Comp 251: Assignment 1

Answers must be returned online by October 16th (11:59pm), 2020.

General instructions (Read carefully!)

- **Important:** All of the work you submit must be done by only you, and your work must not be submitted by someone else. Plagiarism is academic fraud and is taken very seriously. For Comp251, we will use software that compares programs for evidence of similar code. This software is very effective and it is able to identify similarities in the code even if you change the name of your variables and the position of your functions. The time that you will spend modifying your code, would be better invested in creating an original solution.

  Please don't copy. We want you to succeed and are here to help. Here are a couple of general guidelines to help you avoid plagiarism:

  Never look at another assignment solution, whether it is on paper or on the computer screen. Never share your assignment solution with another student. This applies to all drafts of a solution and to incomplete solutions. If you find code on the web, or get code from a private tutor, that solves part or all of an assignment, do not use or submit any part of it! A large percentage of the academic offenses in CS involve students who have never met, and who just happened to find the same solution online, or work with the same tutor. If you find a solution, someone else will too. The easiest way to avoid plagiarism is to only discuss a piece of work with the Comp251 TAs, the CS Help Centre TAs, or the COMP 251 instructors.

- Your solution must be submitted electronically on codePost. Here is a short **tutorial** to help you understand how the platform works. You will receive an email inviting you to join the class there in early October.

- To some extent, collaborations are allowed. These collaborations should not go as far as sharing code or giving away the answer. You must indicate on your assignments (i.e. as a comment at the beginning of your java source file) the names of the people with whom you collaborated or discussed your assignments (including members of the course staff). If you did not collaborate with anyone, you write "No collaborators". If asked, you should be able to orally explain your solution to a member of the course staff.

- This assignment is due on October 16^{th} at 11h59:59 pm. It is your responsibility to guarantee that your assignment is submitted on time. We do not cover technical issues or unexpected difficulties you may encounter. Last minute submissions are at your own risk.

- This assignment includes a programming component, which counts for 100% of the grade, and an optional long answer component designed to prepare you for the exams. This component will not be graded, but a solution guide will be published.

- Multiple submissions are allowed before the deadline. We will only grade the last submitted file. Therefore, we encourage you to submit as early as possible a preliminary version of your solution to avoid any last minute issue.

- Late submissions can be submitted for 24 hours after the deadline, and will receive a flat penalty of 20%. We will not accept any submission more than 24 hours after the deadline. The submission site will be closed, and there will be no exceptions, except medical.

- In exceptional circumstances, we can grant a small extension of the deadline (e.g. 24h) for medical reasons only. However, such request must be submitted before the deadline, and justified by a medical note from a doctor, which must also be submitted to the McGill administration.

- Violation of any of the rules above may result in penalties or even absence of grading. If anything is unclear, it is up to you to clarify it by asking either directly the course staff during office hours, by email at (`cs251@cs.mcgill.ca`) or on the discussion board on Piazza (recommended). Please, note that we reserve the right to make specific/targeted announcements affecting/extending these rules in class and/or on the website. It is your responsibility to monitor Piazza for announcements.

- The course staff will answer questions about the assignment during office hours or in the online forum. We urge you to ask your questions as early as possible. We cannot guarantee that questions asked less than 24h before the submission deadline will be answered in time. In particular, we will not answer individual emails about the assignment that are sent sent the day of the deadline.

Programming component

- You are provided some starter code that you should fill in as requested. Add your code only where you are instructed to do so. You can add some helper methods. Do not modify the code in any other way and in particular, do not change the methods or constructors that are already given to you, do not import extra code and do not touch the method headers. The format that you see on the provided code is the only format accepted for programming questions. **Any failure to comply with these rules will result in an automatic 0.**
- Public tests cases are available on codePost. You can run them on your code at any time. If your code fails those tests, it means that there is a mistake somewhere. Even if your code passes those tests, it may still contain some errors. We will grade your code with a more challenging, private set of test cases. We therefore highly encourage you to modify that tester class, expand it and share it with other students on the discussion board. Do not include it in your submission.
- Your code should be properly commented and indented.
- **Do not change or alter the name of the files you must submit, or the method headers in these files**. Files with the wrong name will not be graded. Make sure you are not changing file names by duplicating them. For example, main (2).java will not be graded. Make sure to double-check your zip file.
- Do not submit individual files. Include the .java files into a .zip file and upload it to codePost. **The zip must include ONLY the .java files, and no subfolders.**

- **You will automatically get 0 if the files you submitted on codePost do not compile, since you can ensure yourself that they do. Note that public test cases do not cover every situation and your code may crash when tested on a method that is not checked by the public tests. This is why you need to add your own test cases and compile and run your code from command line on linux.**

Homework

Exercise 1 (100 points)**.** ***Building a Hash Table*** We want to compare the performance of hash tables implemented using chaining and open addressing. In this assignment, we will consider hash tables implemented using the multiplication and linear probing methods. We will (respectively) call the hash functions h and g and describe them below. Note that we are using the hash function h to define g.

$$\begin{aligned} \text{Collisions solved by chaining (multiplication method):} \quad & h(k) = ((A \cdot k) \mod 2^w) >> (w - r) \\ \text{Open addressing (linear probing):} \quad & g(k, i) = (h(k) + i) \mod 2^r \end{aligned}$$

In the formula above, r and w are two integers such that $w > r$, and A is a random number such that $2^{w-1} < A < 2^w$. In addition, let n be the number of keys inserted, and m the number of slots in the hash tables. Here, we set $m = 2^r$ and $r = \lceil w/2 \rceil$. The *load factor* α is equal to $\frac{n}{m}$.

