1. Code comprehension
To approach these problems, we can use

P(MMD)(AS)

1. Parentheses(): compute everything inside of the ()
2. Mod or multiplication or division: whichever comes first going left to right
3. Addition or subtraction: whichever comes first going from left to right

a.

```
(6-4)*3+12/5%2
  2*3+12/5%2
      6+12/5%2
      6+2%2
      6+0
      6
```

1. (): Even though we are subtracting, we must do the () first
2. No more () so now we can do the first (MMD) going left to right
3. Go to the next MMD since it has greater precedence than A; 12/5=2 since it is the largest rounded down whole number
4. 2%2= 0 since 2 divided by 2 is 1 with no remainder

b.

```
3*0.5+1+"7"+7+(6-2)
3*0.5+1+"7"+7+4
  1.5+1+"7"+7+4
      2.5+"7"+7+4
      "2.57"+7+4
      "2.577"+4
      "2.5774"
```

1. Begin with parentheses
2. Go left to right to find the first MMD
3. No more MMD → go left to right to find the first AS
4. When we add a String to a number, the outcome becomes a String
5,6. Similar logic from step 4; the number gets treated as a String and is added to the end

2.5 * 3 ≥ 10%6 || 13/4 != 3

7.5 ≥ 10%6 || 13/4 != 3

7.5 ≥ 4 || 13/4 != 3

true || 13/4 != 3

true || 3 != 3

true || false

true

1. When approaching these Problems with boolean conditions, we want to solve until we have values that are comparable
2. 10 % 6 = 4 since 10 divided by 6 is 1 remainder 4
3. Now, we have 2 comparable values. 7.5 ≥ 4, so this is true
4. 13/4 = 3 since integer division rounds down
5. Since 3 = 3, this is false
6. When you or true and false, the out come is true. When you and true and false, the outcome is false

true || false = true
true && false = false

b.

```
public static int m(int x, int y){
   while (x>0 && y>0) {
      x = x - y;
      y--;
      system.out.print(x + ", ");
   }
   return y;
}
```

☒ m(14,9);
☐ m(5,0);
☐ m(-17,-8);
☒ m(11,-3);
☒ m(10,10);

```
m(14, 9):
while (14 > 0 && 9 > 0){
   x = 14 - 9;          → x=5
   y--;                 → y=8
}
while (5 > 0 && 8 > 0){
   x = 5 - 8;           → x=-3
   y--;                 → y=7
}
return 7
```

Writing out what happens for each iteration of the loop can help us keep track of our x and y values

We can ignore the print statement since we only care about the return value

```
m(5,0);
return 0
```

we do not ~~enter~~ the while loop ~~condition~~ since y ≯ 0. Therefore, we return 0

```
m(-17,-8)
return -8
```

we do not enter the while loop since y ≯ 0 and x ≯ 0. Therefore, we return -8

```
m(11,-3)
return -3
```

We do not enter the while loop since y ≯ 0. Since we return -3, this would be odd

```
m(10, 10)
while (10>0 && 10>0) {
    x = 10 - 10;      → x = 0
    y--;              → y = 9
}
return 9
```

Since we return 9, this would be odd

C.

```
public static void mystery(int x, int y) {
    int z = 0;
    // Point A
    while (x<y) {
        // Point B
        z++;
        if (z % 2 == 0) {
            x = x * 2;
            // Point C
        } else {
            y--;
            // Point D
        }
    }
    // Point E
    S.o.pln(z);
}
```

A: We have not entered the while loop yet, so we do not know if these conditions are true

B: Since we entered the while loop, we will only reach point B if x<y

C: Since we entered the if statement, z % 2 == 0 at this point

D: Since we skipped past the if statement and entered the else branch, then z % 2 != 0

E: We exited the loop at point E, so x ≮ y

☐ At Point A, x<y must be true
☒ At Point B, x<y must be true
☒ At Point C, z % 2 == 0 must be true
☐ At Point D, z % 2 == 0 must be true
☐ At Point E, x<y must be true

2. Array Code Tracing

Consider the following method:

```
public static int[] mystery(int[][] list) {
    int[] result = new int[list.length];
    for (int i = 0; i < list.length; i++) {
        int n = 1;
        for (int j = 0; j < list[0].length; j++) {
            n *= list[i][j];
        }
        result[i] = n;
    }
    return result;
}
```

Creating a new array whose length is the number of rows in the 2D array "list"

array traversal going through each row in the outer loop, and each column in the inner loop.

