Unit 1: Introduction to Data Structures & Algorithms

<u>Data Type</u>

Data type is a data storage format that can contain a specific type or range of values.

When computer programs store data in variables, each variable must be assigned a specific data type. Some common data types include integers, floating point numbers, characters, strings, and arrays.

<u>Data Structure</u>

- The data structure name indicates itself that organizing the data in memory. Data structure is a way of organizing all data items and establishing relationship among those data items.
- Data structures are the building blocks of a program.
- Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc.
- A static data structure is one whose capacity is fixed at creation. For example, array.
- A dynamic data structure is one whose capacity is variable, so it can
  for example, linked list, binary tree etc.
- Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artificial intelligence, Graphics and many more.
- Data structure mainly specifies the following four things:
  - Organization of data.
  - Accessing methods
  - Degree of associativity
  - Processing alternatives for information
- algorithm and its associated data structures form a program.
- Algorithm + Data structure = Program

Classification of Data Structure

There are two types of data structures:

- Primitive data structure (Build in Data Structure)
- Non-primitive data structure (User Defined Data Structure)

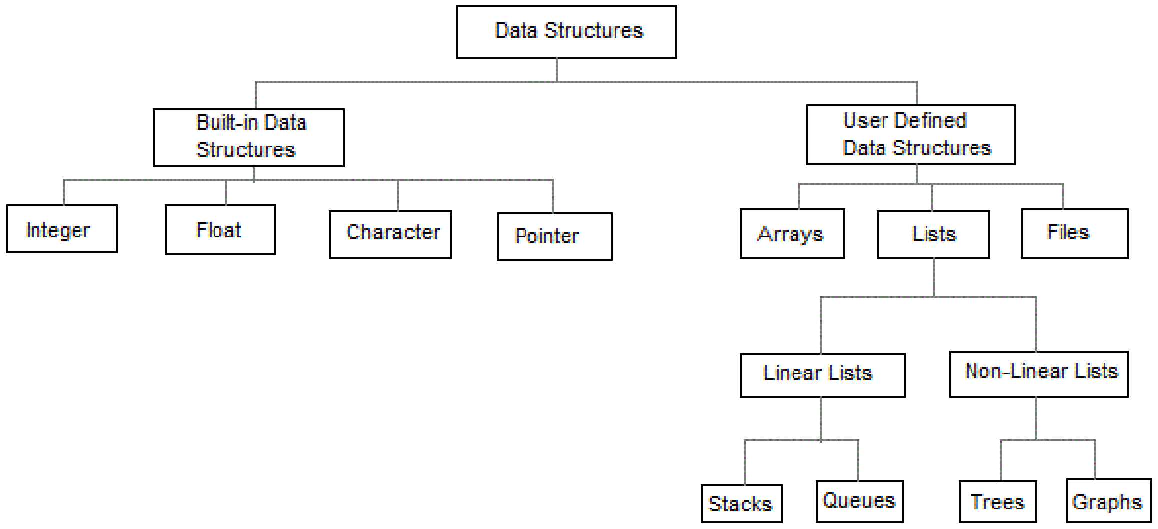
Primitive Data structure

The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value.

Non-Primitive Data structure

The non-primitive data structure is divided into two types:

- Linear data structure
- Non-linear data structure



The data structures can also be classified on the basis of the following characteristics:

Linear:	In Linear data structures, the data items are arranged in a linear sequence. Example: Array
Non-Linear:	In Non-Linear data structures, the data items are not in sequence. Example: Tree, Graph
Homogeneous:	In homogeneous data structures, all the elements are of same type. Example: Array
Non-Homogeneous:	In Non-Homogeneous data structure, the elements may or may not be of the same type. Example: Structures
Static:	Static data structures are those whose sizes and structures associated memory locations are fixed, at compile time. Example: Array

Dynamic:	Dynamic structures are those which expands or shrinks depending upon the program need and its execution. Also, their associated memory locations changes. Example: Linked List created using pointers

Advantages of Data structures

- **Efficiency:** If the choice of a proper data structure, it makes the program very efficient in terms of time and space.
- **Reusability:** The data structures provide reusability means that multiple client programs can use the data structure.
- **Abstraction:** The data structure provides the level of abstraction. The client cannot see the internal working of the data structure, so it does not have to worry about the implementation part. The client can only see the interface.

