Chapter 3

Complexity and Asymptotic Analysis

In this part of the course we will look at measures of program execution efficiency. In other words, how to answer the question *which program is better?* We will begin by discussing how we can analyse algorithms directly and then use this to motivate a technique for approximating their behaviours as the problem size increases. This discussion will focus on *upper bounds* on the amount of time required. We will extend the discussion to also consider *lower bounds* and *space complexity*. We then meet the concept of *complexity classes*, which are a way in which we can group functions together based on their complexity. The chapter will finish with a discussions about practical applications of these concepts.

Learning Outcomes

At the end of this Chapter you will:

- Be familiar with the methods used to analyze the efficiency of algorithms written in an imperative programming language;
- Be able to explain the ‘Big-Oh’ notation (and its relatives $\Omega$ and $\Theta$);
- Be able to analyse and compare the asymptotic complexities of imperative programs; and
- Be able to describe the relationship between time complexity and space complexity and the notion of a *complexity class*.
3.1 Setting the Scene

To set the scene we answer a few clarifying questions about what this chapter is focussing on.

What are we analysing?

We are analysing programs (or algorithms, they mean the same thing here) to measure their complexity. This is usually taken to mean predicting the resources required by the algorithm. Sometimes this is memory, communications bandwidth, or computer hardware, but usually we are concerned with the time taken to execute the algorithm. And for the first part of this chapter this is what we will focus on.

Which version of a program are we analysing?

If we wanted to write a program to check if an input is larger than 1000 we could write the following program:

```plaintext
result := 0
while (n>0 & result=0)
    if n ! = 1000 then n := n−1 else result := 1
```

If we give this program 1000 it terminates immediately with the correct answer. If we give the program 999 it will perform 999 iterations before it terminates with the correct answer. If we give the program 10000 it will perform 9000 iterations before it terminates with the correct answer. If we assume that the program could receive numbers in the range 1 to 10000 with uniform probability then the average number of iterations performed will be

\[
\frac{1}{10000} + \left(\sum_{n=1}^{999} \frac{1}{10000}n\right) + \left(\sum_{n=1001}^{10000} \frac{1}{10000}(n−1000)\right)
\]

\[
= \frac{1}{10000} \left(1 + \frac{999\times1000}{2} + \frac{10000\times1001}{2} - \frac{1000\times1001}{2}\right) + (10000 \times -1000)
\]

\[
= \frac{1}{10000} (1 + 499500 + (5005000 - 500500) - 10000000) = 4000.4
\]

Which took some non-trivial calculations to find out. Which behaviour are we interested in? The best case? The worst case? The likely case? How do we know what the likely case is? Above I assumed that the inputs were drawn uniformly from some range but we rarely know such details. If we are implementing a sorting function are the inputs likely to be already sorted?
If we are implementing a function to turn an NFA into a DFA, is the input almost an NFA already?

Practically we often think about the common case but in these notes we will mainly be focussing on the worst case as this allows us to more easily compare programs without worrying about what the expected use case is.

What’s the difference between programs and problems?

Previously we were very particular about the difference between a program and the function it computes i.e. the problem it solves. That same distinction is still important here. Two programs can compute the same function (solve the same problem) and have different complexities. But we might also be interested in the inherent complexity of a problem itself. For example, finding the maximum value in an array is a problem that inherently requires us to look at every element in an array. No solution can be a correct solution without doing this. This suggests a lower bound on the complexity of any solution. Throughout this chapter we will abuse terminology to talk about a problem’s complexity. When we do this we mean the best possible program that could solve that problem.

Therefore, when we talk about programs we will usually think in terms of upper bounds (worst case) and when we talk about problems we talk about lower bounds (best case). We clarify this more later.

What are we not doing?

This isn’t a course in algorithms. We won’t be introducing complicated algorithms to solve specific problems. In fact, we won’t be looking at many programs. The main learning outcome is to understand the definition of Big-Oh notation and what it means. This is a first year course. We won’t be exploring advanced material on computational complexity. The only model of computation we have met is the while language, so we won’t be looking at things from the context of Turing Machines and we won’t be considering parallel, non-deterministic, or random computation, all of which change the story.

We are not analysing programs in terms of their readability or understandability. This is not what we mean by complexity here. Such things are important but this chapter is interested in the complexity of the computation performed by a program not the program itself.
CHAPTER 3. COMPLEXITY AND ASYMPTOTIC ANALYSIS

3.2 Simplifying Assumptions

To answer the question of how long a program will take to execute we need to answer some more simple questions first. For example, how long does it take a computer to compute $1 + 2$? This will depend on the architecture and its instruction set. For some reduced instruction set architectures, computing $1 + 2$ might require multiple instructions to place 1 and 2 into registers first, taking multiple cycles. For a complex instruction set there may be a single instruction that does everything in one cycle, but typically cycles take longer. This distinction might not matter for adding two 32-bit integers. But what about multiplication or division? What if we are doing security for e-commerce involving the RSA cryptographic protocols, where we might need to perform operations on 200-400 digit numbers. These will be implemented as a sequence of machine instructions operating on vectors of 32 bit unsigned integers. Thus the number of operations performed will depend on the size of the integers concerned. In this course we have been considering $\mathbb{Z}$ in general; but arguably arithmetic does not have fixed cost for all integers. As a simplifying assumption we can assume that we only deal with numbers that fit into 32-bit integers, or at least numbers for which the cost of arithmetic operations on them is fixed.

