2023 Stage 9: Division 2 Solutions

The Judges

Jul 9, 2023

Air Race

Problem

- Simulate a paper airplane flying with a fading wind.

Solution

- This problem requires carefully following the rules stipulated in the problem. It should suffice to translate the rules directly into code.
- The easiest way to compute $\left\lfloor \frac{x}{10} \right\rfloor$ is to use integer division. In languages like C++ and Java, `x / 10` when x is a positive integer will automatically give the result rounded down. In Python 3, `x // 10` will do the same.

Business

Problem

- You are given an array of n integers. Your goal is to partition the array into subarrays of size k (except for possibly the first and last subarray) such that as many subarrays as possible have positive sum. Though $n \leq 3 \cdot 10^4$, k can only take on 10^3 distinct values.

Initial Observations

- Because k can only take on a small number of values relative to n, this hints at brute-forcing all possible valid values of k.
- If we precompute prefix sums - specifically $f(i)$ is the sum of the first i integers in the array for $0 \leq i \leq n$, we can compute the sum of all elements in an arbitrary subarray in $\mathcal{O}(1)$ time. Specifically, the sum of the subarray starting at index a and ending at index b is exactly equal to $f(b+1) - f(a)$.

Business

Solution

- For a fixed starting point and a subarray size k, we can compute the number of subarrays with positive sum in $\mathcal{O}\left(\frac{n}{k}\right)$ time.
- Checking all k possible starting points for a subarray of size k therefore takes $\mathcal{O}(n)$ time.
- Checking all possible values of k, this algorithm therefore runs in $\mathcal{O}(nk)$ time.

Cookie Production

Problem

- You want to make a chocolate chip cookie. In a given turn, you add some cookie squares to your existing cookie. A given square can only be added if it is on the boundary of the cookie or if some adjacent square is not yet filled with a cookie. Compute the minimum number of turns needed to construct the chocolate chip cookie.

Initial Observations

- It seems difficult to know which squares we can fill in first.
- However, if we consider the last turn, we know which squares cannot be filled in on the last turn - any square which is surrounded by cookie on all four sides must be filled in prior to the last turn.
- Therefore, we can consider the reverse process of “eating" the cookie in the minimum number of turns, where a cookie square can be eaten if it is on the boundary or some adjacent square is empty.

Cookie Production

Soluton

- We can solve this other problem using breadth-first search. All squares that are on the boundary or have some adjacent square empty are initialized to a turn counter of 1, and all other squares are set to a turn counter of infinity.
- We maintain a queue of squares we are processing, initialized with the squares that have a turn counter of 1.
- Remove a square from the queue, and if any adjacent squares have a turn counter of infinity, update the turn counter to 1 more than the current turn counter, and append that square to the queue.
- The answer is the maximum turn counter over all squares.

Delicious Waffle

Problem

- A waffle maker rotates 180 degrees every r seconds. A blueberry waffle is inserted with the blueberries pointing up. After f seconds, the waffle maker stops and rotates strictly fewer than 90 degrees back to horizontal. Are the blueberries pointing up or down?

Solution

- The waffle maker makes a full rotation every $2r$ seconds. Therefore, we can take f modulo $2r$. If f is less than $\frac{r}{2}$ or greater than $\frac{3r}{2}$, then the blueberries are pointing up. When f is equal to $\frac{r}{2}$ or $\frac{3r}{2}$, which is not allowed in the problem, the waffle maker is exactly vertical. When f is greater than $\frac{r}{2}$ and less than $\frac{3r}{2}$, the blueberries are pointing down.

Effective Management

Problem

- You are given a ternary array. You are to construct a ternary array where all 0's are contiguous, all 1's are contiguous, and all 2's are contiguous. Maximize the number of indices where your constructed array matches the given array.

Solution

- There are $\mathcal{O}(n^2)$ different ternary arrays you can construct, so checking all of them is too slow.
- However, if we construct our ternary array from left to right, the only information that matters is what integers have been used so far in our constructed ternary array and what the last added element is.
- Therefore, with dynamic programming, we can maintain the maximum number of integers we can match conditioned on having assigned the first i integers, the set of ternary integers we have used so far, and what the ith integer in our ternary array is.

Friends Visit

Problem

- Given $n \leq 1\,000$ days, the amount of mess is increased by m_i each morning, and you can clean c_i amount of mess in the afternoon, determine the minimum number of afternoons you have to spend cleaning so that on d queried nights the mess is zero.
- Divide the days into segments in which the family visits on the last day. Each segment is then an independent subproblem of the same type. In each segment, the mess should be zero by the end of the last day.

