Introduction to Algorithms and Data Structures E2004

IT University of Copenhagen

January 17, 2005

This exam consists of 4 problems containing 3 subproblems each. All problems and all subproblems have equal weight. You have 4 hours to complete the exam. Remember to number the pages and write your name and CPR number on every page. The exam consists of 7 pages including this one.

Exercises asking for time complexity must be answered using O-notation. To achieve best score, growth rates in the O-notation should be as tight as possible.

Problem 1: Comparison Counting Sort

The following algorithm sorts an array by counting, for each of its elements, the number of smaller elements. It uses this information to put each element in the appropriate position in the output.

```
COMPARISON-COUNTING-SORT(A)
   n ← length[A]
   new array Count[1 . . n]
   for i ← 1 to n
        do Count[i] ← 0
   for i ← 1 to n − 1
        do for j ← i + 1 to n
                do if A[i] ≤ A[j]
                     then Count[j] ← Count[j] + 1
                     else  Count[i] ← Count[i] + 1
   new array S[1 . . n]
   for i ← 1 to n
        do S[Count[i] +1] ← A[i]
   return S
```

(a) Demonstrate how this algorithm sorts the array

$$A = \langle 6, 3, 8, 9, 1, 4 \rangle .$$

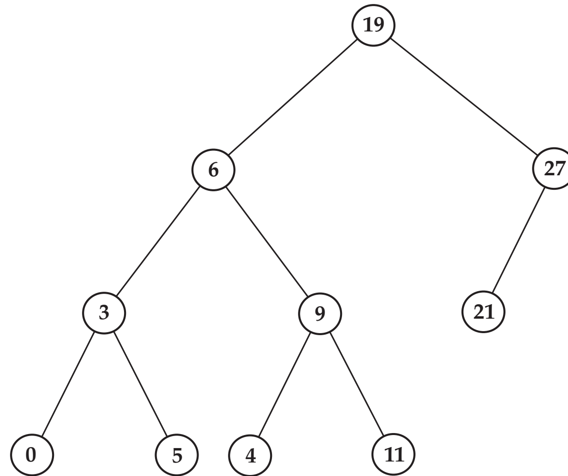
(It is sufficient to show the intermediate values of arrays *Count* and *S* after each iteration of the second and third for loop over *i*.)

(b) Is this algorithm stable? Why?

(c) What is the asymptotic running time of this algorithm? Give a tight asymptotic bound, and argue for your answer.

Problem 2: Around AVL Trees

Consider the following binary tree with key values assigned to nodes:



(a) Explain why this tree fails to be a binary search tree. Change the key value of one of the nodes so that the binary search tree property is restored. Next to each node of the tree, indicate the balance of the node (recall that the *balance* is the signed difference between the height of the left subtree and the height of the right subtree). Is the tree an AVL tree?

(b) Illustrate the AVL deletion algorithm by deleting the node with key value 27 and performing any necessary rebalancing rotation(s). Draw the resulting AVL tree.

Let $k_1, \ldots, k_n$ be the key values of the n nodes in an AVL tree T listed in the sorted order. This list can be obtained by, for example, in-order traversal. Apparently, there is no particularly fast way to find k_i given T and the index i.

(c) Modify the AVL tree data structure by storing $O(1)$ extra information e_v together with each node v so that the operation $\textsc{Find}(T, i)$ of finding the ith key value k_i can be performed in $O(\log n)$ time, and the AVL insertion and deletion procedures can be modified to maintain the values e_v and still take $O(\log n)$ time. Briefly describe the modifications and the algorithm for finding k_i.

Problem 3: Product Assembly Optimization

In this problem we consider n distinct *products* $p_1, \ldots, p_n$. Each product p_i is assigned a *type* of t_i and a single *production rule*. All products p_i are unique, whereas there is no such requirement for the types t_i. In the example of Table 1 there are six products xsonix speaker, PC, geeky speaker, multimedia PC, screen 15”, and screen 17” of three types speaker, PC, and CRT.

For each product p_i its production rule comprises the list of types of required components T_i, and the cost w_i of composing the components in order to obtain p_i. Product p_i can be constructed by first constructing all its required components and then composing them together. The total production cost is the cost of producing all the components plus the cost of the composition.

i	p_i	t_i	T_i	w_i
1	xsonix speaker	speaker	$\langle\rangle$	5,00 kr
2	PC	PC	⟨speaker, CRT⟩	3,00 kr
3	geeky speaker	speaker	$\langle\rangle$	3,00 kr
4	multimedia PC	PC	⟨speaker, speaker, CRT⟩	4,00 kr
5	screen 15”	CRT	$\langle\rangle$	11,00 kr
6	screen 17”	CRT	$\langle\rangle$	22,00 kr

Table 1: A list of production rules for two kinds of PCs.