Can we access and write to all memory in constant time? Not necessarily, to perform an operation on a data value the computer may need to move it into a register, and move the result. More generally, computers have memory hierarchies with different access times. As a simplifying assumption we can assume that all accesses occur in constant time. However, in many practical programs memory access time is the limited factor and in such cases it should take part in the complexity analysis.

As part of complexity analysis we measure the usage of some resource but the units that we measure will have a significant impact on the result. Particularly, for time complexity we need to decide which operations matter and what their associated cost should be.

<table>
<thead>
<tr>
<th>Important</th>
</tr>
</thead>
<tbody>
<tr>
<td>The instructions or operations that we measure, is an informed choice made by the computer scientist. Ideally, this choice should be made so that the results obtained are useful.</td>
</tr>
</tbody>
</table>
3.3. Analyzing Algorithms

Recall the division algorithm introduced in Exercise 1.2.1:

```plaintext
1   r := x;  d := 0;
2   while y <= r do
3       d := d+1;  r := r-y;
```

How many primitive operations are performed executing this program? Let’s count the operations used on each line. Line 1 contains two assignments. Line 2 performs the loop check. If the loop is executed \( N \) times then we perform \( N + 1 \) operations. Similarly, on line 3 we will perform \( N \) operations, \( N \) operations, and \( 2N \) assignments. The only question we now need to address is how big is \( N \)? Notice that variable \( d \) is assigned the value of 0; and each time we iterate through the loop we increment \( d \) by 1. At the end of the execution of the loop, \( d \) will have the value \( \lfloor \frac{x}{y} \rfloor \), and this is thus the value of \( N \) as well. So, if we assume that all of these operations have the same cost, the cost of this program is

\[
3 + 5 \left\lfloor \frac{x}{y} \right\rfloor
\]

Now let us consider the slightly more complex program from Example 1.2.5 where the following program checks whether \( x \) is prime.

```plaintext
1   y:=x-1;
2   z:=1;
3   while (y>1 ∧ z=1) do (    
4       r:=x;
5       (while (y <= r) do r:=r-y);
6       (if (r=0) then z:=0);
7       y:=y-1;
8   )
```

Let’s consider the case where \( x \) is prime, this is the worst case as when \( x \) is non-prime the outer loop will terminate early. In this case we check if every number less than \( x \) divides \( x \). There are \( x-2 \) such numbers so the outer loop executes \( x-2 \) times. The inner loop is the algorithm we used above, so we can reuse the results from that analysis. Given the three initial operations (two assignments and a \( - \)) before the outer loop and the additional 2 assignments, 1 check and one \( - \) inside the outer loop, the cost of this program in this worst case is

\[
3 + (x-2) \left( 5 + 5 \left\lfloor \frac{x}{y} \right\rfloor \right)
\]
CHAPTER 3. COMPLEXITY AND ASYMPTOTIC ANALYSIS

We could also make a similar calculation for cases where the outer loop terminates early, but we would need to know how early, which is not possible to compute in general as we don’t know how primes are distributed in general. But as we motivated previously, the worst case is usually what we want to know about.

Exercise 3.1

Perform a similar count of operations for the square-root program shown in Example 1.2.2.

Exercise 3.2

Perform a similar count of operations for the factorial program you wrote in answer to Exercise 1.8.

Exercise 3.3

Perform a similar count of operations for the log-base-two program you wrote in answer to Exercise 1.11.

3.4 Asymptotic Analysis or Order of Growth

There are a number of the problems with performing an exact analysis of the sort we did in the previous section:

- The exact operation counts often end up being extremely complicated;
- It is often necessary to consider many different cases;
- There is no easy way to analyze sub-components of an algorithm and then combine the results for the whole program; and
- Different compilers (or compiler options) might affect the result.

Computer Scientists use Asymptotic Analysis or “Big-Oh” notation to solve all of these problems; how’s that for economy of effort?

So far we have been counting operations, such as addition, subtraction and comparison. It may not have been apparent, but on different machines, we may get very different performances for a particular algorithm. However, unless it has a very strange instruction set – say a one cycle matrix-multiply instruction – there will be a relationship between the size of the input and the length of time taken to run the algorithm. Consider the following program that sums the numbers from 1 to $n$. 

