ALGORITMI:

COMPLESSITA' COMPUTAZIONALE in tempo e in spazio

tempo → irreversibile 1°

spazio → reversibile 2°

Costo computazionale in tempo di un algoritmo

numero di passi eseguiti (≈ # di operazioni elementari)

Costo computazionale in spazio di un algoritmo

numero di celle di memoria utilizzate durante la sua esecuzione, oltre alle celle occupate dai dati di input

↳ si esprimono in funzione della

DIMENSIONE dei DATI di INGRESSO (Istanza di input)

lunghezza della rappresentazione binaria dei dati di input

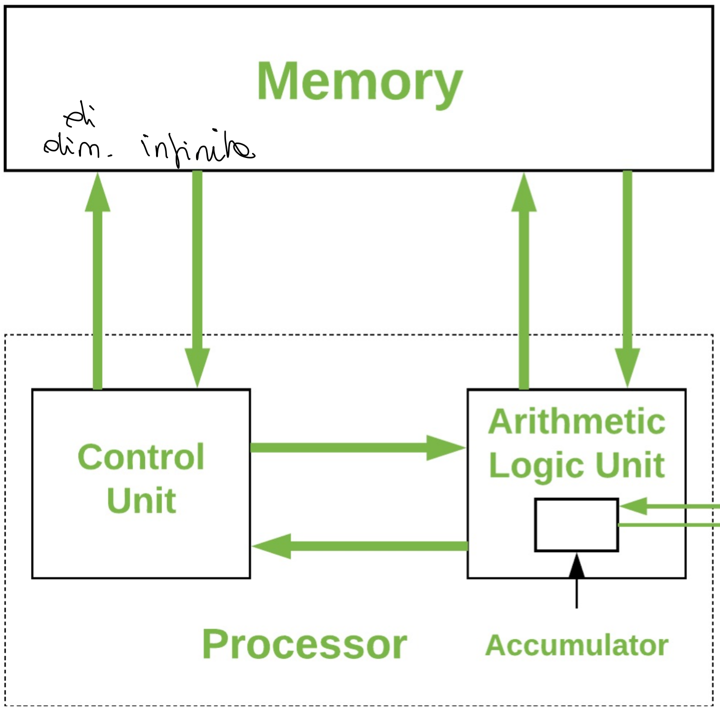
(o una misura equivalente)

↳ # di bit usati per scrivere i dati

ESEMPI	Input	Dim. input
AND di due variabili booleane a, b	a, b	costante
trovare min (o il max) in un array	array a	dimensione dell'array (a.length) ↳ n
somma / moltiplicazione interi grandi A, B	A, B	# di cifre di A e B

RAM

Random Access Machine
(Macchina ad accesso diretto)



Modello astratto

Schema molto semplificato di un calcolatore moderno
(architettura di Von Neumann)

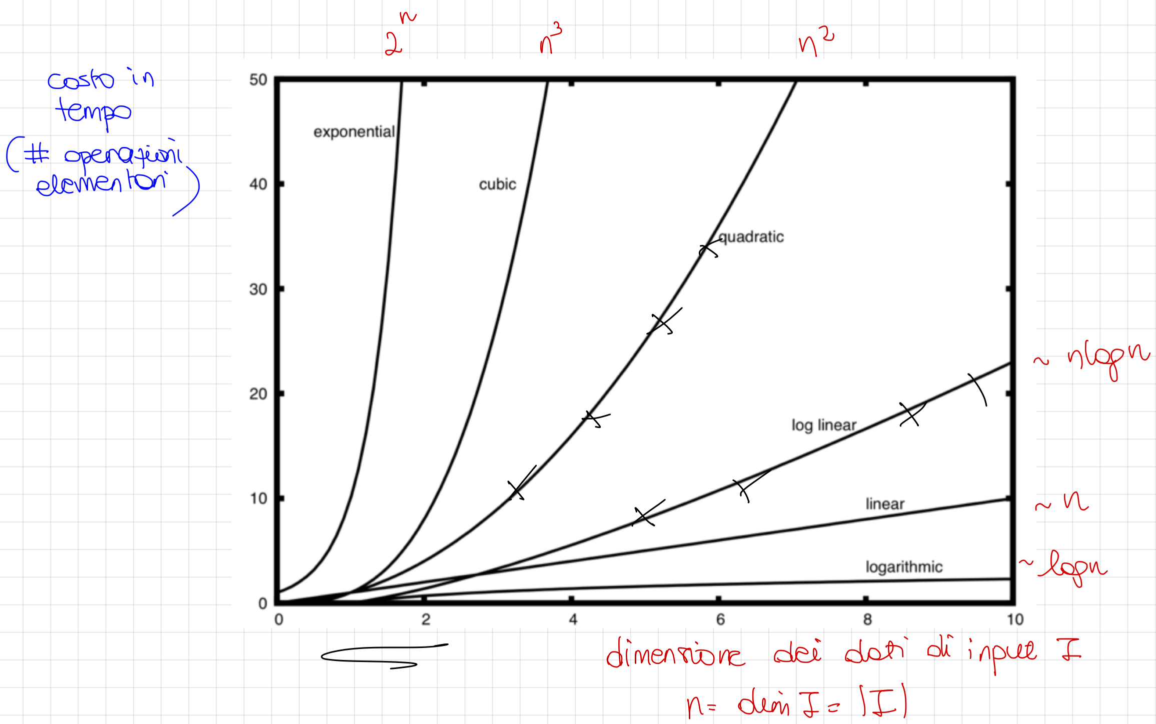
RAM: istruzioni/operazioni elementari

- operazioni aritmetiche (+, -, *, /)
- operazioni logiche (AND, OR, XOR, NOT, ..)
- operazioni di confronto (<, >, ==, ≠, <=, >=)
- operazioni di trasferimento (per spostare i dati: lettura e scrittura da memoria a accumulatore e viceversa)
- operazioni di controllo (per passare il controllo da un'istruzione ad un'altra e gestire salti, chiamate di funzioni)

