KEEP CALM AND LEARN JAVA

I hope this book will be interesting for you. Good luck!

MPS4MDO

Contents

1. IF-ELSE 8
  1.1. Mystery #1 8
  1.2. Mystery #2 8
  1.3. Percentage Grade 9
  1.4. Date Is Before 9
  1.5. Compute Tax 10
2. LOOPS 11
  2.1. Stars Print 11
  2.2. Number Loops #1 11
  2.3. Number Loops #2 11
  2.4. Number Loops #3 11
  2.5. Number Loops #4 12
  2.6. Mystery #1 12
  2.7. Mystery #2 12
  2.8. Mystery #3 13
  2.9. Mystery #4 13
  2.10. Pow 13
  2.11. Sentinel Sum 14
  2.12. Biggest And Smallest 14
  2.13. Compute Sum Of Digits 14
  2.14. Range Of Numbers 14
  2.15. Number Square 15
  2.16. Fizz Buzz 15
  2.17. Fibonacci Constant 15
  2.18. Roll Two Dice 16
  2.19. Piglet 16
3. PARAMETERS AND RETURN 18
  3.1. Box Of Stars 18
  3.2. Triangle 18
  3.3. Print Pay 18
  3.4. Count Digits 18
  3.5. Mystery #1 19
  3.6. BMI 19
  3.7. Mystery #2 20

3.8. Show Twos 20
3.9. Mystery #3 21
3.10. Mystery #4 21
3.11. Mystery #5 22
3.12. Mystery #6 23
3.13. Mystery #7 23
3.14. Mystery #8 24
3.15. Mystery #9 25
3.16. Mystery #10 25
3.17. Mystery #11 26
3.18. Mystery #12 27
3.19. Mystery #13 27
3.20. Circle Area 28
3.21. Average Of 3 28
3.22. Compute Distance 28
3.23. Get Days In Month 29
3.24. Factorial 29
3.25. Roulette 29
3.26. Sum Of Range 30
3.27. Get Last Digit 30
3.28. Get First Digit 30
3.29. Sum Of Digits 30
3.30. Decimal To Binary 30
3.31. Binary To Decimal 30
3.32. Factor Count 31
3.33. Random Over 31
3.34. Is Multiple 31
3.35. Three Consecutive 31
3.36. Is Prime Number 31
3.37. Coin Flip 31
3.38. Random Walk 32
3.39. Play Roulette 33
3.40. Fitness Goal 34
4. STRINGS 36
4.1. Marshall Mather 36
4.2. Repeat 36

4.3. Print Backward 36
4.4. Name Game 36
4.5. Dracula 37
4.6. Stutter 37
4.7. Reverse 37
4.8. Switch Pairs 37
4.9. Contains Twice 37
4.10. Count Words 38
4.11. Is Vowel 38
4.12. Is All Vowels 38
4.13. Caesar Cipher 38
4.14. Add Commas 38
4.15. Remove All 38
4.16. Convert To Alt Caps 39
4.17. Remove Duplicates 39
4.18. DNA Errors 39
4.19. Start End Letter 40
4.20. Word Count 41
4.21. Is Palindrome 41
4.22. Reverse Chunks 41
4.23. Crazy Caps 42
4.24. Name Diamond 42
4.25. Swap Pairs 42
5. ARRAYS 43
5.1. Multi-dimensional arrays 43
5.1.1. Mystery #1 43
5.1.2. Mystery #2 43
5.1.3. Mystery #3 43
5.1.4. Mystery #4 44
5.1.5. Is Unique 44
5.1.6. Matrix Sum 45
5.2. Mystery #1 45
5.3. Mystery #2 45
5.4. Mystery #3 46
5.5. Mystery #4 46
5.6. Mystery #5 47

5.7. Mystery #6 47
5.8. Mystery #7 48
5.9. Mystery #8 48
5.10. Mystery #9 48
5.11. Mystery #10 49
5.12. Mystery #11 50
5.13. Mystery #12 50
5.14. Squared Array 51
5.15. Print 51
5.16. Max Value 51
5.17. Find Range 52
5.18. Get Percent Even 52
5.19. Compute Average 52
5.20. Sorted 52
5.21. Count Duplicates 52
5.22. Switch Pairs 53
5.23. Split 53
5.24. Has Mirror Twice 53
5.25. Longest Sorted Sequence 54
5.26. Contains 55
5.27. Even Before Odd 55
5.28. Index Of 56
5.29. Second Index Of 56
5.30. Num Unique 56
5.31. Collapse 57
5.32. Find Median 57
5.33. Banish 57
5.34. Sieve 58
5.35. Count Unique 59
5.36. Collapse Pairs 59
5.37. n Copies 59
5.38. Remove Palindromes 60
6. COLLECTIONS 61
6.1. ArrayList 61
6.1.1. Mystery #1 61
6.1.2. Mystery #2 61

6.1.3. Mystery #3 61
6.1.4. Mystery #4 62
6.1.5. Mystery #5 62
6.1.6. Mystery #6 63
6.1.7. Mystery #7 63
6.1.8. Remove Even Length 64
6.1.9. Mirror 64
6.1.10. Switch Pairs 64
6.1.11. Twice 64
6.1.12. Repeat 64
6.1.13. Remove Range 65
6.1.14. Delete Duplicates 65
6.1.15. Unique Names 65
6.1.16. Word Count 65
6.1.17. Histogram 66
6.1.18. Add Stars 66
6.1.19. Count In Range 66
6.1.20. Cumulative 67
6.1.21. Find Keith Numbers 67
6.1.22. Intersect 67
6.1.23. Is Keith Number 67
6.1.24. Mean 68
6.1.25. Remove All 68
6.1.26. Remove Bad Pairs 68
6.1.27. Remove Consecutive Duplicates 69
6.1.28. Stretch 69
6.2. Stack and Queue 69
6.2.1. Mystery #1 69
6.2.2. Mystery #2 70
6.2.3. Mystery #3 70
6.2.4. Mystery #4 70
6.2.5. Mystery #5 71
6.2.6. Mystery #6 72
6.2.7. Mystery #7 72
6.2.8. Mystery #8 73
6.2.9. Mystery #9 73

6.2.10. Check Balance 74
6.2.11. Flip Half 74
6.2.12. Is Sorted 75
6.2.13. Mirror 75
6.2.14. Reorder 75
6.2.15. Split Stack 75
6.2.16. Stutter 76
6.3. Set 76
6.3.1. Remove Range 76
6.3.2. Num Unique Values 76
6.3.3. Num In Common 76
6.3.4. Is Happy Number 77
6.3.5. Max Length 77
6.3.6. Remove Duplicates 77
6.3.7. Set Mystery 77
6.3.8. Twice 78
6.3.9. Union Sets 78
6.3.10. Word Count 78
6.4. Map 78
6.4.1. Mystery #1 78
6.4.2. Mystery #2 79
6.4.3. Mystery #3 79
6.4.4. Mystery #4 80
6.4.5. Mystery #5 80
6.4.6. Has Duplicate Value 81
6.4.7. Least Common 81
6.4.8. Count Names 82
6.4.9. Has Three 82
6.4.10. Intersect 82
6.4.11. Max Occurrences 82
6.4.12. Reverse 83
6.4.13. Is Sub Map 83
6.4.14. Get Majority Last Name 83
6.4.15. Rarest Age 84
6.4.16. Teacher 85
6.4.17. Postal Service 86

6.4.18. Biggest Family 87
6.4.19. By Age 88
6.4.20. Friend List 89
6.4.21. Last Names By Age 89
6.4.22. Mystery #6 90
6.4.23. Pair Frequencies 91
6.4.24. Rarest 92
6.4.25. Starters 92
6.4.26. To Morse Code 93
6.4.27. Word Chain 93
6.4.28. Flight Planner 95
7. OBJECTS AND CLASSES 97
7.1. Halloween 97
7.2. Bank Account toString 97
7.3. Bank Account Transfer 98
7.4. Circle 99
7.5. Date 100
7.6. Date-absoluteDay 100
7.7. Clock-advance 101
7.8. Student 102
7.9. Time Span 102

1. IF-ELSE

1.1. Mystery #1

For each call to the following method, indicate what output is produced.

```
public void mystery(int n) {
    print(n + " ");
    if (n > 0) {
        n = n - 5;
    }
    if (n < 0) {
        n = n + 7;
    } else {
        n = n * 2;
    }
    println(n);
}
```

mystery(8);	
mystery(-3);	
mystery(1);	
mystery(0);	

1.2. Mystery #2

Consider the following method. For each call below, indicate what output is produced.

```
public void mystery2(int a, int b) {
    if (a % b == 0) {
        a = a / b;
        if (a < b) {
            b = b - a;
        }
    } else if (b % 2 == 0) {
        b = b / 2;
    } else {
        a = a - b;
    }
    println(a + " " + b);
}
```

mystery2(20, 4);	
mystery2(7, 6);	
mystery2(14, 7);	
mystery2(24, 8);	
mystery2(13, 9);	

1.3. Percentage Grade

What's wrong with the following code? Modify it to produce the intended output. Make sure to properly utilize if/else/if statements to avoid redundancy and avoid unnecessary tests. Do not use && or || in your solution.

What percentage did you earn? 87

You got a B!

```
int percent = readInt("What percentage did you earn? ");
if (percent >= 90) {
    println("You got an A!");
}
if (percent >= 80) {
    println("You got a B!");
}
if (percent >= 70) {
    println("You got a C!");
}
if (percent >= 60) {
    println("You got a D!");
}
if (percent < 60) {
    println("You got an F!");
}
```

1.4. Date Is Before

Write a method **dateIsBefore** that takes as parameters two month/day combinations and that returns whether or not the first date comes before the second date (`true` if the first month/day comes before the second month/day, `false` if it does not). The method will take four integers as parameters that represent the two month/day combinations.

The first integer in each pair represents the month and will be a value between 1 and 12 (1 for January, 2 for February, etc, up to 12 for December). The second integer in each pair represents the day of the month (a value between 1 and 31). One date is considered to come before another if it comes earlier in the year.

Below are sample calls on the method.

Method Call	Return	Explanation
dateIsBefore(6, 3, 9, 20)	true	June 3 comes before Sep 20
dateIsBefore(10, 1, 2, 25)	false	Oct 1 does not come before Feb 25
dateIsBefore(8, 15, 8, 15)	false	Aug 15 does not come before Aug 15
dateIsBefore(8, 15, 8, 16)	true	Aug 15 comes before Aug 16

1.5. Compute Tax

Write a method named **computeTax** that accepts a salary as a parameter and that returns the amount of tax you would owe if you make that salary. The tax is based on your tax bracket as found from the first two columns below. Once you know which row to use, start with the "flat amount" and add the "plus %" of the amount over the amount listed in the final column. For example, if your income is $50,000, then you use the third row of the table and compute the tax as $4,000 plus 25% of the amount over $29,050, which comes to $9,237.50. The total tax on $27,500 is $3,767.50. For $6,000, the tax is $600. For $120,000, the tax is $28,227. Assume your method is passed a value ≥ 0.

over	but not over	flat amount	plus %	of excess over
$0	$7,150	$0	10%	$0
$7150	$29,050	$715	15%	$7,150
$29,050	$70,350	$4000	25%	$29,050
$70,350	*unlimited*	$14,325	28%	$70,350

2. LOOPS

2.1. Stars Print

What output is produced by each of the following code samples?

```
// A)
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 10; j++) {
        print("*");
    }
    println();   // to end the line
}
// B)
for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= i; j++) {
        print("*");
    }
    println();
}
// C)
for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= i; j++) {
        print(i);
    }
    println();
}
```

A:	
B:	
C:	

2.2. Number Loops #1

Write nested `for` loops to produce the following output:

```
1
22
333
4444
55555
```

2.3. Number Loops #2

Write nested `for` loops to produce the following output:

```
....1
...22
..333
.4444
55555
```

2.4. Number Loops #3

Write nested `for` loops to produce the following output:

```
....1
...2.
..3..
```

```
.4...
5....
```

2.5. Number Loops #4

The following program uses nested `for` loops to produce the following output:

```
....1
...2.
..3..
.4...
5....
```

Modify the program to use a single integer constant named `SIZE` that influences how many lines should be drawn. For example, if the `SIZE` is changed to 7, the output should become the following:

```
......1
.....2.
....3..
...4...
..5....
.6.....
7......
```

2.6. Mystery #1

What does the following code print out?

```
for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 10 - i; j++) {
        print(" ");
    }
    for (int j = 1; j <= 2 * i - 1; j++) {
        print("*");
    }
    println();
}
```

Output:	

2.7. Mystery #2

For each call of the method below, write the output that is printed and the value that is returned:

```
public int loopMysteryExam1(int i, int j) {
    while (i != 0 && j != 0) {
        i = i / j;
        j = (j - 1) / 2;
        print(i + " " + j + " ");
    }
    println(i);
    return i + j;
}
```

loopMysteryExam1(5, 0) output	
loopMysteryExam1(5, 0) return	

loopMysteryExam1(3, 2) output	
loopMysteryExam1(3, 2) return	
loopMysteryExam1(16, 5) output	
loopMysteryExam1(16, 5) return	
loopMysteryExam1(80, 9) output	
loopMysteryExam1(80, 9) return	
loopMysteryExam1(1600, 40) output	
loopMysteryExam1(1600, 40) return	

2.8. Mystery #3

For each call of the method below, write the output that is printed and the value that is returned:

```
public int loopMysteryExam3(int x, int y) {
    int z = x + y;
    while (x > 0 && y > 0) {
        x = x - y;
        y--;
        print(x + " " + y + " ");
    }
    println(y);
    return z;
}
```

loopMysteryExam3(7, 5); output	
loopMysteryExam3(7, 5); return	
loopMysteryExam3(20, 4); output	
loopMysteryExam3(20, 4); return	
loopMysteryExam3(40, 10); output	
loopMysteryExam3(40, 10); return	

2.9. Mystery #4

For each call of the method below, write the output that is printed as it would appear on the console.

```
public void loopMysteryExam5(int x, int y) {
    int z = y;
    while (x % z == 0) {
        print("(" + x + " " + z + ") ");
        x = x - z;
        z++;
    }
    println("(" + x + " " + z + ") " + y);
}
```

loopMysteryExam5(12, 4);	
loopMysteryExam5(14, 2);	
loopMysteryExam5(27, 3);	

2.10. Pow

Write a method named `pow` that accepts a base and an exponent as parameters and returns the base raised to the given power. For example, the call pow(3, 4) returns 3 * 3 * 3 * 3 or 81. Do not use Math.pow in your solution. Assume that the base and exponent are non-negative.

2.11. Sentinel Sum

Write a complete console program in a class named SentinelSum that prompts the user for numbers until the user types -1, then outputs the sum of the numbers. Here is an example output:

```
Type a number: 10
Type a number: 20
Type a number: 30
Type a number: -1
Sum is 60
```

2.12. Biggest And Smallest

Write a console program in a class named **BiggestAndSmallest** that prompts the user to type a given number of integers, then prints the largest and smallest of all the numbers typed in by the user. You may assume the user enters a number greater than 0 for the number of numbers to read. Here is an example dialogue:

```
Number of numbers? 4
Number 1: 5
Number 2: 13
Number 3: -5
Number 4: 2
Biggest = 13
Smallest = -5
```

2.13. Compute Sum Of Digits

Write a console program in a class named **ComputeSumOfDigits** that prompts the user to type an integer and computes the sum of the digits of that integer. You may assume that the user types a non-negative integer. Match the following output format:

```
Type an integer: 827104
Digit sum is 22
```

2.14. Range Of Numbers

Write a console program in a class named **RangeOfNumbers** that prompts the user to type two integers and prints the sequence of numbers between the two arguments, separated by commas and spaces. Print an increasing sequence if the first argument is smaller than the second; otherwise, print a decreasing sequence. If the two numbers are the same, that number should be printed by itself. Here is an example dialogue:

```
Start? 2
End? 8
2, 3, 4, 5, 6, 7, 8
```

Your program should also be able to count downward if the end is less than the start. For example:

```
Start? 18
```

```
End? 11
18, 17, 16, 15, 14, 13, 12, 11
```

If the start and end are the same, simply print that number a single time.

```
Start? 42
End? 42
42
```

2.15. Number Square

Write a console program in a class named **NumberSquare** that prompts the user for two integers, a *min* and a *max*, and prints the numbers in the range from *min* to *max* inclusive in a square pattern. Each line of the square consists of a wrapping sequence of integers increasing from *min* and *max*. The first line begins with *min*, the second line begins with *min* + 1, and so on. When the sequence in any line reaches *max*, it wraps around back to *min*. You may assume that *min* is less than or equal to *max*. Here is an example dialogue:

```
Min? 1
Max? 5
12345
23451
34512
45123
51234
```

2.16. Fizz Buzz

Write a console program in a class named **FizzBuzz** that prompts the user for an integer, then prints all of the numbers from one to that integer, separated by spaces. Use a loop to print the numbers. But for multiples of three, print "Fizz" instead of the number, and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz". Drop to a new line after print each 20 numbers. If the user typed 100, the output would be:

```
Max value? 100
1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz 16 17 Fizz 19 Buzz
Fizz 22 23 Fizz Buzz 26 Fizz 28 29 FizzBuzz 31 32 Fizz 34 Buzz Fizz 37 38 Fizz Buzz
41 Fizz 43 44 FizzBuzz 46 47 Fizz 49 Buzz Fizz 52 53 Fizz Buzz 56 Fizz 58 59 FizzBuzz
61 62 Fizz 64 Buzz Fizz 67 68 Fizz Buzz 71 Fizz 73 74 FizzBuzz 76 77 Fizz 79 Buzz
Fizz 82 83 Fizz Buzz 86 Fizz 88 89 FizzBuzz 91 92 Fizz 94 Buzz Fizz 97 98 Fizz Buzz
```

2.17. Fibonacci Constant

Write a console program in a class named **Fibonacci** that displays all the numbers in the Fibonacci Sequence up to a given max, starting with 0. The Italian mathematician Leonardo Fibonacci devised the Fibonacci sequence as a way to model the growth of a population of rabbits. The first two terms in the sequence are 0 and 1, and every subsequent term is a sum of the previous two terms. (The Fibonacci sequence has numerous applications in computer science and shows up in surprising places. It's used to compute logarithms, index and retrieve data, and as a building block in some route-planning algorithms.) Output from one example run:

```
This program lists the Fibonacci sequence.
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
```

This program should continue as long as the value of the term is less than the maximum value. To do this, you should use a while loop, presumably with a header line that looks like this:

```
while (term < MAX_TERM_VALUE)
```

Note that the maximum term value is specified using a named constant.

2.18. Roll Two Dice

Write a console program in a class named **RollTwoDice** that prompts the user for a desired sum, then repeatedly rolls two six-sided dice (using a Random or RandomGenerator object to generate random numbers from 1-6) until the sum of the two dice values is the desired sum. Here is the expected dialogue with the user:

```
Desired sum: 9
4 and 3 = 7
3 and 5 = 8
5 and 6 = 11
5 and 6 = 11
1 and 5 = 6
6 and 3 = 9
```

2.19. Piglet

Write a console program in a class named **Piglet** that implements a simple 1-player dice game called "Piglet" (based on the game "Pig"). The player's goal is to accumulate as many points as possible without rolling a 1. Each turn, the player rolls the die; if they roll a 1, the game ends and they get a score of 0. Otherwise, they add this number to their running total score. The player then chooses whether to roll again, or end the game with their current point total. Here is an example dialogue where the user plays until rolling a 1, which ends the game with 0 points:

```
Welcome to Piglet!
You rolled a 5!
Roll again? yes
You rolled a 4!
Roll again? yes
You rolled a 1!
You got 0 points.
```

Here is another example dialogue where the user stops early and gets to end the game with 10 points:

```
Welcome to Piglet!
You rolled a 6!
Roll again? y
You rolled a 2!
Roll again? y
You rolled a 2!
Roll again? n
You got 10 points.
```

You can use the method `readBoolean` to ask the user a yes/no question.

3. PARAMETERS AND RETURN

3.1. Box Of Stars

Write a method named boxOfStars that accepts two integer parameters representing a width and height in characters, and prints to the console a 'box' figure whose border is * stars and whose center is made of spaces. For example, the call of boxOfStars(8, 5); should print the following output:

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

You may assume that the values passed for the width and height are at least 2.

