Programmazione I

Il Linguaggio C

Ricorsione e Complessità

Daniel Riccio

Università di Napoli, Federico II

15 dicembre 2021

Sommario

- Argomenti
  - Esempi di ricorsione
  - Ricorsione e funzioni iterative con stack
  - Cenni sulla complessità

Divide et Impera

Metodo di approccio ai problemi che consiste nel dividere il problema dato in problemi più semplici

I risultati ottenuti risolvendo i problemi più semplici vengono combinati insieme per costituire la soluzione del problema originale

Generalmente, quando la semplificazione del problema consiste essenzialmente nella semplificazione dei DATI da elaborare (ad es. la riduzione della dimensione del vettore da elaborare), si può pensare ad una soluzione ricorsiva

La ricorsione

Una funzione è detta **ricorsiva** se chiama se stessa

Se due funzioni si chiamano l'un l'altra, sono dette **mutuamente ricorsive**

La funzione ricorsiva sa risolvere direttamente solo casi particolari di un problema detti **casi di base**: se viene invocata passandole dei dati che costituiscono uno dei casi di base, allora restituisce un risultato

Se invece viene chiamata passandole dei dati che NON costituiscono uno dei casi di base, allora **chiama se stessa** (**passo ricorsivo**) passando dei DATI semplificati/ridotti

Analisi

L’apertura delle chiamate ricorsive semplifica il problema, ma non calcola ancora nulla

Il valore restituito dalle funzioni viene utilizzato per calcolare il valore finale man mano che si chiudono le chiamate ricorsive: ogni chiamata genera valori intermedi a partire dalla fine

Nella ricorsione vera e propria non c’è un mero passaggio di un risultato calcolato nella chiamata più interna a quelle più esterne, ossia le return non si limitano a passare indietro invariato un valore, ma c’è un’elaborazione intermedia

La ricorsione

PRO

Spesso la ricorsione permette di risolvere un problema anche molto complesso con poche linee di codice

CONTRO

La ricorsione è poco efficiente perché richiama molte volte una funzione e questo:

- richiede tempo per la gestione dello stack (allocare e passare i parametri, salvare l'indirizzo di ritorno, e i valori di alcuni registri della CPU)
- consuma molta memoria (alloca un nuovo stack frame ad ogni chiamata, definendo una nuova ulteriore istanza delle variabili locali non static e dei parametri ogni volta)

Considerazioni

Una funzione ricorsiva non dovrebbe, in generale, eseguire a sua volta più di una chiamata ricorsiva

Esempio di utilizzo da evitare:

```
long fibo(long n)
{
    if(n<=1)
        return 1;
    else
        return fibo(n-1) + fibo(n-2);
}
```

Ogni chiamata genera altre 2 chiamate, in totale vengono effettuate 2^n chiamate (complessità esponenziale)

