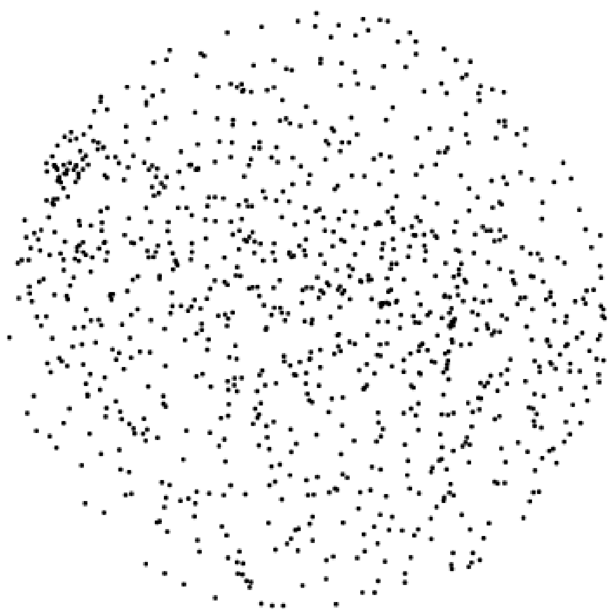
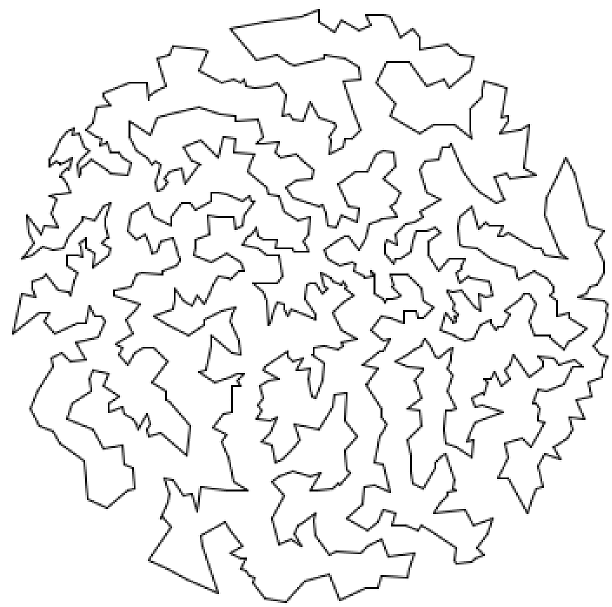
# Traveling Salesperson Problem

This assignment was originally developed by Robert Sedgwick and Kevin Wayne at Princeton University, and subsequently adapted by Baker Franke at the University of Chicago.

**The Traveling Salesperson Problem (TSP)**
Given *N* points in the plane, the goal of a traveling salesperson is to visit all of them (and arrive back home) while keeping the total distance traveled as short as possible. Implement two greedy heuristics to find good (but not optimal) solutions to the *traveling salesperson problem* (TSP).



1,000 points



Optimal Tour

**Perspective.**
The importance of the TSP does not arise from an overwhelming demand of salespeople to minimize their travel distance, 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. The traveling salesperson problem is a notoriously difficult *combinatorial optimization* problem. In principle, one can enumerate all possible tours and pick the shortest one; in practice, the number of tours is so staggeringly large – for N points there are N! tours - it makes this approach basically useless. An algorithm for finding an optimal tour for large N in any reasonable amount of time is currently an unsolved, and open problem in computer science.

**Greedy heuristics.**
However, many methods have been studied that seem to work well in practice, even though they are not guaranteed to produce the best possible tour. Such methods are called *heuristics*. A *greedy heuristic* makes the most optimal choice at each stage of the game with the goal of (hopefully) finding the best overall solution.

Your main task is to implement the *nearest neighbor* and *smallest increase* insertion heuristics for building a tour incrementally. 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.)

### Your Task – The Big Picture

Your task is to complete the **`Tour`** class that represents the sequence of points visited in a TSP tour. Your `Tour` class will be a modification of a Linked List where each node in the list holds a point in the `Tour` (an x,y coordinate). A linear traversal of the List will represent the path the salesperson would travel. NOTE: Since the tour is a cycle, you need to make sure that you consider that the first point comes immediately after the last point in the list when performing such calculations as computing the total distance of the tour. Your class will allow for the adding of one `Point` at a time to the tour, which will insert a node into the List in some fashion.

