CS 314 - Specification 6 - Recursion

Programming Assignment 6: (Pair Assignment) You ***may*** work with one other person on this assignment using the pair programming technique. **If you choose to work with a partner, you must work with someone in the same section as you. In other words you must have the same TA.** Review this paper on pair programming. You are **not** required to work in a pair on the assignment. (You may complete it by yourself if you wish.) If you begin working with one partner and decide to **NOT** finish the assignment with that partner you must complete the assignment individually. (You must both start over from scratch). If you work with a partner, the intent is that you work together, at the same computer. One person "drives" (does the typing and explains what they are doing) and the other person "navigates" (watches and asks questions when something is unclear). You **shall not** partition the work, work on your own, and then put things together.

You and your partner may not acquire from any source (e.g. another student or an internet site) a partial or complete solution to a problem or project that has been assigned. You and your partner may **not** show other students your solution to an assignment. You **may not** have another person (current student other than your partner, former student, tutor, friend, anyone) "walk you through" how to solve an assignment. You may get help from the instructional staff. You may discuss general ideas and approaches with other students. Review the class policy on collaboration from the syllabus. If you took CS314 in a previous semester and worked with a partner you must start from scratch on the assignment. Likewise, if you took CS314 in a previous semester and work with a partner this semester, you must start from scratch.

- Placed online: Tuesday, February 21
- 20 points, ~2% of final grade.
- Due: no later than 11 pm, Thursday, Thursday, March 2
- General Assignment Requirements

The purpose of this assignment is to practice writing and implementing recursive algorithms.

In this assignment you will implement eight recursive methods. Some of the methods (1 - 3) use recursion to solve problems that could just as easily (if not more easily) be solved using simple iteration (for loops or while loops), but you solve them recursively for the practice.

Some of the methods still need a normal loop like the recursive backtracking examples shown in class to make choices. (Such as the determining the space taken up by files and the N Queens examples done in class. Those were both examples of recursive backtracking that used for loops to iterate through choices at a certain point.) The tester code will be placed in a class named RecursiveTester.

Files:

	File	Responsibility
Source Code	Recursive.java.	Provided by me and you
Source Code	RecursiveTester.java. Contains test cases for methods in Recursive.java.	Provided by you and me.
Utility Class	DrawingPanel.java A class for creating simple graphics windows. You need this class for the carpet problem.	Provided by me
Utility class	Stopwatch.java. A class for calculating elapsed time when running other code.	Provided by me
Documentation	Javadoc for the Recursive class.	Provided by me

The provided file Recursive.java contains eight methods to complete. For each method, clearly state the base case and recursive step for each method.

The provided file RecursiveTester.java contains a number of tests of the methods in Recursive.java. Some of these tests may be incorrect. Find and fix any incorrect tests. Your implementation of Recursive.java must pass these tests. Add at least 2 tests for each method other than carpet method to RecursiveTester.java. (14 total tests) When you turn in the file delete the provided tests and just include your tests.

Submission: Fill in the header in the RecursiveTester.java and Recursive.java files. Replace <NAME> with your name. Note, you are stating, on your honor, that you did the assignment on your own or with a single partner.

Create a zip file name a6.zip with your RecursiveTester.java and Recursive.java files. The zip file must not contain any directory structure, just the required file.

See this page for instructions on how to create a zip via Eclipse.

Turn in a6.zip via your Canvas account to programming assignment 6.

- Ensure you files are named RecursiveTester.java and Recursive.java. Failure to do so will result in points off.
- Ensure RecursiveTester.java and Recursive.java are part of the default package. Do not add a `package` statement to either file.
- Ensure your zip has no internal directory structure. When the file is unzipped no directories (folders) are created. (Note, the built in unzip feature of some versions of Windows and Apple OS X "help" by adding a directory when you unzip with the same name as the file. The unzip we use on the CS Linux system does not do this. Neither do unzip programs such as 7zip.)
- Ensure you submit the assignment under Programming Assignment 6 in Canvas.

