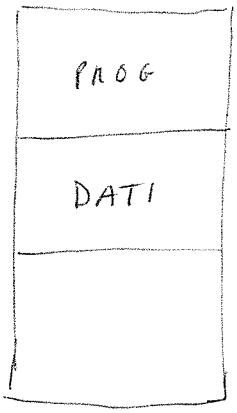
Modello RAM e Complessità Computazionale

Nell'analisi degli algoritmi incontrati nelle lezioni precedenti abbiamo parlato intuitivamente di "passi elementari".

Diamo ora una specifica formale di tale concetto.

I moderni calcolatori mantengono una struttura logica simile a quella introdotta negli anni '50 da John von Neumann che "tiene in memoria" sia i dati che i programmi come sequenze binarie.

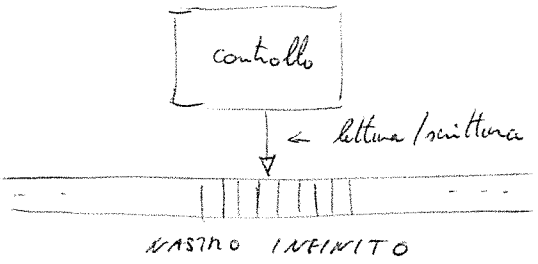


Memoria

Struttura di von Neumann



John von Neumann si ispirò alla macchina universale di Turing



Il modello RAM (Random Access Machine) rappresenta una ASTRAZIONE dei calcolatori reali.

Consiste di un processore di calcolo a cui viene associata una memoria di dimensione ILLIMITATA, in grado di contenere sia i dati sia i programmi.

Il processore dispone di un'unità centrale di elaborazione e di due registri, il CONTATORE di programma, che indica la prossima istruzione da eseguire, e l'ACCUMULATORE. Permette di eseguire le seguenti istruzioni elementari.

- operazioni aritmetiche: $+$, $-$, $*$, $/$
- operazioni di confronto: $<$, $>$, $=$, $\leq$, $\geq$ e così via
- operazioni logiche: and, or, not e così via
- operazioni di trasferimento: lettura e scrittura da ACCUMULATORE a memoria
- operazioni di controllo: salti condizionati e non condizionati

Per analizzare le prestazioni delle strutture di dati e degli algoritmi che studieremo, CONVENIAMO che le suddette operazioni abbiano COSTO UNIFORME.

Ognuna di esse richiede un TEMPO COSTANTE, indipendente dal numero di dati memorizzati nel calcolatore.

Il costo computazionale dell'esecuzione di un algoritmo è espresso in termini di

tempo	spazio
# di istruzioni elementari eseguite	massimo # di celle di memoria utilizzate durante l'esecuzione (oltre a quelle occupate dai dati)

Per ogni problema dato, possono esistere più algoritmi risolutivi.
Il progettista seleziona il migliore in termini di:

complessità di tempo	complessità di spazio

Entrambe le complessità quantificano ed esprimono il costo di un algoritmo attraverso una funzione $f(n)$ della dimensione n dei dati in ingresso (Input)

⇓

Dipende dal problema specifico in esame

e.g. - per un algoritmo di ordinamento, il numero di oggetti da ordinare
- per un algoritmo che verifica se un intero p è primo, il numero di cifre di p

Solitamente si cerca di minimizzare la complessità di tempo e, a parità di costo temporale, la complessità di spazio.

Spazio "riutilizzabile" Tempo "irreversibile"

Nell'analisi delle prestazioni di un algoritmo, tre casi sono di (diverso) interesse:

- caso pessimo: costo massimo su tutte le possibili istanze di dimensione n
- caso medio: costo mediato su tutte le possibili istanze, i.e., $\frac{\sum_i (\text{costo istanza } i)}{\#\text{ tot. istanze}}$
- caso ottimo: costo sull'istanza più favorevole

Il più interessante è solitamente il caso pessimo.

