COSC 1P03 – LAB 6 – Recursion

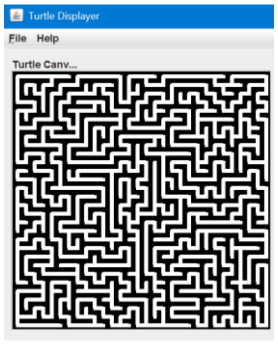
This one's just two (unrelated) tasks. It *can* actually be pretty simple, but considering how incredibly important of a topic recursion is, you're *strongly* encouraged to take your time with this.

Where did you come from, where did you go?

A classic recursion problem is *maze-solving*. We mentioned it in class, because it's a pretty neat algorithm: starting at some location, try to find the exit by treating it as 'a single step, plus the subproblem of getting to the exit from the new location'. If such a subproblem can be solved, then your current location is *also* part of the solution. Otherwise, it isn't. Similarly, if the starting location is not part of a path to the exit, then the entire maze *must* be unsolvable!

It's pretty neat! It's also pretty standard. So let's turn it around and try something adjacent?

How could we *create* the maze?

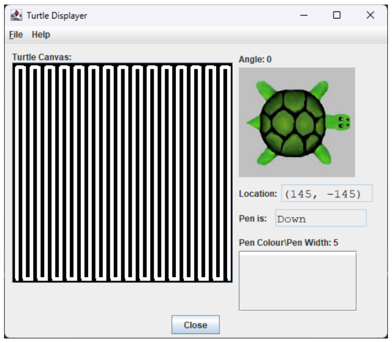


Mazes come in all shapes, sizes, and forms. They have different rules, etc. e.g. some mazes allow for *loops*, while others don't. Some have you start/exit outside of the maze, while others simply go from one location to another. Some try to maximize the amount of traversable space.

Let's keep things simple: we don't want an actual maze; just 'something maze-y'. Specifically:

- We want to divide some space into a 'grid' — let's just fix that as always 30 by 30
- We want all 900 locations to be reached
- We want to connect those cells using cardinal directions, *without* loops

Technically we could satisfy these requirements with this:



Except… no. Just no.
So:

- We'll start with 'all walls' (a black background), and effectively *carve out* the traversable paths
- We'll randomly select some starting location
- From any given location, we'll see if we can go North, South, East, *and* West, but in a random sequence

    - Since we'll sometimes be going ↑→↓←, sometimes ↓←→↑, etc. we'll be taking arbitrary paths from each location, increasing the appearance of randomness

But wait! How do we know we'll get *everywhere*? Because we'll be mathematically guaranteed to:

- At *every* location, we'll go as far as we can from that point, until we hit an already visited location
    - Which will be repeated for each remaining direction
    - Since we only stop when we reach somewhere we've already been, and we do this for every location we ever get to, visiting everywhere is the only way to ever stop

So, what do we do?

First we start with some boilerplate code and a couple convenience functions.

- We're going to return to that old chestnut, Turtle Graphics. So make a class and `import` what you need
- To keep headers simpler, just put your `Turtle` as an instance variable
- We'll need a 2D boolean array to keep track of where we haven't been, so make a 30×30, also as instance
    - We don't need to worry about assigning all 900 `false`s; they're already defined by default

We want a black background, and to be ready for 'carving out' spaces (meaning drawing in white), so:

```
private void blank() {
    pen.setPenWidth(400);
    pen.penDown();
    pen.moveTo(-100,100);
    pen.moveTo(100,100);
    pen.moveTo(100,-100);
    pen.moveTo(-100,-100);
    pen.moveTo(-100,100);
    pen.penUp();
    pen.setPenWidth(5);
    pen.setPenColor(java.awt.Color.white);
}
```

Also, we'll want to deal with indices, not coordinates, so let's make a separate method for drawing:

```
private void travel(int x, int y) {
    pen.moveTo(x*10-145,y*10-145);
}
```

The easiest way to repeatedly pick random directions (but still picking all options eventually) is to use a randomized sequence of four numbers. Doesn't matter *what* those numbers are, so long as we can map each to a different direction, so we might as well go with 78, 83, 69, and 87 (or `N`, `S`, `E`, and `W`) the ASCII mapping:

```
private char[] random() {
    char[] directions={'N','S','E','W'};
    for (int i=0;i<4;i++) {
        int other=(int)(Math.random()*4);
        char swp=directions[i];
        directions[i]=directions[other];
        directions[other]=swp;
    }
    return directions;
}
```

It's clumsy, but it's sufficient for our needs.
So all we need is, like, an algorithm!

Here's the idea:

1. You start at *some* location. So long as you're in a *new* location, you're good!
   (If we've already `visited` this location, we give up on this branch of the path immediately)
2. We mark that we *have* now `visited` this location (if we return here via some other path, we'll know to leave)
3. We tell the Turtle to follow us here to this location
   (We invoke the `travel` method to the current coordinates)
4. We get our randomized sequence to know which directions to try first from this location
5. For each direction within the sequence:
    a. We obviously identify which case it is
    b. Assuming that direction wouldn't put us outside the maze's bounds, we try recursing there!
    c. No matter which way we went, we *need* the Turtle to return back to 'here'!
6. …that's it. That's the entire algorithm

So do it!

Counting is hard!

The last task was interesting, in that we repeatedly returned to previous locations, but it does *not* count as a 'backtracking' algorithm. The reasoning is in the *intent* of the term.

