Unit II

Sorting algorithms: Insertion, bubble, selection, quick and merge sort; Comparison of Sort algorithms. Searching techniques: Linear and Binary search.

Sorting Algorithms

Sorting

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order.

The sorting method can be divided into two categories.

Internal Sorting

Here all the data elements that is be sorted can be process at a time.

External Sorting

External sorting is a term for a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead, they must reside in the slower external memory (usually a hard drive).

Stable and Not Stable Sorting

If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called **stable sorting.**

If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called **unstable sorting**.

In-place Sorting and Not-in-place Sorting

Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms **do not require any extra space and sorting is said to happen in-place**, or for example, within the array itself. This is **called in-place sorting. Bubble sort** is an example of in-place sorting.

However, in some sorting algorithms, the program **requires space** which is more than or equal to the elements being sorted. Sorting which uses equal or more space is **called not-in-place sorting. Merge-sort** is an example of not-in-place sorting.

Insertion Sort

Insertion Sort is an efficient algorithm for sorting a small number of elements.

It is similar to sort a hand of playing cards.

An element which is to be inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

Insertion sort is a comparison based sorting algorithm which sorts the array by **shifting elements one by one from an unsorted sub-array to the sorted subarray**. With each iteration, an element from the input is pick and inserts in the sorted list at the correct location.

Step by Step Process

The insertion sort algorithm is performed using the following steps.

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

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

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

Algorithm

Insertion_Sort(A, n)

Let 'A' be an array of n elements which we want to sort. 'temp' be a temporary variable to exchange the two values.

Insertion_Sort(A, n)

1. **Repeat steps (2) to (4) for i=1 to (n-1)**
2. **Set temp= A[i]**
   Set j=i-1
3. **while temp < A[j] and j>=0 perform**

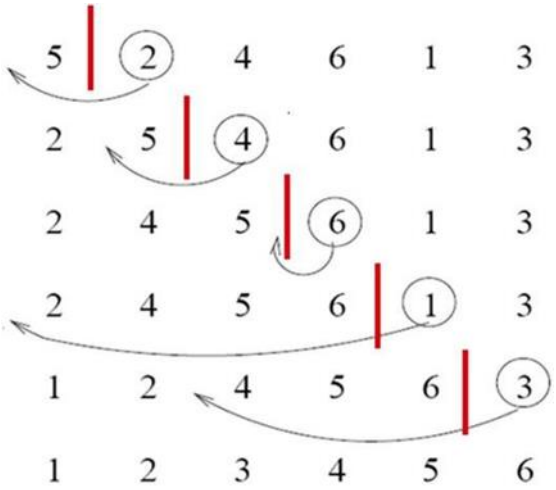
Set A[j+1]=A[j]

Set j=j-1

4. **Set A[j+1]= temp**
5. **Exit**

(Also write a suitable example of the algorithm here)

Example



Insertion Sort Complexity

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

// C++ program for insertion sort

Insertion sort: number of comparisons and exchanges for given data sets.

```
#include<iostream.h>
#include<conio.h>
class insertion
{
int i,n,k,a[10],pos;
public:
```

```
void get();
void sort();
void display();
};
void insertion::get()
{
cout<<"\nEnter the size:";
cin>>n;
cout<<"\nEnter the array:";
for(i=1;i<=n;i++)
{
cin>>a[i];
}
}
void insertion::sort()
{
for(i=2;i<=n;i++)
{
k=a[i];
pos=i;
while(pos>1&&a[pos-1]>k)
{
a[pos]=a[pos-1];
pos=pos-1;
a[pos]=k;
}
}
}
```

```
void insertion::display()
{
cout<<"\nSorted array:";
for(i=1;i<=n;i++)
{
cout<<"\n"<<a[i];
} }
void main()
{
clrscr();
insertion in;
in.get();
in.sort();
in.display();
getch();
}
```

OUTPUT:

Enter the size: 4

Enter the array:

3

2

5

1

Sorted array:

1

2

3

5

Bubble sort

- The working procedure of bubble sort is simplest.
- Bubble sort works on the **repeatedly swapping of adjacent elements until they are not in the intended order.** It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water.
- Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.
- Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world.
- **It is not suitable for large data sets**. The average and worst-case complexity of Bubble sort is $O(n^2)$, where n is a number of items.

Bubble short is majorly used where -

- **complexity does not matter**
- **simple and short code is preferred**

Algorithm

In the algorithm given below, suppose arr is an array of n elements. The assumed swap function in the algorithm will swap the values of given array elements.

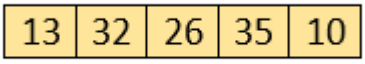
1. **begin BubbleSort(arr)**
2. **for all array elements**
3. **if arr[i] > arr[i+1]**
4. **swap(arr[i], arr[i+1])**
5. **end if**
6. **end for**
7. **return arr**
8. **end BubbleSort**

Working of Bubble sort Algorithm

Now, let's see the working of Bubble sort Algorithm.

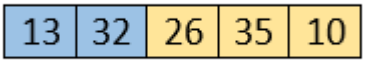
To understand the working of bubble sort algorithm, let's take an unsorted array. We are taking a short and accurate array, as we know the complexity of bubble sort is $O(n^2)$.

