CIS 575: Introduction to Algorithm Analysis
Exam II with suggested answers

April 10, 2013, 2:30-3:20pm

General Notes

- You can have one sheet (each side may be used) of notes, but no other material and no use of laptops or other computing devices.
- If you believe there is an error or ambiguity in any question, mention that in your answer, and *state your assumptions*.
- Please write your name on this page.

Good Luck!

NAME:

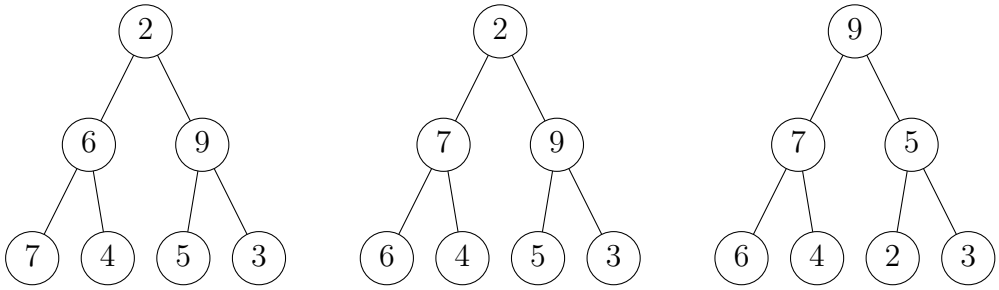
1. *Heaps, 20p.* Consider A as given below (where $A[1] = 2$, etc.)

1	2	3	4	5	6	7
2	6	9	7	4	5	3

Recall that we can view such an array as a binary tree, with $A[i]$ the parent of $A[2i]$ and of $A[2i+1]$ for each $i \in \{1, 2, 3\}$.

Your task is to convert this array into a binary heap which you should draw as a binary tree (whose root will be 9). To do so, it may help you (but is not required) to first draw the tree (whose root is 2) that corresponds to A, and next draw the result of converting each child of the root into a heap.

Answer: First we convert the root's children into heaps; next we sift down the root:



2. *Divide & Conquer, 25p.* We are given the task of finding the sum of the elements in $A[1..n]$ where $n \geq 1$. That is, we must compute $\sum_{i=1}^{n} A[i]$, also written $A[1] + A[2] + ... + A[n-1] + A[n]$.

Write (in pseudo-code) an algorithm that implements this specification and is based on the Divide & Conquer paradigm. The "divide" part should split the problem into 2 problems of approximately half the size.

You do not need to argue for correctness, but you must analyze the running time of your algorithm. (You may assume that addition is in $\Theta(1)$.)

Answer:

```
SUM(A[1..n])
  if n = 1
    return A[1]
  else
    x ← SUM(A[1..n/2])
    y ← SUM(A[n/2+1..n])
    return x+y
```

The running time $T(n)$ is described by the recurrence $T(n) \in 2T(\frac{n}{2}) + \Theta(1)$ and hence we infer (since $\lg_2(2) = 1 > 0$) that $T(n) \in \Theta(n^1) = \Theta(n)$.

3. *Graphs, 30p.* Consider the algorithm below whose input is a directed graph G with nodes 1..n and with a edges. We assume that G is represented as an adjacency *matrix*. Observe that G has an edge from i to j iff G.GET(i,j) does *not* return nil.

```
count ← 0
for i ← 1 to n
    b ← true
    j ← 1
    while j ≤ n and b
        if G.GET(i,j) = nil
            b ← false
        else
            j ← j+1
    if b
        count ← count + 1
return count
```

a. (10p) Briefly explain what this algorithm computes. (That is, how the final value of "count" depends on the shape of G.)

Answer: The algorithm computes the number of nodes from which there are edges to all nodes (including itself).

Continued on next page

Question 3, continued

b. (10p) Express, using Θ-notation, the time it takes to run (once) the body of the **while** loop. You do not need to justify your answer.

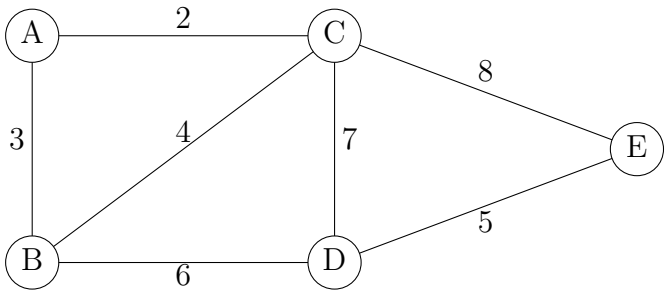
Answer: With a matrix representation, GET can be implemented to run in constant time, and hence the body of the **while** loop will run in time $\Theta(1)$.

c. (10p) Find an upper bound, as tight as possible, for the total number of times that the body of the **while** loop may execute. You should briefly argue for your answer which should be a function of n and of a. (Observe that you should give an exact estimate and *not* use O or Θ notation.)

Answer: Observe that the worst case is when each node i has edges to the nodes with low numbers but no edges to the nodes with high numbers. In that case, for each i the **while** loop will iterate one more time than the number of outgoing edges from i. Hence the total number of iterations will be $n + a$.

Don't forget Question 4 on the back page!

4. *Greedy Algorithms, 25p.* Consider the undirected graph given below where each edge has been assigned a cost:



Sketch the steps performed by Kruskal's algorithm when constructing a minimal-cost spanning tree for this graph. You do not need to explicitly draw the union/find trees, but for each edge you must argue why or why not it is included in the spanning tree.

Answer: We construct the minimum-cost spanning tree T by processing the edges after increasing cost. Initially, we add the edge between A and C, and next the edge between A and B. As a result, the union/find structure has A,B,C in the same component; hence the edge between B and C is ignored. Then the edge between D and E is added to T; finally, the edge between B and D is added. At this points, the union/find structure has all nodes in the same component; hence no more edges will be added. We get the tree, marked in red: