



Computational complexity

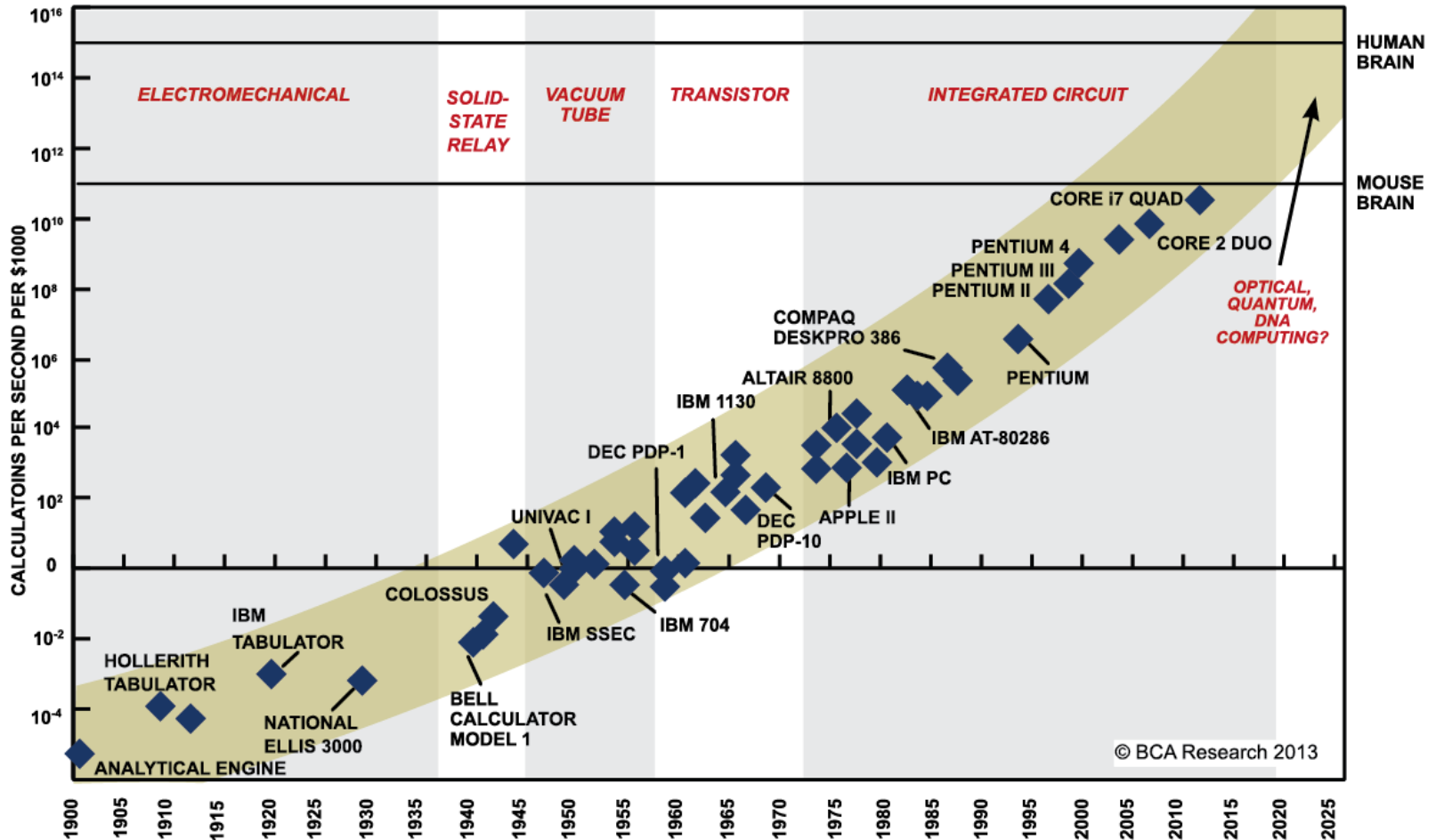
How to measure the difficulty of a problem

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**

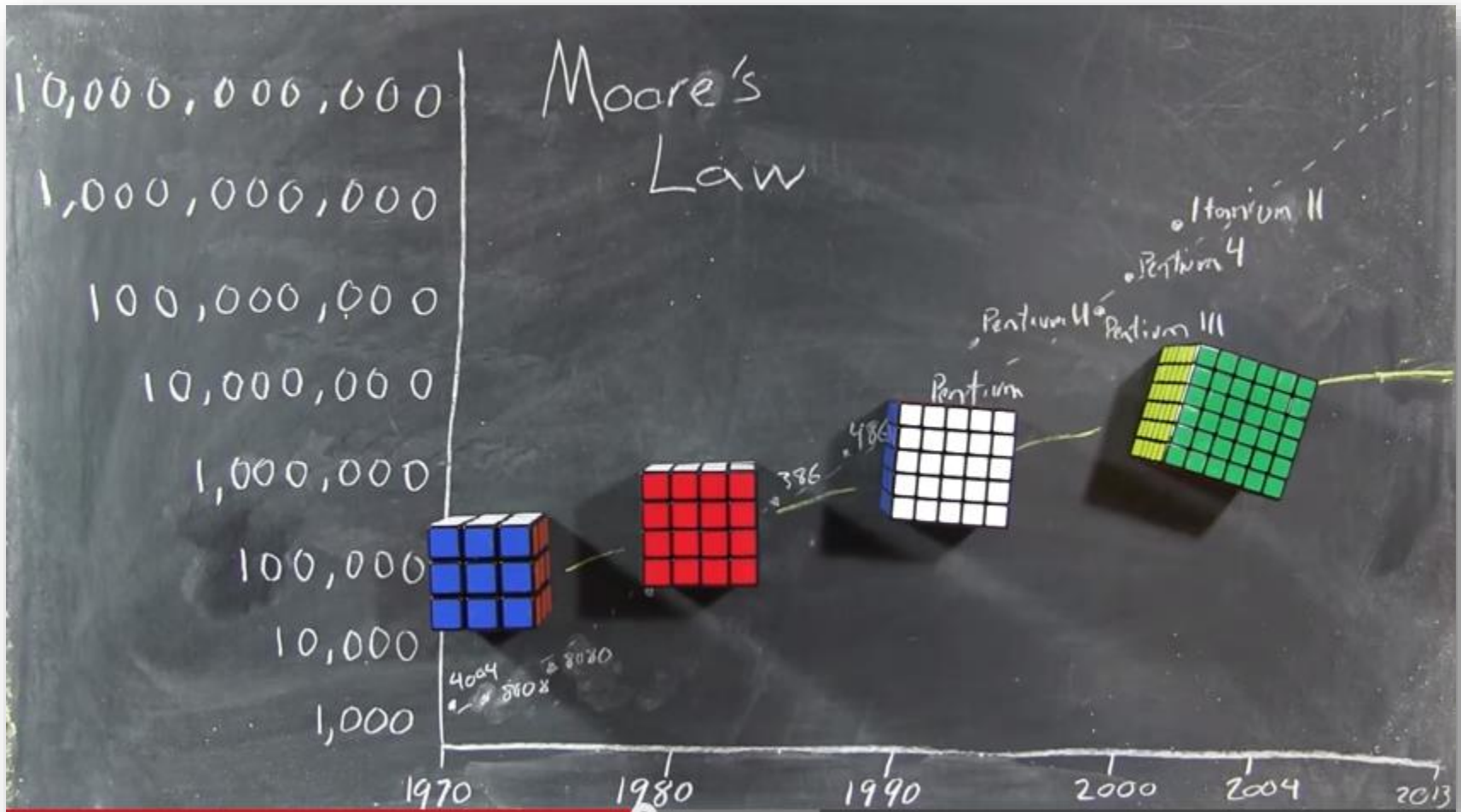


Moore's "Law"?



SOURCE: RAY KURZWEIL, "THE SINGULARITY IS NEAR: WHEN HUMANS TRANSCEND BIOLOGY", P.67, THE VIKING PRESS, 2006. DATAPPOINTS BETWEEN 2000 AND 2012 REPRESENT BCA ESTIMATES.

Moore's "Law"?