Let the elements of array are -

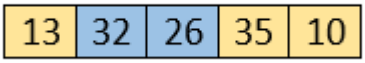


First Pass

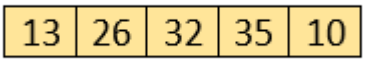
Sorting will start from the initial two elements. Let compare them to check which is greater.



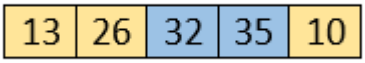
Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.



Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like -

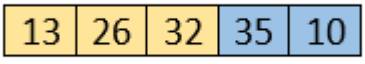


Now, compare 32 and 35.

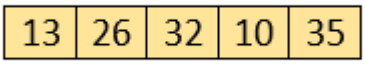


Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.

Now, the comparison will be in between 35 and 10.



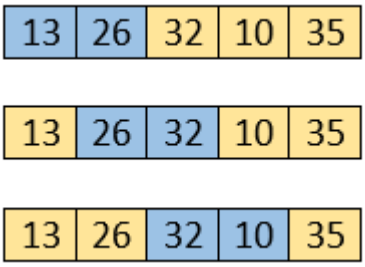
Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be -



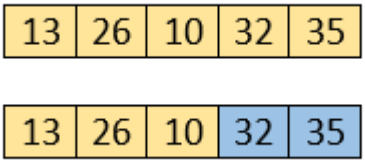
Now, move to the second iteration.

Second Pass

The same process will be followed for second iteration.



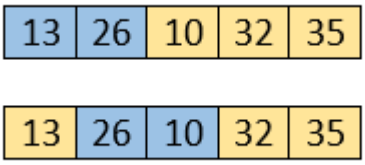
Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -



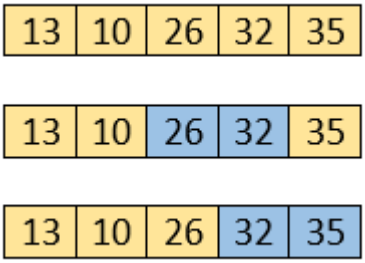
Now, move to the third iteration.

Third Pass

The same process will be followed for third iteration.



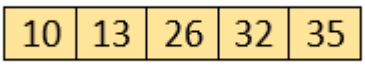
Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -



Now, move to the fourth iteration.

Fourth pass

Similarly, after the fourth iteration, the array will be -



Hence, there is no swapping required, so the array is completely sorted.

Bubble sort complexity

Now, let's see the time complexity of bubble sort in the best case, average case, and worst case. We will also see the space complexity of bubble sort.

1. Time Complexity

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

- **Best Case Complexity** - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is O(n).
- **Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is $O(n^2)$.

- **Worst Case Complexity** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is $O(n^2)$.

2. Space Complexity

Space Complexity	O(1)
Stable	YES

- The space complexity of bubble sort is O(1). It is because, in bubble sort, an extra variable is required for swapping.
- The space complexity of optimized bubble sort is O(2). It is because two extra variables are required in optimized bubble sort.

Now, let's discuss the optimized bubble sort algorithm.

Optimized Bubble sort Algorithm

In the bubble sort algorithm, comparisons are made even when the array is already sorted. Because of that, the execution time increases.

To solve it, we can use an extra variable *swapped*. It is set to true if swapping requires; otherwise, it is set to false.

It will be helpful, as suppose after an iteration, if there is no swapping required, the value of variable swapped will be false. It means that the elements are already sorted, and no further iterations are required.

This method will reduce the execution time and also optimizes the bubble sort.

Algorithm for optimized bubble sort

1. **bubbleSort(array)**
2. **n = length(array)**
3. **repeat**
4. **swapped = false**
5. **for i = 1 to n - 1**

6. if array[i - 1] > array[i], then
7. swap(array[i - 1], array[i])
8. swapped = true
9. end if
10. end for
11. n = n - 1
12. until not swapped
13. end bubbleSort

// C++ program for Bubble sort:

Bubble sort: Print number of comparisons and exchanges for given data sets.

```
#include<iostream.h>
#include<conio.h>
class sort
{
int a[20],i,n,j,t;
public:
void read();
void bubsort();
void display();
};
void sort::read()
{
cout<<"\nenter the number of elements:";
cin>>n;
cout<<"\nenter the elements:";
for(i=1;i<=n;i++)
{
cin>>a[i];
```

```
}
}
void sort::bubsort()
{
for(i=1;i<=n-1;i++)
{
for(j=1;j<=n-i;j++)
{
if(a[j]>a[j+1])
{
t=a[j];
a[j]=a[j+1];
a[j+1]=t;
}
}
}
}
void sort::display()
{
cout<<"\nthe sorted array is:";
for(i=1;i<=n;i++)
{
cout<<a[i]<<"\t";
}
}
int main()
{
clrscr();
```

```
sort o;
o.read();
o.bubsort();
o.display();
getch();
return 0;
}
```

OUTPUT:

Enter the number of elements: 5

Enter the elements: 9 4 6 3 7

The sorted array is: 3 4 6 7 9

Selection Sort

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

Let us see what the idea of Selection Sort is:

- **First it finds the smallest element in the array.**
- **Exchange that smallest element with the element at the first position.**
- **Then find the second smallest element and exchange that element with the element at the second position.**
- **This process continues until the complete array is sorted.**

