A Level Computing

March 26, 2018

Contents

Part 1

Binary Manipulation

Chapter 1.1

What is Binary?

1.1.1 Introduction

In school, you learnt to count and do arithmetic in *decimal*. Just as you have 10 fingers, there are 10 decimal digits $0, 1, 2, ..., 9$. Decimal digits are joined together to form longer decimal numbers. Each column of a decimal number has ten times the weight of the previous column. From right to left, the column weights are $1, 10, 100, 1000$ and so on. In general we can say that the n^{th} has a value of $10^{(n-1)}$.

$$9742_{10} = 9 \times 10^3 + 7 \times 10^2 + 4 \times 10^1 + 2 \times 10^0 \tag{1.1.1}$$

We can generalise this and say for a number n in base b which has digits $n_1, n_2, n_3, \ldots n_{m-1}, n_m$ has a value of N which is given by Equation 1.1.2

$$N = \sum_{i=0}^{i=m} n_i \times b^i \tag{1.1.2}$$

Computers are made up of lots of little tiny electrical switches that have two states: one or off. We represent this using a binary number system. We use the digits 0 or 1 to mean off and on respectively Each digit is known as a bit and is a fundamental unit of information. We find that using a n-digit binary number we can represent 2^n different values. Table 1.1.1 shows this in action.

1.1.1.1 Grouping Bits

A group of eight bits is called a *byte*. One byte can take $2^8 = 256$ possible values. Generally we measure the size of computer storage / memory in bytes

Binary Numbers				Decimal Equivalent
1-Bit	2-Bit	3-Bit	4-Bit	
0	00	000	0000	0
1	01	001	0001	1
	10	010	0010	2
	11	011	0011	3
		100	0100	4
		101	0101	5
		110	0110	6
		111	0111	7
			1000	8
			1001	9
			1010	10
			1011	11
			1100	12
			1101	13
			1110	14
			1111	15

Table 1.1.1: Binary Numbers and their Decimal Equivalent

rather than bits. We can also store data in a nibble, which is 4 bits. This gives you $2^4 = 16$ possible values, or one hexadecimal digit. Often when dealing with binary outputs we represent each byte as two hexadecimal digits. So `11010010` would be represented as `D2`.

Microprocessors handle data in chunks known as *words*. The size of the word depends on the architecture used. Most computers these days use 64-bit computers, so they have 64-bit words. However, Arduino's use 8 bit words.

1.1.1.2 File Sizes

A video file might contain 1,039,297,516 bytes, this is clearly to big to be practical. There are two different schemes for simplifying bytes, one uses powers of 10 and the other powers of 2. The decimal prefixes follow the standard SI units: kilo, mega, giga etc. The binary prefixes follow the following pattern:

kibi Ki - 2^{10}

mebi Mi - 2^{20}

gibi Gi - 2^{30}

tebi Ti - 2^{40}

So our video file from earlier is 1.039 29 GB or 991.15 MiB. The difference between a KiB and kB is why when you install a 1 TB hard drive it appears as 931 GiB.

1.1.2 Exercises

1.1.2.1 Exam Style Questions

1. A flying saucer crashes into a cornfield. The investigators inspect the wreckage and find an engineering manual contain an equation in the Martian number system: $325 + 42 = 411$. If this equation is correct, how many fingers would you expect Martians to have?

2. Alice and Bob are having an argument. Bob says 'All integers greater than zero and exactly divisible by six have exactly two 1's in their binary representation'. Alice disagrees, she says: 'No, but all such numbers have an even number of 1's in their representation.' Do you agree with Alice, Bob, both or neither. Explain.

1.1.2.2 Programming Challenges

1. Write a program in Python to convert numbers from binary to decimal. The user should type in an unsigned binary number. The program should printout the decimal equivalent

   BONUS: Change your program from an arbitrary base b_1 to another base b_2 as specified by the user. Support bases up to 16, using the letters of the alphabet for digits greater than 9.

Chapter 1.2

Boolean Logic

1.2.1 Boolean Algebra

Boolean Algebra was developed by an English Mathematician who specialised in the field of Logic. In 1854 he published *'An Investigation of Laws and Thought' (Boole, 1854)* which layed out the framework. Formally, boolean algebra is defined as of a set of elements, E, a set of functions, F and and a set of basic axioms that define the properties of E and F. In Boolean Algebra the set of elements are variables and constants that can either have a value of `true` or `false` . In addition there are only three operations or functions that are permitted. The first of these is the logical OR , represented by a plus (e.g. $A + B$ [1]). The Logical OR operation returns `true` if either of the values is also `true` . For example $A + B$ will return `true` if either A or B is `true` . Next we have the logical AND , represented by a dot (e.g. $A \cdot B$ [2]). The Logical AND operation returns `true` if all of the values are true will return The third operation is that of negation or complementation, we denote a variable having been negated with a bar (e.g. $\bar{A}$) [3] .

We can combine these operations together to make new operations. As example lets take the exclusive-OR, or XOR function . We define $A \oplus B$ [4] as returning `true` when one, and only one, of its inputs are `true` . The XOR operation can also be represented as

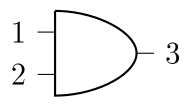
$$A \oplus B = A \cdot \bar{B} + \bar{A} \cdot B$$

[1]Some texts will also use $\cup$ or $\vee$

[2]Some texts will also use $\times$, $\cap$ or $\wedge$. Also note that like in Maths, you can write $A \cdot B$ as AB.

[3]Some texts will also use !, , $\neg$, '

[4]Some texts will also use $\veebar$



sf

As mentioned in the previous chapter bits are the fundamental unit of the computer. This stream of 0's and 1's are what allow you to watch videos online, post to social media and order 200 rubber ducks from the internet. In this chapter we will layout the basic ways that we manipulate data in a binary format.

1.2.2 Logic Gates

Logic gates are simple digital circuits that take one or more binary inputs and produce a binary output. Usually inputs are drawn with a symbol showing the input and the output.

1.2.2.1 NOT Gate

A NOT gate, has one input A and one output, Y

1.2.3 Karnaugh Maps

Sometimes when you are simplifying a boolean equation you end up with a totally different equation instead of a simpler one. Karnaugh Maps were invented in 1953 by Maurice Karnaugh are a graphical method for simplifying Boolean equations. In order to understand how they work you need to recall that when we are minimising a logic problem we are grouping like terms together.

1.2.4 Computer Arithmetic

1.2.5 Number Systems

Part 2

Manipulating Data

Chapter 2.1

Data Types

2.1.1 Integers

2.1.1.1 Number Bases

Decimal

Binary

2.1.1.2 Hexadecimal

2.1.1.3 Negative Numbers

Signed Form

Two's Complement

2.1.2 Decimals

2.1.2.1 Fixed Point Form

2.1.2.2 Floating Point Form

2.1.2.3 Normalisation

2.1.2.4 Rounding Errors

2.1.3 Boolean

2.1.4 Characters

2.1.4.1 ASCII

2.1.4.2 Unicode

2.1.5 String

2.1.6 Date & Time

Chapter 2.2

Arithmetic Operations

Binary Arithmetic Follows mostly the same rules as decimal arithmetic, just in a different base. [1]

[1] This is true of doing maths in any base, the rules still apply just remember what value you round at

2.2.1 Addition

2.2.2 Subtraction

2.2.3 Multiplication

2.2.4 Division

2.2.4.1 Integer Division

2.2.4.2 Float Division

2.2.5 Exponentiation

2.2.6 Rounding

2.2.7 Truncation

2.2.8 Random Number Generation

Chapter 2.3

Storing Data

2.3.1 Analogue and Digital Systems

2.3.1.1 Digital to Analogue Conversion

2.3.1.2 Analogue to Digital Conversion

2.3.2 Images

2.3.2.1 Resolution and Colour Depth

2.3.2.2 Metadata

2.3.2.3 Vector Graphics

SVG

2.3.3 Sound

2.3.3.1 Sample Resolution and Rate

2.3.3.2 Nyquist Theorem

2.3.3.3 MIDI

2.3.4 Errors in Data

2.3.4.1 Error Detecting Codes

Parity EDCs

2.3.4.2 Error Correcting Codes

Hamming Codes

2.3.5 Data Encryption

2.3.5.1 Caesar Cipher

example before going into more depth. Lets say a weather station in the Scottish highlands sends a daily report using a binary code. At the moment it uses the 2-bit codes shown in Table 2.3.1

Weather	**Code**
Raining	`00`
Windy	`01`
Snowing	`10`
Sunny	`11`

Table 2.3.1: Coding four states with a 2-bit code

At first glance this would appear to be as good as you can get. After all theres no way to encode 4 items using just one bit. However we can make use of probability to reduce the *average* length sent. If you make the observations that far most common report is that it is raining, you can assign as small as possible code to [1]

Weather	**Frequency**	**Code**
Raining	75 %	`0`
Windy	15 %	`10`
Snowing	5 %	`110`
Sunny	5 %	`111`

Table 2.3.2: A Huffman code for four items.

