In programming, algorithm is a set of well defined instructions in sequence to solve the problem.

Qualities of a good algorithm

1. Input and output should be defined precisely.
2. Each steps in algorithm should be clear and unambiguous.
3. Algorithm should be most effective among many different ways to solve a problem.
4. An algorithm shouldn't have computer code. Instead, the algorithm should be written in such a way that, it can be used in similar programming languages.

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
```

Write an algorithm to check whether a number entered by user is prime or not.

```
Step 1: Start
Step 2: Declare variables n,i,flag.
Step 3: Initialize variables
        flag←1
        i←2
Step 4: Read n from user.
Step 5: Repeat the steps until i<(n/2)
     5.1 If remainder of n÷i equals 0
            flag←0
            Go to step 6
     5.2 i←i+1
Step 6: If flag=0
           Display n is not prime
        else
           Display n is prime
```

```
Step 7: Stop
```

Write an algorithm to find the factorial of a number entered by user.

```
Step 1: Start
Step 2: Declare variables n,factorial and i.
Step 3: Initialize variables
          factorial←1
          i←1
Step 4: Read value of n
Step 5: Repeat the steps until i=n
     5.1: factorial←factorial*i
     5.2: i←i+1
Step 6: Display factorial
Step 7: Stop
```

Write an algorithm to add two numbers entered by user.

```
Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
        sum←num1+num2
Step 5: Display sum
Step 6: Stop
```

Write an algorithm to find the largest among three different numbers entered by user.

```
Step 1: Start
Step 2: Declare variables a,b and c.
Step 3: Read variables a,b and c.
Step 4: If a>b
           If a>c
              Display a is the largest number.
           Else
              Display c is the largest number.
        Else
           If b>c
              Display b is the largest number.
           Else
              Display c is the greatest number.
```

```
Step 5: Stop
```

Write an algorithm for finding minimum and maximum numbers of a given set

step1: start
step2: Declare the variables a[], n, i, min, max
step3: read n value from the user and enter values in the array.
step4: initialize min=a[0] and max=a[0]
step5: for i=0 to n-1
    if(min>a[i])
    min=a[i];
step6:for i=0 to n-1
    if(max<a[i])
    max=a[i];
step7: display min as the smallest element
step8: display max as the largest element
step9: stop

What is Search?

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

Linear Search Algorithm (Sequential Search Algorithm)

Linear search algorithm finds given element in a list of elements with **O(n)** time complexity where **n** is total number of elements in the list. This search process starts comparing of search element with the first element in the list. If both are matching then results with element found otherwise search element is compared with 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 search element is compared with last element in the list, if that last element also doesn't match, then the result is "Element not found in the list". That means, the search element is compared with element by element in the list.

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 matching, then display "Given element found!!!" and terminate the function
Step 4 - If both are not matching, 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 doesn't match, 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

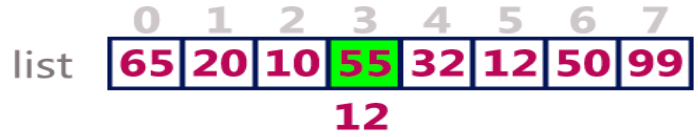
Step 3:

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

Both are not matching. So move to next element

Step 4:

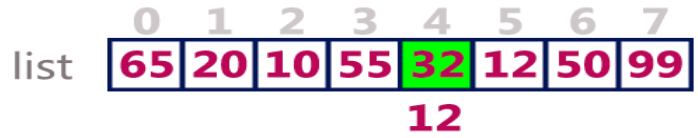
search element (12) is compared with next element (55)



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 for linear search:

```
void main()
{
int a[50],n,i,e;
clrscr();
printf("enter no of elements:\n");
scanf("%d",&n);
printf("enter elements into the list:\n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("enter search element:\n");
scanf("%d",&e);
for(i=0;i<n;i++)
{
if(e==a[i])
{
printf("element found");
getch();
exit();
}
}
printf("element not found");
getch();
}
```

Binary Search Algorithm

Binary search algorithm finds given element in a list of elements with **O(log n)** time complexity where **n** is total number of elements in the list. The binary search algorithm can be used with only sorted list of element. That means, binary search can be used only with lkist of element which are already arraged in a order. The binary search can not be used for list of element which are in random order. This search process starts comparing of 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 left with a sublist of only one element. And if that element also doesn't match with the search element, then the result is "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 matching, then display "Given element found!!!" and terminate the function.

Step 5 - If both are not matching, 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 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.

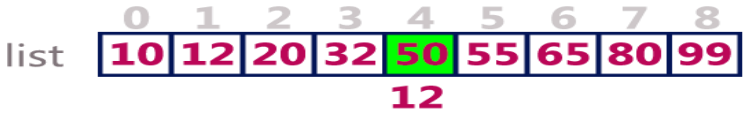
Example

Consider the following list of element 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:

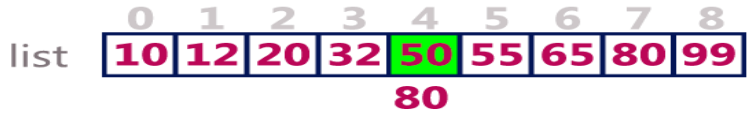
search element (12) is compared with middle element (12)

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)

Both are not matching. So the result is "Element found at index 7"

Program for binary search:

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

void main()
{
int a[50],n,i,l,u,mid,se;
clrscr();
printf("enter no of elements:\n");
scanf("%d",&n);
printf("enter elements in to the list:\n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("enter search element:\n");
scanf("%d",&se);
l=0;
u=n-1;
mid=(l+u)/2;
while(l<=u)
{
if(se==a[mid])
{
printf("element found");
break;
}
if(se>a[mid])
l=mid+1;
else
u=mid-1;
mid=(l+u)/2;
}
if(l>u)
printf("element not found");
getch();
}
```

Bubble Sort

In this method, to arrange elements in ascending order, to begin with the 0^{th} element is compared with the 1^{st} element. If it is found to be greater than the 1^{st} element then they are interchanged. Then the 1^{st} element is compared with the 2^{nd} element, if it is found to be greater, then they are interchanged. In the same way all the elements (excluding last) are compared with their next element and are interchanged if required. This is the first iteration and on completing this iteration the largest element gets placed at the last position. Similarly, in the second iteration the comparisons are made till the last but one element and this time the second largest element gets placed at the second last position in the list. As a result, after all the iterations the list becomes a sorted list. This can be explained with the help of Figure 10-4.

