FLORIDA SOUTHERN COLLEGE

**Data Structures**
***CSC 3280***

# <u>Fall 2020</u>

**Program 3: *FSC Recurse***
**Assigned: Thursday, October 1, 2020**
**Due: Friday, October 16, 2020 by 11:55 PM**

<u>Purpose:</u>
1. Make use of recursion to solve problems.

<u>Read Carefully:</u>

- This program is worth 7% of your final grade.

- **WARNING**: This is an individual assignment; you must solve it by yourself. Please review the definition of cheating in the syllabus, the FSC Honor Code, and the serious consequences of those found cheating.
    - **The FSC Honor Code pledge MUST be written as a comment at the top of your program**.

- When is the assignment due? The date is written very clearly above.
    - **Note:** once the clock becomes 11:55 PM, the submission will be closed! Therefore, in reality, you must submit by 11:54 and 59 seconds.

- <u>LATE SUBMISSION</u>: you are allowed to make a late submission according to the rules defined in the syllabus. Please see course syllabus for more information.

- **<u>Canvas Submission</u>**:
    - This assignment must be submitted online via Canvas.
    - You should submit a single ZIP file, which has **<u>ONLY</u>** your Java file inside it.
        - Do *not* submit your entire NetBeans project folder.
    - Within NetBeans, your project should be named as follows:
        - `FSCrecurse`
        - And when you make the project, NetBeans will automatically create a package with that same name. This is what we want.
    - Thus, you should have one Java file named as follows:
        - `FSCrecurse.java`

# Program 3: FSC Recurse

## The Problems

Below are a series of problems you need to solve using **recursive** methods. You will write a program that will read commands from an input file, with each command referring to one of the recursive problems to be executed. Each command will be followed (on the same line of input) by the respective parameters required for that specific problem.

## Four Problems to Solve Using Recursion:

### (1) sumThing

Of course, all of you are experts at summing numbers recursively. In fact, given two input values, *k* and *n*, you can easily write a recursive method that sums up the digits from *k* to *n*.

Example:
sumNumbers(4, 6) = 4 + 5 + 6 = 15

That is easy. And guess what? We gave that problem to you (and the answer) in the Lab! So if you are unsure of the code, go look up the answer in the lab PDF. For this problem, we want you to write a modified version of this method.

Given two parameters, *k* and *n*, write a recursive method, sumThing(), that sums up all Fibonacci numbers between (and including) the *k*th Fibonacci number and the *n*th Fibonacci number. Hopefully the following two examples will make this clear.

Example:
sumThing(4, 6) = fib(4) + fib(5) + fib(6) = 3 + 5 + 8 = 16

Example:
sumThing(1, 5) = fib(1) + fib(2) + fib(3) + fib(4) + fib(5) = 1 + 1 + 2 + 3 + 5 = 12

Your solution can ONLY include recursion; loops are not allowed.

Note: for this problem, fib(1) is 1 and fib(2) is also 1. The remaining Fibonacci numbers are calculated as per the equation given in class and on the course slides.

***Hint: you CAN use more than one recursive method if it helps you.

## (2) diamondsRforever

Your job is to write a recursive method that prints a large diamond made of small stars ("* "). The size of the diamond will be determined from the maximum width of the diamond, which is given in the input file and is guaranteed to be odd. Example for a diamond with max width of *n* = 7:

```
      *
    * * *
  * * * * *
* * * * * * *
  * * * * *
    * * *
      *
```

***Hint: you CAN use a for loop to draw one, single row of the diamond. But you need recursion to call the method on the "smaller" version of the problem.

## (3) CSflip

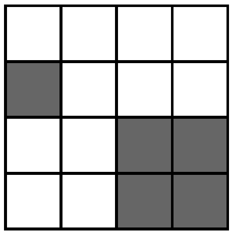
You have an index card with the letter C written on one side, and S on the other. You want to see ALL of the possible ways the card will land if you drop it *n* times. Write a recursive method that prints each session of dropping the cards with C's and S's. For example of you drop it 4 times in a given session, all possible ways to drop it are as follows:

```
CCCC
CCCS
CCSC
CCSS
CSCC
CSCS
CSSC
CSSS
SCCC
SCCS
SCSC
SCSS
SSCC
SSCS
SSSC
SSSS
```

