Chapter 3
Analysis of Algorithms

3.1 Introduction

The analysis of computer algorithms is the process of finding the computational complexity of algorithms, i.e., the amount of time, storage, or other resources needed to execute them. It involves determining a function that relates the length of an algorithm's input to the number of steps it takes (i.e., time complexity), or the number of storage locations it uses (i.e., space complexity). An algorithm is efficient if this function's values are small, or grow slowly compared to a growth in the size of the input. Different inputs of the same length may cause the same algorithm to have different behaviors. Therefore, best, worst and average case descriptions might all be of practical interest. Sometimes, these different behaviors are not specified. In such cases, the function describing the performance of an algorithm is usually an upper bound, determined from the worst -case inputs to the algorithm. Analyzing the computational complexity of an algorithm is a very important task that must be carried out in algorithm if we must know how the algorithm behaves in different scenarios. Hence, once the complexity of an algorithm has been estimated, the question arises whether this algorithm is optimal. An algorithm for a given problem is optimal if its complexity reaches the lower bound over all the algorithms solving this problem.

Analyzing an algorithm helps us to determine the effectiveness of such an algorithm. Does the algorithm take minimum time to execute? Does the algorithm behave in different ways in different computers? Two important ways to characterize the effectiveness of an algorithm are its space complexity and time complexity. Time complexity of an algorithm concerns determining an expression of the number of steps needed as a function of the problem size. Since the step count measure is somewhat coarse, one does not aim at obtaining an exact step count. Instead, one attempt to get only the asymptotic bounds on the step count. The performance evaluation of an algorithm is obtained by totaling the number of occurrences of each operation when running the algorithm. The performance of an algorithm is evaluated as a function of the input size n and is to be considered modulo a multiplicative constant. In this chapter, we discussed asymptotic notations, time and space complexities, best, worst and average-case complexity analysis, analysis of algorithms, factors affecting the execution time of an algorithm such as input size, instruction type, machine speed, quality of the source code of the algorithm being implemented, and the quality of the machine code generate from the source code by the computer. We also discussed asymptotic order of growth, the Big-Theta, Big-Omega, and Big-O notations and their properties. We also discussed the technique for evaluating the runtime complexity of some algorithms using some proofs. Finally, we discussed amortized analysis of algorithms and the potential function (Physicist's) techniques.

3.2 Asymptotic Notations

The analysis of an algorithm is the determination of the amount of resources (such as time and space) necessary to execute them. Most algorithm are designed to work with input of arbitrary length. Usually, the efficiency or running time of an algorithm is stated as the function relating the input length to the number of steps or storage locations. Algorithm analysis is an important part of computation complexity theory, which provide theoretical estimates for the resources needed by any algorithms which solves a given computational problems. These estimates provide an insight into reasonable direction of search for efficient algorithm. Therefore, to analyze the time complexity of an algorithm, we have to rely on certain notations. The order of growth or rate of growth is a notion that gives a simple characterization of the algorithm efficiency it is simple a function, f (n) where n = the input size.

Using the order of growth, we want to examine how fast the running time of an algorithm increases when n increases. To understand this, we take few values for f (n) and n and see how they increase as n increases.

N / f(n)	$Log_2{}^n$	n	$nLog_2{}^n$	n^2	n^3	2^n
5	3	5	15	25	125	32
10	4	10	40	100	10^3	10^8
100		100		10000	10^6	
1000		1000		10^6	10^9	

For the comparison of the relative performance of several algorithms order of growth is important concept. Thus, we talk about asymptotic efficiency of algorithm, it means we concern with how the running time of an algorithm increases with the size of the input in limit, as the size of the input increases without bound.

All the asymptotic notations that we are going to discuss now describes the asymptotic running time of an algorithm in terms of functions whose domains are the set of natural numbers. e.g N {0, 1, 2, . . . 9} and combinations of those numbers.

Asymptotic Notations are **language** that allows us to analyze an algorithm running time by identifying its behavior as the input size for the algorithm increases. This is referred to as algorithms growth rate. It answers questions such as: Does the algorithm suddenly become incredibly slow when the input size grows? Does it maintain its quick run time most of the time as the input size increases? Basically, the main idea of asymptotic analysis is to have a measure of efficiency of algorithms that are independent of machine specific constants and that does not require algorithms to be implemented and time taken by programs to be compared.

Therefore, we study algorithms, we are interested in characterizing them according to their efficiency. We are usually interested in the order of growth of the running time of an algorithm not in the exact running time. This is also referred to as the asymptotic running time. Thus we need to develop a way to talk about rate of growth of functions so that we can compare algorithms. Asymptotic notation gives us a method for classifying functions according to their rate of growth. When it comes to analyzing the complexity of any algorithm in terms of time and space, we can never provide an exact number to define the time required and the space required by the algorithm, instead we express it using some standard notations called asymptotic notations

When we analyze any algorithm, we generally get a formula represent the amount of time required for execution or the time required by the computer to run the lines of code of the algorithm, number of memory accesses, number of comparisons, temporary variables occupying memory space, etc. This formula often contains unimportant details that do not really tell us anything about the running time. As an example, consider an algorithm that has a time complexity of $T(n) = n^2 + 3n + 4$

This is a quadratic equation. Thus for large values of n, the part $3n + 4$ will become too small (insignificant) compared to the n^2 part. Also, we compare the execution times of two algorithms, the constant coefficients of higher order terms are also neglected. Therefore, an algorithm that takes a time $300n^2$ will be faster than some other algorithm that takes n^3 time, for any value of n larger than 300. Since we are only interested in the asymptotic behavior of the growth of the function, the constant factor can be ignored.

Therefore, the word asymptotic means approaching a value or curve arbitrarily closely (i.e., as some sort of limit is taken). This is very similar to the concept of limit in calculus. The only difference is that in asymptotic behaviour of an algorithm, we do not have to find the value of any expression where n is approaching any finite number infinity but to use the same model to ignore the constant factors and insignificant parts of an expression, to device a better way of representing complexities of algorithms, in a single coefficient, so that comparison between algorithms can be done easily.

