10 Sorting

01 May 2024 12:04

Agenda :
1. Introduction
2. Problems
3. Basic Sorting Algorithms

1. Introduction

- Sorting is an arrangement of data in particular order on the basis of some parameter

  Example :

  A[ ] = { 2, 3, 9, 12, 17, 19 }
  Above Array is sorted in ascending order.

2. Problems

Problem 1 : Minimize the cost to empty an array.
Given an array of n integers, minimize the cost to empty given array where cost of removing an element is equal to sum of all elements left in an array.

Example :
A[ ] = { 2, 1, 4 }
Ans = 11
After removing 4 cost = 4+2+1 = 7
After removing 2 cost = 2+1 = 3
After removing 1 cost = 1 = 1
Comment
Suggest edit

Total cost = 11

Approach :

Start Removing from the largest element. i.e removing largest element sequentially will result in lowest cost.

[a b c d]

Remove a   a + b + c + d

Remove b   + b + c + d

Remove c   + c + d

Remove d   + d

a + 2b + 3c + 4d

a > b > c > d

- Sort the data in descending order.
- Initialise the ans equal to 0.
- Run a loop for i from 0 to n – 1, where n is the size of the array.
- For every element add arr[i]*i to the ans.
  – TC - O(nlogn)
  – SC - O(n)

Problem 2 : Find Count of Nobel Integers

Given an array of distinct elements of size n, find the count of noble integers.

Note: arr[i] is noble if count of elements smaller than arr[i] is equal to arr[i] where arr[i] is element at index i.

Example :

A[ ] = { 1, -5, 3, 5, -10, 4}

Ans = 3

Explanation

For arr[2] there are three elements less than 3 that is 1, -5 and -10. So arr[0] is noble integer.

For arr[3] there are five elements less than 5 that is 1, 3, 4, 5, -5 and -10. So arr[3] is noble integer.

For arr[5] there are four elements less than 4 that is 1, 3, -5 and -10. So arr[5] is noble integer.

In total there are 3 noble elements.

BruteForce Approach :

Iterate through every element in the array, for every element count the number of smaller elements.

– TC - O(N^2)

– SC - O(1)

Optimized Approach :

Sort the elements and then for each element compare total elements in left side with the current element value.

```
int find_nobel_integers(int arr[], int n) {
   sort(arr);
   int ans = 0;
   for (int i = 0; i < n; i++) {
      if (arr[i] == i) {
         ans = ans + 1;
      }
   }
   return ans;
}
```

– TC - O(nlogn)

– SC - O(1)

Problem 3 : Find count of noble integers (Not Distinct)

Note: Same as previous question, but all elements need not to be distinct

Brute Force Approach :

Iterate and count all smaller elements for current element.

Optimized Approach :

- If the current element is same as previous element then the total number of smaller elements will be same as previous element.
- If current element is not equal to previous element then the total number of smaller elements is equal to its index.

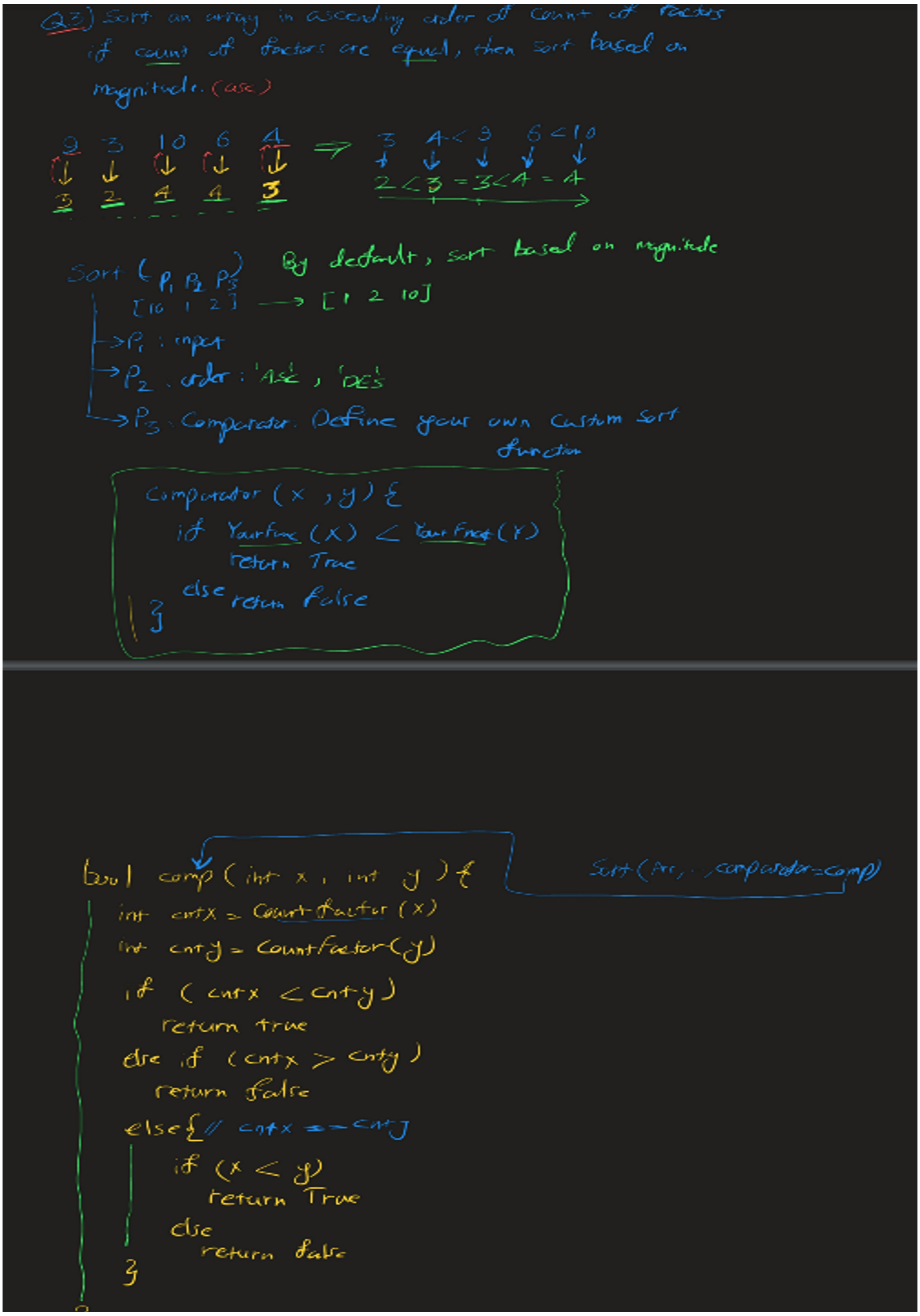
```
int find_nobel_integers(int arr[], int n) {
   sort(arr);
   int count = 0, ans = 0;
   if (arr[0] == 0) ans++;

   for (int i = 1; i < n; i++) {
      if (arr[i] != arr[i - 1])
         count = i;

      if (count == arr[i])
         ans++;
   }
   return ans;
}
```

