
# Algorithms and Complexity

## Introduction to complexity analysis

Paolo Camurati and Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

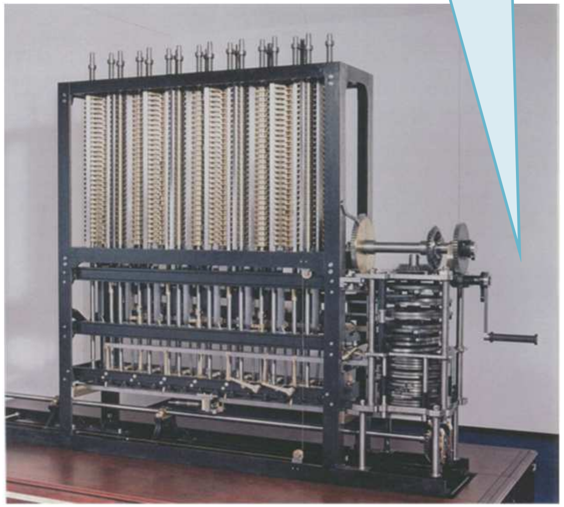
# Complexity Analysis

- Target
  - Predict performance
  - Compare algorithms
  - Provide guarantees

*" As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise. By what course of calculation can these results be arrived at by the machine in the shortest time? "*

Charles Babbage (1864)

How many times do you have to turn the crank?



# Complexity Analysis

- The challenge
  - Will my program be able to solve large practical problem?

Why is my program so slow?
Why does it run out of memory?

  - "Client gets poor performance because programmer did not understand performance characteristics"

# Complexity Analysis

- Modelling the problem
  - Given an algorithm (or a program written in a specific language)
  - Forecast of the resources the algorithm required to be executed
  - Type of resources
    - Time
    - Memory
- We should be able to prove that
  - A lower complexity may compensate hardware efficiency

# Complexity Analysis

- To really understand programs behavior we have to develop a mathematical model
- This model is usually based on the assumption the program runs on a traditional architecture
  - Sequential and single-processor model
- The model has to be
  - Independent on the hardware (CPU, memory, etc.)
  - Independent of the input data of a particular instance of the problem
    - We may eventually analyze best, average, and worst cases

# Complexity Analysis

- Our model will depend on the size **n** of the problem
- Examples
  - Number of bits of the operands for an integer multiplication
  - Number of data to sort for a sorting algorithm
  - Etc.
- Our analysis should give indications on the
  - Execution time → T(n)
  - Memory occupation → S(n)

Time and Space complexity

# Execution Time Analysis

- Donal Knuth (late ’60)
  - T(n) = «number of operations» · «operation cost»
  - Thus we must
    - Evaluates the frequency of all operations
    - Evaluates the cost of each operations

Program dependent

Hardware and software dependent

# A Simple Counting Problem

- Write a program able to
  - Read an integer value **n**
  - Print-out the number **sum** of ordered couples (i, j) such that the two following conditions hold
    - i and j are integer values
    - $1 \leq i \leq j \leq n$
- Example
  - Input: **n** = 4
  - Generated couples
    - (1,1)(1,2)(1,3)(1,4) (2,2)(2,3)(2,4) (3,3)(3,4) (4,4)
  - Output: **sum** = 10

# Algorithm 1: Brute-force

```
int count_ver1 (int n) {
  int i, j, sum;

  sum = 0;

  for (i=1; i<=n; i++) {

    for (j=i; j<=n; j++) {

      sum++;

    }
  }

  return sum;
}
```

It generates all pairs:
$1 \leq i \leq j \leq n$

It counts-them up

It returns the result

Observe that the cycle
for (i=S; i<E, i++)
performs
E-S iterations AND E-S+1 checks

# Algorithm 1: Brute-force

```
int count_ver1 (int n) {
  int i, j, sum;

  sum = 0;

  for (i=1; i<=n; i++) {

    for (j=i; j<=n; j++) {

      sum++;

    }
  }

  return sum;
}
```

1

$1 + (n+1) + n$

$\sum_{i=1}^{n} [1 + (n-i+2) + (n-i+1])$

$\sum_{i=1}^{n} (n-i+1)$

1

We can evaluate the exact number of operartions performed

We suppose ALL operations have the same constant cost (unit cost)

