# Assignment 1: Maze Solver

## Objectives

For the data structure part of the assignment, you must implement a stack API (Application Programming Interface) and a queue API. For the algorithm part of the assignment, you must use these data structures to implement a depth-first and a breadth-first maze solver.

## Requirements

### Data structures

You should implement the stack API described in the `stack.h` file and the queue API described in the `queue.h` file. The stack and queue data structures will be explained during the lecture.

In this assignment, the stack and the queue data structures store integers and will be used by the maze solver algorithms to store maze locations. Locations will be non-negative numbers, so we will only use the stack and the queue to store non-negative numbers. For this assignment, the stack and queue data structures use a fixed block of memory that must be allocated with `malloc` when the data structure is initialized. Adding an element to a stack or queue that is full will fail, and the function will return an error code. To ensure that your maze solver has enough stack or queue space to solve all the test mazes you should initialize them with a size of at least 4000.

### Algorithms

You must implement two versions of a maze solver algorithm that differ in the way they explore the maze. Both versions will be explained in detail during the second lecture, here we will only give a summary:

- Depth-first search (DFS): Search from the start location following a single path but storing alternative path options along the way on a stack, until you find the destination location or until you hit a dead end in the maze. If you have arrived at the destination location, you are done. If you hit a dead end, you go back to the last alternative location you encountered and continue your search from there. Think of this as running headlong into the maze until you hit a dead end, running back to the last stored junction, and repeating the same procedure.
- Breadth-first search (BFS): From the start position, add all the neighboring locations to a queue so they can be searched later. Then take a location from the front of the queue. If it is the destination location, you are done. Otherwise add all its neighbors to the queue to be searched later and repeat the process. Think of this search method as flooding the maze from the starting position with 'water'. A 'waterfront' will slowly and evenly spread out into the maze until this front hits the destination location, finishing the search.

Although the DFS and BFS search methods behave very differently, their implementation differs only in the data structure they use to store the locations that will be searched later. If you use a stack to store these locations, you obtain a depth-first search; if you use a queue, you have a breadth-first search.

In a finite maze, the depth-first search maze solver will always find a path if one exists, but if there are multiple paths from the start to the destination, it might not return the shortest path. The breadth-first search maze solver will even work for infinite mazes and will always return the shortest path. Since you can view a maze as a graph where the maze locations are the nodes and neighbouring locations are connected with edges you can use the standard graph algorithms [1] [2] to implement your maze solver; these algorithms will also be covered in

depth in the second lecture. Although depth-first search is often implemented using the function call stack to store the search locations, your implementation should use the stack data structure from this assignment.

You must submit your work as a tarball. The command `make tarball` will create a tarball for you named `maze_solver_submit.tar.gz` that contains all the files you need to submit.

# The input and output formats

For this assignment, we provide all the functions for working with the maze in the header file `maze.h` and source file `maze.c` so you can focus on the maze solver algorithms. Read the header file carefully, so you know how to use these functions correctly.

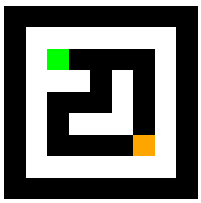
The maze solver source files `maze_solver_dfs.c` and `maze_solver_bfs.c` already contain code to read the maze from stdin, call the solver function, and finally prints the maze with the found path and path length. Let's look at a small example maze:

```
$ cat mazes/maze_7x7_multiple_paths.txt
#######
#S    #
### # #
#   # #
# ### #
#    D#
#######
```

In a maze, 'S' marks the start, and 'D' marks the destination. Walls are represented with #'s, the floor with space characters. The path should be marked with x's. Walls always surround a maze, and movement inside the maze is restricted to four directions UP, DOWN, LEFT, and RIGHT. Now let's run the provided depth-first maze solver on this small maze:

```
$ make
  [..builds the dfs and bfs mazes solvers..]
$ ./maze_solver_dfs < mazes/maze_7x7_multiple_paths.txt
dfs found a path of length: 16
#######
#S    #
### # #
#   # #
# ### #
#    D#
#######
```

Since the solver function is not yet implemented, it just prints the unmodified maze to stdout with a random integer value for the path length. The maze solver also writes the maze in the Portable Pixmap Image format to the file `out.ppm`. Every pixel in this file represents a wall (white) or a floor (black) location. The start location is marked green, the destination location is marked orange, and the path will be marked in red. Working at pixel level allows us to show very large mazes, but you will need to zoom in to see them. You can use the command `display -sample 100x100 out.ppm` to view this PPM file. You can find the `display` command in the package `imagemagick`. Remember to zoom in and turn off anti-aliasing if you use your favorite image viewer.



A depth-first search will not necessarily find the shortest path. So for this maze, the correct output of the depth-first solver is either:

```
$ ./maze_solver_dfs < mazes/maze_7x7_multiple_paths.txt
dfs found a path of length: 12
#######
#Sxx, #
###x# #
#xxx# #
#x### #
#xxxxD#
#######
```

Or it could return the actual shortest path:

```
$ ./maze_solver_dfs_different_implementation < mazes/maze_7x7_multiple_paths.txt
dfs found a path of length: 8
#######
#Sxxxx#
###,#x#
#   #x#
# ###x#
#    D#
#######
```

The breadth-first search will always find the shortest path so the only correct output is:

```
$ ./maze_solver_bfs < mazes/maze_7x7_multiple_paths.txt
bfs found a path of length: 8
#######
#Sxxxx#
###.#x#
#...#x#
#.###x#
#,   D#
#######
```

You are free to mark floor locations of the maze any way you want as long as you mark the found path with x's. This can be very useful for storing the state of the maze locations and for visualizing how the maze is explored. For example, in the output shown above a `','` indicates that the location was stored on the stack or queue for later consideration and a `'.'` indicates that the location has been searched but is not part of the final path.

