CS 218, Spring 2017 Homework 3
Abhishek Kumar SRIVASTAVA April 26, 2017
Student ID: 861307778

I certify that this submission represents my own original work

---

Solution 1. Considering the polynomials represented by following equations:

$$A(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \cdots + a_n \cdot x^n$$

$$B(x) = b_0 + b_1 \cdot x + b_2 \cdot x^2 + \cdots + b_n \cdot x^n$$

we can rewrite the equations as following:

$$A(x) = a + b \cdot x^{n/2}$$

$$B(x) = c + d \cdot x^{n/2}$$

a, b, c and d are equations of polynomials $n/2$.
After multiplying them we get,

$$C(x) = A(x) \cdot B(x) = (a + b \cdot x^{n/2}) \cdot (c + d \cdot x^{n/2})$$

$$C(x) = ac + (ad + cb) \cdot x^{n/2} + bd \cdot x^n$$

The above equation shows that we have 4 terms to multiply *i.e* ac, ad, cb & bd. But we can rewrite $(ad + cd)$ by $(a + b)(c + d) - ac - bd$

$$C(x) = ac + ((a + b)(c + d) - ac - bd) \cdot x^{n/2} + bd \cdot x^n$$

This equation only needs 3 multiplications: ac, bd & $(a + b)(c + d)$.

Therefore for each division we need 3 multiplication of $n/2$ polynomials and $O(n)$ to add them all, which will give the recurrence relation as

$$T(n) = 3 \cdot T(n/2) + c \cdot n$$

Now we can apply Master theorem on the above equation.

Case 1: $f(n) \; \epsilon \; O(n^{\log_b a - \varepsilon})$ where $\varepsilon > 0$.

$$n \epsilon O(n^{\log_2 3 - \varepsilon})$$

For any $0.9 > \varepsilon > 0$ this case holds valid. So case 1 pass.Thats why,

$$\boxed{T(n) = \Theta(n^{\log_2 3})}$$

Solution 2. Part a:

Given Matrix

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

We have to multiply matrix A with itself. Let $B = AA$.

$$B = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} a^2 + bc & ab + bd \\ ac + cd & bc + d^2 \end{bmatrix}$$

As $ab = ba$ where a and b are 1x1 matrices or scalar. We can rewrite the matrices as,

$$\begin{aligned} B &= \begin{bmatrix} a^2 + bc & ba + bd \\ ca + cd & bc + d^2 \end{bmatrix} \\ &= \begin{bmatrix} a^2 + bc & b(a+d) \\ c(a+d) & bc + d^2 \end{bmatrix} \end{aligned}$$

Therefore all the terms which could be used to calculate the product of 2x2 matrices are following 5 products

$$\begin{aligned} M_1 &= a^2 \\ M_2 &= bc \\ M_3 &= b(a+d) \\ M_4 &= c(a+d) \\ M_5 &= d^2 \end{aligned}$$

Part b:

As above from by multiplying two matrices we get,

$$B = \begin{bmatrix} a^2 + bc & ab + bd \\ ac + cd & bc + d^2 \end{bmatrix}$$

but since if a,b,c & d are $n/2$ x $n/2$ matrices and not the 1x1 matrices we cannot apply $ab = ba$ since matrix multiplication does not obey commutative property. It works on the above problem because they are scalar values and they obey commutative property. So,

$$\begin{bmatrix} a^2 + bc & ab + bd \\ ac + cd & bc + d^2 \end{bmatrix} \neq \begin{bmatrix} a^2 + bc & ba + bd \\ ca + cd & bc + d^2 \end{bmatrix}$$

And therefore matrix multiplication cannot be reduced to 5 submatrices multiplications.

Solution 3. Given an array A of length n we have to return i & j such that sum of the subarray is maximized using divide and conquer approach.

Create a function called **SubArrayMaxSum** which will take an *array A*, *start* and *endpoint* of the array as input and it will return corresponding i & j for that array which will give the maximum sum of array from i^{th} & j^{th} index $i.e$ Subarray.

Inside the **SubArrayMaxSum** function first check the base case if it is single element just return that index as both i & j. If it is an array with more than one element divide the array into two equal halves and do **SubArrayMaxSum** recursive call for both halves. After the both halves return their corresponding i & j then merge the subarrays and return optimal i & j which maximize the sum for the whole merged array.

