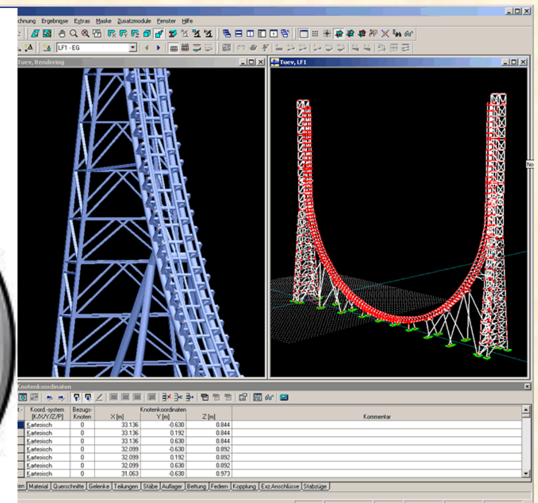
Lecture 2
Complexity Analysis



EECS 281: Data Structures & Algorithms

Assignments

- First reading assignment (now)
  - CLRS chapter 1 (short)
  - Link to textbook available on Canvas through the Syllabus

What Affects Runtime?

- The algorithm
- Implementation details
  - Skills of the programmer
- CPU Speed / Memory Speed
- Compiler (Options used)

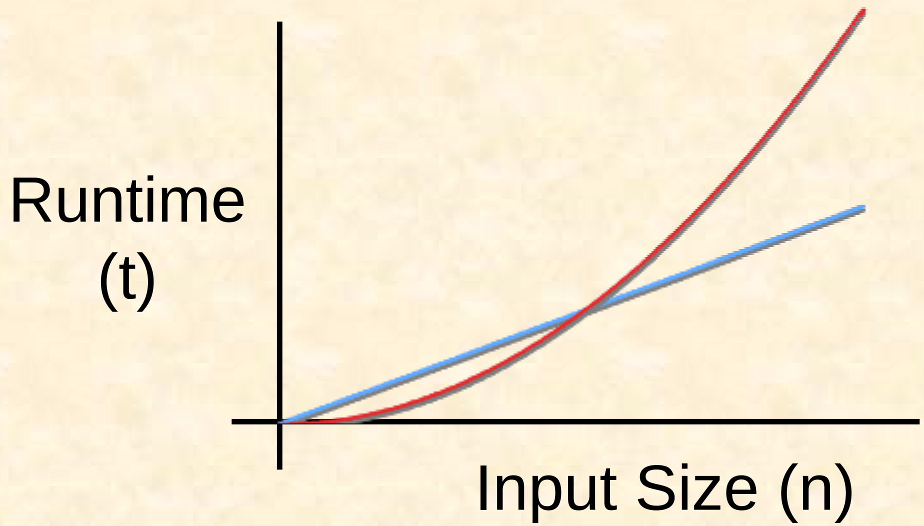
  `g++ -g3` (for debu**g**ging, highest level of information)

  `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?

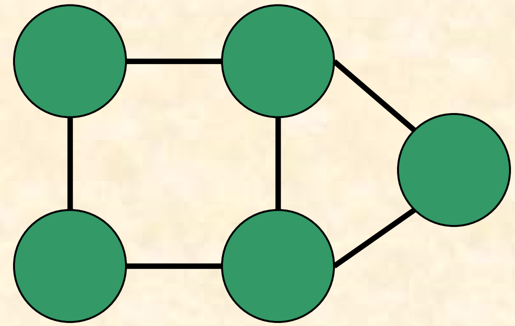
- Notation and terminology
  - *n* = input Size
  - *f(n)* = max number of steps ("*f* of *n*")
    taken by an algorithm
    when input has length *n*
  - *O(f(n))* = complexity class of *f*(*n*) ("Big-*O* of *f* of *n*")

