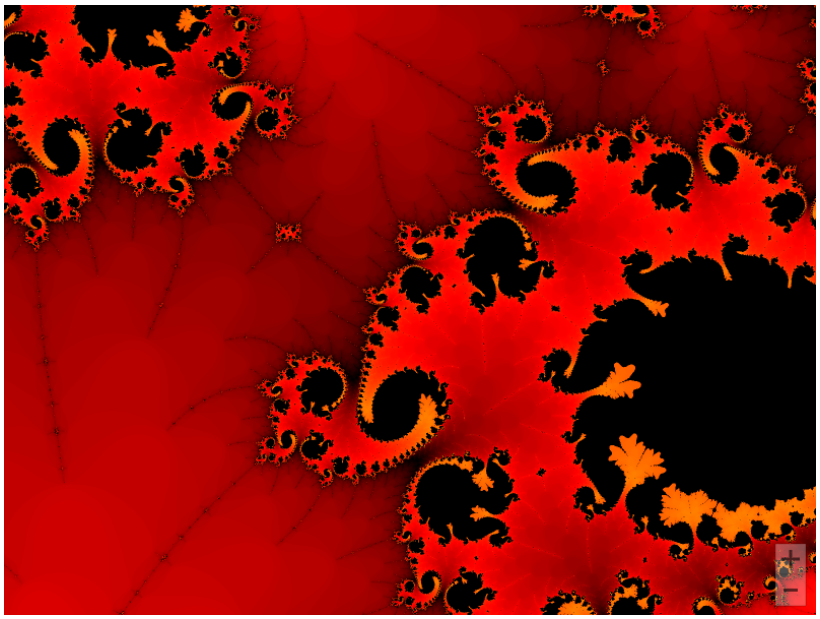
CS 202 Fall 2023 - Assignment 7

Recursion



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. This will mean having functions that call themselves - no loops allowed. 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 within *general_recursion.h* and contains simple, unrelated functions. The second part is *matryoshka.cpp*, which will implement a short Matryoshka Doll class, where dolls are placed within each other. The final is *sweepminer.cpp* in which a legally distinct from Minsweeper game, Sweepminer, will be setup with bombs and distances to those bombs. All other files have been implemented.

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**.

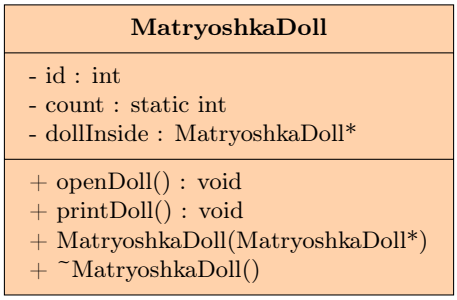
Global Functions (general_recursion.h)

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

- **void countBackwards(int n)** - This function should count backwards from the given **n** until it hits 0 by printing each number followed by a comma. See the sample output for formatting.
- **int sumUpToN(int n)** - This should find the sum of all numbers 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 case and then how you can apply n to some reduced version of the problem for the other two parts.
- **bool isValueInArray(int arr[], int start, int length, const int& value)** - Given a sorted array, this will tell if a given value is in that array. This function is given the array to search, the starting and ending indices in the array where the search is being applied, and the value to search for. For example, if the array was 1, 5, 6, 7, 9, 12, 15, 17, the start was 2, the end was 5, and the value to search for was 1, the function would return false since the number 1 is not in the sub array from index 2 to 5 - 6, 7, 9, 12. This function should be implemented using binary search, which is explored further in the included video, but will be talked about briefly here. Binary search is designed to be a quick way to search given that a structure is already sorted. For an array, it involves checking the middle of the list, 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 the same as the middle then the value has been found. Some calculations to implement this have been provided in the skeleton.

MatryoshkaDoll

The MatryoshkaDoll class represents a Russian Matryoshka Doll (sometimes also called nesting dolls) that can be stacked within each other. These dolls offer a fair visual representation of recursion, since each doll contains within itself another doll with potentially even more dolls inside. This class will simulate just that, and the dolls will be recursively opened to display them to the screen as well as deallocate any dynamically allocated dolls.



- ***int id*** - The id for the doll. These are automatically assigned via the constructor and serve as a way to distinguish dolls from one another. The first doll created has id 0, the second has id 1, etc.
- ***static int count*** - A count of how many dolls have been created.
- ***MatryoshkaDoll* dollInside*** - A pointer to the MatryoshkaDoll contained within this one. If there is no doll contained within, this will point to nullptr.
- **MatryoshkaDoll(MatryoshkaDoll* innerDoll)** - Constructor for the class which sets the id and increments the count. Sets the doll contained within the one being constructed to the *innerDoll* parameter.
- **printDoll()** - Prints the doll's id and count of how many dolls there are.
- **openDoll()** - This should open the doll followed by all of the dolls inside itself. First, print the doll's information using the **printDoll** function, and then open all the dolls inside if there are any. Note that depending on how you approach this, your base case might be implicit (that is, there's no specific condition for it).
- **~MatryoshkaDoll()** - The destructor should deallocate all dolls contained within the one being deallocated. This function should be two lines of code, if there is more you might be overcomplicating things. While short, think through how you might utilize aspects of C++ to effectively deallocate all dolls using recursion. The outer most doll will be deallocated by the main.

