RECAP

- Problem classes (categorization with respect to complexity of solution algorithms)
  → standardization
- Abstract decision problems
  $\Pi_A : I \to \{yes, no\}$

togliamo solution set

- Optimization problems can be paired with a computationally equivalent decision problem
  → ~~dd~~ upper/lower bound on cost to instance
- If decision problem admits a poly-time algorithm then same holds for optimization problem (AND VICEVERSA)

Morale: Concentrate on decision problems

Classes of decision problems

- Problems become elements of sets (classes): need a standard format
- Σ is already standardized:

  $\Sigma = \{yes, no\}$,

  $\pi_A : \Sigma \to \{yes, no\}$

e.g. rappresentazione di grafo per SUP

- Need to standardize Σ:
  - From set of abstract structures of arbitrary nature
  - To set of binary strings $\subseteq \{0,1\}^*$

1 a 1

DEF: An encoding is an injective function, mapping an abstract set Σ to $\{0,1\}^*$

$e : \Sigma \to \{0,1\}^*$

An encoding turns an abstract decision problem Π_A into a concrete decision problem

$$\Pi_A : I \longrightarrow \{0,1\}$$

$$\downarrow e(\cdot)$$

$$\Pi_C : \{0,1\}^* \longrightarrow \{0,1\}$$

The two problems are related as follows:

(1) $\forall x \in \{0,1\}^*$:

$$\Pi_C(x) = 1 \iff \exists i \in I : (x = e(i)) \wedge (\Pi_2(i) = \text{yes})$$

NOTE 1: $\Pi_C(x)$ is defined on all strings in $\{0,1\}^*$. From $(1, \Rightarrow)$ we have:

$$\forall x \in \{0,1\}^* : \nexists i \in I : x = e(i) \rightarrow \Pi_C(x) = 0$$

NOTE 2: The size of a concrete instance is $|x|$!

dimensione definita in maniera semplice

An algorithm A_{Π_C} for a concrete problem takes a binary string as input and outputs 0 or 1

WARNING: The choice of the encoding is a delicate issue with critical impact on the complexity analysis of the algorithm

RECALL: Worst case complexity:

$T(n) = \max\{$ cost of instructions executed by $A_{\Pi}(i)$, with size $(i) = n\}$

(standard cost model: each instruction costs 1)

Given an algorithm A_{Π_C} for a concrete problem Π_C:

$T_{A_{\Pi_C}}(n) = \max_{|x|=n}\{\#\text{instructions executed by } A_{\Pi_C}(x)\}$

abstract instance

Since $|x| = |e(i)| = n$ the independent parameter is the encoding length!

→ "Unreasonably" lengthy encodings may "mask" the inefficiency of an algorithm!

EXAMPLE Let $\mathcal{I} = \mathbb{N} - \{0\}$

BINARY ENCODING: If $n \in \mathcal{I} = \mathbb{N} - \{0\}$

$$e_b(n) = \langle a_k, a_{k-1}, \ldots, a_0 \rangle$$

with $k = \lfloor \log_2 n \rfloor$ and

$$n = \sum_{j=0}^{k} a_j 2^j$$

NOTE 1: $|e_b(n)| = \lfloor \log_2 n \rfloor + 1$

$$= \Theta(\log n)$$

UNARY ENCODING: If $m \in \mathbb{N} = \{0, 1, \ldots\}$

$e_u(m) = \langle \underbrace{1, \ldots, 1}_{m \text{ times}} \rangle$ (string of m 1's)

NOTE 2: $|e_u(m)| = m$

NOTE 3: $m = |e_u(m)| = 2^{\log_2 m} = \Theta\left(2^{|e_b(m)|}\right)$

The unary encoding is exponentially longer than the binary encoding!

RIPERCUSSIONS ON COMPLEXITY ANALYSIS

RECALL: $T(n)$ uses encoding length as the independent variable n

EXAMPLE Consider this algorithm

```
GAUSS(m)
    sum ← 0
    for i ← 1 to m do sum ← sum + i
    return sum
```

GAUSS computes $\sum_{i=0}^{m} i = m(m+1)/2$ executing m iterations. But

$$m = |e_u(m)| = \Theta\left(2^{|e_b(m)|}\right)$$

$\Rightarrow$

$$T^U_{GAUSS}\left(\underbrace{|e_u(m)|}_{n}\right) = \underbrace{|e_u(m)|}_{n}$$

n $\leftarrow$ independent parameter n

GAUSS takes linear time in the size of the input under the UNARY encoding!

