159.271 Computational Thinking for Problem Solving

Tutorial 2: Randomly Generating a Maze

In this tutorial you will develop a recursive algorithm to generate a random maze. The maze will be represented by an **`n x m`** two dimensional array, with entries from **`[brick, floor, cheese]`**. Walls consist of lines of bricks. A mouse will be able to move along floor squares in any direction. The maze generated will not contain corridors as such, but only many small rectangular rooms. Each room will have either one door in one corner of the room or two doors in opposite corners. The cheese will be placed in a room that is chosen randomly from among the rooms that are far enough from the start location.

Precondition: The routine **`AddWalls`** is passed a two dimensional array (list of lists in Python) representing the maze as constructed so far, and the coordinates of a room within it. The room will have a surrounding wall except for one door in one of its corners. The room will be empty of walls. The routine is also passed a flag indicating whether or not cheese should be added somewhere in the room.

Postcondition: The output is the same with a randomly chosen submaze added within the indicated room and cheese added as appropriate.

Initial conditions: To meet the preconditions of **`AddWalls`**, the main routine first constructs the four outer walls with the top left corner square left as a door into the maze and as the start square for the mouse. Calling **`AddWalls`** on this single room completes the maze.

Subinstances: If the indicated room has height and width of at least 5, then the routine **`AddWalls`** will choose a single location **`(i,j)`** uniformly at random from among all those in the room that are not right next to one of its outer walls. Use **`x = random.randrange(lower,upper)`** to generate a random number in the (inclusive) range **`[lower, upper-1]`**. A wall is added within the room all the way across row **`i`** and another is added all the way down column **`j`**, subdividing the room into four smaller rooms. To act as a door connecting these four rooms, a square centered at location **`(i,j)`** will remain as "floor". Then four subinstances (friends) are called to fill a maze in each of these four smaller rooms. If our room is to have cheese, then one of the three rooms not containing the door to our original room is randomly selected to contain the cheese.

Base cases: What should the routine **`AddWalls`** do in the case that the indicated room doesn't have height and width of at least 5? When is it appropriate to put the cheese in place?

Your job is to write the Python code for 1. generating a maze of rooms, and 2. displaying such a maze, using the Pygame template provided on Stream.

The easiest way to display the maze is to work through the two dimensional array in which it is stored and to draw an appropriate sized and colored rectangle for each cell in the array, at the appropriate location on the screen. You will need to ensure that the "doors" are big enough, perhaps by setting all neighbors of a cell designated a door to also remain as floor

tiles.

You may find the following Pygame drawing functions useful:

- **pygame.draw.rect**

  `pygame.draw.rect(Surface, color, [left,top,width,height], width=0): return Rect`
  Draws a rectangular shape on the Surface. The list argument [left,top,width,height] is made up of four integers, specifying the left and top alignments and the width and height of the rectangle. The last width argument is the thickness to draw the outer edge. If this width is zero then the rectangle will be filled.

- **pygame.draw.line**

  `pygame.draw.line(Surface, color, start_pos, end_pos, width=1): return Rect`
  Draws a straight line segment on a Surface. The start_pos and end_pos arguments are pairs of integers (x1, y1), (x2, y2). The width argument is the thickness to draw the line. The ends are squared off for thick lines.

- **pygame.draw.ellipse**

  `pygame.draw.ellipse(Surface, color, [left,top,width,height], width=0): return Rect`
  Draws an elliptical shape on the Surface. The list argument defines the rectangle is the area that the ellipse will fill. The width argument is the thickness to draw the outer edge. If width is zero then the ellipse will be filled.

Example usage:

```
def drawRect((i,j), screen):
    pygame.draw.rect(screen, white, [i, j, 40, 40])
```

This routine will draw a white square on the surface "screen", 40 pixels wide with top left corner at position (i,j)

```
def drawLine(left, i, top, j, screen):
            pygame.draw.line(screen, black, (left, i), (top,j), 5)
```

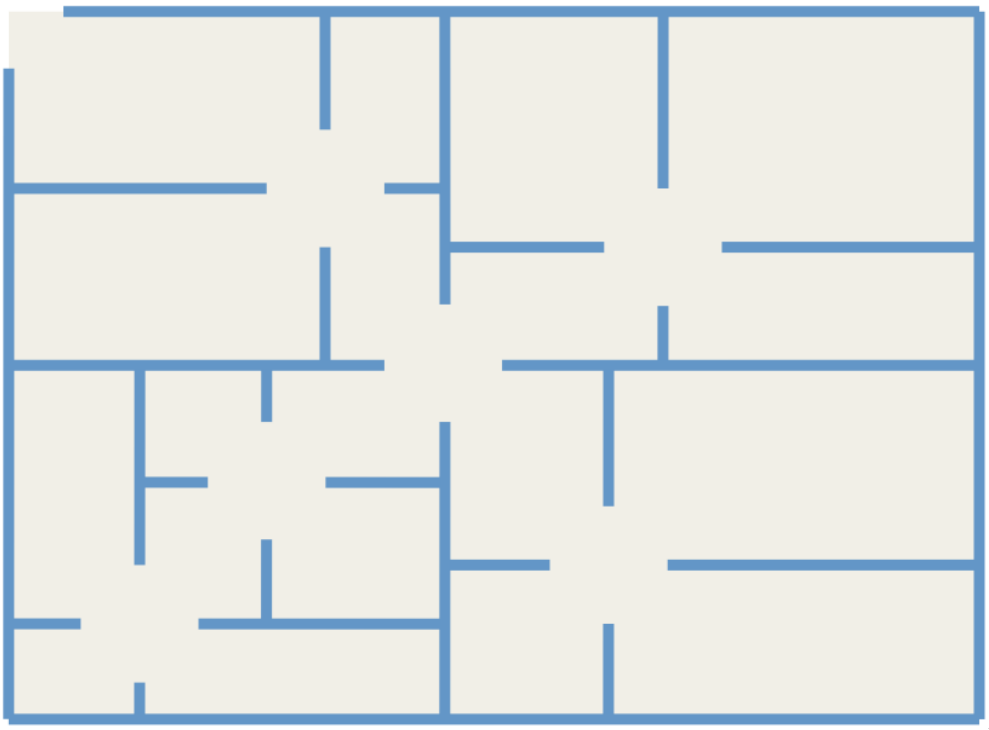
This routine will draw a black line on the surface "screen", 5 pixels wide from point (left, i) to point (top,j).

Code for creating and manipulating a surface is shown here:

```
# Set the height and width of the screen surface
size=[500,500]
screen=pygame.display.set_mode(size)
```

```
# Set a background colour
screen.fill(white)

#Set the colour of a particular pixel
screen.set_at((i,j), red)
```


An Example Maze

Searching the maze: One way of representing the maze is to use a graph, where each room is represented as a vertex and two vertices are connected by an edge if the rooms that they represent share a door. Think about how you might represent and generate such a graph and how you would go about searching it.