CSE 431/531: Algorithm Analysis and Design — Fall 2022

Homework 3 Solutions

Instructor: Shi Li — **Deadline: 10/23/2022**

Your Name: ______________________ Your Student ID: ________________

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

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) $T(n) = 5T(n/3) + O(n)$. $T(n) = O(\underline{n^{\log_3 5}})$.

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

Problem 2 Given two n-digit integers, you need output their product. Design an $O(n^{\log_2 3})$-time algorithm for the problem, using the polynomial-multiplication algorithm as a black box to solve the problem.

Assume the two n-digit integers are given by two 0-indexed arrays A and B of length n, each entry being an integer between 0 and 9. The i-th integer in an array corresponds to the digit with weight 10^i. For example, if we need to multiple 3617140103 and 3106136492, then the two arrays are $A = (3, 0, 1, 0, 4, 1, 7, 1, 6, 3)$ and $B = (2, 9, 4, 6, 3, 1, 6, 0, 1, 3)$. That is, $A[0] = 3, A[1] = 0, A[2] = 1$, etc. You need to output the product "11235330870604938676". You can assume the two integers both have n digits, and there are no leading 0's.

You can use the polynomial-multiplication algorithm as a black-box; you do not need to give its code/pseudo-code.

Treat A and B as polynomials, and call the $O(n^{\log_2 3})$-time polynomial multiplication algorithm to obtain the product polynomial C of A and B.
  ▷ We extend C by one entry so that it has indices from 0 to $2n-1$. That is, the length of C is $2n$ and $C[2n-1] = 0$.

for $i \leftarrow 0$ to $2n-2$ **do**
  $C[i+1] \leftarrow C[i+1] + \lfloor C[i]/10 \rfloor$
  $C[i] \leftarrow C[i] \bmod 10$
for $i \leftarrow 2n-1$ downto 0 **do**
  if $i \neq 2n-1$ or $A[i] \neq 0$ **then** $print(A[i])$

Problem 3 Suppose you are given n pictures of human faces, numbered from 1 to n. There is a face comparison program A that, given two different indices i and j from $1, 2, \cdots, n$, returns whether face i and face j are the same, i.e., are of the same person. A majority face is a face that appears more than $n/2$ times in the n pictures.

The problem, then, is to decide whether there is a majority face or not, using the algorithm A as a black box. You need to design and analyze an algorithm that only calls A $O(n \log n)$ times.

Remark. A can only return whether two faces i and j are the same or not. If they are not the same, A can *not* tell you whether "face $i <$ face j" or "face $i >$ face j".

Example. Suppose $n = 5$ and the function calls to $\mathcal{A}$ and their returned values are as follows: $A(1, 2) =$ different, $A(1, 3) =$ different, $A(2, 3) =$ same, $A(3, 4) =$ different, $A(3, 5) =$ same. Then your algorithm can correctly return "yes" since it knows that faces 2, 3 and 5 are the same. *This example is only for the purpose of helping you understand the problem. You should not use it as a guide to design your algorithm.*

The crucial observation is the following: if some face is a majority face in a set S, and S is the disjoint union of two sets S_1 and S_2, then the face is either a majority face in S_1, or a majority face in S_2 (or both).

Algorithm 1 majority(ℓ, r) \\return the majority face in $\{\ell, \ell+1, \cdots, r\}$, or -1 if it does not exist

if $\ell = r$ **then return** ℓ
$m \leftarrow \lfloor (\ell + r)/2 \rfloor$
$a \leftarrow$ majority(ℓ, m)
$b \leftarrow$ majority$(m+1, r)$
if $a \neq -1$ **then**
    $count \leftarrow 0$
    for $i \leftarrow \ell$ to r **do**
        if $A(a, i) =$ "same" **then** $count \leftarrow count + 1$
    if $count > (r - \ell + 1)/2$ **then return** a
if $b \neq -1$ **then**
    $count \leftarrow 0$
    for $i \leftarrow \ell$ to r **do**
        if $A(b, i) =$ "same" **then** $count \leftarrow count + 1$
    if $count > (r - \ell + 1)/2$ **then return** b
return -1

The recurrence for the running time of the algorithm is $T(n) = 2T(n/2) + O(n)$ and thus the final running time is $O(n \log n)$.

Problem 4 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 $O(\log n)$-time algorithm to find a local minimum of A. (There could be multiple local minimums in A; you only need to output one of them.)

Algorithm 2 local-minimum(l, r)

```
if l = r then return l
m = ⌊(l + r)/2⌋
if A[m] < A[m + 1] then
    return local-minimum(l, m)
else
    return local-minimum(m + 1, r)
```

local-minimum(l, r) returns a local minimum in $A[l..r]$. Thus, local-minimum$(1, n)$ will return a local minimum in A. 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 $O(\log n)$ (the recurrence is $T(n) = T(n/2) + O(1)$.)