address book with the entries nicely ordered; so that, for example, Howard Barringer’s number comes after Richard Banach’s, which in turn comes before Ning Zhang’s; then finding a telephone number is considerably easier. Suppose that I wished to find Howard’s number in the ordered directory. First I open the book about half way and discover a number for John Latham. This tells me that the number for Howard is earlier in the book, so I make another attempt one quarter of the way through; this time I find a number for Steve Furber. Still not early enough in the book, and so on . . . . In the worst case, how long will this algorithm take to find a number for someone, given that my address book has \( N \) entries?
2.2.2 Robustness

If we are happy with asymptotic analysis, we must now show that it suffices to just count comparisons; after all there are many more steps executed by the 'C' program when finding a number in a telephone directory. The actual number of operations associated with each call to \texttt{find} will be bounded, by \( K \) say. Therefore, the number of operations for the whole program will also be bounded by \( K \) times the worst case number of comparisons.

This is both the strength and the weakness of asymptotic bounds on the complexity of algorithms. The strength lies in the universality of the complexity. We can be sure that for inputs of sufficient size the dynamic programming \texttt{nfib} algorithm will eventually out-perform that of divide-and-conquer \texttt{nfib}. And, of course, that is also the weakness: we can’t be sure that the inputs we are interested in will be of sufficient size.

2.3 Time analysis of recursion

Since recursive functions form such a major part of programming with a functional language, it will pay to understand how to do a time-complexity analysis of recursive algorithms. Let’s consider this in the context of providing a measure for the \texttt{findNumber} function of the previous section.

Given a directory of \( n \) entries, the time taken in the worst case, \( T(n) \) will be one of the following cases:

- If \( n = 0 \) then the tree must be a leaf and \( T(n) = 1 \), \textit{i.e.} we made one comparison.
- If \( n \neq 0 \) then, because we wish to deal with the worst case, the name must be in one of the sub-trees. But how big is each sub-tree? The answer is \( n/2 \) and hence \( T(n) = 1 + T(n/2) \). (We are treating the test for whether the tree is a branch as part of the comparison.)

This gives rise to the following \textit{recurrence relation}:

\[
\begin{align*}
T(0) &= 1 \\
T(n) &= 1 + T(n/2)
\end{align*}
\]

From this we conclude that \( T(n) = \log_2 n \).

2.3.1 Average case complexity

Although the worst case complexity is a useful measure of an algorithm’s performance, it is possible that an algorithm will perform well for most
inputs, but produce a poor performance for just a few inputs, which we might be willing to ignore. What we need to measure in this case is the average-case time complexity of the algorithm. Here we consider the time taken in the average, considering all possible inputs, and the probability of their occurring.

Despite the difference, some algorithms have the same asymptotic worst-case and average-case behaviour. Others exhibit a much better performance in the average-case than in the worst-case. The most famous example is quicksort. A little experimenting should convince you that for most inputs this algorithm runs in time $O(n \log n)$. On the other hand, comparing the time taken to sort the lists $[1..10]$ and $[1..100]$ ought to demonstrate that the worst-case time complexity is $O(n^2)$.

2.4 Upper and lower bounds

Having seen how the naïve telephone directory search algorithm (with time complexity $O(n)$) improved by the binary search technique (with time complexity $O(\log n)$), it is natural to ask whether there is an end to this process. It’s useful to think of a problem having some (possibly undiscovered) optimal solution; if you have an algorithm to solve the problem, then you have a worst case bound on the time complexity of the problem. In the case of searching the telephone directory, we may be sure that solving the problem should take no longer than $O(\log n)$.

But is this the best we can do? Is there some better algorithm out there waiting to be discovered?

One thing we might attempt to do is to prove that a problem has a lower bound time complexity. In the case of searching, we can show that the lower bound is $O(\log n)$ and hence that the binary search is asymptotically the best that we can do. Instead of saying that the lower case time complexity of the search problem is $O(\log n)$, we may write that it is $\Omega(\log n)$.

2.5 Observations

2.5.1 Closed problems and Algorithmic gaps

Often we can establish lower bounds on the algorithmic problem. For example searching an unordered list of length $n$ is easily shown to require $n$

\footnote{It is of course possible to improve the constant factor that is hidden in the big-O of the time complexity.}
comparisons in the worst case. We were able to supply a linear-time algorithm for this problem and hence we are able to conclude that the upper bounds and the lower bounds actually meet. We call a problem were we have shown that an asymptotic lower bound of the worst case is the same as the asymptotic complexity of an algorithm, a closed problem.

Another example of a closed problem is searching an ordered list; it has upper and lower bounds of $O(\log n)$. Many problems are not closed in this way; their upper and lower bounds do not meet. Such cases are referred to as algorithmic gaps. An example of such a problem is that of finding the minimum cost railway network between $n$ towns. The important point about such gaps is that they highlight our lack of knowledge about the solution not that there is a something wrong with the problem.

### 2.5.2 Research on efficiency

The topics discussed in this lecture (and the next one) are labelled under the heading: concrete complexity theory. This is a particularly strong research topic within the US academic community. A good example is the minimum cost railway network problem. The best known algorithm for this problem runs in time bounded by $O(f(n) \times n)$ where $f(n)$ grows so slowly that $f(n) < 6$ for all numbers we will be interested in.

