Growth Functions and Asymptotic Analysis

Instructor: Krishna Venkatasubramanian

CSC 212

Announcements

- **Go to office hours.**
- Next Quiz ---- September 24
  - Covers material from Sept 12, 17, and 19
- Same format as today's quiz about 20 minutes long.

Running Time of Algorithms

- Running time of algorithm determines how **“quickly”** it executes.
- **Computed based on # of basic steps** in the algorithm that are executed
  - Loops result in repeated computational sets leading to larger running time than those without
- **Size of the inputs can also affect number of basic steps**
  - Sorting longer arrays need more time than shorter ones!
  - In such cases**, size of input usually dictates # of basic steps!**

Runtime affected by # of Basic Steps

- Two algorithms for performing the same tasks can have different running times **depending upon # of steps it has**
- Here, the size of the input is the same --- **1 number** --- but the presence of loops dictates running time.

```python
import time

def SumOfN(n):
    start = time.time()
    theSUM = 0

    for i in range(1,n+1):
        theSUM +=i

    end = time.time()

    return theSUM, end-start


def directSumOfN(n):
    start = time.time()
    theSUM = (n*(n+1))/2
    end = time.time()

    return theSUM,end-start


def main():

    print("Sum of N with for loop")
    for i in range(5):
        print("Sum is %d required %10.7f seconds"%SumOfN(1000000))

    print("Sum of N function direct ")
    for i in range(5):
        print("Sum is %d required %10.7f seconds"%directSumOfN(1000000))
```

w/ FOR LOOP

DIRECT SUMMATION

```
Sum of N with for loop
Sum is 500000500000 required  0.0627120 seconds
Sum is 500000500000 required  0.0636330 seconds
Sum is 500000500000 required  0.0593448 seconds
Sum is 500000500000 required  0.0563250 seconds
Sum is 500000500000 required  0.0615969 seconds
Sum of N function direct
Sum is 500000500000 required  0.0000012 seconds
Sum is 500000500000 required  0.0000000 seconds
Sum is 500000500000 required  0.0000000 seconds
Sum is 500000500000 required  0.0000012 seconds
Sum is 500000500000 required  0.0000007 seconds
```

Runtime affected by input size: Insertion Sort (Recap)

Assume n elements

```
def InsertionSort(A)
    for k in range(1,len(A))

        key = A[k]

        i = k -1

        while i > 0  and   A[i] > key

            A[i+1] = A[i]

            i = i - 1

        A[i+1] = key
```

	Cost	Times
for k in range(1,len(A))	c1	n
key = A[k]	c2	n-1
i = k -1	c4	n -1
while i > 0 and A[i] > key	c5	$\sum_2^n t_j$
A[i+1] = A[i]	c6	$\sum_2^n (t_j -1)$
i = i - 1	c7	$\sum_2^n (t_j -1)$
A[i+1] = key	c8	n-1

WHY?

$$T(n) = c1 * n + c2(n-1) + c4(n-1) + c5\sum_2^n t_j + c6\sum_2^n (t_j \ -1) \ + c7\sum_2^n (t_j \ -1) + c8(n-1)$$

Asymptotic Running Time

- Given the Running Time (T(n) = f(n)) equation for an algorithm,
  - it is useful to see how "fast" the algorithm executes for large input sizes
- This is called **Asymptotic Running Time**
  - How the running time increases as the size of the input increases without bound (also called *in the limit*)
- Usually, **algorithms with better asymptotic running time are better ("faster")** but for all but very small inputs
- The **notation** we use to describe **Asymptotic Running Time are $\boldsymbol{\Theta}$, $\boldsymbol{O}$, $\boldsymbol{\Omega}$, …**

Θ-notation ("Big –Theta")

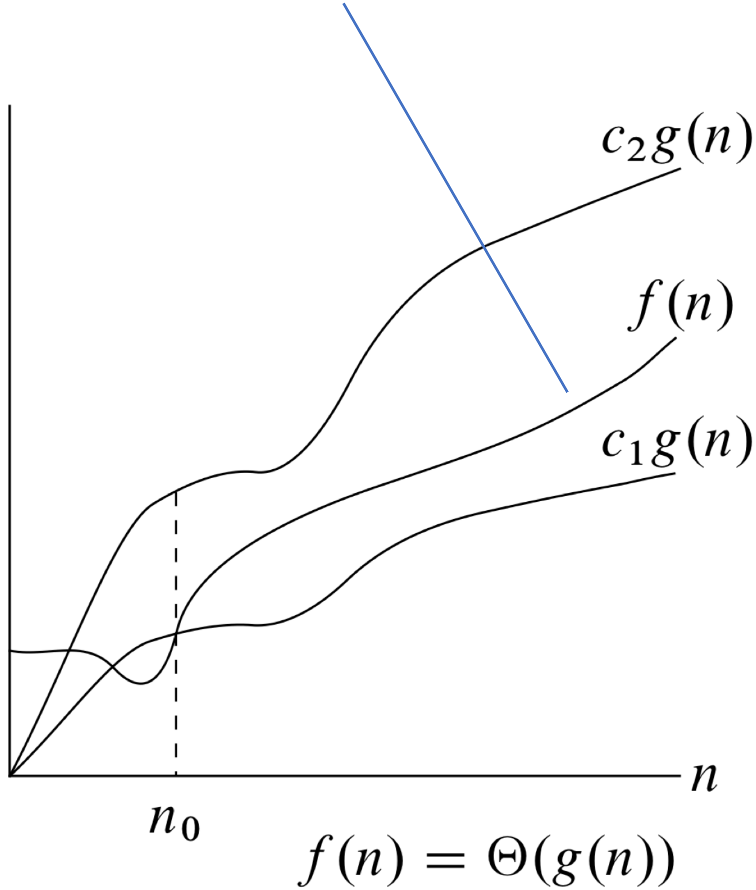
- **Definition:** For a given function *g(n)*, $\Theta(g(n))$ is a **set of functions** such that
  - $\Theta(g(n))$ = { *f(n):* there exists positive constants c1, c2, and n0 s.t. 0≤ $c1g(n) \leq f(n) \leq c2g(n)$ for all n $\geq n0$ }