Run Length Encoding

Run length compression is a form of compression that works best when you have long runs of repeated data. Lets take an example string:

[1]Note that you can't just assign

Lempel-Ziv

Quadtrees

2.3.6.2 Lossy Compression

Lossy compression algorithms

JPEG Compression

Chapter 2.4

Classification of Algorithms

Algorithms are the most important and durable part of computer science. However in order to use them effectively we need a way of evaluating efficiency without implementing them in software.

Imagine you had an problem that involved taking a list of items and spat them out after doing some arbitrary processing. You have a selection of algorithms that would be appropriate for this task, but how would you evaluate its performance? The simplest way would be implement each to run it over every possible combination of and measure how long it took. There are two issues with this approach, the first is that the majority of resources used in the implementation of the algorithms are wasted once one is selected. The second issue is that even for one algorithm, we have an infinite number of possible inputs. Lets say our list contains one item, there are an infinite number of values just between 0 and 1. Even if we limit ourselves to only using the numbers $1...n$ where n is the number of inputs required we quickly run into the other issue that the number of possible combinations for this set is $n!$. For just 10 items there are $3,628,800$ combinations that we will have to try. Clearly this is unworkable when in most applications we are talking about data sets on the order of thousands. Clearly we need a method of being able to study algorithms without actually implementing them and testing them.

2.4.1 Best, Worst and Average Case Complexity

Complexity is a measure of how a problem grows as the size increases . If we take a problem and run it over all possible arrangements of n keys we can

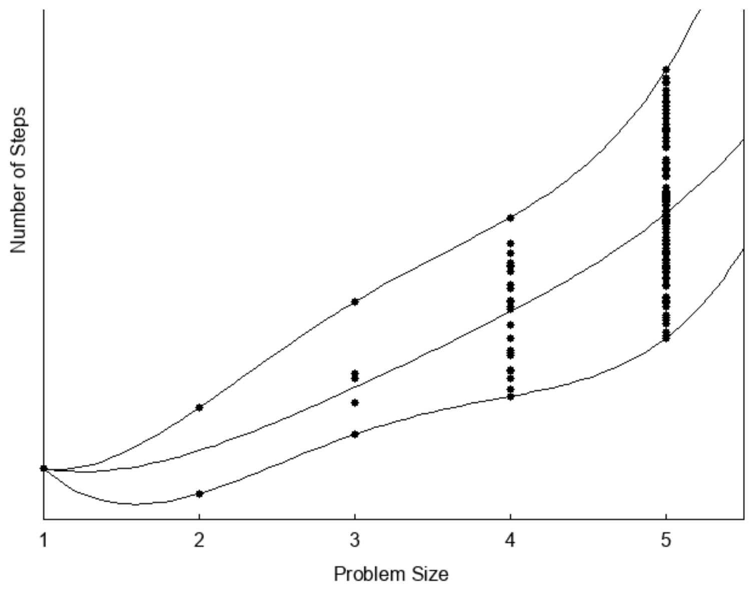


Figure 2.4.1: Best, worst and average-case complexity

plot a graph (see fig. 2.4.1) where the x-axis represents the size of the input problem and the y-axis denotes the number of steps taken by the algorithm in this instance. We can then define three interesting functions over the plot of these points

- The *worst-case complexity* of the algorithm is the function defined by the maximum number of steps taken in any particular instance of size n. This represents the urver passing through the highest point in each column.
- The *best-case complexity* is the function defined by the minimum number of steps take in any instance of size n. This represents the curve passing through the lowest point of each column.
- The *average-case complexity* of the algorithm, which is the function defined by the average number of steps over all instances of size n.

The worst-case complexity turns out to be the most useful of these measures in practice. At first glance this might seem counter-intuitive Perhaps the best analogy is think about what happens if you bring £n into a casino to gamble. The best case is that you walk out owning the place, it is possible but is so unlikely that you should not even think about it. The worst case, that you lose all n dollars is easy to calculate and distressingly likley to happen. The average case, that the typical bettor loses 87.32 % of the money that they bring into the casino, is difficult to establish and its meaning is subject to debate. What exactly does *average* mean? Stupid people loose more that smart people, so are you smarter or stupider than the average person, and by how much? Card counters at blackjack do better on average than customers that accept three or more free drinks. It is far easier to avoid all of these complexities and obtain a very useful result just by considering the worst case.

The last important thing to realise is that each of these time complexities define a numerical function, representing time versus problem size. These functions are as well defined as any other numerical function, be it $y = x^2 - 2x + 1$ or the price of a particular stock as a function of time. However time complexities are such complex functions that we must simplify them to work with. For this we need 'Big Oh' notation.

2.4.2 Big Oh Notation

The issue with using numerical functions when doing complexity analysis tends to be two fold:

- *They have two many bumps* - An algorithm such as binary search typically runs a bit faster for arrays of size $n = 2^k - 1$ (where k is an integer), because the array partitions works out nicely. Whilst this detail is not particularly significant, it does mean that the *exact* complexity function is likely to be very complicated, with little up and down bumps.
- *Requires too much detail to specify precisely* - Counting the exact number of RAM instructions executed in the worst case requires that the algorithm be specified to the detail of a complete computer program. This means that individual foibles of the programming language start to creep in, e.g. the peformance of a `case` statement in C vs a nested if in Python would start to play a role.

2.4.2.1 Formal Definition

The formal definition of Big Oh states that $f(n) = \mathcal{O}(g(n))$ means $c \cdot g(n)$ is an *upper bound* on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\leq c \cdot g(n)$, for a large enough n (i.e., $n \geq n_0$ for some constant n_0). [1]

[1] We should also add for completeness there is also a function $\Omega(g(n))$ which provides a lower bound, and $\Theta(g(n))$ that exists bounded between $\Omega(g(n))$ and Big Oh.

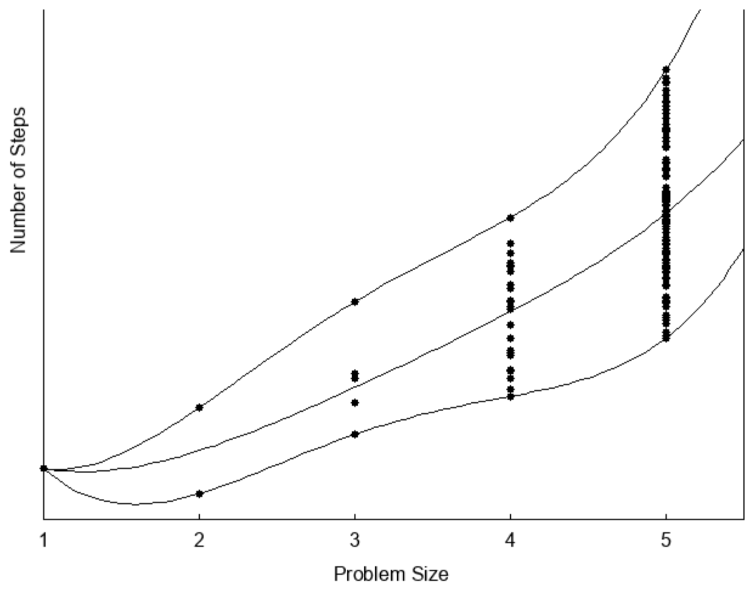


Figure 2.4.2: Illustrating the Big Oh notations

But what does that actually mean? First it assumes a constant n_0 beyond which they are always satisfied. This means we are not concerned about small values of n. This seems reasonable on two fronts: first, most programs take about the same time for $n \leq 10$ so we are not to concerned with performance; second, there more significant fluctuations that exist for low n. We are more concerned how our algorithm behaves at $n = 10\,000$ and $n = 100\,000$. Second it is not concerned with any multiplicative constants. The function $f(n) = 2n$ and $g(n) = n$ are identical in big Oh analysis for different values of c. Figure 2.4.2 shows how we can draw an upper and lower bound on some function $f(n)$.

