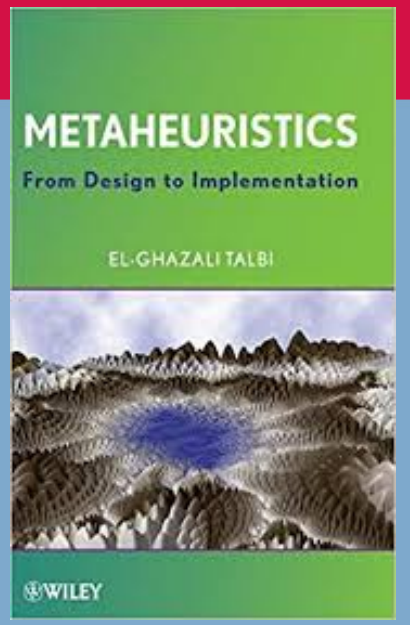
CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

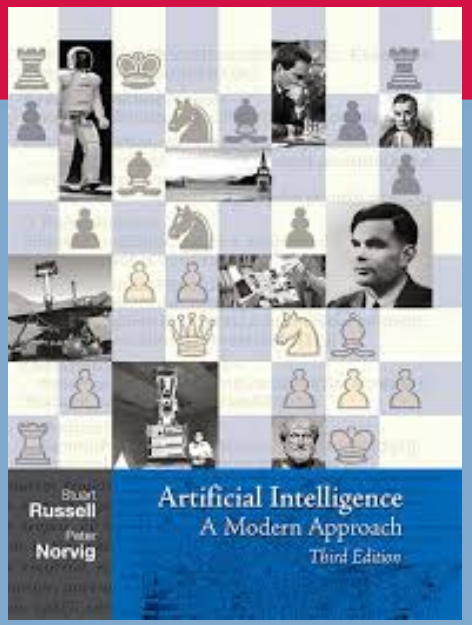
Computing@CIT

Metaheuristic Optimization

Introduction

Dr. Diarmuid Grimes





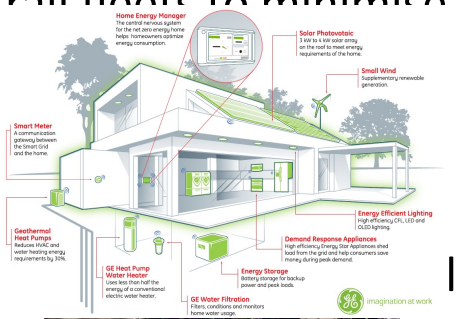
Personal and Research Background

Diarmuid.Grimes@cit.ie

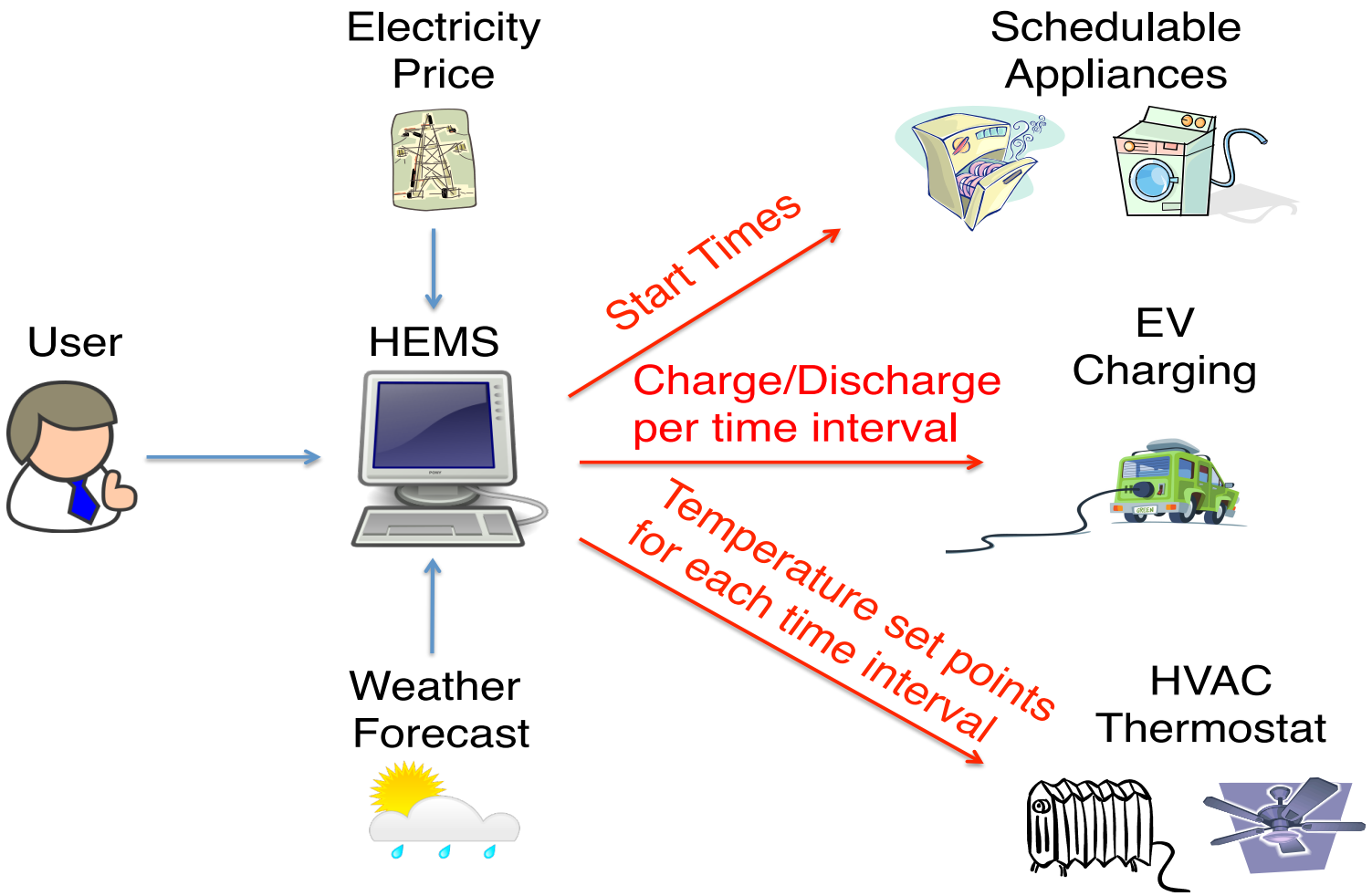
B223L

Ext: 5506

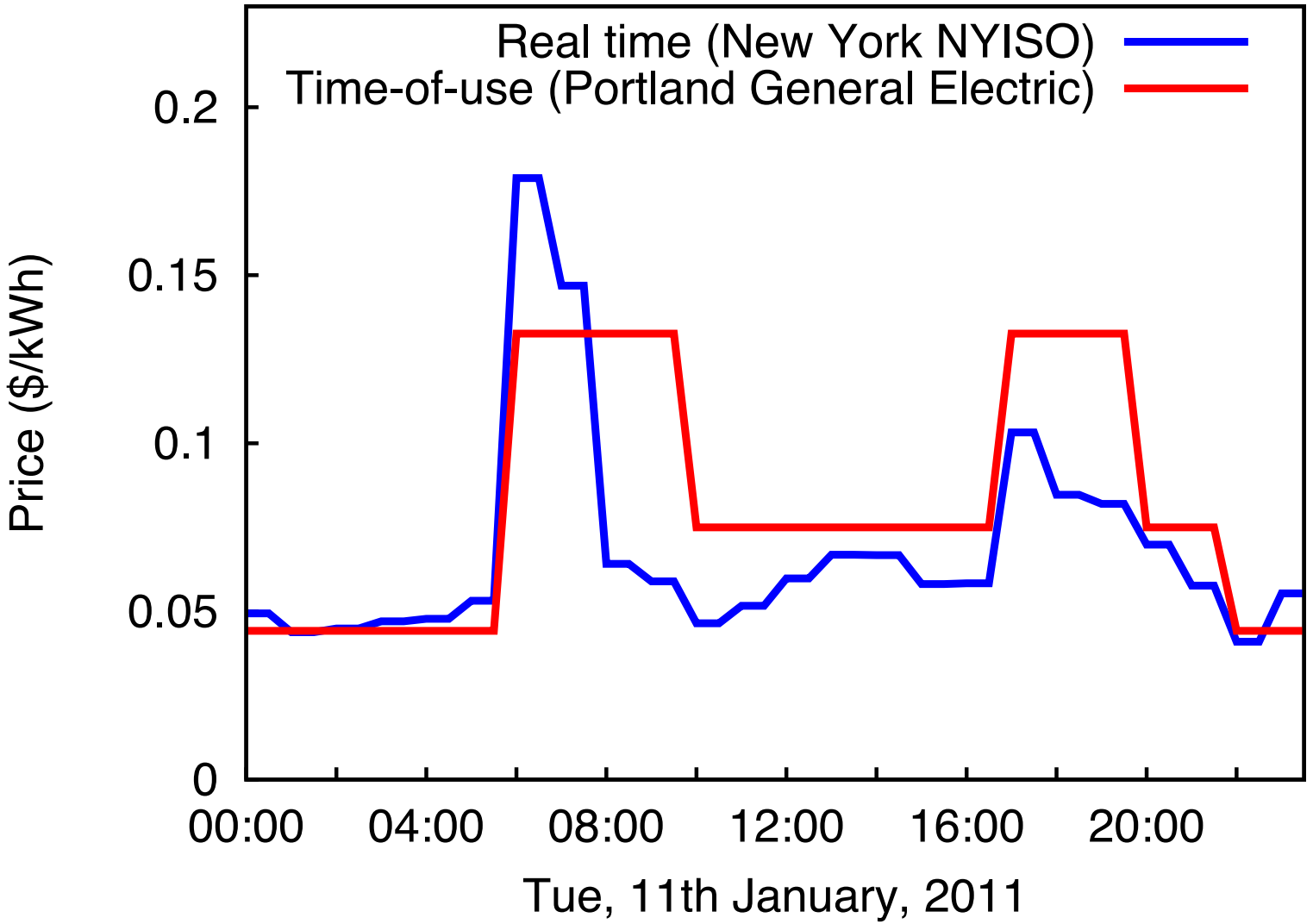
- Combinatorial Optimisation
  - Generic approaches for CP, LNS, Genetic Algorithms
  - Dedicated methods for different application areas:
    - Energy (minimize consumption/costs in residential / industrial sectors)
    - Transport (schedule maintenance for rail fleets to minimise cost function, schedule timetable of urban electric rail fleet to minimise peak usa
- Machine Learning
  - Application areas:
    - Energy (real time electricity price pre l c r demand and renewable supply)
    - Transport (predicted health of rail-fleet as axle bearings using condition monitoring)
  - One of key findings: ”best” predictor according to ML metrics not best predictor for optimisation



Home Energy Management System



Time-Variable Pricing Tariffs



Home Energy Management Problem

- Schedulable appliances: Start time
- For each time interval:
  - EV: Charge/discharge power → battery state
  - HVAC: Heating/cooling power → temperature

“Best” ML predictor for Optimization

- Standard price forecasting metrics did not result in expected results, i.e. best forecast according to standard metrics such as MSE/MAE did not result in best performance when used for energy aware scheduling!

