Algoritmi e Strutture dati

Introduzione al corso

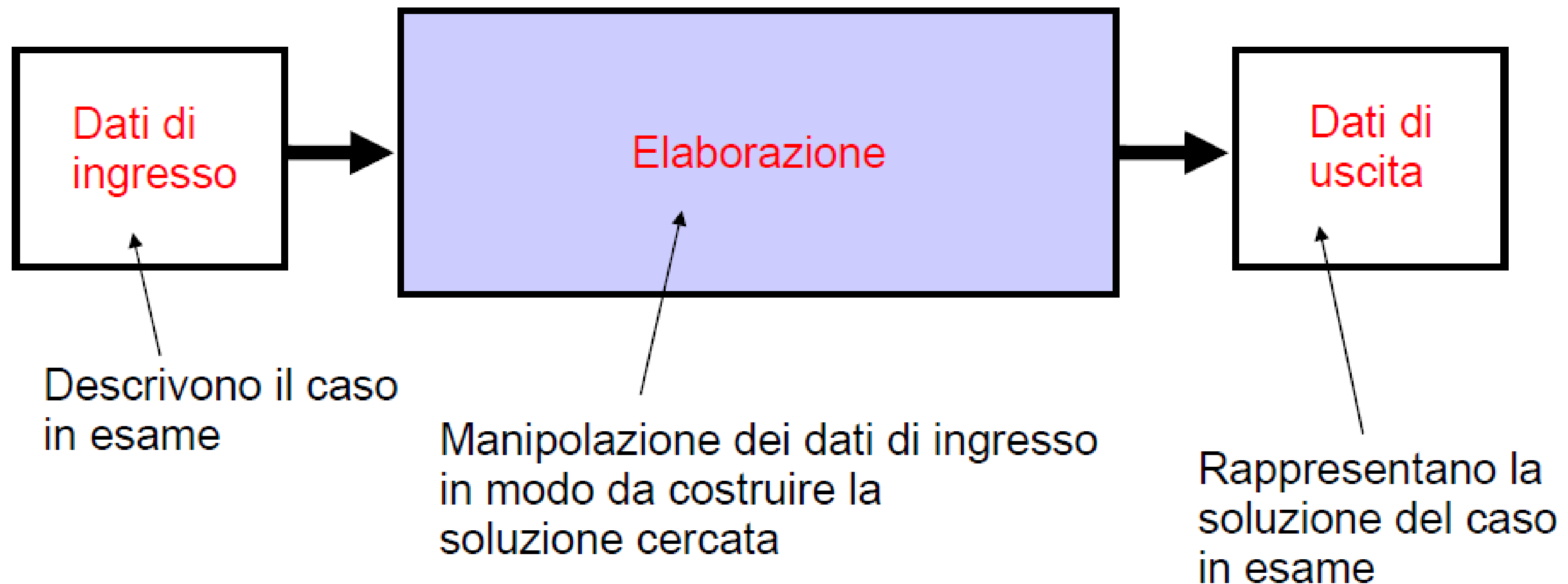
Dr Maria Federico

Cosa studieremo

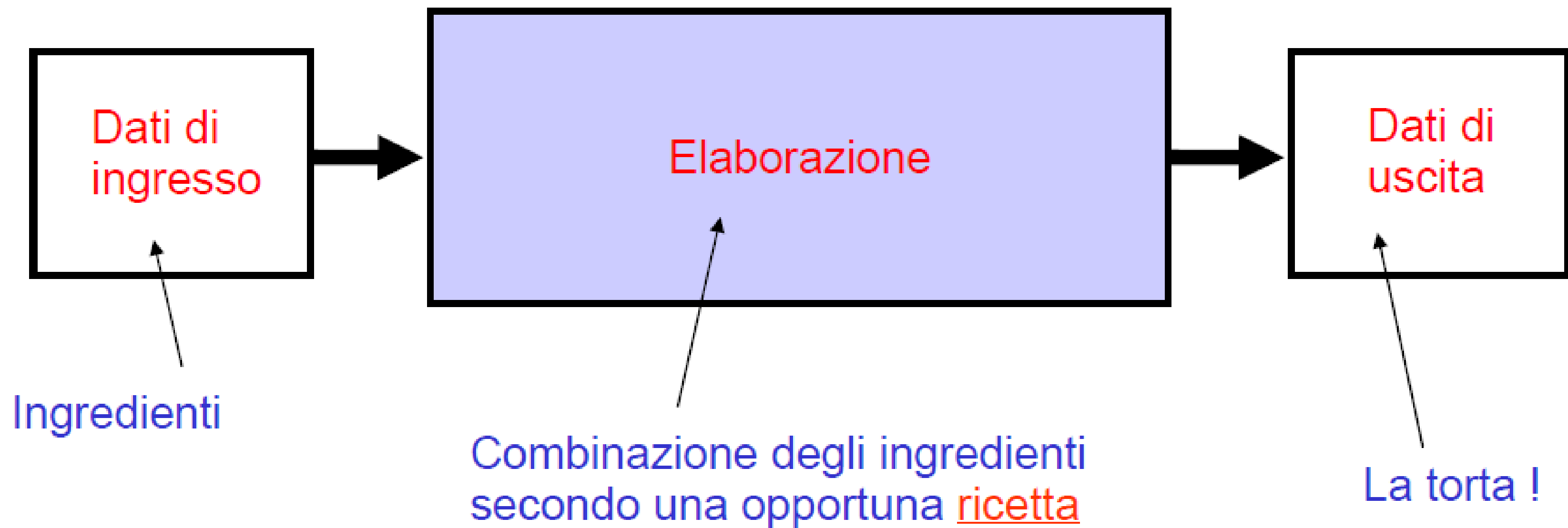
- **Algoritmi** = descrizione precisa di una sequenza di azioni che devono essere eseguite per giungere alla risoluzione di un problema
  - **Sintesi / disegno / progetto**
  - **Analisi dell'efficienza**

- **Strutture dati** = è fondamentale che i dati siano ben organizzati e strutturati in modo che l'algoritmo li possa elaborare efficientemente

