# Pointer

In programming language, when we create a variable, it is assigned some space the computer memory. Every memory location has its address. To know the location in the computer memory where the data is stored, C++ provides the **& (reference)** operator. The **&** operator returns the address that a variable occupies.

string food = "Pizza";

cout << &food; // output : 0x7fff5fbff8ac

**Pointer:**

1. A **pointer** is a variable that stores address of another variable.
2. A pointer can also be used to refer to another pointer function.
3. A pointer can be incremented/decremented to point to the next/ previous memory location.
4. The purpose of pointer is to save memory space and achieve faster execution time.
5. The pointer in C++ language can be declared using ∗ (asterisk symbol).

**Datatype \*value\_name;**

**Example:**

int main () {

int number=30;

int ∗   p;

p=&number

}

**Pointers and Arrays:**

1. Arrays and pointers work based on a related concept.
2. The array name itself denotes the base address of the array.
3. To assign the address of an array to a pointer, you should not use an ampersand (&).

**Example:**

**int arr [20];**

**int \* ip;**

**ip = arr;**

### Types of Pointers:

1. Null Pointer
2. Void Pointer
3. Wild pointer
4. Dangling pointer
5. Complex pointer
6. Near pointer
7. Far pointer
8. Huge pointer

**Null Pointer:**

1. If there is no exact address that is to be assigned, then the pointer variable can be assigned a NULL.
2. It should be done during the declaration.
3. The value of null is 0.
4. It is useful for handling errors when using malloc function.

**Example :**

**int main() {**

**int \*ptr = NULL;**

**cout << ptr ;**

**return 0;**

**}**

**Void pointer:**

1. void pointer is also called as a generic pointer.
2. It does not have any standard data type.
3. A void pointer is created by using the keyword void.
4. It can be used to store an address of any variable.

**Example:**

**int main () {**

**int n = 10;**

**int \*ptr = &n;**

**cout<< \*(int\*) ptr ;**

**}**

**Dangling pointer:**

1. A dangling pointer is a pointer which points to some non-existing memory location.
2. When we free the pointer but not re-initialize the pointer, then pointer is still pointing to the deallocated memory.
3. So, we need to re-initialized the memory.

**Example:**

int main() {

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

……….

………

free(ptr);

}

**Wild pointer:**

1. Wild pointers are also known as uninitialized pointers.
2. These pointers usually point to some arbitrary memory location and may cause a program to cash or misbehave

**Example:**

int main() {

int \*ptr ;

\*ptr = 10;

}

# Memory allocation

1. Memory allocation is a process by which computer programs and services are assigned with physical or virtual memory space.
2. Memory allocation is the process of reserving a partial or complete portion of computer memory for the execution of programs and processes.
3. Memory allocation is achieved through a process known as memory management.
4. Memory allocation is primarily a computer hardware operation but is managed through operating system and software applications.

## Memory Layout

When we create a C program and run the program, its executable file is stored in the RAM of the computer in an organized manner.



A typical memory representation of a C program consists of the following sections.

1. Text segment/ code segment (instructions)
2. Initialized data segment
3. Uninitialized data segment (bss)
4. Heap
5. Stack

**1. Text segment**

The text segment is also known as the code segment.

When we compile any program, it creates an executable file like a.out, .exe, etc., that gets stored in the text or code section of the RAM memory.

If we store the instructions in the hard disk, then the speed for accessing the instructions from the hard disk becomes slower as hard disk works on the serial communication so taking the data from the hard disk will be slower, whereas the RAM is directly connected to the data and address bus so accessing the data from the RAM is faster.

**2. Data section**

The data which we use in our program will be stored in the data section. Since the variables declared inside the main() function are stored in the stack, but the variables declared outside the main() method will be stored in the data section. The variables declared in the data section could be stored in the form of initialized, uninitialized, and it could be local or global. Therefore, the data section is divided into four categories, i.e., initialized, uninitialized, global, or local.

Let's understand this scenario through an example.

#include<stdio.h>

int var1;

int var2 = 10;

void function1()

{

    printf("I am function1");

}

int main()

{

   function1();

   return 0;

}

In the above code, var1 and var2 variables are declared outside the main() function where var1 is the uninitialized variable, whereas the var2 is an initialized variable. These variables can be accessed anywhere in the program because these variables are not a part of the main() in the stack.

