Matcom Online Grader Round #34

Contest Editorial

by

Daniel Otero Baguer

October 23th, 2019

Problem A: A special permutation

This problem can be solved using dynamic programming and the Divide and Conquer principle. Let's first forget about the second condition and let $f(n)$ be the amount of permutations of size n that do not contain an increasing block of size larger than 2. Then $f(0) = 1$ and $f(1) = 1$. The main observation is to look at the position of the element n. If we put n at position x (indexed from 0), then we could try to count the number of ways of filling the positions before x (a) and the positions after x (b). The number of ways of filling the (b) positions is $f(n-x-1)$. This is because all elements will be smaller than n and therefore n cannot augment the size of any increasing block. However with the (a) positions we have to be more careful, since if the sequence we put there ends with an increasing block of size 2 it will grow after adding n. We need then to count the number of sequences that do not contain an increasing block of size larger than 2 and do not end with an increasing block of size 2. Let this number be $g(n)$. Assuming that we can compute $g(n)$ we are ready to compute

$$f(n) = \sum_{x=0}^{x=n-1} g(x) f(n-x-1) \binom{n-1}{x}. \tag{1}$$

Here we the element n at each possible position and also multiply by $\binom{n}{x}$ which is the number of ways of choosing the elements that will be at (a) positions (the rest will at (b) positions). The number $g(n)$ can be computed in a similar way:

$$g(n) = \sum_{x=0}^{x=n-1} g(x) g(n-x-1) \binom{n-1}{x}. \tag{2}$$

To solve the original problem let $\hat{f}$ be the number of permutations with the extra restriction that 1 and n should be inverted. Then

$$\hat{f}(n) = \sum_{x=0}^{x=n-2} g(x) f(n-x-1) \binom{n-2}{x}. \tag{3}$$

The changes are that n cannot be at the last position (the sum goes until $x = n-2$) and that when we count the number of ways of distributing the elements, instead of $\binom{n-1}{x}$, we take $\binom{n-2}{x}$, because we put 1 always in the (b) positions which means that there is one element less to choose from.

Complexity: $O(n^2)$

Complete code: https://matcomgrader.com/submission/140233

Problem B: Bottles recycling

This problem was the easiest one. The idea is that one can buy one bottle, then return it, then buy another one, return it, ... and so on, i.e. repeatedly buy and return a single bottle. Each time we return a bottle we get the y pesos back, that means for each bottle after the first one we only need to pay x pesos (because we can use again the y we got back from the previous one). Each bottle costs then x pesos except the first one that costs $x+y$. Therefore, this way the number of bottles we can buy is $k = \lfloor \frac{\max(L-y,0)}{x} \rfloor$. It is easy to check that indeed there is no way of buying $k+1$ bottles.

Trivial solution

Another possible solution is just to simulate the process in a greedy way. We buy as many bottles as we can in the first step and return them. That is buy $k_1 = \lfloor \frac{L}{x+y} \rfloor$, and subtract $k_1 x$ from the total money (we only paid x per bottle after getting back the y pesos from each bottle), then repeat this process again and so on until there is not enough money for buying any bottle.

We have that $y \geq x$ and $x \leq 100$, this yields

$$
\begin{aligned}
kx = x\lfloor \frac{L}{x+y} \rfloor &= x\left(\frac{L}{x+y} - \{ \frac{L}{x+y} \} \right) && (4)\\
&\geq x\left(\frac{L}{x+y} - 1 \right) && (5)\\
&\geq x\left(\frac{L}{2x} - 1 \right) && (6)\\
&= \frac{L}{2} - x && (7)\\
&\geq \frac{L}{2} - 100 && (8)
\end{aligned}
$$

Therefore at the beginning when $L \approx 10^{18}$ it will decrease very fast (in the first step will decrease $\frac{10^{18}}{2} - 100$, then $\frac{10^{18}}{4} - 50$ and so on). When it is small, i.e. $L \approx 200$, since $x \geq 1$ it will decrease by at least 1 each step but since L is already small this will finish fast.

Complexity (solution 1): $O(1)$
Complexity (solution 2): $O(\log L)$

Complete code (solution 1): https://matcomgrader.com/submission/140185
Complete code (solution 2): https://matcomgrader.com/submission/140236

Problem C: Cocktails

In this problem we are given a list of N liquids $\{(c_i, d_i)\}$, where c_i and d_i are the color and the density resp. of the i-th liquid. We are asked to check if it is possible to make with these liquids a cocktail with several color-layers $\{o_i\}$. The condition is that that if we use the i-th liquid for the x-th color layer and the j-th liquid for the y-th color layer with $x < y$ then it most hold $d_i < d_j$, $c_i = o_x$ and $c_j = o_y$.

The solution follows a greedy approach. We assign liquids to color layers in order starting from the top layer. For each color-layer we assign a liquid of the corresponding color that has the lowest possible density (if it is not the first layer then the density must be greater than the density of the liquid assigned to the previous layer). To that we just need to sort the liquids by density and start assigning them in that order.

```
cur = 0
last_density = -1

for i in range(m):
  while cur < n and (liquids[cur].col != o[i] \
        or (color == liquids[cur].col and liquids[cur].density == last_density)):
    cur += 1
  if cur == n:
    possible = False
    break
  else:
    last_density = liquids[cur][1]
```

Complexity: $O(N \log(N))$

Complete code: https://matcomgrader.com/submission/140110

Problem D: Dictionary search

In this problem we are given N pairs of words. The task is for each query, that consist of two prefixes p and q, find out how many pairs (a_i, b_i) from the given list, satisfy that p is a prefix of a_i (i) and q is a prefix of b_i (ii).