### Implementation Details

The starting project provides you with the `TourInterface`, a start for an implementation of the `Tour` which includes an inner `ListNode` class that you should use for maintaining the List. The `DrawingPanel` code is also included.

The major work of the assignment is to implement the methods of the `TourInterface` which are briefly described here:

**`Tour Interface` Methods:**

> **`void print()`** //prints the entire tour to standard output
> **`void draw(Graphics G)`** //displays the tour using the given graphics context
> **`double distance()`** //return the total distance of the tour
> **`int size()`** //return the number of points in the tour (i.e. nodes in the list).
> **`void add(Point P)`** //appends P to the end of the list.
> **`void insertNearest(Point P)`** // inserts P into the list using nearest neighbor heuristic
> **`void insertSmallest(Point P)`** // inserts P into the list using the smallestDistance heuristic

### Getting Started

What follows is a suggested plan of attack for doing this assignment in short pieces. There are several ways to go about it. The following is merely a suggestion.

1. The starting project code has things mostly laid out for you. The `main` method runs but doesn't do anything because the `Tour` class is unimplemented at the moment. But your first step is to poke around the starting code to see what's provided and what's not.

2. You'll want to review the inner `ListNode` class that is a part of `Tour` class. Here we are introducing the concept of a private inner class, which is a separate class within a class. It is very similar in concept to a private helper method, which is particularly useful to the class where it is defined, but to nothing else outside the class. A private inner class is instantiated as

a part of the outer class instantiation. In this application, a ListNode is only meaningful within the concept of a Tour of points, and we can create as many ListNode objects as needed to complete a given Tour. Because an inner class is a member of the outer class, the outer class has access to the private instance variables of the private inner class. In other words, there is no need for 'getter' and 'setter' methods for the outer class to access or modify the inner class instance variables. In this case, the Tour class has direct access to the ListNode private instance variables. Note that ListNode has different names for its instance variables, namely data and next.

3. First, declare the instance variables, and complete the constructor of the Tour class. Then, write the size, add and print methods of the Tour class first. The size method just returns the size of the list. The add method should work just like the LinkedQueue add method which appends a node to the end of the list with the given Point as its data value. The print method should simply traverse the list and print out each Point on its own line. Then run the main method in the TSPClientDriver class to see that the five sample points are added to the list and then displayed properly. Here is what your display should look like.

```
Created Tour with 5 points.
Tour has distance -1.0
Here is the tour in order of points visited
(30.0,150.0)
(179.29,150.0)
(79.09,101.2)
(86.45,236.3)
(187.9,91.2)
```

   Make sure this is working correctly before you continue any further.

4. Write the draw method which accepts the Graphics context to use. The two methods from the Graphics class you will probably want to use are

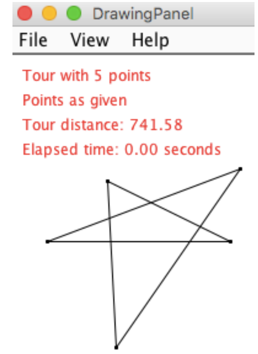
```
fillOval(int leftEdge, int topEdge, int width, int height)
drawLine(int fromX, int fromY, int toX, int toY)
```

   Thus a call to `g.fillOval(x-2, y-2, 5, 5)` will display a circle with a diameter of five pixels centered at x, y.

   Start by simply drawing a circle (an oval with same size width and height) on the graphics for each point in the list (we'll worry about connecting the dots later). Reminder: the getX() and getY() methods of the inherited Point2D class return double values, but graphics coordinates have to be int values, so you will need to cast them as integers to use them. A DrawingPanel has already been constructed in the graphPoints method of the TSPClientDriver class, which then calls your draw method. (Note that when you run the main method, three DrawingPanel windows will automatically be generated and they will all be stacked on top of each other in the center of the screen. They need to be separated in order to see each of them.)

5. Now improve the draw method to draw lines between the points. Don't forget to connect the last point to the first.

6. Finally, begin to write the primary `Tour` methods that we care about. It's suggested you write these methods in the following order:

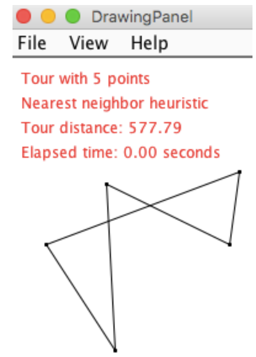



**`distance()`**

Caclulate the total distance of the `Tour`. NOTE: the `Point` class has a method that calculates the distance between itself and another point. Don't forget to count the distance between the last point and the first.

