Chapter Summary

Recursion is an algorithmic technique in which a method calls itself. A method that uses recursion is called a recursive method.

Recursive methods include two cases: a base case that the method can solve directly without recursion, and a recursive case in which the method reduces a problem into a simpler problem of the same kind using a recursive call.

Recursive method calls work internally by storing information about each call into a structure called a call stack. When the method calls itself, information about the call is placed on top of the stack. When a method call finishes executing, its information is removed from the stack and the program returns to the call underneath.

A recursive method without a base case, or one in which the recursive case doesn't properly transition into the base case, can lead to infinite recursion.

A helper method is written to help solve a subtask of an overall problem. Recursive helper methods often have parameters in addition to the ones passed to the overall recursive method that calls them, to allow them to more easily implement the overall recursive solution.

Recursion can be used to draw graphical figures in complex patterns, including fractal images. Fractals are images that are recursively self-similar, and they are often referred to as "infinitely complex."

Self-Check Problems

Section 12.1: Thinking Recursively

1. What is recursion? How does a recursive method differ from a standard iterative method?

A function that calls itself.

2. What are base cases and recursive cases? Why does a recursive method need to have both?

Base cases is a non-recursive case in a recursive function, a recursive case results in recursion

3. Consider the following method:

```
public static void mystery1(int n) {
    if (n <= 1) {
        System.out.print(n);
    } else {
        mystery1(n / 2);
        System.out.print(", " + n);
    }
}
```

For each of the following calls, indicate the output that is produced by the method:

a. `mystery1(1);` 1

b. `mystery1(2);` 1, 2

c. `mystery1(3);` 1, 3

d. mystery1(4); 1, 2, 4

c. mystery1(16); 1, 2, 4, 8, 16

f. mystery1(30); 1, 3, 7, 15, 30

g. mystery1(100); 1, 3, 6, 12 25, 50, 100

4. Consider the following method:

```
public static void mystery2(int n) {
    if (n > 100) {
        System.out.print(n);
    } else {
        mystery2(2 * n);
        System.out.print(", " + n);
    }
}
```

For each of the following calls, indicate the output that is produced by the method:

a. mystery2(113); 113

b. mystery2(70); 140, 70

c. mystery2(42); 168, 84, 42

d. mystery2(30); 120, 60, 30

e. mystery2(10); 160, 80, 40, 20, 10

5. Consider the following method:

```
public static void mystery3(int n) {
    if (n <= 0) {
        System.out.print("*");
    } else if (n % 2 == 0) {
        System.out.print("(");
        mystery3(n – 1);
        System.out.print(")");
    } else {
        System.out.print("[");
        mystery3(n – 1);
        System.out.print("]");
    }
}
```

For each of the following calls, indicate the output that is produced by the method:

a. mystery3(0); *

b. mystery3(1); [*]

c. mystery3(2); ([*])

d. mystery3(4); ([([*])])

c. mystery3(5); [([([*])])]

6. Consider the following method:

```
public void mysteryXY(int x, int y) {
    if (y == 1) {
        System.out.print(x);
    } else {
        System.out.print(x * y + ", ");
        mysteryXY(x, y - 1);
        System.out.print(", " + x * y);
    }
}
```

For each of the following calls, indicate the output that is produced by the method:

a. `mysteryXY(4, 1);` 4

b. `mysteryXY(4, 2);` 8, 4, 8

c. `mysteryXY(8, 2);` 16, 8, 16

d. `mysteryXY(4, 3);` 12, 8, 4, 8, 12

e. `mysteryXY(3, 4);` 12, 9, 6, 3, 6, 9, 12

7. Convert the following iterative method into a recursive method:

```
// Prints each character of the string reversed twice.
// doubleReverse("hello") prints oolllleehh
public static void doubleReverse(String s) {
    for (int i = s.length() – 1; i >= 0; i--) {
        System.out.print(s.charAt(i));
        System.out.print(s.charAt(i));
    }
}
```

```
public static void doubleReverseRecursive (String s) {
    if(s.length()>1) doubleReverseRecursive(s.substring(1));
    SOP(""+s.charAt(0) +s.charAt(0));
}
```

Section 12.2: A Better Example of Recursion

8. What is a call stack, and how does it relate to recursion?

The methods that were called ~~that~~ eventually called the current one. Recursion has a method calling itself, adding to the callstack

9. What would be the effect if the code for the reverse method were changed to the following?

```
public static void reverse(Scanner input) {
    if (input.hasNextLine()) {
        // recursive case (nonempty file)
        String line = input.nextLine();
        System.out.println(line);       // swapped order
        reverse(input);                 // swapped order
    }
}
```

