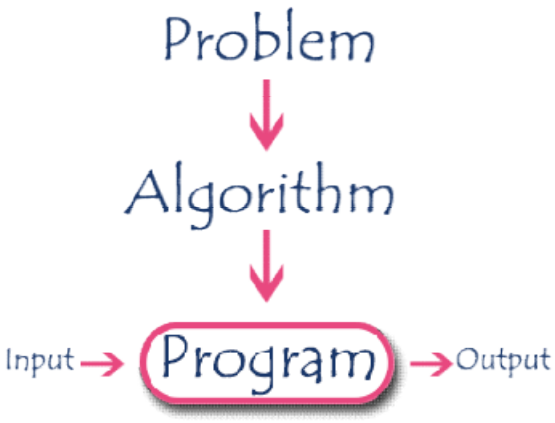
Introduction to Algorithms

What is an algorithm?

An algorithm is a step by step procedure to solve a problem. In normal language, algorithm is defined as a sequence of statements which are used to perform a task. In computer science, an algorithm can be defined as follows...

An algorithm is a sequence of unambiguous instructions used for solving a problem, which can be implemented (as a program) on a computer.

Algorithms are used to convert our problem solution into step by step statements. These statements can be converted into computer programming instructions which forms a program. This program is executed by computer to produce solution. Here, program takes required data as input, processes data according to the program instructions and finally produces result as shown in the following picture.



Specifications of Algorithms

Every algorithm must satisfy the following specifications...

1. **Input -** Every algorithm must take zero or more number of input values from external.
2. **Output -** Every algorithm must produce an output as result.
3. **Definiteness -** Every statement/instruction in an algorithm must be clear and unambiguous (only one interpretation).
4. **Finiteness -** For all different cases, the algorithm must produce result within a finite number of steps.
5. **Effectiveness -** Every instruction must be basic enough to be carried out and it also must be feasible.

Example for an Algorithm

Let us consider the following problem for finding the largest value in a given list of values.
Problem Statement : Find the largest number in the given list of numbers?
Input : A list of positive integer numbers. (List must contain at least one number).
Output : The largest number in the given list of positive integer numbers.

Consider the given list of numbers as 'L' (input), and the largest number as 'max' (Output).

Algorithm

1. **Step 1:** Define a variable 'max' and initialize with '0'.
2. **Step 2:** Compare first number (say 'x') in the list 'L' with 'max', if 'x' is larger than 'max', set 'max' to 'x'.
3. **Step 3:** Repeat step 2 for all numbers in the list 'L'.
4. **Step 4:** Display the value of 'max' as a result.

Write an algorithm to find all roots of a quadratic equation $ax^2+bx+c=0$.

```
Step 1: Start
Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;
Step 3: Calculate discriminant
         D←b2-4ac
Step 4: If D≥0
              r1←(-b+√D)/2a
              r2←(-b-√D)/2a
              Display r1 and r2 as roots.
```

```
        Else
              Calculate real part and imaginary part
              rp←b/2a
              ip←√(-D)/2a
              Display rp+j(ip) and rp-j(ip) as roots
Step 5: Stop
5: Stop
```

Max-Min Problem

Let us consider a simple problem that can be solved by divide and conquer technique.

Problem Statement

The Max-Min Problem in algorithm analysis is finding the maximum and minimum value in an array.

Solution

To find the maximum and minimum numbers in a given array ***numbers[]*** of size **n**, the following algorithm can be used. First we are representing the **naive method** and then we will present **divide and conquer approach**.

Naïve Method

Naïve method is a basic method to solve any problem. In this method, the maximum and minimum number can be found separately. To find the maximum and minimum numbers, the following straightforward algorithm can be used.

```
Algorithm: Max-Min-Element (numbers[])
max := numbers[1]
min := numbers[1]

for i = 2 to n do
```

```
   if numbers[i] > max then
      max := numbers[i]
   if numbers[i] < min then
      min := numbers[i]
return (max, min)
```

Analysis

The number of comparison in Naive method is **2n - 2**.

The number of comparisons can be reduced using the divide and conquer approach. Following is the technique.

Divide and Conquer Approach

In this approach, the array is divided into two halves. Then using recursive approach maximum and minimum numbers in each halves are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half.

In this given problem, the number of elements in an array is $y-x+1$, where **y** is greater than or equal to **x**.
$Max-Min(x,y)$ will return the maximum and minimum values of an array $numbers[x...y]$.

```
Algorithm: Max - Min(x, y)
if y - x ≤ 1 then
   return (max(numbers[x], numbers[y]), min((numbers[x], numbers[y]))
else
   (max1, min1):= maxmin(x, ⌊((x + y)/2)⌋)
   (max2, min2):= maxmin(⌊((x + y)/2) + 1)⌋,y)
return (max(max1, max2), min(min1, min2))
```

Analysis

Let ***T(n)*** be the number of comparisons made by $Max-Min(x,y)$, where the number of elements $n=y-x+1$.

If ***T(n)*** represents the numbers, then the recurrence relation can be represented as

$$T(n)=\begin{cases} T(\lfloor \frac{n}{2} \rfloor)+T(\lceil \frac{n}{2} \rceil)+2 & \text{for } n>2 \\ 0 & \text{for } n=2 \\ 1 & \text{for } n=1 \end{cases}$$

Let us assume that ***n*** is in the form of power of **2**. Hence, **n = 2^k** where **k** is height of the recursion tree.

So,

$$T(n)=2.T(\frac{n}{2})+2=2.(2.T(\frac{n}{4})+2)+2.....=3\frac{n}{2}-2$$

Compared to Naïve method, in divide and conquer approach, the number of comparisons is less. However, using the asymptotic notation both of the approaches are represented by **O(n)**.

Algorithm to check whether the given number is Prime or not

Aim:

Write a C program to check whether the given number is prime or not.

Algorithm:

```
Step 1: Start
Step 2: Read number n
Step 3: Set f=0
Step 4: For i=2 to n-1
Step 5: If n mod i==0 then
Step 6: Set f=1 and break
Step 7: Loop
Step 8: If f=0 then
print 'The given number is prime'
else
print 'The given number is not prime'
```

Step 9: Stop

Program code

```
#include<stdio.h>
#include<conio.h>
void main( )
{
clrscr();
int n,i,f=0;
printf("Enter the number: ");
scanf("%d",&n);
for(i=2;i<n;i++)
{
if(n%i==0)
{
f=1;
break;
}
}
if(f==0)
printf("The given number is prime");
else
printf("The given number is not prime");
getch();
}
```

Output

```
Enter the number : 5
The given number is prime
```

Linear Search Algorithm
(Sequential Search)

What is Search?

Searching is a process of finding a value in a list of values. In other words, searching is the process of locating the given value position in a list of values.

Searching is the process of finding the given element in a list of elements.

Linear search algorithm finds the given element in a list of elements with **O(n)** time complexity where **n** is the total number of elements in the list. This search process starts comparing the search element with the first element in the list. If both are matched then the result is **"element found"** otherwise the search element is compared with the next element in the list. If both are matched, then the result is **"element found"**. Otherwise, repeat the same with the next element in the list until the search element is compared with the last element in the list. If that last element also doesn't match with the search element, then the result we get is **"Element not found in the list"**. That means, the search element is compared with all the elements in the list sequentially until the match found.

Linear search is implemented using following steps...

- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the first element in the list.
- **Step 3:** If both are matched, then display "Given element found!!!" and terminate the function
- **Step 4:** If both are not matched, then compare search element with the next element in the list.

- **Step 5:** Repeat steps 3 and 4 until the search element is compared with the last element in the list.
- **Step 6:** If the last element in the list is also not matched, then display "Element not found!!!" and terminate the function.

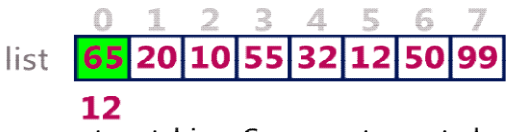
Example

Consider the following list of element and search element...

search element **12**

Step 1:

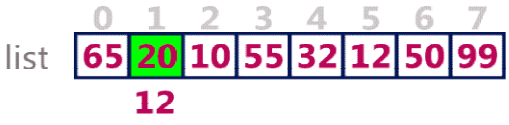
search element (12) is compared with first element (65)



Both are not matching. So move to next element

Step 2:

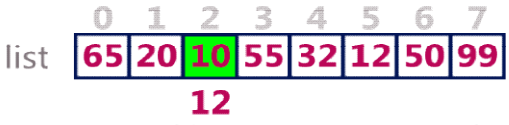
search element (12) is compared with next element (20)



Both are not matching. So move to next element

Step 3:

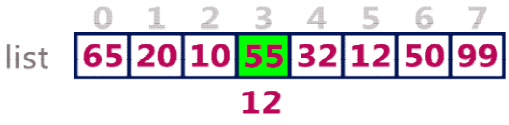
search element (12) is compared with next element (10)



Both are not matching. So move to next element

Step 4:

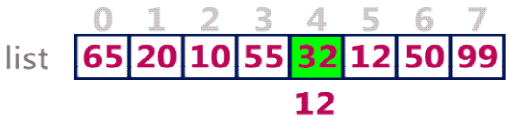
search element (12) is compared with next element (55)



Both are not matching. So move to next element

Step 5:

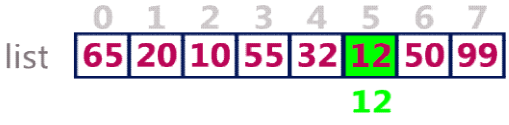
search element (12) is compared with next element (32)



Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)



Both are matching. So we stop comparing and display element found at index 5.

Program to implement Linear Search Algorithm using C.

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

void main(){
int list[20],size,i,sElement;

printf("Enter size of the list: ");
scanf("%d",&size);

printf("Enter any %d integer values: ",size);
for(i = 0; i < size; i++)
scanf("%d",&list[i]);

printf("Enter the element to be Search: ");
scanf("%d",&sElement);

  // Linear Search Logic
for(i = 0; i < size; i++)
  {
if(sElement == list[i])
     {
printf("Element is found at %d index", i);
break;
     }
  }
if(i == size)
printf("Given element is not found in the list!!!");
getch();


}
```

BINARY SEARCH ALGORITHM

Searching is a process of finding a value in a list of values. In other words, searching is the process of locating the given value position in a list of values.

Searching is the process of finding the given element in a list of elements

Binary search algorithm finds the given element in a list of elements with **O(log n)** time complexity where **n** is the total number of elements in the list. The binary search algorithm can be used only with sorted list of elements. That means, binary search can be used only with list of elements that are already arranged in order. The binary search cannot be used with

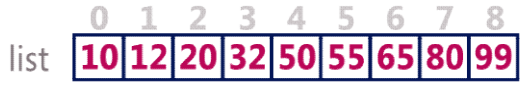
unordered list of elements. This search process starts comparing the search element with the middle element in the list. If both are matched, then the result is "**element found**". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for left sublist of the middle element. If the search element is larger, then we repeat the same process for right sublist of the middle element. We repeat this process until we find the search element in the list or until we are left with only one element in the sublist. And if that element also doesn't match with the search element, then we get the result as "Element not found in the list".

Binary search is implemented using following steps...

- **Step 1:** Read the search element from the user
- **Step 2:** Find the middle element in the sorted list
- **Step 3:** Compare, the search element with the middle element in the sorted list.
- **Step 4:** If both are matched, then display "Given element found!!!" and terminate the function
- **Step 5:** If both are not matched, then check whether the search element is smaller or larger than middle element.
- **Step 6:** If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- **Step 7:** If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- **Step 8:** Repeat the same process until we find the search element in the list or until the sublist contains only one element.
- **Step 9:** If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.

Example

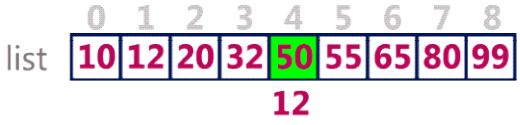
Consider the following list of elements and search element...



search element **12**

Step 1:

search element (12) is compared with middle element (50)



Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

Step 2:

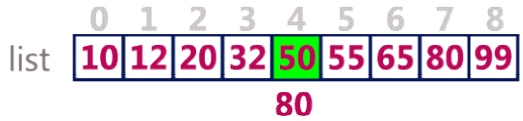
search element (12) is compared with middle element (12)

Both are matching. So the result is "Element found at index 1"

search element **80**

Step 1:

search element (80) is compared with middle element (50)



Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

Step 2:

search element (80) is compared with middle element (65)

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

Step 3:

search element (80) is compared with middle element (80)

Program to implement Binary Search Algorithm using C.

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

void main()
{
int first, last, middle, size, i, sElement, list[100];
clrscr();

printf("Enter the size of the list: ");
scanf("%d",&size);

printf("Enter %d integer values in Assending order\n", size);

for (i = 0; i < size; i++)
scanf("%d",&list[i]);

printf("Enter value to be search: ");
scanf("%d", &sElement);

first = 0;
last = size - 1;
middle = (first+last)/2;

while (first <= last) {
if (list[middle] <sElement)
first = middle + 1;
else if (list[middle] == sElement) {
printf("Element found at index %d.\n",middle);
break;
      }
else
last = middle - 1;

middle = (first + last)/2;
   }
if (first > last)
printf("Element Not found in the list.");
getch();
}
```

Bubble Sort Algorithm

Bubble Sort is a simple algorithm which is used to sort a given set of `n` elements provided in form of an array with `n` number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will **swap** both the elements, and then move on to compare the second and the third element, and so on.

If we have total `n` elements, then we need to repeat this process for `n-1` times.