Applications of Data Structure and Algorithms

- Fibonacci number series
- Knapsack problem
- Tower of Hanoi
- All pair shortest path by Floyd-Warshall
- Shortest path by Dijkstra
- Project scheduling

Operations on Data Structures

The basic operations that are performed on data structures are as follows:

Create:	Create operation results in reserving memory for the data structure.
Insertion:	Insertion means addition of a new data element in a data structure.
Update:	Modifies the data in the data structure.
Deletion:	Deletion means removal of a data element from a data structure if it is found
Searching:	Searching involves searching for the specified data element in a data structure.
Traversal:	Traversal of a data structure means processing all the data elements present in it.
Sorting:	Arranging data elements of a data structure in a specified order is called sorting.

Merging: Combining elements of two similar data structures to form a new data structure of the same type, is called merging.

Abstract Data Type

- The abstract datatype is special kind of datatype, whose behavior is defined by a set of values and set of operations.
- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called abstract because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.
- Some examples of ADT are Stack, Queue, List etc.
- Application programmers are concerned only with using that type and calling its methods without worrying much about how the type is implemented.

Benefits of using Abstract Data Types

- Code is easier to understand.
- Implementations of ADTs can be changed (e.g., for efficiency) without requiring changes to the program that uses the ADTs.
- ADTs can be reused in future programs.

Memory Allocation in C

There are two types of memory allocation

1. Compile time or Static allocation
2. Run time or dynamic memory allocation (using pointer)

Dynamic Memory Allocation

Dynamic Memory Allocation can be defined as a procedure in which the size of a data structure is changed during the runtime.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

<u>malloc() method:</u>

“malloc” or **“memory allocation”** method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It initializes each block with default garbage value.

Syntax:

ptr = (cast-type*) malloc(byte-size)

For Example:

ptr = (int*) malloc(100 * sizeof(int));

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

```
#include <stdio.h>

#include <stdlib.h>

#include<conio.h>

int main()

{



int *ptr;

int n, i;



   // Get the number of elements for the array

   n = 5;

   printf("Enter number of elements: %d\n", n);
```

```
// Dynamically allocate memory using malloc()
ptr = (int*)malloc(n * sizeof(int));

// Check if the memory has been successfully
// allocated by malloc or not
if (ptr == NULL) {
    printf("Memory not allocated.\n");
    exit(0);
}
else {
    // Memory has been successfully allocated
    printf("Memory successfully allocated using malloc.\n");

    printf("Enter the element of array");
    for (i = 0; i < n; ++i) {
      scanf("%d",ptr+i);
    }


    printf("The elements of the array are: ");
    for (i = 0; i < n; ++i) {
        printf("%d, ", ptr[i]);
    }
    }

   getch();

    return 0;

}
```

Memory allocation to structure variable

```
struct student
 {
   Int roll_no;
   Char name[10];
   Float per;

};

Struct student *ptr;

Ptr=(struct student *) malloc(sizeof(struct student);
```

calloc() method:

“calloc” or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value ‘0’.

