COMP 2402 Winter 2023

Lab 2 Specifications

Due: Wednesday February 8 3:00pm
Topic focus: Lec 4 - 7
lates accepted within 24 hours
submit on gradescope.ca early and often

Academic Integrity

Please carefully read the following rules regarding academic integrity.

You may:

- Discuss **general approaches** with course staff and/or your classmates, because bouncing ideas off of people is helpful;
- Use code and/or ideas from the **textbook**, because you don't have to memorize every detail of every data structure and algorithm;
- Use a search engine / the internet to look up **basic Java syntax**, because syntax is not the focus of this course and in the real world you have access to syntax,
- Send your code / screenshare your code with **course staff**, because if we have time (and we don't always have time to look at individuals' code) this is one way we can help.

You may **not**:

- Send or otherwise share code or code snippets with classmates, because looking at a someone else's code only requires a superficial understanding, whereas we want you to engage deeply with the concepts in a way you can only do by experiencing the code;
- Write code with a peer (collaborating on high-level algorithms ok; on code, not ok), because the line here is so blurry that it is just too easy to overstep it and overshare;
- Use code not written by you, unless it is code from the textbook (and you should cite it in comments), because this is plagiarism and doesn't involve learning the course material;
- Use AI programmers such as chatGPT or copilot for anything related to this course, because in order to learn the concepts you need to experience coding with the concepts;
- Use a search engine / the internet to acquire approaches to, source code of, or videos relating to the lab, because then you aren't getting practice with problem solving which will put you at a disadvantage when the internet does not have the answers you seek;
- Use code from previous iterations of the course, unless it was solely written by you, because we want you to solve these new problems fresh to really engage with the material.

Note that contract cheating sites are known, unauthorized, and regularly monitored. Some of these services employ misleading advertising practices and have a high risk of blackmail and extortion. Automated tools for detecting plagiarism will be employed in this course.

If you ever have questions about what is or is not allowable regarding academic integrity, **please do not hesitate to reach out to course staff**. We are happy to answer. Sometimes it is difficult to determine the exact line, but if you cross it the punishment is severe and out of our hands. Academic integrity is upheld in this course to the best of Prof Alexa's abilities, as it protects the students that put in the effort to work on the course assessments within the allowable parameters.

Every student should be familiar with the Carleton University student academic integrity policy. Any violation of these rules is a very serious offence and will be treated as such; they are reported to the Dean of Academic Integrity, who launches an investigation. A student found in violation of academic integrity standards may be awarded penalties which range from a reprimand to receiving a grade of F in the course or even being expelled from the program or University. Examples of punishable offences include: plagiarism and unauthorized co-operation or collaboration. Information on this policy may be found here.

Plagiarism. As defined by Senate, "plagiarism is presenting, whether intentional or not, the ideas, expression of ideas or work of others as one's own". Such reported offences will be reviewed by the office of the Dean of Science. Standard penalty guidelines can be found here.

Unauthorized Co-operation or Collaboration. Senate policy states that "to ensure fairness and equity in assessment of term work, students shall not co-operate or collaborate in the completion of an academic assignment, in whole or in part, when the instructor has indicated that the assignment is to be completed on an individual basis". Please refer to the course outline statement or the instructor concerning this issue.

ChatGPT and co-pilot AI Programmers.

It is not appropriate or ethical to use ChatGPT or any other tool to cheat on programming assignments. Here are a few reasons why:

- Cheating undermines the educational process and devalues the effort of those who have completed the assignment honestly. It is unfair to other students and can damage the reputation and integrity of the course or program.
- Cheating can have serious consequences, including failure of the course, academic probation, or even expulsion. These consequences can have a lasting impact on your academic and professional career.
- Learning to code is about more than just completing assignments. It is about developing the skills and knowledge to understand and solve complex problems. Cheating does not foster these skills and can actually hinder your ability to learn and grow as a

programmer. In your programming career you will find that there is generally not one perfect answer, but rather a few “okay” answers that have advantages and disadvantages that you have to weigh critically. This course is about learning how to assess the pros and cons of various approaches. You cannot learn how to do this with an AI programmer, and frankly, it is not very good at it.
- There are resources available to help you understand and complete programming assignments honestly. We offer student hours and 24/7 course forums where you can get help from instructors or peers. Utilizing these resources can help you gain a deeper understanding of the material and improve your coding skills.

Copyright

All lab materials (including source code, pre- and post- labs, workshop materials, and solutions videos, among others) are protected by copyright. Prof Alexa is the exclusive owner of copyright and intellectual property of all course materials, including the labs. You may use course materials for your own educational use. You may not reproduce or distribute course materials publicly for commercial purposes, or allow others to, without express written consent.

