

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.

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)

```

 $n \leftarrow \text{length}[A]$ 
new array  $\text{Count}[1..n]$ 
for  $i \leftarrow 1$  to  $n$ 
    do  $\text{Count}[i] \leftarrow 0$ 
for  $i \leftarrow 1$  to  $n - 1$ 
    do for  $j \leftarrow i + 1$  to  $n$ 
        do if  $A[i] \leq A[j]$ 
            then  $\text{Count}[j] \leftarrow \text{Count}[j] + 1$ 
            else  $\text{Count}[i] \leftarrow \text{Count}[i] + 1$ 
new array  $S[1..n]$ 
for  $i \leftarrow 1$  to  $n$ 
    do  $S[\text{Count}[i] + 1] \leftarrow A[i]$ 
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 .)

Solution (a).

After the first loop: after the last iteration $\text{Count} = \langle 0, 0, 0, 0, 0, 0 \rangle$

The 2nd loop: {

- after iteration for $i = 1$ $\text{Count} = \langle 3, 0, 1, 1, 0, 0 \rangle$
- after iteration for $i = 2$ $\text{Count} = \langle 3, 1, 2, 2, 0, 1 \rangle$
- after iteration for $i = 3$ $\text{Count} = \langle 3, 1, 4, 3, 0, 1 \rangle$
- after iteration for $i = 4$ $\text{Count} = \langle 3, 1, 4, 5, 0, 1 \rangle$
- after iteration for $i = 5$ $\text{Count} = \langle 3, 1, 4, 5, 0, 2 \rangle$

The 3rd loop: {

- after iteration for $i = 1$ $S = \langle 0, 0, 0, 6, 0, 0 \rangle$
- after iteration for $i = 2$ $S = \langle 0, 3, 0, 6, 0, 0 \rangle$
- after iteration for $i = 3$ $S = \langle 0, 3, 0, 6, 8, 0 \rangle$
- after iteration for $i = 4$ $S = \langle 0, 3, 0, 6, 8, 9 \rangle$
- after iteration for $i = 5$ $S = \langle 1, 3, 0, 6, 8, 9 \rangle$
- after iteration for $i = 6$ $S = \langle 1, 3, 4, 6, 8, 9 \rangle$

(b) Is this algorithm stable? Why?

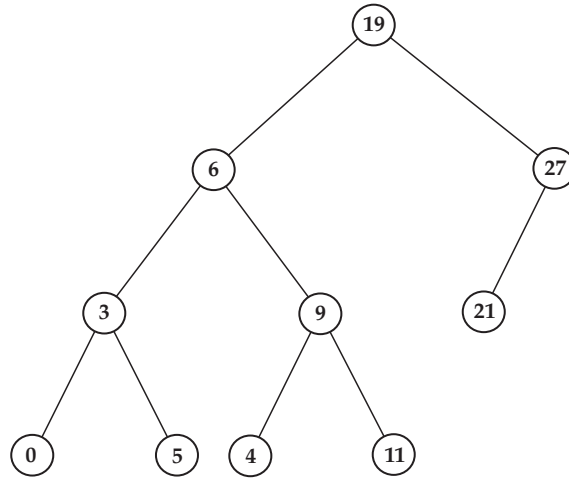
Solution (b). This algorithm is stable. Rationale: If $A[i] = A[j]$ and $i < j$ then always $Count[i] \leq Count[j]$, because $Count[j]$ is increased in such case, not $Count[i]$. This has the consequence that $A[i]$ will not be placed after $A[j]$ in the output array S . $A[i]$ will actually be placed before $A[j]$, which follows from the correctness of the algorithm - it would not be a sorting algorithm, if it placed two elements in the same cell.

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

Solution (c). The first and last loops have n iterations. The running time is dominated by the two nested inner loops, which have a total cost $\Theta(n^2)$ (can be calculated using the sum of arithmetic series). The bound is tight, because the number of iterations only depends on the size of the input array.

