Question 1:
1. Converting to and evaluating mathematical expressions in the Reverse Polish Notation (RPN):

Thought Process:

INFIX ⟶ RPN
Process: Shunting Yard Algorithm

Key Words:
Infix notation: A + B
Postfix notation: A B +

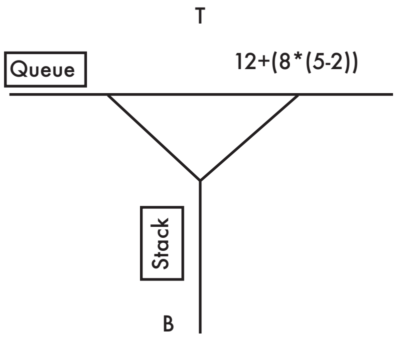
Shunting Yard Algorithm

12 + (8 * (5 - 2)) =

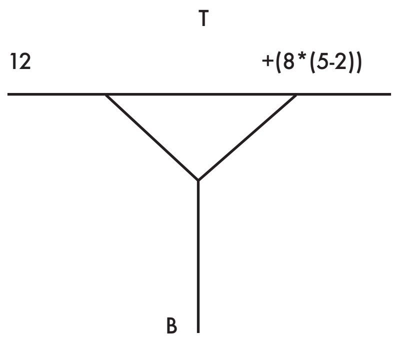
Written in infix expression

Reverse Polish Notation (also known as **postfix** notation) is a system of formula notation without brackets or special punctuation, frequenly used to represent the order in which arithmetical operators are performed in computers and calculators. We use the stack in the Shunting Yard Algorithm (developed by **Edsger Dijkstra**) to convert an **infix** sum (where operators are written in-between their operands) to Reverse Polish Notation (where operators are written after their operands). Evaluating mathematical expressions expressed in the reverse polish notation involves pushing numeric values onto the stack and performing computations to recieve the answer of a sum.
An example of this is shown in the following:

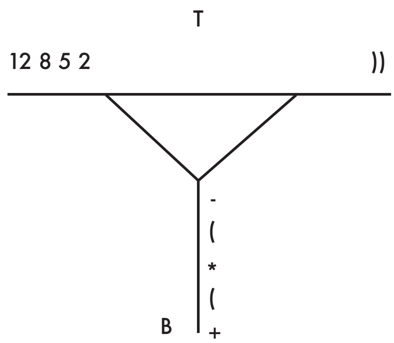
Bottom = B
Top = T



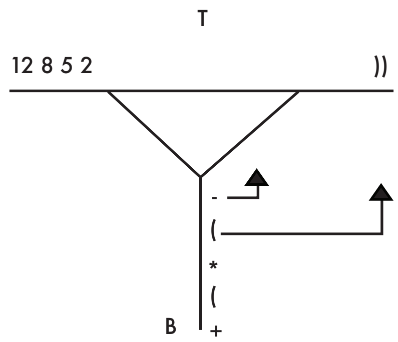
If it is a number slide it across. Read a token. If number add it to queue.



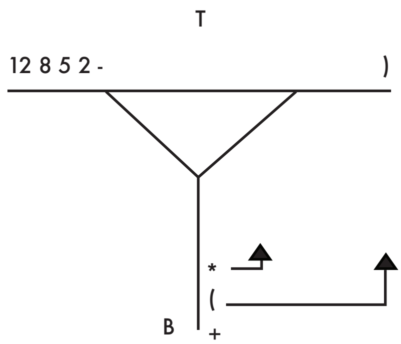
Operators and left parenthasis always go to the bottom.



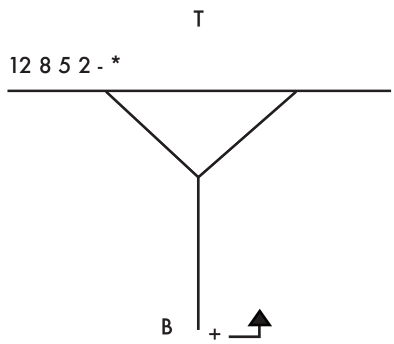
This is what you should end up with until you get to the two left parenthesis))



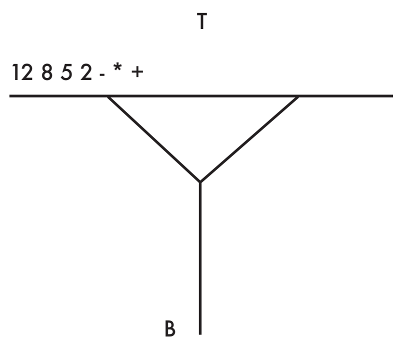
((left parenthesis cancels out)) right parenthesis. We pop - operator out and slide it across to the numbers queue.



