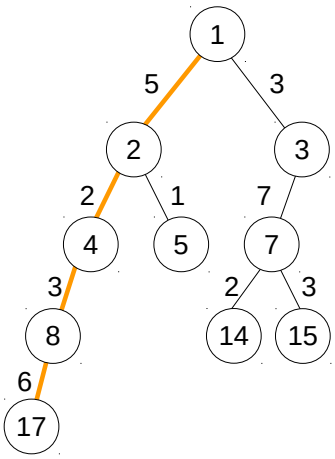
CS3460: Data Structures Fall 2022
Final Exam
Due Date: Wednesday, December 07; 23:00 hrs
Total Points: 20

Honor Code: The following document is an individual exam. As such, this exam should be solved individually, without any discussion, exchange of electronic or hard documents, or any other malpractice that constitutes to cheating in an academic institution. Even a pleasant conversation that involves revealing how much of the test one completes constitutes a violation. If you are unsure what constitutes plagiarism, please refer to the following guidelines set forth by the Appalachian State University `http://academicintegrity.appstate.edu/`, or consult with your instructor. By continuing to participate in this exam, you implicitly agree to uphold the honor code. Failure to uphold the honor code will result in an **F grade** for the entire course.

Please write well-tested Java programs to solve the following problems. You are allowed to use any code that is posted on ASULearn, or code that you have written for any of the homework problems. You may also use the Internet as a resource for general information, for example, in figuring out the syntax and the semantics of the Java programming language, or obtaining general descriptions of the data structures studied in class. However, you are not allowed to search for specific solutions in the Internet, which includes code forums like GeeksForGeeks, StackOverflow, and other such online platforms.

For any of the problems, you are allowed to create multiple files that cater to your implementation, but please submit all the necessary Java code for a particular solution, so that they can be run simply by compiling and executing in a Java environment.

Good Luck!



1. **Fun Skiing (10 points):** Marty McClearen is going on a skiing adventure from a resort in Black Mountain. The skiing range consists of a set of junctions, which can be seen as checkpoints along skiing paths. These ski-paths are defined as a binary tree where the root junction (or node) is the skiing resort, and all ski-paths go down the hill. At each junction, there is a possibility to go on two different ski-paths, the one to the left or on the right. A sample skiing range is given in the figure above.

   We will assume that node 1 will represent the ski resort, and is always the root. Marty has methodically calculated how much fun it is to ski along each of these ski-paths. These can be seen as weights along the edges connecting two junctions. For example, in the figure above, the ski-path between nodes 1 and 2 take 5 units of funness, and the ski-path between nodes 7 and 14 take 2 units of funness.

   Marty has paid for only one skiing journey – so he can start at the root of the tree, and and ski along a single root to leaf path in the tree. He would like to figure out the most fun path, which is a root to leaf path with maximum total funness. This is indicated by the orange edges in the graph. If there are multiple such paths, you may print any one of them.

   Input and Output: In a binary tree with n nodes, there are exactly $n-1$ edges (this can be proven using induction). So the $n-1$ edges are represented using three numbers, the label of the node that is the parent, the label of the node that is the child (either left or right), and the weight of the edge. The input that represents the figure is shown below, and in file `sampleinput.in1`.

   `Sample Input (see sampleinput.in1):`

```
1 2 5
1 3 3
2 4 2
2 5 1
3 7 7
4 8 3
7 14 2
7 15 3
8 17 6
```

   As output, please print the value of the largest funness of a root to leaf path in the tree on the first line, then print the path itself from the root down to the leaf, one node per line. See sample output in file `sampleinput.out1`.

   `Sample Output (see sampleinput.out1):`

```
Largest Funness: 16
Node: 1
Node: 2
Node: 4
Node: 8
Node: 17
```

Startup Code: To help you get started with the program, you are given some startup code. The file `BinaryNode.java` provides a node object in this tree. It contains a `label` that represents the label on the node. The array `childs` contains only two elements, the first of which is a reference to the left child, while the second is a reference to the right child. Similarly, the array `weights` contains two elements, the weights (the funness values) of the left and right ski-paths. There are several accessors and mutators that get and set the different fields of this class. For example, calling `getLeftChild` will return a reference to the left child. Please see this class for all the available methods that you can call. The `toString` method can be used to print an object of this class. You are allowed to modify this class by adding other fields or methods that cater to your implementation, but please do not change existing code, as they are needed by your program.

