UNIT V
SORTING and SEARCHING

Sorting algorithms: Insertion sort - Quick sort - Merge sort - Searching: Linear search –Binary Search

5.1. Sorting algorithms

Sorting is a process in which records are arranged in ascending or descending order. For example, in telephone directory the names of the subscribers and their phone numbers are written in alphabetial order.

Criteria for Evaluating Sorts

Sorting algorithms can be compared based on several factors.

- **Run-time:** The number of operations performed (usually swap and compare).
- **Memory:** The amount of memory needed beyond what is needed to store the data to be sorted.
  - Some algorithms sort "in place" and use no (or a constant amount of) extra memory.
  - Others use a linear amount, and some an exponential amount.
  - Clearly less memory is preferred, although space/time trade-offs are possible.
- **Stability:** An algorithm is stable if it preserves the relative order of equal keys.

Types of Sorting

1. Insertion Sort
2. Shell Sort
3. Heap Sort
4. Merge Sort (*Divide and Conquer*)
5. Quick Sort(*Divide and Conquer*)
6. Indirect Sorting
7. Bucket Sort

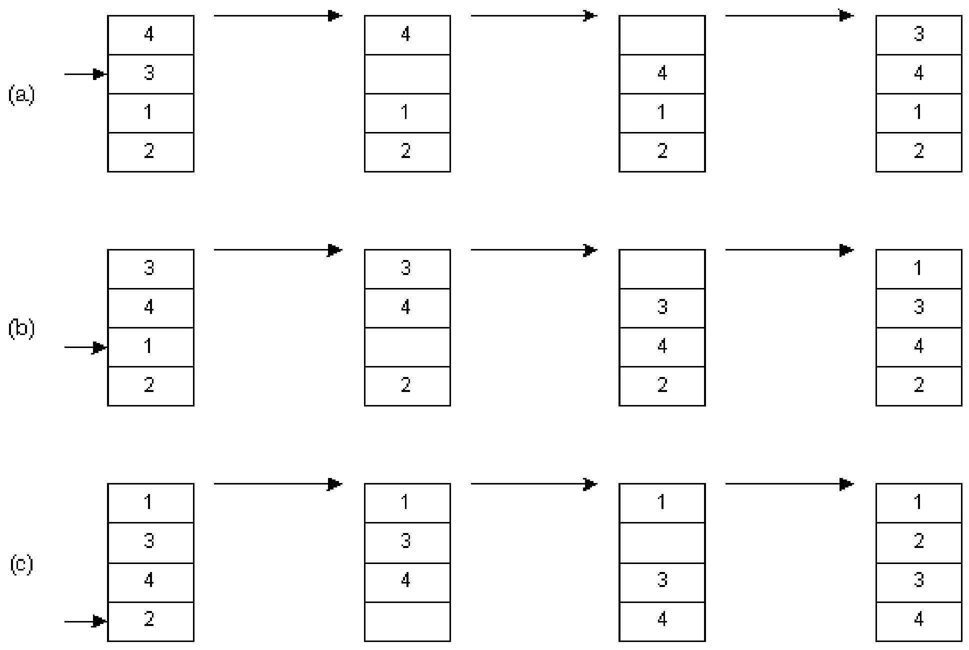
5.1.1. Insertion sort

In insertion sort the element is inserted at appropriate place. For example, consider an array of n elements. In this type, swapping of elements is done without taking any temporary variable. The greater numbers are shifted towards the end of the array and smaller are shifted to beginning. Here, a real life example of playing cards can be cited. We keep the cards in increasing order. The card having least value is placed at the extreme left and the largest one at the other side. In between them the other cards are managed in ascending order.

