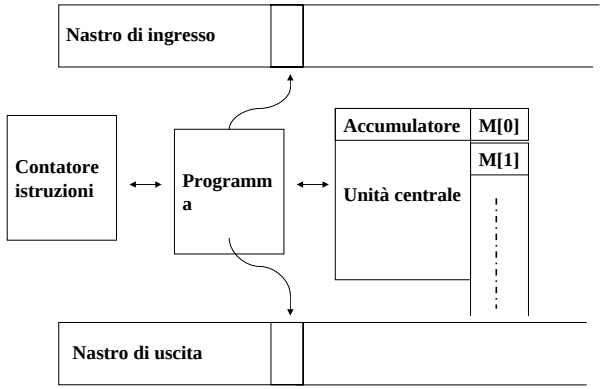
Algoritmi e Strutture Dati

Modelli di calcolo e metodologie di analisi

Basato su materiale di C. Demetrescu, I. Finocchi, G.F. Italiano

Macchina RAM



Linguaggio di una macchina RAM

- Un *programma* (algoritmo) è una sequenza di istruzioni, eventualmente marcate da *etichette.*
- Istruzioni
  - trasferimento: LOAD, STORE
  - aritmetiche: ADD, SUB, MULT, DIV, REM
  - controllo: HALT, JUMP, JGTZ, JEQZ
  - I/O: READ, WRITE
- Operando
  - riferimento diretto
  - riferimento indiretto (*)
  - valore immediato (=)

Esempio: Dati x e y, calcolare x^y

```
        READ  1
        READ  2
        LOAD  =1
        STORE 3
CICLO   LOAD  2
        JEQZ  FINE
        LOAD  1
        MULT  3
        STORE 3
        LOAD  2
        SUB   1
        STORE 2
        JUMP  CICLO
FINE    WRITE 3
        HALT
```

4 operazioni

2(y+1) operazioni

7y operazioni

2 operazioni

Complessità temporale = 9y + 8

Complessità spaziale = 1

Esempio: Invertire n interi

```
        READ  1          OUT   WRITE *2
        LOAD  =2               LOAD  2
        STORE 2                SUB   1
        LOAD  1                STORE 2
IN      JEQZ  OUT              SUB   2
        LOAD  2                JGTZ  OUT
        ADD   1                HALT
        STORE 2
        READ  *2
        LOAD  1
        SUB   1
        STORE 1
        JUMP  IN
```

4 operazioni

(n+1) operazioni

6n operazioni

1 operazione

8n operazioni

Complessità temporale = 15n + 6 Complessità spaziale = 1

Dimensione dell'input

- *Lunghezza dell'input:*
  - Nr. di bit che costituiscono l'input in una opportuna codifica (*registri di lunghezza variabile*)
  - Numero di registri usati (*registri di lunghezza fissa*)
- Data una stringa *w* di input, |*w*| ne indica la *lunghezza*
  - *per un numero n, |n|= log n+1*
- La *dimensione* è la lunghezza dell'input o un parametro che la caratterizza

ESEMPI:

Problema: x^y
Dimensione:
- **2**
- **log *x* + log *y***

Problema: ***Inverti***
Dimensione:
- **n+1**
- **<= log *n* + *n* log *vMax***

Macchina RAM: misure di complessità

- Spazio:
  - numero di registri impegnati
  - numero di bit utilizzati
- Tempo:
  - $\sum_{i=1,m} k_i t_i$ (tempi diversi per le varie operazioni)
  - numero di operazioni (*costi uniformi)*
    - *Hp:* registri di capienza illimitata, o valori limitati)
  - ogni operazione ha un costo proporzionale alla dimensione degli operandi *(costi logaritmici)*
    - Un valore può occupare più registri
    - Es: dati due interi n e m di valore illimitato, il prodotto n*m ha costo log(n) + log(m) e non costante!