Still this is mildly unsatisfactory, and we might hope that one of two things might happen: firstly, that a linear algorithm will be found; or failing that, that a non-linear lower bound – matching the known upper bound – will be found.

### 2.6 Asymptotic Analysis

(From [GT02, Section 1.2])

#### 2.6.1 The “Big-Oh” Notation

The key idea of “Big-Oh” is to capture the notion of “eventually”; in the sense of “If the problem is large enough, eventually Algorithm A is better than Algorithm B”. Although the “Big-Oh” notation is capable of considerable extension, let’s start with polynomials.

**Polynomials** We begin with an easy definition of the “Big-Oh” notation for polynomials.
2.6. ASYMPTOTIC ANALYSIS

Definition 2.3

If \( f : \mathbb{N}_{\geq 0} \rightarrow \mathbb{R} \) is a polynomial of degree \( k \) — i.e. it can be written as
\[
f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0
\]
then we say that
\[
f(n) \text{ is } O(n^k).
\]

(Pronounced as “\( f(n) \) is Big-Oh of \( n^k \),” or “\( f(n) \) is order \( n^k \).”)

In short: we take the largest power appearing in the polynomial and drop the coefficient.

Example 2.4

Let us suppose that \( f(n) = 4n + 3 \). Is \( f(n) = O(n^2) \)? Is it \( O(n^0) \) (usually written \( O(1) \))?

It is usual to try to express the “Big-Oh” function as the smallest of the possible candidates. Which of the following two replies to the question “How far is it to the railway station?” is the most helpful?

- “It is less than twelve hours away”; or
- “It is less than five minutes away”.

The polynomials above give rise to a hierarchy, since any function that is \( O(n^3) \) is also \( O(n^4) \), and so on.

Not everything can be expressed as a polynomial, so we have a quick look at other possibilities. Before we do so, we give a formal definition of the “Big-Oh” operator \( O \)

Definition 2.5

For \( f, g : \mathbb{N}_{\geq 0} \rightarrow \mathbb{R} \) we write \( f(n) \) is \( O(g(n)) \) if, and only if, there exist \( c \in \mathbb{R}_{>0} \) and \( k \in \mathbb{N}_{>0} \), such that for all \( n > k \):
\[
f(n) \leq cg(n)
\]

This leads to the following “rules” for “Big-Oh”. In what follows we assume that \( f_i(n) \) is \( O(g_i(n)) \) and \( f(n) \) is \( O(g(n)) \).

\[
\begin{align*}
k + f(n) & \text{ is } O(g(n)) & \text{Ignore addition of a constant} \\
kf(n) & \text{ is } O(g(n)) & \text{Ignore multiplication by a constant} \\
f_1(n) + f_2(n) & \text{ is } O(\max(g_1(n), g_2(n))) \\
f_1(n)f_2(n) & \text{ is } O(g_1(n)g_2(n))
\end{align*}
\]
Logarithms  One common possibility for \( f(n) \) is \( a \log(n) \). As usual with the “Big-Oh” notation we can ignore the coefficient \( a \) (use Definition 2.5). In addition, the base of the logarithm doesn’t matter.

**Lemma 2.6**

If \( b > 1, \ n \geq 0, \) and \( f(n) = a \log_b(n) \) then there exists \( a' \) such that:

\[
f(n) = a' \log_2(n)
\]

**Proof**

Suppose – without loss of generality that \( a \) and \( a' \) are positive – and that \( a \log_b(n) = a' \log_2(n) \). But \( \log_b(n) = \log_b(2) \log_2(n) \). Thus

\[
a' = a \log_b(2).
\]

We will therefore assume that all of our logarithms are base 2 (unless we decide otherwise, of course).

How does \( O(\log(n)) \) fit into our polynomial hierarchy?

- Since \( \log(n) \) is an increasing function, we have that any constant function \( f(n) \) (i.e. \( f(n) \) is \( O(1) \)) is \( O(\log(n)) \).

- We claim that if \( f(n) \) is \( O(\log(n)) \) then it is also \( O(n) \). Raise both sides to the power 2, to obtain the result.

It is also worth pointing out that we can have successive applications of the logarithm function, i.e. we can talk about \( O(\log(\log(n))) \). This is frequently written without internal brackets as \( O(\log \log n) \).

**Exponentials**  Another sort of function we will need to permit as \( f(n) \) are expressions of the form: \( f(n) = ax^n \) (for \( x > 1 \)). Let us first show that the value of \( x \) doesn’t matter.

**Lemma 2.7**

If \( x > 1, \ n \geq 0, \) and \( f(n) = ax^n \) then there exists \( a' \) such that:

\[
f(n) = a'2^n
\]
2.6. ASYMPTOTIC ANALYSIS

Proof

Suppose – without loss of generality that \( a \) and \( a' \) are positive – and that \( ax^n = a'2^n \), then taking the logarithm of both sides gives:

\[
\log(a) n \log(x) = \log(a') n \log(2).
\]

Hence

\[
\log(a') = \frac{\log(a) \log(x)}{\log(2)}.
\]

\[\square\]

In other words, with exponentials it doesn’t matter what the value of \( x \) is, except that it is greater than 1. (Why must \( x \) be greater than 1?)

