SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF INFORMATION TECHNOLOGY

UNIT - I

Design and Analysis of Algorithm – SCSA1403

Introduction **9 Hrs.**

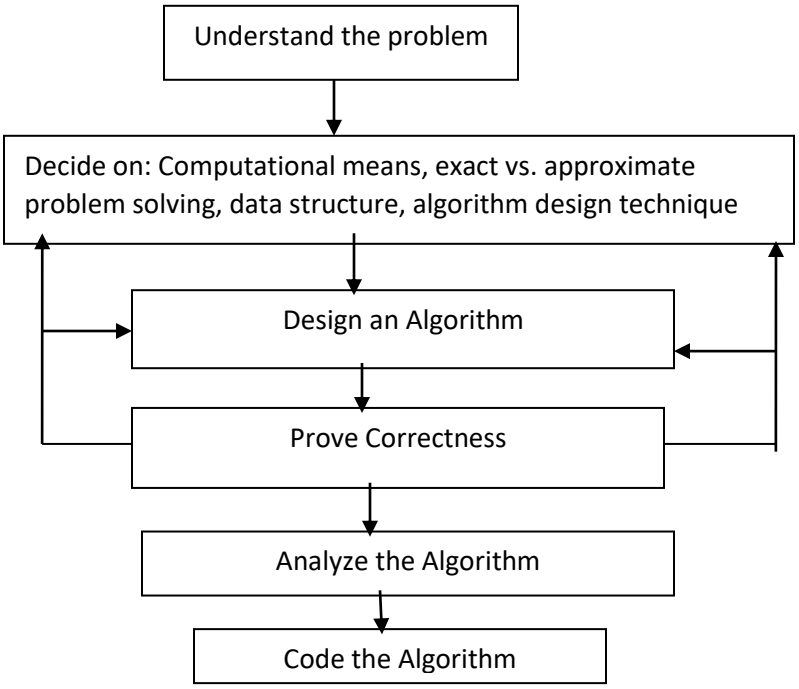
Fundamentals of Algorithmic Problem Solving - Time Complexity - Space complexity with examples - Growth of Functions - Asymptotic Notations: Need, Types - Big Oh, Little Oh, Omega, Theta - Properties - Complexity Analysis Examples - Performance measurement - Instance Size, Test Data, Experimental setup.

Fundamentals of Algorithmic Problem Solving

An Algorithm is a finite sequence of instructions or steps (i.e. inputs) to achieve a particular task. All algorithms must satisfy the following criteria:

1. Input- zero or more quantities are externally supplied.
2. Output- At least one quantity is produced.
3. Definiteness-Each instruction is clear and unambiguous.
4. Finiteness- The algorithm terminates after a finite number of steps.
5. Effectiveness- the degree to which something is successful in producing a desired result.

Algorithms can be considered to be procedural solutions to problems. There are certain steps to be followed in designing and analyzing an algorithm



1. **Understanding the problem**:

   The problem given should be understood completely. Check if it is similar to some standard problems and if a Known algorithm exists, otherwise a new algorithm has to be devised.

2. **Ascertain the capabilities of the computational device**: Once a problem is understood we need to know the capabilities of the computing device this can be done by knowing the type of the architecture, speed and memory availability.

3. **Exact /approximate solution:** Once algorithm is devised, it is necessary to show that it computes answer for all the possible legal inputs.

4. **Deciding data structures** : Data structures play a vital role in designing and analyzing the algorithms. Some of the algorithm design techniques also depend on the structuring data specifying a problem's instance.

   Algorithm + Data structure = Programs

5. **Algorithm design techniques**: Creating an algorithm is an art which may never be fully automated. By mastering these design strategies, it will become easier for you to devise new and useful algorithms.

6. **Prove correctness:**

   Correctness has to be proved for every algorithm. For some algorithms, a proof of correctness is quite easy; for others it can be quite complex. A technique used for proving correctness by mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs. But we need one instance of its input for which the algorithm fails. If it is incorrect, redesign the algorithm, with the same decisions of data structures design technique etc

7. **Analyze the algorithm**

   There are two kinds of algorithm efficiency: time and space efficiency. Time efficiency indicates how fast the algorithm runs; space efficiency indicates how much extra memory the algorithm needs.