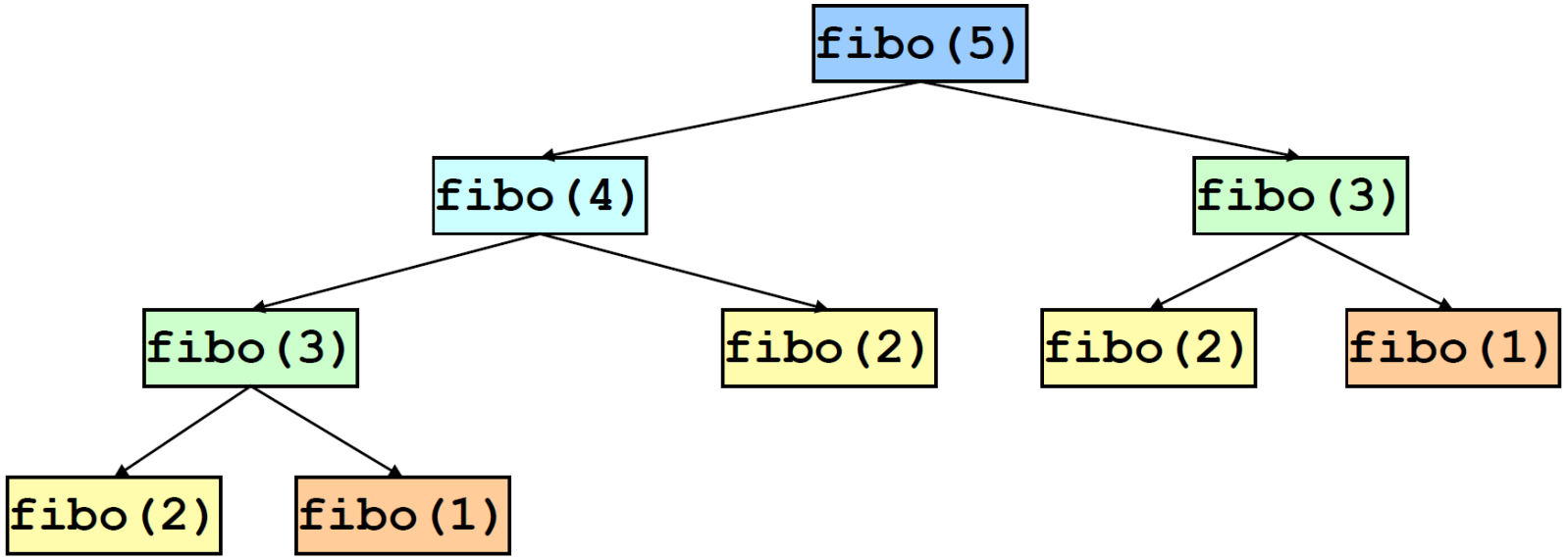
Considerazioni

Inoltre si osservi che:

fibo(n) chiama **fibo**(n-1) e **fibo**(n-2).

fibo(n-1) chiama **fibo**(n-2), ecc.

si hanno calcoli ripetuti, inefficiente!



Ricorsione con stack esplicito

L'uso di uno stack esplicito permette di simulare le chiamate ricorsive ad una funzione.

Ricorsione

Stack dei record di attivazione

Caso base → return valore

Caso generico n → chiamata ricorsiva su m<n

Condizione di arresto == caso base

Iterazione

Stack esplicito **S**

Caso base → accumulo in una variabile F

Caso generico n → inserisco m<n nello stack

Condizione di arresto == stack vuoto

Fibonacci con stack esplicito

```
long fibo(long n)
{
    if(n<=1)
        return 1;
    else
        return fibo(n-1) + fibo(n-2);
}
```

```
APush(&S, n);
F=0;
while(AStackVuoto(&S)==0){
    APop(&S, &n);
        if(n <= 1)
            F = F + 1;
        else {
            APush(&S, n-1);
            APush(&S, n-2);
}
```

Fibonacci con stack esplicito

```c
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

int main(int argc, char *argv[])
{
  int n;
  int F=0;
  AStack S;

  CreaAStack(&S, 100);

  n = atoi(argv[1]);

  APush(&S, n);

  while(AStackVuoto(&S)==0){

    APop(&S, &n);

    if(dato <= 1)
      F = F + 1;
    else {
      APush(&S, n-1);
      APush(&S, n-2);
    }

    StampaAStack(S);
    printf("   F=%d", F);
  }

  printf("\nIl valore di Fibonacci
          per %d è %d\n",
          atoi(argv[1]), F);
}
```