We want to estimate the number of collisions when inserting keys with respect to keys and the choice of values for A.

We provide you a set of three template files within `COMP251HW1.zip` that you will complete. This file contains three classes, a main class and one for each hash function. Those contain several helper functions, namely `generateRandom` that enables you to generate a random number within a specified range. Details on which functions are included, how to use them, and where to add in your code can be found as comments in the java files. Please read them with attention.

Your first task is to complete the two java methods `Open_Addressing.probe` and `Chaining.chain`. These methods must implement the hash functions for (respectively) the linear probing and multiplication methods. They take as input a key k, as well as an integer $0 \leq i < m$ for the linear probing method, and return a hash value in $[0, m[$.

Next, you will implement the method `insertKey` in both classes, which inserts a key k into the hash table and returns the number of collisions encountered before insertion, or the number of collisions encountered before giving up on inserting, if applicable. Note that for this exercise as well as for the rest of the homework, we define the number of collisions in open addressing as the number of keys encountered, or "jumped over" before inserting or removing a key. For chaining, we simply consider the number of other keys in the same bin at the time of insertion as the number collisions. You can assume the key is not negative.

You will also implement a method `removeKey`, this one only in `Open_Addressing`. This method should take as input a key k, and remove it from the hash table while visiting the minimum number of slots possible. Like `insertKey`, it should output the number of collisions. If the key is not in the hash table, the method should simply not change the hash table, and output the number

of slots visited. You will notice from the code and comments that empty slots are given a value of -1. If applicable, you are allowed to use a different notation of your choice for slots containing a deleted element.

You can then implement tests in the `main` class. Make sure to test your assignment thoroughly by thinking about all the different situations that can occur when dealing with hash tables.

For this assignment, you will need to submit a zip file containing the completed version of the three provided java files on codePost.

Exercise 2 (0 points)**.** ***Least common multiple*** This problem aims to study an algorithm that computes, for an integer $n \in \mathbb{N}$, the least common multiple (LCM) of integers $\leq n$.

For a given integer $n \in \mathbb{N}$, let $P_n = p_1^{x_1} p_2^{x_2} \cdots p_k^{x_k}$, where $p_1, p_2, \cdots, p_k$ is a strictly increasing sequence of prime numbers between 2 and n and for each $i \in \{1, \cdots k\}$, x_i is the integer such that $p_i^{x_i} \leq n < p_i^{x_i+1}$. For example, $P_9 = 2^3 \cdot 3^2 \cdot 5 \cdot 7$.

More precisely, we're going to compute all P_j, $j \in \{1, \cdots, n\}$ and store pairs of integers (p^α, p) in a heap, a binary tree where the element stored in the parent node is **strictly smaller** than those stored in children nodes. For two given pairs of integers (a, b) and (a', b'), $(a, b) < (a', b')$ if and only if $a < a'$. Let h denotes the tree height, we admit that $h = \Theta(\log n)$. All levels of the binary tree are filled with data except for the level h, where elements are stored from the left to the right. After computing P_j, all pairs (p^α, p) are stored in the heap such that p is a prime number smaller or equal to j and α is the **smallest** integer such that $\mathbf{j < p^\alpha}$. For instance, after computing P_9, we store $(16, 2)$, $(27, 3)$, $(25, 5)$, and $(49, 7)$ in the heap.

The algorithm is iterative. We store in the variable **LCM** the least common multiple computed so far. At first, **LCM**$= 2$ is the LCM of integers smaller than 2 and the heap is constructed with only one node with value $(4, 2)$. After finish the $(j-1)$-th step, we compute the j-th step as follows:

1. If j is a prime number, multiply **LCM** by j and insert a new node (j^2, j) in the heap.
2. Otherwise, if the root (p^α, p) satisfies $j = p^\alpha$, then we multiply **LCM** by p, change the root's value by $(p^{\alpha+1}, p)$, and reconstruct the heap.

We're going to prove, step by step, that the time complexity of this algorithm is $O(n\sqrt{n})$.

3.1 - 0 points
In operation 1, a new node is inserted. What is the complexity of this operation?

3.2 - 0 points
In operation 2, the heap is reconstructed. What is the time complexity of this operation?

3.3 - 0 points
The number of prime numbers smaller than n concerned in the operation 2 is less than $\sqrt{n}$. Prove that the number of times N we need to execute operation 2 to compute P_n is asymptotically negligible compared to n. Tip: you can prove this by proving N is $o(n)$, where o (little o) denotes a strict upper bound.

3.4 - 0 points
Assume the complexity of assessing whether an integer is a prime number is $\sqrt{n}$ and suppose multiplication has a time complexity of 1. Prove that the algorithm's complexity is $O(n\sqrt{n})$.

3.5 - 0 points
Prove that, for a given heap of height h with n nodes, we have $h = \Theta(\log n)$. No partial credit will be awarded.