3.2. Triangle

Write a method named triangle that accepts an integer parameter representing a size in characters, and prints to the console a right-aligned right triangle figure whose non-hypotenuse sides are that length. For example, the call of triangle(5); should print the following output:

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

You may assume that the value passed for the size is at least 1.

3.3. Print Pay

Write a method named printPay that computes and prints the amount of money an employee should earn. Your method accepts two parameters: a real number for the employee's hourly salary, and an integer for the number of hours the employee worked. Every hour over 8 is paid at 1.5x the normal salary. For example, the call of printPay(10.00, 11); should print the following output:

```
Hours worked: 11
Pay earned: $125.00
```

You may assume that the value passed for the salary and hours are non-negative. Use the printf method to format real numbers properly.

3.4. Count Digits

Write a method named countDigits that accepts an integer parameter and returns the number of digits in that integer. For example, countDigits(38015) returns 5. For negative numbers, return the same value as if the number were positive. For example, countDigits(-72) returns 2.

3.5. Mystery #1

The following console program uses parameters and produces two lines of output. What are they?

```
public class ParameterMystery1 extends ConsoleProgram {
    public void run() {
        int x = 9;
        int y = 2;
        int z = 5;

        mystery(z, y, x);
        mystery(y, x, z);
    }
    public void mystery(int x, int z, int y) {
        println(z + ", " + (y - x));
    }
}
```

line 1	
line 2	

3.6. BMI

Write a console program in a class named **BMI** that prompts for user input and calculates 2 people's body mass index (BMI), using the following formula:

BMI = weight / height^2 * 703

The BMI rating groups each person into one of the following four categories:

BMI	Category
below 18.5	class 1
18.5 - 24.9	class 2
25.0 - 29.9	class 3
30.0 and up	class 4

Match the following example output:

```
This program reads data for two people
and computes their body mass index (BMI).

Person 1's information:
height (in inches)? 70.0
weight (in pounds)? 194.25
BMI = 27.9
```

```
class 3

Person 2's information:
height (in inches)? 62.5
weight (in pounds)? 130.5
BMI = 23.5
class 2

Have a nice day!
```

You should break down your program into several **methods**, each of which helps solve the overall problem. Use the `printf` method to format real numbers with proper precision.

3.7. Mystery #2

The following console program uses parameters and produces three lines of output. What are they?

```
public class ParameterMysterySection1 extends ConsoleProgram {
    public void run() {
        int a = 4;
        int b = 7;
        int c = -2;

        m(a, b, c);
        m(c, 3, a);
        m(a + b, b + c, c + a);
    }
    public void m(int c, int a, int b) {
        println(b + " + " + c + " = " + a);
    }
}
```

line 1	
line 2	
line 3	

3.8. Show Twos

Write a method named **showTwos** that shows the factors of 2 in a given integer. For example, the following calls:

```
showTwos(7);
showTwos(18);
showTwos(68);
showTwos(120);
```

Should produce this output:

```
7 = 7
18 = 2 * 9
68 = 2 * 2 * 17
120 = 2 * 2 * 2 * 15
```

The idea is to express the number as a product of factors of 2 and an odd number. The number 120 has 3 factors of 2 multiplied by the odd number 15. For odd numbers (e.g. 7), there are no factors of 2, so you just show the number itself. Assume that your method is passed a number greater than 0.

3.9. Mystery #3

The following console program uses returns and produces three lines of output. What are they?

```
public class ReturnMystery1 extends ConsoleProgram {
    public void run() {
        int a = 4;
        int b = 2;
        int c = 5;

        a = mystery(c, b);
        c = mystery(b, a);
        println(a);
        println(b);
        println(c);
    }
    public int mystery(int b, int c) {
        return c + 2 * b;
    }
}
```

line 1	
line 2	
line 3	

3.10. Mystery #4

The following console program uses parameters and produces eight lines of output. What are they?

```
public class ParameterMysterySection2 extends ConsoleProgram {
    public void run() {
        int a = 137;
        int b = 42;

        println("a = " + a);
        foo(b);
        println("a = " + a);
        println("b = " + b);

        a = bar(b, a + b);
        println("a = " + a);
        a = bar(a, b);
        println("a = " + a);
    }
    public void foo(int a) {
        println("a = " + a);
        a = 160;
    }
    public int bar(int c, int b) {
        int d = b - c;
        println("d = " + d);
        return d % 10;
    }
}
```

line 1	
line 2	
line 3	
line 4	
line 5	
line 6	
line 7	
line 8	

3.11. Mystery #5

The following console program uses parameters and produces five lines of output. What are they?

```
public class ParameterMysteryExam1 extends ConsoleProgram {
    public void run() {
        String x = "java";
        String y = "tyler";
        String z = "tv";
        String rugby = "hamburger";
        String java = "donnie";

        hamburger(x, y, z);
        hamburger(z, x, y);
        hamburger("rugby", z, java);
        y = hamburger(y, rugby, "x");
        hamburger(y, y, "java");
    }

    public String hamburger(String y, String z, String x) {
        println(z + " and " + x + " like " + y);
        return z;
    }
}
```

line 1	
line 2	
line 3	
line 4	
line 5	

3.12. Mystery #6

The following console program uses parameters and produces five lines of output. What are they?

```
public class ParameterMysteryExam2 extends ConsoleProgram {
    public void run() {
        String x = "happy";
        String y = "pumpkin";
        String z = "orange";
        String pumpkin = "sleepy";
        String orange = "vampire";

        orange(y, x, z);
        x = orange(x, z, y);
        orange(pumpkin, z, "y");
        z = "green";
        orange("x", "pumpkin", z);
        orange(x, z, orange);
    }

    public String orange(String z, String y, String x) {
        println(y + " and " + z + " were " + x);
        return x + y + z;
    }
}
```

line 1	
line 2	
line 3	
line 4	
line 5	

3.13. Mystery #7

The following console program uses parameters and produces four lines of output. What are they?

```
public class ParameterMysteryExam3 extends ConsoleProgram {
    public void run() {
        int a = 5;
        int b = 1;
        int c = 3;
        int three = a;
        int one = b + 1;

        axiom(a, b, c);
        axiom(c, three, 10);
        three = axiom(b + c, one + three, a + one);
        a++;
        b = 0;
        a = axiom(three, 2, one);
    }

    public int axiom(int c, int b, int a) {
        println(a + " + " + c + " = " + b);
        return a + b;
    }
}
```

line 1	
line 2	
line 3	
line 4	

3.14. Mystery #8

The following console program uses parameters and produces four lines of output. What are they?

```
public class ParameterMysteryExam4 extends ConsoleProgram {
    public void run() {
        int x = 5;
        int y = 1;
        int z = 9;
        int w = y + 2;

        surprise(x, y);
        x = surprise(z, w);
        w++;
        z = surprise(w, x);
        surprise(y, z);
    }

    public int surprise(int y, int x) {
        x++;
        println(x + " " + y);
        y--;
        return x;
    }
}
```

line 1	
line 2	
line 3	

line 4	

3.15. Mystery #9

The following console program uses parameters and produces four lines of output. What are they?

```
public class ParameterMysteryExam5 extends ConsoleProgram {
    public void run() {
        int a = 10;
        int b = 4;
        int c = a + b;

        int d = mystery(a, b, c);
        b--;
        c = mystery(c, d, a);
        mystery(a, c, b);
        println(a + " " + b + " " + c + " " + d);
    }

    public int mystery(int b, int c, int a) {
        c++;
        println(c + " " + a + " " + b);
        return a + 2;
    }
}
```

line 1	
line 2	
line 3	
line 4	

3.16. Mystery #10

The following console program uses parameters and produces four lines of output. What are they?

```
public class ParameterMysteryExam6 extends ConsoleProgram {
    public void run() {
        String scarlett = "mustard";
        String suspect = "peacock";
        String lounge = "ballroom";
        String pipe = "study";
        String dagger = "pipe";
        String weapon = "dagger";

        clue(weapon, suspect, pipe);
        clue(scarlett, pipe, suspect);
        dagger = clue(dagger, "lounge", scarlett);
        clue(dagger, lounge, "dagger");
    }

    public String clue(String suspect, String room, String weapon) {
        println(room + " in the " + weapon + " with the " + suspect);
        return room;
    }
}
```

line 1	
line 2	
line 3	
line 4	

3.17. Mystery #11

For the following console program, trace through its execution by hand to write the three lines of output that are produced when it runs:

```
public class Hogwarts extends ConsoleProgram {
    public void run() {
        bludger(2001);
    }

    private static void bludger(int y) {
        int x = y / 1000;
        int z = (x + y);
        x = quaffle(z, y);
        println("bludger: x = " + x + ", y = " + y + ", z = " + z);
    }

    private static int quaffle(int x, int y) {
        int z = snitch(x + y, y);
        y /= z;
        println("quaffle: x = " + x + ", y = " + y + ", z = " + z);
        return z;
    }

    private static int snitch(int x, int y) {
        y = x / (x % 10);
        println("snitch: x = " + x + ", y = " + y);
        return y;
    }
}
```

line 1	
line 2	
line 3	

3.18. Mystery #12

For the following console program, trace through its execution by hand to write the four lines of output that are produced when it runs:

```
public class OneTwoThree extends ConsoleProgram {
    public void run() {
        int a = 100;
        addOne();
        addTwo(a);
        a = addThreeAndReturnResult(a);
        println("run: a = " + a + ", b = " + b);
    }

    private static void addOne() {
        int a = 101;
        b++;
        println("addOne: a = " + a + ", b = " + b);
    }

    private static void addTwo(int a) {
        a += 2;
        b += 2;
        println("addTwo: a = " + a + ", b = " + b);
    }

    private static int addThreeAndReturnResult(int a) {
        a += 3;
        b += 3;
        println("addThreeAndReturnResult: a = " + a + ", b = " + b);
        return a;
    }

    /* Private instance variable */
    private int b = 200;
}
```

line 1	
line 2	
line 3	
line 4	

3.19. Mystery #13

For the following console program, trace through its execution by hand to write the five lines of output that are produced when it runs:

```
public class Mystery extends ConsoleProgram {
    public void run() {
        ghost(13);
    }

    private static void ghost(int x) {
        int y = 0;
        for (int i = 1; i < x; i *= 2) {
            y = witch(y, skeleton(x, i));
        }
        println("ghost: x = " + x + ", y = " + y);
    }

    private static int witch(int x, int y) {
        x = 10 * x + y;
        println("witch: x = " + x + ", y = " + y);
        return x;
    }

    private static int skeleton(int x, int y) {
        return x / y % 2;
    }
}
```

line 1	
line 2	
line 3	
line 4	
line 5	

3.20. Circle Area

Write a method named `circleArea` that accepts the radius of a circle as a parameter (as a real number) and returns the area of a circle with that radius. For example, the call of `area(2.0)` should return `12.566370614359172`. You may assume that the radius passed is a non-negative number.

3.21. Average Of 3

Write a method named `averageOf3` that accepts three integers as parameters and returns the average of the three integers as a real number. For example, the call of `averageOf3(4, 7, 13)` returns `8`.

3.22. Compute Distance

Write a method named **computeDistance** that accepts four integer coordinates *x1*, *y1*, *x2*, and *y2* as parameters and computes the distance between points (*x1*, *y1*) and (*x2*, *y2*) on the Cartesian plane. The formula for the distance between two points is:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

For example, the call of computeDistance(10, 2, 3, 5) would return 7.615773105863909.

3.23. Get Days In Month

Write a method named **getDaysInMonth** that accepts an integer parameter representing a month (between 1 and 12) and returns the number of days in that month in a non-leap year. For example, the call getDaysInMonth(9) would return 30 because September has 30 days.

Month	1 Jan	2 Feb	3 Mar	4 Apr	5 May	6 Jun	7 Jul	8 Aug	9 Sep	10 Oct	11 Nov	12 Dec
Days	31	28	31	30	31	30	31	31	30	31	30	31

3.24. Factorial

Write a method named factorial that accepts an integer n as a parameter and returns the factorial of n, or n!. A factorial of an integer is defined as the product of all integers from 1 through that integer inclusive. For example, the call of factorial(4) should return 1 * 2 * 3 * 4, or 24. The factorial of 0 and 1 are defined to be 1. You may assume that the value passed is non-negative and that its factorial can fit in the range of type int.

3.25. Roulette

Write a console program in a class named **Roulette** that simulates the gambling game of Roulette, with the following characteristics. The player begins with $10 and bets (up to) $3 per spin of the wheel. If the wheel comes up 1-18, the player wins $3. Otherwise, player loses $3. Play until the player gets $1000 or drops to $0. At the end, print the max money the player ever earned. Here is an example dialogue:

```
bet $3, spin 15, money = $13
bet $3, spin 35, money = $10
bet $3, spin 7, money = $13
bet $3, spin 4, money = $16
bet $3, spin 28, money = $13
bet $3, spin 19, money = $10
bet $3, spin 21, money = $7
bet $3, spin 26, money = $4
bet $3, spin 36, money = $1
bet $1, spin 22, money = $0
max = $16
```

You should break down your program into several **methods**, each of which helps solve the overall problem. (Because this program uses random numbers, our site cannot perfectly verify its behavior; we will look to see that you have the right overall output format, but it is up to you to verify whether you are handling the spin/money logic correctly.)

3.26. Sum Of Range

Write a method named sumOfRange method that accepts two integer parameters min and max and returns the sum of the integers from min through maxinclusive. For example, the call of sumOfRange(3, 7) returns 3 + 4 + 5 + 6 + 7 or 25. If min is greater than max, return 0.

3.27. Get Last Digit

Write a method named getLastDigit that returns the last digit of an integer. For example, the call of getLastDigit(3572) should return 2.

3.28. Get First Digit

Write a method named getFirstDigit that returns the first digit of an integer. For example, getFirstDigit(3572) should return 3.

3.29. Sum Of Digits

Write a method named sumOfDigits that accepts an integer parameter and computes and returns the sum of all the digits of that number. For example, sumOfDigits(38015) returns 3+8+1+0+5 or 17. For negative numbers, return the same value as if the number were positive. For example, sumOfDigits(-72) returns 7+2 or 9.

3.30. Decimal To Binary

Write a method named **decimalToBinary** that accepts an integer as a parameter and returns an integer whose digits look like that number's representation in binary (base-2). For example, the call of decimalToBinary(43) should return 101011.

Constraints: Do not use a string in your solution. Also do not use any built-in base conversion methods like Integer.toString .

3.31. Binary To Decimal

Write a method named **binaryToDecimal** that accepts an integer parameter whose digits are meant to represent binary (base-2) digits, and returns an integer of that number's representation in decimal (base-10). For example, the call of binaryToDecimal(101011) should return 43.

Constraints: Do not use a string in your solution. Also do not use any built-in base conversion methods like Integer.toString .

3.32. Factor Count

Write a method named factorCount that accepts an integer (assumed to be positive) as its parameter and returns a count of its positive factors. For example, the eight factors of 24 are 1, 2, 3, 4, 6, 8, 12, and 24, so the call of factorCount(24) should return 8.

3.33. Random Over

Write a method named **randomOver** that accepts two integer parameters n and max and repeatedly prints random numbers between 0 and max (inclusive) until a number greater than or equal to n is printed. At least one line of output should always be printed, even if the first random number is above n. Here is a sample log from the call of randomOver(900, 1000);

```
Random number: 235
Random number: 15
Random number: 810
Random number: 147
Random number: 915
```

3.34. Is Multiple

Write a method named isMultiple that accepts two non-negative int parameters a and b, and returns true if a is a multiple of b, and false otherwise. For example, the call of isMultiple(15, 5) would return true because 15 = 5 * 3. You may assume that a and b are non-negative integers and that b is not 0.

3.35. Three Consecutive

Write a method named threeConsecutive that accepts three integers as parameters and returns true if they are three consecutive numbers; that is, if the numbers can be arranged into an order such that their values differ by exactly 1. For example, the call of threeConsecutive(3, 2, 4) would return true.

3.36. Is Prime Number

Write a method named isPrimeNumber that accepts an integer as a parameter and returns true if that integer is a prime number. A prime number is an integer that has no factors other than 1 and itself. The number 2 is defined as the smallest prime number.

3.37. Coin Flip

Write a method named **coinFlip** that simulates repeatedly flipping a two-sided coin until a particular side (Heads or Tails) comes up several times consecutively (in a row). Your method accepts two parameters, an integer *k* and a character *side*, where *side* is expected to be 'H' for Heads or 'T' for Tails. You should keep simulating the flipping of the coin until *k* occurrences of the given side are seen consecutively. For example, if the call is `coinFlip(3, 'H');` , you should flip the coin until Heads is seen 3 times in a row. Here is an example output from the call of `coinFlip(4, 'T');`

```
T H T H T T H T T H H T H H H H H T T T T
You got T 4 times in a row!
```

If a negative *k* is passed, and/or a side value is passed that is not `'H'` or `'T'`, your method should print `ERROR!` and exit immediately.

Use a `RandomGenerator` or `Random` object to give an equal chance to a head or a tail appearing. Each time the coin is flipped, what is seen is displayed (H for heads, T for tails), separated by spaces. When *k* consecutive occurrences of the given side occur, a congratulatory message is printed. Match our output format exactly.

3.38. Random Walk

Write a method named **randomWalk** that simulates a 1-dimensional "random walk" algorithm. A *random walk* is where an integer value is repeatedly increased or decreased by 1 randomly many times until it hits some threshold. Your method should accept the integer *threshold* as a parameter, then start an integer at 0 and adjust it by +1 or -1 repeatedly until its value reaches positive or negative *threshold*. For example, if the call of `randomWalk(3);` is made, your method would randomly walk until it hits 3 or -3. Each time the value is adjusted, it is printed in the format shown. When you have reached the threshold, report the number of steps that were taken from the starting point of 0, as well as the maximum position that was reached during the walk. (If the walk ever reaches positive *threshold*, that is the maximum position.)

The log below shows the output from an example call of `randomWalk(3);` . You should match the output format below exactly, though the numbers are randomly generated. Use a `RandomGenerator` object and give an equal chance of moving by +1 and -1 on each step. If the threshold parameter passed to your method is not greater than 0, your method should produce no output.

```
Position = 0
Position = 1
Position = 0
Position = -1
Position = -2
Position = -1
Position = -2
Position = -3
Finished after 7 step(s)
Max position = 1
```

(Because this problem uses random numbers, it is hard for our system to perfectly verify your code. Make sure to match our output format exactly.)

3.39. Play Roulette

Write a method named **playRoulette** that generates random numbers to simulate playing a simplified version of the casino game Roulette, as described below. Roulette has a wheel laid flat with colored numbered areas numbered from 0-36. (In some versions, the wheel also has a "00", but not in this problem.) In each round, the wheel is spun and a ball is tossed onto it, landing on one of the numbers. A player bets money on groups of numbers where he/she thinks the ball will land.

For our game, the player bets that the ball will land on any **even number**. If the ball lands on an even number, the player wins; and if the ball lands on any odd number, the player loses. The **number 0** region is special; if the ball lands there, the player loses.

Your method accepts two **parameters**: an integer for the dollars the player starts with; and another integer for the dollars the player will bet each round. At the start, print three headings of "bet", "spin", and "money" separated by a tab (\t). Then repeatedly use random numbers to simulate spins of the roulette wheel. The player bets the given amount each time, or as close to that amount as the player can afford, until their money reaches 0. If the player doesn't have enough money to bet the given amount, they bet all of their current money. For example, if the bet amount is $5 but the player has only $2, the player bets $2. If the player wins (ball lands on a positive even number), the player receives the amount they bet. If the player loses (ball lands on an odd number or 0), the player loses the amount they bet. After each spin, print the bet amount, the number where the ball landed, and the player's money, separated by a tab (\t). Keep going as long as it takes for the money to reach 0. At the end, print the maximum money the player ever had during the session.

The following log represents the console output from an example call to your method. Because the code uses randomness, each run produces unique output, but your method should exactly match the behavior and output structure shown.

```
playRoulette(10, 3);
```

```
start with $10, betting up to $3
bet     spin    money
$3      11      $7
$3      22      $10
$3      5       $7
$3      30      $10
$3      24      $13
$3      30      $16
$3      32      $19
$3      0       $16
$3      9       $13
$3      11      $10
$3      1       $7
$3      33      $4
$3      12      $7
```

```
$3      0      $4
$3      19     $1
$1      14     $2
$2      34     $4
$3      5      $1
$1      31     $0
max money: $19
```

Assumptions: You may assume that the parameter values passed will be reasonable; that is, the starting money will be non-negative and the bet amount will be a positive integer. The bet amount might be greater than the starting money.

