COMP 355 - Advanced Algorithms - Spring 2014

Problem Set #3: Divide-and-Conquer Algorithms and Shortest Path Implementation

Issued: 2/19/2014       Due: 3/3/2014 (at beginning of class)
Code for Problem #4 due via Moodle by 2pm (3/3/2014)

Homework Information: Some of the problems are probably too long to attempt the night before the due date, so plan accordingly. No late homework will be accepted. Feel free to work with others, but the work you hand in must be your own. Please list the people with whom you collaborated.

1. (20 points) Given a list \( A = \langle a_1, \ldots, a_n \rangle \) of integers, an *off-parity inversion* is a pair \( a_i \) and \( a_j \) such that \( i < j, a_i > a_j, \) and \( a_i \) and \( a_j \) are of different parities. (In other words, it is an inversion in which one number is even and the other is odd.) See the example below in Figure 1.

![Figure 1: Problem 1: Off-Parity Inversion Counting Example](image1)

Design an \( O(n \log n) \) time algorithm that counts the number of off-parity inversions in a list \( A \) containing \( n \) elements. Justify your algorithm’s correctness and derive its running time.

2. (25 points) You are given a set of \( n \) lines in the plane. Each line is given by a pair of numbers \( (a_i, b_i) \), which represents the line \( y = a_i x + b_i \). That is, \( a_i \) gives the slope of the line and \( b_i \) gives its y-intercept. You are told that \( a_i < 0 \) (all slopes are negative) and \( b_i > 0 \) (all intercepts are positive). You are asked to count the number of intersections that occur in the positive \((x, y)\)-quadrant. (For example, in Figure 2 below, there are six intersections in the first quadrant, shown as black dots.)

![Figure 2](image2)

Of course, this would be easy to do in \( O(n^2) \) time, but I want you to develop an answer for this problem that runs in \( O(n \log n) \) time.

For simplicity, you may assume that the \( a_i \) values are all distinct and the \( b_i \) values are also all distinct. You may also assume that no two lines intersect exactly on the x-axis or the y-axis.
In computer graphics it is often desirable to compute the silhouette of a collection of objects. In this problem, we’ll consider a simple example involving rectangles.

You are given a collection of (possibly overlapping) rectangles that extend upwards from the x-axis. Each rectangle is defined by a triple \((a_i, b_i, h_i)\) where \(a_i\) and \(b_i\) denote the rectangle’s left and right x-coordinates and \(h_i\) denotes the height of the rectangle. The union of these rectangles defines an upper envelope, which consists of a sequence of non-overlapping intervals from left to right along the x-axis. Each interval is associated with a height value of the tallest rectangle that spans this interval.

For example, the input for the rectangles shown in Figure 3 might be as follows. (Note that the rectangles are not given in any particular order.)

\[
(1, 4, 2), (11, 12, 4), (8, 10, 7), (7, 13, 5), (2, 6, 4), (9, 15, 3), (3, 5, 3).
\]

The output consists of seven intervals (including one interval of height 0). Suppose that the output consists of \(m\) intervals. We represent this as an array \(x[1..m+1]\) such that the \(i\)th interval spans \(x[i]\) to \(x[i+1]\), and an array \(t[1..m]\) where \(t[i]\) is the height of the tallest rectangle spanning the \(i\)th interval. The output for the above input would be:

\[
x = <1, 2, 6, 7, 8, 10, 13, 15> \text{ and } t = <2, 4, 0, 5, 7, 5, 3>.
\]

Design an \(O(n \log n)\) algorithm which, given a sequence of \(n\) such triples, computes the upper envelope of these union of the rectangles. Derive the running time of your algorithm. You may assume that the values \(a_i\), \(b_i\), and \(h_i\) are all distinct.
4. (30 points) **Programming Problem**: In anticipation of the upcoming lectures on dynamic programming, implement both the Bellman-Ford single-source shortest path algorithm and the Floyd-Warshall all-pairs shortest paths algorithm discussed in class. You may assume that the input will not have any negative cycles, but a true implementation of these algorithms would include a check to ensure no negative cycles.

