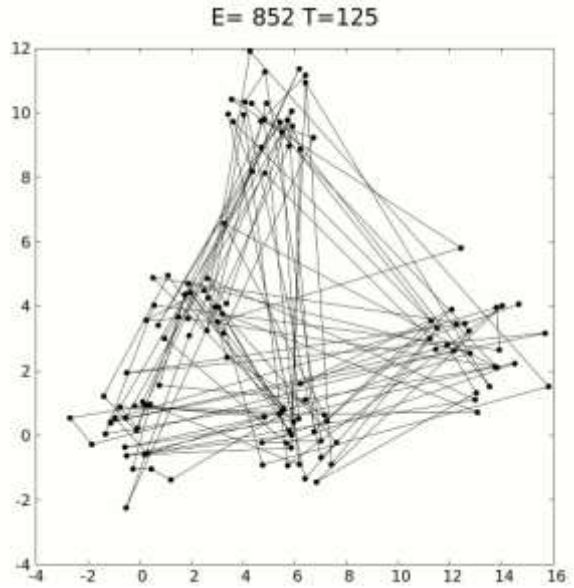
SCSA1403
Design and Analysis of Algorithms



B.E CSE 2020-2024

IV SEMESTER

Dr. A. Jesudoss
Associate Professor
Dept. of CSE
jesudossa.cse@sathyabama.ac.in

COURSE OBJECTIVES

➢ To analyze the performance of algorithms under various scenarios.

➢ To learn mathematical background for algorithm analysis & solving the recurrence equations.

➢ To learn various algorithm design techniques.

➢ To understand and apply the algorithms.

SCSA1403	DESIGN AND ANALYSIS OF ALGORITHMS	L	T	P	Credits	Total Marks
		3	*	0	3	100

COURSE OBJECTIVES

- To analyze the performance of algorithms under various scenarios.
- To learn mathematical background for algorithm analysis & solving the recurrence equations.
- To learn various algorithm design techniques.
- To understand and apply the algorithms.

UNIT 1 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.

UNIT 2 MATHEMATICAL FOUNDATIONS 9 Hrs.

Solving Recurrence Equations - Substitution Method - Recursion Tree Method - Master Method - Best Case - Worst Case Average Case Analysis - Sorting in Linear Time - Lower bounds for Sorting: - Counting Sort - Radix Sort - Bucket Sort.

UNIT 3 BRUTE FORCE AND DIVIDE-AND-CONQUER 9 Hrs.

Brute Force:- Travelling Salesman Problem - Knapsack Problem - Assignment Problem - Closest Pair and Convex Hul Problems - Divide and Conquer Approach:- Binary Search - Quick Sort - Merge Sort - Strassen's Matrix Multiplication.

UNIT 4 GREEDY APPROACH AND DYNAMIC PROGRAMMING **9 Hrs.**

Greedy Approach:- Optimal Merge Patterns- Huffman Code - Job Sequencing problem- -- Tree Vertex Splitting Dynamic Programming:– Dice Throw-- Optimal Binary Search Algorithms.

UNIT 5 BACKTRACKING AND BRANCH AND BOUND **9 Hrs.**

Backtracking:- 8 Queens - Hamiltonian Circuit Problem - Branch and Bound - Assignment Problem - Knapsack Problem:- Travelling Salesman Problem - NP Complete Problems - Clique Problem - Vertex Cover Problem .

Max. 45 Hrs.

COURSE OUTCOMES

On completion of the course, student will be able to

CO1 - Determine the suitable algorithmic design technique for a given problem.
CO2 - Identify the limitations of algorithms in problem solving.
CO3 - Analyze the efficiency of the algorithm based on time and space complexity.
CO4 - Implement asymptotic notations to analyze worst-case and average case running times of algorithms.
CO5 - Interpret the fundamental needs of algorithms in problem solving.
CO6 - Describe the various algorithmic techniques and its real time applications.

TEXT/REFERENCE BOOKS

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, "Introduction to Algorithms", 3rd Edition, PHI Learning Private Limited, 2012.
2. Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, "Data Structures and Algorithms David E. Goldberg, "Genetic Algorithm In Search Optimization And Machine Learning" Pearson Education India, 2013.
3. Anany Levitin, "Introduction to the Design and Analysis of Algorithms", 3rd Edition, Pearson Education, 2012.
4. Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, Fundamentals of Computer Algorithms, 2nd Edition, Universities Press, 2007.

END SEMESTER EXAMINATION QUESTION PAPER PATTERN

Max. Marks : 100 **Exam Duration : 3 Hrs.**

PART A : 10 Questions of 2 marks each-No choice **20 Marks**

PART B : 2 Questions from each unit with internal choice, each carrying 16 marks **80 Marks**

UNIT I - Introduction

1. Fundamentals of Algorithmic Problem Solving
2. Time Complexity
3. Space complexity with examples
4. Growth of Functions & Asymptotic Notations: Need, Types
5. Big Oh, Little Oh, Omega, Theta – Properties
6. Complexity Analysis Examples
7. Performance measurement
8. Instance Size
9. Test Data
10. Experimental setup.

Algorithm

- An algorithm can be defined as a **finite set of steps**, which has to be followed **while carrying out a particular problem**.
- It is a **process of executing actions step by step**.
- An algorithm is a **distinct computational procedure** that takes input as a set of values and results in the output as a set of values by solving the problem.

Analysis of an algorithm:

- Determination of the amount of the resources(time and space) necessary to execute them.

Characteristics of Algorithms

- **Input:** It should externally supply zero or more quantities.
- **Output:** It results in at least one quantity.
- **Definiteness:** Each instruction should be clear and ambiguous.
- **Finiteness:** An algorithm should terminate after executing a finite number of steps.
- **Effectiveness:** Every instruction should be fundamental to be carried out, in principle, by a person using only pen and paper.
- **Feasible:** It must be feasible enough to produce each instruction.

Characteristics of Algorithms

- **Flexibility:** It must be flexible enough to carry out desired changes with no efforts.
- **Efficient:** The term efficiency is measured in terms of time and space required by an algorithm to implement. Thus, an algorithm must ensure that it takes little time and less memory space meeting the acceptable limit of development time.
- **Independent:** An algorithm must be language independent, which means that it should mainly focus on the input and the procedure required to derive the output instead of depending upon the language.

Advantages of Algorithms

- **Effective Communication:** Since it is written in a natural language like English, it becomes easy to understand the step-by-step delineation of a solution to any particular problem.
- **Easy Debugging:** A well-designed algorithm facilitates easy debugging to detect the logical errors that occurred inside the program.
- **Easy and Efficient Coding:** An algorithm is nothing but a blueprint of a program that helps develop a program.
- **Independent of Programming Language:** Since it is a language-independent, it can be easily coded by incorporating any high-level language.

Disadvantages of Algorithms

- Developing algorithms for complex problems would be time-consuming and difficult to understand.
- It is a challenging task to understand complex logic through algorithms.