- This is called an **asymptotic tight-bound** for f(n)
  - Really, f(n) ∈ $\Theta\left(g(n)\right)$

- For all values of n $\geq n0$ the value of f(n) = g(n) within a constant factor

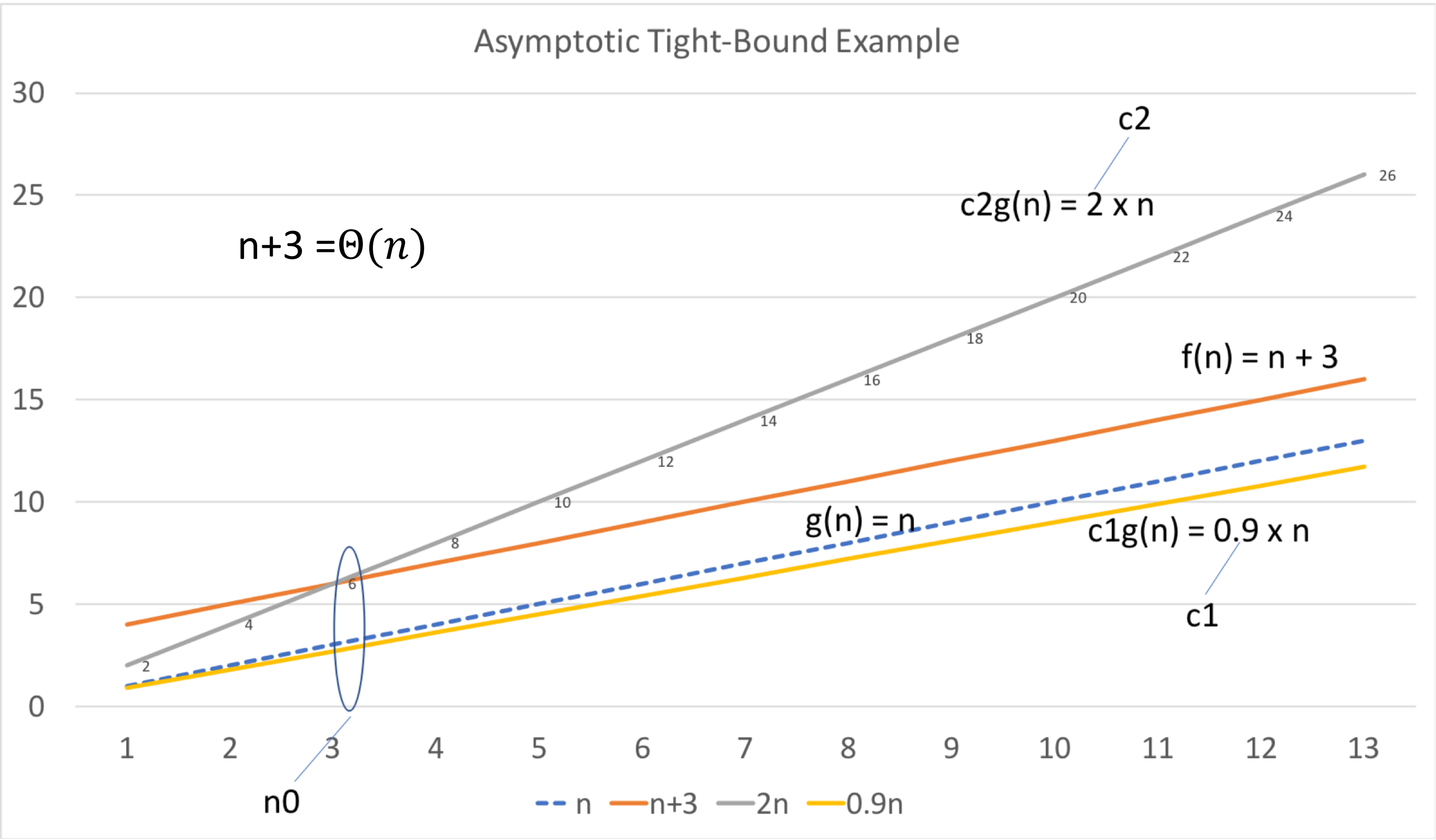
- Focus on **large values of n**

Algorithm's Running Time
(e.g., calculated as we did for insertion sort)



