UNIT-V

Introduction to Algorithms: Algorithms for finding roots of quadratic equations, finding Minimum and maximum numbers of a given set, finding if a number is prime number, etc. Basic searching in an array of elements (linear and binary search techniques), Basic algorithms to sort array of elements (Bubble, Insertion and Selection sort algorithms), Basic concept of order of complexity through the example programs

1. Explain the algorithm for finding roots of quadratic equation.

Ans:

The general form of a quadratic equation is **($ax^2 + bx + c = 0$).** The highest degree in a quadratic equation is 2. Hence, a quadratic equation will have two roots.

Solving a quadratic equation:
The formula to find the roots of a quadratic equation is given as follows

x = [-b +/- sqrt(-b^2 - 4ac)]/2a

The discriminant of the quadratic equation is

k = (**b^2 - 4ac).**

Depending upon the nature of the discriminant, the roots can be found in different ways.

1. If the **discriminant is positive**, then there are two distinct real roots.
2. If the **discriminant is zero**, then the two roots are equal.
3. If the **discriminant is negative**, then there are two distinct complex roots.

Algorithm to find all the roots of a quadratic equation:

1. Input the value of a, b, c.
2. Calculate k = b*b - 4*a*c
3. If (d < 0)

   Display "Roots are Imaginary, calculater1 = (-b +i*sqrt(k))/ 2a and r2 =(b + i*sqrt(k))/ 2a.

   else if (d = 0)

   Display "Roots are Equal" and calculate r1 = r2 = (-b / 2*a)

   else

   Display "Roots are real and calculate r1 = -b + sqrt(k) / 2*a and r2 = -b - sqrt(k) / 2*a
4. Print r1 and r2.
5. End the algorithm

2. Write a C Program for finding roots of quadratic equation.

Ans :

```
#include <stdio.h>
#include <math.h>
int main()
{
   double a, b, c, discriminant, root1, root2, realPart, imaginaryPart;
   printf("Enter coefficients a, b and c: ");
   scanf("%lf %lf %lf",&a, &b, &c);
   discriminant = b*b-4*a*c;
   if (discriminant > 0)       // condition for real and different roots
   {
      // sqrt() function returns square root
      root1 = (-b+sqrt(discriminant))/(2*a);
      root2 = (-b-sqrt(discriminant))/(2*a);
      printf("root1 = %.2lf and root2 = %.2lf",root1 , root2);
   }
   else if (discriminant == 0)    //condition for real and equal roots
   {
      root1 = root2 = -b/(2*a);
      printf("root1 = root2 = %.2lf;", root1);
   }
   else     // if roots are not real
   {
      realPart = -b/(2*a);
      imaginaryPart = sqrt(-discriminant)/(2*a);
     printf("root1 = %.2lf+%.2lfi and root2 = %.2f-%.2fi", realPart, imaginaryPart, realPart, imaginaryPart);
   }
   return 0;
}
```

3. Define algorithm and write algorithm to generate prime number series between m and n, where m and n are integers.

Ans:

A step by step procedure is called algorithm.

Prime number is a number which is exactly divisible by one and itself only.

Algorithm:

Step 1: start

Step 2: read m and n

Step 3: initialize i = m,

Step 4: if i <= n goto step 5

If not goto step 10

Step 5: initialize j = 1,c=0

Step 6: if j <= i do as the follow.

If not goto step 7

i)if i%j == 0 increment c

ii) increment j

iii) goto Step 6

Step 7: if c == 2 print i

Step 8: increment i

Step 9: goto step 4

Step 10: stop

4. Write a C Program to generate prime number series between m and n, where m and n are integers.

Ans:

Program:

```
#include<stdio.h>
void main()
{
  int m,n, i, j, count;
  printf("Prime no.series\n");
  printf("Enter m and n values\n");
  scanf("%d%d",&m, &n);
  printf("The prime numbers between %d to %d\n",m,n);
  for(i = m; i <= n; i++)
  {
        count = 0;
         for(j = 1; j <=i; j++)
         if(i % j == 0)
        {
                  count++;
        }
        if(count == 2)
        {
                  printf("%d\t", i);
        }
  }
}
```

Input & Output:

```
Prime no. series
Enter m and n Values
  2     10
The prime numbers between 2 to 10
2  3  5  7
```

5. Write an algorithm and program to find the minimum and maximum number in a given set.

Ans:

Problem Description: Given an array A[] of size n, you need to find the maximum and minimum element present in the array. Your algorithm should make the minimum number of comparisons.