Friends Visit

Solution 1: Greedy

- On the last afternoon, if the mess is greater than zero, then you must clean on the last day; if the mess is already zero by then, you don't have to clean on the last day and can save an option of cleaning on the last day to remove mess created on previous days.
- Now consider a previous afternoon when the mess is greater than zero, you will have an option to clean on that day, along with all the cleaning options that you saved for the following days. Among those options, you should pick the cleaning with the largest c_i, until the mess becomes zero.

Friends Visit

Solution 1: Greedy

- This yields a greedy solution working backwards: Initialize an empty cleaning option set S and total mess $t = 0$. For each day in reverse, add m_i to t and c_i to S. If $t > 0$, pick the largest values from S to reduce t to zero. The number of values picked corresponds to the number of afternoons spent cleaning.
- If we maintain S using a BBST or a heap, this greedy algorithm runs in $O(n \log n)$. The low constraints of the problem also allows you find the max value from S in linear time, so that $O(n^2)$ also passes.

Friends Visit

Solution 2: DP

- Let $f(i, k)$ be the max amount of mess we can have starting on day i, such that we can clean k times in the following days and have no mess by the end of the last day. Assume the last day is day n.
- $f(i, k) < 0$ means it's impossible to clean all mess by day n. In terms of arithmetics we treat any negative value as negative infinity.
- We have:

$$f(i,k) = max \begin{cases} f(i+1,k) - m_i & \text{if } i < n \\ & \text{(don't clean on day } i\text{)} \\ f(i+1,k-1) + c_i - m_i & \text{if } i < n, k > 0 \\ & \text{(clean on day } i\text{)} \\ c_i - m_i & \text{if } i = n \text{ and } k > 0 \\ -m_i & \text{if } i = n \text{ and } k = 0 \end{cases}$$

Friends Visit

Solution 2: DP

- Find the smallest k as our final answer such that $f(1, k) \geq 0$. This can be done by iterating k incrementally.
- There are $O(n^2)$ DP states in total and the transition takes constant time. Therefore the DP solution runs in $O(n^2)$ time and space.

Giant Walkway

Problem

- Given a list of strings and multiple pairs of strings, compute for each pair of strings how many strings are between the paired strings.

Solution

- It is too slow to do a linear scan for each string for each query pair.
- Instead, we maintain a map from string to the index it is at in the list.
- We then output $|a - b| - 1$, where a and b are the indices of the two strings in the pair.

Hunting The Eclipses

Problem

- The sun and the moon align for an eclipse occasionally. It was d_s years ago when the sun was last in the right place, and d_m years ago when the moon was last in the right place. The sun is in the right place once every y_s years, and the moon is in the right place once every y_m years. When will the next eclipse happen?

Solution

- We are guaranteed that an eclipse will happen in the next 5000 years.
- Therefore, we can check the years starting from one year in the future and check if the sun and moon will be in the right place - y is a valid year for an eclipse if $(y + d_m)$ is divisible by y_m and $(y + d_s)$ is divisible by y_s.
- There is a faster solution using the Chinese Remainder Theorem, but this was not required to solve the problem.

Interesting Puzzle

Problem

- You have $n+1$ tubes each with the capacity to hold three balls. There are $3n$ balls distributed among the tubes, three balls each of n distinct colors. In a single move, you can take a ball from one tube and move it on top of all the other balls in a tube that has fewer than three balls in it. In $20n$ moves or fewer, get all tubes to be either completely empty or have all three balls of some color.

Solution

- There are many different approaches to get this to happen within $20n$ moves. We'll outline one approach that fills in the left n tubes. This solution will operate in multiple phases.

Initialization

- We start by emptying the rightmost tube, arbitrarily moving balls from there into tubes to the left that have space. This takes at most three moves.
- We proceed by making tube 1 be monochromatic, at which point future moves will not interact with it at all. We need to be able to perform this in fewer than 20 moves due to the overhead we incurred.

Making the Leftmost Tube Monochromatic

- Let the bottom ball in the leftmost tube have color c. We will move all balls with color c into this tube.
- If the tube is already monochromatic, we're done.
- If the topmost ball has color c and the middle one doesn't, we can reverse the two balls as follows:

Making the Leftmost Tube Monochromatic, continued

- Let the leftmost tube be l, the rightmost tube with balls be r, and the empty tube be e. Move a ball from r to e, the topmost ball with color c into e, the middle ball from l to r, the topmost ball with color c from e to l, and the last ball from e back to l. This takes five operations.
- Now, it remains to move balls from other tubes into the leftmost tube.
- If such a ball is not the bottom-most ball in its tube, we can remove the incorrect balls out of tube l into e, any balls above that ball into e, and then move that ball directly into l. Moving all balls back into e, this takes at most seven moves to fix one ball.
- If such a ball is the bottom-most ball in its tube, we can reverse the entire tube by moving all balls into tube e, at which point we can apply the above logic to move balls out of l until we can take the (now topmost ball) from e and move it into l. This takes at most eight moves.