Checklist: Did you remember to:

- review and follow the general assignment requirements?
- work on the assignment with at most one other person?
- fill in the headers in RecursiveTester.java and Recursive.java? If working with a partner only one of you turns in the zip, but you must be sure both of your names and UTEIDs are in the header information of the Java files you turn in.
- ensure your methods in Recursive.java pass the tests in RecursiveTester.java and add your own tests? Delete the old tests after you are sure you pass them. Add at least 2 tests for each method other than the carpet method.
- turn in your a6.zip file with your Java source code files (RecursiveTester.java and Recursive.java) to Canvas before 11 pm, Thursday, March 2? If working with a partner only one of you turns in the zip, but you must be sure both of your names and UTEIDs are in the header information of the Java files you turn in. If you are using slip days, both of you must have the required number of slip days. You cannot donate slip days to one another.

Individual Problem Descriptions from Recursive.java:

1. Binary Conversion

```
public String getBinary(int n)
```

(Simple Recursion) Write a method to recursively creates a String that is the binary representation of an int N.

Here is an example. The binary equivalent of 13 may be found by repeatedly dividing 13 by 2. If there is a remainder a 1 gets concatenated to the resulting String otherwise a zero gets concatenated to the resulting String. (The tricky part is figuring out the right time and place to do the concatenation.)

13 / 2 = 6 r 1
6 / 2 = 3 r 0
3 / 2 = 1 r 1
1 / 2 = 0 r 1

13 in base 2 is 1101. The method returns the String "1101".

2. Reversing a String

```
public String revString(String target)
```

(Simple Recursion) Write a recursive method that returns a String that is the reverse of the input String.

(It is acceptable to use String concatenation in your answer. StringBuilder not required.)

```
System.out.println( recursiveObject.revString("Calvin and Hobbes"));
```

would return a String equal to this: `"sebboH dna nivlaC"`

3. nextIsDouble

```
public int nextIsDouble(int[] data){
```

(Simple Recursion) This method returns the number of elements in data that are followed directly by double that element.

For example, the given array `{1, 2, 4, 8, 16, 32, 64, 128, 256}` the method would return 8. The elements 1, 2, 4, 8, 16, 32, 64, and 128 are all followed directly by double their value.

Given the array `{1, 3, 4, 2, 32, 8, 128, -5, 6}` the method would return 0. No element in the array is followed directly by double that element.

