COMP 2402 Winter 2023

Lab 4 Specifications

Due: Wednesday March 15 3:00pm (or, 11:59pm if you wish)
Topic focus: Lec 12 - 15
lates accepted by March 16 3:00pm
submit on gradescope.ca early and often

Note about Lab 4 & Midterm Test

As you might notice, Lab 4 is due 1-2 days after your midterm test.

Lab 4 is a good study tool, so I recommend you work on it as part of your studying for the midterm. (The labs are cumulative, and so Lab 4 does review older material!)

Having said that, while there are 80 points available on this lab, we will take your score out of 60. This will be done on brightspace. Any points you get over 60 will be considered bonus points.

Separately, for any problem you get 10/10 on prior to Wednesday March 8th at 3pm I will give one extra point (so, 11/10 – except I probably have to account for this differently on brightspace.) This is to encourage you to start the lab early and not leave it until after the midterm.

More information about the midterm test can be found on the midterm FAQ piazza post.

Feedback on Lab 2 Feedback

I won't go into the same level of detail as I did with Lab 1 feedback, only due to time reasons.

Mostly, you are generally learning things from both types of questions on the lab (both the interface practice and the implementation practice.) This is good news.

The bad news is that it is taking a lot of time for all of you, more than any of us would like. There are a few reasons listed for this, such as

- hard to understand specifications,
- autograder tests failing but not being able to see the tests themselves,
- local tests that aren't extensive enough to catch all the possible problems you might encounter.

I will try to be clearer on the specifications going forward. I'm not *trying* to be unnecessarily convoluted, so I do apologize that it happens. Sometimes the questions end up being a bit convoluted in order to ensure that, say, a SSet is the best choice over a DLList, etc. But I can and will do better. I do have a TA in charge of proofreading the labs, but sometimes things get by both me and that TA.

As for the autograder tests, I am a bit limited in what information I can pass along, due to the way the autograder is structured. **I will continue to add the information about the size of n** (or k, or x, etc.), and things like "lots of duplicates" or "all distinct." I can't say things like "this test is trying to check that you didn't use a List when a SSet was the better choice" even if that is what the test is testing for, because that gives away the solution! But more importantly, the autograder does limit what I can tell you and what I can do. I cannot, for example, have a timeout at 3 seconds but then keep your program running and say "you hit the 3 second limit, because your program took 7 seconds". I just don't have that control over the autograder.

As for the **local tests, I've beefed them up in lab 4 to the extent that I can**. I can't make them exhaustive as I cannot always anticipate all the different ways you might write different solutions and different errors. But I'm trying to add more so that there is more chance to catch your errors locally. **I added a large local test for each of Parts 1-6**. I apologize for the space they take up. Feel free to delete them!

Finally, there were some comments about the prelab/postlabs. **I can/will remove the administrative questions as you wish**, although those are the "freebies," and everyone can always use reminders about what resources are available. I figured not everyone did lab 1, so repeating those questions on lab 2 was a good idea. But I suppose most people will have done at least one of prelab1 and/or prelab 2, so by prelab 3 I can take them out.

I think my biggest takeaway from your feedback is that Part 4 was too challenging! If I were to use such a problem in the future, I would find a way to make it have less edge cases and easier to understand.

Thanks, all of you, for your feedback! I hope you feel that I have listened to what you have to say and that I will try to make improvements going forward.

