Algorithms and Data Structures

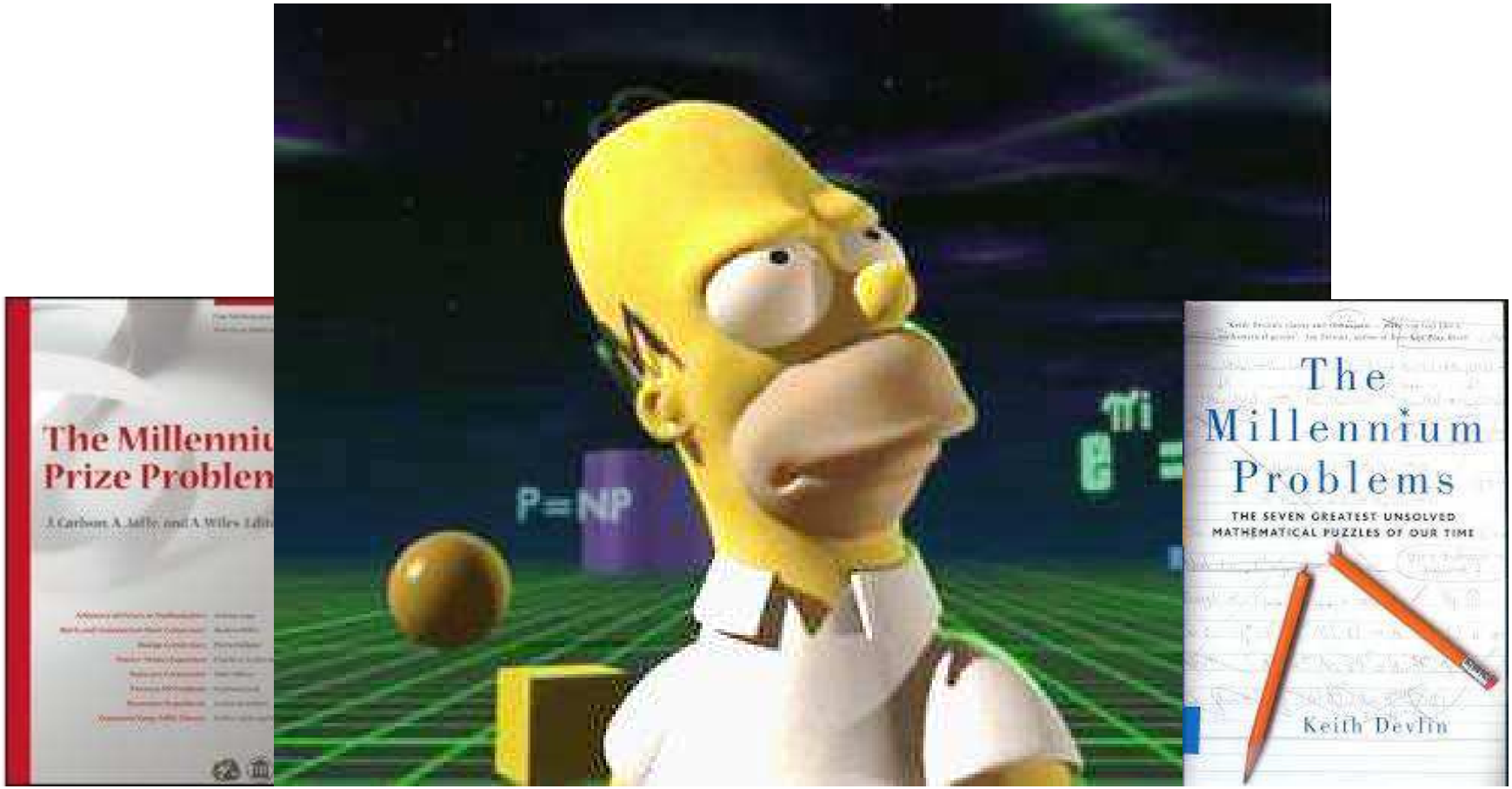
NP-Completeness

Guest lecture by Harald Søndergaard

Semester 2, 2019

A Million Dollar Question: Is P = NP?

This is one of the seven "millennium problems": The Clay Institute's seven most important unsolved mathematical problems.



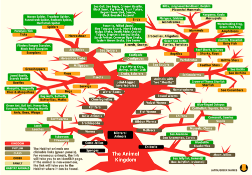
Who Wants to Be a Millionaire?

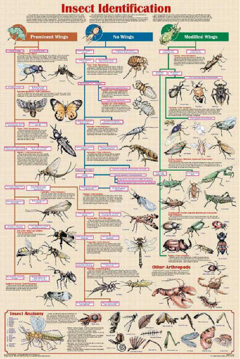
The "P versus NP" problem comes from computational complexity theory.

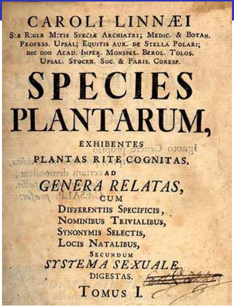
If you could use USD 1,000,000, solve this. Plus, you will most likely win a Turing Award as a bonus.

