CS60003 Algorithm Design and Analysis, Autumn 2009–10

Mid-Semester Test

Maximum marks: 45 September 20, 2009 Total time: 2 hours

Roll no: ____________ **Name:** ______________________________

[*Write your answers in the question paper itself. Be brief and precise. Answer all questions.*]

1. Consider the following function that accepts a positive integer n as input.

```
playWith (n)
{
    while ($n > 1$) {
        Increment $n$ by $1$.
        while ($n$ is even), set $n = n/2$.
    }
}
```

(a) Prove that the function terminates for every input integer $n \geqslant 1$. **(5)**

Solution Clearly, the function terminates for $n = 1$. Except perhaps in the first iteration of the outer while loop, the loop body starts with an odd value of n. In the loop body, the value of n changes to $(n+1)/2^k$ for some $k \geqslant 1$. We have $(n+1)/2^k \leqslant (n+1)/2 < n$ for all $n > 1$, that is, each iteration strictly reduces the value of n.

(b) Determine a tight bound on the running time of the above function. You should supply an argument to corroborate that your bound is tight (that is, achievable). **(5)**

Solution The worst case occurs when the reduction of n (by division) occurs only once in the inner while loop. But for all $n > 3$, we have $(n+1)/2 < (2/3)n$. The first iteration may change n to $n+1$ (if the initial value of n is even). After that, at most $\log_{3/2}(n+1)$ iterations of the outer loop reduce n to a value $\leqslant 3$. For $n = 3$, the outer while loop is executed only once (twice for $n = 2$, but this case does not arise unless we start with $n = 2$). Since the order notation hides constant multiplicative factors, we can change the base of logarithms from $3/2$ to any other suitable constant value. Moreover, since each increment and division can be performed in unit time, the running time of playWith(n) is $O(\log n)$.

This bound is tight, since $\Omega(\log n)$ iterations are needed, for instance, for $n = 2^t + 1$.

2. **(a)** Let S and T be strings each of length n. Your task is to determine whether T can be obtained by cyclically rotating S. For example, the string star can be obtained by cyclically rotating the string tars, whereas the string arts cannot be obtained by cyclically rotating the string tars. Supply an $O(n)$-time algorithm to solve this problem. **(5)**

Solution Let $S = a_0 a_1 \ldots a_{n-1}$. Search for T in the string $a_0 a_1 \ldots a_{n-1} a_0 a_1 \ldots a_{n-2}$ of length $2n - 1$. Use the KMP string matching algorithm.

(b) Let S and T be strings of lengths m and n respectively. Your task is to determine whether T is a sub-sequence of S, that is, whether the symbols of T occur in S in the same order as they appear in T, but not necessarily contiguously. For example, the string grim is a sub-sequence of the string algorithm, whereas the string gram is not. Supply an $O(m + n)$-time algorithm to solve this problem. **(5)**

Solution

```
Initialize $i = j = 0$.
while ($i < m$) {
    If ($S[i]$ equals $T[j]$) {
        increment $j$ by $1$.
        If ($j$ equals $n$), return "success".
    }
    Increment $i$ by $1$.
}
Return "failure".
```

3. Consider the line-sweep algorithm for computing line-segment intersections. Suppose that some of the input segments are allowed to be vertical (that is, parallel to the y-axis). You may, however, assume that no two given vertical segments are collinear.

(a) Define a new type of event "Vertical Segment" to deal with this situation. Describe how you can efficiently handle this event. You are required to maintain the running time of the original algorithm. **(5)**

Solution The event queue Q contains a single event for each vertical segment L_i. When this event occurs, we delete the event from Q. Suppose that the sweep line information S is maintained as a height-balanced binary search tree with top-to-bottom ordering of active segments. We locate the insertion position of the upper end point of L_i in S. From this position, we keep on looking at successors in S, until the successor lies below the bottom end point of L_i. Let all these successors be $L_{j_1}, L_{j_2}, \ldots, L_{j_r}$. We report the intersection of L_i with each of $L_{j_1}, L_{j_2}, \ldots, L_{j_r}$.

