Hi, my name is Mike Trinka, and I am one of the people that contributed to the making of this year’s Code Quest problem packet. When I was in high school doing these types of competitions, I always wanted to talk about the problems once the competition was over. Whether you are a student, a coach, or just someone who happened across this document, I hope it helps you out. The following are my ramblings and thoughts on my solutions to the 2014 problem packet.

If you have any further questions or want to talk further about anything in here, feel free to email me at [mike.r.trinka@lmco.com](mailto:mike.r.trinka@lmco.com). Enjoy!

**Prob01: When I Say Code, You Say Quest!**

We always try to have a simple warm up problem that hopefully every team can do, and this was that problem. It’s a simple test using the mod operator, but it could be implemented using a loop if a team didn’t know the mod operator.

**Prob02: Deal the Cards**

There are a couple of gotchas lurking in this problem. None are exceedingly difficult, but they can add up at debug time. First, you have to figure out a way to map between the short suit names that are given in the input file and the suit names that you are asked to print. The same goes for the face values. Once you’ve figured that out and sorted the cards into their categories, adding the s for plural items is the last tricky thing.

**Prob03: Valley Sort**

There are a ton of ways to do this problem. Here’s how I chose to do it:

* Find the number of elements on each line and create an int array for each line to hold the values as well as the valley sorted values.
* Use Arrays.sort() to sort my array in ascending order. Teams could easily implement their own sorting algorithm if they didn’t know this utility of the language.
* Start from the end of the sorted array and fill in my valley array with the valley sorted values from the outside in.

In my implementation I had to do an extra check at the end to deal with arrays with an odd number of elements because I filled in the valley sorted elements two at a time.

**Prob04: Pig Latin**

This problem was very straightforward. There are only three possibilities:

* The word has no vowels – easy because you just print the word as is
* The word starts with a vowel – easy because you just add yay to the end
* The word starts with a consonant and has at least one vowel. This is the tricky condition because of dealing with qu.

**Prob05: Book Worm**

This problem looks tricky because of the write-up, but it’s really pretty straightforward. Every line has 13 digits in it somewhere. Filtering out the numbers is the first thing. After that, using the numbers in the formula given in the write-up is the whole problem.

If the result is a multiple of 10, you’re done. If not, then you have to take the extra step to find out how far off the last digit is and figure out what it should have been to make the sum a multiple of 10. The error we saw the most on this step was not doing the final mod 10. Teams were printing out that the final digit should have been a 10, which is not a single digit. The correct answer in those cases would be 0.

**Prob06: Product of a Grid**

This was a pretty straightforward looping problem. I heard that one of our people did it recursively, but if you watch your array indices looping suffices as well. If you loop over a two dimensional array, then there are really only 4 directions to consider because the loop counters are moving, so they take care of the overlapping conditions. In other words, it’s not necessary to consider all 8 directions, only half of them.

**Prob07: Morse Code**

This problem had a lot of typing, and you had to make sure that you got the mapping of text to Morse correct, which could have been a pain. However, once the mapping is done, the problem breaks down to a substitution problem.

**Prob08: Rectangle Art**

This was the problem that was the subject of the most questions during the contest. Most questions went something like this:

“It says that the work area is 20x20, but the points in the example input go from 0 to 20, which means that there are 21 spaces. Did you guys mess up writing the problem?”

Looking at a piece of graph paper, a 20x20 grid uses the x,y coordinates from 0,0 (bottom left) to 20,20 (top right). Most kids just looked at the numbers and assumed that they were array indices.

**Prob09: Check Please!**

Once you figure out that you don’t have to store the text for more than 28 numbers, this problem becomes much easier. Most errors that we saw had something to do with when a place had a zero in it, and “Dollar” vs. “Dollars” got a lot of the teams. Parsing the number and printing it correctly was a bit tricky.

**Prob10: All Aboard!**

I thought this was easiest to implement using an inner class that I called TrainTrack. I kept the links between up and down tracks in an array of TrainTrack variables within my class. To me, the logic of running the problem was the easy part – but parsing the data into the appropriate data structure to be able to solve the problem easily was the harder part. Once I had my data structure in place, the rest of the problem followed quickly.

