Tecniche di analisi degli algoritmi

Moreno Marzolla, Lorenzo Donatiello

Dipartimento di Infromatica, Università di Bologna

29 ottobre 2017

Modello di calcolo

Consideriamo un modello di calcolo costituito da una macchina a registri così composta:

- Esiste un dispositivo di input e un dispositivo di output;
- La macchina ha N locazioni di memoria, con indirizzo da 1 a N; ciascuna locazione può contenere un valore (intero, reale...);
- l'accesso in lettura o scrittura ad una qualsiasi locazione richiede tempo costante;
- La macchina dispone di un set di registri per mantenere i parametri necessari alle operazioni elementari e per il puntatore all'istruzione corrente;
- La macchina ha un programma composto da un insieme finito di istruzioni

Costo computazionale

Definizione

Indichiamo con $f(n)$ la quantità di risorse (tempo di esecuzione, oppure occupazione di memoria) richiesta da un algoritmo su input di dimensione n, operante su una macchina a registri.

Siamo interessati a studiare l'*ordine di grandezza* di $f(n)$ ignorando le costanti moltiplicative e termini di ordine inferiore.

Misura del costo computazionale

Utilizzare il tempo effettivo di esecuzione di un programma come costo computazionale presenta numerosi svantaggi:

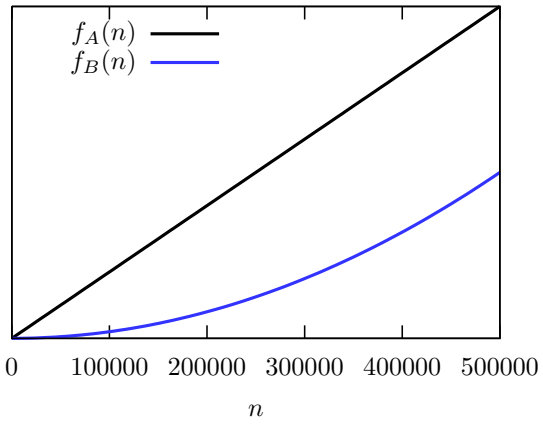
- Implementare un dato algoritmo può essere laborioso;
- Il tempo è legato alla specifica implementazione (linguaggio di programmazione usato, caratteristiche della macchina usata per effettuare le misure, ...);
- Potremmo essere interessati a stimare il costo computazionale usando input troppo grandi per le caratteristiche della macchina su cui effettuiamo le misure;
- Determinare l'ordine di grandezza a partire da misure empiriche non è sempre possibile;

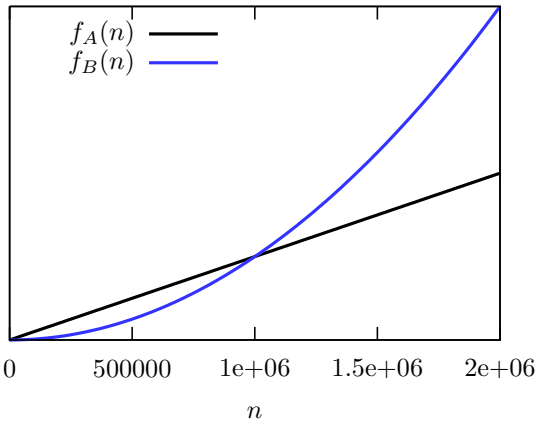
Costo computazionale
Esempio

Consideriamo due algoritmi A e B che risolvono lo stesso problema.

- Sia $f_A(n) = 10^3 n$ il costo computazionale di A;
- Sia $f_B(n) = 10^{-3} n^2$ il costo computazionale di B.

Quale dei due è preferibile?





La notazione asintotica $O(f(n))$

Definizione

Data una funzione costo $f(n)$, definiamo l'insieme $O(f(n))$ come l'insieme delle funzioni $g(n)$ per le quali esistono costanti $c > 0$ e $n_0 \geq 0$ per cui vale:

$$\forall n \geq n_0 \ : \ g(n) \leq cf(n)$$

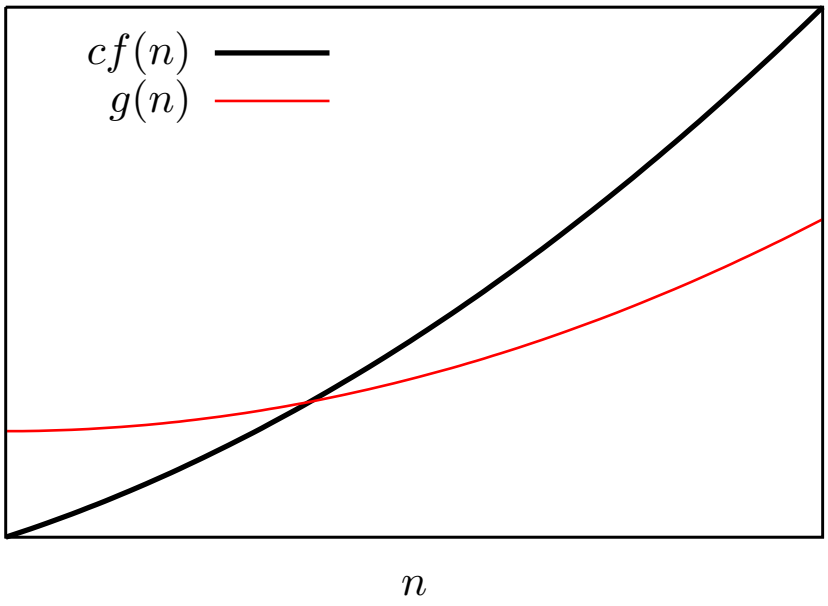
In maniera piú sintetica:

$$O(f(n)) = \{g(n) \ : \ \exists c > 0, n_0 \geq 0 \text{ tali che } \forall n \geq n_0 : g(n) \leq cf(n)\}$$

Nota: si utilizza la notazione (sebbene non formalmente corretta) $g(n) = O(f(n))$ per indicare $g(n) \in O(f(n))$.