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 4 aims to encourage students:

- to think about situation where a **PriorityQueue, HashTable/USet/Map would be preferable to a SSet or a List and vice-versa**, by deciding which of these to use, if any, to solve a variety of problems (Interface Practice);
    - note that the ods Hashtable appears to have a bug so you should use JCF's HashSet, HashMap, etc.
- 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 BinaryHeap** data structures as well as how the **Hashtable uses haschode and equals**, by implementing some new methods in the MyBinaryHeap and HCPractice classes (Implementation Practice), and
- to construct algorithms and/or use ideas similar to those seen in lecture so far, such as
    - using the **heap order property**,
    - **swapping parent & child values**, as in a heap's bubbleUp/trickleDown,
    - **rehashing and resizing a hashtable**.
- 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 4 (multiple-choice questions) on brightspace after reading this specifications document carefully, and optionally watching the lab 4 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 10th, 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 4 Zip File (Lab4.zip on brightspace under the lab 4 module), which contains a skeleton of the code you need to write. You should have the following files in the comp2402w23l4 directory: Part1.java, Part2.java, Part3.java, Part4.java, Part5.java, Part6.java, DefaultComparator.java, MyBinaryHeap.java, HCPractice.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, Treap, ScapegoatTree, RedBlackTree, 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 Lab4.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 data structure other than those provided: (ods/textbook implementations of) ArrayStack, ArrayQueue, ArrayDeque, DualArrayDeque, RootishArrayStack, SLList, DLList, SEList, SkiplistList, BinaryHeap, RedBlackTree, etc. You may use java.util.HashMap, java.util.HashSet, java.util.Hashtable, and/or java.util.Dictionary, as I think the ods/textbook hashtable implementation has some bugs (!!). In Part 6 you may additionally use java.util.TreeMap and ods.SkiplistSSet (which are not allowed on other parts.) They are not necessary for at least one optimal solution, but if you want them you can use them!

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 3 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, for each node in level-order (i.e. in the order in which you read them in from the file), if node/line i violates the Heap Order property (in that it is (strictly) smaller than its parent), swap it with its parent (just the one swap. Don't continue up the tree.) For example, if the input is

30 ← has no parent, so it's fine

10 ← 10 < 30 so we swap 10 and 30
20 ← 20> 10 so this is good
3 ← 3 < 30 so we swap 3 and 30
2 ← 2 < 3 so we swap 2 and 3
50 ← 50 > 20 so this is good
15 ← 15 < 20 so we swap 15 and 20
40 ← 40 > 30 so this is good
60 ← 60 > 30 so this is good

then the output is

10
2
15
30
3
50
20
40
60

For more examples, see the tests in the tests/ directory that have the form lab4-p1-X-in.txt for the input, and the matching lab4-p1-X-out.txt for the expected output.

Note: a complete binary tree is a binary tree where every level is as full as possible, except for possibly the last level, which is filled in from left to right. (This is in contrast to a complete full binary tree, which is a complete binary tree where even the last level is exactly full.)

Part 2. [10 marks] **(Lab 3 Part 3 Redux)** 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; you're going to store and output certain multiples of x, where x>0 is a parameter to the execute method. As you read the file, store the multiples of x; each time you see a multiple of x^2, print out the smallest multiple of x you still have; each time you see a multiple of 2^x you get rid of the smallest multiple of x you still have (if you see something that is both a multiple of x^2 and 2^x print then remove; if you have something that is a multiple of x, x^2 and 2^x then print, add, then remove.) For example, if x=3 (so x^2=9 and 2^x=8) and the input is

3	← multiple of x	[3]
6	← multiple of x	[3, 6]
9	← multiple of x and x^2 – output 3	[3, 6, 9]
9	← multiple of x and x^2 – output 3	[3, 6, 9, 9]
5	← not a multiple of x, x^2 or 2^x	[3, 6, 9, 9]
8	← multiple of 2^x – get rid of the 3	[6, 9, 9]

9	← multiple of x and x^2 – output 6	[6, 9, 9, 9]
16	← multiple of 2^x – get rid of the 6	[9, 9, 9]
8	← multiple of 2^x – get rid of 9	[9, 9]
8	← multiple of 2^x – get rid of 9	[9]
27	← multiple of x and x^2 – output 9	[9, 27]
72	← multiple of x, x^2 and 2^x – output 9 *then* get rid of 9	[27, 72]
9	← multiple of x and x^2 – output 27	[9, 27, 72]
8	← multiple of 2^x – get rid of 9	[27, 72]
8	← multiple of 2^x – get rid of 27	[72]
8	← multiple of 2^x – get rid of 72	[]
8	← multiple of 2^x – nothing to remove	[]
9	← multiple of x and x^2 – output nothing	[9]
9	← multiple of x and x^2 – output 9	[9]

Then the output is

3
3
6
9
9
27
9

For more examples, see the tests in the tests/ directory that have the form lab4-p2-X-in.txt for the input, and the matching lab4-p2-X-out.txt for the expected output.

