Banner number: Name:

Final Exam

CSCI 3110: Design and Analysis of Algorithms

August 5, 2015

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 (36%), Group 2 (40%), 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 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. Keey 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 15 pages, including this title page. Notify me immediately if your copy has fewer than 15 pages.**

Question 1.1 (Worst-case and average-case running time, randomization) 9 marks

(a) Define what the worst-case running time of a deterministic algorithm is.

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

(b) Define what the average-case running time of a deterministic algorithm is.

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

(c) The average-case running time of a deterministic algorithm and the expected running time of a randomized algorithm are both expectations. Explain the difference between the two. What do these two performance measures say about the real performance of an algorithm in an application where little is known about the distribution of possible inputs?

The average-case running time of a deterministic algorithm is an expectation over the probability distribution of all posible inputs. The expected running time of a randomized algorithm is an expectation over the random choices the algorithm makes while it runs. In an application where the input distribution is unknown, this distribution may differ greatly from the one assumed in the average-case analysis. Thus, the average-case running time may be a poor predictor of the actual running time observed on most inputs in the application. The expected running time of the randomized algorithm is based on a known *distribution of the random choices the algorithm makes and is completely independent of the input distribution. Thus, on average, the algorithm is expected to exhibit this running time no matter what the input distribution is.*

Question 1.2 (Asymptotic growth of functions) **9 marks**

(a) Using the definitions of O, Ω, and Θ, prove that $f(n) \in \Theta(g(n))$ if and only if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

Here are the three definitions:

$$
\begin{aligned}
f(n) \in \Omega(g(n)) &\Leftrightarrow \exists c_1 > 0, n_1 \forall n \geq n_1 : f(n) \geq c_1 \cdot g(n) \\
f(n) \in O(g(n)) &\Leftrightarrow \exists c_2 > 0, n_2 \forall n \geq n_2 : f(n) \leq c_2 \cdot g(n) \\
f(n) \in \Theta(g(n)) &\Leftrightarrow \exists c_1 > 0, c_2 > 0, n_0 \forall n \geq n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)
\end{aligned}
$$

Thus, if $f(n) \in \Theta(g(n))$, then $f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$, that is, $f(n) \in O(g(n))$. Similarly, if $f(n) \in \Theta(g(n))$, then $f(n) \geq c_1 \cdot g(n)$ for all $n \geq n_0$, that is, $f(n) \in \Omega(g(n))$.

Finally, if $f(n) \in \Omega(g(n))$ and $f(n) \in O(g(n))$, then $c_1 \cdot g(n) \leq f(n)$ and $f(n) \leq c_2 \cdot g(n)$ for all $n \geq \max(n_1, n_2)$, that is, $f(n) \in \Theta(g(n))$.

(b) If a function $f(n)$ is not in $\Omega(g(n))$, does that mean that $f(n) \in o(g(n))$? If so, prove it. If not, provide two functions $f(n)$ and $g(n)$ such that $f(n)$ is neither in $\Omega(g(n))$ nor in $o(g(n))$ and prove that this is the case.

Consider the functions

$$
f(n) = \begin{cases} n & n \text{ is even} \\ 1 & n \text{ is odd} \end{cases}
$$

and $g(n) = n$.

Then $f(n) \notin o(g(n))$ because for $c = 1/2$, there exists no n_0 such that $f(n) \leq n/2$ for all $n \geq n_0$: for all even n, we have $f(n) = g(n)$.

Similarly, $f(n) \notin \Omega(g(n))$ because for any $c > 0$, and $n_0 = \frac{1}{c} + 1$, we have $f(n) = 1 < cn = c \cdot g(n)$ for all odd $n \geq n_0$.

(c) Using the definition of Θ-notation, prove that $n^3 + n^2 \lg n - 10n \in \Theta(n^3)$.

First the upper bound:

$$\begin{aligned} n^3 + n^2 \lg n - 10n &\le n^3 + n^2 \lg n && \textit{for all } n \ge 0 \\ &\le n^3 + n^3 && \textit{for all } n \textit{ because } \lg n < n \textit{ for all } n \\ &= 2n^3. \end{aligned}$$

Thus, $n^3 + n^2 \lg n - 10n \le 2n^3$ *for all* $n \ge 0$.