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

n = V + E

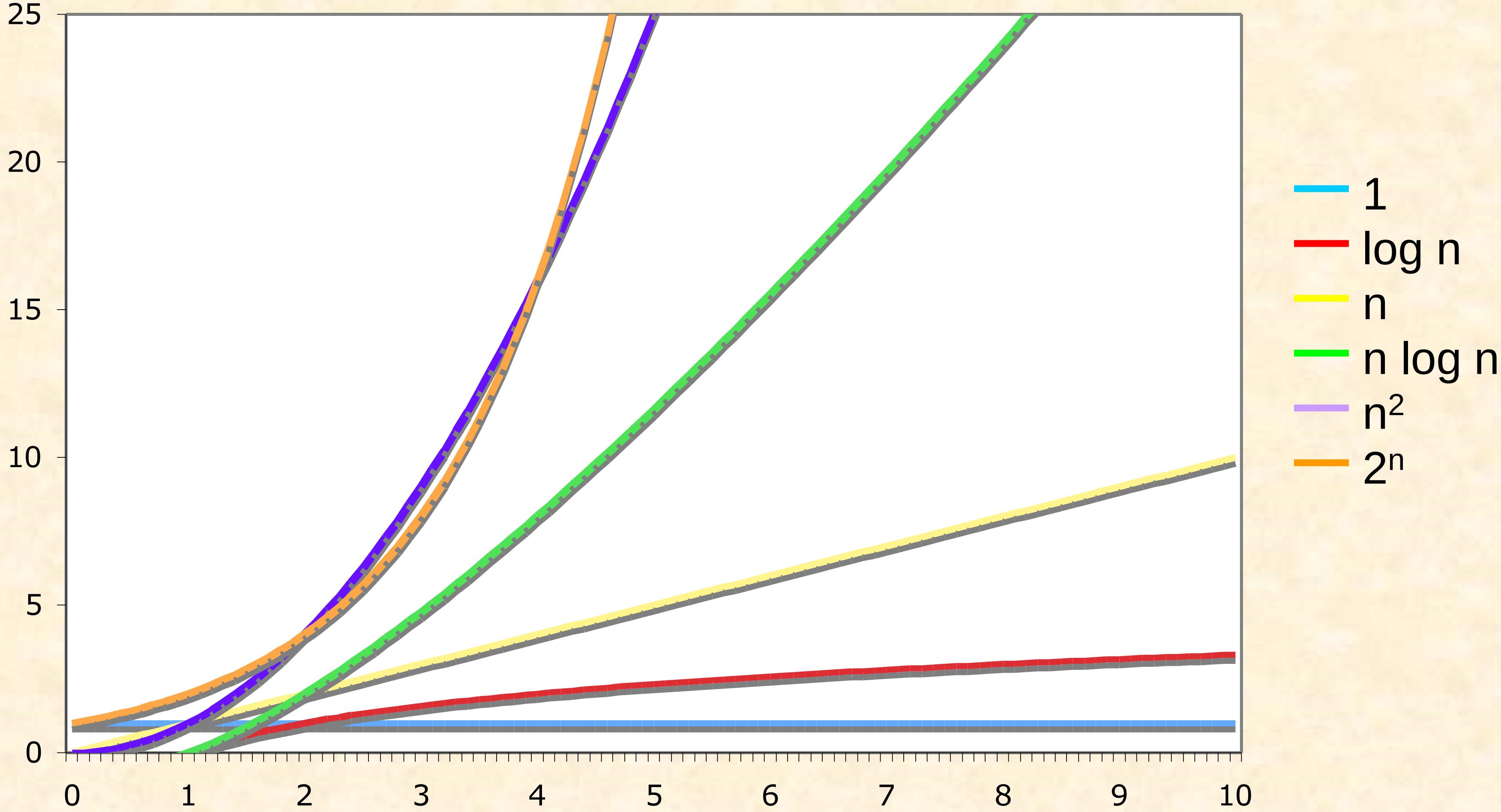
Using V and E tells which contributes more to the total number of steps

Examples: E log V, EV, V^2 log E

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

Examples of f(n) Runtime



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

```
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()
```

Line	Steps
1	
2	1 step
3	1 + 1 + n * (2 steps)
4	1 step
5	
6	1 step
7	

Total steps: 4 + 3n

```
1  int func2(int n) {
2    int sum = 0;
3    for (int i = 0; i < n; i++) {
4        for (int j = 0; j < n; j++)
5            sum++;
6    } // for i
7    for (int k = 0; k < n; k++) {
8        sum--;
9    } // for
10   return sum;
11 } // func2()
```

Line	Steps
1	
2	1 step
3	1 + 1 + n * (2 steps)
4	1 + 1 + n * (2 steps)
5	1 step
6	
7	1 + 1 + n * (2 steps)
8	1 step
9	
10	1 step
11	

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

Counting Steps: `for` Loop

- Remember the basic form of the 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)

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?

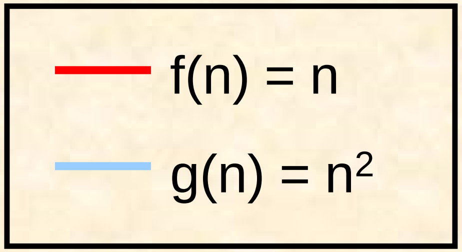
```
void f(int *out, const int *in, int size) {
  int product = 1;
  for (int i = 0; i < size; ++i)
    product *= in[i];

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

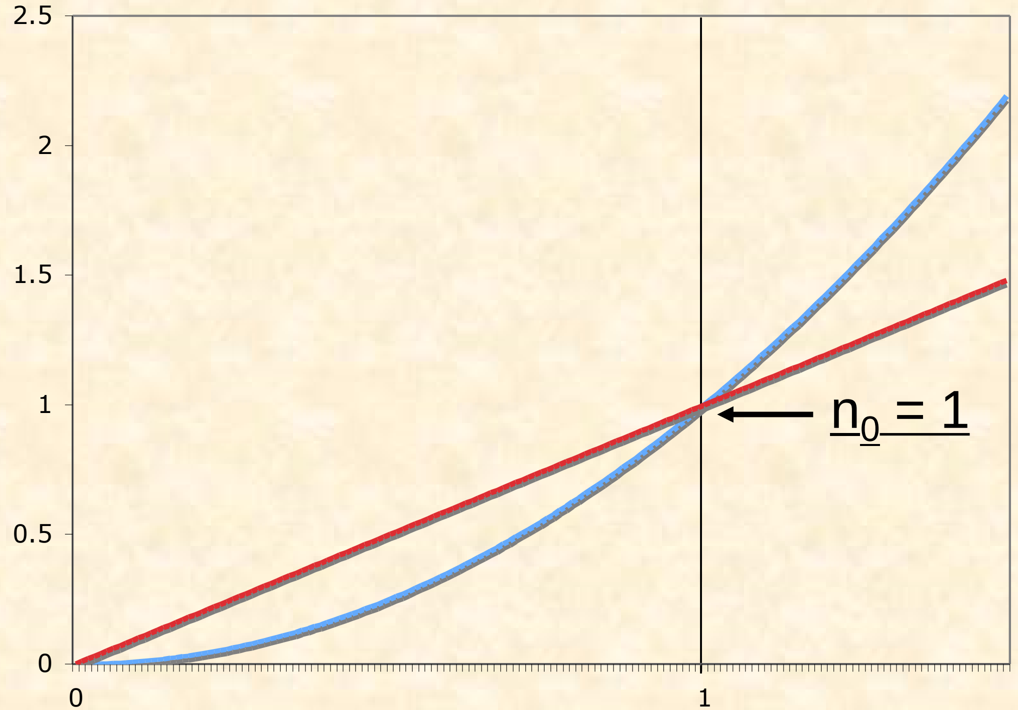
Big-O - Definition 1

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 $n = O(n^2)$?





Big-O: Sufficient (but not necessary) Condition

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

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

$f(n) = \log_2 n$

$g(n) = 2n$

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

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

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

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

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

$g(n) = 100$

$$\lim_{n\to\infty}\left(\frac{\sin\left(\frac{n}{100}\right)}{100}\right)$$

: Condition does not hold but it is true that $f(n) = O(g(n))$

Big-O: Can We Drop Constants?

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

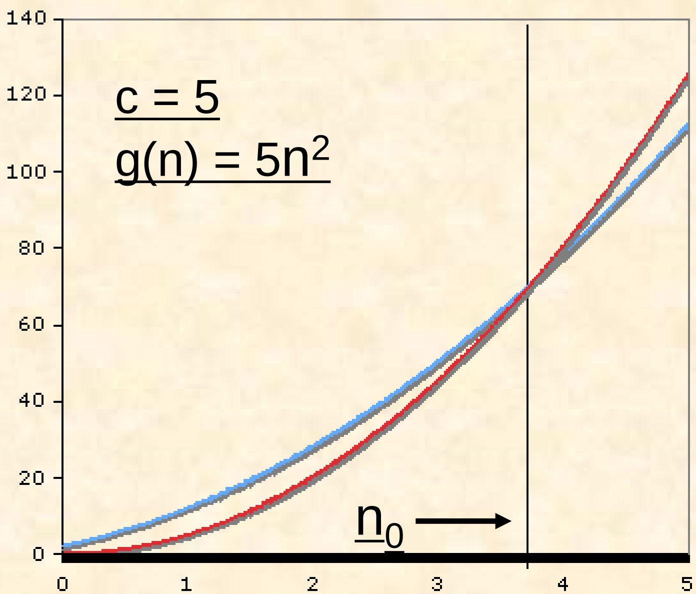
$f(n) = 3n^2 + 7n + 42$
$g(n) = n^2$

Definition	Sufficient Condition

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 (Q)	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) = Θ(g(n)) if and only if there exists an integer n_0 and real constants c1 and c2 such that for all $n \geq n_0$: $c1 \cdot g(n) \leq f(n) \leq c2 \cdot g(n)$	f(n) = Ω(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	$n0 Î Z, $c Î R : " n ³ n0, f(n) £ c ×g(n)	$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$	$n0 Î Z, $c Î R : " n ³ n0, f(n) ³ c ×g(n)
$f_1(n)=2n+1$	O(n) or $O(n^2)$ or $O(n^3)$…	Q(n)	Ω(n) or Ω(1)
$f_2(n)=n^2+n+5$	$O(n^2)$ or $O(n^3)$ …	$Q(n^2)$	$\Omega(n^2)$ or Ω(n) or Ω(1)

Example: $O(\log n)$ Time

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

Line	Steps
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)

k times

n items = 1*…*2*2*2

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

n/2 items = 1*...*2*2

n/4 items = 1*…*2

⋮

1 item = 1

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

Additional Examples of $O(\log n)$ Time

