Banner number: Name:

Final Exam

CSCI 3110: Design and Analysis of Algorithms

December 13, 2007

Group 1		Group 2		Group 3		$\sum$
Question 1.1		Question 2.1		Question 3.1	☐	
Question 1.2		Question 2.2		Question 3.2	☐	
Question 1.3		Question 2.3		Question 3.3	☐	
$\sum$		$\sum$		$\sum$		

Instructions:

- The questions are divided into three groups. You have to answer **all questions in Groups 1 and 2** and **exactly two questions in Group 3**. In the above table, put check marks in the **small** boxes beside the questions in Group 3 you want me to mark. I will not mark unmarked questions. If you mark all three questions, I will choose which two to mark at random.
- Provide your answer in the box after each question. If you absolutely need extra space, use the backs of the pages; but try to avoid it. Keey your answers short and to the point.
- **You are not allowed to use a cheat sheet.**
- If you are asked to design an algorithm and you cannot design one that achieves the desired running time, design a slower algorithm that is correct. A correct and slow algorithm earns you 50% of the marks for the algorithm. A fast and incorrect algorithm earns 0 marks.
- When designing an algorithm, you are allowed to use algorithms and data structures you learned in class as black boxes, without explaining how they work, as long as these algorithms and data structures do not directly answer the questions.
- **Read every question carefully before answering. In particular, do not waste time on an analysis if none is asked for, and do not forget to provide one if it is required.**
- **Do not forget to write your banner number and name on the top of this page.**
- **This exam has 13 pages, including this title page. Notify me immediately if your copy has fewer than 13 pages.**

Question 1.1 (Algorithm design paradigms) 10 marks

Name three algorithm design paradigms and explain for each of them what characterizes an algorithm that is based on this paradigm. At least one of the paradigms you describe should apply to solving optimization problems.

1.

Greedy algorithms *solve optimization problems by making locally optimal choices.*

2.

A ***divide-and-conquer*** *algorithm partitions the input into one or more smaller instances of the same problem, solves them recursively and then combines their solutions.*

3.

An algorithm based on ***graph exploration*** *tries to learn the structure of a graph by systematic exploration of edges leading to previously unexplored vertices.*

State which of the above paradigms can be used to solve optimization problems. (If you named more than one such paradigm above, list all of them here.)

Greedy algorithms

Question 1.2 (Randomization) 10 marks

Define the following terms:

Worst-case running time of a deterministic algorithm

The worst-case running time of a deterministic algorithm is a function T such that $T(n)$ is the maximum of the running times of the algorithm on all inputs of size n.

Average-case running time of a deterministic algorithm

The average-case running time of a deterministic algorithm is a function T such that $T(n)$ is the average of the running times of the algorithm on all inputs of size n. This average is calculated assuming an appropriate probability distribution of the possible inputs of size n.

What is the difference between the average-case running time of a deterministic algorithm and the expected running time of a randomized algorithm?

For the deterministic algorithm, the average is taken over all possible inputs. For the randomized algorithm, the average is taken over all possible outcomes of the random choices the algorithm makes.

What is a possible advantage of a randomized algorithm for a given problem compared to a deterministic algorithm whose worst-case running time is *the same* as the expected running time of the randomized algorithm?

The randomized algorithm is probably much simpler.

Question 1.3 (Computational complexity) 10 marks

What is the difference between a decision problem and an optimization problem?

A decision problem asks a yes/no question. An optimization problem asks to choose the best from a set of possible solutions to a problem, according to an appropriate quality measure of the solutions.

Define the complexity class P.

P *is the class of all formal languages (ie, decision problems) that can be decided in polynomial time.*

Define the complexity class NP.

NP *is the class of all formal languages that can be verified in polynomial time.*

Define the term *NP-hard problem* and *NP-complete problem*.

A problem P is NP*-hard if $P \in$ P implies that every problem in* NP *belongs to* P*, ie,* P = NP*. An* NP*-complete problem is an* NP*-hard problem that belongs to* NP.

In the following table, put a check mark in row x and column y if problem y belongs to complexity class x.

	Sorting	Hamiltonian cycle	Is there a path of length at most k from x to y in graph G?
P			✓
NP		✓	✓

Question 2.1 (Correctness proof) — 15 marks

Just as a reminder, here is Dijkstra's algorithm:

DIJKSTRA(G, s)

```
 1  mark every vertex v ∈ G as unexplored
 2  d(v) ← +∞, for all v ∈ G
 3  d(s) ← 0
 4  insert every vertex v ∈ G into a priority queue Q, with priority d(v)
 5  while Q ≠ ∅
 6        do v ← DeleteMin(Q)
 7           mark v as explored
 8           for every out-neighbor w of G
 9           if d(w) > d(v) + w(vw)
10              then d(w) ← d(v) + w(vw)
11                   DecreaseKey(Q, w, d(w))
```

Prove that, when this procedure finishes, $d(v)$ is the length of a shortest path from s to v in G, provided that all edges in G have non-negative weights.

