IOI Squad March Contest, 2009 Solutions

IOI Squad 2009

1 Hot Dates

This is the stable marriage problem. A na ive solution would be to try every possible matching, testing whether the matching is stable. This method is $O(N!)$ and should score 30%.

We can do better than that. To do this, we give start with everyone being single. We then iterate through the males. If a male is single we pick any female. If she is available, then we engage them. If not, then we check to see whether she would prefer to be with the proposing man over the one she is already engaged to. If she prefers the proposing man, the two of them get engages and and the man that she was engaged to is set to single. Otherwise, we move on to the next female on the males list. This continues until there are no more single men.

There is, however, another problem. You are given survey scores, not preferences. Thus you need to sort the male preferences. You do not have to sort the females as keeping track of their current scores on the male they are paired up with will be sufficient.

This sorting has an efficiency of $O(N^2 \log N)$ as there are N sets of data each with N elements. The matching can be done in average case O(N log (N)). This gives the entire algorithm an efficiency of $O(N^2 \log N)$.

2 Big Numbers

The naive brute force, where you try all the numbers from 1 to $N^M - 1$ and tested them, incrementing the count if you found a beautiful number, scored 30%, provided the solution was properly implemented.

To get 50% you had to use pruning. To count the number of Beautiful Numbers you have test all the numbers of length 1 through to M, but most of these numbers don't have consecutive digits that differ by 1, meaning you are testing a lot of unnecessary numbers. An important observation you can make is that if you have a number of length x, that satisfies the afore mentioned criterion, then you can easily generate up to 2 numbers of length $x + 1$, by taking the last digit of the number and adding and subtracting 1 from it(being careful not to have digits less than 0 and greater than $N - 1$). So if you have the number "12", you can generate "121" and "123", provided the N is big enough. Doing this until you get numbers of length M and by looping through all starting digits of the numbers you will generate all possible numbers of length

1 through to M. At each length interval you just have to test that each digit is used atleast once and if so, then you can increment the count. However just looping over the digits at each interval and seeing if you've used all the digits would have scored you only 40%. To get the full 50% you had to use a bitmask, that you updated at each interval.

And to get full marks you had to use DP. C_{abc} is the count(modded by 1000000007) of numbers with the last digit a, with length b and the digits used c (as represented in a bitmask, i.e. a number from 0 to $2^N - 1$), where the numbers satisfy the condition that consecutive digits differ by 1.

The starting values are: $C_{i(1)(2^i)} = 1$ for all i from $1 \to N-1$. For all other values in the table the starting value is 0.

$C_{abc} = (\sum_{i=0}^{2^N-1}(C_{(a-1)(b-1)(i)} \times (i|2^{a-1} == c))) + (\sum_{i=0}^{2^N-1}(C_{(a+1)(b-1)(i)} \times (i|2^{a+1} == c)))$

Then the number of beautiful numbers is equal to $\sum_{i=0}^{N-1} C_{iM(2^N-1)}$.

3 Connecting the Grid

This problem is essentially incrementally finding cycles in an undirected graph. One way to do this is to perform a depth-first search after every move and check to see if the cycle count has increased since the last edge has been added. This would get you about 30%.

Alternatively, you could keep an array of the blocks on the board indicating to which connected component each block belongs. When an edge is added, the array is searched and the blocks belonging to the component of the one end-point of the edge are updated to belong to the same component as the other end-point. If the two end-points are already in the same component, then joining them will create a cycle. Storing the points in an array will get you 40% and using a hash table will get you 50%.

The key observation is that this technique can be augmented by using union-find, but due to memory issues this will still fall short of full score, getting you 70%. The 100% solution is to use union-find and to store the vertices in a hash table as they are used.