Banner number: Name:

# Practice Final Exam

## CSCI 3110: Design and Analysis of Algorithms

Fall 2015

| Group 1 | | Group 2 | | Group 3 | | |
|---|---|---|---|---|---|---|
| Question 1.1 | | Question 2.1 | | Question 3.1 | | $\sum$ |
| Question 1.2 | | Question 2.2 | | Question 3.2 | | |
| Question 1.3 | | Question 2.3 | | Question 3.3 | | |
| Question 1.4 | | Question 2.4 | | | | |
| $\sum$ | | $\sum$ | | $\sum$ | | |

---

**Instructions:**

- The questions are divided into three groups: Group 1 (40 marks = 40%), Group 2 (36 marks = 36%), and Group 3 (24 marks = 24%). You have to answer **all questions in Groups 1 and 2** and **exactly two question in Group 3**. In the above table, put check marks in the **small** boxes beside the two questions in Group 3 you want me to mark. If you select less than or more than two questions, I will randomly choose which two 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.**
- 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 13 pages, including this title page. Notify me immediately if your copy has fewer than 13 pages.**

**Question 1.1 (Average-case and worst-case running time)** **12 marks**

(a) Define the term "average-case running time of a deterministic algorithm".

*The average-case running time of a deterministic algorithm is a function $T(\cdot)$ such that $T(n)$ is the average running time of the algorithm over all possible inputs of size $n$.*

(b) Define the term "worst-case running time of a deterministic algorithm".

*The worst-case running time of a deterministic algorithm is a function $T(\cdot)$ such that $T(n)$ is the maximum running time of the algorithm over all possible inputs of size $n$.*

(c) Explain the difference between a deterministic algorithm and a randomized algorithm.

*The behaviour of a deterministic algorithm is completely determined by its input. A randomized algorithm on the other hand makes random choices during its computation. Thus, it may behave differently when run twice on the same input.*

(d) Explain the difference between the average-case running time of a deterministic algorithm and the expected running time of a randomized algorithm.

*Both running time bounds are expectations. In the case of the average-case running time of a deterministic algorithm, the expectation is taken over a probability distribution over the set of possible inputs. In the case of the expected running time of a randomized algorithm, the expectation is taken over a probability distribution over the random choices the algorithm makes*

## Question 1.2 (Asymptotic growth of functions) — 5 marks

(a) Give a formal definition of the set $\Theta(f(n))$.

*$\Theta(f(n))$ is a set of functions. It contains a function $g(n)$ if and only if there exist constants $c_1 > 0$, $c_2 > 0$, and $n_0 \geq 0$ such that for all $n \geq n_0$, $c_1 f(n) \leq g(n) \leq c_2 f(n)$.*

(b) Give a formal definition of the set $o(f(n))$.

*$o(f(n))$ is a set of functions. It contains a function $g(n)$ if and only if, for all $c > 0$, there exists a constant $n_0 \geq 0$ such that for all $n \geq n_0$, $g(n) \leq c f(n)$.*

## Question 1.3 (Amortized analysis) — 8 marks

Let $D$ be a data structure whose operations have an amortized cost of at most $t$.

(a) Can you give a guaranteed bound on the cost of a sequence of $m$ operations on $D$? If so, state this bound. Justify your answer.

*Yes. The cost of the sequence is at most $tm$. This is the very definition of amortized cost.*

(b) Can you give a guaranteed bound on the cost of an individual operation on $D$? If so, state this bound. Justify your answer.

*No. While an amortized cost of $t$ per operation guarantees that a sequence of $m$ operations never takes more than $tm$ time, for any $m$, an individual operation in this sequence can be substantially more costly.*

(c) If $t$ is not the amortized but the expected cost of operations on $D$, can you give any of the above guarantees? If so, which ones. Justify your answer.

*None of the above guarantees can be given because, even though we would expect the cost of an operation to be at most $t$, the cost of an operation may exceed $t$ and this may even be true for every single operation in a sequence of $m$ operations.*

## Question 1.4 (Complexity classes) **15 marks**

(a) Formally define the complexity class P.

