

Algorithmic Complexity



CMSC 132

**Department of Computer Science
University of Maryland, College Park**

Algorithm Efficiency

■ Efficiency

- Amount of resources used by algorithm
 - Time, space

■ Measuring efficiency

- Benchmarking
- Asymptotic analysis

Benchmarking

■ Approach

- Pick some desired inputs
- Actually run implementation of algorithm
- Measure time & space needed

■ Industry benchmarks

- SPEC – CPU performance
- MySQL – Database applications
- WinStone – Windows PC applications
- MediaBench – Multimedia applications
- Linpack – Numerical scientific applications

Benchmarking

■ Advantages

- Precise information for given configuration
 - Implementation, hardware, inputs

■ Disadvantages

- Affected by configuration
 - Data sets (usually too small)
 - Hardware
 - Software
- Affected by special cases (biased inputs)
- Does not measure **intrinsic** efficiency

Asymptotic Analysis

■ Approach

- Mathematically analyze efficiency
- Calculate time as function of input size n
 - $T \approx O[f(n)]$
 - T is on the order of $f(n)$
 - “Big O” notation

■ Advantages

- Measures intrinsic efficiency
- Dominates efficiency for large input sizes

Search Example

- **Number guessing game**
 - Pick a number between $1 \dots n$
 - Guess a number
 - Answer “correct”, “too high”, “too low”
 - Repeat guesses until correct number guessed

Linear Search Algorithm

■ Algorithm

1. Guess number = 1
2. If incorrect, increment guess by 1
3. Repeat until correct

■ Example

- Given number between 1...100
- Pick 20
- Guess sequence = 1, 2, 3, 4 ... 20
- Required 20 guesses

Linear Search Algorithm

■ Analysis of # of guesses needed for 1...n

- If number = 1, requires 1 guess
- If number = n, requires n guesses
- On average, needs $n/2$ guesses
- Time = $O(n)$ = Linear time

Binary Search Algorithm

■ Algorithm

- Set Δ to $n/4$
- Guess number = $n/2$
- If too large, guess number $- \Delta$
- If too small, guess number $+ \Delta$
- Reduce Δ by $1/2$
- Repeat until correct

Binary Search Algorithm

■ Example

- Given number between 1...100

- Pick 20

- Guesses =

 - 50, $\Delta = 25$, Answer = too large, subtract Δ

 - 25, $\Delta = 12$, Answer = too large, subtract Δ

 - 13, $\Delta = 6$, Answer = too small, add Δ

 - 19, $\Delta = 3$, Answer = too small, add Δ

 - 22, $\Delta = 1$, Answer = too large, subtract Δ

 - 21, $\Delta = 1$, Answer = too large, subtract Δ

 - 20

- Required 7 guesses

Binary Search Algorithm

- **Analysis of # of guesses needed for 1...n**
 - If number = $n/2$, requires 1 guess
 - If number = 1, requires $\log_2(n)$ guesses
 - If number = n , requires $\log_2(n)$ guesses
 - On average, needs $\log_2(n)$ guesses
 - Time = $O(\log_2(n))$ = **Log** time

Search Comparison

- **For number between 1...100**
 - Simple algorithm = 50 steps
 - Binary search algorithm = $\log_2(n) = 7$ steps
- **For number between 1...100,000**
 - Simple algorithm = 50,000 steps
 - Binary search algorithm = $\log_2(n)$ (about 17 step
- **Binary search is much more efficient!**

Asymptotic Complexity

■ Comparing two linear functions

Size	Running Time	
	$n/2$	$4n+3$
64	32	259
128	64	515
256	128	1027
512	256	2051

Asymptotic Complexity

- **Comparing two functions**
 - $n/2$ and $4n+3$ behave similarly
 - Run time roughly doubles as input size doubles
 - Run time increases **linearly** with input size
- **For large values of n**
 - $\text{Time}(2n) / \text{Time}(n)$ approaches exactly 2
- **Both are $O(n)$ programs**

Asymptotic Complexity

■ Comparing two log functions

Size	Running Time	
	$\log_2(n)$	$5 * \log_2(n) + 3$
64	6	33
128	7	38
256	8	43
512	9	48

Asymptotic Complexity

- Comparing two functions
 - $\log_2(n)$ and $5 * \log_2(n) + 3$ behave similarly
 - Run time roughly increases by constant as input size doubles
 - Run time increases **logarithmically** with input size
- For large values of n
 - $\text{Time}(2n) - \text{Time}(n)$ approaches constant
 - Base of logarithm does not matter
 - Simply a multiplicative factor
$$\log_a N = (\log_b N) / (\log_b a)$$
- Both are $O(\log(n))$ programs

