The 2021 ICPC Mid-Atlantic USA Regional Contest

Editorial

February 26, 2022

A Espresso!

The problem can be solved by simulating the serving of orders in $O(n)$ time. Use a variable to keep track of the current amount of water in the tank. Whenever there is not enough water for the next order, reset the water amount to a full tank and increment the answer by one. Do not forget to deduct the water amount consumed by the current order after refilling (a common mistake).

B Ultimate Binary Watch

The problem can be solved by decomposing each of the four digits into their binary forms. This can be done using bitwise operation, e.g. the condition `((1 << 3) & digit) > 0` can be used to check if a digit has the bit for $2^3 = 8$ in most programming languages. To render the watch face, you may allocate a 2D array of characters and then fill in the character each bit. Be careful not to add extra spaces. Make sure to compare your program's output against the sample output with a check on the spaces.

C Concert Rehearsal

This problem cannot be solved by simulation because there can many students ($n \leq 2 \cdot 10^5$), the number of days and the recital hall's availability on each day can be big numbers ($p, k \leq 10^9$). Simulating the whole process can take up to $O(pq)$ time and will not finish in time.

A key observation is that if a day starts with student i's performance, then it will always ends with a fixed student's performance. Let that student be $end[i]$. We shall compute $end[i]$ for every student first. Again, we cannot simulate for each student, which would take $O(np)$ time and is too slow. Instead we can precompute a prefix sum array of the performance times and then do binary search ($O(n \log n)$) or two pointers ($O(n)$) on it to compute $end[i]$ more efficiently.

It is tempting to simulate the process using $end[i]$, which however is $O(k)$ and still too slow. A second observation is that under the Pigeonhole Principle, after $n + 1$ days we are sure to encounter two days that start with a same student's performance. This means that the process enters a cycle. We can compute the length of the cycle and divide k by the length to get the number of rehearsal passes in all these cycles in $O(1)$ time. Any remainder days can be simulated in $O(n)$.

D Land Equality

This problem can be solved by a small number of case work:

- If there are two zeroes, then we can give each descendant one zero so that their land prosperity difference is zero.
- If there are no zeroes, then the two descendant's land will both have prosperity values that are powers of two, say 2^a vs. 2^b. We want to minimize $|a-b|$, and must divide the twos as evenly as possible. Be careful about integer overflow, as the maximum answer is 2^31.
- When there is a single zero, we shall further divide the cases:
  - If there are no twos, then all non-zero cells contain a 1. So one descendant will get 0, and the other will get 1.
  - If there are no ones, then all non-zero cells contain a 2. We shall give a 2 to one descendant, and all the remaining cells to the other descendant (including that zero). The difference is 2.
  - When there are both ones and twos, this is the trickiest case. A common mistake is to think that we can give a 1 to one descendant, and the other descendant gets everything else (including the zero). A breaker case is a board of size 1×3: `0 1 2`, in which there is no way to give that 1 to one descendant without separating the remaining cells into two connected parts. This can be handled by checking if the board has at least two rows or two columns, in which case it is always possible to give 1 to one descendant as the remaining cells will guarantee to be connected. If the board has only one row/column, it can be seen that it is only possible to give 1 to one descendants if that 1 is the first/last cell of the row/column.

E Even Substrings

We can compute a prefix letter parity mask $mask[1..n]$ that has 6 bits: the jth bit in $mask[i]$ is set when the prefix $s[1..i]$ has an odd number of the jth character. To answer how many substrings in $s[l..r]$ are even, we can iterate all $2^6 = 64$ masks and count for how many times each mask m appears in $mask[l-1..r]$. A mask m that appears c times adds $c(c-1)/2$ to our answer. To add those counts efficiently, we can use a segment tree. In each segment tree node, we maintain how many times each of the 2^6 masks appear in the range covered by that node. Modifying a character $s[i] = x$ can be handled using lazy propagation, as we are changing all the masks in $mask[i..n]$. The total running time is $O(2^6(n+q)\log n)$.

F Bracket Pairing

The problem can be solved in two different ways.

Enumeration. Observe that there are only $4^{10} \approx 1M$ choices in the left and right half of the sequence. We can enumerate those choices in each half separately, and then meet their results in the middle. We only care about the brackets that are still open/close in the enumeration of the left/right half. This gives us a solution of $O(4^{10})$.

