ALGORITHMS AND COMPLEXITY (I)

Blue Waters

13 quadrillion calculations per second

13,000,000,000,000,000

13 quadrillion calculations per second

13,000,000,000,000,000

7.42×10^{37} millennia

An *algorithm* is a well-defined computational procedure that takes some values as input and produces some values as output.
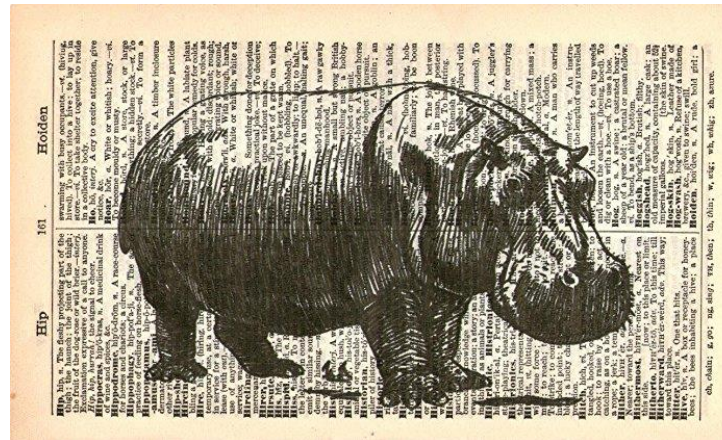
Search a 1024 page dictionary to find which page the word "hippopotamus" is on.

Option 1: start on page 1 and begin reading

Option 2: turn to middle, if word is before that page, it must be in first half; otherwise last half. Repeat.

1024/2→512/2→256/2→128/2→64/2→32/2→16/2→8/2→4/2→2/2→1
10 comparisons

1. Algorithms help us to understand *scalability*
2. Algorithmic mathematics provides a *language* for talking about program behavior
3. Performance is the *currency* of computing

Performance often draws the line between **what is feasible** and **what is impossible…**

1. Correctness
2. Time Complexity
3. Space Complexity

A (Correct, but Bad) Algorithm for the Assignment Problem:

Enumerate all possible assignments and choose the cheapest one.

E.g., assign members of set N1 to members of set N2
Let $n = |N1| = |N2|$ Total: **n!** possible assignments
Suppose n = 70
70! =**1,978,571,669,969,891,796,072,783,721, 689,098,736,458,938,142,546,425,857,555, 362,864,628,009,582,789,845,319,680,000, 000,000,000,000**

ALGORITHMS AND COMPLEXITY (II)

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Pseudocode

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Hides program design issues
- Preferred notation for describing algorithms

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*

Algorithm 1 Adaboost (with Stump as the base learner)

Input: S: training set $\{(x(1),\ldots,x(N)\}$ with labels $\{y(1)\ldots,y(N)\}$
    T: test set $\{u(1),u(2)\ldots u(N')\}$
    K: number of rounds
Output: predictions $\{h(u_1),h(u_2)\ldots h(u_{N'})\}$ for all test instances.

    For all $i=1:N,\ w_1(i)=1/N$
    for $r=1$ to K do
        For all $i=1:N,\ p_r(i)=w_r(i)/\sum_i(w_r(i))$
        $S_r = sampleByWeight(S,p_r,L)$, where $L=N$
        generate a new stump stp_r, call $stp_r.learn(S_r)$
        $h_r^S = stp_r.classify(S)$
        $\epsilon_r = \sum_i p_r(i)\mathbf{1}[h_r^S(i)\neq y(i)]$
        if $\epsilon_r > 1/2$ or $\epsilon_r = 0$ then
            $K = r-1$
            Exit the loop
        end if
        $\beta_r = \epsilon_r/(1-\epsilon_r)$
        For all $i=1:N,\ w_{r+1}(i)=w_r(i)\beta_r^{1-\mathbf{1}[h_r^S(i)\neq y(i)]}$
    end for
    For all $r=1:K,\ h_r^T = stp_r.classify(T)$
    For all $i=1:N',\ h(u_i)=\arg\max_{y\in Y}\sum_{r=1}^{K}\log(1/\beta_r)\mathbf{1}[h_r^T(u_i)=y]$
    return $\{h(u_1),h(u_2)\ldots h(u_{N'})\}$

$\text{JOHNSON}(G,w)$

1   compute G', where $G'.V = G.V \cup \{s\}$,
      $G'.E = G.E \cup \{(s,v): v \in G.V\}$, and
      $w(s,v) = 0$ for all $v \in G.V$
2   if $\text{BELLMAN-FORD}(G',w,s) == \text{FALSE}$
3   print "the input graph contains a negative-weight cycle"
4   else for each vertex $v \in G'.V$
5   set $h(v)$ to the value of $\delta(s,v)$
        computed by the Bellman-Ford algorithm
6   for each edge $(u,v) \in G'.E$
7        $\hat{w}(u,v) = w(u,v) + h(u) - h(v)$
8   let $D = (d_{uv})$ be a new $n \times n$ matrix
9   for each vertex $u \in G.V$
10   run $\text{DIJKSTRA}(G,\hat{w},u)$ to compute $\hat{\delta}(u,v)$ for all $v \in G.V$
11   for each vertex $v \in G.V$
        $d_{uv} = \hat{\delta}(u,v) + h(v) - h(u)$
urn D

Algorithm 2: Division

nction divide (x,y);
put: Two n-bit integers x and y, where $y \geq 1$
utput: The quotient and remainder of x divided by y
$x = 0$ then
    return $(q,r) = (0,0)$
se
    set $(q,r) = \text{divide}(\lfloor\frac{x}{2}\rfloor,y)$;
    $q = 2 \times q,\ r = 2 \times r$;
    if x is odd then
        $r = r + 1$
    end
    if $r \geq y$ then
        $r = r - y,\ q = q + 1$
    end
return (q,r)

Al
Re

En
1:
2:
3:
4:
5:
6:
7:
8:
9:
10:
11:
12:
13:
14:
15:
16:    end if
17:    end for
18:    $stepNo \leftarrow$
19: end loop

SAT
form,
cayR
dure

ical k-mer for x

10:       add x_{rep} to B
11: for all reads s do
12:    for all k-mers x in s do
13:       $x_{rep} \leftarrow \min(x,\text{revcomp}(x))$
14:       if $x_{rep} \in T$ then
15:          $T[x_{rep}] \leftarrow T[x_{rep}] + 1$

