Problem 1 Write the Java method below, which takes a two-dimensional array *A* of positive integers. The method returns the index of the row that has the largest sum. Assume no two rows have the same sum. For example, if *A* is the array below, then row 0's values sum to 21, row 1's values sum to 19, row 2's values sum to 23, and row 3's values sum to 20. Since row 2 has the highest sum, the method would return the value 2.

	0	1	2	3
0	5	10	1	5
1	1	2	1	15
2	8	7	5	3
3	6	1	12	1

```
public static int indexOfHighestRowSum(int [][] A)  {...}
```

Problem 2 Write the Java method below that takes an integer array *freq* and a character array *text*. These two arrays have the same length. The method returns a character array that has *freq*[0] repetitions of the first character in the *text* array, then *freq*[1] repetitions of the next character in the *text* array, and so on. For example, if the arrays *freq* and *text* are as follows:

freq	3	1	5	2

text	m	s	k	p

The method would return the following array:

m	m	m	s	k	k	k	k	k	p	p

```
public static char [] repeatChars(int [] freq, char [] text) {...}
```

Problem 3 Write a Java program that prompts the user to enter integers from the keyboard one at a time. The program stops reading integers once the user enters the same value three times consecutively (meaning three times, one after the other). Once input is completed the program is to display the message "`Same entered 3 in a row`". Here are two ***sample*** runs:

```
Enter an integer:  77
Enter an integer:  5
Enter an integer:  5
Enter an integer:  5

Same entered 3 in a row.
```

```
Enter an integer:  77
Enter an integer:  77
Enter an integer:  12
Enter an integer:  77
Enter an integer:  20
Enter an integer:  20
Enter an integer:  185
Enter an integer:  185
Enter an integer:  185

Same entered 3 in a row.
```

```
public static void main(String[] args)  {...}
```

Problem 4 Write a Java program that has the user enter three decimal numbers. The program then outputs whether they are entered in strictly increasing order, strictly decreasing order, or neither. Here are sample runs:

```
Enter value:  3.75
Enter value:  5.9
Enter value:  11.3

INCREASING!
```

```
Enter value:  4.7
Enter value:  4.69
Enter value:  4.61

DECREASING!
```

```
Enter value:  3.7
Enter value:  5.9
Enter value:  1.5

NEITHER!
```

```
Enter value:  3.75
Enter value:  3.75
Enter value:  8.98

NEITHER!
```

```
public static void main(String[] args) {...}
```

Problem 5 Write the Java method below that takes two integer arrays *A* and *B* and returns true only if the values in *A* appear consecutively in *B*. You may assume all values in *A* are distinct. However, the values in *B* may or may not include duplicates. Here are several sample inputs and return values:

Argument Array *A*	**Argument Array *B***	**Return Value**
55 | 33 | 22	7 | 6 | **55** | **33** | **22** | 1	true
5 | 3 | 2	5 | 66 | 44 | 3 | 2 | 11	false
88 | 17 | 9 | 50	77 | **88** | **17** | **9** | **50** | 11 | 4 | -3	true
5 | 3 | 2	5 | 3 | 3 | 3 | 2	false
50 | 49 | 48 | 60 | 53	50 | 49 | 48 | 60	false

```
public static boolean appearsConsecutively(int [] A, int [] B) {...}
```

Problem 6 Write a Java program that prompts the user to enter three integers *a*, *b*, and *c*. As output the program is to display the number of integers entered by the user that are odd. The output statement must exactly match the sample output statements provided below. Here are four sample program runs:

```
Enter a:  32
Enter b:  47
Enter c:  59

Two integers were odd.
```

```
Enter a:  33
Enter b:  103
Enter c:  97

Three integers were odd.
```

```
Enter a:  50
Enter b:  940
Enter c:  99

One integer was odd.
```

```
Enter a:  402
Enter b:  1000
Enter c:  2

No integers were odd.
```

```
public static void main(String[] args)  {...}
```

Problem 7 Write a Java program that prompts the user to enter two integers *a* and *b*. You may assume without checking that $b > 0$. Without using Math.pow or any other library methods, compute and output the value a^b. For example, if the user enters $a = 2$ and $b = 5$, the program would output the value 32 since $2^5 = 2 \times 2 \times 2 \times 2 \times 2 = 32$. Here are two sample program runs:

```
Enter a:  2
Enter b:  5

2 to the power 5 is 32.
```

```
Enter a:  10
Enter b:  4

10 to the power 4 is 10000.
```

```
public static void main(String[] args)  {...}
```

