Operations Research (Master's Degree Course)

7. Complexity

Silvano Martello
DEI "Guglielmo Marconi", Università di Bologna, Italy

Complexity Theory

- **Instance** of a problem P = specific input for a numeric case of the problem;

  (Problem P = (infinite) set of all its instances.)
- **Complexity of an algorithm** = measure of the **time** it takes, **in the worst case**, to solve an instance of P;
- **(Computational) complexity of a problem** P = complexity of the best algorithm for P.
- **Time:** number of elementary steps, or number of milliseconds on a specific computer, or ...

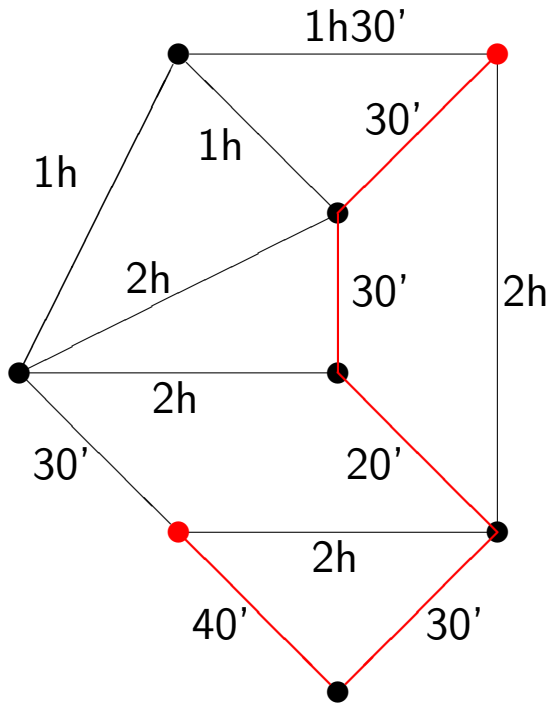
  Time as a **function of the instance size** (**any** reasonable measure).
- **Size of an instance =** number of bits needed to encode the input (**Definition**)

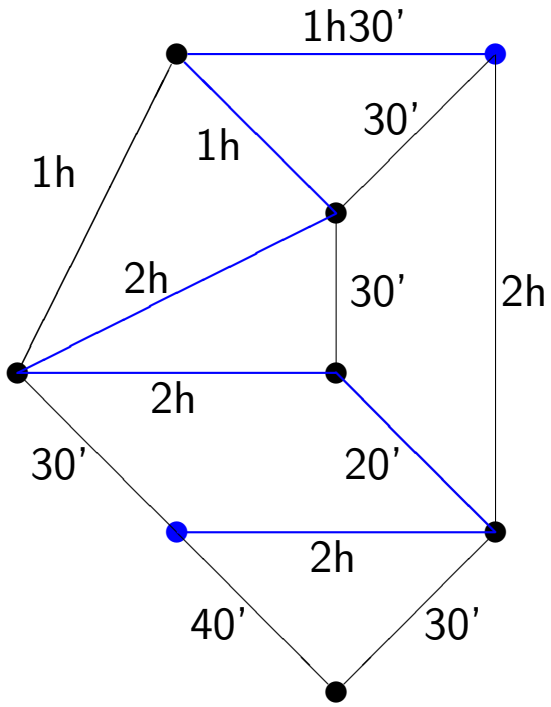
  frequently (**but not always!**) equivalent to the number of values in the input.
- **Example**: SEARCH: *given a value* b*, and a vector* $a_1, \ldots, a_n$*, decide if* $\exists\ i : a_i = b$.
  - the binary encoding of a value x needs $\sim \lceil \log x \rceil$ bits. Hence
  - size of an instance $\backsimeq \lceil \log n \rceil + \lceil \log b \rceil + \sum_{i=1}^{n} \lceil \log a_i \rceil$. Hence
  - By (reasonably) assuming that identical words with maximum number of bits are used, $\sim (n+2)\lceil \log(\max\{n, a_1, \ldots, a_n, b\}) \rceil$ bits are needed.
  - For n sufficiently large, the size of an instance is proportional to n.
  - Algorithm: n comparisons in the worst case: $\exists$ constant α such that (running time) $\leq \alpha n$.
  - Problem SEARCH has time complexity $O(n)$.

