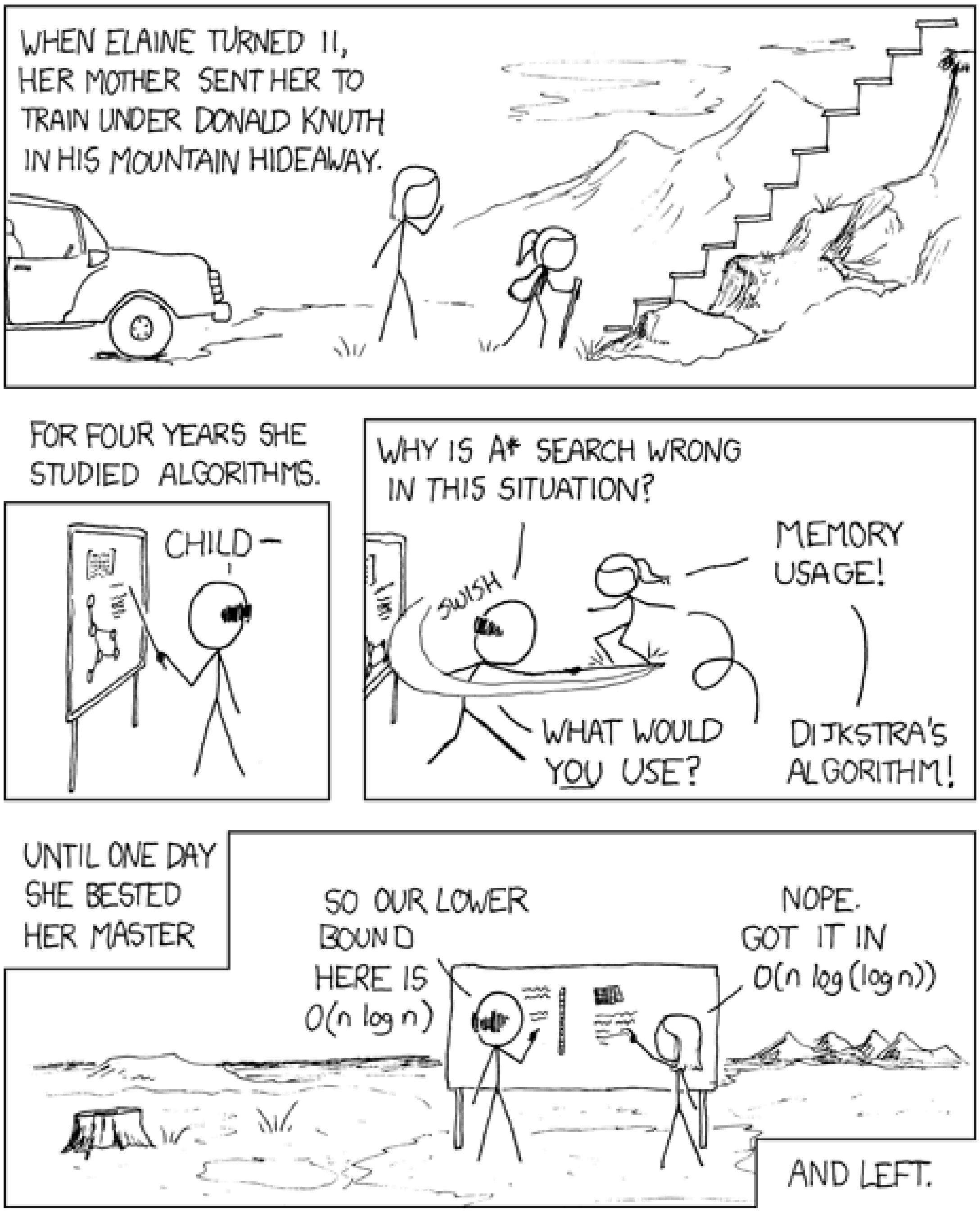
CS240
Fall 2014

Mike Lam, Professor

Algorithm Analysis



HW1 Grades are Posted

- Grades were generally good
- Check my comments!
  - Come talk to me if you have any questions

PA1 is Due 9/17 @ noon

- Web-CAT submission will be active soon
- We will provide a few basic "public" tests
  - These are **not** exhaustive!
- You should thoroughly test your own code
  - Do not rely on the Web-CAT tests to do your debugging for you

HW2 is Posted, Due 9/19

- Due Sept 19 @ 14:30 (2:30pm)
- Algorithm analysis practice
- Submit a PDF
  - LaTeX (.tex template provided)
    - Texmaker, Lyx, or ShareLatex.com
  - MS Word / LibreOffice w/ equation editor
    - Export as PDF!
  - Scan of **EXTREMELY NEAT** handwriting!

