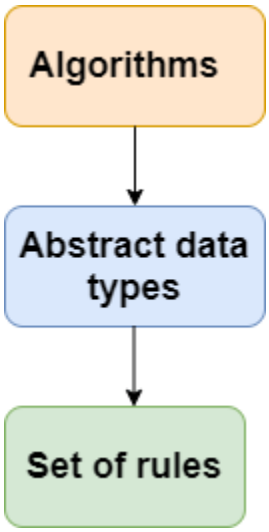
UNIT I

What is Data Structure?

The data structure name indicates itself that organizing the data in memory. There are many ways of organizing the data in the memory as we have already seen one of the data structures, i.e., array in C language. Array is a collection of memory elements in which data is stored sequentially, i.e., one after another. In other words, we can say that array stores the elements in a continuous manner. This organization of data is done with the help of an array of data structures. There are also other ways to organize the data in memory. Let's see the different types of data structures.

The data structure is not any programming language like C, C++, java, etc. It is a set of algorithms that we can use in any programming language to structure the data in the memory.

To structure the data in memory, 'n' number of algorithms were proposed, and all these algorithms are known as Abstract data types. These abstract data types are the set of rules.



Types of Data Structures

There are two types of data structures:

- Primitive data structure

- Non-primitive 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

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.

When one element is connected to the 'n' number of elements known as a non-linear data structure. The best example is trees and graphs. In this case, the elements are arranged in a random manner.

Data structures can also be classified as:

- **Static data structure:** It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed.
- **Dynamic data structure:** It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

Major Operations

The major or the common operations that can be performed on the data structures are:

- **Searching:** We can search for any element in a data structure.
- **Sorting:** We can sort the elements of a data structure either in an ascending or descending order.
- **Insertion:** We can also insert the new element in a data structure.

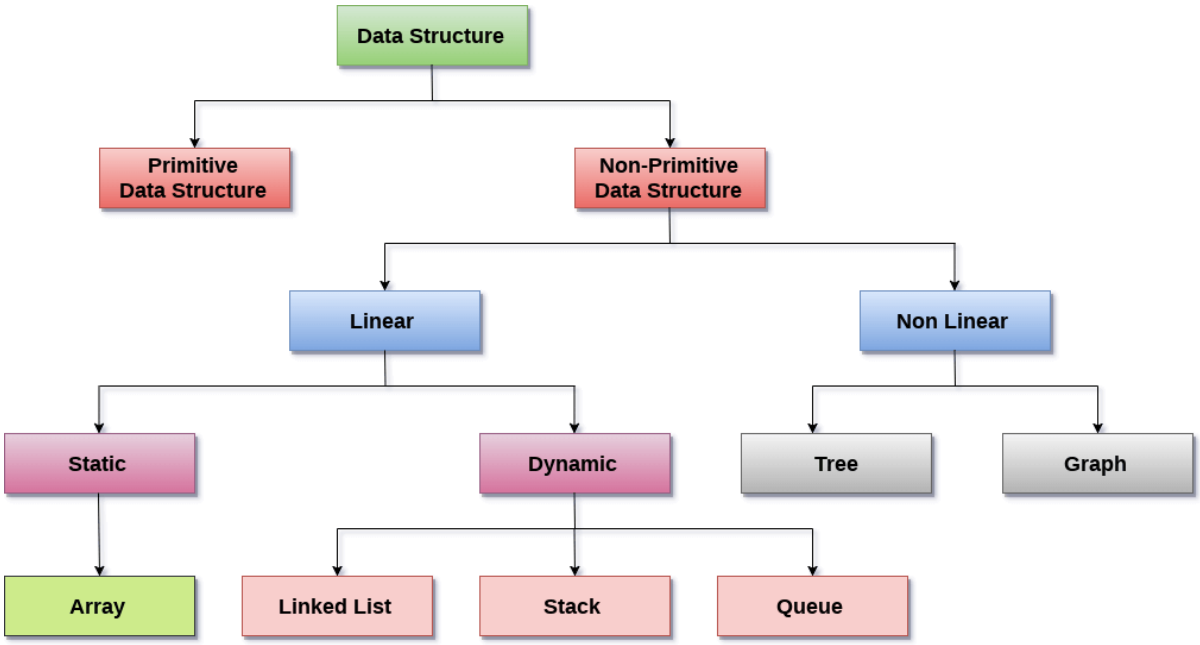
- **Updation:** We can also update the element, i.e., we can replace the element with another element.
- **Deletion:** We can also perform the delete operation to remove the element from the data structure.

Advantages of Data structures

The following are the advantages of a data structure:

- **Efficiency:** If the choice of a data structure for implementing a particular ADT is proper, it makes the program very efficient in terms of time and space.
- **Reusability:** The data structure provides reusability means that multiple client programs can use the data structure.
- **Abstraction:** The data structure specified by an ADT also 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 sethe interface.

Data Structure Classification



Linear Data Structures: A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures are given below:

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].

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:

Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the herierchy 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 children except the leaf nodes whereas each node can have atmost one parent except the root node. Trees can be classfied 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 can not have the one.

Operations on data structure

1) **Traversing:** Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

Example: If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will devide that sum by the number of subjects i.e. 6, in order to find the average.

2) **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.

If the size of data structure is **n** then we can only insert **n-1** data elements into it.

3) **Deletion:**The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure then **underflow** occurs.

4) **Searching:** The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

5) **Sorting:** The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) **Merging:** When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging

Array in Data Structure

Arrays are defined as the collection of similar types of data items stored at contiguous memory locations. It is one of the simplest data structures where each data element can be randomly accessed by using its index number.

In C programming, they are the derived data types that can store the primitive type of data such as int, char, double, float, etc. For example, if we want to store the marks of a student in 6 subjects, then we don't need to define a different variable for the marks in different subjects. Instead, we can define an array that can store the marks in each subject at the contiguous memory locations.

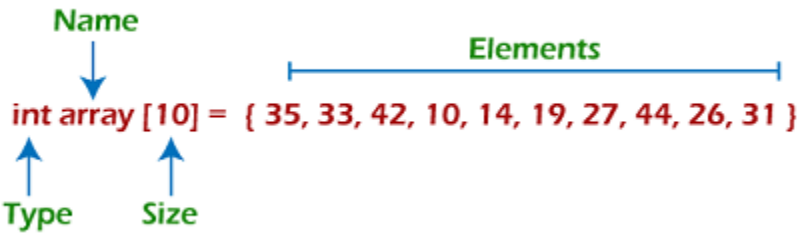
Properties of array

There are some of the properties of an array that are listed as follows -

- Each element in an array is of the same data type and carries the same size that is 4 bytes.
- Elements in the array are stored at contiguous memory locations from which the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Representation of an array

We can represent an array in various ways in different programming languages. As an illustration, let's see the declaration of array in C language -



As per the above illustration, there are some of the following important points -

- Index starts with 0.
- The array's length is 10, which means we can store 10 elements.
- Each element in the array can be accessed via its index.

Why are arrays required?

Arrays are useful because -

- Sorting and searching a value in an array is easier.
- Arrays are best to process multiple values quickly and easily.
- **Arrays are good for storing multiple values in a single variable**

Basic operations

Now, let's discuss the basic operations supported in the array -

- Traversal - This operation is used to print the elements of the array.

- Insertion - It is used to add an element at a particular index.
- Deletion - It is used to delete an element from a particular index.
- Search - It is used to search an element using the given index or by the value.
- Update - It updates an element at a particular index.

