2023 Pacific Northwest Division 1 Solutions

The Judges

Feb 24, 2024

ABC String

Problem

- You are given a string with n `A`'s, `B`'s, and `C`'s. Partition the string into the minimum number of subsequences such that each subsequence is the concatenation of several permutations of `ABC`, not necessarily the same permutation in each instance.

Initial Observations

- Note that if the first three letters of a string are some permutation of `ABC`, then if we partition the remainder of the string optimally, we can always prepend the first three characters to an arbitrary subsequence.
- Therefore, we can greedily assign characters to subsequences looping over the characters in order.

ABC String

Solution

- If there is an active subsequence of length 2 that does not contain the character we are currently considering, assign the current character to that subsequence and set that subsequence to be empty.
- Otherwise, if there is some subsequence of length 1 that does not contain the character we are currently considering, assign the current character to that subsequence.
- Otherwise, create a new subsequence consisting of just that character.
- Since we always reset the subsequence lengths when they reach 3, this algorithm runs in $\mathcal{O}(|s|)$.

Acceptable Seating Arrangements

Problem

- You are given a grid of integers of size at most 20×20 where each row is sorted in increasing order. In a single operation, you can pick two elements and swap them, as long as all rows are still sorted. Given a final configuration where all rows are sorted, get from the initial configuration to the final configuration in at most 10^4 operations.

Initial Observations

- Let $r = c = 20$, and note that $20^3 < 10^4$. If we can place a single element in the correct location in at most c operations, we can place all elements in $rc^2 < 10^4$ operations.
- We therefore try to place the elements in the correct locations in increasing order of value.

Acceptable Seating Arrangements

Solution

- Let the current element we are trying to place correctly be x. Inductively, all elements less than x are correctly placed.
- If x is in the same row in both configurations, then x must be correctly placed already, by the induction hypothesis.
- Otherwise, by the induction hypothesis, x must go to the leftmost column in the row of the final configuration that is not already filled, let that element be called y.
- The only scenario where that is not directly possible is if y is greater than the element currently to the right of x.
- However, in that case, we can swap y with the largest element in the row where x currently is that is less than y.
- We repeat the above until x can be placed in the correct place. This must happen after at most c operations.

Candy Factory

Problem

- You have n factories that each produce some number of candies. A bag of candies is valid if it contains exactly k candies, and no two candies came from the same factory. Each factory has already produced some candies, and you wish to bag all the candies such that all bags are valid. You can order any factory to produce additional candies. Compute the minimum number of additional candies over all factories that need to be produced such taht all produced candies can be put into valid bags.

Initial Observations

- If it is possible to produce additional candies such that exactly b valid bags can be formed, it is also possible to produce additional candies such that exactly $b + 1$ valid bags can be formed. Therefore, we can binary search for the answer.

Candy Factory

Solution

- We wish to identify if b bags can be formed.
- Two clearly necessary conditions are that $b \cdot k$ must be greater than or equal to the number of candies produced, and that no factory produced more than b candies.
- We leave as an exercise to the reader the proof that these two conditions are sufficient.
- This algorithm runs in $\mathcal{O}(n \log n)$.

Alternate Solutions

- Because n and k were small, a fast simulation based on the above that ran in $\mathcal{O}(nk \log n)$ could pass.
- It is also possible to derive a closed-form answer using the above observations.

Cramming for Finals

Problem

- You are given a binary grid where n entries are 1. Compute, over all entries that are zero, the minimum number of 1 entries that are Euclidean distance at most d away from the given zero entry.

Initial Observations

- n and d are small relative to the grid size. If we can get an answer of zero, we can immediately short-circuit.

Cramming for Finals

Solution

- We start by doing a line sweep. For a fixed row, compute for each 1 entry which columns in the given row it affects. Since these columns are contiguous, we can efficiently compute this by first maintaining a difference array and then computing a running sum over the difference array, taking the minimum over the difference array over all entries that are zero.
- A given circle has $\mathcal{O}(d)$ interesting points along its circumference, so this line sweep only needs to consider $\mathcal{O}(nd)$ intervals. Due to the difference array, some sorting is required internally, so this algorithm runs in $\mathcal{O}(nd \log n)$.

Eccentric Excursion

Problem

- You are given a tree of n vertices. For a given k, compute the lexicographically smallest permutation of vertices such that exactly $n - k - 1$ pairs of adjacent vertices in the permutation are connected by an edge.

