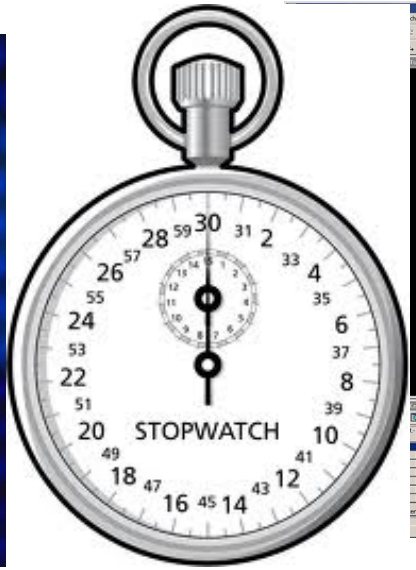
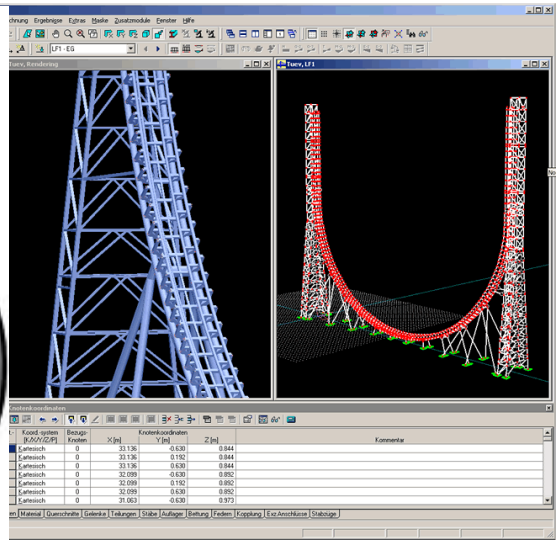
Lecture 3
Complexity Analysis





EECS 281: Data Structures & Algorithms

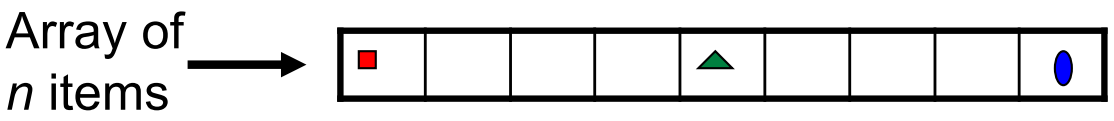
Complexity Analysis: Overview

Data Structures & Algorithms

Complexity Analysis

- What is it?
  - Given an algorithm and input size *n*, how many steps are needed?
  - Each step should take *O*(1) time
  - As input size grows, how does number of steps change?
    - Focus is on TREND
- How do we measure complexity?
  - Express the rate of growth as a function *f*(*n*)
  - Use the big-O notation
- Why is this important?
  - Tells how well an algorithm scales to larger inputs
  - Given two algorithms, we can compare performance before implementation

Metrics of Algorithm Complexity



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

Using a linear search over *n* items,
how many comparisons will it take to find item *x*?

- Best-Case
  - Least number of comparisons required, given ideal input
  - Analysis performed over inputs of a given size
  - Example: Data is found in the first place you look
- Worst-Case
  - Most number of comparisons required, given hard input
  - Analysis performed over inputs of a given size
  - Example: Data is found in the last place you could possibly look
- Average-Case
  - Average number of comparisons required, given any input
  - Average performed over all possible inputs of a given size