It is known as **bubble sort**, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

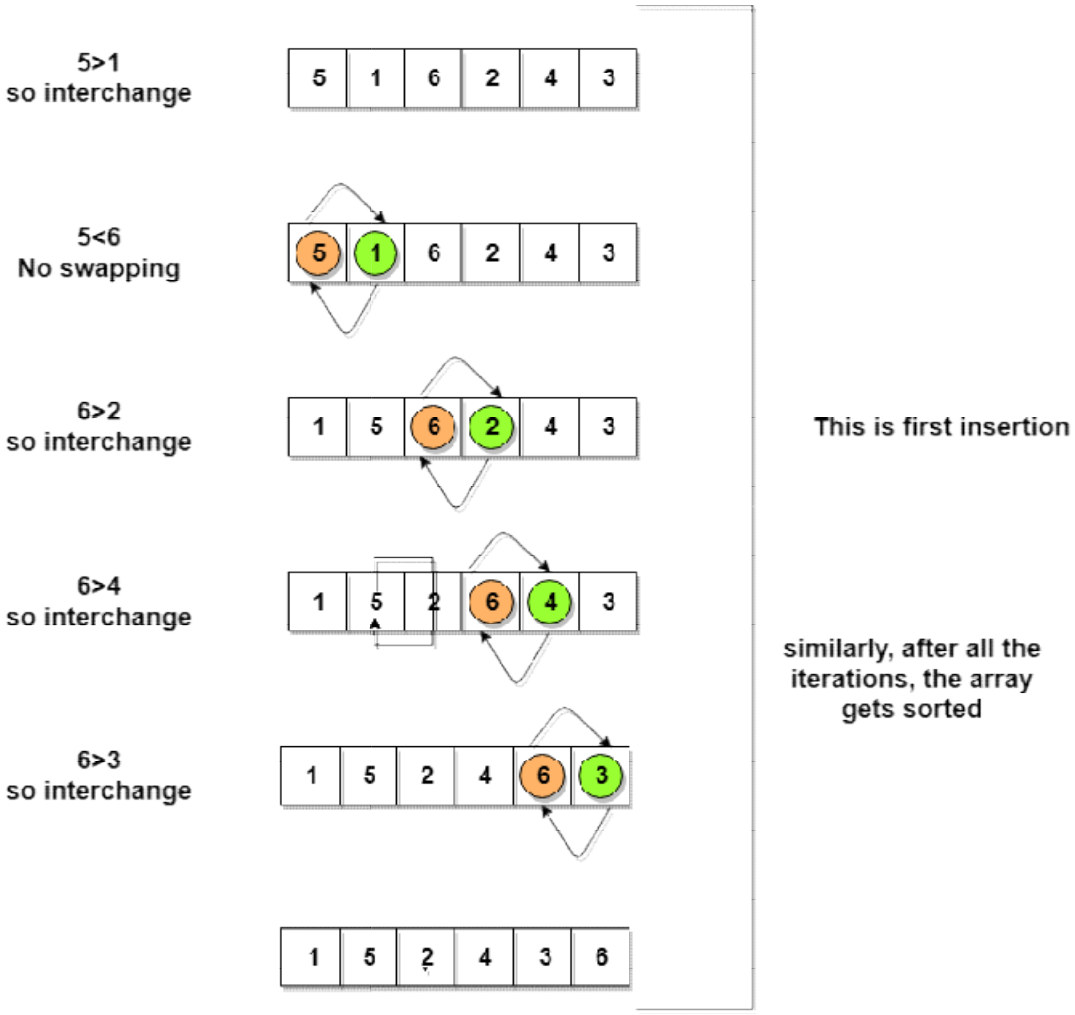
Implementing Bubble Sort Algorithm

Following are the steps involved in bubble sort(for sorting a given array in ascending order):

1. Starting with the first element(index = 0), compare the current element with the next element of the array.
2. If the current element is greater than the next element of the array, swap them.
3. If the current element is less than the next element, move to the next element. **Repeat Step 1**.

Let's consider an array with values `{5, 1, 6, 2, 4, 3}`

Below, we have a pictorial representation of how bubble sort will sort the given array.



So as we can see in the representation above, after the first iteration, 6 is placed at the last index, which is the correct position for it.

Similarly after the second iteration, 5 will be at the second last index, and so on.

Time to write the code for bubble sort:

```
// below we have a simple C program for bubble sort

#include <stdio.h>

void bubbleSort(int arr[], int n)
{
    int i, j, temp;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if( arr[j] > arr[j+1])
            {
                // swap the elements
                temp = arr[j];
                arr[j] = arr[j+1];
```

```
                arr[j+1] = temp;
            }
        }
    }

    // print the sorted array
    printf("Sorted Array: ");
    for(i = 0; i < n; i++)
    {
        printf("%d  ", arr[i]);
    }
}

int main()
{
    int arr[100], i, n, step, temp;
    // ask user for number of elements to be sorted
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);
    // input elements if the array
    for(i = 0; i < n; i++)
    {
        printf("Enter element no. %d: ", i+1);
        scanf("%d", &arr[i]);
    }
    // call the function bubbleSort
    bubbleSort(arr, n);

    return 0;
}
```

Although the above logic will sort an unsorted array, still the above algorithm is not efficient because as per the above logic, the outer `for` loop will keep on executing for **6** iterations even if the array gets sorted after the second iteration.
So, we can clearly optimize our algorithm.

Optimized Bubble Sort Algorithm

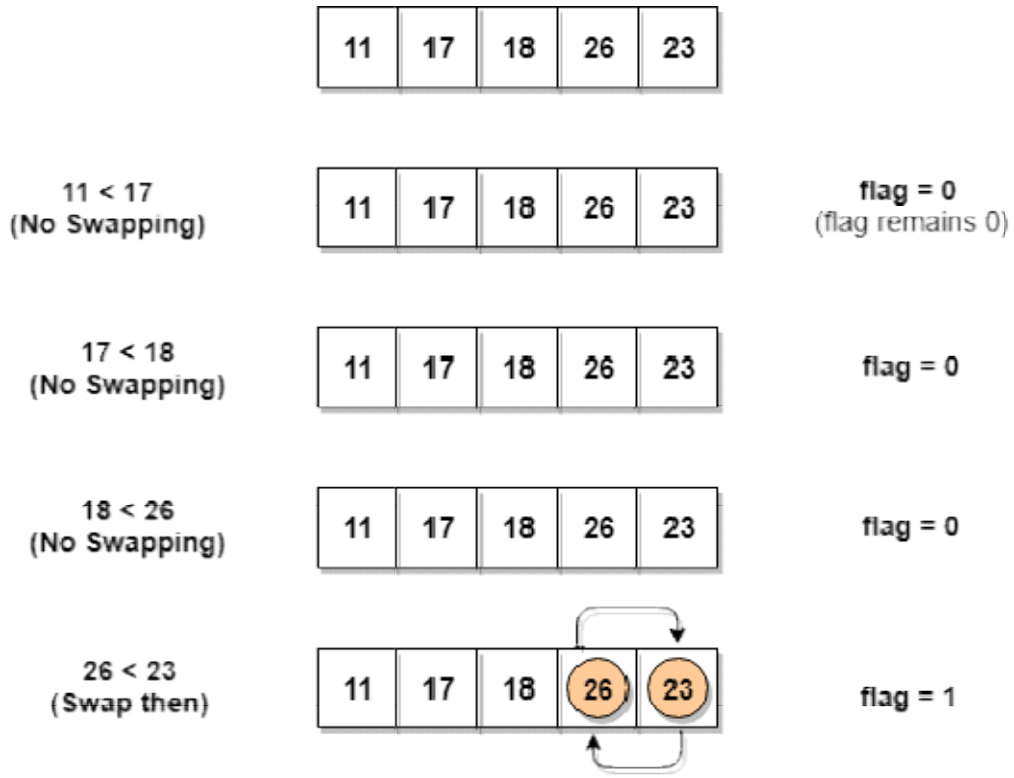
To optimize our bubble sort algorithm, we can introduce a `flag` to monitor whether elements are getting swapped inside the inner `for` loop.

Hence, in the inner `for` loop, we check whether swapping of elements is taking place or not, everytime.

If for a particular iteration, no swapping took place, it means the array has been sorted and we can jump out of the `for` loop, instead of executing all the iterations.

Let's consider an array with values `{11, 17, 18, 26, 23}`

Below, we have a pictorial representation of how the optimized bubble sort will sort the given array.



As we can see, in the first iteration, swapping took place, hence we updated our `flag` value to `1`, as a result, the execution enters the `for` loop again. But in the second iteration, no swapping will occur, hence the value of `flag` will remain `0`, and execution will break out of loop.

```
// below we have a simple C program for bubble sort
```

```
##include <stdio.h>

void bubbleSort(int arr[], int n)
{
    int i, j, temp;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            // introducing a flag to monitor swapping
            int flag = 0;
            if( arr[j] > arr[j+1])
            {
                // swap the elements
                temp = arr[j];
                arr[j] = arr[j+1];
```

```
                arr[j+1] = temp;
                // if swapping happens update flag to 1
                flag = 1;
            }
        }
        // if value of flag is zero after all the iterations of inner
loop
        // then break out
        if(!flag)
        {
            break;
        }
    }

    // print the sorted array
    printf("Sorted Array: ");
    for(i = 0; i < n; i++)
    {
        printf("%d  ", arr[i]);
    }
}

int main()
{
    int arr[100], i, n, step, temp;
    // ask user for number of elements to be sorted
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);
    // input elements if the array
    for(i = 0; i < n; i++)
    {
        printf("Enter element no. %d: ", i+1);
        scanf("%d", &arr[i]);
    }
    // call the function bubbleSort
    bubbleSort(arr, n);

    return 0;
}
```

In the above code, in the function `bubbleSort`, if for a single complete cycle of `j` iteration(inner `for`loop), no swapping takes place, then `flag` will remain `0` and then we will break out of the `for` loops, because the array has already been sorted.

Complexity Analysis of Bubble Sort

In Bubble Sort, `n-1` comparisons will be done in the 1st pass, `n-2` in 2nd pass, `n-3` in 3rd pass and so on. So the total number of comparisons will be,

```
(n-1) + (n-2) + (n-3) + ..... + 3 + 2 + 1
Sum = n(n-1)/2
i.e O(n²)
```

Hence the **time complexity** of Bubble Sort is **O(n²)**.

The main advantage of Bubble Sort is the simplicity of the algorithm.

The **space complexity** for Bubble Sort is **O(1)**, because only a single additional memory space is required i.e. for `temp` variable.

Also, the **best case time complexity** will be **O(n)**, it is when the list is already sorted.

Following are the Time and Space complexity for the Bubble Sort algorithm.

- Worst Case Time Complexity [Big-O]: **O(n²)**
- Best Case Time Complexity [Big-omega]: **O(n)**
- Average Time Complexity [Big-theta]: **O(n²)**
- Space Complexity: **O(1)**

Insertion Sort Algorithm

Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending).

Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

Step by Step Process

The insertion sort algorithm is performed using following steps...

- **Step 1 -** Assume that first element in the list is in sorted portion and all the remaining elements are in unsorted portion.

- **Step 2:** Take first element from the unsorted portion and insert that element into the sorted portion in the order specified.
- **Step 3:** Repeat the above process until all the elements from the unsorted portion are moved into the sorted portion.

Following is the sample code for insertion sort...

Insertion Sort Logic

```
//Insertion sort logic
for i = 1 to size-1 {
temp = list[i];
      j = i-1;
while ((temp < list[j]) && (j > 0)) {
list[j] = list[j-1];
         j = j - 1;
      }
list[j] = temp;
   }
```

Example

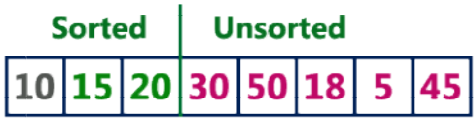
Consider the following unsorted list of elements...

Asume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...

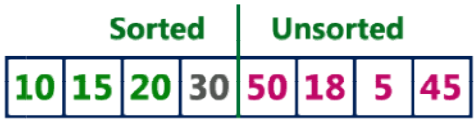
Move the first element 15 from unsorted portion to sorted portion of the list.

To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position

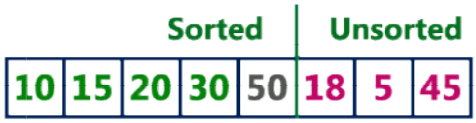
To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.



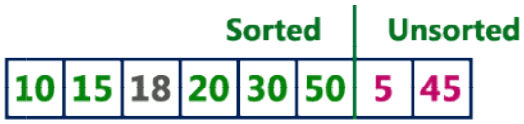
To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.



To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.



To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.



To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these element, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.

Complexity of the Insertion Sort Algorithm

To sort an unsorted list with **'n'** number of elements, we need to make **(1+2+3+.....+n-1) = (n (n-1))/2** number of comparisions in the worst case. If the list is already sorted then it requires **'n'** number of comparisions.

Worst Case : $O(n^2)$

Best Case : $\Omega(n)$

Average Case : $\Theta(n^2)$

Implementaion of Insertion Sort Algorithm using C Programming Language

```
#include<stdio.h>
#include<conio.h>
void main(){
int size, i, j, temp, list[100];

printf("Enter the size of the list: ");
scanf("%d", &size);

printf("Enter %d integer values: ", size);
for (i = 0; i < size; i++)
scanf("%d", &list[i]);

   //Insertion sort logic
for (i = 1; i < size; i++) {
temp = list[i];
      j = i - 1;
while ((temp < list[j]) && (j >= 0)) {
list[j + 1] = list[j];
         j = j - 1;
      }
list[j + 1] = temp;
   }

printf("List after Sorting is: ");
for (i = 0; i < size; i++)
printf(" %d", list[i]);
getch();
```

}

Selection Sort Algorithm

Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending). In selection sort, the first element in the list is selected and it is compared repeatedly with all the remaining elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped so that first position is filled with smallest element in the sorted order. Next we select the element at second position in the list and it is compared with all the remaining elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated till the entire list is sorted.

Step by Step Process

The selection sort algorithm is performed using following steps...

- **Step 1 -** Select the first element of the list (i.e., Element at first position in the list).
- **Step 2:** Compare the selected element with all the other elements in the list.
- **Step 3:** In every comparision, if any element is found smaller than the selected element (for Ascending order), then both are swapped.
- **Step 4:** Repeat the same procedure with element in the next position in the list till the entire list is sorted.

Following is the sample code for selection sort...

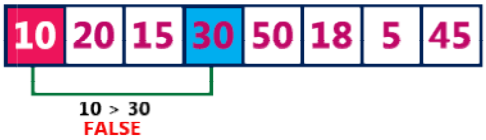
Selection Sort Logic

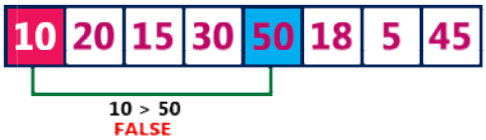
```
//Selection sort logic
for(i=0; i<size; i++){
for(j=i+1; j<size; j++){
if(list[i] > list[j])
           {
temp=list[i];
list[i]=list[j];
list[j]=temp;
           }
      }
  }
```

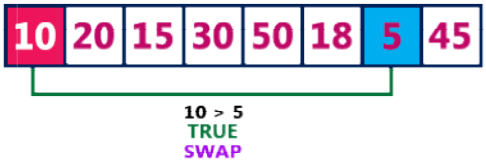
Consider the following unsorted list of elements...

