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

# Homework 3

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

Name: Amudheswaran Sankaranarayanan

UBID : 50454389

UB Name: amudhesw

**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) $T(n) = 5T(n/3) + O(n)$.

>> $T(n) = O(n^{\log_3 5})$

(b) $T(n) = 3T(n/3) + O(n)$.

>> $T(n) = O(n^1 \log n)$

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

>> $T(n) = O(n^{\frac{5}{2}})$

(d) $T(n) = 8T(n/2) + O(n^2)$.

>> $T(n) = O(n^3)$

**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.

>> This problem can be solved like polynomial multiplication with $x = 10$

$If\ number\ A\ =\ 3617140103$

$and$

$B\ =\ 3106136492$

$Their\ array\ representaions\ are:$

$$A\ =\ (3,0,1,0,4,1,7,1,6,3)$$

$$and$$

$$B\ =\ (2,9,4,6,3,1,6,0,1,3)$$

$We\ pass\ these\ values\ into\ our\ algorithm\ that\ inherits\ the\ polynomial\ multiplication\ algorithm$

```
2>
Algorithm : productUsingPolynomialMultiplication
Inputs : two arrays representing the two numbers where each array element corresponds to the digit with weight 10^i
Output : an integer value equivalent to the product of two numbers represented in the two arrays
algorithms being used : polynomial multiplication algorithm using divide&conquer technique , returns an array of coefficients

productUsingPolynomialMultiplication(A,B):

    C <- dcPolyMultiply(A,B)
    result <- 0

    for i in range(0, length(C) ), do:
        result <- result + ( pow(10, i) * C[i]  )
    end

    return result
```

Time complexity: Since it uses a for loop to construct the integer after the divide and conquer polynomial multiplication algorithm to produce the coefficients, the overall time complexity is

$$T(n) = 3T(n/2) + O(n) + O(n)$$

$$> T(n) = O(n^{\log_2 5})$$

**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.

```
Instructor's hint : The hint is the following. Suppose you have an array of integers and you break it into two subarrays. If an integer is a majority of the whole array, then it must be
the majority of at least one of the two subarrays.

Algorithm : findMajorityFace
Input : categorizedFacesList > An array of size "n" same as the one used in blackbox algorigthm A , but now has a unique tag associated to those faces that come under a common sub-group
        low > index from which the comparisons begins            [at inital call, this is the first index position of the input]
        high > the index till which the comparisons occur        [at inital call, this is the last  index position of the input]
        faceFreqs > a dictionary holding frequency of each face type
Output :dominantFace returns a dominant facetype

findMajorityFace( categorizedFacesList , low, high, faceFreqs , dominantFace)
    mid = low + (low+high)/2
    # base case #
    if low == high , do:
        dominantFace = categorizedFacesList[mid]
        faceFreqs[ dominantFace ] += 1

        return dominantFace
    end

    # splitting the domain in halves #
    LeftDominantFace <- findMajorityFace( categorizedFacesList , low, mid-1)
    RightDominantFace <- findMajorityFace( categorizedFacesList , mid,high)

    if faceFreqs[LeftDominantFace] > faceFreqs[RightDominantFace], do:
        dominantFace = LeftDominantFace
        return True
    else if faceFreqs[LeftDominantFace] < faceFreqs[RightDominantFace]
        dominantFace = RightDominantFace
        return True

    return False
```

Time complexity: nlogn

**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.)

```python
import math

def firstLocalMinimaFinder(localMinimaArray, low, high)-> int:
    A = localMinimaArray #[math.inf,1,2,5,12,8,3,6,math.inf]
    mid =low + (low+ high) //2
    midL = mid-1
    midR = mid+1
    if (A[mid] < A[midR] and A[mid] < A[midL])  :

        return mid
    else:
        if A[mid] >  A[midL]:
            firstLocalMinimaFinder(A , low , midL)
        else:
            firstLocalMinimaFinder(A , midR , high)

localMinimaArray = [math.inf,7, 5, 2, 11, 13, 37,math.inf]
low = 0
high = len(localMinimaArray)-1
print("Local Minima is at :",firstLocalMinimaFinder(localMinimaArray,low,high) )
```

✓ 0.5s Python

```
Local Minima is at : 3
```

Time complexity : O(logn) as it is divides the search by half the search-space in each recursive call