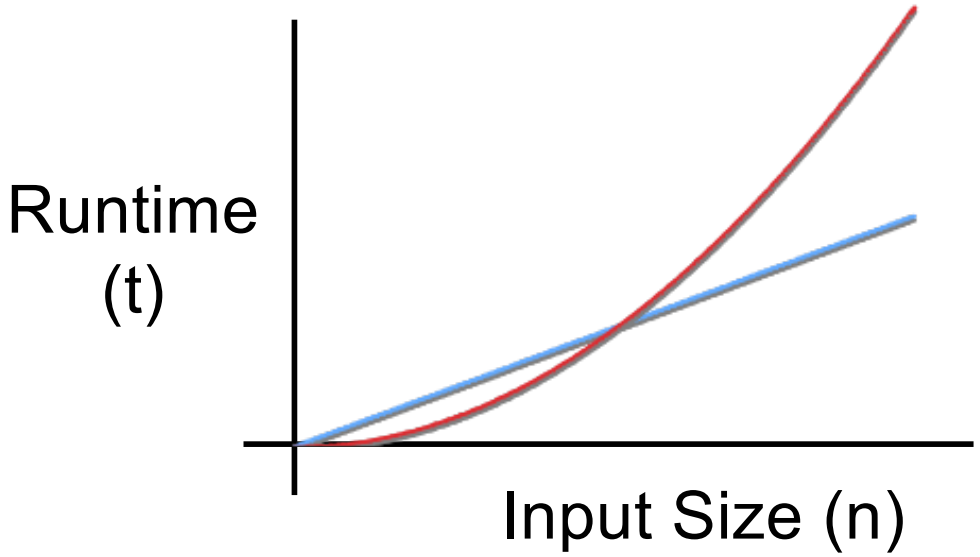
What Affects Runtime?

- The algorithm
- Implementation details
  - Skills of the programmer
- CPU Speed / Memory Speed
- Compiler (Options used)
  - `g++ -g3` (for debu**g**ging, highest level of information)
  - `g++ -O3` (**O**ptimization level **3** for speed)
- Other programs running in parallel

- Amount of data processed (Input size)

Input Size versus Runtime

- Rate of growth independent of most factors
  - CPU speed, compiler, etc.
- Does doubling input size mean doubling runtime?
- Will a “fast” algorithm still be “fast” on large inputs?



How do we measure input size?