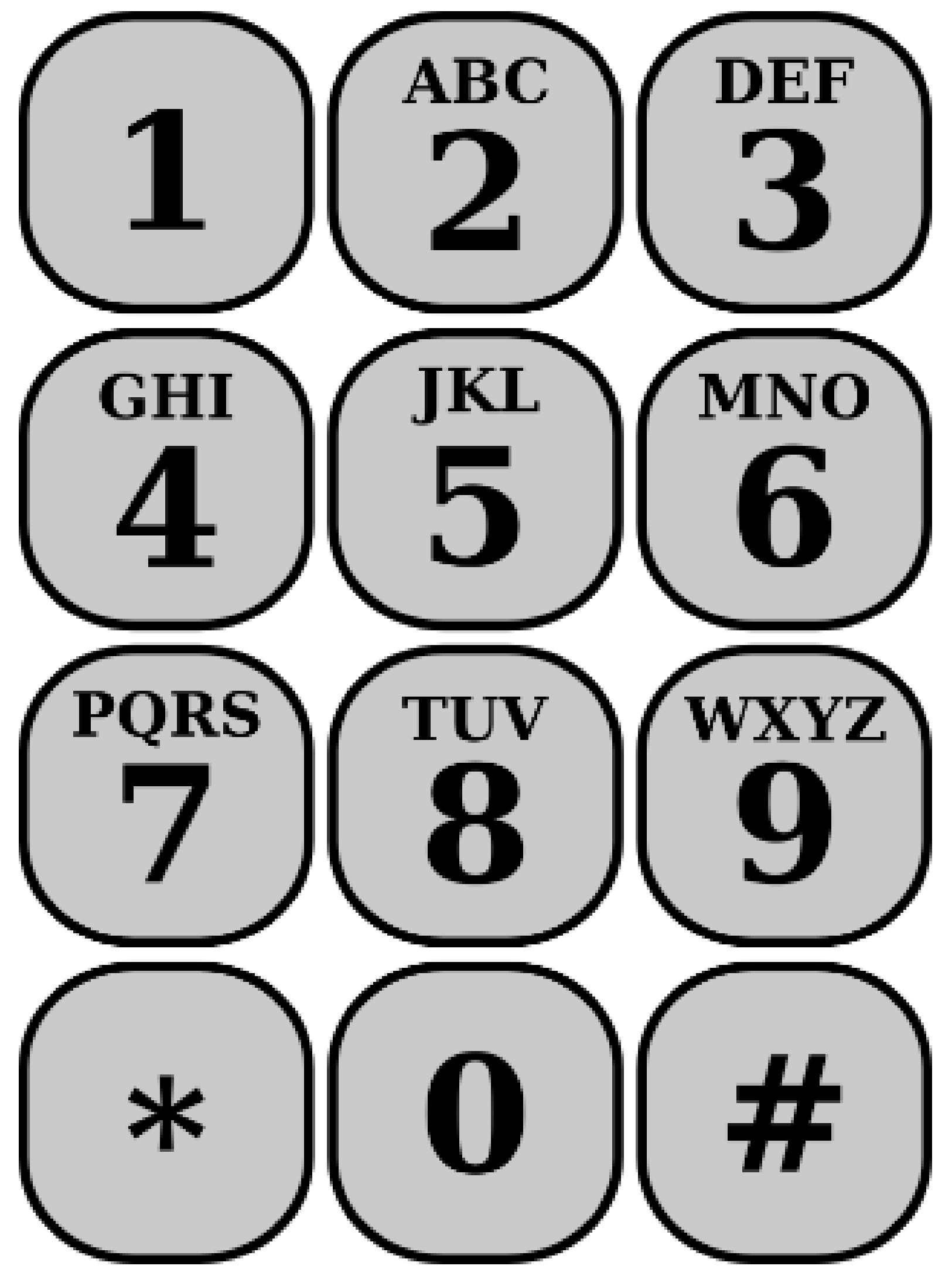
Given the array `{1, 0, 0, -5, -10, 32, 64, 128, 2, 9, 18}` the method would return 5. The elements 0, -5, 32, 64 and 9 are all followed directly by double their value.

Write a helper method that is sent the array and the current position or index in the array. That is where the actual recursion takes place. The public method shown above simply kicks off the recursion.

4. Phone Mnemonics

```
public ArrayList<String> listMnemonics(String number)
```

(From *Thinking Recursively* by Eric Roberts.) On some ancient telephone keypads the letters of the alphabet are mapped to various digits as shown in the picture below:



In order to make their phone numbers more memorable, service providers like to find numbers that spell out some word (called a **mnemonic**) appropriate to their business that makes that phone number easier to remember. (This was more valuable before the web and internet made it possible to look up the number for almost every place of business there is.) For example, the phone number for a recorded time-of-day message in some localities is 637-8687 (NERVOUS). Write a method listMnemonics that generate all possible letter combinations that correspond to a given number, represented as a string of digits. ***This is a recursive backtracking problem, but you are trying to find all the "solutions", not just one. In fact, you are trying to find all possible Strings, not just Strings that are actual "words".***

For example, if you call

```
ArrayList<String> result = recursiveObject.listMnemonics("623")
```

your method shall generate and return an ArrayList containing the following 27 possible letter combinations that correspond to that prefix:

MAD MBD MCD NAD NBD NCD OAD OBD OCD

MAE MBE MCE NAE NBE NCE OAE OBE OCE

MAF MBF MCF NAF NBF NCF OAF OBF OCF

Note, the order of the Strings in the ArrayList is unimportant.