**Prob11: Spiral Text**

This problem looks so easy until you start doing it. First, you had to get it that the square couldn’t have an even number of spaces on a side (because then there wouldn’t be an exact center – we got a lot of questions about that part).

The part that was the trickiest for me was turning the corners when I was filling the spiral. You have to be careful not to use the corner space twice and overwrite your work from the previous segment.

**Prob12: Caesar Scytale**

This problem was so cool to me. There were a couple of gotchas in it. First, you have to loop through all the possible array sizes that could have been used to do the Scytale cipher. If an array size doesn’t exactly fit the size of the text given, then it needs to be skipped.

Once you’ve got a suitable array size, you have to fill the array and unravel it, and then strip the extra X characters off the end at this point. Then you deal with the Caesar cipher, which is a loop to try all possible shifts with a check for the word “Dear”.

The core concepts here are not too difficult, but they are wrapped in a lot of complicated looking terminology and problem write-up which may have scared some students off.

**Prob13: Diamond Path**

To me, the most difficult thing about this problem was reading in the input file. In my solution of 269 lines, 128 of those lines are dedicated to reading the input, and another 65 of them are my Node class to hold my data in the structure I wanted it. Once I built my tree, I armed myself with this critical piece of information about best path problems:

In a best path problem, every sub-path of the best path is itself a best path starting from the node in question.

Most people look at a tree and say to themselves “a simple depth first search will let me find the best path”, and they are correct. But will it run quickly enough? Starting from the bottom and working toward the top lets you make a single pass through the nodes to obtain the correct answer.

**Prob14: 3D Bingo**

This took last year’s bingo problem to another level. I used two inner classes – one for a bingo card and one for a bingo cube. The logic to check for a winning condition can give you fits, but that’s really the heart of this problem. I chose to check for bingos involving a single card separately from the cube. That’s just personal preference.

The other thing you have to make sure you do is to find all the permutations of the cards to form a cube. I chose the ugly for loop implementation, but you could use recursion just as easily.

**Prob15: Pretty Print**

This is for all you String parsing gurus out there. I really had a hard time with this problem because I do not like String parsing. My code is super ugly and it would probably take me a long time to find a bug in it. Use it at your own risk. ☺

**Prob16: Treasure Hunt**

Skip down to the “Final Solution” section if you don’t care about the progression of my thinking.

**Progression of Thought**

I started out in classic fashion – a brute force depth first recursive search for the treasure. Of course, this took forever (to run, not to code). I let it run for about 18 minutes on the judging input and didn’t find the treasure once.

I started looking for ways to cut down the amount of whitespace on the map (therefore cutting down on potential paths), so I created a method to “optimize” the map. It would fill in areas of the map that were obviously not on the best path (like a space surrounded by 3 x spaces). That made things a little better, but still not workable.

I started looking for ways to cut a path short in the middle of it. I calculated the distance from the treasure to each point on the map and used that to see if my current path would exceed the current best path length. This only works if you find the treasure, so it didn’t do anything for my problem of not finding any correct paths quickly.

Then I thought about backtracking. At first I thought that the most you should step on any space was 2. This led to me finding the treasure fairly quickly, and my algorithm completed in about 2 minutes. However, I thought some more. Consider this scenario:

xxxxx

xxtxx

xt tx

xx xx

H

In this case, the treasure hunter might need to collect all three torches before a long stretch with no more torches to collect. The space in the middle that touches the three torches would need to be stepped on 4 times. This scenario was the inspiration behind the “four steps” map. So, I increased my threshold and I was back to not finding a solution. I needed a better way.

**My Final Solution**

I thought about what the treasure hunter was really doing. He starts on his start point. He wants to get to the treasure. What stops him? Only the distance from wherever he is to the treasure combined with the amount of light he has left. If he doesn’t have enough light, he needs a torch.

This line of thinking led to my big breakthrough: the space between key squares on the board *doesn’t matter*. All that matters is the *distance* between those key squares. The whitespace between the key squares is just a distraction, because it doesn’t matter which path the hunter takes from place to place – it only matters if that path is the quickest.