## **(4) simplifyImage**

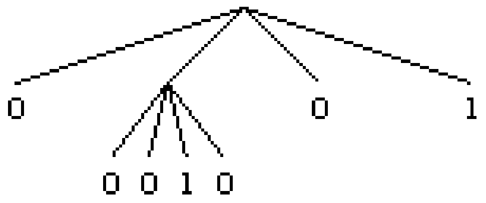
Pictures are often different shapes and sizes, and they are made up of millions of tiny pixels. Of course, each pixel can be a different color, resulting in a very beautiful image! For this problem, our pictures are not so beautiful. Instead, our pictures will only be square, and they will be monochromatic; this means each pixel has only two possible color options: (1) black or (0) white.

Consider this image on the right. This image has a total of 16 pixels. In fact, it is a 4 pixel x 4 pixel image. We can actually "code" (or encode) this picture with zeroes and ones. If every single pixel is black, we can say the color of the entire picture is 1 (black). And if every single pixel is white, we can say the color of the entire picture is 0 (white).



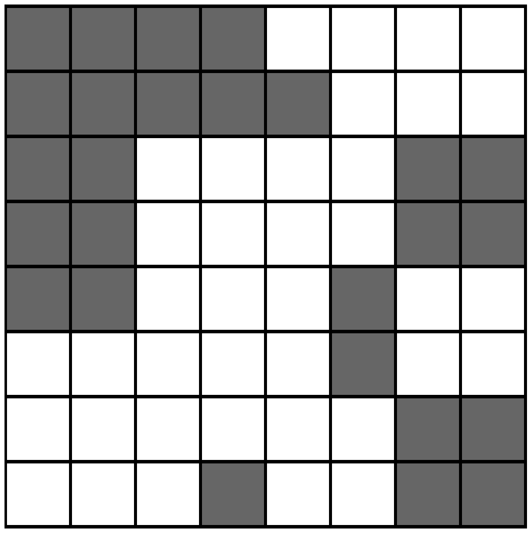
But what about with this example picture? Some pixels are white and some pixels are black. How can we encode this picture? Easy! We can represent this picture as a "tree"…think about the famous Family Tree, where you have great-great-great grandparents, and their children, and their children, and so on. Well, we can represent this image in a similar way, and the result is shown in the tree below. The top (or root) of the tree represents the entire image. Because the whole picture is not all the same color, we can divide the picture into four parts (or four quadrants):

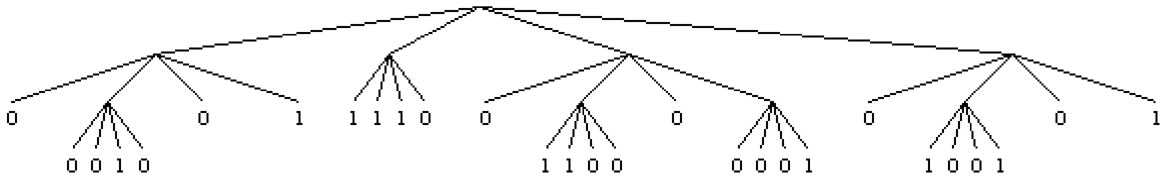
1. top-right quadrant
2. top-left quadrant
3. bottom-left quadrant
4. bottom-right quadrant



Each of those quadrants represents a "child" in this tree. Let us look at each of the quadrants in the exact order listed above. The top-right quadrant is all white; meaning, each of the four pixels is white. Therefore, we can represent this child with a 0. Now, look at the top-left quadrant. It is not all white or all black. Therefore, we must apply the same approach, recursively! Meaning, we must divide that region into four parts (four quadrants), and then we must examine each quadrant individually. So when we do this, each of the new four parts has only 1 pixel. Therefore, each part will certainly be only a 0 or a 1. The "coding" of the overall top-left quadrant will have the values 0, 0, 1, 0 (and this is in the same order of top-right, top-left, bottom-left, bottom-right). Finally, the bottom-left quadrant is all white; its value will be 0. And the bottom-right quadrant is all black; its value will be 1.