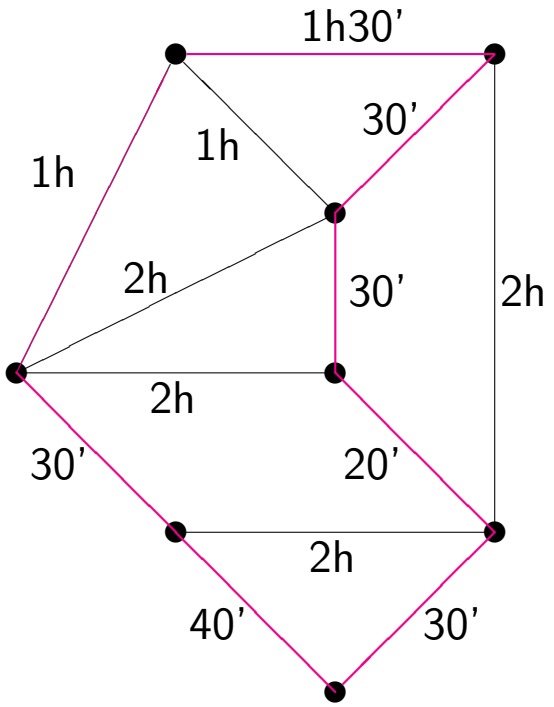
Complexity Theory (cont'd)

Remind other problems:

- **KP01**: solved in $O(2^n)$ time (worst case) by branch-and-bound;
- graph with n vertices:







  - find the shortest path that connects two vertices of a graph: solved in $O(n^2)$ time;
  - find the longest path that connects two vertices of a graph: solved in $O((n-1)!)$ time;
  - find the shortest tour through all vertices of a graph: solved in $O((n-1)!)$ time.

Complexity Theory (cont'd)

- Problems:

SEARCH:	$O(n)$	KP01:	$O(2^n)$
SHORTEST PATH IN A GRAPH:	$O(n^2)$	LONGEST PATH IN A GRAPH:	$O((n-1)!)$
		SHORTEST TOUR IN A GRAPH:	$O((n-1)!)$

Time complexity

polynomial function of the input size;	**exponential** function of the input size;
"easy" problems;	"hard" problems.

- For most optimization problems no polynomial-time algorithm is known.
- It is widely conjectured that such algorithm cannot exist.
- Complexity theory provides a rigorous treatment of these issues.

Complexity Theory (cont'd)

- Some examples of polynomial and exponential growth:

alg.	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$	$n = 60$
$O(n)$	10^{-5} "	$2 \cdot 10^{-5}$ "	$3 \cdot 10^{-5}$ "	$4 \cdot 10^{-5}$ "	$5 \cdot 10^{-5}$ "	$6 \cdot 10^{-5}$ "
$O(n^2)$	10^{-4} "	$4 \cdot 10^{-4}$ "	$9 \cdot 10^{-4}$ "	$16 \cdot 10^{-4}$ "	$25 \cdot 10^{-4}$ "	$36 \cdot 10^{-4}$ "
$O(2^n)$	10^{-3} "	1 "	17.9 '	12.7 days	35.7 years	366 centuries

- Technology will not help:

  consider the **largest instance** we can currently solve in **one CPU hour**:

  using a **1,000 times faster** CPU, in **one hour**,

  – an $O(n)$ time algorithm would solve a **1,000 times lager instance**;

  – an $O(2^n)$ time algorithm would solve an instance **10 units larger**.

Recognition version vs Optimization version

- Problem $\mathrm{KP01\_R}$: given an instance of KP01 and a threshold value ϑ, does there exist a solution of value at least equal to ϑ?
- Solution to problem $\mathrm{KP01}$: maximum of a function (**Optimization version, OV**).
- Solution to problem $\mathrm{KP01\_R}$: "yes/no" (**Recognition version, RV**);
- Problem $\mathrm{KP01\_R}$ looks easier than KP01. However,
- **Property** *From the point of view of polynomial-time solvability, the two versions have the same complexity.*

  Proof A (maximization) problem (F, d) in OV (find $f^* \in F : d(f^*) \geq d(y)\ \forall y \in F$)

  has a corresponding RV form: (F, d, ϑ) ($\exists f \in F : d(f) \geq \vartheta$?).
  - A polynomial-time algorithm for the OV immediately solves the RV (single execution).
  - A polynomial-time algorithm for the RV solves the OV through **binary search**:
    - $U :=$ upper bound: first execution with $\vartheta = \frac{1}{2}U$;
    - **if** solution $=$ "yes" **then** next execution with $\vartheta = \frac{3}{4}U$ **else** with $\vartheta = \frac{1}{4}U$, and so on.
  - The number of executions is $O(\log U)$, polynomial provided U has a binary encoding with a number of bits polynomial in the problem size (very reasonable).
  - A product of polynomials is a polynomial. $\square$.
- $\Longrightarrow$ Complexity theory, which has been developed for problems in RV, also holds for problems in OV.

