Comp 251: Assignment 2

Answers must be submitted online by March 11th (11:55:00 pm), 2021.

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.

- 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 March 11^{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.

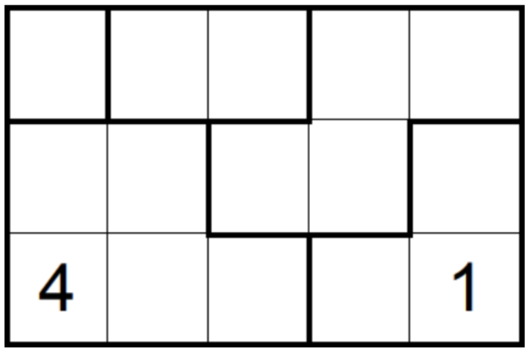
- Multiple submissions are allowed before the deadline. We will only grade the last submitted file. Therefore, we encourage you to submit as early as possible a preliminary version of your solution to avoid any last minute issue.
- Late submissions can be submitted for 24 hours after the deadline, and 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-winter@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

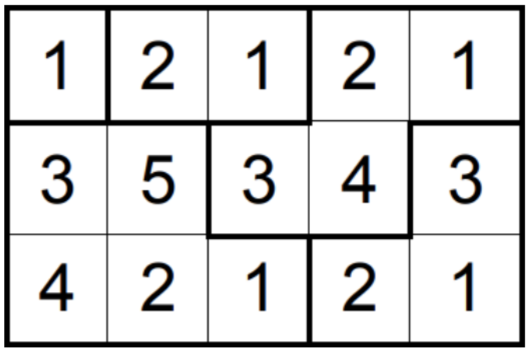
- You are provided some starter code that you should fill in as requested. Add your code only where you are instructed to do so. You can add some helper methods. Do not modify the code in any other way and in particular, do not change the methods or constructors that are already given to you, do not 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.
- **You can submit only individual files on codePost, which will compile and execute correctly under Java 8.** If you get more than 0 on the public tests, it means codePost accepted your files.
- **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.

Exercise 1 (Complete Search (35 points)) The University and the School of Computer Science approved yesterday the law "Sudoku". This law establishes that no computer science student can graduate without implementing (at least once) a Sudoku solving program. However, to make things more interesting, the law makes clear that a modification of the standard Sudoku must be solved before applying for graduation. As in Sudoku, this modification will place numbers in a grid (where the number of rows (R) and columns (C) will be in the following limits: $1 \leq R, C \leq 7$); but this grid will not be constrained to size 9 X 9. Furthermore, the grid will have outlined regions (not constrained to be a 3X3 region as in the standard sudoku) that must contain the numbers from 1 to n, where n is the number of the cells (i.e., squares) in the region. The most interesting new feature comes from the fact that the same number can never touch itself, not even diagonally.

The figure below depicts the initial incomplete grid and the grid with an accepted solution. Please note that this solution is unique (i.e., there is no other solution to the puzzle based on the described constrains). Furthermore, for the graduation test, you are guaranteed that your solver will be tested against problems that have a unique solution.



Incomplete Grid



Solution Grid

We will model the problem in java using an Object Oriented Approach. Particularly, we will model the Sudoku `Board` as an object that has regions (i.e., the outlined areas in the board) and values (i.e., and `int[][]` matrix containing the digits in each cell). Each `Region` is an object with two private attributes 1) An array of `Cells` (i.e., the squares of the grid) and 2) an `int` representing the number of cells. Finally, a `Cell` object will represent each square in the board, where the private attributes of the class denotes the position of the square in the board. Each of those classes has a set of getters and setter methods to access/modify their private information. The starter code contains four classes:

- `Cell`: This class represents a single cell of the game board. The `row` and `column` fields store the corresponding indices of the cell.
- `Region`: This class represents one region of the board. It contains an array of `Cell` objects. You can add fields to the Region class, but make sure your code still compiles on codePost.
- `Board`: This class represents the game board. It contains an `int[][]` matrix containing the digits in each cell, along with a array of `Region` objects representing the regions of the board.
- `Game`: This class encompasses the above classes and contains the main method.

The `Game` class has an attribute named `sudoku` of type `Board` that will contain all the initial information needed to complete the Sudoku puzzle. Inside the Class `Game`, you will also find the function `solver`. The signature of the function `solver` in the java file `Game.java` is as follows.

```
public int[][] solver() {
    //To Do => Please start coding your solution here
    return sudoku.getValues();
}
```

Your job for this part of the assignment is to complete the function `solver`, which takes an incomplete puzzle grid (stored initially in the `board_values` attribute of the object `sudoku`) and returns the output solution grid (i.e., the same `board_values` attribute of the object `sudoku`, but this time with all the cells filled). Please notice that a cell in the board is empty if it contains a -1 (i.e., no value).

