Sorting

Sorting

Introduction

Sorting refers to arranging data in a particular format. The sorting algorithm specifies the way to organize data in a specific order, the most common being numerical or lexicographical order. The importance of sorting lies in the fact that data searching can be optimized to a very high level if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats.

We have the inbuilt **sort()** function in most languages, which indirectly sorts the array or vector. Many sorting techniques are available with different space and time complexities; every sorting technique has advantages and disadvantages. The fastest sorting technique can sort the entire array in just **O(N*log(N)),** where **N** is the number of elements in the array.

There are two types of sorting algorithms.

(1) Stable sorting algorithms

The sorting algorithm in which the order of the same element remains identical is termed a stable sorting algorithm.

E.g., Counting sort, Merge sort.

(2) Unstable sorting algorithms

The sorting algorithm in which the order of the same element does not matter is termed an unstable sorting algorithm.

E.g., quicksort.

In this module, we will be discussing the following sorting techniques.

- Merge sort.
- Quick sort
- Count sort

Merge Sort

Merge sort works on the principle of divide and conquer. Divide and conquer divides the problem into smaller halves, then tries to find a solution for those smaller halves. After that, combine the solution of those smaller halves to find a solution for the whole problem.

Problem statement:

The problem statement says that you are given an array of **N**, integers and you have to sort the elements present in the array using the merge sort algorithm.

Algorithm:

1. Divide the unsorted array into **N** subarrays.
2. Calculate the middle index element by the formula **(l + r)/2** .(keep on calculating until all the elements in the array are sorted)
3. Repeatedly merge subarrays to produce newly sorted subarrays until there is only 1 subarray remaining. This will be the sorted array.

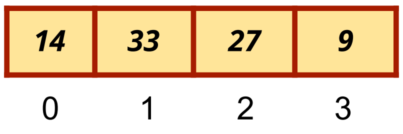
Approach:

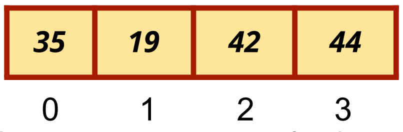
- Let us consider the array to be:

  arr = {14, 33, 27, 9, 35, 19, 42, 44}

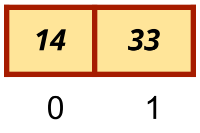
14	***33***	***27***	***9***	***35***	***19***	***42***	***44***
0	1	2	3	4	5	6	7

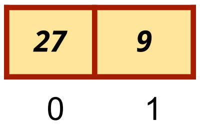
- We are aware of the phenomenon that merges sort divides the whole array into equal halves until the principle of atomicity (when there is only one element left at the end, and it cannot be divided further) is achieved. We can observe that our array consists of 8 elements, and hence we divide the whole array into 2 sub-arrays, each array consisting of 4 elements.
- So now, our array will become **arr1 = {14, 33, 27, 9}** and **arr2 = {35, 19, 42, 44}.**

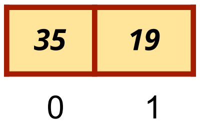


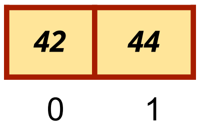


- The arrays obtained in the previous step, we further subdivide them into equal halves so now, this array would become **arr1 = { 14, 33}, arr2 = { 27, 9},arr3 = { 35, 19} & arr4 = { 42, 44}**

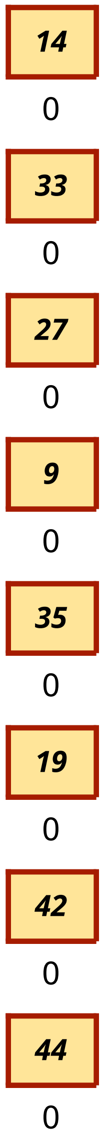






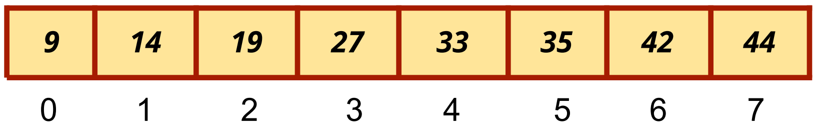


- Still the principle of atomicity is not achieved yet, so we further divide the arrays obtained in the above step.Now the array would become **arr1 = {14}, arr2 = {33}, arr3 = {27}, arr4 = {9}, arr5 = {35}, arr6 = {19},arr7 = {42} and arr8 = {44}**



- In the previous step, we can observe that the principle of atomicity is achieved, i.e all the arrays cannot be divided further. So now we can combine them.
- We will first compare the element for both the lists and then insert them into another list in a sorted manner.
- We observe that elements 14 & 33 are already sorted. We compare elements 27 and 9 and in the target list of 2 values, we insert 9 first, followed by 27. We change the order of elements 19 and 35 whereas 42 and 44 are placed sequentially.