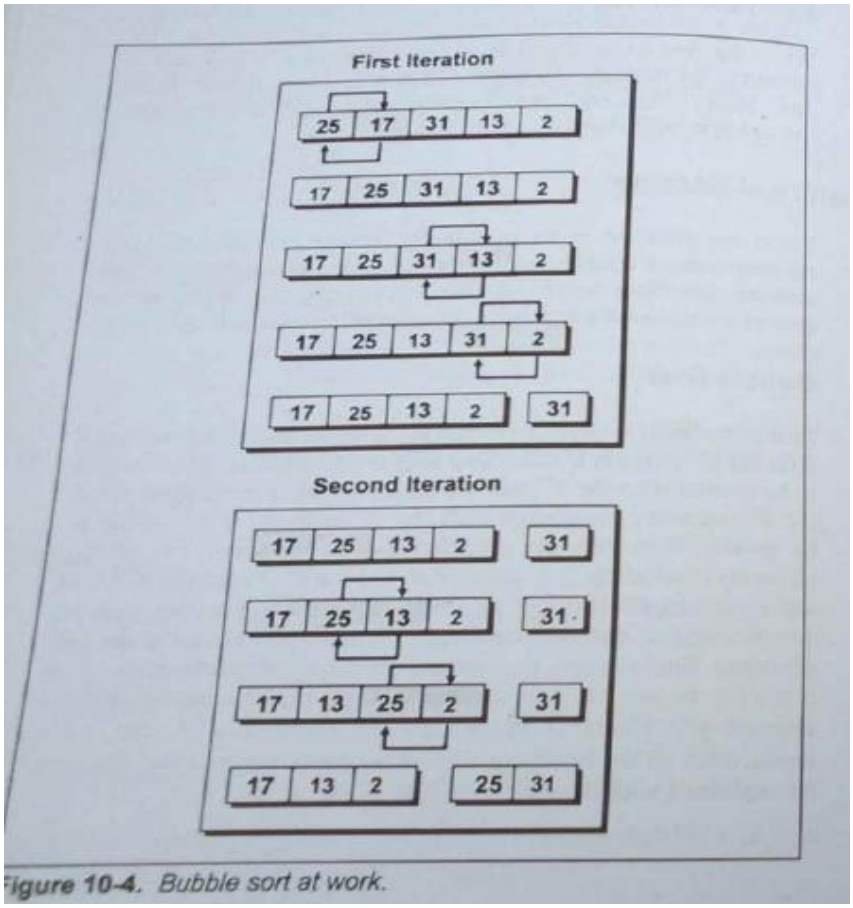


Figure 10-4. *Bubble sort at work.*

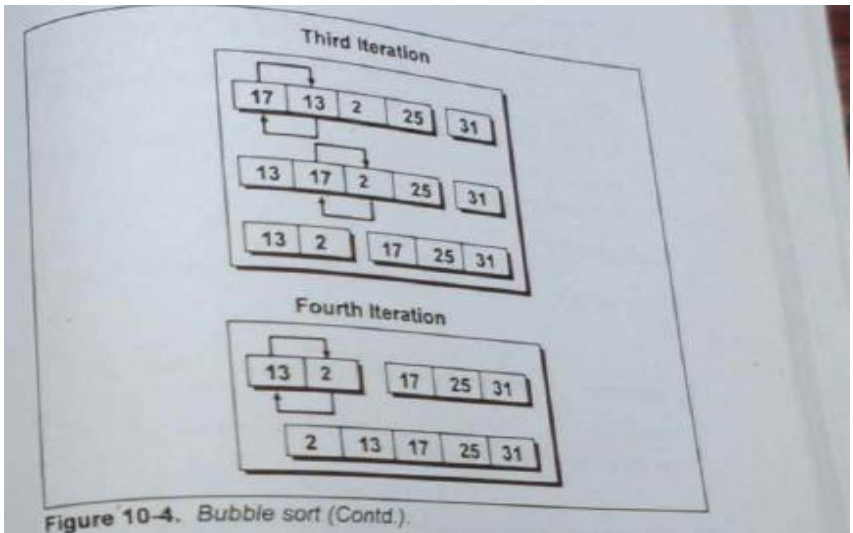


Figure 10-4. *Bubble sort (Contd.).*

Suppose an array **arr** consists of 5 numbers. The bubble sort algorithm works as follows:

(a) In the first iteration the 0^{th} element 25 is compared with 1^{st} element 17 and since 25 is greater than 17, they are interchanged.

(b) Now the 1^{st} element 25 is compared with 2^{nd} element 31. But 25 being less than 31 they are not interchanged.

(c) This process is repeated until $(n-2)^{nd}$ element is compared with $(n-1)^{th}$ element. During the comparison if $(n-2)^{nd}$ element is found to be greater than the $(n-1)^{th}$, then they are interchanged, otherwise not.

(d) At the end of the first iteration, the $(n-1)^{th}$ element holds the largest number.

(e) Now the second iteration starts with the 0^{th} element 17. The above process of comparison and interchanging is repeated but this time the last comparison is made between $(n-3)^{rd}$ and $(n-2)^{nd}$ elements.

(f) If there are **n** number of elements then **(n - 1)** iterations need to be performed.

Complexity

The complexity of the bubble sort algorithm is tabulated in Table 10-2.