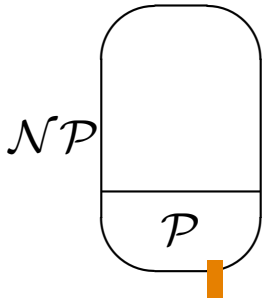
Classes $\mathcal{P}$ and $\mathcal{NP}$

- $\mathcal{P}$ = class of all problems in RV for which $\exists$ a $\mathcal{P}$olinomial-time algorithm

  (equivalently, problems that are solvable in polynomial time by a deterministic Turing machine).
- $\mathcal{NP}$ = class of all problems in RV such that if the solution is "yes" then it can be certified in polynomial time

  (equivalently, problems solvable in polynomial time by a **non**-deterministic Turing machine),

  (equivalently, problems solvable through a branch-decision tree of polynomial height).
- If a problem P is in $\mathcal{NP}$, the existence of a polynomial-time algorithm cannot be ruled out.
- $\mathcal{P} \subseteq \mathcal{NP}$.



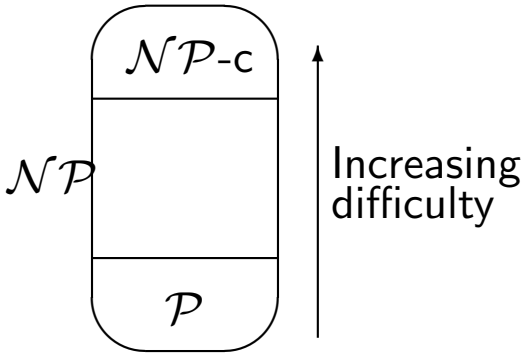
- Almost all known combinatorial optimization problems in RV are in $\mathcal{NP}$.

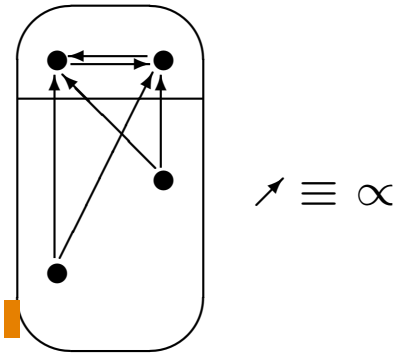
Polynomial transformation

- A problem $\mathrm{A} \in \mathcal{NP}$ is **polynomially transformable** into a problem $\mathrm{B} \in \mathcal{NP}$ ($\mathrm{A} \propto \mathrm{B}$) if $\exists$ polynomial-time algorithm which, $\forall$ instance of A, defines an instance of B that has solution "yes" if and only if the instance of A ha solution "yes". $\Rightarrow$
- If $\exists$ polynomial-time algorithm for B then $\exists$ polynomial-time algorithm for A.
- If $\mathrm{A} \propto \mathrm{B}$ and $\mathrm{B} \propto \mathrm{C}$, then $\mathrm{A} \propto \mathrm{C}$. **Proof** A sum of polynomials is a polynomial. $\square$

$\mathcal{NP}$-complete problems

- **A problem $\mathrm{A} \in \mathcal{NP}$ is called $\mathcal{NP}$-complete if $\forall\ \mathrm{B} \in \mathcal{NP}$, $\mathrm{B} \propto \mathrm{A}$.** $\Rightarrow$
- **If $\exists$ polynomial-time algorithm for A then $\exists$ polynomial-time algorithm for all problems** $\in \mathcal{NP}$, i.e.,
- A is "more difficult" (not easier) than any problem $\in \mathcal{NP}$, i.e.,
- any problem on $\mathcal{NP}$ can be seen as a special case of A.





$\nearrow \equiv \propto$

How to prove that a problem A is $\mathcal{NP}$-complete?

