**Laboratory work №2.**

1. **Increasing Decreasing String**

**Source code:**

**class** Solution {

**public**:

string sortString(string s) {

string result = "";

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

**while** (s.size() > 0){

**if** (s.size() > 0){

result += s[0];

s.erase (0, 1);

}

**while** (**true**){

**bool** flag = **false**;

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

**if** (result.back() < s[i]){

result += s[i];

flag = **true**;

s.erase (i, 1);

**break**;

}

}

**if** (!flag)

**break**;

}

**if** (s.size() > 0){

result += s[s.size() - 1];

s.erase (s.size() - 1, 1);

}

**while** (**true**){

**bool** flag = **false**;

**for** (**int** i = **int**(s.size()) - 1; i >= 0; --i){

**if** (result.back() > s[i]){

result += s[i];

flag = **true**;

s.erase (i, 1);

**break**;

}

}

**if** (!flag)

**break**;

}

}

}

};

1. **Average Salary Excluding the Minimum and Maximum Salary**

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.

**Explanation:** Minimum salary and maximum salary are 1000 and 4000 respectively.

Average salary excluding minimum and maximum salary is (2000+3000)/2= 2500

**Source code:**

**class** Solution {

**public**:

**double** average(vector<**int**>& salary) {

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

**double** sum = 0;

**for** (**int** i = 1; i < salary.size() - 1; ++i){

sum += salary[i];

}

**return** sum / (salary.size() - 2);

}

};

Solving this task, we notice, that all salaries are unique, you can actually do this in linear time by going further, finding the minimum and maximum at the same time, and then just subtracting their sum and you know you are doing this division. But I just sorted the salaries and just subtracted the maximum and minimum. And just divides by the number of elements, doing the whole thing with the number of elements minus 2.

**3.Relative Sort Array**

**Source code:**

**class** Solution {

**public**:

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

vector < **int** > res;

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

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

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

res.push\_back(arr2[i]);

}

**int** m = res.size();

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

**bool** flag = **false**;

**for** (**int** j = 0; j < arr2.size(); ++j)

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

flag = **true**;

**break**;

}

**if** (!flag)

res.push\_back(arr1[i]);

}

sort (res.begin() + m, res.end());

**return** res;

}

};

When we start to solve this task, we will put all the elements from arr2 in the vector. Then we count each element in arr1, if it is in arr2, we count it, otherwise we put it in the others. Then print the numbers from the vector and others.

**4.Sort the Matrix Diagonally**

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.

**5.Maximum Number of Coins You Can Get**

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.

**Explanation:**

Choose the triplet (2, 7, 8), Alice Pick the pile with 8 coins, you the pile with **7** coins and Bob the last one.

Choose the triplet (1, 2, 4), Alice Pick the pile with 4 coins, you the pile with **2** coins and Bob the last one.

The maximum number of coins which you can have are: 7 + 2 = 9.

Other point of view, if we choose this arrangement (1, **2**, 8), (2, **4**, 7) you only get 2 + 4 = 6 coins which is not optimal.

**Source code:**

**class** Solution {

**public**:

**int** maxCoins(vector<**int**>& piles) {

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

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

**int** i = 1; **int** res = 0; **int** cnt = 0;

**while**(cnt < piles.size()) {

res += piles[i];

i += 2;

cnt += 3;

}

**return** res;

}

};

**Time: O(NlogN)**

Here, we will consider the options through the loop, and inside the while we run to the end, and sum up all the elements of the array. Then there I will 1 increase by 2 once, because the first one, the current largest, will always be returned to Alice.

**6.Sort Integers by The Power Value**

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.

**Explanation:**

The power of 12 is 9 (12 --> 6 --> 3 --> 10 --> 5 --> 16 --> 8 --> 4 --> 2 --> 1)

The power of 13 is 9

The power of 14 is 17

The power of 15 is 17

The interval sorted by the power value [12,13,14,15]. For k = 2 answer is the second element which is 13.

Notice that 12 and 13 have the same power value and we sorted them in ascending order. Same for 14 and 15.

**Source code:**

**class** Solution {

**public**:

**static** **int** fifi (**int** n){

**int** cnt = 0;

**while** (n != 1){

**if** (n % 2 == 0)

n /= 2;

**else**

n = 3 \* n + 1;

cnt++;

}

**return** cnt;

}

**static** **bool** cmp (**int** x, **int** y){

**int** x1 = fifi(x);

**int** y1 = fifi(y);

**if** (x1 == y1)

**return** x < y;

**return** x1 < y1;

}

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

vector < **int** > res;

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

res.push\_back (i);

**if** (res.size() == 1)

**return** res[0];

sort (res.begin(), res.end(), cmp);

**return** res[k - 1];

}

};

In this function fifi this is going to give the number of steps for each integer to get into get to want right so while n is not equal to 1 we are going to do this this calculation right if we X is even do X by 2 if X is our do 3 into X plus 1. So same thing we are going to do here if while it is not equal to 1 right if n percentage 2 is equal to 0 that means even then you are going to divide by 2 otherwise multiply the N by 3 and add it to 1 and keep incrementing the steps till n becomes 1 so finally written the number of steps so likewise so now so create a node array right the length of the array will be low - sorry high minus low plus 1 right so that many elements that you know to calculate the powers right so and then so number so each each for each number I write get the power value of I and store them in the norm right so so nodes these numbers are a right sort them using the power compare clause right so this is a power comfortless once you start it then you have calculated the power values and the numbers are sorted according to the 4 values.

**7.Largest Perimeter Triangle**

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

**Source code:**

**class** Solution {

**public**:

**int** largestPerimeter(vector<**int**>& A) {

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

**int** n = A.size();

**for** (**int** i = n -1; i >= 2; --i)

**if** (A[i] < A[i - 2] + A[i - 1])

**return** (A[i] + A[i - 2] + A[i - 1]);

**return** 0;

}

};

Here, we need to find the largest perimeter of a triangle with a non-zero area. And to do this,

first we will sort, when the numbers are already in ascending order, we run through the loop and check A[i] < A[i-2] + A[i-1], then we output the sum of a[i], a[i-2], a[i-1].

**8.Intersection of Two Arrays**

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

**Source code:**

**class** Solution {

**public**:

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

**int** n = nums1.size();

**int** m = nums2.size();

vector < **int** > ans;

**if** (n == 0 || m == 0)

**return** ans;

sort (nums1.begin(), nums1.end());

sort (nums2.begin(), nums2.end());

**for** (**int** i = 0; i < n; ++i){

**for** (**int** j = 0; j < m; ++j){

**if** (nums1[i] < nums2[j])

++i;

**else** **if** (nums1[i] > nums2[j])

++j;

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

**if** (ans.empty() || ans.back() != nums1[i])

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

}

}

}

**return** ans;

}

};

**9.K Closest Points to Origin**

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

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

**Explanation:**

The distance between (1, 3) and the origin is sqrt(10).

The distance between (-2, 2) and the origin is sqrt(8).

Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin.

We only want the closest K = 1 points from the origin, so the answer is just [[-2,2]].