Why?!!!! – example with 3 tasks of 1kw, 10kw, and 100kw, each lasting 30 mins.

- Price forecast 1 for the next 3 time periods is €1, €2, €3
- Price forecast 2 for the next 3 time periods is €1000, €100, €10
- Actual price for the next 3 time periods is €3, €2, €1

Which is best forecast, F1 or F2?

- According to MAE/MSE F1 is **much better**

But …

- If F1 used: €3x100+€2x10+€1x1=€321
- If F2 used: €3x1+€2x10+€1x100=€123

Real life deployment needed for finetuning!

- Deployed in a home using a time-of-use price tariff
- User thought there was an issue as the home wasn't as warm on Sunday mornings
  - Issue wasn't in the algorithm/code but that price tariff for Sunday was flat so did not preheat home in cheaper night time as on other days
- More importantly, assumption had been made!
  - During night time temperature bounds were relaxed, but still constraint was added to ensure it didn't get too cold
  - Did not consider that it could get too hot!
  - User was waking in the middle of the night in sauna-like conditions
  - Again issue wasn't in the algorithm/code, but in the settings.
    - Price increased by so much at 5am that optimal according to the settings was to really heat the house at 2am and allow heat loss to reduce temperature to required setting at 7am!

Acknowledgments

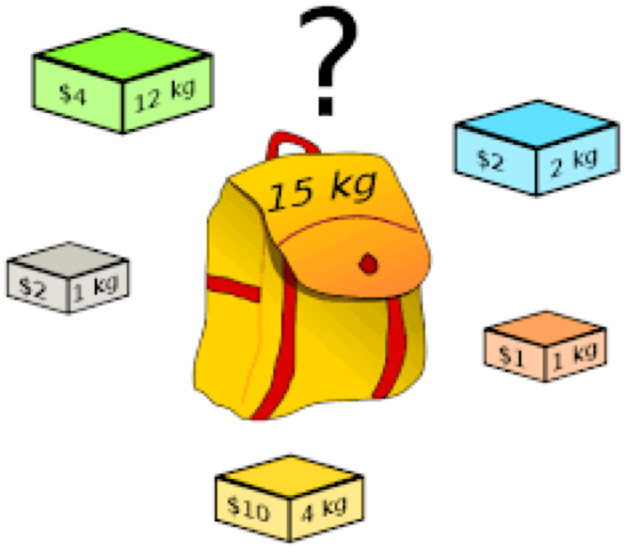
I am grateful to Dr. Alejandro Arbelaez who shared with me the following material.

Module Intro

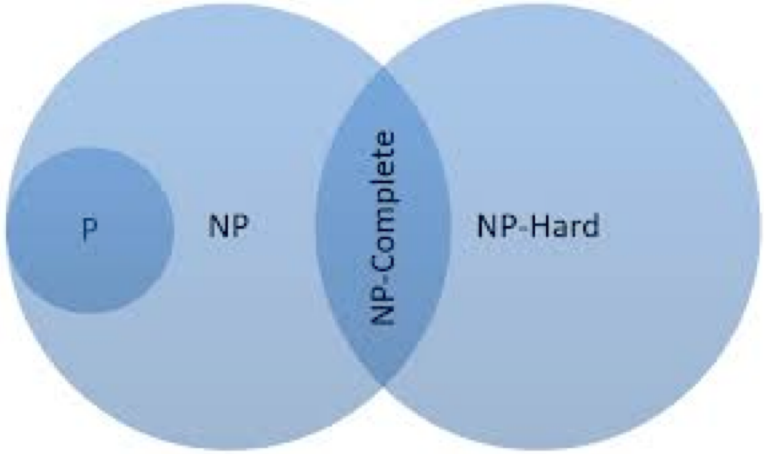
Learning Outcomes

- Module Descriptor

https://courses.cit.ie/index.cfm/page/module/moduleId/13443



Combinatorial Optimization



Complexity

Learning Outcomes

- Module Descriptor

https://courses.cit.ie/index.cfm/page/module/moduleId/13443

- Categorize a **real-life problem** with respect to its **computational complexity**
- Assess the **benefits** and **limitations** of **meta-heuristics** to solve NP-hard problems
- Solve an NP-hard problem with **meta-heuristics** to find a satisfactory lower-bound solution
- Analyze the average **performance** of a randomised algorithm to solve an NP-hard problem
- Apply **nature-inspired** and **local search** meta-heuristics to solve real-life problems

Main Components

- Complexity
- Combinatorial Problems
- Local Search algorithms
- Nature inspired algorithms

Content

- Introduction
- NP-completeness
- Population-based Meta-heuristics
  - Genetic Algorithms
  - Particle Swarm optimization
  - Ant-Colony Optimization
- Single-solution based Meta-heuristics
  - Local Search
  - Simulated Annealing
  - Etc..

Programming Language

- This module is designed for students with prior programming experience (and basic probability & statistics knowledge)

- Programming Language → Python

Assessment Breakdown

- Week 6 (50%)
- Week 12 (50%)

Module workload

2 hours lecture every week

2 hours lab

3 hours Independent studies

Social Golfer Problem

- Groups to work together
  - Discuss weeks lectures
  - Discuss reading paper
  - Peer review code
  - Get to know each other a little!
  - Groups will change weekly

https://www.belfasttelegraph.co.uk/sport/golf/booking-a-tee-time-is-like-getting-tickets-for-the-rolling-stones-how-golfs-return-blazed-a-trail-for-sport-in-new-world-39222396.html

Combinatorial Problems

Combinatorial Problems

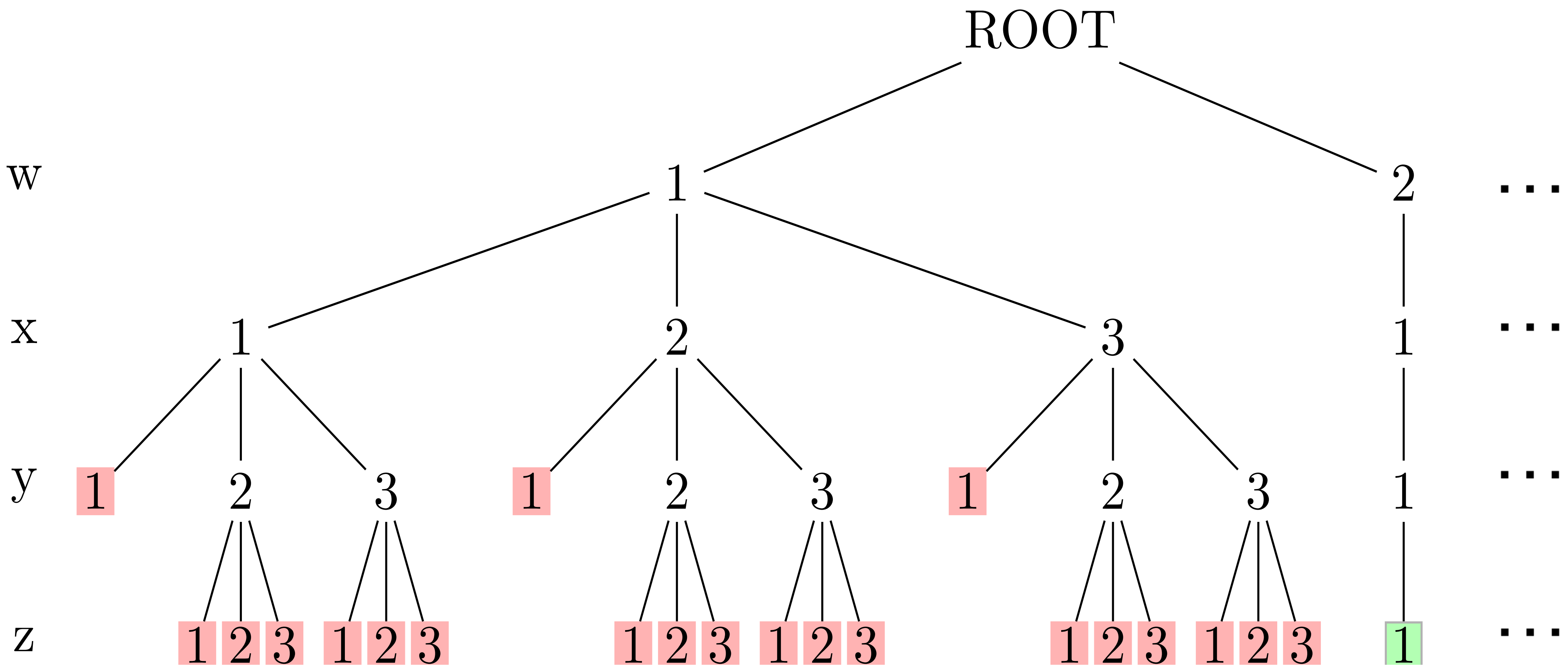
- Set of *n* variables ***V***.
  - E.g. Sudoku, variables are the empty squares that we need to fill

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

Combinatorial Problems

- Set of *n* variables ***V***.
  - E.g. Sudoku, variables are the empty squares that we need to fill
- Set of finite domains ***D***, possible values each variable can take.
  - For sudoku each variable has the domain 1-9
- Set of constraints ***C***, each constraint involves at least one variable:
  - For sudoku, each variable is involved in 3 *AllDifferent* constraints across row, column and subgrid.
  - Alternative constraints:
    - Unary constraint: $x_i > 10$
    - Precedence constraint: $x_j + p_j < x_k$
- Size of search space d^n: typically too large for brute force search.

Search Space/Tree



Search Space/Tree

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

- Variable $x_{i,j}$ for every empty cell
- Domains {1…9}
- Constraint enforcing all different in values in each row, in each column, and each subgrid.