As an example lets take the function $3n^2 - 100n + 6$, which we've plotted in 2.4.3, and try and work out what its big Oh notation is.

$$3n^2 - 100 + 6 = \mathcal{O}(n^2), \text{for } c = 3 \because 3n^2 > 3n^2 - 100n + 6 \tag{2.4.1}$$

$$3n^2 - 100 + 6 = \mathcal{O}(n^3), \text{for } c = 1 \because n^3 > 3n^2 - 100n + 6 \text{ when } n > 3 \tag{2.4.2}$$

$$3n^2 - 100 + 6 \neq \mathcal{O}(n), \forall c \because c \cdot n < 3n^2, \text{when } n > c \tag{2.4.3}$$

The slightly confusing part is when you see the expression $n^2 = \mathcal{O}(n^3)$, but if you go back to the definitions and remember that Big Oh is defined in terms of an *upper bound.* This means that we can read the '=' here as meaning *one of the functions that are.* Clearly n^2 is one of the functions that are $\mathcal{O}(n^3)$.

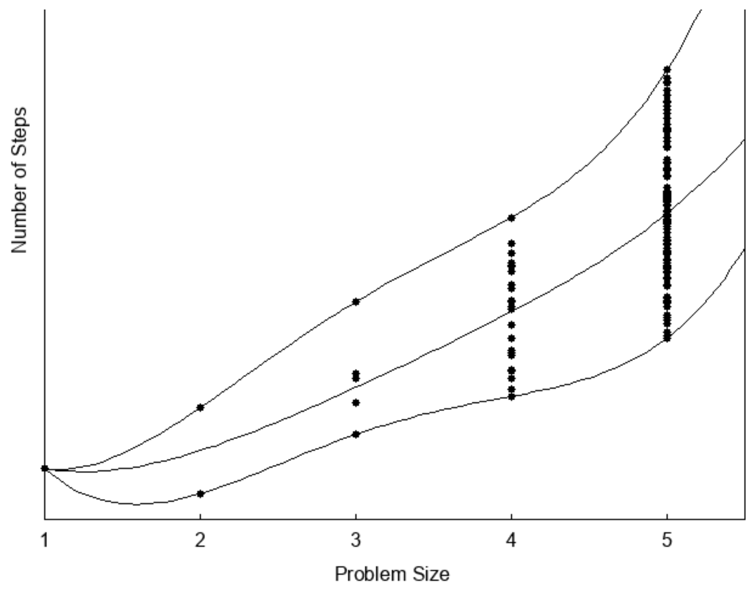


Figure 2.4.3: Fitting a function to f(n) for a variety of constants

2.4.2.2 Grow Rates and Dominance Relations

With the Big Oh notation, we discard the multiplicative constants, thus the functions $f(n) = 0.001n^2$ and $g(n) = 1000n^2$ are treated identically, even though $g(n)$ is a million times larger than $f(n)$ for all values of n. The reason why we are content with the fairly coarse analysis provided in Table 2.4.1, which shows the growth rate of several common time analysis functions. In particular it shows how long algorithms that use $f(n)$ operations take to run on a computer that executes one instruction in 1 ns (e.g. 1 GHz). We can draw the following conclusions:

- All such algorithms take roughly the same time for $n = 10$
- Any algorithm with $n!$ running time becomes useless for $n \geq 20$.
- Algorithm whose running time is 2^n have a greater operating range, but become impractical for $n > 40$.
- Quadratic-time algorithms remain usable up to about $n = 10\,000$, but quickly deteriorate with larger inputs.
- Linear-time and $n \lg n$ algorithms remain practical on inputs of one billion item.

- A $\mathcal{O}(\lg n)$ algorithm hardly breaks a sweat for any imaginable value of n.

The bottom line is that despite ignoring constant factors, we get an excellent idea of whether a given algorithm is appropriate for a problem of a given size.

Dominance Relations

We can group Big notation into a set of classes, such that all the functions in a particular class are equivalent in respect to Big Oh. We say that a faster growing function *dominates* a slower-growing one. Below we shall list the most common function classes in order of increasing dominance.

Constant functions, $f(n) = 1$ Such functions might measure the cost of adding two numbers, printing out the lyrics to a song, or inserting into a sorted array. There is no dependence on the size of n.

Logarithmic functions, $f(n) = \log n$ This turns up in algorithms such as binary search. They grow quite slowly as n gets big but faster than the constant function.

Linear functions, $f(n) = n$ Such functions measure the cost of looking at each item once (or twice, or ten times) in an n-element array, say to compute the average value.

Superlinear functions, $f(n) = n \log n$ This class of functions arises in algorithms such as Quicksort and Mergesort.

Quadratic functions, $f(n) = n^2$ These measure the cost of looking at most or all *pairs* of items in an n-element universe. This arises in algorithms such as insertion and selection sort.

Cubic functions, $f(n) = n^3$ These functions enumerate over all triples of items in an n-element universe.

Exponential functions, $f(n) = c^n \forall c > 1$ Functions like 2^n arise when enumerating all subsets of n items.

Factiorial functions, $f(n) = n!$ Functions like this arise when generating all permutations or orderings of n items.

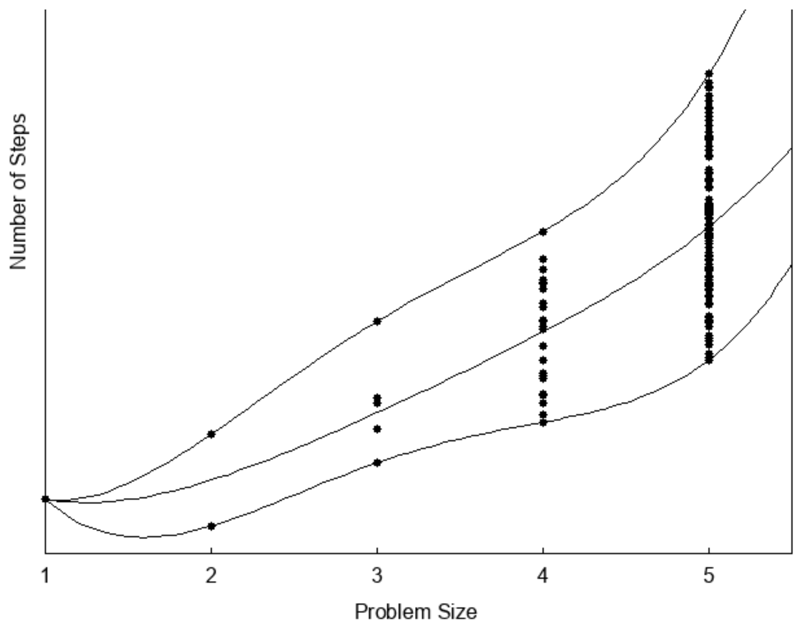


Figure 2.4.4: The growth rates of various common functions

Figure 2.4.4 shows these graphically. We'll discuss the importance of dominance relationships in the next section. However the important thing to understand is:

$$n! \gg c^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \log n \gg 1 \tag{2.4.4}$$

As a shorthand we often refer to problems that have polynomial or less solutions are known as traceable problems and ones with greater than polynomial solutions are called in-traceable.

2.4.2.3 Working with the Big Oh

When reasoning what Big Oh class different

Adding Functions

The sum of two functions is governed by the dominant one, i.e

$$\mathcal{O}(f(n)) + \mathcal{O}(g(n)) \to \mathcal{O}(\max(f(n), g(n))) \tag{2.4.5}$$

Multiplying Functions

There are two types of multiplication that we need to worry about with Big Oh, the first is with a constant factor, the second is with a second function.

n	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 µs	0.01 µs	0.033 µs	0.1 µs	1 µs	3.63 ms
20	0.004 µs	0.02 µs	0.086 µs	0.4 µs	1 ms	77.1 yrs
30	0.005 µs	0.03 µs	0.147 µs	0.9 µs	1 s	8.4×10^{15} yrs
40	0.005 µs	0.04 µs	0.213 µs	1.6 µs	18.3 min	
50	0.006 µs	0.05 µs	0.282 µs	2.5 µs	13 d	
100	0.007 µs	0.1 µs	0.644 µs	10 µs	4×10^{13} yrs	
1000	0.010 µs	1 µs	9.966 µs	1 ms		
10 000	0.013 µs	10 µs	130 µs	100 ms		
100 000	0.017 µs	0.10 ms	1.67 ms	10 s		
1 000 000	0.020 µs	1 ms	19.93 ms	16.7 min		
10 000 000	0.023 µs	0.01 s	0.23 s	1.16 d		
100 000 000	0.027 µs	0.1 s	2.66 s	115.7 d		
1 000 000 000	0.030 µs	1 s	29.90 s	31.7 yrs		

Table 2.4.1: A Comparison of the worst case performances of different implementations of a Dictionary for different operations.

