COSC 222 Data Structures

Algorithm Analysis

Efficiency

- **Measure of efficiency** is needed to compare one algorithm to another (assuming that both algorithms are correct and produce the same answers)

- Suggest some ways of how to measure efficiency
  - Time (How long does it take to run?)
  - Space (How much memory does it take?)
  - Other attributes?

    Expensive operations, e.g. I/O

    Energy, Power

    Ease of programming, legal issues, etc.

Analyzing Runtime

```
old2 = 1;
old1 = 1;
results = 0;
for (i=3; i<n; i++) {
      result = old2+old1;
      old1 = old2;
      old2 = result;
}
```

How long does this take?

Analyzing Runtime

- A simple mechanism: **`currentTimeMillis`** method of the System class

```
long startTime = System.currentTimeMillis( ); // record the starting time

/* (run the algorithm) */

long endTime = System.currentTimeMillis( ); // record the ending time
long elapsed = endTime – startTime; // compute the elapsed time
```

- Limitation: the measured times will vary greatly from machine to machine

Example

- Two algorithms for constructing long strings in Java

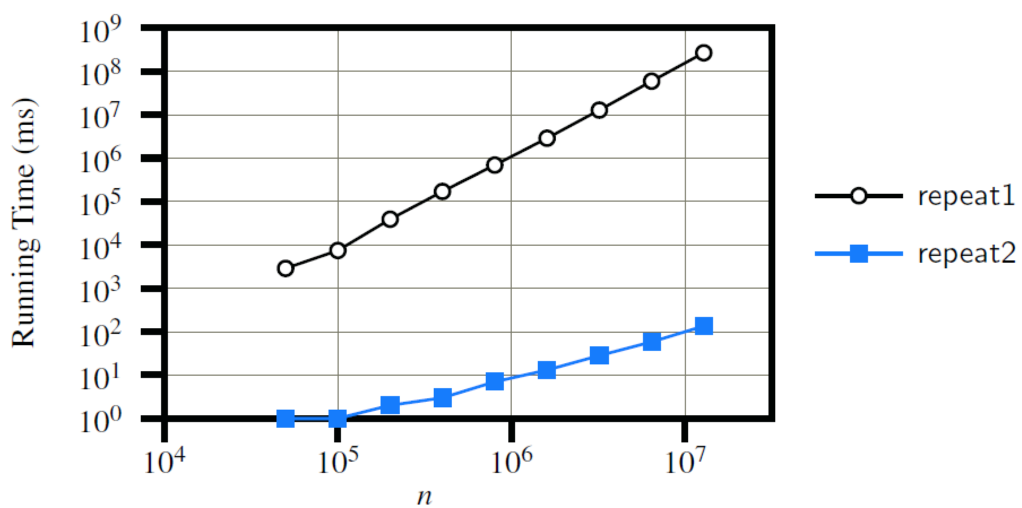
```java
/** Uses repeated concatenation to compose a String with n
copies of character c. */
 public static String repeat1(char c, int n) {
 String answer = "";
 for (int j=0; j < n; j++)
    answer += c;
 return answer;
 }

/** Uses StringBuilder to compose a String with n copies of
character c. */
public static String repeat2(char c, int n) {
 StringBuilder sb = new StringBuilder( );
 for (int j=0; j < n; j++)
    sb.append(c);
 return sb.toString( );
 }
```

Example

- **`repeat1`** is already taking more than 3 days to compose a string of 12.8 million characters, **`repeat2`** is able to do the same in a fraction of a second
- As the value of n is doubled, the running time of **`repeat1`** typically increases more than **`fourfold`**, while the running time of **`repeat2`** approximately **`doubles`**

n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,874,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421	135



Limitations of Experiments

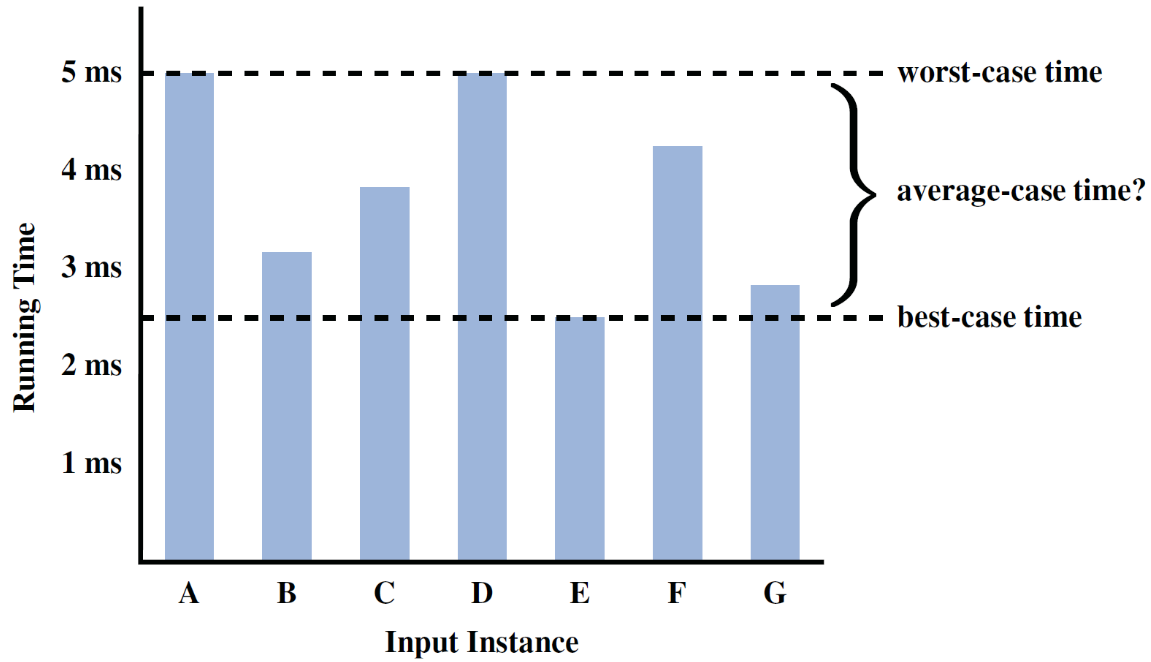
- While experimental studies of running times are valuable, there are three major limitations to their use for algorithm analysis:
  - It is necessary to **implement** the algorithm, which may be difficult
  - Experiments can be done only on a limited **set of test inputs**; hence, they leave out the running times of inputs not included in the experiment
  - In order to compare two algorithms, the same **hardware and software environments** must be used

Theoretical Analysis

- Evaluate the efficiency of an algorithm independent of the hardware/software environment
- Uses a high-level description of the algorithm instead of an implementation
- Takes into account all possible inputs

Theoretical Analysis

- In order to analyze the time complexity of an algorithm:



  - Consider the **worst-case** scenario
  - Count the **number of operations**
  - Express the number **as a function of input size n**

Number of Operations

- What is meant by "number of operations"?
  - Assigning a value to a variable
  - Following an object reference
  - Performing an arithmetic operation (for example, adding two numbers)
  - Comparing two numbers
  - Accessing a single element of an array by index
  - Calling a method
  - Returning from a method

Analyzing Runtime

```
old2 = 1;
old1 = 1;
results = 0;
for (i=3; i<n; i++) {
      result = old2+old1;
      old1 = old2;
      old2 = result;
}
```

How many operations does this take?

IT DEPENDS

- What is n?

- Running time is a function of n such as **T(n)**
- This is really nice because the runtime analysis doesn't depend on hardware or subjective conditions anymore

Input Size

- What is meant by the input size n? Provide some application-specific examples.
- Dictionary:
  - # words
- Restaurant:
  - # customers or # food choices or # employees
- Airline:
  - # flights or # luggage or # costumers

- We want to express the number of operations performed as a function of the input size n.

The Constant Function

- The simplest function we can think of is the ***constant function***, that is,

$$f(n) = c$$

- For any argument n, the constant function $f(n)$ assigns the value c.
- In other words, $f(n)$ will always be equal to the constant value c.

The Logarithm Function

- One of the interesting and sometimes even surprising aspects of the analysis of data structures and algorithms is the ubiquitous presence of the ***logarithm function***,

$$f(n) = \log_b n, \text{ for some constant } b > 1.$$

- The value b is known as the ***base*** of the logarithm.
- This base is common, we will typically omit it from the notation when it is 2. That is, for us,

$$\log n = \log_2 n.$$

Logarithm Rules

- Given real numbers $a > 0$, $b > 1$, $c > 0$, and $d > 1$, we have:

1. $\log_b(ac) = \log_b a + \log_b c$
2. $\log_b(a/c) = \log_b a - \log_b c$
3. $\log_b(a^c) = c \log_b a$
4. $\log_b a = \log_d a / \log_d b$
5. $b^{\log_d a} = a^{\log_d b}$

The Linear Function

- Another simple yet important function is the ***linear function***,

$$f(n) = n.$$

- That is, given an input value n, the linear function f assigns the value n itself.

The N-Log-N Function

- The function that assigns to an input n the value of n times the logarithm base-two of n

$$f(n) = n\log n,$$

- This function grows a little more rapidly than the linear function and a lot less rapidly than the quadratic function

The Quadratic Function

- Given an input value n, the function f assigns the product of n with itself

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

- There are many algorithms that have nested loops
  - the inner loop performs a linear number of operations
  - the outer loop is performed a linear number of times
  - Thus, the algorithm performs $n \cdot n = n^2$ operations.

The Cubic Function

- An input value n the product of n with itself three times

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

- The cubic function appears less frequently in the context of algorithm analysis than the constant, linear, and quadratic functions

Polynomials

- The linear, quadratic and cubic functions can each be viewed as being part of a larger class of functions, the ***polynomials***.

- A ***polynomial*** function has the form

$$f(n) = a_0+a_1n+a_2n^2+a_3n^3+\cdots+a_dn^d$$

where $a_0, a_1, \ldots, a_d$ are constants, called the ***coefficients*** of the polynomial.

- The following functions are all polynomials:

$f(n) = 2+5n+n^2$

$f(n) = 1+n^3$

$f(n) = 1$

$f(n) = n$

$f(n) = n^2$

The Exponential Function

- Another function used in the analysis of algorithms is the ***exponential function***,

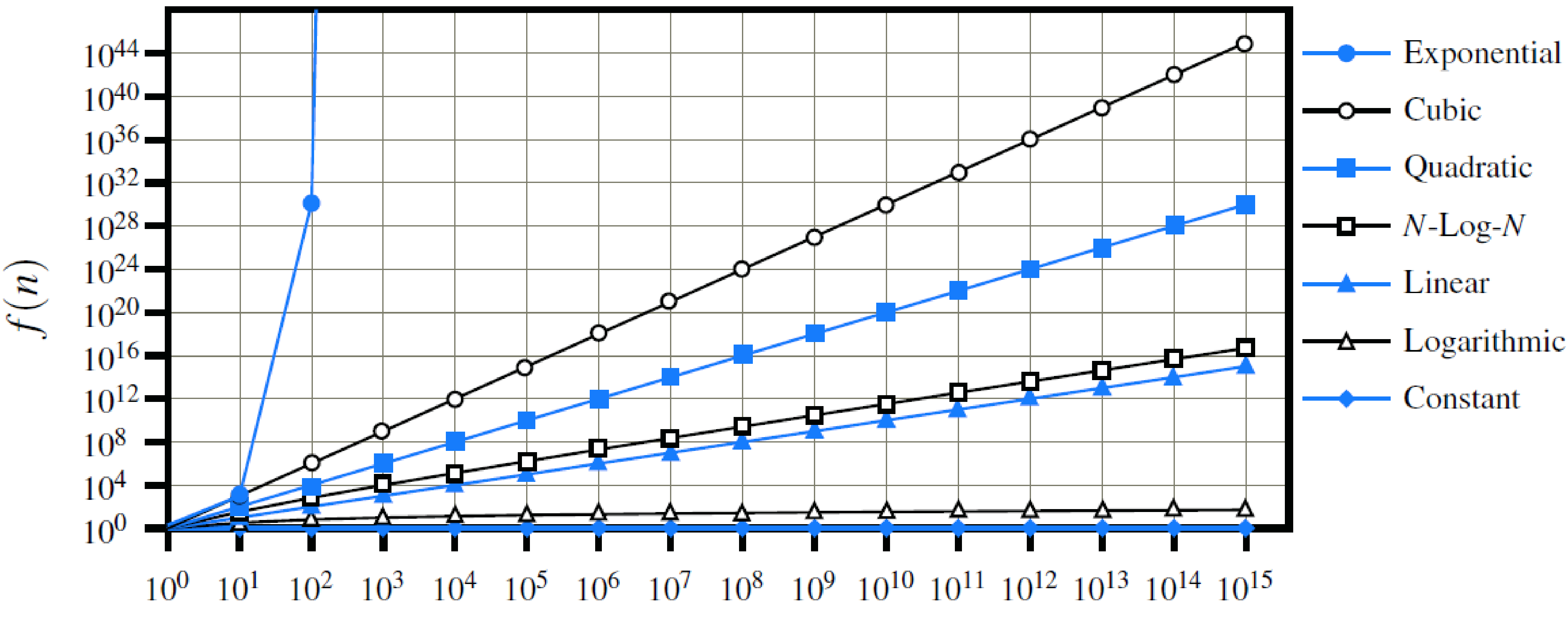
$$f(n) = b^n,$$

where b is a positive constant, called the ***base***, and the argument n is the ***exponent***.

Comparing Growth Rates

- The seven common functions used in algorithm analysis

constant	logarithm	linear	*n*-log-*n*	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n



Asymptotic Analysis

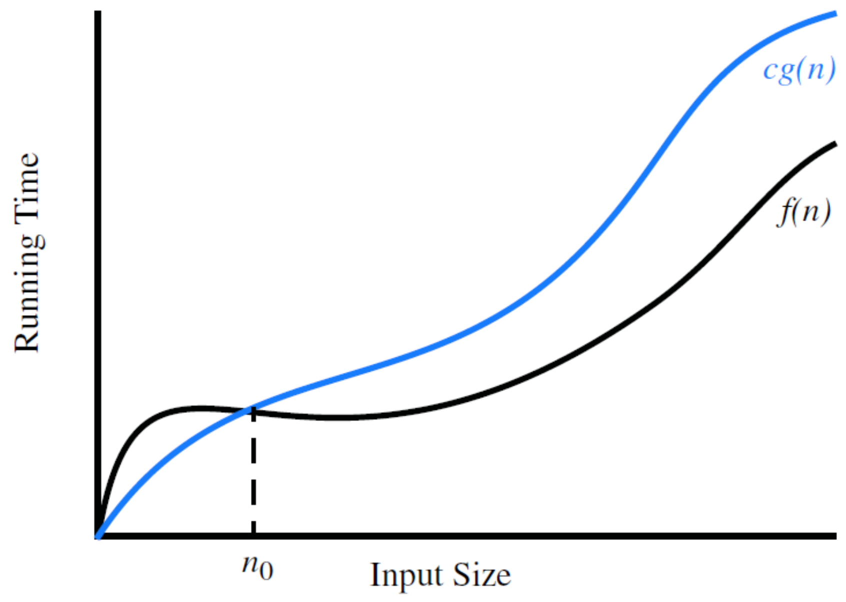
- We **focus on the growth rate** of the running time as a function of the **input size *n***
- This approach reflects that each basic step in a pseudocode description or a high-level language implementation may correspond to a small number of primitive operations
  - without capturing so many details
  - without worrying about what happens for small inputs

The “Big-Oh” Notation

- Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers.
- We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

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

- This definition is often referred to as the “big-Oh” notation, for it is sometimes pronounced as “ $f(n)$ is ***big-Oh*** of $g(n)$.”



Prove n log n ∈ O(n^2)

- We say f (n) is O(g(n)) if we can find f (n) ≤ c ·g(n), for n ≥ n_0.
- f (n) = n log n, O(g(n)) = O(n^2)
- Guess or figure out values of c and n_0 that will work.

$$n \log n \leq cn^2$$

$$\log n \leq cn$$

- This is fairly trivial: log n <= n (for n>1) c=1 and n0 = 1 works!

The "Big-Oh" Notation

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$

- Example: the function n^2 is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since c must be a constant

Try it Activity

- Prove $T(n) = n^3 + 20n + 1 \in O(n3)$
  - $n^3 + 20n + 1 \leq cn^3$ for $n > n_0$
  - $1 + 20/n^2 + 1/n^3 \leq c$ holds for $c=22$ and $n_0 = 1$

- Prove $T(n) = n^3 + 20n + 1 \in O(n^4)$
  - $n^3 + 20\ n + 1 \leq cn^4$ for $n > n_0$
  - $1/n + 20/n^3 + 1/n^4 \leq c$ holds for $c=22$ and $n_0 = 1$

- Prove $T(n) = n^3 + 20\ n + 1 \in O(n^2)$
  - $n^3 + 20\ n + 1 \leq cn^2$ for $n > n_0$
  - $n + 20/n + 1/n^2 \leq c$ You cannot find such c or n0

Asymptotic Analysis Hacks

- Eliminate low order terms
  - $4n + 5 \Rightarrow 4n$
  - $0.5\ n \log n - 2n + 7 \Rightarrow 0.5\ n \log n$
  - $2^n + n^3 + 3n \Rightarrow 2^n$

- Eliminate coefficients
  - $4n \Rightarrow n$
  - $0.5\ n \log n \Rightarrow n \log n$
  - $n \log (n^2) = 2\ n \log n \Rightarrow n \log n$

Typical Growth Rates in Order

- constant: $O(1)$
- logarithmic: $O(\log n)$ ($\log_k n$, $\log n^2 \in O(\log n)$)
- poly-log: $O(\log^k n)$ (= $O(\log n)^k$, k is a constant >1)
- Sub-linear: $O(n^c)$ (c is a constant, 0 < c < 1)
- linear: $O(n)$
- (log-linear): $O(n \log n)$ (usually called "n log n")
- (superlinear): $O(n^{1+c})$ (c is a constant, 0 < c < 1)
- quadratic: $O(n^2)$
- cubic: $O(n^3)$
- polynomial: $O(n^k)$ (k is a constant)
- exponential: $O(c^n)$ (c is a constant > 1)

Which One is faster?

Post #1

- $n^3 + 2n^2$
- $n^{0.1}$
- $n + 100n^{0.1}$

Post #2

$100n^2 + 1000$

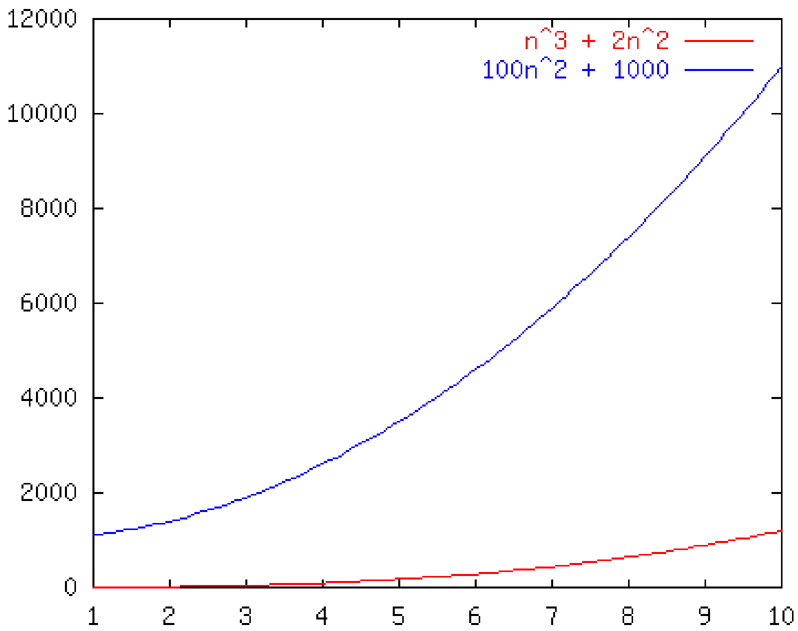
$\log n$

$2n + 10 \log n$

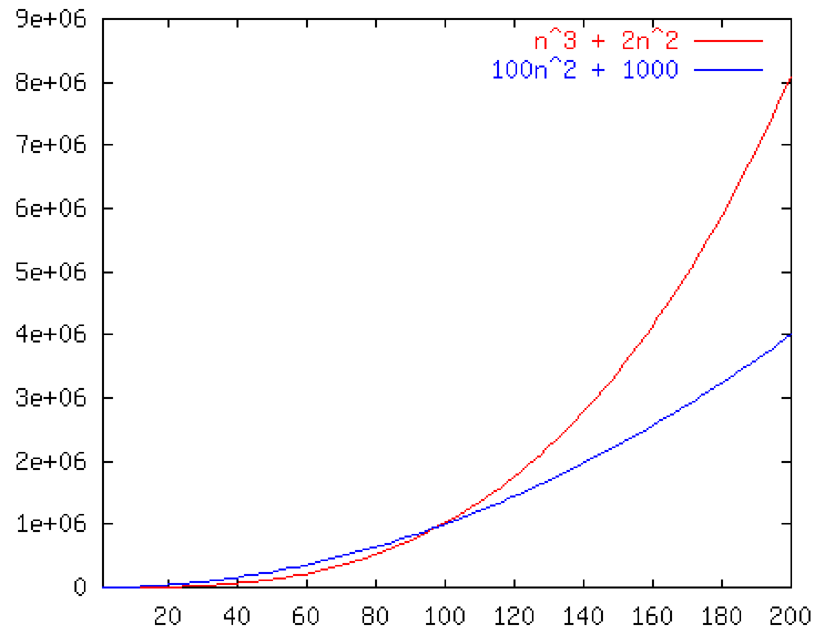
Note that faster means smaller, not larger!

