Chapter 1

Basic Concepts

Algorithm

An Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions. In addition every algorithm must satisfy the following criteria:

Input: there are zero or more quantities, which are externally supplied;

Output: at least one quantity is produced;

Definiteness: each instruction must be clear and unambiguous;

Finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

Effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

In formal computer science, one distinguishes between an algorithm, and a program. A program does not necessarily satisfy the fourth condition. One important example of such a program for a computer is its operating system, which never terminates (except for system crashes) but continues in a wait loop until more jobs are entered.

We represent algorithm using a pseudo language that is a combination of the constructs of a programming language together with informal English statements.

Performance of a program:

The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical, and the other experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity:

The time needed by an algorithm expressed as a function of the size of a problem is called the *time complexity* of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

Space Complexity:

The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:

Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.

Data space: Data space is the space needed to store all constant and variable values. Data space has two components:

- Space needed by constants and simple variables in program.
- Space needed by dynamically allocated objects such as arrays and class instances.

Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.

Instruction Space: The amount of instructions space that is needed depends on factors such as:

- The compiler used to complete the program into machine code.
- The compiler options in effect at the time of compilation
- The target computer.

Algorithm Design Goals

The three basic design goals that one should strive for in a program are:

1. Try to save Time
2. Try to save Space
3. Try to save Face

A program that runs faster is a better program, so saving time is an obvious goal. Like wise, a program that saves space over a competing program is considered desirable. We want to "save face" by preventing the program from locking up or generating reams of garbled data.

Classification of Algorithms

If 'n' is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.

1 Next instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is a constant.

Log n When the running time of a program is logarithmic, the program gets slightly slower as n grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction., When n is a million, log n is a doubled. Whenever n doubles, log n increases by a constant, but log n does not double until n increases to n^2.

n When the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element. This is the optimal situation for an algorithm that must process n inputs.

n. log n This running time arises for algorithms that solve a problem by breaking it up into smaller sub-problems, solving then independently, and then combining the solutions. When n doubles, the running time more than doubles.

n^2 When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop) whenever n doubles, the running time increases four fold.

n^3 Similarly, an algorithm that process triples of data items (perhaps in a triple-nested loop) has a cubic running time and is practical for use only on small problems. Whenever n doubles, the running time increases eight fold.

2^n Few algorithms with exponential running time are likely to be appropriate for practical use, such algorithms arise naturally as "brute-force" solutions to problems. Whenever n doubles, the running time squares.

Complexity of Algorithms

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size 'n' of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size 'n'. Complexity shall refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size 'n' of the input data but aiso on the particular data. The complexity function f(n) for certain cases are:

1. Best Case : The minimum possible value of f(n) is called the best case.
2. Average Case : The expected value of f(n).
3. Worst Case : The maximum value of f(n) for any key possible input.

The field of computer science, which studies efficiency of algorithms, is known as analysis of algorithms.

Algorithms can be evaluated by a variety of criteria. Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of a problem. We will associate with the problem an integer, called the size of the problem, which is a measure of the quantity of input data.

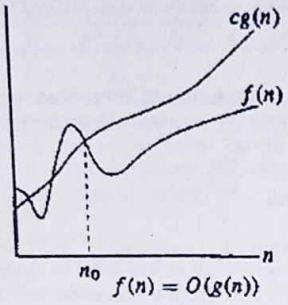
Rate of Growth:

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

1. Big-OH (O) [1],
2. Big-OMEGA (Ω),
3. Big-THETA (θ) and
4. Little-OH (o)

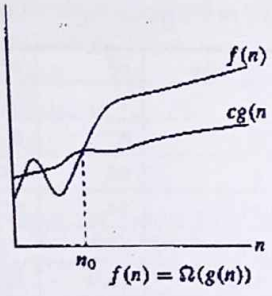
Big-OH O (Upper Bound)

f(n) = O(g(n)), (pronounced order of or big oh), says that the growth rate of f(n) is less than or equal (≤) that of g(n).



