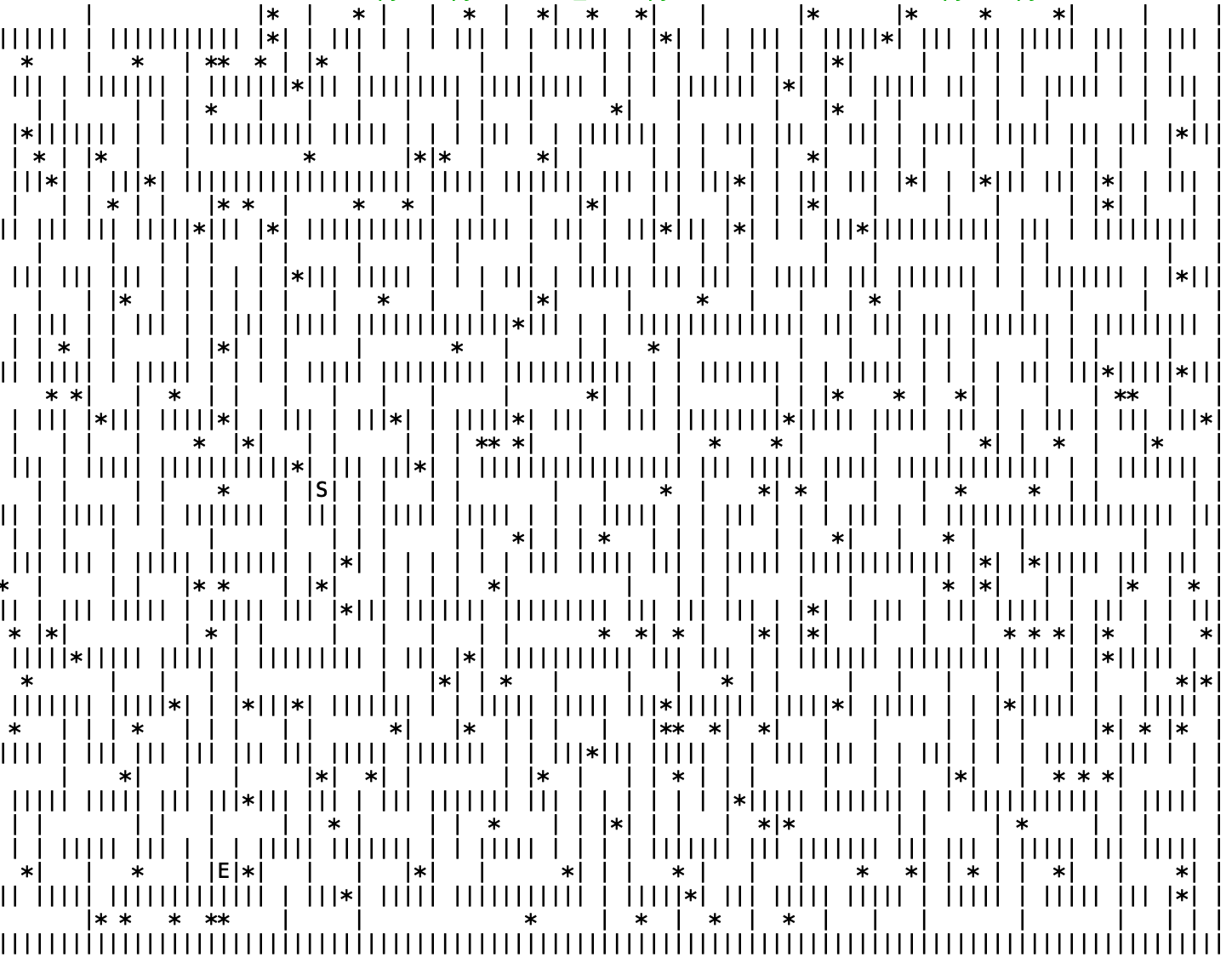
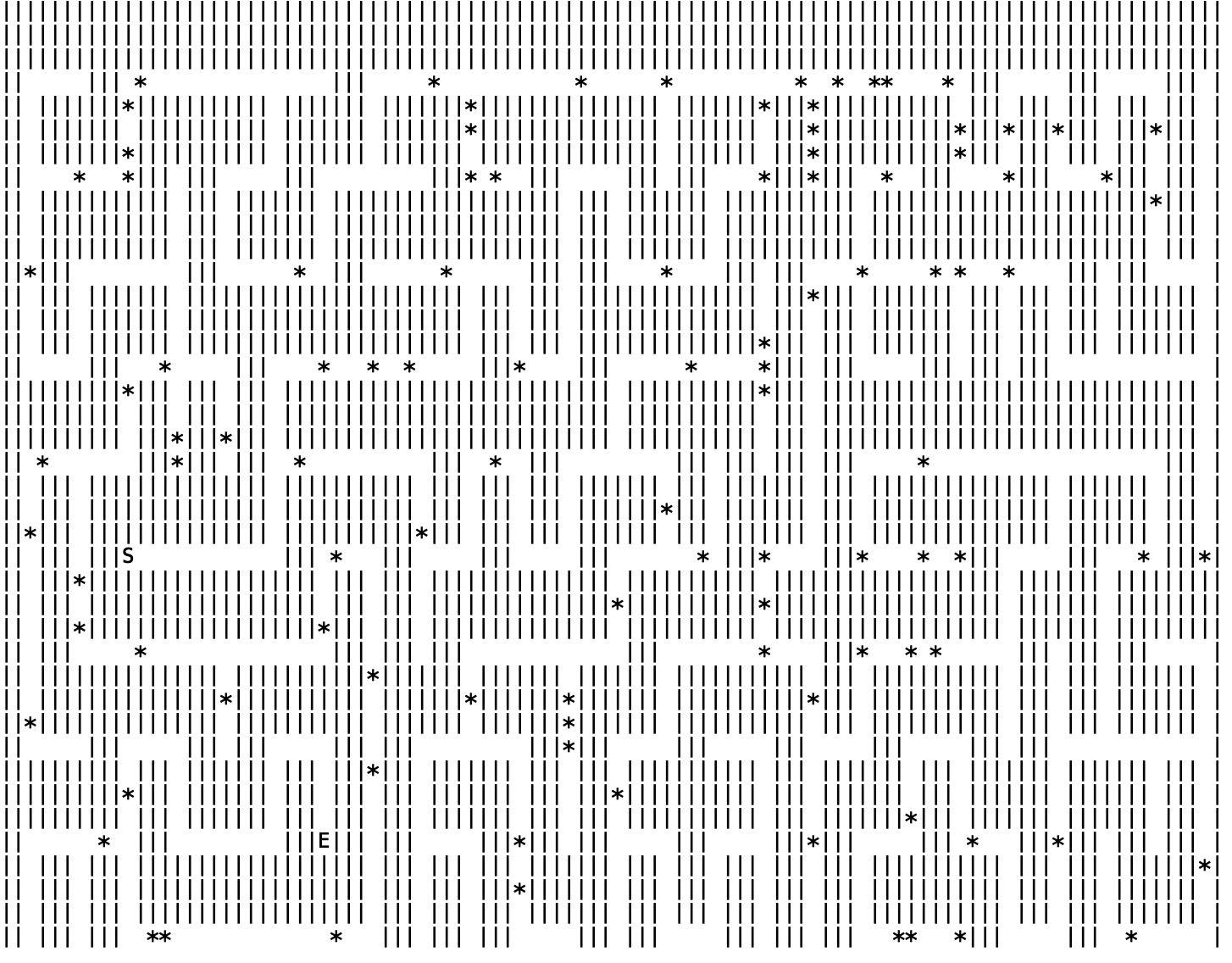
*1) While building the maze, we attempted moving 2 cells at a time.*