Rappresentazione grafica

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



Esempio

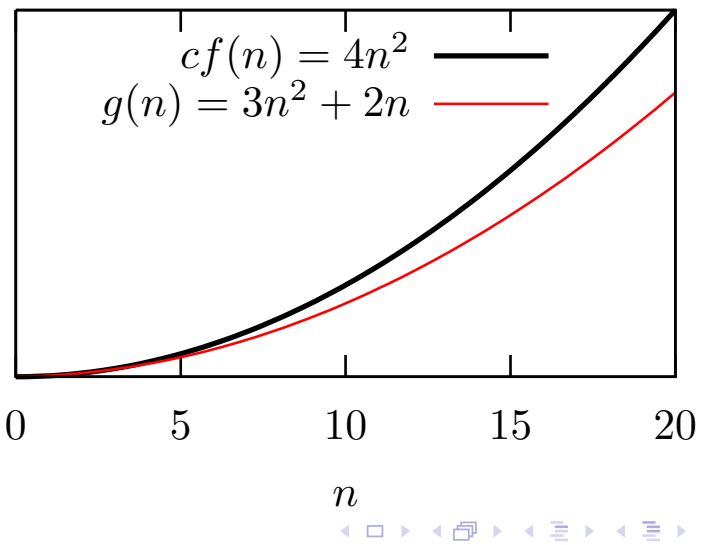
Sia $g(n) = 3n^2 + 2n$ e $f(n) = n^2$. Dimostriamo che $g(n) = O(f(n))$.

Dobbiamo trovare due costanti $c > 0, n_0 \geq 0$ tali che $g(n) \leq cf(n)$ per ogni $n \geq n_0$, ossia:

$$3n^2 + 2n \leq cn^2 \tag{1}$$

$$c \geq \frac{3n^2 + 2n}{n^2} = 3 + \frac{2}{n}$$

se ad esempio scegliamo $n_0 = 10$ e $c = 4$, si ha che la relazione (1) è verificata.



La notazione asintotica $\Omega(f(n))$

Definizione

Data una funzione costo $f(n)$, definiamo l'insieme $\Omega(f(n))$ come l'insieme delle funzioni $g(n)$ per le quali esistono costanti $c > 0$ e $n_0 \geq 0$ per cui vale:

$$\forall n \geq n_0 \ : \ g(n) \geq cf(n)$$

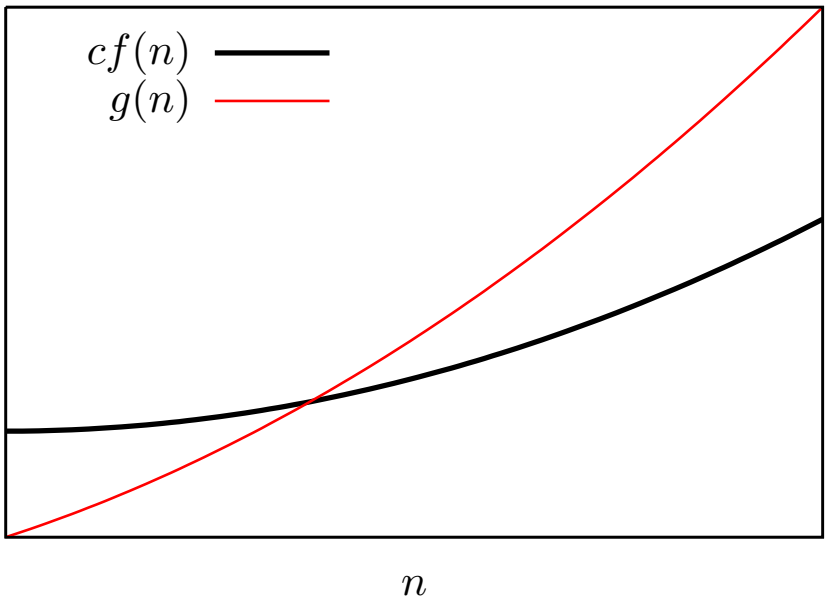
In maniera piú sintetica:

$$\Omega(f(n)) = \{g(n) \ : \ \exists c > 0, n_0 \geq 0 \text{ tali che } \forall n \geq n_0 : g(n) \geq cf(n)\}$$

Nota: si utilizza la notazione $g(n) = \Omega(f(n))$ per indicare $g(n) \in \Omega(f(n))$.

