Nordic Collegiate Programming Contest 2005

Solution sketches

Problem A: Crashing Robots

(Author: Børge Nordli)

This is a straightforward problem, using no advanced algorithms. You need a matrix representing the warehouse, where the robots positions are marked. You also need an array giving the current rotation (and possibly position) of all robots. If you do not store the robot positions, it is still possible to scan through the matrix to find the appropriate robot.

For each instruction, if it is a rotation, rotate the robot accordingly. If it is a move, you check whether there is a robot or a wall in its path, and check which crash will occur first.

Remember to read the remainder of the test case when a crash occurs!

(It is possible to solve it without the warehouse matrix, by only checking the robots array, but neither space nor time is an issue in this problem.)

Problem B: Funny Games

(Author: Øyvind Grotmol)

As with all turn-based, two-player, finite, deterministic games, the goal is to categorize the positions as winning or losing (a position is losing if no winning position can be reached from it). Dynamic programming or memoization is often used to achieve sufficient speed, as such a technique normally reduces the amount of work per game position to near constant.

In this case a game position would be the size of the planet. The trouble is that the number of reachable positions is really large, being exponential in the parameter X. The crucial observation is that one can calculate with winning and losing size intervals, instead of singular values. The key is identifying the points that separate winning from losing intervals. This can be done bottom-up, starting with the value 1. Given that v is a separation point, the values $\frac{v}{F_1}$, $\frac{v}{F_2}$, ..., $\frac{v}{F_k}$ must be checked to see if they are separation points. Once all candidate separation points below X have been investigated, the winning and losing intervals are determined and one only has to see which interval X belongs to.

Problem C: Nullary Computer

(Author: Erling Alf Ellingsen)

This is one of the harder problems in the set, but might seem a bit harder than it is. The Nullary Computer could be programmed to calculate anything, but you are to make it do something fairly simple. Since there only are 26 registers, and you must sort 24 numbers, you have two scratch registers. First, you need to build your basic tool: As a part of sorting, you often need to compare two numbers, and if the "left" one is larger than the "right" one, you swap them. Since your computer has limited memory, this swap-min-max routine should be as small as possible. Here is an example of code that swaps `A` and `B` if `A` is larger:

```
a(Yb(Z)a)z(Az)y(By)
```

`a( ... a)`	While there are more elements in A...
`Y`	... increment Y (in effect copying A there), and ...
`b(Z)`	... move one element from B to Z.
	At this point, $Y = A_0$, $Z = min(A_0, B_0)$, $A = 0$, and $B = B_0 - Z$
`z(Az)`	Move Z to A
`y(By)`	Add Y to B
	Now, $B = A_0 + B_0 + min(A_0, B_0) = max(A_0, B_0)$

Now you must use this function as a comparator in a sorting network. The simplest sorting network is bubble-sort. If your comparator is as short as this one, 19 symbols, bubble-sort would be $19 \cdot n \cdot (n-1)/2 = 19 \cdot 24 \cdot 23/2 = 5244$ symbols long, which is below the limit of 5432. If your swap-min-max routine is longer than 19 symbols, you would need a smaller sorting network than bubble-sort.

Problem D: The Embarrassed Cryptographer

(Author: Nils Grimsmo)

You are are given a very large number $K \leq 10^{100}$, which you must check if has a factor below a certain limit $L \leq 10^6$. The number K does not fit into a computer word, so you must make a representation of this number yourself, or use f.ex. `java.math.BigInteger`.

You do not have time to factorise this number completely, so you can only check factors up to the given limit. To be efficient, you should only check prime factors $\leq L$. The fastest simple way to find prime numbers, is to run through the natural numbers bottom up, and each time you find a prime number, store it in an array. To determine if a number p is prime, try to divide it by every prime $\leq \sqrt{p}$, which you have already found.

The running time is $\mathcal{O}\left(L\sqrt{L}\right)$. When generating your prime numbers, you cannot use BigInteger to represent the already found primes. Then the constant factor becomes very high. Instead, you should generate the primes represented as normal integers.