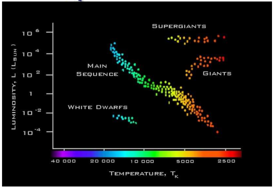
Or, solve any of the other millennium problems—all but one remain unsolved.

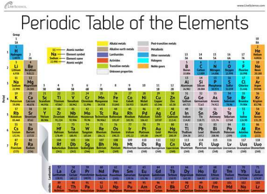
Taxonomy

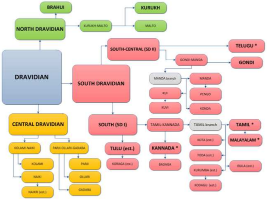





CAROLI LINNÆI
S:Æ R:GIÆ M:TIS SVECIÆ ARCHIATRI; MEDIC. & BOTAN.
PROFESS. UPSAL; EQUITIS AUR. DE STELLA POLARI;
nec non ACAD. IMPER. MONSPEL. BEROL. TOLOS.
UPSAL. STOCKH. SOC. & PARIS. CORESP.
SPECIES
PLANTARUM,
EXHIBENTES
PLANTAS RITE COGNITAS,
AD
GENERA RELATAS,
CUM
DIFFERENTIIS SPECIFICIS,
NOMINIBUS TRIVIALIBUS,
SYNONYMIS SELECTIS,
LOCIS NATALIBUS,
SECUNDUM
SYSTEMA SEXUALE
DIGESTAS.
TOMUS I.







Abstract Complexity

Rather than asking about the complexity of algorithms, complexity theory asks:

"What is the inherent difficulty of the problem?"

How do we know when we have come up with an algorithm which is optimal (in the asymptotic sense).

Algorithmic Problems

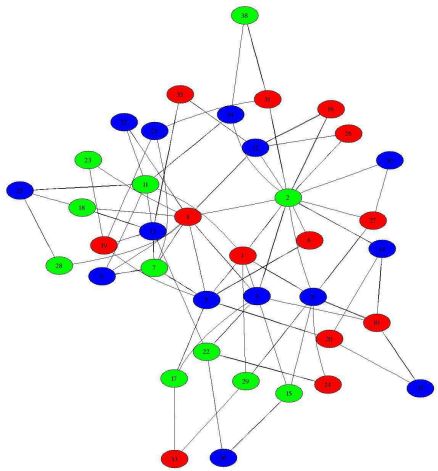
When we talk about a problem in computer science we almost always mean a family of instances of a general problem.

An algorithm for the problem has to work for all possible instances (input).

Example: The sorting problem—an instance is a sequence of items.

Example: The graph colouring problem—an instance is a graph.

Example: Equation solving problems—an instance is a set of, say, linear equations.



Algorithms and Their Demand on Resources

How efficient is a given algorithm
(in terms of time and/or space consumption)?

It is natural to measure that as a function of input size n.

Typical Functions in Algorithm Classification

1: Running time independent of input.

$\log n$: Typical for "divide and conquer" solutions, for example, lookup in a balanced search tree.

Linear: When each input element must be processed once and processing is independent of other elements.

$n \log n$: Each input element processed once and processing involves other elements too, for example, sorting.

n^2, n^3: Quadratic, cubic. Processing all pairs (triples) of elements.

2^n: Exponential. Processing all subsets of elements.

