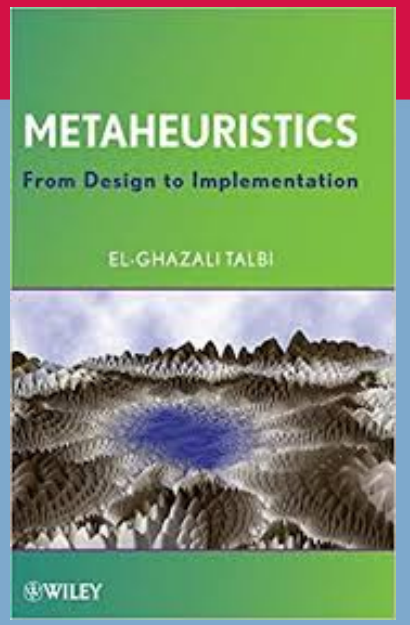
CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

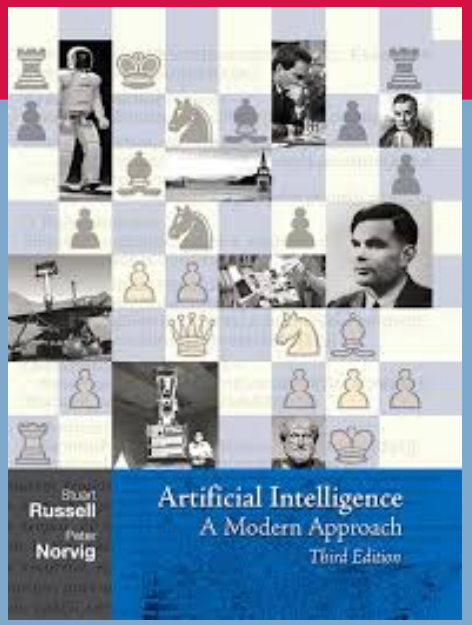
Computing@CIT

Metaheuristic Optimization

Week 2

Dr. Diarmuid Grimes





This module

Provide Answers to these questions:

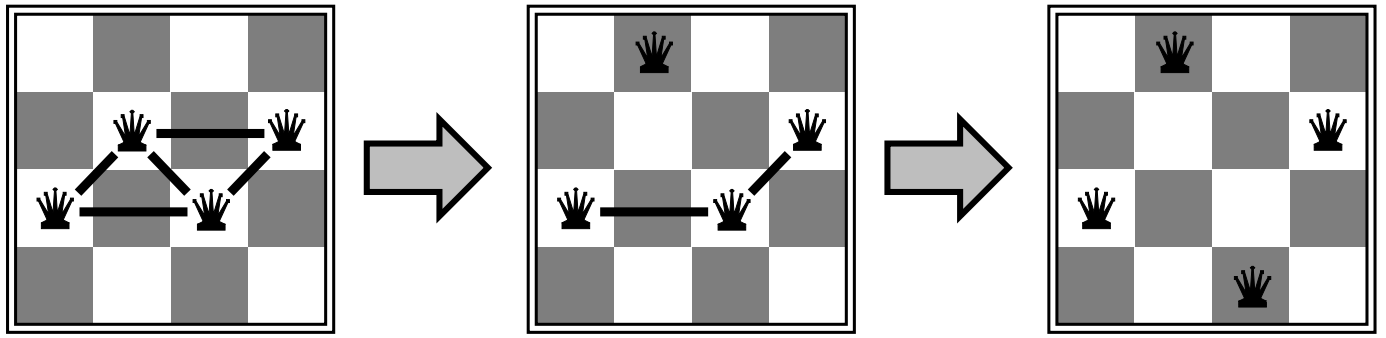
- Which metaheuristic methods are available and what are their features?
- How can metaheurists be used to solve computationally hard problems?
- How should heuristics methods be studied and analyzed empirically?
- How can heuristic algorithms be designed, developed, and implemented?

Heuristic example from Russel Norvig

Example: n-queens

Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

Move a queen to reduce number of conflicts



h = 5 h = 2 h = 0

Almost always solves n-queens problems almost instantaneously for very large n, e.g., $n = 1million$

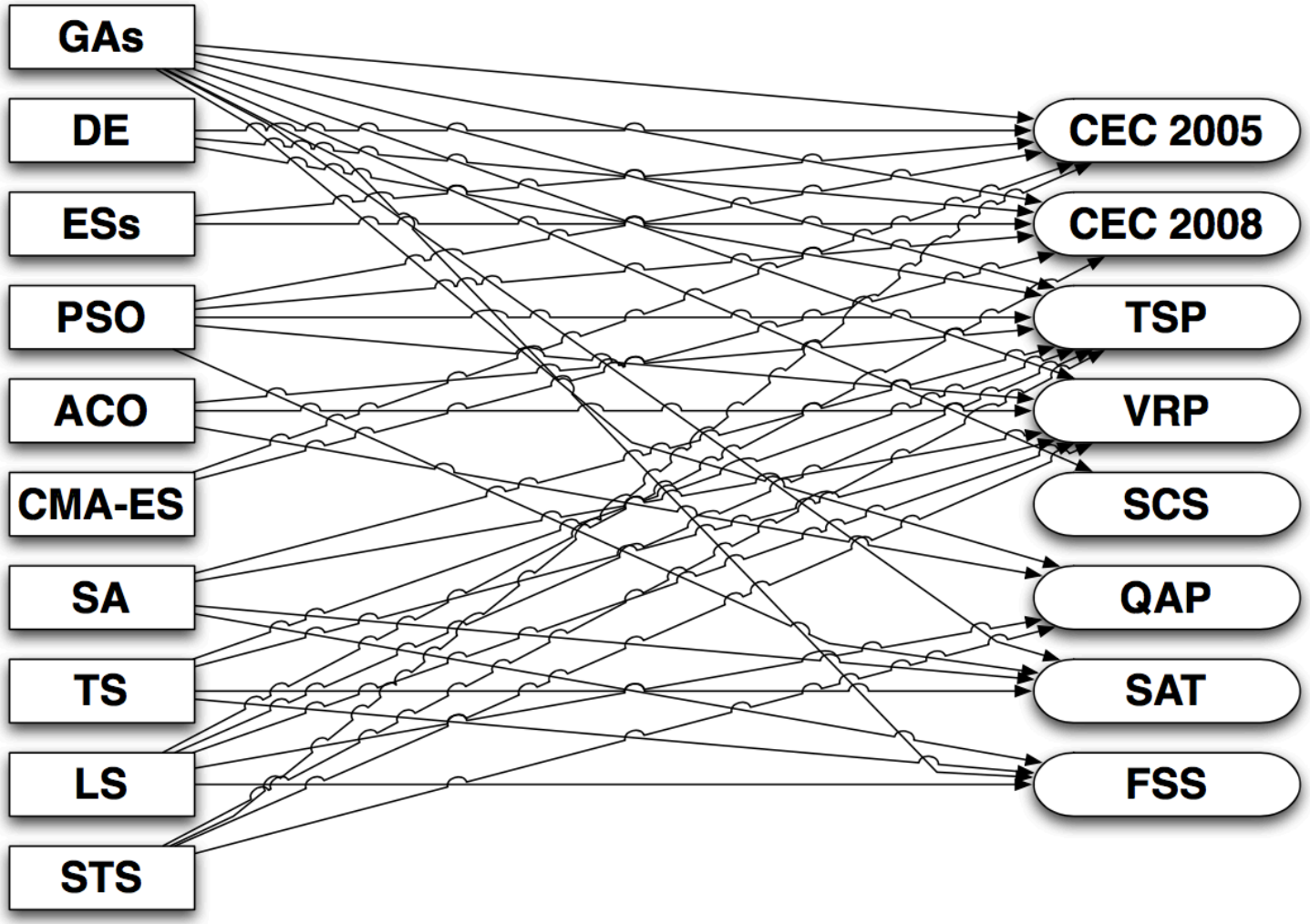
Motivations

- Different Metaheuristics
- Difficult selection of the appropriate metaheuristic for a given problem
- Synergies among different metaheuristics

Different Metaheuristics

- **Genetic Algorithms → GAs**
- Differential Evolution → DE
- **Ant Colony Optimization → ACO**
- Evolutionary Computation → ECs
- **Local Search → LS**
- **Tabu Search → TS**
- **Particle Swarm Optimization → PSO**
- More …

Difficult Selection of a Metaheuristic



Difficult Selection of a Metaheuristic

Algorithm	# of Publications	% Total
Local Search	61,700	39.81%
Evolutionary Strategies	19,400	12.51%
Genetic Algorithms	18,900	12.19%
Simulated Annealing	13,300	8.58%
Tabu Search	8,960	5.78%
Ant Colony Optimization	8,830	5.70%
Scatter Search	5,620	3.63%
Differential Evolution	4,460	2.88%
Particle Swarm Optimization	2,300	1.48%
Others	11,520	7.43%

155,000 results for the TSP in Google Scholar

Difficult Selection of a Metaheuristic

- You may be confused by the different methods, but there are **many** more out there!
- I'll only cover a ***selection*** of the metaheuristics which I consider essential
- **There is no such thing as a best metaheuristic!**. This has actually been proven mathematically! No Free Lunch Theorem (FLT), select the best metaheuristic for the problem at hand

Difficult Selection of a Metaheuristic

- **There is no such thing as a best metaheuristic!**. This has been actually proven mathe[illegible]tic for the pr[illegible]

IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, VOL. 1, NO. 1, APRIL 1997