Rappresentazione grafica

$$g(n) = \Omega(f(n))$$



Esempio

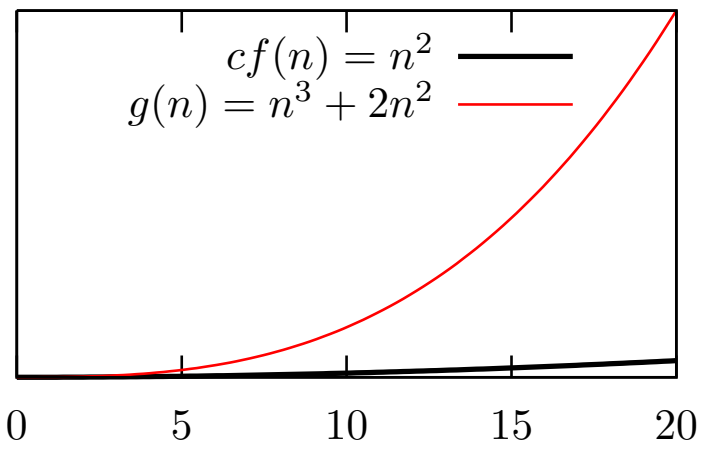
Sia $g(n) = n^3 + 2n^2$ e $f(n) = n^2$, e dimostriamo che $g(n) = \Omega(f(n))$.

Dobbiamo trovare due costanti $c > 0$, $n_0 \geq 0$ tali che per ogni $n \geq n_0$ sia $g(n) \geq cf(n)$, ossia:

$$n^3 + 2n^2 \geq cn^2 \tag{2}$$

$$c \leq \frac{n^3 + 2n^2}{n^2} = n + 2$$

se ad esempio scegliamo $n_0 = 0$ e $c = 1$, si ha che la relazione (2) è verificata.



La notazione asintotica $\Theta(f(n))$

Definizione

Data una funzione costo $f(n)$, definiamo l'insieme $\Theta(f(n))$ come l'insieme delle funzioni $g(n)$ per le quali esistono costanti $c_1 > 0$, $c_2 > 0$ e $n_0 \geq 0$ per cui vale:

$$\forall n \geq n_0 \ : \ c_1 f(n) \leq g(n) \leq c_2 f(n)$$

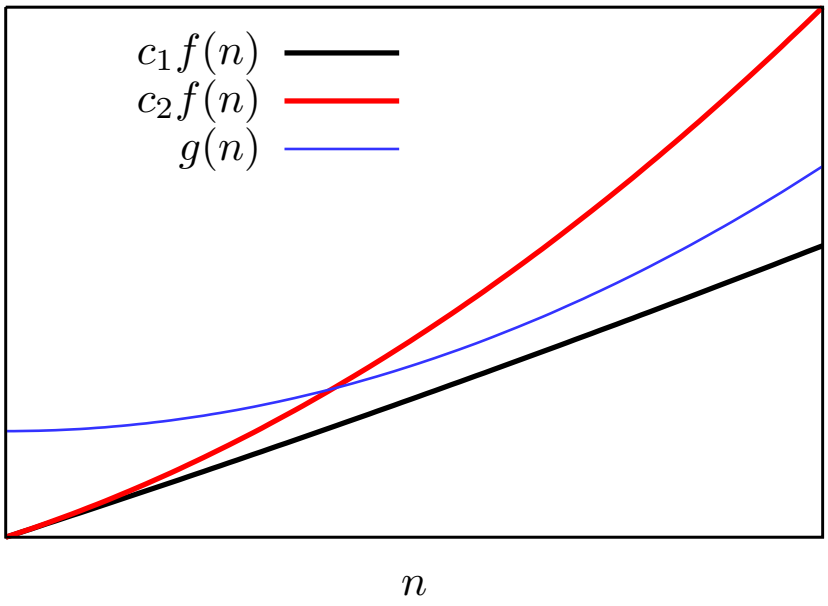
In maniera piú sintetica:

$$\Theta(f(n)) = \{g(n) \ : \exists c_1 > 0, c_2 > 0, n_0 \geq 0 \text{ tali che } \forall n \geq n_0 : c_1 f(n) \leq g(n) \leq c_2 f(n)\}$$

Nota: si utilizza la notazione $g(n) = \Theta(f(n))$ per indicare $g(n) \in \Theta(f(n))$.

Rappresentazione grafica

$$g(n) = \Theta(f(n))$$



Spiegazione intuitiva

- Se $g(n) = O(f(n))$ significa che l'ordine di grandezza di $g(n)$ è "minore o uguale" a quello di $f(n)$;
- Se $g(n) = \Theta(f(n))$ significa che $g(n)$ e $f(n)$ hanno lo stesso ordine di grandezza;
- Se $g(n) = \Omega(f(n))$ significa che l'ordine di grandezza di $g(n)$ è "maggiore o uguale" a quello di $f(n)$.

Alcune proprietà delle notazioni asintotica

Simmetria

$g(n) = \Theta(f(n))$ se e solo se $f(n) = \Theta(g(n))$

Simmetria Trasposta

$g(n) = O(f(n))$ se e solo se $f(n) = \Omega(g(n))$

Transitività

Se $g(n) = O(f(n))$ e $f(n) = O(h(n))$, allora $g(n) = O(h(n))$.
Lo stesso vale per Ω e Θ.

