Function Name: `sumtorial`

Inputs:

1. (*double*) A positive integer

Outputs:

1. (*double*) The sumtorial of the input number

Background:

As a Universal Studios intern, you've been tasked with creating all of the trees for *The Lorax 2!* Every aspect of each tree has to be perfect - the color of each leaf, the texture of the branches, and most importantly the number of leaves per branch. You soon discover that the sumtorial (where you add all the successive numbers) gives you the perfect ratio for your branches. Realizing that you'll have to do millions of calculations for all the trees in the movie, you have the brilliant idea to write a MATLAB function to help you.

Function Description:

Write a function that takes in a number, and recursively finds the sumtorial of this number. To find the sumtorial, use the following information:

$$\text{sumtorial}(1) = 1$$

$$\text{sumtorial}(n) = n + \text{sumtorial}(n-1)$$

Notes:

- You must use recursion to solve this problem.

Function Name: `palingrove`

Inputs:

1. (*cell*) An MxN cell array containing information about a grove of trees

Outputs:

1. *(cell*) A PxN new cell array containing only the rows that are palindromes

Background:

You need to fulfill your final social science requirement before getting out, so you enroll in HTS 2803, History of Trees. You thought the class would be boring, but you find yourself inspired while reading about the nineteenth century pioneer John Chapman (better known as Johnny Appleseed). You decide to plant your very own orchard, containing not just apple trees but a wide variety of topiary.

Function Description:

A *palindrome* is a word, phrase, or sequence that reads the same backward as forward (e.g., "racecar"). To add symmetry to your orchard, you decide that every row in the grove should be a palindrome.

Write a function that takes in a cell array outlining the plan for your new orchard and removes any rows in the array that are not palindromes.

Example:

```
>> orchard = {'Fig', 'Apple', 'Fig'
             'Walnut', 'Maple', 'Orange'
             'Cherry', 'Palm', 'Cherry'}

>> newOrchard = palingrove(orchard)

newOrchard = {'Fig', 'Apple', 'Fig'
             'Cherry', 'Palm', 'Cherry'}
```

Notes:

- You must use recursion to determine if a particular row is a palindrome.
- The number of columns may be even or odd.
- Palindromes are case insensitive.

Function Name: `treeFarm`

Inputs:

1. (*double*) The length of the base of the tree
2. (char) A character to use as a leaf

Outputs:

1. (*char*) The created tree

Background:

You have just learned the most exciting concept in the history of programming: *Recursion*! You want to put this new power to use in the best way possible: growing trees for our planet.

Function Description:

Write a recursive function that takes in the length of the base of the tree as well as any single character with which to grow the tree (a leaf) and outputs an array of class `char` that contains the pyramid. The dimensions of your final array should be `<length(base)>` by `<(2*length(base))-1>`. Each row should have 1 more leaf than the row above it and a single space (ASCII value 32) should separate each leaf character within a row. Any other index within the array that does not contain a pyramid character should be filled with a space (ASCII value 32).

For example, for a tree of base length 10 and constructed from the character `'/'`, your function should output a 10x19 character array as shown below (in order to better visualize the actual construction of the array, the ASCII values for this array have been printed below with the leaf character's value highlighted)

```
         /
        / /
       / / /
      / / / /
     / / / / /
    / / / / / /
   / / / / / / /
  / / / / / / / /
 / / / / / / / / /
/ / / / / / / / / /
```

Continued...

```
32 32 32 32 32 32 32 32 32 47 32 32 32 32 32 32 32 32 32
32 32 32 32 32 32 32 32 47 32 47 32 32 32 32 32 32 32 32
32 32 32 32 32 32 32 47 32 47 32 47 32 32 32 32 32 32 32
32 32 32 32 32 32 47 32 47 32 47 32 47 32 32 32 32 32 32
32 32 32 32 32 47 32 47 32 47 32 47 32 47 32 32 32 32 32
32 32 32 32 47 32 47 32 47 32 47 32 47 32 47 32 32 32 32
32 32 32 47 32 47 32 47 32 47 32 47 32 47 32 47 32 32 32
32 32 47 32 47 32 47 32 47 32 47 32 47 32 47 32 47 32 32
32 47 32 47 32 47 32 47 32 47 32 47 32 47 32 47 32 47 32
47 32 47 32 47 32 47 32 47 32 47 32 47 32 47 32 47 32 47
```

