CodeNection 2023 Final Round Editorial

Competition Team of CodeNection 2023

Contents

1 Introduction

The rounds are prepared by the Competition Team of CodeNection 2023:

- Wong Yen Hong, MMU
- Tung Tze Yang, MMU
- Marcus Chin Wei Hern, MMU
- Kalla Deveshwara Rao A/L Rama Ra, MMU
- Law Chin Keat, MMU
- Chan Kar Kin, MMU

The first seven problems in this editorial belong to the closed category, while the last seven belong to the open category. The source codes for solutions are written in C++ and they can be found in this GitHub repository:

`codenection-2023` - https://github.com/wyhong3103/codenection-2023

Lastly, we would like to express our gratitude to the testers for testing the round and providing critical comments:

- rabbitsthecat, HKUST
- Geremie Yeo
- Melody Koh Si Jie, MMU
- Tham Joe Ming, MMU
- Lim Xin Yee, MMU

2 Editorial

2.1 Codey and CodeNection 2

Solution

If a string is reversed an even number of times, the reverse operation has essentially no effect on the string. So, if n is an even number, output the original string; otherwise, output the reversed string.

Time Complexity: $O(1)$

2.2 Codey and Connection

Solution

This problem is about determining whether the string `CODENECTION` is a subsequence of the given array. To do this, we can maintain a pointer pointing to the current target letter and iterate from the first letter to the last letter in the given array. If the letter at the current iteration is equal to the target letter, we move the target letter pointer to the next one. We repeat this process until there are no more letters to point to or we finish iterating through the array. If we finish iterating through the array before finding all the target letters, then `CODENECTION` is not found; otherwise, it is found.

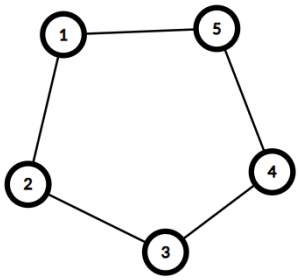
Time Complexity: $O(n)$

2.3 Codey and Schedule

Fun fact: This problem was actually inspired by an actual scheduling problem I encountered when organizing a sports competition.

Solution

To solve this problem, we can think of a graph with n nodes, where each node forms one big cycle. Each node in the graph represents one team, while the edge (i, j) denotes a round with teams i and j.



The answer to this problem is simply the edges of the graph. To understand why this is true, each node is connected to two different nodes, and it has only two edges. This is analogous to each team playing two rounds against different teams.

Time Complexity: $O(n)$

2.4 Codey and School Supplies

Inspired by: Codeforces Round 510 - Vitamins

Solution

To solve this, we first have to identify the minimum cost for each subset s_j of `ABC` that is available in the store. Formally, let's denote the minimum cost of a subset s_j as c_j, and, $c_j = min(a_i)$ for all i, $1 \leq 1 \leq n$ where $b_i = s$.

Then we can find the minimum cost among the array of sets of school supplies that can collectively include `ABC`, which consists of the following:

- `A`, `B`, `C`
- `A`, `BC`
- `AC`, `B`
- `AB`, `BC`
- `AC`, `BC`
- `AC`, `AB`
- `ABC`

One of the testers, rabbitsthecat, proposed an alternative solution that uses a DP (Dynamic Programming) approach. This solution is easier to implement and can be applied to a larger set of items. Let's explore the idea.

For each of the sets b_i, we will represent the set in a bitmask which can be denoted as $mask_i$. For example, if b_i = `AC`, $mask_i$ would be 101, each bit denoting the presence of the `ABC`. We could define the DP state $dp[s]$ as the minimal cost to form a subset of items denoted by the bitmask s and progressively update it from the first set of items to the last.

The DP transition is performed from the first set of items to the last, denoted by $mask_i$, and it is defined as follows:

$$dp[s|mask_i] = min(dp[s|mask_i], dp[s] + a_i)$$

where $s|mask_i$ denotes the bitwise OR operation between s and $mask_i$. This type of DP is also known as bitmask DP.

