Sorting

Sorting any sequence means to arrange the elements of that sequence according to some specific criterion.

In-Place Sorting: An in-place sorting algorithm uses constant extra space for producing the output (modifies the given array only). It sorts the list only by modifying the order of the elements within the list.

Insertion Sort

Insertion Sort is an In-Place sorting algorithm. This algorithm works in a similar way of sorting a deck of playing cards.

The idea is to start iterating from the second element of array till last element and for every element insert at its correct position in the subarray before it.

Algorithm:

```
Step 1: If the current element is 1st element of array,
        it is already sorted.
Step 2: Pick next element
Step 3: Compare the current element will all elements
        in the sorted sub-array before it.
Step 4: Shift all of the elements in the sub-array before
        the current element which are greater than the current
        element by one place and insert the current element
        at the new empty space.
Step 5: Repeat step 2-3 until the entire array is sorted.
```

Time Complexity: $O(N^2)$, where N is the size of the array.

```cpp
//insertion sort
//time complexity O(n^2) in worst case /happen input array is in revese order
//in place  and stable
//used for small array sorting
//time complexity O(n) in best case /happen when array is already in sorted

//single elemenet is always be sorted

#include <iostream>
#include <algorithm>
using namespace std;

void iSort(int arr[],int n){

    for(int i=1;i<n;i++){
        int key = arr[i];
        int j=i-1;
        while(j>=0 && arr[j]>key){
            arr[j+1]=arr[j];
            j--;
        }
        arr[j+1]=key;
    }
}

int main() {

    int arr[]={50,20,40,60,10,30};

    int n=sizeof(arr)/sizeof(arr[0]);
    iSort(arr,n);

    for(auto x: arr)
        cout<<x<<" ";
}
```

OUTPUT

10 20 30 40 50 60

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. The algorithm maintains two subarrays in a given array.

1. The subarray which is already sorted.
2. Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

```cpp
//selection sort is comparsion base algorithm
//it takes less memory as compare to other algorthitm
//it is not stable
// it does not require extra memory for sorting


// This method sorts, a set of unsorted elements in two steps.
// • In the first step, find the smallest element in the structure.
// • In the second step, swap the smallest element with the
// element at the first position.
// • Then, find the next smallest element and swap with the
// element at the second position.
// • Repeat these steps until all elements get arranged at proper
// positions.


#include<iostream>
using namespace std;

void selectionSort(int arr[], int n)
{
    for(int i=0;i<n;i++)
    {
        int min_ind=i;
        for(int j=i+1;j<n;j++)
        {
            if(arr[j]<arr[min_ind])
                min_ind=j;
        }
        if(min_ind!=i)
            swap(arr[i],arr[min_ind]);
    }
}

int main()
{
    int a[] = {2, 1, 3, 4};
    selectionSort(a, 4);
    for(int i = 0;i < 4; i++){
        cout<<a[i]<<" ";
    }
    return 0;
}
```

OUTPUT:

1 2 3 4

Bubble Sort

```cpp
#include<iostream>
using namespace std;

//in bubble sort element is compare with its adjacent element

void bubbleSort(int arr[], int n)
{
    for(int i=0;i<n-1;i++)
    {
        for(int j=0;j<n-i-1;j++)
            {
                if(arr[j]>arr[j+1])
                    swap(arr[j],arr[j+1]);
            }
    }
}

int main()
{
    int a[] = {2, 1, 3, 4};
    bubbleSort(a, 4);
    for(int i = 0;i < 4; i++){
        cout<<a[i]<<" ";
    }
    return 0;
}
```

OUTPUT:

1 2 3 4

OPTIMIZED BUBBLE SORT

```
#include<iostream>
using namespace std;

void bubbleSort(int arr[], int n)
{
    bool swapped;
    for(int i=0;i<n-1;i++)
    {
        swapped=false;
        for(int j=0;j<n-i-1;j++)
        {
            if(arr[j]>arr[j+1])
            swap(arr[j], arr[j+1]);
            swapped=true;
        }
        if(swapped==false)
            break;
    }
}

int main()
{
    int a[]={2,1,3,4};
    bubbleSort(a,4);
    for(int i=0;i<4;i++)
        cout<<a[i]<<" ";

    return 0;
}
```

OUTPUT

1 2 3 4

SORT STL CODE VECTOR

```cpp
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;

int main()
{
    vector<int> v={10,20,5,7};
    sort(v.begin(),v.end());

    for(int x:v)
        cout<<x<<" ";

    sort(v.begin(),v.end(),greater<int>());

    cout<<endl;
    for(int x:v)
        cout<<x<<" ";

}
```

OUTPUT:

