<u>Unit - V: Introduction to Algorithms:</u>

Algorithms for to finding roots of a 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

<u>Definitions of an algorithm.</u>

1. Algorithm is a step-by-step procedure, to solve a given logical problem.
2. A method of representing the step – by – step logical procedure for solving a problem in natural language (like English, etc.) is called as algorithm.
3. 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.

<u>Characteristics of an Algorithm</u>

1. **<u>Finiteness −</u>** Algorithms must terminate after a finite number of steps.
2. **<u>Definiteness:</u>** - Each step in the algorithm should be clear and unambiguous.
3. **<u>Effectiveness</u>** – All the operations used in the algorithm can be performed exactly in a fixed duration of time.
4. **<u>Input −</u>** An algorithm valid inputs.
5. **<u>Output −</u>** An algorithm should have well-defined outputs, and should match the desired output.
6. **<u>Generality:</u>** Algorithm suitable to solve the similar problems

<u>pseudocode.</u>

- Algorithm may write any language but algorithm written in English like language is called pseudocode.
- pseudocode is similar to everyday English

Basic concept of order of complexity

Definitions:

Time complexity:

Time complexity of an algorithm quantifies/expressed the amount of time taken by an algorithm to run as a function of the length of the input.

Space complexity:

Space complexity of an algorithm expressed the amount of space or memory taken by an algorithm to run as a function of the length of the input.

- Time and space complexity depends on lots of things like hardware, operating system, processors, etc.
- However, we don't consider any of these factors while analyzing the algorithm.
- **We will only consider the execution time of an algorithm**.

Time complexity:

The **Time complexity** of algorithms is most commonly expressed/measured using the **Big O notation.**

Big O notation usually called as order of

i.e called as order of one(O(1)) Order of n (O(n))…

There are different types of time complexities:

1. **Constant Time Complexity: O(1)**

   When time complexity is constant (notated as "O(1)"), the size of the input (n) doesn't matter.

2. **Linear Time Complexity: O(n)**

   When time complexity grows in direct proportion to the size of the input, you are facing Linear Time Complexity, or O(n).

3. **Logarithmic Time Complexity: O(log n)**

   Algorithms with this complexity make computation amazingly fast. An algorithm is said to run in logarithmic time if its time execution is proportional to the logarithm of the input size.

   This means that instead of increasing the time it takes to perform each subsequent step, the time is decreased at a magnitude that is inversely proportional to the input "n".

4. **Linearithmic: O(n log n)**

   Linearithmic time complexity it's slightly slower than a linear algorithm. However, it's still much better than a quadratic algorithm.

5. **Quadratic Time Complexity: O(n^2)**

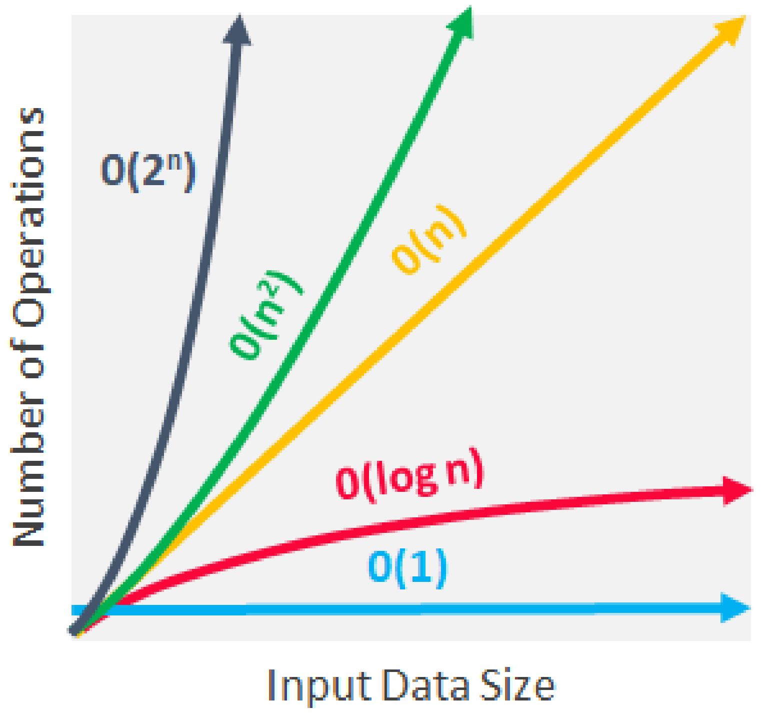
In this type of algorithms, the time it takes to run grows directly proportional to the square of the size of the input (like linear, but squared).

   Nested **for Loops** run on quadratic time, because you're running a linear operation within another linear operation, or n*n which equals n^2.

6. **Exponential Time Complexity: O(2^n)**

   In exponential time algorithms, the growth rate doubles with each addition to the input (n).

   Exponential time complexity is usually seen in Brute-Force algorithms.



Some of Time complexity with example

Big O Notation	Name	Example(s)
O(1)	Constant	# Odd or Even number,
O(log n)	Logarithmic	# Finding element on sorted array with **binary search**
O(n)	Linear	# Find max element in unsorted array, #Linear Search

Big O Notation	**Name**	**Example(s)**
O(n log n)	Linearithmic	# Sorting elements in array with **merge sort**
$O(n^2)$	Quadratic	# Duplicate elements in array **(naïve)**, # Sorting array with **bubble sort**
$O(n^3)$	Cubic	# 3 variables equation solver
$O(2^n)$	Exponential	# Find all subsets
O(n!)	Factorial	# Find all permutations of a given set/string

<u>FOR EVREY ALGORITHM/PROGRAM HAVE 3-CASE</u>

<u>Best Case</u>

The best case of Big-O notation basically means the scenario in which the shortest time is required for the function to perform the task.

<u>Average Case</u>

For the Average case scenario, the algorithm has to take the average time possible to achieve its task.

<u>Worst Case</u>

For the worst case scenario, the algorithm has to take the longest time possible to achieve its task.

<u>Example of algorithm to find sum of two numbers:</u>

```
Step1:  Begin
Step2:  read a,b
Step3:  add a and b and store in variable c
```

```
Step4:  display c

Step5:  stop
```

1. Algorithms for to finding roots of a quadratic-equations

Description:

- The quadratic-equations **ax^2+bx+c**
- To find the of roots of quadratic equation,
- Calculate Quadrant/ discriminate = b2-4ac

    If b2-4ac >0 then roots are real and distinct

    If b2-4ac =0 then roots are real and equal

    If b2-4ac <0 then roots are imaginary

Algorithm:

```
Step 1: Start
Step 2: Declare variables a, b, c, D, r1, r2, rp and ip;
Step 3: Calculate discriminate D = b*b-4*a*c
Step 4: if (D == 0)
Display "Roots are Equal" and calculate r1 = r2 = (-b / 2*a)
Step 5: else if (D > 0)
        Display " Roots are real and distinct”,
        Calculate r1 = (-b + sqrt(D)/ 2*a)
                  r2 = (-b - sqrt(D)/ 2*a)
Step 6: else if (D<0)
Calculate rp = -b/(2*a); and ip = sqrt(-D)/(2*a);
Display "Roots are Imaginary”, r1= rp+I ip and r2=rp- I ip;

Step7:  Stop
```

<u>Where</u>

rp =real part

ip=imaginary part

r1=root one

r2= root two

D= Discriminate

<u>Write c program on finding roots of a quadratic-equations</u>

<u>PROGRAM:</u>

```
#include<stdio.h>
#include<math.h>
void main()
{
float  a, b, c, d, root1, root2;
printf("\n Enter the values of a, b, c:\n");
 scanf("%f%f%f", &a, &b, &c);
 if(a == 0 || b == 0 || c == 0)
{
 printf("\n Error: Roots can't be determined”\n");
 }
 else
 {
    d = (b * b) - (4.0 * a * c);
     if(d > 0.00)
    {
printf("\n Roots are real and distinct:");
                root1 = (-b + sqrt(d)) / (2.0 * a);
                 root2 = (-b - sqrt(d) )/ (2.0 * a);
printf("\n Root1 = %f \nRoot2 = %f\n", root1, root2);
       }
     else if (d < 0.00)
                {
                 printf("\n Roots are imaginary:");
                root1 = -b / (2.0 * a) ;
                root2 = sqrt(-d) / (2.0 * a);
    printf("\n Root1 = %.2f  +i %f", root1, root2);
    printf("\n Root2 = %.2f  -i %f\n", root1, root2);
      }
     else if (d == 0.00)
                {
      printf("\n Roots are real and equal:");
      root1 = -b / (2.0 * a);
     root2 = root1;
     printf("\n Root1 = %.2f", root1);
     printf("\n Root2 = %.2f\n", root2);
    }
    }
```

```
}// main
```

OUTPUT:

```
Enter the values of a, b, c:

1  2  3

Roots are imaginary

Root1 = -1.000 +1.4 i
```

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

Description:
A prime number is a positive integer which is divisible only by 1 and itself.

For example: 2, 3, 5, 7, 11, 13.

PROGRAM

```
#include <stdio.h>
void main()
{
  int i, n, c= 0;
  printf(" Enter a number: \n");
  scanf("%d", &n);
  for(i=1; i<=n; i++)
  {
    if(n%i==0)
    {
      c++;
    }
  }
  if(c==2)
  {
    printf(" \n %d  is a prime number\n",n);
  }
  else
  {
      printf("\n %d  is not a prime number \n",n);
   }
}
```

OUTPUT:

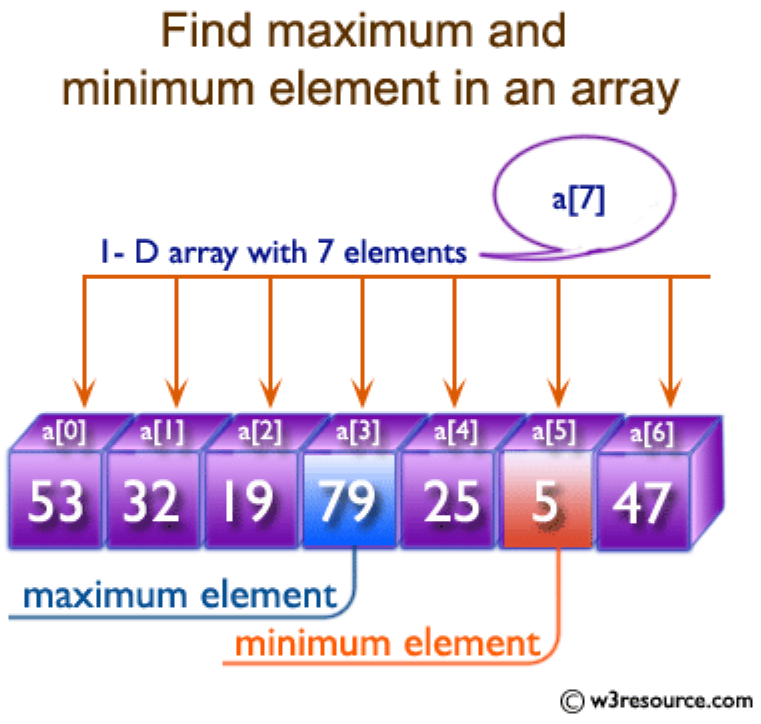
```
Enter a positive integer:
7
7 is a prime number.
```

Algorithm:

```
Step 1: Start
Step 2: Declare variables n, i, c.
Step 3: Initialize variables i=1,c=0.
Step 4: Read number n from user
Step 5: if i<=n then goto step 6 otherwise step 9
Step 6: If n % i ==0 then goto step 7 otherwise step 8
Step 7: Set c=c+1(increment by c)
Step 8: increment i and goto step 5
Step 9: If c==2 then
          print 'The given number is prime'
        else
          print 'The given number is not prime'
Step 10: Stop
```

Algorithm to finding minimum and maximum numbers of a given set,

Write a C program to input elements in an array from user, find maximum and minimum element in array. C program to find biggest and smallest elements in an array.