Constraints: You should not use data structures such as arrays or strings to help you solve this problem. You may declare as many simple variables (such as `int`s or `double`s) as you like.

3.40. Fitness Goal

Write a method named **fitnessGoal** that accepts an integer parameter named goal, and prompts the user for how many minutes he/she exercises each day until they increase their minutes for goal consecutive days in a row.

For example, the call of `fitnessGoal(3);` must prompt the user until they increase their minutes for 3 days straight. If the user exercises for 5 minutes the first day and 10 minutes the second day, the user increased their minutes of exercise. But if the user exercises for only 8 minutes on the third day (or any amount less than 10), he/she did not increase, so the counting starts over. On the fourth day, the user would now need to exceed 8 to be increasing.

After prompting for minutes of exercise each day, print "Great job!" if their minutes are increasing or "Start over." if the minutes did not increase. We will implicitly assume that the user did not exercise on the day before the method begins running, so no matter how many minutes they exercise on day 1, they did a great job and have been increasing for 1 day.

The following log represents the console output from an example call of `fitnessGoal(3);` Your method should exactly match the behavior and output structure shown. (Console user input is shown **like this** for clarity.)

```
Train until you increase for 3 days.
Minutes? 5
Great job!
Minutes? 10
Great job!
Minutes? 8
Start over.
Minutes? 9
Great job!
Minutes? 11
```

```
Great job!
Minutes? 2
Start over.
Minutes? 10
Great job!
Minutes? 15
Great job!
Minutes? 20
Great job!
Reached your goal in 9 total days!
```

Notice that the method terminates when the user has increased for 3 straight days. In this case, it is the increase from 2 minutes to 10, then from 10 to 15, then from 15 to 20. If the call had instead been `fitnessGoal(5);`, we would re-prompt the user until their minutes had been increasing for 5 straight days.

Assumptions: You may assume that the parameter value passed will be a positive integer, and you may assume that the user will type a positive integer for the number of minutes each day. You don't need to worry about pluralizing words in the output such as "day" vs "days."

Constraints: You should not use data structures such as arrays or strings to help you solve this problem. You may declare as many simple variables (such as `int`s or `double`s) as you like.

4. STRINGS

4.1. Marshall Mather

Given the following variable declarations:

```
// index        0123456789012345678901234
String.s1   = "Snoop Dogg";
String s2   = "Marshall 'Eminem' Mathers";
String s3   = s2.substring(18);
String book = "Art & Science of Java";
```

Give the results of the following expressions. Make sure to indicate a value of the proper type (e.g. strings in " " quotes).

s1.length()	
s2.length()	
s1.indexOf("o")	
s2.indexOf("x")	
s1.substring(6, 9)	
s2.substring(21)	
book.substring(6, 12)	
s3.toLowerCase()	

4.2. Repeat

Write a method named `repeat` that accepts a `String` and a number of repetitions as parameters and returns the `String` concatenated that many times. For example, the call of `repeat("hello", 3)` returns `"hellohellohello"`. If the number of repetitions is 0 or less, return an empty string.

4.3. Print Backward

Write a method named **printBackward** that accepts a `String` as its parameter and prints the characters in the opposite order. For example, a call of `printBackward("hello there!");` should print the following output:

```
!ereht olleh
```

If the empty string is passed, no output should be produced.

4.4. Name Game

Write a console program in a class named **NameGame** that prints the following rhyme about the person's first and last name. You may assume that the user types a string with exactly one space.

```
What is your name? Fifty Cent
Fifty, Fifty, bo-Bifty
Banana-fana fo-Fifty
Fee-fi-mo-Mifty
FIFTY!
```

```
Cent, Cent, bo-Bent
Banana-fana fo-Fent
Fee-fi-mo-Ment
CENT!
```

4.5. Dracula

What is the value of `s` after the following code is finished running?

```
String s = "dracula";
for (int i = 0; i < s.length(); i++) {
    char a = s.charAt(0);
    String b = s.substring(1);
    s = b + a;
}
```

A	"Dracula"
B	"Alucard"
C	"Alucarddracula"
D	"Ddrraaccuullaa"
E	"Draculaalucard"

4.6. Stutter

Write a method named `stutter` that accepts a string parameter returns a new string replacing each of its characters with two consecutive copies of that character. For example, a call of `stutter("hello")` would return `"hheellllloo"`.

4.7. Reverse

Write a method named `reverse` that accepts a string parameter returns a new string with the characters in the opposite order. For example, A call of `reverse("Pikachu")` would return `"uchakiP"`.

4.8. Switch Pairs

Write a method named `switchPairs` that accepts a string as a parameter and returns that string with each pair of neighboring letters reversed. If the string has an odd number of letters, the last letter should not be modified. For example, the call `switchPairs("example")` should return `"xemalpe"` and the call `switchPairs("hello there")` should return `"ehll ohtree"`.

4.9. Contains Twice

Write a method named `containsTwice` that accepts a string and a character as parameters and returns `true` if that character occurs two or more times in the string. For example, the call of `containsTwice("hello", 'l')` should return `true` because there are two `'l'` characters in that string.

4.10. Count Words

Write a method named countWords that accepts a string as its parameter and returns the number of words in it. A word is a sequence of one or more non-space characters. For example, the call of countWords("What is your name?") should return 4.

4.11. Is Vowel

Write a method named isVowel that returns whether a string is a vowel (a single-letter string containing a, e, i, o, or u, case-insensitively).

4.12. Is All Vowels

Write a method named **isAllVowels** that returns whether a string consists entirely of vowels (a, e, i, o, or u, case-insensitively). For example, isAllVowels("eiEIo") should
return true while isAllVowels("banana") should return false.

For this problem, you may assume that a working solution already exists to the previous problem, isVowel, and you may call it in your solution.

4.13. Caesar Cipher

Write a console program in a class named **CaesarCipher** that implements a *Caesar cipher* or *rotation cipher*, which is a crude system of encoding strings by shifting every letter forward by a given number. Your program should prompt the user to type a message and an encoding "key" (number of places to shift each character) and display the shifted message. For example, if the shift amount is 3, then the letter A becomes D, and B becomes E, and so on. Letters near the end of the alphabet wrap around; for a shift of 3, X becomes A, and Y becomes B, and Z becomes C. Here are two example dialogues:

```
Your message? Attack zerg at dawn
Encoding key? 3
DWWDFN CHUJ DW GDZQ
Your message? DWWDFN CHUJ DW GDZQ
Encoding key? -3
ATTACK ZERG AT DAWN
```

4.14. Add Commas

Write a method named addCommas that accepts a string representing a number and returns a new string with a comma at every third position, starting from the right. For example, the call of addCommas("12345678") returns "12,345,678".

4.15. Remove All

Write a method named removeAll that accepts a string and a character as parameters, and removes all occurrences of the character. For example, the call of `removeAll("Summer is here!", 'e')` should return `"Summr is hr!"`. Do not use the string `replace` method in your solution.

4.16. Convert To Alt Caps

Write a method named `convertToAltCaps` that accepts a string as a parameter and returns a version of the string where alternating letters are uppercase and lowercase, starting with the first letter in lowercase. For example, the call of `convertToAltCaps("Pikachu")` should return `"pIkAcHu"`.

4.17. Remove Duplicates

Write a method named `removeDuplicates` that accepts a string parameter and returns a new string with all consecutive occurrences of the same character in the string replaced by a single occurrence of that character. For example, the call of `removeDuplicates("bookkeeeeeper")` should return `"bokeper"` .

4.18. DNA Errors

Write a method named **dnaErrors** that accepts two strings representing DNA sequences as parameters and returns an integer representing the number of errors found between the two sequences, using a formula described below. DNA contains *nucleotides*, which are represented by four different letters A, C, T, and G. DNA is made up of a pair of nucleotide strands, where a letter from the first strand is paired with a corresponding letter from the second. The letters are paired as follows:

- A is paired with T and vice-versa.
- C is paired with G and vice-versa.

Below are two perfectly matched DNA strands. Notice how the letters are paired up according to the above rules.

```
"GCATGGATTAATATGAGACGACTAATAGGATAGTTACAACCCTTACGTCACCGCCTTGA"
 |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
"CGTACCTAATTATACTCTGCTGATTATCCTATCAATGTTGGGAATGCAGTGGCGGAACT"
```

In some cases, errors occur within DNA molecules; the task of your method is to find two particular kinds of errors:

- *Unmatched nucleotides*, in which one strand contains a dash ('-') at a given index, or does not contain a nucleotide at the given index (if the strings are not the same length). Each of these counts as **1 error**.
- *Point mutations*, in which a letter from one strand is matched against the wrong letter in the other strand. For example, A might accidentally pair with C, or G might pair with G. Each of these counts as **2 errors**.

For example, consider these two DNA strands:

```
index 01234567890123456789012
     "GGGA-GAATCTCTGGACT"
     "CTCTACTTA-AGACCGGTACAGG"
```

This pair of strands has three point mutations (at indexes 1, 15, and 17), and seven unmatched nucleotides (dashes at indexes 4 and 9, and nucleotides in the second string with no match at indexes 18-22). The point mutations count as a total of 3 * 2 = 6 errors, and the unmatched nucleotides count as 7 * 1 = 7 errors, so your method would return an error count of 6+7 = 13 total errors if passed the two above strands.

You may assume that each string consists purely of the characters A, C, T, G, and - (the dash character), but the letters could appear in either upper or lowercase. The strings might be the same length, or the first or second might be longer than the other. Either string could be very long, very short, or even the empty string. If the strings match perfectly with no errors as defined above, your method should return `0`.

4.19. Start End Letter

Write a method named **startEndLetter** that accepts a character (`char`) as a parameter. The method repeatedly prompts the user to enter console input until the user types two consecutive words that both start and end with that letter; for example, the word "shells" starts and ends with 's'. The method then prints a message showing the last word typed.

Your code should be **case-insensitive**; for example, if the character passed is the lowercase `'t'`, you should also consider it a match if the user types a word that starts and ends with an uppercase `'T'`. If the user types a one-letter word, that word is considered to start and end with its single letter. For example, the word `"A"` starts and ends with `'a'`.

The following log represents the console output from a call to your method. (User input is shown **like this**.) Your method should exactly match the output structure and behavior shown below when given similar input.

```
startEndLetter('s');
Looking for two "s" words in a row.
Type a word: I
Type a word: love
Type a word: CS
Type a word: students
Type a word: PROGRAMS
Type a word: SCISSORS
Type a word: melon
Type a word: pens
Type a word: Q
Type a word: scores
Type a word: SOS
"s" is for "SOS"
```

Assumptions: You may assume that the parameter value passed will be a lowercase letter from `'a'` to `'z'` inclusive. You may also assume that the user will type a valid single-word response to each prompt and that the word will contain at least one character (the user will not type a blank line).

Constraints: You should not use any data structures such as arrays to help you solve this problem. You may declare as many simple variables such as ints or strings as you like. Be mindful of the differences between Java types `String` and `char` when solving this problem.

4.20. Word Count

Write a method named **wordCount** that accepts a string as its parameter and returns the number of words in the string. A word is a sequence of one or more non-space characters (any character other than ' '). For example, the call of `wordCount("hello, how are you?")` should return `4`.

Constraints: Do not use a `Scanner` to help you solve this problem. Do not use any data structures such as arrays to help you solve this problem. Do not use the `String` method `split` on this problem. But you can declare as many simple variables like `int`, `char`, etc. as you like. Declaring `String` variables is also fine.

4.21. Is Palindrome

Write a method named **isPalindrome** that accepts a string parameter and returns `true` if that string is a palindrome, or `false` if it is not a palindrome.

For this problem, a *palindrome* is defined as a string that contains exactly the same sequence of characters forwards as backwards, case-insensitively. For example, "madam" or "racecar" are palindromes, so the call of `isPalindrome("racecar")` would return `true`. Spaces, punctuation, and any other characters should be treated the same as letters; so a multi-word string such as `"dog god"` could be a palindrome, as could a gibberish string such as "123 $$ 321". The empty string and all one-character strings are palindromes by our definition. Your code should ignore case, so a string like `"Madam"` or `"RACEcar"` would also count as palindromes.

4.22. Reverse Chunks

Write a method named **reverseChunks** that accepts a string *s* and integer *k* as parameters and returns a new string that reverses the relative order of every *k* characters of *s*. For example, the call of `reverseChunks("MehranSahami", 3)` should view the string in groups of 3 characters at a time, reversing `"Meh"`into `"heM"`, and `"ran"` into `"nar"`, and so on, returning a result of `"heMnarhaSima"`.

If the string's length is not an exact multiple of *k*, the last chunk of fewer-than-*k* characters at the end of the string should be left in its original order. For example, if the call is `reverseChunks("MartyStepp", 4)`, the first chunk `"Mart"` becomes `"traM"` and the second chunk `"ySte"` becomes `"etSy"`. The last two characters, `"pp"`, are fewer than 4, so they are left as-is. So the result returned should be `"traMetSypp"`.

You may assume that the value passed for *k* will be a positive integer.

Constraints: You should not create any data structures such as arrays. But you may create as many strings as you like, and you may use as many simple variables (such as `int`s) as you like.

4.23. Crazy Caps

Write a method named `crazyCaps` that accepts a string as a parameter and returns a new string with its capitalization altered such that the characters at even indexes are all in lowercase and odd indexes are all in uppercase. For example, if a variable `s` stores "`Hey!! THERE!`", the call of `crazyCaps(s);` should return "`hEy!! tHeRe!`".

4.24. Name Diamond

Write a method named **nameDiamond** that accepts a string as a parameter and prints it in a "diamond" format as shown below. For example, the call of `nameDiamond("MARTY");` should print:

```
M
MA
MAR
MART
MARTY
 ARTY
  RTY
   TY
    Y
```

4.25. Swap Pairs

Write a method named `swapPairs` that accepts a string as a parameter and returns a new string such that each pair of adjacent letters will be reversed. If the string has an odd number of letters, the last letter is unchanged. For example, if a string variable `s` stores "`example`", the call of `swapPairs(s);` should return "`xemalpe`". If `s` had been "`hello there`", the call would produce "`ehll ohtree`".

5. ARRAYS

5.1. Multi-dimensional arrays

5.1.1. Mystery #1

Consider the following method:

```
public void arrayMystery2D(int[][] numbers) {
    for (int r = 0; r < numbers.length; r++) {
        for (int c = 0; c < numbers[0].length - 1; c++) {
            if (numbers[r][c + 1] > numbers[r][c]) {
                numbers[r][c] = numbers[r][c + 1];
            }
        }
    }
}
```

If a two-dimensional array numbers is initialized to the following values, what are its contents after the call of `arrayMystery2D(numbers);` ?

```
int[][] numbers = {
    {3, 4, 5, 6},
    {4, 5, 6, 7},
    {5, 6, 7, 8}
};
arrayMystery2D(numbers);
```

row 0	
row 1	
row 2	

5.1.2. Mystery #2

Consider the following method. For each multi-dimensional array listed below, write the final array state that would result if the given array were passed as a parameter to the method.

```
public void array2dMystery2(int[][] a) {
    for (int r = 0; r < a.length; r++) {
        for (int c = a[0].length - 2; c > 0; c--) {
            a[r][c] = a[r][c - 1] + a[r][c + 1];
        }
    }
}
```

int[][] a1 = {{2, 1, 6}, {5, 8, 9}, {1, -5, -4}};	
int[][] a2 = {{1, 2, 3, 4, 5}, {2, 4, 6, 8, 10}, {0, 1, 2, 3, 4}};	

5.1.3. Mystery #3

Consider the following method. For each multi-dimensional array listed below, write the final array state that would result if the given array were passed as a parameter to the method.

```
public void array2dMystery3(int[][] a) {
    for (int r = 0; r < a.length - 1; r++) {
        for (int c = 0; c < a[0].length - 1; c++) {
            if (a[r][c + 1] > a[r][c]) {
                a[r][c] = a[r][c + 1];
            } else if (a[r + 1][c] > a[r][c]) {
                a[r][c] = a[r + 1][c];
            }
        }
    }
}
```

`int[][] a1 = {{3, 4, 5, 6}, {4, 2, 6, 1}, {1, 6, 7, 2}};`	
`int[][] a2 = {{1, 2, 3, 0, 5}, {2, 4, 6, 8, 10}, {9, 5, 1, 2, 4}};`	

5.1.4. Mystery #4

Consider the following method. For each multi-dimensional array listed below, write the final array state that would result if the given array were passed as a parameter to the method.

```
public void array2dMystery4(int[][] a) {
    for (int r = 1; r < a.length - 1; r++) {
        for (int c = 1; c < a[0].length - 1; c++) {
            int sum1 = a[r - 1][c - 1] + a[r + 1][c + 1];
            int sum2 = a[r - 1][c + 1] + a[r + 1][c - 1];
            if (sum1 > sum2) {
                a[r][c] = sum1;
            } else {
                a[r][c] = sum2;
            }       }    }}
```

`int[][] a1 = {`
`{3, 4, 5, 6, 2},`
`{4, 2, 6, 1, 3},`
`{5, 7, 4, 9, 1},`
`{1, 6, 7, 2, 8}`
`};`	

`int[][] a2 = {`
`{0, 1, 0, 1, 0, 1},`
`{2, 1, 2, 1, 2, 1},`
`{0, 5, 0, 5, 0, 5},`
`{3, 2, 3, 2, 3, 2},`
`{1, 4, 1, 4, 1, 4}`
`};`	

5.1.5. Is Unique

Write a method named **isUnique** that accepts a 2-D array of integers as a parameter and that returns a `boolean` value indicating whether or not the values in the array are unique (`true` for yes, `false` for no). The values in the list are considered unique if there is no pair of values that are equal. For example, if a variable called `matrix` stores the following values:

```
int[][] matrix = {{3, 8, 12}, {2, 9, 17}, {43, -8, 46}, {203, 14, 97}};
```

Then the call of `isUnique(matrix)` should return `true` because there are no duplicated values in this array. If instead the array stored these values:

```
int[][] matrix2 = {{4, 7, 2}, {3, 9, 12}, {-47, -19, 308}, {3, 74, 15}};
```

Then the call should return `false` because the value 3 appears twice in this list. Notice that given this definition, an array of 0 or 1 elements would be considered unique. Your code should work for an array of any size, even one with 0 rows or columns.

Constraints: You may use one auxiliary data structure of your choice to help you solve this problem. Your method should not modify the array that is passed in.

5.1.6. Matrix Sum

Write a method named `matrixSum` that accepts as parameters two 2D arrays of integers, treats the arrays as 2D matrices and adds them, returning the result. The sum of two matrices A and B is a matrix C where for every row i and column j, $C_{ij} = A_{ij} + B_{ij}$. For example, if A is *{{1, 2, 3}, {4, 4, 4}}* and B is *{{5, 5, 6}, {0, -1, 2}}*, the call of `matrixSum(a, b)` should return *{{6, 7, 9}, {4, 3, 6}}*. You may assume that the arrays passed as parameters have the same dimensions.

5.2. Mystery #1

What are the values of the elements in array `a1` after the following code executes?

```
public void mystery1(int[] a1, int[] a2) {
    for (int i = 0; i < a1.length; i++) {
        a1[i] += a2[a2.length - i - 1];
    }
}
int[] a1 = {1, 3, 5, 7, 9};
int[] a2 = {1, 4, 9, 16, 25};
mystery1(a1, a2);
```

a1[0]	
a1[1]	
a1[2]	
a1[3]	
a1[4]	

5.3. Mystery #2

Consider the following method:

```
public void arrayMystery2(int[] a) {
    for (int i = 1; i < a.length - 1; i++) {
        a[i] = a[i - 1] - a[i] + a[i + 1];
    }
}
```

Indicate in the right-hand column what values would be stored in the array after the method arrayMystery executes if each integer array below is passed as a parameter to it.

int[] a1 = {42, 42};
arrayMystery2(a1);	

int[] a2 = {6, 2, 4};
arrayMystery2(a2);	

int[] a3 = {7, 7, 3, 8, 2};
arrayMystery2(a3);	

int[] a4 = {4, 2, 3, 1, 2, 5};
arrayMystery2(a4);	

int[] a5 = {6, 0, -1, 3, 5, 0, -3};
arrayMystery2(a5);	

5.4. Mystery #3

Consider the following method:

```
public void mystery(int[] list) {
    for (int i = 1; i < list.length; i++) {
        list[i] = list[i] + list[i - 1];
    }
}
```

Indicate in the right-hand column what values would be stored in the array after the method mystery executes if the integer array in the left-hand column is passed as a parameter to mystery.