Risolvere un problema



Problema: torta di carote



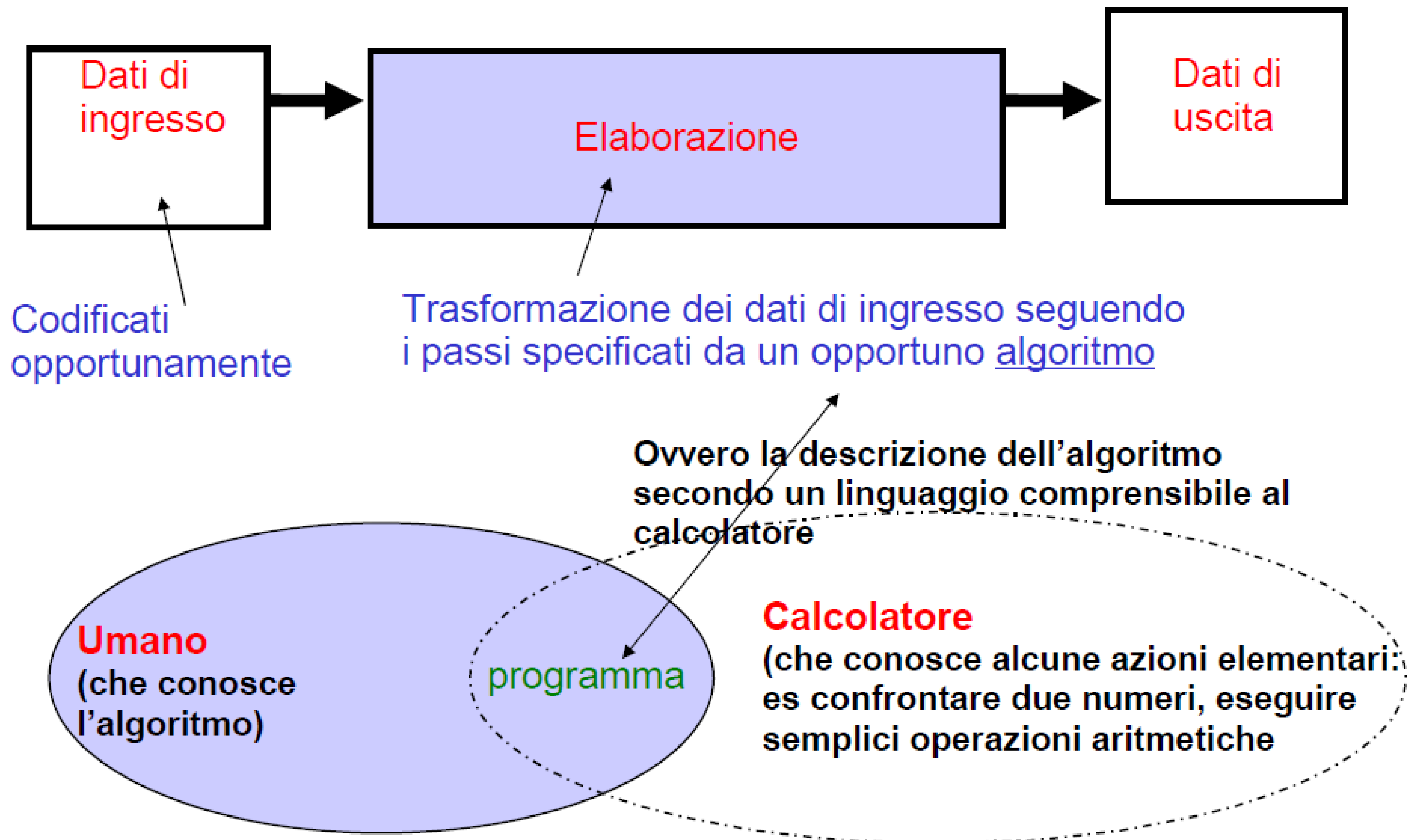
Ricetta torta di carote

Risolvere un problema con il computer

- Vogliamo essere capaci di specificare la strategia seguita dal passo di elaborazione in modo da farla eseguire automaticamente dal computer

- Dobbiamo riuscire a descrivere accuratamente i passi della soluzione attraverso azioni che il calcolatore è in grado di effettuare e con un linguaggio che è in grado di comprendere

Problemi, algoritmi, programmi



Problemi, algoritmi, programmi

- **Problema computazionale** = specifica in termini generali la relazione che deve valere tra input e output
- **Algoritmo** = descrive una procedura computazionale (sequenza di passi) ben definita per trasformare l'input nell'output
- **Programma** = rappresentazione di un algoritmo utilizzando un linguaggio non ambiguo e direttamente comprensibile dal computer

Esempio: problema dell'ordinamento

- **Input:** una sequenza di n numeri

$$<a_1, a_2, \ldots, a_n>$$

- **Output:** una permutazione (riarrangiamento) $< a'_1, a'_2, \ldots, a'_n >$ tale che

$$a'_1 \leq a'_2 \leq \ldots \leq a'_n$$

- **Istanza del problema:**

$$<31, 41, 59, 26, 41>$$

- **Soluzione:** $<26, 31, 41, 41, 59>$

Algoritmo: definizione

- **Algoritmo** = procedimento di calcolo
- Deriva dal latino ***algorithmus*** ← Mohammed **al Khowarismi** matematico usbeco del IX sec d.C. famoso per aver scritto un noto trattato di algebra

- Ci sono tante definizioni del termine algoritmo

Algoritmo: definizione

Un algoritmo è un sequenza **ordinata** di passi elementari **eseguibili** e **non ambigui** che giunge certamente a **terminazione**

- **Sequenza ordinata**
  - **SI** un algoritmo deve avere una struttura ben stabilita in termini di ordine di esecuzione dei suoi passi
  - **NO** i passi devono essere eseguiti secondo una sequenza lineare che consiste nel 1° passo seguito dal 2° e così via. Es. algoritmi paralleli = più sequenze di passi affidate a processori diversi

Algoritmo: definizione

Un algoritmo è un sequenza **ordinata** di passi elementari **eseguibili** e **non ambigui** che giunge certamente a **terminazione**

- **Passi eseguibili**
  - L'istruzione "Elencare tutti gli interi positivi" è impossibile da eseguire perché gli interi positivi sono infiniti
  - Qualsiasi insieme di istruzioni che la contiene NON è un algoritmo!!

Algoritmo: definizione

Un algoritmo è un sequenza **ordinata** di passi elementari **eseguibili** e **non ambigui** che giunge certamente a **terminazione**

- **Passi non ambigui**
  – A partire da dati iniziali le istruzioni sono applicabili in modo <u>deterministico</u>

  – L'esecuzione di ogni passo non richiede abilità creative (eccezione algoritmi non deterministici)

Algoritmo: definizione

Un algoritmo è un sequenza **ordinata** di passi elementari **eseguibili** e **non ambigui** che giunge certamente a **terminazione**

- **Terminazione**
  - L'esecuzione dell'algoritmo deve portare ad una conclusione
  - Ipotesi che deriva dall'informatica teorica che ha portato alla definizione di funzioni calcolabili e non (tesi di Church-Turing)

Algoritmo: definizione

Un algoritmo è un sequenza **ordinata** di passi elementari **eseguibili** e **non ambigui** che giunge certamente a **terminazione**

- **Terminazione**
  - Dalla tesi di Church-Turing discende che non tutti i problemi sono risolvibili mediante un algoritmo
  - Un programma potrebbe ciclare all'infinito; tecnicamente tale programma NON rappresenta un algoritmo

Algoritmi e ricette

- Ma quindi una ricetta è proprio un algoritmo?

  … **NO**, ovvero è molto simile ma con due importanti differenze:

  – La sequenza di azioni contiene spesso degli elementi di *ambiguità* risolti da un esecutore intelligente
    - Es. sale q.b.; sbatti le uova

  – Non tutti i possibili casi sono specificati
    - Es. se c'è puzza di bruciato si spegne il forno anche se l'algoritmo non lo specifica

Algoritmo: altre definizioni

Un algoritmo è un elenco **finito** di istruzioni t.c.