Problem 8 Write the following method:

```
public static int howManyInCommon(int [] A, int [] B)  {...}
```

The method receives two integer arrays *A* and *B* as parameters. Both *A* and *B* are already in sorted order from lowest to highest. Array *A* has no repeated integers, and array *B* has no repeated integers, but there may be integers that appear in both array *A* and array *B*. Arrays *A* and *B* do not necessarily have the same length.

The method is to return the number of values that appear in both array *A* and array *B*. For example, if *A* and *B* were the arrays below, the return value would be 3, since the arrays have 3 values in common: 15, 22, and 93.

	0	1	2	3	4	5	6
A	6	8	15	22	77	93	98

	0	1	2	3	4
B	5	15	22	44	93

```
public static int howManyInCommon(int [] A, int [] B)  {...}
```

Problem 9 Write the following method:

```
public static int [] newSmallerArray(int [] A, int b) {...}
```

The method receives two parameters, an integer array *A* and an integer variable *b*. The method returns a new array (do not change *A*) that is identical to *A* except that all cells that contain the value stored in the variable *b* are no longer present. For example, if *b* = 77 and *A* is:

0	1	2	3	4	5	6	7	8	9
6	103	77	49	0	83	77	77	444	444

then the new, returned array would be the following:

0	1	2	3	4	5	6
6	103	49	0	83	444	444

```
public static int [] newSmallerArray(int [] A, int b) {...}
```

Problem 10 Write the following method:

```
public static int indexOfFirstPair(int [] myArray)  {...}
```

The method receives a single parameter, an integer array myArray. The method is to find the location of the first pair of adjacent matching values in the array. Once found, the method is to return the index of the first value of this pair. Note: this must be exactly a pair – it cannot be part of three or more values in a row. If there is no such index, the method is to return the value –1 (negative 1).

In the following example, the value 88 appears as three in a row, so it is not an exact pair. The value 77 appears as four in a row, so again, it is not a pair. The values 44 and 99 have exactly two consecutive values in a row. The two 44 values come before the 99 pair, so the return value is the index of the first 44, which is 10.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
88	88	88	63	29	77	77	77	77	50	44	44	8	0	99	99

Problem 11 Write the method **public static int** rangeProduct(**int** a, **int** b). Assume that $a < b$. Your method must compute and return the product of the integers in the range from a to b. For example, if $a = 3$ and $b = 6$, your method would compute and return the product $3 \times 4 \times 5 \times 6 = 360$.

```
public static int rangeProduct(int a, int b)  {...}
```

Problem 12 Write a program that has the user enter three integers *a*, *b*, and *c*. The user may enter the same value more than once. Print to the console the number of times the largest integer among *a*, *b*, and *c* was entered. Here are some sample runs:

```
Enter a:  32
Enter b:  48
Enter c:  32

The largest integer 48
was entered once.
```

```
Enter a:  923
Enter b:  188
Enter c:  923

The largest integer 923
was entered twice.
```

```
Enter a:  50
Enter b:  50
Enter c:  50

The largest integer 50
was entered three times.
```

```
public static void main(String[] args)  {...}
```

Problem 13 Write a program that has the user enter integers between 0 and 9999 until the user enters some integer twice. Your program then prints to the console window how many distinct integers were entered. You may assume all integers entered by the user are between 0 and 9999. Here are some sample runs:

```
Enter an int:  32
Enter an int:  32

You entered 1 value.
```

```
Enter an int:  32
Enter an int:  9285
Enter an int:  575
Enter an int:  7153
Enter an int:  8
Enter an int:  575

You entered 5 values.
```

```
Enter an int:  7
Enter an int:  6
Enter an int:  5
Enter an int:  1
Enter an int:  4
Enter an int:  3
Enter an int:  2
Enter an int:  1

You entered 7 values.
```

```
public static void main(String[] args)  {...}
```

Problem 14 Write the method: **public static int** largestInCommon(**int** [] A, **int** [] B). This method takes two arrays *A* and *B*, each containing positive integers only (no error checking necessary) and returns the largest value that is common to both *A* and *B*. If there is no value that is contained in both *A* and *B*, return the value –1. Here are some examples:

If *A* = {3, 8, 5, 2, 7, 9} and *B* = {5, 1, 22, 7, 2, 15, 3}, the return value would be 7.
If *A* = {35, 12, 19, 35, 45} and *B* = {55, 99, 12}, the return value would be 12.
If *A* = {33, 11, 77, 44, 55} and *B* = {99, 88, 222, 66, 1000}, the return value would be –1

```
public static int largestInCommon(int [] A, int [] B)  {...}
```

Problem 15 In US currency, a quarter is worth 25 cents, a nickel is worth 5 cents, and a penny is worth 1 cent. Write a Java program that prompts the user to enter an integer representing a dollar amount that consists of all pennies. You may assume the user input value is greater than zero. The program is to distribute that amount into quarters, nickels and pennies, using as few coins as possible. Note: you may not use any currency values other than quarters, nickels, and pennies. Here are ***sample*** runs:

```
Enter an integer:  67

To make 67 cents, you need:

2 quarters
3 nickels
2 pennies
```

```
Enter an integer:  184

To make 84 cents, you need:

7 quarters
1 nickels
4 pennies
```

```
public static void main(String[] args)  {...}
```

Problem 16 Write the method below that takes four integer arguments and returns **true** if and only if at least three of the four values are the same. For example, if the values passed in are 5, 3, 5, 5, it would return **true**. If the values passed in are 7, 3, 7, 9, it would return **false**.

```
public static boolean atLeastThreeMatch(int a, int b, int c, int d) {...}
```

Problem 17 Write the method below that takes two integer arguments *a* and *b*. The method prompts the user to enter integers on the keyboard one at a time, until *a* and *b* have both been entered. Date entry terminates once both values have been entered. The method then prints to the console window the number of integers entered. For example, if *a* were 25 and *b* were 17, here are two sample runs of your method:

```
Enter an int:  948
Enter an int:  49
Enter an int:  17
Enter an int:  0
Enter an int:  25

You had to enter 5 integers
before typing both 25 and 17.
```

```
Enter an int:  6
Enter an int:  25
Enter an int:  10
Enter an int:  25
Enter an int:  25
Enter an int:  109
Enter an int:  25
Enter an int:  17

You had to enter 8 integers
before typing both 25 and 17.
```

```
public static void howManyEnteredBeforeSeenBoth(int a, int b)  {...}
```

Problem 18 Write the method below that takes an integer array *myArray*. The method returns the value in array *myArray* that appears the most. For example, if *myArray* were the array below, the method would return the value 39, because 39 appears five times in *myArray* and no other integer appears that many times. You may assume that some value in *myArray* does appear more frequently than all others.

myArray	81	81	39	17	39	17	17	17	81	81	39	5	39	39

```
public static int mostFrequentValue(int [] myArray)  {...}
```

Problem 19 Write the method **public static int** longestRun(**int** [] myArray) that returns the length of the longest run in array *myArray*. For example, if *myArray* were the array below, the method would return the value 3, because the integer 17 appears three times in a row consecutively and no other integer in *myArray* appears consecutively more than three times. You may assume that there is only one longest run—there are no ties for the longest run.

A	81	81	39	17	39	17	17	17	81	81	39	5	39	39

```
public static int longestRun(int [] myArray) {...}
```

Problem 20 This method takes an array *myA* of doubles. It returns a new array of length one longer than the original array. The average of the values in *myA* is included in the new, returned array, and furthermore, all values in *myA* that are greater than the average appear to the ***right*** of the average in the new array, and all values that are smaller than the average appear to the left of the average in the new array. For example, if *myA* is

0	1	2	3	4	5	6
1.5	2.0	1.0	9.0	1.5	9.5	2.5

the average value is 3.85, and so the array you return could look like this:

0	1	2	3	4	5	6	7
1.5	2.0	1.0	1.5	2.5	3.0	9.0	9.5

```
public static double [] separate(double [] myA) {...}
```

Problem 21 A year *y* is defined to be a ***leap year*** as follows:

> Rule: If *y* is divisible by 4, then *y* is a leap year. Otherwise, it is **not** a leap year.
> Exception: If *y* is *also* divisible by 100, then *y* is ***not*** a leap year.
> Exception to the Exception: If *y* is also divisible by 400, then *y* **IS** a leap year.