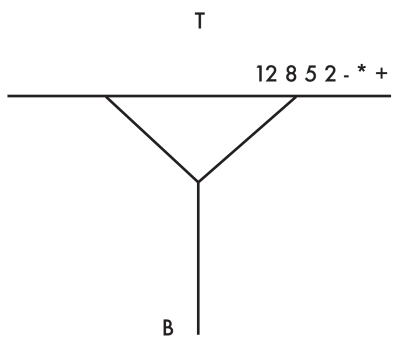
Next we do the same with the * and).



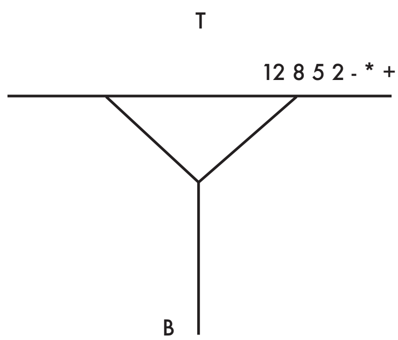
Lastly, we pop out the + sign and slide it across to the numbers queue.



We slide our answer across to the right side.



This is the Reverse Polish Notation (RPN)of the above infix expression.



To solve the sum in RPN, we use a call stack. We will see this on the next page.

Question 1:
1. Converting to and evaluating mathematical expressions in the Reverse Polish Notation (RPN):

Thought Process:

RPN $\longrightarrow$ Call Stack

Key Words: Pop & Push

Evaluate RPN using Call Stack

Call Stack

12 8 5 2 - * +

Call stack is a data structure that stores data. It has a push() and pop() function. Push() pushes numbers onto the stack. Pop() pops numbers off the stack when it needs to perform an operation. Then the number is pushed back onto the stack. This allows us to evaluate RPN using a call stack. This can be seen in the following example:

Variables allocated on the stack are stored directly to the memory and access to this memory is very fast, and it's allocation is dealt with when the program is compiled. When a function or a method calls another function which in turns calls another function etc., the execution of all those functions remains suspended until the very last function returns its value. The stack is always reserved in a LIFO order, the most recently reserved block is always the next block to be freed. This makes it really simple to keep track of the stack, freeing a block from the stack is nothing more than adjusting one pointer.

T						B

B = Bottom of stack
T = Top of stack

T		2	5	8	12	B

Push number onto stack

T		2	5	8	12	B

Pop first two numbers out of stack and use operator on them: 5 - 2 = 3. Then push answer back into stack. * Note importance of order

T			3	8	12	B

Pop first two numbers out of top of stack and use operator on them: 8 * 3 = 24. Push answer back into stack.

T				24	12	B

Pop first two numbers out of top of stack and use operator on them: 12 + 24. Push answer back into stack.

T					36	B

The number 36 is your final answer.

Recursion is a function that calls itself. Every recursive function has two cases: a **base case (n ==0)** and the **recursive case (n * factorial (n-1)**. A stack has two operations: push and pop. All function calls go onto the call stack. The call stack can get very large, which takes up a lot of memory.

We will calculate the factorial of 3! (3 * 2 * 1) using a **Call Stack**. The topmost box (the active frame) in the stack tells you what call to factorial you're currently on. Example:

```
public class factorial {

      public static void main(String[] args) {
            System.out.print(factorial(3));
            }

            public static int factorial(int n) {
                  if (n == 0)
                        return 1;
                  else
                        return (n * factorial(n - 1));
            }
}
```

Call Stack & Recursion

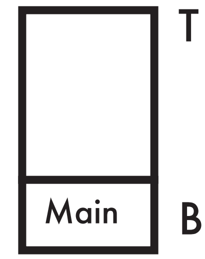
Factorial: 3!