Lets first focus on (i). Then, we only need to find the pairs such that p is a prefix of a_i. To do that we can sort the given pairs by the first word. Then we can do binary search to find the first word (from the beginning of the list) a_l such that $a_l \geq p$ and the first word (from the beginning of the list) a_r such that $a_r > p + \text{'} \sim \text{'}$. The number of pairs such that p is a prefix of a_i would be $r - l$. The problem is that when we sort the pairs by the a_i, the b_i wont be sorted. Therefore we cannot use the same trick but we have reduced the problem to:

- Given an interval $[l,\ r]$ in the sorted list (interval where the pairs satisfy (i)), find the number of pairs such that q is a prefix of b_i, i.e. (ii) holds.

To solve that problem we can create a Segment Tree for the sorted sequence of pairs (a_i, b_i). In each node we will have the list of pairs of words that correspond to its interval sorted by b_i. Then, the query for one node will be: how many pairs satisfy that q is a prefix of the second word. Since the pairs in the list of each node are sorted by the second word we can use the binary search trick.

The memory usage is fine because there will be at most $C = 5000000$ characters (bytes) in the given words (including the queries). Each word will be repeated on several nodes of the Segment Tree but they will be at most $\log(N) \approx 17$. The total memory will be $17 \times C$ bytes $= 85000000$ bytes ≈ 85 MB.

Lets now compute the cost of a query (p, q). First we do the binary search trick with p and the a_is. The cost of that is $O(|p| \log N)$. Then we do the Segment Tree query. We know that we will visit $O(\log N)$ nodes. At each node the cost of the binary search trick will be $O(|q| \log N)$. That means the total cost of one query will be $O(|p| \log N) + |q| \log^2 N = O((|p| + |q|) \log^2 N)$. We have to add up the costs of all the queries:

$$\sum_{i=1}^{Q} O\left((|p_i| + |q_i|) \log^2 N\right) < C \log^2 N \tag{9}$$

Build complexity: $O(C \log N)$

Queries complexity: $O(C \log^2 N)$

Complete code: https://matcomgrader.com/submission/140155

Problem E: Exciting tournament

We model the problem as a graph where each node is a player, and the relations of type player a beats player b are the edges. Because of the conditions given in the problem, it turns out that the graph is a Directed Acyclic Graph (DAG).

For each node v we compute the length $h(v)$ of the maximum path on the graph that ends at v. That is a sequence of nodes $v_1, v_2, \cdots, v_k, v$ where v_1 beats v_2, v_2 beats v_3 and so on. The solution follows from the following observation:

Lema 1. *For every pair of different players a and b, such that $h(a) = h(b)$, it holds that there is no edge (in any direction) connecting a and b.*

Proof. Assume there exist two different players such that $h(a) = h(b)$ and that a beats b without loss of generality. Then there is a path that ends at b with length $h(a) + 1$, which is the longest one that ends at a plus the edge connecting a with b. This is a contradiction because $h(a) + 1 > h(a) = h(b)$. This means that the length $h(b)$ of the longest path that ends at b cannot be equal to $h(a)$. □

Let L be the longest path on the graph, i.e. $L = \max_{1 \leq v \leq n} h(v)$. Then for the tournament they need at least L rooms. Moreover, with L rooms is enough. Lets number the rooms $[0, 1, 2 \cdots, L-1]$. If we assign each player v the room $h(v)$, we know already that there wont be any edges between players in the same room (because of Lemma 1).

The problem is now reduced to find out for each vertex/player v if removing it reduces the length of the longest path on the graph. The first necessary condition (i) to check that for each vertex v, is to check if v belongs to any path with length L. To do that, we also need for each vertex v the length $d(v)$ of the longest path on the graph that starts at v.

Lema 2. *A node v belongs to any path of length L if and only if $h(v) + d(v) = L$.*

Proof. Trivial □

The other necessary condition (ii) is that when we remove node v there is no other node u which still belongs to some path of length L, i.e. all paths of length L pass through v. This equivalent to say that v satisfies (ii) if there is no other u with $h(v) = h(u)$ that satisfies (i).

The whole problem is solved then looping over the vertices and checking if conditions (i) and (ii) hold. One can compute $h(v)$ and $d(v)$ with dynamic programming over the DAG. The edges are given in such a way that there is no need to do a Topological order, if there is an edge going from a to b then $a < b$.

The following code fragment shows how to implement it.

```
h = [0] * n
d = [0] * n

L = 0

for v in range(n):
  for j in range(len(graph[v])):
    u = graph[v][j]
    h[u] = max(h[u], h[v] + 1)
    L = max(L, h[u])

for v in reversed(range(n)):
  for j in range(len(graph[v])):
    u = graph[v][j]
    d[v] = max(d[v], d[u] + 1)

ans = []
cnt = [0] * n
vid = [0] * n
for v in range(n):
  if h[v] + d[v] == L:
    cnt[h[v]] += 1
    vid[h[v]] = v

for i in range(n):
  if cnt[i] == 1:
    ans.append(vid[i])
```

Complexity: $O(N + M)$

Complete code: https://matcomgrader.com/submission/140237