No Free Lunch Theorems for Optimization

David H. Wolpert and William G. Macready

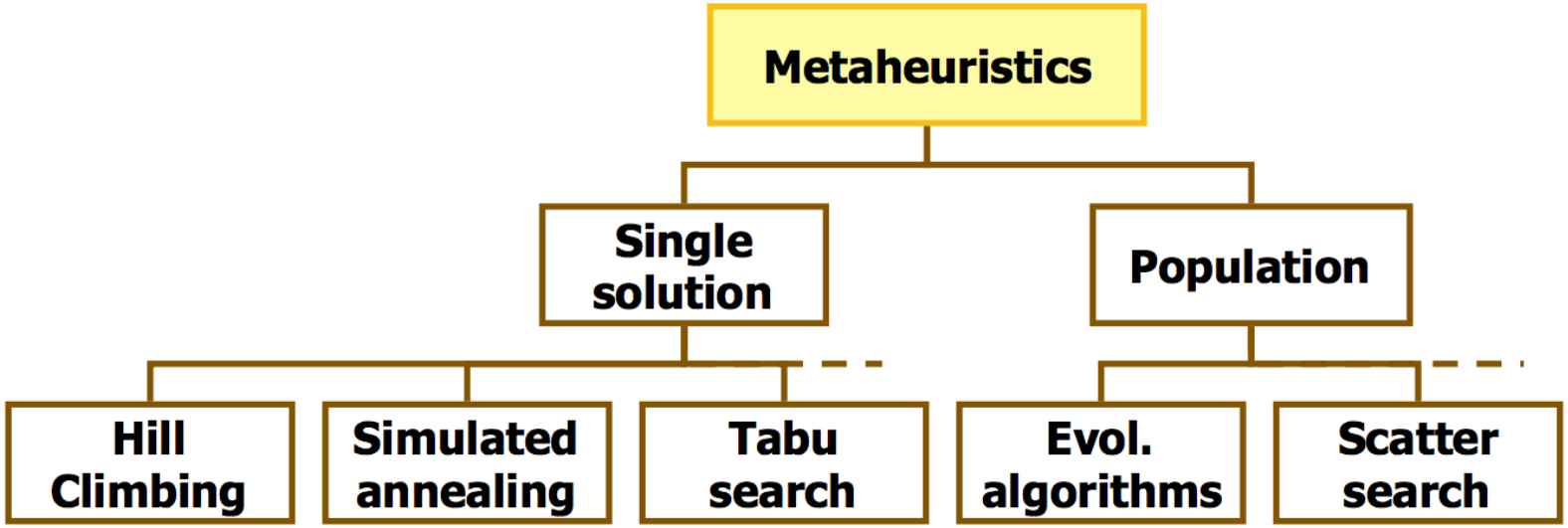
***Abstract*—A framework is developed to explore the connection between effective optimization algorithms and the problems they are solving. A number of "no free lunch" (NFL) theorems are presented which establish that for any algorithm, any elevated performance over one class of problems is offset by performance over another class. These theorems result in a geometric interpretation of what it means for an algorithm to be well suited to an optimization problem. Applications of the NFL theorems to information-theoretic aspects of optimization and benchmark measures of performance are also presented. Other issues addressed include time-varying optimization problems and *a priori* "head-to-head" minimax distinctions between optimization algorithms, distinctions that result despite the NFL theorems' enforcing of a type of uniformity over all algorithms.**

***Index Terms*— Evolutionary algorithms, information theory, optimization.**

information theory and Bayesian analysis contribute to an understanding of these issues? How *a priori* generalizable are the performance results of a certain algorithm on a certain class of problems to its performance on other classes of problems? How should we even measure such generalization? How should we assess the performance of algorithms on problems so that we may programmatically compare those algorithms?

Broadly speaking, we take two approaches to these questions. First, we investigate what *a priori* restrictions there are on the performance of one or more algorithms as one runs over the set of all optimization problems. Our second approach is to instead focus on a particular problem and consider the effects of running over all algorithms. In the current paper we present results from both types of analyses but concentrate largely on the first approach. The reader is referred to the

Taxonomy of Metaheuristics



- Population-based metaheuristics: Evolutionary Algorithms, Scatter Search, Ant Systems, etc.
- Solution-based metaheuristics: Hill Climbing, Simulated Annealing, Tabu Search, etc…

Complexity

Time and space

- To ***analyze*** an algorithm means:
  - developing a formula for predicting *how fast* an algorithm is, based on the size of the input (***time complexity***), and/or
  - developing a formula for predicting *how much memory* an algorithm requires, based on the size of the input (***space complexity***)
- Usually time is our biggest concern
  - Most algorithms require a fixed amount of space

What does “size of the input” mean?

- If we are searching an array, the “size” of the input could be the size of the array
- If we are merging two arrays, the “size” could be the sum of the two array sizes
- If we are computing the n^{th} Fibonacci number, or the n^{th} factorial, the “size” is n
- We choose the “size” to be the parameter that most influences the actual time/space required
  - It is *usually* obvious what this parameter is
  - Sometimes we need two or more parameters

Average, best, and worst cases

- Usually we would like to find the *average* time to perform an algorithm
- However ….
  - Sometimes the “average” isn’t well defined
    - Example: Sorting an “average” array
      - Time typically depends on how out of order the array is
      - How out of order is the “average” unsorted array?
  - Sometimes finding the average is too difficult
- Often we have to be satisfied with finding the *worst* (longest) time required
  - Sometimes this is even what we want (say, for time-critical operations)
- The *best* (fastest) case is seldom of interest

Average, best, and worst cases -- Sorting

Method	Worst Case	Average Case
Selection sort	n^2	n^2
Inserstion sort	n^2	n^2
Merge sort	n log n	n log n
Quick sort	n^2	n log n

Constant time

- *Constant time* means there is some constant k such that this operation always takes k nanoseconds
- A Python statement takes constant time if:
  - It does not include a loop
  - It does not include calling a method whose time is unknown or is not a constant
- If a statement involves a choice (if or switch) among operations, each of which takes constant time, we consider the statement to take constant time
  - This is consistent with *worst-case analysis*

What is the runtime of g(n)?

```
def g(n):
  for i in range(n):
    f()
```

$$\text{Runtime}(g(n)) \approx n \cdot \text{Runtime}(f())$$

```
def g(n):
  for i in range(n):
    for j in range(n):
      f()
```

$$\text{Runtime}(g(n)) \approx n^2 \cdot \text{Runtime}(f())$$

What is the runtime of g(n)?

```
def g(n):
  for i in range(n):
    for j in range(i+1):
        f()
```

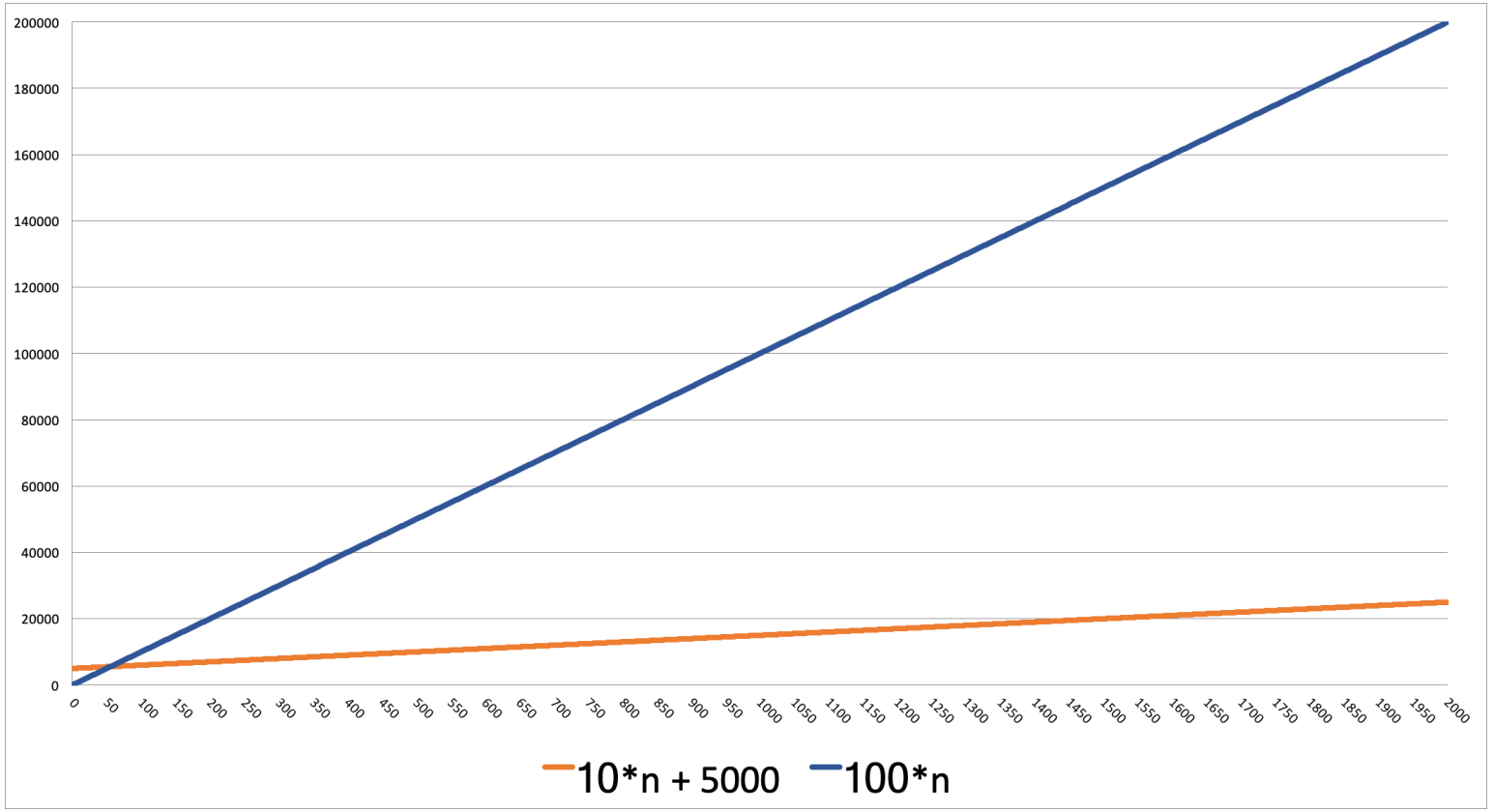
$$\begin{aligned}\text{Runtime}(g(n)) &\approx (1+2+3+\cdots+n)\cdot\text{Runtime}(f()) \\ &\approx \frac{n^2+n}{2}\cdot\text{Runtime}(f())\end{aligned}$$

Constant time is (usually) better than linear time

- Suppose we have two algorithms to solve a task:
  - Algorithm A takes 5000 time units
  - Algorithm B takes 100*n time units
- Which is better?
  - Clearly, algorithm B is better if our problem size is small, that is, if $n < 50$
  - Algorithm A is better for larger problems, with $n > 50$
  - So B is better on small problems that are quick anyway
  - But A is better for large problems, *where it matters more*
- We usually care most about very large problems
  - But not always!

Constant time is (usually) better than linear time

- Suppose we have two algorithms to solve a task:
  - Algorithm A takes $5000 + 10*n$ time units
  - Algorithm B takes $100*n$ time units



Complexity analysis

- A technique to characterize the execution time of an algorithm independently from the machine, the language and the compiler.