ROUTINE

```
void insertionSort(int arr[],int length)
{
        int i,j,tmp;
        for(i= 1;i<length;i++)
        {
                j = i;
                while (j > 0 && arr[j - 1] > arr[j])
                {
                        tmp = arr[j];
                        arr[j] = arr[j - 1];
                        arr[j - 1] = tmp;
                        j--;
                }
        }
}
```

Example Starting near the top of the array in the following Figure (a), we extract the 3. Then



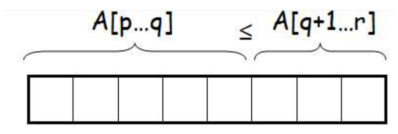
the above elements are shifted down until we find the correct place to insert the 3. This process repeats in Figure (b) with the next number. Finally, in Figure (c), we complete the sort by inserting 2 in the correct place.

Assuming there are n elements in the array, we must index through n - 1 entries. For each entry, we may

need to examine and shift up to n - 1 other entries, resulting in a $O(n^2)$ algorithm. The insertion sort is an *in-place* sort. That is, we sort the array in-place. No extra memory is required. The insertion sort is also a *stable* sort. Stable sorts retain the original ordering of keys when identical keys are present in the input data.

5.1.2. Quick sort

Sort an array A[p…r]



Divide - Partition the array A into 2 subarrays A[p..q] and A[q+1..r], such that each element of A[p..q] is smaller than or equal to each element in A[q+1..r]. Need to find index q to partition the array.

Conquer - Recursively sort A[p..q] and A[q+1..r] using Quicksort

Combine - Trivial: the arrays are sorted in place. No additional work is required to combine themThe entire array is now sorted

Quicksort is a fast sorting algorithm, which is used not only for educational purposes, but widely applied in practice. On the average, it has O(n log n) complexity, making quicksort suitable for sorting big data volumes. The divide-and-conquer strategy is used in quicksort. Below the recursion step is described:

1. **Choose a pivot value.** We take the value of the middle element as pivot value, but it can be any value, which is in range of sorted values, even if it doesn't present in the array.
2. **Partition.** Rearrange elements in such a way, that all elements which are lesser than the pivot go to the left part of the array and all elements greater than the pivot, go to the right part of the array. Values equal to the pivot can stay in any part of the array. Notice, that array may be divided in non-equal parts.
3. **Sort both parts.** Apply quicksort algorithm recursively to the left and the right parts.

There are two indices **i** and **j** and at the very beginning of the partition algorithm **i** points to the first element in the array and **j** points to the last one. Then algorithm moves **i** forward, until an element with value greater or equal to the pivot is found. Index **j** is moved backward, until an element with value lesser or equal to the pivot is found. If **i** ≤ **j** then they are swapped and i steps to the next position (**i + 1**), j steps to the previous one **(j - 1)**. Algorithm stops, when **i** becomes greater than **j**.

After partition, all values before **i-th** element are less or equal than the pivot and all values after **j-th** element are greater or equal to the pivot.

On the average quicksort has O(n log n) complexity, but strong proof of this fact is not trivial and not presented here. Still, you can find the proof in [1]. In worst case, quicksort runs O(n2) time, but on the most "practical" data it works just fine and outperforms other O(n log n) sorting algorithms

Elements are, 50, 30, 10, 90,80,20,40,70

Split the array into two parts.
The left sulist will contain the elements less than Pivot(i.e.) and right sublist contains elements greater than pivot.

	50	30	10	90	80	20	40	70
i/pivot								j

Step 2:

We will increment i. If A[i]<=Pivot, we will continue to increment it untill the element pointed by i is greater than A[Low]..

	50	30	10	90	80	20	40	70
pivot		i						j

Step 3:

increment i as A[i]<=A [Low].

	50	30	10	90	80	20	40	70
pivot			i					j

Step 4:

As A[i]>A [Low], We will stop incrementing i

	50	30	10	90	80	20	40	70
pivot				i				j

Step 5:

As A[i]>Pivot (i.e. 70>50). We will decrement j. We will continue to decrement j untill the element pointed by j is less than A[Low].

50	30	10	90	80	20	40	70
pivot			i			J	

Step 6:

Now we cannot decrement j because 40<50. Hence we will swap A[i] and A[j] i.e 90 and 40.

