CMSC 420: Spring 2021

Programming Assignment 0: Tour and Locator

Handed out: Tue, Feb 2. Due: **Sun, Feb 14 (11:00pm)**. (Fair warning: Don't wait until too late, since TA support will be limited over the weekends.) See the course syllabus for the late policy. Will be discussed in class on Tue, Feb 2.

Overview: Our programming project this semester will involve implementing algorithms for efficiently computing transportation tours for a set of points in 2-dimensional space. A *tour* is a cycle that visits all the points of some set. Computing efficient tours is fundamental to many transportation applications. This assignment will be a short warm-up exercise, which is designed to (re)familiarize yourself with Java programming, and to gain practice with the submission and grading process. You will not need to implement any fancy data structures.

Tours: Let $P = \{p_1, \ldots, p_n\}$ be a set of n points in two-dimensional space $\mathbb{R}^2$ (see Fig. 1(a)). A *tour* is defined to be a cycle that visits each point of P exactly once (see Fig. 1(b)). The most famous example is the *travelling salesperson problem* (TSP), a famous NP-hard problem that involves computing the tour of minimum length (see Fig. 1(c)).

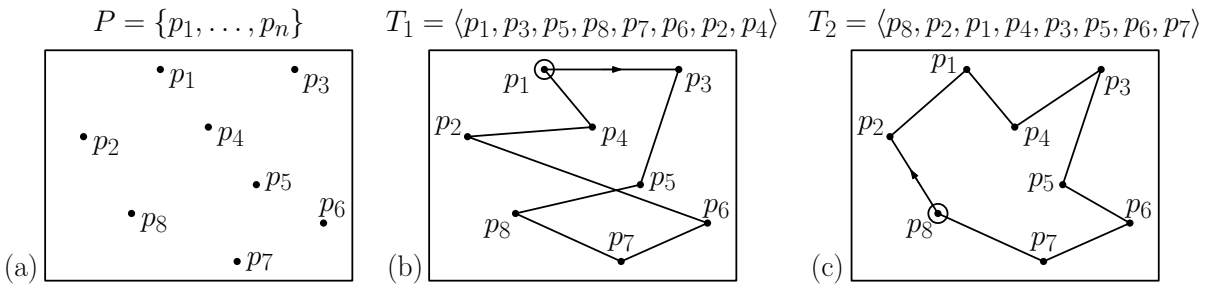

Figure 1: (a) A point set P, (b) a tour of P starting at p_1, and (c) another tour starting at p_8.

Representing a tour: Given a set of points P, we can represent a tour simply as a list (or more conveniently in Java as an `ArrayList`) of points. The associated tour is defined to be the cycle defined by visiting each point of the list in order, returning finally from the last point to the first.

Modifying through reversals: One way to modify any tour is by reversing an arbitrary sublist. For example, consider the tour shown in Fig. 2(a). Let us assume that the points of the tour are indexed from 0 to $n-1$, and let i and j be any two indices, where $0 \leq i < j \leq n-1$. We can form a new tour by reversing the sublist running from indices i through j (see Fig. 2(b)). This has the effect of replacing two edges $(i-1, i)$ and $(j, j+1)$ with the edges $(i-1, j)$ and $(i, j+1)$, and reversing all the edges in between.

Tour Object: In this assignment, you will implement a simple data structure, called `Tour`, that will maintain a tour for a set of points. Among other things, it will support sublist reversals as shown in Fig. 2. It will support other operations, such as the ability to add new points to tour, and various methods for listing out the points in the current tour.

