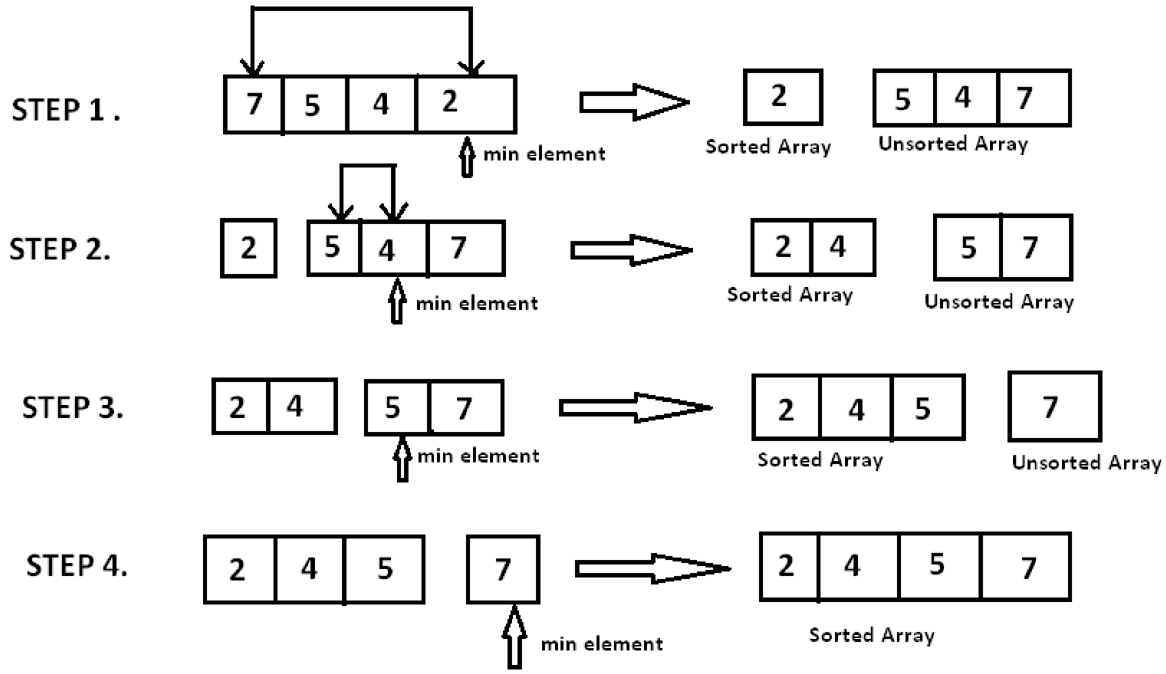
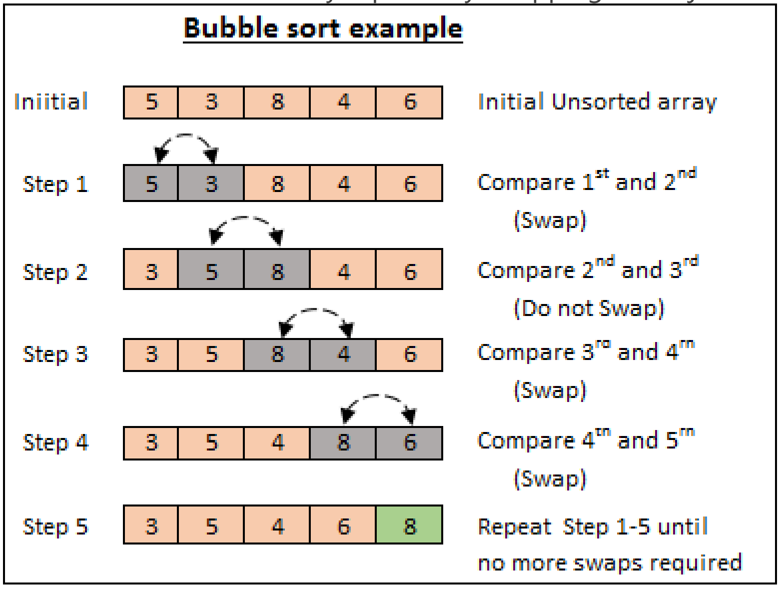
Sorting is the rearrangement of elements in into a specific order.
Sorting algorithms helps in making the solution easier and efficient.
Some basic yet important sort algorithms, that one must know are as follows:

