Noctem Virtual II Editorial

Duck Escapade

Solution:

Observe that if X ducks are able to make it to the other side, then they might as well be the X ducks with highest starting energy. As a result, we can sort ducks by their starting energy and then greedily use quacks on each duck starting from highest energy to lowest energy until we run out.

Code

Solution Credits: Marco Frigo

Social Distancing

Solution:

Sort the people by x-coordinate from smallest to largest, then perform a left to right sweep. If at any point there are 2 people within 6 feet of another, then you know one of them has to go. Since we are sweeping left to right, it's intuitive to always remove the person to the right so we can leave more space for the future.

Code

Solution Credits: Marco Frigo

Whitejack

Solution:

Notice that if player 2 has any of the cards $K - a_1, K - a_2, \ldots, K - a_N$, then they win. Therefore we need a way to figure out if any given subarray of array B contains any of these cards in order to answer the queries. To do so, we can use prefix sums on array b. For any card that is one of $K - a_1, K - a_2, \ldots, K - a_N$ in array b, we can mark it with a 1, otherwise 0. Then we can keep track of prefix sums which will allow us to get the range sum of any range. Observe that if the

range sum of a subarray is greater than zero, then it is winning, otherwise it is not. This allows us to answer queries in O(1) time.

Code

Solution Credits: Steven Tan

Evacuation

Solution:

If we can find the maximum number of reachable nodes X, then the answer is ceil($\frac{M}{X}$). To find the maximum number of reachable nodes, we can use recursion. The maximum number of reachables nodes from any node can be found through the max reachable nodes from each of its children, as we can just take the K children with most reachable nodes as the ones we distribute to. Then, our base case is a leaf node, which has one reachable node. Using this method, we are able to find the max reachable nodes from node 1, and then our answer is just ceil($\frac{M}{X}$).

Code

Solution Credits: Steven Tan

Cleaning Windows

Solution:

Let's brute force by the highest window Jerome cleans, and for each highest window find the most windows Jerome can clean. For each highest window he cleans, if he cannot clean all windows he goes through in time, then he can just skip the dirtiest windows until he is able to clean them in time. This will result in finding the most windows he is able to clean at a given height because he skips the dirtiest ones, which will leave the most time for cleaning the rest. However, if we just do this naively, it takes $O(N^2)$ time. To do this faster, we can observe that if any window was skipped at an earlier height, then that window will be skipped for all future heights as well. This means we can keep a sorted set of all windows we will clean and for each height we will just remove the highest values in the set until we can clean in time. This results in a O(N log N) solution.

Code

Solution Credits: Marco Frigo

Word Game

Solution:

To solve this, we can use casework.

Case 1: Length of K is Odd

In this case, as long as K appears in S as a substring, player 1 wins. This is because player 1 can pick the middle character of K, and then whatever player 2 does player 1 can just do the opposite (i.e player 2 expands left, then player 1 expands right). This will always result in player 1 winning. On the other hand, if K does not appear in S, then obviously it is a draw. We can find substring occurrences of K in S efficiently using hashing or the knuth-morris-pratt algorithm.

Case 2: Length of K is Even

Since K is of even length, there are now two "middle characters" (i.e "b" and "c" are the middle two characters of "aabcdd"). Note that for any occurrence of string K in S, if player 1 picks any of the middle two characters of K then he loses, because player 2 can perform a similar tactic of extending the opposite direction until he wins. Let's call any such character a "forbidden" character. This brings us to a claim: Player 2 only wins if all characters in string S that are not the leftmost or rightmost $|K|/2$ characters are forbidden, otherwise its a tie. This is because if all of those characters are forbidden, then no matter which character player 1 picks first, he loses. If he picks a forbidden character, he loses, and if he picks any of the leftmost/rightmost $|K|/2$ characters, he also loses because player 2 can essentially force him to the edge and ensure that string K is created. On the other hand, if any character is not forbidden, then player 1 can just pick that character and perform the same tactic of extending in a opposite direction to ensure a tie. Forbidden characters can be found through the same substring occurrence finding as we do in case 1, which solves the problem.

Code

Solution Credits: Steven Tan

Colored Edges

Solution:

An optimal solution will never require over N toggles, will only involve toggling each node at most once, and the final walkable path created should also be a simple path (if there were any loops, then you can always just avoid the loops and get an equal or better solution). This allows the use dynamic programming on the graph in a way similar to bellman ford. Let dp[i][j] = minimum number of toggles required to create a walkable path from node 1 to node i, with node i being toggled or not (j = 0 means not toggled, j = 1 means toggled). The reason this dp works is because at any given state we only care about how we toggled the most recent node; previous toggles and nodes are irrelevant because we know an optimal solution will never toggle or interact with them again. Here are the transitions (let x be a neighbor of node i, and we will be performing these transitions for all neighbors of i):

if edge from x to i is white:
dp[i][0] = min(dp[i][0], dp[x][0])
dp[i][1] = min(dp[i][1], dp[x][1] + 1)

if edge from x to i is black:
dp[i][1] = min(dp[i][1], dp[x][0] + 1)
dp[i][0] = min(dp[i][0], dp[x][1])

Each time we run through this dp array we are essentially "relaxing" all the edges and finding paths that are one longer. We therefore need to perform this run through N times to find the optimal solution. Complexity is O(N * M), as one run through is O(M) and we perform that N times.

Code

Solution Credits: Steven Tan

Banana Factory

Solution:

A naive solution is to bash all subsets of bananas and place one subset on one conveyer belt and the rest on the other conveyer belt. This is however too slow, with a complexity of 2^N. However, observe that at any given point if we are considering which conveyer belt to place banana i on, we only care about the last k - 1 bananas placed on the conveyer belts because any earlier bananas are irrelevant to the current state (those bananas have already gone off the conveyer belt). This vastly reduces the number of states, as instead of 2^N, we only need to store how we split the past K - 1 bananas, therefore only using 2^{K-1} states. This means that we can use dynamic programming to solve this problem, with the state storing which banana we are putting on the conveyer belt and

a bitmask storing how the previous K - 1 bananas were placed. The complexity is $O(N * 2^{K-1})$, which is fast enough.

Code

Solution Credits: Steven Tan

Phone

Solution:

First observation: Since operations on one digit are independent of other digits, we can split the problem into finding the expected number of operations of turning each digit in X to the corresponding in Y, and then summing all of them up. So now we just need to figure out the expected number of operations to turn any digit A into any other digit B. Next observation is that we don't care what letter A and B are themselves, just the "distance" between them. This means that we just need to find expected operations to turn a digit into another that's 0, 1, 2, 3, ... , 9 distance away from it. To do this, let's set up some variables $E_0, E_1, \ldots, E_9$, where E_i is the expected number of operations it takes to reach a digit i distance from it. We can set up a linear system of 10 equations that looks like this:

$E_0 = 0$
$E_1 = \frac{1}{k}(E_1 + 1) + \frac{1}{k}(E_0 + 1) + \frac{1}{k}(E_9 + 1) + \ldots + \frac{1}{k}(E_{(2-k) \mod 10})$
$\ldots$
$E_9 = \frac{1}{k}(E_9 + 1) + \frac{1}{k}(E_8 + 1) + \frac{1}{k}(E7 + 1) + \ldots + \frac{1}{k}(E_{10-k})$

To solve this system, we can use Gaussian Elimination, or Cramer's Rule which finds the determinant recursively. Cramer's Rule is slower, but it still works as the system is relatively small.

Code

Solution Credits: Steven Tan