CS325

Assignment 2

April 12 2015

Neale Ratzlaff

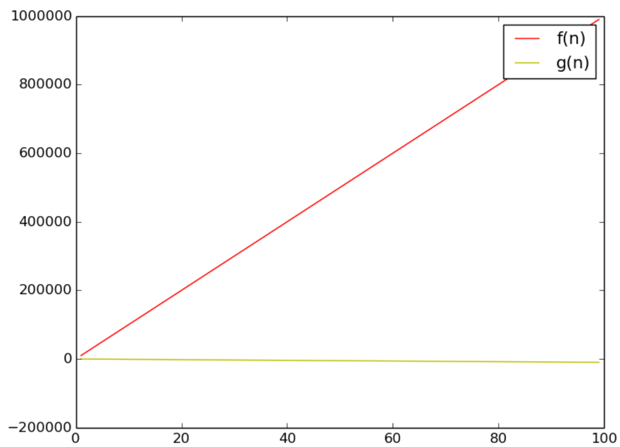
1)

a)

$f(n) = 10000n \quad g(n) = .00001n^2 - 100n$

f(n) is Ω g(n).

The two functions are close to equal when $n = 10^{10}$. But the n^2 term in g(n) dominates Despite the graph shown below, g(n) overtakes f(n) at $n > 10^{10}$

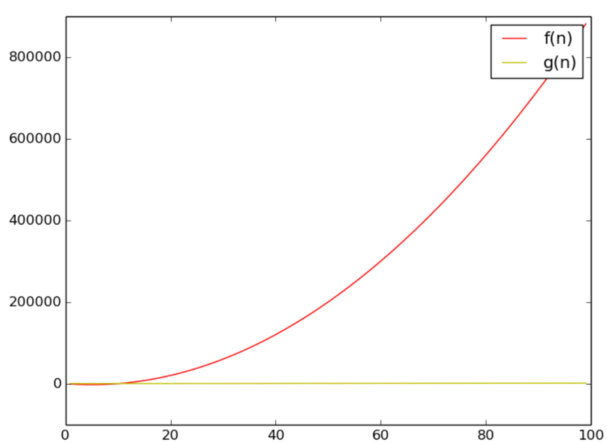


b)

$f(n) = 100n^2, \quad g(n) = 0.01n^2 + 10n + 5$

f(n) = Θ g(n)

Since both functions share an n^2 term, the coefficient then determines which function will grow faster. f(n) becomes theta of g(n), because there are constants such that f(n) will be a upper or lower bound of g(n)

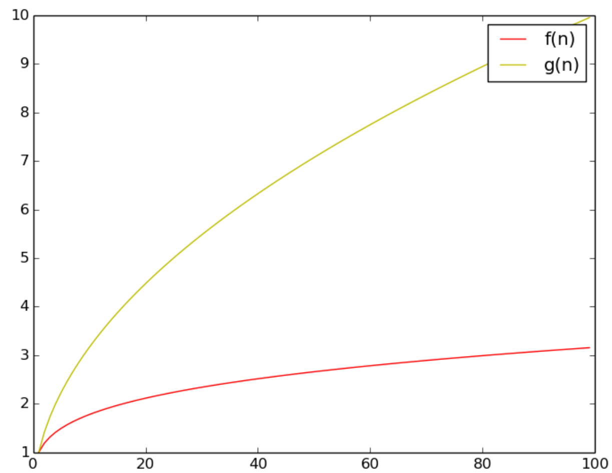


c)

$f(n) = n^{.25}$, $g(n) = n^{.5}$

f(n) = O g(n)

g(n) has a higher exponential term than f(n), (i.e. .25 < .5) so g(n) grows faster

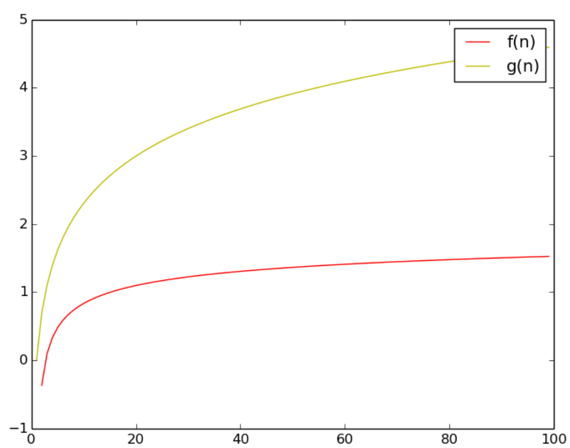


d)