Definition 2.8

If \( f : \mathbb{N}_{\geq 0} \to \mathbb{R} \) can be written as \( f(n) = ax^n \) (for \( x > 1 \)), then we say that

\( f(n) \) is \( O(2^n) \).

This is such an important special case that we say that: “\( f(n) \) is exponential in \( n \).”

Where does this fit into our polynomial heirarchy? Is there a natural number \( k \) such that \( f(n) = 2^n \) is \( O(n^k) \)?

Be aware that this is not the limit.

Definition 2.9

If \( f : \mathbb{N}_{\geq 0} \to \mathbb{R} \) can be written as \( f(n) = ax^{x^n} \) (for \( x > 1 \)), then we say that

\( f(n) \) is \( O(2^{2^n}) \).

This is also an important special case, and we say that: “\( f(n) \) is doubly exponential in \( n \).”

Another useful result is Stirling’s approximation formula for the factorial function.

Definition 2.10

\[
n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n.
\]

(Note that this is Stirling’s original result. There are many improvements possible, and these, too, are often known as “Stirling’s Formula”.)

Where does \( O(n!) \) fit into our exponential heirarchy?
Combining logarithms, exponentials and polynomials  We can freely mix exponential and logarithms and polynomials. The key to such mixtures is Definition 2.5.

Some common complexity classes are summarized in Table 2.1.

### Table 2.1: Terminology for classes of functions

<table>
<thead>
<tr>
<th></th>
<th>$O(1)$</th>
<th>$O(\log n)$</th>
<th>$O(n)$</th>
<th>$O(n \log n)$</th>
<th>$O(n^2)$</th>
<th>$O(n^k)$</th>
<th>$O(2^n)$</th>
<th>$O(2^{2^n})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>logarithmic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>linear</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>quasi-linear</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>quadratic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>polynomial</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>exponential</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>doubly-exponential</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.7 Next Lecture

In the next lecture we will find out how to determine the time- and space-complexity of a program by generating recurrence relations. We will also see how to solve simple recurrence relations.
Lecture 3

Recurrence Relations

In this lecture we show how to derive recurrence relations for the time complexity of an algorithm. We also show how to express (some) recurrence relations in “closed form”.

3.1 Recurrence Relations for Recursive Functions

We have talked glibbly about recurrence relations, but where do they come from? Let’s start with the simpler case where our program is described as a recursive function.

3.1.1 Simply Recursive Functions

Let us suppose that we have a program expressed as a recursive function; something like:

\[
\begin{align*}
  f(0) &= e_1 \\
  f(n + 1) &= e_2
\end{align*}
\]

This is recursive if, and only if, expression \( e_2 \) contains a sub-expression involving \( f \).

If we are analysing the time complexity of this recursive function, we will expect it to terminate. It will terminate if we can find some quantity that is getting smaller on each iteration and also if there is a lower limit. Normally we use \( \mathbb{N} \) for this purpose, and show that the quantity is always non-negative, and decreases by one for each iteration. A common case is where the occurrence of \( f \) in \( f(n + 1) \) is as \( f(n) \).

As we discussed in Lecture 2, we can hopefully associate a number with the time it takes to evaluate expressions \( e_1 \) and \( e_2 \). This could be the number
of (expensive operations), or the number of small chunks of computation associated with each right-hand side.

**What happens if there’s another function involved?** We will have to calculate it’s time-complexity first. Provided it isn’t defined in terms of \( f \) we will be OK. If it is defined in terms of \( f \) then see the following Section 3.1.2.

**What happens if there’s a data structure as argument?** It is normal to treat a tree-like data structure by considering it’s depth; *i.e.* the maximum length from any leaf to the root. Then it’s a natural number problem again.

If we are considering a function such as:

\[
\begin{align*}
  f(0) &= e_1 \\
  f(n+1) &= e_2[f(n)]
\end{align*}
\]

then we get a time complexity recurrence relation as follows:

\[
\begin{align*}
  T_0 &= a_1 \\
  T_{n+1} &= T_e(T_n) + a_2
\end{align*}
\]

where \( T_e \) is the time complexity function for the expression \( e_2 \). If we are lucky this function might be the identity function and we get:

\[
\begin{align*}
  T_0 &= a_1 \\
  T_{n+1} &= T_n + a_2
\end{align*}
\]

Now we can express \( T_n \) as \( na_2 + a_1 \).

**Exercise 3.1**

How can we prove this is right? Do it.

**Exercise 3.2**

What is the recurrence relation for the recursive function:

\[
\begin{align*}
  f(0) &= nfib(0) \\
  f(n+1) &= nfib(n+1) + f(n)
\end{align*}
\]

**Exercise 3.3**

What is the recurrence relation for the recursive function:

\[
\begin{align*}
  f(0) &= nfib(0) \\
  f(n+1) &= nfib(f(n))
\end{align*}
\]
3.1.2 Mutually Recursive Functions

We can extend the result to cover the general case where two or more functions are defined in terms of one another. What we have in mind is something like:

\[
\begin{align*}
    f(0) &= 1 \\
    f(n + 1) &= g(n) \\
    g(0) &= f(0) + 2 \\
    g(n + 1) &= f(n + 1) + 3
\end{align*}
\]

In this case it should be obvious that we can replace \( g(n) \) on the second line with either \( f(0) + 2 \) or \( f(n + 1) + 3 \) depending on the value of \( n \).