- To prove that a problem A is $\mathcal{NP}$-complete we need to prove that

  1) $\mathrm{A} \in \mathcal{NP}$ **and**

  2) $\forall\ \mathrm{B} \in \mathcal{NP}$, $\mathrm{B} \propto \mathrm{A}$, **or, alternatively, 2')** $\exists$ problem $\mathcal{NP}$-complete $\mathrm{C} : \mathrm{C} \propto \mathrm{A}$.

- For at least one problem we need to prove **2)**.

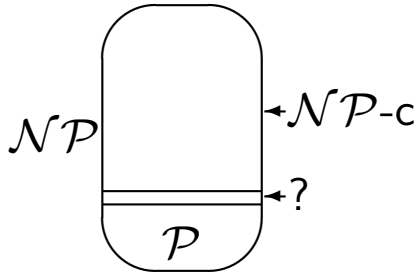
until 1970 No $\mathcal{NP}$-complete problem is known.

- **Satisfiability Problem (SAT)**: given boolean variables $x_1, \ldots, x_n$ ($x_i \in \{\text{True, False}\}$), and a boolean formula (operators: and, or, not), $\exists$ assignment of values to x : the result is "True"?
- $\mathrm{SAT} \in \mathcal{NP}$ (certificate = sequence of n True/False values).

1971 Cook proves that SAT is $\mathcal{NP}$-complete.

1972 Karp proves that $\mathrm{SAT} \propto$ KP01_R, $\mathrm{SAT} \propto$ SHORTEST TOUR IN A GRAPH (RV),
$\mathrm{SAT} \propto$ 19 more problems.

1973-1979 It is proved that almost all optimization problems for which no polynomial-time algorithm is known are $\mathcal{NP}$-complete.
Some problems are **"open"**: they are in $\mathcal{NP}$, but
no polynomial-time algorithm is known,
no $\mathcal{NP}$-completeness proof has been found.



$\mathcal{NP}$-complete problems

- *0-1* LP *(in RV) is $\mathcal{NP}$-complete.*

  Proof

  1) 0-1 LP $\in \mathcal{NP}$ (certificate = sequence of n 0/1 values).

  2') KP01 is a special case of 0-1 LP when there is only one constraint $\square$

- Similarly,
- ILP *(in RV) is $\mathcal{NP}$-complete*:

  01 LP (in RV) is a special case of ILP (in RV).

- MILP *(in RV) is $\mathcal{NP}$-complete*:

  ILP (in RV) is a special case of MILP (in RV).

The $\mathcal{P}$ vs $\mathcal{NP}$ question

- A polynomial-time algorithm for a single (**any!**) $\mathcal{NP}$-complete problem would produce a polynomial-time algorithm for **all** $\mathcal{NP}$-complete problems, i.e.,

  it would prove that $\mathcal{P} = \mathcal{NP}$.
- However, in the last 60 years **nobody** has been able to find such an algorithm, hence

  most mathematicians and computer scientists believe that its existence is unlikely.
- On the other hand, formally proving that $\mathcal{P} \neq \mathcal{NP}$ is considered by many to be impossible with the current mathematical knowledge.
- The $\mathcal{P}$ **vs** $\mathcal{NP}$ **question** is one of the seven Millennium Prize Problems, for which the *Clay Mathematics Institute* has awarded a US$ 7,000,000 prize (US$ 1,000,000 each).
- `http://www.claymath.org/millennium/` Good luck! ☺
- Few problems are still open: they are in $\mathcal{NP}$, no polynomial-time algorithm is known, there is no proof that they are $\mathcal{NP}$-complete.
- The OV of an $\mathcal{NP}$-complete RV problem is usually called $\mathcal{NP}$-**hard**
  (but the formal definition is less simple).

Complexity of Linear Programming

until 1979 LP is open:

- The simplex algorithm is polynomial in practice: it "rarely" needs more than $m \log n$ iterations but in the worst case it is not: instances have been build for which it takes $2^{m/2-1}$ iterations.
- No polynomial-time algorithm is known.
- LP (in RV) $\in \mathcal{NP}$ (certificate = sequence of n values).
- No $\mathcal{NP}$-completeness proof is known.

1979 Khachiyan invents a polynomial-time algorithm for LP.

New York Times, November 27, 1979:

An Approach to Difficult Problems

Mathematicians disagree as to the ultimate practical value of Leonid Khachiyan's new technique, but concur that in any case it is an important theoretical accomplishment.

