Math 188 Midterm Solutions 2/11/00

Name: Student ID: Section:

Conditions: No notes, books, consultation, or any kind of assistance allowed. The time limit will be announced. Be sure to show your work where indicated.

1. This innocent-looking piece of dynamic programming code does a surprising task:

Input: Array A of size n
Set $x = 0$ and set $y = 0$
For $i = 1$ to n do
  $x = \max(x + A[i], 0)$
  $y = \max(x, y)$
  Print $A[i]$, x, and y
Return y

(2 pts) (a) Trace the above algorithm on the input $A[1], \ldots, A[5] = (1, -2, 3, 4, -5)$, printing out the values of $A[i]$, x, and y whenever the line "Print $A[i]$, x, and y" occurs. Do this in a table with columns labeled by values of i, and rows labeled by $A[i]$, x, and y.

i	1	2	3	4	5
$A[i]$	1	-2	3	4	-5
x	1	0	3	7	2
y	1	1	3	7	7

(2 pts) (b) Briefly describe what this algorithm does on a general array of integers. (hint: test on other arrays if necessary.)

This algorithm finds the largest sum of any subsequence of $A[1], A[2], \ldots, A[n]$.

(2 pts) (c) Interpret the value printed for y when $i = 2$.

When $i = 2$, y is the largest sum of any subsequence of $A[1], A[2]$ which in this case is just the value of $A[1]$.

2. Use the reverse of the page if necessary. You may recall this definition of binomial coefficients:

$$C(n,k) = \left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k!(n-k)!}. \tag{1}$$

However, we want to analyze a procedure Binom(n,k) which computes $C(n,k)$ according to the recursive definition:

$$C(n,k) = \begin{cases} C(n-1,k) + C(n-1,k-1) & \text{if } n \geq k \\ 1 & \text{if } k = 0 \\ 0 & \text{if } k > n. \end{cases} \tag{2}$$

(2 pts) (a) Make a table of values of $C(n,k)$ for $n,k = 0,\ldots,4$. Construct it so that n runs across the columns and k runs down the rows.

k, n	0	1	2	3	4
0	1	1	1	1	1
1	0	1	2	3	4
2	0	0	1	3	6
3	0	0	0	1	4
4	0	0	0	0	1

(2 pts) (b) Imagine a procedure Binom(n,k) that computes $C(n,k)$ by making recursive calls according to (2). For instance, if the procedure call Binom$(5,3)$ is made, it recursively makes calls to Binom$(4,3)$ and Binom$(4,2)$. What is a good lower bound (Ω) to the number of procedure calls for Binom$(n,3)$? (hint: compute $C(n,3)$ using (1) and consider that the recursion for Binom(n,k) only terminates with values of 1 and 0.)

Using the hint, we have Binom$(n,3) = C(n,3) = \frac{n!}{3!(n-3)!} = \frac{n(n-1)(n-2)}{3 \cdot 2 \cdot 1} = O(n^3)$. Since Binom$(n,k)$ recursively calls itself until it is adding 1's and 0's, Binom$(n,3)$ must make at least $O(n^3)$ calls that result in returning 1. So Binom$(n,3)$ is $\Omega(n^3)$.

(4 pts) (c) Use dynamic programming to construct an $O(n^2)$ algorithm in pseudocode for computing $C(n,k)$, for $k \leq n$. (aside: notice the difference in time complexity in the two algorithms.)

```
for i = 0 to n
    for j = 0 to k
        if j = 0 then C(i,j) = 1
        if j > i then C(i,j) = 0
        else C(i,j) = C(i-1,j) + C(i-1,j-1)
```

Since the first **for** loop is from 0 to n and the second is from 0 to k where $k \leq n$, and each step in the second **for** loop takes constant time, the total time is $O(n^2)$.

(6 pts) 3. Given two unsorted sets of integers S_1 and S_2 (each of size n), describe an $O(n \log n)$ algorithm for determining if S_1 and S_2 have any elements in common. Your description should be in pseudocode or short English sentences. For full credit clearly indicate and justify the time complexity of each main step in your algorithm as well as the total time complexity.

Step 1: Sort S_1 and S_2. Each sort takes $O(n \log n)$ so Step 1 takes $O(n \log n)$.

Step 2: Let the sorted elements of S_1 and S_2 be $S_1[1], S_1[2], \ldots, S_1[n]$ and $S_2[1], S_2[2], \ldots, S_2[n]$.

Set $i = j = 1$.
While $i \leq n$ and $j \leq n$ do
    If $S_1[i] = S_2[j]$ then return "S_1 and S_2 have a common element."
    If $S_1[i] < S_2[j]$ then increment i and loop
    If $S_1[i] > S_2[j]$ then increment j and loop
Return "S_1 and S_2 have no elements in common."

Since i and j each range at most from 1 to n, Step 2 takes $O(n)$. The total time complexity is therefore $O(n \log n) + O(n) = O(n \log n)$.

Another possibility for Step 2 is to assume that in Step 1 the two lists have been sorted into balanced binary tree data structures so that retrieval takes $O(\log n)$. Then start with the first element in the sorted S_1 and search S_2 for that element. If you don't find it repeat with the second element of S_1. Worst case you will perform n searches that take $O(\log n)$ each, for a total of $O(n \log n)$. Added to Step 1, the total complexity is still $O(n \log n)$.