It prints the line in ~~normal~~ original order

10. What would be the effect if the code for the reverse method were changed to the following?

```
public static void reverse(Scanner input) {
    if (input.hasNextLine()) {
```

It is printed in the original order

```
        // recursive case (nonempty file)
        reverse(input);                         // moved this line
        String line = input.nextLine();
        System.out.println(line);
    }
}
```

Section 12.3: Recursive Functions and Data

11. The following method is an attempt to write a recursive pow method to compute exponents. What is wrong with the code? How can it be fixed? There is no base case.

```
public static int pow(int x, int y) {
    return x * pow(x, y - 1);
}
```

12. What are the differences between the two versions of the pow method shown in Section 12.3? What advantage does the second version have over the first version? Are both versions recursive?
There are many versions, each is recursive. One adds error checks, another adds efficiency

13. Consider the following method:

```
public static int mystery4(int x, int y) { // return x % y;
    if (x < y) {
        return x;
    } else {
        return mystery4(x – y, y);
    }
}
```

For each of the following calls, indicate the value that is returned:

a. `mystery4(6, 13)` 6
b. `mystery4(14, 10)` 4
c. `mystery4(37, 10)` 7
d. `mystery4(8, 2)` 0
e. `mystery4(50, 7)` 1

14. Consider the following method:

```
public static int mystery5(int x, int y) {
    if (x < 0) {
        return –mystery5(–x, y);
    } else if (y < 0) {
        return –mystery5(x, –y);
    } else if (x == 0 && y == 0) {
        return 0;
    } else {
        return 100 * mystery5(x / 10, y / 10) + 10 * (x % 10) + y % 10;
    }
}
```

For each of the following calls, indicate the value that is returned:

a. `mystery5(5, 7)` 57

b. `mystery5(12, 9)` 200 + 20 + 9 = 229

c. `mystery5(-7, 4)` -74

d. `mystery5(-23, -48)` 20 + 4 + 30 + 8 = 62

e. `mystery5(128, 343)` = 120

15. Consider the following method:

```
public static int mystery6(int n, int k) {
    if (k == 0 || k == n) {
        return 1;
    } else if (k > n) {
        return 0;
    } else {
        return mystery6(n - 1, k - 1) + mystery6(n - 1, k);
    }
}
```

For each of the following calls, indicate the value that is returned:

a. `mystery6(7, 1)` 7 6 5 4 3 2 1
1 1 1 1 1 1 1 = 7

b. `mystery6(4, 2)` 6

c. `mystery6(4, 3)` 4

d. `mystery6(5, 3)` 10

e. `mystery6(5, 4)` 5

16. Convert the following iterative method into a recursive method:

```
// Returns n!, such as 5! = 1*2*3*4*5
public static int factorial(int n) {
    int product = 1;
    for (int i = 1; i <= n; i++) {
        product *= i;
    }
    return product;
}
```

```
public static int factoral (int n) {
  if (n < 0) throw new IllegalArgumentException("Factorial
         is undefined for negative numbers");
  else if (n <= 1) return 1;
  else return n * factoral(n-1);
```

17. The following method has a bug that leads to infinite recursion. What correction fixes the code?

```
// Adds the digits of the given number.
// Example: digitSum(3456) returns 3+4+5+6 = 18
public static int digitSum(int n) {
    if (n ~~>~~ < 10) {
        // base case (small number)
        return n;
    } else {
        // recursive case (large number)
        return n % 10 + digitSum(n / 10);
    }
}
```

18. Sometimes the parameters that a client would like to pass to a method don't match the parameters that are best for writing a recursive solution to the problem. What should a programmer do to resolve this issue?

Overloading or helper method

19. The Fibonacci sequence is a sequence of numbers in which the first two numbers are 1 and each subsequent number is the sum of the previous two Fibonacci numbers. The sequence is 1, 1, 2, 3, 5, 8, 13, 21, 34, and so on. The following is a correct, but inefficient, method to compute the *n*th Fibonacci number:

```
public static int fibonacci(int n) {
    if (n <= 2) {
        return 1;
    } else {
        return fib(n – 1) + fib(n - 2);
    }
}
```

```
private static final ArrayList<Integer> fibonacci =
    new ArrayList<>(Arrays.asList(0, 1));
public static int fibonacci(int n) {
    if(n < fibonacci.size()) return fibonacci.get(n);
    else {
        final int r = Math.addExact(fibonacci(n-1), fibonacci.get(n-2));
        fibonacci.add(r);
        return r;
    }
}
```

The code shown runs very slowly for even relatively small values of n; it can take minutes or hours to compute even the 40th or 50th Fibonacci number. The code is inefficient because it makes too many recursive calls. It ends up recomputing each Fibonacci number many times. Write a new version of this method that is still recursive and has the same header but is more efficient. Do this by creating a helper method that accepts additional parameters, such as previous Fibonacci numbers, that you can carry through and modify during each recursive call.

