Question 1: Count Palindromes

[Google]
You are given the following:

String s of length n consisting of lowercase alphabets
Task

Determine the number of subsequences of string s such that the subsequence becomes palindromic with some rearrangement of the subsequence (if needed).

Notes

A string a is said to be a subsequence of string b if a can be obtained from b by deleting some characters without changing the ordering of the remaining characters.
An empty string is palindromic.
Example

Assumptions

n = 3
s = "abb"
Approach

The subsequences {"", "a", "b", "b", "bb"} of string s are already palindromic without any rearrangement. In addition, the subsequence {"abb"} can be rearranged into the palindrome "bab", so it is counted as well. No other subsequence can be rearranged into a palindrome, so the answer is 6.

Test case 1

Sample Input 1

3
5
fihig
8
hhhghhih
6
gfihff
Sample Output 1

10

128
20

```
#include <iostream>
// fihig
// f i h i g ii ihi "" fii=>ifi iig=>igi
// a string
// 1
// every single character also a palindrome
// duplicates can form two length palindromes
// higih => ihghi
// generating all the subsequences is not really useful.
// O(string length time).
// 1 + str1.length()
// hhhghhih
// 128
// h: 6, i: 1, g:1
// 1+8+6/2+ nC2 +nc4+ ncn
int main()
{
    std::cout<<"Hello World";

    return 0;
}
```

Question 2: Online Auction Sniper Detector

[Cisco]
An online auction site is investigating "sniping" — last-second bidding bursts intended to outpace honest bidders. The fraud team defines a suspicious window: any contiguous time window of length W seconds in which at least K bids were placed by the same user is flagged.

Bids arrive in chronological order. For each bid, the fraud team wants two answers:

Is the current bid part of a suspicious window? (i.e., does the user have ≥ K bids — including this one — within the last W seconds?)
What is the smallest user id that is currently sniping (i.e., has ≥ K bids in the latest W-second window ending at the current bid's timestamp)? Output -1 if no user qualifies.
Input

N — number of bids
W — window length (seconds)
K — sniping threshold (bids per user)
bids[i] = (t_i, u_i) — timestamp and user id; timestamps strictly increasing

Input Format

N W K
t_1 u_1
t_2 u_2
...
t_N u_N
Sample Input:

6 10 3
0 1
2 1
3 2
8 1
12 1
15 2
Output Format

One line per bid: flag smallest_sniper. flag = 1 if the current bid's user has ≥ K bids in the last W seconds; smallest_sniper is the smallest sniping user id, or -1.
flag_1 sniper_1
flag_2 sniper_2
...
flag_N sniper_N
Sample Output:

0 -1
0 -1
0 -1
1 1
0 -1
0 -1
Output Explanation:

Bid 1 (t=0, u=1): user 1 has 1 bid in (-10, 0] → not sniping.
Bid 2 (t=2, u=1): user 1 has 2 bids in (-8, 2] → not sniping.
Bid 3 (t=3, u=2): user 1 has 2 bids, user 2 has 1: nobody sniping.
Bid 4 (t=8, u=1): user 1 has 3 bids (at 0, 2, 8) within window → sniping! User 2 has 1 bid; smallest sniper is 1.
Bid 5 (t=12, u=1): the bids at t=0 and t=2 are now outside the window (2, 12] (both timestamps are ≤ 2). User 1 has only 2 bids in-window (at 8 and 12) → not sniping. User 2 has 1 bid (at 3). Nobody has ≥ K bids → 0 -1.

Bid 6 (t=15, u=2): the bid at 2 is outside the window (≤ 5). User 1 has bids at 8, 12 → 2 bids. User 2 has bid at 3 (out: 3 ≤ 5? yes) → out. Bid at 15 puts user 2 at 1 bid in (5, 15]. Nobody sniping → 0 -1.
Examples

Example 1: Threshold Never Reached

Input:

3 100 5
0 1
50 2
99 1
Output:

0 -1
0 -1
0 -1
Example 2: Two Simultaneous Bidders Break

Input:

6 10 2
0 5
1 3
2 5
3 3
4 5
5 3
Output:

0 -1
0 -1
1 5
1 3
1 3
1 3
Constraints

1 ≤ N ≤ 200,000
1 ≤ W ≤ 10^9
#include <iostream>
using namespace std;
// N bids [user ids]

```
// W K
// bids
// 6 10 3
// 0 1
// 2 1
// 3 2
// 8 1
// 12 1
// 15 2

// N=3 W=100 K=5
// 0 1
// 50 2
// 99 1

// 6 10 2
// 0 5
// 1 3
// 2 5
// 3 3
// 4 5
// 5 3

vector<int> flagFrauds(int N, int W, int k, vector<vector<int>>&bids) {
    vector<int> result(N);
    // minStamp
    // priority queue : timestamp, user_id
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<vector<pair<int,int>>>> pq;
    for(vector<int> bid : bids) {
        int timeS = bid[0], user_id = bid[1];

    }
}

int main()
{
    std::cout<<"Hello World";

    return 0;
}
```

Question 3:Minimum and Maximum Moves to Fill First Column

[Trilogy]
Minimum and Maximum Moves to Fill First Column
Problem Description
You are given a rectangular board divided into a uniform grid (square cells). Some cells of the board are occupied with blocks, and others are empty. You are trying to add more and more blocks to the board, and your task is to fill the first column with them. You can add a block to the field in the following way: first, you choose the row index, then you throw the new block into the chosen row from the left. The block appears in the leftmost cell of the row and starts moving to the right, until it reaches another block or the end of the row. When that happens, the block starts falling down until it reaches another block or the last row.

