Thomas Banghart
CS 165 Oregon State University
Spring 2019

Since we only had 3 members in our group (Group 4) I will review my group members code to my own submitted code. I will then review my own code to point out some obvious flaws that were discussed within my group. I hope that only having three group members does not negatively affect our grades(!).

Aileen Murphy:

We chose Aileen's code to be the group submission with good reason. Aileen's was by far the most readable code which conquered the confusion of recursion with ease. In particular, their `vectorPuzzle()` function was really organized into two functions which helped tremendously. The `vectorPuzzle()` that would be called for grading purposes simply passed the original vector, a position, and a new vector of `used` elements to another, overloaded `vectorPuzzle()` function that handled the main game logic. This was the most creative way to handle the problem of keeping track of where the "token" had been that any of us had come up with, and, quite frankly, I would not have thought of it. Both Ryan and I had taken advantage of the fact that the vector that would be passed to `vectorPuzzle()` would be positive so we marked where we had been by turning the current position to a negative integer before making a "move".

By using an additional vector of "used" elements, Aileen was able to solve one of the more challenging aspects of this program. It was not that keeping track of elements was the hard part but making sure we were able to reliably check if an element had been visited by compartmentalizing the logic in a helper function. I had lumped both the "can move" and "had been visited" logic together in two separate functions `moveRight()` and `moveLeft()`. This made the code difficult to read as the conditional logic was filled with cryptic && statements that referenced vect[checkBounds] which isn't very intuitive. Lastly, I really like how Aileen handled the recursive call. Rather than multiple return statements (which I had based on if we could move right or left – then a `return` was made) Aileen had one 'or' return statement. This was a truly elegant solution as it allowed the recursive call to be made to check all possibilities. That is to say – we could see the outcome if we were to move right, then we'd check what the outcome would be if we moved left if moving right had failed. This is a much better solution than what I had done, but I will elaborate on this later.

Ryan Colletta:

Although Ryan's code did handle the recursive call much better than mine, I find their code the least readable of the group. There are functions whose purpose are unclear to me. For example, `debugRow()`) which is outputting characters while checking if moving the "token" would be out of bounds or not. While I am sure this is great for testing purposes, it currently clutters the code. I also was displeased with the inline `if` statements where the condition was right next to the return statement. This is certainly a common way to write if statements, but I thought that it made the code less readable. There are also no comments on their code, which may be alright since this is a simpler program, but I would have liked to see some comments to keep track of what was going on.

Ryan also manipulates the game vector itself which, although not against the program assignment, makes keeping track of the recursive call a bit confusing. Both Aileen and I never use a `pop_back()` or`push_back` function on the game vector itself and overall I think that's a better way to handle the situation. By having a "static" game vector, we're able to visualize the "token" moving across the vector more easily and avoid removing and adding elements to a vector. That said, Ryan and I both make use of multiple recursive calls, or at the very least to recursively call `vectorPuzzle()` at different points. In a similar way as I have organized my own code, Ryan allows for a recursive call after checking if the token can move right. Specifically, the call is made in an if() statement right before checking if the token can be moved to the left. This helps us to see if the token can be moved all the way to the right and prevents the fatal issue with my code that I will discuss next.

<u>Refactoring my own code:</u>

Being able to discuss my code with my groupmates allowed me to see some fundamental issues with the way I organized my program. Specifically, my recursive call only works if there is not a game vector that has a "split" decision. By "split" decision, I am referring to a vector that may have a value at an index that, if we moved right, would return false, but if we moved left, would return true. The problem that I had, and which Aileen and Ryan were able to avoid, was that I allowed it to make a recursive call before seeing if we're able to traverse the whole vector in one direction from a given index. Below is the snippet with the main issue:

```
if(moveRight(vect, position)){

                int newPos = position + vect[position];
                //To keep track of where we been, turn the old value to -1
                vect[position] = -1;

                return isSolveable(vect, newPos);
        }

        //if its not possible to move right, see if we can move left
        else if(moveLeft(vect, position)){

                int newPos = position - vect[position];
                ////To keep track of where we been, turn the old value to -1
                vect[position] = -1;

                return isSolveable(vect, newPos);
        }
```

So here we have an if-else if statement where I first check to go right and then, if we can't go right, check to see if we can go left. In doing so, I ignore the possibility that going right, rather than left, prevents a correct solution – if we can move right, we do, and then make recursive call. I should have looked more closely at this problem and done something similar to Ryan or Aileen where the recursive call is made to check if going one direction works or not, and then return true or false for that direction and then check the other. Aileen’s code is the better example of this as it is a or statement which will have to check one direction completely, and if it is false, it will then have to check the other, and ultimately return the result.