The data section consists of two segments:

Uninitialized data segment

Initialized data segment

Uninitialized data segment

The uninitialized data segment is also known as a .bss segment that stores all the uninitialized global, local and external variables. If the global, static and external variables are not initialized, they are assigned with zero value by default.

The .bss segment stands for Block Started by symbol. The bss segment contains the object file where all the statically allocated variables are stored. Here, statically allocated objects are those objects without explicit initialization are initialized with zero value. In the above code, var1 is an uninitialized variable so it is stored in the uninitialized data section.

Let's look at some examples of uninitialized data segment.

#include<stdio.h>

char a;    // uninitialized global variable..

int main()

{

    static int a;   // uninitialized static variable..

    return 0;

}

**Initialized data segment**

An initialized data segment is also known as the data segment. A data segment is a virtual address space of a program that contains all the global and static variables which are explicitly initialized by the programmer.

The values of variables in a data segment are not read-one, i.e., they can be modified at run time. This data segment can be further classified into categories:

Initialized read-only area: It is an area where the values of variables cannot be modified.

Initialized read-write area: It is an area where the values of variables can also be altered.

For example: the global variables like char str[] = "javatpoint" and int a=45; will be stored in the initialized read-write area. If we create the global variable like const char\* string = "javatpoint"; the literal "javatpoint" would be stored in the initialized read area, whereas the char pointer variable would be stored in the initialized write area.

#include<stdio.h>

char string[] = "javatpoint";  // global variable stored in initialized data segment in read-write area..

int main()

{

   static int i = 90;   // static variable stored in initialized data segment..

   return 0;

}

3. Stack

When we define a function and call that function then we use the stack frame. The variables which are declared inside the function are stored in the stack. The function arguments are also stored in the function as the arguments are also a part of the function. Such a type of memory allocation is known as static memory allocation because all the variables are defined in the function, and the size of the variables is also defined at the compile time. The stack section plays a very important role in the memory because whenever the function is called, a new stack frame is created.

Stack is also used for recursive functions. When the function is called itself again and again inside the same function which causes the stack overflow condition and it leads to the segmentation fault error in the program.

4. Heap

Heap memory is used for the dynamic memory allocation. Heap memory begins from the end of the uninitialized data segment and grows upwards to the higher addresses. The malloc() and calloc() functions are used to allocate the memory in the heap. The heap memory can be used by all the shared libraries and dynamically loaded modules. The free() function is used to deallocate the memory from the heap.

#include<stdio.h>

int main()

{

    int \*ptr = (int\*)malloc(sizeof(int)) ; // memory gets allocated in the heap segment.

    return 0;

}

Memory allocation has two core types;

1. Static Memory Allocation: The program is allocated memory at compile time.
2. Dynamic Memory Allocation: The programs are allocated with memory at run time.

**Static memory allocation**

1. Static memory allocation allocated by the compiler.
2. Exact size and type of memory must be known at compile time. ​​​​​​​
3. It uses [stack](https://www.geeksforgeeks.org/stack-data-structure/) for managing the static allocation of memory

int x, y;

float a[5];

**Run-time / Dynamic allocation**

memory is allocated at run time.

It uses [heap](https://www.geeksforgeeks.org/heap-data-structure/) for managing the dynamic allocation of memory

memory can be increased while executing program.

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

# Structure

A structure is a composite data type that defines a grouped list of variables that are to be placed under one name in a block of memory. It allows different variables to be accessed by using a single pointer to the structure.

**Syntax**

struct structure\_name

{

    data\_type member1;

    data\_type member2;

    .

    .

    data\_type memeber;

};

**Advantages:**

1. It can hold variables of different data types.
2. We can create objects containing different types of attributes.
3. It allows us to re-use the data layout across programs.
4. It is used to implement other data structures like linked lists, stacks, queues, trees, graphs etc.