A more usual way for this situation to arise is in compilers. We might have a syntactic data structure “Expression Tree” that is referred to by another one “Statement Trees”. These in turn might be referred to from “Expression Trees”

3.2 Recurrence Relations for While Programs

At first sight it may appear that deriving recurrence relations for imperative constructs is very different from that presented above for recursive functions. Actually, it’s very similar. The first stage is to decide what operations we wish to count. Common ones include: arithmetic division (it’s time-consuming on ARM and in While); boolean conditions associated with branching (if then else); and the number of times we go around a while loop.

The key insight is that if a while loop is to terminate then – just as in the case of the recursive function – there must be some quantity that is getting smaller and that is bounded below. In other words, there is a natural number that is decreasing for every iteration of the loop. If this is not true, then the loop is non-terminating.

Exercise 3.4

What is the time complexity of a non-terminating program?

Exercise 3.5

What is the effect of the following program?
z := 0;
while (y <= x) do (x := x-y; z := z+1)

Does this always terminate?
Whenever the loop terminates, what is the decreasing natural number, that ensures termination?

3.3 Linear Recurrence Relations

Definition 3.6
A recurrence relation is linear if, and only if, it is of the form:

\[
\begin{align*}
T(0) & = a_0 \\
\cdots \\
T(k) & = a_k \\
T(n + k) & = b_k T(n + k - 1) + \cdots + b_1 T(n) + b_0
\end{align*}
\]

Exercise 3.7
Is the recurrence relation associated with \texttt{nfib} linear?

Exercise 3.8
Is the recurrence relation associated with \texttt{f} linear?

\[
\begin{align*}
f(0) & = a_0 \\
f(n + 1) & = f(n) + a_1
\end{align*}
\]

Exercise 3.9
Is the recurrence relation associated with \texttt{f} linear?

\[
\begin{align*}
f(0) & = a_0 \\
f(2n) & = f(n/2) + a_1 \\
f(2n + 1) & = f(n/2) + a_1
\end{align*}
\]
What is the closed-form for the associated \(T_n\)?

Exercise 3.10
Is the recurrence relation associated with \texttt{f} linear?

\[
\begin{align*}
f(0) & = \texttt{nfib}(0) \\
f(n + 1) & = \texttt{nfib}(f(n))
\end{align*}
\]
3.4 Solving Linear Recurrence Relations

We can solve—or reduce to closed form—general linear recurrence relations by using the technique of “moment generating functions”, see [Knu79, Page 80ff].

In brief, instead of considering the sequence of natural numbers \( \{T_i\} \), we consider the moment generating function \( G(z) \):

\[
G(z) = T_0 + zT_1 + z^2T_2 + \cdots + z^nT_n + \cdots
\]

Consider what happens when we “substitute” the above into the recurrence relation:

\[
\begin{align*}
T_0 &= a_1 \\
T_{n+1} &= T_n + a_2
\end{align*}
\]

We get:

\[
G(z) - zG(z) = T_0 + z(T_1 - T_0) + \cdots + z^n(T_n - T_{n-1}) + \cdots
\]

\[
= a_1 + a_2z + \cdots + a_2z^n + \cdots
\]

In other words (re-arranging, and with \(|z| < 1\)):

\[
(1 - z)G(z) = \frac{a_2}{1 - z} + (a_1 - a_2)
\]

We can now extract \( T_n \) as:

\[
T_n = \left[ \frac{d^n}{dz^n} G(z) \right]_{z=0}
\]

**Exercise 3.11**

Check that this is true for \( n = 0 \), and \( n = 1 \).

Of course in this case it is easy to see that \( T_n = na_2 + a_1 \).

How does this technique help us to solve the Fibonacci recurrence?

\[
\begin{align*}
T_0 &= 0 \\
T_1 &= 1 \\
T_{n+2} &= T_{n+1} + T_n
\end{align*}
\]

Using the “method of moments”, we get:

\[
(1 - z - z^2)G(z) = z
\]

The important part is the quadratic term on the left. Solving \( 1 - z - z^2 = 0 \) we get \( z = \frac{1}{2}(1 \pm \sqrt{5}) \). We write \( \phi = \frac{1}{2}(1 + \sqrt{5}) \), and \( \hat{\phi} = \frac{1}{2}(1 - \sqrt{5}) \).

Although there is a way to get at the answer algebraically, it is almost always easier to try out combinations of the form \( b_1\phi^n + b_2\hat{\phi}^n \) and see if we can guess the answer. Then check your guess using induction.
Lecture 4

Computability and Decidability

4.1 Computability

Informally, a function \( f : \mathbb{N} \rightarrow \mathbb{N} \) is computable, if and only if we can write a computer program that implements it. You might wonder whether it matters in which language we write the program, and usually it does not. However, for the sake of accuracy, we will take it to mean a program written in the **While** language (c.f. “Understanding Programming Languages”).

That is, a programming language with arithmetic expressions: \( a \) (note no division operation), boolean expressions \( b \), and statements \( S \):

\[
\begin{align*}
  a & := n \mid x \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1 \\
  b & := \text{true} \mid \text{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b_1 \mid b_0 \land b_1 \\
  S & := x := a \mid \text{skip} \mid S_1 ; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S
\end{align*}
\]

**Are there non-computable functions?** The short answer is: “Yes”. At this point people often ask to see the definition of a non-computable function. The problem is that we are used to seeing programs to specify our functions, and if we have a program, the function must be – by definition – computable.

Instead there are two ways we can proceed. If we are mathematically sophisticated we can claim that the number of computer programs is **countably infinite**, whilst the number of functions from \( \mathbb{N} \) to \( \mathbb{N} \) is **uncountably infinite**.
We will show that the first claim is true, but the second is not really part of this course.

**Theorem 4.1**

*Every program written in While has a unique \( n \in \mathbb{N} \) called it’s code number.*

**Lemma 4.2**

*We can code the variables (\( x \)) and numerals (\( n \)) in a program.*

**Proof**

For variables treat each character in the variable’s name as it’s ASCII code, and the variable’s value as a base 256 “decimal” expansion. The numeral \( n \) can be represented by the natural number it is meant to represent. We’ll write these codings as \( V \) and \( N \) respectively.

\[ \begin{array}{c}
V(a) = 0 \\
\vdots \\
V(z) = 25 \\
V(A) = 26 \\
\vdots \\
V(Z) = 51 \\
V(ax) = 0 + 52V(x) \\
\vdots \\
V(zx) = 25 + 52V(x) \\
V(Ax) = 26 + 52V(x) \\
\vdots \\
V(Zx) = 51 + 52V(x)
\end{array} \]

**Lemma 4.3**

*Every pair of natural numbers can be combined to give a code number.*
4.1. COMPUTABILITY

Proof

Assign the code \( c = 2^n(2m + 1) - 1 \) as the code for the pair \((n, m)\).

Then for every \( c \in \mathbb{N} \), we can extract the associated \( n \) by adding 1 and then dividing by 2 until we have an odd number. By subtracting 1 from the odd number and dividing by 2 we also get \( m \). This shows that \( P(n, m) = 2^n(2m + 1) \) is a bijection between \( \mathbb{N} \times \mathbb{N} \) and \( \mathbb{N} \).

\[ \square \]

Note very carefully, that all of the required operations to construct a pair code and to decode can be written in \textbf{While}.

Lemma 4.4

Every arithmetic expression has a unique code number.

Proof

We define \( A(a) \) by cases as follows:

\( a = x \) Use the variable coding \( V \). Return \( 5V(x) \).

\( a = n \) Use the numeral coding \( N \). Return \( 1 + 5N(n) \).

\( a = a_1 + a_2 \) Return \( 2 + 5P(A(a_1), A(a_2)) \).

\( a = a_1 - a_2 \) Return \( 3 + 5P(A(a_1), A(a_2)) \).

\( a = a_1 \times a_2 \) Return \( 4 + 5P(A(a_1), A(a_2)) \).

The function \( A \) is a bijection between \( a \) and \( \mathbb{N} \).

\[ \square \]

Exercise 4.5

Define a coding function \( B \) for the boolean expressions \( b \) of the \textbf{While} language.

Exercise 4.6

Define a coding function \( S \) for the statements \( S \) of the \textbf{While} language. You will need to use the pairing code function \( P \) twice for \texttt{if-then-else} statements.
At this point we have shown that there is a bijection between our While programs and the natural numbers. This is the definition of what it means to be countably infinite.

**Definition 4.7**

A set $S$ is *countably infinite* (also known as *denumerable*) if and only if there exists a bijective function $f : S \rightarrow \mathbb{N}$.

**Definition 4.8**

A set $S$ is *uncountably infinite* (or *uncountable*) if and only if the set $S$ is not finite and not countably infinite.

**Proposition 4.9**

The set of functions $f : \mathbb{N} \rightarrow \mathbb{N}$ is uncountable.

Being horribly informal about it: “There are lots more functions than there are computable functions.”

Before we leave the issue of coding, we should also point out that an implementation of the While language can also be performed in While. To do this we need a way to code a finite sequence of integers; we code the sequence of integers $[k_1, k_2, \ldots, k_n]$ as:

$$2^{k_0}3^{k_1} \cdots p_n^{k_n} - 1$$

where $p_n$ is the $n$th prime number. The above is a bijection between finite sequences and $\mathbb{N}$; and this suffices to show that a Statement/State pair can be encoded. If we then model how the execution progresses (Structural Operational Semantics from COMP36411) the execution of a program is a function from $\mathbb{N} \rightarrow \mathbb{N}$.

Notice that all of the coding/decoding mechanisms involve simple arithmetic operations that can be implemented in While; this is no accident. As we shall now see

### 4.2 Church-Turing Hypothesis

The Church-Turing Hypothesis is that all sensible notions of what it means to be computable are the same.

**Hypothesis 4.10**
4.3. THE HALTING PROBLEM

All notions of computability are equivalent.

Exercise 4.11

Why is this not a theory?

We shall take as our definition of computability the use of the While language.

Definition 4.12

A function \( f : \mathbb{N} \rightarrow \mathbb{N} \) is computable if, and only if, there is a While program which, given input \( x \), computes \( y = f(x) \).

There are alternative definitions of computability, such as \( \lambda \)-calculus, and Turing machines, which are useful in certain contexts.

An alternative proof that there must be functions that cannot be programmed in While (or any other programming language) is to consider the halting problem.

4.3 The Halting Problem

In this section we will see why we are always guaranteed jobs as computer programmers. We could agree that a very basic correctness criteria we would wish our algorithms to have is that they terminate for every possible input\(^1\). As we shall see in this section this is not possible.

Consider the following program:

\[
\text{while not } (x = 1) \text{ do } x := x - 2
\]

Assuming that the legal inputs (\( i.e. \) values of \( x \)) for this function are the positive natural numbers (1, 2, \ldots), it is clear that the program will halt for odd values of \( x \) and fail to terminate for even values of \( x \). Here’s another one to try:

\[
\text{while not } (x = 1) \text{ do if even}(x) \text{ then } x := \text{div}(2) \text{ else } x := 3 \times x + 1
\]

Does this program halt on all possible inputs? No one knows; it does halt for all the values that people have tried so far. Let us now define a version

\(^1\)Awkward coves may wish to raise the issue of real-time controllers and operating systems as examples of programs that are not intended to terminate. Let them.
of the halting problem. Suppose that we have a function $p$ that takes one (arbitrarily-sized) integer to another. What we would like to have is a function $h$ which given a function $p$ and an input value $n$ returns $\text{True}$ if $p$ will terminate on the input $n$, and $\text{False}$ otherwise. Let’s assume that we have translated the text that represents the function $p$ to an integer.

A special case of the putative halt-tester function is $n$ which takes a number representing a function $p$; it returns $\text{True}$ if $p(p)$ halts, and $\text{False}$ otherwise. The counter-example is now the funny function $f$:

$$f(p) = \text{if } n(p) \text{ then (while true do skip) else False}$$

**Exercise 4.13**

What is the result of applying $f$ to $f$?

What we have now shown is that the While language is not sufficiently expressive to write the halt-tester function $h$. Is this really the end of the line, or might it not be possible to extend While with some new construct that would make it possible to write $h$? The answer – provided by the Church-Turing Thesis – is no; While (and any other sensible programming language) is already as powerful as any computing notation.

**Decision Procedures** If we treat the function $f : \mathbb{N} \rightarrow \mathbb{N}$ as a function that returns a boolean; typically with 0 representing false and 1 representing true, then a decision procedure is a computable function returning 0 or 1 as appropriate.

**Semi-decision Procedures** A semi-decision procedure is one that will always return 1 when the result is true, but may either fail to terminate or return 0 when the result is false. The Halting problem is an example of a semi-decision procedure.

**4.4 Certificates**

Notice that the halting problem has a finite certificate; if a program $p$ halts on an input $x$: we simply run the program $p(x)$, and behold, it stops. Of course the trouble arises if we attempt this with an input value $x$ for which $p$ doesn’t halt. In this case our certificate is now infinite; i.e. testing the certificate will not halt. Problems such as the halting problem are said
to have one-way certificates; or alternatively that the problem is partially decidable.

What happens for decision problems for which we have finite certificates for both True and False? The answer is no, and the reason is that we can attempt to test for both cases at once in a simulation of a parallel test.

### 4.5 Relative Computing

Be very careful about this. The idea is to add special functions to the While language; as we have seen if we add a “halt-tester” function we can generate a contradiction. It is also possible to augment our language with features that cannot be practically realized. These features are often called “Oracles” when they are boolean-valued, such as the putative halt-tester function. My best advice is to treat any results in this area very sceptically.

### 4.6 A Lower Bound for search

To demonstrate that “search is $O(\log n)$” we need to be very, very, precise about what we mean. In particular, we must specify what counts as our computer’s primitive operations, and also the nature of the data structure being searched.

**Example 4.14**

It is possible to sort a list of integers in the range 0...1023 in linear time, provided that we can afford to set aside 1024 memory locations. (Bucket Sort).

Without constraints on the data we’d expect that sorting requires $O(n \log n)$ operations.

**Example 4.15**

Suppose that we are searching the following telephone directory:

[Aardvark → 1001, Badger → 1002, ... Zebra → 1026]

Then we can search the list in constant time.

So, let’s agree that there is no relationship between the names in our telephone directory and the numbers.
Example 4.16

The SpiNNaker architecture includes a 1024 entry three-way content addressable memory. With this architecture one can look up a 1024 entry telephone directory in $O(1)$ time.

At bottom, the SpiNNaker CAM architecture is really a parallel processor; we’ll discuss them later. Instead, suppose that we have to write our program in While. We will suppose that the “name” we are looking for is passed in as an integer in variable $x$, and that the associated telephone number is delivered as output in variable $y$.

As there are no arrays in the language, this must mean that for a directory of size $n$ there must be $n$ different assignment statements of the form $y := m$ in our program.

