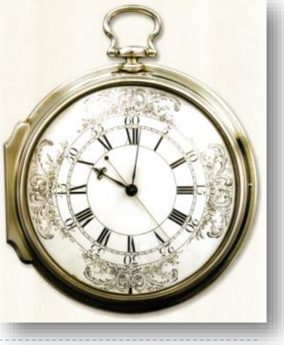
Computational complexity

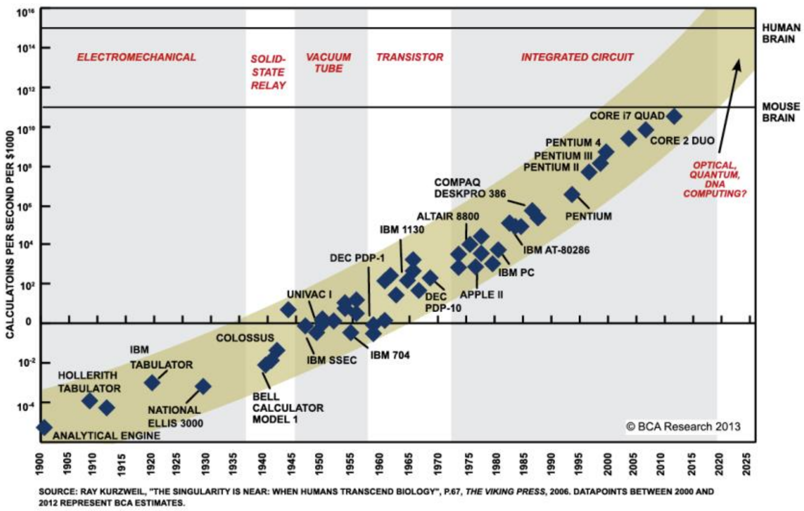
How to measure the difficulty of a problem

How to Measure Efficiency?

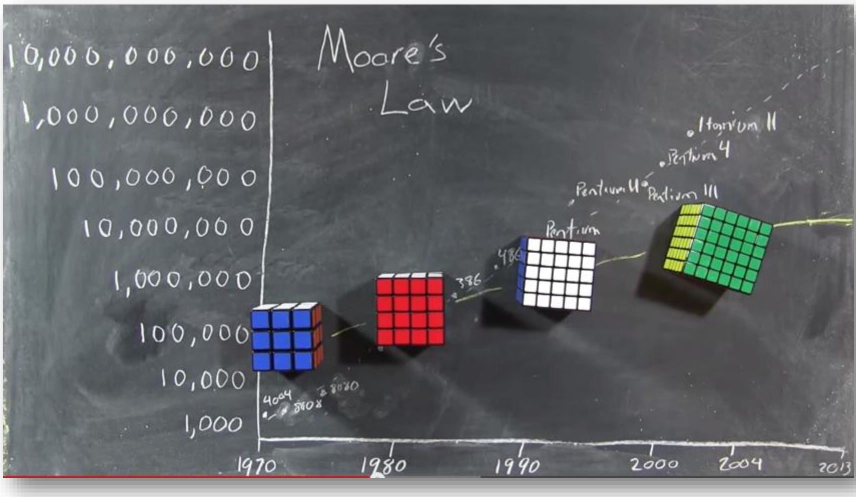
- Critical resources
  - programmer's effort
  - time, space (disk, RAM)
- Analysis
  - empirical (run programs)
  - analytical (asymptotic algorithm analysis)
- Worst case vs. Average case



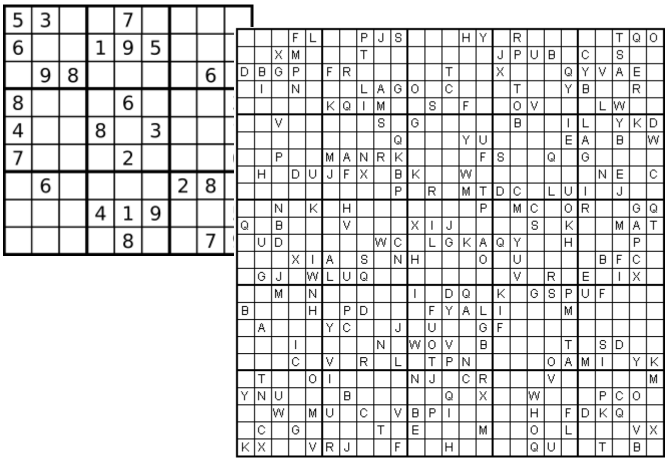
Moore’s “Law”?



