Comp 251: Assignment 3

Instructor: Jérôme Waldispühl

Due on December 1st at 11:55:00 pm

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, on the same course page as for previous assignments.

- 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 December 1^{st} at 11h55:00 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.
- **You can submit either a zip file or individual files on codePost.** 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.

1. (50 points) **Ford-Fulkerson**
   We will implement the Ford-Fulkerson algorithm to calculate the Maximum Flow of a directed weighted graph. Here, you will use the files WGraph.java and FordFulkerson.java, which are available on the course website. Your role will be to complete two methods in the template FordFulkerson.java.
   The file `WGraph.java` is similar to the file that you used in your previous assignment to build graphs. The only differences are the addition of setter and getter methods for the Edges and the addition of the parameters "source" and "destination". There is also an additional constructor that will allow the creation of a graph cloning a WGraph object. Graphs are encoded using a similar format than the one used in the previous assignment. The only difference is that now the first line corresponds to two integers, separated by one space, that represent the "source" and the "destination" nodes. An example of such file can be found on the course website in the file `ff2.txt`. These files will be used as an input in the program `FordFulkerson.java` to initialize the graphs. This graph corresponds to the same graph depicted in [CLRS2009] page 727.
   Your task will be to complete the two static methods `(fordfulkerson WGraph graph)` and `pathDFS(Integer source, Integer destination, WGraph graph)`. The second method `pathDFS` finds a path via Depth First Search (DFS) between the nodes "source" and "destination" in the "graph". You must return an ArrayList of Integers with the list of unique nodes belonging to the path found by the DFS. The first element in the list must correspond to the "source" node, the second element in the list must be the second node in the path, and so on until the last element (i.e., the "destination" node) is stored. If the path does not exist, return an empty path. The method `fordfulkerson` must compute an integer corresponding to the max flow of the "graph", as well as the graph encoding the assignment associated with this max flow.
   Once completed, compile all the java files and run the command line `java FordFulkerson ff2.txt`. Your program will output a String containing the relevant information. An example of the expected output is available in the file `ff2testout.txt`. This output keeps the same format than the file used to build the graph; the only difference is that the first line now represents the maximum flow (instead of the "source" and "destination" nodes). The other lines represent the same graph with the weights updated to the values that allow the maximum flow. The file `ff226testout.txt` represents the answer to the example showed in [CLRS2009] Page 727. You are invited to run other examples of your own to verify that your program is correct.

2. (50 points) **Bellman-Ford**
   We want to implement the Bellman-Ford algorithm for finding the shortest path in a graph where edges can have negative weights. Once again, you will use the object `WGraph`. Your task is to complete the method `BellmanFord(WGraph g, int source)` and `shortestPath(int destination)` in the file `BellmanFord.java`.
   The method `BellmanFord` takes an object `WGraph` named `g` as an input and an integer that indicates the source of the paths. If the input graph `g` contains a negative cycle, then the method should throw an exception (see template). Otherwise, it will return

an object `BellmanFord` that contains the shortest path estimates (the private array of integers `distances`), and for each node, its predecessor in the shortest path from the source (the private array of integers `predecessors`).
The method `shortestPath` will return the list of nodes as an array of integers along the shortest path from the source to the destination. If this path does not exists, the method should throw an exception (see template).
An example input is available in `bf1.txt`.

3. (0 points) **Knapsack Problem**
We have seen in class the Knapsack problem and a dynamic programming algorithm. One could define the Knapsack problem as following:

Definition. Let $n > 0$ be the number of distinct items and $W > 0$ be the knapsack capacity. For each item i, $w_i > 0$ denotes the item weight and $v_i > 0$ denotes its value. The goal is to maximize the total value

$$\sum_{i=1}^{n} v_i x_i$$

while

$$\sum_{i=1}^{n} w_i x_i \leq W$$

where $x_i \in \{0, 1\}$ for $i \in \{1, \ldots, n\}$.

Algorithm. We recall the recursive form of the dynamic programming algorithm. Let $OPT(i, w)$ be the maximum profit subset of items $1, \ldots, i$ with weight limit w. If OPT does not select the item i, then OPT selects among items $\{1, \ldots, i-1\}$ with weight limit w. Otherwise, OPT selects among items $\{1, \ldots, i-1\}$ with weight limit $w - w_i$. We could formalize as

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{OPT(i-1, w), v_i + OPT(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

Correctness of dynamic programming algorithm (0)
Usually, a dynamic programming algorithm can be seen as a recursion and proof by induction is one of the easiest way to show its correctness. The structure of a proof by strong induction **for one variable**, say n, contains three parts. First, we define the **Proposition** $P(n)$ that we want to prove for the variable n. Next, we show that the proposition holds for **Base case(s)**, such as $n = 0, 1, \ldots$ etc. Finally, in the **Inductive step**, we assume that $P(n)$ holds for any value of n strictly smaller than n', then we prove that $P(n')$ also holds.

Use the proof by strong induction **properly** to show that the algorithm of the Knapsack problem above is correct.

Bounded Knapsack Problem (0)
Let us consider a similar problem, in which each item i has $c_i > 0$ copies (c_i is an integer). Thus, x_i is no longer a binary value, but a non-negative integer at most equal to c_i, $0 \leq x_i \leq c_i$. Modify the dynamic programming algorithm seen at class for this problem.
Note: One could consider a new set, in which item i has c_i occurrences. Then, the algorithm seen as class can be applied. However, this could be costly since c_i might be large. Therefore, the algorithm you propose should be different than this one.

Unbounded Knapsack Problem (0-bonus)
In this question, we consider the case where the quantity of each item is unlimited. Thus, x_i could be any non-negative integer. Provide a dynamic programming algorithm for this problem.