Santa's Programming Challenge 2017 – editorial

A Christmas Lights

It is not hard to see that the sequence is correct if and only if each number in the sequence is equal to the previous one plus 1, with the exception that after n we should see 1, not $n + 1$.

It is enough to check this in a loop, and as soon as some i-th element is not the proper successor of the $(i - 1)$-th, output `YES` and i (since then i is the first position at which we are certain that the lights do not work properly).

B Letters To Santa

We process characters one by one and maintain a single counter. The counter starts at zero, and we increment it whenever we see `(` and decrement whenever we see `)`, except if it comes right after `:` or `;`. The counter should never become negative, and should be zero at the end – otherwise the letter's author is naughty.

Why does that work – that is, why is this a necessary and sufficient condition for the parentheses to be matchable as described in the statement? Imagine maintaining, instead of the counter, a stack that stores occurences of `(`. When a new opening parenthesis arrives, we push it onto the stack. When a new closing parenthesis arrives, we pop an opening parenthesis from the stack, and match this pair together. If there are no parentheses on the stack at the end, then everything has been matched properly. Moreover, if at any point we have seen more closing parentheses than opening ones, then clearly there is no correct way to match them. The same holds if the numbers of opening and closing parentheses are not equal (in which case our counter will be nonzero at the end).

C Sleigh

The goal is to maximize the number of happy reindeer. We solve this problem using a greedy approach, which consists in forming pairs that make both reindeer happy, as long as possible.

First consider any two reindeer of the same color that want to be paired with the same color. Pairing them together will make them happy, and in the rest of this paragraph we explain that we should always do it. This is because if we do not make any such pairs (for example, no $a-a$ pairs), then: consider any two a reindeer; they will be matched to some other two reindeer, in a way that makes at most two out of the four happy (in each of the two pairs, the only way to make the a happy would be to match it with a b, but then the b is unhappy). Thus it makes sense to make at least one such pair... and then we repeat the argument. So we match all such possible pairs ($a-a$ and $c-c$).

By the same argument, we should match all $b-d$ pairs possible. Thus there are either no b left or no d left; suppose the latter. The only pairs that give happiness to at least one reindeer now are $a-b$ and $b-c$. Make as many of them as possible (that will be $\min(a+c,b)$). Pair up the rest in any way.

Our algorithm can be implemented to run in constant time. There are also many other possible solutions that are slower (we can afford up to $O(n^3)$ or even $O(n^4)$ time).

D Bad Presents

We could try to simulate the process, but that sounds way too slow – there could be up to 10^9 passes over all presents, each of which could take 10^5 steps. We need a better perspective.

One that works is looking at every present i separately, and asking: how many seconds will be spent on present i before present 1 is ready? It turns out that we can compute this in just $O(1)$ time, and then add up all the answers.

It is easy to see that $\lceil\frac{a_1}{b}\rceil$ seconds will be spent on present 1 before it is ready. For every other present, it will be processed for at most $\lceil\frac{a_1}{b}\rceil-1$ seconds (because in the $\lceil\frac{a_1}{b}\rceil$-th pass, present 1 becomes ready and we terminate without looking at the others), and possibly fewer if it is ready earlier. More precisely, $\min\left(\lceil\frac{a_1}{b}\rceil-1,\lceil\frac{a_i}{b}\rceil\right)$ seconds will be spent on present $i\neq 1$. In total, the answer is

$$\lceil\frac{a_1}{b}\rceil+\sum_{i=2}^{n}\min\left(\lceil\frac{a_1}{b}\rceil-1,\lceil\frac{a_i}{b}\rceil\right).$$

E Downhill

In this problem we are given a directed graph where every edge (b,a) has $b>a$. In other words, it is a directed *acyclic* graph (DAG) (with $(n,n-1,n-2,...,2,1)$ being a topological ordering for it). We are asked to find the number of paths from n to 1 in this graph.

We start by constructing the adjacency list representation of the graph. Actually, it will be more convenient to do it for the transposed graph (with the edges reversed). Namely, we create n lists $\texttt{in}[1],\texttt{in}[2],...,\texttt{in}[n]$ (or, more likely, `vector<int>`s, `ArrayLists`, etc.), where $\texttt{in}[i]$ will store the numbers of vertices with an edge *to* i. We fill these lists while reading the edges from the input: when seeing an edge (b, a), add b to $\texttt{in}[a]$.

Now the task can be solved using dynamic programming. For $i = n, ..., 1$ we compute $\texttt{dp}[i]$: the number of paths starting at n and ending at i. Of course, $\texttt{dp}[1]$ is the answer we want. For vertex n, we have $\texttt{dp}[n] = 1$. For $i = n-1, n-2, ..., 1$, we get the following recurrence by fixing every possible second-to-last vertex b on the path (before i):

$$\texttt{dp}[i] = \sum_{b \in \texttt{in}_i} \texttt{dp}[b]\,.$$

Remember to perform all arithmetic operations modulo 500 000 003 and be mindful of possible overflows.

F Claussian Distribution

Let us first state a key observation. Let s_i be the longest word among $s_1, \ldots, s_n$. We need to match it to another word that is its prefix. We claim that the best option is to match s_i with the longest such word s_j. Intuitively, this is because it is worse to have longer words left to be still paired up than to have their prefixes; the prefixes would allow us to form the same pairs (or more) as the longer words. For a proof, see the following paragraph.