Moore’s “Law”?



Sudoku



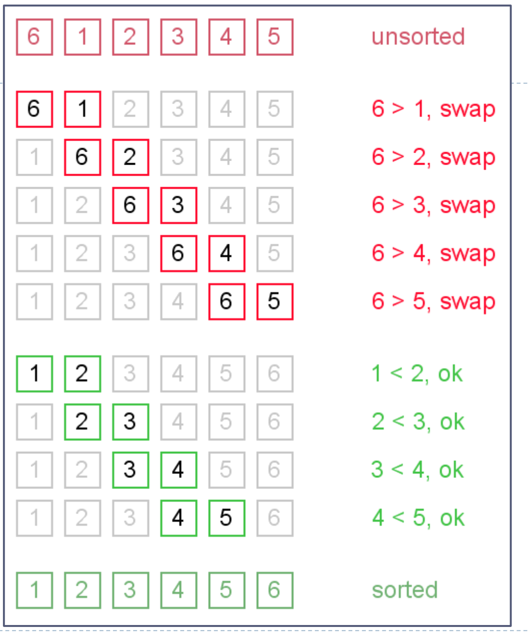
Problems and Algorithms

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

Analytical Approach

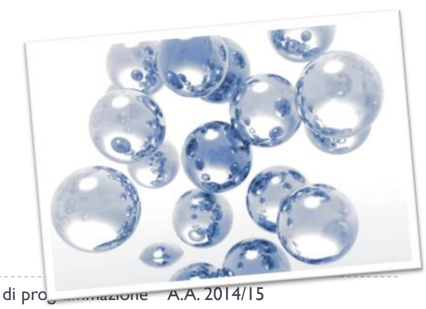
- An algorithm is a mapping
- For most algorithms, running time depends on "size" of the input
- Running time is expressed as T(n)
  - some function T
  - input *size* n

Bubble sort



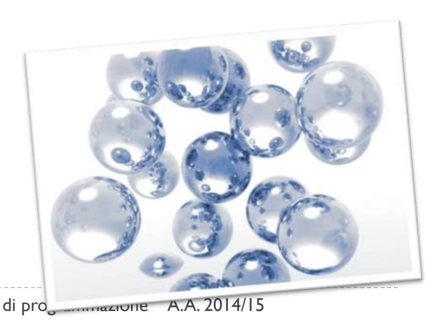
Analysis

- 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



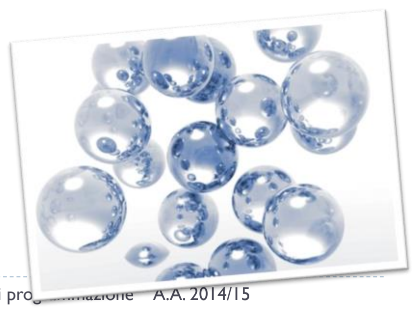
Analysis

- 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: $T(n) \approx n^2$



Intuitive motivations

- Asymptotic notation captures behavior of functions for large values of x.
- Dominant term of $3x^3 + 5x^2 - 9$ is $3x^3$
- As x becomes larger and larger, other terms become insignificant and only $3x^3$ remains in the picture

Complexity Analysis

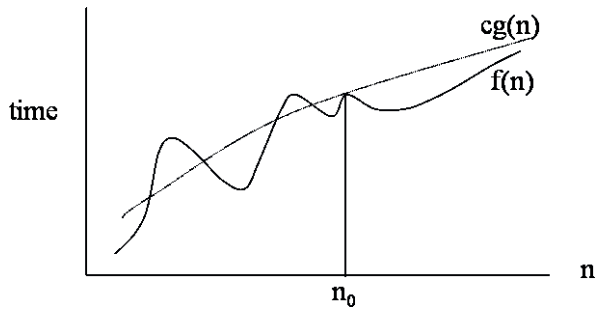
- O(·)
  - big o (big oh)
- Ω(·)
  - big omega
- Θ(·)
  - big theta

