Mid-Semester Test

Maximum marks: 50 | September 26, 2008 | Total time: 2 hours

---

1. Suppose that the running time $T(n)$ of an algorithm on an input of size n satisfies

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + cn \log n$$

for all $n \geqslant 2$, where c is a positive constant. Deduce that $T(n) = \Theta(n \log^2 n)$. **(10)**

Solution **Step 1:** First show, by induction on n, that $T(n)$ is an increasing function of n. This implies that $T(2^t) \leqslant T(n) \leqslant T(2^{t+1})$, where $2^t \leqslant n < 2^{t+1}$. **2**

Step 2: Solve the recurrence for $n = 2^t$. **4**

$$\begin{aligned} T(2^t) &= 2T(2^{t-1}) + c't2^t \qquad \text{(where } c' = c\log 2 > 0 \text{ is a constant)} \\ &= 2[2T(2^{t-2}) + c'(t-1)2^{t-1}] + c't2^t \\ &= 2^2T(2^{t-2}) + c'[(t-1)+t]2^t \\ &= 2^2[2T(2^{t-3}) + c'(t-2)2^{t-2}] + c'[(t-1)+t]2^t \\ &= 2^3T(2^{t-3}) + c'[(t-2)+(t-1)+t]2^t \\ &\cdots \\ &= 2^tT(1) + c'[1+2+\cdots+(t-2)+(t-1)+t]2^t \\ &= d2^t + c't(t+1)2^{t-1} \qquad \text{(where } d = T(1) \text{ is a positive constant)} \\ &= (c't^2 + c't + 2d)2^{t-1}. \end{aligned}$$

Step 3: Upper bound
Consider n in the range $2^t \leqslant n < 2^{t+1}$. We have

$$T(n) \leqslant T(2^{t+1}) = (c'(t+1)^2 + c'(t+1) + 2d)2^t \leqslant (c'(\lg n + 1)^2 + c'(\lg n + 1) + 2d)n.$$

It follows that $T(n) = O(n \log^2 n)$. **2**

Step 4: Lower bound
For n satisfying $2^t \leqslant n < 2^{t+1}$, we have

$$T(n) \geqslant T(2^t) = (c't^2 + c't + 2d)2^{t-1} \geqslant (c'(\lg n - 1)^2 + c'(\lg n - 1) + 2d)\frac{n}{4}.$$

Therefore, $T(n) = \Omega(n \log^2 n)$. **2**

Let M denote the maximum of these absolute differences, and m the minimum of them. The problem of determining M (resp. m) is called the maximum-difference (resp. minimum-difference) problem.

(a) Design an $O(n)$-time algorithm to compute M. **(5)**

Solution **The algorithm:** **3**

First, obtain the minimum element a_s in the array.
Then, obtain the maximum element a_t in the array.
Finally, return $a_t - a_s$.

Correctness: Assume $a_i \geqslant a_j$. Then, $|a_i - a_j| = a_i - a_j$ is maximized, when a_i is as large as possible and a_j is as small as possible. **1**

Running time: The minimum of an array of n elements can be found in $O(n)$ time. Similar is the case for the maximum. **1**

(b) Design an $O(n \log n)$-time algorithm to compute m. **(5)**

Solution **The algorithm:** **3**

Merge sort the array A in ascending order.
Let $a_{i_1}, a_{i_2}, \ldots, a_{i_n}$ be the sorted version of A.
Compute and return the minimum of $a_{i_2} - a_{i_1}, a_{i_3} - a_{i_2}, \ldots, a_{i_n} - a_{i_{n-1}}$.

Correctness: The minimum difference $|a_i - a_j|$ is achieved when a_i and a_j are consecutive in the sorted version of A. **1**

Running time: Merge sorting an array of size n requires $O(n \log n)$ time. Computing the minimum of $a_{i_j} - a_{i_{j-1}}$ over $j = 2, 3, \ldots, n$ takes $O(n)$ time. **1**

Element uniqueness: Determine whether an array of n integers contains duplicates.

It can be proved (using techniques other than reduction) that *element uniqueness* has a lower bound of $\Omega(n \log n)$ (under reasonable models of computation). Using this result, prove that the algorithm of Part (b) is optimal. **(5)**

Solution We reduce *element uniqueness* to *minimum difference* as follows.

Let A be the input array for *element uniqueness*.
Pass A itself to a *minimum difference* algorithm.
If the *minimum difference* algorithm returns 0, return "elements are not unique",
else return "elements are unique".

So *element uniqueness* $\leqslant$ *minimum difference*. Since *element uniqueness* has a lower bound of $\Omega(n \log n)$ and the above reduction algorithm runs in $\mathrm{O}(n)$ (that is, $\mathrm{o}(n \log n)$) time, it follows that any algorithm for *minimum difference* must run in $\Omega(n \log n)$ time (in the worst case).

