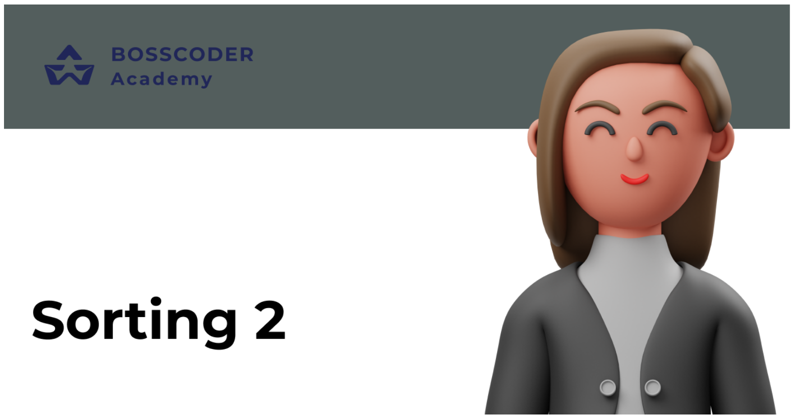
BOSSCODER ACADEMY UPSKILL WITH US

Sorting 2

Gyanu Mayank

17th February, 2023



Agenda
1. Problems based on merge sort
2. QuickSort and Problems related to it
3. Advantages of quick sort over merge sort

Problems

1. **Inversion Count of Array**
   Given an array **a[]**. The task is to find the inversion count of **a[]**. Where two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j.
   Example
   Input : N = 5, array[] = {1,2,3,4,5}
   Output: 0
   Brute Force Approach
   We can use two loops and see if a[i]>a[j]

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

int main()
{
  int n;
  cin >> n;
  int arr[n];
  for (int i = 0; i < n; i++)
    cin >> arr[i];
  int cnt = 0;
  for (int i = 0; i < n; i++)
  {
    for (int j = i + 1; j < n; j++)
    {
      if (arr[i] > arr[j])
        cnt++;
    }
  }
  cout << cnt << endl;
}
```

Optimised Approach

if we do it by brute force, it will cost O(n^2) time complexity because we have to add two nested loops to check the 2nd condition, but if we have the sorted array, then it could be easier to get the answer.If an array is sorted, the array inversions are 0, and if reverse sorted, the array inversions are maximum.To sort the array, we will use mergeSort. In mergeSort, we will deep in one element, compare two elements, and put it in a new array in sorted order. By doing this for log(N) time, we will get code such that it will count the inversion of the smaller array and slowly it will count for larger and larger array.The single element is always sorted after slicing to the bottom and getting them on an element as an array. Before returning the merged array with sorted numbers, we will count the inversion from there.1st condition i < j above in the image,you can see that the right element's index is always greater, so while computing the inversion, we should take care only 2nd condition, which is if i < j then A[j] < A[i] to make a pair and add one to the count.

Implementation

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

int merge(int arr[], int temp[], int left,
{
  int inv_count = 0;
  int i = left;
  int j = mid;
  int k = left;
  while ((i <= mid - 1) && (j <= right))
  {
    if (arr[i] <= arr[j])
    {
      temp[k++] = arr[i++];
    }
    else
    {
      temp[k++] = arr[j++];
      inv_count = inv_count + (mid - i);
  while (i <= mid - 1)
    temp[k++] = arr[i++];

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

  for (i = left; i <= right; i++)
    arr[i] = temp[i];

  return inv_count;
}

int merge_Sort(int arr[], int temp[], int l
{
  int mid, inv_count = 0;
  if (right > left)
  {
    mid = (left + right) / 2;

    inv_count += merge_Sort(arr, temp, left
    inv_count += merge_Sort(arr, temp, mid

    inv_count += merge(arr, temp, left, mic
  }
  return inv_count;
}

int main()
{
  int n;
  cin >> n;
  int arr[n];
  for (int i = 0; i < n; i++)
    cin >> arr[i];
  cout << ans << endl;

  return 0;
}
```

Time Complexity: O(n * log n)
Space Complexity: O(n)

QuickSort
QuickSort is an in-place sorting algorithm with worst-case time complexity of o(n^2).QuickSort can be implemented iteratively and recursively.
The algorithm can be broken down into 3 parts.
1. Partitioning the array about the pivot.
2. Passing the smaller arrays to the recursive calls.
3. Joining the sorted arrays that are returned from the recursive call, and the pivot.

