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}.

```
1: function T_odd(int n)
2:     sum ← 0 [1]
3:     while n > 0 do [?? + 1]
4:         sum ← sum + (n&1) [3]
5:         n ← (n >> 1) [2]
6:     end while
7:     return sum [1]
8: end function
```

Solving the ?? Part

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}.

The function cannot be $O(n)$ as the amount of while loop iterations does not linearly increase as the number increases. As seen below with the first 9 inputs, going from 5 to 7 does not increase the amount of while loops iterations by a constant factor but is actually the same.

Number	While Loop Iterations
1	1
3	2
5	3
7	3
9	4
11	4
13	4
15	4
17	5

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

$$\left\lceil \frac{\log(n)}{\log(2)} \right\rceil$$

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:

$$\left\lceil \frac{\log(n)}{\log(2)} \right\rceil + 1$$

This equation was then multiplied by the number of primitive operations within the while loop which add up to be 5. This was then multiplied by the existing equation to give:

$$\left\lceil \frac{5\log(n)}{\log(2)} \right\rceil + 5$$

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:

$$\left\lceil \frac{5\log(n)}{\log(2)} \right\rceil + 7$$

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.

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 $\left\lceil \frac{5\log(n)}{\log(2)} \right\rceil + 7$ we can start from there.

First, remove all constants and lower order terms which leaves us with $\frac{\log n}{\log(2)}$.

Using the definition for $O(n)$ which is: Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that $f(n) \leq c * g(n)$ for $n \geq n_0$

$$\text{Suppose } g(n) = n, n_0 = 5, c = 3$$

$$\text{Then } \frac{log(n)}{log(2)} \leq 3n$$

$$log(n) \leq 3n * log(2)$$

$$log(n) \leq 3n * log(2), n = 5$$

$$log(5) \leq 3 * 5 * log(2) \text{ is true}$$

$$\therefore \quad \frac{\log n}{\log(2)} \text{ is } O(n) \text{ when } n \geq 5, c = 3$$

1c - Finding Ω

Using the definition for $O(n)$ which is: Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\Omega(g(n))$ if there are positive constants c and n_0 such that $f(n) \geq c * g(n)$ for $n \geq n_0$

$$\text{Suppose } g(n) = log(n), n_0 = 5, c = 3$$

$$\text{Then } \frac{log(n)}{log(2)} \geq 3log(n) \ \frac{log(n)}{log(2)} \geq 3log(n)$$

$$\text{Since } \frac{log(n)}{log(2)} \text{ is always } \geq 3log(n)$$

$$\therefore \quad \frac{\log n}{\log(2)} \text{ is } \Omega(log(n)) \text{ when } n \geq 5, c = 3$$

In terms of $\Theta(n)$, I believe it does not exist as it only exists when Big-O == Big-Omega and $O(g(n)) \neq \Omega(g(n))$ so it does not exist.

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]
      $sum \leftarrow sum + i + j$ [3]
    end for
  end for
  return sum
end function

T_{even}

I am assuming that the condition for the for loop is $i \leq n$

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

My team mate is wrong as the best case does not define the bounds which are what Big-Ω and Big-O are. A worst and best case scenario describe a singular scenario and are a singular point within the space of the algorithm where Big-Ω and Big-O represent bound the lower and higher bounds respectively. They also define where worst and best cases can take place within the algorithm's asymptotic runtime.

1h - Some $\Theta(f(n))$ Proof

Statement - $\Theta(g(n))$ only exists when $O(g(n)) = \Omega(g(n))$.

This can be proved using the definition given as: $f(n)$ is $\Theta(g(n))$ if there exist positive constants c_1, c_2 and n_0 such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n \geq n_0$

Since $O(g(n)) = \Omega(g(n))$, for $\Theta(f(n))$ to exist, it must be $g(n)$ otherwise the definition does not hold.

Therefore $\Theta(f(n))$ is $\Theta(g(n))$ when $O(g(n)) = \Omega(g(n))$.

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
```

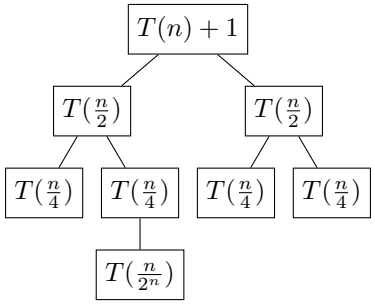
2b

The steps taken within my function would take in the array and then call the recursive function FINDPOSITIONRECURSE 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 →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 →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 →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 ← low + (high − low)/2
        if mid == A[mid] then
            return true
        end if
        if mid < A[mid] then
            high ← (mid + 1)
        else
            low ← (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, which is very much unwanted.

In terms of space complexity, I believe that

Question 3

Question 4

4a

4b

Returns a true if the drilled plot is a water source and false if it is not.

```
1: function IsWaterSource(G[n], int x, int y)
2:     return boolean isSource
3: end function
```

Returns a char relating to the source direction ('N', 'E', 'S', 'W')

```
1: function GetSourceDirection(G[n], int x, int y)
2:     return char direction
3: end function
```

```
1: function FindWaterSourceMinimal(G[n])
2:
3: end function
```