# COMP 3804 Assignment 1

January 27, 2021

**Question 1.**

Name: Braeden Hall
Student Number: 101143403

**Question 2.**

1. $n = 3^k$
   $k = \log_3 n$

$$\begin{aligned} T(n) &= 1 + 2 \cdot T(n/3) \\ &= 1 + 2\Big[1 + 2 \cdot T(n/3^2)\Big] \\ &= 1 + 2 + 2^2 \cdot T(n/3^2) \\ &= 1 + 2 + 2^2\Big[1 + 2 \cdot T(n/3^3)\Big] \\ &= 1 + 2 + 2^2 + 2^3 \cdot T(n/3^3) \\ &= \cdots = \\ &= 1 + 2 + 2^2 + \cdots + 2^k \cdot T(n/3^k) \end{aligned}$$

   $T(n/3^k) = T(3^k/3^k) = T(1) = 1$
   $1 + 2 + 2^2 \cdots + 2^k \cdot 1 = 2^{k+1} - 1$
   $2^{k+1} = 2 \cdot 2^k = 2 \cdot 2^{\log_3 n} = 2 \cdot n^{\log_3 2}$
   Therefore, $T(n) = 2 \cdot n^{\log_3 2} - 1 \leq 2 \cdot n^{\log_3 2} = O(n^{\log_3 2})$

2. $n = k + 1$

$$\begin{aligned} T(n) &= 1 + 2 \cdot T(n-1) \\ &= 1 + 2\Big[1 + 2 \cdot T(n-2)\Big] \\ &= 1 + 2 + 2^2 \cdot T(n-2) \\ &= 1 + 2 + 2^2 \cdot \Big[1 + 2 \cdot T(n-3)\Big] \\ &= 1 + 2 + 2^2 + 2^3 \cdot T(n-3) \\ &= \cdots = \\ &= 1 + 2 + 2^2 + \cdots + 2^k \cdot T(n-k) \end{aligned}$$

   $T(n-k) = T(k+1-k) = T(1) = 1$
   $1 + 2 + 2^2 + \cdots + 2^k \cdot 1 = 2^{k+1} - 1 = 2^n - 1$
   Therefore, $T(n) = 2^n - 1 \leq 2^n = O(2^n)$

3. $n = 2^{2^k}$
   $\sqrt{n} = n^{\frac{1}{2}}$
   $k = \log_2 \log_2 n$

$$\begin{aligned} T(n) &= 1 + T\left(n^{\frac{1}{2}}\right) \\ &= 1 + \left[1 + T\left(n^{\frac{1}{2^2}}\right)\right] \\ &= 2 + T\left(n^{\frac{1}{2^2}}\right) \\ &= 2 + \left[1 + T\left(n^{\frac{1}{2^3}}\right)\right] \\ &= 3 + T\left(n^{\frac{1}{2^3}}\right) \\ &= \cdots = \\ &= k + T\left(n^{\frac{1}{2^k}}\right) \end{aligned}$$

$T\left(n^{\frac{1}{2^k}}\right) = T\left(\left(2^{2^k}\right)^{\frac{1}{2^k}}\right) = T\left(2^{\frac{2^k}{2^k}}\right) = T(2^1) = T(2) = 1$
$k + 1 = \log_2 \log_2 n + 1$
Therefore, $T(n) = \log_2 \log_2 n + 1 = O(\log_2 \log_2 n)$, since for large values of $n$ the '+1' is negligible.

**Question 3.**

*Refer to Algorithm 1 in the Appendix*

Description of running time:

By 2 we know that both $x + y$ and $x - y$ can be computed in linear or $O(n)$ time. Since $x$ and $y$ are both $n$ bit integers it is possible that their sum could have $n + 1$ bits. This means that the first call to $\mathcal{A}$ may have a running time of $S(n + 1)$, the second call to $\mathcal{A}$ will have a running time of at most $S(n)$. Both $a$ and $b$ will be integers with strictly less than $n^2$ bits, so that means by 2 they can be subtracted in some constant times linear time, or $O(n)$. The divisor in the last statement before the return is a power of 2, therefore, by 3 in can be done in $O(k)$ time, where $k = 2$ in this instance.

