Comp 251: Assignment 2

Answers must be submitted online by October 29th (11:55 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 would spend modifying your code to avoid being caught 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 should already be on the platform since this is the second assignment.

- 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, 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 29^{th} at 11:59 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.

- 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 Ed (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 Ed 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.

Technical considerations

- 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. You can discuss test cases with fellow students, but do not share files. List all the students you collaborated with on the top of your source code.
- Note that your code will be evaluated with Java 8 on a linux environment (similar to ubuntu). It is your responsibility to make sure your code compiles and runs correctly when executed from command line in this type of environment.
- 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. Do not add packages or change the import statements.
- Submit your files individually on codePost (no zips)
- **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 encounter runtime errors when tested on private tests. This is why you need to add your own test cases and compile and run your code from command line on a linux system (see the relevant tutorial on MyCourses).**

Exercise 1 (Disjoint sets (24 points)) We want to implement a disjoint set data structure with union and find operations. The template for this program is available on the course website and named `DisjointSets.java`.

In this question, we model a partition of n elements with distinct integers ranging from 0 to $n-1$ (i.e. each element is represented by an integer in $[0, \cdots, n-1]$, and each integer in $[0, \cdots, n-1]$ represent one element). We choose to represent the disjoint sets with trees, and to implement the forest of trees with an array named `par`. More precisely, the value stored in `par[i]` is parent of the element `i`, and `par[i]==i` when `i` is the root of the tree and thus the representative of the disjoint set.

You will implement union by rank and the *path compression* technique seen in class. The rank is an integer associated with each node. Initially (i.e. when the set contains one single object) its value is 0. Union operations link the root of the tree with smaller rank to the root of the tree with larger rank. In the case where the rank of both trees is the same, the rank of the new root increases by 1. You can implement the rank with a specific array (called `rank`) that has been added to the template, or use the array `par` (this is tricky). Note that path compression does not change the rank of a node.

Download the file `DisjointSets.java`, and complete the methods `find(int i)` as well as `union(int i, int j)`. The constructor takes one argument n (a strictly positive integer) that indicates the number of elements in the partition, and initializes it by assigning a separate set to each element.

The method `find(int i)` will return the representative of the disjoint set that contains `i` (do not forget to implement path compression here!). The method `union(int i, int j)` will merge the set with smaller rank (for instance `i`) in the disjoint set with larger rank (in that case `j`). In that case, the root of the tree containing `i` will become a child of the root of the tree containing `j`), and return the representative (as an integer) of the new merged set. Do not forget to update the ranks. In the case where the ranks are identical, you will merge `i` into `j`.

Once completed, compile and run the file `DisjointSets.java`. It should produce the output available in the file `unionfind.txt` available on the course website.

Note: You will need to complete this question to implement Exercise 2.

Exercise 2 (Minimum Spanning trees (24 points)) We will implement the Kruskal algorithm to calculate the minimum spanning tree (MST) of an undirected weighted graph. Here you will use the file `DisjointSets.java` completed in the previous question and two other files: `WGraph.java` and `Kruskal.java` available on the course website. You will need the classes `DisjointSets` and `WGraph` to execute `Kruskal.java`. Your role will be to complete two methods in the template `Kruskal.java`.

The file `WGraph.java` implements two classes `WGraph` and `Edge`. An `Edge` object stores all informations about edges (i.e. the two vertices and the weight of the edge), which are used to build graphs.

The class `WGraph` has two constructors `WGraph()` and `WGraph(String file)`. The first one creates an empty graph and the second uses a file to initialize a graph. Graphs are encoded using the following format. The first line of this file is a single integer n that indicates the number of nodes in the graph. Each vertex is labelled with an number in $[0, \cdots, n-1]$, and each integer in $[0, \cdots, n-1]$ represents one and only one vertex. The following lines respect the syntax "n_1 n_2 w", where n_1 and n_2 are integers representing the nodes connected by an edge, and w the weight of this edge. n_1, n_2, and w must be separated by space(s). An example of such file can be found on the course website with the file `g1.txt`. These files will be used as an input in the program `Kruskal.java` to initialize the graphs. Thus, an example of a command line is `java Kruskal g1.txt`.

The class `WGraph` also provide a method `addEdge(Edge e)` that adds an edge to a graph (i.e. an object of this class). Another method `listOfEdgesSorted()` allows you to access the list of edges

of a graph in the increasing order of their weight.
You task will be to complete the two static methods `isSafe(DisjointSets p, Edge e)` and `kruskal(WGraph g)` in `Kruskal.java`. The method `isSafe` considers a partition of the nodes p and an edge e, and should return `True` if it is safe to add the edge e to the MST, and `False` otherwise. The method `kruskal` will take a graph object of the class `WGraph` as an input, and return another `WGraph` object which will be the MST of the input graph.
Once completed, compile all the java files and run the command line `java Kruskal g1.txt`. An example of the expected output is available in the file `mst1.txt`. You are invited to run other examples of your own to verify that your program is correct. Note that you need to compile it with the two other files for it to work.

Exercise 3 (Greedy algorithms (52 points)) In this exercise you will plan your homework with a greedy algorithm. The input is a list of homeworks defined by three arrays: deadlines (when the homework is due), weights (relative importance of the homework towards your final grade), and completion times (number of hours required to complete a homework). These arrays have the same size, and they contain integers between 0 to 99. The index of each entry in the array represents a single homework. For example, homework 2 is defined as a homework with deadlines[2], weights[2], and completionTime[2].

Your task is to output a `homeworkPlan`. `homeworkPlan` is an ArrayList of integers, of size equal to the number of time slots available to complete the homeworks. Each entry in the array represents a 1 hour timeslot, starting at 0, and ending at last-deadline$+x-1$, where x is whatever additional time was needed to complete late homework. For each timeslot, `homeworkPlan` indicates the homework which you plan to do during that slot.

You can only work on one project at a time, and must work on it until complete. The homeworks are due at the beginning of a time slot x, so the last time slot you can work on it unpenalized is x-1. If a homework is due at t=14, then the last opportunity to work on it without consequence is t=13. Homework can be submitted late. For every hour surpassing the due date, lose 10% of the weight assigned to that homework. If a homework takes 3 hours to complete, and is worth 10%, submitting 2 hours late would result in 8% instead of 10%. Starting a homework past the due date results in 0 credits. Note that sometimes you will be given too much homework to complete in time, and that is okay.

You should also update grade in `HW_Sched`, which is your total grade based on the sum of the weights of the homework you completed. Completing all but one homework with a weight of 10 and no late penalties should yield a grade of 90%.

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 the assignments will be ordered.

Given two assignments A1 and A2, the method `compare` should output:

0, if the two items are equivalent
1, if A1 should appear after A1 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.

You will solve this problem by designing and implementing a greedy algorithm, which will greedily optimize weight per hour. This greediness will constitute the code of your algorithm, but you will also need to think about edge cases and deal with them in order to minimize the penalties you may get. *Note: an obvious example of "edge case" is a situation where you are choosing between two tasks that have the same weight per hour.*

Submit your three Java files on codePost.