For each element in an "inner" array, multiply them together.

→ n = n • list[i][j]

Set the index at "i" in the result array to be the multiplied values

Part A: Consider the following code:

```
int[][] arr = {{3, 15, 1},
               {-8, 1, 7},
               {7, 11, 0},
               {-1, -9, 4}};
int[] result = mystery(arr);
```

2D array being passed into the "mystery" method.

What are the contents in **`arr`** after this code is executed?

```
{{3, 15, 1},
 {-8, 1, 7},
 {7, 11, 0},
 (-1, -9, 4}}
```

The contents in arr are unchanged. The method only uses the values in arr without modifying the 2D array in any way.

Part B: Consider the following code:

```
int[][] arr = {{3, 15, 1},
               {-8, 1, 7},
               {7, 11, 0},
               {-1, -9, 4}};
int[] result = mystery(arr);
```

→ 3 • 15 • 1 = 45

→ -8 • 1 • 7 = -56

→ 7 • 11 • 0 = 0

→ -1 • -9 • 4 = 36

These values become the elements in the result array after the method finishes running.

What are the contents in **`result`** after this code is executed?

`{45, -56, 0, 36}`

Part C: Which of the following best describes what the method `mystery` does? (Choose **one**)

☐ Returns a new array holding the last element in each row of arr, and modifies arr to contain the products of each row.

☐ Returns a new array holding the last element in each column of arr, and modifies arr to contain the products of each column.

☐ Returns a new array holding the products of the elements in each row of `arr`, and fills `arr` with 1's.

☐ Returns a new array holding the products of the elements in each column of `arr`, and fills `arr` with 1's.

☑ **Returns a new array containing the products of the elements in each row of arr, leaving `arr` unchanged.**

☐ Returns a new array containing the products of the elements in each column of arr, leaving `arr` unchanged.

Option 1:

Incorrect, because the purpose of the "mystery" method is not to get the last element in each row of arr. Also, arr is not modified.

Option 2:

Incorrect, for nearly the same reason that "mystery" is not getting the last element in each column. Again, arr is not modified.

Option 3:

The first sentence is correct. The new array does contain the products in each row. But, arr does not become replaced with all 1's.

Option 4:

Correct, result contains the products of each row in arr

option 5:

Incorrect, result does not contain the product of each column, because the array traversal loops through the 2D array in the following way:

1) Start at i=0
2) Start at n=1
3) Start at j=0
4) Loop through arr[0].length, which is the length of 1 row in arr.
5) For each element (arr$\underset{row}{[i]}\underset{col}{[j]}$), multiply them together.
6) After the j loop finishes going through indices [0][0], [0][1], [0][2
set the index at result[i] = n
7) Repeat 2-6 for each value of i, when i < list.length, which is the number of rows in arr.

3. Debugging

Consider a static method called **`battle`** that simulates a battle between two players, which takes two parameters:

make sure to gather info about your parameters!

- `int minDamage` - the minimum amount of damage a player can inflict upon the other (guaranteed to be at least 0)
- `int maxDamage` - the maximum amount of damage a player can inflict upon the other (guaranteed to be greater than `minDamage`)

this means that the battle ends if one of the players' health hits below 1 (< 1)

players should randomly switch off

The "health" of a player is represented by a number initially set to 100. Each player randomly attacks the other, subtracting damage from the attacked player's health, until one of the player's health falls below 1. The next player to attack is **randomly** determined, and the damage inflicted is a **random number** between the minimum damage value and the maximum damage value (inclusive).

↳ this range: minDamage ≤ damage ≤ maxDamage

*notice key word: inclusive!

For example, suppose the following call was made:

```
battle(20, 50);
```

This call to a *correct* implementation of the method might produce output like the following. (Due to the randomness involved in the method, this exact output may not be produced every time it is run.):

notice how there is no mention about any "returns" ↳ void method!

```
Let's get ready to rumble!!!
Player 2 attacks! 35 damage...P1: 65, P2: 100
Player 1 attacks! 37 damage...P1: 65, P2: 63
Player 1 attacks! 43 damage...P1: 65, P2: 20
Player 2 attacks! 21 damage...P1: 44, P2: 20
Player 2 attacks! 43 damage...P1: 1, P2: 20
Player 2 attacks! 32 damage...P1: -31, P2: 20
Player 2 wins!
```

from the next page, we have exactly two bugs to fix (no more, no less). check out the next page for buggy output!

