Banner number: Name:

Final Exam

CSCI 3110: Design and Analysis of Algorithms

August 10, 2018

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: Group 1 (40%), Group 2 (36%), and Group 3 (24%). You have to answer **all questions in Groups 1 and 2** and **exactly two questions in Group 3**. In the above table, put a check mark in the **small** box beside the question in Group 3 you want me to mark. If you select none or both questions, I will randomly choose which one to mark.
- 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. Keep your answers short and to the point.
- **You are not allowed to use a cheat sheet.**
- **Make sure your answers are clear and legible. If I can't decipher an answer or follow your train of thought with reasonable effort, you'll receive 0 marks for your answer.**
- 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 question.
- **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 12 pages, including this title page. Notify me immediately if your copy has fewer than 12 pages.**

Question 1.1 (Asymptotic growth of functions) 6 marks

(a) *Formally* define the worst-case running time of an algorithm.

The worst-case running time of an algorithm is a function $T(\cdot)$ of the algorithm's input size. For a fixed input size n, $T(n)$ is the maximum running time of the algorithm over all inputs of size n.

(b) *Formally* define the set $\Theta(f(n))$.

$\Theta(f(n))$ is the set of all functions $g(n)$ such that there exist constants $n_0 \geq 0$, $c_1 > 0$, and $c_2 > 0$ such that $c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$ for all $n \geq n_0$.

(c) Consider two algorithms A and B that both solve some problem P in $\Theta(n \lg n)$ time in the worst case and assume that your primary goal is to solve P as quickly as possible on large inputs. Based on the available information, can you decide which of the two algorithms is preferable? If yes, explain how. If not, explain why not.

We only know that $c_1 n \lg n \leq T_A(n) \leq c_2 n \lg n$ and $c_3 n \lg n \leq T_B(n) \leq c_4 n \lg n$ for appropriate constants $c_1, c_2, c_3, c_4 > 0$ and for sufficiently large n. It is possible that $c_1 \gg c_4$ or $c_3 \gg c_2$. In the former case, B is preferable over A; in the latter, A is preferable over B. So, no, it is not possible to decide which of the two algorithms is preferable based on the provided information.

Question 1.2 (Worst case, average case, and amortization) 6 marks

Consider three data structures D_1, D_2, and D_3. All three support the same operations. Any operation on D_1 takes $O(\lg^2 n)$ time in the average case and in the worst case. Any operation on D_2 takes $O(n)$ time in the worst case but $O(\lg n)$ amortized time. D_3's behaviour depends on the input. For a uniform random sequence of operations on D_3, the expected cost per operation on D_3 is in $O(\lg n)$. The worst-case cost per operation on D_3 is in $O(n)$. Now consider using any of these data structures in an algorithm that performs $O(n)$ data structure operations and spends a negligible amount of time on tasks other than data structure operations. You may assume that the sequence of operations the algorithm performs on the data structure is a uniform random sequence. In the following table, list the average-case and worst-case running times of the algorithm when using D_1, D_2 or D_3:

Data structure	D_1	D_2	D_3
Worst-case running time	$O(n \lg^2 n)$	$O(n \lg n)$	$O(n^2)$
Average-case running time	$O(n \lg^2 n)$	$O(n \lg n)$	$O(n \lg n)$

Question 1.3 (Complexity classes) 8 marks

(a) *Formally* define the complexity class P.

P is the class of all formal languages that can be decided in polynomial time. Formally, a language $L \subseteq \Sigma^*$ belongs to P if there exists an algorithm D which, for any string $x \in \Sigma^*$, answers yes if and only if $x \in L$, and the running time of D on any string $x \in \Sigma^*$ is in $O(|x|^c)$, for some constant c.

(b) *Formally* define the complexity class NP.

NP is the class of all formal languages that can be verified in polynomial time. Formally, a language $L \subseteq \Sigma^*$ belongs to NP if there exists a language $L' \subseteq \Sigma^* \times \Sigma^*$ such that $L' \in P$ and any string $x \in \Sigma^*$ belongs to L if and only if there exists a string $y \in \Sigma^*$ with $|y| \in O(|x|^c)$ and such that $(x, y) \in L'$.

(c) *Formally* define what an NP-hard language is.

A language L is NP-hard if $L \in P$ implies that $P = NP$.

