# ***Goal***

Write a Java program using a circularly linked list to estimate the solution to the *Traveling Salesperson Problem (TSP)*. You will:

* *Learn about the notorious traveling salesperson problem.*
* *Learn to implement linked lists.*
* *Gain more practice with data types.*

*Note: In this document, “linked list” always refers to a “circularly linked list”.*

# ***Background***

**What is the Traveling Salesperson Problem?**

Given *n* points, the goal of the traveling salesperson problem is to find the shortest path that visits every point once and returns to the starting point. We call an ordering of *n* points a ***tour***.

Consider the two four-point tours shown below. Both tours visit the same set of points; however they take different paths. The tour on the left uses the ordering A → B → C → D → A, while the tour on the right uses A → B → D → C → A. A traveling salesperson would prefer the tour on the left, as it minimizes the distance they would have to travel[[1]](#footnote-0).

| *Length: 12 units* | *Length: 14.48 units* |
| --- | --- |

**Greedy Heuristics**

You will implement two greedy heuristics to find good, but not optimal, solutions to the traveling salesperson problem. We ask you to use greedy heuristics to estimate the solution because TSP is a notoriously difficult combinatorial optimization problem[[2]](#footnote-1). In principle, you can enumerate all possible tours (*n* factorial) and pick the shortest one; in practice, the number of possible tours is so staggeringly large that this approach is inapplicable. No one by far knows of an efficient method that can find the shortest possible tour for large *n*. Instead many methods have been studied that seem to work well in practice, even though they do not guarantee to produce the best possible tour. Such methods are called *heuristics*.

# ***Implementation Tasks***

**Representing Points**

We provide a Point data type in the project folder (for reference, you can also find it [here](https://www.cs.princeton.edu/courses/archive/spring20/cos126/assignments/tsp/files/Point.java))*.* Use this data type to represent a single (x, y) coordinate.

| public class **Point** {  *// creates the point (x, y)*  public **Point(double x, double y)**  *// returns the Euclidean distance between the two points*  public double **distanceTo(Point that)**  *// draws this point to standard drawing*  public void **draw()**  *// draws the line segment between the two points*  public void **drawTo(Point that)**  *// returns a string representation of this point*  public String **toString()**  } |
| --- |

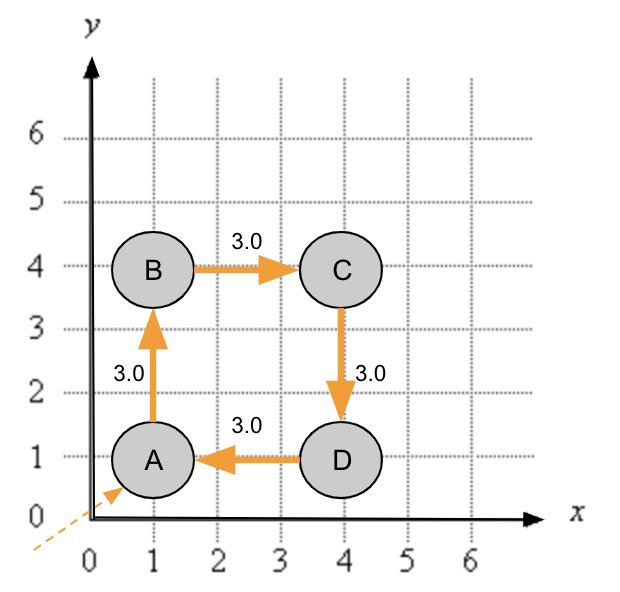
You will create a Tour data type that represents the sequence of points visited in a TSP tour. Represent the tour as a *circularly linked list* of nodes, with one node for each point in the tour. Recall that in a circularly linked list, the last node contains a pointer back to the first node. (Read Section 4.3. Pay particular attention to the parts that deal with linked lists.)

**Calculating heuristics**

Your main task is to implement the **nearest neighbor** and **smallest increase** insertion heuristics for building a tour incrementally, one point at a time. Start with a one-point tour (from the first point back to itself), and iterate the following process until there are no points left:

* *Nearest neighbor heuristic:* Read in the next point, and add it to the current tour *after* the point to which it is closest. If there is more than one point to which it is closest, insert it after the first such point you discover.
* *Smallest increase heuristic:* Read in the next point, and add it to the current tour *after* the point where it results in the least possible increase in the tour length. If there is more than one point, insert it after the first such point you discover.

