1.Reverse an array:

Method1:Iterative way

a. for(int i=0;i<n/2;i++)swap(arr[i],arr[n-i-1]);
b. start=0 and end=n-1.

```
while(start<end){
swap(arr[start++],arr[end–]);}
```

Method2:Recursive approach

```
void reverserec(int arr[],int n){
    if(n<=0)return;
    else{
        cout<<arr[n-1]<<" ";
        reverserec(arr,n-1);
    }
}
```

Complexity=O(n).

2.Signs of heaps:

1.if k is given
2.if smallest/largest.

priority_queue<int> q;----->max heap.
priority_queue<int,vector<int> geater<int>>------->min heap.

3.Kth max and min element:

Method 1:
A simple solution is to sort the given array using an O(N log N) sorting algorithm like Merge Sort, Heap Sort, etc, and return the element at index k-1 or n-k in the sorted array.

```
Smallest: int findKthLargest(vector<int>& nums, int k) {
        int n=nums.size();
        sort(nums.begin(), nums.end());
        return nums[k-1];
    }
Largest:returnnums[n-k];
```

Time Complexity of this solution is O(N Log N)

Method2:Heap

```
int findKthLargest(vector<int>& nums, int k) {
        priority_queue<int, vector<int>, greater<int>> q;
        int n=nums.size();
        for(int i=0;i<n;i++){
            if(q.size()<k){
                q.push(nums[i]);
            }
            else if(q.top()<nums[i]){
```

```
            int t=q.top();
            q.pop();
            q.push(nums[i]);
          }
          else continue;
        }
        return q.top();
    }
```

Complexity=O(k+(n-k)logk);

Note:findkthsmllest—>max heap
priority_queue<int> q;

4.K-sorted array/near sorted array:

Given an array of n elements, where each element is at most k away from its target position

Method1:use any sorting algo
complexity,time=O(n*logn)

Method2:Heap

```
int main(){
        int n,k;
        cin>>n>>k;
        int a[n];
        for(int i=0;i<n;i++)cin>>a[i];
        int size=(n==k)?k:k+1;
```

```
    priority_queue<int,vector<int>,greater<int>> q(a,a+size);
    int idx=0;
    for(int i=k+1;i<n;i++){
        a[idx++]=q.top();
        q.pop();
        q.push(a[i]);
    }
    while(!q.empty()){
        a[idx++]=q.top();
        q.pop();
    }
    for(int i=0;i<n;i++)cout<<a[i]<<" ";
    return 0;
}
```

Complexity:
The Min Heap based method takes O(k) + O((m) * log(k)) time where m = n – k and uses O(k) auxiliary space.

5.K-closest elements in array(sorted array).

Given a sorted array arr[] and a value X, find the k closest elements to X in arr[].

Here we r subtracting k from every array element then those which r near to k will get value 0 or near to it,then we will use max heap and will store pair of

p({abs(arr[i]-k,k),arr[i]});

Method 1:

```
vector<int> findClosestElements(vector<int>& arr, int k, int x)
{
      priority_queue<pair<int,int>> p;
      for(int i=0;i<arr.size();i++){
         if(p.size()<k)p.push({abs(arr[i]-x),arr[i]});
         else {
            if(((p.top()).first > abs(arr[i]-x)) || ((p.top()).first ==
abs(arr[i]-x) && arr[i]<(p.top()).second)){
                p.pop();
                p.push({abs(arr[i]-x),arr[i]});
             }
            else continue;
         }
      }
      vector<int> v;
      while(!p.empty()){
         v.push_back((p.top()).second);
         p.pop();
      }
      sort(v.begin(),v.end());//if resultant vector should be  sort
      return v;
   }
```

Complexity,time=O(n*logk).

6.K frequent elements:

Given an array of n numbers and a positive integer k. The problem is to find k numbers with most occurrences, i.e., the top k numbers having the

maximum frequency. If two numbers have the same frequency then the larger number should be given preference.

Method 1:Unordered_set

```
 vector<int> topKFrequent(vector<int>& nums, int k) {
        int n=nums.size();
        unordered_map<int,int> m;
        vector<int> v;

priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pai
r<int,int>>> p;
        for(int i=0;i<n;i++){
            m[nums[i]]++;
        }
        for(auto i=m.begin();i!=m.end();i++){
            p.push({i->second,i->first});
            if(p.size()>k)p.pop();
        }
        while(!p.empty()){
            pair<int, int> top = p.top();
            p.pop();
            v.push_back(top.second);
        }
        reverse(v.begin(),v.end());
        return v;
    }
```

Complexity Analysis:

Time Complexity: O(d logk + d), where d is the count of distinct elements in the array.