Traversal operation

This operation is performed to traverse through the array elements. It prints all array elements one after another. We can understand it with the below program –

Following is the algorithm to traverse through all the elements present in a Linear Array –

1 Start
2. Initialize an Array of certain size and datatype.
3. Initialize another variable 'i' with 0.
4. Print the ith value in the array and increment i.
5. Repeat Step 4 until the end of the array is reached.
6. End

```
1. #include <stdio.h>
2. void main() {
3.    int Arr[5] = {18, 30, 15, 70, 12};
4. int i;
5.    printf("Elements of the array are:\n");
6.    for(i = 0; i<5; i++) {
7.       printf("Arr[%d] = %d,  ", i, Arr[i]);
8.    }
9. }
```

Output

```
Elements of the array are:
Arr[0] = 18,  Arr[1] = 30,  Arr[2] = 15,  Arr[3] = 70,  Arr[4] = 12,
```

Insertion operation

This operation is performed to insert one or more elements into the array. As per the requirements, an element can be added at the beginning, end, or at any index of the array. Now, let's see the implementation of inserting an element into the array.

Following is an algorithm to insert elements into a Linear Array until we reach the end of the array −

1. Start
2. Create an Array of a desired datatype and size.
3. Initialize a variable ‘i’ as 0.
4. Enter the element at ith index of the array.
5. Increment i by 1.
6. Repeat Steps 4 & 5 until the end of the array.
7. Stop

```c
#include <stdio.h>
int main()
{
    int arr[20] = { 18, 30, 15, 70, 12 };
    int i, x, pos, n = 5;
    printf("Array elements before insertion\n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    x = 50; // element to be inserted
    pos = 4;
    n++;

    for (i = n-1; i >= pos; i--)
        arr[i] = arr[i - 1];
    arr[pos - 1] = x;
    printf("Array elements after insertion\n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
```

```
22.     return 0;
23. }
```

Output

```
Array elements before insertion
18 30 15 70 12
Array elements after insertion
18 30 15 50 70 12
```

Deletion operation

As the name implies, this operation removes an element from the array and then reorganizes all of the array elements.

Algorithm to Delete an element from an Array:

- **Step 01:** Start
- **Step 02:** [Initialize counter variable.] `Set i = pos - 1`
- **Step 03:** Repeat Step 04 and 05 for `i = pos - 1` to `i < size`
- **Step 04:** [Move i^{th} element backward (left).] set `a[i] = a[i+1]`
- **Step 05:** [Increase counter.] Set `i = i + 1`
- **Step 06:** [End of step 03 loop.]
- **Step 07:** [Reset size of the array.] set `size = size - 1`
- **Step 08:** Stop

In the above algorithm, step 2 to step 5 shifts (moves) each such element one location (position) backward (left) whose position is greater than the position of the element which we wish to delete.

The next step, Step 6, ends the loop.

And, at the last step, Step 7, decrements (reduces) the size of the array by one.

Implementation in C:

```
// writing a program in C to delete an element from an array

 void main()

{

     int i, size, pos;

     int a[] = {2, 4, 6, 8, 12};

    size = sizeof(a)/sizeof(a[0]);

     printf("The array elements before deletion operation:\n");

     for(i = 0; i < size; i++)

        printf("a[%d] = %d\n", i, a[i]);

     printf("\nEnter the position from where you wish to delete the
element: ");

     scanf("%d", &pos);

     printf("\nThe array elements after deletion operation are:\n");

     for(i = pos - 1; i < size; i++)

        a[i] = a[i+1];

     size = size - 1;

     for(i = 0; i < size; i++)

        printf("a[%d] = %d\n", i, a[i]);

}
```

If you compile and run the above program, it will produce the following result**:**

Output:

```
The array elements before deletion operation:

a[0] = 2

a[1] = 4

a[2] = 6

a[3] = 8

a[4] = 12

Enter the position from where you wish to delete the element: 3

The array elements after deletion operation are:

a[0] = 2

a[1] = 4

a[2] = 8

a[3] = 12
```

Search operation

This operation is performed to search an element in the array based on the value or index.

Algorithm to Search an element in an Array:

- **Step 01:** Start
- **Step 02:** [Initialize counter variable.] `Set i = 0`
- **Step 03:** Repeat Step 04 and 05 for `i = 0` to `i < n`
- **Step 04:** `if a[i] = x`, then jump to step 07

- **Step 05:** [Increase counter.] Set `i = i + 1`
- **Step 06:** [End of step 03 loop.]
- **Step 07:** Print `x` found at `i + 1` position and go to step 09
- **Step 08:** Print `x` not found (`if a[i] != x`, after all the iteration of the above `for` loop.)
- **Step 09:** Stop

Explanation:

To search for a given ITEM (here, the ITEM is '`x`') in an array '`a`' we compare the ITEM '`x`' with each element of the array '`a`' one by one.

That is, we test whether `a[0] = x`, and then we test whether `a[1] = x`, and so on… up till we reach to location where the ITEM '`x`' first occurs in the array '`a`'.

And, when we reach to our desired element's location which we are searching for, then we print " `x` found at `i + 1` position" and then, the iteration of the for loop breaks and jump out of the loop. After that, the program exits with successful execution.

And, If after all iteration, when we don't find our desired element then, we print "The element 'x' not found". After that, the program exits with successful execution.

Implementation in C:

```
// writing a program in C to search an element in an array


#define MAX 50
int main()
{
      int i, n, x;
      int a[MAX];
     printf("Enter the size of the array: ");
      scanf("%d", &n);
printf("\nEnter %d elements:\n", n);
      for(i = 0; i < n; i++)
         scanf("%d\n", &a[i]);
```

```
    printf("\nEnter the element to search: ");
    scanf("%d", &x);
    for(i = 0; i < n; i++)
    {
        if(a[i] == x)
        {
            printf("%d found at %d position.", x, i+1);
            return 0;
        }
    }
    printf("%d not found", x);
    return 0;
}
```

If you compile and run the above program, it will produce the following result**:**

Output:

```
# searching an element which is present in the array
 Enter the size of the array: 10
Enter 10 elements:
10
14
19
26
27
31
33
35
42
```

```
44
 Enter the element to search: 33
33 found at 7 position.
 # searching an element which is not present in the array
 Enter the size of the array: 10
Enter 10 elements:
10
14
19
26
27
31
33
35
42
44
 Enter the element to search: 25
25 not found
```

Update operation

This operation is performed to update an existing array element located at the given index.

Algorithm to Update an element in an Array:

- **Step 01:** Start
- **Step 02:** Set `a[pos-1] = x`
- **Step 03:** Stop

Implementation in C:

```
// writing a program in C to update an element in an array


void main()
{
     int i, size, x, pos;
     int a[]={1, 3, 5, 7, 9};
     size=sizeof(a)/sizeof(a[0]);
     printf("The array elements before update operation are:");
     for(i=0;i<size;i++)
        printf("\na[%d]= %d", i, a[i]);
     printf("\nThe position where you wish to update the element: ");
     scanf("%d", &pos);
     printf("The new element: ");
     scanf("%d", &x);
     a[pos-1]=x;
     printf("The array elements after update operation:");
     for(i=0;i<size;i++)
        printf("\na[%d]= %d", i, a[i]);
}
```

If you compile and run the above program, it will produce the following result:

Output:

```
The array elements before update operation are:
a[0]= 1
a[1]= 3
```

```
a[2]= 5
a[3]= 7
a[4]= 9
The position where you wish to update the element: 2
The new element: 84
The array elements after update operation:
a[0]= 1
a[1]= 84
a[2]= 5
a[3]= 7
a[4]= 9
```

Types of Arrays

Arrays in C are classified into three types:

- One-dimensional arrays
- Two-dimensional arrays
- Multi-dimensional arrays

Array Declaration

While declaring a one-dimensional array in C, the data type can be of any type, and also, we can give any name to the array, just like naming a random variable. **Syntax:**

```
int arr[5]; //arr is the array name of type integer, and 5 is the size of the array
```

Array Initialization

We can skip the writing size of the array within square brackets if we initialize array elements explicitly within the list at the time of declaration. In that case, it will pick elements list size as array size.