Next the lower bound:

$$\begin{aligned} n^3 + n^2 \lg n - 10n &\ge n^3 - 10n && \textit{for all } n \ge 1 \textit{ because } n^2 \lg n \ge 0 \textit{ in this case} \\ &\ge n^3 - \frac{n^3}{2} && \textit{for all } n^2/2 \ge 10\textit{, which holds for } n \ge 5 \\ &= \frac{n^3}{2}. \end{aligned}$$

Thus, $n^3 + n^2 \lg n - 10n \ge n^3/2$ *for all* $n \ge 5$.

Putting both inequalities together, we have $n^3/2 \le n^3 + n^2 \lg n - 10n \le 2n^3$ *for all* $n \ge \max(0, 5) = 5$*, that is,* $n^3 + n^2 \lg n - 10n \in \Theta(n^3)$.

Question 1.3 (Complexity classes) 9 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 belongs to P if there exists a polynomial-time algorithm A such that $A(x)$ answers "yes" for any string $x \in \Sigma^*$ 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. Formally, a language L belongs to NP if there exist a polynomial-time algorithm A and a constant c with the property that any string $x \in \Sigma^*$ belongs to L if and only if there exists a string $y \in \Sigma^*$ of length $|y| \in O(|x|^c)$ such that $A(x, y)$ answers "yes".

(c) Formally define what a polynomial-time reduction is.

A polynomial-time reduction from a language L_1 to a language L_2 is a polynomial-time algorithm A whose output $A(x)$ given input x belongs to L_2 if and only if x belongs to L_1.

(d) Formally define what it means for a language to be NP-hard.

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

(e) Prove that, if there exists a polynomial-time reduction from an NP-hard language L_1 to a language L_2, then L_2 is also NP-hard.

First observe that a polynomial-time reduction R from L_1 to L_2 implies that $L_1 \in P$ if $L_2 \in P$: Since R runs in time $O(|x|^c)$ on any input $x \in \Sigma^*$, its output has size $|x| + O(|x|^c) \in O(|x|^c)$. Since $L_2 \in P$, we have an algorithm D that takes time $O(|y|^d)$ on any input string y and decides whether $y \in L_2$. Since $R(x) \in L_2$ if and only if $x \in L_1$, we can thus decide whether $x \in L_1$ by applying D to $R(x)$. This takes $O(|R(x)|^d) = O(|x|^{cd})$ time. The total cost of this decision algorithm for L_1 is the cost of running R plus the cost of running D on $R(x)$, which is $O(|x|^c) + O(|x|^{cd}) \in O(|x|^{cd})$. Thus, $L_1 \in P$.

Now, since $L_2 \in P$ implies that $L_1 \in P$ and $L_1 \in P$ implies that $P = NP$ because L_1 is NP-hard, then $L_2 \in P$ implies that $P = NP$, that is, L_2 is NP-hard.

Question 2.1 (Recurrence relations) **10 marks**

Given two integers $a \geq b$, their greatest common divisor $gcd(a,b)$ is the greatest integer c that divides both a and b. Euclid's algorithm computes $gcd(a,b)$:

```
GCD(a, b)
1  if b = 0
2      then return a
3      else return GCD(b, a mod b)
```

Here, $a \bmod b$ denotes the remainder that is left when dividing a by b: $a \bmod b = a - b \cdot \lfloor a/b \rfloor$.

Prove that the running time of the algorithm is $O(\lg b)$, where $\lg x := \max(1, \log_2 x)$. (Hint: First prove that the running time is $O(\lg n)$, where $n = a + b$; show that $a + b$ decreases by a constant factor from one recursive call to the next. Then argue that the first recursive call made by the top-level invocation GCD(a,b) satisfies $n < 2b$.)

