POLITECNICO
DI TORINO

e-Lite

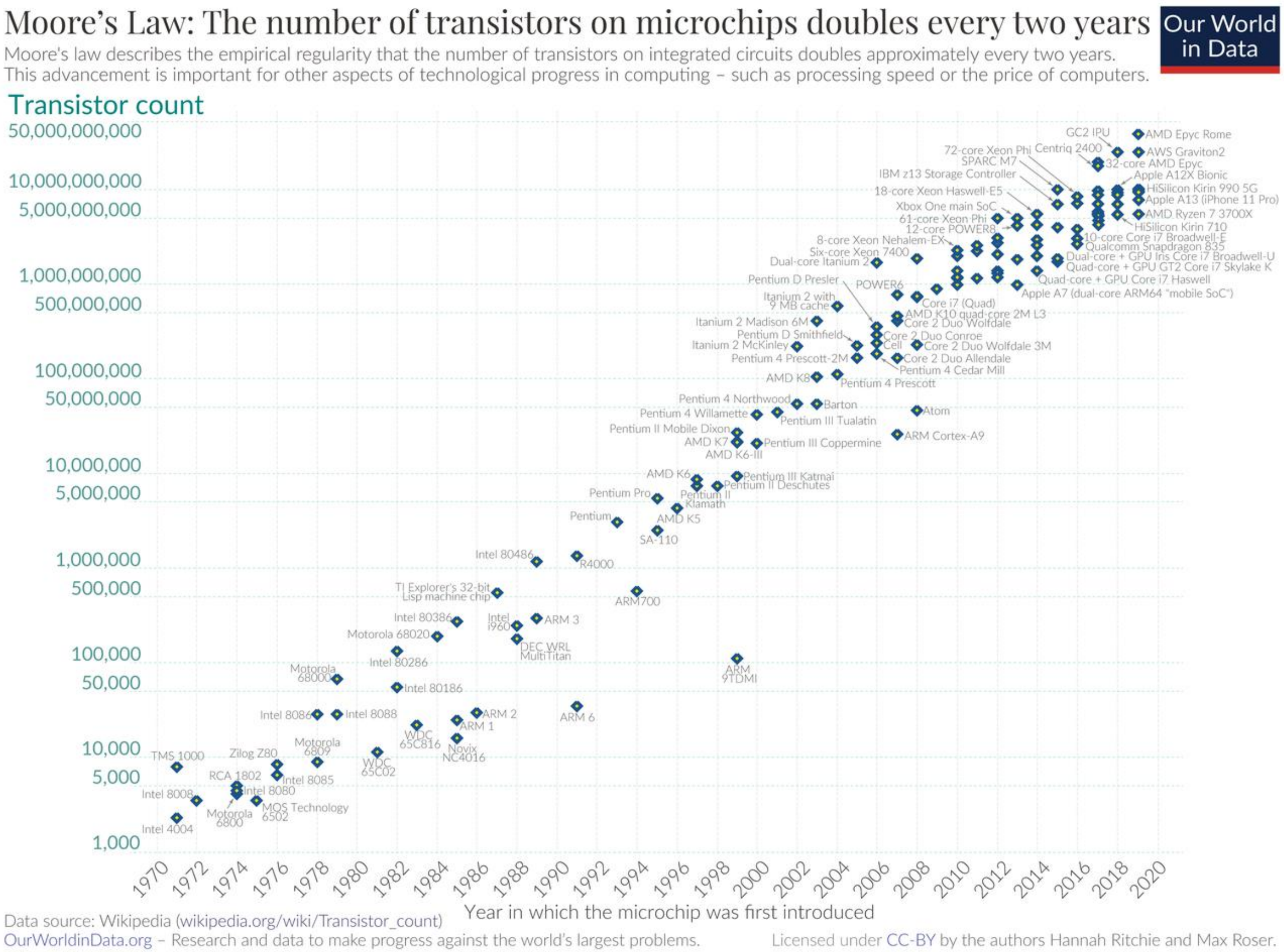
Computational complexity

Tecniche di Programmazione – A.A. 2020/2021

How to Measure Efficiency?

- Critical resources
  - programmer's effort
  - time, space (disk, RAM)
- Analysis
  - empirical (run programs)
  - analytical (asymptotic algorithm analysis)
- Worst case vs. Average case

Moore's "Law"?



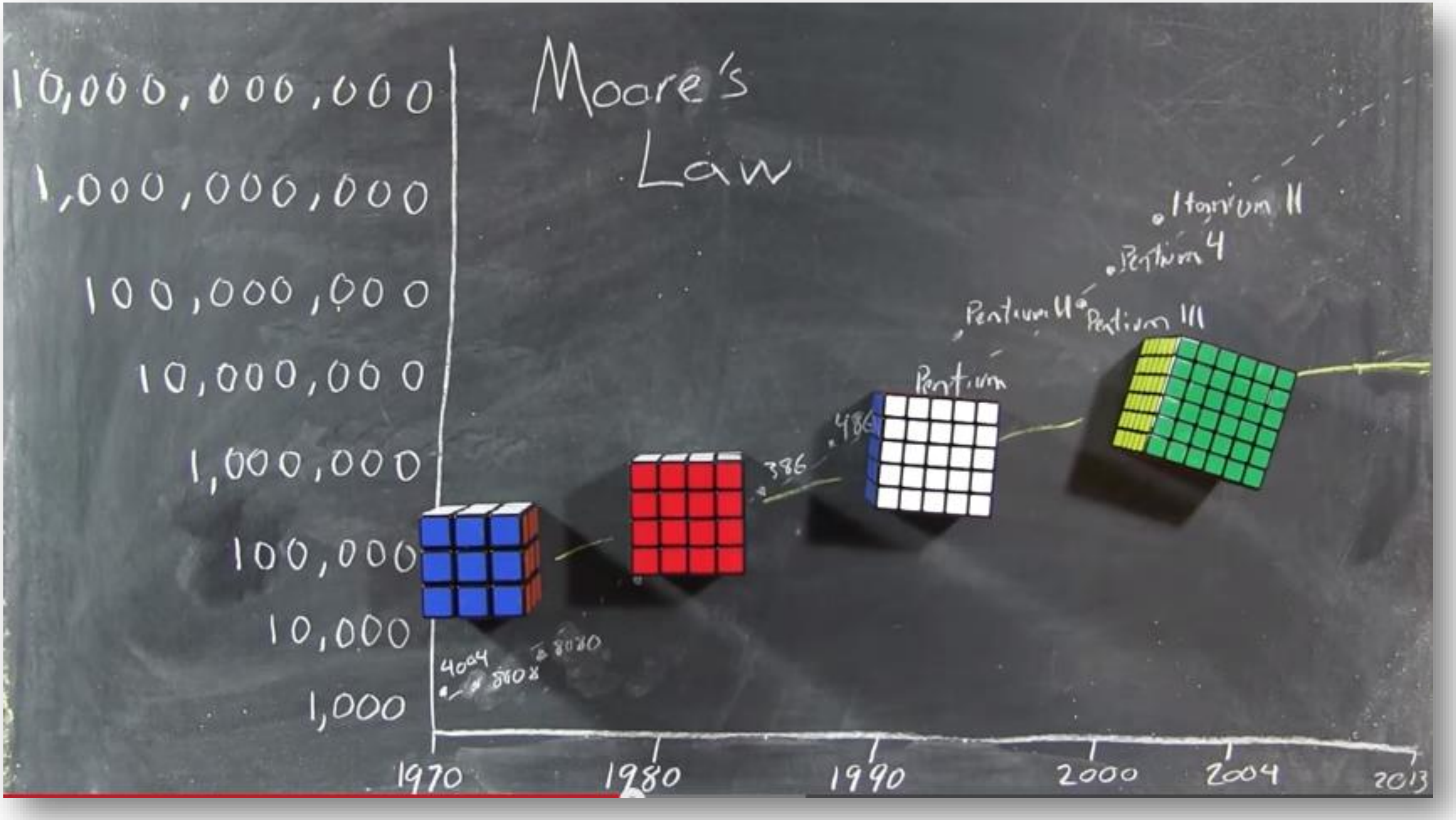
Moore's Law: The number of transistors on microchips doubles every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years.
This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)
OurWorldinData.org – Research and data to make progress against the world's largest problems.

https://en.wikipedia.org/wiki/Moore%27s_law

Moore's "Law"?


Moore's Law
10,000,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000
4004
8008
8080
386
486
Pentium
Pentium II
Pentium III
Pentium 4
Itanium II
1970
1980
1990
2000
2004
2013

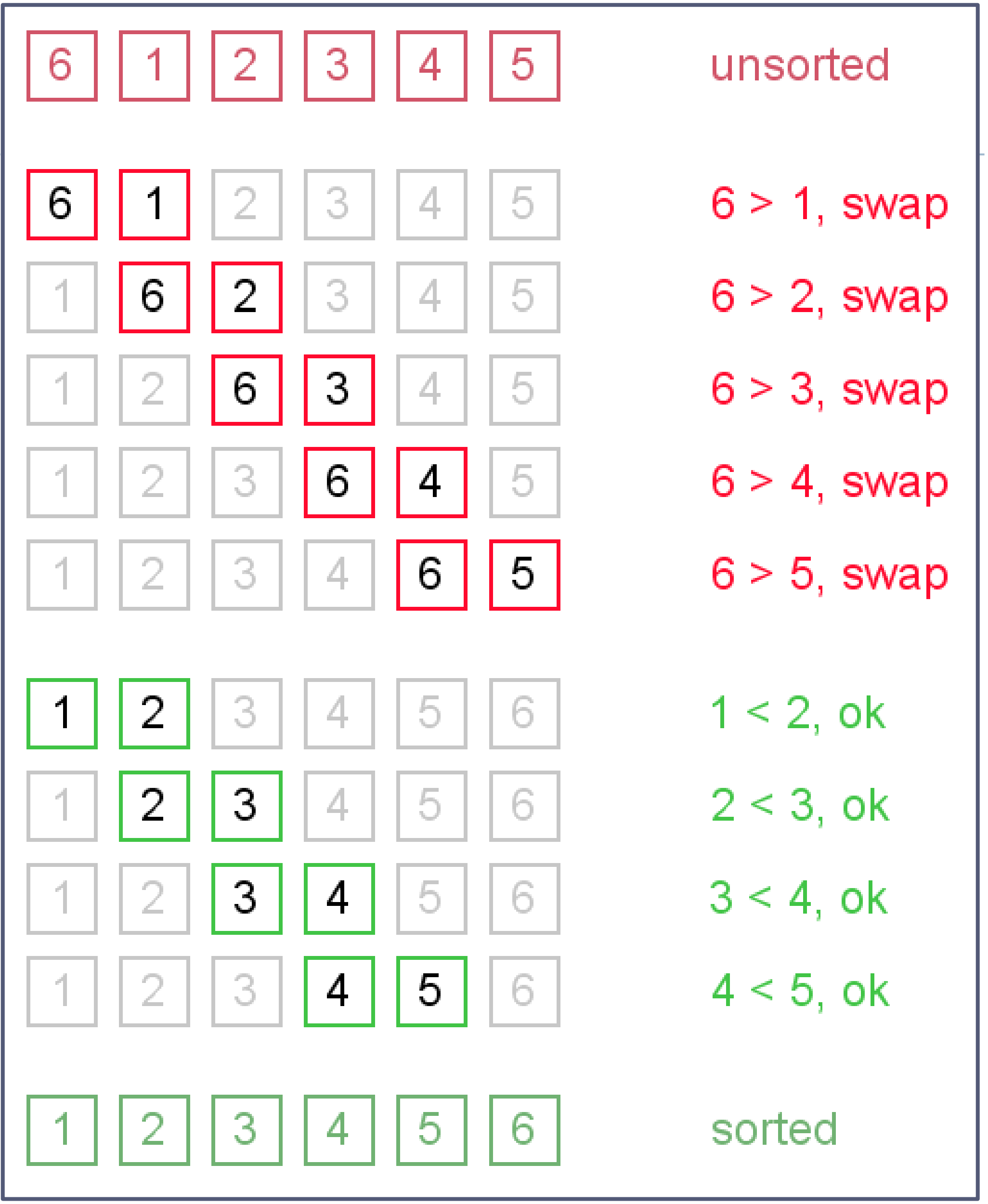
Problems and Algorithms

- We know the efficiency of the solution
- … but what about the difficulty of the problem?
- Different concepts
  - Algorithm complexity
  - Problem complexity

Analytical Approach

- For most algorithms, running time depends on “size” of the input
- Running time is expressed as T(n)
  - some function T
  - input *size* n

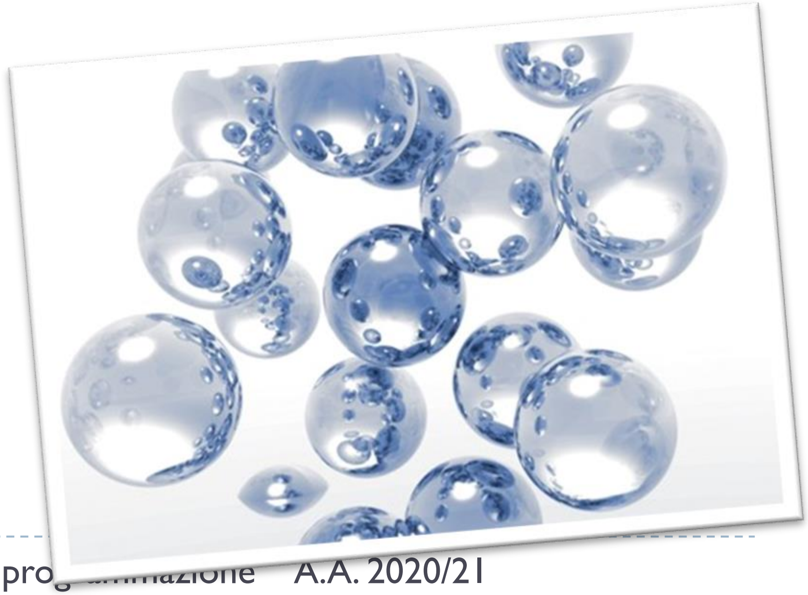
Bubble sort



6	1	2	3	4	5	unsorted
6	1	2	3	4	5	6 > 1, swap
1	6	2	3	4	5	6 > 2, swap
1	2	6	3	4	5	6 > 3, swap
1	2	3	6	4	5	6 > 4, swap
1	2	3	4	6	5	6 > 5, swap
1	2	3	4	5	6	1 < 2, ok
1	2	3	4	5	6	2 < 3, ok
1	2	3	4	5	6	3 < 4, ok
1	2	3	4	5	6	4 < 5, ok
1	2	3	4	5	6	sorted

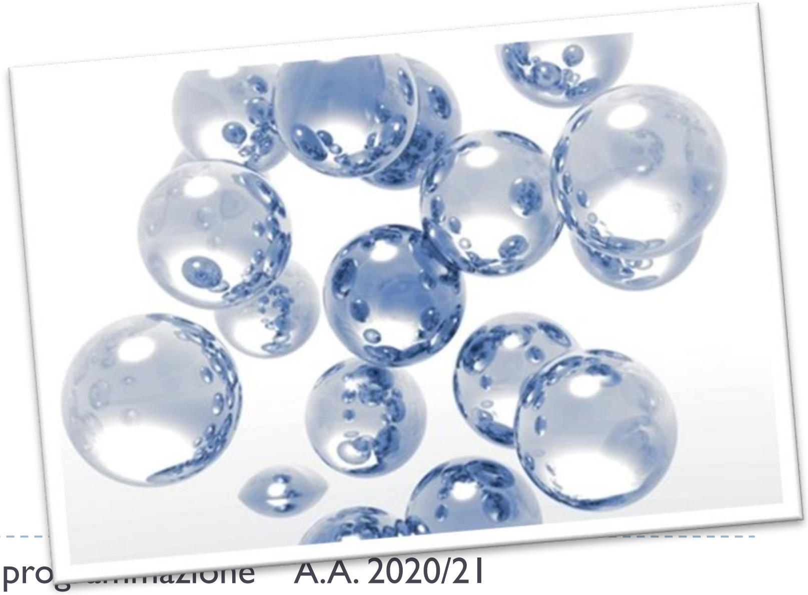
Analysis

- The bubble sort takes $(n^2-n)/2$ "steps"
- Different implementations/assembly languages
  - Program A on an Intel Pentium IV: $T(n) = 58*(n^2-n)/2$
  - Program B on a Motorola: $T(n) = 84*(n^2-2n)/2$
  - Program C on an Intel Pentium V: $T(n) = 44*(n^2-n)/2$
- Note that each has an n^2 term
  - as n increases, the other terms will drop out



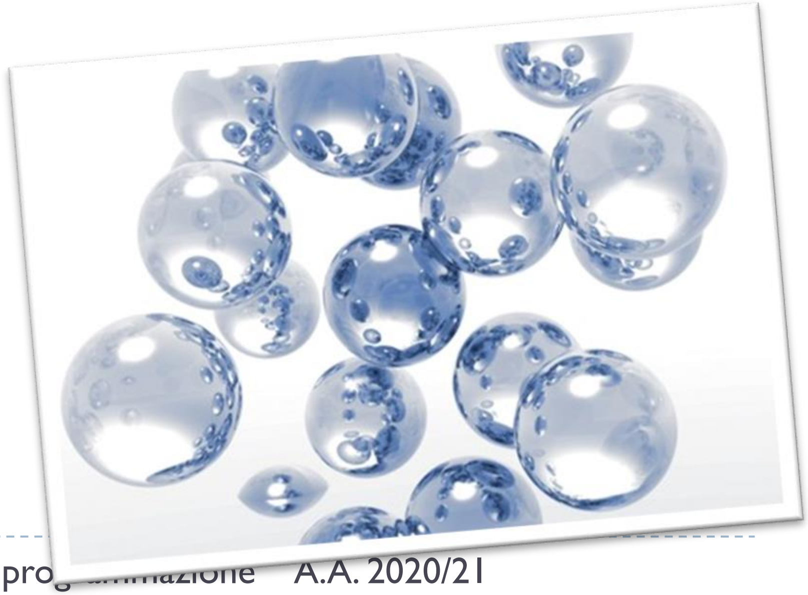
Analysis

- As a result:
  - Program A on Intel Pentium IV: $T(n) \approx 29n^2$
  - Program B on Motorola: $T(n) \approx 42n^2$
  - Program C on Intel Pentium V: $T(n) \approx 22n^2$



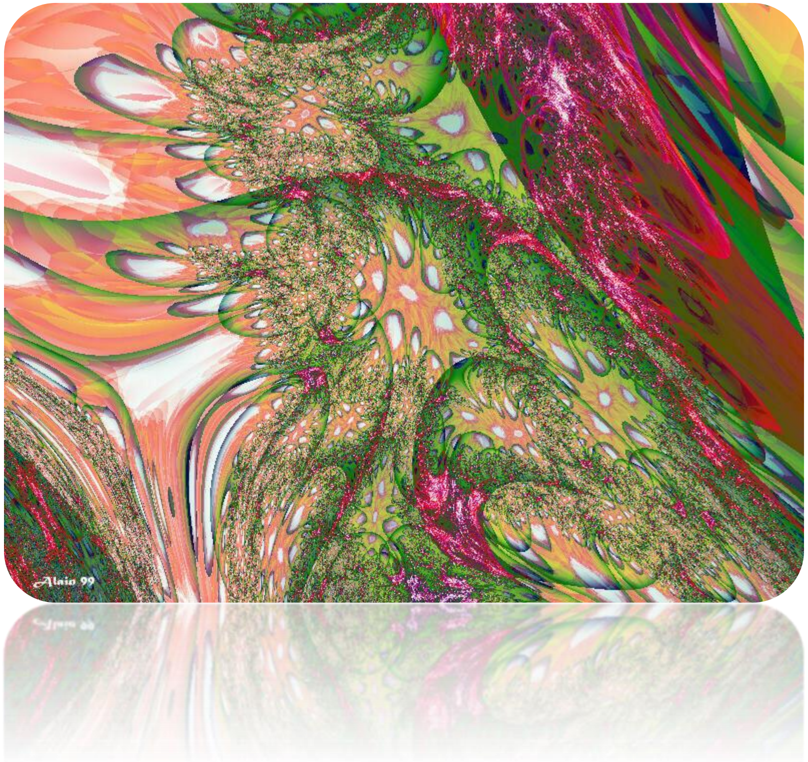
Analysis

- As processors change, the constants will always change
  - The exponent on n will not
  - We should not care about the constants
- As a result:
  - Program A: $T(n) \approx n^2$
  - Program B: $T(n) \approx n^2$
  - Program C: $T(n) \approx n^2$
- Bubble sort: $T(n) \approx n^2$



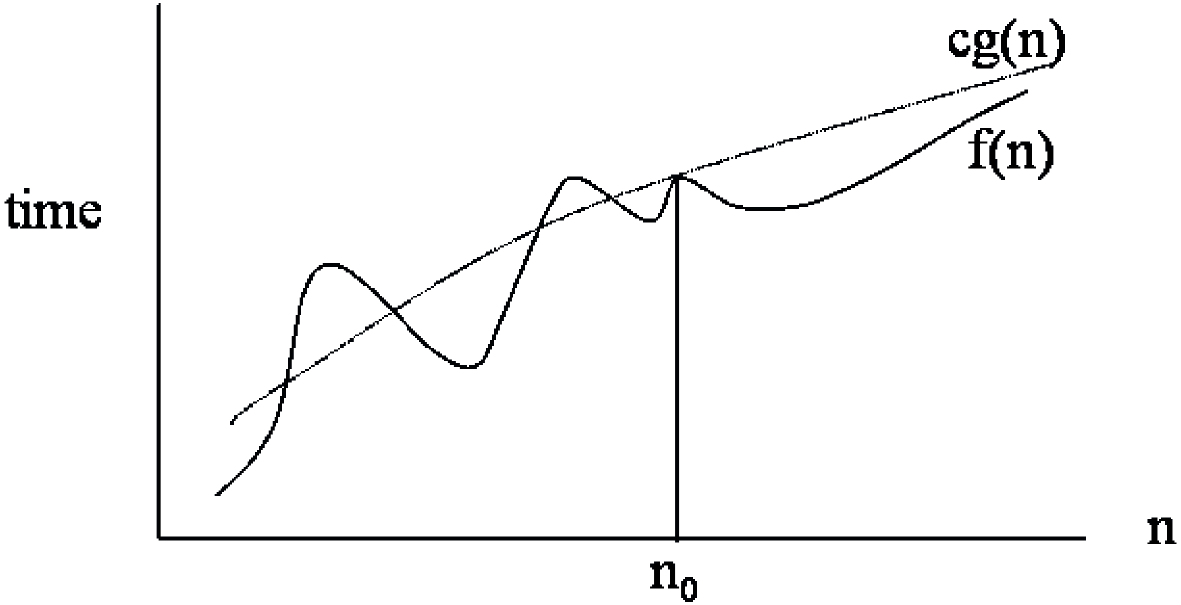
Complexity Analysis

- $O(\cdot)$
  - big o (big oh)
- $\Omega(\cdot)$
  - big omega
- $\Theta(\cdot)$
  - big theta



