CSE 431/531: Algorithm Analysis and Design — Fall 2021

Homework 3

Instructor: Shi Li — **Deadline: 11/4/2021**

Your Name: ____________________ Your Student ID: ______________

Problems	1	2	3	Total
Max. Score	20	30	30	80
Your Score				

Problem 1. For each of the following recurrences, use the master theorem to give the tight asymptotic upper bound.

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

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

(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 $O(n \log n)$ time to solve the problem. Write down the recurrence for the running time, and use the master theorem to show that the running time is indeed $O(n \log n)$.

We 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)

```
count ← 0, A ← []
i ← 1; j ← 1
while i ≤ n1 or j ≤ n2 do
    if j > n2 or (i ≤ n1 and B[i] ≤ 2 × C[j]) then
        i ← i + 1
        count ← count + (j − 1)
    else
        j ← j + 1
i ← 1; j ← 1
while i ≤ n1 or j ≤ n2 do
    if j > n2 or (i ≤ n1 and B[i] ≤ C[j]) then
        append B[i] to A; i ← i + 1
    else
        append C[j] to A; j ← 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 two sorted arrays A and B with total size n, and a positive integer $k \le n$, you need to design an $O(\log n)$-time algorithm that outputs the k-th smallest number in the union of A and B. You need to prove that the running time of your algorithm is indeed $O(\log n)$.

For example, if $A = [3, 5, 12, 18, 50], B = [2, 7, 11, 30]$, and $k = 4$ then you need to output 7 since the union of A and B is $[2, 3, 5, 7, 11, 12, 18, 30, 50]$ after sorting.

In the algorithm, we maintain four indices lA, rA, lB, rB and an integer k' between 1 and $rA - lA + 2 + rB - lB$. We guarantee that our goal is to output the k'-th smallest number in $A[lA..rA] \uplus B[lB..rB]$. The pseudo-code is given in Algorithm 2.

We explain Step 6 and Step 8 of the algorithm.

- As $A[mA] \le B[mB]$ we know elements in $A[lA..mA] \uplus B[lB..mB]$ are smaller than or equal to elements in $B[mB + 1..rB]$. $A[lA..mA] \uplus B[lB..mB]$ contains $mA - lA + 2 + mB - lB$ elements. So, when $k' \le mA - lA + 2 + mB - lB$, any element in $B[mB + 1..rB]$ is too big to be the k'-th smallest element. So, Step 6 reduced the problem correctly.
- Equivalently, our goal is to find the $(rA - lA + 3 + rB - lB - k')$-th largest number in $A[lA..rA] \uplus B[lB..rB]$. If $k' > mA - lA + 2 + mB - lB$, then $rA - lA + 3 + rB - lB - k' \le rA - mA + rB - mB$. If $A[mA] \le B[mB]$, then elements in $A[mA + 1..rA] \uplus B[mB + 1..rB]$ are larger than or equal to elements in $A[lA..mA]$. So, any element in $A[lA..mA]$ is too small to be the k'-th smallest element. So, Step 8 reduced the problem correctly.

So, in each iteration of Loop 2, we either reduced the size of $A[lA..rA]$ by a factor of 2, or the size of $B[lB..rB]$ by a factor of 2. In at most $O(\log n)$ iterations, one of the two sub-arrays will become of size 1. The running time of the algorithm is $O(\log n)$.

Algorithm 2 kth-smallest-number(A, B, k)

$lA \leftarrow 1, \quad rA \leftarrow$ size of $A, \quad lB \leftarrow 1, \quad rB \leftarrow$ size of $B, \quad k' \leftarrow k$
while $rA > lA$ and $rB > lB$ **do**
    $mA \leftarrow \left\lfloor \frac{lA+rA}{2} \right\rfloor$ and $mB \leftarrow \left\lfloor \frac{lB+rB}{2} \right\rfloor$
    if $A[mA] \leq B[mB]$ **then**
        if $k \leq mA - lA + 2 + mB - rB$ **then**
            $rB \leftarrow mB$
        else $\triangleright$
            $lA \leftarrow mA + 1, k' \leftarrow k' - (mA - lA + 1)$
    else $\triangleright$ $A[mA] > B[mB]$
        handle this case in a symmetric way
if $lA = rA$ **then**
    return $\min\{A[lA], B[lB + k' - 1]\}$
else $\triangleright$ $lB = rB$
    return $\min\{B[lB], A[lA + k' - 1]\}$