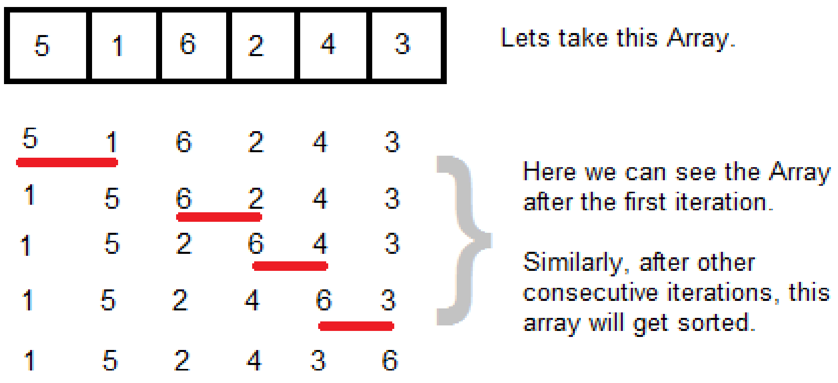
DATA STRUCTURES

(this is an additional material and not exhaustive study material)

Sorting Algorithms: Basic concepts - Bubble Sort - Insertion Sort - Selection Sort - Quick Sort – Shell sort - Heap Sort - Merge Sort - External Sorting.
Searching: Linear Search, Binary Search.

Bubble sort

Bubble Sort is an algorithm which is used to sort **N** elements that are given in a memory for eg: an Array with**N** number of elements. Bubble Sort compares all the element one by one and sort them based on their values. It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order. Complexity of Bubble Sort is **O(n^2)**.


5 1 6 2 4 3
Lets take this Array.
5 1 6 2 4 3
1 5 6 2 4 3
1 5 2 6 4 3
1 5 2 4 6 3
1 5 2 4 3 6
Here we can see the Array after the first iteration.
Similarly, after other consecutive iterations, this array will get sorted.

```
void main()
{    int a[50], n, i, j, temp = 0;
     printf("Enter how many numbers to sort:\n");
     scanf("%d", &n);
     printf("Enter the %d numbers:\n", n);
     for (i = 0; i < n; i++)
            scanf("%d", &a[i]);
                                          // Bubble sort starts
    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;
                }
            }         // inner for ends
        }           // outer for ends
 printf("\n\n\n\t\tThe sorted numbers using Buble sort is:\n");
 for (i = 0; i < n; i++)
      printf("\n\t\t%d", a[i]);
}               // program ends
```

Insertion Sort

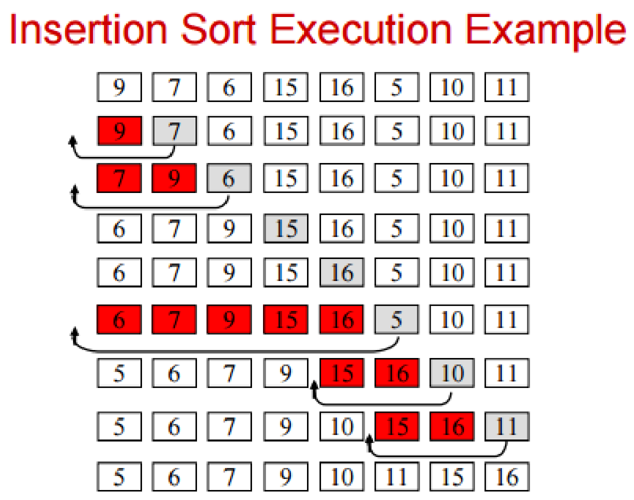
Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

The resulting array after *k* iterations has the property where the first *k* + 1 entries are sorted.

Worst Case Time Complexity : O(n^2)

Example :



Program for Insertion sort

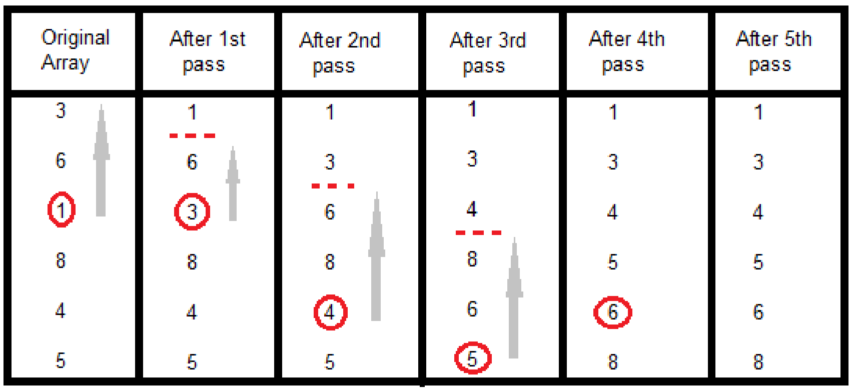
```
 //  read the array
 // Insertion sort begins
                                         // Insertion Sort Begins
     for(i=1; i< n; i++)
      {
            Temp = A[i];
            j = i-1;
            while(Temp < A[j] && j>=0)
            {      A[j+1] = A[j];
                   j = j-1;
            }
            A[j+1] = Temp;
      }
 // Print the sorted array
```