For example, suppose our current tour contains the following four points that are stored in a circularly linked list as A → B → C → D → A.



*Current Tour: Length 12 units*

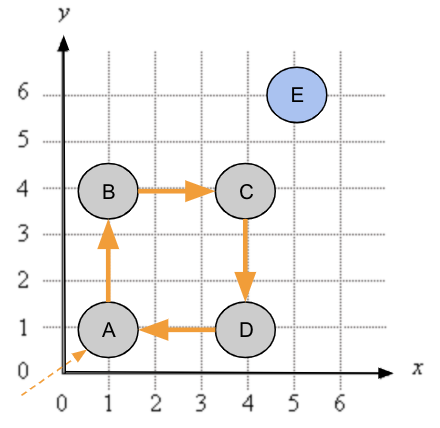
The next point input is E, located at (5, 6). Depending on the heuristic, point E will be incorporated into the existing tour in one of two ways. Nearest neighbor will visit E after point C, as C is the closest existing point to E. Smallest increase will determine that visiting E after point B results in the smallest overall increase in the length of the tour.

| *Nearest Neighbor: Length 16.33 units* | *Smallest Increase: Length 15.70 units* |
| --- | --- |

To find the nearest neighbor to a node, loop through the linked list and maintain a reference to the closest node seen. After checking all the locations, insert the new point into the linked list after the reference node.

To find the smallest increase, your code should loop through your linked list and consider how the tour length is impacted by inserting your current point into every index. Maintain a reference to the best position to insert the node, and insert the current point into the linked list after the reference node.

Here is an example of showing the calculations needed for performing one insert using the Smallest Increase heuestic on a 4-point tour. The blue node E represents the new point to insert.



The figure below shows how you can find the 5-point tour with the smallest increase heuristic. Notice that there will be 4 possible tours with E, depending on where to insert it. All 4 tours create two new pointers (depicted in red) and remove one pointer. To find the tour with the smallest increase, calculate which insertion point minimizes the change in the tour length after the insertion (i.e. delta) and insert your node there.

|  |  |  |  |
| --- | --- | --- | --- |
| Remove: AB **(-3.0)**  Insert: AE **(6.40)**, EB **(4.47)**  **Delta = 7.87** | Remove: BC **(-3.0)**  Insert: BE **(4.47)**, EC **(2.23)**  **Delta = 3.70** | Remove: CD **(-3.0)**  Insert: CE **(2.23)**, ED **(5.10)**  **Delta = 4.33** | Remove: DA **(-3.0)**  Insert: DE **(5.10)**, EA **(6.40)**  **Delta = 8.50** |

***Tour.java***

Create a Tour data type that represents the sequence of points visited in a TSP tour. Represent the tour as a circularly linked list of nodes, one for each point in the tour. Each Node contains two references: one to the associated Point and the other to the next Node in the tour. Each constructor **must** take constant time. All instance methods **must** take time linear (or better) in the number of points currently in the tour.

Within Tour.java, define a nested class Node:

| private class **Node** {  private Point **p**;  private Node **next**;  } |
| --- |

Your Tour data type must implement the following API. You **must not** add public methods to the API; however, you **may** add private instance variables or methods (which are only accessible in the class in which they are declared).

| public class **Tour** {  // creates an empty tour  public ***Tour***()  // creates the 4-point tour a->b->c->d->a (for debugging)  public ***Tour***(Point a, Point b, Point c, Point d)  // returns the number of points in this tour  public int ***size***()    // returns the length of this tour  public double ***length***()    // returns a string representation of this tour  public String ***toString***()    // draws this tour to standard drawing  public void ***draw***()    // inserts p using the nearest neighbor heuristic  public void ***insertNearest***(Point p)  // inserts p using the smallest increase heuristic  public void ***insertSmallest***(Point p)  // tests this class by calling all constructors and instance methods  public static void ***main***(String[] args)  } |
| --- |

***Constructors:*** For the default constructor, initialize the instance variables to represent an *empty* list. In the 4-point constructor, add the given points into the circularly linked list ordered as A → B → C → D → A.