For Example:

Input: A[] = { 4, 2, 0, 8, 20, 9, 2}

Output: Maximum: 20, Minimum: 0

Searching linearly: Increment the loop by 1

We initialize both minimum and maximum element to the first element and then traverse the array, comparing each element and update minimum and maximum whenever necessary.

Pseudo-Code:

```
int[] getMinMax(int A[], int n)
{
   int max = A[0]
   int min = A[0]
   for ( i = 1 to n-1 )
   {
      if ( A[i] > max )
         max = A[i]
      else if ( A[i] < min )
         min = A[i]
   }
   // By convention, let ans[0] = maximum and ans[1] = minimum
   int ans[2] = {max, min};
   return ans
}
```

Program:

```
#include <stdio.h>
#define MAX_SIZE 100   // Maximum array size
int main()
{
   int arr[MAX_SIZE];
   int i, max, min, size;
   printf("Enter size of the array: ");    /* Input size of the array */
   scanf("%d", &size);
```

```
    printf("Enter elements in the array: ");   /* Input array elements */
    for(i=0; i<size; i++)
    {
        scanf("%d", &arr[i]);
    }
    max = arr[0];   /* Assume first element as maximum and minimum */
    min = arr[0];
    /* Find maximum and minimum in all array elements.*/
    for(i=1; i<size; i++)
    {
        if(arr[i] > max) /* If current element is greater than max */
        {
            max = arr[i];
        }
        if(arr[i] < min)  /* If current element is smaller than min */
        {
            min = arr[i];
        }
    }
    /* Print maximum and minimum element */
    printf("Maximum element = %d\n", max);
    printf("Minimum element = %d", min);
    return 0;
}
```

6. What is meant by searching? Explain the linear search algorithm with example?

Ans:

Searching:

Searching is an operation or a technique that helps finds the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching techniques that is being followed in the data structure is listed below:

- Linear Search or Sequential Search
- Binary Search

Linear Search:

- Linear Search is the simplest searching algorithm.
- It traverses the array sequentially to locate the required element.
- It searches for an element by comparing it with each element of the array one by one.
- So, it is also called as Sequential Search.

Linear Search Algorithm is applied when-

- No information is given about the array.
- The given array is unsorted or the elements are unordered.
- The list of data items is smaller.

Linear Search Algorithm:

Consider-

- There is a linear array 'a' of size 'n'.
- Linear search algorithm is being used to search an element 'item' in this linear array.
- If search ends in success, it sets loc to the index of the element otherwise it sets loc to -1.

```
Linear_Search (a , n , item , loc)
{
    for i = 0 to (n - 1)
    {
        if (a[i] == item)
        {
            set loc = i
            Exit
        }
    }
set loc = -1
}
```

Linear Search Example:

Consider-

- We are given the following linear array.
- Element 15 has to be searched in it using Linear Search Algorithm.

Linear Search Example

Now,

- Linear Search algorithm compares element 15 with all the elements of the array one by one.
- It continues searching until either the element 15 is found or all the elements are searched.

Linear Search Algorithm works in the following steps-

Step-01:

- It compares element 15 with the 1st element 92.
- Since 15 ≠ 92, so required element is not found.
- So, it moves to the next element.

Step-02:

- It compares element 15 with the 2nd element 87.
- Since 15 ≠ 87, so required element is not found.
- So, it moves to the next element.

Step-03:

- It compares element 15 with the 3rd element 53.
- Since 15 ≠ 53, so required element is not found.
- So, it moves to the next element.

Step-04:

- It compares element 15 with the 4th element 10.
- Since 15 ≠ 10, so required element is not found.
- So, it moves to the next element.

Step-05:

- It compares element 15 with the 5th element 15.
- Since 15 = 15, so required element is found.
- Now, it stops the comparison and returns index 4 at which element 15 is present.

