Algorithms
Computer Science 140 & Mathematics 168
Spring 2007
Homework 4b
Due Tuesday, February 13

- The material this week is not covered in our book.
- Problem 4 (and the bonus Problem 5) do not specify which algorithm design paradigm to use. Among the challenges in these problems is to determine which of the paradigms (i.e. divide-and-conquer, dynamic programming, or greed) is most appropriate. You are highly encouraged to do *both* of these problems. An important skill in algorithm design is to determine which approach is the right one. Unlike a problem set in a book, in real life you'll rarely be told which approach to use for the problem that you're trying to solve. These two problems are intended to help you develop that skill!
- You are encouraged, but no longer required, to use LaTeX to typeset your solutions to assignments in this course.

1. **[15 Points] Greed Must be used with Extreme Caution!** In class we talked about the problem of finding the maximum number of non-conflicting courses from a given set of courses. We argued that if the courses are sorted in order of completion time, then the following greedy algorithm is guaranteed to find the maximum number of non-conflicting courses: Choose the course with earliest completion time. Cancel out all courses that conflict with that course. Now repeat the process for the remaining courses.

   Professor I. Lai of the Pasadena Institute of Technology has proposed the following four alternative greedy algorithms for this problem. None of them are correct! To show this, construct a separate counterexample (a set of intervals) for each algorithm below such that the algorithm does not find the optimal solution for your counterexample. Make sure to explain what the algorithm would do on your counterexample and what the optimal solution would be.

   Algorithm 1: Sort the courses by ending time as before. Now, run our original greedy algorithm in the opposite direction. That is,

choose the course that ends **latest**. Then cancel out all courses that conflict with that course. Now repeat the process for the remaining courses.

Algorithm 2: Sort the courses by increasing starting time (rather than ending time). Now, choose the course that starts first. Then cancel out all courses that conflict with that course. Now repeat the process for the remaining courses.

Algorithm 3: Forget about sorting the courses. Choose a course of shortest duration (that is the course that has the least length). Then cancel out all courses that conflict with that course. Now repeat the process for the remaining courses.

Algorithm 4: Don't sort the courses. Choose a course that conflicts with the fewest other courses, breaking ties arbitrarily. Then cancel out the courses that conflict with that course. Now repeat the process for the remaining courses. (This is the most challenging of the four parts of this problem!)

2. **[20 Points] Minqueues!** A Minqueue is an abstract data type (ADT) that supports the following operations:

   ENQUEUE(x): Inserts the number x into the Minqueue.

   DEQUEUE(): Removes the element that has been in the Minqueue for the longest time.

   FIND-MIN(): Returns the smallest value in the Minqueue but *does NOT* remove it from the Minqueue.

   (a) Your boss, Gill Bates, has proposed that a Minqueue be implemented using a simple doubly linked list data structure where ENQUEUEing is done in constant time by adding to the end of the queue, DEQUEUEing is done in constant time by removing the element at the front of the queue, and FIND-MIN is performed by searching through the linked list for the smallest element. Show that with this implementation of a Minqueue, there exists some sequence of n operations (i.e., some sequence of ENQUEUE, DEQUEUE, and FIND-MIN) that takes $\Omega(n^2)$ time.

   (b) Your officemate, Dr. Constance Argood, has proposed a clever data structure for this problem which she calls "Real Queue and

Helper Queue." Here's how it works: When an element is enqueued, it is placed into a regular queue (implemented as a doubly linked list). However, it is *also* placed at the tail of a special "helper queue" (also implemented as doubly linked list). The helper queue will always contain a subset of those elements in the real queue in sorted order from small elements at the front to large elements in the back. In particular, when an element x is inserted at the tail end of the helper queue, it checks to see if it is smaller than the element right in front of it in the helper queue. If so, it removes the element in front of it in the helper queue and again compares itself to its predecessor. It repeatedly removes its predecessors until it gets to a point that its predecessor is smaller than it! To dequeue, we simply remove that element from the head of the real queue. We also check to see if it is at the head of the helper queue, in which case we remove it from that queue as well. Finally, to determine which element is the minimum, we simply look at the front of the helper queue! Here's your task: Prove that this data structure can perform any sequence of n operations in time $O(n)$ for amortized time of $O(1)$ per operation. Specifically, you should prove this **three different ways:** using the **aggregation**, **accounting**, and **potential** methods. Give a separate proof for each technique. *Note: Normally the accounting method seems most natural. However, here the accounting is a bit subtle and you must be very careful. Make sure to explain how the accounting really pays for* ***all*** *of the work that is performed.*

3. **[30 Points] Stack + Stack = Queue.** The Shmorbodian Division of Millisoft Research has a really great implementation of a stack. Like any good stack implementation, it supports PUSH and POP in $O(1)$ time. Your boss wants you to implement a queue, but not from scratch! Your boss conjectures that a queue can be implemented using two stacks.