The first product in Table 1, the xsonix speaker is a basic product, which has no subcomponents, and it can be obtained for 5 kr. At the same time a multimedia PC requires two components of type speaker and a single one of type CRT. The cost of putting the components together is 4 kr. One of the possible ways to build a multimedia PC is to apply the rules: $\langle 1, 1, 5, 4\rangle$, i.e. by combining two xsonix speakers and one screen 15”. The total production cost is:

$$W = w_1 + w_1 + w_5 + w_4 = 5 + 5 + 11 + 4 = 25 \text{ kr} \ .$$

Despite the fact that there is only one production rule for each product, the list of production rules is ambiguous—for each product it may contain more than one way to build it, yielding various costs. This is caused by the fact that more than one product can satisfy the same type requirement.

Let $E \subseteq \{1, \ldots, n\} \times \{1, \ldots, n\}$ be a binary relation on products, such that $(i, j) \in E$ if product p_i can be a direct component of product p_j (so t_i is an element of T_j).

(a) Draw a directed graph of such relation for the example of Table 1. Then give a new way to build a multimedia PC, which has a different cost than above. Compute this cost.

(b) It is known that the list of production rules does not contain cycles (i.e. it cannot be that a product of type PC is a component of itself, either directly or indirectly). Indicate an algorithm which in $O(n+m)$ time can reorder the rules in such a way that one only needs to apply rules with indices smaller than i, when constructing product p_i (where m denotes the number of dependencies in the graph: $m = |E|$).

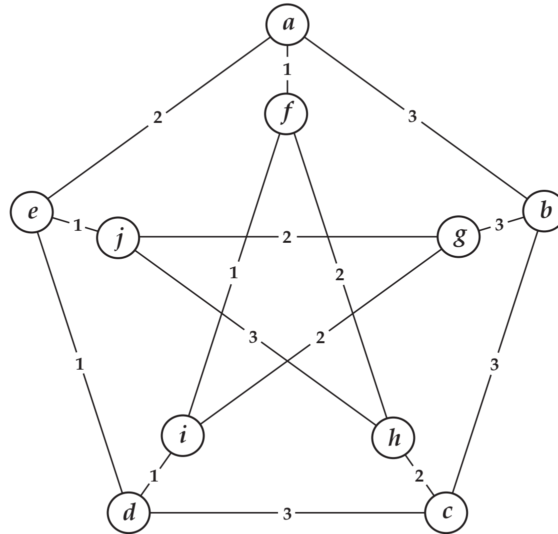
Let arrays t, T and w describe product types, component types and composition costs respectively (see Table 1 for example). The following algorithm returns the cost of the cheapest way to build product p_i. It assumes that the rules are ordered in such a way that one only needs to apply rules with indices smaller than i, when constructing product p_i.

```
RECURSIVE-ASSEMBLY(t, T, w, i)
    new array C[1 .. length[T_i]]
    for k ← 1 to length[T_i]
        do C[k] ← ∞
    for k ← 1 to length[T_i]
        do for j ← 1 to i − 1
               do if t_j = T_i[k]
                      then C[k] ← min(C[k], RECURSIVE-ASSEMBLY(t, T, w, j))
    c ← w_i
    for k ← 1 to length[T_i]
        do c ← c + C[k]
    return c
```

(c) Improve the efficiency of the above algorithm by applying the memoization technique or the dynamic programming technique. Explain which of the two techniques is likely to be faster if the list of production rules describes many products and we are only interested in computing the cost of the cheapest construction scenario for a single product.

Problem 4: Graphs, Trees, Paths

Here is an undirected graph P with weighted edges:



(a) Construct a minimum spanning tree for the graph P.

In an undirected weighted graph one clearly can, just as with directed weighted graphs, speak of the length of paths: Just view each undirected edge as a pair of directed edges in opposite directions with the same weight. Hence the notion of shortest path between two vertices also makes sense.

(b) Construct a shortest path tree for P with a as the source vertex. Do edges in your shortest path tree form a minimum spanning tree? Why?

Let us now turn our attention to undirected trees without edge weights. In any tree T, there is exactly one simple path between any two vertices. (Recall that a path is *simple* if it never passes any vertex more than once.) Since trees are particular cases of graphs, one could use data structures for general graphs to represent trees. With the two standard data structures for graphs, finding the simple path between two vertices will typically involve some kind of search and will likely take at least $O(n)$ worst-case time, where n is the number of vertices in T. To handle queries of the form "Find the simple path from u to v" quickly, one could of course store all these paths in an $n \times n$ table. However this generally takes more than $O(n^2)$ space which may be prohibitively much under some circumstances.

(c) Design a data structure for storing unweighted trees so that the operation SP(u, v) that prints out the simple path from vertex u to vertex v takes $O(f)$ time, where f is the length (i.e. the number of edges in) the simple path from u to v. For full marks, your data structure should fit into $O(n)$ space, but anything taking $O(n^2)$ space is still worth writing down.