Coding Environment Setup

If you want suggestions on how to set up your coding environment, please watch this video or read the “Setting Up Your Programming Environment” post on piazza.

You might find Prof Alexa’s 20-minute “Lab 0 walkthrough” video useful, even though it is for Lab 0, if you want guidance for how to download lab zip files, how to run some programs from the command line, how to open a project in IntelliJ, and how to submit to gradescope.ca.

How to Get Help

Please don't be a hero! If you are having trouble with your programming, we have many resources available to help you. While there *is* value in trying to figure out your own problems, the value (and likelihood of success) diminishes as time goes on. It is common to "get stuck in a rut" and we can help you get unstuck. We want you to spend your time learning, and part of that is letting us help you learn how to better get unstuck. Sometimes you can't do that on your own. We're here to help! If you find yourself stuck on a problem for more than an hour, please use one of the following resources to get help, then put it aside until help arrives (this requires not doing things at the last minute, of course.)

Lab 2 Workshop

Two of our TAs will hold a lab-specific workshop at a time TBD (announced on piazza). This workshop is **completely optional**, and there will be a video posted afterwards if you don't want to attend synchronously. The idea of the workshop is to give you a guided, hands-on start with the concepts of the lab. The TAs will ensure that you all understand what the main goals of the particular lab are and the main goals of the particular parts (time permitting). It is not a debugging session, nor is it student hours, but the idea is that time spent on the workshop can possibly save you a lot of time on the lab itself.

Problem Solving & Programming Tips Document

- A document specifically for this course, full of tips to help you complete your labs faster and better! It is a live document and might be beefed up as the course progresses.

Piazza

- Our primary forum where most course communication should happen.
- Good for getting a quick response, since it goes to a wider audience than email.
- Good for viewing questions that other students have. Chances are, if you have the question, so does someone else. Tags (e.g. "lab2" for lab 2) help manage questions.
- Good for general questions that don't divulge your approach (e.g. "Am I allowed to use standard JCF data structures in my solution?" "Does anyone else get this strange "file not found" error when submitting?" etc.) These questions can/should be posted publicly.
- Good for asking specific-to-you questions of the course staff, which you can/should post "privately" to Instructors on piazza. This gets you the benefits of a wider audience yet keeps your personal details private to course staff.
- Use private posts to Prof Alexa on piazza rather than email to ensure it gets to her.

Student Hours

- There are both in-person and online (discord) student hours.
- Good for quick questions that may have some back and forth.
- Good for clarifications.
- Good for “tips and tricks.”
- Not good for debugging sessions. The person holding the student hour may have many people waiting, and as such cannot spend 20 minutes helping you debug, unfortunately. They may give you a push in the right direction, then ask you to go back in line while you work with that push, while they help someone else.
- Please do not join the queue before student hours begin or rejoin the queue immediately after leaving a TA “just in case” you have another question. This is a shared resource; let others have a chance to use it as well. Only join when you have an immediate question.

Discord

- Good for light social interactions and commiseration.
- Discord is an official class forum, and as such you should keep your behaviour “professional.” Be respectful. Jokes are great, just not at the expense of an individual or a specific group. Disrespectful behaviour (intentional or not) will be met with our zero-tolerance policy (removal from all course forums.)
- Discord is not a good place for asking questions, as the course staff will be spending most of their time monitoring piazza. Piazza has a better system for tracking open questions and answers, and so it is more time efficient for the course staff to spend their time there instead of discord. **Public piazza questions are our highest priority** over all other communication.

Grading via Gradescope Autograder

This programming lab will be tested and autograded by a computer program on gradescope.ca. For this to work, there are some important rules you must follow:

- Keep the package structure of the provided zip file. **If you find a "package comp2402w23l2;" directive at the top of a file, leave it there.** If you (or your IDE) removes this, you will fail the autograder tests.
- **Do not add any import statements;** you should be able to complete the lab without any more than what is initially provided. If you (or your IDE) adds more imports, you will fail the autograder tests.
- Do not rename or change the visibility of any methods already present. If a method or class is public, leave it that way.
- Do not change the main(String[]) method of any PartX.java file. These are setup to read command line arguments and/or open input and output files. Don't change this behaviour. Instead, learn how to use command-line arguments to do your own testing, or use the provided python scripts. More details in the Testing section.
- **Submit early and reasonably often.** The autograder compiles and runs your code, and gives you a mark. You can submit as many times as you like; your most recent grade for each part is recorded.
- Your code has to compile in order to run the tests. We will not manually grade your work, so if your code does not compile you will get a 0.
- Do not use the autograder to debug by continuously submitting and adjusting your code based on failed tests. You should debug locally! More on that later.
- Write efficient code. The autograder places a limit on how much time it will spend executing your code, even on inputs with a million lines. For many questions it also places a limit on how much memory your code can use. If you choose and use your data structures correctly, your code should easily execute within the time limit. Choose the wrong data structure, or use it the wrong way, and your code will be too slow or use too much space for the autograder to work (resulting in a failure of that test). Since efficiency is the main goal of this course, this puts emphasis where it is needed.