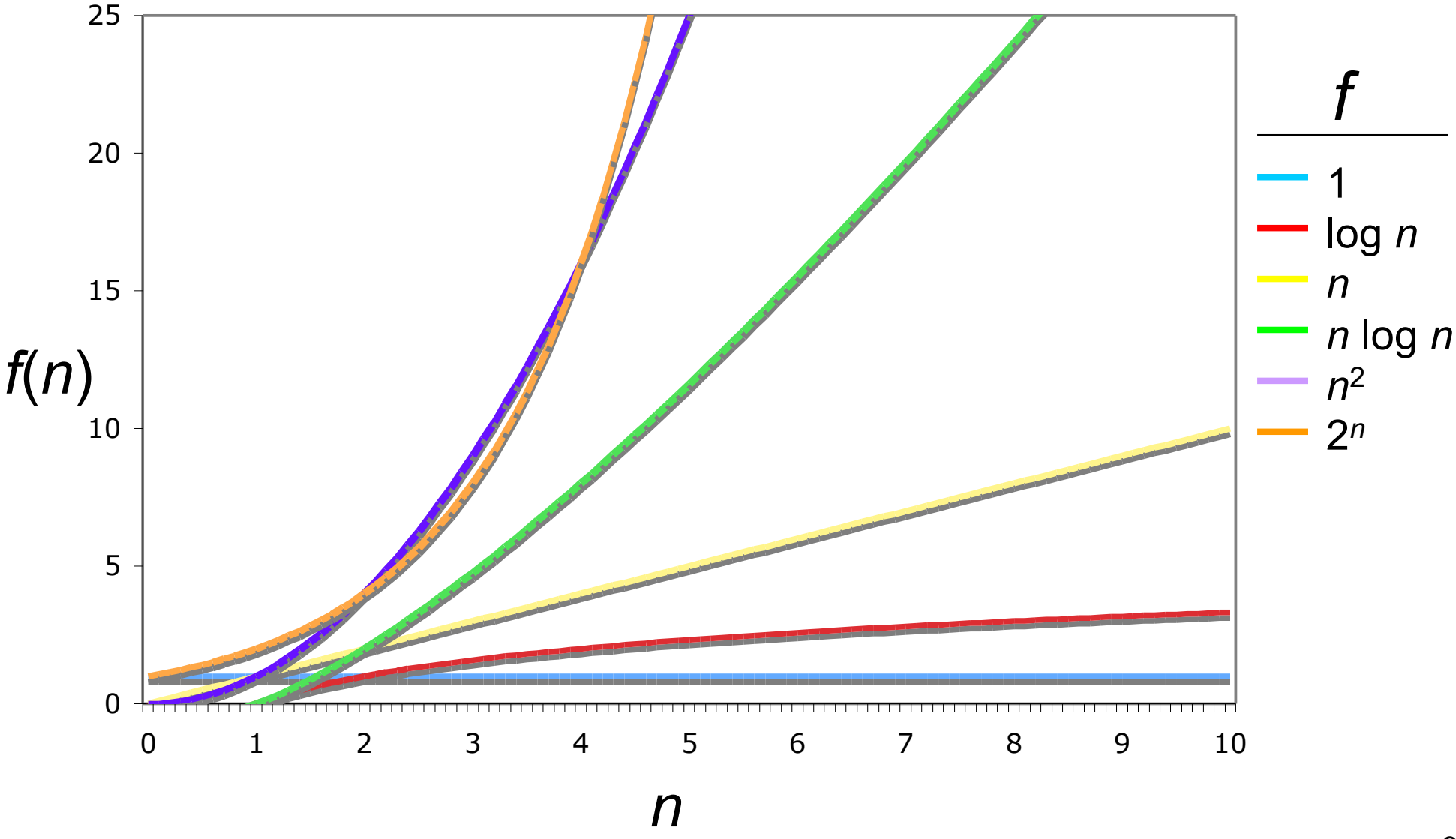
Measuring & Using Input Size

- Number of bits
  - In an **int**, a **double**? (32? 64?)
- Number of items: what counts as an item?
  - Array of integers? One integer? One digit? …
  - One string? Several strings? A char?
- Notation and terminology
  - *n* Input size
  - *f(n)* Maximum number of steps taken by an algorithm when input has size *n* ("*f* of *n*")
  - *O(f(n))* Complexity class of *f*(*n*) ("Big-O of *f* of *n*")

Common Orders of Functions

Notation	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Loglinear, Linearithmic
$O(n^2)$	Quadratic
$O(n^3)$, $O(n^4)$, …	Polynomial
$O(c^n)$	Exponential
$O(n!)$	Factorial
$O(2^{2^n})$	Doubly Exponential

Graphing $f(n)$ Runtimes

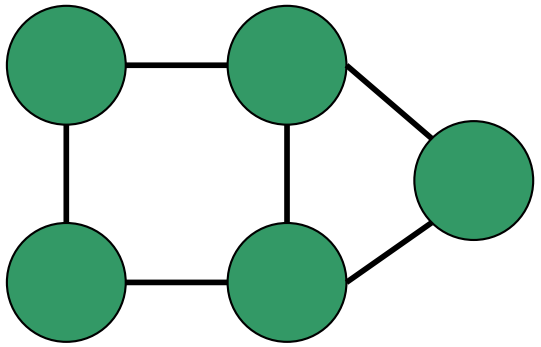


Input Size Example

Graph $G = (V, E)$:

V = 5 Vertices

E = 6 Edges



What should we measure?

- Vertices?
- Edges?
- Vertices and Edges?

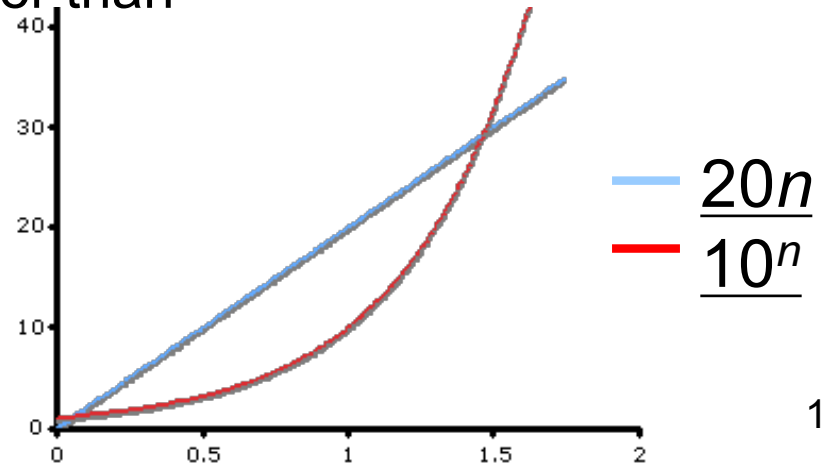
When in doubt, measure input size in bits