To remove the top of priority queue O(log d) time is required, so if k elements are removed then O(k log d) time is required and to traverse the distinct elements O(d) time is required.
Auxiliary Space: O(d), where d is the count of distinct elements in the array.
To store the elements in HashMap O(d) space complexity is needed.

Method 2:
Use one more array for storing occurences of each element and then push pair of both element and count in priority queue.

7.K closest points to origin:

Given a list of points on the 2-D plane and an integer K. The task is to find K closest points to the origin and print them.

```
vector<vector<int>> kClosest(vector<vector<int>>& points, int k)
{

      vector<vector<int>> v;
      priority_queue<pair<int,pair<int,int>>> p;
      for (int i = 0; i < points.size(); i++)
   {
         int l=points[i][0]*points[i][0]+points[i][1]*points[i][1];
         p.push({l,{points[i][0],points[i][1]}});
         if(p.size()>k)p.pop();

    }
```

```
    while(!p.empty()){
        pair<int,int> top=p.top().second;
        p.pop();
        v.push_back({top.first,top.second});
    }
    //reverse(v.begin(),v.end());
    return v;
}   //Method2:we can use ordered hasmap also….<l,i>;
```

8.Minimum cost of ropes:

There are given n ropes of different lengths, we need to connect these ropes into one rope. The cost to connect two ropes is equal to the sum of their lengths. We need to connect the ropes with minimum cost.

For example, if we are given 4 ropes of lengths 4, 3, 2, and 6.

```
long long minCost(long long arr[], long long n) {
    priority_queue<int,vector<int>,greater<int>> q;
    for(long long int i=0;i<n;i++){
        q.push(arr[i]);
    }
    long long int sum=0;
    while(!q.empty()){
        if(q.size()<=1)return sum;
        int c=q.top();q.pop();
        int d=q.top();q.pop();
        sum+=c+d;
        q.push(c+d);
    }
}
```

Complexity,time=O(n*logn)
space=O(n).

9.Sum of elements between k1'th and k2'th smallest elements:

Given an array of integers and two numbers k1 and k2. Find the sum of all elements between given two k1’th and k2’th smallest elements of the array. It may be assumed that (1 <= k1 < k2 <= n) and all elements of array are distinct.

```
long long sumBetweenTwoKth( long long A[], long long N,
long long K1, long long K2)
   {
      priority_queue<long int> q1;
      priority_queue<long int> q2;

      for(int i=0;i<N;i++){
         q1.push(A[i]);
         q2.push(A[i]);
         if(q1.size()>K1)q1.pop();
         if(q2.size()>K2)q2.pop();
      }
      long int s1=0,s2=0;
      while(!q1.empty()){
         s1+=q1.top();q1.pop();
      }
      q2.pop();
      while(!q2.empty()){
         s2+=q2.top();q2.pop();
```

```
        }

        return s2-s1;
    }
```

Complexity,time=O(n*logk)
space=O(k).

10.Sort Array containing 0,1,2.

Method 1:Using variables lo,mid,high

```
 void sort012(int a[], int n)
    {
        int lo=0,mid=0,high=n-1;
        while(mid<=high){
            switch(a[mid]){
                case 0:swap(a[lo++],a[mid++]);
                        break;
                case 1:mid++;break;
                case 2:swap(a[mid],a[high--]);break;
            }
        }
        return;
    }
```

Complexity,time=O(n),space=O(1)

Method 2:Keep count of 0,1 and 2

```
void sort012(int a[], int n)
```

```
{
    int i=0;
    int c0=0,c1=0,c2=0;
    for(;i<n;i++){
        if(a[i]==0)c0++;
        else if(a[i]==1)c1++;
        else c2++;
    }
    i=0;
    while(c0--)a[i++]=0;
    while(c1--)a[i++]=1;
    while(c2--)a[i++]=2;

      return;
}Complexity,time=O(n),space=1.
```

11.Move all negative elements to end

Method1:
We can simply store all positive values first then negative values by using an temp array.

```
void segregateElements(int arr[],int n)
{
    int t[n];
    int i=0,j=0;
  for(int i=0;i<n;i++){
      if(arr[i]>=0)t[j++]=arr[i];
  }
  if(j==n || j==0)return;
  for(int i=0;i<n;i++){
```

```
        if(arr[i]<0)t[j++]=arr[i];
      }
      memcpy(arr, t, sizeof(t));
      return;
    }
```

Complexity,time=O(2n),space=O(n).