*P is the class of all decision problems that can be solved in polynomial time, that is, the class of all formal languages that can be decided in polynomial time. Formally, a language $L$ belongs to $P$ if there exists a polynomial-time algorithm $A$ such that $A(x)$ answers "yes" if and only if $x \in L$.*

(b) Formally define the complexity class NP.

*NP is the class of all formal languages that can be verified in polynomial time. That is, a language $L$ belongs to NP if and only if there exists a language $L' \in P$ such that $x \in L$ if and only if there exists a $y \in \Sigma^*$ such that $(x, y) \in L'$ and $|y| \leq a|x|^c$, for some constants $a$ and $c$.*

(c) Define what it means for a problem to be NP-hard.

*A problem is NP-hard if a polynomial-time solution to this problem implies that $P = NP$.*

(d) Define what it means for a decision problem to be NP-complete.

*A decision problem is NP-complete if it is NP-hard and does in fact belong to NP.*

(e) Can an optimization problem be NP-complete? Justify your answer.

*No. Since NP contains only formal languages and every formal language gives rise to a decision problem, an optimization problem cannot be in NP. Thus, an optimization problem can be NP-hard but it cannot be NP-complete.*

## Question 2.1 (Sorting) — 6 marks

Provide an algorithm that can sort $n$ numbers in $O(n\sqrt{n})$ time. You are allowed to use any algorithms you know, even sorting algorithms, as building blocks without stating their details. To earn full marks, your algorithm description should be as simple as possible while giving a clear description of the algorithm.

*Use Merge Sort. This takes $O(n \lg n)$ time and thus satisfies the constraint that the algorithm should run in $O(n\sqrt{n})$ time.*

## Question 2.2 (Lower bounds and reductions) — 10 marks

Given that sorting using only comparisons requires $\Omega(n \lg n)$ time, prove that there cannot exist a comparison-based priority queue implementation that supports both INSERT and DELETEMIN operations in $o(\lg n)$ time. (If such a priority queue implementation existed, you should be able to design a sorting algorithm with running time $o(n \lg n)$.)

*Here's a simple sorting algorithm that uses a priority queue to accomplish its task: First insert the n elements into the priority queue. Then perform n* DELETEMIN *operations to remove the n elements in sorted order.*

*The running time of this algorithm is $O(n + n \cdot t_i(n) + n \cdot t_d(n))$, where $t_i(n)$ is the cost of an* INSERT *operation and $t_d(n)$ is the cost of a* DELETEMIN *operation. If $t_i(n) \in o(\lg n)$ and $t_d(n) \in o(\lg n)$, then the algorithm would have running time $o(n \lg n)$. Since the priority queue uses only comparisons to implement its operations, we would thus have a comparison-based sorting algorithm that violates the lower bound of $\Omega(n \lg n)$. Thus, we must have $t_i(n) \in \Omega(\lg n)$ or $t_d(n) \in \Omega(\lg n)$.*

## Question 2.3 (Kruskal's algorithm) 10 marks

Given a connected graph $G$ such that no two edges of $G$ have the same weight, prove that Kruskal's algorithm finds a minimum spanning tree of $G$. You may refer to the high-level description of the algorithm:

KRUSKAL($G$)

1 $T$ = a graph with the same vertex set as $G$ and no edges
2 **while** $T$ is not connected
3 let $e$ be the edge in $G$ with minimum weight among all edges whose endpoints belong to different components of $T$
4 add edge $e$ to $T$
5 **return** $T$

*We need to prove that the output $T$ of the algorithm is a tree, that is, is connected and contains no cycle and that its weight is minimal among all spanning trees of $G$.*

***T is connected:*** *The while loop terminates only once $T$ is connected and it must terminate eventually. To see this observe that, as long as $T$ is not connected, $G$ has an edge whose endpoints belong to different connected components of $T$. Thus, every iteration adds an edge of $G$ to $T$. In the worst case, we add all edges of $G$ to $T$, at which point $T$ must be connected and the while loop terminates.*

***T has no cycle.*** *Assume it does contain a cycle $C$ and let $e$ be the last edge in $C$ added to $T$. Then, just before adding $e$ to $T$, both endpoints of $e$ already belong to the same connected component of $T$. This contradicts the fact that we add $e$ to $T$.*