$f(n) = \Theta(g(n))$

Θ-notation: Graphical Example



Θ-notation: Numerical Example

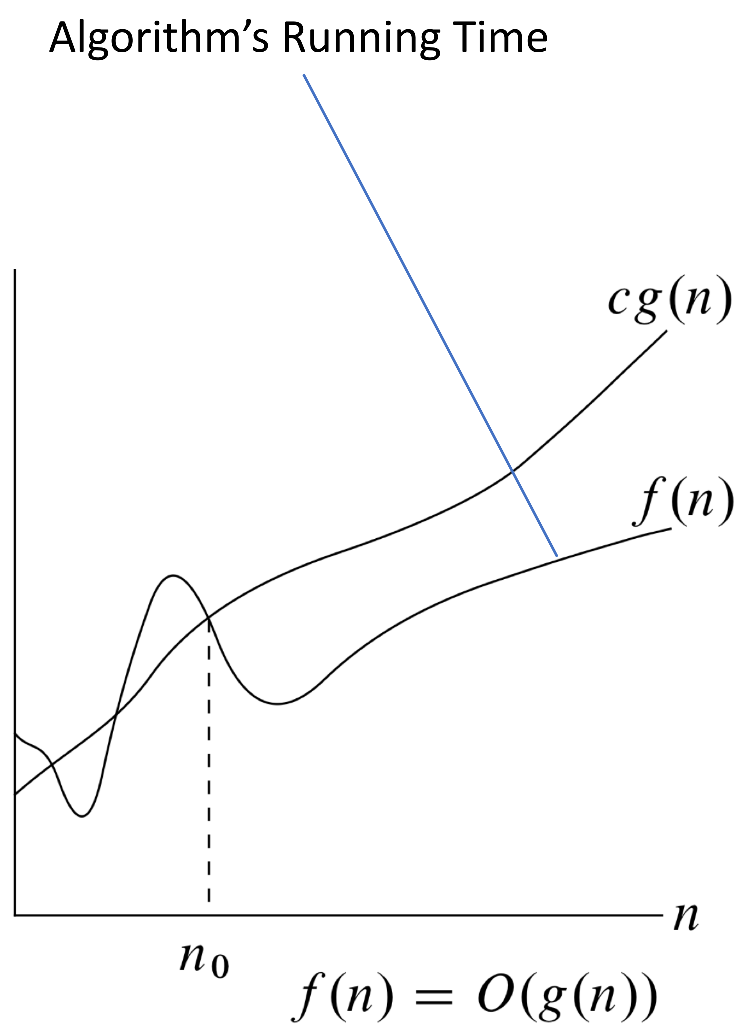
- Assume $f(n) = 1/2n^2 – 3n$
- We say **$f(n) = \Theta(n^2)$**, if this is true then
  - $c1n^2 <= 1/2n^2 – 3n <= c2n^2$
  - **$c1 <= 1/2 - 3/n <= c2$**

Remember:
c1,c2 and n are positive constants

- The right inequality is true for n>=1 and c2 >= ½
- The left inequality is true for n>=7 and c1 <= 1/14
- Thus if we choose
  - c1 = 1/14, c2 = ½, and n0 = 7 we can make the inequality true
- Thus $f(n) = \Theta(n^2)$
- **Note, other c1, c2, and n0 may also exist that make the inequality true**
- Suffice it to say, that we can find one groups of values

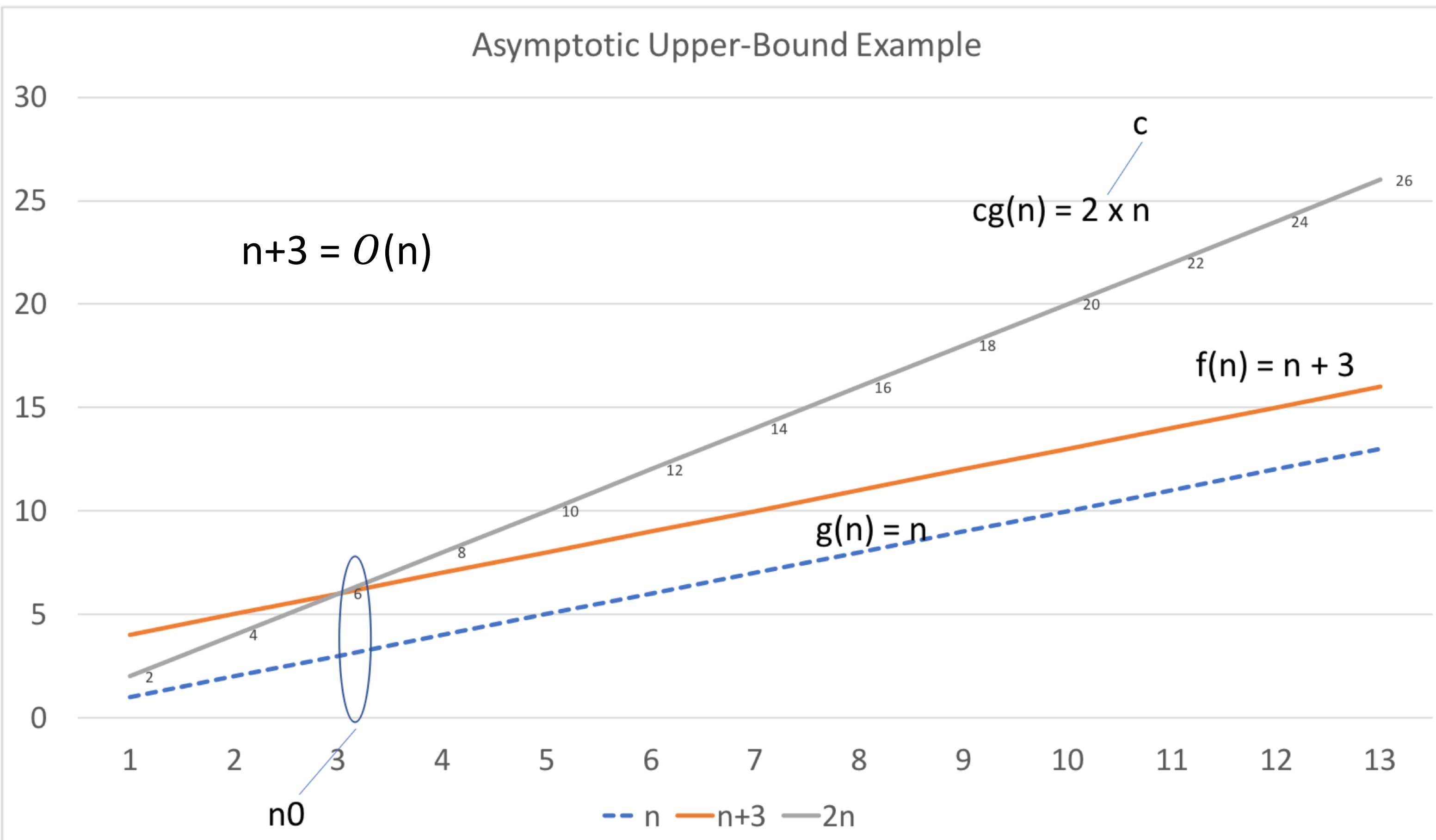
O-notation ("Big –O")