Selection Sort

The **selection sort** improves on the bubble sort by making only one exchange for every pass through the list. In order to do this, a selection sort looks for the largest value as it makes a pass and, after completing the pass, places it in the proper location. As with a bubble sort, after the first pass, the largest item is in the correct place. After the second pass, the next largest is in place. This process continues and requires n−1 passes to sort *n* items, since the final item must be in place after the (n−1) st pass.

Worst Case Time Complexity : O(n^2)

Example :



Original Array	After 1st pass	After 2nd pass	After 3rd pass	After 4th pass	After 5th pass
3	1	1	1	1	1
6	6	3	3	3	3
1	3	6	4	4	4
8	8	8	8	5	5
4	4	4	6	6	6
5	5	5	5	8	8

Program for Selection Sort :

```
//  read the array
// Selection sort begins
for (i = 0; i < size - 1; i++)
{        min = i;
         for (j = i + 1; j < size; j++)
               {    if (a[j] < a[min])
                         min = j;
               }
          temp=a[i];
          a[i]=a[min];
          a[min]=temp;
}
// Print the sorted array
```

Quicksort

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.

QUICK SORT Algorithm

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 left most element or middle element or right most element as pivot value.
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.

```
void quicksort(int x[10],int first, int last)
{   int pivot,j,temp,i;

    if(first<last)
  {    pivot=first;
        i=first+1;
        j=last;

        while(i<j)
     {
            while(x[i]<=x[pivot]&&i<last)     // moving i right
               i++;
            while(x[j]>x[pivot])              // moving j left
               j--;
            if(i<j){                          // swapping i and j
                 temp=x[i];
                 x[i]=x[j];
                x[j]=temp;   }
        }                                     // repeat till i and j does not overlap

        temp=x[pivot];                // finally swapping with the pivot bcoz first < last
        x[pivot]=x[j];
        x[j]=temp;
        quicksort(x,first,j-1);
        quicksort(x,j+1,last);
   }     // if first < last
}  //   quick sort ends
```

Shell Sort

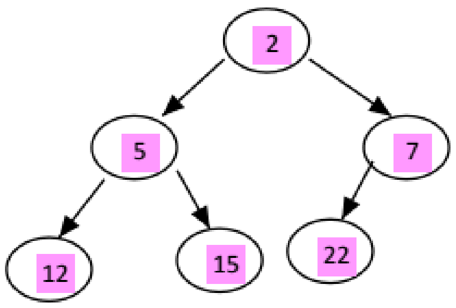
- Shellsort works by comparing elements that are distant rather than adjacent elements in an array.
- Shellsort makes multiple passes through a list and sorts a number of equally sized sets using the insertion sort.
- The distance between comparisons decreases as the sorting algorithm runs until the last phase in which adjacent elements are compared

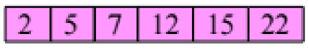
```
/*c program for sorting array using shell sorting method*/
#include<stdio.h>
#include<conio.h>
void main()
{
 int arr[30];
 int i,j,k,tmp,num;
 printf("Enter total no. of elements : ");
 scanf("%d", &num);
 for(k=0; k<num; k++)
 {
   printf("\nEnter %d number : ",k+1);
   scanf("%d",&arr[k]);
 }
 for(i=num/2; i>0; i=i/2)
 {
   for(j=i; j<num; j++)
   {
     for(k=j-i; k>=0; k=k-i)
     {
       if(arr[k+i]>=arr[k])
           break;
       else
       {
           tmp=arr[k];
           arr[k]=arr[k+i];
           arr[k+i]=tmp;
       }
     }
   }
 }
 printf("\t**** Shell Sorting ****\n");
 for(k=0; k<num; k++)
     printf("%d\t",arr[k]);
 getch();
}
```

Heap Sort

Heapsort uses the property of Heaps to sort an array. The Heap data structure is an array object that can be viewed as a complete and balanced binary tree. Min (Max)-Heap has a property that for every node other than the root, the value of the node is at least (at most) the value of its parent. Thus, the smallest (largest) element in a heap is stored at the root, and the subtrees rooted at a node contain larger (smaller) values than does the node itself.