O(·)

- 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

$$f(n): N \to R^+ \ and \ g(n): N \to R^+$$

- if there exists positive constants c and n_0 such that

$$f(n) \leq cg(n), \quad for\ all\ n > n_0$$

- 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

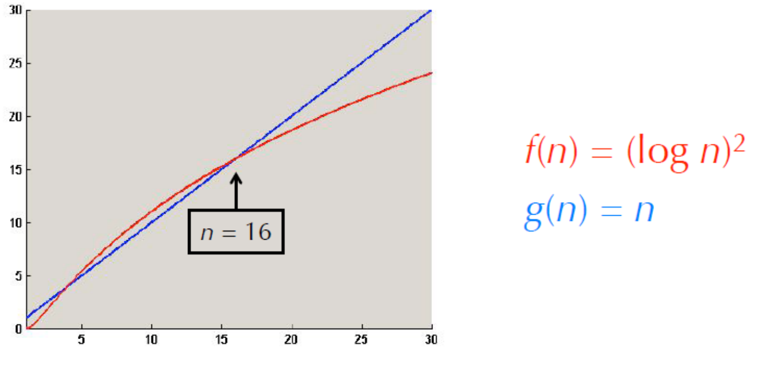
$$0 \leq \lim_{n \to \infty} \frac{f(n)}{g(n)} = c < \infty$$

- then we can write

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

Example

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



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

Notational Issues

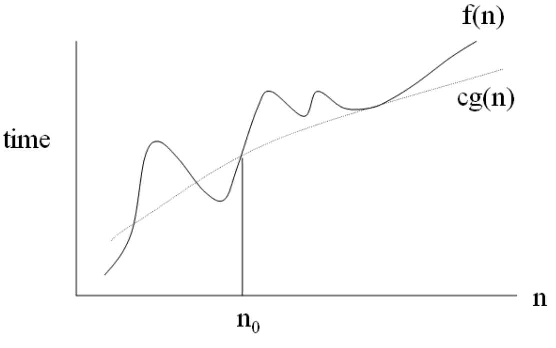
- Big-O notation is a way of comparing functions
- Notation quite unconventional
  - e.g., $3x^3 + 5x^2 - 9 = O(x^3)$
- Doesn't mean
- "$3x^3 + 5x^2 - 9$ equals the function $O(x^3)$"
- "$3x^3 + 5x^2 - 9$ is big oh of x^3"
- But
  - "$3x^3+5x^2 - 9$ is dominated by x^3"

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 Ω(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

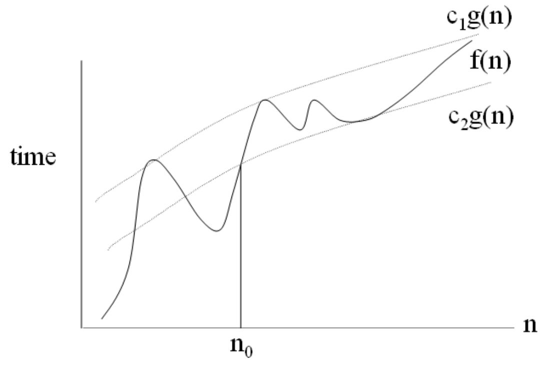
$$f(n) \geq cg(n), \quad \text{for all } n > n_0$$