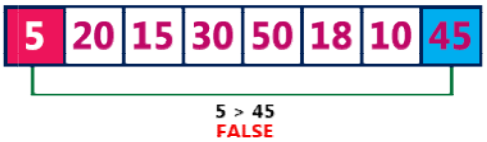
Iteration #1

Select the first position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.









List after 1st iteration

Iteration #2

Select the second position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 2nd iteration

Iteration #3

Select the third position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 3rd iteration

Iteration #4

Select the fourth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

Complexity of the Selection Sort Algorithm

To sort an unsorted list with **'n'** number of elements, we need to make **((n-1)+(n-2)+(n-3)+.....+1) = (n (n-1))/2** number of comparisions in the worst case. If the list is already sorted then it requires **'n'** number of comparisions.

Worst Case : $O(n^2)$

Best Case : $\Omega(n^2)$

Average Case : $\Theta(n^2)$

Implementaion of Selection Sort Algorithm using C Programming Language

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

void main(){

intsize,i,j,temp,list[100];
clrscr();

printf("Enter the size of the List: ");
scanf("%d",&size);

printf("Enter %d integer values: ",size);
for(i=0; i<size; i++)
scanf("%d",&list[i]);

  //Selection sort logic

for(i=0; i<size; i++){
for(j=i+1; j<size; j++){
if(list[i] > list[j])
           {
temp=list[i];
list[i]=list[j];
list[j]=temp;
           }
      }
  }
```

```
printf("List after sorting is: ");
for(i=0; i<size; i++)
printf(" %d",list[i]);

getch();
}
```

Performance Analysis

What is Performance Analysis of an algorithm?

If we want to go from city "A" to city "B", there can be many ways of doing this. We can go by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one which suits us. Similarly, in computer science there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one. Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem. When there are multiple alternative algorithms to solve a problem, we analyse them and pick the one which is best suitable for our requirements. Formal definition is as follows...

Performance of an algorithm is a process of making evaluative judgement about algorithms.

It can also be defined as follows...

Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.

That means when we have multiple algorithms to solve a problem, we need to select a suitable algorithm to solve that problem.
We compare algorithms with each other which are solving same problem, to select the

best algorithm. To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, execution speed of that algorithm, easy to understand, easy to implement, etc.,

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

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem? Etc.,

When we want to analyse an algorithm, we consider only the space and time required by that particular algorithm and we ignore all the remaining elements. Based on this information, performance analysis of an algorithm can also be defined as follows...

Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.

Performance analysis of an algorithm is performed by using the following measures...

1. Space required to complete the task of that algorithm (**Space Complexity**). It includes program space and data space
2. Time required to complete the task of that algorithm (**Time Complexity**)

Space Complexity

What is Space complexity?

When we design an algorithm to solve a problem, it needs some computer memory to complete its execution. For any algorithm, memory is required for the following purposes...

1. To store program instructions.
2. To store constant values.
3. To store variable values.
4. And for few other things like funcion calls, jumping statements etc,.

Space complexity of an algorithm can be defined as follows...

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.

Generally, when a program is under execution it uses the computer memory for THREE reasons. They are as follows...

1. **Instruction Space:** It is the amount of memory used to store compiled version of instructions.
2. **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.
3. **Data Space:** It is the amount of memory used to store all the variables and constants.

To calculate the space complexity, we must know the memory required to store different datatype values (according to the compiler). For example, the C Programming Language compiler requires the following...

1. 2 bytes to store Integer value.
2. 4 bytes to store Floating Point value.
3. 1 byte to store Character value.
4. 6 (OR) 8 bytes to store double value.

Consider the following piece of code...

```
int square(int a)

{

        return a*a;
```

```
}
```

In the above piece of code, it requires 2 bytes of memory to store variable **'a'** and another 2 bytes of memory is used for **return value**.

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

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

Consider the following piece of code...

Example 2

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

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

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

If the amount of space required by an algorithm is increased with the increase of input value, then that space complexity is said to be Linear Space Complexity.

Time Complexity

What is Time complexity?

Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer time required is called time complexity. Time complexity of an algorithm can be defined as follows...

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

Generally, running time of an algorithm depends upon the following...

1. Whether it is running on **Single** processor machine or **Multi** processor machine.
2. Whether it is a **32 bit** machine or **64 bit** machine.
3. **Read** and **Write** speed of the machine.
4. The amount of time required by an algorithm to perform **Arithmetic** operations, **logical** operations, **return** value and **assignment**operations etc.,
5. **Input** data

Calculating Time Complexity of an algorithm based on the system configuration is a very difficult task because, the configuration changes from one system to another system. To solve this problem, we must assume a model machine with specific configuration. So that, we can able to calculate generalized time complexity according to that model machine.

To calculate time complexity of an algorithm, we need to define a model machine. Let us assume a machine with following configuration...

1. It is a Single processor machine
2. It is a 32 bit Operating System machine
3. It performs sequential execution
4. It requires 1 unit of time for Arithmetic and Logical operations
5. It requires 1 unit of time for Assignment and Return value
6. It requires 1 unit of time for Read and Write operations

Now, we calculate the time complexity of following example code by using the above defined model machine...

Consider the following piece of code...