IPOTESI DI COSTO UNIFORME

ogni operazione elementare richiede tempo costante di esecuzione

↳ # di operazione elementari fornisce una misura del tempo di esecuzione in ordine di grandezza (a meno ~~di~~ di fattori costanti)



L'algoritmo più efficiente è quello il cui costo in tempo cresce più lentamente al crescere della dimensione dei dati di input.

WORD MODEL

dati abbastanza piccoli da poter essere contenuti in una cella di memoria
↓
operazioni sui dati (confronto, somme, etc.) di costo in tempo costante

BIT MODEL

i numeri da elaborare crescono e non possono più essere contenuti in una cella di memoria

Lo costo delle operazioni si valuta sulle cifre

Analisi al caso ottimo, pessimo e medio

problema Π, algoritmo A che risolve Π

istanza di input I, di dimensione $\dim I = |I| = n$

Complessità al caso ottimo

Costo minimo su tutte le istanze possibili di dimensione n

$$T_{ottimo}(n) = \min_{I,\ |I|=n} \underbrace{T(I)}_{\text{costo di } A \text{ su } I}$$

Complessità al caso pessimo

Costo massimo su tutte le istanze possibili di dimensione n

$$T_{pess.}(n) = \max_{I,\ |I|=n} \underbrace{T(I)}_{\text{costo di } A \text{ su } I}$$

Complessità al caso medio

media del costo su tutte le istanze di dimensione n (pesate sulle probabilità con cui si presentano le diverse istanze di dim n)

$$T_{medio}(n) = \sum_{I,\ |I|=n} \underbrace{prob(I)}_{\text{probabilità che si presenti } I} * T(I)$$

↳ studieremo la complessità al caso pessimo

- il costo al caso pessimo fornisce un limite superiore al tempo di esecuzione su qualsiasi istanza di dim. n (l'algoritmo non impiegherà mai un tempo maggiore)
- il caso pessimo si presenta molto spesso per molti problemi
- il costo al caso medio è spesso sfavorevole come al caso pessimo, e l'analisi è più complessa

↳ Un algoritmo è considerato più efficiente di altri se il suo costo in tempo al caso PESSIMO ha un tasso di crescita inferiore (esegue meno operazioni)

ESEMPIO: RICERCA dell' ELEMENTO MASSIMO in un ARRAY di n INTERI NON ORDINATO

HAO

```
int[] v = [18,30,23];
int m = v[0]; // troviamo il voto più alto
int i = 1;
// calcoliamo l'esecuzione partendo da qui!
while (i < v.length) {
  if (m < v[i]) { m := v[i]; }
  i := i + 1;
}
```

$n = v.length$

ESEMPIO: RICERCA dell' ELEMENTO MASSIMO in un ARRAY di n INTERI NON ORDINATO

HAO

```
int[] v = [18,30,23];
int m = v[0]; // troviamo il voto più alto
int i = 1;
// calcoliamo l'esecuzione partendo da qui!
while (i < v.length) {
  if (m < v[i]) { m := v[i]; }
  i := i + 1;
}
```


MAX(v) // INPUT: array v di n interi: v.length = n

PseudoCodice

tempo costante
```
m = v[0];
n = v.length;
for (i = 1; i < n; i++) {
  { if (m < v[i]) m = v[i]; }
}
print m;  // output: m (valore max in v)
```

si ripete n-1 volte

tempo costante

i++ <=> i := i+1

$T(n)$: cresce come n (a meno di costanti)
sempre (al caso ottimo, medio e pessimo)

Non si può fare meglio (se l'array non è ordinato)

Metrica alternativa:
contiamo solo i confronti (che sono l'operazione principali)

$$C(n) = n-1$$

↳ $T(n) \sim C(n)$
il costo in tempo cresce ~~...~~ come $C(n)$

Ricerca Sequenziale in un array NON ordinato

INPUT: array a di n interi, non ordinato
intero k

OUTPUT: posizione di k in a, se k è presente — Ricerca con successo
-1 se k non occorre in a — Ricerca senza successo

```
int pos = -1;
int i = 0;
while (i < a.length && pos == -1) {
    if (a[i] == k) {
        pos := i;
    }
    i := i + 1;
} print pos;     //OUTPUT
```

Caso ottimo: $C(n) = 1$ $(a[0] = k)$

Caso pessimo $C(n) = n$ → $(a[n-1] = k)$
k non occorre in a
↳ ricerca senza successo

Caso medio (contiamo solo i confronti)

casi	# di confronti
k in posizione 0	1
" 1	2
" 2	3
⋮	
" n − 2	n − 1
" n − 1	n
k non è presente	n

n+1 casi

Abbiamo n+1 casi possibili

ipotesi: equiprobabili: ciascuno si può presentare con probabilità 1/n+1

$$C_{medio}(n) = \sum_{I : |I| = n} \text{prob}(I) * C(I) = \frac{1}{n+1}\Big(\underbrace{\sum_{i=1}^{n} i}_{k\ \text{presente}} + \underbrace{n}_{k\ \text{assente}}\Big)$$

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

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

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