Case 1

$n^3 + 2n^2$ vs. $100n^2 + 1000$



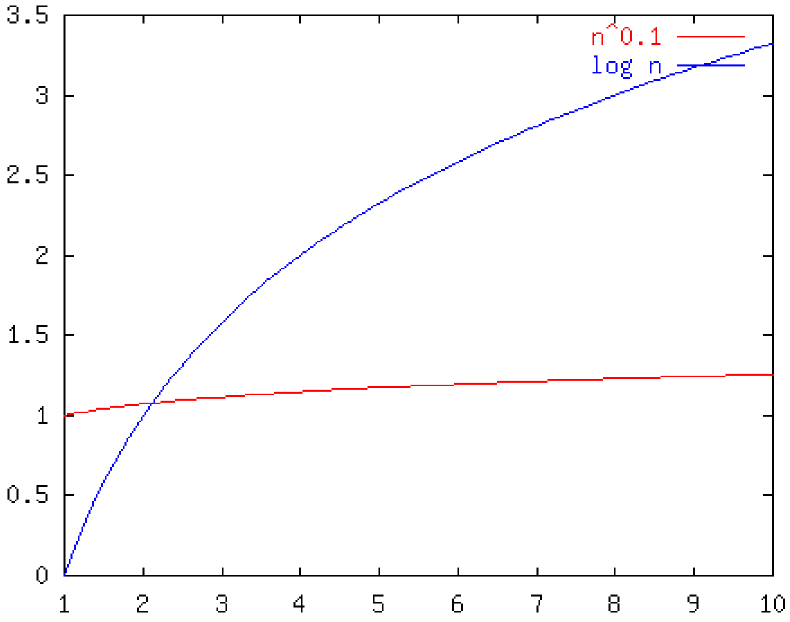
cubic: $O(n^3)$

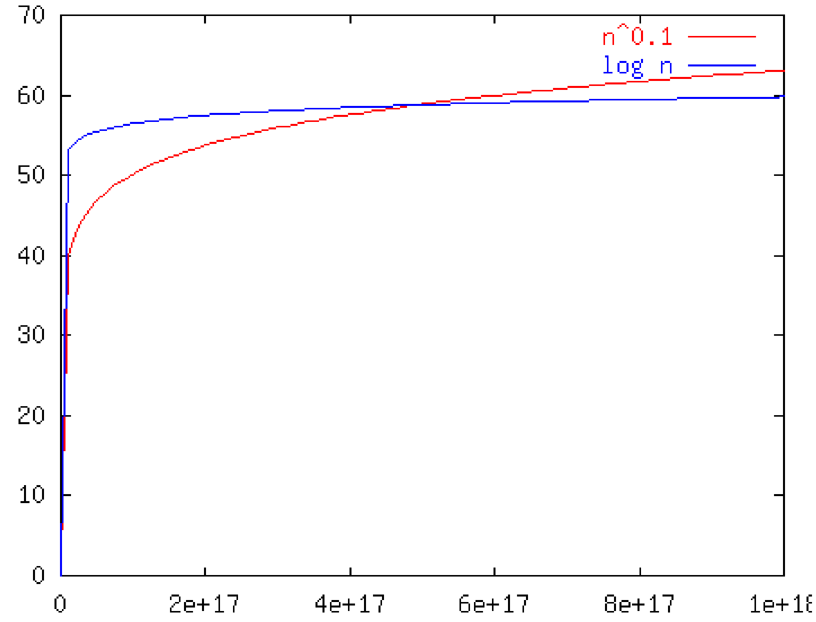


quadratic: $O(n^2)$

Case 2

$n^{0.1}$ vs. log n



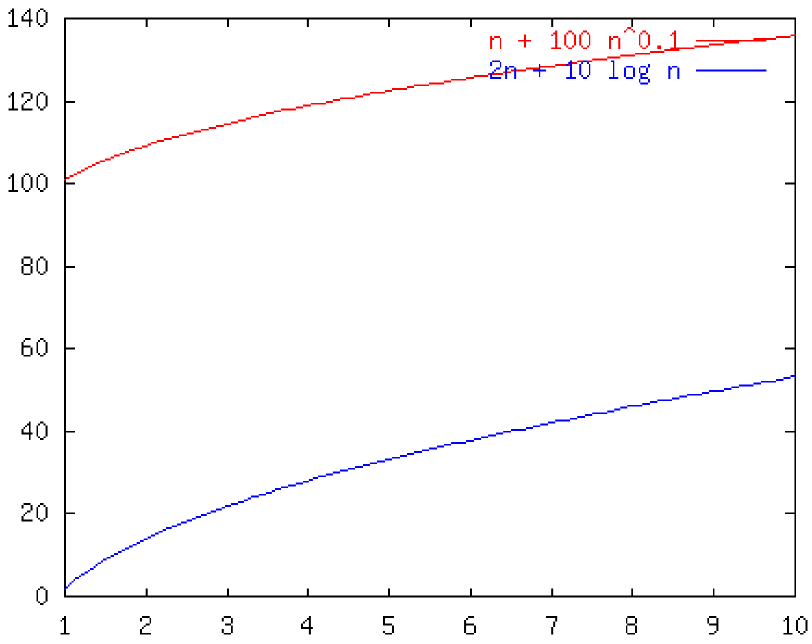


Sub-linear: $O(n^c)$ (c is a constant, $0 < c < 1$)