Asymptotic Complexity

■ Comparing two quadratic functions

Size	Running Time	
	n^2	$2n^2 + 8$
2	4	16
4	16	40
8	64	132
16	256	520

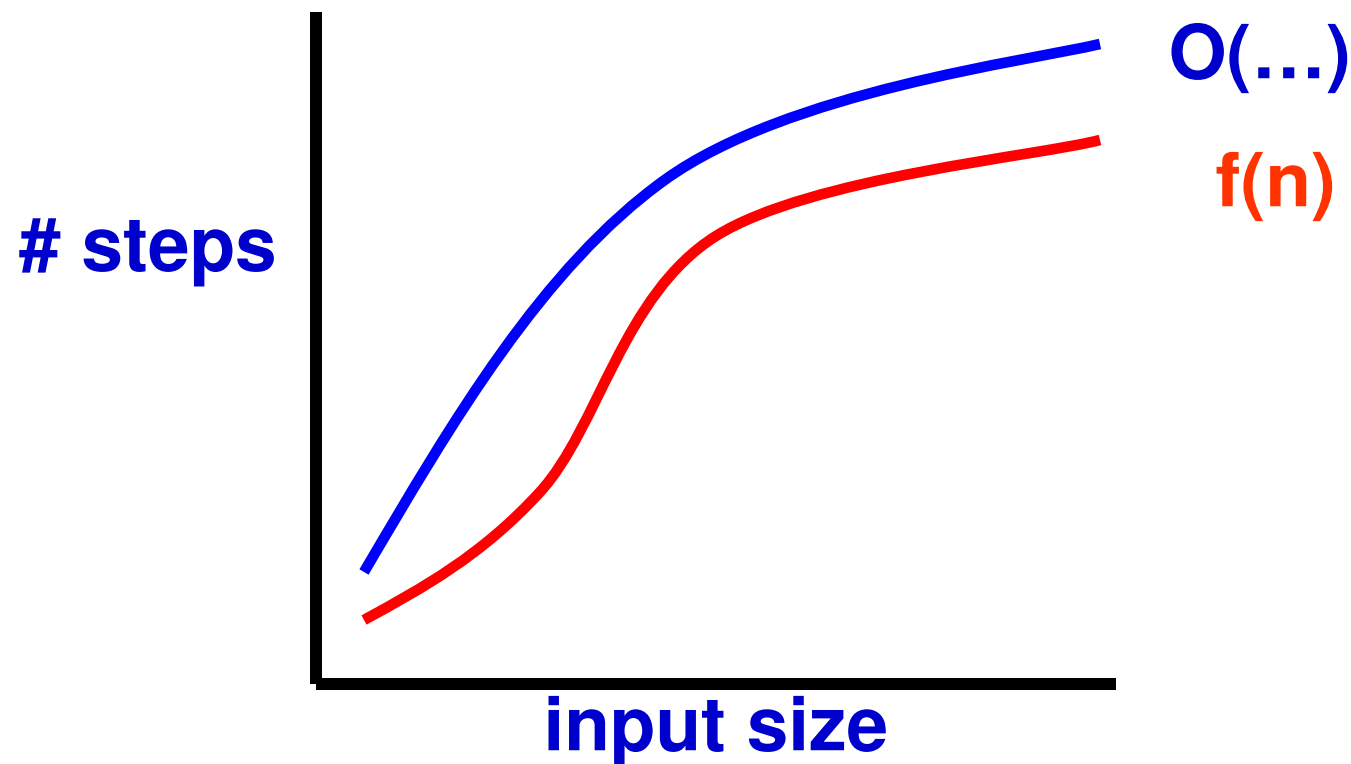
Asymptotic Complexity

- Comparing two functions
 - n^2 and $2n^2 + 8$ behave similarly
 - Run time roughly increases by 4 as input size doubles
 - Run time increases quadratically with input size
- For large values of n
 - $\text{Time}(2n) / \text{Time}(n)$ approaches 4
- Both are $O(n^2)$ programs

Big-O Notation

■ Represents

- Upper bound on number of steps in algorithm
- Intrinsic efficiency of algorithm for large inputs



Formal Definition of Big-O

■ **Function $f(n)$ is $O(g(n))$ if**

■ **For some positive constants M, N_0**

■ **$M \times g(n) \geq f(n)$, for all $n \geq N_0$**

■ **Intuitively**

■ **For some coefficient M & all data sizes $\geq N_0$**

■ **$M \times g(n)$ is always greater than $f(n)$**

Big-O Examples

■ $5n + 1000 \Rightarrow O(n)$

■ Select $M = 6$, $N_0 = 1000$

■ For $n \geq 1000$

■ $6n \geq 5n+1000$ is always true

■ Example $\Rightarrow$ for $n = 1000$

■ $6000 \geq 5000 + 1000$

Big-O Examples

■ $2n^2 + 10n + 1000 \Rightarrow O(n^2)$

■ Select $M = 4$, $N_0 = 100$

■ For $n \geq 100$

■ $4n^2 \geq 2n^2 + 10n + 1000$ is always true

■ Example $\Rightarrow$ for $n = 100$

■ $40000 \geq 20000 + 1000 + 1000$

Observations

- **Big O categories**

- $O(\log(n))$

- $O(n)$

- $O(n^2)$

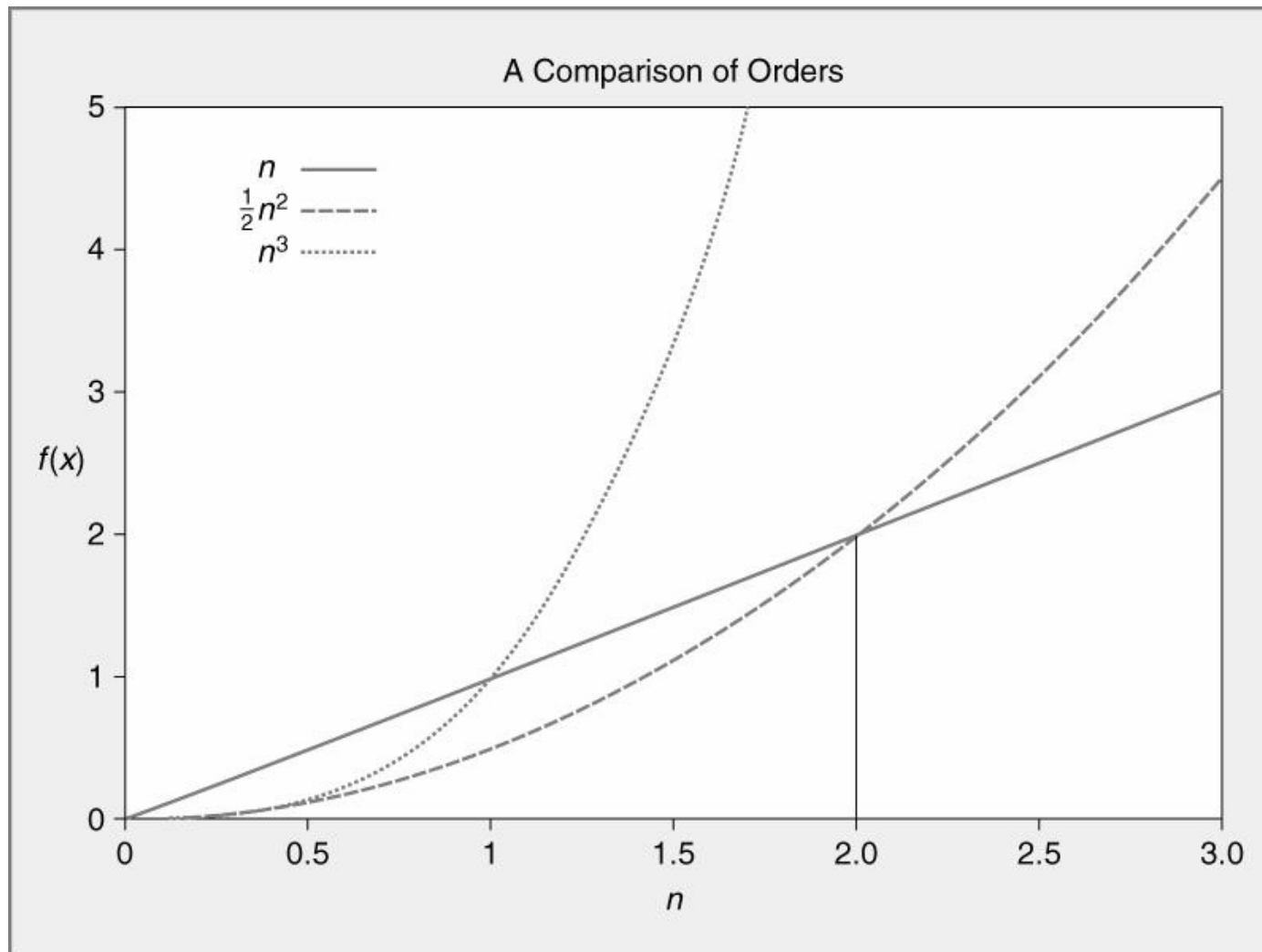
- **For large values of n**

- Any $O(\log(n))$ algorithm is faster than $O(n)$

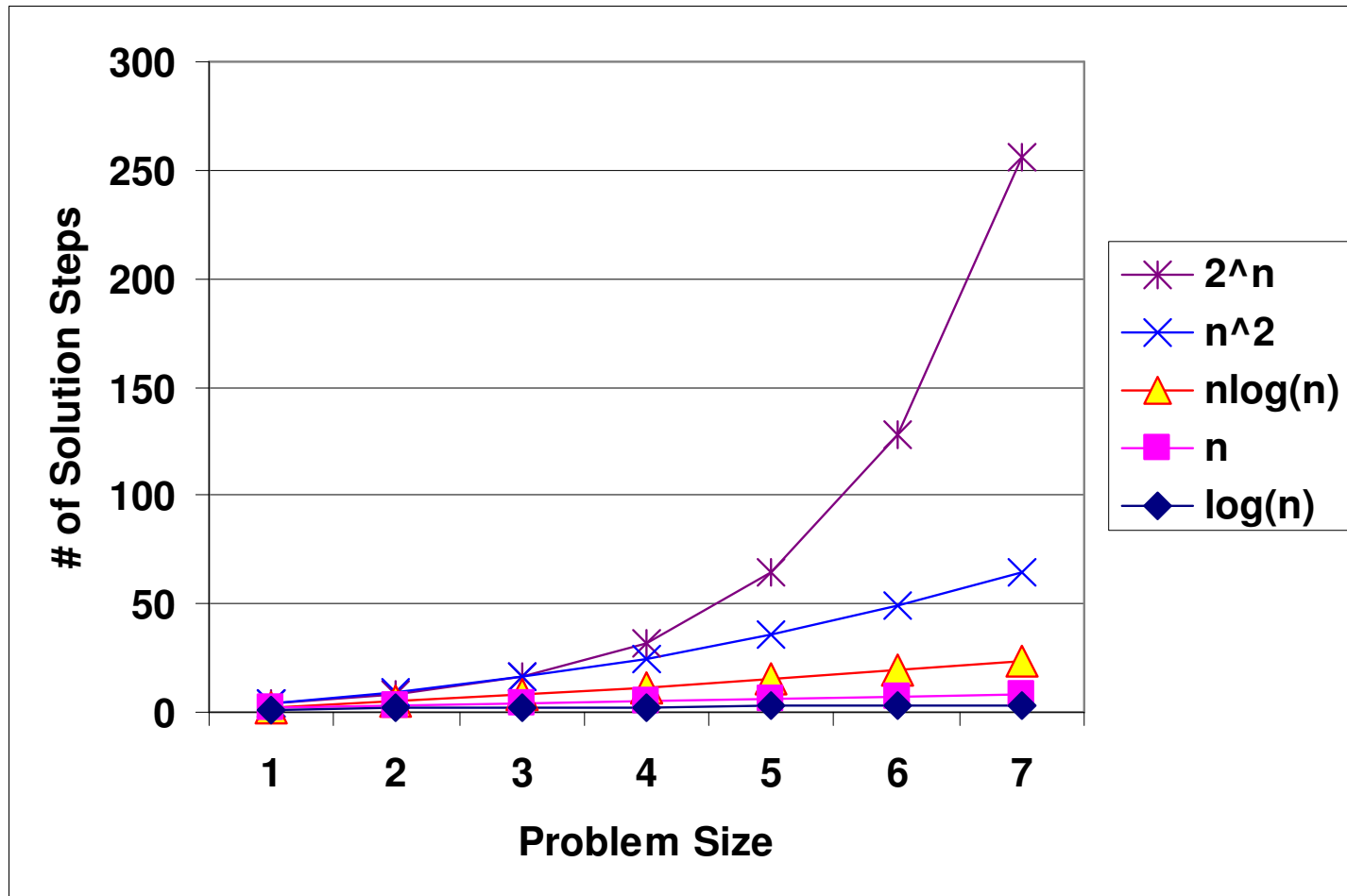
- Any $O(n)$ algorithm is faster than $O(n^2)$

- **Asymptotic complexity is fundamental measure of efficiency**

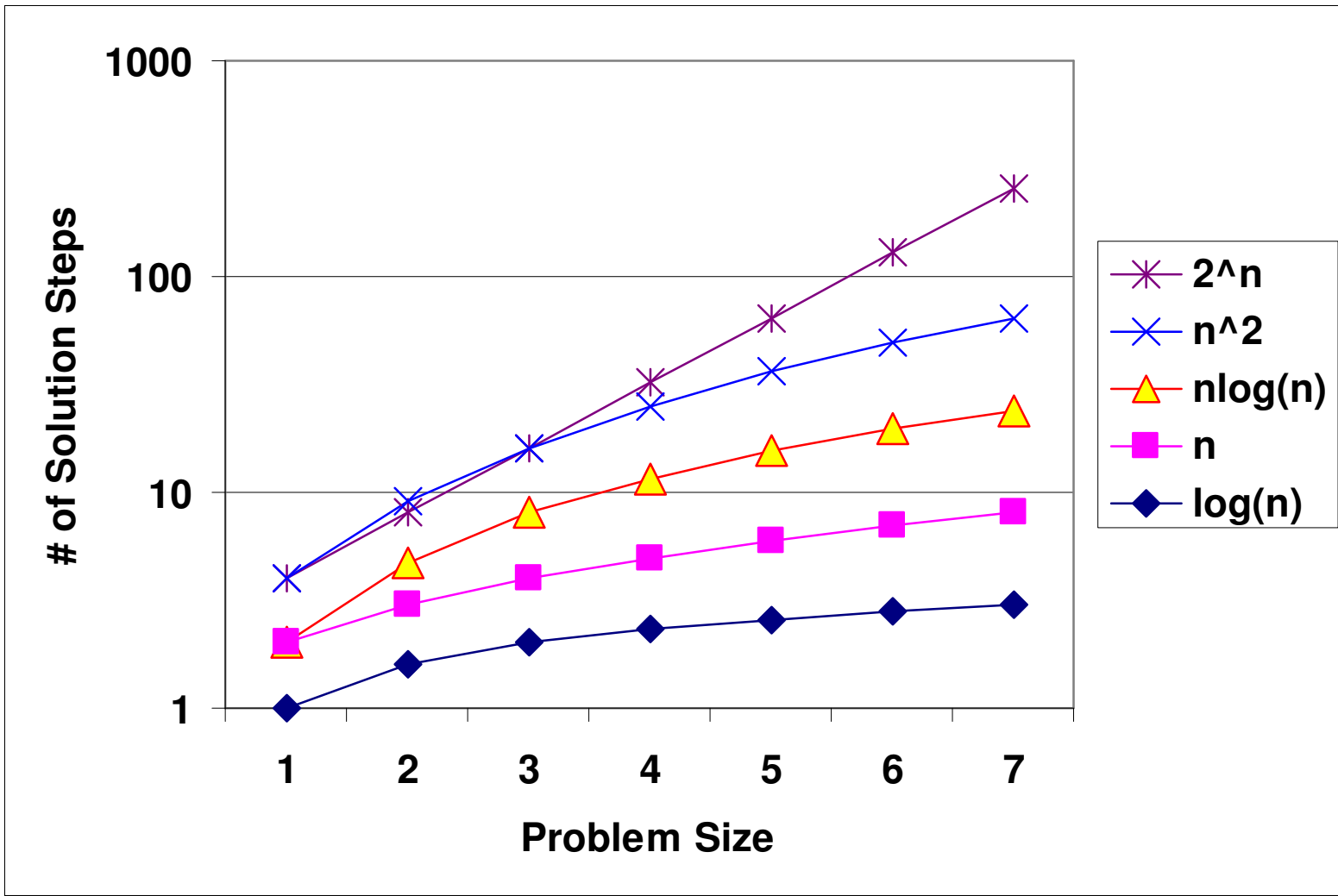
Comparison of Complexity



Complexity Category Example



Complexity Category Example



Calculating Asymptotic Complexity

■ As **n** increases

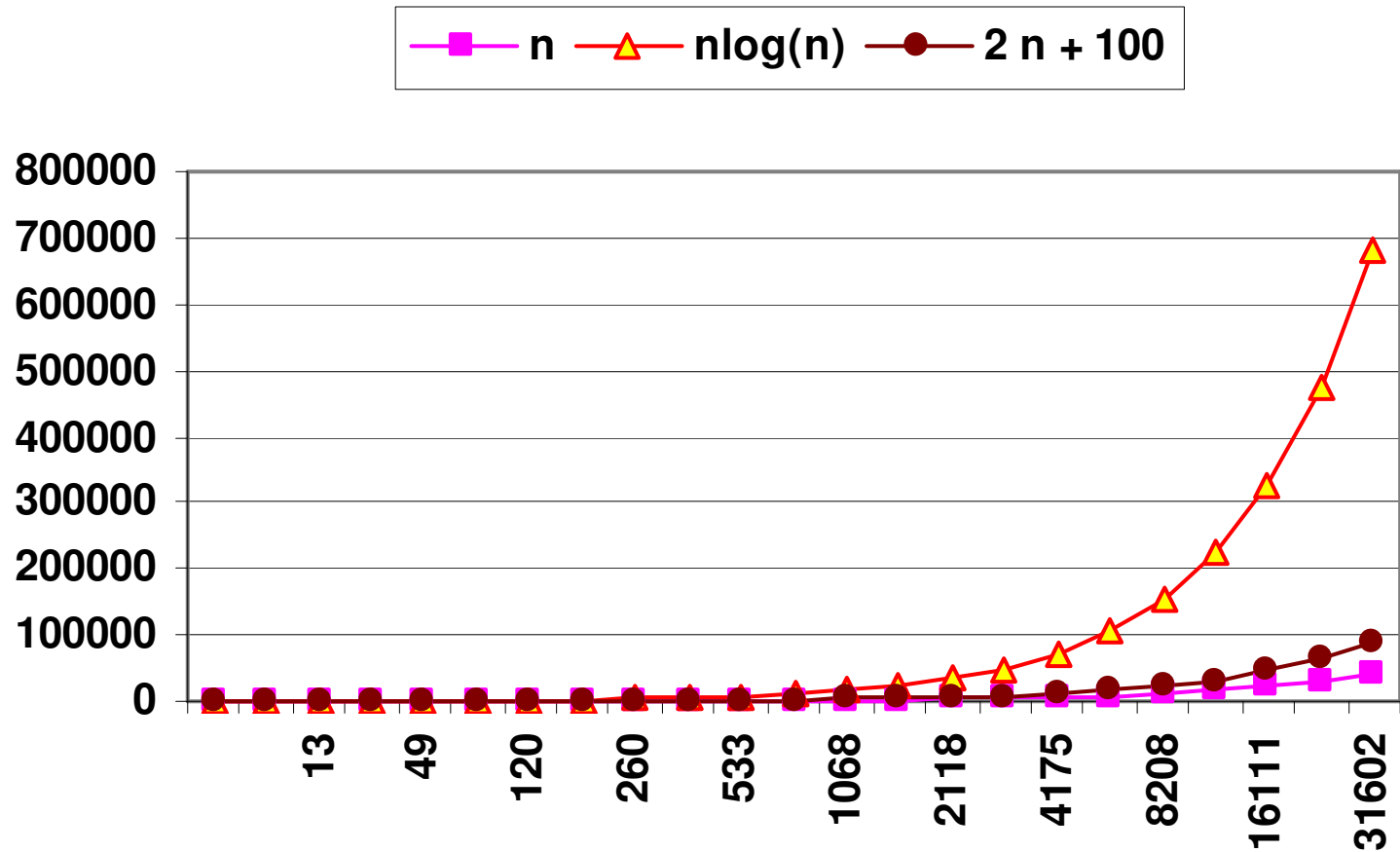
- Highest complexity term dominates
- Can ignore lower complexity terms