Sweepminer

Based on the popular smash hit, Minesweeper, Sweepminer is a similar game but with a slightly different setup. Sweepminer is played with a 2D grid where bombs are placed randomly and must be flagged as unsafe. Unlike Minesweeper, each spot in the grid contains a number which represents the distance to the closest bomb, where bomb spaces have a distance of 0.

A Position struct will also be provided to make interacting with the board easier. This class simply represents a (x, y) position on the board and comes with optional helper functions to make using the class easier. When using arrays, consider that the origin (0, 0) is the top left of the array. The y values get bigger as you go downwards, which is opposite of standard mathematics. We should also note that since the array is allocated to be row major, the row will come before the column when indexing, meaning the array is accessed with the y value first, then the x value. To help avoid confusion, helper functions which will be provided and described below, although if you decide to interact with the board array and the Position class yourself, the previously mentioned attributes need to be considered.

For this class. you will be asked to dynamically allocate and deallocate the 2D board as well as recursively find the distance to a bomb to setup the board. Let's first go through how inserting bombs onto the field works and take a look at an example board.

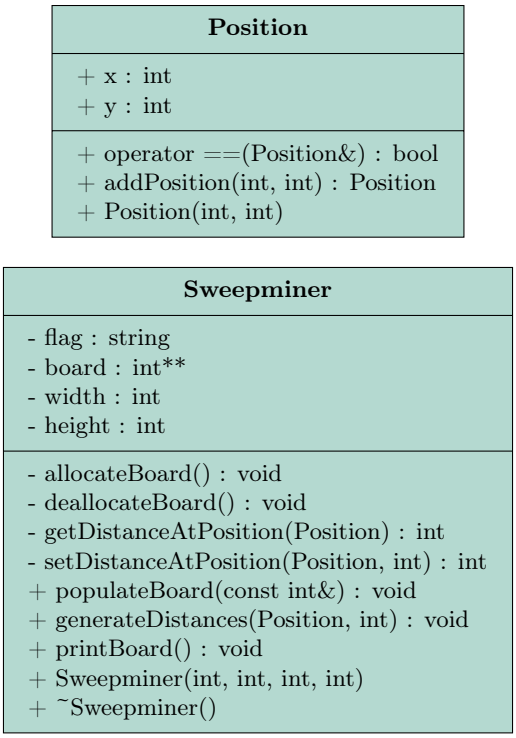
3	2	1	2	3	4
2	1	F	1	2	3
3	2	1	2	3	4
4	3	2	3	4	5

The above board has two bombs marked with an F (for flag). Each square in the board represents the Manhattan With only one bomb, the distances are fairly straightforward. The bomb's spot at Position (2, 1) has a distance of 0 (marked with an F), the squares above, below, left, and right have a distance of 1, and the squares adjacent to those have a distance of 2, etc. Let's now introduce a second bomb.

3	2	1	2	3	2
2	1	F	1	2	1
3	2	1	2	1	F
4	3	2	3	2	1

In the above picture, the squares closest to each bomb are marked in the same color as that bomb, while squares equidistant to multiple bombs are marked in black. It can be hard to make out with such a small grid, but what are being formed here are pixelated circles for each bomb where the colliding parts form a boundary. The challenge here is locating which bomb is the closest. While there may be more efficient approaches, we will naively insert one bomb at a time and simply mark all of the tiles that are now closest to that bomb. In this example, let's assume the blue bomb in the bottom right, (5, 2), is inserted second. The values from the first example will be overwritten with the new values in blue, since those squares are now closer to the second bomb than the first. To easily find these distances, we will begin at the new bomb with distance 0, and then mark all adjacent squares with distance + 1. If the new distance being placed into the square is less than the existing distance at that spot, we can overwrite it to mark the spot's distance as closer to the new bomb. This approach is nice in that it both addresses overwriting old values that were far away from existing bombs, but also addresses an issue that a keen observer may have noticed. The issue being that it is possible using this rule to backtrack onto a square that we have already put a distance into, however note that every time we move one square, the distance gets larger by one, meaning that when this issue happens, any values will not be replaced, since they will always be smaller. The number of squares we check will also be bounded, and will stop once all of the boundary squares in black or the walls of the board are hit.

This will be implemented recursively, where the distance in a square will be checked. If the distance to a bomb is less than the distance in that square, it can be overwritten and adjacent squares can be recursively tried as long as they are in bounds of the board. Otherwise, we know we have a base case of being at a boundary and do not need to try any adjacent squares. For a more animated example, please see the video for this assignment.



