Joshua Sloan
University of Wyoming
jsloan3@uwyo.edu

Homework 1
COSC 5110 - Analysis of Algorithms

1 Exercise 0.1

a) $f = \theta(g)$
b) $f = \mathcal{O}(g)$
c) $f = \theta(g)$
d) $f = \theta(g)$
e) $f = \theta(g)$
f) $f = \theta(g)$
g) $f = \Omega(g)$
h) $f = \Omega(g)$
i) $f = \Omega(g)$
j) $f = \Omega(g)$
k) $f = \Omega(g)$
l) $f = \mathcal{O}(g)$
m) $f = \mathcal{O}(g)$
n) $f = \theta(g)$
o) $f = \Omega(g)$
p) $f = \mathcal{O}(g)$
q) $f = \mathcal{O}(g)$

2 Exercise 1.4

Show that $log(n!) = \theta(n \log n)$.

Proof:

To show that $log(n!) = \theta(n \log n)$, we must show how the upper (log(n^n)) and lower (log($(\frac{n}{2})^{n/2}$)) bound of $log(n!)$ compare to $\theta(n \log n)$.

In other words, we must show both:

$log(n^n) = \mathcal{O}(n \log n)$
$log((\frac{n}{2})^{n/2}) = \Omega(n \log n)$

Beginning with $log(n^n) = \mathcal{O}(n \log n)$,
given that our log base is arbitrary, we will assign a log base of n to both sides S.T.
$\log_n(n^n) = \mathcal{O}(n \log_n n) \Rightarrow$ n = n*1, showing that $log(n^n) = \mathcal{O}(n \log n)$.

Now we must show: $log((\frac{n}{2})^{n/2}) = \Omega(n \log n)$.
By simplifying the left side of the equation, we can see that:
$log((\frac{n}{2})^{n/2})$ = $\frac{n}{2}$log$(\frac{n}{2})$ = $\frac{1}{2}$n*log($\frac{1}{2}$n),
which = $\Omega(n \log n)$.

$\because log(n^n) = \mathcal{O}(n \log n)$ and $log((\frac{n}{2})^{n/2}) = \Omega(n \log n)$
we conclude that $log(n!) = \theta(n \log n)$. □

3 Exercise 1.5

Show that:

$$\sum_{t=1}^{n} \frac{1}{t} = \theta(\log n)$$

Proof:

To show this, we must show how the upper and lower bound of the harmonic series compare to $\log n$.
To do this, we are going to decrease each denominator to the next power of two for the upper bound and increase each denominator to the next power of two for the lower bound S.T.

The original harmonic series =
$\sum_{t=1}^{n} \frac{1}{t} = \sum_{t=1}^{n} 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} + ...$

so the upper bound is:
$\sum_{t=1}^{n} 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + ...$

and the lower bound is:
$\sum_{t=1}^{n} \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{16} + ...$

By taking any value k along the harmonic series, where $1 < k < t$, we can see that the sum of the upper bound from 1 to k is $\approx \log_2 k$ with a steadily decreasing error as k $\to \infty$ (for example: for k=4, sum of series = 2+$\frac{1}{4}$ where $\log_2 4 = 2$ and for k=8, sum of series = 3+$\frac{1}{8}$ where $\log_2 4 = 3$, etc.). Because the error decreases toward infinity, for sufficiently large values k, it is negligible, S.T. the upper bounded harmonic sum from 1 to k = $\log_2 k$ and therefore, $\mathcal{O} \log_2 k$.

Similarly for the lower bounded harmonic series, by taking any value k along the harmonic series, where $1 < k < t$, we can see that the sum of the lower bound from 1 to k is $\approx \frac{1}{2}\log_2 k$ with a steadily decreasing error as k $\to \infty$ (for example: for k=4, sum of series = 1+$\frac{1}{8}$ where $\log_2 4 = 2$ and for k=8, sum of series = 1.5+$\frac{1}{16}$ where $\log_2 4 = 3$, etc.). Because the error decreases toward infinity (and even faster than our case for the upper bound), for sufficiently large values k, it is negligible, S.T. the lower bounded harmonic sum from 1 to k = $\frac{1}{2}\log_2 k$, which is just a coefficient of $\frac{1}{2}$ from $\log_2 k$ and therefore, $\Omega \log_2 k$.

$\therefore$ because the upper bound is $\mathcal{O}(\log_2 n)$ and the lower bound is $\Omega(\log_2 n)$, it follows that $\sum_{t=1}^{n} \frac{1}{t} = \theta(\log n)$. $\square$

4 Exercise 1.31

a)
We know that the number of bits needed to represent an integer N is $\log_2$N. Because we have already shown that $log(n!) = \theta(n \log n)$ in Exercise 1.4, it stands that the number of bits needed to represent N! is $\theta(\log N!)$.$\square$

b)
```
Factorial(N) // input is an n-bit integer, N
  if N = 0,1 then
    return N
  else
    return N*Factorial(N-1)
  end if
```
The recurrence relation for the recursive algorithm given above is: T(n) = T(n-1) + θ(1).