With a constant the answer is simple, you just ignore it as in Equation 2.4.6. Obviously this doesn't hold for $c \geq 0$ as multiplying even $\mathcal{O}(n!!)$ is quickly wiped out if you multiply it by zero, but for most cases we will come across this will not be the case.

$$\mathcal{O}(c \cdot f(n)) \to \mathcal{O}(f(n)) \tag{2.4.6}$$

However for when you are multiplying a function by another function both results are important. This leads to the result in Equation 2.4.7.

$$\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) \to \mathcal{O}(f(n) \cdot g(n)) \tag{2.4.7}$$

e.g.

$$\mathcal{O}(n^2) \cdot \mathcal{O}(\log n) \to \mathcal{O}(n^2 \log n) \tag{2.4.8}$$

2.4.2.4 Example Derivation

In this section we are going to reason our way through some common algorithms and see if we can derive the big Oh notation for a given algorithm.

String Pattern Matching

Pattern matching is the most fundamental algorithm that we perform on text strings. This algorithm implements the find command available in any web browser or text editor.

Problem: Substring Pattern Matching *Input:* A text string t and a pattern string p. *Output:* Does t contain the pattern p, and if so where?

```
int findmatch(char *p, char *t) {
        int i, j;               /* counters */
        int m, n;               /* string lengths */

        m = strlen(p);
        n = strlen(t);

        for (i=0; i <= (n - m); i++) {
                j = 0;
                while((j < m) && (t[i+j] == p[j])) {
                        j = j+1;
                }
                if (j == m) {
                        return(i);
                }
        }

        return(-1);
}
```

Figure 2.4.5: Algorithm for finding a matching string.

What is the worst-case running time of these two nested loops? The inner *while* looop goes around at most m times, and potentially far less when the pattern match fails. The outer loop goes around at most $n-m$ times, since no complete match can be found once there are not enough characters left. With nested loops we multiply the complexity [2] so we get a worst case complexity

[2]This is because every time we run the outer the loop we run the complete inner loop

given in equation 2.4.9, the +2 refers to the two extra statements that are within the outer *for* loop.

$$\mathcal{O}((n-m)(m+2)) \tag{2.4.9}$$

We also need to consider the time taken to find the length of each of the strings. Lets just assume that we explicitly count the number of characters until we hit the end of the string, this suggests that it would take linear time. So the total running time would be given in equation 2.4.10

$$\mathcal{O}(n+m+(n-m)(m+2)) \tag{2.4.10}$$

Simplifying the constant term,

$$\mathcal{O}(n+m+(n-m)m) \tag{2.4.11}$$

Multiplying it out,

$$\mathcal{O}(n+m+nm-m^2) \tag{2.4.12}$$

We know that $n \geq m$ because a sub string that longer than the original text would be impossible. So,

$$n+m \leq 2n \rightarrow \mathcal{O}(n) \tag{2.4.13}$$

Substituting 2.4.13 in leaves:

$$\mathcal{O}(n+nm-m^2) \tag{2.4.14}$$

We also note that $n \leq nm$, since $m \geq 1$ for any interesting problem thus:

$$n+nm \rightarrow \mathcal{O}(nm) \tag{2.4.15}$$

This allows us to drop the additive n and leave us with Equation 2.4.16

$$\mathcal{O}(nm-m^2) \tag{2.4.16}$$

Finally we observe that the $-m^2$ term is negative so only serves to lower the value within. Because Big Oh is an upper bound, we can drop any negative without invaliding the output. We also know that because $n \geq m$, $mn \geq m^2$ so it isn't big enough to cancel out any term that is left. This leaves a worst case complexity of $\mathcal{O}(nm)$

```
for (i = 1; i <= x; i++) {
        for (j = 1; j <= z; j++) {
                C[i][j] = 0;
                for (k = 1; k <= y; k++) {
                        C[i][j] += A[i][k] * B[k][j];
                }
        }
}
```

Figure 2.4.6: Multiplying two matrices, A and B to get C.

Matrix Multiplication

Matrix multiplication is a fundamental operation in linear algebra. *Problem:* Matrix Multiplication *Input:* Two matrices, A (dimensions $x \times y$) and B (dimensions $y \times z$) *Output:* A $x \times z$ matrix C where $C[i][j]$ is the dot product of the ith row of A and the jth column of B.

The number of multiplication operations is given by $M(x, y, z)$ which is the result of a nested summation shown in 2.4.17

$$M(x,y,z) = \sum_{i=1}^{x} \sum_{j=1}^{y} \sum_{k=1}^{z} 1 \tag{2.4.17}$$

We evaluate from the right inward. The sum of z ones is z so:

$$M(x,y,z) = \sum_{i=1}^{x} \sum_{j=1}^{y} z \tag{2.4.18}$$

$$M(x,y,z) = \sum_{i=1}^{x} yz \tag{2.4.19}$$

$$M(x,y,z) = xyz \tag{2.4.20}$$

Thus the running of a matrix multiplication is $\mathcal{O}(xyz)$, for the case that $x = y = z$ we get $\mathcal{O}(n^3)$ i.e. a cubic algorithm.

Binary Search and Logarithms

Binary Search is a good example of an $\mathcal{O}(\log n)$ algorithm. It is covered in more detail in section 2.6.1.2 but for now imagine you have a telephone book. You want to locate person, p in a telephone book containing n names. In a binary search you start by comparing it to the middle or $^n/_2$ name, say

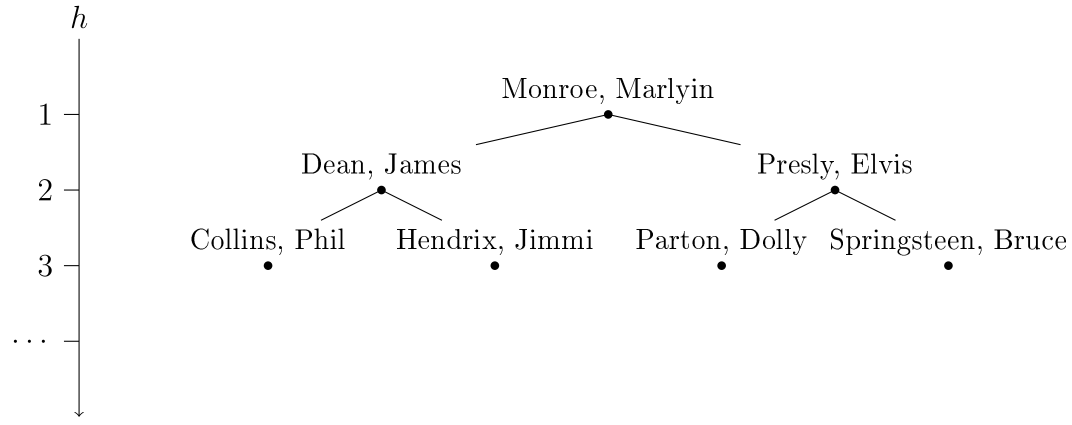


Figure 2.4.7: the results of a possible Binary Search after h comparisons.

Monroe, Marilyn. You then discard half the book depending if the name came before or after middle. You repeat this process until you get to having only one name left. The important question for our analysis is how many steps will it take to get down to this one name? Lets start by turning the problem on its head. After h comparisons how many different values could you have compared? For $h = 1$ the answer is trivial - 1. For $h = 2$, we have the first element that we compared and the two either that we jump to next. Fig 2.4.7 demonstrates how the number of items grows for h. We can work out that after h jumps we can cover $2^h - 1$ items. Equation 2.4.21 shows how this is derived to get a complexity of $\mathcal{O}(\lg n)$

$$n = 2^h - 1 \tag{2.4.21}$$
$$\lg n = \lg(2^h) - \lg 1 \tag{2.4.22}$$
$$\lg n = h \tag{2.4.23}$$

2.4.3 Greedy Algorithms

2.4.4 P vs NP Problems

One of the most profound open problems in computer science asks 'Is $P = NP$?'. In simple language it is asking whether *verification* is easier than initial *discovery*. A good analogy is lets say during the course of an exam you happen to notice the answer of the student next to you. Are you now better off? You wouldn't turn it in without checking, since you could answer the question correctly if you took enough time to find it from scratch. The question is whether you can really verify the answer faster than you could find it from scratch.

In particular we are referring to two classes of problems: P and NP. The class P is an exclusive club that can only be joined whence a problem has demonstrated that there is a polynomial-time algorithm to solve it. A less-exclusive club welcomes all problems who can be *verified* in polynomial time. This obviously includes all members P. We call this club NP standing for *non-deterministic polynomial time.* The question is whether problems in NP that cannot be in P. Most people who actually study this field believe that $P \neq NP$ but haven't found a proof yet.