Question 2.1 (Solving recurrences) **7 marks**

Solve the following recurrences using whichever method you like (Master Theorem, substitution or recursion tree), that is, provide a function $f(n)$ such that $T(n) \in \Theta(f(n))$ and prove that $T(n) \in O(f(n))$. You do not need to prove that $T(n) \in \Omega(f(n))$ and you do not need to use the same method for both recurrences.

(a) $T(n) = 4T(n/3) + n\lg n$

Since $\lg n \in o(n^\epsilon)$ *for all* $\epsilon > 0$*, we have* $n \lg n \in O(n^{\log_3 4-\epsilon})$ *for all* $0 < \epsilon < \log_3 4 - 1$*. Since* $\log_3 4 > 1$*, such an* ϵ *exists. Thus, by the Master Theorem,* $T(n) \in \Theta(n^{\log_3 4})$.

(b) $T(n) = T(n/2) + T(n/3) + n\lg n$

We claim that $T(n) \in \Theta(n \lg n)$.

To prove that $T(n) \in O(n\lg n)$*, observe that* $T(n) \in O(1) \le cn\lg n$ *for* $2 \le n < 6$ *and* c *large enough.*

For $n \ge 6$*, we have*

$$\begin{aligned}
T(n) &= T\left(\frac{n}{2}\right) + T\left(\frac{n}{3}\right) + n\lg n \\
&\le \frac{cn}{2}\lg\frac{n}{2} + \frac{cn}{3}\lg\frac{n}{3} + n\lg n && \text{(by the inductive hypothesis)} \\
&\le \frac{5cn}{6}\lg n + n\lg n && \text{(because } \lg\frac{n}{3} \le \lg\frac{n}{2} \le \lg n\text{)} \\
&\le cn\lg n && \forall c \ge 6.
\end{aligned}$$

Question 2.2 (Correctness proof) **6 marks**

Consider the binary search algorithm for finding a given element x in a sorted array A:

```
BINARYSEARCH($A, l, r, x$)
1 if $r < l$
2     then return FALSE
3 $m = \lfloor (l + r)/2 \rfloor$
4 if $x = A[m]$
5     then return TRUE
6     else if $x < A[m]$
7             then return BINARYSEARCH($A, l, m - 1, x$)
8             else return BINARYSEARCH($A, m + 1, r, x$)
```

Prove that the invocation BINARYSEARCH$(A, 1, n, x)$ returns TRUE if and only if $x \in A[1\,..\,n]$.

We prove by induction on $r - l$ that the invocation BINARYSEARCH(A, l, r, x) *returns* TRUE *if and only if $x \in A[l\,..\,r]$. Setting $l = 1$ and $r = n$ then proves the claim.*

If $r < l$, then $A[l\,..\,r]$ is empty, so $x \notin A[l\,..\,r]$ and the algorithm correctly returns FALSE *in line 2.*

If $r \geq l$, then $l \leq m \leq r$, so $(m-1)-l < r-l$ and $r-(m+1) < r-l$. If $x = A[m]$, then $x \in A[l\,..\,r]$ because $l \leq m \leq r$, so the algorithm correctly returns TRUE *in line 5.*

If $x < A[m]$, then $x < A[i]$ for all $m \leq i \leq r$ because A is sorted. Thus, $x \in A[l\,..\,r]$ if and only if $x \in A[l\,..\,m-1]$. Since $(m-1)-l < r-l$, the invocation BINARYSEARCH$(A, l, m-1, x)$ *returns* TRUE *if and only if $x \in A[l\,..\,m-1]$, that is, if and only if $x \in A[l\,..\,r]$ given that $x < A[m]$. Thus, the algorithm returns the correct answer in line 7.*

If $x > A[m]$, then $x > A[i]$ for all $l \leq i \leq m$ because A is sorted. Thus, $x \in A[l\,..\,r]$ if and only if $x \in A[m+1\,..\,r]$. Since $r-(m+1) < r-l$, the invocation BINARYSEARCH$(A, m+1, r, x)$ *returns* TRUE *if and only if $x \in A[m+1\,..\,r]$, that is, if and only if $x \in A[l\,..\,r]$ given that $x > A[m]$. Thus, the algorithm returns the correct answer in line 8.*

Question 2.3 (Polynomial-time reductions) **5 marks**

Let $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Sigma^*$ be two formal languages. Assume L_1 is NP-hard and there exists a polynomial-time reduction R from L_1 to L_2. Prove that this implies that L_2 is also NP-hard.