For example: 2015, 2013, 2002, 1900, 1800, and 2100 are all ***not*** leap years, whereas 2016, 2012, 1976, 2000, 1600, 2400 ***are*** all leap years. Write the boolean method below which takes a positive integer *y* and returns **true** if *y* is a leap year and returns **false** if *y* is ***not*** a leap year.

```
public static boolean isLeapYear(int y) {...}
```

Problem 22 Write a Java program that prompts the user to enter integers from the keyboard one at a time. The program stops reading integers the first time an integer entered is smaller than the previous integer entered. The program then displays the last two integers entered. Here are two ***sample*** runs:

```
Enter an integer:  5
Enter an integer:  3

You entered 2 integers.
The last one entered was 5
and the one before that was 3.
```

```
Enter an integer:  5
Enter an integer:  9
Enter an integer:  12
Enter an integer:  37
Enter an integer:  20

You entered 5 integers.
The last one entered was 20
and the one before that was 37.
```

```
public static void main(String[] args)  {...}
```

Problem 23 Write the method **public static int** sumOfOdds(**int** [] myArray) that takes an integer array *myArray*. The method returns the sum of the odd integers in array *myArray*. For example, if *myArray* were the array below, the method would return the value 23, since the odd integers in *myArray* are 15, 7, and 1, and their sum is 23.

myArray	20	40	15	12	7	1	56

```
public static int sumOfOdds(int [] myArray) {...}
```

Problem 24 Write the Java method below, which takes a positive integer *n* and returns a reference to a new two-dimensional character array with *n* rows and *n* columns with the cells initialized to an 'X' pattern as shown in the following examples: If *n* = 5, the method would return a reference to an array initialized as shown below on the left. If *n* = 7, the method would return a reference to the array on the right (you may assume that *n* is odd):

	0	1	2	3	4
0	X	-	-	-	X
1	-	X	-	X	-
2	-	-	X	-	-
3	-	X	-	X	-
4	X	-	-	-	X

	0	1	2	3	4	5	6
0	X	-	-	-	-	-	X
1	-	X	-	-	-	X	-
2	-	-	X	-	X	-	-
3	-	-	-	X	-	-	-
4	-	-	X	-	X	-	-
5	-	X	-	-	-	X	-
6	X	-	-	-	-	-	X

```
public static char [][] xPatternArray(int n)  {...}
```

Problem 25 Write the method below that takes two arrays of characters *x* and *y* as arguments. The method creates and returns a new array of characters that is made by taking the vowels in array *x* and the consonants in array *y*. Remember that vowels are the letters 'A', 'E', 'I', 'O', and 'U' (and NOT 'Y'), and the consonants are all the other letters. Assume *x* and *y* contain ***only CAPITAL*** letters (***no*** lower case letters).

In the example below, the vowels in *x* are EAOUU, and the consonants in *y* are BZBHHP, so the new array returned would be the array containing the characters EAOUUBZBHHP in that order.

	0	1	2	3	4	5	6	7	8	9	10	11
x	E	Q	A	F	G	O	C	C	U	U	M	M

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
y	B	Z	B	U	U	H	H	O	A	E	A	E	P	A

	0	1	2	3	4	5	6	7	8	9	10
array returned	E	A	O	U	U	B	Z	B	H	H	P

```
public static char [] vowelConsonant(char [] x, char [] y)  {...}
```

Problem 26 Write the method **public static int** middleValue(**int** a, **int** b, **int** c) that takes three integer arguments and returns the value that would be in the middle if the arguments were written in sorted order. Assume all three integers passed in are distinct. Here are some examples:

- *middleValue*(4, 9, 7) would return 7.
- *middleValue*(40, 70, 90) would return 70.
- *middleValue*(700, 900, 400) would return 700.
- *middleValue*(8, 2, 5) would return 5.

```
public static int middleValue(int a, int b, int c)  {...}
```