The Tyranny of Growth Rate

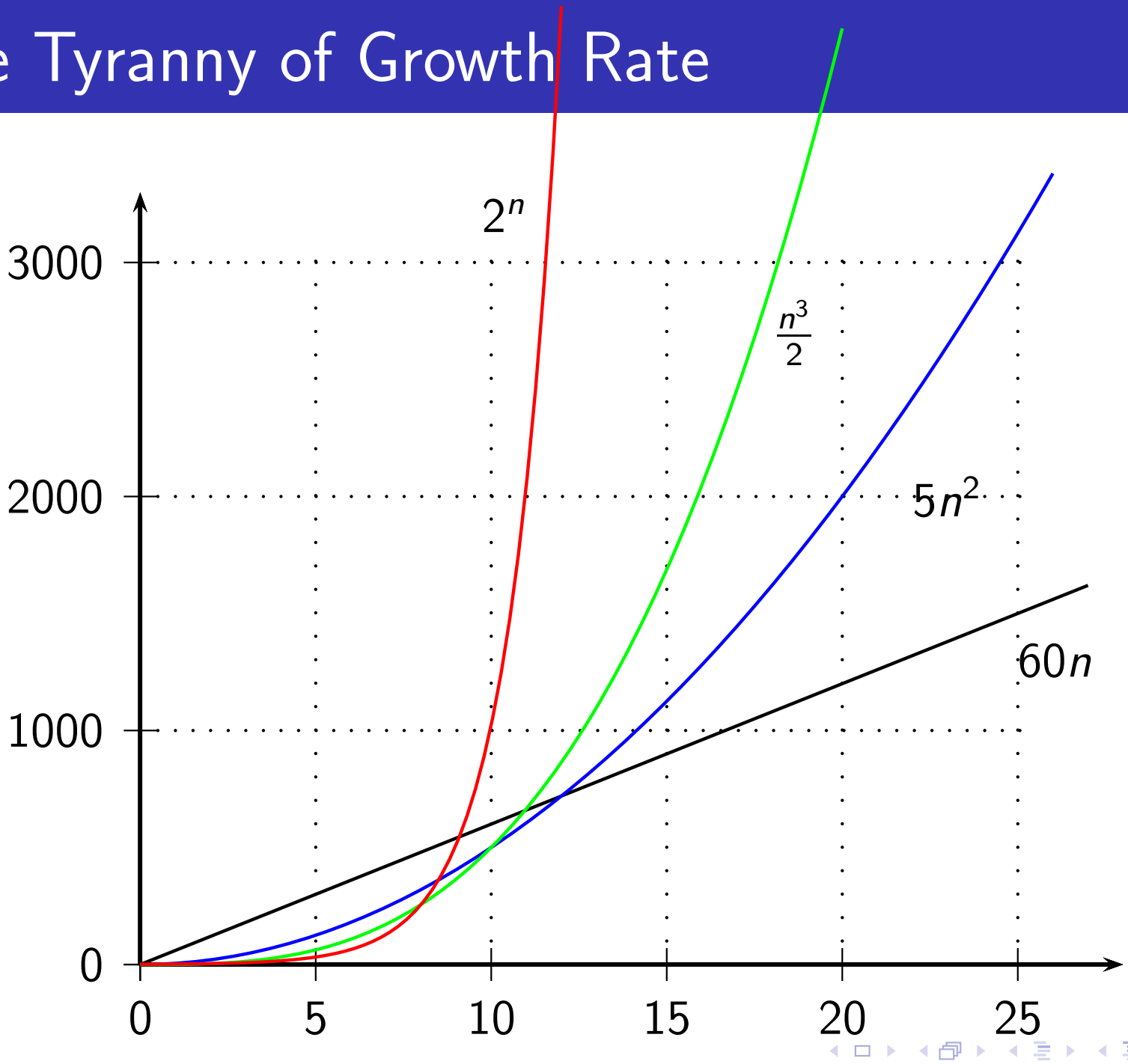
n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10^1	3	10^1	$3 \cdot 10^1$	10^2	10^3	10^3	$4 \cdot 10^6$
10^2	7	10^2	$7 \cdot 10^2$	10^4	10^6	10^{30}	$9 \cdot 10^{157}$
10^3	10	10^3	$1 \cdot 10^4$	10^6	10^9	–	–

10^{30} is one thousand times the number of nano-seconds since the Big Bang.

At a rate of a trillion (10^{12}) operations per second, executing 10^{30} operations would take a computer in the order of 10^{10} years.

That is more than the estimated age of the Earth.

The Tyranny of Growth Rate



How to Classify Problems?

Let us call a problem tractable if it has a polynomial-time algorithm that solves it, intractable otherwise.

For most of the practically relevant computational problems, very little is known about their absolute hardness.

A lot is known about their relative hardness.

Easy and Hard Problems

A path in a graph G is simple if it visits each node of G at most once.

Consider this problem for undirected graphs G:

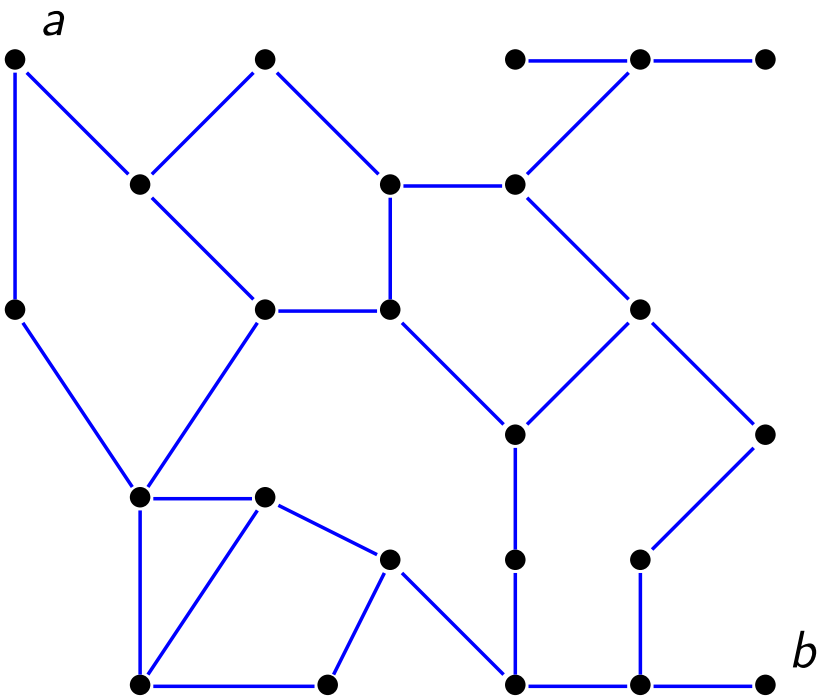
> SPATH: Given G and two nodes a and b in G,
> is there a simple path from a to b of length at most k?

And this problem:

> LPATH: Given G and two nodes a and b in G,
> is there a simple path from a to b of length at least k?