***T has minimum weight.*** *We use the Cut Theorem to prove this. Consider an edge $e = (u, w)$ added to $T$, and let $U$ be the vertex set of the connected component of $T$ containing $u$ at the time we add $e$ to $T$. Let $W := V \setminus U$. Then every edge with one endpoint in $U$ and one in $W$ has its endpoints in different components of $T$ and thus has a greater weight than $e$, by the choice of $e$. Since $e$ is the unique lightest edge with one endpoint in $U$ and one in $W$, the Cut Theorem states that $e$ must belong to every MST of $G$. Since this is true for every edge $e$ we add to $T$, $T$ cannot have a greater weight than any MST of $G$, that is, $T$ is itself an MST.*

## Question 2.4 (Recurrence relations) 10 marks

Solve the following two recurrences, that is, determine a function $f(n)$ for each recurrence such that $T(n) \in \Theta(f(n))$. Use the Master Theorem to solve one of them and induction to solve the other one.

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

*Since $n \lg n \in O(n^{\log_3 4 - \epsilon})$ for any $0 < \epsilon < \log_3 4 - 1$ and such an $\epsilon$ exists because $\log_3 4 > 1$, the Master Theorem states that $T(n) \in \Theta(n^{\log_3 4})$.*

(b) $T(n) = T(n/5) + T(5n/7) + \Theta(n)$

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

*$T(n) \in \Omega(n)$ follows immediately because $T(n/5) \geq 0$ and $T(5n/7) \geq 0$, for all $n \geq 0$, and $T(n) = T(n/5) + T(5n/7) + \Theta(n)$.*

*To prove that $T(n) \in O(n)$, that is, $T(n) \leq cn$, for some $c > 0$ and $n_0 \geq 0$ and for all $n \geq n_0$, we use induction on $n$.*

*The base case is $1 \leq n < 5$. In this case, $T(n) \in \Theta(1)$, so $T(n) \leq cn$, for $c$ sufficiently large.*

*For the inductive step, $n \geq 5$, we observe that $1 \leq n/5 < n$ and $1 \leq 5n/7 < n$, so we can apply the inductive hypothesis to $T(n/5)$ and $T(5n/7)$. This gives*

$$
\begin{aligned}
T(n) &\leq T(n/5) + T(5n/7) + dn && \text{where } d > 0 \text{ is an appropriate constant} \\
&\leq cn/5 + 5cn/7 + dn && \text{by the inductive hypothesis} \\
&\leq \left(\frac{7+25}{35}c + d\right)n \\
&= \left(\frac{32}{35}c + d\right)n \\
&\leq cn && \text{as long as } 3c/35 \geq d, \text{ that is, } c \geq 35d/3.
\end{aligned}
$$

*Thus, $T(n) \leq cn$ for all $n \geq 1$ as long as $c \geq 35d/3$ and $c$ is large enough to make the base case work.*

## Question 3.1 (Divide and conquer) — 12 marks

Let A be an array containing $n$ numbers (positive and negative). Develop an algorithm that finds the two indices $1 \leq i \leq j \leq n$ such that $S_{ij} := \sum_{k=i}^{j} A[k]$ is maximized. For example, in the array $A = [10, -12, 5, 7, -2, 4, -11]$, the sub-array $A[3,6]$ has the sum $S_{3,6} = 5 + 7 - 2 + 4 = 14$ and no other sub-array contains elements that sum to a value greater than 14, so for this input the algorithm should output $(3,6)$. The running time of your algorithm should be $O(n \lg n)$. Justify briefly why your algorithm achieves this running time and why it gives the correct answer.