Algorithm

SELECTION-SORT(A,n) Here A is a one dimensional array to be sorted and n is the size of the array

SELECTION-SORT(A, n)

1. Repeat steps (2) to (4) for j = 1 to n-1

2. [initialize smallest with j]

Set smallest = j

3. Repeat for i = j+1 to n

if A[i] < A[smallest] then

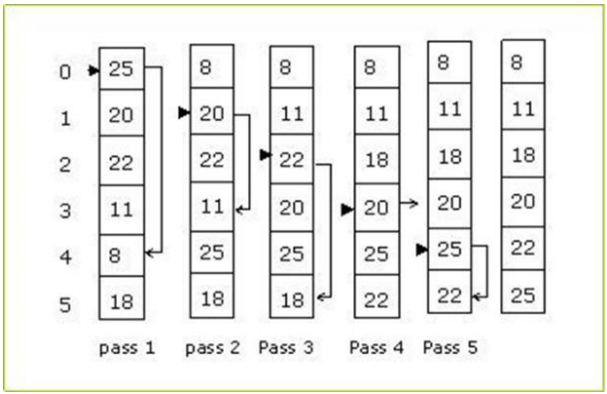
Set smallest = i

4. Exchange A[j] and A[smallest]

5. Exit

(Also write a suitable example of the algorithm here)

Let us see how selection sort works:

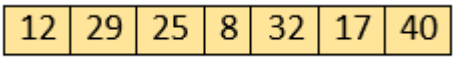


Working of Selection sort Algorithm

Now, let's see the working of the Selection sort Algorithm.

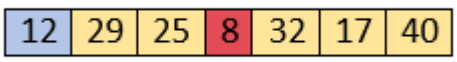
To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to understand the Selection sort via an example.

Let the elements of array are -

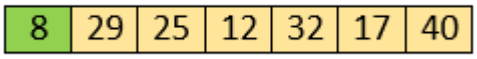


Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.

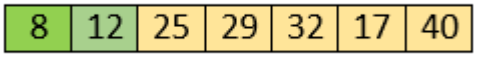


So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.

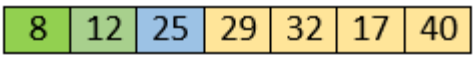


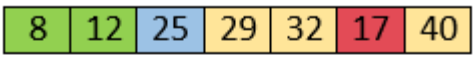
For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

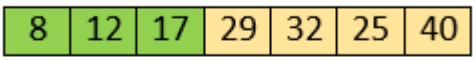
Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.

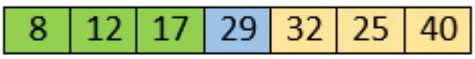


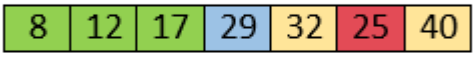
The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.

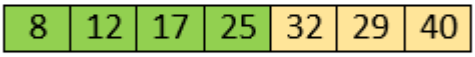


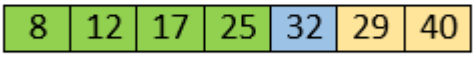












Now, the array is completely sorted.

// C++ program for Selection sort:

Selection sort: Print number of comparisons and exchanges for given data sets .

```
#include<iostream.h>
#include<conio.h>
class sort
{
int i,j,n,a[20];
public:
void read();
void selectsort();
void display();
int minimum(int i);
};
void sort::read()
{
cout<<"\nEnter number of elements:";
cin>>n;
cout<<"\nEnter the elements:\n";
for(i=1;i<=n;i++)
{
cin>>a[i];
}
}
void sort::selectsort()
{
for(i=1;i<=n;i++)
{
```

```
int min_index=minimum(i);
int temp=a[i];
a[i]=a[min_index];
a[min_index]=temp;
}
}
int sort::minimum(int i)
{
int min_index=i;
for(j=i+1;j<=n;j++)
{
if(a[j]<a[min_index])
min_index=j;
}
return min_index;
}
void sort::display()
{
cout<<"\n Resultant array:\n";
for(i=1;i<=n;i++)
{
cout<<a[i]<<"\t";
}
}
void main()
{
clrscr();
sort s;
```

```
s.read();
s.selectsort();
s.display();
getch();
}
```

OUTPUT:

Enter the number of elements: 5

Enter the elements: 2 5 8 1 9

Resultant array: 1 2 5 8 9

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

Quick Sort

It is one of the most popular sorting techniques. This algorithm works by partitioning the array to be sorted and each partition is in turn sorted recursively. It is a **faster and highly efficient** sorting algorithm

This algorithm follows the **divide and conquer approach**. Divide and conquer is a technique of breaking down the algorithms into sub problems, then solving the sub problems, and combining the results back together to solve the original problem.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

In partition, one of the element is chosen as the key value and rest of the array elements are grouped into two partitions such that, **one partition contains elements smaller than the key value**. **Another partition contains elements larger than the key value.**

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Quicksort **picks an element as pivot,** and then **it partitions the given array around the picked pivot element**. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.

Algorithm

Quick_Sort(a, l, h)

Where a is the array to be sorted, l is the position of the first element in the list, h is the position of the last element in the list.