If you had a large graph G, which of the two problems would you rather have to solve?

Easy and Hard Problems



There are fast algorithms to solve SPATH.

Nobody knows of a fast algorithm for LPATH.

It is likely that the LPATH problem cannot be solved in polynomial time.

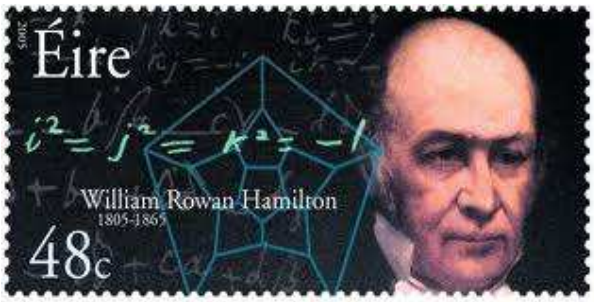
(But we don't know for sure.)

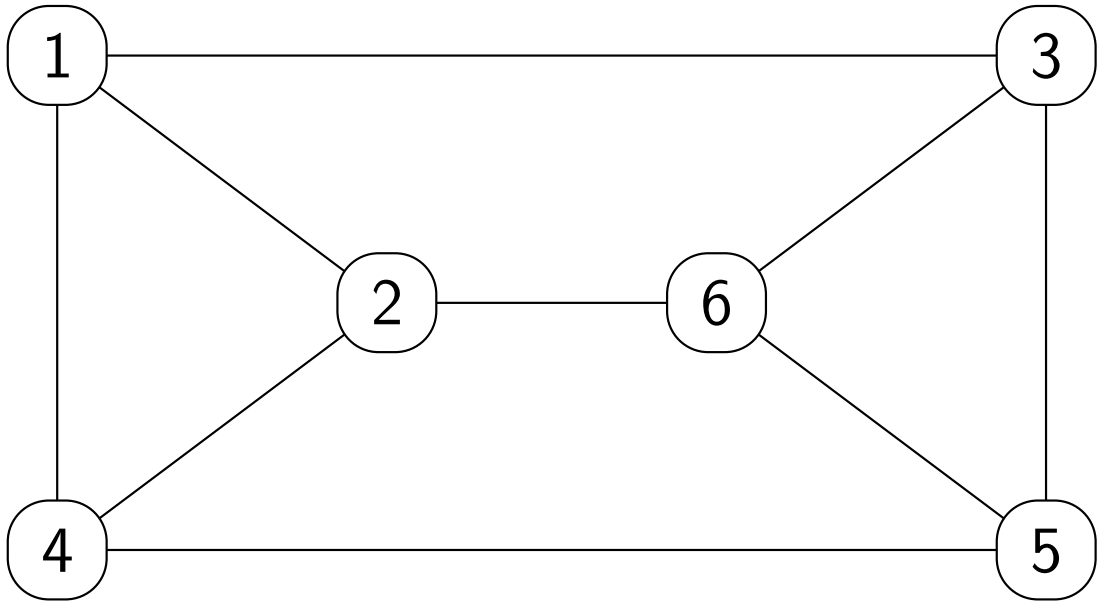
Hamiltonian Tours

The Hamiltonian tour problem is this:

> HAM: In a given graph, is there a simple path which visits all nodes of the graph, returning to the origin?

Is there a Hamiltonian tour of this graph?



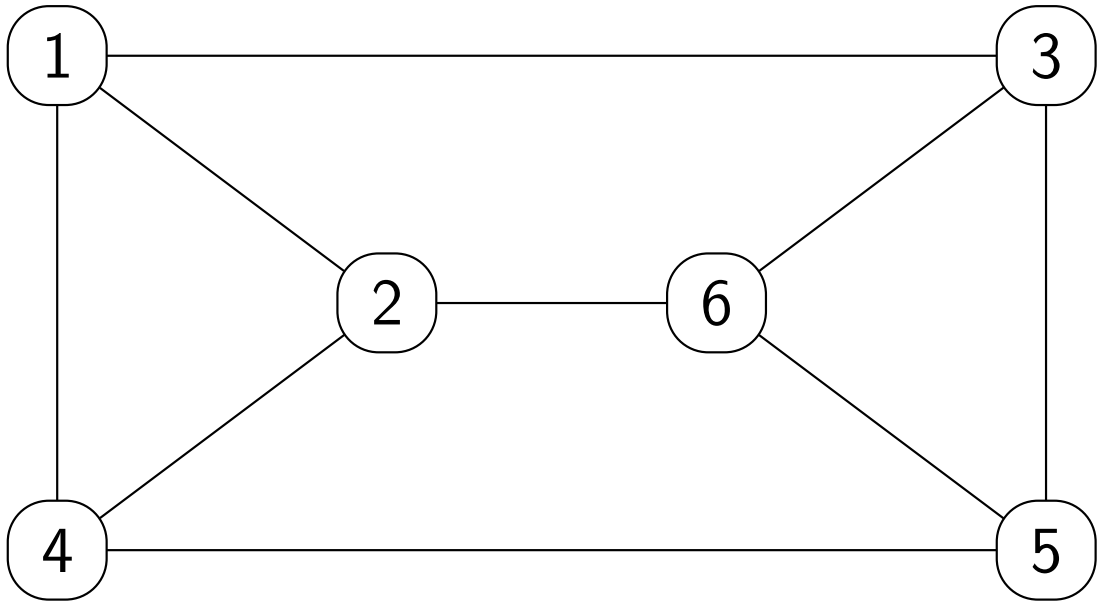


Eulerian Tours

The Eulerian tour problem is this:

> EUL: In a given graph, is there a path which visits each edge of the graph once, returning to the origin?