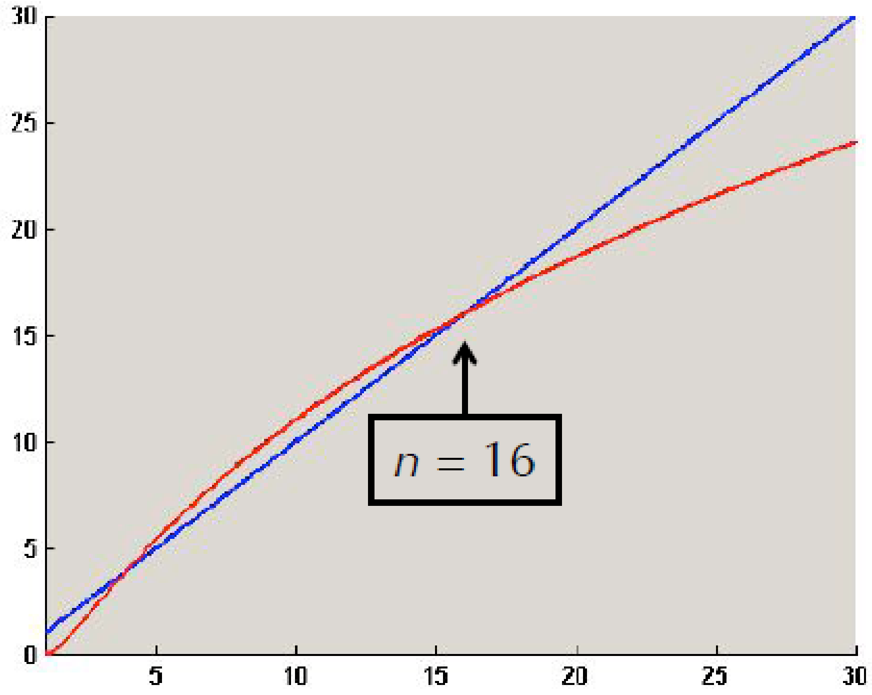
O(·) = Upper Bounding Running Time

- Upper Bounding Running Time
- f(n) is O(g(n)) if f grows “at most as fast as” g



Example

- $(\log n)^2 = O(n)$



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

$g(n) = n$

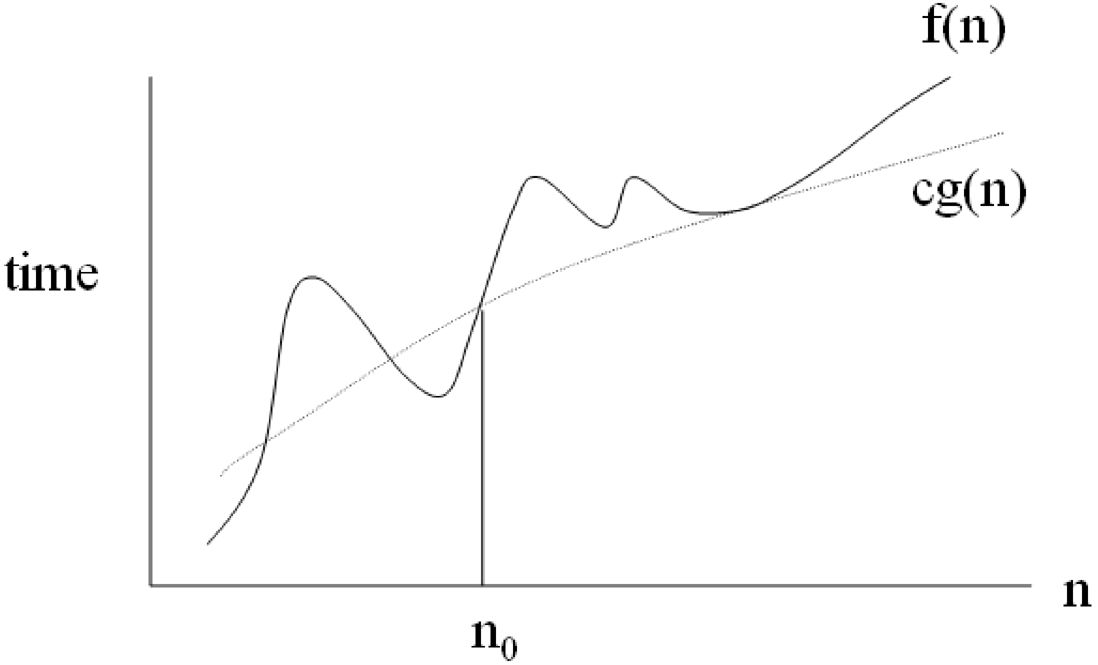
$(\log n)^2 \leq n$ for all $n \geq 16$, so $(\log n)^2$ is $O(n)$

Common Misunderstanding

- $3x^3 + 5x^2 – 9 = O(x^3)$
- However, also true are:
  - $3x^3 + 5x^2 – 9 = O(x^4)$
  - $x^3 = O(3x^3 + 5x^2 – 9)$
  - $\sin(x) = O(x^4)$
- Note:
  - Usage of big-O typically involves mentioning only the most dominant term
  - “The running time is $O(x^{2.5})$”

Ω(·) = Lower Bounding Running Time

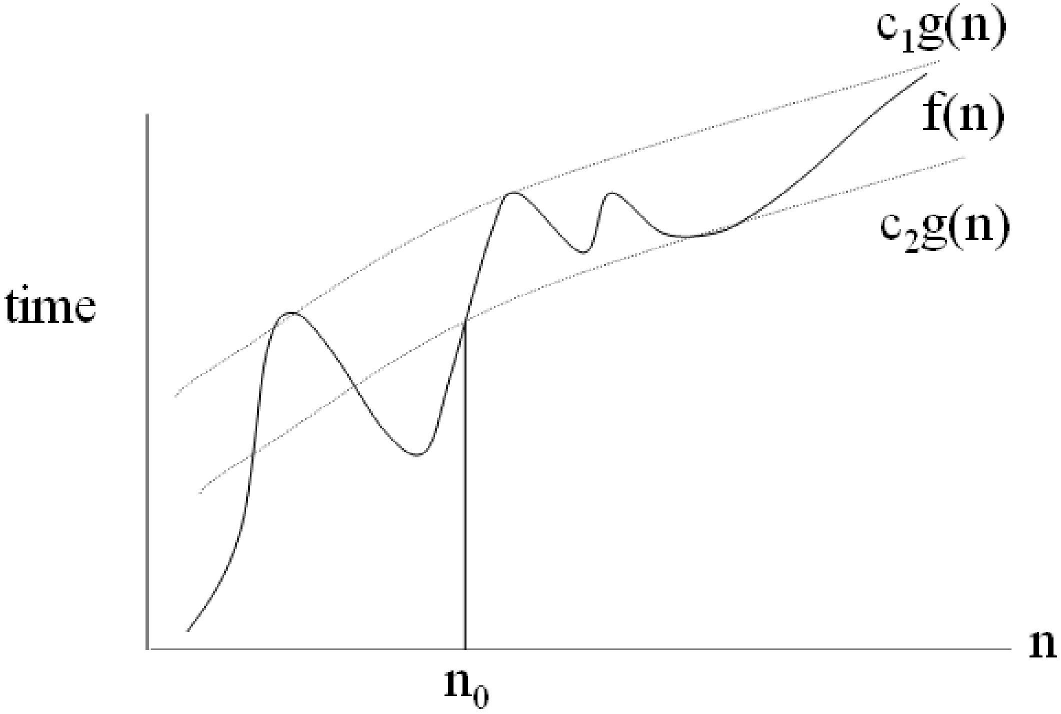
- f(n) is Ω(g(n)) if f grows "at least as fast as" g



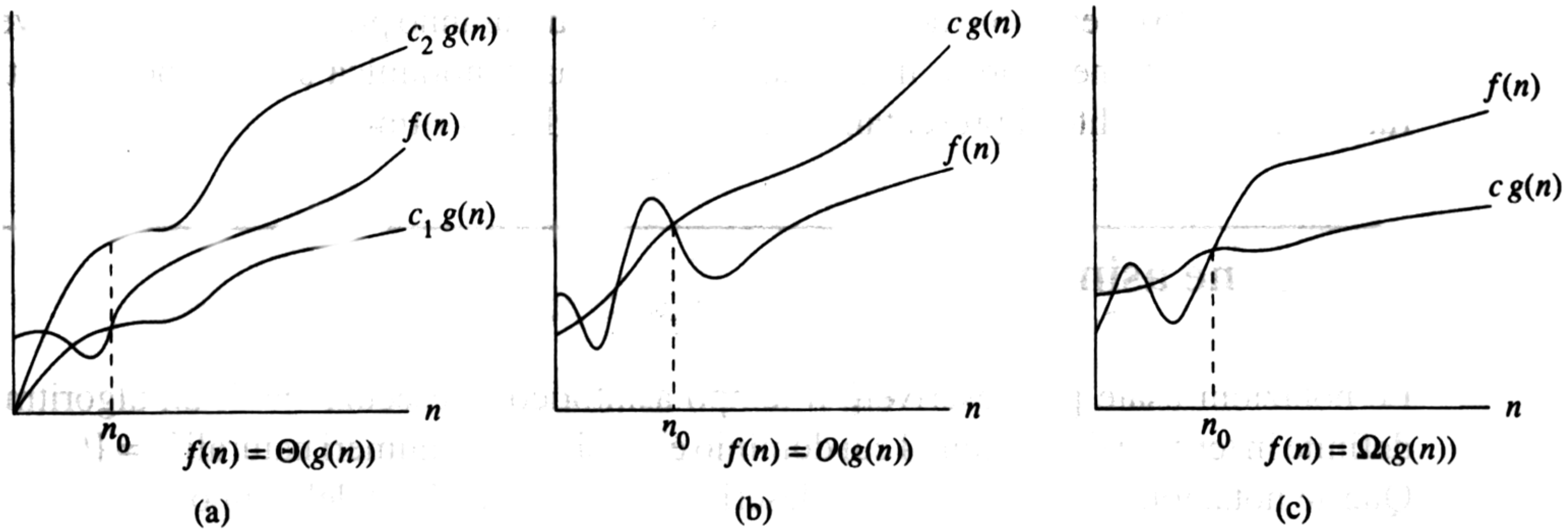
- **cg(n) is an approximation to f(n) bounding from below**

Θ(·) = Tightly Bounding Running Time

- f(n) is Θ(g(n)) if f is essentially the same as g, to within a constant multiple



Big-Θ, Big-O, and Big-Ω



$f(n) = \Theta(g(n))$

(a)

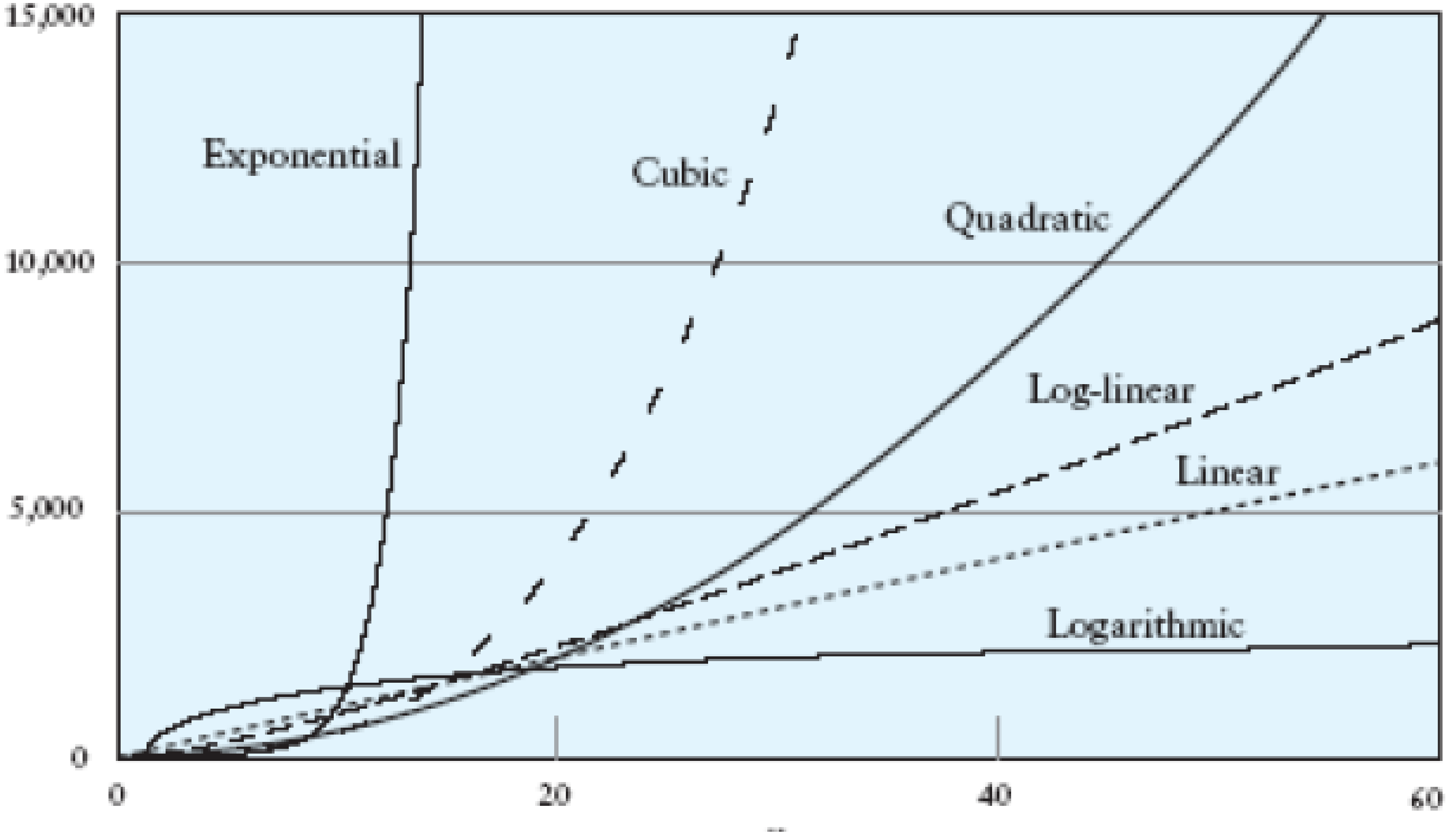
$f(n) = O(g(n))$

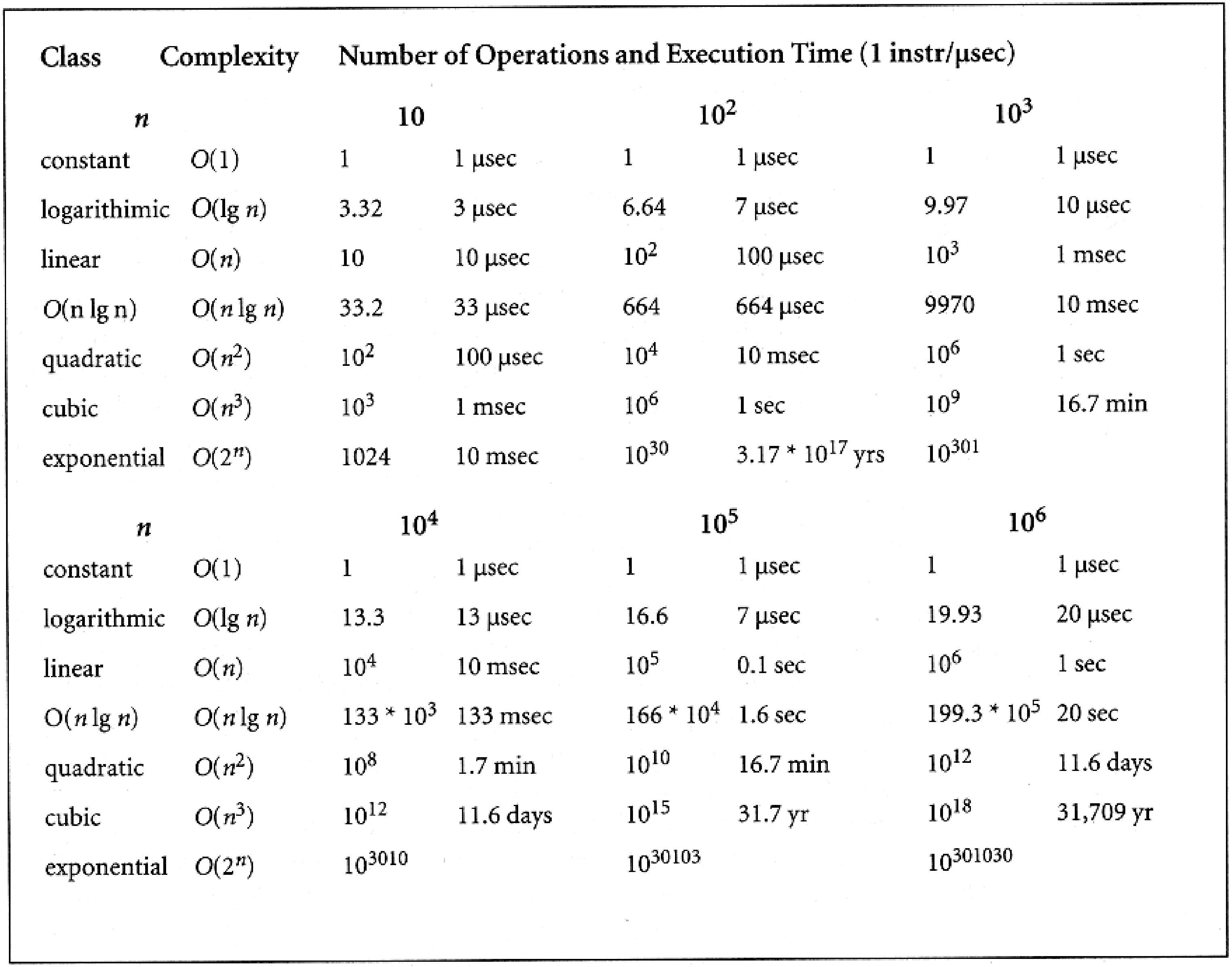
(b)

$f(n) = \Omega(g(n))$

(c)

Practical approach

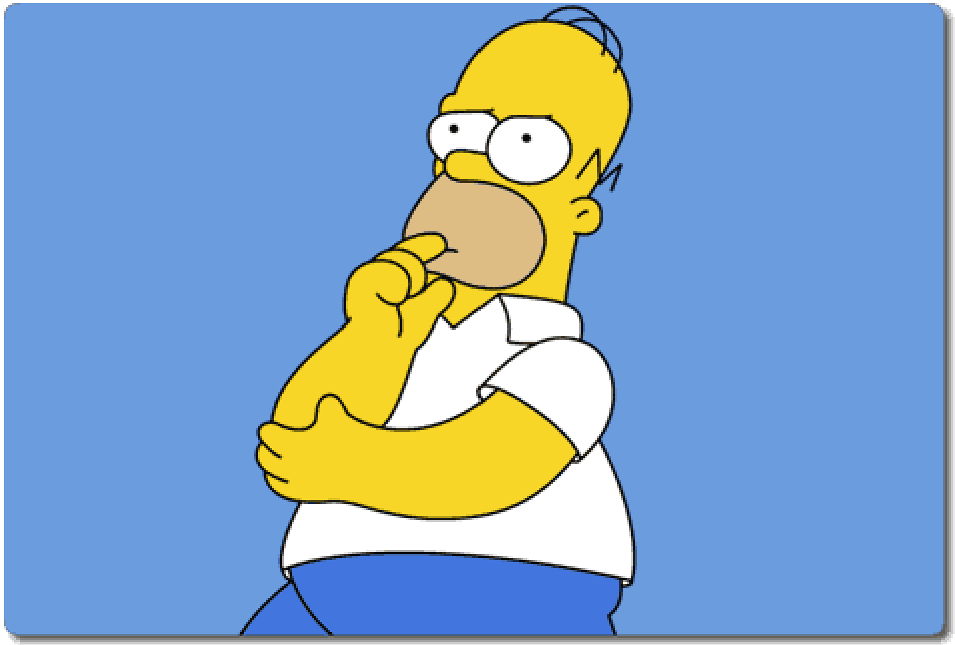




Class	Complexity	Number of Operations and Execution Time (1 instr/μsec)					
n		10		10^2		10^3	
constant	$O(1)$	1	1 μsec	1	1 μsec	1	1 μsec
logarithimic	$O(\lg n)$	3.32	3 μsec	6.64	7 μsec	9.97	10 μsec
linear	$O(n)$	10	10 μsec	10^2	100 μsec	10^3	1 msec
O(n lg n)	$O(n \lg n)$	33.2	33 μsec	664	664 μsec	9970	10 msec
quadratic	$O(n^2)$	10^2	100 μsec	10^4	10 msec	10^6	1 sec
cubic	$O(n^3)$	10^3	1 msec	10^6	1 sec	10^9	16.7 min
exponential	$O(2^n)$	1024	10 msec	10^{30}	3.17 * 10^{17} yrs	10^{301}	
n		10^4		10^5		10^6	
constant	$O(1)$	1	1 μsec	1	1 μsec	1	1 μsec
logarithmic	$O(\lg n)$	13.3	13 μsec	16.6	7 μsec	19.93	20 μsec
linear	$O(n)$	10^4	10 msec	10^5	0.1 sec	10^6	1 sec
O(*n* lg *n*)	$O(n \lg n)$	133 * 10^3	133 msec	166 * 10^4	1.6 sec	199.3 * 10^5	20 sec
quadratic	$O(n^2)$	10^8	1.7 min	10^{10}	16.7 min	10^{12}	11.6 days
cubic	$O(n^3)$	10^{12}	11.6 days	10^{15}	31.7 yr	10^{18}	31,709 yr
exponential	$O(2^n)$	10^{3010}		10^{30103}		10^{301030}	

Would it be possible?

Algorithm	Foo	Bar
Complexity	$O(n^2)$	$O(2^n)$
n = 100	10s	4s
n = 1000	12s	4.5s



Determination of Time Complexity

- Because of the approximations available through Big-O , the actual T(n) of an algorithm is not calculated
  - T(n) may be determined empirically
- Big-O is usually determined by application of some simple 5 rules

Rule #1

- **Simple** program **statements** are assumed to take a constant amount of time which is

O(1)

Rule #2

- Differences in execution time of simple statements is ignored

Rule #3

- In **conditional** statements the worst case is always used

Rule #4 – the “sum” rule

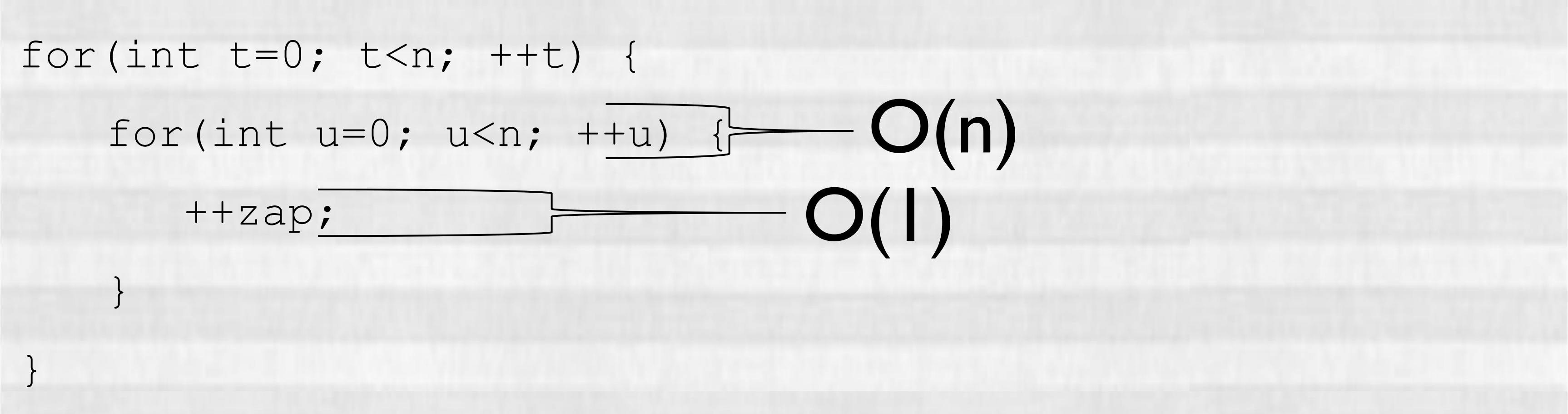
- The running time of a **sequence** of steps has the order of the running time of the largest
- E.g.,
  - $f(n) = O(n^2)$
  - $g(n) = O(n^3)$
  - $f(n) + g(n) = O(n^3)$

Worst case
(valid for big-O, not
for big-θ)

Rule #5 – the “product” rule

- If two processes are constructed such that the second process is **repeated** a number of times for each execution of the first process, then O is equal to the **product** of the orders of magnitude of the two processes
- E.g.,
  - For example, a two-dimensional array has one for loop inside another and each internal loop is executed n times for each value of the external loop.
  - The function is $O(n^2)$

Nested Loops

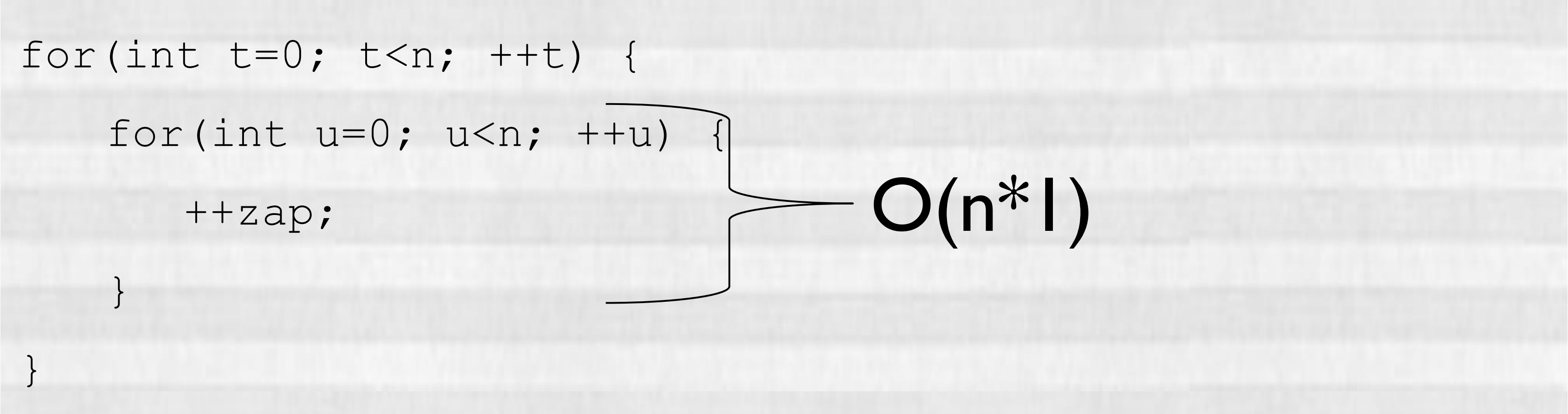


```
for(int t=0; t<n; ++t) {
    for(int u=0; u<n; ++u) {
        ++zap;
    }
}
```

O(n)

O(1)

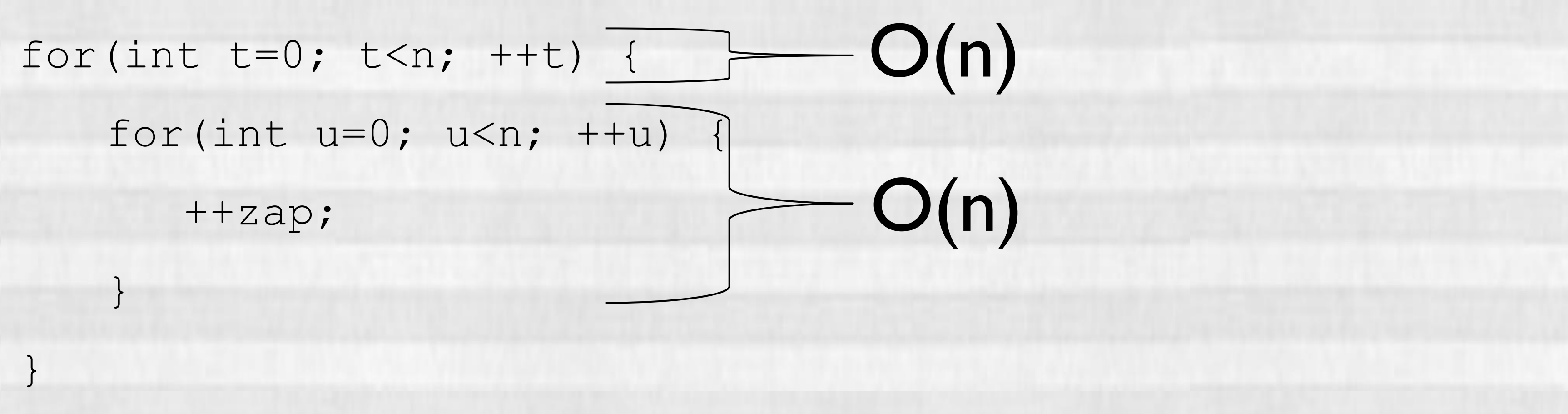
Nested Loops



```
for(int t=0; t<n; ++t) {
    for(int u=0; u<n; ++u) {
        ++zap;
    }
}
```

O(n*1)

Nested Loops



```
for(int t=0; t<n; ++t) {
   for(int u=0; u<n; ++u) {
      ++zap;
   }
}
```

Nested Loops

```
for(int t=0; t<n; ++t) {
    for(int u=0; u<n; ++u) {
        ++zap;
    }
}
```

$O(n^2)$

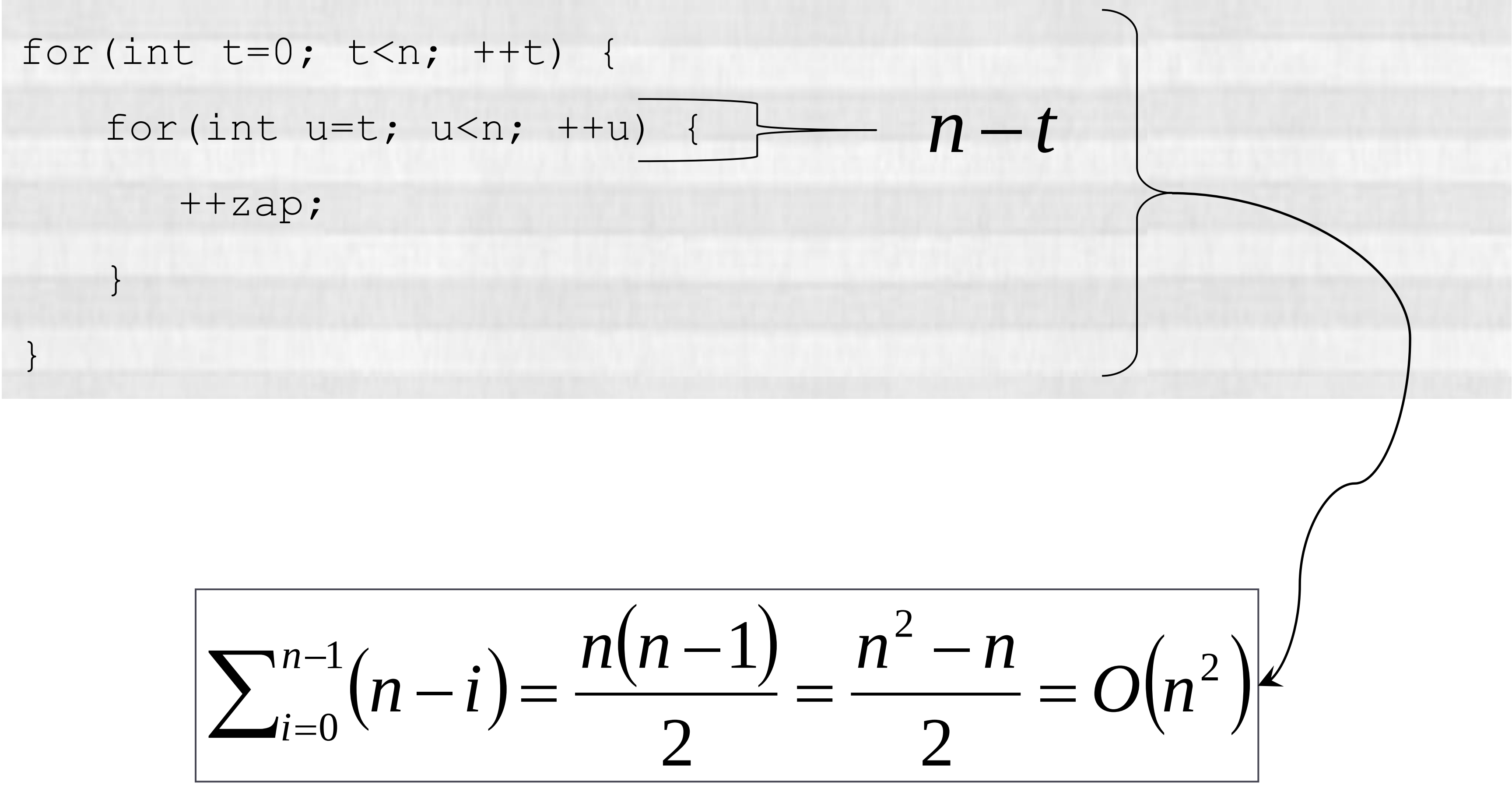
Nested Loops

- Note: Running time grows with nesting rather than the length of the code

```
for(int t=0; t<n; ++t) {
    for(int u=0; u<n; ++u) {
        ++zap;
    }
}
```

$O(n^2)$

More Nested Loops



```
for(int t=0; t<n; ++t) {
    for(int u=t; u<n; ++u) {
        ++zap;
    }
}
```

$n-t$

$$\sum_{i=0}^{n-1}(n-i)=\frac{n(n-1)}{2}=\frac{n^2-n}{2}=O\left(n^2\right)$$

Sequential statements

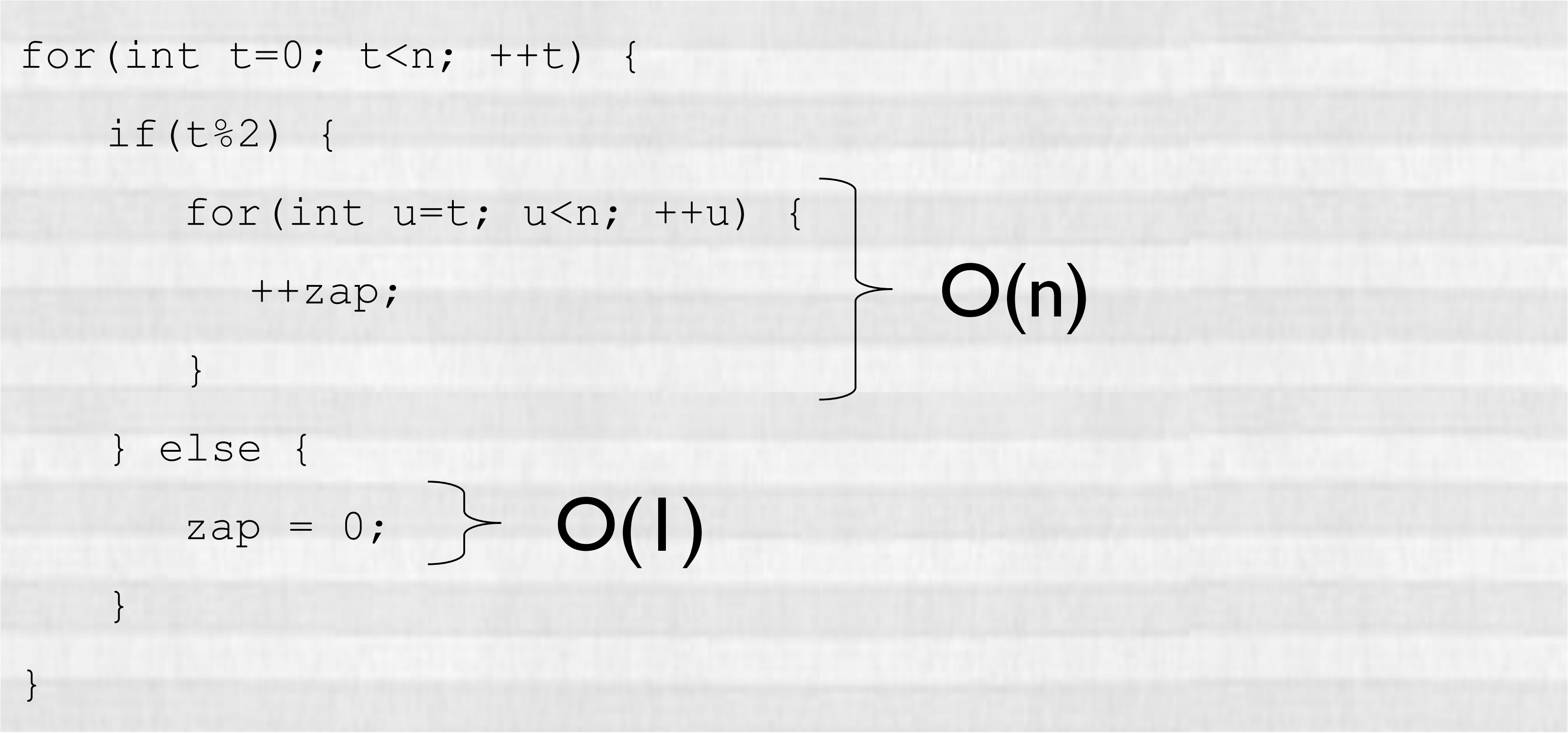
```
for(int z=0; z<n; ++z)
    zap[z] = 0;
for(int t=0; t<n; ++t) {
    for(int u=t; u<n; ++u) {
        ++zap;
    }
}
```

$O(n)$

$O(n^2)$

- Running time: $\max(O(n), O(n^2)) = O(n^2)$

Conditionals



```
for(int t=0; t<n; ++t) {
   if(t%2) {
      for(int u=t; u<n; ++u) {
         ++zap;
      }
   } else {
      zap = 0;
   }
}
```

Conditionals

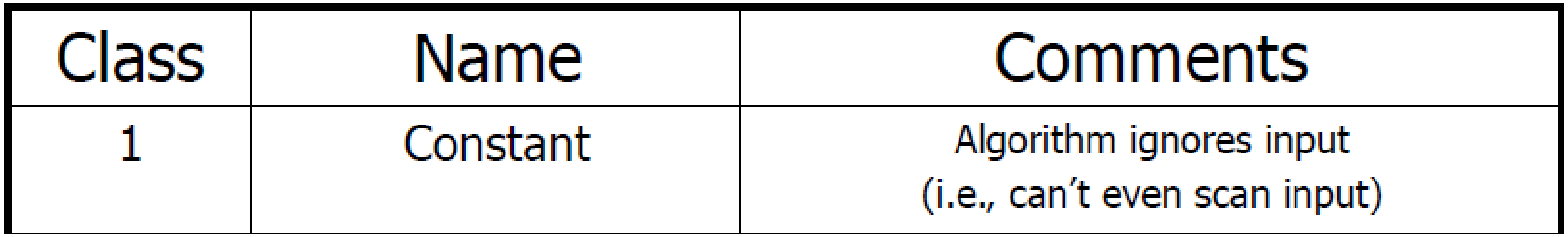
```
for(int t=0; t<n; ++t) {
    if(t%2) {
        for(int u=t; u<n; ++u) {
            ++zap;
        }
    } else {
        zap = 0;
    }
}
```

$O(n^2)$

Tips

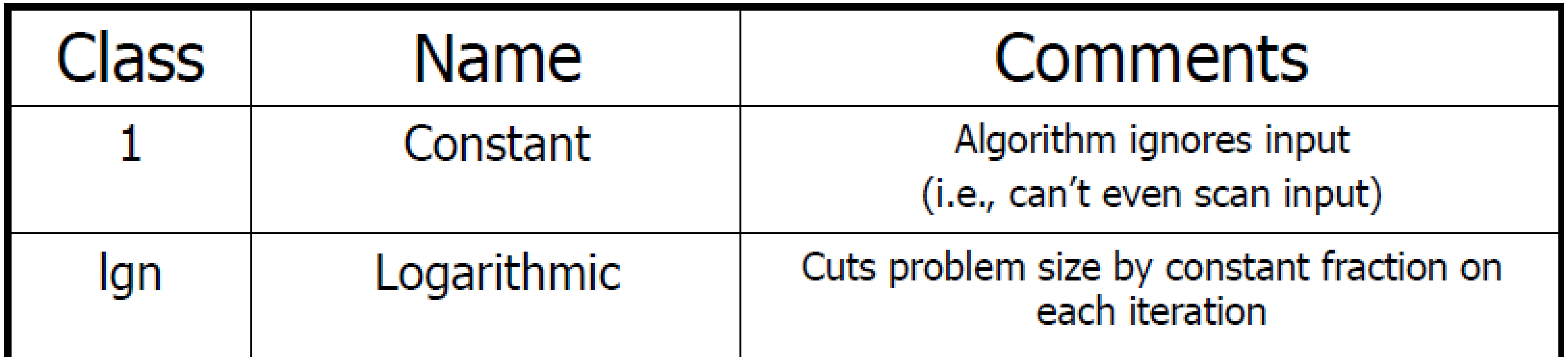
- Focus only on the dominant (high cost) operations and avoid a line-by-line exact analysis
- Break algorithm down into "known" pieces
- Identify relationships between pieces
  - Sequential is additive
  - Nested (loop / recursion) is multiplicative
- Drop constants
- Keep only dominant factor for each variable

Basic Asymptotic Efficiency Classes



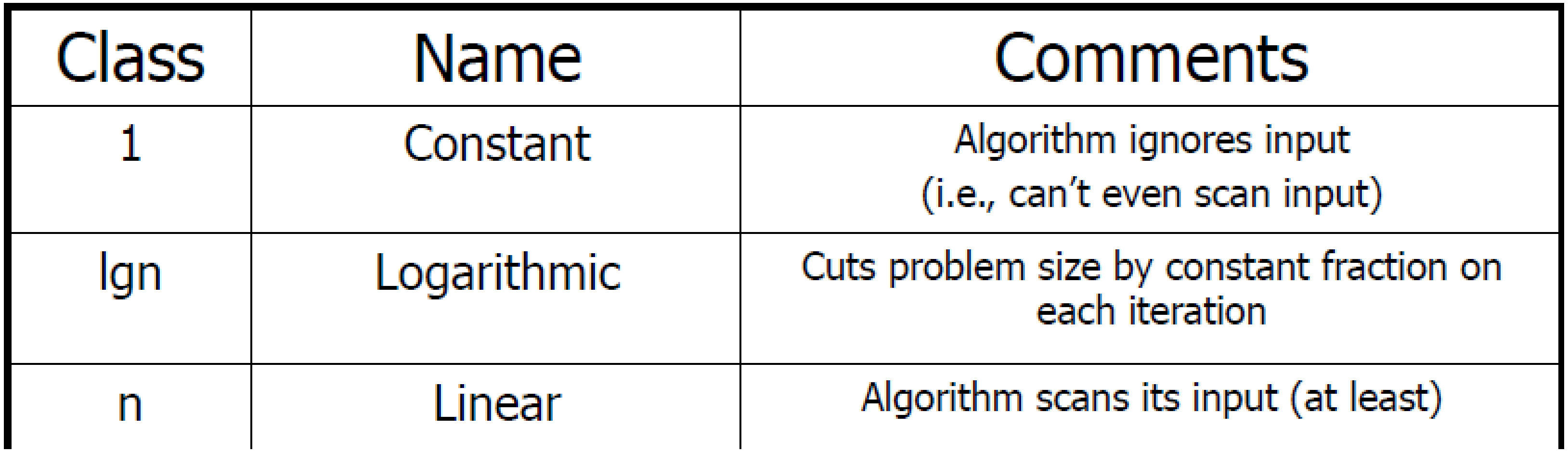
Class	Name	Comments
1	Constant	Algorithm ignores input (i.e., can't even scan input)

Basic Asymptotic Efficiency Classes



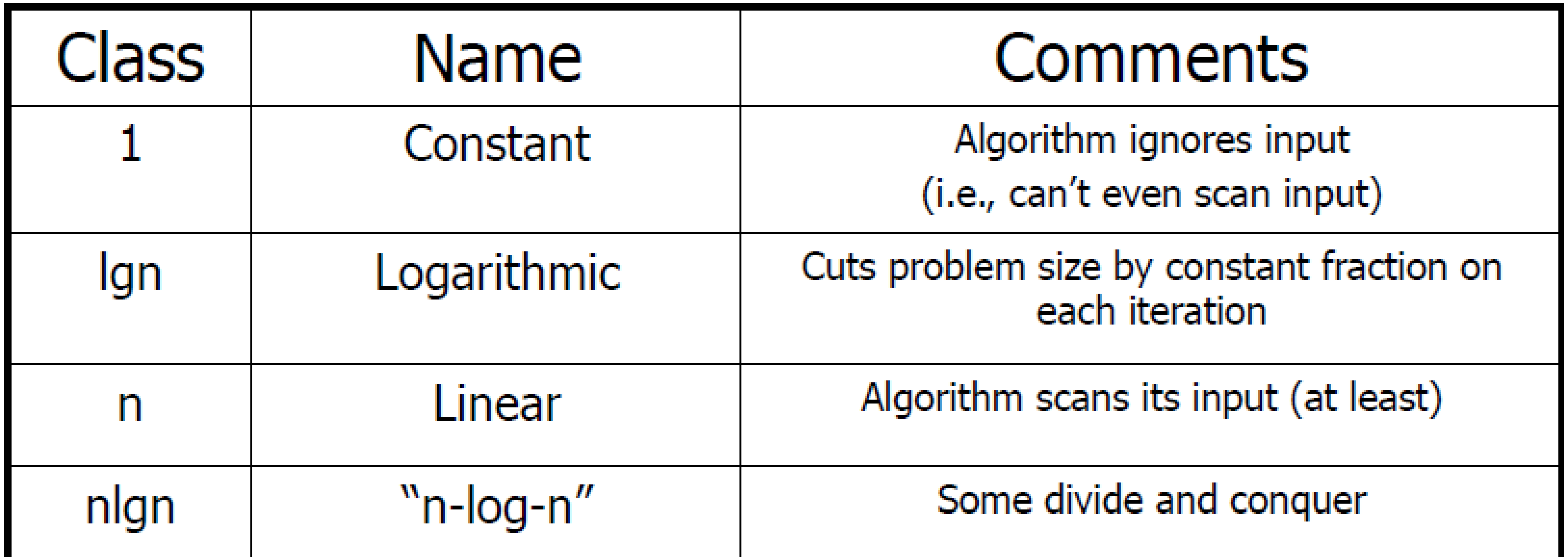
Class	Name	Comments
1	Constant	Algorithm ignores input (i.e., can't even scan input)
lgn	Logarithmic	Cuts problem size by constant fraction on each iteration

Basic Asymptotic Efficiency Classes



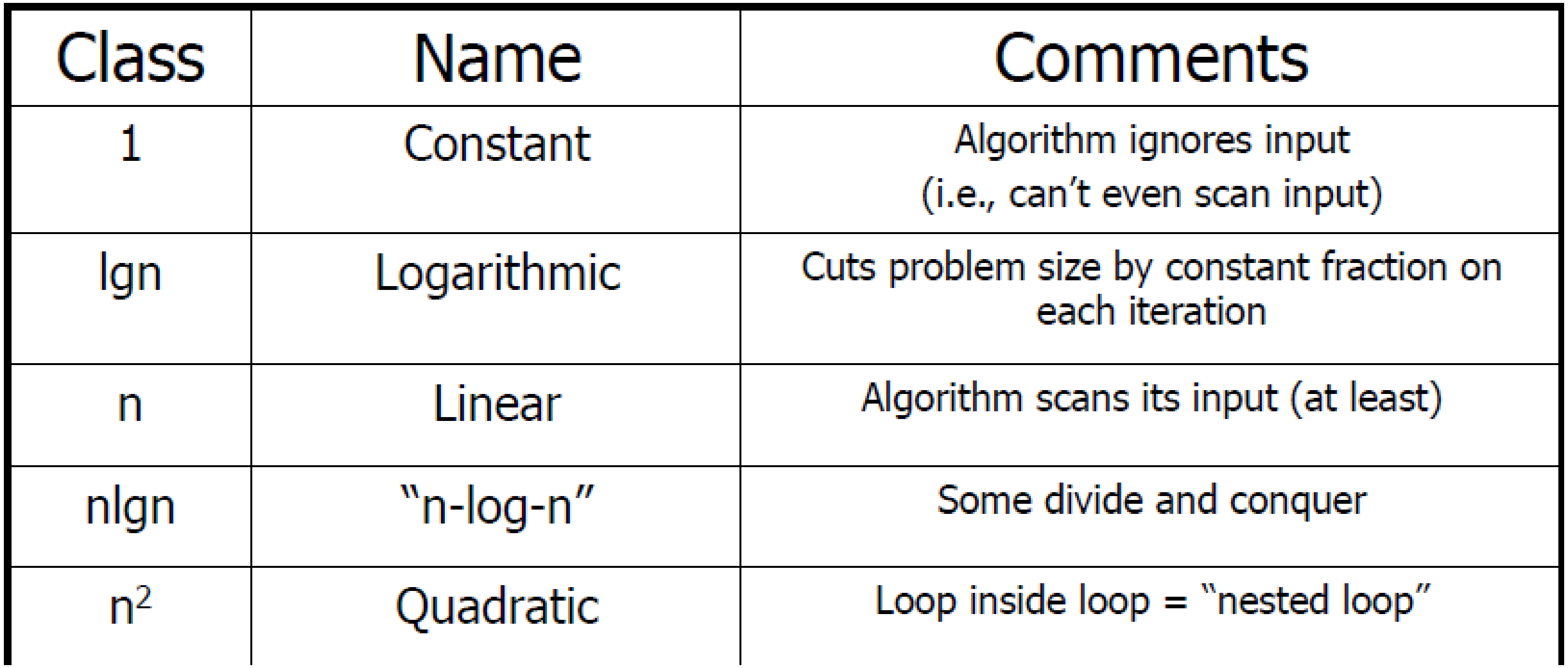
Class	Name	Comments
1	Constant	Algorithm ignores input (i.e., can't even scan input)
lgn	Logarithmic	Cuts problem size by constant fraction on each iteration
n	Linear	Algorithm scans its input (at least)

Basic Asymptotic Efficiency Classes



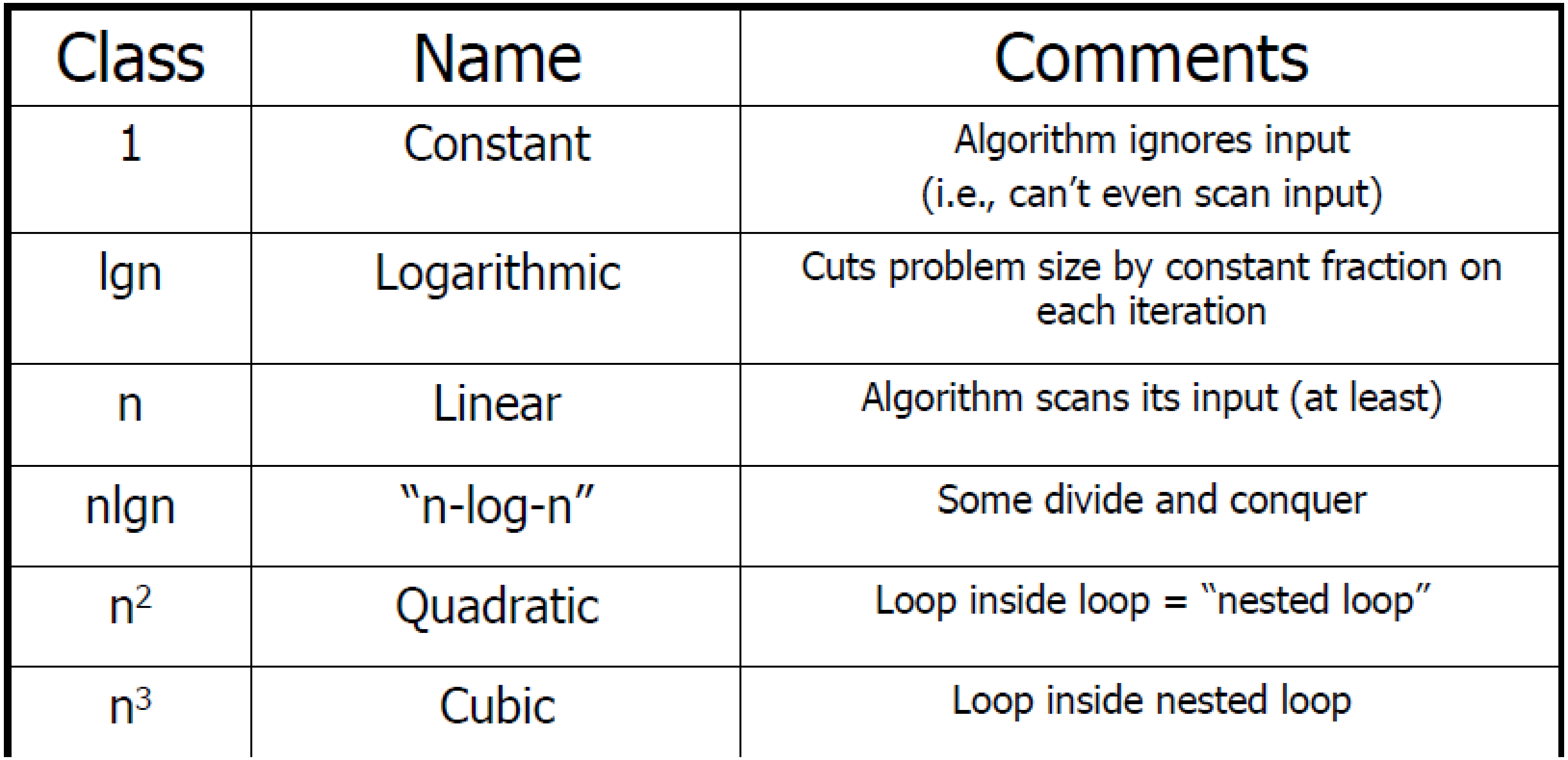
Class	Name	Comments
1	Constant	Algorithm ignores input (i.e., can't even scan input)
lgn	Logarithmic	Cuts problem size by constant fraction on each iteration
n	Linear	Algorithm scans its input (at least)
nlgn	"n-log-n"	Some divide and conquer

Basic Asymptotic Efficiency Classes



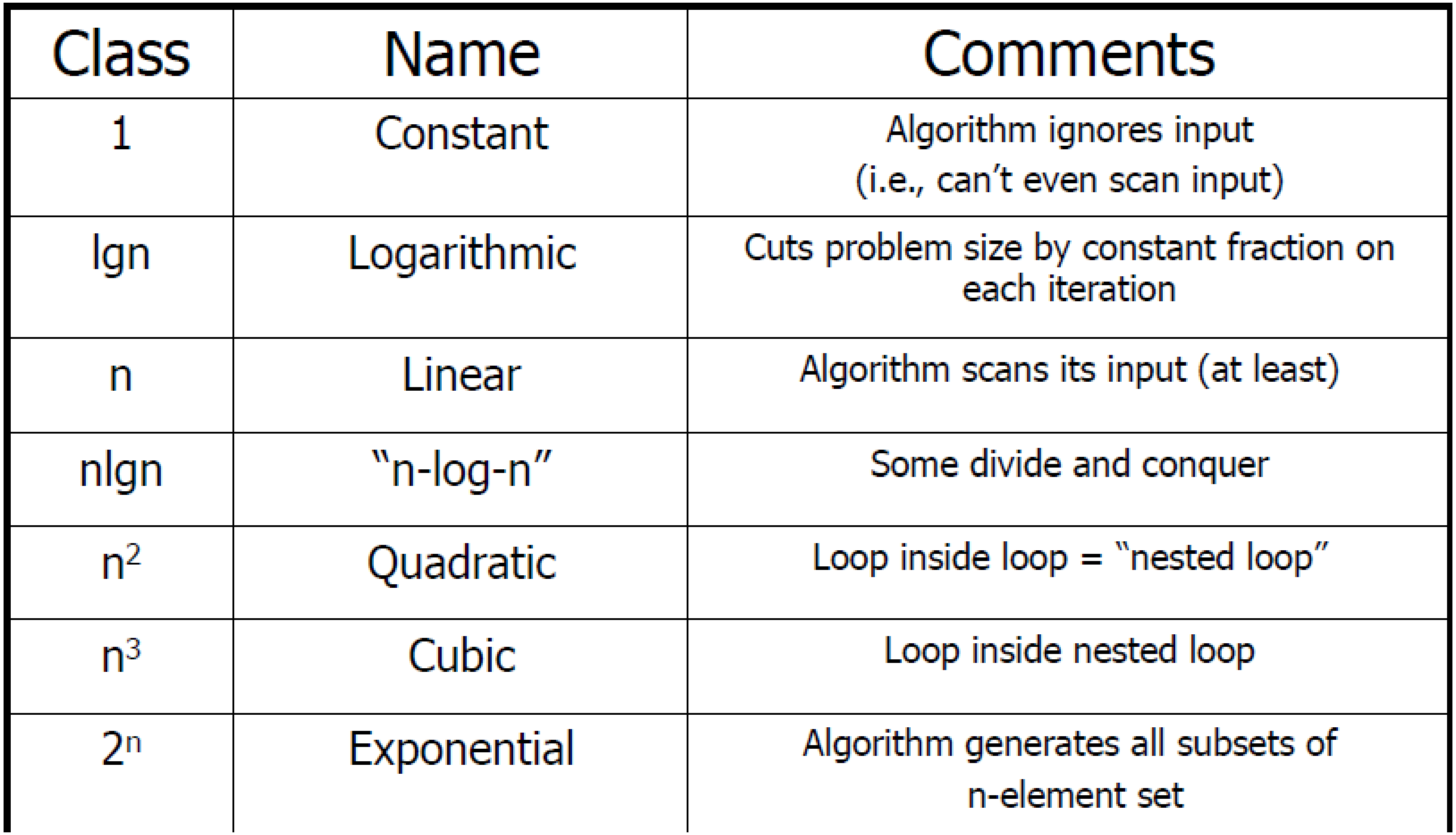
Class	Name	Comments
1	Constant	Algorithm ignores input (i.e., can't even scan input)
lgn	Logarithmic	Cuts problem size by constant fraction on each iteration
n	Linear	Algorithm scans its input (at least)
nlgn	"n-log-n"	Some divide and conquer
n^2	Quadratic	Loop inside loop = "nested loop"

