COMP 2402 Winter 2023

Lab 3 Specifications

Due: Wednesday March 1 3:00pm (or, 11:59pm if you wish)
Topic focus: Lec 8 - 11
lates accepted by Thursday March 2 3:00pm
submit on gradescope.ca early and often

Feedback on Lab 1 Feedback

Hi all, thank you for taking the time to provide feedback after Lab 1; I know you are all very busy! I am reading over your feedback carefully, and making some adjustments to the course in response. You can read more about your feedback and the changes I will and not make (and why) here.

Changes from Lab 1 & Lab 2

In Lab 1 you were allowed to use JCF but not the textbook (ODS) implementations.
In Lab 2 you were allowed to use the textbook (ODS) implementations but not JCF. But, I accidentally provided you too much access to the ODS implementation, in that I put the data structures in the same comp2402w23l2 package, which meant you could access protected data and methods of the data structures (which you shouldn't be able to and shouldn't want to do – let's keep abstraction!)
Thus in Lab 3 the textbook (ODS) implementations are in a separate ods package in a separate ods directory. I've imported the appropriate packages at the top of each Part 1-6 java file so you shouldn't have to do anything differently provided you keep the directory structure as in the zip file.

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

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 3 aims to encourage students:

- to think about situations where a **List would be preferable to a SSet and vice-versa**, by deciding which of these to use, if any, to solve a variety of problems (Interface Practice);
- to think about the pros and cons of the **SkiplistSSet**, **Treap** and **ScapegoatTree** implementations of the SSet interface by deciding which of these to use, if any, to solve a variety of problems (Interface Practice);
- to **revisit some older problems**, slightly modified, to see if the modifications require a new approach (spoiler: they usually do!);
- to **only store the data you need** in order to solve the problem;
- to deeply understand the **implementation of the SkiplistSSet and BinaryTree** data structures, by implementing some new methods in the MySkiplistSSet and MyIntBST classes (Implementation Practice), and
- to construct algorithms and/or use ideas similar to those seen in lecture so far, such as
    - using **tree traversals** (a.k.a. iteration in a certain way) to save time,
    - **building a complete binary tree**,
    - using the **binary search tree property**,
    - **unwinding recursive algorithms to iterative ones** to **avoid prohibitively large call stacks**.
- 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 3 (multiple-choice questions) on brightspace after reading this specifications document carefully, and optionally watching the lab 3 workshop video once it is available. The pre-lab is **due 7 (academic) days before** the lab programming deadline (in this case, by Wednesday February 15th, 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

Start by downloading and decompressing the Lab 3 Zip File (Lab3.zip on brightspace under the lab 3 module), which contains a skeleton of the code you need to write. You should have the following files in the comp2402w23l3 directory: Part1.java, Part2.java, Part3.java, Part4.java, Part5.java, Part6.java, MySkiplistSSet.java, MyIntBST.java, MySSet.java, DefaultComparator.java, SSet.java and then a bunch of files that you will not edit but might use in the ods/ directory (ArrayStack, ArrayQueue, ArrayDeque, RootishArrayStack, SLList, DLList, SEList, SkiplistList, SkiplistSSet, Treap, ScapegoatTree, etc.); in the tests/ directory you should have a lot of .txt files.

Please see previous labs for what it might look like when you unzip the Lab3.zip file, etc.

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

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 Lab3.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))%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] **(Lab 1 Part 1 Redux)** Given a file where each line is a single (32-bit) **positive integer**, read in the lines of the file one at a time; output the sum **modulo 240223** of each distinct value that is a multiple of its previous line. For example, if the input is

```
3          ← first line has no previous line; we don't include it ever
9          ← 9 a multiple of 3, so we add it to our sum
8          ← 8 not a multiple of 9, so we don't include it
240224     ←240224 a multiple of 8, so we add it to our sum
3
9          ← 9 a multiple of 3, but we've already seen 9 so we don't add it
```

Then the output is 10, since (9+240224)%240223=10. Be careful of integer overflow: if x+y > Integer.MAX_VALUE then x+y will (incorrectly) be a negative number. Use the mathematical properties of mod highlighted above to avoid such overflow.

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 lab3-p1-X-in.txt for the input, and the matching lab3-p1-X-out.txt for the expected output;

you might look at the student_config_Lab3p1.json to look at the values of the parameter x that are used for the given input-output pairs.

Part 2. [10 marks] **(Lab 1 Part 3 Redux)** Given a file where each line is a single (32-bit) **positive** integer, read in the lines of the file one at a time; output the lines that are divisible by the previous line, in (increasing) sorted order, with ties removed. For example, if the input is

4
2 ← not divisible by the previous line
8 ← divisible by 2 so 8 will be in output
8 ← divisible by 8 so 8 will be in output
2
4 ← divisible by 2 so 4 will be in output

The output is therefore

4
8

as we have 3 lines selected (4, 8, 8) however we don't want any ties output.