Example:

```
int nums[5] = {0, 1, 2, 3, 4}; //array nums is initialized with elements 0,1,2,3,4
```

If we want to initialize all elements of an integer array to zero, we could simply write:

```
int <array name>[size] = {0};
```

Array Accessing

In one-dimensional arrays in C, elements are accessed by specifying the array name and the index value within the square brackets. **Array indexing starts from 0 and ends with size-1.** If we try to access array elements out of the range, the compiler will not show any error message; rather, it will return some garbage value.

Syntax:

```
<arr_name>[index];
```

Example:

```
int nums[5] = {0, 1, 2, 3, 4};
printf("%d", nums[0]); //Array element at index 0 is printed
printf("%d", nums[-1]); //Garbage value will be printed
```

C Program to illustrate declaration, initialization, and accessing of elements of a one-dimensional array in C:

```
#include < stdio.h >

int main() {
   //declaring and initializing one-dimensional array in C
   int arr[3] = {10, 20, 30};

    // After declaration, we can also initialize the array as:
    // arr[0] = 10; arr[1] = 20; arr[2] = 30;

    for (int i = 0; i < 3; i++) {
        // accessing elements of array
        printf(" Value of arr[%d]: %d\n", i, arr[i]);
    }
}
```

Output:

```
 Value of arr[0]: 10
 Value of arr[1]: 20
 Value of arr[2]: 30
```

In this C programming code, we have initialized an array at the time of declaration with size 3 and array name as arr. At the end of the code, we are trying to print the array values by accessing its elements.

Rules for Declaring One Dimensional Array in C

- Before using and accessing, we must declare the array variable.
- In an array, indexing starts from 0 and ends at size-1. For example, if we have arr[10] of size 10, then the indexing of elements ranges from 0 to 9.
- We must include data type and variable name while declaring one-dimensional arrays in C.
- Each element of the array is stored at a contiguous memory location with a unique index number for accessing.

Initialization of One-Dimensional Array in C

After declaration, we can initialize array elements or simply initialize them explicitly at the time of declaration. One-Dimensional arrays in C are initialized either at Compile Time or Run Time.

Compile-Time Initialization

Compile-Time initialization is also known as **static-initialization**. In this, array elements are initialized when we declare the array implicitly.

Syntax:

```
<data_type> <array_name> [array_size]={list of elements};
```

Example:

```
int nums[5] = {0, 1, 2, 3, 4};
```

C Program to illustrate Compile-Time Initialization:

```
#include <stdio.h>
int main(){
    int nums[3]={0,1,2};
    printf(" Compile-Time Initialization Example:\n");
    printf(" %d ",nums[0]);
    printf("%d ",nums[1]);
    printf("%d ",nums[2]);
}
```

Output:

```
0 1 2
```

In this C program code, we have initialized an array nums of size 3 and elements as 0,1 and 2 in the list. This is **compile-time initialization**, and then at the end, we have printed all its values by accessing index-wise.

Run-Time Initialization

Runtime initialization is also known as **dynamic-initialization**. Array elements are initialized at the runtime after successfully compiling the program.

Example:

```
scanf("%d", &nums[0]); //initializing 0th index element at runtime
dynamically
```

C Program to illustrate Run-Time Initialization:

```
#include < stdio.h >

int main() {

    int nums[5];
```

```
    printf("\n Run-Time Initialization Example:\n");
    printf("\n Enter array elements: ");

    for (int i = 0; i < 5; i++) {
        scanf("%d", & nums[i]);
    }

    printf(" Accessing array elements after dynamic Initialization: ");

    for (int i = 0; i < 5; i++) {
        printf("%d ", nums[i]);
    }

    return 0;
}
```

Input

```
Run-Time Initialisation Example:

Enter array elements: 10 20 30 40 50
```

Output:

```
Accessing array elements after dynamic Initialization: 10 20 30 40 50
```

To demonstrate **runtime initialization**, we have just declared an array nums of size 5 in this C programming code. After that, within a loop, we ask the user to enter the array values to initialize them after compiling the code. In the end, we have printed its values by accessing them index-wise.

Copying One-Dimensional Arrays in C

If we have two arrays - array1 and array2, one is initialized and another array is just declared, and suppose, if we have to copy array1 elements to array2 then we can't simply just write:

```
int array1[5] = {0, 1, 2, 3, 4};
int array2[5];
array2 = array1; //This statement is wrong, it will produce an error
```

The primary condition to copy an array is that the copy array's size should be less than the original array.

Program to illustrate copying of elements of a one-dimensional array in C

```
#include < stdio.h >

int main() {
    int array1[5] = {10, 20, 30, 40, 50};
    int array2[5];
```

```
    printf("Copying One-Dimensional Arrays in C:\n");
    printf("Array1 elements: ");

    for (int i = 0; i < 5; i++) {
        printf("%d ", array1[i]);
        array2[i] = array1[i]; // Copying array1 elements to array2
    }

    printf("\nArray2 elements after copying: ");

    for (int i = 0; i < 5; i++) {
        printf("%d ", array2[i]);
    }
}
```

Output:

```
Copying One-Dimensional Arrays in C:
Array1 elements: 10 20 30 40 50
Array2 elements after copying: 10 20 30 40 50
```

In this C programming code, we have taken two arrays: array1 and array2. array1 has been initialized at the time of declaration, and to illustrate the concept of copying array elements, we are assigning array1 values to array2 within a loop. At the end, we printed the values of both arrays.

The single-dimensional array is one of the most used types of the array in C. It is a linear collection of similar types of data, and the allocated memory for all data blocks in the single-dimensional array remains consecutive.

Syntax for Declaration of Single Dimensional Array

Below is the syntax to declare the single-dimensional array.

```
data_type array_name[array_size];

//Example
int evenNumbers[5];
```

- **data_type :** is a type of data of each array block.
- **array_name :** is the name of the array using which we can refer to it.
- **array_size :** is the number of blocks of memory array going to have.

Initialization of Single Dimensional Array

In this scenario, the rest of the memory block will be filled with the default value, which is **null** for string array, 0 for integer array, etc. See the example below for further illustration.

```c
#include <stdio.h>

int main() {
  int arr[5] = {7, 8};

  for (int i = 0; i < 5; i++) {
    printf("%d ", arr[i]);
  }
  return 0;
}
```

Output :

```
7 8 0 0 0
```

Accessing Elements of Single Dimensional Array

We can access the element of the single-dimensional array by providing the index of the element with the array name. The index of the array starts with zero.

```
arrayName[index];
```

Example

C Program to Enter 5 Numbers and Print them in Reverse Order.

```c
#include <stdio.h>

int main() {
  int arr[5] = {7, 8, 1, 10, 6}; // one-dimensional array

  for (int i = 4; i >= 0; i--) {
    printf("%d ", arr[i]); // prints array element
  }
  return 0;
```

```
}
```

Output :

```
6 10 1 8 7
```

Explanation :

- First of all, we have created an array of size 5 with some initialized values.
- Later, we are traversing on that array from the last index to the first index, which is 0.

Two-Dimensional Arrays

Two Dimensional Arrays can be thought of as an array of arrays, or as a matrix consisting of rows and columns.

Following is an example of a 2D array:

1	**7**	**2**
9	**3**	**8**

This array has 22 rows and 33 columns.

Two dimensional array in C, will follow zero-based indexing, like all other arrays in C.

Declaration of two dimensional Array in C

Now we will go over the syntax for declaring a Two-dimensional array in C.

```
data_type array_name[i][j]
```

Here i and j are the size of the two dimensions, i.e., i denotes the number of rows while j denotes the number of columns.

Example:

```
int A[10][20];
```

Here we declare a two-dimensional array in C, named A, which has 10 rows and 20 columns.

Example

```
int A[2][3] = {{3, 2, 1}, {8, 9, 10}};
```

Two-dimensional array example in C

Consider a scenario where you want to store the scores of students in a small classroom. You have 4 students, and each student has taken 3 tests. This data can be efficiently represented using a 2D array.