For example, if the board looks like this

...#
..#
.#..
.#...
blocks are denoted by '#' and empty cells are denoted by '.'.

and you throw a block into the first or the second row from the top, it will end up like this:

##.#
##.#
.#..
.#...
Your task is to calculate the minimum and the maximum number of moves required to fill the first column of the board with blocks.

Function Signature
vector solution(vector> field)
Examples
Example 1:
Input: field = [['.', '#', '#'], ['.', '.', '#'], ['.', '.', '.']]
Output: [4, 4]
Explanation: We need to choose the first row once and the third row three times.

Example 2:
Input: field = [['.', '#', '#'], ['.', '.', '#'], ['.', '.', '.']]
Output: [3, 6]
Explanation: To minimize the number of moves we need to choose the first row three times. To maximize this number, we need to choose the third row three times, then the second row twice, and the first row one time.

Constraints
Execution time limit: 0.5 seconds (cpp)
Memory limit: 1 GB
Input: array.array.char field
A non-empty rectangular matrix of characters representing a field.
Guaranteed constraints:
1 <= field.length <= 12
1 <= field[0].length <= 12
Output: array.integer
Array of two elements, where the first element is the minimum number of moves required to fill the first column and the second one is the maximum number of moves required to fill the first column. It is guaranteed that the maximum number of moves doesn't exceed 12.

```
#include <iostream>
using namespace std;
--->
   |
   |
// field = [['.', '#', '#'],
//          ['.', '.', '#'],
//          ['.', '.', '.']]
// [['.', '#', '#', '#', '#', '.', '#', '#', '#', '#', '#','#']]
// To find the max number of moves.
// 1. go to each row. and store the index that has first hash block
// 2. 1,2,3 = sum of these is the maximum moves we can make.
// For minimum
// count empty in the colum minus that from n.
// go row wise count empties that you'll need to fill in order to reach the last row.
// if everything above is filled then you might need to fill the last row to fill the first column.

int main()
{
   std::cout<<"Hello World";

   return 0;
}
```

Question 4: Stripe Merchant Fraud Risk Scoring

[Stripe]
The fraud detection team at Stripe aims to reduce merchant fraud risk for Stripe with minimal pain to good merchants through machine learning. To provide initial data for training this

machine learning model, you are tasked with developing a system to assess the fraud risk associated with transactions made to various merchants.

Description

You are given **transactions_list,** which is a list of transactions for a certain day, **merchants_list** which is a list of merchants, and **rules_list** which is a list of rules corresponding to each transaction (more information below). A transaction denotes a payment made by a customer to a merchant. Each merchant in **merchants_list** is assigned a **base_score**. Your task is to assign each merchant a **fraud_score** based on their transaction(s), where each transaction can have different risk factors based on certain categories (e.g., transaction amount, frequency of similar transactions, user activity). To do this:

1. Initialize a merchant's current_score as their base_score
2. For each transaction in the transactions_list,
    a. calculate and update the merchant's current_score.
    b. This should be done in separate passes over the entire list for each step:
        i. If the transaction amount is greater than the corresponding rule's **min_transaction_amount**, multiply current_score by the corresponding rule's **multiplicative_factor**
        ii. If the same **customer_id** has made three or more transactions to that **merchant_id** (including the current transaction), add all the corresponding rules' **additive_factors** to the **current_score**, cumulatively (e.g., if the merchant's score at the third transaction is X which include the first, second, and third **additive_factors**, the fourth transaction should be X + the fourth's transaction **additive_factor**).
        iii. If a transaction is the third or more from the same customer_id in the same hour for the same merchant_id (including current transaction), do the following:
            1. If the hour is between 12 to 17 (inclusive): Add the penalty each time (this includes the first and second transaction as well as all the ones after the third).
            2. If the hour is between 9 to 11, or 18 to 21 (inclusive): Subtract the penalty each time (this includes the first and second transaction as well as all the ones after the third).
            3. If the hour falls outside the above ranges, don't do anything.

Input
Note: You may assume that there won't be any integer overflows.

transactions_list: A list of length n ($1 \leq n \leq 1000$) where each transaction is represented as a comma-separated string.

A string **merchant_id** (length > 0) of the merchant who receives payment
An integer **amount** of the payment

A string **customer_id** (length > 0) of the customer who makes payment
An integer hour (0 ≤ hour ≤ 23) representing the hour of the transaction

rules_list: A list of length n (1 ≤ n ≤ 1000) where each rule is represented as a comma-separated string (and corresponds to each transaction in transactions_list)
An integer min_transaction_amount of the corresponding transaction to consider fraudulent.
An integer multiplicative_factor
An integer additive_factor
An integer penalty
merchants_list: A list of length m (1 ≤ m ≤ 1000) where each item is a merchant profile (comma-separated string) that gives additional context for the score.
A string merchant_id (length > 0) of the merchant who receives the payment
An integer base_score of the merchant