7. Write a C Program to implement Linear Search Algorithm?

Ans :

```
#include <stdio.h>
 int linear_search(int[],int.int);
 int main()
{
   int array[100], search, c, n, position;
   printf("Input number of elements in array\n");
   scanf("%d", &n);
   printf("Input %d numbers\n", n);
   for (c = 0; c < n; c++)
      scanf("%d", &array[c]);

   printf("Input a number to search\n");
   scanf("%d", &search);

   position = linear_search(array, n, search);

   if (position == -1)
      printf("%d isn't present in the array.\n", search);
   else
      printf("%d is present at location %d.\n", search, position+1);

   return 0;
}
int linear_search(int a[], int n, int find)
{
   int c;

   for (c = 0 ;c < n ; c++ )
  {
      if (a[c] == find)
            return c;
   }
   return -1;
}
```

The time required to search an element using the algorithm depends on the size of the list. In the best case, it's present at the beginning of the list, in the worst-case, element is present at the end. Its time complexity is O(n).

8. Expalin the Binary Search Algorithm with an Example?

Ans:

Searching-

- Searching is a process of finding a particular element among several given elements.
- The search is successful if the required element is found.
- Otherwise, the search is unsuccessful.

Binary Search:

- Binary Search is one of the fastest searching algorithms.
- It is used for finding the location of an element in a linear array.
- It works on the principle of divide and conquer technique.
- Binary Search Algorithm can be applied only on **Sorted arrays**.

So, the elements must be arranged in-

- Either ascending order if the elements are numbers.
- Or dictionary order if the elements are strings.

To apply binary search on an unsorted array,

- First, sort the array using some sorting technique.
- Then, use binary search algorithm.

Binary Search Algorithm-

Consider-

- There is a linear array 'a' of size 'n'.
- Binary search algorithm is being used to search an element 'item' in this linear array.
- If search ends in success, it sets loc to the index of the element otherwise it sets loc to -1.
- Variables beg and end keeps track of the index of the first and last element of the array or sub array in which the element is being searched at that instant.
- Variable mid keeps track of the index of the middle element of that array or sub array in which the element is being searched at that instant.

```
int binary_search(int a[], int beg, int end, int item)
{
    Set beg = 0
    Set end = n-1
    while ( (beg <= end) and (a[mid] ≠ item) )
    {
            Set mid = (beg + end) / 2
            if (item < a[mid])
            {
                    Set end = mid - 1
            }
            Else if(item >a[mid])
            {
                    Set beg = mid + 1
            }
            else
```

```
            {
                    Set loc=mid;
            }
        }
        Set loc = -1
}
```

Explanation

Binary Search Algorithm searches an element by comparing it with the middle most element of the array.

Then, following three cases are possible-

Case-01

If the element being searched is found to be the middle most element, its index is returned.

Case-02

If the element being searched is found to be greater than the middle most element, then its search is further continued in the right sub array of the middle most element.

Case-03

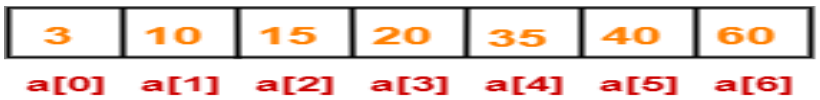
If the element being searched is found to be smaller than the middle most element, then its search is further continued in the left sub array of the middle most element.

This iteration keeps on repeating on the sub arrays until the desired element is found or size of the sub array reduces to zero.

Binary Search Example-

Consider-

- We are given the following sorted linear array.
- Element 15 has to be searched in it using Binary Search Algorithm.



Binary Search Example

Binary Search Algorithm works in the following steps-

Step-01:

- To begin with, we take beg=0 and end=6.
- We compute location of the middle element as-

mid= (beg + end) / 2

= (0 + 6) / 2

= 3

- Here, a[mid] = a[3] = 20 ≠ 15 and beg < end.
- So, we start next iteration.

Step-02:

- Since a[mid] = 20 > 15, so we take end = mid – 1 = 3 – 1 = 2 whereas beg remains unchanged.
- We compute location of the middle element as-

mid= (beg + end) / 2

= (0 + 2) / 2

= 1

- Here, a[mid] = a[1] = 10 ≠ 15 and beg < end.
- So, we start next iteration.