**Program:**

**void** main( )

{

struct employee

{

**int** id ;

**float** salary ;

**int** mobile ;

} ;

struct employee e1,e2,e3 ;

clrscr();

printf ("\nEnter ids, salary & mobile no. of 3 employee\n"

scanf ("%d %f %d", &e1.id, &e1.salary, &e1.mobile);

scanf ("%d%f %d", &e2.id, &e2.salary, &e2.mobile);

scanf ("%d %f %d", &e3.id, &e3.salary, &e3.mobile);

printf ("\n Entered Result ");

printf ("\n%d %f %d", e1.id, e1.salary, e1.mobile);

printf ("\n%d%f %d", e2.id, e2.salary, e2.mobile);

printf ("\n%d %f %d", e3.id, e3.salary, e3.mobile);

getch();

}

Data Structure and Algorithm

# Data Structures

**Data:**

Data can be defined as a representation of facts, concepts, or instructions in a formalized manner, which should be suitable for communication, interpretation, or processing by human or electronic machine. Data is represented with the help of characters such as alphabets (A-Z, a-z), digits (0-9) or special characters (+, -, /, \*, <,>, = etc.).Data can be defined as an elementary value or the collection of values.

Example - **student's name and its id are the data about the student.** Catch

**Group Items:**

Data items which have subordinate data items are called Group item. Example **- name of a student can have first name and the last name.**

**Record:**

Record can be defined as the collection of various data items. Example - **if we talk about the student entity, then its name, address, course, and marks can be grouped together to form the record for the student.**

**File:**

A File is a collection of various records of one type of entity. **Example - if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.**

**Attribute and Entity:**

An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

**Field:**

Field is a single elementary unit of information representing the attribute of an entity.

# Data type:

Data types in C++ are mainly divided into three types:

1. Primitive Data Types:
2. Derived Data Types:
3. Abstract or User-Defined Data Types

**1. Primitive Data Types**: These data types are built-in or predefined data types and can be used directly by the user to declare variables. Primitive data types available in C++ are:

1. Integer
2. Character
3. Boolean
4. Floating Point
5. Double Floating Point
6. Valueless or Void
7. Wide Character

2**. Derived Data Types:** The data types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types. These can be of four types namely:

1. Function
2. Array
3. Pointer
4. Reference

**3. Abstract or User-Defined Data Types:** These data types are defined by the user itself. Like, as defining a class in C++ or a structure. C++ provides the following user-defined datatypes:

1. Class
2. Structure
3. Union
4. Enumeration
5. Typedef defined Datatype

# What are Data Structures?

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently.

Data Structures are:

1. arrays,
2. Linked List,
3. Stack,
4. Queue

**Need of Data Structures:**

1. **Processor speed:** To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.
2. **Data Search:** Consider an inventory size of 106 items in a store, if our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.
3. **Multiple requests:** If thousands of users are searching for the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process in order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

**Advantages of Data Structures**

1. **Efficiency:** Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a particular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.
2. **Reusability:** Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.
3. **Abstraction:** Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

# Types of Data Structures



There are two types of data structures:

1. Primitive data structure
2. Non-primitive data structure

**Primitive Data structure:**

The primitive data structures are primitive data types that can hold a single value. They are ---

1. int,
2. char,
3. float,
4. double, and
5. pointer

**Non-Primitive Data structure**:

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

1. Linear data structure (Arrays, linked list, Stacks, and Queues)
2. Non-linear data structure (**trees and graphs)**

# ****Linear Data Structure****

The arrangement of data in a sequential manner is known as a linear data structure. The data structures used for this purpose are Arrays, linked list, Stacks, and Queues. In these data structures, one element is connected to only one another element in a linear form.

We will discuss the above data structures in brief in the coming topics. Now, we will see the common operations that we can perform on these data structures. It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

**Linear data structures can also be classified as:**

1. **Static data structure: -- Array**
2. **Dynamic data structure:**  -- Linked list, Stack , Queue

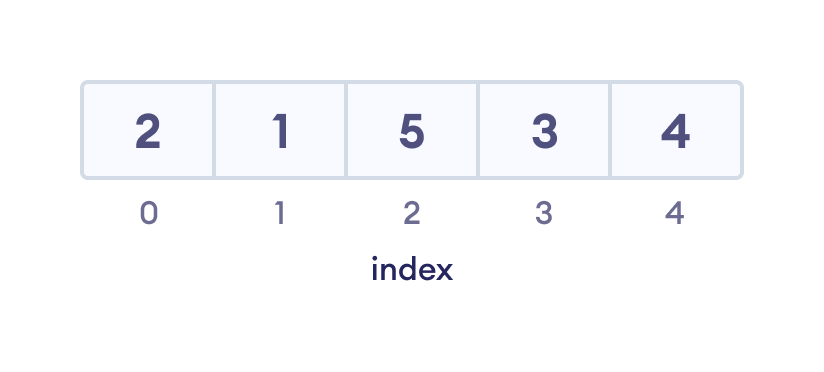
**Arrays:**

An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array age are:

age[0], age[1], age[2], age[3],......... age[98], age[99].