We need to prove that $L_2 \in P$ implies that $P = NP$. Since $L_1 \in P$ implies that $P = NP$, it suffices to prove that $L_2 \in P$ implies that $L_1 \in P$.

So assume $L_2 \in P$, that is, there exists a decision algorithm D_2 such that, for any $x \in \Sigma^*$, $D_2(x) = true$ if and only if $x \in L_2$; the running time of D_2 on input x is in $O(|x|^{c_2})$ for some constant c_2.

We construct a decision algorithm D_1 for L_1 as $D_1(x) = D_2(R(x))$, that is, we first apply R to the input of D_1, then pass the result $R(x)$ to D_2, and return the answer this invocation of D_2 returns. Since $x \in L_1 \Longleftrightarrow R(x) \in L_2$ (R is a reduction from L_1 to L_2) and $R(x) \in L_2 \Longleftrightarrow D_2(R(x)) = true$ (D_2 decides L_2), we have $x \in L_1 \Longleftrightarrow D_2(R(x)) = D_1(x) = true$, that is, D_1 decides L_1.

The running time of D_1 on input x is the cost of running R on x plus the cost of running D_2 on $R(x)$. Since R is a polynomial-time reduction, its running time on input x is in $O(|x|^c)$ for some constant c. In time $O(|x|^c)$, R can produce an output of size at most $O(|x|^c)$, so $|R(x)| \in O(|x|^c)$. The running time of D_2 on $R(x)$ is in $O(|R(x)|^{c_2}) \subseteq O(|x|^{cc_2})$. Thus, the total cost of D_1 is in $O(|x|^c + |x|^{cc_2})$, which is polynomial in $|x|$.

Since D_1 decides L_1 and its running time on any input x is polynomial in $|x|$, $L_1 \in P$, which is what we had to show.

Question 3.1 (Dynamic programming) **6 marks**

Two players, A and B, play a game on a row of n coins with values $C_1, \ldots, C_n$. The two players take turns. In each turn, the player whose turn it is removes the first or last coin from the row of coins and gains its value. Assuming A is the first player to take a coin and both players play an optimal strategy, determine the amount of money each player wins. Your algorithm should take $O(n^2)$ time. Argue briefly that your algorithm achieves this running time and is correct.

Let $S_{i,j} = \sum_{h=i}^{j} C_h$ be the total value of coins $C_i, \ldots, C_j$. Given coins $C_i, \ldots, C_j$ remaining on the table, let $F_{i,j}$ be the value that can be won by whoever moves first on this subset of coins. Then the value we aim to determine is $F_{1,n}$, the value player A wins, and $S_{1,n} - F_{1,n}$, the value player B wins.

$F_{i,i} = C_i$ for all $1 \le i \le n$ because the first player takes the only available coin on the table and the game ends.

For $i < j$, the first player can take C_i or C_j. In the former case, the first player gains the value of coin C_i and the second player is the first player to move on $\langle C_{i+1}, \ldots, C_j \rangle$. Thus, the second player wins $F_{i+1,j}$ and the first player wins $C_i + S_{i+1,j} - F_{i+1,j} = S_{i,j} - F_{i+1,j}$. In the latter case, the first player gains the value of coin C_j and the second player is the first player to move on $\langle C_i, \ldots, C_{j-1} \rangle$. Thus, the second player wins $F_{i,j-1}$ and the first player wins $C_j + C_{i,j-1} - F_{i,j-1} = C_{i,j} - F_{i,j-1}$. The first player chooses the strategy that maximizes his winnings, so $F_{i,j} = S_{i,j} - \min(F_{i,j-1}, F_{i+1,j})$.