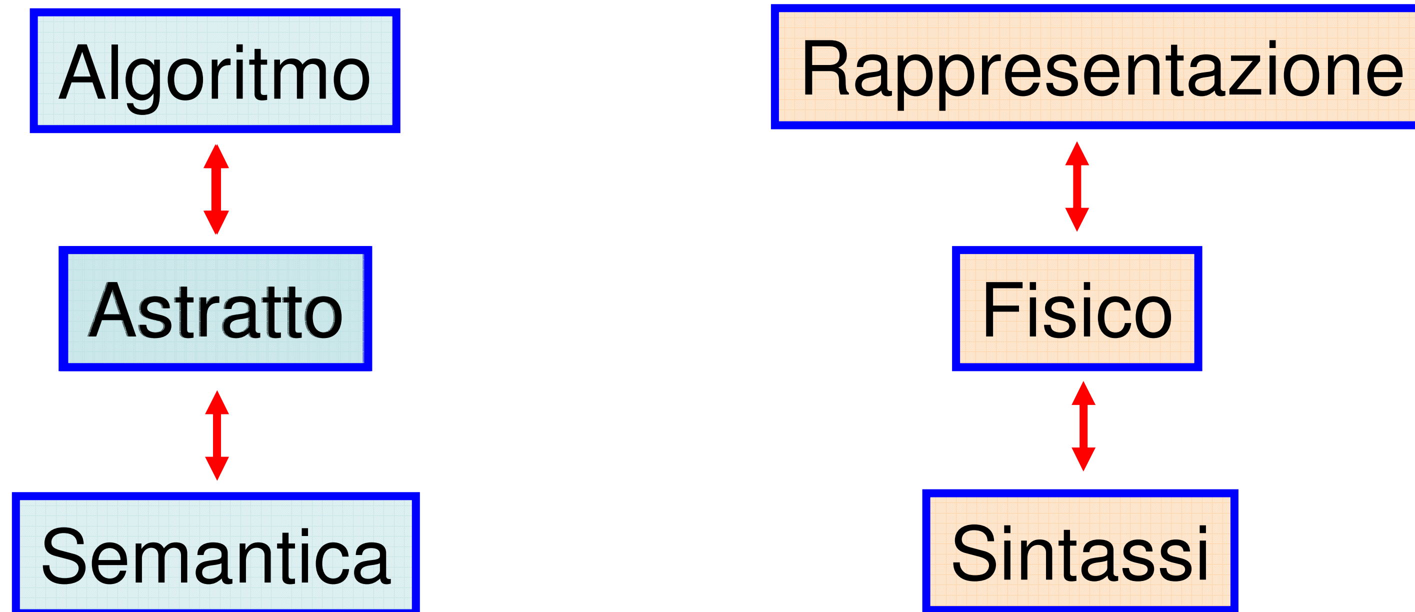
1. a partire da dati iniziali, le istruzioni sono applicabili in modo **deterministico**
2. c'è un criterio univoco per stabilire quando l'algoritmo **termina**
3. uno stato finale deve sempre essere raggiungibile in un **numero finito di passi**

Algoritmo: altre definizioni

La descrizione di un procedimento risolutivo di un problema può considerarsi un algoritmo se rispetta alcuni requisiti essenziali:

- **Finitezza**: un algoritmo deve essere composto da una sequenza finita di passi elementari
- **Eseguibilità**: il potenziale esecutore deve essere in grado di eseguire ogni singola azione in tempo finito con le risorse a disposizione
- **Non-ambiguità**: l'esecutore deve poter interpretare in modo univoco ogni singola azione

Rappresentazione degli algoritmi



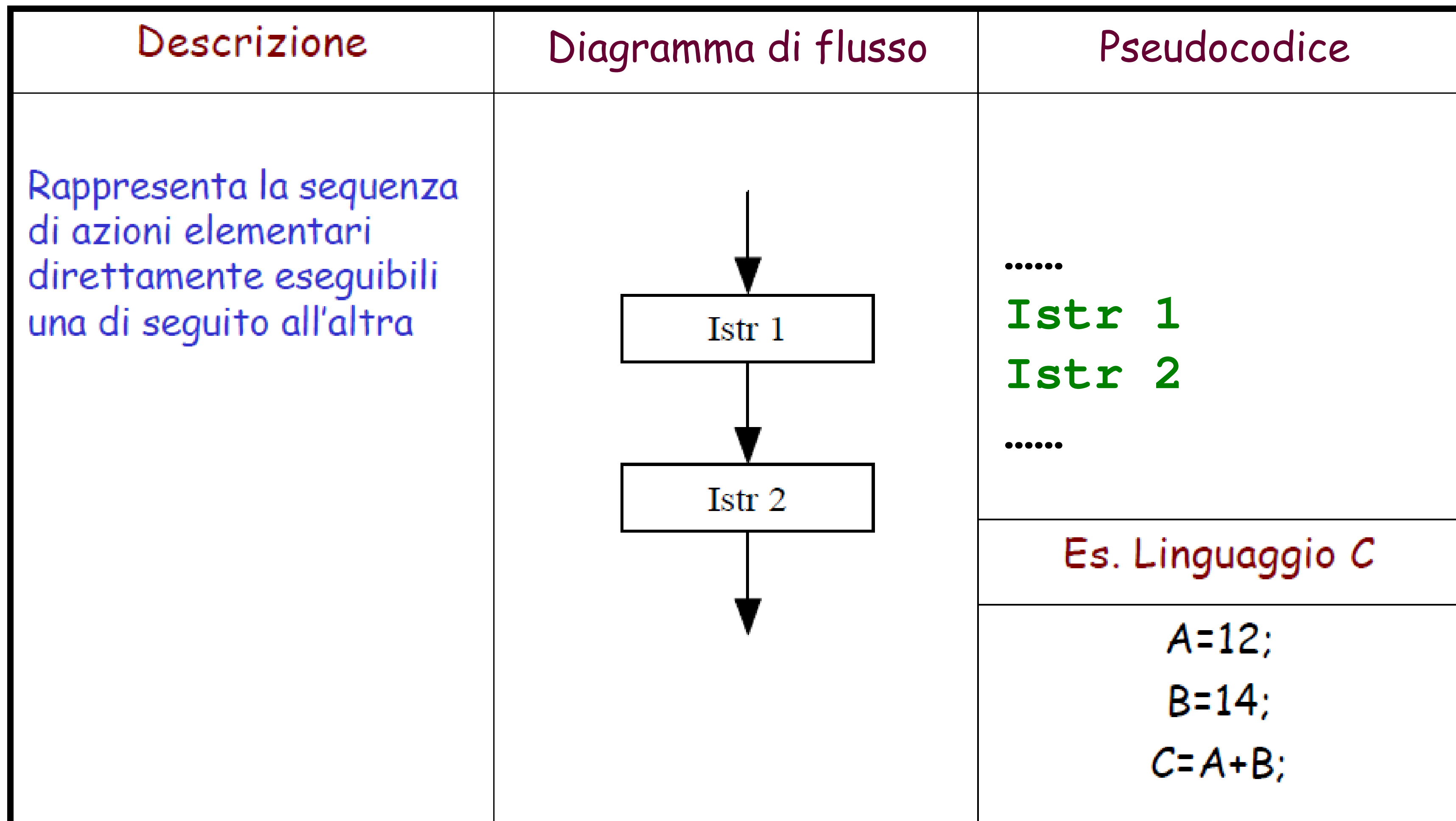
Rappresentazione degli algoritmi

- Lo stesso algoritmo può essere rappresentato in vari modi
  - formula, sequenza di istruzioni, disegno, a parole...
  - a diversi **livelli di astrazione** (linguaggio macchina, assembly, linguaggio ad alto livello: Pascal, C, Java, …)
  - noi utilizzeremo un linguaggio astratto ad alto livello o **pseudocodice**:
    - per evitare dettagli inutili
    - per sottolineare il fatto che **un algoritmo è completamente indipendente dal linguaggio**
- Ogni rappresentazione si basa su un insieme di **primitive** ben definite, comprensibili all'esecutore

