

Laboratory work report

Submitted by:

Madiyar Zhumabekov

IS-1812K

Submitted to:

Yeskendir Sultanov

**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/increasing-decreasing-string/>

**Task:** Given a string s. You should re-order the string using the following algorithm:

* Pick the smallest character from s and append it to the result.
* Pick the smallest character from s which is greater than the last appended character to the result and append it.
* Repeat step 2 until you cannot pick more characters.
* Pick the largest character from s and append it to the result.
* Pick the largest character from s which is smaller than the last appended character to the result and append it.
* Repeat step 5 until you cannot pick more characters.
* Repeat the steps from 1 to 6 until you pick all characters from s.

In each step, If the smallest or the largest character appears more than once you can choose any occurrence and append it to the result.

Return the result string after sorting s with this algorithm.

**Answer:** I have declared the variables that will be used. Wrote a loop that finds the index of the letter. After using the while loop starting from the 0 element to the last element, I check the first 3 conditions. If checks if there is an element in this index. if there is, it adds it to String. In the second while loop starts with the last element and checks 4-6 conditions. If checks if there is an element in this index. if there is, it adds it to String. Time complexity O(N2).

**My code:**

class Solution {

public String sortString(String s) {

int[] arr = new int[26];

StringBuilder sb = new StringBuilder();

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

char c = s.charAt(i);

arr[c-'a']++;

}

int n = s.length();

int i = 0;

while(n>0) {

i = 0;

while(i<arr.length) {

if(n==0){

break;

}

if(arr[i]>0) {

sb.append((char)('a'+i));

arr[i]--;

n--;

}

i++;

}

i=arr.length-1;

while(i>=0) {

if(n==0){

break;

}

if(arr[i]>0) {

sb.append((char)('a'+i));

arr[i]--;

n--;

}

i--;

}

}

return sb.toString();

}

}

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

**Task:** Given an array of unique integers salary where salary[i] is the salary of the employee i.

Return the average salary of employees excluding the minimum and maximum salary.

**Answer:** I have declared the variables that will be used. Used a loop that finds the max and min in the array. In the second loop, it finds the sum of all elements except min and max. In return, divide by the number of elements except min and max. Time complexity: O(N).

**My code:**

class Solution {

public double average(int[] salary) {

int min = salary[0];

int max = salary[0];

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

if(min>salary[i]){

min = salary[i];

}

if(max<salary[i]){

max = salary[i];

}

}

double sum = 0;

double count = 0;

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

if(min!=salary[i] && max!=salary[i]){

sum+=salary[i];

count ++;

}

}

return sum/count;

}

}

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

**Task:**  Given two arrays arr1 and arr2, the elements of arr2 are distinct, and all elements in arr2 are also in arr1.

Sort the elements of arr1 such that the relative ordering of items in arr1 are the same as in arr2. Elements that don't appear in arr2 should be placed at the end of arr1 in ascending order.

**Answer:** I have declared the variables that will be used. In the first loop, elements from the second array are added to the HashSet. In the second loop, we write all the elements that are in both arrays. Sort the first array. In the third cycle check for uniqueness of elements if the element is not met is recorded in the array. Time complexity: O(nm).

**My code:**

class Solution {

public int[] relativeSortArray(int[] arr1, int[] arr2) {

int[] result = new int[arr1.length];

HashSet<Integer> set = new HashSet<Integer>();

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

set.add(arr2[i]);

}

int temp = 0;

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

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

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

result[temp]=arr1[j];

temp ++;

}

}

}

Arrays.sort(arr1);

for(int i = 0; i<arr1.length; i++){

if(!set.contains(arr1[i])){

result[temp] = arr1[i];

temp++;

}

}

return result;

}

}

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

**Task:** Given a m \* n matrix mat of integers, sort it diagonally in ascending order from the top-left to the bottom-right then return the sorted array.

**Answer:**  I have declared the variables that will be used. I used map and Priority queue. All diagonals have the same value when subtracting i-j. We can treat i-j as a key due to the fact that it is unique for each diagonal. If there's a key that adds an element if not then creates it. And then adds elements to the array. Time coplexity: O(NM).

**My code:**

class Solution {

public int[][] diagonalSort(int[][] mat) {

Map<Integer, PriorityQueue<Integer>> map = new HashMap();

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

for (int j = 0; j < mat[i].length; j++) {

int key = i - j;

if (map.containsKey(i - j)){

map.get(key).add(mat[i][j]);

}

else {

map.put((key), new PriorityQueue());

map.get(key).add(mat[i][j]);

}

}

}

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

for (int j = 0; j < mat[i].length; j++) {

int key = i - j;

mat[i][j] = map.get(key).poll();

}

}

return mat;

}

}

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

**Task:** There are 3n piles of coins of varying size, you and your friends will take piles of coins as follows:

In each step, you will choose any 3 piles of coins (not necessarily consecutive).

Of your choice, Alice will pick the pile with the maximum number of coins.

