For this program, you will write MIPS assembly language functions to solve a one-way maze using a recursive depth-first search algorithm. You will be provided with code that will construct and parse the mazes as well as the solution string. Your code will need to utilize the provided functions to interact with the existing data structures. Your function will perform the core operation of choosing how to move through the maze so that you are guaranteed to reach the other side. The algorithm that you will be coding could be implemented in C++ as the following:

```
linked_list Maze_Solver (maze m, int x, int y, int previous_direction) {
	int directions = m.get(x, y);
	if (directions & 16 > 0)
		return add_head(null, "X");
	if ((directions & 8 > 0) && (previous_direction != 8)) {
		linked_list l = Maze_Solver (m, x, y+1, 4);
		if (l != null)
			return add_head(l, "U");
	}
	if ((directions & 4 > 0) && (previous_direction != 4)) {
		linked_list l = Maze_Solver (m, x, y-1, 8);
		if (l != null)
			return add_head(l, "D");
	}
	if ((directions & 2 > 0) && (previous_direction != 2)) {
		linked_list l = Maze_Solver (m, x-1, y, 1);
		if (l != null)
			return add_head(l, "L");
	}
	if ((directions & 1 > 0) && (previous_direction != 1)) {
		linked_list l = Maze_Solver (m, x+1, y, 2);
		if (l != null)
			return add_head(l, "R");
	}
	return null;
}
```

You will be provided with code to perform the `get` and `add_head` functions, but will need to write the rest of the algorithm.

Your program will not need to interact with the console. Instead, you will use a test suite that has been provided for you. The test suite will call your function (Maze_Solver) with the parameters ($a0 contains the address of the maze object, $a1 contains the parameter x, $a2 contains the parameter y and $a3 contains the direction that you just moved) and will wait for your function to return its results in $v0. The test suite will also tell you whether your function has performed the expected actions. Finally, the test suite will also test to make sure that your program can handle several types of malformed data structures.

To get your program to run, you will need to include the provided function. However, you should submit your code without the extra function. You can combine your code to that of the necessary function by running the batch script provided on HuskyCT with your file name as a parameter (ex: P4Combine.bat "Program #4.asm"). The script will place the result in a file named output.asm that can be loaded into QTSPIM and run as normal. You will need to rerun this script every time you edit your program so that the

output file has your latest code. Alternatively, you can use an equivalent command to concatenate the code from each of the files and place it in an output file.
Unix: cat <Program #4 Function Name>.asm "Program #4 - Test Suite.asm" > <output>.asm

Your program should include appropriate comments indicating what the code should be doing and what registers are being used for. After displaying the results, your program should exit cleanly. Your programs should be turned in through HuskyCT before class starts on the due date. You should test your programs using the SPIM simulator to ensure their functionality before submitting them.

Expected output:

```
Test #1 passed.
Test #2 passed.
Test #3 passed.
Test #4 passed.
Test #5 passed.
Test #6 passed.
Test #7 passed.
Test #8 passed.
Test #9 passed.
Test #10 passed.
Test #11 passed.
----------Tests completed.----------
```

Objectives:

1. To practice calling functions in MIPS assembly language.
2. To introduce and practice working with bit flags and masks.
3. To introduce and practice building functions in MIPS assembly language.