2003/2004 ACM International Collegiate Programming Contest
University of Ulm Local Contest

Judges' Comments on the Problems and their Solutions

Problem	Name	Rating	Method
A	Assistance Required	Easy	Precalculation by Simulation
B	The Bottom of a Graph	Medium	Strongly Connected Components Topological Sorting
C	Fixed Partition Contest Management	Hard	Brute Force
D	Drink, on Ice	Medium	Geometry
E	Edge	Easy	Simulation
F	Fold	Medium	Dynamic Programming
G	Genetic Code	Medium	Precalculation by Backtracking
H	Largest Rectangle in a Histogram	Hard	Linear Search using a Stack or Order-Statistic Trees or Rewrite System or ...

Problem A: Assistance Required

While the large contest party really took place at SWERC 1997, the contestants were not forced to wash up by this method.

The lucky numbers can be precalculated, e.g., by using a linked list that contains all numbers not yet removed from the queue. The 3000th lucky number is 33809.

Judges' test data consists of testing all possible numbers twice.

Rating: Easy

Reference

Lutz, G.
Lucky Numbers
SWERC 1997, Problem X

Problem B: The Bottom of a Graph

First, calculate the connected components of the graph and process them independently one after the other. For each connected component, calculate its strongly connected components (e.g., by using the depth first search based algorithm). For the moment, interprete each strongly connected component as a single node. Sort every connected component, which now is a directed acyclic graph, topologically. Find the strongly connected components that have no outgoing edges. Those nodes that belong to such a strongly connected component (give up the single node interpretation now) are sinks. Output them in sorted order.

Judges' test data consists of several hand-crafted tests along with randomly generated input. The total number of test cases is 32.

Rating: Medium

Problem C: Fixed Partition Contest Management

This problem can be solved by trying all possible assignments brute-force, and calculating for each assignment the average solution time. In a concrete assignment, every contestant has to solve his problems in the order of the required time, shortest first, to minimize the accumulated time.

Note that this problem was posed with similar wording in a final contest. There, however, *1 <= m <= 10* and *1 <= n <= 50* was specified. With such large bounds, the brute-force approach is no longer appropriate. There exists a polynomial time solution for this problem using a reduction to a minimal cost at maximal flow problem in a certain network graph.

Judges' test data consisted of 18 hand-crafted test cases along with 100 randomly generated test cases. Since there may be more than one optimal schedule, a verification program supported the judging process. A technicality: the rounding facilities of several compilers and runtime libraries may be obscure, if not incorrect. We therefore recommend rounding manually to the required precision, although our verification program was generous in this respect.

Rating: Hard

Reference

Ruhl, M. (?)
Fixed Partition Memory Management
World Finals 2001, Problem G

Problem D: Drink, on Ice

There are essentially two ways to solve this problem. Either you calculate the total energy of the system according to the picture described where the case of temperature 0 needs careful treatment. Or, you simulate the process by freezing water and/or melting ice, treating several cases specially.

Judges' test data consisted of 36 hand-crafted tests.

Rating: Medium

Problem E: Edge

The solution is straight-forward. While traversing the edge, the current position and orientation must be remembered and updated.

Judges' test data consisted of testing all possible inputs of length 5 or shorter along with 30 randomly generated input strings and a test case that models the sheet of a real-world product.

Rating: Easy

Problem F: Fold

This problem is solved by dynamic programming. We compute the minimum number of required folding steps for every subsequence of the input string. The order of computation is by increasing subsequence length.

For the empty sequence, i.e., a single stripe, zero folding steps are required. To compute the solution for stripes *i..j*, where *i<j*, we try every possible intermediate location *i<=k<=j-1* for folding. In every case we are left with two subsequences: the stripes *i..k* and the stripes *k+1..j*.

