Assignment1

江书洋 (519030910043)

2021 年 10 月 16 日

Problem 1. Solve the following recurrence relations and give a Θ bound for each of them.

1. $T(n) = 5T(n/4) + n$
2. $T(n) = 7T(n/7) + n$
3. $T(n) = 49T(n/25) + n^{3/2}\log n$
4. $T(n) = 3T(n-1) + 2$

Solution. I present all time complexities with master theorem.

$$T(n) = a(\frac{T}{b}) + O(n^d)$$

$$T(n) = \begin{cases} O(n^d) & a < b^d \\ O(n^{\log_b a}) & a > b^d \\ O(n^d \log n) & a = b^d \end{cases}$$

1. $a = 5, b = 4, d = 1, T(n) = n^{log_4 5}$
2. $a = 7, b = 7, d = 1, T(n) = n\log n$
3. $a = 49, b = 25, d = 1.5, T(n) = n^{3/2}\log n$

$$T(n) = O(n^{3/2}\log n)(1 + \frac{49}{log25 \times 25^{3/2}} + \cdots + (\frac{49}{log25 \times 25^{3/2}})^{\log_{25} 49})$$

$$T(n) = n^{3/2}\log n$$

4. $T(n) = O(3^n)$ because $T(n) + 1 = 3(T(n-1)+1) \longrightarrow T(n) = O(3^n - 1) = O(3^n)$

Problem 2. Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements. Design an efficient algorithm using divide-and-conquer (and give the time complexity).

Solution. Divide the k arrays into two parts, one part contains $\lfloor k/2 \rfloor$ arrays and the other part contains $k - \lfloor k/2 \rfloor$. First use the same algorithm to process the first part and render a sorted array with $n\lfloor \frac{k}{2} \rfloor$ elements and process the another part and then render a sorted array with $nk - n\lfloor \frac{k}{2} \rfloor$ elements as well.

After obtaining the two arrays denoted as A and B, merge the two sorted array with the intuition in merge sort. In other words, use two pointers i and j and point each of them to the first element of those two sorted array, respectively. Set k to the first element of array to be filled.

- repeat
  - append $\min\{A[i], B[j]\}$
  - if $A[i]$ is smaller, then move i to i + 1; other wise move j to j + 1
  - break if $i > n\lfloor \frac{k}{2} \rfloor$ or $j > nk - \lfloor \frac{k}{2} \rfloor$
- append the remainder of the non-empty list to the answer list

After recursively call the algorithm for almost $\log k$ times and merge each two arrays, we can obtain the final sorted array with nk elements. The recurrence relation is $T(nk) = 2T(nk/2) + O(nk)$. Because n here is a constant, so the simplified relation is $T(k) = 2T(k/2) + O(k)$. From master theorem, its time complexity is $O(k \log k)$. So combined with n, the final time complexity is $O(nk \log k)$

Problem 3. You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values-so there are 2n values total-and you may assume that no two values are the same. You' d like to determine the median of this set of 2n values, which we will define here to be the n-th smallest45g value. However, the only way you can access these values is through queries to the databases. In a single query, you can specify a value k to one of the two databases, and the chosen database will return the k-th smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible. Give an algorithm that find the median value using O(log n) queries.

Solution.

Algorithm 1 median-search(A, B, al, ar, bl, br)

Input: input two databases A, B, and the number of elements in each dataset n and four pointers of each array al, ar, bl, br
Output: Median number in two databases

```
1: if al=ar and bl=br then
2:     atemp = search $amid^{th}$ element in dataset A
3:     btemp = search $bmid^{th}$ element in dataset B
4:     return $\frac{atemp+btemp}{2}$
5: end if
6: $amid = \frac{al+ar}{2}$
7: $bmid = \frac{bl+br}{2}$
8: atemp = search $amid^{th}$ element in dataset A
9: btemp = search $bmid^{th}$ element in dataset B
10: if atemp < btemp then
11:     return median-search(A, B, amid + 1, ar, bl, bmid - 1)
12: else
13:     return midian-search(A, B, al, amid - 1, bmid + 1, br)
14: end if
```

Problem 4. Show that the QuickSort algorithm runs in $O(n^c)$ time on average for some constant $c < 2$ if the pivot is chosen randomly.