...
sum := 0; while (n > 0) do (sum := sum + n; n := n - 1)

When \( n = 50 \) the program should (roughly) perform half the number of steps as when \( n = 100 \). Consider two machines, one where operations take \( A \) units of time and another where they take \( B \) units of time. If \( A = 2 \) and \( B = 10 \) then on the first machine the \( n = 100 \) case is considerably faster than the \( n = 50 \) case on the second. But on the same machines the relationship between the running time and the size of \( n \) is the same. We don’t want to have to consider how fast performing operations is on any particular machine, so how do we abstract away from this point that one machine might implement addition faster than another machine? We should ignore any constants like \( A \) and \( B \) in our calculations.

**Observation 3.4.1**

This observation allows us to ignore the constant scaling factor for the time taken.

In short: if the algorithm is acceptably fast, but a speed-up is required — just buy a faster machine! The other observation is that in the end, for big enough problems, we can ignore some of the terms of our exact complexity, because they become insignificant. We explore this idea in the following examples but you have already met this idea in Section 5.1 of the COMP11120 material.

**Example 3.4.2**

Consider the two functions, each of type \( \mathbb{N} \to \mathbb{N} \):

\[
\begin{align*}
  f(n) &= n \\
  g(n) &= n^2
\end{align*}
\]

It does not matter which value of \( n \) we select, it will always be the case that \( n \leq n^2 \). We say that \( n^2 \) dominates \( n \).

**Example 3.4.3**

Suppose that the two functions (again \( \mathbb{N} \to \mathbb{N} \)) this time are:

\[
\begin{align*}
  f(n) &= 100n \\
  g(n) &= n^2
\end{align*}
\]
Now, if $n < 100$ then $f(n) > g(n)$. For example at $n = 10$ we get $f(n) = 1000$ and $g(n) = 100$. But eventually $g(n)$ will be bigger than $f(n)$. In this case “eventually” means for any value of $n > 100$; for other functions this may differ. See Figure 3.1 for a relevant graph.

There are some less intuitive relationships between functions, as illustrated in Figure 3.1. For example, $\frac{x}{100}$ might feel like it’s going to be very small but it is a linear function of $x$ and will grow linearly with $x$. However, $\log_2(x)$ grows much more slowly than $x$. Notice that $\frac{1000}{100} = 10$ and $\log_2(1000) < 10$ (as $2^{10} = 1024$).

**Definition 3.4.4**

We say the function $g$ **eventually dominates** function $f$, whenever there exists $k : \mathbb{N}$ such that:

$$\forall (n : \mathbb{N}). n > k \Rightarrow g(n) > f(n)$$
3.5. **PROPERTIES OF ‘BIG-OH’**

In other words, whenever \( n \) is greater than \( k \), and we have \( g(n) > f(n) \), then we can say that \( g \) eventually dominates \( f \).

**Observation 3.4.5**

This observation allows us to ignore terms in a function that are eventually dominated by another – bigger – term in the function.

Putting Observations 3.4.1 and 3.4.5 together we have arrived at the following useful conclusion for polynomial complexities: if

\[
T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0
\]

then it will eventually be dominated by \( Cn^k \), for some \( C > 0 \). If, in addition, we do not care about the scaling constant we can simple talk about the function \( n^k \). This is important enough to restate.

**Important**

Observation 3.4.5 tells us that we can view any polynomial function \( f \) of order/degree \( k \) as having the same complexity as the function \( n^k \) as there will always be a constant \( C \) such that \( Cn^k \) will eventually dominate \( f \).

We can now introduce the “Big-Oh” notation. We will define a set of functions associated with a particular function \( g \). This set of functions is written \( O(g) \), and members of the set have the following property: if \( f \in O(g) \) then there is a (positive) constant \( C > 0 \) such that \( f \) is eventually dominated by \( C \times g \). Formally, we have the following definition:

**Definition 3.4.6**

If \( f, g : \mathbb{N} \to \mathbb{R}_{\geq 0} \) and \( f \in O(g) \), then there exists \( k \in \mathbb{N} \) and \( C > 0 \) such that for all \( n > k \):

\[
f(n) \leq C g(n)
\]

### 3.5 Properties of ‘Big-Oh’

In this section we will demonstrate some of the properties of the notation. You should be aware that many people are informal and sloppy in their use
of the notation, so be prepared for this. In particular, instead of writing the function properly you will see things such as \( O(1) \) and \( O(n) \). These are actually shorthand for the polynomial functions \( f(n) = 1 \) and \( f(n) = n \) respectively. In places we may also be sloppy.

The first simple property of the notation is that it is reflexive (if you cannot remember what this means then look back at the material in COMP11120).