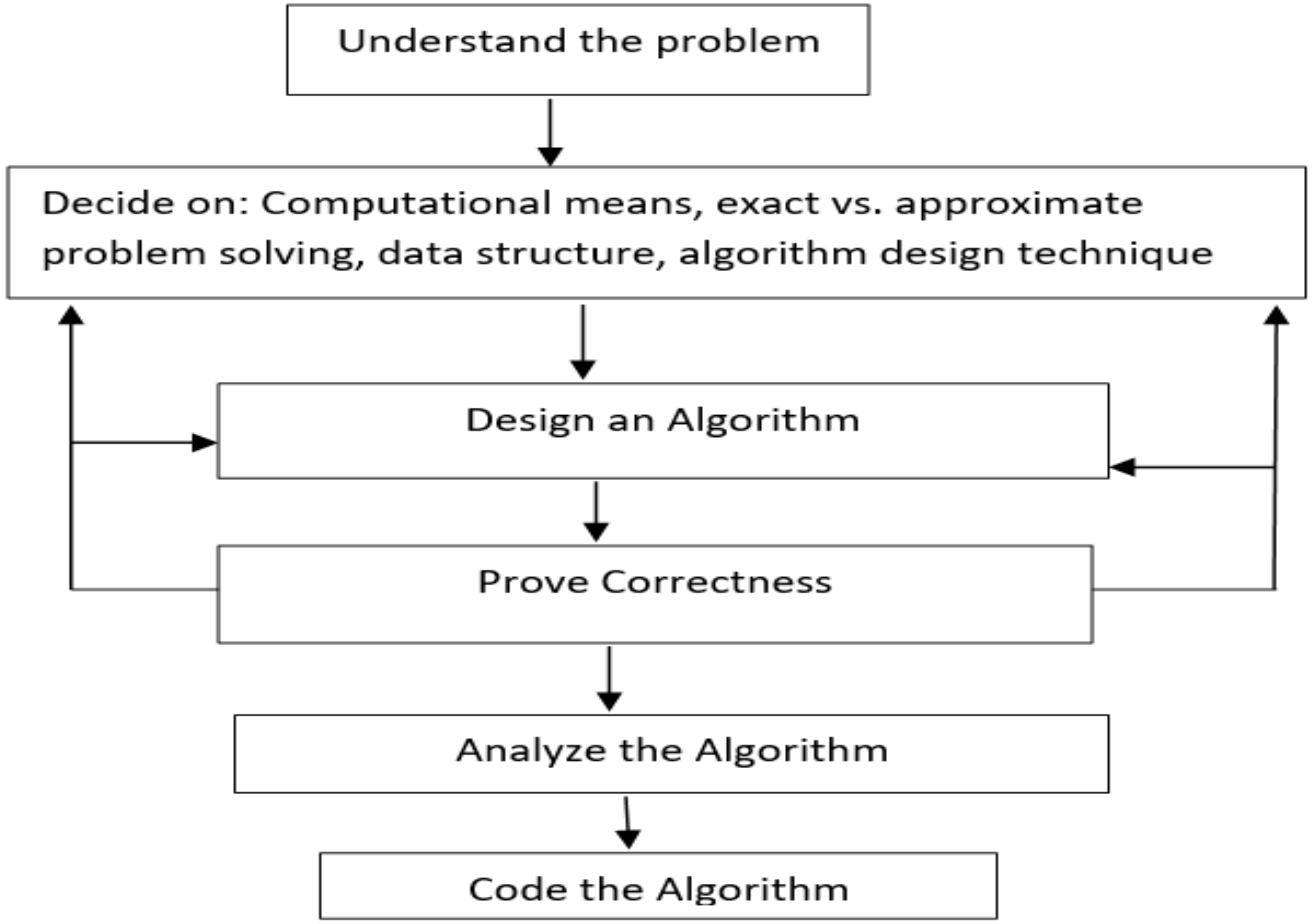
Problem:

Suppose there are 60 students in the class. How will you calculate the number of absentees in the class?

Algorithm:

1.Initialize a variable called as **Count** to zero, **absent** to zero, **total** to 60
2.FOR EACH Student PRESENT DO the following:
Increase the **Count** by One
3.Then Subtract **Count** from **total** and store the result in **absent**
4.Display the number of absent students

Steps in Design and Analysis of Algorithm:



Steps in Design and Analysis of 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.

Steps in Design and Analysis of Algorithm:

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.

Steps in Design and Analysis of Algorithm:

Algorithm design techniques: some of the main algorithm design techniques:

✓**Brute-force or exhaustive search** – generate and test – every possible combination

✓**Divide and Conquer** – Recursively breaks down into a problem into 2 or more

✓**Greedy Algorithms** - an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.

✓**Dynamic Programming** - Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later.

Steps in Design and Analysis of Algorithm:

✓**Branch and Bound Algorithm** – used for problem solving such as optimization problems, minimization problem

✓**Randomized Algorithm** - An algorithm that uses random numbers to decide what to do next anywhere in its logic

✓**Backtracking -** An algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time. e.g. **SudoKo** solving Problem, we try filling digits one by one. Whenever we find that current digit cannot lead to a solution, we remove it (backtrack) and try next digit. This is better than naive approach (generating all possible combinations of digits and then trying every combination one by one) as it drops a set of permutations whenever it backtracks.

Steps in Design and Analysis of Algorithm:

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 is 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

Steps in Design and Analysis of Algorithm:

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
- 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.

Important Problem Types:

1. Sorting
2. Searching
3. String Processing
4. Graph Problems
5. Numerical Problems
6. Combinatorial Problems
7. Geometric Problems

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.**

Performance of a program:

1) Algorithm Analysis

An algorithm analysis is a technique that's used to measure the performance of the algorithms.

Speed is one of the key parameters in determining the potential of an algorithm.

There are some other factors, like user-friendliness, security, maintainability, and usage space, that determine the quality of an algorithm.

Space and time complexity are metrics used to measure parameters.

Performance of a program:

2) Experimental Analysis of Algorithms

Every algorithm gives an output based on some parameters, like the number of loops, sample input size, and various others. In an experimental analysis, these data points are plotted on a graph to understand the behavior of the algorithm. We consider the worst-case running times.

The graphs show the running time of an algorithm with increasing input size for worst, average, and best-case running time in the form of a histogram and a plotted graph. We chose the x-axis as the input size because the running time depends on the input size. As an experimental analysis depends on the output results, an algorithm cannot be measured unless an equivalent program is implemented.

Limitations of Experimental Studies

•1. Implementing algorithms can be a tedious process.
•2. Hardware and software environments must be the same to compare algorithms (which is practically impossible).
•3. Input samples may not cover all the possible inputs and scenarios.

SPACE COMPLEXITY

- Memory space occupied by the program.
- Space complexity is the sum of the following components:
    1. Instruction space
    2. Data space
    3. Environment stack space

SPACE COMPLEXITY

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

SPACE COMPLEXITY

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

Space complexity is the sum of the following components:

(i) Instruction space:

-Source program

-Compiled version

-Executable version

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.

SPACE COMPLEXITY

Space complexity is the sum of the following components:

(ii) Data space:

-constants

-Variables

-Arrays

-Structures

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.

SPACE COMPLEXITY

Space complexity is the sum of the following components:

(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.

SPACE COMPLEXITY

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.

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

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.

Whenever a solution to a problem is written some memory is required to complete. For any algorithm memory may be used for the following:

1. Variables (This include the constant values, temporary values)
2. Program Instruction
3. Execution

Space complexity is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.

Sometime Auxiliary Space is confused with Space Complexity. But Auxiliary Space is the extra space or the temporary space used by the algorithm during it's execution.

Space Complexity = Auxiliary Space + Input space

SPACE COMPLEXITY

Consider the following piece of code...

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

If int is 4 bytes,

Then 4 bytes for a and 4 bytes for return type. Totally 8 bytes

Auxilliary space

SPACE COMPLEXITY

- Return value space - If any algorithm requires a fixed amount of space for all input values then that space complexity is said to be Constant Space Complexity
- Eg 2:

```
{
    int z = a + b + c;
    return(z);    // auxilliary space
// irrespective of no. of variables in program – 4 bytes for return types
}
```

a,b,c, z – 16 bytes

Return =- 4 bytes = 20 bytes

- 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*****.**

SPACE COMPLEXITY

- Eg.3

```
#include<stdio.h>
int main()
{
  int a = 5, b = 5, c;
  c = a + b;
  printf("%d", c);
}
```

A, b, c – 12 bytes

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;
}
```

SPACE COMPLEXITY

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*****.**

SPACE COMPLEXITY

```
float abc(float a, float b, float c)
{
    return(a+b+c+(a+b-c)/(a+b)+4.0)
}
```

- 3 float variables declared. So 12 bytes
- Retuning float variables occupies 4 bytes.
- Total 16 bytes.

```
Int add(int a[],intb[],int c[],n)
{
   for(i=0;i<n;i++)


        c[i]=a[i]+b[i]
  return(c[i])
}
```

- A[] takes 2n bytes.
- B[] takes 2n bytes.
- C[] takes 2n bytes.
- i takes 2 bytes.
- n takes 2 bytes.
- Returning array occupies 2n bytes.
- Total 8n+4 bytes
- O(n) (ignore constant)
- (4n2+n+8 = o(2)
- (also for 1 to n – loop goes n times
- Hence O(n)

SPACE COMPLEXITY