Algorithm	Worst Case	Average Case	Best Case
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Table 10-2. *Complexity of bubble sort.*

Program for bubble sort:

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

void main()
{
int a[50],n,i,j,temp;
clrscr();
printf("enter no of elements:\n");
scanf("%d",&n);
printf("enter elements in to the list:\n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}

for(i=0;i<n-1;i++)
{
for(j=0;j<n;j++)
{
if(a[j]>a[j+1])
{
temp=a[j];
```

```
a[j]=a[j+1];
a[j+1]=temp;
}
}
}
printf("the sorted list:\n");
for(i=0;i<n;i++)
{
printf("%d\t",a[i]);
}
getch();
}
```

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 - Asume that first element in the list is in sorted portion of the list and remaining all elements are in unsorted portion.

Step 2: Consider first element from the unsorted list and insert that element into the sorted list in order specified.

Step 3: Repeat the above process until all the elements from the unsorted list are moved into the sorted list.

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.

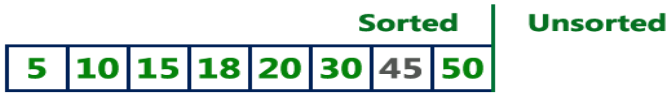
To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.

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 a 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 already sorted, then it requires **'n'** number of comparisions.

Worst Case: O(n^2)
Best Case : n-1
Average Case : O(n^2)

Program for insertion sort:

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

void main()
{
int a[50],n,i,j,k,temp;
clrscr();
printf("enter no of elements:\n");
scanf("%d",&n);
printf("enter elements in to the list:\n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}

for(i=1;i<n;i++)
{
for(j=0;j<i;j++)
{
if(a[j]>a[i])
{
temp=a[j];
a[j]=a[i];
for(k=i;k>j;k--)
a[k]=a[k-1];
a[k+1]=temp;
}
}
}

printf("the sorted list:\n");
for(i=0;i<n;i++)
{
printf("%d\t",a[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 remaining all the elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped. Then we select the element at second position in the list and it is compared with remaining all 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 other elements in the list.

Step 3: For every comparision, if any element is smaller than selected element (for Ascending order), then these two are swapped.

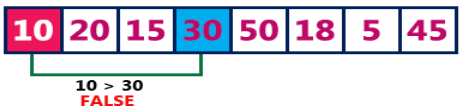
Step 4: Repeat the same procedure with next position in the list till the entire list is sorted.

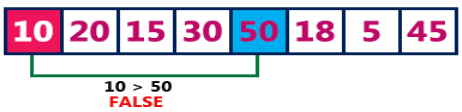
Example

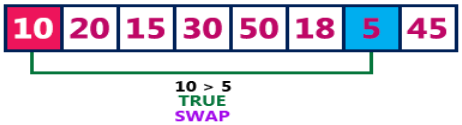
Consider the following unsorted list of elements...

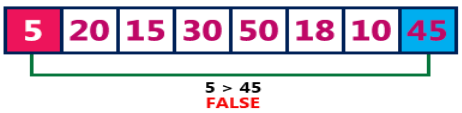
Iteration #1

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.

List after 4th iteration

Iteration #5

Select the fifth 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 5th iteration

Iteration #6

Select the sixth 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 6th iteration

Iteration #7

Select the seventh 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 7th iteration

Final sorted list

Complexity of the Selection Sort Algorithm

To sort a 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 already sorted, then it requires **'n'** number of comparisions.

Worst Case: $O(n^2)$
Best Case : $O(n^2)$
Average Case : $O(n^2)$

Program for selection sort:

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

void main()
{
int a[50],n,i,j,temp;
clrscr();
printf("enter no of elements:\n");
scanf("%d",&n);
printf("enter elements in to the list:\n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}

for(i=0;i<n-1;i++)
{
for(j=i+1; j<n;j++)
{
if(a[i]<a[j])
{
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
}
}
printf("the sorted list:\n");
for(i=0;i<n;i++)
{
printf("%d\t",a[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**)

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.

Note - When we want to perform analysis of an algorithm based on its Space complexity, we consider only Data Space and ignore Instruction Space as well as Environmental Stack. That means we calculate only the memory required to store Variables, Constants, Structures, etc.,

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...

Example 1

```
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.