Use the following helper method:

```
public String digitLetters(char ch)
```

This method returns the letters associated with a given digit.

Create a helper method that does most of the real work:

```
private void recursiveMnemonics(ArrayList<String> result, String mnemonicSoFar, String digitsLeft)
```

The ArrayList named result is used to store completed mnemonics. mnemonicSoFar is the String that has been built up so far based on various choices of letters for a digit. Initially it is the empty String. digitsLeft consists of the digits from the original number that have not been used so yet. Initially digitsLeft is the entire phone number. (Or collection of digits.)

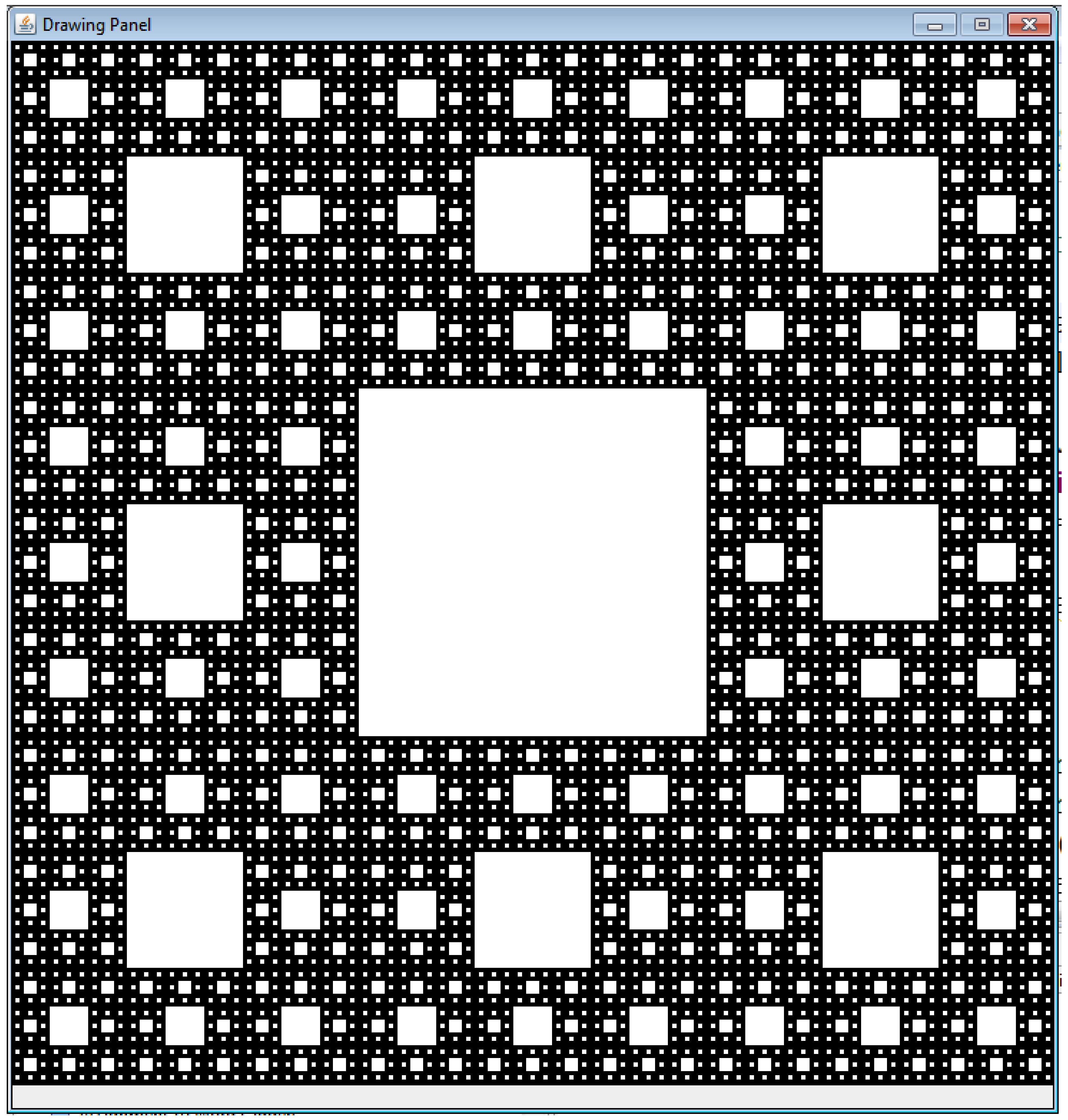
To kick off the recursion you call method recursiveMnemonics from listMnemonics:

```
ArrayList<String> result = new ArrayList<String>();
recursiveMnemonics(result, "", number); // when starting the mnemonic is the empty string
return result;
```