```
Alg sum(n)
{
if (n=0) then return 0;
else
  return n+sum(n-1)
}
```

Each call requires atleast 4 bytes. (assuming int 2 bytes)
Depth of recursion n+1.
Recursion state space is 4(n+1)

Sum (3) : - each iteration 4 bytes
n =2 + return 2 = 4

if n= 3 false
return n+ sum(2)

Sum(2):
if n = 2 false
return n+sum(1)

Sum(1):
if n = 1 false
return n + sum (0)

Sum(0):
if n = 0 true

Last iteration it check for false condition.
Hence, n loop will have n+1 iterations which is the depth of recursion.
Hence Recursion State Space is 4(n+1)

Does it mean for all recursions always – the depth of Recursion will be n+1 ?

No. It is based on condition.

If our condition in previous program is (N==1), then the depth of recursion is N.

Sum (3) : - each iteration 4 bytes
n =2 + return 2 = 4

if n= 3 false
return n+ sum(2)

Sum(2):
if n = 2 false
return n+sum(1)

Sum(1):
if n = 1 true
return

N item – we iterate from 1 to n so

Depth of recursion is n

In previous program, it was 0 to n

Hence depth recursion was n+1

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.

TIME COMPLEXITY- Example

- Imagine a classroom of 100 students in which you gave your pen to one person. Now, you want that pen. Here are some ways to find the pen and what the O order is.
- **O(n):** Going and asking each student individually is O(N).
- You go and ask the first person of the class, if he has the pen. Also, you ask this person about other 99 people in the classroom if they have that pen and so on
  $O(n^2)$:two loops I & j
  O(log n): Now I divide the class into two groups, then ask: “Is it on the left side, or the right side of the classroom?” Then I take that group and divide it into two and ask again, and so on. Repeat the process till you are left with one student who has your pen. This is what you mean by O(log n).

TIME COMPLEXITY- Example

- I'd use the O(n) if one student had the pen and only they knew it.
- I'd use the O(log n) search if all the students knew, but would only tell me if I guessed the right side.

TIME COMPLEXITY

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.

TIME COMPLEXITY

- Estimated by counting the number of elementary functions performed by the algorithm.
- Time complexity depends on the following factors:

1. Compiler Used
2. Target computer

TIME COMPLEXITY

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

TIME COMPLEXITY

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.

TIME COMPLEXITY

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.

TIME COMPLEXITY

Time Complexity of algorithm/code is **not** equal to the actual time required to execute a particular code but the number of times a statement executes. We can prove this by using time command.

We can prove this by using time command.

For example, Write code in C/C++ or any other language to find maximum between N numbers, where N varies from 10, 100, 1000, 10000.

And compile that code on Linux based operating system (Fedora or Ubuntu) with below command:

gcc program.c – o program

run it with time ./program

TIME COMPLEXITY

- You will get surprising results i.e. for N = 10 you may get 0.5ms time and for N = 10, 000 you may get 0.2 ms time.
- Also, you will get different timings on the different machine. So, we can say that actual time requires to execute code is machine dependent (whether you are using pentium1 or pentiun5) and also it considers network load if your machine is in LAN/WAN.
- Even you will not get the same timings on the same machine for the same code, the reason behind that the current network load.
  Now, the question arises if time complexity is not the actual time require executing the code then what is it?

TIME COMPLEXITY

- **The answer is :** Instead of measuring actual time required in executing each statement in the code, we consider how many times each statement execute. For example: int main()

```
{
    printf("Hello World");
}
```

Output

Hello World

TIME COMPLEXITY

In above code “Hello World!!!” print only once on a screen.

So, time complexity is constant: O(1) i.e. every time constant amount of time require to execute code, no matter which operating system or which machine configurations you are using.

TIME COMPLEXITY

```
void main()
{
    int i, n = 8;
    for (i = 1; i <= n; i++) {
        printf("Hello Word !!!\n");
    }
}
```

TIME COMPLEXITY

Output

Hello Word !!!

Hello Word !!!

Hello Word !!!

Hello Word !!!

Hello Word !!!

Hello Word !!!

Hello Word !!!

Hello Word !!!

Hello Word !!!

In above code “Hello World!!!” will print N times. So, time complexity of above code is O(N).

TIME COMPLEXITY

```
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        statement;
    }
}
```

Time complexity is quadratic.

$0(n^2)$.

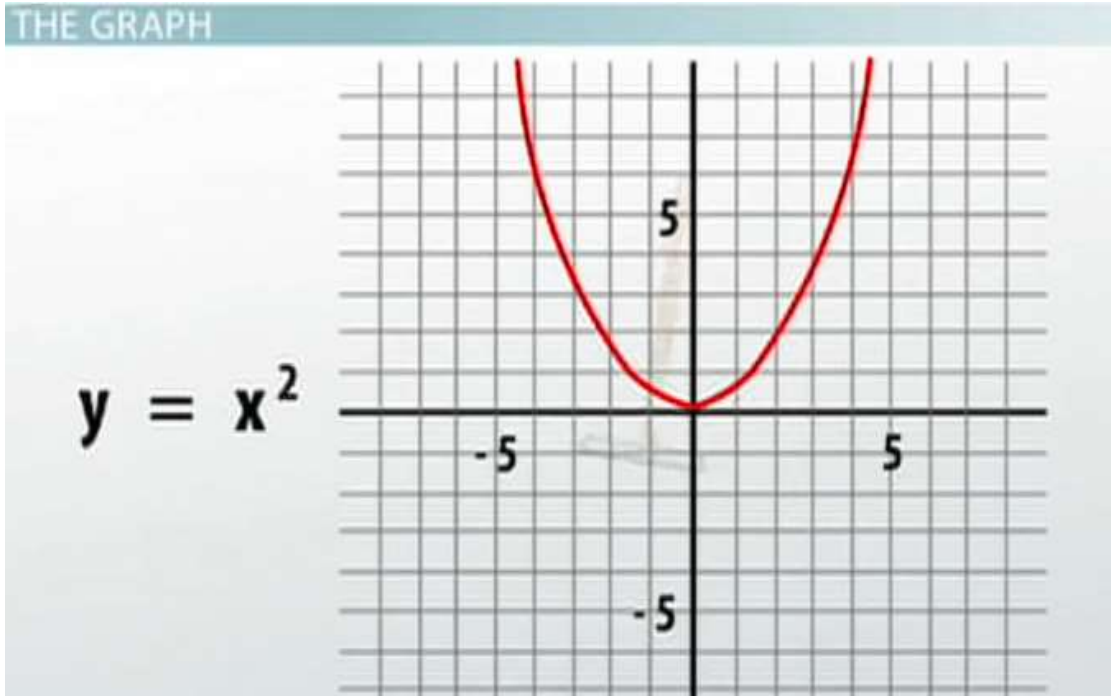
- i loop executed for n+1 times
- j loop executed for n(n+1) times
- Time=n+1+n2+n➔$O(n^2)$

What is quadratic?
any equation containing one term in which the unknown is squared and no term in which it is raised to a higher powersolve for x in the *quadratic equation* $x^2 + 4x + 4 = 0$

Quadratic means Equation of degree 2

The highest exponent of this function is 2

Y= ax^2 + bx + c
Y= x^2 + 3x + 1
Y = X^2
Y = 2x^2 + 4x – 9
Y = x^2 – 9
A = pi * r^ 2



$2x^2 + 4x + 2 = 0$

Quadratic Equation

$ax^2 + bx + c = 0$

Where :- $a \neq 0$, a, b & c → Real Numbers

a is the coefficient of X^2
b is the coefficient of X^2
c is the constant

$x^2 + 5x + 6 = 0$

$a = 1$
$b = 5$
$c = 6$

$-3x^2 + 12 = 0$

$a = -3$
$b = 0$
$c = 12$

TIME COMPLEXITY

```
for(i=1;i<n;i=i*2)
{
Statement;
}
```

Time complexity is O(log n).