   Describe how a queue can be implemented using two stacks. Then, prove that your stack-based queue has the property that any sequence of n operations (selected from ENQUEUE and DEQUEUE) takes a total of $O(n)$ time resulting in amortized $O(1)$ time for each of these operations! You should prove this **three different ways:** using the **aggregation**,

accounting, and **potential** methods. Give a separate proof for each technique.

4. **[20 Points] Napquest!** Napquest is a new website that provides the shortest distance between pairs of cities. In particular, Napquest only considers maps in which the cities can be lined up from west to east, with all of the roads pointing east. For example, Figure 1 shows a Napquest map with five cities labelled 1 through 5. The numbers on the edges (roads) correspond to the lengths of those roads.

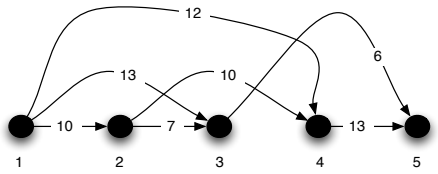

Figure 1: A map in which the cities are lined up east to west and all roads point east.

Here's your task:

(a) Describe (in clear English or pseudo-code) an algorithm for finding the total length of the shortest path between a pair of vertices provided by the user (for example, the length of the shortest path from city 1 to 5 in the map above is 19 - it's the path from city 1 to city 3 to city 5.) If there is no path between the given pair of vertices, the answer should be ∞. For full credit your algorithm should be as fast as possible. (*Note:* Please remember that you should not use any outside materials - the web or other written materials - in solving these problems. Moreover, if you happen to know of some shortest path algorithms already, they are probably much too hard to prove correct and much slower than optimal for the kinds of maps under consideration here.)

(b) Prove the correctness of your algorithm (using induction, contradiction, or whatever proof technique is most appropriate).

(c) Carefully derive the running time of your algorithm. Express your running time as a function of n (the number of cities) and m (the number of roads).

5. **[25 Point] OPTIONAL BONUS PROBLEM: East-West Traveling Salesperson Tours!** The Euclidean Traveling Salesperson Problem is the following: Given n points in the plane (corresponding to cities), find a cycle (or "tour") that visits each vertex (city) exactly once and minimizes the total Euclidean distance travelled. For example, Figure 2 shows 7 points in the plane. Figure 2(a) shows the optimal traveling salesperson tour. Each vertex is visited once and only once. Thus, a salesperson using this cycle could start at her home city, visit each city exactly once, and return home at a minimum total travel distance.

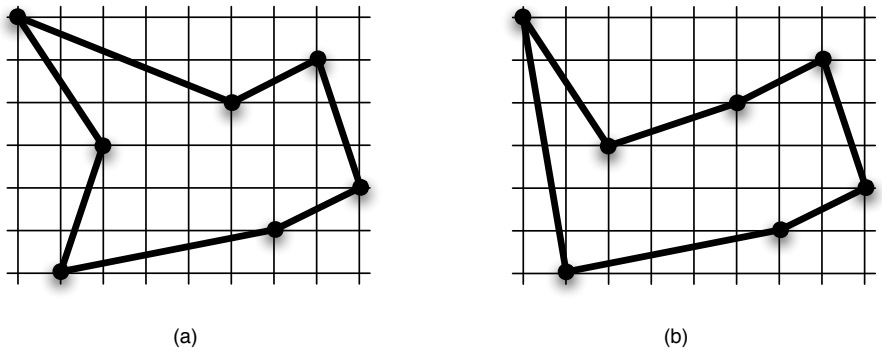

Figure 2: Seven points in the plane. (a) A shortest travelings salesperson tour. (b) A shortest "East-West" traveling salesperson tour.

Unfortunately, there is no known polynomial time algorithm for solving this problem. (Of course, we could enumerate all $O(n!)$ possible cycles and determine which one is cheapest, but that would take too long for any interesting value of n.) Worse yet, the problem is NP-complete (more about that later in the semester), meaning that if we could find an efficient algorithm for this problem then we would have instantly solved tens of thousands of open problems in mathematics and computer science. (NP-completeness is awesome!)

However, a special kind of traveling salesperson cycle called an "East-West" cycle can be found very efficiently. An "East-West" cycle starts at the westernmost point. It then heads east visiting some number of the cities until it gets to the easternmost point. It then heads back west visiting the cities that have not yet been visited. Figure 1(b) shows an

optimal East-West cycle for the same 7 points shown in Figure 1(a). *You should assume that no two points have the same x-coordinate.*

Describe an efficient algorithm for finding an optimal East-West cycle for the Euclidean Traveling Salesperson problem. Explain briefly but convincingly why your algorithm is correct (in essence, give the sketch of the proof of correctness), and analyze the running time of your algorithm.