8. **Coding**

   Programming the algorithm by using some programming language. Formal verification is done for small programs. Validity is done by testing and debugging. Inputs

should fall within a range and hence require no verification. Some compilers allow code optimization which can speed up a program by a constant factor whereas a better algorithm can make a difference in their running time. The analysis has to be done in various sets of inputs.

Complexity

Performance of a program: The performance of a program is measured based on the amount of computer memory and time needed to run a program.

The two approaches which are used to measure the performance of the program are:

1. **Analytical method** → called the Performance Analysis.
2. **Experimental method** → called the Performance Measurement.

Space Complexity

Space complexity: The Space complexity of a program is defined as the amount of memory it needs to run to completion.

As said above the space complexity is one of the factor which accounts for the performance of the program. The space complexity can be measured using experimental method, which is done by running the program and then measuring the actual space occupied by the program during execution. But this is done very rarely. We estimate the space complexity of the program before running the program.

The **reasons for estimating the space complexity** before running the program even for the first time are:

(1) We should know in advance, whether or not, sufficient memory is present in the computer. If this is not known and the program is executed directly, there is possibility that the program may consume more memory than the available during the execution of the program. This leads to insufficient memory error and the system may crash, leading to severe damages if that was a critical system.

(2) In Multi user systems, we prefer, the programs of lesser size, because multiple copies of the program are run when multiple users access the system. Hence if the program occupies less space during execution, then more number of users can be accommodated.

Space complexity is the sum of the following components:

(i) Instruction space:

The program which is written by the user is the source program. When this program is compiled, a compiled version of the program is generated. For executing the program an executable version of the program is generated. The space occupied by these three when the program is under execution, will account for the instruction space.

The instruction space depends on the following factors:

- Compiler used – Some compiler generate optimized code which occupies less space.
- Compiler options – Optimization options may be set in the compiler options.
- Target computer – The executable code produced by the compiler is dependent on the processor used.

(ii) Data space:

The space needed by the constants, simple variables, arrays, structures and other data structures will account for the data space.

The Data space depends on the following factors:

- Structure size – It is the sum of the size of component variables of the structure.
- Array size – Total size of the array is the product of the size of the data type and the number of array locations.

(iii) Environment stack space:

The Environment stack space is used for saving information needed to resume execution of partially completed functions. That is whenever the control of the program is transferred from one function to another during a function call, then the values of the local variable of that function and return address are stored in the environment stack. This information is retrieved when the control comes back to the same function.

The environment stack space depends on the following factors:

- Return address
- Values of all local variables and formal parameters.

The Total space occupied by the program during the execution of the program is the sum of the fixed space and the variable space.

(i) **Fixed space** - The space occupied by the instruction space, simple variables and constants.

(ii) **Variable space** – The dynamically allocated space to the various data structures and the environment stack space varies according to the input from the user.

$$\boxed{\textit{Space complexity } S(P) = c + S_p}$$

c -- Fixed space or constant space

S_p -- Variable space

We will be interested in estimating only the variable space because that is the one which varies according to the user input.

Consider the following piece of code...

```
int square(int a)
{
        return a*a;
}
```

That means, totally it requires 4 bytes of memory to complete its execution. And this 4 bytes of memory is fixed for any input value of 'a'. This space complexity is said to be ***Constant Space Complexity*.**

If any algorithm requires a fixed amount of space for all input values then that space complexity is said to be Constant Space Complexity.

Consider the following piece of code...

```
int sum(int A[], int n)
{
  int sum = 0, i;
  for(i = 0; i < n; i++)
    sum = sum + A[i];
  return sum;
}
```

In above piece of code it requires '**n*2**' bytes of memory to store array variable '**a[]**' 2 bytes of memory for integer parameter '**n**' 4 bytes of memory for local integer variables '**sum**' and '**i**' (2 bytes each) 2 bytes of memory for **return value**.