An array with each element represented by an index

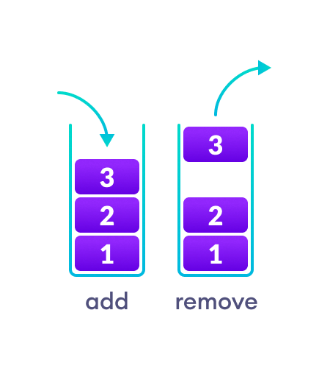
**Linked List:**

Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.



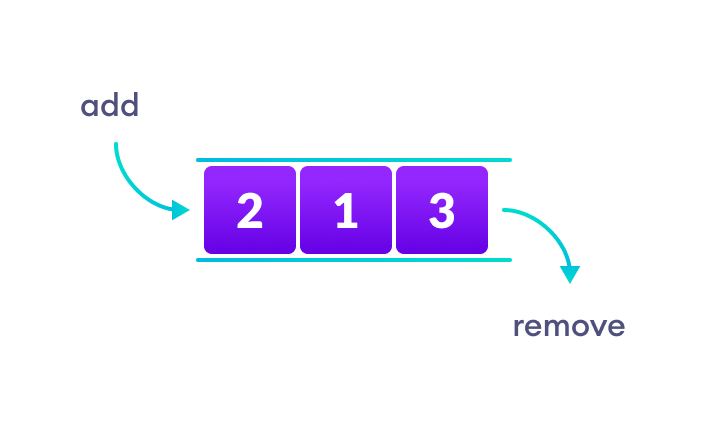
**Stack:**

Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**. A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.



**Queue:**

Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**. It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.



**Non-Linear Data Structures:**

This data structure does not form a sequence i.e., each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

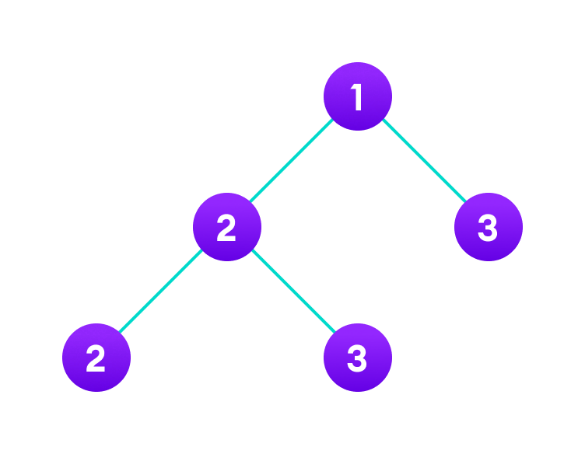
Types of Non-Linear Data Structures are given below:

1. **Trees:**
2. **Graphs:**

**Trees:**

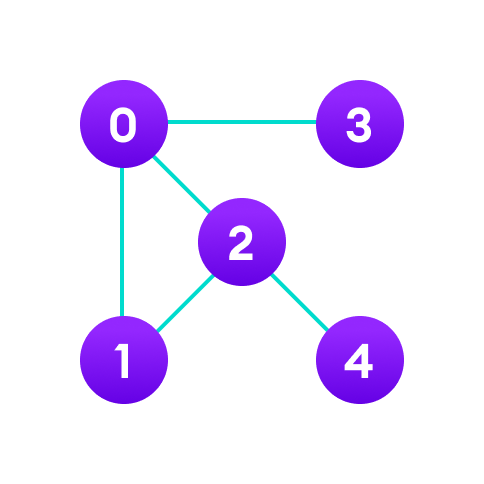
Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called **leaf node** while the topmost node is called **root node**. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one child except the leaf nodes whereas each node can have at most one parent except the root node. Trees can be classified into many categories which will be discussed later in this tutorial.



**Graphs:**

Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree cannot have the one.



# Algorithm

An algorithm is a process, or a set of rules required to perform calculations or some other problem-solving operations especially by a computer. The formal definition of an algorithm is that it contains the finite set of instructions which are being carried in a specific order to perform the specific task. It is not the complete program or code; it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudocode.