50	30	10	90	80	20	40	70
pivot			i			j	

Step 7:

As A[i] is less than A[Low] and A[j] is greater than A[Low] we will continue incrementing i and decrementing j, until the false conditions are obtained..

50	30	10	40	80	20	90	70
pivot			i			j	

Step 8:

We will stop incrementing i and stop decrementing j. As i is smaller than j we will swap 80 and 20.

50	30	10	40	80	20	90	70
pivot				i	j		

Step 9:

As A[i]<A[Low] and A[j]>A[Low], we will continue incrementing i and decrementing j

50	30	10	40	20	80	90	70
pivot				i	j		

Step 10:

As A[j]<A[Low] and j has crossed i. That is j<i, we will swap A[Low] and A[j]

50	30	10	40	20	80	90	70
pivot					i, j		

50	30	10	40	20	80	90	70
pivot				j	i		

Step 11:

We will now start sorting left sublist, assuming the first element of left sublist as pivot element thus now new pivot =20

50	30	10	40	50	80	90	70
pivot				j	i		

Step 12:

Now we will set i and j pointer and then we will start comparing A[i] with A[Low] or A[pivot]. Similarly comparison with A[j] and A[pivot].

20	30	10	40	50	80	90	70
pivot	i		j	Occupied Its position			

Step 13:

As A[i] > Pivot, hence stop incrementing i. Now as A[j]> A[Pivot], hence decrement j.

20	30	10	40	50	80	90	70
pivot	i		j				

Step 14:

Now j cannot be decremented because 10<20. Hence we will swap A[i] and A[j]

20	30	10	40	50	80	90	70
pivot	i	j					

Step 15:

As A[i]<A[Low], increment i.

20	10	30	40	50	80	90	70
pivot	i	j					

Step 16:

Now as A[i]>a[Low], or A[j]> A[Pivot] decrement j.

20	10	30	40	50	80	90	70
pivot		i,j					

Step 17:

As A[j] < A[Low] we cannaot decrement j now. We will now swap a[Low] and A[j] as j has crossed i and i>j

20	10	30	40	50	80	90	70
pivot	j	i					

Step 18:

As there is only one element in left sublist hence we will sort right sublist.

10	20	30	40	50	80	90	70
Left sublist	pivot	Right Sublist					

Step 19:

As left sublist is sorted completely we will sort right subist, assuming first element of right sublist as pivot

10	20	30	40	50	80	90	70

Step 20:

As A[i]> A[pivot], hence we will stop incrementing i. similarly A[j] < A[Pivot]. Hence we stop decrementing j. Swap A[i] and A[j].

10	20	30	40	50	80	90	70
					Pivot	i	j

Step 21:

As A[i] < A[Pivot], increment i

10	20	30	40	50	80	70	90
					Pivot	i	j Step

22:

As A[i]>A[Pivot], decrement j.

10	20	30	40	50	80	70	90
					Pivot		i,j

Step 23:

Now swap A[Pivot] and A[j]

10	20	30	40	50	80	70	90
					Pivot	j	i

Step 24:

The left now contains 70 and right sublist contains only 90. We cannot further subdivide the list

10	20	30	40	50	70	80	90
						Pivot	

Hence list is

```
        10      20      30      40      50      70      80      90
```

This is the sorted list

ROUTINE

```
void quickSort(int arr[], int left, int right)
{
      int i = left,
      j = right;
      int tmp;
      int pivot = arr[(left + right) / 2]; /* partition */
      while (i <= j)
      {
            while (arr[i] < pivot) i++; while (arr[j] > pivot)
                  j--;
            if (i <= j)
            {
                  tmp = arr[i];
                  arr[i] = arr[j];
                  arr[j] = tmp;
                  i++;
                  j--;
            }
      }; /* recursion */
      if (left < j) quickSort(arr, left, j);
      if (i < right)
            quickSort(arr, i, right);
}
```

