2.0 INTRODUCTION

The concept of an algorithm is fundamental to Computer Science. Designing an efficient algorithms for a program plays a crucial role in developing large scale computer systems. When a program is run on a computer, two of the most important consideration are:

1. Time Complexity
2. Space Complexity

The Time Complexity of a program/algorithm is the amount of computer time that it needs to run to completion. The space complexity of a program is the amount of memory that it needs to run to completion.

There are other issues for example:

(i) Does it do what we want it to do?

(ii) Does it work correctly according to the original specifications of the task?

(iii) Is there documentation which describes how to use it and how it works?

(iv) Are subroutines created in such a way that they perform logical sub-functions?

(v) Is the code readable?

The above criteria are all vitally important when it comes to writing software, most especially for large systems. Though we will not be discussing how to reach these goals, one has to try to achieve them while, writing a program.. Hopefully this more subtle approach will gradually infect your own program writing habits so that you will automatically strive to achieve these goals.

The analysis of running time generally has received more attention than memory and we will follow the same practice here. This is justified by the fact that many programs of interest have relatively modest memory requirements. Also any program that uses extremely large amounts of memory automatically requires a lot of time.

Some of algorithms discussed in data structure book are straightforward while others are complicated and tricky but virtually all have complexity of $0(n_3)$ where ii defines the size of input data. There is another class oil problems, called NP complete whose complexity may be described by exponential time.

2.1 OBJECTIVES

- Define time and space complexity

- How to analyse small programs
- Define what is NP-Complete problem
- List NP-Complete Problems

2.2 TIME COMPLEXITY

In analysing algorithm we will not consider the following information although they are very important.

(i) The machine we are executing on;
(ii) Its machine language instruction set;
(iii) The time required by each machine instruction;
(iv) The translation; a compiler will make from the source to the machine language.

It is possible to determine these figures by choosing a real machine and an existing compiler. Another approach would be to define a hypothetical machine (with imaginary execution times), but make the times reasonably close to those of existing hardware so that resulting figures would be representative. Neither of these alternatives seems attractive. In both cases the exact times we would determine would not apply to many machines or to any machine. Also, there would be the problem of the compiler, which could vary from machine to machine. Moreover, it is often difficult to get reliable `timing figures because of clock limitations and a multi-programming or time sharing environment. finally, the difficulty of learning another machine language outweighs the advantage of finding "exact" fictitious times. All these considerations lead us to limit our goals for an a priori analysis. Instead, we will concentrate on developing only the frequency count for all statements. The anomalies of machine configuration and language will be lumped together when we do our experimental studies. Parallelism will not be considered.

Consider the three examples of figure 1 below:

```
                       for 1 = 1 to ii do      for 1 = 1 to n do
                          x = x + 1            for j = 1 to n do
x = x + 1              end                             x = x +1
                                               end
                                      end
(a)                    (b)                     (c)
```

Figure 1: Three simple programs for frequency counting

In program (a) we assume that the statement x = x+1 is not contained within any loop either explicit or implicit. Then its **frequency count** is one. In program (b) the same statement will be executed n times and in program (c) n^2 are said to be different and increasing orders of magnitude just like 1, 10, 100 would be if we let n = 10. In our analysis of execution we will be concerned chiefly with determining the order of magnitude of an algorithm. This means determining those statements which may have the greatest frequency count.

The following formulas are useful in counting the steps executed by an algorithm:

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

$$1^2 + 2^2 + \ldots\ldots + n^2 = \frac{n(n+1)(2n+1}{6}$$

To clarify some of these ideas, let us look at a simple program for computing the n-th Fibonacci number. The Fibonacc sequence starts as

0,1, 1,2,3,5,8,13,21,34,55 . ...

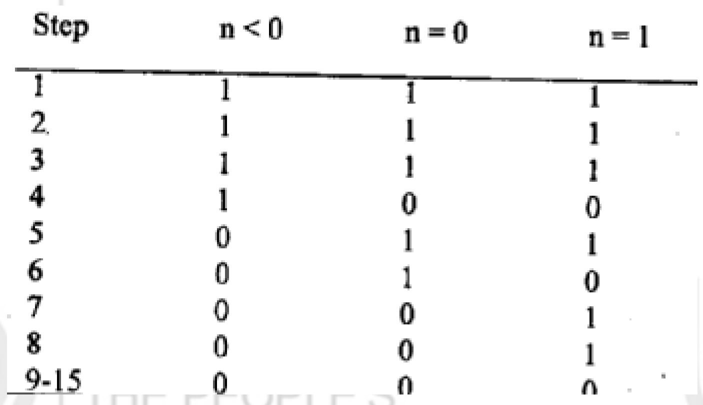
Each new term is obtained by taking the sum of the two previous terms. If we call the first term of the sequence F_o then $F_o = 0$, F_1, = 1 and in general

$$F_n = F_{n-1} + F_{n-2},\ n >= 2.$$

The program on the following page takes any non-negative integer n and prints the value F..

