Student - s4530974

COMP3506 - Homework 1

Semester 2 - 2020

Contents

List of Figures

Question 1

1a - The Running Time of T_{odd}

If n is odd then the if statement skips to this section of the function which then becomes T_{odd}.

```
function T_odd(int n)
    sum ← 0 [1]
    while n > 0 do [?? + 1]
        sum ← sum + (n&1) [3]
        n ← (n >> 1) [2]
    end while
    return sum [1]
end function
```

Solving the ?? Part

CHECK - I have no idea what this is uhhhh

The running of the while loop part of the function is determined by the value of n and its most significant bit, you can assume that this bit would determine the times that the while loop would execute within T_{odd}. Since the most significant bit is being used as an indicator of how many times the while loop executes, you can use a formula to find which power of 2 the most significant bit would lie on. This is done with the equation

$$\frac{\log n}{\log(2)}$$

Multiplying this equation by the amount of operations within the while loop and then adding 1 to the amount of iterations to account for the while loop check will give:

$$(\frac{\log n}{\log(2)} * 5) + 1$$

You also have to account for the assigning of sum and returning of sum at the beginning as well as the if statement. This adds on 2 extra to the constant at the end resulting in the ending formula of:

$$(\frac{\log n}{\log(2)} * 5) + 3$$

An assumption made was that the $n\&1$ and $sum + (n\&1)$ counted as 1 and 2 primitive operations each respectively which is why I counted the line with the assignment as 3. I am also assuming that $\frac{\log n}{\log(2)}$ gives an int rounded down.

1b - Finding a suitable function for $O(n)$

A suitable function $g(n)$ that would fit T_{odd} such that $g(n)\epsilon O(n)$. Since we know that the runtime function can be described as $(\frac{\log n}{\log(2)} * 5) + 3$ we can start from there.

First, remove all constants and lower order terms which leaves us with $\frac{\log n}{\log(2)}$. We then have to find the bound for $O(n)$ which can be defined by $f(n) \leq c * g(n)$.

Suppose $g(n) = n^2, n_0 = 2, c = 3$

Then

1c - Finding Ω

The Lower Bound for $T_{odd}\epsilon\Omega(n)$ when $c =??$ and $n_0 =??$

Prove above thing

In terms of $\Theta(n)$, I believe it does not exist as there is no guaranteed run time or within T_{odd} that will be executed. It also does not exist as the upper and lower bounds are too small for $\Theta(n)$ to be defined.

1d - Bounds for T_{even}

```
function T_even(int n)
    for i = 0 to n do [n-1]
        for j = i to n^2 do [n^2 − i]
            sum ← sum + i + j [3]
        end for
    end for
    return sum
end function
```

The Upper Bound for $T_{even} \epsilon O(n^3)$ when $c =??$ and $n_0 =??$

The Lower Bound for $T_{even} \epsilon \Omega(n)$ when $c =??$ and $n_0 =??$

1e

The best case for this algorithm would be if n = 1. The run time of this would be

The worst case for this algorithm would be contained within T_{even} and would be $O(n^3)$

1f

1g

1h

Question 2

2a

```
function FINDPOSITIONRECURSE(A[n], int low, int high)
    if high ≥ low then
        mid ← low + (high − low)/2
        if mid == A[mid] then
            return true
        end if
        if mid < A[mid] then
            return FINDPOSITIONRECURSE(A, low, (mid - 1))
        else
            return FINDPOSITIONRECURSE(A, (mid + 1), high)
        end if
    else
        return false
    end if
end function


function FINDPOSITION(A[n])
    return FINDPOSITIONRECURSE(A, 0, n - 1)
end function
```

