Florida High School Programming Series
2014 Playoff

Problem Writer's Analysis by Arup Guha

Problem 1: Dallas Buyers' Club

This problem requires a greedy solution. If we want to have the best chance of matching a subsequence to a string, we want to take the first character in the subsequence and find the very first occurrence of that character in the string and greedily match them. Any other alternative solution leaves fewer options in matching future letters. Thus, it can be proved that the greedy algorithm must do at least as well as any competing algorithm. Once we make this realization, we iterate this process, advancing to the next character in the subsequence each time and subsequently searching for the very first spot after the previous match in the string where that character occurs in the string. If we end up matching all the characters in the subsequence before the string runs out of characters, then the subsequence can be found. Otherwise, it can't. The key issues that might occur are out of bounds errors in the implementation of this algorithm. Good test cases include the same string itself, a string and the same string with a letter added and vice versa, and other close test cases that come near matching all the letters and either barely do so or barely fail. One very short case is good to test as well (one letter and one letter).

Problem 2: Building Olaf

This problem is meant to be either the second or third easiest in the set of problems. Nothing is required but some mathematical problem solving, the formula for the volume of a sphere and knowledge of a mathematics library. The key realization is that if the radii of the snowballs making the snowman are in a given proportion, then the ratio of the *volumes* of those snowballs are in a cubic ratio to those originally given. So, for example, if the ratios of the radii are 1: 3:7, then the ratios of their volumes is 1^3: 3^3: 7^3 or 1:27:343. This volume information is critical to ascertain the volume of the smallest snowball. In this example, the volume of the smallest snowball would be 1/371 of the total volume of the snowman. Once this idea is determined, the solution to the problem is relatively straight-forward.

Problem 3: The Great Gatsby

This problem can be solved in multiple ways. One way would be to take the input as it is, and sweep through both schedules, with pointers to each time interval in each schedule, maintaining the various durations of time in iterating to the next "important" time period. While this solution is more efficient, especially for large data that spans millions of minutes, it's also more error prone. Since there are always 1440 minutes in a day, we can simply convert our original input into a list of 1440 integers, 1 for Tom and Daisy's location at each minute of the day. Once the data is stored in this manner, it's a relatively easy task to solve the problem at hand: Simply iterate through both arrays, adding 1 to a counter for each pair of elements that are distinct. Consider the following example of 10 minutes of time:

Tom	2	3	4	2	2	2	3	3	3	2
Daisy	3	3	3	4	4	2	2	2	2	2

In this case, running a loop index through this section of array, we would add one to our counter for minutes 0, 2, 3, 4, 6, 7, and 8 for a total of 7 minutes that Tom and Daisy are apart during the 10 minutes listed.

Problem 4: Hunger Games

I felt that this was the most difficult problem in the set. After carefully reading the question, one can determine that was is desired are the coordinates of the incenter of the specified triangle. With previous knowledge this is a well-known problem that is easily solved. Without information about this formula, the process required to solve it involves a good deal of coordinate geometry for the high school level.

The standard solution to this problem involves finding two angle bisectors of the three angles of the triangle and finding the intersection of those two lines. Using the atan2 function, one can find the angle heading of each of the angle bisectors. Coupling this information with the point of the triangle that defines the angle, we can determine the vector equation of an angle bisector. Do this for two different angles of the triangle and then run the regular line intersection code. Typically, a system of two equations can be set up to solve for the intersection of two lines. It's usually best if these equations are expressed in vector form instead of Cartesian form. The latter increases the chance of divide by 0 errors, since vertical lines have infinite slope.

My solution avoided the standard coordinate geometry method and tried to utilize more geometry. One can solve for the radius of the inscribed circle fairly easily, given the lengths of the sides of the triangle. (My slides that accompany this write up show how to visually do this.) Once we know this radius and the length of the portion of the right triangle that connects a vertex of the original triangle to the foot of the smaller triangle where the radius of the inscribed circle is the height, we can use these two lengths and knowledge of the angle in which we are heading to use vectors to "move" from a vertex of the original triangle to the incenter. Thus, this solution finds a starting point and two translation vectors, that, when added to the starting point, arrive at the incenter.

Surely, other solutions exist as well!

Problem 5: Seating Arrangements

This problem is definitely one of the three hardest problems in the set. A cursory read tends to indicate that recursion may be involved in breaking down the problem, but a very basic recursive breakdown like a solution to factorial will not work due to the seating restrictions. There are multiple solutions that work. I will outline two of those solutions here.

