Homework #1

AI2615: Algorithm Design and Analysis

$*$ If there is any problem, please contact TA Jinyi Wang.

1. (a) $T(n) = 5T(n/4) + n.$
   (b) $T(n) = 7T(n/7) + n$
   (c) $T(n) = 49T(n/25) + n^{3/2}logn.$
   (d) $T(n) = 3T(n-1) + 2$

Solution. Refered from **Haoxuan Xu**

(a) $a = 5, b = 4, d = 1$. Hence, $a > b^d$, by master theorem, we can get $T(n) = \Theta(n^{log_4 5})$

(b) $a = 7, b = 7, d = 1$. Hence, $a = b^d$, by master theorem, we can get $T(n) = \Theta(nlogn)$

(c) $T(n) = O(n^{3/2}logn) \cdot (1 + 49 \cdot (\frac{1}{25})^{3/2} + \dots 49^k(\frac{1}{25^k})^{3/2} + \dots 49^{log_{25}n} \cdot (\frac{1}{25^{log_{25}n}})^{3/2})$
$= O(n^{3/2}logn) \cdot (1 + (49/125) + \dots (49/125)^k + \dots (49/125)^{(log_{25}n)})$.
Hence, $T(n) = \Theta(n^{3/2}logn)$

(d) $T(n) + 1 = 3(T(n-1) + 1)$, from which we can get $T(n) + 1 = 3^n(T(0) + 1) = \Theta(3^n)$.
Hence, $T(n) = \Theta(3^n)$.

$\square$

2. **A k-way merge operation.** 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. Referred from **Junqi Xie**.
We can recursively divide the k sorted arrays into 2 parts, and merge the 2 resulting arrays into a single one, just as the merging process in merge sort. In the following pseudo code, Merge is a function for merging 2 sorted arrays, just as in merge sort.

```
1: function MERGE_LIST($L$)
2:     $len \leftarrow$ LEN($L$)
3:     if $len \leq 1$ then
4:         return $L$
5:     else
6:         $l \leftarrow$ MERGE_LIST($L[: len/2]$)
7:         $r \leftarrow$ MERGE_LIST($L[len/2 + 1 :]$)
8:         return MERGE($l, r$)
9:     end if
10: end function
```

Since $T(k) = 2T(k/2) + O(n)$, the overall time complexity

$$T(k, n) = \Theta(nk \log k)$$

□

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 smallest value. However, the only way you can access these values is through queries to the databases. In a singlequery, you can specify a value k to one of the two databases, and the chosen databasewill 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 algorithmthat find the median value using $O(logn)$ queries.

Solution. Referred from **Haoxuan Xu**
We can consider two databases as two sorted arrays a_1, a_2(index starts from 0), and through each query we can input k and get $a_1[k-1]/a_2[k-1]$.

Algorithm 1: Median value search algorithm

```
Input: 2 sorted arrays $a_1, a_2$, each with $n$ elements.
 1: Initialization: $l_1, l_2 \leftarrow 0, r_1, r_2 \leftarrow n-1$.
 2: repeat
 3:    $m_1 \leftarrow (l_1 + r_1)/2, m_2 \leftarrow (l_2 + r_2)/2, evenFlag \leftarrow ((r_1 - l_1)\%2 = 0)$.
 4:    Inputing $m_1 + 1, m_2 + 1$, get $a_1[m_1], a_2[m_2]$ through two queries.
 5:    if $a_1[m_1] = a_2[m_2]$ then
 6:       return $a_1[m_1]$.
 7:    else if $a_1[m_1] > a_2[m_2]$ then
 8:       $r_1 \leftarrow m_1, l_2 \leftarrow m_2 + evenFlag$.
 9:    else
10:       $l_1 \leftarrow m_1 + evenFlag, r_2 \leftarrow m_2$.
11:    end if
12: until $l_1 = r_1$.
13: $m \leftarrow \min(a_1[l_1], a_2[l_2])$.
14: return $m$.
Output: The median $m$.
```

Correctness. If the median of the two arrays is equal, then it is also clearly the median of the two arrays and can be returned directly. If not, each recursion will sieve the smaller quarter and the larger quarter of the input range, narrowing the possible range to the middle half, and finally it will definitely come to the base case, where both arrays have only one number left, and the smaller one is the median value of the two arrays.

