Problem Solutions
09.22.2006

Some code is in JAVA, other code is in C++. In general I would recommend C++ for USACO, though JAVA can be sometimes better for Topcoder due to better string manipulation.

1 Problem 1: dsum

The most obvious solution would be to simply take the digits and sum them repeatedly. If $S(n)$ is the sum of the digits of n, then, since $S(n) \leq 9 * (\log_{10}(n) + 1)$, we can see that in the worst case we will need to sum the digits at most 3 times, since $S(n) \leq 9 * (\log_{10}(2^{31} - 1) + 1) < 84$, so $S(S(n)) \leq 16$ (the largest sum occurs when $n = 79$), and so $S(S(S(n))) \leq 9$. Therefore, this solution would be fast enough. However, there is another solution that is even faster and easier to code. If $n = b_0 + 10b_1 + 100b_2 + \ldots$, then $S(n) = b_0 + b_1 + b_2 + \ldots$, and $n - S(n) = 9b_1 + 99b_2 + \ldots$, meaning $n - S(n)$ is divisible by 9, so n and $S(n)$ have the same remainder when divided by 9. This means that the result upon repeatedly summing the digits of n will be the remainder when n is divided by 9, unless that remainder is zero, in which case it will be 9 (we also need a special case for $n = 0$, though the code below nicely takes care of it).

2 Problem 2: inbox

For any given page, we have a few possibilities. If there is no junk mail on that page, we can simply continue. If there is junk mail, we can either select all the junk mail and delete it, or select everything, deselect the good mail, and delete the remainder. We can simply take the minimum clicks among these two for each page, as the choices are independent from each other, to determine how many clicks it takes to remove the junk mail for a given page size. Then we just loop through each page size and take the minimum of all of them. Since there are at most 50 sizes to try, and at most 50 messages, our algorithm will be more than fast enough. (Note: this is an example of a *greedy* algorithm, which will be the topic of the next meeting.)

3 Problem 3: sudo

The most straightforward approach would be to just try all possibilities and count up the solutions. As it turns out, this is fast enough since there are at most $4! = 24$ ways to arrange a single 2x2 square, and so at most $24!^4$ possibilities, which is easily small enough to run in time.

4 Problem 4: tarea

The first important thing to know is that the area of a triangle in terms of its sides is $\sqrt{s * (s-a) * (s-b) * (s-c)}$, where $s = \frac{a+b+c}{2}$. If we perform a search in which we keep track of the unused sides, we can just recursively pick 3 unused sides and take the best result. This is too slow, but we can speed it up significantly if we keep track of which configurations of unused sides we have already visited so that we don't have to calculate them again. There are only 2^{16} such configurations, and at each point we need to make at most $\binom{16}{3}$ decisions. This cuts it close to the time limit, but we can optimize further by noting that the order we pick stuff in doesn't matter, so without loss of generality the first unused sides is one of the sides, or we never use it. This reduces our operations to $\binom{16}{2}2^{16}$, which easily runs in time.

5 Problem 5: mview

The size of the input data indicates that we will mostly likely need an $O(N)$ solution to this problem. A straightforward, but too slow, solution would be to keep track of all of the heights of the triangles at any given (integer) point. This algorithm could work if we modify it to require less data (that is, if we can find a way to get by without keeping track of all of the heights, but only some of them). We can first of all ignore any triangles that are completely contained within another triangle. A key insight is that, since all of the triangles are similar, a triangle that starts after another triangle can only intersect it on its way down. In addition, if three triangles, A, B, and C, all intersect but none of them are completely contained in the others, then they will become both visible and invisible in the order that they start, and they will also end in the order that they start. This means that at each x coordinate we need only keep track of the currently tallest triangle (at that x coordinate), the triangle that ends last of any triangles that have started so far, and the points at which any triangles that have started so far become visible. Then, whenever we get to a point when another triangle becomes visible, we set that triangle as the tallest triangle, and whenever a new triangle starts, we check to see if it ends later than the triangle that currently ends last.

6 Problem 6: dprod

We can first break the given number into its prime factors. Here zero seems to be a special case, so if we are given zero we can simply to check to see if any of the digits are 0. If one of them is, the answer is 0, otherwise the answer is 1. If the number has any prime factors other than 2, 3, 5, or 7, it cannot be the product of digits, since each digit only has prime factors of 2, 3, 5, or 7. Otherwise we can generate all combinations of digits with the desired product, find out how many alterations of our digits are necessary for each one, then pick the minimum of all of these. To get an upper bound on how many combinations there are, first note that 5 and 7 have 5 or 7 as prime factors, so we only need to worry about 2's and 3's. To get m 2's and n 3's, we can pick at most $min(m, n)$ 6's, $\frac{m}{3}$ 8's, $\frac{m}{2}$ 4's, and $\frac{n}{2}$ 9's. The rest of the digits are uniquely determined by the remaining factors. As the number is less than 2^{63}, there are at most 63 2's and at most 41 3's in its prime factorization, yielding an upper bound of $min(m, n) * \frac{m}{3} * \frac{m}{2} * \frac{n}{2} \leq 41 * \frac{63}{3} * \frac{63}{2} * \frac{41}{2} \leq 556000$. Since we can determine how many swaps are necessary very quickly (about 20 operations, looping through each of the digits 0 through 9 twice assuming we precalculate each digit's prime factors), our program uses less than 10 million operations (actually it uses much less than that, as our bound was very liberal) and so should run in time.

7 Problem 7: hypno

First, it doesn't matter what order we perform the operations in. Secondly, we can ignore subtracting as it is the same as adding 9 times, and we can without loss of generality assume that we add between 0 and 9 times, as 10 additions is equivalent to 0 additions. Finally, for any given configuration, we can determine the maximal sum if we only alter the rows, and precalculate this value for all configurations. Since the rows are independent of each other, we only need to store the maximal sum for each individual row configuration, of which there are 10^6. Then we can loop through all alterations of the columns and the main diagonal, and take our precalculated maximal value for each configuration, and return the best one. This is 10^7 operations, which might run in time but is probably too slow since our inner loop is very costly. To fix this, we note that, without loss of generality, we can stipulate that the first column is not altered at all, as altering every single

row, then altering each of the other four columns one less time will have the same effect. This cuts our operations down to 10^6. Finally, since the modulus operator is somewhat slow, and we know what values we will use it over, we can speed up the inner loop a little bit by precalculating its value.