9.2 - Week 9 - Applied - Practical

Introduction

Objectives

- Become better acquantied with uses for Stacks, Queues and Iterators.
- Be able to compare and contrast their utility and complexity.

Iterating Iterables

Implement the following Iterators:

- `second(iter)`: returns an iterator that yields every second value from `iter`.
- `ignore_first(k, iter)`: returns an iterator that ignores the first `k` values in `iter`, and yields everything after that.
- `limit_iteration(k, iter)`: returns an iterator that yields the first `k` values of `iter`, but stops after this.

For example we can evaluate the following:

- `ignore_first(1, limit_iteration(4, second(iter([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]))))`
- `=ignore_first(1, limit_iteration(4, iter([2, 4, 6, 8, 10])))`
- `=ignore_first(1, iter([2, 4, 6, 8]))`
- `=iter([4, 6, 8])`

so `[x for x in ignore_first(1, limit_iteration(4, second(iter(range(1, 10)))))] == [4, 6, 8]` .

After you've done this, try to come up with a combination of these modifications to transform `range(1, 61)` into `iter([19, 27, 35])`.

Mystery Lists again!?😱

You knew it was coming... 😔

Below is an operation `mystery` of the LinkedList ADT, which takes in another instance of LinkedList:

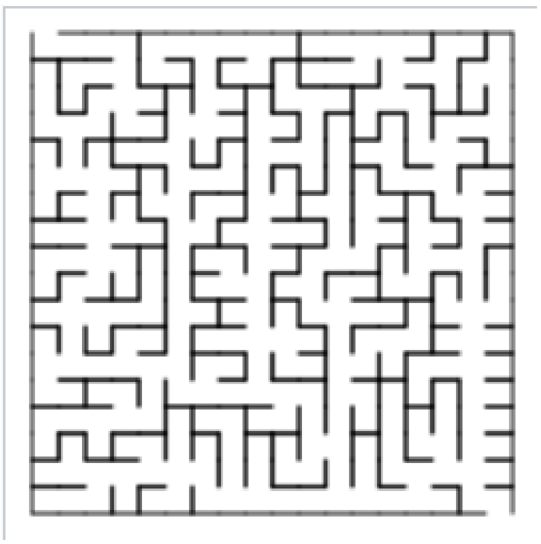
```
class LinkedList(ListADT):

    def mystery(self, other: LinkedList) -> None:
        p1 = self.head
        while p1 is not None:
            p2 = other.head
            while p2 is not None:
                if p2.item == 0:
                    break
                p1.item *= p2.item
                p2 = p2.link
            p1 = p1.link
```

1. Analyse the best and worst case time complexity of this algorithm
2. Think of a faster way to implement this algorithm
3. Implement it and analyse the best and worst case time complexity.

Maze

Over the next few tasks we'll use a stack to generate *and solve* random mazes, like the one below:



Maze Generation with Stacks & Queues

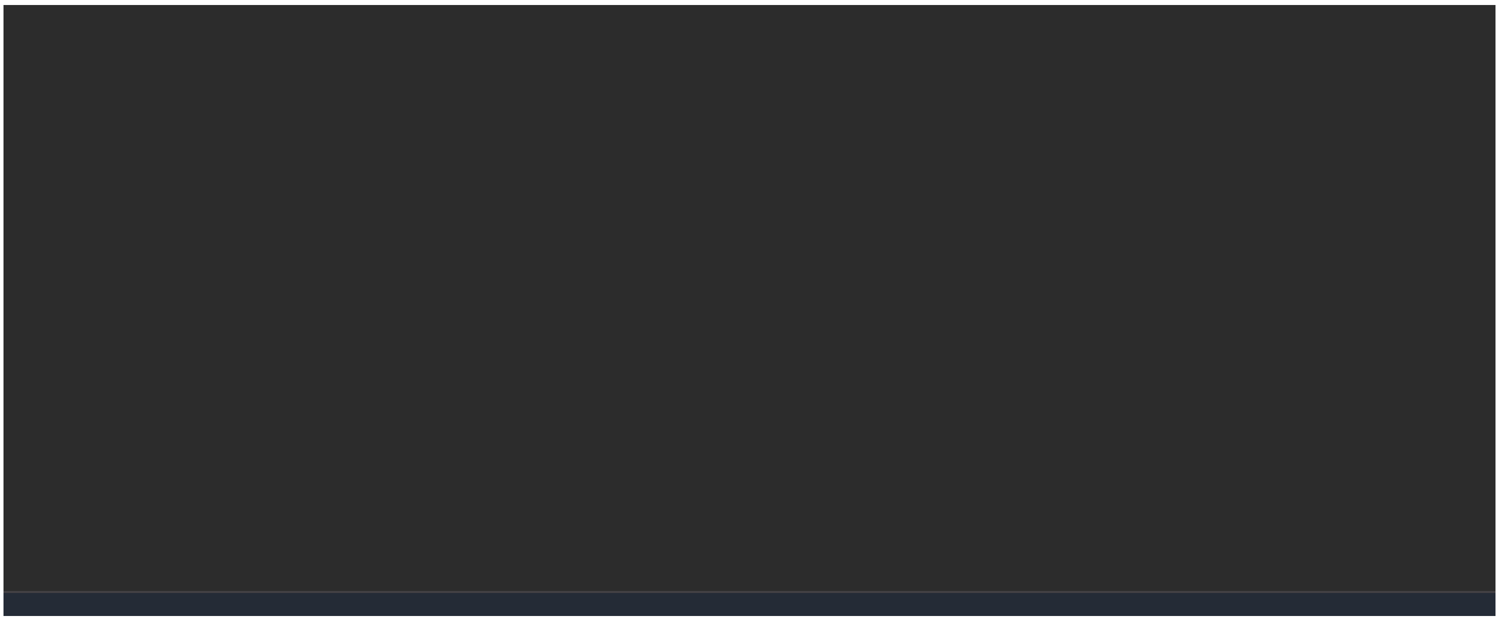
The most basic maze generation algorithm goes as follows:

```
* Initialise the stack to include only the position of the start node.
* Set the `visited` for the start node to True.
* While the stack is not empty:
    * Pop a position off of the stack, let's call the node at this position current
    * Find all unvisited and reachable neighbours
    * If one exists:
        * Choose a random unvisited and reachable neighbour, call it next_node
        * Add the position of current back onto the stack
        * Set the `visited` for the next_node to True.
        * `connect` the position of current to the position of next_node
        * Add the position of next_node onto the stack.
```