It should come as no surprise a website exists that does this kind of thing, but checks the results against a dictionary: http://www.phonespell.org/

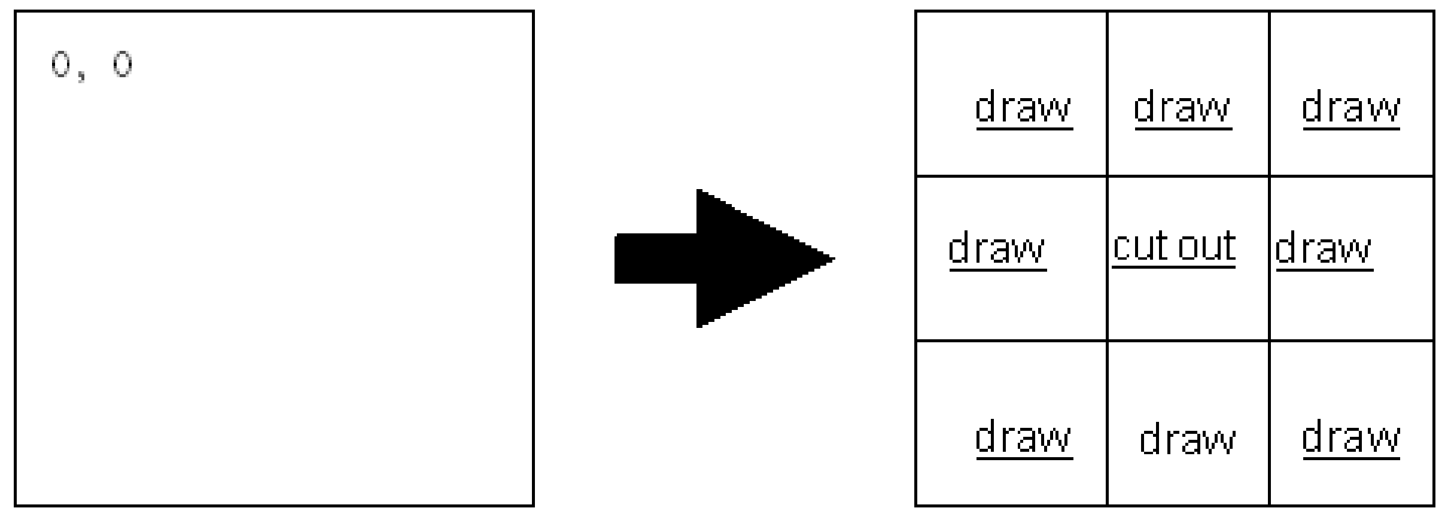
5. Sierpinksi Carpet

Create a method to draw a Sierpinski carpet. Here is an example:



The carpet is created via the following algorithm.

- if the length of the square is less than or equal to the limit value do nothing.
- else (the length of the square must be greater than the limit value) divide the square into a 3 by 3 grid which will contain 9 sub squares. "Cut out" the middle square and then create a Sierpenski carpet in the 8 remaining sub squares.