Let's declare and initialize a 2D integer array for this purpose:

```c
#include <stdio.h>

int main() {
    int scores[4][3] = {
        {95, 87, 91},    // Scores for Student 1
        {88, 76, 93},    // Scores for Student 2
        {78, 85, 89},    // Scores for Student 3
        {92, 88, 94}     // Scores for Student 4
    };

    // Access and print the scores
    printf("Student Scores:\n");
    for (int student = 0; student < 4; student++) {
        printf("Student %d: ", student + 1);
        for (int test = 0; test < 3; test++) {
            printf("%d ", scores[student][test]);
        }
        printf("\n");
    }

    return 0;
}
```

In this example:

- We declare a 2D integer array called scores with dimensions [4][3]. This represents 4 students and 3 test scores for each student.
- We initialize the array with the scores for each student using nested braces.
- We then use nested loops to access and print the scores for each student and each test.

When you run this program, it will output:

```
Student Scores:
Student 1: 95 87 91
Student 2: 88 76 93
Student 3: 78 85 89
Student 4: 92 88 94
```

Accessing the Elements of a 2D Array

Example

Suppose we have a 2D array representing a 3x3 grid, and we want to access and print the values of its elements.

Consider the following 2D array:

```c
#include <stdio.h>

int main() {
    int grid[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    // Access and display individual elements
    printf("Element at row 2, column 1: %d\n", grid[2][1]); // Accessing
element at row 2, column 1

    return 0;
}
```

Output

```
Element at row 2, column 1: 8
```

Changing Elements in a 2D Array

Example

Suppose we have a 2D array representing a 3x3 grid, and we want to change the value of an element at a specific row and column in the grid.

Consider the following 2D array:

```c
#include <stdio.h>

int main() {
    int grid[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    // Display the original grid
    printf("Original Grid:\n");
    for (int row = 0; row < 3; row++) {
        for (int col = 0; col < 3; col++) {
            printf("%2d ", grid[row][col]);
        }
        printf("\n");
    }

    // Change the element at row 2, column 1 (0-based indexing)
    grid[2][1] = 42;

    // Display the modified grid
    printf("\nModified Grid:\n");
    for (int row = 0; row < 3; row++) {
        for (int col = 0; col < 3; col++) {
            printf("%2d ", grid[row][col]);
        }
        printf("\n");
    }

    return 0;
}
```

Output

```
Original Grid:
 1  2  3
 4  5  6
 7  8  9

Modified Grid:
 1  2  3
 4  5  6
 7 42  9
```

As you can see, we successfully changed the value of an element in the 2D array.

UNIT 1

Data Type

A *data type* is the most basic and the most common classification of data. It is this through which the compiler gets to know the form or the type of information that will be used throughout the code. So basically data type is a type of information transmitted between the programmer and the compiler where the programmer informs the compiler about what type of data is to be stored and also tells how much space it requires in the memory. Some basic examples are int, string etc. It is the type of any variable used in the code.

- CPP

```
#include <iostream.h>

using namespace std;



void main()

{

    int a;

    a = 5;



    float b;

    b = 5.0;
```

```
    char c;

    c = 'A';



    char d[10];

    d = "example";

}
```

As seen from the theory explained above we come to know that in the above code, the variable ‘a’ is of data type integer which is denoted by int a. So the variable ‘a’ will be used as an integer type variable throughout the process of the code. And, in the same way, the variables ‘b’, ‘c’ and ‘d’ are of type float, character and string respectively. And all these are kinds of data types.

Data Structure

A *data structure* is a collection of different forms and different types of data that has a set of specific operations that can be performed. It is a collection of data types. It is a way of organizing the items in terms of memory, and also the way of accessing each item through some defined logic. Some examples of data structures are stacks, queues, linked lists, binary tree and many more. Data structures perform some special operations only like insertion, deletion and traversal. For example, you have to store data for many employees where each employee has his name, employee id and a mobile number. So this kind of data requires complex data management, which means it requires data structure comprised of multiple primitive data types. So data structures are one of the most important aspects when implementing coding concepts in real-world applications.

Difference between data type and data structure:

Data Types	Data Structures
Data Type is the kind or form of a variable which is being used throughout the program. It defines that the particular variable will assign the values of the given data type only	Data Structure is the collection of different kinds of data. That entire data can be represented using an object and can be used throughout the entire program.
Implementation through Data Types is a form of abstract implementation	Implementation through Data Structures is called concrete implementation
Can hold values and not data, so it is data less	Can hold different kind and types of data within one single object
Values can directly be assigned to the data type variables	The data is assigned to the data structure object using some set of algorithms and operations like push, pop and so on.
No problem of time complexity	Time complexity comes into play when working with data structures
Examples: int, float, double	Examples: stacks, queues, tree

What is Abstract Data Type?

An Abstract Data Type (ADT) is a programming concept that defines a high-level view of a data structure, without specifying the implementation details. In other words, it is a blueprint for creating a data structure that defines the behavior and interface of the structure, without specifying how it is implemented.

An ADT in the data structure can be thought of as a set of operations that can be performed on a set of values. This set of operations actually defines the behavior of the data structure, and they are used to manipulate the data in a way that suits the needs of the program.

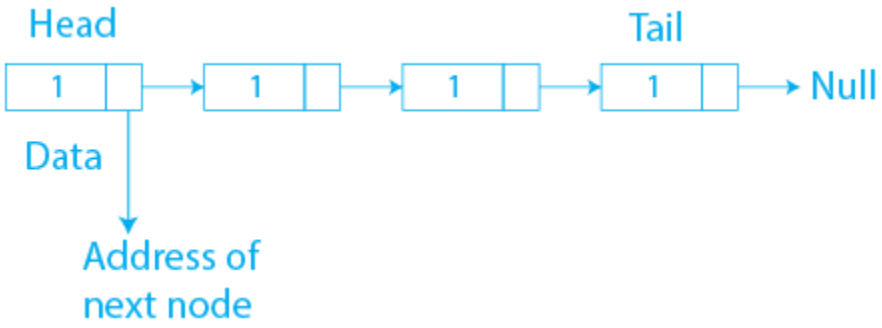
This makes it easier for programmers to reason about the data structure, and to use it correctly in their programs.

Examples of abstract data type in data structures are List, Stack, Queue, etc.

Abstract Data Type Model

List ADT

Lists are linear data structures that hold data in a non-continuous structure. The list is made up of data storage containers known as "nodes." These nodes are linked to one another, which means that each node contains the address of another block. All of the nodes are thus connected to one another via these links. You can discover more about lists in this article: Linked List Data Structure.

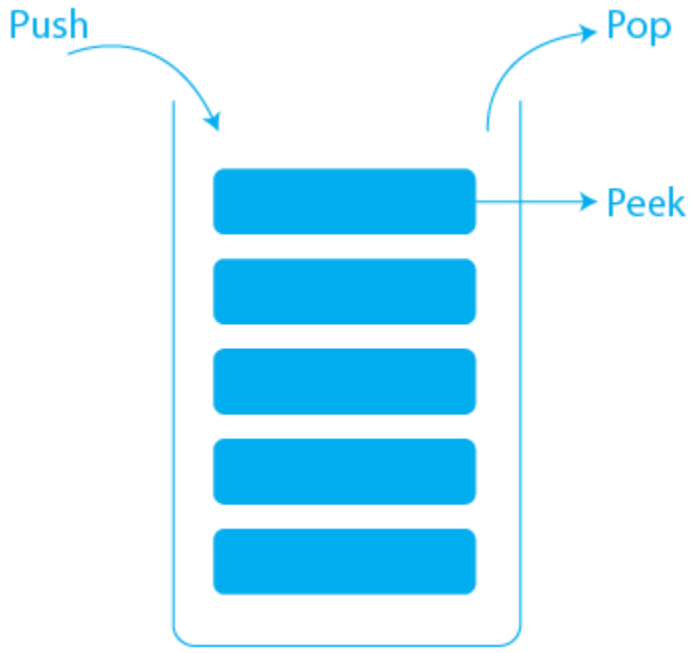


Some of the most essential operations defined in List ADT are listed below.