Big-OMEGA Ω (Lower Bound)

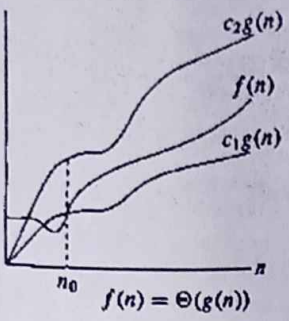
f(n) = Ω(g(n)) (pronounced omega), says that the growth rate of f(n) is greater than or equal to (≥) that of g(n).



[1] In 1892, P. Bachmann invented a notation for characterizing the asymptotic behavior of functions. His invention has come to be known as *big oh notation*.

Big–THETA θ (Same order)

f(n) = Θ(g(n)) (pronounced theta), says that the growth rate of f(n) equals (=) the growth rate of g(n) [if f(n) = O(g(n)) and T(n) = Ω (g(n)].



Little–OH (o)

T(n) = o(p(n)) (pronounced little oh), says that the growth rate of T(n) is less than the growth rate of p(n) [if T(n) = O(p(n)) and T(n) ≠ θ(p(n))].

Analyzing Algorithms

Suppose 'M' is an algorithm, and suppose 'n' is the size of the input data. Clearly the complexity f(n) of M increases as n increases. It is usually the rate of increase of f(n) we want to examine. This is usually done by comparing f(n) with some standard functions. The most common computing times are:

$O(1)$, $O(\log_2 n)$, $O(n)$, $O(n.\log_2 n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$, $n!$ and n^n

Numerical Comparison of Different Algorithms

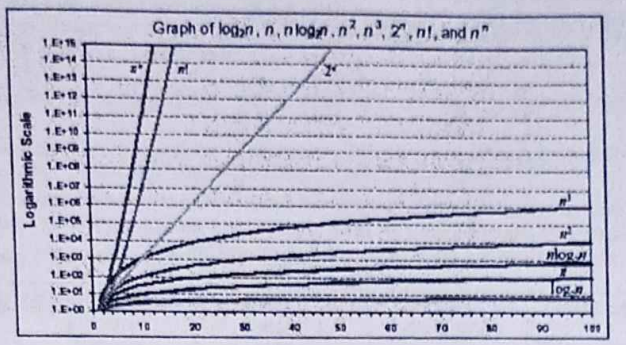
The execution time for six of the typical functions is given below:

n	$\log_2 n$	$n*\log_2 n$	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65,536
32	5	160	1024	32,768	4,294,967,296
64	6	384	4096	2,62,144	Note 1
128	7	896	16,384	2,097,152	Note 2
256	8	2048	65,536	1,677,216	????????

Note1: The value here is approximately the number of machine instructions executed by a 1 gigaflop computer in 5000 years.

Note 2: The value here is about 500 billion times the age of the universe in nanoseconds, assuming a universe age of 20 billion years.

Graph of log n, n, n log n, n^2, n^3, 2^n, n! and n^n



O(log n) does not depend on the base of the logarithm. To simplify the analysis, the convention will not have any particular units of time. Thus we throw away leading constants. We will also throw away low-order terms while computing a Big-Oh running time. Since Big-Oh is an upper bound, the answer provided is a guarantee that the program will terminate within a certain time period. The program may stop earlier than this, but never later.

One way to compare the function f(n) with these standard function is to use the functional 'O' notation, suppose f(n) and g(n) are functions defined on the positive integers with the property that f(n) is bounded by some multiple g(n) for almost all 'n'. Then,

$$f(n) = O(g(n))$$

Which is read as "f(n) is of order g(n)". For example, the order of complexity for:

- Linear search is O (n)
- Binary search is O (log n)
- Bubble sort is O (n^2)
- Merge sort is O (n log n)

The rule of sums

Suppose that $T_1(n)$ and $T_2(n)$ are the running times of two programs fragments P_1 and P_2, and that $T_1(n)$ is O(f(n)) and $T_2(n)$ is O(g(n)). Then $T_1(n) + T_2(n)$, the running time of P_1 followed by P_2 is O(max f(n), g(n)), this is called as rule of sums.

For example, suppose that we have three steps whose running times are respectively $O(n^2)$, $O(n^3)$ and O(n. log n). Then the running time of the first two steps executed sequentially is O ($max(n^2, n^3)$) which is $O(n^3)$. The running time of all three together is O(max (n^3, n. log n)) which is $O(n^3)$.

The rule of products

If $T_1(n)$ and $T_2(n)$ are $O(f(n)$ and $O(g(n))$ respectively. Then $T_1(n)*T_2(n)$ is $O(f(n)$ $g(n))$. It follows term the product rule that $O(c\ f(n))$ means the same thing as $O(f(n))$ if 'c' is any positive constant. For example, $O(n^2/2)$ is same as $O(n^2)$.
Suppose that we have five algorithms A_1–A_5 with the following time complexities:

A_1 : n
A_2 : n log n
A_3 : n^2
A_4: n^3
A_5: 2^n

The time complexity is the number of time units required to process an input of size 'n'. Assuming that one unit of time equals one millisecond. The size of the problems that can be solved by each of these five algorithms is:

Algorithm	Time complexity	Maximum problem size		
		1 second	1 minute	1 hour
A_1	n	1000	6×10^4	3.6×10^6
A_2	n log n	140	4893	2.0×10^5
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

The speed of computations has increased so much over last thirty years and it might seem that efficiency in algorithm is no longer important. But, paradoxically, efficiency matters more today then ever before. The reason why this is so is that our ambition has grown with our computing power. Virtually all applications of computing simulation of physical data are demanding more speed.

The faster the computer run, the more need are efficient algorithms to take advantage of their power. As the computer becomes faster and we can handle larger problems, it is the complexity of an algorithm that determines the increase in problem size that can be achieved with an increase in computer speed.

Suppose the next generation of computers is ten times faster than the current generation, from the table we can see the increase in size of the problem.

Algorithm	Time Complexity	Maximum problem size before speed up	Maximum problem size after speed up
A_1	n	S1	10 S1
A_2	n log n	S2	≈10 S2 for large S2
A_3	n^2	S3	3.16 S3
A_4	n^3	S4	2.15 S4
A_5	2^n	S5	S5 + 3.3

Instead of an increase in speed consider the effect of using a more efficient algorithm. By looking into the following table it is clear that if minute as a basis for comparison, by replacing algorithm A4 with A3, we can solve a problem six times larger; by replacing A4 with A2 we can solve a problem 125 times larger. These results are for more impressive than the two fold improvement obtained by a ten fold increase in speed. If an hour is used as the basis of comparison, the differences are even more significant.

We therefore conclude that the asymptotic complexity of an algorithm is an important measure of the goodness of an algorithm.

The Running time of a program

When solving a problem we are faced with a choice among algorithms. The basis for this can be any one of the following:

i. We would like an algorithm that is easy to understand, code and debug.

ii. We would like an algorithm that makes efficient use of the computer's resources, especially, one that runs as fast as possible.

Measuring the running time of a program

The running time of a program depends on factors such as:

1. The input to the program.
2. The quality of code generated by the compiler used to create the object program.
3. The nature and speed of the instructions on the machine used to execute the program, and
4. The time complexity of the algorithm underlying the program.

The running time depends not on the exact input but only the size of the input. For many programs, the running time is really a function of the particular input, and not just of the input size. In that case we define T(n) to be the worst case running time, i.e. the maximum overall input of size 'n', of the running time on that input. We also consider $T_{avg}(n)$ the average, over all input of size 'n' of the running time on that input. In practice, the average running time is often much harder to determine than the worst case running time. Thus, we will use worst–case running time as the principal measure of time complexity.

Seeing the remarks (2) and (3) we cannot express the running time T(n) in standard time units such as seconds. Rather we can only make remarks like the running time of such and such algorithm is proportional to n^2. The constant of proportionality will remain un-specified, since it depends so heavily on the compiler, the machine and other factors.

Asymptotic Analysis of Algorithms:

Our approach is based on the *asymptotic complexity* measure. This means that we don't try to count the exact number of steps of a program, but how that number grows with the size of the input to the program. That gives us a measure that will work for different operating systems, compilers and CPUs. The asymptotic complexity is written using big-O notation.

Rules for using big-O:

The most important property is that big-O gives an upper bound only. If an algorithm is $O(n^2)$, it doesn't have to take n^2 steps (or a constant multiple of n^2). But it can't

take more than n^2. So any algorithm that is O(n), is also an $O(n^2)$ algorithm. If this seems confusing, think of big-O as being like "<". Any number that is < n is also < n^2.