Fibonacci con stack esplicito

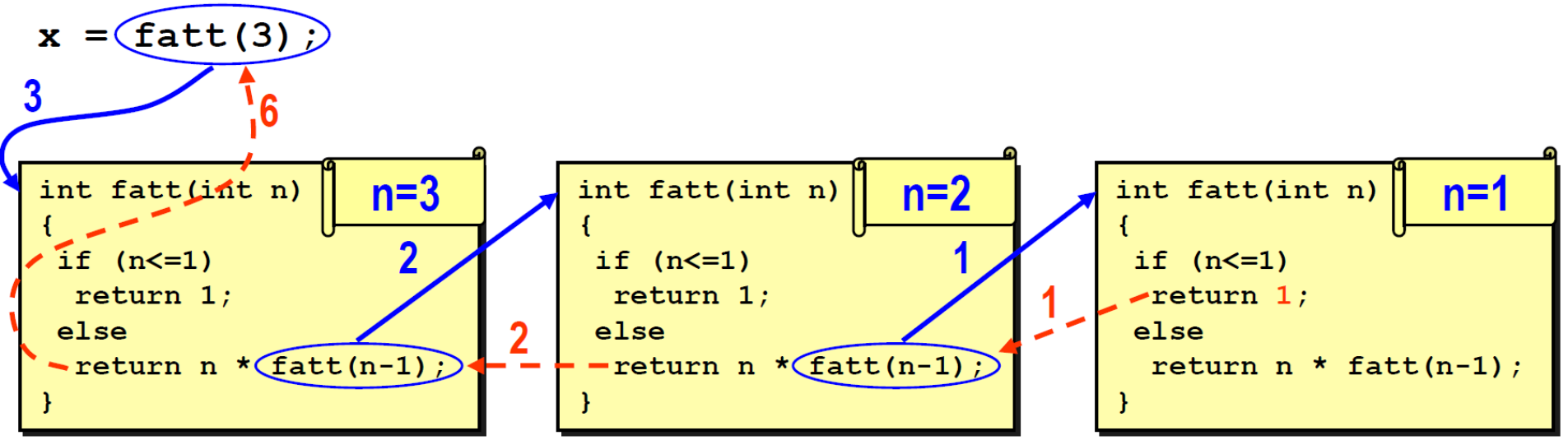
:\>FibonacciConStack.exe 4

```
[Stack]-> 2 -> 3 -|    F=0
[Stack]-> 0 -> 1 -> 3 -|    F=0
[Stack]-> 1 -> 3 -|    F=1
[Stack]-> 3 -|    F=2
[Stack]-> 1 -> 2 -|    F=2
[Stack]-> 2 -|    F=3
[Stack]-> 0 -> 1 -|    F=3
[Stack]-> 1 -|    F=4
[Stack]->    F=5
Il valore di Fibonacci per 4 Þ 5
```

Fattoriale – funzione ricorsiva

```
int fact(int n){
    if(n<=0)
        return 1;
    else
        return n*fact(n-1);
}
```

```
main(){
    int x;
    x = fact(3);
}
```



Fattoriale con stack

```c
int fact(int n)
{
   if(n<=0)
      return 1;
   else
      return n*fact(n-1);
}
```

```c
APush(&S, n);
F=1;
while(AStackVuoto(&S)==0){
   APop(&S, &n);
      if(n <= 0)
         F = 1*F;
      else {
         F = n*F;
         APush(&S, n-1);
}
```

Fattoriale con stack

```c
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

int main(int argc, char *argv[])
{
  int n;
  int F = 1;
  AStack S;

  CreaAStack(&S, 100);

  n = atoi(argv[1]);

  APush(&S, n);

  while(AStackVuoto(&S)==0){

    APop(&S, &n);

    if(n<=0)
      F = 1*F;
    else{
      F = n*F;
      APush(&S, n-1);
    }

    StampaAStack(S);
    printf("   F=%d", F);
  }

  printf("\nIl Fattoriale di %d è
          %d\n", atoi(argv[1]),
                                  F);
}
```

Fibonacci con stack esplicito

:\>FattorialeConStack.exe 6

```
[Stack]-> 5 -|    F=6
[Stack]-> 4 -|    F=30
[Stack]-> 3 -|    F=120
[Stack]-> 2 -|    F=360
[Stack]-> 1 -|    F=720
[Stack]-> 0 -|    F=720
[Stack]->     F=720
Il Fattoriale di 6 Þ 720
```