*a) What would the maze look like when moving a larger number of cells?*

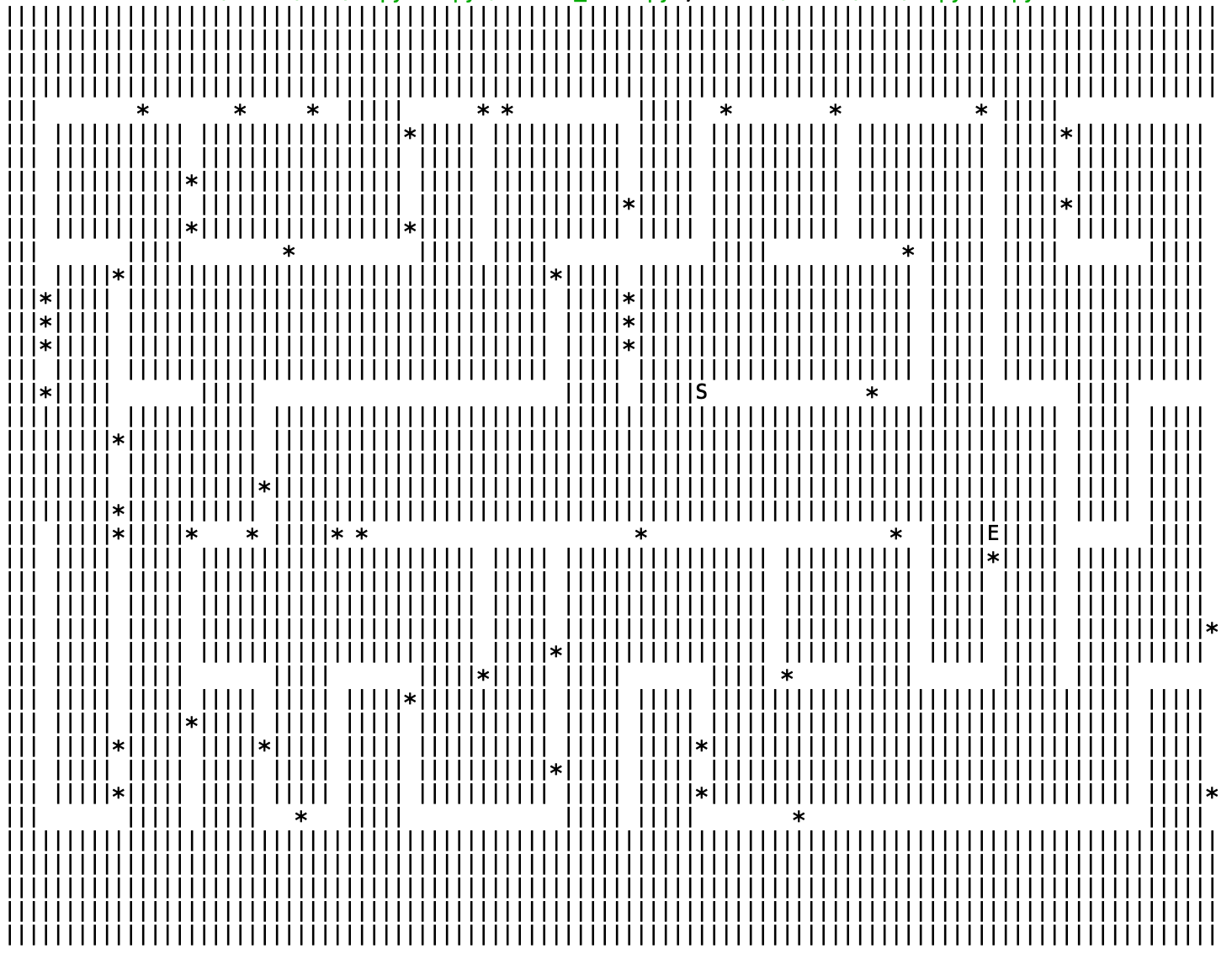
The maze ‘walls’ become increasingly thicker. I guess this is to be expected because we’re jumping more cells at a time. Because the walls become thicker, the number of possible paths also becomes fewer.



Maze with 2 cells at a time



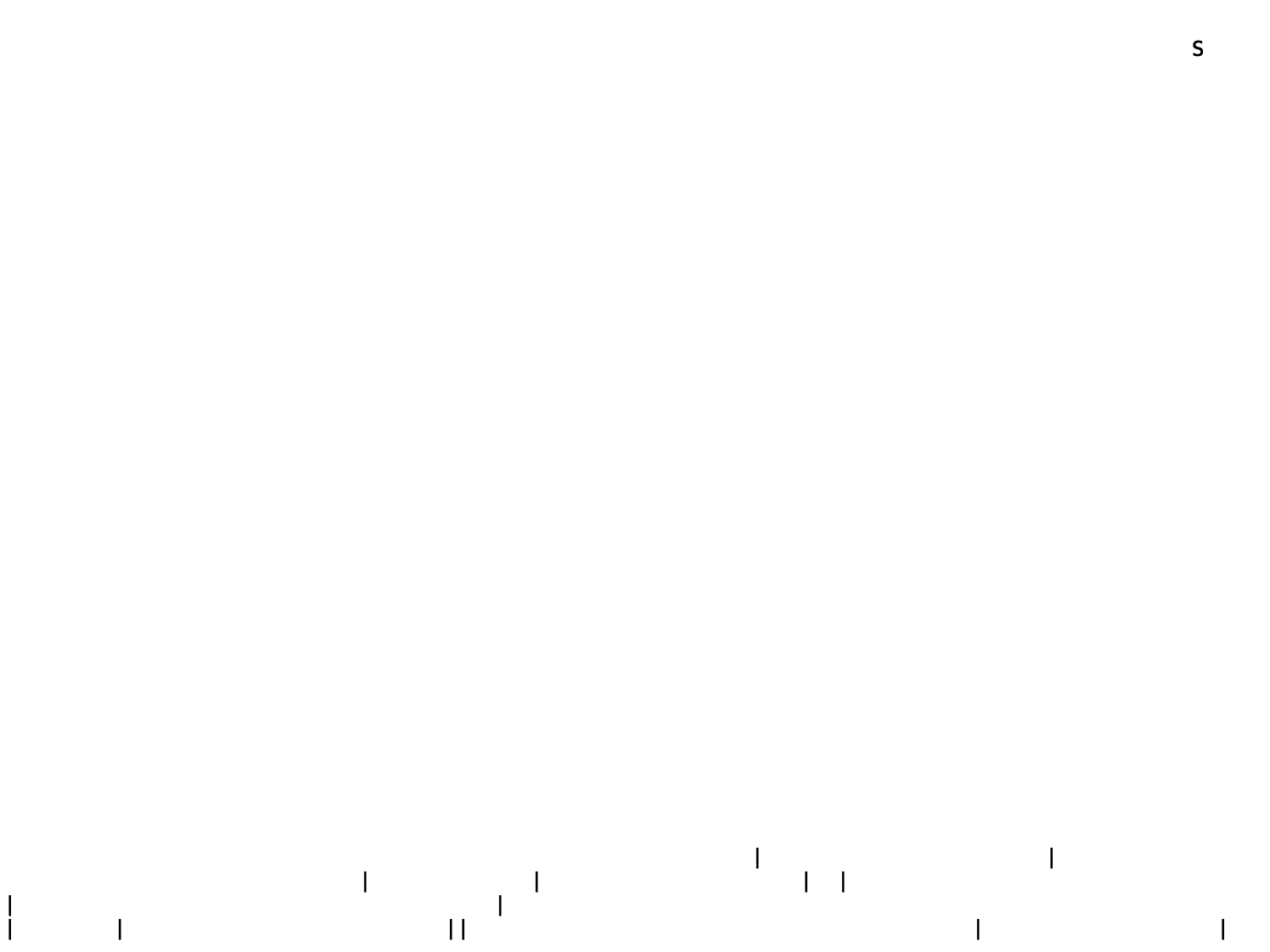
Maze with 4 cells at a time



Maze with 6 cells at a time

*b) What would the maze look like if this number was not constant?*

The maze looks very empty, such as below.