2.4.5 The Halting Problem

So far all the problems that we have covered

2.4.6 Turing Machines

2.4.7 State Transition Diagrams

2.4.8 Exercises

2.4.8.1 Exam Style Questions

1. What value is returned by the following function? Express your answer as a function of n. Give the worst-case running time using the Big Oh notation

Chapter 2.5

Data Structures

Data structures

We can class Data structures into two distinct category's based on whether or not they use pointers:

- *Contiguously-allocated structures* are composed of single slabs of memory, and include arrays, matrices, heaps and hash tables
- *Linked data structures* are composed of distinct chunks of memory bound together by *pointers* and include lists, trees and graph adjacency lists.

2.5.1 Arrays

The *array* is the fundamental contiguously-allocated data structure. Arrays are structures of fixed-size data records such that each element can be efficiently located by its *index*.

There are several advantages to using contiguously-allocated arrays:

Constant-time access given the index - Because the index of each element maps directly to a particular memory address, we can access arbitray data tiems instantly provided we know the index.

Space Effciency - Arrays consist purely of data, sono space is wasted with links or other formatting information. Further, end-of-record information is not needed because arrays are built from fixed-size records

Memory locality - A common programming idiom involves iterating through all the elements of a data structure. Arrays are good for this because they exhibit excellent memory locality. Physical continuity between successive *cache memory* in modern computer architectures

The disadvantages of arrays is that we cannot adjust their size in the middle of a programs' execution. Our program will fail soon as we try to add the $(n+1)$ customer, if we have only allocated room for n records. We could compensate by allocating extremely large arrays, but this can waste space, restricting what our programs actually do.

2.5.1.1 Dynamic Arrays

Dynamic Arrays are a way around the restriction placed by arrays being fixed size. With a dynamic array we enlarge arrays as we need them. Suppose we start with an array of size 1, and double its size from m to $2m$ each time we run out of space. This doubling process involves allocating a new array of size $2m$, copying the contents of the old array to the lower half of the new one, and returning the space used by the old array to the storage allocation system. Snippet. 2.5.1 shows an implementation that takes an array and a size and returns a new array.

```
int* double_array(int *array, int size) {
        int i;
        int* new_array = malloc(sizeof(int) * 2 * size);
        for(i = 0; i < size; i++) {
                new_array[i] = array[i]
        }
        free(array);
        return new_array;
}
```

Snippet 2.5.1: Implementing a dynamic array in C

On average each of the n elements in the array only move $2n$ times, so the total work is the on the order of $\mathcal{O}(n)$. Adding a new element to an array no longer takes constant time *in the worst case* instead all querys will be fast, except for those relatively few queries triggering array doubling.

2.5.2 Linked Lists

Linked lists are one of the fundemental data structures which are often extended to make complex structures. All linked data structures share certain properties as shown in snippet 2.5.2. In particular:

```
typedef struct list {
        item_type  item; // data item
         struct list *next; // pointer to successor
}
```

Snippet 2.5.2: Linked List declaration

- Each node in our data structure (here `list`) contains one or more data fields (here `item`) that retain the data that we need to store
- Each node contains a pointer field to at least one other node (here `next`). This means that much of the space used in linked data structures has to be devoted to pointers, not data.
- Finally, we need a pointer to the head of the structure so that we know where to start.

The list is the simplest linked structure. The three basic operations supported by lists are searching, insertion and deletion. In doubly-linked lists, each node points to both its successor and its predecessor

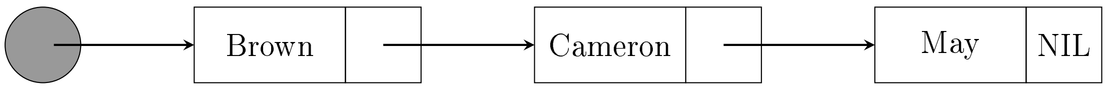


Figure 2.5.1: a singly linked list for the 3 most recent prime minsters

2.5.2.1 Searching a List

Searching for item x in a linked list can be done either utilising iteration or recursion. Snippet. 2.5.3 is an example of a recursive implementation. If x is in the list it is either in the first item or it is located in the smaller rest of the list. Eventually we will reach a point where we have an empty list. Clearly x cannot be there so we can return $NULL$.

2.5.2.2 Insertion into a list

Insertion into a list is. We have no particular need at this moment to maintain the list in any particular order so we can instead choose to insert each new item where ever is easiest. In a singly linked list this is the beginning as it avoids any need to traverse the list, but does require us to update the

```
list* search_list(list *l, item_type x) {
        if (l == NULL) {
                return(NULL);
        }

        if (l->item == x) {
                return(l);
        } else {
                return( search_list(l->next, x));
        }
}
```

Snippet 2.5.3: Searching for item x in list l

pointer at the end of the list, denoted l. Snippet. 2.5.4 shows the pointer manipulation required. Just a quick note, the double star (`**l`) denotes that l is a *pointer to a pointer* to a list node. Thus the last line, `*l=p;`, copies p to the place pointed to by l, which is the external variable maintaining access to the head of the list.

```
void insert_list(list **l, item_type x) {
        list *p; /* temporary pointer */
        p = malloc( sizeof(list));
        p-> item = x;
        p-> next = *l;
        *l = p;
}
```

Snippet 2.5.4: Inserting item x into list l

2.5.2.3 Deletion from a List

Deletion from a linked list is a bit more complicated than insertion. First we don't have the luxury of just deleting where ever is easiest. Instead we have to search to find the pointer to the *predeccessor* of the item to be deleted. Snippet. 2.5.5 shows how we can use recursion to find the necessary pointer.

Now that we have found the pointer to the doomed node, the actual deletion operation is simple, once ruling out the case that the to-be-deleted element doesn't exist. Note that special care should be taken to reset the

```
list predecessor_list(list *l, item_type x) {
        if((l == NULL) || (l->next == NULL)) {
                retur(NULL);
        }

        if ((l->next)->item == x) {
                return(l);
        } else {
                return( predecessor_list(l->next, x));
        }
}
```

Snippet 2.5.5: Finding the predecessor to item x in list l

pointer to the head of the list (l) when the first element is deleted.

```
void delete_list(list **l, item_type x) {
        list *p;                          /* item pointer */
        list *pred;                   /* predecessor pointer */
        list *search_list(), *predecessor_list();

        p= search_list(*l, x);
        if (p != NULL) {
                pred = predecessor_list(*l, x);
                if (pred == NULL) {
                        *l = p->next; /* Case when x is head */
                } else {
                        pred->next = p->next;
                }
                free(p);
        }
}
```

Snippet 2.5.6: Deleting item x from list l

2.5.3 Comparison of Arrays and Linked Lists

The relative advantages of linked lists over static arrays include:

- Overflow on linked structures can never occur unless memory is actually full.
- Insertions and deletions are simpler than for contiguous (array) lists.
- With large records, moving pointers is easier and faster than moving the items themselves

The relative advantages of arrays include:

- Linked structures require extra space for storing pointer fields.
- Linked lists do not allow efficient random access to items
- Arrays allow better memory locality and cache performance than random pointer jumping.

2.5.4 Fields, Records and Files

2.5.4.1 Fields and Records

2.5.4.2 Text Files

CSV

XML

JSON

2.5.5 Binary Files

2.5.6 Abstract Data Types

2.5.7 Stacks and Queues

Stacks and Queues are often called *containers*, denoting that they are data structure that permits storage and retrieval of data items *independent of content*. We distinguish between containers by the particular retrieval order that they support.

Stacks Support retrieval by last-in, first-out (LIFO) order. Stacks are simple to implement and very efficient For this reason stacks are probably the best container to use when retrieval order doesn't matter. The *put* and *get* operations are often called *push* and *pop*.

Queues Support retrieval by first in, first out (FIFO) order. This is usually the fairest way to control waiting times for services. You want the container holding jobs ina FIFO order to minimize the *maximum* time spent waiting. Queues are somewhat trickier to implement than stacks and thus are more appropriate for applications where the order is important. The *put* and *get* operations are often called *enqueue* and *dequeue.*

Stacks and queues can be effectively implemented using either arrays or linked lists. The key issue is whether an upper bound on the size of the container is known in advance.

2.5.8 Dictionaries

The *dictionary* data type permits access to data items by content. You stick an item into a dictionary so you can find it where you need it. The primary operations of dictionary support are:

- *Search(D,k)* - Given a search key, k, return a pointer to the element in dictionary D whose key value is k, if one exists.