Use V and E to determine which contributes more to the total number of steps

- Big-O examples: $E \log V$, EV, $V^2 \log E$

From Analysis to Application

- Algorithm comparisons are independent of hardware, compilers and implementation tweaks
- Predict which algorithms will eventually be faster
  - *For large enough inputs*
  - $O(n^2)$ time algorithms will take longer than $O(n)$ algorithms
- Constants can often be ignored because they do not affect asymptotic comparisons
  - Algorithm with $20n$ steps runs faster than algorithm with 10^n steps. Why?



Complexity Analysis: Overview

Data Structures & Algorithms

Complexity Analysis: Counting Steps

Data Structures & Algorithms

Q: What counts as one step in a program ?

A: Primitive operations

- **`a)`** Variable assignment
- **`b)`** Arithmetic operation
- **`c)`** Comparison
- **`d)`** Array indexing or pointer reference
- **`e)`** Function call (not counting the data)
- **`f)`** Function return (not counting the data)

Runtime of 1 step is independent on input

Counting Steps: `for` Loop

- The basic form of a for-loop:

```
for (initialization; test; update)
```

- The initialization is performed once (1)
- The test is performed every time the body of the loop runs, plus once for when the loop ends (n + 1)
- The update is performed every time the body of the loop runs (n)

Counting Steps: Polynomial

```
1  int func1(int n) {
2    int sum = 0;
3    for (int i = 0; i < n; ++i) {
4        sum += i;
5    }  // for
6    return sum;
7  }  // func1()
```

```
1
2    1 step
3    1 + 1 + 2n steps
4     1 step
5
6    1 step
7
```

Total steps: $4 + 3n$

```
8   int func2(int n) {
9     int sum = 0;
10    for (int i = 0; i < n; ++i) {
11        for (int j = 0; j < n; ++j)
12            ++sum;
13    }  // for i
14    for (int k = 0; k < n; ++k) {
15        --sum;
16    }  // for
17    return sum;
18  }  // func2()
```

```
8
9   1 step
10  1 + 1 + 2n steps
11   1 + 1 + 2n steps
12    1 step
13
14  1 + 1 + 2n steps
15   1 step
16
17  1 step
18
```