My answer still won't find the 50th fibonacci number as it will throw an ArithmeticException: because of the Integer overflow at the 47th.

Section 12.4: Recursive Graphics

20. What is a fractal image? How does recursive programming help to draw fractals?

A fractal is visual recursion, a shape that contains itself

21. Write Java code to create and draw a regular hexagon (a type of polygon).

Requires Supplement 3G

Section 12.5: Recursive Backtracking

22. Why is recursion an effective way to implement a backtracking algorithm?

Both are [illegible] as easy as ~~and~~ iterative a recursive function in a for loop

23. What is a decision tree? How are decision trees important for backtracking?

A tree diagram that represents the path taken by a recursive backtracking function

24. Draw the decision tree that would have resulted for Figure 12.9 if the backtracking solution had explored NE first instead of last in the recursive explore method. (Hint: the tree changes at every level.)

It would still be the same ans, just some stuff have [illegible].

25. The original North/East backtracking solution printed the following ways of traveling to (1, 2) in this order. In what order would they be printed if the solution had explored NE first instead of last?

```
3 moves: N N E
4 moves: N E N
2 moves: N NE
5 moves: E N N
1 moves: NE N
```

26. Figure 12.12 shows only part of the decision tree for the first two levels. How many entries are there at the second level of the full tree? How many are at level 3 of the full tree?

2nd level $8^2 = 64$

3rd Level $8^3 = 512$

Checking every square $O(\frac{n^2!}{(n^2-n)!})$
1 queen per colum $O(n^n)$

27. If our 8 Queens algorithm tried every possible square on the board for placing each queen, how many entries are there at the 8th and final level of the full tree? What does our algorithm do to avoid having to explore so many possibilities?
64!/56! ≈ 1.78 × 10^{14} possibilites. Our algorithm solves this by only placing one queen in each column, solving it 8^8 = 16,777,215 posabilitys

28. The 8 Queens `explore` method stops once it finds one solution to the problem. What part of the code causes the algorithm to stop once it finds a solution? How could the code be modified so that it would find and output every solution to the problem? The base case & the return true if recursion is true. Add a string parameter containing the current path, and in the base case, print it. Also, remove the return from the loop and instead store it in a var outside and return the

Exercises

1. Write a recursive method called `starString` that accepts an integer as a parameter and prints to the console a string of stars (asterisks) that is 2^n (i.e., 2 to the n^{th} power) long. For example,

- `starString(0)` should print `*` (because $2^0 == 1$)
- `starString(1)` should print `**` (because $2^1 == 2$)
- `starString(2)` should print `****` (because $2^2 == 4$)
- `starString(3)` should print `********` (because $2^3 == 8$)
- `starString(4)` should print `****************` (because $2^4 == 16$)

The method should throw an `IllegalArgumentException` if passed a value less than `0`.

2. Write a method called `writeNums` that takes an integer `n` as a parameter and prints to the console the first n integers starting with 1 in sequential order, separated by commas. For example, consider the following calls:

```
writeNums(5);
System.out.println(); // to complete the line of output
writeNums(12);
System.out.println(); // to complete the line of output
```

These calls should produce the following output:

```
1, 2, 3, 4, 5
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

Your method should throw an `IllegalArgumentException` if passed a value less than `1`.

3. Write a method called `writeSequence` that accepts an integer n as a parameter and prints to the console a symmetric sequence of n numbers composed of descending integers that ends in 1, followed by a sequence of ascending integers that begins with 1. The following table indicates the output that should be produced for various values of n:

```
Method call              Output produced
-----------------------------------------
writeSequence(1);        1
writeSequence(2);        1 1
writeSequence(3);        2 1 2
writeSequence(4);        2 1 1 2
writeSequence(5);        3 2 1 2 3
writeSequence(6);        3 2 1 1 2 3
writeSequence(7);        4 3 2 1 2 3 4
writeSequence(8);        4 3 2 1 1 2 3 4
writeSequence(9);        5 4 3 2 1 2 3 4 5
writeSequence(10);       5 4 3 2 1 1 2 3 4 5
```