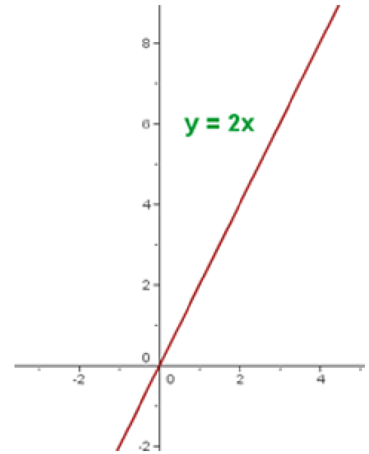
COS 212
Complexity Analysis

Algorithm Complexity

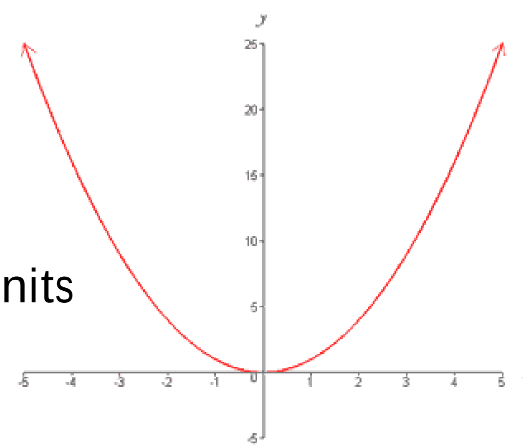
- The same problem can frequently be solved with different algorithms which differ in efficiency
- Some algorithms require more space than others
  - Technically, you could always buy more memory
- Some require more time
  - No easy solutions here: even the fastest CPU has a limit
  - Time is usually a more important constraint
- Algorithm time efficiency is measured in logical units rather than real-time units such as "milliseconds per byte of data"
- Algorithm time efficiency is determined by the relationship between the size n of data to be processed and the amount of time t required to process the data
- $t = f(n)$
- How do we determine $f(n)$? How do we interpret $f(n)$?

Interpreting f(n): Asymptotic Complexity

- Suppose you manage to work out the relationship f(n) between *t* and *n* (we'll talk about how to do it later)
- Algorithm A: $f(n) = 2n$
  - t is double n
  - the relationship is linear
  - for 10 data units, you'll need 20 time units
  - for 100 data units, you'll need 200 time units
  - for 1000 data units, you'll need 2000 time units



- Algorithm B: $f(n) = n^2$
  - t is n squared
  - the relationship is quadratic
  - for 10 data units, you'll need 100 time units
  - for 100 data units, you'll need 10,000 time units
  - for 1000 data units, you'll need 1,000,000 time units



- Which algorithm is more efficient? Which one is more complex?

Asymptotic Complexity

- Consider:
- Algorithm C: $f(n) = 2n + n^2$
  - Is it quadratic or linear?
  - Which term contributes more?
  - for 10 data units, you'll need 120 time units
  - for 100 data units, you'll need 10,200 time units
  - for 1000 data units, you'll need 1,002,000 time units
  - n^2 dominates $2n$
- Which algorithm is more efficient: A ($2n$) or C?
  - A
- Which algorithm is more efficient: B (n^2) or C?
  - Treat as equals
- Why: because we only care about asymptotic complexity, i.e. an approximation that describes the **order of complexity**
- How to determine asymptotic complexity:
  - Determine which term contributes the most
  - Discard the other terms!

