HW2

Keerthana Golla

September 2023

1 Problem - 1

Algorithm 1 Finding the Dominant Element

```
function findDominantElement(nums, left, right)
    if left = right then
        return nums[left]
    end if
    mid ← (left + right)//2
    leftDominant ← findDominantElement(nums, left, mid)
    rightDominant ← findDominantElement(nums, mid + 1, right)
    if leftDominant = rightDominant then
        return leftDominant
    end if
    leftCount ← countOccurrences(nums, leftDominant)
    rightCount ← countOccurrences(nums, rightDominant)
    if leftCount > (right−left+1)/2 then
        return leftDominant
    else if rightCount > (right−left+1)/2 then
        return rightDominant
    else
        return "NULL"
    end if
end function

function findDominant(nums)
    return findDominantElement(nums, 0, length(nums) − 1)
end function

function countOccurrences(nums, target)
    count ← 0
    for num in nums do
        if num = target then
            count ← count + 1
        end if
    end for
    return count
end function
```

Above is the pseudo code for finding the dominant element.We will call findDominant(nums) function and it returns the dominant element if present, else $NULL$ (where nums is the input array).

Correctness :

Base Case: When the input sequence has only one element ($left == right$), the algorithm trivially returns that element, and it is indeed the dominant element (if it exists). So, the base case is correct.

Inductive Hypothesis: Assume that the algorithm correctly identifies the dominant element in subarrays of size less or equal to k, where $k > 1$.

Inductive Step: Now, let's consider the case of a subarray of size $k + 1$, where $k \geq 1$. The algorithm divides the subarray into two halves, $left$ and $right$, and recursively finds the dominant element in each half:

$$leftDominant = findDominantElement(nums, left, mid)$$

$$rightDominant = findDominantElement(nums, mid + 1, right)$$

The Induction hypothesis comes into play where we assumed that the algorithm correctly identifies the dominant element in these two subarrays, $leftDominant$ and $rightDominant$, respectively, as they are both of size less than or equal to k.

We then proceed to check the below two possibilities:

a. If $leftDominant$ and $rightDominant$ are the same, it directly returns that value as the dominant element. This is correct because if the same element is dominant in both halves, it will be dominant in the entire subarray.

b. If $leftDominant$ and $rightDominant$ are different, it counts the occurrences of each in the entire subarray and compares these counts with $(right-left+1)/2$, which is half the size of the subarray. If either $leftDominant$ or $rightDominant$ has more than half the occurrences in the entire subarray, it returns that element as the dominant element. This is correct because if there is a dominant element in the entire subarray, it must be one of these two elements, and their counts will add up to be more than half of the subarray.

Since the algorithm correctly handles the base case and makes the correct decision in the inductive step, it correctly identifies the dominant element (if it exists) in the entire sequence.

Termination: The algorithm eventually terminates because it divides the input subarray into smaller subarrays, and the size of the subarray reduces by half in each recursive call. Eventually, it reaches subarrays of size 1 (the base case), and it returns a single element as the dominant element or "NULL."

Therefore, the algorithm is correct in identifying the dominant element (if it exists) and returns "NULL" when there is no dominant element in the sequence.

Time Complexity :

since we are dividing the array into two and calling the findDominantElement function recursively but now with size n/2. And in the function countOccurrences ,we are iterating through "nums" which is $\mathcal{O}(n)$. so the time complexity taken is $T(n) = 2T(n/2)+\mathcal{O}(n)$ We can simplify the analysis using masters theorem , (case-2) i.e, If $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, where $a \geq 1$, $b > 1$, and $f(n)$ is an asymptotically positive function, and there exists a constant $\epsilon > 0$,If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$. which is **O(nlogn)**

2 Problem - 2

Algorithm 2 Find Median of Two Sorted Arrays of same length

```
function FindMedianSortedArrays(A, B)
    if length(A) ≠ length(B) then
        throw ValueError("Input arrays must have equal lengths")
    end if
    if length(A) = 1 then
        return min(A[0], B[0])
    end if

    medA ← FindMedian(A)
    medB ← FindMedian(B)
    if medA = medB then
        return medA
    end if
    if medA < medB then
        return FindMedianSortedArrays(Subarray of last [⌊length(A)/2⌋ A elements, Subarray of the first [⌊length(B)/2⌋ B elements)
    else
        return FindMedianSortedArrays(Subarray of first [⌊length(A)/2⌋ A elements, Subarray of the last [⌊length(B)/2⌋ B elements)
    end if
end function
function FindMedian(arr)
    n ← length of arr
    if n is odd then
        return arr[⌊n/2⌋]
    else
        return ← arr[(n/2) − 1]
    end if
end function
```

Correctness Using Induction:

Assumption: Let us define the median of 2k elements as the element that is greater than k − 1 elements and less than k elements.

Base Case: For arrays of length 1, the algorithm directly calculates the min of the both arrays A,B and returns it , since it will be the median.

Inductive Hypothesis: Assume that the algorithm correctly finds the median for arrays of length k, where $k > 1$.

Inductive Step: We need to show that the algorithm correctly finds the median for arrays of length $k + 1$.

The algorithm calculates $medA$ and $medB$ for arrays of length $k + 1$. It then compares $medA$ and $medB$.

- If $medA$ equals $medB$, the algorithm returns either of them, which is correct. - If $medA$ is less than $medB$, the algorithm recursively searches for the median in the right half of A (elements after $medA$) and the left half of B (elements before $medB$). By the inductive hypothesis, the algorithm correctly finds the median for subarrays of length k, and therefore, it will also correctly find the median for the arrays of length $k + 1$. - If $medA$ is greater than $medB$, the algorithm recursively searches for the median in the left half of A and the right half of B, which is similar to the previous case and will be correct by the inductive hypothesis.

Termination: The algorithm eventually terminates because the array size is exponentially decreasing (2n->n->n/2...) in each recursive call. Eventually, it reaches subarrays of size 1 (the base case), and it returns a median or throws error if subarrays are of different length.

The algorithm reduces the problem size by half in each recursive call, and it handles the base case correctly. Therefore, it will correctly find the median for arrays of any length k.

Time Complexity :
since we The algorithm reducees the problem size by half in each recursive call. And in the function FindMedian takes constant time. so the time complexity taken is $T(n) = T(n/2) + \mathcal{O}(constant)$ We can simplify the analysis using masters theorem , (case-2) i.e, If $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, where $a \geq 1$, $b > 1$, and $f(n)$ is an asymptotically positive function, and there exists a constant $\epsilon > 0$,If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$. which is **O(logn)**