

## Homework 3

ECE 1762 Algorithms and Data Structures  
Fall Semester, 2007

**Due: November 6, 2007. To Prof. Veneris' assistant, SF-2001, until 3pm!!**

**Warning: This is LONG homework !!!**

**For each algorithm you design you should give a detailed description of the idea, proof of correctness, termination, analysis and proof of time and space complexity. If not, your answer will be incomplete and you will miss credit!**

1. **[Dynamic Programming and Greedy Algorithms, 10+5 points]** A *traveling salesman tour* (TST) in an undirected complete graph  $G$ , with positive edge weights  $w$ , is a closed path that visits every vertex in the graph exactly once. The weight of TST is the sum of its edge weights. The traveling salesman problem is to find a TST of smallest weight. Let  $n$  be the number of vertices.
  - (a) Write a dynamic programming formulation for this problem and analyze its running time.
  - (b) Consider the following greedy approach. Start with an arbitrary vertex  $v_1$ , as a closed path from  $v_1$  to  $v_1$  that we call  $P_1$ . In the  $i$ -th step, let  $v_i$  be the vertex not in  $P_{i-1}$  which is closest to any of the vertices in  $P_{i-1}$  (it is the  $v$  not in  $P_{i-1}$  that minimizes  $w(v, u)$  over all  $u$  in  $P_{i-1}$ ). Say  $v_i$  is closest to  $v_j$  in  $P_{i-1}$ . Then insert  $v_i$  between  $v_j$  and a neighbor in  $P_{i-1}$  to obtain  $P_i$ . Stop with  $P_n$ . How bad can the obtained TST in comparison with the optimal one? That is, how large can the ratio of the weight of the TST obtained to the weight of the optimal TST? (here you are expected to construct examples)
  
2. **[Greedy Algorithms, 20 points]** Consider the problem of making change for  $n$  cents using the fewest number of coins. Assume that each coin's value is an integer.
  - (a) Describe a greedy algorithm to make change consisting of quarters, dimes, nickels and pennies. Prove that your algorithm yields an optimal solution.
  - (b) Suppose that the available coins are in the denominations that are powers of  $c$ , *i.e.*, the denominations are  $c^0, c^1, c^2, \dots, c^k$  for some integers  $c > 1$  and  $k \geq 1$ . Show that the greedy algorithm always yields an optimal solution.
  - (c) Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every  $n$ .
  - (d) Give an  $O(nk)$ -time algorithm that makes change for any set of  $k$  different coin denominations, assuming that one of the coins is a penny.
  
3. **[Dynamic Programming, 15 points]** A sequence of integers  $x_1, \dots, x_n$  is *strictly  $k$ -modal* if for some  $i_1, \dots, i_{k-1}$  with  $1 < i_1 < i_2 < \dots < i_{k-1} < n$ , each of the subsequences

$$x_{i_j}, x_{i_j+1}, \dots, x_{i_{j+1}-1}, x_{i_{j+1}}$$

with  $i_0 = 1$  and  $i_k = n$ , is strictly monotone, alternating between strictly increasing and strictly decreasing (the first one can be either increasing or decreasing). For example, the sequence 1, 3, 6, 3, 2, 5, 7, 8, 9 can be split into 1, 3, 6, 6, 3, 2, 2, 5, 7, 8, 9, and so it is 3-modal.

Now, given a sequence of numbers, we are interested in subsequences that are  $k$ -modal. For example, consider the sequence 1, 6, 3, 8, 3, 9, 2, 4, 2, 1. Its subsequence 1, 3, 8, 9, 4, 2, 1 is strictly 2-modal (note that the order in the original sequence is preserved in the subsequence).

Describe an algorithm that given an input array  $A[1 \dots n]$  and a parameter  $k$ , returns a subsequence of  $A[1], A[2], \dots, A[n]$  that is longest among all its strictly  $k$ -modal subsequences. If there are no  $k$ -modal subsequences, then the output would be the empty subsequence. You may want to try the 1-modal case first. Try to make your algorithm as efficient as possible. Running time  $O(kn \log n)$  is possible. You may first get a slower algorithm and then improve it (if you can't, then at least you'll get some partial credit).