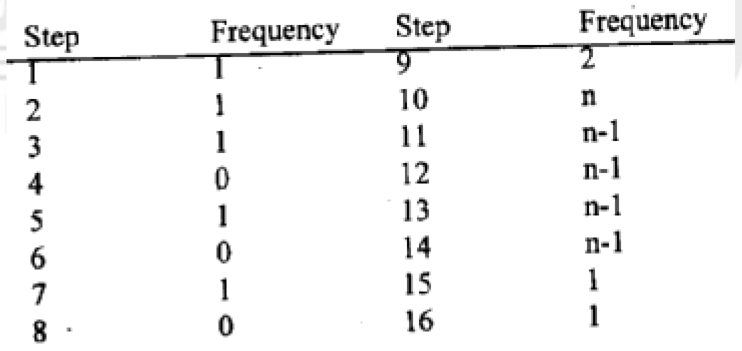
```
1      procedure FIBONACCI
2          read (n)
3-4        if n < 0 then [print ('error'); stop]
5-6        if n = 0 then [print ('0'); stop]
7-8        if n = 1 then [print ('1'); stop]
9          fnm2=0; fnm1=1
10         for I = 2 to n do
11             fn = fnm1 + fnm2
12             fnm2 = fnm1
13             fnm1 = fn
14         end
15         print (fn)
16     end FIBONACCI
```

The first problem in beginning an analysis is to determine some reasonable values of n. A complete set would include four cases: $n < 0$, $n = 0$, $n = 1$ and $n > 1$. Below is a table which summarises the frequency counts for the first three cases.



Step	n < 0	n = 0	n = 1
1	1	1	1
2	1	1	1
3	1	1	1
4	1	0	0
5	0	1	1
6	0	1	0
7	0	0	1
8	0	0	1
9-15	0	0	0

These three cases are not very interesting. None of them exercises the program very much. Notice, though, how each **if** statement has two parts: the **if** condition and the **then** clause. These may have different execution counts. The most interesting case for analysis comes when $n > 1$. At this point the for loop will actually be entered. Steps 1, 2, 3, 5, 7 and 9 will be executed once, but steps 4, 6 and 8 not at all. Both commands in step 9 are executed once. Bow, for $n > = 2$ how often is step 10 executed: not $n - 1$ but n times. Though 2 to n is only $n - 1$ executions, remember that there will be a last return to step 10 where i is incremented to $n + 1$, the test $1 > n$ made and the branch taken to step 15. Thus, steps 11, 12, 13 and 14 will be executed $n - 1$ times but step 10 will be done n times. We can summarise all of this with a table.



Step	Frequency	Step	Frequency
1	1	9	2
2	1	10	n
3	1	11	n-1
4	0	12	n-1
5	1	13	n-1
6	0	14	n-1
7	1	15	1
8	0	16	1

Figure 2: Execution Count for Computing F.

Each statement is counted once, so step 9 has 2 statements and is executed once for a total of 2, Clearly, the actual time taken. by each statement will vary. The **for** statement is really a combination of several statements, but we will count it as one. The total count then is $5_n + 5$. We will often write this as 0(n), ignoring the two constants 5. This notation means that the order of magnitude is proportional to n.

2.3 THE BIG-0 NOTATION

The notation f(n) = 0(g(n)) (read as f of n equals big-oh of g of n) has a precise mathematical definition.

Definition: f(n) = 0(g(n)) iff there exist two constants c and n_o such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$.

f(n) will normally represent the computing time of some algorithm. When we say that the computing time of an algorithm is 0(g(n)), we mean that its execution takes no more than a constant time g(n). n is a parameter which characterises the inputs and / or outputs. For example n might be the number of inputs or the number of outputs or their sum or the magnitude of one of them. For the Fibonacci program n represents the magnitude of the input and the time for this program is written as T(FIBONACCI) = 0(n).

We write 0(1) to mean a computing time which is a constant. 0(n) is called linear, $0(n^2)$ is called quadratic, $0(n^3)$ is called cubic and $0(2^n)$ is called exponential. If an algorithm takes time 0(log n) it is faster, for sufficiently large n, than if it had taken 0(n). Similarly, 0(n log n) is better than $0(n^2)$ but not as good as 0(n). These seven computing times, 0(1), 0(log n), 0(n), 0(n log n), $0(n^2)$, $0(n^3)$ and $0(2^n)$ are the ones which are usually referred in data structure book.

