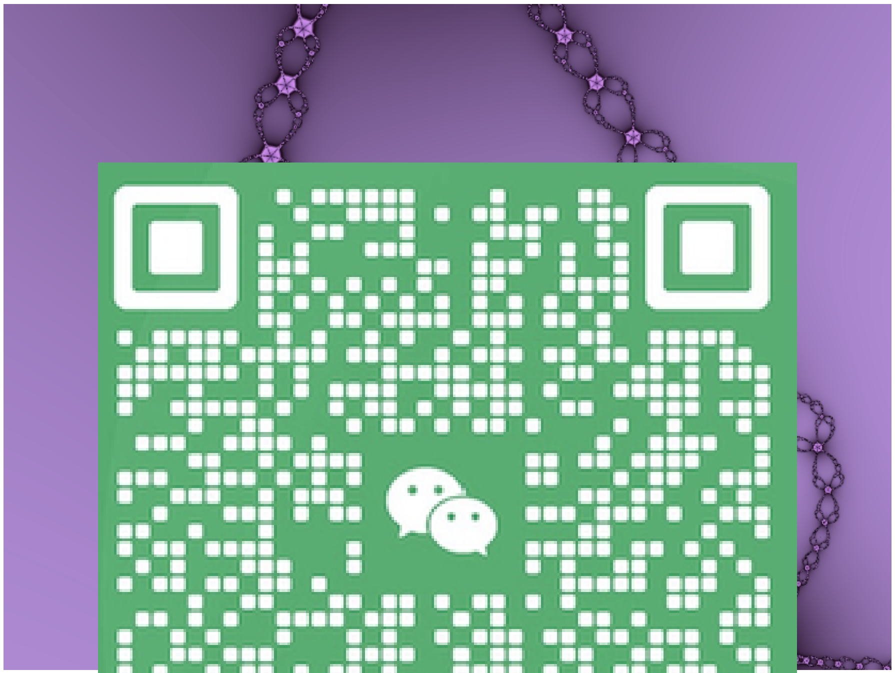
CS 202 Spring 2024 - Assignment 7

Recursion



Contents

1 Overview

Recursion is a powerful tool for problem solving. While not always the most efficient, it allows us to reason through problems that might otherwise be difficult to solve. This assignment differs slightly from the others in that it is not a cohesive program so much as a set of problems. You will be asked to solve several problems recursively. Functions will call themselves - no loops allowed unless explicitly mentioned. Each function will be described below and in more detail in the associated video.

Recall the recursion has three necessary parts - the actual recursion (when a function calls itself or when we define a problem in terms of itself), a base case (a trivial or defined case to solve that can be done in a few steps), and a reduction operation (that reduces some parameter to change the problem for the next recursive call). For each of the problems provided, try to reason through trivial solutions to each and think about how you can call the function again with some modified parameter. A key to thinking recursively is to limit your scope - don't think about the whole big problem, just think about what needs to happen on the one single call to the function. What does the function need to do on just this *one* step and what is the next step (i.e. the next call to make)?

The assignment is split amongst three parts that can be approached in any order. The first set of problems are contained [illegible] functions. The next two parts involve classes wi[illegible] set of classes, found in **cake.h**, are the **Cake** a[illegible] of each other to form a whole cake. The second[illegible] PacMaze classes, where the former is a 2D (x, y[illegible]

2 Classes and Functions

Below is a list of functions and variables that are important for this assignment. ***Variables are in green***, **functions that you will need to write are in red**, **functions implemented for you already are in blue**, and **functions that are abstract that will be implemented later are in magenta**.

General Recursion (general_recursion.h)

The general_recursion header file contains four global functions that are unrelated and can each be solved separately. All functions must be solved recursively - that is, call themselves. No loops allowed.

- **int findSum(int n)** - This should find the sum of all non-negative integers up to the given n. For example, if n = 4, then 1 + 2 + 3 + 4 = 10. Remember that all recursive functions must have a base case, a recursive call, and a reduction operation. Think about what number is trivial to take the sum
  up to for the base n of the problem for the
  other two parts.