Notes:

- The leaf character is guaranteed to be of length 1, and the size of the base is guaranteed to be at least 1.
- You must use recursion to solve this problem.

Hint:

- You can build the tree recursively by adding one row at a time and adding the appropriate padding on the side.

Function Name: `deepestPath`

Inputs:

1. (*cell*) A 1xN nested cell array

Outputs:

1. *(char*) A character vector describing the deepest path in the woods.

Background:

Recursion just shook your entire world and sense of well being. You no longer can tell left from right, or a double from a char. To get a grip on reality, you decide to take a long, introspective walk into the depths of a nearby forest. You have several different path options, and you are trying to decide which one will give you the longest path before you reach a dead end. You quickly realize the complexity of the situation, and you are about to give up when you find a map! Alas, a solution! The map you found is a map of nested cell arrays, and you immediately know what to do.

Function Description:

Write a recursive function that will search through a nested cell array to find the maximum depth of nested cells. Then output the string `'The deepest path is <num layers> cells deep.'`.

Example:

```
>> ca = {1, {true,{4,{'trees'}}}, {{{8,'leaves'}}}}
>> description = deepestPath(ca)
description = 'The deepest path is 4 cells deep.'
```

Notes:

- It does not matter what is found inside of the nested cell. You are only concerned with the depth.

Hints:

- What should the depth be when the input is not a cell/cell array?
- You will need a wrapper function to format the output string.
- Consider how you can call the recursive function on every element in the cell array.

Function Name: `runeScape`

Inputs:

1. (*struct*) 1x1 structure representing your character
2. (*struct*) 1x1 structure representing an item you want to wield

Outputs:

1. (*char*) Sentence describing how much total experience you need to use the item

Background:

Because you're a sensible and reasonable human being, you spend every waking moment playing RuneScape (old school, not RS3). Skilling (rather than killing) is serious business, so you earned a Ph.D in applied mathematics solely to understand the complex equations governing the amount of experience you need to reach the next level. Woodcutting is a skill, so this problem is about trees.

In the midst of one of your 14-hour skilling sessions, you come across an item that your character is not high enough level to equip! Upset as you are, you realize that this is a good opportunity to finally put your mathematics training to good use, and you want to write a function that will determine the most efficient way to level your character up so you can wield the item.

Function Description:

Write a function to recursively calculate the amount of experience you need to level up in a desired skill. Your character is represented by a 1x1 structure and contains your username (as a string) as well as all your characters' skills as fields. Each skill field contains your character's level in that skill as a double.

The item you want your character to equip will also be represented by a 1x1 structure. One field, `Name`, will be the name of the item as a string, and the rest of the fields will be double values representing the skill level requirements for that item.

`characterStruct:`

```
Username: <username>
<skill 1>: <level in skill 1>
<skill 2>: <level in skill 2>
...
<skill N>: <level in skill N>
```

`itemStruct:`

```
Name: <name of item>
<skill 1>: <level requirement of skill 1>
<skill 2>: <level requirement of skill 2>
...
<skill M>: <level requirement of skill M>
```

Continued...

In order to equip the item, your character must level up to match all the required levels. Your character can level up in a skill by gaining experience in that skill. The amount of experience your character needs to level up, $\mathrm{Exp}(n)$, depends on the level the skill is leveling up to, n, as well as the amount of experience required to level up to level $n-1$, according to the following formula:

$$\mathrm{Exp}(n) = \left\lceil n^{\frac{7}{2}} + \mathrm{Exp}(n-1) \right\rceil$$

$$\mathrm{Exp}(2) = 83$$

where the half-square brackets on the outside of the equation represent `ceil()`.