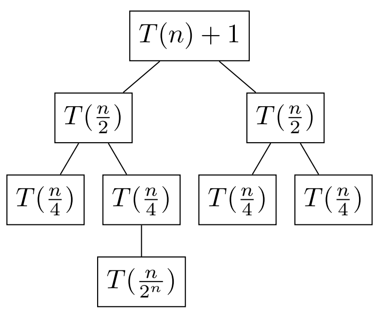
2b

The steps taken within my function would take in the array and then call the recursive function FIND-POSITIONRECURSE low and high values set as 0 and n - 1 respecitvely. The function would then begin searching to see if $A[i] == i$. The recursion would be as follows:

- -1 $\rightarrow$the function would calculate and set mid as 4, it would then check if A[4] == 4, which it is not. It would then check if 4 ¡ A[4], which it is. It would the enter recursion and pass in low as the same value but change high to be that of (mid - 1).
- 0 $\rightarrow$the function would calculate and set mid as 1, it would then check if A[1] == 1, which it is not. It would then check to see if 1 ¡ A[1], which it is not. It would then enter recursion again and pass in high as the same value and change low to be that of (mid + 1).
- 2 $\rightarrow$the function would calculate and set mid as 2, it would then check to see if A[2] == 2, which it is. The function would then exit recursion and return true.

2c

The worst case for my algorithm is where none of the values in A meet the conditions and the if high and low variables do not meet the requirements for the if statement $high \geq low$ and false is returned. The recurrence for this worse case would look like the following recurrence tree:



When put into a formula, the recurrence equation would look like the following:

$$\begin{aligned} T(n) &= T(\tfrac{n}{2})... + 1 \\ T(n) &= T(\tfrac{n}{2}) + T(\tfrac{n}{4}) + T(\tfrac{n}{8})...T(\tfrac{n}{2^k}) + 1 \end{aligned} \tag{1}$$

The constant is 1 because that is what is used for the return statement out of recursion.

Since the tree eventually ends as $T(\frac{n}{2^k})$, you could break this function down to

Since we know that $T(1) = 1$ as given by the base case. We can do the following:

$$\frac{n}{2^k} = 1$$

$$n = 2^k$$

$$k = log_2 n$$

Therefore $O(g(n))$ of this algorithm is $O(log(n))$.

2d

2di

Since we know that $O(log(n))$ and that

$$T(n) = \begin{cases} T(\frac{n}{2^k}) + 1, & \text{if } n > 1. \\ 1, & n = 1. \end{cases} \tag{2}$$

With $a = 1, b = 2^k, c = 1, g(n) = 1$.

Since $g(n) \epsilon \Theta(n^d), d = 0$

Using the Master Theorem, since $a = 2^{k^0} = 1$ then $T(n) = \Theta(log(n))$. The algorithm is therefore $\Theta(log(n))$.

2dii

$$T(n) = 5 * T(\frac{n}{3}) + n^2 + 2n$$

Where $a = 5, b = 3, g(n) = n^2 + 2n$

Since $g(n) \epsilon \Theta(n^d)$

Then $g(n) \epsilon \Theta(n^2)$ and $d = 2$

Sicne $5 < 3^2$, the Θ bounds for $T(n)$ is $\Theta(n^2)$ where 2 is d for when $T(1) = 100$.

2diii

Since n = k, the Θ is $\Theta(1)$ for when $T(1) = 1$.

2e

function FINDPOSITION(A[n])
    low $\leftarrow 0$
    high $\leftarrow (n - 1)$
    while *high* $\geq$ *low* **do**

$mid \leftarrow low + (high - low)/2$
if mid == A[mid] **then**
    return true
end if
if $mid < A[mid]$ **then**
    $high \leftarrow (mid + 1)$
else
    $low \leftarrow (mid - 1)$
end if
end while
return false
end function

The runtime complexity of my second solution is within $O(log(n))$ time. This is because while there is no recursion within my program, it is still a binary search implementation which works by splitting search size by 2 which eventually

2f

I believe that the better function to run within Java will be my iterative one. This is because if there are more elements within A, there runs the risk of getting a Stack Overflow during the program.

Question 3