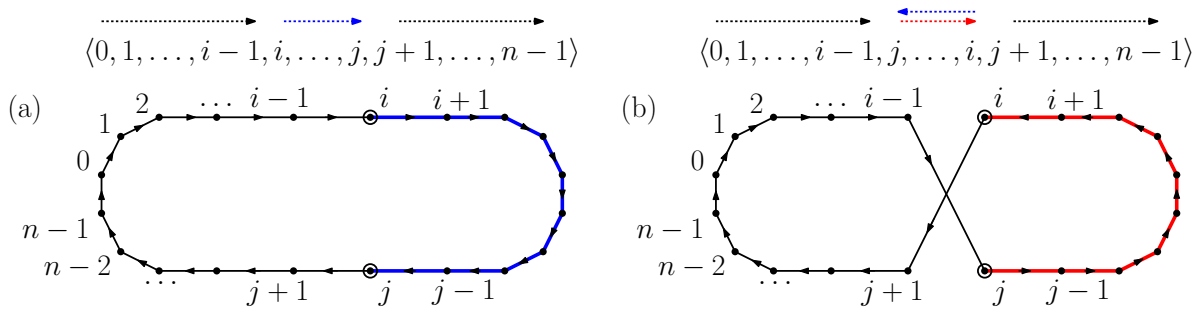

Figure 2: (a) A tour and (b) result of reversing the subtour from i to j.

Labeled Points: In order to refer to points in our data structure, we will associate each one with a string label. For example, if the points are airports, it will be convenient to identify each by its 3-letter IATA code. For example, Los Angeles International airport is given by the code "`LAX`" and Dulles International Aiport by "`IAD`". Its coordinates are given by its longitude and latitude. Each airport will also be associated with other information (its name, city, country, latitude and longitude), but we will not be using these for this assignment.

Of course, it would be too restrictive to build a data structure that works only on airport objects. The `Tour` class will be designed so that it can be applied to any generic type, called a *labeled point*. Such an object is defined by its (x, y) coordinates (of type `float`) and its *label* (of type `String`). A labeled point is any Java class that implements the following Java interface:

```
public interface LabeledPoint2D {
    public float getX(); // get point's x-coordinate
    public float getY(); // get point's y-coordinate
    public String getLabel(); // get the label
    // ... (and a few other methods, which we won't worry about now)
}
```

The `Tour` class (which you will implement) will be a generic Java object based on a type `Point`. This type can be any Java object that implements the `LabeledPoint2D` interface. In particular, our `Airport` class (which we will provide you) does this.

```
Airport.java: (We will provide this)
    public class Airport implements LabeledPoint2D { // An Aiport is a labeled point
        // ...
    }

Tour.java: (You will fill in the details here)
    public class Tour<Point extends LabeledPoint2D> { // A Tour stores labeled points
        // ... (You will fill in the rest of this)
    }

SomeApplication.java: (We will also provide this)
    ...
    Tour<Airport> theTour; // This stores a tour of Airports
```

I'm really confused! This is a lot to take in, so don't worry too much if this is a bit confusing. The program you need to write is actually pretty short. We will provide you with a skeleton implementation containing all of the above. All that you will need to do is fill in one file, `Tour.java`, which implements all the operations we ask of you. We will even provide the function declarations, and you just need to fill in their contents.

Public Interface: Here is a formal definition of the public interface of the `Tour<Point>` class:

`Tour():` This constructor performs whatever initializations are needed to create an empty tour. For our purposes, a tour can be represented as an expandable array, say a Java array-list, containing objects of type `Point`, that is, `ArrayList<Point>`. Thus, your constructor might create a new (empty) array-list object.

`String append(Point pt):` This appends a labeled point `pt` to the end of the current tour (e.g., by appending it to the aforementioned array-list).

It returns a string that summarizes the result of the operation. This string starts with the prefix `"append(XXX):␣"`, where "`XXX`" is the label associated with the point and "␣" denotes a single space. If the tour already contains a point with this same label, this is followed by the string `"Error - Label exists (operation ignored)"`. Otherwise, it adds this point to the end of the current tour. Letting i denote the index where the new point is placed, the prefix is followed by the string `"Added to tour at index` i`"`. (In standard Java style, we assume that indexing starts at zero.) An example is shown in Table 1. (Input lines have been shortened.)

Table 1: Example of commands and output strings.

Input:	**Output:**
`append:IAD: ...`	`append(IAD): Added to tour at index 0`
`append:BWI: ...`	`append(BWI): Added to tour at index 1`
`append:LAX: ...`	`append(LAX): Added to tour at index 2`
`append:IAD: ...`	`append(IAD): Error - Label exists (operation ignored)`
`list-tour`	`list-tour: 0:IAD 1:BWI 2:LAX`
`list-labels`	`list-labels: BWI:1 IAD:0 LAX:2`
`index-of:LAX`	`index-of(LAX): 2`

`String listTour():` This operation returns a string containing all the labels of the points in tour order. Each label is preceded with the index of this point in the tour. The output string starts with the prefix `"list-tour:␣"`, and it followed with a blank-separated sequence of the form "i`:XXX`", where i is the index (ranging from 0 up to $n-1$), and "`XXX`" is the label associated with this point. An example is given above.