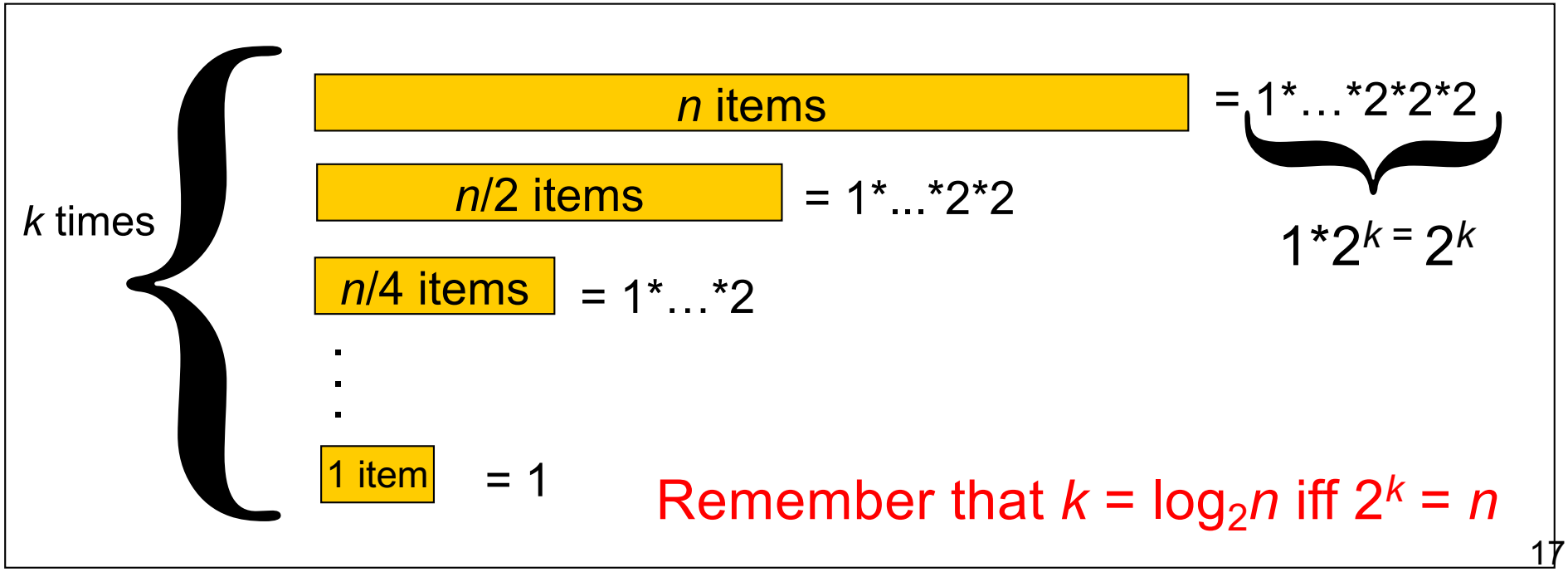
Total steps: $3n^2 + 7n + 6$

Counting Steps: Logarithmic

```
1  int func3(int n) {
2    int sum = 0;
3    for (int i = n; i > 1; i /= 2)
4        sum += i;
5
6    return sum;
7  }  // func3()
```

```
1
2   1 step
3   1 + 1 + ~log n * (2 steps)
4     1 step
5
6   1 step
7
```

Total: $4 + 3 \log n = O(\log n)$



Examples of *O*(log *n*) Time

```
1  uint32_t logB(uint32_t n) {
2    // find binary log, round up
3    uint32_t r = 0;
4    while (n > 1) {
5      n /= 2
6      r++;
7    }  // while
8    return r;
9  }  // logB()
```

```
10 int *bsearch(int *lo, int *hi, int val) {
11   // find position of val between lo,hi
12   while (hi >= lo) {
13     int *mid = lo + (hi - lo) / 2;
14     if (*mid == val)
15       return mid;
16     else if (*mid > val)
17       hi = mid - 1;
18     else
19       lo = mid + 1;
20   }  // while
21   return nullptr;
22 }  // bsearch()
```

Algorithm Exercise

How many multiplications, if size = ***n*****?**

```
1   // REQUIRES: in and out are arrays with size elements
2   // MODIFIES: out
3   // EFFECTS: out[i] = in[0] *...* in[i-1] * in[i+1] *...* in[size-1]
4  void f(int *out, const int *in, int size) {
5    for (int i = 0; i < size; ++i) {
6      out[i] = 1;
7      for (int j = 0; j < size; ++j) {
8        if (i != j)
9          out[i] *= in[j];
10     }  // for j
11   }  // for i
12 }  // f()
```

Algorithm Exercise

How many multiplications and divisions, if size = *n*?

```
1   void f(int *out, const int *in, int size) {
2     int product = 1;
3     for (int i = 0; i < size; ++i)
4       product *= in[i];
5
6     for(int i = 0; i < size; ++i)
7       out[i] = product / in[i];
8   }  // f()
```