Sudoku

5	3		7			
6			1	9	5	
	9	8				6
8			6			
4			8	3		
7			2			
	6				2	8
			4	1	9	
			8			7

		F	L		P	J	S			H	Y		R			T	Q	O	
		X	M			T						J	P	U	B		C	S	
D	B	G	P		F	R				T		X			Q	Y	V	A	E
	I		N			L	A	G	O	C			T		Y	B		R	
				K	Q	I	M		S	F			O	V			L	W	
	V					S	G					B			I	L	Y	K	D
							Q			Y	U				E	A	B		W
	P			M	A	N	R	K			F	S		Q	G				
H		D	U	J	F	X		B	K		W						N	E	C
								P	R	M	T	D	C		L	U	I	J	
		N		K	H						P	M	C	O	R			G	Q
Q	B			V			X	I	J			S	K			M	A	T	
	U	D				W	C	L	G	K	A	Q	Y		H			P	
		X	I	A	S		N	H			O	U				B	F	C	
G	J		W	L	U	Q						V	R		E		I	X	
		M	N					I	D	Q	K		G	S	P	U	F		
B			H	P	D			F	Y	A	L	I			M				
	A			Y	C		J	U		G	F								
		I				N	W	O	V	B				T	S	D			
		C	V	R	L	T	P	N				O	A	M	I	Y	K		
	T		O	I			N	J	C	R			V					M	
Y	N	U			B				Q	X		W			P	C	O		
		W	M	U	C	V	B	P	I			H	F	D	K	Q			
	C		G			T	E			M		O	L			V	X		
K	X		V	R	J		F		H			Q	U		T	B			

Problems and Algorithms

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

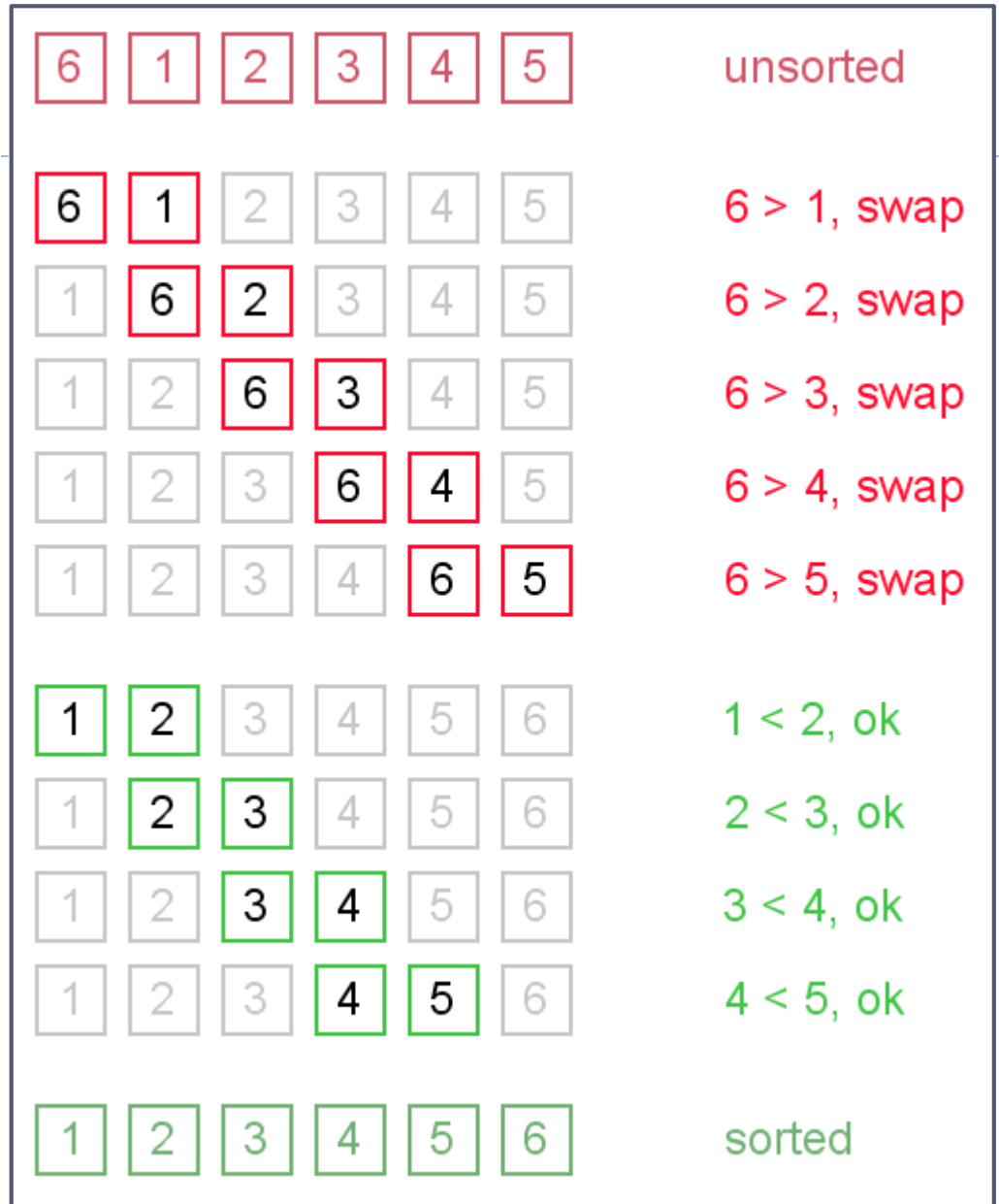


Analytical Approach

- ▶ An algorithm is a mapping
- ▶ 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



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



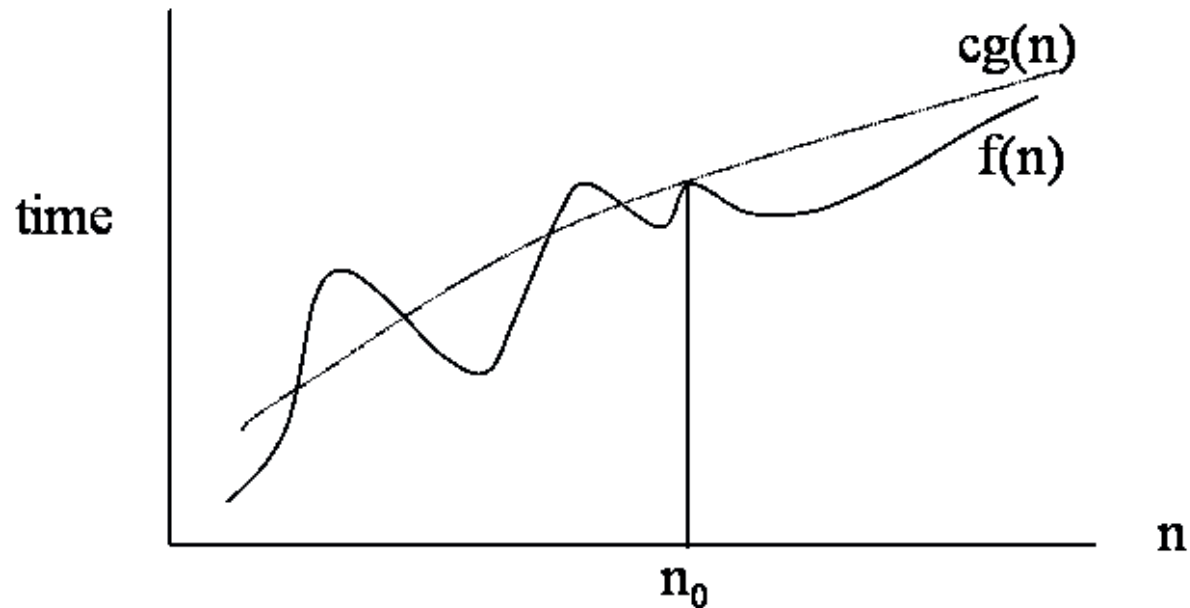
$O(\cdot)$

- ▶ Upper Bounding Running Time
- ▶ Why?
 - ▶ Little-oh
 - ▶ “Order of”
 - ▶ D’Oh