This problem cannot be analyzed via the Master Theorem, therefore, we will solve the recurrence relation using backward substitution S.T.

$$T(n) = \begin{cases} T(n-1) + \theta(1), n > 1 \\ 1, n = 1 \end{cases}$$

T(n) = T(n-1) + 1
= 1*[T(n-2) + 1] + 1
= 1*[T(n-3) + 1] + 2
= 1*[T(n-4) + 1] + 3
$\vdots$

= T(n-k) + k

suppose n = k+1

T(n) = T(k+1-k) + k
= T(1) + k
= 1 + k = n (given n = k+1) and $\therefore \mathcal{O}(n)$. □

5 Exercise 1.32

a)
Square(N) // input is a number, N

```
  for q=1 to N do
    temp = (q * q)
    if temp = N then
      return TRUE
    end if
    if temp > N then
      return FALSE // optimization to reduce unnecessary checks
    end if
  end for
  return FALSE
```

Because in the worst case we know that our algorithm will perform a constant amount of operations (3, i.e. 1 multiplication and 2 if-statement checks) N-times (although the optimization check should reduce this down further by some coefficient of n), we can state that this algorithm is $\mathcal{O}(n)$. □

b)
If N = q^k, show k≤log(N) or N=1

Proof:
Given N = q^k

⇒ log(N) = log(q^k)

⇒ log(N) = k*log(q)

⇒ $\frac{log(N)}{log(q)}$ = k

$\therefore$ k = $\log_q(N)$.

In the case that N=1, we know that k≤log(N) does not hold, as $\log_q(N)$ is 0 (or undefined if q = 1).
However, for all other N $\geq$ 2, k≤log(N) holds given that k = $\log_q(N)$. □

c)
Power(N) // input is a number, N

```
  for k=1 to N do
    for q=1 to N do
      temp = 1
      for i=0 to k do
        temp = temp * q
      end for
      if temp = N then
        return TRUE
      end if
      if temp > N then
        BREAK // optimization to reduce unnecessary checks
      end if
    end for
  end for
  return FALSE
```

The run time of this algorithm is $\mathcal{O}(n^3)$, which is polynomial and therefore, falls within the bounds of an efficient algorithm (although definitely not fast as far as polynomial time is concerned and can be optimized further to reduce the run-time). The algorithm is $\mathcal{O}(n^3)$ because intuitively, even in the worst case where the outer most loops complete all n-cycles and the inner most loop grows to complete (n-1) multiplications (plus the constant number if-statement checks), the run-time would still only total $\mathcal{O}(n * n * (n-1)) = \mathcal{O}(n^3)$ (although the BREAK line checks should allow redundant checks to be limited by some coefficient of n). □

6 Exercise 2.4

The recurrence relations for the three algorithm are as follows:
Algorithm A) $T(n) = 5T\left(\frac{n}{2}\right) + \theta(n)$
Algorithm B) $T(n) = 2T(n-1) + \theta(1)$
Algorithm C) $T(n) = 9T\left(\frac{n}{3}\right) + \theta(n^2)$

Algorithm A's running time can be calculated using the Master Theorem S.T.
a = 5, b = 2, d = 1
given that $1 < log_2 5$, T(n) = $\mathcal{O}(n^{\log_2 5})$.

Algorithm B's running time cannot be calculated using the Master Theorem given that b $\ngtr 1$.
Consequently, we must find the run-time of Algorithm B via backward substitution S.T.

$$T(n) = \begin{cases} 2T(n-1) + \theta(1), n > 1 \\ 1, n = 1 \end{cases}$$

T(n) = 2T(n-1) + 1
= 2*[T(n-2) + 1] + 1
= 4*[T(n-3) + 1] + 2
= 8*[T(n-4) + 1] + 3
$\vdots$
= 2^kT(n-k) + k

suppose n = k+1

T(n) = 2^kT(k+1-k) + k
= 2^kT(1) + k
= 2^k + k = O(2^{n-1} + (n-1)) (given n = k+1) and $\therefore$ $\mathcal{O}(2^n)$.

Algorithm C's running time can be calculated using the Master Theorem S.T.
a = 9, b = 3, d = 2
given that $2 = log_3 9$, T(n) = $\mathcal{O}(n^2 logn)$.

Therefore, I would clearly choose Algorithm C, given it has lowest time-complexity of the three algorithms provided. □

7 Exercise 2.5

a) T(n) = 2T(n/3) + 1
The run time of (a) can be calculated using the Master Theorem S.T.
a = 2, b = 3, d = 0
given that $0 < log_3 2$, T(n) = $\theta(n^{\log_3 2})$.

b) T(n) = 5T(n/4) + n
The run time of (b) can be calculated using the Master Theorem S.T.
a = 5, b = 4, d = 1
given that $1 < log_4 5$, T(n) = $\theta(n^{\log_4 5})$.

c) T(n) = 7T(n/7) + n
The run time of (c) can be calculated using the Master Theorem S.T.

a = 7, b = 7, d = 1
given that $1 = log_7 7$, T(n) = $\theta(n \log n)$.