- **Definition:** For a given function *g(n)*, $O(g(n))$ is a **set of functions** such that
  - $O(g(n))$ = { *f(n):* there exists positive constants c, and n0 s.t. $0 \leq f(n) \leq cg(n)$ for all n $\geq n0$ }
- This is called an **asymptotic upper-bound** for f(n)
- For all values of n $\geq n0$ the value of f(n) is always <= g(n) within a constant factor
- Focus on **large values of n**



$f(n) = O(g(n))$

O-notation: Graphical Example



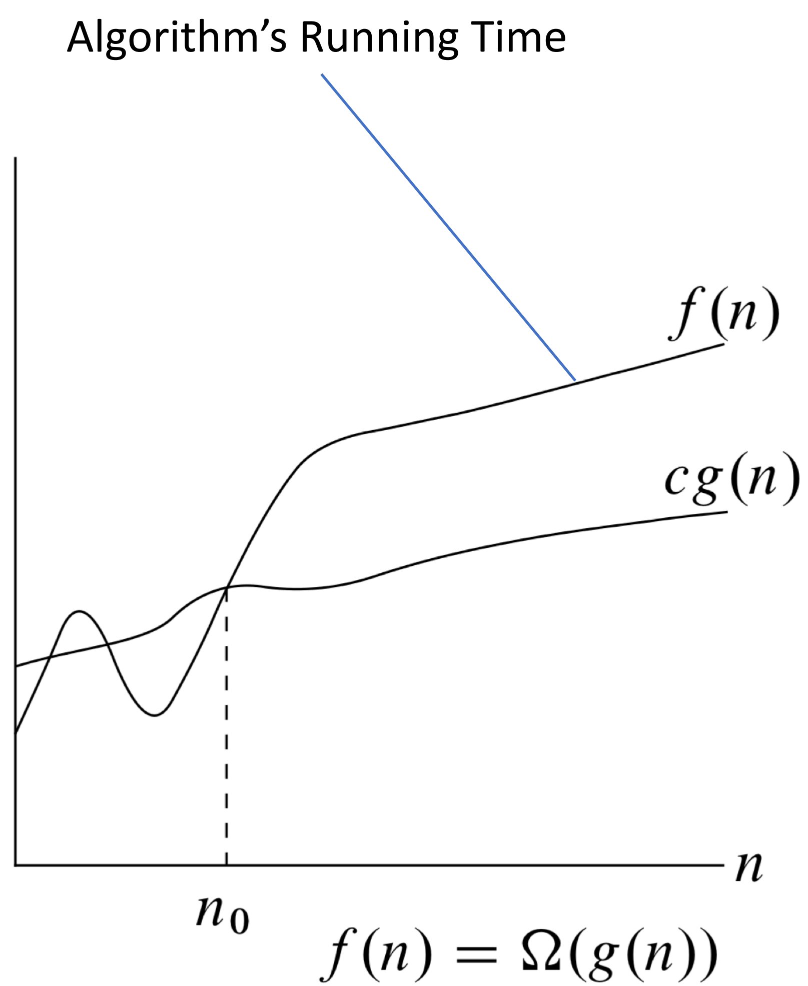
O-notation: Numerical Example

Remember:
c and n are positive constants

- Assume $f(n) = 1/2n^2 - 3n$
- We say f(n) = $O(n^2)$, if this is true then
  - $0 <= 1/2n^2 - 3n <= cn^2$
  - **$0 <= 1/2 - 3/n <= c$**
- The right inequality is true for n>=1 and c >= 1/2
- The left inequality is true for n >=4
- Thus if we choose
  - c = ½, and n0 = 4 we can make the inequality true
- Thus **f(n) = $O(n^2)$**