```
int sum(int a, int b)
{
   return a+b;
}
```

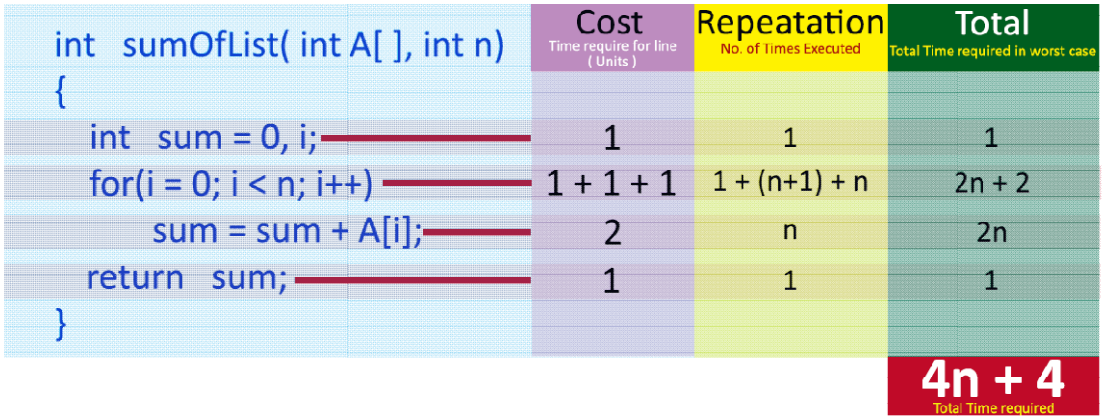
In above sample code, it requires 1 unit of time to calculate a+b and 1 unit of time to return the value. That means, totally it takes 2 units of time to complete its execution. And it does not change based on the input values of a and b. That means for all input values, it requires same amount of time i.e. 2 units.

If any program requires fixed amount of time for all input values then its time complexity is said to be Constant Time Complexity.

Consider the following piece of code...

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

For the above code, time complexity can be calculated as follows...



In above calculation **Cost** is the amount of computer time required for a single operation in each line. **Repeatation** is the amount of computer time required by each operation for all its

repeatations.
Total is the amount of computer time required by each operation to execute. So above code requires **'4n+4' Units** of computer time to complete the task. Here the exact time is not fixed. And it changes based on the **n**value. If we increase the **n** value then the time required also increases linearly.

Totally it takes '4n+4' units of time to complete its execution and it is *Linear Time Complexity*.

If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be Linear Time Complexity.

Asymptotic Notations

What is Asymptotic Notation?

Whenever we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm. But when we calculate complexity of an algorithm it does not provide exact amount of resource required. So instead of taking exact amount of resource we represent that complexity in a general form (Notation) which produces the basic nature of that algorithm. We use that general form (Notation) for analysis process.

Asymptotic notation of an algorithm is a mathematical representation of its complexity.

Note - In asymptotic notation, when we want to represent the complexity of an algorithm, we use only the most significant terms in the complexity of that algorithm and ignore least significant terms in the complexity of that algorithm (Here complexity can be Space Complexity or Time Complexity).

For example, consider the following time complexities of two algorithms...

- **Algorithm 1 : $5n^2 + 2n + 1$**
- **Algorithm 2 : $10n^2 + 8n + 3$**

Generally, when we analyze an algorithm, we consider the time complexity for larger values of input data (i.e. **'n'** value). In above two time complexities, for larger value of **'n'** the term **'2n + 1'** in algorithm 1 has least significance than the term **'$5n^2$'**, and the term **'8n + 3'** in algorithm 2 has least significance than the term **'$10n^2$'**. Here, for larger value of **'n'** the value of most significant terms (**$5n^2$** and **$10n^2$**) is very larger than the value of least significant terms (**2n + 1** and **8n + 3**). So for larger value of **'n'** we ignore the least significant terms to represent overall time required by an algorithm. In asymptotic notation, we use only the most significant terms to represent the time complexity of an algorithm.

Majorly, we use THREE types of Asymptotic Notations and those are as follows...

1. **Big - Oh (O)**
2. **Big - Omega (Ω)**
3. **Big - Theta (Θ)**

Big - Oh Notation (O)

Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.

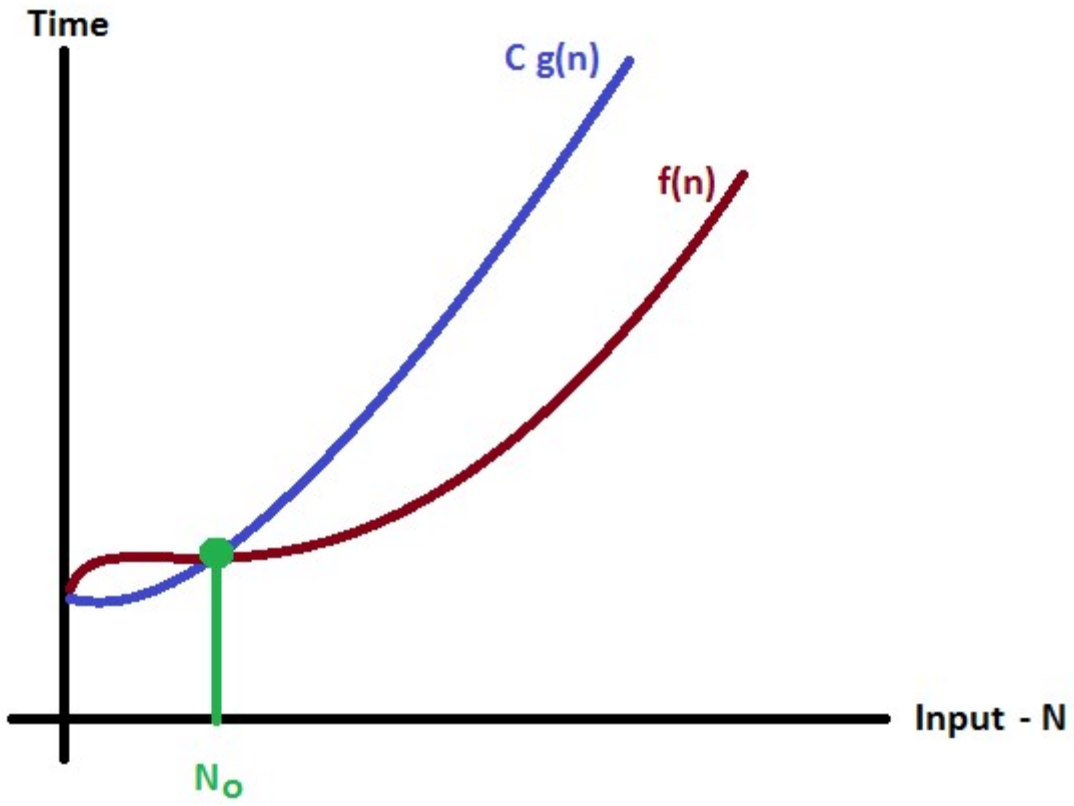
That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

Big - Oh Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as O(g(n)).

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

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.

Example

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= C g(n)** for all values of **C > 0** and **n_0>= 1**

f(n) <= C g(n)

⇒3n + 2 <= C n

Above condition is always TRUE for all values of **C = 4** and **n >= 2**.

By using Big - Oh notation we can represent the time complexity as follows...

3n + 2 = O(n)

Big - Omege Notation (Ω)

Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.

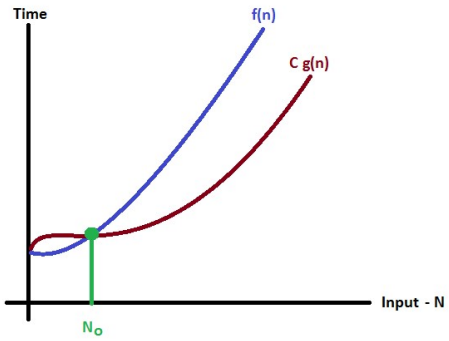
That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity.

Big - Omega Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as Ω(g(n)).

f(n) = Ω(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C g(n) is less than f(n) which indicates the algorithm's lower bound.

Example

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n_0>= 1**

f(n) >= C g(n)

⇒3n + 2 >= C n

Above condition is always TRUE for all values of **C = 1** and **n >= 1**.

By using Big - Omega notation we can represent the time complexity as follows...

3n + 2 = Ω(n)

Big - Theta Notation (Θ)

Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.

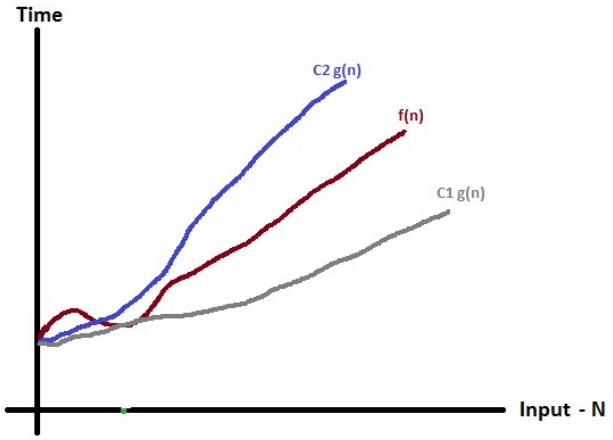
That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.

Big - Theta Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If C_1 g(n) <= f(n) <= C_2 g(n) for all n >= n_0, C_1 > 0, C_2 > 0 and n_0 >= 1. Then we can represent f(n) as Θ(g(n)).

f(n) = Θ(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C_1 g(n) is less than f(n) and C_2 g(n) is greater than f(n) which indicates the algorithm's average bound.

Example

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **Θ(g(n))** then it must satisfy **C_1 g(n) <= f(n) <= C_2 g(n)** for all values of **C_1 > 0, C_2 > 0** and **n_0>= 1**

C_1 g(n) <= f(n) <= C_2 g(n)

⇒C_1 n <= 3n + 2 <= C_2 n

Above condition is always TRUE for all values of **C_1 = 1, C_2 = 4** and **n >= 2**.

By using Big - Theta notation we can represent the time compexity as follows...

3n + 2 = Θ(n)

Asymptotic Notations

What is Asymptotic Notation?

Whenever we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm. But when we calculate complexity of an algorithm it does not provide exact amount of resource required. So instead of taking exact amount of resource we represent that complexity in a general form (Notation) which produces the basic nature of that algorithm. We use that general form (Notation) for analysis process.

Asymptotic notation of an algorithm is a mathematical representation of its complexity.

Note - In asymptotic notation, when we want to represent the complexity of an algorithm, we use only the most significant terms in the complexity of that algorithm and ignore least significant terms in the complexity of that algorithm (Here complexity can be Space Complexity or Time Complexity).

For example, consider the following time complexities of two algorithms...

- **Algorithm 1 : $5n^2 + 2n + 1$**
- **Algorithm 2 : $10n^2 + 8n + 3$**

Generally, when we analyze an algorithm, we consider the time complexity for larger values of input data (i.e. **'n'** value). In above two time complexities, for larger value of **'n'** the term **'2n + 1'** in algorithm 1 has least significance than the term **'$5n^2$'**, and the term **'8n + 3'** in algorithm 2 has least significance than the term **'$10n^2$'**. Here, for larger value of **'n'** the value of most significant terms (**$5n^2$** and **$10n^2$**) is very larger than the value of least significant terms (**2n + 1** and **8n + 3**). So for larger value of **'n'** we ignore the least significant terms to represent overall time required by an algorithm. In asymptotic notation, we use only the most significant terms to represent the time complexity of an algorithm.

Majorly, we use THREE types of Asymptotic Notations and those are as follows...

1. **Big - Oh (O)**
2. **Big - Omega (Ω)**
3. **Big - Theta (Θ)**

Big - Oh Notation (O)

Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.

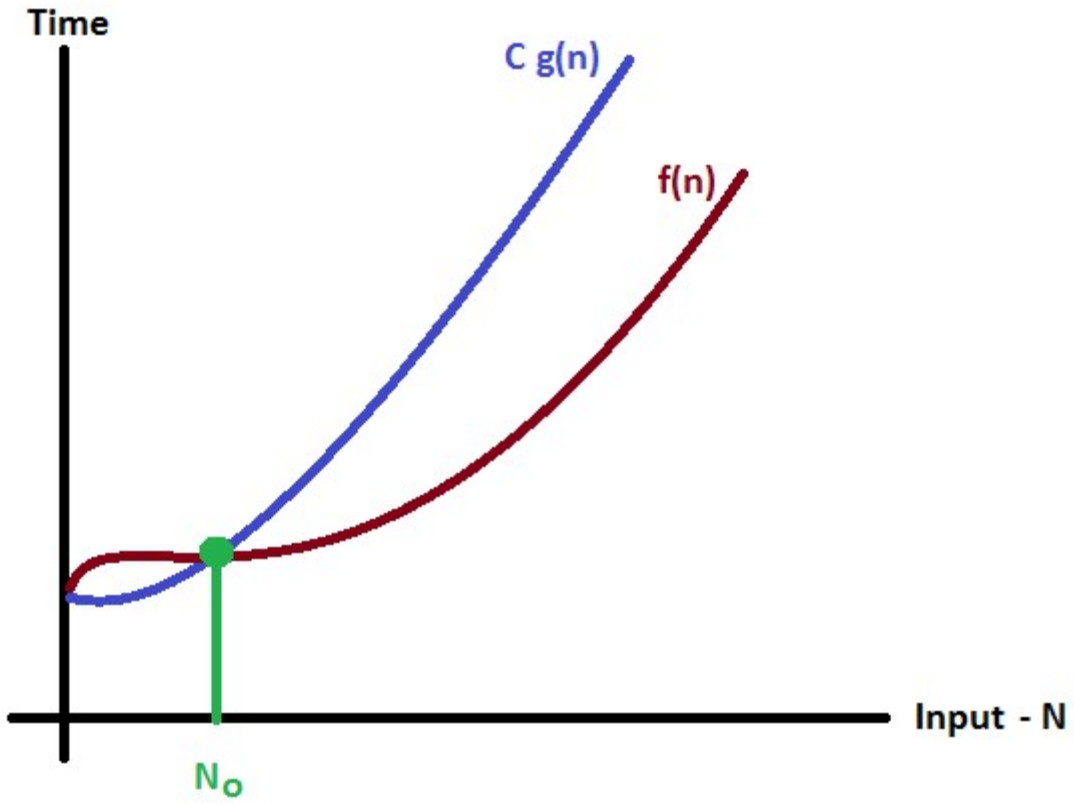
That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

Big - Oh Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as O(g(n)).

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

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.

Example

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= C g(n)** for all values of **C > 0** and **n_0>= 1**

f(n) <= C g(n)

⇒3n + 2 <= C n

Above condition is always TRUE for all values of **C = 4** and **n >= 2**.

By using Big - Oh notation we can represent the time complexity as follows...

3n + 2 = O(n)

Big - Omege Notation (Ω)

Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.

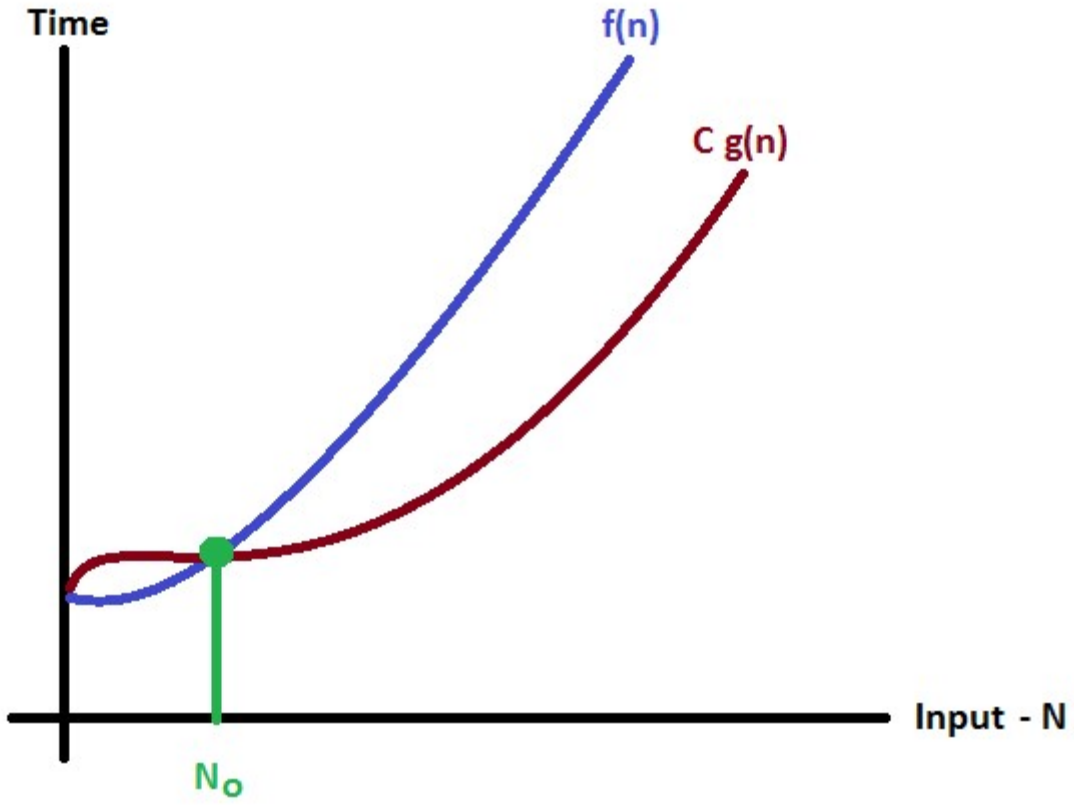
That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity.

Big - Omega Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as Ω(g(n)).

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

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C g(n) is less than f(n) which indicates the algorithm's lower bound.

Example

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n_0>= 1**

f(n) >= C g(n)

⇒3n + 2 >= C n

Above condition is always TRUE for all values of **C = 1** and **n >= 1**.

By using Big - Omega notation we can represent the time complexity as follows...

3n + 2 = Ω(n)

Big - Theta Notation (Θ)

Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.

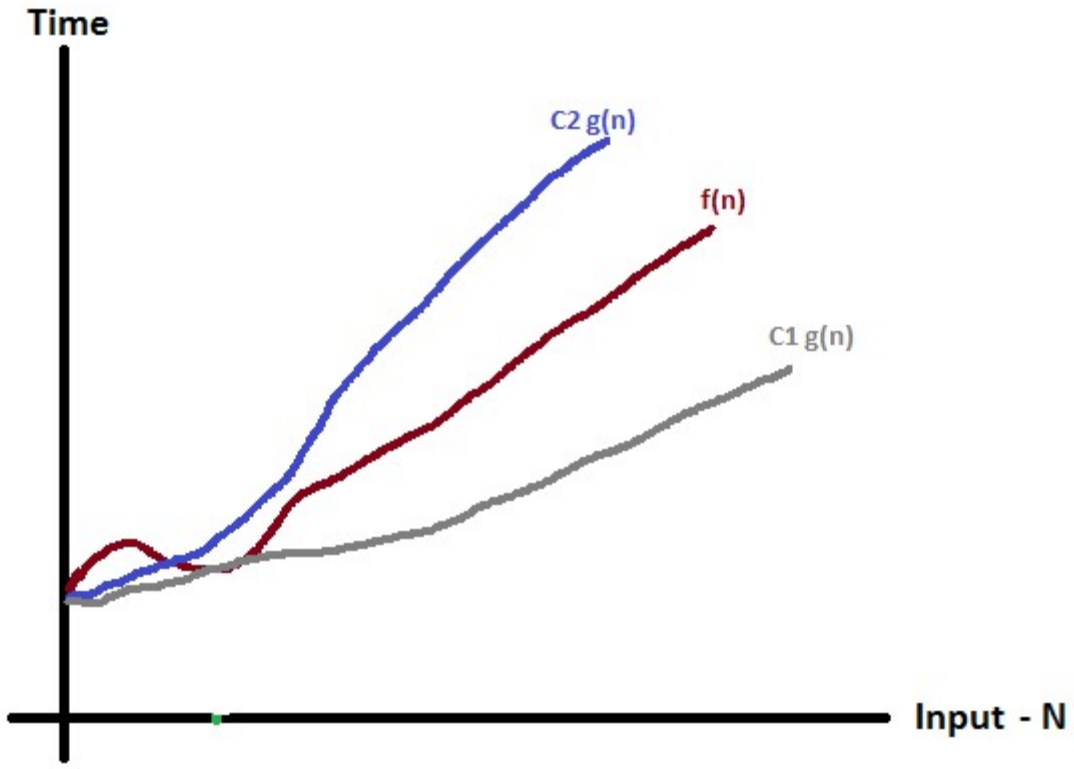
That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.

Big - Theta Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If C_1 g(n) <= f(n) <= C_2 g(n) for all n >= n_0, C_1 > 0, C_2 > 0 and n_0 >= 1. Then we can represent f(n) as Θ(g(n)).

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

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C_1 g(n) is less than f(n) and C_2 g(n) is greater than f(n) which indicates the algorithm's average bound.

Example

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **Θ(g(n))** then it must satisfy **C_1 g(n) <= f(n) <= C_2 g(n)** for all values of **C_1 > 0, C_2 > 0** and **n_0>= 1**

C_1 g(n) <= f(n) <= C_2 g(n)

⇒C_1 n <= 3n + 2 <= C_2 n

Above condition is always TRUE for all values of **C_1 = 1, C_2 = 4** and **n >= 2**.

By using Big - Theta notation we can represent the time compexity as follows...

3n + 2 = Θ(n)

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

Asymptotic Notations

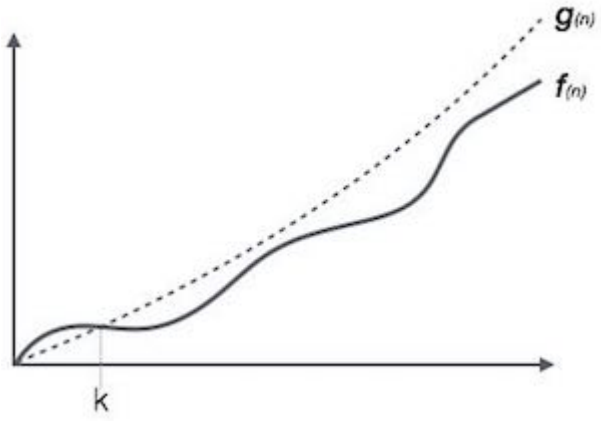
Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big Oh Notation, O

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

longest amount of time an algorithm can possibly take to complete.

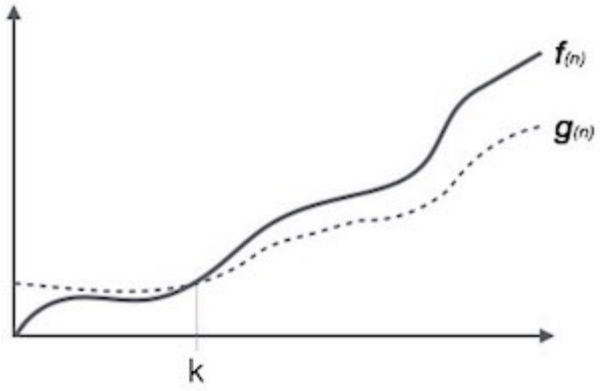


For example, for a function ***f*(n)**

$O(f(n)) = \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } f(n) \leq c.g(n) \text{ for all } n > n_0. \}$

Omega Notation, Ω

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

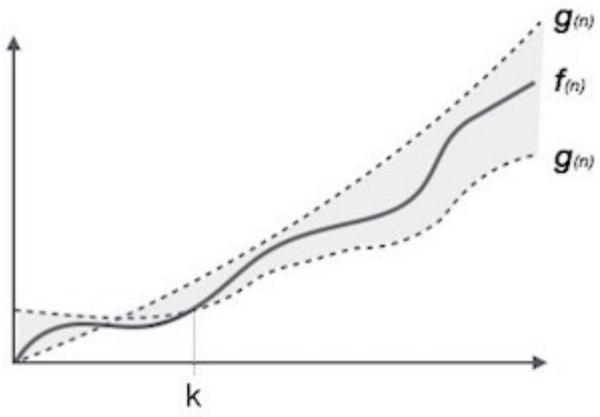


For example, for a function ***f*(n)**

$\Omega(f(n)) \geq \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } g(n) \leq c.f(n) \text{ for all } n > n_0. \}$

Theta Notation, θ

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



$\theta(f(n)) = \{ g(n) \text{ if and only if } g(n) = O(f(n)) \text{ and } g(n) = \Omega(f(n)) \text{ for all } n > n_0. \}$

Common Asymptotic Notations

Following is a list of some common asymptotic notations –

constant	–	O(1)
logarithmic	–	O(log n)
linear	–	O(n)
n log n	–	O(n log n)
quadratic	–	O(n2)
cubic	–	O(n3)
polynomial	–	nO(1)
exponential	–	2O(n)

Searching

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful and the process returns the location of that element, otherwise the search is called unsuccessful.

There are two popular search methods that are widely used in order to search some item into the list. However, choice of the algorithm depends upon the arrangement of the list.

- Linear Search
- Binary Search

Linear Search

Linear search is the simplest search algorithm and often called sequential search. In this type of searching, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match found then location of the item is returned otherwise the algorithm return NULL.

Implementing Linear Search

In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

The time complexity of Linear search algorithm is O(n), we will analyse the same and see why it is O(n) after implementing it.

Following are the steps of implementation that we will be following:

1. Traverse the array using a `for` loop.
2. In every iteration, compare the `target` value with the current value of the array.
    - If the values match, return the current index of the array.
    - If the values do not match, move on to the next array element.
3. If no match is found, return `-1`.

To search the number 5 in the array given below, linear search will go step by step in a sequential order starting from the first element in the given

array.

Complexity of algorithm

Complexity	Best Case	Average Case	Worst Case
Time	O(1)	O(n)	O(n)
Space			O(1)