Is there a Eulerian tour of this graph?



More Problems

Try to rank these problems according to inherent difficulty:

- SAT: Given a propositional formula φ, is φ satisfiable?
- SUBSET-SUM: Given a set S of positive integers and a positive integer t, is there a subset of S that adds up to t?
- 3COL: Given a graph G, is it possible to colour the nodes of G using only three colours, so that no edge connects two nodes of the same colour?

Polynomial-Time Verifiability

SAT, SUBSET-SUM, 3COL, LPATH, and HAM share an interesting property.

If somebody claims to have a solution (a "yes instance") then we can quickly test their claim.

In other words, these problems seem hard to solve, but solutions allow for efficient verification. They are polynomial-time verifiable.

That property is shared by a very large number of interesting problems in planning, scheduling, design, information retrieval, networks, games, ...

Turing Machines

A Turing machine has an infinite tape through which it takes its input and performs its computations.

It can

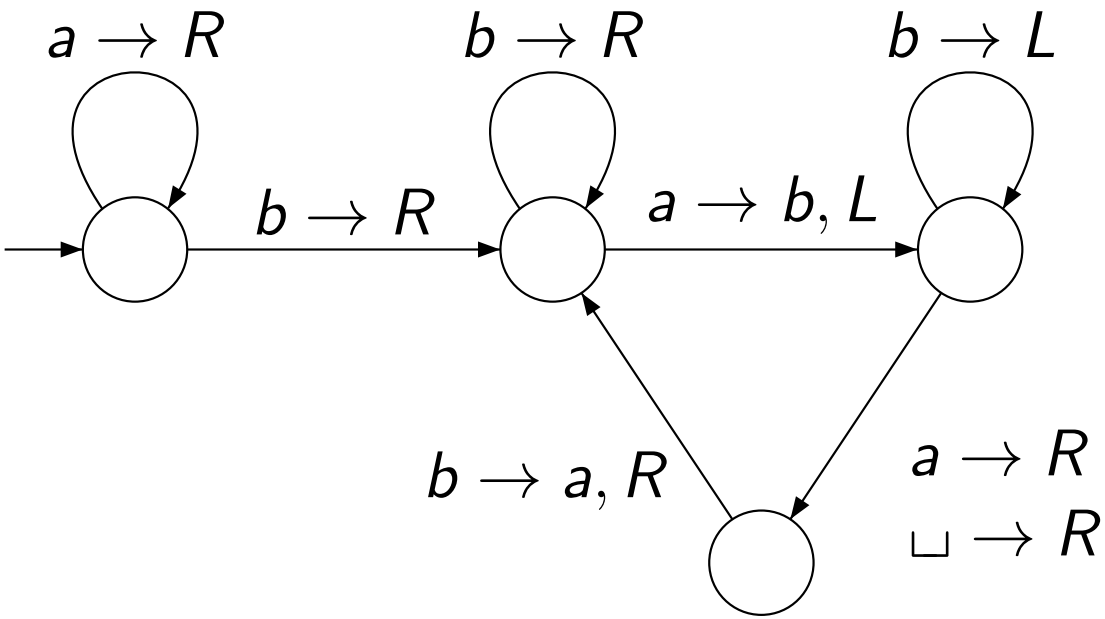
- both read from and write to the tape,
- move either left or right over the tape.

The tape is unbounded in both directions.

A Turing machine may fail to halt.

Turing Machine Example: Sorting

This (halting) Turing machine sorts a sequence of as and bs.



The tape alphabet is $\{\sqcup, a, b\}$.

Turing Machines and Computability

Surprisingly, Turing machines appear to have the same "computational power" as any other sensible computing device.

By this we mean that any function that can be implemented in C, Java, Haskell, ... can be implemented on a Turing machine.

The Turing machine has a certain universality property: There is a Turing machine which is able to simulate any other Turing machine.

Here we shall assume that Turing machines are used to implement decision procedures: algorithms with a yes/no answer.

Non-Deterministic Turing Machines

One variant of the Turing machine has a powerful guessing capability: If different moves are available, the machine will favour a move that ultimately leads to a 'yes' answer.

Adding this kind of non-determinism to the capabilities of Turing machines does not change what Turing machines can compute.

However, it may conceivably have an impact on efficiency.

What such a non-deterministic Turing machine can compute in polynomial time corresponds exactly to the class of polynomial-time verifiable problems.

P and NP

P is the class of problems solvable in polynomial time by a deterministic Turing machine.

NP is the class of problems solvable in polynomial time by a non-deterministic Turing machine.

Clearly $P \subseteq NP$. **Is $P = NP$?**

Most computer scientists consider this computer science's

most important open question

A Handle on Relative Hardness

We can classify "hardness" of problems using a concept of "reducibility".