We determine whether these parts can be folded upon each other. This is the case if the shorter one, after being reversed and complemented (i.e., *A*s and *V*s exchanged), is a prefix of the longer one. Then, we may fold between stripes *k* and *k+1*, and the number of folding steps for the remaining longer part has already been computed. For efficiency, the matching test for the two parts is accelerated by precomputation.

Judges' test data was the same as the test data for problem E: Edge.

Rating: Medium

Problem G: Genetic Code

While lake Vostok exists as described in the problem statement, the three-base genome construction is fiction.

Observe that any subsegment of a Thue-sequence is itself a Thue-sequence. Therefore, it suffices to calculate the longest required sequence once, and answer all queries by taking, e.g., prefixes of that sequence. Such a sequence can even be precalculated and stored in the submitted source code.

Thue proved the existence of an infinitely long Thue-sequence, so the case of no genomes never arises. There is even a linear time method to construct such a sequence by repeatedly, simultaneously replacing characters by certain strings. Such a solution is not required for *n<=5000*, where a backtracking approach is adequate. Note, however, that the testing condition for backtracking must not be too inefficient.

If a sequence *s* is a Thue-sequence, then appending a character *c* gives a Thue-sequence *sc* iff there is no suffix of *sc* that is a square word. A square word is the concatenation of two identical words. Therefore, Thue-sequences are also called square-free sequences. Testing the suffixes requires quadratic time in the length of the sequence and is efficient enough. Asymptotically even more efficient testing in linear time in the sequence length can be performed by dynamically constructing and modifying suffix-trees.

Judges' test data consisted of 198 test cases including *1<=n<=100* and *n=100,200,300,...,5000*.

Rating: Medium

Reference

Hehner, E.C.R
The Logic of Programming
Chapter 9.3, p. 256, Exercise 10, Prentice Hall International, Inc., 1984
ISBN 0-13-539966-1

Problem H: Largest Rectangle in a Histogram

Note that the area of the largest rectangle may exceed the largest 32-bit integer. Due to the large numbers of rectangles, the naive $O(n^2)$ solution is too slow. Even though *O(n*log(n))* or *O(n)* is required, there are several kinds of solutions to this problem. In the following, we will identify a histogram with the sequence of the heights of its rectangles.

1. Divide-and-conquer using order-statistic trees.
   Initially, build a binary, node- and leaf-labeled tree that contains the histogram as its frontier, i.e., as its leaves from left to right. Mark each inner node with the minimum of the labels of its subtree. Note that such a tree is most efficiently stored in an array using the heap data structure, where the children of node *i* are located at positions *i*2* and *i*2+1*, respectively. With such an order-statistic tree, the minimum element in a subhistogram (i.e., a subsegment of the sequence of heights) can be found in *O(log(n))* steps by using the additional information in the inner nodes to avoid searching all leaves.
   To calculate the maximum rectangle unter a subhistogram, we thus find the minimum height in that subhistogram, and divide it at just that place into two smaller histograms. The maximum rectangle is

either completely in the left part, or completely in the right part, or it contains the rectangle with minimum height. The first two cases are solved recursively, while in the third case we know that the width of the maximum rectangle is the width of the whole subhistogram, since we chose the minimum height. Because every element serves exactly once as a place to divide, and we spend *O(log(n))* for every division, the complexity of this algorithm is *O(n*log(n))*.