Combinatorial problem example: Timetabling

- Assign every lecture/lab a classroom and a timeslot.
- Combinatorics are huge, particularly if we don't consider constraints.
- 2 variables for each lecture/lab to be assigned
- Domains?
  - Set of possible classrooms (>200) / timeslots (45)
- Sample constraints?
  - lecturer can't teach 2 classes at the same time,
  - room can't hold 2 classes at same time,
  - classroom has capacity

CO.MSC-A! - KARIN_9_1 - MSc. in Artificial Intelligence

NOTE THAT THESE TIMETABLES ARE PROVISIONAL. PLEASE CONFIRM WITH HEAD OF DEPARTMENT.

Weeks 2 (9 Sep 2019-15 Sep 2019)

	Monday		Tuesday		Wednesday	Thursday	Friday
9:00				CO.MSC-AI-A Knowledge Repr. C127	CO.MSC-AI-A, CO.MSC-AI-B Natural Lang. Proc. B260	CO.MSC-AI-A, CO.MSC-AI-B Natural Lang. Proc. B214	CO.MSC-AI-B, CO.MSC-AI-A Knowledge Repr. B269
9:15							
9:30							
9:45							
10:00	CO.MSC-AI-B, CO.MSC-AI-A Metaheuristic Optim. C214				CO.MSC-AI-A, CO.MSC-AI-B Rsrch. Practice & Ethics B260	CO.MSC-AI-A, CO.MSC-AI-B Metaheuristic Optim. B225	CO.MSC-AI-B Knowledge Repr. C127
10:15							
10:30							
10:45							
11:00		CO.MSC-AI-A, CO.MSC-AI-B Big Data Processing D237		CO.MSC-AI-A, CO.MSC-AI-B Knowledge Repr. B260	CO.MSC-AI-A Prac. Mach. Learning C127	CO.MSC-AI-A, CO.MSC-AI-B Prac. Mach. Learning B149	
11:15							
11:30							
11:45							
12:00	CO.MSC-AI-A, CO.MSC-AI-B Rsrch. Practice & Ethics B149			CO.MSC-AI-A, CO.MSC-AI-B Prac. Mach. Learning B149		CO.MSC-AI-A, CO.MSC-AI-B Big Data Processing B149	
12:15							
12:30							
12:45							
13:00							
13:15							
13:30							
13:45							
14:00	CO.MSC-AI-B Metaheuristic Optim. C127			CO.MSC-AI-A Big Data Processing C127	CO.MSC-AI-A Metaheuristic Optim. C127	CO.MSC-AI-B Prac. Mach. Learning C127	
14:15							
14:30							
14:45							
15:00		CO.MSC-AI-A Rsrch.	CO.MSC-AI-B Rsrch.				
15:15							

Combinatorial problem example: Timetabling

- CS @ CIT (Rough numbers):
  - > 2000 campus students
  - > 300 Modules
  - 50 full time staff
  - > 50 rooms
- Example from https://www.unitime.org/uct_description.php
  - *Purdue is a large (39,000 students) public university with a broad spectrum of programs at the undergraduate and graduate levels. In a typical term there are 9,000 classes offered using 570 teaching spaces. Approximately 259,000 individual student class requests must be satisfied*
  - Problem decomposed into
    - a centrally timetabled large lecture room problem (about 800 classes being timetabled into 55 rooms with sizes up to 474 seats),
    - individually timetabled departmental problems (about 70 problems with 10 to 500 classes),
    - and a centrally timetabled computer laboratory problem (about 450 classes timetabled into 36 rooms with 20 to 45 seats).

Combinatorial problem example: Timetabling

(insertion) heuristic approach

- Repeat following step until all assigned:
  - Choose module+classgroup, choose time slot and choose room

- But how to choose?
  - Random?
  - Dedicated heuristic,
    - e.g. choose room with smallest non-negative value for (*capacity* – *classize*)
    - Start with variables that will be hard to satisfy, e.g. large classes that can only fit into a few rooms (compared to small classes that can fit in most rooms)

Combinatorial Problems

- Set of *n* variables ***V***.
- Set of finite domains ***D***: possible values each variable can take.
- Set of constraints ***C***, stating what combinations of values are allowed on subsets of the variables.
- Size of search space d^n, where d is the domain size i.e. the number of values in the domain of a variable.

Combinatorial Optimization Problems

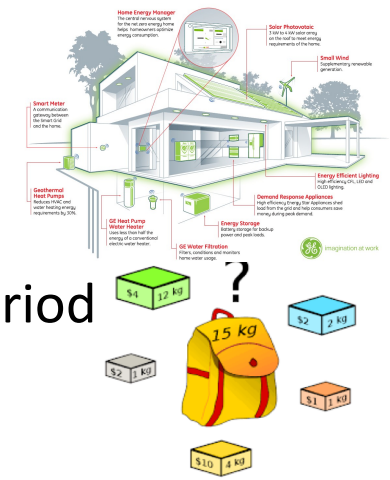
Optimization and Objective Functions

- Sudoku is a *Satisfaction* problem. Satisfaction problems are where we just want to find a solution, i.e. an assignment such that no constraint is violated.

- Optimization refers to choosing the ***best*** solution from some set of available alternatives, so we have a criteria that quantifies the quality of a solution.

- This criteria is referred to as an **objective function** that computes the quality of the solution based on the values assigned to the variables.

- Find a solution that minimizes/maximizes the objective value.
  - Minimize energy cost across schedulable appliances in the home in a 24 hour period
  - Maximize the profit of items placed in the knapsack.



Examples

- Combinatorial problems arise in many areas of computer science and applications domains:
  - Finding shortest / cheapest round trips (TSP)
  - Finding models of propositional formulae (SAT)
  - Planning, scheduling, time-tabling
  - Vehicle routing
  - Location and clustering
  - Internet data packet routing
  - Protein structure prediction

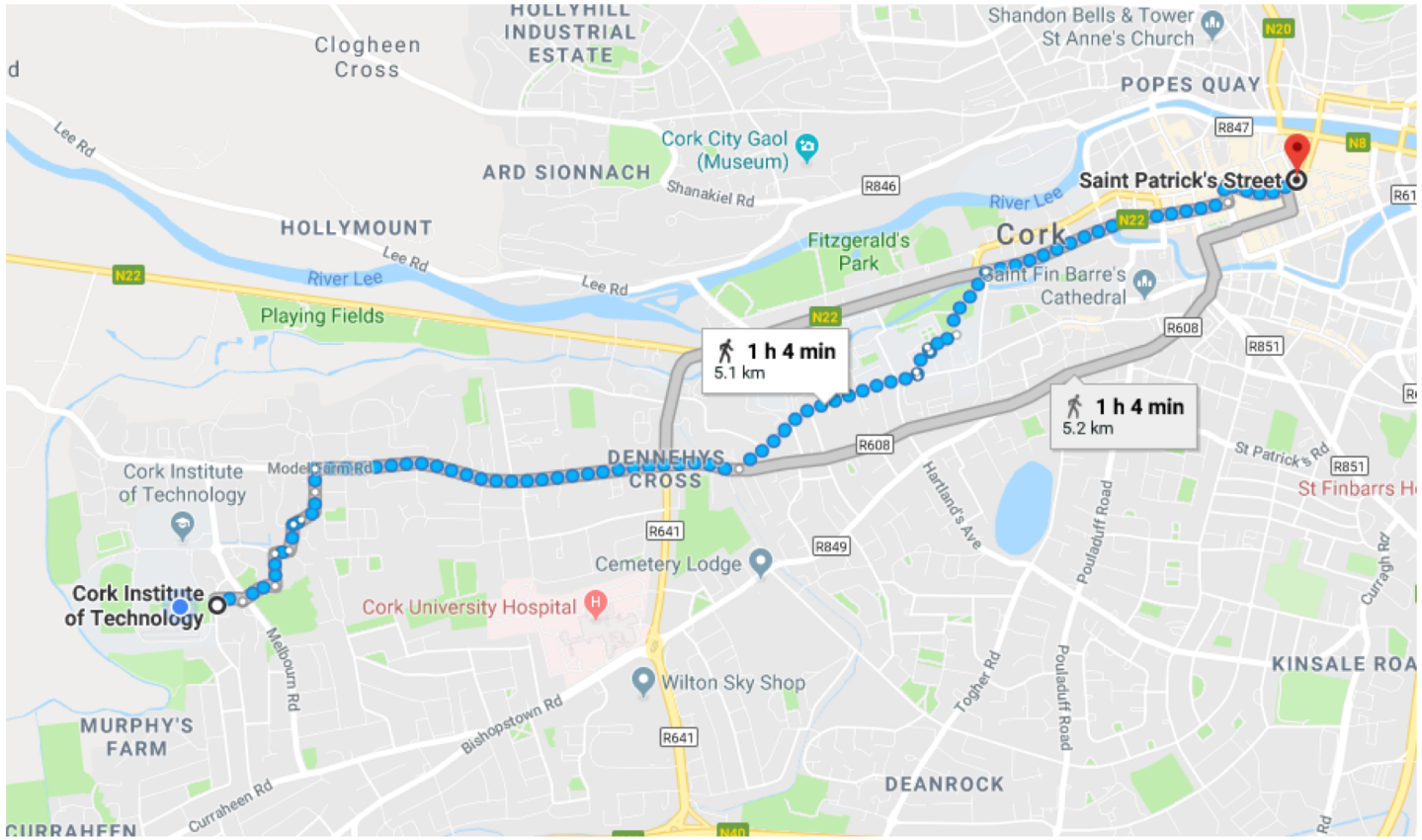
- Range from quite easy to hard ones
  - Here we focus on the hard ones