5 7 10 20

20 10 7 5

SORT STL OWN OBJECT VECTOR

```cpp
#include<iostream>
#include<algorithm>
using namespace std;

struct Point{
    int x,y;

};

bool MyComp(Point p1, Point p2)
{
    return p1.x<p2.x;
}

int main()
{
    Point arr[]={ {3,10},{2,8},{5,4}};
    int n=sizeof(arr)/sizeof(arr[0]);
    sort(arr,arr+n,MyComp);

    for(auto i: arr)
        cout<<i.x<<" "<<i.y<<endl;
}
```

OUTPUT:

2 8

3 10

5 4

Merge Two Sorted Array

```cpp
//time O((m+n)*log(m+n))
//auxiliary space theta(m+n)

//time O((m+n)*log(m+n))
//auxiliary space theta(m+n)

#include <iostream>
#include <algorithm>
using namespace std;

void merge(int a[], int b[], int m, int n){

    int c[m+n];
    for(int i=0;i<m;i++)
        c[i]=a[i];
    for(int j=0;j<n;j++)
        c[j+m]=b[j];

    sort(c,c+m+n);

    for(int i=0;i<m+n;i++)
        cout<<c[i]<<" ";
}

int main() {

    int a[]={10,15,20,40};
    int b[]={5,6,6,10,15};

    int m=sizeof(a)/sizeof(a[0]);
    int n=sizeof(b)/sizeof(b[0]);
    merge(a,b,m,n);
}
```

OUTPUT:

5 6 6 10 10 15 15 20 40

Efficient solution of Merge Two Sorted Array:

```
//theta(m+n)

#include<iostream>
#include<algorithm>
using namespace std;

void merge(int a[], int b[], int m , int n)
{
    int i=0,j=0;
    while (i<m && j<n)
    {
        if(a[i]<b[j])
            cout<<a[i++]<<" ";
        else
            cout<<b[j++]<<" ";
    }

    while(i<m)
        cout<<a[i++]<<" ";

    while(j<n)
        cout<<b[j++]<<" ";

}

int main() {

    int a[]={10,15,20,40};
    int b[]={5,6,6,10,15};

    int m=sizeof(a)/sizeof(a[0]);
    int n=sizeof(b)/sizeof(b[0]);
    merge(a,b,m,n);
}
```

OUTPUT:

5 6 6 10 10 15 15 20 40

Merge Function Of Merge Sort

```cpp
// Here we take a single array with three points namely, the lower, the middle
 and the highest point.
// The elements from the lower to the middle are sorted and the elements from
the (middle+1) to the higher are sorted.
// The task is to merge these two sorted parts into one.

#include<iostream>
#include<algorithm>
using namespace std;

void merge(int arr[], int l, int m, int h)
{
    int n1=m-l+1,n2=h-m;
    int left[n1],right[n2];

    for(int i=0;i<n1;i++)
        left[i]=arr[i];
    for(int j=0;j<n2;j++)
        right[j]=arr[m+1+j];

    int i=0,j=0,k=l;
    while(i<n1 && j<n2)
    {
        if(left[i]<=right[j])
            arr[k++]=left[i++];
        else
            arr[k++]=right[j++];
    }

    while(i<n1)
        arr[k++]=left[i++];
    while(j<n2)
        arr[k++]=right[j++];
}

int main() {

    int a[]={10,15,20,40,8,11,15,22,25};
    int l=0,h=8,m=3;

    merge(a,l,m,h);
    for(int x: a)
        cout<<x<<" ";
}
```

OUTPUT: 8 10 11 15 15 20 22 25 40

Merge Sorting algorithm

Merge Sort is a Divide and Conquer algorithm. It divides the input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one in a sorted manner. See following implementation for details:

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

Time Complexity: **Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.**
T(n) = 2T(n/2) + Θ(n)
The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is Θ(nLogn).

Time complexity of Merge Sort is Θ(nLogn) **in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.**

Auxiliary Space: **O(n)**

```cpp
//merge sort is divide and conquer algorithm
// it is stable algorithm
//it work in theta(nlongn) times and o(n) auxiliary space
//well suited for external sorting

#include<iostream>
using namespace std;
void merge(int arr[], int l, int m, int h)
{
    int n1=m-l+1,n2=h-m;
    int left[n1],right[n2];
```

```
    for(int i=0;i<n1;i++)
        left[i]=arr[i+l];
    for(int j=0;j<n2;j++)
        right[j]=arr[m+1+j];

    int i=0,j=0,k=l;
    while (i<n1 && j<n2)
    {
        if(left[i]<=right[j])
            arr[k++]=left[i++];
        else
            arr[k++]=right[j++];
    }

    while(i<n1)
        arr[k++]=left[i++];

    while(j<n2)
        arr[k++]=right[j++];

}

void mergeSort(int arr[], int l,int r)
{
    if(l<r)
    {
        int m=l+(r-l)/2;
        mergeSort(arr,l,m);
        mergeSort(arr,m+1,r);
        merge(arr,l,m,r);
    }
}

int main()
{
    int a[]={10,5,30,15,7};
    int l=0,r=4;

    mergeSort(a,l,r);
    for(int x: a)
        cout<<x<<" ";


    return 0;

}
```