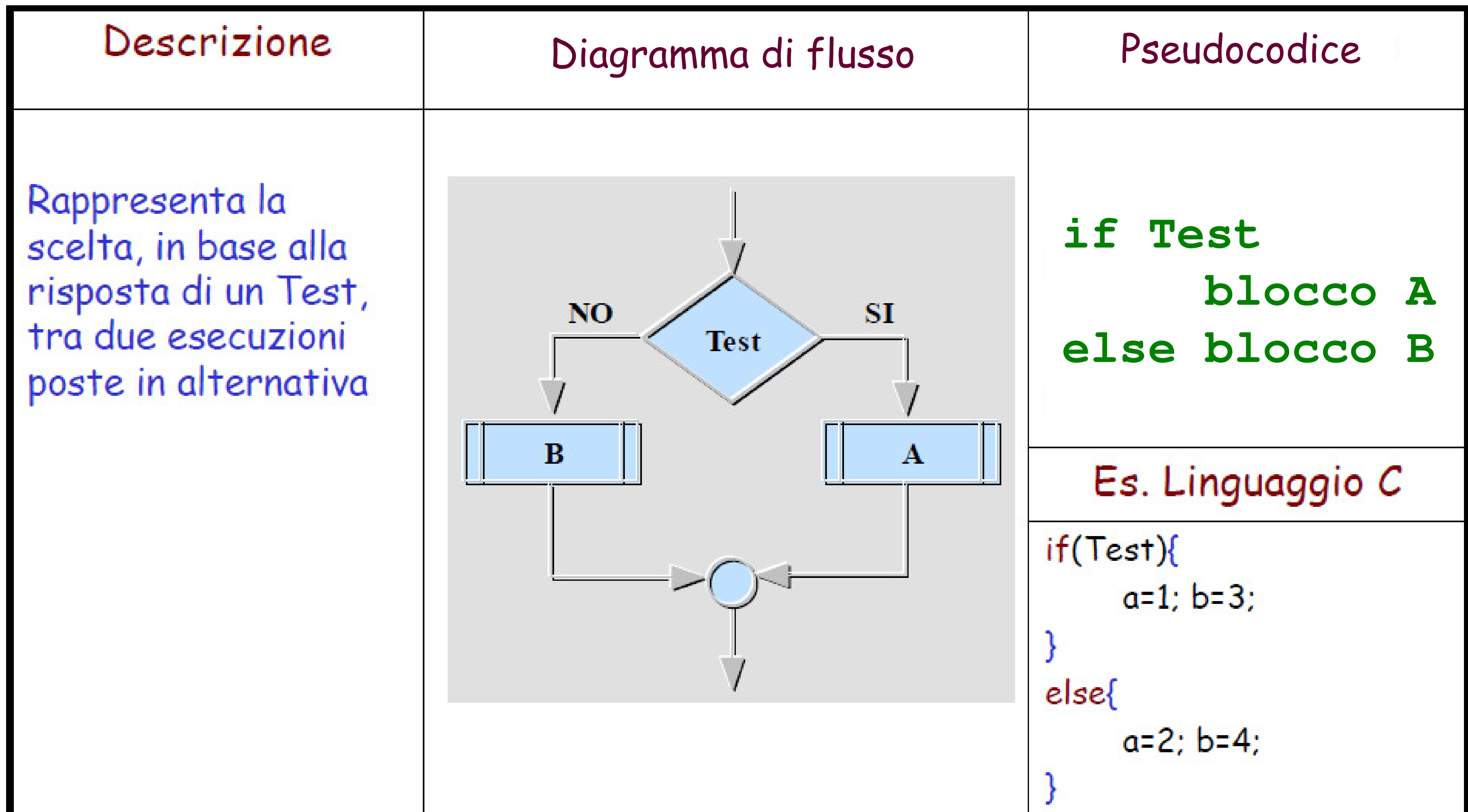
Pseudocodice programmazione strutturata

- Il teorema di **Jacopini-Böhm** afferma che qualunque algoritmo può essere descritto utilizzando esclusivamente tre strutture di controllo fondamentali:
  - struttura **sequenziale**
  - struttura **condizionale** (o di selezione)
  - struttura **iterativa**

