Politecnico di Torino

# Algorithms and Problem-solving

Paolo Camurati

Edited by Josie E. Rodriguez Condia

## Algorithm

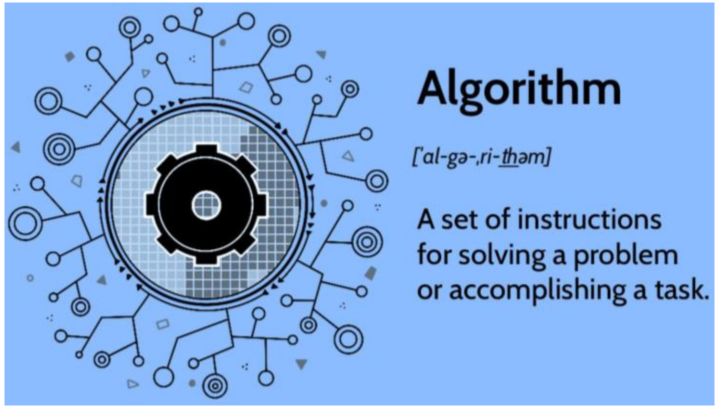
Finite sequence of instructions that:

- solve a problem
- satisfy the following criteria:
  - They receive **input** values
  - They produce **output** values
  - They are **clear**, non ambiguous **and executable**
  - They **terminate** after a **finite number of steps**
- work on data structures

Algorithm: al-Khwarizmi, Persian/Uzbek mathematician  11th cent. a.C.

Muḥammad ibn Mūsā al-Khwārizmī
محمد بن موسى الخوارزمي

## Algorithm


Algorithm
['al-gə-,ri-thəm]
A set of instructions for solving a problem or accomplishing a task.

Finite sequence of instructions that:

- solve a problem
- satisfy the following criteria:
  - They receive **input** values
  - They produce **output** values
  - They are **clear**, non ambiguous **and executable**
  - They **terminate** after a **finite number of steps**
- work on data structures

Algorithm: al-Khwarizmi, Persian/Uzbek mathematician 11th cent. a.C.

## Algorithm vs. procedure

If, for every possible **input value**, **termination** is guaranteed in **a finite** number of **steps**, then

↓

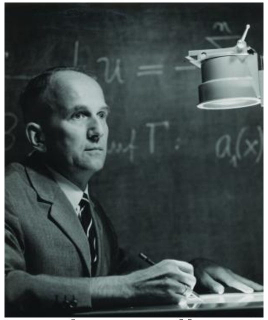
**algorithm**

else

↓

**procedure**

## Collatz's Conjecture (1937)


Lothar Collatz

Let function **f(n)** be defined as:

$$\mathbf{f(n)} = \begin{cases} \mathbf{n/2} \text{ for even } \mathbf{n} \\ \mathbf{3n + 1} \text{ for odd } \mathbf{n} \end{cases}$$

Given any natural number **n**, will function **f(n)** converge to **1** in a finite number of **steps**?

We are unable to answer!

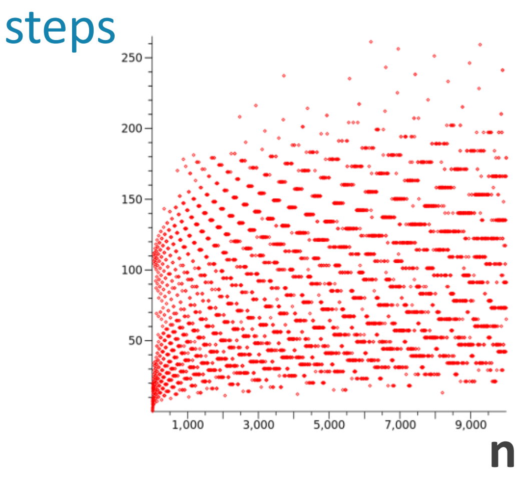
**n** = 11 (odd)

11→34→17→52→26→13→40→20→10→5→16→8→4→2→1

**f(11)** converges in **15** steps

**n** = 27 (odd)

**f(27)** converges in **111** steps



We can't guarantee that **f(n)** converges to **1** for all values of **n**.
We have no guarantee that the process terminates, thus it is a **precedure**, not an **algorithm**.

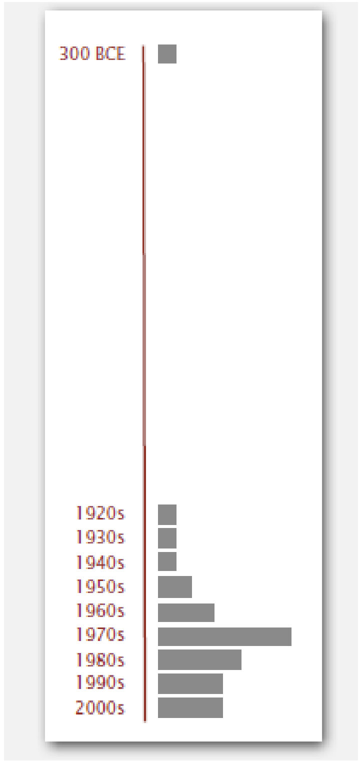
```
#include <stdio.h>
int main() {
  int n;
  printf("Input natural number:  ");
  scanf("%d", &n);

  while (n > 1) {
    printf("%d   ", n);
    if ((n%2) ==1)
      n = 3*n + 1;
    else
      n = n/2;
  }
  printf("%d \n", n);
  return 0;
}
```

Collatz’s conjecture description in c language

# Algorithms in History

- **Egyptian Multiplication** (Rhind and Ahmes papyrus, around 1900 b.C.)
- **Greatest Common Divisor** (Euclid’s algorithm, 4th cent. b.C.)
- **Formalization by Church and Turing** (20th cent., Thirties)
- **Recent developments**



## Egyptian Multiplications

Assumptions:

- No need for multiplication tables
- Enough to **multiply by 2**, thus it is easy to compute the **powers of 2**.

Let *x* and *y* be natural numbers, create 2 columns: the **left** one with **multiples** of **x** (according to the powers of 2), the **right** one **with powers of 2** $\leq$ **y**.

Find **the powers of 2** whose sum is **y**, the result is the sum of the corresponding rows of the **left column**.

**Example:** **x** = 18 **y** = 33

$18 \cdot 33 = 18 + 576 = 594$

**Mathematically:** $33 = 32 + 1 = 2^5 + 2^0$

$18 \cdot 33 = 18 \cdot (1 + 32) = 18 + 576 = 594$

| X | Y |
|---|---|
| 18 | 1 |
| 36 | 2 |
| 72 | 4 |
| 144 | 8 |
| 288 | 16 |
| 576 | 32 |
| Multiples | Powers of two $\leq$ **y** |

## Greatest Common Divisor

The Greatest Common Divisor ***gcd*** of 2 non-zero integers ***x*** and ***y*** is the greatest among the common divisors of ***x*** and ***y***.
Assume that initially ***x*** > ***y***.

Inefficient algorithm based on the decomposition in prime factors:

**x** = $p_1^{e_1} \cdot p_2^{e_2} \cdot \cdot \cdot p_r^{e_r}$ **y** = $p_1^{f_1} \cdot p_2^{f_2} \cdot \cdot \cdot p_s^{f_s}$

gcd(x,y) = $\prod p_i^{\min(e_i, f_i)}$

Example: gcd(96,54) = 6

$96 = 2^5 \cdot 3^1$
$54 = 2^1 \cdot 3^3$
$\mathrm{gcd}(96, 54) = 2^1 \cdot 3^1 = 6$

## Euclid’s Algorithm

**Recursive!** Topic of 2nd year course

Version 1: subtraction

```
if x > y
    gcd(x, y) = gcd(x-y, y)
else
    gcd(x, y) = gcd(x, y-x)
termination:
    if x=y  return x
```

```
gcd(96, 54) =  gcd(42, 54)
gcd(42, 54) =  gcd(42, 12)
gcd(42, 12) =  gcd(30, 12)
gcd(30, 12) =  gcd(18, 12)
gcd(18, 12) =  gcd(6, 12)
gcd(6, 12)  =  gcd(6, 6) = 6
```

Version 2 Euclid-Lamé (1844)-Dijkstra: **remainder of integer division** (%)

```
if $x > y$
    $gcd(x, y) = gcd(y, x\%y)$
termination:
    if $y = 0$ return x
```

```
gcd(96, 54) =  gcd(54, 42)
gcd(54, 42) =  gcd(42, 12)
gcd(42, 12) =  gcd(12, 6)
gcd(12, 6)   =  gcd(6, 0) = 6
```

## Why algorithms?

To solve problems.

Used on computers:

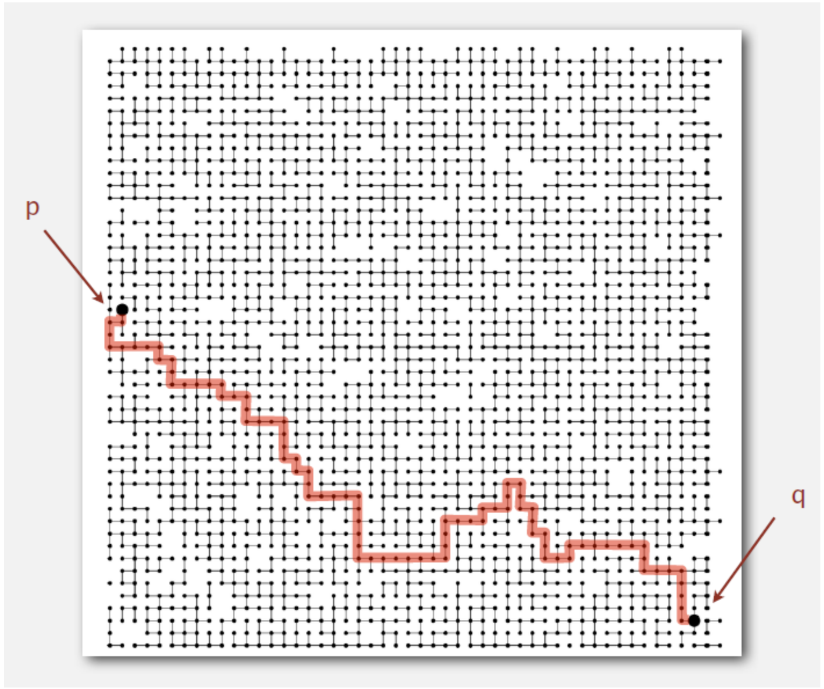
- to increase **speed**
- to process **more data**
- to do something **impossible** otherwise
- to satisfy intellectual curiosity
- to better **program**
- to create **models**
- for **fun** or for **money**.

To solve problems in many domains:

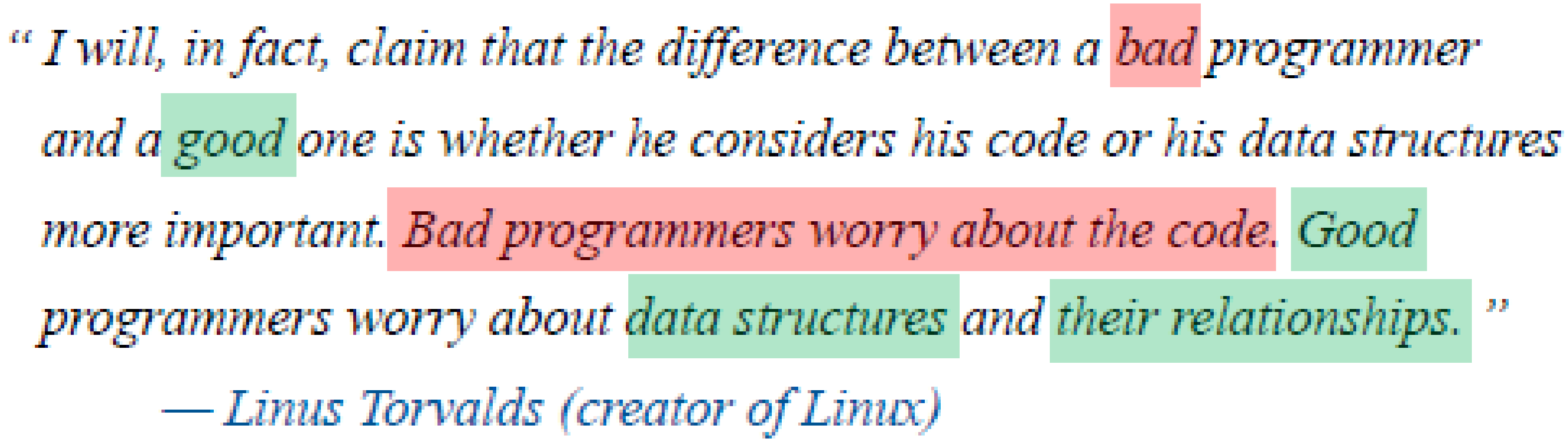
- **Internet**: Web search, packet routing, distributed file sharing
- **Biology**: human genome
- **Computers**: CAD tools, file systems, compilers
- **Graphics**: virtual reality, videographics
- **Multimedia**: MP3, JPG, DivX, HDTV
- **Social Networks**: recommendations, news feed, advertisement
- **Security**: e-commerce, cell phones
- **Physics**: particle collision simulation …

To do something otherwise impossible:

- are the two dark dots connected in this network (**network connectivity**)?