Example

Input

Input array elements: 53, 32, 19, 79, 25, 5, 47

Output

Maximum = 79

Minimum = 5

PROGRAM:

```
#include <stdio.h>

void main()
{
   int a[50],i,n,min,max;
   printf("\nEnter size of the array :\n ");
   scanf("%d",&n);
   printf("\n Enter elements in array :\n ");
   for(i=0; i<n; i++)
      scanf("%d",&a[i]);
   min=max=a[0]; // 1st element of the array.
// Compare min, max values with a[i],
for(i=0; i<n; i++)
   {
        if(max>a[i])
                         max=a[i];   // a[i] contain the  maximum value
         if(min<a[i])
                           min=a[i];  // a[i] contain the  minimum value
   }
    printf("\n minimum of array is : %d",min);
   printf("\n maximum of array is : %d",max);
}
```

Output:

```
Enter size of the array:
 5
Enter elements in array:
12
21
13
54
15
minimum of an array is: 12
maximum of an array is: 54
```

Algorithm

Step 1: Start

Step 2: Declare variables a[50], i, n, max, min

Step 3: Initialize variables i=0,

Step 4: Read array size i.e n from user

Step 5: Read the array elements from user (keyboard)

```
Step 6: initialize min=max=a[0].
Step 7: If i < n then goto step 8 otherwise step 9
Step 8: i. if max > a[i] assign max value to a[i]
        ii. if min < a[i] assign min value to a[i]
        Increment i by 1 and goto step 7
Step 9: print "The minimum and maximum value in an array"
Step 10: Stop
```

SEARCHING

Basic searching in an array of elements (linear and binary search techniques),

- Searching means finding the element in an array element
- Searching has two possibilities

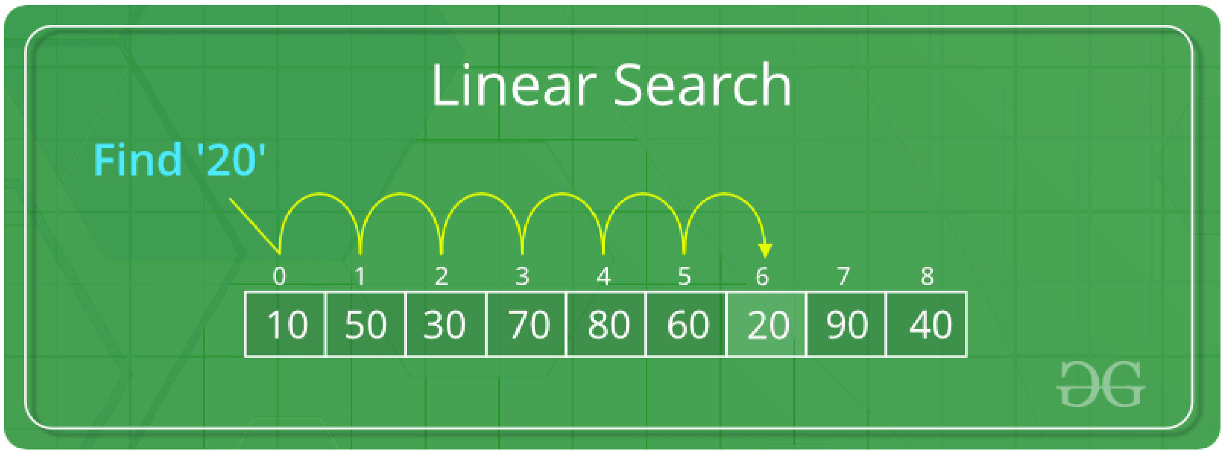
1. Search successful (i.e search element found in an array of elements)
2. Search un-successful (i.e search element not found in an array of elements)

- Searching has two techniques (methods)

1. **Linear search**
2. **binary search**

1. **Linear search**

- Linear search is also called **sequential search**
- Linear search is a method for searching a search element within an array of elements.
- It sequentially checks one by one of the array of elements for the target element until a match is found (or until all the elements have been searched of that array.)
- If the search element is no found in list of array elements then search has failed



From above

example

Key element is 20 found in array of elements in 7 th place

A simple approach is to do linear search, i.e

- Start from the leftmost element of arr[] and one by one compare **key element** with each element of arr[]
- If **key element** matches with an array of element, print the element
- If **key element** doesn't match with any of elements in the array, print not found

<u>Linear search is implemented using following steps...</u>

<u>Algorithm:</u>

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 is 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 search element is compared with last element in the list.

Step 6 - If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

<u>Example</u>

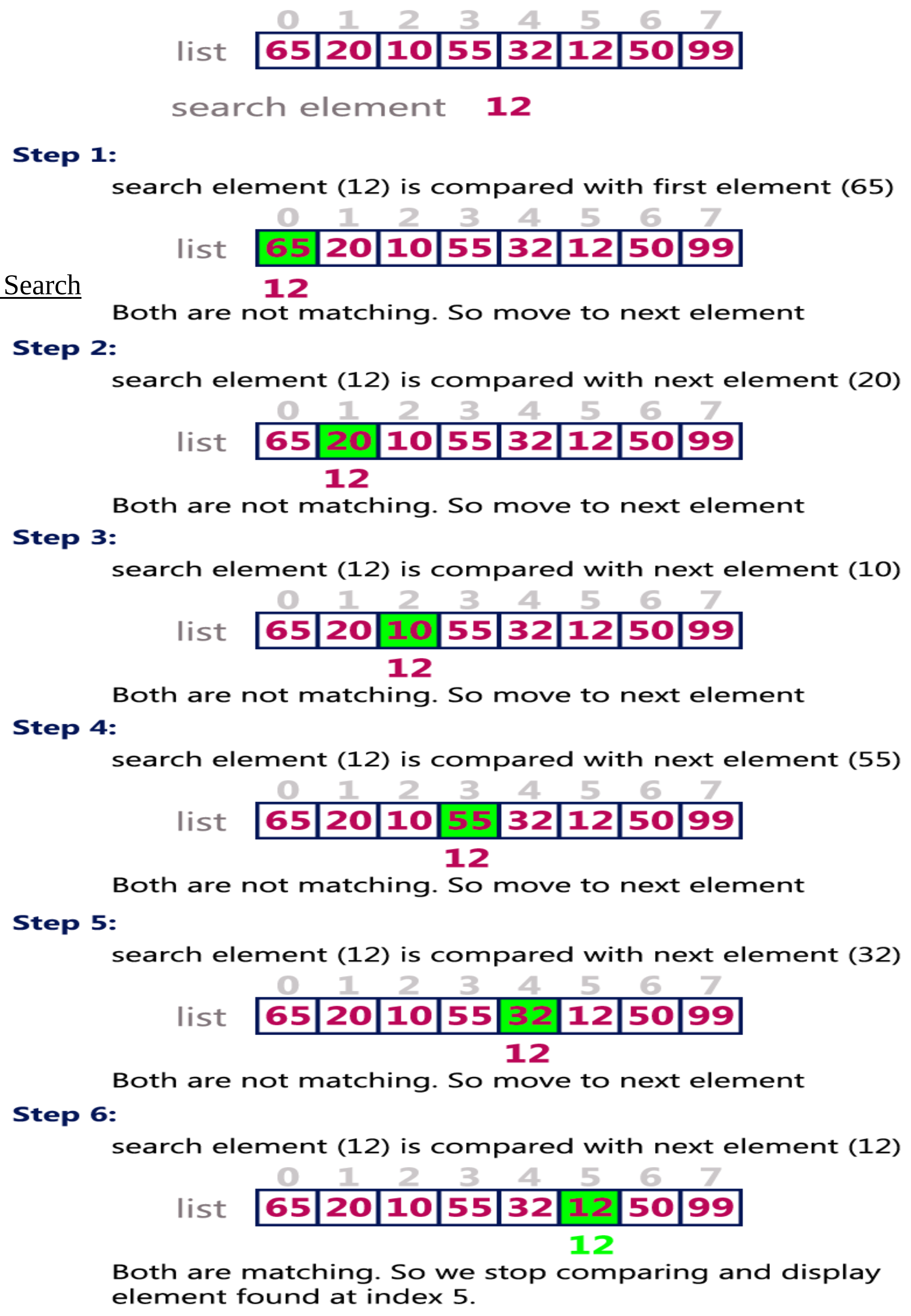
Let us consider the following list of elements in array and

Find 12th element

Array elements are {65, 20, 10, 55, 32, 12, 50, 99}

Example

Linear Search



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 (LINEAR SEARCH)

```
#include<stdio.h>

Void main()
{
        int a[20],i,key,n;
        printf("\n How many elements:\n");
        scanf("%d",&n); //5

        printf("Enter array elements:\n");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]); // 1-D array element

        printf("\nEnter element to search:");
        scanf("%d",&key);   6

        for(i=0;i<n;i++)
                if(a[i]==key)
                        {
printf("\n The %d Element found at position %d",key,i+1);
                        break;
                                }
             if(i==n)
                printf("\n %d Element not found",key);

      }
```

Output

```
How many elements:

4

Enter array elements:

6

8

 9
```

```
1

Enter element to search:

9

The 9 Element found at position 3
```

<u>Linear program using function:</u>

<u>Program:</u>

```
#include<stdio.h>
  void   linear_search(int a[20], int k, int n);
void main()
{
            int a[20],i,key,n;
            printf("\n How many elements:\n");
            scanf("%d",&n); // n- means size of array

            printf("Enter array elements:\n");
            for(i=0;i<n;i++)
                        scanf("%d",&a[i]); // a[i] array of elements

            printf("\nEnter element to search:");
            scanf("%d",&key);

            linear_search( a, key,n); // function call

            }

void linear_search(int a[20], int k, int n);
{
int i;
            for(i=0;i<n;i++)
                        if(a[i]==k)
                                    {
            printf("\n The %d Element found at position %d",k,i+1);
                                    break;
                                                }
                        if(i==n)
```

```
                printf("\n %d Element not found",k);
}
```

<u>Output</u>

```
How many elements:
4
Enter array elements:
6
8
 9
1
Enter element to search:
9
The 9 Element found at position 2
```

<u>ALGORITHM</u>

```
Step 1: Start
Step 2: Declare variables a[20],i, key,n
Step 3: Initialize variables i=0,
Step 4: Read array size i.e n from user
Step 5: Enter the array elements from user (keyboard)
Step 6: Enter the search element from keyboard
```

Step 7: If i < n then goto **step 8** otherwise **step 11**

Step 8: if a[i]== key then goto step 10 otherwise step 9

Step 9: set i=i+1 (i is incremented by 1) and goto step 7

Step 10: print “search element is found” and break goto step 12

Step 11: print “search element is not found”

Step 12: Stop

Time- Complexity of algorithm

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

Time complexity

1. **Average Case:** If the searched element is other than the first and the last element of the array.

For example:

If the input array is {4, 6, 1, 2, 5, 3}

and if the element searched is 6,

then the expected output will be Position 2.

Average case time complexity: O(n)

2. **Best Case:** If the searched element is the first element of the array.

For example:

If the input array is {4, 6, 1, 2, 5, 3}

and the key searched for is 4,

Then the expected output will be Position 1.

Best case time complexity: O(1)

3. <u>Worst Case:</u> If the element searched for is the last element of the array.

For example:

If the input array is {4, 6, 1, 2, 5, 3}

and the key searched for is 3,

then the expected output will be Position 6.

Worst case time complexity: O(n)

<u>Advantages of a linear search</u>

1. The list of array elements not in sorted order
   i.e (ascending order)
2. Linear search perform fast searches of small to medium lists.
3. Not affected by insertions and deletions.