First observe that, for $n \leq 2$, the algorithm runs in constant time because either it returns immediately because $b = 0$ or we have $a = b = 1$, in which case the recursive call GCD$(1,0)$ *returns immediately. Thus, $T(n) \leq c \lg n$ for c sufficiently large.*

For $n > 2$, observe that the invocation GCD(a,b) *makes at most one recursive call* GCD$(b, a \bmod b)$ *and apart from that takes constant time. Thus, $T(n) = T(a+b) \leq T(b + (a \bmod b)) + d$. Now it suffices to prove that $b + (a \bmod b) \leq 2(a+b)/3$. To prove this, observe that $b + (a \bmod b) \leq a$ because $a \geq b$. Thus, if $b \geq a/2$, then $a + b \geq \frac{3}{2}a \geq \frac{3}{2}(b + (a \bmod b))$. If $b \leq a/2$, then observe that $a \bmod b < b$, so $b + (a \bmod b) < 2b$ while $a + b \geq 2b + b = 3b$, so again $a + b \geq \frac{3}{2}(b + (a \bmod b))$. This gives $T(n) \leq T(2n/3) + d$.*

Using the inductive hypothesis, we now obtain $T(n) \leq c \lg(2n/3) + d = c \log_2(2n/3) + d = c \log_2 n - c \log_2(3/2) + d \leq c \lg n$ as long as we choose $c \geq d / \lg(3/2)$.

Now, to finish the proof we already observed above that $a \bmod b < b$. That is, for the first recursive call made by the top-level invocation GCD(a,b)*, we have $n = b + (a \bmod b) < 2b$. As we have just shown, the running time of the call* GCD$(b, a \bmod b)$ *is thus $O(\lg n) = O(\lg(2b)) = O(\lg b)$. Adding the constant cost of the top-level call to this, we obtain a total running time of $O(1) + O(\lg b) = O(\lg b)$.*

Question 2.2 (Correctness proofs) **10 marks**

Here's your standard binary search algorithm for a sorted array A:

```
BinarySearch(A, i, j, x)
1  if j < i
2     then return no
3  m ← ⌊(i+j)/2⌋
4  if A[m] = x
5     then return yes
6     else if A[m] > x
7             then return BinarySearch(A, i, m − 1, x)
8             else return BinarySearch(A, m + 1, j, x)
```

To find an element x in array A, you'd call $\textsc{BinarySearch}(A, 1, n, x)$. Prove that $\textsc{BinarySearch}(A, 1, n, x)$ returns yes if and only if $x \in A$.

We prove by induction on $s = j - i + 1$ that the invocation $\textsc{BinarySearch}(A, i, j, x)$ returns yes if and only if $x \in A[i\,..\,j]$. Applying this to $i = 1$ and $j = n$ proves the claim.

For $s = 0$, the array $A[i\,..\,j]$ is empty, so $x \notin A[i\,..\,j]$. Since $j < i$ in this case, the algorithm correctly answers no.

For $s > 1$, observe that $i \le \lfloor \frac{i+j}{2} \rfloor \le j$, that is, $i \le m \le j$. Thus, if $A[m] = x$, then $x \in A[i\,..\,j]$, that is, the algorithm correctly answers yes in this case.

If $A[m] \ne x$, then either $A[m] > x$ or $A[m] < x$.

If $A[m] > x$, then $x \notin A[m\,..\,j]$ because A is sorted. Thus, $x \in A[i\,..\,j]$ if and only if $x \in A[i\,..\,m-1]$, that is, it suffices to decide whether $x \in A[i\,..\,m-1]$. Since $m \le j$ and thus $m-1 < j$, we have $m-1-i+1 < j-i+1 = s$, that is, by the inductive hypothesis, the recursive call $\textsc{BinarySearch}(A, i, m-1, x)$ correctly answers whether $x \in A[i\,..\,m-1]$.

If $A[m] < x$, an analogous argument shows that $x \in A[i\,..\,j]$ if and only if $x \in A[m+1\,..\,j]$ because $x \notin A[i\,..\,m]$, and that the recursive call $\textsc{BinarySearch}(A, m+1, j, x)$ correctly decides whether $x \in A[m+1\,..\,j]$.