The class `BinaryTree.java` allows you to accept the input and build this tree. Please do not modify this file, and there is no need to understand how this is done. You will be writing your code in `FunSki.java`, which already builds this tree from input, and gives you a reference to the root node of this tree. You only need to write code to find the largest funness path in this tree. You may execute your program as

```
$ java FunSki < sampleinput.in1
```

Please make sure that the output format matches the sample output. We will be testing with different test cases.

2. **Forming Teams (15 points):** In a physical education class, for performing a rigorous endurance exercise, the instructor of the class tries to divide the students into two groups. The instructor has already compiled a list of students in class, and noted down the interactions among students. Specifically, she has created a social network where each node in the network corresponds to a student, and an edge between two students corresponds to friendships between the students. In order to improve camaraderie, she decides to break the teams in such a way that no friend of a student are in the same team. That is, if student x is picked to be in team 1, then all of x's friends will be in team 2, and their friends will be in team 1, and so on. Of course, if the network has a three cycle (a cycle of three friends, or any odd cycle), then such a division is impossible. Please write a program, that given a social network, informs whether it is possible to divide the students into two teams as above. If so, please print for each student, which team they will belong to.

   Input and Output:

   The first input line has two integers n and m, where n is the number of students, and m is the number of friendships between students. The students are numbered $1, 2, \ldots, n$. Then, there are m lines describing the friendships. Each line has two integers a and b, where students a and b are friends. Every friendship is between two different students. You can assume that there is at most one friendship between any two students.

   As output, please print an example of how to build the teams. For each student, print "1" or "2" depending on to which team the student will be assigned. You can print any valid team assignment. That is, your program will be judged as correct as long as some individual x is on some team (say Team 1), and all of their friends are in the other team (say Team 2). But note that x could also be on Team 2, while their friends are in the other team. So as long as your output is consistent with the specifications, your program will be judged as correct.

Please print the team assignments on a single line, all elements space separated. If there are no solutions, print "impossible". A sample input and output is shown below, and is also in files `forming-teams.in1` and `forming-teams.out1` respectively.

`Sample Input (see forming-teams.in1):`

```
10 13
1 2
1 4
2 3
3 5
3 6
3 7
1 5
1 7
2 10
4 10
4 8
4 9
9 2
```

`Sample Output (see forming-teams.out1):`

```
1 2 1 2 2 2 2 1 1 1
```

Some startup code is provided in `FormingTeams.java`. This program reads the input file and stores the network in an adjacency list representation, which is an array of a list of neighbors for each node. Nodes are labeled 1 to n. We use `ArrayList` objects to store the neighbors for each node, as this already gives us an expanding array. So the object `adjacency_list[x]` stores the neighbors of node `x`. You may use the Java documentation for `ArrayList`, at `https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html`.

To iterate through the `adjacency_list` and print its neighbors, we can use a `for` loop like;

```
for (int x: adjacency_list) {
  for (Integer y: adjacency_list[x]) { // this loops through the neighbor of x
    int z = y.intValue(); // converts the Integer object to int
    // z a neighbor of x.
    ...
  }
}
```

For full credit, your program should work for $n \leq 10^5$ and $m \leq 2 \cdot 10^5$. As a hint, modify depth first search to assign nodes as either in team 1 or 2. If you ever assign a node as both team 1 and 2, you will know that there is an odd cycle, and it is impossible to form teams.

Submission: Please submit all the files as a single zip archive `final.zip` through ASULearn. The zip archive should only contain the individual files of the assignment, and these files should not be

inside folders. Moreover, please do not include other extraneous files. Only include all files that belong to your solution.

Input/ Output Instructions: For all programs, until and otherwise stated, we will be taking the input from standard input (`System.in`) and will be sending the output to standard output (`System.out`). Since we will be using an automatic grading system, please make sure that your output format exactly matches the description above. So make sure that there are no extraneous output in terms of spaces, newlines and other characters.

Notes on Coding: Please do not include user-defined packages in your code. Your code should run in the Unix/Linux machine using the commands `javac` and `java`.

You may not use any `Java Collection` objects for Problem #1, but can use `ArrayList` for Problem #2. Please use the appropriate Java documentation for more information.