Be sure to enter your array in correct format, and entering elements with their appropriate type. For example, for an int[] holding the elements 1, 2, and 5, this would be entered as: {1, 2, 5}

{8}	
{6, 3}	
{2, 4, 6}	
{1, 2, 3, 4}	
{7, 3, 2, 0, 5}	

5.5. Mystery #4

Consider the following method:

```
public void mystery2(int[] list) {
    for (int i = 0; i < list.length - 1; i++) {
        if (i % 2 == 0) {
            list[i]++;
        } else {
            list[i]--;
        }
    }
}
```

For each array below, indicate what the array's contents would be after the method mystery were called and passed that array as its parameter.

{6, 3}	
{2, 4, 6}	
{1, 2, 3, 4}	
{2, 2, 2, 2, 2}	
{7, 3, 2, 0, 5, 1}	

5.6. Mystery #5

Consider the following method:

```
public void mystery3(int[] nums) {
    for (int i = 0; i < nums.length - 1; i++) {
        if (nums[i] > nums[i + 1]) {
            nums[i + 1]++;
        }
    }
}
```

For each array below, indicate what the array's contents would be after the method mystery were called and passed that array as its parameter.

{42}	
{14, 7}	
{7, 1, 3, 2, 0, 4}	
{10, 8, 9, 5, 5}	
{12, 11, 10, 10, 8, 7}	

5.7. Mystery #6

Write the output produced by the following method when passed each of the following arrays:

```
public void arrayMysteryExam2(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        if (a[i] > a[i + 1]) {
            int temp = a[i];
            a[i] = a[i + 1];
            a[i + 1] = temp;
            a[0]++;
        }
    }
    println(Arrays.toString(a));
}
```

{5, 2, 5, 2}	
{30, 10, 20, 50, 40}	
{99, 88, 77, 66, 55, 44}	

5.8. Mystery #7

Write the output produced by the following method when passed each of the following arrays:

```
public void arrayMysteryExam4(int[] a) {
    for (int i = 1; i < a.length; i++) {
        a[i] = a[a.length - 1 - i] - a[i - 1];
    }
    println(Arrays.toString(a));
}
```

{6, 2, 4}	
{4, 2, 3, 1, 2, 5}	
{6, 0, -1, 3, -2, 0, 4}	

5.9. Mystery #8

Write the output produced by the following method when passed each of the following arrays:

```
public void arrayMysteryExam6(int[] a) {
    for (int i = 1; i < a.length - 1; i++) {
        a[i] = a[i - 1] + a[i + 1];
        if (a[i] % 2 == 0) {
            a[i] = a[i] / 2;
        }
    }
    println(Arrays.toString(a));
}
```

{1, 3, 4, 6}	
{8, 13, 0, 6, 11, 2}	

5.10. Mystery #9

The following program produces 4 lines of output. Write each line of output below as it would appear on the console.

```
public class ReferenceMystery1 extends ConsoleProgram {
    public void run() {
        int y = 1;
        int x = 3;
        int[] a = new int[4];

        mystery(a, y, x);
        println(x + " " + y + " " + Arrays.toString(a));

        x = y - 1;
        mystery(a, y, x);
        println(x + " " + y + " " + Arrays.toString(a));
    }

    public void mystery(int[] a, int x, int y) {
        if (x < y) {
            x++;
            a[x] = 17;
        } else {
            a[y] = 17;
        }
        println(x + " " + y + " " + Arrays.toString(a));
    }
}
```

line 1	
line 2	
line 3	
line 4	

5.11. Mystery #10

The following program produces 4 lines of output. Write each line of output below as it would appear on the console.

```
public class ReferenceMystery2 extends ConsoleProgram {
    public void run() {
        int x = 1;
        int[] a = new int[4];

        x = x * 2;
        mystery(x, a);
        println(x + " " + Arrays.toString(a));

        x = x * 2;
        mystery(x, a);
        println(x + " " + Arrays.toString(a));
    }

    public void mystery(int x, int[] a) {
        x = x * 2;

        if (x > 6) {
            a[2] = 14;
            a[1] = 9;
        } else {
            a[0] = 9;
            a[3] = 14;
        }
        println(x + " " + Arrays.toString(a));
    }
}
```

line 1	
line 2	
line 3	
line 4	

5.12. Mystery #11

The following program produces 4 lines of output. Write each line of output below as it would appear on the console.

```
public class ReferenceMystery4 extends ConsoleProgram {
    public void run() {
        int[] a = {2, 0, 1};
        int b = 3;
        mystery(a, b, a[0]);
        println(Arrays.toString(a) + " " + b);

        b = a[0] + a[1] + a[2];
        mystery(a, a[1], a[2]);
        println(Arrays.toString(a) + " " + b);
    }

    public void mystery(int[] a, int b, int c) {
        for (int i = 0; i < a.length; i++) {
            a[i] = a[i] * 2;
        }
        b++;
        c--;
        println(Arrays.toString(a) + " " + b + " " + c);
    }
}
```

line 1	
line 2	
line 3	
line 4	

5.13. Mystery #12

The following program produces 4 lines of output. Write each line of output below as it would appear on the console.

```
public class ReferenceMystery5 extends ConsoleProgram {
    public void run() {
        int[] a = new int[2];
        a[0] = 10;
        a[1] = 20;
        int b = 3;
        int c = 5;

        mystery(a, b, c);
        println(Arrays.toString(a) + " " + b + " " + c);

        a[1]++;
        mystery(a, a[0], b);
        println(Arrays.toString(a) + " " + b + " " + c);
    }

    public void mystery(int[] a, int b, int c) {
        b = b + c;
        for (int i = 0; i < a.length; i++) {
            a[i]++;
        }
        c = c + a[0];
        println(Arrays.toString(a) + " " + b + " " + c);
    }
}
```

line 1	
line 2	
line 3	
line 4	

5.14. Squared Array

Write a variable declaration and `for` loop necessary to create and initialize an integer array named `squares` that contains the following values:

0 1 4 9 16 25 36 49 64 81 100

5.15. Print

Write a method named **print** that accepts an array of integers as a parameter and prints them, one per line, in the format shown. Your code should work for an array of any size. For example, if an array named `a` contains the elements `[32, 5, 27, -3, 2598]`, then the call of `print(a);` should produce the following output:

```
element [0] is 32
element [1] is 5
element [2] is 27
element [3] is -3
element [4] is 2598
```

5.16. Max Value

Write a method named `maxValue` that accepts an array of integers as a parameter and returns the maximum value in the array. For example, if an array named a passed stores {17, 7, -1, 26, 3, 9}, the call of maxValue(a) should return 26. You may assume that the array contains at least one element. Your method should not modify the elements of the array.

5.17. Find Range

Write a method named **findRange** that accepts an array of integers as a parameter and returns the range of values contained in the array, which is equal to one more than the difference between its largest and smallest element. For example, if the largest element is 17 and the smallest is 6, the range is 12. If the largest and smallest values are the same, the range is 1.

Constraints: You may assume that the array contains at least one element (that its length is at least 1). You should not modify the contents of the array.

5.18. Get Percent Even

Write a method named getPercentEven that accepts an array of integers as a parameter and returns the percentage of the integers in the array that are even numbers. For example, if an array a stores {6, 4, 9, 11, 5}, then your method should return 40.0 representing **40%** even numbers. If the array contains no even elements or is empty, return 0.0. Do not modify the array passed in.

5.19. Compute Average

Write a method named **computeAverage** that computes and returns the mean of all elements in an array of integers. For example, if an array named acontains [1, -2, 4, -4, 9, -6, 16, -8, 25, -10], then the call of computeAverage(a) should return 2.5.

Constraints: You may assume that the array contains at least one element. Your method should not modify the elements of the array.

5.20. Sorted

Write a method named **sorted** that accepts an array of doubles as a parameter and returns true if the list is in sorted order and false if not. For example, if an array named list stores {1.5, 4.3, 7.0, 19.5, 25.1, 46.2} respectively, the call of sorted(list2) should return true. You may assume the array has at least one element. A one-element array is considered to be sorted. Do not modify the array passed in.

5.21. Count Duplicates

Write a method named **countDuplicates** that accepts an array of integers as a parameter and that returns the number of duplicate values in the array. A duplicate value is a value that also occurs earlier in the array. For example, if an array named a contains [1, 4, 2, 4, 7, 1, 1, 9, 2, 3, 4, 1], then the call of countDuplicates(a) should return 6 because there are three duplicates of the value 1, one duplicate of the value 2, and two duplicates of the value 4.

Constraints: The array could be empty or could contain only a single element; in such cases, your method should return 0. Do not modify the contents of the array.

5.22. Switch Pairs

Write a method named **switchPairs** that accepts an array of strings as a parameter and switches the order of pairs of values in the array. Your method should swap the order of the first two values, then switch the order of the next two, and so on. For example, if the array stores:

```
String[] a = {"a", "bb", "c", "ddd", "ee", "f", "g"};
switchPairs(a);
```

Your method should switch the first pair ("a", "bb"), the second pair ("c", "ddd") and the third pair ("ee", "f"), to yield this array:

```
{"bb", "a", "ddd", "c", "ee", "f", "g"}
```

If there are an odd number of values, the final element is not moved.

You may assume that the array is not null and that no element of the array is null.

5.23. Split

Write a method named **split** that accepts an array of integers as a parameter and returns a new array twice as large as the original, replacing every integer from the original array with a pair of integers, each half the original. If a number in the original array is odd, then the first number in the new pair should be one higher than the second so that the sum equals the original number. For example, if a variable named a refers to an array storing the values {18, 7, 4, 24, 11}, the call of split(a) should return a new array containing {9, 9, 4, 3, 2, 2, 12, 12, 6, 5}. (The number 18 is split into the pair 9, 9, the number 7 is split into 4, 3, the number 4 is split into 2, 2, the number 24 is split into 12, 12 and the number 11 is split into 6, 5.)

5.24. Has Mirror Twice

Write a method named **hasMirrorTwice** that accepts two arrays of integers *a1* and *a2* as parameters and returns true if *a1* contains all the elements of *a2*in reverse order at least twice (and false otherwise). For example, if *a2* stores the elements {1, 2, 3} and *a1* stores the elements {6, <u>3, 2, 1</u>, 4, 1, <u>3, 2, 1</u>, 5}, your method would return true.

Assume that both arrays passed to your method will have a length of at least 1. This means that the shortest possible mirror will be of length 1, representing a single element (which is its own mirror). A sequence that is a palindrome (the same forwards as backwards) is considered its own mirror and should be included in your computations. For example, if *a1* is `{6, 1, 2, 1, 4, 1, 2, 1, 5}` and *a2* is `{1, 2, 1}`, your method should return `true`. The two occurrences of the mirror might overlap, as shown in the fourth sample call below.

The following table shows some calls to your method and their expected results:

Arrays	Returned Value

`int[] a1 = {6, 1, 2, 1, 3, 1, 3, 2, 1, 5};`
`int[] a2 = {1, 2};`
	`hasMirrorTwice(a1, a2)` returns `true`

`int[] a3 = {5, 8, 4, 18, 5, 42, 4, 8, 5, 5};`
`int[] a4 = {4, 8, 5};`
	`hasMirrorTwice(a3, a4)` returns `false`

`int[] a5 = {6, 3, 42, 18, 12, 5, 3, 42, 3, 42};`
`int[] a6 = {42, 3};`
	`hasMirrorTwice(a5, a6)` returns `true`

`int[] a7 = {6, 1, 2, 4, 2, 1, 2, 4, 2, 1, 5};`
`int[] a8 = {1, 2, 4, 2, 1};`
	`hasMirrorTwice(a7, a8)` returns `true`

`int[] a9 = {0, 0};`
`int[] aa = {0};`
	`hasMirrorTwice(a9, aa)` returns `true`

`int[] ab = {8, 9, 2, 1};`
`int[] ac = {5, 7, 1, 2, 9, 8};`
	`hasMirrorTwice(ab, ac)` returns `false`

Do not modify the contents of the arrays passed to your method as parameters.

5.25. Longest Sorted Sequence

Write a method named **longestSortedSequence** that accepts an array of integers as a parameter and that returns the length of the longest sorted (nondecreasing) sequence of integers in the array. For example, if a variable named `array` stores the following values:

```
int[] array = {3, 8, 10, 1, 9, 14, -3, 0, 14, 207, 56, 98, 12};
```

Then the call of `longestSortedSequence(array)` should return `4` because the longest sorted sequence in the array has four values in it (the sequence -3, 0, 14, 207). Notice that sorted means nondecreasing, which means that the sequence could contain duplicates. For example, if the array stores the following values:

```
int[] array2 = {17, 42, 3, 5, 5, 5, 8, 2, 4, 6, 1, 19}
```

Then the method would return `5` for the length of the longest sequence (the sequence 3, 5, 5, 5, 8). Your method should return `0` if passed an empty array. Your method should return `1` if passed an array that is entirely in decreasing order or contains only one element.

Constraints: You may not use any auxiliary data structures (arrays, lists, strings, etc.) to solve this problem. Your method should not modify the array that is passed in.

5.26. Contains

Write a method named **contains** that accepts two arrays of integers *a1* and *a2* as parameters and that returns a boolean value indicating whether or not *a2*'s sequence of elements appears in *a1* (`true` for yes, `false` for no). The sequence of elements in *a2* may appear anywhere in *a1* but must appear consecutively and in the same order. For example, if variables called `a1` and `a2` store the following values:

```
int[] a1 = {1, 6, 2, 1, 4, 1, 2, 1, 8};
int[] a2 = {1, 2, 1};
```

Then the call of `contains(a1, a2)` should return `true` because `a2`'s sequence of values `{1, 2, 1}` is contained in `a1` starting at index `5`. If `a2` had stored the values `{2, 1, 2}`, the call of `contains(a1, a2)` would return `false` because `a1` does not contain that sequence of values. Any two arrays with identical elements are considered to contain each other, so a call such as `contains(a1, a1)` should return `true`.

You may assume that both arrays passed to your method will have lengths of at least 1. You may not use any `String`s to help you solve this problem, nor methods that produce `String`s such as `Arrays.toString`.

5.27. Even Before Odd

Write a method named **evenBeforeOdd** that accepts an array of integers as a parameter and rearranges its elements so that all even values appear before all odds. For example, if the following array is passed to your method:

```
int[] numbers = {5, 2, 4, 9, 3, 6, 2, 1, 11, 1, 10, 4, 7, 3};
```

Then after the method has been called, one acceptable ordering of the elements would be:

```
{4, 2, 4, 10, 2, 6, 3, 1, 11, 1, 9, 5, 7, 3}
```

The exact order of the elements does not matter, so long as all even values appear before all odd values. For example, the following would also be an acceptable ordering:

```
{2, 2, 4, 4, 6, 10, 1, 1, 3, 3, 5, 7, 9, 11}
```

Do not make any assumptions about the length of the array or the range of values it might contain. For example, the array might contain no even elements or no odd elements.

You should not use any temporary arrays to help you solve this problem. (But you may declare as many simple variables as you like, such as ints.) You may not use any other data structures such as Strings orArrayLists. You should not use Arrays.sort in your solution.

5.28. Index Of

Write a method named **indexOf** that returns the index of a particular value in an array of integers. The method should return the index of the first occurrence of the target value in the array. If the value is not in the array, it should return -1. For example, if an array called list stores the following values:

```
int[] list = {42, 7, -9, 14, 8, 39, 42, 8, 19, 0};
```

Then the call indexOf(list, 8) should return 4 because the index of the first occurrence of value 8 in the array is at index 4. The call indexOf(list, 2)should return -1 because value 2 is not in the array.

5.29. Second Index Of

Write a method named **secondIndexOf** that returns the index of the second occurrence of a particular value in an array of integers. If the value does not appear in the array at least twice, you should return -1. For example, if an array called list stores the following values:

```
     // index  0  1   2   3  4   5   6  7   8  9  10
int[] list = {42, 7, -9, 14, 8, 39, 42, 8, 19, 0, 42};
```

Then the call secondIndexOf(list, 42) should return 6 because the index of the second occurrence of value 42 in the array is at index 6. The call secondIndexOf(list, 14) should return -1 because value 14 does not occur at least twice in the array.

5.30. Num Unique

Write a method named **numUnique** that accepts a sorted array of integers as a parameter and that returns the number of unique values in the array. The array is guaranteed to be in sorted order, which means that duplicates will be grouped together. For example, if a variable called list stores the following values:

```
int[] list = {5, 7, 7, 7, 8, 22, 22, 23, 31, 35, 35, 40, 40, 40, 41};
```

Then the call of numUnique(list) should return 9 because this list has 9 unique values (5, 7, 8, 22, 23, 31, 35, 40 and 41). It is possible that the list might not have any duplicates. For example if list instead stored this sequence of values:

```
int[] list = {1, 2, 11, 17, 19, 20, 23, 24, 25, 26, 31, 34, 37, 40, 41};
```

Then a call on the method would return 15 because this list contains 15 different values.

If passed an empty list, your method should return 0. Remember that you can assume that the values in the array appear in sorted (nondecreasing) order.

5.31. Collapse

Write a method named **collapse** that accepts an array of integers as a parameter and returns a new array where each pair of integers from the original array has been replaced by the sum of that pair. For example, if an array called a stores {7, 2, 8, 9, 4, 13, 7, 1, 9, 10}, then the call of collapse(a)should return a new array containing {9, 17, 17, 8, 19}. The first pair from the original list is collapsed into 9 (7 + 2), the second pair is collapsed into 17 (8 + 9), and so on.

If the list stores an odd number of elements, the final element is not collapsed. For example, if the array had been {1, 2, 3, 4, 5}, then the call would return {3, 7, 5}. Your method should not change the array that is passed as a parameter.

5.32. Find Median

Write a method named **findMedian** that accepts an array of integers as its parameter and returns the median of the numbers in the array. The median is the number that will appear in the middle if you arrange the elements in order. Assume that the array is of odd size (so that one sole element constitutes the median) and that the numbers in the array are between 0 and 99 inclusive.

For example, the median of {5, 2, 4, 17, 55, 4, 3, 26, 18, 2, 17} is 5, and the median of {42, 37, 1, 97, 1, 2, 7, 42, 3, 25, 89, 15, 10, 29, 27} is 25.

5.33. Banish

Write a method named **banish** that accepts two arrays of integers *a1* and *a2* as parameters and removes all occurrences of *a2*'s values from *a1*. An element is "removed" by shifting all subsequent elements one index to the left to cover it up, placing a 0 into the last index. The original relative ordering of *a1*'s elements should be retained.

For example, suppose the following two arrays are declared and the following call is made:

```
int[] a1 = {42, 3, 9, 42, 42, 0, 42, 9, 42, 42, 17, 8, 2222, 4, 9, 0, 1};
int[] a2 = {42, 2222, 9};
banish(a1, a2);
```

After the call has finished, the contents of a1 should become:

```
{3, 0, 17, 8, 4, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

Notice that all occurrences of the values 42, 2222, and 9 have been removed and replaced by 0s at the end of the array, and the remaining values have shifted left to compensate.

Do not make any assumptions about the length of the arrays or the ranges of values each might contain. For example, each array might contain no elements or just one element, or very many elements. (If *a2* is an empty array that contains no elements, *a1* should not be modified by the call to your method.) You may assume that the arrays passed are not `null`. You may assume that the values stored in *a2* are unique and that *a2* does not contain the value `0`.

You may not use any temporary arrays to help you solve this problem. (But you may declare as many simple variables as you like, such as `int`s.) You also may not use any other data structures or complex types such as `String`s, or other data structures such as the `ArrayList` class.

5.34. Sieve

Write a complete console program named `Sieve` that uses the "Sieve of Eratosthenes" algorithm to print a list of prime numbers between 2 and a given maximum. You will represent the numbers using an **array**.

In the third century B.C., the Greek astronomer Eratosthenes developed an algorithm for finding all the prime numbers up to some upper limit *N*. To apply the algorithm, you start by writing down a list of the integers between 2 and *N*. For example, if *N* is 10, you would write down the following list:

2 3 4 5 6 7 8 9 10

You then underline the first number in the list and cross off every multiple of that number. Thus, after executing the first step of the algorithm, you will underline 2 and cross off every multiple of 2:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~

From here, you simply repeat the process: underline the first number in the list that is neither crossed nor underlined, and then cross off its multiples. Eventually, every number in the list will either be underlined or crossed out, as shown below. The underlined numbers are prime.

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~

Your program should prompt the user to enter a max value *N*, and then perform the Sieve of Eratosthenes algorithm on the range of numbers 2 through *N*inclusive. You may assume that the user types a number that is at least 2. Here is an example output from one run of your program, with user input shown **like this**:

```
Max value N? 100
Primes: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

You must use an **array** in your solution to this problem. Part of your task in this problem is figuring out how to use an array to represent the list of numbers and how to remember which ones are prime, which ones are "underlined" vs "crossed off," and so on.