Struttura sequenziale



Struttura condizionale



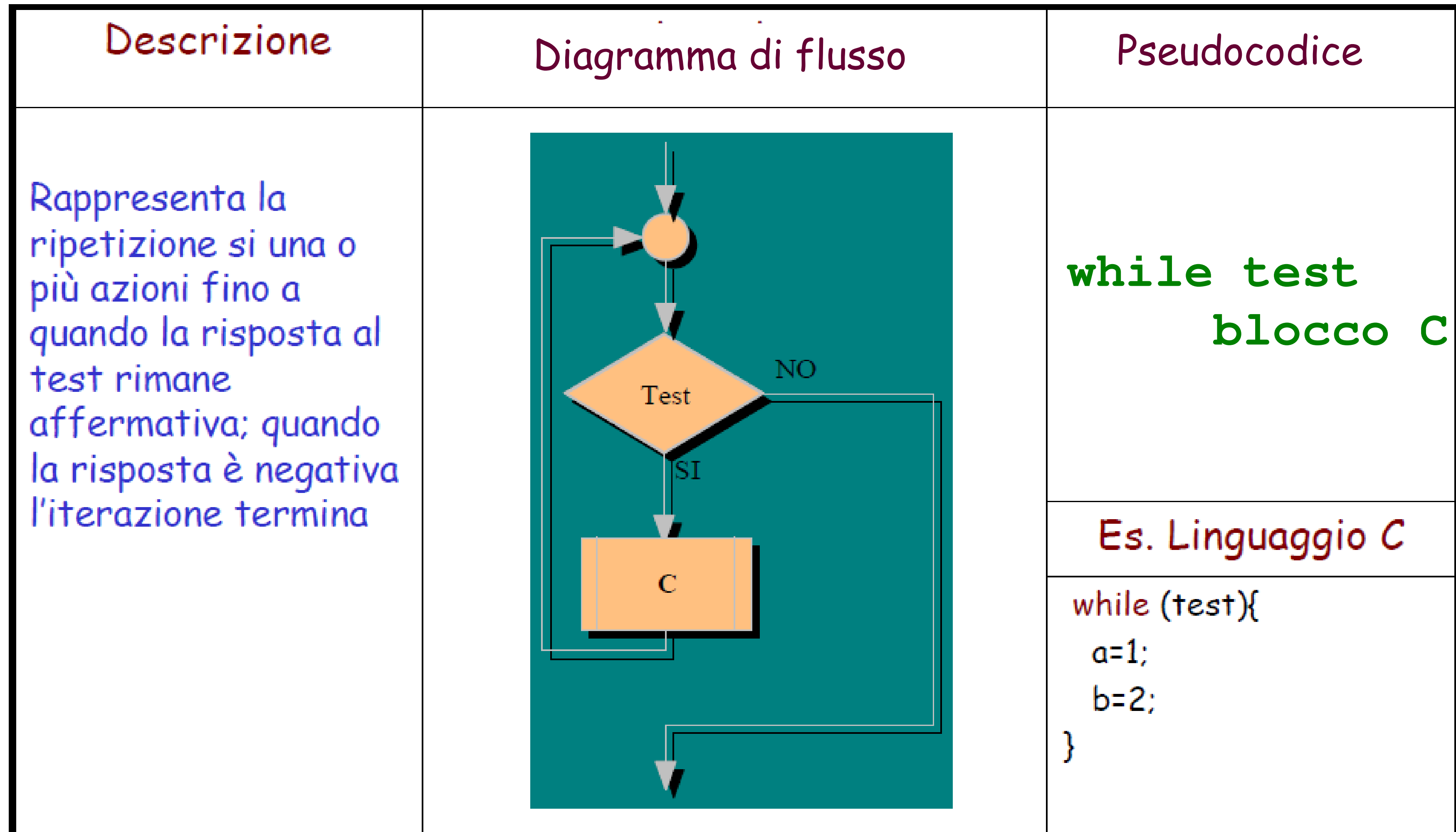
Descrizione	Diagramma di flusso	Pseudocodice
Rappresenta la scelta, in base alla risposta di un Test, tra due esecuzioni poste in alternativa	NO / Test / SI / B / A	if Test blocco A else blocco B
		Es. Linguaggio C

		if(Test){
		a=1; b=3;
		}
		else{
		a=2; b=4;
		}

Struttura condizionale

```
if test1
     blocco A
elseif test2
          blocco B
    elseif …
```

Struttura iterativa



Struttura iterativa

```
for i = 0 to A.length
  blocco istr

for i = A.length downto 0
  blocco istr
```

Pseudocodice
esempio di algoritmo

- **Problema:** calcolare il minimo di un insieme di numeri interi, maggiori o uguali a zero
- **Input:** un vettore di numeri interi

$$< a_0; a_1; a_2; \ldots ; a_{n-1} >$$

- **Output:** un numero intero *m* tale che vale la seguente relazione:

$$\forall\, 0 \leq i \leq n-1, \;\; m \leq a_i$$

- **Istanza:** < 23, 5, 7, 8, 10, 2, 3 >
- **Soluzione:** m=2

Pseudocodice
esempio di algoritmo

```
Min(A)
  min = A[0]
  for i=1 to A.length
     if A[i] < min
          min = A[i]

  return min
```

Programma, processo, algoritmo

- **Programma** = rappresentazione fisica formale di un algoritmo progettata per essere eseguita da un computer

- **Processo** = l'attività di esecuzione dell'algoritmo rappresentato dal programma

Cosa studieremo: sintesi di algoritmi

- **Dato un problema costruire un algoritmo per risolverlo**

- Durante il corso studieremo alcuni metodi di sintesi:
  - *Ricorsione*
  - *Tecnica divide et impera*
  - *Programmazione dinamica*

Cosa studieremo: analisi di algoritmi

- **Dato un algoritmo *A* e un problema *P* dimostrare che *A* risolve *P* (correttezza) e valutare la quantità di risorse usate da *A* (complessità computazionale)**

- Un algoritmo è **corretto** se, *per ogni istanza di input*, termina con l'output corretto

- Per gli algoritmi studiati durante il corso saranno presentate tecniche matematiche per permettere l'analisi della complessità, es. metodi di risoluzione di *relazioni di ricorrenza* per algoritmi ricorsivi

Cosa studieremo: analisi di algoritmi

- **Lo studio teorico dell'efficienza (performance) di un programma e dell'uso delle risorse**

- Cos'è più importante della performance?

modularità correttezza manutenibilità

funzionalità robustezza user-friendliness

tempo di programmazione semplicità

estendibilità affidabilità

Perché noi studiamo algoritmi e performance?

- Spesso l'efficienza segna il confine tra possibile e impossibile (es. applicazioni real-time)
- L'efficienza degli algoritmi mette le basi per tutte le altre cose importanti che abbiamo citato (es. aspettare tanto una risposta dal sistema non è per nulla user-friendly)

Cosa studieremo: analisi di algoritmi

- Come rendere veloci i programmi
- Analizzeremo
  - **Tempo di calcolo** impiegato da un algoritmo per risolvere un problema
  - **Spazio** occupato durante la computazione (memoria RAM o disco)

in modo da poter confrontare algoritmi diversi e progettare algoritmi efficienti

Analisi di algoritmi: Modello di calcolo

- Modello delle risorse e dei costi dell'uso delle risorse
- Modello **RAM** = Random-Access Machine
  - 1 processore
  - Istruzioni sequenziali
  - Istruzioni aritmetiche (add, sub, mul, div, mod), per spostare dati (load, store), di controllo (salto [in]condizionato, chiamata a subroutine, return); costo costante

Complessità di un algoritmo

- **T(n)** = **tempo di esecuzione** = numero di operazioni elementari eseguite
- **S(n)** = **spazio di memoria** = numero di celle di memoria utilizzate durante l'esecuzione
- **n** = **dimensione (taglia) dei dati di ingresso**
  - Es. intero positivo x: $n = 1 + \lfloor \log_2 x \rfloor$, cioè il numero di cifre necessarie per rappresentare x in notazione binaria
  - Es. vettore di elementi: n = numero delle componenti
  - Es. grafo: n = numero di nodi

T(n) tempo di elaborazione

- **Caso peggiore:** (spesso)

  $T(n)$ = tempo **massimo** dell'algoritmo su *qualsiasi* input di dimensione n

- **Caso medio:** (talvolta)

  $T(n)$ = tempo **atteso** su tutti gli input di dimensione n

  È necessaria un'assunzione sulla distribuzione statistica degli input

- **Caso migliore:** (fittizio)

  Ingannevole per algoritmi lenti che sono veloci su *qualche* input

Caso peggiore

- Generalmente si cerca un limite superiore perché
  - Fornisce una garanzia all'utente
  - Per alcuni algoritmi si verifica molto spesso
  - Il caso medio spesso è cattivo quasi quanto quello peggiore