*We use a divide and conquer algorithm to find the subarray $A[i,j]$ of $A[l,r]$ that maximizes $S_{ij}$. The top-level invocation then has parameters $l = 1$ and $r = n$.*

*If $l = r$, the invocation simply returns $i = j = l$ because this is the only non-empty subarray of $A[l,r]$.*

*If $l < r$, then let $m = \lceil (l+r)/2 \rceil$. The subarray $A[i,j]$ of $A[l,r]$ that maximizes $S_{ij}$ is of one of three forms:*

- *If $j < m$, then it's a subarray of $A[l, m-1]$ and is in fact the subarray of $A[l, m-1]$ that maximizes $S_{ij}$. Thus, a recursive call with parameters $l$ and $m-1$ finds this pair $(i,j)$.*
- *If $i \geq m$, then it's a subarray of $A[m,r]$ and is in fact he subarray of $A[m,r]$ that maximizes $S_{ij}$. Thus, a recursive call with parameters $l = m$ and $r$ finds this pair $(i,j)$.*
- *If $i < m \leq j$, then $A[i, m-1]$ maximizes $S_{ij}$ among all subarrays with $l \leq i < m$ and $j = m-1$ and $A[m,j]$ maximizes $S_{ij}$ among all subarrays with $l = m$ and $m \leq j \leq r$. Indeed, if there was, for example an index $l \leq i' < m$ such that $i' \neq i$ and $S_{i',m-1} > S_{i,m-1}$, then $S_{i',j} = S_{i',m-1} + S_{m,j} > S_{i,m-1} + S_{m,j} = S_{i,j}$, a contradiction because we assumed that $A[i,j]$ maximizes $S_{i,j}$.*

*This leads to the following algorithm.*

**Extra space for Question 3.1**

```
MAXSUBINTERVAL(A, l, r)
 1  if l = r
 2       return (l, r, A[l])
 3  m = ⌈(l + m)/2⌉
 4  (i_l, j_l, S_l) = MAXSUBINTERVAL(A, l, m − 1)
 5  (i_r, j_r, S_r) = MAXSUBINTERVAL(A, m, r)
 6  S = 0
 7  S'_l = 0
 8  i_m = m
 9  for i = m − 1 downto l
10       S = S + A[i]
11       if S > S'_l
12            S'_l = S
13            i_m = i
14  S = 0
15  S'_r = 0
16  j_m = m
17  for i = m to r
18       S = S + A[i]
19       if S > S'_r
20            S'_r = S
21            j_m = i
22  i = i_m
23  j = j_m
24  S = S'_l + S'_r
25  if S_l > S
26       i = i_l
27       j = j_l
28       S = S_l
29  if S_r > S
30       i = i_r
31       j = j_r
32       S = S_r
33  return (i, j, S)
```

*Apart from the two recursive calls with input sizes $T(\lceil n/2 \rceil)$ and $T(\lfloor n/2 \rfloor)$, each invocation spends linear time to iterate over the elements in $A[l, r]$ in lines 6–21 and constant time in lines 22–32 to choose the best possible solution among the three possible cases we discussed before. Thus, its running time is $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$.*

**Question 3.2 (Dynamic programming)** **12 marks**

Let $S$ be a sequence of $m$ integer pairs $\langle (x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m) \rangle$. Each of the values $x_i$ and $y_i$, for all $1 \leq i \leq m$, is an integer between 1 and $n$. A *domino sequence* is a subsequence $\langle (x_{i_1}, y_{i_1}), (x_{i_2}, y_{i_2}), \ldots, (x_{i_t}, y_{i_t}) \rangle$ such that $1 \leq i_1 < i_2 < \cdots < i_t \leq n$ and, for all $1 \leq j < t$, $y_{i_j} = x_{i_{j+1}}$. Note that it isn't necessarily true that $i_{j+1} = i_j + 1$, that is, the elements of the domino sequence don't have to be consecutive in $S$, but they have to appear in the right order.

Example: For $S = \langle (1,3), (4,2), (3,5), (2,3), (3,8) \rangle$, both $\langle (1,3), (3,5) \rangle$ and $\langle (4,2), (2,3), (3,8) \rangle$ are domino sequences.

Use dynamic programming to find a longest domino sequence of $S$ in $O(n+m)$ time. Argue briefly that the running time of your algorithm is indeed $O(n+m)$ and that its output is indeed a longest domino sequence of $S$.