Let us use $D(v)$ to denote the distance from s to v in G, and let us call a vertex v settled *if $v \notin Q$. It is easy to see that the algorithm maintains the invariant that $d(v) = \min_u d(u) + w(uv)$ for every vertex $v \neq s$, where the minimum is taken over all settled neighbors of v. We have to prove that, when v becomes settled, ie, is taken from Q, then $d(v) = D(v)$. We prove this by induction on the number of settled vertices.*

The first vertex to be settled is s because initially all vertices except s have infinite priority. Since $d(s) = 0 = D(s)$, this establishes the base case.

Now consider a vertex v that is settled after s. Then we have $d(v) = d(u) + w(uv)$ for some settled vertex u. By the induction hypothesis, we have $d(u) = D(u)$, that is, $d(v)$ is the length of the shortest path P from s to v that visits u immediately before v. Assume that P is not the shortest path from s to v. Then there exists a shorter path P' from s to v. Let w be the first vertex in P' that is not settled yet, let P_1 be the subpath of P' from s to w, and let P_2 be the subpath of P' from w to v. Since all edges in G have non-negative weights, the length $w(P_2)$ of path P_2 is non-negative. Hence, the length $w(P_1)$ of path P_1 is at most the length of P', which is less than $d(v)$. Since all vertices in P_1 except w are settled, w's predecessor x on P_1 must be settled. Hence, $d(w) \leq d(x) + w(xw) = D(x) + w(xw) \leq w(P_1) < d(v)$. This, however, is a contradiction because w would be removed from Q before v.

Question 2.2 (Analysis of algorithms) 15 marks

The following algorithm is an algorithm for multiplying two square matrices in subcubic time. In the description of the algorithm, we use n to denote the number of rows and columns in the two input matrices, and $m_{i,j}$ denotes the entry in row i and column j of matrix M. We assume that n is a power of 2. If $n \geq 2$, we use $M_{1,1}$, $M_{1,2}$, $M_{2,1}$, and $M_{2,2}$ to denote the top-left, top-right, bottom-left, and bottom-right quarter of matrix M, respectively.

STRASSEN(A, B)

1 **if** $n = 1$
2 **then return** $a_{1,1} \cdot b_{1,1}$
3 $M_1 \leftarrow$ STRASSEN$(A_{1,1} + A_{2,2}, B_{1,1} + B_{2,2})$
4 $M_2 \leftarrow$ STRASSEN$(A_{2,1} + A_{2,2}, B_{1,1})$
5 $M_3 \leftarrow$ STRASSEN$(A_{1,1}, B_{1,2} - B_{2,2})$
6 $M_4 \leftarrow$ STRASSEN$(A_{2,2}, B_{2,1} - B_{1,1})$
7 $M_5 \leftarrow$ STRASSEN$(A_{1,1} + A_{1,2}, B_{2,2})$
8 $M_6 \leftarrow$ STRASSEN$(A_{2,1} - A_{1,1}, B_{1,1} + B_{1,2})$
9 $M_7 \leftarrow$ STRASSEN$(A_{1,2} - A_{2,2}, B_{2,1} + B_{2,2})$
10 $C_{1,1} \leftarrow M_1 + M_4 - M_5 + M_7$
11 $C_{1,2} \leftarrow M_3 + M_5$
12 $C_{2,1} \leftarrow M_2 + M_4$
13 $C_{2,2} \leftarrow M_1 - M_2 + M_3 + M_6$
14 **return** C

Analyze the running time of this algorithm, that is, provide an expression $T(n) = \Theta(f(n))$ and prove that this is indeed the right bound for the running time of the algorithm.

We observe that the algorithm makes 7 recursive calls on $n/2 \times n/2$ matrices. Apart from that, the algorithm performs a constant number of additions and subtractions of $n/2 \times n/2$ matrices, which take $\Theta(n^2)$ time. Hence, the running time of the algorithm is given by the recurrence

$$T(n) = 7T(n/2) + \Theta(n^2).$$

Since $\log_2 7 > 2$, the Master Theorem tells us that this recurrence solves to $T(n) = \Theta(n^{\log_2 7})$, which is $o(n^3)$.