Syntax:

```
ptr = (cast-type*)calloc(n, element-size);
```

For Example:

ptr = (float*) calloc(25, sizeof(float));

This statement allocates contiguous space in memory for 25 elements each with the size of the float.

malloc()	**calloc()**
Malloc() function will create a single block of memory of size specified by the user.	Calloc() function can assign multiple blocks of memory for a variable.
Malloc function contains garbage value.	The memory block allocated by a calloc function is always initialized to zero.
Calloc is slower than malloc.	Malloc is faster than calloc.
It is not secure as compare to calloc.	It is secure to use compared to malloc.
Time efficiency is higher than calloc().	Time efficiency is lower than malloc().
It does not perform initializes of memory.	It performs memory initialization.

free() method:

“free” method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax:

```
free(ptr);
```

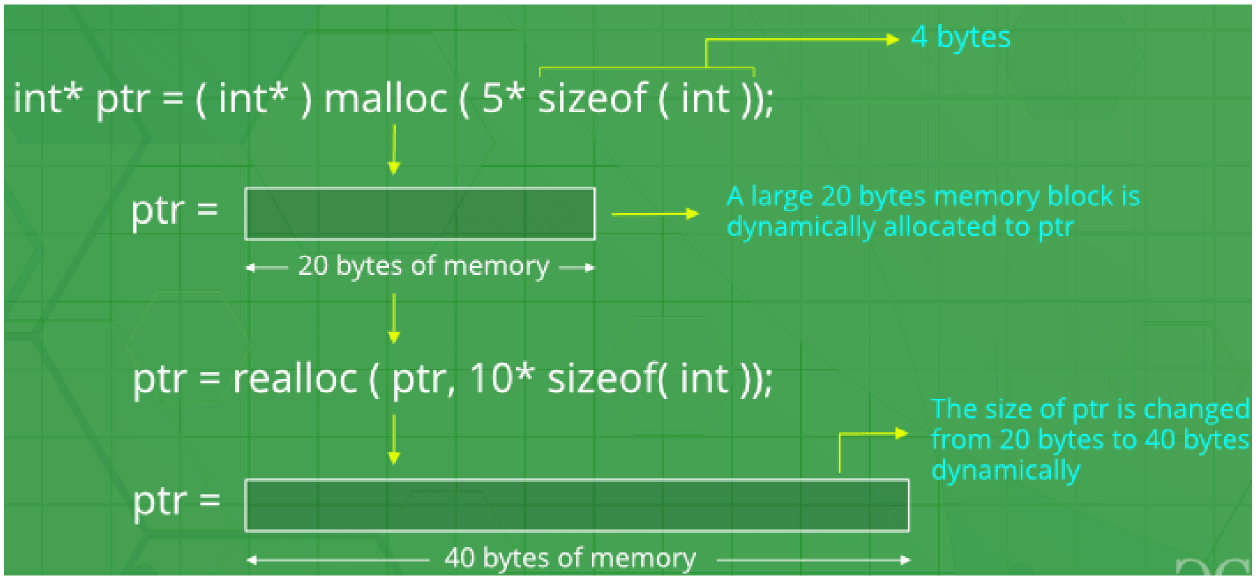
realloc() method

“realloc” or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with default garbage value.

Syntax:

```
ptr = realloc(ptr, newSize);
```

where ptr is reallocated with new size 'newSize'.


int* ptr = (int*) malloc (5* sizeof (int));
4 bytes
ptr =
A large 20 bytes memory block is dynamically allocated to ptr
20 bytes of memory
ptr = realloc (ptr, 10* sizeof(int));
The size of ptr is changed from 20 bytes to 40 bytes dynamically
ptr =
40 bytes of memory

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include<conio.h>

int main()
{


   int* ptr;
   int n, i,s;

    clrscr();
    s=0;

   printf("Enter the size of array");
   scanf("%d",&n);
     s=n;
   // Dynamically allocate memory using calloc()
   ptr = (int*)calloc(n, sizeof(int));

   if (ptr == NULL) {
          printf("Memory not allocated.\n");
```

```
            exit(0);
        }
        else {


          printf("Enter the elements of array");
              for (i = 0; i < n; i++)
          {
                scanf("%d",ptr+i);
              }

          // Get the new size for the array

              printf("\nEnter the new size of the array:");
               scanf("%d",&n);
              // Dynamically re-allocate memory using realloc()
              ptr = realloc(ptr, n * sizeof(int));


              printf("Memory successfully re-allocated using realloc.\n");

          printf("Enter the elements of an array");
              for (i = s; i < n;i++)
              {
                scanf("%d", ptr+i);
              }