```
unsigned logB(unsigned n) {
  // find binary log, round up
  unsigned 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()
// Q: how can this code be optimized ?
```

Complexity Analysis

- What is it?
  - Each step should take O(1) time
  - Given an algorithm and input size n, how many steps are needed?
  - As input size grows, how does number of steps change?
    - Focus is on TREND
- How do we measure it?
  - Express the rate of growth as a function f(n)
  - Use the big-O notation
- Why do we care?
  - Tells us 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

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

Best-Case: 1 step
Worst-Case: n steps
Average-Case: n/2 steps

- Best-Case
  - Least number of steps 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 steps 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 steps required, given any input
  - Average performed over all possible inputs of a given size

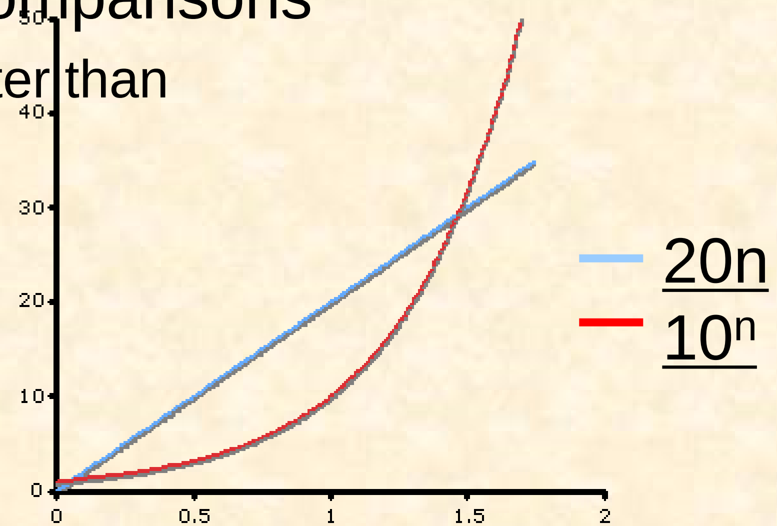
Amortized Complexity

- A type of worst-case complexity
- Analysis performed over a sequence of inputs of a given size
  - The sequence is selected to be a worst case
- Considers the average cost of one step over a sequence of operations
  - Best/Worst/Average-case only consider a single operation
  - <u>Different from average-case complexity</u>!
- Key to understanding expandable arrays and STL's vector class, STL's implementations of stacks, queues, priority queues, hash tables

- Example: pre-paid telephone cards
  - Pay $20 upfront and call many times, until $20 is exhausted
  - Amortizes to, say, 10c per minute (then recharge with another $20)
  - Better than paying for each call at international rates
  - Worst-case sequences of calls: any sequence that exhausts $20
  - Sequences that do not require recharge are not worst-case sequences

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?



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

- True or false? Why?
- $\log_3(n) = O(\log_2 n)$
- $\log_2(n) = O(\log_3 n)$

Job Interview Question

- Implement this function

```
//returns x^n
int power(int x, unsigned int n);
```

- The obvious solution uses n - 1 multiplications
  - $2^8 = 2*2* \ldots *2$
- Less obvious: O(log **n**) multiplications
  - Hint: $2^8 = ((2^2)^2)^2$
  - Does it work for 2^7 ?
- Write two solutions