Step-03:

- Since a[mid] = 10 < 15, so we take beg = mid + 1 = 1 + 1 = 2 whereas end remains unchanged.
- We compute location of the middle element as-

mid= (beg + end) / 2

= (2 + 2) / 2

= 2

- Here, a[mid] = a[2] = 15 which matches to the element being searched.
- So, our search terminates in success and index 2 is returned.

Binary Search Algorithm Advantages-

The advantages of binary search algorithm are-

- It eliminates half of the list from further searching by using the result of each comparison.
- It indicates whether the element being searched is before or after the current position in the list.
- This information is used to narrow the search.
- For large lists of data, it works significantly better than linear search.

Binary Search Algorithm Disadvantages-

The disadvantages of binary search algorithm are-

- It employs recursive approach which requires more stack space.
- Programming binary search algorithm is error prone and difficult.
- The interaction of binary search with memory hierarchy i.e. caching is poor.(because of its random access nature)

Time Complexity Analysis-

Binary Search time complexity analysis is done below-

- In each iteration or in each recursive call, the search gets reduced to half of the array.
- So for n elements in the array, there are $\log_2 n$ iterations or recursive calls.

Thus, we have-

Time Complexity of Binary Search Algorithm is $O(\log_2 n)$.

Here, n is the number of elements in the sorted linear array.

This time complexity of binary search remains unchanged irrespective of the element position even if it is not present in the array.

9. Write a c program to implement Binary Search Algorithm using Recursive and Non-Recursive(Iteartive) approach?

Ans:

Bianary Searh:

The binary search algorithm is an algorithm that is based on compare and split mechanism. The binary Search algorithm is also known as *half-interval search, logarithmic search, or binary chop*. The binary search algorithm, search the position of the target value in a sorted array. It compares the target value with the middle element of the array. If the element is equal to the target element then the algorithm returns the index of the found element. And if they are not equal, the searching algorithm uses a half section of that array, Based on the comparison of the value, the algorithm uses either of the first-half (when the value is less than the middle) and the second half (when the value is greater than the middle). And does the same for the next array half.

To implement the binary search we can write the code in two ways. these two ways defer in only the way we call the function that checks for the binary search element. they are:

- **Using iterations**: this means using a loop inside the function that checks for the equality of the middle element.
- **Using recursion:** In this method, the function calls itself again and again with a different set of values.

Using Iterative or Non- Recursive Approach:

```
include<stdio.h>
int iterativeBsearch(int A[], int size, int element);
int main()
{
        int A[] = {0,12,6,12,12,18,34,45,55,99};
        int n=55;
        printf("%d is found at Index %d \n",n,iterativeBsearch(A,10,n));
        return 0;
}
int iterativeBsearch(int A[], int size, int element)
{
        int start = 0;
        int end = size-1;
        while(start<=end)
        {
                int mid = (start+end)/2;
                if( A[mid] == element)
                {
```

```
                        return mid;
                }
                else if( element < A[mid] )
                {
                        end = mid-1;
                }
                 else
                {
                        start = mid+1;
                }
        }
        return -1;
}
```

Using Recursive Approach:

```
#include<stdio.h>
int RecursiveBsearch(int A[], int start, int end, int element)
{
         if(start>end)
                return -1;
        int mid = (start+end)/2;
        if( A[mid] == element )
                return mid;
        else if( element < A[mid] )
                RecursiveBsearch(A, start, mid-1, element);
        else
                RecursiveBsearch(A, mid+1, end, element);
}
int main()
{
         int A[] = {0,2,6,11,12,18,34,45,55,99},n=55;
        printf("%d is found at Index %d \n",n,RecursiveBsearch(A,0,9,n));
        return 0;
}
```

10. What are the differences between Linear Search and Binary Search ?

Ans:

- The linear search follows sequence and Binary search doesn't follow. The linear search starts searching from the starting to ending point. Binary searching starts from the middle point.
- For binary search, we need sorted elements. Linear search does not need sorted elements. It searches the entire element in all position until it gets the desired elements.
- The number of comparison in Binary Search is less than Linear Search as Binary Search starts from the middle for that the total comparison is logN.
- The time complexity of Linear search is:
  - Best case = O(1)
  - Average case = n(n+1)/2n = O(n)
  - Worst case = O(n)
- The time complexity of Binary search is:
  - Best case = O(1)
  - Average case = logn(logn+1)/2logn = O(logn)
  - Worst case = O(logn)

11. What is sorting? Explain the Bubble Sort Algorithm with an example

Ans:

Sorting:

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios −

- Telephone Directory − The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- Dictionary − The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

Bubble Sort Algorithm

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where n is the number of items.