Easy example

Find the best (most valuable) element from the set of alternatives

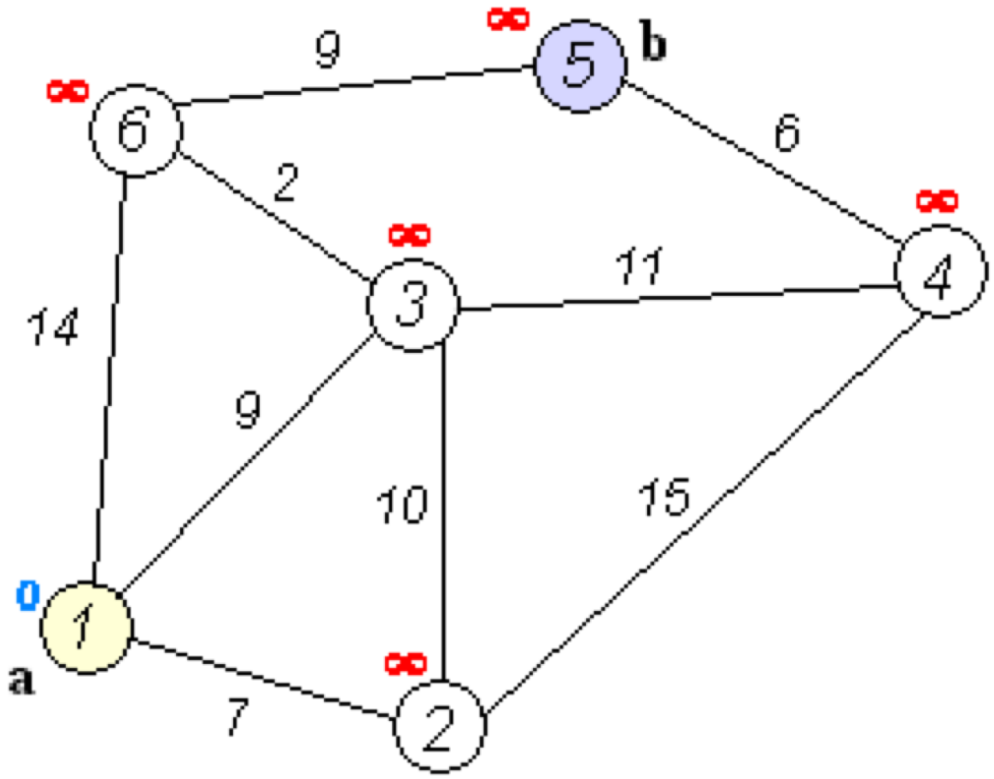
A more difficult (but still “easy”) one

Find best (shortest) route from A to B in an edge-weighted graph



A more difficult (but still “easy”) one

Find best (shortest) route from A to B in an edge-weighted graph



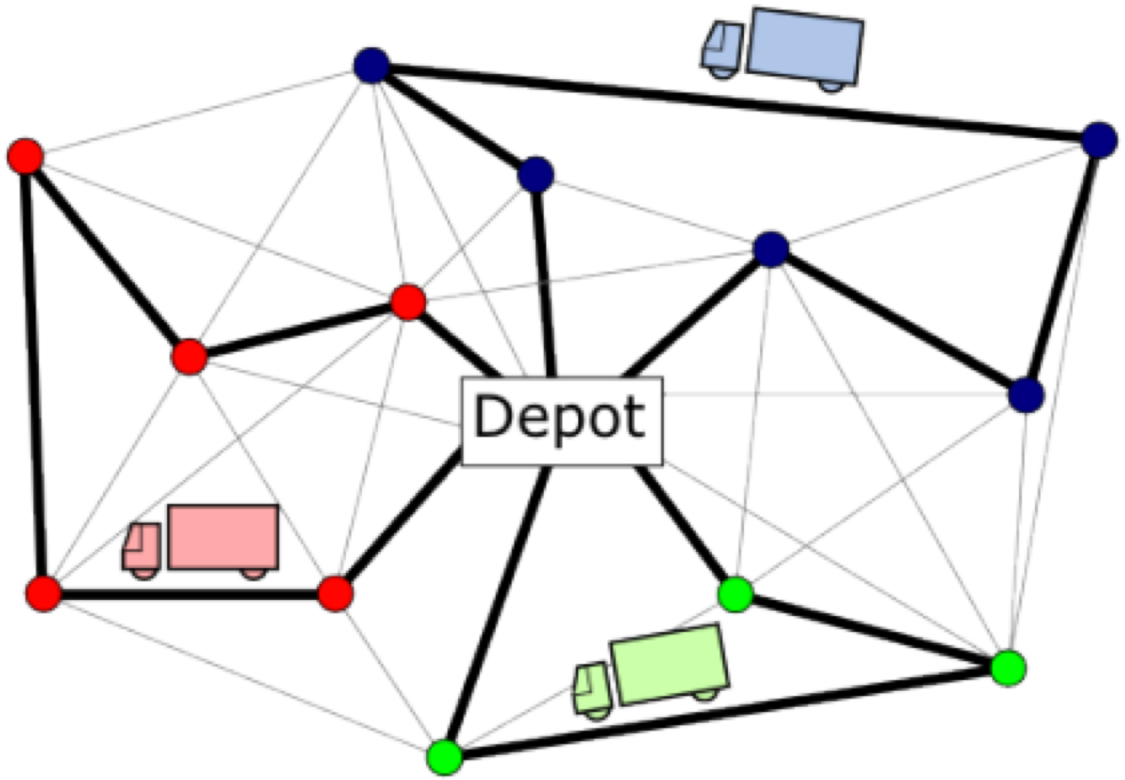
https://en.wikipedia.org/wiki/Dijkstra's_algorithm#/media/File:Dijkstra_Animation.gif

A harder one

Find best (shortest) round trip through some cities, aka Traveling Salesman Problem (TSP)

A real-life like problem

TSP arises as sub-problem, e.g., in vehicle routing problems (VRPs)



Combinatorial Optimisation Problems

- Realistic problems can involve many complicating details

- Examples in VRP case are:
  - Time windows, access restrictions, priorities, split delivery, …
  - Capacity constraints, different cost of vehicles, …
  - Working time constraints, breaks, …
  - Stochastic travel times or demands, incoming new request, …

- We will focus on simplified models of (real-life) problems
  - Useful for illustrating algorithm principles
  - They are “hard” to capture essence of more complex problems
  - Are treated in research to yield more general insights

Simple Scheduling Example

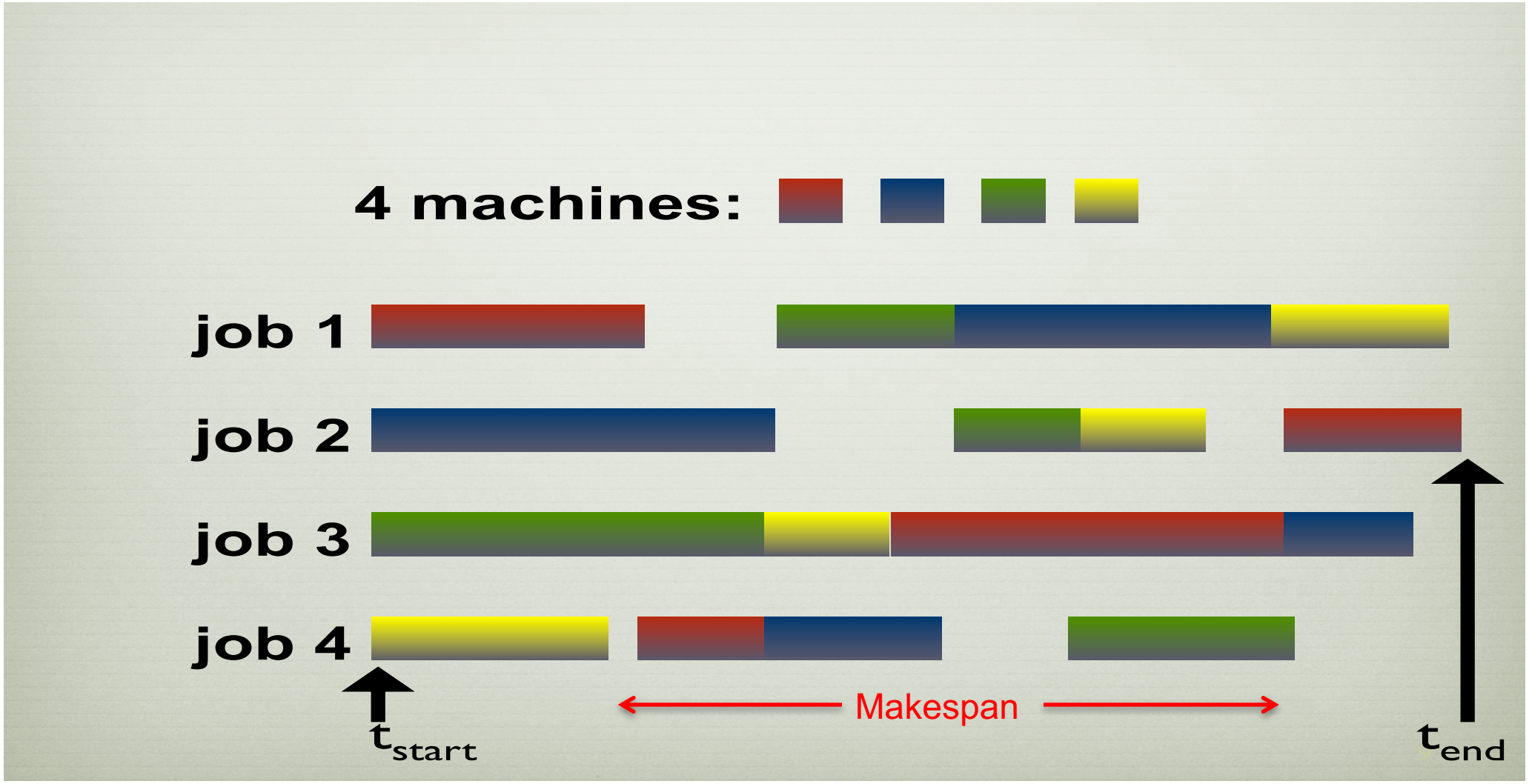
An *n x m* open shop problem (OSP):

- *n* jobs, *m* machines.
- *m* tasks per job.
- Resource (disjunctive) constraint on tasks of a job or machine:

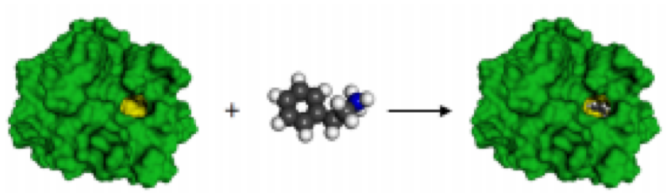
$$(t_i + p_i \leq t_j) \vee (t_j + p_j \leq t_i)$$

- Objective: Minimize the *makespan*, i.e. the length of time between the earliest start time and the latest finish time across all tasks.

Open Shop Scheduling



Optimization problems arise everywhere!



Most such problems are very hard (***NP-hard!***)

How to solve Combinatorial Optimisation Problems?