1. Selection Sort: It sorts an array by repeatedly finding the minimum element (considering decresing order) from unsorted part and putting it at the beginning.




```
void selection_sort(vector <int> arr, int n){

        int min_index=i;

        for(int j=i;j<n;j++){

                if(arr[j]<arr[min_index])

                        min_index=j;

                }

                swap(arr[i],arr[min_index]);

        }
```

2. Bubble Sort: It works by repeatedly swapping the adjacent wrong elements.

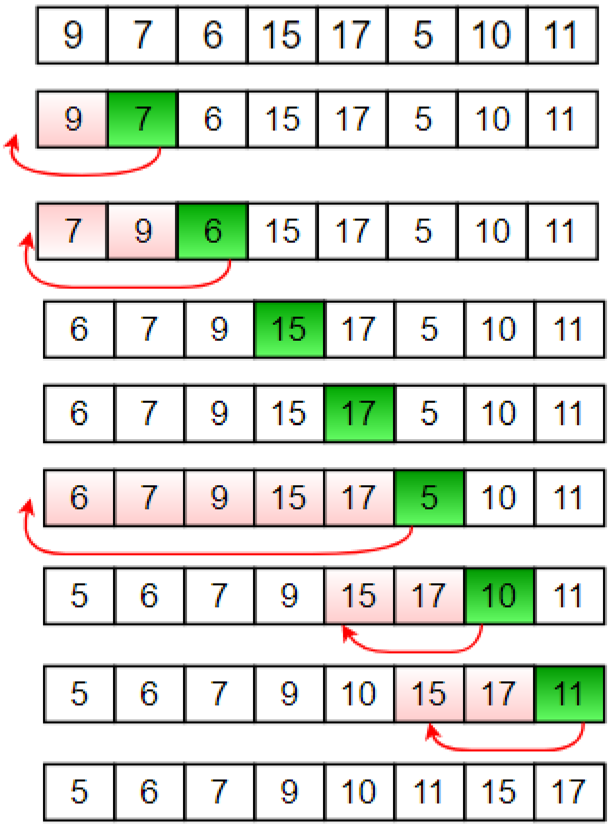



```
void bubble_sort(vector <int> arr, int n){

        for(int itr=1;itr<n;itr++){

                for(int j=0;j<(n-itr-);j++){

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

                                swap(arr[j],arr[j+1]);

                }

        }

}
```

Recursive Bubble Sort:

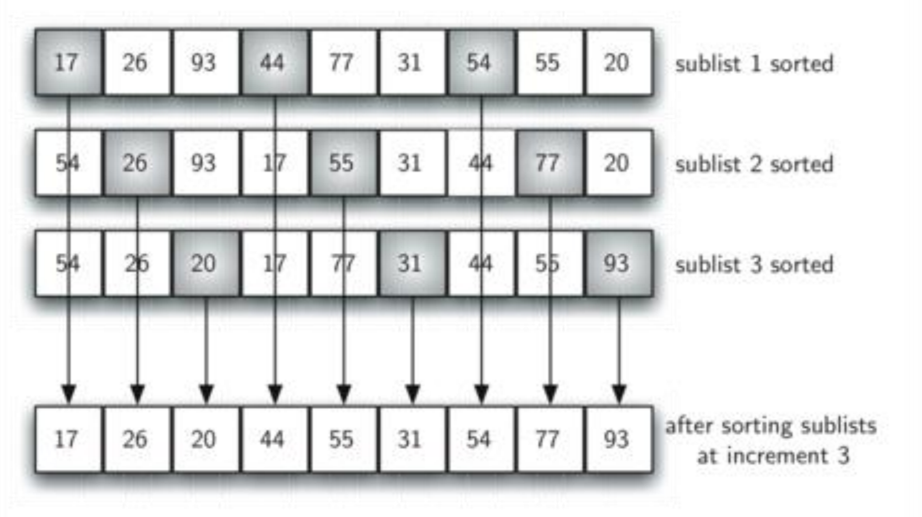
```
void bubble_sort(vector <int> arr, int n){

    if (n == 1)        // Base case

        return;



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

        if (arr[i] > arr[i+1])

            swap(arr[i], arr[i+1]);



    bubble_sort(arr, n-1);

}
```

3. Insertion Sort: Values from unsorted part is picked and placed at the sorted position.



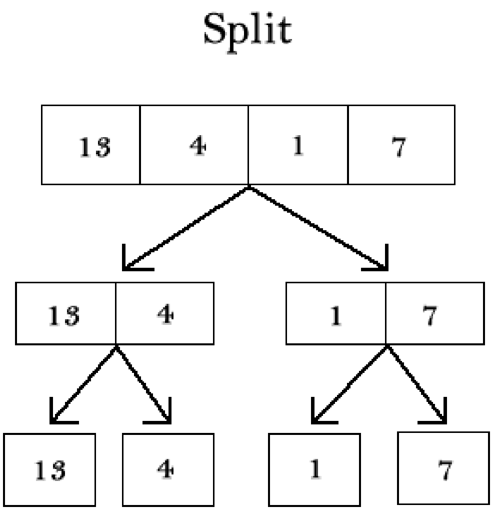

```
void insertion_sort(vector <int> arr, int n){

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

                int e=arr[i];

                int j=i-1;

                while(j>=0 && arr[j]<arr[j]>e){

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

                        j=j-1;

                }

                a[j+1]=e;

        }
```

4. Shell Sort: It is the variation of Insertion sort, as here the elements are moved far ahead.




```
void shell_sort(vector <int> arr, int n){

        for (int gap = n/2; gap > 0; gap /= 2) {

        for (int i = gap; i < n; i += 1)  {

            int temp = arr[i];

            int j;

            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)

                arr[j] = arr[j - gap];

            arr[j] = temp;

        }

    }
```

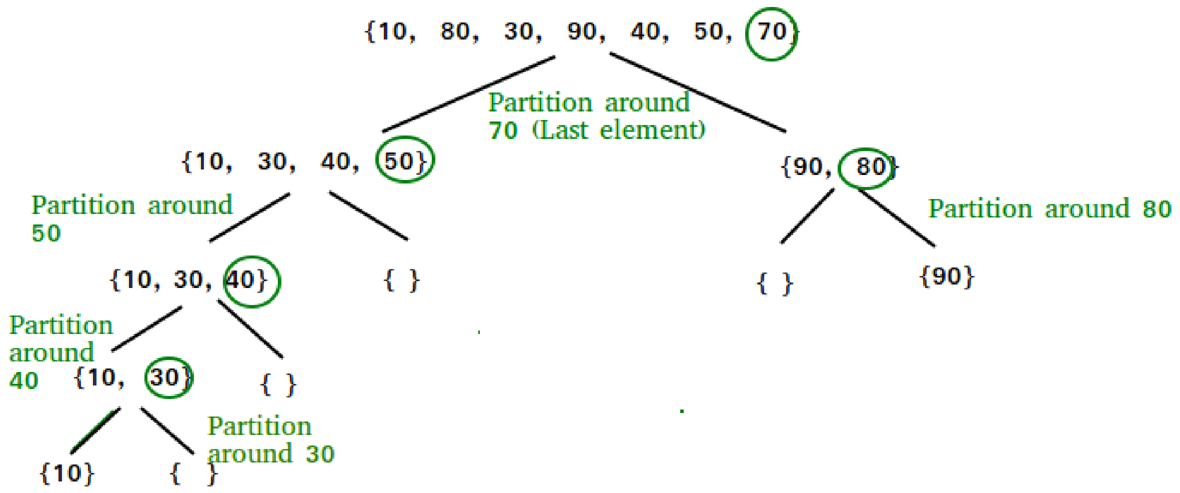
5. Merge Sort: It is based on **Divide and Conquer algorithm**. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.




```
void merge_sort(int *arr, int start, int end){

        if(start==end)

                return;

        int mid=(start+end)/2;

        merge_sort(arr,start,mid);

        merge_sort(arr,mid+1,end);

        merge(arr,start,end);

}


void merge(int *arr, int start, int end){

        int mid=(start+end)/2;

        int i=start;

        int j=mid+1;

        int k=start;

        int temp[100];

        while(i<=mid && j<end){

                if(arr[i] < arr[j])

                        temp[k++]=arr[i++];
```

```
            else
                    temp[k++]=arr[j++];
            }
    while(i<=mid){
                    temp[k++]=arr[i++];
            }
    while(j<=end){
                    temp[k++]=arr[j++];
            }
    for(i=start;i<=end;i++)
            arr[i]=temp[i];
```