Position Members

- ***int x*** - The horizontal component of the Position.
- ***int y*** - The vertical component of the Position. Bigger numbers are down and smaller numbers up.
- **bool operator ==(Position& other)** - Compares two positions, which are equal if both the x and the y coordinates are the same. Provided for convenience.
- **Position addPosition(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.
- **Position(int x, int y)** - Constructor to make a Position with the given x and y coordinates.

Sweepminer Members

- ***const string flag*** - A constant for printing a flagged bomb as a red F. This is used by the skeleton code and is not needed for your code.
- ***int** board*** - A 2D dynamically allocated array of integers. This should be row major, where the first dimension is the rows and the second is the columns.
- ***int width*** - The width of the 2D board array.
- ***int height*** - The height of the 2D board array.
- **allocateBoard()** - Dynamically allocates the 2D board array. This should not linearize the array. You can assume the ***width*** and ***height*** have been set before this is called.
- **deallocateBoard()** - Deallocates the 2D board array allocated in **allocateBoard**. This is called be the destructor.

- **generateDistances(Position pos, int distance)** - This is the recursive function to be implemented for the Sweepminer class. It should populate the board with the distances for a single bomb as shown in the above example. This should generate the distance for the spot at the provided Position and then try to generateDistances for any adjacent spots. If the distance already at the current spot is closer to another bomb, then the spot does not need to be set nor do any of the adjacent squares. You must also check that any spots are in bounds of the array before accessing them, otherwise you may get a segfault (For example, in a 3x4 board, row 6 does not exist, neither does column -1, etc.). Like the openDoll function, the bases cases here may be implicit, where the calls that return may not be represented by some explicit conditional.
- **getDistanceAtPosition(Position pos)** - Returns the current distance value at the given Position on the board. Exists for convenience.
- **setDistanceAtPosition(Position pos, int dist)** - Sets the distance value at the given Position on the board. Exists for convenience.
- **populateBoard(const int& bomb_count)** - Randomly puts the given number of bombs onto the board.
- **printBoard()** - Prints all of the distances for the board in a pretty fashion. This is intended for debugging purposes and is also used by the skeleton.
- **Sweepminer(int width, int height, int num_bombs = 3, int seed = 1024)** - Creates a Sweepminer board with the given dimensions. Sets the corresponding members of the class, allocates the board, and then populates the board with the given number of bombs.
- **~Sweepminer()** - Deallocates the dynamically allocated board.

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 viable.

The functions that need to be written are located in **general_recursion.h**, **matryoshka.cpp**, and **sweepminer.cpp**. The functions that need code will be marked with a TODO comment. Each function can be tested separately, so debug at your own pace.

There is also a preprocessor directive included in the skeleton code for the sweepminer class to make the debugging process earlier. Please uncomment the DEBUG preprocessor definition at the top of the sweepminer.cpp file if you would like to see the state of the board and where each bomb were inserted. The code generated when this is uncommented will print information in between each bomb insertion and can be used to get an idea of whether the board state is correct or where your program is crashing.

Sample Output

Below is the expected output for the test cases for each function

countBackwards

```
Counting backwards from 4:
4, 3, 2, 1, 0
Counting backwards from 86:
86, 85, 84, 83, 82, 81, 80, 79, 78, 77, 76, 75, 74, 73, 72, 71, 70, 69, 68, 67, 66, 65, 64, 63, 62, 61,
```

sumUpToN

```
Sum of 1 to 10: 55
Sum of 1 to 99: 4950
```

isValueInArray

```
Array:
{1, 2, 6, 8, 10, 12, 13, 18, 19, 20, 21, 21, 22, 23, 28, 30}

Is 18 in the array? Yes
Is 9 in the array? No
Is 23 in the array? Yes
```

Matryoshka Doll class

```
Opening doll (8 / 9)
Opening doll (7 / 9)
Opening doll (6 / 9)
Opening doll (5 / 9)
Opening doll (4 / 9)
Opening doll (3 / 9)
Opening doll (2 / 9)
Opening doll (1 / 9)
Opening doll (0 / 9)
```

Sweepminer class

(Note that this may depend on your OS since it's randomly seeded. If your output doesn't match, try CodeGrade)

```
-------------------------------------------------
|  9  8  7  6  5  4  3  2  1  F  1  2  3  4  5  6 |
|  8  7  6  5  6  5  4  3  2  1  2  3  4  5  6  7 |
|  7  6  5  4  5  5  4  3  2  1  2  3  4  5  6  7 |
|  6  5  4  3  4  4  3  2  1  F  1  2  3  4  5  6 |
|  5  4  3  2  3  3  2  3  2  1  2  2  3  4  5  6 |
|  4  3  2  1  2  2  1  2  3  2  2  1  2  3  4  5 |
|  3  2  1  F  1  1  F  1  2  2  1  F  1  2  3  4 |
|  4  3  2  1  2  2  1  2  3  3  2  1  2  3  4  5 |
-------------------------------------------------
```