**Lemma 3.5.1**

\[ f \in O(f) \]

**Proof**

Pick \( k = 0 \) and \( C = 1 \). Then for all \( n \in \mathbb{N} \):

\[ f(n) = Cf(n) \leq Cf(n) \]

\[ \square \]

The trick we used in the above proof is that the definition of the notation says that there exists \( k \) and \( C \) so we are allowed to provide these values to demonstrate that a function belongs to a certain set.

**Exercise 3.4**

Show that “Big-Oh” is *not* symmetric; i.e. if \( f \in O(g) \) then it is not necessarily the case that \( g \in O(f) \).

**Exercise 3.5**

Show that “Big-Oh” is transitive; i.e. if \( f \in O(g) \) and \( g \in O(h) \), then \( f \in O(h) \).

**Exercise 3.6**

Argue that “Big-Oh” is a pre-order relation. **Hint:** Recall the definition given for a relation to be a pre-order from COMP11120. Or, just look it up!

**Exercise 3.7**

Is “Big-Oh” anti-symmetric? i.e. if \( f \in O(g) \) and \( g \in O(f) \), then must the two functions \( f \) and \( g \) be the same?
Let us next show that the constant functions $O(1)$ are a strict subset of the linear functions $O(n)$. Before we do this, let us consider what $O(1)$ means. These are all the functions whose running time is independent of the size of the input.

**Lemma 3.5.2**

$O(1) \subsetneq O(n)$

i.e. $O(1)$ is a strict subset of $O(n)$.

**Proof**

Suppose $f \in O(1)$. Then there exists $k, C$ such that for all $n > k$:

$$f(n) \leq C \cdot 1 = C$$

We must now show that $f \in O(n)$ as well. Choose $k' = \max(k, 1)$ and $C' = C$. Then for all $n > k'$ we will have:

$$f(n) \leq C \leq Cn$$

Thus $f \in O(n)$.

Now we will show that there is at least one function that is in $O(n)$ that is not in $O(1)$. We choose the function $g(n) = n$. This is clearly in $O(n)$ by Lemma 3.5.1. However, it is not in $O(1)$. Suppose that $g$ was a member of $O(1)$. Then there exists $k$ and $C$ such that for all $n > k$

$$g(n) = n \leq C$$

But, by choosing $n > \max(C, k)$, we will violate this condition.

Next we establish the observation we made in Observation 3.4.5 that we can treat a polynomial $g$ containing the largest polynomial $n^k$ in the same way as $n^k$. Here this means that $O(g) = O(n^k)$. 


Lemma 3.5.3

Suppose \( g(n) \) is a polynomial, i.e.
\[
g(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0
\]
Provided that the leading coefficient \( a_k > 0 \), then the set \( O(g) \) is the same set as \( O(n^k) \)

Proof

The proof is in two parts. First, suppose \( f \in O(g) \), then by Definition 3.4.6 there exists \( K \) and \( C \) such that for \( n > K \):
\[
f(n) \leq C g(n)
\]
Provided \( n \geq 1 \), and \( i > j \) we have that \( n^i \geq n^j \). So take \( K' = \max(K,1) \), and
\[
C' = \sum_{i=0}^{k} C|a_i|, \text{ then } f(n) \leq C' n^k, \text{ and thus } f \in O(n^k).
\]
Now suppose that \( f \in O(n^k) \). Then there are \( K \) and \( C \) such that for \( n > k \) we have:
\[
f(n) \leq C n^k
\]
Choose \( K' = \max(K,1) \) and \( C' = C/a_k \) (provided \( a_k > 0 \)), then
\[
f(n) \leq C' g(n)
\]
As required.

\[\square\]

Next we show that the complexity of polynomial classes is strict. That is, for example,
\[
O(n) \subsetneq O(n^2) \subsetneq O(n^3) \subsetneq \ldots
\]
which should not be surprising.

Lemma 3.5.4

\[
O(n^m) \subsetneq O(n^{m+1})
\]
3.5. PROPERTIES OF ‘BIG-OH’

Proof

Suppose \( f \in O(n^m) \). Then there exists \( k, C \) such that for all \( n > k \):

\[
f(n) \leq Cn^m \leq Cn^{m+1}
\]

as \( n^m \leq n^{m+1} \) and therefore \( f \in O(n^{m+1}) \). However, we can show that \( g(n) = n^{m+1} \) is not in \( O(n^m) \) as follows. Let us suppose that it is, then there exists \( k, C \) such that for all \( n > k \):

\[
g(n) = n^{m+1} \leq Cn^m
\]

however if we let \( n = C + 1 \) (we can control \( n \) as this must hold for all \( n \)) we get

\[
(C + 1)^{m+1} = (C + 1)^m(C + 1) \leq (C + 1)^m C
\]

and dividing through by \( (C + 1)^m \) gives us

\[
C + 1 \leq C
\]

which is clearly contradictory and our assumption that \( n^{m+1} \in O(n^m) \) is invalid.

\[\square\]

Finally, we consider \( O(\log(n)) \). Notice that Big-Oh notation allows us to ignore the base of the logarithm as we have

\[
\log_b(x) = \frac{\log_a(x)}{\log_a(b)}
\]

and therefore \( \log_a(b) \) (which will be constant) can be taken as a constant scaling factor. To see why we are particularly interested in \( O(\log(n)) \) see the later section with examples using arrays.