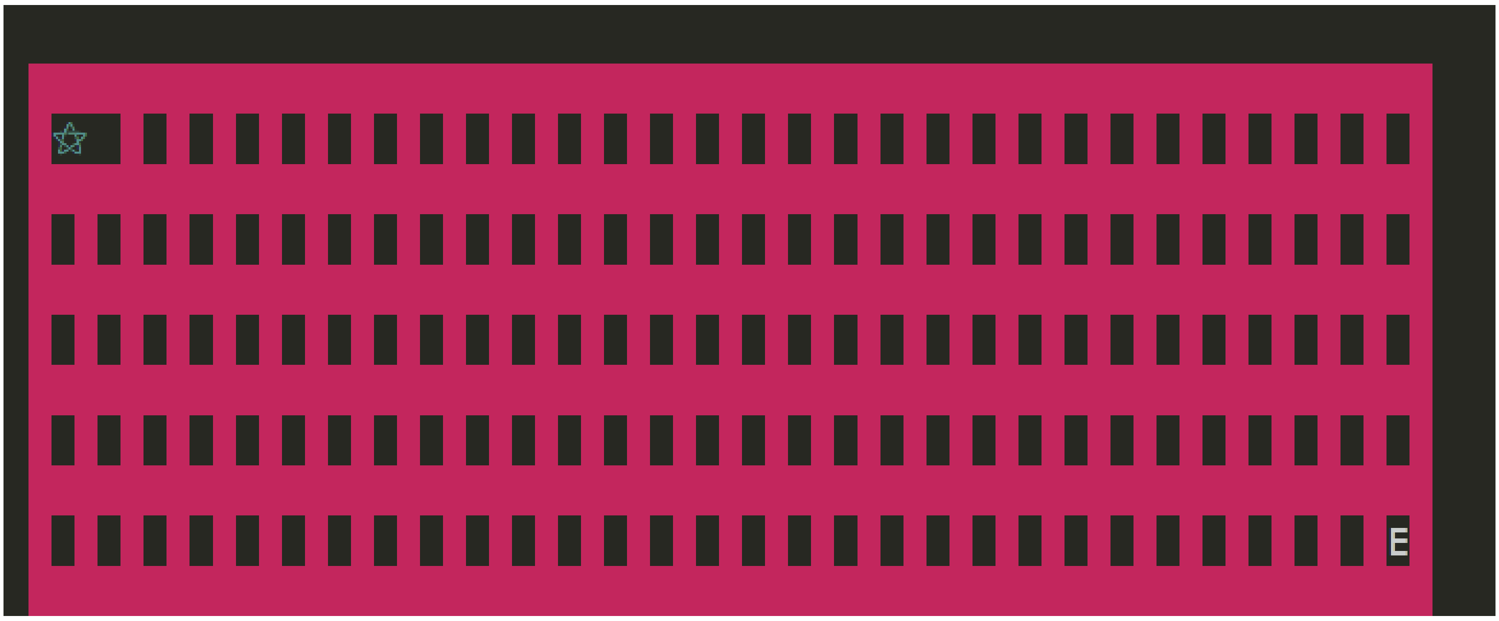
So we just repeatedly connect neighbours until we get a maze.