# Algorithm 1: Brute-force

$$T(n) = 4 + 2n + \sum_{i=1}^{n}(5 + 3n - 3i)$$

$1$

$1 + (n+1) + n$

$\sum_{i=1}^{n}[1 + (n - i + 2) + (n - i + 1])$

$\sum_{i=1}^{n}(n - i + 1)$

$1$

$$T(n) = 4 + 2n + \underbrace{\sum_{i=1}^{n}(5)}_{5n} + \underbrace{\sum_{i=1}^{n}(3n)}_{3n^2} - \sum_{i=1}^{n}(3i)$$

$$T(n) = 4 + 7n + 3n^2 - 3\sum_{i=1}^{n} i$$

Finite arithmetic progression

$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$

$$T(n) = 4 + 7n + 3n^2 - 3\frac{n(n+1)}{2}$$

$$\boldsymbol{T(n) = 1.5n^2 + 5.5n + 4}$$

Quadratic behaviour

# Algorithm 2: First Refinement

```
int count_ver1 (int n) {
  int i, j, sum;
  sum = 0;
  for (i=1; i<=n; i++) {
    for (j=i; j<=n; j++) {
      sum++;
    }
  }
  return sum;
}
```

It generates all pairs:
$1 \leq i \leq j \leq n$

```
int count_ver2 (int n) {
  int i, sum;

  sum = 0;

  for (i=1; i<=n; i++) {
    sum = sum + (n-i+1);
  }

  return sum;
}
```

# Algorithm 2: First Refinement

```
int count_ver2 (int n) {
  int i, sum;

  sum = 0;

  for (i=1; i<=n; i++) {
    sum = sum + (n-i+1);
  }

  return sum;
}
```

1

$1 + (n + 1) + n$

$$\sum_{i=1}^{n}(4)$$

1

# Algorithm 2: First Refinement

$$T(n) = 1 + 1 + 1 + 1 + n + n + 4n$$

$$\boldsymbol{T(n) = 6n + 4}$$

Linear behaviour

$1$

$1 + (n+1) + n$

$\sum_{i=1}^{n}(4) = 4n$

$1$

# Algorithm 3: Second Refinement

```
int count_ver2 (int n) {
  int i, sum;

  sum = 0;

  for (i=1; i<=n; i++) {
    sum = sum + (n-i+1);
  }

  return sum;
}
```

- The for cycle computes

$$\sum_{i=1}^{n}(n-i+1)$$

$$= n^2 + n - \sum_{i=1}^{n} i$$

$$= n\,(n+1) - \frac{n(n+1)}{2}$$

$$= \frac{n(n+1)}{2}$$

# Algorithm 3: Second Refinement

- The for cycle computes
  - $\sum_{i=1}^{n}(n-i+1)=\frac{n(n+1)}{2}$
  - Which can be used to substitute the entire cycle

```
int count_ver2 (int n) {
  int i, sum;
  sum = 0;
  for (i=1; i<=n; i++) {
    sum = sum + (n-i+1);
  }
  return sum;
}
```

It generates all pairs:
$1 \le i \le j \le n$

```
int count_ver3 (int n) {
  return n * (n+1) / 2;
}
```

# Algorithm 3: Second Refinement

- The for cycle computes
  - $\sum_{i=1}^{n}(n-i+1) = \frac{n(n+1)}{2}$
  - Which can be used to substitute the entire cycle

```
int count_ver3 (int n) {
  return n * (n+1) / 2;
}
```

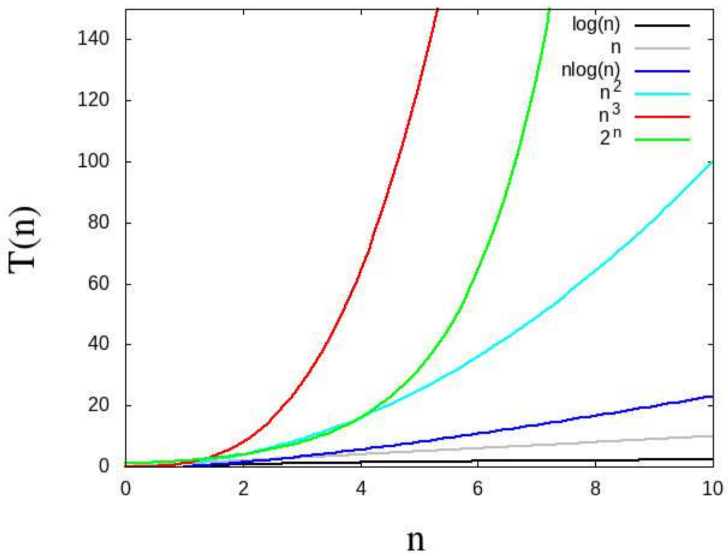
4

$\boldsymbol{T(n) = 4}$

Constant behaviour

# Summary

| Algorithm | T(n) | Order of T(n) |
|---|---|---|
| Version 1 | $1.5n^2 + 5.5n + 4$ | $n^2$ |
| Version 2 | $6n + 4$ | $n$ |
| Version 3 | $4$ | constant |

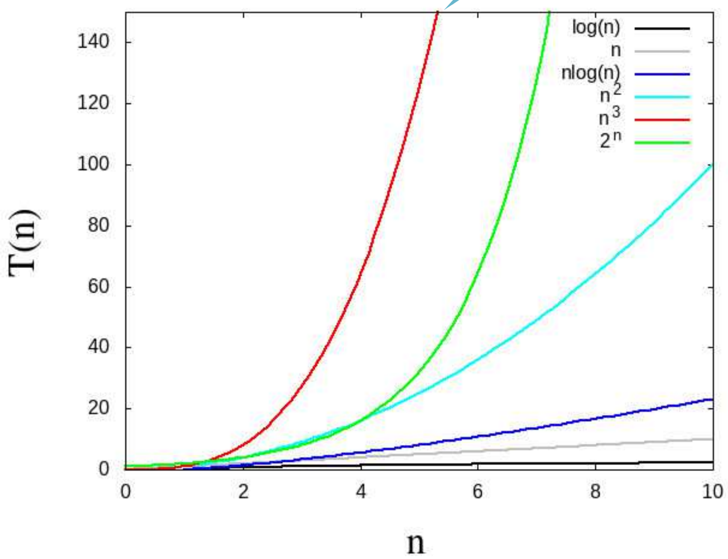


# Algorithm Classification

| Asymptotic behavior | Algorithm Class |
|---|---|
| 1 | Constant |
| log n | Logarithmic |
| n | Linear |
| n log n | Linearithmic |
| $n^2$ | Quadratic |
| $n^3$ | Cubic |
| $2^n$ | Exponential |

Complexity grows much faster than the input size

Practically?



## Summary

A lower complexity may really compensate hardware efficiency !

- Hypothesis
  - 1 operation = 1 nsec = $10^{-9}$ sec

Wall-clock (elapsed) time

| Asymptotic behavior | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^7$ |
|---|---|---|---|---|---|
| n | 1μs | 10μs | 100μs | 1ms | 10ms |
| 20 n | 20μs | 200μs | 2ms | 20ms | 200ms |
| n log n | 9.96μs | 132μs | 1.66ms | 19.9ms | 232ms |
| 20 n log n | 199μs | 2.7ms | 32ms | 398ms | 4.6sec |
| $n^2$ | 1ms | 100ms | 10s | 17min | 1.2day |
| 20 $n^2$ | 20ms | 2s | 3.3min | 5.6h | 23day |
| $n^3$ | 1s | 17min | 12day | 32years | 32 millenium |

## Some more examples

- Discrete Fourier Transform
  - Decomposition of a N-sample waveform into periodic components
  - Applications: DVD, JPEG, astrophysics, ….
  - Trivial algorithm: Quadratic ($n^2$)
  - FFT (Fast Fourier Transform): Linearitmic ($n \cdot \log n$)
- Simulation of N bodies
  - Simulates gravity interaction among n bodies
  - Trivial algorithm: Quadratic ($n^2$)
  - Barnes-Hut algorithm: Linearitmic ($n \cdot \log n$)

# Asymptotic Analysis

- Goal
  - Guess an upper-bound for T(n) for an algorithm on n data in the worst possible case
  - Asymptotic
    - For small n, complexity is irrelevant
    - Understand behaviour for $n \rightarrow \infty$

"Order or growth" classification is very important

# Asymptotic Analysis

- Three main analysis
  - Worst case
  - Average case
  - Best case
- Why worst-case analysis?
  - Conservative guess
    - Avoid complex hyphotesis on data
  - Worst case is very frequent
  - Average (and best) case
    - Either it coincides with the worst case
    - It is not definable, unless we resort to complex hypothesis on data

Running time is going somewhere in between

Design for the worst case

# Tilde Notation

- Estimate running time (or memory) as a function of input size n
  - Analyze to "within a constant factor"
- Ignore lower order terms
  - When n is large, terms are negligible
  - When n is small, terms are not negligible but we do not care about them
- Definition

$$f(n) \sim g(\mathrm{n}) \Leftrightarrow \lim_{\mathrm{n}\to\infty} \frac{f(\mathrm{n})}{g(\mathrm{n})} = 1$$

# Tilde Notation

- Examples
  - $\frac{1}{6}n^3 + 2n + 16 \sim \frac{1}{6}n^3$
  - $\frac{1}{6}n^3 + 100n^{4/3} + 16 \sim \frac{1}{6}n^3$
  - $\frac{1}{6}n^3 + \frac{5}{12}n^2 + 16 \sim \frac{1}{6}n^3$

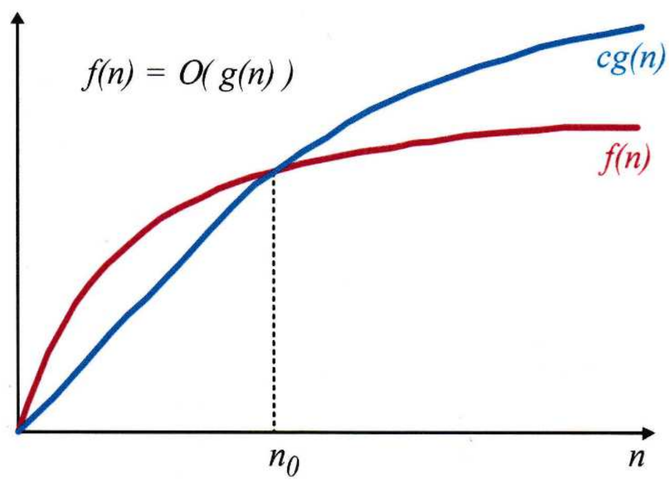
# O Asymptotic Notation

- Definition

$$f(n) = O(g(n)) \Leftrightarrow \exists\, c>0, \exists\, n_0>0 \text{ such that } \forall n \geq n_0$$
$$0 \leq f(n) \leq cg(n)$$

g(n) = loose upper bound for f(n)

Big-Oh Notation Develops upper bounds



# O Asymptotic Notation

- Examples
  - $T(n) = 3n+2 = O(n)$
    - $c=4$ and $n_0=2$
  - $T(n) = 10n^2+4n+2 = O(n^2)$
    - $c=11$ and $n_0=5$

- Theorem
  - If $T(n) = a_m n^m + \ldots. + a_1 n + a_0$
    - Then $T(n) = O(n^m)$

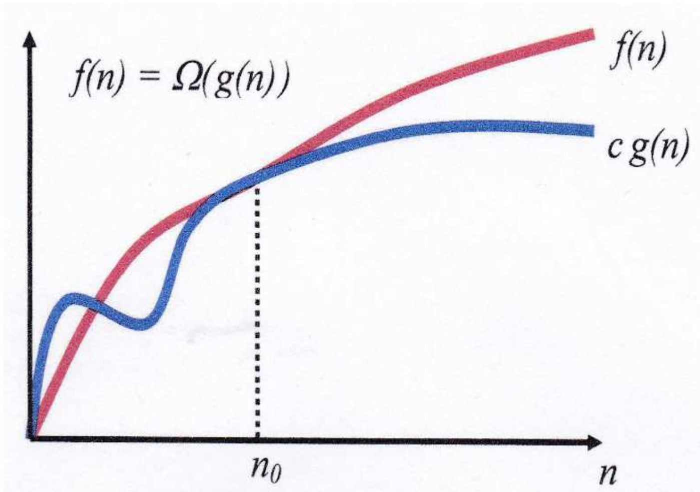
# $\Omega$ Asymptotic Notation

- Definition

$$f(n) = \Omega(g(n)) \Leftrightarrow \exists\ c>0,\ \exists\ n_0>0 \text{ such that } \forall n \geq n_0$$
$$0 \leq c\ g(n) \leq f(n)$$

g(n) = loose lower bound for f(n)

Big-Omega Notation Develops lower bounds



# $\Omega$ Asymptotic Notation

- Examples
  - $T(n) = 3n+3 = \Omega(n)$
    - $c=3$ and $n_0=1$
  - $T(n) = 10n^2+4n+2 = \Omega(n^2)$
    - $c=1$ and $n_0=1$

## Theorem

- If $T(n) = a_m n^m + \dots + a_1 n + a_0$
  - Then $T(n) = \Omega(n^m)$

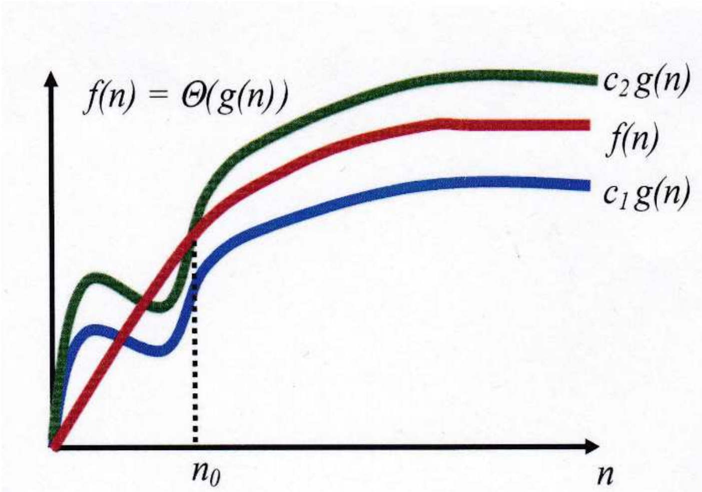
# Θ Asymptotic Notation

❖ Definition

$$f(n) = \Theta(g(n)) \Leftrightarrow \exists\ c_1, c_2 > 0,\ \exists\ n_0 > 0 \text{ such that } \forall n \geq n_0$$
$$0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$$

g(n) = tight asymptotic bound for f(n)

Big-Theta Notation
Classify algorithms
Asymptotic order of growth



# $\Theta$ Asymptotic Notation

- Examples
  - $T(n) = 3n+2 = \Theta(n)$
    - $c1=3$, $c2=4$ and $n_0=2$
  - $T(n) = 3n+2 \neq \Theta(n^2)$
  - $T(n) = 10n^2+4n+2 \neq \Theta(n)$

## Theorem

- If $T(n) = a_m n^m + \ldots. + a_1 n + a_0$
  - Then $T(n) = \Theta(n^m)$

# Theorems

- Given two functions f(n) and g(n)
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow$ f(n) = O(g(n))
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow$ f(n) = Ω(g(n))
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = const \Rightarrow$ f(n) = Θ(g(n))
  - f(n) = Θ(g(n)) ⇔ g(n) = Θ(f(n))
  - f(n) = Θ(g(n)) ⇔

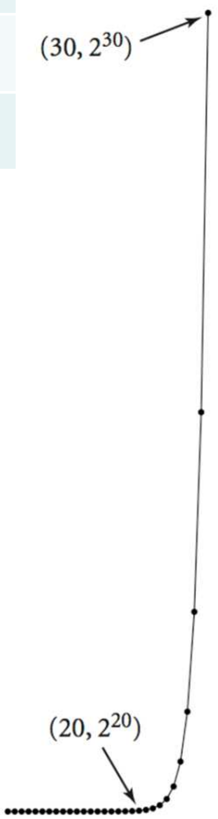
    f(n) = O(g(n)) and f(n) = Ω(g(n))
  - etc.

## Exponential growth

- Exponential growth dwarfs technological change
- Example
  - The **Travelling Salesman Problem Algorithm** on **n** points needs **n!** steps using **brute force**
  - Suppose
    - We have a giant parallel computing device
      - With as many processors as electrons in the universe
    - Where each processor has power of today's supercomputers
    - And each processor works for the life of the universe

# Exponential growth

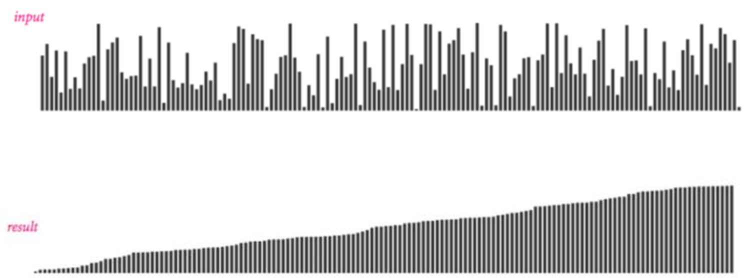
| Quantity | Value |
|---|---|
| Electrons in universe | $10^{79}$ |
| Instruction per seconds (supercomputers) | $10^{13}$ |
| Age of universe (seconds) | $10^{17}$ |



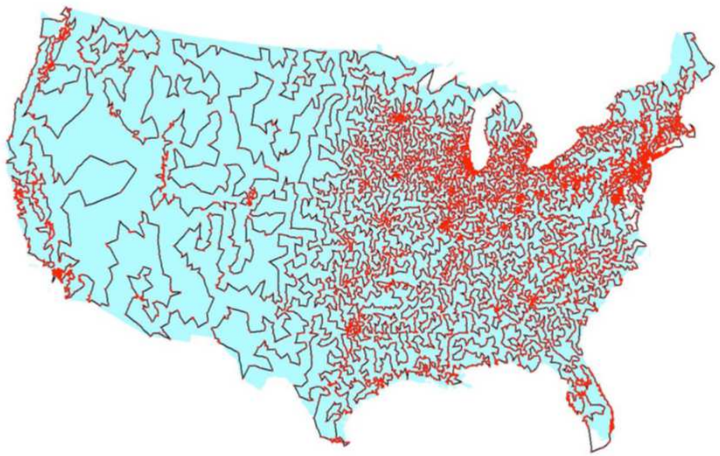

- Then
  - $1000! >> 10^{1000} >> 10^{79} \cdot 10^{13} \cdot 10^{17}$
- The parallel machine will not help to solve a 1000 point TSP problem, via brute force

# Exponential growth

- Which problems can be solved in practice?
  - Those with poly-time algorithms
- Which problems have poly-time algorithms?
  - Not so easy to know !




Many known poly-time algorithms for sorting



No known poly-time algorithms for TSP

# The P Class

- Decidable and tractable decision problems
  - There exists a polynomial algorithm that solves them (Edmonds-Cook-Karp thesis, 1970s)
  - That is, P problems are solvable in polinomial time
    - An algorithm is polynomial iff, working on n data, given a constant c>0, it terminates in a finite number of steps upper-bounded by $n^c$
    - In practice c should not exceed 2
  - Problems in P are supposed to be tractable

Most of the problems we are going to consider are in P

## The NP Class

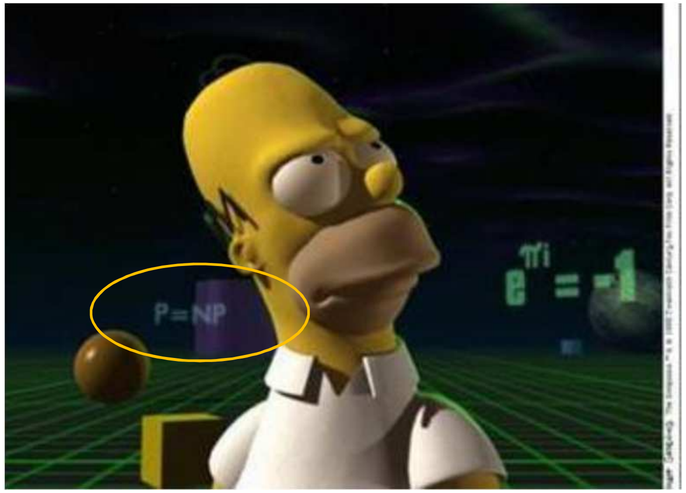
- Nondeterministic machine can guess the desired solution to a problem
- Example
  - int v[N] = {0};
  - Initializes entries to 0
  - A nondeterministic machine may inizialize entries to the final solution
- NP problems are problems solvable in poly time on a nondeterministic machine