■ Examples

- $2n + 100 \Rightarrow O(n)$
- $n \log(n) + 10n \Rightarrow O(n \log(n))$
- $\frac{1}{2}n^2 + 100n \Rightarrow O(n^2)$
- $n^3 + 100n^2 \Rightarrow O(n^3)$
- $\frac{1}{100}2^n + 100n^4 \Rightarrow O(2^n)$

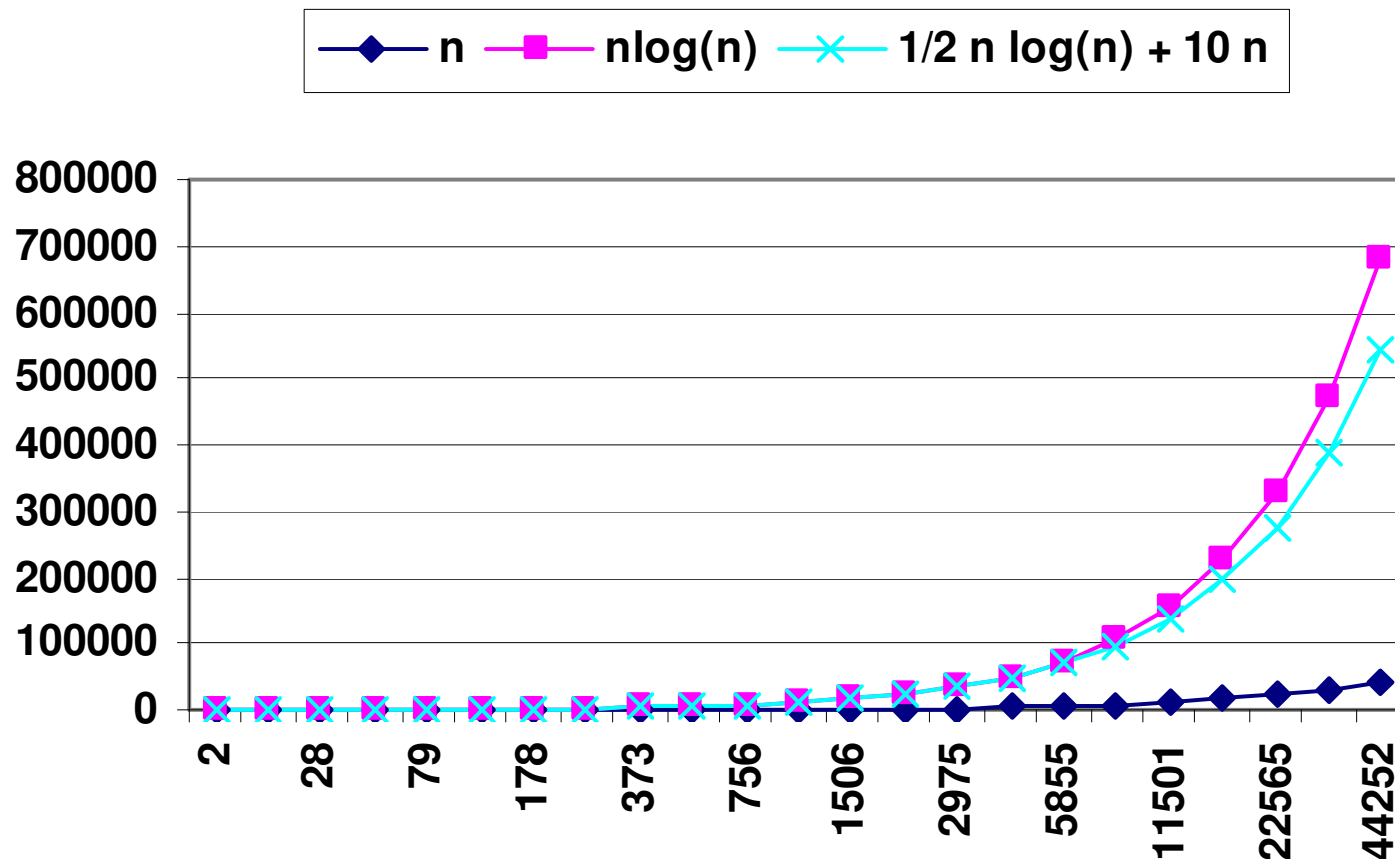
Complexity Examples

■ $2n + 100 \Rightarrow O(n)$



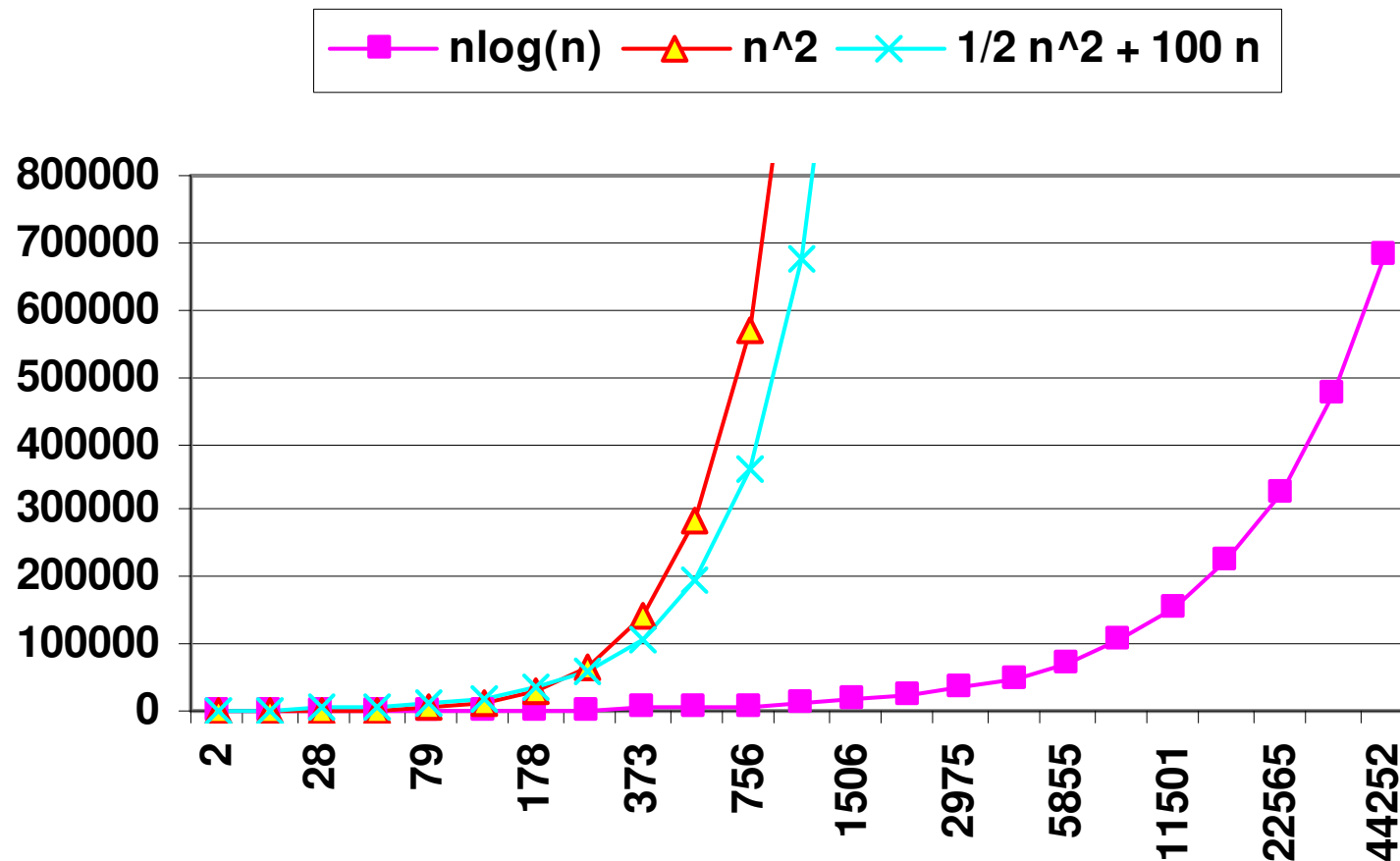
Complexity Examples

■ $\frac{1}{2} n \log(n) + 10 n \Rightarrow O(n \log(n))$



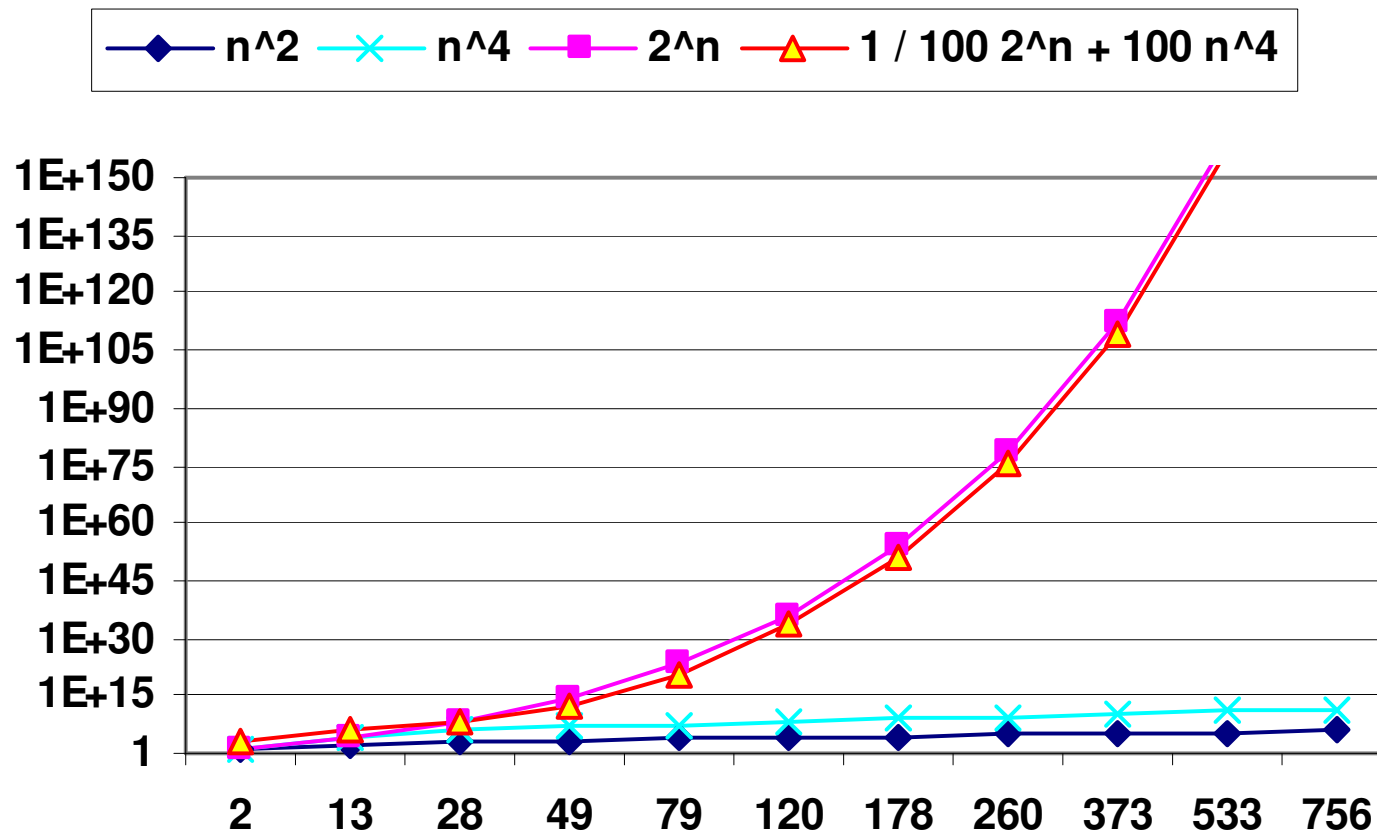
Complexity Examples

■ $\frac{1}{2} n^2 + 100 n \Rightarrow O(n^2)$



Complexity Examples

■ $\frac{1}{100} 2^n + 100 n^4 \Rightarrow O(2^n)$



Types of Case Analysis

- Can analyze different types (cases) of algorithm behavior
- Types of analysis
 - Best case
 - Worst case
 - Average case

Types of Case Analysis

■ Best case

- Smallest number of steps required
- Not very useful
- Example $\Rightarrow$ Find item in first place checked

Types of Case Analysis

■ Worst case

- Largest number of steps required
- Useful for upper bound on worst performance
 - Real-time applications (e.g., multimedia)
 - Quality of service guarantee