Ordini di grandezza

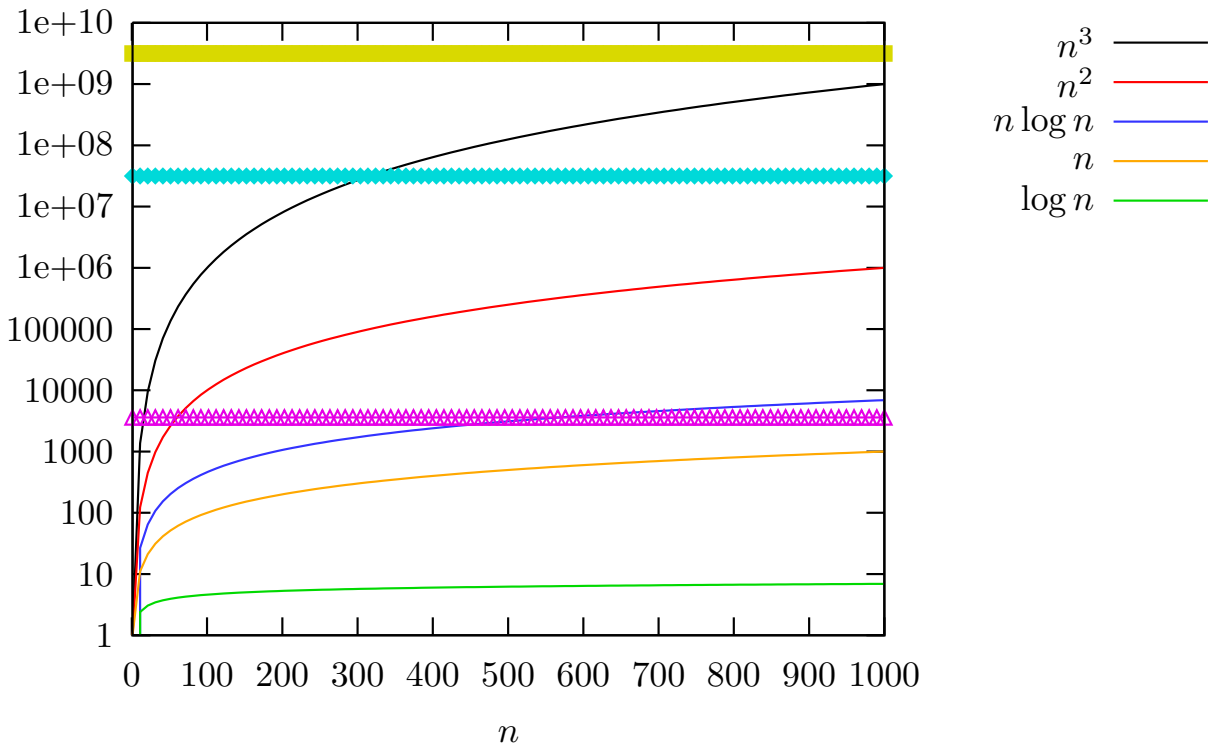
In ordine di costo crescente:

	Ordine	**Esempio**
$O(1)$	costante	Determinare se un numero è pari
$O(\log n)$	logaritmico	Ricerca di un elemento in un array ordinato
$O(n)$	lineare	Ricerca di un elemento in un array disordinato
$O(n \log n)$	pseudolineare	Ordinamento mediante Merge Sort
$O(n^2)$	quadratico	Ordinamento mediante Bubble Sort
$O(n^3)$	cubico	Prodotto di due matrixi $n \times n$ con l'algoritmo "intuitivo"
$O(c^n)$	esponenziale, base $c > 1$	
$O(n!)$	fattoriale	Calcolare il determinante di una matrice mediante espansione dei minori
$O(n^n)$	esponenziale, base n	

In generale:

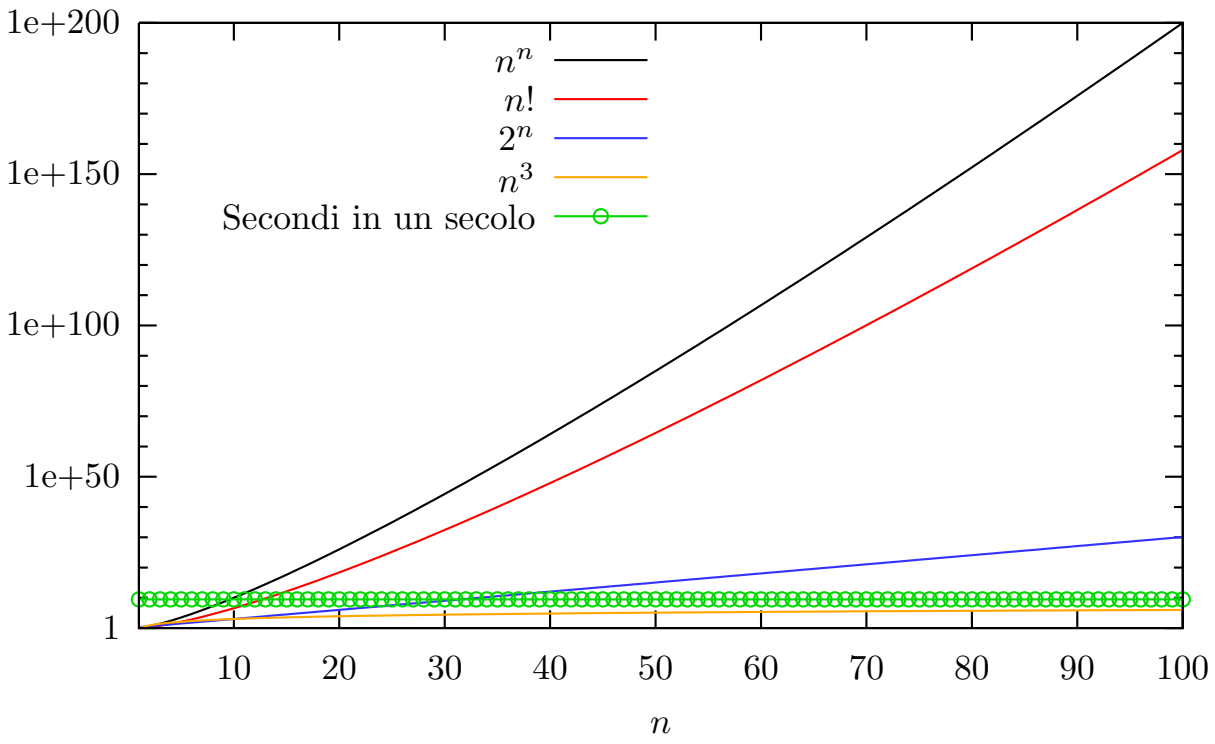
- $O(n^k)$ con $k > 0$ è ordine polinomiale
- $O(c^n)$ con $c > 1$ è ordine esponenziale

Confronto grafico tra gli ordini di grandezza



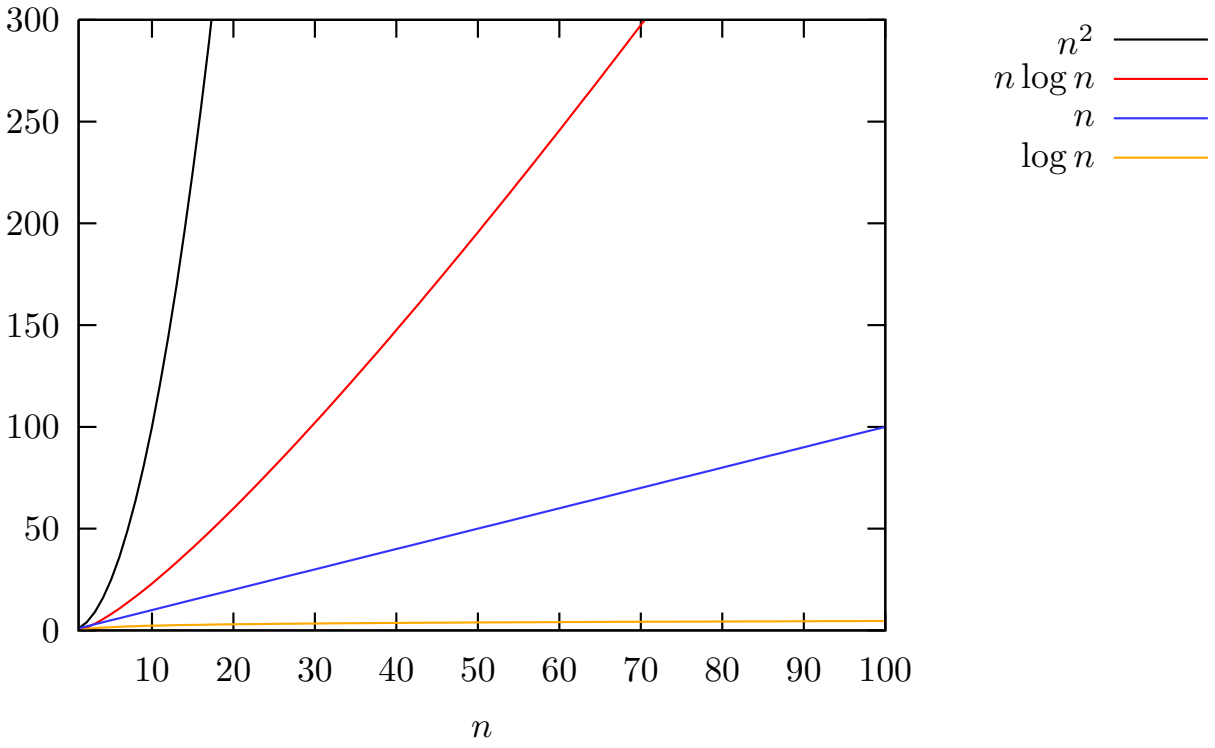
Nota: scala *y* logaritmica; le linee orizzontali segnano il numero di secondi in un'ora, in un anno e in un secolo (rispettivamente, dal basso verso l'alto)

Confronto grafico tra gli ordini di grandezza



Nota: scala *y* logaritmica!

Confronto grafico tra gli ordini di grandezza



Vero o falso?

$6n^2 = \Omega(n^3)$?

Applicando la definizione, dobbiamo dimostrare se

$$\exists c > 0, n_0 \geq 0 \ : \ \forall n \geq n_0 \quad 6n^2 \geq cn^3$$

Cioè $c \leq 6/n$.