Initial Observations

- For a tree, if l is the fewest number and r is the largest number such that exactly l and r pairs of adjacent vertices in some permutation could be connected by an edge, then all intermediate values are attainable.
- l is almost always 0. Otherwise, it is 1.

Eccentric Excursion

Solution

- We will incrementally build the permutation element by element. Given the permutation so far, we check to see if appending the current element would still be valid. To compute r, there are multiple ways to compute the maximum matching. This can be done with a tree DP or a greedy assignment with a precomputed postorder traversal of the tree.
- We need to carefully check if $l = 1$, which requires some casework based on the previous node and whether the rest of the tree is a star.
- Since checking that a node is valid takes $\mathcal{O}(n)$ time, this algorithm runs in $\mathcal{O}(n^3)$ time.

Item Selection

Problem

- You are given a UI with some items initially selected. You can click between pages, toggle individual selections, select all items on a page, or deselect all items on a page. Compute the minimum number of clicks needed to get to a final configuration of items selected.

Initial Observations

- In terms of moving between pages, it is never optimal to change direction more than once. Therefore, one should either move left and then move right, or move right and then move left. It is possible that the movements are empty.
- On a given page, we should only use at most one of the select all and deselect all options.

Item Selection

Solution

- For each page, we do some casework to compute the minimum number of button clicks based on whether we use select all, deselect all, or neither.
- We then keep track of all pages that require changes. We compute the minimum number of clicks needed to navigate pages by either going left and then going right, or going right and then going left.

Matrix Fraud

Problem

- You are given a binary matrix and the ability to toggle individal entries. Compute the minimum number of entries to toggle such that each row and each column have a 1, and within each row and each column, all 1 are contiguous.

Initial Observations

- If we iterate over the rows in order, we can maintain a DP that maps the exact interval that contains 1's to the minimum number of operations needed to get exactly that interval to contain 1's.
- $r \times c \leq 2 \cdot 10^5$, which means that at least one of r and c must be less than or equal to $\sqrt{2 \cdot 10^5}$.

Matrix Fraud

DP formulation

- Our DP tracks $f(r_i, c_l, c_r)$ which is equal to the minimum number of operations needed such that the matrix up to row r_i has at least one 1 in every column from column 1 to column c_r and one 1 in every row from row 1 to row r_i, and row r_i contains 1's exactly on the interval $[c_l, c_r]$.
- We can transition from $f(r_i, c_l, c_r)$ to $f(r_i + 1, c_l', c_r')$ if and only if $c_l' \geq c_l$ and $c_r' \geq c_r$. There are $\mathcal{O}(rc^2)$ states and $\mathcal{O}(c^2)$ transitions per state, so the DP currently runs in $\mathcal{O}(rc^4)$.

Matrix Fraud

DP formulation

- Assume without loss of generality that $r \geq c$ - transposing the matrix and solving this problem on the transposed matrix is equivalent.
- This observation gets us down to $\mathcal{O}(r^2 c^2 \min(r, c))$ which is still too slow.
- If we look harder at the transition for $f(r_i + 1, c'_l, c'_r)$, we see that the transition uses the 2D prefix minimum for $f(r_i, c_l, c_r)$. If we compute these prefix minima, we can optimize the transition down to $\mathcal{O}(1)$. The number of states is now $\mathcal{O}(rc \min(r, c))$, and since $\min(r, c) \leq \sqrt{2 \cdot 10^5}$, this is fast enough.

Oncall Team

Problem

- You have a team of n engineers, each of who is familiar with some of m services. A team has a *robustness level* of k if, no matter which k services go down, k different engineers can each be assigned to one of the services such that each engineer is familiar with their assigned service. Compute the team's robustness level.

Initial Observations

- m is much smaller than n.
- Hall's Theorem gives us a simple way to check if a bipartite graph has a perfect matching.
- By Hall's Theorem, if the robustness level of a team is k, then there exists some collection of x engineers where $x > k$ and the union of the services they are familiar with is some subset of size k.

Oncall Team

Solution

- For each subset of x engineers, we compute the size of the union of the services they are familiar with. If the size of the union, y, is greater than or equal to x, we get no information on the robustness level. Otherwise, the robustness of the team must be at most y by Hall's Theorem.
- Naively, this runs in $\mathcal{O}(n2^m)$. We can optimize this to $\mathcal{O}(nm + m2^m)$ by short-circuiting when we observe that the size of the union must be at least x.

