IDI Open Programming Contest April 17th, 2010

Solution sketches

A Soundex
B Sheep Frenzy
C LOL (Easy)
D Treasure Hunt
E Cross Country Race
F Beads
G Sleeping at Work
H Is it a Number (Easy)
I Proud Penguin
J Travelling Tom

Problem A

Soundex

Problem author: Ruben Spaans

The naïve solution of generating all 26^{1000} strings and testing if the string has the same Soundex code will naturally not run within the time limit.

The trick is to observe that the search space has overlapping subproblems and optimal substructure, and hence it can be solved using dynamic programming. The subproblem consists of the following information: the number of characters processed so far, how many characters in the Soundex code we have matched so far, and whether the last character was encoded into a digit or was ignored (due to being a vowel).

This can for example be implemented as backtracking with memoization. For each state, try all combinations of choosing the next letter, and call the backtracking function recursively for the new subproblem.

Problem B

Sheep Frenzy

Problem author: Eirik Reksten

This problem looks like a standard AI search problem, as we need to find the optimal way for Ulgr to move around the map eating sheep. Running a breadth-first search through all states (location and selection of eaten sheep) times out (unless it runs out of memory first), however, as there could be a total of $50 \times 50 \times 2^{16} = 163,840,000$ different states in such an approach. Instead, we need to simplify the problem.

One way to do this is to note that when moving from one sheep to another, you'll always want to take the shortest route. Since it also never will do you any good to move to other locations on the map than this, the distances between the sheep (and Ulgrs starting location) is everything we need. A good idea at this point is to give each of the interesting locations a unique identifier (i.e. a number).

In order to find the distances between all these, we can simply run a BFS from each of the locations, and note the distance to each of the others. If the BFS can't reach all interesting locations, we return `Impossible`. These 17 BFS'es has $50 \times 50 = 2500$ states each, and you'll therefore visit a total of $42,500$ states.

After having found all these distances, you are left with the Travelling Salesperson Problem on a complete graph, with the extra simplification that you don't have to return home after visiting all the sheep. Trying all possible permutations of sheep leaves us with $16! = 20,922,789,888,000$ permutations, so this is obviously out of the question. Once again, we need to discover a slight simplification. We note that the whole state can be defined as what sheep we already have eaten and which one we ate last (our position). In addition one move consists of moving to another sheep and eating it. This is a much simpler graph with a total of $17 \times 2^{16} = 1,114,112$ states. It can easily be solved with Dijkstra, or even a mere recursion with memoization.

Problem C

Laughing out Loud

Problem author: Ruben Spaans

The easiest way to solve this problem is probably to check if each string contains certain patterns. For each pattern, check if it occurs as a substring in the string. Check all patterns and the answer is the lowest number of edit operations across all matching patterns. If none of the patterns exist, the answer is 3.

- `lol`: If this substring occurs, we don't have to do anything - the answer is 0.
- `ll`, `lo`, `ol`: From each one of these configurations it is possible to add one letter to obtain `lol`, so the answer is 1.
- `l?l`: This pattern was not in the examples, so it is probably the easiest one to miss. Replace the middle character with `o`, so the answer is 1. The easiest way to implement this check in Java is to use a regular expression.
- `o`, `l`: Add the two missing letters, which is two operations.
- If the string does not contain any `l`'s or `s`, 3 operations are needed.

Solutions that try all combinations of one and two edits are also permitted. The time limit was pretty generous, so such a solution didn't have to be efficient. There is no need to check all combinations of three edits, since we can always just write `lol` in three operations.

Problem D

Treasure Hunt

Problem author: Eirik Reksten

The solution for this problem is based on the observation that an optimal fence/line can always lie infinitely close to a mine point. To see this, assume that you have an optimal line. If this line does not lie next to a mine, you can move the line directly towards the closest mine, while still not losing any treasures on the other side. At the same time, the line can lie infinitely close to a treasure on the other side (using the same reasoning as with the mine).

Knowing this, we won't need to test any other lines than those that pass through a mine and a treasure ($O(N * M)$ lines). Taking care to count treasures that lie on the line correctly (the line will after all have to be slightly rotated to adhere to the rules), the solution can then be found in $O(N * N * (N + M))$ time. This will time out, however, as $N * M * (N + M)$ can be as high as 6750000000.

Another important observation is that only the mines that are part of the convex hull of the mine points are interesting. Lines through any other mine point will have mines on both sides. This alone does not improve asymptotic running time, but it still forms the basis for the intended solution.

The intended solution lets the line "roll around" the convex hull of the mines, keeping track of which treasures lie on the outside and inside of the line. It starts with a line through two adjacent mines in the hull, and divides the treasures into two lists (inside and outside). These lists are then sorted according to the angle compared to the last of the two mine points (the pivot). It then iterates through these lists, moving the next (angle- wise) point to the other list (as when the line rotates past this point).

When the line reaches the next mine in the hull, this mine is set as pivot, the lists are resorted and the process repeated. The process ends when reaching the first pivot again. The answer will be the largest size of the list of outside treasures throughout the whole algorithm. An important consideration is dealing with collinear points. Always move these from the outside list to the inside one before the other way around (otherwise you're allowing mines to lie on the line).