Consider the following proposed buggy implementation of **`battle`**:

20 50

```
public static void battle(int minDamage, int maxDamage) {
    System.out.println("Let's get ready to rumble!!!");
    Random r = new Random();
    int player = 0;
    int playerOneHealth = 100;
    int playerTwoHealth = 100;
    while (playerOneHealth > 0 || playerTwoHealth > 0) {
        int damage = r.nextInt(maxDamage - minDamage + 1);
        player = r.nextInt(2) + 1;
        if (player == 1) {
            playerTwoHealth -= damage;
        } else {
            playerOneHealth -= damage;
        }
        System.out.print("Player " + player + " attacks! " + damage + " damage");
        System.out.println("P1: " + playerOneHealth + ", P2: " + playerTwoHealth);
    }
    System.out.println("Player " + player + " wins!");
}
```

① with ||, the battle will continue until both players' healths hit < 1

consider: true || false => true

true && false => false

both conditions (health > 0 for both) must be true for battle to go on

① ~~||~~ &&

② + minDamage

② remember r.nextInt(max - min + 1) + min?

max - min + 1 gets the total number of values for our random number generation between min and max inclusive (our range)

↳ but we need to + min to ensure we are starting at our min value, not 0!

for battle(20, 50) => r.nextInt(50 - 20 + 1) + 20

↳ 31 numbers in range

↳ starting value

now, we have a range [20, 50] for our random number generation!

only two places to fix!

This implementation contains two bugs that are causing it to not work as intended!

For the same input as before, the buggy implementation might produce the following output:

definitely check out the buggy output for any bugs!

```
Let's get ready to rumble!!!
Player 1 attacks! 4 damage...P1: 100, P2: 96
Player 2 attacks! 11 damage...P1: 89, P2: 96
Player 2 attacks! 27 damage...P1: 62, P2: 96
Player 1 attacks! 16 damage...P1: 62, P2: 80
Player 2 attacks! 22 damage...P1: 40, P2: 80
Player 2 attacks! 24 damage...P1: 16, P2: 80
Player 1 attacks! 17 damage...P1: 16, P2: 63
Player 2 attacks! 23 damage...P1: -7, P2: 63
Player 1 attacks! 27 damage...P1: -7, P2: 36
Player 1 attacks! 18 damage...P1: -7, P2: 18
Player 2 attacks! 23 damage...P1: -30, P2: 18
Player 1 attacks! 19 damage...P1: -30, P2: -1
Player 1 wins!
```

shouldn't our range be between [20, 50]?

we should have ended the program here by now since P1 health already hit <1 above

Your task: Annotate (write on) the code below to indicate how you would fix the two bugs. You may add (using arrows to indicate where to insert), remove (by crossing out), or modify (with a combination) any code you choose. However, the fix should not require a lot of work.

You must *correctly identify* both of the lines with issues, or *correctly identify and fix* **one** of the bugs for an S grade.

You must *correctly identify* both of the lines with the bugs **and** *correctly fix* both of the bugs for an E grade.

```java
public static void battle(int minDamage, int maxDamage) {
    System.out.println("Let's get ready to rumble!!!");
    Random r = new Random();
    int player = 0;
    int playerOneHealth = 100;
    int playerTwoHealth = 100;
    while (playerOneHealth > 0 && playerTwoHealth > 0) {
        int damage = r.nextInt(maxDamage - minDamage + 1) + minDamage;
        player = r.nextInt(2) + 1;
        if (player == 1) {
            playerTwoHealth -= damage;
        } else {
            playerOneHealth -= damage;
        }
        System.out.print("Player " + player + " attacks! " + damage + " damage");
        System.out.println("P1: " + playerOneHealth + ", P2: " + playerTwoHealth);
    }
    System.out.println("Player " + player + " wins!");
}
```