Quick_Sort(a, l, h)

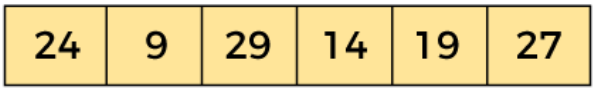
1. [Initialization]
   low= l high= h
   key=a[(l+h)/2]
2. Repeat through step 7 while(low <= high)
3. Repeat step 4 while(a[low]<key)
4. low=low+1
5. repeat step 6 while(a[high] > key)
6. high=high-1
7. if(low<=high)
   a) temp=a[low]
   b) a[low]=a[high]
   c) a[high]=temp
   d) low=low+1
   e) high=high-1
8. if(l<high) Quick_Sort(a, l , high)
9. if(low<h) Quick_Sort(a, low , h)
10. Exit

Working of Quick Sort Algorithm

Now, let's see the working of the Quicksort Algorithm.

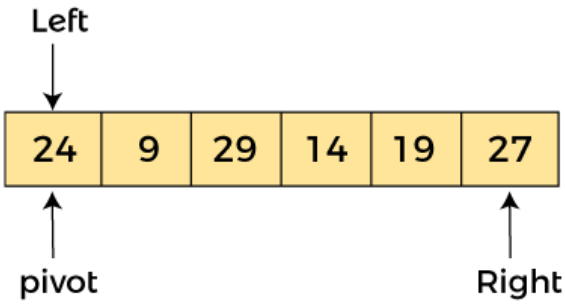
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -

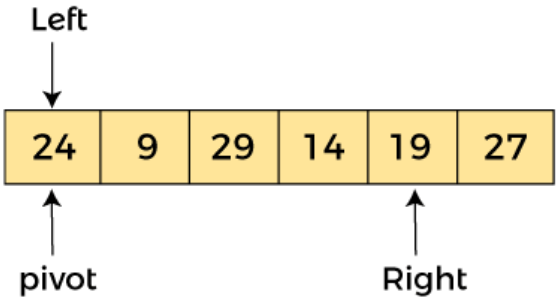


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

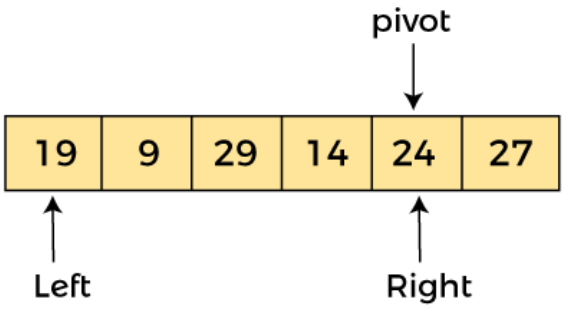


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



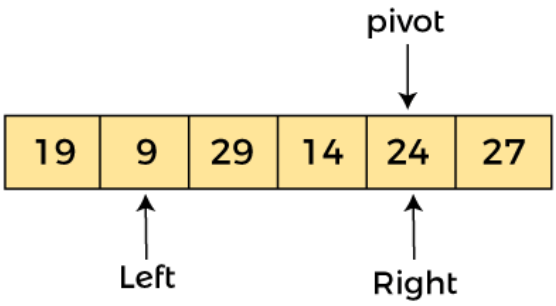
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

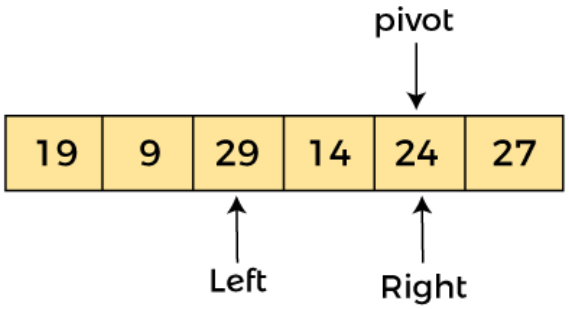


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

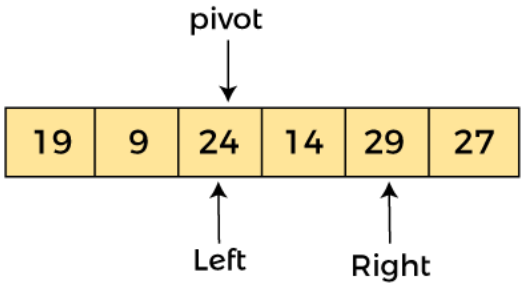
As a[pivot] > a[left], so algorithm moves one position to right as -



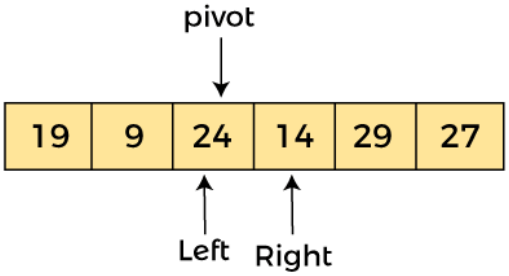
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