Asymptotic Complexity: Big-O

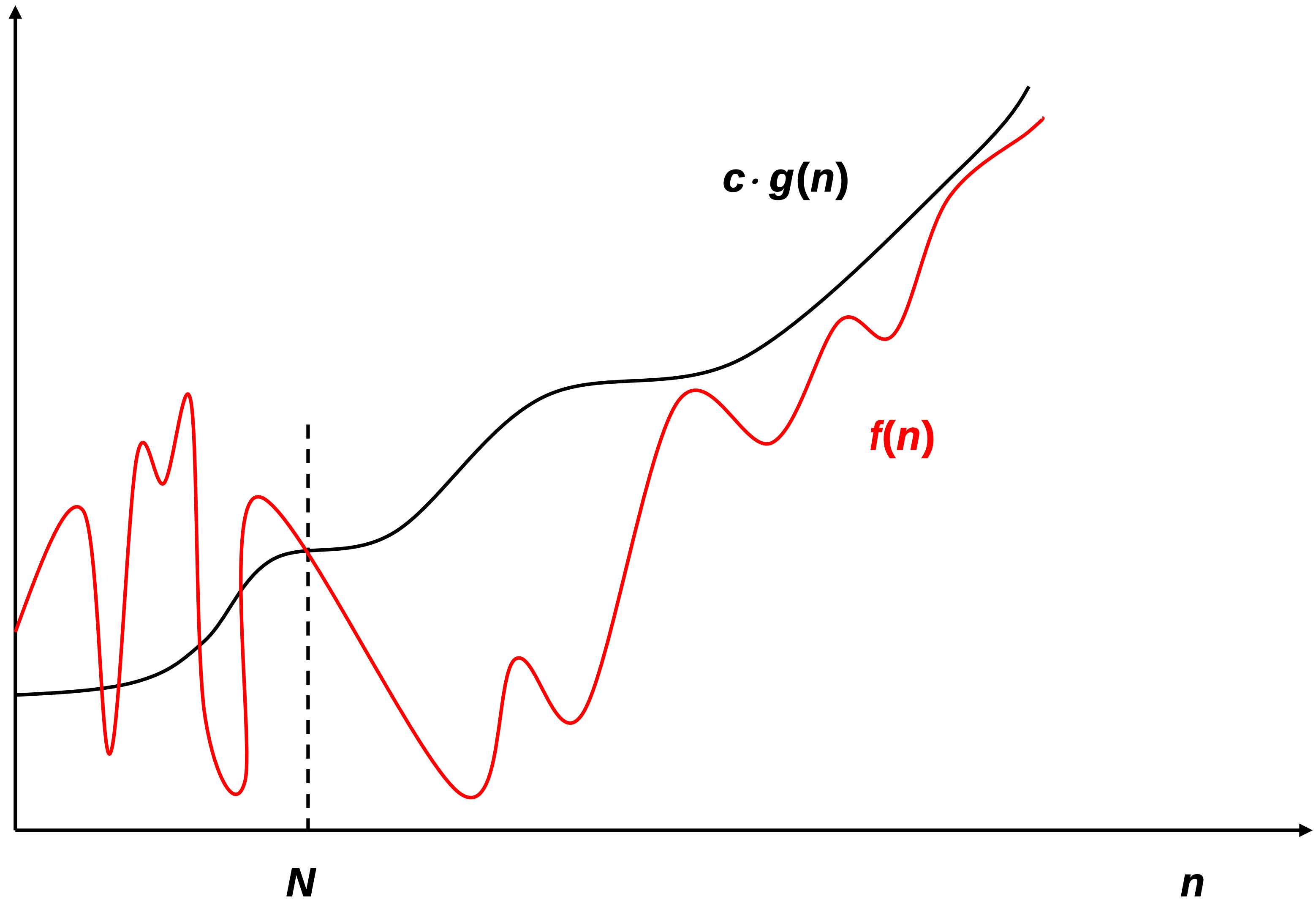
- How to determine order of complexity of $f(n)$:
  - Determine which term contributes the most
  - Discard the other terms!
- Paul Gustav Bachmann introduced the Big-O notation in 1894 to represent the asymptotic complexity of $f(n)$:

$O(g(n))$

- If $f(n)$ is $O(g(n))$, then there are positive numbers c and N such that $f(n) \leq cg(n)$ for all $n \geq N$

- What this basically means: $f(n)$ grows at most as fast as $g(n)$, but never faster
- $g(n)$ is the "upper bound" of $f(n)$
- What is $O(g(n))$ of $f(n) = 2n$?
  - It is $O(n)$
  - Why? Because for any N, and $c = 2$, $2n \leq 2n$