Lemma 3.5.5

\[
O(1) \subsetneq O(\log n) \subsetneq O(n)
\]

(In Computer Science we usually take \( \log(n) \) to be \( \log_2(n) \).)

Exercise 3.8

Prove Lemma 3.5.5. Hint: The inverse to \( \log_2(x) \) is \( 2^x \).
3.6 Some Exercises

3.6.1 Thinking about Functions

In this course I want you to become familiar with how functions grown and the idea that even though one function may eventually dominate another function, this may not be practically relevant. For example $n^2$ eventually dominates $1000000$ but only when $n$ is bigger than 1 million so if the input $n$ is unlikely to be this large then $n^2$ is a better solution.

To help understand this point we consider the following exercise that requires you to compare functions. You should try to complete this exercise without a calculator at first as it is good to get a feel for the comparative size of functions without needing to calculate their exact value (you can check your answers with a calculator). To help with this you should remember that

\[
\begin{align*}
2 &= 2^1 & \log_2(2) &= 1 \\
4 &= 2^2 & \log_2(4) &= 2 \\
\vdots & & \vdots \\
128 &= 2^7 & \log_2(128) &= 7 \\
256 &= 2^8 & \log_2(256) &= 8 \\
512 &= 2^9 & \log_2(512) &= 9 \\
1024 &= 2^{10} & \log_2(1024) &= 10
\end{align*}
\]

and therefore, for example, $\log_2(500) < 9$ as $\log_2(a) < \log_2(a + b)$ for $b > 0$.

Exercise 3.9

Complete the following table by placing $<, =, >$ in each box to indicate the relationship between either $f(n)$ and $g(n)$ for some $n$ or the relationship between $O(f)$ and $O(g)$ where we abuse notation and write $<$ for strict subset etc.

<table>
<thead>
<tr>
<th>$f(n)$</th>
<th>$g(n)$</th>
<th>$f(n) \ ? \ g(n)$</th>
<th>$O(f) \ ? \ O(g)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$2n$</td>
<td>$n = 5 \quad n = 250$</td>
<td>$&lt;$</td>
</tr>
<tr>
<td>$n$</td>
<td>$n^2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n^2 + n$</td>
<td>100$n$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\frac{n^5}{2n^2}$</td>
<td>$5n^3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\log_2(n)$</td>
<td>$\frac{n}{2^x}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n - 2$</td>
<td>$\log_2(\log_2(n))$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n^a$</td>
<td>$2n^2$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3.6. SOME EXERCISES

I have filled in the first row already. For both \( n = 5 \) and \( n = 250 \) the function \( g(n) = 2n \) will be bigger than \( f(n) = n \). However, they are both in the same Big-Oh class as \( O(n) = O(2n) \) due to the constant scaling factor observation.

3.6.2 Looking at Programs

Now that we have the Big-Oh notation we should apply it to some programs. In the `while` language it can be relatively easy to spot when things are getting complex as we can only use loops to multiply the operations we perform and we require nested loops to obtain large complexities. In the real world you will often be dealing with recursive functions, in such cases things can become less clear.

However, note that loops do not necessarily imply certain complexities. Consider the following program:

```plaintext
sum := 0; n := 10000;
while (n > 0) do (sum := sum+n; n := n-1)
```

This program computes the sum of the numbers up to 10000 and contains a loop that iterates 10000 times. However, its complexity is \( O(1) \) as its running time is independent of any input. This demonstrates a limitation of asymptotic complexity and a warning about counting loops to guess complexity.

An example of a program with non-constant complexity is our division program (repeated again for the forgetful).

```plaintext
r := x; d := 0; while y ≤ r do d := d+1; r := r–y;
```

As we saw in Section 3.3 the number of iterations required is linear in \( \frac{x}{y} \), so the complexity of this program is \( O(n) \), given \( n = \frac{x}{y} \).

As an aside, we need to be careful here and the topic of complexity analysis for multiple variables is...complex\(^1\). In general, people tend to also be sloppy about this. Here we defined exactly how the complexity was related to the relationship between our inputs.

Exercise 3.10

Write `while` programs with the complexities \( O(1) \), \( O(n) \), \( O(n^2) \), and \( O(\log_2(n)) \).