- In the next iteration, we compare lists of two data values and merge them into a list of found data values, placing them all in sorted order.
- After final merging our array will look like **arr = { 9, 14, 19, 27, 33, 35, 42, 44}**



Code:

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

int merge(vector<int>& a, int start, int mid, int end){
    vector<int>left,right;
    for(int i=start;i<=mid;i++) left.push_back(a[i]);
    for(int i=mid+1;i<end;i++) right.push_back(a[i]);

    int n=left.size();
    int m=right.size();
    int left_ptr=0;
    int right_ptr=0;
    int curr=start;
    while(left_ptr<n && right_ptr<m){
        if(left[left_ptr]<right[right_ptr]){
            a[curr] = left[left_ptr];
            curr++;
            left_ptr++;
        }
        else{
            a[curr] = right[right_ptr];
            curr++;
            right_ptr++;
        }
    }
    while(left_ptr<n){
        a[curr] = left[left_ptr];
        curr++;
        left_ptr++;
    }
    while(right_ptr<m){
        a[curr] = right[right_ptr];
        curr++;
        right_ptr++;
    }
}

void MergeSort(vector<int>& a, int start, int end){
    if(start>=end) return;
    int mid = (start+end)/2;
    MergeSort(a,start,mid);
    MergeSort(a,mid+1,end);
    merge(a,start,mid,end);
}

void solve(){
    int n;
    cin>>n;
    vector<int>a(n);
    for(int i=0;i<n;i++) cin>>a[i];
    MergeSort(a,0,n-1);
    for(int i=0;i<n;i++) cout<<a[i]<<" ";
}
```

Time-complexity: O[N*(log N))] (In all cases) where **N** is the number of elements in the array.

Space complexity: O(N) where **N** is the number of elements in the array.

Advantages:

- Merge sort is faster for large lists because it doesn't traverse the whole list since the divide and conquer approach is followed in merge sort.
- Running time is consistent.

Disadvantages:

- The algorithm does the whole process even the array is already sorted.
- Utilizes more memory space.

Quick Sort

In quick sort, an element is chosen as a pivot element, and then partitions are made on the given array based on the chosen pivot element. In quick sort, the whole array is divided into two sub-arrays in which one subarray consists of values that are smaller than the specified value **(Pivot)** and the other sub-array consists of values that are greater than the specified value **(pivot)**.

Problem statement:

The problem statement says that you are given an array of **N** integers and you have to sort the elements present in the array using the quick sort algorithm.

Approach:

Select an element as a pivot element and partition the array based on the pivot element.

Algorithm:

- The selection of a pivot element can be random. i.e., any element present in the array can be chosen as a pivot element.
- We can select the median as the pivot element.
- The element present on the left of the element present in the right can be chosen as a pivot element.

Code:

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

int Partition(vector<int>& a, int left , int right){
    int pivot = left;
    int end = a[right];

    for(int i=left;i<right;i++){
        if(a[i]<=end){
            swap(a[pivot],a[i]);
            pivot++;
        }
    }

    swap(a[pivot],a[right]);
    return pivot;
}

void QuickSort(vector<int>& a, int left , int right){
    if(left>=right) return;
    int pivot = Partition(a,left,right);
    QuickSort(a,left,pivot-1);
    QuickSort(a,pivot+1,right);
}


void solve(){
    int n;
    cin>>n;
    vector<int>a(n);
    for(int i=0;i<n;i++) cin>>a[i];
    QuickSort(a,0,n-1);
    for(int i=0;i<n;i++) cout<<a[i]<<" ";
}
```

Time-complexity: N is the number of elements in the array.

1. **Worst case: O(N^2)**
2. **Average case: O(N*log(N))**
3. **Best case: O(N)**

Space complexity: O(log N), where **N** is the number of elements in the array.

Advantages:

- It is in-place since it uses only a small auxiliary stack.
- It requires only n (log n) time to sort n items.
- It has an extremely short inner loop.

Disadvantages:

- It is recursive. Especially if recursion is not available, the implementation is highly complicated.
- It requires quadratic time **O(N)** in the worst-case scenario.