## To better program:



“ *I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.* ”

— *Linus Torvalds (creator of Linux)*

To create models:

- in many sciences computational models are replacing mathematical ones

Closed-form expressions

$$E = mc^2$$
$$F = ma$$
$$F = \frac{Gm_1 m_2}{r^2}$$
$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V(r)\right]\Psi(r) = E\,\Psi(r)$$

```
for (double t = 0.0; true; t = t + dt)
   for (int i = 0; i < N; i++)
   {
      bodies[i].resetForce();
      for (int j = 0; j < N; j++)
         if (i != j)
            bodies[i].addForce(bodies[j]);
   }
```

Mathematical formula　　　　Computational Model

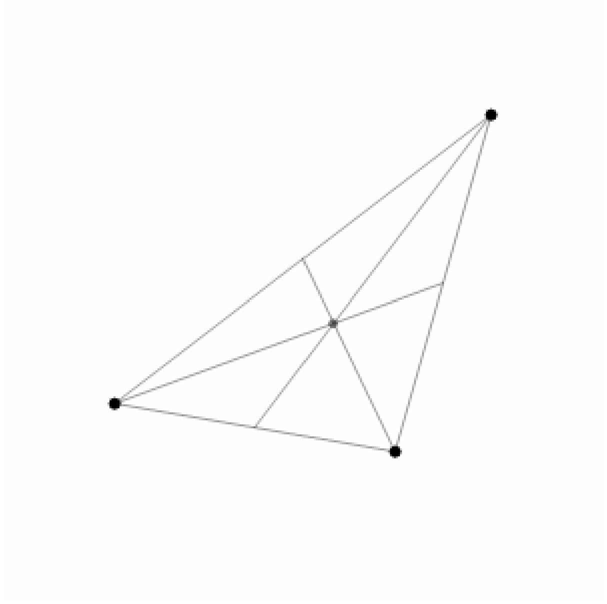
To create models:

- in many sciences computational models are replacing mathematical ones

Three-body problem

$$\ddot{\mathbf{r}}_1 = -Gm_2 \frac{\mathbf{r}_1 - \mathbf{r}_2}{|\mathbf{r}_1 - \mathbf{r}_2|^3} - Gm_3 \frac{\mathbf{r}_1 - \mathbf{r}_3}{|\mathbf{r}_1 - \mathbf{r}_3|^3},$$
$$\ddot{\mathbf{r}}_2 = -Gm_3 \frac{\mathbf{r}_2 - \mathbf{r}_3}{|\mathbf{r}_2 - \mathbf{r}_3|^3} - Gm_1 \frac{\mathbf{r}_2 - \mathbf{r}_1}{|\mathbf{r}_2 - \mathbf{r}_1|^3},$$
$$\ddot{\mathbf{r}}_3 = -Gm_1 \frac{\mathbf{r}_3 - \mathbf{r}_1}{|\mathbf{r}_3 - \mathbf{r}_1|^3} - Gm_2 \frac{\mathbf{r}_3 - \mathbf{r}_2}{|\mathbf{r}_3 - \mathbf{r}_2|^3}.$$

Mathematical formula



# The computational model

- Who or what executes an algorithm?
  - **Human being**
  - **Machine**

- Are there **limits** on the **power** of the machines we can **build**?
- Does a **universal model of computation** exist?

# The computational model

- Who or what executes an algorithm?
  - **Human being**
  - **Machine**

- Are there **limits** on the **power** of the
- Does a **universal model of computat**

The Turin~~g~~ Machine

# The computational model

- Who or what executes an algorithm?
  - **Human being**
  - **Machine**

- Are there **limits** on the **power** of the
- Does a **universal model of computat**

The Turin~~g~~ Machine
**Just joking!**

## The computational model

- Who or what executes an algorithm?
  - **Human being**
  - **Machine**

- Are there **limits** on the **power** of the machines we can **build**?
- Does a **universal model of computation** exist?

The Turing Machine ✓

# The computational model

- Entscheidungsproblem ("decision problem")

(David Hilbert and Wilhelm Ackermann)

UNIVERSITY OF GOTTINGEN
GERMANY

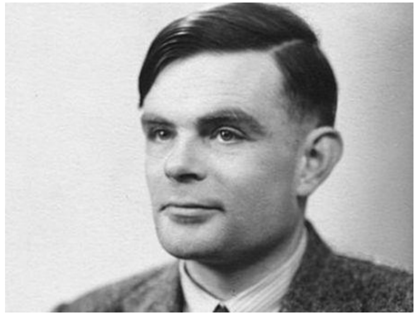
Given a set of mathematical axioms, is there a mechanical process

— **a set of instructions, which today we'd call an algorithm** —

that can always determine whether a given statement is true?

# The Turing Machine

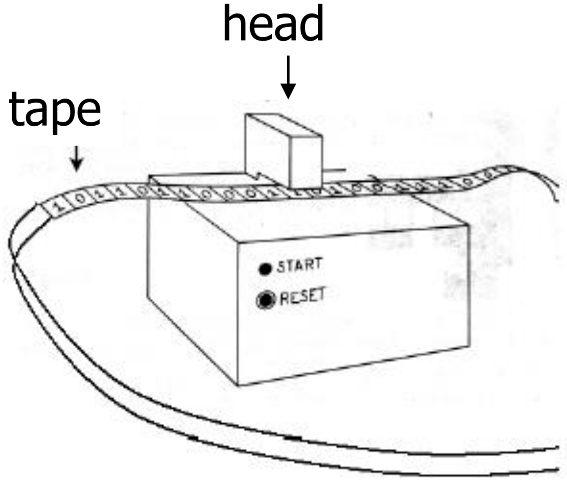
The **Turing Machine** is a model that, given a function ***f*** and an ***input***, when **computation** is over, returns the corresponding ***output***.


Alan Turing

**“If the machine can calculate a function, then the function is computable”.**

It is defined as:

- A **tape**
- A **head**
- An **initial state Q0**
- An **internal state Q1**
- A **program (problem/question)**

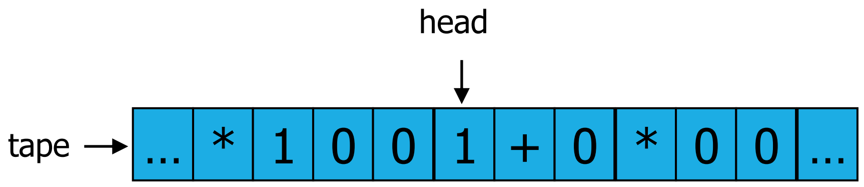


The **tape**:

- **Stores input**, **output** and **intermediate** results
- Has infinite length and is divided in **cells**
- Each **cell** contains a symbol ∈ alphabet

The **head**:

- Points to a **cell** on the tape
- **Reads** or **writes** the current **cell**
- Moves 1 **cell** to the right or to the left



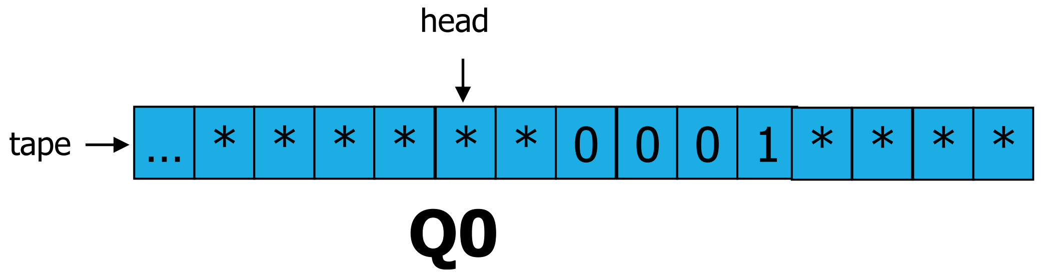
The **internal state** is the configuration of the machine as a function of the current **input** and of the **past** «history».

The machine, as a function of the **current state** and **input**, writes a value on the **tape** and moves the **head** to the left or to the right (program).

## Example:

Machine designed to tell us if the input number is zero:

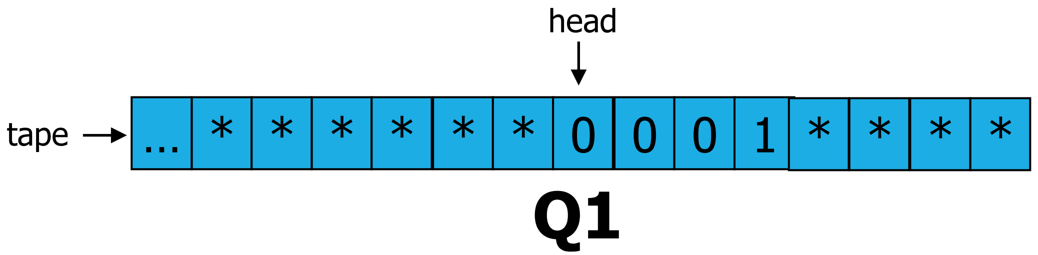
### Case 1:



**Example:**

Machine designed to tell us if the input number is zero:

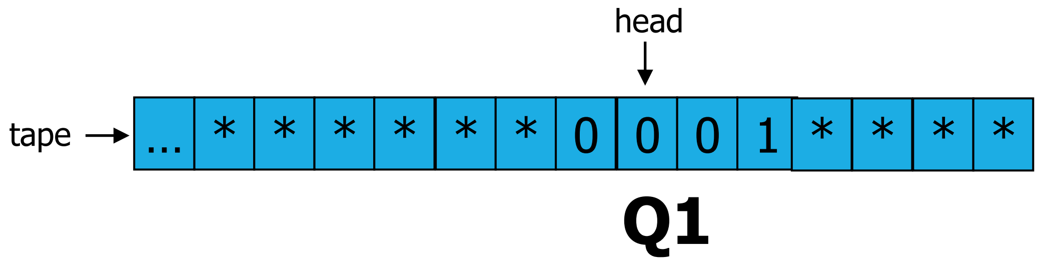
**Case 1:**



**Example:**

Machine designed to tell us if the input number is zero:

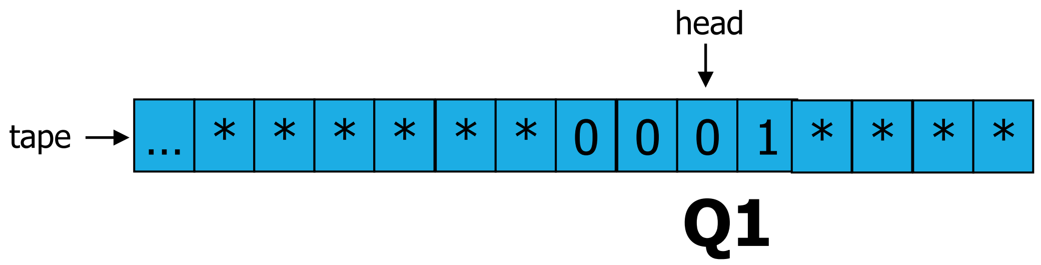
**Case 1:**



**Example:**

Machine designed to tell us if the input number is zero:

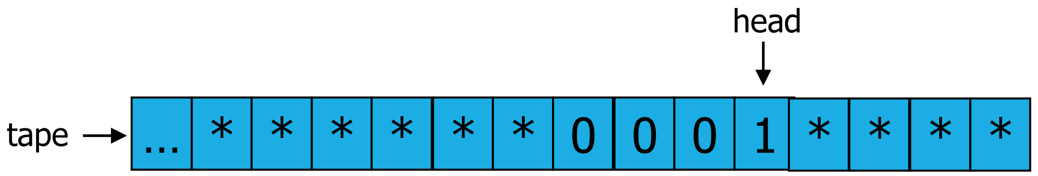
## Case 1:



**Example:**

Machine designed to tell us if the input number is zero:

**Case 1:**



**reject**

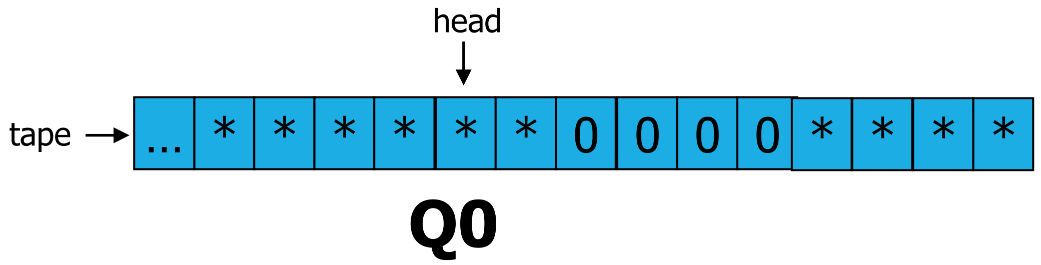
**No, input number is not zero**

**Halt!**

**Example:**

Machine designed to tell us if the input number is zero:

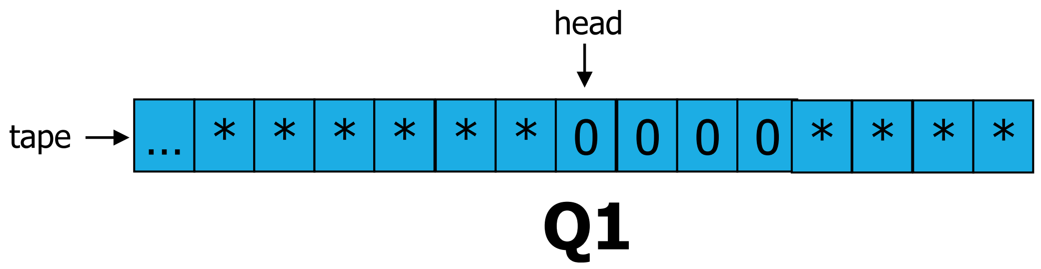
**Case 2:**



**Example:**

Machine designed to tell us if the input number is zero:

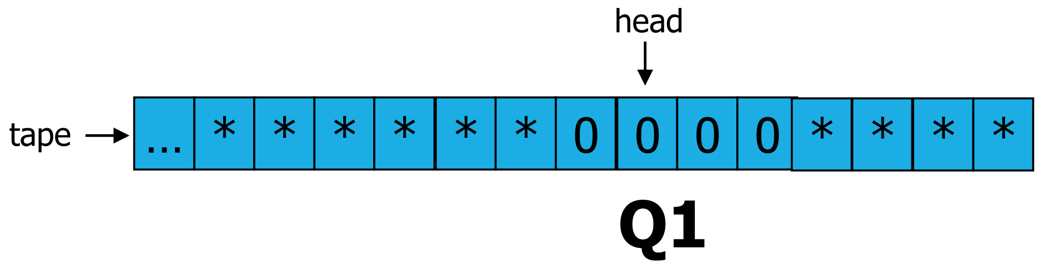
## Case 2:



**Example:**

Machine designed to tell us if the input number is zero:

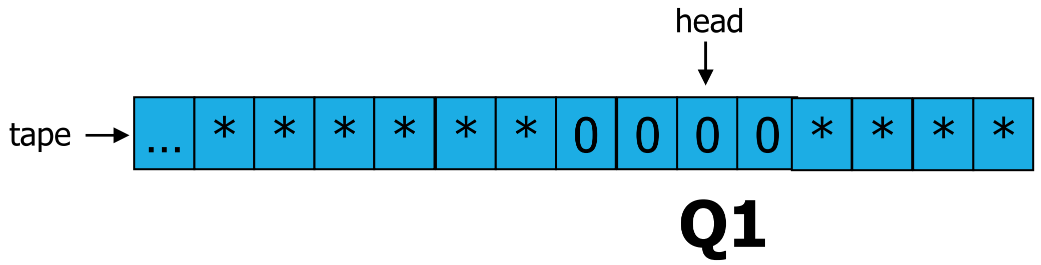
## Case 2:



**Example:**

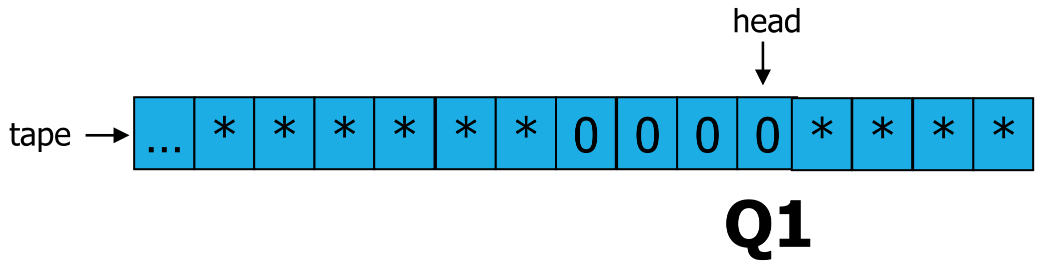
Machine designed to tell us if the input number is zero:

**Case 2:**



**Example:**
Machine designed to tell us if the input number is zero:

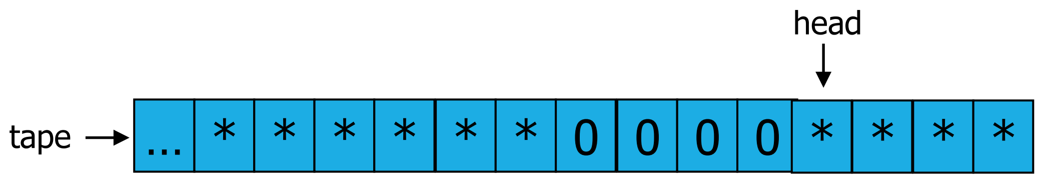
**Case 2:**



**Example:**

Machine designed to tell us if the input number is zero:

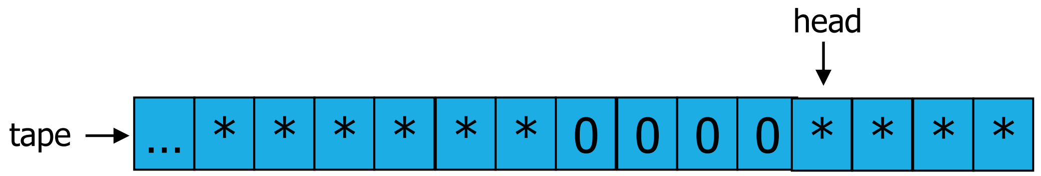
## Case 2:



**Example:**

Machine designed to tell us if the input number is zero:

**Case 2:**



**Accept**

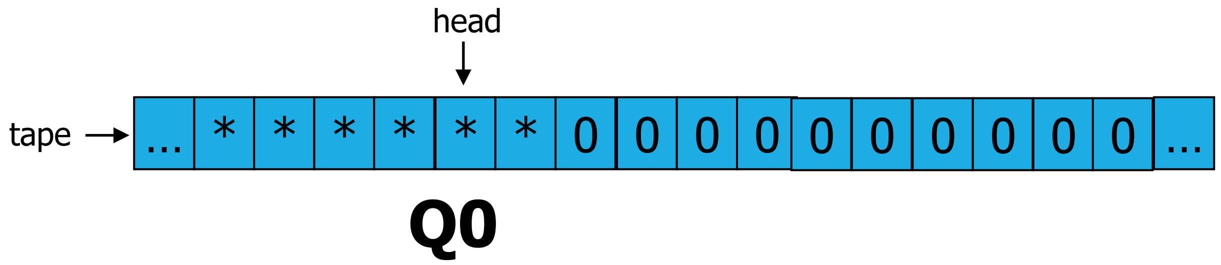
**Yes, input number is zero**

**Halt!**

**Example:**

Machine designed to tell us if the input number is zero:

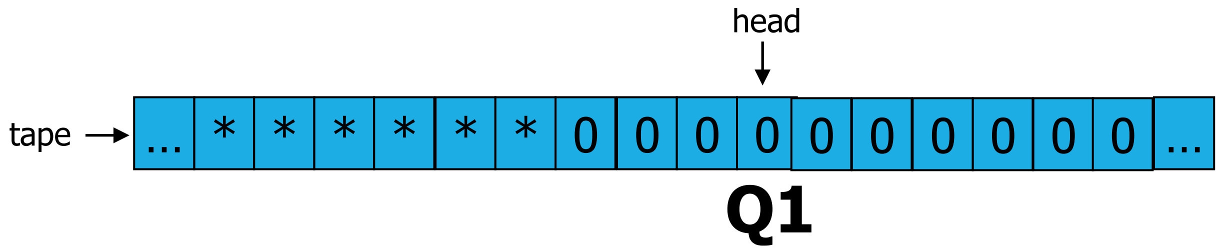
**Case 2:**



**Example:**

Machine designed to tell us if the input number is zero:

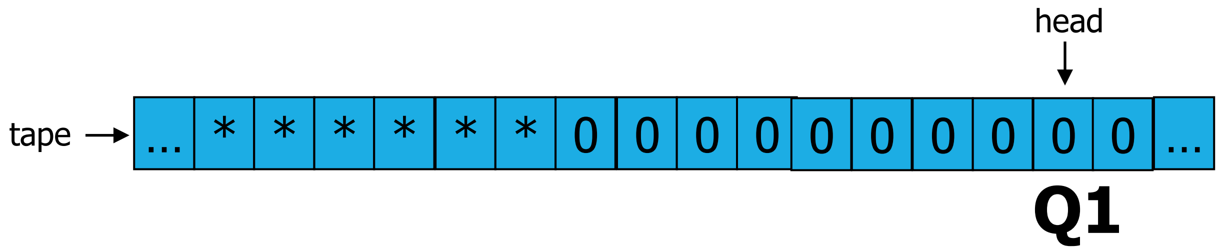
## Case 2:



**Example:**

Machine designed to tell us if the input number is zero:

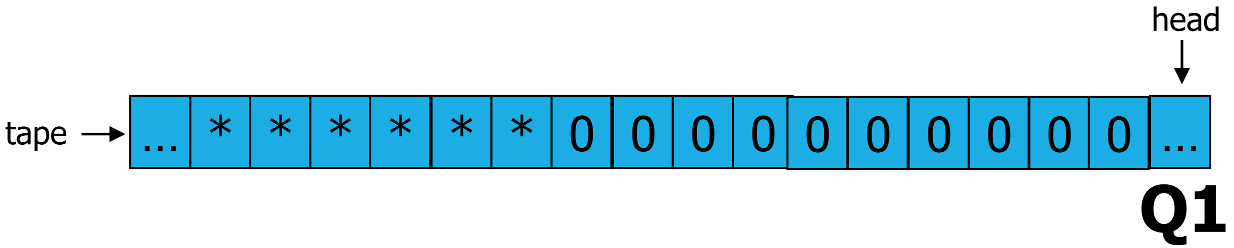
## Case 2:



**Example:**

Machine designed to tell us if the input number is zero:

**Case 2:**



**it falls into an infinite loop and continues reading the tape forever**

## The Church-Turing Thesis(1936)

«*The Turing Machine can compute any function that may be computed by a physically harnessable machine*».

Thesis, not theorem, because it is a statement on the physical world not subject to proof, but in 80 years no counter-examples have been found.

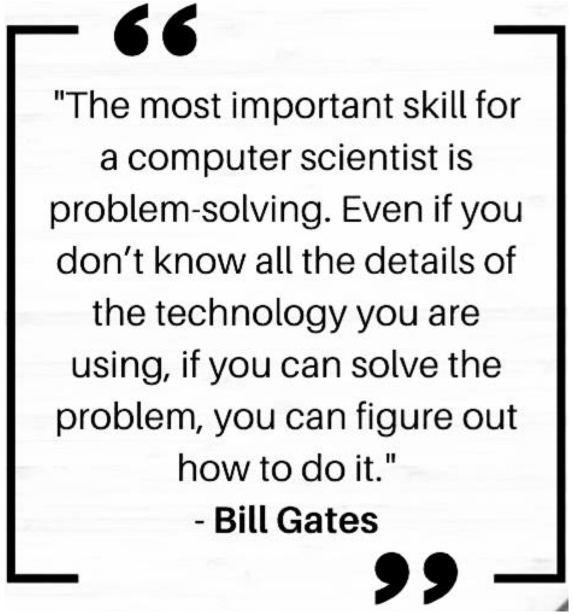
All computational models found so far are equivalent to the Turing Machine.

**Concept** **Real-life machine**

**Universal Turing machine** $\longrightarrow$ **Von Neumann Architecture**

**probabilistic Turing machines!!**

## Problem solving


"The most important skill for a computer scientist is problem-solving. Even if you don't know all the details of the technology you are using, if you can solve the problem, you can figure out how to do it."

- **Bill Gates**

"A problem well put is half solved."
— John Dewey

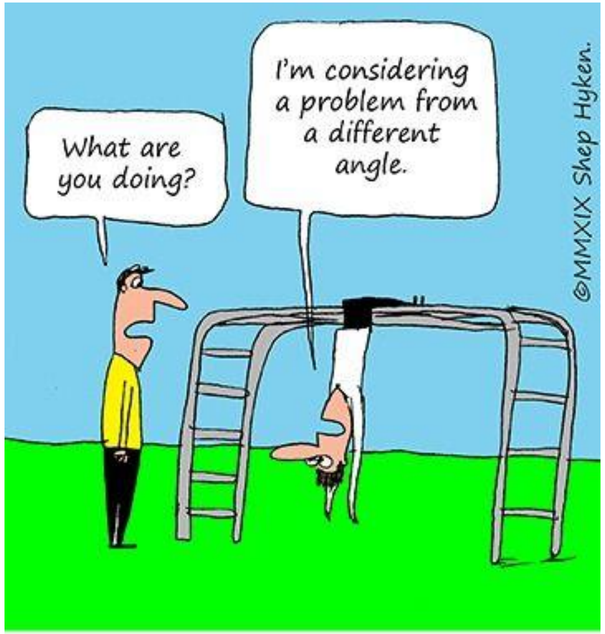
## Problem-solving

It an activity of thought that:

- An **organism** or an **Artificial Intelligence device** put in place to reach a desired condition starting from an initial condition

GPT-3

It is highly creative, design-oriented.



Opposite thinking is a creative way to find an alternative solution to a problem.

## An Approach to problem-solving

1. **Problem analysis**:
   Reading specifications, understanding the problem, identification of the known class of problems the current one belongs to
2. **Methodology identification**:
   selection among known algorithmic paradigms (divide and conquer, incremental, dynamic programming, greedy, etc.)
3. **Approach selection**:
   Selection of the best approach in terms of complexity analysis

4. **Decomposition in subproblems**:
   identification of subproblems and of their interaction in view of a modular approach
5. **Definition of the resolving algorithm**:
   Identification of the sequence of elementary steps, of the data it operates on  and demonstration of correctness
6. **Critical reflection**:
   Identification of critical issues and of possible improvements.

## Computational problems

- **Formal models** associated to a set of questions answered by a computer program processing data
- Set of **infinite instances of a problem**, each being associated with a solution
- **Problem instance:**  problem operating on specific data.

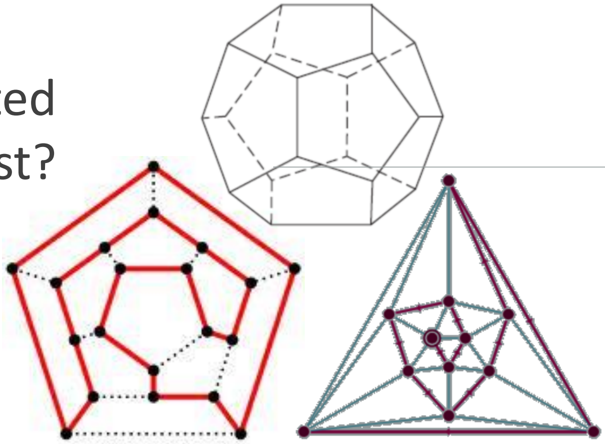
## Types of problems

- **Decision problems:** problems with a **yes/no** answer
  - given 2 integers **x** and **y**, does **x** exactly divide **y**?
  - given a positive integer **x**, is it prime?
  - given a positive integer **n**, do two positive and > **1** integers **p** and **q** exist such that **n = pq**?