4. [**Hashing, 15 points**] Problem 11.3-3, Page 236 (Problem 12.3-3, Page 232).
5. [**Skewed Heaps, 20 points**] In this problem you are required to develop a data structure similar to that of the leftist heap from Homework 2. In *leftist heaps*, the **Merge** operation preserved the heap ordering and the balance (leftist bias) of the underlying tree. *Skewed heaps* use the same idea for merging heap-ordered trees. **SkewHeapMerge** is performed by merging the rightmost paths of two trees without keeping any explicit balance conditions. This means that there's no need to store the rank of the nodes in the tree. This is similar to self-adjusting trees, since no information is kept or updated.

Good performance of those data structures is guaranteed by a “rebalancing step”—like the splay in self-adjusting trees, only simpler. At each step of the merging along the rightmost paths of the two heaps, we swap *all* of the left and right children of nodes along this path, except for the last one. The modified procedure for merging two skewed heaps looks as follows:

```

function SkewedHeapMerge( $h, h'$ ) : heap
    if  $h$  is empty then return  $h'$ 
    else if  $h'$  is empty then return  $h$ 

    if the root of  $h' \preceq$  the root of  $h$  then
        exchange  $h$  and  $h'$  (*  $h$  holds smallest root *)

    if right( $h$ ) = nil then
        right( $h$ ) :=  $h'$  (* last node, we don't swap *)
    else
        right( $h$ ) := SkewedHeapMerge(right( $h$ ),  $h'$ )
        swap left and right children of  $h$ 

    return  $h$ 

```

The above recursive routine can also be done iteratively. In fact, it can be done more efficiently than the leftist heap **Merge** (by a constant factor), because everything can be done in one pass, while moving down the rightmost path. In the case of leftist heaps, we go down the rightmost

path, and then back up to recompute ranks. In leftist heaps, that also requires either a recursive algorithm or pointers from nodes to their parents<sup>1</sup>.

Since there is no balance condition, there's no guarantee that these trees will have  $O(\log n)$  worst-case performance for the merge (and hence all of the other operations). But they do have good *amortized performance*.

Here's the intuition for the above: In the previous merge algorithm we only had to swap children when the right one had a larger rank than the left. In this merge algorithm, we always swap children, so we might actually replace a right child of "small" rank with one of "large" rank. But the *next time* we come down this path, we will correct that error, because the right child will be swapped onto the left.

- (a) Show that a **SkewedHeapMerge** of two skewed heaps uses amortized time  $O(\log_2 n)$  by the use of the *accounting method*.
- (b) Show that **Insert** and **DeleteMin** have the same amortized time bounds.

*Hint:* Use weights instead of ranks. Define the *weight* of a node to be the number of nodes in the subtree rooted at that node (below that node, including the node itself). Let node  $x$  be a *heavy* node if its weight is more than half of the weight of its parent. Otherwise it is *light* one. What can we say about light and heavy nodes in any binary tree? Look at any root–leaf path in the tree. *How many* light nodes can you encounter in such path? Why? The best case is when the light nodes are also right children of their parents. For the accounting method, every time we swap, we must pay \$1. Moreover, if it happens to swap a heavy child to a right child position, then you must deposit a 'penalty' amount.

---

<sup>1</sup>Just a note on implementation here. Sometimes we may want to be able to move up a tree as easily as we move down; so every node will also include a pointer to its parent. That means that a node has a pointer to its left child, a pointer to its right child, and a pointer to its parent, increasing the amount of space needed to store trees. To decrease the *space* requirement, you can do this. Look at three nodes, a parent  $p$  and its two children  $r$  and  $l$ . Now  $p$  will have a pointer to  $l$ , but not to  $r$ ;  $l$  has a pointer to  $r$  and  $r$  has a pointer back to  $p$ . So, all the left children in a tree have pointers to their right siblings (brothers) and to their own left children. All the right children have pointers to their parents and to their right children. If you draw a picture, this should make more sense. Now every node still has two pointers, and you can visit left nodes, right nodes and parent nodes easily.