That means, totally it requires '2n+8' bytes of memory to complete its execution. Here, the amount of memory depends on the input value of 'n'. This space complexity is said to be ***Linear Space Complexity*.**

```
1.  Algorithm Rsum (a, n)
2.  {
3.      if (n <= 0) then return 0.0;
4.      else return Rsum ((a, n-1) + a[n] );
5.  }
```

Recursive function for sum:

In the above algorithm instances are characterized by n. the recursion stack space includes space for the formal parameters, the local variables, and the return address. Assume that the return address requires only 2 byte of memory. Each call to Rsum requires at least 3 * 2 = 6 byte (including space for the value of n, the return address, and a pointer to a[]). Since the depth of recursion is n+1, the recursion state space needed is >=3*2(n+1) = 6(n+1).

Time Complexity

Time complexity: Time complexity of the program is defined as the amount of computer time it needs to run to completion.

The time complexity can be measured, by measuring the time taken by the program when it is executed. This is an experimental method. But this is done very rarely. We always try to estimate the time consumed by the program even before it is run for the first time.

The **reasons for estimating the time complexity** of the program even before running the program for the first time are:

(1) We need real time response for many applications. That is a faster execution of the program is required for many applications. If the time complexity is estimated beforehand, then modifications can be done to the program to improve the performance before running it.

(2) It is used to specify the upper limit for time of execution for some programs. The purpose of this is to avoid infinite loops.

The time complexity of the program depends on the following factors:

- *Compiler used* – some compilers produce optimized code which consumes less time to get executed.
- *Compiler options* – The optimization options can be set in the options of the compiler.
- *Target computer* – The speed of the computer or the number of instructions executed per second differs from one computer to another.

The total time taken for the execution of the program is the sum of the compilation time and the execution time.

(i) **Compile time** – The time taken for the compilation of the program to produce the intermediate object code or the compiler version of the program. The compilation

time is taken only once as it is enough if the program is compiled once. If optimized code is to be generated, then the compilation time will be higher.

(ii) **Run time or Execution time** - The time taken for the execution of the program. The optimized code will take less time to get executed.

$$\boxed{\textit{Time complexity } T(P) = c + T_p}$$

c -- Compile time

T_p -- Run time or execution time

We will be interested in estimating only the execution time as this is the one which varies according to the user input.

So the time T(p) taken by a program p is the sum of the compile time and the run time. The compile time does not depend on the instance characteristics. But run time is depending on the instance characteristics. This run time is denoted by tp(instance characteristics).

The many of the factors tp depends on the number of additions, subtractions, multiplications, divisions, compares, loads, stores and so on, so tp(n) of the form

$$Tp(n) = Ca\ ADD(n) + Cs\ SUB(n) + Cm\ MUL(n) + Cd\ DIV + \ldots..$$

Where n denotes the instance characteristics, and Ca, Cs, Cm, Cd and so on respectively, denote the time needed for an addition, subtraction, multiplication, division, and so on and ADD, SUB, MUL, DIV and so on are functions whose values are the numbers of additions, subtractions, multiplications, divisions, and so on.

The Tp(n) is obtain a count for the total number of operations. To obtain number of operations, just count only the number of program steps. A program step is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics.

The number of steps any program statement is assigned depends on the kind of statement. For example, comments count as zero step, an assignment statement which does not involve any calls to other algorithms is counted as one step, in an iterative statement such as the for, while and repeat until statement, we consider the step counts only for the control part of the statement.

The control parts for For and while statements have the following forms.

For i = <expr> to <expr1> do While<expr>do

Each execution of the control part of a while statement is given a step count equal to the number of step counts assignable to <expr>. The step count for each execution of the control part of a for statement is one.

We can determine the number of steps needed by a program to solve a particular problem instance is one of two ways. In the first method, we introduce a new variable, count, into the program. This is a global variable with initial value 0. each time a statement is executed, count is incremented by one.