Note: main function

We have already implemented a `main` function to read the data from the files, to create all the initial information (i.e., the incomplete grid and the description of its separate regions) and finally to create the `Board` object called `sudoku` (which will contain all the initial information of the sudoku game). Please note that this function will not be graded, and it is there only to make sure that all of the Comp251 students understand the input of the problem and to test their own code.

Exercise 2 (Dynamic Programming (35 points)) As described in our previous assignment, the teaching staff of Comp251 is getting ready to come back to in-person classes. Please remember that Comp251 lectures will happen then in the McGill Gym (because space constrains). Knowing how fun it is to have in-person midterms (is not it?), we are already planning our first exam. Each student will take the exam once at the time. For the exam, we will arrange a row on N chairs, numbered 1 to N. The student will be seated initially in the first chair and can jump to other chairs. The student's first jump must be to the second chair. Each subsequent jump must satisfy the following two constraints:

- If the jump is in the forward direction, it must be one square longer than the preceding jump.
- If the jump is in the backward direction, it must be exactly the same length as the preceding jump.

Every time that the student jump to sit in a chair, I will deduct some marks. The exam will be over once the student sit down in the last chair (i.e., the N chair). It is in the interest of the student to get from the first chair (i.e., chair 1) to the last chair (i.e., chair N) losing as few marks as possible. For this question of the assignment, you will complete the function `lost_marks` which determines the smallest total marks lost by the student in the attempt to get to the chair N.

The signature of the function `lost_marks` in the java file `Midterm.java` is as follows.

```
public static int lost_marks(int[] penalization) {
    //To Do => Please start coding your solution here

}
```

The function `lost_marks` gets as a parameter an array of integers called `penalization`, which stores the marks that we will deduct each time that the student sits in that specific chair. Please notice that `penalization[0]` represents the lost marks for the first chair (chair 1) and `penalization[N-1]` represents the lost marks for the last chair (chair N), in general, `penalization[i]` represents the lost marks for the i-th chair. The number of chairs N will be in the following limits $2 \leq N \leq 1000$ and I will always take less than 500 points (i.e., marks) as penalization for sitting in a specific chair.

Lets now see an example to make things even more clear:
Given the `penalization = {1, 2, 3, 4, 5, 6}` array, the goal of the student is to go from

the first chair to the last chair (notice that from the length of the array you know that $N = 6$). One possible way (actually this is the optimal way) to reach N would be by using the first-second-first-third-sixth chair. Please notice that this way satisfy the two jump constrains mentioned before. The total marks lost in this way is 12. Let's see the way with the penalization values beside them in parenthesis: (first-second(2)-first(1)-third(3)-sixth(6)). Please notice that the student is not penalized for the initial state of the exam (i.e., the first chair in which the student start the exam) because he did not jump to that chair (the student just started the exam there).

Ok, let's see another example to make things even more clear.

Given the `penalization = {2, 3, 4, 3, 1, 6, 1, 4}` array, the goal of the student is to go from the first chair to the last chair (notice that from the length of the array you know that $N = 8$). One possible way (actually this is the optimal way) to reach N would be by using the first-second-fourth-seventh-fourth-eight chair. Please notice that this way satisfy the two jump constrains mentioned before. The total marks lost in this way is 14. Let's see the way with the penalization values beside them in parenthesis: (first-second(3)-fourth(3)-seventh(1)-fourth(3)-eight(4)). Please notice that the student is not penalized for the initial state of the exam (i.e., the first chair in which the student start the exam) because he did not jump to that chair (the student just started the exam there).

Exercise 3 (Greedy algorithms (30 points)) In this exercise, you will plan your homework with a greedy algorithm. The input is a list of homeworks defined by two arrays: deadlines and weights (the relative importance of the homework towards your final grade). These arrays have the same size and they contain integers between 1 and 100. The index of each entry in the arrays represents a single homework, for example, `Homework 2` is defined as a homework with deadline `deadlines[2]` and weight `weights[2]`. Each homework takes exactly one hour to complete.