- Useful for:
  - evaluating the variations of execution time with regard to the input data
  - comparing algorithms

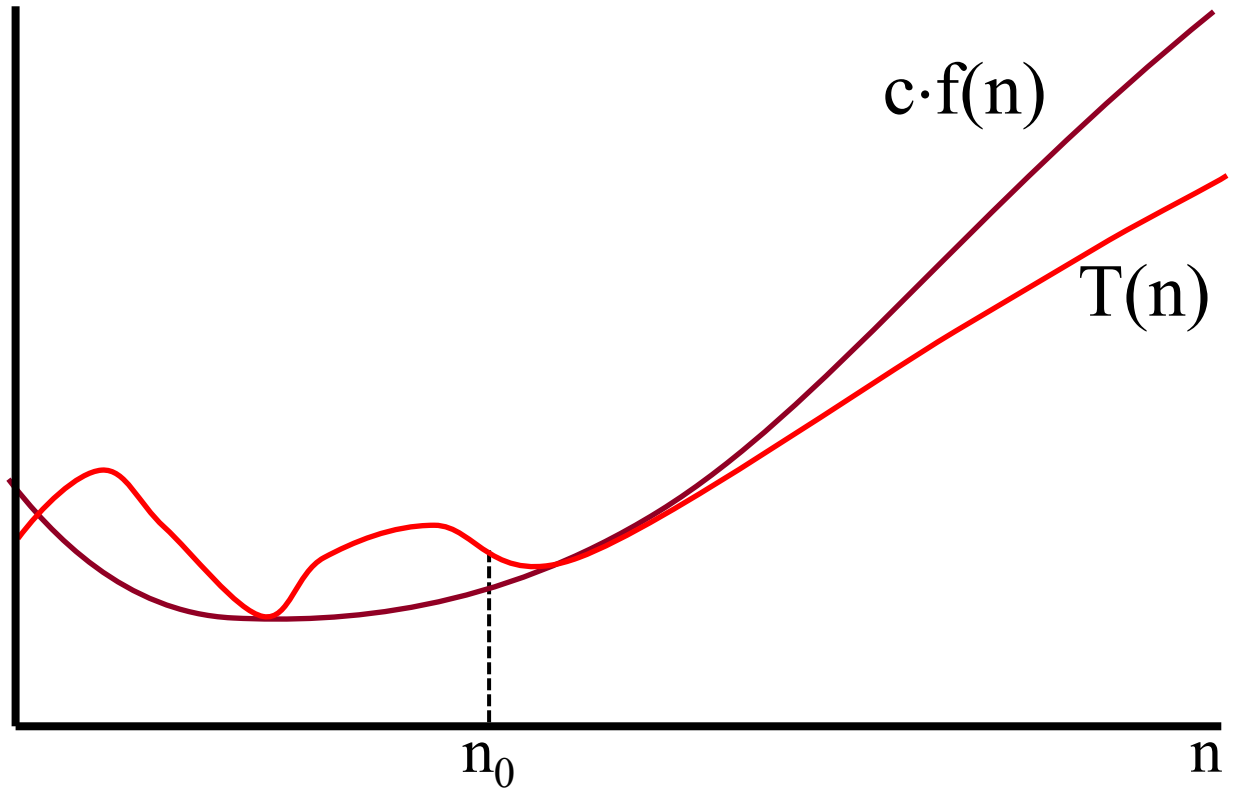
- We are typically interested in the execution time of large instances of a problem, e.g., when n → ∞, (asymptotic complexity).

Big O

- A method to characterize the execution time of an algorithm:
  - Adding two square matrices is $O(n^2)$
  - Searching in a dictionary is $O(\log n)$
  - Sorting a vector is $O(n \log n)$
  - Solving Towers of Hanoi is $O(2^n)$
  - Multiplying two square matrices is $O(n^3)$
  - …
- The O notation only uses the dominating terms of the execution time. Constants are disregarded.

Big O: formal definition

- Let $T(n)$ be the execution time of an algorithm when the size of input data is n.
- $T(n)$ is $O(f(n))$ if there are positive constants c and n_0 such that $T(n) \leq c \cdot f(n)$ when $n \geq n_0$.



Simplifying the formulae

- Throwing out the constants is one of *two* things we do in analysis of algorithms
  - By throwing out constants, we simplify $12n^2 + 35$ to just n^2
- Our timing formula is a polynomial, and may have terms of various orders (constant, linear, quadratic, cubic, etc.)
  - We usually discard all but the *highest-order* term
    - We simplify $n^2 + 3n + 5$ to just n^2

Big O notation

- When we have a polynomial that describes the time requirements of an algorithm, we simplify it by:
  - Throwing out all but the highest-order term
  - Throwing out all the constants
- If an algorithm takes $12n^3+4n^2+8n+35$ time, we simplify this formula to just n^3
- We say the algorithm requires $O(n^3)$ time
  - We call this Big O notation
  - (More accurately, it's Big Ω)

Can we justify Big O notation?

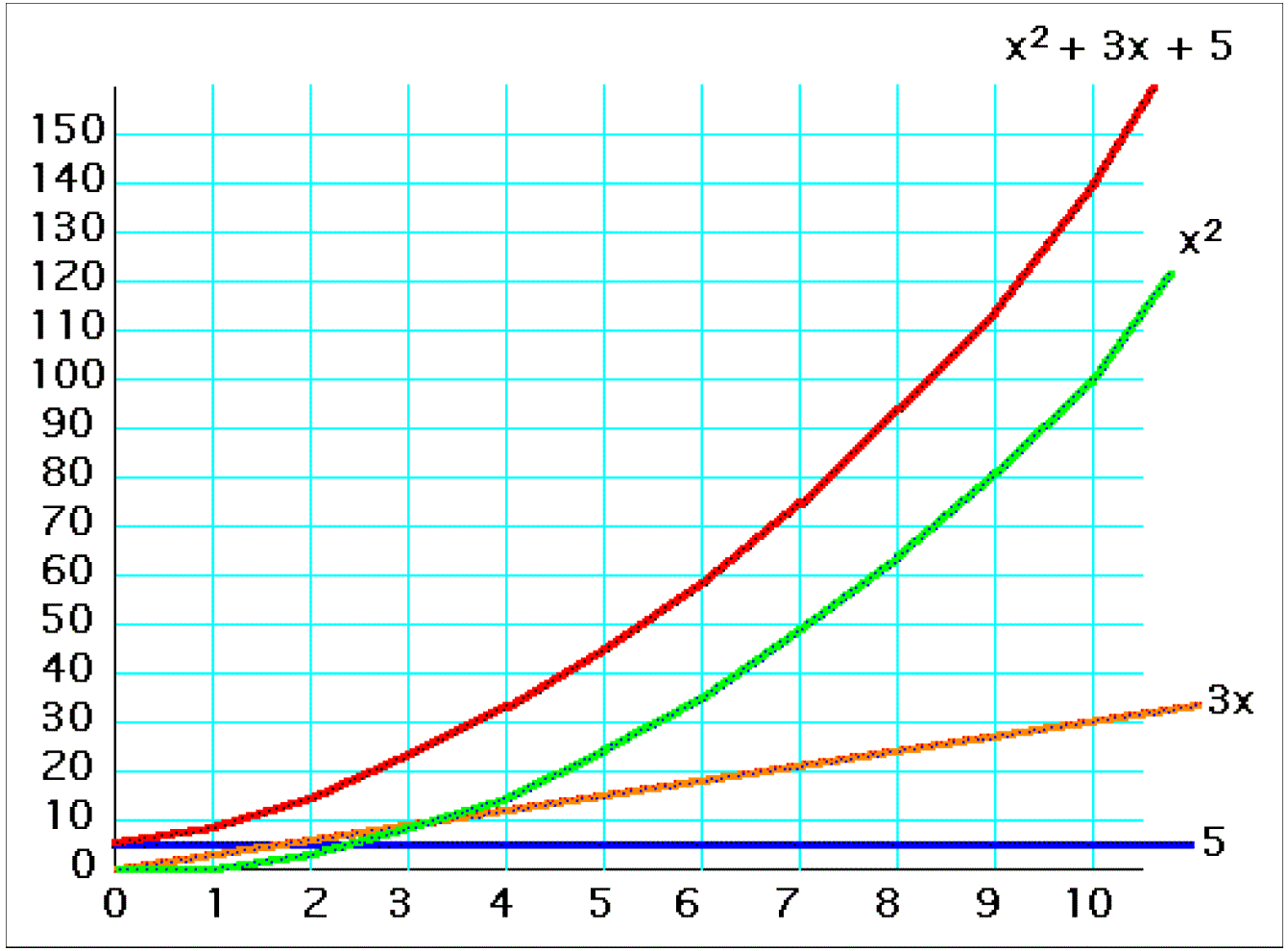
- Big O notation is a *huge* simplification; can we justify it?
  - It only makes sense for *large* problem sizes
  - For sufficiently large problem sizes, the highest-order term swamps all the rest!
- Consider $R = x^2 + 3x + 5$ as x varies:

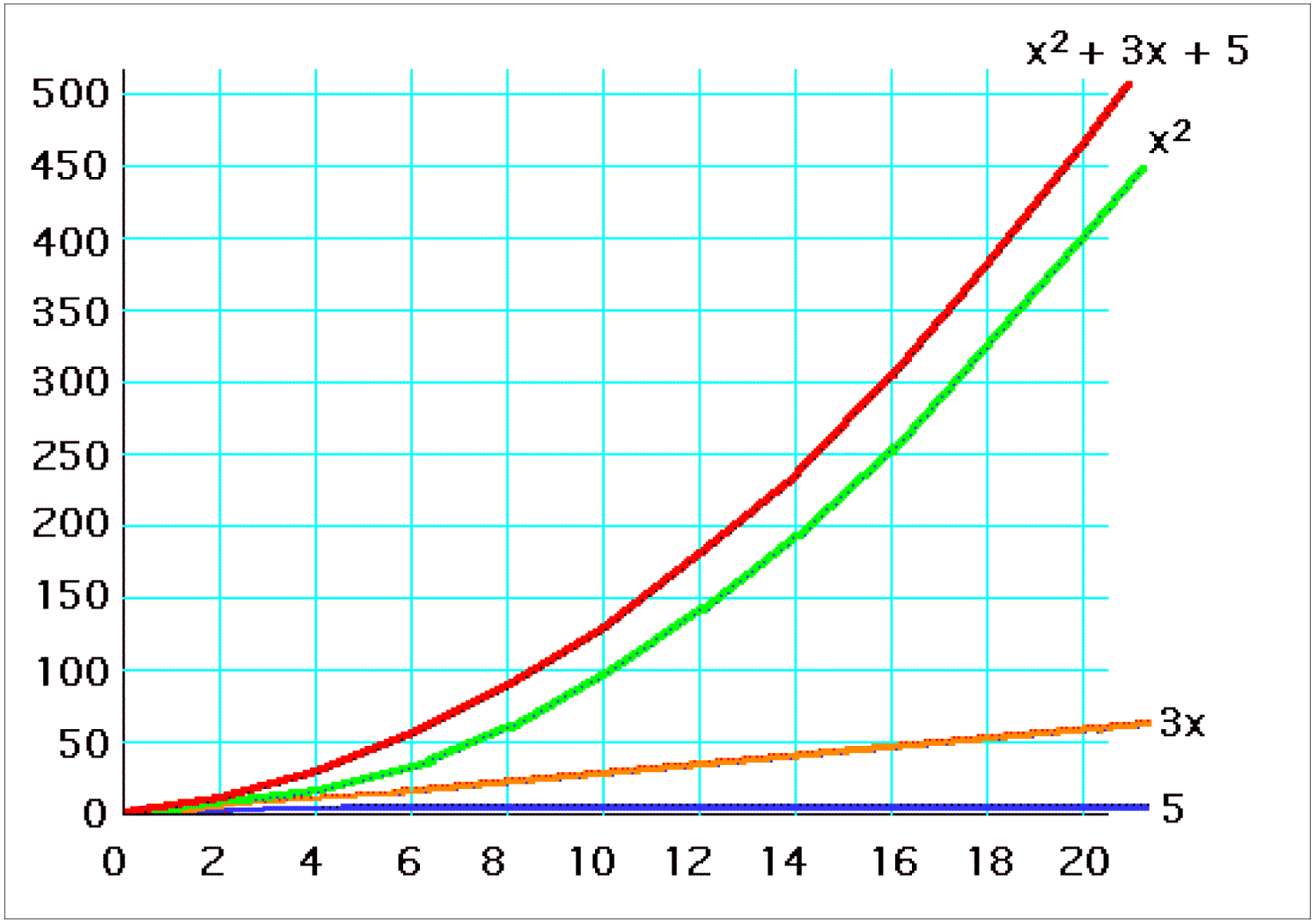
$x = 0$	$x^2 = 0$	$3x = 0$	$5 = 5$	$R = 5$
$x = 10$	$x^2 = 100$	$3x = 30$	$5 = 5$	$R = 135$
$x = 100$	$x^2 = 10000$	$3x = 300$	$5 = 5$	$R = 10{,}305$
$x = 1000$	$x^2 = 1000000$	$3x = 3000$	$5 = 5$	$R = 1{,}003{,}005$
$x = 10{,}000$	$x^2 = 10^8$	$3x = 3*10^4$	$5 = 5$	$R = 100{,}030{,}005$
$x = 100{,}000$	$x^2 = 10^{10}$	$3x = 3*10^5$	$5 = 5$	$R = 10{,}000{,}300{,}005$