Solutions Posted

- New "Files" section on Canvas
- Selected solutions
  - Lab 3 (dictionaries)
  - Lab 4 (class hierarchy)
  - Homework 1 (basic Python)
- DO NOT distribute these outside the class!
  - This is an honor code violation

Algorithm Analysis

- Motivation: “what” and “why”
- Mathematical functions
- Comparative & asymptotic analysis
- Big-O notation (not "Big-Oh"!)

Analyzing algorithms

- We want **efficient** algorithms
  - What metric should we use?
  - How should we normalize?
  - How should we compare?

Empirical Analysis

- "Run it and see"
  - Use the `time` module in Python
  - Vary experiment parameters
    - Input size, algorithm used, number of cores, etc.
  - Report running times in a graph or table

Problems with Empirical Analysis

- Hard to compare across environments
  - Hardware/software differences
- Hard to be comprehensive
  - How many experiments do we need to run?
  - Did we test all relevant input sizes?
- You actually need the code!
  - We have to invest development time

Case Study

- Which is better?

Input Size	Algorithm A	Algorithm B
10	1 s	330 s
20	2 s	430 s
30	3 s	490 s
40	4 s	530 s

Case Study

- Which is better?

Input Size	Algorithm A	Algorithm B
10	1 s	330 s
20	2 s	430 s
30	3 s	490 s
40	4 s	530 s
1,000	100 s	997 s
10,000	1,000 s	1,329 s
100,000	10,000 s	1,661 s
1,000,000	100,000 s	1,993 s
	~28 hours	**~33 minutes**

Case Study

- Which is better?

Input Size	Algorithm A	Algorithm B
10	1.2 s	1.1 s
100	2.0 s	1.9 s
1,000	3.4 s	3.3 s
10,000	4.5 s	4.7 s
100,000	5.9 s	5.9 s
1,000,000	7.0 s	6.8 s

Case Study

- Which is better?

Input Size	Algorithm A	Algorithm B	Algorithm A	Algorithm B
10	1.2 s	1.1 s	1 MB	1 MB
100	2.0 s	1.9 s	2 MB	11 MB
1,000	3.4 s	3.3 s	3 MB	96 MB
10,000	4.5 s	4.7 s	4 MB	1 GB
100,000	5.9 s	5.9 s	5 MB	12 GB
1,000,000	7.0 s	6.8 s	6 MB	140 GB

Case Study

- Which is better?

```python
def search(array, item):
    found = False
    for i in array:
        if i == item:
            found = True
    return found
```

```python
def search(array, item):
    left = 0
    right = len(array)
    while right > left+1:
        mid = (right-left)//2 + left
        if array[mid] > item:
            right = mid
        elif array[mid] < item:
            left = mid+1
        else:
            left = mid
            right = mid+1
    return left < len(array) and \
        array[left] == item
```

Case Study

- Which is better?

```
def search(array, item):
    found = False
    for i in array:
        if i == item:
            found = True
    return found
```

```
def search(array, item):
    found = False
    for i in array:
        if i == item:
            found = True
            break
    return found
```

Case Study

- Which is better?

```
def search(array, item):
    found = False
    for i in array:
        if i == item:
            found = True
    return found
```

```
def search(array, item):
    found = False
    for i in array:
        if i == item:
            found = True
            break
    return found
```

Best: *n* comparisons
Worst: *n* comparisons
Average: *n* comparisons

Case Study

- Which is better?

```
def search(array, item):
    found = False
    for i in array:
        if i == item:
            found = True
    return found
```

Best: *n* comparisons
Worst: *n* comparisons
Average: *n* comparisons

```
def search(array, item):
    found = False
    for i in array:
        if i == item:
            found = True
            break
    return found
```

Best: 1 comparison
Worst: *n* comparisons
Average: *n*/2 comparisons

Lessons Learned

- Running times can be deceiving
  - We have to normalize by input size
- CPU time isn't the only metric of interest
  - Memory usage, I/O time, power usage, etc.
  - Focus on “primitive operations” (for simplicity)
- Code length has little bearing on performance
  - More complicated code can be faster
- Best, worst, average cases can all be different
  - Focus on the worst case (for guarantees)

Analyzing algorithms

- We want **efficient** algorithms
  - What metric should we use?
  - How should we normalize?
  - How should we compare?