Example:

When the statements to increment count are introduced then the algorithm will be

```
Algorithm Sum(a, n)
{
 s: = 0.0;
 //count = count + 1 - count is global, it is initially zero
 for i=1 to n do
 {
          //count = count + 1 - For for
          s = s + a[i];  //count = count + 1 - for assignment
  }
 count = count +1  //for last time of for
 count = count +1  // for the return
 return s;
}
```

for every initial value of count, the above algorithm compute the same finial value for count. It is easy to see that in the for loop, the value of count will increase by a total of 2n. if count is zero to start with, then it will be(2n + 3) on termination. So each invocation of sum (the above algorithm) executes a total of (2n + 3) steps.

Example 2 :

When the statements to increment count are introduced in Recursive function for sum ,we will get the following algorithm.

```
Algorithm Rsum (a, n)
{
 // count = count + 1 - for the if conditional
 if (n<= 0) then
 {
```

```
        // count = count + 1 -for the return
        return 0.0;
    }
    else
    {
        // count = count +1- for the addition, function invocation and return
        return Rsum (a, n-1) + a[n];
    }
}
```

let tRsum (n) be the count value when above algorithm is terminates.

We can see that tRsum(0) = 2, if n = 0. when n > 0, count increase by 2 plus whatever increase result from the invocations of Rsum from within the else clause. From the definition of tRsum, it follows that this additional increase is tRsum (n-1), so if the value of count is zero initially, its value at the time of termination is (2 + tRsum (n-1)), n > 0.

When analyzing a recursive program for its step count, we often obtain a recursive formula for the step count, for example.

$$tRsum(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + tRsum(n-1) & \text{if } n > 0 \end{cases}$$

These recursive formulas are referred to as recurrence relations. One way to solve the recurrence relation is

$$\begin{aligned} tRsm(n) &= 2 + tRsum(n-1) \\ &= 2 + 2 + tRsum(n-2) \\ &= 2(2) + tRsum(n-2) \\ &= n(2) + tRsum(0) \\ &= 2n + 2 \qquad n >= 0 \end{aligned}$$

so the step count for Rsum is 2n + 2.

The second method is determined the stop count of an algorithm is to build a table in which we list the total number of steps contributed by each statement. This figure is often arrived at by first

determining the number **of steps per execution (s/e) of the statements** and the total number of times (is frequency) each statement is executed.

The (s/e) of a statement is the amount by which the count changes as a result of the execution of that statement, by combining these two quantities, the total contribution of each statement is obtained. By adding the contribution of all statements, the step count for the entire algorithm is obtained.

In table 1, the number of steps per execution and the frequency of each of the statements in sum have been listed. The total number of step required by the algorithm is determined to be (2n + 3). It is important to note that the frequency of the for statement is (n + 1) and not n. This is so because i has to be incremented to (n + 1) before the for loop can terminate

Table 1:

Statement	s/e	frequency	Total steps
Algorithm Sum (a,n)	0	------	0
{	0	------	0
s: =0.0;	1	1	1
for i = 1 to n do	1	(n + 1)	(n + 1)
s = s + a[i];	1	n	n
return s;	1	1	1
}	0	------	0
Total			(2n + 3)

In the table 2, is gives the steps count for Rsum for the algorithm 2. Notice that under the s/e column, the else clause has been given a count of (1 + tRsum (n-1)). This is the total cost of this time each time it is executed. If includes all the steps that get executed as a result of the invocation of Rsum from the else clause. The frequency and total steps column have been split into two parts.

One for the case (n = 0) and other for the case (n > 0). this is necessary because the frequency for some statements is different for each of these cases.