Complexity Since $T(n) = T(n/2) + O(1)$, the overall time complexity

$$T(n) = O(\log n)$$

□

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.

   Remark: The algorithm actually runs in $O(nlogn)$ time on average, but for this question it suffices to give an analysis better than $O(n^2)$.

   Proof. Referred from **Yuhao Zhang**
   In each round $w.p.\frac{1}{3}$, the pivot is greater than 1/3 and less than 2/3 of the n numbers (good case). In this case,it takes $O(n)$ time to process , and both the left and right sides have at most $\frac{2n}{3}$ numbers. It indicates $T(n) \leq 2T(\frac{2n}{3}) + O(n)$.
   And $w.p.\frac{2}{3}$, the pivot is less than 1/3 or greater than 2/3 of the n numbers (bad case). In this case, it takes $O(n)$ time to process, and. It takes at most $T(n)$ time in the following steps. So it means that:

$$
\begin{aligned}
E[T(n)] &\leq \frac{1}{3}(2T(\frac{2n}{3}) + O(n)) + \frac{2}{3}(T(n) + O(n)) \\
E[T(n)] &\leq \frac{1}{3}(2E[T(\frac{2n}{3})] + O(n)) + \frac{2}{3}(E[T(n)] + O(n)) \\
\frac{1}{3}E[T(n)] &\leq \frac{2}{3}E[T(\frac{2n}{3})] + O(n) \\
E[T(n)] &\leq 2E[T(\frac{2n}{3})] + O(n) \ .
\end{aligned}
$$

   Then by Master theorem, $a = 2, b = \frac{3}{2}, d = 1, d < \log_b a < 1.71, E[T(n)] = O(n^{1.71})$. So we find a constant $c = 1.71 < 2$.
   Note: You can also consider $(2/5, 3/5)$ (or others) as a partition

   □

5. Given an nm 2-dimensional integer array $A[0,\dots n-1;0,\dots m-1]$ 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 haveto find all of them).

   (a) 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.

   (b) 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.

   (c) Generalize your algorithm such that it works for generalmandn. The running time of your algorithm should smoothly interpolate between the running times for the first two parts.

Solution. Referred from **Biaoshuai Tao**
(a) Consider the altitudes at $A[n/2]$ and $A[n/2+1]$. It is not hard to observe that if $A[n/2] > A[n/2+1]$, then there exists a water pool in $A[n/2+1,\dots,n]$; if $A[n/2] < A[n/2+1]$, then there exists a water pool in $A[1,\dots,n/2]$. Therefore, by looking for the altitudes at these two positions, we can reduce our searching space by half. Similarly, if we recursively look for the two middle positions, we can always reduce the searching space by half. This gives us the recursive Algorithm 1, where FindPool(Start, End) finds a pool between position "Start" and "End", and computing FindPool$(1,n)$ gives us the result.

Algorithm 2: FindPool(Start, End)

```
1: If End=Start, output Start or End and terminate
2: $m = \lceil 0.5(\text{Start} + \text{End}) \rceil$
3: If $A[m] > A[m+1]$, FindPool($m+1$, End)
4: If $A[m] < A[m+1]$, FindPool(Start, $m$)
```

It is straightforward that the recurrence relation of time complexity is

$$T(n) = T(n/2) + c,$$

and master theorem gives us $T(n) = O(\log n)$.