- Example $\Rightarrow$ Find item in last place checked

Quicksort Example

■ Quicksort

- One of the fastest comparison sorts
- Frequently used in practice

■ Quicksort algorithm

- Pick **pivot** value from list
- Partition list into values smaller & bigger than pivot
- Recursively sort both lists

Quicksort Example

■ Quicksort properties

- Average case = $O(n \log(n))$

- Worst case = $O(n^2)$

 - Pivot $\approx$ smallest / largest value in list

 - Picking from front of nearly sorted list

■ Can avoid worst-case behavior

- Select random pivot value

Types of Case Analysis

■ Average case

- Number of steps required for “typical” case
- Most useful metric in practice
- Different approaches
 - Average case
 - Expected case
 - Amortized

Approaches to Average Case

■ Average case

- Average over all possible inputs
- Assumes some probability distribution, usually uniform

■ Expected case

- Algorithm uses randomness
- Worst case over all possible input
- Average over all possible random values

■ Amortized

- For all long sequences of operations
- Worst case total time divided by # of operations

Amortization Example

- Adding numbers to end of array of size k
 - If array is full, allocate new array
 - Allocation cost is $O(\text{size of new array})$
 - Copy over contents of existing array
- Two approaches
 - Non-amortized
 - If array is full, allocate new array of size $k+1$
 - Amortized
 - If array is full, allocate new array of size $2k$
 - Compare their allocation cost

Amortization Example

■ Non-amortized approach

■ Allocation cost as table grows from 1..**n**

Size (k)	1	2	3	4	5	6	7	8
Cost	1	2	3	4	5	6	7	8

■ Total cost $\Rightarrow n(n+1)/2$

■ Case analysis

■ Best case $\Rightarrow$ allocation cost = **k**

■ Worse case $\Rightarrow$ allocation cost = **k**

■ Amortized case $\Rightarrow$ allocation cost = $(n+1)/2$

Amortization Example

■ Amortized approach

- Allocation cost as table grows from 1..**n**

Size (k)	1	2	3	4	5	6	7	8
Cost	2	0	4	0	8	0	0	0

- Total cost $\Rightarrow 2(n - 1)$

■ Case analysis

- Best case $\Rightarrow$ allocation cost = 0
- Worse case $\Rightarrow$ allocation cost = $2(k - 1)$
- Amortized case $\Rightarrow$ allocation cost = 2

- An individual step might take longer, but faster for any sequence of operations