              // Print the elements of the array
              printf("The elements of the array are: ");
              for (i = 0; i < n; i++) {
                 printf("%d, ", ptr[i]);
              }

              free(ptr);
        }
      getch();
        return 0;
    }
```

Advantages of dynamic memory allocation:

- Dynamic Allocation is done at run time.
- Data structures can grow and shrink to fit changing data requirements.
- We can allocate (create) additional storage whenever we need them.

- We can de-allocate (free/delete) dynamic space whenever we are done with them.
- Thus we can always have exactly the amount of space required - no more, no less.

Disadvantages of dynamic memory allocation:

- As the memory is allocated during runtime, it requires more time.
- Memory needs to be freed by the user when done. This is important as it is more likely to turn into bugs that are difficult to find.

Algorithm

An algorithm is a precise specification of a sequence of instructions to be carried out in order to solve a given problem. Each instruction tells what task is to be done. There should be a finite number of instructions in an algorithm and each instruction should be executed in a finite amount of time.

Write an algorithm to find the greatest number among three numbers:

```
Step 1: Read three numbers and store them in X, Y and Z
Step 2: Compare X and Y. if X is greater than Y then go to step 5 else step 3
Step 3: Compare Y and Z. if Y is greater than Z then print "Y is greatest" and go to step
7 otherwise go to step 4
Step 4: Print "Z is greatest" and go to step 7
Step 5: Compare X and Z. if X is greater than Z then print "X is greatest" and go to step
7 otherwise go to step 6
Step 6: Print "Z is greatest" and go to step 7
        Step 7: Stop
```

The performance of an algorithm can be analyzed based on following principles:

- Correctness: Correctness of an algorithm helps in producing correct result.
- Simplicity: Simplicity of an algorithm helps in verifying the correctness of an algorithm. It helps in writing and modifying program easily.