```
1.  #include<stdio.h>
2.  void main ()
3.  {
4.     int a[10] = {10, 23, 40, 1, 2, 0, 14, 13, 50, 9};
5.     int item, i,flag;
6.     printf("\nEnter Item which is to be searched\n");
7.     scanf("%d",&item);
8.     for (i = 0; i< 10; i++)
9.     {
10.       if(a[i] == item)
11.       {
12.          flag = i+1;
13.          break;
14.       }
15.       else
16.       flag = 0;
17.    }
18.    if(flag != 0)
19.    {
20.       printf("\nItem found at location %d\n",flag);
21.    }
22.    else
23.    {
24.       printf("\nItem not found\n");
25.    }
26. }
```

Output:

```
Enter Item which is to be searched
20
Item not found
Enter Item which is to be searched
23
Item found at location 2
```

Binary Search

Binary search is the search technique which works efficiently on the sorted lists. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.

Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

Binary search algorithm is given below.

BINARY_SEARCH(A, lower_bound, upper_bound, VAL)

- Step 1: [INITIALIZE] SET BEG = lower_bound
  END = upper_bound, POS = - 1
- Step 2: Repeat Steps 3 and 4 while BEG <=END
- Step 3: SET MID = (BEG + END)/2
- Step 4: IF A[MID] = VAL
  SET POS = MID
  PRINT POS
  Go to Step 6
  ELSE IF A[MID] > VAL
  SET END = MID - 1
  ELSE
  SET BEG = MID + 1
  [END OF IF]
  [END OF LOOP]
- Step 5: IF POS = -1
  PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
  [END OF IF]
- Step 6: EXIT

Complexity

SN	Performance	Complexity
1	Worst case	O(log n)
2	Best case	O(1)
3	Average Case	O(log n)
4	Worst case space complexity	O(1)

Example

Let us consider an array arr = {1, 5, 7, 8, 13, 19, 20, 23, 29}. Find the location of the item 23 in the array.

In 1st step :

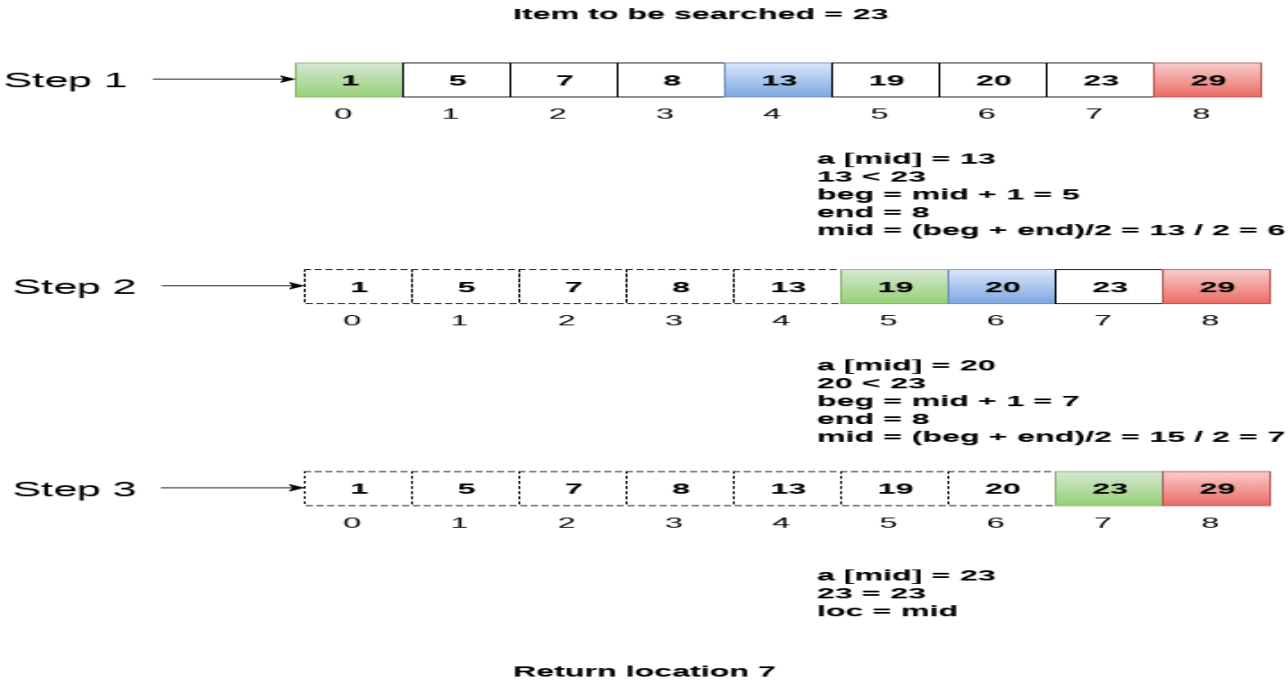
1. BEG = 0
2. END = 8ron
3. MID = 4
4. a[mid] = a[4] = 13 < 23, therefore

in Second step:

1. Beg = mid +1 = 5
2. End = 8
3. mid = 13/2 = 6
4. a[mid] = a[6] = 20 < 23, therefore;

in third step:

1. beg = mid + 1 = 7
2. End = 8
3. mid = 15/2 = 7
4. a[mid] = a[7]
5. a[7] = 23 = item;
6. therefore, set location = mid;
7. The location of the item will be 7.



Binary Search Program using Recursion

```
#include<stdio.h>
int binarySearch(int[], int, int, int);
void main ()  {
   int arr[10] = {16, 19, 20, 23, 45, 56, 78, 90, 96, 100};
   int item, location=-1;
   printf("Enter the item which you want to search ");
   scanf("%d",&item);
   location = binarySearch(arr, 0, 9, item);
   if(location != -1)
   {
      printf("Item found at location %d",location);
   }
   else
   {
      printf("Item not found");
   }
}
int binarySearch(int a[], int beg, int end, int item)
{
   int mid;
   if(end >= beg)
   {
      mid = (beg + end)/2;
      if(a[mid] == item)
      {
         return mid+1;
      }
      else if(a[mid] < item)
      {
         return binarySearch(a,mid+1,end,item);
      }
      else
      {
         return binarySearch(a,beg,mid-1,item);
      }

   }
   return -1;
} Output:
```

```
Enter the item which you want to search
19
Item found at location 2
```

Sorting

Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending).

1)Bubble Sort
2)Selection Sort
3)Insertion Sort

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.
Example:
First Pass:
(5 1 4 2 8) –> (1 5 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 5 4 2 8) –> (1 4 5 2 8), Swap since 5 > 4
(1 4 5 2 8) –> (1 4 2 5 8), Swap since 5 > 2
(1 4 2 5 8) –> (1 4 2 5 8), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
Second Pass:
(1 4 2 5 8) –> (1 4 2 5 8)
(1 4 2 5 8) –> (1 2 4 5 8), Swap since 4 > 2
(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.
Third Pass:
(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)

// C program for implementation of Bubble sort

```
#include <stdio.h>

void swap(int *xp, int *yp)
{
        int temp = *xp;
        *xp = *yp;
        *yp = temp;
}
```

```
// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
int i, j;
for (i = 0; i < n-1; i++)

        // Last i elements are already in place
        for (j = 0; j < n-i-1; j++)
                if (arr[j] > arr[j+1])
                        swap(&arr[j], &arr[j+1]);
}

/* Function to print an array */
void printArray(int arr[], int size)
{
        int i;
        for (i=0; i < size; i++)
                printf("%d ", arr[i]);
        printf("\n");
}

// Driver program to test above functions
int main()
{
        int arr[] = {64, 34, 25, 12, 22, 11, 90};
        int n = sizeof(arr)/sizeof(arr[0]);
        bubbleSort(arr, n);
        printf("Sorted array: \n");
        printArray(arr, n);
        return 0;
}
```

Output:

```
Sorted array:
11 12 22 25 34 64 90
```

<!—Illustration :

i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i =1	0	3	1	5	8	2	4	7	**9**
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	**8**	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	**7**		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	**5**			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	**4**				
	1	1	2	3					
i = 6	0	1	2	**3**					
		1	**2**						

2)Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

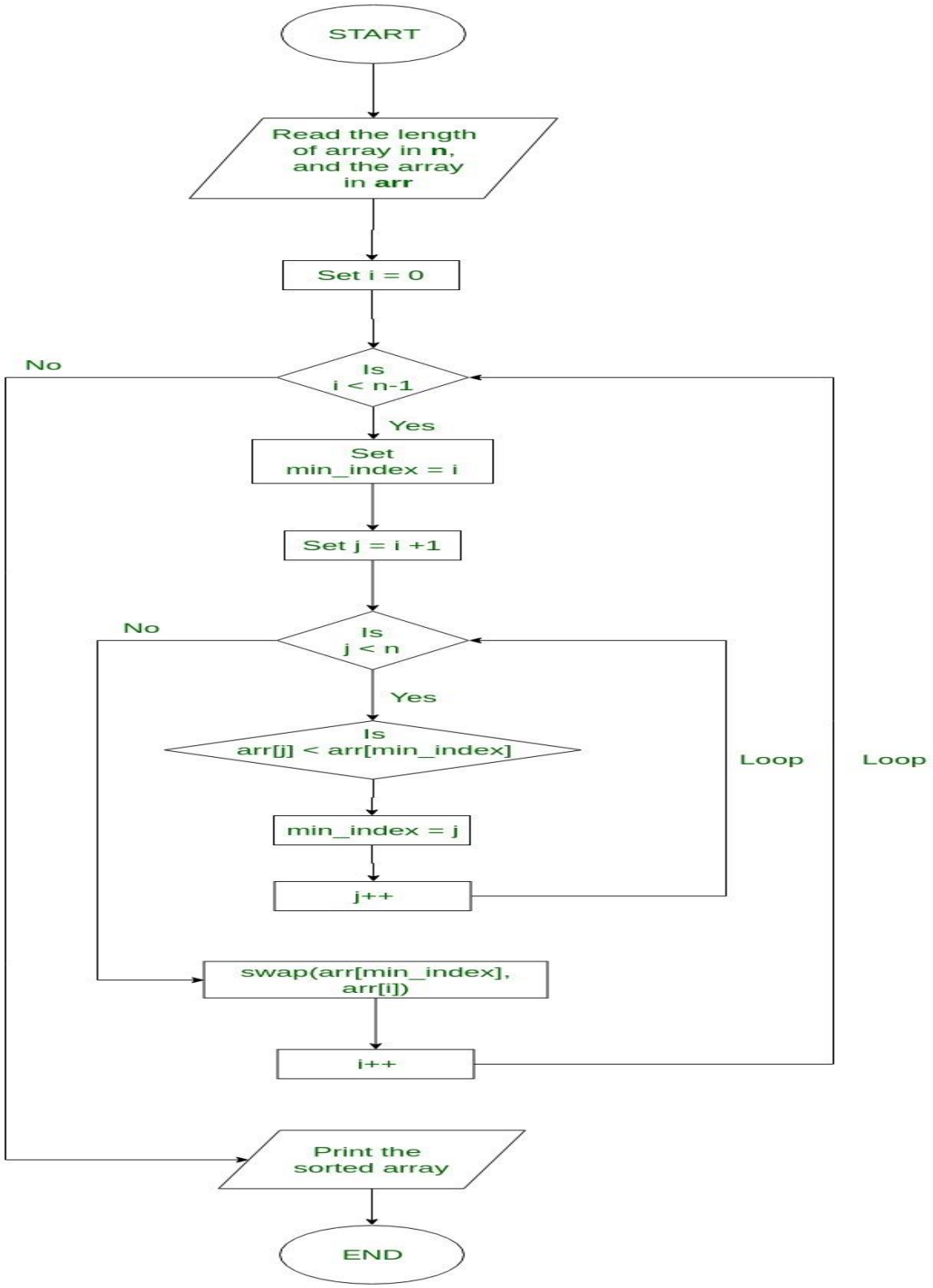
1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Following example explains the above steps:

```
arr[] = 64 25 12 22 11


// Find the minimum element in arr[0...4]

