Apply to join CSE’s Undergraduate Mentoring Program!

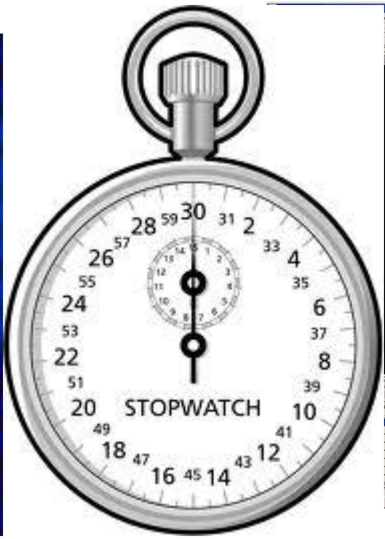
The purpose of this program is to increase students’ sense of community and retention in CSE as they take the core intro courses.

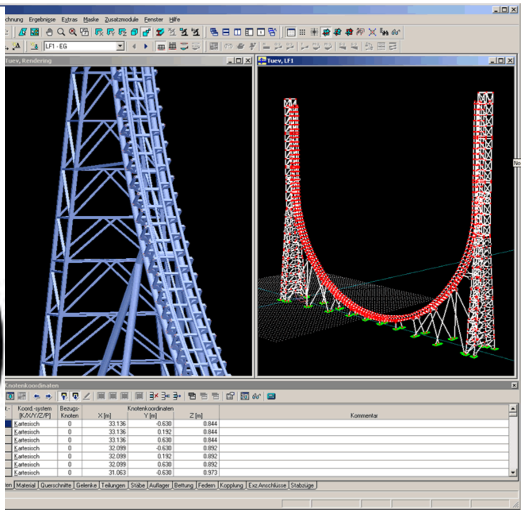
Please note - this is not a tutoring service!

Students can expect the following:

- Individual mentorship with junior & senior CSE students
- Develop community within a cohort over the semester
- Get easier access to CSE & CoE resources and faculty
- Participate in social activities (game nights, alumni meetup, catered study session, etc.) with fellow mentees

Lecture 3
Complexity Analysis





t.-	Koord.-system [K/X/Y/Z/P]	Bezugs-Knoten	Knotenkoordinaten X [m]	Y [m]	Z [m]	Kommentar
	Kartesisch	0	33.136	-0.630	0.844	
	Kartesisch	0	33.136	0.192	0.844	
	Kartesisch	0	33.136	0.630	0.844	
	Kartesisch	0	32.099	-0.630	0.892	
	Kartesisch	0	32.099	0.192	0.892	
	Kartesisch	0	32.099	0.630	0.892	
	Kartesisch	0	31.063	-0.630	0.973	

EECS 281: Data Structures & Algorithms

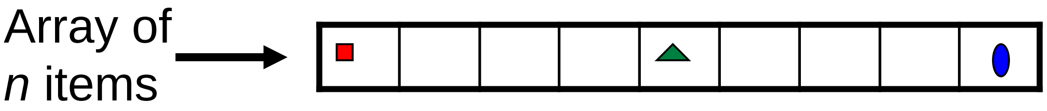
Complexity Analysis: Overview

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


Array of
n items

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)

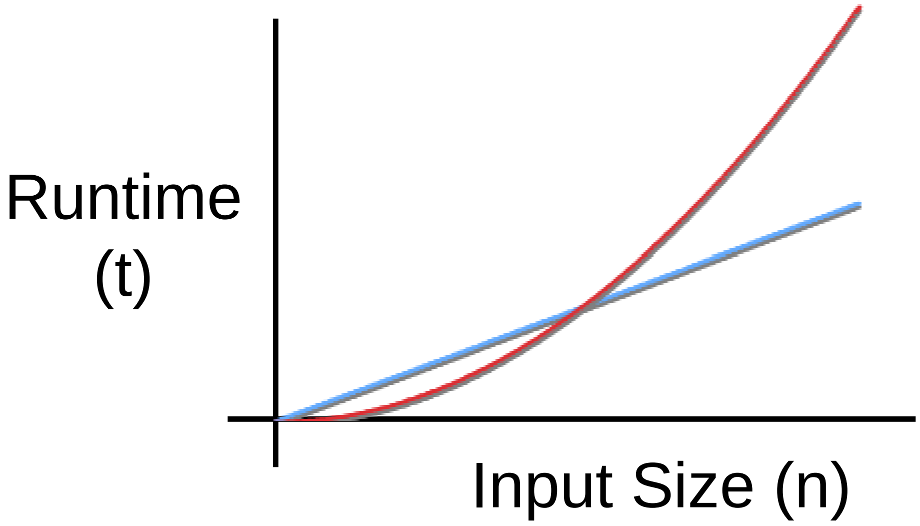
  `g++ -g3` (for debu**g**ging, highest level of information)

  `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?