Analyzing algorithms

- We want **efficient** algorithms
  - What metric should we use?
    - Worst-case primitive operations
  - How should we normalize?
  - How should we compare?

Analyzing algorithms

- We want **efficient** algorithms
  - What metric should we use?
    - Worst-case primitive operations
  - How should we normalize?
    - By input size
  - How should we compare?

Analyzing algorithms

- We want **efficient** algorithms
  - What metric should we use?
    - Worst-case primitive operations
  - How should we normalize?
    - By input size
  - How should we compare?
    - Asymptotic analysis

Functions

- First, a brief foray into mathematics...

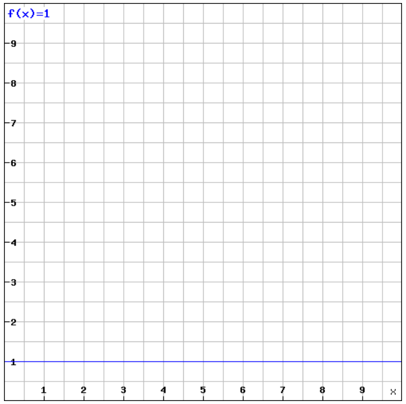
(don't worry, it will be brief!)

Functions

- Constant function:

$$f(n) = c$$

$$O(1)$$



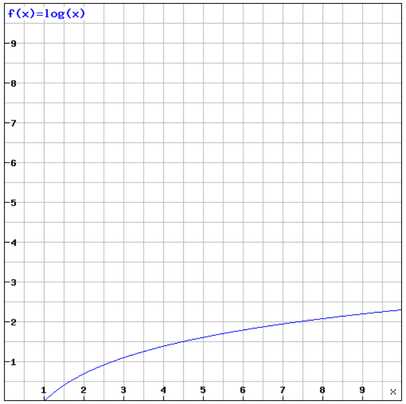
- Input size doesn't matter
- As long as *c* is relatively small, constant time is as good as it gets!

Functions

- Logarithm function:

$$f(n) = log_b n$$

$$O(\log n)$$



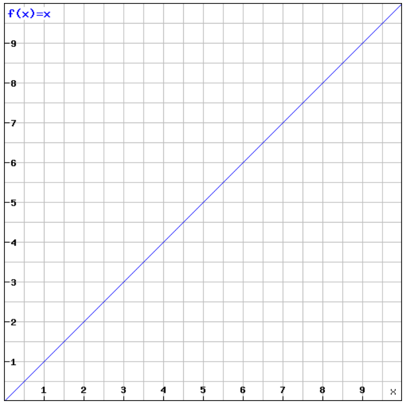
- Grows logarithmically with input size
  - Usually the base (b) is 2
- Usually encountered with divide-and-conquer methods

Functions

- Linear function:

$$f(n) = n$$

$$O(n)$$



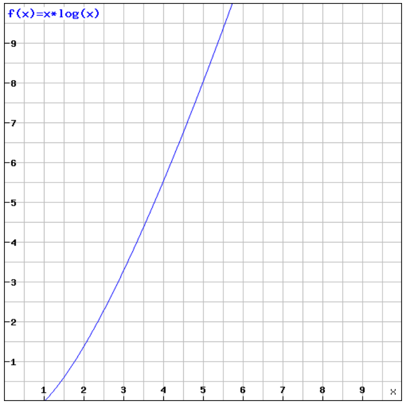
- Grows linearly with input size
- Often, this is the best we can hope for
  - Reading objects into memory is O(n)

Functions

- Linearithmic ("quasi-linear") function:

$$f(n) = n \log_b n$$

$$O(n \log n)$$



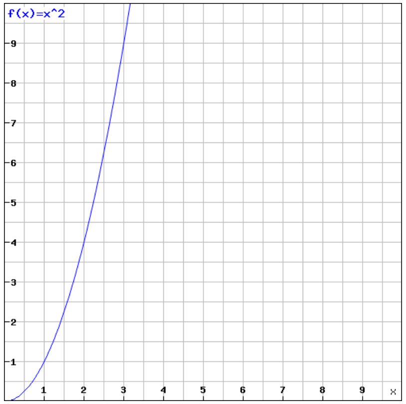
- Grows slightly faster than linear
- Many important algorithms are O(n log n)
  - Most of the "good" sorting algorithms