For more examples, see the tests in the tests/ directory that have the form lab3-p2-X-in.txt for the input, and the matching lab3-p2-X-out.txt for the expected output.

Part 3. [10 marks] (Lab 1 Part 4 Redux-ish) Given a file where each line is a single (32-bit) (possibly negative, possibly 0) integer, read in the lines of the file one at a time; each time you see a multiple m of x^2, print out the smallest multiple of x > m seen in the file up to the current line, where x>0 is a parameter to the execute method. For example, if x=2 and the input is

4 ← multiple of x and x^2 – there are no multiples of x > 4 so far, do nothing
6 ← multiple of x but not x^2
5 ← neither multiple of x nor x^2
9 ← neither multiple of x nor x^2
30 ← multiple of x but not x^2
8 ← multiple of x and x^2 – 4, 6, 30 multiples of x and 30 > 8, output 30
8 ← multiple of x and x^2 – 4, 6, 30, 8 multiples of x and 30 > 8, output 30
4 ← multiple of x and x^2 – 4, 6, 30, 8 multiples of x and 6 > 4, output 6
10 ← multiple of x but not x^2

Then the output is

30
30
6

For more examples, see the tests in the tests/ directory that have the form lab3-p3-X-in.txt for the input, and the matching lab3-p3-X-out.txt for the expected output.

Part 4. [10 marks] (Lab 3 Part 2 Redux.) Given a file where each line is a single (32-bit) **positive** integer, read in the lines of the file one at a time; output the lines that are divisible by the previous line, in (increasing) sorted order, ~~with ties removed~~. For example, if the input is

4
2 ← not divisible by the previous line
8 ← divisible by 2 so 8 will be in output
8 ← divisible by 8 so 8 will be in output
2
4 ← divisible by 2 so 4 will be in output

The output is therefore

4
8
8

as we have 3 lines selected (8, 8, 4) and we want them in increasing (sorted) order.

For more examples, see the tests in the tests/ directory that have the form lab3-p4-X-in.txt for the input, and the matching lab3-p4-X-out.txt for the expected output.

Tip: Note that this is very similar to Part 2; presumably there is a reason for that :-) In particular, their differences will require them to have different solutions. If you've solved Part 2 and you try to use the same approach to Part 4, where will performance suffer? Can that be avoided?

Note: For this problem, you are allowed to use the following 3 imports:

java.util.Random;
java.util.Arrays;
java.util.Comparator;

I can think of 3 different approaches, each using one of these. There are quite possibly other approaches that use none of these (I have one that almost works, maybe with more time I could get it to work but I already have 3 working approaches so I'll stop.)

Part 5. [10 marks] (Lab 2 Part 5 Redux, -ish.) 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; imagine a complete balanced binary tree created out of the file by making groups of i lines for i=1, 2, 4, 8, 16, … the nodes at level i-1 of the tree (so line 0 is the (level 0) root node, the next two lines are the left and right child of the root, respectively, and then the next four lines are the

root's grandchildren, and so on.) In this tree, count up the number of nodes that violate the BST property; that is, node v violates the BST property if it has a left child that is greater than v, or a right child that is less than v (we allow ties.) Output this count. If the file is empty, output 0. For example, if the input is

```
8       ←root node, left child 3 (fine), right child 5 (bad!)
3       ←left child 7 (bad!), right child 4 (fine)
5       ←left child 9 (bad!), right child 1 (also bad! but redundant)
7       ← leaf node, no children
4       ← leaf node, no children
9       ← leaf node, no children
1       ← leaf node, no children
```

then the count of "bad" nodes is 3 (for the 8, 3 and 5.)

For more examples, see the tests in the tests/ directory that have the form lab3-p5-X-in.txt for the input, and the matching lab3-p5-X-out.txt for the expected output.

Part 6. 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; imagine creating a complete balanced binary tree based on the indices of the tree, in that the middle line of the file is the root value, the top half of the file is (recursively) the left half of the (balanced) tree, and the bottom half of the file is (recursively) the right half of the (balanced) tree. Your task is to print out this tree in level-order (that is, translate the file into the balanced binary tree described above, then print out the level i nodes for levels i=0, 1, 2,...). For example, if the input is

```
0
1       ← root of left child
2
3       ← root of the tree so that the left subtree is complete
4
5       ← root of right child
6
```

Then the level-order traversal is

```
3
1
5
0
2
4
6
```

You may always assume that we have a complete, full binary tree, that is, that every level of the tree is as full as possible.Note that the values themselves don't change the tree; if our original input had the first line changed from 0 to 1000, the output would be

```
3
```

1
5
1000
2
4
6

For more examples, see the tests in the tests/ directory that have the form lab3-p6-X-in.txt for the input, and the matching lab3-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 MySkiplistSSet and MyIntBST 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) MySkiplistSSet and MyIntBST classes are (incomplete) implementations. Your task is to complete the implementation of the MySkiplistSSet and MyIntBST 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 MySkiplistSSet class, implement

```
public void rebuild()
```

which rebuilds the current skiplist from scratch to resemble the "deterministic", or "fixed," Skiplist from our Introduction to Skiplists lecture. In particular, all L1 "shortcuts" have length 2, all L2 "shortcuts" have length 4, all L3 "shortcuts" have length 8, and so on. Do this rebuild in O(n) time (or, equivalently, L1 contains every other element of L0, L2 contains every other element of L1, L3 contains every other element of L2, and so on.)