5.35. Count Unique

Write a method named **countUnique** that accepts an array of integers as a parameter and returns a count of the number of unique values that occur in the array. If the array contains multiple occurrences of the same element value, only one of those occurrences should count toward your total. For example, if an array named `numbers` stores `{7, 7, 2, 2, 1, 2, 2, 7}`, the call of `countUnique(numbers)` should return `3` because there are 3 unique values: 7, 2, and 1.

Constraints: In solving this problem, do not create any other data structures such as arrays, strings, etc., though you may create as many simple variables (e.g. `int`, `double`) as you like. **Do not modify** the array passed to your method as the parameter, such as by sorting or rearranging its element values.

5.36. Collapse Pairs

Write a method named **collapsePairs** that accepts an array of integers as a parameter and modifies the array so that each of its pairs of neighboring integers (such as the pair at indexes 0-1, and the pair at indexes 2-3, etc.) are combined into a single sum of that pair. The sum will be stored at the even index (0,2,4, etc.) if the sum is even and at the odd index (1,3,5, etc.) if the sum is odd. The other index of the pair will change to 0.

For example, if an array named `a` stores the values `{7, 2, 8, 9, 4, 22, 7, 1, 9, 10}`, then the call of `collapsePairs(a);` should modify the array to contain the values `{0, 9, 0, 17, 26, 0, 8, 0, 0, 19}`. The first pair from the original array is collapsed into 9 (7 + 2), which is stored at the odd index 1 because 9 is odd. The second pair is collapsed into 17 (8 + 9), stored at the odd index 3; the third pair is collapsed into 26 (4 + 22), stored at the even index 4; and so on. The figure below summarizes the process for this example array:

```
before: index  0   1   2   3   4   5   6   7   8   9
        value {7,  2,  8,  9,  4, 22,  7,  1,  9, 10}

               \   /   \   /   \   /   \   /   \   /
                \ /     \ /     \ /     \ /     \ /
                 V       V       V       V       V

after:  index  0   1   2   3   4   5   6   7   8   9
        value {0,  9,  0, 17, 26,  0,  8,  0,  0, 19}
```

5.37. n Copies

Write a method named **nCopies** that accepts an array of integers *a* as a parameter and returns a new array *a2*, with each element value *n* from *a* replaced by *n* consecutive copies of the value *n* at the same relative location in the array. For example, if an array named `a` stores the following element values:

```
{3, 5, 0, 2, 2, -7, 0, 4}
```

Then the call of `int[] a2 = nCopies(a);` should return a new array `a2` containing the following elements. The idea is that the value 3 was replaced by three 3s; the 5 was replaced by five 5s; and so on.

```
{3, 3, 3, 5, 5, 5, 5, 5, 2, 2, 2, 2, 4, 4, 4, 4}
```

Any element whose value is 0 or negative should not be kept in the returned array (as with 0 and -7 above).

The array you return must have a **length** that is exactly long enough to fit its elements. For example, the result array `a2` above contains 3 + 5 + 2 + 2 + 4 = 16 total elements, so the returned array's length must be exactly 16 in that case.

Constraints: In solving this problem, you must create a single new array to be returned, but aside from that, do not create any other data structures such as temporary arrays or strings. You may use as many simple variables (such as `int`s) as you like.

5.38. Remove Palindromes

Write a method named **removePalindromes** that that removes all strings that are palindromes from an array of strings.

Your method accepts an array of strings as a parameter and modifies its contents, replacing every string in the array that is a palindrome with an empty string, "". For example, if an array named `a` stores the following element values:

```
String[] a = {"Madam", "raceCAR", "", "hi", "A", "Abba", "banana", "dog God",
              "STOP otto POTS", "Madame", "Java", "LevEL", "staTS"};
```

Then the call of `removePalindromes(a);` should change it to contain the following element values. Notice that the palindromes from the array such as `"Madam"` and `"LevEL"` have been replaced by "".

```
{"", "", "", "hi", "", "banana", "", "", "Madame", "Java", "", ""}
```

Constraints: You are to modify the existing array in-place. Do not create any other data structures such as temporary arrays. You may create as many strings as you like, and you may use as many simple variables (such as `int`s) as you like.

Note: You may want to go solve the string problem isPalindrome first and use it as part of your solution to this problem.

6. COLLECTIONS

6.1. ArrayList

6.1.1. Mystery #1

Write the output produced by the method below when passed each of the following ArrayLists:

```
public void mystery1(ArrayList<Integer> a) {
    for (int i = a.size() - 1; i > 0; i--) {
        if (a.get(i) < a.get(i - 1)) {
            int n = a.get(i);
            a.remove(i);
            a.add(0, n);
        }
    }
    println(a);
}
```

[2, 6, 1, 8]	
[10, 30, 40, 20, 60, 50]	
[-4, 16, 9, 1, 64, 25, 36, 4, 49]	

6.1.2. Mystery #2

Write the final contents when the following method is passed each list below:

```
public void arrayListMystery1(ArrayList<Integer> v) {
    for (int i = 0; i < v.size(); i++) {
        int n = v.get(i);
        if (n % 10 == 0) {
            v.remove(i);
            v.add(n);
        }
    }
    println(v);
}
```

{1, 20, 3, 40}	
{80, 3, 40, 20, 7}	
{40, 20, 60, 1, 80, 30}	

6.1.3. Mystery #3

Write the output produced by the following method when passed each of the following lists:

```
public void mystery(ArrayList<Integer> list) {
    for (int i = 0; i < list.size(); i++) {
        int n = list.get(i);
        if (n % 10 == 0) {
            list.remove(i);
            list.add(n);
        }
    }
    println(list);
}
```

[1, 20, 3, 40]	
[80, 3, 40, 20, 7]	
[40, 20, 60, 1, 80, 30]	

6.1.4. Mystery #4

What are the contents of the list after the following code executes?

```
ArrayList<Integer> vec = new ArrayList<Integer>();
for (int i = 1; i <= 5; i++) {
    vec.add(2 * i);         // {2, 4, 6, 8, 10}
}
int size = vec.size();
for (int i = 0; i < size; i++) {
    vec.add(i, 42);    // add 42 at index i
}
println(vec);
```

Lists content:

A	{2, 4, 6, 8, 10, 42, 42, 42, 42, 42}
B	{42, 42, 42, 42, 42, 2, 4, 6, 8, 10}
C	{42, 2, 42, 4, 42, 6, 42, 8, 42, 10}
D	{2, 42, 4, 42, 6, 42, 8, 42, 10, 42}
E	other

6.1.5. Mystery #5

What are the contents of the list after the following code executes?

```
ArrayList<Integer> vec = new ArrayList<Integer>();
for (int i = 1; i <= 8; i++) {
    vec.add(10 * i);   //   0   1   2   3   4   5   6   7
}                      // {10, 20, 30, 40, 50, 60, 70, 80}
for (int i = 0; i < vec.size(); i++) {
    vec.remove(i);
}
println(vec);
```

Lists content:

A	{2, 4, 6, 8, 10, 42, 42, 42, 42, 42}
B	{42, 42, 42, 42, 42, 2, 4, 6, 8, 10}
C	{42, 2, 42, 4, 42, 6, 42, 8, 42, 10}
D	{2, 42, 4, 42, 6, 42, 8, 42, 10, 42}
E	other

6.1.6. Mystery #6

Write the output produced by the following method when passed each of the following lists:

```
public void collectionMystery1(ArrayList<Integer> list) {
    for (int i = 0; i < list.size(); i++) {
        int n = list.get(i);
        list.remove(i);
        if (n % 2 == 0) {
            list.add(i);
        }
    }
    println(list);
}
```

{5, 2, 5, 2}	
{3, 5, 8, 9, 2}	
{0, 1, 4, 3, 1, 3}	

6.1.7. Mystery #7

Write the output produced by the following method when passed each of the following lists:

```
public void collectionMystery4(ArrayList<Integer> v) {
    for (int i = 1; i < v.size(); i += 2) {
        if (v.get(i - 1) >= v.get(i)) {
            v.remove(i);
            v.add(0, 0);
        }
    }
    println(v);
}
```

{10, 20, 10, 5}	
{8, 2, 9, 7, -1, 55}	
{0, 16, 9, 1, 64, 25, 25, 14, 0}	

6.1.8. Remove Even Length

Write a method named **removeEvenLength** that accepts an `ArrayList` of strings as a parameter and that removes all of the strings of even length from the list. For example, if an `ArrayList` variable named `list` contains the values `["hi", "there", "how", "is", "it", "going", "good", "sirs"]`, the call of `removeEvenLength(list);` would change it to store `["there", "how", "going"]`.

6.1.9. Mirror

Write a method named **mirror** that accepts an `ArrayList` of strings as a parameter and produces a mirrored copy of the list as output, with the original values followed by those same values in the opposite order. For example, if an `ArrayList` variable named `list`contains the values `["a", "b", "c"]`, after a call of `mirror(list);` it should contain `["a", "b", "c", "c", "b", "a"]`.

You may assume that the list is not `null` and that no element of the array is `null`.

6.1.10. Switch Pairs

Write a method named **switchPairs** that accepts an `ArrayList` of strings as a parameter and switches the order of pairs of values in the array. Your method should swap the order of the first two values, then switch the order of the next two, and so on. For example, if the array stores {"a", "bb", "c", "ddd", "ee", "f", "g"}, then the call of `switchPairs(a);` should switch the first pair ("a", "bb"), the second pair ("c", "ddd") and the third pair ("ee", "f"), to yield this list:

```
{"bb", "a", "ddd", "c", "ee", "f", "g"}
```

If there are an odd number of values, the final element is not moved. You may assume that the list is not `null` and that no element of the array is `null`.

6.1.11. Twice

Write a method named **twice** that accepts an `ArrayList` of strings as a parameter and that appends a second occurrence of the entire list to itself. For example, if an `ArrayList` named `list` stores the values `{"how", "are", "you?"}`, the call of `twice(list);` should modify the list to store `{"how", "are", "you?", "how", "are", "you?"}`.

6.1.12. Repeat

Write a method named **repeat** that accepts an `ArrayList` of `String`s and an integer k as parameters and that replaces each element with `k` copies of that element. For example, if the list stores the values `["how", "are", "you?"]` before the method is called and k is 4, it should store the values `["how", "how", "how",`

"how", "are", "are", "are", "are", "you?", "you?", "you?", "you?"] after the method finishes executing. If k is 0 or negative, the list should be empty after the call.

6.1.13. Remove Range

Write a method named **removeRange** that accepts an ArrayList of integers and two integer values min and max as parameters and removes all elements values in the range min through max (inclusive). For example, if an ArrayList named list stores [7, 9, 4, 2, 7, 7, 5, 3, 5, 1, 7, 8, 6, 7], the call of removeRange(list, 5, 7); should change the list to store [9, 4, 2, 3, 1, 8].

6.1.14. Delete Duplicates

Write a method named **deleteDuplicates** that accepts as a parameter a sorted ArrayList of Strings and that removes any duplicate values from the list. For example, suppose that an ArrayList named list contains the values {"be", "be", "is", "not", "or", "question", "that", "the", "to", "to"} After calling deleteDuplicates(list); the list should store the following values: {"be", "is", "not", "or", "question", "that", "the", "to"} You should assume that the values in the list are sorted, and that therefore all of the duplicates will be grouped together.

6.1.15. Unique Names

Write a console program named **UniqueNames** that asks the user for a list of names (one per line) until the user enters a blank line (i.e., just presses Enter when asked for a name). At that point the program should print out the list of names entered, where each name is listed only once (i.e., uniquely) no matter how many times the user entered the name in the program. For example, your program should behave as follows:

```
Enter name: Alice
Enter name: Bob
Enter name: Alice
Enter name: Alice
Enter name: Alice
Enter name: Bob
Unique name list contains: Alice Bob
```

6.1.16. Word Count

Write a console program named **WordCount** that reads a file and reports how many lines, words, and characters appear in it. Suppose, for example, that the file lear.txt contains the following passage from Shakespeare's King Lear:

```
Poor naked wretches, wheresoe'er you are,
That bide the pelting of this pitiless storm,
How shall your houseless heads and unfed sides,
Your loop'd and window'd raggedness, defend you
```

```
From seasons such as these?  O, I have ta'en
Too little care of this!
```

Given this file, your program should be able to generate the following sample run (with user input shown like this):

```
File: lear.txt
Lines = 6
Words = 47
Chars = 248
```

For the purposes of this program, a word consists of a consecutive sequence of letters and/or digits, which you can test using the static method `Character.isLetterOrDigit`.

6.1.17. Histogram

Write a console program named **Histogram** that reads a list of exam scores from an input file that contains one score per line and then displays a histogram of those numbers, divided into the ranges 0-9, 10-19, 20-29, and so forth, up to the range containing only the value 100. For example, suppose a file `midtermscores.txt` contains the numbers shown below (one per line):

```
73 58 73 93 82 62 80 53 93 52 92 75 65 95 23 100 75 38 80 77 92 60 98 95 62 87 97 73
78 72 55 58 42 31 78 70 78 74 70 60 72 75 84 87 62 17 92 78 74 65 90
```

Given this file, your program should be able to generate the following sample run (with user input shown like this):

```
File: midtermscores.txt
00-09:
10-19: *
20-29: *
30-39: **
40-49: *
50-59: *****
60-69: *******
70-79: *****************
80-89: ******
90-99: **********
  100: *
```

6.1.18. Add Stars

Write a method named **addStars** that accepts as a parameter an `ArrayList` of strings, and modifies the list by placing a `"*"` element between elements, as well as at the start and end of the list. For example, if a list named `list` contains `{"the", "quick", "brown", "fox"}`, the call of `addStars(list);` should modify it to store `{"*", "the", "*", "quick", "*", "brown", "*", "fox", "*"}`.

6.1.19. Count In Range

Write a method named **countInRange** that accepts three parameters: an `ArrayList` of integers, a minimum and maximum integer, and returns the number of elements in the list within that range inclusive. For example, if the list `v` contains `{28, 1, 17, 4, 41, 9, 59, 8, 31, 30,` 25}, the call of countInRange(v, 10, 30) should return 4. If the list is empty, return 0. Do not modify the list that is passed in.

6.1.20. Cumulative

Write a method named **cumulative** that accepts as a parameter an ArrayList of integers, and modifies it so that each element contains the cumulative sum of the elements up through that index. For example, if the list passed contains {1, 1, 2, 3, 5}, your method should modify it to store {1, 2, 4, 7, 12}.

6.1.21. Find Keith Numbers

Write a method named **findKeithNumbers** that accepts minimum and maximum integers as parameters and prints all of the "Keith numbers in that range (inclusive) in the format shown below. See previous program isKeithNumber to learn the definition of a "Keith number." If the given range contains no Keith numbers, print no output. For example, the call of findKeithNumbers(47, 742); would print the following output:

```
47: [4, 7, 11, 18, 29, 47]
61: [6, 1, 7, 8, 15, 23, 38, 61]
75: [7, 5, 12, 17, 29, 46, 75]
197: [1, 9, 7, 17, 33, 57, 107, 197]
742: [7, 4, 2, 13, 19, 34, 66, 119, 219, 404, 742]
```

6.1.22. Intersect

Write a method named **intersect** that accepts references to two sorted ArrayList of integers as parameters and returns a new list that contains only the elements that are found in both lists. For example, if list list1 and list2 store:

```
{1, 4, 8, 9, 11, 15, 17, 28, 41, 59}
{4, 7, 11, 17, 19, 20, 23, 28, 37, 59, 81}
```

Then the call of intersect(list1, list2) returns the list: {4, 11, 17, 28, 59} Note that you can assume that both lists passed store their elements in sorted order. Do not modify the two lists passed in as parameters.

6.1.23. Is Keith Number

Write a method named **isKeithNumber** that accepts an integer and returns true if that number is a "Keith number". A *"Keith number"* is defined as any *n*-digit integer that appears in the sequence that starts off with the number's *n* digits and then continues such that each subsequent number is the sum of the preceding *n*. (This is not unlike the classic Fibonacci sequence.) All one-digit numbers are trivially Keith numbers. The number 7385 is also a Keith number, because the following sequence ends up back at 7385:

```
7, 3, 8, 5, 23, 39, 75, 142, 279, 535, 1031, 1987, 3832, 7385
```

The sequence starts out 7, 3, 8, 5, because those are the digits making up 7385. Each number after that is the sum of the four numbers that precede it (four, because 7385 has four digits). So the fifth number is the sum of 7+3+8+5, or 23. The sixth number is 3+8+5+23, or 39. And so on, until we eventually get back to 7385, which makes 7385 a Keith number.

You may use a single `ArrayList` or `LinkedList` as auxiliary storage. Your method should not loop infinitely; if you become sure that the number is not a Keith number, stop searching and immediately return `false`.

6.1.24. Mean

Write a method named **`mean`** that accepts as a parameter an `ArrayList` of real numbers, and returns the arithmetic mean (average) of the integers in the list as a real number. For example, if the list passed contains `{2.0, 4.5, 6.5, 1.0}`, your method should return `3.5`. If the list is empty, return `0.0`. Do not modify the list that is passed in.

6.1.25. Remove All

Write a method named **`removeAll`** that accepts as a parameter an `ArrayList` of strings along with an element value string, and modifies the list to remove all occurrences of that string. For example, if the list `v` contains `{"a", "b", "c", "b", "b", "a", "b"}`, the call of `removeAll(v, "b");` should modify it to store `{"a", "c", "a"}`.

6.1.26. Remove Bad Pairs

Write a method named **`removeBadPairs`** that accepts as a parameter an `ArrayList` of integers, and removes any adjacent pair of integers in the list if the left element of the pair is larger than the right element of the pair. Every pair's left element is an even-numbered index in the list, and every pair's right element is an odd index in the list. For example, suppose a variable named `vec` stores the following element values:

```
{3, 7, 9, 2, 5, 5, 8, 5, 6, 3, 4, 7, 3, 1}
```

We can think of this list as a sequence of pairs:

```
{3, 7, 9, 2, 5, 5, 8, 5, 6, 3, 4, 7, 3, 1}
```

The pairs 9-2, 8-5, 6-3, and 3-1 are "bad" because the left element is larger than the right one, so these pairs should be removed. So the call of `removeBadPairs(vec);` would change the list to store:

```
{3, 7, 5, 5, 4, 7}
```

If the list has an odd length, the last element is not part of a pair and is also considered "bad;" it should therefore be removed by your method.

If an empty list is passed in, the list should still be empty at the end of the call.

Constraints: Do not use any other arrays, lists, or other data structures to help solve this problem, though you can create as many simple variables as you like.

6.1.27. Remove Consecutive Duplicates

Write a method named **removeConsecutiveDuplicates** that accepts as a parameter an `ArrayList` of integers, and modifies it by removing any consecutive duplicates. For example, if a list named `v` stores `{1, 2, 2, 3, 2, 2, 3}`, the call of `removeConsecutiveDuplicates(v);` should modify it to store `{1, 2, 3, 2, 3}`.

6.1.28. Stretch

Write a method named **stretch** that accepts as a parameter an `ArrayList` of integers, and modifies it to be twice as large, replacing every integer with a pair of integers, each half the original. If a number in the original list is odd, then the first number in the new pair should be one higher than the second so that the sum equals the original number. For example, if a variable named `v` refers to a list storing the values `{18, 7, 4, 24, 11}`, the call of `stretch(v);` should change `v` to contain `{9, 9, 4, 3, 2, 2, 12, 12, 6, 5}`. (The number 18 is stretched into the pair 9, 9, the number 7 is stretched into 4, 3, the number 4 is stretched into 2, 2, the number 24 is stretched into 12, 12, and 11 is stretched into 6, 5.)