La torre di Hanoi

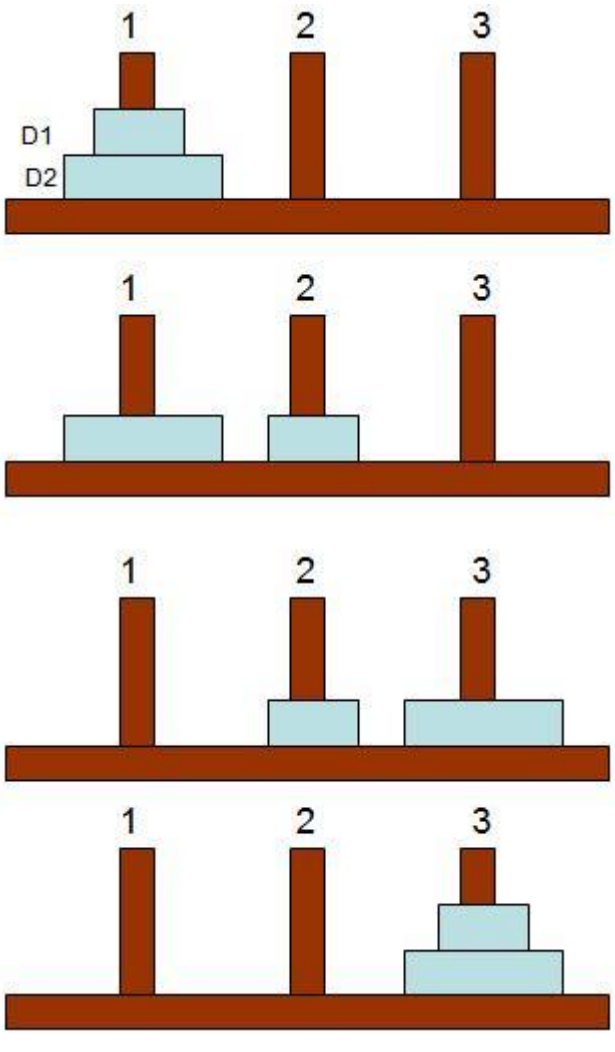
La **Torre di Hanoi** è un rompicapo matematico composto da tre paletti e un certo numero di dischi di grandezza decrescente, che possono essere infilati in uno qualsiasi dei paletti.

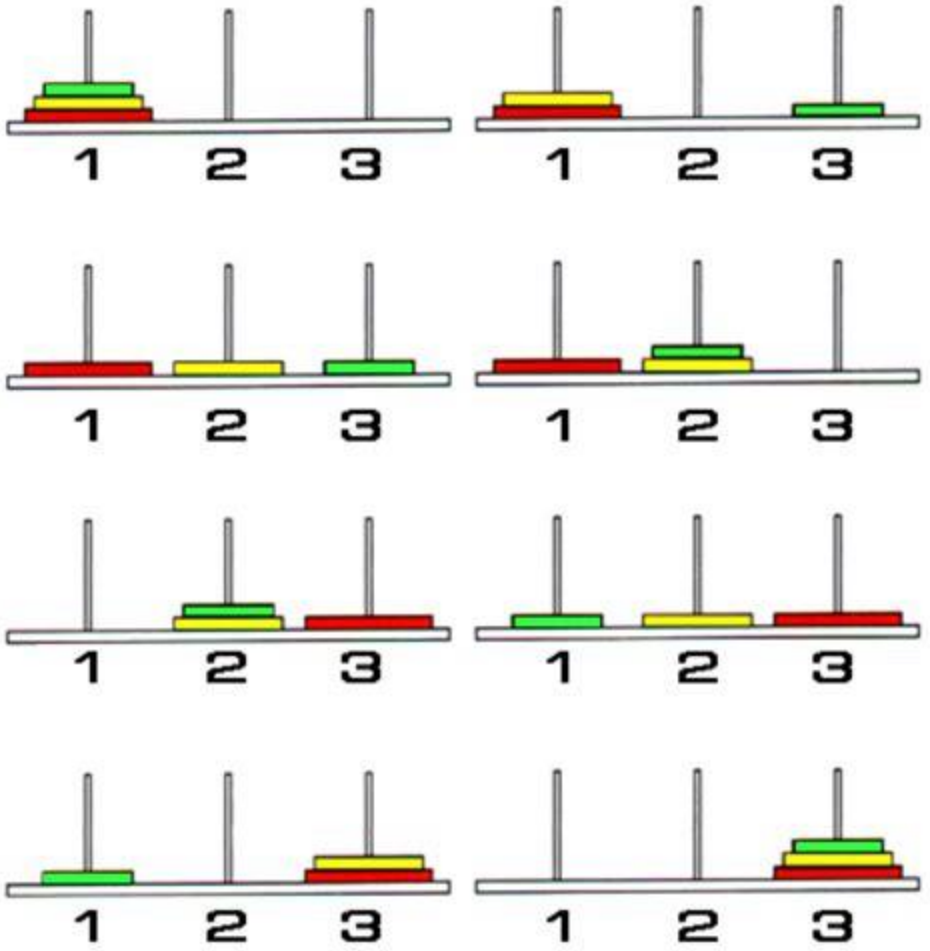
Il gioco inizia con tutti i dischi incolonnati su un paletto in ordine decrescente, in modo da formare un cono.