Question 2.3 (Lower bounds) 10 marks

The idea behind every NP-hardness proof we discussed in class was that of a polynomial-time reduction. A similar idea can be used to prove lower bounds for the running times of algorithms or data structure operations. In class, I mentioned that, using comparisons only to determine the correct order of elements, it is impossible to sort in $o(n \log n)$ time. Using this fact, prove that it is impossible to construct a comparison-based dictionary data structure that supports INSERT and SUCCESSOR operations in $o(\log n)$ time and the MINIMUM operation in $o(n \log n)$ time. (*Hint: Assume that you had such a dictionary. Can you use it to design a comparison-based sorting algorithm that runs in* $o(n \log n)$ *time? This would lead to a contradiction.*)

Here is an algorithm that sorts using only INSERT, SUCCESSOR, *and* MINIMUM *operations on a dictionary:*

```
DictSort(A)
1  D ← an empty dictionary
2  for i ← 1 to n
3        do Insert(D, A[i])
4  x ← Minimum(D)
5  for i ← 1 to n − 1
6        do A[i] ← x
7           x ← Successor(D, x)
8  A[n] ← x
```

Ignoring the cost of the dictionary operations, this algorithm takes $O(n)$ *time because it consists of two loops with* $O(n)$ *iterations, each of which performs a constant amount of work. Apart from that, the algorithm performs* n INSERT *operations,* $n - 1$ SUCCESSOR *operations, and one* MINIMUM *operation. If* INSERT *and* SUCCESSOR *operations took* $o(\log n)$ *time each, then the total cost of these operations performed by the algorithm would be* $o(n \log n)$*. If, in addition, the* MINIMUM *operation took* $o(n \log n)$ *time, then it would add only* $o(n \log n)$ *to the total running time of the algorithm. Hence, the total running time of the algorithm would be* $O(n) + o(n \log n) + o(n \log n) = o(n \log n)$*, which is impossible because the algorithm is a correct sorting algorithm.*

Question 3.1 (Divide and conquer) 15 marks

Given a sequence of numbers $S = (x_1, x_2, \ldots, x_n)$, define the *largest smaller predecessor* (LSP) of x_i to be $\max(\{x_j \mid j < i \text{ and } x_j < x_i\} \cup \{-\infty\})$. Develop an $O(n \log n)$-time algorithm that determines the LSP of every element in S. Provide a discussion of your algorithm that establishes its correctness and running time.

The algorithm starts by dividing S into two subsequences L and R with L containing the first $n/2$ elements and R containing the last $n/2$ elements of S. Then it finds the LSP of each element in L and R relative to its subsequence. Now we observe that all predecessors of an element in L belong to L. Hence, the recursive call on L already returns the correct LSP w.r.t. S for every element in L. For an element x in R, its LSP is either its LSP in R or the largest element less than x in L, whichever is greater. This is true because every element in L is a predecessor of every element in R.

Thus, we can compute LSP's using an augmented version of Merge Sort. Then the recursive calls on L and R return these two lists in sorted order. Now, as we merge L and R, we remember the last element l in L we have placed into S. When placing an element from L into S, this element becomes l. When placing an element x from R into S, we compare the current LSP of x to l. If l is greater, then l becomes the new LSP of x.

The modified merge step in this procedure performs constant additional work per element and, thus, still has linear cost. Thus, the running time of the algorithm is still given by the same recurrence as for Merge Sort: $T(n) = 2T(n/2) + O(n)$. We know that this recurrence solves to $T(n) = O(n \log n)$.

Question 3.1 (Continued)

Question 3.2 (Dynamic programming) 15 marks

Here is a problem closely related to the sequence alignment problem discussed in class: Given two strings $X = x_1x_2\dots x_m$ and $Y = y_1y_2\dots y_n$, a *common substring* of X and Y is a string $Z = z_1z_2\dots z_k$ that can be obtained from both X and Y by removing an appropriate set of letters. Formally, we have $z_h = x_{i_h} = y_{j_h}$, for appropriate indices $1 \le i_1 < i_2 < \dots < i_k \le m$ and $1 \le j_1 < j_2 < \dots < j_k \le n$. Z is a *longest common substring* (LCS) of X and Y if there is no common substring of X and Y that is longer than Z. Develop an algorithm that finds an LCS of X and Y in $O(mn)$ time.

Let us use X_i and Y_j to denote the substrings $x_1x_2\dots x_i$ and $y_1y_2\dots y_j$, respectively. If we use $l(i,j)$ to denote the length of an LCS of X_i and Y_j, then the LCS of X and Y has length $l(m,n)$.