Given any two algorithms to solve a particular problem, how do we determine which of the two algorithms is better? One naive way of doing this is to implement both algorithms and run the two programs on your computer for different input values and see which one takes less time. There are many problems with this approach for analysis of algorithms.

i) It might be possible that for some inputs, the first algorithm performs better than the second; and for some inputs the second algorithm performs better.
ii) Also, it may be possible that for some inputs, the first algorithm performs better on one machine and the second algorithm work better on another machine for some other inputs.

Asymptotic notation is used to handle issues relating to the analysis of algorithms. In asymptotic analysis, we evaluate the performance of an algorithm in terms of input size and not the measure of the actual running time. Thus we try to determine the time (space) it takes by an algorithm as the input size increases. However, one thing is certain: as the input size increases, so is the running time and the amount of space it occupies in the computer memory. Thus it can be said that running time ($T_{(n)}$) of an algorithm is directly proportional to the input size. That is, $T_{(n)}$ is a function of $nT_{(n)} = f_{(n)}$.

Therefore, in asymptotic notation, we are usually not interested in exact running times, but only in the asymptotic classification of the running time, which ignores constant factors and constant additive offsets. That is, we are usually interested in the running times for large values of n. Then constant additive terms do not play any significant role. Again, an exact analysis (e.g. exactly counting the number of operations in a RAM) may be hard but will not lead to more precise results as the computational model is already quite a distance from reality. Also, a linear speed (i.e., by a constant factor) is always possible by for example, implementing the algorithm on a faster machine. Finally, the running time should be expressed by simple functions.

Definition (Formal): Let f denote functions from N to R^+.

i) $\odot(f) = \{g \mid \exists c > 0\ \exists n_0 \in N_0 \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$

(set of functions that asymptotically grow not faster than f).

ii) $\Omega(f) = \{g \mid \exists c > 0\ \exists n_0 \in N_0 \forall n \geq n0 : [g(v) \geq c \cdot f(n)]\}$

(set of functions that asymptotically grow not slower than f)

iii) $\emptyset(f) = \Omega(f) \cap \odot(f)$

(functions that asymptotically have the same growth as f)

iv) $O(f) = \{g \mid \forall c > 0\ \exists n_0 \in N_0 \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$

(set of functions that asymptotically grow slower than f)

v) $W(f) = \{g \mid \forall c > 0\ \exists n0 \in N0\ \forall n \geq n0 : [g(n) \geq c \cdot f(n)]\}$

(set of functions that asymptotically grow faster than f)

However, we can have an equivalent definition using limits notation (assuming that the respective limits exists). f and g are functions from No to R_0^+.

i) $g \in O(f) : 0 \leq \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$

ii) $g \in \Omega(f) : 0 \leq \lim_{n\to\infty} \frac{g(n)}{f(n)} \leq \infty$

iii) $g \in \emptyset(f) : 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$

iv) $g \in O(f) : \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

v) $g \in W(f) : \lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$

In general, asymptotic classification of running times is a good measure for comparing algorithms if the running time analysis is tight and actually occurs in practice (i.e., the asymptotic bound is not a prove theoretical worst – case bound), then the algorithm that has better asymptotic running time will always outperform a weaker algorithm for large enough values of n. However, suppose that we have two algorithms say algorithms A and B respectively. Suppose the running time of algorithm A is a function $f(n) = 1000\log n = O(\log n)$ and algorithm B with running time $g(n) = log_2^n$, clearly $f = O(g)$. However, as long as $\log n \leq 1000$, algorithm B will be more efficient.

Therefore, the analysis of algorithms is the determination of the amount of time and space resources required to execute it. Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as time complexity or volume of memory, known as space complexity. Analysis of algorithm is the process of finding the computational complexity of algorithms, that is, the amount of time, storage (memory), or other resources needed to execute them. It involves determining a function that relates the length

of an algorithms input to the number of steps it takes (i.e., its time complexity) or the number of storage locations it uses (i.e., its space complexity). In algorithm is the amount of resources required for running it. The computational complexity of a problem is the minimum of the complexities of all possible algorithms for this problem. By computation, we mean a type of calculation that includes both arithmetical and non – arithmetical steps which follows a well – defined model such as an algorithm.

Definition (Time Complexity): *The time complexity of an algorithm is the computational complexity that describes the amount of time it takes to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform. Thus the amount of time taken and the number of elementary operations performed by the algorithm differs by at most a constant factor.*

Definition (Space Complexity): *The space complexity of an algorithm or a computer program is the amount of memory space required to solve an instance of the computational problem as a function of the input size. It is the memory required by an algorithm to execute a program and produce output.*

An algorithm is efficient when the function's values are small, or grow slowly compared to a growth in the size of the input. Different input sizes produces different running time. Table 1 shows different input sizes with different running times.

n/f	*N*	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sect	4 sec
$n = 30$	< 1 sect	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	Very long
$n = 100$	< 1 sect	< 1 sec	< 1 sec	1 sec	12,892yrs	10^{17} years	Very long
$n = 1{,}000$	< 1 sec	< 1 sec	< 1 sec	18 min	Very long	Very long	Very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	Very long	Very long	Very long
100,000	< 1 see	2 sec	3 lows	32 years	Very long	Very long	Very long
1,000,000	< 1 sec	20sec	12 days	31,700 year	Very long	Very long	Very long

As seen in Table 1, as the input size increase for each function, so also the running time of each function increases. Thus different inputs of the same input size may cause an algorithm to have different behavior. Algorithms are often quite different from one another, though the objective of these algorithms are the same. For example, we know that a set of numbers can be sorted using different algorithms. Number of comparisons performed by one algorithm may vary with others for the same input. Hence, time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm.

Definition: *Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage during implementation).*

However, the main concern of analysis of algorithms is the required time or performance. Hence, algorithms are analyzed using best case, worst case and average case.