6.2. Stack and Queue

6.2.1. Mystery #1

Write the output produced when the following method is passed each of the following stacks:

```
public void mystery1(Stack<Integer> s) {
    Queue<Integer> q = new LinkedList<Integer>();
    while (!s.isEmpty()) {
        q.add(s.peek());
        q.add(s.pop());
    }
    while (!q.isEmpty()) {
        s.push(q.peek());
        q.remove();
    }
    println(s);
}
```

[2, 3, 1]	
[42, -1, 4, 15, 9]	

[30, 20, 10, 70, 50, 40]	

6.2.2. Mystery #2

What is the output of the following code?

```
Stack<Integer> s = new Stack<Integer>();
s.push(7);
s.push(10);
print(s.peek() + " ");
print(s.pop() + " ");
s.push(3);
s.push(5);
print(s.pop() + " ");
print(s.size() + " ");
print(s.peek() + " ");
s.push(8);
print(s.pop() + " ");
print(s.pop() + " ");
```

Output:

A	7 10 5 3 3 8 3
B	7 7 10 3 3 3 5
C	10 10 5 4 5 8 8
D	10 10 5 2 3 8 3
E	other

6.2.3. Mystery #3

What is the output of the following code?

```
Queue<Integer> queue = new LinkedList<Integer>();
for (int i = 1; i <= 6; i++) {
    queue.add(i);
}                              // {1, 2, 3, 4, 5, 6}

for (int i = 0; i < queue.size(); i++) {
    print(queue.remove() + " ");
}
println(queue + "  size " + queue.size());
```

Output:

A	1 2 3 4 5 6 {} size 0
B	1 2 3 {4, 5, 6} size 3
C	1 2 3 4 5 6 {1, 2, 3, 4, 5, 6} size 6
D	other

6.2.4. Mystery #4

Write the output produced by the following method when passed each of the following queues:

```
public void collectionMystery3(Queue<Integer> q) {
    Stack<Integer> s = new Stack<Integer>();
    int size = q.size();
    for (int i = 0; i < size; i++) {
        int n = q.remove();
        if (n % 2 == 0) {
            s.push(n);
        } else {
            q.add(n);
        }
    }
    println("q=" + q);
    println("s=" + s);
}
```

{1, 2, 3, 4, 5, 6}	
{42, -3, 4, 15, 9, 71}	
{30, 20, 10, 60, 50, 40, 3, 0}	

6.2.5. Mystery #5

Write the output produced by the following method when passed each of the following stacks:

```
public void collectionMystery6(Stack<Integer> s) {
    Queue<Integer> q = new LinkedList<Integer>();
    Stack<Integer> s2 = new Stack<Integer>();

    while (!s.isEmpty()) {
        if (s.peek() % 2 == 0) {
            q.add(s.pop());
        } else {
            s2.push(s.pop());
        }
    }

    while (!q.isEmpty()) {
        s.push(q.remove());
    }
    while (!s2.isEmpty()) {
        s.push(s2.pop());
    }

    println(s);
}
```

{1, 2, 3, 4, 5, 6}	
{42, 3, 12, 15, 9, 71, 88}	

6.2.6. Mystery #6

Write the output produced by the following method when passed each of the following queues: Note that the queues below are written in *{front ... back}* order.

```
public void collectionMystery7(Queue<Integer> queue) {
    Stack<Integer> stack = new Stack<Integer>();
    int qsize = queue.size();
    for (int i = 0; i < qsize; i++) {
        if (queue.peek() % 2 == 0) {
            queue.add(queue.remove());
        } else {
            stack.push(queue.peek());
            stack.push(queue.remove());
        }
    }
    while (!queue.isEmpty()) {
        stack.push(queue.remove());
    }
    while (!stack.isEmpty()) {
        print(stack.pop() + " ");
    }
}
```

{1, 2, 3, 4, 5, 6}	
{55, 33, 0, 88, 44, 99, 77, 66}	
{80, 20, 65, 10, 5, 3, 40, 2, 11}	

6.2.7. Mystery #7

Write the output produced by the following method when passed each of the following stacks: Note that a stack prints in *{bottom ... top}* order.

```
public void collectionMystery8(Stack<Integer> stack) {
    Queue<Integer> queue = new LinkedList<Integer>();
    TreeSet<Integer> set = new TreeSet<Integer>();
    while (!stack.isEmpty()) {
        if (stack.peek() % 2 == 0) {
            queue.add(stack.pop());
        } else {
            set.add(stack.pop());
        }
    }
    for (int n : set) {
        stack.push(n);
    }
    while (!queue.isEmpty()) {
        stack.push(queue.remove());
    }
    println(stack);
}
```

{1, 2, 3, 4, 5}	
{3, 2, 7, 3, 3, 4, 1, 1, 4}	
{9, 7, 14, 7, 22, 7, 3, 14}	
{8, 5, 1, 2, 1, 1, 2, 1, 4, 5}	

6.2.8. Mystery #8

Write the output produced by the following method when passed each of the following queues and ints. Note: A stack displays/prints in {bottom ... top} order, and a queue displays in {front ... back} order.

```
public void collectionMystery9(Queue<Integer> queue, int p) {
    Stack<Integer> stack = new Stack<Integer>();
    int count = 0;
    int size = queue.size();
    for (int i = 0; i < size; i++) {
        int element = queue.remove();
        if (element < p) {
            queue.add(element);
        } else {
            count++;
            stack.push(count);
            stack.push(element);
        }
    }

    while (!stack.isEmpty()) {
        queue.add(stack.pop());
    }

    println(queue);
}
```

{1, 2, 3, 4, 5}, p=4	
{67, 29, 115, 84, 33, 71, 90}, p=50	

6.2.9. Mystery #9

Write the output produced by the following method when passed each of the following stacks and ints. Note: A stack displays/prints in {bottom ... top} order, and a queue displays in {front ... back} order.

```
public void collectionMystery10(Stack<Integer> stack, int n) {
    Stack<Integer> stack2 = new Stack<Integer>();
    Queue<Integer> queue = new LinkedList<Integer>();

    while (stack.size() > n) {
        queue.add(stack.pop());
    }
    while (!stack.isEmpty()) {
        int element = stack.pop();
        stack2.push(element);
        if (element % 2 == 0) {
            queue.add(element);
        }
    }
    while (!queue.isEmpty()) {
        stack.push(queue.remove());
    }
    while (!stack2.isEmpty()) {
        stack.push(stack2.pop());
    }

    println(stack);
}
```

{1, 2, 3, 4, 5, 6}, n=3	
{67, 29, 115, 84, 33, 71, 90}, n=5	

6.2.10. Check Balance

Write a method named **checkBalance** that accepts a string of source code and uses a `Stack` to check whether the braces/parentheses are balanced. Every `(` or `{` must be closed by a `}` or `)` in the opposite order. Return the index at which an imbalance occurs, or `-1` if the string is balanced. If any `(` or `{` are never closed, return the string's length.

Here are some example calls:

```
//    index   0123456789012345678901234567890
checkBalance("if (a(4) > 9) { foo(a(2)); }")      // returns -1 because balanced
checkBalance("for (i=0;i&lt;a(3};i++) { foo{); )")   // returns 14 because } out of
order
checkBalance("while (true) foo(); }{ ()")// returns 20 because } doesn't match any {
checkBalance("if (x) {")  // returns 8 because { is never closed
```

Constraints: Use a single stack as auxiliary storage.

6.2.11. Flip Half

Write a method named **flipHalf** that reverses the order of half of the elements of a `Queue` of integers passed as a parameter. Your method should reverse the order of all the elements in odd-numbered positions (position 1, 3, 5, etc.) assuming that the first value in the queue has position 0. For example, if the queue originally stores this sequence of numbers when the method is called:

```
index: 0  1  2  3  4  5   6   7
front {1, 8, 7, 2, 9, 18, 12, 0} back
```

Then it should store the following values after the method finishes executing:

```
index: 0  1  2  3   4  5  6   7
front {1, 0, 7, 18, 9, 2, 12, 8} back
```

Notice that numbers in even positions (positions 0, 2, 4, 6) have not moved. That sub-sequence of numbers is still: (`1, 7, 9, 12`). But notice that the numbers in odd positions (positions 1, 3, 5, 7) are now in reverse order relative to the original. In other words, the original sub-sequence: (8, 2, 18, 0) - has become: (0, 18, 2, 8).

Constraints: You may use a single stack as auxiliary storage.

6.2.12. Is Sorted

Write a method named **isSorted** accepts a stack of integers as a parameter and returns `true` if the elements in the stack occur in ascending (non-decreasing) order from top to bottom, else `false`. That is, the smallest element should be on top, growing larger toward the bottom. An empty or one-element stack is considered to be sorted. For example, if passed the following stack, your method should return `true`:

```
bottom {20, 20, 17, 11, 8, 8, 3, 2} top
```

The following stack is not sorted (the 15 is out of place), so passing it to your method should return a result of false:

```
bottom {18, 12, 15, 6, 1} top
```

When your method returns, the stack should be in the same state as when it was passed in. In other words, if your method modifies the stack, you must restore it before returning.

Constraints: You may use one queue or one stack (but not both) as auxiliary storage. Do not declare any other auxiliary data structures (e.g. arrays, Grids, ArrayLists, etc.), but you can have as many simple variables as you like. Your solution should run in O(N) time, where N is the number of elements of the stack.

6.2.13. Mirror

Write a method named **mirror** that accepts a queue of strings as a parameter and appends the queue's contents to itself in reverse order. For example, if a queue named `q` stores `{"a", "b", "c"}`, the call of `mirror(q);` should change it to store `{"a", "b", "c", "c", "b", "a"}`.

6.2.14. Reorder

Write a method named **reorder** that accepts as a parameter a queue of integers that are already sorted by absolute value, and modifies it so that the integers are sorted normally. For example, if a queue variable named `q` stores the following elements:

```
front {1, -2, 4, 5, -7, -9, -12, 28, -34} back
```

Then the call of `reorder(q);` should modify it to store the following values:

```
front {-34, -12, -9, -7, -2, 1, 4, 5, 28} back
```

Constraints: You may use a single stack as auxiliary storage.

6.2.15. Split Stack

Write a method named **splitStack** that accepts as a parameter a `Stack` of integers, and re-orders it so that all the non-negative numbers are at the top in the reverse of their original relative order, and all the negative numbers are at the bottom in the reverse of their original relative order. For example, if passed the stack {4, 0, -1, 5, -6, -3, 2, 7}, your method should modify the stack to store {-3, -6, -1, 7, 2, 5, 0, 4}.

Constraints: Do not declare any auxiliary data structures (e.g. arrays, Grids, ArrayLists, etc.) other than a single Queue of integers.

6.2.16. Stutter

Write a method named **stutter** that accepts a queue of integers as a parameter and replaces every element with two copies of itself. For example, if a queue named q stores {1, 2, 3}, the call of stutter(q); should change it to store {1, 1, 2, 2, 3, 3}.

Constraints: Do not use any auxiliary collections as storage.

6.3. Set

6.3.1. Remove Range

Write a method called **removeRange** that accepts three parameters: a Set of integers, a minimum value, and a maximum value. The method should remove any values from the set that are between that minimum and maximum value, inclusive. For example, if a Set named s contains [3, 17, -1, 4, 9, 2, 14], the call of removeRange(s, 1, 10); should modify s to store [17, -1, 14].

6.3.2. Num Unique Values

Write a method named **numUniqueValues** that accepts an ArrayList of integers as a parameter and returns the number of unique integer values in the list. For example, if a list named l contains the values [3, 7, 3, -1, 2, 3, 7, 2, 15, 15], the call of numUniqueValues(l) should return 5. If passed the empty list, you should return 0. Use a Set as auxiliary storage to help you solve this problem. Do not modify the list passed in.

6.3.3. Num In Common

Write a method named **numInCommon** that accepts two Lists of integers as parameters and returns the count of how many unique integers occur in both lists. For example, if two lists named l1 and l2 contains the values [3, 7, 3, -1, 2, 3, 7, 2, 15, 15] and [-5, 15, 2, -1, 7, 15, 36]respectively, your method should return 4 because the elements -1, 2, 7, and 15 occur in both lists. Use one or more Sets as storage to help you solve this problem. Do not modify the lists passed in.

6.3.4. Is Happy Number

Write a method named **isHappyNumber** that returns whether a given integer is "happy". An integer is "happy" if repeatedly summing the squares of its digits eventually leads to the number 1.

For example, 139 is happy because:

$$
\begin{aligned}
1^2 + 3^2 + 9^2 &= 91\\
9^2 + 1^2 &= 82\\
8^2 + 2^2 &= 68\\
6^2 + 8^2 &= 100\\
1^2 + 0^2 + 0^2 &= 1
\end{aligned}
$$

By contrast, 4 is not happy because:

$$
\begin{aligned}
4^2 &= 16\\
1^2 + 6^2 &= 37\\
3^2 + 7^2 &= 58\\
5^2 + 8^2 &= 89\\
8^2 + 9^2 &= 145\\
1^2 + 4^2 + 5^2 &= 42\\
4^2 + 2^2 &= 20\\
2^2 + 0^2 &= 4\\
&\ldots
\end{aligned}
$$

6.3.5. Max Length

Write a method **maxLength** that accepts as a parameter a `TreeSet` of strings, and that returns the length of the longest string in the set. If your method is passed an empty set, it should return 0.

6.3.6. Remove Duplicates

Write a method named **removeDuplicates** that accepts as a parameter an `ArrayList` of integers, and modifies it by removing any duplicates. Note that the elements of the list are not in any particular order, so the duplicates might not occur consecutively. You should retain the original relative order of the elements. Use a `TreeSet` as auxiliary storage to help you solve this problem. For example, if a list named `v` stores `{4, 0, 2, 9, 4, 7, 2, 0, 0, 9, 6, 6}`, the call of `removeDuplicates(v);` should modify it to store `{4, 0, 2, 9, 7, 6}`.

6.3.7. Set Mystery

What is the output of the following code?

```
TreeSet set = new TreeSet();
set.add(74);
set.add(12);
set.add(74);
set.add(74);
set.add(43);
set.remove(74);
set.remove(999);
set.remove(43);
set.add(32);
set.add(12);
set.add(9);
set.add(999);
println(set);
```

Output:

A	{9, 12, 32, 999}
B	{12, 74, 74, 32, 12, 9, 999}
C	{9, 12, 12, 32, 74, 74, 999}
D	{12, 32, 9, 999}

6.3.8. Twice

Write a method named **twice** that accepts as a parameter a list of integers and returns a set containing all the numbers in the list that appear exactly twice. For example, if a list variable `v` stores `{1, 3, 1, 4, 3, 7, -2, 0, 7, -2, -2, 1}`, the call of `twice(v)` should return the set `{3, 7}`.

If you want an extra challenge: Use only `TreeSet` as auxiliary storage.

6.3.9. Union Sets

Write a method named **unionSets** that accepts as a parameter a HashSet of TreeSets of integers, and returns a TreeSet of integers representing the union of all of the sets of ints. (A union is the combination of everything in each set.) For example, calling your method on the following set of sets:

```
{{1, 3}, {2, 3, 4, 5}, {3, 5, 6, 7}, {42}}
```

Should cause the following set of integers to be returned:

```
{1, 2, 3, 4, 5, 6, 7, 42}
```

6.3.10. Word Count

Write a method named **wordCount** that accepts a file name as a parameter and opens that file and that returns a count of the number of unique words in the file. Do not worry about capitalization and punctuation; for example, "Hello" and "hello" and "hello!!" are different words for this problem. Use a `TreeSet` as auxiliary storage.

6.4. Map

6.4.1. Mystery #1

Write the output that is printed when the given method below is passed each of the following maps as its parameter. Recall that maps print in a *key=value*format. Your answer should display the right keys and values in the order they are added to the result map.

```
public void collectionMystery1(HashMap<String, String> map) {
    HashMap<String, String> result = new HashMap<String, String>();
    for (String k : map.keySet()) {
        String v = map.get(k);
        if (k.charAt(0) <= v.charAt(0)) {
            result.put(k, v);
        } else {
            result.put(v, k);
        }
    }
    println(result);
}
```

{two=deux, five=cinq, one=un, three=trois, four=quatre}	
{skate=board, drive=car, program=computer, play=computer}	
{siskel=ebert, girl=boy, heads=tails, ready=begin, first=last, begin=end}	
{cotton=shirt, tree=violin, seed=tree, light=tree, rain=cotton}	

6.4.2. Mystery #2

Write the output that is printed when the given method below is passed each of the following pairs of maps as its parameter. Recall that maps print in a *{key1=value1, key2=value2, ..., keyN=valueN}* format. Your answer should display the right keys and values in the order they are added to the resulting map.

```
public HashMap<String, String> collectionMystery2(HashMap<String, Integer>
map1, HashMap<Integer, String> map2) {
    HashMap<String, String> result = new HashMap<String, String>();
    for (String s1 : map1.keySet()) {
        if (map2.containsKey(map1.get(s1))) {
            result.put(s1, map2.get(map1.get(s1)));
        }
    }
    return result;
}
```

map1={bar=1, baz=2, foo=3, mumble=4}, map2={1=earth, 2=wind, 3=air, 4=fire}	
map1={five=105, four=104, one=101, six=106, three=103, two=102}, map2={99=uno, 101=dos, 103=tres, 105=cuatro}	
map1={a=42, b=9, c=7, d=15, e=11, f=24, g=7}, map2={1=four, 3=score, 5=and, 7=seven, 9=years, 11=ago}	

6.4.3. Mystery #3

Write the output that is printed when the given method below is passed each of the following pairs of maps as its parameter. Recall that maps print in a *{key1=value1, key2=value2, ..., keyN=valueN}* format. You should assume that when looping over the map, it loops over the keys in the order that they are declared below.

```
public void collectionMystery3(HashMap<String, String> map) {
    ArrayList<String> list = new ArrayList<String>();
    for (String key : map.keySet()) {
        if (map.get(key).length() > key.length()) {
            list.add(map.get(key));
        } else {
            list.add(0, key);
            list.remove(map.get(key));
        }
    }
    println(list);
}
```

{horse=cow, cow=horse, dog=cat, ok=yo}	
{bye=hello, bird=dog, hi=hello, hyena=apple, fruit=meat}	
{a=b, c=d, e=a, ff=a, gg=c, hhh=ff}	

6.4.4. Mystery #4

Write the output that is printed when the given method below is passed each of the following maps and lists as its parameters. Recall that maps print in a *{key1=value1, key2=value2, ..., keyN=valueN}* format.

Though hash maps usually have unpredictable ordering, for this problem, you should assume that when looping over the map or printing a map, it visits the keys in the order that they were added to the map or the order they are declared below. If a map calls `put()` on a key that already exists, it replaces that key and keeps it in its current position in the ordering.

```
public void collectionMystery4(HashMap<String, String> map, ArrayList<String>
list) {
    HashMap<String, String> result = new HashMap<String, String>();
    for (int i = 0; i < list.size(); i++) {
        String s = list.get(i);
        if (result.containsKey(s)) {
            result.put(s, result.get(s) + result.get(s));
        } else if (map.containsKey(s)) {
            result.put(map.get(s), s);
        }
    }
    println(result);
}
```

map = {Marty=Stepp, Cynthia=Lee, Keith=Schwarz, Bruce=Lee, Mehran=Sahami};
list = [Cynthia, Bruce, Lee, Eric, Schwarz, Keith, Sahami]	

map = {dog=woof, cat=meow, horse=whinny, frog=ribbit, duck=dog}; list = [dog, horse, dog, woof, meow, cat, meow, woof]	

6.4.5. Mystery #5

Write the output that is printed when the given method below is passed each of the following maps and lists as its parameters. Recall that maps print in a *{key1=value1, key2=value2, ..., keyN=valueN}* format.

Though hash maps usually have unpredictable ordering, for this problem, you should assume that when looping over the map or printing a map, it visits the keys in the order that they were added to the map or the order they are declared below. If a map calls put() on a key that already exists, it retains its current position in the ordering.

```
public void collectionMystery5(ArrayList<String> list1, ArrayList<String>
list2) {
    HashMap<String, String> result = new HashMap<String, String>();

    for (int i = 0; i < list1.size(); i++) {
        String s1 = list1.get(i);
        String s2 = list2.get(i);

        if (!result.containsKey(s1)) {
            result.put(s1, s2);
        } else if (!result.containsKey(s2)) {
            result.put(s2, s1);
        } else {
            result.put(s1 + s2, s1);
        }
    }
    println(result);
}
```

list1 = ["cat", "cat", "long", "long", "longcat"]
list2 = ["mew", "purr", "cat", "cat", "purr"]	

list1 = ["a", "b", "a", "ab", "ab", "y", "abb"]
list2 = ["b", "c", "b", "b", "c", "abb", "y"]	

6.4.6. Has Duplicate Value

Write a method named **hasDuplicateValue** that accepts a Map from strings to strings as a parameter and returns true if any two keys map to the same value. For example, if a map named m stores {Marty=Stepp, Stuart=Reges, Jessica=Miller, Amanda=Camp, Meghan=Miller, Hal=Perkins}, the call of hasDuplicateValue(m) would return true because both "Jessica" and "Meghan" map to the value "Miller". Return false if passed an empty or one-element map. Do not modify the map passed in.