Esempi di costi temporali su RAM

x^y

- Costi uniformi: t(*y*)= 9 *y* + 8
- assumendo *k*=|*y*|=log *y*, $T(k)= 9\times2^k + 8$

Inverti

- t(*n*)= 15 *n* + 86
- assumendo *k*=*n*, la funzione di costo rimane lineare

Analisi asintotica

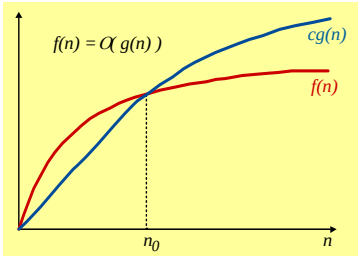
- Calcolo esatto delle funzioni di costo
  - troppo complicato
  - dettaglio eccessivo
- Importante analizzare il comportamento dell'algoritmo al crescere della dimensione dell'input (verso infinito)
- Possiamo ignorare
  - Eventuali fattori costanti
  - Termini non dominanti asintoticamente
- E' necessario uno strumento formale di scrittura approssimata

Notazione asintotica O

f(n) = tempo di esecuzione / occupazione di memoria di un algoritmo su input di dimensione n

f(n) = O(g(n))

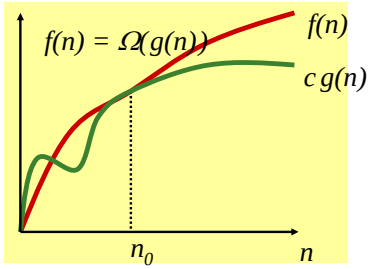
se ∃ due costanti c>0 e $n_0 \geq 0$ tali che $f(n) \leq c\, g(n)$ per ogni $n \geq n_0$



Notazione asintotica Ω

f(n) = Ω(g(n))

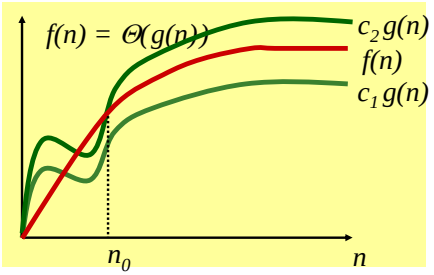
se ∃ due costanti c>0 e $n_0 \geq 0$ tali che $f(n) \geq c\, g(n)$ per ogni $n \geq n_0$



Notazione asintotica Θ

f(n) = Θ(g(n))

se ∃ tre costanti $c_1, c_2 > 0$ e $n_0 \geq 0$ tali che $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ per ogni $n \geq n_0$



Notazione asintotica: esempi

Sia $g(n)=3n^2+10$

- $g(n)=O(n^2)$: scegliere c=4 e $n_0=10$
- $g(n)=\Omega(n^2)$: scegliere c=1 e $n_0=0$
- $g(n)=\Theta(n^2)$: infatti $g(n)= \Theta(f(n))$ se e solo se $g(n)= O(f(n))$ e $g(n)= \Omega(f(n))$
- $g(n)=O(n^3)$ ma $g(n)\neq\Theta(n^3)$

Alcune funzioni: andamento al crescere di *n*

funzione \ *n*	1	x10	x20	x30	x40	X50
n	0.000001	0.00001	0.00002	0.00003	0.00004	0.00005
n^2	0.000001	0.0001	0.0004	0.0009	0.0016	0.0025
n^3	0.000001	0.001	0.008	0.027	0.064	0.125
2^n	0.000001	0.001	1.0	17.9 min	12.7 giorni	35.7 anni
3^n	0.000001	0.059	58 min	6.5 anni	3855 secoli	$2x10^8$ secoli

Analisi per linguaggi ad alto livello

- Modello a costi uniformi
  - Tutte le operazioni hanno costo unitario
  - Individuazione delle istruzioni "dominanti"
  - *Esempio:*

    for(int i=0;i<n;i++) somma+=i;

    somma+=i è una istruzione dominante
- Istruzione dominante
  - il suo contributo al costo di esecuzione è nel caso peggiore *d(n)*
  - *T(n)* è *O(d(n))*

Modello a costi logaritmici

Una macchina reale non può operare a costo costante su numeri arbitrariamente grandi

- per manipolare un intero *n* servono *log(n)*, ossia il costo è proporzionale al numero di bit occupato
- l'accesso all'elemento *i-esimo* di un vettore avviene in tempo proporzionale a *log(i)*

Domanda:
quanto costa calcolare i numeri di Fibonacci?