The middle square is "cut out" by drawing a white rectangle.

I have provided the starter method. You must supply the method to complete the carpet. The header for the method to actually draw the carpet is:

```
private static void drawSquares(Graphics g, int size, int limit, double x, double y)
```

Use the fillRect method from the Graphics class. Here is its signature.

```
fillRect(int x, int y, int width, int height)
    Fills the specified rectangle.
```

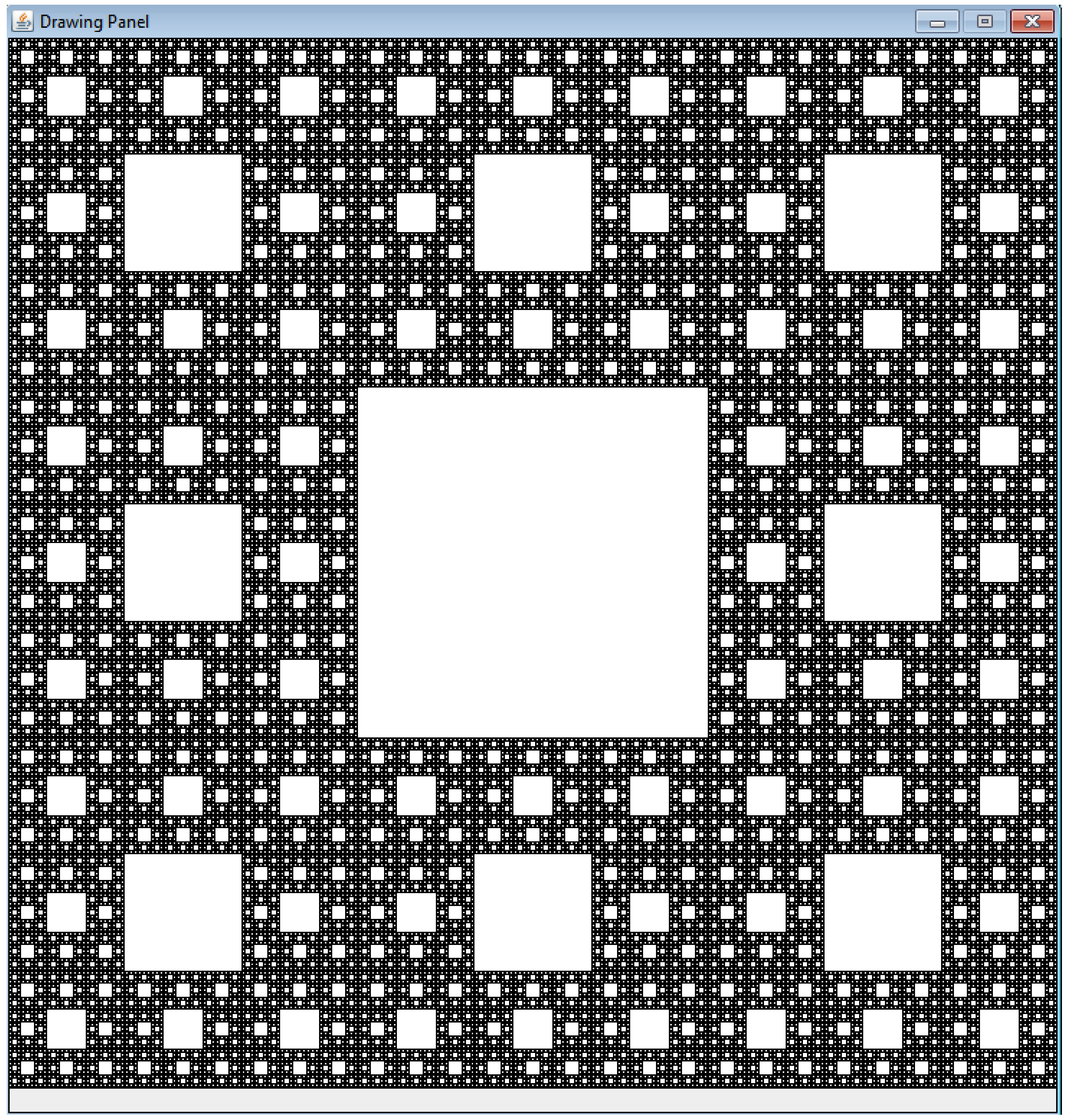
Recall point 0, 0 is at the top, left of the drawing panel and y increases as you go down (not up).