Thus, in each of these three cases, the invocation $\textsc{BinarySearch}(A, i, j, x)$ correctly decides whether $x \in A[i\,..\,j]$.

Question 2.3 (Amortized analysis) 10 marks

Recall the trusty old queue data structure. ENQUEUE(Q, x) appends element x to the end of the queue. DEQUEUE(Q) removes the frontmost element from Q and returns it. This structure is easy to implement using a doubly-linked list or a singly-linked list with an additional pointer to the tail of the list. Both implementations support ENQUEUE and DEQUEUE operations in constant time but neither can be implemented purely functionally. Doubly-linked lists are impossible to implement purely functionally. Functional singly-linked lists support only the following operations: access the first element in the list, prepend a new element to the list, remove the first element from the list, and query whether the list is empty. Each such operation takes constant time. A simple functional queue implementation now consists of two singly linked lists F and B: $Q = (F, B)$. An ENQUEUE(Q, x) operation prepends x to B. A DEQUEUE(Q) operation tests whether F is empty. If F is not empty, it removes the first element from F and returns it. If F is empty, it tests whether B is empty. If B is also empty, no element is returned because the queue is empty. If B is non-empty, then the DEQUEUE operation first reverses B, which is easily done using the following pseudo-code:

```
REVERSE(L)
1  R ← an empty list
2  while L is not empty
3       do Remove the first element from L and prepend it to R
4  return R
```

It then replaces F with this reversed copy of B, sets B to be the empty list, and finally continues as in the case when $F \neq \emptyset$ (which is now the case). Prove that the amortized cost per ENQUEUE and DEQUEUE operation on this data structure is constant.

It suffices to define a potential function that ensures that the potential of the queue is never negative and that the actual cost of an operation plus the resulting change of the potential of the queue is constant.

A potential function that satisfies this property is the size of B. Since B's length is non-negative, so is the potential. Now consider the two operations.

An ENQUEUE(Q, x) *operation has constant cost because all it does is prepend x to B, a constant-time operation. It also increases the length of B by one, so the potential increases by one. The amortized cost is thus constant (constant actual cost plus constant increase in potential).*

For a DEQUEUE(Q) *operation, we distinguish whether or not F is empty.*

If $F \neq \emptyset$, then the DEQUEUE *operation takes constant time to remove the front element from F and it does not change the potential because it does not alter B. Thus, its amortized cost is constant.*

If $F = \emptyset$, it first reverses B and then does the same as when $F \neq \emptyset$. Reversing B has cost $O(|B|)$. Since the reversed B becomes the new F and B becomes empty, this also decreases the potential by B, so the amortized cost of reversing B is 0! Adding the cost of removing the frontmost element from the new F, we once again obtain constant amortized cost.

Question 3.1 (Divide and conquer) **9 marks**

Consider an array A storing n numbers and consider the problem of finding the kth smallest element in A. In this question, we measure the complexity of an algorithm not in terms of the number of operations it performs but in terms of the number of comparisons it performs; we do not care about the number of other operations it performs, such as additions, memory accesses, etc. To be precise, even comparisons are counted only if at least one of the compared elements is an element of A; comparisons between array indices, for example when evaluating loop conditions, are not counted.

Finding the minimum element in A using $n-1$ comparisons is easy. We also showed in class that we can find the kth smallest element in A in $O(n)$ time and thus using $O(n)$ comparisons. Now let's care about constant factors and let's focus on finding the second-smallest element in A. This can easily be done using $2n-3$ comparisons. This question asks you to do better by using divide and conquer. In particular, you are asked to find the second-smallest element in A using only $n+\lceil \lg n \rceil - 2$ comparisons. Argue briefly why your algorithm does indeed return the second-smallest element and why it performs $n + \lceil \lg n \rceil - 2$ comparisons.

Hint: Think about how to find the minimum using divide and conquer rather than using a straight scan of the array A. By keeping track of the outcomes of the comparisons performed by this algorithm, you should be able to narrow down the search for the second-smallest element to a very small candidate set.

