**QUESTIONS:**

1. While building the maze, we attempted moving 2 cells at a time.
   1. What would the maze look like when moving a larger number of cells?
2. The maze would have wider corridors of empty cells in it.
   1. What would the maze look like if this number was not constant?
3. The maze would have corridors of empty cells that goes wider and narrower randomly.
4. What algorithms could you use to find a path through this maze? Compare and contrast at least 2.
5. There are two common approaches to using a graph search to progress through a graph:

* depth-first search, known as DFS follows each possible path to its end
* breadth-first search, known as BFS broadens its search from the point of origin to an ever-expanding circle of neighboring vertices

DFS, which employs either recursion or a stack data structure, is useful for determining whether a path exists between two points.

BFS, which generally relies on a queue data structure, is helpful in finding the shortest path between two points.

1. How does knowing the algorithm used to generate the maze influence the best algorithm to solve it with?

A. As it is known that maze was build to be solved by max amount of time and used certain algorithm, probably another algorithm should be used.

1. As a patron picking up swag along the way, how might you best store the list of items you’ve collected?

A. It depends on what I may want to do with these items later. If I'd like to simply collect it, a list would be enough. I'd add them simillary to how I added coordinates that I passed.

1. If the farmer asked you to sort the items you collected before leaving the maze, what sorting algorithms would you consider using (assume a much larger list of possible swag)?

A. I'd use Quicksort. Quicksort is an efficient algorithm for sorting values in a list. A single element, the pivot, is chosen from the list. All the remaining values are partitioned into two sub-lists containing the values smaller than and greater than the pivot element. It is very fast and efficient algorithm.

1. How does the quantity and variety of swag influence your answer?

A. Low quantity and/or of swag can be sorted with less efficient algorithm.

**CODE:**

from random import randint

#function that summs all methods to build a maze

def build\_maze(m, n, swag):

grid = [["wall" for col in range(n)] for row in range (m)]

start\_i, start\_j = randint(0, m-1), randint(0, n-1)

grid[start\_i][start\_j] = 'start'

mow(grid, start\_i, start\_j)

explore\_maze(grid, start\_i, start\_j, swag)

return grid

#function that print maze

def print\_maze(grid):

for row in grid:

printable\_row = ''

for cell in row:

if cell == 'wall':

char = '|'

elif cell == 'start':

char = 'X'

elif cell == 'empty':

char = ' '

else:

char = cell[0]

printable\_row += char

print(printable\_row)

#function that moves in 4 dimentions depending on conditions

def mow(grid, i, j):

directions = ['U', 'D', 'L', 'R']

while len(directions) > 0:

directions\_index = randint(0, len(directions) - 1)

direction = directions.pop(directions\_index)

if direction == 'U':

if i - 2 < 0:

continue

elif grid[i-2][j] == 'wall':

grid[i-2][j] = 'empty'

grid[i-1][j] = 'empty'

mow(grid, i-2, j)

elif direction == 'D':

if i + 2 >= len(grid):

continue

elif grid[i+2][j] == 'wall':

grid[i+2][j] = 'empty'

grid[i+1][j] = 'empty'

mow(grid, i+2, j)

elif direction == 'L':

if j - 2 < 0:

continue

elif grid[i][j-2] == 'wall':

grid[i][j-2] = 'empty'

grid[i][j-1] = 'empty'

mow(grid, i, j-2)

else:

if j + 2 >= len(grid[0]):

continue

elif grid[i][j+2] == 'wall':

grid[i][j+2] = 'empty'

grid[i][j+1] = 'empty'

mow(grid, i, j+2)

#function to explore maze

def explore\_maze(grid, start\_i, start\_j, swag):

grid\_copy = [row[:] for row in grid]

bfs\_queue = [[start\_i, start\_j]]

directions = ['U', 'D', 'L', 'R']

while bfs\_queue:

i, j = bfs\_queue.pop(0)

if grid[i][j] != 'start' and randint(1, 10) == 1:

grid[i][j] = swag[randint(0, len(swag)-1)]

grid\_copy[i][j] = 'visited'

for direction in directions:

explore\_i, explore\_j = i, j

if direction == 'U':

explore\_i = i - 1

elif direction == 'D':

explore\_i = i + 1

elif direction == 'L':

explore\_j = j - 1

else:

explore\_j = j + 1

if explore\_i < 0 or explore\_j < 0 or explore\_i >= len(grid) or explore\_j >= len(grid[0]):

continue

elif grid[explore\_i][explore\_j] == 'wall' and grid\_copy[explore\_i][explore\_j] != 'vivited':

bfs\_queue.append([explore\_i, explore\_j])

grid[i][j] = 'end'

#example for testing

print\_maze(build\_maze(15,25,['candy corn', 'werewolf', 'pumpkin']))