See if you get the same distance for the 5-point tour as shown in the picture at the right. (741.58).

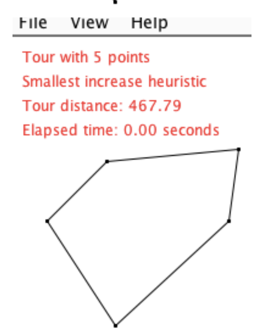
**`insertNearest(Point p)`**




This method will change how you add points to the list. Follow the nearest neighbor heuristic – adding this `Point` into the list immediately after the point it is closest to. Note: since adding one point using this method will take O(N) time, then adding all points in the list will take O($N^2$) time.

See if you get the same distance for the 5-point tour as shown in the picture at the right. (577.79).

**`insertSmallest(Point p)`**



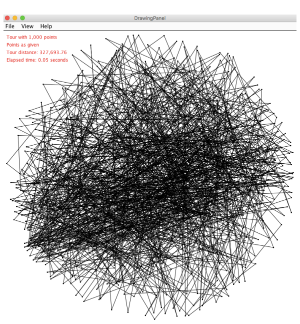

Write the `insertSmallest` method which should follow the smallest increase heuristic – add a point into the list at the location that causes the total distance of the Tour to increase the least.

Careful: When you run `insertSmallest` on a very large data set, say something with more than 10,000 points, it can take a while. If it takes a REALLY long time, you've probably got a O($N^3$) algorithm. If you're careful/clever you can get this down to O($N^2$). Think about it.

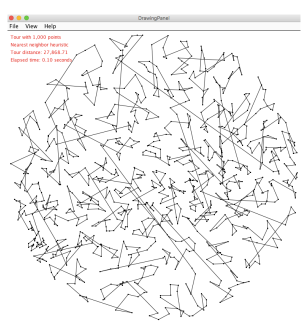
See if you get the same distance for the 5-point tour as shown in the picture at the right. (467.79).

**Testing on larger data sets**

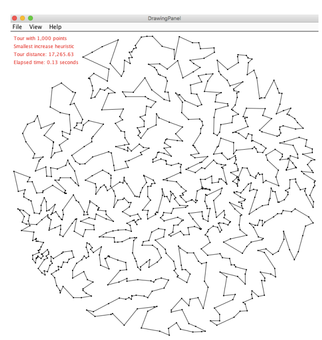
The starting project contains several different data sets (look for the .txt files). By just changing the `file` name being read in the `TSPClientDriver` you can test on larger data sets to see how fast or slow the various heuristics work. Here's what the `tsp1000.txt` file looks like using the various methods (note that your correct distance calculations will match those listed below *exactly*).



`add()`
distance: 327,693.76



`insertNearest()`
distance: 27,868.71



`insertSmallest()`
distance: 17,265.63

**Requirements**

- All five of the `Tour` methods are implemented correctly (`add()`, `draw()`, `distance()`, `insertNearest()`, `insertSmallest()`).
- Each of the above methods have *abundant comments* that fully describe your code.
- All three complete tours are properly displayed in an individual `DrawingPanel`.
- The README file (click on the white lined box in the BlueJ window to open it) contains
    - both your and your partner's name, and date.
    - a complete and thorough description of how both of your algorithms work.
    - the overall complexity (using Big-O analysis) of both algorithms.
- Your `insertSmallest` algorithm should process the `tsp1000.txt` data file in a reasonable amount of time (under 5 seconds). In order to receive full credit, it needs to be an $O(N^2)$ algorithm, which runs in less than 1 second (approximately).

**Turning it in**

- Name your project file directory Traveling Salesperson.
- Zip up the entire project file directory by control-clicking on the directory name and then select Compress "Traveling Salesperson" from the menu.
- Submit the zipped directory using the Schoology assignment page. I only need one submission for both you and your partner (pick one). In other words, only one of you needs to turn this in for both of you to get credit for it (of course both of your names will be listed in the README file).

**Academic Honesty**

- This is a pair-programming project, but not a group project. In submitting your project you are implicitly stating that it is your own creation (you and your partner) and not someone else's work (apart from what is already provided for you).
- Your code should reflect a unique solution to this challenging problem which will be significantly different from anyone else's solution. You can count on your code being evaluated line by line against the work of others, so please do not be tempted to submit even portions of code that is not your own work.
- Submissions that are deemed to be plagiarized from the work of others will receive a zero on this assignment, and cannot be resubmitted for re-evaluation.