

CS 141 (Closed-book) Midterm Test

Feb. 13, Wed., 1:10 - 2pm, 2008

Total: 40 points

QUESTION 1. [10 pts] Consider the following recursive algorithm:

```
Algorithm LoopAndRecurse(A[1..n]);
  var i: integer;
  begin
    if n > 1 then
      for i := n-1 downto 1 do
        A[n] := A[n] + A[i];
      call LoopAndRecurse(A[1..n-1])
    end;
```

Let $T(n)$ denote the (worst-case) time complexity of algorithm LoopAndRecurse. Analyze the algorithm to obtain a tight asymptotic bound on $T(n)$ using a recurrence relation.

Answer:

$$\begin{aligned} T(n) &= T(n-1) + n - 1 && 3\text{pts} \\ &= T(n-2) + (n-2) + (n-1) && 2\text{pts} \\ &= \dots \\ &= T(n-i) + (n-i) + (n-i+1) + \dots + (n-1) && 1\text{pt} \\ &= T(1) + 1 + 2 + \dots + n - 1 && 1\text{pt} \\ &= 0 + \sum_{i=1}^{n-1} i && 1\text{pt} \\ &= \Theta(n^2) && 2\text{pts} \end{aligned}$$

It's okay to use the Θ notation in early steps. Not all intermediate steps are necessary.

QUESTION 2. [10 pts total] Let $A[1..n]$ be an array of integers. The *prefix max* problem asks for an array $B[1..n]$, where $B[i] = \max\{A[1], A[2], \dots, A[i]\}$. That is $B[i]$ records the maximum value in $\{A[1], A[2], \dots, A[i]\}$. The following algorithm solves the prefix max problem.

```

Algorithm PrefixMax(A[1..n]: integer);
  var i,j: integer;
  begin
    for i := 1 to n do
      B[i] = A[1];
      for j := 2 to i do
        if A[j] > B[i] then B[i] := A[j];
      end;
    end;
  end;

```

- (a) [3 pts] What is the time complexity of algorithm PrefixMax?
- (b) [7 pts] Observing that $B[i] = \max\{A[i], B[i-1]\}$, for all $i > 1$, design another algorithm with an improved time complexity. Also analyze the time complexity of this improved algorithm.

Answer:

The time complexity of this algorithm is (3 pts)

$$\sum_{i=1}^n \Theta(i) = \Theta\left(\sum_{i=1}^n i\right) = \Theta(n^2).$$

Observing that $B[i] = \max\{A[i], B[i-1]\}$, for all $i > 1$, we can improve the above algorithm as follows: (5 pts)

```

Algorithm PrefixMax(A[1..n]: integer);
  var i,j: integer;
  begin
    B[1] = A[1];
    for i := 2 to n do
      if A[i] > B[i-1] then B[i] := A[i]
      else B[i] := B[i-1];
    end;
  end;

```

The time complexity of this improved algorithm is clearly $\Theta(n)$. (2 pts)

QUESTION 3. [10 pts] Let $G = (V, E)$ be a DAG, where $|V| = n$ and $|E| = m$. A vertex v is called a *root* of G if there is a path from v to every vertex in G . A DAG may or may not have a root in general. Describe briefly an $O(n + m)$ time algorithm to check if G has a root. How should G be represented in order to achieve the desired time complexity of $O(n + m)$?

You may call the topological sorting algorithm learned in class without giving its definition. Your description of the algorithm can be on a very high level (*e.g.* you could just list the sequence of major steps), as long as you have enough details to analyze the time complexity.

Hint: A DAG may have at most one root. This question is very similar to Question 5 in your second homework.

Answer:

The idea is to run topological sorting on G and then check if the first vertex (instead of the last vertex) in the sorted order has a path to every other vertex in G by starting a DFS or BFS at this vertex.

If G is represented as adjacency lists, then this algorithm runs in time $O(n + m)$ since both topological sorting and DFS/BFS take $O(n + m)$ time.

Give $3+4 = 7$ pts for the algorithm (a brief sketch or some kind of pseudocode) and 3 pts for the time complexity analysis.

QUESTION 4. [10 pts] Let A and B be two sets of integers, represented as unsorted arrays. Suppose that $|A| = n$, $|B| = m$, and $n \leq m$. Write an efficient algorithm (in high-level pseudocode) to decide if A and B contain any common elements in $O(m \log n)$ time. Also analyze the time complexity of your algorithm to make sure that it is really $O(m \log n)$.

Hint: Sorting and binary search. You may call any sorting algorithm with providing a definition.

Answer:

This question is similar to the last question in your homework 1 in spirit, but slightly easier. We sort A into an ascending array by Mergesort or Heapsort, and then check if each of the elements of B appears in A by binary search.

The sorting takes time $O(n \log n)$ and the binary searches take a total time of $O(m \log n)$. Since $n \leq m$, $O(n \log n) + O(m \log n) = O(m \log n)$.

Give $4+4 = 8$ pts for the algorithm (or some sketch of it) and 2 pts for the time complexity analysis. Deduct 2 pts for sorting both A and B .