OUTPUT: 5 7 10 15 30

Intersection Of Two Sorted Array

Naïve:

```cpp
#include<iostream>
using namespace std;

void intersection(int a[], int b[], int m, int n)
{
    for(int i=0;i<m;i++)
    {
        if(i>0 && a[i]==a[i-1])
            continue;

        for(int j=0;j<n;j++)
        {
            if(a[i]==b[j])
            {
                cout<<a[i]<<" ";
                break;
            }
        }
    }
}

int main()
{
    int a[]={3,5,10,10,10,15,15,20};
    int b[]={5,10,10,15,30};

    int m=sizeof(a)/sizeof(a[0]);
    int n=sizeof(b)/sizeof(b[0]);

    intersection(a,b,m,n);

    return 0;
}
```

OUTPUT

5 10 15

Efficient Solution of Intersection Of Two Sorted Array

```
#include<iostream>
using namespace std;

void intersection(int a[], int b[], int m, int n)
{
    int i=0,j=0;
    while(i<m && j<n)
    {
        if(i>0 && a[i-1]==a[i])
        {
            i++;
            continue;
        }

        if(a[i]<b[j])
            i++;
        else if(a[i]>b[j])
            j++;

        else
        {
            cout<<a[i]<<" ";
            i++;
            j++;
        }

    }
}

int main()
{
    int a[]={3,5,10,10,10,15,15,20};
    int b[]={5,10,10,15,30};

    int m=sizeof(a)/sizeof(a[0]);
    int n=sizeof(b)/sizeof(b[0]);

    intersection(a,b,m,n);
}
```

OUTPUT:

5 10 15

Union Of Two Sorted Array

Naïve:

```cpp
#include<iostream>
#include<algorithm>
using namespace std;

void printUnion(int a[], int b[], int m, int n)
{
    int c[m+n];
    for(int i=0;i<m;i++)
        c[i]=a[i];

    for(int i=0;i<n;i++)
        c[m+i]=b[i];
    sort(c,c+m+n);

    for(int i=0;i<m+n;i++)
    {
        if( i==0 || c[i]!=c[i-1])
            cout<<c[i]<<" ";
    }

}

int main()
{
    int a[]={3,8,10};
    int b[]={2,8,9,10,15};

    int m=sizeof(a)/sizeof(a[0]);
    int n=sizeof(b)/sizeof(b[0]);

    printUnion(a,b,m,n);

    return 0;
}
```

OUTPUT:

2 3 8 9 10 15

Efficient Solution of Union Of Two Sorted Array

```cpp
#include<iostream>
using namespace std;

//time complexity O(m+n)
//auxilary space o(1)

void printUnion(int a[], int b[], int m, int n)
{
    int i=0,j=0;
    while (i<m && j<n)
    {
        if(i>0 && a[i-1]==a[i])
        {
            i++;
            continue;
        }
        if(j>0 && b[j-1]==b[j])
        {
            j++;
            continue;
        }

        if(a[i]<b[j])
            cout<<a[i++]<<" ";

        else if(a[i]>b[j])
            cout<<b[j++]<<" ";
        else
        {
            cout<<a[i++]<<" ";
            j++;
        }
    }

    while(i<m)
    {
        if(i==0 || a[i-1]!=a[i])
            cout<<a[i]<<" ";
        i++;
    }

    while(j<n)
    {
        if(j==0 || b[j-1]!=b[j])
            cout<<b[j]<<" ";
        j++;
    }
```

```
}


int main()
{
    int a[]={3,8,8};
    int b[]={2,8,8,10,15};

    int m=sizeof(a)/sizeof(a[0]);
    int n=sizeof(b)/sizeof(b[0]);

    printUnion(a,b,m,n);
}
```

OUTPUT:

2 3 8 10 15

Count Inversion in Array

Naïve solution :

```
//condition for count inversion array
//i<j and arr[i]>arr[j]

#include<iostream>
using namespace std;

int countInversion(int arr[], int n)
{
    int res=0;
    for(int i=0;i<n-1;i++)
    {
        for(int j=i+1;j<n;j++)
        {
             if(arr[i]>arr[j])
                res++;
        }
    }

    return res;
}

int main()
{
    int arr[]={2,4,1,3,5};

    int n=sizeof(arr)/sizeof(arr[0]);

    cout<<countInversion(arr,n);
}
```

OUTPUT:

