Rainbow Sort (Medium)

Given an array of balls, where the color of the balls can only be Red, Green or Blue, sort the balls such that all the Red balls are grouped on the left side, all the Green balls are grouped in the middle and all the Blue balls are grouped on the right side. (Red is denoted by -1, Green is denoted by 0, and Blue is denoted by 1).

Examples

- {0} is sorted to {0}
- {1, 0} is sorted to {0, 1}
- {1, 0, 1, -1, 0} is sorted to {-1, 0, 0, 1, 1}

Assumptions

- The input array is not null.

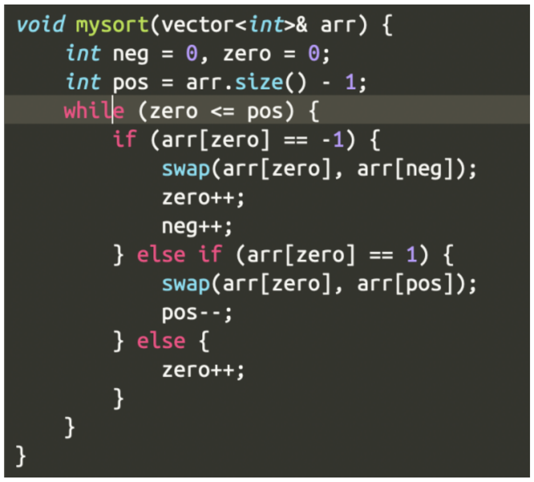
Corner Cases

- What if the input array is of length zero? In this case, we should not do anything as well.

```
// sort an array of balls arr using rainbow sort, similar to quick sort.
// general solution for rainbow sort problem
void rainbowSort(vector<int>& arr, int left, int right, int color_from, int
color_to){
    if(left==right || color_from == color_to)
        return;
    int pivot = color_from + (color_to - color_from) / 2;
    int i = left;
    int j = right;
    while(i <= j){
        if(arr[i] <= pivot)
            i++;
        else{
            swap(arr[i], arr[j]);
            j--;
        }
    }
    rainbowSort(arr, left, i-1, color_from, pivot);
    rainbowSort(arr, i, right, pivot+1, color_to);
}
```

Time Complexity: O(nlogk), k is the number of colors.

//Ok, but it’s not necessary to use rainbow sort
//A more simple solution:



```
void mysort(vector<int>& arr) {
    int neg = 0, zero = 0;
    int pos = arr.size() - 1;
    while (zero <= pos) {
        if (arr[zero] == -1) {
            swap(arr[zero], arr[neg]);
            zero++;
            neg++;
        } else if (arr[zero] == 1) {
            swap(arr[zero], arr[pos]);
            pos--;
        } else {
            zero++;
        }
    }
}
```

Rainbow Sort II (Medium)

Given an array of balls, where the color of the balls can only be Red, Green, Blue or Black, sort the balls such that all balls with same color are grouped together and from left to right the order is Red->Green->Blue->Black. (Red is denoted by 0, Green is denoted by 1, Blue is denoted by 2 and Black is denoted by 3).

Examples

- {0} is sorted to {0}
- {1, 0} is sorted to {0, 1}
- {1, 3, 1, 2, 0} is sorted to {0, 1, 1, 2, 3}

Assumptions

- The input array is not null.

Same answer as Rainbow Sort (Medium) in class2 homework.
//Ok

Move 0s To The End I (Easy)

Given an array of integers, move all the 0s to the right end of the array. The relative order of the elements in the original array does not need to be maintained.

Assumptions:

- The given array is not null.

Examples:

- {1} --> {1}
- {1, 0, 3, 0, 1} --> {1, 3, 1, 0, 0} or {1, 1, 3, 0, 0} or {3, 1, 1, 0, 0}

```
// move all the 0s to the right end of the array arr,
// and remain the relative order in the original array.
void moveZeroToEnd(vector<int>& arr){
    int n = arr.size() - 1;
    int count = 0;
    for(int i = 0; i < n; i++){//Here, should be i <= n, because n is size - 1
        if(arr[i] != 0)
            swap(arr[count++], arr[i]);
    }
}
```

Time Complexity: O(n), since we only traverse the array one time.
//Please be careful for the index, I may suggest using i < n because it’s more conventional and natural.

Binary Search

First Occurrence (Medium)

Given a target integer T and an integer array A sorted in ascending order, find the index of the first occurrence of T in A or return -1 if there is no such index.

Assumptions

- There can be duplicate elements in the array.