- *Insert(D,x)* - Given a data item x, add it to the set in the dictionary D.

- *Delete(D,k)* - Given a pointer to a given data item x in the dictionary D, remove it from D.

Certain dictionary data structues also efficently support other useful operations:

- *Max(D) or Min(D)* - Retrieve the item with the largest (or smallest) key from D. This enables the dictionary to serve as a priority queue.

- *Predecessor(D,x) or Successor(D,x)* - Retrieve the item from D whose key is immediately before (or after) x in sorted order. This allows us to iterate through the elements of the data structure.

2.5.8.1 Using a List or an Array?

In this section we will look at the advantages and disadvantages or using arrays or linked list for implementing a Dictionary. There are more complicated approaches using binary search trees and hash tables which we will come across shortly. Table 2.5.1 shows a comparison of the worst case performance of the basic dictionary operations. We will discuss the results of this and the implementations of each of the operations to fully grasp it.

Dictionary Operations	Unsorted Array	Sorted Array	Singly Unsorted List	Doubly Unsorted List	Singly Sorted List	Double Sorted List
$Search(L, k)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
$Insert(L, x)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
$Delete(L, x)$	$\mathcal{O}(1)$*	$\mathcal{O}(n)$	$\mathcal{O}(n)$*	$\mathcal{O}(1)$	$\mathcal{O}(n)$*	$\mathcal{O}(1)$
$Successor(L, x)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
$Predecessor(L, x)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$*	$\mathcal{O}(1)$
$Max(L)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
$Min(L)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$*	$\mathcal{O}(1)$

Table 2.5.1: A Comparison of the worst case performances of different implementations of a Dictionary for different operations.

Search For an unsorted array A search is implemented by testing k against (potentially) each element in the array [Linear Search]. In an sorted array you can then use a Binary search giving a worst case performance of $\log n$. In a liked list sorting doesn't provide the same benefits. Binary Search is no Longer possible, because we can't access the middle element without traversing all the elements before it. However with a sorted list we can quickly terminate the search for items that are not in the array. For example if you were searching for $Major, John$ and reached $May, Theresa$ we can deduce that he doesn't exist in the dictionary. However in the worst case, say searching for $Zebra$ it still must take linear time.

Insertion For the unsorted array, insertion is an easy process, you need to increment the size of the array n and then add the new item to the bottom of the array. Similarly for unsorted arrays you can just add the item to the end of the dictionary, which takes constant time. For sorted arrays you need to make room for a new item, which may require moving many items causing insertion to become a linear-time operation. For sorted list, adding a new item is a relatively simple process. However you still need to search for the correct place in the dictionary, taking constant time.

Deletion We are given an pointer to the element to be deleted, which is a relatively simple process when dealing with an array. However it does leave a whole that needs to be filled. For an unsorted array you can just write over $A[x]$ with $A[n]$, however for the sorted variation you will need to move each of the elements $A[x+1]$ to $A[n]$ up one position taking constant time. For singly sorted lists the pointer x, isn't want we need to remove the element. Instead we need to the pointer to the successor, this will take linear time as we will have to search through all the elements to get to its successor. For doubly linked list this is a simpler operation than sorted arrays as removing an item from a list is an easier problem than filling a hole by moving array elements.

Traversal Operations (Predecessor and Successor) refer to the item appearing before x in **sorted order**. Thus for a sorted array the answer is $A[x-1]$ and $A[x+1]$. For sorted arrays this requires more searching as you need to find the item that is bigger or smaller than x. We have a similar issue with unsorted lists and sorted lists. However we again have the problem with lacking a pointer to $x-1$ for the singly linked list, requiring a computationally expensive search to get the required item.

Maximum and Minimum We face a similar issue with finding the largest/smallest item as we did with finding successors and predecessors. For sorted arrays the answer is simply $A[n]$ and $A[1]$ respectively. For a sorted doubly linked list the answer is the head and foot element. For other implementations at least one of the operations will require a linear sweep to get the required element.

2.5.9 Binary Trees

Binary Trees are a type of data structure that allow for both fast search (like sorted arrays) and flexible updates (like linked lists). Binary Search, discussed further in section 2.6.1.2 (48) requires that you have fast access to two elements. Specifically the median elements above and below the given node.

A *rooted binary tree* is recursively defined as either being:

- empty

- Consisting of a node called the root, together with two rooted binary trees called left and right sub-trees respectively.

The order amongst the two daughter nodes matters in rooted trees, so left is different from right. In a binary *search* tree each node has a label that is asingle key such that for any node labelled x , all nodes in the left subtree of x have keys $< x$ and all the nodes in the right subtree of x have keys $> x$. This search tree labelling is very speial. For any binary tree on n nodes and any set of n keys, there is exactly one labelling that gives that makes it a binary search tree.

2.5.9.1 Implementing a Binary Search Tree

Each binary tree node has a *left* and *right* pointers, an opitonal *parent* pointer and a data field. We can then define a tree structure as below:

```
typedef struct tree {
        item_type               item;
        struct tree *parent;
        struct tree *left;
        struct tree *right;
} tree;
```

Snippet 2.5.7: Structure of a tree node with optional parent parameter

The basic operations supported by a binary tree are searching, traversal, insertion, and deletion.

Searching in a Tree

To find a particular node in a binary search tree you start from the root node and work your way down the tree. For a given key k, and a node that has a value of x, there are 3 possible outcomes.

- - Return current node
- - Go to the right subtree
- - Go to the left subtree

Because each subtree of a binary tree is itself a binary search tree we can move through the tree recursively. Below you'll find a search algorithm that runs in $\mathcal{O}(h)$ time, where h denotes the height of the tree.

This algorithm runs in $\mathcal{O}(h)$ time, where h is the height of the tree.

```
tree *search_tree(tree *l, item_type x) {
        if (l == NULL) return NULL;

        if (l->item == x) return(l);

        if (x < l->item) {
                return( search_tree(l->left, x) );
        }
        else {
                return( search_tree(l->right, x) );
        }
}
```

Snippet 2.5.8: Structure of a tree node with optional parent parameter

Finding the Minimum and Maximum itemize

A special case of searching is finding the maximum and minimum elements in the tree. By definition the smallest element in the tree must be in the left subtree as all keys in the left subtree must have values less than that of the root. Applying this logic recursivly leads to the conclusion that the smallest item must be on the leftmost decendant on the root. Similarly, the maximum item must be on the rightmost descendant

```
tree *find_minimum(tree *t) {
        tree *min; //pointer to minimum elements

        if (t == NULL) return(NULL);

        min = t;
        while (min -> left != NULL) {
                min = min->left;
        }
        return(min);
}
```

Snippet 2.5.9: The find minimum function.

Traversal of a Tree

Boole (1854)

```
void traverse_tree (tree *l) {
        if (l != NULL) {
                traverse_tree(l->left);
                process_item(l->item);
                traverse_tree(l->right);
        }
}
```

Snippet 2.5.10: The traversing a binary tree in order..

Insertion in a Tree

Deletion from a Tree

2.5.9.2 Balanced Binary Search Trees

2.5.10 Priority Queues

2.5.11 Hashing and Strings

2.5.11.1 Collision Resolution

2.5.12 Graphs

2.5.12.1 Flavours of Graphs

2.5.12.2 Data Structures for Graphs

Adjacency Matrix

Adjacency Lists

2.5.13 Vectors

2.5.13.1 Dot Product

2.5.13.2 Angle Between Vectors

2.5.14 Exercises

2.5.14.1 Exam Style Questions

2.5.14.2 Programming Challenges

Chapter 2.6

Algorithms

2.6.1 Searching

2.6.1.1 Linear Search

2.6.1.2 Binary Search

Binary Tree Search

2.6.2 Sorting

2.6.2.1 Bubble Sort

2.6.2.2 Merge Sort

2.6.2.3 Insertion Sort

2.6.2.4 Quick Sort

2.6.3 Graph Traversal

2.6.3.1 Breadth First

2.6.3.2 Depth First

2.6.3.3 Dijkstra

2.6.3.4 A Star

2.6.3.5 Floyd-Warshall

Part 3

The Physical Computer

Chapter 3.1

Hardware

3.1.1 Relationship between Hardware and Software

3.1.2 Main Memory

3.1.2.1 Stored Program Concept

What separates computers from other machines is their ability to perform an infinite array of tasks, from interactive games, to word processing and performaing scientific calculations, all without needing change hardware . A washing machine contains a basic processor to control the different cycles and temperatures, but trying to get it to perform even a basic calculation you will struggle. This flexibility comes down to a concept known as the *Stored Program* concept. The idea behind it is remarkably simple, you have a fixed hardware platform capable of executing a fixed repertoire of instructions. At the same time these instructions can be used and combined to create arbitrary sophisticated programs. Moreover, the logic of these programs are not embedded in hardware, as it was in early computers, instead they are stored and manipulated in the computer's memory.