Big O: example

- Let $T(n) = 3n^2 + 100n + 5$, then $T(n) = O(n^2)$
- Proof:
  - Let $c = 4$ and $n_0 = 100.05$
  - For $n \geq 100.05$, we have that $4n^2 \geq 3n^2 + 100n + 5$

- $T(n)$ is also $O(n^3)$, $O(n^4)$, etc.
  Typically, the smallest complexity is used.


x² + 3x + 5
x²
3x
5
150
140
130
120
110
100
90
80
70
60
50
40
30
20
10
0
0 1 2 3 4 5 6 7 8 9 10


$x^2 + 3x + 5$
x^2
3x
5
500
450
400
350
300
250
200
150
100
50
0
0
2
4
6
8
10
12
14
16
18
20

Common time complexities

- $O(1)$ constant time
- $O(\log n)$ log time
- $O(n)$ linear time
- $O(n \log n)$ log linear time
- $O(n^2)$ quadratic time
- $O(n^3)$ cubic time
- $O(2^n)$ exponential time
- $O(n!)$ factorial time

Big O: examples

$T(n)$	Complexity
$5n^3 + 200n^2 + 15$	
$3n^2 + 2^{300}$	
$5 \log_2 n + 15 \ln n$	
$2 \log n^3$	
$4n + \log n$	
2^{64}	
$\log n^{10} + 2\sqrt{n}$	
$2^n + n^{1000}$	

Complexity analysis: examples

Let us assume that `f()` has complexity $O(1)$

```
for i in range(n):
        f()
```

$\longrightarrow O(n)$

```
for i in range(n):
    for j in range(n):
                f()
```

$\longrightarrow O(n^2)$

```
for i in range(n):
    for j in range(i+1):
                f()
```

$\longrightarrow O(n^2)$

```
for i in range(n):
    for j in range(n):
         for k in range(n):
                f()
```

$\longrightarrow O(n^3)$

```
for i in range(m):
    for j in range(n):
         for k in range(p):
                f()
```

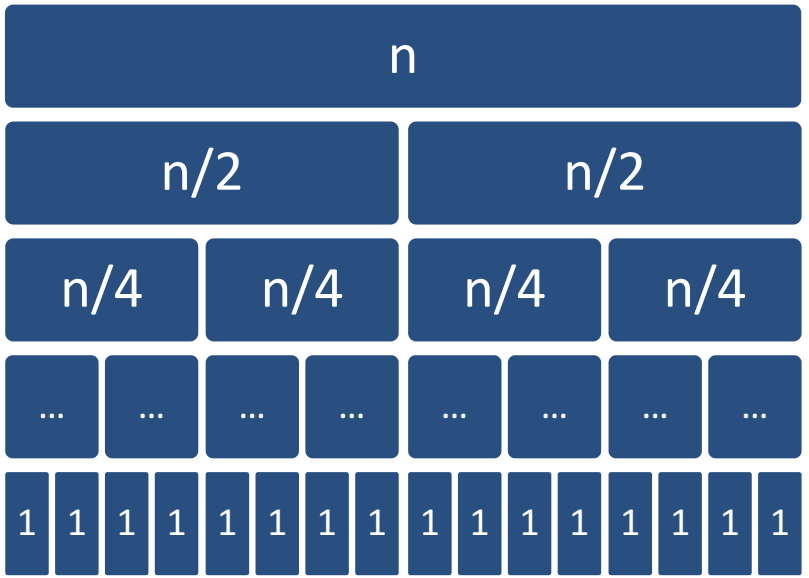
$\longrightarrow O(mnp)$

Warnings about O-Notation

- Big-O notation cannot compare algorithms in the same complexity class.
- Big-O notation only gives sensible comparisons of algorithms in different complexity classes when **n** is large .
- Consider two algorithms for same task:
  Linear: **f(n) = 1000 n**
  Quadratic: **f'(n) = n^2/1000**
  The quadratic one is faster for n < 1000000.

Complexity analysis: recursion

```
def f(n) :
    if (n > 0) :
        DoSomething(n) # O(n)
        f(n/2)
        f(n/2)
```



$$
\begin{aligned}
T(n) &= n + 2 \cdot T(n/2) \\
&= n + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + 8 \cdot \frac{n}{8} + \cdots \\
&= \underbrace{n + n + n + \cdots + n}_{\log_2 n} = n \log_2 n
\end{aligned}
$$

$T(n)$ is $O(n \log n)$

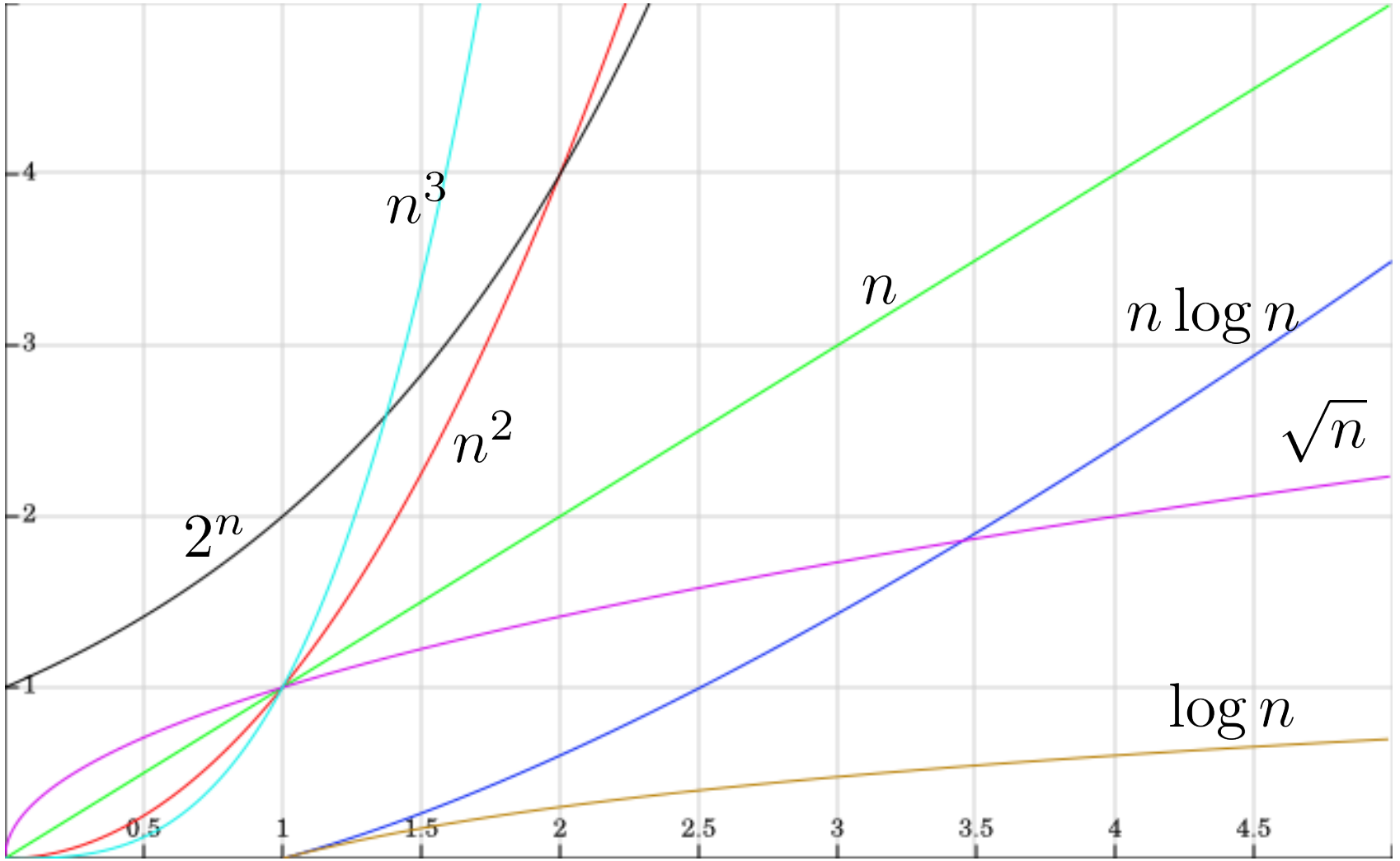
Complexity analysis: recursion

```
def f(n) :
    if (n > 0) :
        DoSomething(n) # O(n)
        f(n-1)
```