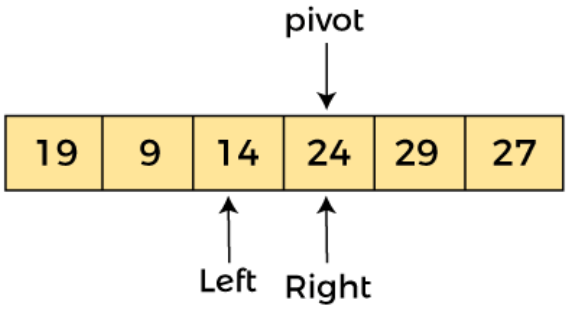
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



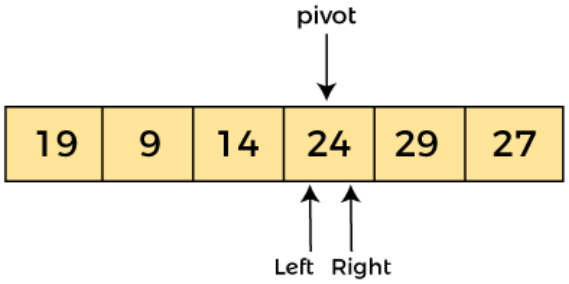
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



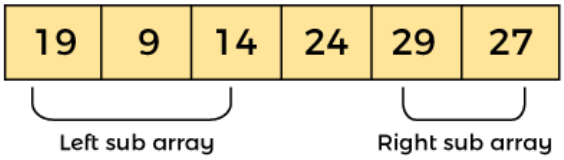
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



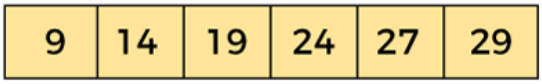
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -



Quicksort complexity

Now, let's see the time complexity of quicksort in best case, average case, and in worst case. We will also see the space complexity of quicksort.

1. Time Complexity

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

- **Best Case Complexity -** In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **O(n*logn)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **O(n*logn)**.
- **Worst Case Complexity -** In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **$O(n^2)$**.

Though the worst-case complexity of quicksort is more than other sorting algorithms such as **Merge sort** and **Heap sort**, still it is faster in practice. Worst case in quick sort rarely occurs because by changing the choice of pivot, it can be implemented in different ways. Worst case in quicksort can be avoided by choosing the right pivot element.

2. Space Complexity

Space Complexity	O(n*logn)
Stable	NO

- The space complexity of quicksort is O(n*logn).

(Also write a suitable example of the algorithm here)

Quick sort.

```
#include<iostream.h>
#include<conio.h>
class quicksort
{
int a[10],l,u,i,j,limit;
public:
void read();
void quick(int l,int u);
void display();
};
void quicksort::read()
{
cout<<"\nEnter the limit:";
cin>>limit;
cout<<"\nEnter the elements:";
for(i=0;i<limit;i++)
{
cin>>a[i];
}
i=0;
u=limit-1;
}
void quicksort::display()
{
cout<<"\nElements are...\n";
for(i=0;i<limit;i++)
```

```
{
cout<<"\n"<<a[i];
}
quick(l,u);
cout<<"\nSorted elements are...\n";
for(i=0;i<limit;i++)
{
cout<<"\n"<<a[i];
}
}
void quicksort::quick(int l,int u)
{
int p,temp;
if(l<u)
{
p=a[l];
i=l;
j=u;
while(i<j)
{
while(a[i]<=p&&i<j)
i++;
while(a[j]>p&&i<=j)
j--;
if(i<=j)
{
temp=a[i];
a[i]=a[j];
```

```
a[j]=temp;
}
}
temp=a[j];
a[j]=a[l];
a[l]=temp;
quick(l,j-1);
quick(j+1,u);
}
}
void main()
{
clrscr();
quicksort ob;
ob.read();
ob.display();
getch();
}
```

OUTPUT:

Enter the limit: 5

Enter the elements: 2 4 5 3 1

Elements are…

2

4

5

3

1

Sorted elements are…

1

2

3

4

5

Merge sort

Merge sort is a sorting technique based on divide and conquer technique.

With worst-case time complexity being **O(n log n),** it is one of the most respected algorithms.

It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the merge() function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

Working of Merge sort Algorithm

To understand merge sort, we take an unsorted array as the following

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

We further divide these arrays and we achieve atomic value which can no more be divided.

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

After the final merging, the list should look like this −

Now we should learn some programming aspects of merge sorting.

Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Step 1 − if it is only one element in the list it is already sorted, return.

Step 2 − divide the list recursively into two halves until it can no more be divided.

Step 3 − merge the smaller lists into new list in sorted order.

Merge sort complexity

Now, let's see the time complexity of merge sort in best case, average case, and in worst case. We will also see the space complexity of the merge sort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	O(n*logn)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is **O(n*logn)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is **O(n*logn)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is **O(n*logn)**.

2. Space Complexity

Space Complexity	O(n)
Stable	YES

- The space complexity of merge sort is O(n). It is because, in merge sort, an extra variable is required for swapping.

Algorithm

In the following algorithm, arr is the given array, beg is the starting element, and end is the last element of the array.

