CSE 431/531: Algorithm Analysis and Design	Spring 2022

Homework 3

Instructor: Shi Li	**Deadline: 4/4/2022**

Your Name: ____________________ Your Student ID: ______________

Problems	1	2	3	4	Total
Max. Score	16	24	25	25	80
Your Score					

The total score for the 4 problems is 90, but your score will be truncated at 80.

Problem 1 For each of the following recurrences, use the master theorem to give the tight asymptotic upper bound. You just need to give the final bound for each recurrence.

(a) $T(n) = 4T(n/3) + O(n)$. $T(n) = O(\underline{n^{\log_3 4}})$.

(b) $T(n) = 3T(n/3) + O(n^2)$. $T(n) = O(\underline{n^2})$.

(c) $T(n) = 4T(n/2) + O(n^2\sqrt{n})$. $T(n) = O(\underline{n^2\sqrt{n}})$.

(d) $T(n) = 8T(n/2) + O(n^3)$. $T(n) = O(\underline{n^3 \log n})$.

Problem 2 We consider the following problem of counting stronger inversions. Given an array A of n positive integers, a pair $i, j \in \{1, 2, 3, \cdots, n\}$ of indices is called a strong inversion if $i < j$ and $A[i] > 2A[j]$. The goal of the problem is to count the number of strong inversions for a given array A. Give a divide-and-conquer algorithm that runs in $O(n \lg n)$ time to solve the problem.

We shall modify the divide-and-conquer algorithm for counting inversions slightly. The only thing that needs to be changed is the procedure merge-and-count(B, C, n_1, n_2). As in the algorithm for counting inversions, we are given two sorted arrays: B of length n_1, and C of length n_2. But now we need to count the number of *strong* inversions between B and C, and merge B and C. The two tasks are performed using two different while loops: in the first while loop, we count the number of strong inversions between B and C, that is, the number of pairs (i, j) such that $B[i] > 2 \times C[j]$. In the second while loop, we merge B and C into a sorted array. The procedure is given by the following pseudo-code:

Algorithm 1 merge-and-count(B, C, n_1, n_2)

$\mathit{count} \leftarrow 0$, $A \leftarrow$ array of $n_1 + n_2$ 0's
$i \leftarrow 1$; $j \leftarrow 1$
while $i \le n_1$ or $j \le n_2$ **do**
  if $j > n_2$ or ($i \le n_1$ and $B[i] \le 2 \times C[j]$) **then**
    $i \leftarrow i + 1$
    $\mathit{count} \leftarrow \mathit{count} + (j - 1)$
  else
    $j \leftarrow j + 1$
$i \leftarrow 1$; $j \leftarrow 1$
while $i \le n_1$ or $j \le n_2$ **do**
  if $j > n_2$ or ($i \le n_1$ and $B[i] \le C[j]$) **then**
    $A[i + j - 1] \leftarrow B[i]$; $i \leftarrow i + 1$
  else
    $A[i + j - 1] \leftarrow C[j]$; $j \leftarrow j + 1$
return (A, count)

The recurrence for the running time is still $T(n) = 2T(n/2) + O(n)$. So, the running time of the algorithm is $O(n \log n)$.

Problem 3 Given an array A of n **distinct** numbers, we say that some index $i \in \{1, 2, 3 \cdots, n\}$ is a local minimum of A, if $A[i] < A[i-1]$ and $A[i] < A[i+1]$ (we assume that $A[0] = A[n+1] = \infty$). Suppose the array A is already stored in memory. Give an $O(\lg n)$-time algorithm to find a local minimum of A.

In the main algorithm, we call local-minimum$(1, n)$. The procedure local-minimum is defined as follows.

Algorithm 2 local-minimum(l, r)

if $l = r$ **then return** l
$m = \lfloor (l + r)/2 \rfloor$
if $A[m] < A[m + 1]$ **then**
  return local-minimum(l, m)
else
  return local-minimum$(m + 1, r)$

local-minimum(l, r) will return a local minimum in the sub-array $A[l..r]$. We guarantee the following property:
(*) When we call local-minimum(l, r), we have $A[l-1] > A[l]$ and $A[r] < A[r+1]$.

(*) is true when $l = 1$ and $r = n$. If $l < r$, we break $A[l..r]$ into two arrays $A[l..m]$ and $A[m+1..r]$. If $A[m] < A[m+1]$ then we call local-minimum(l, m) and we are guaranteed that $A[l-1] > A[l]$ and $A[m] < A[m+1]$; if $A[m] > A[m+1]$ then we call local-minimum(l, m) and we are guaranteed that $A[m] > A[m+1]$ and $A[r] < A[r+1]$. Thus, we always maintain (*).

Thus, when $l = r$, we have $A[l-1] > A[l]$ and $A[l] < A[l+1]$; so $A[l]$ is a local minimum. The running time of the algorithm is $O(\lg n)$.

Problem 4 Consider a $2^n \times 2^n$ chessboard with one arbitrary chosen square removed. Prove that any such chessboard can be tiled without gaps by L-shaped pieces, each composed of 3 squares. Figure 1 shows how to tile a 4×4 chessboard with the square on the left-top corner removed, using 5 L-shaped pieces. Use divide-and-conquer to solve the problem.

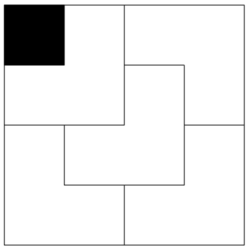

Figure 1: Using 5 tiles to cover a chessboard of size 4×4, with the left-corner missing.

We give a procedure that covers the board using L-shaped tiles. Divide the $2^n \times 2^n$ board into 4 sub-boards of size $2^{n-1} \times 2^{n-1}$. Consider the 4 squares in the center of board, one from each sub-board (see Figure 2). We use an L-shaped piece to cover 3 of the 4 squares. The only square that is not covered is in the sub-board that contains the removed square. We then remove the 3 squares from the board. Then we can divide the board into 4 sub-boards, each containing exactly one removed square. The 4-sub problems can be solved recursively. The recursion stops when we have a 2×2 board with one square removed, which clearly can be covered by a L-shaped piece.

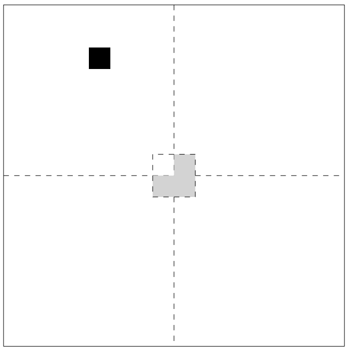

Figure 2: Choose a center L-shape and reduce to 4 sub-problems