3

Efficient Solution for count inversion in Array

```
//condition for count inversion array
//i<j and arr[i]>arr[j]

#include<iostream>
using namespace std;

int countAndMerge(int arr[], int l, int m,int r)
{
    int n1=m-l+1,n2=r-m;
    int left[n1],right[n2];
    for(int i=0;i<n1;i++)
        left[i]=arr[i+l];
    for(int j=0;j<n2;j++)
        right[j]=arr[m+1+j];

    int res=0,i=0,j=0,k=l;
    while(i<n1 && j<n2)
    {
        if(left[i]<right[j])
            arr[k++]=left[i++];
        else
        {
            arr[k++]=right[j++];
            res=res+(n1-i);
        }
    }

    while(i<n1)
        arr[k++]=left[i++];
    while(j<n2)
        arr[k++]=right[j++];
    return res;
}

int countInv(int arr[], int l, int r)
{
    int res=0;
    if(l<r)
    {
        int m=(r+l)/2;

        res+=countInv(arr,l,m);
        res+=countInv(arr,m+1,r);
        res+=countAndMerge(arr,l,m,r);
    }
    return res;
}
```

```
int main()
{
    int arr[]={2,4,1,3,5};

    int n=sizeof(arr)/sizeof(arr[0]);

    cout<<countInv(arr,0,n-1);
}
```

:OUTPUT:

3

Naïve Partition

```cpp
// naive parition require three taversal but is is stable

#include <bits/stdc++.h>
using namespace std;
void partition(int arr[], int l, int h, int p)
{
    int temp[h-l+1],index=0;
    for(int i=l;i<=h;i++)
        if(arr[i]<=arr[p] && i != p)
            {
                temp[index]=arr[i];index++;
            }
    temp[index++] = arr[p];
    for(int i=l;i<=h;i++)
        if(arr[i]>arr[p])
            {
                temp[index]=arr[i];index++;
            }
    for(int i=l;i<=h;i++)
        arr[i]=temp[i-l];
}

int main() {

    int arr[]={5,13,6,9,12,11,8};

    int n=sizeof(arr)/sizeof(arr[0]);

    partition(arr,0,n-1,3);

    for(int x: arr)
        cout<<x<<" ";
}
```

OUTPUT

5 6 8 9 13 12 11

Lomuto Partition

```cpp
//it traverse only one time but
// is stable sorting algorithm

#include<iostream>
using namespace std;
int iPartition(int arr[], int l , int h)
{
    int pivot=arr[h];
    int i=l-1;
    for(int j=l;j<=h-1;j++)
    {
        if(arr[j]<pivot)
        {
            i++;
            swap(arr[i],arr[j]);
        }
    }
    swap(arr[i+1],arr[h]);
    return i+1;
}

int main() {

    int arr[]={10,80,30,90,40,50,70};

    int n=sizeof(arr)/sizeof(arr[0]);

    iPartition(arr,0,n-1);

    for(int x: arr)
        cout<<x<<" ";
}
```

OUTPUT:

10 30 40 50 70 90 80

Hoare Partition

```cpp
//it traverse only one time but it is unstable

#include<iostream>
using namespace std;
int partition(int arr[], int l, int h)
{
    int pivot=arr[l];
    int i=l-1,j=h+1;
    while(true){
        do{
            i++;
        }while(arr[i]<pivot);

        do{
            j--;
        }while(arr[j]>pivot);

        if(i>=j)
            return j;
        swap(arr[i],arr[j]);
    }
}

int main() {

    int arr[]={5,3,8,4,2,7,1,10};

    int n=sizeof(arr)/sizeof(arr[0]);

    partition(arr,0,n-1);

    for(int x: arr)
        cout<<x<<" ";
}
```

OUTPUT:

1 3 2 4 8 7 5 10

Quick Sort

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

```
partition key function (naive , lomuto , hoare)
```

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Analysis of QuickSort

Time taken by QuickSort in general can be written as following.

```
T(n) = T(k) + T(n-k-1) + Θ(n)
```

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

```
 T(n) = T(0) + T(n-1) + Θ(n)
which is equivalent to
 T(n) = T(n-1) + Θ(n)
```

The solution of above recurrence is $\Theta(n^2)$.

Best Case: The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

```
 T(n) = 2T(n/2) + Θ(n)
```

The solution of above recurrence is **Θ(nLogn)**. It can be solved using case 2 of Master Theorem.

Average Case: To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.
We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

```
 T(n) = T(n/9) + T(9n/10) + Θ(n)
```

Solution of above recurrence is also O(nLogn)

Although the worst case time complexity of QuickSort is $O(n^2)$ which is more than many other sorting algorithms like Merge Sort **and** Heap Sort**, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.**

Quick Sort Using Lomuto Partition

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

int iPartition(int arr[], int l, int h)
{
    int pivot=arr[h];
    int i=l-1;
    for(int j=l;j<=h-1;j++){
        if(arr[j]<pivot){
            i++;
            swap(arr[i],arr[j]);
        }
    }
    swap(arr[i+1],arr[h]);
    return i+1;
}

void qSort(int arr[],int l,int h){
    if(l<h){
        int p=iPartition(arr,l,h);
        qSort(arr,l,p-1);
        qSort(arr,p+1,h);
    }
}

int main() {

    int arr[]={8,4,7,9,3,10,5};

    int n=sizeof(arr)/sizeof(arr[0]);

    qSort(arr,0,n-1);

    for(int x: arr)
        cout<<x<<" ";
}
```

OUTPUT:

3 4 5 7 8 9 10

Quick Sort Using Hoare Partition