④

```
public static int longWords(scanner input,
                            int numWords){
   //If we look at the first call to longWords,
   the output has 5 "Next word?" lines but
   only 4 "_ more words..." output
   //This indicates that we might need to
   fencepost → we will have one initial scanner
   call outside of the loop
   System.out.print("Next word? ");            Character
   String word = input.next();                 count and
   // Because we need to keep track of the ↙
   longest word throughout iterations, we
   can create variables to store these values
   String longest = word;          ← (word is currently
   int totalChars = word.length();    the longest since
                                      it is the only
                                      word)
        (fenceposting)
            ↓
   for (int i = 1; i < numWords; i++) {
      System.out.println((numWords - i) + " more words...");
      System.out.print("Next word? ");
      word = input.next();
      // we can add the length of each word to
      the character count variable to keep a
      running total
      totalChars += word.length();
      // We can also update the longest word
      if necessary per iteration
      if (word.length() > longest.length()) {
         longest = word;
      }
   }                                           return
}                                                totalChars;
S.o.pln("Longest word: " + longest);
```

5. General Programming 2

2 parameters

Write a static method named **`gumballTricks`** that accepts a Random object and an integer n that represents the number of tricks as parameters. Your method should use the Random object to randomly choose a trick from Gumball's repertoire: spin, bang!, and boop. Each outcome should be equally likely. Your method should print out each of the randomly-generated tricks followed by a space, and then both print and return the greatest number of spins that occurred in a row.

This method-
1) Takes in 2 parameters: Random, int
2) Prints values out
3) Returns an int: max # of spins

Assuming that the following variable has been initialized:

```
Random r = new Random();
```

Here are some example calls to the method with their resulting console output and return value:

Call	Console Output	Returned

`gumballTricks(r, 8);`	`spin boop bang! spin spin spin bang! spin`
	`Run of 3 spins after 8 tricks.`
	longest accurrence of spins in a row
		3

`gumballTricks(r, 5);`	`bang! boop boop bang! boop`
	`Run of 0 spins after 5 tricks.`
	No spins
		0

`gumballTricks(r, 2);`	`spin bang!`
	`Run of 1 spins after 2 tricks.`
	1 spin
		1

`gumballTricks(r, 10);`	`bang! spin spin boop spin spin spin spin bang! boop` `Run of 4 spins after 10 tricks.`	4

`gumballTricks(r, 3);`	`bang! bang! boop`
	`Run of 0 spins after 3 tricks.`
	No spins
		0

`gumballTricks(r, 1);`	`bang!`
	`Run of 0 spins after 1 tricks.`
	Still prints out "tricks"
		0

Note that if there is only one trick, your final line should end with "1 tricks" (not 1 trick).

Do not check if there is only 1 trick

You must exactly reproduce the format of the console output shown above, though the actual output may differ due to randomness. It is okay for the line of tricks to end with a space. You may assume that the integer passed as a parameter to your method is greater than 0.

You may not construct any extra data structures (e.g. arrays, `ArrayLists`) to solve this problem.

We only need to use some counter variables to solve this problem.

See more ↓

In my method header, I know I'll take in a Random, and an int, as seen in the description and the example method calls. Additionally, I can see that I should return an int, which represents the max number of spins. - public static int gumballTricks(Random r, int n)

Given my <u>int</u> parameter, this represents the total number of tricks that gumball will perform. Therefore, I want to construct a <u>for-loop</u> that will repeat for the given number of times. - for(int i=0; i<n; i++)

Now thinking about how I will keep track of the number of "spins" in a row, I will declare a variable <u>before</u> the for loop to count this value for each iteration of the loop. - int spinsInRow = 0

However, I will need to create an additional variable that will keep track of the <u>greatest</u> number of spins in a row. - int maxSpinsInRow = 0

Moving into the for-loop, next I need to consider how to make each trick equally likely. This can be represented by a random number generation with a range of 0 (inclusive) to 3 (exclusive) since each trick can be represented by a different value in the range. - int trick = r.nextInt(3)

Next, I will need to construct a conditional structure to check whether the trick was "spin" or not. I can arbitrarily choose "0" to represent "spin," "1" for "bang!" and "2" for "boop," but these must stay <u>consistent</u> for all iterations of the loop.

<u>If</u> the value returned by my random .nextInt(3) call is 0, I will print out "spin!" Also, I will update my "spin" counter by 1, and also check if it is greater than the max value so far.

1 2

<u>Else</u>, I will reset the counter back to 0, and print out <u>either</u> "bang!" or "boop."