Fissato c è sempre possibile scegliere un valore di n sufficientemente grande tale che $6/n < c$, per cui l'affermazione è falsa. □

Vero o falso?

$10n^3 + 2n^2 + 7 = O(n^3)$?

Applicando la definizione, dobbiamo dimostrare se

$$\exists c > 0, n_0 \geq 0 \ : \ \forall n \geq n_0 \quad 10n^3 + 2n^2 + 7 \leq cn^3$$

Possiamo scrivere:

$$\begin{aligned} 10n^3 + 2n^2 + 7 &\leq 10n^3 + 2n^3 + 7n^3 \qquad (\text{se } n \geq 1) \\ &= 19n^3 \end{aligned}$$

Quindi la disuguaglianza è verificata ponendo $n_0 = 1$ e $c = 19$. □

Risultato

In generale possiamo provare che

$a_k n^k + a_{k-1} n^{k-1} + a_{k-2} n^{k-2} \cdots + a_1 n + a_0 = O(n^k)$

Possiamo scrivere:
$n^3 + 100n + 200 = O(n^3)$
$20n^3 + n^5 + 100nO(n^5)$
$3n^7 + n^5 = O(n^7)$

Relazioni

In generale possiamo provare che per

$\forall(a > 0, b > 0, k > 0), (logn^b)^a = O(n^k)$

$\forall k > 0, a > 1, n^k = O(a^n)$

$3log(n^2)^3 = O(n)$

$n^{10} = O(2^n)$

Domande

- Dimostrare che $\log_2 n = O(n)$;
- Cosa cambia se il logaritmo di cui sopra non è in base 2?
- Dimostrare che $n \log n = O(n^2)$;
- Dimostrare che, per ogni $\alpha > 0$, $\log n = O(n^\alpha)$ (suggerimento: da quanto visto sopra si può affermare che $\log n^\alpha = O(n^\alpha)$, quindi...)
- Dove collochereste $O(\sqrt{n})$ nella tabella degli ordini di grandezza? Perché?

Costo di esecuzione

Definizione

Un algoritmo $\mathcal{A}$ ha costo di esecuzione $O(f(n))$ su istanze di ingresso di dimensione n rispetto ad una certa risorsa di calcolo se la quantitá $r(n)$ di risorsa sufficiente per eseguire $\mathcal{A}$ su una qualunque istanza di dimensione n verifica la relazione $r(n) = O(f(n))$.

Nota Risorsa di calcolo per noi significa tempo di esecuzione oppure occupazione di memoria.

Complessità dei problemi

Definizione

Un problema $\mathcal{P}$ ha complessità $O(f(n))$ rispetto ad una data risorsa di calcolo se esiste un algoritmo che risolve $\mathcal{P}$ il cui costo di esecuzione rispetto a quella risorsa è $O(f(n))$.

Alcune regole utili

Somma

Se $g_1(n) = O(f_1(n))$ e $g_2(n) = O(f_2(n))$, allora
$g_1(n) + g_2(n) = O(f_1(n) + f_2(n))$

Prodotto

Se $g_1(n) = O(f_1(n))$ e $g_2(n) = O(f_2(n))$, allora
$g_1(n) \cdot g_2(n) = O(f_1(n) \cdot f_2(n))$

Eliminazione costanti

Se $g(n) = O(f(n))$, allora $a \cdot g(n) = O(f(n))$ per ogni costante $a > 0$

Osservazione

Utilizzando gli ordini di grandezza, ogni operazione elementare ha costo $O(1)$; un contributo diverso viene dalle istruzioni condizionali e iterative.

```
if ( F_test ) {
  F_true
} else {
  F_false
}
```

Supponendo:

- $\text{F\_test} = O(f(n))$
- $\text{F\_true} = O(g(n))$
- $\text{F\_false} = O(h(n))$

Allora il costo di esecuzione del blocco if-then-else è

$$O(\max\{f(n), g(n), h(n)\})$$

Analisi nel caso ottimo, pessimo e medio

Sia $\mathcal{I}_n$ l'insieme di tutte le possibili *istanze di input* di lunghezza n. Sia $T(I)$ il tempo di esecuzione dell'algoritmo sull'istanza $I \in \mathcal{I}_n$.

- Il costo nel caso pessimo (*worst case*) è definito come

$$T_{\text{worst}}(n) = \max_{I \in \mathcal{I}_n} T(I)$$

- Il costo nel caso ottimo (*best case*) è definito come

$$T_{\text{best}}(n) = \min_{I \in \mathcal{I}_n} T(I)$$

- Il costo nel caso medio (*average case*) è definita come

$$T_{\text{avg}}(n) = \sum_{I \in \mathcal{I}_n} T(I)P(I)$$

  dove $P(I)$ è la probabilità che l'istanza I si presenti.

la complessità $T(n)$ di un algoritmo A è $O(n^2)$, vuol dire che A non richiede mai tempo superiore a cn^2, per produrre il suo output in corrispondenza ad un qualsivoglia input di ampiezza n, per valore di c opportuno e n sufficientemente grande.