f(n) = log(log(n)), $g(n) = \log^2(n)$

f(n) = O g(n)

f(n) is always less than g(n). log of a log grows far slower than $\log^2$

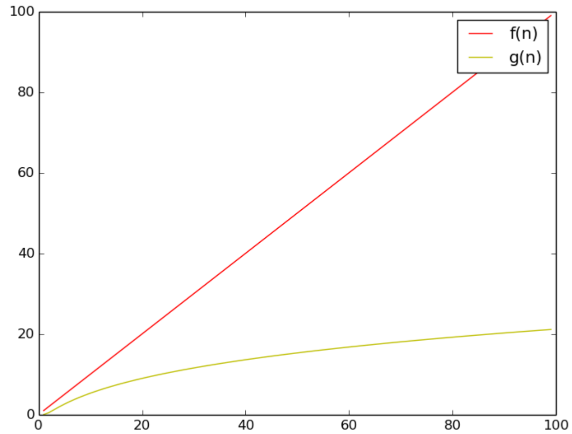


e)

f(n) = n, $g(n) = \log^2(n)$

f(n) = O g(n)

n grows far faster than a log(n). Since a log is a reverse exponentiation, the log of a number will always be less than its square root, making is less than a linear function even when the log is squared

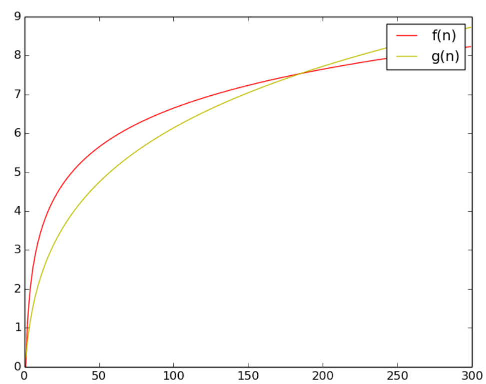


f)

f(n) = lg(n), g(n) = $\log^3(3n)$

f(n) = Ω g(n)

f(n) will grow slower than g(n) because 2^n will grow slower than 10^n. log base 2 will need a higher number to equal the same amount reached by log base 10.

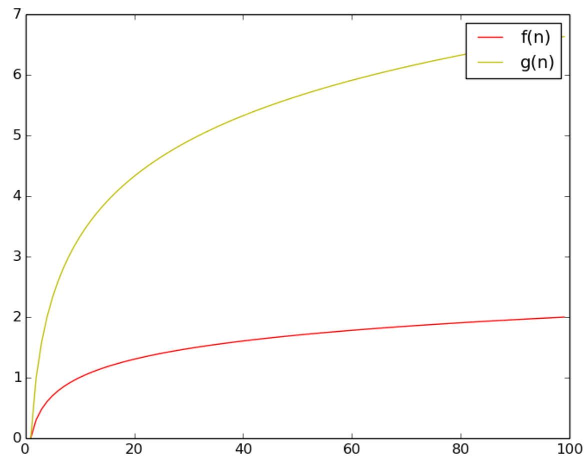


g)

f(n) = log(n), g(n) = lg(n)

f(n) = O g(n)

f(n) grows slower than g(n) due to the fact that $\log_{10}(n)$ is slower growing than $\log_2(n)$. lg(n) must exponentiate to a higher number n to equal the n that is operated on by $\log_{10}(n)$

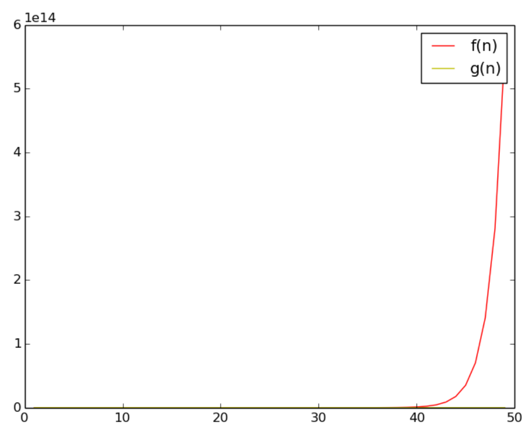


h)

f(n) = 2^n, g(n) = $10n^5$

f(n) = Ω g(n)

By far, f(n) grows faster than g(n). Any term to the nth power will grow faster than any constantly exponentiated value.

