

Introduction to Algorithms and Data Structures

IT University of Copenhagen

June 17, 2002

This exam consists of 3 exercises containing in total 13 subexercises. Each of these 13 subexercises is given the same weight in the evaluation. The exam consists of 6 pages. You have 4 hours to complete the exam. Remember to number the pages and write your name and CPR number on every page.

For exercises in which algorithms must be specified, the asymptotic time complexity of the specified solution will be taken into account when grading. Exercises asking for time complexity must be answered using O -notation. It is weighted in the evaluation that growth rates in the O -notation are expressed with least possible asymptotic growth.

Exercise 1

This exercise is about rooted binary trees. The representation and notation for binary trees is similar to section 10.4 pp. 214-215 in CLRS.

Consider the following procedure, which as input takes the root x of a binary tree. We assume that all nodes x have a field $size[x]$ which contains an integer.

ZERO(x)

```
1 if  $x \neq \text{NIL}$ 
2   then  $size[x] \leftarrow 0$ 
3     ZERO(right[ $x$ ])
4     ZERO(left[ $x$ ])
```

The idea is that after the execution of the procedure all fields $size[x]$ will be 0.

a) Give the time complexity of the procedure ZERO(x) in O -notation, where x is the root of a tree with n nodes.

answer a)

$O(n)$

In the following subexercises we let $T(x)$ denote the *subtree rooted at x* in tree T cf. CLRS pp. 1087. Assume now that we want a procedure INITSIZE(x), which given the root x of a tree with n nodes initializes $size[y]$ to be the number of nodes in the subtree $T(y)$ for all nodes y in T .

b) Give the pseudocode for INITSIZE(x) such that the running time is $O(n)$.

answer b)

```
InitSize(x)
  if  $x = \text{NIL}$ 
  then return 0
  else  $size[x] \leftarrow 1 + \text{InitSize}(\text{left}[x]) + \text{InitSize}(\text{right}[x])$ 
  return  $size[x]$ 
```

The running time is clearly $O(n)$. Correctness can be proved using induction.

For a tree T , we say that an edge (u, v) with u parent to v is *green* if the number of nodes in $T(u)$ is at least two times the number of nodes in $T(v)$. That is, after the execution of INITSIZE, $size[u] \geq 2size[v]$ for all green edges (u, v) .

c) Describe a procedure which calculates the number of green edges for the entire tree T . Give the time complexity of your procedure and argue for your answer. Do also argue for the correctness of your algorithm.

answer c)

GreenEdges(x)

```
1  if x = NIL
2      then return 0
3  sum ← 0
4  if left[x] ≠ NIL and size[x] ≥ 2·size[left[x]]
5      then sum ← sum + 1
6  if right[x] ≠ NIL and size[x] ≥ 2·size[right[x]]
7      then sum ← sum + 1
8  sum ← sum + GreenEdges(left[x]) + GreenEdges(right[x])
9  return sum
```

Again correctness can be proved by induction. The time complexity of the algorithm is $O(n)$, which includes the initial call to `InitSize(x)`.

d) Using O -notation, give an upper bound on the number of green edges on a path from a node to the root of T . Argue for your answer.

answer d)

For each green edge (u, v) , where u is father to v is $size[u] \geq 2size[v]$. For two vertices v, w , where v is a leaf, w is a vertice on the path from the root to v and the distance between v and w is i it will be the case that $size[w] \geq 2^i size[v] = 2^i$. It follows that $i \leq \lg n$.

Consider the following procedure.

GREENPATHSUM(x)

```
1 if  $x \neq \text{NIL}$  and  $\text{left}[x] \neq \text{NIL}$  and  $\text{right}[x] \neq \text{NIL}$ 
2   then for  $i \leftarrow 1$  to  $\text{size}[x]$ 
3     do  $\text{timecount} \leftarrow \text{timecount} + 1$ 
4     if  $\text{size}[x] \geq 2\text{size}[\text{left}[x]]$ 
5       then GREENPATHSUM( $\text{left}[x]$ )
6       else GREENPATHSUM( $\text{right}[x]$ )
```

e) Give the time complexity of GREENPATHSUM(x). Argue for your answer.

answer e)

The recurrence

$$t(1) \leq c$$

$$t(n) \leq t\left(\frac{n}{2}\right) + n$$

suggests a running time of $O(n)$.