- **int intLog(const** n number and given log
  base. Logarithm the base goes into the
  number. For exa times. For a base case,
  consider a trivial l og regardless of base? Is
  there a number w ly?). Then, recursively
  find the log of so **se static variables to**
  solve the proble it local.

- **void printRever** rds to cout. This should
  print one characte . Consider what string
  would be trivial t hat string.

- **bool isValueInA** - Given a sorted array,
  this will tell if a g to search, the starting
  index, length, and , 6, 7, 9, 12, 15, 17, the
  start was 2, the le would return false since
  the number 1 is n 2. This function should
  be implemented u video, but will be talked
  about briefly here iven that a structure is
  already sorted. F ue being searched for, if
  the value is less than the middle, it must be in the array to the left, if it is greater it is in the array to the right, and if it is equal to the middle then the value has been found. Some calculations for lengths and indices have been provided in the skeleton.

Cake (cake.cpp)

The cake.cpp file centers around functions which be used to bake a delicious recursive cake. Two classes are provided: Cake and CakeLayer. The Cake is made of several CakeLayers stacked upon one another. In this file, you are asked to write the functions to recursively add CakeLayers to the top of the Cake and print the Cake layer by layer. You are also responsible for eating the Cake (destroying it with destructors).

Globals

- *const int CAKE_ WIDTH = 33* A constant used for formatting the cake when printing layers. Used by the skeleton

CakeLayer class

The CakeLayer class represents a single layer within the cake (pound cake, icing, etc.). It contains a few members for formatting and printing as well as a pointer to the CakeLayer directly above it within the Cake. The CakeLayers will be stacked on top of each other and held together by each layer knowing about it's neighbor to the top (if a

- **- *int width*** - The on
- **- *int height*** - Th ton
- **- *char drawCha*** y the skeleton
- **+ *CakeLayer**** n the Cake. If there are no layers above ***th***
- **+ void printLay** ut
- **+ CakeLayer(in** ') - Constructor which initializes CakeLa is initialized to nullptr
- **+ ~CakeLayer(** s the only function you need to write. Th thought.

Cake Class

The Cake class holds m

- **+ *CakeLayer**** is is initialized in main, you can assume it
- **void printCake(CakeLayer* current_ layer)** - Prints the entire cake recursively. The Cake should be printed from the top downwards. This should print the given layer as well as the rest of the Cake (HINT: When does the rest of the Cake need printed with respect to the current layer?).
- **+ void printCake()** - A wrapper function which does the initial call to the other printCake overload. Simply calls the Cake to begin printing starting with the bottom layer.
- **- void addNewLayerToTop(CakeLayer* base_ layer, CakeLayer* new_ layer)** - Adds the given new layer to the top of a Cake with the given base layer. The base is effectively the bottom of the Cake, but should not be confused with the ***bottom*** member. The new layer should be added to the top of the Cake by setting the top layer's ***layerAbove*** to that new layer. You can assume that main will create all objects before calling this function, including the bottom of the Cake
- **+ void addNewLayerToTop(CakeLayer* new_ layer)** - Wrapper function which calls the recursive overload of addNewLayerToTop. Simply attempts to add the new layer starting with the bottom as the base.

- **+ void addBase(CakeLayer* bottom)** - Initializes the bottom of the Cake to the given layer. This exists as a separate function to make addNewLayerToTop simpler to do recursively. Used by the skeleton
- **+ Cake()** - Initializes the Cake to be empty
- **+ ~Cake()** - Destroys all layers from the Cake by deleting the bottom CakeLayer, which should recursively delete all layers above it

PacMaze (pacmaze.cpp)

The functions within pacmaze.cpp create a maze similar to that of the hit video game Pacman™. Two classes can be found within pacmaze.h: Position and PacMaze. The former is used to represent a place within the latter. You are asked to populate an empty maze with pellets so PacMan™can eat them.