3.1.2.2 Addressable Memory

3.1.3 Secondary Storage

3.1.3.1 Hard Disk Drive

3.1.3.2 Solid State Drive

NAND Flash Memory

SSD Controller

Pages

3.1.3.3 Optical Drive

Chapter 3.2

The Processor

3.2.1 The Little Man Computer

The power of the computer does not arise from complexity, instead it comes from its ability to perform simple operations at an extremely high rate of knots. This means that actual design of a computer is also simple. We'll start our adventure in investigating the processor by looking at the Little Man Computer (LMC). Whilst it is not a physical computer it is a simplified model to allow us to understand the basics of computer architecture before moving on to looking at the more complex RM processor.

3.2.1.1 Physical Layout

We shall start by looking at the physical layout of the Little Man Computer, illustrated in Fig. 3.2.1. The LMC consists of a walled post-room, indicated by thick line. Inside this their are various items.

- 100 *pigeon holes* - numbered with an address ranging from 00 to 99. Each pigeon hole is designed to hold a single slip of paper, upon which is written a three digit decimal number.
- *A Calculator* - the calculator is used to enter and temporarily hold numbers, and also to add or subtract. The display is three digits wide and there is no provision made for negative numbers
- A *Hand Counter* - The type that you click to increment the count. There is a reset button located outside the post-room.
- The *Little Man* - It is his role to perform certain tasks to be defined shortly.

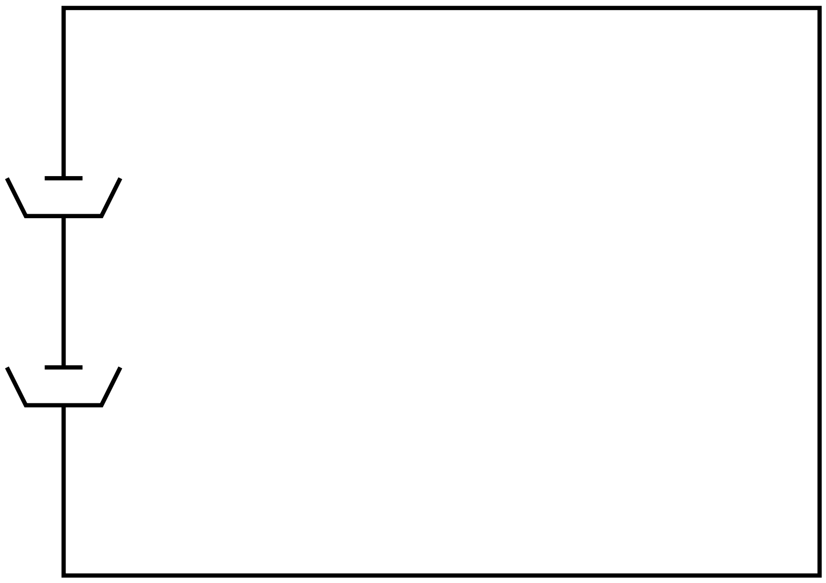


Figure 3.2.1: Layout of the Little Man Computer

- An *in basket* and an *out basket*. These allow a user to communicate with the Little Man by putting a three digit number into the in basket. Similarly the out box allows the Little Man to write a three digit number on a slip of paper and leave it in the out basket.

3.2.1.2 Instructions of the LMC

We would like the Little Man to do some useful work for us. For this purpose we have invented a small group of instructions that he can perform. Each instruction will consist of 3 digits. We use the first digit to tell the Little Man what instruction to perform, this is known as the opcode . The last two digits are known as the operand and tell the Little Man to get data to be used as part of the instruction. Figure 3.2.2 shows the split of the instruction into its opcodes and operands.

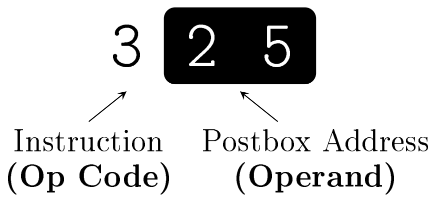


Figure 3.2.2: The structure of an LMC command

Whilst the structure of the operand is relatively straight forward, `25` represents mailbox 25 for example, we need more information to decode the opcodes. The processor instruction set defines the instructions, and the machine codes that go with it. The are particular for each processor.

There are 11 different instructions that the Little Man has to decode.

COFFEE BREAK (or HALT) - opcode 0 The little man takes a rest. He ignores the address portion of the instruction.

ADD - opcode 1 The Little Man walks over to the post-box specified in the instruction. He reads the three-digit number located in the mailbox and walks over to the calculator and *adds it to the number already in the calculator.*

SUBTRACT - opcode 2 The Little Man walks over to the post-box specified in the instruction. He reads the three-digit number located in the mailbox and walks over to the calculator and *subtracts it from the number already in the calculator.* [1].

STORE - opcode 3 The Little Man walks over to the calculator and reads the number there. He writes that number of a slip of paper and puts it in the post-box specified in the operand. The number in the calculator is unchanged specified in the instruction and the previous value is overwritten.

LOAD - opcode 5 Similar to the `ADD` instruction, the Little Man walks over to the post-box specified in the instruction. He reads the three-digit number located in the mailbox and walks over to the calculator and types it into the calculator. This leaves the original mailbox unchanged but the calculators previous value is overwritten.

BRANCH UNCONDITIONALLY (or JUMP) - opcode 6 This instruction tells the little man to walk over to the instruction counter and *change* the counter to the location shown in the operand.

BRANCH ON ZERO - opcode 7 The Little Man will walk over the calculator and observe the number stored there. If its current value is zero, he will walk over to the instruction location counter and modify its value to the address specified by the operand.

[1]For the purposes of the model we'll assume that the calculator handles negative numbers correctly but the Little Man ignores any minus sign

BRANCH ON POSITIVE - opcode 8 The Little Man will walk over the calculator and observe the number stored there. If its current value is *positive*, he will walk over to the instruction location counter and modify its value to the address specified by the operand.

INPUT - opcode 901 The Little Man walks over to the basket and picks up the slip of paper in the basket. He then walks over to the calculator and punches it into the calculator. If there are multiple slips of paper the Little Man only takes the oldest.

OUTPUT - opcode 902 The Little Man walks over to the calculator and writes down the number that he sees there on a piece of paper. He walks over to the out basket and places the slip of paper there for the user.

3.2.2 Processor Performance

3.2.2.1 Number of Cores

3.2.2.2 Cache Size

3.2.2.3 Clock Speed

3.2.2.4 Word Length

3.2.3 Assembly Programming

3.2.3.1 Registers

It is said that Assembly programming is programming by moving data.

Register Transfer Language

Throughout this text we are going to use a shorthand called register transfer language. RTL is an algebraic notation how information is accessed from memories and registers, and how it is operated on. It is not a programming language but a notation.

It is important to distinguish between a memory location. RTL uses square brackets to indicate the contents of a memory location. So, the expression

```
[6] = 3
```

is interpreted as *the contents of memory location 6 contains the value 3.* If instead we are dealing with registers, we use their name rather than their address. For example,

`[R4] = PQR`

A left arrow (←) indicates the transfer of data. The left side indicates the destination of the data defined by the source of the data defined on the right. So the expression,

`[MAR] ← [PC]`

indicates that the contents of the program counter are copied into the memory address register. This means that if you wanted to copy the contents of memory location 4 to memory location 7 you would use:

`[7] ← [4]`

3.2.3.2 Addressing Modes

In most high level languages the line `int x = 5;` would just assign the value of 5 to the variable `x`. However, in assembly programming there are many different ways to the simple function, `ADD R0,5`, can be interpreted depending on how exactly the value of 5 is passed. We're going to be look at an simplified system containing just 2 registers, and a 3 bit Memory, containing 8 values, as illustrated in Fig. 3.2.3

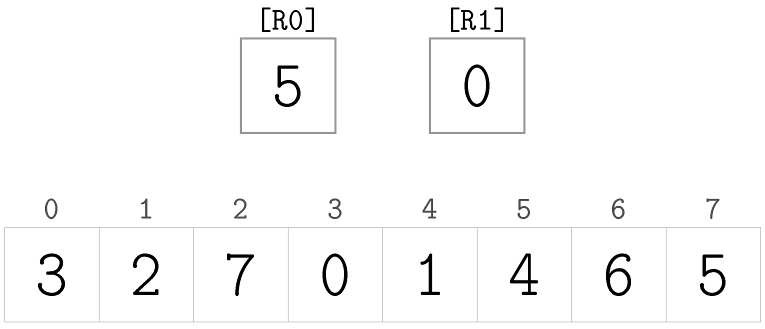


Figure 3.2.3: A Simplified Memory System

We'll look at 5 different methods and how they are implemented in LMC[2] and ARM.