Mr. Khachiyan's method is believed to offer an approach for the linear programming of computers to solve so-called "traveling salesman" problems. Such problems are among the most intractable in mathematics. They involve, for instance, finding the shortest route by which a salesman could visit a number of cities without his path touching the same city twice.

Each time a new city is added to the route, the problem becomes very much more complex. Very large numbers of variables must be calculated from large numbers of equations using a system of linear programming. At a certain point, the compexity becomes so great that a computer would require billions of years to find a solution.

In the past, "traveling salesmen" problems, including the efficient scheduling of airline crews or hospital nursing staffs, have been solved on computers using the "simplex method" invented by George B. Dantzig of Stanford University.

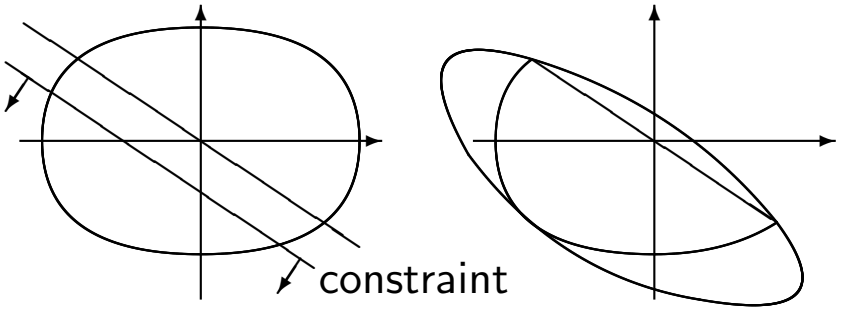
As a rule, the simplex method works well, but it offers no guarantee that after a certain number of computer steps it will always find an answer. Mr. Khachiyan's approach offers a way of telling right from the start whether or not a problem will be soluble in a given number of steps.

Two mathematicians conducting research at Stanford already have applied the Khachiyan method to develop a program for a pocket calculator, which has solved problems that would not have been possible with a pocket calculator using the simplex method.

Mathematically, the Khachiyan approach uses equations to create imaginary ellipsoids that encapsulate the answer, unlike the simplex method, in which the answer is represented by the intersections of the sides of polyhedrons. As the ellipsoids are made smaller and smaller, the answer is known with greater precision. MALCOLM W. BROWNE

Complexity of Linear Programming (cont'd)

- The Khachiyan algorithm comes from results on non-linear programming, and does not consider the combinatorial aspect (vertices).
- It generates a series of ellipsoids in R^n, of decreasing volume, encapsulating the solution:



ellipsoid method

- Its worst (but also average) case running time grows with n^6 $\Rightarrow$ it cannot be used in practice.

1984 Karmarkar ($Bell\ Labs$): algorithm with worst-case running time growing with $n^{3.5}$.

- The algorithm is based on gradient projection techniques. It moves through the interior of the polytope (**interior point method**). Karmarkar claims that the algorithm is 50 times faster than the simplex algorithm, but many researchers do not find such results.

1988 AT&T obtains a patent on the Karmarkar algorithm, and sells an 8-processor Alliant computer incorporating it at US$ 9,000,000: only two copies sold.
The algorithm is efficient for problems with very sparse constraint matrix

Today Patent expired: commercial interior point LP solvers are available.
The simplex algorithm is still the main tool for LP (great efficiency of today solvers).

Dynamic Programming (DP, R. Bellman 1956)

- **Subset-Sum Problem (SSP)**: $\max z = \sum_{j=1}^{n} w_j x_j$

$$\sum_{j=1}^{n} w_j x_j \leq c$$

$$x_j \in \{0, 1\} \; \forall \; j.$$

- **Example:** $w = (11, 18, 7, 22, 29), \; c = 35.$
- $M_j = \{$ feasible values that can be obtained by only using the first j elements $\}$:
- $M_0 = \{0\};$
  $M_1 = \{0, 11\};$
  $M_2 = \{0, 11, 18, 29\};$
  $M_3 = \{0, 11, 18, 29, 7, 25\};$

$$M_j = M_{j-1} \cup \{W + w_j \; : \; W \in M_{j-1} \; , \; W + w_j \leq c\}$$

  $M_4 = \{0, 11, 18, 29, 7, 25, 22, 33\};$
  $M_5 = M_4;$
  $z = \max\{W : W \in M_5\} = 33.$

Dynamic Programming for SSP

- **procedure Subset Sum DP**:
  begin
  $M_0 := \{0\}$;
  for $j := 1$ **to** n **do**
  begin
  $M_j := M_{j-1}$;
  for each $W \in M_{j-1}$ **do**
  if $W + w_j \leq c$ and $W + w_j \notin M_{j-1}$
  then $M_j := M_j \cup \{W + w_j\}$
  end;
  $z := \max\{W \ : \ W \in M_n\}$
  end.

- **Can we obtain a DP algorithm for KP01?** Differences:
  - SSP: M_j contains values $W = \sum_{k \in S} w_k \ \ (S \subseteq \{1, \ldots, j\})$;
    if $\sum_{k \in S'} w_k = \sum_{k \in S''} w_k$ the choice is irrelevant.
  - KP01: M_j must contain pairs $(\sum_{k \in S} p_k, \sum_{k \in S} w_k)$;
    if $\sum_{k \in S'} w_k = \sum_{k \in S''} w_k$ and $\sum_{k \in S'} p_k \geq \sum_{k \in S''} p_k$ (S' **dominates** S'')
    only the pair corresponding to S' is stored.

Dynamic Programming algorithm for KP01

- **procedure Knapsack DP**:
  begin
  $M_0 := \{(0,0)\}$;
  for $j := 1$ **to** n **do**
  begin
  $M_j := M_{j-1}$;
  for each $(P,W) \in M_{j-1}$ **do**
  if $W + w_j \leq c$ **then**
  begin
  $M_j := M_j \cup \{(P + p_j, W + w_j)\}$;
  if $M_{j-1} \ni (P',W') \ : \ W' = W + w_j$ **then**
  if $P' \leq P + p_j$ **then** $M_j := M_j \setminus (P',W')$
  else $M_j := M_j \setminus \{(P + p_j, W + w_j)\}$;
  end
  end;
  $z := \max\{P \ : \ (P,W) \in M_n\}$
  end.
- The pairs (P,W) (the values W for SSP) are called **states**.

Example: $p = (\ 7,\ 18,\ 8,\ 20,\ 25\ )$;
$w = (\ 11,\ 18,\ 7,\ 22,\ 29\ )$, $c = 35$.

$M_0 = \{(0,0)\}$

$M_1 = \{(0,0), (7,11)\}$

$M_2 = \{(0,0), (7,11), (18,18), (25,29)\}$

$M_3 = \{(0,0), (7,11), (18,18), (25,29), (8,7), (\cancel{15,18}), (26,25)\}$

$M_4 = \{(0,0), (7,11), (18,18), (\cancel{25,29}), (8,7), (26,25), (20,22), (27,33), (28,29)\}$

$M_5 = \{(0,0), (7,11), (18,18), (8,7), (26,25), (20,22), (27,33), (28,29), (\cancel{25,29})\} \rightarrow z = 28.$

Complexity of the DP algorithm for KP01

- n iterations;
- at each iteration:
  - $|M_{j-1}|$ iterations;
  - the 2nd elements W of all pairs in M_{j-1} are different from each other, and $\leq c$;
  - $\Rightarrow |M_{j-1}| \leq c+1$;
- each iteration in the inner loop takes constant time by storing M_j

  as a vector M of $c+1$ elements with: $\mathrm{M}(W) = \begin{cases} P & \text{if state } (P,W) \text{ exists} \\ \emptyset & \text{otherwise} \end{cases}$
- $\Rightarrow$ **Overall time complexity $O(nc)$ polynomial !!! ???**

Pseudo-polynomial Algorithms

- nc **is not a polynomial function** of the input length! Indeed:
- Suppose by simplicity that $p_j, w_j \leq c \; \forall j$. The binary size of an instance is bounded by

$$\sim (2n+2)\lceil \log c \rceil \quad \text{bits.}$$

- nc **is an exponential function** of the input length:

$$nc = n \cdot 2^{\log c}$$

- Algorithms of this kind are called **pseudo-polynomial algorithms**.
- *Given a combinatorial problem* A *(on integer values),* $\forall I \in \mathrm{A}$ *let*
  - $NUM(I)$ =*largest integer in* I;
    - **Examples:** instance I of SSP: $NUM(I) = \max(c, \max_j\{w_j\}) = c$;
      instance I of KP01: $NUM(I) = \max(c, \max_j\{w_j\}, \max_j\{p_j\})$;
  - $DIM(I)$ = size (dimension) of I (= number of bits to encode I).