| **Hint!** Use the 4-point example (described above) as a test case in main().   | public static void main(String[] args) {  // define 4 points, corners of a square  Point a = new Point(1.0, 1.0);  Point b = new Point(1.0, 4.0);  Point c = new Point(4.0, 4.0);  Point d = new Point(4.0, 1.0);  // create the tour a -> b -> c -> d -> a  Tour squareTour = new Tour(a, b, c, d);  } | | --- | |
| --- | --- |

***size():*** returns the number of points in the current Tour, which is equivalent to the number of nodes in the linked list. The size of an empty Tour is 0. You can loop through your circularly linked list using a do-while loop:

| **Node** current = start;  **do** {  current = current.next;  } ***~~while~~*** *~~(current.next != start);~~* **while** (current != start); |
| --- |

| **Hint!** To test the size() method call the size() method for the square tour you created in main, which has four points.   | // print the size to standard output  int size = squareTour.size();  StdOut.println("# of points = " + size); | | --- | |
| --- | --- |

***length():*** returns the length of the current Tour, which is equivalent to the distance of the path through all the points in the linked list. Use the distanceTo() method of the point data type to find the distance between successive points. The length of an empty Tour is 0.0.

| **Hint!** Test the length() method on the 4-point tour—the length should be 12.0.   | // print the tour length to standard output  double length = squareTour.length();  StdOut.println("Tour length = " + length); | | --- | |
| --- | --- |

***toString():*** returns a string containing the points, one per line, by calling the toString() method for each point, starting with the first point in the tour. An empty Tour must be represented as an empty String.

| **Hint!** Create a StringBuilder object and append each Point to the StringBuilder object.  You can implicitly call the toString() method for each Point p:   | System.out.println(p) | | --- |   Test the toString() method on the 4-point tour in main():   | // print the tour to standard output  StdOut.println(squareTour); | | --- |   You should get the following output:   | (1.0, 1.0)  (1.0, 4.0)  (4.0, 4.0)  (4.0, 1.0) | | --- | |
| --- | --- | --- | --- |

***draw():*** draws the Tour to standard drawing by calling the drawTo() method for each pair of consecutive points. It must produce no other output. If the Tour is empty, then nothing must be drawn on the standard drawing.

| **Hint!** Remember to set your X and Y scale. For debugging the square tour, add to main():   | StdDraw.setXscale(0, 6);  StdDraw.setYscale(0, 6); | | --- | |
| --- | --- |

***insertNearest():*** insert the given point into the current tour using the *Insert Nearest* heuristic*.*

| **Hint!** To test insertNearest(), try inserting various points into the 4-point tour. Try creating a point E at coordinate (5.0, 6.0), as in the example graphs provided earlier in the documentation. In main, add the following test code and check if the drawing looks as expected.   | Point e = new Point(5.0, 6.0);  squareTour.insertNearest(e);  squareTour.draw(); | | --- | |
| --- | --- |

***insertSmallest():*** insert the given point into the current tour using the *Smallest Neighbor* heuristic.

| **Hint!** Test this method similarly to as you tested the insertNearest method. |
| --- |

## 

## ***Testing***

The input files begin with two integers *width* and *height*, followed by pairs of *x*- and *y*-coordinates. All *x*-coordinates will be real numbers between 0 and *width*; all *y*-coordinates will be real numbers between 0 and *height*.

A good debugging strategy for most programs is to test your code on inputs that you can easily solve by hand. Start with 1- and 2-point problems. Then, do a 4-point problem. Choose the data so that it is easy to work through the code by hand. Draw pictures. If your code does not do exactly what your hand calculations indicate, determine where they differ. Use the StdOut.println() method to trace.

The files for this assignment include many sample inputs. Most are taken from [TSPLIB](http://www.math.uwaterloo.ca/tsp/data/index.html).

For usa13509.txt we get tour lengths of 77449.9794 and 45074.7769 for nearest insertion and smallest insertion, respectively. For circuit1290.txt we get 25029.7905 and 14596.0971, respectively.

