Problem Set 3: Maze and Word Count

Please send back to me via NYU Classes

- A zip archive named as
  `PS03_<your name as Last_First>.zip`
  containing the C code files that implements all aspects of all problems.

Total points: 100

Problem 1
Solve Maze using Recursion
50 Points

You are given the files **maze.c, maze.h** and **maze.txt**. It is your task to create **maze_recursion.c** which contains function **solveMaze()** that recursively solves the maze.

The structure of **maze_recursion.c** should be

```
#include <stdbool.h>
#include "maze.h"

bool solveMaze(int i, int j) {

     (your code here)

}
```

You must write the function **solveMaze(i, j),** is such a way that it can call itself recursively to solve the maze.

The maze is file **maze.txt,** which is read into the global character array grid[i][j]. This is done in **maze.c**. You position is (i, j) where i (row) and j (column) are declared in **main()** and are the array indices in grid[i][j]. Variable i is the row index, or North/South, and variable j is the column index or East/West

To start (that is, on the first call to your function):

- The position (i, j) is at 'S' in the maze.

On each call to **solveMaze():**

- If the current maze grid position (i, j) is the END_MARKER ('G') then you found the goal and you are done, so return **true**;
- If the current maze grid position (i, j) is the VISITED_MARKER ('.') then you have already been at this position and do not want to back track, so return **false**.

- If the current maze grid position (i, j) is "illegal" then you cannot go to this position, so return **false**. Illegal positions are:
    - A Wall, or WALL_MARKER ('|')
    - Anywhere outside of the maze. This would be an invalid grid[i][j] array index (i.e. less than 0 or greater than DIM-1).

  If you haven't encountered any of the above conditions, then
    - Drop a breadcrumb, i.e. set the current maze grid position (i, j) character to VISITED_MARKER ('.') to indicate that you have visited this position.
    - Display maze grid by calling display()
    - Move one grid step in the N (or S, E, W) direction and call **solveMaze()** -- this is the recursion.
    - If any of the calls to **solveMaze()** in the previous step returns **true**, then you found the goal, and so:
        - Set grid character at the current position (i, j) to the SOLUTION_MARKER ('*'). This is the "backtrace" path that is the implicit solution provided by the recursion process.
        - Display maze grid by calling display()
        - Return **true**
- If any all the calls to solveMaze() in the previous step (N , S, E, W) are false, then return **false**.

It may be helpful to define and initialize the four points of the compass relative to current position. Note that the Maze grid has origin (0,0) at upper left corner (so N is -1, S is +1, etc.).

```
int Ni = i-1;     int Nj = j;
int Si = i+1;     int Sj = j;
int Ei = i;       int Ej = j-1;
int Wi = i;       int Wj = j+1;
```

Because of the "early exit" properties of an `if()` statement having a compound relational expression, the steps in each of the four points of the compass can be structured in this way:

```
if (solveMaze(North) || solveMaze(South) ||
        solveMaze(East) || solveMaze(West)) {

    (statements if true)

}
else {
    return false;
}
```

Problem 2
Count Words in a File
50 Points

Create a program **word_count.c** that counts the number of words in the supplied text file **words.txt**. Your program must have command-line arguments as follows:

```
./word_count input_file.txt
```

The beginning of your program will have the following lines:

```
#include <stdio.h>
#include <string.h>
#define LINE_LEN 80

int main(int argc, char *argv[])
{
    int count;
    char *ifile, line[LINE_LEN], *word;
    FILE *fp;
```

The first section of your program will:

- Declare variables, e.g.
- Print out usage instructions, e.g. if no input file is present in the command line.
- Parse command line args
- Open input file

The next section of your program will count the words in the file **words.txt**. You should use two nested `while()` loops to do this. The first loops over every line in the input file. The second loops over words in a line.

Use functions **fgets()** to read a line of text from the input file into the variable **line.** Continue to get the lines from the file until reading reaches the end of file, in which case **fgets()** returns NULL. The return value of **fgets()** can be the condition of the outer `while()` loop.

Use function **strtok()** to split **line** into words. Within the body of the outer while() loop, you will need to have two calls to strtok(), the first as

word = strtok(line, " \n")

which gets the first word in the line. If there are no words in the line i.e. word == NULL, continue on to the next line of text in the input file.

The second call to **strtok()** (also within the while() loop) gets the remaining words in the line. This must have the argument NULL

word = strtok(NULL, " \n")

This can be structured such that the value of word is the condition of the inner while() loop and the body of the while() loop just increments count.

Since **fgets()** includes the **'\n'** line termination character, use **" \n"** (i.e. space and newline) as the delimiter string in **strtok().**

At the end of your program, print the total word count as:

```
File <filename> contains <N> words.
```