B = Bottom of stack
T = Top of stack

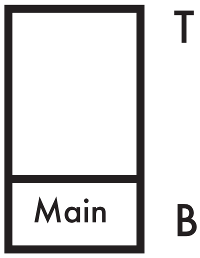
Question 1:
2. As a way of managing the allocation and memory for local variables (the Call Stack)

Key Words: Pop & Push

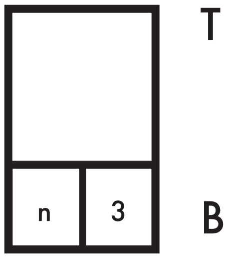
Call Stack



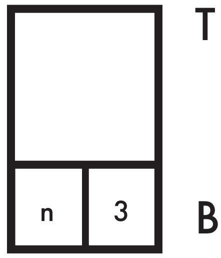
Main got put on the call stack



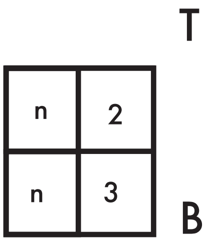
To print out it has to find out what the factorial of 3 is. However it does not know what the factorial of 3 is so it has to work it out.



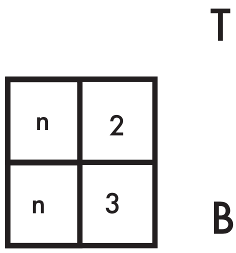
n is not == to 1 so we skip and go down to:
n * factorial (n-1)n is 3.



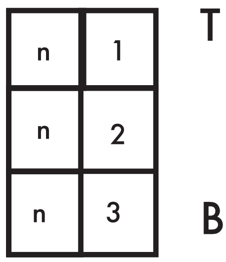
3 * factorial (3-1) = 2
So this will call factorial 2



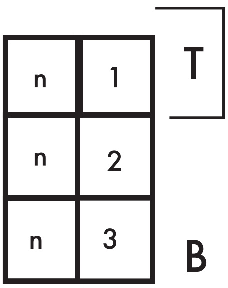
2 goes on the call stack.



2 * factorial (2-1) = 1
So this will call factorial 1

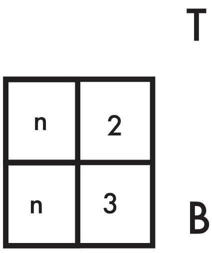


1 goes on the call stack.

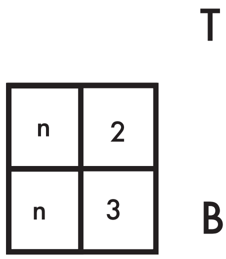


This is the first box to get popped off the stack, which means its the first call we return from. Returns 1.
Note* We made 3 calls to fact but we had not finished a single call until now.

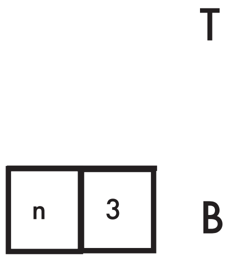
n is now == to 1. So we now return 1.



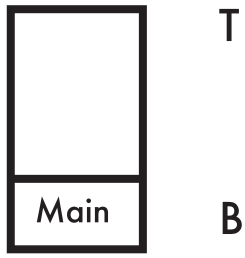
factorial (n - 1) is replaced with 1. n ==2.
return 2 * 1 = 2



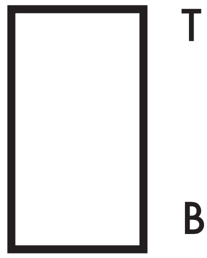
factorial (n - 1) is replaced with 2. n ==3.
return 3 * 2 = 6



factorial (n - 1) is replaced with 2. n ==3.
return 3 * 2 = 6



The only thing left in the stack is main. So all it has to do is print out 6.



Leaving an empty call stack.