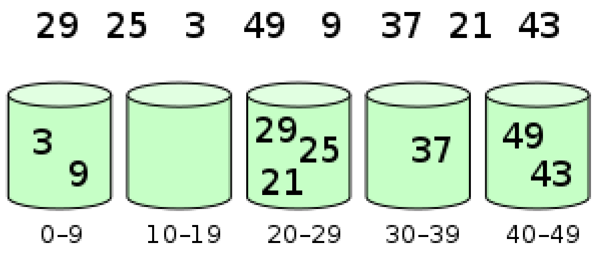
6. Quick Sort: It is alse based on **Divide and Conquer algorithm**. It picks an element as pivot and partitions the given array around the picked pivot.




```
void quick_sort(int *arr, int start, int end){
        if(start>=end)
                return;
        int p=partition(arr,start,end);
        quick_sort(arr,start,p-1);
        quick_sort(arr,p+1,end);
}
```

```
void partition(int *arr,int start,int end){

        int i=start-1;

        int j=start;

        int pivot=arr[end];

        for(;j<=end-1;){

                if(arr[j]<=pivot){

                        i++;

                        swap(arr[i],arr[j]);

                }

                j=j+1;

        }

        swap(arr[i+1],arr[end]);

        return i+1;

}
```

7. Bucket Sort: Bucket sort, or bin sort, is a sorting algorithm that works by distributing the elements of an array into a number of buckets.




```
void bucket_sort(int arr[], int n){

    vector<int> a[101];

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

        int a[i] = n * arr[i];

        a[a[i]].push_back(arr[i]);

    }


        for(int i=100;i>=0;i--){

                for(auto it:a)

                        cout<<it<<" ";
```

```
        }
}
```

8. Wave Sort: Given an unsorted array of integers, sort the array into a wave like array. An array 'arr[0..n-1]' is sorted in wave form if arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= .....

```
void wave_sort(int arr[], int n){
        for(int i=0;i<n;i+=2){
                if(i!=0 && arr[i]<arr[i-1])
                        swap(arr[i],arr[i-1]);
                if(i!=(n-1) && arr[i]<arr[i+1])
                        swap(arr[i],arr[i+1]);
        }
          for (int i = 0; i < n; i++) {
                  cout<<arr[i]<<" ";
         }
}
```

9. Single Pass (DNF): Given an array arr[] consisting 0s, 1s and 2s. Sorting is required for this special case.

```
void dnf_sort(int arr[], int n){
        int low=0;
        int high=n-1;
        int mid=0;
        while(mid<=high){
                if(arr[mid]==0){
                        swap(arr[mid],arr[low]);
                        low++;
                        mid++;
                }
                if(arr[mid]==1)
                        mid++;
                if(arr[mid]==2){
```

```
                        swap(arr[mid],arr[high]);

                        high--;

                }

        }

}
```