3. We often need to compute the convex hull (smallest enclosing convex polygon) of general geometric objects.

(a) Design an $\mathrm{O}(n \log n)$-time algorithm to compute the convex hull of n triangles in the plane. **(5)**

Solution **The algorithm:** Let P_i, Q_i, R_i be the vertices of the i-th triangle. Compute the convex hull of the $3n$ points P_i, Q_i, R_i, $i = 1, 2, \ldots, n$. Output this convex hull. **2**

Correctness: Since a triangle is a convex polygon, it is immediate that a convex region encloses a triangle if and only if it encloses the three vertices of the triangle. **2**

Running time: Use an $\mathrm{O}(n \log n)$-time algorithm (like sorting followed by Graham's scan or Preparata and Hong's divide-and-conquer algorithm) for the computation of the convex hull. Here, we have $3n$ points. So the running time is $\mathrm{O}(3n \log(3n))$ which is again $\mathrm{O}(n \log n)$. **1**

non-convex) in the plane. **(5)**

Solution **The algorithm:** Let P_i, Q_i, R_i, S_i be the vertices of the i-th quadrilateral. Compute the convex hull of the $4n$ points P_i, Q_i, R_i, S_i, $i = 1, 2, \ldots, n$. Output this convex hull. **2**

Correctness: Any simple quadrilateral can be triangulated by two triangles. For example, let $PQRS$ be a quadrilateral. Since the sum of the internal angles of any simple quadrilateral is 360^0, a quadrilateral cannot have two or more internal angles $> 180^0$. If $PQRS$ contains such an angle, we rename the vertices (if necessary) and assume that the internal angle at P is $> 180^0$. But then, the triangles PQR and PRS constitute a triangulation of $PQRS$. **2**

Running time: Use an $O(n \log n)$-time algorithm (like sorting followed by Graham's scan or Preparata and Hong's divide-and-conquer algorithm) for the computation of the convex hull. Here, we have $4n$ points. So the running time is $O(4n \log(4n))$ which is again $O(n \log n)$. **1**

(c) What is the smallest convex polygon enclosing a circle? **(5)**

Solution No such polygon exists. For any polygon enclosing a circle, we can find a smaller polygon (with more edges) that encloses the circle.

substring of S and T. Design an $O(mn)$-time dynamic programming algorithm for solving this problem. **(15)**
(**Hint:** Consider the longest common suffix (or its length) $E_{i,j}$ of $S[0 \ldots i]$ and $T[0 \ldots j]$.)
(**Remark:** This problem can be solved in $O(m+n)$ time by using sophisticated data structures like *generalized suffix trees*.)

Solution **The algorithm:** We use an auxiliary two-dimensional array E of size $m \times n$. The variable $maxlen$ stores the maximum common substring found so far, whereas the variable $endpos$ stores the index of the last character of this common substring in the string S. **7**

Initialize $maxlen = 0$.

/* Initialize the first column */
for $i = 0, 1, \ldots, m-1$
    if ($A[i]$ equals $B[0]$)
        set $E[i][0] = 1$,
        $maxlen = 1$, and
        $endpos = i$.
    else set $E[i][0] = 0$.

/* Initialize the first row */
for $j = 1, 2, \ldots, n-1$
    if ($A[0]$ equals $B[j]$)
        set $E[0][j] = 1$,
        $endpos = 0$, and
        $maxlen = 1$.
    else set $E[0][j] = 0$.

/* Update the remaining $E[i][j]$ values in the row-major order */
for $i = 1, 2, \ldots, m-1$
    for $j = 1, 2, \ldots, n-1$
        if ($A[i]$ equals $B[j]$) set $E[i][j] = E[i-1][j-1] + 1$, else set $E[i][j] = 0$.
        if ($E[i][j] > maxlen$)
            set $maxlen = E[i][j]$.
            set $endpos = i$.

/* Return the longest common substring */
return $S[endpos - maxlen + 1 \ldots endpos]$.

Correctness: The length $E_{i,j}$ of the longest common suffix of $S[0 \ldots i]$ and $T[0 \ldots j]$ satisfies the recursive definition

$$E_{i,j} = \begin{cases} E_{i-1,j-1} + 1 & \text{if } S[i] = T[j] \\ 0 & \text{otherwise} \end{cases}$$

as long as $i \geqslant 1$ and $j \geqslant 1$. The boundary conditions are

$$E_{i,0} = \begin{cases} 1 & \text{if } S[i] = T[0] \\ 0 & \text{otherwise,} \end{cases} \quad \text{and} \quad E_{0,j} = \begin{cases} 1 & \text{if } S[0] = T[j] \\ 0 & \text{otherwise.} \end{cases}$$

The order, in which the values $E_{i,j}$ are computed above, ensures that the value of $E_{i-1,j-1}$ is already available during the computation of $E_{i,j}$ for $i \geqslant 1$ and $j \geqslant 1$. **6**

Running time: Initialization of the first column requires $\Theta(m)$ time. Initialization of the first row requires $\Theta(n)$ time. The subsequent doubly nested loop runs $(m-1)(n-1)$ times with each iteration taking $\Theta(1)$ time. The total running time is, therefore, $\Theta(mn)$. **2**