Analysis of Algorithm Complexity

Algorithms usually possess the following qualities and capabilities:

- Easily modifiable if necessary.
- They are correct for clearly defined solution.
- Require less computer time, storage and peripherals, i.e. they are more economical.
- They are documented well enough to be used by others who do not have a detailed knowledge of the inner working.
- The solution is pleasing and satisfying to its designer and user.
- They are able to be used as a sub-procedure for other problems.

Two or more algorithms can solve the same problem in different ways. So, quantitative measures are valuable in that they provide a way of comparing the performance of two or more algorithms that are intended to solve the same problem. This is an important step because the use of an algorithm that is more efficient in terms of time, resources required, can save time and money.

Computational Algorithm Complexity

We can characterize an algorithm's performance in terms of the size (usually n) of the problem being solved. More computing resources are needed to solve larger problems in the same class. Table 3.1 illustrates the comparative cost of solving the problem for a range of n values.

Table 3.1 shows that only very small problems can be solved with an algorithm that exhibit exponential behavior. An exponential problem with $n = 100$ would take immeasurably longer time. At the other extreme, for an algorithm with logarithmic dependency would merely take much less time (13 steps in case of $\log_2 n$ in Table 3.1). These examples emphasize the importance of the way in which algorithms behave as a function of the problem size. Analysis of an algorithm also provides the theoretical model of the inherent computational complexity of a particular problem.

Table 3.1

$Log_2 n$	n	$n log_2 n$	n^2	n^3	2^n
1	2	2	4	8	4
3.322	10	33.22	10^2	10^3	$>10^3$
6.644	10^2	664.4	10^4	10^6	$>>10^{25}$
9.966	10^3	9966.0	10^6	10^9	$>>10^{250}$
13.287	10^4	132877	10^8	10^{12}	$>>10^{2500}$

To decide how to characterize the behavior of an algorithm as a function of size of the problem n, we must study the mechanism very carefully to decide just what constitutes the dominant mechanism. It may be the number of times a particular expression is evaluated, or the number of comparisons or exchanges that must be made as n grows. For example, comparisons, exchanges, and moves count most in sorting algorithm. The number of comparisons usually dominates so we use comparisons in the computational model for sorting algorithms.

The Order of Notation

The O-notation gives an upper bound to a function within a constant factor. For a given function $g(n)$, we denote by $O(g(n))$ the set of functions. $O(g(n)) = \{f(n)$: there exist positive constants c and $n0$, such that $0 <= f(n) <= cg(n)$ for all $n >= n0\}$

Using the O-notation, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure. For example, a double nested loop structure of the following algorithm immediately yields $O(n^2)$ upper bound on the worst case running time.

```
for i=0 to n
    for j=0 to n
        print i,j
        next j
        next i
```

What we mean by saying "the running time is $O(n^2)$" is that the worst case running time (which is a function of n) is $O(n^2)$. Or equivalently, no matter what the particular input of size n is chosen for each value of n, the running time on that set of inputs is $O(n^2)$.

Rules for Using the Big-O Notation

Big-O bounds, because they ignore constants, usually allow for very simple expression for the running time bounds. Below are some

program development life cycle with suitable example.

2. Define the term compiler, interpreter, loader and Linker with suitable example.
3. Define the term programming language and also provide the classification of programming language with suitable example.
4. What do you mean by flow chart and define all the symbols used in flow chart and give one example?
5. Define the term structured programming with suitable example. Also define the various constructs used in structured programming.
6. Differentiate between higher level programming language and lower level programming language with suitable example.
7. Describe top-down and bottom-up approaches of system development.
8. What do you understand by efficiency of algorithm? How can it be evaluated?
9. Draw a flow chart to input 10 numbers and print the maximum number between them.
10. Write the algorithm to input 10 numbers and print the maximum number between them.
11. Write an algorithm and draw a flow chart to input 10 numbers and print the sum of that numbers.
12. Write an algorithm and draw a flow chart to input a number and check whether that number is prime or not.
13. Write an algorithm and draw a flow chart to input a number and check whether that number is palindrome or not.
14. Write an algorithm and draw a flowchart to input a number and check whether that number is Armstrong or not.
15. Write an algorithm and draw a flowchart to input a number and print the reverse of that number.

Result from Step 3: – 0100
To verify, note that 9 – 13 = – 4

Subtracting Using 2's Complement

For subtracting a smaller number from a larger number, the 2's complement method is as follows:

1. Determine the 2's complement of the smaller number
2. Add the 2's complement to the larger number
3. Discard the final carry (there is always one in this case).