Counting Sort

Problem statement:

The problem statement says that you are given an array of **N**, integers and you have to sort the elements present in the array using the counting sort algorithm.

Algorithm:

- First of all, we find the maximum **max** and minimum **min** elements from the whole array.
- After finding the minimum and maximum elements we create a frequency array from index **min** to **max**.
- In the frequency array, store the count of every element which you encounter while iterating over an array.

Approach:

Consider an array i.e arr = {2, 3, 1, 1, 5, 4, 2, 1}

2	*3*	*1*	*1*	*5*	*4*	*2*	*1*
0	1	2	3	4	5	6	7

- Firstly, we find the maximum and minimum elements from your array, which **max=5** and **min**=**1,** respectively.
- Create another array that will store the frequency of elements present in your original array of size **max-min+1(5-1+1=5)**.

- Once a frequency array is created, initialize all the indexes of the frequency array with 0. i.e. **frequency_array = {0, 0, 0, 0, 0};**

0	*0*	*0*	*0*	*0*
0	1	2	3	4

- Then we Iterate in the original array from left to right. While iterating from left to right for the first time, element 2 is encountered. So we increment the count at index 2 from 0 to 1 in our frequency array.

- Similarly, elements **{3, 1, 1, 5, 4, 2, 1}** are encountered in the given order and, we increment the value at the indices of the **frequency_array** accordingly.

- After the above process, our **frequencey_ array** would be equal to: **{3, 2, 1, 1, 1}.**

3	*2*	*1*	*1*	*1*
0	1	2	3	4

- In the end, we traverse our **frequency_array** and print the index as many times as the value stored in the **frequency_array** corresponds to that index.
- Our final output would be equal to: **{1, 1, 1, 2, 2, 3, 4, 5}.**

1	*1*	*1*	*2*	*2*	*3*	*4*	*5*
0	1	2	3	4	5	6	7

Code:

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


void solve(){
    int n;
    cin>>n;
    vector<int>a(n);
    for(int i=0;i<n;i++) cin>>a[i];

    int maxi = *max_element(a.begin(),a.end());
    int mini = *min_element(a.begin(),a.end());
    vector<int>f(maxi-mini+2);

    for(int i=0;i<n;i++){
        f[a[i]-mini]++;
    }

    for(int i=0;i<f.size();i++){
        for(int j=0;j<f[i];j++){
            cout<<(i+mini)<<" ";
        }
    }

}
```

Time-complexity: O(N) where **N** is the size of the array.

Space complexity: O(N + K) where **N i**s the size of the array and **K** is the maximum element present in the array.

Advantages:

- Faster execution when compared with other sorting algorithms.
- Easy to implement.

Disadvantages:

- This sorting technique will prove costly when there are large numbers present in the array.
- Not feasible for fractional values.
- Not helpful for negative indexes.

Finding the kth largest and smallest element

Problem Statement:

You are given an array, and you have to find out the kth smallest and largest element present in an array.
The array given to you is **arr =[4, 5, 1, 2, 3]**

4	***5***	***1***	***2***	***3***
0	1	2	3	4

We need to find the 2nd largest and 2nd smallest elements from the above-given array.

Approach:
We would be using the quick sort technique to solve this problem.

Algorithm:

- In quick sort, we first choose the pivot element. In our case, we will select the last element (i.e., element 3) present in the array as the pivot element.
- We will place all the elements smaller than the pivot element on the left-hand side of the pivot element, and we will place all the elements greater than the pivot element on the right-hand side of the pivot element.
- So, in our array, we can observe that elements 1 and 2 are less than the pivot element; therefore it will be placed on the left-hand side, and elements 4 and 5 are greater than the pivot element; hence they would be placed on the right-hand side of the pivot element.
- Hence, after performing all the above-mentioned steps, our array will now look like **arr = [1, 2, 3, 4, 5].**

1	***2***	***3***	***4***	***5***
0	1	2	3	4

Looking for the 2nd smallest number, we can conclude that the 2nd smallest number would be present on the left-hand side. We can observe that all the elements on the left side of the pivot element are smaller than 3. Hence we ignore all the elements present on the right-hand side of the pivot element and perform a quick sort on all the elements present on the left-hand side of the pivot element.

On the left-hand side of the pivot element, we have elements 1 and 2, and we observe that it is already sorted. Element 2 is present at the 2nd position, and hence we conclude that the 2nd smallest element present in the array is 2. For the 2nd largest element, we can say that we are looking for the 2nd element from the end of the array. In other words, we can also say that we are looking for the 4th smallest element. This whole scenario can be represented as n – k + 1, where n = number of elements. i.e., in the solution, we need the “kth” smallest number and the “n – k + 1” smallest number because n – k + 1 is the kth largest number.

This whole algorithm is called a quick-select function because we are not sorting the entire left part and right part of the array; we are performing sorting only on the half or the required part from the array (only that part of the array would be sorted where the answer lies).

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

int Partition(vector<int>& a, int left , int right){
    int pivot=left;
    int end = a[right];

    for(int i=left;i<right;i++){
        if(a[i]<=end){
            swap(a[i],a[pivot]);
            pivot++;
        }
    }
    swap(a[right],a[pivot]);
    return pivot;
}
```

```
int QuickSelect(vector<int>a, int left , int right , int k){
    // base condition
    if(left>=right) return a[left];
    // calculating the pivot element correct position
    int pivot = Partition(a,left,right);
    //checking if pivot element is the kth smallest element
    if(pivot+1 == k) return a[pivot];
    // otherwise look for left or right , depending on k
    else if(pivot+1 > k) return QuickSelect(a,left,pivot-1,k);
    return QuickSelect(a,pivot+1,right,k);
}


void solve(){
    //taking input
    int n,k;
    cin>>n>>k;
    vector<int>a(n);
    for(int i=0;i<n;i++) cin>>a[i];

    //calling Quickselect to print the kth smallest number &
    // kth largest(or n-k+1 th smallest) number
    cout<<QuickSelect(a,0,n-1,k)<<" "<<QuickSelect(a,0,n-1,n-k+1);
    cout<<"\n";
}
```

