CS21003 Algorithms I, Autumn 2012–13

Mid-semester Test

Maximum marks: 34 | Time: 25-Sep-2012 | Duration: 2 hours

Roll no: _____ **Name:** _____

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

1. Write brief answers (one or two sentences each) to the following six parts. **(2×6)**

(a) Suppose that the running time $T(n)$ of a recursive algorithm is an increasing function of the input size n, and satisfies the recurrence $T(n) = 10T(n/3) + \Theta(n^2)$ whenever n is a power of three. Find $T(n)$ as $\Theta(f(n))$ for some simple function $f(n)$.

$\Theta(n^{\log_3 10})$ (since $\log_3 10 > \log_3 9 = 2$)

(b) What is the minimum number of nodes in an AVL tree of height three?

$F_{3+3} - 1 = F_6 - 1 = 8 - 1 = 7$ (using the standard formula for Fibonacci trees)

(c) What is the maximum number of nodes in an AVL tree of height three?

$1 + 2 + 2^2 + 2^3 = 15$ (this is the full binary tree of height three)

(d) You are given a sorted array A of size n. For simplicity, assume that n is a power of three. You want to find whether a value a is present in A. You first compute the two indices $M_1 = n/3$ and $M_2 = 2n/3$. If $a < A[M_1]$, recursively search for a in $A[0 \ldots M_1 - 1]$, else if $a < A[M_2]$, recursively search for a in $A[M_1 \ldots M_2 - 1]$, else recursively search for a in $A[M_2 \ldots n - 1]$. This is called the *ternary search* algorithm. Write the recurrence for the running time of this algorithm.

$T(n) = T(n/3) + \Theta(1)$

(e) What is the number of comparisons made by the ternary search algorithm of Part (d) in the worst case?

$2 \log_3 n + 1$ (two comparisons in each iteration and a final check for equality outside the loop)

(f) Prove or disprove: The minimum value of a max-heap must be found in a leaf node. (Assume that the values stored in the heap are distinct from one another.)

True. A non-leaf node has at least one child storing a strictly smaller value.

2. You are given a max-heap of n elements in the contiguous array representation. Your task is to create a binary tree storing the same heap in the standard pointer-based representation. Write a $\Theta(n)$-time function to perform this task. **(6)**

Solution The following recursive function performs this task. We assume that we maintain only the left and right child pointers in each node of the output tree.

```
treenode *heap2tree ( int *A, int n, int i )
{
   treenode *p;

   if (i >= n) return NULL;
   p = (treenode *)malloc(sizeof(treenode));
   p -> value = A[i];
   p -> left = heap2tree(A,n,2*i+1);
   p -> right = heap2tree(A,n,2*i+2);
   return p;
}
```

The outermost call should be at the root:

```
T = heap2tree(A,n,0);
```

Roll no: _____ **Name:** _____

3. You are given an array A of n non-zero floating-point numbers $a_0, a_1, a_2, \ldots, a_{n-1}$. Let P be the product of all the elements of A, that is, $P = a_0 a_1 a_2 \cdots a_{n-1}$, and $b_i = P/a_i = a_0 \cdots a_{i-1} a_{i+1} \cdots a_{n-1}$ for $i = 0, 1, 2, \ldots, n-1$. You want to compute an array B storing $b_0, b_1, b_2, \ldots, b_{n-1}$. You are writing the code for a mobile phone whose processor is so primitive that floating-point division operations are not permitted. You are also not allowed to use any math library call (like **`pow(A[i],-1)`**). Write an $O(n \log n)$-time algorithm for solving the problem assuming that addition, subtraction, multiplication, comparison and assignment are the only allowed operations (on integers and floating-point numbers). Deduce that the running time of your algorithm is $O(n \log n)$. **(6)**

Solution We use the following divide-and-conquer approach to solve this problem. Here, we assume that the mobile phone supports pointer arithmetic. If not, the following code can be easily rewritten by passing the start and last indices in a subarray.

```
void mobAppl ( float *A, int n, float *B )
{
   int i, m, M;
   float s, t;

   if (n == 1) { B[0] = 1; return; }
   /* Compute m = n / 2 (floor) without the division */
   m = M = 0; while (M <= n - 2) { ++m; ++M; ++M; }
   mobAppl(A, m, B);
   mobAppl(A + m, n - m, B + m);
   s = A[0] * B[0]; /* We have s = a_0 a_1 ... a_{m-1} */
   t = A[m] * B[m]; /* We have t = a_m a_{m+1} ... a_{n-1} */
   for (i = 0; i < m; ++i) B[i] *= t;
   for (i = m; i < n; ++i) B[i] *= s;
}
```

If $T(n)$ denotes the running time of this algorithm on an array of size n, we have

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n \geqslant 2 \end{cases}$$

whenever n is a power of two. By the master theorem for divide-and-conquer recurrences, we have $T(n) = \Theta(n \log n)$.