Since the largest possible running time of any of the statements in this algorithm is $S(n + 1)$, therefore the running time of algorithm $\mathcal{B}$ is $O(S(n + 1))$.

**Question 4.**

1. APPLE($n$) is a recursive algorithm, but the recursions always take place on smaller and smaller values of $n$. At some point each separate recursive call will make a call on the base case $n = 1$, this will cause the current call to terminate and return, which will cause/help its parent call terminate and return, and so on and so forth, until we return to the original (root) call. This root call will then either terminate or make another recursive call which will follow the same cycle and then terminate the root. Either way the root call will always terminate.

2. $$A(n) = \begin{cases} 0 & \text{if } n = 1 \\ 1 + A(m) + A(n - m) & \text{if } n \geq 2 \end{cases}$$

Claim: $A(n) = n - 1$
Base case:
Sub $n = 1$ into $A(n) = n - 1$

$$\begin{aligned} A(1) &= 1 - 1 \\ &= 0 \end{aligned}$$

This is the defined value for $A(1)$.

Inductive Hypothesis: Assume that $A(m) = m - 1$ for each $m \in \{1, 2, \ldots, n-1\}$.

Inductive Step:
Sub inductive hypothesis into $A(n) = 1 + A(m) + A(n - m)$

$$\begin{aligned} A(n) &= 1 + (m - 1) + (n - m - 1) \\ &= 1 + n - 1 - 1 \\ &= n - 1 \end{aligned}$$

Therefore, QED.

**Question 5.**

*Refer to Algorithm 2 in the Appendix*

Description of running time:

The first expensive step in my algorithm is the call to sort the set $S$ with a size of $n$. There are known algorithms that sort sequences of numbers in $O(n \log_2 n)$ time, so that is the assumed run time of that operation. My algorithm then calls the recursive subroutine $\mathcal{M}$.

This subroutine has the base case where $n = 1$ which will return the unchanged sequence $S$. If we are not in the base case, $\mathcal{M}$ makes 2 recursive calls, each on sequences half the length of the original. After the recursive calls come back, $\mathcal{M}$ traverses the both of the arrays returned by the 2 recursive calls and adds the some amount of elements to a new array that is returned at the end of the subroutine. This iteration takes worst case linear time (if every element was maximal in its respective half of the array), so $O(n)$.

In general, the subroutine makes 2 recursive calls ($a = 2$) on sequences of half the original length ($b = 2$) and then does a linear amount of work with the results ($d = 1$). Therefore, by case 2 of the master theorem, the subroutine $\mathcal{M}$ takes $O(n \log_2 n)$ time. As both the sorting and the subroutine take $O(n \log_2 n)$ time, it follows to say that the algorithm's running time is $O(n \log n)$.

**Question 6.**

*Refer to Algorithm 3 in the Appendix*

Description of running time:

Finding a value for $h$ for which $A[h] > x$ takes $O(\log_2 \frac{h}{2})$ since $h$ starts at 2. After $h$ has been determined (and assuming $A[h]! = x$, in which case the algorithm terminates) we know that if $x$ exists in the array it is between the values of $A\left[\frac{h}{2}\right]$ and $A[h-1]$ (the range is inclusive). Next the algorithm calls the binary search algorithm on the section of the array previously specified, this takes time proportional to the logarithm of the length of the sequence ($\frac{h}{2}$ in this case), i.e. $O(\log_2 \frac{h}{2})$. The rest of the algorithm just does simple comparisons which take constant time.

Since we know the range of the array that $x$ can be found in we therefore know that $n$ must be a value such that $\frac{h}{2} \leq n < h$. This means that $\log_2 n \geq \log_2 \frac{h}{2}$ which is the run time of the algorithm. Therefore, the algorithm runs in $O(\log n)$ time.