Control flow

- `if…then… [else …]`
- `while… do…`
- `repeat… until…`
- `for …do…`
- **indentation replaces braces**

Method declaration

Algorithm *method* (*arg* [, *arg*…])

    Input …

    Output …

Expressions

$\leftarrow$ assignment (similar to R)

$=$ equality testing (like "$==$")

n^2 mathematical formulas and superscripts allowed

procedure iterated hill-climber
begin
$t \leftarrow 0$
initialize *best*
repeat

procedure simulated annealing
begin

procedure tabu search
begin
tries $\leftarrow 0$

procedure evolutionary algorithm
begin
$t \leftarrow 0$
initialize $P(t)$
evaluate $P(t)$
while (**not** termination-condition) **do**
begin
$t \leftarrow t + 1$
select $P(t)$ from $P(t-1)$
alter $P(t)$
evaluate $P(t)$
end
end
until $t = MAX$
end

a tou
a se
he be
ppro
tabu
ew t
pdat
co
nt =
ies +
rent t
then update g
until tries = MA
end

Algorithm 3 NSGA-II algorithm
1: **procedure** NSGA-II($\mathcal{N}', g, f_k(\mathbf{x}_k)$) $\triangleright \mathcal{N}'$ members evolved g generations to solve $f_k(\mathbf{x})$
2:   Initialize Population $\mathbb{P}'$
3:   Generate random population - size $\mathcal{N}'$
4:   Evaluate Objective Values
5:   Assign Rank (level) Based on Pareto dominance - *sort*
6:   Generate Child Population
7:     Binary Tournament Selection
8:     Recombination and Mutation
9:   **for** $i = 1$ to g **do**
10:       **for** each Parent and Child in Population **do**
11:           Assign Rank (level) based on Pareto - *sort*
12:           Generate sets of nondominated vectors along PF_{known}
13:           Loop (inside) by adding solutions to next generation starting from the *first* front until $\mathcal{N}'$ individuals found determine crowding distance between points on each front
14:       **end for**
15:       Select points (elitist) on the lower front (with lower rank) and are outside a crowding distance
16:       Create next generation
17:           Binary Tournament Selection
18:           Recombination and Mutation
19:   **end for**
20: **end procedure**

ALGORITHMS AND COMPLEXITY (III)

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation

- Characterizes running time as a function of the input size, n

- Takes into account all possible inputs

- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

primitive operations

- Basic computations performed by an algorithm
- Largely independent from the programming language
- Exact definition not important
- Assumed to take a constant amount of time

Examples:

- **Evaluating an expression**
- **Assigning a value to a variable**
- **Indexing into an array**
- **Calling a method**
- **Returning from a method**

Algorithm *arrayMax*(*A*, *n*)

    currentMax ← *A*[0]

    for *i* ← 1 **to** *n* − 1 **do**

        if *A*[*i*] > *currentMax* **then**

            currentMax ← *A*[*i*]

    { increment counter *i* }

    return *currentMax*

\# operations

2

2 + *n*

2(*n* − 1)

2(*n* − 1)

2(*n* − 1)

1

Total   7*n* − 1

Define:

a = time taken by the fastest primitive operation

b = time taken by the slowest primitive operation

Let $T(n)$ be *worst-case time* of *arrayMax*

Then: $a(7n - 1) \leq T(n) \leq b(7n - 1)$

Hence, the running time $T(n)$ is bounded by two *linear functions*.

The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm *arrayMax*

Write an algorithm using pseudocode that reads a positive integer N and calculates the sum of all integers 1..N, i.e., 1 + 2 + ... + N = ?

Algorithm Name: **Sum** (N)

Input: integer $N \geq 0$

Output: sum of integers 1 to N

<u># of operations</u>

$total \leftarrow 0$ 1

for $i \leftarrow 1$ to N do   3+N

  $total \leftarrow total + i$ 2N

                              (increment counter)   N

return $total$ 1

Algorithm Name: **Sum** (N)

Input: integer $N \geq 0$

Output: sum of integers 1 to N

$$total \leftarrow \frac{N\,(N+1)}{2}$$

return *total*

# of operations

4

1

ALGORITHMS AND COMPLEXITY (IV)

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Worst-case: (usually)

   $T(n)$ = maximum time of algorithm on any input of size n

Average-case: (sometimes)

   $T(n)$ = expected time of algorithm over all inputs of size n

   Need assumption of statistical distribution of inputs

Best-case: (bogus)

   Cheat with a slow algorithm that works fast on *some* input

Example: the problem of sorting

Input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

Output: permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

Example:

Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9

INSERTION-SORT (A, n)

   for $j \leftarrow 2$ **to** n

       do $key \leftarrow A[j]$

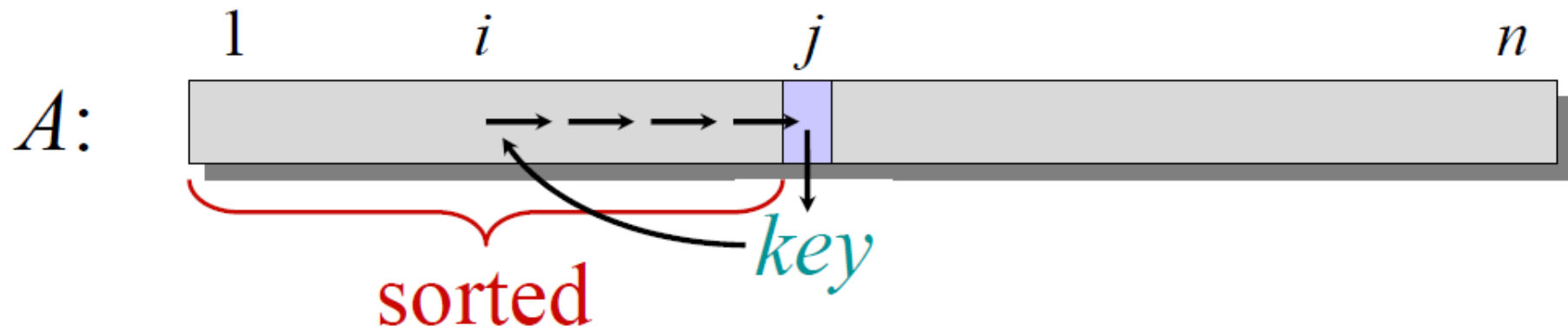 $i \leftarrow j - 1$

           while $i > 0$ and $A[i] > key$

              do $A[i+1] \leftarrow A[i]$

                $i \leftarrow i - 1$

       $A[i + 1] \leftarrow key$

A:

1   i j n

key

sorted

8   2   4   9   3   6

2   8   4   9   3   6

j=2

key = A[j] = A[2] = 2

i=j-1=1

A[i] = A[1]=8 > key =2

so, A[i+1] $\leftarrow$ A[i]

     A[2] $\leftarrow$ A[1] = 8

$$\text{INSERTION-SORT}\ (A, n) \quad \triangleright A[1 \ldots n]$$
$$\textbf{for}\ j \leftarrow 2\ \textbf{to}\ n$$
$$\textbf{do}\ key \leftarrow A[j]$$
$$i \leftarrow j - 1$$
$$\textbf{while}\ i > 0\ \text{and}\ A[i] > key$$
$$\textbf{do}\ A[i+1] \leftarrow A[i]$$
$$i \leftarrow i - 1$$
$$A[i+1] = key$$

i $\leftarrow$ i-1 =0   A[i+1]=A[0+1]=A[1] $\leftarrow$ key = 2

8   2   4   9   3   6

2   8   4   9   3   6

j=3;   key = A[j] = A[3] = 4;   i=j-1=2

A[i] = A[2]=8 > key =4   A[i] = A[1]
so, A[i+1] ← A[i]   so, end w
    A[3] ←A[2] = 8
                                A[i+1]=A[1+1]=A[2] ← key = 4

i ← i-1 =1

INSERTION-SORT (A, n) ▷ $A[1 \dots n]$
   for $j \leftarrow 2$ **to** n
      do $key \leftarrow A[j]$
         $i \leftarrow j - 1$
         while $i > 0$ and $A[i] > key$
            do $A[i+1] \leftarrow A[i]$
               $i \leftarrow i - 1$
      $A[i+1] = key$

8   2   4   9   3   6

2   8   4   9   3   6

2   4   8   9
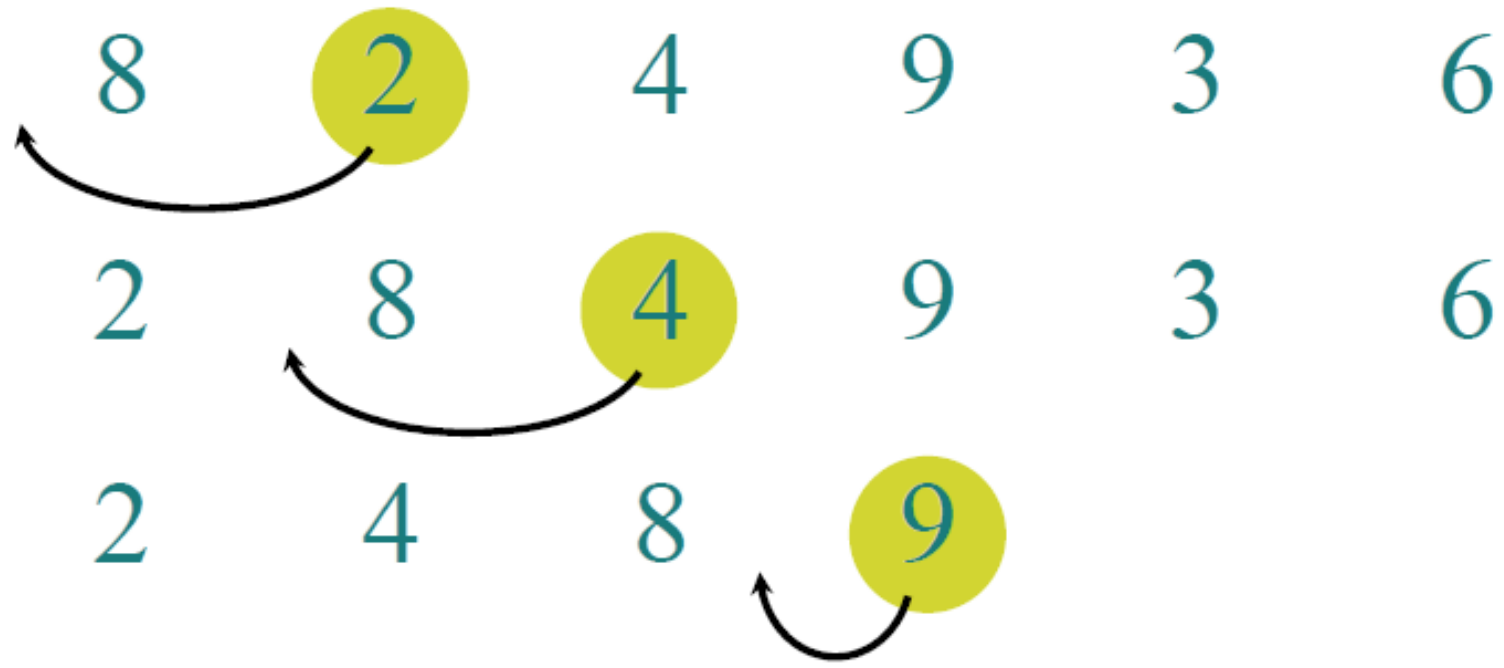
running time

- ~~Running time depends on the input: e.g., an already sorted sequence is easier to sort~~ **Usually: worst-case!**

- Parameterize the running time by the *size of the input*, e.g. short sequen... than long ones

- Generally, we seek ...ing time, because everybo...

- Speed of computer

$$\text{INSERTION-SORT}\,(A, n) \qquad \triangleright A[1 \,..\, n]$$
$$\textbf{for } j \leftarrow 2 \textbf{ to } n$$
$$\qquad \textbf{do } key \leftarrow A[j]$$
$$\qquad\quad i \leftarrow j - 1$$
$$\qquad\quad \textbf{while } i > 0 \text{ and } A[i] > key$$
$$\qquad\qquad \textbf{do } A[i+1] \leftarrow A[i]$$
$$\qquad\qquad\quad i \leftarrow i - 1$$
$$\qquad A[i+1] = key$$

ALGORITHMS AND COMPLEXITY (V)

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, *n*
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

What is insertion sort's worst-case time?

