Quiz 2

This take-home quiz contains 6 problems, each worth 25 points, for a total of 150 points. Each problem should be answered on a separate sheet (or sheets) of three-hole punch paper, so that problems can be graded separately. Mark the top of each sheet with your name, 6.046J/18.410J, the problem number, your recitation time and TA, and the date. The exam is due at 11:00 a.m. on Thursday, November 16th, to the course secretary in NE43-366. Late exams will not be accepted unless you have made prior arrangements with your TA.

The quiz should take you about 12 hours to do, but you have three days in which to do it. Plan your time wisely. Do not overwork. Ample partial credit will be given for good solutions.

Use algorithms and theorems from CLR to simplify your solutions. Do not dwell on the obvious. Full proofs are unnecessary, but you should provide convincing indications or arguments for your solutions. Write up your solutions to be understood.

Sometimes, translating a problem into computer science terms requires assumptions that affect the solution. You may also need an assumption if the problem statement is incomplete or ambiguous. If you make assumptions, state them plainly. Do not assume too much, however, because solutions to trivial problems will not receive much credit.

Bugs, etc.: Bug reports will be sent out to the 6.046 class email list. If you are not on the list, please let your TA know so that s/he can add your name.

Policy on academic honesty: This quiz is “limited open book.” You may use your notes from the course, and the CLR textbook, but no other sources whatsoever may be consulted. Specifically, you may not communicate with any person except the 6.046 staff about any aspect of the exam until after 11:00 A.M. on Thursday, even if you have already handed in your exam. Moreover, you may not use notes or solutions from other times that this course or other related courses have been taught. If at any time you feel that you may have violated these conditions, it is imperative that you contact your TA or Professors immediately. If you have any questions about what resources may or may not be used during the quiz, contact your TA or send email to 6.046-staff@theory.lcs.mit.edu.

READ THESE INSTRUCTIONS ONCE A DAY DURING THE EXAM. GOOD LUCK!
Problem 1. Trick or Treat (10 Points).
You are given an \(m \times n\) Hershey Bar which you are to crack into \(mn\) pieces, each sized \(1 \times 1\). An elementary move in this model takes a single \(j \times k\) piece and cracks it along a single vertical or horizontal edge. How many moves does it take to reduce the bar to a collection of \(1 \times 1\) pieces?

Problem 2. A Mysterious Graph Algorithm (30 Points).
Consider the following algorithm: Given a connected, weighted, undirected graph \(G\), repeatedly remove the heaviest edge whose removal will not disconnect the graph. What does this algorithm produce at termination? Prove the algorithm correct. Give pseudo-code for an efficient implementation of the algorithm (include pseudo-code for any non-trivial subroutines). What is your algorithm’s running time in terms of \(V\) and \(E\), the number of vertices and edges in \(G\)?

Problem 3. A Queue from Two Stacks (30 Points).
Suppose we wish to realize a first-in-first-out queue, using only two stacks and the primitive stack operations \textbf{PUSH} and \textbf{POP}.
Consider this implementation of \textbf{ENQUEUE} and \textbf{DEQUEUE} (ignoring error-handling):

\textbf{STACK} \(P, Q; \quad \triangleright \text{Initially empty.}

\textbf{ENQUEUE}( \text{Key} \; k) \\
\quad \textbf{PUSH} (k, P);

\textbf{DEQUEUE}() \\
\quad \text{if (\;\text{Q is Empty})} \\
\makebox[0.5in]{\quad \textbf{while (\;P is not Empty)} \\
\makebox[1.0in]{\quad \textbf{PUSH} (\;\text{POP} (\;P\;), \;Q);} \\
\quad \text{return POP (\;Q);}

Consider an initially empty queue, and any sequence of \(n\) operations, each of which is either \textit{Enqueue} or \textit{Dequeue}.

\textbf{A.} Make a table with four columns labeled “Operation,” “State of \(P\),” “State of \(Q\),” and “Result of D”, and fill it in for the sequence of operations E1, E2, D, E3, D, D, E4, E5, D, D (where Ex means “Enqueue(x)” and D means “Dequeue”). Show that the data structure dequeues keys in the order that they were enqueued.

\textbf{B.} Use the Accounting Method to bound the total cost of all operations. Deduce an amortized cost per operation.

\textbf{C.} Use the Potential Method to bound the total cost of all operations. Deduce an amortized cost per operation.
Problem 4. Converting Between Graph Representations (30 Points).

Consider a graph $G$ embedded in the plane such that no two edges of the graph meet except at vertices. Such a graph partitions the plane into a set of $F$ polygonal “faces,” each of which is bounded by three or more edges. If we consider only faces of finite extent, then Euler’s relation holds that $V - E + F = 1$.

Suppose we are given the graph described not in the customary way, but as a list of independent faces. Each face is specified as a list of vertices in counter-clockwise order. Each vertex is specified as two integer-valued coordinates $x, y$.

Here is an example graph and its independent-faces description:

```
F1: [(2, 1), (5, 3), (1, 5)]
F2: [(5, 3), (8, 6), (5, 7), (1, 5)]
F3: [(9, 2), (8, 6), (5, 3)]
F4: [(5, 3), (2, 1), (6, 1)]
```

The “flattened” description above is well-suited for some operations like land surveying. For other applications, however, the flattened description is a poor one. Consider, for example, running depth-first or breadth-first search on a flattened graph, or finding a minimum spanning tree for the graph.

A. Give an algorithm which takes as input a flattened description of a graph $G$, and produces as output a standard adjacency-list representation of $G$. Make your algorithm as efficient as you can; it is possible to achieve worst-case space usage, and expected running time, linear in $V$, $E$ and $F$. You will probably have to design a data structure, using data structures we have seen in lecture, to do so. (Hint: as your algorithm scans the input, how can it efficiently “recognize” vertices and/or edges that it has previously encountered?) Ignore parsing details; you can assume that your parser correctly delimits faces and vertices in the input.

B. Argue informally for the correctness of your algorithm.

C. Analyze your algorithm’s space usage and running time.