6.4.7. Least Common

Write a method **leastCommon** that accepts a HashMap whose keys are strings and whose values are integers as a parameter and returns the integer value that occurs the fewest times in the map. For example, if a map named m contains {Alyssa=22, Char=25, Dan=25, Jeff=20, Kasey=20, Kim=20, Mogran=25, Ryan=25, Stef=22}, the call of leastCommon(m) returns 22. If there is a tie, return the smaller integer value. If the map is empty, throw an IllegalArgumentException.

6.4.8. Count Names

Write a console program named **CountNames** that asks the user for a list of names (one per line) until the user enters a blank line (i.e., just presses Enter when asked for a name). At that point the program should print out how many times each name in the list was entered. A sample run of this program is shown below.

```
Enter name: Alice
Enter name: Bob
Enter name: Alice
Enter name: Chelsea
Enter name: Bob
Enter name: Alice
Enter name:
Entry [Alice] has count 3
Entry [Bob] has count 2
Entry [Chelsea] has count 1
```

6.4.9. Has Three

Write a method named **hasThree** that accepts a list of strings as a parameter and returns `true` if any string value occurs at least 3 times in the list. For example, in the list `["to", "be", "or", "be", "to", "be", "hamlet"]`, the word `"be"` occurs 3 times, so your method would return `true` if passed that list. Use a `HashMap` as auxiliary storage to help solve the problem. Do not modify the list that is passed in.

6.4.10. Intersect

Write a method named **intersect** that accepts two `HashMaps` of strings to integers as parameters and that returns a new map whose contents are the intersection of the two. The intersection of two maps is defined here as the set of keys and values that exist in both maps. So if some value K maps to value V in both the first and second map, include it in your result. If K does not exist as a key in both maps, or if K does not map to the same value V in both maps, exclude that pair from your result. For example, consider the following two maps:

```
{Janet=87, Logan=62, Whitaker=46, Alyssa=100, Stefanie=80, Jeff=88, Kim=52,
Sylvia=95}
{Logan=62, Kim=52, Whitaker=52, Jeff=88, Stefanie=80, Brian=60, Lisa=83, Sylvia=87}
```

Calling your method on the preceding maps would return the following new map:

```
{Logan=62, Stefanie=80, Jeff=88, Kim=52}
Do not modify the maps passed in as parameters.
```

6.4.11. Max Occurrences

Write a method named **maxOccurrences** that accepts a list of integers as a parameter and returns the number of times the most frequently occurring integer (the "mode") occurs in the list. Solve this problem using a map as auxiliary storage. If the list is empty, return `0`. Do not modify the list passed in as a parameter.

6.4.12. Reverse

Write a method named **reverse** that accepts a map from integers to strings as a parameter and returns a new map of strings to integers that is the original's "reverse". The reverse of a map is defined here to be a new map that uses the values from the original as its keys and the keys from the original as its values. Since a map's values need not be unique but its keys must be, it is acceptable to have any of the original keys as the value in the result. In other words, if the original map has pairs (k1, v) and (k2, v), the new map must contain either the pair (v, k1) or (v, k2).

For example, for the following map:

```
{42=Marty, 81=Sue, 17=Ed, 31=Dave, 56=Ed, 3=Marty, 29=Ed}
```

Your method could return the following new map (the order of the key/value pairs does not matter):

```
{Marty=3, Sue=81, Ed=29, Dave=31}
```

Do not modify the map passed in as the parameter.

6.4.13. Is Sub Map

Write a method named **isSubMap** that accepts two hash maps from strings to strings as its parameters and returns `true` if every key in the first map is also contained in the second map and maps to the same value in the second map.

For example, given the maps below, `map1` is a sub-map of `map2`, so the call of `isSubMap(map1, map2)` would return `true`. The order of the parameters does matter, so the call of `isSubMap(map2, map1)` would return `false`.

But `map3` is not a sub-map of `map2` because the key `"Alisha"` is not in `map2` and also because the key `"Smith"` does not map to the same value as it does in `map2`; therefore the call of `isSubMap(map3, map2)` would return `false`. The empty map is considered to be a sub-map of every map, so the call of `isSubMap(map4, map1)` would return `true`.

```
map1: {Smith=949-0504, Marty=206-9024}
map2: {Marty=206-9024, Hawking=123-4567, Smith=949-0504, Newton=123-4567}
map3: {Alisha=321-7654, Hawking=123-4567, Smith=888-8888}
map4: {}
```

6.4.14. Get Majority Last Name

Write a method named **getMajorityLastName** that accepts as its parameter a hash map from strings to strings; the keys of the map represent first names and the values represent last names. If there is a single common last name that is present in more than half of the key/value pairs in the map passed in (a "majority" last name), your method should return that last name. If there is no majority last name, your method should return the string `"?"` .

For example, if the map contains the following key/value pairs, the majority last name is "Smith" because it occurs 5 times, which is more than half of the nine pairs in the map. Therefore your method would return "Smith".

```
{Hal=Perkins, Mark=Smith, Mike=Smith, Stuart=Reges, David=Smith, Jean=Reges,
Geneva=Smith, Amie=Smith, Bruce=Reges}
```

The following maps don't have any majority last name because no last name occurs strictly greater than half the time. Therefore when passed either of the maps below, your method would return `"?"` .

```
{Marty=Stepp, Mehran=Sahami, Keith=Schwarz, Cynthia=Lee, Yogurt=Schwarz}
{Tywin=Lannister, Rob=Stark, Sansa=Stark, Tyrion=Lannister}
```

If the map that contains only one key/value pair, that pair's value is the majority last name. An empty map does not have any majority last name.

Constraints: You may declare at most one auxiliary data structure to help you solve this problem. Do not modify the map that is passed in to your method as a parameter.

6.4.15. Rarest Age

Write a method named **rarestAge** that accepts as a parameter a `HashMap` from students' names (strings) to their ages (integers), and returns the least frequently occurring age. Consider a map variable named `m` containing the following key/value pairs:

```
{Char=45, Dan=45, Jerry=23, Kasey=10, Jeff=10, Elmer=45, Kim=10, Ryan=45, Mehran=23}
```

Three people are age 10 (Kasey, Jeff, and Kim), two people are age 23 (Jerry and Mehran), and four people are age 45 (Char, Dan, Elmer, and Ryan). So a call of `rarestAge(m)` returns `23` because only two people are that age.

If there is a tie (two or more rarest ages that occur the same number of times), return the youngest age among them. For example, if we added another pair of Steve=23 to the map above, there would now be a tie of three people of age 10 (Kasey, Jeff, Kim) and three people of age 23 (Jerry, Mehran, Steve). So a call of `rarestAge(m)` would now return `10` because 10 is the smaller of the rarest values. This implies that if every person in the map has a unique age, your method would return the smallest of all the ages in the map.

If the map passed to your method is `null` or empty, your method should return `0`. You may assume that no key or value stored in the map is `null`. Otherwise you should not make any assumptions about the number of key/value pairs in the map or the range of possible ages that could be in the map.

Constraints: You may create one collection of your choice as auxiliary storage to solve this problem. You can have as many simple variables as you like. You should not modify the contents of the map passed to your method.

6.4.16. Teacher

Write a method named **teacher** that produces a mapping of students' grades. Your method accepts two parameters: a class roster telling you each student's name and percentage earned in the course, and a grade mapping telling you the minimum percentage needed to earn each unique grade mark. The class roster is a hash map from students' names (strings) to the percentage of points they earned in the course (integers). The grade mapping is a hash map from percentages (integers) to grades (strings) and indicates the minimum percentage needed to earn each kind of grade. The task of your method is to look at the student roster and grade mapping and use them to build and return a new `HashMap` from students' names to the letter grades they have earned in the class, based on their percentage and the grade mapping. Each student should be given the grade that corresponds to the highest value from the grade mapping that is less than or equal to the percentage that the student earned.

Suppose that the class roster is stored in a map variable named `roster` that contains the following key/value pairs, and that the grade mapping is stored in a map variable named `gradeMap` that contains the key/value pairs below it:

```
roster: {Mort=77, Dan=81, Alyssa=98, Kim=52, Lisa=87, Bob=43, Jeff=70, Sylvia=92,
Vikram=90}
gradeMap: {52=D, 70=C-, 73=C, 76=C+, 80=B-, 84=B, 87=B+, 89=A-, 91=A, 98=A+}
```

The idea is that Mort earned a C+ because his grade is at least 76%; Dan earned a B- because he earned at least 80%; and so on. If a given student's percentage is not as large as any of the percentages in the map, give them a default grade of "F". So your method should build and return the following map from students' names to their letter grades when passed the above data:

```
return value:
  {Mort=C+, Dan=B-, Alyssa=A+, Kim=D, Lisa=B+, Bob=F, Jeff=C-, Sylvia=A, Vikram=A-}
```

Though maps often store their elements in unpredictable order, for this problem you may assume that the grade mapping's key/value pairs are stored in ascending order by keys (percentages).

If either map passed to your method is empty, your method should return an empty map. You should not make any assumptions about the number of key/value pairs in the map or the range of possible percentages that could be in the map.

Constraints: You may create one new map as storage to solve this problem. (This is the map you will return.) You may not declare any other data structures. You can have as many simple variables as you like, such as integers or strings. Do not modify the maps that are passed in to your method as a parameter.

6.4.17. Postal Service

Write a method named **postalService** that helps the postal service sort mail for customers based on "ZIP" codes. A ZIP code is an integer that represents a given city or region of the USA. Your method accepts two hash maps as parameters:

1. a {customer name → ZIP code} map, where each key is a person's full name as a string and each value is that person's numeric ZIP code as an integer. For example, the pair Jon Smith=12345 would mean that the customer named John Smith lives at the ZIP code of 12345.
2. a {ZIP code → city} map, where each key is an integer ZIP code and each value is the name of the city for that zip code as a string. For example, the pair 12345=Mordor would mean that the ZIP code 12345 corresponds to the city of Mordor.

Your job is to build and return a new hash map where the keys are city names, and values are strings showing the last name (surname) of every customer whose ZIP code is in that city.

In the customer map passed to your method, names are represented in "FirstName LastName" format, where the given name (first name) is followed by a space and then the surname (last name), such as "Michael Jackson". But a name could have more words in it, such as "Oscar de la Hoya", or could contain just a single word, such as "Cher". For any name, regardless of how many words it contains, the last word in that name is considered to be the customer's last name / surname.

For example, two maps named `people` and `cities` might store the following pairs, shown in *key=value* format:

```
(1) 'people' customer map:
  {Ally T Obern=85704, Madonna=11430, David Q Shaw=90045, Mike Tom Brooks=85704,
   Jerry Cain=11430, Kate Jan Martin=68052, Jane Su=68052, Jessica K. R.
Miller=94305, Marty Doug Stepp=95050, Nick T=94305, Sara de la Pizza=68052, Stu T.
Reges=94305, Prince=94305, Dany Khaleesi Mother of Dragons Targaryen=9999999}
(2) 'cities' ZIP/city map:
  {11430=NewYork, 22222=Duluth, 68052=Springfield, 71384=Omaha, 85704=Tucson,
   90045=Redmond, 94305=Stanford, 95050=SantaClara, 9999999=Westeros}
```

In the result map you must build, each key is a string representing a city name, and the associated value is a string with the last names of all people in that ZIP code, separated by `" and "` if there is more than one person in the ZIP code in the data. The order names are listed for a given ZIP code does not matter. If a given city does not correspond to any people in the customer map, that city name should not be included in your result map.

When passed the maps above, the call of `postalService(people, cities)` should return the following map:

```
  {Tucson=Obern and Brooks, NewYork=Madonna and Cain, Redmond=Shaw,
   Springfield=Martin and Su and Pizza, Stanford=Miller and T and Reges and Prince,
   SantaClara=Stepp, Westeros=Targaryen}
```

Assumptions: You may assume that ZIP codes will be non-negative, but otherwise you should not make any assumptions about the range of numbers you will see as ZIP codes. You may assume that names are non-empty strings consisting of at least one character, but otherwise you should also not make any assumptions about the length or content of names. You may assume that every ZIP code found in the customer map will also be found in the ZIP/city map, but not every ZIP/city found in the cities map might be present in the people/customers map. If the customer map passed is empty, return an empty map. Otherwise do not make assumptions about the size of either map.

Constraints: You may declare one map to help you solve this problem. (This is the map you will return.) You may not declare any other data structures. You can have as many simple variables as you like, such as integers or strings. Do not modify the maps that are passed in to your method as parameters.

6.4.18. Biggest Family

Write a method named **biggestFamily** that reads an input file of people's names and prints information about which family has the most people in it. Your method accepts a string parameter representing a filename of input.

The input file contains a collection of names, one per line, in the format of the example shown at right. Each line of the file contains a first name (given name), a single space, and a last name (surname / family name). For example, in the name "Ned Stark", the word "Ned" is the first name and "Stark" is the last name. You may assume that every line follows this exact format and that first and last names are single words.

```
Jon Snow
Ned Stark
Gregor Clegane
Cersei Lannister
Tyrion Lannister
Sandor Clegane
Jaime Lannister
Catelyn Stark
Theon Greyjoy
Arya Stark
Cersei Smith
Ned Jones
```

Your method should open and read the contents of this input file and figure out which last name(s) occur most frequently in the data, and print the members of that family in ABC order in exactly the format shown below.

If multiple families are tied for the most members, print each of the tied families in the same format. For example, in the data at right, the largest families are Stark and Lannister, each of which has 3 members listed, so your method should print the Lannisters in ABC order and then the Starks in ABC order.

For example, if the input above is in `families.txt`, then the call of `biggestFamily("families.txt");` should print:

```
Lannister family: Cersei Jaime Tyrion
Stark family: Arya Catelyn Ned
```

It is possible that more than two families might tie for the most members. One example of such a case is if every person in the file has a different last name. In such a case, you should print all of the tying families in this same format.

Assumptions: You may assume that the file exists, and that it contains at least one name, that every line of input in the file is in the exact valid format described above, and that no two lines of the file will be exactly the same (though a given first or last name might occur multiple times).

Constraints:

- You may open and read the file only once. Do not re-open it or rewind the stream.
- You should choose an efficient solution. Choose data structures intelligently and use them properly.
- You may create up to two collections (stack, queue, set, map, etc.) or nested/compound structure as auxiliary storage. A nested structure, such as a set of lists, counts as one collection. A temporary collection variable that is merely a replica or reference to some other collection (such as, Stack v = myQueue.remove();) is fine and does not count as a second structure. (You can have as many simple variables as you like, such as ints or strings.)

6.4.19. By Age

Write a method named **byAge** that accepts three parameters: 1) a map where each key is a person's name (a string) and the associated value is that person's age (an integer); 2) an integer for a minimum age; and 3) an integer for a max age. Your method should return a new map with information about people with ages between the min and max, inclusive.

In your result map, each key is an integer age, and the value for that key is a string with the names of all people at that age, separated by "and" if there is more than one person of that age. The order of names for a given age should be in alphabetical order, such as "Bob and Carl" rather than "Carl and Bob". (This is the order in which they naturally occur in the parameter map.) Include only ages between the min and max inclusive, where there is at least one person of that age in the parameter map. If the map passed is empty, or if there are no people in the map between the min/max ages, return an empty map.

For example, if a map named `ages` stores the following key:value pairs:

```
{Allison=18, Benson=48, David=20, Erik=20, Galen=15, Grace=25,
 Helene=40, Janette=18, Jessica=35, Marty=35, Paul=28, Sara=15,
 Stuart=98, Tyler=6, Zack=20}
```

The call of `byAge(ages, 16, 25)` should return the following map:

```
{18=Allison and Janette, 20=David and Erik and Zack, 25=Grace}
```

For the same map, the call of `byAge(ages, 20, 40)` should return the following map:

```
{20=David and Erik and Zack, 25=Grace, 28=Paul, 35=Jessica and Marty, 40=Helene}
```

Constraints: Obey the following restrictions in your solution.

- You will need to construct a map to store your results, but you may not use any other structures (arrays, lists, etc.) as auxiliary storage. (You can have as many simple variables as you like.)
- You should not modify the contents of the map passed to your method. Declare your method in such a way that any caller can be sure that this will not happen.
- Your solution should run in no worse than O(N log N) time, where N is the number of pairs in the map.

6.4.20. Friend List

Write a method named **friendList** that accepts a file name as a parameter and reads friend relationships from a file and stores them into a compound collection that is returned. You should create a map where each key is a person's name from the file, and the value associated with that key is a setof all friends of that person. Friendships are bi-directional: if Marty is friends with Danielle, Danielle is friends with Marty.

The file contains one friend relationship per line, consisting of two names. The names are separated by a single space. You may assume that the file exists and is in a valid proper format. If a file named `buddies.txt` looks like this:

```
Marty Cynthia
Danielle Marty
```

Then the call of `friendList("buddies.txt")` should return a map with the following contents:

```
{Cynthia=[Marty], Danielle=[Marty], Marty:[Cynthia, Danielle]}
```

Constraints:

- You may open and read the file only once. Do not re-open it or rewind the stream.
- You should choose an efficient solution. Choose data structures intelligently and use them properly.
- You may create one collection (stack, queue, set, map, etc.) or nested/compound structure as auxiliary storage. A nested structure, such as a set of lists, counts as one collection. (You can have as many simple variables as you like, such as ints or strings.)

6.4.21. Last Names By Age

Write a method named **lastNamesByAge** that accepts three parameters: 1) a `TreeMap` where each key is a person's full name (a string) and the associated value is that person's age (an integer); 2) an integer for a minimum age; and 3) an integer for a max age. Your method should return a new map with information about people with ages between the min and max, inclusive.

In your result map, each key is an integer age, and the value for that key is a string with the last names of all people at that age, separated by "and" if there is more than one person of that age. The order of names for a given age should be in the order they occurred in the parameter map. Include only ages between the min and max inclusive, where there is at least one person of that age in the parameter map. If the map passed is empty, or if there are no people in the map between the min/max ages, return an empty map.

Some names are in the format "first last" such as "Marty Stepp". But a name could have more tokens, such as "Oscar de la Hoya", or could contain just a single token, such as "Cher". For example, if a map named `ages` stores the following key/value pairs:

```
{Allison L. Smith=18, Benson Kai Lim=48, David L Shaw=20, Erik Thomas Jones=20,
 Galen Wood=15, Madonna=25, Helene Q. Martin=40, Janette Siu=18,
 Jessica K. Miller=35, Marty Douglas Stepp=35, Paul Beame=28, Sara de la Pizza=15,
 Stuart T. Reges=98, Tyler Rigs=6, Prince=20}
```

The call of `lastNamesByAge(ages, 16, 25)` should return the following map:

```
{18=Smith and Siu, 20=Shaw and Jones and Prince, 25=Madonna}
```

For the same map, the call of `lastNamesByAge(ages, 20, 40)` should return the following map:

```
{20=Shaw and Jones and Prince, 25=Young, 28=Beame, 35=Miller and Stepp, 40=Martin}
```

Constraints: Obey the following restrictions in your solution.

- You will need to construct a map to store your results, but you may not use any other structures (arrays, lists, etc.) as auxiliary storage. (You can have as many simple variables as you like.)
- You should not modify the contents of the map passed to your method. Declare your method in such a way that any caller can be sure that this will not happen.
- Your solution should run in no worse than O(N log N) time, where N is the number of pairs in the map.