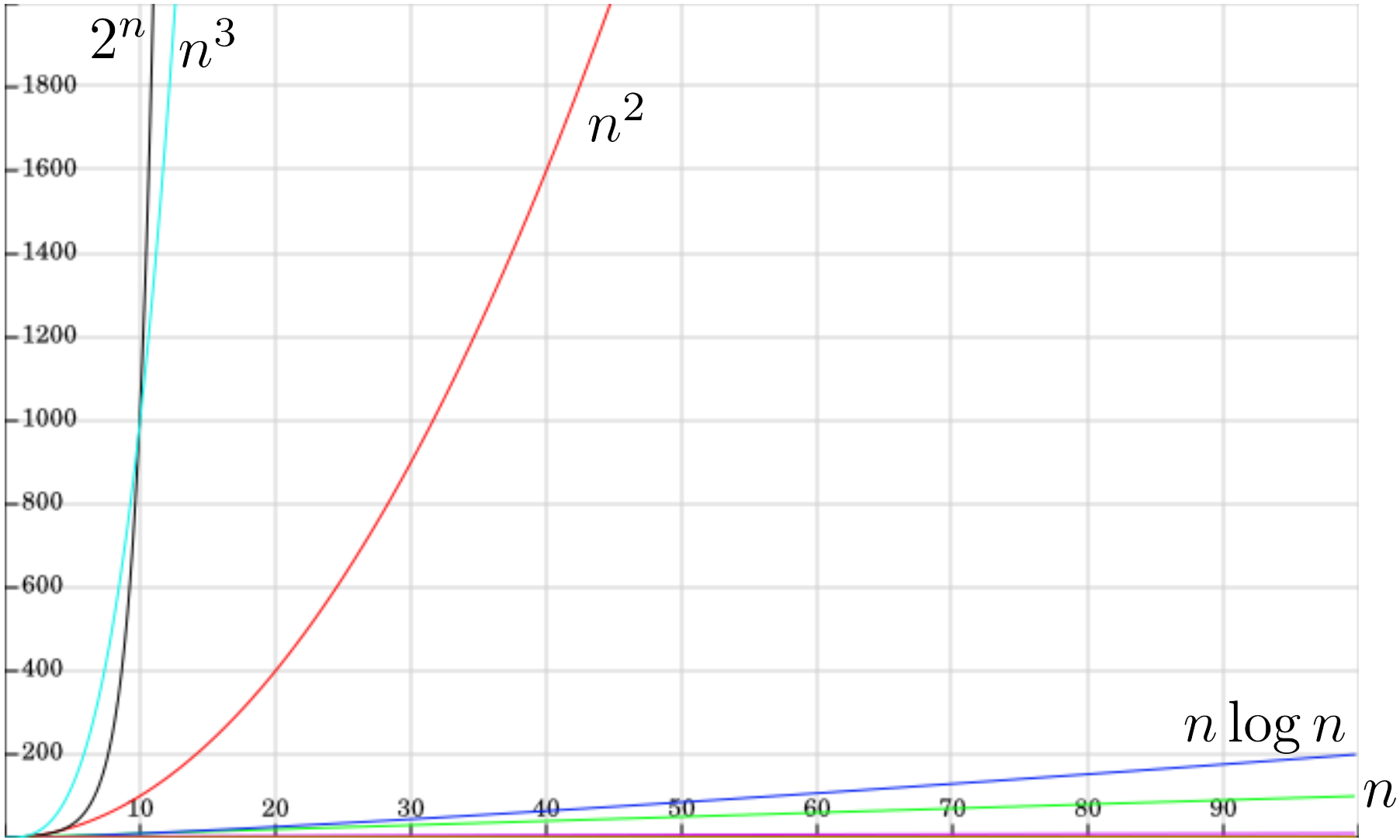
$$
\begin{aligned}
T(n) &= n + T(n-1) \\
T(n) &= n + (n-1) + (n-2) + \cdots + 2 + 1 \\
T(n) &= \frac{n^2 + n}{2}
\end{aligned}
$$

$$T(n) \text{ is } O(n^2)$$

Asymptotic complexity (small values)



Asymptotic complexity (larger values)



Execution time: example

Let us consider that every operation can be executed in 1 ns (10^{-9} s)

	Time		
Function	$(n = 10^3)$	$(n = 10^4)$	$(n = 10^5)$
$\log_2 n$	10 ns	13.3 ns	16.6 ns
$\sqrt{n}$	31.6 ns	100 ns	316 ns
n	1 μs	10 μs	100 μs
$n \log_2 n$	10 μs	133 μs	1.7 ms
n^2	1 ms	100 ms	10 s
n^3	1 s	16.7 min	11.6 days
n^4	16.7 min	116 days	3171 yr
2^n	$3.4 \cdot 10^{284}$ yr	$6.3 \cdot 10^{2993}$ yr	$3.2 \cdot 10^{30086}$ yr

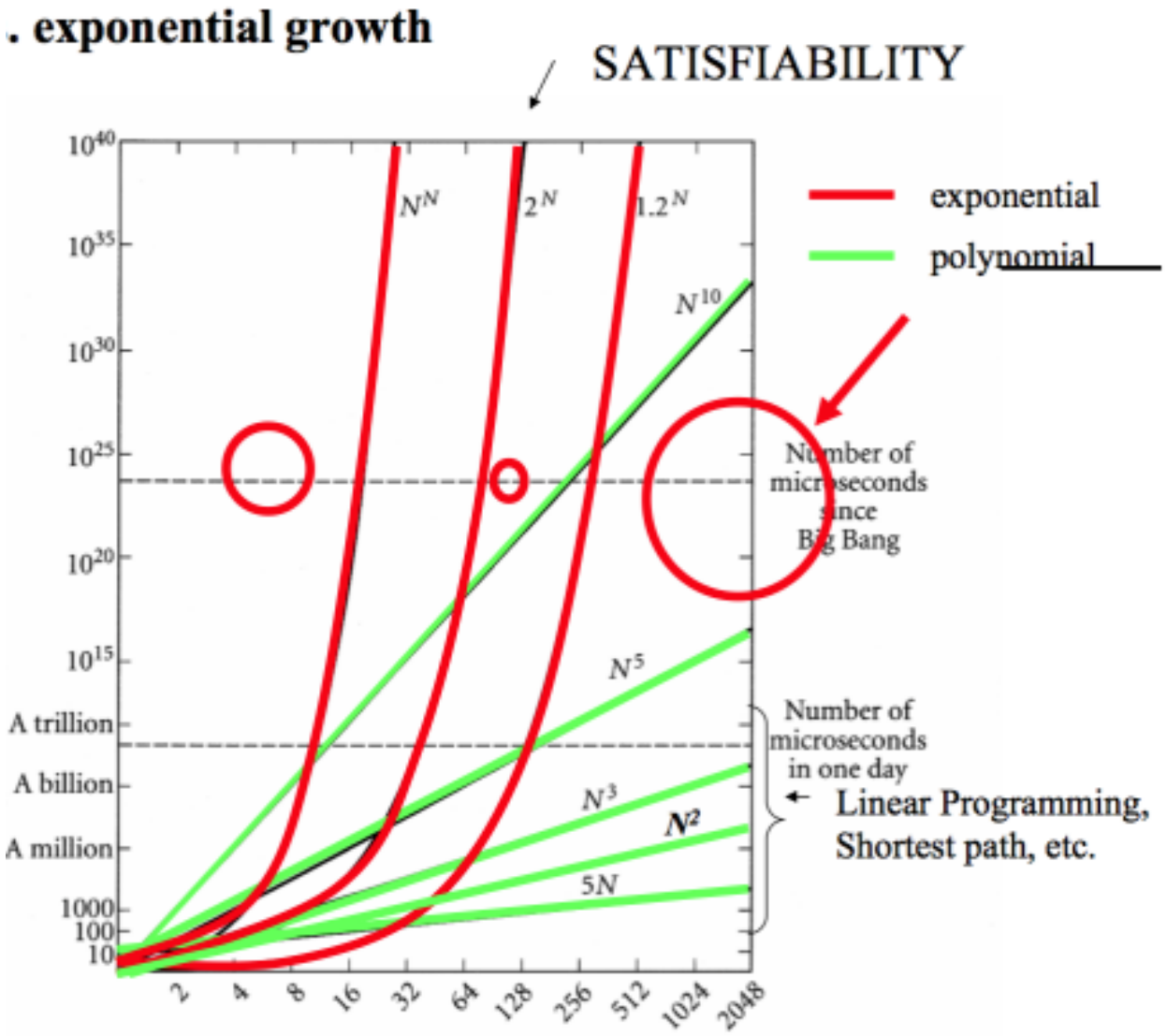
How about “big data”?

Source: Jon Kleinberg and Éva Tardos, Algorithm Design, Addison Wesley 2006.

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Polynomial vs. Exponential growth



“Relatives” of Big-Oh

- “Relatives” of the Big-Oh
  - Ω (f(n)): Big Omega – asymptotic *lower* bound
  - Θ (f(n)): Big Theta – asymptotic *tight* bound
- Big-Omega – think of it as the inverse of O(n)
  - g(n) is Ω (f(n)) if f(n) is O(g(n))
- Big-Theta – combine both Big-Oh and Big-Omega
  - f(n) is Θ (g(n)) if f(n) is O(g(n)) and g(n) is Ω (f(n))
- Make the difference:
  - 3n+3 is O(n) and is Θ (n)
  - 3n+3 is O(n^2) but is not Θ (n^2)

Summary

- Complexity analysis is a technique to analyze and compare algorithms (not programs).
- It helps to have preliminary back-of-the-envelope estimations of runtime (milliseconds, seconds, minutes, days, years?).
- Worst-case analysis is sometimes overly pessimistic. Average case is also interesting (not covered in this course).
- In many application domains (e.g., big data) quadratic complexity, $O(n^2)$, is not acceptable.
- Recommendation: avoid last-minute surprises by doing complexity analysis before writing code.

Two key problems in computer science

- The Boolean Satisfiability Problem (SAT)
- Traveling Salesman Problem

The Boolean Satisfiability Problem SAT

SAT

- Definition of the **satisfiability problem:** Given a Boolean formula, determine whether this formula is satisfiable or not.

- A ***literal***: x_i or $\neg x_i$
- A ***clause***: $x_1 \vee x_2 \vee \neg x_3 \equiv C_i$
- A ***formula***: in *conjunctive normal form* (CNF)

$$C_1 \& C_2 \& \ldots \& C_m$$

Goal: Find an assignment of True/False for all the (Boolean) variables s.t. all clauses are True

The Satisfiability Problem

The ***satisfiability*** problem

A logical formula:

$$x_1 \vee x_2 \vee x_3$$
$$\& \ \neg x_1$$
$$\& \ \neg x_2$$

the assignment :

$$x_1 \leftarrow F \ , \ x_2 \leftarrow F \ , \ x_3 \leftarrow T$$

will make the above formula true.

$(\neg x_1, \ \neg x_2 , x_3)$ represents $x_1 \leftarrow F \ , \ x_2 \leftarrow F \ , \ x_3 \leftarrow T$

SAT

- If there is ***at least one*** assignment which satisfies a formula, then we say that this formula is ***satisfiable***; otherwise, it is ***unsatisfiable***.
- An unsatisfiable formula:

$$x_1 \vee x_2$$

$$\& \; x_1 \vee \neg x_2$$

$$\& \; \neg x_1 \vee x_2$$

$$\& \; \neg x_1 \vee \neg x_2$$

SAT Example: Timetabling

Sample SAT encoding (more details in paper: Achá, Roberto Asín, and Robert Nieuwenhuis. "Curriculum-based course timetabling with SAT and MaxSAT." *Annals of Operations Research* 218, no. 1 (2014): 71-91.)

