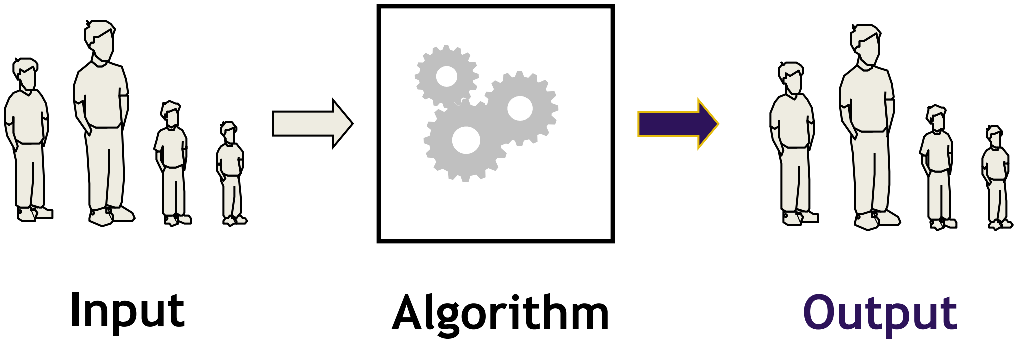
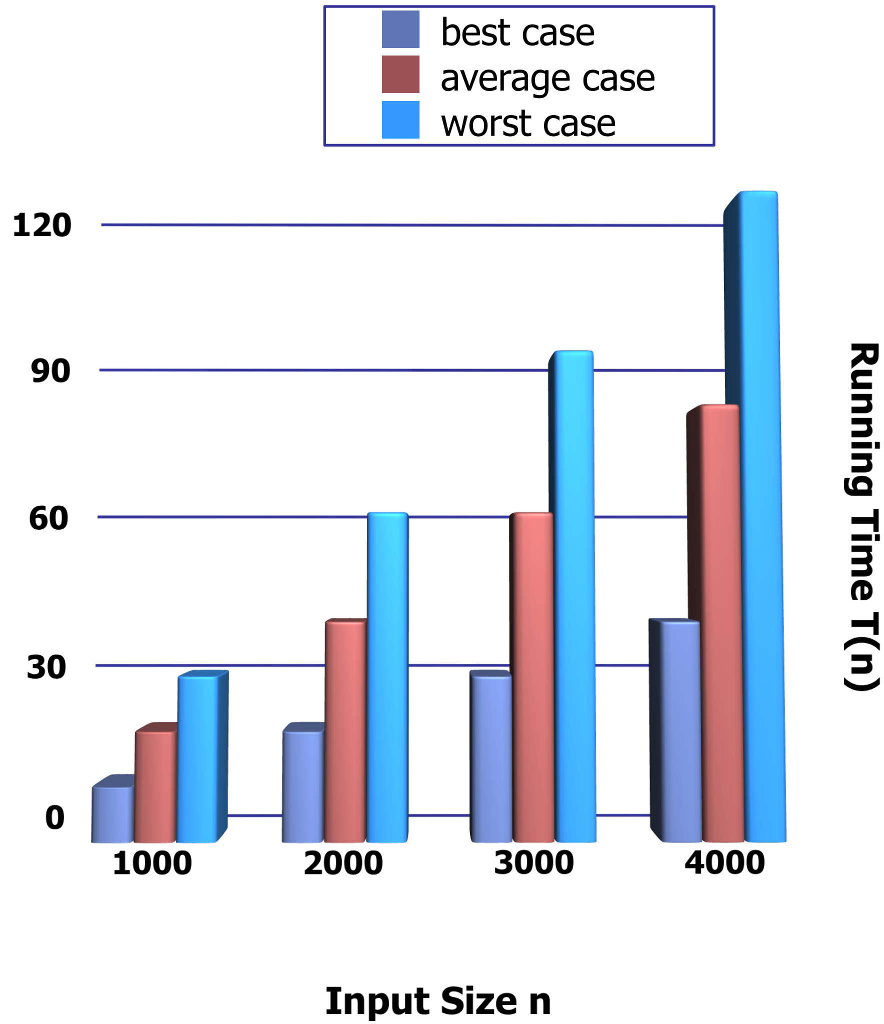
Analysis of Algorithms



An **algorithm** is a step-by-step procedure of unambiguous instructions for solving a problem in a finite amount of time.

Algorithm Running Time

- Algorithms operate on input to produce a result.
- The ***running time*** of an algorithm typically grows with the input size.
- **Average case** performance difficult to measure
- Focus on the **worst case** performance
  - Easier to analyze
  - Pays to be pessimistic
  - Identify bottlenecks/ weakness



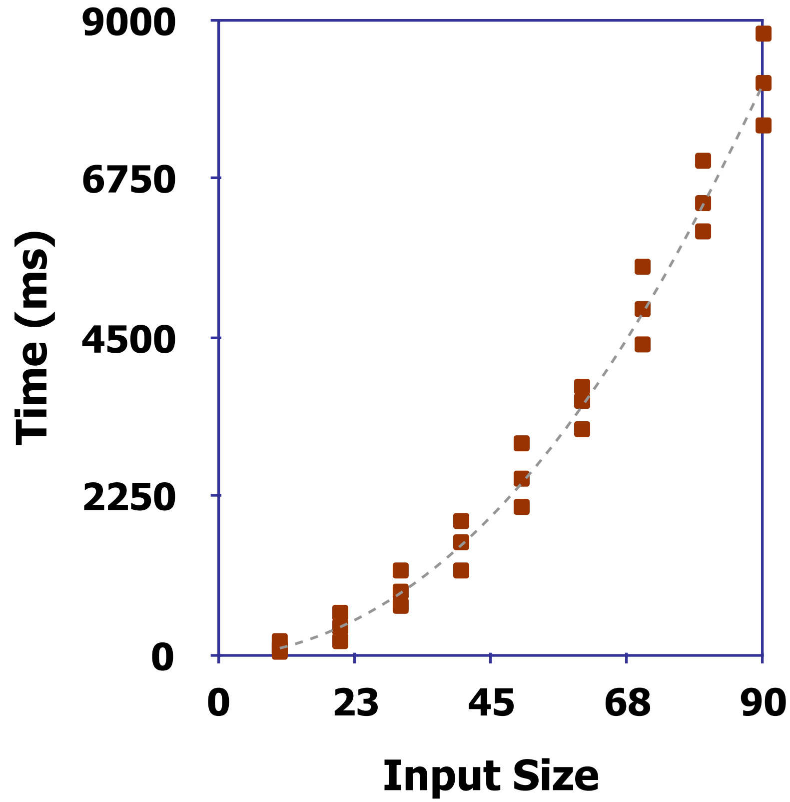
Running Time - Dependencies

- The running time (clock) is dependent on:
  - Hardware, bandwidth
  - Design and implementation
  - Compiler optimization
  - Input

1. *Algorithms, just as humans, have strengths and weaknesses.*
2. *Perform well on a particular input, but equally poor on a different input.*

Measure Performance - Experimental Studies

- Implement **algorithm**
- Run the program on inputs of varying ***size/composition.***
- Measure runtime:
  clock-time, cpu-time
- Plot the results.



Experimental Studies - Limitation

- Must implement the algorithm first
- Difficult to obtain a ***good*** *(large and varied)* range of inputs
  - Real world data – costly to acquire & time consuming
  - Synthetically created data – makes claims suspect
- Need equal hardware and software environments to compare performance
- Difficult to be exhaustive, or use enough sample inputs to be able to make reliable claims about the algorithm.

Theoretical Analysis

- **Pseudo code -** high-level description of algorithm
- **Running time -** function of the input size
  e.g. T(n) or f(n)
- Takes into account all possible inputs
- We care about very large input sizes: as n *approaches* infinity
- **Metric -** count number of primitive operations

Theoretical Analysis - Benefits

Study algorithm performance across all:
- machines
- programming languages

Fairly compare two algorithms
- **Examples:** sort, search

Asymptotic Analysis – Compare running time as a function of the input size in the limit, *i.e.*, as n approaches infinity.

Student Expectations

For algorithms in CS14:

- Discuss an algorithm's behavior and performance
- Specify in pseudo code:
  - Mixture of natural language and programming language constructs
  - Less ambiguous than natural language (+)
  - Closer to implementation language (+)
  - Implementation details omitted. (+)
  - Not actual source code! (-)
- Analyze the running time in O-notation
- Implement in code (C++)

Pseudocode

- high-level description of an algorithm
- more structured and less ambiguous than English
- less detailed than a programming language
- preferred notation for describing algorithms
- hides implementation details

Example: find max element of an array

Algorithm *arrayMax*(A, n)
  Input array A of n integers
  Output maximum element of A

  $currentMax \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if $A[i] > currentMax$ **then**
      $currentMax \leftarrow A[i]$
  return $currentMax$

Pseudocode v. C++ : side-by-side comparison

Algorithm *arrayMax*(A,*n*)
    Input: An array A storing n ≥ 1 integers.
    Output: The maximum element of A.
    currentMax ← *A[0]*
    for i ← n to n – 1 do
        if *currentMax* < *A[i]* then
            currentMax ← *A[i]*
    return *currentMax*

```
int arrayMax(int A[], int n){


int currentMax = A[0];
   for(int i = 1; i < n; i++){
        if(currentMax < A[i])
           currentMax = A[i];
   }
    return currentMax;
}
```

Analysis - Counting Primitive Operations

- Basic computations performed by an algorithm
  - Assigning a value to a variable, x = 5, x = y
  - Function call, max(5,7)
  - Performing an arithmetic operation, e.g., 5+7
  - Comparison, e.g., x < 5
  - Indexing into an array, a[5]
  - Evaluating an expression (4+n)*5
  - Returning from a function, return
- Above are Primitive Operations ("Atomic" in book)
  - a low level instruction whose execution time depends on environment's hardware and software .
  - for analysis purposes, constant time instruction, O(1) – "Big-Oh of 1" or "constant time"

Counting Primitive Operations

- By inspecting the pseudo code, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Algorithm *printArray*(*A*, *n*)

$i \leftarrow 0$	1 assignment
while $i < n$ **do**	n + 1 comparisons
cout << A[i] << endl	n outputs
i ++	n increments

1 + (n+1) + n + n = *3n* + *2* operations
Proportional to n, 3 times n + constant

Counting Primitive Operations

- (Stop here) Quick exercise -

Algorithm *foo*(*n*)

$x \leftarrow 0, y \leftarrow 0$

while $x < \mathrm{n}$ **do**

x ++

while $y < \mathrm{n}$ **do**

y ++

$y \leftarrow 0$

Remember that...

$$printArray = 3n + 2$$

Algorithm *printArray*(*A*, *n*)

$i \leftarrow 0$ — 1 assignment

while $i < n$ **do** — n + 1 comparisons

cout << A[i] << endl — n outputs

i++ — n increments

1 + (n+1) + n + n = $3n$ + 2 operations

Proportional to n, 3 times n + constant

Estimating Running Time

- Algorithm ***printArray*** executes $3\boldsymbol{n} + 2$ primitive operations.
- If we *define*:

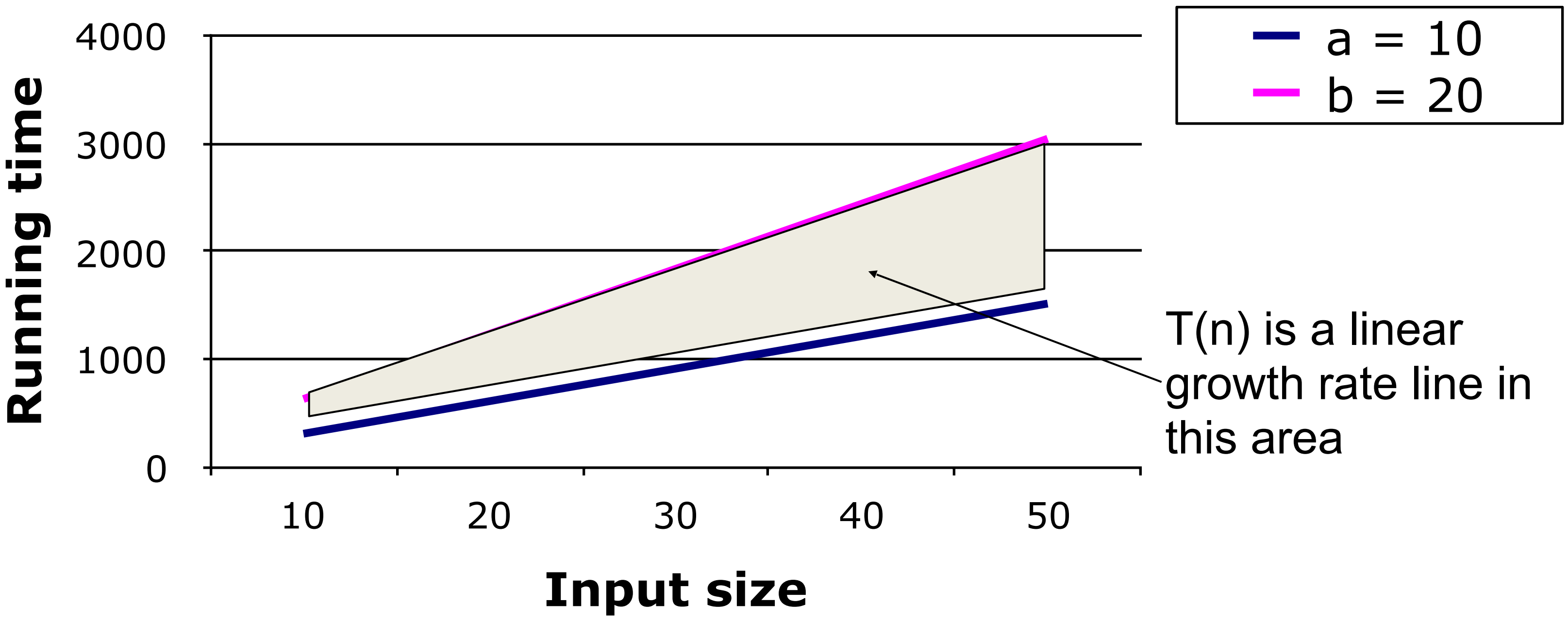
  $\boldsymbol{a}$ = Time taken by the fastest primitive operation

  $\boldsymbol{b}$ = Time taken by the slowest primitive operation
- Let $\boldsymbol{T}(\boldsymbol{n})$ be worst-case time of ***printArray.*** Then

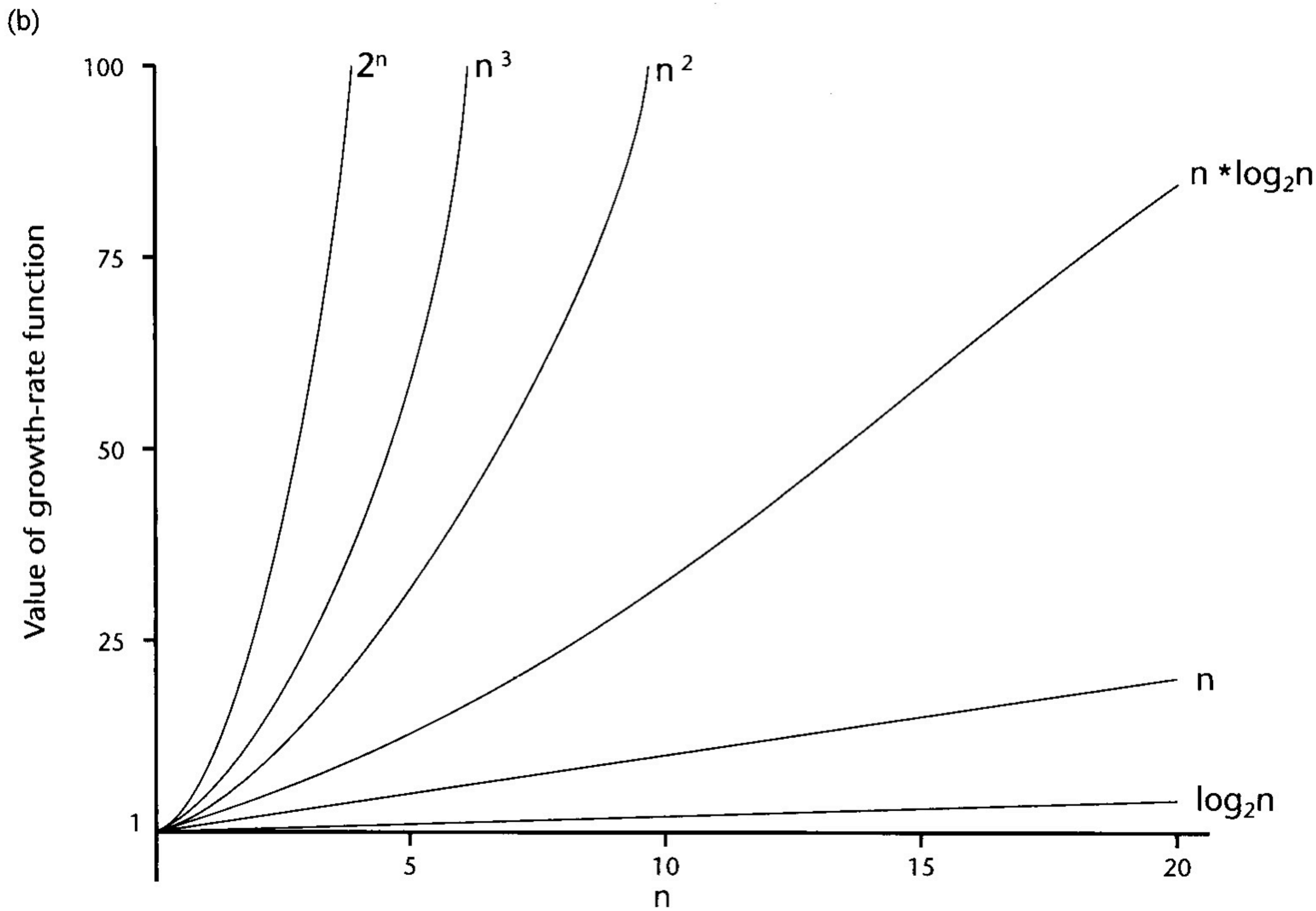
$$\boldsymbol{a}\,(3\boldsymbol{n} + 2) \leq \boldsymbol{T}(\boldsymbol{n}) \leq \boldsymbol{b}(3\boldsymbol{n} + 2)$$

- Hence, the running time $\boldsymbol{T}(\boldsymbol{n})$ is bounded by two linear functions.

Growth Rate of Running Time

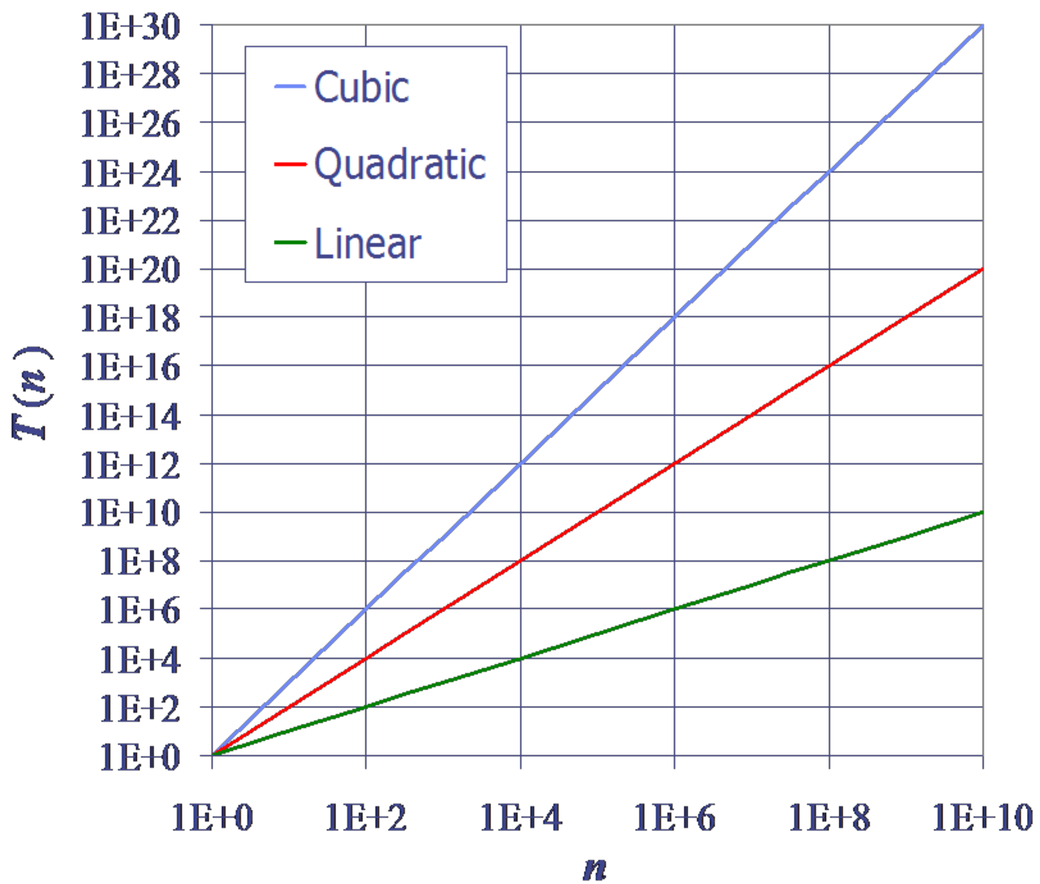


Growth Rates



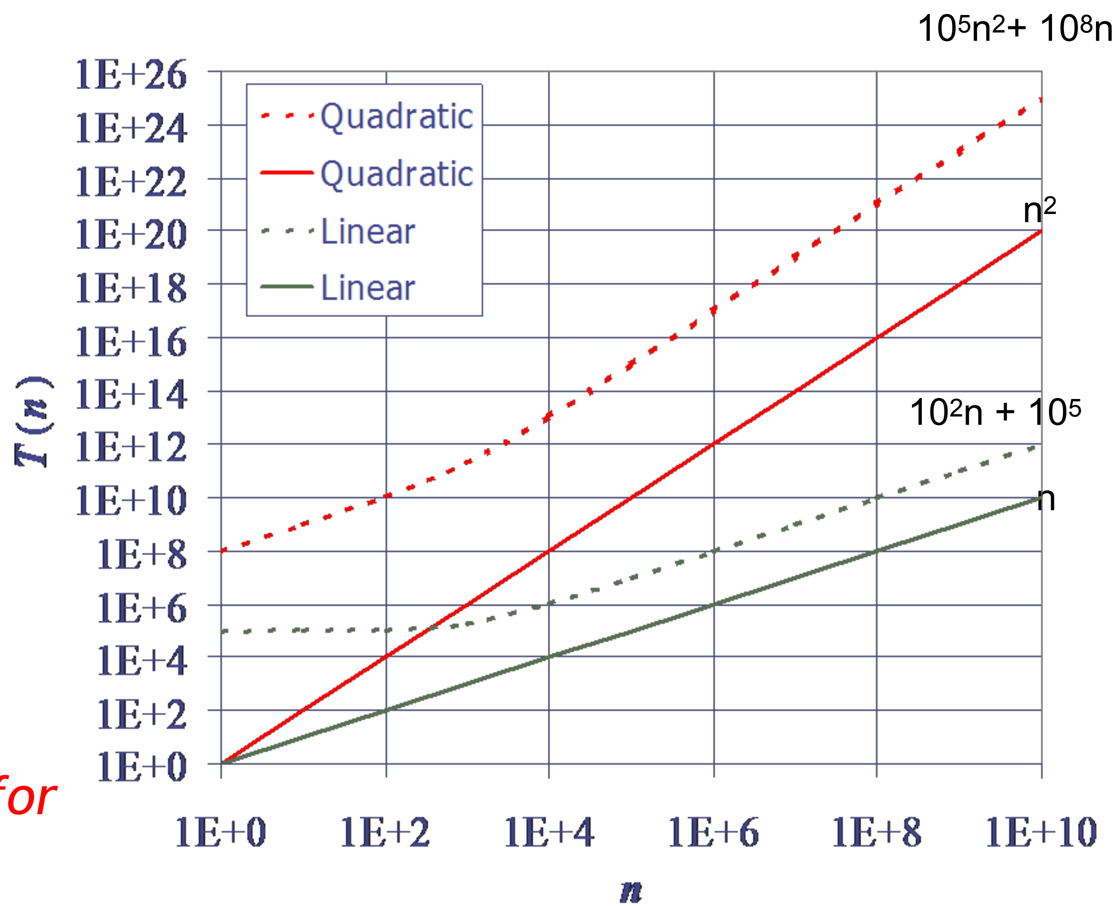
Growth Rates

- Growth rates of functions:
  - Linear ≈ n
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
- In a log-log chart, the slope of the line corresponds to the growth rate of the function



Constant Factors & Lower-Order Terms

- The growth rate is not affected by
  - constant factors
  - lower-order terms
- Examples
  - $10^2n + 10^5$ is a linear function
  - $10^5n^2 + 10^8n$ is a quadratic function
  - **Beware**- *very large constant terms and for small n.*



Constant Factors & Lower-Order Terms

- Examples
  - ***2n*** and ***100n*** have the same relative growth rates
  - ***10n*** and ***10n + 4*** have the same relative growth rates
  - $3n^2 + 10n + 7$ and n^2 have the same relative growth rates
  - ***10000n + 1000*** and ***n*** have the same relative growth rates

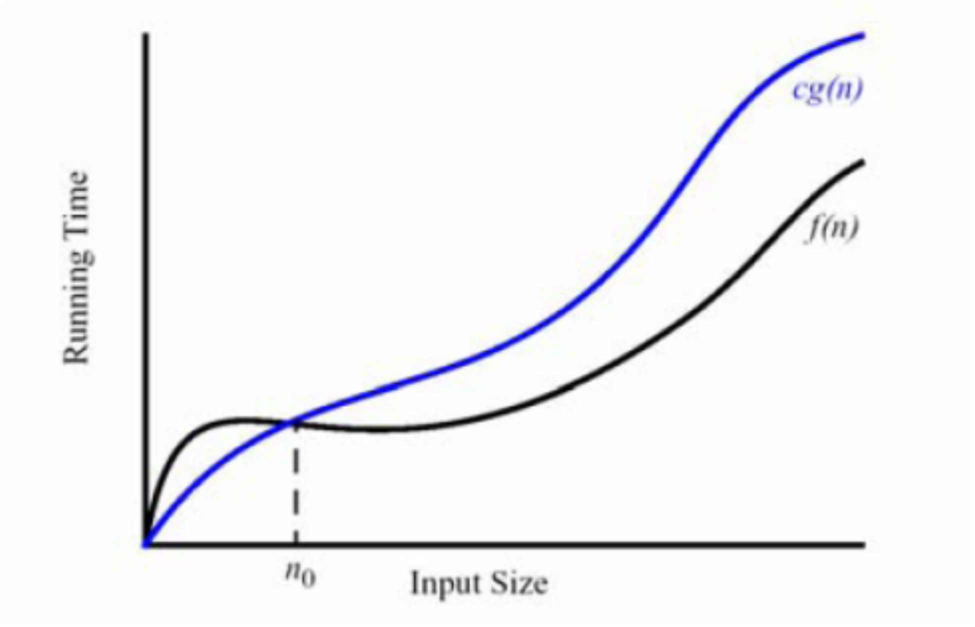
Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that:

$$f(n) \le cg(n) \quad \text{for } n \ge n_0$$

- We say…
  - "*f(n)* is ***Big-Oh*** of ***g(n)***" or
  - "*f(n)* is ***order*** ***g(n)***"

Big-Oh Illustrated



- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that:

$$f(n) \le cg(n) \text{ for } n \ge n_0$$

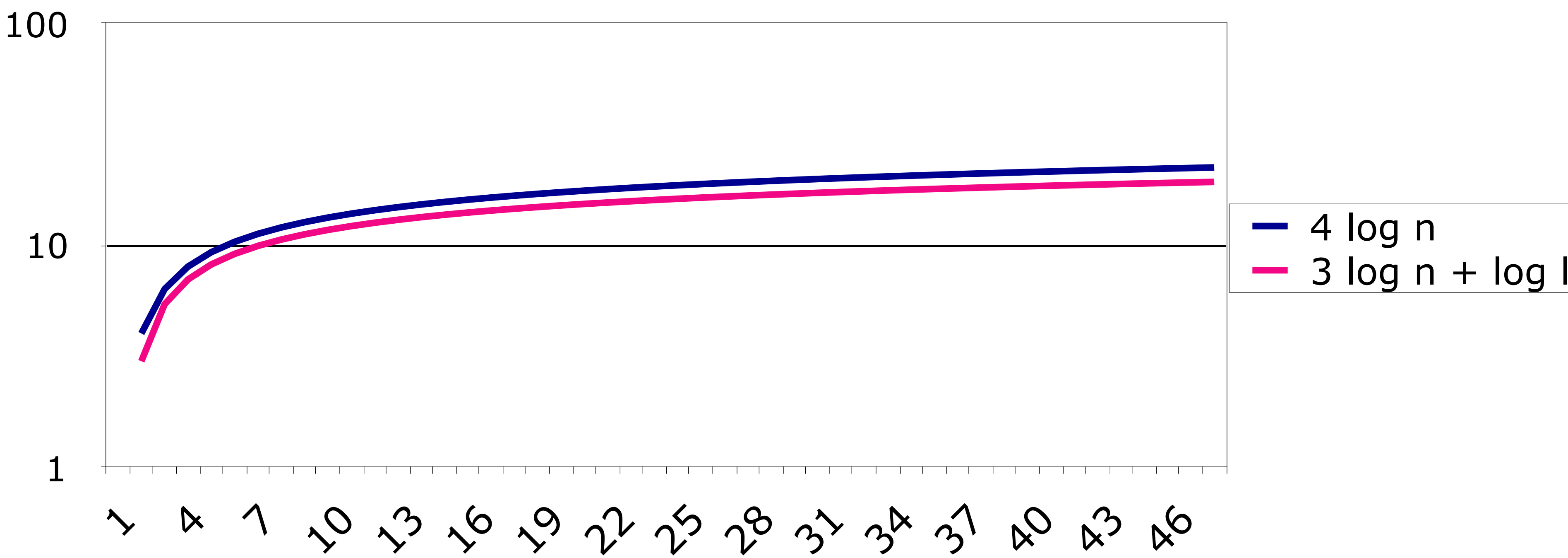
Typical Growth Rates
Big-Oh Notation

- Special Classes of Algorithms
  - Constant : $O(1)$
  - Logarithmic $O(\log n)$
  - Linear: $O(n)$
  - $O(n \log n)$ - "$n \log n$"
  - Quadratic: $O(n^2)$
  - Cubic: $O(n^3)$
  - Polynomial: $O(n^k)$, e.g., $O(n^6)$

- **Exponential:** $O(2^n)$, ***can be*** $O(a^n)$, ***where*** $a > 1$

Big-Oh

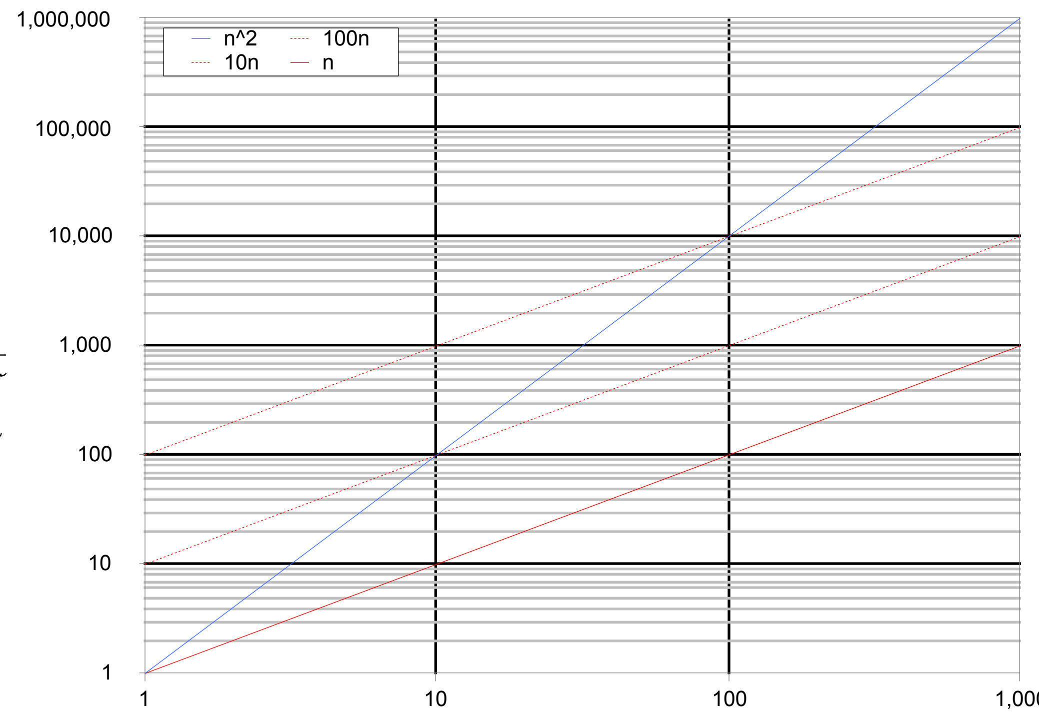
- ***Big-Oh***
  - **Definition** - $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$ for $n \geq n_0$
  - $f(n)$ is $O(g(n))$ if $f(n)$ is asymptotically **less than or equal** to $g(n)$
  - Example: $\boldsymbol{3 \log n + \log \log n = O(\log n)}$ for $c = 4$ and $n \geq 2$



Big-Oh : NOT O(n^2)

Example - the function n^2 is not $O(n)$

- $n^2 \le cn$
- $n \le c$
- The above inequality cannot be satisfied since c must be a constant.

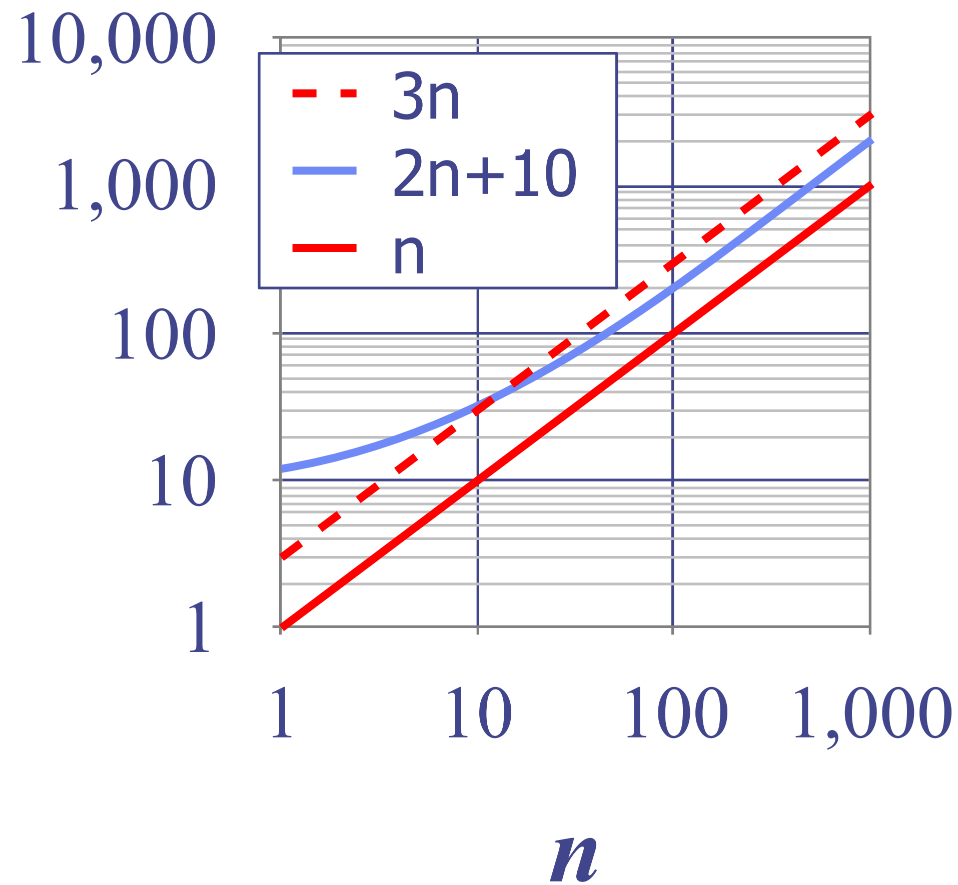


Big-Oh : O(n)

- **Definition** - Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that:

$$f(n) \leq cg(n) \quad \text{for } n \geq n_0$$

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



Growth of Several Functions

$$\log n < \log^2 n < \sqrt{n} < n < n \cdot \log n < n^2 < n^3 < 2^n$$

- Above functions are ordered by increasing growth rates.
- That is a function $f(n)$ preceded a function $g(n)$ in the list if $f(n)$ is $O(g(n))$.

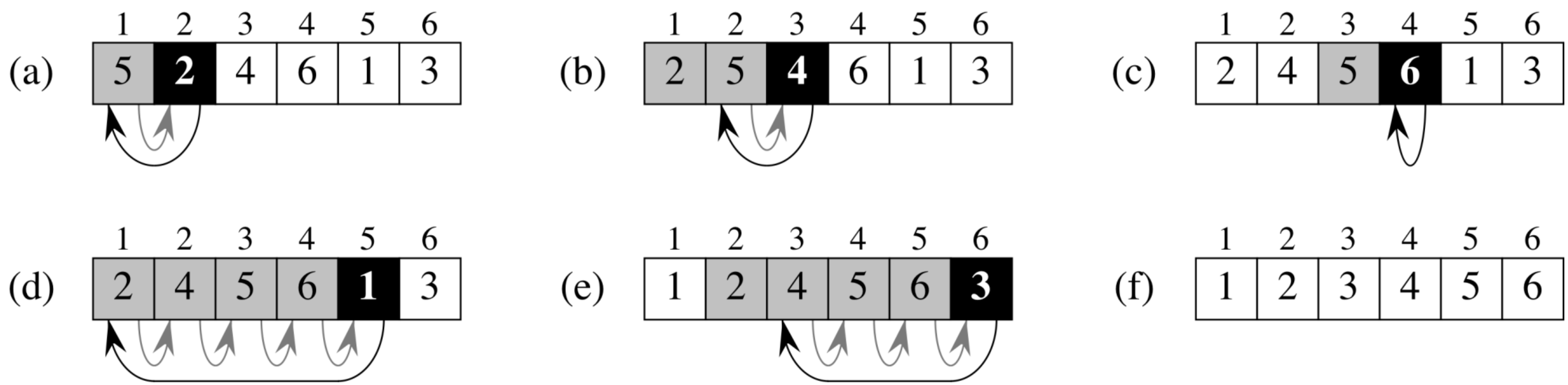
Sorting Problem

- **Input:** A sequence of *n* numbers $\{a_1, a_2, \ldots, a_n\}$
- **Output:** A permutation (reordering) $\{a'_1, a'_2, \ldots, a'_n\}$ of the input sequence such that $a'_I \leq a'_2 \leq \ldots \leq a'_n$

- **Problem statement** specifies in general terms desired input/output relationship.
- An **algorithm** is a tool for solving a well-specified **computational problem**.
- Can be several ways to solve particular problem

Insertion Sort

Example



Insertion-Sort Pseudocode

INSERTION-SORT(A, n)

for $j = 2$ **to** n
    $key = A[j]$
    // Insert $A[j]$ into the sorted sequence $A[1 \mathinner{.\,.} j-1]$.
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$

Rules for Analyzing Running Time

- Loops
  - The running time of a loop is at most the running time of the statements inside the loop times the number of iterations

```
for ( x = 0; x < N; x ++ ) {
        statement 1
        statement 2
        …
        statement c
}
```

N iterations
c statements

$O(cN) = O(N)$

Rules for Analyzing Running Time

- Nested Loops - analyze inside out
  - The running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all loops

```
for ( x = 0; x < N; x ++ ) {
            for ( y = 0; y < N; y ++ ) {
                        statement 1
            }
}
```

N*N iterations
1 statements

$O(N*N) = O(N^2)$

Rules for Analyzing Running Time

- Consecutive statements
  - Just add them together - largest one counts

```
statement 1
statement 2
for ( x = 0; x < N; x ++ ) {
        statement 3
}
```

2 statements

N iterations

$O(2+N) = O(N)$

Rules for Analyzing Running Time

- if/else statements
  - The running time is never more than the running time of the test plus the larger of the running times of statement S1 and S2

```
if ( condition )
        S1
else
        S2
```

O(running time of condition)
+
max (O(running time of S1),
O(running time of S2))

Analyzing Running Time

- (Stop here) Quick exercise - give the Big-Oh running time of the following code

```
for ( x = 0; x < N; x ++ )
      array[x] = x*N;

for (x = 0; x < N; x ++ ) {
      if ( x < (N/2) )                (
            cout << array[x];
      else
            for (  y = 0; y < N; y ++ )
                  cout << y*array[x];
}
```

Big-Oh Rules

- If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$.
  1. Drop lower-order terms
  2. Drop constant factors
     - $f(n) = 4n^4 + n^3 \Rightarrow O(n^4)$
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- You can combine growth rates
  - $O(f(n)) + O(g(n)) = O(f(n) + g(n))$
  - $O(n) + O(n^3 + 5) = O(n + n^3 + 5) = O(n^3)$

Calculating Big-Oh

- (Stop here) Quick exercise - Give the Big-Oh notation for the following functions:
  - $n + \log(n) =$
  - $8n \log(n) + n^2 =$
  - $6n^2 + 2^n + 300 =$
  - $n + n \log(n) + \log(n) =$
  - $40 + 8n + n^7 =$

Big-Omega Notation

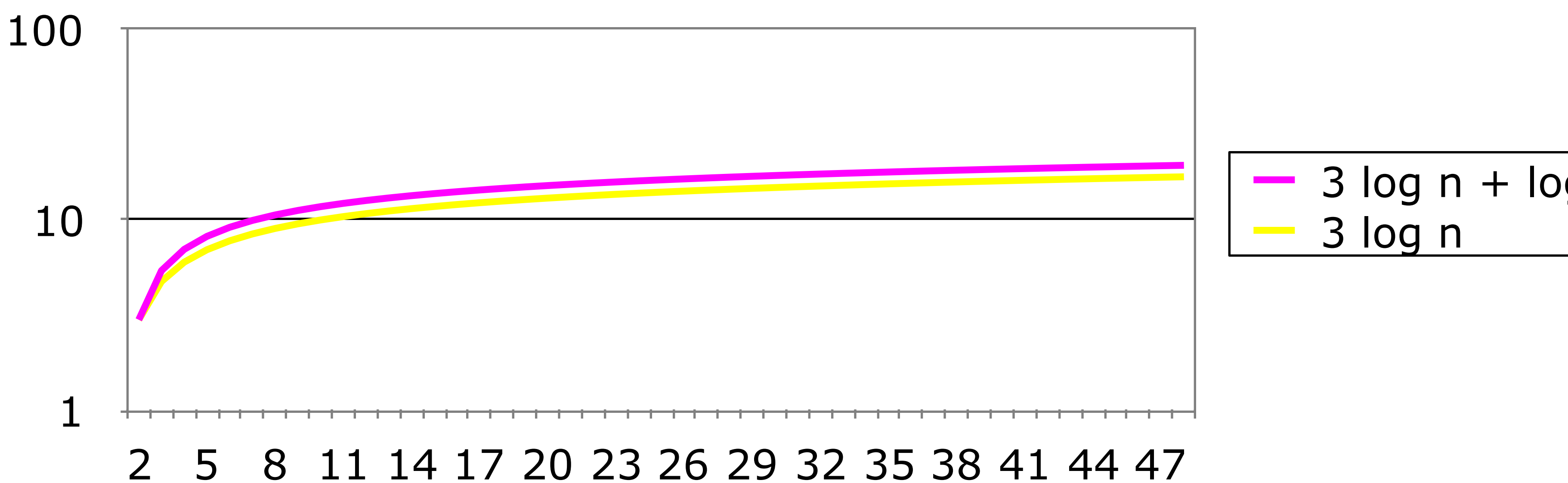
- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\Omega(g(n))$ if there are positive constants c and n_0 such that:

$$f(n) \geq cg(n) \quad \text{for } n \geq n_0$$

- We say that…
  - "*f(n)* is ***Big-Omega*** of *g(n)* if *g(n) is **Big-Oh*** of *f(n)* "

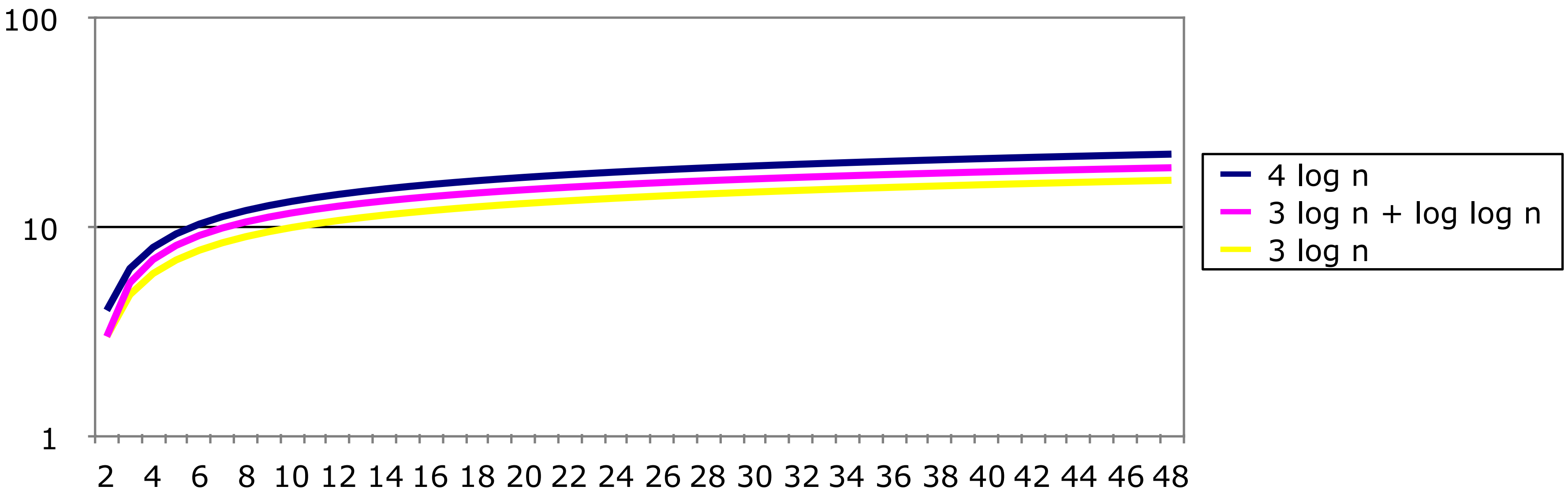
Big-Omega example

- ***Big-Omega***
  - **Definition** - $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq cg(n)$ for $n \geq n_0$
  - $f(n)$ is $\Omega(g(n))$ if $f(n)$ is asymptotically **greater than or equal** to $g(n)$
  - Example: $\boldsymbol{3 \log n + \log \log n = \Omega(\log n)}$ for c = 3 and n ≥ 2



Big-Theta example

- ***Big-Theta***
  - **Definition** - *f(n)* is $\Theta(g(n))$ if there are constants $c_1 > 0$ and $c_2 > 0$ and an integer constant $n_0 \geq 1$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for $n \geq n_0$
  - *f(n)* is $\Theta(g(n))$ if *f(n)* is asymptotically **equal** to *g(n)*
  - Example: $3 \log n + \log \log n = \Theta(\log n)$ for $c_1 = 3$ and $c_2 = 4$ and $n \geq 2$



Relatives of *Big-Oh*

"little-oh"

- $f(n)$ is $o(g(n))$ if, **for any** constant c > 0, there is an integer constant $n_0 \geq 0$ such that $f(n) < cg(n)$ for $n \geq n_0$
- $f(n)$ is $o(g(n))$ if $f(n)$ is asymptotically **strictly less** than $g(n)$

"little-omega"

- $f(n)$ is $\omega(g(n))$ if, **for any** constant $c > 0$, there is an integer constant $n_0 \geq 0$ such that $f(n) > cg(n)$ for $n \geq n_0$
- $f(n)$ is $\omega(g(n))$ if is asymptotically **strictly greater** than $g(n)$

Suppose each operation takes 1 nanoseconds (10^{-9} seconds)

n	lg n	n	n lg n	n^2	2^n	n!
10	0.003μs	0.01μs	0.033μs	0.1μs	1μs	3.63ms
20	0.004μs	0.02μs	0.086μs	0.4μs	1ms	77.1years
30	0.005μs	0.02μs	0.147μs	0.9μs	1sec	>10^{15}years
100	0.007μs	0.1μs	0.644μs	10μs	>10^{13}years	
10,000	0.013μs	10μs	130μs	100ms		
1,000,000	0.020μs	1ms	19.92 ms	16.7min		

- For n < 10, the difference is insignificant.
- Θ (n!) algorithms are useless well before n = 20.
- Θ (2^n) algorithms are practical for n < 40.
- Θ (n^2) and Θ (n lg n) are both useful, but Θ (n lg n) is significantly faster.

Array Running Times

- Unsorted insert
  - O(1) - add to end
- Sorted insert
  - O(N) - shift items
- Get number items
  - O(1) – keep a counter
  - Otherwise O(N)
- Print
  - O(N)
- Sorted remove
  - O(N) - shift items
- Unsorted remove
  - O(1) - move last
- Linear search
  - O(N)

Linked List Running Times

Singly-linked

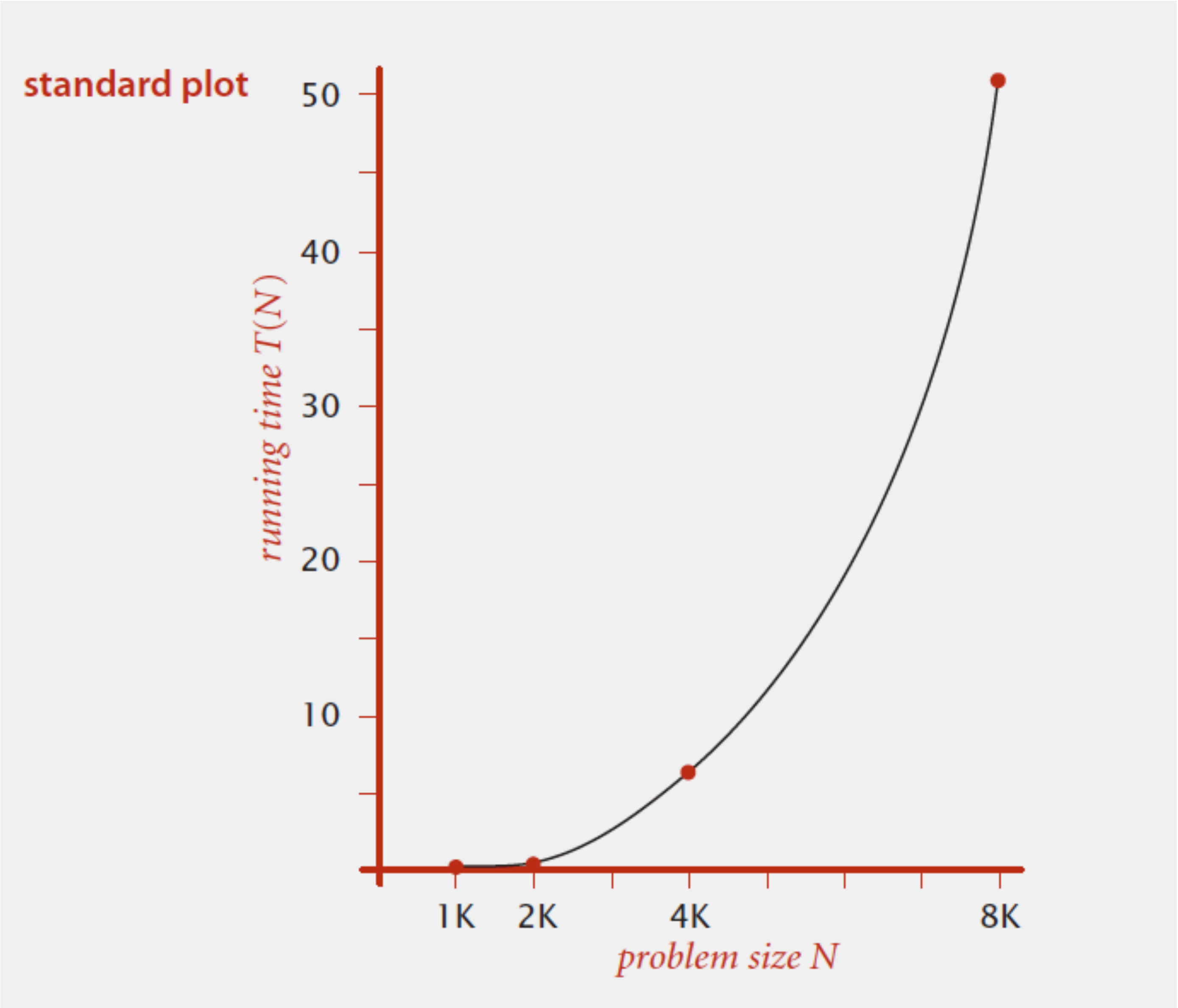
- **Insert head**
  - O(1)
- **Insert tail**
  - O(N)
- **Get number of items**
  - O(1) – counter
  - O(N) – traverse
- **Print**
  - O(N)
- **Remove**
  - O(N) – search
  - O(1) – remove
- **Linear Search**
  - O(N)