**We need algorithms because of the following reasons:**

1. **Time is precious**
2. Scalability
3. **Performance:**
4. Memory

Algorithm Example:

The following are the steps required to add two numbers entered by the user:

Step 1: Start

Step 2: Declare three variables a, b, and sum.

Step 3: Enter the values of a and b.

Step 4: Add the values of a and b and store the result in the sum variable, i.e., sum= a + b.

Step 5: Print sum

Step 6: Stop

**Approaches of Algorithm**

**The following are the approaches used after considering both the theoretical and practical importance of designing an algorithm:**

1. **Brute force algorithm:** The general logic structure is applied to design an algorithm. It is also

known as an exhaustive search algorithm that searches all the possibilities to provide the required solution. Such algorithms are of two types:

1. **Optimizing:** Finding all the solutions of a problem and then take out the best solution or if the value of the best solution is known then it will terminate if the best solution is known.
2. **Sacrificing:** As soon as the best solution is found, then it will stop.
3. **Divide and conquer:** It is a very implementation of an algorithm. It allows you to design an algorithm in a step-by-step variation. It breaks down the algorithm to solve the problem in different methods. It allows you to break down the problem into different methods, and valid output is produced for the valid input. This valid output is passed to some other function.
4. **Greedy algorithm:** It is an algorithm paradigm that makes an optimal choice on each iteration with the hope of getting the best solution. It is easy to implement and has a faster execution time. But there are very rare cases in which it provides the optimal solution.
5. **Dynamic programming:** It makes the algorithm more efficient by storing the intermediate results. It follows five different steps to find the optimal solution for the problem:
6. It breaks down the problem into a subproblem to find the optimal solution.
7. After breaking down the problem, it finds the optimal solution out of these subproblems.
8. Stores the result of the subproblems is known as memorization.
9. Reuse the result so that it cannot be recomputed for the same subproblems.
10. Finally, it computes the result of the complex program.

**Branch and Bound Algorithm:**

The branch and bound algorithm can be applied to only integer programming problems. This approach divides all the sets of feasible solutions into smaller subsets. These subsets are further evaluated to find the best solution.

**Randomized Algorithm:**

As we have seen in a regular algorithm, we have predefined input and required output. Those algorithms that have some defined set of inputs and required output, and follow some described steps are known as deterministic algorithms. What happens that when the random variable is introduced in the randomized algorithm? In a randomized algorithm, some random bits are introduced by the algorithm and added in the input to produce the output, which is random in nature. Randomized algorithms are simpler and efficient than the deterministic algorithm.

**Backtracking:**

Backtracking is an algorithmic technique that solves the problem recursively and removes the solution if it does not satisfy the constraints of a problem.

The major categories of algorithms are given below:

1. Sort: Algorithm developed for sorting the items in a certain order.
2. Search: Algorithm developed for searching the items inside a data structure.
3. Delete: Algorithm developed for deleting the existing element from the data structure.
4. Insert: Algorithm developed for inserting an item inside a data structure.
5. Update: Algorithm developed for updating the existing element inside a data structure.

Algorithm Complexity

The performance of the algorithm can be measured in two factors:

**Time complexity:**

The time complexity of an algorithm is the amount of time required to complete the execution. The time complexity of an algorithm is denoted by the big O notation. Here, big O notation is the asymptotic notation to represent the time complexity. The time complexity is mainly calculated by counting the number of steps to finish the execution. Let's understand the time complexity through an example.

sum=0;

**for** i=1 to n

sum= sum + i ;

**return** sum;

In the above code, the time complexity of the loop statement will be at least n, and if the value of n increases, then the time complexity also increases. While the complexity of the code, i.e., return sum will be constant as its value is not dependent on the value of n and will provide the result in one step only. We generally consider the worst-time complexity as it is the maximum time taken for any given input size.

**Space complexity:**

An algorithm's space complexity is the amount of space required to solve a problem and produce an output. Similar to the time complexity, space complexity is also expressed in big O notation.

For an algorithm, the space is required for the following purposes:

1. To store program instructions
2. To store constant values
3. To store variable values
4. To track the function calls, jumping statements, etc.

Auxiliary space: The extra space required by the algorithm, excluding the input size, is known as an auxiliary space. The space complexity considers both the spaces, i.e., auxiliary space, and space used by the input.