Table 2:

Statement	s/e	frequency		total steps	
	n = 0	n > 0		n = 0	n > 0
Algoirhtm Rsum (a, n)	0	-	-	0	0

{					
if (n <=0) then	1	1	1	1	1
return 0.0;	1	1	0	1	0
else return Rsum (a, n-1) + a[n];	1 + x	0	1	0	1 + x
}	0	-	-	0	0
Total				2	(2 + x)

Where x = tRsum(n-1)

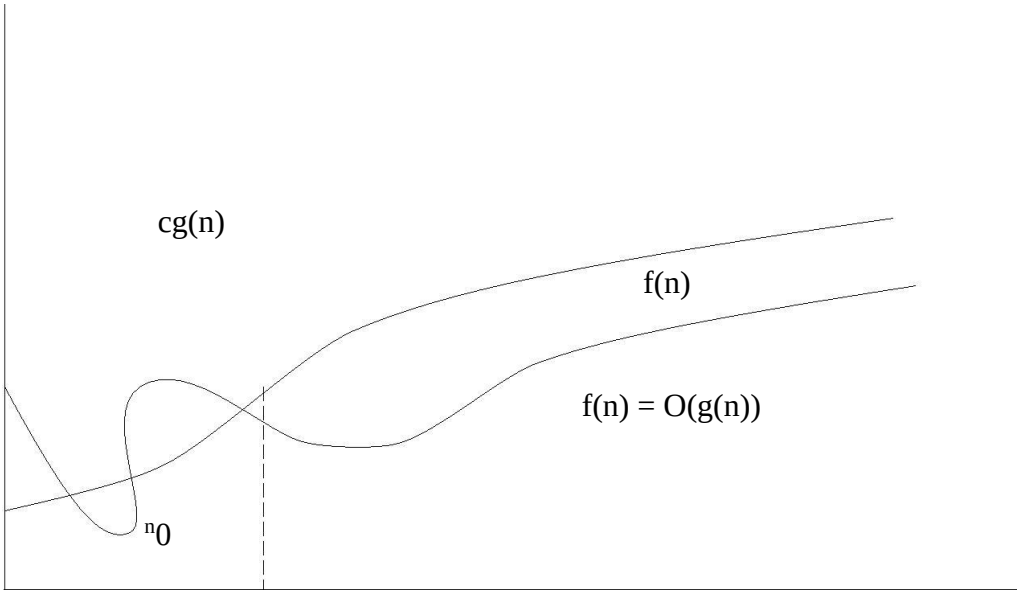
Growth of Functions and Aymptotic Notation

- When we study algorithms, we are interested in characterizing them according to their efficiency.
- We are usually interesting 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*.
- 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.

Big-O Notation

- **Definition:** f (n) = O(g(n)) iff there are two positive constants c and n_0 such that
  $$|f(n)| \leq c\,|g(n)| \text{ for all } n \geq n_0$$
- If f (n) is nonnegative, we can simplify the last condition to
  $$0 \leq f(n) \leq c\,g(n) \text{ for all } n \geq n_0$$
- We say that " f (n) is big-O of g(n)."

As n increases, f (n) grows no faster than g(n). In other words, g(n) is an *asymptotic upper bound* on f (n).



Example: $n^2 + n = O(n^3)$

Proof:

- Here, we have $f(n) = n^2 + n$, and $g(n) = n^3$
- Notice that if $n \geq 1$, $n \leq n^3$ is clear.
- Also, notice that if $n \geq 1$, $n^2 \leq n^3$ is clear.
- **Side Note:** In general, if $a \leq b$, then $n^a \leq n^b$ whenever $n \geq 1$. This fact is used often in these types of proofs.
- Therefore,