- Boolean variable for every class,hour pair $X_{c,h}$
- Boolean variable for every class,room pair $X_{c,r}$
- For every class, will have a clause enforcing that at least 1 of the $X_{c,h}$ variables is true and another set of clauses which together enforce that at most 1 of the $X_{c,h}$ variables is true
- For every pair of classes c,c' that share a student or a lecturer, we have a clause for every hour ($\neg X_{c,h}$ v $\neg X_{c',h}$) and similarly for every room. These clauses enforce that at least that both can't take the value True for the same hour/room.
- Room capacity is enforced simply by adding a clause with one literal ($\neg X_{c,r}$) for every room with less capacity than the size of class *c*

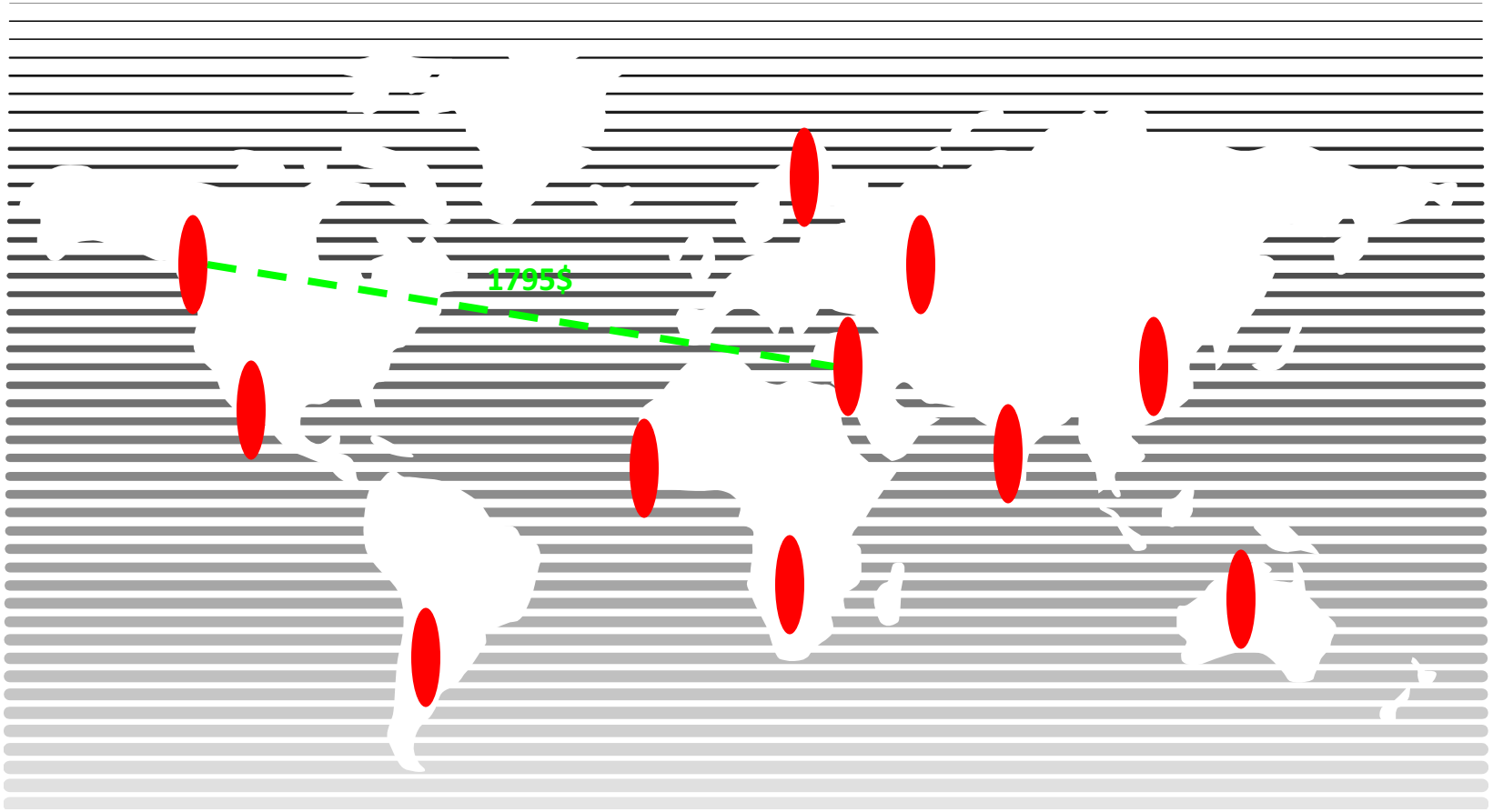
Traveling Salesman Person

TSP

The Mission: A Tour Around the World

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?

The Problem: Traveling Costs Money



TSP Example

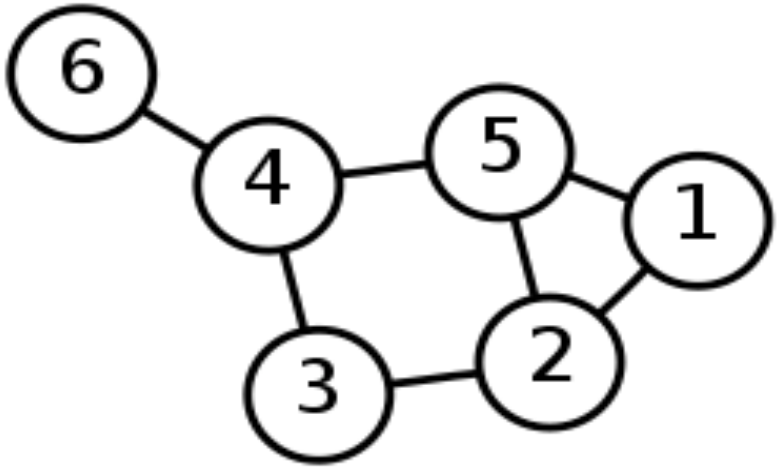
City Id	X	Y
1	100	100
2	20	20
3	10	10
4	0	0

Euclidean Distance: sqrt((x_i – x_j)^2 + (y_i – y_j)^2)

Optimal?
1, 2, 4, 3
Alternative?
3, 4, 2, 1
Suboptimal Solution?
3, 2, 4, 1

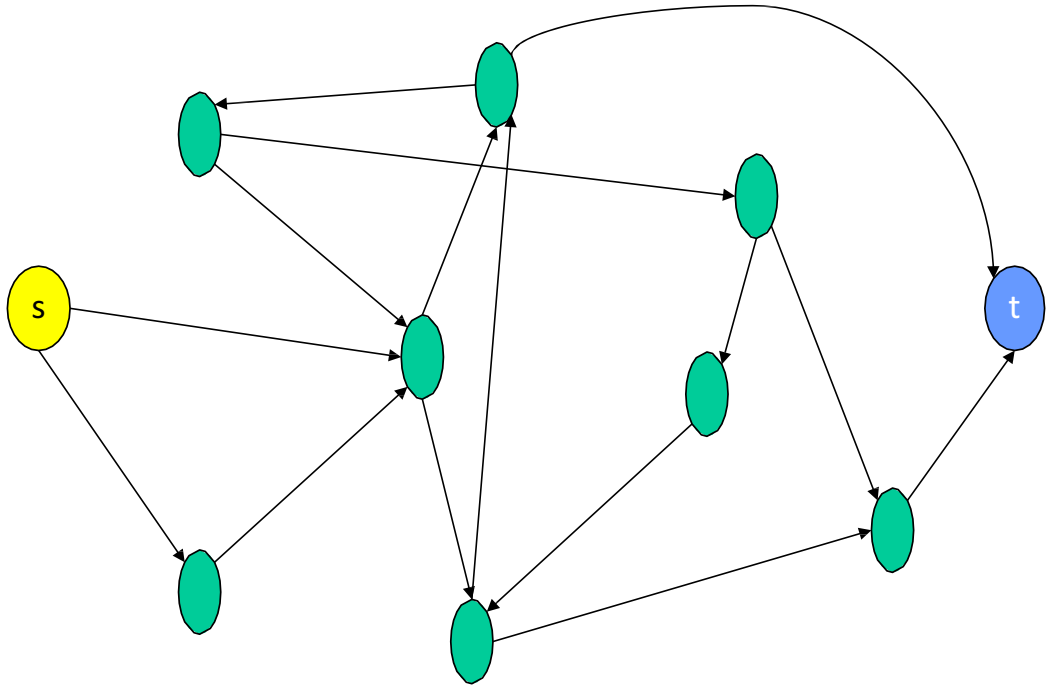
Graphs

*"a **graph** is a structure amounting to a set of objects in which some pairs of the objects are in some sense "related". The objects correspond to mathematical abstractions called vertices (also called nodes or points) and each of the related pairs of vertices is called an edge(also called an arc or line).*

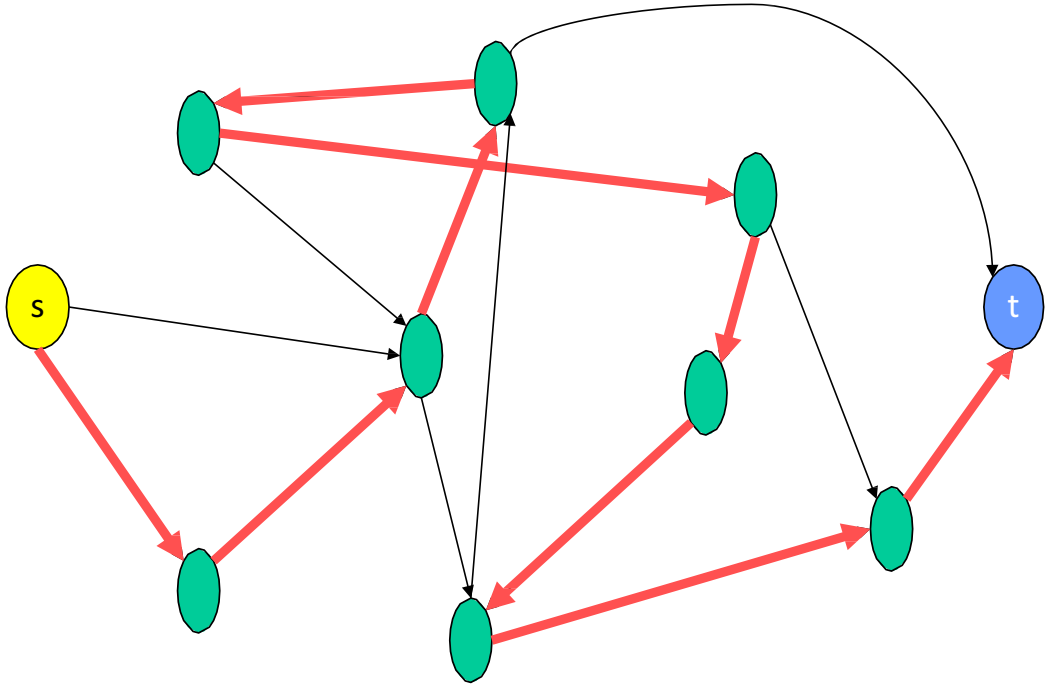


Hamiltonian Path

- **Instance**: a directed Graph. G = (V, E) and two vertices s, t ∈ V
- **Problem**: to decide if there exists a path from **s** to **t**, which goes through each node once



Can You Find One Here?


s
t

SAT vs. Hamiltonian Path vs. TSP

- Finding a solution
- Finding the optimal solution

Polynomial Time Algorithms

- Most of the algorithms we have mentioned so far run in time that is upper bounded by a polynomial in the input size
  - sorting: $O(n^2)$, $O(n \log n)$, …
  - matrix multiplication: $O(n^3)$, $O(n^{\log_2 7})$
  - graph algorithms: $O(V+E)$, $O(E \log V)$, …
- In fact, the running time of these algorithms are bounded by small polynomials.