Formally, we are claiming that if there exists any solution, then there exists a solution where s_i and s_j are matched together. Why is that the case? Well, assume it's not, and consider any solution. There, we must have matched s_i to some other prefix s_k, with $|s_k| < |s_j| < |s_i|$. Then s_j needs to be matched to some other prefix s_ℓ, with $|s_\ell| < |s_j| < |s_i|$. Since these are all prefixes of s_i, the two shortest words s_k and s_ℓ can also form a pair (with the shorter being the prefix of the longer). So we can swap out the pairs and, rather than (s_i, s_k) and (s_j, s_ℓ), instead have (s_i, s_j) and (s_k, s_ℓ).

Therefore a correct greedy algorithm would be as follows. In each step, find the longest string and match it with its longest possible prefix. Now we need to describe an efficient way to, at each step, find the longest string and find its longest prefix. We first insert all the strings in a TRIE data structure (if you have never heard of it – it is well-described on Wikipedia). This allows us to find the the longest prefix of each string in time linear with respect to the length of the string. Also, by sorting the strings according to their length, we can find the longest string at each step. Notice that during the execution of our algorithm we need to make sure that we do not match any string twice; this can be done by marking-off the already matched strings. The running time of our algorithm is linear: $O(|s_1| + |s_2| + ... + |s_n|)$.

G Getting Out

Let us first make two simplifications. First, assume that Santa always needs to press the blue button at the very end. Since actually every optimum sequence ends with a red button, we will just need to subtract 1 at the end to get our true result. Second, assume that no more than n consecutive red button presses are allowed. Later we will see how to lift this assumption.

For the simplified problem there is a simple dynamic programming solution. Namely, let us define, for $0 \le y \le x$,

$$\texttt{dp}[y] = \text{the minimum number of presses required to obtain the number } y.$$

Clearly, we have $\texttt{dp}[0] = 1$ (blue button at the end), and we set $\texttt{dp}[y] = \infty$ for $y < 0$. Moreover, for $y \ge 1$ we have, by considering all possible lengths $k = 1, 2, ..., n$ of the last sequence of consecutive red button presses, that

$$\texttt{dp}[y] = \min\{\texttt{dp}[y - s_k] + k + 1 : k = 1, 2, \ldots, n\},$$

where $s_k := \sum_{i=1}^{k} a_i$ (this can be precomputed in linear time).

Given our simplification, $\texttt{dp}[x]$ is not quite the solution to the original problem. However, note that every sequence of button presses that produces the number x can be transformed into one where only the *last* block of red button presses (if any) is longer than n. This means that, given the numbers $\texttt{dp}[y]$, one just has to go over this array again (over all y's) and check the cost of placing the "last" block of $x - y$ red buttons, so as to get x. This takes time $O(1)$ per one y. In total, the time complexity of the solution is $O(nx)$, and it can be implemented in space $O(x)$.

H Christmas Tree

Observe first that the network of bus routes forms a tree, with stations being its vertices and bus routes being its edges. We can traverse this tree in the direction from the root (EPFL) towards the leaves using buses, and we can move using backward links (trains) but it costs us 1 CHF. (Note that it makes no sense to use a train in the direction from the root.)

The solution consists of two phases: (1) Prepare a data structure to process the queries. (2) Answer the queries.

In the first phase we will precompute the answers to all queries with budgets $k = 1, 2, 4, \ldots, 2^L$ for all the stations (vertices in the tree given in input), where $L = \lceil \log m \rceil$. We will explain later how to achieve this in $O(n \log m)$ time. Now, let us show how to implement the second phase, i.e., how to answer queries given such a structure.

The key observation one has to make is that being at vertex v and given a budget of k CHF, we can pick any k_1 and k_2 such that $k_1 + k_2 = k$ and compute the solution as follows:

(1) Find the highest (in the tree, i.e., the closest to EPFL) vertex w which can be reached from v using k_1 CHF.

(2) Find the highest (in the tree, i.e., the closest to EPFL) vertex u which can be reached from w using k_2 CHF.

Then (as one can easily prove) the answer to our query is u. Thus, in order to answer any query, it is enough to write k as $\sum_{i=0}^{L} b_i 2^i$ (its binary representation) and use our data structure, which provides answers for all queries with the budget being a power of 2.

Finally, it remains to show how to efficiently construct such a data structure. The same observation as made above will be useful for this task as well. We construct the structure by first computing all the answers for $k = 1$, then for $k = 2$, $k = 4$, etc., up to $k = 2^L$.

Let us first show how to do that for $k = 1$. This is a simple exercise in dynamic programming on a tree. We go from bottom to top, i.e., from leaves to the root of a tree. At a vertex v we find the highest (closest to the root) vertex that can be reached through a direct backward link (i.e., a train) from v; further, we also inspect the solutions for children of the vertex v and set the solution for v as the highest vertex among all these.

Now, we need to show how to compute solutions for a budget $2k$, given all solutions for budget k. Let highest(v, k) be the highest vertex one can reach starting from v and given a budget k. Then, as discussed above (with $k_1 = k_2 = k$), we have

$$\text{highest}(v, 2k) = \text{highest}(\text{highest}(v, k), k).$$

This allows us to compute the answer for budget $2k$ in $O(n)$ time. Thus, the total running time of the solution is $O((n + q) \log m)$ and the space occupied is $O(n \log m)$.