Nordic Collegiate Programming Contest
NCPC 2007

October 6th, 2007

Solution sketches

Problem A

Phone

Problem author: Øyvind Grotmol

Here are two different methods for solving this problem:

Method A - Use of set First add all phone numbers to a set A. Then for each of the phone numbers, remove the digits one by one from the end and check if the remainder is in A. The standard libraries typically provide efficient enough set structures based on either hashing or a balanced tree.

Method B - Sorting First sort the phone numbers lexicographically. Then check each phone number if it is a prefix of the next number in the list.

Problem B

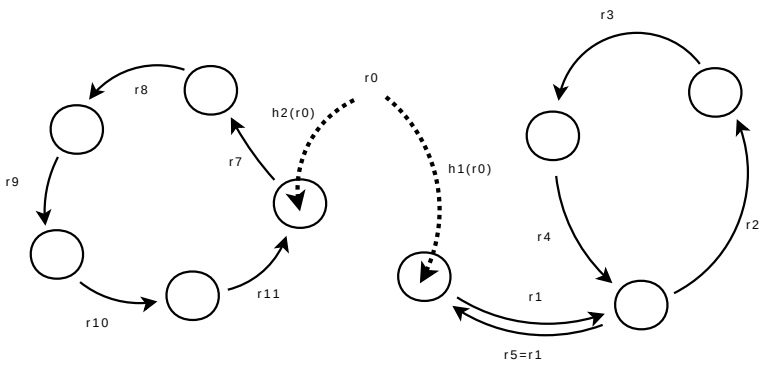
Cuckoo Hashing

Problem author: Andreas Björklund

The intended solution is to run the algorithm described in the problem statement. The only cumbersome step is to detect when the insertion process is trapped in an infinite loop.

A hacker's solution would be to read all test cases and simulate them in parallel, one move at a time and after some carefully set maximum time judge every non-finished tables as "rehash necessary".

An easier solution follows from some reasoning. Say you are trying to place word r_0 and and it gives rise to an infinite loop. The insertion process first evicts $r_1 = T[h_1(r_0)]$ and sets $T[h_1(r_0)] = r_0$. You then proceed to place r_1 in its alternative place evicting the already resident word r_2 and so on. The infinite sequence $(r_0, r_1, \cdots)$ is bound to get back to an already moved word $r_k = r_l$ for some $k > l$



since the number of words are finite. Furthermore, if $r_k = r_l$ then $r_{k+1} = r_{l-1}$ because now you are trying to put $r_k = r_l$ back into its original place and there lies r_{l-1}. So eventually you reach $r_p = r_0$ for some $p > k$ and you evict r_{p+1} from $T[h_2(r_0)]$ and set $T[h_2(r_0)] = r_p$ trying the second hash function for r_0. But by the same argument as above you will at some point get $r_q = r_p$ for some $q > p$ and we are back where we begun, *almost*: We again want to put r_0 in $T[h_1(r_0)]$ but the table T is not identical to what it was the first time! However, a little more thought (exercise!) reveals that at r_{2p} you are back in exactly the same situation as you were when you first started trying to insert r_0. Thus we have a simple test for infinite loops: when we get back to the original situation of trying to place r_0 in $T[h_1(r_0)]$ we break proclaiming the existence of an infinite cycle!

Problem C

Optimal Parking

Problem author: Nils Grimsmo

It is obvious that it is not optimal to park left of the left-most store or right of the right-most store. When parking between these boundaries, you will always have to walk from the car to the left-most store and back, and then from the car to the right-most and back. This means you will always walk twice the distance from the left-most to the right-most store.

Problem D

Copying DNA

Problem author: Nils Grimsmo

This problem can not be solved greedily or with dynamic programming. You must test every order of building the parts of the target string. However, you can get a big speed-up from memoising the optimal answers to partial solutions, as many branches in the brute-force search will end up in the same state. Also, when trying to build a part of the target string starting at a given position, it will always pay off to build as large a portion of the string as possible.

Problem E

Circle of Debt

Problem author: Andreas Björklund

The solution is by dynamic programming, iterating over the six note and coin denominations. The idea is to observe that in an optimal solution, notes and coins of the same denomination cannot be given from more than one person to more than one person. The reason is that any such transaction can be reduced to one of the same balance with less transfered money. Thus for each note and coin denomination it is sufficient to investigate the two cases

1. When one of them gives money of the present denomination to one or both of the others.

2. When one of them receive money of the present denomination from the other two.

The next observation is that it is sufficient to keep a state consisting of a pair

$$(a = \text{how much money does Alice got}, b = \text{how much money does Bob got})$$

after k of the six note and coin types have been investigated, i.e. which such pairs (a, b) are reachable by exchanging money of the first k denominations. Note in particular that we do not need to keep track of how much money Cynthia has since that is always the remaining money (their total money minus $a + b$).