The more straight-forward but time-consuming approach would be to use recursion to go through all possible sets of pairs, counting the number of valid sets that arise. In this solution, the recursive function would take in a boolean array storing which seats are currently taken and return an integer representing the number of different ways the remaining couples can be arranged to sit. For example, if the input to the recursive function was as follows:

True True False True False True False False

then our goal would be to figure out the number of different ways the remaining two couples can occupy positions 2, 4, 6 and 7 (using 0 indexing from the left). To solve this sort of subproblem we do the following:

1) Find the first untaken seat (in this specific example, 2)
2) Start a counter set to 0 to count the total number of arrangements mentioned.
3) Iterate through the remaining seats, skipping the next seat (in this case, starting at 4)
4) For each seat not taken, place both people from a couple in these two seats and recursively call the function with this state of the seating array. Take this return value and add it into the counter for total solutions. Remember to reset the seating array in between trying each different arrangement.
5) Return the value of this counter once the loop placing the second person finishes.

Another more efficient solution is as follows:

Write a recursive function that places people in the seats left to right. The recursive function will take in three parameters:

1) The number of couples where neither person has been placed.
2) The number of couples where one person has been placed.
3) Whether or not the last person placed was the first or second person in that couple.

If no one needs to be placed, one solution satisfies the query. If either value for number of people to be placed is negative, return 0. Solve the recursive case as follows:

In all cases, it will be valid to place a person from the new couple in the next available slot, since they are guaranteed not to be from the same couple as the previous slot. Since we aren't distinguishing between couples, our recursive function call will return the number of ways to seat the remaining people. Namely, one fewer couple will need to be seated, one extra single will need to be seated and the last person placed was the first person from that couple. The answer to this recursive call is equal to the number of different placements where a new couple starts at this slot and this value should be added to a running counter of total solutions.

Alternatively, we could place a person from a couple previously placed in this slot. If we do this, then we need to determine whether or not the previous person was the first person in a couple or the last. If it was a first person from a couple, then one of the possible choices of people are ineligible. If it was a last person from a couple, all choices are eligible. Here, the choice of which person matters because it will be matching with a previous partner at a specified location, so we need to take the recursive call result and multiply it either by the number of singles or the number of singles minus one, depending on that third input variable.

As an example,

Define f(couples, singles, false) to be the number of solutions where couples is the number of couples where neither has been seated, singles are the number of people who's partners have already been seated and false indicates that the last person placed ended a couple, equals

f(couples, singles, false), = f(couples-1, singles+1, true) + singles*f(couples, singles-1, false)

Alternatively

f(couples, singles, true) = f(couples-1, singles+1, true) + (singles - 1)*f(couples, singles-1, false)

If we are placing n couples, our initial recursive call would be:

f(n, 0, false)

since we still need to place n couples and the previous person placed (no one) was not the first person in a new couple.

Problem 6: Movie Trip

This is the banger in the set, but it's more difficult than last year's banger. Most contests gradually ramp up in difficulty over the years and I wanted to that a bit with this contest. So, instead of only requiring integer input, I made the easy question have floating point input. In addition, I required a specified number of decimals to print out, which is certainly non-trivial in many languages. I believe intuitively, all students in the contest know how to solve this problem. Thus, solving it becomes showing a level of competence with the various parts of a student's chosen language. In short, this is probably the only question in the contest that is more of a syntax question than a problem solving question.

Problem 7: Twelve Years a Slave

This problem requires finding the shortest distances between various vertices in a graph. Since the graph size is no more than 300 vertices, Floyd-Warshall's Algorithm, which solved the all pairs shortest path problem in $O(V^3)$ time where V is the number of vertices, will suffice to use as a tool to solve this problem.

Basically, if you run Floyd-Warshall's Algorithm once before processing the input, you'll have a list of the shortest distances between all pairs of vertices. Then, for each query, if you want to determine the number of different movie theaters you could visit in between a trip from vertex start to vertex end, simply iterate through each movie theater (the odd vertices) and see if

```
d[start][theater] + d[theater][end] <=
timealloted - MOVIETIME
```

if this inequality is satisfied, add one to a counter keeping track of the number of valid movie theaters. Whent his finishes, your counter will have the correct value for the query.

The goal with putting this question in the set was to encouarge the best students to look up Floyd Warshall's Algorithm so that they can learn it for future contests. It's relatively easy and short to code and is a very, veyr powerful tool to have in your toolbox.

Problem 8: The Wolf of Wall Street

To maximize expectation, the greedy choice of betting the most on the game you're most likely to win is the strategy to take. Mathematically, it's easy to prove that doing so is at least as good as any competing choice that can be made. Intuitively, I expected most students to make this leap without a formal proof. What turned out to be the difficulty in this question was the definition of expectation. I had assumed that most teams would have at least one student who remembered the formal mathematical definition of expectation from school. Unfortunately, it turned out that this wasn't the case. Once a question was asked and the judges issued a clarification, more teams quickly solved the problem, indicating that their stumbling block was knowing to use the mathematical definition of expectation. In short, if a wager is W dollars and the probability of winning that wager is p, then the expected return on that wager is $Wp - W(1 - p) = W(2p - 1)$. Bascially, the outcome of winning is +W dollars which occurs with probability p and the outcome of losing is -W dollars which occurs with probability 1-p. Once this formula is in hand, simply sort both the prices and the probabilities and sum up applying this formula to each matching pair listed.