- It depends on the speed of our computer:
  - relative speed (on the same machine)
  - absolute speed (on different machines)

BIG IDEA:

- Ignore machine-dependent constants.
- Look at *growth* of $T(n)$ as $n \to \infty$

"Asymptotic Analysis"

INSERTION-SORT (A, n)

    for $j \leftarrow 2$ **to** n

        do $key \leftarrow A[j]$

            $i \leftarrow j - 1$

            while $i > 0$ **and** $A[i] > key$

                do $A[i+1] \leftarrow A[i]$

                    $i \leftarrow i - 1$

            $A[i+1] = key$

$$T(n) = \frac{1}{2} p \left(n^2 - n \right) \qquad \text{for some } 1 \leq p \leq P$$

$$\frac{1}{2} \left(n^2 - n \right) \leq T(n) \leq \frac{1}{2} P \left(n^2 - n \right)$$

Pro-tip: *constants and lower-order terms don't matter*

"Big O" notation describes the asymptotic upper bound of an algorithms growth rate

$$T(n) \in \mathcal{O}\left(f(n)\right)$$

The asymptotic lower bound on growth rates is expressed by "Big Omega " notation

$$T(n) \in \Omega\left(f(n)\right)$$

If the upper and lower bounds are the same, "Big Theta"

$$T(n) \in \Theta\left(f(n)\right)$$

$$T(n) \in \mathcal{O}\left(f(n)\right)$$
$$\text{iff } \exists N \geq 0, c > 0,$$
$$T(n) \leq cf(n) \ \forall n \geq N$$

$$T(n) \in \mathcal{O}\left(f(n)\right) \text{ iff } \exists N \geq 0, c > 0,$$
$$T(n) \leq cf(n) \; \forall n \geq N$$

Examples:

One possible set of values

$$T(n) = 0.01n^3 \in \mathcal{O}(n^3)$$

$c = 0.02; N = 0;$
$T(n) \leq 0.02n^3 \; \forall n \geq 0$

$$T(n) = 6n^2 + 10n + 7 \in \mathcal{O}(n^2)$$

$c = 10; \; N = 4$
$T(n) \leq 10n^2 \; \forall n \geq 4$

$$T(n) = n^3 + 10^6 n^2 \in \mathcal{O}(n^3)$$

$c = 10; \; N = 112{,}000$
$T(n) \leq 10n^3 \; \forall n \geq 112000$

Many upper bounds exist…

$$T(n) = n^2 \in \mathcal{O}(n^2), \mathcal{O}(n^3), \ldots, \mathcal{O}(n^{100}), \ldots$$

Note: usually when people talk about Big-O, they normally talking about the smallest upper bound