10. Counting Sort: It is a sorting technique based on keys between a specific range.

Input Data

0	4	2	2	0	0	1	1	0	1	0	2	4	2

Count Array

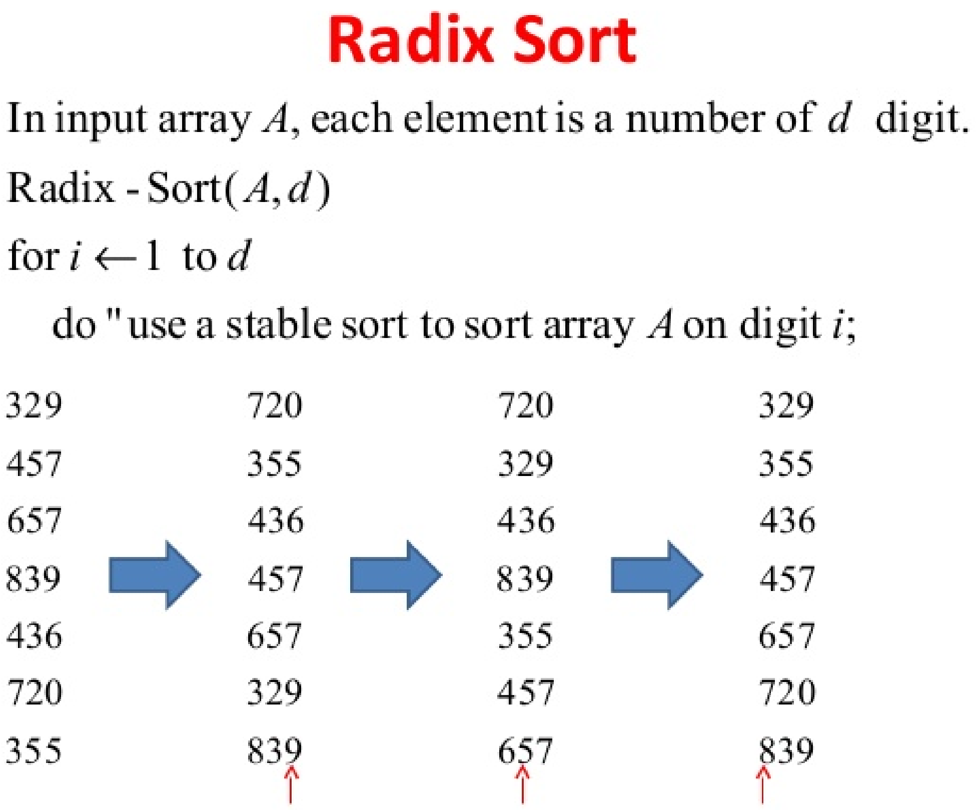
0	1	2	3	4
5	3	4	0	2

Sorted Data

0	0	0	0	0	1	1	1	2	2	2	2	4	4

```
void count_sort(int arr[],int n){

    int output[n];

    int count[10];

    memset(count, 0, sizeof(count));

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

        ++count[arr[i]];

    for (i = 1; i<10;i++)

        count[i] += count[i - 1];

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

        output[count[arr[i]] - 1] = arr[i];

        --count[arr[i]];

    }

}
```