I costrutti che di norma vengono usati nello pseudocodice e nei linguaggi di programmazione, in termini di costo (caso pessimo) possono essere valutati come segue

→ IF (COND) { BLOCCO1 } ELSE { BLOCCO2 }

costo = costo(COND) + max { costo(BLOCCO1), costo(BLOCCO2) }

→ FOR (i = 0 ; i < m ; i = i + 1) { CORPO }

$$costo = c \cdot m + \sum_{i=0}^{m-1} t_i$$

dove
- t_i = costo della i-esima iterazione di CORPO
- $c \cdot m$ = costo valutazione condizione del ciclo m volte

→ WHILE (COND) { CORPO }

DO { CORPO } WHILE (COND) / esegue CORPO almeno una volta /

costo: sia m il # di volte in cui COND è soddisfatta

sia t'_i il costo della valutazione di COND alla i-esima iterazione

sia t_i il costo di CORPO all'i-esima iterazione

Si noti che COND viene valutata una volta in più rispetto a CORPO nel primo costrutto e lo stesso numero di volte nel secondo

Tuttavia, in entrambi i casi, il costo è al più

$$\sum_{i=0}^{m} (t_i' + t_i).$$

(Nelle analisi, rispetto al ciclo FOR, la parte difficile è valutare / stabilire su quanto vale.)

Infatti,

$$\left(\sum_{i=0}^{m} t_i' + \sum_{i=0}^{m-1} t_i\right) \qquad \left(\sum_{i=0}^{m} t_i' + \sum_{i=0}^{m} t_i\right)$$

↑ valutazione condizione — ↑ valutazione corpo

↑ valutazione condizione — ↑ valutazione corpo

WHILE (COND) — DO {CORPO} WHILE (COND)

$$\Rightarrow \quad \leq \sum_{i=0}^{m} (t_i' + t_i)$$

→ Chiamata a FUNZIONE

costo = costo del corpo della funzione + costo del passaggio dei parametri

→ BLOCCO di istruzioni e costrutti

costo = somma dei costi delle singole istruzioni e dei costrutti.

Esempi. Problema: calcolo del massimo di n numeri $a_0, a_1, \dots, a_{n-1}$

```
Max (a0, a1, -- an-1)                          Costo
max = a0;                                  →   c1
for (i=1; i<n; i=i+1) {                    →   c2 · n
                                                (condizione n volte)
   if (ai > max)                           }
                                           c3 → Σ_{i=1}^{n-1} c3
      max = ai;                                (corpo eseguito n-1 volte)
}
return max
```

$$\text{Costo totale} = c_1 + c_2 \cdot n + \sum_{i=1}^{n-1} c_3 = c_1 + c_2 \cdot n + c_3 \sum_{i=1}^{n-1} 1$$

$$= c_1 + c_2 \cdot n + c_3 (n-1) = c_1 + c_2 \cdot n + c_3 n - c_3$$

$$= \underbrace{(c_2 + c_3)}_{a} n + \underbrace{c_1 - c_3}_{b}$$

$$= a \cdot n + b \leq a' \cdot n$$

(al più un fattore costante per la dimensione n dell'input)

Costo $\in$ cresce linearmente in n.

Problema: Dati n punti, di coordinate $(x_1, y_1) \dots (x_n, y_n)$, determinare la coppia di punti più vicina.

```
Vicina ( (x1, y1), ... , (xn, yn) )
1.  min = sqrt( (x1 - x2)^2 + (y1 - y2)^2 )
2.  for (i = 1; i <= n; i = i+1) {
3.      for (j = i+1; j <= n; j = j+1) {
4.          d = sqrt( (xi - xj)^2 + (yi - yj)^2 );
5.          if (d < min) min = d;
        }
    }
    return d;
```

L'algoritmo calcola la distanza minima tra due punti
(Modificatelo per far restituire la coppia di punti più vicina)

Quanto costa?

