**Laboratory work #2. Time Complexity. Sorting.**

Solve this problems using sorting algorithms and define their time, space complexity.

Deadline: 15th September 2020 Week 3

<https://leetcode.com/problems/average-salary-excluding-the-minimum-and-maximum-salary/>

class Solution {

public:

double average(vector<int>& salary) {

double total = 0;

double max=0;

double min=1000000;

for (int i = 0; i < salary.size();i++){

total += salary[i];

if (max < salary[i])

max = salary[i];

if (min > salary[i])

min = salary[i];

}

double avgExc = (total - max - min)/(salary.size()-2);

return avgExc;

}

};

In this task, we had to find Average salary excluding maximum and minimum.

To do this, I didn’t use sort algorithms, I just looped through the array-salary and found the minimum and maximum among all salaries. Also, I found the total salary including maximum and minimum. Then to find the Avg excluding minimum and maximum, I just reduced them from total and divided to array.size()-2;

The time complexity if O(N)-in a worst case. Because I used only 1 loop here.

I think that space complexity is O(n) because we used loop.

<https://leetcode.com/problems/relative-sort-array/>

class Solution {

public:

vector<int> relativeSortArray(vector<int>& arr1, vector<int>& arr2) {

int swap\_count = 0;

for (int i = 0; i < arr2.size();i++ ){

for (int j = 0; j < arr1.size();j++ ){

if (arr1[j]==arr2[i]){

swap(arr1[j],arr1[swap\_count]);

swap\_count++;

}

}

}

sort(arr1.begin()+swap\_count, arr1.end());

return arr1;

}

};

To solve this task, I put the elements of arr1 which is in arr2 to the count’s index. For example, initially count is zero so the first element of arr1 which is met in arr2 will be placed to the beginning of arr1. Next one will be placed on 2nd and so on(count++).   
And then, in the end of arr1 there are unsorted elements and to sort them I just used sort method. Swap\_count is counter of how many swaps we did, which also means the beginning of unsorted part of arr1.

Time Complexity : O(n\*n) because I used 2 loops.(I also used sort but for Big O notation we need only the highest one).

<https://leetcode.com/problems/maximum-number-of-coins-you-can-get/>

class Solution {

public:

int maxCoins(vector<int>& piles) {

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

int sum = 0;

for (int i = piles.size()/3;i < piles.size();i = i+2){

sum += piles[i];}

return sum;}};

Time complexity: O(n\*logn)

First I sorted array with sort method. Then, I started loop from piles.size()/3 because first 1/third of array will be for Bob. Then starting from piles.size()/3, we will take each second coin so that it will be less than Alice’s but more than Bob’s. That’s all.

<https://leetcode.com/problems/sort-integers-by-the-power-value/>

Time Complexity: O(n\*n);

class Solution {

public:

int power( int x) {

int count = 0;

while (x != 1){

if (x % 2==0){

x=x/2;

}

else if (x % 2==1){

x=3\*x+1;

}

count++;

}

return count;

}

int getKth(int lo, int hi, int k) {

int size = hi - lo + 1;

int arr\_power[size];

int arr[size];

int fill = 0;

for (int i = lo; i <= hi; i++){

arr[fill] = i;

arr\_power[fill] = power(i);

fill++;

}

for (int i = 0; i < size-1; i++){

for (int j = 0; j < size-i-1; j++){

if (arr\_power[j]>arr\_power[j+1]){

swap(arr[j],arr[j+1]);

swap(arr\_power[j],arr\_power[j+1]);

} }

}

return arr[k-1];

}

};

Space Complexity: O(n)

Firstly, I wrote method that will return power of any number. And then I created 2 arrays and filled first one with range (lo,hi) and second one with powers of the 1st array. Then I just used bubble sort to sort the array with powers and also sorted the first array (depending on powers of elements). Then to returned kth element in the sorted array(k-1).

<https://leetcode.com/problems/largest-perimeter-triangle/>

class Solution {

public:

int largestPerimeter(vector<int>& A) {

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

for (int i = A.size()-3;i >= 0; i--){

if (A[i] + A[i+1] > A[i+2]){

return A[i]+A[i+1]+A[i+2];

}

}

return 0;

}

};

To solve this problem, firstly I used sort method to sort the array and then started loop from the last 3 elements and checked if they can make triangle. If they can, I just returned their sum. And if there is no sum returned, I just return 0. Time complexity is O(n\*logn) due to sort method and space complexity is O(n) .

<https://leetcode.com/problems/intersection-of-two-arrays/>

class Solution {

public:

bool dont\_exist(vector<int> nums, int n){

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

if (nums[i] == n) {

return false;

}

}

return true;

}

vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {

vector<int> inter;

for (int i = 0; i < nums1.size(); i++){

for (int j = 0; j < nums2.size(); j++){

if (nums1[i]==nums2[j] && dont\_exist(inter,nums1[i])){

inter.push\_back(nums1[i]);

} } }

return inter;

}

};

Firstly we create boolean method if number is already exist in array. Then we just loop through both arrays to find the intersection-number and if number doesn’t exist in “inter” array, we push this number to this array. Time complexity is O(n\*n\*n) because we used 2 loops and boolean method inside of them, which also uses loop.

In my opinion space complexity is O(n).

<https://leetcode.com/problems/k-closest-points-to-origin/>

class Solution {

public:

int distance(vector<int> point){

return point[0]\*point[0]+point[1]\*point[1];

}

vector<vector<int>> kClosest(vector<vector<int>>& points, int K) {

vector<int> distances;

vector<vector<int>> f;

for (int i = 0; i < points.size(); i++){

distances.push\_back(distance(points[i]));

}

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

int count = 0;

for (int i = 0; i < points.size(); i++) {

if (distance(points[i]) <= distances[K-1]){

f.push\_back(points[i]);

count++;

}

}

return f;

}

};

First we create method which counts the distances. Then we call it to assign elements to array “distances”. So “distances” will store the distance of each point. And then we sort this array using sort method. Finally, we add element to final array if the distance of point is less than the last needed element(K-1). We need only K elements so we need the only ones which are less than distances[K-1].

Time Complexity: we used sort method that’s why it is O(N).

Space Complexity: I think it is O(n).

<https://leetcode.com/problems/largest-number/>

PS. I saw the answer from the internet but tried to not copy.

class Solution {

public:

static bool compareNum(string a, string b) {

return a + b > b + a;

}

string largestNumber(vector<int>& nums) {

vector<string> arr;

for (auto i:nums){

arr.push\_back(to\_string(i));

}

sort(arr.begin(),arr.end(),compareNum);

string ans="";

for (auto s:arr){

ans+=s;

}

while(ans[0]=='0' && ans.length()>1)

ans.erase(0,1);

return ans;

}

};

Time complexity is- O(n\*logn)

Space Complexity

O(N).(I think)

First we create boolean method “compareNum” which compares 2 string numbers. The + just concatenates s1 and s2 and the > dictates that s1 should appear before s2 in the sorted array if s1 + s2 comes after s2 + s1 in the lexicographic order, otherwise s2 should appear before s1. Then after declaring the array of nums as strings, we sort it with our custom comprator “compareNum”.

Then we add all the elements of arr to result string. And to get rid of leading zeros, we used loop to delete zeros from beginning of array.

<https://leetcode.com/problems/sort-the-matrix-diagonally/>

<https://leetcode.com/problems/increasing-decreasing-string/>

Couldn’t solve this 2 tasks and also I’m not sure about computing of space complexity.