5.1.3. Merge sort

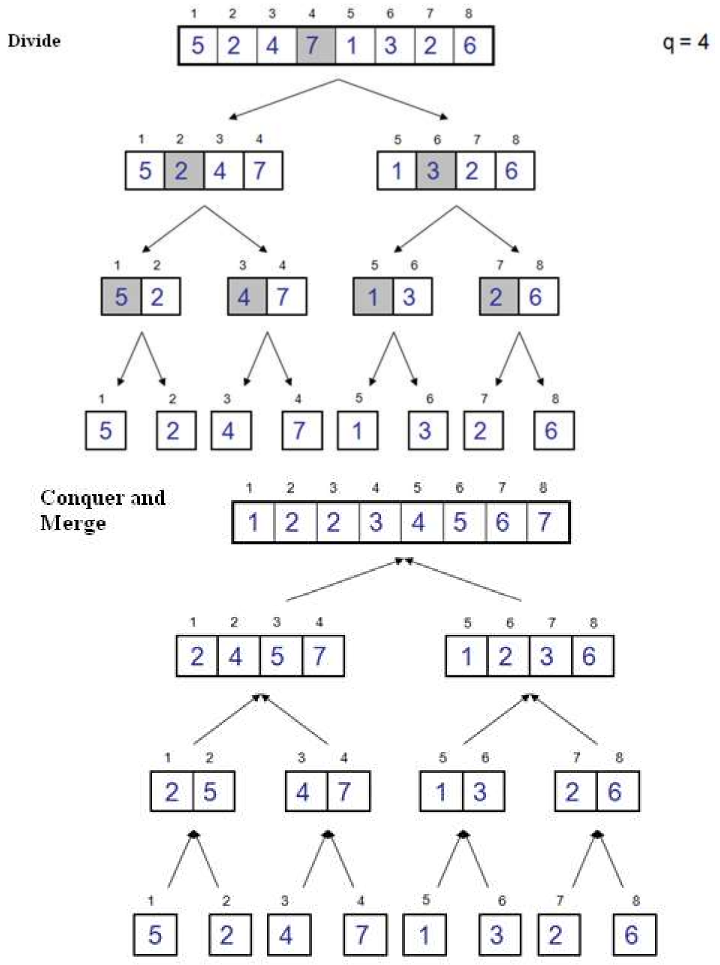
To sort an array A[p . . r]:

Divide - Divide the n-element sequence to be sorted into two subsequences of n/2 elements each

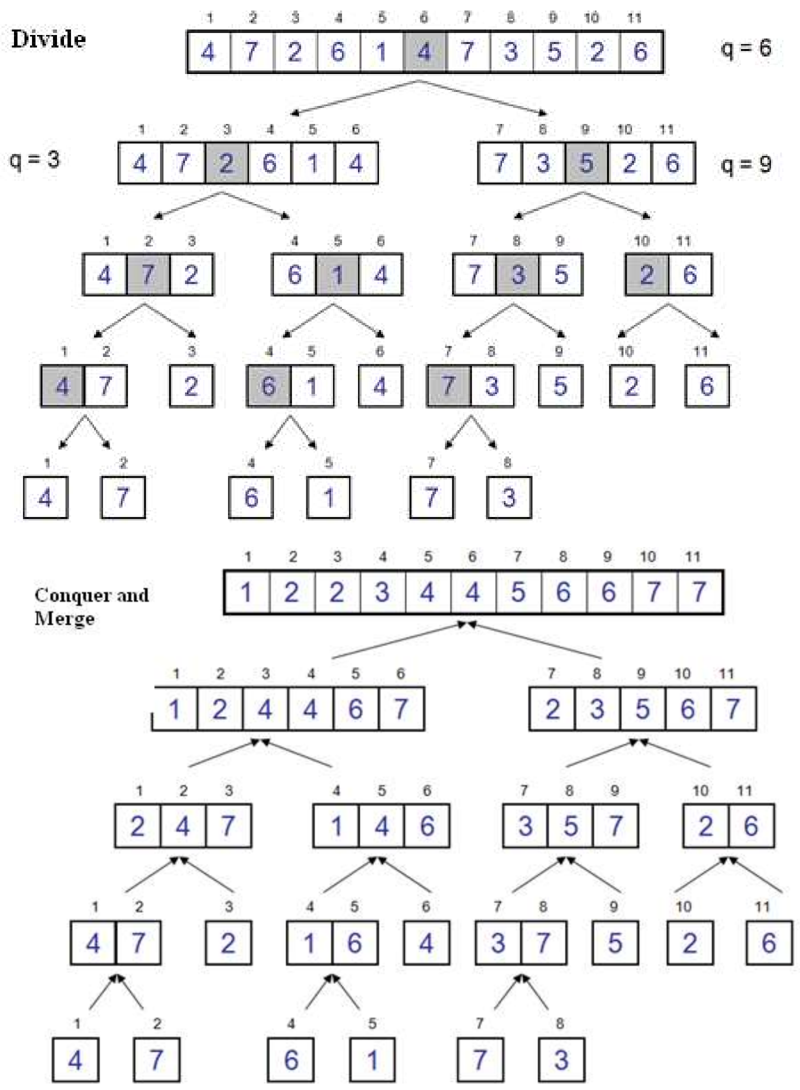
Conquer - Sort the subsequences recursively using merge sort. When the size of the sequences is 1 there is nothing more to do.