Exercise 2

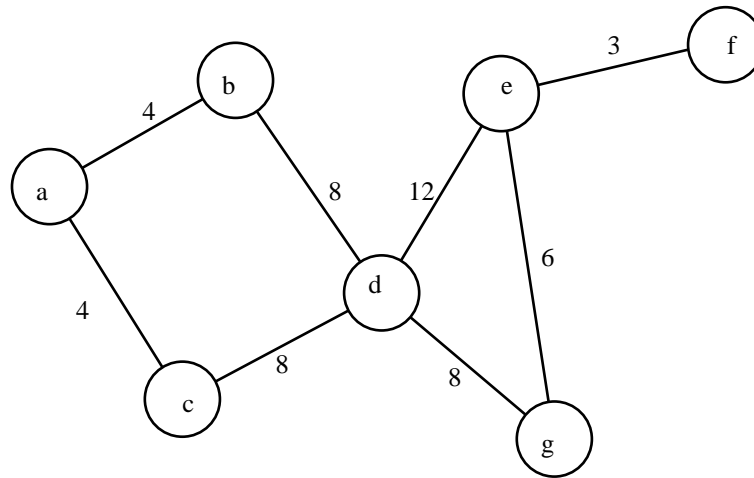


Figure 1: The graph H

a) Give the edges in a minimum spanning tree for the graph H in figure 1.

answer a)

$(a,b), (a,c), (c,d), (d,g), (g,e), (e,f)$

b) Give the adjacency-list representation for H in the same way as Figure 22.1 pp. 528 in CLRS.

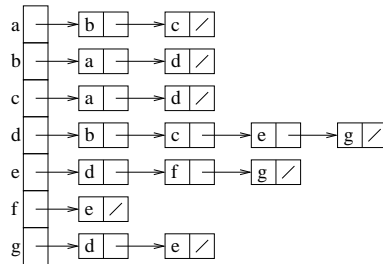
answer b)

There are two possible solutions:

adjacency-matrix representation

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>
<i>a</i>	0	4	4	0	0	0
<i>b</i>	4	0	0	8	0	0
<i>c</i>	4	0	0	8	0	0
<i>d</i>	0	8	8	0	12	0
<i>e</i>	0	0	0	12	0	3
<i>f</i>	0	0	0	0	3	0

or adjacency-list representation



Let G be a connected, undirected graph with edge weights $w(u, v)$ defined as the product of the degrees (as defined on page 1081 in CLRS) of the nodes u and v . That is, $w(u, v) = \deg(v) \cdot \deg(u)$, where $\deg(u)$ and $\deg(v)$ denotes the degree of node u and node v respectively. In the graph H in figure 1 the edge weights are defined in this way.

c) Give an efficient algorithm to compute a minimum spanning tree for graphs where the edge weights are defined as in G above.

answer c)

The weights of the edges must be integers between 1 and $(n - 1)^2$. If you use radix sort in the Kruskal algorithm an improvement in the running time can be obtained, since the time to sort the edges is now linear in the number of edges. The number of set-operations is now at most:

- n Make-Set operations (MST-Kruskal line 2 and 3)
- $2m$ Find-Set operations (MST-Kruskal line 6)
- at most $n-1$ Union operations (MST-Kruskal line 8)

This, according to CLRS p. 508, suggests a running time of $O(m\alpha(n))$, where m is the number of edges and n the number of vertices (this is if both union-by-rank and path compression is used).

The correctness follows from the fact that the only change in the Kruskal algorithm is the sorting algorithm used.

d) Let G be a given *directed* graph with positive edge weights and let v be a given node v in G . Describe an algorithm that computes the shortest simple cycle in G (i.e., *simple cycle* as in CLRS p. 1081) that contains v . Give the time complexity of your algorithm and argue for the correctness of your algorithm.

answer d)

The shortest simple cycle can be found in the following steps:

- a) Create a minimum spanning tree using Prim's algorithm starting in v .
- b) Remove the edges in the tree found from G .
- c) For each vertex in G investigate whether it has an edge back to v if this is the case mark it.
- d) Among the marked nodes and back-edges select the node u and edge e such that $w(v, u) + w(e)$ is minimized.
- e) The cycle can now be found by selecting all edges on the path from v to u (I assume u is the vertex selected in d). The edges can be found by following $\pi(u)$ and the edge selected in d).

The correctness of the algorithm can be proved by contradiction. The time for this algorithm is dominated by the call to Prim, which takes $O(m \log n)$ time (or $O(m + n \log n)$ if Prim's algorithm is implemented using Fibonacci heaps).