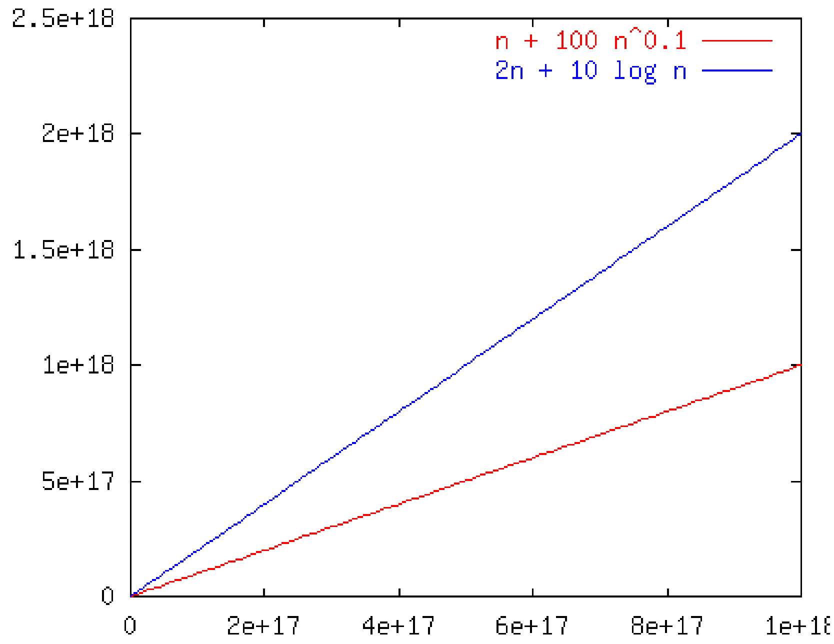
logarithmic: O(log n)

Case 3

$n + 100n^{0.1}$ vs. $2n + 10 \log n$



linear: O(n)



linear: O(n)

O(…) Examples

Let $f(n) = 3n^2 + 6n – 7$

- $f(n)$ is $O(n^2)$
- $f(n)$ is $O(n^3)$
- $f(n)$ is $O(n^4)$
- …

$f(n) = 4\ n \log n + 34\ n – 89$

- $f(n)$ is $O(n \log n)$
- $f(n)$ is $O(n^2)$

- If it's $O(n^2)$, it's also $O(n^3)$ etc! However, we always use the smallest one

Analyzing Code

```
for (int count = 0; count < n; count++)
  {
      /* some sequence of O(1) steps */
  }
```

O(n)

```
for (int count = 0; count < n; count+=2)
  {
      /* some sequence of O(1) steps */
  }
```

O(n)

Analyzing Code

```
count = 1
while (count < n)
{
count *= 2;
/* some sequence of O(1) steps */
}
```

O(logn)

count = 1, 2, 4, 8, 16, 32, …

$= 2^0, 2^1, 2^2, \ldots 2^k$

$n = 2^k$

or, $k = \log n$

Analyzing Code

```
for (int count = 0; count < n; count++)
   {
       for (int count2 = 0; count2 < n; count2++)
       {
         /* some sequence of O(1) steps */
       }
    }
```

$O(n^2)$

Analyzing Code

```
for (int count = 0; count < n; count++)
  {
      for (int count2 = count; count2 < n; count2++)
      {
          /* some sequence of O(1) steps */
      }
  }
```

$O(n^2)$

Analyzing Code

Algorithm 1

```
int x = 0;
int y = 0;
for (int i=0 ; i<n ; i++) {
  for (int j=0 ; j<n ; j++) {
    x += 2;
    y += x*2;
  }
}
```

Algorithm 2

```
int x = 0;
int y = 0;
for (int i=0 ; i<n ; i++)
  x += 2;
for (int j=0 ; j<n ; j++)
  y += 2*j;
```

Analyzing Code

Algorithm 1

```
int x = 0;
int y = 0;
for (int i=0 ; i<n ; i++) {
  for (int j=0 ; j<n ; j++) {
    x += 2;
    y += x*2;
  }
}
```

Algorithm 2

```
int x = 0;
int y = 0;
for (int i=0 ; i<n ; i++)
  x += 2;
for (int j=0 ; j<n ; j++)
  y += 2*j;
```

Algorithm 2 is asymptotically **faster** than Algorithm 1.

Relatives of Big-Oh

- **big-Omega**
- f(n) is Ω (g(n)) if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \geq c\, g(n) \text{ for } n \geq n_0$$

- **big-Theta**
- f(n) is Θ(g(n)) if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that

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

Questions?