Task 1:

Familiarize yourself with the code given in `maze.py`, and then try to implement `make_maze`, given the comments above. Provided you are calling `connect` correctly in this function, running `python maze.py` will then (badly, lots of flashing) animate the creation of the maze:



This will likely look better on your local machine, and not the ed terminal:



Task 2:

Try to understand *why* this algorithm works, and what sort of mazes it creates. Think about what the maze would look like if we swapped out the stack for a queue, and then do this, and confirm/deny your hypothesis.

Maze Solving with Stacks & Queues

Now that we've built a maze, surely we want to solve it too!

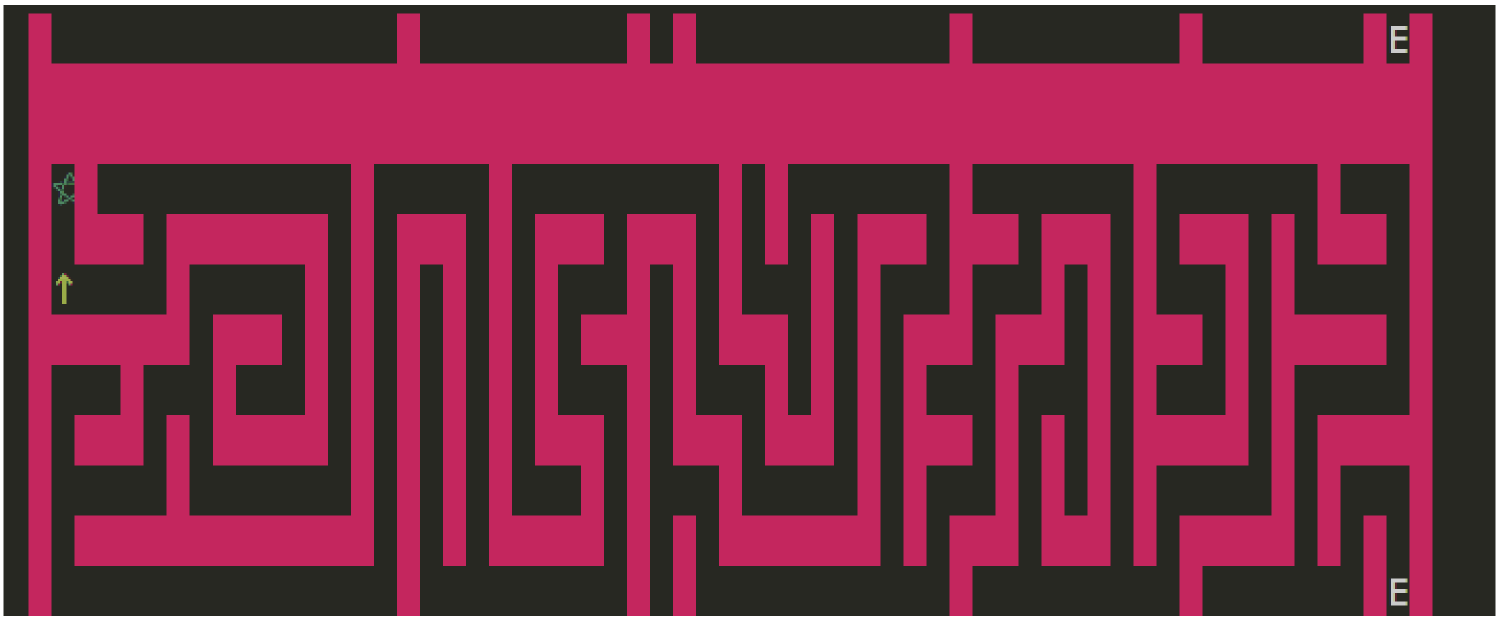
Again, stacks and queues can come to the rescue. Here, rather than breaking down walls to make the maze, our search will simply keep track of how to get back to the start node.

The pseudocode is as follows:

```
* Initialise the queue, containing the start_node
* While the end is not found:
    * serve a current node from the queue
    * for each direction, if current has an unmet neighbour then:
        * get the node in that direction, call it new_node.
        * Set `new_node.prev = current`
        * Set `new_node.visited = True`
        * Append new_node to the queue.
    * Print the maze and call time.sleep(WAIT_TIME) for the animated version.
```

Once we've done this for the end node, then we can simply reverse the path to get a path from start to finish. Use this information to mark every node on the path to the end as `end_path = True`.

After implementing all of this, you should get:



Try replacing the stack with a queue. What do you think will happen?