**Data Structures and Algorithms**

**ASSIGNMENT 1**

**Due Date**: Announced on Moodle

**NOTE:** Late submissions will not be accepted

**TO SUBMIT:** Submission will be on Moodle

**PROBLEM 1**

Programming languages must evaluate expressions following the syntax of the language. In this assignment, you will write a tool that will evaluate simple mathematical expressions given in different forms like 8 + 7 \* 5 (infix) and 8 7 5 \* + (postfix).

 You are used to evaluating mathematical expressions that are written using *infix notation*, that is, each operator appears between the operands (numbers), such as

4 \* 5

or

6 \* ( 10 - 7 ).

However, that is not the only way to write expressions. In *postfix notation*, the operator is written *after* the operands. The previous examples would be written as

4 5 \*

and

6 10 7 - \*,

respectively.

Postfix notation has two advantages over infix notation. First, the order of operations is unambiguous without requiring parentheses. As we will see, 3 + ( 4 \* 5 ) can be written as 3 4 5 \* + , while ( 3 + 4 ) \* 5 can be written as 3 4 + 5 \*. Second, computers can evaluate postfix expressions easily using a **stack**.

In the very first example above, you can save the *operands*(4 and 5) on a stack and when you finally read an operator, just pop two operands and apply the operator. The second example works similarly - push the three operands (6, 10, and 7). When you read the - sign, pop the top two operands (7 and 10), apply the -, then push the answer back. When you read the \* sign, pop the top two (3 and 6) and apply the \* and push the answer. When you reach the end of the input, pop the answer. Your stack should now be empty - if not, the expression was malformed (too many operands). You can also detect if there were too many operators - can you see how?

Of course, calculators and computers don't require you to enter your expressions in postfix - they do it for you. Surprisingly, there is an algorithm to convert infix expressions to postfix that also uses a **stack**. Hint: in this postfix to infix conversion algorithm, the stack holds operators, not operands. If you try a couple examples, you'll see that this makes sense, since the order of the operands doesn't change, but the operators must somehow be stored and moved later in the output string.

So to evaluate infix, just convert it to postfix, then apply the algorithm described above!

Your starting code includes an abstract class called Evaluator, that has an abstract method to evaluate strings and return an int. You will write PostfixEvaluator and InfixEvaluator **subclasses** in which you will override this method. This is not the only way to do this, but is good extensible, object-oriented design.

We want you to use the infixToPostfix conversion algorithm as the first step of evaluating infix expressions, so the tests include specific tests to isolate that method. Note that this method takes a String argument and returns a String.