`String listLabels():` This operation returns a string containing all the labels of the points in the tour in alphabetical order of the labels. Each label is succeeded with the index of this point in the tour. The output string starts with the prefix `"list-labels:␣"`, and it followed with a blank-separated sequence of the form "`XXX:`i", where "`XXX`" is the label associated with this point, and i is its index in the tour. An example is given above.

`String indexOf(String label):` This operation finds the point with the given label and returns its index in the tour. The output string starts with the prefix `"index-of(XXX):␣"`,

where "XXX" is the given label. If no point with the given label appears in the tour, this is followed with the string "Not-found". Otherwise, it is followed with the index of the point in the tour.

`String reverse(String label1, String label2)`: This operation reverses the subtour between the two given labels. The output starts with the prefix "reverse(XXX,YYY):␣", where "XXX" is the first label and "YYY" is the second label. Assuming that there are two distinct points in the tour with these labels, let i and j denote their indices in the tour. Swap i and j if needed so that $i < j$. Then reverse the order of points in the sublist from i to j, as shown in Fig. 2. Following the prefix, output "Successfully reversed subtour of length k", where k is the number of points in the sublist that was reversed. Some examples are shown in Table 2.

There are a few error cases to consider, which are processed in the following order:

- If no point of the tour has label XXX, then the prefix is followed by "Error - Label XXX does not exist (operation ignored)"
- If no point of the tour has label YYY, then the prefix is followed by "Error - Label YYY does not exist (operation ignored)"
- If XXX and YYY are the same, then the prefix is followed by "Error - Labels are equal (operation ignored)"

Table 2: Example of reverse operations.

Input:	**Output:**
list-tour	list-tour: 0:IAD 1:BWI 2:LAX 3:DCA 4:JFK 5:ATL 6:SFO
reverse:BWI:ATL	reverse(BWI,ATL): Successfully reversed subtour of length 5
list-tour	list-tour: 0:IAD 1:ATL 2:JFK 3:DCA 4:LAX 5:BWI 6:SFO
reverse:IAD:IAD	reverse(IAD,IAD): Error - Labels are equal (operation ignored)
reverse:LAX:CDG	reverse(LAX,CDG): Error - Label CDG does not exist (operation ignored)
reverse:DFW:CDG	reverse(DFW,CDG): Error - Label DFW does not exist (operation ignored)
reverse:DFW:DFW	reverse(DFW,DFW): Error - Label DFW does not exist (operation ignored)

Locators: This completes the description of the input/output behavior of the program. There is, however, an issue related to the program's efficiency. Consider the operation `indexOf` described above. Based on our description so far, this operation would take worst-case time $O(n)$ to implement on a tour of length n, since it would involve searching through the entire tour to find the point with the given label. We would like to do better.

A common issue arising in data structure design is that we insert an object in the data structure at one time and later we wish to locate where this object appears our the data structure. We would like the `indexOf` operator to run in at most $O(\log n)$ time. Our suggestion on how to achieve this is to employ a Java `TreeMap` to store a collection of key-value pairs, where the key is the point's label and the value is the point's index in the tour. For example, given the tour from Table 1, this map would store the pairs $\{(\texttt{BWI}, 1), (\texttt{IAD}, 0)(\texttt{LAX}, 2)\}$. Now, to perform the operation `indexOf`, we can search the `TreeMap` for the given label to retrieve the associated index in $O(\log n)$ time.

Note that whenever the index of a point changes, as can happen during a `reverse` operation, you will need to look up the point in the `TreeMap`, and modify its associated value based on the point's new index.

Running-time Requirements: Our grading of your program will involve an inspection of your code for the manner in which you implement the above operations. Assuming that the current tour contains n points, we require that the above operations run in the following worst-case asymptotic times:

`append:` $O(\log n)$ (amortized) time, assuming you can add an item to Java's `ArrayList.add()` in $O(1)$ amortized time and can add an entry to a tree map in $O(\log n)$ time.