Best – Case Analysis

In the best – case analysis, we calculate lower bound on running time of an algorithm. The best case involves knowing the case that causes minimum number of operations to be executed. The best case is constant (not dependent on n). Therefore, the time complexity in the best case is usually $\theta(1)$. Usually, in the best-case analysis, we often consider the time taken by the algorithm to be the time it takes on the best input size. Also, the best-case analysis is betters. Guaranteeing a lower bound on an algorithm does not provide information as in the worst case, since an algorithm may take years to run. Therefore, the best-case algorithm is used to describe an algorithm's behavior under optimal conditions. For example, the best case for a simple linear search on a list occurs when the desired element is the first element in the list. Hence, development and choice of algorithms is seldomly based on best – case performance.

Average – Case Analysis

The average – case complexity of an algorithm is the amount of some computational resource used by the algorithm, averaged over all possible inputs. Average – case analysis requires a notion of an "average" input to an algorithm, which leads to the problem of devising a probability distribution over inputs. Alternatively, a randomized algorithms can be used. The analysis of such algorithms leads to the related notion of an expected complexity. However, it must be noted that the average – case complexity is worth determining for a number of reasons; 1) the average case complexity is a more accurate measure of an algorithm performance for intractable problems, 2) average – case complexity analysis provides tools and techniques to generate hard instances of problems which can be cartelized in areas such as cryptography and derandomization. 3) average – case complexity allows discriminating the most efficient algorithm in practice among algorithms of equivalent based case complexity.

Worst – Case Analysis

The worst – case complexity measures the resources that an algorithm requires given an input of arbitrary size. It gives an upper bound on the resources required by the algorithm. These resources could be the running time, memory and any other resource(s) that the algorithm requires given an input of arbitrary size. In the case of running time, the worst – case time – complexity indicates the longest running time performed by an algorithm given any input of size n, and thus guarantees that the algorithm will finish in the indicated period of time. The order of grow (i.e., linear, logarithmic) of the worst – case complexity is commonly used to compare the efficiency of two algorithms.

Below is a table which summarizes the time and space complexities of sorting algorithms.

Algorithm	Data Structure	Tc : Best	TC : Ave	TC : Worst	SC : Worst
Quick Sort	Array	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$	O(1)
Merge Sort	Array	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	O(1)
Heap Sort	Array	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	O(1)
Smooth Sort	Array	$O(n)$	$O(n \log(n))$	$O(n \log(n))$	O(1)
Bubble Sort	Array	$O(n)$	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	Array	$O(n)$	$O(n^2)$	$O(n^2)$	O(1)
Selection Sort	Array	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Bogo Sort	Array	$O(n)$	$O(nn!)$	$O(\infty)$	O(1)

However, it is often very difficult to estimate the behavior of most input most of the time. As a result, we content ourselves to a worst-case behavior. Most of the time, the complexity of $g(n)$ is approximated by its family $o(f(n))$ where $f(n)$ is one of the following functions. n (linear complexity), log n (logarithmic complexity), n^a where $a \geq 2$ (polynomial complexity), a^n (exponential complexity).

Worst – case analysis gives a safe analysis (the worst – case is never underestimated), but one which can be overly pessimistic since there may be no realistic input that would take many steps. Also, finding the average case complexity can be very difficult, so typically, algorithms are measured using the worst – case time complexity. In certain application domains (e.g., air traffic control, surgery) knowing the worst – case time complexity is of crucial importance.

3.3 Analyzing Algorithms

The process of determining how much computing time and storage space (memory) al algorithm will require is called algorithm analysis. Analyzing an algorithm is quite tasking. This is because, it requires high-level mathematical skills. Analyzing algorithm allows programmers and computer scientists make quantitative decisions about the efficiency of one program over the other. Usually, when analyzing algorithms, certain parameters about the efficiency of an algorithm come into play. These parameters are: (i) best-case running time, (ii) worst-case running time, and (iii) average-case running time (see section 1.1.9 for discussion). These parameters help us to determine which algorithm is more efficient and faster.

Algorithm efficiency is measured using two criteria: **space utilization** and **time efficiency.** Space utilization is the amount of space required to store data while time efficiency is the amount of time required to process the data. Before we can measure the time efficiency of an algorithm, we have to get the execution time. Execution time is the amount of time spent in executing instructions of a given algorithm. It is dependent on the particular computer (hardware) being used. To express the execution time, we use the notion:

T(n), where T is the function and n is the size of the input.

There are several factors that affect the execution time. These are:

- Input size
- Instruction type
- Machine speed
- Quality of source code of the algorithm implementation
- Quality of the machine code generated from the source code by the compiler

Input Size

The input size is a very important factor when discussing the execution time. The larger the input size, the longer it takes the code to execute. That is, the input execution time is directly proportional to the input size. It can be represented in a simple equation as shown below:

$$T(n) = f(n)$$

Instruction Type
The execution time also depends on the instruction type cum the programming language used. Normally, the execution time of machine language is expected to be faster than the execution time for an assembly language and high-level language compilers respectively. This is because, a machine language does not need an interpreter. Therefore, the time it will take a program to interpret and compile would be avoided if a program is written in machine language.

Machine Speed
The speed of a machine is also very important when discussing the execution time. The higher the configuration of a computer, the faster the program is executed hence the smaller the execution time. The reverse is true for a computer that has lower configuration. Therefore, the speed of a computer is directly proportional to the configuration of a computer. Thus a problem would be solved faster if the configuration and hence power of a computer is also very high.

Quality of the Source Code of the Algorithm Implementation
The quality of the source code is also very important when considering the execution time. A good and quality code is far better than a poorly written code. Good and quality code may be smaller in length and thus runs faster than a lengthy code using the same computer. A program written by a very experienced programmer cannot be compared with a program written by a beginner or learner. Basically, a piece of code written by a highly experienced programmer will be more efficient and run faster than a piece of code written by an amateur.

Quality of the Machine Code Generated from the Source Code by the Compiler
As seen earlier, the quality of the machine code generated is very important. A code written using Java 1.0 can never be the same as the quality of code written using Java 8.0 and higher versions. Thus modern compilers tends to generate high quality machine code from the source code compared to the quality of code generated using a lower version of a particular compiler.