Your task is to output a `homeworkPlan`: an array of length equal to the last deadline. Each entry in the array represents a one-hour timeslot, starting at 0 and ending at 'last deadline - 1'. For each time slot, `homeworkPlan` indicates the homework which you plan to do during that slot. You can only complete a single homework in one 1-hour slot. The homeworks are due at the beginning of a time slot, in other words if an assignment's deadline is `x`, then the last time slot when you can do it is `x - 1`. For example, if the homework is due at t=14, then you can complete it before or during the slot t=13. If your solution plans to do `Homework 2` first, then you should have `homeworkPlan[0]=2` in the output. Note that sometimes you will be given too much homework to complete in time, and that is okay.

Your homework plan should maximize the sum of the weights of completed assignments.

To organize your schedule, we give you a class `HW_Sched.java`, which defines an `Assignment` object, with a number (its index in the input array), a weight and a deadline.

The input arrays are unsorted. As part of the greedy algorithm, the template we provide sorts the homeworks using Java's `Collections.sort()`. This sort function uses Java's `compare()` method, which takes two objects as input, compares them, and outputs the order they should appear in. The template will ask you to override this `compare()` method, which will alter the way in which Assignments will be ordered. You have to determine what comparison criterion you want to use. Given two assignments A1 and A2, the method should output:

- 0, if the two items are equivalent
- 1, if a1 should appear after a2 in the sorted list
- -1, if a2 should appear after a1 in the sorted list

The `compare` method (called by `Collections.sort()`) should be the only tool you use to reorganize lists and arrays in this problem. You will then implement the rest of the `SelectAssignments()` method.

Submit all the Java files on codePost. You will only be evaluated directly for time complexity on Q2. However, your code has to execute within 30 seconds on codepost for all test cases for all questions to be graded. This means that you should be particularly mindful of the possibility of infinite loops. An infinite loop or a runtime error will affect your global grade for the assignment, so take the time to think about the different inputs your assignment can receive, and make sure it never crashes. To help your testing, we have made half the test cases available for all questions on codePost. Take advantage of this, but you must complement these tests with your own to succeed on this assignment. We have added an example of a test file for Q1 and Q2. You can run them from command line with the following command: `java className < file`

Exercise 4 (Change-Making Problem (0 points) In this problem, we give a formal look at the problem of returning a given amount of money using the minimum number of coins (including bills), for a given coin system.

For example, the best way to give someone 7 dollars is to add up a 5 dollars bill and a 2 dollars coin. For simplicity, we will consider all denominations to be coins.

First, let us define the problem: a coin system is a m-tuple $c = (c_1, c_2, \dots, c_m)$ such that $c_1 > c_2 > \dots > c_m = 1$. For a given coin system c and a positive integer x representing the amount of money we want to gather, we want to find a solution (a m-tuple of non-negative integers) $k = (k_1, k_2, \dots, k_m)$ such that $x = \sum_{i=1}^{m} k_i c_i$ so as to minimize $\sum_{i=1}^{m} k_i$.

There exists a greedy algorithm to find the optimal solution for certain coin systems: for a given x, we select the largest coin $c_i \leq x$. Then, we repeat this step for $x - c_i$, and continue repeating it until x becomes 0. For instance, with the coin system $(10, 5, 2, 1)$, the algorithm decomposes $x = 27$ into $10, 10, 5$, and 2.

We describe a coin system as *canonical* if and only if the solution given by the greedy algorithm described above is optimal for any positive integer x. For example, all systems $(a, 1)$ with $a > 1$ are canonical. For any positive integer x, we can express such a change system in the form of Euclidean division: $x = aq + r$ with $r < a$. The greedy solution for x is then the tuple $g = (q, r)$. To prove that the solution is optimal, we can proceed as follows:

Let's consider $g' = (q', r')$ different than g such that $x = aq' + r'$. Because q is already at its highest value under x and cannot be above x, we have $q' < q$, otherwise $q' = q$ causes $g' = g$. Since $(q' + r') - (q + r) = (q' + x - aq') - (q + x - aq) = (a - 1)(q - q') > 0$, g' would sum up to more coins than the initial solution g, for any g' satisfying the problem definition. Thus, the solution g is optimal, and the system $(a, 1)$ is canonical.

4.1 (0 points) Design a non-canonical system of 3-tuple $c = (c_1, c_2, c_3)$ and prove your system is non-canonical.

4.2 (0 points) Let q and n be two integers ≥ 2. Prove that the system $c = (q^n, q^{n-1}, \dots, q, 1)$ is canonical.

4.3 (0 bonus points) Prove that the Euro system $c = (200, 100, 50, 20, 10, 5, 2, 1)$ is canonical. There will be no partial credit for this question.