- **front():** returns the value of the node present at the front of the list.
- **back():** returns the value of the node present at the back of the list.
- **push_front(int val):** creates a pointer with value = val and keeps this pointer to the front of the linked list.
- **push_back(int val):** creates a pointer with value = val and keeps this pointer to the back of the linked list.
- **pop_front():** removes the front node from the list.
- **pop_back():** removes the last node from the list.
- **empty():** returns true if the list is empty, otherwise returns false.
- **size():** returns the number of nodes that are present in the list.

Stack ADT

A stack is a linear data structure that only allows data to be accessed from the top. It simply has two operations: push (to insert data to the top of the stack) and pop (to remove data from the stack). (used to remove data from the stack top).



Some of the most essential operations defined in Stack ADT are listed below.

- **top():** returns the value of the node present at the top of the stack.
- **push(int val):** creates a node with value = val and puts it at the stack top.
- **pop():** removes the node from the top of the stack.
- **empty():** returns true if the stack is empty, otherwise returns false.
- **size():** returns the number of nodes that are present in the stack.

Queue ADT

A queue is a linear data structure that allows data to be accessed from both ends. There are two main operations in the queue: push (this operation inserts data to the back of the queue) and pop (this operation is used to remove data from the front of the queue).

Some of the most essential operations defined in Queue ADT are listed below.

- **front():** returns the value of the node present at the front of the queue.
- **back():** returns the value of the node present at the back of the queue.
- **push(int val):** creates a node with value = val and puts it at the front of the queue.
- **pop():** removes the node from the rear of the queue.
- **empty():** returns true if the queue is empty, otherwise returns false.
- **size():** returns the number of nodes that are present in the queue.

Advantages of ADT in Data Structures

The advantages of ADT in Data Structures are:

- Provides abstraction, which simplifies the complexity of the data structure and allows users to focus on the functionality.
- Enhances program modularity by allowing the data structure implementation to be separate from the rest of the program.
- Enables code reusability as the same data structure can be used in multiple programs with the same interface.
- Promotes the concept of data hiding by encapsulating data and operations into a single unit, which enhances security and control over the data.

What is algorithm and why analysis of it is important?

In the analysis of the algorithm, it generally focused on CPU (time) usage, Memory usage, Disk usage, and Network usage. All are important, but the most concern is about the CPU time.

- **Performance:** How much time/memory/disk/etc. is used when a program is run. This depends on the machine, compiler, etc. as well as the code we write.
- **Complexity:** How do the resource requirements of a program or algorithm scale, i.e. what happens as the size of the problem being solved by the code gets larger.

Algorithm Analysis:
Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Types of Algorithm Analysis:

1. Best case
2. Worst case

3. Average case

- **Best case:** Define the input for which algorithm takes less time or minimum time. In the best case calculate the lower bound of an algorithm. Example: In the linear search when search data is present at the first location of large data then the best case occurs.
- **Worst Case**: Define the input for which algorithm takes a long time or maximum time. In the worst calculate the upper bound of an algorithm. Example: In the linear search when search data is not present at all then the worst case occurs.
- **Average case**: In the average case take all random inputs and calculate the computation time for all inputs.
  And then we divide it by the total number of inputs.
  Average case = all random case time / total no of case

What is Time Complexity?

In Computer science, there are various problems and several ways to solve each of these problems using different algorithms. These algorithms may have varied approaches, some might be too complex to Implement while some may solve the problem in a lot simpler way than others. It is hard to select a suitable and efficient algorithm out of all that are available. To make the selection of the best algorithm easy, calculation of complexity and time consumption of an algorithm is important this is why **time complexity analysis** is important, for this **asymptotic analysis** of the algorithm is done.
There are three cases denoted by three different notations of analysis:

- **Big-oh(O) Notation:** Denotes the upper bound of any algorithm's runtime i.e. time is taken by the algorithm in the worst case.
- **Big-omega(Ω) Notation:** Denotes the best runtime of an algorithm.
- **Big-Theta(Θ) notation:** Denotes average case time complexity.

Time Complexity: The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.

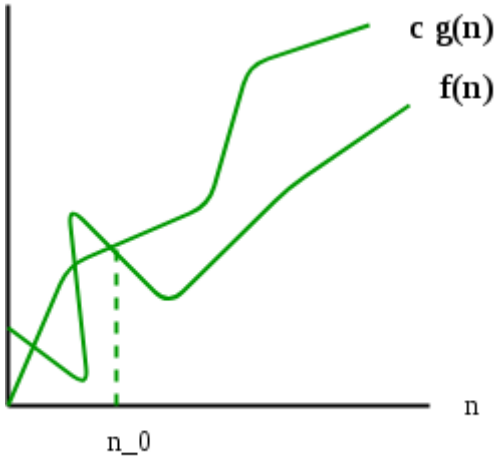
1. Big-O Notation (O-notation):

Big-O notation represents the upper bound of the running time of an algorithm. Therefore, it gives the worst-case complexity of an algorithm.

.It is the most widely used notation for Asymptotic analysis.
.It specifies the upper bound of a function.
.The maximum time required by an algorithm or the worst-case time complexity.
.It returns the highest possible output value(big-O) for a given input.

If f(n) describes the running time of an algorithm, f(n) is O(g(n)) if there exist a positive constant C and n0 such that, $0 \leq f(n) \leq cg(n)$ for all $n \geq n0$

It returns the highest possible output value (big-O)for a given input.
The execution time serves as an upper bound on the algorithm's time complexity.



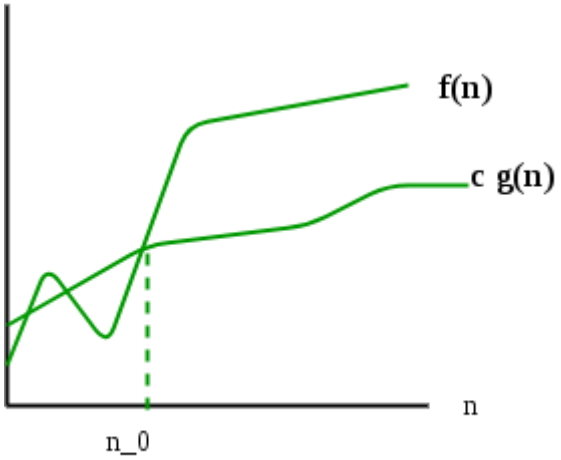
$f(n) = O(g(n))$

The Big-O notation is useful when we only have an upper bound on the time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

2. Omega Notation (Ω-Notation):

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

The execution time serves as a lower bound on the algorithm's time complexity.
It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time.
Let g and f be the function from the set of natural numbers to itself. The function f is said to be Ω(g), if there is a constant $c > 0$ and a natural number n0 such that $c*g(n) \leq f(n)$ for all $n \geq n0$



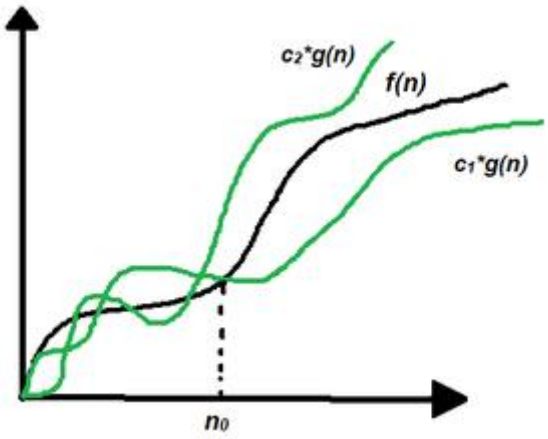
f(n) = Omega(g(n))

Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as Ω(n), but it is not very useful information about insertion sort, as we are generally interested in worst-case and sometimes in the average case.

3.Theta Notation (Θ-Notation):

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the ***average-case*** *complexity of an algorithm.*
.Theta (Average Case) You add the running times for each possible input combination and take the average in the average case.