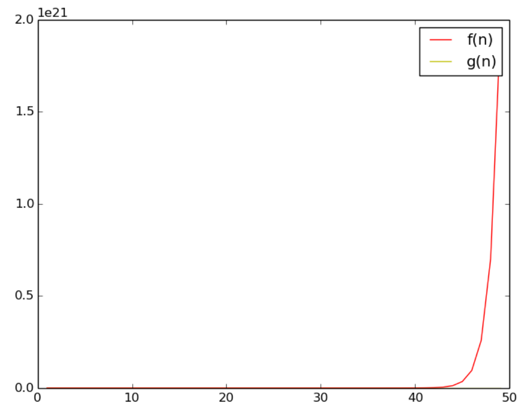


i)

f(n) = e^n, g(n) = 2^n

f(n) = Θ g(n)

Both f(n) and g(n) grow at the same rate but differ by the constant under the exponent. So f(n) can be multiplied by a c_1 and a c_2 that serve as upper and lower bounds respectively for g(n)

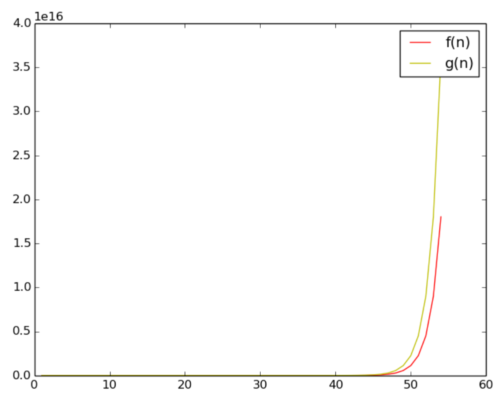


j)

$f(n) = 2^n$, $g(n) = 2^{n+1}$

f(n) = Θ g(n)

g(n) can be expressed as $2*2^n$, Making room for f(n) to be multiplied by a constant that would make f(n) an upper or lower bound on g(n).

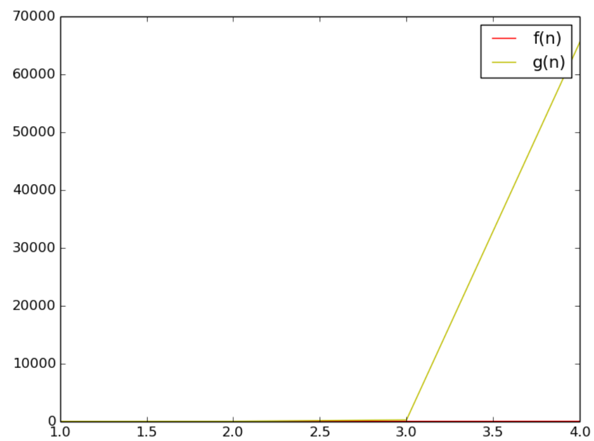


k)

$f(n) = 2^n$, $g(n) = 2^{(2^n)}$

f(n) = O g(n)

g(n) grows far faster than f(n) given the extra exponent term that f(n) doesn't have. g(n) is effectively equal to $2^{f(n)}$

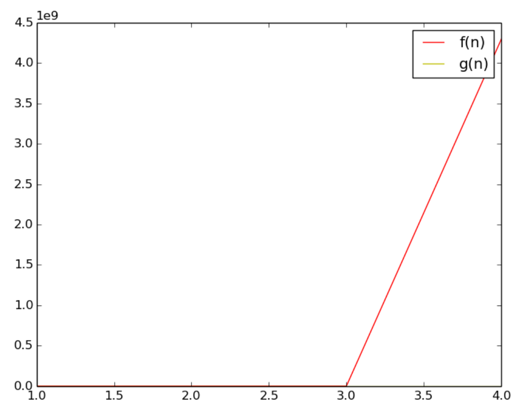


l)

f(n) = $n^{(2^n)}$, g(n) = 2^n

f(n) = Ω g(n)

f(n) will grow faster than g(n) because of the larger exponent

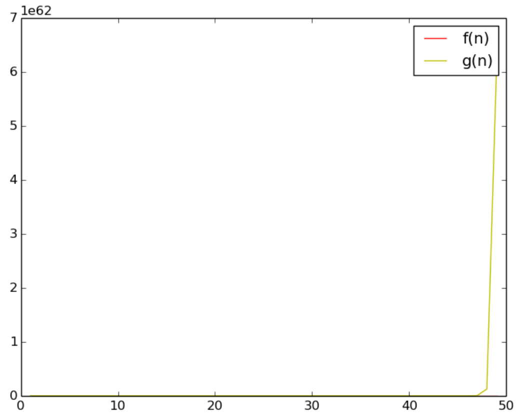


m)

f(n) = 2^n, g(n) = n!