After implementing Tour.java, use the client program [NearestInsertion.java](https://www.cs.princeton.edu/courses/archive/spring20/cos126/assignments/tsp/tsp/NearestInsertion.java), which reads the points from standard input; runs the nearest neighbor heuristic; prints the resulting tour, its length, and its number of points to standard output; and draws the resulting tour to standard drawing. [SmallestInsertion.java](https://www.cs.princeton.edu/courses/archive/spring20/cos126/assignments/tsp/tsp/SmallestInsertion.java) is analogous but runs the smallest increase heuristic. For example:

| > java-introcs NearestInsertion < tsp1000.txt  (185.0411, 457.8824)  (198.3921, 464.6812)  (195.8296, 456.6559)  (216.8989, 455.126)  (213.3513, 468.0186)  (241.4387, 467.413)  (259.0682, 473.7961)  (221.5852, 442.8863)  ...  (264.57, 410.328)  Tour length = 27868.7106  Number of points = 1000 | > java-introcs SmallestInsertion < tsp1000.txt  (185.0411, 457.8824)  (195.8296, 456.6559)  (193.0671, 450.2405)  (200.7237, 426.3461)  (200.5698, 422.6481)  (217.4682, 434.3839)  (223.1549, 439.8027)  (221.5852, 442.8863)  ...  (186.8032, 449.9557)  Tour length = 17265.6282  Number of points = 1000 |
| --- | --- |
| 1000 points nearest  Nearest Insertion Tour | 1000 points smallest  Smallest Insertion Tour |

# 

# 

# ***Analysis***

In your readme.txt, estimate the running time (in seconds) of your program as a function of the number of points *n*. You should use the client program TSPTimer.java to help you estimate the running time as a function of the input size *n*. TSPTimer takes a command-line argument n, runs the two heuristics on a random input of size n, and prints how long each took. Run the two heuristics for *n* = 1,000, and repeatedly double *n* until the execution time exceeds 60 seconds.

In order to get consistent timing results:

* Repeat each experiment for *n* more than once.
* Execute with the -Xint flag. The -Xint flag turns off various compiler optimizations, which helps normalize and stabilize the timing data that you collect. For example:

| > java-introcs -Xint TSPTimer 1000  Tour length = 26338.42949015926  Nearest insertion: 0.056 seconds  Tour length = 15505.745750759515  Smallest insertion: 0.154 seconds |
| --- |

# ***Submission***

Submit Tour.java and a completed readme.txt.

# ***Enrichment***

* Here's a [13,509-point problem](https://www.cs.princeton.edu/courses/archive/spring20/cos126/assignments/tsp/tsp13509.gif) that contains each of the 13,509 cities in the continental US with a population over 500. The [optimal solution](https://www.cs.princeton.edu/courses/archive/spring20/cos126/assignments/tsp/tsp13509-sol.jpg) was discovered in 1998 by Applegate, Bixby, Chvatal, and Cook using theoretical ideas from *linear and integer programming*. The "record" for the largest TSP problem ever solved exactly is a [85,900-point](http://www.math.uwaterloo.ca/tsp/pla85900/index.html) instance that arose from microchip design in the 1980s. It took over 136 CPU-years to solve.
* Here's an excellent and very accessible [book](http://www.amazon.com/Pursuit-Traveling-Salesman-Mathematics-Computation/dp/0691152705) about the TSP.
* Here's a nice pictorial survey of the [history of the TSP problem](http://www.math.uwaterloo.ca/tsp/).
* Some folks even use the TSP to create and sell art. Check out [Bob Bosch's page](http://www.oberlin.edu/math/faculty/bosch/tspart-page.html). You can even [make your own TSP artwork](http://www.oberlin.edu/math/faculty/bosch/making-tspart-page.html).
* You can also implement a better TSP heuristic. You are not required to use the Tour data type or linked lists. We will award a special prize to whoever finds the shortest tour for a mystery tour with about 1,000 points. Here are some ideas.
  + API requirements: Name your program TSP.java. The only public method in TSP.java is main(), which reads a sequence of points from standard input (in the standard format) and prints the resulting tour to standard output, one point per line.
  + Performance requirement: Your solution must solve a 1,000-point instance in at most a few seconds and a 10,000-point tour in at most a minute.

| *Copyright © 2000–2020, Robert Sedgewick and Kevin Wayne, All Rights Reserved* |
| --- |

1. The importance of the TSP does not arise from an overwhelming demand of salespeople to minimize their travel length, but rather from a wealth of other applications such as vehicle routing, circuit board drilling, VLSI design, robot control, X-ray crystallography, machine scheduling, and computational biology. [↑](#footnote-ref-0)
2. The travelling salesperson problem is in fact *NP-Complete*, as we will learn in class shortly. [↑](#footnote-ref-1)