Hacker Cup 2016 Qualification Round Solutions

FACEBOOK HACKER CUP · TUESDAY, JANUARY 12, 2016

Here are the solutions to Hacker Cup 2016 Qualification Round problems. If you had a rejected solution and want to find out where you went wrong, read on and download the official input and output!

Input / Output: https://www.dropbox.com/sh/yanr824t...

All problems in this round were written by Jacob Plachta.

Boomerang Constellations

For a given central star, one boomerang constellation exists for each (unordered) pair of other stars which are equidistant from it. Therefore, for each potential central star, we can calculate the distances to all other stars (ideally just their squared distances, so that no floating point numbers are required), sort them, and then count the number of stars at each distinct distance away. If there are `x` other stars that are each a given distance `d` away, they can form `x * (x + 1) / 2` boomerang constellations with the central star. This gives us a solution with `O(N^2 log N)` complexity.

High Security

There are a number of ways to solve this problem, such as with dynamic programming or even maximum flow. However, the simplest method is with an `O(N)` greedy algorithm, as described below.

Let's define a "segment" to be a contiguous sequence of empty cells in a single row. If a guard is placed anywhere in a segment, they can see all of the other cells in that segment. For the most part, one guard will be required per segment. However, some segments of length 1 may not require a guard themselves. In particular, each segment can save at most one adjacent length-1 segment in the other row from requiring a guard (since its guard can be placed such that it covers the other segment's single cell). The exception is that two adjacent length-1 segments (one in each row) cannot both save each other from requiring a guard - one guard will still be required to cover both of these segments.

The Price is Correct

For each left index `a`, let `b` be the largest possible right index such that `B[a] + B[a+1] + ... + B[b] <= P` (unless `B_a > P`, in which case there's no such index and we can let `b = a - 1` for convenience). Note that there are then exactly `b - a + 1` valid contiguous sequences with a as their left index (namely, the sequences `a..a, a..(a+1), ..., a..b`).

For each of the `N` potential left indices `a`, the corresponding maximum right index `b` can be found in `O(log(N))` time using binary search, assuming that the prefix sums of the `B` array have been precomputed. Alternatively, we can observe that when a is increased by 1, its corresponding b index is always increased by 0 or more (never decreased). This allows us to achieve a time complexity of `O(N)` by employing the "sliding window" technique, iterating over all values of `a` from 1 to `N` while

also shifting `b` forward from 1 to `N` as appropriate.

Text Editor

When printing the chosen list of words, operations are saved when consecutive words have common prefixes. As such, it can be shown that an optimal solution can always be produced by printing words in lexicographical order.

After sorting all `N` words, we must decide which `K` of them to use. In order to calculate the number of operations required to print a word, we only care about the previous word that was printed. As such, this problem can be solved with dynamic programming, with the state `DP[i][c]` = minimum number of operations required to print `c` words, the last of which was word `i`. For each initial word `i`, `DP[i][1] = L[i] + 1` (where `L[i]` is the length of the `i` th word). The final answer will then be the smallest value `DP[i][K] + L[i]`, for some final word `i`.

As for the transitions, if the previous word printed was `i` and we'd like to print word `j` next (such that `i < j`), we'll need to compute the number of operations required to accomplish this, and then use it to update `DP[j][c+1]` based on `DP[i][c]`. If words `i` and `j` have a longest common prefix of length `p`, then we'll need to delete `L[i] - p` letters, type `L[j] - p` letters, and then print the word. If, for each `i` between 1 and `N - 1`, we precompute `LCP[i]` = the length of the longest common prefix of words `i` and `i + 1` (which can be done easily in `O(L[1] + L[2] + ... + L[N])`, then the required value `p` is simply equal to `min{LCP[i], LCP[i+1], ..., LCP[j-1]}`. As such, each transition can be performed in `O(1)` time, and the total complexity of the DP is `O(N^2 * K)`.

1 Reply

View 22 more comments