Intuitively, if A can be reduced to B and we have an efficient algorithm for B then we have one for A.

Crucially, the reduction must happen in polynomial time.

A is polynomial-time reducible to B, $A \leq_P B$ iff there is some polynomial-time computable function f, such that for all w,

$$w \in A \text{ iff } f(w) \in B.$$

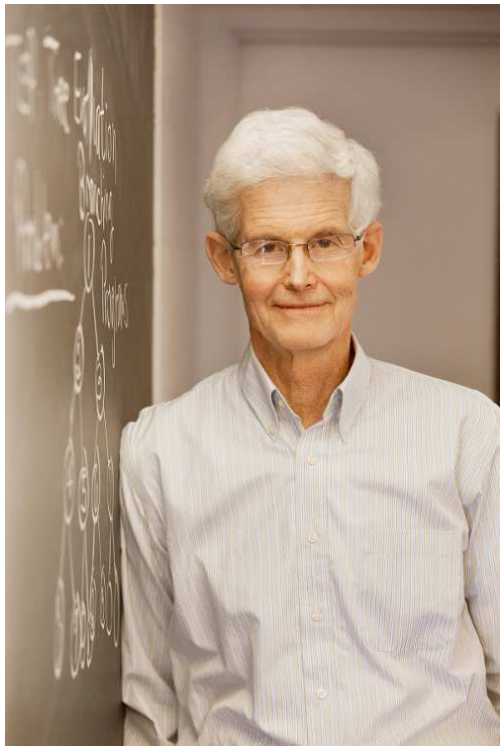
Theorem: If $A \leq_P B$ and $B \in P$ then $A \in P$.

NP-Completeness

B is NP-hard iff every $A \in NP$ is reducible to B in polynomial time.

B is NP-complete iff

1. $B \in NP$, and
2. B is NP-hard.



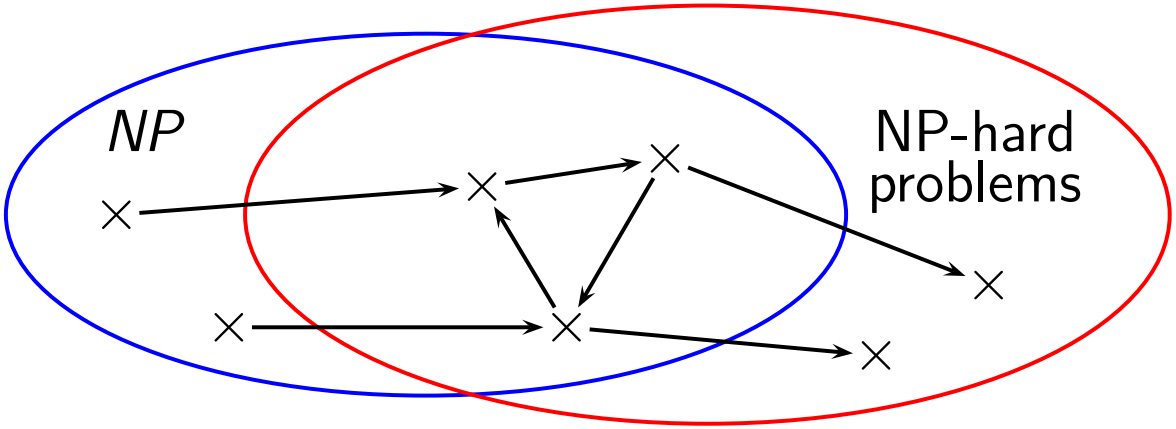
Let us call the class of NP-complete languages NPC.

Our interest in this class owes a lot to this result:

Theorem: If $B \in NPC$ and $B \in P$ then $P = NP$.

NP-Completeness and NP-Hardness

NP-complete problems are the NP-hard problems in *NP*.



Here arrows indicate polynomial time reduction (a transitive relation).

Note: All problems in *NPC* stand or fall together!

NP-Completeness Problems

SAT, SUBSET-SUM, 3COL, LPATH, and HAM, as well as thousands of other interesting and practically relevant problems have been shown to be NP-complete.

This explains the continuing interest in NP-completeness and related concepts from complexity theory.

For such problems we do not know of solutions that are essentially better than exhaustive search.

There are many other interesting complexity classes, including space complexity classes and probabilistic classes.

Dealing with NP-Completeness

Pseudo-polynomial problems (SUBSET-SUM and KNAPSACK are in this class): Unless you have really large data, don't worry; for reasonably-sized sets and numbers, the bad behaviour will not have kicked in yet.

Clever engineering to tenaciously push the boundary: SAT solvers.

Approximation algorithms: Settle for less than perfection.

Live happily with intractability: Sometimes the bad instances just never turn up in practice. Example: Hindley-Milner type inference for functional programming languages.