\(^1\)e.g. see *On Asymptotic Notation with Multiple Variables* by Rodney R. Howell
CHAPTER 3. COMPLEXITY AND ASYMPTOTIC ANALYSIS

Exercise 3.11

What are the Big-Oh complexities of the following programs:

- The square-root program shown in Example 1.2.2
- The primes program shown in Example 1.2.5
- The factorial program you wrote in answer to Exercise 1.8
- The log-base-two program you wrote in answer to Exercise 1.11

A far less obvious program for complexity analysis is our gcd program from Example 1.2.4:

\[
\text{while } (x \neq y) \text{ do (}
\begin{align*}
\text{if } (x > y) \text{ then} & \quad x := x - y \\
\text{else} & \quad y := y - x \\
\end{align*}
\)
\]
\[z := x\]

We need to identify the worst case inputs. It turns out that this is when \(x\) and \(y\) are consecutive Fibonacci numbers. A full description of the complexity in this case is beyond the scope of this course. But I encourage the interested reader to search for the answer online.

3.7 Further Examples Using Arrays

In the real world programs often contain arrays so it is useful to briefly consider some examples of complexity analysis with such programs. Additionally, I want to give an example of a practically interesting program with logarithmic complexity and to do this I need to use arrays.

Here I consider programs in the \texttt{while}_{\texttt{arr}} language introduced in Section 1.5 and reuse some of the programs introduced there. Note that this language is not examinable and the ideas in this section are here to give you a more realistic feel for the use of complexity analysis.

Recall the program introduced in Example 1.5.4 for computing the dot-production program for \(n\)-dimensional vectors:

\[
dot := 0; \text{ for } i = 1 \text{ to } n \text{ do } dot := dot + a[i] \times b[i]
\]
This program is $O(n)$ for vectors of dimension $n$ as for each element in the vector we perform a constant number of operations (which can be hidden in the constant scaling factor).

As a more involved example, consider the problem of writing a telephone directory look-up program. Suppose that we have two arrays, each of dimension $n$. The first, called $e$, holds the employee number, and the second, called $t$, holds the corresponding employee’s telephone number. Further suppose that the employee numbers in array $e$ are held in order, i.e.

$$i < j \implies e[i] < e[j]$$

One way to write the telephone directory look-up program to find the telephone number of employee with number $ex$ is as follows:

```plaintext
tel := -1;
for i = 1 to n do
    if e[i] = ex then tel = t[i] else skip
```

This has the effect of looking for employee $ex$’s telephone number starting at the beginning of the directory, and assigning the variable $tel$ the telephone number if an entry is found. What is the complexity of this program? We do a constant amount of work for each element in the array so it is $O(n)$.

**Exercise 3.12**

Modify the linear search algorithm above to give a best case complexity of $O(1)$. What is the best case for linear search?

But this is a poor algorithm. Instead we should start in the middle and do a “binary chop” based on the employee number at the middle of the directory.

```plaintext
lo := 0;
hi := n;
mid := n/2;
tel := -1;
while ¬(e[mid] = ex) ∧ ¬(lo = hi) do
    (if e[mid] ≤ ex then lo := mid
        else hi := mid; mid := (hi-lo)/2;)
if e[mid] = ex then tel := t[mid] else tel := -1
```

The key idea in this algorithm is that the area being searched is halved each time around the loop. This is the binary search program shown previously in Example 1.5.2 (see the comment about division there).
What is the complexity here? Every time we go around the loop we effectively halve the part of the array we are looking at. Are we familiar with a function that iteratively halves its input? The complexity of this program is $O(\log_2(n))$.

**Exercise 3.13**

Compute the complexity of the sorting algorithm in Example 1.5.3 and for the program you wrote in Exercise 1.28. Research the best algorithm for sorting in terms of complexity.

### 3.8 Lower Bounds

There is a counterpart to the ‘Big-Oh’ notation: the ‘Big-Omega’ notation. The idea here is that by saying that $f \in \Omega(g)$ we are saying that eventually the function $f$ eventually dominates $g$; i.e. $g$ is a lower bound to $f$.

Formally, we define $\Omega$ as follows:

**Definition 3.8.1**

If $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ and $f \in \Omega(g)$, then there exists $k \in \mathbb{N}$ and $C > 0$ such that for all $n > k$:

$$Cg(n) \leq f(n)$$

The important point to remember is: *algorithms* have an upper bound on their execution time (“quicksort is average-case $O(n \log n)$”); and *problems* have a lower bound (“sorting must be $\Omega(n)$ since we must compare each element at least once”).

---

**Important**

This distinction is important. Saying that a problem has a lower bound means that any solution to that problem must be at least that complex. This is a far stronger argument than the arguments we have been making so far, which have been about particular algorithms/programs.

---

\(^2\)If you haven’t met quicksort then I strongly suggest looking it up and reading about its complexity.
3.8. LOWER BOUNDS

Because these differ (by a factor of \( \log n \)) there is a chance that we will find a better algorithm for sorting. In fact the best current sorting algorithm is also \( O(n \log n) \). The situation where the best algorithm is worse than the problems lower bound is very common, indeed it is unusual for there to be no difference. The existence of a difference is called an “algorithmic gap”.