BUT $T^B_{GAUSS}\left(\underbrace{|e_b(m)|}_{n'}\right) = m = \Theta\left(2^{\underbrace{|e_b(m)|}_{n'}}\right)$

$2^{n'}$

GAUSS takes exponential time in the size of the input under the BINARY encoding!

The same algorithm is associated to drastically different complexities by the two encodings

QUESTION: Is GAUSS good or bad?

BAD! For $m = 2^{32}$ GAUSS executes $m \approx 4$ billion sums!

while

```
GAUSS2(m)
    return m·(m+1)/2
```

executes 3 operations between numbers of size ≤ 64!

HORACE The linear complexity under the unary encoding is MISLEADING. The fact that GAUSS is bad is masked by the sheer length of the unary encoding

We will use only concise encodings whose length is comparable (i.e., at most polynomially larger) with a minimum-length encoding.

EXAMPLES

- Naturals must be encoded to the base $b > 2$, $b = O(1)$
- A set of naturals is encoded by concatenating the encodings of its elements

From now on we will ignore the details of the encoding and assume that a concise encoding is in use!

NOTATION. Given a set of abstract instances $\mathcal{I}$:
$\forall i \in \mathcal{I}$, $\langle i \rangle \in \{0,1\}^*$ denotes a concise encoding of i.
e.g., $\langle G = (V, E) \rangle$

Standardization is now complete!

possiamo interpretare funzione come funziona caratteristica di linguaggio -> ogni problema espresso come linguaggio formale

$$\Pi_C : \{0,1\}^* \longrightarrow \{0,1\}$$

ANY concrete decision problem maps binary strings (instance encodings) into bits (1/0 encoding yes/no).

FORMAL LANGUAGE REPRESENTATION OF CONCRETE DECISION PROBLEMS

RECAP: Given an alphabet Σ, a formal language L over Σ is a set of finite strings:

$$L \subseteq \Sigma^* \text{ (Kleene star)}$$

- L may / may not contain the empty string ε

$$\varepsilon \begin{cases} \notin \\ \in \end{cases} L$$

- Empty language: $\emptyset$ ($\{\varepsilon\} \neq \emptyset$!)
- $L_1 \cdot L_2$ concatenation operator:

$$L_1 \cdot L_2 = \{z \in \Sigma^* : \exists x \in L_1, y \in L_2 : z = \langle x, y \rangle\}$$

- Powers: $L^0 = \{\varepsilon\}$; $L^i = L \cdot L^{i-1}$

- Kleene star:

$$L^* = \bigcup_{i=0}^{\infty} L^i = \{z \in \Sigma^* : \exists k \geq 1 : \exists x_1, \ldots, x_k \in L : z = \langle x_1, x_2, \ldots, x_k \rangle\} \cup \{\varepsilon\}$$

ALL FINITE CONCATENATIONS OF STRINGS $\in L$

There is a one-to-one correspondence between concrete problems and formal languages over $\Sigma = \{0,1\}$

$\Pi_C \longleftrightarrow L_{\Pi_C} = \{x \in \{0,1\}^* : \Pi_C(x) = 1\}$

encodings of positive instances

We will use the two concepts interchangeably

Problem $\Pi_C \longleftrightarrow$ Algorithm solving Π_C

Language $L_{\Pi_C} \longleftrightarrow$ Algorithm deciding L_{Π_C} (Automaton)

DEF: A accepts x if $A(x) = 1$
A rejects x if $A(x) = \emptyset$

DEF: $L_A = \{x \in \{0,1\}^* : A(x) = 1\}$

(language decided by A)

DEF A language L is decided in polynomial time if $\exists A$:

1. $L = L_A$
2. $\exists k \in \mathbb{N} : \forall x \in \{0,1\}^* :$
$T_A(|x|) = O(|x|^k)$

(A runs in polynomial time)

DEF A complexity class is is a set of languages

(equivalently: a set of concrete (decision) problems)

DEF $P = \{L \subseteq \{0,1\}^* : L$ is decided in polynomial time$\}$

ALTERNATIVE DEF: (equivalent)

$$P = \{ \Pi_C : \{0,1\}^* \to \{0,1\} : \Pi_C \text{ is solved in polynomial time} \}$$

EXAMPLE

SEARCH:

- INSTANCE: $\langle S, n \rangle$: $S \subseteq \mathbb{N}$ finite, $n \in \mathbb{N}$
- QUESTION: $n \in S$? chiave è in insieme?