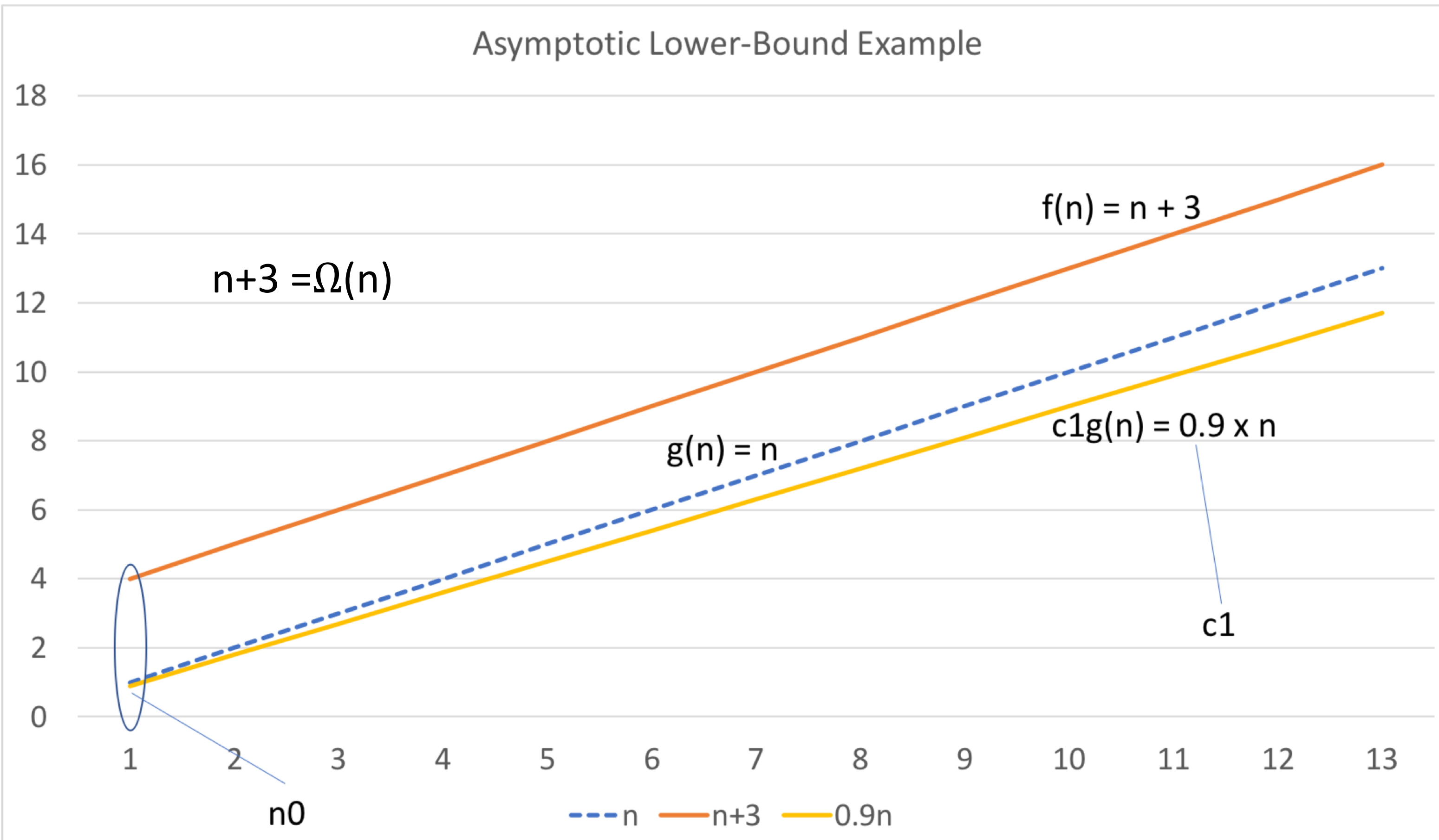
Ω-notation ("Big –Omega")

- **Definition:** For a given function *g(n)*, $\Omega(g(n)$ is a **set of functions** such that
  - $\Theta(g(n))$ = { *f(n):* there exists positive constants c, and n0 s.t. $0 \leq cg(n) \leq f(n)$ for all n $\geq n0$ }
- This is called an **asymptotic lower-bound** for f(n)
- For all values of n $\geq n0$ the value of g(n) is always <= f(n) within a constant factor
- Again, focus on **large values of n**



$f(n) = \Omega(g(n))$

Ω-notation: Graphical Example



Theta, O and Omega

$$\text{If } f(n) = \Theta(g(n))$$

Then

$$f(n) = O(g(n))$$

&

$$f(n) = \Omega(g(n))$$

Coming up with Asymptotic Running Time

- We will mostly talk of Running Time of an algorithm in terms of the **Big-O notation** (unless otherwise specified) --- **WORST CASE RUNNING TIME**

- To calculate the Big-O running time of an algorithm**, lower order terms are ignored**.
  - E.g., For T(n) = an^2 + bn + c, the **value of n^2 will dominate** values of *b*n* or *c* or *a* for large values of n
  - Therefore, here **T(n) = O(n^2)**

- More generally, for any polynomial

  p(n) = $\sum_{i=0}^{d} a_i \, n^i$ where a_i is a constant and $a_d > 0$,

  p(n) = $\boldsymbol{O}$(n^d)

- Similarly, for a zero—degree polynomial *q(n)* or a constant function --- e.g., a given algorithm step

q(n) = $\boldsymbol{O}$(n^0) = $\boldsymbol{O}$(1)

Note: Correct but Meaningless

Could you say?

$3n^2 + 2 = O(n^6)$ or $3n^2 + 2 = O(n^7)$

$O(n^2)$

is a tighter asymptotic upper bound

Yes, but this is like answering:

- **What is the world's record for running one mile?**
  - **Less than 3 days.**
- **How long does it take to drive from here to Chicago?**
  - **Less than 11 years.**

Do not get confused: O-Notation

O(1) or “Order One”

- **DOES NOT mean that it takes only one operation**
- **DOES mean that the work doesn’t change as N changes**
- **Is notation for “constant work”**

O(N) or “Order N”

- **DOES NOT mean that it takes N operations**
- **DOES mean that the work changes in a way that is proportional to N**
- **Is a notation for “work grows at a linear rate”**

Running Time for Insertion Sort

- FASTEST CASE – sorted array input

$$\boxed{T(n) = c1 * n + c2(n-1) + c4(n-1) + \text{c5 (n}-1) + \text{c8(n}-1)}$$

  - T(n) = An + B = **O(n)**
  - Where A = c1+c2+c4+c5+c8 and B = -(c2+c4+c5+c8)

- SLOWEST CASE – inverse sorted array input

$$\boxed{T(n) = c1 * n + c2(n-1) + c4(n-1) + c5\frac{n(n+1)}{2} + c6\frac{n(n-1)}{2} + c7\frac{n(n-1)}{2} + c8\ (n\text{-}1)}$$

  - T(n) = $An^2 + Bn + C$ = **$O(n^2)$**
  - What are A, B, and C ???