```
for(i=0;i<n;i++)
{
   for(j=1;j<n;j=j*3)
}
```

Time complexity is $O(n \log_3 n)$

- Loop control variable is incremented geometrically.
- i takes the value 1 2 4 8 16 ……. n
- 2^0 2^1 2^2 2^3 2^4 2^k
- **$2^k = n$**
- **$K = \log_2 n$**

Remember:

- **$2^3 = 8$**
- **$3 = \text{Log}_2\ 8$**
- **3 is logarithm of 8 to the base 2**

- **$10^2 = 100$**
- **$2 = \log_{10} 100$**

TIME COMPLEXITY

Basic Time Complexities:

- Constant time
- Logarithmic Time
- Linear time
- Quadratic time

Constant time **O(1)**

Constant Time Complexity describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.

It doesn't matter whether the array has 5 or 500 indices, looking up a specific location of the array would take the same amount of time, therefore the function has a constant time look-up.

Logarithmic Time - **O(log(n))**

The most common attributes of logarithmic running-time function are that:

the choice of the next element on which to perform some action is one of several possibilities, and only one will need to be chosen.
Or
the elements on which the action is performed are digits of n

This is why, for example, looking up people in a phone book is O(log n). You don't need to check every person in the phone book to find the right one; instead, you can simply divide-and-conquer by looking based on where their name is alphabetically, and in every section you only need to explore a subset of each section before you eventually find someone's phone number.

TIME COMPLEXITY

Linear Time Complexity **O(N)**

describes an algorithm or program who's complexity will grow in direct proportion to the size of the input data.

In other words, the larger the input, the greater the amount of time it takes to perform the function.

Quadratic Time Complexity **O(N²)**

Quadratic Time Complexity represents an algorithm whose performance is directly proportional to the squared size of the input data set (think of Linear, but squared). Within our programs, this time complexity will occur whenever we nest over multiple iterations within the data sets.

TIME COMPLEXITY

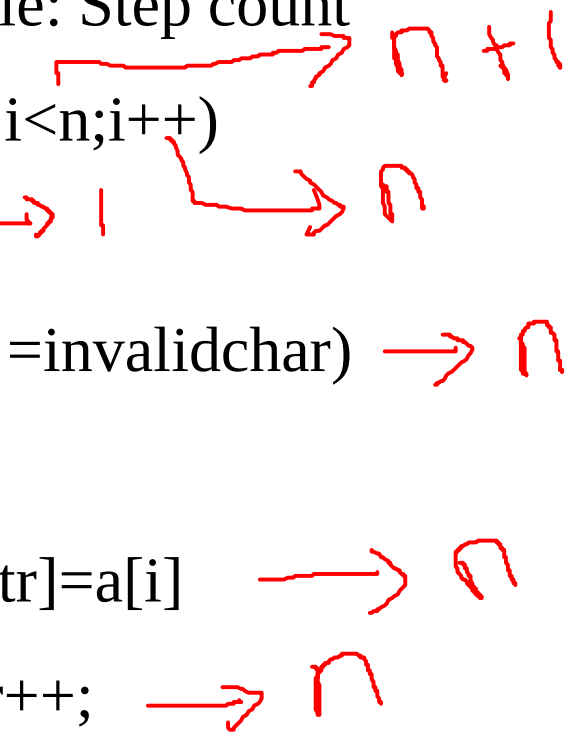
Basic Time Complexities:

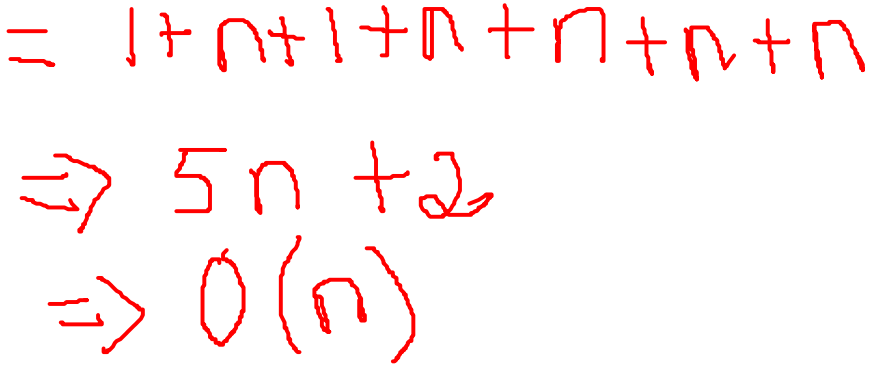
- Constant time
- Logarithmic Time
- Linear time
- Quadratic time

TIME COMPLEXITY

Example: Step count

```
for(i=0;i<n;i++)
{
 if (a[i]!=invalidchar)
{
      a[ptr]=a[i]
       ptr++;
}}
```





- 5n+2=>O(n)

TIME COMPLEXITY

Example: Step count

```
Sum(a,n)
{
total=0;                          ⟶ 1
for i=0 to n-1                    ⟶ n+1
      total= total+a[i]           ⟶ n
return total                      ⟶ 1
}
```

⇒ 1+n+1+n+1

⇒ 2n+3

⇒ O(n)

- 2n+3=>O(n)

TIME COMPLEXITY

Example: Step count

```
i=1;                    ---> 1
sum=0;                  ---> 1
While(i<=n)             ---> n+1
{
  j=1;                  ---> n
  while(j<=n)           ---> n(n+1)
  {
      sum=sum+1;        ---> n(n)
      j=j+1             ---> n(n)
  }
 i=i+1                  ---> n
}
```

- $3n^2+3n+3 => O(n^2)$

$\Rightarrow 3n^2+3n+3$

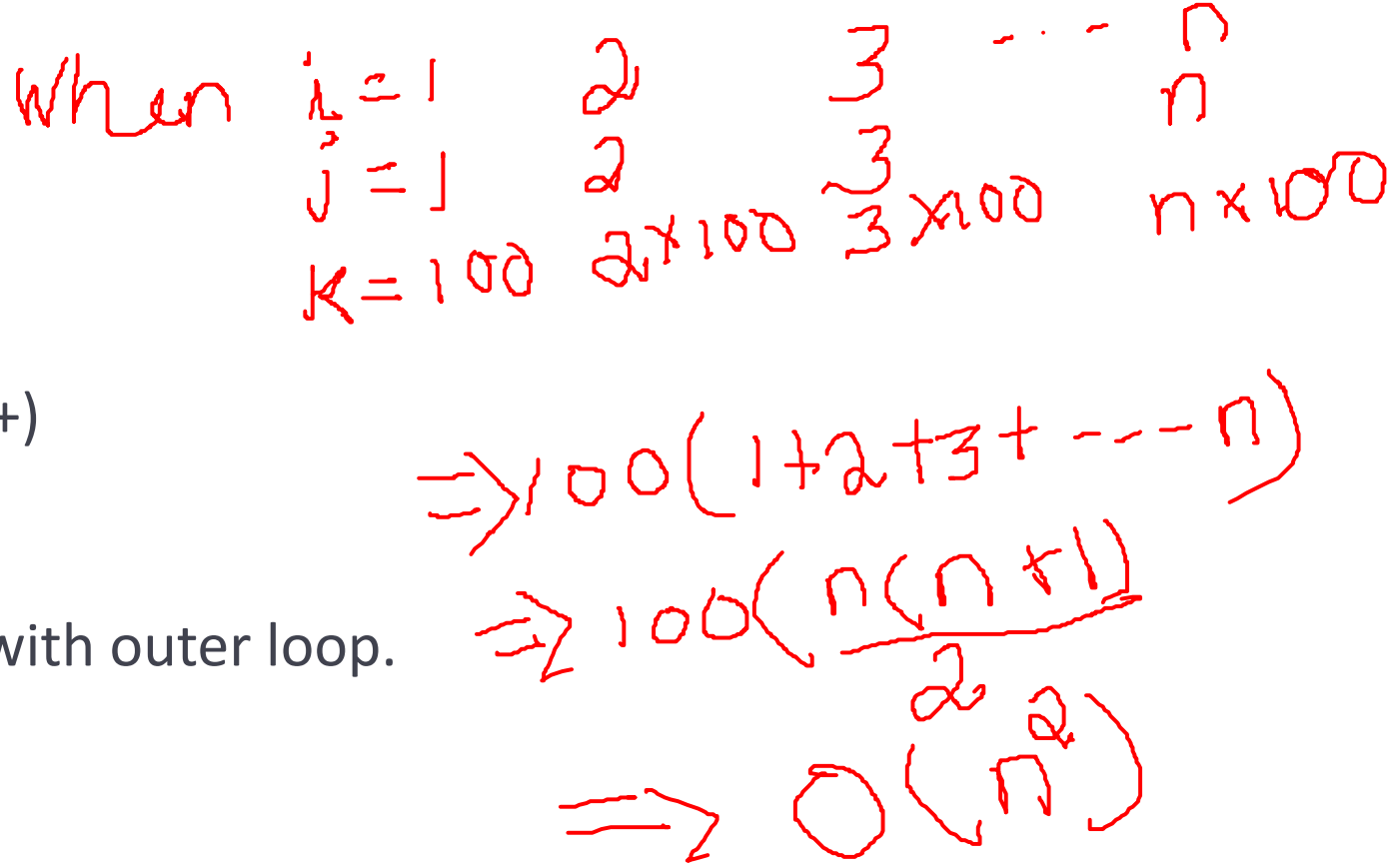
$\Rightarrow O(n^2)$

TIME COMPLEXITY