`listTour:` $O(n)$ time.

`listLabels:` $O(n)$ time, assuming Java's `TreeMap.entrySet()` function returns the entries sorted by key value in this time.

`indexOf:` $O(\log n)$ time, assuming this is the search time of Java's `TreeMap`.

`reverse:` $O(k \log n)$ time, where k is the length of the tour being reversed. It should take only $O(k)$ time to perform the reversal, but k locator values in the `TreeMap` need to be updated.

If you prefer, you can use a `HashMap` instead of a `TreeMap`. Note that the run time of `listLabels` will go up to $O(n \log n)$, since the strings will need to be sorted. This is acceptable for full credit.

Program structure: We will provide a driver program that will input a set of commands. You need only implement the `Tour` class and the functions listed above. Here is the public interface:

```
package cmsc420_s21;

public class Tour<Point extends LabeledPoint2D> {
    public Tour() { }                           // Constructor
    public String append(Point pt) { ... }      // Append point to tour
    public String listTour() { ... }            // List in tour order
    public String listLabels() { ... }          // List in alpha order
    public String indexOf(String label) { ... } // Index of label
    public String reverse(String label1, String label2) { ... } // Reverse
}
```

Skeleton Code: We will provide you with some skeleton code to start with. This consists of the following:

`Tour.java:` **This is the only file you need modify.** A skeletal version of the main class for the extended binary search tree.

`Airport.java:` A class that stores information about airports.

`LabeledPoint2D.java:` The interface for the labeled point type.

`Point2D.java:` A small utility class for storing (x, y) coordinates.

`Tester.java`: Main program for testing your implementation. It inputs commands either from a file or standard input and sends output to another file or standard output. **(You may modify this file to select different input/output files.)**

`CommandHandler.java`: A class that processes commands that are read from the input file and produces the appropriate function calls to the member functions of your `Tour` class.

You may submit additional files as well, but it is not necessary. Other than `Tour.java` avoid modifying or reusing any of the above files, since we will overwrite them with our own when testing your program. Use the package "`cmsc420_s21`" for all your source files.

Testing/Grading: We will be using Gradescope's autograder and JUnit for testing and grading your submissions. All the tests and the expected results are visible. We will provide a link to the final test data on the class Projects page. We will check style and efficiency manually, and this will constitute 20% of the final score.

Submission Instructions:

Submissions will be made through Gradescope. There is no limit to the number of submissions you can make. The last submission will be graded. Here is what to do:

- Log into the CMSC420 page on Gradescope, select this assignment, and select "`Submit`". A window will pop up (see Fig. 3). Drag your file `Tour.java` into the window. If you generated other files, zip them up and submit them all. (You do not need to include the files from the skeleton code, included `Airport.java`, `LabeledPoint2D.java`, `Point2D.java`, `Tester.java`, and `CommandHandler.java`.) Select "`Upload`".

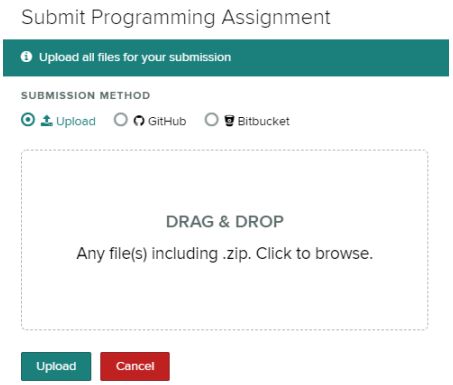

Figure 3: Gradescope submission.

After a few minutes, Gradescope will display the results (see Fig. 4). In this case, 20 out of 25 points are determined by the autograder, and we will assign the final 5 points based on inspecting the source code of your program for style and efficiency.

