15295 Fall 2019 #3 Dynamic Programming -- Problem Discussion
September 11, 2019

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. Interesting Signs
The problem is relatively easy to solve--you have 3 cases (taking 2 letters from the left, 2 from the right, 1 from each side), and you can create 3 subproblems with these letters and recurse on these sub-instances, until it either doesn't work or it does (returning true/false depending on the result). However, one quickly realizes that with large inputs, recursing with 3 subproblems each time wastes runtime and operations.

The way to solve this is by using a memo table--by keeping track of which cases we've seen, we can avoid a lot of repetitive recursing, and therefore our program completes much faster.

PS: If you've taken 15-122, the lab "Fibonnaci has Bad Internet" implements one quite well!
-- 122 ta (jk this is Abi)

B. Hard problem

C. Coloring Trees

We can define Subproblem(t, c, k) as the problem of finding the minimum amount of paint needed to color the trees starting from the t-th tree onwards such that the t-th tree is colored with color c and the beauty of the sequence of trees starting from the i-th tree onwards is k. There are at most 100*100*100 = 10**6 of these subproblems (but actually much less than this because only cases where c <= k <= (n - t) are valid). To compute each subproblem, we want to find
min(min(Subproblem(t + 1, c', k - 1)) + (cost of painting tree t with color c if unpainted), Subproblem(t + 1, c, k) + (cost of painting tree tree with color c if unpainted)), where the inner min is taken over all colors c' != c. Computing each subproblem takes O(k) time, so the overall computation time is about 10**8, which works for C++ (though a better algorithm or more careful implementation might be needed for Python).

-- Ram

D. Bad Luck Island
Let dp[r][s][p] be the probability that we end in a state with nonzero rock and zero scissors and paper, starting with r rocks, s scissors, and p papers. Observe that because of the symmetry of the problem, we can re-run this computation two more times (on cyclic shifts of r, s, and p) in order to find the other two results.

How do we do state transitions? Let us call a matchup good if it is between two things of different types. Let us call a matchup bad if it is between two things of the same type. It is not hard to see that the following recurrence gives state transitions

dp[r][s][p] = P(next good matchup is r and s) * dp[r][s - 1][p] + P(next good matchup is s and p) * P[r][s][p - 1] + P(next good matchup is p and r) * dp[r - 1][s][p]

… and our base cases will (trivially) be dp[r][0][0] = 1 for every positive value of r.

Now, we only need to compute the matchup probabilities in question. Let us focus on the probability that the next good matchup is between r and s (the others are analogous). First, we know the total number of matchups is T = (r + s + p) * (r + s + p - 1) / 2. Next, the number of bad matchups is B = (r(r - 1) + s(s - 1) + p(p - 1)) / 2.
Now it is easy to see that the probability of taking exactly k bad matchups before getting a matchup between r and s (B/T)^k * (r * s / T). If we sum this from k = 0 to infinity (to encompass all possibilities where the first good matchup is between r and s), we obtain 1 / (1 - B/T) * (r * s/T) as our final probability.

The other probabilities are analogous. Substituting these into the recurrence above give us a way to compute state transitions, as desired. Finally, we remark that the time complexity of this algorithm is O(r * s * p), where all constituent variables are at most 100, so this is good enough to pass.
-- Wassim

Nice. But there's a simpler way to do the probability calculation. Just condition the event space to only consider the non-trivial matchups (e.g. rock matching with rock is trivial).

The number of non-trivial matchups is thus x = r*s + s*p + p*r. And the number of ways of getting a (rock,scissors) matchup is r*s. Thus the probability of rock winning in a random non-trivial matchup is r*s/x.
---DS

E. Pashmak and Graph
Let us first sort the edges by weight, and let dp[v] denote the longest increasing path ending at the vertex v. Iterate over all the edges, and upon encountering a directed edge (u, v), we would like to perform the update dp[v] = max(dp[v], 1 + dp[u]); that is to say, we can now consider a path going through this edge from u to v. Observe that because we process these updates in order of increasing edge weight, every update can only consider edges that are no larger than the current edge in weight, so our algorithm will only compute the length of non-decreasing paths.

However, the problem requires strictly increasing paths. How can we modify the above approach to ignore paths that repeat edge weights? One way is to compute all updates for edges of a particular weight at once, and then write them into the dp array after doing so. That way, one edge of some particular weight cannot be used to form a path with another edge of the same weight. This gives an O(E log E) solution (because of the sorting; the dp computation is only O(E) if done carefully).

(An addendum: Since the edge weights are not too large (<= 3 * 10^5), it is actually possible

to solve this problem in O(E), without a sorting step; simply place the edges into buckets based on their weight, and iterate over the buckets in increasing order).
— Wassim

F. Little Pony and Harmony Chest

We know that for all i, b_i can be at most 59; if it were higher than 59, then we could just replace it with 1 and get a lower cost, since a_i is at most 30. So all numbers used in b cannot have prime factors higher than 59. There are only 17 primes up to and including 59. We can define our subproblems as follows - let Minimize(i, S) be the minimum cost of the subsequence from a_i to a_n, given that we've already used the prime factors in S (so none of the elements from b_i to b_n can have any element in S as a prime factor). i can range from 1 to 100 and |S| <= 17 so there are at most 100 * (2**17) subproblems that we'd have to compute (and if we solve this top down, there are less since we only need to compute Minimize(1, {}) and some (i, S) pairs won't occur). In the subproblem Minimize(i, S), we can run through the numbers from 1 to 59; for every such number k, check if k has any common prime factor with the set S, and if not, consider the subproblem Minimize(i+1, S'(k)) where S'(k) is the union of S and the prime factors of k. Then, we know that Minimize(i, S) = min(cost |a_i - k| + Minimize(i+1, S'(k))). We can implement this efficiently by representing S using the first 17 bits of an int, and precomputing the factors of all k from 1 to 59 (also representing them similarly), so that checking for common prime factors and computing S'(k) can be done as O(1) bit operations; this also makes sure that we don't hit the space limit. In total, we need on the order of about 60 operations per subproblem, totalling to 60*100*(2**17) operations for the whole computation, which is about 8*(10**8) operations, and works in C++.

-- Ram

G. Spinning Up Palindromes