4. A *bipartite graph* is a (simple) undirected graph $G = (V, E)$ whose vertex set V can be partitioned in two disjoint subsets V_1 and V_2 such that $V = V_1 \cup V_2$, no two vertices in V_1 are connected by an edge, and no two vertices in V_2 are connected by an edge. You are given a connected bipartite graph (V, E), but the sets V_1 and V_2 are not supplied to you. Design an efficient algorithm to compute V_1 and V_2. Deduce the running time of your algorithm. Also prove the correctness of your algorithm. **(10)**

Solution Let us name the vertices of G as $0, 1, \ldots, n-1$. We assume that the graph G is supplied in the adjacency-list representation, that is, we can retrieve all the neighbors of any node v in time proportional to the number of neighbors of v. We let m denote the number of edges in G. Since G is connected and bipartite, we have $n - 1 \leqslant m \leqslant n^2/4$. The following pseudocode builds the parts V_1 and V_2 iteratively. An array **`included`** indexed by V shows which vertices are already included in the parts V_1 or V_2 (0 means *not included*, 1 means *included in* V_1, and 2 means *included in* V_2). A queue Q of unprocessed vertices is maintained.

```
Set Q = (0), V1 = {0}, V2 = ∅, and included[i] = 1.
For i = 1, 2, ..., n − 1, set included[i] = 0.
While (Q is not empty) {
   Let u be the front of Q, and set Q = DEQUEUE(Q).
   For each neighbor v of u {
      If (included[v] equals 0) {
         Decide the part for v by setting i = 3 − included[u].
         Set V_i = V_i ∪ {v}, included[v] = i, and Q = ENQUEUE(Q, v).
      }
   }
}
```

Running time: The initialization takes $\Theta(n)$ time. We assume that each basic operation on the queue Q can be done in $O(1)$ time. Since each vertex is enqueued once and dequeued once, the total effort associated with the queue Q is $\Theta(n)$. Under the adjacency-list representation of G, the remaining task in the **`While`** loop takes $\Theta(m)$ time (where m is the number of edges in G). So the total running time of the above algorithm is $O(n + m) = O(|V| + |E|)$.

Correctness: A bipartite graph cannot contain a cycle of odd length. For example, if $(v_0, v_1, v_2, \ldots, v_{2k})$ is a cycle in a bipartite graph G, then $v_0, v_2, v_4, \ldots, v_{2k}$ must belong to one part, and $v_1, v_3, v_5, \ldots, v_{2k-1}$ in the other part. But then, there is an edge between v_0 and v_{2k}, a contradiction.

Suppose that v and v' are included in V_2 by the above algorithm as neighbors of u and u' in V_1, respectively. There exist intermediate vertices $u_1, u_2, \ldots, u_k$ in V_1 and $v_1, v_2, \ldots, v_k$ in V_2 such that v_i is included as a neighbor of u_i for $i = 0, 1, 2, \ldots, k+1$, and u_{i+1} is included as a neighbor of v_i for $i = 0, 1, 2, \ldots, k$, where we have renamed $u = u_0$, $u' = u_{k+1}$, $v = v_0$ and $v' = v_{k+1}$. (If $u_1 = u_2$, then $k = 0$.) If G contains the edge (v, v'), then $(v_0, u_1, v_1, u_2, v_2, \ldots, u_{k+1}, v_{k+1})$ is a cycle of odd length in G, that is, G is not bipartite. So there cannot exist an edge between two vertices included in V_2. Likewise, we can prove that there cannot exist an edge between two vertices included in V_1.

Finally, note that since G is connected, each vertex in V is included in either V_1 or V_2.

Roll no: _____ **Name:** _____

For rough work and leftover answers