I have posted 2 input text files on the course website (tinyEWD.txt and mediumEWD.txt). Included below are example runs using tinyEWD.txt. You need to **turn in your output** for each algorithm when run on the mediumEWD.txt file. You should use 0 as the source vertex for Bellman-Ford. For the Floyd-Warshall algorithm, submit the shortest path distances from vertex 1 to every other vertex as well as the predecessors for each of those shortest paths (i.e. distances[1] and predecessors[1] assuming both of those are 2-d lists that you created). You may write both algorithms in the same file and allow the user to select which algorithm to run. You will need to parse the input file into some sort of graph representation. In this assignment, you may represent the graph however you like (adjacency matrix, adjacency list, something else).

**Example Runs:**

```bash
> python BellmanFord.py tinyEWD.txt 0
Distances from source:
[0.00, 0.93, 0.26, 0.99, 0.26, 0.61, 1.51, 0.60]
Predecessors:
[None, 5, 0, 7, 6, 4, 3, 2]
```

```bash
> python FloydWarshall.py tinyEWD.txt
All Pair-Wise Distances:
[ 0.00, 0.93, 0.26, 0.99, 0.26, 0.61, 1.51, 0.60]
[-0.59, 0.00, -0.39, 0.29, -0.44, -0.09, 0.81, -0.07]
[-0.15, 0.67, 0.00, 0.73, 0.00, 0.35, 1.25, 0.34]
[-0.88, -0.06, -0.68, 0.00, -0.73, -0.38, 0.52, -0.36]
[-0.12, 0.67, 0.08, 0.76, 0.00, 0.35, 1.28, 0.37]
[-0.27, 0.32, -0.07, 0.61, -0.12, 0.00, 1.13, 0.25]
[-1.40, -0.58, -1.20, -0.49, -1.25, -0.90, 0.00, -0.88]
[-0.49, 0.33, -0.29, 0.39, -0.34, 0.01, 0.91, 0.00]
```

Predecessors:

```bash
[None, 7, 0, 7, 7, 7, 7, 2]
[6, None, 6, 1, 6, 6, 3, 6]
[7, 7, None, 7, 7, 7, 7, 2]
[6, 6, 6, None, 6, 6, 3, 6]
[7, 5, 7, 7, None, 4, 7, 4]
[6, 5, 6, 1, 6, None, 3, 6]
[6, 5, 6, 7, 6, 4, None, 4]
[6, 6, 6, 7, 6, 6, 3, None]
```
Challenge Problem. Challenge problems count for extra credit points.

Given a positive integer \( n \) consider a square grid consisting of \( n \) rows and \( n \) columns. We can index each element of the grid as \( g[i, j] \), where \( 1 \leq i, j \leq n \). An \( n \times n \) grid graph is an undirected graph whose nodes form an \( n \times n \) grid, and each grid node is connected by an edge to the (up to four) nodes immediately north, south, east, and west of it (assuming they exist). See figure below (a).

Grid graphs are often used to represent terrains, where each node is labeled with an elevation \( e[i, j] \). You may assume that there are no flat spots, in the sense that, if two nodes are adjacent, they have different elevation values. A node is called a local depression if all of its neighbors have strictly higher elevation values than its own (circled in above figure (b)). Suppose you are given an \( n \times n \) grid graph \( G = (V,E) \). (Observe that \( G \) has \( n^2 \) vertices and \( O(n^2) \) edges.)

Design an algorithm that finds any one local depression in \( G \). Of course, this would be easy to do in \( O(n^2) \) time. To get credit, you must solve the problem in \( O(n \log n) \) time or, even better, \( O(n) \) time. (This may seem impossible, since you do not have enough time to inspect every node of the graph, but it can be done. The fact that there are no flat spots is critical.)