# COMP 2402 Winter 2023

## Lab 5 Specifications

Due: Wednesday March 29 3:00pm (or, 11:59pm if you wish)
Topic focus: Lec 1 - 19
lates accepted by March 30 3:00pm
**submit on gradescope.ca early and often**
**(still) marked out of 80**

# Notes about Lab 5

This lab is back to being marked out of 80 points. However, each part you complete over a week early (10/10 by Wednesday March 22 3:00pm) will get 1 extra bonus point added to our bonus points bucket. (See piazza for how bonus is computed; tldr: don't trust brightspace.)

In this lab, you now have all of `java.util` available to you, which gives you access to the Java Collections Frameworks (JCF) (Java's implementation of many of the data structures we've been discussing in this course.)

While you still have access to the ods/textbook implementations, I encourage you to try to use JCF when the equivalent JCF data structure exists. Below is a table that provides a mapping between the ods/textbook implementations and JCF's implementations. The JCF documentation for its implementations are linked to from this table. It is not a bad idea to practice reading the documentation (it's actually not so bad once you try it!)

Some things to note, however:

- there are some ods/textbook implementations that don't have a correspondence in JCF, such as RootishArrayStack, Treaps, Graphs, etc. You can see these in the table.
    - most of these files have main methods that construct and test these data structures, so you can see examples that show how to declare them and how to call their methods there.
- on the flip side, there are some JCF implementations that we didn't explicitly see, such as the LinkedHashSet (a combination of a HashSet with an iterator that isn't completely arbitrary.)
- Java's ArrayDeque is slightly different from the ods ArrayDeque, (java's does not allow you to add or remove from an arbitrary index (only the front and back.)) It does have a remove(o) method, but it removes the first instance of element o, not the element at index o. (You can see this difference in the documentation.)
- In the case that there is a naming conflict (i.e. ods.ArrayDeque vs. java.util.ArrayDeque) you can declare which one you mean using the full path, e.g.
    - ods.ArrayDeque ad = new ods.ArrayDeque()

- java.util.ArrayDeque ad = new java.util.ArrayDeque()

| **interface** | **ods/textbook** | **JCF** |
|---|---|---|
| List | ArrayStack | ArrayList |
| List (ods), Deque (JCF) | ArrayDeque | ArrayDeque (beware its remove(i)!) |
| List | DLList | LinkedList |
| SSet | RedBlackTree | TreeSet |
| SSet | SkiplistSet | ConcurrentSkiplistSet (size is O(n)!) |
| Priority Queue | BinaryHeap | PriorityQueue |
| USet | HashSet | HashSet |
| USet + Iterator | N/A | LinkedHashSet |
| Map | HashMap | HashMap |
| Graph | AdjacencyLists | N/A |
| SSet of Integers | YFastTrie,<br>XFastTrie,<br>BinaryTrie | N/A |
| Iterator (next only) | Iterator | Iterator |
| ListIterator (next & prev) | ListIterator | ListIterator |
| List | RootishArrayStack | N/A |
| List | SkiplistList | N/A |
| SSet | Treap | N/A |
| SSet | ScapegoatTree | N/A |

# 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 5 aims to encourage students:

- to move towards using the Java Collections Framework (JCF) where possible, instead of using the ods/textbook implementations of our data structures
    - see the "Notes about Lab 5" section at the top of the specifications for details on how to make this shift.
    - you also have access to the ods Algorithms class, which contains some of the sorting and graph algorithms we've discussed.
- to think about when it is best to use a List, Deque, SSet, USet, or Map, among others DSs;
- to **only store the data you need** in order to solve the problem;
- to **revisit some older problems**, slightly modified (or, not!), to see if our modifications or new tools require or provide new approaches;
- to deeply understand the i**mplementation of the Graph** data structure (Implementation Practice), and
- to construct algorithms and/or use ideas similar to those seen in lecture so far, such as
    - thinking about **sorting algorithms**,
    - **constructing a graph**,
    - **traversing a graph**.
- 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. 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 5 (multiple-choice questions) on brightspace after reading this specifications document carefully, and optionally watching the lab 5 workshop video once it is available. The pre-lab is **due 5 (academic) days before** the lab programming deadline (in this case, by Friday March 24th, 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 5 Zip File (Lab5.zip on brightspace under the lab 5 module), which contains a skeleton of the code you need to write. You should have the following files in the comp2402w23l5 directory: Part1.java, Part2.java, Part3.java, Part4.java, Part5.java, Part6.java, MyAdjacencyLists.java, UGraph.java, and then a bunch of files that you will not edit but might use in the ods/ directory; 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 Lab5.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 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 imports other than those provided. For this lab, you have all of the JCF available to you through java.util.*. You also have many ods/textbook implementations if you prefer those (e.g. ArrayDeque, RootishArrayStack, SEList, Treap, ScapegoatTree, Graph, Algorithms).