Examples

- A = {1, 2, 3}, T = 2, return 1
- A = {1, 2, 3}, T = 4, return -1
- A = {1, 2, 2, 2, 3}, T = 2, return 1

Corner Cases

- What if A is null or A of zero length? We should return -1 in this case.

```
// find first occurance of integer target in array arr
int first(vector<int> arr, int target){
    if(arr.empty())
        return -1;
    int left = 0;
    int right = arr.size() - 1;
    while(left < right - 1){
        int mid = left + (right - left) / 2;
        if(arr[mid] == target)
            right = mid;
        else if(arr[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    // post-processing
    if(arr[left] == target)
        return left;
    if(arr[right] == target)
        return right;
    return -1;
}
```

Time Complexity: O(logn), binary search variant 1.2, T(n) = T(n/2)+O(1).
//Ok, great!

Last Occurrence (Medium)

Given a target integer T and an integer array A sorted in ascending order, find the index of the last occurrence of T in A or return -1 if there is no such index.

Assumptions

- There can be duplicate elements in the array.

Examples

- A = {1, 2, 3}, T = 2, return 1
- A = {1, 2, 3}, T = 4, return -1
- A = {1, 2, 2, 2, 3}, T = 2, return 3

Corner Cases

- What if A is null or A is array of zero length? We should return -1 in this case.

```
// find last occurance of integer target in array arr
int last(vector<int> arr, int target){
    if(arr.empty())
        return -1;
    int left = 0;
    int right = arr.size() - 1;
    while(left < right - 1){
        int mid = left + (right - left) / 2;
        if(arr[mid] == target)
            left = mid;
        else if(arr[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    // post-processing
    if(arr[right] == target)
        return right;
    if(arr[left] == target)
        return left;
    return -1;
}
```

Time Complexity: O(logn), binary search variant 1.2, T(n) = T(n/2)+O(1).
//Ok, no problem!

Closest In Sorted Array

Given a target integer T and an integer array A sorted in ascending order, find the index i in A such that A[i] is closest to T.

Assumptions

- There can be duplicate elements in the array, and we can return any of the indices with same value.

Examples

- A = {1, 2, 3}, T = 2, return 1
- A = {1, 4, 6}, T = 3, return 1
- A = {1, 4, 6}, T = 5, return 1 or 2
- A = {1, 3, 3, 4}, T = 2, return 0 or 1 or 2

Corner Cases

- What if A is null or A is of zero length? We should return -1 in this case.

```cpp
// Find closest element to integer target in array arr
int findClosest(vector<int> arr, int target){
    if(arr.empty())
        return -1;
    int left = 0;
    int right = arr.size() - 1;
    while(left < right - 1){
        int mid = left + (right - left) / 2;
        if(arr[mid] == target)
            return mid;
        else if(arr[mid] < target)
            left = mid;
        else
            right = mid;
    }
    // post-processing
    if(abs(arr[left] - target) <= abs(arr[right] - target))
        return left;
    else
        return right;
}
```

Time Complexity: O(logn), binary search variant 1.1, T(n) = T(n/2)+O(1).
//OK, good!

Search Insert Position (Medium)

Given a sorted array and a target value, return the index where it would be if it were inserted in order.

Assumptions
If there are multiple elements with value same as target, we should insert the target before the first existing element.

Examples

[1,3,5,6], 5 → 2

[1,3,5,6], 2 → 1

[1,3,5,6], 7 → 4

[1,3,3,3,5,6], 3 → 1

[1,3,5,6], 0 → 0

```
// Return the index where it would be if it were inserted in order.
int searchInsert(vector<int>& arr, int target) {
    if(arr.empty())
        return 0;
    int left = 0;
    int right = arr.size() - 1;
    while(left <= right){
        int mid = left + (right - left) / 2;
        if(arr[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return left;
}
```

Time Complexity: O(logn), similar to classic binary search.
//Ok, great!

Classical Binary Search(Easy)

Given a target integer T and an integer array A sorted in ascending order, find the index i such that A[i] == T or return -1 if there is no such index.

Assumptions

- There can be duplicate elements in the array, and you can return any of the indices i such that A[i] == T.

Examples

- A = {1, 2, 3, 4, 5}, T = 3, return 2
- A = {1, 2, 3, 4, 5}, T = 6, return -1
- A = {1, 2, 2, 2, 3, 4}, T = 2, return 1 or 2 or 3

Corner Cases

- What if A is null or A is of zero length? We should return -1 in this case.

```
//classic version of binary search
int binarySearch(vector<int>& arr, int target){
    if(arr.empty())
        return -1;
    int left = 0;
    int right = arr.size() - 1;
    while(left <= right){
        int mid = left + (right - left) / 2;
        if(arr[mid] == target)
            return mid;
        else if(arr[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}
```

Time Complexity: O(logn)//OK, good!