Binary Search II

Find Local Minimum (Medium)

Given an unsorted integer array, return any of the local minimum's index.

An element at index i is defined as local minimum when it is smaller than all its possible two neighbors a[i - 1] and a[i + 1]

(you can think a[-1] = +infinite, and a[a.length] = +infinite)

Assumptions:

- The given array is not null or empty.
- There are no duplicate elements in the array.
- There is always one and only one result for each case.

```
// return an index of local minimum
int localMin(vector<int>& arr){
    if(arr.empty())
        return -1;
    int left = 0;
    int right = arr.size() - 1;
    while(left <= right){
        int mid = left + (right - left) / 2;
        if((mid == 0 || arr[mid - 1] > arr[mid]) && (mid == arr.size()-1 ||
arr[mid] < arr[mid + 1]))
            return mid;
        else if(mid > 0 && arr[mid] > arr[mid - 1])
            right = mid - 1;
        else
            left = mid + 1;
    }
}
```

Time Complexity: O(logn), same as binary search.

Search In Shifted Sorted Array I(Medium)

Given a target integer T and an integer array A, A is sorted in ascending order first, then shifted by an arbitrary number of positions.
For Example, A = {3, 4, 5, 1, 2} (shifted left by 2 positions). Find the index i such that A[i] == T or return -1 if there is no such index.

Assumptions

- There are no duplicate elements in the array.

Examples

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

Corner Cases

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

```
// search an element in a sorted array using binary search
int search(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] >= arr[left]){ // mid is on the left side of tuning point
            if(target >= arr[left] && target < arr[mid])
                right = mid - 1;
            else
                left = mid + 1;
        }
        else{ // mid is on the right side of tuning point
            if(target <= arr[right] && target > arr[mid])
                left = mid + 1;
            else
                right = mid - 1;
        }
    }
    return -1;
}
```

Time Complexity: O(logn), same as binary search.

K Closest In Sorted Array (Medium)

Given a target integer T, a non-negative integer K and an integer array A sorted in ascending order, find the K closest numbers to T in A.

Assumptions

- A is not null
- K is guranteed to be >= 0 and K is guranteed to be <= A.length

Return

- A size K integer array containing the K closest numbers(not indices) in A, sorted in ascending order by the difference between the number and T.

Examples

- A = {1, 2, 3}, T = 2, K = 3, return {2, 1, 3} or {2, 3, 1}
- A = {1, 4, 6, 8}, T = 3, K = 3, return {4, 1, 6}

```
// Find closest element to integer target in array arr using binary search
// Same code in homework 2
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;
}

vector<int> findClosestElements(vector<int>& arr, int k, int target) {
    int n = arr.size() - 1;
    if(target <= arr[0])
        return vector<int> (arr.begin(), arr.begin() + k);
```

```
    else if(target >= arr[n])
        return vector<int> (arr.end() - k, arr.end());
    int index = findClosest(arr, target);
    int left = max(0, index - k + 1);
    int right = min(n, index + k - 1);
    while(right - left + 1 > k){
        if(abs(arr[left] - target) <= abs(arr[right] - target))
            right--;
        else
            left++;
    }
    return vector<int> (arr.begin() + left, arr.begin() + right + 1);
}
```

Time Complexity: O(logn) for function findClosest(), O(2k) for the rest part of function findClosestElements. Overall, it takes O(logn + k).

Array

2 Sum Closest (Medium)

Find the pair of elements in a given array that sum to a value that is closest to the given target number. Return the values of the two numbers.

Assumptions

- The given array is not null and has length of at least 2

Examples

- A = {1, 4, 7, 13}, target = 7, closest pair is 1 + 7 = 8, return [1, 7].

```
// same code in homework2
int partition(vector<int>& arr, int left, int right){
    int i = left;
    int j = right - 1;
    int& pivot = arr[right];
    while(i<=j){
        if(arr[i] < pivot)
            i++;
        else{
            swap(arr[i], arr[j]);
            j--;
        }
    }
    swap(arr[i], pivot);
    return i;
}

void quickSort(vector<int>& arr, int left, int right){
    if(left < right) {
        int pos = partition(arr, left, right);
        quickSort(arr, left, pos - 1);
        quickSort(arr, pos + 1, right);
    }
}

// Find the pair of elements in arr that sum to a value that is closest to target
vector<int> twoSumClosest(vector<int>& arr, int target){
    vector<int> result;
    int n = arr.size() - 1;
    // sort the array arr using quick sort
    quickSort(arr, 0, n);
    int left = 0, right = n;
```

```
    while(left < right - 1){
        int sum = arr[left] + arr[right];
        if(sum > target) {
            if(abs(sum - target) < abs(arr[left] + arr[right - 1] - target))
                break;
            right--;
        }
        else {
            if(abs(sum - target) < abs(arr[left + 1] + arr[right] - target))
                break;
            left++;
        }
    }
    result.push_back(arr[left]);
    result.push_back(arr[right]);
    return result;
}
```

Time Complexity: O(nlogn) for quick sort, O(n) for the rest part of function twoSumClosest(). Overall, it takes O(nlogn).

2 Sum Smaller (Medium)

Determine the number of pairs of elements in a given array that sum to a value smaller than the given target number.

Assumptions

- The given array is not null and has length of at least 2

Examples

- A = {1, 2, 2, 4, 7}, target = 7, number of pairs is 6({1,2}, {1, 2}, {1, 4}, {2, 2}, {2, 4}, {2, 4})

```
int twoSumSmaller(vector<int>& arr, int target){
    int n = arr.size() - 1;
    // sort the array arr using quick sort
    quickSort(arr, 0, n);
    int left = 0, right = n;
    int count = 0;
    while(left < right){
        int sum = arr[left] + arr[right];
        if(sum < target){
            count += (right - left);
            left++;
        }
        else
            right--;
    }
    return count;
}
```

Time Complexity: O(nlogn) for quick sort, O(n) for the rest part of function twoSumSmaller(). Overall, it takes O(nlogn).