A colpo d'occhio osserviamo che:

- quando $i = 1$, il costo è $c \cdot (n-1)$ (← esecuzioni 2° ciclo for)
  → (costo complessivo operazioni del 2° ciclo)
- quando $i = 2$, il costo è $c \cdot (n-2)$
- quando $i = 3$, il costo è $c \cdot (n-3)$

- quando $i = n-1$, il costo è $c \cdot 1$ (← il 2° ciclo for è eseguito una sola volta)

Pertanto, se sommiamo tutti i costi parziali, otteniamo

$$c\cdot(n-1) + c(n-2) + \dots + c\cdot 1 = c \cdot \underbrace{\sum_{k=1}^{n-1} k}_{\text{Formula di Gauss può essere usata}} = c \cdot \frac{n\cdot(n-1)}{2} \leftarrow d\cdot n^2$$

$$\downarrow$$

$$\frac{c}{2}\cdot n^2 - \frac{c}{2} n$$

se scelgo d opportunamente

Quindi, il costo dell'algoritmo è <u>al più</u> un fattore costante per il quadrato della dimensione dell'input n.

<u>Nota 1</u>: formula di Gauss

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} = S_n \qquad \left(\begin{array}{c}\text{Somma dei}\\ \text{primi } n \text{ numeri}\end{array}\right)$$

Dim.

$$\begin{array}{lllll} 1 & + 2 & + 3 & + \dots & + n \\ n & + n-1 & + n-2 & + n-3 & \dots + 1 \\ \hline (n+1) & + (n+1) & + \dots & & + (n+1) \end{array} \qquad \begin{array}{l} = S_n \\ + \\ = S_n \\ \hline 2 S_n \end{array}$$

n volte

$$\Rightarrow \quad 2\cdot S_n = n\cdot(n+1) \quad \Longleftrightarrow \quad S_n = \frac{n(n+1)}{2}$$

Nota 2: Ovviamente, anche se facciamo un'analisi del costo più puntuale, raggiungiamo la stessa conclusione

costi: linea 1 $\longrightarrow$ c_1

linea 2 $\longrightarrow$ $c_2 \cdot (n) + \sum_{i=1}^{n-1} t_i$

- $c_2 \cdot (n)$: costo condizione 1° for, ripetuto n volte
- $\sum_{i=1}^{n-1} t_i$: costo CORPO 1° for, ripetuto $n-1$ volte

linea 3 $\longrightarrow$ $c_3 \cdot (n-i) + \sum_{j=1}^{n-i} t_j$

- $c_3 \cdot (n-i)$: costo condizione 2° for, ripetuto $n-i$ volte
- $\sum_{j=1}^{n-i} t_j$: costo CORPO 2° for, ripetuto $n-i$ volte

c_4 $\longleftarrow$ linee 4 e 5

In totale, quindi:

$$c_1 + c_2 \cdot n + \sum_{i=1}^{n-1} t_i \qquad \text{dove} \qquad t_i = \left(c_3 \cdot (n-i) + \sum_{j=1}^{n-i} c_4\right)$$

$$= c_1 + c_2 n + \sum_{i=1}^{n-1} (c_3 + c_4)(n-i)$$

$$= c_1 + c_2 \cdot n + (c_3 + c_4) \cdot \sum_{i=1}^{n-1} (n-i)$$

$$(n-1) + (n-2) + \dots + 1$$

$$= c_1 + c_2 \cdot n + (c_3 + c_4) \cdot \frac{n \cdot (n-1)}{2} \leq d \cdot n^2$$

per una costante d scelta opportunamente.

$$= c_3 (n-i) + c_4 \cdot \sum_{j=1}^{n-i} 1 =$$

$$= c_3 (n-i) + c_4 (n-i) \cdot 1$$

$$= (c_3 + c_4)(n-i).$$

Osservazione: supponiamo di avere a disposizione due algoritmi di ordinamento per i quali abbiamo fatto "i conti" e risultano avere complessità temporali uguali a $2n^2$ e $50n \cdot \log n$.

Sia A un calcolatore che esegue 1 miliardo (10^9) di istruzioni al secondo.

Sia B un calcolatore che esegue 10 milioni (10^7) di istruzioni al secondo.

$\Rightarrow$ A molto più potente di B

Proviamo ad ordinare $n = 10^6$ (un milione) di numeri dando ad A il primo algoritmo e a B il secondo

		complessità algoritmo		
$\frac{10^9 \text{ istruzioni}}{\text{secondo}} \rightarrow$	Calcolatore A	$2 \cdot n^2$	$\frac{2 \cdot (10^6)^2}{10^9}$	2000 secondi
$\frac{10^7 \text{ istruzioni}}{\text{secondo}} \rightarrow$	Calcolatore B	$50 \cdot n \cdot \log n$	$\frac{50 \cdot (10^6) \cdot \log(10^6)}{10^7}$	100 secondi

Conclusione: B ci mette molto meno tempo di A!