// and place it at beginning

11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

Flowchart of the Selection Sort:



Flowchart for Selection Sort

```
// C program for implementation of selection sort
#include <stdio.h>

void swap(int *xp, int *yp)
{
	int temp = *xp;
	*xp = *yp;
	*yp = temp;
}

void selectionSort(int arr[], int n)
{
	int i, j, min_idx;

	// One by one move boundary of unsorted subarray
	for (i = 0; i < n-1; i++)
	{
		// Find the minimum element in unsorted array
		min_idx = i;
		for (j = i+1; j < n; j++)
		if (arr[j] < arr[min_idx])
			min_idx = j;

		// Swap the found minimum element with the first element
		swap(&arr[min_idx], &arr[i]);
	}
}

/* Function to print an array */
void printArray(int arr[], int size)
{
	int i;
	for (i=0; i < size; i++)
		printf("%d ", arr[i]);
	printf("\n");
}

// Driver program to test above functions
int main()
{
	int arr[] = {64, 25, 12, 22, 11};
	int n = sizeof(arr)/sizeof(arr[0]);
	selectionSort(arr, n);
	printf("Sorted array: \n");
	printArray(arr, n);
	return 0;
}
```

Output:

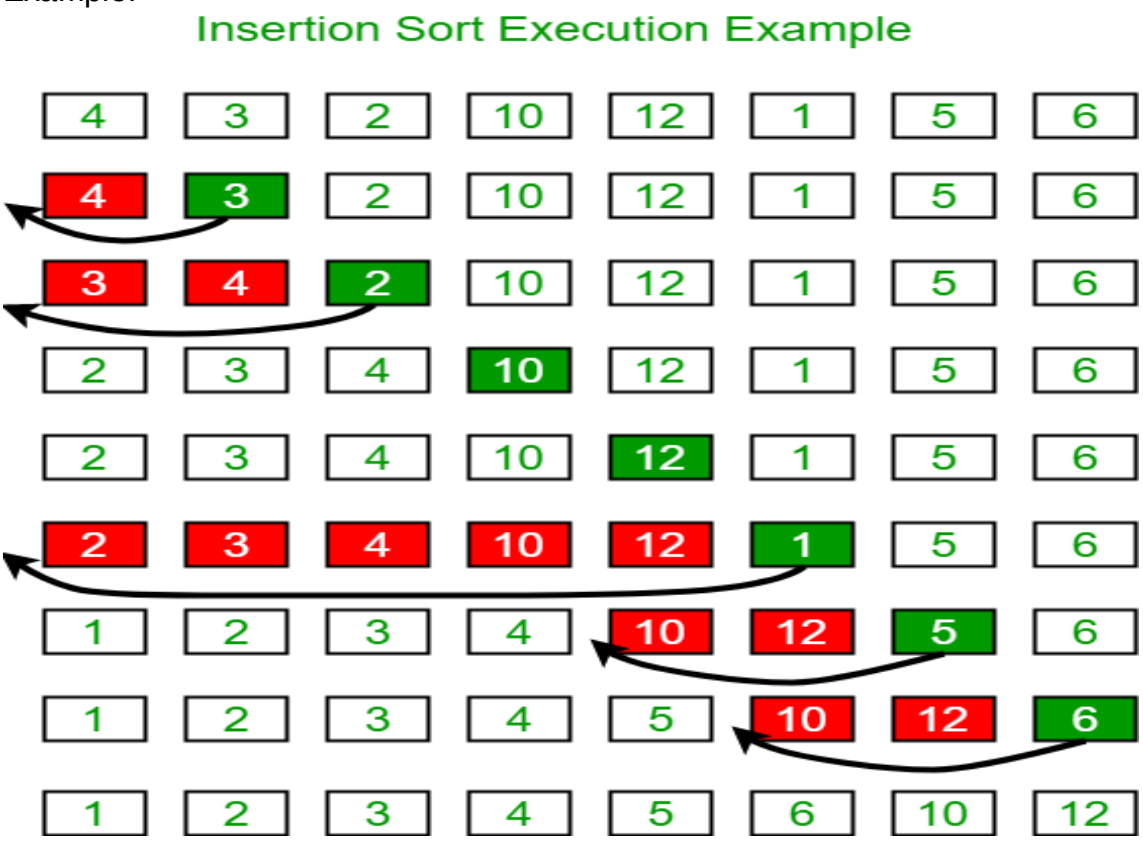
```
Sorted array:
11 12 22 25 64
```

Time Complexity: $O(n^2)$ as there are two nested loops.

3)Insertion Sort

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

Algorithm
// Sort an arr[] of size n
insertionSort(arr, n)
Loop from i = 1 to n-1.
……a) Pick element arr[i] and insert it into sorted sequence arr[0…i-1]
Example:



Another Example:
12, 11, 13, 5, 6
Let us loop for i = 1 (second element of the array) to 4 (last element of the array)

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12
11, 12, 13, 5, 6
i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13
11, 12, 13, 5, 6
i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.
5, 11, 12, 13, 6
i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.
5, 6, 11, 12, 13

```
// C program for insertion sort
#include <math.h>
#include <stdio.h>

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
	int i, key, j;
	for (i = 1; i < n; i++) {
		key = arr[i];
		j = i - 1;

		/* Move elements of arr[0..i-1], that are
		greater than key, to one position ahead
		of their current position */
		while (j >= 0 && arr[j] > key) {
			arr[j + 1] = arr[j];
			j = j - 1;
		}
		arr[j + 1] = key;
	}
}

// A utility function to print an array of size n
void printArray(int arr[], int n)
{
	int i;
	for (i = 0; i < n; i++)
		printf("%d ", arr[i]);
	printf("\n");
}

/* Driver program to test insertion sort */
int main()
{
	int arr[] = { 12, 11, 13, 5, 6 };
	int n = sizeof(arr) / sizeof(arr[0]);

	insertionSort(arr, n);
	printArray(arr, n);

	return 0;
}
```

Output:

```
5 6 11 12 13
```

Time Complexity: O(n*2)

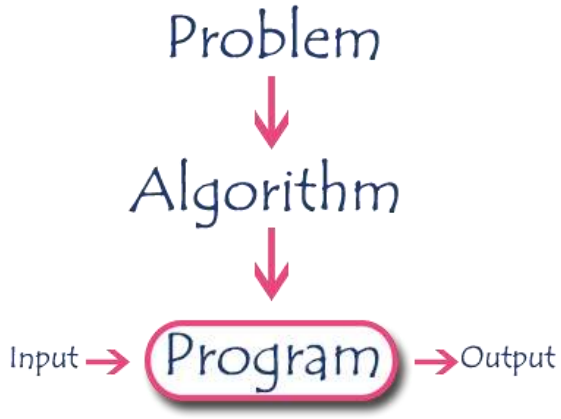
Introduction to Algorithms

What is an algorithm?

An algorithm is a step by step procedure to solve a problem. In normal language, the algorithm is defined as a sequence of statements which are used to perform a task. In computer science, an algorithm can be defined as follows...

An algorithm is a sequence of unambiguous instructions used for solving a problem, which can be implemented (as a program) on a computer.

Algorithms are used to convert our problem solution into step by step statements. These statements can be converted into computer programming instructions which form a program. This program is executed by a computer to produce a solution. Here, the program takes required data as input, processes data according to the program instructions and finally produces a result as shown in the following picture.



Specifications of Algorithms

Every algorithm must satisfy the following specifications...

1. **Input** - Every algorithm must take zero or more number of input values from external.
2. **Output** - Every algorithm must produce an output as result.
3. **Definiteness** - Every statement/instruction in an algorithm must be clear and unambiguous (only one interpretation).

4. **Finiteness -** For all different cases, the algorithm must produce result within a finite number of steps.
5. **Effectiveness -** Every instruction must be basic enough to be carried out and it also must be feasible.

Example for an Algorithm

Let us consider the following problem for finding the largest value in a given list of values.
Problem Statement : Find the largest number in the given list of numbers?
Input : A list of positive integer numbers. (List must contain at least one number).
Output : The largest number in the given list of positive integer numbers.

Consider the given list of numbers as 'L' (input), and the largest number as 'max' (Output).

Algorithm

1. **Step 1:** Define a variable 'max' and initialize with '0'.
2. **Step 2:** Compare first number (say 'x') in the list 'L' with 'max', if 'x' is larger than 'max', set 'max' to 'x'.
3. **Step 3:** Repeat step 2 for all numbers in the list 'L'.
4. **Step 4:** Display the value of 'max' as a result

Write an algorithm to find all roots of a quadratic equation $ax^2+bx+c=0$.

```
#include <stdio.h>

#include <math.h>

int main() {

    double a, b, c, discriminant, root1, root2, realPart, imagPart;

    printf("Enter coefficients a, b and c: ");

    scanf("%lf %lf %lf", &a, &b, &c);

    discriminant = b * b - 4 * a * c;

    // condition for real and different roots

    if (discriminant > 0) {

        root1 = (-b + sqrt(discriminant)) / (2 * a);

        root2 = (-b - sqrt(discriminant)) / (2 * a);

        printf("root1 = %.2lf and root2 = %.2lf", root1, root2);
```

```
    }
    // condition for real and equal roots
    else if (discriminant == 0) {
        root1 = root2 = -b / (2 * a);
        printf("root1 = root2 = %.2lf;", root1);
    }
    // if roots are not real
    else {
        realPart = -b / (2 * a);
        imagPart = sqrt(-discriminant) / (2 * a);
        printf("root1 = %.2lf+%.2lfi and root2 = %.2f-%.2fi", realPart, imagPart, realPart, imagPart);
    }
    return 0;
}
```

Max-Min Problem

Let us consider a simple problem that can be solved by divide and conquer technique.

Problem Statement

The Max-Min Problem in algorithm analysis is finding the maximum and minimum value in an array.

Solution

To find the maximum and minimum numbers in a given array a[] of size n, the following algorithm can be used.

Naive Method

Naive method is a basic method to solve any problem. In this method, the maximum and minimum number can be found separately. To find the

maximum and minimum numbers, the following straightforward algorithm can be used.

```
#include <conio.h>
int main()
{
   int a[500],i,n,min,max;
   printf("Enter size of the array : ");
   scanf("%d",&n);
   printf("Enter elements in array : ");
   for(i=0; i<n; i++)
   {
      scanf("%d",&a[i]);
   }
   min=max=a[0];
   for(i=1; i<n; i++)
   {
       if(a[i]<min)
         min=a[i];
       if(a[i]>max)
         max=a[i];
   }
    printf("minimum of array is : %d",min);
    printf("\nmaximum of array is : %d",max);
    return 0;
}
```

Analysis

The number of comparison in Naive method is **2n - 2**.

The number of comparisons can be reduced using the divide and conquer approach. Following is the technique.

Algorithm to check whether the given number is Prime or not

Aim:

Write a C program to check whether the given number is prime or not.

Algorithm:

```
Step 1: Start
Step 2: Read number n
Step 3: Set f=0
Step 4: For i=2 to n-1
Step 5: If n mod i==0 then
Step 6: Set f=1 and break
Step 7: Loop
Step 8: If f=0 then
print 'The given number is prime'
else
print 'The given number is not prime'
Step 9: Stop
```

Program code

```
#include<stdio.h>
#include<conio.h>
void main( )
{
clrscr();
int n,i,f=0;
printf("Enter the number: ");
scanf("%d",&n);
for(i=2;i<n;i++)
{
if(n%i==0)
{
f=1;
break;
}
```

```
}
if(f==0)
printf("The given number is prime");
else
printf("The given number is not prime");
getch();
}
```

Output

```
Enter the number : 5
The given number is prime
```

Linear Search Algorithm (Sequential Search)

What is Search?

Searching is a process of finding a value in a list of values. In other words, searching is the process of locating the given value position in a list of values.

Searching is the process of finding the given element in a list of elements.

Linear search algorithm finds the given element in a list of elements with **O(n)** time complexity where **n** is the total number of elements in the list. This search process starts comparing the search element with the first element in the list. If both are matched then the result is **"element found"** otherwise the search element is compared with the next element in the list. If both are matched, then the result is **"element found"**. Otherwise, repeat the same with the next element in the list until the search element is compared with the last element in the list. If that last element also doesn't match with the search element, then the result we get is **"Element not**

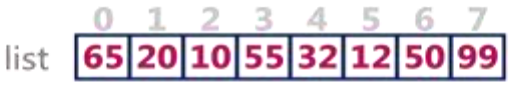
found in the list". That means, the search element is compared with all the elements in the list sequentially until the match found.

Linear search is implemented using following steps...

- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the first element in the list.
- **Step 3:** If both are matched, then display "Given element found!!!" and terminate the function
- **Step 4:** If both are not matched, then compare search element with the next element in the list.
- **Step 5:** Repeat steps 3 and 4 until the search element is compared with the last element in the list.
- **Step 6:** If the last element in the list is also not matched, then display "Element not found!!!" and terminate the function.

Example

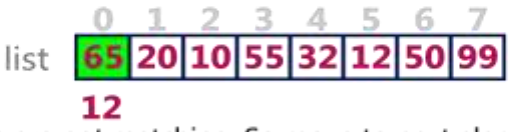
Consider the following list of element and search element...



search element **12**

Step 1:

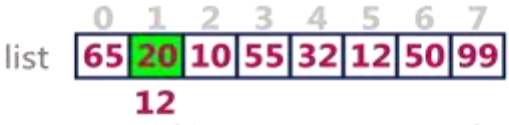
search element (12) is compared with first element (65)



Both are not matching. So move to next element

Step 2:

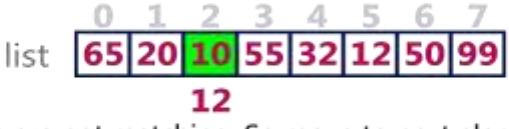
search element (12) is compared with next element (20)



Both are not matching. So move to next element

Step 3:

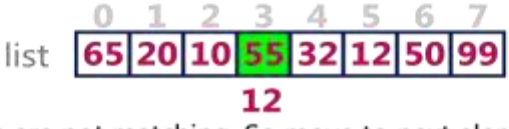
search element (12) is compared with next element (10)



Both are not matching. So move to next element

Step 4:

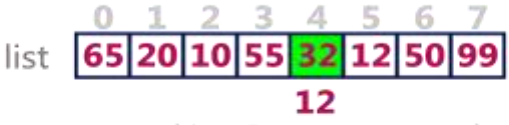
search element (12) is compared with next element (55)



Both are not matching. So move to next element

Step 5:

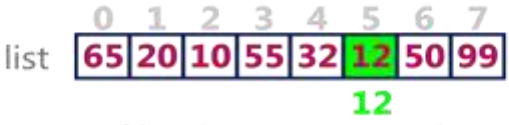
search element (12) is compared with next element (32)



Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)



Both are matching. So we stop comparing and display element found at index 5.