[2]LMC only has one register, the accumulator. In our model R0 represents the accumulator

Immediate

The most simple form of memory addressing is immediate which is where you do not involve the memory at all. For example lets say you wanted to add 5 to what ever is in the target register. You just use a literal operand, which in most assembly languages is defined by using the # symbol. So to complete the This requires no relation to the memory and is probably the easiest to understand. This is useful because you don't want to have to set aside memory locations or registers to store constants. Imagine how much slower your computer would run if every time it wanted to increment something it would have to look up the value of 1 in memory.

RTL	`[R0] = 5`
LMC	Does not Exist.
ARM	`MOV r0, #5`

Direct

Direct addressing , sometimes referred to as absolute addressing is where the operand specifies the location of the data. So to use absolute addressing you would provide the location in memory or in a register. For example `ADD R0, R0, R1` uses absolute addressing to set the contents of register 0 to be equal to register 1 plus register 0.

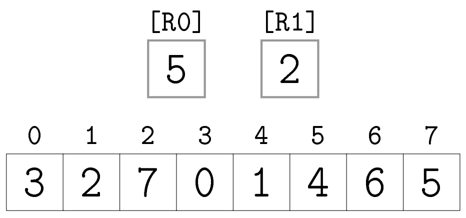
RTL	`[R0]` $\leftarrow$ `[5]`
LMC	`LDA 5`
ARM	`MOV R0, 5`

Indirect

In indirect addressing , the operand is specified indirectly via the contents of a pointer. For example the command `LDR R0, [R1]`, would go the memory location given by `R1` and would then take the value there and load it into `R0`. So for example, in situation below, `R1` points to `[2]`, which has a value of 7. This would then be loaded into `R0`.

RTL	`[R0]` $\leftarrow$ `[[5]]`
LMC	`LDA 5; ADD 5XX; STA TEMP; TEMP;...; 5XX VAR 500` [3]
ARM	`MOV R1,##0;` *`LDA R0, [R1, ##5];`*

[2]Note that for `[5] = 101` this leads to an infinite loop

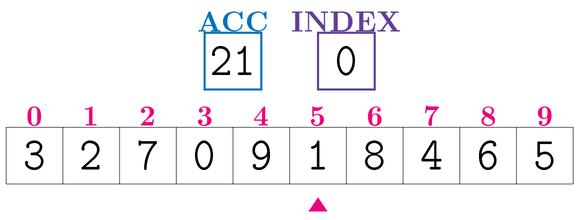


Register Addressing

Register Addressing makes use of the value of a register in order to provide more complex modes of addressing.

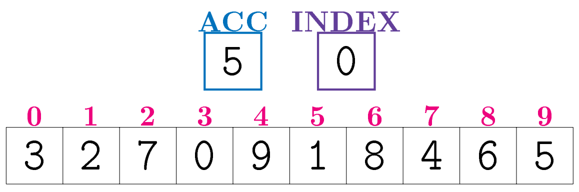
You most commonly find this when dealing with `BRA` instructions, When you do the second instruction is now relative to last instruction. So if the next command is `ADD 4`, rather than being relating to the information in the 4th memory location, its moves the point along 4 spaces. So instead it refers to information in the 6th memory location. So the final result would be 20.

If we then called `ADD -1` then we would shift the point to 1 space to the left. So it would go the fifth memory location, and then add this value.



Indexed Addressing

Indexed addressing uses a special register called the index address. The index register is incremented each time it is referred to. The memory address given by the instruction is then addressed using the INDEX address.



3.2.3.3 ARM

3.2.3.4 Savage-1

3.2.4 Building the Savage-1

3.2.4.1 Clock

3.2.4.2 Buses

3.2.4.3 System Registers

3.2.4.4 The Arithmetic Logic Unit

3.2.4.5 Fetch, Decode Execute Cycle

3.2.4.6 Opcodes and Operands

3.2.4.7 The Control Unit

3.2.4.8 I/O and Interrupts

3.2.4.9 Pipelining

3.2.5 Types of Processor

3.2.5.1 Harvard and Von Neumann

3.2.5.2 Parallel Processing

Chapter 3.3

Software

Part 4

Web Applications

Chapter 4.1

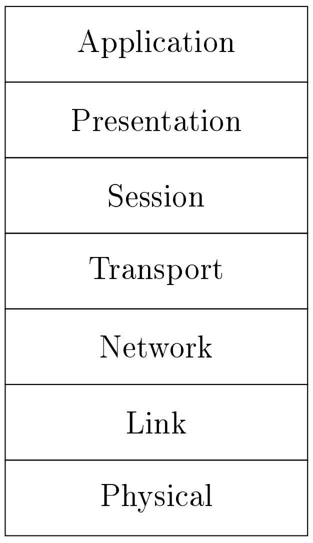
Networking and the Internet

A network is a way of

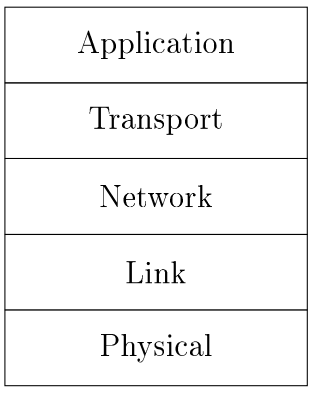
4.1.1 OSI Model and the TCP/IP Stack

The Internet is an *extremely* complicated system. There are many pieces that must work together in order for you to receive your cat video. Given the enormous complexity of all the interconnecting parts you would think that it might be a nightmare to understand. Luckily we can break it down into layers.

In the late 1970s, the International Organisation for Standardisation (ISO) proposed that computer networks be organised around a seven layer model. Figure 4.1.1b



(a) OSI



(b) TCP/IP Stack

Figure 4.1.1: The two different layering models for

Chapter 4.2

The Internet

Chapter 4.3

Fundamentals of Databases

Part 5

Programming

Chapter 5.1

Programming Concepts

Chapter 5.2

Aspects of Software Development

Chapter 5.3

Procedural Programming

Chapter 5.4

String Handling Operations

Chapter 5.5

Fundamentals of Functional Programming

Chapter 5.6

Object-Orientated Programming

Chapter 5.7

Regular Languages

Data structures

Part 6

The Appendices

Introduction to Computer Science

Appendix A

Scheme of Work

Appendix B

Introduction to UNIX

UNIX is an operating system designed to be used on any kind of computer. It is made up of three parts: the kernel; the shell and the programs.

B.1 The Kernel

The kernel of UNIX is the hub of the operating system: it allocates

Appendix C

Introduction to C

One of the most popular languages ever developed is called C. It was created by a group including Dennis Ritchie and Brian Kernighan at Bell Laboratories between 1969 and 1973 to rewrite the UNIX operating system from its original assembly language. By many measures, C [1] is the most widely used language in existence. Its popularity stems from a number of factors:

- Available on a tremendous variety of platforms, from super computers down to embedded microcontrollers.
- Relative Ease of use.
- Moderate level of abstraction providing higher productivity that assembly language, yet giving the programmer a good understanding of how the code will be execute.
- Suitability for generating high performance programs.
- Ability to interact directly with the hardware.

C.1 My First Program

A C program is a plain text file that describes operations for the computer to perform. The text file is then *compiled*, converting into from the human readable format that is was written in to the machine readable format that is its needed to be executed. C programs are generally contained in one or more text files that end with the extention '.c'.

In general, a C program is organised into one or more functions. Every program must include the `main` function, which is where the program starts

[1]including a family of closely related languages such as C#, C++ and Objective-C

```
#include <stdio.h>

int main(void) {
        printf("Hello World! \n ");
}
```

Which Outputs:

```
$~ Hello World!
```

Snippet C.1: Our first see program

executing. Most programs use other functions defined elsewhere in the code and/or in in a library. We can split our code in snippet C.1 into the header, the `main` function, and the body.

Appendix D

Introduction to Python

Appendix E

Introduction to LaTeX

Appendix F

Introduction to Version Control

Index

Bibliography

Boole, G. (1854), *An Investigation of the Laws of Thought.*