Lo scopo del gioco è portare tutti i dischi su un paletto diverso, rispettando due semplici regole:

1. si può spostare solo un disco alla volta
2. un disco può essere spostato solo su un altro disco più grande, mai su uno più piccolo.

La soluzione della torre di Hanoi





La soluzione della torre di Hanoi

Narra la leggenda che all'inizio dei tempi, Brahma portò nel grande tempio di Benares, sotto la cupola d'oro che si trova al centro del mondo, tre colonnine di diamante e sessantaquattro dischi d'oro, collocati su una di queste colonnine in ordine decrescente, dal più piccolo in alto, al più grande in basso

È la sacra Torre di Brahma che vede impegnati, giorno e notte, i sacerdoti del tempio nel trasferimento della torre di dischi dalla prima alla terza colonnina

Essi non devono contravvenire alle regole precise, imposte da Brahma stesso, che richiedono di spostare soltanto un disco alla volta e che non ci sia mai un disco sopra uno più piccolo

Quando i sacerdoti avranno completato il loro lavoro e tutti i dischi saranno riordinati sulla terza colonnina, la torre e il tempio crolleranno e sarà la fine del mondo

La soluzione della torre di Hanoi

La soluzione base del gioco della torre di Hanoi si formula in modo ricorsivo

Siano i paletti etichettati con A, B e C, e i dischi numerati da 1 (il più piccolo) a *n* (il più grande)

L'algoritmo si esprime come segue:

1) Sposta i primi *n*-1 dischi da A a B. (Questo lascia il disco *n* da solo sul paletto A)
2) Sposta il disco *n* da A a C
3) Sposta *n*-1 dischi da B a C

Per spostare *n* dischi si richiede di compiere un'operazione elementare (spostamento di un singolo disco) ed una complessa, ossia lo spostamento di *n*-1 dischi

Tuttavia anche questa operazione si risolve nello stesso modo, richiedendo come operazione complessa lo spostamento di *n*-2 dischi

Iterando questo ragionamento si riduce il processo complesso ad uno elementare, ovvero lo spostamento di *n*-(*n*-1)=1 disco

La soluzione della torre di Hanoi

La funzione che implementa la soluzione è definita in modo ricorsivo

Prende in input **4** parametri interi:

- **n**: numero di dischi da spostare
- **A**: il piolo da cui li spostiamo
- **B**: il piolo verso cui li spostiamo
- **C**: il piolo che adoperiamo come appoggio

Se il numero n di dischi da spostare è maggiore di 1, funzione chiama se stessa in modo ricorsivo due volte

1) Sposta n-1 dischi dal piolo **A** al piolo di appoggio **C** usando come appoggio il piolo **B**
2) Sposta n-1 dischi dal piolo **C** al piolo **B** usando come appoggio il piolo **A**

```
hanoi(n, A, B, C)
    Se n>1
     hanoi(n-1, A, C, B);

    Sposta un disco da A a B

    Se n>1
     hanoi(n-1, C, B, A);
```

La soluzione della torre di Hanoi

```
void Hanoi(int n, int A, int B, int C){

   static int pioli[3]={0, 0, 0};

   if((pioli[0]+pioli[1]+pioli[2])==0)
      pioli[0]=n;

   if(n>1)
      Hanoi(n-1, A, C, B);

   printf("\nSposta un disco dal piolo %d al piolo %d\n", A, B);
   pioli[A-1] -= 1;
   pioli[B-1]  += 1;
   printf("Pioli (%d, %d, %d)\n", pioli[0], pioli[1], pioli[2]);

   if(n>1)
      Hanoi(n-1, C, B, A);
}
```