**Search problems:** does a **valid solution** exist and which one is it? The solution belongs to a possibly infinite **space of solutions**:

- **Hamilton's game** (1859): in dodecahedron (some say an icosahedron), assigning the name of a town to each vertex, find a path that spans all towns, crossing each town once and only once and returning at the town it started from
- **Graph Theory**: Hamiltonian cycle: given an undirected graph, does a simple cycle spanning all vertices exist? Which one is it?
- which is the k-th prime number
- given an array of integers, **sort** it in ascending order

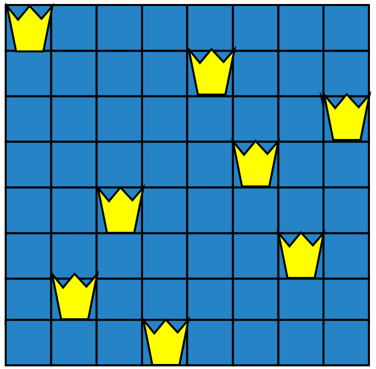
Sir William
Rowan Hamilton



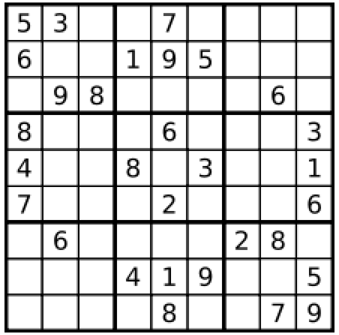
**Verification problems:** given a solution (certificate), make sure that it is really one:

- Does any chess queen threaten any other queen?
- The puzzle on the right complies with Sudoku rules (digits from 1 to 9 on rows, columns and 3x3 squares without repetitions)

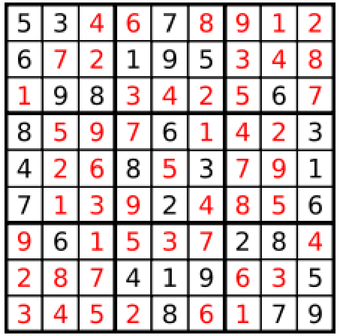
8 queens



Sudoku



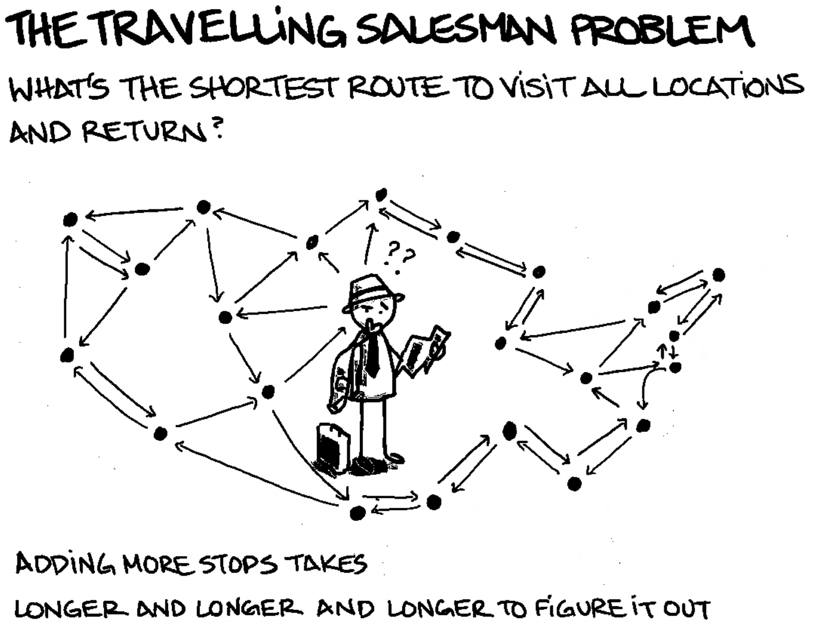
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

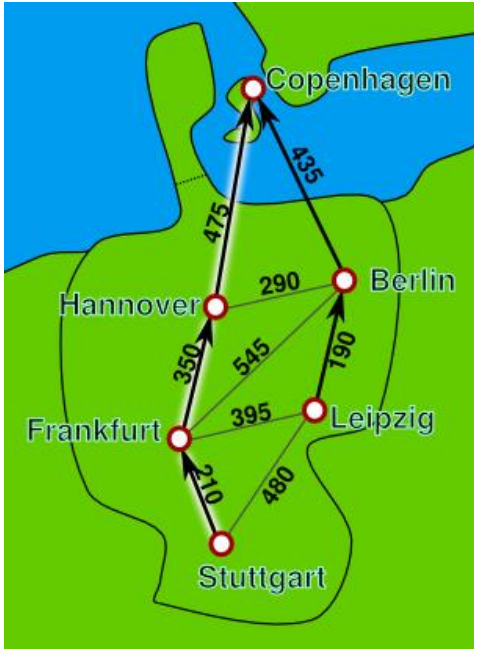


| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 6 | 7 | 8 | 9 | 1 | 2 |
| 6 | 7 | 2 | 1 | 9 | 5 | 3 | 4 | 8 |
| 1 | 9 | 8 | 3 | 4 | 2 | 5 | 6 | 7 |
| 8 | 5 | 9 | 7 | 6 | 1 | 4 | 2 | 3 |
| 4 | 2 | 6 | 8 | 5 | 3 | 7 | 9 | 1 |
| 7 | 1 | 3 | 9 | 2 | 4 | 8 | 5 | 6 |
| 9 | 6 | 1 | 5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

**Optimization problems**: if a solution exists, which one is the **best one**?

- **shortest path:** given a weighted directed graph, which is the shortest simple path, if it exists, between nodes i and j?
- **Travelling Salesman Problem**: shortest Hamiltonian cycle.


THE TRAVELLING SALESMAN PROBLEM
WHAT'S THE SHORTEST ROUTE TO VISIT ALL LOCATIONS AND RETURN?
??
ADDING MORE STOPS TAKES LONGER AND LONGER AND LONGER TO FIGURE IT OUT



Optimization problems may have a decision version if we set a limit on valid solutions:

- **shortest paths:** given a weighted and directed graph, given nodes **i** and **j** and a maximum distance **d**, does a simple path between **i** and **j** exist, whose length is ≤ **d**?

Every **optimization problem** is at least as **hard** as its **decision version**.

## Decision problems

They can be formulated as Yes/NO problems

They may be:

- **decidable** (there exists an **algorithm** that **solves** them)
  - **Example:** determine whether a number is prime

**Algorithm #1:**

try the numbers between **2** and at most **n**, stop:

- either when a factor is found (**n**%**fact**==0, **n** isn't prime)
- or because **fact** becomes **n** (**n** is prime)

```
int Prime(int n) {
  int fact;
  if (n == 1)
    return 0;
  fact = 2;
  while (n % fact != 0)
    fact  = fact +1;
  return (fact == n);
}
```

**Algorithm #2:**

try all numbers between **2** and √**n**, stop:

- as soon as a factor is found (**n%fact**==0, **n** isn't prime)
- or when the loop is over (**n** is prime)

```
int PrimeOpt(int n) {
  int fact, found=0;
  if (n == 1)
    return 0;
  fact = 2;
  while (fact <= sqrt(n) && found ==0) {
    if (n % fact == 0)
      found = 1;
     else
      fact  = fact +1;
  }
  return found;
}
```

Differences between **Algorithm #1** and **#2**? In the maximum number of steps they might execute:

- **Algorithm #1:** at most **n** steps
- **Algorithm #2:** at most $\lceil\sqrt{\mathbf{n}}\rceil$ steps

They differ in their **COMPLEXITY**!

They may be:

- **undecidable** (there is **no algorithm** that **solves them**)
  - given an algorithm **A** and data **D**, both arbitrary, decide whether computation **A(D)** terminates in a **finite number of steps**.

    *(Turing halting problem, 1937)*
  - search for a counterexample to ***Goldbach's Conjecture*** (XVII cent.):

every even integer greater than 2 is the sum of 2 prime numbers **p** and **q**

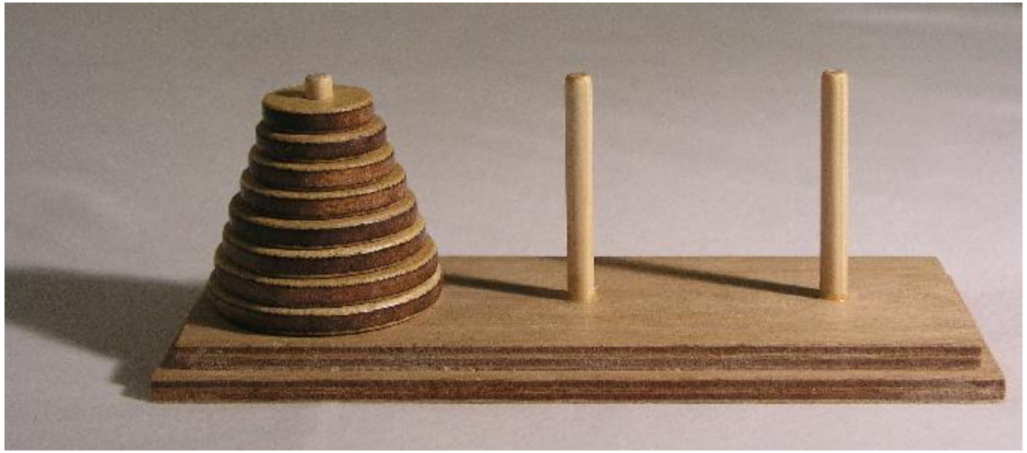
$$\forall\, n \in \aleph, (n > 2) \wedge (n \text{ even})) \Rightarrow (\exists\, p, q \in \wp \in, n = p + q)$$

```
#define  upper 20
```

```
void Goldbach(void) {
  int n = 2, counterexample, p, q;
  do {
    n = n + 2;
    printf("I try for n = %d\n", n);
    counterexample = 1;
    for (p = 2; p <= n-2; p++){
      q = n - p;
      if (Prime(p) == 1 && Prime(q) == 1){
        counterexample = 0;
        printf("%d %d\n", p, q);
      }
    }
  } while (counterexample == 0 && n < upper);
    if (counterexample == 1)
      printf("Counterexample is: %d \n", n);
    else
      printf("Until n= %d none found\n", upper);
  return;
}
```

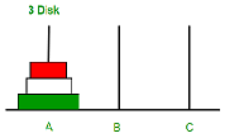
**Decidable** decision problems may be:

- **tractable**, i.e., **solvable** in “reasonable” time:
  - sort an array of n integers
- **intractable**, i.e. **non solvable** in “reasonable” time:
  - The Towers of Hanoi



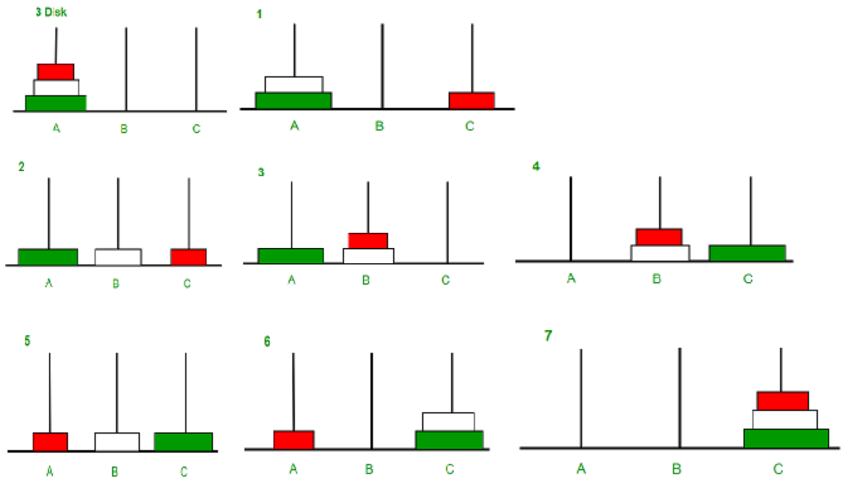
## The Towers of Hanoi (E. Lucas 1883)

- **Initial configuration:**
  - 3 pegs, 3 disks of decreasing size on first peg (A)
- **Final configuration:**
  - 3 disks on third peg (C)
- **Rules (requirements)**
  - access only to the top disk
  - on each disk only smaller disks
- **Generalization: n** disks and **k** pegs.

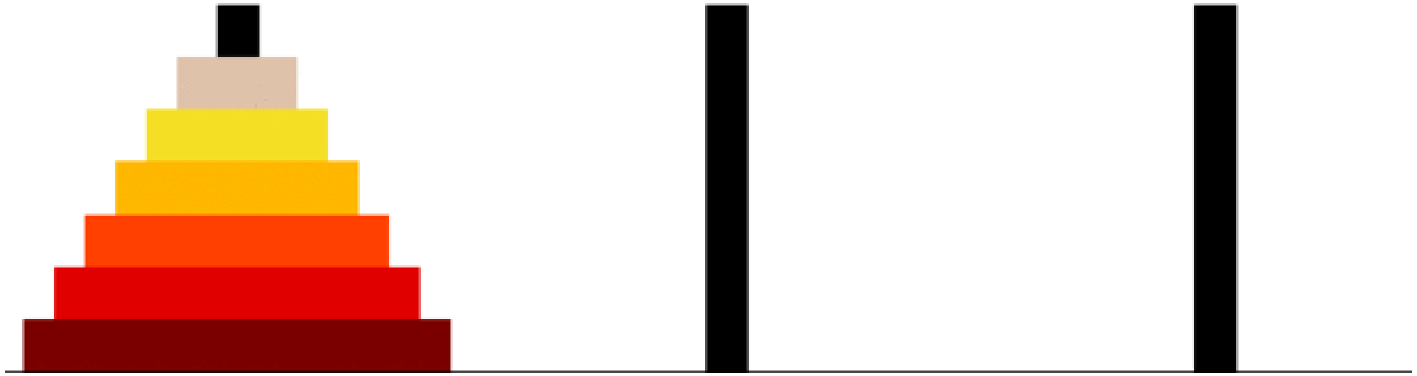


## The Towers of Hanoi (E. Lucas 1883)

- **Initial configuration:**
  - 3 pegs, 3 disks of decreasing size on first peg (A)
- **Final configuration:**
  - 3 disks on third peg (C)
- **Rules (requirements)**
  - access only to the top disk
  - on each disk only smaller disks
- **Generalization: n** disks and **k** pegs.