$$n^2 + n \leq n^3 + n^3 = 2n^3$$

- We have just shown that

$$n^2 + n \leq 2n^3 \text{ for all } n \geq 1$$

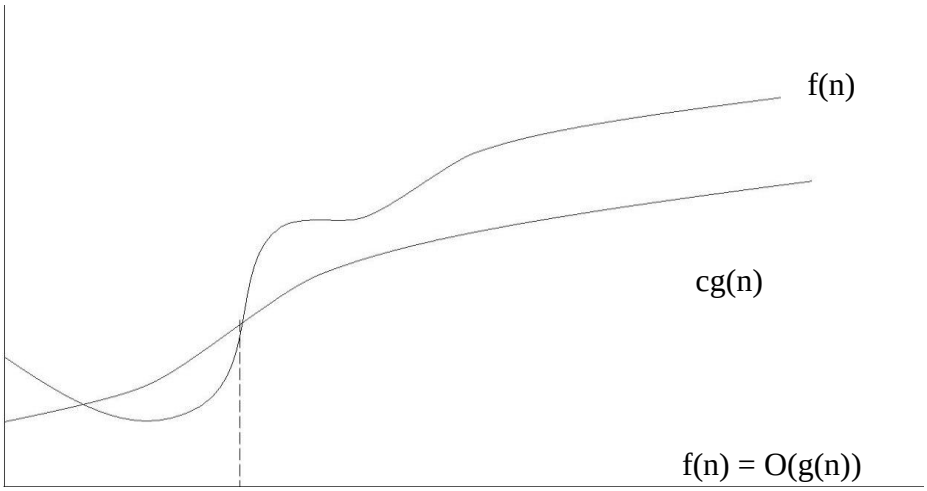
- Thus, we have shown that $n^2 + n = O(n^3)$ (by definition of Big- O, with $n_0 = 1$, and $c = 2$.)

Ω notation

- **Definition:** $f(n) = \Omega(g(n))$ iff there are two positive constants c and n_0 such that
$$|f(n)| \geq c\,|g(n)| \text{ for all } n \geq n_0$$
- If f (n) is nonnegative, we can simplify the last condition to
$$0 \leq c\,g(n) \leq f(n) \text{ for all } n \geq n_0$$
- We say that " f (n) is omega of g(n)."

- As n increases, f (n) grows no slower than g(n). In other words, g(n) is an *asymptotic lower bound* on f (n).

•



Example: $n^3 + 4n^2 = \Omega(n^2)$

Proof:

- Here, we have f (n) = $n^3 + 4n^2$, and g(n) = n^2
- It is not too hard to see that if $n \geq 0$,

$$n^3 \leq n^3 + 4n^2$$

- We have already seen that if $n \geq 1$,

$$n^2 \leq n^3$$

- Thus when $n \geq 1$,

$$n^2 \leq n^3 \leq n^3 + 4n^2$$

- Therefore,

$$1n^2 \leq n^3 + 4n^2 \text{ for all } n \geq 1$$

- Thus, we have shown that $n^3 + 4n^2 = \Omega(n^2)$ (by definition of Big- Ω, with $n_0 = 1$, and $c = 1$.)

Θ notation

- **Definition:** f (n) = Θ(g(n)) iff there are three positive constants c_1, c_2 and n_0 such that

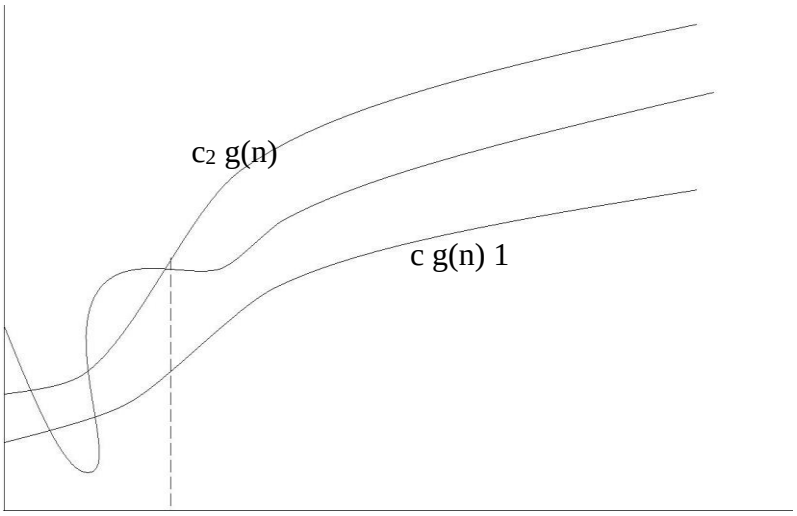
$$c_1|g(n)| \leq |f(n)| \leq c_2|g(n)| \text{ for all } n \geq n_0$$

- If f (n) is nonnegative, we can simplify the last condition to

$$0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n) \text{ for all } n \geq n_0$$

- We say that " f (n) is theta of g(n)."