Complete the following method:

```
public static void drawSquares(Graphics g, int size, int limit, double x, double y){
```

Again, the x and y coordinates are passed as doubles. When you call the fillRect method cast the x and y coordinates to ints.

Here is an extreme Carpet with min size set to 1:



6. __Water flowing off a map.__

```
public boolean canFlowOffMap(int[][] map, int row, int col)
```

In this problem an area of land is represented as a 2D array of ints. The value at each element represents the elevation of that point on land. The higher the elevation, the higher the number, the lower the elevation the lower the number. So a flat plain at an elevation of 100 feet above sea level looks like this:

100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100

A "river" flowing through a 200-foot gorge looks like this. Cells that make up the river are shaded gray and yellow.

100	**99**	200	200	200	200	200	200	200	200
2000	**98**	2000	200	200	200	200	200	200	200
2000	**97**	96	200	200	200	200	200	200	200
200	2000	**95**	200	200	200	**85**	**84**	**83**	200
200	2000	**94**	**93**	**92**	200	**86**	2000	**82**	200

200	150	200	2000	**91**	200	**87**	2000	**81**	200
200	200	200	2000	**90**	**89**	**88**	2000	**80**	200
200	150	100	200	200	200	200	2000	**79**	200
200	200	200	200	200	200	200	2000	**78**	**77**
200	200	200	200	200	200	200	200	2000	**76**

Method `canFlowOffMap` has a parameter that represents a map of elevations as 2D array of ints. It also has parameters that represent the starting location of a drop of water on the map.

- The drop of water can move to one of the four adjacent cells (above, below, left, and right of the current cell) **if** the elevation in that cell is less than the elevation of the current cell.
- Water in any cell on the edge of the map can flow off the map.
- In other words, water in a cell on the border of the map is assumed to be able to flow off the map regardless of the cells around it.

In the first example, the plain, water in any cell not on the border of the 2d array, cannot flow to any other cell because all adjacent cells because all the elevations are the same. Water on any of the bordering cells could flow off the map. In the second example if a drop of water started at the cell at row 2, column 2 (the yellow cell with a value of 96) it could eventually flow off the map by following the river. (Each segment of the river is one foot lower than the previous.)

Recall that water on the border can also flow off. So, in the second example if we started at the cell in row 3, column 0 (the red cell with a value of 200) then we can flow off the map. The elevation in that cell is 200. The water can't flow to any of the other cells adjacent to it, but that doesn't matter because that cell is on the border and so if start there I can flow off the map.

If the starting cell is row 3, column 1 (the blue cell with a value of 200), the water can flow off the map. That cell has an elevation of 200. The water can't flow west or south because those cells are also at 200. But the water can flow north or east to the cells in the river. (the water falls off the bank into the river) and then off the map via the river.

Finally, the cell at row 6, column 1 (the green cell with a value of 200) can't flow off the map. It tries north and dead ends. It can't move east or west. It can try south, but eventually dead ends.

When thinking about the problem it may help to think that off the edge of border rows and columns is an infinitely deep canyon. So if you are on a border of the map, (row 0 , column 0, last row, or last column) then you can flow off the map. (Base case?)

`canFlowOffMap` returns true if a drop of water starting at the location specified by row, column can reach the edge of the map, false otherwise.