Part 3. [10 marks] **(Lab 3 Part 1)** 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 (distinct in the sum) 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 9 is already in sum 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.

Note that this is **exactly the same question as Lab 3 Part 1**, 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 1, using different data structures?

For more examples, see the tests in the tests/ directory that have the form lab4-p3-X-in.txt for the input, and the matching lab4-p3-X-out.txt for the expected output.

Part 4. [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 lines that are divisible by the previous line, with duplicates grouped together with the first instance that value is seen. That is, all instances of a particular value will get output consecutively, in the order in which the values were first seen. 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
8 ←divisible by 4 so will be in the output

The output is therefore

8
8
8
4

as the 8s are the first multiple seen, then the 4.

For more examples, see the tests in the tests/ directory that have the form lab4-p4-X-in.txt for the input, and the matching lab4-p4-X-out.txt for the expected output.

Part 5. [10 marks] **(Lab 2 Part 6 & Lab 3 Part 5 Remix.)** 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.) Each line u is red or black, depending on u%2, where:

`u%2 == 0`: (red) node u is red
`u%2 == 1`: (black) node u is black

In this tree, if the root is red, recolour it black by incrementing its value by 1. For every other node u in level-order (i.e. in the order in which you read them in from the file), if

node/line u violates the “no red-red edge” property (in that both node u and its parent w are red), perform a “pushBlack” operation on u’s grandparent (and w’s parent) g, which recolours g and g’s children as follows:

```
pushBlack(g)
   g--                // change g’s colour
   g.left++           // change g’s left child’s colour
   g.right++          // change g’s right child’s colour
```

Note that we’re only doing one pushBlack per node u (we don’t continue up the tree as you would in a red-black tree, so you may not get rid of all red-red edges in this way.) For example, if the input is

30 ← (R) root is red; recolour 30→31 (B)
10 ← (R) since the root is now black (31), this does not form a R-R edge ✓
20 ← (R) since the root is now black (31), this does not form a R-R edge ✓
3 ← (B) this is black so it’s not part of a R-R edge ✓
2 ← (R) its parent 10 is also red; pushBlack(root) 31→30, 10→11, 20→21
50 ← (R) since its parent is now black (21), this is not a R-R edge ✓
15 ← (B) this is black so it’s not part of a R-R edge
40 ← (R) its parent 3 is black so it’s not part of a R-R edge
60 ← (R) its parent 3 is black so it’s not part of a R-R edge
8 ← (R) its parent 2 is also red; pushBlack(11) 11→10, 3→4, 2→3

And so we would output the final list

30
10
21
4
3
50
15
40
60
8

Note that the final tree still has red-red edges that we might have created by fixing later edges. That’s okay. We’re not in it to win it this time.

For more examples, see the tests in the tests/ directory that have the form lab4-p5-X-in.txt for the input, and the matching lab4-p5-X-out.txt for the expected output.

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;

1. start with the table of size 2
2. hash line i to (line i’s value) % table.size
3. if there is a collision:
    - resize the table to (2*table.size)+(index of collision)

- rehash the elements in the existing table starting from index 0. If there is a collision at any point, resize the table again immediately to (2*table.size)+(index of this newest collision), and repeat the rehash/resize process (until there are no more collisions).
- when the existing table finally has no collisions, try to add line i to this table; if there is a collision, repeat step 3.

4. return the sequence of collision-free table sizes starting with 2.