Lastly, after the for loop, I'll <u>print out</u> the max # of spins, and then <u>return it</u>

6. Array Programming

notice how we are returning a new array: this means we should be building it here in our method! ↳ not reference semantics

Write a static method named **weave** that accepts two arrays of integers as a parameter and that returns a new array that is the result of alternating the values from the two arrays, starting with the first value of the first array. For example, if variables named a1 and a2 store the following values:

0 1 2

```
int[] a1 = {1, 2, 3};
int[] a2 = {4, 5, 6};
```

our parameters

then the call of `weave(a1, a2)` would return a new array containing the following values:

0 0 1 1 2 2

new array: [1, 4, 2, 5, 3, 6]

0 1 2 3 4 5

notice how this new array weaves a1 and a2, starting with index 0 of a1, then index 0 of a2, and traverses to the right through all elements in both arrays

It is possible that the two arrays may have different lengths, in which case after running out of values from the shorter array, the remaining slots of the result array are filled with the leftover elements of the longer array. For example, if variables named a1 and a2 store the following values:

main edge case: ↗aka something to consider

arrays have different lengths

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

then the call of `weave(a1, a2)` would return a new array containing the following values:

weave | attach

[1, 7, 2, 8, 3, 9, | 4, 5, 6]

notice how we first weave a1 and a2 (until a2 runs out of elements), then attach the excess elements at the end (here, a1.length > a2.length)

You are not permitted to create any additional data structures (e.g. arrays, `ArrayLists`, `Strings`) other than the result array that you return.

only create one new array!

My Annotated Solution:

① write method header: parameters - int[] a1, int[] a2 ; return: int[]

```
public static int[] weave (int[] a1, int[] a2) {
    // since we are returning an int[], create a new int[]
    // this int[] will contain all elements from a1 and a2, so its length should be a1.length+a2.length
    int[] result = new int[a1.length + a2.length];
```

② weave: we should first determine which array (a1 or a2) has the shortest length;
we will weave until whichever array first runs out of elements

```
a1: [1, 2, 3]     a2: [4, 5, 6]
     0  1  2           0  1  2
         0  0  1  1  2  2
result: [1, 4, 2, 5, 3, 6]
         0  1  2  3  4  5
```

notice how a1[0] is mapped to result[0]
a2[0] is mapped to result[1]
a1[1] is mapped to result[2]
a2[1] is mapped to result[3]
⋮

=> index i of a1 is mapped to index 2i of result (if i=1, result index = 1·2 = 2)
index i of a2 is mapped to index 2i+1 of result (if i=1, result index = 1·2 + 1 = 3)

```
    for (int i=0; i < Math.min(a1.length, a2.length); i++) {
        result[2i] = a1[i];
        result[2i+1] = a2[i];
    }
```

③ attach excess elements of one array after the weave in result

```
a1: [1, 2, 3 | 4, 5, 6]     a2: [7, 8, 9]
     0  1  2 | 3  4  5           0  1  2
         0  0  1  1  2  2 | 3  4  5
result: [1, 7, 2, 8, 3, 9 | 4, 5, 6]
         0  1  2  3  4  5 | 6  7  8
```

* we need to determine whichever array is longer so that we can attach those elements to the end of weave in result

↳ notice how our starting index to attach is two times the minimum array length (3·2 = 6). to iterate through the result, we would start from this index until result.length exclusive.

↳ notice how the difference between index i of result and the corresponding index of the array with excess elements is the minimum array length; we can use "i - Math.min(a1.length, a2.length)" to retrieve the element in the longer array

```
    for (int i = 2·Math.min(a1.length, a2.length); i < result.length; i++) {
        if (a1.length > a2.length) {
            result[i] = a1[i - Math.min(a1.length, a2.length)];
        } else {
            result[i] = a2[i - Math.min(a1.length, a2.length)];

        }
    }
    return result; // return our int[] result
}
```

Write your solution to problem #6 here:

```
public static int[] weave(int[] a1, int[] a2) {
    int[] a3 = new int[a1.length + a2.length];
    int shorterLength = Math.min(a1.length, a2.length);
    for (int i = 0; i < shorterLength * 2; i++) {
        if (i % 2 == 0) {
            a3[i] = a1[i / 2];
        } else {
            a3[i] = a2[i / 2];
        }
    }
    if (a1.length > a2.length) {
        for (int i = shorterLength; i < a1.length; i++) {
            a3[i + shorterLength] = a1[i];
        }
    } else {
        for (int i = shorterLength; i < a2.length; i++) {
            a3[i + shorterLength] = a2[i];
        }
    }
    return a3;
}
```