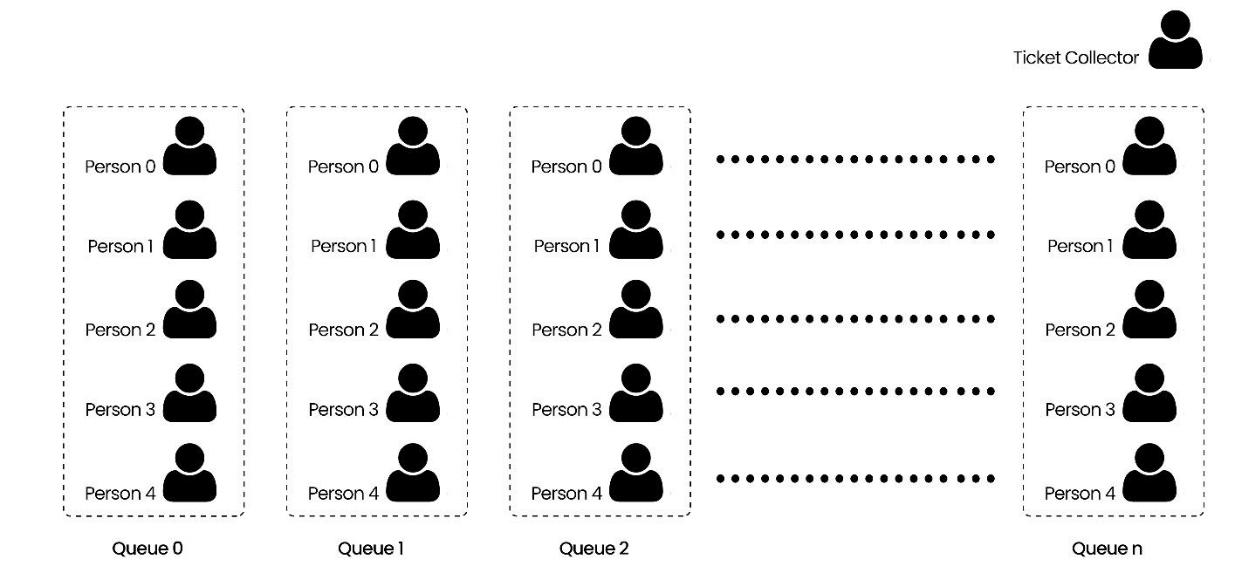
### Requirements for the Evaluator

1. Your algorithm must work in linear time with respect to the number of tokens (operators and operands). Thus, you may only read through the String once and build up the output string once using a loop - no backtracking or inserting output into arbitrary elements in the input string. (Exception: you may use two passes to evaluate infix expressions, one to convert to postfix and one to evaluate the postfix, in keeping with my statement above
2. Both algorithms must use a stack.
3. Neither algorithm can use recursion - recursion does use a stack internally, but we want the use of the stack to be explicit here.
4. Both algorithms must handle the four basic operators (+, -, \*, and /), plus exponents (^).
5. For full credit, you must be able to handle parentheses in infix expressions.
6. For full credit, you must throw ArithmeticExceptions when the input is malformed, as demonstrated in the tests.
7. All math will be integer math, thus you will drop remainders during division if they occur.
8. You must follow the standard order of operations precisely. Therefore you may not change the order things are added and you must apply operations of the same precedence from left to right.

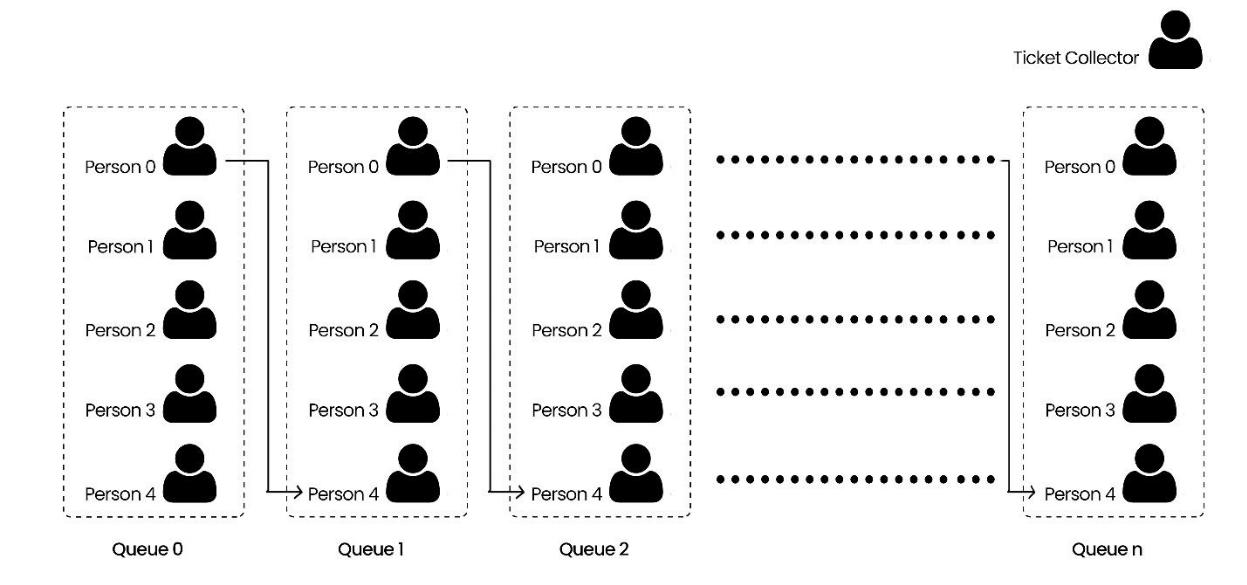
**PROBLEM 2:**

**Movie Ticket Bookings using Queue ADT:**

In a world without digital bookings when people had to buy movie tickets**. Imagine it’s 1990s and** a blockbuster movie has just released. To avoid longer queues and to keep track of queues outside the cinema, they have decided to divide the queue into sub-queues but due to limited resources the cinemas can only afford one ticket collector who is standing at the front of the last sub-queue. The ticket collector takes 2 seconds to process a person. The queues look like this:



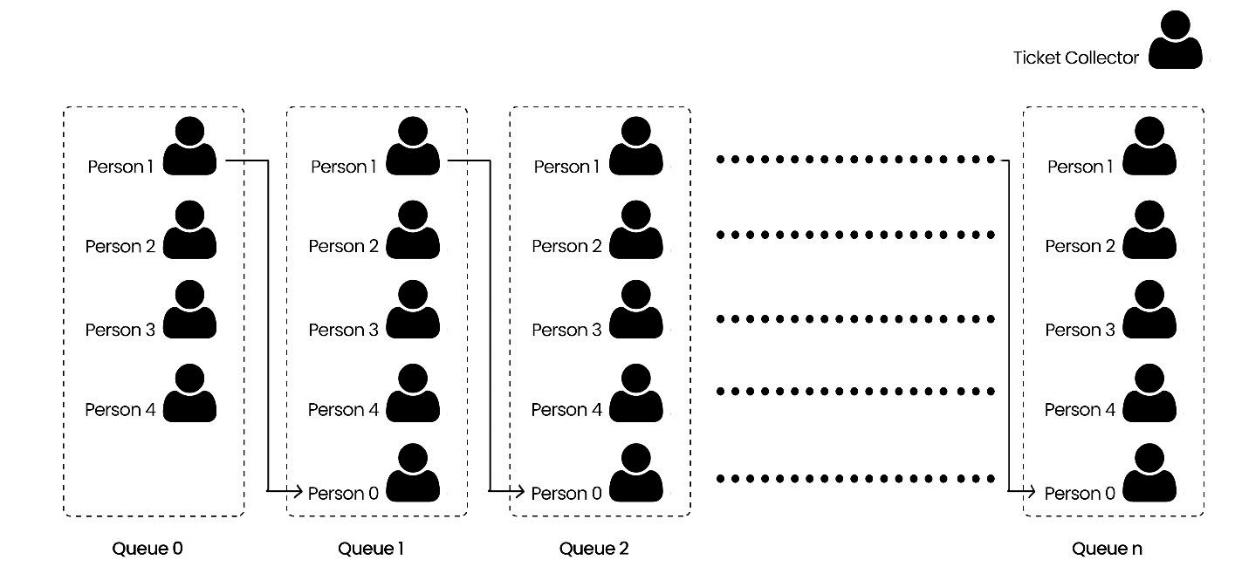
The queues work in the following way:



i.e: After processing **“**Person 0**”** of **“**Queue n**”**, the queue moves forward like this:

1. **“Person 0” of “Queue n**-**1” leaves his/her queue and enters in “Queue n”**.
2. **“Person 0” of “**Queue n-2**” leaves his/her queue and enters in “Queue n**-1**”.**
3. **“Person 0” of “Queue n**-3**” leaves his/her queue and enters in “Queue n**-2**”.**
4. This keeps happening all the way to queue 0 and at the end, **“Person 0” of “Queue** 0**” leaves his/her queue and enters in “Queue** 1**”.**

After the queue is moved one-step forward**, here’s how it looks:**



By the end, the ticket collector processes all the people in the queues so that everyone can enjoy the movie.

You are going to simulate the above explained process using Queues ADT in python. The flow of the program will be like this:

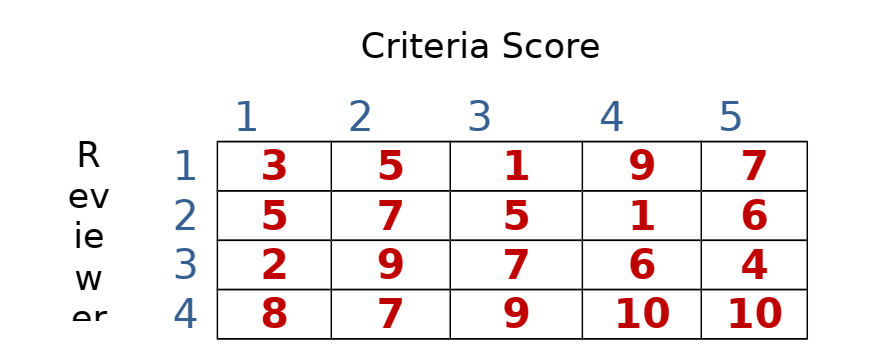
User inputs an integer and N queues (<int>) are created. After creating N queues, you will enqueue some number of persons. For your ease, you can use same number of persons in every queue (minimum number of people in a queue must be 7). The ticket collector starts to process persons in **“**Queue n**”** one by one until all the persons are processed OR all the queues are empty.

**Problem 3:**

Scientific papers submitted to a conference are reviewed by multiple reviewers before an acceptance decision is made. Each reviewer fills in a review form with prespecified criteria (e.g., idea

(1), novelty (2), contribution (3), paper organization (4), and performance evaluation (5)). Each criterion is evaluated out of 10. On average, IEEE requires that each paper to be reviewed by at least four reviewers. Each paper would eventually receive an evaluation score matrix similar to the following:

Criteria Score



The conference chair has set the following rules for paper acceptance:

1- The average of each individual reviewer must be above 5 (i.e., if the average of any reviewer is less than 5, the paper is rejected, regardless of other scores).

2- The overall average of all the reviewers is greater than or equal 7.

He also wanted to consider a round 2 possible acceptance by marking papers as borderline. The paper is “borderline” if the overall score is greater than or equal 6 and less than 7.

We are tasked to write a python program to help the program chair with the review process. The chair will enter the scores of reviewers either from a file or from the standard input (your choice) and then the program return “Rejected”, “accepted”, or “borderline”. You can assume the score matrix is always 4x5.