# Fundamentals

CSE 326
Data Structures
Lecture 2

## Mathematical Background

- Today, we will review:
  - › Logs and exponents and series
  - › Asymptotics and order of magnitude notation
  - › Solving recursive equations



## Powers of 2

- Many of the numbers we use will be powers of 2
- Binary numbers (base 2) are easily represented in digital computers
  - › each "bit" is a 0 or a 1
  - › $2^0=1$, $2^1=2$, $2^2=4$, $2^3=8$, $2^4=16$, $2^8=256$, …
  - › an n-bit wide field can hold $2^n$ positive integers:
    - $0 \leq k \leq 2^n-1$



## Unsigned binary numbers

- Each bit position represents a power of 2
- For unsigned numbers in a fixed width field
  - › the minimum value is 0
  - › the maximum value is $2^n-1$, where n is the number of bits in the field
- Fixed field widths determine many limits
  - › 5 bits = 32 possible values ($2^5 = 32$)
  - › 10 bits = 1024 possible values ($2^{10} = 1024$)



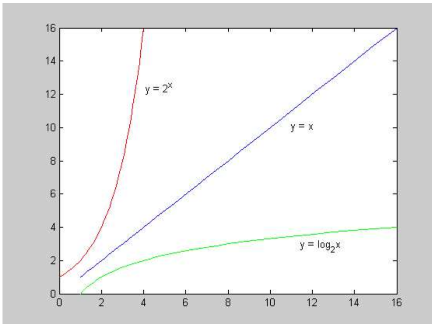
## Binary and Decimal

| $2^8=256$ | $2^7=128$ | $2^6=64$ | $2^5=32$ | $2^4=16$ | $2^3=8$ | $2^2=4$ | $2^1=2$ | $2^0=1$ | $\text{Decimal}_{10}$ |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | 1 | 1 | 3 |
| | | | | | 1 | 0 | 0 | 1 | 9 |
| | | | | | 1 | 0 | 1 | 0 | 10 |
| | | | | | 1 | 1 | 1 | 1 | 15 |
| | | | | 1 | 0 | 0 | 0 | 0 | 16 |
| | | | | 1 | 1 | 1 | 1 | 1 | 31 |
| | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 127 |
| | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 255 |



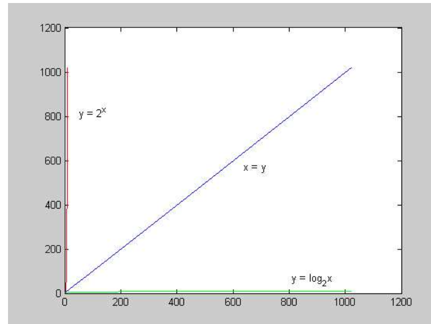
## Logs and exponents

- Definition: $\log_2 x = y$ means $x = 2^y$
  - › the log of x, base 2, is the value y that gives $x = 2^y$
  - › $8 = 2^3$, so $\log_2 8 = 3$
  - › $65536 = 2^{16}$, so $\log_2 65536 = 16$
- Notice that $\log_2 x$ tells you how many bits are needed to hold x values
  - › 8 bits holds 256 numbers: 0 to $2^8-1$ = 0 to 255
  - › $\log_2 256 = 8$




```
x = 0:.1:4
y = 2.^x
plot(x,y,'r')
hold on
plot(y,x,'g')
plot(y,y,'b')
```

$2^x$ and $\log_2 x$




```
x = 0:10
y = 2.^x
plot(x,y,'r')
hold on
plot(y,x,'g')
plot(y,y,'b')
```

$2^x$ and $\log_2 x$

## Floor and Ceiling

$\lfloor X \rfloor$ Floor function: the largest integer $\leq$ X

$\lfloor 2.7 \rfloor = 2$ $\quad \lfloor -2.7 \rfloor = -3$ $\quad \lfloor 2 \rfloor = 2$

$\lceil X \rceil$ Ceiling function: the smallest integer $\geq$ X

$\lceil 2.3 \rceil = 3$ $\quad \lceil -2.3 \rceil = -2$ $\quad \lceil 2 \rceil = 2$



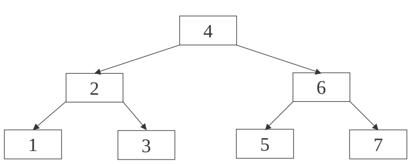
## Facts about Floor and Ceiling

1. $X - 1 < \lfloor X \rfloor \leq X$
2. $X \leq \lceil X \rceil < X + 1$
3. $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$ if n is an integer