  An algorithm for A *is* **pseudo-polynomial** *if it solves any instance* I *of* A *in a time bounded by a polynomial function of* $DIM(I)$ **and** $NUM(I)$.

Strongly $\mathcal{NP}$-complete problems

- Given a combinatorial problem A and a polynomial function $p : N \rightarrow N$, let A_p be the **restriction** of A to those instances I for which $NUM(I) \leq p(DIM(I))$.

  Examples: • $\mathrm{KP01\_R}_n = \{$ instances of $\mathrm{KP01\_R} : c, p_j, w_j \leq n\}$;

  • $\mathrm{KP01\_R}_{n^2} = \{$ instances of $\mathrm{KP01\_R} : c, p_j, w_j \leq n^2\}$;

- A problem A is called **strongly $\mathcal{NP}$-complete** if $\exists$ polynomial p : A_p is $\mathcal{NP}$-complete.

- **Theorem** No strongly $\mathcal{NP}$-complete problem can admit a pseudo-polynomial algorithm, unless $\mathcal{P} = \mathcal{NP}$.

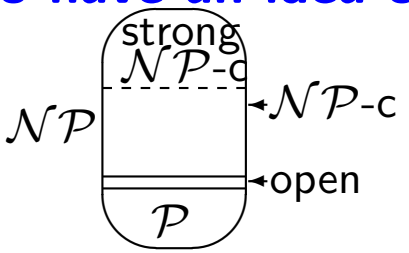
  Proof Given a strongly $\mathcal{NP}$-complete problem A, suppose there exists a pseudo-polynomial algorithm for A. The algorithm would solve any instance I of A in $q(DIM(I), NUM(I))$ time, q being a polynomial. Given a polynomial p, the algorithm would solve problem A_p (which is $\mathcal{NP}$-complete) in $q(DIM(I), p(DIM(I)))$ time, polynomial. □

- $\mathrm{KP01\_R}$ is **not** strongly $\mathcal{NP}$-complete:
  - $\mathrm{KP01\_R}_n$ is **not** $\mathcal{NP}$-complete: it can be solved in $O(n^2)$ time by DP;
  - $\mathrm{KP01\_R}_{n^2}$ **is not** $\mathcal{NP}$-complete: it can be solved in $O(n^3)$ time by DP.
- It can be proved that Shortest tour in a graph (in RV) remains $\mathcal{NP}$-complete even if all traveling times are equal to '1' or '2';

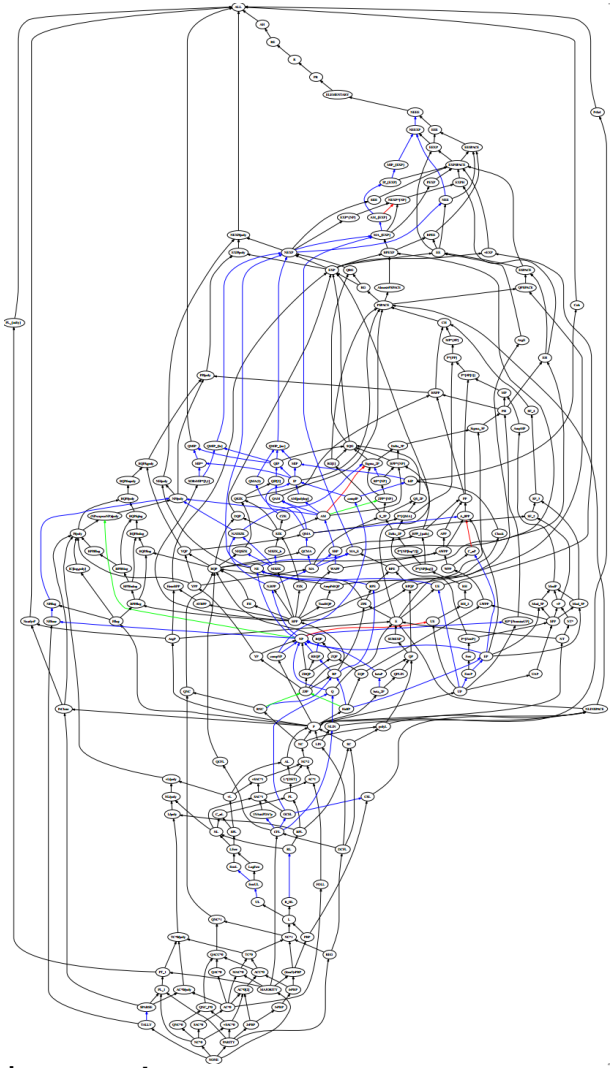
  $\Longrightarrow$ Shortest tour in a graph (in RV) is strongly $\mathcal{NP}$-complete.
- ILP, MILP, $\mathrm{LP01}$ (in RV) are strongly $\mathcal{NP}$-complete.

