REPORT LAB 5

Problem 1:

**1. Base Cases of function puzzle:**

The base cases in the puzzle function are:

* base == limit: This condition checks if the base is equal to the limit. If it is, the function returns 1.

**2. Recursive Case(s) of function puzzle:**

The recursive case in the puzzle function is:

* *else*: This block executes when neither of the base cases is met (i.e., base is not greater than limit and not equal to limit). In this case, the function performs the recursive call: return base \* puzzle(base + 1, limit). Here, it returns the product of base and the result of calling puzzle with base + 1 as the new base and limit remaining the same.

A computer screen shot of code

Description automatically generated

**3. Output for Function Calls:**

a. System.out.print(puzzle(14,10));

In this call, base is 14 and limit is 10. Since 14 is greater than 10, the base case base > limit applies, and the function returns -1.

b. System.out.print(puzzle(4,7));

1. The function starts with base = 4 and limit = 7.
2. Since base (4) is less than or equal to limit (7), the recursive case applies.
3. The function calls itself recursively with base + 1 = 5 and limit = 7.
4. This process continues until base becomes 7.
5. At base = 7 (which is still less than or equal to limit), the function cannot call itself recursively anymore because base would be greater than limit.
6. Since base == limit, the condition for the base case (that we previously assumed to be incorrect) might not actually be a base case in this context. Instead, the calculation continues without a recursive call.
7. The function returns 1 (assuming this is the intended behavior for base == limit).

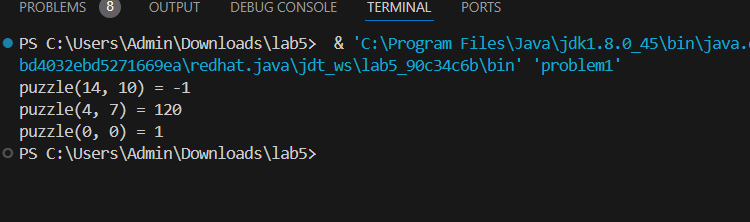
Following the recursive calls back up the chain, the product is calculated: 4 (from the initial call) \* 5 (from the second call) \* 6 (from the third call) \* 7 (from the fourth call) \* 1 (assumed base case behavior) = 120.

c. System.out.print(puzzle(0,0));

In this case, base and limit are both 0. The base case base == limit applies directly, and the function returns 1.

A screenshot of a computer program

Description automatically generated



Problem 2:

In this problem, we recursively evaluate the sum: sum = 1 + 1/2 + 1/3 +...+1/n, n > 1.

When n = 1, sum = 1

If n> 1, the method adds 1/n to the sum of the previous terms. This is done recursively by calling sum(n - 1) until the base case is reached.

A screenshot of a computer program

Description automatically generated

* Scanner Input: value of n
* Input Validation: checks if the entered value if n >1 or not . If n <= 1, it prints a message asking the user to enter a valid value greater than 1.
* Recursive Sum Calculation:

Output if n <= 1:

A computer screen shot of a program

Description automatically generated

Output if n >1:

A screen shot of a computer

Description automatically generated

Problem 4:

findMin function: find the minimum of the array

* If the array has only one element n => min =n
* If n > 1, the function calls itself recursively to find the minimum value in the remaining elements (the array size decreases by one with each call). Afterward, it compares the current element a[n-1] with the minimum value from the rest of the array and returns the smaller of the two.

findSum function: calculate sum of the elements in the array

* If the array has only one element n => sum = n
* If n> 1, the function calls itself recursively to sum the remaining elements (the array size decreases by one with each call). It then adds the current element a[n-1] to the sum of the remaining elements.

A screenshot of a computer program

Description automatically generated

We input the size of the array (n), after that input the elements so that we can get sum and minimum at the end.

A black screen with white text

Description automatically generated

Problem 6:

To calculate the Greatest Common Divisor (GCD) of two numbers:

* If b == 0, then the GCD is a (gcd(a, 0) = a).
* Otherwise, gcd(a, b) = gcd(b, a % b) where % is the modulo operator.

A screen shot of a computer program

Description automatically generated

gcd(int p, int q): This method takes two integers p and q, returns GCD.

* If q == 0, the GCD is p. This is because any number divided by 0 gives the other number as the GCD.
* If q != 0, we call gcd(q, p % q), which reduces the problem size. The % operator finds the remainder when p is divided by q. The process repeats until q becomes 0, at which point p will be the GCD.

A screen shot of a computer

Description automatically generated

First call: gcd(56, 98)

p = 56, q = 98. Since q != 0, call gcd(98, 56 % 98) → gcd(98, 56).

Second call: gcd(98, 56)

p = 98, q = 56. Since q != 0, call gcd(56, 98 % 56) =>gcd(56, 42).

Third call: gcd(56, 42)

p = 56, q = 42. Since q != 0, call gcd(42, 56 % 42) => gcd(42, 14).

Fourth call: gcd(42, 14)

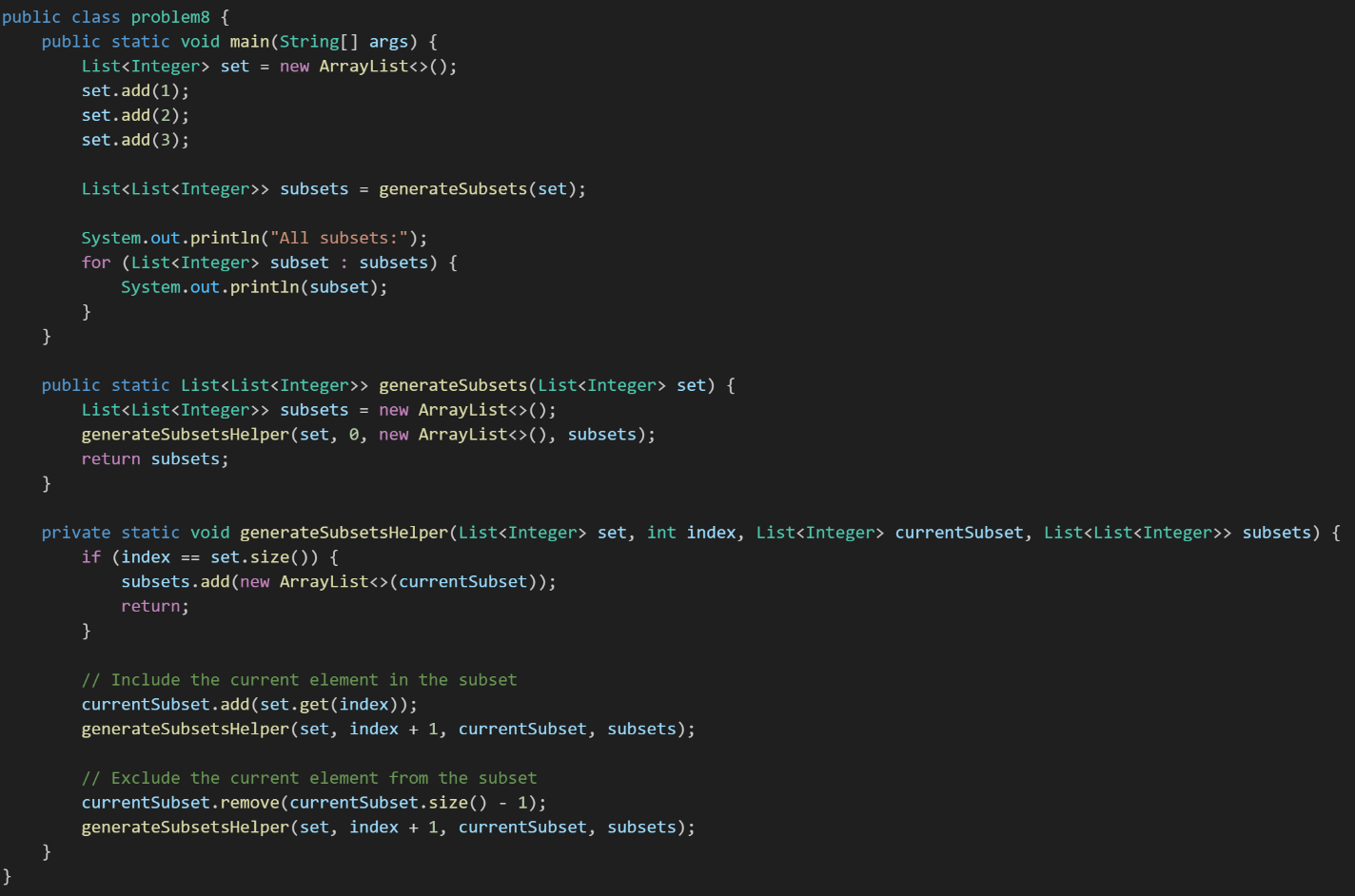
p = 42, q = 14. Since q != 0, call gcd(14, 42 % 14) => gcd(14, 0).

Base case: gcd(14, 0). Since q == 0, return p, which is 14

Problem 8:

In this problem, he function takes a set as input and returns a list of all subsets of the given set.

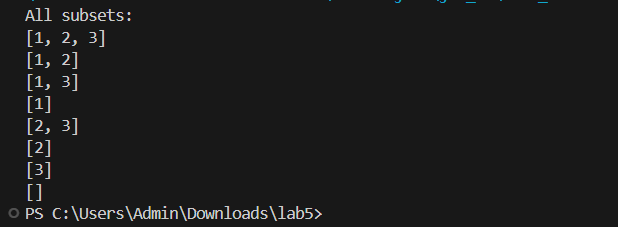
* If the set is empty, the function returns a list containing the empty set.
* If the set is not empty, the function generates subsets by either including or excluding the first element and recursively generating subsets for the rest of the elements.
* **Combination**: It combines the subsets generated by including and excluding the first element to form all possible subsets of the given set.



This recursive approach effectively explores all possible combinations of elements in the set, resulting in the generation of all subsets.

In this situation, we have the **main** method that initializes a list **set** with elements **{1, 2, 3}**, the **set** list as input and the output is prints all subsets generated.

Output:



Sierpinski triangle fractal triangle:

Idea: Dividing an equilateral triangle into smaller triangles. The Sierpinski Triangle is built by recursively subdividing an equilateral triangle into 4 smaller triangles and removing the central triangle.

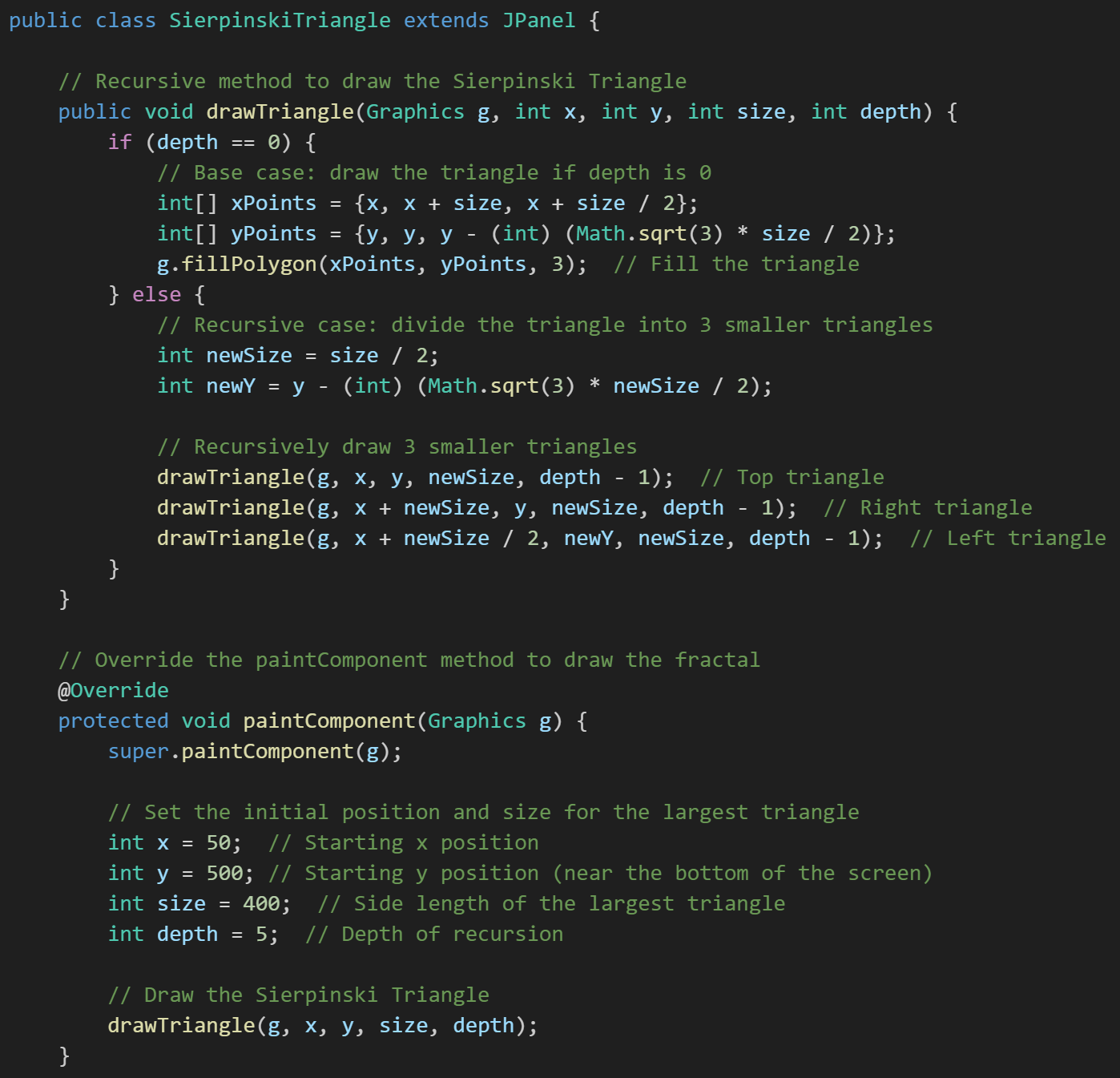
* Start with an equilateral triangle and recursively subdivide it into smaller triangles.
* In each recursive step, draw three smaller triangles instead of the original one.
* The recursion stops when the size of the triangle is small enough to be drawn.

Implement:

If the side length of the triangle is small enough, draw the triangle.

For a larger triangle:

* Divide it into 3 smaller triangles by calculating the midpoints of the sides.
* Recursively apply the same process to each of the 3 smaller triangles.



drawTriangle method:

* This is the recursive method that draws the Sierpinski Triangle.
* The method takes 5 parameters:
  + Graphics g: The graphics object for drawing.
  + x, y: The coordinates of the top vertex of the triangle.
  + size: The side length of the triangle.
  + depth: The depth of recursion. When depth == 0, we draw the triangle.
* If depth is greater than 0, the triangle is divided into 3 smaller triangles, and the method is called recursively on each smaller triangle.

paintComponent method:

* This method is overridden from JPanel to handle the drawing of the fractal.
* It sets the initial values for the triangle's position, size, and depth of recursion.
* It then calls drawTriangle to draw the Sierpinski Triangle fractal.

A screen shot of a computer code

Description automatically generated

main:

* Creates a JFrame and adds the SierpinskiTriangle panel to it.
* Sets the size of the window and makes it visible.

The initial triangle is drawn at the bottom of the window, with a size of 400 pixels and a depth of 5 for recursion.

The drawTriangle method keeps dividing the triangle into smaller triangles and drawing them.

As the recursion depth increases, the triangles become smaller and the fractal pattern emerges.

Output:

**A screenshot of a computer screen

Description automatically generated**