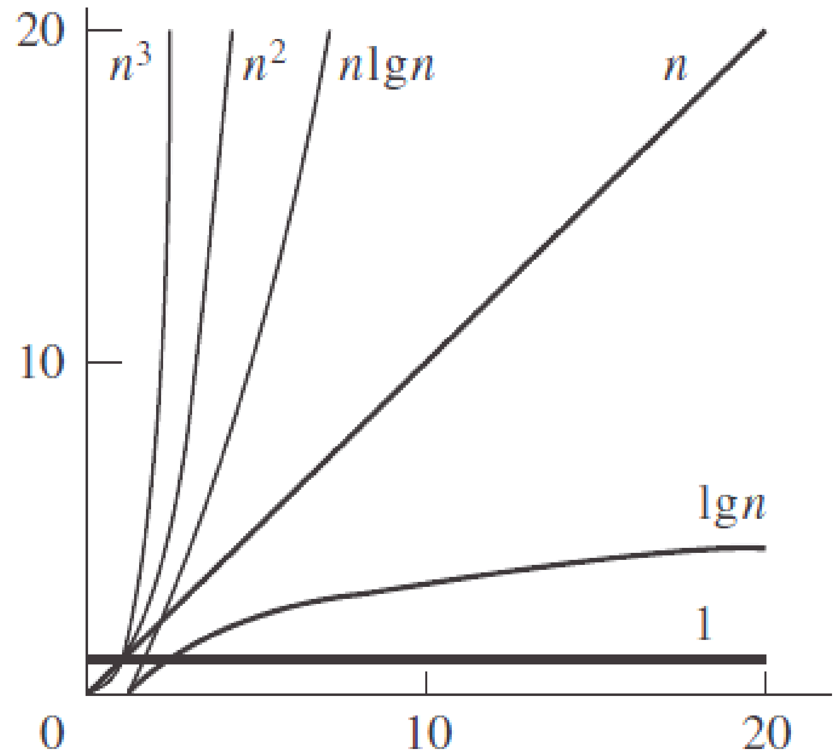
Asymptotic Complexity: Big-O



Asymptotic Complexity: Big-O

$log_2 x = \lg x$

1. $f(n) = n^2$
   - Big-O: $O(n^2)$
2. $f(n) = 1000000\, n^2$
   - Big-O: $O(n^2)$
3. $f(n) = 2n + n^2$
   - Big-O: $O(n^2)$
4. $f(n) = 10 + n + \log n$
   - Big-O: $O(n)$
5. $f(n) = 10\, n^3 + 364\, n^2$
   - Big-O: $O(n^3)$



- What is that straight line at the bottom?
  - O(1), also referred to as constant complexity, describes algorithms where execution time does not depend on the size of the data structure (number of data units)

Big-O Properties

1
- If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$
- i.e., $f(n) = O(g(n)) = O(O(h(n))) = O(h(n))$

2
- If $f(n)$ is $O(h(n))$ and $g(n)$ is $O(h(n))$, then $f(n) + g(n)$ is $O(h(n))$