- As n increases, f (n) grows at the same rate as g(n). In other words, g(n) is an *asymptotically tight bound* on f (n).



Example: $n^2 + 5n + 7 = \Theta(n^2)$

Proof:

- When $n \geq 1$,

$$n^2 + 5n + 7 \leq n^2 + 5n^2 + 7n^2 \leq 13n^2$$

- When $n \geq 0$,

$$n^2 \leq n^2 + 5n + 7$$

- Thus, when $n \geq 1$

$$1n^2 \leq n^2 + 5n + 7 \leq 13n^2$$

Thus, we have shown that $n^2 + 5n + 7 = \Theta(n^2)$ (by definition of Big- Θ, with $n_0 = 1$, $c_1 = 1$, and $c_2 = 13$.)

Arithmetic of Big-O, Ω, and Θ notations

- Transitivity:
  - $f(n) \in O(g(n))$ and
    $g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$
  - $f(n) \in \Theta(g(n))$ and
    $g(n) \in \Theta(h(n)) \Rightarrow f(n) \in \Theta(h(n))$
  - $f(n) \in \Omega(g(n))$ and
    $g(n) \in \Omega(h(n)) \Rightarrow f(n) \in \Omega(h(n))$

- Scaling: if f (n) ∈ O(g(n)) then for any k > 0, f (n) ∈ O(kg(n))
- Sums: if $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$ then $(f_1 + f_2)(n) \in O(\max(g_1(n), g_2(n)))$

Order of growth

Measuring the performance of an algorithm in relation with the input size 'n' is called order of growth.

n	logn	nlogn	n^2	2^n
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65536
32	5	160	1024	4294967296

It is clear that logarithmic function is the slowest growing function and Exponential function 2^n is the fastest function.

Properties of Big oh

Following are some important properties of big oh natations:

1. If there are two functions $f_1(n)$ and $f_2(n)$ such that $f_1(n)=O(g_1(n))$ and $f_2(n)=O(g_2(n))$ then $F_1(n)+f_2(n)=\max(O(g_1(n)) ,O(g_2(n)))$.
2. If there are two functions $f_1(n)$ and $f_2(n)$ such that $f_1(n)=O(g_1(n))$ and $f_2(n)=O(g_2(n))$ then $F_1(n) * f_2(n)=O(g_1(n)) *(g_2(n))$.
3. If there exists a function f_1 such that $f_1=f_2*c$ where c is the constant then,f_1 and f_2 are equivalent. That means $O(f_1+f_2)=O(f_1)=O(f_2)$.
4. If f(n)=O(g(n)) and g(n)=O(h(n)) then f(n)=O(h(n)).
5. In a polynomial the highest power term dominates other terms.
   For example if we obtain $3n^3+2n^2+10$ then its time complexity is $O(n^3)$.

6. Any constant value leads to O(1) time complexity. That is if f(n)=c then it Ɛ O(1) time complexity.

Basic Efficiency Classes

Different efficiency classes and each class possessing certain characteristic.

Name of efficiency class	Order of growth	Description	Example
Constant	1	As the input size grows then we get constant running time.	Scanning array elements
Logarithmic	logn	When we get logarithmic running time then it is sure that the algorithm does not consider all its input rather the problem is divided into smaller parts on each iteration	Perform binary search operation.
Linear	n	The running time of algorithm depends on the input size n	Performing sequential search operation.
Nlogn	nlogn	Some instance of input is considered for the list of size n.	Sorting the elements using merge sort or quick sort.
Quadratic	n^2	When the algorithm has two nested loops then this type of efficiency occurs.	Scanning matrix elements.
Cube	n^3	When the algorithm has three nested loops then this type of efficiency occurs.	Performing matrix multiplication.
Exponential	2^n	When the algorithm has very faster rate of growth then this type of efficiency occurs.	Generating all subsets of n elements.
Factorial	n!	When the algorithm is computing all the permutations then this type of efficiency occurs	Generating all permutations.

