

Banner number:

Name:

Final Exam

CSCI 3110: Design and Analysis of Algorithms

August 5, 2015

Group 1		Group 2		Group 3		Σ
Question 1.1		Question 2.1		Question 3.1	<input type="checkbox"/>	
Question 1.2		Question 2.2		Question 3.2	<input type="checkbox"/>	
Question 1.3		Question 2.3		Question 3.3	<input type="checkbox"/>	
Σ		Σ		Σ		

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. 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 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.

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

(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?

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 $f(n) \in \Omega(g(n))$.

- (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.

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

Question 1.3 (Complexity classes)

9 marks

(a) Formally define the complexity class P

(b) Formally define the complexity class NP

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

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

(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.

Question 2.1 (Recurrence relations)

10 marks

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

$\text{GCD}(a, b)$

```
1  if  $b = 0$ 
2    then return  $a$ 
3    else return  $\text{GCD}(b, a \bmod b)$ 
```

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 $\text{GCD}(a, b)$ satisfies $n < 2b$.)

Question 2.2 (Correctness proofs)

10 marks

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

$\text{BINARYSEARCH}(A, i, j, x)$

```
1  if  $j < i$ 
2    then return no
3   $m \leftarrow \lfloor \frac{i+j}{2} \rfloor$ 
4  if  $A[m] = x$ 
5    then return yes
6  else if  $A[m] > x$ 
7    then return  $\text{BINARYSEARCH}(A, i, m - 1, x)$ 
8  else return  $\text{BINARYSEARCH}(A, m + 1, j, x)$ 
```

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

Question 2.3 (Amortized analysis)

10 marks

Recall the trusty old queue data structure. $\text{ENQUEUE}(Q, x)$ appends element x to the end of the queue. $\text{DEQUEUE}(Q)$ 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 $\text{ENQUEUE}(Q, x)$ operation prepends x to B . A $\text{DEQUEUE}(Q)$ 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:

$\text{REVERSE}(L)$

```
1   $R \leftarrow$  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.

Question 3.1 (Divide and conquer)

9 marks

Consider an array A storing n numbers and consider the problem of finding the k th 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 k th 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.

Extra space for Question 3.1

A large, empty rectangular box with a thin black border, occupying most of the page. It is intended for providing an answer to Question 3.1.

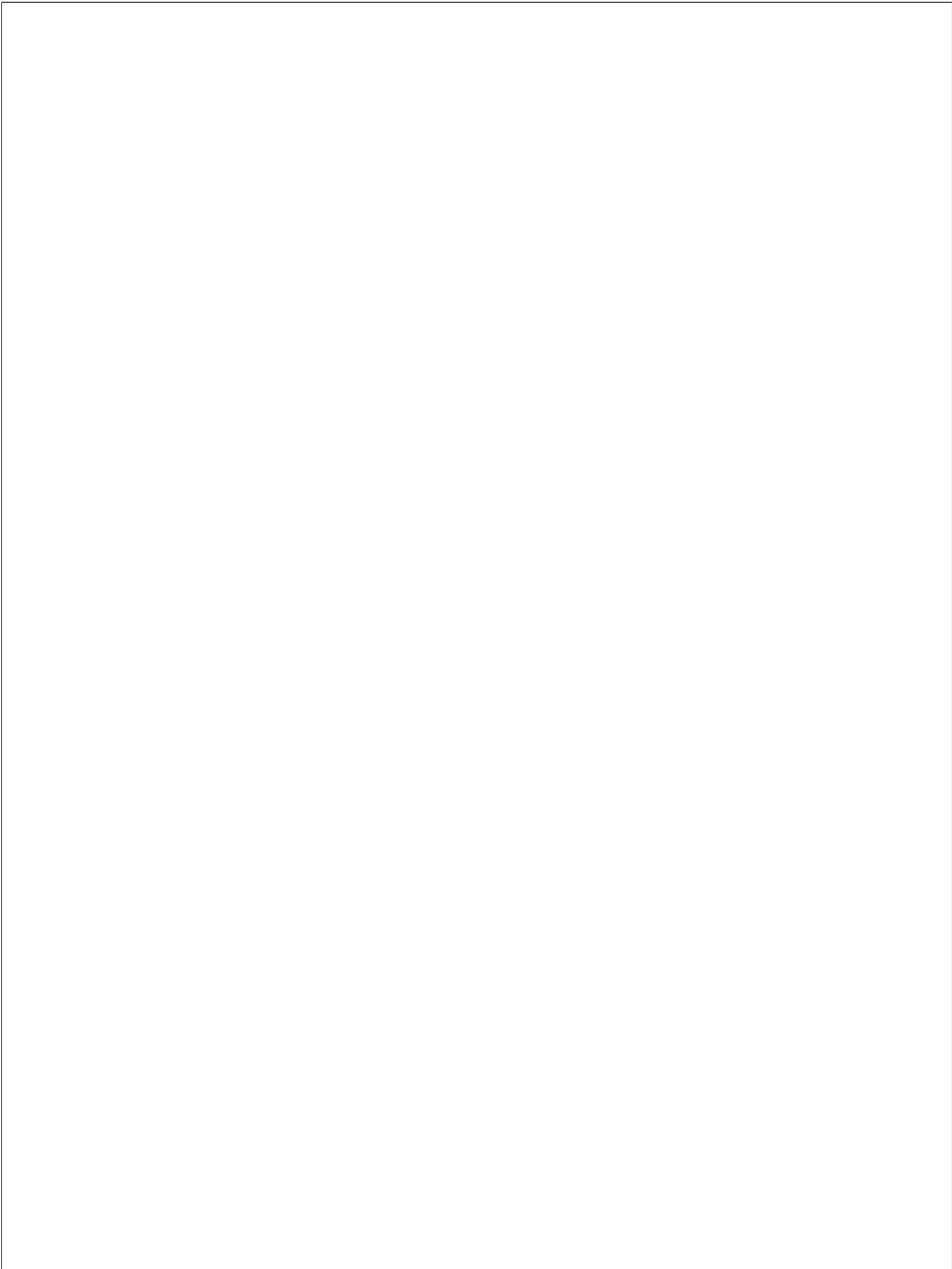
Question 3.2 (Dynamic programming)**9 marks**

Let S be a sequence of m integer pairs $\langle (x_1, y_1), (x_2, y_2), \dots, (x_m, y_m) \rangle$. 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 $\langle (x_{i_1}, y_{i_1}), (x_{i_2}, y_{i_2}), \dots, (x_{i_t}, y_{i_t}) \rangle$ such that $1 \leq i_1 < i_2 < \dots < i_t \leq m$ 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 .