Upper Bounding Running Time

- ▶ $f(n)$ is $O(g(n))$ if f grows “at most as fast as” g



Big-O (formal)

- ▶ Let f and g be two functions such that

$$f(n): N \rightarrow R^+ \text{ and } g(n): N \rightarrow R^+$$

- ▶ if there exists positive constants c and n_0 such that

$$f(n) \leq cg(n), \text{ for all } n > n_0$$

- ▶ then we can write

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

Big-O (formal alt)

- ▶ Let f and g be two functions such that

$$f(n): N \rightarrow R^+ \text{ and } g(n): N \rightarrow R^+$$

- ▶ if there exists positive constants c and n_0 such that

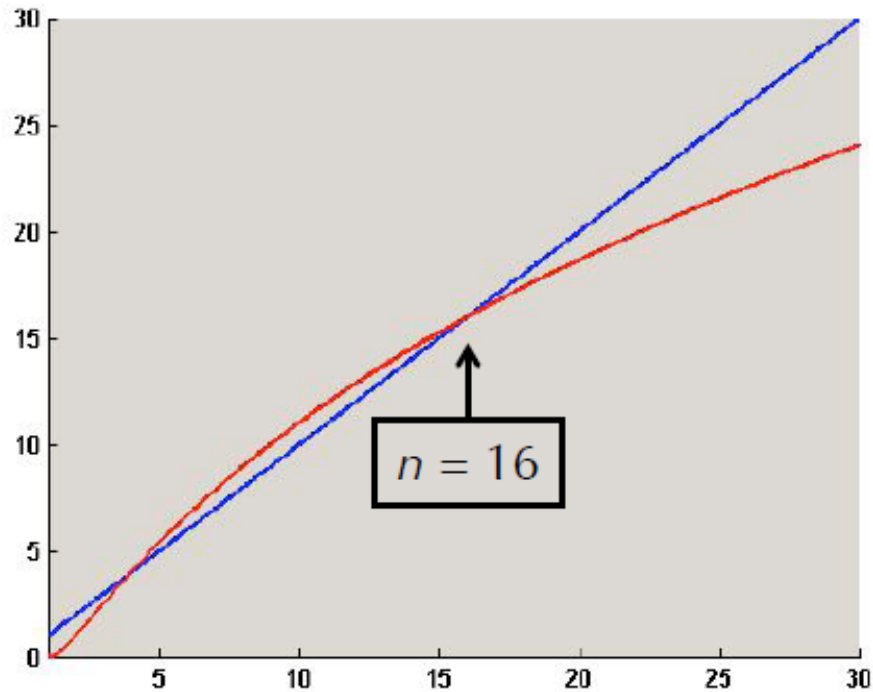
$$0 \leq \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c < \infty$$

- ▶ then we can write

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

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 $O(n)$

Notational Issues

- ▶ Big-O notation is a way of comparing functions
- ▶ Notation quite unconventional
 - ▶ e.g., $3x^3 + 5x^2 - 9 = O(x^3)$
- ▶ Doesn't mean
 - ▶ “ $3x^3 + 5x^2 - 9$ equals the function $O(x^3)$ ”
 - ▶ “ $3x^3 + 5x^2 - 9$ is big oh of x^3 ”
- ▶ But
 - ▶ “ $3x^3 + 5x^2 - 9$ is dominated by x^3 ”

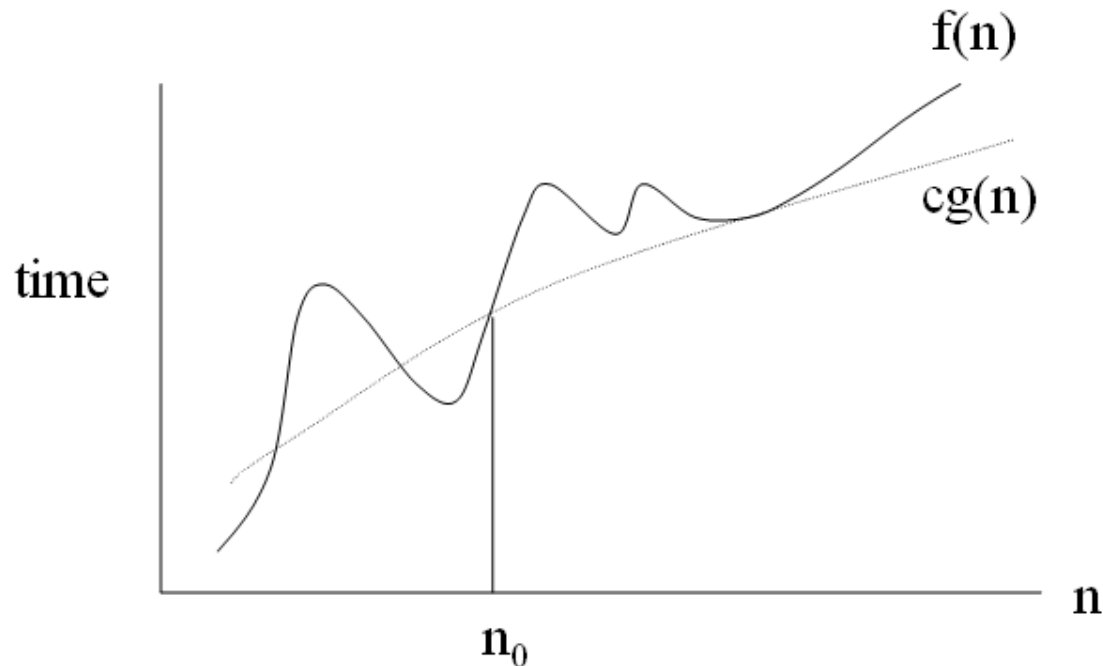
Common Misunderstanding

- ▶ $3x^3 + 5x^2 - 9 = O(x^3)$
- ▶ 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 $O(x^{2.5})$ ”



Lower Bounding Running Time

- ▶ $f(n)$ is $\Omega(g(n))$ if f grows “at least as fast as” g



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

Big-Omega (formal)

- ▶ Let f and g be two functions such that

$$f(n): N \rightarrow R^+ \text{ and } g(n): N \rightarrow R^+$$

- ▶ if there exists positive constants c and n_0 such that

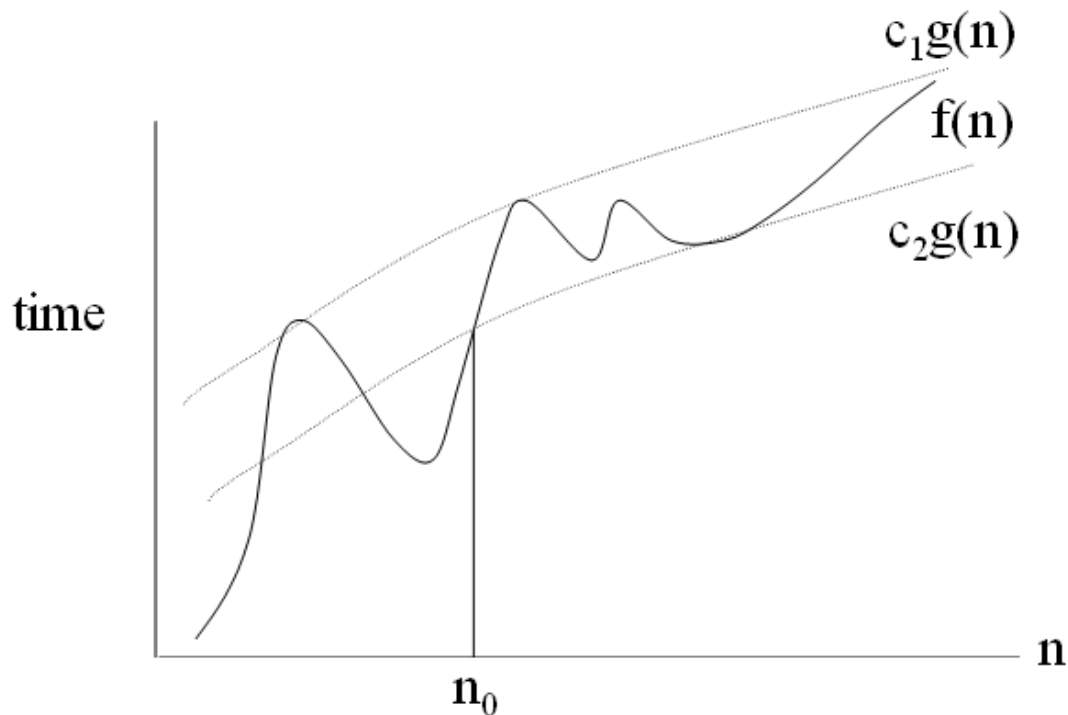
$$f(n) \geq cg(n), \text{ for all } n > n_0$$

- ▶ then we can write

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

Tightly Bounding Running Time

- ▶ $f(n)$ is $\Theta(g(n))$ if f is essentially the same as g , to within a constant multiple



Big-Theta (formal)

- ▶ Let f and g be two functions such that

$$f(n): N \rightarrow R^+ \text{ and } g(n): N \rightarrow R^+$$

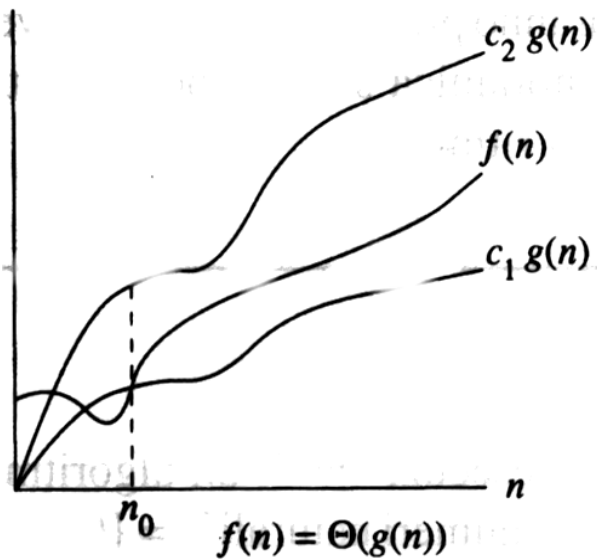
- ▶ if there exists positive constants c_1 , c_2 and n_0 such that

$$c_1g(n) \leq f(n) \leq c_2g(n), \text{ for all } n > n_0$$

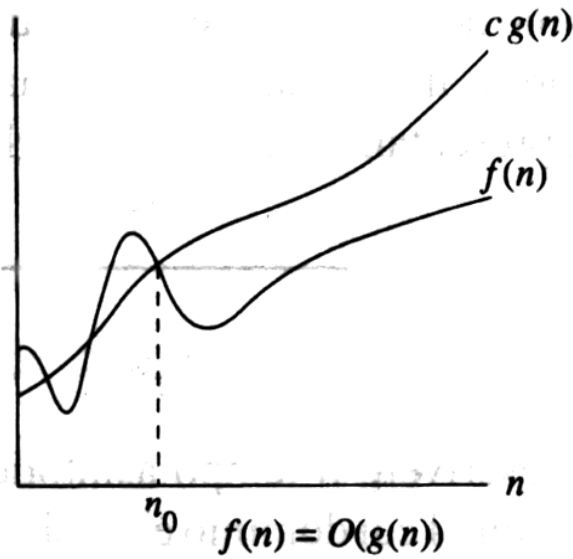
- ▶ then we can write

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

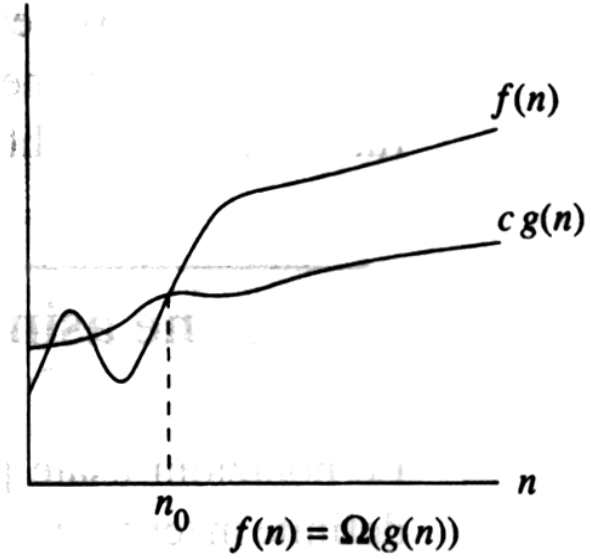
Big- Θ , Big- O , and Big- Ω



(a)



(b)



(c)

Big- Ω and Big- Θ

- ▶ Big- Ω : reverse of big- O . I.e.

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

iff

$$g(x) = O(f(x))$$

- ▶ so $f(x)$ asymptotically dominates $g(x)$

Big- Ω and Big- Θ

- ▶ Big- Θ : domination in both directions. I.e.

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

iff

$$f(x) = O(g(x)) \ \&\& \ f(x) = \Omega(g(x))$$

Problem

- ▶ Order the following from smallest to largest asymptotically. Group together all functions which are big- Θ of each other:

$$x + \sin x, \ln x, x + \sqrt{x}, \frac{1}{x}, 13 + \frac{1}{x}, 13 + x, e^x, x^e, x^x$$

$$(x + \sin x)(x^{20} - 102), x \ln x, x(\ln x)^2, \lg_2 x$$

Solution

$$1/x$$

$$13+1/x$$

$$\ln x \lg_2 x$$

$$x + \sin x, \cos x$$

$$x \ln x$$

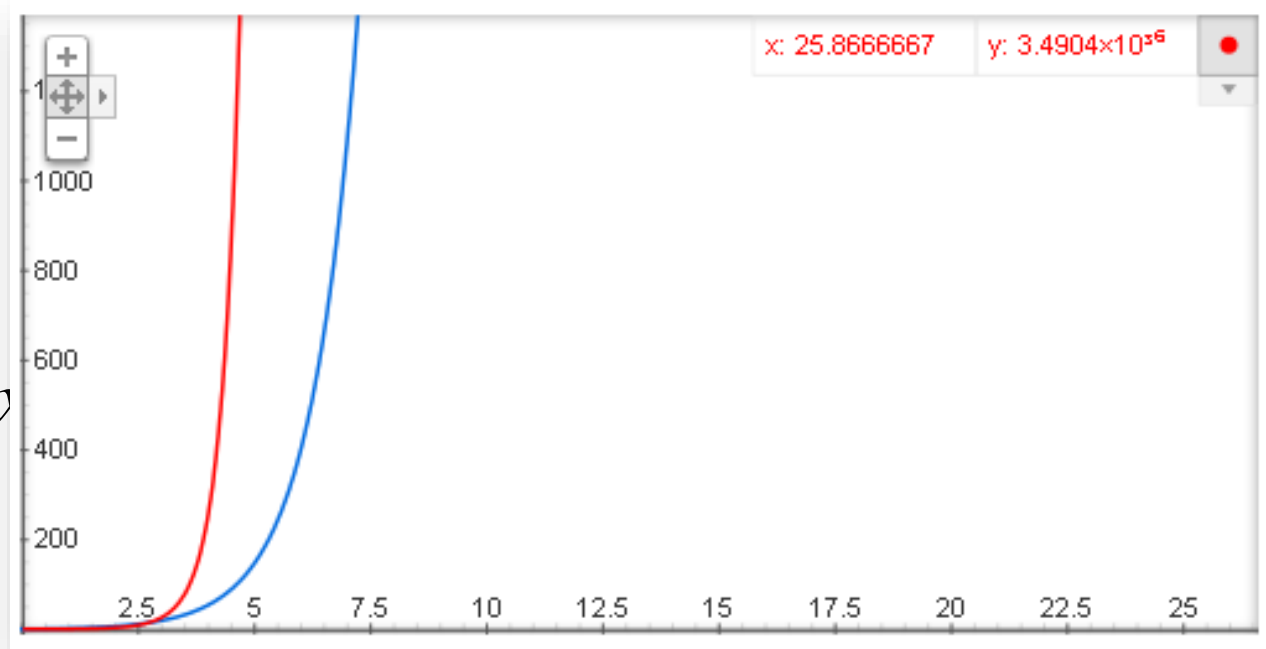
$$x(\ln x)^2$$

$$x^e$$

$$(x + \sin x)(x^{20} - 102)$$

$$e^x$$

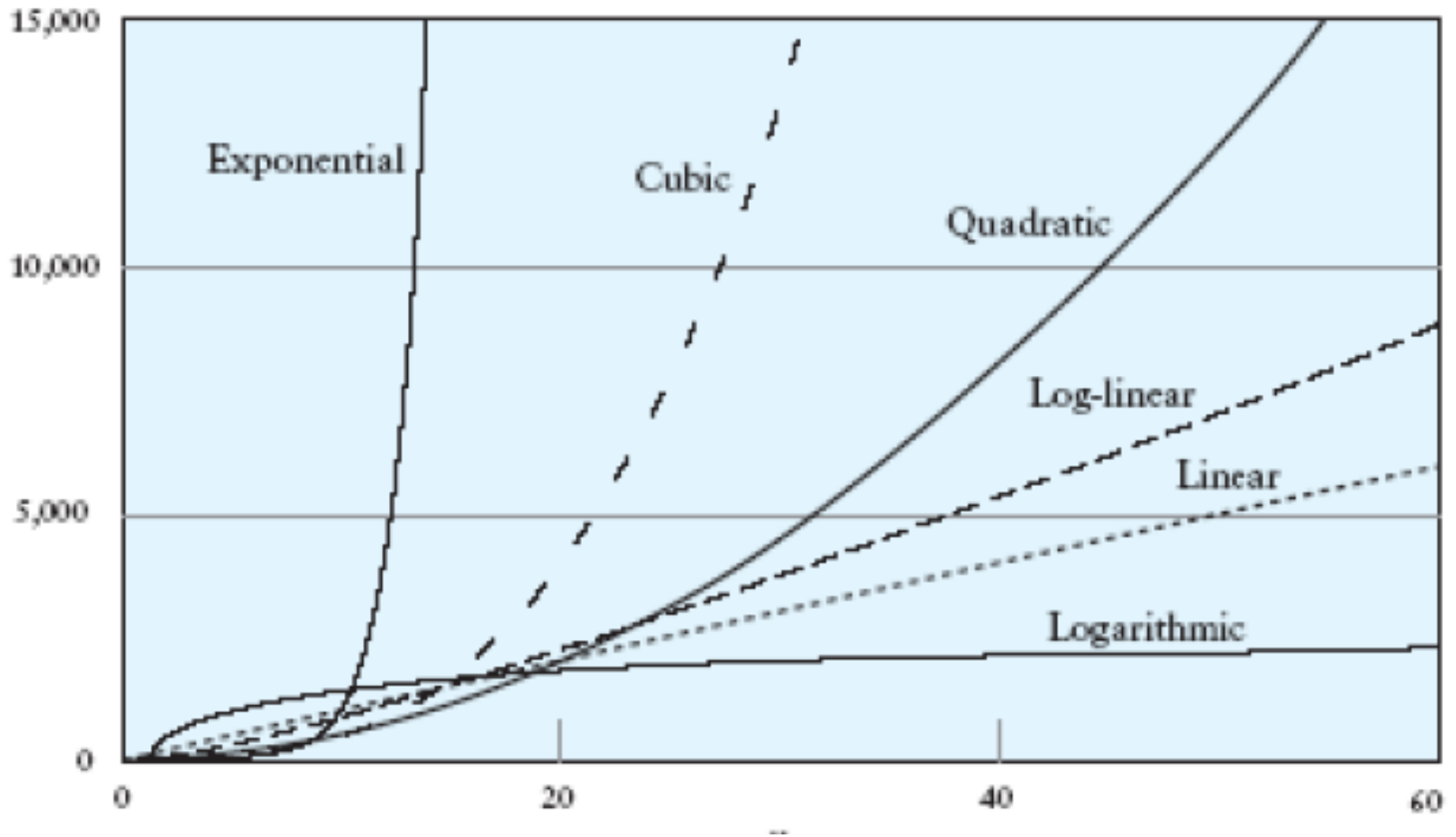
$$x^x$$



Practical approach



Practical approach



Class	Complexity	Number of Operations and Execution Time (1 instr/ μ sec)					
		n		10^2		10^3	
constant	$O(1)$	1	1 μ sec	1	1 μ sec	1	1 μ sec
logarithmic	$O(\lg n)$	3.32	3 μ sec	6.64	7 μ sec	9.97	10 μ sec
linear	$O(n)$	10	10 μ sec	10^2	100 μ sec	10^3	1 msec
$O(n \lg n)$	$O(n \lg n)$	33.2	33 μ sec	664	664 μ sec	9970	10 msec
quadratic	$O(n^2)$	10^2	100 μ sec	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}	
		10^4		10^5		10^6	
constant	$O(1)$	1	1 μ sec	1	1 μ sec	1	1 μ sec
logarithmic	$O(\lg n)$	13.3	13 μ sec	16.6	7 μ sec	19.93	20 μ sec
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-Oh , the actual $T(n)$ of an algorithm is not calculated
 - ▶ $T(n)$ may be determined empirically
- ▶ Big-Oh 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.,
 - ▶ $f(n) = O(n^2)$
 - ▶ $g(n) = O(n^3)$
 - ▶ $f(n) + g(n) = O(n^3)$

Rule #5 – the “product” rule

- ▶ If two processes are constructed such that second process is repeated a number of times for each n in the first process, then O is equal to the product of the orders of magnitude for both products
- ▶ 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

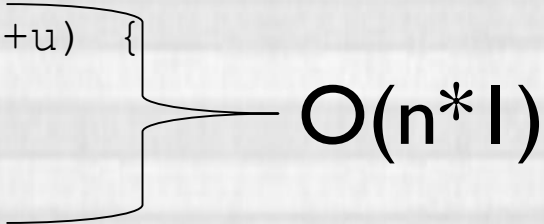
```
for(int t=0; t<n; ++t) {  
    for(int u=0; u<n; ++u) {  
        ++zap;  
    }  
}
```

$O(n)$

$O(1)$

Nested Loops

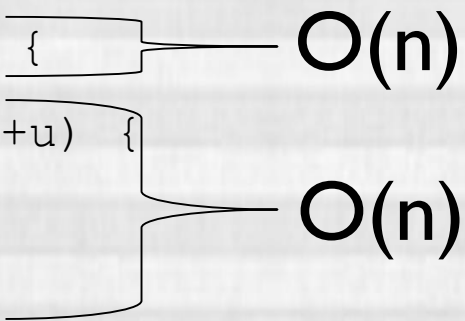
```
for(int t=0; t<n; ++t) {  
    for(int u=0; u<n; ++u) {  
        ++zap;  
    }  
}
```



$O(n \cdot I)$

Nested Loops

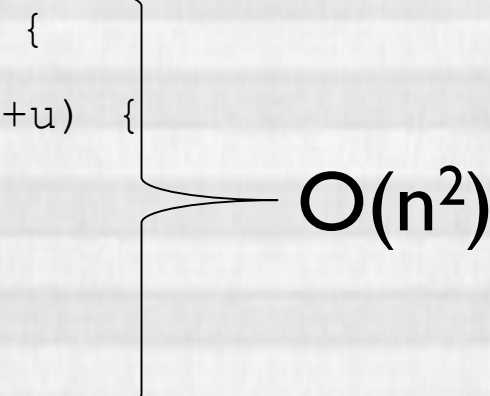
```
for(int t=0; t<n; ++t) {  
    for(int u=0; u<n; ++u) {  
        ++zap;  
    }  
}
```



The diagram illustrates the complexity of the nested loops. A bracket on the right side of the outer loop is labeled $O(n)$. A larger bracket on the right side of the inner loop and its body is labeled $O(n)$.

Nested Loops

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

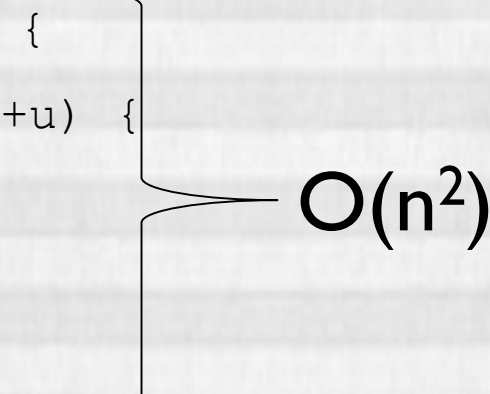


$O(n^2)$

Nested Loops

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

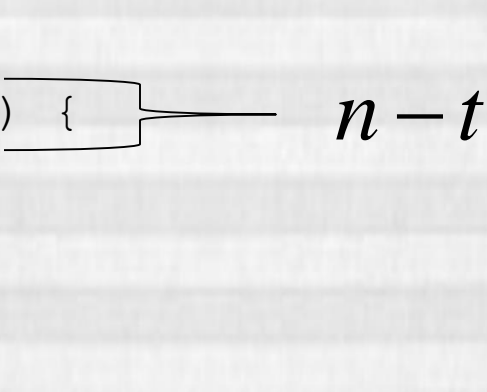
```
for(int t=0; t<n; ++t) {  
    for(int u=0; u<n; ++u) {  
        ++zap;  
    }  
}
```



$O(n^2)$

More Nested Loops

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



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

Sequential statements

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

$O(n^2)$

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

Conditionals

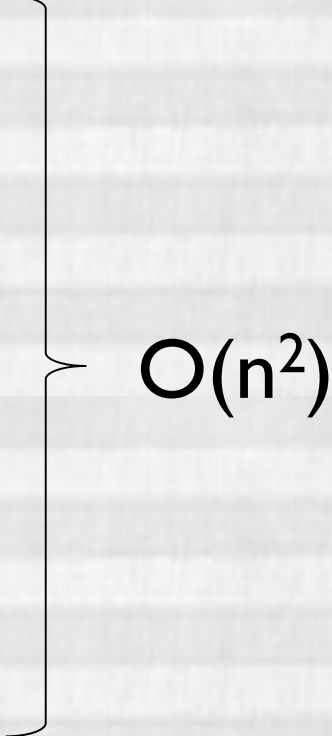
```
for(int t=0; t<n; ++t) {  
    if(t%2) {  
        for(int u=t; u<n; ++u) {  
            ++zap;  
        }  
    } else {  
        zap = 0;  
    }  
}
```

$O(n)$

$O(1)$

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

Caveats

- ▶ Real time vs. complexity



Caveats

- ▶ Real time vs. complexity
- ▶ CPU time vs. RAM vs. disk

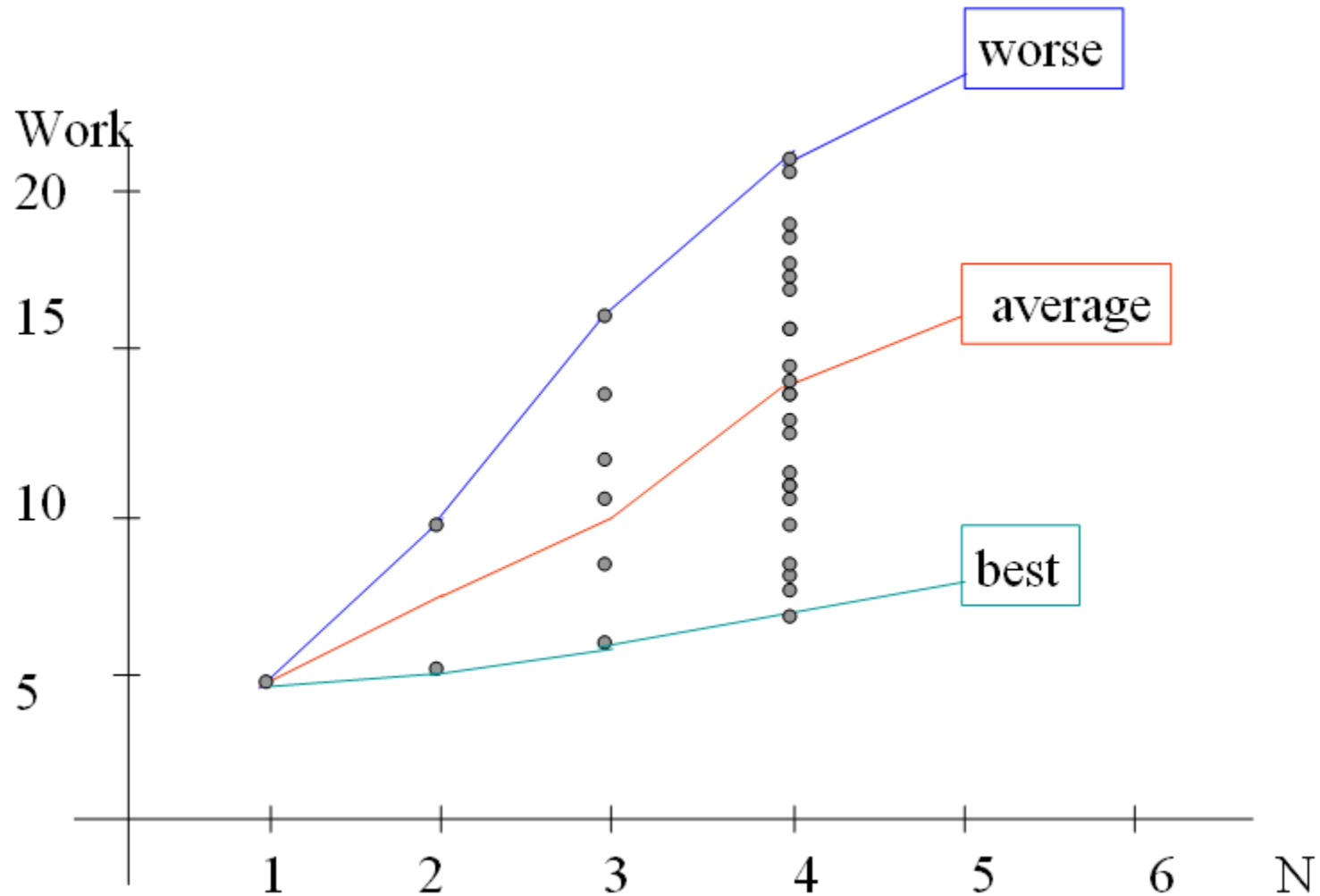


Caveats

- ▶ Real time vs. complexity
- ▶ CPU time vs. RAM vs. disk
- ▶ Worse, Average or Best Case?

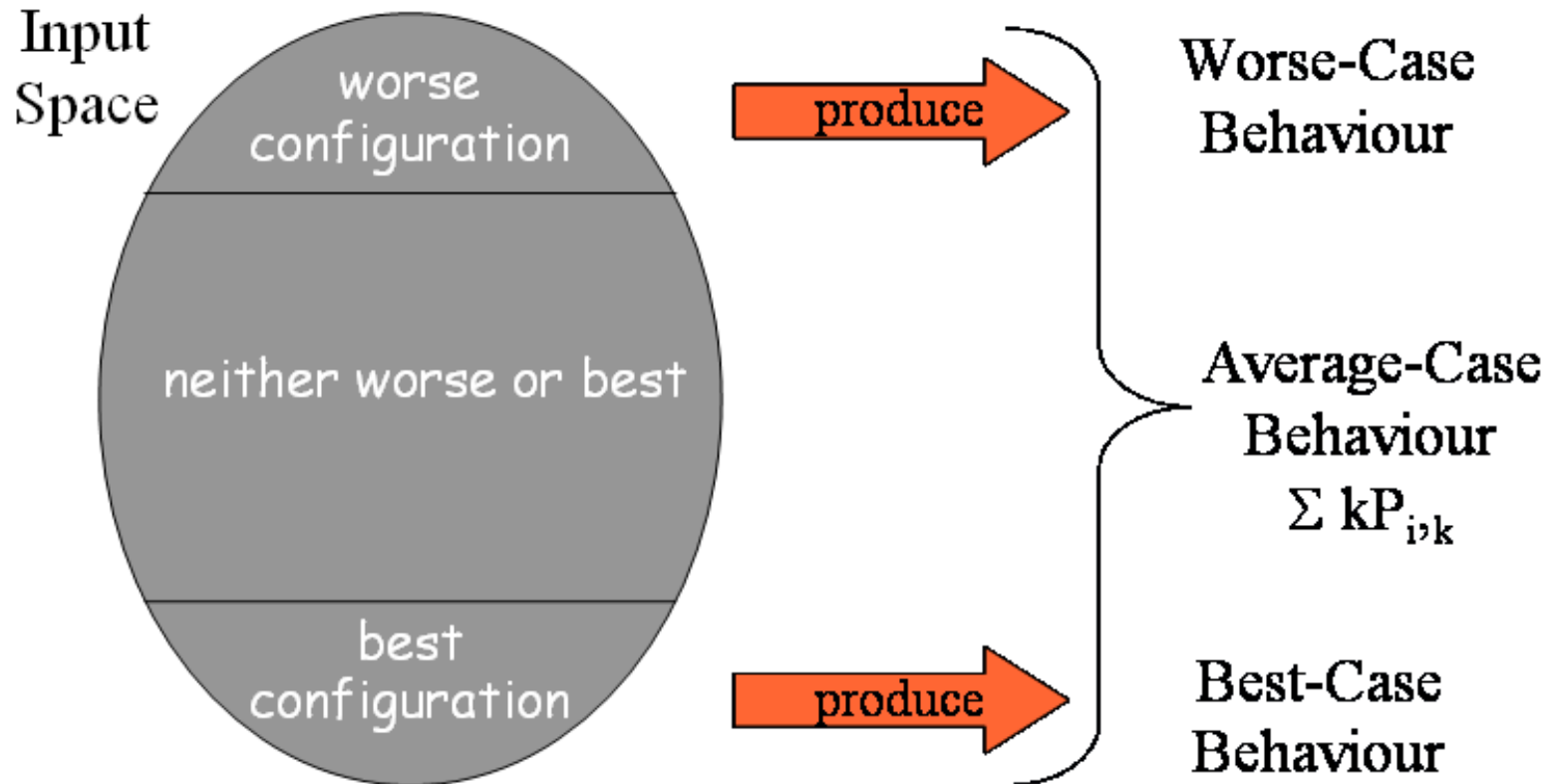


Worse, Average or Best Case?



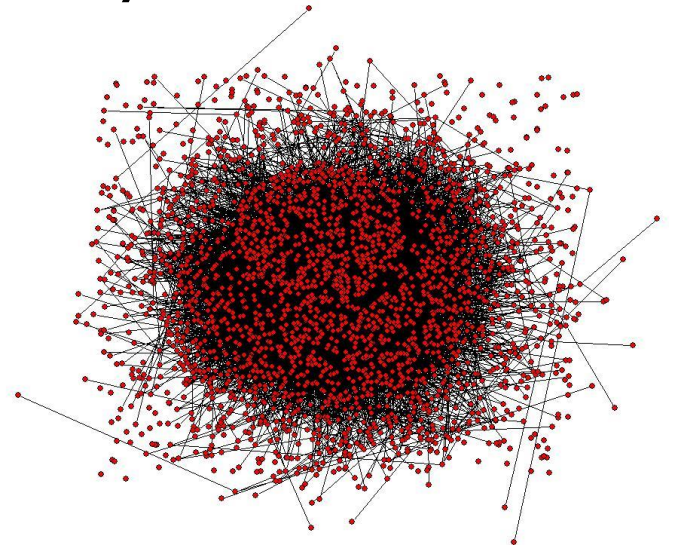
Worse, Average or Best Case?

- ▶ Depends on input problem instance type



Computational Complexity Theory

- ▶ In computer science, computational complexity theory is the branch of the theory of computation that studies the resources, or cost, of the computation required to solve a given computational problem
- ▶ Complexity theory analyzes the difficulty of computational problems in terms of many different computational resources



Note

Solve a problem

vs.

Verify a solution

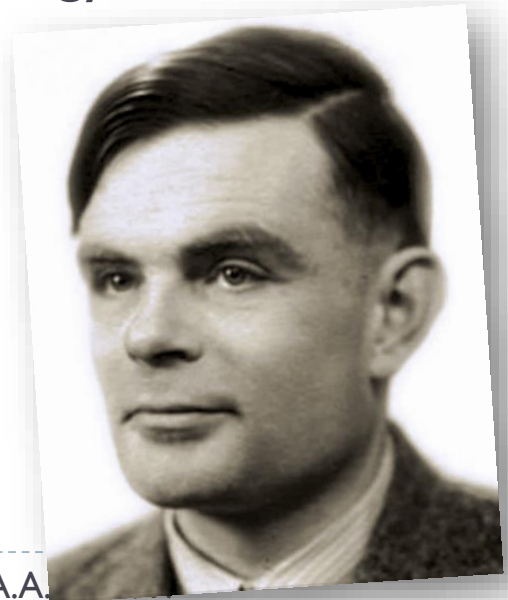
- ▶ E.g.,
 - ▶ Sort
 - ▶ Shortest path

Complexity Classes

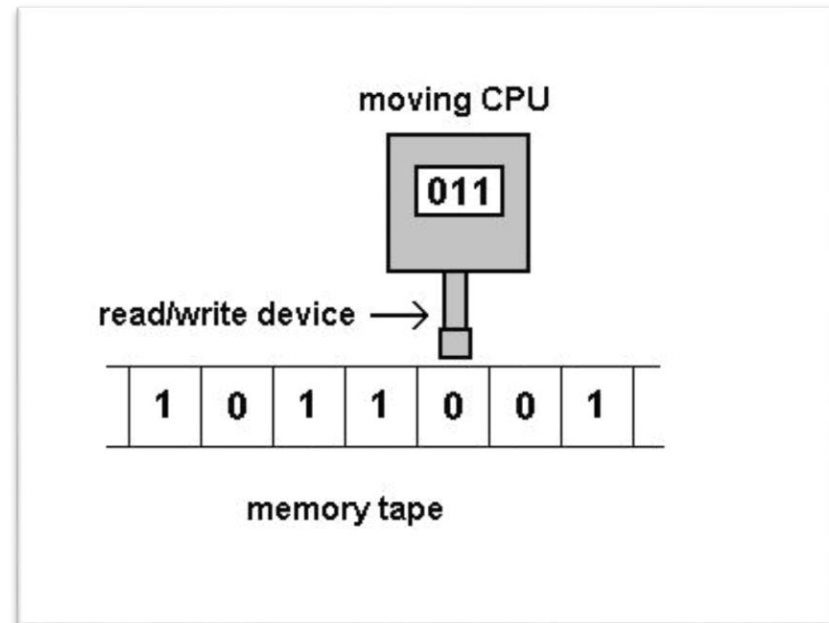
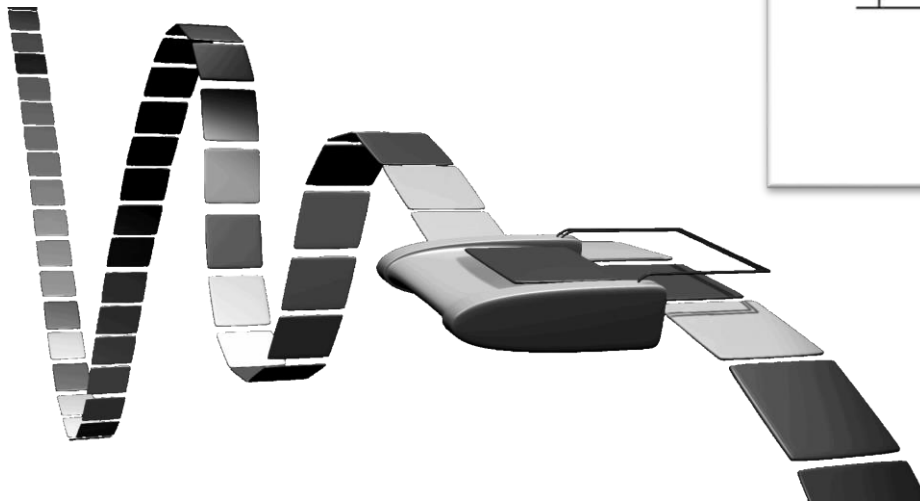
- ▶ A complexity class is the set of all of the computational problems which can be solved using a certain amount of a certain computational resource

