

## Problem Set 6

Due: Tuesday, November 27

*Note:* This assignment includes an insanely large number of points (300) in extra credit problems. All the extra credit problems are very interesting, but most require some time to “simmer” in the head. I suggest you read all of them right away, and later work (a little bit at a time) on the ones you find most interesting.

**Reading:** Handouts 21 (Red-Black Trees), 24 (Memoization and Dynamic Programming), 28 (Matrix-Chain Multiplication); *CLRS* Chapter 13, 14.1–14.2, 15; *CLR* Chapter 14, 15.1–15.2, 16.

**Suggested Problems:** *CLRS* 13.1-6, 13.1-7, 13.3-1, 13.3-5, 13.4-1, 15.2-1, 15.4-1; *CLR* 14.1-4, 14.1-5, 14.3-1, 14.3-2, 14.3-5, 14.4-1, 16.1-1, 16.3-1

**Required Problems:** You should write up and turn in the solutions to the problems listed below: The points awarded per problem are given in brackets.

### Problem 1 [15]: Red-Black Trees

In the following parts, you are asked to draw some red-black trees. As a simple check to avoid errors, verify that every tree you draw is a legal red-black tree.

- a. [7] Draw the sequence of red-black trees  $T_0, T_1, \dots, T_{12}$  that results from inserting the following letters one-by-one (from left to right) into an empty tree:

T H E Q U I C K Y A M S

For eliminating red-red violations, use the first (simpler) of the two algorithms presented in Handout #21.

- b. [8] Draw the sequence of red-black trees  $T_{12}, T_{13}, \dots, T_{24}$  that results from deleting the following letters one-by-one (from left to right) from the tree that results from part a.

T H E Q U I C K Y A M S

When deleting a non-fringe node, replace it by its predecessor, not its successor.

### Problem 2 [20]: Augmenting Red-Black Nodes

- a. [10]: **Successor/Predecessor** Describe how to augment *each node* of a red-black tree with extra fields so that calculating the predecessor and successor of any given node can be performed in constant time. (See *CLRS* Sections 14.1–14.2/*CLR* Section 15.1 – 15.2 on augmenting red-black trees.) You show that the new fields can be maintained efficiently during **Insert** and **Delete**. This requires showing two things: (1) maintaining the field after BST insertion or deletion takes time  $O(\lg(n))$  and (2) the fields can be updated in  $O(1)$  for each rotation during the fixup phase.

- b. [10]: **Min/Max** Describe how to augment *each node* of a red-black tree with extra fields so that calculating the minimum and maximum of the subtree rooted at any given node can be performed in constant time.

**Problem 3 [15]: Longest Common Subsequence**

Use the memoized longest common subsequence (LCS) algorithm presented in class (and Handout #24) to determine *all* of the longest common subsequences of the sequences BACCABABC and ACBABCCAB. As part of your solution, you should draw a 2-dimensional table in which some slots are filled with tree nodes labeled  $\oplus$ ,  $\otimes$ , and  $\emptyset$ .

**Problem 4 [25]: Longest Monotonically Increasing Subsequence**

Solve *CLRS* Exercise 15.4-5/*CLR* Exercise 16.3-5 using the two strategies described below. You may assume that the input is a sequence  $a_1, \dots, a_n$  that is represented as either an array or a list. The input sequence may have duplicates, but your output sequence should not have duplicates. For each strategy:

- describe your algorithm (in English is fine)
  - briefly argue why it is correct
  - briefly argue why it takes  $O(n^2)$  time.
- a.** [10] Develop a solution that uses the  $\Theta(mn)$  LCS algorithm as a black box as part of the solution. *Note:* you can express a solution of this form *very* concisely!
- b.** [15] Develop a solution that does not use the LCS algorithm as a black box, but instead uses an auxiliary array  $M[1..n]$  or list  $[m_1, \dots, m_n]$ , where each  $M[i]$  or  $m_i$  stores a list of the longest monotonically increasing sequence in  $a_1, \dots, a_n$  that begins with  $a_i$ .

### Problem 5 [25]: Tree Counting

For the purpose of this problem, a binary tree is either (1) a leaf or (2) a node with left and right subtrees. (In this definition, nodes do not carry values.) The following function counts the number of distinct binary trees with  $n$  nodes:

```
Count-Trees (n) =  
  if n = 0 then  
    return 1 ▷ There is one leaf  
  else  
    count ← 0  
    for i = 0 to n-1 do  
      count ← count + (Count-Trees(i) * Count-Trees((n-1)-i))  
    return count
```

For example,

```
Count-Trees(0) = 1  
Count-Trees(1) = 1  
Count-Trees(2) = 2  
Count-Trees(3) = 5
```

$\text{Count-Trees}(n)$  is also known as the  $n$ th Catalan number. It can be shown that  $\text{Count-Trees}(n)$  takes time exponential in  $n$  (*CLRS* Problem 12-4/*CLR* Problem 13-4).