– TC - O(nlogn)
– SC - O(1)

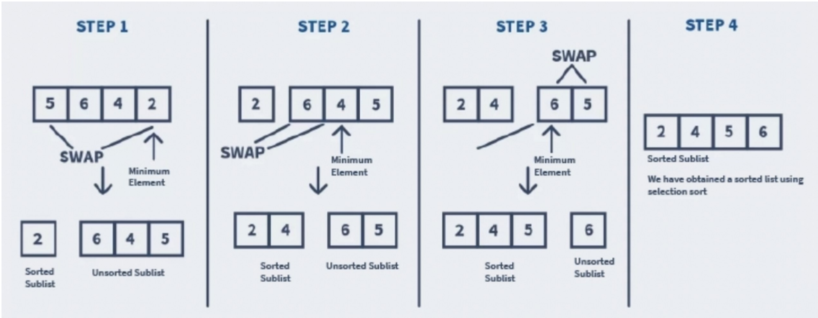
Problem 4 : Sort an array in ascending order of count of factors if count of factors are same then sort on base of magnitude.



1. Basic Sorting Algorithms

A. Selection Sort

- To begin with, place all the students in the unarranged queue.
- From this unarranged queue, search for the shortest student and place him/her in the list of arranged students.
- Again, from the unarranged queue, select the second-shortest student. Place this student in the arranged queue, just after the smallest student.
- Repeat the above-given steps until all the students are placed into the arranged queue.




```
void selectionSort(int arr[], int size) {
   int i, j, minIndex;
   for (i = 0; i < size - 1; i++) {
      // set minIndex equal to the first unsorted element
      minIndex = i;
      //iterate over unsorted sublist and find the minimum element
      for (j = i + 1; j < size; j++) {
         if (arr[j] < arr[minIndex]) {
            minIndex = j;
         }
      }
      // swapping the minimum element with the element at minIndex to place it at its correct position
      swap(arr[minIndex], arr[i]);
   }
}
```

Time Complexity : O(N2) since we have to iterate entire list to search for a minimum element every time.
Space Complexity : O(1)

B. Insertion Sort

Approach :

Line 2: We don't process the first element, as it has nothing to compare against.
Line 3: Loop from i=1 till the end, to process each element.
Line 4: Extract the element at position i i.e. array[i]. Let it be called E.
Line 5: To compare E with its left elements, loop j from i-1 to 0
Line 6, 7: Compare E with the left element, if E is lesser, then move array[j] to right by 1.
Line 8: Once we have found the position for E, place it there.

```
void insertionSort(int arr[], int n) {
   for (int i = 1; i < n; i++) { // Start from 1 as arr[0] is always sorted
      Int currentElement = arr[i];
      Int j = i - 1;
      // Move elements of arr[0..i-1], that are greater than key,
      // to one position ahead of their current position
      while (j >= 0 && arr[j] > currentElement) {
         arr[j + 1] = arr[j];
         j = j - 1;
      }
      // Finally place the Current element at its correct position.
      arr[j + 1] = currentElement;
   }
}
```

Time Complexity:

Worst Case: O(N^2), when the array is sorted in reverse order.

Best Case: O(N), when the data is already sorted in desied order, in that case there will be no swap.

Space Complexity: O(1)

Note:

Both Selection & Insertion are in-place sorting algorithms, means they don't need extra space.
Since the time complexity of both can go to O(N^2), it is only useful when we have a lesser number of elements to sort in an array.