$T(n) \in \mathcal{O}\left(f(n)\right)$ iff $\exists N \geq 0, c > 0,$
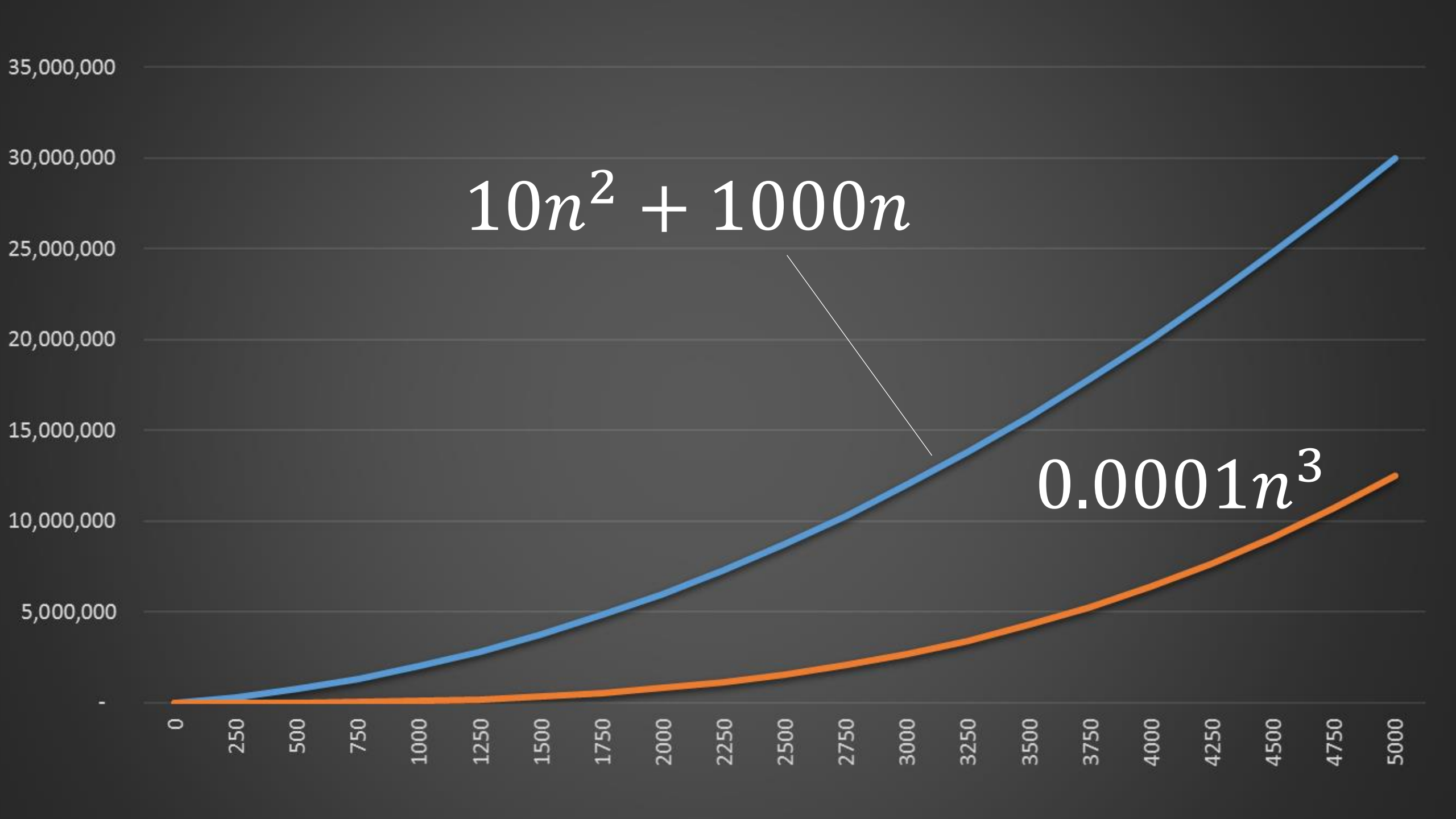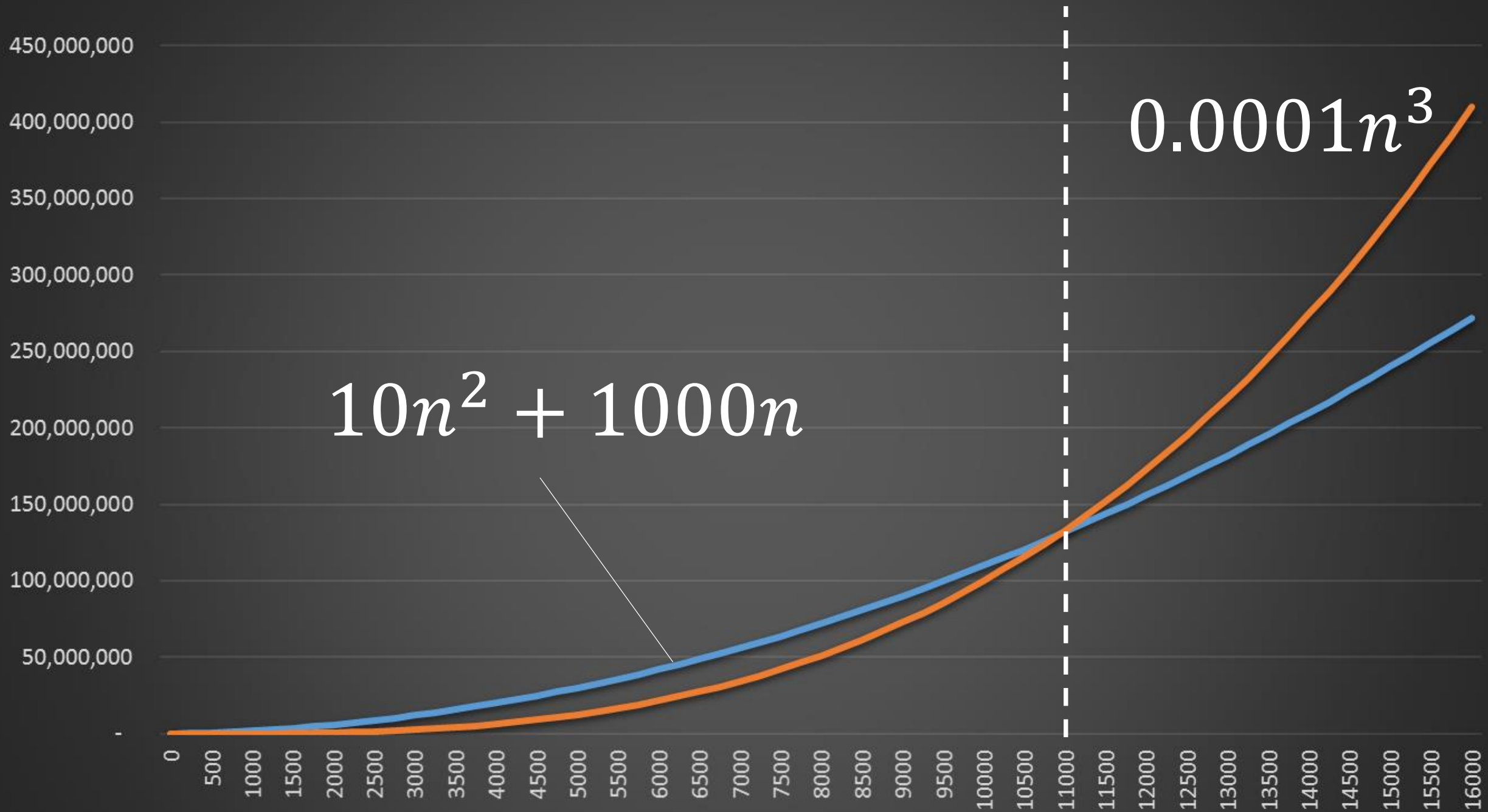$T(n) \leq cf(n) \; \forall n \geq N$

More examples:

$$0.0001n^3 \notin \mathcal{O}(n) \qquad 10n^2 + 1000n \notin \mathcal{O}(n)$$

$$0.0001n^3 \notin \mathcal{O}(n^2) \qquad 10n^2 + 1000n \in \mathcal{O}(n^2)$$

$$0.0001n^3 \in \mathcal{O}(n^3) \qquad 10n^2 + 1000n \in \mathcal{O}(n^3)$$

$$0.0001n^3 \in \mathcal{O}(n^4) \qquad 10n^2 + 1000n \in \mathcal{O}(n^4)$$

$10n^2 + 1000n$

$0.0001n^3$

$$T(n) \in \mathcal{O}\left(f(n)\right) \text{ iff } \exists N \geq 0, c > 0,$$
$$T(n) \leq cf(n) \; \forall n \geq N$$