**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 Lab5.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 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 **last x lines** that are divisible by the previous line, in reverse order, where x >= 0 is a parameter to the execute method. For example, if the input is

4
2 ← not divisible by the previous line
8 ← divisible by 2 so 8 is a multiple
8 ← divisible by 8 so 8 is a multiple
2
4 ← divisible by 2 so 4 is a multiple
8 ←divisible by 4 so 8 is a multiple

The multiples in forwards order are

8

8
4
8

So if x=2 we output

8
4

if x=3 we output

8
4
8

For more examples, see the tests in the tests/ directory that have the form lab5-p1-X-in.txt for the input, and the matching lab5-p1-X-out.txt for the expected output.

Part 2. [10 marks] **(Lab 5 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 **last x distinct values** from lines that are divisible by the previous line, in reverse order from the end, where x >= 0 is a parameter to the execute method. For example, if the input is

4
2 ← not divisible by the previous line
2 ← divisible by 2 so 2 is a multiple
8 ← divisible by 2 so 8 is a multiple
8 ← divisible by 8 so 8 is a multiple
2
4 ← divisible by 2 so 4 is a multiple
8 ←divisible by 4 so 8 is a multiple

The multiples in forwards order are

2 ← “third” multiple we output
8
8
4 ← “second” multiple we output
8 ← “first” multiple we output

So if x=2 we output the last 2 distinct multiples (in reverse order)

8
4

if x=3 we output the last 3 distinct multiples

8
4
2

As another example, consider

1

3
6
12
1
3
6
30

if x=3 we output

30
6
3

For more examples, see the tests in the tests/ directory that have the form lab5-p2-X-in.txt for the input, and the matching lab5-p2-X-out.txt for the expected output.

Part 3. [10 marks] **(Lab 3 Part 4 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 **first x** lines that are divisible by the previous line, in (increasing) sorted order, ~~with ties removed~~, where x is a parameter to the execute method For example, if x=4 and 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

4
8
8

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

However, if we have x=2 then the output would be

8
8

as the 8s were the first 2 multiples read in.

Note that this is **almost the same question as Lab 3 Part 4**, so the focus of this question is on performance, not on correctness. How can you improve the performance over your solution of Lab 3 Part 4, using a different approach? In particular, I'd like a solution that uses as little space as possible.

For more examples, see the tests in the tests/ directory that have the form lab5-p3-X-in.txt for the input, and the matching lab5-p3-X-out.txt for the expected output.

Part 4. [10 marks] **(Lab 3 Part 4 and Lab 4 Part 4 Redux.)** Given a file where each line is a single (32-bit) integer in the range [1, 240222], read in the lines of the file one at a time; for each line that is divisible by its previous, group it together with other lines of the same value, and order these groups in sorted order from lowest to highest multiple. But instead of outputting the multiple (as done in Lab 4 Part 4 and other similar problems), output its previous line. In a group of duplicate multiples, the previous lines should be output in the order they were seen. That is, all lines with a common next multiple will get output consecutively, in the order in which the values were first seen.

This sounds more confusing than it is. Let me give you an alternate description similar to Lab 3 Part 4 (and Lab 3 Part 2): output the lines that are divisors of their next line, in (increasing) sorted order of the next lines. For example, if the input is

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

The output is therefore

2 ← due to the multiple 4, multiple of 2
2 ← due to the first multiple 8, multiple of 2
8 ← due to the second multiple 8, multiple of 8
4 ← due to the last multiple 8, multiple of 4

as the 4 is the smallest multiple seen (its previous is 2), then the 3 8s, with their previous lines of 2, 8, and 4 (in the order they were seen.

For more examples, see the tests in the tests/ directory that have the form lab5-p5-X-in.txt for the input, and the matching lab5-p5-X-out.txt for the expected output.

Part 5. [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; output the line if it is divisible by **any prior** line. For example, if the input is

6
8 ← not divisible by the only prior line
4 ← not divisible by either prior lines

2 ← not divisible by any prior line
10 ← divisible by 2 so **output 10**
6 ← divisible by 2 (and 6) so **output 6**
5 ← not divisible by any prior line
25 ← divisible by 5 so **output 25**
8 ← divisible by 2 (and 4) so **output 8**

For more examples, see the tests in the tests/ directory that have the form lab5-p5-X-in.txt for the input, and the matching lab5-p5-X-out.txt for the expected output.

Note: Unlike other problems we've seen, there are some helper methods provided to you if you want them. I imagine that there are many possible solutions to this problem, and some of you might want to explore solutions that use prime divisors (or just divisors) or primality testers or gcd algorithms, etc. I tried to anticipate your needs and provide them to you. You should feel free to use these and/or modify them as you see fit.