Exercise 3

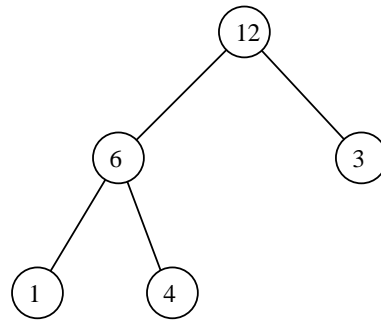
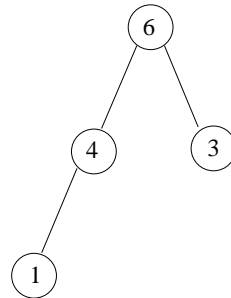


Figure 2: The heap B

A heap with five elements with the values 1, 3, 4, 6 og 12 is shown in Figure 2.

a) Draw the heap B after execution of a HEAP-EXTRACT-MAX operation.

answer a)



The following subexercises b) - d) are about extending the number of operations on heaps to include operations other than the ones described on page 129 in CLRS. The existing operations must still be supported by the heap within the same time complexity as on page 129 in CLRS. In the following we assume that the values of the elements are positive real numbers for which we can perform addition and division in constant time.

Consider the following operation.

FUSION (A, x, y): Removes from heap A the elements x og y (i.e., with the value $A[x]$ and $A[y]$) and adds a new element with the value $A[x] + A[y]$.

b) Give an algorithm for the operation FUSION. Argue for the running time.

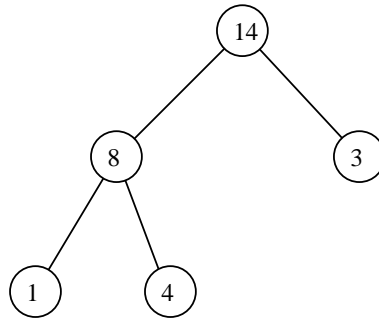
answer b)

First you take out the elements x and y with Heap-Delete. Next you add the new value $A[x] + A[y]$ to the heap with Heap-Insert. The running time is $O(\log n)$ since every call to the heap operations used takes time $O(\log n)$ and the time used to add the two numbers is constant. The correctness of the algorithm follows by the correctness of the operations used.

Consider the following new heap operation.

ADDUPTO (A, q, v): Adds the positive value v to all elements y in the heap A , where $A[y] \geq q$.

As an example, performing **ADDUPTO**($B, 6, 2$) on the heap B in figure 2 will give the new heap:



c) Describe an algorithm for the operation **ADDUPTO** such that the time complexity is $O(k)$, where k is the number of elements which change value after the execution of the operation.

answer c)

A procedure could look like:

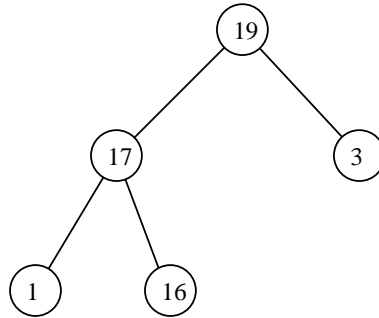
```
ADDUPTO(A, q, v)
1  if value[root[A]] ≥ q
2      then value[root[A]] ← value[root[A]] + v
3      ADDUPTO(left[A])
4      ADDUPTO(right[A])
```

The running time is clearly $O(k)$ since the recursion stops if there isn't any values to update in a subtree. For the same reason the algorithm is correct, since all the big values are updated when they are found in the root of a tree.

Consider the following operation:

ADDFUNNY(A, k): Adds the value $\frac{60}{A[x]}$ to the value of an element x if x is among the k largest elements in the heap A .

As an example, after the execution of `ADDFUNNY(3)` the heap B in figure 2 will contain five elements with the values 1, 3, 16, 17 and 19 since the three previously largest elements 12, 6 and 4 become 17, 16 and 19 respectively. After executing this operation, the heap could look as follows:



d) Describe an algorithm for the operation `ADDFUNNY(k)` such that the time complexity is $O(k \lg k)$. The time complexity of the other operations must be maintained.

answer d)

The challenge is to find the k largest elements in time $O(k \log k)$, which can be done in the required time in the following way.

- a) First mark the root of the heap. Count down a counter c that initially is k
- b) If $c = 0$ you are done
- c) Put both of the children of the previously marked element in a maxheap M
- d) Mark and remove the largest element from M , count down c and go to b

There will be done $O(k)$ operations of c) and d), each taking $O(\log k)$. Now the elements to do the required operation on is selected in time $O(k \log k)$.

After that the value $\frac{60}{A[x]}$ is added. Remark that the ordering of the k elements can be changed when this is done, but since only the k largest elements are affected, the ordering of the rest of the heap is not changed. Sorting the k elements will maintain the heap property in time $O(k \log k)$. All of the operations took $O(k \log k)$.