As an example using the equation, the amount of experience required to level up to level 3 from level 2 would be

$$\mathrm{Exp}(3) = \left\lceil 3^{\frac{7}{2}} + \mathrm{Exp}(2) \right\rceil = 130$$

Because your character could be deficient in more than one skill, to find the amount of experience your character needs to use the item, you should calculate the total amount of experience required to level up all deficient skills to the required skill levels as specified in the item structure.

After calculating the total amount of experience (across all deficient skills) needed to use the item, output the following sentence:

```
'<username> needs <# of total experience points> more experience points
                  to use the <name of the item>.'
```

Example:

`characterStruct:`

```
Username: 'xXcoolGuy2009Xx'
Attack: 54
Strength: 40
Defence: 58
Runecrafting: 1
Woodcutting: 99
```

`itemStruct:`

```
Name: 'dragon scimmmy'
Attack: 60
Strength: 60
```

```
>> str = runescape(characterStruct, itemStruct)
str ⇒ 'xXcoolGuy2009Xx needs 352118003 more experience points to use the
dragon scimmy.'
```

Extra Credit

Function Name: `lostWoods`

Inputs:

1. (*char*) The file name of an image of a maze
2. (*double*) The row index of the maze entrance
3. (*double*) The column index of the maze entrance

File Outputs:

1. An image of the solved maze

Background:

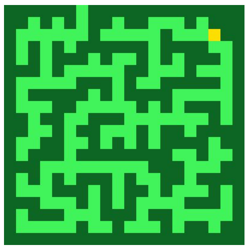
You stand at the entrance of a dark, deep forest. You only see shadow ahead of you, but still you step in. The old stories say that somewhere deep in these woods lie riches and glory beyond your wildest imagination, and the allure of the treasure overcomes your fear. You take a few steps into the woods and are shrouded in darkness--you stop while you can still see the light of the entrance behind you. *There must be a better way...*

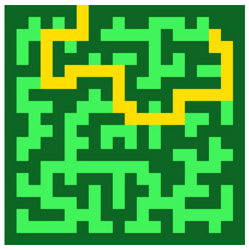
Function Description:

Write a function in MATLAB that takes in the name of an image file. This image is a map of the forest, and you are going to find the path to the treasure before you even step into the maze! The image can be of any size, but the rows and columns are divisible by 20- the maze is made of 20x20 solid colored squares. The walls of the maze are dark green, RGB(13,102,35), the paths of the maze are a lighter green, RGB(66,244,92), and the treasure is a gold color, RGB(255,223,0). The second and third inputs to the function are the indices of where the start square is on the edge of the maze. Your job is to solve the maze by drawing a path from the maze entrance to the treasure. To draw this path, change the color from the lighter green to gold (RGB(255,223,0)) at every square along the path. You will save this to a new image that has the original filename with `'_solved'` appended.

Example:

If the image on the left is called `'maze.png'`, then running `lostWoods('maze.png',1,7)` should produce the image on the right, `'maze_solved.png'`.





Continued...

Notes:

- There will only be one unique path to the treasure.
- You cannot move diagonally, just up/down/left/right.
- A helper function has been provided to generate additional test cases, `makeMaze()`
    - Run it with the format `[r,c] = makeMaze(row, col, name)`, where `row` and `col` are the dimensions of the maze you want to make and `name` is the file name.
    - It will produce an image with the designated name as well as `r` and `c`, the start indices.

Hints:

- The recursion part of this is very similar to that in `deepestPath.`
- Think about how you would solve the maze if you could only see one square at a time.
    - You don't know where the goal is unless you are currently at the goal, so your only choice is to try going in each direction possible and repeating.
- Try to isolate the maze solving aspect of this problem from the image solving aspect of the problem and solve these two problems separately.