Central European Olympiad in Informatics 2017

July 2017, Ljubljana, Slovenia

Problem Solutions - Practice Session

Jenga

Vid Kocijan

Exhaustive search. A Minimax algorithm can be used to determine which states of the game are winning and which losing. Each state of the game is determined by the states of the storeys. Each storey can be in 8 states (all possible combinations of missing blocks), 3 of which automatically crash the tower, so we can ignore them. We can observe that the order of storeys in the tower is not important. Hence, a state of the game is described by the number of storeys in each state and the number of blocks in the topmost storey. The state of the game is automatically a losing one if no block can be removed without crashing the game. If we can make one move in such way that the new state of the game will be a losing one, the current state of the game is a winning one. Otherwise it is a losing one.

With the described recursive formula we can determine if the starting position is a winning or a losing one and decide if we want to start first or second. By playing correctly we will remain in winning positions for the rest of the game. On our turn we can make any move that will change the state of the game into a losing one (so the opponent will be in a losing position) and play until the opponent crashes the tower.

Dynamic programming. We can speed up the algorithm by storing the calculated values (winning or losing) for each state. We can safely assume that there will less than $3n$ storeys at the end of the game. If we count all the possible combinations of 5 storey states we will get a $O(n^5)$ possible states of the game, which is enough for the $N \leq 40$ subtask.

The number of states can be reduced. We can rearrange them into 3 categories: "Full storey", "one side block missing" and "the rest". From the storeys in the first category any block can be removed. From the storeys in the second category the side block can be removed. From the storeys in the third category no block can be removed. The state of the game is hence described by the number of storeys in first and second category and the number of blocks in the top storey. This gives us a $O(n^2)$ overall complexity.

Multiply

Tomaž Hočevar

This is an easy task. Just multiply the two numbers.

Don't forget about 64-bit integers to pass the second subtask.

The numbers in the third subtask are too big for most integer types available in programming languages. This requires an implementation of long (grade-school) multiplication in $O(nm)$.

In the last subtask the numbers are just a bit too large for what long multiplication can handle in the given time limit. A speed-up by a constant factor will be good enough. Instead of doing long multiplication digit by digit, we can group the digits into block of 8, essentially representing the number in base 10^8. This should speed up the solution 64-times.

Of course there exist subquadratic algorithms for multiplying two numbers such as Karatsuba or even Fast Fourier Transform. Neither is necessary for solving this problem. Although the easiest way to solve it is to use Python or Java's BigInteger class that already implement such subquadratic methods.

Museum

Tomaž Hočevar

Let's root the tree structure of rooms at the starting room. If the tourist had to return to the starting room, he would make a depth-first traversal of some subtree of size k containing the root. In the process he would visit every edge (passage) of the subtree exactly twice. Because he can end his visit somewhere else, the edges on path from the finishing node to the root contribute their length only once and not twice as the others.

Brute force. The constrains in the first subtask are small enough that we can try out all possible subtrees and paths within them that represent the last part of the trip. There are less than 2^n subtrees and n "weight-one" paths.

Dynamic programming. Second subtask guarantees that every node has a degree at most 3. This implies that every node has at most two subtrees (we'll call them the left $L(x)$ and right $R(x)$ subtree) except for the root that has at most three. Let $f(x, k, s)$ represent the solution of a subproblem of finding the shortest tour within the subtree rooted at x that visits k nodes. Flag s indicates whether the tour has to finish back at the start x (T) or not (F). Let $C(x, y)$ represent the length of the edge between x and y. To solve a subproblem we have to decide how many nodes we want to visit in the left subtree (l) and how many in the right ($k-1-l$). If we don't have to return to the start ($s = F$), we can traverse one of the edges at a single instead of double cost. Equations below illustrate this. We can handle the root separately by checking all distributions of k over its subtrees. From here on we will ignore the parameter s for brevity.

$$f(x,k,T) = \min_{0\le l\le k-1} 2C(x,L(x)) + f(L(x),l,T) + 2C(x,R(x)) + f(R(x),k-1-l,T)$$

$$f(x,k,F) = \min_{0\le l\le k-1} \min \begin{cases} C(x,L(x)) + f(L(x),l,F) + 2C(x,R(x)) + f(R(x),k-1-l,T) \\ 2C(x,L(x)) + f(L(x),l,T) + C(x,R(x)) + f(R(x),k-1-l,F) \end{cases}$$

High degree. How can we deal with the situation where the number of subtrees is large? We can't afford to test all distributions of $k-1$ over the subtrees. We will use another dynamic programming to find a solution for the subtree rooted at x from solutions of subtrees rooted at children of x ($y_1, \ldots, y_n$). If we fix the number of nodes to visit in y_n (e.g. l), we are left with a subproblem of distributing $k-l$ over $y_1, \ldots, y_{n-1}$. The solution we are interested in is $f(x, k) = g(n, k-1)$.

$$g(n,k) = \min_{0\le l\le k} (g(n-1,k-l) + f(y_n,l))$$

Note that we use only $g(n-1, *)$ for computing $g(n, *)$. Therefore we need only $O(n)$ additional memory for this auxiliary dynamic programming. Time complexity of this solution is $O(nk^2)$ and is good enough to solve the third subproblem.

Further optimization. Let $S(x)$ represent the size of the subtree rooted at x. It doesn't make sense to solve a subproblem $f(x, k)$ for values of k larger than $S(x)$. Let's make a similar optimization for computing values $g(n, k)$ which represent the bottleneck of our solution. To compute values $g(n, *)$ we have to consider only pairs of subproblems $g(n-1, a)$ and $f(y_n, b)$ such that $a \le S(y_1) + \cdots + S(y_{n-1})$ and $b \le S(y_n)$. We can visualize this as pairing every node in subtree of y_n with all nodes in subtrees of $y_1, \ldots, y_{n-1}$. If we do this for every x, we will pair every pair of nodes in the tree exactly once (when solving the subproblem for their lowest common ancestor), which shows that this optimization leads to a $O(n^2)$ solution.