Extra space for Question 3.2

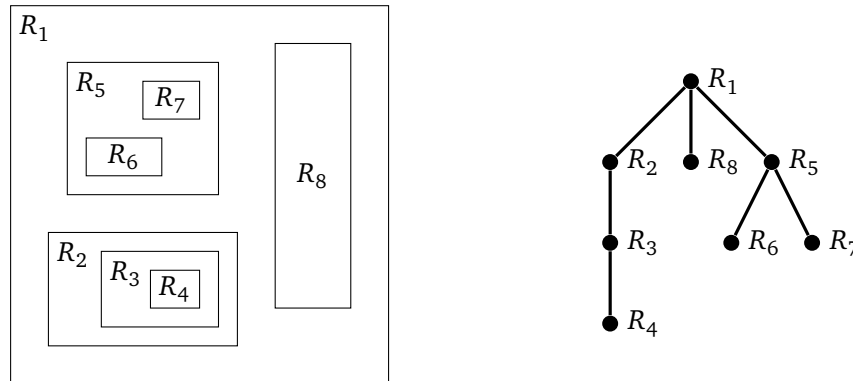


Question 3.3 (Application of data structures)

9 marks

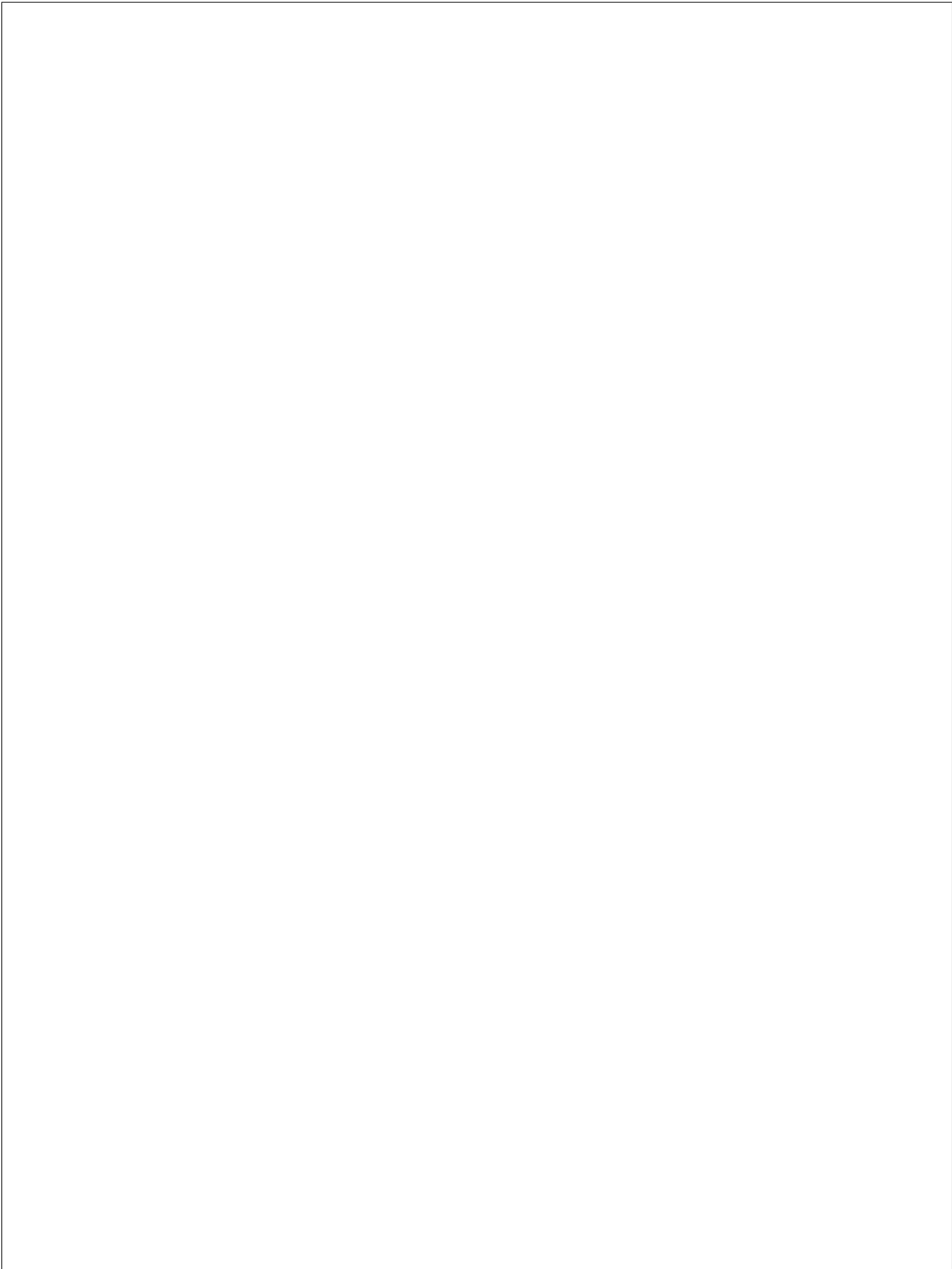
Let $\mathcal{R} = \{R_1, R_2, \dots, R_n\}$ be a set of rectangles in the plane. We call \mathcal{R} *properly nested* if (i) $R_i \subseteq R_1$ for all $1 < i \leq n$, that is R_1 contains all rectangles in \mathcal{R} and (ii) there are no two rectangles in \mathcal{R} whose boundaries intersect, 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 $\{(i, p_i) \mid 1 < i \leq n\}$ such that R_{p_i} is the parent of R_i in the tree for all $1 < i \leq n$. 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.

Extra space for Question 3.3



Extra space for Question 3.3

