# Student - s4530974

## COMP3506 - Homework 1

Semester 2 - 2020

## Contents

# 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)
    $sum \leftarrow 0$ [1]
    while $n > 0$ do [?? + 1]
        $sum \leftarrow sum + (n\&1)$ [3]
        $n \leftarrow (n >> 1)$ [2]
    end while
    return $sum$ [1]
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:

$$\left\lceil \frac{\log(n)}{\log(2)} \right\rceil = \lceil \log_2(n) \rceil$$

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:

$$\lceil \log_2(n) \rceil + 1$$

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:

$$\lceil 5 \log_2(n) \rceil + 5$$

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:

$$\lceil 5\log_2(n)\rceil + 7$$

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 $\lceil 5\log_2(n)\rceil + 7$ we can start from there.

First, remove all constants and lower order terms which leaves us with $\lceil \log_2(n)\rceil$.

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 $f(n) \le c * g(n)$ for $n \ge n_0$

$$\text{Suppose } g(n) = log(n), n_0 = 5, c = 8$$
$$\text{Sub } c = 3,\ \lceil \log_2(n)\rceil \le 8log(n)$$
$$\text{Sub } n = 5,\ \lceil \log_2(5)\rceil \le 8log(n)$$
$$3 \le 5.59 \text{ is true}$$
$$\therefore \quad \lceil \log_2(n)\rceil\ \epsilon\ O(log(n)) \text{ when } n \ge 5, c = 8$$

## 1c - Finding $\Omega$

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 $f(n) \ge c * g(n)$ for $n \ge n_0$

$$\text{Suppose } g(n) = log(n), n_0 = 5, c = 3 \text{ Sub } c = 3,\ \lceil \log_2(n)\rceil \ge 3log(n)$$
$$\text{Sub } n = 5,\ \lceil \log_2(5)\rceil \ge 3log(5)$$
$$3 \ge 2.096...$$
$$\therefore \quad \lceil \log_2(n)\rceil\ \epsilon\ \Omega(log(n)) \text{ when } n \ge 5, c = 3$$

In terms of $\Theta(n)$, I believe it does exist. 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 $c_1 * g(n) \le f(n) \le c_2 * g(n)$ for all $n \ge n_0$.

Since $g(n)$ in this case would be $log(n)$ and $f(n)$ would be $log(n)$. Suppose $c_1 = 1$, $c_2 = 2$ and $n = 2$

$1 * log(n) \le log(n) \le 2 * log(n)$

$log(2) \le log(2) \le 2log(2)$, for all $n \ge 2$

## 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
```

$T_{even}$ has a runtime complexity of $(n + 1)(n^2)(3) = 3n^3 + 3n^2$

I am assuming that the condition for the for loop is $i \le n$ rather than $i < n$.

The Upper Bound for $T_{even}\ \epsilon\ O(n^3)$ when $c = 1$ and $n \ge 1$

The Lower Bound for $T_{even}\ \epsilon\ \Omega(n)$ when $c = 1$ and $n \ge 1$

## 1e

The worst case will always occur within $T_{even}$ as it has a larger runtime complexity. Therefore, the Big-$\Omega$ of the worst case in the algorithm is $\Omega(n^3)$ occuring in $T_{even}$. The same logic can be applied vice-versa to $T_{odd}$ and as such, the Big-$O$ of the best case in the algorithm is $O(log(n))$ occuring in $T_{odd}$.

## 1f

The Big-$\Omega$ Bound for the algorithm is $\Omega(log(n))$ while the Big-$O$ Bound for the algorithm is $O(n^3)$. In terms of Big-$\Theta$, I believe that it does not exist as $\Omega(g(n)) \neq \Theta(g(n))$.

## 1g

My team mate is wrong as the best case does not define the bounds which are what Big-$\Omega$ 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-$\Omega$ 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 $high \geq low$ then
        $mid \leftarrow low + (high - low)/2$
        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
```

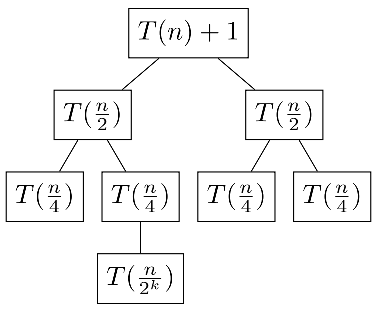
### 2b

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 - Recurrences

### 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 $\Theta$ bounds for $T(n)$ is $\Theta(n^2)$ where 2 is d for when $T(1) = 100$.