Let g and f be the function from the set of natural numbers to itself. The function f is said to be Θ(g), if there are constants c1, c2 > 0 and a natural number n0 such that c1* g(n) ≤ f(n) ≤ c2 * g(n) for all n ≥ n0



Theta notation

The above expression can be described as if f(n) is theta of g(n), then the value f(n) is always between c1 * g(n) and c2 * g(n) for large values of n (n ≥ n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.

The execution time serves as both a lower and upper bound on the algorithm's time complexity.

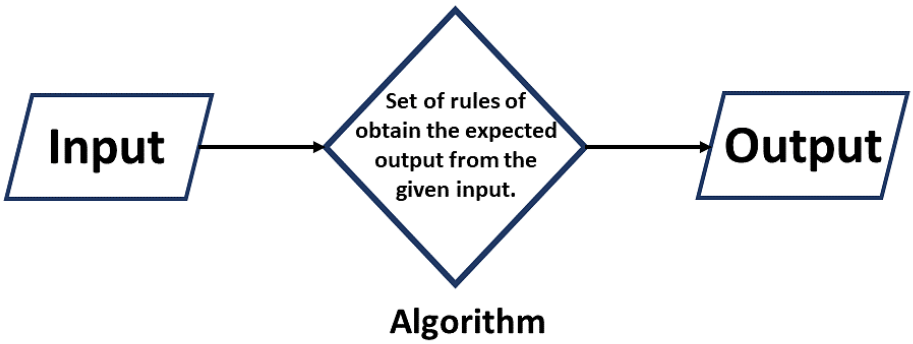
What Is An Algorithm

An algorithm is a step-by-step procedure that defines a set of instructions that must be carried out in a specific order to produce the desired result. Algorithms are generally developed independently of underlying languages, which means that an algorithm can be implemented in more than one programming language. Unambiguity, fineness,

effectiveness, and language independence are some of the characteristics of an algorithm.

An algorithm is a set of commands that must be followed for a computer to perform calculations or other problem-solving operations.

- According to its formal definition, an algorithm is a finite set of instructions carried out in a specific order to perform a particular task.
- It is not the entire program or code; it is simple logic to a problem represented as an informal description in the form of a flowchart.

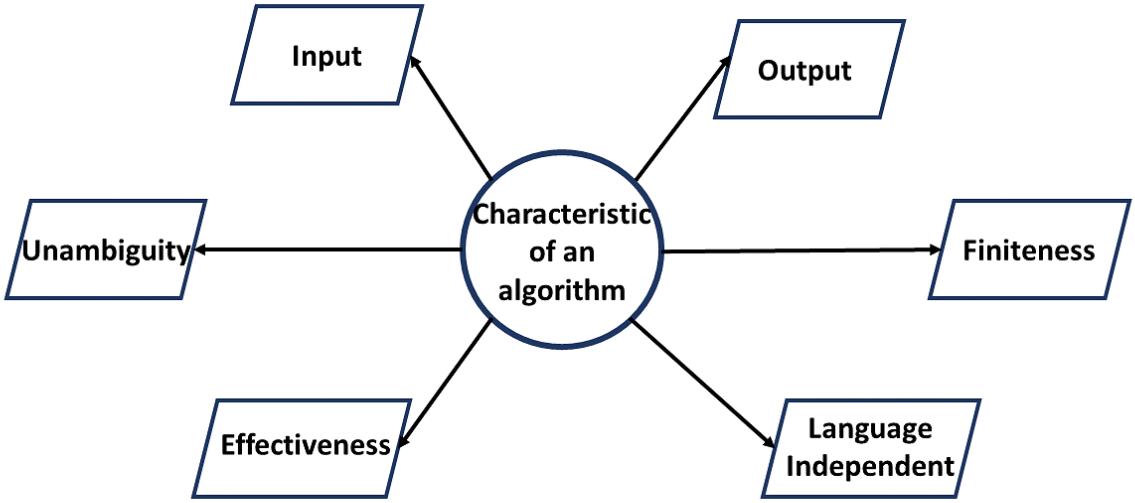


Algorithm

- Problem: A problem can be defined as a real-world problem or real-world instance problem for which you need to develop a program or set of instructions. An algorithm is a set of instructions.
- Algorithm: An algorithm is defined as a step-by-step process that will be designed for a problem.
- Input: After designing an algorithm, the algorithm is given the necessary and desired inputs.
- Processing unit: The input will be passed to the processing unit, producing the desired output.
- Output: The outcome or result of the program is referred to as the output.

Characteristics of an Algorithm

An algorithm has the following characteristics:



- Input: An algorithm requires some input values. An algorithm can be given a value other than 0 as input.
- Output: At the end of an algorithm, you will have one or more outcomes.
- Unambiguity: A perfect algorithm is defined as unambiguous, which means that its instructions should be clear and straightforward.
- Finiteness: An algorithm must be finite. Finiteness in this context means that the algorithm should have a limited number of instructions, i.e., the instructions should be countable.

- Effectiveness: Because each instruction in an algorithm affects the overall process, it should be adequate.
- Language independence: An algorithm must be language-independent, which means that its instructions can be implemented in any language and produce the same results.

How to Write an Algorithm?

- There are no well-defined standards for writing algorithms. It is, however, a problem that is resource-dependent. Algorithms are never written with a specific programming language in mind.
- As you all know, basic code constructs such as loops like do, for, while, all programming languages share flow control such as if-else, and so on. An algorithm can be written using these common constructs.

Example

Now, use an example to learn how to write algorithms.

Problem: Create an algorithm that multiplies two numbers and displays the output.

Step 1 – Start

Step 2 − declare three integers x, y & z

Step 3 − define values of x & y

Step 4 − multiply values of x & y

Step 5 − store result of step 4 to z

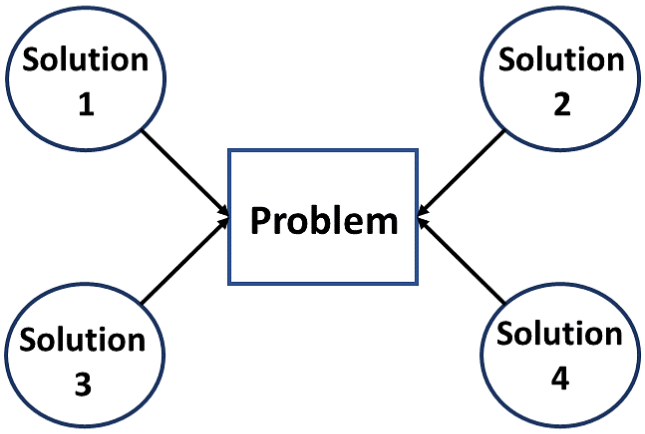
Step 6 − print z

```
Step 7 – Stop
```

Algorithms instruct programmers on how to write code. In addition, the algorithm can be written as:

```
Step 1 − Start mul

Step 2 − get values of x & y

Step 3 − z ← x * y

Step 4 − display z

Step 5 – Stop
```

In algorithm design and analysis, the second method is typically used to describe an algorithm. It allows the analyst to analyze the algorithm while ignoring all unwanted definitions easily. They can see which operations are being used and how the process is progressing. It is optional to write step numbers. To solve a given problem, you create an algorithm. A problem can be solved in a variety of ways.