<u>Disadvantages of a linear search</u>

1. Slow searching of large lists
2. This process is slow and inefficient.

Binary search:

- Binary search is a massive improvement over the sequential search.
- For binary search to work, the item in the list must be in sorted order.
- Binary search follows **divide and conquer approach**

Note: for Binary search the list elements must be in sorted order i.e ascending order

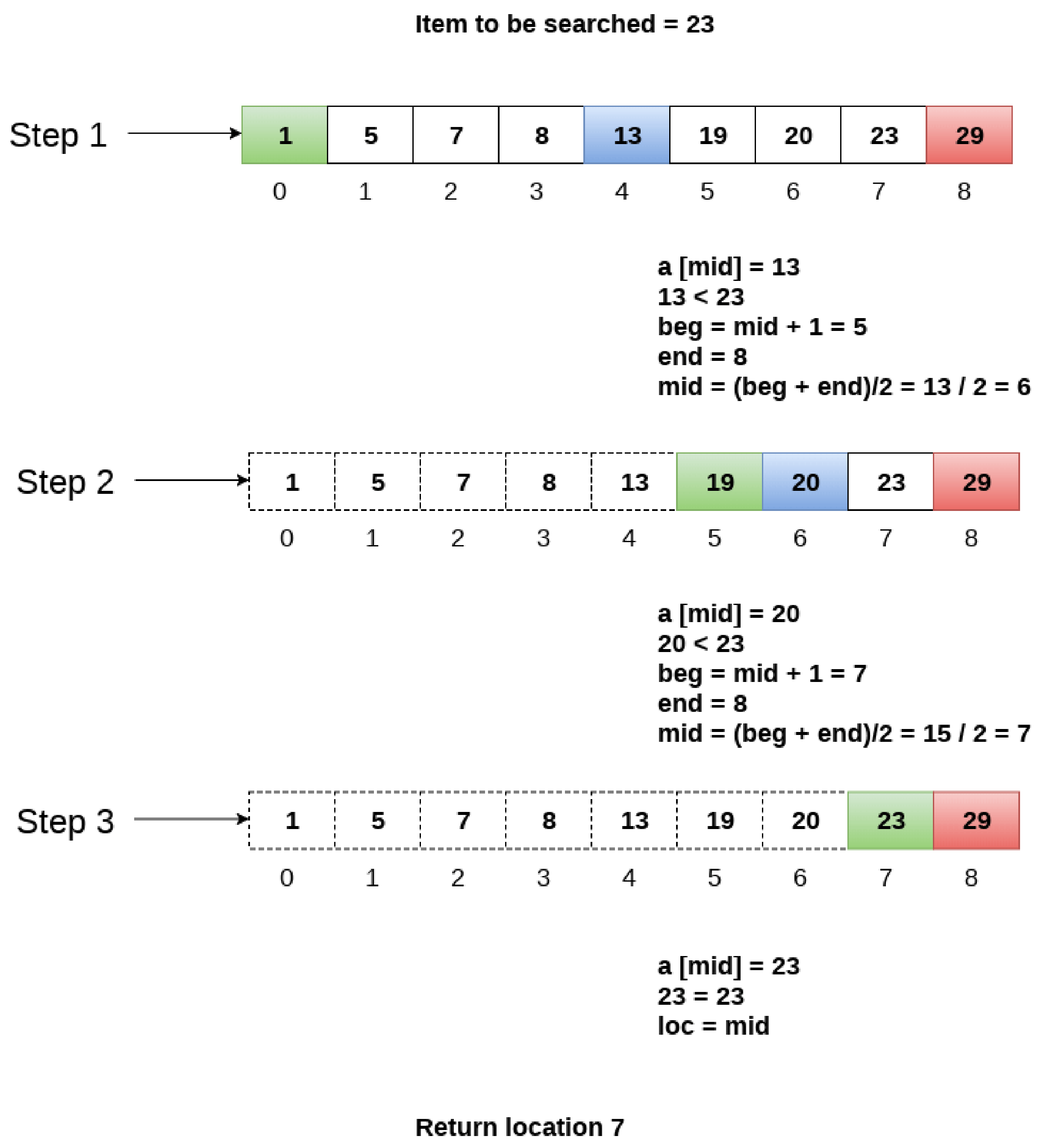
Simple approach of Binary Search:

- 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.
- It is a searching technique that is better than the liner search technique as the number of iterations decreases in the binary search.

Example

Let us consider an array a= {1, 5, 7, 8, 13, 19, 20, 23, 29}.

Find the location of the item 23 in the array.



<u>Binary search is implemented using following steps...</u>

<u>Algorithm</u>

Step 1 - Read the search element from the user.

Step 2 - Find the middle element in the sorted list.

mid=((lb+ub)/2)

Step 3 - Compare the search element with the middle element in the sorted list.

Step 4 - If both are matched, then display "Given element is found!!!" and terminate the function.

Step 5 - If both are not matched, then check whether the search element is smaller or larger than the middle element.
(i.e key< mid or key > mid)

Step 6 - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sub-list of the middle element.

Step 7 - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sub-list of the middle element.

Step 8 - Repeat the same process until we find the search element in the list or until sub-list contains only one element.

Step 9 - If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

Another way (Another algorithm)

1. Compare k with the middle element.
2. If k matches with middle element, we return the mid index.
3. Else If k is greater than the mid element, then k can only lie in right half sub-array after the mid element. So we continue for right half.
4. Else (K is smaller) continue for the left half.
5. Otherwise Search Element Is Not Found

Program(binary search)

```
#include <stdio.h>
void main()
{
   int a[10];
   int i, j, n, temp, key;
   int low, mid, high;
   printf("Enter the size of array:\n");
   scanf("%d", &n); //5
   printf("Enter the elements one by one: \n");
   for (i = 0; i < n; i++)
      scanf("%d", &a[i]); //

   /*  Bubble sorting begins */

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

 printf("\n Array elements in  Sorted order:\n");
   for (i = 0; i < n; i++)
      printf("%d\n", a[i]);
   printf("\n Enter the element to be searched: \n");
   scanf("%d", &key); //5
   /*  Binary searching begins */
   low = 0;
   high = n-1;
   do
   {
      mid = (low + high) / 2;
      if (key < a[mid])
         high = mid - 1;
      else if (key> a[mid])
```

```
            low = mid + 1;
    } while (key != a[mid] && low <= high);
    if (key == a[mid])
    {
        printf("SEARCH SUCCESSFUL \n");
    printf("\n The %d elements is found  at %d ”, key,mid+1);
    }
    else
        printf("SEARCH FAILED”);
}
```

Output:

```
Enter the size of array:
5
Enter the elements one by one:
7
3
8
2
9
Array elements in Sorted order:
2
3
7
8
9
Enter the element to be searched:
8
SEARCH SUCCESSFUL
The 8 elements is found  at 4
```

Binary program using function:

Program:

```
#include <stdio.h>
void binary_search(int a[50],int key, int n);
void main()
```

```
{
int a[50], i,n,j, k,temp;
    printf("\nEnter size of an array: ");
    scanf("%d", &n);
    printf("\n Enter the %d array elements”, n);
    for(i=0; i<n; i++)
        scanf("%d", &a[i]);
            for(i=0;i<n;i++)
                        {
                        for(j=i+1;j<n;j++)
                        {
                        if(a[i]>a[j])
                        {
                        temp=a[i];
                        a[i]=a[j];
                        a[j]=temp;
                        }
                        }
                        }
                        printf("\nthe array in sorted order:-\n");
                        for(i=0;i<n;i++)
                        printf("%d\n",a[i]);
    printf("\n Enter element to be searched:\n ");
    scanf("%d", &k);
 binary_search(a, k,n);
}
void binary_search(int a[50],int key,int n)
```

```
{

int low,high,mid;
    low = 0;
    high = n-1;
    mid = (low + high) / 2;
    while (low <= high && key != a[mid])
    {
        if (key < a[mid])
            high = mid - 1;
        else
            low = mid + 1;
        mid = (low + high) / 2;
    }
    if (a[mid] == key)
       printf("\n%d Element is found at location %d",key, mid+1);
    else
        printf("\n %d Element doesn't exist", key);
}
```

Output:

```
Enter size of an array: 5
Enter the  5 array elements:
25 10 35 89 64
The array in sorted order:
10 25 35 64 89
Enter Element to be searched: 35
35 Element is found at location 3
```

Binary Search Algorithm Advantages-

1. Very efficient and fast searching.
2. Binary search algorithm finds a given element in a list of elements with O(log n)
3. Binary search follows **divide and conquer approach**
4. For large lists of data, it works significantly better than linear search.

Binary Search Algorithm Disadvantages-

1. The element in array/list must be in sorted order. (Ascending order)
2. It employs recursive approach which requires more stack space.
3. Programming binary search algorithm is error prone and difficult.

Time 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)

Time complexity

Expected Input and Output

1. Average Case: When the element to be searched is other than the middle element. For example:

If the input array is {1, 2, 3, 4, 5, 6} and the key to be searched for is 6 then the expected output will be "Search Successful".

Average case time complexity: O(log n).

2. Best Case: If the element to be searched is the middle element of the array. For example:

If the input array is {1, 5, 9}
and the key to be searched is 5,
then the expected output will be "Search Successful".

Best case time complexity: O(1).

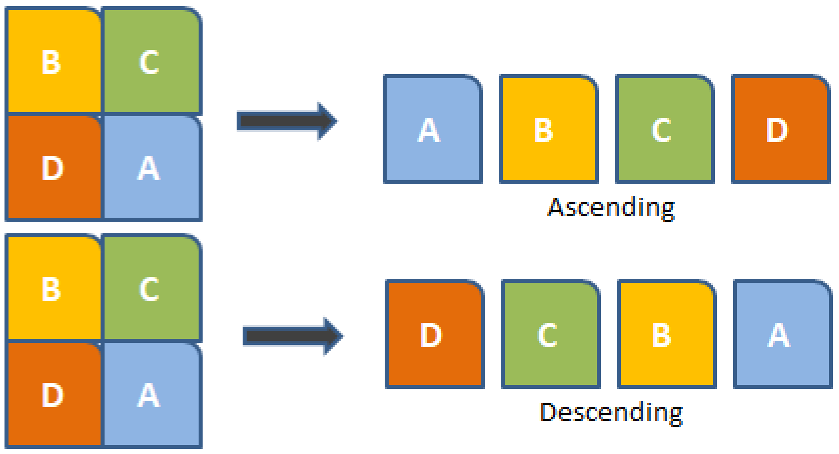
3. Worst Case: If the element to be searched for is not present in the array.

If the input array is {1, 3, 6, 8, 9}
and the key to be searched for is 4,
then the expected output will be "Search Unsuccessful".

Worst case time complexity: O (log n).

SORTING

- Arranging the data in ascending or descending order is known as sorting.
- Sorting is very important from the point of view of our practical life.
- The best example of sorting can be phone numbers in our phones. If, they are not maintained in an alphabetical order we would not be able to search any number effectively.



The sorting mechanism has been divided into are two main categories:-

1. Internal sorting
2. External sorting

1) Internal sorting

Generally a sort is classified as internal only if the data which is being sorted is in main memory.

