Reflection Document
Assignment 1 - Cis*3190 Legacy Systems
Brayden Cowell - 0844864 - bcowell@mail.uoguelph.ca

Key

space	wall	Start	Finish	Path	Backtrack
.	*	o	e	#	@

Instructions

Compile: gfortran a1.95
Run: ./a.out
Put a maze in the same directory as a1.95 and enter the filename of your maze.

Algorithm

```
Create an empty stack named S;
Push the start character onto S;
While S is NOT empty do
        Pop the current_cell from S;
        if (current_cell in at the end of the maze ('e')) then
            print a congratulations;
            break out of the loop
        end if;
        Mark the current_cell as visited ('#');
        if (path available to East) then
            Push next_cell and direction East onto stack;
        else if (path available to West) then
            Push next_cell and direction West onto stack;
        else if (path available to North) then
            Push next_cell and direction North onto stack;
        else if (path available to South) then
            Push next_cell and direction South onto stack;
        else (no path available)
            Mark the current_cell as backtracked ('@');
            Push the previous_cell and opposite direction onto stack;
        end if;
    end do;
```

Data Structure

For my stack module I included the size of the stack and two allocatable 1-D character arrays. One was for storing directions in order to backtrack and the other holds current characters for solving the path. Note: If the maze is unsolvable then my program will output “No solution found!” along with the attempted paths. Also contained in my module are three subroutines that use the stack; push, pop, and backtrack.

Push

The push subroutine simply pushes a character at a position in the maze, onto the stack. For parameters push takes a stack, a current_cell character, and a direction character. The subroutine then increases the allocatable arrays size by one and adds the current_cell character to the end of the array. It does this by creating a temporary stack with size of n+1 , copying over the old stack to the temporary one, and then putting the new character to the end of the new stack. Push then returns the new enlarged stack.

Pop

The pop subroutine does exactly what you expect it to. For parameters pop takes a stack. The subroutine then decreases the allocatable arrays size by one and gets the character at the end of the array. It does this by creating a temporary stack with size of n-1 , copying over the old stack to the temporary one. Pop then returns the character along with the new stack.

Backtrack

This subroutine is essentially equal to the pop subroutine. The key difference is that instead of popping the current_cell from the path it pops the last direction it traversed. I made this a separate subroutine because it is possible for the actual path stack to differ from the direction stack.

This way I was able to solving the maze non-recursively by reversing direction when I encountered a dead-end.
I chose to implement an adjustable stack size instead of opting for the easier stack route. I did this mostly for fun, as I wanted to play around with Fortran’s memory allocation.

Fortran 95

I actually started writing my maze solver in Fortran 90, but later chose to switch to Fortran 95. In Fortran 95 local allocatable variables are automatically deallocated upon exit from a subroutine. Since I am using an allocatable character stack and am changing it's size frequently, I wanted to be assured no memory was leaking. And even though I explicitly deallocated everything I allocated, I like the extra strength it provides.

Algorithm for parsing the maze from file

```
do i = 1, rows
    read string
    do j = 1, columns
        current_cell = string(j:j)
        maze(j,i) = current_cell
    end do
end do
```

I tried to prioritize the implementation of specific code segments based on their difficulty. The order looked something a little like this:

1. Print instructions and read in filename.
2. Make sure file exists.
3. Read in the dimensions of the maze.
4. Allocate a 2D array of dimension size.
5. Assign each character in the file to its position in the matrix.
6. Write the algorithm for solving the maze.
7. Create the stack data structure.
8. Create subroutines for push and pop.
9. Added subroutine for backtracking.

Questions

- What were the greatest problems faced while designing the algorithm in Fortran?

I think the greatest problem I faced was figuring out how to incorporate backtracking into my stack module. In the end I chose to have two 1D character arrays instead of multiple stacks, however it would have been possible either way. I also found Fortran's error messages to be quite unhelpful.

- Was it easy to create a data structure in Fortran?

I found it very easy to create a data structure in Fortran. Mostly because Fortran's custom types are very similar to C's structures.

- What do you think about the concept of the Fortran module?

I thought the concept of modules was very easy to understand. The functionality is very similar to a C header file.

- What particular features made Fortran a good language?

Fortran had nice array notation and seemed to excel at computing maths.
It was also very easy to read and similarly to C, I felt like Fortran fills a niche role and will not be replaced for some time.

- Would it have been easier to write the program in a language such as C?

Other than the time it took to learn Fortran's syntax, I found it quite similar to C. So I think this program would have taken around the same time write in C.

- Is there a better way of writing the program?

Other than implementing it in a language that has graphing functions prebuild I think there is no better way. I chose to use a DFS algorithm based on the size of the mazes. A BFS would have been a quicker solve if the maze were much larger and had more dead ends.

- Given your knowledge of programming, was Fortran easy to learn?

Since I have an intermediate understanding of C I found Fortran very easy to pickup.

- What structures made Fortran usable?

Fortran was basically used for scientific and mathematical computations. Fortran takes advantage of things like; instruction cache, CPU pipelines and vector arrays to provide one of the fastest compilation times for number crunching.