As for running time, each treasure will at most be moved between the inside and outside lists two times, yielding a run time of $O(N)$ for these operations. A dominating factor is then the sorting of the lists. They will be sorted at most $O(M)$ times, since the amount of mines in the convex hull can be as many as M. The size of the lists will never be more than N, so an upper bound is described by $O(M * N * log(N))$. $M * N * log(N)$ is less than 71 million, and this will run in time.

For those who like challenges, you can try finding a solution with running time $O(M * log(M) + M * N + N * log(N))$ or better.

Problem E

Cross Country

Problem author: Eirik Reksten

Due to the relatively low amount of racers in this problem, it can be solved using a simple simulation approach (at least one racer will always be eliminated). For each race, calculate whether or not each racer will catch up with the queue in front of him. If he does, he should do another race. If not, he should not.

To check whether a racer catches up with one starting in front of him during the race, we can use the $s = v*t$ equation mentioned in the notes section of the problem description. Racer 2 will catch up with racer 1 if and only if $t_2 + d \leq t_1$, where t_i is the time racer i spends on the race, and d is the delay from racer 1 starts until racer 2 starts. Combining with the beforementioned equation, we get $d * v_1 * v_2 \leq s * (v_2 - v_1)$.

Using this equation, we can simulate race by race, racer by racer, until everyone has finished, and print the output.

Problem F

Beads

Problem author: Børge Nordli and Eirik Reksten

If you try to solve this by storing a single number for each box, and iterating through them for every single query, you could potentially end up with 30000 queries on up to 100000 boxes. This would require $3000000000 = 3 \cdot 10^9$ operations, and break the time limit. We need a data structure to facilitate faster processing of the queries, and still not lose too much on the insertions.

One possibility is to store the structure of boxes in a tree, storing a number in each node. The leaf nodes represent the single boxes, while the internal nodes represent sequences of boxes. The number in an internal node is the sum of the numbers in all leaf nodes in its sub tree.

To put beads in a box, you traverse from the root of the tree to the correct leaf, adding the amount of beads in every node along the way. This gives an asymptotic running time of $O(log(N))$ for this operation, where N is the amount of boxes.

For queries, these also start in the root. If the sequence for the node you are at is entirely contained within the query sequence, return its value. If the sequences do not intersect at all, return 0. Otherwise, return the sum of the recursive call for each of the two subtrees. This method will potentially iterate down to the leaf on both ends of the query sequence. For each end, you will visit at most $2 * log(N)$ nodes, so the asymptotic running time for this operation is $O(log(N))$ as well.

A structure such as described above is also called a Segment Tree. Another, more advanced, solution is using a Fenwick Tree (also called a Binary Indexed Tree). There are judge solutions using both variants.

Problem G

Sleeping at Work

Problem author: Ruben Spaans

This problem can be solved with dynamic programming. The subproblem consists of the number of minutes elapsed at work so far, the total number of minutes slept so far, and the number of minutes slept in the current streak.

At each subproblem, there are two decisions: Sleep during the next minute, or stay awake (thus resetting the current sleep streak). Both decisions aren't necessarily always possible; it is not possible to sleep if the required amount has been reached, or the current streak is the maximal before the boss will react.

One way to implement the solution is to use `for` loops to traverse through every state, perform each decision and update the energy level of the next state if it is higher than the previous value.

The runtime of the solution is $O(NMR)$, where N is the number of minutes in a workday, M is the required amount of sleep in minutes, and R is the longest nap you can take without getting caught by the boss. We can get away with $O(MR)$ memory usage by observing that for any given minute m we're processing, we only need to keep all states for minutes m and $m+1$ in the memory at the same time.

A faster solution is possible by defining a state as the number of minutes spent at work so far, and the number of minutes slept so far. For each state, there are now $R+1$ decisions: Don't sleep at all, or sleep for $1, 2, 3, \ldots, R$ minutes. Both solutions were acceptable for this problem.

Problem H

Is it a Number? (Easy)

Problem author: Eirik Reksten

This problem should be straightforward. Read the case input (by line), and trim() the resulting string. Verify that the remaining text consists of one or more digits (using a regular expression or simple iteration). If it is invalid, print `invalid input`. Otherwise, remove all leading zeros and print the number.

Care must be taken if the input is actually zero, in which case that last zero should not be removed before printing the output.

Problem I

Proud Penguin

Problem author: Eirik Reksten

Consider the following easier version of the problem:

Given a maximum climb height, how much water do we need to obtain this height?

This problem can be solved greedily in $O(N)$ by iterating through the track and filling in water after every hilltop (simply store the water level for each point along the track). Be sure to iterate in both directions, so that you take care of hills on both sides of the ponds. To calculate the amount of water used, just iterate on last time and sum the amounts for every segment.

Returning to the original problem, this can then be reduced to a simple search for the lowest climb height where we have enough water. Using the previously explained as a check, we can do this effectively using a simple binary search.

Problem J

Travelling Tom

Problem author: Eirik Reksten

As there are no restrictions on how many times a city can be visited on the path, this problem is really a series of shortest path searches. For every pair of cities you want to travel between, you need to find the cheapest route between these cities.

The easiest way of doing this is using a single application of Floyd-Warshall's shortest path algorithm, which finds the cheapest route between all pairs of cities. Then just iterate the travel route, and add the costs to find the answer. The running time of Floyd-Warshall is $O(N^3)$.

Another solution is to use a series of N one-to-all shortest path searches (ie. Dijkstra). As Dijkstra is $O(N^2)$ worst case, this approach will run in $O(N^3)$ as well.