Combine - Merge the two sorted subsequences

Example 1: n power of 2



Example 2: n not a power of 2



The following procedure *mergesort* sorts a sequence *a* from index *lo* to index *hi*.

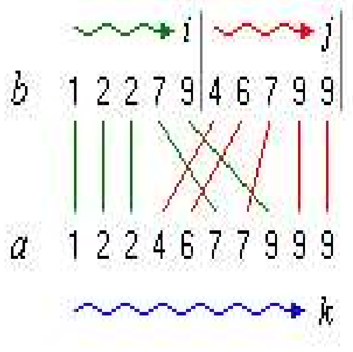
```
void mergesort(int lo,int hi)
{
   if (lo<hi)
   {
         int m=(lo+hi)/2;
         mergesort(lo, m);
         mergesort(m+1, hi); merge(lo, m, hi);
```

```
        }
    }
```

First, index *m* in the middle between *lo* and *hi* is determined. Then the first part of the sequence (from *lo* to *m*) and the second part (from *m*+1 to *hi*) are sorted by recursive calls of *mergesort*. Then the two sorted halves are merged by procedure *merge*. Recursion ends when *lo* = *hi*, i.e. when a subsequence consists of only one element.

The main work of the Mergesort algorithm is performed by function *merge*. There are different possibilities to implement this function.

First, the sequence to be sorted is decomposed into two halves (*Divide*). Each half is sorted independently (*Conquer*). Then the two sorted halves are merged to a sorted sequence (*Combine*).



Function *merge* is usually implemented in the following way: The two halves are first copied into an auxiliary array *b*. Then the two halves are scanned by pointers *i* and *j* and the respective next-greatest element at each time is copied back to array *a* (Figure 2).

At the end a situation occurs where one index has reached the end of its half, while the other has not. Then, in principle, the rest of the elements of the corresponding half have to be copied back. Actually, this is not necessary for the second half, since (copies of) the remaining elements are already at their proper places.

```
    void merge(int lo, int m, int hi)
    {
        int i, j, k;
        i=0;
        j=lo; // copy first half of array a to auxiliary array b
        while (j<=m)
                b[i++]=a[j++];
        i=0;
        k=lo;
        while (k<j && j<=hi)
        if (b[i]<=a[j])
                a[k++]=b[i++];
        else
                a[k++]=a[j++]; // copy back remaining elements of
        first half (if any)
        while (k<j)
                a[k++]=b[i++];
    }
```

The sorting algorithm Mergesort produces a sorted sequence by sorting its two halves and merging them. With a time complexity of $O(n \log(n))$ Mergesort is optimal. Similar to Quicksort, the Mergesort algorithm is based on a divide and conquers strategy. Total time for Merge : $\Theta(n)$

5.2. Searching

Searching is a technique in which the record is searched based on some key field. There are two methods of searching

1) Linear search

2) Binary search

5.2.1. Linear search

The data is arranged in the list. At every iteration the record will be compared with the help of the key. In this case sometimes all the elements get compared with the key value.