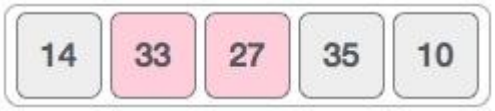
How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this −

Next we compare 33 and 35. We find that both are in already sorted positions.

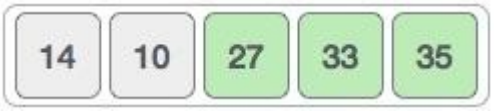
Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

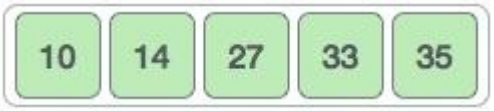
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Algorithm

```
void bubbleSort( int a[], int n )
{

  for i = 0 to n-1
  {
    for j = 0 to n-1
    {
      /* compare the adjacent elements */
      if list[j] > list[j+1])
       {
        /* swap them */
        temp=list[j]
        list[j]=list[j+1];
        list[j+1]=temp;
       }
    }
  }
}
```

12. Write a C Program to implement Bubble Sort Algorithm.

Ans:

Program:

```
#include <stdio.h>
#define MAX 10
int list[MAX] = {1,8,4,6,0,3,5,2,7,9};
void display()
{
         int i;
        printf("[");
         // navigate through all items
        for(i = 0; i < MAX; i++)
        {
                printf("%d ",list[i]);
        }
        printf("]\n");
}
```

```
void bubbleSort()
{
        int temp;
        int i,j;
       // loop through all numbers
       for(i = 0; i < MAX-1; i++)
       {
                // loop through numbers falling ahead
                for(j = 0; j < MAX-1-i; j++)
                {
                        // check if next number is lesser than current no
                        //   swap the numbers.
                        //  (Bubble up the highest number)
                        if(list[j] > list[j+1])
                        {
                                temp = list[j];
                                list[j] = list[j+1];
                                list[j+1] = temp;
                        }
                }
        }
}
void main()
{
       printf("Input Array: ");
       display();
       printf("\n");

        bubbleSort();
        printf("\nOutput Array: ");
        display();
}
```

13. Explain the Selection Sort Algorithm with an example.

Ans :

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

How Selection Sort Works?

Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

After two iterations, two least values are positioned at the beginning in a sorted manner.

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −

Algorithm

```
void selection sort( int list[], int n)//   list  : array of items   n     : size of list
{
      for i = 1 to n - 1
      {
                /* set current element as minimum*/
                min = i
                 /* check the element to be minimum */
               for j = i+1 to n
                {
                          if (list[j] < list[min])
                          {
                                    min = j;
                          }
                }
                 /* swap the minimum element with the current element*/
                temp= list[min]
                list[min]=list[i]
                list[i]=temp;
      }
 }
```

14. Write a C program to implement selection sort Algorithm?

Ans:

Program:

```
#include <stdio.h>
#define MAX 7
int intArray[MAX] = {4,6,3,2,1,9,7};
void display()
{
          int i;
          printf("[");
          // navigate through all items
          for(i = 0;i < MAX;i++)
          {
                    printf("%d ", intArray[i]);
          }
          printf("]\n");
}
void selectionSort()
{
       int indexMin,i,j,temp;
       // loop through all numbers
        for(i = 0; i < MAX-1; i++)
       {
            // set current element as minimum
             indexMin = i;
```

```
            // check the element to be minimum
            for(j = i+1;j < MAX;j++)
            {
                  if(intArray[j] < intArray[indexMin])
                  {
                        indexMin = j;
                  }
            }
            // swap the numbers
            temp = intArray[indexMin];
            intArray[indexMin] = intArray[i];
            intArray[i] = temp;
      }
}
void main()
{
      printf("Input Array: ");
      display();

      selectionSort();
      printf("Output Array: ");
      display();
}
```

15. Explain the Insertion Sort Algorithm with an Example.

Ans:

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

How Insertion Sort Works?