Method 2:Efficient with constant space complexity→not keeping order

```
void shiftall(int arr[], int left,
              int right)
{
  while (left<=right)
 {
    if (arr[left] < 0 && arr[right] < 0)
      left+=1;

    else if (arr[left]>0 && arr[right]<0)
    {
      int temp=arr[left];
      arr[left]=arr[right];
      arr[right]=temp;
      left+=1;
      right-=1;
    }
    else if (arr[left]>0 && arr[right] >0)
      right-=1;
    else{
      left += 1;
```

```
      right -= 1;
    }
  }
}
void display(int arr[], int right){
   for (int i=0;i<=right;++i){
    cout<<arr[i]<<" ";
  }
  cout<<endl;
}
 int main()
{
  int arr[] = {-12, 11, -13, -5,
               6, -7, 5, -3, 11};
  int arr_size = sizeof(arr) /
                 sizeof(arr[0]);

  // Function Call
  shiftall(arr,0,arr_size-1);
  display(arr,arr_size-1);
  return 0;
}
```

Time comp=O(N)

Space comp=O(1)

12.Union of two sorted arrays.

```
int doUnion(int a[], int n, int b[], int m)  {
        int t[n+m];
```

```
        int i=0,j=0,k=0,c=0;
        while(i<n && j<m){
            if(a[i]<b[j]){t[k++]=a[i];i++;}
            else if(a[i]>b[j]){t[k++]=b[j];j++;}
            else {t[k]=a[i];i++;j++;}
            c++;
        }
        while(i<n){t[k++]=a[i];i++;c++;}
       while(j<m){t[k++]=a[j];j++;c++;}

        return c;
    }
```

Complexity=O(n+m).

Method1:Naive approach

```
bool binary_search(int x,int a[],int l,int r){
        if(r>=l){
        int mid=l+(r-l)/2;
        if(a[mid]==x)return true;
        if(a[mid]<x)return binary_search(x,a,mid+1,r);
        else return binary_search(x,a,l,mid-1);}
        else return false;
    }
    //Function to return the count of number of elements in
union of two arrays.
    int doUnion(int a[], int n, int b[], int m)  {
        int r[n+m];
        int i=0;
        int c=0;
```

```
        for(;i<n;i++){
            r[i]=a[i];c++;
        }
        for(int j=0;j<m;j++){
            if(binary_search(b[j],a,0,n))
            continue;
            else{ r[i]=b[j];c++;i++;};
        }
        return c;
    }
```

Expected Time Complexity : O((n+m)log(n+m))
Expected Auxilliary Space : O(n+m)

Method2:Using sets:

```
int doUnion(int a[], int n, int b[], int m)  {
        set<int> s;int c=0;
        for(int i=0;i<n;i++){s.insert(a[i]);}
        for(int i=0;i<m;i++){s.insert(b[i]);}
        return s.size();
    }
```

Time Complexity: O(m * log(m) + n * log(n))

14.cycly rotate an array by one:

```
void rotate(int arr[], int n)
{
    int a=arr[n-1];
```

```
    for(int i=n-1;i>0;i--){
        arr[i]=arr[i-1];
    }
    arr[0]=a;
}
```

Complexity=O(n).

15.Find duplicates in O(n) time and O(1) extra space in the array containing elements from 0 to n-1:

```
vector<int> duplicates(int arr[], int n) {
        vector<int> v;
        for(int i=0;i<n;i++){
            arr[arr[i]%n]=arr[arr[i]%n]+n;
        }
        for(int i=0;i<n;i++){
            if(arr[i] >= n*2)v.push_back(i);
            else continue;
        }
        if(v.size()==0){v.push_back(-1);return v;}
        else return v;
    }
```

16:Kadane's Algorithm Largest Sum Contiguous Subarray:

```
 long long maxSubarraySum(int a[], int n){
```

```
        long int n1=INT_MIN,m=0;
        for(int i=0;i<n;i++){
            m=m+a[i];
            if(n1<m)n1=m;
            if(m<0)m=0;

        }
        return n1;
    }
```

Complexity:Time=O(N)
Space=O(1)

17.Minimise the maximum difference between heights:

Given heights of n towers and a value k. We need to either increase or decrease the height of every tower by k (only once) where k > 0. The task is to minimize the difference between the heights of the longest and the shortest tower after modifications and output this difference.

```
int getMinDiff(int arr[], int n, int k) {
      sort(arr,arr+n);
     int tempmin=arr[0];
     int tempmax=arr[n-1];
     int ans=arr[n-1]-arr[0];
     for(int i=1;i<n;i++){
         if(arr[i]-k<0)continue;
         tempmin=min(arr[0]+k,arr[i]-k);
         tempmax=max(arr[n-1]-k,arr[i-1]+k);
```

```
        ans=min(ans,tempmax-tempmin);
    }
    return ans;
    }
```
Complexity:O(N*LogN).

18.Minimum no. of Jumps to reach end of an array:

```
int minJumps(int arr[], int n)
    {
        int maxR=arr[0],step=arr[0],jump=1;//maxR=max reacha
        if(n<=1)return 0;
        if(arr[0]==0)return -1;
        for(int i=1;i<n;i++){
            if(i==n-1)return jump;
            maxR=max(maxR,i+arr[i]);
            step--;
            if(step==0){
                jump++;
                if(i>=maxR)return -1;
                step=maxR-i;
            }
        }
    }
```
Complexity:Time=O(N).