As an example, consider a set of data with an input n = 100000 and t = 1msec. Then we can compute the running time of an algorithm using the function f(n) as follows.

F(n)	Running Time
$\log_2 2$	19.93 msec
n	1.00 sec
$n\log_2^n$	19.93 seconds
n^2	11.57 days
n^3	317.10centuries
2^n	Eternity

Complexity is a measure of the resources that must be expended in developing, implementing and maintaining an algorithm. Algorithms are often assessed by the execution time and the accuracy of optimality of the results. An algorithm which is complex to implement requires skilled developers, longer implementation time, and has a higher risk of implementation errors. Usually, complex algorithms ten to be highly specialized and they do not necessarily work well when the problem changes. Algorithm can be studied theoretically or empirically. Theoretical analysis allows mathematical proofs of the execution times of the algorithms but can typically be

used for worst-case analysis only. Empirical analysis is often necessary to study how an algorithm behaves using different data inputs.

To analyze the time complexity of an algorithm, we have to rely on notations. When we say we are analyzing algorithm using notations, we mean we are analyzing the resource usage of an algorithm within a constant multiple. We say "within constant multiple" because other constant multiples creep in when translating an algorithm to executable code. These are:

- Programmer ability and effectiveness
- Programming language
- Compiler
- Computer hardware

These concepts are discussed earlier in the last section.

3.4 Asymptotic Order of Growth

An algorithm can exhibit a growth rate on the order of a mathematical function if beyond a certain input size n, the function $f(n)$ times a positive constant provides an upper bound or limit for the run – time of that algorithm. That is, for a given input size n greater than some n_0 and a constant c, the running time of that algorithm will never be larger than $C \times f(n)$. This concept is often expressed as Big O notation. For example, since the run – time of insertion sort grows quadratically as its input size increases, insertion sort can be said to be of order $O(n^2)$. Thus the Big O notation is a convenient way to express the worst – case scenario for a given algorithm, although it can also be used to express the average – case – for example, the worst case scenario for quick sort is $O(n^2)$, but the average – case run – time is $O(n \log n)$.

The order of growth or rate of growth is a notion that gives a simple characterization of the algorithm's efficiency. It is simply a function f(n) where n is the input size. Using the order of growth, we want to examine how fast the running time of an algorithm increases when n increases. To understand this, we take few values for f(n) and n and see how they increase as n increases.

Table 3.1: The growth rate of some functions

N	$\log_2^*$	n	$n\log_2^*$	n^2	n^3	2^n
5	3	5	15	25	125	32
10	4	10	40	100	10^3	10^3
100	7	100	700	10^4	10^6	10^{30}
1000	10	1000	10^4	10^6	10^9	10^{300}

As another example, consider the growth of the six popular functions.

Table 3.2: The growth rate of some functions

N	Logn	n	Nlogn	n^2	n^3	2^n
4	2	4	8	16	64	16
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296

64	6	64	384	4,094	262,144	$1.84*10^{19}$
128	7	128	896	16,384	2,097,152	$3.40*10^{38}$
256	8	256	2,048	65,536	16,777,216	$1.15*10^{77}$
512	9	512	4,608	262,144	134,217,728	$1.34*10^{154}$
1024	10	1,024	10,240	1,048,576	1,073,741,824	$1.79*10^{308}$

From the tables 3.1 and 3.2, it is clearly seen that as n grows in size, the function also grows in size. For the comparison of the relative performance of several algorithms, order of growth is an important concept. The graph in figure 3.1 shows the rate of growth of common computing functions. This graph was plotted from the value in table 3.2. The graph clearly shows

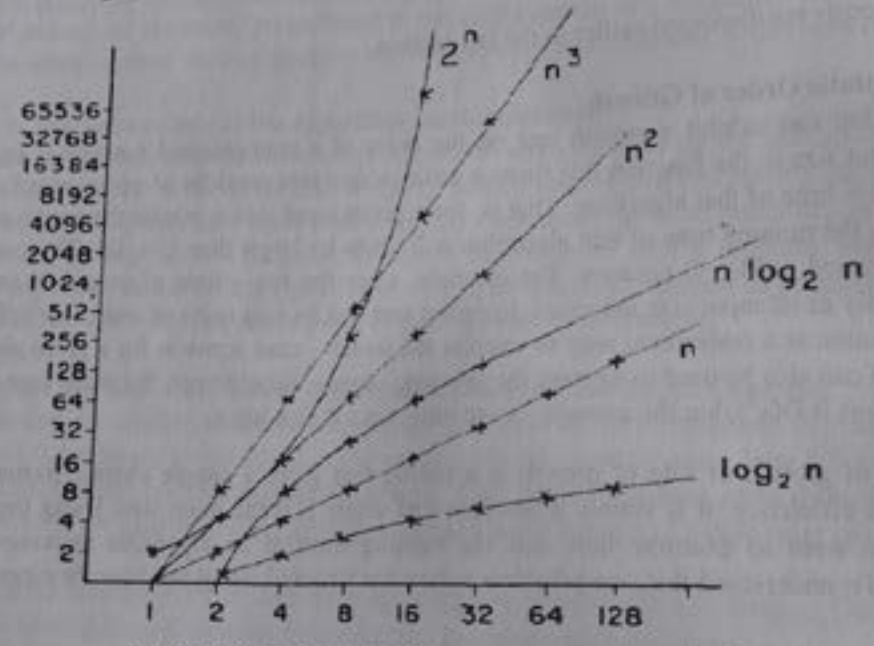

Fig. 3.1: Rate of growth of common computing time functions

Thus when we talk about asymptotic efficiency of algorithms, it means we are concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound. All the asymptotic notations that we are going to discuss now describe the asymptotic running time of an algorithm in terms of functions whose domains are the set of natural numbers:

$$N = \{0, 1, 2, \ldots\}$$

Therefore, the asymptotic order of growth is a way of comparing functions that ignores constant factors and small input sizes. Asymptotic analysis is used to evaluate the performance of an algorithm in terms of input size using the running time and space as our criteria since the running time and space of an algorithm increases as the input size increases. As an example, consider the search problem in an array. For a sorted array, the linear search is used and the order of growth is linear ($O(n)$). However, for binary search, order of grow is logarithmic, $O(n \log n)$. To understand how asymptotic analysis solves the above problems in analyzing algorithms, if we

perform searching using the linear search for a sorted array on a fast computer and binary search on a slow computer; for small values of input array size n, the fast computer may take less time. However, will start to take less time compared to the linear search even though the binary search is being run on a slow machine. The reason is the order of growth of binary search with respect to input size logarithmic which the order of growth of linear search is linear. Hence, the machine dependent constants can always be ignored after certain values of input size.