3
- an^k is $O(n^k)$

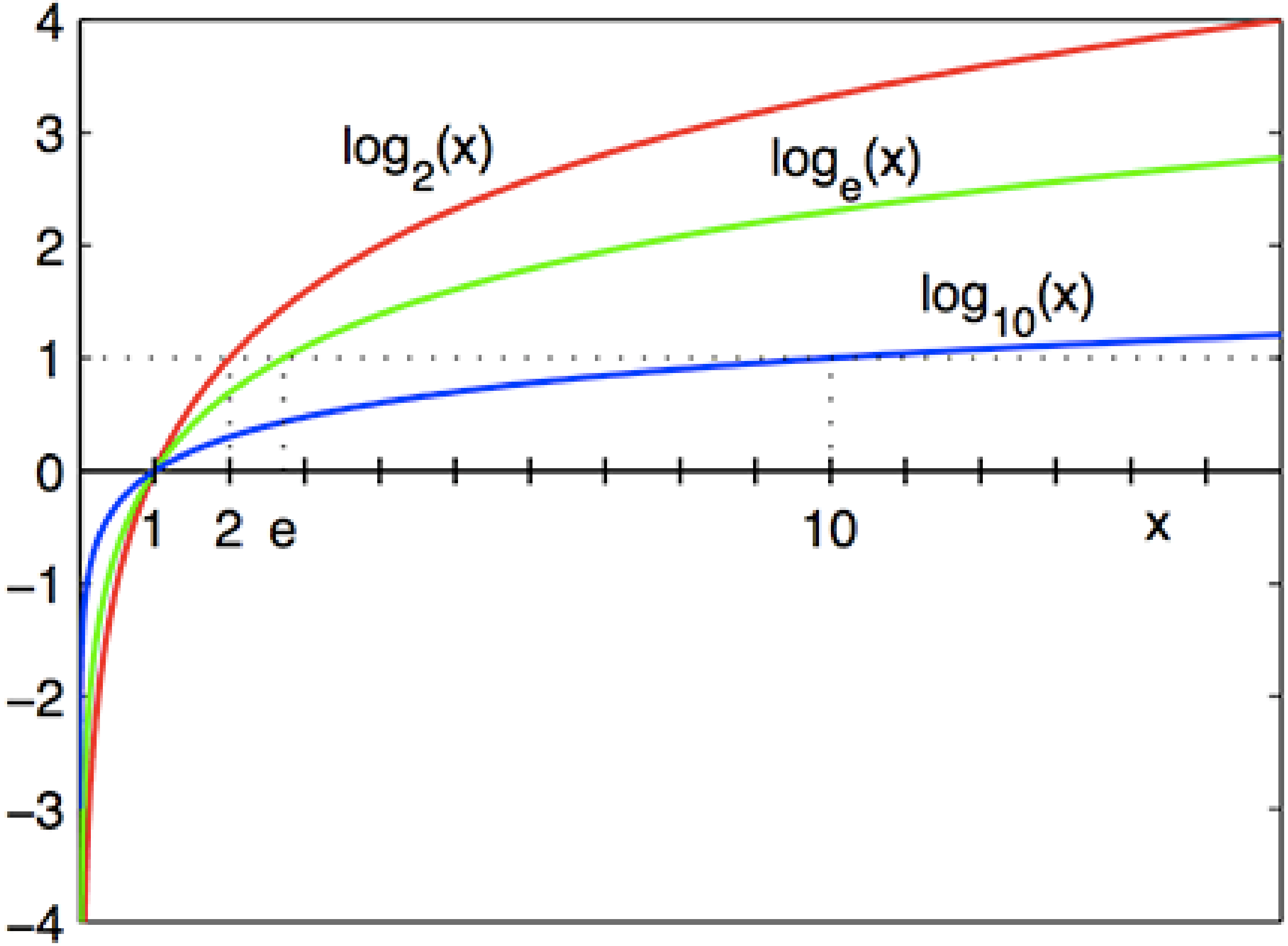
4
- n^k is $O(n^{k+j})$ for any $j > 0$

5
- If $f(n) = c \cdot g(n)$, then $f(n) = O(g(n))$

6
- $\log_a n = O(\log_b n)$ for any numbers $a, b > 1$

7
- $\log_a n$ is $O(\log_2 n)$ for any positive $a \neq 1$

Logarithm: $a^y = x, \log_a x = y$



Calculating Big-O: examples

- Now that you understand Big-O, let’s apply it to actual algorithms
- Count every assignment operation as “work” that contributes to complexity
- Example 1:

```
int i = 0, j = 1;   // 2 assignments
while(i < n) {
   i++;             // assignment
   j += 2;          // assignment
}
```

- How many assignments in total?
- 2 + 2n
- What is the complexity?
  - O(n)

Calculating Big-O: examples

- Example 2:

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

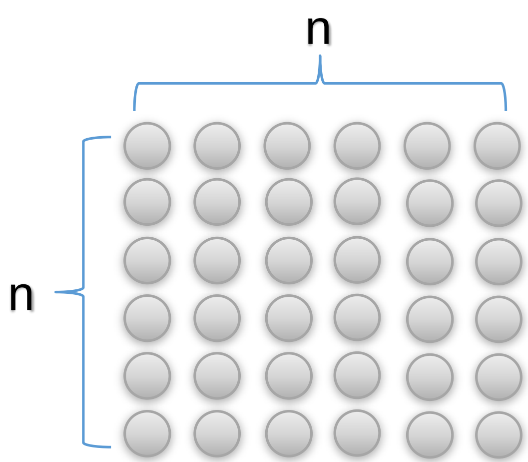
- How many assignments in total?

- $1 + 1 + n\,(1 + 1 + n(1 + 1))$

Calculating Big-O: examples

- Example 2:

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



- How many assignments in total?

- $1 + 1 + n\,(1 + 1 + n(1 + 1)) = 2 + 2n + 2n^2$
- Complexity?

- **$O(n^2)$**

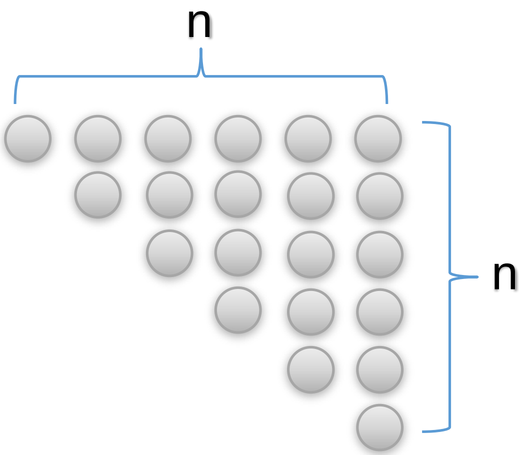
Calculating Big-O: examples

- Example 3:

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

$$1 + 2 + \cdots + n = (n(n+1))/2$$

- How many assignments in total?