MERGE_SORT(arr, beg, end)

if beg < end

set mid = (beg + end)/2

MERGE_SORT(arr, beg, mid)

MERGE_SORT(arr, mid + 1, end)

MERGE (arr, beg, mid, end)

end of if

END MERGE_SORT

// C++ program for Merge Sort

```
#include <iostream>
using namespace std;
// Merges two subarrays of array[].
// First subarray is arr[begin..mid]
// Second subarray is arr[mid+1..end]
void merge(int array[], int const left, int const mid, int const right)
{
        auto const subArrayOne = mid - left + 1;
        auto const subArrayTwo = right - mid;

        // Create temp arrays
        auto *leftArray = new int[subArrayOne],
                *rightArray = new int[subArrayTwo];

        // Copy data to temp arrays leftArray[] and rightArray[]
        for (auto i = 0; i < subArrayOne; i++)
```

```
            leftArray[i] = array[left + i];
        for (auto j = 0; j < subArrayTwo; j++)
            rightArray[j] = array[mid + 1 + j];
        auto indexOfSubArrayOne = 0, // Initial index of first sub-array
            indexOfSubArrayTwo = 0; // Initial index of second sub-array
        int indexOfMergedArray = left; // Initial index of merged array
        // Merge the temp arrays back into array[left..right]
        while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo <
subArrayTwo) {
            if (leftArray[indexOfSubArrayOne] <=
rightArray[indexOfSubArrayTwo]) {
                array[indexOfMergedArray] =
leftArray[indexOfSubArrayOne];
                indexOfSubArrayOne++;
            }
            else {
                array[indexOfMergedArray] =
rightArray[indexOfSubArrayTwo];
                indexOfSubArrayTwo++;
            }
            indexOfMergedArray++;
        }
        // Copy the remaining elements of
        // left[], if there are any
        while (indexOfSubArrayOne < subArrayOne) {
            array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
            indexOfSubArrayOne++;
            indexOfMergedArray++;
```

```
        }
        // Copy the remaining elements of
        // right[], if there are any
        while (indexOfSubArrayTwo < subArrayTwo) {
                array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
                indexOfSubArrayTwo++;
                indexOfMergedArray++;
        }
}

// begin is for left index and end is
// right index of the sub-array
// of arr to be sorted */
void mergeSort(int array[], int const begin, int const end)
{
        if (begin >= end)
                return; // Returns recursively

        auto mid = begin + (end - begin) / 2;
        mergeSort(array, begin, mid);
        mergeSort(array, mid + 1, end);
        merge(array, begin, mid, end);
}

// UTILITY FUNCTIONS
// Function to print an array
void printArray(int A[], int size)
{
```

```
    for (auto i = 0; i < size; i++)
        cout << A[i] << " ";
}

// Driver code
int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    auto arr_size = sizeof(arr) / sizeof(arr[0]);

    cout << "Given array is \n";
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    cout << "\nSorted array is \n";
    printArray(arr, arr_size);
    return 0;
}
// This code is contributed by Mayank Tyagi
// This code was revised by Joshua Estes
```

Searching of Array Element

Searching is used to find the location where element is available or not.

There are two types of searching techniques as given below:

1. Linear or sequential searching
2. Binary searching

Linear or sequential searching

In linear search, search each element of an array one by one sequentially and see whether it is desired element or not. A search will be unsuccessful if all the elements are accessed and the desired element is not found.

Algorithm

Linear_Search(a, n, item)

Here a is a linear array with n elements, and item is a given item of information. This algorithm finds the location loc of item in a, or set loc=0 if the search is unsuccessful.

Linear_Search(data, n, item)

1. **[insert the item at the end of the array data]**
   Set data[n+1]=item
2. **[initialize counter]**
   Set loc=1
3. **Repeat while**

data[loc]!=item

Set loc=loc+1

4. **[unsuccessful]**
   If loc=n+1, then set loc=0
5. **Exit**

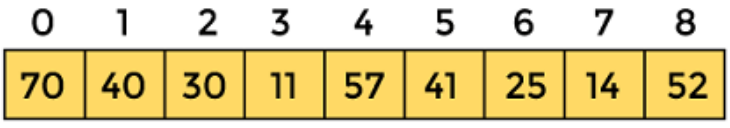
(Also write a suitable example of the algorithm here)

Working of Linear search

Now, let's see the working of the linear search Algorithm.

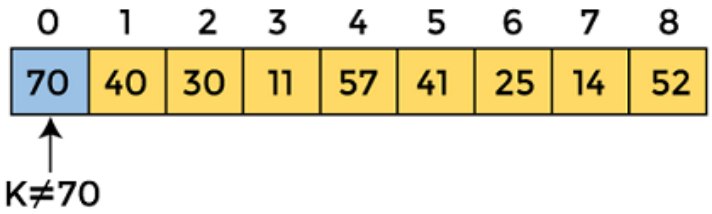
To understand the working of linear search algorithm, let's take an unsorted array. It will be easy to understand the working of linear search with an example.