Classifying Problems

Coarse categorization of problems:

1. Problems solvable in polynomial time (i.e., there exists an algorithm solving this problem in polynomial time). We will consider these problems "tractable“.
2. Those not solvable in polynomial time

We shall denote by P the class of all polynomial time solvable ***decision*** problems.

Decision Problems and the class P

A computational problem with yes/no answer is called a decision problem.

We shall denote by P the class of all decision problems that are solvable in polynomial time.

Why Polynomial Time?

- It is convenient to define decision problems to be tractable if they belong to the class P, since:
  - the class P is closed under composition.
  - the class P becomes more or less independent of the computational model.

[Typically, computational models can be transformed into each other by polynomial time reductions.]

The Class NP

NP is the class of all decision problems for which a candidate solution can be **verified** in polynomial time.

- The class P is a subset of NP.
- NP is short for ***nondeterministic*** polynomial time (since these problems can be solved on a nondeterministic Turing machine in polynomial time, as you will see later).
- NP does not stand for not-P! Why?

Verifying a Candidate Solution

- Difference between solving a problem and verifying a candidate solution:
- Solving a problem: is there a path in graph G from node u to node v with at most k edges?
- Verifying a candidate solution: is $v_0, v_1, \ldots, v_\ell$ a path in graph G from node u to node v with at most k edges?

Verifying a Candidate Solution

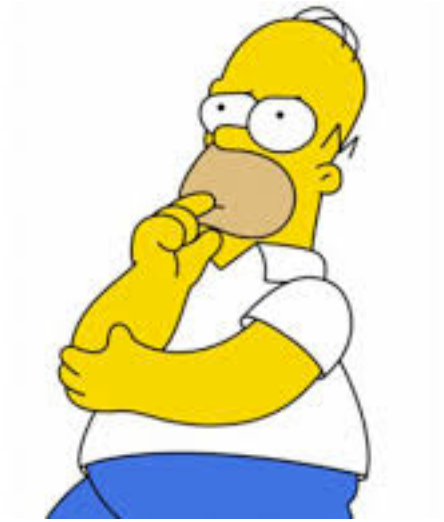
- A Hamiltonian cycle in an undirected graph is a cycle that visits every node exactly once.
- Solving a problem: Is there a Hamiltonian cycle in graph G?
- Verifying a candidate solution: Is $v_0, v_1, \ldots, v_\ell$ a Hamiltonian cycle of graph G?

Verifying a Candidate Solution vs. Solving a Problem

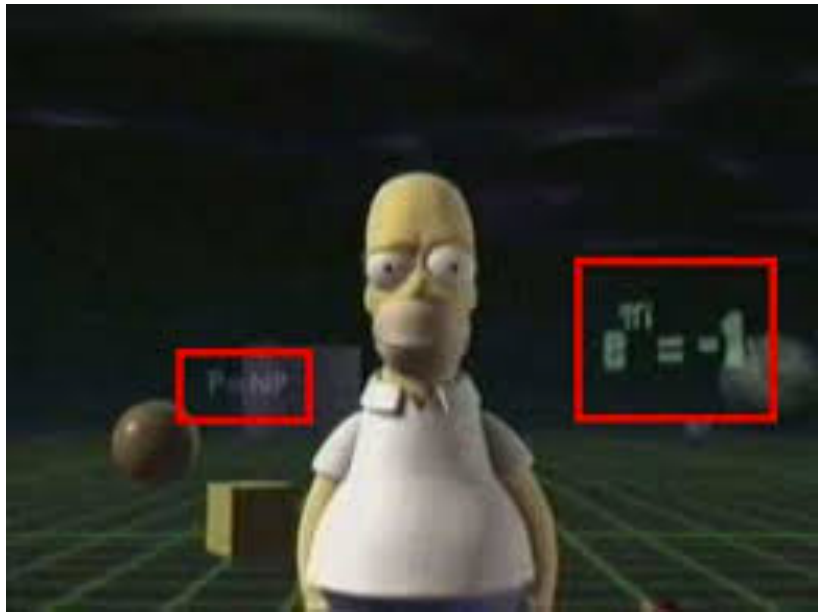
- Intuitively it seems much harder (more time consuming) in some cases to solve a problem from scratch than to verify that a candidate solution actually solves the problem.
- If there are many candidate solutions to check, then even if each individual one is quick to check, overall it can take a long time

Verifying a Candidate Solution

- Many practical problems in computer science, math, operations research, engineering, etc. are **poly-time** verifiable but have no known **poly-time** algorithm
  - Wikipedia lists problems in computational geometry, graph theory, network design, scheduling, databases, program optimization and more



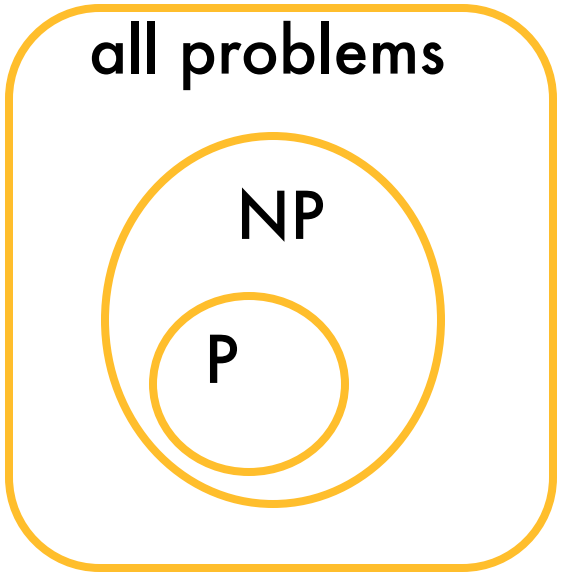
P versus NP



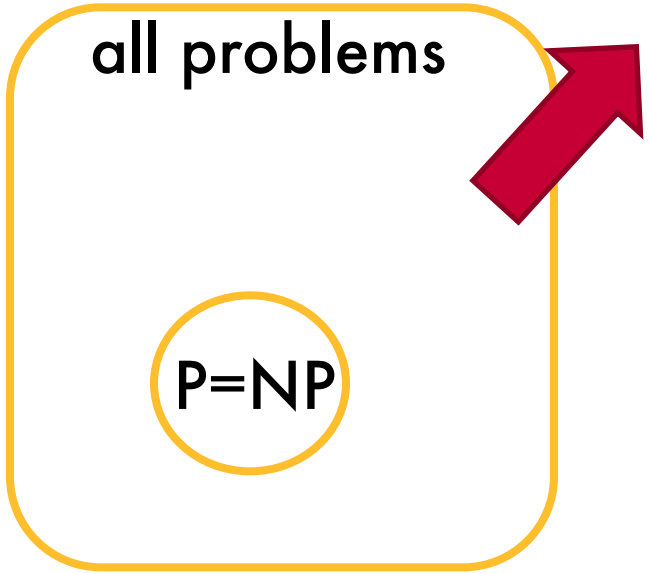
P vs. NP

- Although poly time verifiability ***seems*** like a weaker condition than poly time solvability, no one has been able to prove that it is weaker (i.e., describes a larger class of problems)
- So it is unknown whether P = NP.

P and NP



or



Current belief

Difficult, but still possible (unlikely)

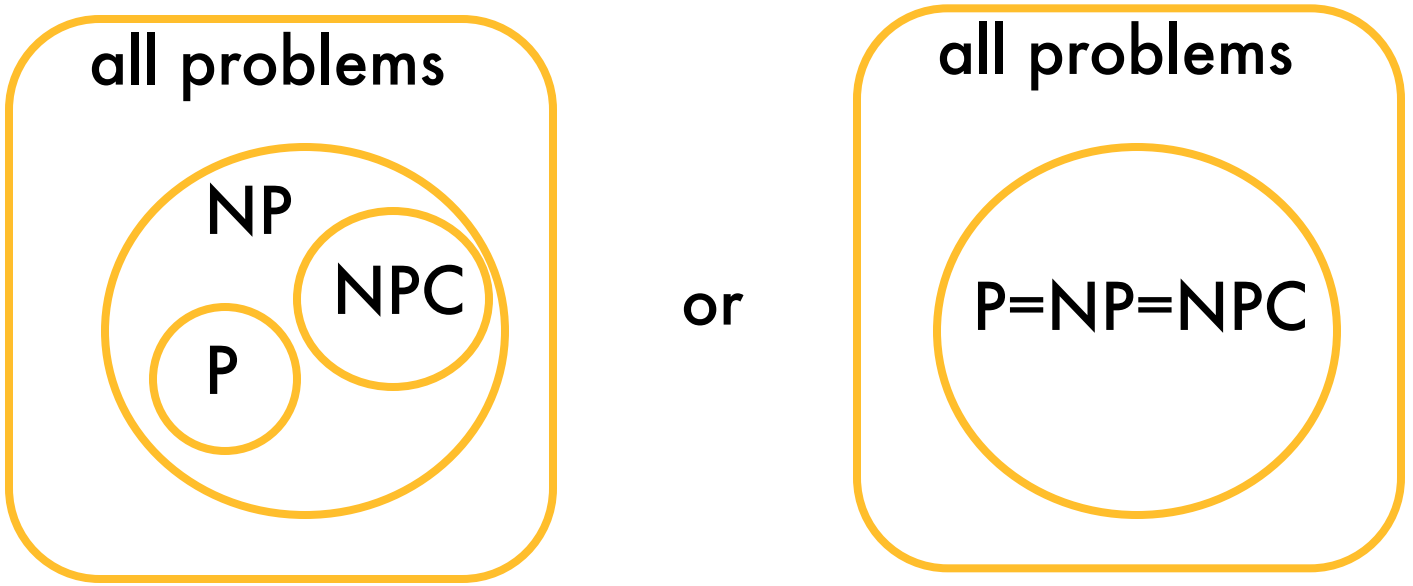
NP-Complete Problems

- NP-complete problems is class of ***hardest*** problems in NP.
- If an NP-complete problem can be solved in polynomial time, then all problems in NP can be, and thus P = NP.
- Small subset of problems; the general idea is that if we can reduce a problem X to a problem Y, and problem Y is NP-complete then so must X be

TSP?