First we use a divide-and-conquer algorithm to find the minimum. The minimum element in a subarray $A[i..j]$ is of course the smaller of the minimum of the subarray $A[i..k]$ and the minimum of the subarray $A[k+1..j]$, for any $i \le k < j$. In order to aid finding the second-smallest element, we do not only return the minimum m but a pair (m, l), where l is the list of elements m was compared to. As we'll argue below, l must contain the second-smallest element and $|l| \le \lceil \lg n \rceil$, so we simply report the minimum of l as the final result.

```
MINIMUM(A, i, j)
1  if i = j
2     then return (A[i], ∅)
3     else  k ← ⌊(i+j)/2⌋
4           (m1, l1) ← MINIMUM(A, i, k)
5           (m2, l2) ← MINIMUM(A, k + 1, j)
6           if m1 < m2
7              then return (m1, ⟨m2⟩ ∘ l1)
8              else  return (m2, ⟨m1⟩ ∘ l2)
```

This algorithm performs $n-1$ comparisons, which we prove by induction: For $n = 1$, we perform $0 = n-1$ comparisons because $n = j - i + 1$, that is, $i = j$, so the algorithm returns in line 2. For $n > 1$, the two recursive calls perform $k - i$ and $j - (k+1)$ comparisons, respectively, by the induction hypothesis. The current invocation performs one additional comparison in line 6. So the total number of comparisons is $k - i + (j - (k+1)) + 1 = j - i = n - 1$.

To see that the list l contains the second smallest element in A, where $(m, l) = \text{Minimum}(A, 1, n)$, we use induction again. For $n = 1$, $l = \emptyset$ and contains the second-smallest element because there is no second-smallest element. For $n > 1$, let (m_1, l_1) and (m_2, l_2) be the pairs returned by the two recursive calls and assume w.l.o.g. that $m_1 < m_2$. Let m'_1 be the second-smallest element in $A[i..k]$. We observe that the second smallest element m' in $A[i..j]$ must be one of m'_1 or m_2: If $m' \in A[k+1..j]$ and $m' \neq m_2$, then $m' > m_2 > m_1$, so it is not *the second smallest element in $A[i..j]$, a contradiction. Similarly,*

Extra space for Question 3.1

if $m' \in A[i\,..\,k]$ and $m' \neq m'_1$, then $m' > m'_1 > m_1$, so again it is not the second smallest element in $A[i\,..\,j]$. Since $m'_1 \in l_1$, by the inductive hypothesis, this implies that m' belongs to the list $\langle m_2 \rangle \circ l_1$ we return.

Since the second-smallest element of A is in l (and is easily seen to be the smallest element in l), we run Minimum *again on l. As we argued above, this takes $|l| - 1$ comparisons. In total, the algorithm therefore performs $n - 1 + |l| - 1 = n + |l| - 2$ comparisons. It thus suffices to prove that $|l| \leq \lceil \lg n \rceil$.*

We again use induction on n to prove this. For $n = 1$, we have $|l| = 0 = \lceil \lg n \rceil$ because $i = j$ in this case, so the result of the call Minimum(A, i, j) *is returned in line 2. For $n > 1$, we have $|l| \leq 1 + \max(|l_1|, |l_2|)$, where l_1 and l_2 are the lists returned by the two recursive calls the invocation makes. The input sizes of these two recursive calls are $n_1 = \lceil n/2 \rceil$ and $n_2 = \lfloor n/2 \rfloor$. Thus, by the inductive hypothesis, $|l| \leq 1 + \lceil \lg \lceil n/2 \rceil \rceil$. Now let n' be the smallest power of 2 no less than n. Then $\lceil n/2 \rceil \leq n'/2$ and $\lceil \lg n \rceil = \lceil \lg n' \rceil$. Thus, $1 + \lceil \lg \lceil n/2 \rceil \rceil \leq 1 + \lceil \lg(n'/2) \rceil = 1 + \lceil \lg n' - 1 \rceil = \lceil \lg n' \rceil = \lceil \lg n \rceil$.*

Question 3.2 (Dynamic programming) **9 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 $1 \le i \le m$, 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 $1 \le i_1 < i_2 < \cdots < i_t \le n$ and, for all $1 \le j < t$, $y_{i_j} = x_{i_{j+1}}$. 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 ith 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 $1 \le y \le n$, so we initialize all entries in L to 0, which takes $O(n)$ time.