19.Merge intervals:

Given a set of time intervals in any order, merge all overlapping intervals into one and output the result which should have only mutually exclusive intervals. Let the intervals be represented as pairs of integers for simplicity. For example, let the given set of intervals be {{1,3}, {2,4}, {5,7}, {6,8}}. The intervals {1,3} and {2,4} overlap with each other, so they should be merged and become {1, 4}. Similarly, {5, 7} and {6, 8} should be merged and become {5, 8}

```
vector<vector<int>> merge(vector<vector<int>>& v) {
        sort(v.begin(),v.end());
        stack<vector<int>> s;
         vector<vector<int>> v1;
        s.push(v[0]);
        for(int i=1;i<v.size();i++){
            vector<int> t=s.top();
            if(t[1] < v[i][0]){s.push(v[i]);}
           else  if(t[1] < v[i][1]){
                t[1]=v[i][1];
                s.pop();
                s.push(t);
            }
        }
        while(!s.empty()){
            v1.push_back(s.top());
            s.pop();
        }
        reverse(v1.begin(),v1.end());
        return v1;


    }
```

20.Inversion count:

Input: arr[] = {8, 4, 2, 1}
Output: 6

Explanation: Given array has six inversions:
(8, 4), (4, 2), (8, 2), (8, 1), (4, 1), (2, 1).

Method 1:O(n^2)

Method 2:Using merge sort

```
int merge(long int arr[],long int temp[],long int left,long int
mid,long int right){
    long long int i=left;int icount=0;
    long int j=mid;
    long int k=left;
    while((i<=mid-1) && (j<=right)){
        if(arr[i]<=arr[j]){
            temp[k++]=arr[i++];
        }
        else{
            temp[k++]=arr[j++];
            icount+=mid-i;//because left and right subarrays are sorted, so
all the remaining elements in left-subarray (a[i+1], a[i+2] … a[mid]) will be
greater than a[j].
        }
    }
```

```
    while(i<=mid-1)temp[k++]=arr[i++];
    while(j<=right)temp[k++]=arr[j++];
    for(int i=left;i<=right;i++)arr[i]=temp[i];

    return icount;
}
int merge_sort(long int arr[],long int temp[],long int left,long
int right){
    long long int icount=0,mid=0;
    if(right>left){
        mid=(right+left)/2;
        icount+=merge_sort(arr,temp,left,mid);
         icount+=merge_sort(arr,temp,mid+1,right);

         icount+=merge(arr,temp,left,mid+1,right);
    }
    return icount;
}
long long int inversionCount(long long arr[], long long N)
{
    long int temp[N];
    return merge_sort(arr,temp,0,N-1);
}
```

Complexity:O(Nlogn)

22:Count pairs with given sum :

Given an array of integers, and a number ‘sum’, find the number of pairs of integers in the array whose sum is equal to ‘sum’.

Method 1:Bruteforce

A simple solution is to traverse each element and check if there's another number in the array which can be added to it to give sum.

Complexity:Time=O(N^2)

Method 2:Using unordered map

We need to find (a,b),if a=arr[i] then b=k-arr[i],so we r finding b in map

```
int getPairsCount(int arr[], int n, int k) {
        unordered_map<int,int> m;
        int result=0;
        for(int i=0;i<n;i++){
            int b=k-arr[i];
            if(m[b])result+=m[b];
            m[arr[i]]++;
        }
        return result;
    }
```

Complexity:Time=O(n) space=O(N)

23.Common elements :

Given three arrays sorted in non-decreasing order, print all common elements in these arrays.

Method 1:
A simple solution is to first find intersection of two arrays and store the intersection in a temporary array, then find the intersection of third array and temporary array.
Time complexity of this solution is O(n1 + n2 + n3) where n1, n2 and n3 are sizes of ar1[], ar2[] and ar3[] respectively.

We are using extra space also.

Method 2:Using sets

```
vector <int> commonElements (int A[], int B[], int C[], int n1,
int n2, int n3)
    {
        int i=0,j=0,k=0;
        vector<int> v;
        unordered_set<int> s;
        while(i<n1 && j<n2 && k<n3){
            if(A[i]==B[j] && B[j]==C[k]){
                s.insert(A[i]);
                i++;j++;k++;
            }
            else if(A[i]<B[j])i++;
            else if(B[j]<C[k])j++;
```

```
            else k++;
        }
        for(auto i=s.begin();i!=s.end();i++)v.push_back(*i);
        sort(v.begin(),v.end());
        return v;
    }
```

Complexity:Time=O(N) space=o(1)