To be clear, here is a larger example of a square 64 pixel (8x8) image and the resulting tree:



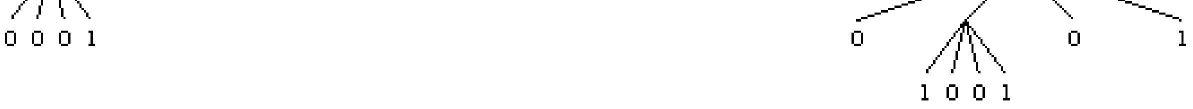


<u>Now to our problem…</u>

Does this tree perfectly model the colors of every pixel? Yes, it does! However, the coding of the tree gives too much detail. Instead of modeling the color of every single pixel perfectly, we have a new idea: if the color of a quadrant (or the whole tree) meets some threshold, such as 75%, for example, then we can estimate that the color of that quadrant is that color.

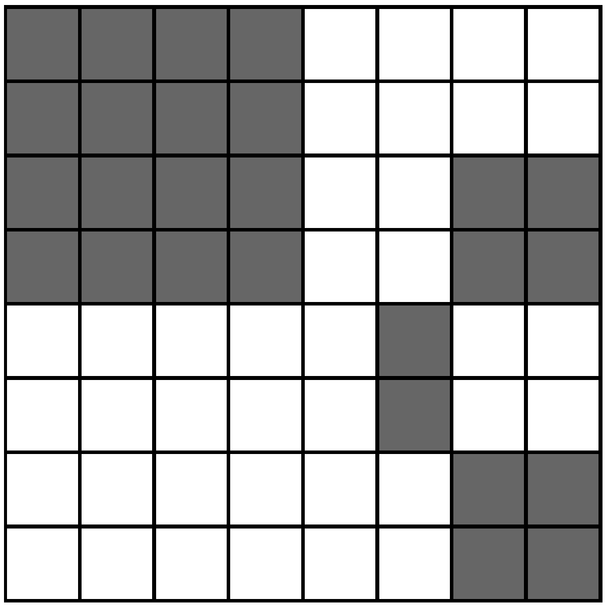
Let us use 75% as a threshold and now re-draw the resulting tree using the picture above.

Here is the result, and we will explain below:



Explanation of above tree:

The first question: are 75% of ALL the pixels white or black? If so, then the color of the entire tree will be 1 (black) or 0 (white), depending on which color representing 75% or more of the pixels. If no color represents at least 75% of the pixels of the tree, then this tree must be decomposed into, or divided into, the four quadrants as mentioned previously. So we start with the top-right quadrant, which has 16 pixels. How many are black? 5. How many are white? 11. Therefore, neither color has a 75% majority. As a result, in order to correctly represent that top-right quadrant, we must again recursively decompose it, or divide it, into four quadrants using the same process. Now we have four new quadrants, each only 4 pixels (2x2). Starting with the top right 2x2 quadrant, look at the colors. All are white! Therefore, we can represent that quadrant with a 0. Now look at the top-left quadrant. We see three white pixels and one black pixel. The white pixels are 75% of the total (3 out of 4 pixels is 75%). Therefore, because the threshold of 75% is reached, we can represent that top-left quadrant with a 0 (white). Next, the bottom-left quadrant has four pixels, all white. We can therefore represent this quadrant with a 0 (white). And finally, the bottom-right quadrant has four pixels, all black. Therefore, we can represent the bottom quadrant as a 1 (black).



Using this new, less-detailed encoding scheme, we can encode the top-right quadrant of 16 pixels as 0, 0, 0, 1. The other three 4x4 pixel (16 pixel) quadrants can be encoded in the same way, resulting in the tree shown on the previous page. And if we now “draw” the image based on this less-detailed tree encoding, we would get the following result shown to the right.

Your job:
Given an image width/height in pixels, a threshold (as an int), and the actual colors of the pixels of the image, your job is to write a recursive method, simplifyImage(), that will determine the new simplified version of the image that results from this less-detailed encoding.

| **Sample Input** | **Expected Output** |
|---|---|
| 4 75<br>1101<br>1111<br>0111<br>0011 | 1111<br>1111<br>1111<br>1111 |
| **Sample Input**<br>8 75<br>11111000<br>11110000<br>11000011<br>11000011<br>11000100<br>00000100<br>00010011<br>00010011 | **Expected Output**<br>11110000<br>11110000<br>11110011<br>11110011<br>00000100<br>00000100<br>00000011<br>00000011 |

**Implementation**

Each recursive method MUST have a wrapper method enclosing it where you will do input/output file processing as well as calling the actual recursive method.