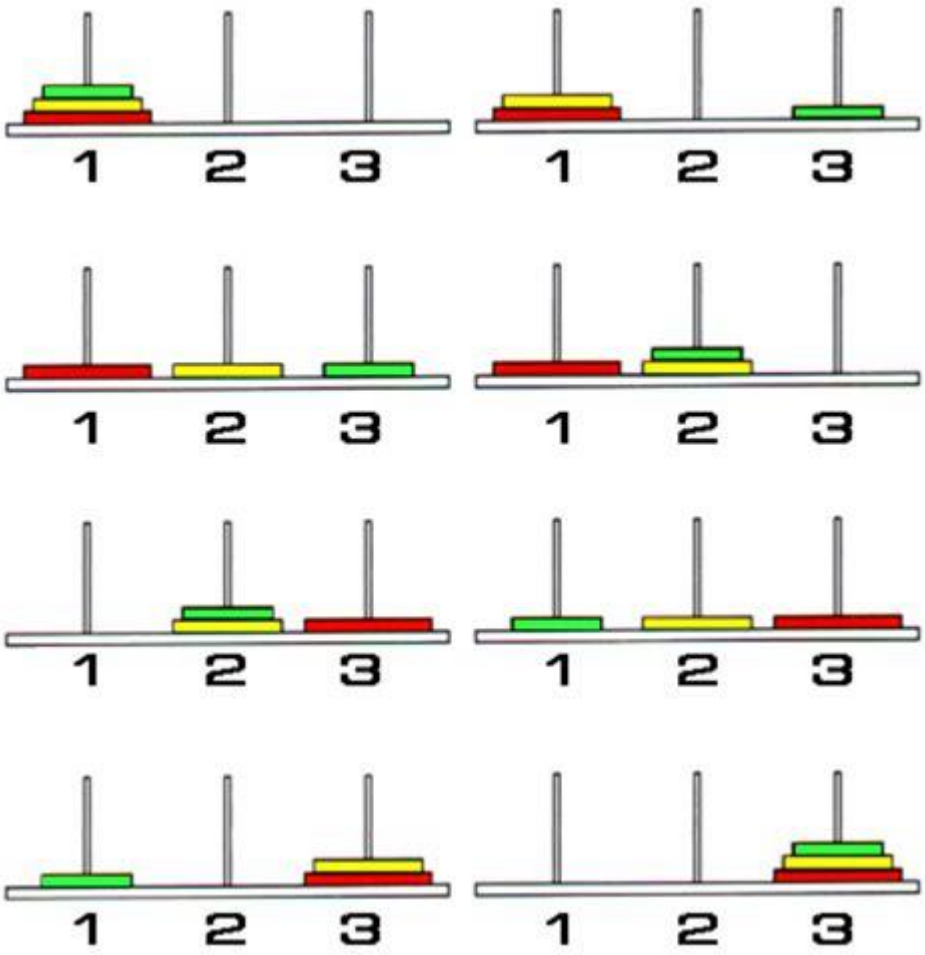
La soluzione della torre di Hanoi

```c
#include <stdio.h>
#include <stdlib.h>

void Hanoi(int n, int A, int B, int C);

int main(int argc, char *argv[])
{
   int ndischi=0;

   ndischi = atoi(argv[1]);

   Hanoi(ndischi, 1, 3, 2);
}
```

:\>Hanoi.exe 3

```
Sposta un disco dal piolo 1 al piolo 3
Pioli (2, 0, 1)

Sposta un disco dal piolo 1 al piolo 2
Pioli (1, 1, 1)

Sposta un disco dal piolo 3 al piolo 2
Pioli (1, 2, 0)

Sposta un disco dal piolo 1 al piolo 3
Pioli (0, 2, 1)

Sposta un disco dal piolo 2 al piolo 1
Pioli (1, 1, 1)

Sposta un disco dal piolo 2 al piolo 3
Pioli (1, 0, 2)

Sposta un disco dal piolo 1 al piolo 3
Pioli (0, 0, 3)
```

La soluzione della torre di Hanoi




```
Sposta un disco dal piolo 1 al piolo 3
Pioli (2, 0, 1)

Sposta un disco dal piolo 1 al piolo 2
Pioli (1, 1, 1)

Sposta un disco dal piolo 3 al piolo 2
Pioli (1, 2, 0)

Sposta un disco dal piolo 1 al piolo 3
Pioli (0, 2, 1)

Sposta un disco dal piolo 2 al piolo 1
Pioli (1, 1, 1)

Sposta un disco dal piolo 2 al piolo 3
Pioli (1, 0, 2)

Sposta un disco dal piolo 1 al piolo 3
Pioli (0, 0, 3)
```

La soluzione della torre di Hanoi

La proprietà matematica base è che il numero minimo di mosse necessarie per completare il gioco è $2^n - 1$, dove n è il numero di dischi.