- Optimality: Optimality helps in finding the best possible way to solve given problems.
- Readability: makes the code readable.
- Space complexity
- Time complexity

<u>**Space complexity**</u>

- Space complexity is the amount of space required to solve an algorithm.
- The space complexity of a computer program is the amount of memory space required to solve a problem as a function of **size of the input.**

  S(p) =c+sp(i)

  where **c** is a fixed space, which is independent of input and output . It is a constant.

where **sp(i)** is variable space and it depends on i.

Components of space complexity

There are three components of space complexity:

Instruction or Code Space:

- This space holds the collected version of the program instructions.

Data space:

- This space holds all the static data such as constant, variables and dynamic data such as growing structure.

Environment Stack Space:

- This space is used to store return address for the functions. So that, execution can begin again where they left earlier and it may also be used for parameters.

Time complexity

- Time complexity is the amount of time taken by an algorithm to execute a task.
- The time complexity of a program is the amount of time taken to solve a problem or function of size as inputs.
- If the time complexity of an algorithm is better, then the algorithm will carry out its work as fast as possible.

Components of Time Complexity

- The factors that affects the space complexity also affects time complexity.
- The time needed to execute an algorithm depends on several factors.
- Usually, it is difficult to find the exact time, so that we approximate the value.
- This approximate value can be found by assuming program ‘p’ by taking time is equal to the sum of compile time and run time.
- Compile time does not depend on the size of input hence, we will not consider compile-time and hence we will confirm ourselves to consider only the run time which depends on the size of the input.

  Let ‘t_p’ be the run time of the program.

  Again, we know that ‘ t_p’ depends on several factors. It is difficult to determine the exact value, hence we have to settle for less.