We now turn to the ways in which we can construct our access to the assignments. This must depend in some way on the input $x$. The only operations on $x$ at our disposal are comparisons: specifically the operations $x \leq k$ and $x = k$ ($k$ is some constant; after all we have no other variables).

That’s it. So our program is constructed from the following statements:

$$
S ::= \text{if } x \leq k \text{ then } S_1 \text{ else } S_2 \\
| \quad \text{if } x = k \text{ then } S_1 \text{ else } S_2 \\
| \quad y := m
$$

Therefore, in order to minimize the time complexity of our algorithm, we must minimize the depth of the syntax tree of $S$. This can be achieved by making the tree balanced. In other words each subtree $S_1$ and $S_2$ of the statement $\text{if } x \leq k \text{ then } S_1 \text{ else } S_2$ must have the same number (or as close as possible) of assignments in each branch. So the depth of the binary tree is $O(\log n)$ for a tree with $n$ entries.
Lecture 5

NP-Completeness

5.1 P and NP

5.1.1 Defining P and NP

We need to be a bit more careful about measuring the size of the input and output than we have been up until now. For example, integer addition cannot be a constant-time operation (consider the carry propagation problem that exists in any hardware implementation). We must therefore now consider the bit measure of the input (and output) sizes.

**Definition 5.1**

An algorithm is *c*-incremental if, and only, if any primitive operation on objects represented by at most $b$ bits results in an object with at most $b + c$ bits ($c \geq 0$).

**Exercise 5.2**

Is integer addition *c*-incremental? Is integer multiplication *c*-incremental? What about division?

**Lemma 5.3**

*If a c-incremental algorithm is worst-case $T(N)$ then it is $O(n^2t(n))$ where $n$ is the number of bits required to encode $N$."

In other words provided the language supports “reasonable” operations – *i.e.* it is *c*-incremental – then any polynomial-time algorithm remains polynomial-time when it’s bit-complexity is considered.
The Complexity Class \( P \)

A decision problem is a task undertaken by a decision procedure; i.e. computational problems that deliver a boolean result. If the potential inputs to the program are regarded as a set, then the subset of inputs that generate “true” from the decision procedure defines that decision procedure. In particular if the input is a string, then the subset of all strings accepted by the decision procedure is the language \( L \)

**Definition 5.4**

The *The Complexity Class* \( P \) is the set of all decision problems (or languages) that can be accepted in worst-case polynomial-time.

Note carefully that we have not said anything about the running time of the case where the input \( x \notin L \).

**Lemma 5.5**

*If* \( L \in P \) *then the complement of* \( L \) *is also in* \( P \).

The Complexity Class \( NP \)

We augment the *While* language with a \texttt{choose}(\( x \)) construct. This assigns either 0 or 1 to the variable \( x \). An algorithm *nondeterministically accepts* a language \( L \) if there is a set of assignments to the choose variables that will permit the algorithm to accept the language \( L \). In other words, we are assuming that magic takes place, and that the program can always guess correctly which of the possible choose assignments will lead to acceptance.

**Definition 5.6**

The *The Complexity Class* \( NP \) is the set of all decision problems (or languages) that can be nondeterministically accepted in worst-case polynomial-time.

**Definition 5.7**

A language \( L \) is said to be in \texttt{co-NP} if the complement of \( L \) is in \( NP \).

**Hypothesis 5.8**
Although no-one knows, it is widely believed that

\[ \text{co-NP} \neq \text{NP}. \]

Hypothesis 5.9

Although no-one knows, it is widely believed that

\[ P \neq \text{NP}. \]

Hypothesis 5.10

We don’t even know whether

\[ P = \text{NP} \cap \text{co-NP}. \]

An Alternative Definition of NP

There is an alternative definition of what it means for a decision problem to be in NP, and we will need this alternative when we discuss the Cook-Levin Theorem in Subsection 5.2.2.

We say that a language \( L \) is verified by an algorithm \( A \) if, given a string \( x \in L \) as input, there is another string \( y \) such that \( A \) outputs “yes” on input \( z = x + y \). We call the string \( y \) the certificate for membership of \( L \).

Theorem 5.11

A language \( L \) can be (deterministically) verified in polynomial time if and only if \( L \) can be nondeterministically accepted in polynomial time.

Proof

See [GT02, Theorem 13.2, page 596].

\[ \square \]

5.1.2 The Defining Problem in NP

The circuit satisfiability problem is to take as input the logic circuit of a boolean function – described as a collection of logic gates connected by wires in a directed acyclic graph – and to ask whether there is any assignment to the circuit’s inputs for which the circuit will generate a true output. We call this problem **CIRCUIT-SAT**
Lemma 5.12

\textbf{CIRCUIT-SAT is in NP.}

Proof

For each of the \( n \) inputs \( x_i \) we use \texttt{choose}(\( x_i \)) to guess the correct input. We then check that this assignment gives the correct result. (It might not, if there is no correct assignment). This is linear in the size of the circuit.

\square

5.2 NP-Completeness

The idea of nondeterministic acceptance is weird because it violates every principle of what it is possible to do in practice with a conventional computer. Indeed my hunch is that this will turn out to be true for nonconventional computers such as quantum computers and DNA or brain-inspired computers too.