Osservazione
In molti casi pratici il modello a costi uniformi può bastare

Esempio: x^y

```
public static int potenza(int x, int y) {
  int p=1;
  for (int i=1; i<y; i++) p *= x;
  return p;
}
```

Siano k=log y, m= log x

Complessità:

- Con costi uniformi è $\Theta(y) = \Theta(2^k)$
- Con costi logaritmici, è $O(y^2 \log x + y \log y) = O(m\, 2^{2k} + k\, 2^k)$
  - $dim(x) \leq dim(p) \leq \log(x^y) = y \log x$
  - $dim(i) \leq dim(y) = \log y$

Metodi di analisi

Caso peggiore, migliore e medio

- Misureremo le risorse di calcolo usate da un algoritmo (tempo/memoria) rispetto alla dimensione n delle istanze
- Istanze diverse, a parità di dimensione, potrebbero però richiedere risorse diverse
- Distinguiamo quindi ulteriormente tra analisi nel caso peggiore, migliore e medio

Caso peggiore

- Sia *costo(I)* il costo di esecuzione di un algoritmo sull'istanza *I*

$$T_{worst}(n) = \max_{\text{istanze } I \text{ di dimensione } n} \{ costo(I) \}$$

- Intuitivamente, $\boldsymbol{T_{worst}(n)}$ è il costo di esecuzione sulle istanze di ingresso che comportano più lavoro per l'algoritmo

Caso migliore

- Sia *costo(I)* il costo di esecuzione di un algoritmo sull'istanza *I*

$$T_{best}(n) = \min_{\text{istanze } I \text{ di dimensione } n} \{costo(I)\}$$

- Intuitivamente, $T_{best}(n)$ è il costo di esecuzione sulle istanze di ingresso che comportano meno lavoro per l'algoritmo

Caso medio

- Sia *P(I)* la probabilità di avere in ingresso un'istanza *I*

$$T_{avg}(n) = \sum_{\text{istanze I di dimensione n}} \{P(I) \times costo(I)\}$$

- Intuitivamente, $T_{avg}(n)$ è il costo di esecuzione nel caso medio, ovvero sulle istanze di ingresso "tipiche" per il problema
- Richiede conoscenza di una distribuzione di probabilità sulle istanze
  - Solitamente conviene partizionare le istanze di dimensione *n* in classi di equivalenza rispetto alla funzione di costo

Esempio 1

Ricerca di un elemento *x* in una lista *L* non ordinata

```
algoritmo ricercaSequenziale(lista L, elem x) → booleano
1.   for each ( y ∈ L ) do
2.       if ( y = x ) then return trovato
3.   return non trovato
```

$T_{best}(n) = 1$	x è in ultima posizione
$T_{worst}(n) = n$	$x \notin L$ oppure è in ultima posizione
$T_{avg}(n) = (n+1)/2$	assumendo che le istanze siano equidistribuite (e che $x \in L$)

Esempio 2 (1/2)

Ricerca di un elemento *x* in un array *L* ordinato

```
algoritmo ricercaBinariaIter(array L, elem x) → booleano
1.   a ← 1
2.   b ← lunghezza di L
3.   while ( L[(a + b)/2] ≠ x ) do
4.       m ← (a + b)/2
5.       if ( L[i] > x ) then b ← m − 1
6.       else a ← m + 1
7.       if ( a > b ) then return non trovato
8.   return trovato
```

Confronta *x* con l'elemento centrale di *L* e prosegue nella metà sinistra o destra in base all'esito del confronto

Esempio 2 (2/2)

$T_{best}(n) = 1$ l'elemento centrale è uguale a x

$T_{worst}(n) = \log(n)$ $x \notin L$ oppure viene trovato all'ultimo confronto

La dimensione del sotto-array su cui si procede si dimezza dopo ogni confronto;
dopo l'i-esimo confronto il sottoarray di interesse ha dimensione $n/2^i$

Risulta $n/2^i = 1$ per $i=\log_2 n$

$T_{avg}(n)= \log(n) -1+1/n$ assumendo che le istanze siano equidistribuite

Complessità di un problema

- E' la complessità dell'algoritmo più efficiente che lo risolve
- Metodi per stimare un lower bound (Ω) per tutti gli algoritmi:
  - ispezione dell'input, metodo dell'avversario, alberi di decisione
- Esempi:
  - La ricerca in vettore ordinato di dimensione *n* ha complessità ***$\log_2(n)$***
  - Ordinare un vettore di dimensione *n* ha complessità ***$n \log_2(n)$***

Analisi di algoritmi ricorsivi

Esempio

L'algoritmo di ricerca binaria può essere riscritto ricorsivamente come:

algoritmo `ricercaBinariaRic`$(array\ L,\ elemento\ x) \rightarrow booleano$

1. $n \leftarrow$ lunghezza di L
2. **if** $(n = 0)$ **then return** non trovato
3. $i \leftarrow \lceil n/2 \rceil$
4. **else if** $(L[i] = x)$ **then return** trovato
5. **else if** $(L[i] > x)$ **then return** `ricercaBinariaRic`$(x, L[1; i-1])$
6. **else return** `ricercaBinariaRic`$(x, L[i+1; n])$

Come analizzarlo?