Merge will be done by first calculating sum of left subarray and right subarray. Then use variables called $MaxSum$, $TempSum$, i_final & j_final. $MaxSum$ & $TempSum$ is assigned sum of left subarray and i_final & j_final values are set to i & j value returned by left subarray as well. Then we will calculate the sum from the end of the left subarray index $i.e$ j_left^{th} index till end of right subarray $i.e$ j_right^{th} index. Calculate the sum by adding one element at a time and checking some conditions. Maintain a temp i & j which keeps track of subarray with sum between $temp\_i^{th}$ & $temp\_j^{th}$ element $i.e$ $TempSum$. If sum goes below zero, reset $TempSum$ value to 0 & $temp\_i^{th}$ & $temp\_j^{th}$ to next index value and continue. If $TempSum$ value is greater than $MaxSum$ set $temp\_jth$ index value to this index and $TempSum$ to $MaxSum$. After the loop ends if $MaxSum$ is greater than sum of right subarray calculated before merge then set i_final & j_final as $temp\_i^{th}$ & $temp\_j^{th}$ value otherwise as i_left^{th} & j_left^{th}. Return i_final & j_final.

Since we are dividing an array of n into 2 $n/2$ halves calling them recursively and merging them again after recursion is over in $O(n)$ in worst case, Therefore the recurrence relation can be written as

$$T(n) = 2 \cdot T(n/2) + O(n)$$

Applying Master theorem on the above equation.

Case 1: $f(n)\ \epsilon\ O(n^{\log_b a - \varepsilon})$ where $\varepsilon > 0$.

$$n \epsilon O(n^{\log_2 2 - \varepsilon} = O(n^{1-\varepsilon})$$

For any $\varepsilon > 0$ this will not be true, So case 1 fails.

Case 2: $f(n)\ \epsilon\ \Theta(n^{\log_b a} \cdot \log^k n)$ where $k \geq 0$.

$n\ \epsilon\ \Theta(n^{\log_2 2} \cdot \log^k n) = \Theta(n \cdot \log^k n)$

$k = 0$ satisfy the equation. So case 2 holds. Thats why,

$$\boxed{T(n) = \Theta(n \cdot \log n)}$$

Which is required Time complexity for this problem.

Psuedocode for the above described method is following:

```
function SUBARRY_MAXSUM(A,start,end)
    if start == end then
        return start,end
    end if
    mid = (start + end)/2                              ▷ Calculate mid point of Array

    //Divide
    i_left, j_left = SUBARRY_MAXSUM(A,start,mid)                    ▷ Left Subarray
    i_right, j_right = SUBARRY_MAXSUM(A,mid+1,end)                 ▷ Right Subarray

    //Merge
    Left_Sum = sum_{k=i_left}^{j_left} A_k                      ▷ Sum of left subarray
    Right_Sum = sum_{k=i_right}^{j_right} A_k                  ▷ Sum of right subarray

    if Left_Sum ≠ 0 then
        MaxSum = TempSum = Left_Sum                                 ▷ Initialization
    else
        MaxSum = TempSum = 0                            ▷ To handle if sum is negative
    end if

    i_final = temp_i = i_left                                       ▷ Initialization
    j_final = temp_j = j_left                                       ▷ Initialization

    for k = i_left + 1 to j_right do        ▷ calculation of sum by merging both subarray
        TempSum = TempSum + A[k]
        if TempSum < 0 then                  ▷ If TempSum is negative ,reset subarray
            TempSum = 0
            temp_i = k + 1
            temp_j = k + 1
        else
            if TempSum > MaxSum then         ▷ If TempSum is larger, store parameters
                MaxSum = TempSum
                temp_j = k
            end if
        end if
    end for

    if MaxSum < Right_Sum then      ▷ If Right subarray is sum is larger without merge
        i_final = i_right
        j_final = j_right
    else
        if MaxSum > 0 then             ▷ To handle if merged subarray is negative sum
            i_final = temp_i
            j_final = temp_j
        end if
    end if

    return i_final,j_final                  ▷ returning final i and j index values
end function
```