When you submit your code for a given part, the autograder runs tests on your code. These are bigger and better tests than the small number of tests provided in the "Local Tests" section further down this document. They are obfuscated from you, because you should try to find exhaustive tests of your own code. This can be frustrating because you are still learning to think critically about your own code, to figure out where its weaknesses are. But have patience with yourself and make sure you read the question carefully to understand its edge cases.

Submitting on Gradescope

You will submit your code to each part to a part-specific autograder on gradescope.ca (**not ~~gradescope.com~~**) You will need a (free) account before doing so (see the Syllabus for details.) if you registered before Monday, January 9th then you should already have an email and be enrolled in our course. If you registered late, please read the Late Registration section of the Syllabus for what you should do. If you have issues, please post to piazza to the Instructors (or the class) and we'll help.

You can submit your code as individual files (e.g. Part1.java, Part2.java) or as a zip file that contains the java files in the root directory (not in sub-directories.) Any excess or unexpected files will be removed by the autograder, so please keep your code to the required files.

Warning: Do not wait until the last minute to submit your lab. **You can submit multiple times and your best score for each part is recorded** so there is no downside to submitting early.

The Lab Itself

Purpose & Goals

Generally, the labs in this course are meant to give you the opportunity to practice with the topics of this course in a way that is challenging yet also manageable. At times you may struggle and at others it may seem more straight-forward; just remember to keep trying and practicing, and over time you will improve.

Specifically, Lab 2 aims to encourage students:

- to think about the pros and cons of the ArrayStack, ArrayQueue, ArrayDeque, DualArrayDeque, RootishArrayStack, SLList, DLList, SEList, and SkiplistList implementations of a List, Stack, Queue, or Deque interface by deciding which of these to use, if any, to solve a variety of problems (Interface Practice);
- to only store the data you need in order to solve the problem;
- to deeply understand the implementation of the DLList and SkiplistList data structures, by implementing some new methods in the MyDLList and MySkiplistList classes (Implementation Practice), and
- to construct algorithms and/or use ideas similar to those seen in lecture so far, such as
    - using two stacks to store data (similar to a DualArrayDeque),
    - using nested data structures to store data, and find data within the nested structure (similar to a RootishArrayStack),
    - using nested data structures to store data, moving data around those nested structures (similar to a SEList),

    - using a pointer (e.g. an iterator) to iterate through a linked list to save time (as in an SLList, DLList, and SkiplistList).
- to maintain or adopt good academic and programming habits through the pre-lab, and
- to reflect on your choice of data structures and algorithms through the post-lab,
- to get practice with JCF, that is, Java's implementation of the data structures from lecture, except through the textbook's code (since that best matches up with the methods from lecture—there are some differences). This is not a course about the actual programming, but you use programming to get concrete practice with the concepts from lecture.

Pre-lab [6.7 marks]

Do the pre-lab 2 (multiple-choice questions) on brightspace after reading this specifications document carefully, and optionally watching the lab 2 workshop video once it is available. The pre-lab is **due 7 days before** the lab programming deadline (in this case, by Wednesday February 1st, 3:00pm.) No lates accepted.

The pre-lab questions are meant to encourage good study and programming habits, such as reading the specifications in their entirety, trying the problems out on examples before starting to code, knowing what resources you have available to you if you get stuck, and last but not least, starting early!

Programming Setup

(Prof Alexa made a 20-minute Lab 0 Walkthrough video that may help you do the following steps, even though it is for Lab 0.)

Start by downloading and decompressing the Lab 2 Zip File (Lab2.zip on brightspace under the lab 2 module), which contains a skeleton of the code you need to write. You should have the following files in the comp2402w23l2 directory: Part1.java, Part2.java, Part3.java, Part4.java, Part5.java, Part6.java, MyDLList.java, MySkiplistList.java, MyList.java, and then a bunch of files that you will not edit but might use (ArrayStack, ArrayQueue, ArrayDeque, RootishArrayStack, SLList, DLList, SEList, SkiplistList); in the tests/ directory you should have a lot of .txt files.