On the top-right of the page, it shows the scores of the individual tests as generated by the autograder. (If there are compilation errors, these will be displayed on this page.) The center of the window shows a line-by-line summary, with the output generated by your program on the left and the expected output on the right. If there are mismatches,

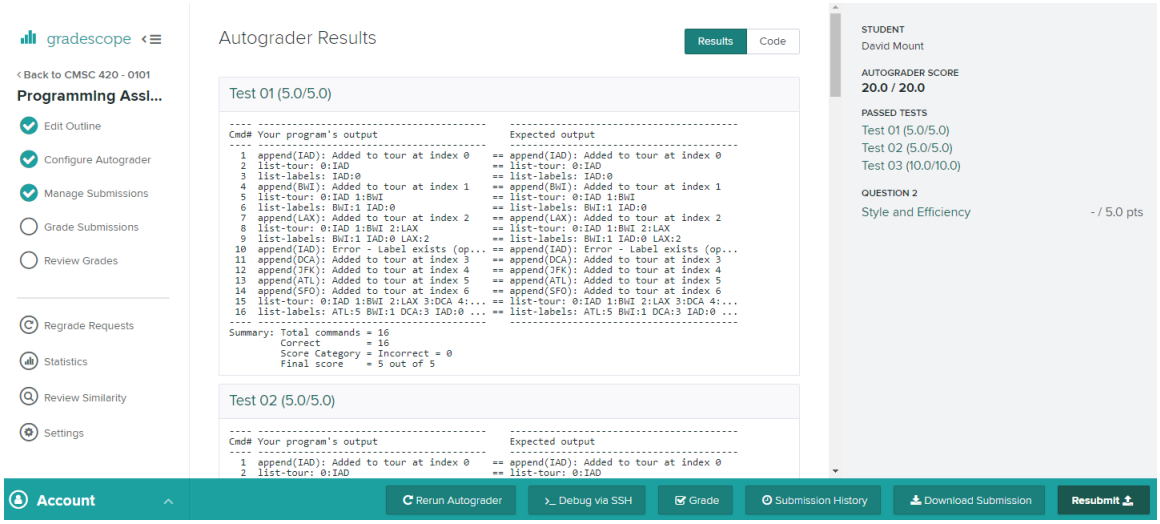

Figure 4: Gradescope autograder results (correct).

these will be highlighted (see Fig. 5). The final score is based on the number of commands for which your program's output differs from ours. Note that the comparison program is very primitive. It compares line by line (without considering the possibility of inserted or deleted lines) and is sensitive to changes in case and the addition of white-space.

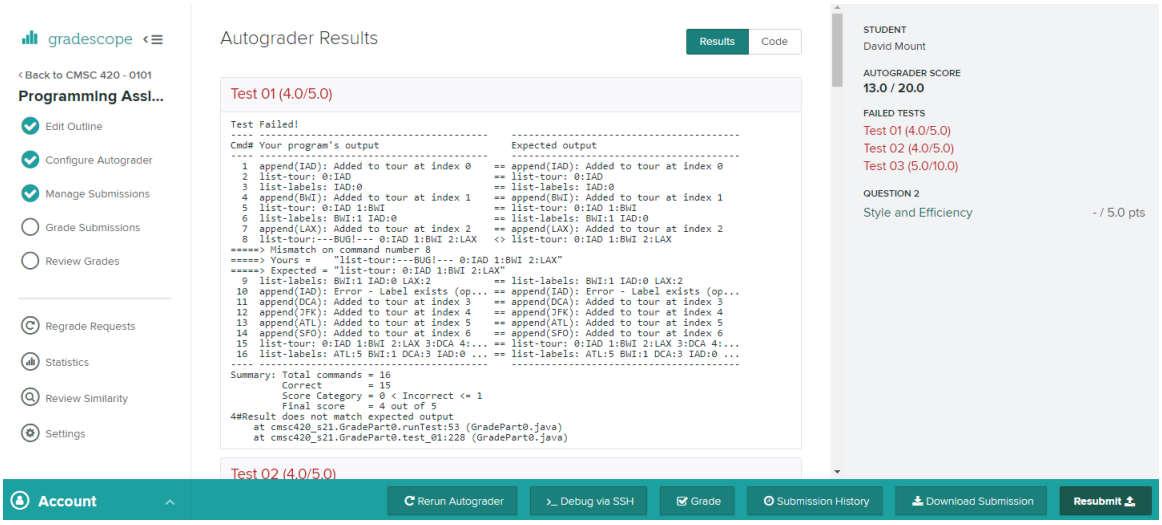

Figure 5: Gradescope autograder results (incorrect).