The asymptotic order of growth is divided into three (3) types. They are:

i) Big Oh(O)
ii) Big Theta
iii) Big Omega (Ω)

Big Oh (O) Notation

The Big – O notation defines an asymptotic upper bound for a function $f(n)$.

Definition (Big O): *A function $f(n)$ is said to be $O(g(n))$ if there are constants c and N such that*

$$f(n) \leq cg(n) \qquad \forall n \geq N$$

Thus by taking $g(n) = n^2$, $c = 1$ and $N = 1$, we conclude that the running time of an algorithm, say insertion sort is $O(n^2)$.

Let $f(n)$ and $g(n)$ be any non – negative functions defined on a set of all real numbers. We say $f(n) = O(g(n))$ for all functions $f(n)$ that have a lower or the same order of growth as $g(n)$, within a constant multiple as $n \to \infty$, say $100n \neq 5 \in O(n^2)$, $x^4 + n + 1 \notin O(n^2)$, $\frac{1}{2}n(n-1) \in O(n^2)$, etc.

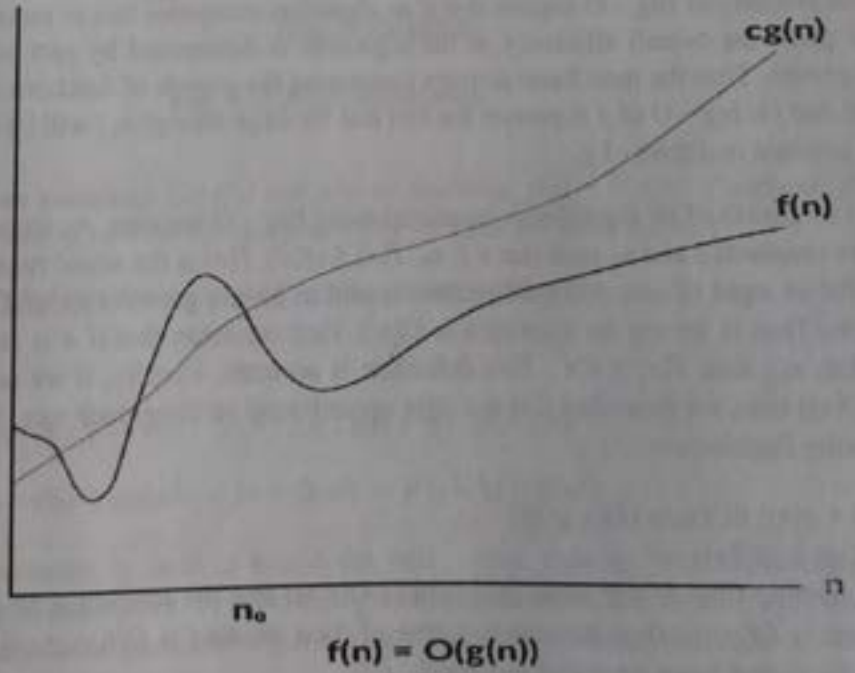

Fig. 3.2: Big-O notation

Thus if $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n) + t_2(n) \in O(\max(g_1(n)), g_2(n))$.

For any four arbitrary real numbers, a_1, b_1, a_2 and b_2 such that $a_1 \leq b_1$ and $a_1 \leq b_2$.

We have:

$$a1 + a2 \leq 2 \max \{b_1, b_2\}$$

Since $t_1(n) \in O(g_1(n))$, then there exists some constant C_1 and non – negative integer n_1 such that

$$t1(n) \leq C_1 g_1(n) \qquad \forall n \geq n_1$$

Also, since $t_2(n) \in O(g_2(n))$, then there exists some constant c_2 and non – negative integer n_2 such that

$$t_2(n) \leq c_2 g_2(n) \ \forall \text{ all } n \geq n_2$$

Let $c_3 = \max\{c_1, c_2\}$ and $n_0 = \max\{n_1, n_2\}$

$$\begin{aligned} t_1(n) + t_2(n) &\leq C_1 g_1(n) + C_2 g_2(n) \\ &\leq C_3 g_1(n) + C_3 g_2(n) \\ &= C_3 \{g_1(n) + g_2(n)\} \\ &\leq 2\, C_3 \max g_1(n), g_2(n)\} \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max \{g_1(n), g_2(n)\}$, with the constants c and n_0 required by the O definition being $2\, C_3 = 2 \max\{c_2, c_2)$ and $\max\{n_1, n_2\}$ respectively.

The above property of Big – O implies that if an algorithm comprises two or more consecutively executed parts, the overall efficiency of the algorithm is determined by part with the highest order of growth. Thus the most basic concept concerning the growth of functions is big – O. The statement that f is big – O of g expresses the fact that for large enough n, f will be bounded above by some constant multiples of g.