2) External sorting

It can be external, if the data is being sorted in the auxiliary storage (i.e HDD)

<u>Sorting Methods (internal sorting method)</u>

Many methods are used for sorting, such as:

1. **Bubble sort**

2. Selection sort

3. Insertion sort

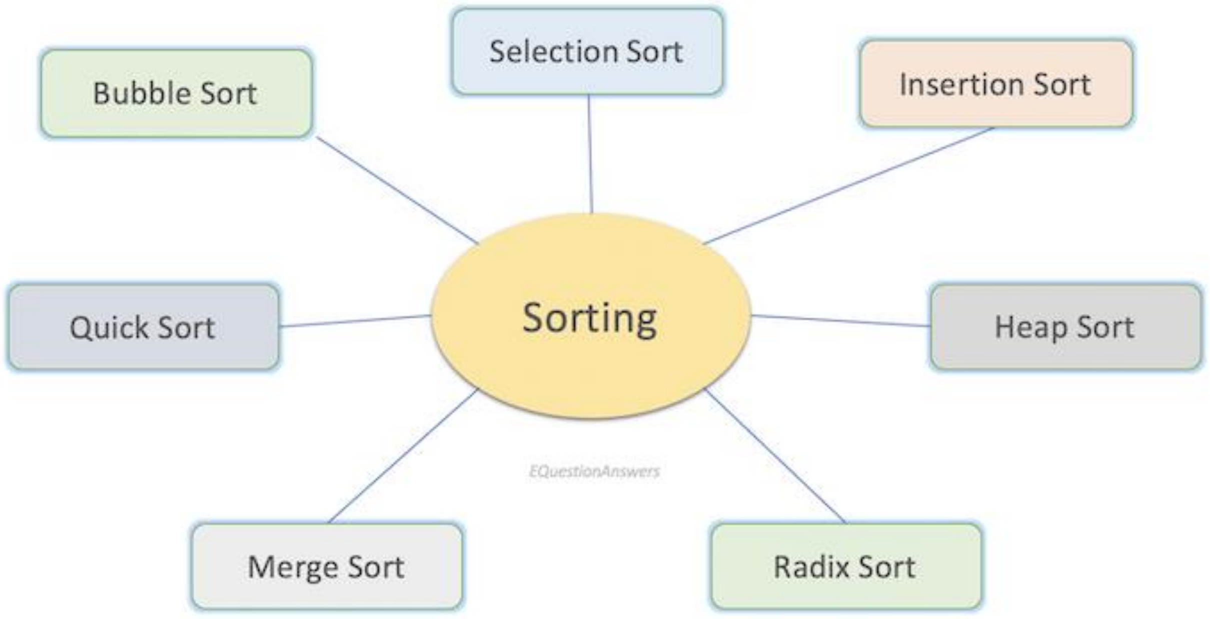
4. Quick sort

5. Merge sort

6. Heap sort

7. Radix sort

8. Shell sort



Sorting Methods

http://www.equestionanswers.com/c/c-sorting.php

Sorting algorithms:

There are the several internal sorting used in practical fields.

1. **Bubble Sort-** A sorting algorithm which compares one element to its next element and if requires it swaps like a bubble.
2. **Selection Sort -** A sorting algorithm which selects a position in the elements and compares it to the rest of the positions one by one.
3. **Insertion Sort -** A sorting algorithm which selects one element from the array and is compared to the one side of the array. Element is inserted to the proper position while shifting others.

4. **Quick Sort -** A sorting algorithm which divides the elements into two subsets and again sorts recursively.
5. **Heap Sort -** A sorting algorithm which is a comparison based sorting technique based on Binary Heap data structure. ,
6. **Merge sort -** A sorting algorithm which divides the elements to subgroups and then merges back to make a sorted.
7. **Radix Sort -** A sorting algorithm used for numbers. It sorts the elements by rank of the individual digits.

1. BUBBLE SORT

Outline

1. Definition of Bubble sort
2. Explain of Bubble sort method with Example
3. Bubble sort Algorithm(pseudocode)
4. Bubble sort program(with and without functions)
5. Time complexity and space complexity
6. Advantages and Dis-advantages

Bubble sort

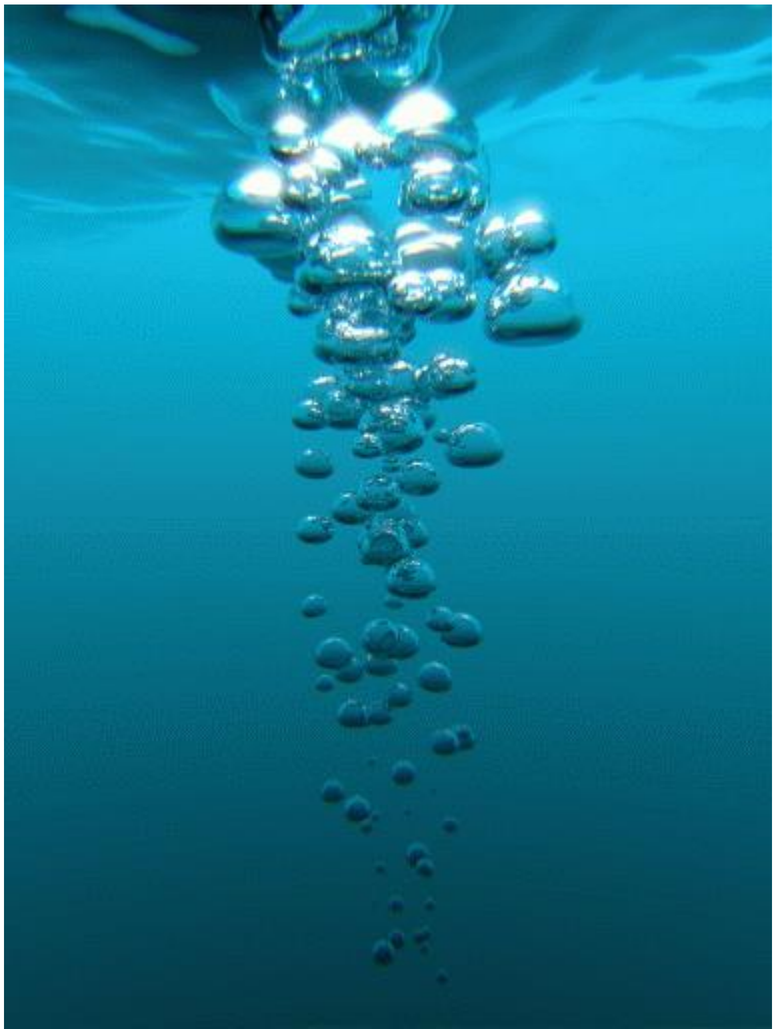
- Bubble sort is also known as **sinking sort** or **exchange sort or adjacent sort**

Definition:

Bubble sort is a simple sorting algorithm in which each element is compared with **adjacent element** and swapped if their position is incorrect/in order (ascending order)

Some main point be noted

- Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.
- It is known as bubble sort, because with every complete iteration the largest element **bubbles up** towards the last place or the highest index just like water bubble rises up to the water surface.
- This algorithm is suitable for small data sets
- Its average and worst case complexity are of O(n^2) where n is the number of items.



Bubble sort

- ➢ The largest bubble will reach the surface first
- ➢ The second largest bubble will reach the surface next

  And so on.

Steps to implement bubble sort:

1. In first cycle,
2. Start by comparing 1st and 2nd element and swap if 1st element is greater.
   - ❖ After that do the same for 2nd and 3rd element.
   - ❖ At the end of cycle you will get max element at the end of list.
3. Now do the same in all subsequent cycles.
4. Perform this for (number of elements – 1) times.
5. You will get sorted list.

Algorithm of Bubble sort

Step 1- start

Step 2- Declare a [50], N, i, j.

Step 3- Enter the array elements

Step 4 – Starting with the current array element (index = 0 to N), compare with the next element of the array.

Step 5 – If the current element is greater than the next element of the array, swap them.

Step 6 – If the current element is less than the next element, move to the next element.

Step 7 – Repeat Step 4 to 6 till the list is sorted.

Step 8- stop

Sorting elements using Bubble Sort algorithm

Let's say we have the following unsorted numbers 5, 4, 3, 1, 2 and we want to sort them in ascending order.

So, there are total 5 elements. Hence we will need (5 - 1)

i.e. 4 pass.

Pass #1

In Pass #1 we will perform **(n - i)** i.e. (5 - 1) i.e. 4 comparisons.

Start: 5, 4, 3, 1, 2

Compare 5 and 4: swap as they are not in order

[5, 4], 3, 1, 2 ---> 4, 5, 3, 1, 2

Compare 5 and 3: swap as they are not in order

4, [5, 3], 1, 2 ---> 4, 3, 5, 1, 2

Compare 5 and 1: swap as they are not in order

4, 3, [5, 1], 2 ---> 4, 3, 1, 5, 2

Compare 5 and 2: swap as they are not in order

4, 3, 1, [5, 2] ---> 4, 3, 1, 2, 5

End: 4, 3, 1, 2, 5

Pass #2

In Pass #2 we will perform (5 - 2) i.e. 3 comparisons.

Start: 4, 3, 1, 2, 5

Compare 4 and 3: swap as they are not in order

[4, 3], 1, 2, 5 ---> 3, 4, 1, 2, 5

Compare 4 and 1: swap as they are not in order

3, [4, 1], 2, 5 ---> 3, 1, 4, 2, 5

Compare 4 and 2: swap as they are not in order

3, 1, [4, 2], 5 ---> 3, 1, 2, 4, 5

End: 3, 1, 2, 4, 5

Pass #3

In Pass #3 we will perform (5 - 3) i.e. 2 comparisons.

Start: 3, 1, 2, 4, 5

Compare 3 and 1: swap as they are not in order

[3, 1], 2, 4, 5 ---> 1, 3, 2, 4, 5

Compare 3 and 2: swap as they are not in order

1, [3, 2], 4, 5 ---> 1, 2, 3, 4, 5

End: 1, 2, 3, 4, 5

Pass #4

In Pass #4 we will perform (5 - 4) i.e. 1 comparison.

Start: 1, 2, 3, 4, 5

Compare 1 and 2: they are in order so, no swapping required

1, 2, 3, 4, 5 ---> 1, 2, 3, 4, 5

End: 1, 2, 3, 4, 5

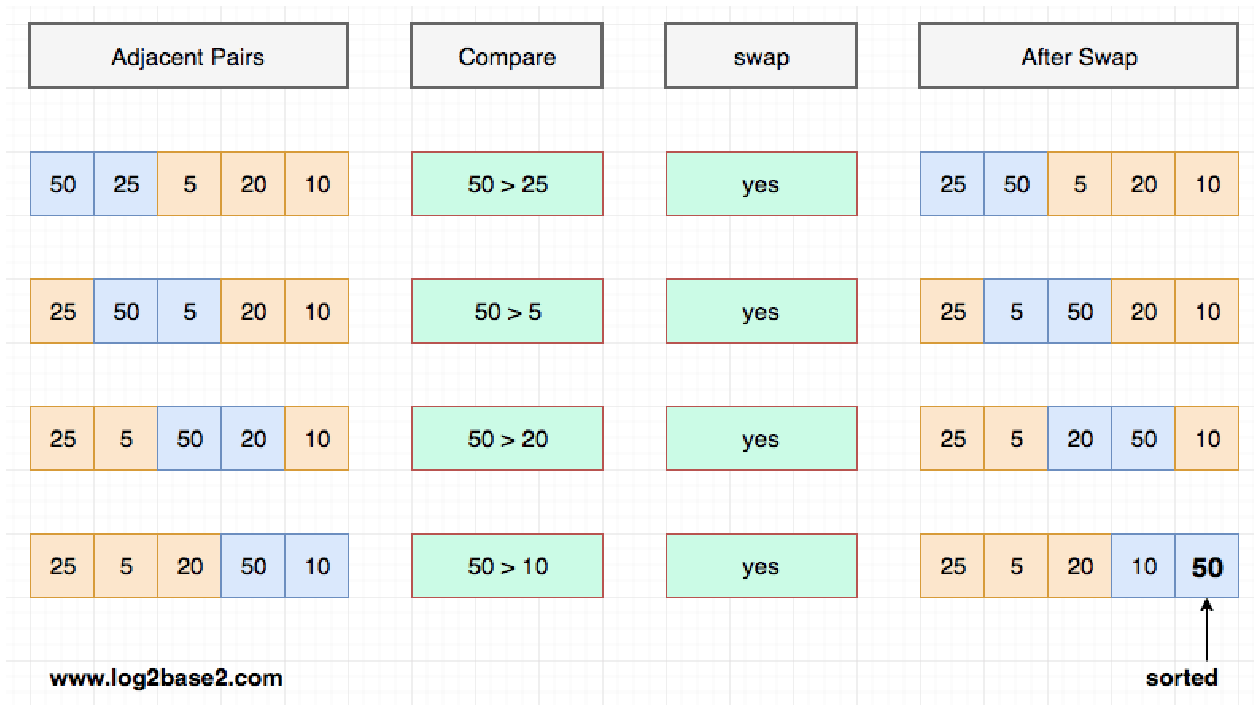
Example: 2

Let's take an array of 5 elements.