Time Complexity: $O(n)$

2.5 Codey and Zombies

Inspired by: Educational Codeforces Round 155 - Chips on the Board

Solution

Claim: To eliminate all the zombies, it is either necessary to place one bomb in each row or one bomb in each column.

Proof: Let's assume that not every row and not every column has a bomb. Let i be the row that lacks a bomb, and j be the column that lacks a bomb. In this case, cell (i, j) in the field would not be burned because neither row i nor column j has a bomb.

Now that we have proven our claim, the answer for the minimum is:

$$min\left(\left(\sum_{i=1}^{n} a_i\right) + n \cdot min(b), \left(\sum_{i=1}^{m} b_i\right) + m \cdot min(a)\right)$$

This is because we can place bombs in every row, and then we choose the column that costs the least fire powder, and vice versa.

For maximum, we just need to place a bomb at each cell in the field. Manually iterating through each cell in the field to calculate $\sum_{i=1}^{n}\sum_{j=1}^{m} a_i + b_j$ would not pass the time limit. It turns out we can rearrange the equation to the following form:

$$\sum_{i=1}^{n}\sum_{j=1}^{m} a_i + b_j = \sum_{i=1}^{n}\left(m \cdot a_i + \sum_{j=1}^{m} b_j\right)$$

This allows us to calculate it efficiently by storing $\sum_{j=1}^{m} b_j$ and reusing it in every iteration of a_i.

Time Complexity: $O(max(n, m))$

2.6 Codey and Sightseeing

Solution

It is a well-known fact that you can find the shortest distance from one node to every other node in an undirected and unweighted graph using BFS (Breadth First Search), which is also known as single-source BFS. The time complexity of BFS is $O(V + E)$, where V is the number of vertices and E is the number of edges. However, performing BFS on every place of interest would not pass the time limit. In this case, we can utilize multiple-source BFS, which has a similar time complexity to single-source BFS, and we can determine the shortest distance of every location to its closest place of interest. It is not difficult to derive this technique from the standard single-source BFS. One way to see why this works is to think of doing a BFS on a fake node that has edges to multiple sources.

After finding the shortest distance from every location to its closest place of interest, we can simply sum these distances and output the answer.

Time Complexity: $O(n + m)$

2.7 Codey and Apples

Inspired By: SPOJ - HALLOW

Solution

One interesting thing about this problem is that $m \leq n$, and it turns out that this is the key to the solution. Without such constraints, this problem might not be solvable within the specified time limit.

To solve this problem, we need to find a subset of the n bags that can sum up to a number that is divisible by m. Let's make the claim that it is always possible to obtain a subset of the n bags that can be divided by m when $m \leq n$.

To prove this, we will examine how the remainder x_i changes as we sum up each number:

$$\begin{aligned}
a_1 \bmod m &\equiv x_1 \bmod m \\
(a_1 + a_2) \bmod m &\equiv x_2 \bmod m \\
(a_1 + a_2 + a_3) \bmod m &\equiv x_3 \bmod m \\
&\vdots \\
(a_1 + a_2 + ... + a_n) \bmod m &\equiv x_n \bmod m
\end{aligned}$$

How many possible x_i values are there? There are m possible values, as the remainder of a number when divided by m is between 0 and $m - 1$. So, by the Pigeonhole Principle, when $n > m$, it is guaranteed that there will be at least one duplicated x_i. Let's say $x_1 = x_3$, then:

$$
\begin{aligned}
(a_1 + a_2 + a_3) \bmod m &\equiv (a_1) \bmod m \\
(a_1 + a_2 + a_3) \bmod m - a_1 \bmod m &\equiv 0 \bmod m \\
(a_2 + a_3) \bmod m &\equiv 0 \bmod m
\end{aligned}
$$

Recall that when $x \bmod m \equiv 0$, it essentially means x is divisible by m. Hence, $a_2 + a_3$ is divisible by m. When $n > m$, it is guaranteed that there will be a solution. What if $n = m$? When $n = m$, it is not always the case where there will be a duplicated x_i, in the worst-case scenario, each residue from 0 to $m - 1$ will have exactly one occurrence in x. However, if $x_i = 0$, that means $a_1 + ... + a_i \equiv 0 \bmod m$, which means $a_1 + ... + a_i$ is divisible by m.