For example, if your Skiplist originally looks like this:

```
L7:       |----------------------------22
L6:       |----------------------------22
L5:       |----------------------------22
L4:       |----------------------------22
L3:       |----------------------------22-------28
L2:       |-------6---------14----18----22-------28
L1:       |-------6---------14-16-18-20-22-------28
L0:       |-0-2-4-6-8-10-12-14-16-18-20-22-24-26-28-30
```

Then after you rebuild, it will look like this:

```
L4:       |---------------------------------------30
L3:       |-----------------14----------------------30
L2:       |-------6---------14----------22----------30
L1:       |---2---6---10----14----18----22----26----30
L0:       |-0-2-4-6-8-10-12-14-16-18-20-22-24-26-28-30
```

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

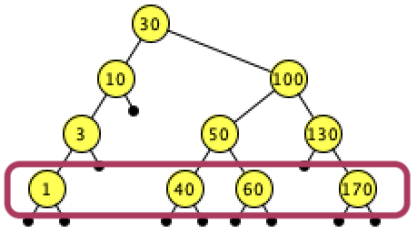
There are some helper methods provided in the class and used in the existing main method. Feel free to use them, but keep in mind that the autograder might depend on a certain implementation.

(There is a toString method that I think is helpful, although not fully tested. You can modify that without breaking the autograder, if you wish. There are also some tests in the main method; you can and should adjust these as you see fit in order to test your methods as you go along.)

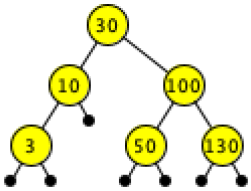
Part 8. [10 marks] In the MyIntBST class, implement

public int trim()

which trims off all the leaves of depth h, where h is the height of the tree and returns the number of nodes trimmed. For example, the single-node tree of height 0 gets its one node trimmed (and becomes empty) and returns 1. The following tree has height 3, and 4 nodes at its deepest depth 3 (1, 40, 60, 170).



The trim method therefore returns 4 (the number of nodes trimmed) and the tree becomes



(There is a print and toString method that may be helpful. You can modify them without breaking the autograder, if you wish. There are also some tests in the main method; you can and should adjust these as you see fit in order to test your methods as you go along.)

Note: the syntax for declaring a node in the MyIntBST class and initializing it to point to the root is this:

```
BinarySearchTree.BSTEndNode<Integer> u = r;
```

Post-Lab [13.3 marks]

Do the post-Lab 3 (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 March 8th, 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 3 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 Lab3-p1-2-in.txt as follows:

Lab3 % java comp2402w23l3.Part1 tests/Lab3-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/Lab3-p1-2-out.txt.

If the Part in question also takes in a parameter x, you'd do somethin glike this:
Lab3 % java comp2402w23l3.Part3 **2** tests/Lab3-p3-2-in.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:

Lab3 % python3 run_local_tests.py student_config_Lab3p1.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_Lab3pX.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 3 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 (MySkiplistSSet and MyIntBST), 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 the appropriate sets as you wish in the main method (see the existing main methods for some sample tests).

Autograder Tests

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

Tips, Tricks, and FAQs

See the Lab 3 FAQ post on piazza (and/or the Lab3 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.

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 3 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. “lab3” for lab 3) 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 comp2402w23l3;" 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.

Common Errors & Fixes

Note: <file>.java uses or overrides a deprecated API.

Note: Recompile with -Xlint:deprecation for details.

These notes occur together sometimes when you are compiling with textbook (ODS) code. You can just ignore them, or you can recompile with the suggested "-Xlint:deprecation" as a flag (after the javac command, before the lab2402w23/PartX.java file.)

Error: Could not find or load main class Part1

If you run from the command line and you run from within the comp2402w23l3 directory, you'll be able to compile, but you won't be able to run. You have to go up a directory (outside the comp2402w23l3 directory) to run. You can also compile from there.

Exception in thread "main" java.lang.NumberFormatException

If you run from the command line and you're running one of the programs that takes in an additional parameter x, you might get this error if you do not specify the parameter x on the command line. To fix this, you need to put an integer (your desired x value) before the name of the input file (equivalently, after the name of the class you are running.) For example:

```
% java comp2402w23l3.Part3 <x-value> <input.txt> <output.txt>
```

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?