Complexity Analysis: Counting Steps

Data Structures & Algorithms

Complexity Analysis: Big-O Notation

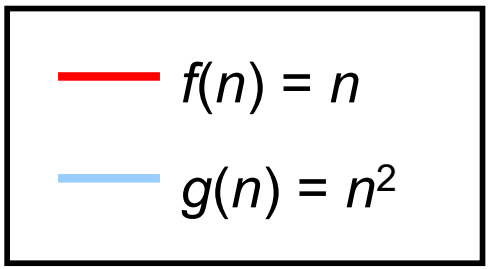
Data Structures & Algorithms

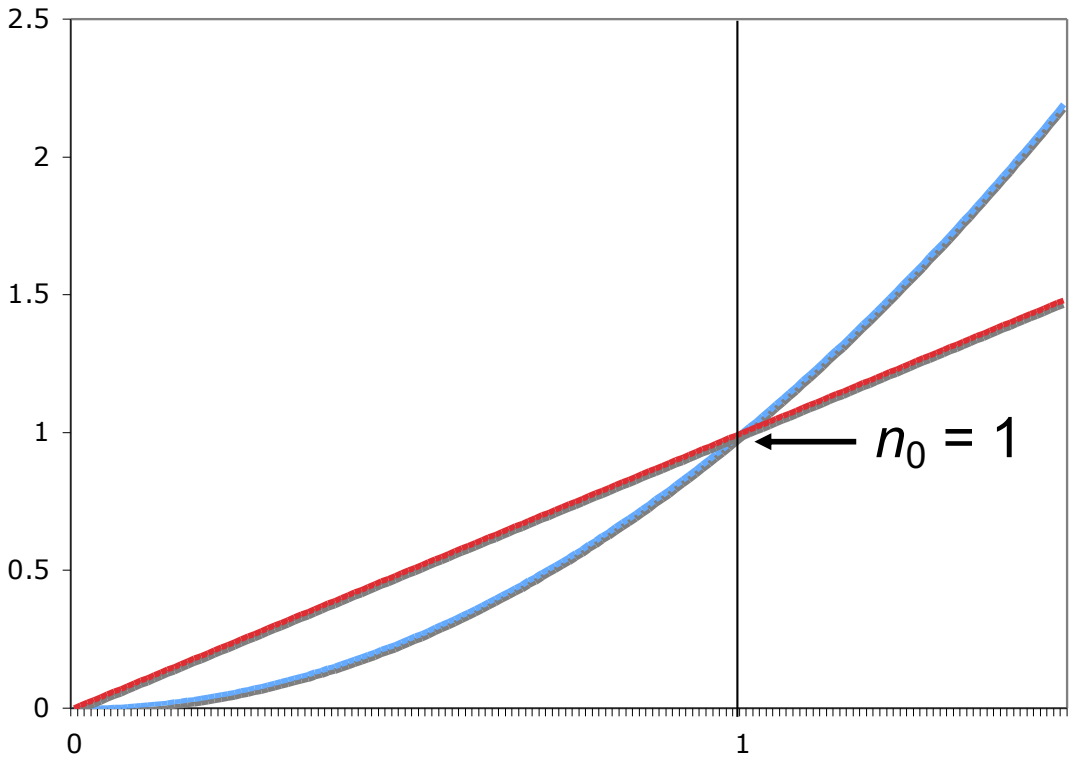
Big-O Definition

$f(n) = O(g(n))$ if and only if there are constants

$c > 0$, $n_0 \geq 0$ such that $f(n) \leq c * g(n)$ whenever $n \geq n_0$

Is $n = O(n^2)$?





Big-O: Sufficient (but not necessary) Condition

$$\text{If}\left[\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right)=d<\infty\right]\text{ then } f(n)\text{ is } O(g(n))$$

$\log_2 n = O(2n)$?

$f(n) = \log_2 n$

$g(n) = 2n$

$\lim_{n\to\infty}\left(\frac{\log n}{2n}\right)$: ∞/∞

$\lim_{n\to\infty}\left(\frac{1}{2n}\right)$: Use L'Hopital's Rule

$0 = d < \infty$: $\log_2 n = O(2n)$

$\sin\left(\frac{n}{100}\right) = O(100)$?

$f(n) = \sin\left(\frac{n}{100}\right)$

$g(n) = 100$

$\lim_{n\to\infty}\left(\frac{\sin\left(\frac{n}{100}\right)}{100}\right)$: Condition does not hold but it is true that $f(n) = O(g(n))$

Big-O: Can We Drop Constants?

$3n^2 + 7n + 42 = O(n^2)$?