Ad esempio avendo 3 dischi, il numero minimo di mosse è 7.

Di conseguenza, secondo la leggenda, i monaci di Hanoi dovrebbero effettuare almeno **18.446.744.073.709.551.615** mosse prima che il mondo finisca, essendo n = 64.

In altre parole, anche supponendo che i monaci facciano una mossa al secondo il mondo finirà tra **5.845.580.504** secoli.

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 degli elementi
- Es. grafo: n,m = numero dei vertici, numero archi

Tempo di elaborazione T(n)

- **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 = tempo di ogni input x **la probabilità che ci sia quell'input** (media pesata)

  È necessaria un'assunzione sulla distribuzione statistica degli input (spesso *distribuzione uniforme*)

- **Caso migliore:** (fittizio = prob. non si verificherà mai)

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

Caso peggiore

- Generalmente si cerca un limite superiore perché:
  - Fornisce una garanzia all'utente
    - L'algoritmo non potrà impiegare più di così
  - Per alcuni algoritmi si verifica molto spesso
    - Es. ricerca in un DB di informazione non presente
  - Il caso medio spesso è cattivo quasi quanto quello peggiore
    - Non sempre è evidente cosa costituisce un input medio

Esempio: T(n) di una funzione iterativa

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

Costo	Numero di volte
c_1	1
c_2	n
c_3	n-1
c_4	n-1

$T(n) = c_1 + n*c_2 + (n-1)*c_3 + (n-1)*c_4 = (c_2+c_3+c_4)*n + (c_1-c_3-c_4) =$
$= a*n + b$ **funzione lineare**

Indipendenza 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)

IDEA:

- Ignorare le costanti dipendenti dalla macchina
- Studiare il ***tasso di crescita*** di T(n) con $n \to \infty$

“Analisi asintotica”

Indipendenza dalla macchina

Nota:

A causa dei fattori costanti e dei termini di ordine inferiore che vengono ignorati

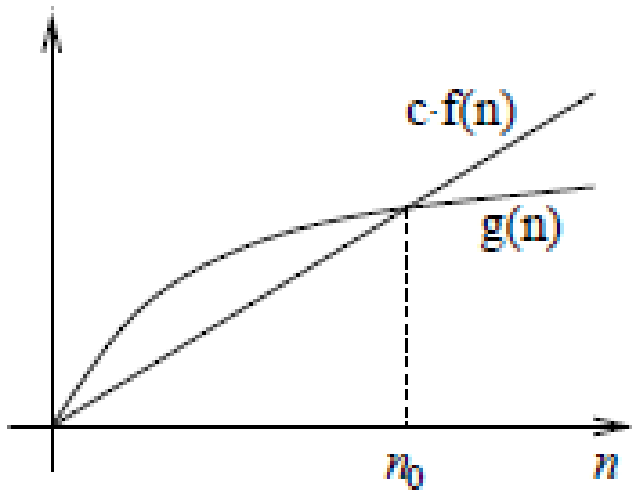
Se T1(n) > T2(n) [*asintoticamente*]:

- Per **input piccoli** algoritmo1 può richiedere meno tempo di algoritmo2, ma…
- Per **input sufficientemente grandi** algoritmo2 sarà eseguito più velocemente di algoritmo1

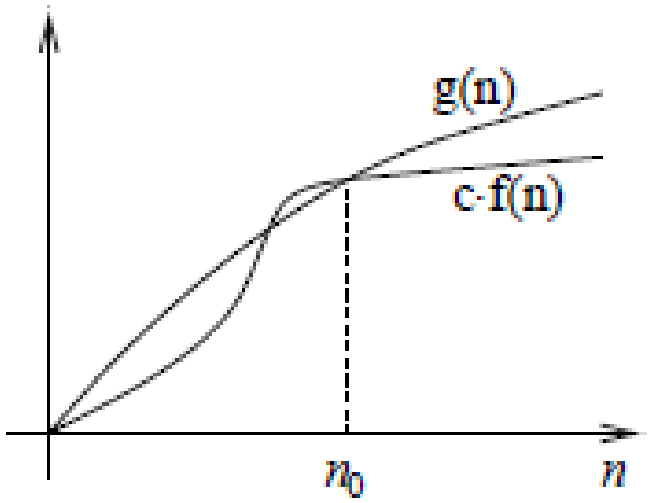
Definizioni: O-grande e Ω

Definizione1. T(n) è O(g(n)) se esistono due numeri positivi c ed N tali che $T(n) \leq c\, g(n)$ per qualsiasi $n \geq N$.