```
A()
{
    int i,j,k,n;
    for(i=1;i<=n;i++)
        for(j=1;j<=i;j++)
            for(k=1;k<=100;k++)
                printf("CSE")
}
```

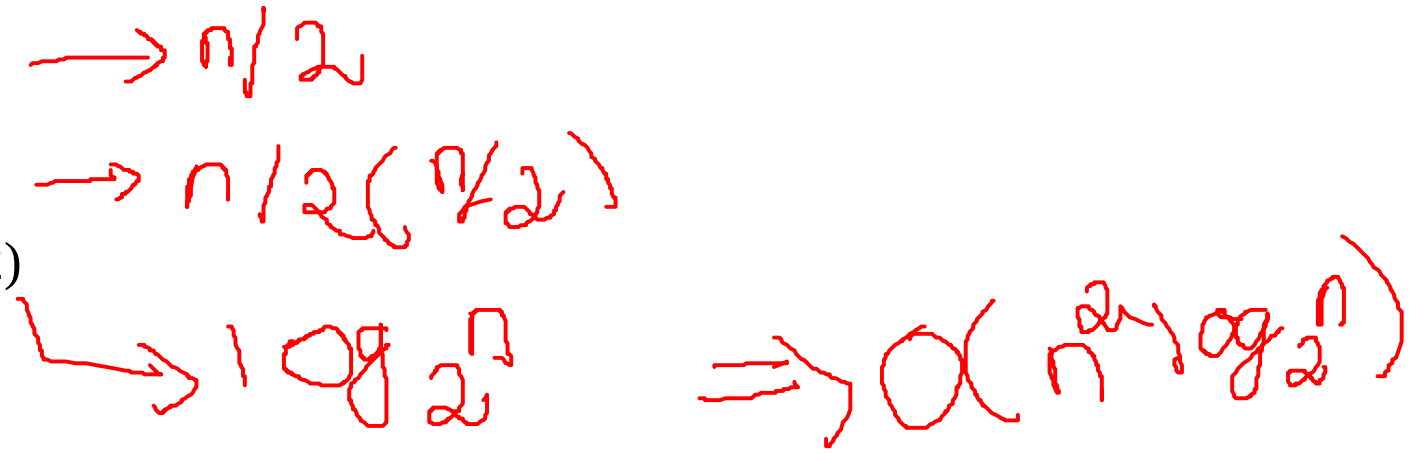
- Inner loop is dependent with outer loop.

Time complexity is $O(n^2)$



TIME COMPLEXITY

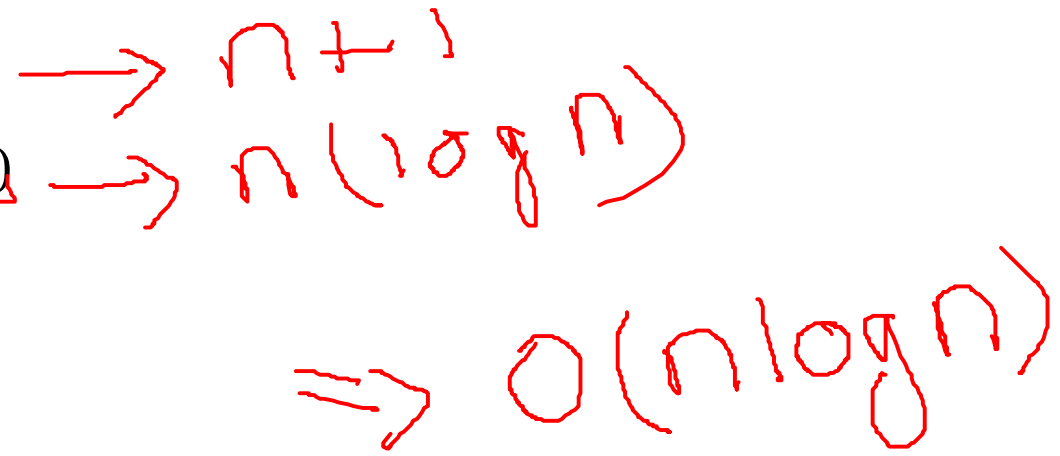
```
A()
{
int i,j,k;
for(i=n/2;i<=n;i++)
for(j=1;j<=n/2;j++)
for(k=1;k<=n; k=k*2)
   printf("AAA")
}
```



- No dependency
- $O(n^2 \log_2 n)$

TIME COMPLEXITY

```
A()
{
for(i=1;i<=n;i++)
for(j=1;j<=n; j=j+i)
printf("AAA")
}
```



- Dependency
- O(nlogn)

TIME COMPLEXITY

```
A()
{
for(i=1;i<=n;i++)
    for(j=1;j<=i^2;j=j++)
        for(k=1;k<=n/2;k++)
            printf("AAA")
}
```

- Dependency
- $O(n^4)$

TIME COMPLEXITY

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

- Time Complexity is $O(n^3)$
- Cubic

TIME COMPLEXITY- Example

- Imagine a classroom of 100 students in which you gave your pen to one person. Now, you want that pen. Here are some ways to find the pen and what the O order is.
- **O(n):** Going and asking each student individually is O(N). **O(1):** You go and ask the first person of the class,
- **O(log n):** Now I divide the class into two groups, then ask: “Is it on the left side, or the right side of the classroom?” Then I take that group and divide it into two and ask again, and so on. Repeat the process till you are left with one student who has your pen. This is what you mean by O(log n).

TIME COMPLEXITY

	COST	N0. OF TIME	TOTAL TIME
int sum(int a[],int n)	(no. Of operations		
{			
int sum=0;i;	1	1	1
for(i=0;i<n;i++)	3	1+(n+1)+n	2n+2
sum=sum+a[i]	2	n	n
return(sum)	1	1	1
}			

- Time complexity is O(n)

TIME COMPLEXITY

ADDITIONAL INFORMATION :

For example:

Let us consider a model machine which has the following specifications:

-Single processor

-32 bit

-Sequential execution

-1 unit time for arithmetic and logical operations

-1 unit time for assignment and return statements

TIME COMPLEXITY

Pseudocode:
Sum(a,b){
return a+b //Takes 2 unit of time(constant) one for arithmetic operation and one for return.(as per above conventions) cost=2 no of times=1
}

Tsum= 2 = C =O(1)

2n is not cost 2 x n

It is because 2 bytes for each element in array with n elements

TIME COMPLEXITY

S/E – steps in execution

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];	2	n	2n
return s;	1	1	1
}	0	------	0
Total			(3n + 3)

TIME COMPLEXITY

Pseudocode:

```
list_Sum(A,n){//A->array and n->number of elements in the array
total =0          // cost=1  no of times=1
for i=0 to n-1     // cost=2  no of times=n+1 (+1 for the end false condition)
sum = sum + A[i]   // cost=2  no of times=n
return sum         // cost=1  no of times=1
}
```

$$Tsum=1 + 2 * (n+1) + 2 * n + 1 = 4n + 4 = C1 * n + C2 = O(n)$$

TIME COMPLEXITY

How to Compare Algorithms?

To compare algorithms, let us define a few objective measures:

Execution times: Not a good measure as execution times are specific to a particular computer.
A number of statements executed: Not a good measure, since the number of statements varies with the programming language as well as the style of the individual programmer.
Ideal solution: Let us assume that we express the running time of a given algorithm as a function of the input size n (i.e., f(n)) and compare these different functions corresponding to running times. This kind of comparison is independent of machine time, programming style, etc.

Asymptotic Notation

Asymptotic Notations are the expressions that are used to represent the complexity of an algorithm.

- There are three types of analysis that we perform on a particular algorithm.
- Best Case: In which we analyse the performance of an algorithm for the input, for which the algorithm takes less time or space.
- Worst Case: In which we analyse the performance of an algorithm for the input, for which the algorithm takes long time or space.
- Average Case: In which we analyse the performance of an algorithm for the input, for which the algorithm takes time or space that lies between best and worst case.

Growth of Functions and Asymptotic 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.

Groth of Functions and Asymptotic Notation

Types of Data Structure Asymptotic Notation

1. Big-O Notation (O) – Big O notation specifically describes worst case scenario.

2. Omega Notation (Ω) – Omega(Ω) notation specifically describes best case scenario.

3. Theta Notation (θ) – This notation represents the average complexity of an algorithm.

Big-O Notation (O)

Big O notation specifically describes **worst case scenario or maximum amount of time an algorithm can possibly take to complete**. It is the formal way to express the **upper bound running time** complexity of an algorithm. Lets take few examples to understand how we represent the time and space complexity using Big O notation.

O(1)

Big O notation O(1) represents the complexity of an algorithm that always execute in same time or space regardless of the input data.

O(1) example

The following step will always execute in same time(or space) regardless of the size of input data.

Accessing array index(int num = arr[5])

Big-O Notation (O)

O(n)
Big O notation O(N) represents the complexity of an algorithm, whose performance will grow linearly (in direct proportion) to the size of the input data.

O(n) example
The execution time will depend on the size of array. When the size of the array increases, the execution time will also increase in the same proportion (linearly)

Traversing an array

Big-O Notation (O)

O(n^2)

Big O notation O(n^2) represents the complexity of an algorithm, whose performance is directly proportional to the square of the size of the input data.

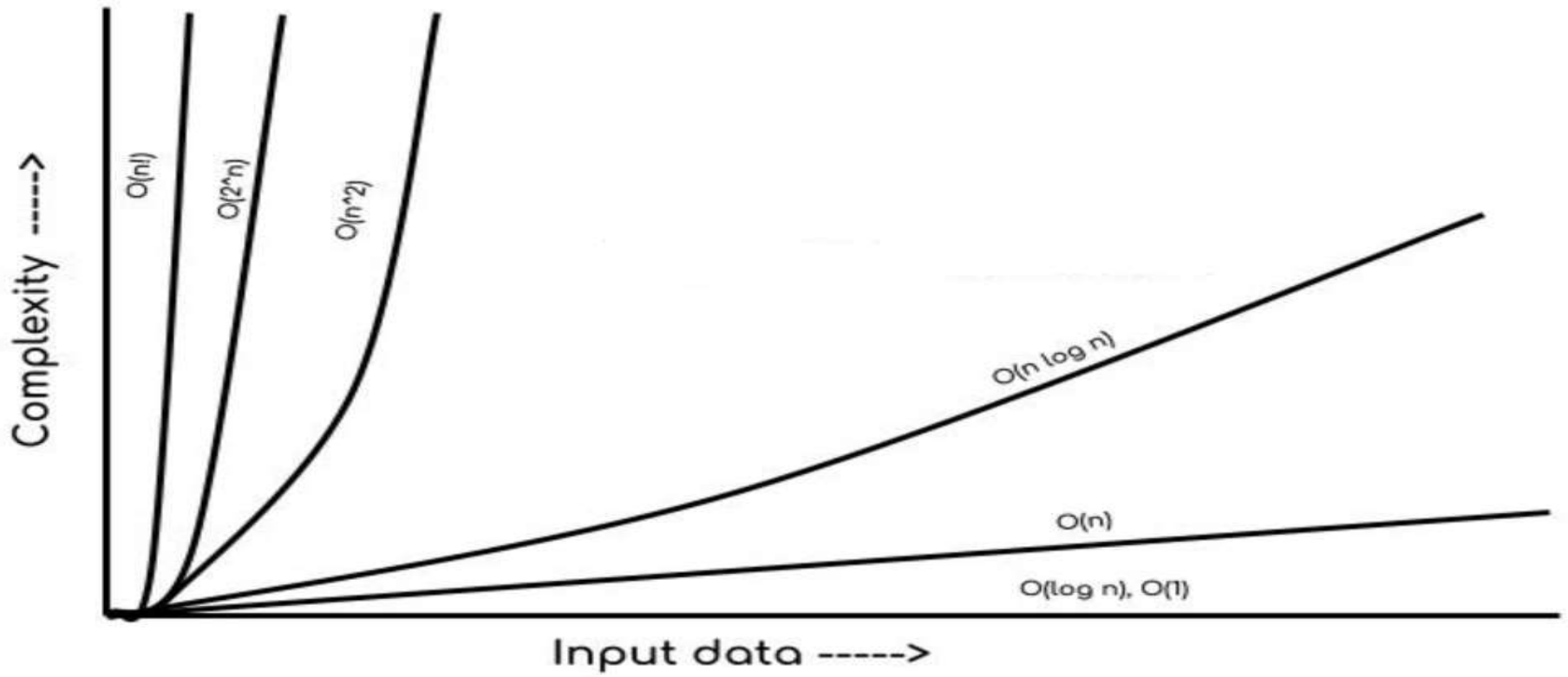
O(n^2) example

Traversing a 2D array

Similarly there are other Big O notations such as: logarithmic growth O(log n), log-linear growth O(n log n), exponential growth O(2^n) and factorial growth O(n!).

Big-O Notation (O)