1. Ignoring constant factors: O(c f(n)) = O(f(n)), where c is a constant; e.g. $O(20\ n^3) = O(n^3)$

2. Ignoring smaller terms: If a<b then O(a+b) = O(b), for example $O(n^2+n) = O(n^2)$

3. Upper bound only: If a<b then an O(a) algorithm is also an O(b) algorithm. For example, an O(n) algorithm is also an $O(n^2)$ algorithm (but not vice versa).

4. n and log n are "bigger" than any constant, from an asymptotic view (that means for large enough n). So if k is a constant, an O(n + k) algorithm is also O(n), by ignoring smaller terms. Similarly, an O(log n + k) algorithm is also O(log n).

5. Another consequence of the last item is that an O(n log n + n) algorithm, which is O(n(log n + 1)), can be simplified to O(n log n).

Calculating the running time of a program:

Let us now look into how big-O bounds can be computed for some common algorithms.

Example 1:

Let's consider a short piece of source code:

```
x = 3*y + 2;
z = z + 1;
```

If y, z are scalars, this piece of code takes a *constant* amount of time, which we write as O(1). In terms of actual computer instructions or clock ticks, it's difficult to say exactly how long it takes. But whatever it is, it should be the same whenever this piece of code is executed. O(1) means *some* constant, it might be 5, or 1 or 1000.

Example 2:

$2n^2 + 5n - 6 = O(2^n)$ $2n^2 + 5n - 6 = O(n^3)$ $2n^2 + 5n - 6 = O(n^2)$ $2n^2 + 5n - 6 \neq O(n)$	$2n^2 + 5n - 6 \neq \Theta(2^n)$ $2n^2 + 5n - 6 \neq \Theta(n^3)$ $2n^2 + 5n - 6 = \Theta(n^2)$ $2n^2 + 5n - 6 \neq \Theta(n)$
$2n^2 + 5n - 6 \neq \Omega(2^n)$ $2n^2 + 5n - 6 \neq \Omega(n^3)$ $2n^2 + 5n - 6 = \Omega(n^2)$ $2n^2 + 5n - 6 = \Omega(n)$	$2n^2 + 5n - 6 = o(2^n)$ $2n^2 + 5n - 6 = o(n^3)$ $2n^2 + 5n - 6 \neq o(n^2)$ $2n^2 + 5n - 6 \neq o(n)$

Example 3:

If the first program takes $100n^2$ milliseconds and while the second takes $5n^3$ milliseconds, then might not $5n^3$ program better than $100n^2$ program?

As the programs can be evaluated by comparing their running time functions, with constants by proportionality neglected. So, $5n^3$ program be better than the $100n^2$ program.

$$5\ n^3/100\ n^2 = n/20$$

for inputs $n < 20$, the program with running time $5n^3$ will be faster than those the one with running time $100\ n^2$. Therefore, if the program is to be run mainly on inputs of small size, we would indeed prefer the program whose running time was $O(n^3)$

However, as 'n' gets large, the ratio of the running times, which is n/20, gets arbitrarily larger. Thus, as the size of the input increases, the $O(n^3)$ program will take significantly more time than the $O(n^2)$ program. So it is always better to prefer a program whose running time with the lower growth rate. The low growth rate function's such as O(n) or O(n log n) are always better.

Example 4:

Analysis of simple for loop

Now let's consider a simple for loop:

```
for (i = 1; i<=n; i++)
        v[i] = v[i] + 1;
```

This loop will run exactly n times, and because the inside of the loop takes constant time, the total running time is proportional to n. We write it as O(n). The actual number of instructions might be 50n, while the running time might be 17n microseconds. It might even be 17n+3 microseconds because the loop needs some time to start up. The big-O notation allows a multiplication factor (like 17) as well as an additive factor (like 3). As long as it's a linear function which is proportional to n, the correct notation is O(n) and the code is said to have *linear* running time.