## The Towers of Hanoi (E. Lucas 1883)



## nxn Chess

Given an chess game, will player white win?

**Initial configuration:**

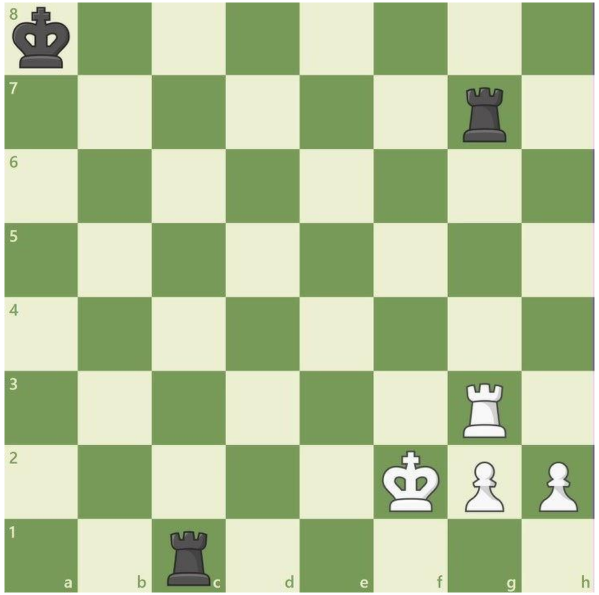
- 10 white pieces, 11 black pieces
- **Rules (requirements)**
  - Chess rules

## nxn Chess

Given an chess game, will player white win?

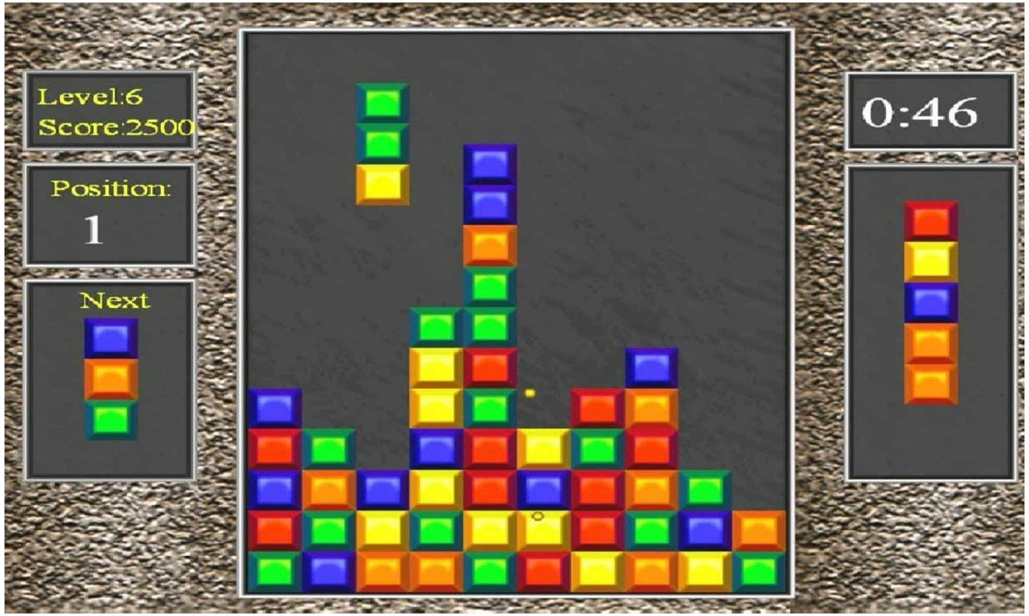
**Initial configuration:**

- 4 white pieces, 3 black pieces
- **Rules (requirements)**
  - Chess rules

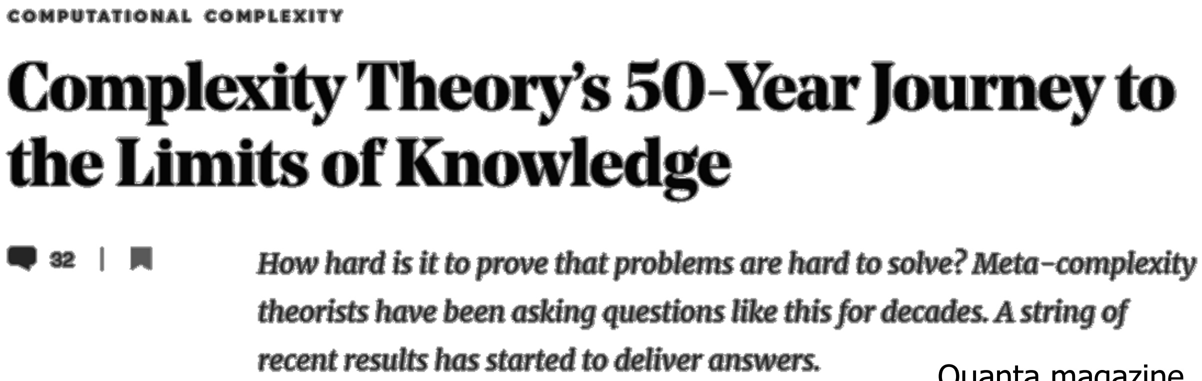


## Tetris

Given an tetris game, will you win?



## Problem/Computational Complexity


COMPUTATIONAL COMPLEXITY

**Complexity Theory's 50-Year Journey to the Limits of Knowledge**

32

*How hard is it to prove that problems are hard to solve? Meta-complexity theorists have been asking questions like this for decades. A string of recent results has started to deliver answers.*

Quanta magazine

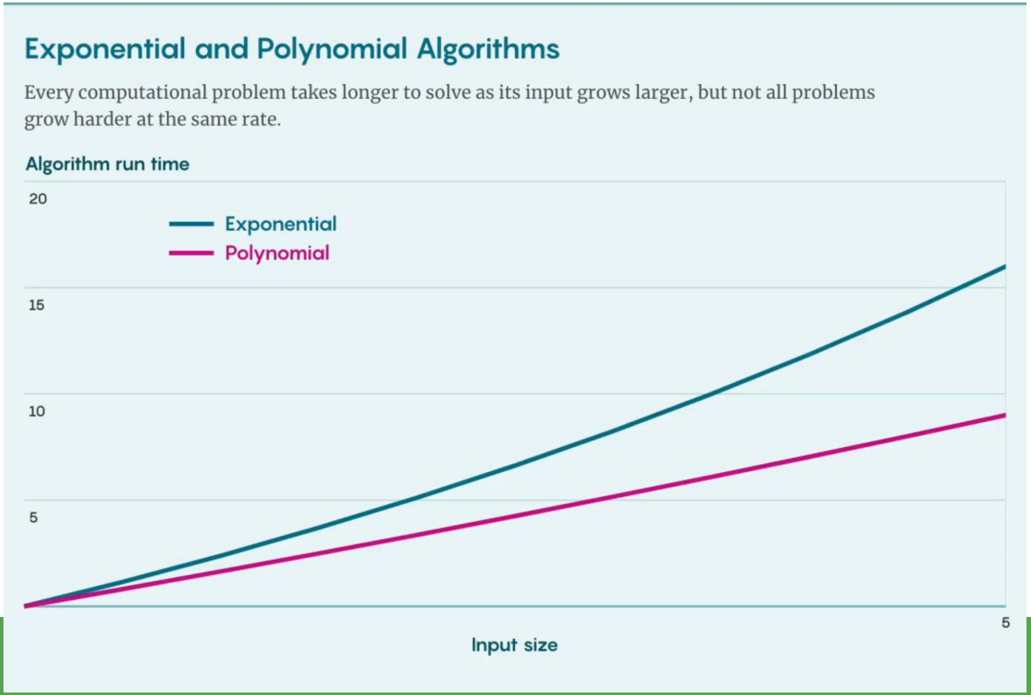
## Computational algorithms

**Function, method, approach** used to solve **problems**

**P** = Polynomial algorithms

**EXP** = Exponential Algorithms

**R** = Finite time algorithms (problems)



## Class P (complexity)

**Decidable** and **tractable** decision problems

$\Leftrightarrow$

$\exists$ a **Polynomial** algorithm that solves them

(Edmonds-Cook-Karp thesis, Seventies)

An algorithm is **Polynomial (Class P)** if, working on **n** data, given a constant **c**>0, it terminates in a **finite number of steps upper-bounded** by $\mathbf{n^c}$.

In practice **c** should not exceed 2.

## Class NP

There **exist** decidable problems for which we have **exponential algorithms**, but we **don’t know any polynomial algorithms**.However we can’t rule out the existence of polynomial algorithms

We have **Polynomial verification** algorithms, to check whether a solution (certificate) is really such

- Sudoku, satisfyability of a Boolean function

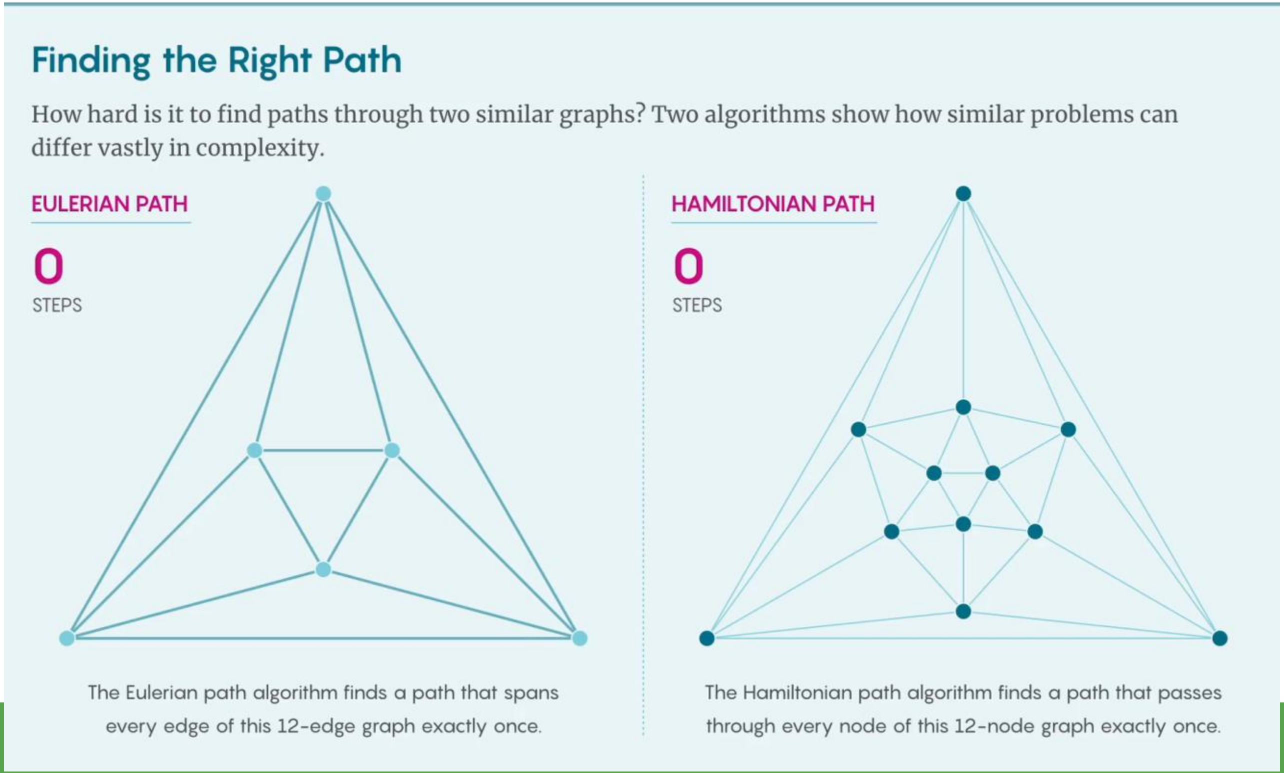
PS: **NP** stands for **Non-deterministic Polynomial** and refers to the nondeterministic Turing machine.

«Guessing, lucky solution algorithms»

## Class NP

**Examples**: Hamiltonian path and Eulerian path problem (Class NP)

*Eulerian path problem (Class P)



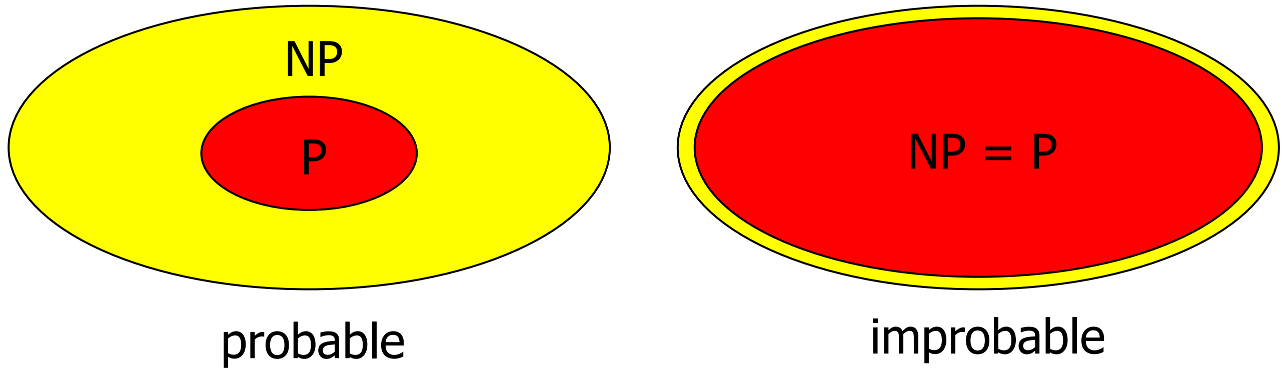
### Finding the Right Path

How hard is it to find paths through two similar graphs? Two algorithms show how similar problems can differ vastly in complexity.