int arr[5] = {50, 25, 5, 20, 10}

Step 1

In Step #1 we will perform **(n - i)** i.e. (5 - 1) i.e. 4 comparisons

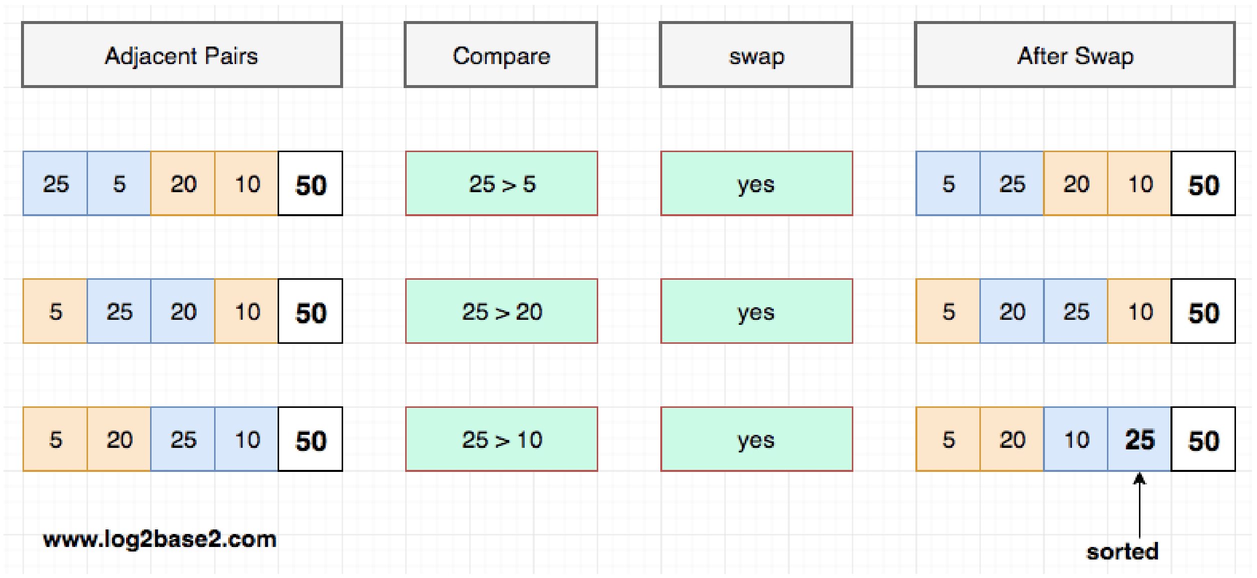


- ➢ We can notice that one element has been sorted after the above process.
- ➢ In general, to sort N element using bubble sort, we need to do the same process N-1 times.

- ➢ From next iteration onwards, we can skip the sorted elements. i.e. 50(Bubble element)

Step 2

In step #2 we will perform (5 - 2) i.e. 3 comparisons.

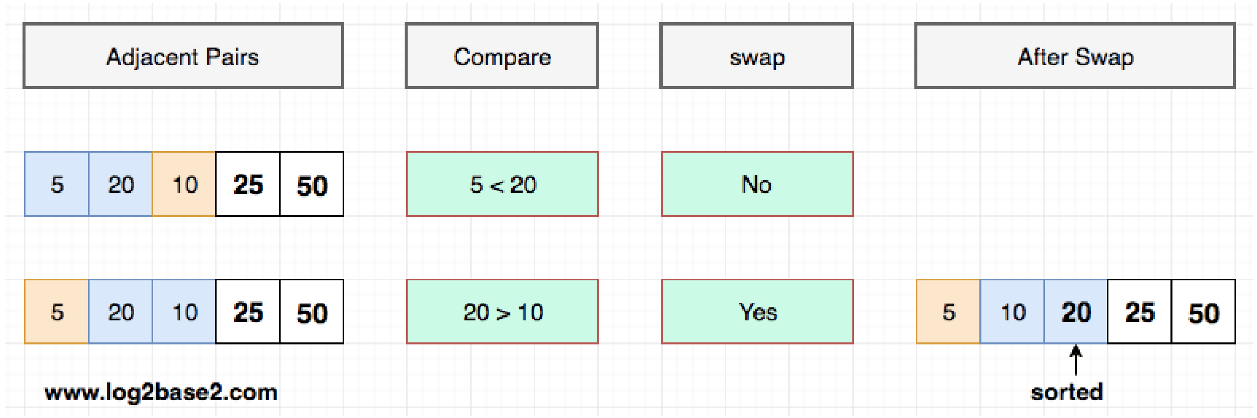


Adjacent Pairs	Compare	swap	After Swap
25 5 20 10 **50**	25 > 5	yes	5 25 20 10 **50**
5 25 20 10 **50**	25 > 20	yes	5 20 25 10 **50**
5 20 25 10 **50**	25 > 10	yes	5 20 10 **25** **50**

www.log2base2.com
sorted

<u>Step 3</u>

In step #3 we will perform (5 - 3) i.e. 2 comparisons.

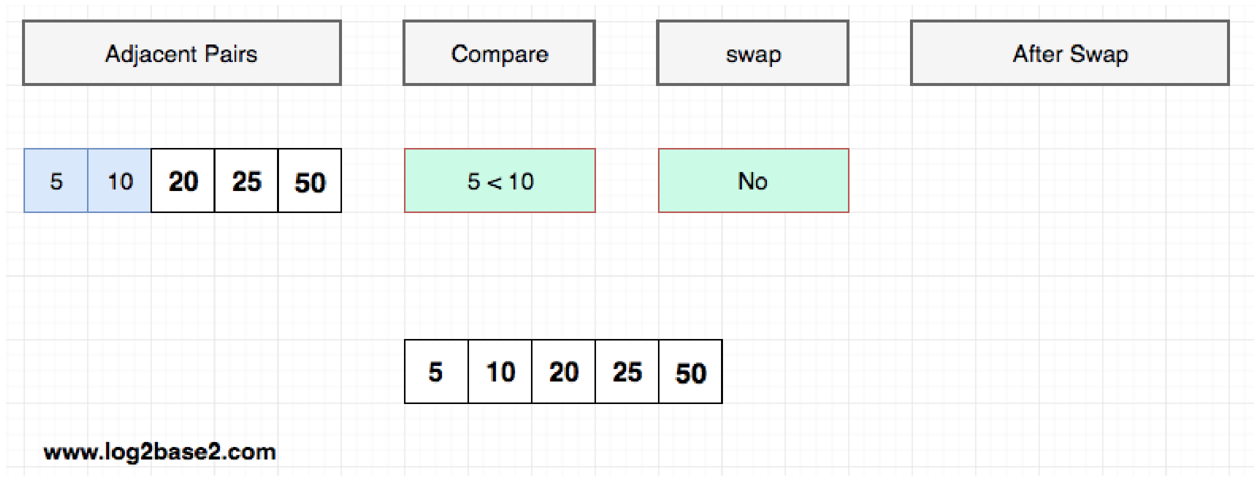


Adjacent Pairs	Compare	swap	After Swap
5 20 10 **25** **50**	5 < 20	No	
5 20 10 **25** **50**	20 > 10	Yes	5 10 **20** **25** **50**

www.log2base2.com
sorted

Step 4

In step #4 we will perform (5 - 4) i.e. 1 comparison.



Algorithm Bubble Sort

```
Step 1: start
Step 2: Declare a [50], N, i, j.
Step3: Read the array elements from key board
Step 4: For i = 0 to N-1 repeat Step 5
Step 5: For j = i + 1 to N – i-1 repeat 6
Step 6: if A[j] > A[j+1]
                Swap A[J] and A[i]
[End of Inner for loop]
[End if Outer for loop]
Step 7: stop
```

}

Bubble sort Program (without function)

```
void main()
{
int a[50], n, i, j, temp;
printf("Enter number of elements:\n");
scanf("%d", &n);  //5
printf("Enter %d Numbers:\n", n);
for(i = 0; i < n; i++)
scanf("%d", &a[i]); // 50 25 20 5 10
for(i = 0 ; i < n - 1; i++)
{
        for(j = 0 ; j < n-i-1; j++)
                {
                if(a[j] > a[j+1])
                        {
                        temp=a[j];
                        a[j]=a[j+1];
                        a[j+1]=temp;
                }
  }
}
printf("Sorted Array:\n");

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

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

}
```

<u>Output:</u>

```
Enter number of elements:

5

Enter 5 Numbers:

3

4

2

5

1


Sorted Array:

1       2       3       4       5
```

<u>Program Explanation</u>

- Then there are two ‘for loops’.
- The first ‘for loop’ runs from I value equal to zero all the way till it is less than n-1. The outer array takes care of the element of the value to be compared to all the other elements.

- Inside the for loop, there is another for loop that starts from j=0 all the way to j<n-i-1.
- This loop takes care of all elements. Inside, there is an if statement that compares if a[j]>a[j+1].
- If this condition is satisfied then swapping takes place. A variable called swap is used for this. First, a[j] is assigned to swap, followed by a[j+1] being assigned at a[j] and at last swap is assigned to a[j+1].
- This continues till all the elements are sorted. After this, the sorted array is printed.

Bubble sort Program using function:

```
#include <stdio.h>

void bubble_sort(int a[50], int n) // Function definition

 {

   int i , j, temp;

   for (i = 0; i < n-1; i++)

         {

   for (j = 0; j < n - i - 1; j++)

                  {

         if (a[j] > a[j + 1])

                           {  // swap if order is broken

            temp = a[j];

            a[j] = a[j + 1];

            a[j + 1] = temp;

         }

      }
printf("Printing the sorted array:\n");

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

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

}

void main()

{

  int a[50], n, i;

  printf("Enter number of elements in the array:\n");

  scanf("%d", &n);

  printf("Enter %d integers\n", n);

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

    scanf("%d", &a[i]);

  bubble_sort(a, n); //Function call


}
```

Output:

```
Enter number of elements in the array:

5

Enter 5 integers:
```

```
3

4

2

5

1

Printing the sorted array:

1	2	3	4	5
```

```
// C program for bubble sort with function (Extra Program)

#include <stdio.h>

void bubbleSort(int arr[50], int n); // Function declaration

Void 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:\n ");

   scanf("%d", &n);
```

```
    // input elements if the array
    for(i = 0; i < n; i++)
    {
        printf("\n Enter element no:%d: ", i+1);
        scanf("%d", &arr[i]);
    }
    // call the function bubbleSort
    bubbleSort(arr, n);  // function call
}

void bubbleSort(int arr[50], int n) // function  definition
{
    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:\n");

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

    {

        printf("%d\t", arr[i]);

    }

}
```

Output:

```
Enter the number of elements to be sorted:

5

Enter element no:1: 6
Enter element no:2: 4

Enter element no:3: 5

Enter element no:4: 3

Enter element no:5: 2

Sorted Array:
```

2 3 4 5 6

<u>Time Complexity Analysis of Bubble Sort</u>

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,

<u>Output</u>

i.e n(n-1) (n-2) (n-3)

n+(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).

i.e order of n square

<u>Time Complexity</u>

<u>Best case:</u>

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

<u>Average case and worst case:</u>

Time complexity Average case and worst case of Bubble Sort is O(n2).