24.Rearrange the array in alternating positive and negative items with O(1) extra space:

```
class Solution{
public:

    void rearrange(int arr[], int n) {
        int i=0,j=0;
        vector<int> v;
        while(i<n || j<n){
            if(i<n)
            {
            while(arr[i]<0)i++;
            v.push_back(arr[i]);
            i++;
            }
            if(j<n)
            {
            while(arr[j]>=0 )j++;
            v.push_back(arr[j]);
            j++;
```

```
            }
        }
            for(int i=0;i<n;i++)arr[i]=v[i];
            return;

    }
};
```

25.Subarray with 0 sum:

```
class Solution{
    public:
    //Complete this function
    //Function to check whether there is a subarray present
with 0-sum or not.
    bool subArrayExists(int arr[], int n)
    {
        int sum=0;
        unordered_set<int> m;
        for(int i=0;i<n;i++){
            sum+=arr[i];
            if(sum==0  || (m.find(sum)!=m.end()))return true;

            m.insert(sum);
        }
        return false;
    }
};
```

26.Factorial of large numbers:

```
class Solution {
public:

    int multiply(int x,int t[],int t_size){
        int carry=0;
        for(int i=0;i<t_size;i++)
        {
            int prod=t[i]*x+carry;
            t[i]=prod%10;
            carry=prod/10;
        }
        while(carry){
            t[t_size]=carry%10;
            carry=carry/10;
            t_size++;
        }
        //cout<<t_size<<endl;
        return t_size;

    }
    vector<int> factorial(int N){
        vector<int> v;
        int t[10000];
        int t_size=1;
        t[0]=1;
        for(int i=2;i<=N;i++)t_size=multiply(i,t,t_size);
        //cout<<t_size<<endl;
```

```
        for(int i=t_size-1;i>=0;i--)
        v.push_back(t[i]);

        return v;
    }
};
```

Complexity ,O(NlogN!)

27.find maximum product subarray

Method 1:Naive approach

Generate all subarrays:

```
int maxSubarrayProduct(int arr[], int n)
{
    // Initializing result
    int result = arr[0];

    for (int i = 0; i < n; i++)
    {
        int mul = arr[i];
        // traversing in current subarray
        for (int j = i + 1; j < n; j++)
        {
            // updating result every time
```

```
            // to keep an eye over the maximum product
            result = max(result, mul);
            mul *= arr[j];
        }
        // updating the result for (n-1)th index.
        result = max(result, mul);
    }
    return result;
}
```

Complexity:O(N)^2

Method 2:Efficient approach

```
long long maxProduct(vector<int> a, int n) {

          long int ans = a[0];
           long int crnt = 1;
           for(int i=0;i<n;i++)
           {
              if(crnt==0)crnt = 1;
              crnt*=a[i];
              ans = max(ans,crnt);
           }
           crnt = 1;
           for(int i=n-1;i>=0;i--)
           {
              if(crnt==0)crnt = 1;
              crnt*=a[i];
```

```
            ans = max(ans,crnt);
          }
          return ans;
}
};
```

Complexity:
O(N).

28.Find longest consecutive subsequence:

Method 1:Naive approach

Sort the array and then check
Complexity:O(N*logN) space:O(1)

Method 2:min heap

```
int findLongestConseqSubseq(int arr[], int N)
   {
     priority_queue<int,vector<int>,greater<int>> p;
      for(int i=0;i<N;i++){
          p.push(arr[i]);
      }
      int res=1;
     int count=1;
     int prev=p.top();
```

```
    p.pop();
    while(!p.empty()){
        int t=p.top();p.pop();
        if(t-prev==1)count++;
        else if(t-prev==0)continue;
        else count=1;
         res=max(count,res);
        prev=t;
    }
    return res;
   }
```

Complexity:
Time—>O(N)
Space—> O(N)

29.Given an array of size n and a number k, find all elements that appear more than " n/k " times.

Method 1:

```
 vector<int> majorityElement(vector<int>& nums) {
       vector<int> v;
       unordered_map<int,int> m;
       int n=nums.size();
       for(int i=0;i<n;i++){
           m[nums[i]]++;
       }
```

```
        int k=n/3;
        for(auto i=m.begin();i!=m.end();i++){
            if((i->second)>k)v.push_back(i->first);
            else continue;
        }
        return v;
    }
```

Complexity:Time=O(N)
Space=O(N)

30.Maximum profit by buying and selling a share only 1—1:

Method 1:Brute force approach

For every we will traverse entire array to find max element:
complexity:time=O(N) space=O(1)

Method 2:
we can use another arbiratory array where we store the maxx_ellement posible in the higher index.