Generally,

isPrime(v) – returns T if v is prime, F otherwise
primeFactors(v) – returns a List of the prime factors of v (with duplicates)
factors(v) – returns a List of all factors of v (including 1 and v itself)
gcd(a,b) – returns the greatest common divisor of a and b

Part 6. Given a file where each line is a single **distinct** (32-bit) **non-negative** integer, read in the lines of the file one at a time. Construct a directed graph out of the lines as follows:

1. there are x nodes $u_0$, $u_1$, $u_2$, …, $u_{x-1}$, where x is given as input to the execute method;
2. the edges are defined by pairs of consecutive lines; that is, (line 2i, line 2i+1) define the directed edges based on their value % x
   e.g. edge from node (value of line 2i)% x to node (value of line 2i+1) % x

Once you've constructed this graph, run a breadth-first-search (BFS) out of node $u_0$ (the node represented by line 0 of the file) and output the values in the order they are encountered in BFS. (Where when there are multiple edges to add on an iteration of BFS, add the edge that was encountered in the input file earliest. This is likely the default behaviour but if you've solved the problem differently, ensure this property.)

For example, if x=5 then there are 5 nodes $u_0$, $u_1$, $u_2$, $u_3$, $u_4$ and if the input is

0 ← even-indexed line; it is the start of an edge from $u_0$ since line's val is 0
1 ← odd-indexed line; it is the end of the edge ($u_0$, $u_1$) since line's val is 1
1 ← even-indexed line; it is the start of an edge from $u_1$ since line's val is 1
2 ← odd-indexed line; it is the end of the edge ($u_1$, $u_2$) since line's val is 2

2 ← even-indexed line; it is the start of the edge from $u_2$ since line's val is 2
8 ← odd-indexed line; end of the edge ($u_2$, $u_3$) since line's val is 8 and 8%5=3
3 ← even-indexed line; it is the start of the edge from $u_3$ since line's val is 3
14 ← odd-indexed line; end of the edge ($u_3$,$u_4$) since line's val is 14 and 14%5=4.

then the graph is actually a path $u_0 \to u_1 \to u_2 \to u_3 \to u_4$ and a BFS from $u_0$ would output

0
1
2
3
4

For more examples, see the tests in the tests/ directory that have the form lab5-p6-X-in.txt for the input, and the matching lab5-p6-X-out.txt for the expected output.

Note: there will always be an even number of lines. There may be duplicate edges, however, but the ods graph implementation will not add duplicates.

Note: You should look at the ods.Graph, ods.AdjacencyLists, and ods.Algorithms classes, which contain the (directed) graph interface, an implementation of that interface, and an implementation of the bfs graph algorithm, respectively. AdjacencyLists.mesh is an example of constructing a graph using an AdjacencyList.

## Implementation Practice [20 marks]

The 2 parts in this section ask you to implement two methods in the MyAdjacencyLists class, with the goal of getting your hands dirty with its implementation. Construct algorithms for these methods that are time- and space-efficient.

This lab is more flexible than previous labs: if you want to make changes to the implementation of the underlying undirected graph methods, you may. You should ensure they remain time-and space-efficient. It will be okay if the edges(i) method becomes O(degree(i)), if necessary, provided your other methods still run quickly.

Part 7. [10 marks] Implement the `controlSet(S)` method, which takes as input a set S of nodes, and returns the size of the control set of S. The control set of S is the set of vertices that are within distance 1 of some vertex in S. (i.e either it is in S, or it has an edge to some vertex in S.)

The idea is to implement this method using the class variables provided. You may change the underlying class variables and/or the implementation of the provided methods, provided that they still work and are reasonable in their time and space usage.

Part 8. [10 marks] In the MyAdjacencyLists class, implement the `numRegions()` method, which returns the number of regions in a graph. A region is a set of nodes such that there is a path between any two nodes in a region. (Visually, a region is a blob of nodes that do not overlap with any other blob.)

For example, for a graph that is a path (such as a SLList if the directionality of the edges were removed) there would be one region. If we removed any one edge from the path graph, then we would have two regions. A graph on n nodes with 0 edges has n regions.

There are some basic tests in `MyAdjacencyLists.main` that you should examine and play around with. If you want to run `MyAdjacencyLists.main` with the assertions, use the -ea parameter, e.g.

```
% java -ea comp2402w23l5.MyAdjacencyLists
```

## Post-Lab [13.3 marks]

Do the post-Lab 5 (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 April 5th, 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 5 sample solutions video, to figure out what went wrong for you, and to still learn what you were meant to learn.

# Local Tests

As usual, local tests are provided as in Labs 1-4. If you need more information, follow the instructions from those labs.

# Tips, Tricks, and FAQs

See the Lab 5 FAQ post on piazza (and/or the Lab5 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.

# 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 5 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. "lab5" for lab 5) 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 comp2402w23l5;" or "package ods;" 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. If you think Prof Alexa has forgotten to add an import, please ask on piazza and she'll check it out.
- 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. Marks are awarded for performance, not correctness for this reason.

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 most recent 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 lab2402w23l5/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 comp2402w23l5 directory, you'll be able to compile, but you won't be able to run. You have to go up a directory (outside the comp2402w23l5 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?