- then we can write

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

Tightly Bounding Running Time

- f(n) is Θ(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

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

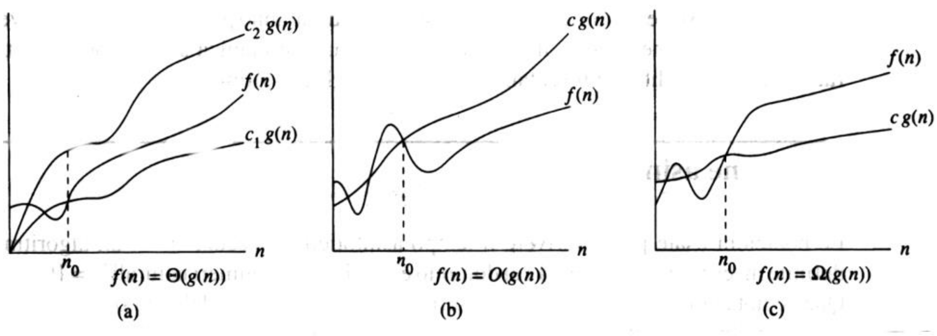
- if there exists positive constants c_1, c_2 and n_0 such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n), \quad for\ all\ n > n_0$$

- then we can write

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

Big-Θ, Big-O, and Big-Ω



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

$1/x$

$13+1/x$

$\ln x \ \lg_2 x$

$x+\sin x, x$

$x \ln x$

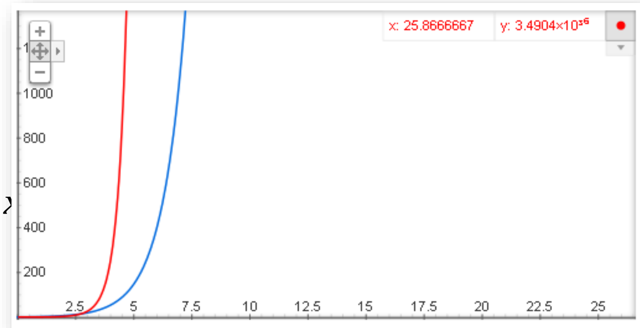
$x(\ln x)^2$

x^e

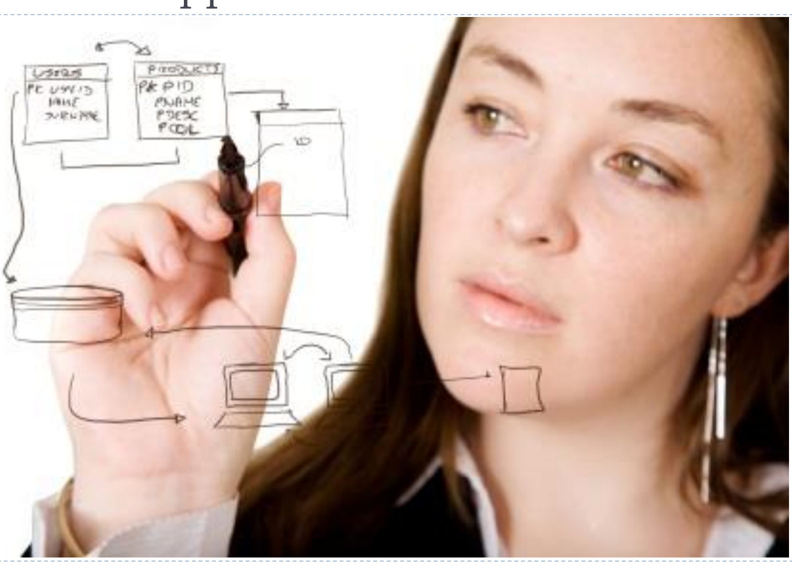
$(x+\sin x)(x^{20}-102)$

e^x

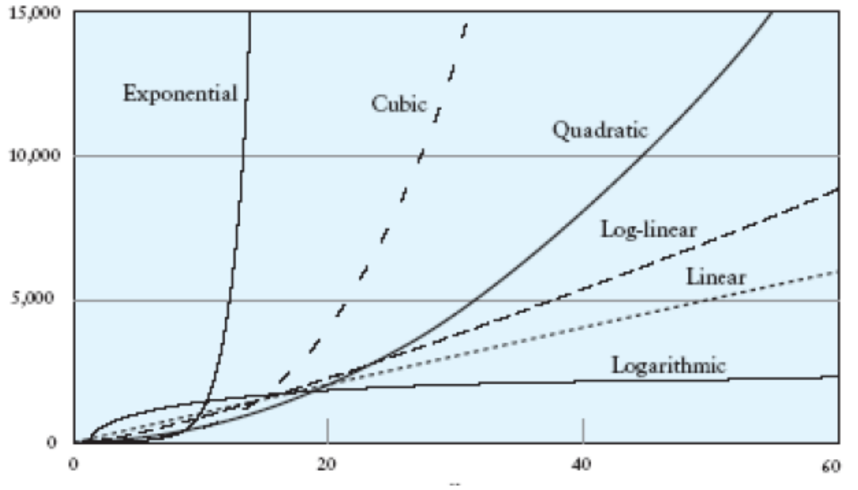
x^x



Practical approach



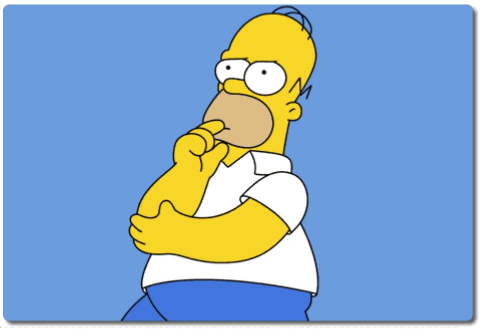
Practical approach