Many possible approaches

- Systematic enumeration is probably not realistic, at least for interesting problems
- Some approaches may eliminate certain assignment or partial tours through careful reasoning (as you will see in a later semester in the module Decision Analytics)
- Other intuitive approach: start with some good guess and then try to improve it iteratively

The latter is an example of a *HEURISTIC* approach to optimization

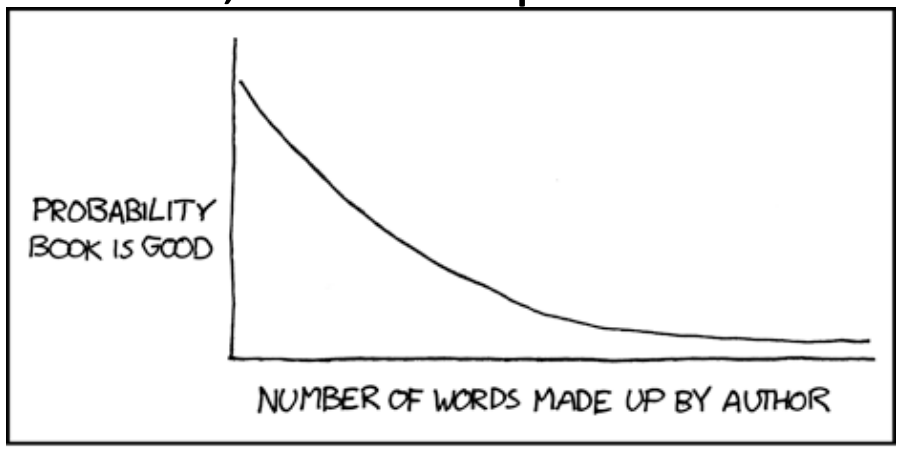
Solving (combinatorial) optimization problems

- Systematic enumeration (brute force)
- Problem specific, dedicated algorithms
- Generic methods for exact optimization
- **Heuristic methods**

Heuristics

What is a heuristic?

- A heuristic is a rule of thumb, i.e. an imperfect criteria to base a decision on.



https://xkcd.com/483/

- Method which, although unable to determine a perfectly accurate solution, should produce a set of good quality approximations to exact solutions.

- Heuristics were initially based essentially on experts' knowledge and experience and aimed to explore the search space in a particularly convenient way

Heuristic methods

- **Heuristic methods** intend to compute efficiently, **good solutions** to a problem **with no guarantee of optimality**
- Range from rather simple to quite sophisticated approaches
- Inspiration often from
  - human problem solving
  - rules of thumb, common sense rules
  - design of techniques based on problem-solving experience
  - natural processes
    - evolution, swarm behaviors, annealing, ...
- usually used when there is no other method to solve the problem under given time or space constraints
- Often simpler to implement / develop than other methods

Main characteristics

- A heuristic is designed to provide better computational performance as compared to conventional optimization techniques, at the expense of lower accuracy

- The ***rules of thumb*** underlying a heuristic are often very specific to the problem under consideration

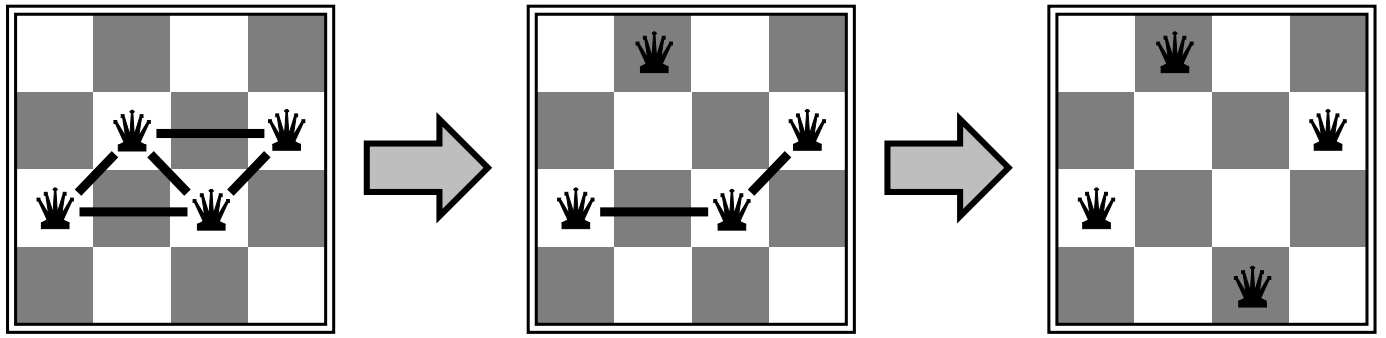
- Heuristics use domain-specific representations

Heuristic example from Russel Norvig: N-Queens Problem

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

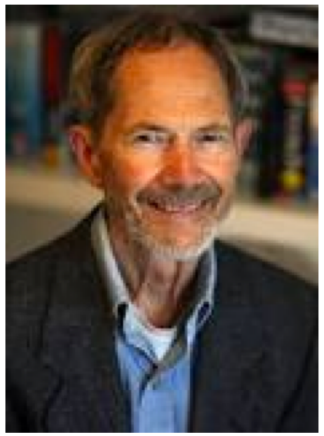


Almost always solves n-queens problems almost instantaneously for very large n, e.g., $n = 1million$

What are Metaheuristics?

- The term ***metaheuristics*** was proposed by Glover at mid-80s as a family of searching algorithms able to define a high level heuristic used to guide other heuristics for a better evolution in the search space

- The most attractive feature of a metaheuristic is that its application requires no special knowledge on the optimization problem to solve

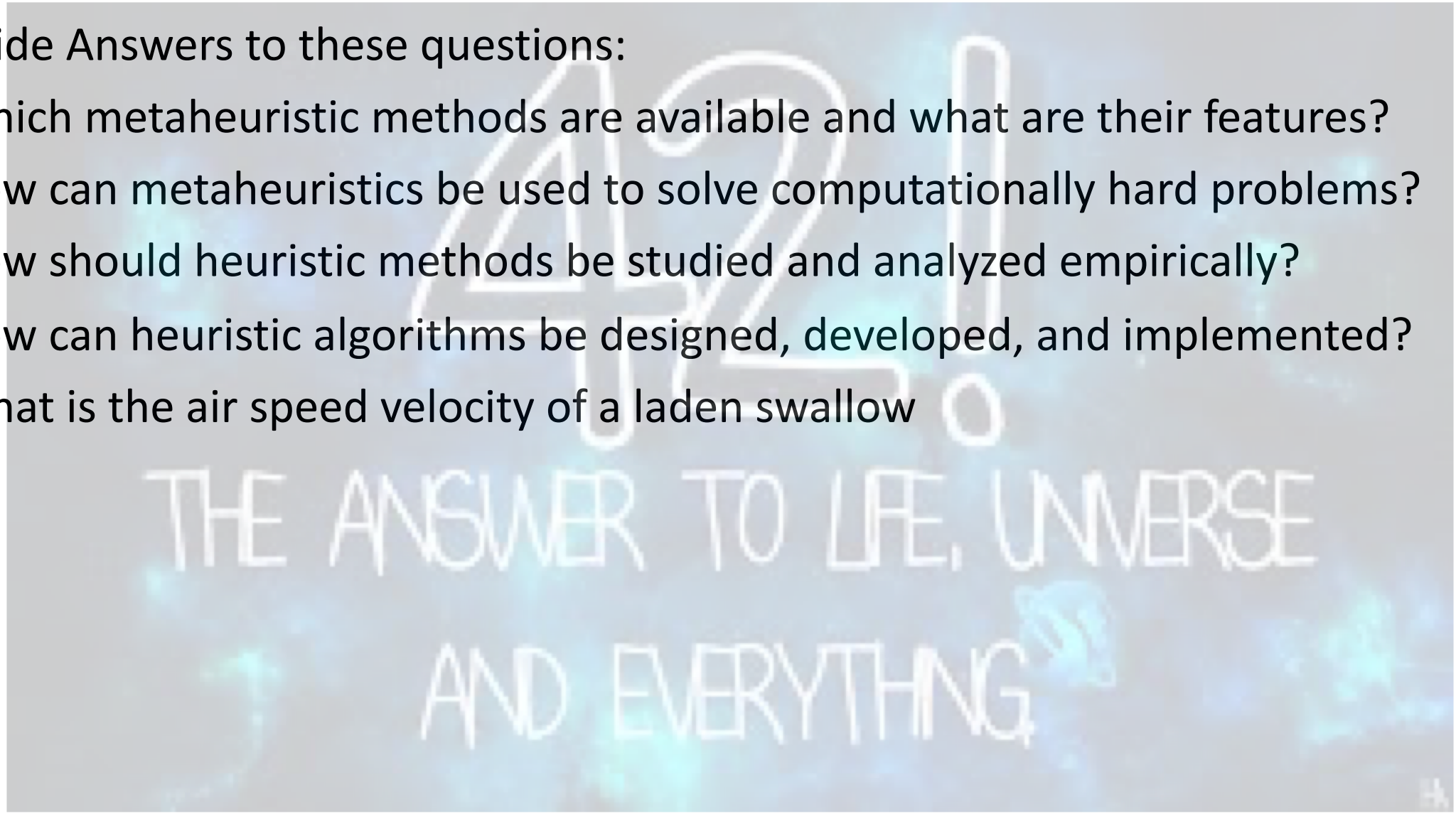


Fred W. Glover

This module

Provide Answers to these questions:

- Which metaheuristic methods are available and what are their features?
- How can metaheuristics be used to solve computationally hard problems?
- How should heuristic methods be studied and analyzed empirically?
- How can heuristic algorithms be designed, developed, and implemented?
- What is the air speed velocity of a laden swallow