Examples:

Linear

```
for ( i=0 ; i<n ; i++ )
    m += i;
```

Time Complexity O(n)

Quadratic

```
for ( i=0 ; i<n ; i++ )
        for( j=0 ; j<n ; j++ )
            sum[i] += entry[i][j];
```

Time Complexity O(n^2)

Cubic

```
For(i=1;i<=n;i++)
  For(j=1;j<=n;j++)
     For(k=1;k<=n;k++)
       Printf("AAA");
```

Time Complexity is O(n^3)

Logarithmic

```
For(i=1;i<n;i=i*2)
   Printf("AAA")
```

Time Complexity O(log n)

Linear Logarithmic (Nlogn)

```
For(i=1;i<n;i=i*2)
   For(j=1;j<=n,j++)
   Printf("AAA")
```

Performance Measurement

Performance measurement is concerned with obtaining the space and time requirements of a particular algorithm. These quantities depend on the compiler and options used as well as on which the algorithm is run. To obtain the computing or run time of a program, we need a clocking procedure. Clock() that returns the current time in milliseconds. This function returns the number of clock ticks since the program started.

To determine the worst case time requirements of functions Insertion sort. First we need to

1.Decide on the values of n for which the times to be obtained.

2. Determine, for each of the above values of n, the data that exhibits the worst-case behavior.

Choosing Instant size

We decide on which values of n to use according to two factors: the amount of timing we want to perform and what we expect to do with the times. In insertion sort the worst case complexity is $O(n^2)$. It is Quadratic in n. We can obtain the time for all other values of n from this quadratic function. We need the times for more than three values of n for the following reasons:

1.Asymptotic analysis tells the behavior only for sufficiently large values for n. For smaller values of n, the run time may not follow the asymptotic curve. To determine the point beyond which the asymptotic curve is followed, we need to examine the times for several values of n.

2.Even in the region where the asymptotic behavior is exhibited, the times may not lie exactly on the predicted curve because of the effects of low-order terms that are discarded in the asymptotic analysis. For instance, a program with asymptotic complexity $O(n^2)$ can have an actual complexity that is c1n2+c2nlogn+c3n+c4- or any function of n in which the highest order term is c1n2 for some constant c1, c1>0.

Developing the test Data

For many programs, we can generate manually or by computer the data that exhibits the best and worst case time complexity. The average complexiy, is usually quite difficult to demonstrate. In insertion sort, the worst case data for any n is a decreasing sequence such as n, n-1, n-2,….1. The best case data is sorted sequence such as 0,1,…..n-1.

When we are unable to develop the data that exhibits the complexity we want to measure, we can pick the least(maximum, average) measured time from some randomly generated data as an estimate of the best(worst,average) behavior.

Setting up the Experiment

Having selected the instance sizes and developed the test data, then write the program that will measure the desired run times. For the insertion sort , this program to be written as.

```
Void main()
{
  Int a[1000], step=10;
  Clock_t start, finish;
  For(int n=0;n<=1000;n+=step)
  {
     For (int i=0;i<n;i++)
         a[i]=n-i;
         start=clock();
         insertionsort(a,n);
         finish=clock();
         cout<<n<<(finish-start)/CLK_TCK;
         if(n==100) step=100;
    }
```

The measure times are given below.

n	Time	n	Time
0	0	100	0
10	0	200	0.054945
20	0	300	0
30	0	400	0.054945
40	0	500	0.10989
50	0	600	0.109890
60	0	700	0.164835
70	0	800	0.164835
80	0	900	0.274725
90	0	1000	0.32967

In the above example, no time is needed to sort arrays with 100 or fewer numbers and that there is no difference in the times to sort 500 through 600 numbers.All measurements are accurate to with in one clock tick. If CLK-TCK=18.2 on our computer, the actual times may deviate from the measured times by up to one tick or 1/18.2˜0.055 seconds.