If $i = 0$ or $j = 0$, then X_i or Y_j is the empty string, and obviously $l(i,j) = 0$. So assume that $i > 0$ and $j > 0$ and that $Z = z_1z_2\dots z_k$ is an LCS of X_i and Y_j. In this case, there are two cases: If $x_i = y_j$, then $z_k = x_i = y_j$ and the sequence Z_{k-1} is an LCS of X_{i-1} and Y_{j-1}. If $x_i \neq y_j$, then we may have $z_k = x_i$, $z_k = y_j$ or neither, but we cannot have $z_k = x_i = y_j$. Hence, Z is either an LCS of X_i and Y_{j-1} or of X_{i-1} and Y_j. This leads to the following recurrence:

$$l(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + l(i-1, j-1) & \text{if } i > 0, j > 0, \text{and } x_i = y_j \\ \max(l(i,j-1), l(i-1,j)) & \text{otherwise} \end{cases}$$

The following algorithm uses this recurrence to compute a table l that stores all these values $l(i,j)$.

LCS(X, Y)

```
1  for j ← 0 to n
2        do l[0, j] ← 0
3  for i ← 1 to m
4        do l[i, 0] ← 0
5  for j ← 1 to n
6        do if x_i = y_j
7              then l[i, j] ← 1 + l[i − 1, j − 1]
8              else l[i, j] ← max(l[i, j − 1], l[i − 1, j])
9  return l
```

Question 3.2 (Continued)

An LCS of X and Y is now easy to extract from table l:

```
FIND-LCS(X, Y, l)
 1  S ← an empty stack
 2  i ← m
 3  j ← n
 4  while i > 0 and j > 0
 5        do if x_i = y_j
 6              then PUSH(S, x_i)
 7                   i ← i − 1
 8                   j ← j − 1
 9              else  if l[i − 1, j] > l[i, j − 1]
10                      then i ← i − 1
11                      else  j ← j − 1
12  while S ≠ ∅
13        do x ← POP(S)
14           output x
```

Question 3.3 (Graph algorithms) — 15 marks

In class, we discussed the Bellman-Ford shortest path algorithm, which computes correct shortest paths from a source vertex s to all other vertices in a directed graph G even in the presence of negative edge weights. Of course, distances are well defined only if G does not contain a negative cycle (ie, a directed cycle whose total edge weight is negative). Therefore, it would be useful to be able to detect when G contains a negative cycle. Using the Bellman-Ford algorithm as a starting point, develop an algorithm that correctly decides whether G contains a negative cycle. Your algorithm doesn't have to report a negative cycle, only give a correct yes/no answer. The running time of your algorithm should be $O(nm)$. Prove that your algorithm is correct.

We start by running Bellman-Ford to compute a claimed distance $d(v)$ from s to every vertex $v \in G$. This takes $O(nm)$ time. Now we inspect all edges $xy \in G$. If we find an edge such that $d(y) > d(x)+w(xy)$, we report that G contains a negative cycle. Otherwise, we report that G contains no such cycle. This test clearly takes $O(m)$ time. Thus, the total running time is $O(nm)$.

If G contains no negative cycle, then we know that once Bellman-Ford finishes, every vertex x has $d(x)$ set to its distance from s. This implies in particular that $d(y) \leq d(x) + w(xy)$ for every edge $xy \in G$. Thus, if G contains no negative cycle, we correctly report this fact.

Now assume that G contains a negative cycle $C = x_0x_1 \cdots x_{q-1}$. We claim that we must have $d(x_{i+1}) > d(x_i) + w(x_ix_{i+1})$ for some edge x_ix_{i+1}, where addition of indices is modulo q. This then implies that our algorithm's test whether $d(x_{i+1}) \leq d(x_i) + w(x_ix_{i+1})$ fails, and we correctly report that G contains a negative cycle. It remains to prove the claim.

Assume for the sake of contradiction that $d(x_{i+1}) \leq d(x_i) + w(x_ix_{i+1})$ for every edge in C. Then we have $d(x_1) \leq d(x_0)+w(x_0x_1)$ and, inductively, $d(x_0) \leq d(x_0)+\sum_{i=0}^{q-1} w(x_ix_{i+1})$. Since C is a negative cycle, however, we have $\sum_{i=0}^{q-1} w(x_ix_{i+1}) < 0$ and, hence, $d(x_0) < d(x_0)$, which is obviously impossible. Hence, one of the edges x_ix_{i+1} must satisfy $d(x_{i+1}) > d(x_i) + w(x_ix_{i+1})$.

Question 3.3 (Continued)