For $i > 0$, observe that

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

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 ith iteration as F_i. Then $F_0[y] = 0$ for all $1 \le y \le n$. For $i > 0$, we set $F_i[y] = F_{i-1}[y]$ for all $y \ne y_i$. 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

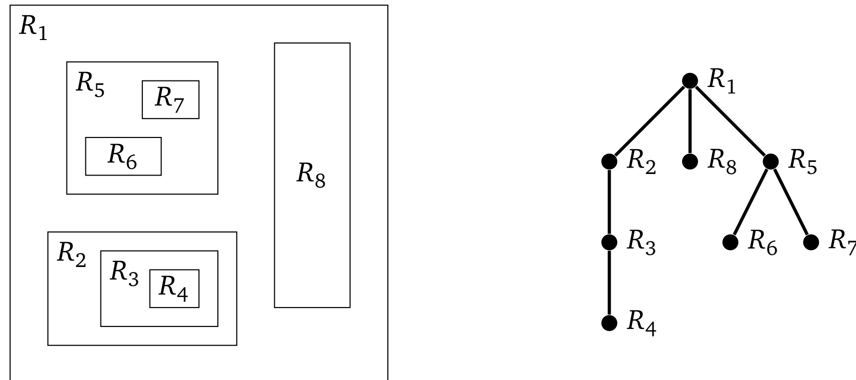
All that's left now is extracting a longest domino sequence, in $O(n+m)$ time.

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 (Application of data structures) 9 marks

Let $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ be a set of rectangles in the plane. We call $\mathcal{R}$ *properly nested* if (i) $R_i \subseteq R_1$ for all $1 < i \le n$, that is R_1 contains all rectangles in $\mathcal{R}$ and (ii) there are no two rectangles in $\mathcal{R}$ whose boundaries intersect, that is, two rectangles are either disjoint or one is completely contained in the other. A properly nested set of rectangles can be represented by a *nesting tree* whose nodes are the rectangles of $\mathcal{R}$ and where R_i is the parent of R_j if and only if $R_j \subset R_i$ and there is no rectangle R_h such that $R_j \subset R_h \subset R_i$.

Example of a properly nested set of rectangles and the corresponding nesting tree:



Develop an $O(n \lg n)$-time algorithm that tests whether a given set $\mathcal{R}$ of rectangles is properly nested. If $\mathcal{R}$ is properly nested, the algorithm should output the nesting tree of $\mathcal{R}$ represented as a set of pairs $\{(i, p_i) \mid 1 < i \le n\}$ such that R_{p_i} is the parent of R_i in the tree for all $1 < i \le n$. If $\mathcal{R}$ is not properly nested, the algorithm should output a pair of indices (i, j) such that the boundaries of R_i and R_j intersect or $i = 1$ and R_1 and R_j are disjoint. Argue briefly that your algorithm runs in $O(n \lg n)$ time and that it produces the correct answer, both when $\mathcal{R}$ is properly nested and when it is not.

First assume that $\mathcal{R}$ is properly nested. Then any point $p \notin R_1$ is not contained in any rectangle; for any point $p \in R_1$, let the smallest enclosing rectangle *be the rectangle R_i that contains p and such that there is no other rectangle $R_j \subset R_i$ that also contains p. The parent R_{p_i} of R_i in the nesting tree of $\mathcal{R}$ is the smallest enclosing rectangle of the bottom-left corner (or in fact any corner) of R_i.*

Now consider the horizontal line ℓ through the bottom boundary of R_i. The intersections of ℓ with rectangle boundaries partition ℓ into segments such that the points in each segment all have the same smallest enclosing rectangle. Thus, all we have to do is find the segment that contains the bottom-left corner of R_i and determine which is the smallest enclosing rectangle for the points on this segment.