The stack and queue data structures each have one function that prints data as a side effect. These are: `stack_stats()` and `queue_stats()`. They must print the word `"stats"` followed by three numbers separated by spaces in the following order to the standard error output stream:

- The total number of successful "push" operations.
- The total number of successful "pop" operations.

- The **maximum** number of elements stored at any time during the lifetime of the data structure.

# Automated Testing

Automatic grading scripts will determine the correctness of your programs. We provide some tests for the stack and queue data structures in the files `check_stack.c` and `check_queue.c`. These tests, which use the check unit test library, are compiled to the executables `check_stack` and `check_queue` and are run when you type `make check`. Add your own tests to gain more confidence that your implementation of the data structures is correct.

We provide some example mazes and expected path lengths to test your depth first and breadth first maze solver algorithms. The script `check_maze_solvers.sh` runs both maze_solvers on a set of example mazes and checks if the path is correct. The command `make check` will also run this script if the data structures checks pass.

You are encouraged to create your own test mazes to test your maze solvers. You can make these by hand or use the provided executable `maze_generator`. See `maze_generator -h` for the usage information.

# Getting started

1. Unpack the provided source code archive; then run `make`. The compiler will produce a lot of warnings and the two executables `maze_solver_dfs` and `maze_solver_bfs`. Check the warnings to ensure that they are indeed the result of the incomplete framework code we provide.
2. Now run the command `make check`. This will compile the provided test cases for the stack and the queue data structures `check_stack` and `check_queue`. It will then try to run the stack and queue data structure tests followed by the maze_solver test script until it encounters an error. Since the first program `check_stack` fails, it will stop there. You can run the tests directly by typing `./check_stack`, `./check_queue`, or `./check_maze_solvers.sh` on the command line.
3. Read the file `stack.h` and study the interface of functions listed there.
4. Implement the data structure in `stack.c` according to the interface description. Use the provided tests to check your implementation. You can find the source code from each test in `check_stack.c` to diagnose any test failures. Write your comments in English and use English function and variable names.
5. Read the file `queue.h` and study the interface of functions listed there.
6. Implement the data structure in `queue.c` according to the interface description. Again use the provided tests to check your implementation.
7. The data structures should now compile without any warnings. Check this with `make clean; make`. This will force the recompilation the data structures.
8. Read the file `maze.h`. These functions handle all maze-related operations and we provide them to simplify your maze solver implementation.
9. Implement the depth-first maze solver algorithm using the stack data structure. First test your implementation with the small example mazes, then use `make check` to test with some larger mazes.
10. Implement the breadth-first maze solver algorithm. This should be as simple as copying the depth-first maze solver algorithm and replacing the stack with the queue data structure. If all is well `make check` should run without any errors.

# Grading

Your grade starts from 0, and the following tests determine your grade:

- +2pt if you have submitted an archive in the correct format, and you have made a real effort to implement `stack.c`, `queue.c`, `maze_solver_bfs.c` and `maze_solver_dfs.c`.
- +1.5pt if your stack API handles push and pop operations correctly and detects stack underflow and overflow situations.
- +1.5pt if your queue API handles push and pop operations correctly and detects queue underflow and overflow situations.
- +1pt if your stack and queue API correctly counts **valid** operations (number of pushes, pops, and the maximum size).
- -1pt if your code produces any warnings using the flags `-Wpedantic -Wall -Wextra` when compiling.
- +1pt if the depth-first maze solver reports a correct path length.
- +1pt if the breadth-first maze solver reports the correct path length.
- +0.5pt if the depth-first maze solver prints a correct path in the maze.
- +0.5pt if the breadth-first maze solver prints the correct path in the maze.
- +1pt if your implementation has the correct style and the correct complexity.
- -0.5pt if you don't check the return value of `malloc` correctly.
- -0.25pt if you don't handle NULL arguments correctly in the stack and queue implementation.

Testing is done with the address sanitizer enabled. You can choose to run your assignment with valgrind for additional error checking. *Note that you cannot use both asan and valgrind simultaneously.* The address sanitizer is enabled by default in our makefile. To generate an executable that you can use with valgrind, you need to run the commands: `make clean; make valgrind`. You can then run your program with valgrind: `valgrind ./maze_solver_bfs < MAZE_FILE`

# Bonus

The stack and queue of the regular assignment will signal an error if they are full. If you have time left, you can score a bonus point if you resize the stack and the queue when they are full:

- +0.5pt if your stack API works correctly and you implement stack resizing.
- +0.5pt if your queue API works correctly and you implement queue resizing.

# Dependencies

All the assignments for this course use a unit test library called check. It should already be installed on your laptop, but if you get errors that mention check or `pkg-config`, run these commands to make sure these packages are installed:

```
sudo apt-get update
sudo apt-get install check
sudo apt-get install pkg-config
```

We highly recommend using a Linux machine for this course or, at the very least, testing your assignment on a Linux machine. If that is not possible, make sure that you have check and `pkg-config` installed on your Mac:

```
shell brew install check
shell brew install pkg-config
```

---

1 https://en.wikipedia.org/wiki/Depth-first_search
2 https://en.wikipedia.org/wiki/Breadth-first_search