From Wikipedia:

*"A wrapper function is a function in a computer program whose main purpose is to call a second function".*

As an example, here is one of the wrapper methods you will use:

```
void sumThing(Scanner input, PrintWriter output);
```

This would be the wrapper method for one of the recursive problems, called **sumThing()** (described, in detail, above). These wrapper methods will simply do three things:

1. deal with the processing of the input file
2. most importantly, the wrapper method will make the initial call to your recursive method that will actually solve the problem.
3. Finally, the wrapper methods will print to the output file. Note: printing to the output file may also occur in the actual recursive methods.

Of course, at your own discretion (meaning, your own choice), you can make additional methods for your recursive methods to use, such as methods to swap values in an array, take the sum of an array of values, printing methods, etc.

> **<u>***HINT</u>**: We have posted an example program with a small recursive method that uses a wrapper method as described above. See the recursion example programs on Blackboard!

**<u>Wrapper Methods DisplayPasswords</u>**
As mentioned, you MUST use the following three wrapper methods <u>EXACTLY</u> as shown:

```
public static void sumThing(Scanner input, PrintWriter output);
public static void diamondsRforever(Scanner input, PrintWriter output);
public static void csFlip(Scanner input, PrintWriter output);
public static void simplifyImage(Scanner input, PrintWriter output);
```

**This means you will make the Scanner variable and PrintWriter variable, inside main, and then you will send those variables to the three wrapper methods.*

**<u>Input File Specifications</u>**

**You will read in input from a file, "FSCrecurse.in".** Have this AUTOMATED. Do <u>not</u> ask the user to the name of the input file. You should read in this automatically. The first line of the input file will have one positive integer, representing the number of commands (lines) inside the input file.

Each of the following n lines will have a command, and each command will be followed by appropriate data as described below (and this data will be on the same line as the command).

The commands (for the three recursive methods), and their relevant data, are described below:

sumThing – This command will be followed on the same line by two positive integers, *k* and *n*, as described above.

diamondsRforever – This command will be followed on the same line by a single positive integer, *n*, as described above.

CSflip – This command will be followed by one positive integer, *n*, as described above.

simplifyImage – This command will be followed on the same line by two positive integers, size and threshold. Then, exactly size lines will follow representing the rows of the image. Each row will have exactly size number of integers (0 or 1), representing the pixels for that row.

## Output Format

Your program must output to a file, called **"FSCrecurse.out". You must follow the program specifications exactly.** Refer to sample output file for exact formatting specifications.

*****WARNING*****

Your program MUST adhere to the EXACT format shown in the sample output file (spacing capitalization, use of dollar signs, periods, punctuation, etc). For grading, we will use very large input files, resulting in very large output files. As such, we will use text comparison programs to compare your output to the correct output. If, for example, you have two spaces between in the output when there should be only one space, this will show up as an error even through you may have the program correct. You WILL get points off if this is the case, which is why this is being explained in detail. Minimum deduction will be 10% of the grade, as the graders will be forced to go to text editing of your program in order to give you an accurate grade. Again, your output MUST ADHERE EXACTLY to the sample output.

## Grading Details

Your program will be graded upon the following criteria:

1) Adhering to the implementation specifications listed on this write-up.
2) Your algorithmic design.
3) Correctness.
4) **Use of Recursion. If your program does not use recursion, you will get a zero.**
5) The frequency and utility of the comments in the code, as well as the use of white space for easy readability. (We're not kidding here. If your code is poorly commented and spaced and works perfectly, you could earn as low as 80-85% on it.)
***Please RE-READ the above note on comments.
6) Compatibility to the **newest version** of NetBeans. (If your program does not compile in NetBeans, you will get a large deduction from your grade.)
7) Your program should include a header comment with the following information: your name, **email**, date, AND HONOR CODE.
8) Your output MUST adhere to the EXACT output format shown in the sample output file.

## Deliverables

You should submit a single java file:

1. **FSCrecurse.java**

***Please do not use packages for this program.

**NOTE: your name, ID, and HONOR CODE should be included as comments in all files!**

**Suggestion: START EARLY!**