Esempio: T(n) di una funzione iterativa

	Costo	Numero di volte
`Min(A)`		
`  min = A[0]`	c_1	1
`  for i=1 to A.length`	c_2	n
`    if A[i] < min`	c_3	n-1
`      min = A[i]`	c_4	n-1
`  return min`		

$T(n) = c_1 + n*c_2 + (n-1)*c_3 + (n-1)*c_4 = (c_2+c_3+c_4)*n + (c_1-c_3-c_4) =$

$= a*n + b$ **funzione lineare**

Tempo di calcolo indipendente dalla macchina

- *Qual è il tempo di calcolo di un algoritmo nel caso peggiore?*

  Dipende dal computer usato
  - **velocità relativa** (confronto sulla stessa macchina)
  - **velocità assoluta** (su macchine diverse)
- **GRANDE IDEA:**
  - Ignorare le costanti dipendenti dalla macchina
  - Studiare la *crescita* di T(n) con $n \to \infty$

"Analisi asintotica"

Regole utili per valutare complessità asintotica

1. Se $T(n) = c$, allora $T(n) = O(1)$, $T(n) = \Omega(1)$, $T(n) = \Theta(1)$
2. Se $T(n) = c \cdot f(n)$, allora $T(n) = O(f(n))$, $T(n) = \Omega(f(n))$, $T(n) = \Theta(f(n))$
3. Se $g(n) = O(f(n))$ e $f(n) = O(h(n))$, allora $g(n) = O(h(n))$ [anche per Ω e Θ]
4. $f(n) + g(n)$ ha complessità $O(max(f(n), g(n)))$ [anche per Ω e Θ]
5. Se $g(n) = O(f(n))$ e $h(n) = O(q(n))$, allora $g(n) \cdot h(n) = O(f(n)g(n))$ [anche per Ω e Θ]

Complessità	$n=10$	$n=20$	$n=50$	$n=100$	$n=10^3$	$n=10^4$	$n=10^5$	$n=10^6$
n	$10\mu s$	$20\mu s$	$50\mu s$	$0,1ms$	$1ms$	$10ms$	$0,1s$	$1s$
$n\log_2 n$	$33,2\mu s$	$86,4\mu s$	$0,28ms$	$0,6ms$	$9,9ms$	$0,1s$	$1,6s$	$19,9s$
n^2	$0,1ms$	$0,4ms$	$2,5ms$	$10ms$	$1s$	$100s$	$2,7h$	$11,5g$
n^3	$1ms$	$8ms$	$125ms$	$1s$	$16,6mn$	$11,5g$	$31,7a$	$\approx 300c$
2^n	$1ms$	$1s$	$35,7a$	$\approx 10^{14}c$	∞	∞	∞	∞
3^n	$59ms$	$58mn$	$\approx 10^8c$	∞	∞	∞	∞	∞

Complessità in tempo	*Max Dimensione* input
n	6×10^7
$n\log_2 n$	28×10^5
n^2	77×10^2
n^3	390
2^n	25

Complessità in tempo	*Max dim. su* $C1$	*Max dim. su* $C2$
n	d_1	$M\cdot d_1$
$n\lg n$	d_2	$\approx M\cdot d_2$ (per $d_2 \gg 0$)
n^2	d_3	$\sqrt{M}\cdot d_3$
2^n	d_4	$d_4+\lg M$

C2 è M volte più veloce di C1

- **Complessità di un algoritmo:** misura del numero di passi che si devono eseguire per risolvere il problema

- **Complessità di un problema:** complessità del migliore algoritmo che lo risolve

Classificazione di problemi

- Possiamo classificare i problemi in base alla quantità di risorse necessarie per ottenere la soluzione
- Per certi gruppi di problemi, le difficoltà incontrate per trovare un algoritmo efficiente sono sostanzialmente le stesse
- Possiamo raggruppare i problemi in tre categorie:
  1. I problemi che ammettono algoritmi di soluzione efficienti;
  2. I problemi che per loro natura non possono essere risolti mediante algoritmi efficienti e che quindi sono intrattabili;
  3. I problemi per i quali algoritmi efficienti non sono stati trovati ma per i quali nessuno ha finora provato che tali algoritmi non esistano

Classi di complessità

- Classi di problemi risolubili utilizzando una certa quantità di risorse (per esempio di tempo)
- **Problemi decidibili** = hanno una soluzione algoritmica
- **Problemi indecidibili** = **non** hanno una soluzione algoritmica

Classe di complessità P

- **Problemi polinomiali** =

  problemi per i quali esistono soluzioni praticabili, cioè di complessità in $\Theta(f(n))$ dove $f(n)$ è un **polinomio** oppure è limitato superiormente da un polinomio
- Esempi: ordinamento di una lista, ricerca in una lista
- Sono problemi **trattabili** = ammettono un algoritmo di soluzione efficiente

Problemi intrattabili

- Problemi che ***non*** possono essere risolti in un tempo polinomiale (quindi non appartengono alla classe P)
- Per questi problemi si può provare che ogni algoritmo risolutivo richiede, nel caso peggiore, un tempo di calcolo **esponenziale** o comunque asintoticamente superiore ad ogni polinomio
- Quindi pur essendo risolubili per via automatica, richiedono un tempo di calcolo molto elevato, tale da rendere ogni algoritmo di fatto inutilizzabile anche per dimensioni piccole dell'input

Classe di complessità NP

- **Problemi polinomiali non deterministici** = problemi risolvibili in tempo **polinomiale** da un **algoritmo non deterministico**, ma per i quali non è ancora stata trovata una soluzione deterministica in tempo polinomiale

- Algoritmo non deterministico = si basa sulla creatività del meccanismo che esegue il programma

Problema del commesso viaggiatore (classe NP)

Un commesso viaggiatore deve visitare alcuni suoi clienti in città diverse senza superare il budget per le spese di viaggio: il suo problema è trovare un percorso (che parta dalla sua abitazione, arrivi nelle varie città da visitare e poi lo riconduca a casa) la cui lunghezza totale non superi i chilometri consentiti

Problema del commesso viaggiatore (classe NP)

- Soluzione tipica:
  - Si considerano i potenziali percorsi in modo sistematico confrontando la lunghezza di ogni percorso con il limite chilometrico finché si trova un percorso accettabile oppure sono state considerate tutte le possibilità
- **Non** è una soluzione in tempo polinomiale
  - Il numero dei tragitti da considerare aumenta più velocemente di qualsiasi polinomio al crescere del numero delle città *(n!)*

Problema del commesso viaggiatore (classe NP)

