Trevor Swan **CSDS233 ASSIGNMENT 1**

1. $f(N) = O(g(N))$: if $\exists\ c, n_0 > 0$ s.t. $f(N) \le c\,g(N)\ \forall\ N \ge n_0$.

a) $2N + 5 = O(N)$ True

$f(N) = 2N+5$, $g(N) = N$

$\lim_{N\to\infty} \frac{2N+5}{N} \overset{LH}{=} \lim_{N\to\infty} \frac{2}{1} = 2$

$2N+5 = O(N)$: is also true.

b) $0.01N = O(N^{0.99})$ False

$f(N) = 0.01N$, $g(N) = N^{0.99}$

$\lim_{N\to\infty} \frac{0.01N}{N^{0.99}} \overset{LH}{=} \lim_{n\to\infty} \frac{0.01}{0.99N^{-.01}} = \lim_{n\to\infty} \frac{0.01N^{0.01}}{0.99} = \infty$

Limit goes to infinity, statement is false!

c) $2^N = O(2^{N/2})$ False

$f(N) = 2^N$, $g(N) = 2^{N/2}$

$\lim_{N\to\infty} \frac{2^N}{2^{N/2}} = \lim_{N\to\infty} 2^{N/2}$

$= 2^{\lim_{N\to\infty} \frac{N}{2}} = \infty$

d) $\ln(N) = O(\sqrt{N})$ True

$f(N) = \ln N$, $g(N) = \sqrt{N}$

$\lim_{N\to\infty} \frac{\ln N}{\sqrt{N}} \overset{LH}{=} \lim_{N\to\infty} \frac{N^{-1}}{\frac{1}{2}N^{-1/2}}$

$= \lim_{N\to\infty} \frac{2\sqrt{N}}{N} \overset{LH}{=} \lim_{N\to\infty} \frac{N^{-1/2}}{1} = 0$

While $O(\sqrt{N})$ is true, it is more correct to say $\ln(N) = o(\sqrt{N})$, little-o.

e) $N\ln^2(N^2) = O(N^2 \ln N)$ True

$f(N) = N\ln^2(N^2)$, $g(N) = N^2 \ln N$

$\lim_{N\to\infty} \frac{N\ln^2(N^2)}{N^2 \ln(N)} = \lim_{N\to\infty} \frac{\ln^2(N^2)}{N\ln(N)} = \lim_{N\to\infty} \frac{2\ln^2(N)}{N\ln(N)}$

$= \lim_{N\to\infty} \frac{2\ln(N)}{N} \overset{LH}{=} \lim_{N\to\infty} \frac{N^{-1}}{1} = 0$

$N\ln^2(N^2) = o(N^2 \ln N)$ is more correct.

2.

c)

```
public static void func(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            System.out.println(i + " " + j);
        }
    }
}
```

$O(n^2)$

- Outer loop runs n times
- Inner loop runs n times per outer loop iteration
- Total iterations: $n \times n = n^2$

b)

```
public static void func(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            System.out.println(i + " " + j);
        }
    }
}
```

- When $i = 0$ → inner loop runs n times
- When $i = 1$ → inner loop runs $(n-1)$ times
- ⋮
- When $i = n-1$ → inner loop runs $n-(n-1) = 1$ times

$n + (n-1) + (n-2) + \dots + 1 = ?$

$$\sum_{1}^{n} n = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$$

$O(n^2)$

$$\lim_{n \to \infty} \frac{\frac{1}{2}n^2 + \frac{1}{2}n}{n^2} = \frac{1}{2}$$ ✓

c)

```
public static void func(int n) {
    for (int i = 0; i < n; i++) {
        if (i == 0) {
            for (int j = 0; j < n; j++) {
                System.out.println(i + " " + j);
            }
        }
    }
}
```

- When $i = 0$ → inner loop runs n times
- When $i > 0$ → inner loop doesn't run

$n + n = 2n$ (iterations of inner loop)

↑ Checks for condition

$O(n)$

$$\lim_{n \to \infty} \frac{2n}{n} = 2$$ ✓

3.

```
public static int sqrt(int x, int low, int high) {
    if (low > high || x < 0 || low < 0 || high < 0)
        return -1;
    int p = low + (high - low) / 2;
    if (p * p == x)
        return p;
    else if (p * p > x)
        return sqrt(x, low, p - 1);
    else
        return sqrt(x, p + 1, high);
}
```