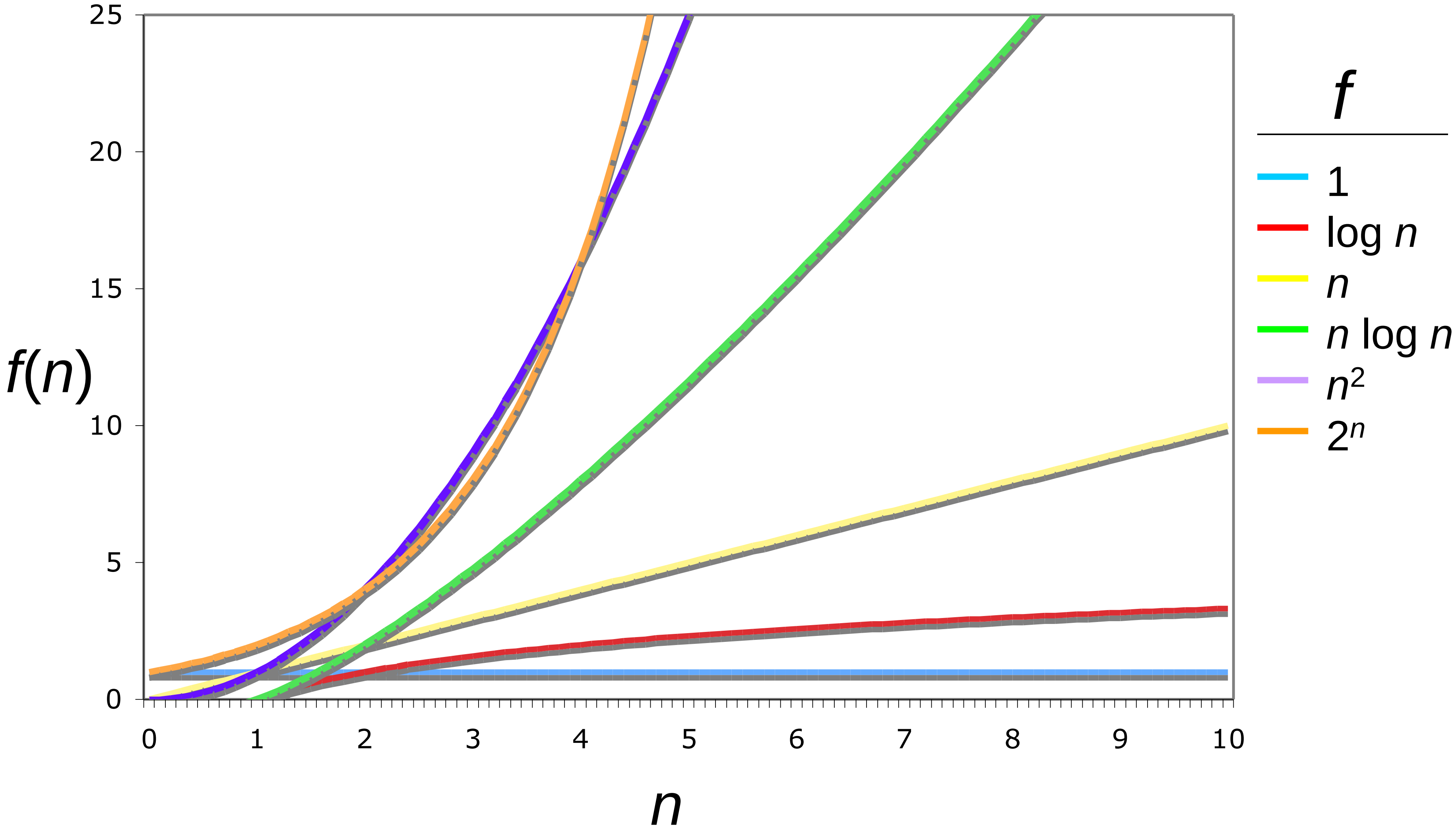
Measuring & Using Input Size

- 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

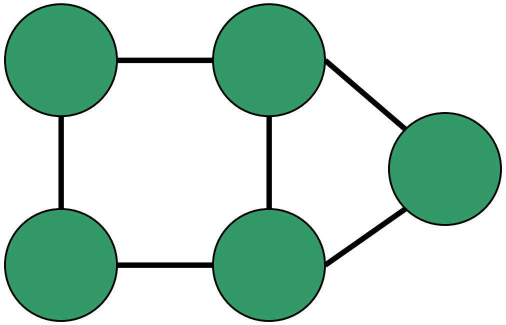


Input Size Example

Graph $G = (V, E)$:

V = 5 Vertices

E = 6 Edges



What should we measure?

- Vertices?
- Edges?
- Vertices and Edges?

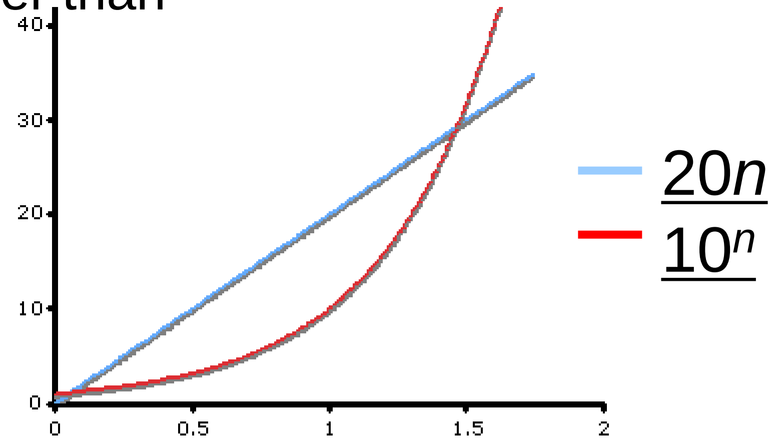
When in doubt, measure input size in bits

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

- Variable assignment
- Arithmetic operation
- Comparison
- Array indexing or pointer reference
- Function call (not counting the data)
- 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 + 2*n* steps
4	1 step
5	
6	1 step
7	

Total steps: 4 + 3*n*

```
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 + 2*n* steps
11	1 + 1 + 2*n* steps
12	1 step
13	
14	1 + 1 + 2*n* steps
15	1 step
16	
17	1 step
18	

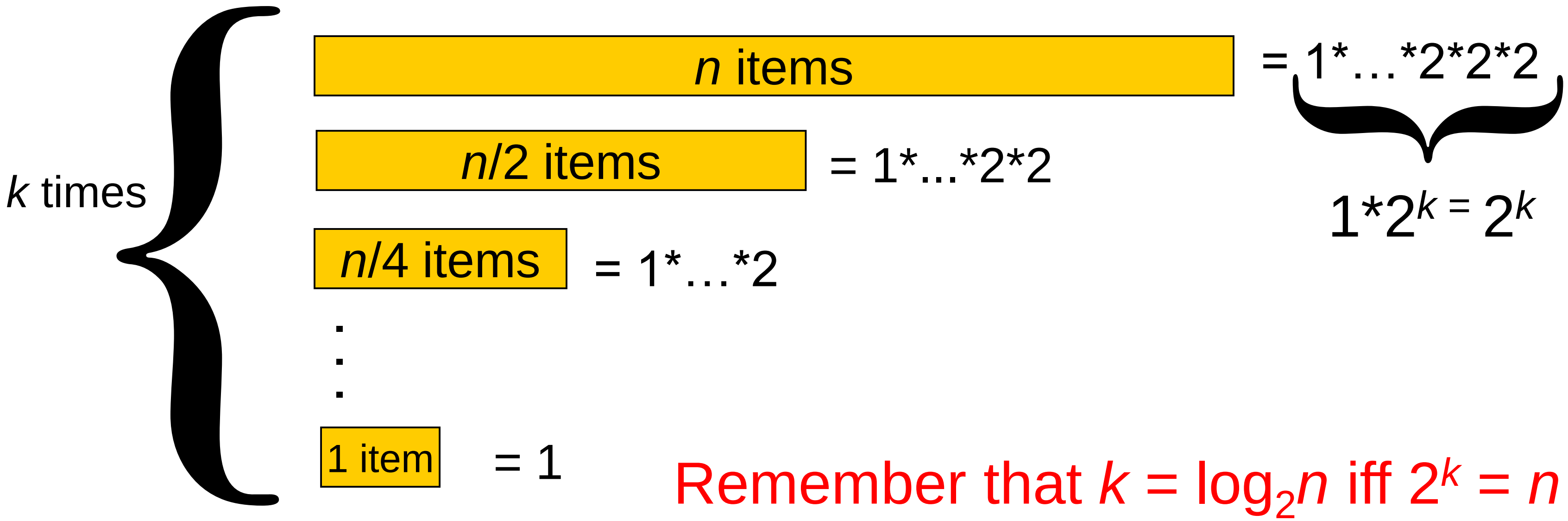
Total steps: $3n^2 + 7n + 6$

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 $1 + 1 + \sim\log n * (2 \text{ steps})$
4   1 step
5
6 1 step
7

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



k times

n items $= \underbrace{1*\ldots*2*2*2}_{1*2^k = 2^k}$

$n/2$ items $= 1*\ldots*2*2$

$n/4$ items $= 1*\ldots*2$

⋮

1 item $= 1$

Remember that $k = \log_2 n$ iff $2^k = n$

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