NON si intende che A impiega tempo cn^2 per ogni input di ampiezza n. NON esistono input per cui l'algoritmo richiede tempo $> cn^2$.

la complessità $T(n)$ di un algoritmo A è $\Omega(n^2)$, vuol dire che A richiede almeno tempo cn^2, per produrre il suo output nel caso peggiore, per valore di c opportuno e n sufficientemente grande.

ESISTE almeno un input di ampiezza n (n suff. grande) su cui A richiede tempo cn^2

Analisi di algoritmi non ricorsivi

```
if n pari the return 0
  else
  for i = 1 to n do
      x= x+10
      return x
```

la complessità dell'algoritmo è $\Theta(n)$.

Analisi di algoritmi non ricorsivi

```
for i = 1 to 2n do
        x= x+10
```

la complessità dell'algoritmo è $\Theta(n)$.

```
for i = 1 to 2n do
        for j = 1 to n do
                x= x+10
```

la complessità dell'algoritmo è $\Theta(n^2)$.

```
for i = 1 to 2n do
        for j = 1 to n do
                for k = 1 to j
                x= x+10
```

la complessità dell'algoritmo è $\Theta(n^3)$.

Analisi di algoritmi non ricorsivi

```
i=n
        while i >= 1 do
                {x= x+1, i= i/2}
```

la complessità dell'algoritmo è $\Theta(\log n)$.

```
i=n
        while i >= 1 do
                for j= 1 to n {x= x+1}
                i= i/2
```

la complessità dell'algoritmo è $\Theta(n \log n)$.

Analisi di algoritmi non ricorsivi

algoritmo Esercizio(A[1..n] di float)

1. for i = 1 to n do
2. B[i] = A[i]
3. for i = 1 to n do {
4. j=n
5. while j>1 do{
6. B[i] = B[i] + A[i], j= j-1}
7. for i = 1 to n do
8. t = t + B[i]

Analisi:
Le linee 1. e 2. caratterizzate da tempo $\Theta(n)$.
Le linee da 3. a 6. caratterizzate da tempo $\Theta(n^2)$.

Le linee 7. e 8. caratterizzate da tempo $\Theta(n)$.
In totale, la complessità dell'algoritmo è$T(n) = \Theta(n) + \Theta(n^2) + \Theta(n) = \rightarrow\!\!\shortmid(n^2)$

Analisi di algoritmi non ricorsivi

algoritmo Esercizio(A[1..n] di float)

1. for i = 1 to n do
2. B[i] = A[i]
3. for i = 1 to n do {
4. j=n-i+1
5. while j>1 do{
6. B[i] = B[i] + A[i], j= j-1}
7. for i = 1 to n do
8. t = t + B[i]

Analisi:
Le linee 1. e 2. caratterizzate da tempo $\Theta(n)$.
Le linee da 3. a 6. caratterizzate da tempo ……????? .

Le linee 7. e 8. caratterizzate da tempo $\Theta(n)$.
In totale, la complessità dell'algoritmo è $T(n) = \Theta(n) + ?????+ \Theta(n) = ??????$

Analisi di algoritmi non ricorsivi

```
Procedura (A[1..n] float, k integer)
        s=0
        for i = 1 to k do
        s= s+ A[i]
        s= s/k

Main (AA[1..m])
        i=m
        while i >= 1 do Procedura (A[1..m],i)
        i = i-2
```

Costo computazionale ?

Analisi di algoritmi non ricorsivi

Ricerca il valore minimo contenuto in un array non vuoto

```
// Restituisce la posizione dell'elemento minimo in
algoritmo Minimo( A[1..n] di float ) -> int
  int m:=1; // Posizione dell'elemento minimo
  for i:=2 to n do
    if ( A[i]<A[m] ) then
      m = i;
    endif
  endfor
  return m;
}
```

Analisi

- Sia n la lunghezza del vettore v.
- Il corpo del ciclo viene eseguito $n - 1$ volte;
- Ogni iterazione ha costo $O(1)$
- Il costo di esecuzione della funzione `Minimo` rispetto al tempo è quindi $O(n)$ (o meglio, $\Theta(n)$: perché?).

Ricerca sequenziale
Caso ottimo e pessimo

```
Restituisce la posizione della prima occorrenza del
valore ``val'' nell'array A[1..n]. Ritorna -1 se
 il valore  non e' presente
Trova( array A[1..n] di int, int val ) -> int
   for i:=1 to n do
     if ( A[i]==val ) then
       return i;
     endif
   endfor
   return -1;
```

- Nel caso ottimo l'elemento è all'inizio della lista, e viene trovato alla prima iterazione. Quindi $T_{\text{best}}(n) = O(1)$
- Nel caso pessimo l'elemento non è presente nella lista (oppure è presente nell'ultima posizione), quindi si itera su tutti gli elementi. Quindi $T_{\text{worst}}(n) = \Theta(n)$
- E nel caso medio?

Ricerca sequenziale
Analisi del caso medio

Non avendo informazioni sulla probabilità con cui si presentano i valori nella lista, dobbiamo fare delle ipotesi semplificative.

Assumiamo che, dato un vettore di n elementi, la probabilità P_i che l'elemento cercato si trovi in posizione i ($i = 1, 2, \ldots n$) sia $P_i = 1/n$, per ogni i (assumiamo che l'elemento sia sempre presente).

Il tempo $T(i)$ necessario per individuare l'elemento nella posizione i-esima è $T(i) = i$.