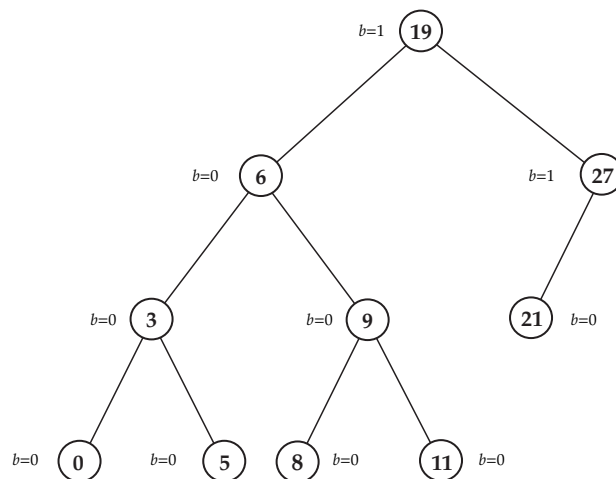
Problem 2: Around AVL Trees

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?

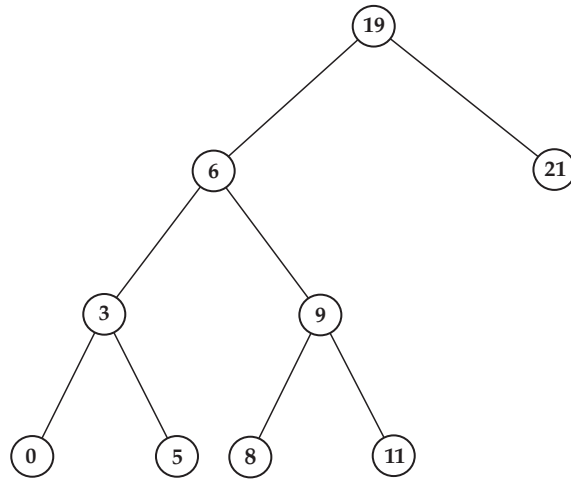
Solution (a). The subtree determined by the node with key value 6 has a node with key value $4 < 6$ in its *right* subtree which breaks the binary search tree property. Changing the key value 4 to, say, 8 restores the search tree property. The resulting tree together with the balances looks like this:



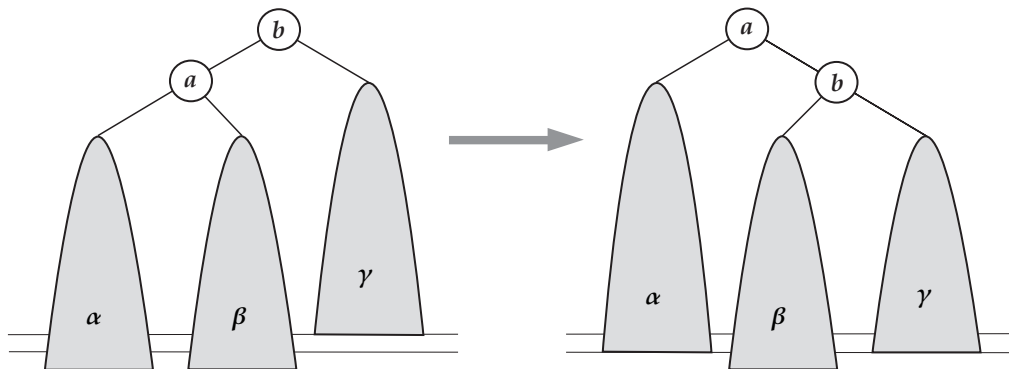
(*b* stands for balance.) The tree is AVL since it is a binary search tree and all balances are among $\{-1, 0, 1\}$.

(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.