Functions

- Quadratic function:

$$f(n) = n^2$$

$$O(n^2)$$



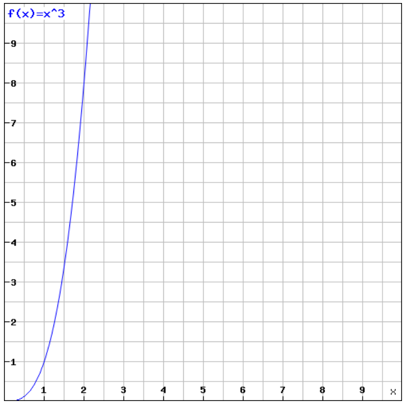
- Scales quadratically with input size
- Usually arises from nested loops

Functions

- Cubic function:

$$f(n) = n^3$$

$$O(n^3)$$



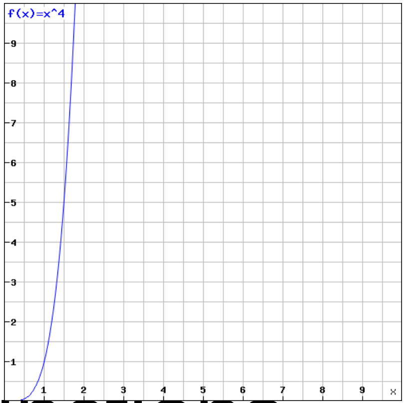
- Scales cubically with input size
- Usually arises from triply-nested loops

Functions

- Polynomial function:

$$f(n) = n^x$$

$$O(n^x)$$



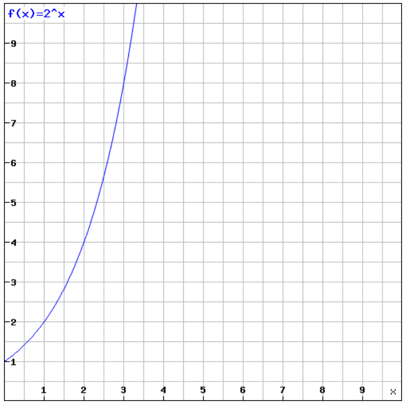
- Generalization of quadratic/cubic functions
- We want *x* to be as small as possible
  - Usually, $x > 4$ is impractical

Functions

- Exponential function:

$$f(n) = b^n$$

$$O(b^n)$$



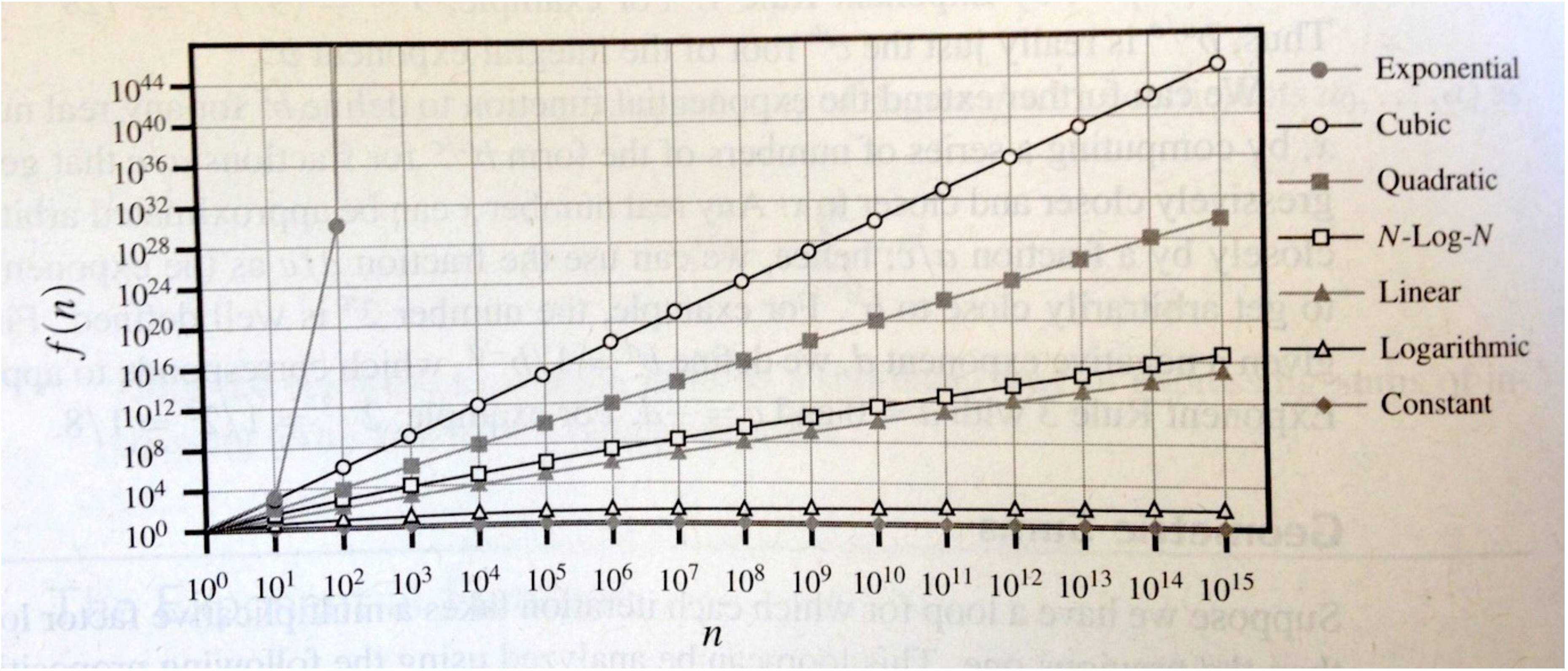
- Usually the base is 2
- Currently infeasible when n > ~100
- Avoid this!

Functions

- There are worse functions
  - Factorial: $f(n) = n!$
  - Double exponential: $f(n) = b^{b^n}$
- We won't be using these in this class
  - But you should know the other eight!

Comparing Functions

- Plotting all functions on one graph is difficult
  - Use log-log axes



Comparing Functions

- We now have an ordering of functions:

1. Constant: $f(n) = 1$ (slowest-growing)
2. Logarithmic: $f(n) = \log n$
3. Linear: $f(n) = n$
4. Linearithmic: $f(n) = n \log n$
5. Quadratic: $f(n) = n^2$
6. Cubic: $f(n) = n^3$
7. Polynomial: $f(n) = n^b$
8. Exponential: $f(n) = b^n$ (fastest-growing)

Functions

- We have actually described eight function families
  - There are a infinite number of functions in each family, with different constant scalar factors
  - Example: n, 3n, and 42n are all linear functions
  - Example: n^2, $3n^2$, and $42n^2$ are all quadratic
  - Within a family, smaller constants are better
  - How do we compare between families?
    - Use our function ordering!

Comparing Functions

- So we won't talk about the **running time** of an algorithm ...
- … but rather we'll talk about how **fast** the running time **grows** as the problem size increases …
- … and **compare** the growth rates of various algorithms

Comparing Functions

- This type of analysis is called "asymptotic analysis"
- Because it deals with the behavior of functions in the **asymptotic** sense as *n* (input size) increases to infinity

Asymptotic Analysis

- Big-O notation
  - Method for mathematically comparing functions
  - Provides us with a robust way of saying "this function grows faster than that one"
  - We will use that statement as a proxy for: "this algorithm is more efficient than that one"

Big-O Notation

- Formal definition:
  - Let $f(n)$ and $g(n)$ be functions
    - Mapping input sizes to running time
  - We say this:

$$f(n) \text{ is } O(g(n))$$

  - If there is a constant $c > 0$ and an integer $n_0 \geq 1$ such that:

$$f(n) \leq c \cdot g(n) \qquad \text{for } n \geq n_0$$

Big-O Notation

- Informally, we say "*f(n) is O(g(n))*" if f(n) grows as slow or slower than g(n)
  - According to our ordering of function growth
- Or: "*Algorithm X is O(f(n))*" if the growth rate of the running time of Algorithm X is O(f(n))
  - Examples:
    - "Linear search is O(n)"
    - "Binary search is O(log n)"
    - "Matrix multiplication is O(n^3)"

