Problem Set 3

Reading: Maurer & Ralston: Chapter 1. Cormen, Leiserson & Rivest: Chapters 1 and 11 (Handout 11).

The problems are divided into oral problems (prefixed "**O**") and written problems (prefixed "**W**"). Monday the 21st is a holiday, so you will not be asked to present them in tutorial. The written problems are due at the beginning of class on Tuesday, February 22.

Problem O3-1. Imagine that you have an array $A[1\,..\,n]$, and you would like to implement two stacks using two pointers into the array. Moreover, you would like neither stack to run out of space until the number of the elements in both stacks combined is n. Describe a simple and efficient implemention of the two stacks.

Problem O3-2. Give code or pseudocode of an algorithm to merge two sorted lists. The algorithm should take two sorted lists of numbers as input and produce a single sorted list as output. Write another merging algorithm in which the input and output lists are terminated with an ∞ sentinel. Compare the coding complexity of the two algorithms. What are the asymptotic (Θ) running times of your algorithms in terms of the total number of elements in the lists.

Problem O3-3. Consider a multiple-array representation of a doubly linked list as described in Chapter 11 of CLR (Handout 11). It is sometimes desirable to maintain such a list in a compact fashion, using only the first m locations in the multiple-array representation if there are m items in the linked list. In other words, the unused objects always occupy the highest locations. Explain how to allocate and free objects in $\Theta(1)$ (constant) time per operation, preserving the compact representation from operation to operation. Recall that the elements of a linked list need not be kept in any particular order. Assume that there are no pointers to elements of the linked list outside the list itself. Can this method be made to work if the list is singly linked?

Problem O3-4. Look at the **while** loop in lines 5–7 of the INSERTION-SORT procedure on page 3 of the CLR handout. The loop essentially executes a linear search backward through the sorted array to find a place for the key. Recall that unlike a linked list, an array has its items in specific, consecutive locations in memory. Can we replace this search with a binary search (see M&R, page 129) to improve the worst-case running time of insertion sort to $\Theta(n \lg n)$ for sorting n numbers? How does the answer change if the items to be sorted were stored in a linked list rather than an array?

Problem O3-5. Euclid's Algorithm for GCD. The greatest common divisor of two natural numbers a and b, denoted $\gcd(a, b)$, is the largest natural number that (evenly) divides both. We write $d \mid a$ if $a = kd$ for some natural number k. Euclid's algorithm for computing $\gcd(a, b)$ is

```
EUCLID(a, b)
if b = 0
    then return a
    else return EUCLID(b, a mod b)
```

where $a \bmod b$) is the remainder when a is divided by b. As an example, the computation of $\text{EUCLID}(30, 21)$ makes three recursive calls:

$$\begin{aligned} \text{EUCLID}(30, 21) &= \text{EUCLID}(21, 9) \\ &= \text{EUCLID}(9, 3) \\ &= \text{EUCLID}(3, 0) \\ &= 3. \end{aligned}$$

Prove this algorithm is correct by first showing

$$\gcd(a, b) \mid \gcd(b, a \bmod b)$$

and

$$\gcd(b, a \bmod b) \mid \gcd(a, b)$$

to conclude $\gcd(a, b) = \gcd(b, a \bmod b)$, and then arguing that the algorithm must terminate.

Problem O3-6. The following algorithm partitions an array $A[1 \,..\, n]$ into two regions, $A[1 \,..\, i]$ and $A[i + 1 \,..\, n]$, such that every element in the first region is less than or equal to an input value x, and every element in the second region is greater than x.

```
PARTITION(x, A)
i ← 0
for j ← 1 to n
     do if A[j] ≤ x
           then i ← i + 1
                exchange A[i] ↔ A[j]
return i
```

(a) Argue that PARTITION is correct.

(b) What is the maximum number of times that any single element can be moved by PARTITION?

(c) Argue that PARTITION runs in $\Theta(n)$ time on an n-element array.

Problem O3-7. Radix sort. Suppose that you have the complete set of 792 Topps baseball cards, and you would like to put them in numerical order. Each card has a photo on the front and a unique number on the back. You could sort them by the hundred's digit first, and then sort each of the resulting ten sets by the ten's digit, and so on, but that would create too many intermediate piles and too much confusion.

Instead, you use the following algorithm. First, you partition the cards into ten piles according to the *one's* digit. Then, you stack the piles face-up, with the 0's pile on top and the 9's pile on bottom. You repeat the procedure, sorting this time by the ten's digit, going through the cards from top to bottom and placing each card *face-down* in its proper pile. Placing the cards face-down maintains the relative order of cards that have the same digit in this round of sorting. (Sorting algorithms with this property are called **stable**.) Finally, you repeat for the hundred's digit.

Generalize this algorithm to an arbitrary number of digits. Use induction on the number of digits to prove that the algorithm properly sorts the cards.

Problem W3-8. Read Section 3.8 of M&R, which gives three systematic methods to process all nodes in a binary tree: preorder, inorder, and postorder tree walks (or tree traversals).

(a) Ignoring internal nodes of a binary tree, how do the orders that leaf nodes are processed differ among the three methods?

(b) Using a stack, give a *non*recursive algorithm to perform an inorder walk of the nodes in a binary tree. Asymptotically, how large a stack do you require as a function of some parameter of the binary tree?

(c) One node u is an **ancestor** of another node v in a binary tree if the unique simple path from the root of the binary tree to v contains u. (Thus, by this definition, u can be its own ancestor.) Prove that if u and v are nodes in a binary tree, then u is an ancestor of v if an only if u precedes v in preorder and u follows v in postorder.

Problem W3-9. A string of parentheses and brackets is said to be *well formed* if the parentheses and brackets "match up" correctly, as for example, they do in the string [(( ) [ ])] but not in the strings [(( )]) and [[( )]. Formally, we can give an inductive definition for well-formed strings. A string w of parentheses and brackets is well formed if

1. $w = (\,)$ or $w = [\,]$, or
2. $w = (\,v\,)$ or $w = [\,v\,]$, where v is well formed, or
3. $w = u\,v$, where u and v are well formed.

(a) Give a simple algorithm using a stack for testing whether a string of parentheses and brackets is well formed.

(b) Sketch a correctness proof for your algorithm.

(c) What is the asymptotic (Θ) worst-case running time of your algorithm on a string of length n?

Problem W3-10. You have a collection of n boxes that have the same dimension but distinct weights. Suppose you wish to find the heaviest of n boxes using only a balance which can tell you which of two boxes is heavier. A **comparison** consists of using the balance to compare the weights of two boxes.

(a) Give an algorithm for determining the heaviest box using $n - 1$ comparisons.

(b) Prove that $n - 1$ comparisons are necessary to determine the heaviest box in the worst case. Argue by looking at a directed graph on n nodes—one node for each box—with an edge from node u to node v if a direct comparison showed that box u was heavier than box v.

(c) Give an algorithm for determining both the heaviest *and the second-heaviest* boxes using only $n + \lg n - 2$ comparisons.

(d) Give an algorithm for determining both the heaviest and the lightest boxes using only $\lceil 3n/2 \rceil - 2$ comparisons.