Space: an empty spot in the puzzle where word should be placed
Spot: a character within a space
Global Variables: the array of times, the puzzleList array (array of empty spaces the words fit into), and the list of all intersections of spaces

Main algorithm:

Input: a folder containing puzzle text files
Output: Completed puzzles are printed, ones that are impossible are not, times for each puzzle's computation are displayed as well

```
Create an empty list of times (global)
FOR each puzzle text file in puzzle folder
    Start the timer and parse the file to find the length and width of the puzzle as well as the
    number of words
    Parse the file a second time and copy the format of the puzzle to a 2D list (grid)
    Use two double-for loops to first find horizontal spaces and then the vertical ones.
    These spaces are stored as lists of tuples, the tuples containing the X and Y coordinates
    on the puzzle board. Store these spaces in a list (puzzleList) (global).
    Use a double for loop to compare every space stored in the puzzleList and create a list of
    common tuples (intersecting spots) (global)
    Create an empty occupied spaces list
    Find the space with the most intersections in the puzzle
    Call the Solving Algorithm with the grid, the space with the most intersections, the list of
    intersecting spots, the list of words, and the empty occupied spaces list
    SET a result variable, the grid, and a garbage variable to the return value of the above
    function
    Stop the timer and add the time elapsed to the times list
    IF the result was successful, print the puzzle number and finished grid
    ELSE print that the puzzle was not possible with the puzzle number
FOR each time stored in times
    Print the puzzle number and the corresponding time elapsed
```

Solving Algorithm:

Input: 2D grid representing the game board, the current space being solved for, list of words, and spaces currently occupied on the board (initially empty)
Output: a tuple containing 1. The return value (tells the function whether to stop and try another word, continue, or finish because a solution has been found) 2. A grid and 3. A word list
Use: Call this function on any space in the puzzle and the completed puzzle will be sent back recursively as a grid

```
Find the list of words which could fit in the space length and letter wise
FOR each word in that word list
    Make deep copies of both the occupied spaces list and the grid
    Set the continue variable to 1 (0 means stop, 1 means go, 2 means done)
    Attempt to place the word into the space provided (Place Word Algorithm)
```

```
        IF unsuccessful, set the continue variable to 0 (which will send the program back to FOR)
        IF the continue variable is 1
            Add the input space to the cloned occupied spaces list
            Create a new word list containing all words but the newly added one
            FOR every space that intersects the current input space (Intersecting Spaces
            Algorithm)
                CALL this function with the cloned grid, current space in for loop, list of
                intersecting spots, the new word list, and cloned occupied spaces
                SET (continue variable, cloned grid, new word list) equal to the return
                value of the above function
                IF the continue variable is 0, break away from this for loop
                IF the continue variable is 2, return (2, cloned grid, new word list)
            IF continue variable is 1, return (1, cloned grid, new word list)
    Return (0, input grid, input word list) if the function made it out of the first FOR loop
```

Intersecting Spaces Algorithm:

Input: a space, list of occupied spaces
Output: a list containing the spaces that intersect the input space

```
    Create an empty return list
    FOR each space* in puzzleList
        IF the space* is not the input space and the space* is not in the occupied spaces list
            FOR each spot in the list of common spots
                IF the spot exists in the input space and in the space*
                    Add the space* to the return list
    Return the return list
```

Find Best Space Algorithm:

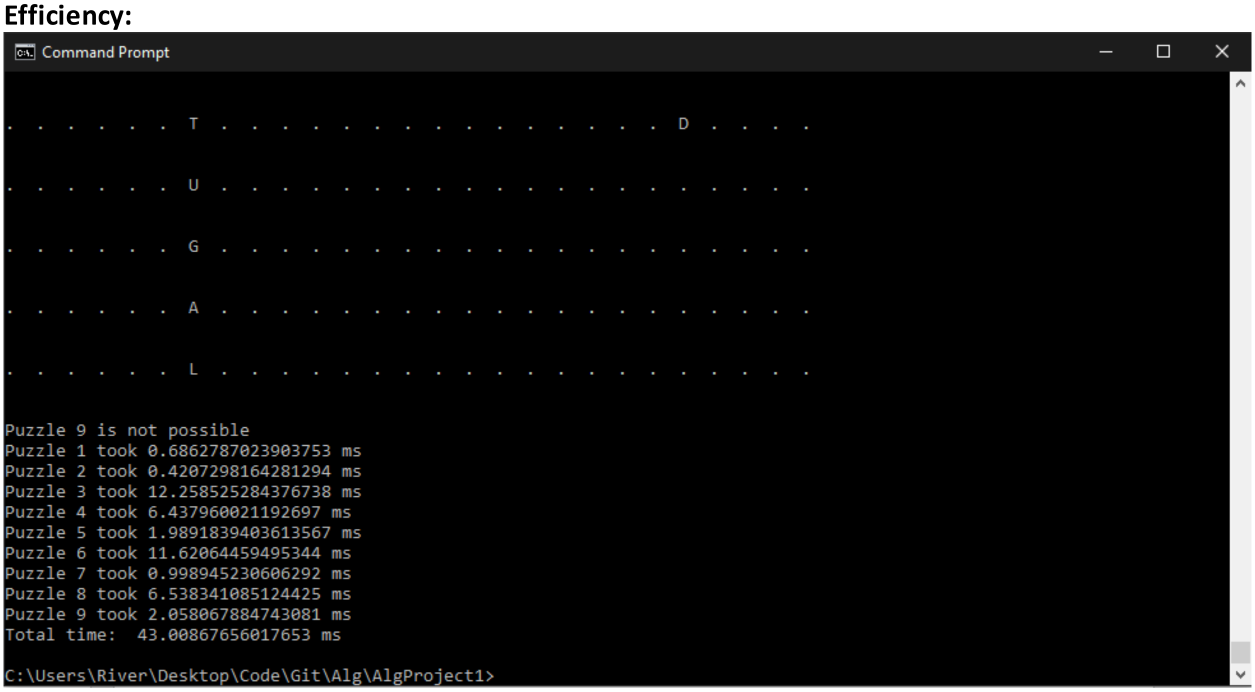
Input: none
Output: the index of the space with the most intersections

```
    Initialize array of numbers of intersections
    FOR each space in the puzzle list
        Append a zero to the array of numbers
        FOR each spot in the space
            IF the spot is in the list of common spots
                Increment the number of intersections at that index
    Find the max value in the list
    RETURN the index of the max value
```

Possible Words Algorithm:
Input: a grid, a space, and a wordlist
Output: a list of words that can fit in the space given its current state

FOR each spot in the space
    If the corresponding spot does not contain a "*"
        Add the letter to an array
        Add the index to an array
FOR each word in the input word list
    Initialize test bool to true
    IF the length of the word is the same as the length of the space
        FOR each letter in the letters array
            IF the word does not have the same letter at the same index
                Test bool is set to false
                Break
        IF the test bool is true
            Add the word to the return list
Return the return list

Efficiency:



Across the board beating the better benchmark times, and absolutely crushing the large puzzle solve times, especially on 6 and 9. I've spent quite a bit of time revising this so I'm proud of it.

The main solve algorithm iterates through a for loop N times, attempting to assign length of N characters to a grid, and calling itself again with N-1 for the input size. However, this doesn't necessarily mean the algorithm is any closer to being solved since it can still go back recursively and try something else. This behavior means that it can technically, if the perfect storm arises and none of the input optimization was used, the algorithm has an efficiency of a brute-force attempt of trying every single combination of words. This behavior gives it a theta(N!) time efficiency on its own. In a best case scenario, you can input N and every single word will work the first time you try it. In this case, the time efficiency would be closer to theta(N).

Officially, the main operation is the calling of the recursive function within itself. This can occur a minimum of N times and a maximum of N! times. I believe the proper definition in that case is that it is theta(N!)