*We build two tables $L[1..n]$ and $P[1..m]$. $L[y]$ is the length of the longest domino sequence whose last domino $(x_j, y_j)$ satisfies $y_j = y$. $P[i]$ is the predecessor of $(x_i, y_i)$ in the longest domino sequence that has $(x_i, y_i)$ as its last domino.*

*We build these tables iteratively. Let $L_i$ denote the state of $L$ after the $i$th iteration. In this case, we want that $L_i[y]$ is the length of the longest domino sequence in $\langle (x_1, y_1), (x_2, y_2), \ldots, (x_i, y_i) \rangle$ whose last domino $(x_j, y_j)$ satisfies $y_j = y$.*

*Then we have $L_0[y] = 0$ for all $1 \leq y \leq n$, so we initialize all entries in $L$ to 0, which takes $O(n)$ time.*

*For $i > 0$, observe that*

$$L_i[y] = \begin{cases} L_{i-1}[y] & \text{if } y_i \neq y \\ \max(L_{i-1}[y], L_{i-1}[x_i] + 1) & \text{if } y_i = y \end{cases}.$$

*The two cases when $L_i[y] = L_{i-1}[y]$ correspond to the case when the longest domino sequence ending in $y$ is completely contained in $\langle (x_1, y_1), (x_2, y_2), \ldots, (x_{i-1}, y_{i-1}) \rangle$. If this sequence is not contained in $\langle (x_1, y_1), (x_2, y_2), \ldots, (x_{i-1}, y_{i-1}) \rangle$, then its last domino must be $(x_i, y_i)$, so $y_i = y$ and the subsequence obtained by removing $(x_i, y_i)$ must be a longest domino sequence in $\langle (x_1, y_1), (x_2, y_2), \ldots, (x_{i-1}, y_{i-1}) \rangle$ ending in $x_i$.*

*Computing $L_i$ from $L_{i-1}$ takes constant time because the only entry that needs to be changed is $L[y_i]$. Since computing the final table $L = L_m$ requires $m$ such iterations and each iteration takes constant time, we can thus compute $L$ in $O(n+m)$ time.*

*To build the table $P$, we need to construct a third table $F[1..n]$ along with $L$. $F[y]$ is the index $j$ of the last domino $(x_j, y_j)$ in the longest domino sequence ending in $y$. Again, we refer to the state of $F$ after the $i$th iteration as $F_i$. Then $F_0[y] = 0$ for all $1 \leq y \leq n$. For $i > 0$, we set $F_i[y] = F_{i-1}[y]$ for all $y \neq y_i$. For $y = y_i$, we set $F_i[y_i] = F_{i-1}[y_{i-1}]$ if $L_i[y_i] = L_{i-1}[y_i]$. Otherwise, we set $F_i[y_i] = i$. This computation of $F$ can be incorporated in the computation of $L$ without increasing the cost by more than a constant factor, so computing $L$ and $F$ takes $O(n+m)$ time.*

*Now, given $F$, we can easily augment each iteration so it also computes $P[i]$: $P[i] = F[x_i]$. Once again, the cost of each iteration is increased by only a constant, so the total cost of computing $L$, $F$, and $P$ is $O(n+m)$.*