<u>Space complexity:</u>

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

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

Complexity	**Best Case**	**Average Case**	**Worst Case**
Time	O(n)	$O(n^2)$	$O(n^2)$
Space			O(1)

Advantage and Disadvantages of Bubble Sort:

Advantages

1. The primary advantage of the bubble sort is that it is popular and easy to implement.
2. No external memory is needed.
3. The space requirement is at a minimum.

Disadvantages

1. Very expensive, $O(n^2)$ in worst case and average case.
2. Bubble sort does not deal well with a list containing a huge number of items.
3. The bubble sort is mostly suitable for academic teaching but not for real-life applications.

INSERTION SORT

Outline

1. Definition of insertion sort
2. Explain of insertion sort method with Example
3. Insertion sort Algorithm(pseudocode)
4. insertion sort program(with and without functions)
5. Time complexity and space complexity
6. Advantages and Dis-advantages

1. Definition of insertion sort

Insertion Sort is a sorting algorithm where the array is sorted by taking one element at a time. The principle behind insertion sort is to take one element, iterate through the sorted array & find its correct position then insert in the sorted array**.**

Simple Definition:

In this Insertion sort method the insert / current element is checked with preceding (i.e previous) elements, if they are not in correct order (ascending order) insert the current element in proper order

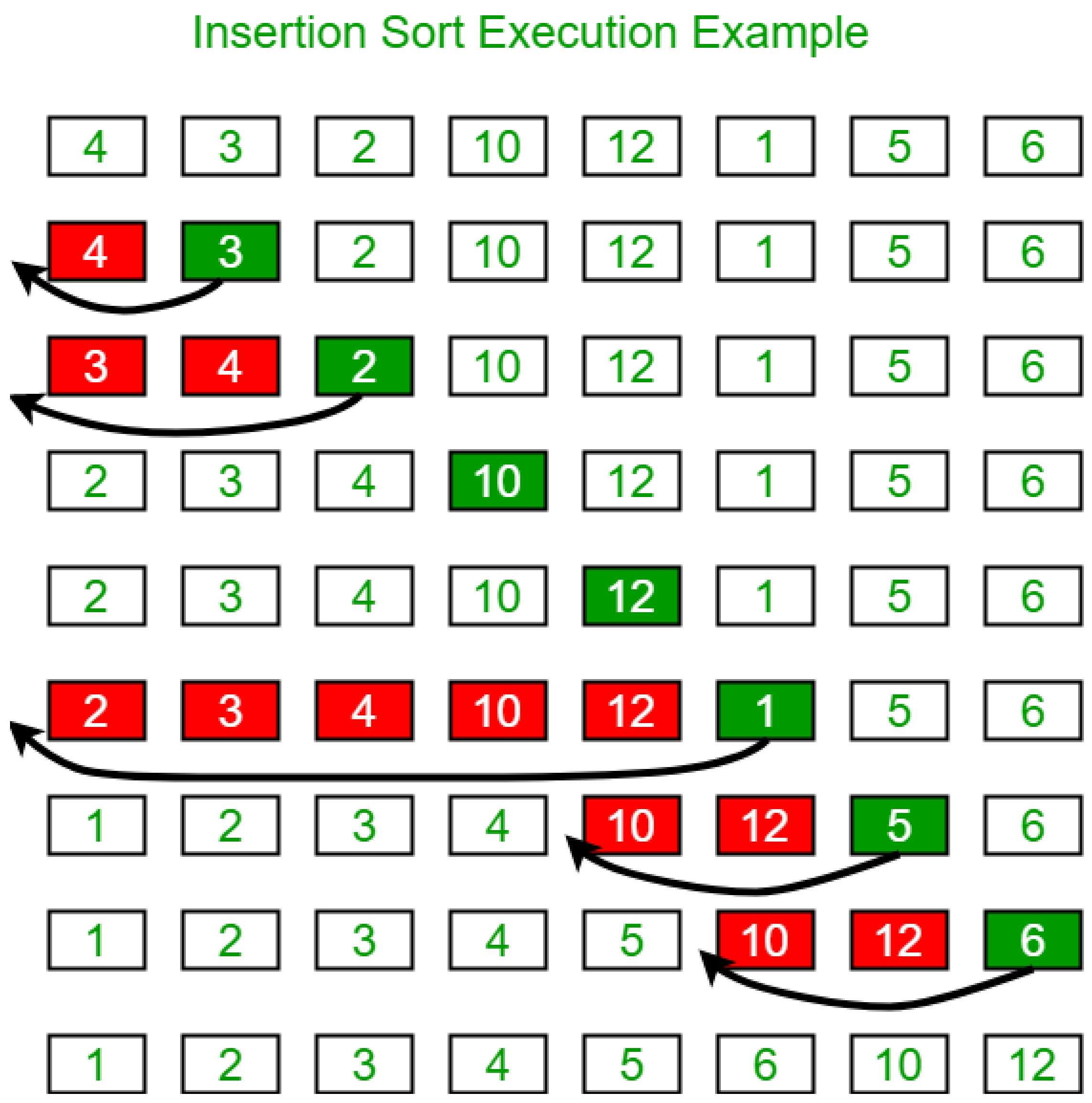
Example: Arranging the Playing cards

- Insertion Sort works in a similar manner as we arrange a deck of cards.
- Insertion sort algorithm picks elements one by one and places it to the right position where it belongs in the sorted list of elements.

2. Explain of insertion method with Examples

Example 1: The array elements {4, 3, 2, 10, 12, 1, 5, 6}

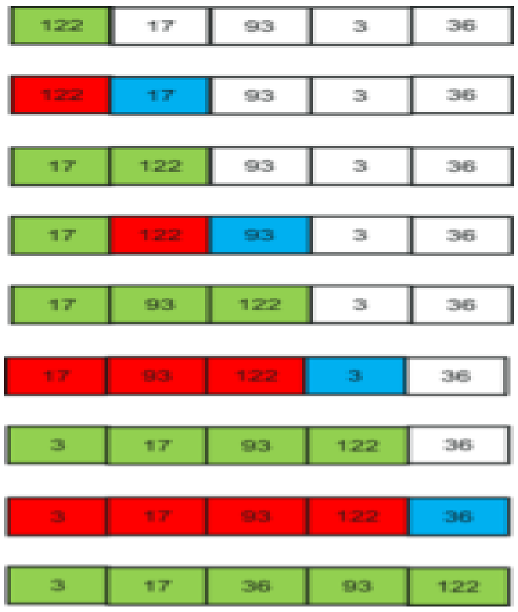
Sort the list elements
using insert Sort method



After sort: 1,2,3,4,5,6,10,12

<u>Example 2:</u> Array of elements :{ **122, 17, 93, 3, 36}**

Sort the list elements using insertion sort



Let’s understand how insertion sort is working in the above image.

<u>Example 2:</u> Array of elements :{ **122, 17, 93, 3, 36}**

The lists of elements are 5 then we required 5-1 comparisons

<u>Step1:</u> {122, **17,** 93, 3, 36}

for i = 1 then iterate the 17 (i.e 2nd element) to 36 (last element)

i = 1. Then pick up 17 elements is smaller than 122,then inserts 17 before 122

Then new list is 17, 122, 93, 3, 36

<u>Step 2</u>: 17, 122, **93,** 3, 36

i = 2. Then pick up 93 element, check preceding elements in array (i.e 17,122) and inserts 93 proper position i.e before 122

Then new list is 17, 93,122, 3, 36

<u>Step 3</u>: 17, 93,122, **3,** 36

i = 3. Then pick up 3 element, check preceding elements in array (i.e 17,93,122) and inserts 3 proper position i.e before 17

Then new list is 3, 17, 93, 122, 36

<u>Step 4:</u> 3, 17, 93, 122, **36**

i = 4. Then pick up 36 element, check preceding elements in array (i.e 3,17,93,122) and inserts 36 proper position i.e before 93

Then new list is 3, 17, 36, 93, 122

The sorted array is: 3, 17, 36, 93, 122

3. <u>Algorithm for Insertion Sort</u>

Step 1- start

Step 2- Declare array [50]

Step 3- Read the array elements from keyboard

Step 4- If the element is the first one, it is already sorted.

Step 5 – Move to next element, select another element in array

Step 6- Compare the current element with all preceding elements in the sorted array and insert proper position.

Step 7 – Repeat step 5 to 6 until the complete list is sorted

Step 8- Stop

<u>Algorithm for Insertion Sort in C</u>

Let ARR is an array with N elements

```
1. Read ARR

2. Repeat step 3 to 8 for I=1 to N-1

3. Set Temp=ARR[I]

4. Set J=I-1

5. Repeat step 6 and 7 while Temp<ARR[J] AND J>=0

6. Set ARR[J+1]=ARR[J] [Moves element forward]

7. Set J=J-1

        [End of step 5 inner loop]

8. Set ARR[J+1]=Temp [Insert element in proper place]

 [End of step 2 outer loop]

9. Stop
```

<u>Insertion sort program using without function</u>

```
#include<stdio.h>
void main()
{
   int  i, j, temp, n, a[50];
   printf("Enter the size of array:\n ");
   scanf("%d",&n);
 printf("Enter the elements :\n ");
   for(i = 0; i < n; i++)
```

```
        scanf("%d",&a[i]); // 1-D array

    for(i = 1; i < n; i++)
    {
        temp = a[i];
        for(j = i; j > 0 && a[j-1] > temp; j--)
            a[j] = a[j-1];
        a[j] = temp;
    }
printf("\n The sorted elements are ::\n ");
    for(i = 0; i < n; i++)
        printf("%d  ",a[i]);
    printf("\n");
}
```

Output:

```
Enter the size of array:

7

Enter the elements:

10 30 80 20 60 40 50

The sorted elements are:

 10 20 30 40 50 60 80
```

Insertion sort program using with function

```
#include<stdio.h>
void InsertionSort(int a[50], int n)
{
    int j, i;
    int temp;
    for(i = 1; i < n; i++)
    {
        temp = a[i];
        for(j = i; j > 0 && a[j-1] > temp; j--)
            a[j] = a[j-1];
        a[j] = temp;
    }
}
 void main()
{
    int i, n, a[50];
    printf("Enter the size of array:\n ");
    scanf("%d",&n);
    printf("Enter the elements :\n ");
    for(i = 0; i < n; i++)
    {
        scanf("%d",&a[i]);
    }
    InsertionSort(a,n); //Function call
    printf("\nThe sorted elements are ::\n ");
    for(i = 0; i < n; i++)
        printf("%d  ",a[i]);
    printf("\n");
}
```

Output:

```
Enter the size of array:

7

Enter the elements:

10 30 80 20 60 40 50

The sorted elements are:

 10 20 30 40 50 60 80
```

Insertion sort program using with function (while loop)

```
#include<stdio.h>

int main()
{
    int i,j,n,temp,a[30];

    printf("Enter the number of elements:\n");

    scanf("%d",&n);

    printf("\nEnter the elements:\n");

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

    {

        scanf("%d",&a[i]);

    }


    for(i=1;i<=n-1;i++)

    {

        temp=a[i];

        j=i-1;


        while((temp<a[j])&&(j>=0))
        {
           a[j+1]=a[j];    //moves element forward
           j=j-1;
        }
        a[j+1]=temp;    //insert element in proper place
     }
     printf("\n Sorted list is as follows:\n");
     for(i=0;i<n;i++)
     {
        printf("%d ",a[i]);
     }
     return 0;
}
```

<u>OUTPUT:</u>

```
Enter the number of elements:

7

Enter the elements:

10 30 80 20 60 40 50

Sorted list is as follows:

 10 20 30 40 50 60 80
```

<u>TIME COMPLEXITY:</u>

- insertion sort, the time complexity is of the order **<u>O(n)</u>** **(best case)**
- And in the **average or worst case scenario the** complexity is of the order O (n2).