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

Note - When we calculate time complexity of an algorithm, we consider only input data and ignore the remaining things, as they are machine dependent. We check only, how our program is behaving for the different input values to perform all the operations like Arithmetic, Logical, Return value and Assignment etc.,

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...

Example 1

```
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...

Example 2

```
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...

	Cost Time require for line (Units)	Repeatation No. of Times Executed	Total Total Time required in worst case
int sumOfList(int A[ ], int n)			
{			
int sum = 0, i;	1	1	1
for(i = 0; i < n; i++)	1 + 1 + 1	1 + (n+1) + n	2n + 2
sum = sum + A[i];	2	n	2n
return sum;	1	1	1
}			
			4n + 4 Total Time required

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)

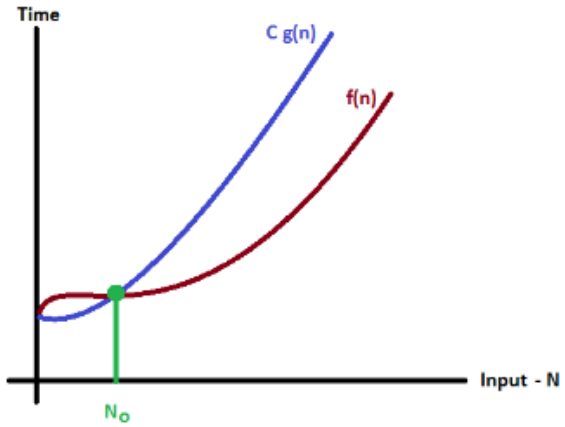
Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity. That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity. Big - 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



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)

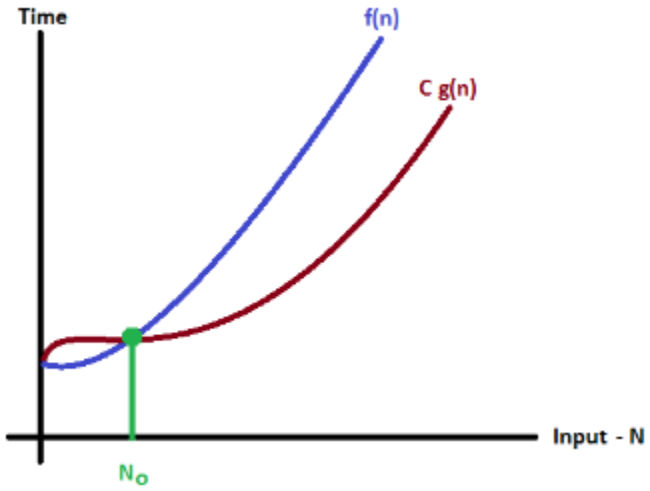
Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity. That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity. 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)

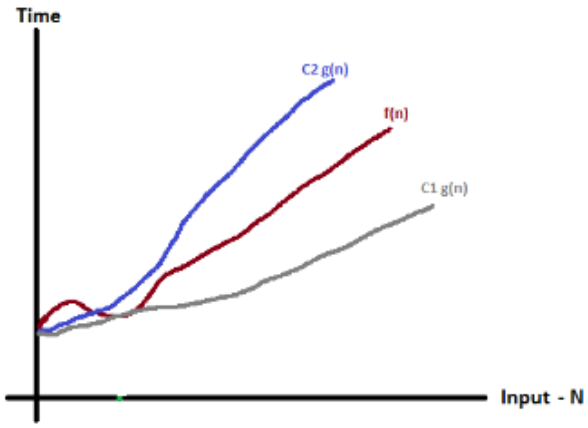
Big - Theta Notation (Θ)

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)