Problem 27 Write the method **public static int** goingWhichWay(**int** [] myArray) that takes an integer array myArray. Do not change myArray. If myArray is already sorted in strictly increasing order, output "UP" to the console window, and return the value 111. If myArray is already sorted in strictly decreasing order, output "DOWN" to the console window and return the value 222. If neither of these is true for myArray, output "NEITHER" to the console window, and return the value 333. Examples:

myArray	3	8	17	25	89	94	should print "INCREASING" and return 111.
myArray	77	66	55	33	22	11	should print "DECREASING" and return 222.
myArray	20	30	50	40	10	5	should print "NEITHER" and return 333.
myArray	20	30	50	50	70	90	should print "NEITHER" and return 333 (it is not ***strictly*** increasing).

```
public static int goingWhichWay(int [] myArray)  {...}
```

Problem 28 Write the method **public static int** howManyStepsToGetTo1(**int** n) that takes a positive integer *n* as a parameter (no error checking needed) and performs the following procedure: If *n* is even, then divide it by 2. If *n* is odd, multiply it by 3 and then add 1. If the resulting value is 1, stop. Otherwise repeat this procedure until you get the value 1. The method returns the number of steps it takes the procedure to get to the value 1.

For example, if *n* = 1, the method returns 0, since *n* is already 1. If *n* = 3, the method would return the value 7, since it would require the following 7 steps to get to 1 after starting from 3:

1. Since 3 is odd, we multiply it by 3 and add 1 to get 10.
2. Since 10 is even, we divide it by 2 to get 5.
3. Since 5 is odd, we multiply it by 3 and add 1 to get 16.
4. Since 16 is even, we divide it by 2 to get 8.
5. Since 8 is even, we divide it by 2 to get 4.
6. Since 4 is even, we divide it by 2 to get 2.
7. Since 2 is even, we divide it by 2 to get 1. We stop now because we have reached 1.

```
public static int howManyStepsToGetTo1(int n)  {...}
```

Problem 29 Three integers *a*, *b*, and *c* can make a triangle if and only if all three are positive and no single one of them is greater than or equal to the sum of the other two. Write a Java program that prompts the user for three integers: *a*, *b*, and *c*. Then output whether these three values can form a triangle or not. Sample runs:

```
Enter a:  5
Enter b:  0
Enter c:  3

These CANNOT make a triangle.
```

```
Enter a:  7
Enter b:  15
Enter c:  4

These CANNOT make a triangle.
```

```
Enter a:  6
Enter b:  5
Enter c:  7

These CAN make a triangle.
```

```
Enter a:  20
Enter b:  100
Enter c:  100

These CAN make a triangle.
```

```
public static void main(String[] args)  {...}
```

Problem 30 Write a Java method that takes three integer arguments: *size*, *start*, and *increment*. The method returns a newly created integer array of length *size*, whose first value is the value in the variable *start*, and all subsequent values are larger than the previous value in the array by the amount *increment*. For example, a call to `createNewArray(8, 5, 4)` would return the following array (which has length 8, first value is 5, and all subsequent values are 4 more than the previous value):

0	1	2	3	4	5	6	7
5	9	13	17	21	25	29	33

```
public static int [] createNewArray(int size, int start, int increment) {...}
```

Problem 31 Write a Java program that prompts the user to enter integers and reads them in until they enter the integer 10. Once this happens, the program stops prompting for input, and then outputs the minimum value entered and the maximum value entered, and either of these may be the final 10 that is entered. Sample runs:

```
Enter int:  -5
Enter int:   9
Enter int:  -7
Enter int:   4
Enter int:  10

min:  -7
max:  10
```

```
Enter int:  20
Enter int:  35
Enter int:  15
Enter int:  10

min:  10
max:  35
```

```
Enter int:   7
Enter int:  12
Enter int:   6
Enter int:   9
Enter int:  10

min:  6
max:  12
```

```
Enter int:   10

min:  10
max:  10
```

```
public static void main(String[] args)  {...}
```

Problem 32 Write a Java method called *glaciered* that takes two integer arguments *k* and *m*, both of which you may assume are positive values. Return the largest integer that is both a factor of *k* and a factor of *m*. E

- For example, if *k* = 12 and *m* = 30, then all of the following are both factors of *k* and *m*: 1, 2, 3, and 6. The return value here would be 6, since that is the largest factor of both *m* and *k*.
- Say *k* = 80 and *m* = 20. The common factors are: 1, 2, 4, 5, 10, and 20, so the method would return 20.
- Final example: *k* = 15 and *m* = 32. The only common factor is 1, so the method would return 1 here.

```
public static int glaciered(int k, int m)  {...}
```