The order of growth of an algorithm is measured using Big – O notation. As algorithm is $O(f(n))$ if there are constants c and n_0 such that $n \geq n_0$, $T(n) \leq cf(n)$. $T(n)$ is the actual running time of the program for an input of size n. The algorithm is said to have a growth rate of $f(n)$, or to run in $O(f(n))$ time. Thus if we say an algorithm is $O(n^2)$, then we mean that if n is sufficiently large (greater than n_0), then $T(n) \leq Cn^2$. This definition is accurate. Usually, if we say an algorithm runs in $O(f(n))$ time, we mean that $f(n)$ is a tight upper bound on the growth rate. Some important rules for order function are:

1. $O(f(n) + g(n)) \in O(\max (f(n), g(n))$
2. $O(c * f(n)) \in O(f(n))$
3. If the running time of one code fragment is $O(f(n))$ and the running time of another code fragment is $O(g(n))$, then the running time of their product is $O(f(n)\ g(n))$. These rules is useful for nested loops or nested recursion

Big Theta Notation

Big theta notation (θ) allows us to state the upper bound for the growth ratio of a function. For a function $f(n)$, $f(n) \in \theta(g(n))$ if there exists a positive integer n_0 and non – negative constant c_1 and c_2 such that

$$C_1 g(n) \leq f(n) \leq C_2 g(n), \qquad \forall n \geq n_0$$

If $f(n) \in \theta(g(n))$, then the two functions have the same growth behavior. For instance, $n^2 + n + 5 \in \theta(n^2)$.

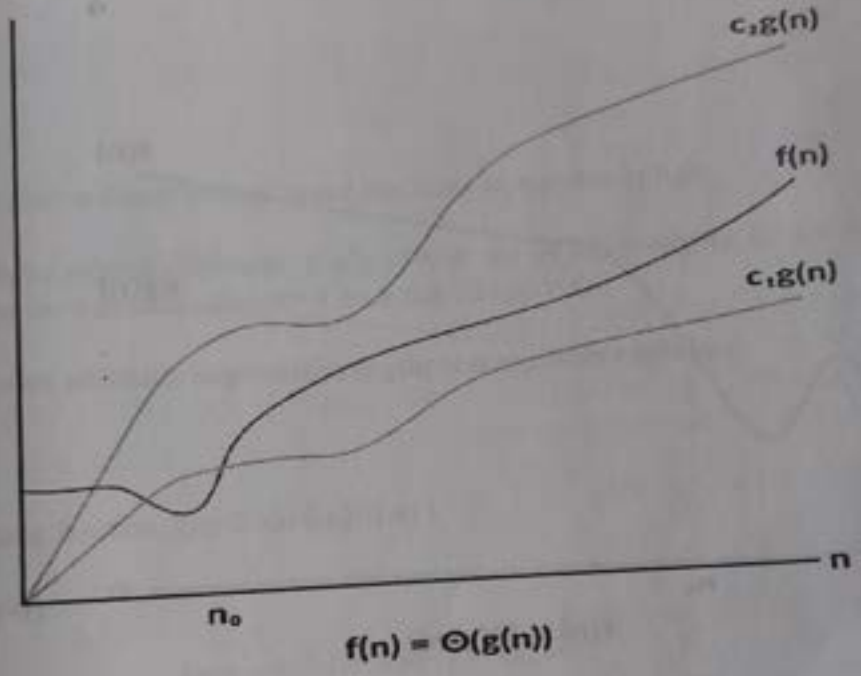

Fig. 3.3: Big-Thetanotation

Definition (Theta notation): *Let $f(n)$ and $g(n)$ be functions, $f(n) = \theta(g(n))$ if and only if positive constants c_1, c_2 and n_0 exist such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$, for all n, $n \geq n_0$.*

We unite $f(n) = \Omega(g(n))$ implies $f(n)$ is a member of the set $\Omega(g(n))$.

For example:

$$n^2/_2 - 3n = \emptyset(n^2) \rightarrow n^2/_2 - 3n = O(n^2) \text{ and } n^2/_2 - 3n = \Omega(n^2)$$

$$n^2/_2 - 3n = O(n^2) \text{ and } n^2/_2 - 3n = \Omega(n^2) \rightarrow n^2/_2 - 3n = \emptyset(n^2)$$

Thus the Ω – notation is used to bound the best – case running time of an algorithm. An algorithm is $\Omega(g(n)$ means that: 1) the running time is at least constant times $g(n)$, for sufficiently large n, 2) no matter what particular input of size n is chosen for each value of n.

Omega Notation

The omega (Ω) notation provides an asymptotic lower bound on a function to within a constant factor. That is, the Omega notation describes the lower bound of the function f. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$. In the set notation, we write as follows: For a given function $g(n)$, the set of functions

Definition (Omega notation): *Let $f(n)$ and $g(n)$ be functions, $f(n) = \Omega(g(n))$ if and only if positive constants c and n_0 exist such that $f(n) \geq c\, g(n)$ for all n, $n \geq n_0$.*

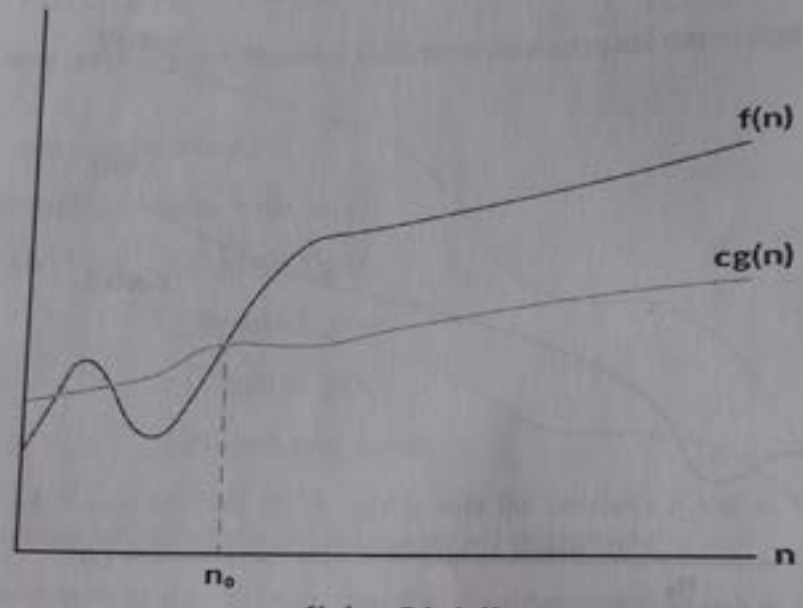

Fig. 3.4: Big-Omega notation

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$. We say that the function $g(n)$ is an asymptotic lower bound for the function $f(n)$.