Within the context of recursion, when we refer to backtracking, we're talking about making multiple possible attempts to achieve the same goal. (If you want to get hyper-pedantic, our maze generation algorithm could be trivially converted into a maze-solving algorithm, and that *would* include backtracking)

So let's start with a stub of a program, before introducing what we'd like to solve:

```java
import java.util.Scanner;
public class Selection {
    public Selection() {
        Scanner input=new Scanner(System.in);
        Node lst=generate(8,1,8);
        dispList(lst);
        System.out.println();

        //The 'fun stuff' goes here!
    }

    //And also here!

    private void dispList(Node ptr) {
        if (ptr==null) return;
        System.out.print(" "+ptr.item);
        dispList(ptr.next);
    }
    private int genInt(int lb, int ub) {
        return (int)(Math.random()*(ub-lb+1)+lb);
    }

    //But start here!

    public static void main(String[] args) {new Selection();}
}
```

Okay, so let's do that 'starting here' bit: clearly I've created a Node type somewhere, to hold `integers`.
I'd like to start by producing some requested number of randomly-generated values (within some range).

There is *zero* reason to do this recursively... other than practice is good?
Let's say the arguments are as follows:

1. How many values we'd like to generate
2. The lower-bound for values
3. The upper-bound for values

You know how recursion goes. One thing we'll need is the 'trivial case'!

- What could be simpler than requesting *zero* values?
    - So if we're requesting `0`, then return `null`

For any other case, rather than looking at it as N values, it's really 1 value, followed by a list of `(N-1)`.

```java
private Node generate(int qty, int lb, int ub) {
    if (qty==0) return null;
    return new Node( genInt(lb,ub), generate(qty-1,lb,ub) );
}
```

Okay, so that wasn't so painful…

We could work out how to make the requests, even though there won't be anything for it to invoke yet:

```
while (true) {
    System.out.print("Target value: ");
    String line=input.nextLine();
    try {
        int choice=Integer.parseInt(line);
        Node selection=choose(choice,lst);
        if (selection==null)
            System.out.println("Narp.");
        else
            dispList(selection);
        System.out.println();
    }
    catch (NumberFormatException nfe) {break;}
}
```

Except you might want yours to not be terrible? Or whatever. I'm not the boss of you.

The point is, we'll be seeing a sequence of values, and we'll have the opportunity to request some *target sum*.

If *any* combination of those values can add up to that target, we'll want to see them.

If none is possible, then that's unfortunate. But we can try again for a different target sum.

So, uh… how do we solve this?

It's pretty obvious the *general* approach will be similar to our list examples from lecture. What's new this time?

- This time, a given location's value may or may not be part of the solution, and you can only determine either way *after* seeing if you can solve the 'subproblem' (of finding a smaller target)
    - Which might also include determining if a later value is included by looking for an *even smaller* target

So what are the trivial cases here?

1) You might be tempted to say *"when the target becomes zero"*
    - You *could* write a solution to do this, but it's more complicated than necessary. There's no reason you should ever *need* to recurse once there's nothing left to solve
2) Finding a Node with a value that *perfectly* matches the target would work!
3) What about hitting an empty list?

We'd need to account for both cases 2 *and* 3, with a couple caveats:

- Obviously we need to account for the `null` *first* (or we'll trigger an exception)
- For the first time, we *need* to include some form of **feedback**!

Consider the values 2 7 2 4 8 8 2 6, and a target of, say, 5.

Is that first 2 part of a solution? Well no, because 5 isn't possible, but we don't know that yet!

We'll only know after trying to solve for the 'remaining 3' on the list's tail.

Two steps further in, we'll encounter another 2 while solving for that 3. Is *that* part of the solution?

We need to try solving for 1 on the remaining values. It's only after we fall off the list that we could confirm the *second* 2 wasn't part of the solution. But even then we haven't confirmed that the first one isn't yet!

We need to somehow return to that first entry, communicate that we couldn't solve for 3 on the tail, and then try solving *for the full 5* on its tail.

That means we ostensibly need to answer 'yes' and 'no'. It wouldn't be weird to just make it a boolean function, but since we also want to solve for a list of values anyhoo, we can kill two birds with one stone:

- Our function will be returning a list containing the solution
- If no such solution exists, we'll be returning an *empty* list (`null`)

So we now have a bit more ammunition for solving this task:

- If we see the precise value we're solving for, we return a list consisting *solely* of that value
- If we're trying to solve on an empty list, we just return 'nothing' back
- Otherwise we'll have two possible cases: 'this value' **is** part of the solution, or it is **not** part of one

The only way we can determine that last one is by trying to solve the subproblem for a target reduced by the current value. If we get a list of value(s) back, then we return a new list comprised of our current value, followed by the subproblem's list.
If, however, we receive an empty list back, then we still don't *know* if it's solvable or not. But whether it is or isn't, that's determined entirely by whether or not the tail after that position can be used to solve the total original target for 'here'.

Don't you just *love* how that's literally the entire algorithm, and yet includes none of the code?
Yeah, that's kinda your job. So you should do that.

And then, that's it!

So long as you fully understand the algorithm, you shouldn't have a significant problem implementing it.
Sample execution:

```
4 6 1 2 1 4 8 5
Target value: 10
 4 6
Target value: 15
 4 6 1 4
Target value: 19
 4 6 1 2 1 5
Target value:
```