**Definition 3.8.2**

We say that there is an algorithmic gap if the problem lower bound complexity is less than the current best algorithm’s upper bound complexity.

One example of a problem for which we know that there is no algorithmic gap is calculating the maximum of a non-empty array of numbers of size \( n \).

```plaintext
max := 0;
for i := 1 to n do
    if max \leq a[i] then max := a[i] else skip
```

Since we have to inspect each value in the array, the lower bound on the calculation is at least \( \Omega(n) \). However, we also know that the above algorithm is \( O(n) \). And thus there is no algorithmic gap. Whenever this happens, we write \( \Theta(n) \).

Formally, we can define \( \Theta \) as follows:

**Definition 3.8.3**

We write \( f \in \Theta(g) \), if, and only if,

\[
f \in \Omega(g) \wedge f \in O(g)
\]

**Exercise 3.14**

Show that

\( f \in \Theta(f) \)

**Exercise 3.15**

Prove that if \( f \in \Theta(g) \), then there exist \( A, B \in \mathbb{R}_{>0} \) and \( k \in \mathbb{N} \) such that whenever \( n > k \):

\[
A.g(n) \leq f(n) \leq B.g(n),
\]

and vice versa. This is an alternative definition of \( \Theta \).
3.9 Space Complexity

So far we have only discussed time complexity. It can also be important that we understand the space complexity of a program i.e. how much memory it requires to run. As with time complexity we need to be careful about what we are counting such that it makes sense. For example, it does not make sense to say that the number $2^{30}$ takes more space to store than the number 1 as in most machines both numbers will be stored in a 32-bit word.

For most of the programs we have considered in the while language the space needed is constant as we have a fixed number of variables. In other cases (programs using arrays) the space complexity consists of the space required to store the input and output. But programs that require us to store intermediate results can have non-trivial space complexity. For example, here is a (silly) program that has $O(n)$ space complexity for computing the sum of numbers from 1 to $n$.

```plaintext
for i := 1 to n do (a[0] := i);
sum := 0
for i := 1 to n do (sum := sum + a[i])
```

The program first creates an array containing the numbers from 1 to $n$ and then sums these up.

These days we are often less concerned with space complexity as time complexity for two reasons:

- Space requirements are often less dependent on the size of the input and in many cases are fixed; and
- We have more space than time

However, it is still useful to think about the space complexity of an algorithm. The field of computational complexity theory also spends a reasonable amount of time considering the relationship between time and space complexity, with some interesting results. One of these results is that space complexity cannot be larger than time complexity if we assume that everything that is written to or read from the used space takes some time to do so.

As a final thought, we can always exchange time complexity for space complexity by storing precomputed results. Assuming the time taken to perform lookup in a lookup table is negligible (not the case for large tables) we can turn a time complexity $O(f(n))$ function into time complexity $O(1)$.

---

3 Sometimes the input is not considered part of the space required for a program.
for \( n < k \) where we store the first \( k \) values of \( f(n) \) in a lookup table. But the space complexity of this lookup will be \( O(k) \). In some cases this trade-off may be worthwhile. For a famous example of this look at the idea of Rainbow tables\(^4\) for reversing cryptographic hash functions. And see

https://en.wikipedia.org/wiki/Space-time_tradeoff

for a further discussion of this topic.

3.10 Complexity Classes

In this short section I want to explain the meaning of a very important question in theoretical computer science: \( P = NP? \) We don’t know the answer to this question (mostly people think the answer is no).

The first thing to do is introduce the idea of a complexity class. This is just a set of functions (problems) with the same asymptotic complexity (in some resource). So, by this definition, \( O(n) \) and \( O(n^2) \) are both complexity classes. However, it is useful to have a more general notion than the one introduced by Big-Oh notation.

We define the complexity class \( P \) as the set of all polynomial-time functions as follows:

\[
P = \{ f \mid \exists k : f \in O(n^k) \} = \bigcup_k O(n^k)
\]

i.e. a function \( f \) is in \( P \) if there is some polynomial \( p \) such that \( f \in O(p) \). There are obvious functions that are not in \( P \), for example \( 2^n \notin P \).

We usually call problems in \( P \) tractable and problems not in \( P \) intractable. Intractability means not practically solvable in a reasonable amount of time. Take a program with complexity \( 2^n \) as an example, if \( n = 100 \) and a machine performs \( 10^{12} \) operations per second then the program would run for \( 4 \times 10^{10} \) years, which is the same order of magnitude as the age of the universe. This is too long to wait.