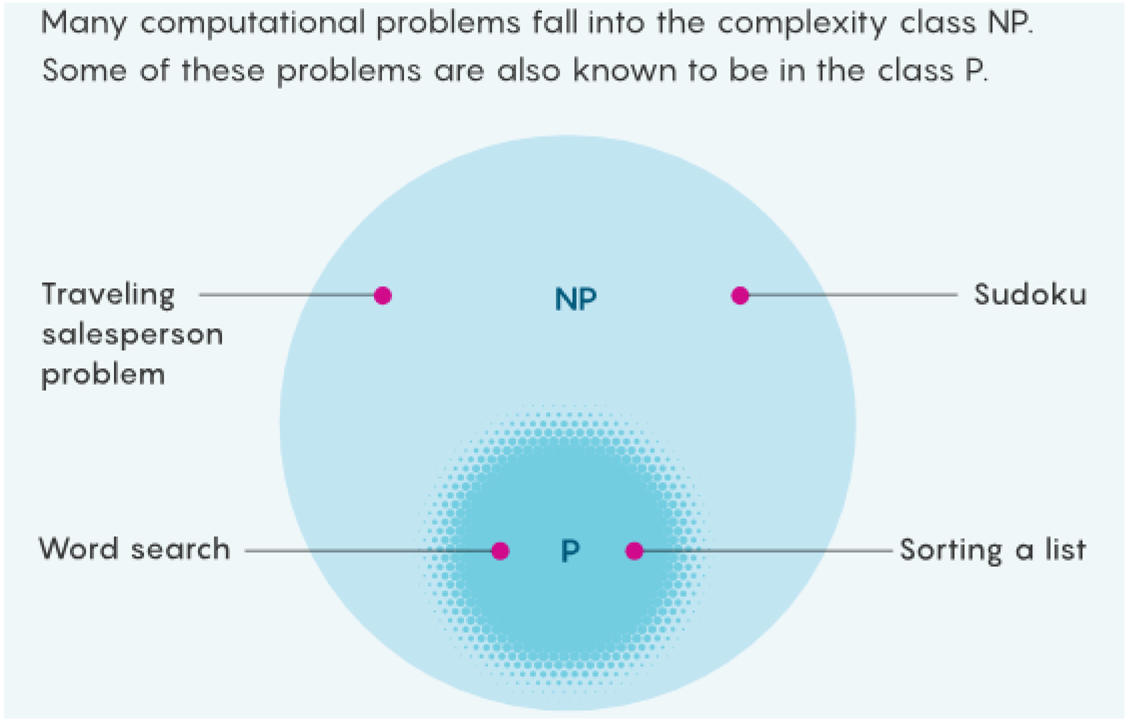
The Eulerian path algorithm finds a path that spans every edge of this 12-edge graph exactly once.

The Hamiltonian path algorithm finds a path that passes through every node of this 12-node graph exactly once.

**P** $\subseteq$ **NP**, but we don't know whether **P** is a proper subset of **NP** or it coincides with **NP**. It is probable that **P** is a proper subset of **NP**.



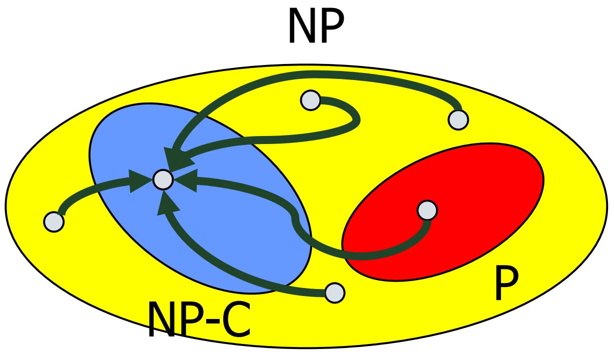
**P** $\subseteq$ **NP**, but we don't know whether **P** is a proper subset of **NP** or it coincides with **NP**. It is probable that **P** is a proper subset of **NP**.


Many computational problems fall into the complexity class NP.
Some of these problems are also known to be in the class P.
Traveling salesperson problem
NP
Sudoku
Word search
P
Sorting a list

# Class NP-C

A problem is **NP-complete** if:

- it is **NP**
- any other problem in **NP** may be reduced to it by means of a polynomial transformation (Cook–Levin theorem)



find a combination of X and Y that makes the expresion True:

**(X** or **Y)** and not **X**

The proof must be checked in polynomial time, but there's no known polynomial-time algorithm

If we find a **polynomial algorithm** for any problem in this class, we could find polynomial algorithms for all NP problems, through transformations

**HIGHLY UNLIKELY!**

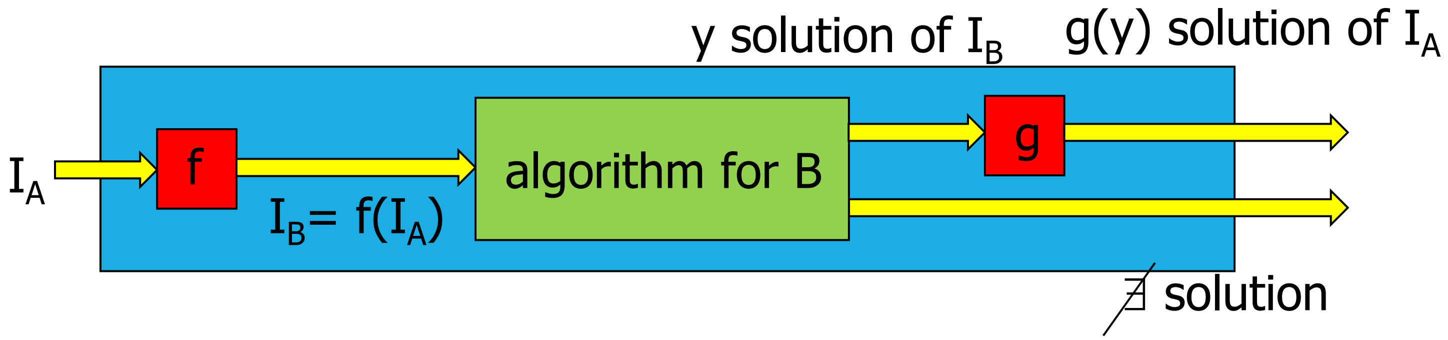
The existence of the **NP-C** class makes it probable that **P** $\subset$ **NP**

Example of **NP-C** problem: satisfiability

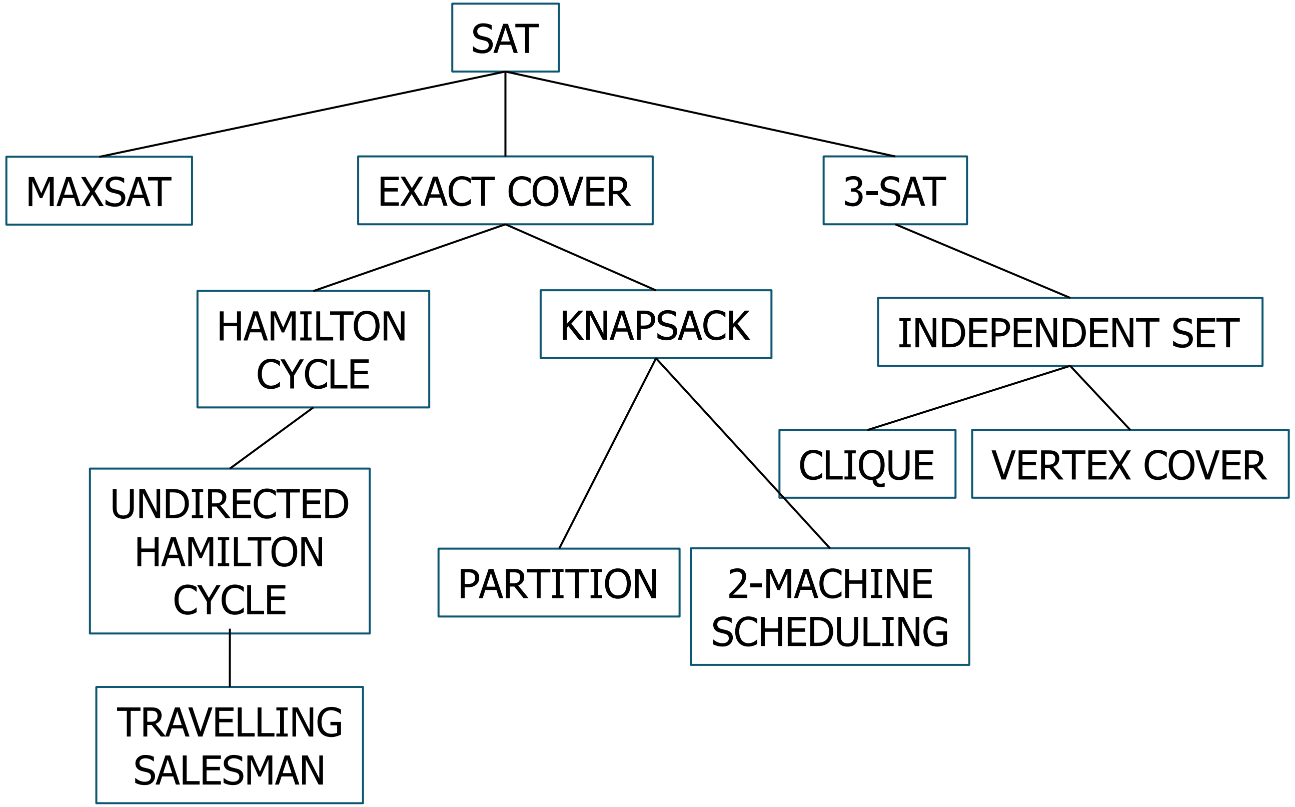
given a Boolean function, find if there exists an assignment to the input variables such that the function is true.

Given 2 decision problems **A** a **B**, a polynomial reduction from **A** to **B** ($A \leq_P B$) is a procedure that transforms every instance $I_A$ of **A** into an instance $I_B$ of **B**:

- with Polynomial cost
- such that the answer to $I_A$ is yes if the answer to $I_B$ is yes

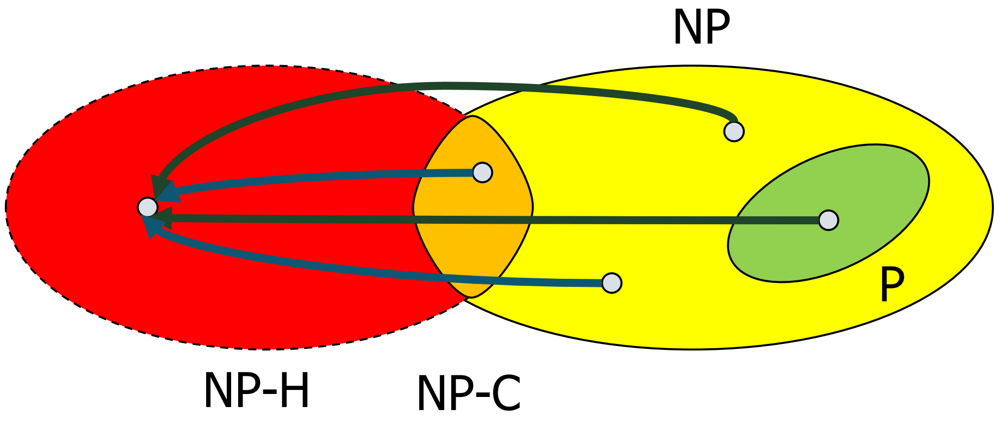


## Examples of reductions



## Class NP-H

- A problem is **NP-hard** if every problem in **NP** may be reduced to it in **polynomial** time (even if it doesn’t belong to **NP***)*
  - Any other problem in **NP** may be reduced to it by means of a **polynomial transformation**