Terminology Note

We refer to Big-Theta, Big-O, Big-Omega as

Running Time

=

Time Complexity

=

Computational Complexity

Comparing Algorithms

- **We will use O-notation from now on to describe algorithm running time.**

- **We can compare different algorithms that solve the same problem:**

  1. **Determine the O(.) for the time complexity of each algorithm**
  2. **Compare them and see which has “better” performance**

Modular Analysis

- **Algorithms typically consist of a sequence of logical steps/sections/modules**

- **We need a way to analyze these more complex algorithms…**

- **It’s easy – analyze the sections and then combine them**

Sequential Steps

- **If steps appear sequentially (one after another), then add their respective O().**

```
loop
. . .
endloop
loop
. . .
endloop
```

N

M

$\longrightarrow O(N + M)$

Embedded Steps

- **If steps appear embedded (one inside another), then multiply their respective O().**

```
loop
   loop
   . . .
   endloop
endloop
```

M

N $\longrightarrow$ O(N*M)

Correctly Determining Big-O

- **Can have multiple factors** (variables that measure input size)
  - **$O(N*M)$**
  - **$O(\log P + N^2)$** $\rightarrow$ $O(N^2)$

- **But keep only the dominant factors:**
  - **$O(N + N\log N)$** $\rightarrow$ $O(N\log N)$
  - **$O(N*M + P)$** $\rightarrow$ $O(N*M)$
  - **$O(V^2 + V\log V)$** $\rightarrow$ $O(V^2)$
- **Drop constants:**
  - **$O(2N + 3N^2)$** $\rightarrow$ $O(N^2)$

Growth Functions

- Using O-notation, we are characterizing an algorithm's running time using a polynomial of some kind!
  - **$O(n)$** --- worst case sumOfN function (in slide 3)
  - **$O(n^2)$** --- worst case insertion sort

- One can have algorithms that have other running times as well such as
  - $O(2^n)$ --- worst case traveling salesman problem
  - $O(n\lg n)$ – worst case merge sort
  - ….

- So, which of the running times are ok, and which are not?

Puzzle

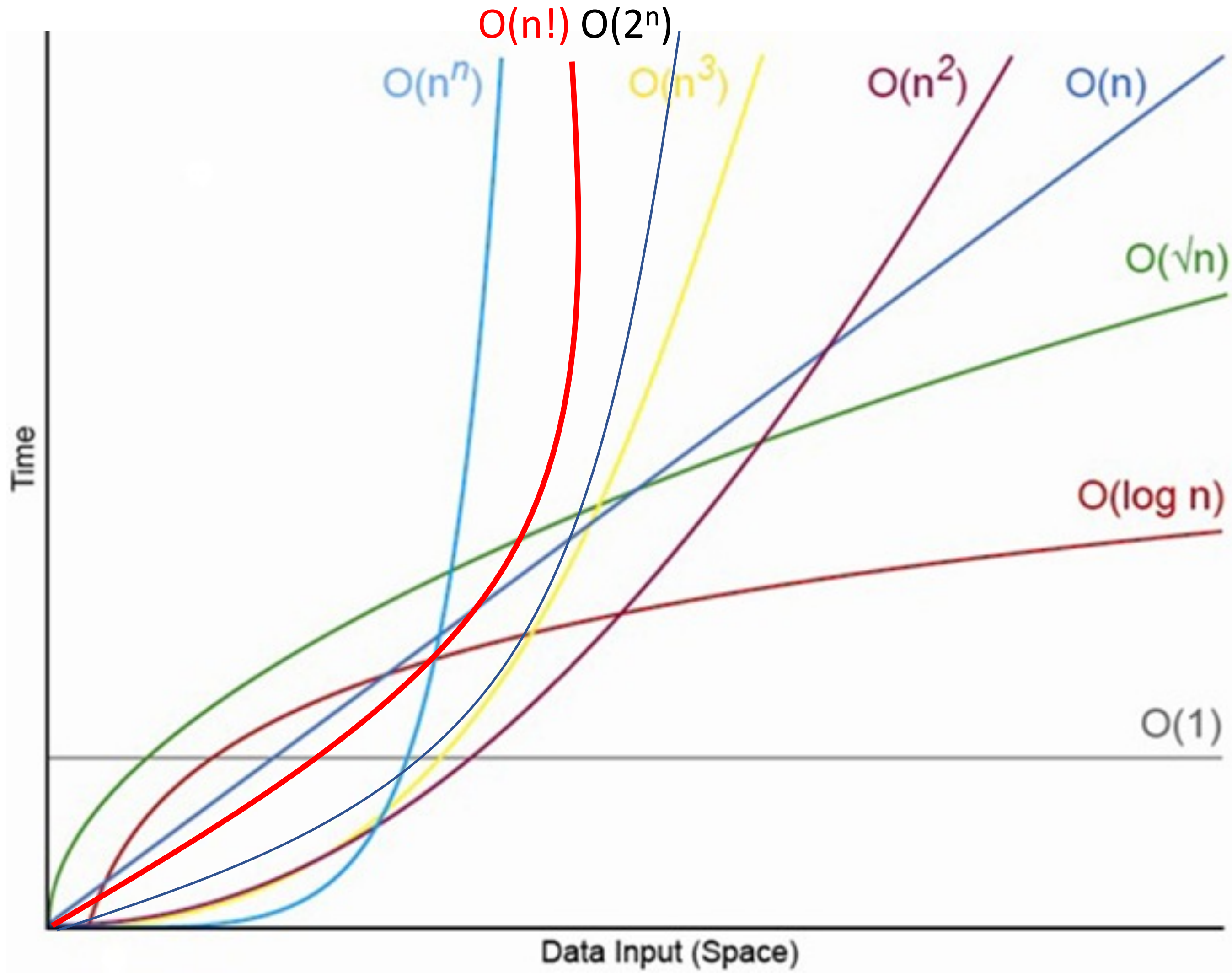
- Imagine a pond.
- Moss starts to grow on it and doubles in size every day.
- One the 10th day the moss fully covers the pond.