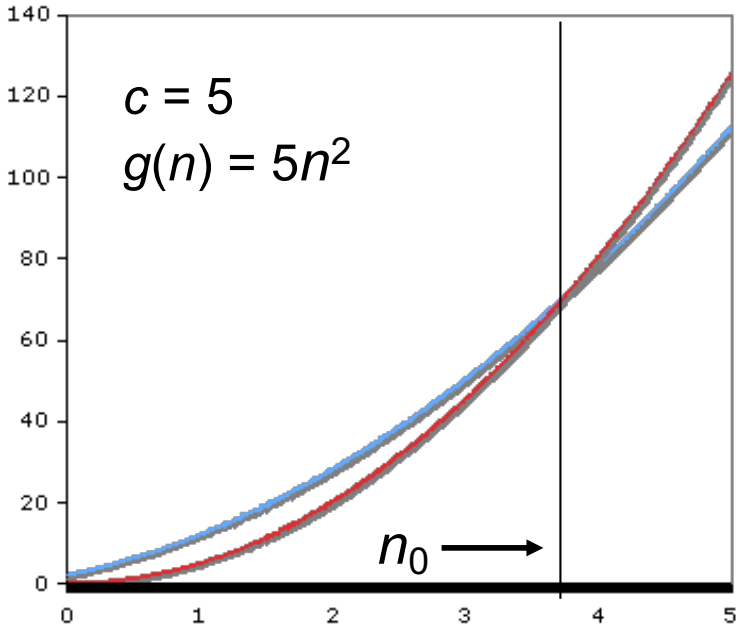
$f(n) = 3n^2 + 7n + 42$

$g(n) = n^2$

Definition

$c > 0, n_0 \geq 0$ such that

$f(n) \leq c \cdot g(n)$ whenever $n \geq n_0$



Sufficient Condition

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

$$\lim_{n\to\infty} \left(\frac{3n^2 + 7n + 42}{n^2} \right)$$

$$\lim_{n\to\infty} \left(\frac{6n + 7}{2n} \right)$$

$$\lim_{n\to\infty} \left(\frac{6}{2} \right)$$

Rules of Thumb

1. Lower-order terms can be ignored
   - $n^2 + n + 1 = O(n^2)$
   - $n^2 + \log(n) + 1 = O(n^2)$

2. Coefficient of the highest-order term can be ignored
   - $3n^2 + 7n + 42 = O(n^2)$

Log Identities

Identity	Example
$\log_a(xy) = \log_a x + \log_a y$	$\log_2(12) =$
$\log_a(x/y) = \log_a x - \log_a y$	$\log_2(4/3) =$
$\log_a(x^r) = r \log_a x$	$\log_2 8 =$
$\log_a(1/x) = -\log_a x$	$\log_2 1/3 =$
$\log_a x = \frac{\log x}{\log a} = \frac{\ln x}{\ln a}$	$\log_7 9 =$
$\log_a a = ?$ $\log_a 1 = ?$	

Power Identities

Identity	Example
$a^{(n+m)} = a^n a^m$	$2^5 =$
$a^{(n-m)} = a^n / a^m$	$2^{3-2} =$
$(a^{(n)})^m = a^{nm}$	$(2^2)^3 =$
$a^{-n} = \frac{1}{a^n}$	$2^{-4} =$
$a^{-1} = ?$ $a^0 = ?$ $a^1 = ?$	

$\log_a(xy) = \log_a x + \log_a y$
$\log_a(x/y) = \log_a x - \log_a y$
$\log_a(x^r) = r \log_a x$
$\log_a(1/x) = -\log_a x$
$\log_a x = \frac{\log x}{\log a} = \frac{\ln x}{\ln a}$

Exercise

$a^{(n+m)} = a^n a^m$
$a^{(n-m)} = a^n / a^m$
$(a^{(n)})^m = a^{nm}$
$a^{-n} = \frac{1}{a^n}$

True or false?