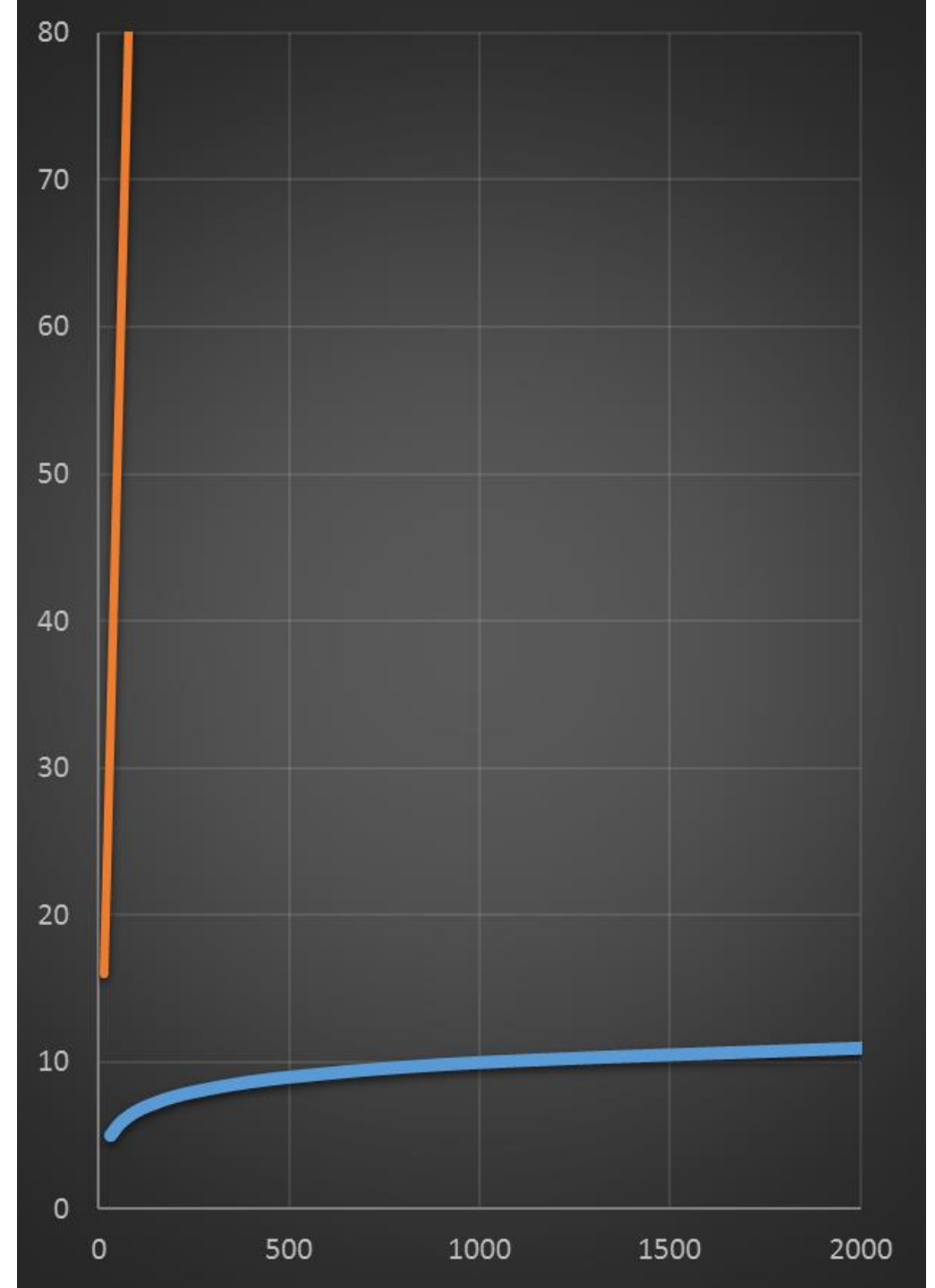$$T(n) \in \Omega\left(f(n)\right) \text{ iff } \exists N \geq 0, l > 0,$$
$$T(n) \geq lf(n) \; \forall n \geq N$$

$$T(n) \in \Theta\left(f(n)\right) \text{ iff } \exists N \geq 0, c > 0, l > 0$$
$$lf(n) \leq T(n) \leq cf(n) \; \forall n \geq N$$

Complexity of the dictionary problem?

- How many comparisons as number of pages increases?
- Obviously less than linear growth...

tractability

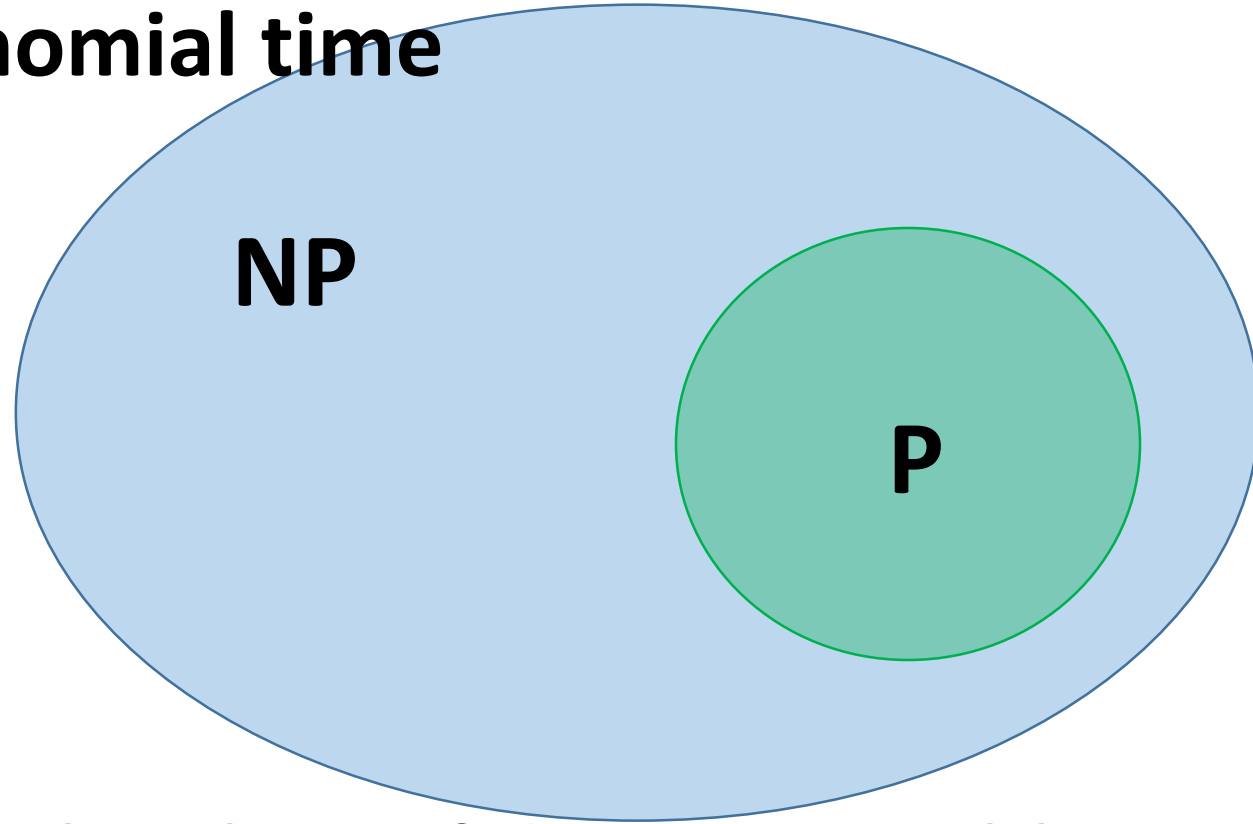|	
$\mathcal{O}(1)$	constant
$\mathcal{O}(\log n)$	logarithmic
$\mathcal{O}(n)$	linear
$\mathcal{O}(n \log n)$	loglinear
$\mathcal{O}(n^2)$	quadratic
$\mathcal{O}(n^c), c > 1$	polynomial
$\mathcal{O}(c^n)$	exponential
$\mathcal{O}(n!)$	factorial

