Insertion Sort(easy)

Given an array of integers, sort the elements in the array in ascending order. The insertion sort algorithm should be used to solve this problem.

Examples

- {1, 2, 3} is sorted to {1, 2, 3}
- {4, 2, -3, 6, 1} is sorted to {-3, 1, 2, 4, 6}

Corner Cases

- What if the given array is null? In this case, we do not need to do anything.
- What if the given array is of length zero? In this case, we do not need to do anything.

```
// sort an array arr using insertion sort.
void InsertionSort(vector<int>& arr, int n){
    int temp, j;
    if(arr.empty()) {
        cout << "arr is empty." << endl;
        return;
    }
    for(int i = 1; i < n; i++){ // number of iterations
        temp = arr[i];
        j = i - 1;
        // move the arr[j] one position ahead if it's greater than temp
        while(j >= 0 && arr[j] > temp){
            arr[j + 1] = arr[j];
            j--;
        }
        // find the position for temp
        arr[j + 1] = temp;
    }
}
```

Time Complexity : O(n^2). There are two loops in the function. Both loops take O(n) time.
//Ok, no problem!

Selection Sort(easy)

Given an array of integers, sort the elements in the array in ascending order. The selection sort algorithm should be used to solve this problem.

Examples

- {1} is sorted to {1}
- {1, 2, 3} is sorted to {1, 2, 3}
- {3, 2, 1} is sorted to {1, 2, 3}
- {4, 2, -3, 6, 1} is sorted to {-3, 1, 2, 4, 6}

Corner Cases

- What if the given array is null? In this case, we do not need to do anything.
- What if the given array is of length zero? In this case, we do not need to do anything.

```cpp
void swap(int* a, int*b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

// sort an array arr using selection sort.
void SelectionSort(vector<int>& arr){
    int n = arr.size();
    int temp, global;
    if(arr.empty()) {
        cout << "arr is empty." << endl;
        return;
    }
    for(int i = 0; i < n - 1; i++){
        global = i;
        for(int j = i + 1; j < n; j++){
            if(arr[j] < arr[global])
                global = j;
        }
        // switch arr[i] and arr[global] (min value)
        swap(arr[i], arr[global]);
    }
}
```

Time Complexity : O(n^2). There are two loops in the function. Both loops take O(n) time.
//Ok, no problem!

Merge Sort(Medium)

Given an array of integers, sort the elements in the array in ascending order. The merge sort algorithm should be used to solve this problem.

Examples

- {1} is sorted to {1}
- {1, 2, 3} is sorted to {1, 2, 3}
- {3, 2, 1} is sorted to {1, 2, 3}
- {4, 2, -3, 6, 1} is sorted to {-3, 1, 2, 4, 6}

Corner Cases

- What if the given array is null? In this case, we do not need to do anything.
- What if the given array is of length zero? In this case, we do not need to do anything.

```cpp
// Merge two sorted sub-array of arr from the end
void merge(vector<int>& arr, int low, int mid, int high){
    int n1 = mid - low;
    int n2 = high - mid - 1;
    int n3 = high;
    vector<int> lArr, rArr;
    // copy two sub-arrays to lArr, rArr
    for(int i = 0; i <= n1; i++)
        lArr.push_back(arr[low + i]);
    for(int i = 0; i <= n2; i++)
        rArr.push_back(arr[mid + 1 + i]);
    // merge two sub-arrays from the end
    while(n1 >= 0 && n2 >= 0){
        if(rArr[n2] > lArr[n1])
            arr[n3--] = rArr[n2--];
        else
            arr[n3--] = lArr[n1--];
    }
    // copy the remaining elements in lArr or rArr (if any) to arr
    while(n1 >= 0) {
        arr[n3--] = lArr[n1--];
    }
    while(n2 >= 0) {
        arr[n3--] = rArr[n2--];
    }
}

// sort an array arr using merge sort.
void mergeSort(vector<int>& arr, int low, int high){
    if(low < high){
        int mid = low + (high - low) / 2;
        mergeSort(arr, low, mid);
        mergeSort(arr, mid+1, high);
        merge(arr, low, mid, high);
    }
}
```

Time Complexity: O(nlogn)

T(n) = 2T(n/2) + O(n), according to the case of 2 Master Theorem, T(n) = O(nlogn).

//Ok, good!

Quick Sort(Medium)

Given an array of integers, sort the elements in the array in ascending order. The quick sort algorithm should be used to solve this problem.