# The NP Class

- NP stands for Non-deterministic Polynomial
- 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, factorization, graph isomorphism

## P versus NP

- Thus
  - P = class of search problems solvable in poly-tyme
  - NP = class of all search problem
- Does P = NP ?




Copyright © 1990, Matt Groening


Copyright © 2000, Twentieth Century Fox

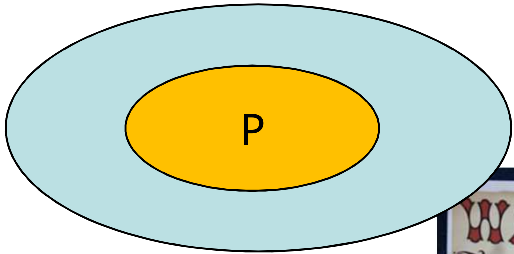
# P versus NP

- ➢ We know that $P \subseteq NP$
- ➢ We don't know whether P is a proper subset of NP or it coincides with NP

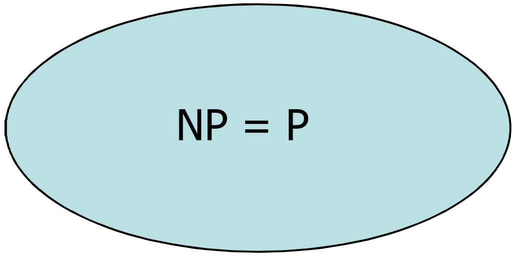
❖ It is probable that P is a proper subset of NP

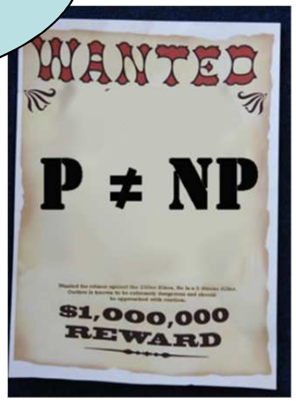
Probable

NP




Improbable



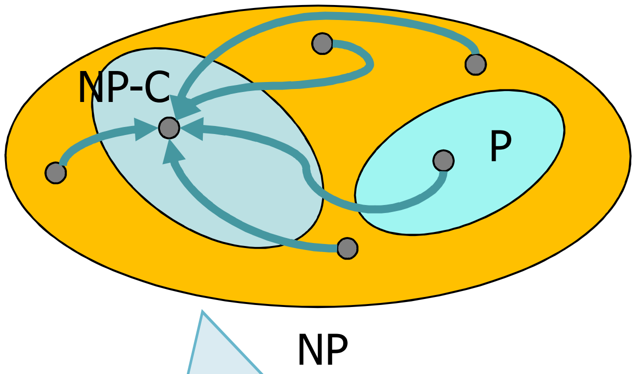

Millennium prize
$1 million for resolution

# The NP-C Class

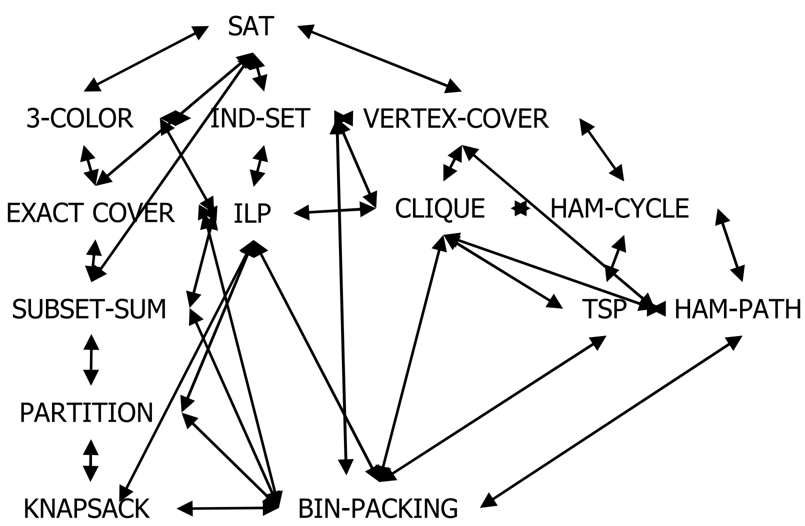
- Definition
  - An NP problem is NP-complete if every problem in NP poly-time reduce to it
- Problems in NP-C are the hardest within NP