Note
$\log(n) = \log_{10}(n)$

a) Ideas for Substituten

$d1 = high - low$

$p = low + \frac{d1}{2}$

- if $p^2 > x \rightarrow high = p-1$
  - ↳ $d2 = (p-1) - low$
- if $p^2 < x \rightarrow low = p+1$
  - ↳ $d2 = high - (p+1)$

☆ p: the midpoint of [low, high]

↳ given as $p = low + \frac{d1}{2}$

- Substitute for $d2 = (p-1) - low$ $p^2 > x$

$d2 = (low + \frac{d1}{2} - 1) - low = \frac{d1}{2} - 1$

- Substitute for $d2 = high - (p+1)$ $p^2 < x$

$d2 = high - (low + \frac{d1}{2} + 1)$

$= high - low - \frac{d1}{2} - 1$

$= d1 - \frac{d1}{2} - 1 = \frac{d1}{2} - 1$

→ In Both Cases $d2 = \frac{d1}{2} - 1$

$\lim_{d1 \to \infty} \frac{d1}{\frac{d1}{2} - 1} \overset{LH}{=} \lim_{d1 \to \infty} \frac{1}{\frac{1}{2}} = 2$ ∴ $\lim_{d1 \to \infty} \frac{d1}{d2} = 2$ must be true!

b) $high - low \approx a^r$; $a = 2$ and r = max number of recursive calls

$high - low = 2^r$

$\log(h-l) = \log(2^r)$

$\log(h-l) = r\log(2)$ ∴ $r = \frac{\log(high - low)}{\log(2)}$

c) Max calls from (b): $r = \frac{\log(h-l)}{\log 2}$, and each call is $O(1)$

We can say $O(r) * O(1) = O(r)$ for this code.

$O\left(\frac{\log(h-l)}{\log 2}\right)$ —Drop Constant Term→ $O(\log(high - low))$

d) Local Variable 'p' and inputs 'x', 'high', and 'low' only take up constant space $O(1)$. The call stack grows logarithmically based on # of recursive calls, whose max is 'r' from (b). Therefore the space complexity is $O(r) * O(1) = O\left(\frac{\log(high-low)}{\log 2}\right)$

Dropping Constants yields $O(\log(high - low))$ Time = space complexity in this scenario!!

4.

a)

```
public static void func(int n) {
    for (int i = n; i > 0; i /= 3)
        System.out.println(i);
}
```

$n = a^r$, n is input and a is the dividing factor of the set.

$i /= 3 \Rightarrow a = 3$

$n = 3^r$

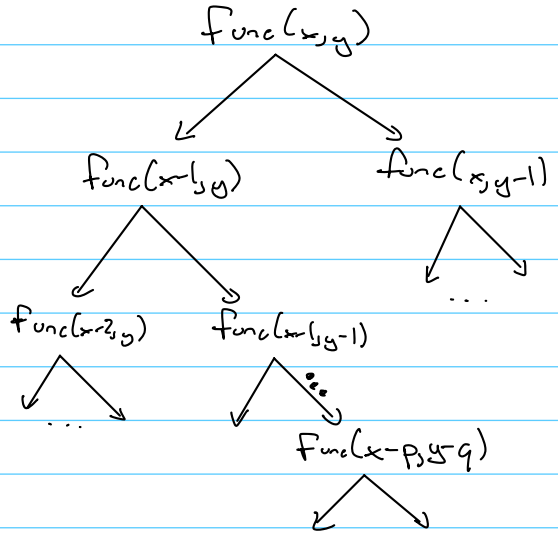
$\log(n) = \log(3^r)$

$\log(n) = r\log(3)$ $\therefore r = \frac{\log(n)}{\log(3)}$, with each iteration performing a constant time operation.

$O\left(\frac{\log(n)}{\log(3)}\right)$ which can be generalized as $O(\log(n))$

b)

```
public static int func(int x, int y) {
    if (x <= 1 || y <= 1)
        return 1;
    return func(x - 1, y) + func(x, y - 1);
}
```



Binary Call Tree Created, which goes down from (x,y) to $(1,y)$ or $(x,1)$

$n = 2^{(x+y)} \Rightarrow O(2^{(x+y)})$

a = total # of recursive calls from a single function call

r = # of total levels the worst case scenario

n = # of function calls made
↳ finding n will provide time complexity

$a = 2$; Each call splits to two recursive calls

$r = x+y$; In the worst case, both branches need to go to 1, so x only may need to be brought to 1 leading to an approximate tree height of $x+y$.