Program to implement Linear Search Algorithm using C.

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

void main(){
int list[20],size,i,sElement;

printf("Enter size of the list: ");
scanf("%d",&size);

printf("Enter any %d integer values: ",size);
for(i = 0; i < size; i++)
scanf("%d",&list[i]);

printf("Enter the element to be Search: ");
scanf("%d",&sElement);

  // Linear Search Logic
for(i = 0; i < size; i++)
  {
if(sElement == list[i])
     {
printf("Element is found at %d index", i);
break;
     }
  }
if(i == size)
printf("Given element is not found in the list!!!");
getch();


}
```

BINARY SEARCH ALGORITHM

Searching is a process of finding a value in a list of values. In other words, searching is the process of locating the given value position in a list of values.

Searching is the process of finding the given element in a list of elements

Binary search algorithm finds the given element in a list of elements with **O(log n)** time complexity where **n** is the total number of elements in the list. The binary search algorithm can be used only with sorted list of elements. That means, binary search can be used only with list of elements that are already arranged in order. The binary search cannot be used with

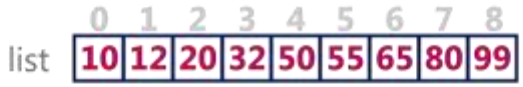
unordered list of elements. This search process starts comparing the search element with the middle element in the list. If both are matched, then the result is **"element found"**. Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for left sublist of the middle element. If the search element is larger, then we repeat the same process for right sublist of the middle element. We repeat this process until we find the search element in the list or until we are left with only one element in the sublist. And if that element also doesn't match with the search element, then we get the result as "Element not found in the list".

Binary search is implemented using following steps...

- **Step 1:** Read the search element from the user
- **Step 2:** Find the middle element in the sorted list
- **Step 3:** Compare, the search element with the middle element in the sorted list.
- **Step 4:** If both are matched, then display "Given element found!!!" and terminate the function
- **Step 5:** If both are not matched, then check whether the search element is smaller or larger than middle element.
- **Step 6:** If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- **Step 7:** If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- **Step 8:** Repeat the same process until we find the search element in the list or until the sublist contains only one element.
- **Step 9:** If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.

Example

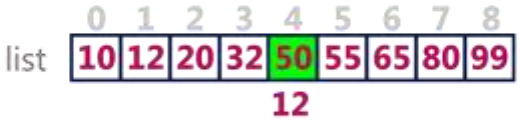
Consider the following list of elements and search element...



search element **12**

Step 1:

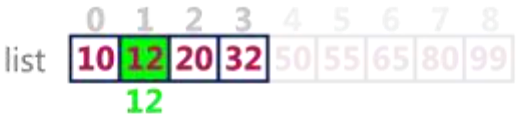
search element (12) is compared with middle element (50)



Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

Step 2:

search element (12) is compared with middle element (12)

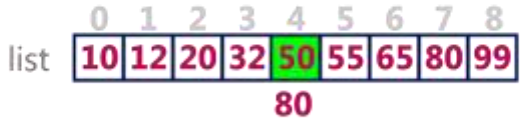


Both are matching. So the result is "Element found at index 1"

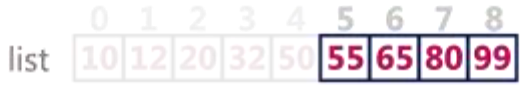
search element **80**

Step 1:

search element (80) is compared with middle element (50)



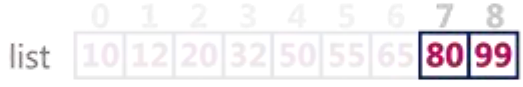
Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).



Step 2:

search element (80) is compared with middle element (65)

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).



Step 3:

search element (80) is compared with middle element (80)

Program to implement Binary Search Algorithm using C.

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

void main()
{
int first, last, middle, size, i, sElement, list[100];
clrscr();

printf("Enter the size of the list: ");
scanf("%d",&size);

printf("Enter %d integer values in Assending order\n", size);

for (i = 0; i < size; i++)
scanf("%d",&list[i]);

printf("Enter value to be search: ");
scanf("%d", &sElement);

first = 0;
last = size - 1;
middle = (first+last)/2;

while (first <= last) {
if (list[middle] <sElement)
first = middle + 1;
else if (list[middle] == sElement) {
printf("Element found at index %d.\n",middle);
break;
      }
else
last = middle - 1;

middle = (first + last)/2;
   }
if (first > last)
printf("Element Not found in the list.");
getch();
}
```

Bubble Sort Algorithm

Bubble Sort is a simple algorithm which is used to sort a given set of n elements provided in form of an array with `n` number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will **swap** both the elements, and then move on to compare the second and the third element, and so on.

If we have total `n` elements, then we need to repeat this process for `n-1` times.

It is known as **bubble sort**, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

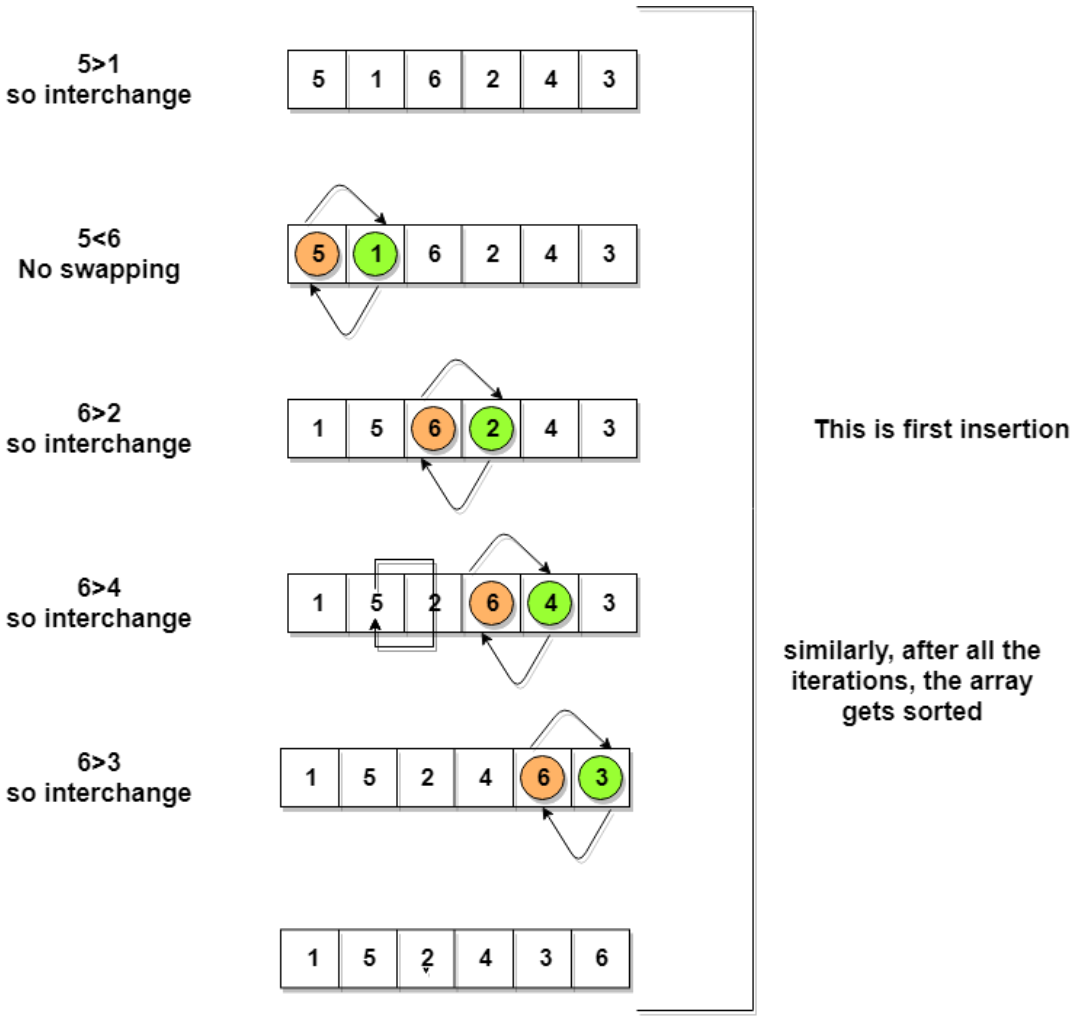
Implementing Bubble Sort Algorithm

Following are the steps involved in bubble sort(for sorting a given array in ascending order):

1. Starting with the first element(index = 0), compare the current element with the next element of the array.
2. If the current element is greater than the next element of the array, swap them.
3. If the current element is less than the next element, move to the next element. **Repeat Step 1**.

Let's consider an array with values `{5, 1, 6, 2, 4, 3}`

Below, we have a pictorial representation of how bubble sort will sort the given array.



So as we can see in the representation above, after the first iteration, 6 is placed at the last index, which is the correct position for it.

Similarly after the second iteration, 5 will be at the second last index, and so on.

Time to write the code for bubble sort:

```
// below we have a simple C program for bubble sort

#include <stdio.h>
void bubbleSort(int [], int);

void bubbleSort(int arr[], int n)
{
    int i, j, temp;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if( arr[j] > arr[j+1])
            {
                // swap the elements
                temp = arr[j];
```

```
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }

    // print the sorted array
    printf("Sorted Array: ");
    for(i = 0; i < n; i++)
    {
        printf("%d  ", arr[i]);
    }
}

int main()
{
    int arr[100], i, n, step, temp;
    // ask user for number of elements to be sorted
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);
    // input elements if the array
    for(i = 0; i < n; i++)
    {
        printf("Enter element no. %d: ", i+1);
        scanf("%d", &arr[i]);
    }
    // call the function bubbleSort
    bubbleSort(arr, n);

    return 0;
}
```

Complexity Analysis of Bubble Sort

In Bubble Sort, `n-1` comparisons will be done in the 1st pass, `n-2` in 2nd pass, `n-3` in 3rd pass and so on. So the total number of comparisons will be,

```
(n-1) + (n-2) + (n-3) + ..... + 3 + 2 + 1

Sum = n(n-1)/2

i.e O(n²)
```

Hence the **time complexity** of Bubble Sort is **O(n²)**.

The main advantage of Bubble Sort is the simplicity of the algorithm.

The **space complexity** for Bubble Sort is **O(1)**, because only a single additional memory space is required i.e. for `temp` variable.

Also, the **best case time complexity** will be **O(n)**, it is when the list is already sorted.

Following are the Time and Space complexity for the Bubble Sort algorithm.

- Worst Case Time Complexity [Big-O]: **$O(n^2)$**
- Best Case Time Complexity [Big-omega]: **(n)**
- Average Time Complexity [Big-theta]: **$O(n^2)$**
- Space Complexity: **O(1)**

Insertion Sort Algorithm

Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending).

Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

Step by Step Process

The insertion sort algorithm is performed using following steps...

- **Step 1:** Assume that first element in the list is in sorted portion and all the remaining elements are in unsorted portion.
- **Step 2:** Take first element from the unsorted portion and insert that element into the sorted portion in the order specified.
- **Step 3:** Repeat the above process until all the elements from the unsorted portion are moved into the sorted portion.

Following is the sample code for insertion sort...

Insertion Sort Logic

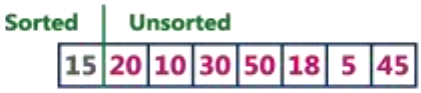
```
//Insertion sort logic
for i = 1 to size-1 {
temp = list[i];
      j = i-1;
while ((temp < list[j]) && (j > 0)) {
list[j] = list[j-1];
         j = j - 1;
      }
list[j] = temp;
   }
```

Example

Consider the following unsorted list of elements...

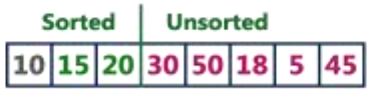
Asume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...

Move the first element 15 from unsorted portion to sorted portion of the list.

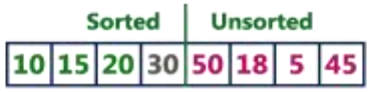


To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position

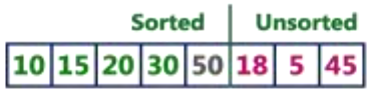
To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.



To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.



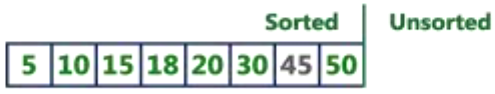
To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.



To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.

To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these element, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.

To move element 45 from unsorted to sorted portion, Compare 45 with 50 and 30. Since 45 is larger than 30, move 50 one position to the right in the list and insert 45 after 30 in the sorted list.



Unsorted portion of the list has became empty. So we stop the process. And the final sorted list of elements is as follows...

Complexity of the Insertion Sort Algorithm

To sort an unsorted list with **'n'** number of elements, we need to make

(1+2+3+.....+n-1) = (n (n-1))/2 number of comparisions in the worst case. If the list is already sorted then it requires **'n'** number of comparisions.

Worst Case : $O(n^2)$

Best Case : $\Omega(n)$

Average Case : $\Theta(n^2)$

Implementaion of Insertion Sort Algorithm using C Programming Language

```
#include<stdio.h>
#include<conio.h>
void main(){
int size, i, j, temp, list[100];

printf("Enter the size of the list: ");
scanf("%d", &size);

printf("Enter %d integer values: ", size);
for (i = 0; i < size; i++)
scanf("%d", &list[i]);

   //Insertion sort logic
for (i = 1; i < size; i++) {
temp = list[i];
      j = i - 1;
while ((temp < list[j]) && (j >= 0)) {
list[j + 1] = list[j];
         j = j - 1;
      }
list[j + 1] = temp;
   }
printf("List after Sorting is: ");
for (i = 0; i < size; i++)
printf(" %d", list[i]);
getch();
}
```

Selection Sort Algorithm

Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending). In selection sort, the first element in the list is selected and it is compared repeatedly with all the remaining elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped so that first position is filled with smallest element in the sorted order. Next we select the element at second position in the list and it is compared with all the remaining elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated till the entire list is sorted.

Step by Step Process

The selection sort algorithm is performed using following steps...