```cpp
#include<iostream>
using namespace std;

int partition(int arr[], int l, int h)
{
    int pivot=arr[l];
    int i=l-1,j=h+1;
    while (true)
    {
        do{
            i++;
        }while(arr[i]<pivot);

        do{
            j--;
        }while(arr[j]>pivot);

        if(i>=j)
            return j;

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

    }
}

void qSort(int arr[], int l, int h)
{
    if(l<h)
    {
        int p=partition(arr,l,h);
        qSort(arr,l,p);
        qSort(arr,p+1,h);
    }
}

int main()
{
    int arr[]={8,4,7,9,3,10,5};

    int n=sizeof(arr)/sizeof(arr[0]);

    qSort(arr,0,n-1);

    for(int x: arr)
        cout<<x<<" ";
}
```

Tail Call Elimination in Quick Sort

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

int partition(int arr[], int l, int h)
{
    int pivot=arr[l];
    int i=l-1,j=h+1;
    while(true)
    {
        do{
            i++;
        }while(arr[i]<pivot);

        do{
            j--;
        }while(arr[j]>pivot);

        if(i>=j)
            return j;
        swap(arr[i],arr[j]);
    }
}

void qSort(int arr[], int l, int h)
{
    Begin:
    if(l<h)
    {
        int p=partition(arr,l,h);
        qSort(arr,l,p);
        l=p+1;
        goto Begin;
    }
}

int main()
{
    int arr[]={8,4,7,9,3,10,5};

    int n=sizeof(arr)/sizeof(arr[0]);

    qSort(arr,0,n-1);

    for(int x: arr)
        cout<<x<<" ";
} OUTPUT: 3 4 5 7 8 9 10
```

K^{th} Smallest Element

Naïve:

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

int kthSmallest(int arr[], int n, int k){
    sort(arr,arr+n);
    return arr[k-1];
}

int main()
{
    int arr[]={10,5,30,12};

    int n=sizeof(arr)/sizeof(arr[0]);

    int k=2;

    cout<<kthSmallest(arr,n,k);
}
```

OUTPUT:

10

Efficient Solution for K^{th} Smallest Element

```cpp

//using lomuto parition

#include<iostream>
using namespace std;

int partition(int arr[], int l,int h)
{
    int pivot=arr[h];
```

```
    int i=l-1;
    for(int j=l;j<=h-1;j++)
    {
        if(arr[j]<pivot)
        {
            i++;
            swap(arr[i],arr[j]);
        }
    }
    swap(arr[i+1],arr[h]);
    //cout<<i+1<<endl; bakichych nahi mahit pn ha element correct position la
aahe manun use kela aahe
    return i+1;
}

int kthSmallest(int arr[], int n, int k)
{
    int l=0, r=n-1;
    while(l<=r)
    {
        int p=partition(arr,l,r);
        if(p==k-1)
            return p;
        else if(p>k-1)
            r=p-1;
        else
            l=p+1;
    }

    return -1;
}

int main()
{
    int arr[]={10,80,30,90,40,50,70};

    int n=sizeof(arr)/sizeof(arr[0]);
    int k=6;

    int index=kthSmallest(arr,n,k);

    cout<<arr[index];
}
```

OUTPUT: 80

Chocolate Distribution Problem

Distribute chocolate such that difference between min and max is minimum

```cpp
#include<iostream>
#include<algorithm>
using namespace std;

int minDiff(int arr[], int n, int m)
{
    if(m>n)
        return -1;
    sort(arr,arr+n);
    int res=arr[m-1]-arr[0];
    for(int i=1;(i+m-1)<n;i++)
        res=min(res,arr[i+m-1]-arr[i]);
    return res;
}

int main()
{
    int arr[]={7,3,2,4,9,12,56};

    int n=sizeof(arr)/sizeof(arr[0]);
    int m=3;

    cout<<minDiff(arr,n,m);
}
```

OUTPUT:

2

Sort An Array With Two Types Of An Elements

Naïve:

```cpp
// a famous interview problem in which you need to segregate an array of eleme
nts containing two types of elements. The types are:

// Segregate negative and positive elements.
// Segregate even and odd elements.
// Sort a binary array.