On what day, was the **pond half-covered** with moss?

Exponential Growth and Linear Growth are very different

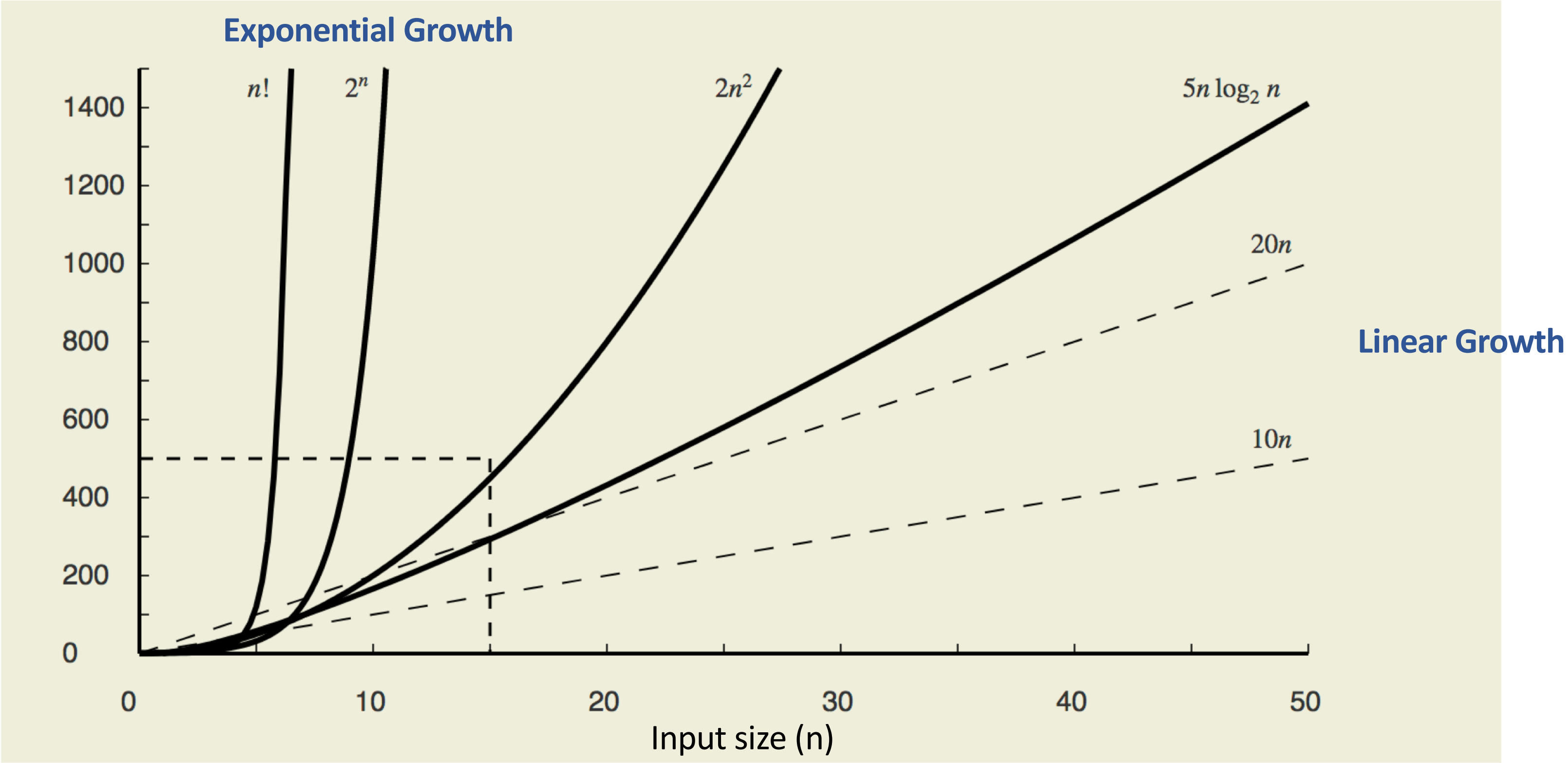
We are used to thinking in terms of linear functions, exponential functions behave differently

Asymptotic Growth Rates



Adapted from : https://github.com/sf-wdi-31/algorithm-complexity-and-big-o

More Asymptotic Growth Rates



Quiz

- You are given this set of functions:
- $n!$
- 2^n
- $2n^2$
- $5n\log n$
- $20n$
- $10n$