Another class of problems that is useful to think about are those where we can check that the solution is correct in polynomial time. This formulation doesn’t fit into the notation we have so far so we will keep the idea abstract. The idea is that if we can check that the solution to a problem is correct in polynomial time then a non-deterministic machine could compute that function in polynomial time, as it could try all possible solutions non-deterministically and check if each solution is correct, all in polynomial time.

\(^4\)https://en.wikipedia.org/wiki/Rainbow_table
This class of problems is the *Non-deterministic Polynomial* class, written \( NP \). Clearly all problems in \( P \) are also in \( NP \) but it is not clear that the other direction holds.

There are many problems we want to solve that are in \( P \), but equally there are many in \( NP \). Here are some examples:

- **Interesting problems in \( P \):**
  - Calculating the greatest common divisor (our gcd problem)
  - Linear programming
  - Determining if a number is prime
  - Context Free Grammar membership
  - Horn satisfiability

- **Interesting problems in \( NP \):**
  - Boolean satisfiability
  - Chinese postman problem
  - Knapsack problem
  - Graph colouring
  - See also Karp’s 21 NP-complete problems

Finally, a common method for showing that a problem is in a particular class is via *reductions*. If we have a function \( f \in C \) for complexity class \( C \) if we can find another function \( g \in C \) such that \( f \circ g = h \) then we know that \( h \in C \). More concretely, *satisfiability* for propositional logic (where clauses have at least 3 literals) is known to be in \( NP \). We can reduce the *graph colouring* problem (for \( > 2 \) colours) to the satisfiability problem by encoding the problem of colouring a graph as a propositional satisfiability problem. As solving the satisfiability problem also solves our graph colouring problem, and the reduction can be done in polynomial time, we know that graph colouring is also in \( NP \). If you want to learn more about these topics then look at the notions of NP-hardness and NP-completeness.
3.11 Practical Complexity Analysis

Important
This section contains material introduced by David Lester based on his experience with SpiNNaker. I have left it in as it gives a good example of a low-level practical application of complexity analysis. It is not examinable.

One issue that has not been addressed when “counting operations” is that of counting the number of instructions which are involved in just moving data about. If you look at the code produced by practically any compiler you will find a great deal of memory/register and register/register transfers going on. The only way to count these operations – which often have an important bearing on how long a program will take to run – is to compile the code, and only then count how many copy operations occur.

To give an idea of what can happen, I will show what happens when the division program of Exercise 1.2.1 is compiled for SpiNNaker. If the C translation of the program is compiled with gcc using the command line:

```
arm-none-eabi-gcc -march=armv5te -c div.c
```

and then disassembled with objdump using the command line:

```
arm-none-eabi-objdump -Dax div.o
```

we obtain the following disassembly:

```
00000000 <division>:
  0: e92d09f0 push {r4, r5, r6, r7, r8, fp}
  4: e28db014 add fp, sp, #20
  8: e24dd008 sub sp, sp, #8
 c: e50b0018 str r0, [fp, #-24]
10: e50b101c str r1, [fp, #-28]
14: e51b3018 ldr r3, [fp, #-24]
18: e51b201c ldr r2, [fp, #-28]
1c: e0030392 mul r3, r2, r3
20: e3530000 cmp r3, #0
24: ba000001 blt 30 <division+0x30>
28: e3a03001 mov r3, #1
```
CHAPTER 3. COMPLEXITY AND ASYMPTOTIC ANALYSIS

The extra magic required to obtain half-way reasonable code is to use the following extra flag in the command line:

```
arm-none-eabi-gcc -march=armv5te -Ofast -c div.c
```

Now the code becomes:

```
00000000 <division>:
    0: e0130091 muls r3, r1, r0
    4: 43e02000 mvmi r2, #0
    8: 53a02001 movpl r2, #1
c: e1500001 cmp r0, r1
10: e1a03000 mov r3, r0
14: e3a00000 mov r0, #0
```
3.11. PRACTICAL COMPLEXITY ANALYSIS

18: ba000004 blt 30 <division+0x30>
1c: e0613003 rsb r3, r1, r3
20: e1510003 cmp r1, r3
24: e2800001 add r0, r0, #1
28: daafffb ble 1c <division+0x1c>
2c: e0000092 mul r0, r2, r0
30: e3520001 cmp r2, #1
34: 10633001 rsbne r3, r3, r1
38: e1a01003 mov r1, r3
3c: e12fffe bx lr

Not only is this code faster, but at just 16 instructions instead of the original 37 it is also much more compact.

**Important** To ensure that *all* operations impacting on performance are fully taken into account you need to *compile* and then *disassemble* the program. There is no way to just guess how a compiler will compile your code.

**Important** Do *not* obsess about these issues too early. Ideally, you will have demonstrated that your code is correct, ensured that the algorithms chosen are the best, and profiled your code to find the key inner loops. Only then should you inspect the code generated for these key inner loops to check that they are acceptably efficient.