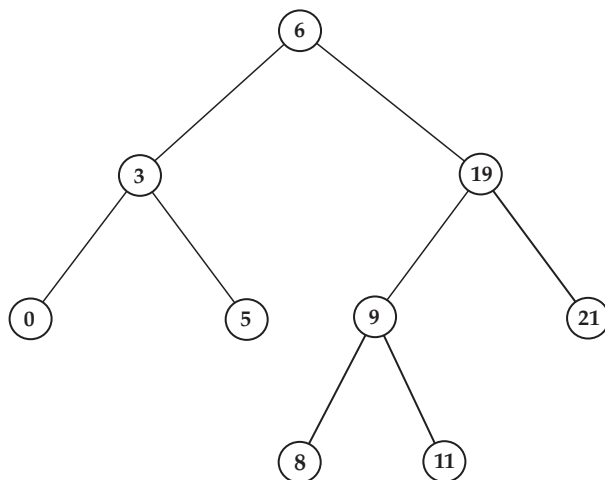
Solution (b). Deletion of the node with key value 27 yields



This is not an AVL tree as the balance of the root is 2. To obtain an AVL tree we perform a rotation of the form



and get the tree



which is properly balanced.

Let k_1, \dots, 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 $\text{FIND}(T, i)$ of finding the i th 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 .

Solution (c). Together with each node v , we store the number n_v of nodes in T_v , the subtree determined by v . To find the i th key value, we start at the root and compare i with n_l where l is the root's left child ($n_l = 0$ if no left child is present). If $i \leq n_l$, we know that k_i is in T_l , and we can apply $\text{FIND}(T_l, i)$ recursively. If $i = n_l + 1$ then k_i is the key value at the root node. If $i > n_l + 1$ then k_i can be found by calling $\text{FIND}(T_r, i - (n_l + 1))$ where r is the root's right child.

While inserting and deleting elements, it is easy to keep track of n_v 's by adding or subtracting 1 from n_v 's for all v on the path from the root to the inserted or deleted node respectively. Traversing this path only takes $O(\log n)$ time. Each rotation only affects n_v for no more than three topmost nodes v of the subtree on which the rotation is performed. For these three rotated nodes, it only takes constant time to re-calculate the new values of n_v by taking the sum of n_c for v 's children c . Thus the extra maintenance does not take more than $O(\log n)$ time per insertion/deletion.

Problem 3: Product Assembly Optimization

In this problem we consider n distinct products p_1, \dots, 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	$\langle \text{speaker, CRT} \rangle$	3,00 kr
3	geeky speaker	speaker	$\langle \rangle$	3,00 kr
4	multimedia PC	PC	$\langle \text{speaker, speaker, CRT} \rangle$	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:

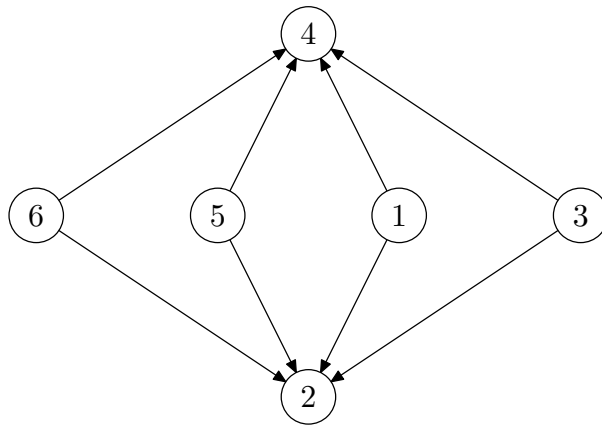
$$W = w_1 + w_1 + w_5 + w_4 = 5 + 5 + 11 + 4 = 25 \text{ kr} .$$

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 $E \subseteq \{1, \dots, n\} \times \{1, \dots, n\}$ 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.

Solution (a).



Another multimedia PC can be built by applying rules $\langle 1, 3, 6, 4 \rangle$.
 Total cost: 34 kr.

(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: $m = |E|$).