For example, if the input is

6 ←6%2=0 so we add to index 0 [6,--]
5 ←5%2=1 so we add to index 1 [6,5]
4 ←4%2=0; collision at index 0, resize to 2*2+0=4 and rehash [4,5,6,--]
3 ←3%4=3 so we add to index 3 [4, 5, 6, 3]
2 ←2%4=2; collision at index 2, resize to 2*4+2=10 and rehash [-,-,2,3,4,5,6,-,-,-]
0 ← 0%10=0 so we add to index 0 [0,-,2,3,4,5,6,-,-,-]
20 ← 20%10=0; collision at index 0, resize to 2*10+0=20 but when we rehash we still have a collision of 20%20=0 so we resize to 2*20+0=40 and we can rehash to get [0,-,2,3,4,5,6,-,....,-,20,-,...,-]

the final sequence of resizes would thus be

2
4
10
40

For more examples, see the tests in the tests/ directory that have the form lab4-p6-X-in.txt for the input, and the matching lab4-p6-X-out.txt for the expected output.

Note that two numbers that aren't colliding in a current size of list might collide in the next rehashing, e.g., if the current size is 4, consider that 6%4 = 2 and 15%4 = 3. If there is now a collision at index 1, size becomes 2*4+1=9. When we rehash, 6%9 = 6 and 15%9 = 6, which would cause a collision that doesn't involve the most recent number read (the one that caused the collision at index 1.) We would resolve this "pre-existing" collision first before trying to place the new element.

Note that for this problem, you may use java.util.TreeMap and/or ods.SkiplistSSet. There are a variety of approaches that may use these (and some that do not!) but I wanted to provide them to you in case you wanted them.

To get the last point, you might need to combine different approaches and decide between them depending on n!

Implementation Practice [20 marks]

The 2 parts in this section ask you to implement a single (new) method in each of the MyBinaryHeap and Flight 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) MyBinaryHeap and Flight classes are (incomplete) implementations. Your task is to complete the implementations of the methods specified below. Default implementations are provided for the 3 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] **(Lab 1 & Lab 2 Redux.)** In the MyBinaryHeap class, implement

public void copy(int k)

This replaces each element x with k copies of element x (as separate elements) but maintains the heap-order property in the heap. You may assume that k ≥ 0.

For example,

- If the MyBinaryHeap h originally represents the list [a,b,c] then after h.copy(3), h now represents the heap that contains the elements a, a, a, b, b, b, c, c, c.
- For any k ≥ 0, if h=[ ] then after hml.copy(k), h now represents [ ].
- For any MyBinaryHeap h, after h.copy(0), h represents [ ].

The idea is to implement this method using the class variables provided. You should not need to use 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 `Flight` class (within the `HCPractice` class), implement the `equals` and `hashCode` methods so that we can correctly store a Set of `Flights` without duplicates, that is, we want to be able to add, remove, and find distinct Flights. Each Flight has 5 characteristics:

`number` – flight number - represented as a String (e.g. "AC886")
`origin` – code of the origin airport - represented as a String (e.g. "YYZ")
`dest` – code of the destination airport - represented as a String
`date` – the date of the originally scheduled departure – represented as a Date
`status` – the flight's current status – represented as an Integer (e.g. 0="on time", 1="delayed", 2="canceled", 3="boarding", etc. The exact values do not matter. But this value might change as the status of the flight changes.

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

```
% java -ea comp2402w23l4.HCPractice
```

Post-Lab [13.3 marks]

Do the post-Lab 4 (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 22nd, 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 4 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 1-3. If you need more information, follow the instructions from those labs.

Tips, Tricks, and FAQs

See the Lab 4 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.

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 4 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. "lab4" for lab 4) 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 comp2402w23l4;" 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 lab2402w23l4/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 comp2402w23l4 directory, you'll be able to compile, but you won't be able to run. You have to go up a directory (outside the comp2402w23l4 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?