6.4.22. Mystery #6

What is the output of the following code?

```
TreeMap map = new TreeMap();
map.put("K", "Schwarz");
map.put("C", "Lee");
map.put("M", "Sahami");
map.put("M", "Stepp");
map.remove("Stepp");
map.remove("K");
map.put("J", "Cain");
map.remove("C, Lee");
println(map);
```

Output:

A	{C=Lee, J=Cain, M=Stepp}
B	{C=Lee, J=Cain, M=Stepp, M=Sahami}
C	{J=Cain, C=Lee, M=Sahami, M=Stepp}

D	{J=Cain, M=Sahami}

6.4.23. Pair Frequencies

Write a method named **pairFrequencies** that prints particular data about two-letter pairs in a collection of words. In the English language, some combinations of adjacent letters are more common than others. For example, 'h' often follows 't' ("th"), but rarely would you see 'x' following 't' ("tx"). Knowing how often a given letter follows other letters in the English language is useful in many contexts. In cryptography, we use this data to crack substitution ciphers (codes where each letter has been replaced by a different letter, for example: A-M, B-T, etc.) by identifying which possible decoding substitutions produce plausible letter combinations and which produce nonsense.

For this problem, you will write a method named `pairFrequencies` that accepts a `Set` representing a dictionary of words. Your method will examine the dictionary and print all 2-character sequences of letters along with a count of how many times each pairing occurs. For example, suppose a `Set` variable named `dict` contains the following words:

```
{"banana", "bends", "i", "mend", "sandy"}
```

This dictionary contains the following two-character pairs: "ba", "an", "na", "an", "na" from banana; "be", "en", "nd", "ds" from bends; "me", "en", "nd" from mend; and "sa", "an", "nd", "dy" from sandy. (Note that "i" is only one character long, so it contains no pairs.) So the call of `pairFrequencies(dict);` would print the following output:

```
an: 3
ba: 1
be: 1
ds: 1
dy: 1
en: 2
me: 1
na: 2
nd: 3
sa: 1
```

Notice that pairings that occur more than once in the same word should be counted as separate occurrences. For example, "an" and "na" each occur twice in "banana".

Constraints: Obey the following restrictions in your solution.

- You may create one additional data structure (stack, queue, set, map, etc.) as auxiliary storage. A nested structure, such as a set of lists, counts as one additional data structure. (You can have as many simple variables as you like.)
- You should not modify the contents of the set passed to your method. Declare your method in such a way that any caller can be sure that this will not happen.
- You should loop over the contents of the set no more than once.
- Your solution should run in no worse than O(N^2) time, where N is the number of pairs in the map.

6.4.24. Rarest

Write a method named **rarest** that accepts a `TreeMap` from strings to strings as a parameter and returns the value that occurs least frequently in the map. If there is a tie, return the value that comes earlier in ABC order. For example, if a variable called `map` containing the following elements:

```
{"Alyssa":"Harding", "Char":"Smith", "Dan":"Smith", "Jeff":"Jones", "Kasey":"Jones",
 "Kim":"Smith", "Morgan":"Jones", "Ryan":"Smith", "Stef":"Harding"}
```

Then a call of `rarest(map)` would return `"Harding"` because that value occurs 2 times, fewer than any other. Note that we are examining the values in the map, not the keys. If the map passed is empty, throw an `IllegalArgumentException`.

Constraints: Obey the following restrictions in your solution.

- You may create one additional data structure (stack, queue, set, map, etc.) as auxiliary storage. (You can have as many simple variables as you like.)
- You should not modify the contents of the map passed to your method. Declare your method in such a way that any caller can be sure that this will not happen.
- Your solution should run in no worse than O(N log N) time, where N is the number of pairs in the map.

6.4.25. Starters

Write a method named **starters** that accepts two parameters: a list of strings, and an integer *k*. Your method should examine the strings in the list passed and return a set of all first characters that occur at least *k* times. In other words, if *k* or more strings in the list start with a particular character at index 0 of the string (case-insensitively), that character should be part of the set that you return. All elements of your set should be in lowercase. For example, consider a list variable called `v` containing the following elements:

```
{"hi", "how", "are", "He", "", "Marty!", "this", "morning?", "fine.", "?foo!", "",
"HOW", "A"}
```

Two words in the list start with "a", one starts with "f", four start with "h", two start with "m", one starts with "t", and one starts with "?". Therefore the call of `starters(v, 2)` should return a set containing:

```
{'a', 'h', 'm'}
```

The call of `starters(v, 3)` on the same list should return a set containing:

```
{'h'}
```

If no start character occurs *k* or more times, return an empty set. The characters should appear in your set in alphabetical order. Note that some of the elements of the list might be empty strings; empty strings have no first character, so your code should not consider them when counting. (But your code shouldn't crash on an empty string.)

Constraints: Obey the following restrictions in your solution.

- You will need to construct your set to be returned, and in addition to that, you may create one additional data structure (stack, queue, set, map, etc.) as auxiliary storage. (You can have as many simple variables as you like.)
- You should not modify the contents of the list passed to your method. Declare your method in such a way that any caller can be sure that this will not happen.
- Your solution should run in no worse than O(N log N) time, where N is the number of pairs in the map.

6.4.26. To Morse Code

Write a method named **toMorseCode** that converts strings into their Morse code equivalents. Morse code is a mapping from each character from A-Z to a sequence of dots and dashes. For example, the string `"SOS"` could be represented in Morse code as `"... --- ..."`.

Your method accepts two parameters: a `TreeMap` from `char` to `String`, and a `String` of text to convert. Assume that the provided client code builds a map from individual text characters to their Morse code equivalents. For example, the key `'A'` maps to `".-"` . Your method accepts such a map, and a string to be converted, and should print out the Morse code equivalent of the given string to the console.

For example, if the letter to Morse code map is stored in a variable called `mapping`, the call of `toMorseCode(mapping, "SOS TITANIC");` should print the following console output:

```
... --- ... - .. - .- -. .. -.-.
```

Note that the string might contain some characters that are not uppercase A-Z letters (like spaces); just skip those characters. You may assume that the mapping passed contains a mapping for every letter from A-Z in uppercase. Do not modify the letter map that is passed in.

6.4.27. Word Chain

Write a method named **wordChain** that accepts two string parameters, the first representing an input filename and the second representing a starting word, and that produces a random "word chain" starting from the given word.

For this problem let's define a "word chain" as a sequence of words where the last two letters of the current word will be the first two letters of the next word. For example, here is a possible word chain that starts from the word "program":

```
program → amender → erected → edaciousness
```

The words you use in your word chain come from the given input file. You should assume that this file contains a sequence of words, one per line. Your method should open and read the contents of this input file and use those words when creating a word chain. If the file exists and is readable, you may assume that its contents consist entirely of words listed one-per-line in lowercase, and that each word in the dictionary is at least 2 letters long. For example, the file `dictionary.txt` might contain words in the format shown below (abbreviated by ...).

```
aah
aahed
...
zygoid
zygote
```

You are producing a random word chain, so the idea is that you should randomly choose a next word whose first two letters are the same as the last two letters of the current word. In our sample chain above, any word starting with "am" would be a valid choice for the second word; and if the second word chosen is "amender", then any word starting with "er" would be a valid choice for the third word. And so on. A word chain might have a duplicate in it; this is okay.

A word chain ends when you reach a two-letter word suffix that is not the start of any word in the dictionary. For example, in the chain shown above, we produced "edaciousness". There are no words in the dictionary that begin with "ss", so the chain ends and the method stops.

Your method should print the word chain to the console, one word per line. Here are several example outputs from the call of `wordChain("dictionary.txt", "computer");` . The implication of the outputs below is that the given dictionary does not contain any words that begin with "gs", or "ss", or "ns", which is why the chains end there.

```
computer        computer          computer
erecting        ere               erotize
ngatis          reservednesses    zecchins
isocyanates     espouse
esthete         serovar
terminism       arpeggio
smug            iodines
uglying         eschalot
ngati           ototoxicity
tidings         tyeing
                nganas
                assentive
                vestibule
                lecherousness
```

Notice that the same word suffix/prefix could occur more than once in the same chain, such as "ng" in "erecting" / "ngatis" and again later in "uglying" / "ngati". If the start word passed in ends with a two-letter sequence that is not the start of any words in the input file, your method should simply print the start word and then exit.

If the file is missing/unreadable, or the start word's length is less than 2, your method should throw an `IllegalArgumentException`.

Constraints: Your solution should read the file only once, not make multiple passes over the file data. Similarly, don't store the entire file contents in a collection and loop over that entire collection multiple times. You should choose an efficient solution. Choose data structures intelligently and use them properly. You may create up to two additional data structures (stack, queue, set, map, etc.) as auxiliary storage. A nested structure, such as a set of lists, counts as one additional data structure. (You can have as many simple variables as you like, such as ints or strings.)

6.4.28. Flight Planner

Write a console program named **FlightPlanner** that reads in a file containing flight destinations from various cities, and then allow the user to plan a round-trip flight route. Here's what a sample run of the program might look like:

```
...
```

The flight data come from a file named flights.txt , which has the following format:

- Each line consists of a pair of cities separated by an arrow indicated by the two character combination ->, as in:
  New York -> Anchorage
- The file may contain blank lines for readability (you should just ignore these).

The entire data file used to produce this sample run appears below.

```
San Jose -> San Francisco
San Jose -> Anchorage
New York -> Anchorage
New York -> San Jose
New York -> San Francisco
New York -> Honolulu
Anchorage -> New York
Anchorage -> San Jose
Honolulu -> New York
Honolulu -> San Francisco
Denver -> San Jose
San Francisco -> New York
San Francisco -> Honolulu
San Francisco -> Denver
```

Your program should:

- Read in the flight information from the file flights.txt and store it in an appropriate data structure.
- Display the complete list of cities.
- Allow the user to select a city from which to start.
- In a loop, print out all the destinations that the user may reach directly from the current city, and prompt the user to select the next city.
- Once the user has selected a round-trip route (i.e., once the user has selected a flight that returns them to the starting city), exit from the loop and print out the route that was chosen.

A critical issue in building this program is designing appropriate data structures to keep track of the information you'll need in order to produce flight plans. You'll need to both have a way of keeping track of information on available flights that you read in from the flights.txt file, as well as a means for keeping track of the flight routes that the user is choosing in constructing their flight plan. Consider how both `ArrayList`s and `HashMap`s might be useful to keep track of the information you care about.

7. OBJECTS AND CLASSES

7.1. Halloween

For the program below, trace through its execution by hand to show what output is produced when it runs.

```
public class Halloween extends ConsoleProgram {
    public void run() {
        int halloweenTown = 10;
        Skeleton bones = new Skeleton("bones");
        Pumpkin king = new Pumpkin(halloweenTown, bones);
        Skeleton skellington = bones;
        skellington.setName("skellington");
        halloweenTown = 5;
        println(king.toString());
    }
}

public class Pumpkin extends ConsoleProgram {
    private int x;
    private Skeleton y;

    public Pumpkin(int z, Skeleton w) {
        x = z;
        y = w;
    }
    public String toString() {
        return y.getName() + " " + x;
    }
}

public class Skeleton extends ConsoleProgram {
    private String name;

    public Skeleton(String n) {
        name = n;
    }
    public String getName() {
        return name;
    }
    public void setName(String newName) {
        name = newName;
    }
}
```

Output:	

7.2. Bank Account toString

Add the following method to the BankAccount class:

```
public String toString()
```

Your method should return a string that contains the account's name and balance separated by a comma and space. For example, if an account object named benben has the name "Benson" and a balance of 17.25, the call of benben.toString() should return:

Benson, $17.25

There are some special cases you should handle. If the balance is negative, put the - sign before the dollar sign. Also, always display the cents as a two-digit number. For example, if the same object had a balance of -17.5, your method should return:

Benson, -$17.50

Your code is being added to the following class:

```
public class BankAccount {
    private String name;
    private double balance;

    // // your code goes here
}
```

7.3. Bank Account Transfer

Suppose that you are provided with a pre-written class BankAccount as shown below. (The headings are shown, but not the method bodies, to save space.) Assume that the fields, constructor, and methods shown are already implemented. You may refer to them or use them in solving this problem if necessary.

Write a method named **transfer** that will be placed inside the BankAccount class to become a part of each BankAccount object's behavior. The transfermethod moves money from this bank account to another account. The method accepts two parameters: a second BankAccount to accept the money, and a real number for the amount of money to transfer.

There is a $5.00 fee for transferring money, so this much must be deducted from the current account's balance before any transfer.

The method should modify the two BankAccount objects such that "this" current object has its balance decreased by the given amount plus the $5 fee, and the other BankAccount object's balance is increased by the given amount. A transfer also counts as a transaction on both accounts.

If this account object does not have enough money to make the full transfer, transfer whatever money is left after the $5 fee is deducted. If this account has under $5 or the amount is 0 or less, no transfer should occur and neither account's state should be modified.

```
// A BankAccount keeps track of a user's money balance and ID,
// and counts how many transactions (deposits/withdrawals) are made.
public class BankAccount {
    private String id;
    private double balance;
    private int transactions;

    // Constructs a BankAccount object with the given id, and
    // 0 balance and transactions.
    public BankAccount(String id)

    // returns the field values
    public double getBalance()
    public String getID()
    public String getTransactions()

    // Adds the amount to the balance if it is between 0-500.
    // Also counts as 1 transaction.
    public void deposit(double amount)

    // Subtracts the amount from the balance if the user has enough money.
    // Also counts as 1 transaction.
    public void withdraw(double amount)

    // your method would go here
}
```

For example, given the following `BankAccount` objects:

```
BankAccount ben = new BankAccount("Benson");
ben.deposit(90.00);
BankAccount mar = new BankAccount("Marty");
mar.deposit(25.00);
```

Assuming that the following calls were made, the balances afterward are shown in comments to the right of each call:

```
ben.transfer(mar, 20.00);     // ben $65, mar $45   (ben loses $25, mar gains $20)
ben.transfer(mar, 10.00);     // ben $50, mar $55   (ben loses $15, mar gains $10)
ben.transfer(mar, -1);        // ben $50, mar $55   (no effect; negative amount)
mar.transfer(ben, 39.00);     // ben $89, mar $11   (mar loses $44, ben gains $39)
mar.transfer(ben, 50.00);     // ben $95, mar $ 0   (mar loses $11, ben gains $ 6)
mar.transfer(ben,  1.00);     // ben $95, mar $ 0   (no effect; no money in account)
ben.transfer(mar, 88.00);     // ben $ 2, mar $88   (ben loses $93, mar gains $88)
ben.transfer(mar,  1.00);     // ben $ 2, mar $88   (no effect; can't afford fee)
```

7.4. Circle

Write a class of objects named `Circle` that remembers information about a circle. You must include the following public members. It may help you to know that there is a constant named `Math.PI` storing the value of π, roughly `3.14159`.

member name	description
`Circle(`*r*`)`	constructs a new circle with the given radius as a real number
`area()`	returns the area occupied by the circle
`circumference()`	returns the distance around the circle
`getRadius()`	returns the radius as a real number
`toString()`	returns a string representation such as `"Circle{radius=2.5}"`

You should define the entire class including the class heading, the private fields, and the declarations and definitions of all the public methods and constructor.

7.5. Date

Write a class of objects named `Date` that remembers information about a month and day. Ignore leap years and don't store the year in your object. You must include the following public methods:

member name	description
`Date(`*`m, d`*`)`	constructs a new date representing the given month and day
`daysInMonth()`	returns the number of days in the month stored by your date object
`getDay()`	returns the day
`getMonth()`	returns the month
`nextDay()`	advances the Date to the next day, wrapping to the next month and/or year if necessary
`toString()`	returns a string representation such as `"07/04"`

You should define the entire class including the class heading, the private fields, and the declarations and definitions of all the public methods and constructor.

7.6. Date-absoluteDay

Write a method named **absoluteDay** that will be placed in the `Date` class. The method should return the "absolute day of the year" between 1 and 365 represented by the Date object. January 1 is absolute day #1, January 2 is day #2, ..., and December 31st is absolute day #365. For example, calling this method on a date representing February 13th should return 44, and calling it on a Date representing September 19th should return 263. The table below summarizes the absolute days for various dates throughout the year.

Date (month/day)	1/1	1/2	1/3	...	1/30	1/31	2/1	2/2	...	2/28	3/1	3/2	...	12/30	12/31
Absolute day	1	2	3	...	30	31	32	33	...	59	60	61	...	364	365

The following table shows the results of calling your method on several `Date` objects.

Object	Call	Returns

`Date jan1 = new Date(1, 1);`	`jan1.absoluteDay()`	1
`Date jan4 = new Date(1, 4);`	`jan4.absoluteDay()`	4
`Date feb1 = new Date(2, 1);`	`feb1.absoluteDay()`	32
`Date mar10 = new Date(3, 10);`	`mar10.absoluteDay()`	69
`Date sep19 = new Date(9, 19);`	`sep19.absoluteDay()`	262
`Date dec31 = new Date(12, 31);`	`dec31.absoluteDay()`	365

You should not solve this problem by writing 12 `if` statements, one for each month; this is redundant and poor style. If you solve the problem that way, you will receive at most half credit. Your method also should not modify the state of the fields of the `Date` object on which it was called. A solution that does so will receive at most half credit.

Recall that your method is allowed to call other methods on `Date` objects or construct other objects if you like.

7.7. Clock-advance

Suppose that you are provided with a pre-written class `Clock` as described below. (The headings are shown, but not the method bodies, to save space.) Assume that the fields, constructor, and methods shown are already implemented. You may refer to them or use them in solving this problem if necessary.

```
// A Clock object represents an hour:minute time during
// the day or night, such as 10:45 AM or 6:27 PM.
public class Clock {
    private int hour;
    private int minute;
    private String amPm;

    // Constructs a new time for the given hour/minute
    public Clock(int h, int m, String ap)

    // returns the field values
    public int getHour()
    public int getMinute()
    public String getAmPm()

    // returns String for time; for example, "6:27 PM"
    public String toString()

    // your method would go here
}
```

Write an instance method named `advance` that will be placed inside the `Clock` class to become a part of each `Clock` object's behavior. The advance method accepts a number of minutes as its parameter and moves your object forward in time by that amount of minutes. The minutes passed could be any non-negative number, even a large number such as 500 or 1000000. If necessary, your object might wrap into the next hour or day, or it might wrap from the morning ("AM") to the evening ("PM") or vice versa. A Clock object doesn't care about what day it is; if you advance by 1 minute from 11:59 PM, it becomes 12:00 AM.

For example, if the following object is declared in client code:

```
Clock time = new Clock(6, 27, "PM");
```

The following calls to your method would modify the object's state as indicated in the comments:

```
time.advance(1);       //  6:28 PM
time.advance(30);      //  6:58 PM
time.advance(5);       //  7:03 PM
time.advance(60);      //  8:03 PM
time.advance(128);     // 10:11 PM
time.advance(180);     //  1:11 AM
time.advance(1440);    //  1:11 AM  (1 day later)
time.advance(21075);   //  4:26 PM  (2 weeks later)
```

Assume that the state of the object is valid at the start of the call and that the amPm field stores either "AM" or "PM".

7.8. Student

Define a class named **Student**. A Student object represents a university student that, for simplicity, just has a name, ID number, and number of units earned towards graduation. Each Student object should have the following public behavior:

- new Student(*name*, *id*)
  Constructor that initializes a new Student object storing the given name and ID number, with 0 units.
- *s*.getName()
  s.getID()
  s.getUnits()
  Returns the name, ID, or unit count of the student, respectively.
- *s*.incrementUnits(*units*);
  Adds the given number of units to this student's unit count.
- *s*.hasEnoughUnits()
  Returns whether the student has enough units (180) to graduate.
- *s*.toString()
  Returns the student's string representation, e.g. "Nick (#42342)".

7.9. Time Span

Define a class named **TimeSpan**. A TimeSpan object stores a span of time in hours and minutes (for example, the time span between 8:00am and 10:30am is 2 hours, 30 minutes). Each TimeSpan object should have the following public methods:

- new TimeSpan(*hours*, *minutes*)
  Constructs a TimeSpan object storing the given time span of hours and minutes.

- `getHours()`
  Returns the number of hours in this time span.
- `getMinutes()`
  Returns the number of minutes in this time span, between 0 and 59.
- `add(`*`hours, minutes`*`)`
  Adds the given amount of time to the span. For example, (2 hours, 15 min) + (1 hour, 45 min) = (4 hours). Assume that the parameters are valid: the hours are non-negative, and the minutes are between 0 and 59.
- `add(`*`timespan`*`)`
  Adds the given amount of time (stored as a time span) to the current time span.
- `getTotalHours()`
  Returns the total time in this time span as the real number of hours, such as `9.75` for (9 hours, 45 min).
- `toString()`
  Returns a string representation of the time span of hours and minutes, such as `"28h46m"`.

The minutes should always be reported as being in the range of 0 to 59. That means that you may have to "carry" 60 minutes into a full hour.