3500 hw 6

Casey Provitera

March 2024

1 Question 1 (40 points)

Find the solution to the given recurrence. If applicable, make use of the Master Theorem. Present the process step by step.

1. T(n) = 16T(n/4)+n^2
   a = 16, b = 4, f(n) = n^2
   $log_b a = log_4 16 = 2$
   $n^{log_b a} = n^2$ = f(n)
   Case 2 of master theorem O(f(n)*logn) = O($n^2 * logn$)

2. T(n) = 7T(n/2)+n^2
   a = 7, b = 2, f(n) = n^2
   $log_b a = log_2 7 = 2.8$
   $n^{log_b a} = n^{2.8} >$ f(n)
   Case 1 of master theorem O($n^{2.8}$)

3. T(n) = 4T(n/1)+$n^2 * \sqrt{n}$
   a = 4, b = 1, f(n^2)
   $log_b a = log_1 4$ = undef
   next steps need to be without the master theorem
   This problem breaks into 4 sub problems but all of them with the same size as the original problem. Intuitively this mean the most work will be done at the leaves since everything we make a new set of sub problems we need to do 4 times the amount of work compared to the step before.

4. T(n) = T(n-1)+1
   Can not use master theorem
   T(n-1) = T(n-2)+1
   T(n-2) = T(n-3)+1
   T(n-1) = T(n-3)+2
   T(n) = T(n-3)+3 This relation can be used to show the the following:
   Since T(n) is simply equal to the previous plus one we can simply when we use substituion we will be in a loop until we are at T(0) which we will also assume is a base case. To get to T(0) we need to have T(n-n) and since we always add the second part of the sub traction statement we can

evaluate to the below equation.
T(n) = T(0) + n
With a final run time of $\theta(n)$

2 Question 2 (20 points)

You are given an array where the values in the array increase up to a certain position p (peak point) and then decrease from position p to the end of the array.
Example:

- Input: Array A = [1, 2, 3, 4, 5, 4, 3, 2, 1]
- Output: Peak point: 5

Your goal is to write a divide and conquer algorithm to identify the peak point p Your algorithm should achieve a time complexity better than O(n), where n is the number of elements in the array.

1. Write a clear divide and conquer algorithm to solve the problem.

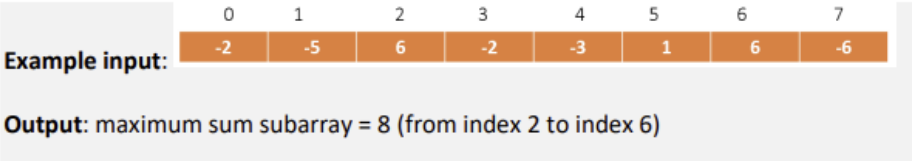
```
THE FOLLOWING IS THE ALGORITHM
def find_peak_point(arr, left, right):
    if left == right:
        return left
    mid = (left + right) // 2
    if arr[mid] < arr[mid + 1]:
        return find_peak_point(arr, mid + 1, right)
    else:
        return find_peak_point(arr, left, mid)
```

2. Provide the recurrence relation that describes the time complexity of your algorithm.
   The recurrence relation for peak point is as follows:
   relation = a*T(n/b)+f(n)
   a = 1
   b = 2
   f(n) = $\theta(1)$
   final relation = 1*T(n/2) + $\theta(1)$

3. Analyze the time complexity of your algorithm. If applicable, use the Master Theorem.
   We can use the master theorem to analysis the time complexity.
   $log_b a = log_2 1 = 0 -> n^0 = 1$
   1 = f(n), case 2 of master theorem, O(f(n)*logn) = O(logn)

3 Question 3 (40 points)

The maximum sum subarray problem involves finding the contiguous subarray within an array of integers that has the largest sum. Given an array of integers, you need to design a divide and conquer algorithm to find the maximum sum of a subarray.



Task: Research the algorithm and make sure that you fully understand the problem:

1. Provide a high-level explanation of how the divide and conquer approach is used to tackle this problem.
   The find maximum sub array problem is a widely known problem with many solutions some being greedy algorithms and some being divide and conquer algorithms. The goal in the problem is to find the maximum sub array within an array of positive and negative values. The divide and conquer approach does this by splitting the array into a left and right group until the original array is split into sub arrays of length 1. Then the function will create a maxleft value which is the maximum value from the left array, a maxright value which is the maximum value from the right array and a maxcross value which is the maximum value from the sub array that crosses the midpoint of the original array. After creating these three values the function will return the maximum of those three, and continue all the way to the top of the recursive stack to get the final answer.

2. Write the algorithm.
   I have made the helper function max sub crossing array as well simply so the real function looks nicer and is easier to understand a high level description of.

```
def find_max_crossing_subarray(arr, low, mid, high):
    left_sum = float('-inf')
    max_left = 0
    max_right = 0
    total = 0

    for i in range(mid, low - 1, -1):
        total += arr[i]
        if total > left_sum:
```

```
            left_sum = total
            max_left = i

    right_sum = float('-inf')
    total = 0

    for j in range(mid + 1, high + 1):
        total += arr[j]
        if total > right_sum:
            right_sum = total
            max_right = j

    return max_left, max_right, left_sum + right_sum


def find_max_subarray(arr, low, high):
    if low == high:
        return low, high, arr[low]

    mid = (low + high) // 2

    left_low, left_high, left_sum = \
        find_max_subarray(arr, low, mid)
    right_low, right_high, right_sum = \
        find_max_subarray(arr, mid + 1, high)
    cross_low, cross_high, cross_sum = \
        find_max_crossing_subarray(arr, low, mid, high)

    if left_sum >= right_sum and left_sum >= cross_sum:
        return left_low, left_high, left_sum
    elif right_sum >= left_sum and right_sum >= cross_sum:
        return right_low, right_high, right_sum
    else:
        return cross_low, cross_high, cross_sum
```

3. Provide the recurrence relation that describes the time complexity of your algorithm.
   The recurrence relation for max sub array is as follows:
   relation = a*T(n/b)+f(n)
   a = 3
   b = 3
   f(n) = $\theta(n)$
   final relation = 3*T(n/3)+$\theta(n)$

4. Analyze the time complexity of your algorithm. If applicable, use the

Master Theorem.
We can use the master theorem to analysis the time complexity.
$log_b a = log_3 3 = 1 -> n^1 = n$
n = f(n), case 2 of master theorem, O(f(n)*logn) = O(nlogn)

Additional Guidelines:

- You can assume that the input is a one-dimensional array of integers.
- Your algorithm should divide the problem into smaller subproblems, solve them recursively, and combine their results to find the maximum subarray.
- Pay close attention to the divide and merge steps in your algorithm.
- Explain the key components of your algorithm and how it ensures the correct maximum subarray is found.
  The most important part of the algorithm is that it checks the right left and cross max against each other, since these are the only places the true max sub array can fall we must check all of them. This ensures we find the correct max sub array because once it finds a large sub array it will not change the answer unless a larger one is found. This can be explained by example by saying if there is a negative number at the next index of the current maximum sub array it will not chnage the answer of the max sub array, but if there is a larger positive number after that negative number it will change the maximum sub array.