Problem E: Electrical Outlets

(Author: Mats Petter Pettersson)

This was by far the easiest problem in the set. Given the number of outlets in each power strip, $O_1, O_2, \ldots, O_K$ the problem is solved the moment you realise it does not matter how you couple the power strips: Start with the wall outlet, and begin adding power strips. Each time you add a new power strip, it "contributes" the number of outlets it has, and "consumes" one outlet (either from the wall, or from another strip). The explicit formula is then

$$1 + (O_1 - 1) + (O_2 - 1) + \ldots + (O_K - 1) = 1 + (\sum_{i=1}^{K} O_i) - K.$$

Problem F: Worst Weather Ever

(Author: Per Austrin)

We have a set of years, $y_1 < y_2 < \ldots < y_n$, and the rain during those years, $r_1, r_2, \ldots, r_n$. A useful data structure to build from this information is an array `prev`, where `prev[i]` contains the last $j < i$ such that it rained at least as much during year y_j as it did during year y_i. This data structure can be built in $\mathcal{O}(n)$ time.

To answer a query $Y < X$, we first try to find i and j such that $y_j = Y$ and $y_i = X$. Using binary search, this takes $\mathcal{O}(\log n)$ time. We then get four cases:

Case 1: both i and j exist In this case, if `prev[i]` $\neq j$, the answer is `false` (this covers both the case that it rained less during year Y than during year X, and the case that there was a very rainy year Z between Y and X). Otherwise, if all years between X and Y are known (i.e. if $X - Y = i - j$), the answer is `true`, otherwise the answer is `maybe`.

Case 2: only i exists In this case, if `prev[i]` $> j$, the answer is `false` (since there is a year Z between Y and X, during which it did not rain less than during year X), otherwise the answer is `maybe`.

Case 3: only j exists For this case, it is handy to augment the data structure by also constructing a `next` array which is the converse of the `prev` array. Then this case can be handled analogously to case 2.

Case 4: neither i nor j exists In this case, the answer is always `maybe`.

All in all, the total time complexity to handle a test case becomes $\mathcal{O}(n + m \log n)$. There are also several other data structures with which you can achieve the same time complexity.

Problem G: Kingdom

(Author: Øyvind Grotmol)

This was expected to be the hardest problem in the set, and there weren't even any submissions during the contest. One way of solving the problem involves shortest distance, binary searching, and maxflow. While each step is relatively standard, it is the combination of several techniques that makes the problem difficult.

First the shortest distance from each town to each city must be determined, using either Dijkstra or Floyd-Warshall. Then a binary search for the optimal mobilizing time is performed.

The trickiest part is to test within the binary search if it is possible to obtain a certain mobilizing time T. This is done using maxflow. Let all the locations not reachable within T from 95050 be sources, and let similarly all the locations not reachable within T from 104729 be sinks. It is possible to control the goat cheese traffic with M soldiers and mobilizing time T if and only if the maximum flow in this graph is at most M. This follows from the concept of min-cut; the maximum flow is equal to the minimum number of edges that must be removed to separate the sources from the sinks.

An alternative to binary searching, is to use incremental max flow calculations. Start by running maxflow on the entire graph. Then sort the edges in decreasing order of mobilizing time. For each edge, let its capacity be unbounded and try to increase the flow using Ford-Fulkerson. When the flow exceeds M, the optimal mobilizing time is the mobilizing time from that last edge, since some soldier must be positioned on it.

Problem H: Necklace Decomposition

(Author: Andreas Björklund)

There are many ways of solving this problem. You can use the following greedy strategy: To find the leftmost necklace in the string S, you first check if $S_{0\ldots n-1}$ is a necklace, then if $S_{0\ldots n-2}$ is a neklace, and so on. If you find that $S_{0\ldots i}$ is a necklace, you report this. This would be longest necklace starting at the leftmost position in S. Then try to find the leftmost necklace in $S_{i+1\ldots n-1}$ by the same procedure, and so on, until you have found a necklace $S_{j\ldots n-1}$ for some j.

Another way of solving it is by first finding all necklaces `0*1*`. Then, as long as there exist two adjacent necklaces S and T such that ST is a necklace, you merge S and T.

To check if a string is a necklace, you do a string comparison with all rotations of the string.

Problem I: Playground

(Author: Øyvind Grotmol)

At first sight, this problem looks very intimidating. What has to be realized is that the problem simply asks whether it is possible to construct a polygon using some (or all) of the given lengths. This is possible if and only if there exists a length a_j such that the sum $s_j = \sum_{a_i \leq a_j} a_i$ of all lengths that are shorter than a_j is at least a_j, i.e. if $s_j \geq a_j$. This condition is very easy to check. Just sort the lengths, traverse the list calculating partial sums, and compare the partial sum to the length that is about to be added.