(b) Prove that you achieve the original running time of $O((n+h)\log n)$ in presence of "Vertical Segment" events. You may assume that a height-balanced binary search tree on k nodes can be so implemented that the predecessor or successor of a node in the tree can be located in $O(\log k)$ time. **(5)**

Solution Suppose that there are h_1 intersection points among non-vertical segments, and there are h_2 intersection points involving vertical segments. The effort spent to identify the former h_1 intersection points remains $O((n+h_1)\log n)$ as in the original algorithm. For a vertical segment L_i, the determination of r intersection points (with $L_{j_1}, L_{j_2}, \ldots, L_{j_r}$) takes $O(r \log n)$ time, since the size of S is always $\leqslant n$. So the total effort associated with handling all "Vertical Segment" events is $O(h_2 \log n)$. Consequently, the total running time is $O((n+h_1+h_2)\log n)$, that is, $O((n+h)\log n)$.

4. You are given a set of n real numbers $x_1, x_2, \ldots, x_n$. Your task is to cover these points by intervals of unit length (that is, by intervals of the form $[a, a+1] = \{x \in \mathbb{R} \mid a \leqslant x \leqslant a+1\}$ for real numbers a). Your goal is to minimize the number of intervals in the cover. (Here is a practical application of this problem. Suppose that the points x_i represent houses on a straight road. You want to cover all the houses by a set of cellular-phone towers each with a maximum range of $1/2$ km in each direction. Naturally, you attempt to serve all the houses with as few towers as possible.)

(a) Consider the following greedy strategy. Choose an interval of unit length to cover the maximum number of points in the given collection. Output this interval, and remove the points covered by this interval from the collection. (Serve the maximum possible number of houses by a single tower.) Repeat until no points are left. Give an example to demonstrate that this greedy algorithm may fail to provide an optimal solution. **(5)**

Solution Consider the points $-1, -0.5, -0.1, 0.1, 0.5, 1$. It is easy to see that no unit interval covers five or more of these points. The greedy strategy first chooses the interval $[-0.5, 0.5]$ to cover the most congested part (that is, the four central points). This is indeed the only unit interval which contains four of the six given points. But then two other intervals are needed, one to cover -1, and the other to cover 1. That leads to a collection of three intervals.

On the contrary, only two intervals $[-1, 0]$ and $[0, 1]$ suffice to cover all of the given six points.

(b) Although the greedy strategy of Part (a) fails, there exist other greedy strategies that efficiently compute optimal solutions. Describe such a strategy. The running time of your greedy algorithm must be bounded by a polynomial in n. **(5)**

Solution First, sort the input points in increasing order. This takes $O(n \log n)$ time. Suppose that the sorted sequence is $x_1, x_2, \ldots, x_n$ itself.

Let x_i be the leftmost point yet to be covered (initially, $i = 1$). Output the interval $[x_i, x_i + 1]$, and remove all the points that lie in the interval. Repeat until no points are left.

The algorithm can be implemented to run in $O(n)$ time (after the initial sorting phase, of course).

(c) Prove that your greedy algorithm correctly computes an optimal solution. **(5)**

Solution Suppose that an optimal solution is provided to us. Assume that $x_1 \leqslant x_2 \leqslant \cdots \leqslant x_n$. There exists (at least) one interval I_1 in the optimal solution, that covers x_1. Take $I_1' = [x_1, x_1 + 1]$. Clearly, I_1' covers no fewer points than I_1.

Suppose that x_i is the leftmost point not covered by I_1'. In the optimal solution, (at least) one interval I_2 covers x_i. We must have $I_2 \neq I_1$, since otherwise I_1' would have covered x_i. Take $I_2' = [x_i, x_i + 1]$. Clearly, $I_1' \cup I_2'$ covers no fewer points than $I_1 \cup I_2$.

Proceeding in this fashion, we can convert the optimal solution to a greedy solution without ever increasing the number of intervals. Since the original collection of intervals was optimal, the converted greedy collection must be optimal too (and must contain the same number of intervals as the original optimal collection.)

ROUGH WORK