Globals

- ***Position RIGH*** [illegible] ne space directly to the right. Think of th [illegible] nce, but optional to use
- ***Position DOW*** [illegible] space directly to below. Exists for conveni [illegible]

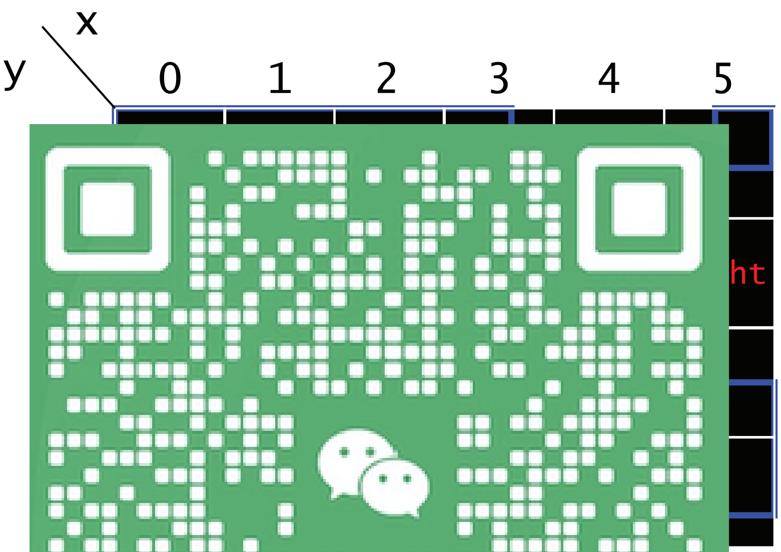
Position Struct

The Position class is us [illegible] igin, (0, 0), of the maze is the top left. In 2D ar [illegible] hile moving downwards (since it represents the [illegible] ce from the left wall of the maze and the y val [illegible] You can see the x and y components labeled i [illegible] implemented for you.

- ***int x*** - The horiz [illegible]
- ***int y*** - The vertic [illegible] smaller numbers up.
- **bool operator** [illegible] which are equal if both the x and the y c [illegible]
- **Position& oper** [illegible] the given Position.
- **Position operat** [illegible] gether elementwise and returns the result.
- **Position operator - (const Position& other)** - Subtracts two Positions together elementwise and returns the result.
- **Position addOffset(int delta_x, int delta_y)** - Adds the offset delta_x and delta_y to a Position and returns the offset Position. This can be used to easily add offsets for convenience. Effectively this function works like the + and - overloads but allows individual integers to be input.
- **void saturateBounds(const int& right, const int& bottom)** - Saturates the x component of the position to be in the range [0, right) and saturates the y component to the range [0, bottom). This is used to keep Positions within the bounds of the 2D maze. The saturate operation will provide the wrap around effect found in Pacman (e.g. going off the left side of the maze wraps back to the right).
- **string toString()** - Returns a nicely formatted string of the form *(x, y)*. This is provided for your debugging convenience.
- **Position(int x, int y)** - Constructor to make a Position with the given x and y coordinates.
- **Position()** - Constructs an unitialized Position.

PacMaze Class

The PacMaze class represents a maze which can be explored by PacMan™and the wacky ghosts Inky™, Pinky™, Blinky™, and Clyde™. The maze is represented using a 2D (linearized) array of characters. Walls within the maze are drawn using the ASCII characters |, +, and -. The maze is created within main without any pellets. You are asked to populate the maze with pellets by recursively traversing it. Two types of pellets can be placed within the maze: normal (small) pellets and power pellets. Power pellets are placed at intersections where all adjacent tiles are open for PacMan to travel through and all other empty tiles are filled with normal pelelts. Helper functions are provided to make interacting with the maze (and the linear array) simpler. Below is an example maze with empty spots showing the x and y values of each Position as well as where power pellets would be placed.



- - char* maze - I ...) grid
- - *int width* - Wi
- - *int height* - He
- - *const char sm* ... ets within the maze
- - *const char po* ... llets within the maze
- **- bool isSolid(const Position& pos)** - Returns true if the given Position within the maze contains a solid wall (|, -, +)
- **- bool isEmpty(const Position& pos)** - Returns true if the given Position within the maze contains no wall nor any pellets
- **- char getCharAtPos(const Position& pos)** - Returns the character at the given Position within the maze
- **- void placePellet(const Position& pos, char pellet)** - Places the given pellet at the given Position within the maze
- **+ PacMaze(int width, int height, char* maze)** - Initializes the maze to the corresponding parameter and sets the width and height members

- **+ void drawMaze()** - Draws the maze in a formatted fashion to standard out. This used color formatting, which may appear strange on CodeGrade. If you see strange strings in the output, note that they are for achieving color within your terminal. It is recommended to debug the class in the terminal first. This function is called by the skeleton, but you may find it useful to aid with debugging when writing the populateMaze function.

- **+ void populateMaze(Position current_pos)** - This function should recursively populate the maze with pellets by placing a pellet in the current position if valid and then recursively trying to do the same with adjacent Positions (right, left, above, below). The function can be approached in several different ways, and the implementation is left as a problem for you to solve. Pellets should not be placed if there is already a pellet at the current position or in positions which contain walls. Helper functions can be used to check for these conditions. If the above, below, right, and left Positions within the maze are not solid, a power pellet should be placed, otherwise a normal pellet should be used. A function call is provided at the top which saturates the Position to the bounds of the PacMaze (i.e. wraps any Positions which would fall out of bounds back to the other side. For example if the width was 10 and a Position had an x component of 10, the saturate would wrap it back around to the left side with value 0) [illegible] the intended solution, consider in what [illegible]

3 Compiling / TO-DO

A makefile is provided to compile all files. Alternatively, there is only one program for this assignment, so compiling all of the cpp files is also fine.

The functions that need to be written are located in **general_recursion.h**, **cake.cpp**, and **pacmaze.cpp**. The functions that need code are indicated by a ***TODO*** comment above them and are color-coded in previous sections. Each function can be tested separately, so debug at your own pace.

It is recommended to start with the general recursion functions and work your way up to the more complex functions located within classes. Cake is intended to be simpler than PacMaze.

4 Sample Runs

Below are sample outputs for each of the functions' test cases.

findSum

```
Sum of 1 to 10: 55
Sum of 1 to 400: 802
```

intLog

```
log2(16) = 4
log10(100000) = 5
log478(1) = 0
```

printReverse

```
"Hello World" revers
"dlroW eybdooG" reve
```

isValueInArray

```
Array 1: { 1, 2, 3,

Is 13 in the array?
Is 1 in the array? Y
Is 4 in the array? Y
Is 20 in the array?

Array 2: { -3, 0, 4, 5, 8, 9, 11, 200 }

Is 200 in the array? Yes
Is -4 in the array? No
```

PacMaze class (without color)

```
Initial Maze:
+ +------+ +---+ +--+ +-
|                     |
| +--+ +--++ +---+ ++ |
+ |  | |   | |   | || +-
  |  | | +++ +-+ | ||
+ +--+ +-+     | | || +-
|          +-+ +-+ ++ |
| +--+ +-+ | |        |
| |  | | | | | +-+ ++ |
| +--+ +-+ +-+ +-+ ++ |
|                     |
| +------+ +---+ +--+ |
Inserting pellets starting from position (1, 0)...
Done inserting!
+o+------+o+---+o+--
|ooooooooooooooooooo
|o+--+o+--++o+---+o+
+o|  |o|   |o|   |o|
oo|  |o| +++o+-+ |o|
+o+--+o+-+ooooo| |o|
|ooooo@oooo+-+o+--o+
|o+--+o+-+o| |oooo@o
|o|  |o| |o| |o+-+o+
|o+--+o+-+o+-+o+-+o+
|ooooooooo@ooooooooo
|o+------+o+---+o+--
```

Cake class

```
Adding the plate...
Adding pound cake...
And cream...
And more pound cake...
How’s it looking?

                ^
               (/)
  ‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘
  ‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘
  ‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  ‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘
  ‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘
  ‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘
--------------------

Pretty good, but we
Adding chocolate cak
And chocolate cream.
And more chocolate c
Adding the finishing
The cake is complete
                ^
               (/)
           #########
           #########
           ~~~~~~~~~
           ‘‘‘‘‘‘‘‘
      ##############
      ##############
      ~~~~~~~~~~~~~~
      ##############
      ##############
  ‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘
  ‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘
  ‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  ‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘
  ‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘
  ‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘
---------------------------------
```