Implementation

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

void swap(int arr[] , int pos1, int pos2){
        int temp;
        temp = arr[pos1];
        arr[pos1] = arr[pos2];
        arr[pos2] = temp;
}

int partition(int arr[], int low, int high,
        int i = low;
```

```
                if(arr[i] > pivot){
                        i++;
                }
                else{
                        swap(arr,i,j);
                        i++;
                        j++;
                }
        }
        return j-1;
}

void quickSort(int arr[], int low, int high
        if(low < high){
        int pivot = arr[high];
        int pos = partition(arr, low, high,

        quickSort(arr, low, pos-1);
        quickSort(arr, pos+1, high);
        }
}

int main()
{
        int n ;
        cin>>n;
        int arr[n];
        for( int i = 0 ; i < n; i++){
                cin>> arr[i];
        }
        quickSort(arr, 0 , n-1);
        cout<<"The sorted array is: ";
        for( int i = 0 ; i < n; i++){
                cout<< arr[i]<<"\t";
        }
```

The running time complexity of quicksort for the best case and the average case is **O(N log N)**. Whereas the time complexity is for the worst case is **O(N2)**. Coming to the space complexity, since the quick sort algorithm doesn't require any additional space other than that to store the original array, therefore, the space complexity of the quick sort algorithm is **O(N)**. N is the size of the input array.

Advantages of Quicksort
1. The average-case time complexity to sort an array of n elements is O(n log n).
2. Generally, it runs very fast. It is even faster than merge sort.
3. No extra storage is required.

Problem Based on Quick Sort

1. **Lock and Key Problem**
   Given a box with locks and keys where one lock can be opened by one key in the box. We need to match the pair.
   Example
   Input:
   The lists of locks and keys.
   nuts = {),@,*,^,(,%, !,$,&,#}
   bolts = { !, (, #, %,), ^, &, *, $, @ }
   Output:
   After matching nuts and bolts:

Approach

This problem is solved by the quick-sort technique. By taking the last element of the bolt as a pivot, rearrange the nuts list and get the final position of the nut whose bolt is the pivot element. After partitioning the nuts list, we can partition the bolts list using the selected nut. The same tasks are performed for left and right sub-lists to get all matches.

Implementation

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

void show(char array[], int n)
{
  for (int i = 0; i < n; i++)
    cout << array[i] << " ";
}

int partition(char array[], int low, int hi
{
  int i = low;
  for (int j = low; j < high; j++)
  {
    if (array[j] < pivot)
    {
      swap(array[i], array[j]);
      i++;
    }
    else if (array[j] == pivot)
    {
      swap(array[j], array[high]);
  }
  swap(array[i], array[high]);
  return i;
}

void nutAndBoltMatch(char nuts[], char bolt
{
  if (low < high)
  {
    int pivotLoc = partition(nuts, low, hig
    partition(bolts, low, high, nuts[pivotL
    nutAndBoltMatch(nuts, bolts, low, pivot
    nutAndBoltMatch(nuts, bolts, pivotLoc +
  }
}

int main()
{
  int n;
  cin >> n;
  char nuts[n], bolts[n];
  for (int i = 0; i < n; i++)
    cin >> nuts[i];
  for (int i = 0; i < n; i++)
    cin >> bolts[i];
  nutAndBoltMatch(nuts, bolts, 0, n - 1);
  cout << "After matching nuts and bolts:"
  cout << "Nuts:  ";
  show(nuts, n);
  cout << endl;
  cout << "Bolts: ";
  show(bolts, n);
  cout << endl;
}
```

BOSSCODER ACADEMY

UPSKILL WITH US

Bosscoder had everything I was looking for

Read

Date: 12th May, 2023

"Bosscoder helped me clear my basics of DSA and System Design"
Rachit Arora
Head of Engineering
SOPTLE

Bosscoder helped me clear my basics of DSA and System Design

Read

Date: 5th May, 2023

View All blogs

BOSSCODER ACADEMY

Helping ambitious learners upskill themselves & shift to top product

Who are we

About us
Blog
Attend a FREE Event
Privacy Policy
Terms & Condition
Pricing and Refund Policy

Contact Us

Email: ask@bosscoderacademy.com

Follow us on

LinkedIn
Youtube
Instagram
Telegram
Reviews on Quora

all about
it.