ALGORITHMS AND COMPLEXITY (VI)

P and NP (**polynomial time** and **nondeterministic polynomial time**)

NP: class of decision problems whose solutions can be <u>**verified**</u> in polynomial time

NP

P

P: the class of decision problems that can be <u>**solved**</u> in polynomial time

decision problems vs. optimization problems

A decision problem asks us to check if something is true
(possible answers: 'yes' or 'no')

e.g.,

PRIMES

Given: a positive integer **n**

Question: is **n** prime?

decision problems vs. optimization problems

An optimization problem asks us to find, among all feasible solutions, one that maximizes or minimizes a given objective

e.g.,

SHORTEST PATH
Given: weighted graph G and two nodes *s* and *t*
Problem: find shortest path from *s* to *t*

KNAPSACK PROBLEM
Given: set of items and their value and volume, volume of knapsack
Problem: determine which items to select to maximize value

decision problems vs. optimization problems

A *decision version* of a given optimization problem can easily be defined using a bound on the value of feasible solutions

e.g.,
SHORTEST PATH DECISION PROBLEM
Given: weighted graph G and two nodes *s* and *t*
Question: is there a path from *s* to *t* of length at most *L?*

KNAPSACK DECISION PROBLEM
Given: set of items and their value and volume, volume of knapsack
Question: is there a combination of items that fit within the knapsack of value greater than or equal to *V*?

discovery vs. verification

- Two important tasks for a scientist are discovery of solutions, and verification of other people's solutions
- It is easier to check that a solution, say to a puzzle, is correct, rather than to find the solution
- That is, *verifying* a solution is easier than *discovering* it
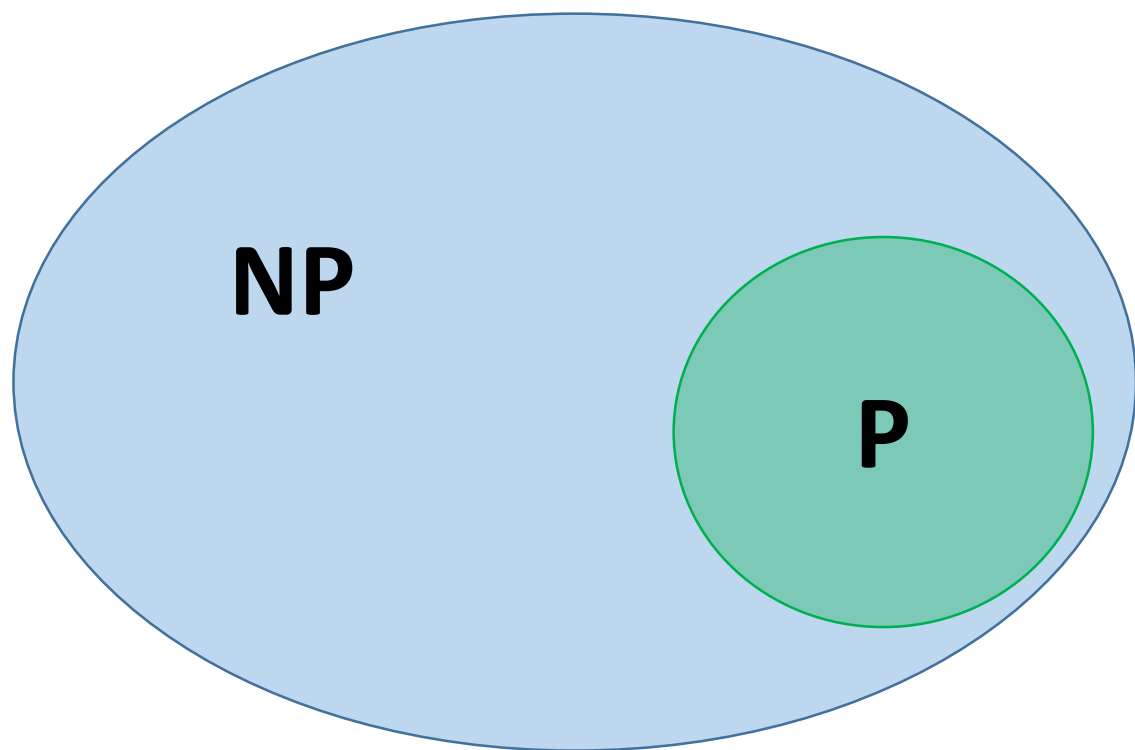- Example: Sudoku