**Space complexity = Auxiliary space + Input size.**

Types of Algorithms

**The following are the types of algorithms:**

1. **Search Algorithm**
2. **Sort Algorithm**

# **Search Algorithm**

On each day, we search for something in our day-to-day life. Similarly, with the case of computer, huge data is stored in a computer that whenever the user asks for any data then the computer searches for that data in the memory and provides that data to the user. There are mainly two techniques available to search the data in an array:

1. **Linear search**
2. **Binary search**

**Linear Search**

Linear search is a very simple algorithm that starts searching for an element or a value from the beginning of an array until the required element is not found. It compares the element to be searched with all the elements in an array, if the match is found, then it returns the index of the element else it returns -1. This algorithm can be implemented on the unsorted list.

**Binary Search**

A Binary algorithm is the simplest algorithm that searches the element very quickly. It is used to search the element from the sorted list. The elements must be stored in sequential order or the sorted manner to implement the binary algorithm. Binary search cannot be implemented if the elements are stored in a random manner. It is used to find the middle element of the list.

# Sorting Algorithms

Sorting algorithms are used to rearrange the elements in an array or a given data structure either in an ascending or descending order. The comparison operator decides the new order of the elements.

Why do we need a sorting algorithm?

1. An efficient sorting algorithm is required for optimizing the efficiency of other algorithms like binary search algorithm as a binary search algorithm requires an array to be sorted in a particular order, mainly in ascending order.
2. It produces information in a sorted order, which is a human-readable format.
3. Searching a particular element in a sorted list is faster than the unsorted list.

# Asymptotic Analysis

The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations.

**Our focus would be on finding the time complexity rather than space complexity, and by finding the time complexity, we can decide which data structure is the best for an algorithm.**

C++ vs Java

**How to find the Time Complexity or running time for performing the operations?**

The measuring of the actual running time is not practical at all. The running time to perform any operation depends on the size of the input.

Suppose we have an array of five elements, and we want to add a new element at the beginning of the array. To achieve this, we need to shift each element towards right, and suppose each element takes one unit of time. There are five elements, so five units of time would be taken. Suppose there are 1000 elements in an array, then it takes 1000 units of time to shift. It

So, that time complexity depends upon the input size.

Therefore, if the input size is n, then f(n) is a function of n that denotes the time complexity.

How to calculate f(n)?

Calculating the value of f(n) for smaller programs is easy but for bigger programs, it's not that easy. We can compare the data structures by comparing their f(n) values. We can compare the data structures by comparing their f(n) values. We will find the growth rate of f(n) because there might be a possibility that one data structure for a smaller input size is better than the other one but not for the larger sizes. Now, how to find f(n).

f(n) = 5n2 + 6n + 12

where n is the number of instructions executed, and it depends on the size of the input.

When n=1

% of running time due to 5n2 = Asymptotic Analysis \* 100 = 21.74%

% of running time due to 6n = Asymptotic Analysis \* 100 = 26.09%

% of running time due to 12 = Asymptotic Analysis \* 100 = 52.17%

From the above calculation, it is observed that most of the time is taken by 12. But, we have to find the growth rate of f(n), we cannot say that the maximum amount of time is taken by 12. Let's assume the different values of n to find the growth rate of f(n).

|  |  |  |  |
| --- | --- | --- | --- |
| n | 5n2 | 6n | 12 |
| 1 | 21.74% | 26.09% | 52.17% |
| 10 | 87.41% | 10.49% | 2.09% |
| 100 | 98.79% | 1.19% | 0.02% |
| 1000 | 99.88% | 0.12% | 0.0002% |

As we can observe in the above table that with the increase in the value of n, the running time of 5n2 increases while the running time of 6n and 12 also decreases. Therefore, it is observed that for larger values of n, the squared term consumes almost 99% of the time. As the n2 term is contributing most of the time, so we can eliminate the rest two terms.

**Therefore,**

f(n) = 5n2

Here, we are getting the approximate time complexity whose result is very close to the actual result. And this approximate measure of time complexity is known as an Asymptotic complexity. Here, we are not calculating the exact running time, we are eliminating the unnecessary terms, and we are just considering the term which is taking most of the time.

