<https://leetcode.com/problems/squares-of-a-sorted-array/>

Given an integer array nums sorted in **non-decreasing** order, return *an array of* ***the squares of each number*** *sorted in non-decreasing order*.

**Example 1:**

Input: nums = [-4,-1,0,3,10]

Output: [0,1,9,16,100]

Explanation: After squaring, the array becomes [16,1,0,9,100].

After sorting, it becomes [0,1,9,16,100].

**Example 2:**

Input: nums = [-7,-3,2,3,11]

Output: [4,9,9,49,121]

**Constraints:**

* 1 <= nums.length <= 104
* -104 <= nums[i] <= 104
* nums is sorted in **non-decreasing** order.

**Follow up:** Squaring each element and sorting the new array is very trivial, could you find anO(n)solution using a different approach?

**Attempt 1: 2023-03-05**

**Solution 1: Sorting (10 min)**

class Solution {

public int[] sortedSquares(int[] nums) {

int[] result = new int[nums.length];

for(int i = 0; i < nums.length; i++) {

result[i] = nums[i] \* nums[i];

}

Arrays.sort(result);

return result;

}

}

Time Complexity:O(nlogn)

Space Complexity:O(1)

**Solution 2: Two Pointers (10 min)**

class Solution {

public int[] sortedSquares(int[] nums) {

int[] result = new int[nums.length];

int i = 0;

int j = nums.length - 1;

int k = nums.length - 1;

// Don't miss the last element when 'i == j'

while(i <= j) {

if(Math.abs(nums[i]) > Math.abs(nums[j])) {

result[k--] = nums[i] \* nums[i];

i++;

} else {

result[k--] = nums[j] \* nums[j];

j--;

}

}

return result;

}

}

Time Complexity:O(n)

Space Complexity:O(1)

**Refer to**

<https://leetcode.com/problems/squares-of-a-sorted-array/solutions/221922/java-two-pointers-o-n/>

The crux over here is that the array is already sorted.

We are comparing the first and last elements because after square these have the possibility of being the highest element.

Both the extremes contain the max element (after square ofc), so we are inserting these elements to the last of the new array to make it sorted.

class Solution {

public int[] sortedSquares(int[] A) {

int n = A.length;

int[] result = new int[n];

int i = 0, j = n - 1;

for (int p = n - 1; p >= 0; p--) {

if (Math.abs(A[i]) > Math.abs(A[j])) {

result[p] = A[i] \* A[i];

i++;

} else {

result[p] = A[j] \* A[j];

j--;

}

}

return result;

}

}

**Refer to**

<https://leetcode.com/problems/squares-of-a-sorted-array/solutions/973576/a-good-interview-problem-interview-explanation-c/>

I really like these problems when the right solutions are not that trivial to come by and the solutions actually require some explanation analysis for the advantages and disadvantages.

The obvious/simplest solution would be just using **built in sort** (or code one yourself) and you can say it would work O(NlogN) time / O(1) space. (some python people can say built in works for O(N), but w/e)

class Solution {

public:

vector<int> sortedSquares(vector<int>& a) {

for (auto &x : a) x \*= x;

sort(a.begin(), a.end());

return a;

}

};

The interviewer may easily ask if you could do better time wise ... and well, here, time to think a bit xD

One can do **counting sort** with the constraints of NUM\_MAX <= 10^4 are not that big for numbers themselves and scale same as the N <= 10^4 so it would be a good idea. Just need to make every number absolute and count them. O(N) time, O(NUM\_MAX <= 10^4) extra space.

class Solution {

public:

vector<int> sortedSquares(vector<int>& a) {

int j = 0, m = 0;

for (auto &x : a) x = abs(x), m = max(m, x);

vector <int> c(m + 1, 0);

for (auto &x : a) c[x]++;

for (int i = 0; i <= m; i++)

while (c[i]--)

a[j++] = i \* i;

return a;

}

};

For people who are here on leetcode for sometime they would come to an idea of two pointers pretty fast, however, with some caveat.

First why two pointers? Well, after a bit of thinking you may notice that after squaring numbers, all negative numbers are big ... they go down, meet at zero with positive numbers ... and then all those positive numbers are squared. So natural solution of merging two sorted lists from the middle will do. Just use two pointers that go from the middle outwards.

class Solution {

public:

vector<int> sortedSquares(vector<int>& a) {

int m = 0, n = a.size(), k = 0;

vector <int> ret(n, 0);

while (m < n && a[m] < 0) m++;

for (auto &x : a) x \*= x;

int i = m - 1, j = m;

while (i >= 0 && j < n) {

if (a[i] < a[j]) ret[k++] = a[i--];

else ret[k++] = a[j++];

}

while (i >= 0) ret[k++] = a[i--];

while (j < n) ret[k++] = a[j++];

return ret;

}

};

Cleaner implementation of the two pointers is to go from the both ends of the list as the biggest numbers are there and meet in the middle instead. This solution is way nicer to code, but needs a bit of **reverse** thinking ;)

O(N) time, O(N) space.

class Solution {

public:

vector<int> sortedSquares(vector<int>& a) {

int i = 0, j = a.size() - 1, k = j;

vector <int> ret(k + 1, 0);

while (k >= 0) {

if (abs(a[i]) > abs(a[j])) ret[k--] = a[i] \* a[i++];

else ret[k--] = a[j] \* a[j--];

}

return ret;

}

};

I believe the most important part is to be able to **thoroughly argue**. One must be very thorough in the opinions and not leave anything unanswered as the interviewer will notice some inaccuracies in the explanation right away. It is usually the giveaway if the person has analytical skills to understand problems completely, what those problems would entail, and how different solutions address those problems.

Try to argue when would each of the solutions above do great, and when not so great. What if we change constraints (increase, decrease), which solutions would fail, which will thrive? One example is to say that counting sort is actually better when numbers are all NUM\_MAX < N, cause that defines extra space and well that would make it a solution that uses less space with the same O(N) time, or if we increase N < 10^7 e.g.

I hope that helps everyone to prepare for the coding interview, just give it a thought for every other problem that makes a huge difference :)