- **Step 1 -** Select the first element of the list (i.e., Element at first position in the list).
- **Step 2:** Compare the selected element with all the other elements in the list.
- **Step 3:** In every comparision, if any element is found smaller than the selected element (for Ascending order), then both are swapped.
- **Step 4:** Repeat the same procedure with element in the next position in the list till the entire list is sorted.

Following is the sample code for selection sort...

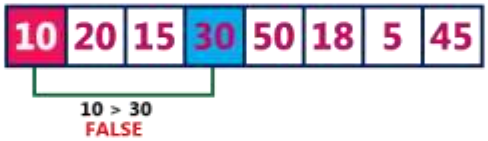
Selection Sort Logic

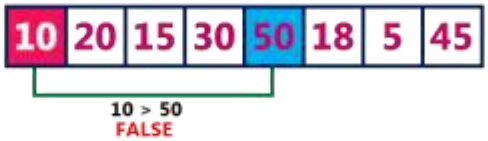
```
//Selection sort logic
for(i=0; i<size; i++){
for(j=i+1; j<size; j++){
if(list[i] > list[j])
           {
temp=list[i];
list[i]=list[j];
list[j]=temp;
           }
      }
  }
```

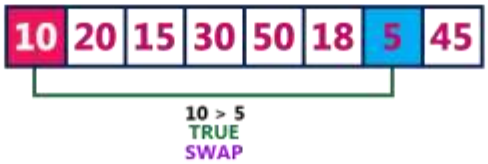
Consider the following unsorted list of elements...

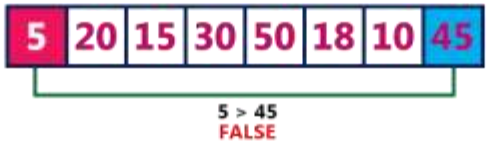
Iteration #1

Select the first position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.









List after 1st iteration

Iteration #2

Select the second position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 2nd iteration

Iteration #3

Select the third position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 3rd iteration

Iteration #4

Select the fourth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

Complexity of the Selection Sort Algorithm

To sort an unsorted list with **'n'** number of elements, we need to make **((n-1)+(n-2)+(n-3)+.....+1) = (n (n-1))/2** number of comparisions in the worst case. If the list is already sorted then it requires **'n'** number of comparisions.

Worst Case : $O(n^2)$

Best Case : $\Omega(n^2)$

Average Case : $\Theta(n^2)$

Implementaion of Selection Sort Algorithm using C Programming Language

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

void main(){

int size,i,j,temp,list[100];
clrscr();

printf("Enter the size of the List: ");
scanf("%d",&size);

printf("Enter %d integer values: ",size);
for(i=0; i<size; i++)
scanf("%d",&list[i]);

  //Selection sort logic

for(i=0; i<size; i++){
for(j=i+1; j<size; j++){
if(list[i] > list[j])
           {
temp=list[i];
list[i]=list[j];
list[j]=temp;
           }
      }
  }

printf("List after sorting is: ");
for(i=0; i<size; i++)
printf(" %d",list[i]);

getch();
}
```

Performance Analysis

What is Performance Analysis of an algorithm?

If we want to go from city "A" to city "B", there can be many ways of doing this. We can go by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one which suits us. Similarly, in computer science there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one. Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem. When there are multiple alternative algorithms to solve a problem, we analyse them and pick the one which is best suitable for our requirements. Formal definition is as follows...

Performance of an algorithm is a process of making evaluative judgment about algorithms.

It can also be defined as follows...

Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.

That means when we have multiple algorithms to solve a problem, we need to select a suitable algorithm to solve that problem.
We compare algorithms with each other which are solving same problem, to select the best algorithm. To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, execution speed of that algorithm, easy to understand, easy to implement, etc.,

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

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem? Etc.,

When we want to analyze an algorithm, we consider only the space and time required by that particular algorithm and we ignore all the remaining elements. Based on this information, performance analysis of an algorithm can also be defined as follows...

Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.

Performance analysis of an algorithm is performed by using the following measures...

1. Space required to complete the task of that algorithm (**Space Complexity**). It includes program space and data space
2. Time required to complete the task of that algorithm (**Time Complexity**)

Space Complexity

What is Space complexity?

When we design an algorithm to solve a problem, it needs some computer memory to complete its execution. For any algorithm, memory is required for the following purposes...

1. To store program instructions.
2. To store constant values.
3. To store variable values.
4. And for few other things like funcion calls, jumping statements etc,.

Space complexity of an algorithm can be defined as follows...

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.

Generally, when a program is under execution it uses the computer memory for THREE reasons. They are as follows...

1. **Instruction Space:** It is the amount of memory used to store compiled version of instructions.
2. **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.
3. **Data Space:** It is the amount of memory used to store all the variables and constants.

To calculate the space complexity, we must know the memory required to store different datatype values (according to the compiler). For example, the C Programming Language compiler requires the following...

1. 2 bytes to store Integer value.
2. 4 bytes to store Floating Point value.
3. 1 byte to store Character value.
4. 6 (OR) 8 bytes to store double value.

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

Consider the following piece of code...

Example 2

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

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

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

If the amount of space required by an algorithm is increased with the increase of input value, then that space complexity is said to be Linear Space Complexity.

Time Complexity

What is Time complexity?

Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer time required is called time complexity. Time complexity of an algorithm can be defined as follows...

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

Generally, running time of an algorithm depends upon the following...

1. Whether it is running on Single processor machine or Multi processor machine.
2. Whether it is a 32 bit machine or 64 bit machine.
3. Read and Write speed of the machine.
4. The amount of time required by an algorithm to perform Arithmetic operations, logical operations, return value and assignment operation etc.
5. Input data

Calculating Time Complexity of an algorithm based on the system configuration is a very difficult task because, the configuration changes from one system to another system. To solve this problem, we must assume a model machine with specific configuration. So that, we can able to calculate generalized time complexity according to that model machine.

To calculate time complexity of an algorithm, we need to define a model machine. Let us assume a machine with following configuration...

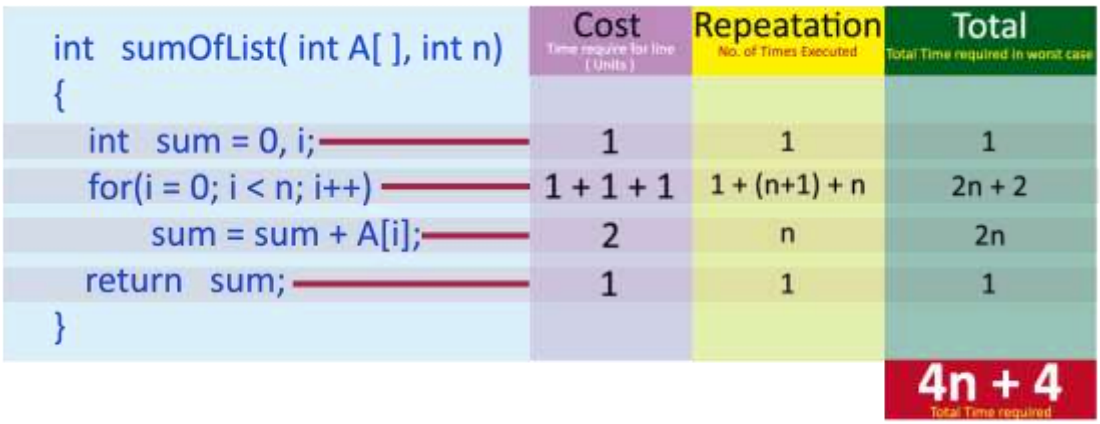
1. It is a Single processor machine
2. It is a 32 bit Operating System machine
3. It performs sequential execution
4. It requires 1 unit of time for Arithmetic and Logical operations
5. It requires 1 unit of time for Assignment and Return value
6. It requires 1 unit of time for Read and Write operations

If any program requires fixed amount of time for all input values then its time complexity is said to be Constant Time Complexity.

Consider the following piece of code...

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

For the above code, time complexity can be calculated as follows...



In above calculation **Cost** is the amount of computer time required for a single operation in each line.

Repeatation is the amount of computer time required by each operation for all its repeatations.

Total is the amount of computer time required by each operation to execute.

So above code requires **'4n+4' Units** of computer time to complete the task. Here the exact time is not fixed. And it changes based on the **n** value. If we increase the **n** value then the time required also increases linearly.

Totally it takes '4n+4' units of time to complete its execution and it is *Linear Time Complexity*.

If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be Linear Time Complexity.

Asymptotic Notations

What is Asymptotic Notation?

Whenever we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm. But when we calculate complexity of an algorithm it does not provide exact amount of resource required. So instead of taking exact amount of resource we represent that complexity in a general form (Notation) which produces the basic nature of that algorithm. We use that general form (Notation) for analysis process.

Asymptotic notation of an algorithm is a mathematical representation of its complexity.

Note - In asymptotic notation, when we want to represent the complexity of an algorithm, we use only the most significant terms in the complexity of that algorithm and ignore least significant terms in the complexity of that algorithm (Here complexity can be Space Complexity or Time Complexity).

For example, consider the following time complexities of two algorithms...

- **Algorithm 1 : $5n^2 + 2n + 1$**
- **Algorithm 2 : $10n^2 + 8n + 3$**

Generally, when we analyze an algorithm, we consider the time complexity for larger values of input data (i.e. **'n'** value). In above two time complexities, for larger value of **'n'** the term **'2n + 1'** in algorithm 1 has least significance than the term **'$5n^2$'**, and the term **'8n + 3'** in algorithm 2 has least significance than the term **'$10n^2$'**.

Here, for larger value of **'n'** the value of most significant terms (**$5n^2$** and **$10n^2$**) is very larger than the value of least significant terms (**2n + 1** and **8n + 3**). So for larger value of **'n'** we ignore the least significant terms to represent overall time required by an algorithm. In asymptotic notation, we use only the most significant terms to represent the time complexity of an algorithm.

Majorly, we use THREE types of Asymptotic Notations and those are as follows...

1. **Big - Oh (O)**
2. **Big - Omega (Ω)**
3. **Big - Theta (Θ)**

Big - Oh Notation (O)

Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.

That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values.

That means Big - Oh notation describes the worst case of an algorithm time complexity.

Big - Oh Notation can be defined as follows...

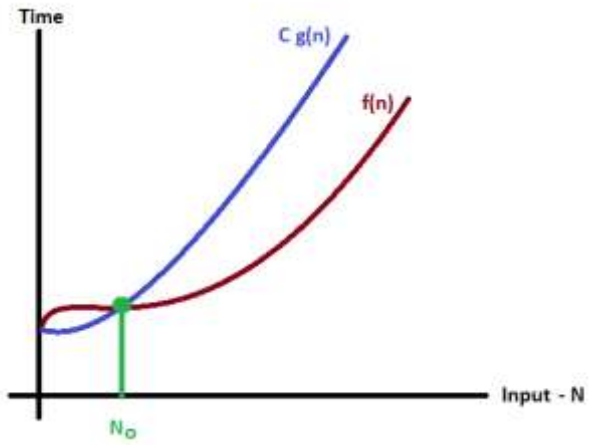
Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as O(g(n)).

or

A function f(n)=O.g(n) iff f(n) <= C g(n) for all n >= n0, C > 0 and n0 >= 1. Then we can represent f(n) as O(g(n)).

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

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.

Example

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= C g(n)** for all values of **C > 0** and **n_0>= 1**

f(n) <= C g(n)

⇒3n + 2 <= C n

Above condition is always TRUE for all values of **C = 4** and **n >= 2**.

By using Big - Oh notation we can represent the time complexity as follows...

3n + 2 = O(n)

Big - Omege Notation (Ω)

Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.

That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity.

Big - Omega Notation can be defined as follows...

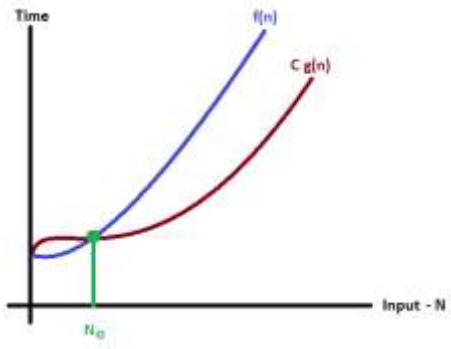
Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as Ω(g(n)).

Or

A function f(n)= Ω g(n) iff f(n) >= C g(n) for all n >= n0, C > 0 and n0 >= 1. Then we can represent f(n) as O(g(n)).

f(n) = Ω(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C g(n) is less than f(n) which indicates the algorithm's lower bound.

Example

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n_0>= 1**

f(n) >= C g(n)

⇒3n + 2 >= C n

Above condition is always TRUE for all values of **C = 1** and **n >= 1**.

By using Big - Omega notation we can represent the time complexity as follows...

3n + 2 = Ω(n)

Big - Theta Notation (Θ)

Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.

That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.

Big - Theta Notation can be defined as follows...

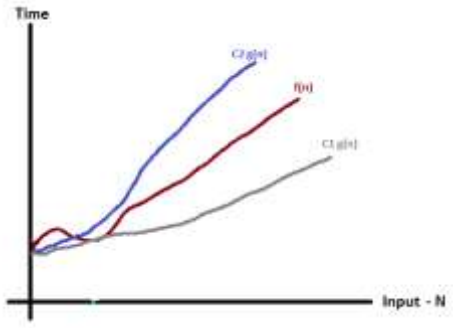
Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If C_1 g(n) <= f(n) <= C_2 g(n) for all n >= n_0, C_1 > 0, C_2 > 0 and n_0 >=1. Then we can represent f(n) as Θ(g(n)).

Or

A function f(n)= Ω g(n) iff g(n) <= f(n) <= C_2 g(n) for all n >= n_0, C_1 > 0, C_2 > 0 and n_0 >=1. Then we can represent f(n) as Θ(g(n)).

f(n) = Θ(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C_1 g(n) is less than f(n) and C_2 g(n) is greater than f(n) which indicates the algorithm's average bound.

Example

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **Θ(g(n))** then it must satisfy **C_1 g(n) <= f(n) <= C_2 g(n)** for all values of **C_1 > 0, C_2 > 0** and **n_0>= 1**

C_1 g(n) <= f(n) <= C_2 g(n)

⇒C_1 n <= 3n + 2 <= C_2 n

Above condition is always TRUE for all values of **C_1 = 1, C_2 = 4** and **n >= 2**.

By using Big - Theta notation we can represent the time compexity as follows...

3n + 2 = Θ(n)