f(n) = O g(n)

A factorial grows the fastest of any function other than n^n**.** So g(n) will be an upper bound of f(n)

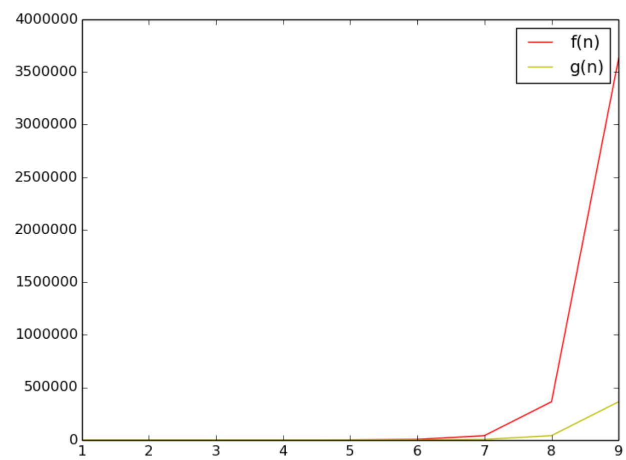


n)

f(n) = (n + 1)!,   g(n) = n!

f(n) = Ω g(n)

f(n) has a larger term being operated on by the factorial. If the added one wasn't present, f(n) would only be Θ g(n). But since the term being factorialized is (n+1), f(n) remains as O g(n)



2)

a) $f_1(n) = O\, f_2(n)$ implies $f_2(n) = O\, f_1(n)$

False

By transpose symmetry $f_1(n) = O\, f_2(n)$ **iff** $f_2(n) = \Omega\, f_1(n)$

b)

If $f_1(n) = O\, g_1(n)$ and $f_2(n) = O\, g_2(n)$ then $(f_1(n) * f_2(n)) = O\,(g_1(n) * g_2(n))$

True

If $f_1(n)f_2(n)$ is not O of $g_1(n)g_2(n)$, there will be some constants that will make g_1 and g_2 larger than f_1 and f_2

$f_1(n) \times f_2(n) = C_1(g_1(n)) \times C_2(g_2(n))$
$f_1(n) = O(g_1(n)) \rightarrow f_1(n) > C_1(g_1(n))$
$f_2(n) = O(g_2(n)) \rightarrow f_2(n) > C_2(g_2(n))$
$f_1(n) \times f_2(n) = O(C_1(g_1(n)) \times C_2(g_2(n)))$

$f_1(n)f_2(n) = O(g_1(n)g_2(n))$

c)

$\max(f_1(n), f_2(n)) = \Theta(f_1(n) + f_2(n))$

$f_1(n) \geq 0$
$f_2(n) \geq 0$
$(f_1(n) + f_2(n)) \geq \max(f_1, f_2)$
$(f_1(n) + f_2(n)) \geq C_1 * \max(f_1, f_2)$

$(f_1(n) + f_2(n)) \leq 2\max(f_1, f_2)$
$(f_1(n) + f_2(n)) \leq C_2 * \max(f_1, f_2)$

$\mathbf{\max(f_1(n), f_2(n)) = \Theta(f_1(n) + f_2(n))}$

3)

a)

Base case: n = 6
$F(6) \geq 2^3$
$13 > 8$

Inductive step: $F(n) \geq 2^{.5(n)}$ for all $n > 6$

$F(n+1) = F(n-1) + F(n)$
$F(n-1) + F(n) \geq 2^{.5(n+1)}$
$F(n-1) + F(n) \geq 2^{.5(n)} * 2^{.5} \rightarrow 2^{.5(n)} * 2^{.5} \geq 1$
This inequality is true if $n > 6$

b)

Using C = .75
Base case: n = 1 with C = .75
$F(1) \leq 2^{.75}$

Inductive step: assume F(n) $\leq 2^{.75(n)}$ for all n
Attempt for n = n + 1:

F(n+1) = F(n-1) + F(n)
F(n-1) + F(n) $\leq 2^{.75(n+1)}$
F(n-1) + F(n) $\leq 2^{.75(n)} * 2^{.75} \rightarrow 2^{.75(n)} * 2^{.75} \geq 1$
This inequality is true if n $\geq$ 1

c)
The largest number for which the fibonacci sequence is the lower bound is given by C = the golden ratio ϕ

