UT Dallas Programming Contest 4-13-2013
Explanations of Problem Solution methods

Ivor Page

A Flip Five

This is a good example of the need to step back and think about the problem before diving in. In its most elementary form the puzzle (not the problem) has a 9 bit binary number as input (the user's clicks) and a nine bit binary number as output (the black and white squares pattern created.)

It isn't obvious at first that every pattern is reachable from the all-white starting point. In fact, in the 4x4 version, and 5x5 version not every pattern can be reached from the all-white starting point. In both of those versions a pattern can be reached by more than one set of user clicks. Also, it should be pretty clear that clicking any square more than once is not useful.

The approach that I took was to think about the number of states, 2^9, and realize that I could generate every possible (reachable) state from the all-white start state.

So I thought about creating a table with the index being the click-pattern, the 9-bit binary value (an int in $[0, 2^9 - 1]$) representing the user's clicks. The contents of row n of that table would be the pattern generated by the click pattern n. For example $000,000,001_2$ is the index for a single click of the top left cell (cell zero), while $000,000,111_2$ is the index for clicks of all the three cells in the top row (cells 0, 1, and 2).

To facilitate constructing the table I created an array of these 9-bit ints that represented the cells of the puzzle that changed state when each of the nine cells was clicked:
static int parts[] = 11, 23, 38, 89, 186, 308, 200, 464, 416;
(when you click cell 0, cells 0, 1, and 3 change state, giving pattern $2^0 + 2^1 + 2^3 == 11$)

To get the table contents for $000,000,111_2$ I took the Ex-Or of 11, 23, and 38..

My intention was to read a pattern from the input, map it into a binary number, and then look up the solution from my table. That would have required searching my table for the pattern. The table was the inverse of what I wanted.
(I hadn't started coding at this point)
I switched my table around so that the index became the binary number representing the goal state, the pattern given in the problem, and the content became the click-pattern that created that given pattern.

Having created this table, I could read each problem scenario and look up the click-pattern that produces it, then count the number of 1s in that click pattern. (It wasn't worth counting the 1s until I had a particular problem to solve).

ALWAYS CONSIDER SOLVING EVERY POSSIBLE INSTANCE OF A PROBLEM.

Since 2^9 isn't very large, this was easily feasible.

Modern computers can do about 100 mill ion simple steps in one second!

Back-tracking will almost certainly lead to TLE.

B We're low on ink again

This problem is straight forward, the only issue being conversion of an int to a string in any radix (or base). I didn't use it, but Java has a function for that:

```
String g = Integer.toString( i, radix /* radix in [2, 36] */ );
```

C Zeros

I kept the upper limit of the input numbers low so that the naive approach of iterating through the numbers from 1 to n, converting each value to a String, and counting the zeros, would work.

There is a much better way to go. Given the number, 12304560789 for example (way larger than required in the problem), you work on each column and compute the number of zeros that that column contributes. For example, the result for the column containing 7 is based on the prefix, 12304560 and the power of ten for the column containing the 7, which is 10^2.

Example, for the input 456, the 6 has prefix 45 and is in the 10^0 position. It contributes 45 zeros. The 5, has prefix is 4, and it is in the 10^1 position. It contributes 40 zeros. The 4 has no prefix and therefore contributes no zeros. The answer is $45 \times 10^0 + 4 \times 10^1 = 85$ zeros.

The computation is different if the column you are considering contains a zero.

Consider 4056. The 6 column contributes 405 zeros, the 5 column contributes 40×10^1 zeros, BUT the zero contributes $(4-1) \times 10^2 + 56 + 1$. That is, the (prefix-1) times ten to the column weight plus the suffix, plus 1 = 357. Each column containing a zero must be treated this way. The total number of zeros for 4056 is then 405 + 400 + 357 = 1162

Example, for 100456 we get 10045 + 10040 + 10000 + (9000 + 457) + (0 + 457) = 39999.

The result of this analysis gives a program that can manage huge numbers in time linear in the number of digits.

D Nessie

This problem is the easiest of the bunch, but is beautifully obfuscated by the diagram.

E Empress Cindy

Another trivial problem. Sort the bills from the purse and subtract them from the amount to be made up, starting with the largest, until you get the amount needed or you fail to get it.

Example, with bills 1, 1, 3, 9, 9, 27, 27, 27, and the amount 67, we subtract the first 27, then the second. Now we have 13 Cins left. Subtracting the 3rd 27 causes a negative results, so we add back and subtract the 9. The 2nd 9 causes a negative result. We have 4 left. We subtract the 3, then the 1.

F Mars Rover

There was a small error in the problem. The final destination was limited to [200,200], not [100,100] as stated in the problem. This is the simple addition of rooted vectors, made a little worse by having to parse the input and to convert from degrees to radians.

G Squares

There was an error in the problem. N has to be in [2,1000], not $1 \leq N \leq 1000$ as stated in the problem.

There is a famous algorithm for finding the closest pair of points in a point field in 2D, Euclidean space, that runs in $O(NlogN)$ time. It's a divide and conquer recursive algorithm. that isn't difficult to code.

Here is the description from Wiki:

1. Sort points according to their x-coordinates.
2. Then, At each level of recursion,
3. Split the set of points for that subset into two equal-sized subsets by a vertical line x=xmiddle.
4. Solve the problem recursively in the left and right subsets. This yields the left-side and right-side minimum distances dLmin and dRmin, respectively.
5. Find the minimal distance dLRmin among the pairs of points in which one point lies on the left of the dividing vertical and the second point lies to the right.

Step 5 only requires examining the points within the strip to the left and right of the x=xmiddle line of width xwidth = 2min(dlmim, dRmin). We sort the points in that strip by their y coordinate and examine the pairs until the y difference is larger than xwidth. There can only be 6 comparisons to make on each recursive call.

The final answer is the minimum among dLmin, dRmin, and dLRmin.

Once you have the closest pair, you need the max of the differences between their x and y coordinates: $max(|x2 - x1|, |y2 - y1|)$.

I limited the judge's dataset so that the more naive approach of doing n^2 comparisons would work within the 5 second time limit.