Little-o Notation

We use **o-notation** to denote an upper bound that is not asymptotically tight.

Definition *Formally define **o(g(n))** (little-oh of g of n) as the set **f(n) = o(g(n))** for any positive constant $c>0$ and there exists a value $n0>0$, such that $0 \leq f(n) < c.g(n)$.*

Therefore,, in the **o-notation**, the function ***f(n)*** becomes insignificant relative to $g(n)$ as n approaches infinity; that is,

$$\lim n \to \infty (f(n)g(n)) = 0$$

Example

Let us consider the function, $f(n)=2.n3+6.n2+5.n+1$

Considering $g(n)=n2$

$$\lim n\to\infty(2.n3+6.n2+5.n+1n4)=0$$

Hence, the complexity of ***f(n)*** can be represented as $o(g(n))$

, i.e. $o(n2)$

ω-Notation

The **ω-notation**is used to denote a lower bound that is not asymptotically tight.

Definition: *We define* ***ω(g(n))*** *(little-omega of g of n) as the set* ***f(n) = ω(g(n))*** *for any positive constant* ***C > 0*** *and there exists a value n0>0*, such that $0 \square c.g(n)<f(n)$

That is, ***f(n)*** becomes arbitrarily large relative to ***g(n)*** as **n** approaches infinity.

Example

Let us consider same function, $f(n)=2.n3+6.n2+5.n+1$

Considering $g(n)=n2$

$$\lim n\to\infty(2.n3+ 6.n2+5.n+1n2)=\infty$$

Hence, the complexity of ***f(n)*** can be represented as $o(g(n))$

, i.e. $\omega(n2)$

3.4.1 Apriori and Apsotiari Analysis

Apriori analysis of an algorithm is analysis is performed prior to running it on a specific system. This analysis is a stage where a function is defined using some theoretical model. Hence, we determine the time and space complexity of an algorithm by just looking at the algorithm rather than running it on a particular system with a different memory, processor, and compiler. In Apriori, it is the reason that we use asymptotic notations to determine time and space complexity as they change from computer to computer; however, asymptotically they are the same. Apostiari analysis of an algorithm is the process of performing algorithm analysis only after running it on a system. It directly depends on the system and changes from system to system. In an industry, we cannot perform Apostiari analysis as the software is generally made for an anonymous user, which runs it on a system different from those present in the industry.

3.4.2 Properties of Big – O, Theta and Omega Notations

In this section, we describe the properties of Big – O, Theta and Omega notations. These properties are listed below:

(i) $f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n)) \Leftrightarrow f(n) \in \theta(g(n))$ this gives an alternative way to show a big Theta relationship

(ii) Transitivity: If $f(n) \in \theta(g(n))$ and $y(n) \in \theta(h(n))$ then $f(n) \in \theta(h(n))$. This is also true for big – O and big Omega notations.

(iii) (Symmetry of Theta): $f(n) \in \theta(g(n)) \Leftrightarrow g(n) \leftarrow \theta(f(n))$.

(iv) (Transpose Symmetry for big O and Omega): $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$

(v) If $(n) \sim g(n)$ then $f(n) \in \theta(g(n))$.

(vi) If $p(n)$ is a polynomial of degree then $p(n) \in \theta(n^k)$.

(vii) For any positive constant e, we have $c.\ f(n) \in \theta(f(n))$.

(viii) For any positive constant c, we have $\log_c(n) \in \theta(\log_2(n))$

(ix) For any constant k, $\log^k(n) \in O(n)$. Actually, it is in $O(\sqrt{n})$ for any $a > 0$.

3.4.3 Evaluating the Run–Time Complexity

The run-time complexity for the worst-case scenario of a given algorithm can sometimes be evaluated by examining the structure of the algorithm and making some simplifying assumptions. Consider the pseudocode below:

```
1. get a positive integer from input
2. if n > 10
3.     print "This might take a while"
4. for i = 1 to n
5.     for j = 1 to i
6.         print i * j
7. print "Done!"
```

A given computer will take a discrete amount of time to execute each of the instructions involved in this algorithm. The specific amount of time to carry out a given instruction will vary depending on which instruction is being executed and which computer is executing it, but on a conventional computer, this amount will be deterministic. Let the actions carried out in step 1 be T_1 step 2 be T_2, etc., for the time used to execute each line of the algorithm.

In the algorithm above, steps 1, 2 and 7 will only be run once. For a worst – case evaluation, we can also assume that step 3 will be run as will. Thus the total amount of time to run steps 1 – 3 and step 7 is:

$$T_1 + T_2 + T_3 + T_7$$

The loops in steps 4, 5 and 6 are somewhat difficult to evaluate. The outer loop test in step 4 will execute $(n + 1)$ times. This will consume $T_4 (n + 1)$ time. The inner loop on the other hand is governed by the value of j, which iterates from 1 to i. On the first pass through the outer loop, j iterates from 1 to i. The inner loop makes one pass so running the inner loop body (step 6) consumes T_6 time, the inner loop test (step 5) consumes $2T_5$ time. During the next pass through the outer loop, j iterates from 1 to 2, the inner loop makes two passes; so running the inner loop body (step 6) consumes $2T_6$ time, and the inner loop test (step 5) consumes $3T_5$ time.