```
O(1) < O(log n) < O (n) < O(n log n) < O(n^2) < O (n^3)< O(2^n) < O(n!)
```



Big-O Notation

- **Definition:** f (n) = O(g(n)) iff there are two positive constants c and n_0 such that

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

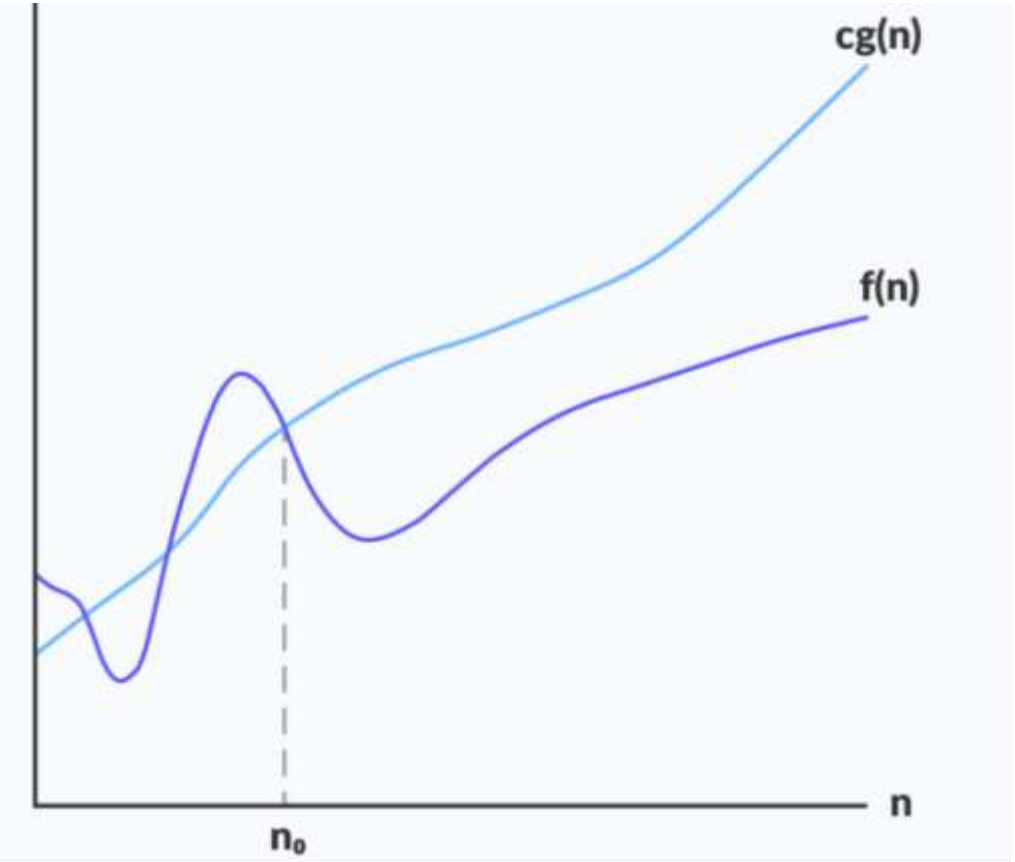
- If f (n) is nonnegative, we can simplify the last condition to

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

- 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).

G(n) represents Big O notation i.e. worst case scenario.
It will be greater than f(n) inspite of increase in N value



Big-O

The big Oh notation provides an upper bound for the function f(n).

> The function $f(n) = O(g(n))$ if and only if there exists positive constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

Examples:

1. $f(n) = 3n + 2$

   Let us take $g(n) = n$

   $c = 4$

   Let us check the above condition

   $3n + 1 \leq 4n$ for all $n \geq 2$ $\quad n_0 = 2$ **starts with 2**

   The condition is satisfied. Hence $f(n) = O(n)$.

n0 represents starting point on which condition f(n) <= c.g(n) gets satisfied.

EXAMPLES

Prove that f(n)=O(g(n)) where f(n)=3n+2 and g(n)=n

Solution:

f(n)<= O(g(n)) - to remove Big, constant is added

f(n)<=c.g(n), c>0, n_0>=1 n0 represents start value that satisfies given equation

3n+2<=c.n. Let us the value of constant as 4

if n = 1

3x1 + 2 <= 4x1

5 <= 4 = false

If n = 2

3x2 + 2 <= 4 x 2

8<=8 True. Therefore f(n)=Og(n) n0 is 2.

if n = 2

3x3 + 2 <= 4 x 3

11 <= 12 True whereas n ranges from 1 to n where n satisfes the condition

Big-O Notation (O)

2. $f(n) = 10n^2 + 4n + 2$

Let us take $g(n) = n^2$

$c = 11$

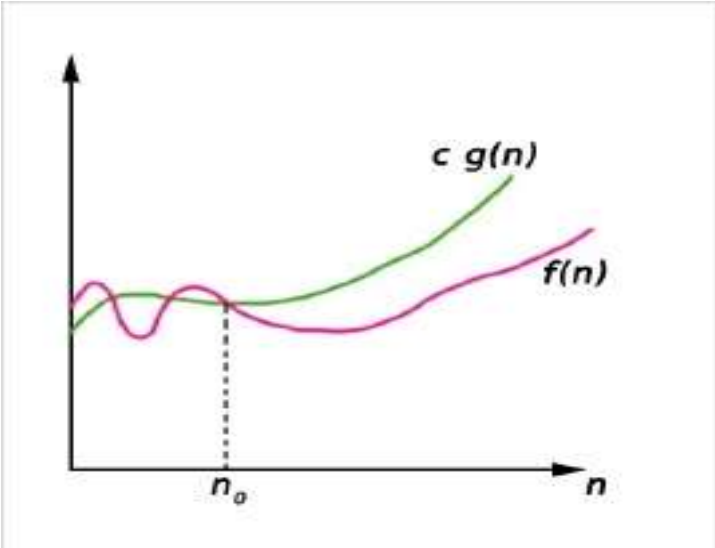
$n_0 = 6$

Let us check the above condition

$10n^2 + 4n + 2 \leq 11n$ for all $n \geq 5$

The condition is satisfied. Hence $f(n) = O(n^2)$.

Performance will grow linearly (in direct proportion) to the size of the input data.

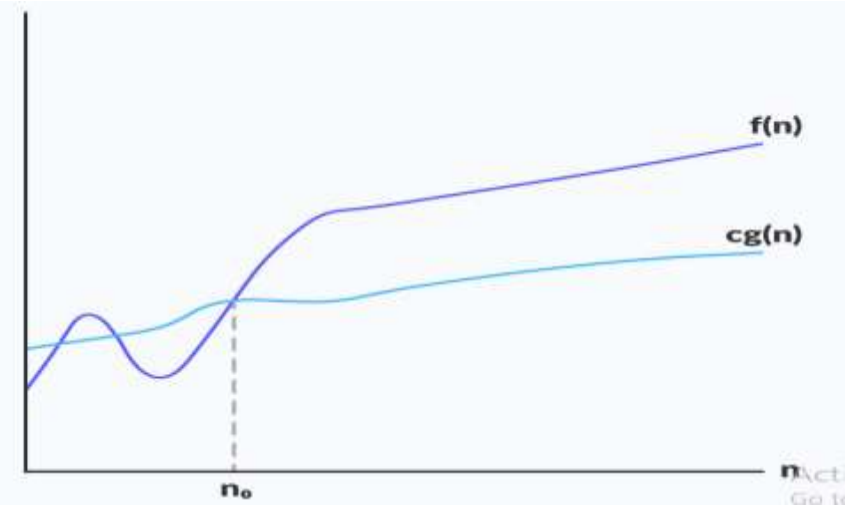


$$f(n) = O(g(n))$$

The function $f(n) = O(g(n))$ if and only if there exists positive constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

Omega Notation (Ω)

Omega notation specifically describes best case scenario or the best amount of time an algorithm can possibly take to complete. It is the formal way to express the lower bound running time complexity of an algorithm. So if we represent a complexity of an algorithm in Omega notation, it means that the **algorithm cannot be completed in less time than this**, it would at-least take the time represented by Omega notation or it can take more (when not in best case scenario).

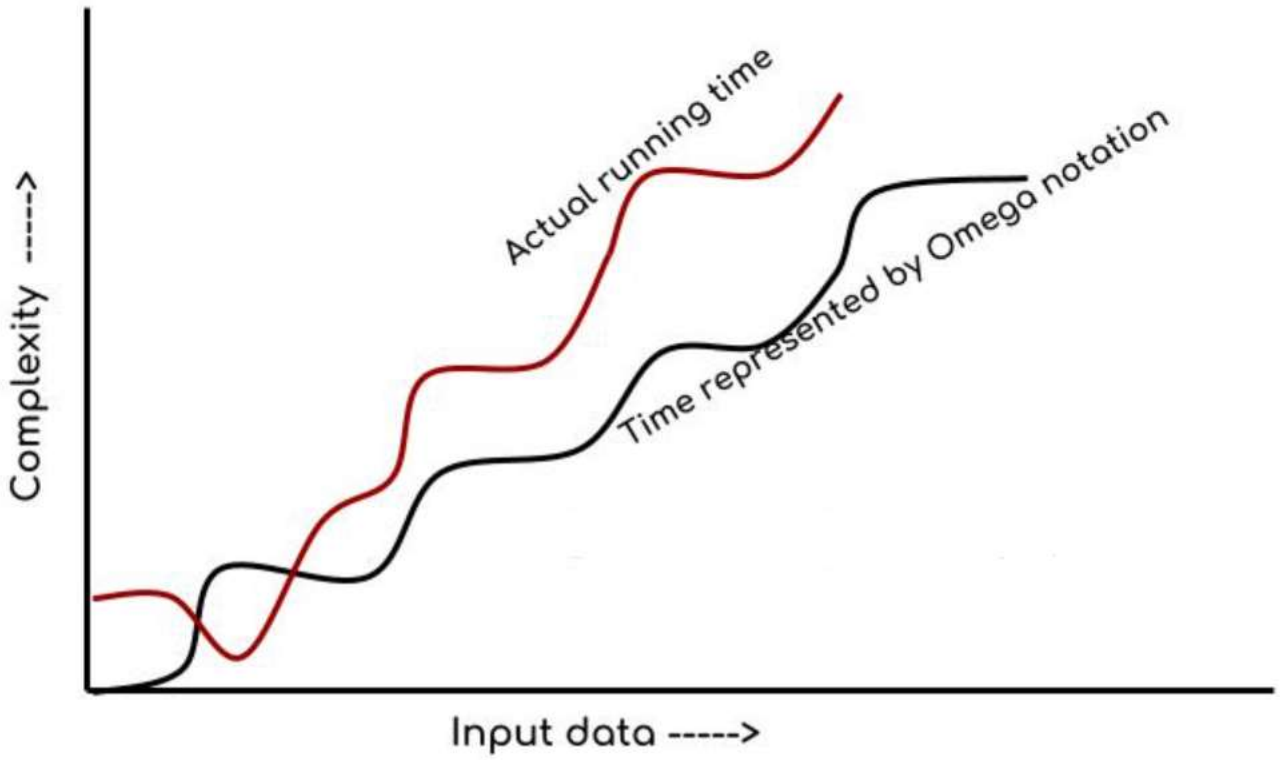


$$f(n) = \Omega(g(n))$$

Why it is f(n) >= c.g(n)?
Because at any cost the actual time taken f(n) will be equal to Best case or more than that.
G(n) represents best case.

The function $f(n) = \Omega(g(n))$ if and only if there exists positive constants c and n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$.

Omega Notation (Ω)



Omega Notation (Ω)

1. $f(n) = 10n^2 + 4n + 2$

Let us take $g(n) = n^2$

$c = 10$

$n_0 = 0$

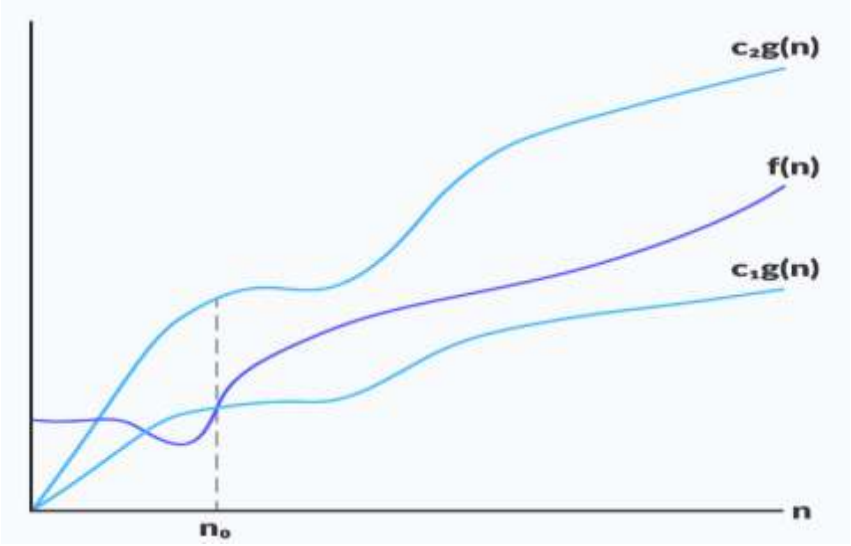
Let us check the above condition

$10n^2 + 4n + 2 \geq 10n$ for all $n \geq 0$