Range Editing

Problem

- You are given an array of n integers. In a single operation, you can take a subarray and change all entries in the subarray to some other integer. Compute the minimum number of operations needed to get to a specific configuration.

Initial Observations

- If the outermost element of the array has the correct value, it is never optimal to change it.
- This gives us an $\mathcal{O}(n^4)$ DP where, for a fixed interval and the current color of the leftmost element in the interval, we can maintain the mininum number of operations needed to get that state.

Range Editing

Solution

- Our DP tracks $f(l, r, c)$ which is equal to the minimum number of operations needed to get the subarray $[l, r]$ correctly assigned, given that the interval currently has a value of c.
- If l is correctly assigned, then we compute $f(l + 1, r, c)$. Otherwise, we have to assign l to the correct color and also determine which prefix it gets colored with. This DP therefore has $\mathcal{O}(n^3)$ states and $\mathcal{O}(n)$ transitions, which gives an $\mathcal{O}(n^4)$ algorithm which is too slow.
- To optimize this, note that there is no reason to explicitly maintain DP states where c does not match element l, since we know that we are forced to set that element properly.
- Therefore, we do not need to maintain c in our state, reducing the number of states to $\mathcal{O}(n^2)$. This DP runs in $\mathcal{O}(n^3)$ and is fast enough.

Segment Drawing

Problem

- You are given n labeled points p_i and line segments $[l_i, r_i]$ on the x-axis, and you must connect each point to its segment with a newly drawn segment. The newly drawn segments may touch but cannot strictly intersect. Compute the minimum sum of lengths of the newly drawn segments.

Initial Observations

- The collection of points p_i can connect to on the line segment $[l_i, r_i]$ form a contiguous subsegment.
- Therefore, it suffices to compute the leftmost and rightmost points each p_i can connect to.

Segment Drawing

Solution

- Scan over the points from left to right in line segment order, and maintain a stack of points. While the current point is not below the topmost point in the stack, pop that point off the stack and check if the popped element's segment possibly interferes with the current point's segment. If so, update the new leftmost point that the current point can connect to. When done, push the current point on the stack.
- Repeat this process in the reverse order, updating the rightmost point each point can connect to.
- If any point now has an empty segment to connect to, it is impossible. Otherwise, the optimal arrangement follows directly from the given intervals - either draw straight down if possible, otherwise connect to the closest of the boundary points.
- Be careful to use integer arithmetic everywhere.

Sequence Guessing

Problem

- You are asked to generate a sorted sequence of integers starting with 0 ending with 10^5 where adjacent entries differ by either 1 or 2. You tell an adversary how long your sequence is, then the adversary will guess integers in $[0, 10^5]$ and you must tell the adversary either the index of that integer in your list or assert that it is not present. Generate and maintain such a sequence while forcing the adversary to guess an integer that is not present 33333 times.

Initial Observations

- $\left\lfloor \frac{10^5}{3} \right\rfloor = 33333.$
- If you always include every multiple of 3, then there are 33333 pairs of adjacent integers and you can always force the adversary to miss by selecting the other integer in the pair.

Sequence Guessing

Solution

- The sequence will take the form $0, a_2, 3, a_4, \ldots, 99996, a_{66666}, 99999, 10^5$.
- Tell the adversary that your sequence has length 66668.
- If the adversary guesses $3x + 1$, and the adversary has not guessed $3x + 2$, report it is not present and include $3x + 2$ in your sequence.
- Otherwise, if the adversary guesses $3x + 2$, and the adversary has not guessed $3x + 1$, report it is not present and include $3x + 2$ in your sequence.
- Otherwise, the entry must already be in your sequence, report its index accurately.

Training

Problem

- Given a starting integer s and n intervals $[l_i, r_i]$, if your integer currently falls inside the interval, you can increment it or leave it unchanged. Compute the maximum possible value your integer can end up as.

Solution

- It is always optimal to increment the integer when given the chance. We leave the proof of this as an exercise to the reader.
- Therefore, we read in the intervals one by one and increment if the integer falls in the given range.

Triple Sevens

Problem

- Given three sets of digits, determine if every set of digits contains a 7.

Solution

- Due to the structure of the problem, there are multiple approaches to check whether every set has a 7.
- For example, one could parse the integers in each line and look for a 7.
- Alternatively, one could read three lines and check if each line has 7 as a substring.