#include<iostream>
using namespace std;

void sort(int arr[], int n)
{
    int temp[n],i=0;

    for(int j=0;j<n;j++)
    {
        if(arr[j]<0)
            temp[i++]=arr[j];
    }

    for(int j=0;j<n;j++)
    {
        if(arr[j]>=0)
            temp[i++]=arr[j];
    }

    for(int j=0;j<n;j++)
        arr[j]=temp[j];
}

int main() {

    int arr[]={13,-12,18,-10};

    int n=sizeof(arr)/sizeof(arr[0]);

    sort(arr,n);

    for(int x:arr)
        cout<<x<<" ";
}
```

OUTPUT: -12 -10 13 18

Efficient Solution for Sort an Array with Two Types of Elements

```
// a famous interview problem in which you need to segregate an array of eleme
nts containing two types of elements. The types are:

// Segregate negative and positive elements.
// Segregate even and odd elements.
// Sort a binary array.

//time complexity theta(n) and space complexity theta(1)
//we can also use lomuto parition

//using haore parition

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

void sort(int arr[], int n)
{
    int i=-1,j=n;
    while(true)
    {
        do{i++;}while(arr[i]<0);
        do{j--;}while(arr[j]>=0);
        if(i>=j) return;
        swap(arr[i],arr[j]);
    }
}

int main()
{
    int arr[]={13,-12,0,18,-10};

    int n=sizeof(arr)/sizeof(arr[0]);

    sort(arr,n);

    for(int x : arr)
        cout<<x<<" ";
}
```

OUTPUT:

-10 -12 0 18 13

Sort An Array With Three Types Of An Elements

Naïve:

```
//  a famous interview problem in which you need to segregate an array of elem
ents containing three types of elements. The types are:

// Sort an array of 0s, 1s, 2s.
// Three-way partitioning when multiple occurrences of a pivot may exist.
// Partitioning around a range.  eg [2,5]

// here we traverse four times

#include<iostream>
using namespace std;

void sort(int arr[], int n)
{
    int temp[n],i=0;

    for(int j=0;j<n;j++)
        if(arr[j]==0)
            temp[i++]=arr[j];


    for(int j;j<n;j++)
        if(arr[j]==1)
            temp[i++]=arr[j];

    for(int j=0;j<n;j++)
        if(arr[j]==2)
            temp[i++]=arr[j];

    for(int j=0;j<n;j++)
        arr[j]=temp[j];

}
int main() {

    int arr[]={0,1,1,2,0,1,1,2};

    int n=sizeof(arr)/sizeof(arr[0]);

    sort(arr,n);

    for(int x:arr)
        cout<<x<<" ";
}   OUTPUT:  0 0 1 1 1 1 2 2
```

Efficient Solution for Sort an Array with Three Types of Elements

```
//here we  use dutch national flag algorithm
// here we traverse only  one times
// time complexity O(n) and space complexity is O(1) because we are not use ex
tra space

#include<iostream>
using namespace std;

void sort(int arr[],int n)
{
    int l=0,h=n-1,mid=0;
    while(mid<=h)
    {
        switch(arr[mid])
        {
            case 0:
                swap(arr[l],arr[mid]);
                l++;
                mid++;
                break;

            case 1:
                mid++;
                break;

            case 2:
                swap(arr[h],arr[mid]);
                h--;
                break;
        }
    }
}

int main() {

    int arr[]={0,1,1,2,0,1,1,2};

    int n=sizeof(arr)/sizeof(arr[0]);

    sort(arr,n);

    for(int x:arr)
        cout<<x<<" ";
}
OUTPUT:  0 0 1 1 1 1 2 2
```

Minimum Difference in array

Naïve:

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

int getMinDiff(int arr[], int n){
    int res = INT_MAX;
    for(int i = 1; i < n; i++){
        for(int j = 0;j<i;j++){
            res = min(res, abs(arr[i] - arr[j]));
            //The function converts negative numbers to positive numbers while
 positive numbers remain unaffected.
        }
    }
    return res;
}

int main() {

    int n;
    cin>>n;
    int arr[n];
    for(int i = 0; i < n; i++){
        cin>>arr[i];
    }

    cout<<getMinDiff(arr, n );
    return 0;
}
```

OUTPUT:

3

5

6

3

1

Efficient Solution Minimum Difference in Array

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

int getMinDiff(int arr[],int n)
{
    sort(arr,arr+n);

    int res=INT32_MAX;
    for(int i=1;i<n;i++)
        res=min(res,arr[i]-arr[i-1]);
    return res;
}

int main() {

    int n;
    cin>>n;
    int arr[n];
    for(int i = 0; i < n; i++){
        cin>>arr[i];
    }

    cout<<getMinDiff(arr, n );
    return 0;
}
```

OUTPUT:

4

6

8

3

9

1

Merge Overlapping Interval

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

struct Interval
{
    int s,e;
};

bool mycomp(Interval a, Interval b)
{
    return a.s<b.s;
}

void mergeIntervals(Interval arr[], int n)
{
    sort(arr, arr+n,mycomp);

    int res=0;

    for(int i=1;i<n;i++)
    {
        if(arr[res].e>=arr[i].s)
        {
            arr[res].e=max(arr[res].e,arr[i].e);
            arr[res].s=min(arr[res].s,arr[i].s);
        }
        else
        {
            res++;
            arr[res]=arr[i];
        }
    }
    for(int i=0;i<=res;i++)
        cout<<"["<<arr[i].s<<","<<arr[i].e<<"]";
}

int main()
{
    Interval arr[]={ {5,10},{3,15},{18,30},{2,7}};
    int n=sizeof(arr)/sizeof(arr[0]);
    mergeIntervals(arr,n);

    return 0;
}
```