We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list

Algorithm:

```
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;
    }
}
```

16. Write a C Program to implement Insertion Sort Algorithm

Ans:

```
// 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]);
    printf("\nBefore Sorting\n");
    printArray(arr, n);

    insertionSort(arr, n);
    printf("\nAfter Sorting\n");
    printArray(arr, n);

    return 0;
 }
```

17. Differentiate the bubble sort, insertion sort and selection sort techniques

Ans:

BUBBLE SORT	INSERTION SORT	SELECTION SORT
A simple sorting algorithm that continuously steps through the list and compares the adjacent pairs to sort the elements	A simple sorting algorithm that builds the final sorted list by transferring one element at a time	A simple sorting algorithm that repeatedly searches remaining items to find the smallest element and moves it to the correct location
Compares the adjacent elements and swap accordingly	Transfers an element at a time to the partially sorted array	Finds the least element and moving it accordingly
Less efficient	More efficient than selection sort	Less efficient than insertion sort
Slower Uses item exchanging	Complex than selection sort	Simpler than insertion sort Uses item selection

18. What is an Algorithm and explain its characteristics

Ans:

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

Characteristics of an Algorithm

Not all procedures can be called an algorithm.

An algorithm should have the following characteristics −

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility** − Should be feasible with the available resources.
- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

19. What is meant by Time and Space Complexities?

Ans:

Time complexity

The time complexity of an algorithm is the amount of time taken by the algorithm to complete its process as a function of its input length, *n*. The time complexity of an algorithm is commonly expressed using *asymptotic notations*:

- Big O - O(n),
- Big Theta - Θ(n)
- Big Omega - Ω(n)

Space complexity

The space complexity of an algorithm is the amount of space (or memory) taken by the algorithm to run as a function of its input length, *n*. Space complexity includes both *auxiliary space* and space used by the input.

Auxiliary space is the temporary or extra space used by the algorithm while it is being executed. Space complexity of an algorithm is commonly expressed using Big O (O(n)) notation.

Many algorithms have inputs that can vary in size, e.g., an array. In such cases, the space complexity will depend on the size of the input and hence, cannot be less that O(n) for an input of size n. For fixed-size inputs, the complexity will be a constant O(1).

20. Give a brief note on asymptotic notations.

Ans:

Asymptotic Analysis

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

An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.

The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

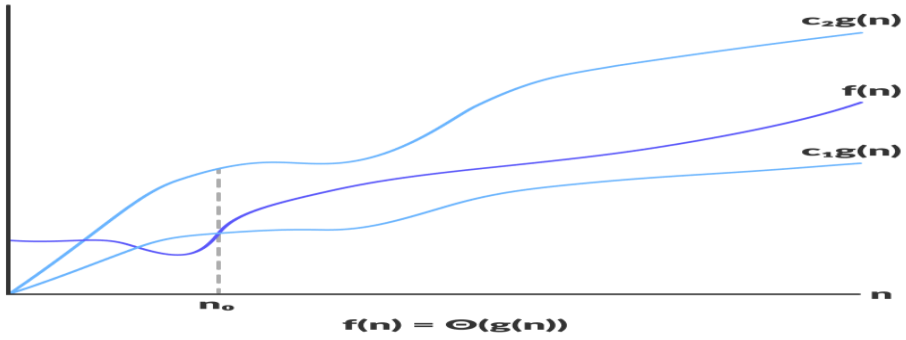
But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations: Theta notation, Omega notation and Big-O notation.

1. Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average case complexity of an algorithm.