Problem 33 Write a Java method that takes an array of integers, all of which you may assume are positive. The method returns the first (leftmost) value in the array that appears 3 or more times in a row (i.e., consecutively). If no value appears 3 times in a row, return the value –1 (negative 1). For the following ***example***, the value 12 would be returned, since 12 is the first integer to appear three or more times in a row.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	20	20	7	7	5	12	12	12	12	7	1	1	1	20

```
public static int firstValueThreeInARow(int [] myArray) {...}
```

Problem 34 Write a Java program that prompts the user for a positive integer *n* and reads an integer from the keyboard. ***You must error check here***: if the user does not enter a positive integer (at least 1), then prompt again and read again and keep doing so until the user does enter a positive integer *n*. Then compute and output to the console window the product of all the integers between 1 and *n* inclusive. For example, if user enters positive integer 5, you would output to the console the value 120, since $1 \times 2 \times 3 \times 4 \times 5 = 120$. Do ***not*** output the value 0—if you do, you fail this question.

```
public static void main(String[] args)  {...}
```

Problem 35 Write the Java method ***canFormRightTriangle*** that determines if its three integer arguments *a*, *b*, and *c* can be the lengths of the sides of a *right* triangle. Three integers can be the lengths of the sides of a right triangle if and only if all three are positive and the square of the largest integer is equal to the sum of the squares of the other two integers. Example: if $a = 4$, $b = 5$, and $c = 3$, then the method would return true, since $b^2 = 5^2 = 25 = 9 + 16 = 3^2 + 4^2 = c^2 + a^2$. Other examples:

The following calls should return *true*:	**The following calls should return *false*:**
`canFormRightTriangle(3, 4, 5)`	`canFormRightTriangle(3, -4, 5)`
`canFormRightTriangle(5, 3, 4)`	`canFormRightTriangle(-5, -4, -3)`
`canFormRightTriangle(6, 10, 8)`	`canFormRightTriangle(0, 0, 0)`
`canFormRightTriangle(10, 8, 6)`	`canFormRightTriangle(5, 0, 5)`
`canFormRightTriangle(12, 5, 13)`	`canFormRightTriangle(1, 2, 3)`

```
public static boolean canFormRightTriangle(int a, int b, int c)  {...}
```

Problem 36 Write a Java program that prompts the user to enter integers one at a time, reads them, and continues to do so until the user enters an integer that is the sum of all previous integers entered. Then stop and output how many integers were entered. Sample runs:

```
Enter int:   2
Enter int:   5
Enter int:   7

You entered 3 ints.
```

```
Enter int:   1
Enter int:   3
Enter int:   5
Enter int:  20
Enter int:  29

You entered 5 ints.
```

```
Enter int:   5
Enter int:  -1
Enter int:  10
Enter int:  10
Enter int:  25
Enter int:  49

You entered 6 ints.
```

```
public static void main(String[] args)  {...}
```

Problem 37 Write a Java method which takes an array of integers as its argument. The method determines which is greater: the sum of the even integers in the array, or the sum of the odd integers in the array, ***or*** if they tie. If the sum of the **evens** is greater, output to the console window "EVENS ARE GREATER" ***and return the value* 2**. If the sum of the **odds** is greater, output to the console window "ODDS ARE GREATER" ***and return the value* 3**. If the two sums tie, then output "TIE" to the console window, ***and return the value* 7**. Examples:

Argument array *A*	Output to the console window	Return Value
5 | 10 | 2 | 7	TIE	7
1 | 3 | 9 | 88 | 5 | 1 | 1	EVENS ARE GREATER	2
2 | 9 | 4 | 11 | 5 | 13 | 20 | 99 | 6	ODDS ARE GREATER	3

```
public static int whichSumIsLarger(int [] myArray) {...}
```

Problem 38 Write the Java method ***interleave***, which takes two integer arrays *A* and *B*, not necessarily the same length, and returns a ***new*** integer array (don't alter *A* or *B*) whose first value is the first value in *A*, and whose 2nd value is the first value in *B* and whose 3rd value is the next value in *A*, and whose 4th value is the next in *B*, etc. If one array runs out, then just copy the rest of the other one into the new array. Here are some examples of arrays *A* and *B* that could be passed in, along the with array that your method should return:

Argument Array *A*	**Argument Array *B***	**New Array *Returned* By Method**
1 | 2 | 3	44 | 55 | 66 | 77 | 88	1 | 44 | 2 | 55 | 3 | 66 | 77 | 88
40 | 30 | 60 | 80 | 70	5 | 2	40 | 5 | 30 | 2 | 60 | 80 | 70
7 | 8 | 9	33 | 22 | 11	7 | 33 | 8 | 22 | 9 | 11

```
public static int [] interleave(int [] A, int [] B)   {...}
```

Problem 39 Write the main method of a program that prompts the user to enter three integers and then displays their sum, unless the sum is a multiple of either 5 or 3 in which case display twice the sum, unless the sum is a multiple of both 5 and 3 in which case display triple the sum. The output must be formatted exactly like below and assume the Scanner class has been imported. Here are three sample runs:

```
Enter an int: 1
Enter another int: 12
One more, please: 4

Result: 17
```

```
Enter an int: 11
Enter another int: 2
One more, please: 7

Result: 40
```

```
Enter an int: 6
Enter another int: 4
One more, please: 5

Result: 45
```

```
public static void main(String[] args)  {...}
```

Problem 40 Write the body of a boolean method named *isPrime* that accepts one integer parameter *n* and returns **true** if *n* is prime and **false** otherwise. Remember that a number is prime when only two numbers divide into it without producing a remainder, 1 and itself. Here are three examples:

Example 1: `isPrime(3)` returns **`true`**

Example 2: `isPrime(9)` returns **`false`**

Example 3: `isPrime(7)` returns **`true`**

```
public static void main(String[] args)  {...}
```

Problem 41 Write an integer method named *elementsBiggerThan* that accepts an array of integers called *a* and an integer called *n* as parameters then returns the number of elements in the array that are greater than *n*. Here are three ***examples***:

Example 1: Assume `a = {4, 2, 1, 3, 9}` then `elementsBiggerThan(a, 0)` returns **`5`**

Example 2: Assume `a = {10, 2, 13, 43, 19, 42}` then `elementsBiggerThan(a, 40)` returns **`2`**

Example 3: Assume `a = {1, 10, -7, 5, 10}` then `elementsBiggerThan(a, 10)` returns **`0`**

```
public static int elementsBiggerThan(int[] a, int n) {...}
```

Problem 42 Write the main method of a class that asks the user to enter three integers *a*, *b*, and *c*. After the integers have been entered, the program computes the sum. If the sum is a multiple of 8 the program displays "red" (unless the last digit of the sum is also a 4 or an 8 in which case it should display "blue" instead of "red"), otherwise the program should display "green". **The output must be formatted exactly like below**. Here are five sample runs:

```
Enter a: 3
Enter b: 11
Enter c: 4

green
```

```
Enter a: 20
Enter b: 3
Enter c: 1

blue
```

```
Enter a: 2
Enter b: 1
Enter c: 1

green
```

```
Enter a: 16
Enter b: 8
Enter c: 8

red
```

```
Enter a: 21
Enter b: 22
Enter c: 5

blue
```

```
public static void main(String[] args)  {...}
```

Problem 43 Write the Java method below, which takes a positive integer *n* and returns a reference to a new two-dimensional integer array with *n* rows and *n* columns with the cells initialized with the triangle pattern given in the examples below. For example, if *n* = 4, the method would return a reference to an array initialized as shown below on the left. If n = 6, the method would return a reference to the array on the right (the values left blank do not need to be initialized):

	0	1	2	3
0	0			
1	0	1		
2	0	1	2	
3	0	1	2	3

	0	1	2	3	4	5
0	0					
1	0	1				
2	0	1	2			
3	0	1	2	3		
4	0	1	2	3	4	
5	0	1	2	3	4	5

```
public static int [][] numberTriangleArray(int n)  {...}
```

Problem 44 When one number divides another without leaving a remainder the first number is called a factor of the second. A number *n* is said to be perfect if the sum of the factors which are less than *n* are equal to *n*. For example, 28 is a perfect number because the sum of its factors which are less than 28 (1, 2, 4, 7, and 14) equals 28. Write a boolean method named *isPerfect* that has one integer parameter *n*, which returns `true` if *n* is perfect or `false` if *n* is not perfect. Assume *n* is positive. Here are three more examples:

Example 1: `isPerfect(12)` returns `false`

Example 2: `isPerfect(6)` returns `true`

Example 3: `isPerfect(20)` returns `false`

```
public static boolean isPerfect(int n) {...}
```

Problem 45 Write an integer method named *addEvenSubOdd* that accepts an array of integers called *a* as a parameter then returns the difference between the sum of elements in the array that are even minus the sum of the elements in the array that are odd. Here are three ***examples***:

Example 1: Assume `a = {4, 2, 1, 3, 9}` then `addEvenSubOdd(a)` returns `-7`

Example 2: Assume `a = {10, 0, 11, 44, 19, 46}` then `addEvenSubOdd(a)` returns `70`

Example 3: Assume `a = {1, 10, -7, 5, 10}` then `addEvenSubOdd(a)` returns `21`

```
public static int addEvenSubOdd(int[] a) {...}
```

Problem 46 Write a method that accepts two parameters: a two-dimensional integer array named *a* and an integer value named *x*. The method makes a new array with the column at index *x* removed from *a*. If *x* is not a valid column index, no columns are removed from the new array. Under no circumstances should you return a reference to the original array. Assume *a* has at least three columns. The method returns a reference to the newly created array. Here are two ***examples***:

Example 1: `a = {{3, 6, 7, 2}, {1, 0, 5, 9}}` and `x = 2` return a reference to `{{3, 6, 2}, {1, 0, 9}}`

Example 2: `a = {{31, 3, 17, 15}, { 7, 40, 5, 9}, {12, 8, 89, 11}}` and `x = 5` return a reference to `{{31, 3, 17, 15}, { 7, 40, 5, 9}, {12, 8, 89, 11}}`

```
public static int[][] removeColumn(int[][] a, int x) {...}
```

Problem 47 Write the Java method below which takes an integer array a and computes and returns the difference of the sum of the integers in even indices minus the product of the integers in the even indices. For example, if a were this array:

	0	1	2	3	4	5	6
a	10	5	100	3	6	2	30

the sum of the integers in even indexed cells would be $10 + 100 + 6 + 30 = 146$. The product of the integers in cells with odd indices would be $5 \times 3 \times 2 = 30$. So the value returned would be $146 - 30 = 116$.

```
public static int evenSumMinusOddProduct(int [] a)  {...}
```

Problem 48 Write the main method of a class that asks the user to enter integers until a sequence of three consecutive increasing integers are entered. Then the program displays the sum of all of the integers that were entered. Here are three sample runs:

```
Enter an int: 3
Enter an int: 4
Enter an int: 5
12
```

```
Enter an int: 20
Enter an int: -2
Enter an int: -1
Enter an int: 0
17
```

```
Enter an int: 1
Enter an int: 7
Enter an int: 8
Enter an int: 2
Enter an int: 9
Enter an int: 10
Enter an int: 11
48
```

```
public static void main(String[] args)  {...}
```

Problem 49 Write a method named *areReversed* that takes two character arrays and returns true if they are the reverse of each other. For example, an array containing only the values {a, b, c} is the reverse of an array that contains only the values {c, b, a}. Here are five more ***examples***:

Argument Array *a*	**Argument Array *b***	**Return Value**
x f q	q f x	true
m s t r	r s t m	false
a b c d	f e d c b a	false
x y z	z y y y x	false
l m n o p	p o n m l	true

```
public static boolean areReversed(char[] a, char[] b) {...}
```

Problem 50 Write a method named *backAgain* that accepts one integer parameter *n* and creates an array containing the values in pattern 1, 2, …, *n* – 1, *n*, *n* – 1, …, 2, 1. Assume *n* is positive. Here are three ***examples***:

If `n = 3` then `backAgain(n)` returns a reference to an array containing the values `{1, 2, 3, 2, 1}`

If `n = 4` then `backAgain(n)` returns a reference to an array containing the values `{1, 2, 3, 4, 3, 2, 1}`

If `n = 2` then `backAgain(n)` returns a reference to an array containing the values `{1, 2, 1}`

```
public static int[] backAgain(int n) {...}
```