In mathematical analysis, asymptotic analysis of algorithm is a method of defining the mathematical bound of its run-time performance. Using the asymptotic analysis, we can easily conclude the average-case, best-case and worst-case scenario of an algorithm.

It is used to mathematically calculate the running time of any operation inside an algorithm.

**Example:** Running time of one operation is x(n) and for another operation, it is calculated as f(n2). It refers to running time will increase linearly with an increase in 'n' for the first operation and running time will increase exponentially for the second operation. Similarly, the running time of both operations will be the same if n is significantly small.

Usually, the time required by an algorithm comes under three types:

1. **Worst case:** It defines the input for which the algorithm takes a huge time.
2. **Average case:** It takes average time for the program execution.
3. **Best case:** It defines the input for which the algorithm takes the lowest time

Asymptotic Notations

The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below:

1. Big oh Notation (O)
2. Omega Notation (Ω)
3. Theta Notation (θ)

Big oh Notation (O)

Big O notation is an asymptotic notation that measures the performance of an algorithm by simply providing the order of growth of the function. This notation provides an upper bound on a function which ensures that the function never grows faster than the upper bound. So, it gives the least upper bound on a function so that the function never grows faster than this upper bound.

It is the formal way to express the upper boundary of an algorithm running time. It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation. It is represented as shown below:



**For example:**

If **f(n)** and **g(n)** are the two functions defined for positive integers. then

**f(n)** = **O(g(n))** as **f(n) is big oh of g(n)** or f(n) is on the order of g(n))

such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0 }

This implies that f(n) does not grow faster than g(n), or g(n) is an upper bound on the function f(n). In this case, we are calculating the growth rate of the function which eventually calculates the worst time complexity of a function, i.e., how worst an algorithm can perform.

**Let's understand through examples**

Example 1: f(n)=2n+3 , g(n)=n

Now, we must find **Is f(n)=O(g(n))?**

To check f(n)=O(g(n)), it must satisfy the given condition:

**f(n)<=c.g(n)**

First, we will replace f(n) by 2n+3 and g(n) by n.

2n+3 <= c.n

Let's assume c=5, n=1 then

2\*1+3<=5\*1

5<=5

For n=1, the above condition is true.

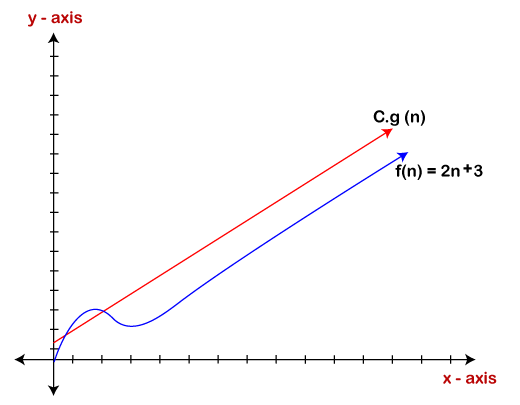
If n=2

2\*2+3<=5\*2

7<=10

For n=2, the above condition is true.

We know that for any value of n, it will satisfy the above condition, i.e., 2n+3<=c.n. If the value of c is equal to 5, then it will satisfy the condition 2n+3<=c.n. We can take any value of n starting from 1, it will always satisfy. Therefore, we can say that for some constants c and for some constants n0, it will always satisfy 2n+3<=c.n. As it is satisfying the above condition, so f(n) is big oh of g(n) or we can say that f(n) grows linearly. Therefore, it concludes that c.g(n) is the upper bound of the f(n). It can be represented graphically as:



The idea of using big o notation is to give an upper bound of a particular function, and eventually it leads to give a worst-time complexity. It provides an assurance that a particular function does not behave suddenly as a quadratic or a cubic fashion, it just behaves in a linear manner in a worst-case.

Omega Notation (Ω)

Omega Notation (Ω) describes lower bound of an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or the best-case time complexity. It determines what is the fastest time that an algorithm can run.

If we required that an algorithm takes at least certain amount of time without using an upper bound, we use big- Ω notation i.e. the Greek letter "omega". It is used to bound the growth of running time for large input size.

If  **f(n)** and **g(n)** are the two functions defined for positive integers. then,  **f(n) = Ω (g(n))** as **f(n) is Omega of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:

**f(n) >= c.g(n) for all n ≥ no and c>0**

**Let's consider a simple example.**

If f(n) = 2n+3, g(n) = n,

Is f(n)= **Ω** (g(n))?