Maze with steps = randint(2,6)

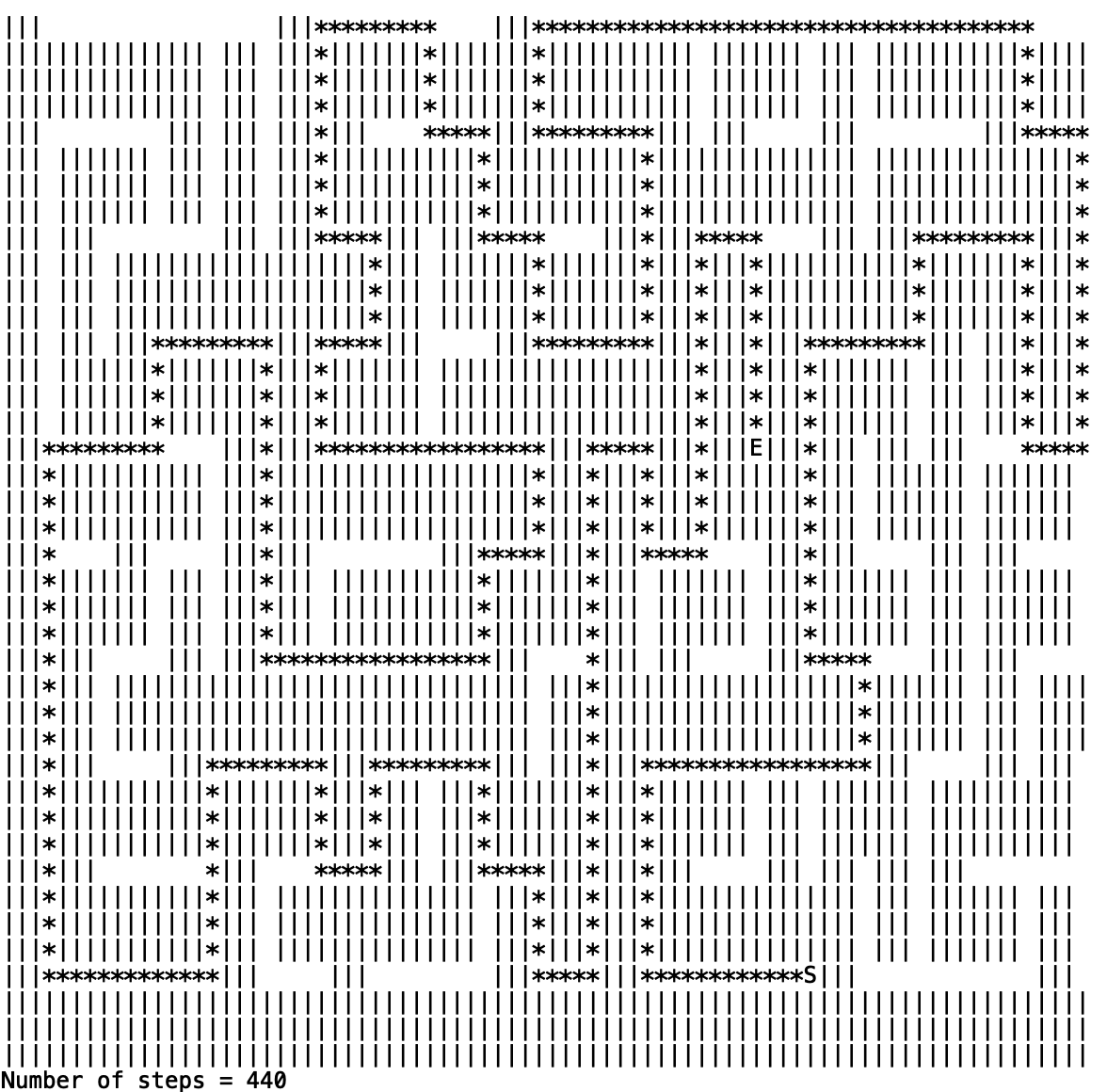
When there are different number of steps, I think that the odds of finding a ‘wall’ become much greater. In the case of constant steps, there is a ‘minimum’ wall thickness (e.g. for step = 6 above, the thickness is 5 lines). When we randomize the number of steps, this minimum wall thickness no longer applies as we may randomly ‘check’ for a point within the wall itself and mow it down. Hence, mowing occurs more frequently and the maze becomes really sparse.