Worst case performance O(nlogn).





HEAP SORT Algorithm:

It starts with building a heap by inserting the elements entered by the user, in its place.

1. Increase the size of the heap as a value is inserted.
2. Insert the entered value in the last place.
3. Compare the inserted value with its parent, until it satisfies the heap property and then place it at its right position.

Now once the heap is built remove the elements from top to bottom, while maintaining the heap property to obtain the sorted list of entered values.

1. heap[1] is the minimum element. So we remove heap[1]. Size of the heap is decreased.
2. Now heap[1] has to be filled. We put the last element in its place and see if it fits. If it does not fit, take minimum element among both its children and replaces parent with it.
3. Again see if the last element fits in that place.

Heap Sort Algorithm

//C Program to sort an array based on heap sort algorithm(MAX heap)

```
#include <stdio.h>
void main()
{
int heap[10], no, i, j, c, root, temp;
printf("\n Enter no of elements :");
scanf("%d", &no);
printf("\n Enter the nos : ");
for (i = 0; i < no; i++)
              scanf("%d", &heap[i]);
for (i = 1; i < no; i++)
{
   c = i;
   do
   {  // to compute the parent/root. Both for even and odd values same root value
      root = (c - 1) / 2;
      if (heap[root] < heap[c])
        // to create MAX heap array-swap only when needed – percolation up
        { temp = heap[root]; heap[root] = heap[c]; heap[c] = temp; }  //swapping
     c = root;
   } while (c != 0);
 }
printf("Heap array : ");
for (i = 0; i < no; i++)
    printf("%d\t ", heap[i]);

// start heap sort - chop & percolate down
for (j = no - 1; j >= 0; j--)
{   temp = heap[0];
   heap[0] = heap[j];    // swap max element with rightmost leaf element  - chop
   heap[j] = temp;
   root = 0;
   do
   {
     c = 2 * root + 1;    // left node of root element, if left child at c then right child at c+1
     if ((heap[c] < heap[c + 1]) && c < j-1)
         c++;               // to check whether the left or right child is greater for swapping
    if (heap[root]<heap[c] && c<j)   // again rearrange to max heap array – percolate down
        {   temp = heap[root]; heap[root] = heap[c]; heap[c] = temp;   }  //swapping
    root = c;
  } while (c < j);
}          // for loop – it will do chop & percolate till all the n-1 nodes are sorted

printf("\n The sorted array is : ");
for (i = 0; i < no; i++)
  printf("\t %d", heap[i]);
}        // end of program
```

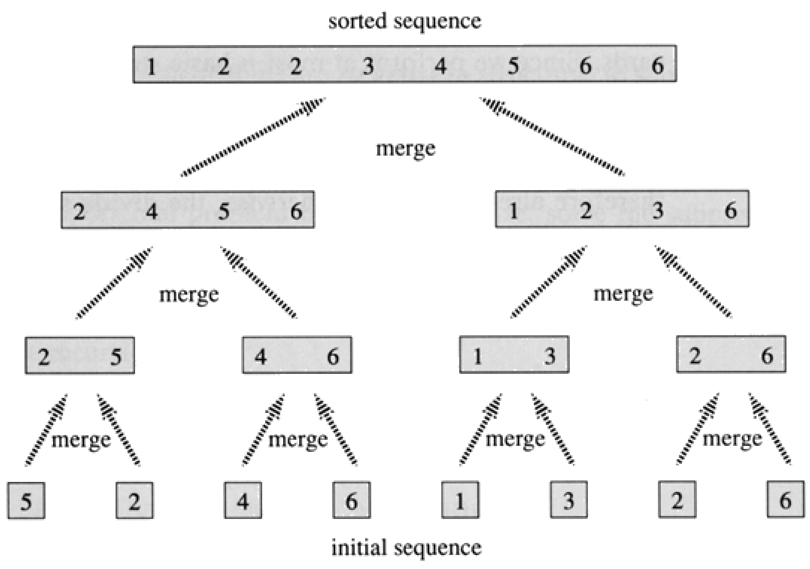
External Sorting

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). External sorting typically uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub-files are combined into a single larger file.

One example of external sorting is the external merge sort algorithm, which sorts chunks that each fit in RAM, then merges the sorted chunks together. We first divide the file into **runs** such that the size of a run is small enough to fit into main memory. Then sort each run in main memory using merge sort sorting algorithm. Finally merge the resulting runs together into successively bigger runs, until the file is sorted.