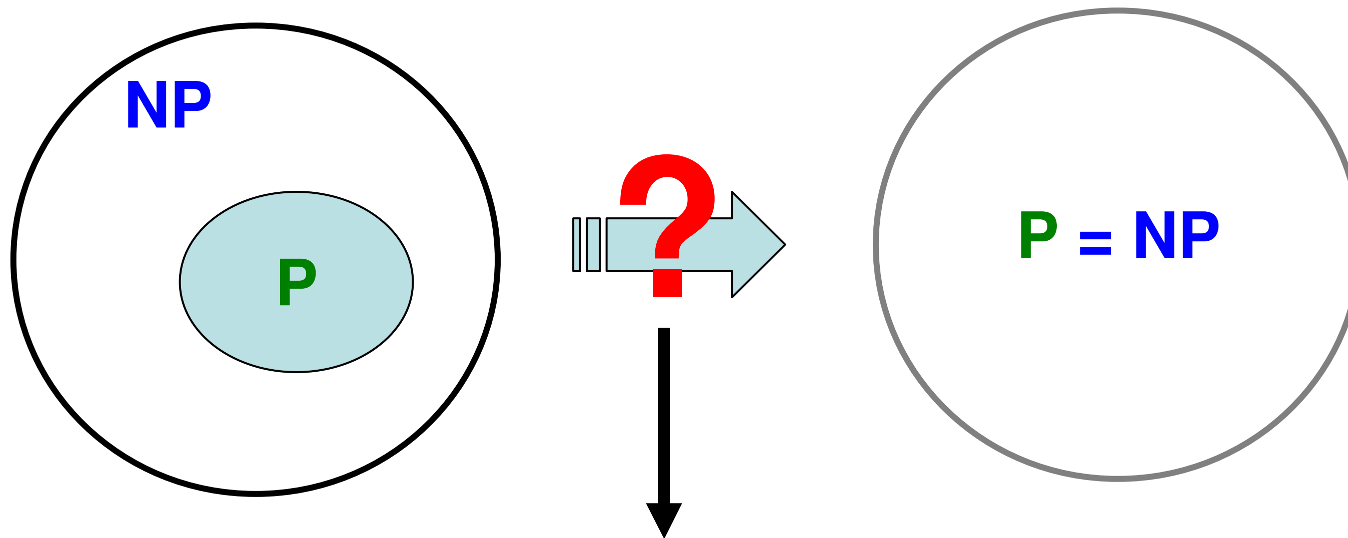
- **Algoritmo non deterministico**
  - Se esiste un percorso accettabile e lo selezioniamo per primo, l'algoritmo termina velocemente

```
Seleziona uno dei possibili percorsi e calcola la
sua distanza.
if(questa distanza non è
    maggiore del chilometraggio consentito)
    then (dichiara un successo)
    else (non dichiarare nulla)
```

istruzione ambigua

Relazione fra P e NP

- Tutti i problemi in P sono anche in NP



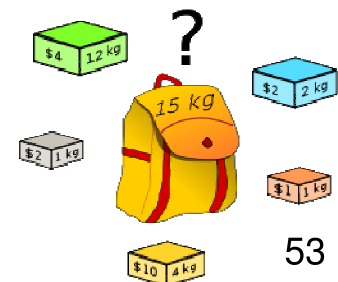
- Tutti i problemi in NP sono anche in P ?

Problemi NP-completi

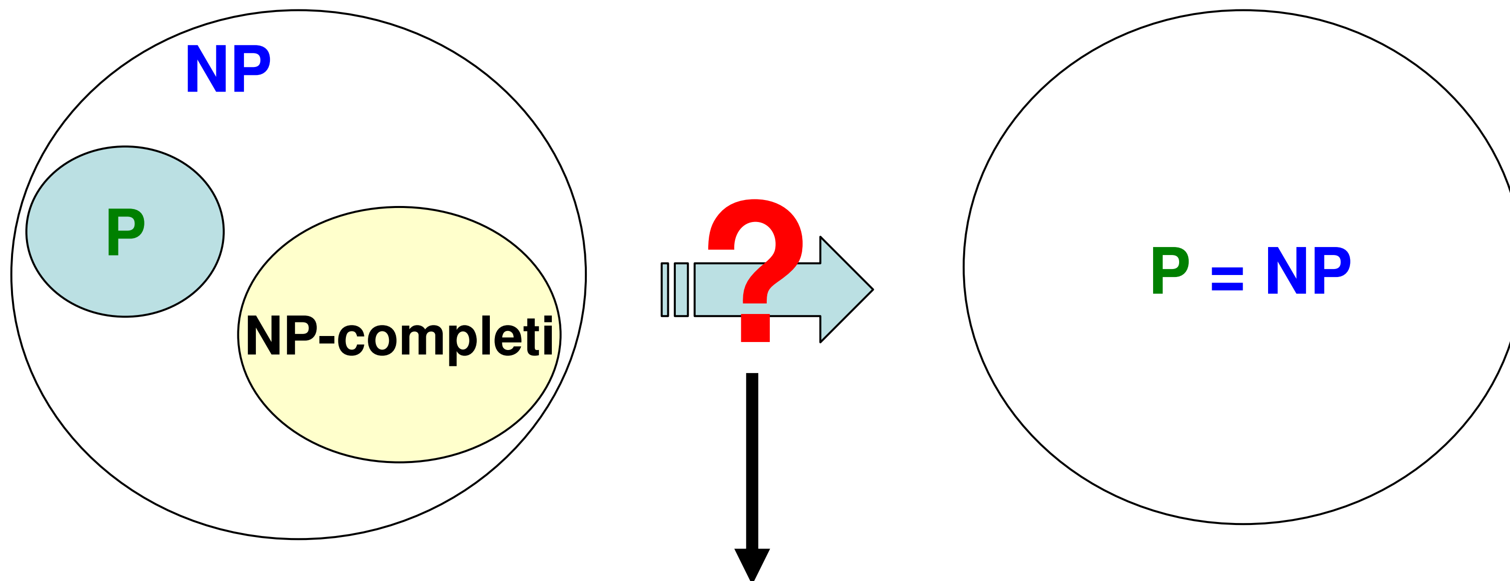
- Problemi in NP con la seguente proprietà:

  una soluzione deterministica polinomiale in termini di tempo per ognuno di essi fornirebbe anche una soluzione polinomiale per tutti gli altri problemi in NP