Si scrive T(n) = O(g(n))



- g(n) limita superiormente T(n)
- g(n) approssima asintoticamente T(n) dall'alto
- g(n) è una buona approssimazione superiore per T(n) quando n è molto grande



Definizione2. T(n) è Ω(g(n)) se esistono due numeri positivi c ed N tali che $T(n) \geq c\, g(n)$ per qualsiasi $n \geq N$.

Si scrive T(n) = Ω(g(n))

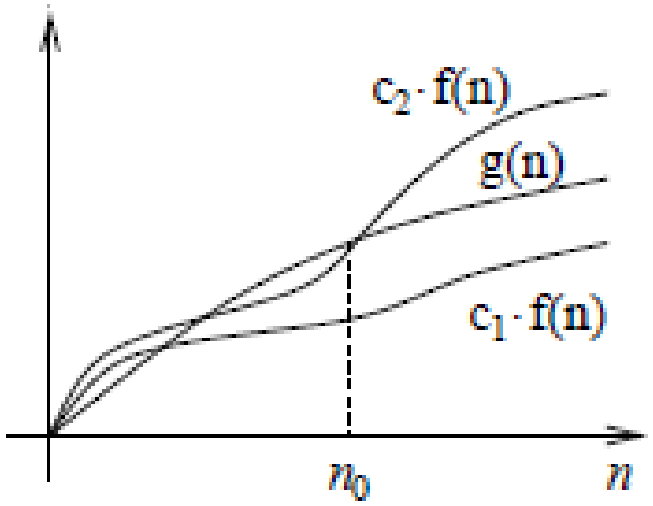
- g(n) limita inferiormente T(n)
- g(n) approssima asintoticamente T(n) dal basso
- g(n) è una buona approssimazione inferiore per T(n) quando n è molto grande

Definizioni: Θ

Definizione3. T(n) e' $\Theta(g(n))$ se esistono numeri positivi c_1, c_2, N tali che $c_1\, g(n) \leq T(n) \leq c_2\, g(n)$ per qualsiasi $n \geq N$

Si scrive $T(n) = \Theta(g(n))$

- g(n) limita T(n)
- g(n) approssima asintoticamente T(n)
- g(n) è una buona approssimazione per T(n) quando n è molto grande



Definizioni: O-grande

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 $g(n) \cdot h(n) = O(f(n) \cdot q(n))$ [anche per Ω e Θ]

Classi di complessità

- $T(n) = O(1)$: complessità costante (cioè $T(n)$ non dipende dalla dimensione n dei dati di ingresso).
- $T(n) = O(\log n)$: complessità logaritmica.
- $T(n) = O(n)$: complessità lineare.
- $T(n) = O(n \cdot \log n)$: complessità pseudolineare (così detta da $n \cdot \log n = O(n^{1+\epsilon})$ per ogni $\epsilon > 0$).
- $T(n) = O(n^2)$: complessità quadratica.
- $T(n) = O(n^3)$: complessità cubica.
- $T(n) = O(n^k), k > 0$: complessità polinomiale.
- $T(n) = O(\alpha^n), \alpha > 1$: complessità esponenziale.

Complessità asintotica e tempo

complessità \ dim. input	10	10^3	10^6
costante - O(1)	*1 μsec*	*1 μsec*	*1 μsec*
logaritmica - O(lg n)	3 *μsec*	*10 μsec*	*20 μsec*
lineare - O(n)	10 *μsec*	*1 msec*	1 sec
n lg n - O(n lg n)	33 *μsec*	*10 msec*	20 sec
quadratica - $O(n^2)$	100 *μsec*	1 sec	10^{12} sec
cubica - $O(n^3)$	1 msec	10^9 sec	10^{18} sec
esponenziale - $O(2^n)$	10 msec	10^{301} sec	10^{301030} sec

11.6 gg

31709 anni !!

16.7 min

Complessità di un problema

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