*2) What algorithms could you use to find a path through this maze? Compare and contrast at least 2.*

We can use BFS. This will be highly useful if the maze has more than one solution, and we are interested in finding the fastest way out, i.e. the path with the least distance.

We can also use DFS, if we are only interested in finding any solution regardless of how long it takes. This is likely to have a faster runtime since we need not consider every possible path.

The A\* algorithm may work in this case, provided that we are given the coordinates of the end point at the beginning. However, the heuristic’s effectiveness is likely to be dampened because of the nature of mazes (we likely need to travel *away* from the end point in order to reach it). In the solution below, notice that we get really close to the endpoint ‘E’, but we are forced to take a really long route to actually reach E.



I tried to implement an algorithm to search for a path (uploaded along with the original task, solution seen above): It seems to be a mix of BFS and DFS: I recreated the grid, but with each coordinate (X,Y) storing a list that contains the path (i.e. coordinates that we need to follow) from start to (X,Y). I explored every adjacent cell until the ‘end’ cell is reached (this is the part that looks like BFS). The contents of the list at the coordinates of the ‘end’ cell is then the solution of the maze. However, I did not keep track of the distance travelled as I only wanted a solution (this is the DFS part). Not sure if this is the best way to approach it though. Just thought it’d be fun to try it out!

*3) How does knowing the algorithm used to generate the maze influence the best algorithm to solve it with?*

With question 2 in mind, knowing the nature of the maze paths can be useful in determining the most suitable algorithm.

For instance, if we know that the algorithm skips a lot of cells for every step in the *mow* function, we can infer that there are fewer branches in the paths. Hence, using a DFS algorithm can give us a reasonably good solution in a short amount of time. If the opposite is true, i.e. there are many branches (such as the case of mowing every 2 cells), then BFS might be a better algorithm to obtain the shortest path.

Another example is if we know that the mow function only moves in a restricted manner, e.g. it only mows upwards and rightwards. If this is the case, then heuristic algorithms such as A\* are more effective because the heuristic value obtained from the Euclidean distance is reasonably accurate.

*4) As a patron picking up swag along the way, how might you best store the list of items you've collected?*

I think that we can store the collection of items as either a dictionary or tuples in a set. The reason is that these two data structures allow us to keep track of the item *count*, i.e. how many of them have been collected (through key-value pairs or tuple pairs). I think that these methods are great if the *order* of items is not important.

If the order of items is important, I think that we can store the items as tuples in a list. This preserves the item count while having the items sorted. The sorting algorithms are discussed in the next question.

*5) 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)?*

If we just need a general sorting algorithm and we do not know the nature or properties of the items, I think that merge sort would be a good idea, as it has a better average case runtime of O (n log n) compared to quick and bubble sort (n2). The relative ‘values’ of the items can be compared using their ASCII values.

*6) How does the quantity and variety of swag influence your answer?*

We can decide the most suitable algorithm based on the time and space complexities that arise from the items.

For example, if the quantity of items is really huge, then merge sort would not be a good idea as it involved splitting the list in half each round. This means that we need to keep creating temporary lists which will take up a lot of memory. Bubble or quicksort would be a better idea since they are done in-place, i.e. just modifying the same list over and over until it is sorted.

Another case to consider is the nature of the list itself: whether it is mostly sorted or not at the start (this will depend on the algorithm that generates the items). If the list is mostly sorted, or has only a small variety, then merge sort is not a very good idea again (no point splitting the list in half if we know that the entire half has already been sorted, for example). Bubble sort will be a better choice in this place since it just runs through the list with few swaps and does not require lots of memory.

If however, there are only a few items but a lot of variety (e.g. only one of each kind), then merge sort would be ideal since the list is unlikely to be sorted. This is where the splitting of the list into halves becomes effective.