Linked List Running Times

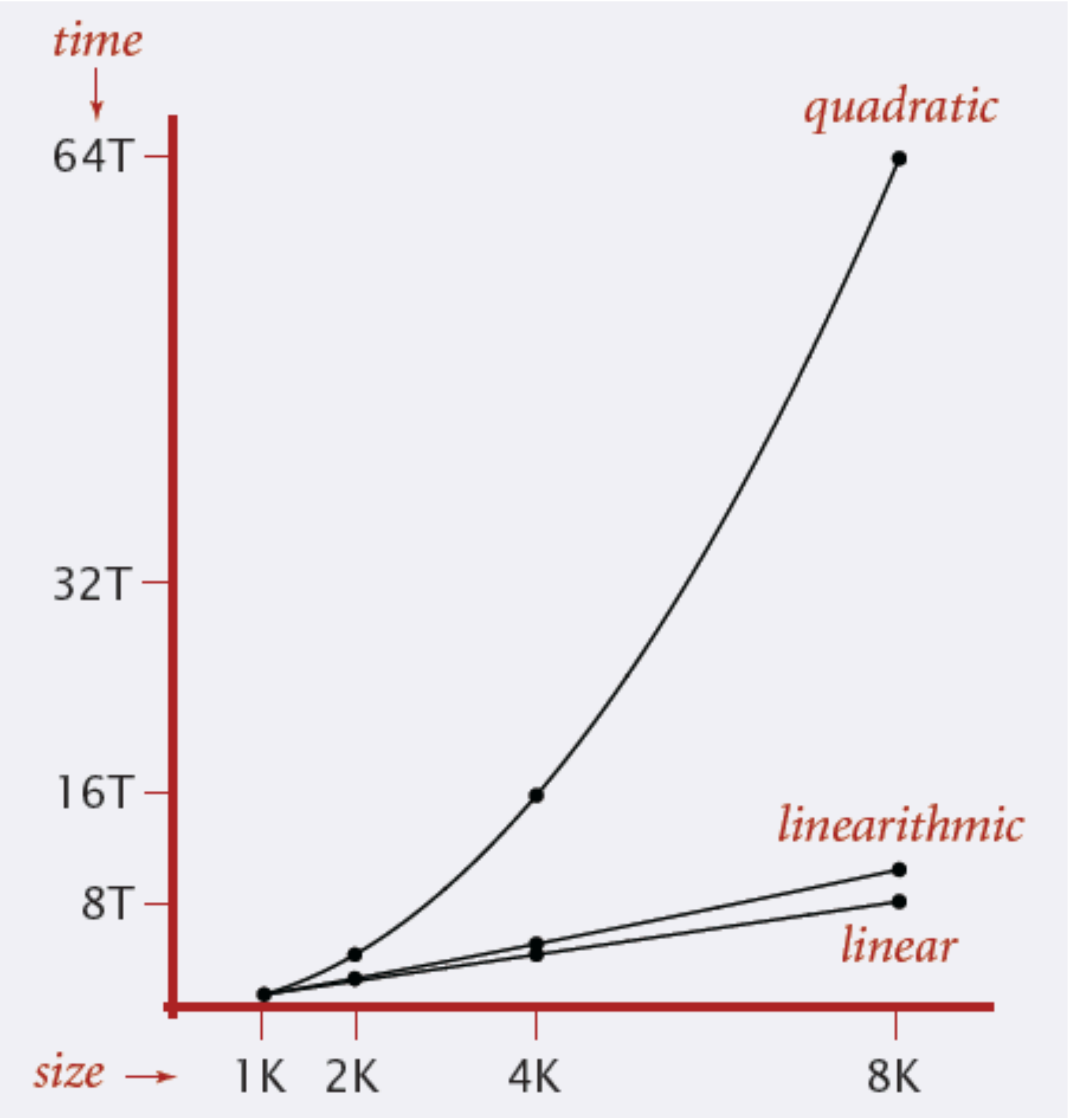
- Singly-linked (head ptr)
  - Insert head
    - O(1)
  - Insert tail
    - O(N)
  - Get number of items
    - O(1) – counter
    - O(N) – traverse
  - Print
    - O(N)
- Remove
  - O(N) – search
  - O(1) – remove
- Linear Search
  - O(N)

order of growth	name	typical code framework	description	example
1	**constant**	`a = b + c;`	statement	add two numbers
$\log N$	**logarithmic**	`while (N > 1) { N = N / 2; ... }`	divide in half	binary search
N	**linear**	`for (int i = 0; i < N; i++) { ... }`	loop	find the maximum
$N \log N$	**linearithmic**	[see mergesort lecture]	divide and conquer	mergesort
N^2	**quadratic**	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }`	double loop	check all pairs
N^3	**cubic**	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }`	triple loop	check all triples
2^N	**exponential**	[see combinatorial search lecture]	exhaustive search	check all subsets

Source: Algorithms 4th Edition, 2011, R. Sedgewick and K. Wayne



Source: Algorithms 4th Edition, 2011, R. Sedgewick and K. Wayne



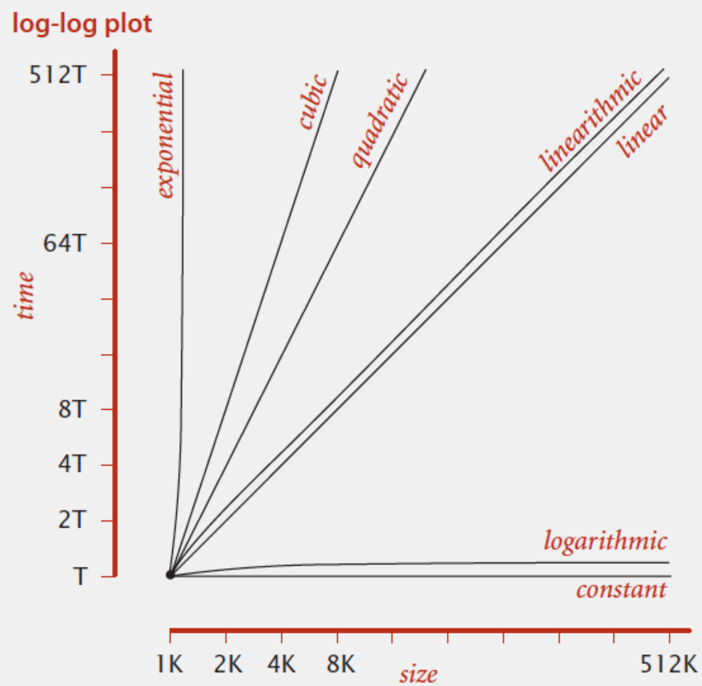
Source: Algorithms 4th Edition, 2011, R. Sedgewick and K. Wayne

Common order-of-growth classifications

Good news. The set of functions

$$1,\ \log N,\ N,\ N \log N,\ N^2,\ N^3, \text{ and } 2^N$$

suffices to describe the order of growth of most common algorithms.



Typical orders of growth

Source: Algorithms 4th Edition, 2011, R. Sedgewick and K. Wayne