Furthermore, it is only meaningful to consider states which are possible to reach with the remaining money. Quick feasibility tests include checking if the sum of the remaining money is enough and if the state is the same as the target state modulo the greatest common divider of the remaining denominations. Here and otherwise it is wise to start with smaller denominations first.

To speed things up even further one can employ a meet-in-the-middle approach. Divide the six denominations in two groups and build two separate pair tables of the above kind, one with money of the first set of denominations and one with money of the set of the last denominations. Then combine the two tables to find the optimal transaction.

Problem F

Full Tank?

Problem author: Nils Grimsmo

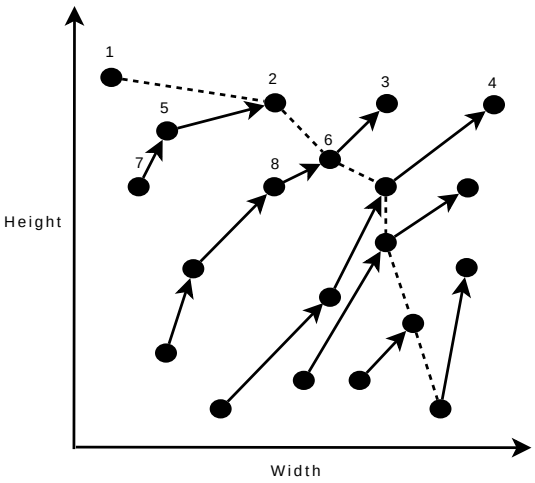
This problem can be solved with shortest-path. The main point is that a node represents being in a given city with a given tank fill. From a given node, you can go to the neighbouring nodes representing being in a neighbouring city with the current tank fill minus what is needed to travel there, and to the neighbouring node representing being in the current city with 1 more litre of gas on the tank.

Since the graph is sparse and large, you have to implement Dijkstra with a heap to make sure the solution is fast enough.

Problem G

Nested Dolls

Problem author: Andreas Björklund



This problem is a bipartite matching problem: try to put as many dolls inside another dolls as possible. However, the standard text book algorithms for the general bipartite matching problem will most likely be too slow. Instead one has to figure out what structure an optimal solution has. It turns out it is possible to find an optimal nesting by a greedy approach after some careful measures. Order the dolls after decreasing height and sort ties after increasing width. Now enumerate the dolls and put each doll in the doll with lowest width in which it is possible (it fits and the doll in question is still free). If several free dolls have the same smallest width take the tallest one. The reason this works can be seen from a contradiction argument. Assume we have an optimal nesting which doesn't have the above property. Then there exist a doll d of lowest number which is not placed in the doll of lowest width d_{low} which would be free at the time d is considered by the greedy algorithm, but is instead (possibly) placed in another doll $d_{opt?}$. But then you could easily exchange the doll d_{wrong}

possibly contained in d_{low} for d simply because d_{wrong} fits in $d_{opt?}$ since it is both shorter and narrower than d_{low} and hence shorter and narrower than $d_{opt?}$.

To quickly find the taller free doll with smallest width larger than the present doll you only need a sorted vector containing the free dolls and binary search. Every time you put a doll in another you update the corresponding vector element to the smaller dolls' width. Observe that the order of the elements in the vector cannot change, you will at worst add new free dolls at the end of the vector whenever no free doll large enough exists.

Another way is to find the largest set of dolls such that none of them can be put in another (dashed in figure). This is obviously a lower bound on the number of nested dolls required but it is also the upper bound, a result which holds generally for partially ordered sets under the name of Dilworth's theorem.

Problem H

Shopaholic

Problem author: Truls A. Bjørklund

This problem can be solved with a greedy approach. You will get a discount on a third of the items, and you want the sum of the costs of these items to be as high as possible. By buying the three most expensive items at the same time, you get the 3rd most expensive item for free. Following the same approach for the remaining items will maximise the overall discount.

This approach is most easily implemented by sorting all items, and summing the cost of every third item.

Problem I

Moogle

Problem author: Øyvind Grotmol

This problem requires a standard dynamic programming solution (or memoisation, whichever you prefer to implement). We want to calculate for every $0 \leq x \leq h$ and $2 \leq y \leq c$ the function $f(x, y)$ which is the minimal total interpolation error for the first $x + 1$ houses when storing at most y house locations among these. Then basically $f(x, y) = \min_{i=1}^{x-1} f(i, y - 1) + e(i, x)$ where $e(i, x)$ is the interpolation error you get for houses $i + 1$ through $x - 1$ when storing only house locations i and x. This e function must be either pre-calculated or memoised for efficiency.