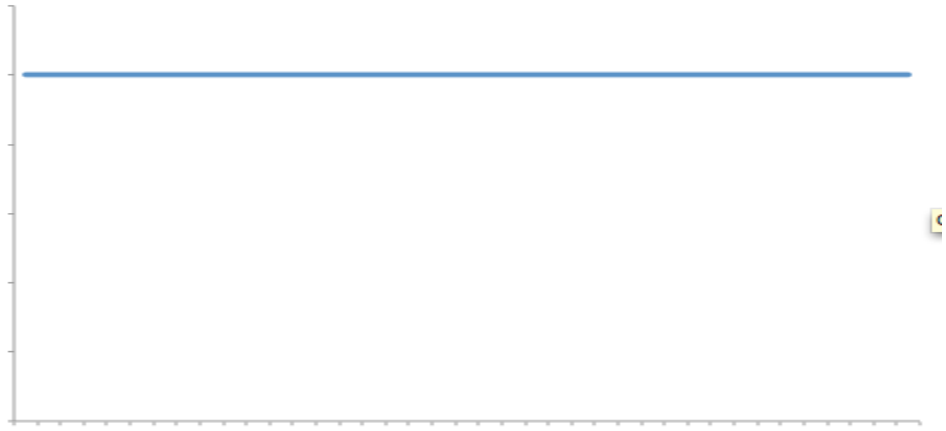

Solution 1
Solution 2
Problem
Solution 3
Solution 4

Growth Rates

Algorithms analysis is all about understanding growth rates. That is as the amount of data gets bigger, how much more resource will my algorithm require? Typically, we describe the resource growth rate of a piece of code in terms of a function. To help understand the implications, this section will look at graphs for different growth rates from most efficent to least efficient.

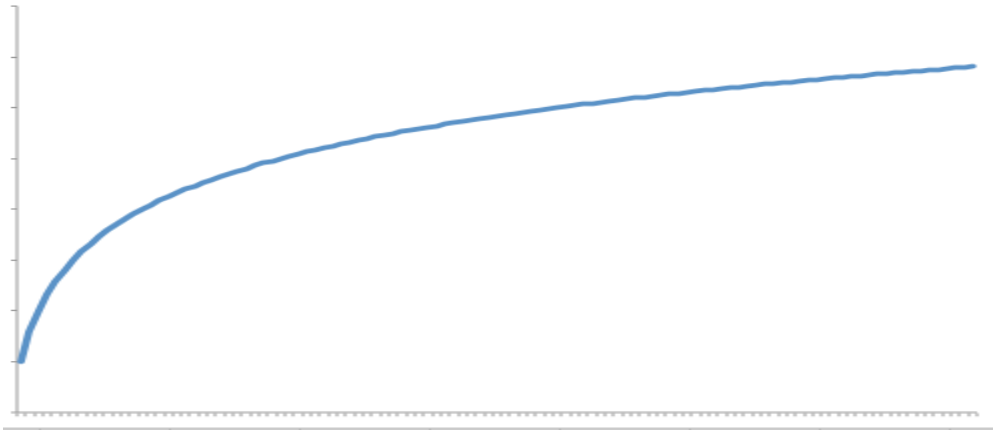
Constant Growth Rate

A constant resource need is one where the resource need does not grow. That is processing 1 piece of data takes the same amount of resource as processing 1 million pieces of data. The graph of such a growth rate looks like a horizontal line



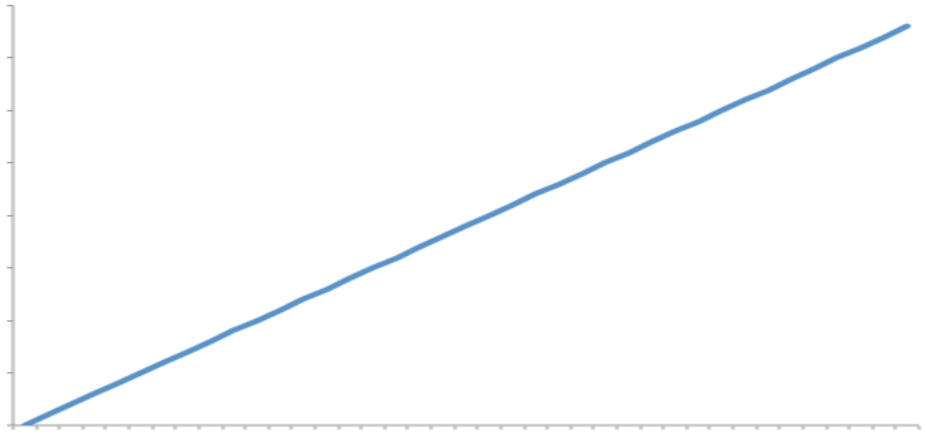
Logrithmic Growth Rate

A logrithmic growth rate is a growth rate where the resource needs grows by one unit each time the data is doubled. This effectively means that as the amount of data gets bigger, the curve describing the growth rate gets flatter (closer to horizontal but never reaching it). The following graph shows what a curve of this nature would look like.



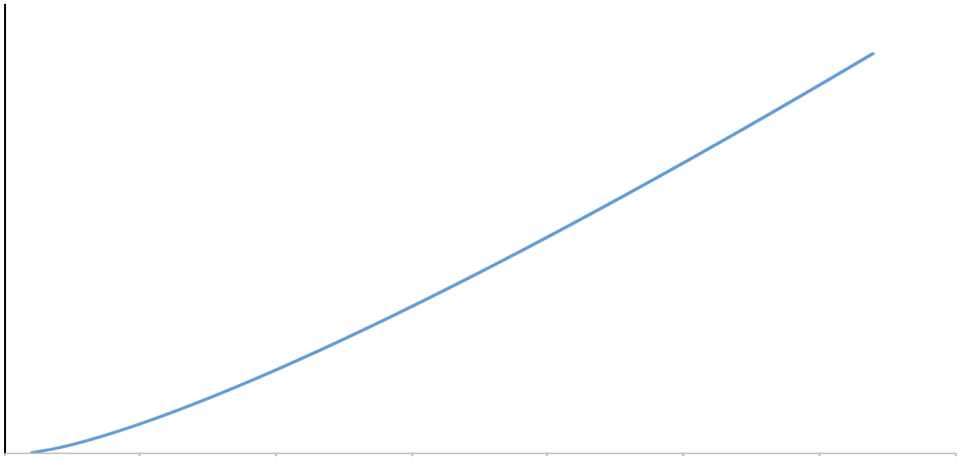
Linear Growth Rate

A linear growth rate is a growth rate where the resource needs and the amount of data is directly proportional to each other. That is the growth rate can be described as a straight line that is not horizontal.



Log Linear

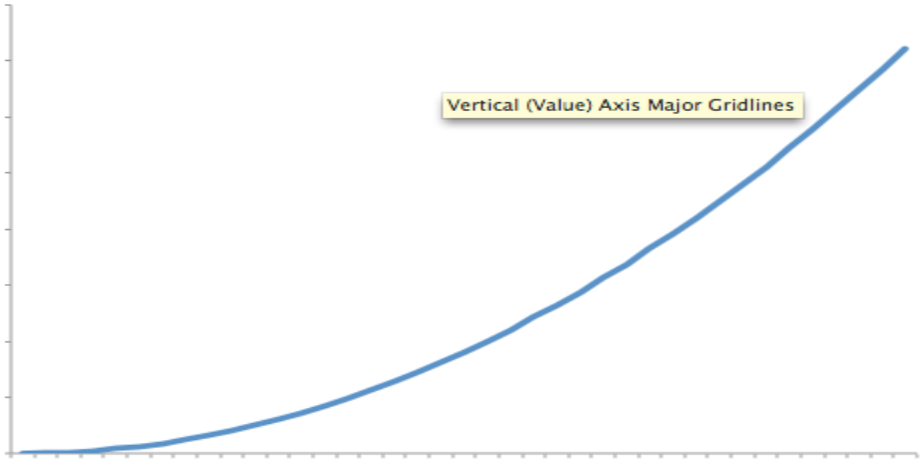
A loglinear growth rate is a slightly curved line. the curve is more pronounced for lower values



than higher ones

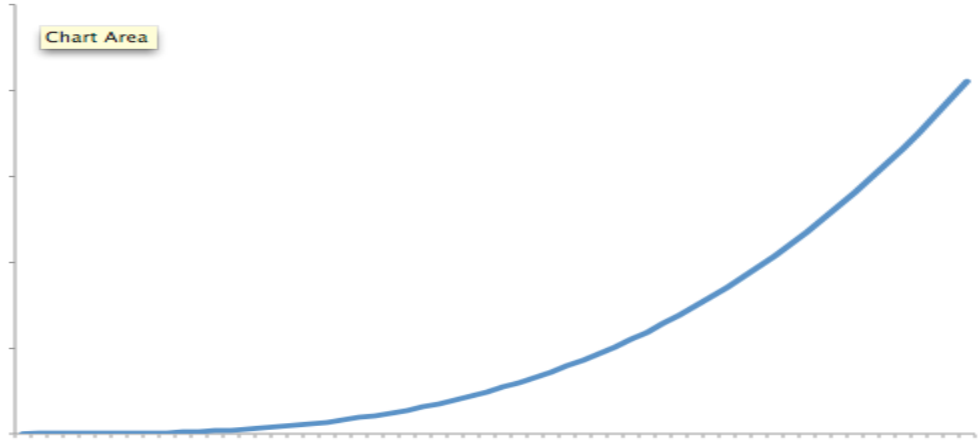
Quadratic Growth Rate

A quadratic growth rate is one that can be described by a parabola.