Approximately the ‘ t_p’ is given by,

$t_p(n) = C_a\,ADD(n) + C_s\,SUB(n) + C_m MUL(n) + \ldots\ldots\ ..$

Here, n=Size of input

C_a=Time needed for addition

C_s=Time needed for subtraction

C_m=Time needed for multiplication

Hence, we can say that the components of time complexity are operations (addition, subtraction, etc.) performed by processor.

In mathematically, time complexity is expressed as

Time complexity $t_p(h) = C + t_p(n)$

Asymptotic Analysis

Asymptotic analysis of an algorithm refers to defining the mathematical bordering of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Usually, the time required by an algorithm falls under three types −

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

Asymptotic notation

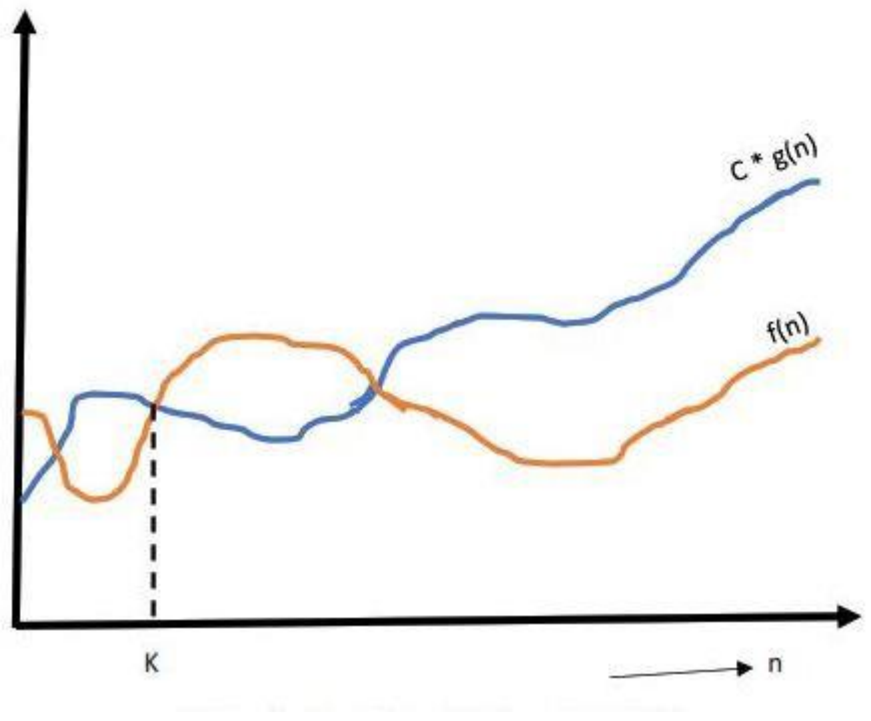
Asymptotic notation describes the algorithm efficiency and performance in a meaningful way. It describes the behavior of time or space complexity for large instance characteristics. Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

The function f(n)=O(g(n)), if and only if there exist a positive constant C and K such that f(n)<=C * g(n) for all n, n≥K.



[Big Oh Notation (f(n) ≤ C * g(n))]

O(1) is used to denote constants.
O(n) is called linear
O($n^{2)}$ is called quadratic
O(n^n) is called exponential

Example:
f(x)=5x3+3x2+4 find big oh(O) of f(x)
solution: f(x)= 5x3+3x2+4<= 5x3+3x3+4x3 if x>0
<=12x3
=>f(x)<=c.g(x)
where c=12 and g(x)=x3

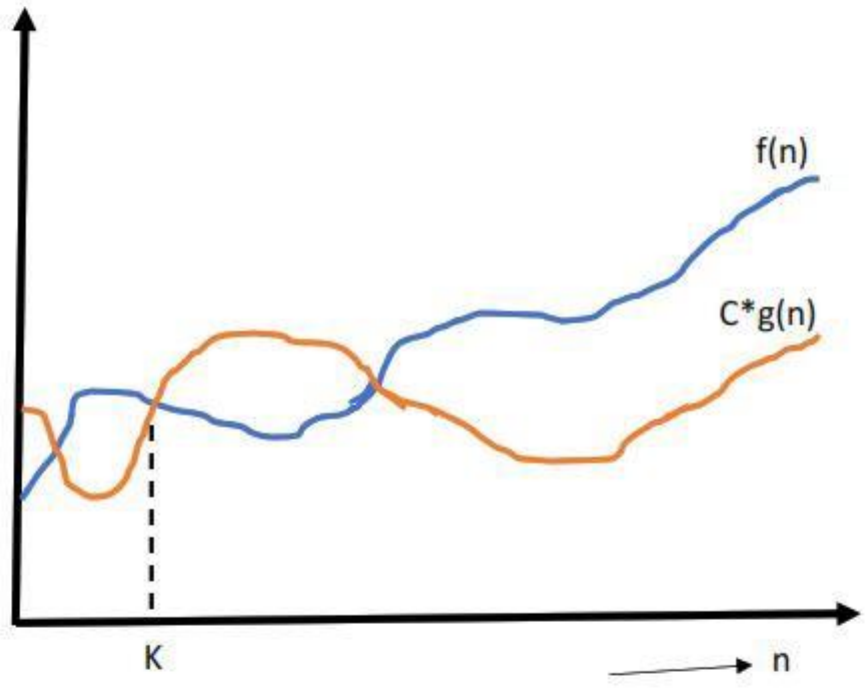
Thus by definition of big oh O(f(x))=O(x3)

Omega Notation, Ω:

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

The function f(n)=Ω(g(n)), if and only if there exist a positive constant C and K such that f(n)≥C * g(n) for all n n≥K.

The above relation says that g(n) is a lower bound of f(n).

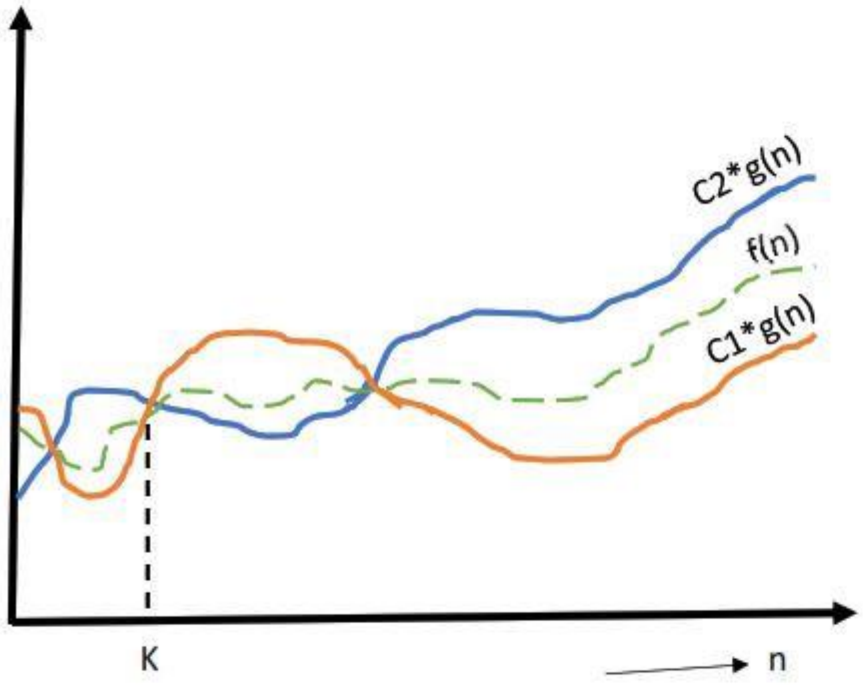


[Omega (f(n)=Ω(g(n))) notation]

Theta Notation, θ:

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.

The function f(n)=θ(g(n)), if and only if there exist a positive constant C1, C2 and K such that C1*g(n) ≤ f(n) ≤ C2 * g(n) for all n n≥K



[Big Theta notation (C1*g(n) ≤ f(n) ≤ C2 * g(n))]