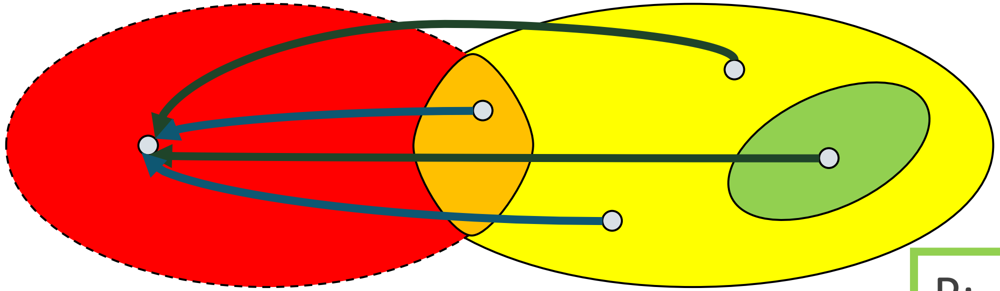


**P:** Polinomial

**NP:** Non-d. Polinomial

**NP:** Non-d. Polinomial C.

NP:
- Factorization
- Graph isomorphism



NP-H
- Matrix Permanent

NP-C:
- Satisfiability
- Hamilton cycle
- Clique

P:
- Graph connettivity
- Primality
- Determinant

# Examples of NP-H problems

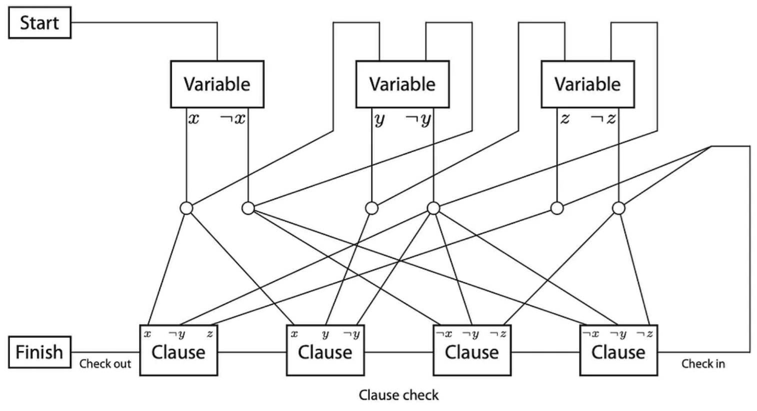
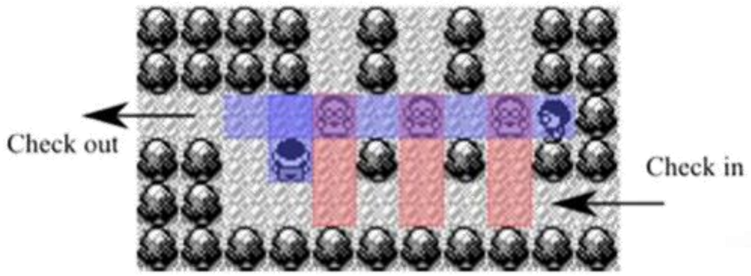
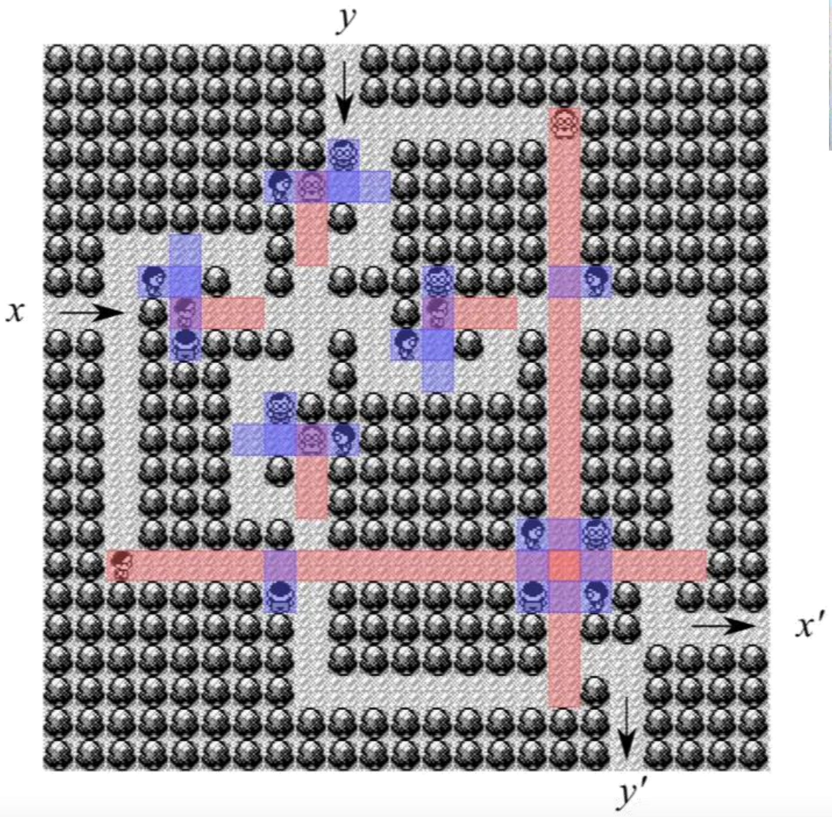


Figure 1: General framework for NP-hardness

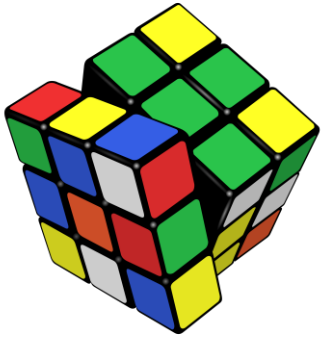




Aloupis, Greg, et al. "*Classic Nintendo games are (computationally) hard.*" ***Theoretical Computer Science*** 586 (2015): 135-160.

# Question?

- NP-hard?
- NP-C?
- NP?
- P?



- NP-hard?
- NP-C?
- NP?
- P?

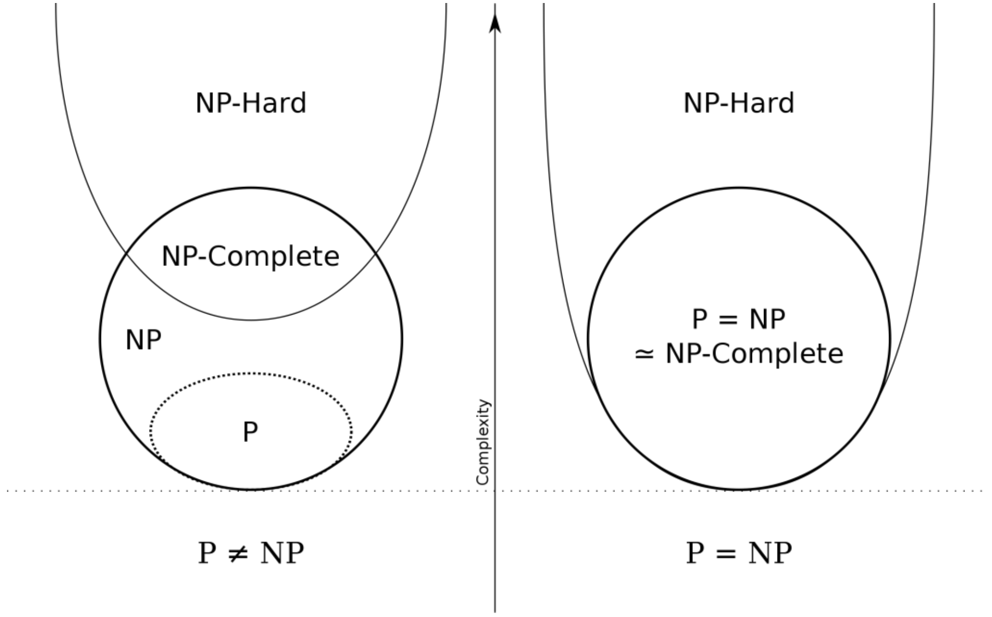
- NP-hard?
- NP-C?
- NP?
- P?

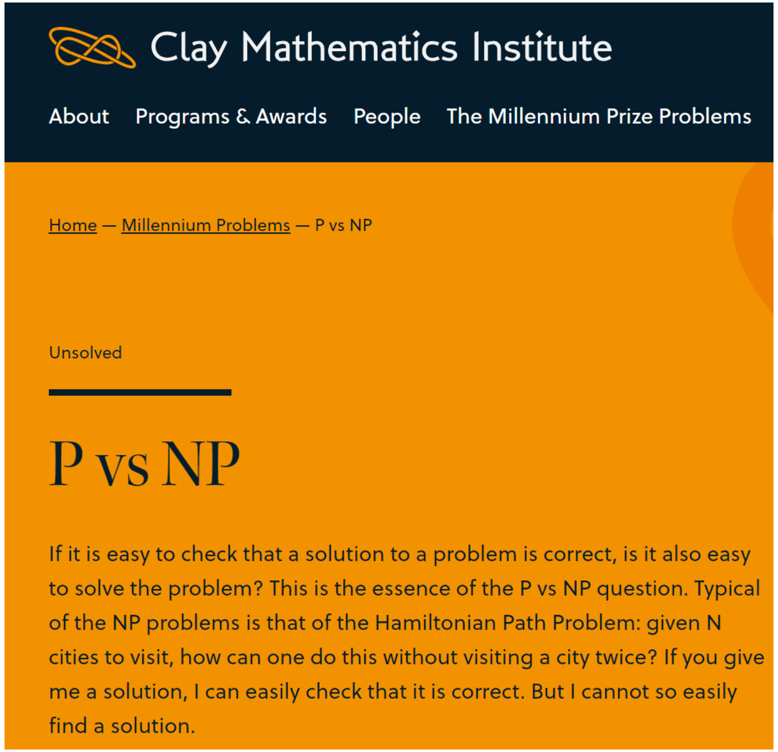
- NP-hard?
- NP-C?
- NP?
- P?

- NP-hard?
- NP-C?
- NP?
- P?

# P versus NP problem (Stephen Cook)

is a major unsolved problem in theoretical computer science




Clay Mathematics Institute

About Programs & Awards People The Millennium Prize Problems

Home — Millennium Problems — P vs NP

Unsolved

P vs NP

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.

## Problem-solving in this course

- **Elementary** problem-solving
- Problems in class **P**
- Classification criteria:
  - Usual approach: based on **data structures** and/or **control statements** used
  - Alternative approach closer to the user and less close to the programmer based on:
    - Problem type
    - Application domain: type of information **items** and problem class
    - Used **data structures**
    - **Algorithmic strategies**.

## Types of Problems in this Course

- **Decision:**
  - Problems whose answer for each instance **is binary**.

Example: given a natural number ***n*** *> 1*, is ***n*** prime?

- **Search:**
  - The answer for each instance is a string of bits that represents an information item.

Example: given **n** distinct data, identify among the **n!** permutations the one that satisfies an ordering relation (< or >). **This is the sorting problem.**

- **Verification:**
  - Given an instance and a supposed solution (certificate), prove that the certificate is indeed a solution for the instance.

  Example: let ***f*** be a **Boolean function** and given a variable assignment, prove that ***f*** is **1** for that **assignment**

- **Selection:**
  - **subcase of verification:** given solutions and acceptance criterion, partition solutions in 2 subsets: solutions that satisfy the criterion and solutions that do not satisfy it

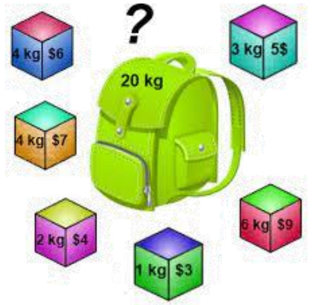
  Example: given the transcripts of records of a group of students, **identify all and only** those students whose **average grade is above a given threshold**.

**e.g., >18**

- **Simulation**:
  - Interactive **representation of reality** based on a computational model of a system, developed to analyze its operation

  Example: given **n counters** as a sequence of **customer arrivals**, each customer requiring attention for **a known time**, estimate **waiting time**
- **Optimization**:
  - Given an instance at a  **cost/gain** function, **select among several solutions the one** whose **cost** is **minimum** or whose **gain** is **maximum**.

  Example: given a **knapsack** with **known capacity**, given **objects** with **weight** and **value:**
  find the subset of objects that may be contained
  in the knapsack with **maximum value** (knapsack problem).



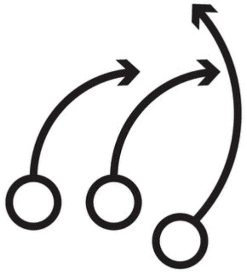
## Other problem classification criteria

- Based on the application domain:
  - numeric
  - mathematical
  - text processing
  - non-numerical data processing
  - etc.
- Based on the **nature** of primitive data:
  - **scalar** and/or **aggregate** data
  - **vector** data.

- Based on the **types** of data:
  - numbers (**integers INT** or **reals Float/Double**)
  - characters (**characters Char** or **strings**)
  - Abstract Data Types (**SYMB**)
- Based on the **statements** used:
  - Elementary
    - conditional statements (**if, switch**), simple or nested iterative statements (**for, while**)
  - Advanced statements/constructs and/or functions.

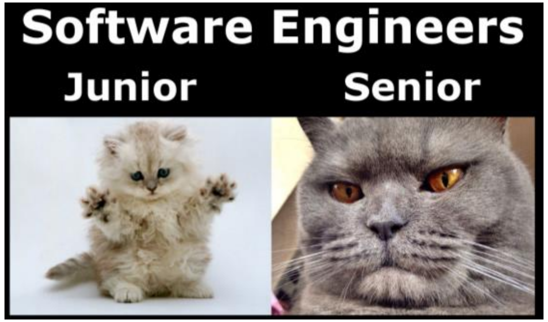
## Steps towards the solution

- Non ambiguous and complete **problem identification**:
  - requires **analysis** and possibly completing the **specifications**
- Building a **formal model** for the problem
- **Definition** of a **resolution algorithm** and **complexity analysis**
- **Algorithm encoding** in a programming language (e.g., *C language*)
- **Validation** of the algorithm and of its **implementation** on significant instances.



## Algorithm = Strategy

- **Selecting an algorithm** is often selecting the **best strategy** to solve a problem
- Selection is **based on** knowledge of:
    - **Elementary statements/constructs** and of the **library functions** supported by the C language
    - **Language statements** (data types, conditional statements, iterative statements)
    - Algorithms known from the **literature**
    - Past **experience** on different kinds of problems.



## Strategy

- **Most problems** solved by **programs** consist in **processing input data** to produce **output data**
- The most important step is **processing**. It requires:
  - The identification of data (intermediate results)
- Data may be scalar and/or aggregate
  - The formalization of the steps (operations) needed to evaluate intermediate results (starting from other data):

**Intermediate steps** are often formalized in terms of **conditional** and/or **iterative** statements.

They can later be **modularized** by means of **functions**.

## In pratice!

- **Experience is a fundamental requisite** (as in many other disciplines) to choose a **good resolution strategy**:

  *An algorithm (a program) is a design!*
- Knowledge **of problems solved in the literature** is a good starting point:
  - **wrong** attitude: solve a problem as if it were seen for the first time, without realizing that work has already been done
  - **right** attitude: one should know and understand underlying theory and be able to apply it.
- Use of **available web resources**:
  - **wrong** attitude: copy without understanding
  - **right** attitude: take on a problem, then discuss with colleagues.

Sometimes, when no other tools are available, **a good starting point** is to analyze the solution to the problem “**by hand**” or “**on paper**”.