4)
The golden ratio and its conjugate are defined as $\frac{1+\sqrt{5}}{2}$ *and* $\frac{1-\sqrt{5}}{2}$ respectively.
With these ratios being the roots that satisfy the equation defined below
The golden ratio is described as for a > b > 0, a + b is to a as a is to b

$\phi = \frac{a+b}{a} = \frac{a}{b}$
substitute $\phi = \frac{b}{a}$
$\frac{a+b}{a} = 1 + \frac{b}{a} = 1 + \frac{1}{\phi}$
so $1 + \frac{1}{\phi} = \phi \rightarrow \phi + 1 = \phi^2$

The conjugate Φ is proven the same way, since Φ is just the second root found when the quadratic formula is applied to equation $\phi^2 + \phi + 1$

5)

a) Python scripts to compute BC1 for n terms

##BC1.py

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.misc
import sys

def BC1(n, k):
if k == 0: return 0
if k == n: return 1
return BC1(n-1, k) + BC1(n-1, k-1)
n = float(sys.argv[1])
k = int(n/2)
```

```
print BC1(n, k)
```

##BC2.py

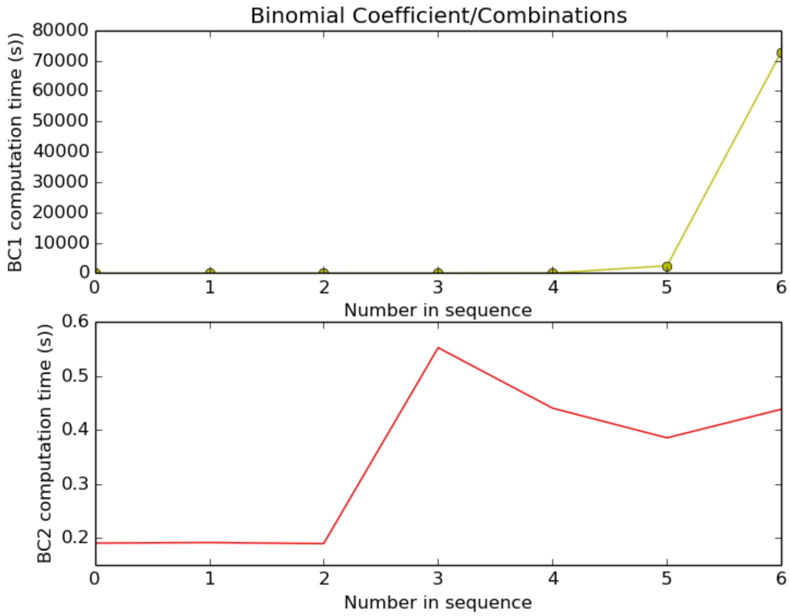
```
import matplotlib.pyplot as plt
import numpy as np
import scipy.misc
import sys

def BC2(n, k):
        if k == 0: return 1
        if k > 0: return BC2(n-1, k-1) * (n/k)
        else: return "need k > 0"

n = float(sys.argv[1])
k = int(n/2)
print BC2(n, k)
```

b)

n	5	10	15	20	30	35	40
BC1 (s)	.191	.197	.195	.263	60.77		72900
BC2 (s)	.190	.191	.189	.191	.552	.385	.438

*** n = 30, 35 and 40 executed on an external amazon EC2 instance with slightly less computational power**

c)



For n = 5, 10, 15, 20, 30, 35, 40

The BC2 algorithm is far faster than the BC1 algorithm. With n > 30, BC2 is faster by at least 1200%. BC2 runs faster simply because the recursion tree only occurs once, as BC1 has to recursive calls per iteration of the function.

pyplot plotting script:

```
#plot.py
import matplotlib.pyplot as plt
import numpy as np
## 5, 10, 15, 20, 25, 30, 35, 40
## assignment 2
x = np.array([.191, .197, .195, .263, 60.77, 2430, 72900])
y = np.array([.190, .191, .189, .552, .440, .385, .438])

plt.subplot(2, 1, 1)
plt.title("Binomial Coefficient/Combinations")
plt.plot(x, "-yo")
plt.xlabel("Number in sequence")
plt.ylabel("BC1 computation time (s))")

plt.subplot(2, 1, 2)
plt.plot(y, "-r")
plt.xlabel("Number in sequence")
plt.ylabel("BC2 computation time (s))")
plt.show()
```