7. Creating fair teams.

```
public int minDifference(int numTeams, int[] abilities) {
```

In this question you are passed an int that represents the number of teams we want to form from a group of people. You are also passed an array of ints that represents the ability scores of all of the individuals we want to assign to teams. (It could be athletic ability, IQ, artistic ability, etc.) Abilities can be any integer value. A team's total ability is simply the sum of the ability scores of the people assigned to that team.

The goal of this method is to find the minimum possible difference between the team with the maximum total ability and the team with the minimum total ability. The constraints are everyone (represented by a value in the array named abilities) must be placed on a team and every team must have at least one member.

You must create the helper method. You must keep track of multiple things such as which element in the abilities array you are currently considering (or which people have already been assigned to a team), what the total ability score is for each team, as well as some way of ensuring each team has at least one member. (You could simply track the number of members per team.)

For example, if we have the following abilities: [1, 2, 3, 4, 5, 6, 7] and want to form three teams the min difference possible between the team with the maximum total ability and the team with the minimum total ability is 1. Your program does not need to show what the members of each team are but, in this example, the minimum possible difference is created by having 2 teams with a total ability of 9 and 1 team with 10. (min 9, max 10). There are multiple ways of doing this with these values one of which is:
Team 1: 1 + 3 + 6 = 10
Team 2: 2 + 7 = 9
Team 3: 4 + 5 = 9

In this problem there is a lot to do when you reach the base case. You must ensure every team has at least one person on it (or it is an invalid set up), and if it is a valid set up find the min and max team ability scores. If the setup is invalid you must return some value to indicate the set up was invalid or the largest possible difference. (Integer,MAX_VALUE perhaps?) **HINT: If not at the base case (all people assigned to teams), handle one person. The choices for the current person are the teams they could be assigned to.**

8. MazeSolver

```
public int canEscapeMaze(char[][] maze)
```

Complete a method to determine if it is possible to escape a maze while collecting all of the coins scattered throughout the maze.

The maze is represented with a rectangular 2d array of chars. Each element of the 2d array is a cell and the maze has 6 types of cells:

- Starting cell: The Starting cell is marked with an S. There is only one Starting cell in the maze. The starting cell becomes a green cell after we leave it.
- Impassable cells. Impassable cells are marked with an asterisk, *. We may not enter impassable cells.
- Green cells. Green cells are marked initially with a G. When we leave a green cell, it becomes a yellow cell as described below.
- Yellow cells. Yellow cells are marked initially with a Y. When we leave a yellow cell, it becomes an impassable cell as described above.
- Money cells. Gold cells are marked with a $. Each money cell contains a single gold coin we will pick up as we travel through the maze. There is no limit to the number of coins we can pick up. When we leave a money cell it becomes a yellow cell as described above.
- Exit cells. Exit cells are marked with an E. If we can reach an Exit cell we can escape the maze. Exit cells may appear any where in the maze, not just on the edges. There may me multiple exit cells in the maze. When we step into an Exit cell, we exit the maze due to being teleported out. In other words, we can't pass through an exit cell to another cell in the maze.

The 2d array of chars represents a bird's eye view of the maze. We can move up, down, left, or right, but not diagonally in the maze. Your method shall return 2 if we can reach an exit cell in the maze after collection all the gold coins present, a 1 if we can reach an exit in the maze but without collecting all the gold coins, or a 0 if it is not possible to exit the maze at all.

Consider the following simple maze:

G$SGE

We start at the cell marked with an S. We move to the left one spot to the cell with the gold coin. We then move back to the right to our starting cell, to the right once more to the regular cell and then right to the exit. Given this maze our method shall return 2.

Consider the following maze:

$Y$SGE

We can't escape this maze while collecting all the gold coins. We can move left to pick up the first coin. When we move left again to the yellow cell. When we leave that cell it becomes impassable. ($*YGGE) We can get to the gold coin on the far left, but we are stuck there and can't exit the maze. We can exit the maze without picking up all the coins so the method shall, in this case, return 1.