OUTPUT: [2,15][18,30]

Meeting the Maximum Guest

you are given arrival and departure times of the guests, you need to find the time to go to the pary so that you can meet maximum people

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

int maxGuest(int arr[],int dep[], int n)
{
    sort(arr, arr+n);
    sort(dep, dep+n);

    int i=1,j=0,res=1,curr=1;
    while(i<n && j<n)
    {
        if(arr[i]<dep[j])
        {
            curr++;
            i++;
        }

        else{
            curr--;
            j++;
        }
        res=max(curr,res);
    }
    return res;
}

int main()
{
    int arr[]={900, 600, 700};
    int dep[]={1000, 800, 730};
    int n=sizeof(arr)/sizeof(arr[0]);

    cout<<maxGuest(arr, dep, n);

    return 0;
}
```

OUTPUT: 2

Cycle Sort

```cpp
//a worst case O(n^2) sorting algorithm
//does minimum memory writes and can be useful for cases where  memory in  wri
te in costly
//in place and not stable
//useful to solve qustion like find minimum swaps required to sort an array

#include<iostream>
using namespace std;

void cycleSortDistinct(int arr[], int n)
{
    for(int cs=0;cs<n-1;cs++)
// jo first elemetn aahe taypeksha small kiti element aahe te count karaych aa
hi to element taychy
    {                                   // correct position la fix karaycha
        int item=arr[cs];
        int pos=cs;
        for(int i=cs+1;i<n;i++)
            if(arr[i]<item)
                pos++;
        swap(item,arr[pos]);
        while (pos!=cs)      // jo paryant first element taychy position la ye
t nahi to paryant hi process chalete
        {
            pos=cs;
            for(int i=cs+1;i<n;i++)
                if(arr[i]<item)
                    pos++;
            swap(item,arr[pos]);
        }
    }
}

int main()
{
    int arr[] = { 20,40,50,10,30 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cycleSortDistinct(arr, n);

    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    return 0;
}
```

OUTPUT: 10 20 30 40 50

Heap Sort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining elements.

What is Binary Heap?

Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible (Source Wikipedia).

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

Array based representation for Binary Heap: Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).

Heap Sort Algorithm for sorting an array in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

Important Notes:

- Heap sort is an in-place algorithm.
- Its typical implementation is not stable, but can be made stable (See this).

Time Complexity: Time complexity of heapify is O(N*LogN). Time complexity of createAndBuildHeap() is O(N) and overall time complexity of Heap Sort is **O(N*LogN)** where N is the number of elements in the list or array.

Heap sort algorithm has limited use because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used.

```cpp
//1-build heap
//2-delete data from heap

//sort array incresing order use max heap
//sort array decresing order use min heap
#include<iostream>
using namespace std;

void heapify(int arr[], int n, int i)
{
    int largest=i;
    int l=2*i+1;
    int r=2*i+2;
    if(l<n && arr[l]>arr[largest])
        largest=l;

    if(r<n && arr[r]>arr[largest])
        largest=r;

    if(largest!=i)
    {
        swap(arr[i],arr[largest]);
        heapify(arr,n,largest);
    }
}

void buildheap(int arr[],int n)
{
    for(int i=n/2-1;i>=0;i--)
        heapify(arr,n,i);
}

void heapSort(int arr[], int n)
{
    buildheap(arr,n);

    for(int i=n-1;i>0;i--)
    {
        swap(arr[0],arr[i]);
        heapify(arr,i,0);
    }
}

void printArray(int arr[], int n)
{
    for(int i=0;i<n;++i)
        cout<<arr[i]<<" ";
    cout<<"\n";
```

```
}

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

    heapSort(arr,n);

    cout<<"Sorted array is \n";
    printArray(arr,n);
}
```

OUTPUT :

Sorted array is

5 6 7 11 12 13

```
// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2*i + 1; // left = 2*i + 1
    int r = 2*i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i)
    {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}
// Main function for heap sort
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i=n-1; i>=0; i--)
    {
        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

Counting Sort

Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.