Example 5:

Analysis for nested for loop

Now let's look at a more complicated example, a nested for loop:

```
for (i = 1; i<=n; i++)
        for (j = 1; j<=n; j++)
                a[i,j] = b[i,j] * x;
```

The outer for loop executes N times, while the inner loop executes n times for every execution of the outer loop. That is, the inner loop executes $n \times n = n^2$ times. The assignment statement in the inner loop takes constant time, so the running time of the code is $O(n^2)$ steps. This piece of code is said to have *quadratic* running time.

Example 6:

Analysis of matrix multiply

Lets start with an easy case. Multiplying two n × n matrices. The code to compute the matrix product C = A * B is given below.

```
for (i = 1; i<=n; i++)
        for (j = 1; j<=n; j++)
                C[i, j] = 0;
                for (k = 1; k<=n; k++)
                        C[i, j] = C[i, j] + A[i, k] * B[k, j];
```

There are 3 nested *for* loops, each of which runs n times. The innermost loop therefore executes n*n*n = n^3 times. The innermost statement, which contains a scalar sum and product takes constant O(1) time. So the algorithm overall takes $O(n^3)$ time.

Example 7:

Analysis of bubble sort

The main body of the code for bubble sort looks something like this:

```
for (i = n-1; i<1; i--)
        for (j = 1; j<=i; j++)
                if (a[j] > a[j+1])
                        swap a[j] and a[j+1];
```

This looks like the double. The innermost statement, the if, takes O(1) time. It doesn't necessarily take the same time when the condition is true as it does when it is false, but both times are bounded by a constant. But there is an important difference here. The outer loop executes n times, but the inner loop executes a number of times that depends on i. The first time the inner for executes, it runs i = n-1 times. The second time it runs n-2 times, etc. The total number of times the inner if statement executes is therefore:

$$(n-1) + (n-2) + \ldots + 3 + 2 + 1$$

This is the sum of an arithmetic series.

$$\sum_{i=1}^{N-1} (n-i) = \frac{n(n-i)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

The value of the sum is n(n-1)/2. So the running time of bubble sort is O(n(n-1)/2), which is $O((n^2-n)/2)$. Using the rules for big-O given earlier, this bound simplifies to $O((n^2)/2)$ by ignoring a smaller term, and to $O(n^2)$, by ignoring a constant factor. Thus, bubble sort is an $O(n^2)$ algorithm.

Example 8:

Analysis of binary search

Binary search is a little harder to analyze because it doesn't have a for loop. But it's still pretty easy because the search interval halves each time we iterate the search. The sequence of search intervals looks something like this:

$$n, n/2, n/4, \ldots, 8, 4, 2, 1$$

It's not obvious how long this sequence is, but if we take logs, it is:

$$\log_2 n, \log_2 n - 1, \log_2 n - 2, \ldots, 3, 2, 1, 0$$

Since the second sequence decrements by 1 each time down to 0, its length must be $\log_2 n + 1$. It takes only constant time to do each test of binary search, so the total running time is just the number of times that we iterate, which is $\log_2 n + 1$. So binary search is an $O(\log_2 n)$ algorithm. Since the base of the log doesn't matter in an asymptotic bound, we can write that binary search is $O(\log n)$.

General rules for the analysis of programs

In general the running time of a statement or group of statements may be parameterized by the input size and/or by one or more variables. The only permissible parameter for the running time of the whole program is 'n' the input size.

1. The running time of each assignment read and write statement can usually be taken to be $O(1)$. (There are few exemptions, such as in PL/1, where assignments can involve arbitrarily larger arrays and in any language that allows function calls in arraignment statements).

2. The running time of a sequence of statements is determined by the sum rule. I.e. the running time of the sequence is, to with in a constant factor, the largest running time of any statement in the sequence.

3. The running time of an if–statement is the cost of conditionally executed statements, plus the time for evaluating the condition. The time to evaluate the condition is normally $O(1)$ the time for an if–then–else construct is the time to evaluate the condition plus the larger of the time needed for the statements executed when the condition is true and the time for the statements executed when the condition is false.

4. The time to execute a loop is the sum, over all times around the loop, the time to execute the body and the time to evaluate the condition for termination (usually the latter is $O(1)$). Often this time is, neglected constant factors, the product of the number of times around the loop and the largest possible time for one execution of the body, but we must consider each loop separately to make sure.