Computability

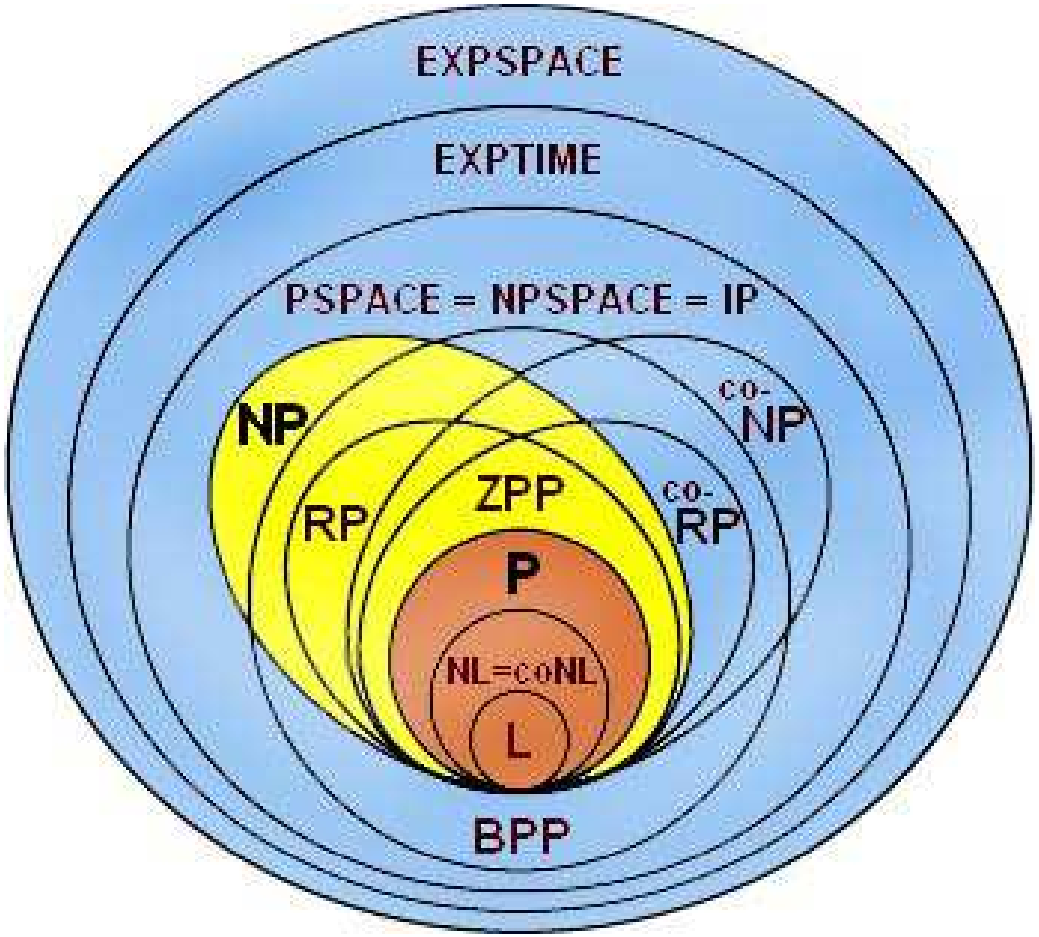
Sometimes we run into more trouble than mere intractability!

There are practically important computational problems that have no algorithmic solution at all!

But that's a topic you'll have to learn about in some other subject, on computability theory.

Complexity Classes: Taxonomy for Problems

There are many other interesting complexity classes, and lots of open questions ...



Open Problems

So we don't know whether $P = NP$, and there are many other unsolved problems of similar type.

We know that $P \subset \text{EXPTIME}$, that is, there are problems that can be solved in exponential time but provably not in polynomial time.

We also know:

$$P \subseteq RP \subseteq NP \subseteq \text{PSPACE} = \text{NPSPACE} \subseteq \text{EXPTIME}$$

But which of these inclusions are strict?
(Some must be; all could be; we just don't know which.)

Appendix: Approximation Algorithms

For intractable optimization problems, it makes sense to look for approximation algorithms that are fast and still find solutions that are reasonably close to the optimal.

Example: Bin Packing

Bin packing is closely related to the knapsack problem.