The time complexity of this algorithm is O(n). The time complexity will increase linearly with the value of n. for higher value of n the linear search is not the satisfactory solution.

Array

	Roll no	**Name**	**Marks**
0	13	Parth	96
1	2	Anand	40
2	13	Lalitha	81
3	1	Madhav	50
4	12	Arun	78
5	3	Jaya	94

Fig: Represents students database for sequential search

From the above fig the array is maintained to store the students record. the record is not at all. If we want to search the students record whose roll number is 12 then the key roll number we will see the every record whether it is of roll number==12. we can obtain we can obtain such a record at array [4] location. Let us now implement the sequential search using C++ program

```
class search
{
      private:
            int a[10],n,key;
      public:
            void get_data();
            void Seq_Search(int);
};
void Search::Seq_Search(int key)
{
      int flag=0,mark;
      for(int i=0;i<n;i++)
      {
            if(a[i]==key)
            {
                  flag=1;
                  mark=i;
                  break;
            }
       }
}
```

Output

```
How many Elements are there in an array? 7
Enter the elements 10 20 30 40 50 60 70
Enter the element which is to be searched 50
The element is present at location
```

5.2.2. Binary Search

Definition

binary search is used to quickly find a value in a sorted sequence (consider a sequence an ordinary array for now). We'll call the sought value the *target* value for clarity. Binary search maintains a contiguous subsequence of the starting sequence where the target value is surely located. This is called the *search space*. The search space is initially the entire sequence. At each step, the algorithm compares the median value in the search space to the target value. Based on the comparison and because the sequence is sorted, it can then eliminate half of the search space. By doing this repeatedly, it will eventually be left with a search space consisting of a single element, the target value.

For example, consider the following sequence of integers sorted in ascending order and say we are looking for the number 55:

0	5	13	19	22	41	55	68	72	81	98

We are interested in the location of the target value in the sequence so we will represent the search space as indices into the sequence. Initially, the search space contains indices 1 through 11. Since the search space is really an interval, it suffices to store just two numbers, the low and high indices. As described above, we now choose the median value, which is the value at index 6 (the midpoint between 1 and 11): this value is 41 and it is smaller than the target value. From this we conclude not only that the element at index 6 is not the target value, but also that no element at indices between 1 and 5 can be the target value, because all elements at these indices are smaller than 41, which is smaller than the target value. This brings the search space down to indices 7 through 11:

55	68	72	81	98

Proceeding in a similar fashion, we chop off the second half of the search space and are left with:

55	68

Depending on how we choose the median of an even number of elements we will either find 55 in the next step or chop off 68 to get a search space of only one element. Either way, we conclude that the index where the target value is located is 7.

If the target value was not present in the sequence, binary search would empty the search space entirely. This condition is easy to check and handle. Here is some code to go with the description:

```
binary_search(A, target):
```

```
lo = 1, hi = size(A)
while lo <= hi:
  mid = lo + (hi-lo)/2
  if A[mid] == target:
     return mid
  else if A[mid] < target:
     lo = mid+1
  else:
     hi = mid-1
```

Complexity of Sorting

Sorting algorithms

Algorithm	**Data Structure**	**Time Complexity:Best**	**Time Complexity:Average**	**Time Complexity: Worst**	**Space Complexity: Worst**
Quick Sort	Array	O(nlog(n))	O(n log(n))	$O(n^2)$	O(log(n))
Merge sort	Array	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Heap sort	Array	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Smooth sort	Array	O(n)	O(n log(n))	O(n log(n))	O(1)
Bubble sort	Array	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Insertion sort	Array	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Selection sort	Array	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)