The skeleton code in the zip file compiles fine. Here's what it looks like when you unzip, compile, and run Part1 from the command line:

```
Lab2 % pwd
/Users/asharp/2402/Lab2
Lab2 % ls
Lab2.zip
```

```
**Lab2 %** unzip Lab2.zip
Archive:  Lab2.zip
   creating: comp2402w23l2/
  inflating: comp2402w23l2/Part6.java
  inflating: comp2402w23l2/MyList.java
 . . . <lines omitted> . . .
  creating: tests/
  extracting: tests/lab2-p2-1-out.txt
. . . <lines omitted> . . .

  extracting: tests/lab2-p6-0-in.txt
**Lab2 %** ls
Lab2.zip                          student_config_lab2p3.json
comp2402w23l2                     student_config_lab2p4.json
run_local_tests.py                student_config_lab2p5.json
student_config_lab2p1.json        student_config_lab2p6.json
student_config_lab2p2.json        tests
**Lab2 %** ls comp2402w23l2
ArrayDeque.java           MyList.java           Part5.java
ArrayQueue.java           MySkiplistList.java   Part6.java
ArrayStack.java           Part1.java            RootishArrayStack.java
DLList.java               Part2.java            SEList.java
DefaultComparator.java    Part3.java            SLList.java
MyDLList.java             Part4.java            SkiplistList.java
**Lab2 %** javac comp2402w23l2/*.java
**Lab2 %** java comp2402w23l2.Part1
Execution time: 2.52683E-4
**Lab2 %** java comp2402w23l2.Part1 2 tests/lab2-p1-2-in.txt
Execution time: 1.9521E-5
**Lab2 %** python3 run_local_tests.py student_config_lab2p1.json
Compiling comp2402w23l2/Part1.java
Compilation of comp2402w23l2/Part1.java successful.

********************************************************************************

Test 0 - compilation only loaded.
Running test 0 - compilation only
Running java comp2402w23l2/Part1 ./tests/lab2-p1-0-in.txt comp2402w23l2/out.txt
*** TEST FAILED on output at line 0 ***
=== Actual output (not as expected) ===
=== Expected output ===
0

            Your output is not correct; debug it on small examples.
            Test FAILED - 0 - compilation only.
. . . <many lines omitted> . . .
```

You can also run the MyDLList and MySkiplistList programs:

```
Lab2 % java comp2402w23l2.MyDLList
Test copy(2) ------
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
Done Test copy------
Test copy(2) ------
[]
[]
Done Test copy------
Test copy(0) ------
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Done Test copy------
Test copy(1) ------
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
Done Test copy------
Test copy(4) ------
[0, 1, 2, 3]
[0, 1, 2, 3]
Done Test copy------
Lab2 % java comp2402w23l2.MySkiplistList
Test copy(2) ------
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
size: 5
get(0) = 0
get(1) = 1
get(2) = 2
get(3) = 3
get(4) = 4
Done Test copy------
```

```
Test copy(2) ------
[]
[]
size: 0
Done Test copy------
        < more lines omitted >
```

If you are having trouble running these programs, **figure this out first before attempting to program**. Check the lab 2 FAQ on piazza and/or the aforementioned setup video, or ask a question there if you are stuck and course staff or another student will likely help you fairly quickly.

Interface Practice [60 marks]

The 6 parts in this section ask you to choose the best data structure (interface) to solve the following problems efficiently (with regards to both time and space).

You might first write solutions that just "get it done" (that is, are correct); once you have those submitted to the autograder and you are thus confident you understand the problem, only at that time think about how you are using data and whether the data structure you've chosen (if any!) is the best one for the task at hand.

You should not need any data structure other than those provided: (textbook implementations of) ArrayStack, ArrayQueue, ArrayDeque, DualArrayDeque, RootishArrayStack, SLList, DLList, SEList, and SkiplistList.

Do not modify the main(String[]) methods for any PartX.java file, as they are expected to remain the same for the autograder tests.

Sample input and output files for each part are provided in the tests/ directory of the Lab2.zip file. If you find that a particular question is unclear, you can probably clarify it by looking at the sample files for that question. Part of problem solving is figuring out the exact problem you are trying to solve.

You may assume that all lines are integers under 32 bits. If you need some help with modular arithmetic, try this khan academy page. In particular, **to avoid integer overflow**, you might want to use the mathematical fact that (a+b)%X = ((a%X) + (b%X))%X and (a*b)%X = (a%X)*(b%X). Note that Java's version of modulo is slightly different, in that the modulo of a negative number in java is negative. Also note that s += b%X is doing s = s + (b%X) which is not probably what you're looking for. Best to expand += and -= and %= if you're doing mods.