- a. [8] Write pseudocode for a memoized version of **Count-Trees**.
- b. [7] Write pseudocode for a dynamic programming version of **Count-Trees**.
- c. [5] Use the solution to part (b) to calculate **Count-Trees**(8).
- d. [5] What is the running time of your solutions to parts (a) and (b)? (They should be the same!) Justify your answer.

### Extra Credit 1 [50]: Breadth-First Numbering

For the purposes of this problem, a binary tree is either (1) a `Leaf` or (2) a `Node` of a left subtree, a value, and a right subtree. In Haskell, this datatype would be declared as follows:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

In this problem, your goal is to write a function `breadthFirstNumbering` that maps an input binary tree to an output binary tree that has exactly the same shape, but whose values indicate the position of the nodes (starting at 1) in a breadth-first traversal of a tree.

For example, suppose that the tree `t` is defined as:

```
t = Node (Node (Node Leaf 'A' Leaf)
            'L'
            (Node (Node Leaf 'G' Leaf)
                  'Q'
                  Leaf))
        'R'
        (Node Leaf
            'I'
            (Node Leaf
                'T'
                (Node (Node Leaf 'H' Leaf)
                      'M'
                      (Node Leaf 'S' Leaf))))))
```

Then `breadthFirstNumbering(t)` should yield the following tree:

```
Node (Node (Node Leaf 4 Leaf)
            2
            (Node (Node Leaf 7 Leaf)
                  5
                  Leaf))
    1
    (Node Leaf
        3
        (Node Leaf
            6
            (Node (Node Leaf 9 Leaf)
                  8
                  (Node Leaf 10 Leaf))))))
```

Your implementation of `breadthFirstNumbering` should be *purely functional*. That is, you should not use any side effects, such as assignments or mutable data structures.

## Extra Credit 2 [50]: Pagong Numbers

*This problem is an adaptation of a problem from the 2000 ACM Boston Preliminary Programming Contest (BOSPPE).*

The inhabitants of the ancient tribe of Pagong had incredible numerical dexterity. Their numeral system was not decimal, but reverse-polish-notation based. Each number was represented by a sequence of digits and operations. The digits, they used were surprisingly similar to those we use today:

```
one two three four five six seven eight nine ten
```

represented our numbers 1 through 10, respectively. To build numbers larger than ten, they composed other numbers with the operations `*` or `+`, respectively, using reverse polish notation, where the operation follows the two operands. Thus, one representation for our number 13 is:

```
eight five +
```

Another representation is:

```
one two + two * two * one +
```

Because there were so many representations of the same number, “high” Pagong dictates that the canonical representation of a number be the shortest possible representation, and where there were ties in length, that the earliest lexicographically shall be the canonical number. (Their lexicographical ordering was the same as ASCII, amazingly). The length includes all characters, including a single space that follows each digit or operation, but the last. Thus, the shortest possible representations of 13 are:

```
three ten +
four nine +
six seven +
seven six +
nine four +
ten three +
```

and of these,

```
four nine +
```

is the earliest and thus canonical representation of 13.

Your task is to write, in your favorite programming language, a procedure `Pagong` that takes a non-negative integer and returns a string that is the canonical Pagong representation. Your procedure should be *efficient* in the sense that it should not compute the answer to the same subproblem more than once. In particular, you should be able to calculate `Pagong(999)` in a reasonable amount of time (a few minutes at most).

Here are some examples:

```
Pagong(5) = "five"
Pagong(64) = "eight eight *"
Pagong(101) = "one ten ten * +"
```

Using your **Pagong** procedure, you should be able to answer the following questions:

1. What number between 1 and 1000 has the largest canonical representation?
2. What is the largest number between 1 and 1000 whose canonical representation does not include “ten”?

**Extra Credit 3 [50]** Do *CLRS* Exercise 15.4-6/*CLR* Exercise 16.3-6.

**Extra Credit 4 [50]** Do *CLRS* Problem 15-2/*CLR* Problem 16-2.

**Extra Credit 5 [50]** Do *CLRS* Problem 15-3 part(a)/*CLR* Problem 16-3. For even more credit, do *CLRS* Problem 15-3 part(b) (only in *CLRS*), which relates part (a) to DNA sequences.

**Extra Credit 6 [25]** Do *CLRS* Problem 15-4/*CLR* Problem 16-4

**Extra Credit 7 [25]** Do *CLRS* Problem 15-6. (There is not a version of this problem in *CLR*.)

*Problem Set Header Page*  
*Please make this the first page of your hardcopy submission.*

## **CS231 Problem Set 6**

### **Due Tuesday, November 27**

Name:

Date & Time Submitted:

Collaborators (*anyone you worked with on the problem set*):

*In the **Time** column, please estimate the time you spend on the parts of this problem set. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the **Score** column when grading your problem set.*

<b>Part</b>	<b>Time</b>	<b>Score</b>
General Reading		
Problem 1 [15]		
Problem 2 [20]		
Problem 3 [15]		
Problem 4 [25]		
Problem 5 [25]		
Extra Credit [300]		
<b>Total</b>		