- The TSP is not in NP since an (optimal) solution (the shortest route through all cities) is not verifiable in polynomial time
  - (TSP is an *NP-Hard* problem)
  - The decision variant of the problem is NP-complete: given a limit X is there a solution that is less than this limit, is there a route whose distance is less than this value

Possible Worlds

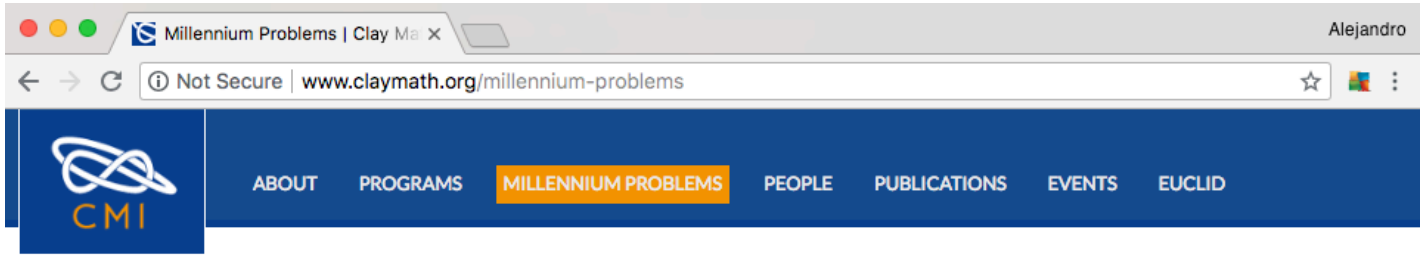


NPC = NP-complete

P = NP Question

- Open question since about 1970
- Great theoretical interest
- Great practical importance:
  - If your problem is NP-complete, then don't waste time looking for an efficient algorithm
  - Instead look for efficient approximations, heuristics, etc.
- Solve for a million dollars!
  - http://www.claymath.org/millennium-problems
  - The Poincare conjecture is solved today

Million dollar problem



Millennium Problems

Yang–Mills and Mass Gap

Experiment and computer simulations suggest the existence of a "mass gap" in the solution to the quantum versions of the Yang-Mills equations. But no proof of this property is known.

Riemann Hypothesis

The prime number theorem determines the average distribution of the primes. The Riemann hypothesis tells us about the deviation from the average. Formulated in Riemann's 1859 paper, it asserts that all the non-physical zeros of the zeta function are complex numbers with real part 1/2.

P vs NP Problem

If it is easy to check that a solution to a problem is correct, is it also easy to solve the problem? This is the essence of the P vs NP question. Typical of the NP problems is that of the Hamiltonian Path Problem: given N cities to visit, how can one do this without visiting a city twice? If you give me a solution, I can easily check that it is correct. But I cannot so easily find a solution.

Navier–Stokes Equation

This is the equation which governs the flow of fluids such as water and air. However, there is no proof for the most basic questions one can ask: do solutions exist, and are they unique? Why ask for a proof? Because a proof gives not only certitude, but also understanding.

Decision Problems

NP-Completeness Theory

As we have already mentioned, the theory is based considering decision problems.

Example:

- Does there exist a path from node u to node v in graph G with at most k edges.
- Instead of: What is the length of the shortest path from u to v? Or even: What is the shortest path from u to v?

Decision Problems

Why focus on decision problems?

- Solving the general problem is at least as hard as solving the decision problem version
- For many natural problems, we only need polynomial additional time to solve the general problem if we already have a solution to the decision problem

Definition of P

- P is the set of all decision problems that can be computed in time $O(n^k)$, where n is the length of the input string and k is a constant
- "Computed" means there is an algorithm that correctly returns YES or NO whether the input string is in the language

Example of a Decision Problem in P

- Given a graph G, nodes u and v, and integer k, is there a path in G from u to v with at most k edges?
- Why is this a decision problem?
  - Has YES/NO answers
- We are glossing over the particular encoding (tedious but straightforward)
- Why is this problem in P?
  - Do BFS on G in polynomial time

Definition of NP

- NP = set of all decision problems for which a candidate solution can be verified in polynomial time
- Does *not* stand for "not polynomial"
  - in fact P is a subset of NP
- NP stands for "nondeterministic polynomial"
  - more info on this in CPSC 433

Example of a Decision Problem in NP

- Decision problem: Is there a path in G from u to v of length at most k?
- Candidate solution: a sequence of nodes $v_0, v_1, \ldots, v_\ell$
- To verify:
  - check if $\ell \leq k$
  - check if $v_0 = u$ and $v_\ell = v$
  - check if each (v_i, v_{i+1}) is an edge of G

Example of a Decision Problem in NP

- Decision problem: Does G have a Hamiltonian cycle?
- Candidate solution: a sequence of nodes $v_0, v_1, \ldots, v_\ell$
- To verify:
  - check if ℓ = number of nodes in G
  - check if $v_0 = v_\ell$ and there are no repeats in $v_0, v_1, \ldots, v_{\ell-1}$
  - check if each (v_i, v_{i+1}) is an edge of G

Going From Verifying to Solving

for each candidate solution do
    verify if the candidate really works
    if so then return YES
return NO

Difficult to use in practice, though, if number of candidate solutions is large

Number of Candidate Solutions

- *"**Is there a path from u to v in G of length at most k?**"*: more than n! candidate solutions where n is the number of nodes in G
- *"**Does G have a Hamiltonian cycle?**"*: n! candidate solutions

Polynomial Reduction

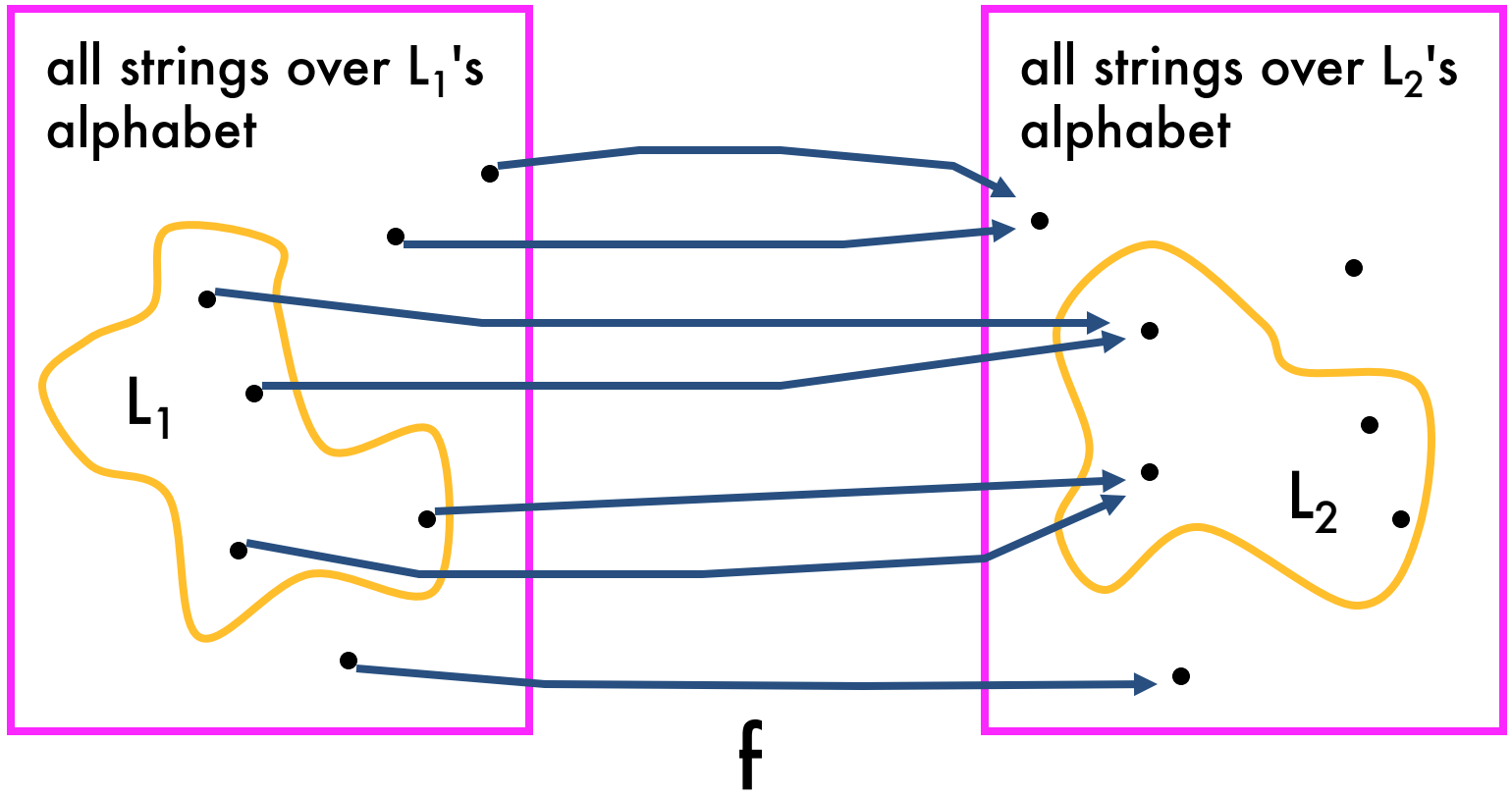
Polynomial Reduction

A ***polynomial reduction*** (or ***transformation***) from language L_1 to language L_2 is a function f from strings over L_1's alphabet to strings over L_2's alphabet such that

(1) f is computable in polynomial time

(2) for all x, x is in L_1 if and only if f(x) is in L_2

Polynomial Reduction



Polynomial Reduction

- YES instances map to YES instances
- NO instances map to NO instances
- computable in polynomial time
- Notation: $L_1 \leq_p L_2$

  Think of it as: L_2 is at least as hard as L_1

Polynomial Reduction Theorem

Theorem: **If** $L_1 \leq_p L_2$ **and** L_2 is in P,

then L_1 is in P.

Proof: Let A_2 be a polynomial time algorithm for L_2. Here is a polynomial time algorithm A_1 for L_1.

input: x

compute f(x)

run A_2 on input f(x)

return whatever A_2 returns

$|x| = n$

takes p(n) time

takes q(p(n)) time

takes O(1) time

Implications

- Suppose that $L_1 \leq_p L_2$
- If there is a polynomial time algorithm for L_2, then there is a polynomial time algorithm for L_1.
- If there is no polynomial time algorithm for L_1, then there is no polynomial time algorithm for L_2.
- Note the asymmetry!

NP-Completeness

Definition of NP-Complete

L is NP-complete if and only if

(1) L is in NP and

(2) for all L' in NP, $L' \leq_p L$.

In other words, L is at least as hard as every language in NP.

Implication of NP-Completeness

Theorem: Suppose L is NP-complete.

(a) If there is a poly time algorithm for L, then P = NP.

(b) If there is no poly time algorithm for L, then there is no poly time algorithm for any NP-complete language.

Showing NP-Completeness

- How to show that a problem (language) L is NP-complete?
- Direct approach: Show

  (1) L is in NP

  (2) every other language in NP is polynomially reducible to L.
- Better approach: once we know some NP-complete problems, we can use reduction to show other problems are also NP-complete. How?

Showing NP-Completeness with a Reduction

To show L is NP-complete:

(1) Show L is in NP.

(2.a) Choose an appropriate known NP-complete language L'.

(2.b) Show $L' \leq_p L$.

Why does this work? By transitivity: Since every language L'' in NP is polynomially reducible to L', L'' is also polynomially reducible to L.