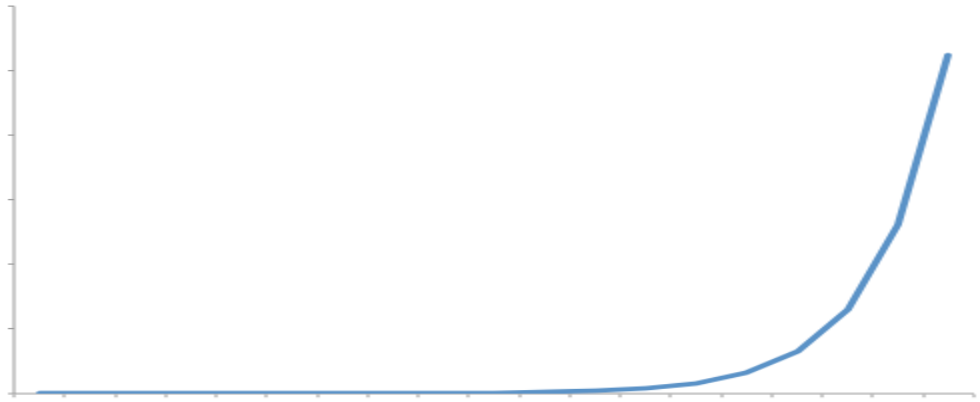
Cubic Growth Rate

While this may look very similar to the quadratic curve, it grows significantly faster



Exponential Growth Rate

An exponential growth rate is one where each extra unit of data requires a doubling of resource. As you can see the growth rate starts off looking like it is flat but quickly shoots up to near vertical (note that it can't actually be vertical)



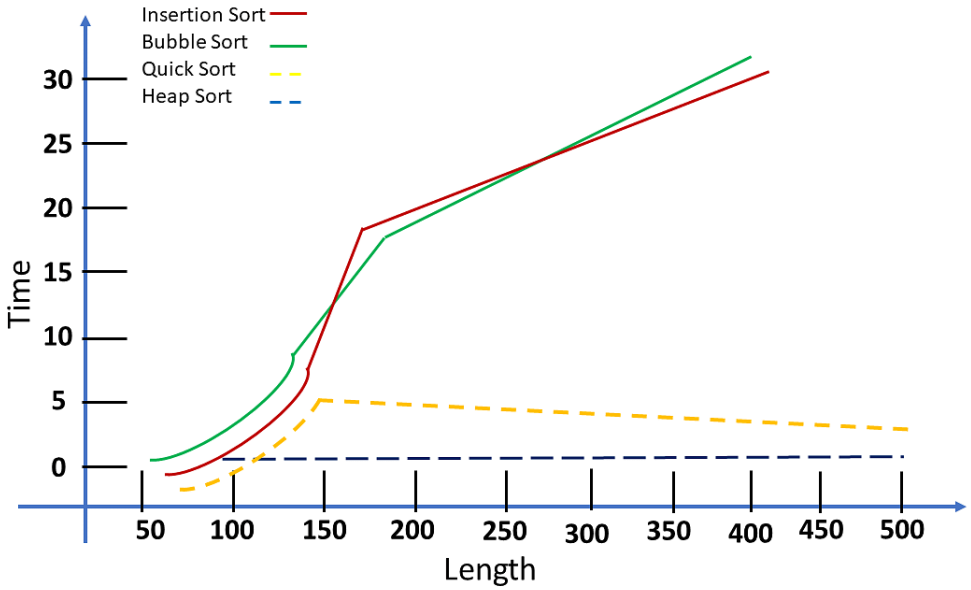
Time Complexity

The input size has a strong relationship with time complexity. As the size of the input increases, so does the runtime, or the amount of time it takes the algorithm to run.

Here is an example.

Assume you have a set of numbers S= (10, 50, 20, 15, 30)

There are numerous algorithms for sorting the given numbers. However, not all of them are effective. To determine which is the most effective, you must perform computational analysis on each algorithm.



Here are some of the most critical findings from the graph:

- This test revealed the following sorting algorithms: Quicksort, Insertion sort, Bubble sort.
- Python is the programming language used to complete the task, and the input size ranges from 50 to 500 characters.
- Insertion sort and Bubble sort algorithms performed far worse, significantly increasing computing time." See the graph above for the results.
- Before you can run an analysis on any algorithm, you must first determine its stability. Understanding your data is the most important aspect of conducting a successful analysis.

Significant in Terms of Space Complexity

Space complexity refers to the total amount of memory space used by an algorithm/program, including the space of input values for execution. Calculate the space occupied by variables in an algorithm/program to determine space complexity.

However, people frequently confuse Space-complexity with auxiliary space. Auxiliary space is simply extra or temporary space, and it is not the same as space complexity. To put it another way,

Auxiliary space + space use by input values = Space Complexity

The best algorithm/program should have a low level of space complexity. The less space required, the faster it executes.

Method for Calculating Space and Time Complexity

Methods for Calculating Time Complexity

To calculate time complexity, you must consider each line of code in the program. Consider the multiplication function as an example. Now, calculate the time complexity of the multiply function:

```
1. mul <- 1
2. i <- 1
3. While i <= n do
4.     mul = mul * 1
5.     i = i + 1
6. End while
```

Let T(n) be a function of the algorithm's time complexity. Lines 1 and 2 have a time complexity of O. (1). Line 3 represents a loop. As a result, you must repeat lines 4 and 5 (n -1) times. As a result, the time complexity of lines 4 and 5 is O. (n).

Finally, adding the time complexity of all the lines yields the overall time complexity of the multiple function fT(n) = O(n).

The iterative method gets its name because it calculates an iterative algorithm's time complexity by parsing it line by line and adding the complexity.

Aside from the iterative method, several other concepts are used in various cases. The recursive process, for example, is an excellent way to calculate time complexity for recurrent solutions that use recursive trees or substitutions. The master's theorem is another popular method for calculating time complexity.

Methods for Calculating Space Complexity

With an example, you will go over how to calculate space complexity in this section. Here is an example of computing the multiplication of array elements:

```
1. int mul, I
2. While i < = n do
3.     mul <- mul * array[i]
4.     i <- i + 1
5. end while
6. return mul
```

Let S(n) denote the algorithm's space complexity. In most systems, an integer occupies 4 bytes of memory. As a result, the number of allocated bytes would be the space complexity.

Line 1 allocates memory space for two integers, resulting in S(n) = 4 bytes multiplied by 2 = 8 bytes. Line 2 represents a loop. Lines 3 and 4 assign a value to an already existing variable. As a result, there is no need to set aside any space. The return statement in line 6 will allocate one more memory case. As a result, S(n)= 4 times 2 + 4 = 12 bytes.

Because the array is used in the algorithm to allocate n cases of integers, the final space complexity will be fS(n) = n + 12 = O (n).

As you progress through this tutorial, you will see some differences between space and time complexity.