Deterministic Turing Machine

- ▶ Deterministic or Turing machines are extremely basic symbol-manipulating devices which — despite their simplicity — can be adapted to simulate the logic of any computer that could possibly be constructed
- ▶ Described in 1936 by Alan Turing.
 - ▶ Not meant to be a practical computing technology
 - ▶ Technically feasible
 - ▶ A thought experiment about the limits of mechanical computation

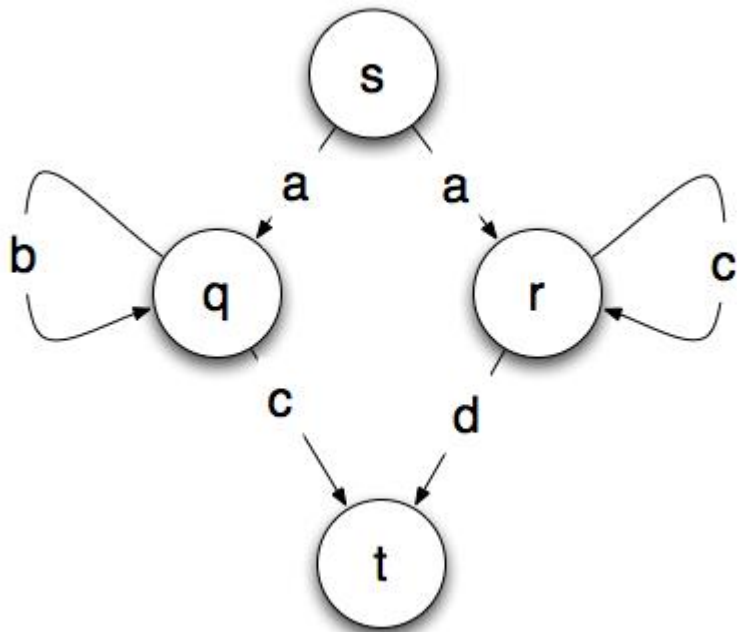


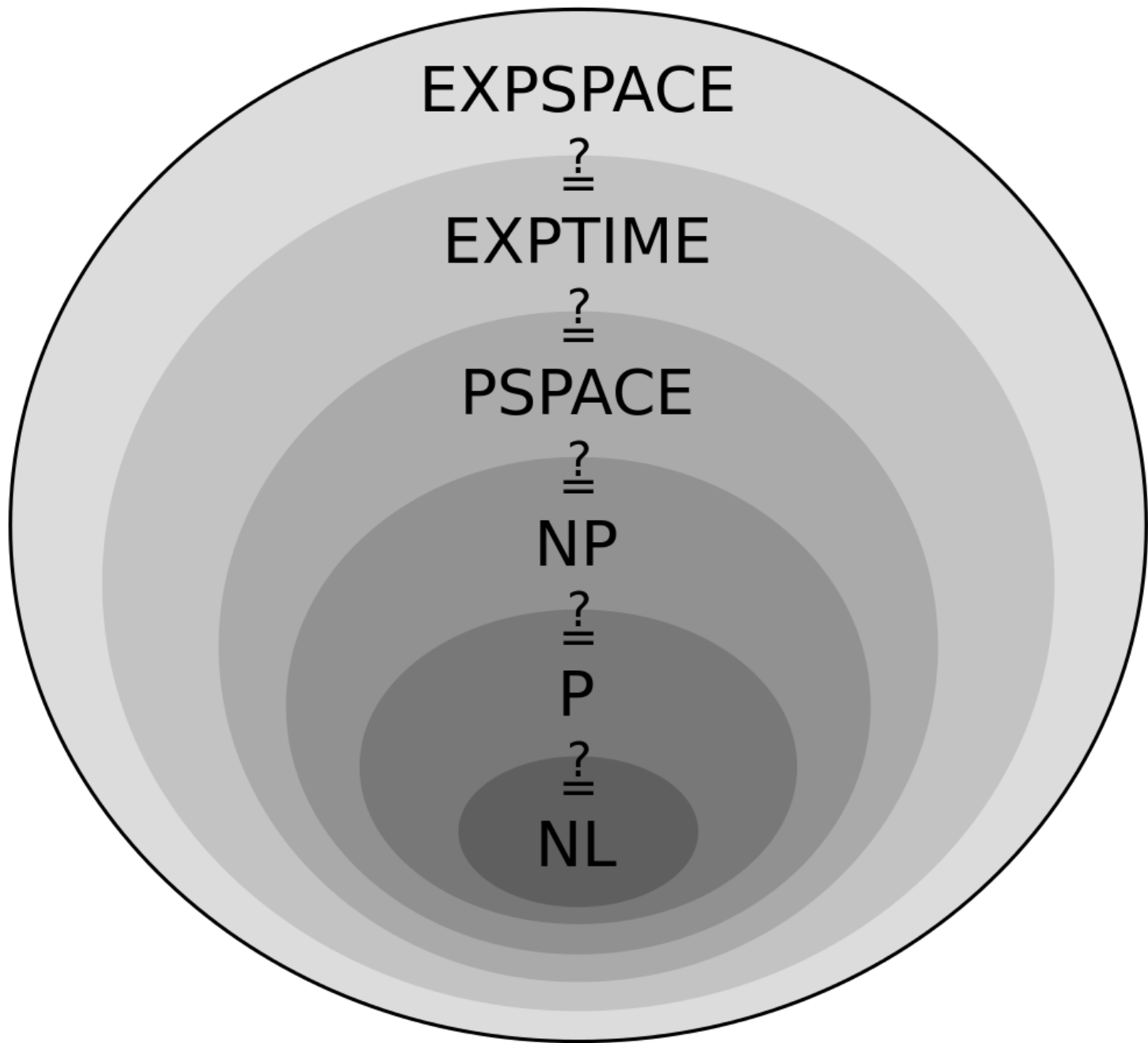
Deterministic Turing Machine



Non-Deterministic Turing Machine

- ▶ Turing machine whose control mechanism works like a non-deterministic finite automaton





Class	Resource	Model	Constraint
DTIME($f(n)$)	Time	DTM	$f(n)$
P	Time	DTM	$O(n^k)$
EXPTIME	Time	DTM	$O(2^{n^k})$
NTIME	Time	NDTM	$f(n)$
NP	Time	NDTM	$O(n^k)$
NEXPTIME	Time	NDTM	$O(2^{n^k})$
DSPACE($f(n)$)	Space	DTM	$f(n)$
L	Space	DTM	$O(\log(n))$
PSPACE	Space	DTM	$O(n^k)$
EXPSPACE	Space	DTM	$O(2^{n^k})$
NSPACE($f(n)$)	Space	NDTM	$f(n)$
NL	Space	NDTM	$O(\log(n))$
NPSPACE	Space	NDTM	$O(n^k)$
NEXPSPACE	Space	NDTM	$O(2^{n^k})$

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)
n lgn	"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)
$\lg n$	Logarithmic	Cuts problem size by constant fraction on each iteration
n	Linear	Algorithm scans its input (at least)
$n \lg n$	"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)
$\lg n$	Logarithmic	Cuts problem size by constant fraction on each iteration
n	Linear	Algorithm scans its input (at least)
$n \lg n$	"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)
$\lg n$	Logarithmic	Cuts problem size by constant fraction on each iteration
n	Linear	Algorithm scans its input (at least)
$n \lg n$	"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)
$\lg n$	Logarithmic	Cuts problem size by constant fraction on each iteration
n	Linear	Algorithm scans its input (at least)
$n \lg n$	"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)	$O(n)$	$O(n)$

*In theory, there is no difference
between theory and practice.*



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 in 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 una licenza identica o equivalente a questa. 
- ▶ <http://creativecommons.org/licenses/by-nc-sa/3.0/>