- $1 + 3n + \sum_{i=0}^{n} 2i$
- $= 1 + 3n + 2(1 + 2 + \ldots + n)$
- $= 1 + 3n + n(n + 1)$
- Complexity?
- **O(n^2)**

Calculating Big-O: examples

- Example 4:

```
for(i = 0; i < 10; i++)
    for(j = 0; j < n; j++)
          x += y;
```

- How many assignments in total?

- $1 + 10(1 + 1 + n\,(1 + 1)) = 1 + 10(2 + 2n) = 21 + 20n$

- Complexity?

- **O(n)**

Calculating Big-O: examples

- Example 5:

```
for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
        for(k = 0; k < n; k++)
            for(m = i-2; m < i; m++)
                x = i + j + k + m;
```

The last loop will start at i-2, and go until i. Number of iterations:
$i - (i - 2) = 2$

- How many assignments in total?

- $1 + n(1 + 1 + n(1 + 1 + n(\sum ???)))$
- How many times will the innermost loop execute?
- What is the Complexity?
- $O(n^3)$

Calculating Big-O: examples

- Example 6:

```
sum = 0, i = 9;
while(i < n) {
   i++;
   for(j = i - 8; j <= i; ++j)
         sum++;
}
```

How many times will the **while** loop execute?

How many times will the **for** loop execute?

What is the Big-O Complexity?

- **O(n)**

Calculating Big-O: examples

- Recursion:

```
int recursive(int n)
{
   if (n <= 0)
         return 1;
   else
         return 3 * recursive(n-1);

}
```

How many times will the recursive function execute?

What is the Big-O Complexity?

- **O(n)**

Calculating Big-O: examples

- Example 7:

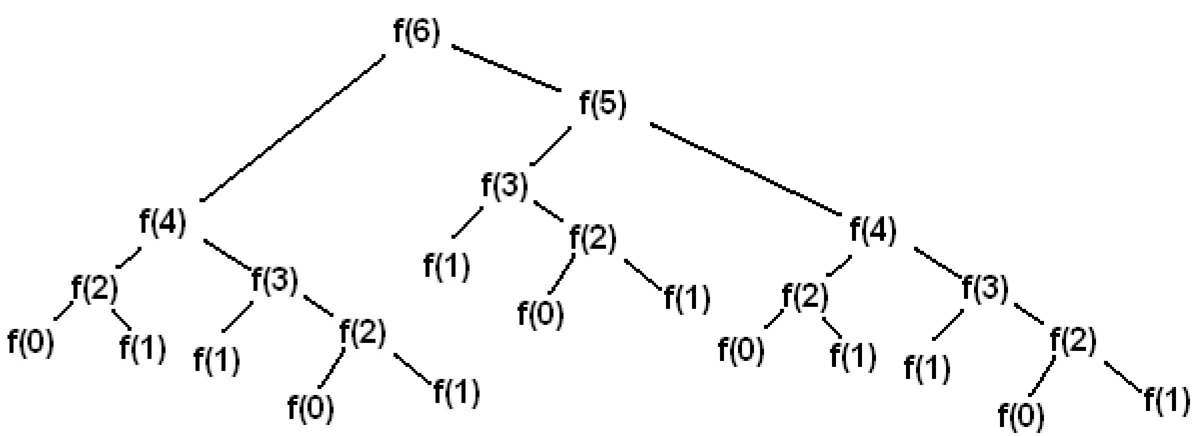
```
int fib(n) {
   if(n <= 1) return n;
   else return fib(n-1) + fib(n-2);
}
```

- What is the complexity?
- Is it O(1 + O(n-1) + O(n-2))?

Complexity is **O(2^n)**

Is it good or bad?

It doesn't get much worse than this!



Calculating Big-O: examples

- Example 8:

```
j = 5;
for(i = n; i >= 1; i/=2)
    j *= j + i;
```

- How many times does the loop execute?
- i takes on values $n/2^0$, $n/2^1$,…, $n/2^m$
- We halve i every time till i < 1
- Therefore,
  - $n/2^m = 1$
- Rewrite as
  - $n = 2^m$
- Now we can calculate m:
  - $m = \log_2 n$
  - *(m is the total number of iterations)*
- Thus, **O(lg n)**

Logarithmic complexity: how many times can you **divide** ***n*** by a given base?