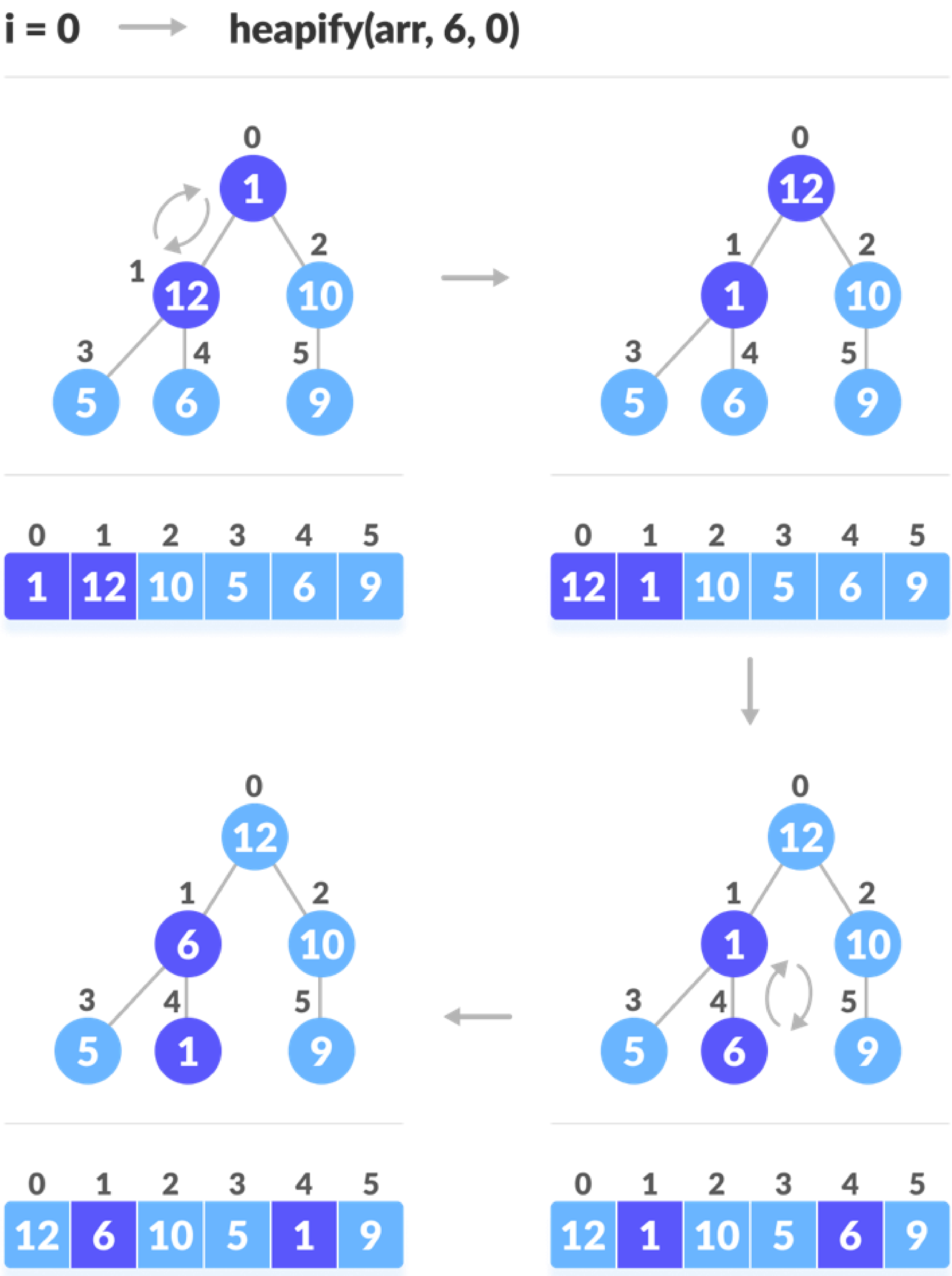
11. Radix Sort: The idea of it is to do digit by digit sort starting from least significant digit to most significant digit. It is upgraded form of counting sort.




```
void count_sort(int arr[],int n){

    int output[n];

    int count[10];

    memset(count, 0, sizeof(count));

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

        ++count[(arr[i]/exp)%10];

    for (i = 1; i<10;i++)

        count[i] += count[i - 1];

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

        output[count[(arr[i]/exp)%10]-1] = arr[i];

        --count[(arr[i]/exp)%10];

    }

}
```

```
void radix_sort(int arr[],int n){

        int max_ele=INT_MIN;

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

                max_ele=max(get_max,arr[i]);

        for(int exp=1;max_ele/exp>=0;exp*=10)

                count_sort(arr,n,exp);

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

                  cout<<arr[i]<<" ";

         }

}
```