This gives the following simple algorithm:

```
CoinGame($C$)
 1  for $i = 1$ to $n$
 2      do $S[i,i] = C[i]$
 3          for $j = i+1$ to $n$
 4              do $S[i,j] = S[i,j-1] + C[j]$
 5  for $i = 1$ to $n$
 6      do $F[i,i] = S[i,i]$
 7  for $k = 1$ to $n-1$
 8      do for $i = 1$ to $n-k$
 9              do $F[i,i+k] = S[i,i+k] - \min(F[i,i+k-1], F[i+1,i+k-1])$
10  return $(F[1,n], S[1,n] - F[1,n])$
```

The running time of this algorithm is easily seen to be in $O(n^2)$. Lines 1–4 consist of two nested loops with at most n iterations each. The same is true for lines 7–9. Lines 5–6 consist of a single loop with n iterations. Since each iteration takes constant time, the algorithm takes $O(n^2)$ time.

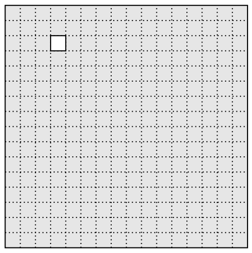
The correctness of the algorithm follows from two observations: (i) Each value $F[i,j]$ is computed as $F[i,j] = S[i,j] - \min(F[i,j-1], F[i+1,j])$ following the recurrence established by the above discussion. (ii) When computing $F[i,j]$, $F[i,j-1]$ and $F[i+1,j]$ have already been computed and thus can be used. This is true because we evaluate all entries in F by increasing difference $k = j - i$.

Extra space for Question 3.1

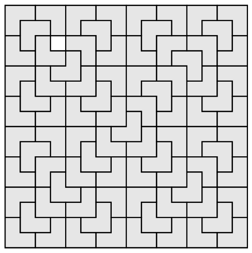
Question 3.2 (Divide and conquer) **6 marks**

A *deficient grid* is a $2^n \times 2^n$ grid with an arbitrary cell missing. Thus, a deficient grid has $4^n - 1$ cells. A *tromino* is a deficient $2^1 \times 2^1$ grid. Develop an algorithm that tiles any deficient grid with trominoes, that is, it places trominoes on the deficient grid so that every grid cell is covered by a tromino and no two trominoes overlap. Your algorithm should have running time $O(4^n)$. Prove that your algorithm is correct and that it achieves this running time.

Hint: *Note that this is a divide and conquer question. Can you express the problem of tiling a deficient $2^n \times 2^n$ grid in terms of tiling deficient $2^{n-1} \times 2^{n-1}$ grids?*


A deficient grid with cell $(4, 14)$ missing

A tromino


A tiling of the deficient grid on the left with trominoes

If $n = 1$, then the grid has the shape of a tromino and we place a tromino of the correct orientation on the grid.

If $n > 1$, we divide it into four $2^{n-1} \times 2^{n-1}$ subgrids. The missing grid cell is contained in exactly one of these $2^{n-1} \times 2^{n-1}$ subgrids. Assume w.l.o.g. that the cell is contained in the top-left subgrid; the other three cases are analogous. Then the top-left subgrid is a deficient grid and can be tiled recursively. We turn the bottom-left subgrid into a deficient grid by removing its top-right corner. Similarly, we remove the top-left corner from the bottom-right subgrid and the bottom-left corner from the top-right subgrid. This turns each of these three subgrids into a deficient grid that can be tiled recursively. Once we have tiled the deficient subgrids recursively, there are exacly three cells that are left uncovered: the top-right corner of the bottom-left subgrid, the top-left corner of the bottom-right subgrid, and the bottom-left corner of the top-right subgrid. However, these three cells together form a tromino, so we can obtain the final tiling by placing a tromino that covers these cells.

The correctness of the algorithm is obvious from this description. For the analysis, observe that each recursive call takes constant time. The number of recursive calls is given by the recurrence

$$R(n) = \begin{cases} 1 & n = 1 \\ 1 + 4R(n-1) & n > 1 \end{cases}.$$

Thus, for $n = 1$, $R(n) < 4^n - 1$. For $n > 2$, we have $R(n) = 1 + 4R(n-1) < 1 + 4 \cdot (4^{n-1} - 1) < 4^n - 1$. This shows that the algorithm takes $O(4^n)$ time.