Examples

- {1} is sorted to {1}
- {1, 2, 3} is sorted to {1, 2, 3}
- {3, 2, 1} is sorted to {1, 2, 3}
- {4, 2, -3, 6, 1} is sorted to {-3, 1, 2, 4, 6}

Corner Cases

- What if the given array is null? In this case, we do not need to do anything.
- What if the given array is of length zero? In this case, we do not need to do anything.

```cpp
// use the last element in arr as the pivot, move it to the correct position in sorted array.
int partition(vector<int>& arr, int low, int high){
    int& pivot = arr[high];
    int i = low;
    int j = high - 1;
    while(i <= j){
        if(arr[i] < pivot)
            i++;
        else{
            swap(arr[i], arr[j]);
            j--;
        }
    }
    swap(arr[i], pivot);
    return i;
}

// sort an array arr using quick sort.
void quickSort(vector<int>& arr, int low, int high){
    if(low < high){
        int pos = partition(arr, low, high);
        quickSort(arr, low, pos-1);
        quickSort(arr, pos+1, high);
    }
}
```

Time Complexity: O(nlogn)

$T(n) = 2T(n/2) + O(n)$, according to the case of 2 Master Theorem, $T(n) = O(n\log n)$.

//Ok, great!
//But about time complexity of the quicksort, it’s not always O(nlogn)
Assume you choose a extreme value for pivot everytime, then it would reduce to O(n^2).
One method to sovle it is to choose a random value, but it’s not stable.
The best method is to find the median everytime so we can always achieve O(nlogn).

Sort In Specified Order(Medium)

Given two integer arrays A1 and A2, sort A1 in such a way that the relative order among the elements will be same as those are in A2.

For the elements that are not in A2, append them in the right end of the A1 in an ascending order.

Assumptions:

- A1 and A2 are both not null.
- There are no duplicate elements in A2.

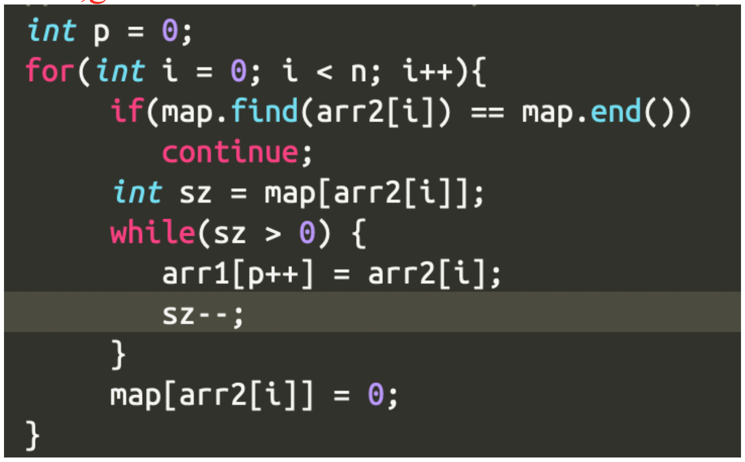
Examples:

- A1 = {2, 1, 2, 5, 7, 1, 9, 3}, A2 = {2, 1, 3}, A1 is sorted to {2, 2, 1, 1, 3, 5, 7, 9}

```cpp
// Sort arr1 in such a way that the relative order among the elements will be same as those are in arr2.
void relativeSortArray(vector<int>& arr1, vector<int>& arr2){
    unordered_map<int, int> map;
    int m = arr1.size();
    int n = arr2.size();
    // use a hash table to store count of each element in arr, it takes O(m) time
    for(int i = 0; i < m; i++){
        if(map.find(arr1[i]) == map.end())
            map[arr1[i]] = 1;
        else
            map[arr1[i]]++;
    }
    // If the element in arr2 present in map, put in arr1 that many times, it takes O(n) time
    int p = 0;
    for(int i = 0; i < n; i++){
        if(map.find(arr2[i]) == map.end())
            continue;
        while(map[arr2[i]] > 0) {
            arr1[p++] = arr2[i];
            map[arr2[i]]--;
        }
    }
    // put the rest of numbers to restArr
    vector<int> restArr;
    for(auto& it : map){
        while(it.second != 0) {
            restArr.push_back(it.first);
            it.second--;
        }
    }
    // sort restArr using quick sort and put in arr1, it takes O(qlogq) time
    int q = restArr.size();
33.     quickSort(restArr, 0, q-1);
34.     for(int i = 0; i < q; i++)
35.         arr1[p++] = restArr[i];
36. }
```

Time Complexity: O(m+n+qlogq).

//Ok ,great!But it can be faster



```
int p = 0;
for(int i = 0; i < n; i++){
     if(map.find(arr2[i]) == map.end())
        continue;
     int sz = map[arr2[i]];
     while(sz > 0) {
        arr1[p++] = arr2[i];
        sz--;
     }
     map[arr2[i]] = 0;
}
```

PS: Next time you submit you code, please remove the number in front of every line. So I can debug your code conveniently. Thanks!