Name & Surname:
ID:

CSE 321 - Quiz #5

December 27^{th}, 2024

1. ***40 pts.*** Given an array of n integers, your task is to find the largest sum of a subarray (i.e. a sequence of consecutive elements in the given array). Design a divide-and-conquer algorithm to find the largest sum. Provide the pseudo-code of your algorithm along with an explanation, and analyze its worst-case time complexity.

Example:
Input: [-2, 1, -3, 4, -1, 2, 1, -5, 4]
Output: 6
Explanation: The subarray [4, -1, 2, 1] has the largest sum, which is 6.

Solution: To solve this problem by using a divide-and-conquer approach, we can break the array into two halves, solve the problem for each half recursively, and then combine the results. The key point here is that the largest subarray sum can be on the left half, on the right half, or on the combination of left and right halves.

Since we divide the array into two halves and then solve both of them at each step, our recurrence relation forms as $T(n) = 2 \cdot T(\frac{n}{2}) + f(n)$. $f(n)$ stands for the combination part which takes linear time since we traverse the array (see *MaxCrossSum* function in the pseudo-code next page). By this, we obtain $T(n) = 2 \cdot T(\frac{n}{2}) + O(n)$. This can be solved by using master theorem.

The worst-case time complexity is $O(n \cdot logn)$.

```
FindMaxSum(A, n):
    Input: An integer array A and its length n.
    Output: Largest sum of a subarray.
    begin
        return FindMaxSumHelper(A, 0, n-1)
    end


FindMaxSumHelper(A, l, r):
    Input: An integer array A, left index l and right index r.
    Output: Largest sum of a subarray, between the provided indices.
    begin
        if l > r then
            return -∞
        end if
        if l == r then
            return A[l]
        end if

        m ← (l + r) // 2
        LeftSolution ← FindMaxSumHelper(A, l, m)
        RightSolution ← FindMaxSumHelper(A, m, r)
        CrossSolution ← MaxCrossSum(A, l, m, r)

        return Max(LeftSolution, RightSolution, CrossSolution)
    end


MaxCrossSum(A, l, m, r):
    Input: An integer array A, left index l, middle index m, and right index r.
    Output: Largest sum of the subarray crossing the middle.
    begin
        LeftSum ← 0
        RightSum ← 0
        LeftMax ← −∞
        RightMax ← −∞

        for i in range (m, l) repeat
            LeftSum ← LeftSum + A[i]
            LeftMax ← Max(LeftSum, LeftMax)
        end for

        for i in range (m, r) repeat
            RightSum ← RightSum + A[i]
            RightMax ← Max(RightSum, RightMax)
        end for

        return Max(LeftMax, RightMax, LeftMax + RightMax - A[m])
    end
```

2. ***40 pts.*** Given an array of n integers, your task is to find the majority element in the array. The majority element is the element that appears more than $\frac{n}{2}$ times in the array. Design a divide-and-conquer algorithm to find the majority element. Provide the pseudo-code of your algorithm along with an explanation, and analyze its worst-case time complexity.

Example: Input: [3, 4, 4, 2, 4, 4, 2, 4] Output: 4

PS: You can assume that the given array always has a majority element.

Solution: As in the first question, we can break the array into two halves, solve the problem for each half recursively, and then combine the results. If the solutions of the left and right halves are different, we must count the occurances of these solutions to decide on which one the choose.

Since we divide the array into two halves and then solve both of them at each step, our recurrence relation forms as $T(n) = 2 \cdot T(\frac{n}{2}) + f(n)$. $f(n)$ stands for the combination part which takes linear time since we traverse the array (see *Count* function in the pseudo-code next page). By this, we obtain $T(n) = 2 \cdot T(\frac{n}{2}) + O(n)$. This can be solved by using master theorem.

The worst-case time complexity is $O(n \cdot logn)$.

```
FindMajorityElement(A, n):
    Input: An integer array A and its length n.
    Output: The majority element.
    begin
        return FindMajorityElementHelper(A, 0, n-1)
    end


FindMajorityElementHelper(A, l, r):
    Input: An integer array A, left index l and right index r.
    Output: The majority element between the provided indices.
    begin
        if l == r then
            return A[l]
        end if

        m ← (l + r) // 2
        LeftMajority ← FindMajorityElementHelper(A, l, m)
        RightMajority ← FindMajorityElementHelper(A, m, r)
        if LeftMajority==RightMajority then
            return LeftMajority
        end if

        LeftCount ← Count(A, l, r, LeftMajority)
        RightCount ← Count(A, l, r, RightMajority)

        if LeftCount > RightCount then
            return LeftMajority
        else
            return RightMajority
        end if
    end


Count(A, l, r, x):
    Input: An integer array A, left index l, right index r, and an element x.
    Output: Occurances of x between the provided indices.
    begin
        c ← 0
        for i in range (l, r) repeat
            if A[i] == x then
                c ← c + 1
            end if
        end for
        return c
    end
```

3. ***20 pts.*** Consider the following recurrence relation and its initial condition.
$f(n) = f(\lfloor \frac{n}{2} \rfloor) + f(n-1) + 1, \quad f(0) = 2$
Design a dynamic programming algorithm to compute $f(n)$ for a given positive integer n. Provide the pseudo-code of your algorithm along with an explanation, and analyze its worst-case time complexity.

Solution: This problem is similar to the Fibonacci numbers. Instead of solving overlapping sub-problems, we should memoize the solutions we have already solved.

The provided pseudo-code has a worst-case time complexity of $O(n)$ since it only has constant-time operations and a loop with n iterations.

```
ComputeF(n):
    Input: A positive integer array n.
    Output: f(n).
    begin
        DP ← Empty array with n + 1 elements
        DP[0] ← 2
        for i in range (1, n) repeat
            DP[i] ← DP[i//2] + DP[i-1] + 1
        end for
        return DP[n]
    end
```