Example
11001 – 10011

Solution

Result from Step 1: 01101
Result from Step 2: 100110
Result from Step 3: 00110
Again, to verify, note that 25 – 19 = 6

For subtracting a larger number from a smaller number, the 2's complement method is as follows:

1. Determine the 2's complement of the larger number.
2. Add the 2's complement to the smaller number.
3. There is no carry of the left-most column. The result is in 2's complement form and is negative.
4. Change the sign and take the 2's complement of the result to get the final answer.

Example
1001 – 1101

Solution

Result from Step 1: 0011
Result from Step 2: 1100
Result from Step 3: – 0100
Again to verify, note that 9 – 13 = –4

Binary Multiplication

Multiplication in the binary system works the same way as in the decimal system:

- 1 * 1 = 1
- 1 * 0 = 0
- 0 * 1 = 0

Example

```
  101
* 11
-----
  101
 1010
-----
 1111
```

Note that multiplying by two is extremely easy. To multiply by two, just add up a 0 on the end.

Binary Division

Binary division follows the same rules as in decimal division. For the sake of simplicity, throw away the remainder.

Example

```
111011/11
10011 r 10
-----------
11)111011
 – 11
-----
  101
 – 11
-----
  101
   11
-----
   10
```

OCTAL NUMBER SYSTEM

Although this was once a democratic number base, particularly in the Digital Equipment Corporation PDP/8 and other old computing system, it is rarely used today. The octal system is founded on the binary system with a 3-bit boundary. The octal number system:

- uses base 8
- includes only the digits 0 to 7 (any other digit would make the number an invalid octal number).

The weighted values for each position are as follows:

8^5	8^4	8^3	8^2	8^1	8^0
32768	4096	512	64	8	1

Octal to Decimal Conversion

To convert from octal to decimal, multiply the value in each position by its octal weight and add each value.

Example 1
Convert $(127662)_8$ into decimal form.

for that machine only depending upon the processor that the machine is using.

ii. **High-level Languages or Problem Oriented Languages:**
These languages are particularly oriented towards describing the procedures for solving the problem in a concise, precise and unambiguous manner. Every high level language follows a precise set of rules. They are developed to allow application programs to be run on a variety of computers. These programming languages are *machine independent*. Languages falling in this category are FORTRAN, BASIC, and PASCAL, etc. They are easy to learn and programs may be written in these languages with much less effort. However, the computer cannot understand them and they need to be translated into machine language with the help of other programs known as compilers or translators.

C LANGUAGE—INTRODUCTION

C is a programming language formulated at AT & T's Bell Laboratories of USA in 1972. It was projected and written by a man named Dennis Ritchie. In the late 70s, C began to replace the more intimate languages of that time like PL/I, ALGOL, etc. No one forced C. It was not made the 'official' Bell Laboratories programming language. Thus, without any ad C's reputation spread and its pool of users grew. Ritchie seems to have been rather surprized that so many programmers favoured C to older languages like FORTRAN or PL/I, or the newer ones like Pascal and APL. But, that is what happened.

The programming language C was primitively developed by Dennis Ritchie of Bell Laboratories and was planned to run on a PDP-11 with a UNIX operating system. Although it was earlier designated to run under UNIX, there has been a great interest in running it under the MS-DOS operating system on the IBM PC and compatibles. It is an first-class language for this surroundings because of the simplicity of expression, the compactness of the code, and the wide range of applicability. Also, due to the simplicity and ease of writing a C compiler, it is usually the first high level language available on any new computer, including microcomputers, minicomputers, and mainframe computer.

C is not the best beginning language because it is somewhat cryptical in nature. It allows the programmer a wide range of procedures from high level down to a very low level, approaching the level of assembly language. There seems to be no limit to the tractability available. One experienced C programmer made the statement, "You can program anything in C", and the statement is well supported by my own feel with the language. Along with the resulting freedom however, you take on a great deal of duty because it is very easy to write a program that destroys itself due to the silly little errors that a good Pascal compiler will flag and call a fatal error. In C, you are a lot on your own as you will soon find. In 1972, C programming language was developed at Bell Laboratories by Dennis Ritchie. C is a simple programming language with a comparatively simple to understand syntax and few keywords. C is useless. C itself has no input/output commands, does not have support for strings as a key data type. There is no predefined useful math functions. C demands the use of libraries as C is useless by itself. This increases the complexity of the C programming language. The use of ANSI libraries and other methods, the issue of standard libraries is settled.

Why Use C?

C has been used successful for every type of programming problem conceivable from operating systems to spreadsheets to expert systems and efficient compilers are available for machines ranging in power from the Apple Macintosh to the Cray supercomputers. The largest measure of C's success seems to be based on purely practical considerations:

- The portability of the compiler
- The standard library concept
- A powerful and varied repertoire of operators
- An elegant syntax