Using that line of thinking, I boiled this question down to a weighted path problem. The key nodes are the hunter’s starting position, the torches, and the treasure. This led me to my artifact array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Index: 0 | 1 | … | n | n+1 |
| Starting Position | torch 1 | … | torch n | Treasure |

**String[] artifactArray**

This array always has the starting position first, and the treasure last. If there are n torches, then there are n+2 elements in the array. The array holds the coordinates of every key space on the board so I could print out the path when I was checking this by hand, and so I could compute the distances between the spaces later. The order of the torches doesn’t matter, but the indices do. From here on out, everything in the problem revolved around these n+2 spaces.

Next I needed a place to store my distance calculations, because I didn’t want to calculate any distance more than once. That led me to my artifact distances array:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Index | 0 | 1 | … | n | n+1 |
| Index |  | Start | t1 | … | tn | T |
| 0 | Start | 0 | Start🡪t1 | … | Start🡪tn | Start🡪T |
| 1 | t1 | t1🡪Start | 0 | … | t1🡪tn | t1🡪T |
| … | … | … | … | … | … | … |
| n | tn | tn🡪Start | tn🡪t1 | … | 0 | tn🡪T |
| n+1 | T | T🡪Start | T🡪t1 | … | T🡪tn | 0 |

**int[][] artifactDistances**

I used this array as follows:

* -1 means I have not calculated the distance between the two points, so I need to do that calculation and save it in both directions (because t1🡪tn is the same as tn🡪t1).
* -2 means that the two points are not connected (i.e., there is a torch surrounded by walls that can never be stepped on).
* A non-negative value is the distance from point to point, using the row and column as the start and end indices of the points I’m currently worried about. Since we store both directions, it doesn’t matter which index is the row and which is the column.

I also needed an array to tell me whether the space I was considering stepping on next was already in the current path or not. I called that array used, and it has the same number of elements as the artifact array.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Index: 0 | 1 | … | n | n+1 |
| Starting Position | torch 1 | … | torch n | Treasure |

**boolean[] used**

For my own testing purposes, I also have a current route array that stores the steps I’ve taken, along with a current depth variable to tell how many steps I have taken. These are not necessary for solving the problem, but they were invaluable during my debugging so I just left them in my final solution.

Now that I had everything set up, I just had to write the recursive solution. The pseudo code is below (I added a page break so it would all fit on a single page).

Set the start point used = true

Add the start point to the current route

Solve recursively (0, 0, 15)

Solve recursively (index we’re on in the artifact array, distance we’ve traveled, light we have left)

* Increase our depth
* Are we at the treasure?
* Yes: this path is complete
  + Is the distance we’ve traveled less than the current best path we know of?
  + Yes: we found a new best path
    - Remember the new best path distance
  + No: this path is worse than our current best – ignore it
* No: we have farther to go
  + Get the distance from where we are to the treasure
  + Can we get to the treasure with the light we have left?
  + Yes: go straight to the treasure
    - Solve recursively(treasure index, distance + distance to treasure, light – distance to treasure)
  + No: we need another torch
    - Loop through all torches: has the torch been used?
    - Yes: ignore it – no sense stepping on a torch twice
    - No: consider it as a next step
      * Get the distance from where we are to the next torch
      * Can we get to the next torch with the light we have left?
      * Yes: torch is still a candidate for the next step
        + Get the distance from the next torch to the treasure
        + Will going from where we are now to the next torch, and then from the next torch to the treasure (which is the best case) put us over the step count of the best path we know so far?
        + Yes: ignore it – no sense trying a path we already know will be worse than what we already have
        + No: try it!

Set the next torch used = true

Solve recursively (next torch index, distance + distance to next torch, light left – distance to next torch + 15)

Set the next torch used = false

* + - * No: we can’t make it – try the next torch in the loop
* Decrease my depth

This solution allowed me to solve our judging input in about 9 seconds. Just to check to make sure my code wouldn’t explode when there were a ton of torches, I made another map where I took the judging map and replaced all the spaces with torches. My code ran for about 9 minutes, but it didn’t get a stack overflow error which is what I was afraid of with so many torches.

Hopefully my explanation makes at least a little sense. Happy hunting!