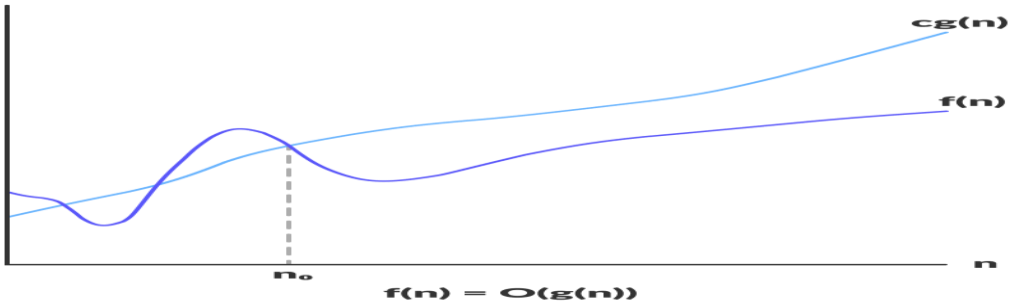


For a function g(n), Θ(g(n)) is given by the relation:

$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2 and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0 \}$

The above expression can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants c_1 and c_2 such that it can be sandwiched between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n. If a function f(n) lies anywhere in between $c_1 g(n)$ and $c_2 > g(n)$ for all $n \geq n0$, then f(n) is said to be asymptotically tight bound.

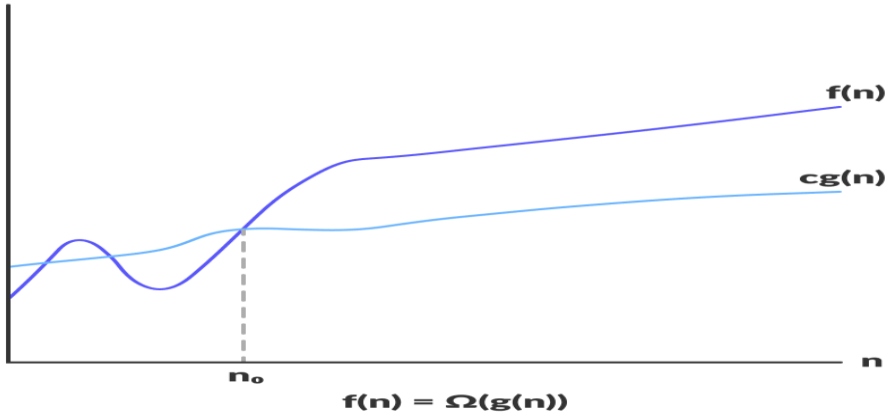
2.Big-O Notation (O-notation): Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst case complexity of an algorithm.



$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0 \}$

The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n. For any value of n, the running time of an algorithm does not cross time provided by O(g(n)). Since it gives the worst case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst case scenario.

3.Omega Notation (Ω-notation): Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides best case complexity of an algorithm.



Omega gives the lower bound of a function

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0 \}$

The above expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c such that it lies above cg(n), for sufficiently large n. For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n)).

21. How do you find the time complexity of a bubble sort?

Ans:

Algorithm:

```
for (c = 0; c < ( n - 1 ); c++)
{
        for (d = 0; d < n - c - 1; d++)
        {
                if (array[d] > array[d+1]) /* For descending order use < */
                {
                        swap       = array[d];
                        array[d]   = array[d+1];
                        array[d+1] = swap;
                }
        }
}
```

Time Complexity Computation:

When you see the code, found in first phase inner loop run n time then in second phase n - 1, and n - 2 and so on. That means in every iteration its value goes down. For example if i have a[] = {4, 2, 9, 5, 3, 6, 11} so the total number of comparison will be –

1st Phase - 7 time
2nd phase - 6 time
3rd Phase - 5 time
4th Phase - 4 time
5th Phase - 3 time
6th Phase - 2 time
7th Phase - 1 time

So you've noticed that the total number of comparisons done is n + (n - 1) + ... + 2 + 1. This sum is equal to n * (n + 1) / 2 which is equal to $0.5\ n^2 + 0.5n$ which is clearly $O(n^2)$.

Let's go through the cases for Big O for Bubble Sort

Case 1) O(n) (Best case) This time complexity can occur if the array is already sorted, and that means that no swap occurred and only 1 iteration of n elements

Case 2) $O(n^2)$ (Worst case) The worst case is if the array is already sorted but in descending order. This means that in the first iteration it would have to look at n elements, then after that it would look n - 1 elements (since the biggest integer is at the end) and so on and so forth till 1 comparison occurs. Big-O = n + n - 1 + n - 2 ... + 1 = $(n*(n + 1))/2 = O(n^2)$