$$L_{SEARCH} : \{ x \in \{0,1\}^* : (x = \langle S, n \rangle) \wedge n \in S \}$$

```
A_SEARCH(x)
  if (x ≠ <S, n>) then return 0
  k ← |S|
  * S = {s_1, ---, s_k} *
  for i ← 1 to k do
      if (s_i = n) then return 1
  return 0
```

$$T_{A_{SEARCH}}(|x|) = O(|x|) \Rightarrow \{SEARCH, L_{SEARCH}\} \in P$$

sostituisce nondeterminismo

POLYNOMIAL-TIME VERIFICATION

SOLVING	VERIFYING
↕	↕
build a solution from scratch	check that a given candidate is a solution

Example: Recurrences

solving = build the closed formula

verifying = given a candidate closed formula check its correctness

Formal languages

solving: algorithm that decides if $x \in L$

verifying: two-input algorithm: x plus "additional info" y. Algorithm checks that y certifies that $x \in L$ (e.g., y = closed formula)

Verifying is easier than solving!

EXAMPLE

DEF Given $G=(V,E)$ undirected, a simple cycle is a sequence:

$\langle v_1, v_2, \dots, v_k \rangle$, $v_i \in V$, $v_1 = v_k$,

$v_i \neq v_j$ $1 \leq i < j < k$, $k \geq 1$

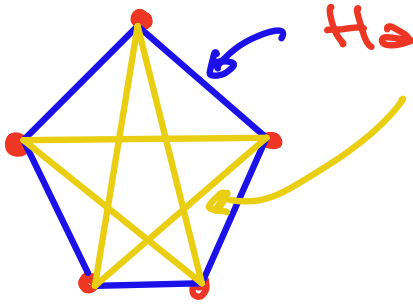
$\{v_i, v_{i+1}\} \in E$ $1 \leq i < k$

HAMILTON

I: $\langle G=(V,E) \rangle$, G undirected

D: is there in G a simple cycle containing all nodes in V?

Example:



Deciding whether a graph contains a Hamiltonian cycle is a difficult problem (no polynomial algorithms are known)

Verifying that a candidate vertex sequence is indeed a Hamiltonian cycle is easy!

VERIFY-HAMILTON (x, y)

```
if (x ≠ <G=(V,E)>) then return 0
if (y ≠ <v_1, ..., v_{|V|+1}>, v_i ∈ V) then return 0
        {check that input is "well-formed"}
if (v_1 ≠ v_{|V|+1}) then return 0
        {cubpath cannot be a cycle !}
if (∃ i,j : 1 ≤ i < j ≤ |V| : v_i = v_j)
        then return 0
  {check that y contains all nodes}
for i ← 1 to |V| do
    if ({v_i, v_{i+1}} ∉ E)
        then return 0
  {check cycle edges}
return 1
```

The algorithm is at most quadratic in the size of the input !

DEF: A two-input algorithm V verifies string $x \in \{0,1\}^*$ iff $\exists\, y \in \{0,1\}^* : V(x, y) = 1$

Any $y : V(x,y) = 1$ is called a certificate for x

For instance, if $x = \langle G = (V, E) \rangle \in L_{HAMILTON}$ a certificate for x is

$y = \langle v_1, \ldots, v_{|V|+1} \rangle$, $v_1 = v_{|V|+1}$, $v_i \neq v_j$ $1 \leq i < j \leq |V|$
$(v_i, v_{i+1}) \in E$

IMPORTANT! if $x \notin L$ then no certificate may exist!

DEF The language verified by a two-input algorithm V is:

$$L_V = \{x \in \{0,1\}^* : V \text{ verifies } x\} =$$

$$= \{x \in \{0,1\}^* : \exists y \in \{0,1\}^* : V(x,y) = 1\}$$

OBSERVATION: VERIFY-HAMILTON is a verification algorithm for $L_{HAMILTON}$ (or, equivalently, for HAMILTON)

IMPORTANT: We only require that one certificate exists but do not ask the algorithm to compute it itself.

algoritmo non costruisce certificato

The certificate is provided to the algorithm externally.

"Computing the certificate" would imply solving the problem.

DEF: A language L is verified in polynomial time iff $\exists V(x,y)$ such that:

1. $L = L_V$

2. $\exists k_1 \geq 0 : \forall x \in L \; \exists y : V(x,y)=1$ and $|y| = O(|x|^{k_1})$

3. $\exists k_2 \geq 0 :$ $T_V(|x|,|y|) = O((|x|+|y|)^{k_2})$

We need SHORT CERTIFICATES!

DEF : P: linguaggi risolti

$$NP = \{ L \subseteq \{0,1\}^* : L \text{ verified in polynomial time} \}$$

NP can be equivalently defined with respect to decision problems.

We proved that HAMILTON $\in$ NP.

Since verifying is easier than solving, we suspect that

if $L \in P \Rightarrow L \in NP$

that is, $P \subseteq NP$