To have an idea of the complexity world

- Our final view:



- A more precise view:



Current (2021) complexity world: **545 classes** and counting.

Practical use of dynamic programming

- DP is a general methodology, that can be used for any combinatoral problem.
  (If used for strongly $\mathcal{NP}$-complete problems, the time will be exponential.)
- Clever enumeration of all feasible solutions (somehow similar to breadth-first branch-and-bound).
- One must also store the solution corresponding to each state.
- **Ex.** (KP01): $\forall$ state (P, W), add vector x : $P = \sum_{k=1}^{j} p_k x_k$ and $W = \sum_{k=1}^{j} w_k x_k$.
- In addition, improving techniques can be adopted. For example, for KP01:
  - item sorting: $p_j/w_j \geq p_{j+1}/w_{j+1}$;
  - dominance:
    if $\exists\ (P', W', x'), (P'', W'', x'')$: $P' \geq P''$ and $W' \leq W''$, only (P', W', x') is stored;
  - bounds: for each $(\overline{P}, \overline{W}, \overline{x}) \in M_j$, compute

    $\overline{UB} = \overline{P}+$ upper bound on $[(p_{j+1}, \ldots, p_n), (w_{j+1}, \ldots, w_n), c - \overline{W}]$;

    if $\overline{UB} \leq \max\{P : (P, W, x) \in M_j\}$, $(\overline{P}, \overline{W}, \overline{x})$ can be eliminated.
- Less frequent than branch-and-bound $\Leftarrow$ huge memory needed (KP01: proportional to nc).
- Used for very difficult problems of small size.
- Mixed algorithms branch-and-bound + dynamic programming.

Implementation of a dynamic programming algorithm for KP01

- Given two integers j, $\widehat{c}$: $1 \leq j \leq n, 0 \leq \widehat{c} \leq c$, consider the subproblem
  $f_j(\widehat{c}) = \max\{\sum_{k=1}^{j} p_k x_k \ : \ \sum_{k=1}^{j} w_k x_k \leq \widehat{c}, x_k \in \{0,1\}, k = 1, \ldots, j\}$
- $f_1(\widehat{c}) = \begin{cases} 0 & \text{for } \widehat{c} = 0, \ldots, w_1 - 1 \\ p_1 & \text{for } \widehat{c} = w_1, \ldots, c \end{cases}$
- $f_j(\widehat{c}) = \begin{cases} f_{j-1}(\widehat{c}) & \text{for } \widehat{c} = 0, \ldots, w_j - 1 \\ \max(f_{j-1}(\widehat{c}),\ f_{j-1}(\widehat{c} - w_j) + p_j) & \text{for } \widehat{c} = w_j, \ldots, c \end{cases}$
- **procedure KP01_DP**:
  begin
  for $\widehat{c} := 0$ **to** $w_1 - 1$ **do** $f_1(\widehat{c}) := 0$;
  for $\widehat{c} := w_1$ **to** c **do** $f_1(\widehat{c}) := p_1$;
  for $j := 2$ **to** n **do**
  for $\widehat{c} := 0$ **to** $w_j - 1$ **do** $f_j(\widehat{c}) := f_{j-1}(\widehat{c})$;
  for $\widehat{c} := w_j$ **to** c **do**
  if $f_{j-1}(\widehat{c} - w_j) + p_j > f_{j-1}(\widehat{c})$ **then**
  $f_j(\widehat{c}) := f_{j-1}(\widehat{c} - w_j) + p_j$
  else $f_j(\widehat{c}) := f_{j-1}(\widehat{c})$
  end.