Extra space for Question 3.2

Question 3.3 (Data structures) **6 marks**

Given a sequence $S = \langle x_1, \dots, x_n \rangle$, a *range minimum* query is given a pair of indices (i, j) such that $1 \le i \le j \le n$ and asks for the element $\min\{x_i, \dots, x_j\}$. Provide a data structure that supports the following operations:

- INSERT(S, i, x): Insert x between the ith element and the $(i+1)$st element of S. x becomes the new $(i+1)$st element and elements $x_{i+1}, \dots, x_n$ are shifted one position to the right. If $i = 0$, then x becomes the new first element of S. If $i = n$, then x becomes the new last element of S.
- DELETE(S, i): Remove the ith element from S. Elements $x_{i+1}, \dots, x_n$ are shifted one position to the left.
- RANGEMINIMUM(S, i, j): If the current sequence of elements in S is $\langle x_1, \dots, x_n \rangle$, then report the element $\min\{x_i, \dots, x_j\}$.

Each operation should take $O(\lg n)$ time. Argue that each operation on your data structure takes $O(\lg n)$ time and that the RANGEMINIMUM operation gives the correct answer. Remember that you are allowed to use data structures we discussed in class as building blocks.

The data structure is a rank-select tree T over the elements in S except that the elements aren't sorted; they are stored at the leaves in the order they appear in S. In addition, every node of T stores the minimum element stored at its descendant leaves.

Range-minimum query: *To answer a* RANGEMINIMUM(S, i, j) *query, we perform a* SELECT(T, i) *and a* SELECT(T, j) *operation. Let P_i and P_j be the two paths traversed by these two queries. We can split P_i into two subpaths P and Q_i such that the nodes in P belong to P_j and the nodes in Q_j do not. Then $P_j = P \circ Q_j$, where the nodes in Q_j are the nodes in P_j that are not in P_i. Observe that the elements in $S[i, j]$ are exactly the elements stored*

- *At the leaves at the bottom of the two paths P_i and P_j,*
- *At descendant leaves of the children of the last node of P that occur between the first nodes of Q_i and Q_j,*
- *At descendant leaves of all right siblings of nodes in Q_i except the first node, and*
- *At descendant leaves of all left siblings of nodes in Q_j except the first node.*

Each node stores the minimum element stored at its descendant leaves. Thus, we report the minimum of

- *The elements stored at the leaves in P_i and P_j,*
- *The minimum elements stored at the children of the last node of P that occur between the first nodes of Q_i and Q_j,*
- *The minimum elements stored at right siblings of all nodes in Q_i except the first node, and*
- *The minimum elements stored at left siblings of all nodes in Q_j except the first node.*

The two SELECT *queries take $O(\lg n)$ time and the two paths P_i and P_j have length $O(\lg n)$ and thus $O(\lg n)$ pendant nodes. Thus we are taking the minimum over $O(\lg n)$ values, in $O(\lg n)$ time. Overall, a* RANGEMINIMUM *query can be answered in $O(\lg n)$ time.*

Extra space for Question 3.3

Insert (w/o rebalancing): *If $i = 0$, we use a* MINIMUM(T) *operation to find the leftmost leaf in T and then add x as the new leftmost child of its parent. Otherwise, we use a* SELECT(T, i) *operation to find the ith leaf of T and then add x as its right sibling. This takes $O(\lg n)$ time. We update the subtree sizes as needed by a rank-select tree. In addition, we recompute the minimum of every ancestor of x as the minimum of the minima stored with all its children, bottom-up. Thus, we can support insertions in $O(\lg n)$ time plus the time needed to rebalance the tree.*

Delete (w/o rebalancing): *We use a* SELECT(T, i) *operation to find the ith leaf of T and delete it. This takes $O(\lg n)$ time. We update the subtree sizes as needed by a rank-select tree. In addition, we recompute the minimum of every ancestor of x as the minimum of the minima stored with all its children, bottom-up. Thus, we can support deletions in $O(\lg n)$ time plus the time needed to rebalance the tree.*

Node split: *When splitting a node, the subtree sizes and the minima to be stored with the newly created nodes can be computed from the subtree sizes and minima stored with their children. Thus, a node split takes constant time.*

Node fusion: *When fusing two nodes, the subtree size and the minimum to be stored with the newly created node can be computed from the subtree sizes and minima stored with its children. Thus, a node fusion takes constant time.*

Since an insertion performs up to $O(\lg n)$ node splits and a deletion peforms at most one node split and up to $O(\lg n)$ node fusions, this shows that insertions and deletions including rebalancing take $O(\lg n)$ time.