IOI Training Camp 1, 2009 Day 1 Solutions

SACO Scientific Committee

1 Broken Compiler

This problem is a variant of the topological sort problem. The only difference is that we are looking for the topological sort which gives you the least lexicographical ordering of vertices visited. Every direct acyclic graph is guaranteed to have at least one topological sort.

The standard topological sort algorithms by iteratively visiting and deleting vertices with an in-degree of zero (or open vertices). The out-edges of these vertices are then deleted, hopefully lowering the in-degree of other vertices. If the algorithm stops before all vertices have been deleted, i.e. there are still unvisited vertices, but none of them have in-degree of zero, then there is a cycle in the graph and there is no valid topological sort for the graph. This algorithm has a complexity of $O(V + E)$.

However, since we want the least lexicographical ordering of the vertices we need to modify this algorithm a bit. Without too much work, it is easy to see that greedily picking the open vertex with the lowest number will give you the least lexicographical ordering. The brute-force approach is to look for the lowest open vertex each time. This will give you a complexity of $O(V^2 + E)$ which should score you only about 50%.

In order, to speed the algorithm up, we keep a priority queue with a list of the open vertices in it. This allows you to speed the algorithm up to $O(VlogV + E)$ which is enough for 100%. Be careful, however, if you use the STL priority_queue as it implements a max-heap instead of a min-heap (the opposite of what you want.) The easiest way around is just to push the negation of the number onto the priority queue and negate it again when you get it off the priority queue.

2 Wrapping the Rock

2.1 40% Solution

The $O(NK)$ solution which was the 100% solution for the 3rd round contain problem, now gets only 40%! For a starting point i, let M_i be the number of sheets needed in a greedy placement placing the first sheet at position S. The greedy placement works by placing a sheet at the current position, moving the current position to the first uncovered section and repeating until the entire rock is wrapped. The optimal solution is the minimum of the set

$M_p, M_{p+1}, \ldots, M_{p+K}$ where p is the position of the first uncovered section. It can be pruned by only checking M_i if there is an uncovered section at i. This is optimal because any arrangement of sheets starting at a position outside the checked range is equivalent to one starting within the range.

2.2 80% Solution

For each magnet, find the first uncovered section that will not be covered if the chosen uncovered section is at the start of a sheet. This can be done in linear time by walking two pointers around the ring, keeping them K apart. Find M_p using this jump table. It can be shown that the optimal answer is either M_p or $M_p - 1$. If we can cover all uncovered sections by greedily placing $M_p - 1$ tiles starting at the location of an uncovered section, then the answer is $M_p - 1$. Otherwise, the answer is just M_p. This can be sped up by precomputing a jump table using an equivalent of Russian Peasant multiplication, which runs computes each entry in $O(\log N)$ time. Since we compute such entries, the solution is $O(N \log N)$.

2.2.1 Russian Peasant Multiplication

1. To calculate $A \times B$, first write A and B each at the head of a column.
2. Calculate $A_i = \lfloor \frac{A_{i-1}}{2} \rfloor$ where $A_0 = A$ until $A_i = 1$ and write down the series under A.
3. Do the same for B, but multiply instead of dividing by 2.
4. Add up all the numbers B_i where A_i is odd. This gives the result of $A \times B$.

The algorithm is based on the binary form of A:

$$\begin{aligned} 85 &= 1010101_2 \\ &= 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 64 + 16 + 4 + 1 \\ 85 \times 18 &= (64 + 16 + 4 + 1) \times 18 \\ &= 1152 + 288 + 72 + 18 \\ &= 1530 \end{aligned}$$

Reference: `http://www.cut-the-knot.org/Curriculum/Algebra/PeasantMultiplication.shtml`

2.3 100% Solution

Build the first jump table from the 80% solution. While doing this, also observe how many uncovered sections are covered by each such placement of a sheet, and find the placement that covers the fewest; call this number F. Now run the greedy placement starting from each possible point in this emptiest range

(plus the first uncovered one). The jump table computed earlier means that the number of steps in the greedy is just the number of sheets required; and since every sheet covers at least F points, this is at most $O(\frac{N}{F})$ such steps. Since there are $F + 1$ possible starting points, $O(N)$ time is sufficient overall.

3 Washing Line

3.1 Brute force

The simplest solution to this problem is to store the weights in an array, and check each possible subsequence of the array for every query. We need linear time to insert an element into an array (since we need to shift everything after the insertion point), and we need quadratic time for each query (since there are $O(N^2)$ pairs of starting and ending points for the subsequence).

So overall we need $O(N^3)$ time (assuming the number of queries is proportional to the number of insertions) to process all the commands. This solution scores around 20%.

3.2 Dynamic programming

One can improve the above solution by using the very simple DP for the "heaviest subsequence" problem to answer queries (this problem is essentially an on-line version of heaviest subsequence).

Consider the heaviest subsequence ending after a certain item: either this subsequence is empty, or it includes adds this item to the heaviest subsequence ending at the previous point. I.e., $h_{i+1} = max\{h_i + w_{i+1}, 0\}$ where h_i is the weight of the heaviest subsequence ending after item i and w_i is the weight of item i. Then the weight of the heaviest subsequence is the maximum of all the h_i's.

We can now answer queries in linear time, so the overall efficiency is now quadratic in the number of queries. This solution scores around 40%.

3.3 Trees

We can improve both the query time and the insertion time to $O(\log N)$ using—you (should have) guessed it[1]—balanced binary trees.

In each node of the binary tree, we store the size of the corresponding subtree (i.e., the number of descendants plus one). This enables us to insert items at arbitrary positions. Many of the test files will lead to highly unbalanced trees, so we need to keep the tree balanced to ensure $O(\log N)$ insertions.

If we build our own tree[2] we can answer queries in constant time. The work needed to calculate the heaviest subsequence is done during updates, but the essential point is that only the parts of the tree which change must be updated.

[1] If it's a data structure, and it supports logarithmic-time operations, it's probably a tree.

[2] In theory, a sufficiently general balanced-tree library could be used. The STL does not contain such a tree, but Haskell's fingertree package does.

We want to know the weight of the heaviest subsequence of the root node (now a node represents not only a value, but also an interval—in fact, one can simplify things by having values only in the leaves, so non-leaves *only* represent intervals). The heaviest subsequence will either be the heaviest subsequence on the left, or some sequence overlapping both. To work out the weight of the latter subsequence, we need the weights of the heaviest subsequences ending at the right of the left child and starting at the left of the right child. To work out *these*, we need the sum of the weights in the children's intervals. Thus, by storing in each node the total weight and the weights of the heaviest subsequence, the heaviest subsequence starting at the left and the heaviest subsequence ending at the right, we can calculate each node's values from its children's values in constant time. When we insert a value into the tree, we simply update all the nodes which become its ancestors.

This solution scores 70% if the trees are left unbalanced, and full marks if balancing is implemented.