Time complexity of Insertion sort:

Complexity	**Best Case**	**Average Case**	**Worst Case**

Time	O(n)	$O(n^2)$	$O(n^2)$
Space			O(1)

Test case 1 – Best case: Time complexity: O(n)

Here, the elements are already sorted

Enter 3 integers

{3, 31, 66}

Sorted list in ascending order:

{3, 31, 66}

Test case 2 – Average case: Time complexity: $O(n^2)$

Here, the elements are entered in random order

Enter 6 integers

{4, 6, 1, 2, 5, 3}

Sorted list in ascending order:

{1, 2, 3, 4, 5, 6 }

Test case 3 – Worst case: Time complexity: $O(n^2)$

Here, the elements are reverse sorted.

Enter 5 integers

{9, 8, 6, 3, 1}

Sorted list in ascending order:

{1, 3, 6, 8, 9}

Advantages of insertion sort

1. Efficient for small sets of data
2. Simple to implement
3. Passes through the array only once.
4. Stable
5. They are adaptive; efficient for data sets that are already sorted.

Dis-Advantages of insertion sort

1. Less efficient on larger list and arrays.
2. As the number of elements increases the performance of the program would be slow.

SELECTION SORT

Outline

1. Definition of selection sort
2. Explain of selection sort method with Example
3. selection sort Algorithm(pseudocode)
4. selection sort program(with and without functions)
5. Time complexity and space complexity
6. Advantages and Dis-advantages

1. Definition of selection sort

In this Selection sort method iterates through the array and finds the smallest number in the array and swaps the first element (**if it is smaller than the first element)** with that smallest element. Next, it goes on sub array to find the second minimum element and swap their positions, so on until all elements are sorted in an array.

- Time Complexity of selection sort is $O(n^2)$
- Auxiliary Space Complexity: O(1)

<u>Another definition:</u>

A Selection Sort is a Sorting algorithm which finds the smallest element in the array and swaps with the first element then with the second element and continues until the entire array is sorted.

<u>Selection Sort</u>

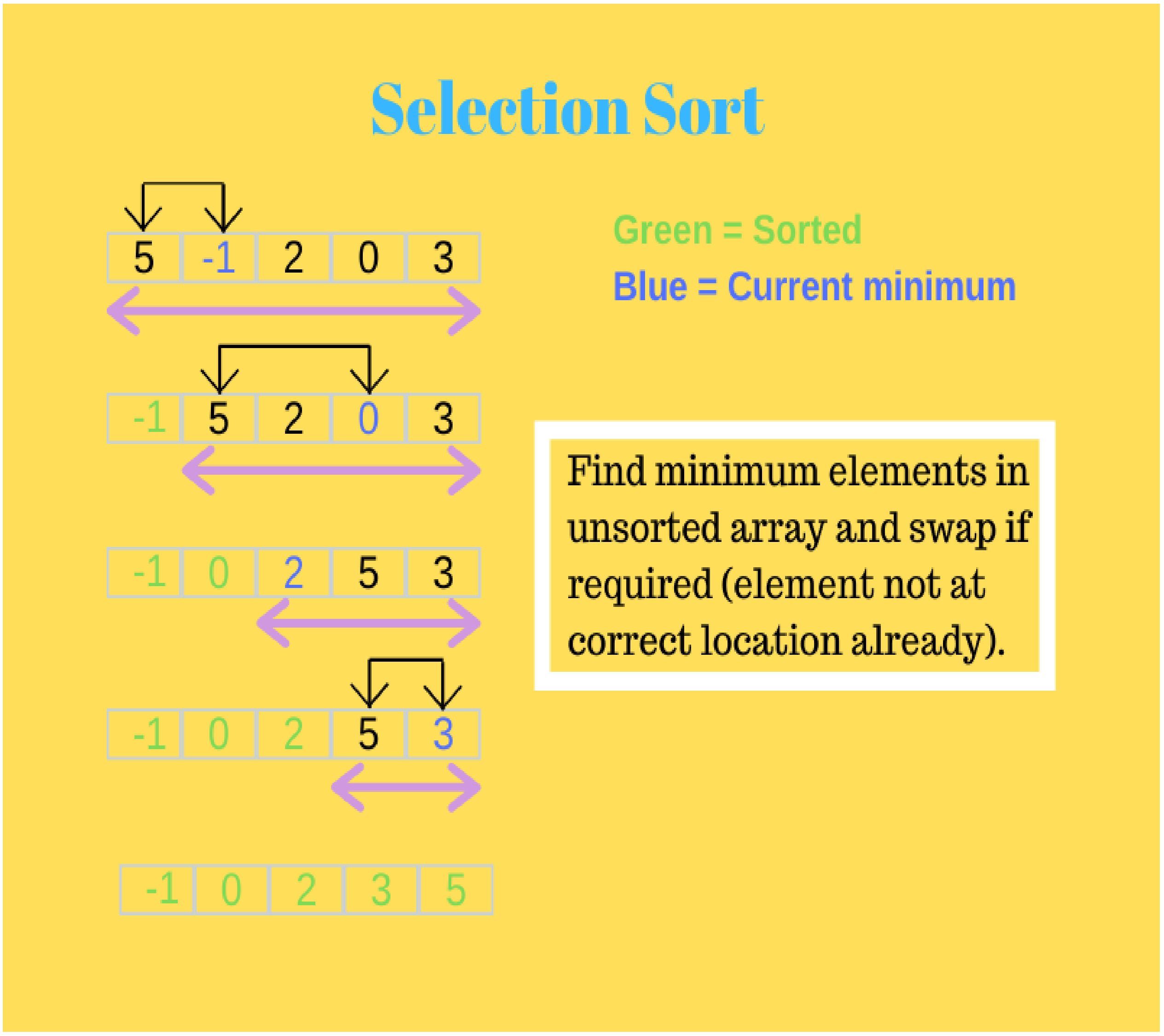
The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two sub-arrays in a given array.
1) The sub-array which is already sorted.
2) Remaining sub-array which is unsorted.

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

2. **<u>Explain of selection sort method with Example</u>**

<u>Example 1</u>: The array elements {5,-1, 2, 0, 3}

Sort the list elements using selection Sort method



Find minimum elements in unsorted array and swap if required (element not at correct location already).

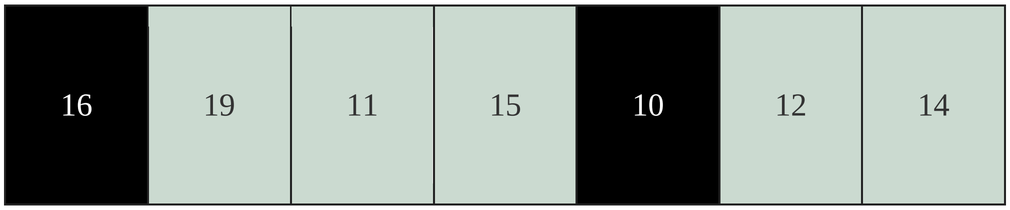
<u>Example 2</u>: The array elements {16, 19, 11, 15, 10, 12, 14}

Sort the list elements using selection Sort method

<u>Initial array</u>

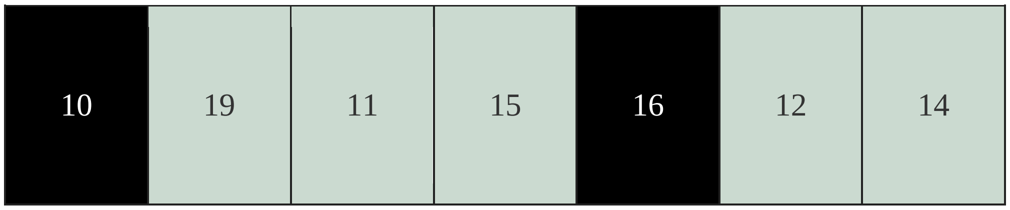
16	19	11	15	10	12	14

Step 1: First iteration:



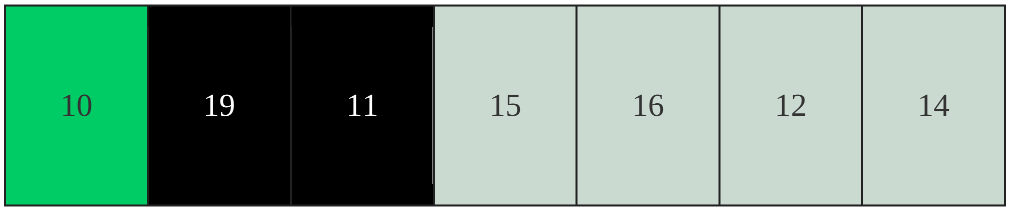
- For the **first position:** the whole list/array of elements are scanned sequentially. In the array presently 16 is stored, we search the whole list and find minimum (smallest) element i.e. 10 is the smallest value.

Then swap 16 with 10, which happen to be the minimum value in the list, appear in the first position of the sorted list.

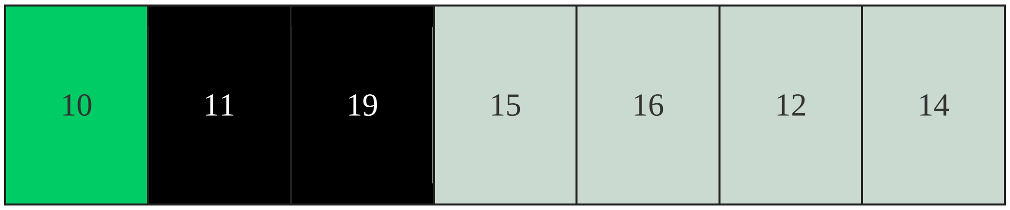


Step 2: Second iteration

- For the second position, where 19 is there, we start scanning the rest of the list in a linear manner

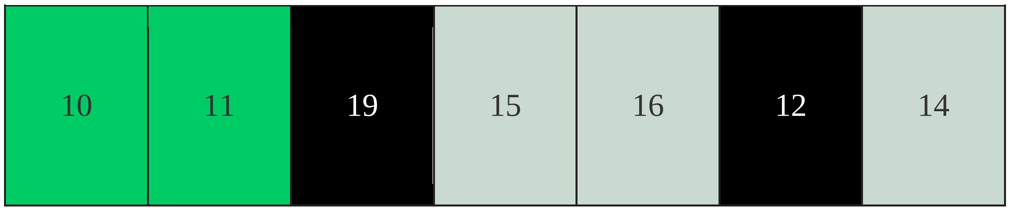


We find that 11 is the second lowest value in the list and it should appear at the second place. We swap 19 with 11 values.

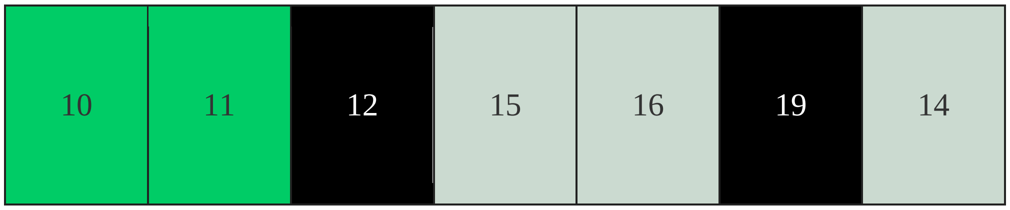


Step 3: Third iteration

- ➢ For the third position, where 19 is there, we start scanning the rest of the list in a linear manner

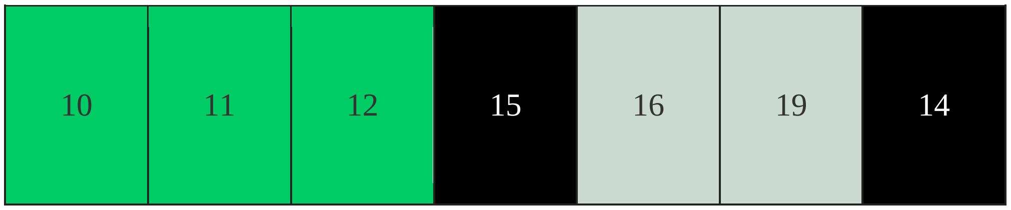


We find that 12 is the third lowest value in the list and it should appear at the third place. We swap 19 with 12 values.