E,g: a1=[4,1,2,0,20,9,8,5,2}

We are creating arb array by starting from last index of a1 array by checking condition

```
i=n-1;
max_so_far=max(max_so_far,a1[i])
arb[i]=max_so_far
 arb=[20,20,20,20,20,9,8,5,2]
```

Then,
For i=0 to n-1:
max_profit=max(max_profit,arb[i]-a1[i]);

Complexity:Time=O(N) Space=O(N)

Method 3:

```
int maxProfit(int prices[], int n)
{
   int buy = prices[0], max_profit = 0;
   for (int i = 1; i < n; i++) {

      // Checking for lower buy value
      if (buy > prices[i])
         buy = prices[i];

      // Checking for higher profit
      else if (prices[i] - buy > max_profit)
```

```
            max_profit = prices[i] - buy;
        }
        return max_profit;
    }
```

Complexity:

Time=O(N) Space=O(1)

32.Maximum profit by buying and selling a share only 1—2:

You are given an integer array prices where prices[i] is the price of a given stock on the ith day.
On each day, you may decide to buy and/or sell the stock. You can only hold at most one share of the stock at any time. However, you can buy it then immediately sell it on the same day.

```
    int maxProfit(vector<int>& prices) {
        int n=prices.size();
        int profit=0;
        for(int i=1;i<n;i++){
            if(prices[i]>prices[i-1])profit+=prices[i]-prices[i-1];
            else continue;
        }
        return profit;
    }
```

Note:

ar[]=8 4 5 3 1 7 9
ar1[]=5 1 3 7 9

arr1 is not subset of ar but is subsequence.

33.Find whether an array is a subset of another array:

m=size(arr1)
n=subarray size(arr2)
Method 1:
Use two loops: The outer loop picks all the elements of arr2[] one by one. The inner loop linearly searches for the element picked by the outer loop. If all elements are found then return 1, else return 0.

Complexity:time=O(n*m) space=O(1)

Method 2:--->Use Sorting and Binary Search

1.Sort arr1[] which takes O(mLogm)
2.For each element of arr2[], do binary search for it in sorted arr1[].
3.If the element is not found then return 0.
4.If all elements are present then return 1.

Complexity:Time=O(mlogm+nlogm)

Method 3:sets

```
string isSubset(int a1[], int a2[], int n, int m) {
    unordered_set<int> s;
    int j=0;
    for(int i=0;i<n;i++){
        s.insert(a1[i]);
    }
  for(int i=0;i<m;i++){
       if(s.find(a2[i])==s.end())return "No";
    }
    return "Yes";

}
```

Complexity:
Time=O(n+m)
space=O(m)

34.Find the triplet that sum to a given value:

Method 1:
A simple method is to generate all possible triplets and compare the sum of every triplet with the given value.It will use 3 loops.

complexity:time=O(n)^3 space=O(1)

Method 2:Sorting

```
bool find3Numbers(int A[], int n, int X)
  {
     int l,r;
     sort(A,A+n);
     for(int i=0;i<n-1;i++)
     {
       l=i+1;
       r=n-1;
       while(r>l){
          if(A[i]+A[l]+A[r]==X)return 1;
          else if(A[i]+A[l]+A[r] >X)r--;
          else l++;
       }
     }

     return 0;
  }
```

Complexity:time=O(N^2) space=O(1)

35.Trapping Rain water problem:

Method 1:

The idea is to traverse every array element and find the highest bars on the left and right sides. Take the smaller of two heights. The difference between the smaller height and

height of the current element is the amount of water that can be stored in this array element.

Complexity:
Time: O(n^2)
space=O(n)

Method 2:

```
ong long trappingWater(int arr[], int n){
        int left_max_so_far[n];
        int right_max_so_far[n];
        int max_so_far=INT_MIN;
        for(int i=0;i<n;i++){
            max_so_far=max(max_so_far,arr[i]);
            left_max_so_far[i]=max_so_far;
        }
        max_so_far=INT_MIN;
        for(int i=n-1;i>=0;i--){
            max_so_far=max(max_so_far,arr[i]);
            right_max_so_far[i]=max_so_far;
        }
        int max_lr=0;
        long int result=0;
        for(int i=0;i<n;i++){
            max_lr=min(right_max_so_far[i],left_max_so_far[i]);
            result+=max_lr-arr[i];
        }
        return result;
    }
```

Complexity:time=O(N)
space=O(N).

Method 3:Most efficient

Instead of maintaining two arrays of size n for storing the left and a right max of each element, maintain two variables to store the maximum till that point. Since water trapped at any element = min(max_left, max_right) – arr[i]. Calculate water trapped on smaller elements out of A[lo] and A[hi] first, and move the pointers till lo doesn't cross hi.

```
int trap(vector<int>& height) {
        int left_max=0,right_max=0;
        int lo=0,hi=height.size()-1;
        int result=0;
        while(lo<=hi){
            if(height[lo]<height[hi]){
                if(height[lo] > left_max)left_max=height[lo];
                else
                    result+=left_max-height[lo];

                lo++;
```

```
        }
        else{
            if(height[hi] > right_max)right_max=height[hi];
            else
                result+=right_max-height[hi];

            hi--;
        }
    }
    return result;
}
```

Here space=O(1)

36.Chocolate Distribution problem:

Method 1:
A simple solution is to generate all subsets of size m of arr[0..n-1]. For every subset, find the difference between the maximum and minimum elements in it. Finally, return the minimum difference.