$10^{100} = O(1)$

$3n^4 + 45n^3 = O(n^4)$

$3^n = O(2^n)$

$2^n = O(3^n)$

$45 \log(n) + 45n = O(\log(n))$

$\log(n^2) = O(\log(n))$

$[\log(n)]^2 = O(\log(n))$

Find $f(n)$ and $g(n)$,

such that $f(n) \neq O(g(n))$

and $g(n) \neq O(f(n))$

Big-O, Big-Theta, and Big-Omega

	Big-O (O)	Big-Theta (Θ)	Big-Omega (Ω)
Defines	Asymptotic upper bound	Asymptotic tight bound	Asymptotic lower bound
Definition	$f(n) = O(g(n))$ if and only if there exists an integer n_0 and a real number c such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$	$f(n) = \Theta(g(n))$ if and only if there exists an integer n_0 and real constants c_1 and c_2 such that for all $n \geq n_0$: $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	$f(n) = \Omega(g(n))$ if and only if there exists an integer n_0 and a real number c such that for all $n \geq n_0$, $f(n) \geq c \cdot g(n)$
Mathematical Definition	$\exists n_0 \in Z, \exists c \in R:$ $\forall n \geq n_0, f(n) \leq c \cdot g(n)$	$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$	$\exists n_0 \in Z, \exists c \in R:$ $\forall n \geq n_0, f(n) \geq c \cdot g(n)$
$f_1(n)=2n+1$	$O(n)$ or $O(n^2)$ or $O(n^3)$…	$\Theta(n)$	$\Omega(n)$ or $\Omega(1)$
$f_2(n)=n^2+n+5$	$O(n^2)$ or $O(n^3)$ …	$\Theta(n^2)$	$\Omega(n^2)$ or $\Omega(n)$ or $\Omega(1)$

Complexity Analysis: Big-O Notation

Data Structures & Algorithms

Complexity Analysis: Amortized Complexity

Data Structures & Algorithms

Amortized Complexity

- A type of worst-case complexity
- Used when the work/time profile is “spiky” (sometimes it is very expensive, but most times it is a small expense)
- Analysis performed over a sequence of operations covering of a given range
  - The sequence selected includes expensive and cheap operations
- Considers the average cost of one operation over a sequence of operations
  - Best/Worst/Average-case only consider operations independently
  - Different from average-case complexity!
- Key to understanding expandable arrays and STL vectors, priority queues, and hash tables

Cell Phone Bill* Example

- Pay $100 once per month, each call and text has no (added) cost
- If you make 1000 calls/texts, each one effectively costs $0.10
- The rate at which money leaves your pocket is very “spiky”
- But each call or text appears to have basically a constant cost: the *amortized* cost per text is $O(1)

*assumes unlimited calls/texts

Common Amortized Complexity

Analyze the asymptotic runtime complexity of the `push` operation for a stack implemented using an array/vector

Method	Implementation
`push(object)`	1. If needed, allocate a bigger array and copy data 2. Add new element at `top_ptr`, increment `top_ptr`

Container Growth Options

1. Constant Growth
   - When container fills, increase size by *c*
   - Amortized cost: $\frac{(1 + \Theta(n)) + c * \Theta(1)}{c \text{ push operations}} = \Theta(n)$
   - Amortized linear
2. Linear Growth
   - When container fills, increase size by *n*
   - Amortized cost: $\frac{(1 + \Theta(n)) + n * \Theta(1)}{n \text{ push operations}} = \Theta(1)$
   - Amortized constant

Complexity Analysis: Amortized Complexity

Data Structures & Algorithms

Complexity Analysis: Balance Exercise

Data Structures & Algorithms

Exercise

- You have *n* billiard balls. All have equal weight, except for one which is heavier. Find the heavy ball using only a balance.
- Describe an $O(n^2)$ algorithm
- Describe an $O(n)$ algorithm
- Describe an $O(\log n)$ algorithm
- Describe another $O(\log n)$ algorithm

Two $O(\log n)$ solutions

- Two groups: $\log_2(n) = O(\log_3 n)$
- Three groups: $\log_3(n) = O(\log_2 n)$
- True or false? Why?

Complexity Analysis: Balance Exercise

Data Structures & Algorithms