Class	Complexity	Number of Operations and Execution Time (1 instr/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-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) {      ── O(n)
        ++zap;                    ── 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;
    }
}
```

O(n)

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;
    }
}
```

$n-t$

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

Sequential statements

```
for(int z=0; z<n; ++z)
    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;
    }
}
```

O(n) — the inner `for` loop

O(1) — `zap = 0;`

Conditionals

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

$O(n^2)$ — the whole loop

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

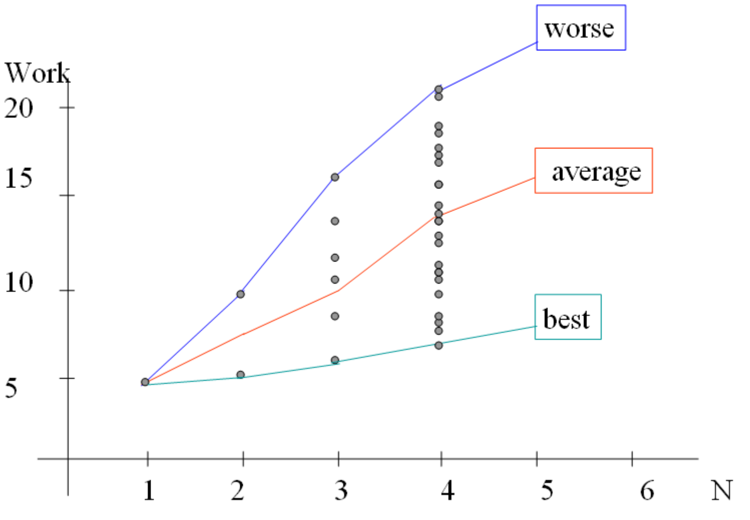
Caveats

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

Caveats

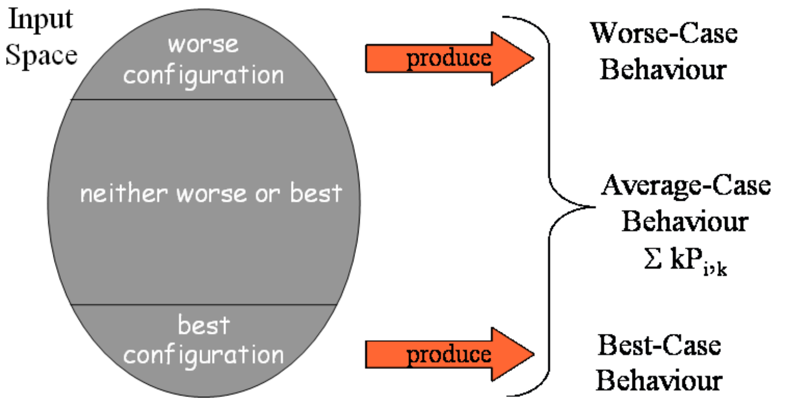
- Real time vs. complexity
- CPU time vs. RAM vs. disk
- Worse, Average or Best Case?