- Esempi: commesso viaggiatore, problema dello zaino, problema della clique

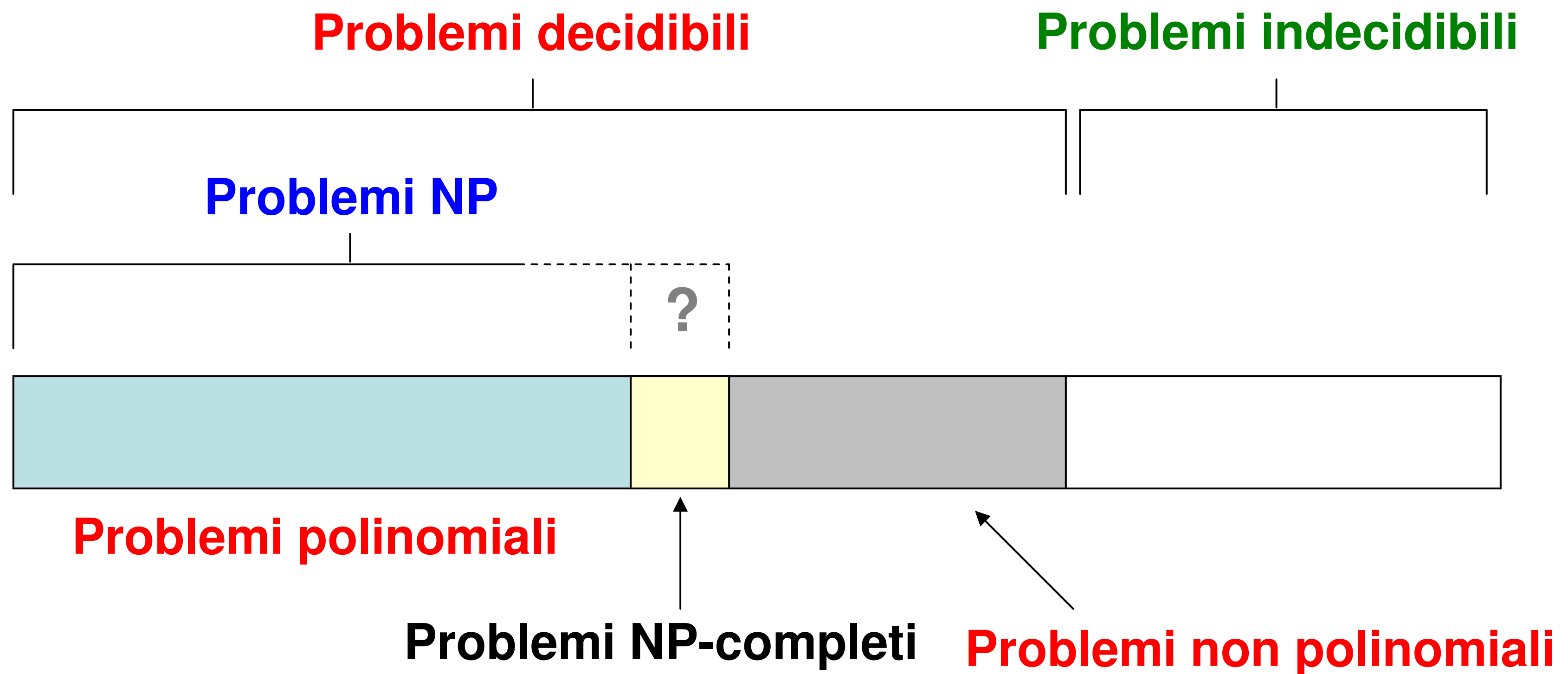


Relazione fra P e NP



- Se si dimostra che anche **solo** per **uno** dei problemi NP-completi esiste una soluzione deterministica polinomiale allora **P e NP sono equivalenti**

Classificazione dei problemi



Strutture dati: Tipo di dato

- I linguaggi di programmazione di alto livello consentono di far riferimento a posizioni nella memoria principale tramite nomi descrittivi (**variabili**) anziché indirizzi numerici
- **Dato**: un particolare valore che una variabile può assumere
- **Tipo di dato**: modello matematico che caratterizza l'insieme di valori che una variabile può assumere, e le operazioni che possono essere eseguite su di essa

Strutture dati: Tipi di dato primitivi

- Esempi:
  - **Intero** (integer):
    - dati numerici costituiti da numeri interi
    - Operazioni: aritmetiche e confronto
  - **Reali** (float, real)
    - dati numerici costituiti da numeri non interi
    - Operazioni simili a interi
  - **Booleani** (boolean)
    - Vero/falso
    - Operatori dell’algebra booleana, confronto
  - **Caratteri** (char)
    - Dati alfanumerici
    - Operazioni: concatenazione, confronto

Strutture dati

- Un modo per memorizzare e organizzare i dati e rendere efficiente l'accesso e la modifica dei dati stessi
- Esempi: array, matrici, grafi, alberi, tabelle hash, heap, liste, code, pile, …
- Una struttura dati consiste di:
  1. un modo sistematico di organizzare i dati
  2. un insieme di operatori che permettono di manipolare gli elementi della struttura

Tipi di strutture dati

- **Statiche**: la dimensione è definita al momento della creazione. Una volta terminato lo spazio allocato, è necessario creare un'altra struttura di dimensione maggiore dello stesso tipo e copiarvi dentro il contenuto della prima
  – Array, record
- **Dinamiche**: la dimensione della struttura dati può variare nel tempo senza limite. L'unico limite è la quantità di memoria disponibile nella macchina su cui andrà in esecuzione l'applicazione
  – Liste, code, pile, alberi, grafi

Tipi di strutture dati

- **Lineari**: i dati sono disposti in sequenza e possono essere nominati come primo, secondo, terzo, …
- **Non lineari**: i dati non sono disposti in sequenza
- **Omogenee**: i dati sono tutti dello stesso tipo
- **Non omogenee**: i dati sono di tipi diverso
- Esempio: il tipo di dato array rappresenta una struttura dati lineare, omogenea, a dimensione fissa

Strutture dati

- **Nozione astratta** svincolata dalla concreta rappresentazione della struttura nel modello di calcolo
- **Implementazione** descrive il modo con il quale la struttura è memorizzata e viene gestita dal calcolatore
  - Ogni struttura dati ammette più implementazioni con un costo diverso
    - Spazio di memorizzazione
    - Tempo per l'esecuzione delle operazioni primitive sulla struttura

Implementazione di una struttura dati

- Per valutare l'efficienza di procedure che usano tipi di dato primitivi si prescinde dalle caratteristiche specifiche di una macchina e si assume un' organizzazione abbastanza generica:
  - i dati sono contenuti in memoria
  - la memoria e divisa in celle, tutte di ugual ampiezza, ognuna delle quali può contenere un dato elementare
  - si accede ad una cella specificandone l'indirizzo
  - il tempo di accesso ad una cella si assume costante

Strutture dati statiche: array

- **Array** = blocco di elementi dello stesso tipo
- Implementazione di un array A di n elementi di tipo primitivo (int, char, ...):
  - è memorizzato in n celle consecutive a partire da un indirizzo ind_A
  - il tempo di accesso ad un generico elemento i è uguale al tempo di accesso della cella di indirizzo $ind_A + i$, quindi è **costante** o O(1)
  - Il passaggio per valore di A ad una procedura ha costo O(n)

Strutture dati statiche: record

- **Record** (o tipo aggregato) = blocco di dati in cui elementi diversi possono appartenere a tipi diversi
- Struct del linguaggio C

```
Struct {
  char Nome[25];
  int Età;
  float ValutazioneCapacità;
}Impiegato;
```

Strutture dati statiche: record

- Implementazione di un record a k campi: è memorizzato in celle di memoria consecutive:
- se i campi del record sono tutti dello stesso tipo allora la rappresentazione in memoria e analoga a quella di un vettore di k elementi
- se i campi del record sono di tipo diverso allora ogni campo occuperà un numero di celle pari al numero necessario per rappresentare il tipo di dato di ogni campo
  - L'accesso al campo di indice i del record ha un costo O(1)
  - Il passaggio per valore di un record ad una procedura ha costo:

$$\sum_{j=1}^{k} \text{sizeof}(\text{campo}_j) \in \mathcal{O}\left(\max_j (\text{sizeof}(\text{campo}_j))\right)$$