15295 Spring 2020 #1 Overture -- Problem Discussion
January 15, 2020

This is where we collectively describe algorithms for these problems. To see the problem statements follow this link. To see the scoreboard, go to this page and select this contest.

A. Pots
Count the most occurences of one element. (e.g. using hash)
A famous guy said: “Use conclusions bravely, don’t prove”.
Actually, proving it is easy using greedy. You just find the longest path, and then find the longest path, and then …
---- Yucheng Dai

B. Row and Column Swaps

Maintain a map between original row/col indices and new row/col indices. For example, if we consider m[i][j], and then perform the operation that swaps row i with row k, when we query m[i][j] we access instead m[k][j]. If we encounter a row swap between rows a and b, we swap the row_map values row_map[a] and row_map[b] (same for cols). The mapping start off as the identity map.

(With a fast language (ex. C++) it is possible to brute force this (actually swap rows and columns of the matrix in memory).)

- ben_dover

C. Maximizing the Bitwise AND

D. Uranium Cubes
First, we create a sorted array of points where the cubes are placed. We also precompute the prefix sums of this array. For every set of values, note that the point to which we should move those values to minimize the cost is the median of that set - so we always choose the median to be the point we move everything to. Also, for every valid set from this array, there is a valid set of the same size with lower cost formed from a continuous subarray with the same median (we can do this by “shrinking” the set we choose to a continuous segment by replacing all points that are further away from the median to unused points that are closer to the center).

We can then binary search for the maximum size of a set that has cost less than B. During each iteration of the binary search, we need to see if there exists a set of a particular size that works. For this, consider each point in the array as a possible median and check if choosing the subarray of that size around it meets the cost bound; also check all pairs of consecutive points, with their average being the median, to cover the case of even-sized sets.This takes O(n) time because we've computed prefix sums. The binary search takes O(log(n)) iterations, so the overall runtime is O(nlog(n)).

Alternatively instead of prefix sum we can directly compute costs using a sliding window: for fixed subset of size s, initially calculate the cost at the leftmost subset (a[0] to a[s-1]) directly. Let a[lo] and a[hi] be the bounds of the current window, and let a[mid] be the median. Sliding the window involves setting the median to a[mid+1], subtracting cost of a[lo], and adding cost of a[hi+1]. For odd s the cost changes by a[lo] + a[hi+1] - a[mid] - a[mid+1]. For even s, choosing a[mid] to be the "lower" of the two central values, the cost changes by a[lo] + a[hi+1] - 2 a[mid+1].

- ben_dover

You can also do this in O(n) with the 2 pointers method. After sorting the array we keep track of one interval with a left and right pointer, and slowly increment them forward from the left to the right.

Initially both pointers are at index 0. We increment the right pointer if it doesn't cause the sum of distances to be greater than B, otherwise we increment the left pointer. We're guaranteed to eventually inspect the optimal interval, since we always find the greatest interval for any fixed left endpoint. Overall we do 2n increments so O(n) time.

To update the sum of distances as we change the pointers, note that the point *j* is the median of the interval, and we add/substract the distance of the added/removed point to the median. Trouble occurs when the median moves forward from a_i to a_{i+1}, however we can simply add |a_{i+1} - a_i| to the sum of distances.

Suppose the median for even size subsets is the "lower value". Then the median is only moved when subset size changes from even to odd. Then there is no need to adjust the sum of distances cost, just add/subtract to distance cost relative to new median.

In the O(n) solution above we may have a case where the lo pointer is greater than hi pointer (case where size 1 subset at lo == hi, and a[hi+1] - a[lo] > B). We can leave this or ensure that in this case the median and hi pointer is forced to move as well.
- ben_dover

E. How Big is that Set?

Let’s try to fix the length of the candidate string. Suppose the desired length is strictly smaller than that of the length of the input; it’s clear that the number will then be smaller. So we can solve these cases together, and take care of the case where we form a number of equal length to N separately. Let the number of non-zero characters in the input be C, and let f[k] denote the frequency of character k. Let the length of the string we’re forming be L (where we take L at the moment to be smaller than the size of N). Then it is clear that we need to distribute L - C zeros in our number, so we can take f[0] = L - C. The number of numbers of length L we can form is then (n choose f[0]) * (n - f[0] choose f[1]) * … * (n - f[0] - … - f[8] choose f[9]). Actually, this is not quite right (the problem doesn’t allow leading zeros), but that’s a small technicality that can be taken care of by fixing the first digit as something nonzero and using this idea to choose digits for the remainder. So iterating over L in this way and summing the possibilities will get us all of the possible strings with length smaller than that of N.

Finally, we can use similar ideas to compute the case where the desired length is equal to that of N. We can iterate over the first point at which our desired string will differ from N (and fix some digit in this location j that is strictly smaller than s[j]) and again use the above counting technique to count the ways to distribute the rest of the digits. This ensures our result is lexicographically smaller.

Finally, if you are not using a language with built in support for integers of arbitrary magnitude, binomial coefficients should be computed using the recurrence (n choose k) = (n - 1 choose k - 1) + (n - 1 choose k), and not using any factorials… the reason being that 50! Is significantly larger than what can be stored in a 64-bit integer, but the binomial coefficient (50 choose k) < 2^50, which is perfectly acceptable when computed without factorials.

-- Wassim

F. Number of Components