```
uint32_t logB(uint32_t n) {
  // find binary log, round up
  uint32_t r = 0;
  while (n > 1) {
    n /= 2
    r++;
  }  // while
  return r;
}  // logB()
```

```
int *bsearch(int *lo, int *hi, int val) {
  // find position of val between lo,hi
  while (hi >= lo) {
    int *mid = lo + (hi - lo) / 2;
    if (*mid == val)
      return mid;
    else if (*mid > val)
      hi = mid - 1;
    else
      lo = mid + 1;
  }  // while
  return nullptr;
}  // bsearch()
```

Algorithm Exercise

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

```
// REQUIRES: in and out are arrays with size elements
// MODIFIES: out
// EFFECTS: out[i] = in[0] *...* in[i-1] * in[i+1] *...* in[size-1]
void f(int *out, const int *in, int size) {
  for (int i = 0; i < size; ++i) {
    out[i] = 1;
    for (int j = 0; j < size; ++j) {
      if (i != j)
        out[i] *= in[j];
    }  // for j
  }  // for i
}  // f()
```

Algorithm Exercise

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

```
void f(int *out, const int *in, int size) {
  int product = 1;
  for (int i = 0; i < size; ++i)
    product *= in[i];

  for(int i = 0; i < size; ++i)
    out[i] = product / in[i];
}  // f()
```

Complexity Analysis: Counting Steps

Data Structures & Algorithms

Complexity Analysis:
Big-O Notation

Data Structures & Algorithms

Big-O Definition

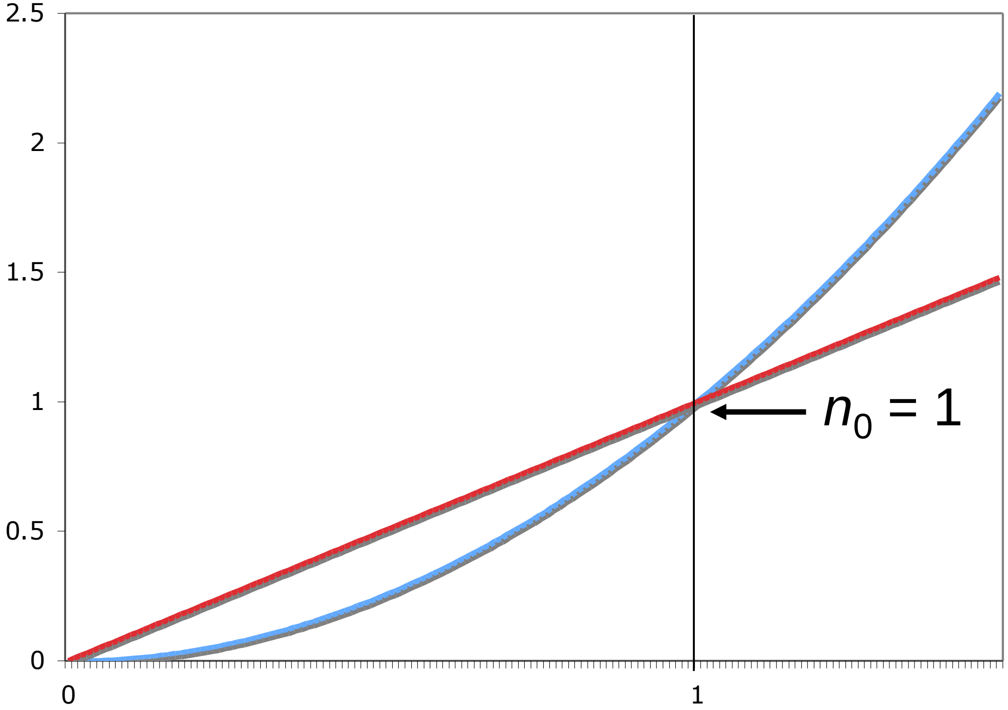
$f(n) = O(g(n))$ if and only if there are constants

$$\left.\begin{array}{l} c > 0 \\ n_0 \geq 0 \end{array}\right\} \text{ such that } f(n) \leq c * g(n) \text{ whenever } n \geq n_0$$

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

🟥 $f(n) = n$

🟦 $g(n) = n^2$



$n_0 = 1$

Big-O: Sufficient (but not necessary) Condition

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

$$\boxed{\log_2 n = O(2\text{n})?}$$

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

$f(n) = \log_2 n$

$g(n) = 2n$

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

$0 = d < \infty$

: $\log_2 n = O(2n)$ ✓

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

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