d) T(n) = 9T(n/3) + n^2
The run time of (d) can be calculated using the Master Theorem S.T.
a = 9, b = 3, d = 2
given that $2 = log_3 9$, T(n) = $\theta(n^2 \log n)$.

e) T(n) = 8T(n/2) + n^3
The run time of (e) can be calculated using the Master Theorem S.T.
a = 7, b = 7, d = 1
given that $1 = log_7 7$, T(n) = $\theta(n \log n)$.

f) T(n) = 49T(n/25) + $n^{\frac{3}{2}} logn$
The run time of (f) can (actually) be calculated using the Master Theorem S.T.
a = 49, b = 25, d = 1.5 (lower bound shows relationship)
given that $1.5 > log_{25} 49$, we know T(n) = $\theta(n^{\frac{3}{2}} \log n)$.

g) T(n) = T(n-1) + 2
The run time of (g) cannot be calculated using the Master Theorem given that b $\not> 1$.
Therefore, we must find the run-time of (g) via backward substitution S.T.

$$T(n) = \begin{cases} T(n-1) + \theta(2), n > 1 \\ 1, n = 1 \end{cases}$$

T(n) = T(n-1) + 2
= 1*[T(n-2) + 2] + 2
= 1*[T(n-3) + 2] + 4
= 1*[T(n-4) + 2] + 6
$\vdots$
= T(n-k) + 2k

suppose n = k+1

T(n) = T(k+1-k) + 2k
= T(1) + 2k
= 1 + 2k = 1 + 2(n-1) (given n = k+1)
= 2n-1 and $\therefore \theta(n)$. $\square$

h) T(n) = T(n-1) + n^c
The run time of (h) cannot be calculated using the Master Theorem given that b $\not> 1$.
Therefore, we must find the run-time of (h) via backward substitution S.T.

$$T(n) = \begin{cases} T(n-1) + \theta(n^c), \text{ n>1 and c} \geq \text{ is a constant} \\ 1, n = 1 \end{cases}$$

T(n) = T(n-1) + n^c
= 1*[T(n-2) + n^c] + n^c
= 1*[T(n-3) + n^c] + 2n^c
= 1*[T(n-4) + n^c] + 3n^c
$\vdots$
= T(n-k) + kn^c

suppose n = k+1

T(n) = T(k+1-k) + kn^c
= T(1) + kn^c
= 1 + kn^c = 1 + (n-1)n^c (given n = k+1)
= 1+ n^{c+1} - n^c = n^c + 1 and $\therefore \theta(n^c)$. $\square$

i) T(n) = T(n-1) + c^n
The run time of (i) cannot be calculated using the Master Theorem given that b $\not> 1$.
Therefore, we must find the run-time of (i) via backward substitution S.T.

$$T(n) = \begin{cases} T(n-1) + \theta(c^n), \text{ n>1 and c } > 1 \text{ is some constant} \\ 1, n = 1 \end{cases}$$

T(n) = T(n-1) + c^n
= 1*[T(n-2) + c^n] + c^n
= 1*[T(n-3) + c^n] + 2c^n
= 1*[T(n-4) + c^n] + 3c^n
$\vdots$
= T(n-k) + kc^n

suppose n = k+1

T(n) = T(k+1-k) + kc^{k+1}
= T(1) + kc^{k+1}
= 1 + kc^{k+1} = 1 + (n-1)c^n
= nc^n = c^{n+1} and $\therefore$ $\theta(c^n)$. $\square$

j) T(n) = 2T(n-1) + 1
(j) is simply Algorithm B from Exercise 2.4 and $\therefore$ the time-complexity is $\theta(2^n)$.

k) T(n) = T($\sqrt{n}$) + 1
The run time of (k) cannot be calculated using the Master Theorem given that b $\not> 1$.
Therefore, we must find the run-time of (k) via backward substitution S.T.

$$T(n) = \begin{cases} T(\sqrt{n}) + 1, n > 2 \\ 1, n = 2 \end{cases}$$

T(n) = T(n$^{\frac{1}{2}}$) + 1
= 1*[T(n$^{\frac{1}{4}}$) + 1] + 1
= 1*[T(n$^{\frac{1}{8}}$) + 1] + 2
= 1*[T(n$^{\frac{1}{16}}$) + 1] + 3
$\vdots$
= T(n$^{\frac{1}{2^k}}$) + k

given our base case is 2, we need to solve for:
$N^{\frac{1}{2^k}}$ = 1
= $\frac{1}{2^k} logn = log(2)$ = $\frac{1}{2^k} logn = log(2)$
= $\frac{1}{2^k} logn = 1$
= $log(n) = 2^k$
= $loglog(n) = log(2^k)$
= $loglog(n) = k$

Because we know that our tree is a single branch where a constant amount of work being done at every given instance, our complexity is represented by the depth of the tree, which is denoted by k at any given instance (or n in the complete case) and $\therefore$ $\theta(loglogn)$. $\square$

I worked with Mason and Danny on this assignment.