(b) We describe the first few steps to illustrate the algorithm. Consider the $(n/2)$-th column $A[n/2,0], A[n/2,1],\dots,A[n/2,n-1]$. We first find the position in this column with the minimum altitude, say, at $A[n/2,3]$. If the left and right adjacent positions $A[n/2-1,3]$ and $A[n/2+1,3]$ both have altitudes larger than $A[n/2,3]$, then we have already found a pool $A[n/2,3]$. If at least one of the two adjacent positions has smaller altitude, say, $A[n/2-1,3]$, then we know there must exist a pool on the left half of the map. This is because the $(n/2)$-th column in this case serves as a "wall", for which the flow $A[n/2,3] \to A[n/2-1,3]$ can never pass through. Therefore, we have reduce the searching space by half of size.

In the second step, we are going to find the minimum altitude in the $(n/2)$-th row on the left half of the map, which is the one in $A[0,n/2], A[1,n/2],\dots,A[n/2,n/2]$. Say, the minimum is $A[5,n/2]$. Again, if both the upper and lower positions of $A[5,n/2]$ have larger altitudes, we have already found a pool. If not, we need to compare the altitude of this minimum $A[5,n/2]$ to the previous one we found, namely $A[n/2,3]$. If the altitude at $A[5,n/2]$ is larger, then the

$(n/2)$-th row serves as a wall to stop the flow from $A[n/2, 3]$ from passing through, and we know there is a pool at the quarter of the map $A[0, 1, \ldots, n/2; 0, 1, \ldots, n/2]$. If the altitude at $A[n/2, 3]$ is larger, then the $(n/2)$-th column serves as a wall instead, and the flow from $A[5, n/2]$ cannot passing through the $(n/2)$-th column. We can then decide which of the upper and lower quarters of the left half map to be kept in the next step recursion by exam which of $A[5, n/2 + 1]$ and $A[5, n/2 - 1]$ has a altitude larger than $A[5, n/2]$. (If both are larger, we can choose either quarter.) In all the situation, the searching space is further reduced by half.

We can continue this process of "cutting" the searching space by half each time, and the "cuts" are made vertically and horizontally in alternation. Notice that we need to record the position of the current lowest position on the "wall" (as the start of flow), and its altitude, such that whenever a new "cut" is made, we keep the position with the smaller altitude as the start of flow. The process stops when we find a pool, either in the case when the lowest position on the new "wall" is itself a pool, or in the case when there is only one position left in the searching space.

By our construction, we can always be sure that there exists a pool in the searching space which is reduced by half at each step. So there is no doubt that we can finally find a pool. As for the time complexity, since in each step we need to search for a minimum at a new "cut", whose size is reduced by half in every two steps, we have

$$\begin{aligned}
T(n) &= n + T(n/2) && \text{(The first vertical ``cut'')}\\
&= (n + n/2) + T(n/4) && \text{(The second horizontal ``cut'')}\\
&= (n + n/2 + n/2) + T(n/8) && \text{(The third vertical ``cut'')}\\
&= (n + n/2 + n/2 + n/4) + T(n/16) && \text{(The forth horizontal ``cut'')}\\
&\vdots\\
&= n + n/2 + n/2 + n/4 + n/4 + n/8 + n/8 + \cdots + 1\\
&= O(n).
\end{aligned}$$

(c) For $m \neq n$, we apply the similar technique in (b) by cutting the searching space by half at each step. There are three ways to cut: 1) always cut vertically; 2) always cut horizontally; 3) cut alternatively (like in (b)). By a similar analysis, the time complexity for these three cutting scheme are respectively

$$T_{\text{vertical}} = O(m \log n) \qquad T_{\text{horizontal}} = O(n \log m) \qquad T_{\text{alternative}} = O(m + n).$$

Given value m and n, we can decide which of $m \log n$, $n \log m$ and $m + n$ is smaller, and then perform the corresponding scheme. Thus, the algorithm first makes the decision by comparison, and then perform the cutting scheme based on the decision. Let $f(n) = \min(m \log n, n \log m, m + n)$, we have the time complexity here is $O(f(n))$. We can see that the running times here smoothly interpolate between the running times of (a) and (b), as taking $m = 1$ we have $O(f(n)) = O(\log n)$ as it is in (a), and taking $m = n$, we have $O(f(n)) = O(m + n) = O(n)$ as it is in (b).

$\square$