Finding the segment is easy: We maintain an (a, b)-tree T over the left and right x-coordinates of all rectangles that intersect ℓ. The bottom-left corner p of R_i is then contained in the segment bounded by the largest x-coordinate preceding p it and the successor of this x-coordinate in T.

To determine the smallest enclosing rectangle, we label every x-coordinate x in T with the smallest enclosing rectangle of the points on the segment of ℓ that has this x-coordinate as its left endpoint. Thus, to find the parent of R_i in the nesting tree, we search T for the largest x-coordinate x no greater than the left x-coordinate of R_i and then report the index of the rectangle stored with x.

The tree T and the labelling of the x-coordinates it stores is easy to maintain as part of a bottom-up plane sweep. Initially, T is empty. We sort the bottom and top y-coordinates of all rectangles in $O(n \lg n)$ time and then process them in order, bottom to top.

Extra space for Question 3.3

When processing the bottom y-coordinate of R_i, we find the largest x-coordinate x in T no greater than R_i's left x-coordinate and report the rectangle R_j stored with x as R_i's parent. We then insert R_i's left and right x-coordinates x_l and x_r immediately after x into T. Since R_j is the smallest enclosing rectangle of R_i, R_j is the smallest enclosing rectangle of the segment with x_r as its left endpoint. R_i is he smallest enclosing rectangle of the segment between x_l and x_r.

When processing the top y-coordinate of R_i, we simply delete x_l and x_r from T because R_i no longer intersects ℓ.

The correctness of the algorithm follows from our discussion so far. Its cost is $O(n \lg n)$ because it sorts the y-coordinates of all rectangles in $O(n \lg n)$ time and then perform a constant number of (a, b)-tree operations, each with cost $O(\lg n)$ for each of the $2n$ y-coordinates.

Now, how can the algorithm fail if $\mathcal{R}$ is not properly nested? Let i be the index such that $R_1, R_2, \ldots, R_{i-1}$ are properly nested and $R_1, R_2, \ldots, R_i$ are not, where the rectangles are numbered in order by their bottom y-coordinates. Then either $R_i \cap R_1 = \emptyset$ or R_i's boundary intersects the boundary of some rectangle R_j, $j < i$.

If $R_1 \cap R_i = \emptyset$, then either T is empty by the time we process R_i's bottom boundary (the y-ranges of R_1 and R_i are disjoint) or R_i's left x-coordinate is less than or greater than all x-coordinates in T (the x-ranges of R_1 and R_i are disjoint). Both conditions are easily checked as part of the query on T using R_i's left boundary, so we can report (R_1, R_i) in this case.

If the boundaries of R_i and R_j intersect, we distinguish two cases: If the x-range of R_j does not contain the x-range of R_i, then the x-range of R_i must contain at least one x-coordinate of R_j. We can test for this condition by performing two searches on T when processing R_i's bottom boundary, one with x_l and one with x_r. If both return the same x-coordinate x, then the x-range of R_i contains no x-coordinate of any rectangle R_j, $j < i$. If they return different x-coordinates, then let x' be the x-coordinate reported by the search with x_r, and let R_j be the rectangle it belongs to. Since the x-range of R_i contains x' and the y-range of R_j contains the bottom boundary of R_i (because $x' \in T$), the boundaries of R_i and R_j intersect, so we report this pair.

If both searches with x_l and x_r return the same x-coordinate, let R_{p_i} be the smallest enclosing rectangle stored with the x-coordinate x these searches return (that is, the rectangle we believe to be R_i's parent in the nesting tree). Since $R_1, R_2, \ldots, R_{i-1}$ are properly nested and R_i's x-range is either disjoint from the x-range of any rectangle R_j, $j < i$, or is completely contained in R_j's x-range, R_i intersects the boundary of some rectangle R_j, $j < i$, if and only if it intersects the boundary of R_{p_i}. We can test this condition in constant time after determining R_{p_i}.

In summary, testing whether $\mathcal{R}$ is properly nested and, if not, reporting a pair of rectangles that proves this increases the cost of the algorithm for properly nested rectangles by only a constant factor, that is, it is still $O(n \lg n)$.

Extra space for Question 3.3