The condition is satisfied. Hence $f(n) = \Omega(n^2)$.

Theta Notation (θ)

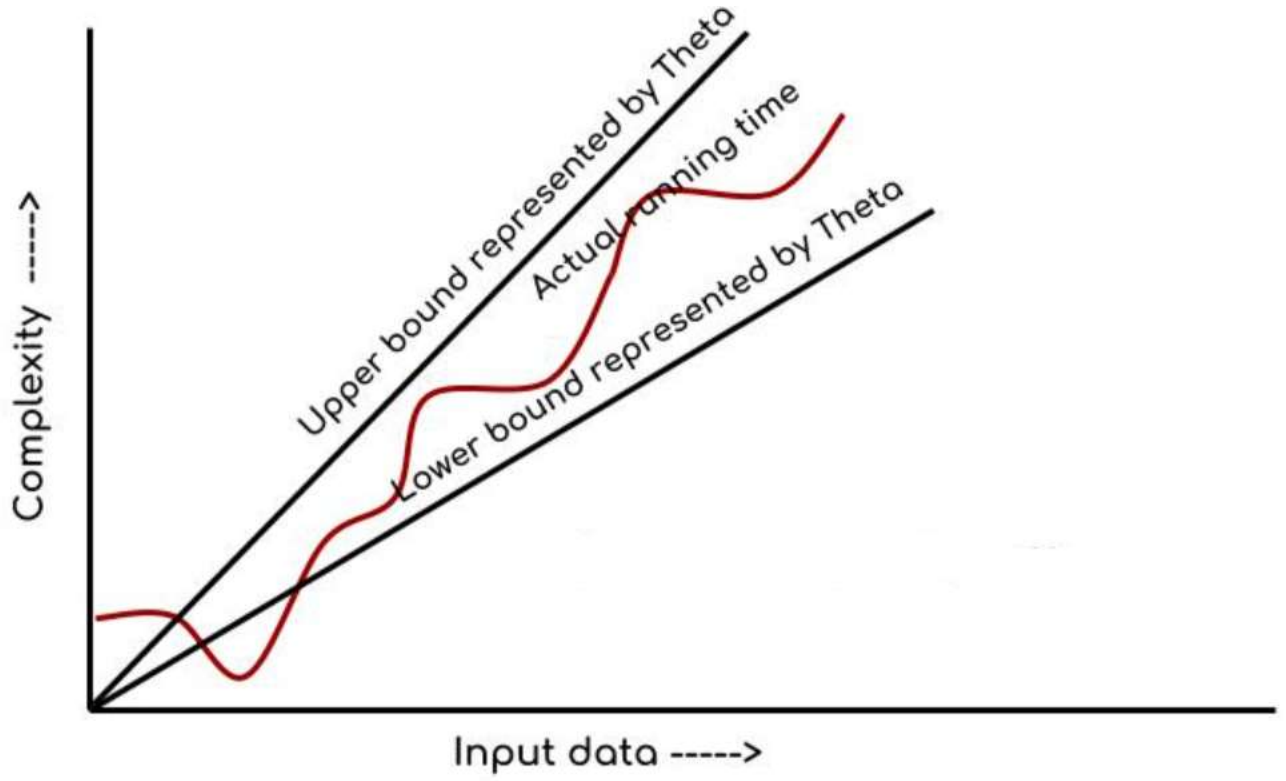
This notation describes both upper bound and lower bound of an algorithm so we can say that it defines exact asymptotic behaviour. In the real case scenario the algorithm not always run on best and worst cases, the average running time lies between best and worst and can be represented by the theta notation.



$$f(n) = \Theta(g(n))$$

$f(n) = \theta(g(n))$ if and only if there exists some positive constants c_1 and c_2 and n_0, such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.

Theta Notation (θ)



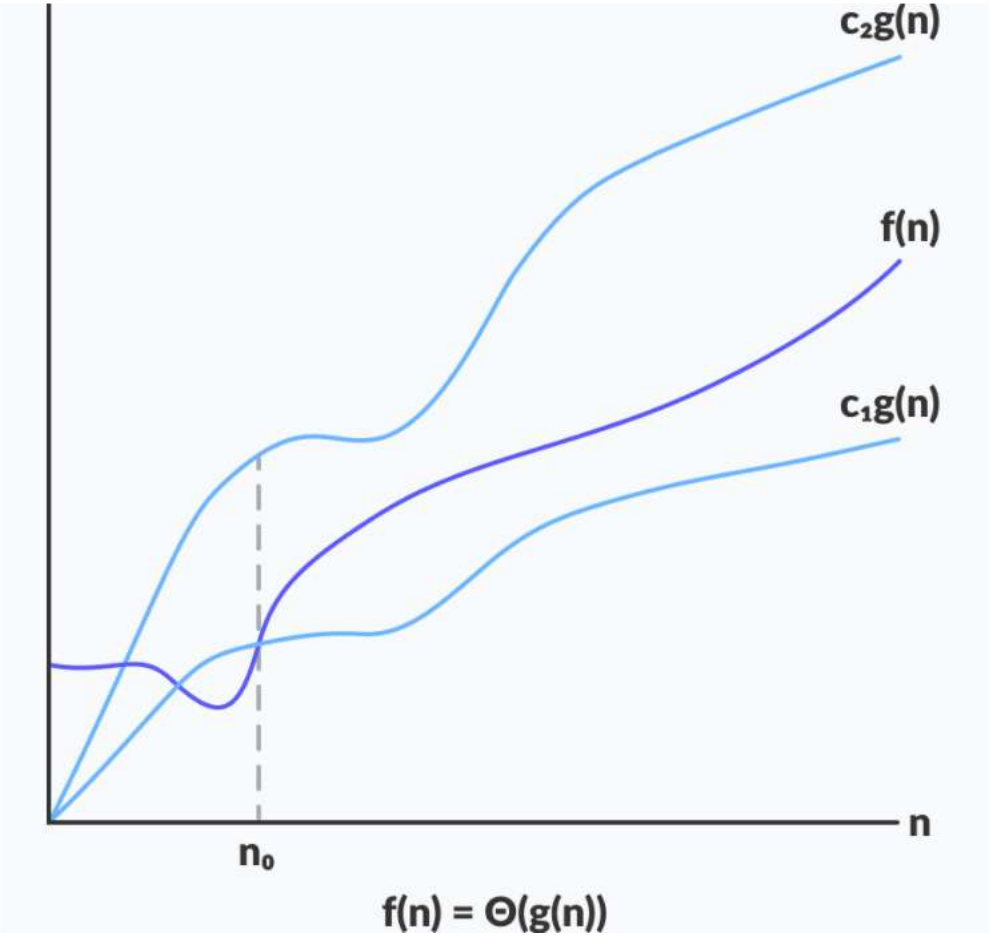
Θ notation

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

  $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$ for all $n \geq n_0$
- If $f(n)$ is nonnegative, we can simplify the last condition to

  $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ 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 $f(n)$.



$f(n) = \theta(g(n))$ if and only if there exists some positive constants c_1 and c_2 and n_0, such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.

BASIC TIME COMPLEXITIES

1	Constant
$\log n$	Logarithmic
n	Linear
$n \log n$	Log Linear
n^2	Quadratic
n^3	Cubic
2^n	Exponential

Properties of Big Oh Notation

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 $O(n^3)$.

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

Order of Growth

- Measuring the performance of an algorithm in relation with the input size ‘n’ is called order of growth.
- It is clear that logarithmic function is the slowest growing function and Exponential function 2^n is the fastest function.

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

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.

Basic Efficiency Classes:

Different efficiency classes and each class possessing certain characteristic.

Name of efficiency class	Order of growth	Description	Example
Quadratic	n2	When the algorithm has two nested loops then this type of efficiency occurs.	Scanning matrix elements.
Cube	n3	When the algorithm has three nested loops then this type of efficiency occurs.	Performing matrix multiplication.
Exponential	2n	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.

Performance Measurement:

Generally, the performance of an algorithm depends on the following elements...

- Whether that algorithm is providing the exact solution for the problem?
- Whether it is easy to understand?
- Whether it is easy to implement?
- How much space (memory) it requires to solve the problem?
- How much time it takes to solve the problem? Etc.,
- These quantities depends on the compiler and options used.
- To obtain the run time of a program, we need a clocking procedure. Clock() that returns the current time in milliseconds.

```
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

No time is needed to sort 100 or fewer elements.

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(n2)

Examples:

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(n3)

Logarithmic

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

Time Complexity O(log n)

Examples:

Linear Logarithmic (Nlogn)

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