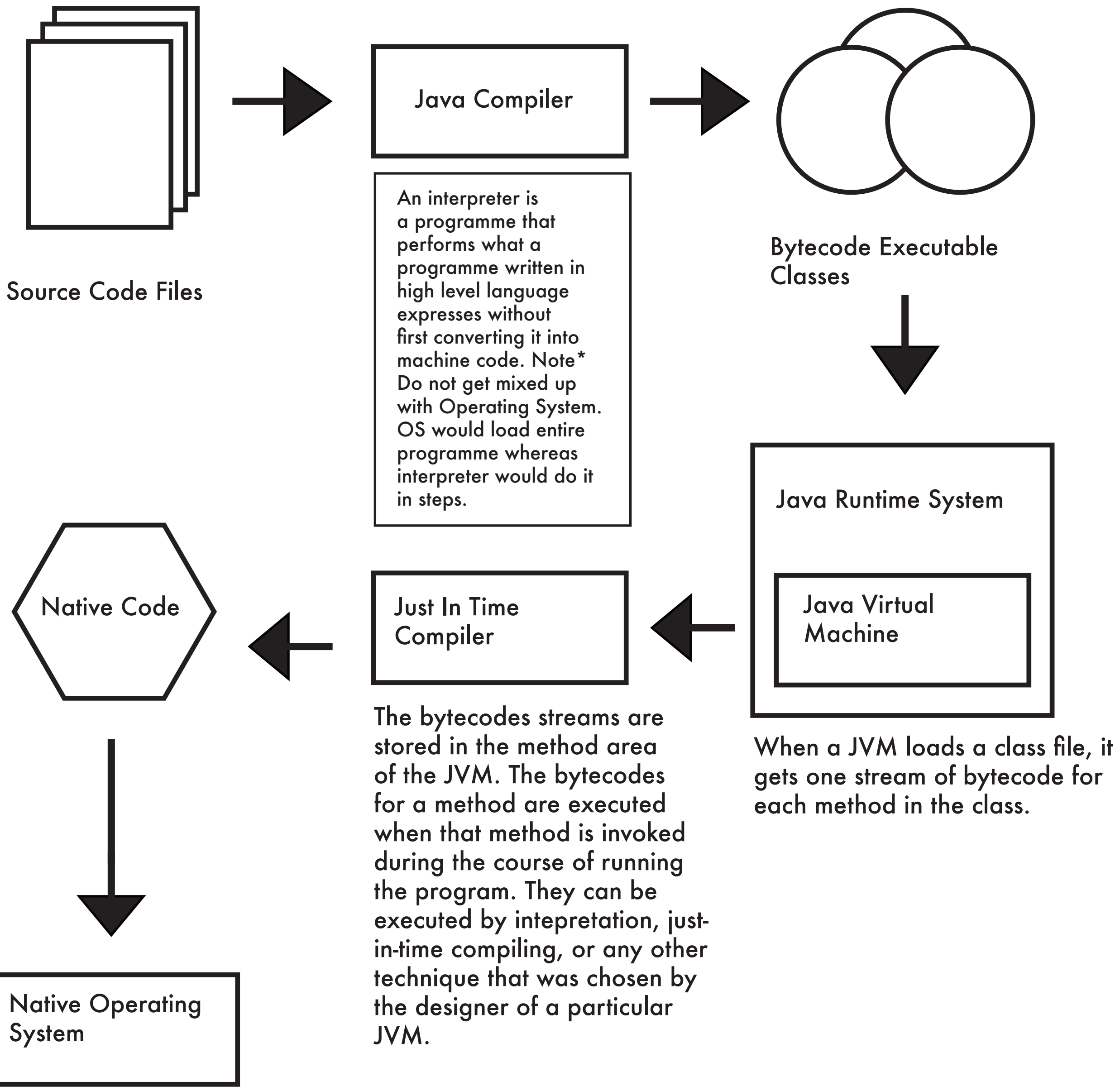
Question 1:
3. For evaluating bytecode as used by the JVM

Thought Process:

Key Words: Compiler, JVM, JIT, platform dependency, jail

JVM

Bytecode is a highly optimized set of instructions designed to be executed by the **Java run-time** system. Its called Byte Code because each instruction is 1-2 bytes. **Source code** is first written in plain text filed ending with the .java extension. After the compilation is successful, **java compiler** will generate an intermediate ".class" file that contains the bytecode. Java is a **portable-language** because without any modification we can use Java byte-code in any platform(which supports Java). So this byte-code is portable and we can use in any other major platforms. **Java Virtual Machine (JVM)** is an extra layer that translates Byte Code into **Machine Code**.
Java Compiler Source Code -> [javac] Byte Code -> JVM Byte Code -> JIT Machine Code
Though it looks like an overhead (overhead is any combination of excess or indirect computation time, memory, bandwidth, or other resources that are required to perform a specific task) but this additional translation allows Java to run applications on all platforms as JVM provides the translation to the Machine code as per the underlying **Operating System**. When we compile a Java Class, it transforms it in the form of bytecode that is platform and machine independent compiled program and store it as a .class file. After that when we try to use a Class, Java ClassLoader loads that class into memory.