Method 2:Using sorting

```
long long findMinDiff(vector<long long> a, long long n, long long m){
    sort(a.begin(),a.end());
```

```
    int j=m-1;
    int i=0;
    int res=INT_MAX;
    while(j<n){
       if(res>a[j]-a[i])res=a[j]-a[i];
       i++;j++;
   }
       return res;
   }
```

Complexity: Time=O(n*logn) space=O(1)

37.Smallest Subarray with sum greater than a given value:

Here we need to return size of that smallest array—>

Method 1:

A simple solution is to use two nested loops. The outer loop picks a starting element, the inner loop considers all elements (on right side of current start) as ending element. Whenever sum of elements between current start and end becomes more than the given number, update the result if current length is smaller than the smallest length so far.

Complexity: Time=O(N^2) **Space=O(1)**

Method 2:

```
int minSubArrayLen(int target, vector<int>& nums)
  {
     int start=0,end=0;
     int n=nums.size();
     int min_len=n+1;
     int curr=0;
     while(end<n){
        while(curr<target && end<n)curr+=nums[end++];
        while(curr>=target && start<n)
        {
           if(end-start < min_len)min_len=end-start;

           curr-=nums[start++];
        }
     }

     if(min_len==n+1)return 0;
     return min_len;
  }
```

Complexity:Time=O(N) **Space=O(1)**

38.Three way partitioning of an array around a given value:

Given an array and a range [lowVal, highVal], partition the array around the range such that array is divided in three parts.
1) All elements smaller than lowVal come first.
2) All elements in range lowVal to highVal come next.
3) All elements greater than highVal appear in the end.

Method 1:Sorting

Method 2:

```
void threeWayPartition(vector<int>& arr,int a, int b)
   {
      int n=arr.size();
      int left=0,right=n-1;
   for(int i=0;i<=right;i++){
      if(arr[i]<a)
      {
         swap(arr[i],arr[left]);
         left++;
      }
      else if(arr[i]>b){
         swap(arr[i],arr[right]);
         right--;
         i--;
      }
   }
   return;
```

}

Complexity: Time=O(N) Space=O(1)

39.Minimum swaps required bring elements less equal K together:

Method 1:Naive approach

A simple solution is to first count all elements less than or equals to k(say 'good'). Now traverse for every sub-array and swap those elements whose value is greater than k. Time complexity of this approach is O(n2)

Method 2:

An effiecient approach is to use two pointer technique and sliding window. Time complexity of this approach is O(n)

1.Find count of all elements which are less than or equals to 'k'. Let's say the count is 'cnt'
2.Using two pointer technique for window of length 'cnt', each time keep track of how many elements in this range are greater than 'k'. Let's say the total count is 'bad'.
3.Repeat step 2, for every window of length 'cnt' and take minimum of count 'bad' among them. This will be the final answer.

```
int minSwap(int arr[], int n, int k) {
      int count=0,bad=0;
      for(int i=0;i<n;i++)if(arr[i]<=k)count++;
      for(int i=0;i<count;i++)if(arr[i] > k)bad++;
      int result=bad;
      for(int i=0,j=count;j<n;i++,j++)
      {
         if(arr[i] >k)bad--;
         if(arr[j]>k)bad++;
         result=min(result,bad);
      }
      return result;
   }
```

40.Minimum no. of operations required to make an array palindrome:

Given an array of positive integers. We need to make the given array a 'Palindrome'. The only allowed operation is"merging" (of two adjacent elements). Merging two adjacent elements means replacing them with their sum. The task is to find the minimum number of merge operations required to make the given array a 'Palindrome'.

```
int findMinOps(int arr[], int n)
{
   int ans = 0; // Initialize result

   // Start from two corners
```

```
for (int i=0,j=n-1; i<=j;)
{
    // If corner elements are same,
    // problem reduces arr[i+1..j-1]
    if (arr[i] == arr[j])
    {
        i++;
        j--;
    }

    // If left element is greater, then
    // we merge right two elements
    else if (arr[i] > arr[j])
    {
        // need to merge from tail.
        j--;
        arr[j] += arr[j+1] ;
        ans++;
    }

    // Else we merge left two elements
    else
    {
        i++;
        arr[i] += arr[i-1];
        ans++;
    }
}

return ans;
```