Let us understand it with the help of an example.

```
For simplicity, consider the data in the range 0 to 9.
Input data: 1, 4, 1, 2, 7, 5, 2
  1) Take a count array to store the count of each unique object.
  Index:     0  1  2  3  4  5  6  7  8  9
  Count:     0  2  2  0   1  1  0  1  0  0

  2) Modify the count array such that each element at each index
  stores the sum of previous counts.
  Index:     0  1  2  3  4  5  6  7  8  9
  Count:     0  2  4  4  5  6  6  7  7  7

The modified count array indicates the position of each object in
the output sequence.

  3) Output each object from the input sequence followed by
  decreasing its count by 1.
  Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.
  Put data 1 at index 2 in output. Decrease count by 1 to place
  next data 1 at an index 1 smaller than this index.
```

Time Complexity: **O(N + K) where N is the number of elements in input array and K is the range of input.**
Auxiliary Space: **O(N + K)**

The problem with the previous counting sort was that it could not sort the elements if we have negative numbers in the array because there are no negative array indices. So what we can do is, we can find the minimum element and store count of that minimum element at zero index.

Important Points:

1. Counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted. Consider the situation where the input sequence is between range 1 to 10K and the data is 10, 5, 10K, 5K.
2. It is not a comparison based sorting. It's running time complexity is O(n) with space proportional to the range of data.

3. It is often used as a sub-routine to another sorting algorithm like radix sort.
4. Counting sort uses a partial hashing to count the occurrence of the data object in O(1).
5. Counting sort can be extended to work for negative inputs also.

Naïve:

```
#include<iostream>
using namespace std;

void countSort(int arr[], int n, int k)
{
    int count[k];
    for(int i=0;i<k;i++)
        count[i]=0;
    for(int i=0;i<n;i++)
        count[arr[i]]++;

    int index=0;
    for(int i=0;i<k;i++)
    {
        for(int j=0;j<count[i];j++)
        {
            arr[index]=i;
            index++;
        }
    }
}
int main()
{
    int arr[] = { 1,4,4,1,0,1 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int k=5;
    countSort(arr, n, k);

    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    return 0;
}
```

OUTPUT: 0 1 1 1 4 4

Efficient Solution For Counting Sort :

```cpp
//not a comparisons based algorithm
//theta(n+k) time complexity
// theta(n+k) auxiliary space
// stable
//used as a subroutine in radix sort
//counting sort is useful when k is really really small

#include<iostream>
using namespace std;

void countSort(int arr[], int n, int k)
{
    int count[k];
    for(int i=0;i<k;i++)
        count[i]=0;
    for(int i=0;i<n;i++)
        count[arr[i]]++;

    for(int i=1;i<k;i++)
        count[i]=count[i-1]+count[i];

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

    for(int i=0;i<n;i++)
        arr[i]=output[i];
}

int main()
{
    int arr[] = { 1,4,4,1,0,1 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int k=5;
    countSort(arr, n, k);

    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    return 0;
}
```

OUTPUT : 0 1 1 1 4 4

Radix Sort

```cpp
#include<iostream>
using namespace std;

void countingSort(int arr[], int n, int exp)
{
    int output[n];
    int count[10]={0};
    for(int i=0;i<n;i++)
        count[(arr[i]/exp) % 10]++;

    for(int i=1;i<10;i++)
        count[i]+=count[i-1];

    for(int i=n-1;i>=0;i--)
    {
        output[count[(arr[i]/exp) %10]-1]=arr[i];
        count[(arr[i]/exp) % 10]--;
    }

    for(int i=0;i<n;i++)
        arr[i]=output[i];
}

void radixSort(int arr[], int n)
{
    int mx=arr[0];
    for(int i=1;i<n;i++)
        if(arr[i]>mx)
            mx=arr[i];

    for(int exp=1; mx/exp>0;exp*=10)
        countingSort(arr,n,exp);
}
int main()
{
    int arr[] = { 319,212,6,8,100,50 };
    int n = sizeof(arr) / sizeof(arr[0]);
    radixSort(arr, n);

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

OUTPUT : 6 8 50 100 212 319

Bucket Sort

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

void bucketSort(int arr[], int n, int k)
{
    int max_val=arr[0];
    for(int i=1;i<n;i++)
        max_val=max(max_val,arr[i]);

    max_val++;
    vector<int> bkt[k];

    for (int i = 0; i < n; i++) {
        int bi = (k * arr[i])/max_val;
        bkt[bi].push_back(arr[i]);
    }

    for (int i = 0; i < k; i++)
        sort(bkt[i].begin(), bkt[i].end());

    int index = 0;
    for (int i = 0; i < k; i++)
        for (int j = 0; j < bkt[i].size(); j++)
            arr[index++] = bkt[i][j];
}



int main()
{
    int arr[] = { 30,40,10,80,5,12,70 };
    int n = sizeof(arr) / sizeof(arr[0]); int k=4;
    bucketSort(arr, n, k);

    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    return 0;
}
```

OUTPUT : 5 10 12 30 40 70 80