Equazioni di ricorrenza

Il tempo di esecuzione può essere descritto tramite una equazione di ricorrenza:

$$T(n) = \begin{cases} c + T(\lceil n-1 \rceil/2) & \text{se } n>1 \\ 1 & \text{se } n=1 \end{cases}$$

Vari metodi per risolvere equazioni di ricorrenza: iterazione, sostituzione, teorema Master...

Metodo dell'iterazione

Idea: "srotolare" la ricorsione, ottenendo una sommatoria dipendente solo dalla dimensione n del problema iniziale

Esempio: $T(n) = c + T(n/2)$

$T(n/2) = c + T(n/4)$ ...

➡ $T(n) = c + T(n/2) = 2c + T(n/4) =$
$= (\sum_{j=1...i} c) + T(n/2^i) = i\,c + T(n/2^i)$

Per $i=\log_2 n$: $T(n) = c \log n + T(1) = O(\log n)$

Metodo della sostituzione

Idea: "indovinare" una soluzione, ed usare induzione matematica per provare che essa sia effettivamente la soluzione dell'equazione di ricorrenza

Esempio: $T(n) = n + T(n/2)$, $T(1)=1$

Assumiamo che la soluzione sia $T(n) \leq c\,n$ per una costante c opportuna

Passo base: $T(1)=1 \leq c\,1$ per ogni c

Passo induttivo: $T(n)= n + T(n/2) \leq n + c\,(n/2) = (c/2+1)\,n$

Quindi $T(n) \leq c\,n$ per $c \geq 2$

Teorema Master

Permette di analizzare algoritmi basati sulla tecnica del *divide et impera*:

- dividi il problema (di dimensione n) in a sottoproblemi di dimensione n/b
- risolvi i sottoproblemi ricorsivamente
- ricombina le soluzioni

Sia f(n) il tempo per dividere e ricombinare istanze di dimensione n. La relazione di ricorrenza è data da:

$$T(n) = \begin{cases} a\,T(n/b) + f(n) & \text{se } n>1 \\ 1 & \text{se } n=1 \end{cases}$$

Teorema Master

La relazione di ricorrenza:

$$T(n) = \begin{cases} a\,T(n/b) + f(n) & \text{se } n>1 \\ 1 & \text{se } n=1 \end{cases}$$

ha soluzione:

1. $T(n) = \Theta(n^{\log_b a})$ se $f(n)=O(n^{\log_b a-\varepsilon})$ per $\varepsilon>0$
2. $T(n) = \Theta(n^{\log_b a} \log n)$ se $f(n) = \Theta(n^{\log_b a})$
3. $T(n) = \Theta(f(n))$ se $f(n)=\Omega(n^{\log_b a+\varepsilon})$ per $\varepsilon>0$ e $a\,f(n/b)\leq c\,f(n)$ per $c<1$ e n sufficientemente grande

Esempi

1) $T(n) = n + 2T(n/2)$
   $a=2, b=2, f(n)=n=\Theta(n$) ➡ $T(n)=\Theta(n \log n)$
   (caso 2 del teorema master)
2) $T(n) = c + 3T(n/9)$
   $a=3, b=9, f(n)=c=O(n$) ➡ $T(n)=\Theta(\sqrt{n})$
   (caso 1 del teorema master)
3) $T(n) = n + 3T(n/9)$
   $a=3, b=9, f(n)=n=\Omega(n$) ➡ $T(n)=\Theta(n)$
   (caso 3 del teorema master)

Riepilogo

- Esprimiamo la quantità di una certa risorsa di calcolo (tempo, spazio) usata da un algoritmo in funzione della dimensione n dell'istanza di ingresso
- La notazione asintotica permette di esprimere la quantità di risorsa usata dall'algoritmo in modo sintetico, ignorando dettagli non influenti
- A parità di dimensione n, la quantità di risorsa usata può essere diversa, da cui la necessità di analizzare il caso peggiore o, se possibile, il caso medio
- La quantità di risorsa usata da algoritmi ricorsivi può essere espressa tramite relazioni di ricorrenza, risolvibili tramite vari metodi generali