Zuhair Ahmed
CS 325-400 Spring 2020
4/12/2020

Homework #2

Problem 1) Give the asymptotic upper bounds for T(n) in each of the following recurrences. Show your work and explain how you solve each case.
Problem 1.a. (2 points)

- $T(n) = b \cdot T(n - 1) + 1$

where b is a fixed positive integer greater than 1.

Using the Master Formula for Decreasing Functions: $a*T(n - d) + O(n^k)$
$a = b$, $d = 1$
Since b is greater than 1 in Master Foruma, This implies that when b = 2, then $O(2^n)$, when b = 3, then $O(3^n)$, etc.

Hence, $T(n) = \mathbf{O(b^n)}$

Problem 1.b. (2 points)

- $T(n) = 3 \cdot T(n/9) + n \cdot \log n$

Using the Master Formula for Dividing Functions: $a*T(n / b) + f(n)$
$a = 3$, $b = 9$, $f(n) = n*\log n$

Hence $T(n)=O(n^{\log_a b}\log^{k+1} n)=\mathbf{O(n*\log n)}$

Problem 2)
4.1-1.
Find-Maximum-Subarray will result in the least negative position. Since the cross sums are computed, the one with the most positive should have the smallest length possible.

4.1-2.

Brute Force Algorithm to Solve Maximum Subarray Problem

```
{
	left = 1;
	right = 1;
	max = Z[1];
	curSum = 0;
	for (i = 1 to n) // Increment left end of subarray
	{
		curSum = 0;
		for (j = i to n) // Increment right end of subarray
		{
			curSum = curSum + Z[j];
			if (curSum > max)
			{
				max = curSum;
				left = i ;
				right = j;
			}
		}
	}
}
```

4.1-3.

I implemented both the Brute-Force method and the Recursive method in C++ the on latest gpp compiler on OSU engineering FLIP servers. The crossover point is at approx. at **22 length array**. I then changed the base case of the recursive algorithm to use the brute-force algorithm whenever the problem is less than $n_{0.}$ This materially made the recursive algorithm for smaller values of n.

4.1-4.

By changing the definition of the maximum-subarray problem to allow the result to be an empty subarray where the sum of the values of an empty subarray 0, we would need to alter other parts of the algorithm. In particular, we would need to change the parts that do not allow empty subarrays to then permit empty subarrays to be the result. One suggestion of how to do this could be to implement a linear scan of the input array to check if it contains any > 0 (positive) entries. If so then run the algorithm as normal. Else, return the empty subarray with sum 0 and force exit the algorithm.

4.1-5.
Linear Time Maximum Subarray Algorithm

```
{
        //For subarray A[]
        M = −999;
        lowM = 0;
        highM = 0;
        Mr = 0;
        lowr = 1;
        for i from 1 to A.length
        {
                Mr+ = A[i] ;
                if Mr > M
                {
                        lowM = lowr;
                        highM = i ;
                        M = Mr ;
                }
                if Mr < 0
                {
                        Mr = 0;
                        lowr = i + 1;
                }
        }

return (lowM, highM, M);
}
```

Problem 3)
Consider the recurrence $T(n) = 3 \cdot T(n/2) + n$
Problem 3.a. (3 points)
• Use the recursion tree method to guess an asymptotic upper bound for T(n).
n
(n/2 n/2 n/2 n) → 3n/2
($n/2^2$ $n/2^2$ $n/2^2$ n) ($n/2^2$ $n/2^2$ $n/2^2$ n) ($n/2^2$ $n/2^2$ $n/2^2$ n) → 9n/4
……(repeat for k times)
Assume $n / 2^k = 1$
$n = 2^k$
k = logn
This summation is a geometric series which will evaluate to 3 * (n^($\log_2$3)/n) - 2
Hence, **O(n^($\log_2$3))** ≈ $O(n^{1.58})$

Problem 3.b. (3 points)

• Prove the correctness of your guess by induction. Assume that values of n are powers of 2.

Base case: assume T(1) = 1, T(2) = 5...therefore T(2) < $c*2^{log3}$ or 5 < $c*2^{log3}$

Inductive Hypothesis: there exists an m < n, T(m) < $c*m^{log3}$ -d*m holds true for some values of c and d

Inductive step:

level 1 - $T(n) = 3 \cdot T(n/2) + n$

level 2 - $T(n/2) = 3 \cdot T(n/2^2) + n/2$ [substitute this back in formula above]

$T(n) = 3\cdot [3\cdot T(n/2^2) + n/2] + n$

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

hence next level would be...

$T(n) = 3[3^2\cdot T(n/2^2) + 2n] + n$

$T(n) = 3^3\cdot T(n/2^3) + 7n$

Assume this goes on for k levels...

Therefore, $T(n) = 3^kT(n/2^k) + 2kn$

Assume $T(n/2^k) = T(1) \rightarrow n = 2^k \rightarrow k = \log n$

T(n) = n^(T(1) + 2kn)

This would give us $3 \cdot n^{(\log_2 3)} - 2n$

Hence, $T(n) = O(n^{(\log_2 3)})$

Problem 4)

Consider the following pseudocode for a sorting algorithm, for 0 < α < 1 and n > 1.

```
badSort(A[0 · · · n - 1])
        if (n = 2) and (A[0] > A[1])
                swap A[0] and A[1]
        else if (n > 2)
                m = [α · n]
                badSort(A[0 · · · m - 1])
                badSort(A[n - m · · · n - 1])
                badSort(A[0 · · · m - 1])
```

Problem 4.a. (3 points)

• Show that the divide and conquer approach of badSort fails to sort the input array if $\alpha \leq 1/2$

If we assume that $\alpha \leq ½$, then badSort will sort first half of array at A[0...m -1], then it sorts second half, and then unfortunately sorts A[0...m -1] again. Problem is that the two halves of the array do not overlap so there could be a value in the second half array that is > for some of the values in the first half array or the other way around. The net result is an array that is not sorted.

Problem 4.b. (2 points)

• Does badSort work correctly if α = 3/4? If not, why? Explain how you fix it.

No it does not work correctly, and will go on and on in an infinite loop. This is because we have m = (¾ * n) = 3 for when n=3. Problem has to do with termination condition which is currently set at n=2, so in theory changing this to n=3 should resolve issue

Problem 4.c. (2 points)

• State a recurrence (in terms of n and α) for the number of comparisons performed by badSort.

T(n) = 3*T(α * n) + 1, this is because function performance recursion on 3 sub calls each time of size α * n each. Also need to add a constant for terminating case

Problem 4.d. (2 points)

• Let α = 2/3, and solve the recurrence to determine the asymptotic time complexity of badSort

If we use the Master Theorem, a = 3, b = 1/α, and α = ⅔ (as stated).
This results in $\log_{1.5}3$ which is approx. 2.7095, hence **$T(n) = \Theta(n^{2.7095})$**

Problem 5)

Ran out of time before midnight submission so will have to skip this coding problem in its entirety. Will submit this PDF to both Canvas and TEACH though for reference.