Solution (b). One may use TOPOLOGICAL-SORT.

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  $k \leftarrow 1$  to  $length[T_i]$ 
    do  $C[k] \leftarrow \infty$ 
  for  $k \leftarrow 1$  to  $length[T_i]$ 
    do for  $j \leftarrow 1$  to  $i - 1$ 
      do if  $t_j = T_i[k]$ 
        then  $C[k] \leftarrow \min(C[k], \text{RECURSIVE-ASSEMBLY}(t, T, w, j))$ 
   $c \leftarrow w_i$ 
  for  $k \leftarrow 1$  to  $length[T_i]$ 
    do  $c \leftarrow c + C[k]$ 
  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.

Solution (c).

MEMOIZED-ASSEMBLY(t, T, w, i)

```

new  $M[1..n]$ 
for  $k \leftarrow 1$  to  $n$ 
    do  $M[k] \leftarrow \infty$ 
return ASSEMBLY-HELPER( $t, T, w, M, i$ )

```

ASSEMBLY-HELPER(t, T, w, M, i)

```

if  $M[i] < \infty$ 
    then return  $M[i]$ 
new  $C[1..length[T_i]]$ 
for  $k \leftarrow 1$  to  $length[T_i]$ 
    do  $C[k] \leftarrow \infty$ 
for  $k \leftarrow 1$  to  $length[T_i]$ 
    do for  $j \leftarrow 1$  to  $i - 1$ 
        do if  $t_j = T_i[k]$ 
            then  $C[k] \leftarrow \min(C[k], ASSEMBLY-HELPER(t, T, w, M, j))$ 
 $M[i] \leftarrow w_i$ 
for  $k \leftarrow 1$  to  $length[T_i]$ 
    do  $M[i] \leftarrow M[i] + C[k]$ 
return  $M[i]$ 

```

DYNAMIC-ASSEMBLY(t, T, w, i)

```

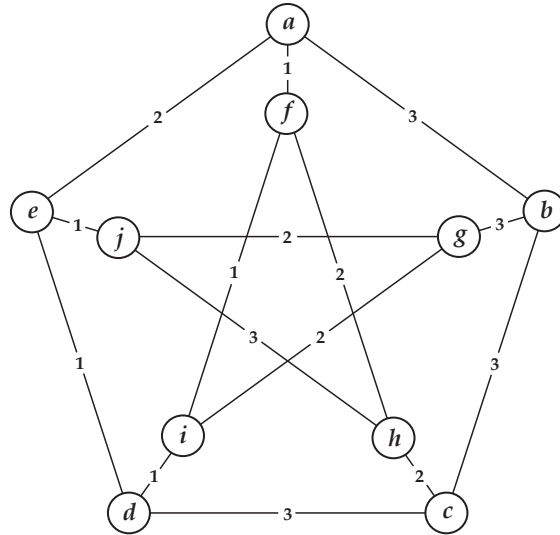
new  $M[1..n]$ 
for  $l \leftarrow 1$  to  $n$ 
    do for  $k \leftarrow 1$  to  $length[T_l]$ 
        do new  $C[1..length[T_l]]$ 
            for  $j \leftarrow 1$  to  $l - 1$ 
                do if  $t_j = T_l[k]$  and  $C[k] > M[j]$ 
                    then  $C[k] \leftarrow M[j]$ 
             $M[l] \leftarrow w_l$ 
            for  $j \leftarrow 1$  to  $l - 1$ 
                do  $M[l] \leftarrow M[l] + C[j]$ 
return  $M[i]$ 

```

The memoization solution may be faster if one only needs to ask one (or relatively few) queries, because it may avoid computing needless parts of the solution, which the dynamic programming always computes entirely.