Time Complexity: **N** is the size of the array.

O(N^2) → Worst case,

O(N) → Average case.

Count Number Of Inversions

Problem Statement :

For a given integer array/list 'ARR' of size 'N', find the total number of 'Inversions' that may exist.

An inversion is defined as a pair of integers in the array/list when the following two conditions are met.

A pair ('ARR[i]', 'ARR[j]') is said to be an inversion when:

1. **'ARR[i] > 'ARR[j]'**
2. **'i' < 'j'**

Where 'i' and 'j' denote the indices ranging from **[0, 'N').**

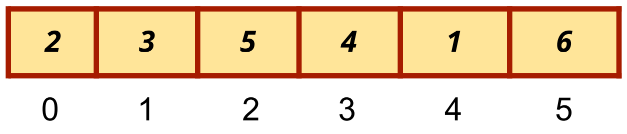
Approach:

(a) Brute force approach

- Take every number and find out the number of elements that are present before it and are larger than the current element, which we are talking about.

(b) Optimized approach

Consider the given array as **arr = [2, 3, 5, 4, 1, 6]**



And we need to find the number of inversions for element **4**.

- First, let us sort all the elements present in the first half. I.e, elements **2, 3, 5**.
- Secondly, let us sort all the elements present in the second half. I.e, 1, 6.

We perform the process of sorting and observe at which step we can count the number of inversions.

Code:

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

```
long long merge(vector<int>& a, int start, int mid, int end){
    vector<int>left,right;
    for(int i=start;i<=mid;i++) left.push_back(a[i]);
    for(int i=mid+1;i<=end;i++) right.push_back(a[i]);

    long long total=0;
    int n=left.size();
    int m=right.size();
    int left_ptr=0;
    int right_ptr=0;
    int curr=start;

    while(left_ptr<n && right_ptr<m){
        if(left[left_ptr]<=right[right_ptr]){
            a[curr] = left[left_ptr];
            curr++;
            left_ptr++;
        }
        else{
            a[curr] = right[right_ptr];
            total+=(n-left_ptr);
            curr++;
            right_ptr++;
        }
    }
    while(left_ptr<n){
        a[curr] = left[left_ptr];
        curr++;
        left_ptr++;
    }
    while(right_ptr<m){
        a[curr] = right[right_ptr];
        curr++;
        right_ptr++;
    }
    return total;
}

long long countInversion(vector<int>& a, int start, int end){
    if(start>=end) return 0;
    int mid = (start+end)/2;
    long long total=0;
    total+=countInversion(a,start,mid);
    total+=countInversion(a,mid+1,end);
    total+=merge(a,start,mid,end);
    return total;
}


void solve(){
    int n;
    cin>>n;
    vector<int>a(n);
    for(int i=0;i<n;i++) cin>>a[i];
    cout<<countInversion(a,0,n-1)<<"\n";
}
```