Question 1:
3. For evaluating bytecode as used by the JVM

Factorial java program to find factorial of 3
Prints:
Factorial of 3 is 6

```java
public class factorial {
    static int factorial(int n) {
        if (n == 0)
            return 1;
        else
            return (n * factorial(n - 1));
    }

    public static void main(String args[]) {
        int i, fact = 1;
        int number = 3;
        fact = factorial(number);
        System.out.println("Factorial of " + number + " is " + fact);
    }

}
```

```
Viviennes-MacBook-Pro:src vivienneobrien$ javap -c factorial
Compiled from "factorial.java"
public class factorial {
  public factorial();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  static int factorial(int);
    Code:
       0: iload_0
       1: ifne          6
       4: iconst_1
       5: ireturn
       6: iload_0
       7: iload_0
       8: iconst_1
       9: isub
      10: invokestatic  #2                  // Method factorial:(I)I
      13: imul
      14: ireturn

  public static void main(java.lang.String[]);
    Code:
       0: iconst_1
       1: istore_2
       2: iconst_3
       3: istore_3
       4: iload_3
       5: invokestatic  #2                  // Method factorial:(I)I
       8: istore_2
       9: getstatic     #3                  // Field java/lang/System.out:Ljava/io/PrintStream;
      12: iload_3
      13: iload_2
      14: invokedynamic #4,  0              // InvokeDynamic #0:makeConcatWithConstants:(II)Ljava/lang/String;
      19: invokevirtual #5                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      22: return
}
```

Decompiled factorial.java in command line to recieve decompiled bytecode.

Decompiled factorial method

Decompiled main method

The bytecode instruction set was designed to be compact. All computation in the JVM centers on the stack. Because the JVM has no registers for storing values, everything must be pushed onto the stack before it can be used in a calculation. Bytecode instructions therefore operate primarily on the stack. For example, in the above bytecode sequence to find the factorial of 3.

This is done by first pushing the local variable onto the stack with the iload_0 instruction, then pushing the integers onto the stack. After both integers have been pushed onto the stack, the imul instruction effectively pops the two integers off the stack, multiplies them, and pushes the result back onto the stack. The result is popped off the top of the stack and stored back to the local variable by the istore_0 instruction. The JVM was designed as a stack-based machine rather than a register-based machine to facilitate efficient implementation on register-poor architectures such as the Intel 486.

Question 2:

Written in java:

swapping.java

```java
public class swapping {

    public static void main(String[] agrs) {
        int a = 10;
        int b = 5;
        a = a+b;
        b = a-b;
        a = a-b;
        System.out.println("After swapping a = " +a+ " and b = " +b);
    }
}
```

// The function of this **iJVM** programme is to swap the values of two variables without a third party variable

```
0: bipush    10      // VARIABLE OF VALUE 10
2: istore_1          // STORE IN 1
3: bipush    5       // VARIABLE OF VALUE 5
4: istore_2          // STORE IN 2
5: iload_1           // LOAD VALUE OF 1
6: iload_2           // LOAD VALUE OF 2
7: iadd              // ADD 2 + 1 (5 + 10 = 15)
8: istore_1          // STORE RESULT IN 1
9: iload_1           // LOAD 1
10: iload_2          // LOAD 2
11: isub             // SUBTRACT 1 -2 (15 - 10 = 5)
12: istore_2         // STORE RESULT IN 2
13: iload_1          // LOAD 1
14: iload_2          // LOAD 2
15: isub             // SUBTRACT B - A (15 - 5 = 10)
16: istore_1         // STORE RETURN IN 1
```

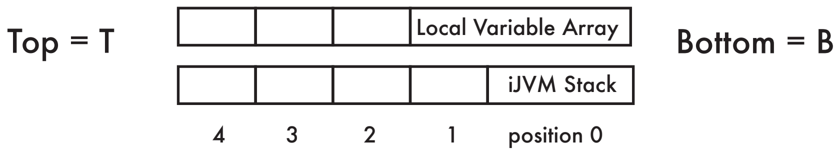
// The function of this LMC programme is to swap the values of two variables without a third party variable

```
LDA VARA        // LOAD VARA OF VALUE 010
ADD VARB        // ADD VARA:10 +VARB: 05 = 15
STA VARA        // STORE RESULT IN VARA (VARA NOW 015)
LDA VARA        // LOAD VARA 015
SUB VARB        // SUBTRACT 15 - 5 = 10
STA VARB        // STORE 10 IN VARB
LDA VARA        // LOAD VARB 10
SUB VARB        // SUBTRACT 15 - 10 = 5
STA VARA        // STORE RESULT IN VARA
END HLT         // HALT the programme
VARA DAT 010    // Variable A
VARB DAT 005    // Variable B
```

Question 3 Part 1:
LMC Table:
Values of relevant memory locations and registers at each step of the program run.