Let the elements of array are -

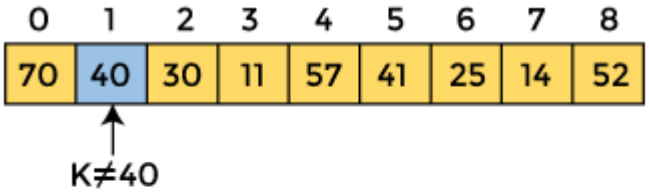


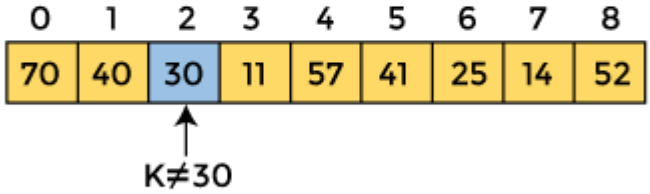
Let the element to be searched is **K = 41**

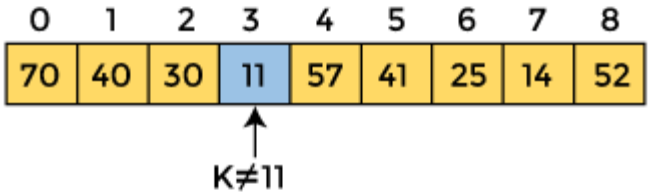
Now, start from the first element and compare **K** with each element of the array.

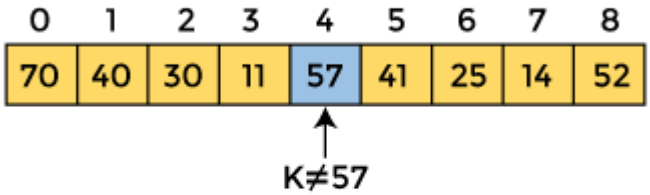


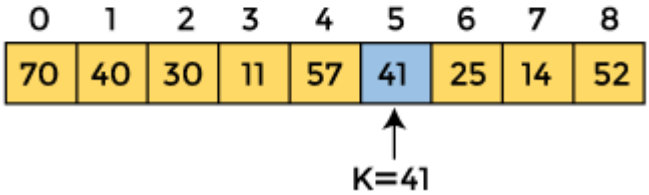
The value of **K,** i.e., **41,** is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.











Now, the element to be searched is found. So algorithm will return the index of the element matched.

Linear Search complexity

Now, let's see the time complexity of linear search in the best case, average case, and worst case. We will also see the space complexity of linear search.

1. Time Complexity

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

- **Best Case Complexity -** In Linear search, best case occurs when the element we are finding is at the first position of the array. The best-case time complexity of linear search is **O(1).**
- **Average Case Complexity -** The average case time complexity of linear search is **O(n).**
- **Worst Case Complexity -** In Linear search, the worst case occurs when the element we are looking is present at the end of the array. The worst-case in linear search could be when the target element is not present in the given array, and we have to traverse the entire array. The worst-case time complexity of linear search is **O(n).**

The time complexity of linear search is **O(n)** because every element in the array is compared only once.

2. Space Complexity

Space Complexity	O(1)

- The space complexity of linear search is O(1).

Binary Searching

Binary Search technique searches the given item in minimum possible comparisons. To do the binary search, first we had to sort the array elements.

The logic behind this technique is given below:

1. First find the middle element of the array
2. Compare the middle element with item
3. There are three cases
    a. If it is a desired element then the search is successful.
    b. If desired item is less than middle element then search only the first half of the array.
    c. If desired item is greater than middle element, search in the second half of the array.

Algorithm

Binary_Search(data, LB, UB, item)

Here data is a sorted array with lower bound LB and upper bound UB, and item is a given item of information. The beg, end, and mid denoted respectively, the beginning, end, and middle location of a segment of element of a. this algorithm finds the location of item in loc or set loc=NULL.

Binary_Search(data, LB, UB, item)

1. [Initialize segment variable]
    Set beg=LB, end=UB, mid=int(beg+end)/2
2. Repeat steps 3 and 4 while beg<=end and data[mid]!=item
3. If item < data[mid], then
    Set end=mid-1
    Else
    Set beg=mid+1
4. Set mid= int(beg+end)/2

5. If a[mid]=item them

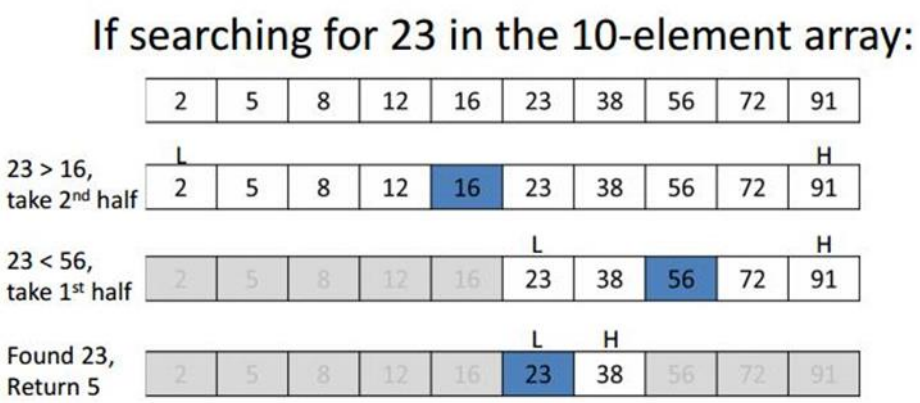
   Set loc = mid

   Else

   Set loc=NULL

6. Exit

(Also write a suitable example of the algorithm here) Example



Working of Binary search

Now, let's see the working of the Binary Search Algorithm.