# The NP-C Class

- 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




NP and NP-C problems are often considered as intractable



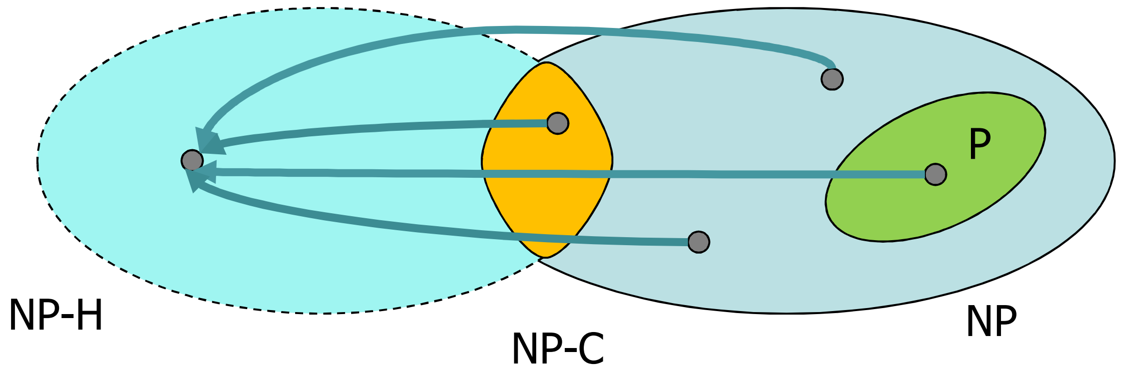
# P versus NP versus NP-C

- If we find a polynomial algorithm for any problem in this class, we could find polynomial algorithms for all NP problems, through transformations
- This is **highly improbable** !
- The existence of the NP-C class makes it probable that P $\subset$ NP

- Example of NP-C problem
  - Satisfyiability
    - Given a Boolean function, find if there exists an assignment to the input variables such that the function is true.
  - Hamilton Cycle, Clique, Graph Connectivity, Primality, Determinant

# The NP-H Class

- A problem is NP-hard if every problem in NP may be reduced to it in polynomial time (even if it does not belong to NP)
- Any other problem in NP may be reduced to it by means of a polynomial transformation
  - Permanent of a matrix



# Memory Occupation

- Memory occupation is as important as time complexity
  - In many algorithms the programmer has to trade-off time and memory contraints
    - Is it better a solution running in 100 seconds and using 10GBytes or one running in 500 seconds and using 3 Gbytes?
    - Is it better a solution using 5GBytes for all its running time or une using 10Gbytes for the first 25% of the time an 2Gbytes for the remaining 75% of the time?

# Memory Occupation

| Basics Objects | Size |
|---|---|
| Bit | 0 or 1 |
| Byte | 8 bits |
| 1KByte | $2^{10}$ Byt2 (1 thousand) |
| 1MByte | $2^{20}$ Bytes (1 million) |
| 1GByte | $2^{30}$ Byte (1 billion) |

| C scalar Type | sizeof(type) |
|---|---|
| char | 1 byte |
| int | 4 Bytes |
| float | 4 Bytes |
| double | 8 Bytes |
| etc. | etc. |

Padding may be used, i.e., each object uses a multiple of 4/8 bytes

# Memory Occupation

- Memory occupation from aggregate types may be computed starting from scalar types

```
int vet[N];

struct type {
  char id[N];
  int i;
  float x;
};
```

N · sizeof (int)

N · sizeof (char) + sizeof (int) + sizeof (float) plus padding

- Total memory S(n) usage can be computed based on those considerations