12. Heap Sort: It is a **comparison based sorting** technique based on Binary Heap data structure.




```
int arr[10]= {0,1,3,17,2,30,7,25,19}

void heap_sort(int idx){

        int left=2*idx;
```

```
        int right=2*idx+1;

        int min_idx=idx;

        int last=arr.size()-1;

        if(left <= last && compare(arr[left],arr[idx])

                min_idx=left;

        if(right <= last && compare(arr[right],arr[idx])

                min_idx=right;

        if(min_idx!=idx){

                swap(arr[idx],arr[min_idx]);

                heap_sort(min_idx);

        }


bool compare(int a,int b){

        if(minHeap)

                return a<b;

        else

                return a>b;

}
```

13. Tree Sort: It is a sorting algorithm that is based on Binary Search Tree data structure.

```
void tree_sort(node *root, int arr[], int &index) {

    if (root != NULL){

        tree_sort(root->left, arr, i);

        arr[i++] = root->key;

        tree_sort(root->right, arr, i);

    }

}
```

14. Topoplogical Sort: It is for Directed Acyclic Graph (DAG) only.

Graph:
map<T,lis>l;

Using DFS

```
template <typename T>

void dfs(){

        map<T,bool> visited;

        list <T>ordering;

        for(auto p : l){

                T node=p.first;

                visited[node]=fasle;

        }

        for(auto p : l){

                T node=p.fisrt;

                if(!visited[node])

                        dfs_helper(node,visited,ordering);

        }

        for(auto node:ordering){

                cout<<node<<" ";

        }

}


void dfs_helper(T src,map <T,bool> &visited,list<T> &ordering){

        visited[src]=true;

        for(T nbr : l[src]){

                if(!visited[nbr]

                        dfs_helper(nbr,visited,ordering);

        }

        ordering.push_front(src);

        return ;

}
```

Using BFS

```
void topological_sort(){

        int *indegree=new int[V];

        for(int i=0;i<V;i++)

                indegree[i]=0;

        for(int i=0;i<V;i++){

                int x=p.first();

                for(auto y:l[i])

                        indegree[y]++;

        }

        queue <int> q

        for(int i=0;i<V;i++){

                if(inegree[i]==0)

                        q.push(i);

        }

        while(!q.empty()){

                int node=q.front()'

                cout<<node<<" ";

                q.pop();

                for(auto nbr:l[node]){

                        indegree[nbr]--;

                        if(inegree[i]==0)

                        q.push(i);

                }

        }

}
```

Time and Space Complexities of all the above algorithms:

DATE: / /

Algorithms	Time complexities: Best	Average	Worst	Space comp.
(1) Selection	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
(2) Bubble	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
(3) Insertion	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
(4) Shell	$\Omega(n)$	$\Theta(n\log n^2)$	$O(n\log n^2)$	$O(1)$
(5) Merge	$\Omega(n\log n)$	$\Theta(n\log n)$	$O(n\log n)$	$O(n)$
(6) Quick	$\Omega(n\log n)$	$\Theta(n\log n)$	$O(n^2)$	$O(n)$
(7) Bucket	$\Omega(n+K)$	$\Theta(n+K)$	$O(n^2)$	$O(nK)$
(8) Wave	$\Omega(n)$	$\Theta(n)$	$O(n)$	$O(n)$
(9) DNF	$\Omega(n)$	$\Theta(n)$	$O(n)$	$O(1)$
(10) Counting	$O(n+K)$	$O(n+K)$	$O(n+K)$	$O(n+2^K)$
(11) Radix	$\Omega(nK)$	$O(nK)$	$O(nK)$	$O(n+k)$
(12) Heap	$\Omega(n\log n)$	$\Theta(n\log n)$	$O(n\log n)$	$O(1)$
(13) Tree	$\Omega(n\log n)$	$\Theta(n\log n)$	$O(n^2)$	$O(n)$
(14) Topological	$\Omega(\lvert V\rvert+\lvert E\rvert)$	$\Theta(\lvert V\rvert+\lvert E\rvert)$	$O(\lvert V\rvert+\lvert E\rvert)$	$O(V+E)$