Worse, Average or Best Case?



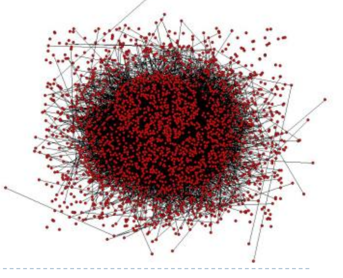
Worse, Average or Best Case?

- 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

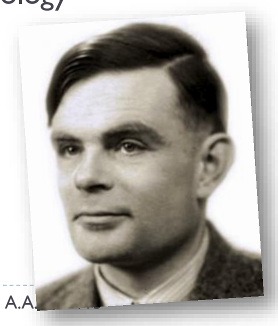
- E.g.,
  - Sort
  - Shortest path

Complexity Classes

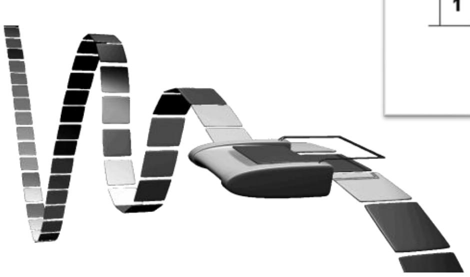
- A complexity class is the set of all of the computational problems which can be solved using a certain amount of a certain computational resource

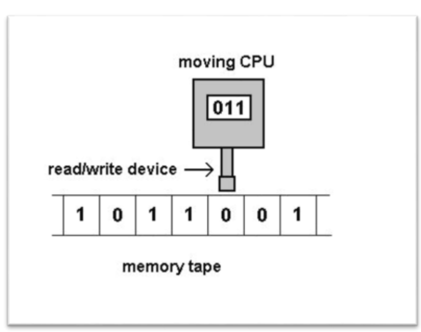
Deterministic Turing Machine

- 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



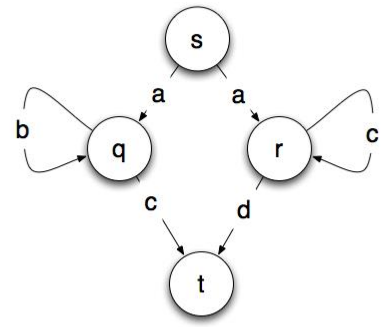
Deterministic Turing Machine

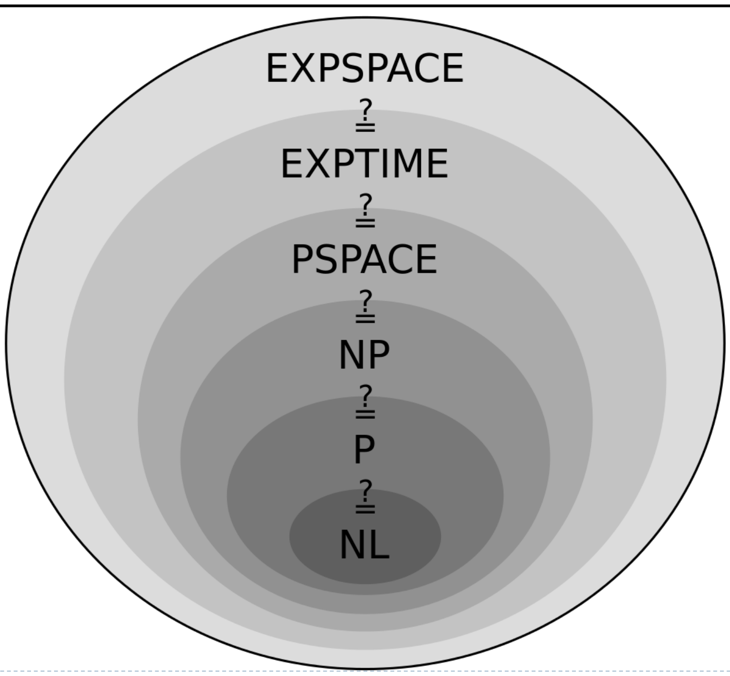




Non-Deterministic Turing Machine

- 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)
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)

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

Licenza d'uso