You will pick the next pile with maximum number of coins.

Your friend Bob will pick the last pile.

Repeat until there are no more piles of coins.

Given an array of integers piles where piles[i] is the number of coins in the ith pile.

Return the maximum number of coins which you can have.

**Answer:** Declared variables that will be used. Sorted the array. Divide by 3 the length of the array. In the loop, we find the sum. Time complexity: O(N).

**My code:**

class Solution {

public int maxCoins(int[] piles) {

Arrays.sort(piles);

int sum=0;

int choice=piles.length/3;

for(int i=piles.length-2; choice>0; i-=2){

sum+=piles[i];

choice--;

}

return sum;

}

}

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

**Task:** Share

The power of an integer x is defined as the number of steps needed to transform x into 1 using the following steps:

if x is even then x = x / 2

if x is odd then x = 3 \* x + 1

For example, the power of x = 3 is 7 because 3 needs 7 steps to become 1 (3 --> 10 --> 5 --> 16 --> 8 --> 4 --> 2 --> 1).

Given three integers lo, hi and k. The task is to sort all integers in the interval [lo, hi] by the power value in ascending order, if two or more integers have the same power value sort them by ascending order.

Return the k-th integer in the range [lo, hi] sorted by the power value.

Notice that for any integer x (lo <= x <= hi) it is guaranteed that x will transform into 1 using these steps and that the power of x is will fit in 32 bit signed integer.

**Answer:** Declared variables that will be used. Added all elements between lo and hi to the list. Sorted them using the method. And just return the element. Time complexity: O(N).

**My code:**

class Solution {

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

List<Integer> list = new ArrayList<>();

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

list.add(i);

}

list.sort((n1, n2) -> {

int p1 = power(n1);

int p2 = power(n2);

if (p1 == p2) {

return n1 - n2;

}

return p1 - p2;

});

return list.get(k - 1);

}

private int power(int n) {

if (n == 1) {

return 0;

}

if (n % 2 == 0) {

return 1 + power(n/2);

}

return 1 + power(3\*n + 1);

}

}

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

**Task:** Given an array A of positive lengths, return the largest perimeter of a triangle with non-zero area, formed from 3 of these lengths.

If it is impossible to form any triangle of non-zero area, return 0.

**Answer:** Sorted the array. In the loop, I checked the non-zero triangle. Calculated the perimeter. Time complexity O(N).

**My code:**

class Solution {

public int largestPerimeter(int[] A) {

Arrays.sort(A);

for (int i = A.length - 1; i >= 2; i--) {

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

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

}

return 0;

}

}

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

**Task:** Given two arrays, write a function to compute their intersection.

**Answer:** Declared a variable HashSet. In the loop, I added variables to the HashSet that occur in both arrays. To add result to the array, I used Iterator. And in the loop, I wrote the HashSet values to the array. Time complexity: O(NM).

**My code:**

class Solution {

public int[] intersection(int[] nums1, int[] nums2) {

HashSet<Integer> ans = new HashSet<Integer>();

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

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

if(nums1[i]==nums2[j]){

ans.add(nums1[i]);

}

}

}

int[] result = new int[ans.size()];

Iterator iterator = ans.iterator();

int i = 0;

while (iterator.hasNext()) {

result[i]=(int) iterator.next();

i++;

}

return result;

}

}

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

**Task:** We have a list of points on the plane. Find the K closest points to the origin (0, 0).

(Here, the distance between two points on a plane is the Euclidean distance.)

You may return the answer in any order. The answer is guaranteed to be unique (except for the order that it is in.)

**Answer:**  Declared variables that will be used. I recorded distances in the array. Sorted the array. In the loop, I wrote to result responses that are equal to or less than K-1 elements in the array. Time complexity: O(N).

**My code:**

class Solution {

public int[][] kClosest(int[][] points, int K) {

int[] d = new int[points.length];

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

d[i] = distance(points[i]);

}

Arrays.sort(dists);

int[][] result = new int[K][2];

int t = 0;

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

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

result[t] = points[i];

t++;

}

}

return result;

}

public int distance(int[] point) {

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

}

}

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

**Task:** Given a list of non negative integers, arrange them such that they form the largest number.

**Answer:** Declared variables that will be used. Wrote the array as an array of strings. Sorted the array using a method inside the comparator. Announced StringBuilder added sorted strings. Time complexity: O(N).

**My code:**

class Solution {

public String largestNumber(int[] nums) {

String[] numsString = new String[nums.length];

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

numsString[i] = String.valueOf(nums[i]);

}

Arrays.sort(numsString, new Comparator<String>() {

public int compare(String a, String b) {

String sort1 = a + b;

String sort2 = b + a;

return sort2.compareTo(sort1);

}

});

if (numsString[0].equals("0")){

return "0";

}

StringBuilder sb = new StringBuilder();

for (String str : numsString){

sb.append(str);

}

return sb.toString();

}

}