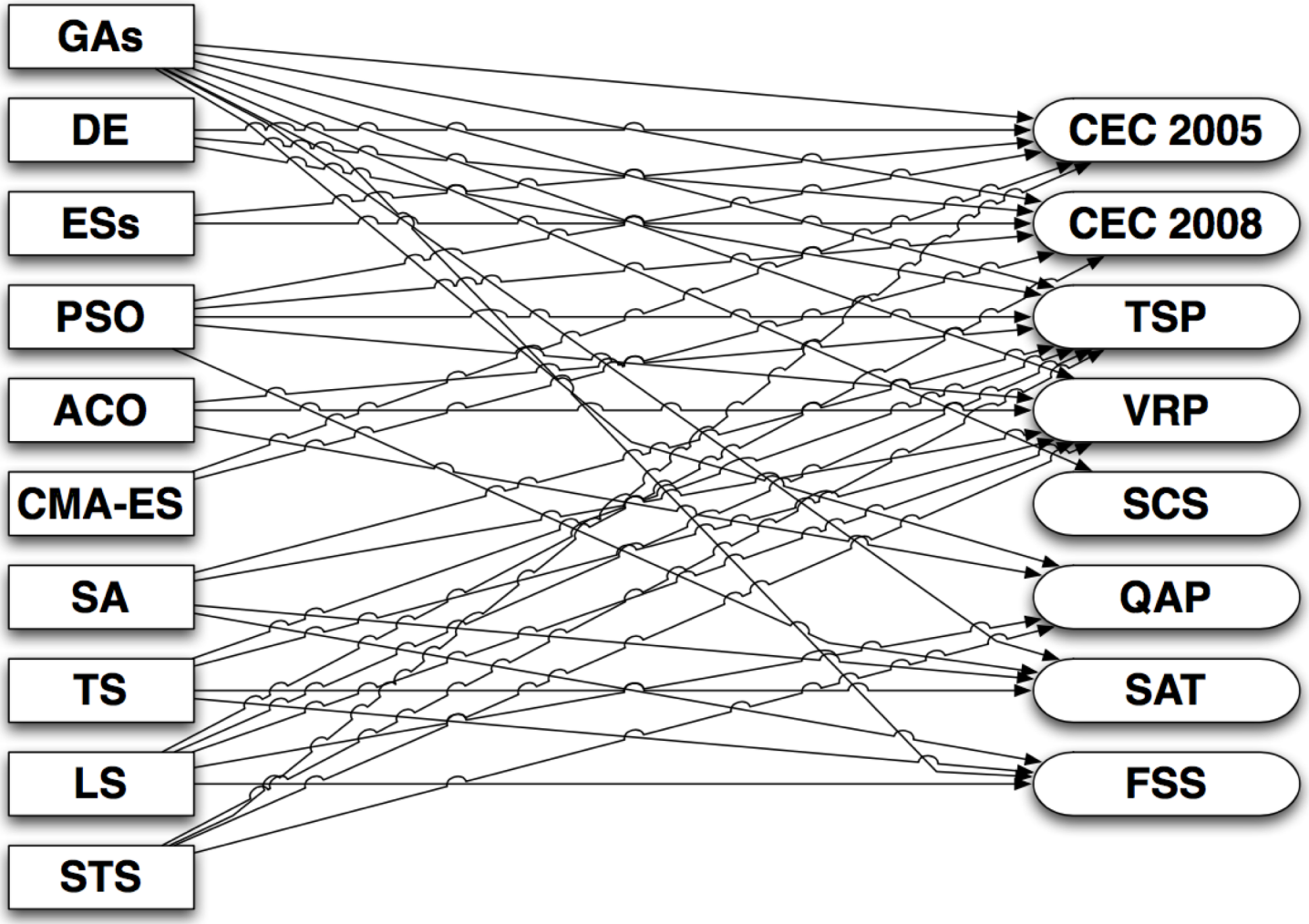
Motivations

- 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

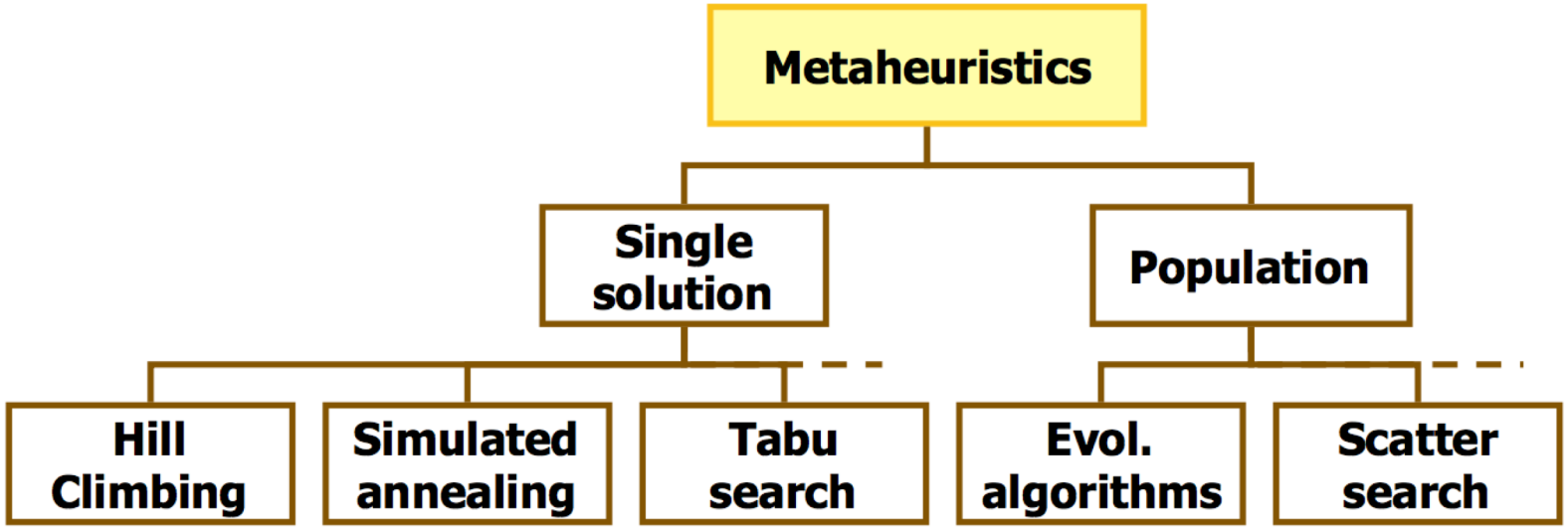
***Abstract*—A framework is developed to explore the connection between effective optimization algorithms and the problems they are solving. A number of "no free lunch" (NFL) theorems are presented which establish that for any algorithm, any elevated performance over one class of problems is offset by 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
Insertion 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()
```

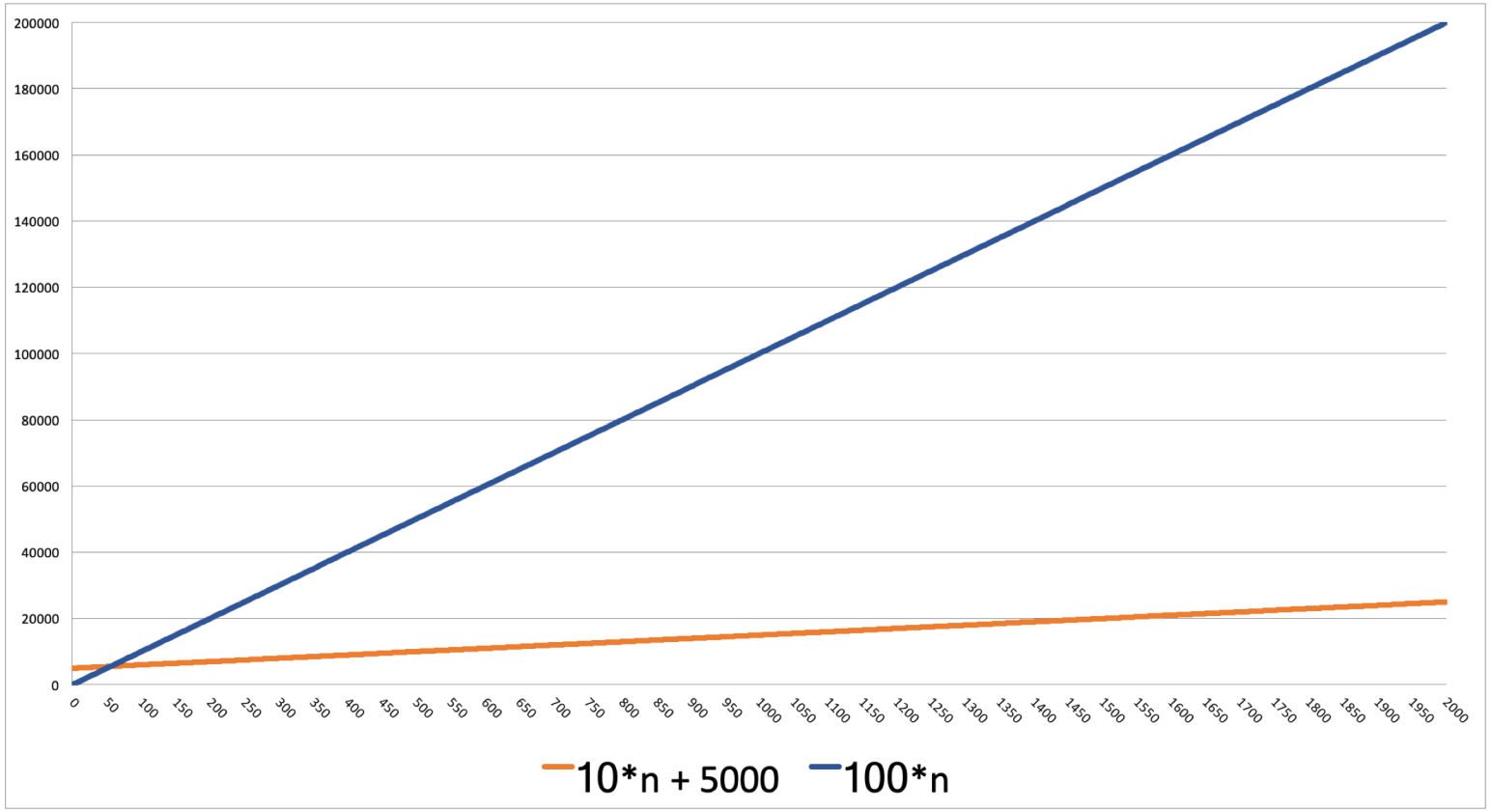
$$\begin{aligned}\text{Runtime}(g(n)) &\approx (1+2+3+\cdots+n)\cdot\text{Runtime}(f()) \\ &\approx \frac{n^2+n}{2}\cdot\text{Runtime}(f())\end{aligned}$$

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

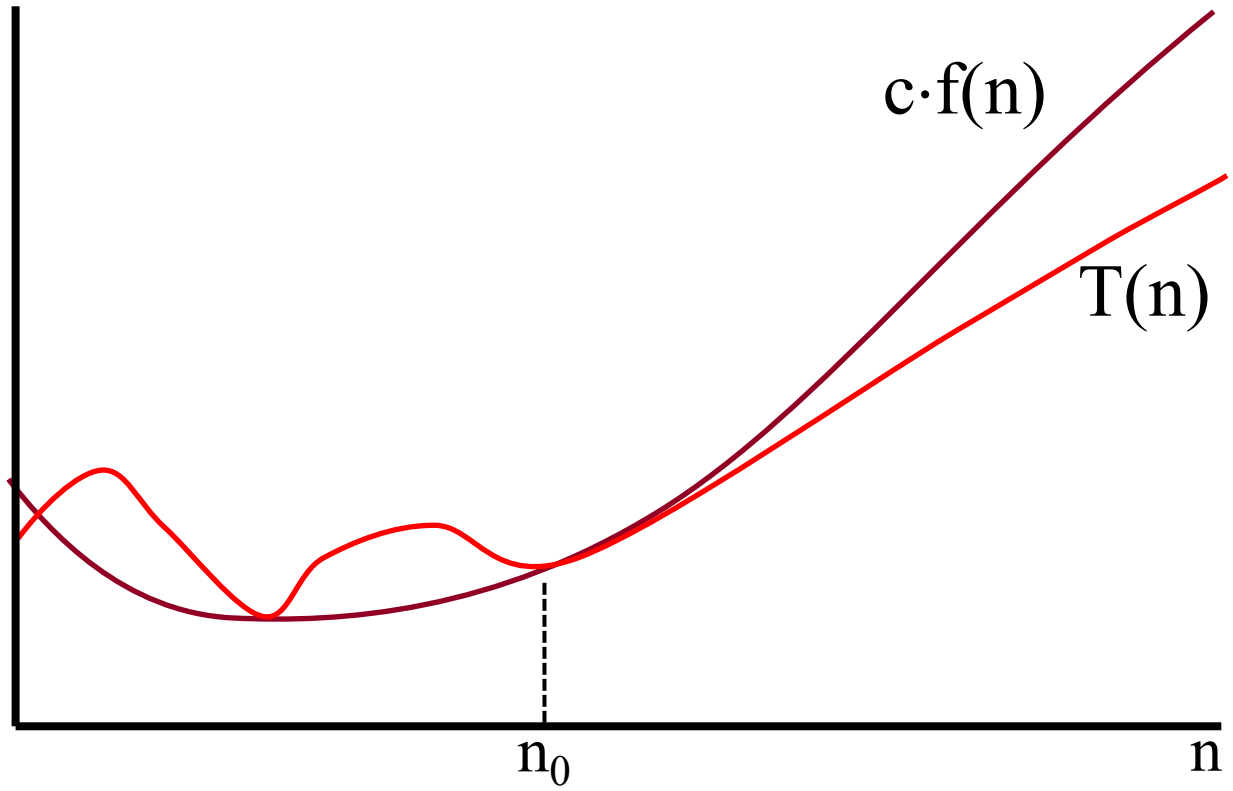
- We are typically interested in the execution time of large instances of a problem, e.g., when n → ∞, (asymptotic complexity).

Big O

- A method to characterize the execution time of an algorithm, how it scales with its input:
  - 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

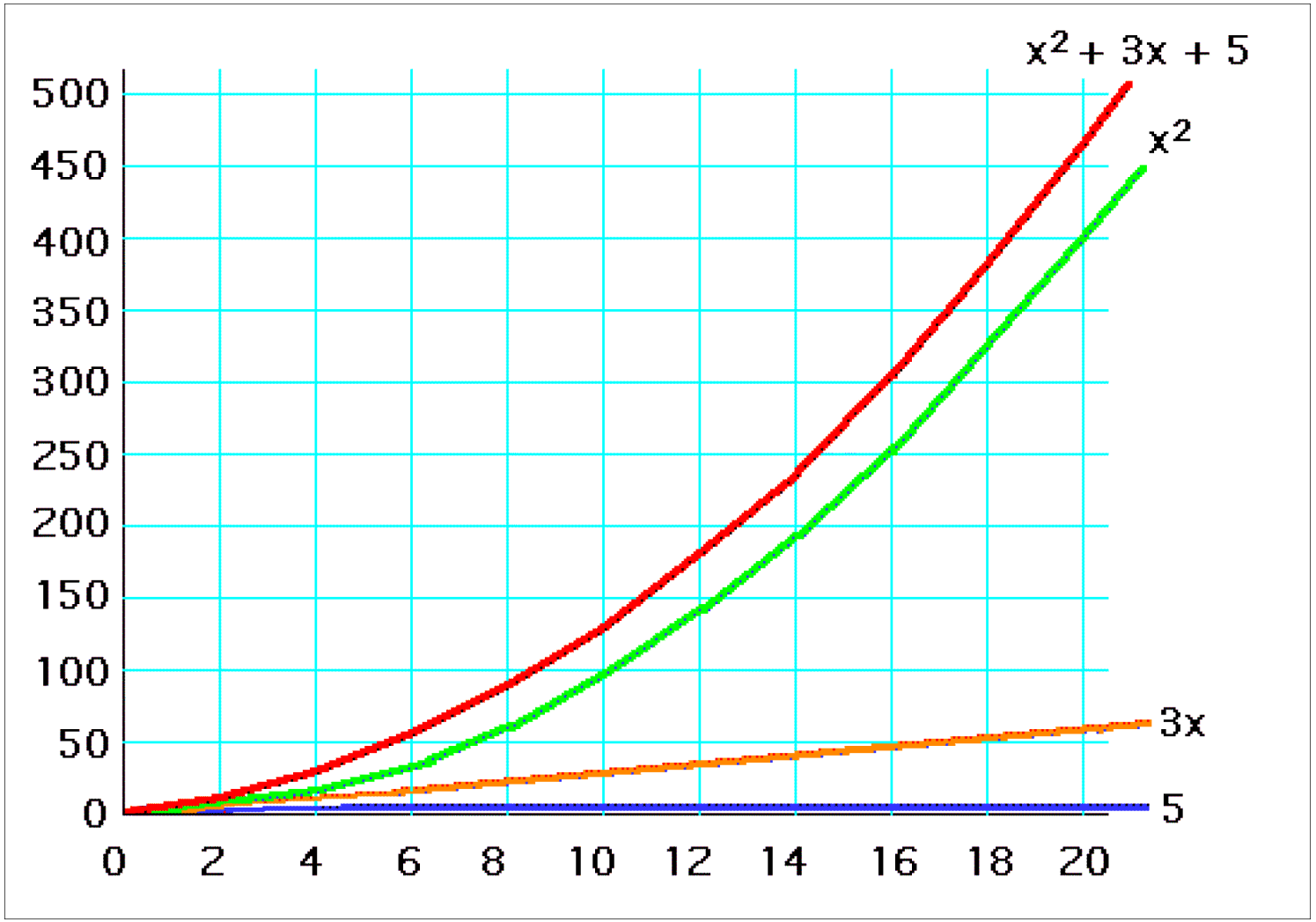
- We say the algorithm requires $O(n^3)$ time
  - We call this **Big O** notation
  - (Really what is typically meant is big theta, i.e. the smallest for which it holds)

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 (interesting problems?)
  - For sufficiently large problem sizes, the highest-order term swamps all the rest!
- Consider $R = x^2 + 3x + 5$ as x varies:

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

$y = x^2 + 3x + 5$, for x=1..20

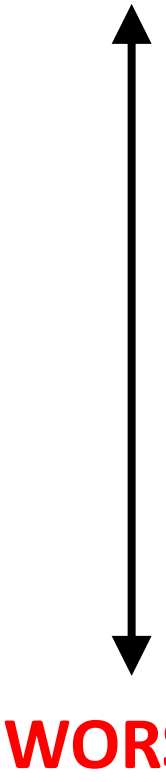


Big O: example

- 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.

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
$5n^3 + 200n^2 + 15$	$O(n^3)$
$3n^2 + 2^{300}$	$O(n^2)$
$5 \log_2 n + 15 \ln n$	$O(\log n)$
$2 \log n^3$	$O(\log n)$
$4n + \log n$	$O(n)$
2^{64}	$O(1)$
$\log n^{10} + 2\sqrt{n}$	$O(\sqrt{n})$
$2^n + n^{1000}$	$O(2^n)$

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()
```

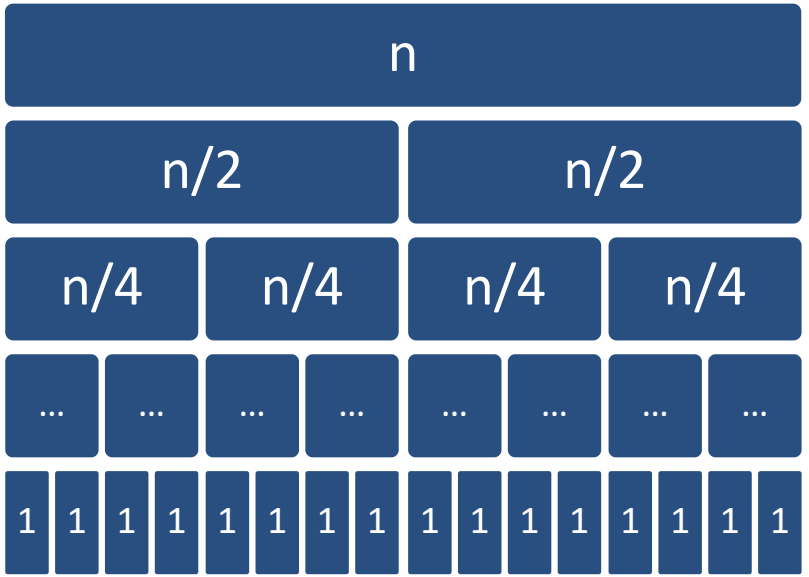
$\longrightarrow O(mnp)$

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 $O(n \log n)$

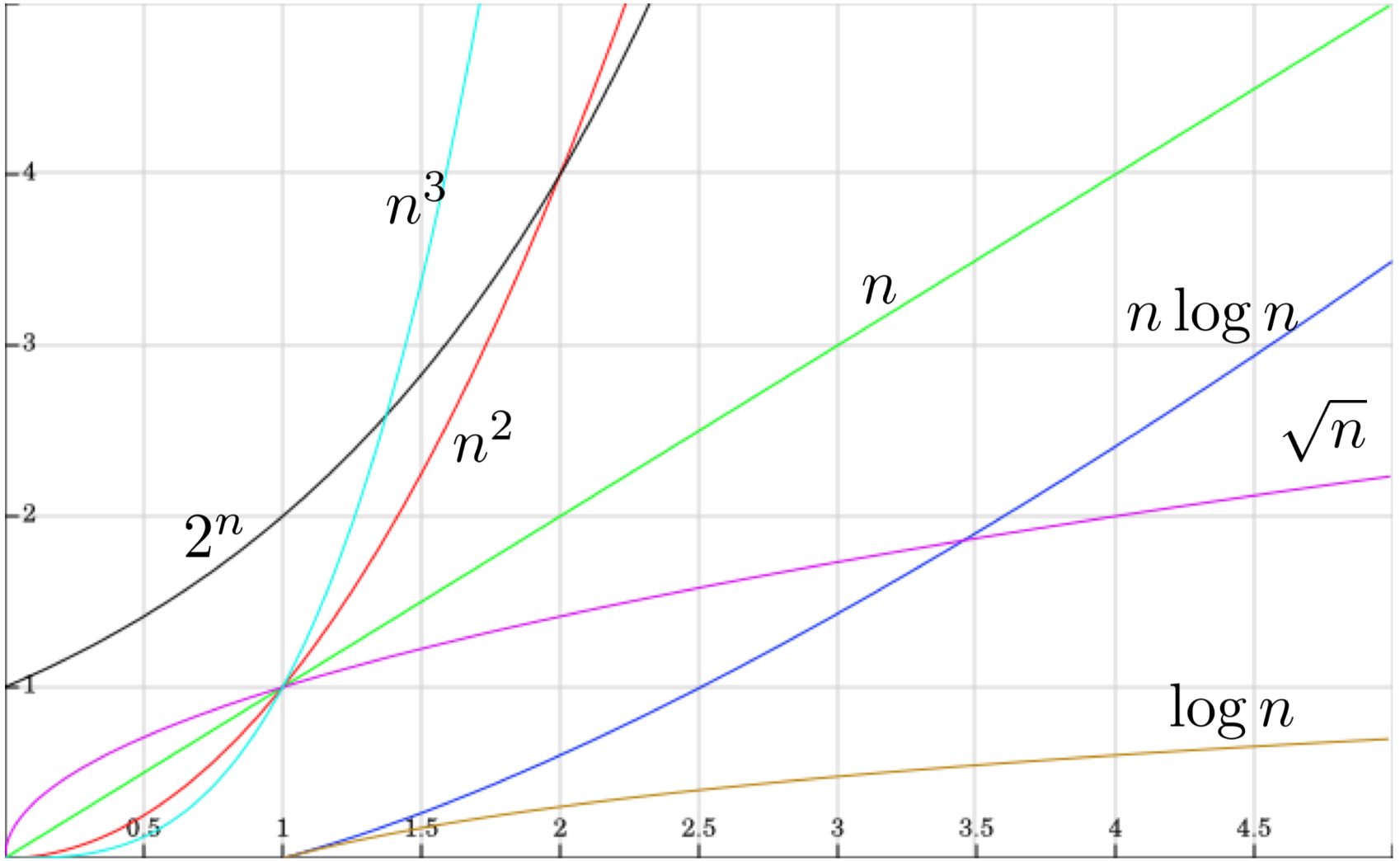
Complexity analysis: recursion

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

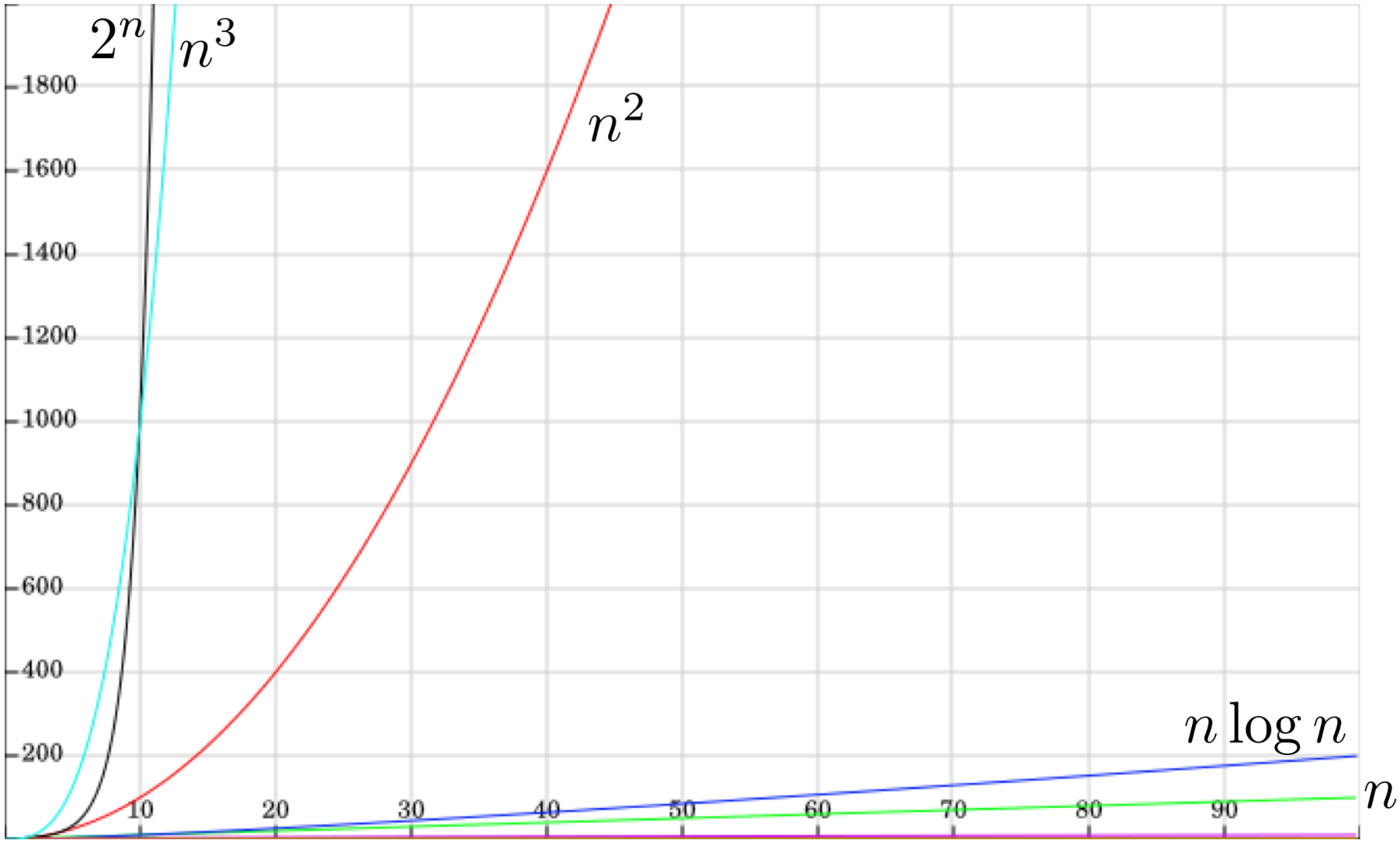
$$
\begin{aligned}
T(n) &= n + T(n-1) \\
T(n) &= n + (n-1) + (n-2) + \cdots + 2 + 1 \\
T(n) &= \frac{n^2 + n}{2}
\end{aligned}
$$

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

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

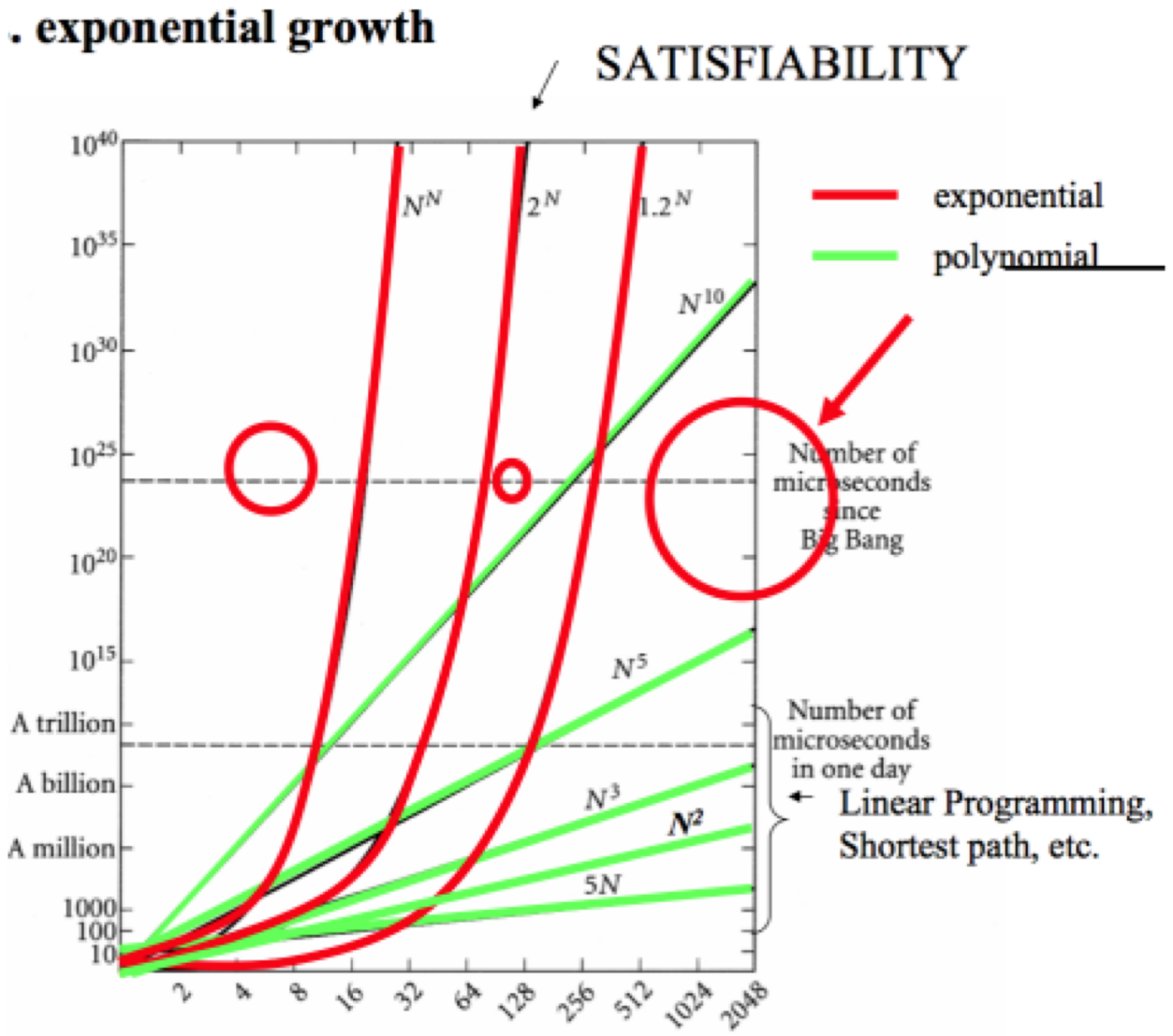
How about “big data”?

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.