Part 1. [10 marks] Given a file where each line is a single (32-bit) **positive integer**, read in the lines of the file one at a time; let $s_1, s_2, \ldots, s_k$ be all multiple of x^2, and $t_1, t_2, \ldots, t_j$ be all multiples of x that occur (strictly) **after** the last multiple of x^2. Return the sum modulo 240223 of $(s_1*t_1) + (s_2*t_2) + \ldots + (s_m*t_m)$ where m=min{k, j}. For example, if x=2, x^2=4 and the input is

4	← multiple x^2
10	← multiple x but not x^2
5	← neither multiple of x nor x^2
16	← multiple of x^2
9	← neither multiple of x nor x^2
8	← multiple of x^2 ← this is the last multiple of x^2
30	← multiple of x but not x^2
6	← multiple of x but not x^2

Then the output is (4*30) + (16*6) = 216, because the multiples of 4 are 4, 16, 8, and the multiples of 2 after the last multiple of 4 are 30, 6. Therefore k=3, j=2, so m = min{3,2}=2.

Note that you might have integer overflow for large values of s_i*t_i, but try to avoid it as much as possible (use the mod-related math facts highlighted above.)

Since you'll be using interfaces and implementations from our textbook, you won't need to import anything.

For more examples, see the tests in the tests/ directory that have the form lab2-p1-X-in.txt for the input, and the matching lab2-p1-X-out.txt for the expected output; you might look at the student_config_lab2p1.json to look at the values of the parameter x that are used for the given input-output pairs.

Part 2. [10 marks] Given a file where each line is a single (32-bit) integer, read in the lines of the file one at a time; output n sums computed the following way: sum i is the sum modulo 240223 of every vth line starting from the 0th until you've summed (i+1) lines, where v>0 is the value of the last line. That is, first you output the 0th line. Then you output the sum of the 0th and vth line. Then you output the sum of the 0th, v^th, and 2v^th line, etc. All of this is modulo 240223, and you treat the list circularly using modular arithmetic. If the file is empty, output 0. For example, if the input is

```
10
20
30
40
50
4       ← v = 4, n = 6
```

then the output is

```
10
```

60	(=10+50)
90	(=60+30)
100	(=90+10)
150	(=100+50)
180	(=150+30)

because we output 6(=n) sums, and the sum on line i is the sum of the lines obtained from the 0th, v^th, 2v^th, …, (i*v)^th lines.

For more examples, see the tests in the tests/ directory that have the form lab2-p2-X-in.txt for the input, and the matching lab2-p2-X-out.txt for the expected output.

Part 3. [10 marks] Given a file where each line is a single (32-bit) **non-negative integer**, read in the lines of the file one at a time; each line x represents a list command (x%4) and a value (x/4), where the commands are:

x% 4 == 0: add(x/4) to the left of the current cursor
x% 4 == 1: remove the element most recently iterated by next/previous
x% 4 == 2: next (move the cursor right by one) (iterates over an element)
x% 4 == 3: previous (move the cursor left one) (iterates over an element)

This is modeled off of the JCF ListIterator, so it might be helpful to read its interface notes. The commands in your file will always be "valid," that is, you will never try to remove something that doesn't exist, nor will you try to call next or previous when there is no next or previous.

For example, initially we have an empty list with the cursor at the start [^]. Consider input

44 → 44 % 4 = 0, 44 / 4 = 11; this is add(11) → [11^]
48 → 48 % 4 = 0, 48 / 4 = 12; this is add(12) → [11, 12^]
4 → 4 % 4 = 0, 4/4 = 1; this is add(4) → [11, 12, 4^]
3 → 3 % 4 = 3; this is previous → [11, 12^, **4**] (4 iterated)
20 → 20 % 4 = 0, 20/4 = 5; this is add(5) → [11, 12, 5^, 4]
2 → 2 % 4 = 2; this is next → [11, 12, 5, **4**^] (4 iterated)
1 → 1 % 4 = 1; this is remove of the most recently iterated item (4) → [11, 12, 5^]
3 → 3 % 4 = 3; this is previous → [11, 12^ **5**] (5 iterated)
3 → 3 % 4 = 3; this is previous → [11, ^**12**, 5] (12 iterated)
1 → 1 % 4 = 1; this is remove of the most recently iterated item (12) → [11, ^5]

The output is therefore

11
5

For more examples, see the tests in the tests/ directory that have the form lab2-p3-X-in.txt for the input, and the matching lab2-p3-X-out.txt for the expected output.

Note that remove will only ever be called if it is preceded by a call to next or previous (as in the JCF listIterator specifications.) If you get an IllegalStateException, it is likely that your code is accidentally violating this property. (The input is such that this should never happen, if your code is correct.)

Part 4. [10 marks] Given a file where each line is a single (32-bit) non-negative integer, read in the lines of the file one at a time; as with Part 3, each line x represents a list command (x%4) and a value (x/4), where the commands are:

x% 4 == 0: add(x/4) to the left of the current cursor
x% 4 == 1: remove the element most recently iterated by next/previous
x% 4 == 2: move cursor to the right by x/4, wrapping around if necessary (iterate the last one)
x% 4 == 3: move the cursor to the left by x/4, wrapping around if necessary (iterate the last one)

This is very similar to Part 3, except next and previous are replaced by more general "right" and "left" commands, and the remove removes the last element iterated by left and right. If x/4=0 then nothing is iterated.