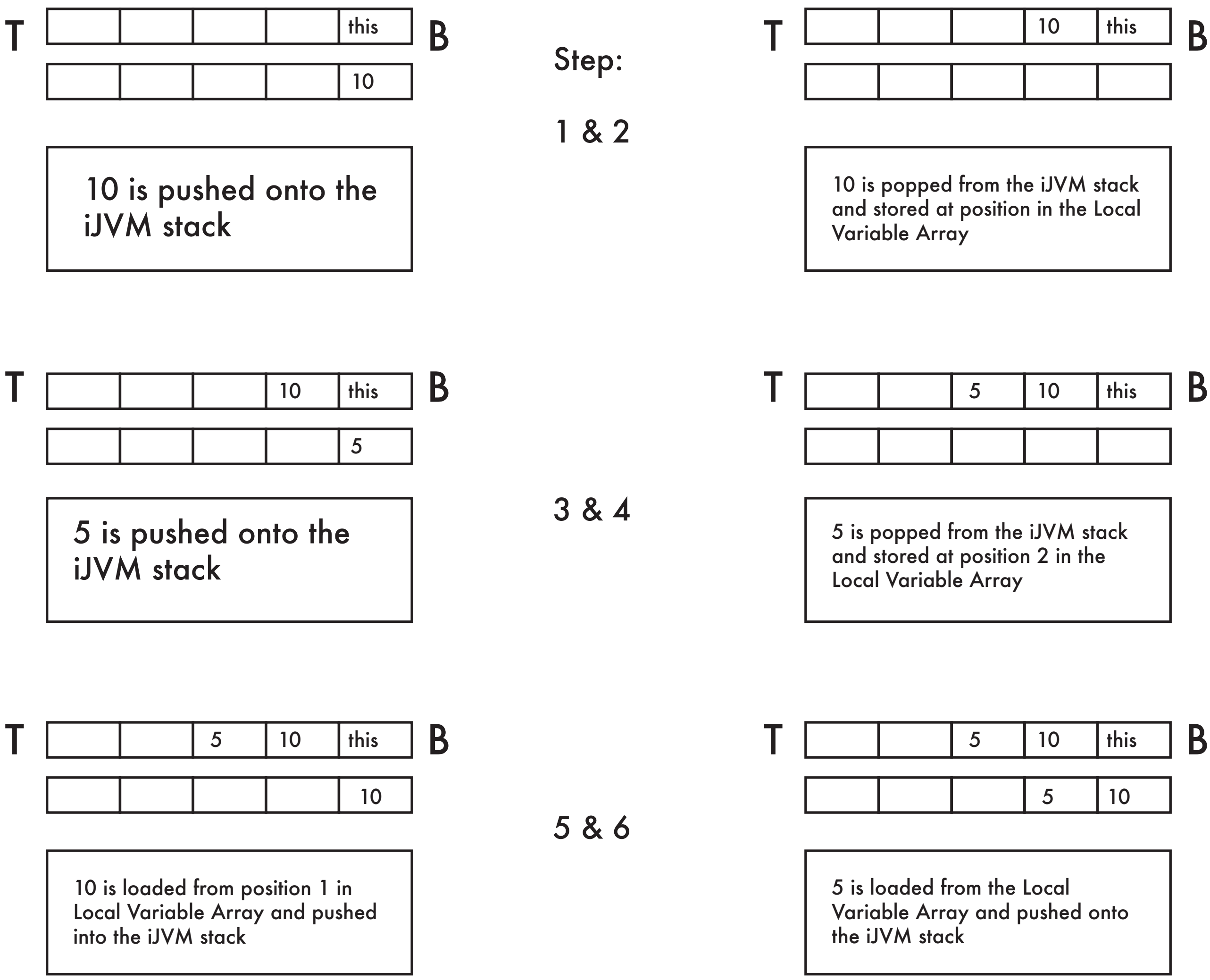
	PC	ACC	VARA	VARB
0	1	10	10	05
1	2	15	10	05
2	3	15	15	05
3	4	15	15	05
4	5	10	15	05
5	6	10	15	10
6	7	10	15	10
7	8	05	05	10
8	9	10	05	10

Question 3 Part 2:
iJVM Stack for iJVM code.