**Question 7.**

Claim: $\mathbb{E}(T) = O(\log n)$

Define the variable $m$ which is equal to the difference between $l$ and $r$ in the current iteration of the loop. For $i = 0, 1, 2 \ldots$

An iteration of the while loop is in phase $i$ if $\left(\frac{3}{4}\right)^{i+1} \cdot n < m \leq \left(\frac{3}{4}\right)^{i} \cdot n$.

*Note: There are $\log_{\frac{4}{3}} n$ possible phases since $\left(\frac{3}{4}\right)^{\log_{\frac{4}{3}} n} \cdot n = 1$, i.e. if we reach the $\log_{\frac{4}{3}} n^{th}$ phase there is only one element in the array, and regardless of whether it is the one we are looking for we have finished searching.*

We call the pivot $p$ bad if it lies within either the first quarter or the last quarter of the sorted sequence of length $m$, $p$ is good otherwise. The length of the good portion of the sequence is $\frac{m}{2}$, and therefore, $Pr(\text{"pivot } p \text{ is good"}) = \frac{1}{2}$. If $p$ is good then in the next iteration of the while loop the length of the sequence is $\leq m - \frac{m}{4} = \frac{3}{4} \cdot m$. If we were previously in phase $i$ we would advance to a phase $\geq i + 1$ in the next iteration since $\frac{3}{4} \cdot m \leq \left(\frac{3}{4}\right)^{i+1} \cdot n$. This means that choosing a good pivot will always advance us into a higher phase.

Now define the random variable $X_i$="the number of loop iterations in phase $i$". Also define $\alpha$ as the probability of success (success in this case being advancing the the next phase).
$\alpha = \frac{1}{2}$, so

$$\begin{aligned} \mathbb{E}(X_i) &\leq \frac{1}{\alpha} \\ &\leq \frac{1}{\frac{1}{2}} \\ &\leq 2 \end{aligned}$$

*Note: I use less than or equal because $p$ may be the index of $x$, in which case the algorithm terminates, or phases may be skipped altogether if a "very good" pivot is chosen (eliminating, say, half or 3 quarters of the sequence).*

We can now write an equation for $T$ as the sum of iterations spent in each phase times the work done on each iteration.

$$\begin{aligned} T &= \sum_{i=0}^{\log_{\frac{4}{3}} n} X_i \cdot c \\ \mathbb{E}(T) &= \sum_{i=0}^{\log_{\frac{4}{3}} n} \mathbb{E}(X_i) \cdot c \\ &\leq 2c \sum_{i=0}^{\log_{\frac{4}{3}} n} 1 \\ &= 2c \cdot \log_{\frac{4}{3}} n \end{aligned}$$

Therefore, the expected value of $T$ is $\leq 2c \cdot \log_{\frac{4}{3}} n$, which is $O(\log_{\frac{4}{3}} n)$.

## Appendix

**Algorithm 1** $\mathcal{B}(x, y)$:

$a \leftarrow \mathcal{A}(x + y)$
$b \leftarrow \mathcal{A}(|x - y|)$
$prod \leftarrow \frac{a-b}{4}$ ▷ $4 = 2^2$
**return** $prod$

**Algorithm 2** Maximal Algorithm

```
function MAXIMAL(S)
    n ← |S|
    Y ← sorted(S)
    return M(Y, n)
end function

function M(S, n)
    if n = 1 then
        return S
    end if

    A ← M(S[0 ... n/2 − 1], n/2)
    B ← M(S[n/2 ... n − 1], n/2)

    max ← []
    i ← 0
    while i < |A| and A[i].y > B[0].y do
        max.append(A[i])
        i ← i + 1
    end while
    for i = 0 to |B| do
        max.append(B[i])
    end for

    return max
end function
```

**Algorithm 3** $InfiniteBinarySearch(A, x)$

```
h ← 2
while A[h] < x do
    h ← 2h
end while
if A[h] = x then
    return true
else
▷ Assume BinarySearch(A, n, x) returns the index of element x in the array if it exists and −1 otherwise
    r ← BinarySearch(A[h/2 ... h − 1], h/2, x)

    if r > 0 then
        return true
    else
        return false
    end if
end if
```