Solution. Suppose the randomly selected pivot has 1/2 possibility to be chosen in the middle of the array. Suppose we need $\tau(n)$ to divide the problem to two sub problems: $T(n) = \tau(n) + T(\frac{1}{4}n) + T(\frac{3}{4}n)$. Take expectation on each side, which renders

$$E[T(n)] = E[\tau(n)] + E[T(\frac{1}{4}n)] + E[T(\frac{3}{4}n)]$$

Because we need at most two expected times to reduce the original problem to two sub problems with $O(n)$, so the final recurrence relation is

$$T'(n) = T'(\frac{1}{4}n) + T'(\frac{3}{4}n) + O(n)$$

Because we know that the lower bound of sorting algorithm is $\Theta(n \log n)$, which has the higher order than $O(n)$, so

$$T'(\frac{1}{4}n) + T'(\frac{3}{4}n) \leq \frac{4}{3}T'(\frac{3}{4})$$

The final result is that

$$T'(n) \leq \frac{4}{3}T'(\frac{3}{4}n) + O(n)$$

With master theorem, the time complexity is $O(n \log n)$, whose order is less than 2 in $O(n^c)$

PROBLEM 5. Given an $n \times m$ 2-dimensional integer array $A[0, \cdots, n-1; 0, \cdots, m-1]$ where $A[i, j]$ denotes the cell at the i-th row and the j-th column, a local minimum is a cell $A[i, j]$ such that $A[i, j]$ is smaller than each of its four adjacent cells $A[i-1, j]$, $A[i+1, j]$, $A[i, j-1]$, $A[i, j+1]$. Notice that $A[i, j]$ only has three adjacent cells if it is on the boundary, and it only has two adjacent cells if it is at the corner. Assume all the cells have distinct values. Your objective is to find one local minimum (i.e., you do not have to find all of them).

1. Suppose $m = 1$ so A is a 1-dimensional array. Design a divide-and-conquer-based algorithm for the problem above. Write a recurrence relation of the algorithm, and analyze its running time.
2. Suppose $m = n$. Design a divide-and-conquer-based algorithm for the problem above. Write a recurrence relation of the algorithm, and analyze its running time.
3. Generalize your algorithm such that it works for general m and n. The running time of your algorithm should smoothly interpolate between the running times for the first two parts.

SOLUTION.

1. The pseudo code is shown below:

Algorithm 2 find-local-minimum1

```
Input: input one array $A$, its begin index $l$ and its end index $r$
Output: local minimum in the array
1: if l + 1= r then
2:     return A[l] if A[l] < A[r] else A[r]
3: end if
4: $m = \frac{r+l}{2}$
5: if $A[m] < A[m+1]$ then
6:     return find-local-minimum1$(A, l, m)$
7: else
8:     if $A[m] > A[m+1]$ then
9:         return find-local-minimum1$(A, m, r)$
```

To get the final answer, the initial input is find-local-minimum1(A, 0, n-1). Recurrence relation is $T(n) = T(n/2) + O(1)$. Each recurrence reduces half of the problem size so the final complexity is $O(\log n)$.

2. In fact, whether m equals to n or not does not affect the algorithm. So the postcode below applies to both situations.

Algorithm 3 find-local-minimum2

```
Input: input one 2-d array $A$, its upper left row index $uli$, its upper left column index $ulj$, its lower right row index $lri$ and its lower right column index $lrj$
Output: local minimum in the matrix
 1: $m_i = \frac{uli+lri}{2}$
 2: $m_j = \frac{ulj+lrj}{2}$
 3: find the minimum value in $A[m_i, :]$ and $A[:, m_j]$ whose index is $(k_i, k_j)$
 4: if $A[k_i][k_j]$ is the local minimum in $A$ then
 5:     return $A[k_i][k_j]$
 6: else
 7:     look around the point and find a point that is less than $A[k_i][k_j]$
 8:     shrink the matrix size toward the point which is less than $A[k_i][k_j]$
 9:     denotes the next four indices as $n\_uli, n\_ulj, n\_lri, n\_lrj$
10:     return find-local-minimum2($A, n\_uli, n\_ulj, n\_lri, n\_lrj$)
```

To get the final answer, input find-local-minimum2(A, 0, 0, n-1, n-1). Because each time we guarantee that the point $A[k_i][k_j]$ is the least one in the small square, following the direction in which the value of number is decreasing can ensure a local minimum. Recurrence relation is $T(n^2) = T(n^2/4) + O(n)$, so the final time complexity is $O(n)$.

3. The algorithm is illustrated in problem 2 where I used a more general way to solve the problem. Although that method may not be the optimal algorithm for problem 2, it can still apply to this problem because I don't assume $m = n$ in the algorithm. So the algorithm pseudo code can refer to 3.

   The recurrence relationship is $T(mn) = T(mn/4) + O(m + n)$ and following the equation below:

$$T(nm) = n + m + \frac{n+m}{2} + \cdots + 1 = 2(n+m)$$

   So if $n > m$ the final relation is that

$$\log n < 2(n+m) < 4n$$

   where $4n$ is derived from $n = m$ in problem 2.