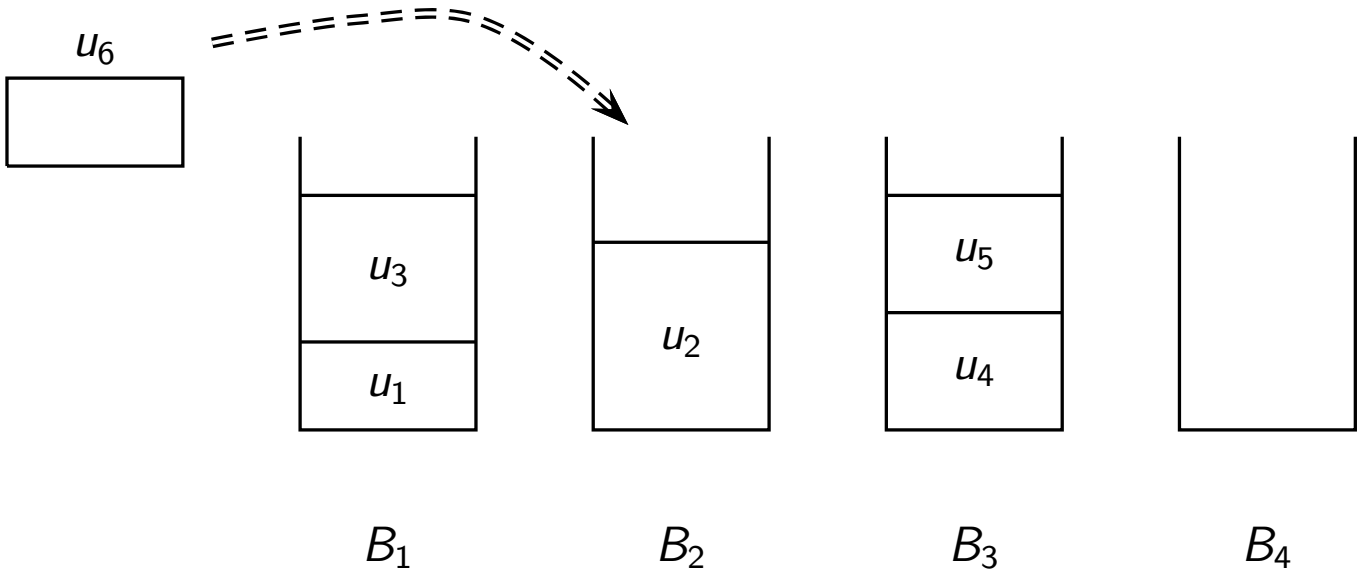
Given a finite set $U = \{u_1, u_2, \ldots, u_n\}$ of items and a rational size $s(u) \in [0, 1]$ for each item $u \in U$, partition U into disjoint subsets $U_1, U_2, \ldots, U_k$ such that

1. the sum of the sizes of items in U_i is at most 1; and
2. k is as small as possible.

The bin-packing problem is NP-hard.

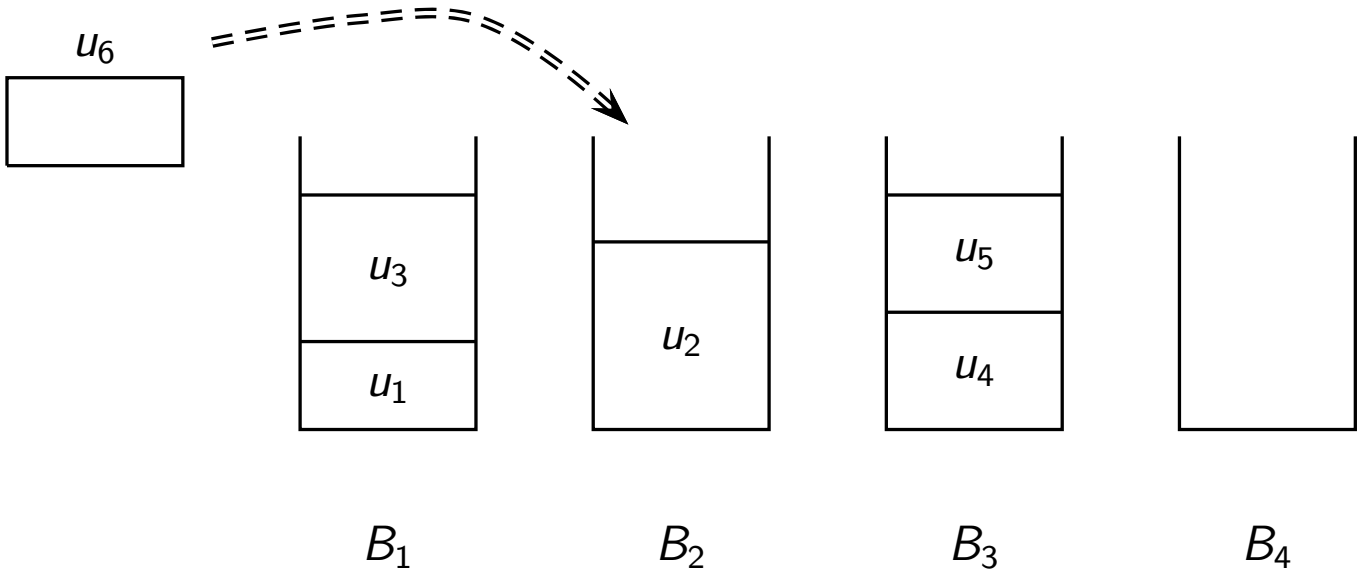
Bin Packing

Each subset U_i gives the set of items to be placed in a unit-sized "bin", with the objective of using as few bins as possible:



An example of using the (fast) "First Fit" heuristic: Use the first bin that has the capacity.

Bin Packing: First Fit as Approximation



Invariant: At most one non-empty bin is half-or-less full. (If there were two such bins, their content would have been placed together.)

Hence the number of bins used with First Fit is never more than twice the minimal number required.

Bin Packing: "First Fit Decreasing"

First Fit behaves worst when we are left with many large items towards the end.

The variant in which the items are taken in order of decreasing size performs better.

The added cost (for sorting the items) is not large.

Detailed analysis shows that bin packing using the "First Fit Decreasing" heuristic guarantees that the number of bins used cannot exceed $\frac{11n}{9} + 4$, where n is the optimal solution.