Organize them by ascending order of growth rate

$10n > 20n > 5n\log n > 2n^2 > 2^n > n!$

Growth Rates Table

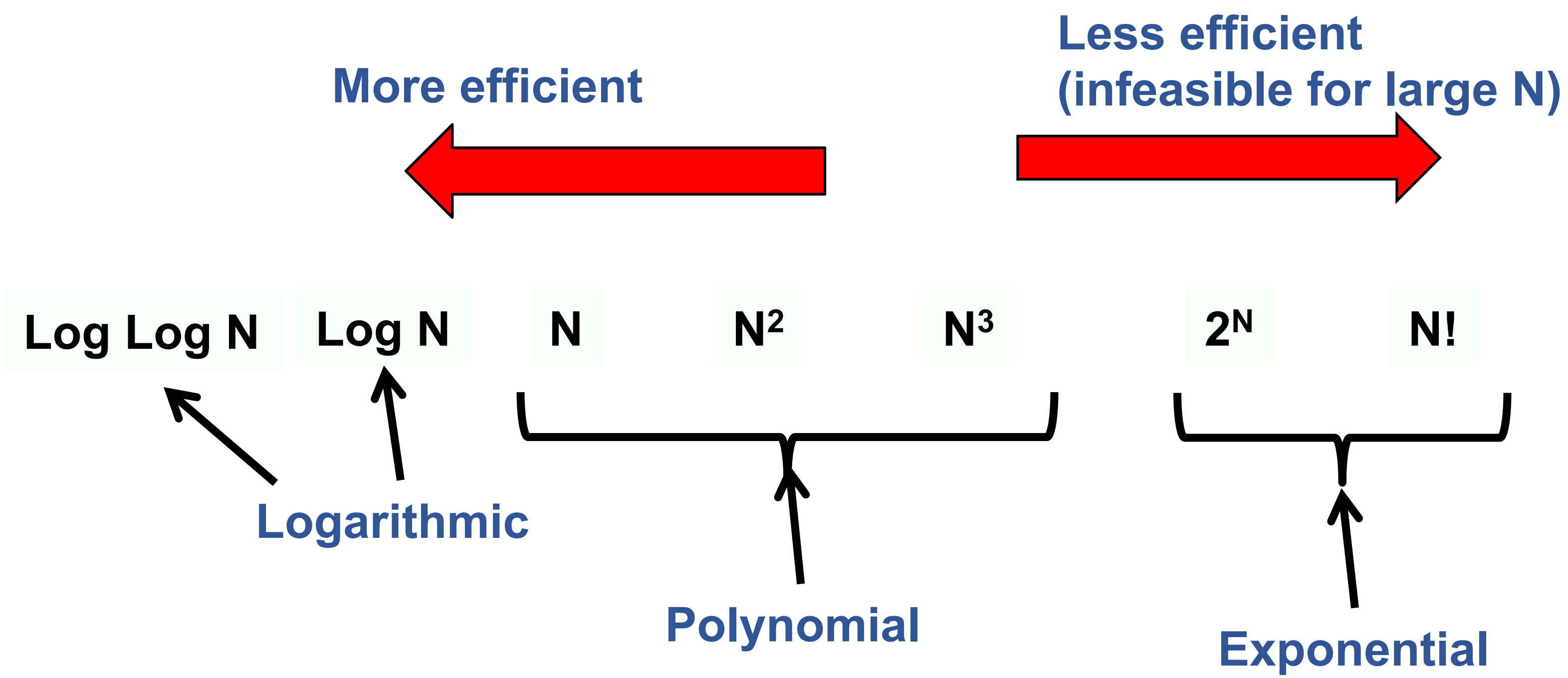
n	log log n	log n	n	n log n	n^2	n^3	2^n
16	2	4	2^4	$4 \cdot 2^4 = 2^6$	2^8	2^{12}	2^{16}
256	3	8	2^8	$8 \cdot 2^8 = 2^{11}$	2^{16}	2^{24}	2^{256}
1024	≈ 3.3	10	2^{10}	$10 \cdot 2^{10} \approx 2^{13}$	2^{20}	2^{30}	2^{1024}
64K	4	16	2^{16}	$16 \cdot 2^{16} = 2^{20}$	2^{32}	2^{48}	2^{64K}
1M	≈ 4.3	20	2^{20}	$20 \cdot 2^{20} \approx 2^{24}$	2^{40}	2^{60}	2^{1M}
1G	≈ 4.9	30	2^{30}	$30 \cdot 2^{30} \approx 2^{35}$	2^{60}	2^{90}	2^{1G}

More Growth Rate Tables

Size of Input	2^n	n^3	n^2	n	nlog2n	log2n
1	2	1	1	1	0	0
10	1024	1000	100	10	33.21928095	3.321928095
100	1.26765E+30	1000000	10000	100	664.385619	6.64385619
1000	1.0715E+301	1000000000	1000000	1000	9965.784285	9.965784285
10000	Are you crazy!	1E+12	100000000	10000	132877.1238	13.28771238
100000	Stop it!	1E+15	10000000000	100000	1660964.047	16.60964047
1000000	NO!	1E+18	1E+12	1000000	19931568.57	19.93156857
10000000	This is nuts!	1E+21	1E+14	10000000	232534966.6	23.25349666
100000000	I give up!	1E+24	1E+16	100000000	2657542476	26.57542476

I made this table for this class. You can too, play with a variety of functions to see which ones grow faster for large input sizes.

Order Of Growth



Practice

- Consider this program. What is it doing?

```
sum = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    sum++;
```

Input size = n

- What is the running time here?
  - The basic operation here is sum++ → can be done in constant time, say **c**
  - Ignore the operation sum =0 → it's so simple → the time take to do is **<< c**
  - So, it's really only about the two **for-loops** of size n

- $T(n) = O(n^2)$
  - For a given input size n, how many steps will be taken?

Practice: What is the O-notation Running Time for the following?

```
for i in range(n):
   for j in range(n):
      for k in range(n):
         k = 2+j+i
```

$O(n^3)$

```
i = n
while i > 0:
   k = 2+2
   i = i//2
```

O(lg n)

```
for i in range(n):
    k = k + i
```

O(n)

```
i = 2
i =i*i+2*(5^6)/(i*9)
```

O(1)

```
for i in range(n):
     k = k+i
 for j in range(n):
     k = k + j
 for k in range(n):
      k = k+ k
```

O(3n) = O(n)

Practice

- What is the Asymptotic Relationship (O or $\Theta - notation$) between

- n^k *in terms of* c^n (assuming $c > 1$ and $k > 1$)
- $\lg n^{\lg 17}$ *in terms of* $\lg 17^{\lg n}$
- $\log_2 n$ *in terms of* $\log_8 n$
- $3n\log_8 n$ *in terms of* $n^3 \lg n$


That's all Folks!
Any Question ?