For example, initially we have an empty list with the cursor at the start [^]. Consider input

44 → 44 % 4 = 0, 44 / 4 = 11; this is add(11) → [11^]
48 → 48 % 4 = 0, 48 / 4 = 12; this is add(12) → [11, 12^]
4 → 4 % 4 = 0, 4/4 = 1; this is add(4) → [11, 12, 4^]
11 → 11 % 4 = 3, 11/4 = 2 ; this is previous(2) → [11^, **12**, 4] (12 iterated)
20 → 20 % 4 = 0, 20/4 = 5; this is add(5) → [11, 5^, 12, 4]
10 → 10 % 4 = 2, 10/4=2; this is next(2) → [11, 5, 12, **4**^] (4 iterated)
1 → 1 % 4 = 1; this is remove of the most recently iterated item (4) → [11, 5, 12^]
3 → 3 % 4 = 3, 3/4=0; this is previous(0) → [11, 5, 12^] (nothing iterated)
15 → 15 % 4 = 3, 15/4=3; this is previous(3) → [^**11**, 5, 12] (11 iterated)
1 → 1 % 4 = 1; this is remove of the most recently iterated item (11) → [^5, 12]
6 → 6 % 4 = 2, 6/4=1; next(1) → [**5**^, 12] (5 iterated)
1 → 1 % 4 = 1; this is remove of the most recently iterated item (5) → [^12]

The output is therefore

12

For more examples, see the tests in the tests/ directory that have the form lab2-p4-X-in.txt for the input, and the matching lab2-p4-X-out.txt for the expected output.

Tip: Note that this is very similar to Part 3; presumably there is a reason for that :-) In particular, their differences will require them to have different solutions. If you've solved Part 3 and you try to use the same approach to Part 4, where will performance suffer? Can that be avoided?

Part 5. [10 marks] Given a file where each line is a single (32-bit) **non-negative** integer, read in the lines of the file one at a time; create a new list out of the original by reading in groups of i lines for i=1, 2, 3, 4, … and reversing the order of each group of lines (but the groups themselves stay in the original order.) From this new list, consider the last group of values: use these as circular indices to access and sum modulo 240223 the elements of the new list. Output this sum. If the file is empty, output 0. For example, if the input is

```
10
20
30
40
50
60
2
3
15
```

then the modified list is

```
10 __
30
20 __
60
50
40 __
15
3
2
```

Thus n=9 and the last group is 15, 3, 2 so we will sum the elements at indices (15%9=6, 3%9=3, and 2%9=2 for a final sum of (15 + 60 + 20)%240223 = 95.

For more examples, see the tests in the tests/ directory that have the form lab2-p5-X-in.txt for the input, and the matching lab2-p5-X-out.txt for the expected output..

Part 6. [10 marks] Given a file where each line is a single (32-bit) **non-negative integer**, read in the lines of the file one at a time into groups of size x, where x is an input to the program. Consider the last such group, which has between 1 and x values (depending on whether n is divisible by x). As with Parts 3 and 4, each line v in the last group represents a command (v%2) and an index (v/2), where the commands are:

v%2 == 0: shift the left-most element of group (v/2) to be the right-most element of group (v/2-1), where we treat the groups as a circular list (i.e. mod the # groups)

v%2 == 1: shift the right-most element of group (v/2) to be the left-most element of group (v/2+1), where we treat the blocks as a circular list (i.e. mod the # groups).

If either of these commands causes a group to become empty, remove it from the list of groups. These commands may ``target” the final group (from which we took our commands); the commands should not be affected (as in: they are fixed before any of the operations occur.) Once all the operations are performed, print out each group followed by "****", but where each group itself is output in reverse order. For example, for x=2 if the input is

```
4
2 ___
1 → 1 % 2 = 1, 1/2=0; shift L from group 0 to group -1 (which is group 1, circularly)
3 → 3 % 2 = 1, 3/2=1; shift L from group 1 to group 0
```

after the first instruction we have the groups

```
2 ___
1
3
4
```

then after the second instruction we have the groups

```
2
1 ___
3
4
```

And then we output these, each in reverse, separated by ****

```
1
2
****
4
3
****
```

If we had the same input but with x=3 we'd have

```
4
2
1 ___
3 → 3 % 2 = 1, 3/2=1; shift L from group 1 to group 0
```

which would lead to groups

```
4
2
1
3 ___
```

and the output of

```
3
1
2
4
****
```

For more examples, see the tests in the tests/ directory that have the form lab2-p6-X-in.txt for the input, and the matching lab2-p6-X-out.txt for the expected output.