Juggle With Dice

Problem

- You are given a list of three-letter words. Is it possible to construct three dice such that, for each word, it is possible to arrange the dice in such a way that the top faces can form the word? All 18 possible letters on the three dice must be distinct.

Initial Observations

- If a word has two or more identical letters, it is impossible.
- If 19 or more distinct letters appear over all words, it is impossible.
- If fewer than 18 distinct letters appear, we can pick arbitrary unique letters that do not appear to fill in the other faces.
- If letters α and β appear in the same word, they must appear on different dice.

Juggle With Dice

Solution

- If the faces and dice are all distinguishable, there are 18! ways to arrange the letters.
- The faces of a die are indistinguishable before adding letters, so we can divide out a factor of 6!.
- The dice are also indistinguishable before adding letters, so we can divide out a factor of 3!.
- This leaves us with $\frac{18!}{(6!)^3 \cdot 3!} = 2858856$ combinations to try.
- We can use recursive backtracking to enumerate and try all of these, pruning when an assignment is clearly invalid.
- Though not necessary to solve the problem, we can recursively try assigning the letters that have the most constraints first to prune the search space.

Kitchen

Problem

- You have n different blades. Blade i can cut pieces of size at most m_i, cutting them in half in h_i seconds. Blades reduce the size at an exponential rate. Compute the minimum number of seconds needed to convert food that is originally size t to size s.

Solution

- For a given piece size, we want to use the blade with the minimal h_i rate. We can ignore blades where $m_i \leq s$ or $m_i > t$.
- We need to be able to solve the equation $t \cdot 0.5^{\frac{x}{h_i}} = s$ for x. Taking logarithms, we can show that $x = \frac{h_i \cdot \log\left(\frac{t}{s}\right)}{\log 2}$.
- We need to reevaluate the best blade for all m_i values in $[s, t]$. We can do this by maintaining the blades sorted by their m_i values. It is too slow to enumerate all eligible blades for each check.

Logistics Manager

Problem

- In a rooted tree, people navigate through the tree by always traveling to the descendant with the lowest ID. n people start at the root and wish to get to specific destinations, traveling through the tree in order. Before each person starts traveling, you can permanently delete some edges from the tree. Compute the index of the first person who cannot make it home.

Initial Observations

- Use the Euler tour technique to represent the tree. Specifically, DFS through the tree in sorted order of children. Let s_v be the time when we first see vertex v in the DFS, and let e_v be the time when we return from vertex v in the DFS.
- We are therefore looking for the first vertex v where there exists a vertex u appearing before v in the destination order list where $e_v < s_u$.

Logistics Manager

Solution

- If we compute the Euler tour of the tree, we can simply loop over the destination vertices in order, track the maximum s_v we have seen, and see when some e_v is less than the maximum e_v seen prior.
- Note that it is not strictly necessary to compute the Euler tour beforehand and then loop over the destination vertices in order. We can perform a preorder traversal of the tree. Prior to returning from the recursive call from a vertex v, we can visit any vertex that is in the call stack of the DFS, so we can loop over destination vertices until we see one we cannot visit.

Making The Palindromes

Problem

- You are given a string of lowercase letters. In a single operation, you take two adjacent characters and mutate both of them. Compute the minimum number of operations needed to make the string a palindrome.

Initial Observations

- If the outermost characters match, neither should be changed.
- If the outermost characters do not match, it is not always optimal to make them match with one operation! The sample case `vetted` shows this - we need more than 2 operations if we make the outermost characters match, but we can do 2 operations by doing `vetted` to `gutted` to `guttug`.

Making The Palindromes

Solution

- We can solve this with dynamic programming. We can reduce this problem to the following - you are given a binary string where in a single operation, you look at two adjacent indices - a 1 must be flipped to a 0, whereas a 0 can stay as either a 0 or be changed to a 1. Your goal is to make the string be all 1's.
- To solve this problem, you can maintain for a state of the form (length of prefix that is all 1's, whether the next bit has been forcibly flipped) the minimum number of operations needed to get to that state.
- To convert the original problem to this reduced one, construct the binary string from left to right by looping over pairs of characters in the original string from the outside going to the middle, adding a 1 if the characters match and a 0 if they don't.