Merge Sort

Merge sort runs in O (n log n) running time. It is very efficient sorting algorithm with near optimal number of comparison. An array of n elements is split around its center producing two smaller arrays. After these two arrays are sorted independently, they can be merged to produce the final sorted array. The process of splitting and merging can be carried recursively till there is only one element in the array. An array with 1 element is always sorted.



Algorithm :

Inputs:
input_file : Name of input file. input.txt
output_file : Name of output file, output.txt
run_size : Size of a run (can fit in RAM)
num_ways : Number of runs to be merged

1) Read input_file such that at most 'run_size' elements
   are read at a time. Do following for the every run read
   in an array.

```
     a) Sort the run using MergeSort
     b) Store the sorted run in a temporary file, say 'i'
        for i'th run.
2) Merge the sorted files
```

Searching :

Searching is the process of finding element in a given list. In this process we check item is available in given list or not.

Searching Techniques

There are basically three types of searching techniques:

- Linear or sequential search
- Binary search

Linear/ Sequential Search

Linear/Sequential searching is a searching technique to find an item from a list until the particular item not found or list not reached at end. We start the searching from 0th index to N-1 index in a sequential manner, if particular item found, returns the position of that item otherwise return failure status or -1.

Implementation of Linear/ Sequential Search

```
int main()
{
  int array[100], search, c, n;
  printf("Enter number of elements in array\n");
  scanf("%d", &n);
   printf("Enter %d integer(s)\n", n);

  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);

  printf("Enter a number to search\n");
  scanf("%d", &x);
  found=0;
  for (c = 0; c < n; c++)
  {    if (array[c] == x)    /* If required element is found */
       {      printf("%d is present at location %d.\n", x, c);
              Found=1; break;   }
  }
  if (found== 0)
    printf("%d isn't present in the array.\n", search);
 }
```

Binary Search

It is special type of search work on sorted list only. During each stage of our procedure, our search for **ITEM** is reduced to a restricted segment of elements in **LIST** array. The segment starts from index LOW and spans to **HIGH**.

The **ITEM** to be searched is compared with the middle element **LIST[MID]** of segment, where **MID** obtained as: **MID = (LOW + HIGH)/2**

We assume that the **LIST** is sorted in ascending order. There may be three results of this comparison:

1. If **ITEM = LIST[MID]**, the search is successful. The location of the ITEM is **LOC := MID**.
2. If **ITEM < LIST[MID]**, the ITEM can appear only in the first half of the list. So, the segment is restricted by modifying **HIGH = MID – 1**.
3. If **ITEM > LIST[MID]**, the ITEM can appear only in the second half of the list. So, the segment is restricted by modifying **LOW = MID + 1**.

Initially, **LOW** is the start index of array, and HIGH is the last index of array. The comparison goes on. With each comparison, low and high approaches near to each other. The loop will continue till **LOW < HIGH**.

Complexity of Binary search

The complexity measured by the number f(n) of comparisons to locate ITEM in LIST where LIST contains n elements. Each comparison reduces the segment size in half. Approximately, the time complexity is equal to $\log_2 n$. It is much less than the time complexity of linear search.

Implementation of Binary Search

```
/*To search item from sorted list*/
void  BinarySearch( int ele[], int item )
{  int POS  = -1   ;
   int LOW  =  0   ;
   int HIGH =  SIZE;
   int MID  =  0   ;

   while( LOW <= HIGH )
   {   MID = (LOW + HIGH)/2;
       if( ele[MID] == item )
       {   POS = MID ;
           Printf("Element found at position %d", POS);    // found exactly in the middle position
          break;
       }
       else if( item > ele[MID] )
            LOW = MID + 1;  // search on the left side
       else
            HIGH = MID - 1;  // search on the right side
   }
  Printf ("Element not found");    //element not found
}
```

Difference between Linear Search and Binary Search

Linear Search	Binary Search
Sorted list is not required.	Sorted list is required.
It can be used in linked list implementation.	It cannot be used in liked list implementation.

It is suitable for a list changing very frequently.	It is only suitable for static lists, because any change requires resorting of the list.
The average number of comparison is very high.	The average number of comparison is relatively slow.

COMPLEXITY COMPARISON:

S.No	Name	Complexity
1	Bubble sort	o(n)
2	Insertion sort	o(n)
3	Selection sort	o(n^2)
4	Quick sort	o(nlog(n))
5	Shell sort	o(nlog(n))
6	Heap sort	o(nlog(n))
7	Merge sort	o(nlog(n))
8	Linear search	o(n)
9	Binary search	o(log(n)).