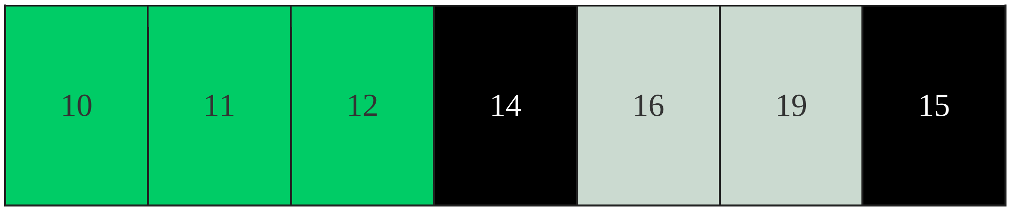


Step 4: Fourth iteration

- ➢ For the fourth position, where 15 is there, we start scanning the rest of the list in a linear manner

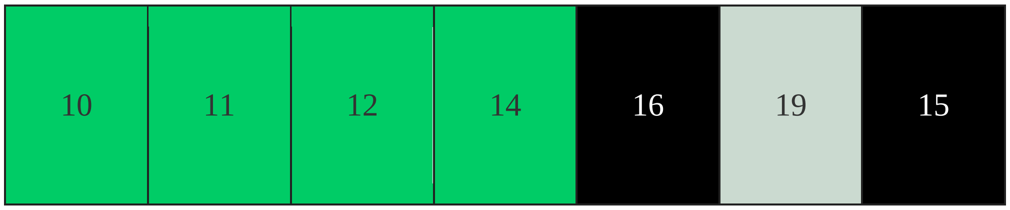


We find that 14 is the fourth lowest value in the list and it should appear at the fourth place. We swap 15 with 14 values.

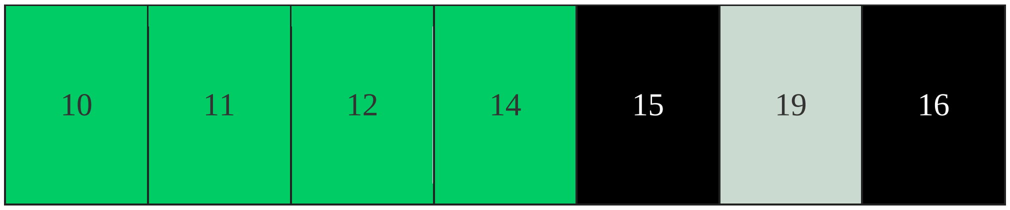


<u>**Step 5:** Fifth iteration</u>

- ➢ For the fifth position, where 16 is there, we start scanning the rest of the list in a linear manner

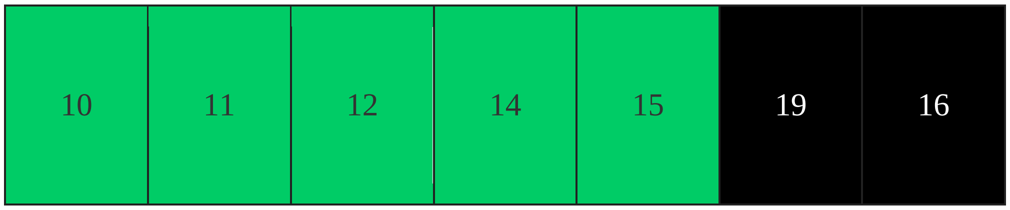


We find that 15 is the fifth lowest value in the list and it should appear at the fifth place. We swap 16 with 15 values.



Step 6: Sixth iteration

- For the sixth position, where 19 is there, we start scanning the rest of the list in a linear manner



We find that 16 is the sixth lowest value in the list and it should appear at the sixth place. We swap 19 with 16 values.

10	11	12	14	15	16	19

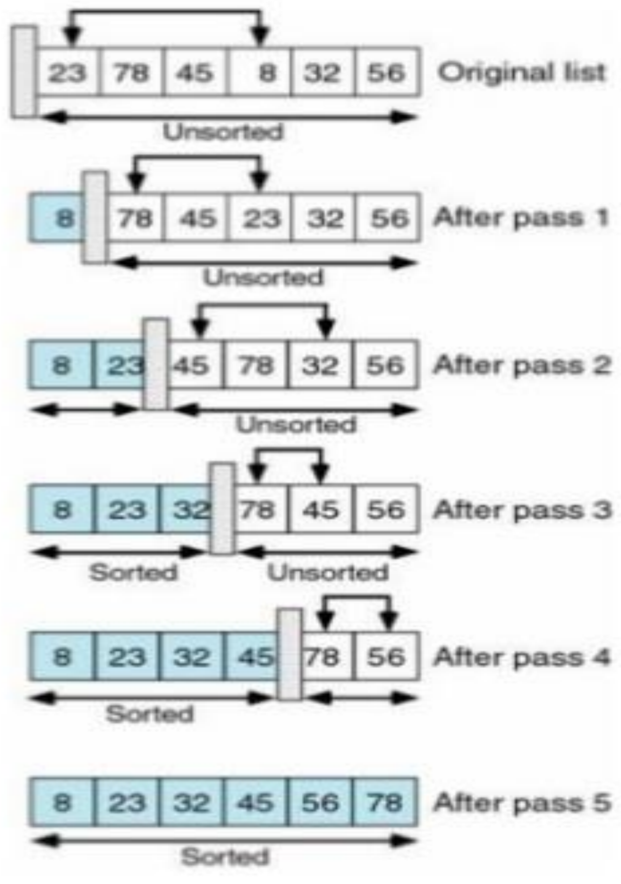
<u>Step 7</u><u>: Final array</u>

Finally sorted array

10	11	12	14	15	16	19

<u>Example 3</u>: The array elements {23, 78, 45, 8,32,56}

Sort the list elements using selection Sort method



3. Selection sort Algorithm(pseudocode)

Step 1: start

Step2: Read the array elements from keyboard

Step 3: Start from the first element in the array and search for the smallest element in the array.

Step 4: Swap the first element with the smallest element of the array.

Step 5: Take a sub-array (excluding the first element of the array as it is at its place) and search for the smallest number in the sub-array (second smallest number of the entire array) and swap it with the first element of the array (second element of the entire array).

Step 6: Repeat the steps 5 for the new sub-arrays until the array gets sorted.

Step 7: stop

Algorithm for Selection Sort:

Step 1 − Set min to the first location

Step 2 − Search the minimum element in the array

Step 3 – swap the first location with the minimum value in the array

Step 4 – assign the second element as min.

Step 5 − Repeat the process until we get a sorted array.

.

<u>Program for Selection Sort:</u>

```
#include<stdio.h>
void main()
{
   int a[100],n,i,j,min,temp;

   printf("\n Enter the Number of Elements:\n ");
   scanf("%d",&n);

   printf("\n Enter %d Elements: ",n);
   for(i=0;i<n;i++)
      scanf("%d",&a[i]); // 1-D array
   for(i=0;i<n-1;i++)
   {
      min=i;
      for(j=i+1;j<n; j++)
      {
         if(a[min]>a[j])
         min=j;
      }
      if(min!=i)
      {
         temp=a[i];
         a[i]=a[min];
         a[min]=temp;
      }
   }

   printf("\n The Sorted array in ascending order:\n ");

         for(i=0;i<n;i++)
      printf("%d ",a[i]);

}
```

Output:

```
Enter the Number of Elements:

5

Enter 5 Elements:

4 1 9 3 6

The Sorted array in ascending order:

1 3 4 6 9
```

Program analysis:

Step for sorting the array elements:

Then it assigns **i=0** and the loop continues till the condition of for loop is true.

1.1. i<n-1 (**0<4**) for loop condition is true

min=i So, **min=0**

Then it assigns j=i+1 (**j=1**) and the loop continues till the condition of for loop is true.

1.1.1. j<n (**1<5**) for loop condition is true

a[0]>a[1] (**4>1**) if condition is true

min=j So, **min=1**

j++ (**j=j+1**) So, **j=2**

1.1.2. j<n (**2<5**) for loop condition is true

a[1]>a[2] (**1>9**) if condition is false

j++ (**j=j+1**) So, **j=3**

1.1.3. j<n (**3<5**) for loop condition is true

a[1]>a[3] (**1>3**) if condition is false

j++ (**j=j+1**) So, **j=4**

1.1.4. j<n (**4<5**) for loop condition is true a[1]>a[4]   (**1>6**)   if condition is false

j++   (**j=j+1**)   So, **j=5**

1.1.5.   j<n   (**5<5**)   for loop condition is false

It comes out of the for loop.

min!=i   (**1!=0**)   if condition is true

temp=a[i]   (temp=a[0])   So, **temp=4**

a[i]=a[min]   (a[0]=a[1])   So, **a[0]=1**

a[min]=temp   (a[1]=temp)   So, **a[1]=4**

i++   (**i=i+1**)   So, **i=1**

So after first iteration the array is: **1 4 9 3 6**

Do the same process until sorted

<u>Program for Selection Sort(function)</u>

```
#include<stdio.h>
Void selectionsort(int a[100], int n); // function declaration
void main()
{
   int a[100],n,i;

   printf("\n Enter the Number of Elements:\n ");
   scanf("%d",&n);
   printf("\n Enter %d Elements: ",n);
   for(i=0;i<n;i++)
      scanf("%d",&a[i]);

 selectionsort(a, n); // function call

}

  Void selectionsort(int a[100], int n); // function definition

              {
         int i,j,min,temp;

   for(i=0;i<n-1;i++)

   {

      min=i;

      for(j=i+1;j<n; j++)

      {

         if(a[min]>a[j])

         min=j;

      }

      if(min!=i)

      {

         temp=a[i];

         a[i]=a[min];

         a[min]=temp;

      }

   }

   printf("\n The Sorted array in ascending order:\n ");

         for(i=0;i<n;i++)
      printf("%d ",a[i]);

}
```

Output:

```
Enter the Number of Elements:

5

Enter 5 Elements:

4 1 9 3 6

The Sorted array in ascending order:

1 3 4 6 9
```

Time Complexity

- **Selection sort Time Complexity Analysis**

Selecting the lowest element requires scanning all n elements (this takes n - 1 comparisons) and then swapping it into the first position.

Finding the next lowest element requires scanning the remaining n - 1 element and so on,
= (n - 1) + (n - 2) + ... + 2 + 1 = n(n - 1) / 2
= O(n^2) comparisons.

Selection sort Time Complexity

Complexity	**Best Case**	**Average Case**	**Worst Case**
Time	$O(n^2)$	$O(n^2)$	$O(n^2)$
Space			O(1)

Advantages of selection sort

1. Performs well on a small list of elements
2. Because it is an in-place sorting algorithm, no additional temporary storage is required

Disadvantages of selection sort

1. The selection sort is its poor efficiency when dealing with a huge list of items.
2. The selection sort requires n-squared number of steps for sorting n elements.
3. Quick Sort is much more efficient than selection sort

Time Complexities of all Sorting Algorithms

Algorithm	**Time Complexity**		
	Best	**Average**	**Worst**
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Bubble Sort	O(n)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Heap Sort	0(n log(n))	O(n log(n))	O(n log(n))
Quick Sort	0(n log(n))	0(n log(n))	O(n^2)

Algorithm	Time Complexity		
	Best	**Average**	**Worst**
Merge Sort	0(n log(n))	0(n log(n))	O(n log(n))
Radix Sort	0(nk)	0(nk)	O(nk)