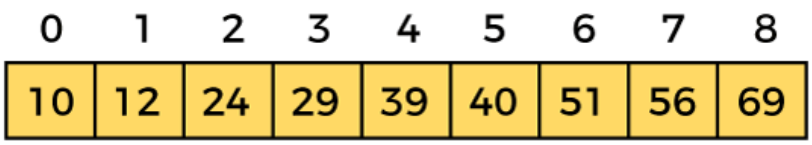
To understand the working of the Binary search algorithm, let's take a sorted array. It will be easy to understand the working of Binary search with an example.

There are two methods to implement the binary search algorithm -

- Iterative method
- Recursive method

The recursive method of binary search follows the divide and conquer approach.

Let the elements of array are -



Let the element to search is, K = 56

We have to use the below formula to calculate the mid of the array -

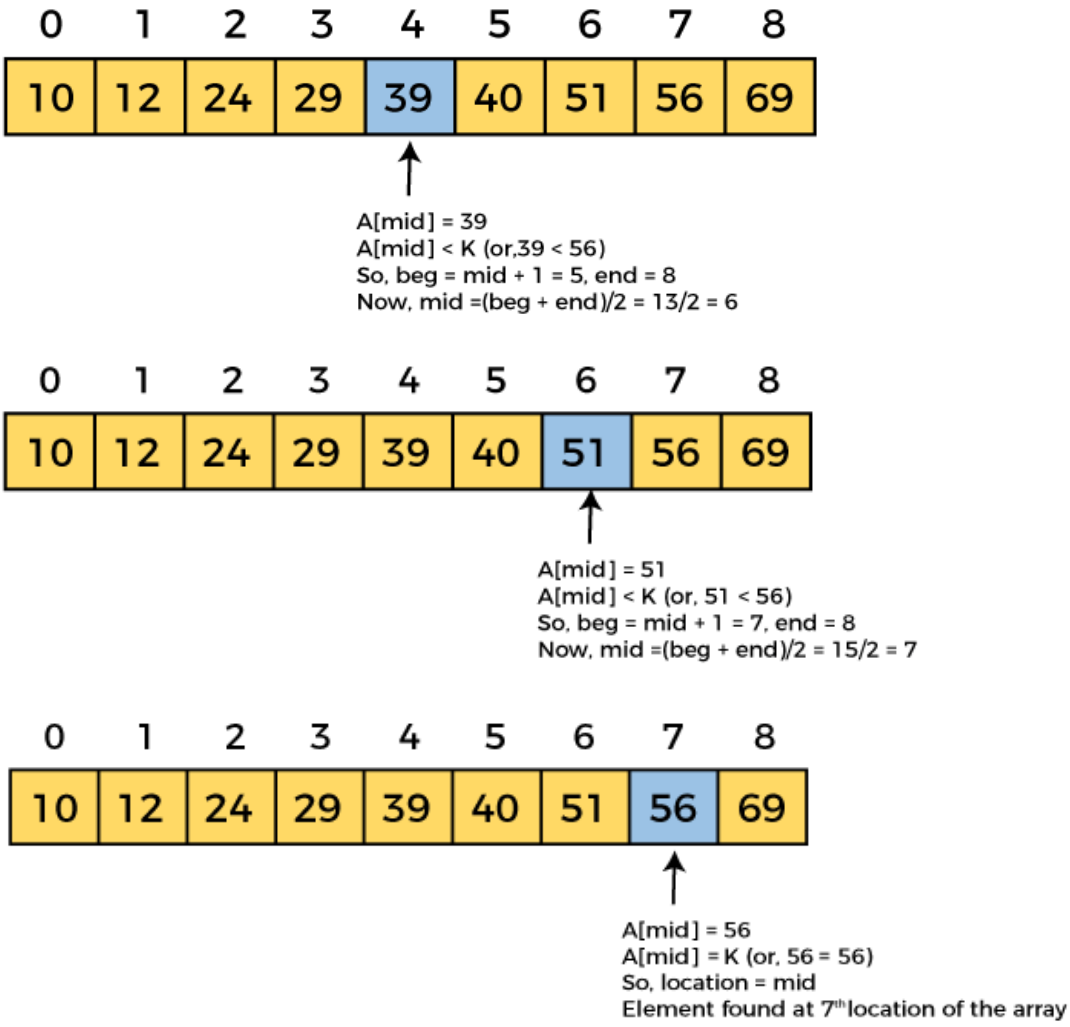
1. **mid = (beg + end)/2**

So, in the given array -

beg = 0

end = 8

mid = (0 + 8)/2 = 4. So, 4 is the mid of the array.



Now, the element to search is found. So algorithm will return the index of the element matched.

Binary Search complexity

Now, let's see the time complexity of Binary search in the best case, average case, and worst case. We will also see the space complexity of Binary search.

1. Time Complexity

Case	Time Complexity
Best Case	**O(1)**
Average Case	**O(logn)**
Worst Case	**O(logn)**

- Best Case Complexity - In Binary search, best case occurs when the element to search is found in first comparison, i.e., when the first middle element itself is the element to be searched. The best-case time complexity of Binary search is O(1).
- Average Case Complexity - The average case time complexity of Binary search is O(logn).
- Worst Case Complexity - In Binary search, the worst case occurs, when we have to keep reducing the search space till it has only one element. The worst-case time complexity of Binary search is O(logn).

2. Space Complexity

Space Complexity	**O(1)**

- The space complexity of binary search is O(1).