sudoku

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

sudoku

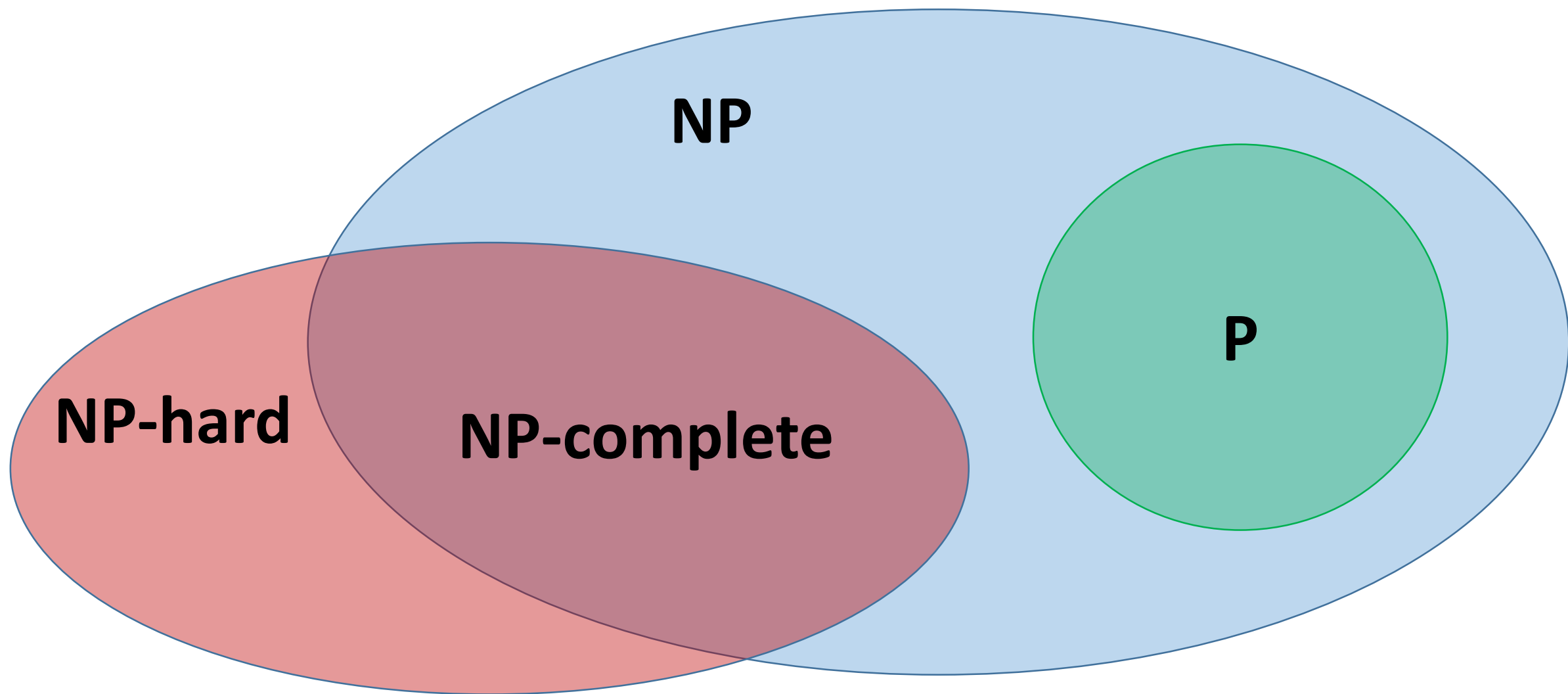| 9 | 8 | 3 | 7 | 2 | 1 | 6 | 5 | 4 |
7	6	5	8	4	9	3	2	1
1	2	4	3	6	5	7	9	8
8	1	7	9	5	6	2	4	3
3	4	6	2	1	8	5	7	9
5	9	2	4	7	3	1	8	6
6	7	8	5	3	4	9	1	2
4	5	1	6	9	2	8	3	7
2	3	9	1	8	7	4	6	5

NP

P

$$P \subseteq NP$$

P=NP
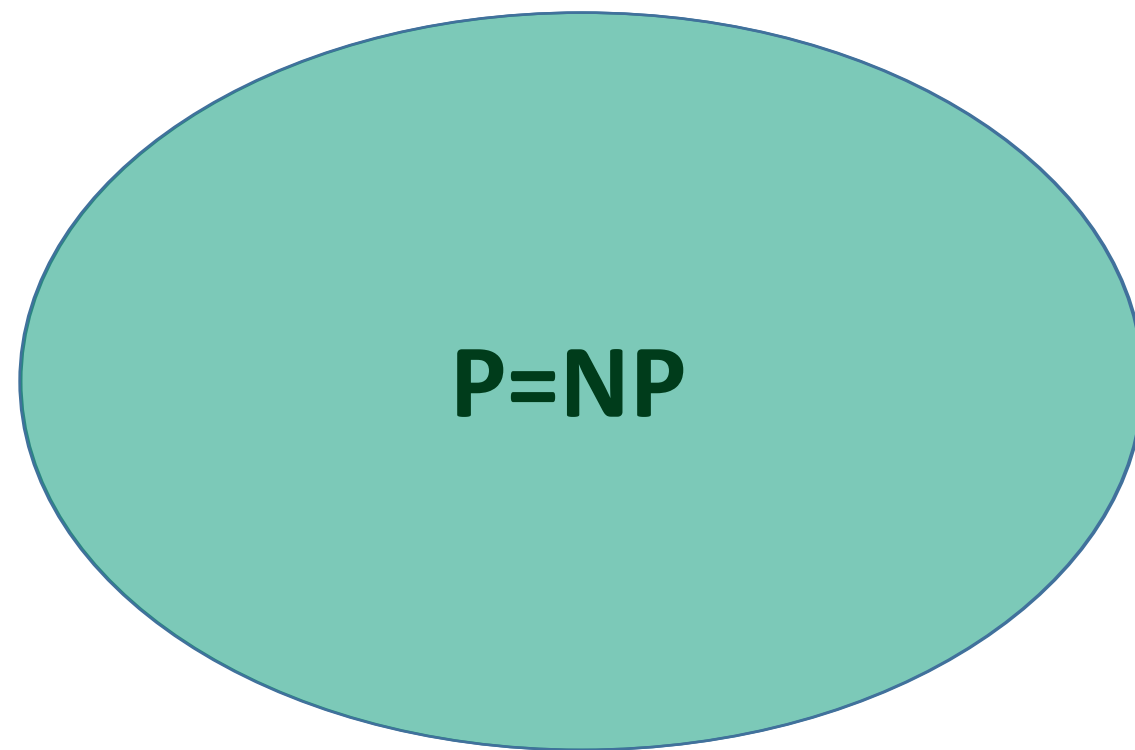
$$P = NP?$$

The P vs. NP problem has been called "one of the deepest questions ever asked by human beings"

NP-complete problems are the "hardest" problems in NP

- Examples: Sudoku and 3-colorability

If there is a fast (polynomial-time) algorithm for *one* NP-complete problem, then there is a fast algorithm for *every* problem in NP!

- For example, a fast algorithm for Sudoku implies P=NP.

The decision versions of Integer programming, TSP, set-covering are all NP-complete. The optimization versions of the problems are NP-hard.

Visualize, assume, simplify, formulate

Problem → Model → Solution

Decide on approach, solve