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

$g(n) = 100$

Big-O: Can We Drop Constants?

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

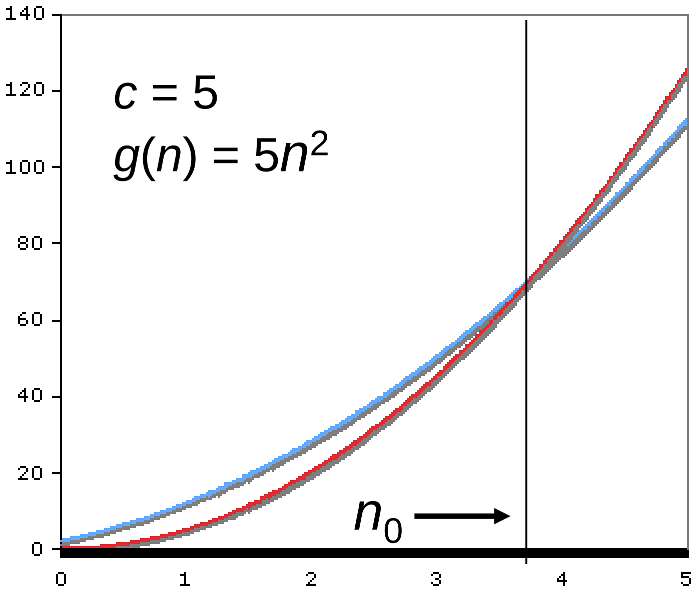
$f(n) = 3n^2 + 7n + 42$

$g(n) = n^2$

Definition

c > 0, n_0 ³ 0 such that

f(n) £ c ×g(n) whenever n ³ n_0



$c = 5$

$g(n) = 5n^2$

Sufficient Condition

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

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

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

$$\lim_{n\to\infty}\left(\frac{6}{2}\right) = 3 < \infty \checkmark$$

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
$\log_a(xy) = \log_a x + \log_a y$	$\log_2 (3*4)$
$\log_a\left(\frac{x}{y}\right) = \log_a x - \log_a y$	$\log_2 (4/3)$
$\log_a(x^r) = r \log_a x$	$\log_2 x^3$
$\log_a\left(\frac{1}{x}\right) = -\log_a x$	$\log_2 1/3$
$\log_a x = \frac{\log x}{\log a} = \frac{\ln x}{\ln a}$	$\log_7 9$
$\log_a a = 1$ $\log_a 1 = 0$	

Power Identities

Identity	Example
$a^{(m+n)} = a^m \cdot a^n$	2^{2+3}
$a^{(m-n)} = \frac{a^m}{a^n}$	2^{3-2}
$(a^m)^n = a^{mn}$	$(2^2)^3$
$a^{-n} = \frac{1}{a^n}$	2^{-4}
$a^{-1} = \frac{1}{a}$ $a^0 = 1$ $a^1 = a$	

Exercise

True or False?

- $10^{100} = O(1)$
- $3n^4 + 45n^3 = O(n^4)$
- $3^n = O(2^n)$
- $2^n = O(3^n)$
- $45 \log(n) + 45n = O(\log(n))$
- $\log(n^2) = O(\log(n))$
- $[\log(n)]^2 = O(\log(n))$

Can you?

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	n_0$ Î Z,$c Î R: " n ³ n$_0$,f(n) £ c ×g(n)	$\Theta(f(n)) = O(f(n)) \bigcap \Omega(f(n))$	n_0$ Î Z,$c Î R: " n ³ n$_0$,f(n) ³ c ×g(n)
$f_1(n)=2n+1$	$O(n)$ or $O(n^2)$ or $O(n^3)$…	Q(n)	$\Omega(n)$ or $\Omega(1)$
$f_2(n)=n^2+n+5$	$O(n^2)$ or $O(n^3)$ …	Q(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`

Exercise

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

Amortized $\Theta(1)$

- Assume vector is filled with ***n*** elements
- Double vector size (1 + $\Theta(n)$) steps
- $\Theta(1)$ push ***n*** times until full
- Amortized cost: $\frac{(1 + \Theta(n)) + n_* \Theta(1)}{n \text{ push operations}} = \Theta(1)$

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