Basic Asymptotic Efficiency Classes



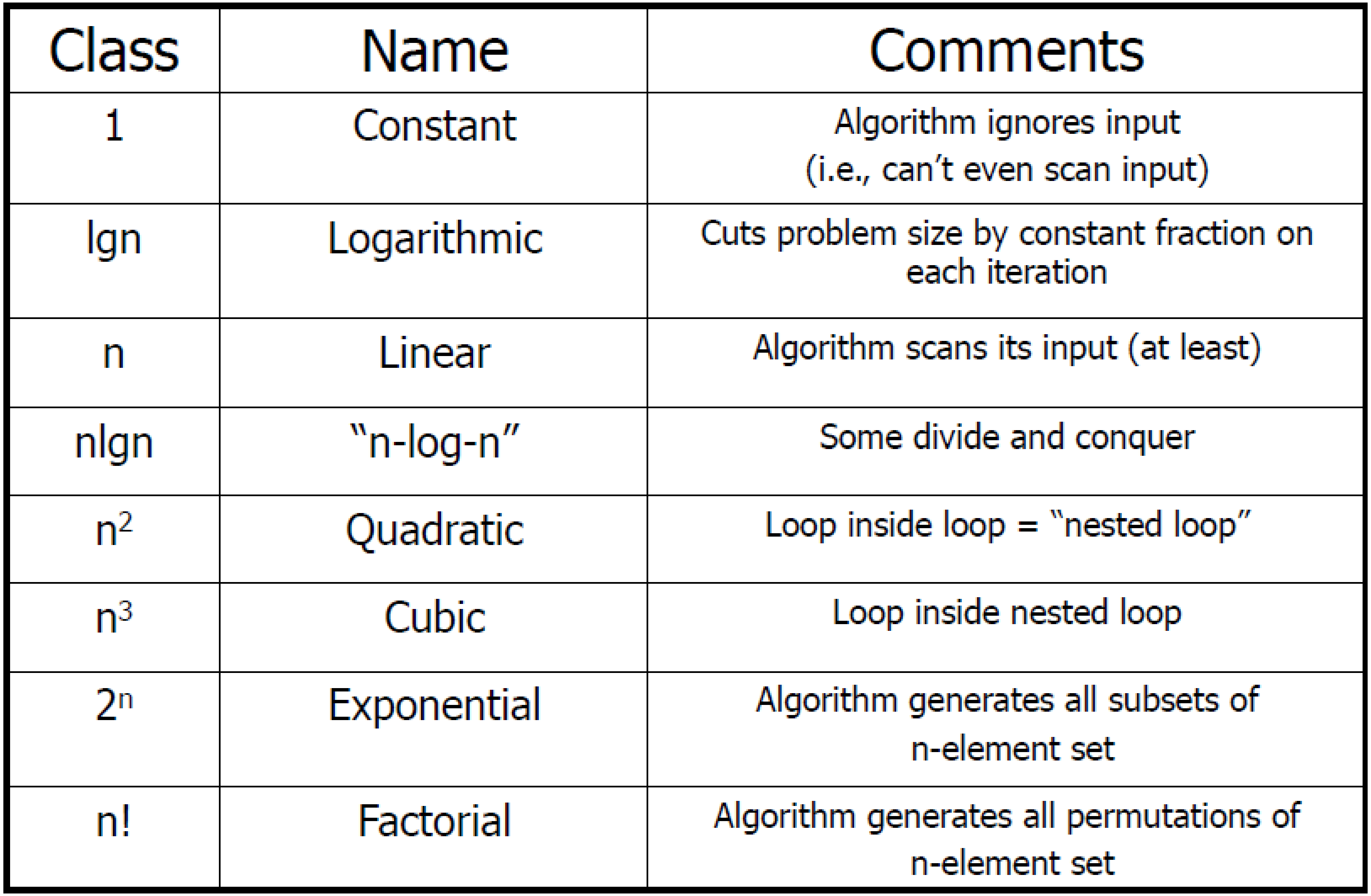
Class	Name	Comments
1	Constant	Algorithm ignores input (i.e., can't even scan input)
lgn	Logarithmic	Cuts problem size by constant fraction on each iteration
n	Linear	Algorithm scans its input (at least)
nlgn	"n-log-n"	Some divide and conquer
n^2	Quadratic	Loop inside loop = "nested loop"
n^3	Cubic	Loop inside nested loop

Basic Asymptotic Efficiency Classes



Class	Name	Comments
1	Constant	Algorithm ignores input (i.e., can't even scan input)
lgn	Logarithmic	Cuts problem size by constant fraction on each iteration
n	Linear	Algorithm scans its input (at least)
nlgn	"n-log-n"	Some divide and conquer
n^2	Quadratic	Loop inside loop = "nested loop"
n^3	Cubic	Loop inside nested loop
2^n	Exponential	Algorithm generates all subsets of n-element set

Basic Asymptotic Efficiency Classes



Class	Name	Comments
1	Constant	Algorithm ignores input (i.e., can't even scan input)
lgn	Logarithmic	Cuts problem size by constant fraction on each iteration
n	Linear	Algorithm scans its input (at least)
nlgn	"n-log-n"	Some divide and conquer
n^2	Quadratic	Loop inside loop = "nested loop"
n^3	Cubic	Loop inside nested loop
2^n	Exponential	Algorithm generates all subsets of n-element set
n!	Factorial	Algorithm generates all permutations of n-element set

ArrayList vs. LinkedList

	ArrayList	LinkedList
add(element)	O(1)	O(1)
remove(object)	O(n) + O(n)	O(n) + O(1)
get(index)	O(1)	O(n)
set(index, element)	O(1)	O(n) + O(1)
add(index, element)	O(1) + O(n)	O(n) + O(1)
remove(index)	O(n)	O(n) + O(1)
contains(object)	O(n)	O(n)
indexOf(object)	O(n)	O(n)

Recursion Complexity

Recursion

Divide et Impera – Divide and Conquer

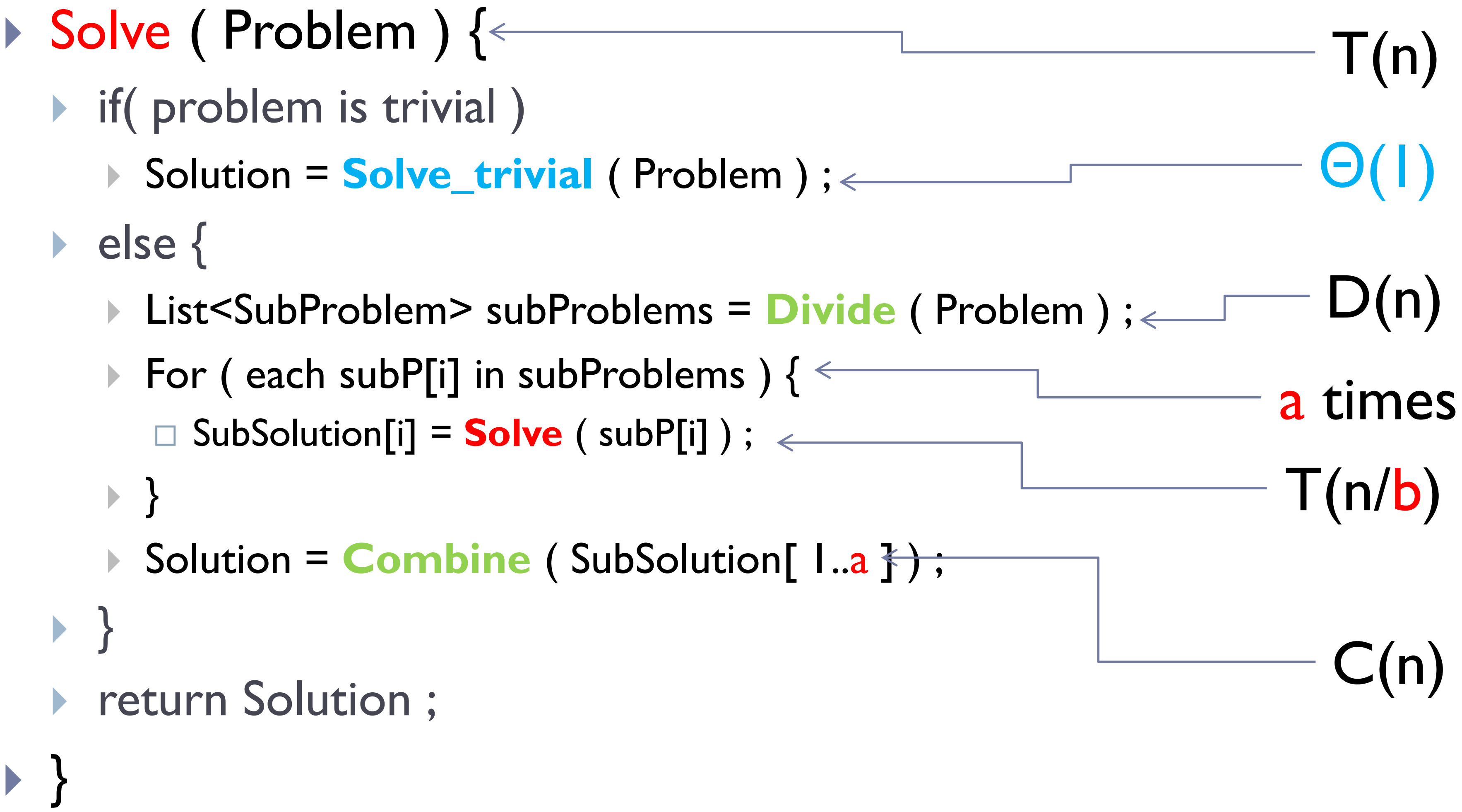
```
Solve ( Problem ) {
    if( problem is trivial )
        Solution = Solve_trivial ( Problem ) ;
    else {
        List<SubProblem> subProblems = Divide ( Problem ) ;
        For ( each subP[i] in subProblems ) {
            SubSolution[i] = Solve ( subP[i] ) ;
        }
        Solution = Combine ( SubSolution[ 1..N ] ) ;
    }
    return Solution ;
}
```

do recursion

What about complexity?

- a = number of sub-problems for a problem
- b = how smaller sub-problems are than the original one
- n = size of the original problem
- T(n) = complexity of **Solve**
  - …our unknown complexity function
- **Θ(1)** = complexity of **Solve_trivial**
  - …otherwise it wouldn't be trivial
- D(n) = complexity of **Divide**
- C(n) = complexity of **Combine**

Divide et Impera – Divide and Conquer



```
Solve ( Problem ) {                                                      $T(n)$
    if( problem is trivial )
        Solution = Solve_trivial ( Problem ) ;                           $\Theta(1)$
    else {
        List<SubProblem> subProblems = Divide ( Problem ) ;              $D(n)$
        For ( each subP[i] in subProblems ) {                            a times
            SubSolution[i] = Solve ( subP[i] ) ;                         $T(n/b)$
        }
        Solution = Combine ( SubSolution[ 1..a ] ) ;                     $C(n)$
    }
    return Solution ;
}
```

Complexity computation

- $T(n) =$
  - $\Theta(1)$ for $n \le c$
  - $D(n) + a\,T(n/b) + C(n)$ for $n > c$

- Recurrence Equation not easy to solve in the general case
- Special case:
  - If $D(n)+C(n)=\Theta(n)$
  - We obtain **$T(n) = \Theta(n \log n)$**.

Examples

Algorithm	Solve_trivial	Divide	a	b	Combine	Complexity
Dicotomic search	1	1	1	2	0	Log(n)
Merge Sort	1	1	2	2	n	n Log(n)
Permutation	0	1	n	n-1	0	Γ(n)
Combinations	0	1	k	n	0	k^n
Factorial	1	0	1	n-1	0	n
Fibonacci	1	0	2	n-1	0	2^n

Licenza d'uso

- Queste diapositive sono distribuite con licenza Creative Commons "Attribuzione - Non commerciale - Condividi allo stesso modo (CC BY-NC-SA)"
- Sei libero:
  - di riprodurre, distribuire, comunicare al pubblico, esporre in pubblico, rappresentare, eseguire e recitare quest'opera
  - di modificare quest'opera
- Alle seguenti condizioni:
  - **Attribuzione** — Devi attribuire la paternità dell'opera agli autori originali e in modo tale da non suggerire che essi avallino te o il modo i cui tu usi l'opera.
  - **Non commerciale** — Non puoi usare quest'opera per fini commerciali.
  - **Condividi allo stesso modo** — Se alteri o trasformi quest'opera, o se la usi per crearne un'altra, puoi distribuire l'opera risultante solo con ur licenza identica o equivalente a questa.
- http://creativecommons.org/licenses/by-nc-sa/3.0/