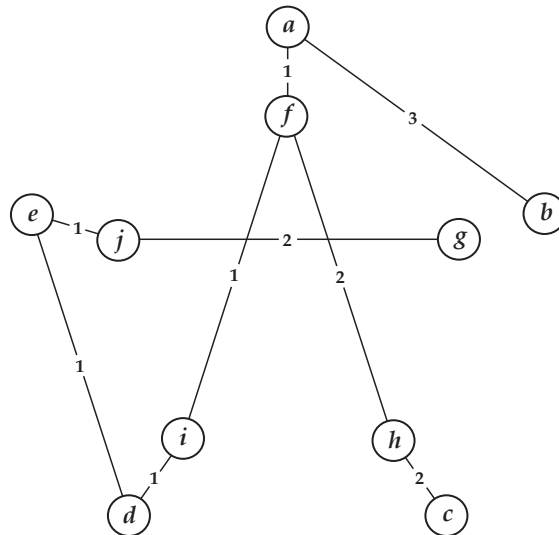
Problem 4: Graphs, Trees, Paths

Here is an undirected graph P with weighted edges:



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

Solution (a).

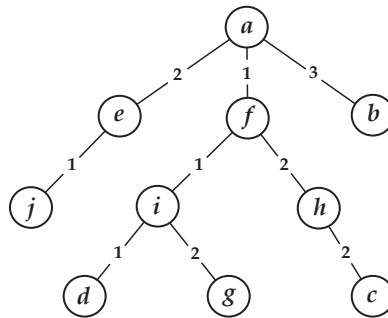


(This is just one of the possible solutions.)

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?

Solution (b). Since all edge weights are positive, a shortest path tree can be obtained by applying Dijkstra's algorithm.



Our shortest path tree is not a minimum spanning tree because the sum of the weights of all edges in this tree is 15. This is strictly larger than 14, the sum of the weights in the minimum spanning tree from (a).

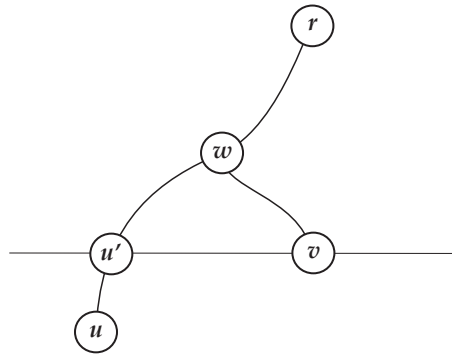
Let us now turn our attention to undirected trees without edge weights. In any tree T , there is exactly one simple path between any two vertices. (Recall that a path is *simple* if it never passes any vertex more than once.) Since trees are particular cases of graphs, one could use data structures for 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.

Solution (c). Given an undirected tree T , select arbitrarily a vertex $r \in T$ to make it into a rooted tree $T' = (T, r)$. We represent this by storing (a link to) the parent $\pi(v)$ with each node v as in shortest path trees. On top of that, we store $d(v)$, the depth of v in T' with each node v . All this clearly takes constant space per node and hence the total space taken by the extra data is $O(n)$. After this has been done, it need not be re-done for each simple path query, so we do not include any time needed for this step in our estimates.

Given two vertices $u, v \in T$, consider the paths p_u and p_v from u and v respectively to the root r of T' . Let w be the node where the two paths meet. Let p_{uw} and p_{vw} be the portions of p_u and p_v respectively from u and v to w . The simple path from u to v is then composed of p_{uw} and p_{vw} .

The procedure $SP(u, v)$ finds w as follows: Assume without loss of generality that $d(u) \geq d(v)$. We first follow the π links from u until we reach a node u' with $d(u') = d(v)$, and then synchronously follow the π links from u' and v until they meet at w .



To print out the path from u to v we just follow p_{uw} , then invert p_{vw} using a local array or a stack and follow the inverted variant of p_{vw} all the way to v .

The time taken by $SP(u, v)$ is clearly proportional to the sum of the lengths of p_{uw} and p_{vw} , and hence to f .