Un algoritmo il cui costo è rappresentato da una funzione "più piccola", al crescere della dimensione del problema, si comporta decisamente meglio di uno con costo quantificato da una funzione "più grande"

Inoltre, le costanti (ed eventuali termini più piccoli) nella funzione influenzano poco o nulla l'esito della comparazione.

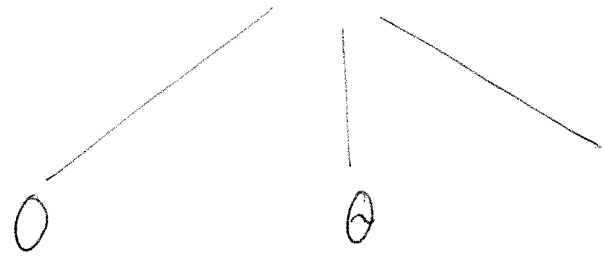
Poiché nelle applicazioni in cui vengono usati gli algoritmi spesso si presentano input di grosse dimensioni, ~~Il~~ costo degli algoritmi viene valutato "al crescere di n" (la dimensione dell'input), per n che diventa grande.

Pertanto, si parla di "analisi asintotica".

Per discutere in termini rigorosi è stata sviluppata una apposita notazione che viene detta "notazione asintotica".

La notazione asintotica, informalmente parlando, ignora le "costanti moltiplicative" e gli "ordini inferiori" della funzione $f(n)$ al crescere di n.

Utilizza tre simboli:



... richiede al più — ... richiede esattamente — ... richiede almeno

$f(n) = 9n^3 + 2n^2 + 3n$ → $O(n^3)$

$f(n) = 7n^2 + 5n\log n + 3$ → $\Theta(n^2)$

$f(n) = 5n^4 + 2n^2 + n + 1$ → $\Omega(n^4)$

O → l'algoritmo richiede al più tempo (spazio) che è ordine di ...

Θ → l'algoritmo richiede esattamente, né più né meno

Ω → l'algoritmo non può far meglio di ...

Una analogia che aiuta a capire il senso delle valutazioni asintotiche, trascurando costanti e termini di ordine inferiore, è la seguente: "un MILIARDARIO è tale sia che possegga 7 miliardi, sia che ne possegga 9 e svariati milioni".

La valutazione asintotica è molto utile per scartare velocemente gli algoritmi inefficienti rispetto a quelli efficienti, senza il bisogno di implementarli.

Ovviamente, per quelli efficienti, al momento dell'implementazione, occorre tener conto del particolare sistema (piattaforma HW/SW, sistema operativo, linguaggio di programmazione, contesto d'uso ...)

Il modello RAM volutamente ignora questi dettagli per un'analisi di alto livello.

Limiti superiori e inferiori ai problemi

Per un dato problema computazionale Π, consideriamo un qualsiasi algoritmo risolutore A.

Se A richiede tempo $t(n)$ per risolvere una generica istanza di Π (di dimensione n), diremo che $O(t(n))$ è un limite superiore alla complessità in tempo del problema Π

Quando riusciamo a dimostrare, attraverso argomenti che tengono conto della natura e della struttura del problema, che qualunque algoritmo A' richiede almeno tempo $f(n)$ per risolvere il problema (un'istanza generica di Π, di dimensione n), allora diremo che $\Omega(f(n))$ è un limite inferiore alla complessità in tempo di Π.

Nessun algoritmo può asintoticamente richiedere meno di $O(f(n))$.

Ne deriva che UN ALGORITMO A è OTTIMO se $t(n)$ è $O(f(n))$ ovvero se la sua complessità asintotica coincide al limite inferiore.

Quando esiste un algoritmo OTTIMO diremo che la complessità di Π è $\Theta(f(n))$

Attenzione a non confondere la complessità in tempo di un algoritmo A con la complessità di un problema Π.

Per lo spazio si procede analogamente.