It must satisfy the condition:

**f(n) >= c.g(n)**

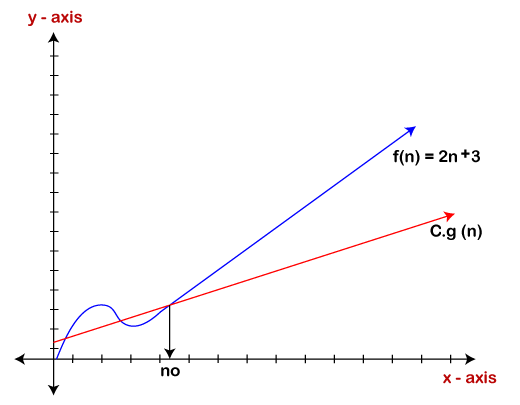
To check the above condition, we first replace f(n) by 2n+3 and g(n) by n.

**2n+3>=c\*n**

Suppose c=1

**2n+3>=n** (This equation will be true for any value of n starting from 1).

Therefore, it is proved that g(n) is big omega of 2n+3 function.



As we can see in the above figure that g(n) function is the lower bound of the f(n) function when the value of c is equal to 1. Therefore, this notation gives the fastest running time. But, we are not more interested in finding the fastest running time, we are interested in calculating the worst-case scenarios because we want to check our algorithm for larger input that what is the worst time that it will take so that we can take further decision in the further process.

Theta Notation (θ)

The theta notation mainly describes the average case scenarios. It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm. Big theta is mainly used when the value of worst-case and the best-case is same. It is the formal way to express both the upper bound and lower bound of an algorithm running time.

Let's understand the big theta notation mathematically:

Let f(n) and g(n) be the functions of n where n is the steps required to execute the program then:

**f(n)= θg(n)**

The above condition is satisfied only if when

**c1.g(n)<=f(n)<=c2.g(n)**

where the function is bounded by two limits. Upper and Lower limit, and f(n) comes in between. The condition **f(n)= θg(n)** will be true if and only if c1.g(n) is less than or equal to f(n) and c2.g(n) is greater than or equal to f(n). The graphical representation of theta notation is given below:



Let's consider the same example where f(n)=2n+3 g(n)=n

As c1.g(n) should be less than f(n) so c1 has to be 1 whereas c2.g(n) should be greater than f(n) so c2 is equal to 5. The c1.g(n) is the lower limit of the of the f(n) while c2.g(n) is the upper limit of the f(n).

c1.g(n)<=f(n)<=c2.g(n)

Replace g(n) by n and f(n) by 2n+3

c1.n <=2n+3<=c2.n

if c1=1, c2=2, n=1

1\*1 <=2\*1+3 <=2\*1

**1** <= **5** <= **2** // for n=1, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n)

**If n=2**

1\*2<=2\*2+3<=2\*2

2<=7<=4 // for n=2, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n)

Therefore, we can say that for any value of n, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n). Hence, it is proved that f(n) is big theta of g(n). So, this is the average-case scenario which provides the realistic time complexity.

Why we have three different asymptotic analysis?

As we know that big omega is for the best case, big oh is for the worst case while big theta is for the average case. Now, we will find out the average, worst and the best case of the linear search algorithm. Suppose we have an array of n numbers, and we want to find the element in an array using the linear search. In the linear search, every element is compared with the searched element on each iteration. Suppose if the match is found in a first iteration only, then the best case would be Ω(1), if the element matches with the last element, i.e., nth element of the array then the worst-case would-be O(n). The average case is the mid of the best and the worst-case, so it becomes **θ(n/1). The constant terms can be ignored in the time complexity so average case would be θ(n)**.

So, three different analyses provide the proper bounding between the actual functions. Here, bounding means that we have upper as well as lower limit which assures that the algorithm will behave between these limits only, i.e., it will not go beyond these limits.

Common Asymptotic Notations

|  |  |  |
| --- | --- | --- |
| constant | - | ?(1) |
| linear | - | ?(n) |
| logarithmic | - | ?(log n) |
| n log n | - | ?(n log n) |
| exponential | - | 2?(n) |
| cubic | - | ?(n3) |
| polynomial | - | n?(1) |
| quadratic | - | ?(n2) |