2. Linear search using order-statistic trees.
   Initially, construct an order-statistic tree as just described. For every element, we find the largest rectangle that includes that element. The height of the rectangle is, of course, the value of the element. Use the order-statistic tree to efficiently find the nearest elements that have smaller heights, both to the left and to the right. The width, and therefore the area of the rectangle can thus be calculated in *O(log(n))* steps, giving a total runtime of *O(n*log(n))*.
3. Sweeping line maintaining intervals.
   Initially, sort the heights so they can be processed in non-increasing order. We sweep a horizontal line downwards through the histogram, maintaining a list of those intervals, where the line intersects the histogram. Actually, the intervals are maintained in an array with one entry for every element of the histogram in the following manner. At the element corresponding to the left end of an interval we store a pointer to the right end, and vice versa.
   As a new element arrives during the sweeping process, this element forms a new interval, and, if there are adjacent intervals, these can be merged in constant time using our representation. The largest rectangle including this element, just as described in the previous algorithm, is available without further expense, since we can read its width from our representation. Actually, it is not quite the largest rectangle, because there may be further elements with equal heights adjacent to the current interval. Performing the sweep in a non-increasing order, however, guarantees that the largest rectangle will have been considered by the time the last element of a group having equal heights is examined. The complexity is dominated by the sorting phase, thus *O(n*log(n))*, although a radix sort is possible if the heights are bounded.
4. Linear search using a stack of incomplete subproblems
   We process the elements in left-to-right order and maintain a stack of information about started but yet unfinished subhistograms. Whenever a new element arrives it is subjected to the following rules. If the stack is empty we open a new subproblem by pushing the element onto the stack. Otherwise we compare it to the element on top of the stack. If the new one is greater we again push it. If the new one is equal we skip it. In all these cases, we continue with the next new element.
   If the new one is less, we finish the topmost subproblem by updating the maximum area w.r.t. the element at the top of the stack. Then, we discard the element at the top, and repeat the procedure keeping the current new element. This way, all subproblems are finished until the stack becomes empty, or its top element is less than or equal to the new element, leading to the actions described above. If all elements have been processed, and the stack is not yet empty, we finish the remaining subproblems by updating the maximum area w.r.t. to the elements at the top.
   For the update w.r.t. an element, we find the largest rectangle that includes that element. Observe that an update of the maximum area is carried out for all elements except for those skipped. If an element is skipped, however, it has the same largest rectangle as the element on top of the stack at that time that will be updated later.
   The height of the largest rectangle is, of course, the value of the element. At the time of the update, we know how far the largest rectangle extends to the right of the element, because then, for the first time, a new element with smaller height arrived. The information, how far the largest rectangle extends to the left of the element, is available if we store it on the stack, too.
   We therefore revise the procedure described above. If a new element is pushed immediately, either because the stack is empty or it is greater than the top element of the stack, the largest rectangle containing it extends to the left no farther than the current element. If it is pushed after several elements have been popped off the stack, because it is less than these elements, the largest rectangle containing it extends to the left as far as that of the most recently popped element.
   Every element is pushed and popped at most once and in every step of the procedure at least one element is pushed or popped. Since the amount of work for the decisions and the update is constant, the complexity of the algorithm is *O(n)* by amortized analysis.
5. Recursive search

For a recursive version of the algorithm just described, see the reference below. Indeed, if the recursion is eliminated the necessary stack is analogous to the explicit stack in the iterative version.

6. Rewrite System
   The problem may be generalized by allowing histograms to be composed of rectangles with varying widths. Then, the stack used in the just described algorithms can be concatenated with the yet unprocessed part of the histogram into one list.
   A three element window is moved over this list, starting at the left end. In every step, different actions are performed according to the relative heights of the three elements under inspection. The actions include updating the maximum, merging two of the three elements by taking the minimum of their heights and the sum of their widths, and sliding the window one element to the left or to the right. Rules are provided that specify the actions for each configuration.
   Actually, the behaviour of the stack-based algorithm is simulated. The correctness of the rewrite system can be shown by proving an invariant about the maximum area of the histogram, observing that every configuration is matched by some rule, and giving a termination proof using a variant expression. Additionally, it can be proved that *O(n)* rewrite steps (each with a constant amount of work) are performed.

Judges' test data consisted of 32 hand-crafted test cases, 33 randomly generated test cases, and one test case with a histogram of width 99998.

Rating: Hard

Reference

Morgan, C.
Programming from Specifications
Chapter 21, pages 209-216, Prentice Hall International (UK) Limited, second edition, 1994
ISBN 0-13-123274-6