Implementation Practice [20 marks]

The 2 parts in this section ask you to implement a single (new) method in each of the MyDLList and MySkiplistList classes, with the goal of getting your hands dirty with the implementations of these classes. Construct algorithms for these methods that are time- and space-efficient, using the underlying implementation that we discussed in lecture.

As with the interface problems, it is good practice to first write solutions that just “get it done” (that is, are correct); once you have those submitted to the autograder and you are thus confident you understand the problem, only at that time think about your algorithm and whether the goal can be accomplished in a more efficient way.

The (provided) MyDLList and MySkiplistList classes are (incomplete) implementations of the (provided) MyList interface. Your task is to complete the implementation of the MyDLList and MySkiplistList classes, as specified below. Default implementations are provided for the 2 methods so that you can compile and run the main methods of the relevant classes right out of the gate; these defaults will not pass any of the tests, however.

Part 7. [10 marks] In the MyDLList class, implement

public void copy(int k)

This replaces each element x with k copies of element x (as separate elements). You may assume that k ≥ 0.

For example,

- If the MyDLList ml originally represents the list [a,b,c] then after ml.copy(3), ml now represents [a, a, a, b, b, b, c, c, c].
- For any k ≥ 0, if ml=[ ] then after ml.copy(k), ms now represents [ ].
- For any MyDLList ml, after ml.copy(0), ms represents [ ].

The idea is to implement this method using the class variables provided. You should not need to use an ArrayStack or any other JCF data structure to solve this problem. You should not change the underlying class variables nor the implementation of the provided methods.

Part 8. [10 marks] In the MySkiplistList class, implement

public void copy(int k)

This replaces each element x with k copies of element x (as separate elements). You may assume that k ≥ 0. This is the same functionality as with the MyDLList, but with a different underlying implementation (an SkiplistList is certainly more complex than a DLList, if you recall from lecture, although it will help you to focus on their similarities.)

The idea is to implement this method using the class variables provided. You should not need to use an ArrayStack, an ArrayDeque, a DLList, or any other data structure to solve this problem. You should not change the underlying class variables nor the implementation of the provided methods.

The examples are the same as with Part 7.

Tip: Skiplists are full of pointers and it can hurt the brain to keep track of them. To make your life simpler, first try solving the problem on just L0 (which is basically an SLList); once you get that working, think about getting it working for L0 and L1, then the whole skiplist. One level at a time. Or, solve the problem using the Skiplist's add(i,x) function and think about where that might be wasting time/space; try to fix that using add as a basis (or maybe even SkiplistSSet add as a basis.) There are many approaches that will work but I promise that all of them will make you manipulate pointers so just take it one step at a time.

There is already a provided toString method and a main method with (very limited) tests; you can and should adjust these as you see fit in order to test your methods as you go along.

Post-Lab [13.3 marks]

Do the post-lab 2 (multiple-choice questions) on brightspace after the programming deadline has passed (and the solutions video is available). The post-lab is due 7 days after the lab programming deadline (in this case, by Wednesday February 15th, 3:00pm.) No lates accepted.

The post-lab questions are meant to encourage retrospection on what you were meant to learn from the programming lab. If you were stuck on a problem, this is an opportunity to look at the solutions and at the lab 2 sample solutions video, to figure out what went wrong for you, and to still learn what you were meant to learn.

Local Tests

For Parts 1-6, you can view some sample input and output files in the tests/ directory. For example, you can run Part1 on lab2-p1-2-in.txt as follows:

```
Lab2 % java comp2402w23l2.Part1 tests/lab2-p1-2-in.txt
. . . < the output here is what your program outputs both to System.out and w> . . .
```

You can compare what your program outputs (the w.println statements, at least) to the expected output, which in this case would be in tests/lab2-p1-2-out.txt.

If you want to compile Part1.java and then run Part1 on all the tests and diff your output with the expected output, I've provided a python3 script for that:

```
Lab2 % python3 run_local_tests.py student_config_lab2p1.json
. . . < this outputs a lot of information about each test and whether it passed/failed> . . .
```

Of course, you have to have python3 installed on your machine in order for this to work. This script is not release-quality, but I hope it holds up for our purposes.

Once you get a sense for how to test your code with these input files, write your own tests that test your program more thoroughly (**the provided tests are not exhaustive!**) You can add them to the student_config_lab2pX.json file, if you wish.

If you find local tests that are illustrative / informative / helpful, feel free to share them on piazza; label the post "Lab 2 Part X local input/output test" or something like that.