Big-O Notation

- Instead of this:

$$f(n) \text{ } is \text{ } O(g(n))$$

- Some people say this:

$$f(n) \in O(g(n))$$

- This is *set notation* describing sets or families of functions
- Both are correct; I tend to use the former

Big-O Notation

- Big-O:

$$f(n) \text{ is } O(g(n)) \quad \text{iff.} \quad f(n) \leq c \cdot g(n) \quad \text{for } n \geq n_0$$

(upper bound)

- Big-Omega:

$$f(n) \text{ is } \Omega(g(n)) \quad \text{iff.} \quad f(n) \geq c \cdot g(n) \quad \text{for } n \geq n_0$$

(lower bound)

- Big-Theta:

$$f(n) \text{ is } \Theta(g(n)) \quad \text{iff.} \quad c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n) \quad \text{for } n \geq n_0$$

(strict bounds: upper and lower)

Big-O Notation

- Limit-based definitions:

$f(n)$ is $O(g(n))$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$$

where c is a constant
and
c < ∞

$f(n)$ is $\Omega(g(n))$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$$

where c is a constant
and
c > 0

$f(n)$ is $\Theta(g(n))$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$$

where c is a constant
and
0 < c < ∞

Big-O Notation

- We now have a strict ordering of complexity classes:

1. Constant: $\Theta(1)$ (slowest-growing)
2. Logarithmic: $\Theta(\log n)$
3. Linear: $\Theta(n)$
4. Linearithmic: $\Theta(n \log n)$
5. Quadratic: $\Theta(n^2)$
6. Cubic: $\Theta(n^3)$
7. Polynomial: $\Theta(n^b)$
8. Exponential: $\Theta(b^n)$ (fastest-growing)

Big-O Notation

- Find the slowest-growing function family for which the Big-O definition is true
  - Example: Don't say Algorithm X is $O(n^3)$ if it is $O(n^2)$ even though the former is technically true as well
  - Walking traveler example
- Drop slower-growing ("lower-order") terms
  - Example: Don't say Algorithm X is $O(n + \log n)$
    - Drop the slower-growing function and say it is $O(n)$
  - Goldfish/elephant example

A Word of Caution

- Sometimes Big-O notation can hide large constant factors
- The fact that Algorithm X is O(n) doesn't matter if the constant is 10^{100}!
- Something to keep in mind

So what is **efficient**?

- "Efficient" vs. "feasible"
- Everything O(n log n) is generally considered efficient for all reasonable input sizes
- For small *n*, any algorithm can be feasible
  - Obviously, the slower-growing, the better
  - Generally, small polynomials are the limit of feasibility
  - Sometimes *approximation* algorithms can help
- Exponentials are right out

Ceiling and Floor Functions

- *log n* is rarely an integer value
- Often we want to coerce values to be integers for the sake of analysis
- We can use the *floor* and *ceiling* functions to round real numbers to nearest integers:
  - = *floor(x)* = largest integer ≤ *x*
  - $\lfloor x \rfloor$ = *ceil(x)* = largest integer ≥ *x*

$\lceil x \rceil$

Little-O Notation

- Big-O:

$$f(n) \text{ is } O(g(n)) \quad \text{iff.} \quad f(n) \le c \cdot g(n) \quad \text{for } \textbf{some } c,\ n \ge n_0$$

- Little-O:

$$f(n) \text{ is } o(g(n)) \quad \text{iff.} \quad f(n) \le c \cdot g(n) \quad \text{for } \textbf{all } c,\ n \ge n_0$$

- Basically means "f(n) grows **much slower** than g(n)"
  - Alternately, "f(n) is **dominated** by g(n)"
- Similarly defined for Little-Ω (ω)

L'Hôpital's Rule

If $\lim_{n \to \infty} f(n) = \lim_{n \to \infty} g(n) = \infty$ and $f'(n)$ and $g'(n)$ exist, then

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{f'(n)}{g'(n)}$$

- This is useful for proving Big-O assertions
- Uses first derivatives *f'(n)* and *g'(n)*

Key Masteries

- You should be able to:
  - Explain why we need asymptotic analysis
  - Compare functions and complexity classes
    - Especially the members of the eight function families we talked about
  - Explain Big-O notation (O, Ω, Θ)
    - Use it to prove relations between complexity classes
  - Describe growth rates for concrete algorithms
    - Using operation counts and Big-O notation