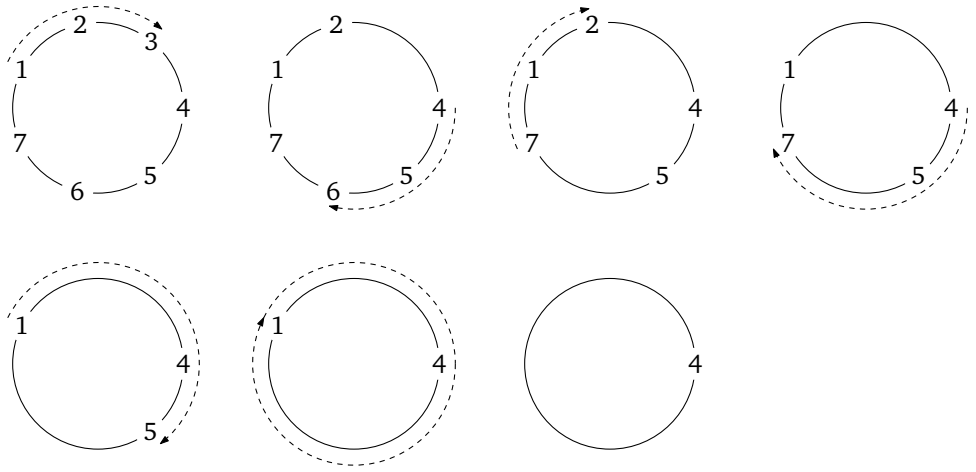
**Extra space for Question 3.2**

*All that's left now is extracting a longest domino sequence, in $O(n+m)$ time.*

*First we scan $L$ to find the index $y$ such that $L[y] = \max_{1 \le i \le n} L[i]$. This is the length of the longest domino sequence because such a sequence must end in some domino. $F[y]$ now stores the index $j$ of the last domino of this sequence. We produce our result sequence $R$ using the following loop: Initially, we set $R = \emptyset$. While $j \ge 1$, we prepend the domino $(x_j, y_j)$ to $R$ and set $j = P[j]$. The cost per iteration is constant. The number of iterations is at most $m$ because $j$ decreases by at least one in each iteration. Thus, reporting $R$ takes $O(n+m)$ time.*

## Question 3.3 (Applications of data structures) 12 marks

Assume $n$ people form a circle and number them in the order they appear around the circle. Now, for some integer $1 \leq m \leq n$, we repeat the following process, starting with person 1, until no people are left in the circle: Walk around the circle until reaching the $m$th person after the person we started at. Remove this person from the circle and continue using her successor as the starting point of the next search. The order in which we remove people from the circle is called the $(n, m)$-*Josephus permutation*. The following example illustrates that the $(7, 3)$-Josephus permutation is $\langle 3, 6, 2, 7, 5, 1, 4 \rangle$.




Develop an algorithm that can find the $(n, m)$-Josephus permutation in $O(n \lg n)$ time for any pair $(n, m)$ such that $1 \leq m \leq n$. Justify briefly why your algorithm achieves this running time and why it gives the correct answer.

*Consider the set* $\{1, 2, \ldots, n\}$. *The first element in the* $(n, m)$*-Josephus permutation is the* $m$*th smallest element. The second element is the* $(2m)$*th smallest element, which is also the* $(2m-1)$*st smallest element of the set obtained after removing the* $m$*th smallest element. The third element is the* $(3m-2)$*nd smallest element of the set obtained after removing the first two elements. We could define the* $(n, m)$*-Josephus permutation in this fashion if it was not for the small problem that the set from which we want to choose the* $j$*th element (which is the* $(jm - j + 1)$*st smallest element in this set) has only* $n - j + 1$ *elements in it and it may be that* $n - j + 1 < jm - j + 1$. *This is exactly the situation we encounter when we complete a full circle and start counting elements from the beginning of the circle again. Thus, we do not choose the* $(jm - j + 1)$*st element in the remaining set but the* $k_j$*th smallest element, where* $k_j = (jm - j + 1) \bmod (n - j + 1)$. *All we need now is a data structure that allows us to efficiently maintain the set of remaining elements and choose the* $k_j$*th smallest element in each iteration. Of course it is exactly a dynamic order statistics data structure that allows us to solve this problem. This data structure supports* INSERT, DELETE, *and* SELECT *operations in* $O(\lg n)$ *time per operation. A* SELECT$(k)$ *operation returns the* $k$*th smallest element in the data structure. Using this data structure, we obtain the following simple algorithm for computing the* $(n, m)$*-Josephus permutation.*

**Extra space for Question 3.3**

JOSEPHUS$(n, m)$

```
 1  D = an empty order statistics data structure
 2  for i = 1 to n
 3      INSERT(D, i)
 4  k = 0
 5  n' = n
 6  for i = 1 to n
 7      k = (k + m − 1) mod n'
 8      Y[i] = SELECT(D, k + 1)
 9      DELETE(D, Y[i])
10      n' = n' − 1
11  return Y
```

*This algorithm performs n insertions into D, n deletions from D, and n* SELECT *queries on D. Apart from that, it clearly takes linear time. Thus, its cost is* $O(n \lg n)$.