Note that these tests are just checking correctness, not performance (as the performance test files are necessarily big). You might pass the local tests but fail performance autograder tests. You might even fail correctness autograder tests, as the local tests are not meant to be exhaustive, but rather try to point out some of the common expected errors, and give you a sense for how to write your own tests. As the course progresses, the local tests will be pared down and you will be expected to test more on your own.

For Parts 7, and 8 (MyDLList and MySkiplistList), the main method of each file provides some tests you can and should add / modify as you go along. There is no python script here, nor any input or output files. You can just construct MyLists as you wish in the main method.

Autograder Tests

See the "Submitting and Autograder Tests" section further up this document.

Tips, Tricks, and FAQs

See the lab 2 FAQ post on piazza (and/or the lab2 filter) for the most up-to-date lab-specific FAQ.

Regarding tips and tricks, please see the Problem Solving & Programming Tips document (which may be enhanced as the semester progresses, so check back frequently!)

Focus on one part at a time. Take a deep breath and take it one step at a time. Start early!

1. First, read the question carefully. Try to understand what it is asking. Use the provided example(s) to help clarify any questions you have.

2. Look at a few of the provided test input/output files (for Parts 1-6) and check your understanding there.
3. Once you think you understand the problem, try to solve the problem by hand on the sample inputs. This helps give you a sense for how you might solve the problem as a base line. The prelab encourages you to do a bit of this, but do more if you need it!
4. Don't try to solve the problem "the best way" right off the bat. First aim for a *correct* algorithm for the problem, even though it might be inefficient. Test your correct algorithm locally to see if it passes those tests. If it doesn't, then maybe you have misunderstood the problem! It's best to ensure you understand the problem before you try to find an "optimized" solution to your mis-interpreted problem. Once you are passing the local tests, submit your correct algorithm to the server to see if it passes the correctness tests there. Rinse and repeat until all local and server *correctness* tests pass.
5. Once you have a correct algorithm that is passing the correctness tests, maybe you lucked out and your first baseline solution passes the efficiency tests. If so, great! But most likely your first baseline solution does not pass all the efficiency tests. That's okay, we were expecting that. Look at the solution you have and think about where and when it is inefficient. What is its current time and space efficiency? Are there particular inputs (e.g. increasing numbers or all the same number or all prime numbers, etc.) for which your algorithm is particularly costly (in time or space)? If so, can you think of a better data structure to improve the operations in these situations? Or can you think of a different algorithm to do better in these situations?
6. You might consider looking at the prelab, which points out the pros/cons of the data structures of this lab. How are these relevant? You might look at the "Purposes & Goals" section of these specifications to see what algorithms and ideas we expect you to practice in this lab; are any of them relevant here?
7. Remember that even if you're not able to get all of the efficiency points, you might still be able to get points on the post-lab provided you are able to engage enough with the material to understand the purpose of the problem. And we are here to help! If you're stuck, please use the course resources to let us help you.

Questions to ask yourself:

- do you always need to look at every line of the file? if not, cut out early when you can to save time.
- do you need to store everything? every other line? only certain lines? if only some, save space by only saving what you need.

Common Errors & Fixes

Error: Could not find or load main class Part1

If you run from the command line and you run from within the comp2402w23l2 directory, you'll be able to compile, but you won't be able to run. You have to go up a directory (outside the comp2402w23l2 directory) to run. You can also compile from there.

My output looks correct but then I fail the tests (either the python script or the autograder)

Is it possible that you are using System.out.println instead of w.println? When you run on the command line, you will see both System.out.println and w.println in your output, but only w.println is written to the files that we test against. This means that if you're testing while printing to the screen, you might see output that appears correct, but you fail the tests. It's possible this is because you're using System.out.println instead of w.println. The autograder checks your output to w, not your output to System.out. You can determine whether you're making this error by testing your file with an output file instead of to the console. Open the file and see what you've output. That's what the autograder sees. It ignores output to System.out (so you can use this for debugging purposes.)

StackOverflow Error

This is usually caused by infinite recursion somewhere.

OutOfMemory Error

This is caused by using too much space. Some questions you might ask yourself:

- Are you storing all lines of the file? If so, do you need to store everything, or is it possible you only need the previous line, or every other line, or all lines divisible by x, etc. If only some, save space by only saving what you need.

Time Limit Reached / Test Time-out

If your test fails because it reached the time limit (usually in the 1-4 second range), this means your solution takes too much time. This means that you are either not using the best data structure for the solution, or you are iterating through the data too many times (or both). Some questions you might ask yourself:

- For the operations you do repeatedly (e.g. add(i, x), remove(i), set(i,x), etc.) what is the runtime of the operation for your choice of data structure. Is there a different data structure that can do this operation significantly faster?

- If you’re running through your data multiple times, do you need to? If you’re running over ALL your data multiple times, is it possible to only run over part of that data? Or, cut out early in some situations?