}

Complexity:O(N)

41.Median of 2 sorted arrays of equal size:

Method 1:Merge the two arrays and then find

Complexity ,time=O(2N)

Method 2:Simply count while Merging

Use the merge procedure of merge sort. Keep track of count while comparing elements of two arrays. If count becomes n(For 2n elements), we have reached the median. Take the average of the elements at indexes n-1 and n in the merged array. See the below implementation.

complexity:O(N)
Method 3:Optimal approach

```
#include<bits/stdc++.h>
using namespace std;

int median(int arr[],int);
int getmedian(int arr1[],int arr2[],int n){
    if(n<=0)return -1;
    if(n==1)return (arr1[0]+arr2[0])/2;
    if(n==2)return (min(arr1[0],arr2[0])+max(arr1[1],arr2[1]))/2;
```

```
    int m1=median(arr1,n);
    int m2=median(arr2,n);
    if(m1==m2)return m1;
    if(m1<m2){
        if(n%2==0)return getmedian(arr1+n/2-1,arr2,n-n/2+1);
        else return getmedian(arr1+n/2,arr2,n-n/2);
    }
    else{
        if(n%2==0)return getmedian(arr1,arr2+n/2-1,n-n/2+1);
        else return getmedian(arr1,arr2+n/2,n-n/2);
    }
}
int median(int arr[],int n){
    if(n%2==0)return (arr[n/2]+arr[n/2-1])/2;
    else return arr[n/2];
}

int main(){
    int arr1[]={1,2,3,4};
    int arr2[]={5,6,7,8};
    int n=sizeof(arr1)/sizeof(arr1[0]);
    cout<<n<<endl;
    cout<<getmedian(arr1,arr2,n);
    return 0;
}
```

42.Merge 2 sorted arrays without using Extra space.

```
void merge(int nums1[], int nums2[], int n, int m) {
        int i=n-1;
      int j=0;
      while(i>=0 && j<m){
        if(nums1[i]>nums2[j])swap(nums1[i],nums2[j]);
        i--;j++;
      }
      sort(nums1,nums1+n);
      sort(nums2,nums2+m);
      }
```

Complexity:Time=O((n+m)log(n+m)) Space=O(1)

HEAPS:

```
#include<bits/stdc++.h>
using namespace std;
void swap(int *a,int *b)
{
   int t=*a;
   *a=*b;
   *b=t;
}
void heapify(int i,vector<int> &v,int size)
{
   int l=2*i+1;
   int r=2*i+2;
```

```
    int smallest=i;
    if(l<size && v[smallest] > v[l])
        smallest=l;
    if(r<size && v[smallest] > v[r])
        smallest=r;
    if(smallest!=i)
    {
        swap(&v[smallest],&v[i]);
        heapify(smallest,v,size);
    }
}
void insert(int val,vector<int> &v)
{
    int i=v.size();
    v.push_back(val);
    while(i>0 && v[(i-1)/2] > v[i])
        {
        swap(&v[(i-1)/2],&v[i]);
        i=(i-1)/2;
        }
}

int extract_min(vector<int> &v)
{
    int mini=v[0];
    v[0]=v[v.size()-1];
    v.pop_back();
    heapify(0,v,v.size());
    return mini;
}
```

```
void decrease_key(int i,int val,vector<int> &v)
{
   v[3]=val;
   while(i>0 && v[(i-1)/2] > v[i])
   {
      swap(&v[(i-1)/2],&v[i]);
      i=(i-1)/2;
   }
}

void delete_key(int i,vector<int> &v)
{
   decrease_key(i,INT_MIN,v);
   extract_min(v);

}
void build_heap(vector<int> &heap,int n)
{
   for(int i=n/2-1;i>=0;i--)
   {
      heapify(i,heap,n);
   }
}
void heapsort(vector<int> heap)
{
   int n=heap.size();
   build_heap(heap,n);
   for(int i=n-1;i>=0;i--)
   {
```

```
        swap(&heap[0],&heap[n-1]);
        heapify(0,heap,i);
    }
}
int main()
{
    vector<int> v;
    v.push_back(1);
    v.push_back(3);
    v.push_back(4);
    v.push_back(5);
    v.push_back(6);
    v.push_back(7);

    /*insert(2,v);
    decrease_key(3,2,v);
    v[0]=9;
    heapify(0,v,v.size());
    cout<<extract_min(v)<<endl;
    delete_key(3,v);*/
    int arr[]={9,8,7,6,5,4,3,2,1};
    int n=sizeof(arr)/sizeof(arr[0]);
    vector<int> hp(arr,arr+n);
    //build_heap(hp,n);
    heapsort(hp);
    reverse(hp.begin(),hp.end());
    for(auto x:hp)cout<<x<<" ";
    return 0;
```

```
}
```