The real purpose and usefulness of the complexity class \textbf{NP} is that it captures a class of problems that appear to be more difficult than polynomial time, but which have not yet been proved to be exponential time. In other words the current best algorithms for these problems are exponential time, but the currently proved lower bound on their complexity are polynomial. This is an algorithmic gap of epic proportions.

5.2.1 Polynomial-Time Reducibility and NP-Hardness

Definition 5.13

We say that a language \( L \) is \textit{polynomial-time reducible} to a language \( M \) if there is a computable polynomial-time function \( f \) with the property:

\[ x \in L \text{ if and only if } f(x) \in M \]

We can express this by the notation \( L \xrightarrow{\text{poly}} M \).

Definition 5.14
5.2. **NP-COMPLETENESS**

We say that the language $M$ is **NP**-hard, if and only if for all $L \in \text{NP}$, $L \xrightarrow{poly} M$.

**Definition 5.15**

If a language $M$ is **NP**-hard, and $M \in \text{NP}$ then we say that $M \in \text{NP}$-complete.

In other words if anyone finds a deterministic polynomial-time algorithm for just one **NP**-complete problem then $P = \text{NP}$.

5.2.2 **The Cook-Levin Theorem**

**Theorem 5.16**

**CIRCUIT-SAT** is **NP**-complete.

**Proof**

We know that **CIRCUIT-SAT** $\in \text{NP}$. So all we have to do is show it is also **NP**-hard.

Consider $L \in \text{NP}$. This means that there is a deterministic algorithm $D$ that accepts any $x \in L$ in polynomial-time $p(n)$, given a polynomial-sized certificate $y$, where $n$ is the size of $x$. The main idea of the proof is to build a large – but polynomial-sized circuit $C$ that simulates the algorithm $D$ on input $x$ in such a way that $C$ is satisfiable if and only if there is a certificate $y$ such that $D$ outputs “yes” on input $z = x + y$.

Recall that if $D$ is an algorithm, then we can implement it as a **While** program. Without going into details, we can build a circuit to implement a computer system that performs a single step of the **While** language. This will consist of at most $cp(n)^2$ logic gates, for some $c > 0$. Call this circuit $S$ (for step).

To simulate the execution of a program with $p(n)$ steps we simply connect $p(n)$ copies of the logic circuit $S$ together, using the output from one step as the input to the next. Call this circuit $C$.

The total size of this complete circuit $C$ is $O(p(n))^3$, which is polynomial.

Consider $x$ which $D$ accepts with certificate $y$ after $p(n)$ steps. Then there is an assignment of values to the inputs of $C$ corresponding to $y$,
such that $C$ will simulate $D$ on this input and the hard-wired values for $x$. Ultimately $C$ will output 1. Thus $C$ is satisfiable. Conversely, suppose that $C$ is satisfiable, then there are a set of inputs (corresponding to the certificate $y$) such that $C$ outputs 1. But as $C$ exactly simulates $D$ this implies that there is an assignment of values to the certificate $y$ such that $D$ outputs “yes”. Thus $D$ verifies $x$ in this case.

\[\square\]

5.3 Important NP-Complete Problems

The following problems are important NP-complete problems.

- Knapsack
- Hamiltonian-cycle
- Travelling Salesman (TSP)
Appendix A

Exercises

A.1 Introduction

Exercise A.1

Use induction to show that:

\[ n\text{fib}(n) = 2 \left( \frac{\phi}{\sqrt{5}}(\phi^n - \hat{\phi}^n) + \hat{\phi}^n \right) - 1 \]

A.2 Algorithmic Efficiency

Exercise A.2

Using Definition 2.5 show that if \( f(n) \) is \( O(g(n)) \) then \( k + f(n) \) is also \( O(g(n)) \).

Exercise A.3

Using Definition 2.5 show that if \( f(n) \) is \( O(g(n)) \) then \( kf(n) \) is also \( O(g(n)) \).

Exercise A.4

Using Definition 2.5 show that if \( f_1(n) \) is \( O(g_1(n)) \) then \( f_1(n) + f_2(n) \) is \( O(\max(g_1(n), g_2(n))) \).

Exercise A.5

37
Using Definition 2.5 show that if $f_i(n)$ is $O(g_i(n))$ then $f_1(n)f_2(n)$ is $O(g_1(n)g_2(n))$.

**Exercise A.6**

Using Definition 2.5 show that if $f(n)$ is $O(1)$ then it is also $O(\log(n))$.
Also show that any function $f(n)$ that is $O(\log(n))$ is also $O(n)$. You may assume that the logarithms are base 2.

**Exercise A.7**

What is the relationship between $O(\log n)$ and $O(\log \log n)$. Use Definition 2.5 to prove your result.

**Exercise A.8**

What can you say about the asymptotic behaviour of $f_1(n) - f_2(n)$?
You may assume that $f_i(n)$ is $O(g_i(n))$, and that $g_i(n)$ is polynomial.

**Exercise A.9**

Where does $O(n!)$ fit into the exponential heirarchy? Use Definition 2.5 and Stirling’s Formula to prove your result.
Appendix B

Exercises

B.1 Recurrence Relations

B.2 Computability and Decidability
Bibliography