If we have two algorithms which perform the same task, and the first has a computing time which is 0(n) and the second O (n^2)then we will usually take the first as superior. The reason for this is that as n increases the time for the second algorithm will get far worse than the time for the first. For example, if the constant for algorithms one and two are 10 and 1/2 respectively, then we get the following table of computing times:

n	10n	$n^2/2$
1	10	1/2
5	50	12-1/2
10	100	50
15	150	112-1/2
20	200	200
25	250	312-112
30	300	450

for n ≤ 20, algorithm two had a smaller computing time but once past that point algorithm one became better. This shows why we choose the algorithm with the smaller order of magnitude, but we emphasise that this is not the whole story. For small data sets, the respective constants must be carefully determined. In practice these constants depend on many factors, such as the language and the machine one is using. Thus, we will usually postpone the establishment of the constant until after the program has been written Then a performance profile can be gathered using real time calculation.

Figures 3 and 4 show how the computing times (counts) grow with a constant equal to one.
Notice

how the times O(n) and O(n log n) grow much more slowly than the others For large data sets, algorithms with a complexity greater than O(n log n) are often impractical. An algorithm which is exponential will work only for very small inputs. For exponential algorithms, even if we improve the constant, say by 1/2 or 1/3), we will not improve the amount of data we can handle by very much.

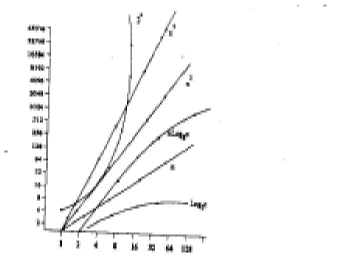


Figure 3: Rate of Growth of Common Computing Time Functions

$\log_2 n$	n	n $\log_2 n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	2,147,483,648

Figure 4: Values for Computing Functions

$\log_2 n$	n	n $\log_2 n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	2,147,483,648

Figure 4: Values for Computing Functions

Given an algorithm, we analyse the frequency count of each statement and total the sum. This may give a polynomial

$$P(n) = c_k n^k + c_{k-1} n^{k-1} + ... + c_1 n + c_o$$

where the c_j are constants. $c_k \neq 0$ and n is a parameter. Using bio-ch notation, P(n) = $0(n^2)$. On the other hand, if any step is executed 2^n times or more the expression

$c2^n + P(n) = 0(12^n)$

Another valid performance measure of an algorithm is the space it requires. Often one can trade space for time, getting a faster algorithm but using more space. In the next section we will discuss it in detail.

2.4 SPACE COMPLEXITY

The space needed by a program is the sum of the following components:

I. Fixed space requirement : The component refers to space requirement that do not depend on the number and size of the program's inputs and outputs. The fixed requirements include the instruction space (space needed to store the code), space for simple variables, fixed size structured variable (such as **structs** in c-language) and constants.

II. Variable space requirements: This component consists of the s pace needed by structured variables whose size depends on the particular instance 1, of the problem being solved. It also includes the additional space required when a function uses recursion.

2.5 NP-COMPLETENES

The term NP-Complete is often used but not always well understood. In practical terms, an NP- problem is one that can be solved by computer only by waiting all extra-ordinarily long time for its solution. Some-of the algorithms discussed in the data structure book are straightforward while others are complicated and tricky, but virtually all have complexity is $0(n^3)$, where ii is the appropriately defined input size. NP-Complete problems are concerned with problems whose complexity may be described by exponential functions, problems for which the best known algorithms would require many years for moderately large inputs.

In reality NP contains an infinite number of problems. Some of them are well known travelling salesperson problem and the graph colouring problem; and like them there are many other problems. Let us understand a bit about a graph colouring problem.

The graph colouring problem is an abstraction of certain types of scheduling problems, for example, suppose that the final exams at a university are to be scheduled during one week with each day a total of 15 time slots. The exams for some courses, say Calculus 1 and Physics 1, must be at different times because many students are in both classes. Let V be the set of courses, and let E be the pairs of courses whose exams can be scheduled in the 15 time slots without conflicts if and only if the graph G = (V, e) can be coloured with 15 colours.

Here N stands for nondeterministic and P stands for Polynomial time. The class NP may be defined as the set of all decision problems that can be solved by non deterministic computer in polynomical time.

2.6 SUMMARY

In this unit, we discussed issues related to analysis of algorithms. We took a simple example and analysed its time complexity. We also discussed in brief about problems of type P-Completeness.

2.7 REVIEW QUESTIONS

1. Match the following:

(a) 0(1)	(i)	exponential
(b) 0(n)	(ii)	cubic
(C) 0(n^2)	(iii)	linear
(d) 0(n^3)	(iv)	constant
(c) 0(2^n)		

2. List two well-known NP-Complete problems

………

……

2.8 MODEL ANSWERS

1. (a) (iv)
   (b) (iii)
   (c) (v)
   (d) (ii)
   (e) (i)

2. The two NP-Complete problems are:

   (i) Graph- Colouring
   (ii) Travelling Sales person