### 2diii

Since n = k, the $\Theta$ 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 the search size by 2 which eventually leads to $log_2(n)$ which gives the same complexity as the recursive one.

## 2f

For the consideration of larger inputs, I believe that the iterative method is better. This is because if there are more elements within A (as n increases), there runs the risk of getting a Stack Overflow due to a recursive depth being reached. This is mainly due to recursive methods relying on the stack which is ineffective for larger inputs.

In terms of space complexity, iterative methods rely on one stack instance which can get quite large. On the other hand as recursive methods continously create a new stack on each resursive call which takes up much more memory. This means that due to these multiple stacks, recursive methods will take up more space resulting in a slower program for larger inputs.

### References

Iteration vs Recursion

Time and Space Complexity of Recursive Algorithms

## Question 3

### 3b

The Big-$O$ for my CartesianPlane class is $O(nm)$ where n is the length and m is the width of the plane.

### 3c

The cartesian plane still needs to reserve space for null elements. Additionally large elements means more wasted space. You could store pointers in the array instead of the actual objects but this adds another level of indirection.

### 3d

$n$ and $m$ represent the length and width of the CartesianPlane

| Method | Big-$O$ | Reasoning |
|---|---|---|
| add | $O(1)$ | It takes constant time to add the value within the 2D array as you only need to get the indexes for the cartian plane |
| get | $O(1)$ | It takes constant time to access a variable within a 2D array as it only needs to index and return the value |
| remove | $O(1)$ | It takes constant time to get the value from the 2D array and return a boolean depending if the value could or could not be returned |
| resize | $O(nm)$ | The for loops within the function increase in complexity as the length-/width increase so the function has a complexity of $O(nm)$ |
| clear | $O(nm)$ | The for loops for setting each value within the 2D array to null/empty values change in complexity with the changing of the length/width of the array |

## Question 4

### 4a - My algorithm but in words

Some things I would like to note about my algorithm that I would need to implement/consider:

- The water source will always be below the corresponding x or y value when a drill breaks
- It's probably best to start from the center
- When a drill breaks, you should always make sure to check if the plot is a water source or not in case that both conditions are met
- Once you have the water direction on one axis, you do not need to search that axis again
- Bounds for searching for the water source would need to be set and changed during the algorithm for an efficient function

**Description of my algorithm**

First I would set bounds $x_n, y_n$ which would represent a bound to search for the water source starting at 0 to n - 1. Once an axis (direction for the water source) has been found, these bounds would change values to indicate this.

Then I would begin my algorithm starting from the center of the plot and dig there (I would round up if the length/width of the plot was even).

From there I would do the following checks:

1. Check if any axis ($x_n$ or $y_n$) has already been found.
   - If it has, skip the checks for the already found axis broken
2. Check if the drill breaks
   - Lower the bounds ($x_n$ or $y_n$) of the axis if the drill is broken.
   - Increase the bounds ($x_n$ or $y_n$) of the axis if the drill has not been broken
3. Check if a water source has been found
   - Set the axis ($x_n$ or $y_n$) as found if the direction is known

I would then dig again from the center of the new bounds set until a water source is eventually found.

The Big-$O$ complexity on the amount of holes to dig would be $O(log_2(n))$ holes as the bounds decrease to half/the center of the grid which leads to a smaller bound with each search. This search is a binary search across two axis and as such, the Big-$O$ notation of my algorithm is $O(log(n))$.

### 4b - Minimal Algorithm

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

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

Digs in the water hole in the x, y position and returns true if the drill is broken and false if it is not

```
function IsBroken(int x, int y)
    return boolean pumpStatus
end function
```

```
function FindWaterSourceMinimal(G[n])
    $x \leftarrow 0$
    $y \leftarrow 0$
    while $x < n$ do
        if $isBroken(x, y)$ then
```

```
            break
        else
            $x \leftarrow x + 1$
        end if
    end while
    while $y < n$ do
        if $isBroken(x, y)$ then
            break
        else
            $x \leftarrow y + 1$
        end if
    end while
    return (x, y)
end function
```

The Big-$O$ bound for my function is $O(2n)$ where n is the length of the grid.