## Data Structures

- **The choice** of the **data structures** depends on:
  - The nature of the problem, **the input data**, the requested results
  - The **algorithmic choices**
- Choice of a data structure = deciding which (**what type**) and how many **variables** are needed to **store** input, intermediate and output **data**
- Sometimes the data structure may be selected before defining the algorithm, but often it is necessary to consider data and algorithm at the same time.

## Choice of the data structure

- It consists in:
  - **identifying the types** of information items to represent (input, intermediate and output data): **numbers** (integers or reals), **characters** or strings, **struct**
  - deciding whether it is necessary to collect data **in arrays** or **matrices** (countable aggregates) or **scalar data** (or struct) are enough
- Some problems may be solved with a few statements (conditional constructs) on scalar data.
- **Many problems need iterations on data**.

## Iteration-based Problems and Arrays

- An **iteration-based problem** **doesn't need an array** when it is enough to work on the generic i-th data and there is no need to «remember» previous values. Scalar or aggregate data are enough.

Example: summing up numbers, searching for the maximum value

- An **iteration-based problem** **needs an array** if it necessary to collect and store all data before processing them. Vector data are required.

Example: input data and output, them in reverse order

## Choice of the Algorithm

- **Selecting the algorithm** (conditional and/or iterative statements) may be **very simple** (for example when suggested directly by the problem)

Example: numeric and mathematical problems

- Else **selecting an algorithm** may be a true «**design**», comparing strategies, complexity, evaluating pro’s and con’s

Example: activity planning based on optimization criteria.

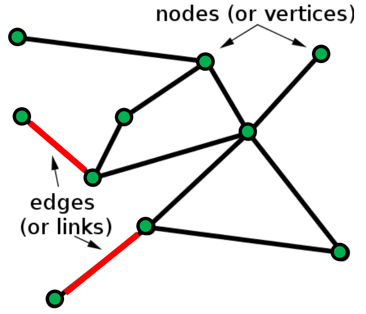
## The Graph theory

Graphs stand or fall by their choice of nodes and edges.
– Watts & Strogatz

## The Graph

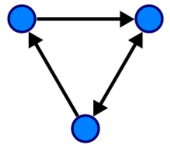
Definition: **G = (V,E)**

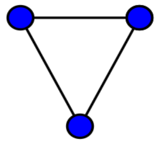
- **V**: finite and non empty set of **vertices** (containing simple or compound data)
- **E**: finite set of **edges**, that define a binary relation on **V**



Directed/Undirected Graphs:

- Directed (*digraph*): edge = **ordered** pair of vertices (**u, v**) ∈ E and **u, v** ∈ V
- Undirected: edge = **unordered** pair of vertices (**u, v**) ∈ E and **u, v** ∈ V

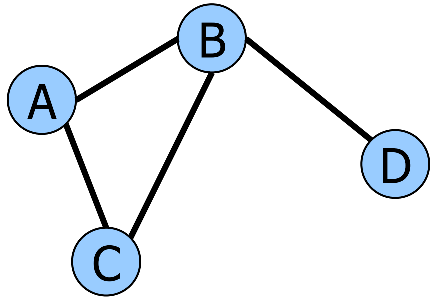




## Vertex-cover

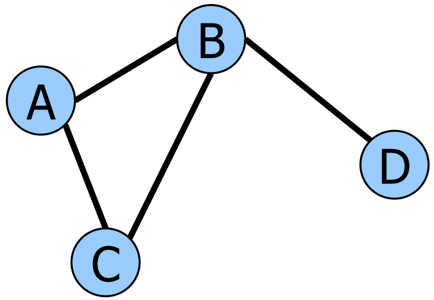
Let **G = (V, E)** be an **undirected graph**, a **vertex-cover** is a subset of vertices **W** $\subseteq$ **V,** such that for all the edges **(u,v)** $\in$ E either **u**$\in$**W** or **v**$\in$**W**

Example

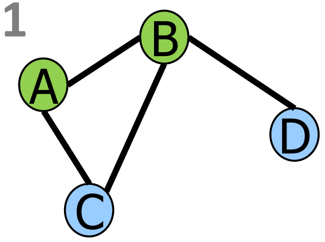


**Target:** find the set of vectices [A, B, C, D] that ensures a **vertex-cover**

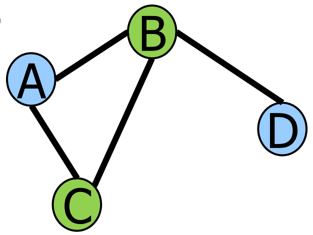
## 7 **vertex-cover** solutions



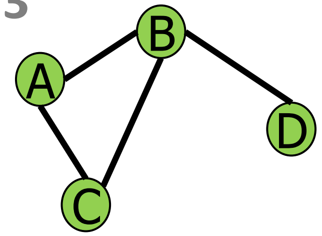
1



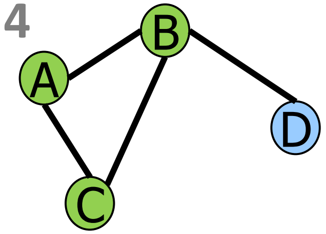
2



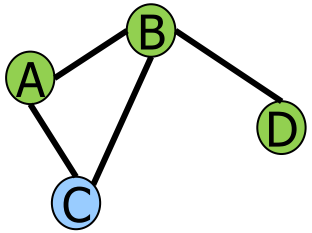
3



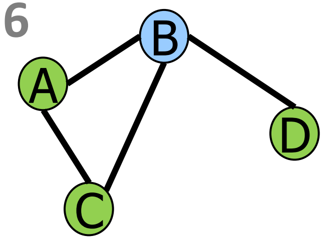
4



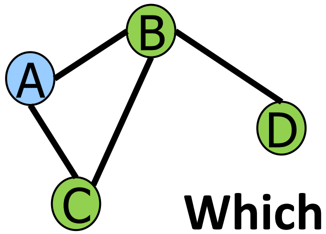
5



6



7

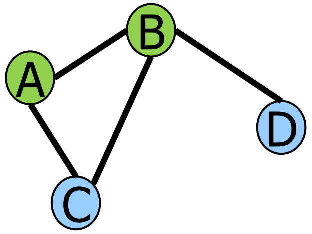


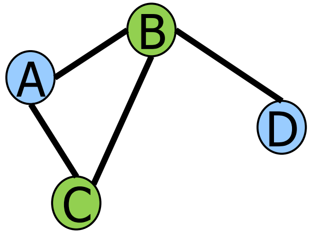
**Which one is the optimal?**

- **Search problem**: find a **vertex-cover**
- **Optimization problem:** find a minimum-cardinality **vertex-cover**
- **Decision problem:** does a **vertex-cover** whose cardinality is $\leq$ **k** exist?

**Decision problem** with **k** = 2

2 solutions





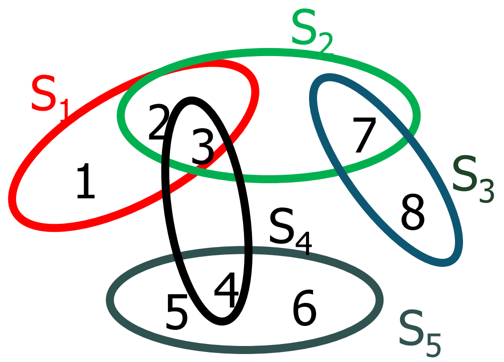
## Set-cover

**Decision** problem:

- let **U** be a set of elements
- let **k** be an integer in the range $\mathbf{1} \leq \mathbf{k} \leq \mathbf{n}$
- $S_1, S_2, \ldots, S_n$ is a collection of subsets of **U**

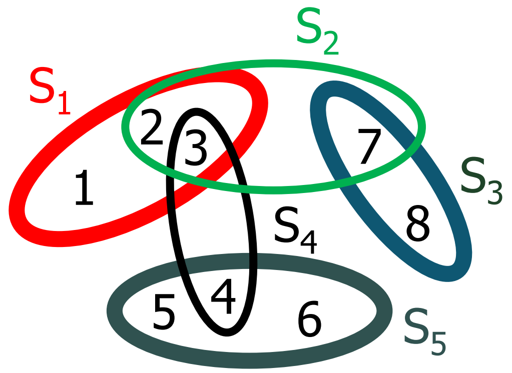
Does a collection of **at most k** subsets exist whose union is **U**?

## Example 1



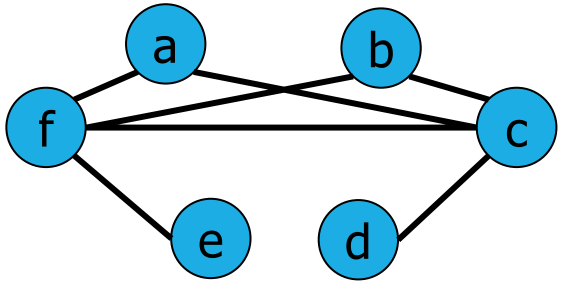
$U = \{1,2,3,4,5,6,7,8\}$
$S_1 = \{1,2,3\}$
$S_2 = \{2,3,7\}$
$S_3 = \{7,8\}$
$S_4 = \{3,4\}$
$S_5 = \{4,5,6\}$

Solution for **k** = 3



# Example 2

**Decision problem**: let **G = (V, E)** be an undirected graph,



**G** = (V, E)

**V** = $\{$a,b,c,d,e,f$\}$

**E** = $\{$(a,c),(a,f),(b,c)
(b,f),(c,d),(c,f)(e,f)$\}$

Does a **vertex-cover** with cardinality $\leq$ **2** exist?

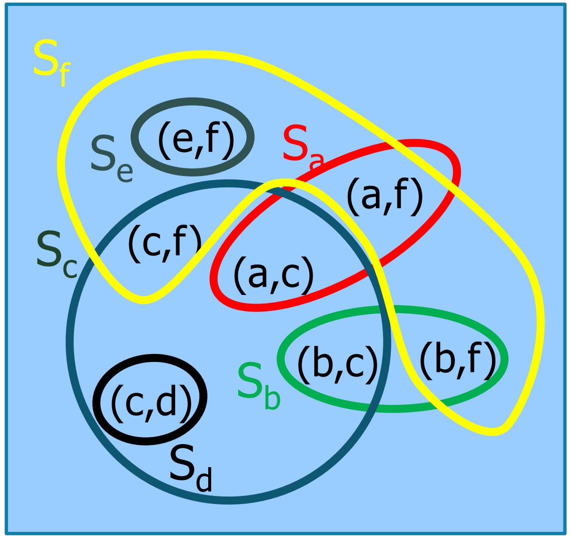
## Example 2

**vertex-cover ≤P set-cover**

Create a **set-cover** decision problem with:

**U = E** and

**$S_i$**= {**edges** that insist on **vertex** i}



U = {(a,c),(a,f),(b,c)
(b,f),(c,d),(c,f)(e,f)}

$S_a$ = {(a,c),(a,f)}

$S_b$ = {(b,c),(b,f) }

$S_c$ = {(a,c),(b,c),(c,d),(c,f)}

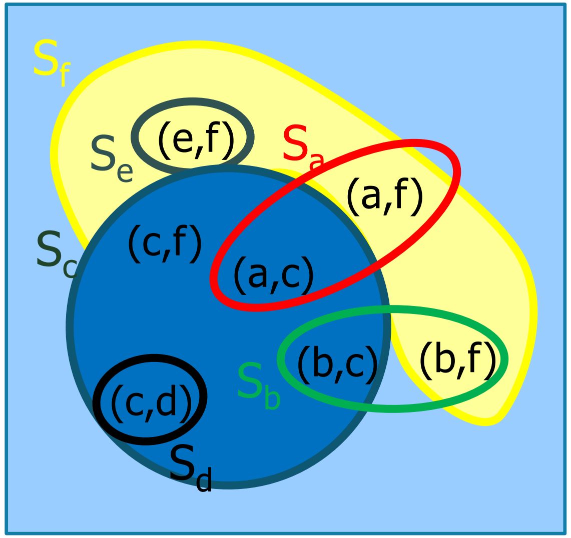
$S_d$ = {(c,d)}

$S_e$ = {(e,f)}

$S_f$ = {(a,f),(b,f),(c,f)(e,f)}

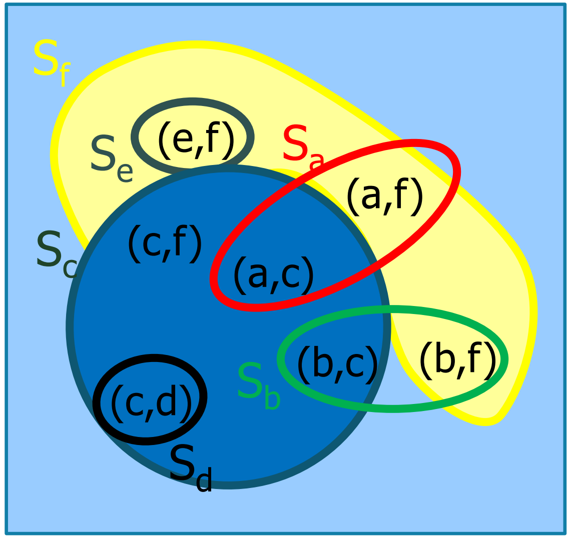
## Example 2

Solving the set-cover problem



$U = \{(a,c),(a,f),(b,c)$
$(b,f),(c,d),(c,f)(e,f)\}$
$S_a = \{(a,c),(a,f)\}$
$S_b = \{(b,c),(b,f)\}$
$S_c = \{(a,c),(b,c),(c,d),(c,f)\}$
$S_d = \{(c,d)\}$
$S_e = \{(e,f)\}$
$S_f = \{(a,f),(b,f),(c,f)(e,f)\}$

## Example 2


$S_f$
$S_e$
(e,f)
$S_a$
(a,f)
$S_c$
(c,f)
(a,c)
(b,c)
(b,f)
$S_b$
(c,d)
$S_d$