Pushing Local Variables onto the stack:
Local variables are stored in a special section of the stack frame. The stack frame is the portion of the stack being used by the currently executing method. Each stack frame consists of three sections – the local variables, the execution environment, and the operand stack. Pushing a local variable onto the stack actually involves moving a value from the local variables section of the stack frame to the operand section. The operand section of the currently executing method is always the top of the stack, so pushing a value onto the operand section of the current stack frame is the same as pushing a value onto the top of the stack.

Popping to Local Variables:
For each opcode that pushes a local variable onto the stack there exists a corresponding opcode (a machine language instruction that specifies what operation is to be performed by the central processing unit (CPU) that pops the top of the stack back into the local variable. The names of these opcodes can be formed by replacing “load” in the names of the push opcodes with “store”. The opcodes that pop ints and floats from the top of the operand stack to a local variable are shown in the following example:



			Local Variable Array

				iJVM Stack

7 & 8

T			5	10	this	B
					15	

10 and 5 are popped from iJVM stack and are added together then they are pushed back onto the iJVM stack

T			5	15	this	B

15 is popped from the iJVM stack and stored at position 1 in the Local Variable Array

9 & 10

T			5	15	this	B
					15	

15 is loaded from position 1 in active frame and pushed back onto iJVM stack

T			5	15	this	B
				5	15	

5 is loaded from position 2 in Local Variable Array and pushed onto iJVM stack

11 & 12

T			5	15	this	B
					10	

15 and 5 are popped from iJVM stack and are subtracted then they are pushed back onto the iJVM stack

T			10	15	this	B

10 is popped from iJVM stack and pushed onto Local Variable Array at position 2.

13 & 14

T			10	15	this	B
					15	

15 is loaded from Local Variable Array position 1 into iJVM stack

T			10	15	this	B
				10	15	

10 is loaded from Local Variable Array position 1 into iJVM stack

15 & 16

T			10	15	this	B
					5	

10 and 5 are popped from iJVM stack and are subtracted then they are pushed back onto the iJVM stack

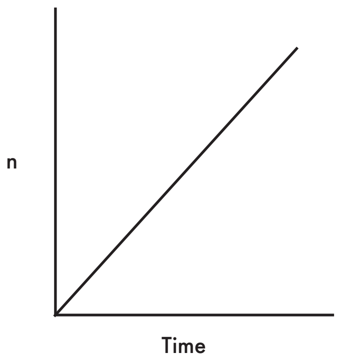
T			5	10	this	B

5 is popped from iJVM stack and pushed onto Local Variable Array at position 2.

Question 4

Question 4a.

```
for i = 0; i < n; i++
    print i ;
```

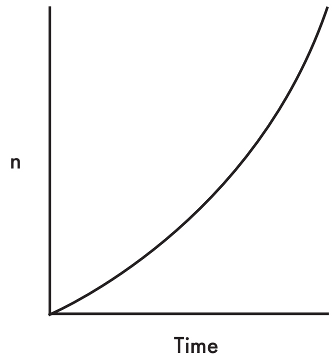


Linear Graph

Reasoning: The complexity is O(n). It is a single loop that takes the length of n. As n increases the amount of operations are performed which increase that the rate of n. This is shown in the linear graph above.

Question 4b.

```
for i = 0; i < n; i++
    for j = 0; j < 3n; j++
        print i, j
```

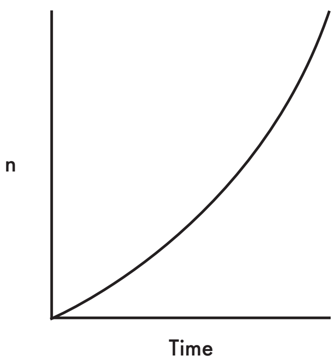


N squared Graph

Reasoning: The complexity is $O(n^2)$. It is a nested loop. The inner loop goes n times for every outer loop iteration which is also n. As n increases the amount of operations needed is squared.

Question 4c.

```
for i = 0; i < n; i++
    print i ;
for i = 0; i < n; i++
    for j = 0; j < 3n; j++
        print i, j
```



N squared Graph

Reasoning: $O(n) + O(n^2)$: The first loop has a complexity of $O(n)$ and the second loop is nested and has a complexity of $O(n^2)$. $O(n) + O(n^2)$ is said to be $O(n^2)$ because larger loop takes priority.