Dynamic Programming Let our bracket sequence be $s[1..n] (n \leq 20)$. Let $f(i, j)$ be the number of ways to fill in the brackets in $s[i..j]$ so that $s[i..j]$ is a valid bracket sequence itself. We can try to cut $s[i..j]$ into two parts that are both valid bracket sequences themselves. We have $f(i, j) = \sum_k f(i + 1, k - 1) \cdot match(i, k) \cdot f(k, j)$ where $match(i, k)$ counts the number of ways to match $s[i]$ with $s[k]$. The DP takes $O(n^3)$ time.

G Parking Lot

The problem can be modeled as a shortest path problem where a pair of grid vertices has an edge that is the segment that connects these two vertices. Let $n = \max(r, c)$. There are $O(n^4)$ edges in total. An edge is invalid and shall be discarded if it cuts through a cell that contains a parked car. Checking if an edge is invalid can be done in $O(n)$ time by iterating each row (or column) that the segment intersects with and finding the intersection points. The graph construction thus takes $O(n^5)$. The problem then can be solved using Dijkstra's algorithm for weighted shortest path in $O(n^4 \log n)$ time. Dynamic programming also works here in $O(n^4)$ given that we will never take an edge that goes to the bottom-left or the top-right regions of our current location.

H Subprime

We can first compute the first $h = 10^5$ primes using Sieve of Eratosthenes in $O(h \log \log h)$ (the largest of them is 1 299 709). Then we can iterate over each prime in the range, and perform a straightforward substring check. Checking one prime takes $O(\max(7, |p|))$ as the primes have at most 7 digits.

I Word Puzzle

The blanks in a word puzzle is a subsequence of the puzzle word. The typed letters must be a rotation of the subsequence. We can first find all the distinct rotations of the typed string s. Since $|s|$ is at most 50, this can be done in $O(|s|^2)$ easily.

Then for each unique subsequence u obtained from those $|s|$ rotations, we count how many subsequences of p equal it. This can be done using standard dynamic programming. The DP state is $f(i, j)$ which is the number of ways to have $u[1..j]$ as a subsequence of $p[1..i]$ and $u[j] = p[i]$. The DP takes $O(|u||p|)$ time for each u. The total time of the solution is $O(|s|^2|p|)$.

J Code Guessing

This problem can be solved by case work. Yet case work is a bit error prone and it is easy to mishandle one of the six cases. A safer alternative is to enumerate what digits Bob has (only 21 possibilities), sort the digits, and counts how many ways can the enumerated Bob's digits match the given sorted positions. If the number of ways is exactly one, then Alice can uniquely determine Bob's digits.

K Tree Number Generator

We can use doubling preprocessing to solve this problem. For each node i and each power of two 2^k, we preprocess the digit concatenation modulo m for the first 2^k nodes along the path from node i to the root. We will need the digit concatenation in both ways: upwards and downwards. While we are constructing the doubling results, we can also preprocess doubling node parents/ancestors for later computing Lowest Common Ancestor (LCA). Preprocessing takes $O(n \log n)$ time.

For each query $Get(a, b)$ we first compute the $\text{LCA}(a, b) = c$. The digits from a to c are concatenated upwards, while the digits from c to b are concatenated downwards. These concatenation results modulo m can be derived from our preprocessed doubling results in $O(\log n)$ time. All queries can be answered in $O(q \log n)$ time.

L Lone Rook

This problem can be solved in a BFS fashion. We maintain a queue to process the cells reachable by the rook. A cell is reachable if a rook can eventually move to it safely after removing all its attacking knights. We preprocess for each cell how many knights attack it as $attacked[i][j]$. Whenever a reachable cell c is popped from the queue, we do the following:

- If c has a knight, then we remove that knight and reduce $attacked[i][j]$ of those cells attacked by this knight. If any of them, say c', becomes no longer attacked, then go in four directions from c' to find a cell reachable by rook. If such a cell exists, then we add c to the queue. This step takes $O(8 \cdot 2(r + c))$ time.
- Go in four directions and add all cells to the queue until we hit a knight. If this knight is attacked, we stop. Otherwise we add the knight's cell to the queue and stop. This step takes $O(2(r + c))$ time.

Every time we add a cell to the queue, we mark that cell as reachable. Each cell will be added to the queue exactly once. The total running time is $8 \cdot 2(r + c)rc$.

M Stream Lag

First we sort the packets by their ID in $O(n \log n)$ time. Then we can calculate the lag in $O(n)$ time by iterating the sorted packets and maintaining the current time t_{now}. Initially $t_{now} = 0$. When the next packet arrives at t, it introduces a lag of $\max(t - t_{now}, 0)$, and t_{now} becomes $max(t_{now}, t) + 1$ after playing it.