In all, the total time required to run the inner loop body can be expressed as an arithmetic progression:

$$T_6 + 2T_6 + 3T_6 + \ldots + (n-1)T_6 + nT_6$$

which can be factored as

$$T_6[1 + 2 + 3 + \ldots + (n-1) + n] = T_6\left[\frac{1}{2}(n^2 + n)\right]$$

The total time required to run the outer loop test can be evaluated similarly

$$2T_5 + 3T_5 + 4T_5 + \ldots + (n-1)\,T_5 + nT_5 + (n+1)\,T_5$$

$$= T_5 + 2T_5 + 3T_5 + 4T_5 + \ldots + (n-1)\,T_5 + nT_5 + (n-1)T_5 - T_5$$

Factorizing out T_5, we have

$$T_5(1 + 2 + 3 + \ldots + (n-1) + n + (n+1)] - T_5$$

$$= \left[\frac{1}{2}(n^2 + n)\right]T_5 + (n+1)T_5 - T_5$$

$$= T_5\left[\frac{1}{2}(n^2 + n)\right] + nT_5$$

$$= \left[\frac{1}{2}(n^2 - 3n)\right]T_5$$

Hence, the total running time for this algorithm is:

$$f(n) = T_1 + T_2 + T_3 + T_7 + (n+1)T_4 + \left[\frac{1}{2}(n^2 + n)\right]T_6 + \left[\frac{1}{2}(n^2 + 3n)\right]T_5$$

which reduces to

$$f(n) = \left[\frac{1}{2}(n^2 + n)\right]T_6 + \left[\frac{1}{2}(n^2 + 3n)\right]T_5 + (n+1)T_4 + T_1 + T_2 + T_3 + T_7$$

As a rule – of – thumb, one can assume that the highest – order term in any given function dominates its rate of growth and thus defines the run – time order. In this example, n^2 is the highest – order term, so one can conclude that $f(n) = O(n^2)$. Formally this can be proven as follows:

Theorem: Prove that $\left[\frac{1}{2}(n^2 + n)\right]T_6 + \left[\frac{1}{2}(n^2 + 3n)\right]T_5 + (n+1)T_4 + T_1 + T_2 + T_3 + T_7 \leq Cn^2 n \geq n_0$

Proof

$\left[\frac{1}{2}(n^2+n)\right]T_6 + \left[\frac{1}{2}(n^2+3n)\right]T_5 + (n+1)T_4 + T_1 + T_2 + T_3 + T_7 \leq (n^2+n)T_6 + (n^2+3n)T_5 + (n+1)T_4 + T_1 + T_2 + T_4 + T_7$ (for $n \geq 0$).

Let k be a constant greater than or equal to $[T_1, T_7]$.

Therefore,

$$(n^2+n) + T_5(n^2+3n) + (n+1)T_4 + T_1 + T_2 + T_3T_7 \leq k(n^2+n) + (n^2+3n) + kn5k$$

$$= 2kn^2 + 5kn + 5k \leq 2kn^2 + 5kn^2 + 5kn^2 \text{ (for } n \geq 1) = 12kn^2$$

Therefore, $\left[\frac{1}{2}(n^2+n)\right]T_6 + \left[\frac{1}{2}(n^2+3n)\right]T_5 + (n+1)T_4 + T_1 + T_2 + T_3 + T_7 \leq Cn^2, n \geq n_0$

forc = $12k$, $n_0 = 1$.

Let $[T_1, T_2, \ldots, T_7]$ be equal to one unit of time so that one unit is greater than or equal to the actual to the actual times for these steps. This will make the algorithm running time to break down to:

$$4 + \sum_{i=1}^{n} i \leq 4 + \sum_{i=1}^{n} = 4 + n^2 \leq 5n^2 (for\ n \geq 1)$$

$= O(n^2)$.

Example 1: Prove that the running time $T(n) = n^3 + 20n + 1$ is $O(n^3)$

Proof

By the Big – O definition, $T(n)$ is $O(n^3)$ if $T(n) \leq Cn^3$ for some $n \geq n_0$. Let us check this condition: If $n^3 + 20n + 1 \leq c.n^3$ then $1 + \frac{20}{n^2} + \frac{1}{n^3} \leq c$. Therefore, the Big – O condition holds for $n \geq n_0 = 1$ and $c \geq 22 = (1 + 20 + 1)$. Larger values of n_0 result in smaller factors c (e.g., for $n_0 = 10$ $c \geq 1.201$, etc) but win any case the above statement is valid.

Example 2: Prove that the running time $T(n) = n^3 + 20n + 1$ is not $O(n^2)$

Proof

By the Big – O notation, $T(n)$ is $O(n^2)$ if $T(n) \leq cn^2$ for some $n \geq n_0$. Let us check this condition: if $n^3 + 20n + 1 \leq cn^2$, then $n + \frac{20}{n} + \frac{1}{n^2} \leq c$. Therefore, the Big – O condition cannot hold (the left side of the latter inequality is growing infinitely, so that there is no such constant factor c).

Example 3: Prove that the running time $T(n) = n^3 + 20n + 1$ is $O(n^4)$

In conclusion, calculate amortized cost of a single operation in the context of a sequence of operations. The three ways to do the analysis are:

- Aggregate method (brute force sum)
- Banker's method (tokens)
- Physicist's method (potential function, Ø)

Again, nothing changes in the code: runtime analysis only.

Exercises

1. What do you understand by analysis of algorithms? Why is analysis of algorithms important in Computer Science
2. Explain the concept of Big-O notation. Differentiate between Big-O, Omega, and Theta notations using suitable diagrams.
3. Explain time and space complexities of algorithms. What do you understand by best case time complexity, average case, and worst case time complexities.
4. What is asymptotic notations? Mention and explain the types of asymptotic notations you know.
5. Explain the concept of Kolmogorov complexity K(n) of a natural number. show that there is no constant n such that n > N implies K(n) > logloglogloglogN.
6. What is Little o notation? What do you understand by apriori and posteori in the analysis of algorithms?
7. What are the properties of Big-O, Omega-O , and Theta notations?
8. Prove that the running time $T(n) = n^3 + 20n + 1$ is $O(n^3)$
9. Prove that the running time $T(n) = n^3 + 20n + 1$ is not $O(n^2)$
10. What do you understand by amortized analysis of algorithm? What is amortized time complexity of an algorithm?
11. State and explain using suitable diagrams the various approaches used in amortized analysis of time complexity of the average running time of an algorithm.