Hence, it is proven that the answer always exists.

2.8 Codey and Facto

Inspired by: Educational Codeforces Round 33 - Counting Arrays

Solution

When it comes to factorizing numbers, it's almost unavoidable to first consider the prime factorization of a number. The prime factorization of a number n can be represented as follows::

$$n = \prod_{i=1}^{k} p_i^{a_i}$$

where each p_i represents a unique prime number, and the product of all the p_i raised to the power of a_i equals n.

Let's first solve a simpler problem: How do we find the number of *facto* with no negative numbers?

For each p_i in the prime factorization of n, we know that p_i occurs a_i times, and the idea is to find the total number of unique ways we can distribute the occurrences of p_i among the *facto*. To do that, we can use the stars and bars combinatorial technique, where a_i represents the number of stars and $m-1$ represents the number of bars. For example, when $m = 3$ and $a_i = 3$, there are 3 stars and 2 bars:

$$\mid \star\star \mid \star$$

which is equivalent to $[p_i^0, p_i^2, p_i^1]$.

Now, we can find the number of *facto* with no negative numbers by multiplying the number of unique ways to distribute each p_i, as represented in the equation below:

$$\prod_{i=1}^{k} \binom{m + a_i - 1}{a_i}$$

The simpler problem is solved.

Let's shift our focus to the negative numbers now. Since each n is a non-negative number, it's evident that the count of negative numbers in the *facto* should be an even number. So, the problem we're going to solve is: How many ways are there to choose an even number of numbers from an array of m numbers?

This is a well known combinatorics problem, and it turns out that:

$$\binom{m}{0} + \binom{m}{2} + ... + \binom{m}{2 \cdot \lfloor \frac{m}{2} \rfloor} = 2^{m-1}$$

Finally, we can combine both of the problems we've solved, which leads to this final equation as our answer:

$$\left(\prod_{i=1}^{k} \binom{m + a_i - 1}{a_i} \right) \cdot 2^{m-1}$$

Finally, directly calculating the answer and then taking the modulo with $10^9 + 7$ may not be feasible in some programming languages due to the large integers involved. To address this, we need to use modular arithmetic to compute the answer, more about it can be found in this Codeforces blog, Modular Arithmetic for Beginners by Spheniscine.

Time Complexity: $O(\sqrt{n} + \log m)$

2.9 Codey and Zoey

Solution

Let's consider DP (Dynamic Programming).

Before we delve into defining the DP state, let's first gather some observations. Codey and Zoey will only be moving up and right since they both took the shortest path to (n, m), and $0 \leq n, m \leq 50$. Additionally, with (i, c, z), where i denotes the number of steps taken, c denotes the number of steps Codey has taken upwards, and z denotes the number of steps Zoey has taken upwards, we can uniquely identify the current positions of Codey and Zoey. Notably, when $c = z$, it means Codey and Zoey are at the same position or have intersected. This is because they have both taken the same number of steps upwards and the same number of steps to the right.

Now, let's define the DP state as $dp[i][j][c][z]$, which represents the total number of ways for both Codey and Zoey to take i steps with j intersections, while Codey has taken c steps upwards, and Zoey has taken z steps upwards.

The DP transition is defined as follows:

if $i = j$,

$$\begin{aligned} dp[i][j][c][z] = & dp[i-1][j-1][c-1][z] + \\ & dp[i-1][j-1][c-1][z-1] + \\ & dp[i-1][j-1][c][z-1] + \\ & dp[i-1][j-1][c][z] \end{aligned}$$

else,

$$\begin{aligned} dp[i][j][c][z] =& dp[i-1][j][c-1][z] + \\ & dp[i-1][j][c-1][z-1] + \\ & dp[i-1][j][c][z-1] + \\ & dp[i-1][j][c][z] \end{aligned}$$

This solution runs in $O(max(n,m)^4)$, but the constraint for this problem is also relaxed to allow $max(n,m)^5$ to pass.

Time Complexity: $O(max(n,m)^4)$