## Example: $\log_2 x$ and tree depth

- 7 items in a binary tree, $3 = \lfloor \log_2 7 \rfloor + 1$ levels






## Properties of logs (of the mathematical kind)

- We will assume logs to base 2 unless specified otherwise
- log AB = log A + log B
- Proof:
  - $A = 2^{\log_2 A}$ and $B = 2^{\log_2 B}$
  - $AB = 2^{\log_2 A} \bullet 2^{\log_2 B} = 2^{\log_2 A + \log_2 B}$
  - so $\log_2 AB = \log_2 A + \log_2 B$
  - note: $\log AB \neq \log A \bullet \log B$

## Other log properties

- log A/B = log A – log B
- log ($A^B$) = B log A
- log log X < log X < X for all X > 0
  - › log log X = Y means $2^{2^Y} = X$
  - › log X grows slower than X
    - called a "sub-linear" function



## A log is a log is a log

- Any base x log is equivalent to base 2 log within a constant factor

$$B = 2^{\log_2 B} \qquad x = 2^{\log_2 x}$$

$$\log_x B = \log_x B$$
$$x^{\log_x B} = B$$
$$(2^{\log_2 x})^{\log_x B} = 2^{\log_2 B}$$
$$2^{\log_2 x \log_x B} = 2^{\log_2 B}$$
$$\log_2 x \log_x B = \log_2 B$$
$$\log_x B = \frac{\log_2 B}{\log_2 x}$$



## Arithmetic Series

- $S(N) = 1 + 2 + \ldots + N = \sum_{i=1}^{N} i$
- The sum is
  - › S(1) = 1
  - › S(2) = 1+2 = 3
  - › S(3) = 1+2+3 = 6
- $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$ Why is this formula useful?



## Algorithm Analysis

- Consider the following program segment:

```
x:= 0;
for i = 1 to N do
  for j = 1 to i do
    x := x + 1;
```

- What is the value of x at the end?



## Analyzing the Loop

- Total number of times x is incremented is executed =

$$1 + 2 + 3 + \ldots = \sum_{i=1}^{N} i = \frac{N(N+1)}{2}$$

- Congratulations - You've just analyzed your first program!
  - › Running time of the program is proportional to N(N+1)/2 for all N
  - › O($N^2$)



## Other Important Series

- Sum of squares: $\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$ for large N
- Sum of exponents: $\sum_{i=1}^{N} i^k \approx \frac{N^{k+1}}{|k+1|}$ for large N and k ≠ -1
- Geometric series: $\sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1}$

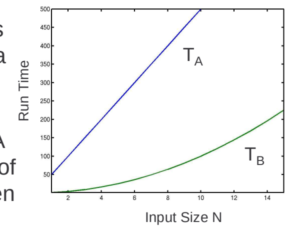
## Mathematical Background

- Today, we will review:
  - › Logs and exponents and series
  - › Asymptotics and order of magnitude notation
  - › Solving recursive equations



## Motivation for Algorithm Analysis

- Suppose you are given two algorithms A and B for solving a problem
- The running times $T_A(N)$ and $T_B(N)$ of A and B as a function of input size N are given

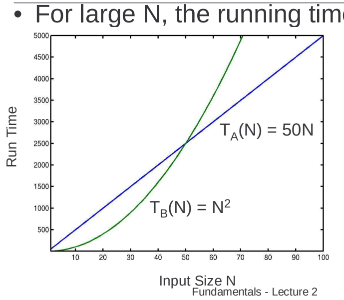



Which is better?



## More Motivation

- For large N, the running time of A and B




Now which algorithm would you choose?



## Asymptotic Behavior

- *Asymptotic* behavior refers to what happens as as $N \to \infty$, regardless of what happens for small N
- Performance for small input sizes may matter in practice, if you are <u>sure</u> that <u>small</u> N will be common <u>forever</u>
- We will compare algorithms based on how they scale for large values of N



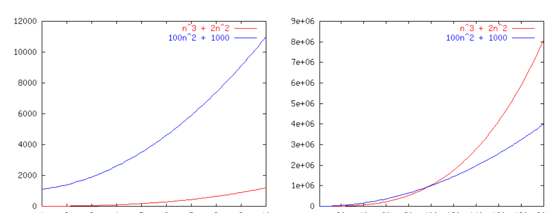
## Which Function Grows Faster?

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



## Which Function Grows Faster?

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



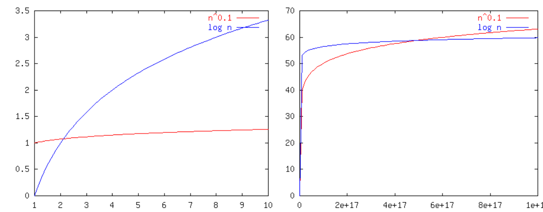
## Which Function Grows Faster?

$n^{0.1}$ vs. $\log n$



## Which Function Grows Faster?

$n^{0.1}$ vs. $\log n$





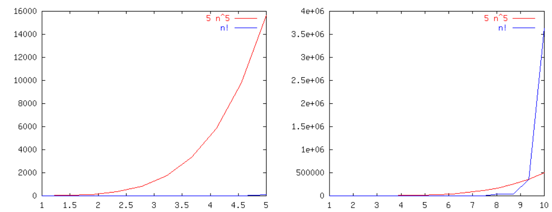
## Which Function Grows Faster?

$5n^5$ vs. $n!$



## Which Function Grows Faster?

$5n^5$ vs. $n!$





## Order Notation

- Mainly used to express upper bounds on time of algorithms. "n" is the size of the input.
- Examples
  - › $3n^3+57n^2+34=O(n^3)$
  - › $10000n + 10\ n \log_2 n = O(n \log n)$
  - › $.00001\ n^2 \neq O(n \log n)$
- Order notation ignores constant factors and low order terms.



## Big-O

- **<u>Def</u>**: *f(n) = O(g(n))* if there exists positive constants c and $n_0$ such that for all $N > n_0$, $f(N) \leq cg(N)$.
- In other words, for large enough n, g is always larger than f.
- So g is an upper bound. (f could be much smaller than g.)

## $16n^3 \log_8(10n^2) + 100n^2 = O(n^3 \log(n))$

- Eliminate low order terms
- Eliminate constant coefficients

$$16n^3 \log_8(10n^2) + 100n^2$$
$$\Rightarrow 16n^3 \log_8(10n^2)$$
$$\Rightarrow n^3 \log_8(10n^2)$$
$$\Rightarrow n^3 \left[\log_8(10) + \log_8(n^2)\right]$$
$$\Rightarrow n^3 \log_8(10) + n^3 \log_8(n^2)$$
$$\Rightarrow n^3 \log_8(n^2)$$
$$\Rightarrow n^3 2 \log_8(n)$$
$$\Rightarrow n^3 \log_8(n)$$
$$\Rightarrow n^3 \log_8(2) \log(n)$$
$$\Rightarrow n^3 \log(n)$$



## Some Basic Time Bounds

- Constant time is O(1)
- Logarithmic time is O(log n)
- Linear time is O(n)
- Quadratic time is $0(n^2)$
- Cubic time is $O(n^3)$
- Polynomial time is $O(n^k)$ for some k.
- Exponential time is $O(c^n)$ for some c > 1.



## Other asymptotics

- Big-Omega: f(n) = Ω(g(n))
  - f(n) ≥ c g(n) for some c > 0 & large enough n.
- Big-Theta: f(n) = Θ(g(n))
  - f(n) = O(g(n)) and f(n) = Ω(g(n))
- Little-O: f(n) = o(g(n))
  - For all c > 0 there is $n_c$ such that for all n > $n_c$, f(n) ≤ c g(n)
  - Limit formulation: $\lim_{n\to\infty} f(n)/g(n) = 0$



## Conventions of Order Notation

Order notation is not symmetric: write $2n^2 + n = O(n^2)$

but never $O(n^2) = 2n^2 + n$

The expression $O(f(n)) = O(g(n))$ is equivalent to

$f(n) = O(g(n))$

The right-hand side is a "cruder" version of the left:

$18n^2 = O(n^2) = O(n^3) = O(2^n)$

$18n^2 = \Omega(n^2) = \Omega(n \log n) = \Omega(n)$

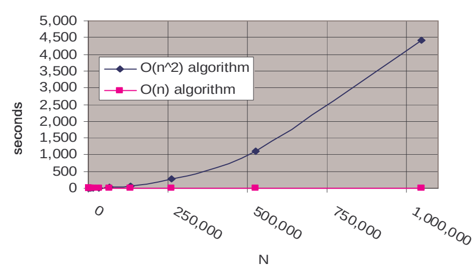


## Kinds of Analysis

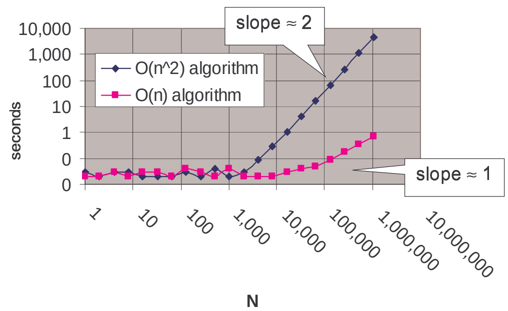
- Asymptotic – uses order notation, ignores constant factors and low order terms.
- Upper bound vs. lower bound
- Worst case – time bound valid for all inputs of length n.
- Average case – time bound valid on average – requires a distribution of inputs.
- Amortized – worst case time averaged over a sequence of operations.
- Others – best case, common case, cache miss



## Estimating Order by Plotting



## Log-Log Plot




## Property of Log/Log Plots

- On a linear plot, a *linear* function is a straight line
- On a log/log plot, *any* polynomial function is a straight line!
  - › The *slope* Δy/Δ x is the same as the exponent

Proof: Suppose $y = cx^k$

Then $\log y = \log(cx^k)$

$\log y = \log c + \log x^k$

$\log y = \log c + k \log x$

vertical axis — y intercept — slope — horizontal axis



## Mathematical Background

- Today, we will review:
  - › Logs and exponents and series
  - › Asymptotics and order of magnitude notation
  - › Solving recursive equations



## Analyzing Recursive Programs

1. Express the running time *T(n)* as a recursive equation
2. Solve the recursive equation
   - For an upper-bound analysis, you can optionally simplify the equation to something larger
   - For a lower-bound analysis, you can optionally simplify the equation to something smaller



## Binary Search

```
function bfind(x:integer, a[]:integer array, i,j:integer)
{   if (j-i < 0) return -1;
     m := (i+j)/ 2;
    if (x = a[m]) return m;
    if (x < a[m]) then
        return bfind(x, a, i, m-1);
    else
        return bfind(x, a, m+1, j); }
Call bfind(x,a,0,n-1) to get the result of binary search
```

What is the worst-case upper bound?

Okay, let's *prove* it is θ(log *n*)…



## Binary Search

```
function bfind(x:integer, a[]:integer array, i,j:integer)
{   if (j-i < 0) return -1;
    m := (i+j)/ 2;
    if (x = a[m]) return m;
    if (x < a[m]) then
        return bfind(x, a, i, m-1);
    else
        return bfind(x, a, m+1, j); }
```

Introduce some constants…

b = time needed for base case

c = time needed to get ready to do a recursive call

n = j-i+1 is the size of the subproblem

Running time T(n) satisfies: $T(1) \le b$

$T(n) \le T(n/2) + c$

## Solving Recursive Equation
### (by Repeated Substitution)

| | |
|---|---|
| $T(n) \le T(n/2) + c$ | Recurrence |
| $\le T(n/4) + c + c$ | $T(n/2) \le T(n/4) + c$ |
| $\le T(n/8) + c + c + c$ | $T(n/4) \le T(n/8) + c$ |
| $T(n) \le T(n/2^k) + kc$ | General form |
| $T(n) \le T(n/2^{\log_2 n}) + c\log_2 n$ | Let $k = \log_2 n$ |
| $= T(n/n) + c\log_2 n$ | |
| $= T(1) + c\log_2 n = b + c\log_2 n = O(\log n)$ | |



## Solving Recursive Equations by Induction

- Repeated substitution and telescoping construct the solution
- If you know the closed form solution, you can validate it by ordinary induction
- For the induction, may want to increase n by a multiple (2n) rather than by n+1



## Inductive Proof

Base case

$T(1) \le b = b + c\log_2 1$

Inductive assumption

$T(n) \le b + c\log_2 n$

Inductive step

$$\begin{aligned} T(2n) &\le T(n) + c \\ &\le b + c\log_2 n + c \\ &\le b + c\log_2 n + c\log_2 2 \\ &\le b + c(\log_2 n + \log_2 2) \\ &\le b + c\log_2 2n \end{aligned}$$