Quindi possiamo concludere che:

$$T_{\text{avg}}(n) = \sum_{i=1}^{n} P_i T(i) = \frac{1}{n} \sum_{i=1}^{n} i = \frac{1}{n} \frac{n(n-1)}{2} = \Theta(n)$$

Esempio
Un algoritmo iterativo di ordinamento

```
 selectionSort(ITEM[] A, integer n)

    for integer i <--1 to n do
     integer j<-- min(A,i,n)
    A[i]<--> A[j]

    integer min(ITEM A, integer k, integer n)
     integer min <-- k
for integer h <--k+1 to n do
      if ( A[h] < A[min] ) then min <-- h
    return min
```

Analisi dell'algoritmo di ordinamento

- La chiamata min(A,i,n) individua l'elemento minimo nell'array $A[i], A[i+1], \ldots A[n]$. Il tempo richiesto è proporzionale a $n - i$, $i = 1, 2, \ldots n$ (perché?);
- L'operazione di scambio ha costo $O(1)$ in termini di tempo di esecuzione;
- Il corpo del ciclo **for** viene eseguito n volte.

Il costo di esecuzione rispetto al tempo dell'intera funzione `Selectiosort` è:

$$\sum_{i=0}^{n-1}(n-i) = n^2 - \sum_{i=0}^{n-1} i = n^2 - \frac{n(n-1)}{2} = \frac{n^2+n}{2}$$

che è $\Theta(n^2)$.

Analisi di algoritmi ricorsivi
derivazione delle relazioni di ricorrenza

Per derivare le relazioni di ricorrenza che descrivono il tempo di esecuzione $T(n)$ di un algoritmo occorre:
1) Determinare la dimensione dell'input n;
2) Determinare quale valore $n_0$di n è usato per la base della ricorsione (generalmente, ma non sempre, $n_0 = 1$).
3) Determinare il valore di $T(n_0)$, in genere avremo che $T(n_0) = c$ per qualche costante c.
$T(n)$ sarà generalmente uguale ad una somma:
$T(n) = T(m_1) + ... + T(m_a)$ (per le chiamate ricorsive),
più la somma di eventuale altra computazione $g(n)$. Spesso le a chiamate ricorsive saranno effettuate tutte su sottoproblemi di dimensione uguale a $f(n)$, dando un termine $aT(f(n))$ nella relazione di ricorrenza.

Analisi di algoritmi ricorsivi
Tipica equazione di Ricorrenza

$$T(n) = \begin{cases} c & \text{se } n = n_0 \\ aT(f(n) + g(n) & \text{altrimenti} \end{cases}$$

n_0 è la base della ricorsione, c tempo di esecuzione nel caso base;
a numero di volte che le chiamate ricorsive vengono effettuate;
$f(n)$ dimensione dei problemi risolti nelle chiamate ricorsive;
$g(n)$ costo computazionale non incluso nelle chiamate ricorsive.

Analisi di algoritmi ricorsivi
Esempio

```
procedure uno (int, n)
if n = 1 then x = 1
else
    uno (n-1);
    uno (n-2);
    for i= 1 to n do x= x+i
 end.
```

$$T(n) = \begin{cases} c & \text{se } n = 1 \\ T(n-1) + T(n-2) + cn & \text{altrimenti} \end{cases}$$

Analisi di algoritmi ricorsivi
Esempio

```
procedure due (int, n)
if n = 1 or n= 2 then x = 1
else
    due (n-1);
       for i= 1 to n do {x= x+25};
    due(n-1);
 end.
```

$$T(n) = \begin{cases} c & \text{se } n \leq 2 \\ 2T(n-1) + cn & \text{altrimenti} \end{cases}$$

Analisi di algoritmi ricorsivi
Esempio

```
procedure tre (int, n)
if n = 1 then x = 1
else if n = 2 the x= 2
else
    for i= 1 to n do
        {tre (n-1); x = x+21}
end.
```

$$T(n) = \begin{cases} c & \text{se } n \leq 2 \\ nT(n-1) + cn & \text{altrimenti} \end{cases}$$

Analisi di algoritmi ricorsivi
Esempio

```
procedure quattro (int, n)
if n = 1 then x = 1
else
    for i= 1 to n do
       {quattro(i); x= x+21;}
end.
```

$$T(n) = \begin{cases} c & \text{se } n = 1 \\ \sum_{i=1}^{n} T(i) + cn & \text{altrimenti} \end{cases}$$

Analisi di algoritmi ricorsivi
Ricerca di un elemento in un array ordinato

```
ITEM binarySearch(ITEM[] A,ITEM v,integer i,integer j)

  if i>j then
     return 0
  else
     integer m <-- (i+j)/2
     if A[m] = v then
         return m
      else if A[m] <v then
         return binarySearch(A, v, m+1, j)
       else return binarySearch(A, v, i,  m-1)
```

Analisi dell'algoritmo di ricerca binaria

Sia $T(n)$ il tempo di esecuzione della funzione `binarySearch` su un vettore di $n = j - i + 1$ elementi.

In generale $T(n)$ dipende non solo dal numero di elementi su cui fare la ricerca, ma anche dalla posizione dell'elemento cercato (oppure dal fatto che l'elemento non sia presente).

- Nell'ipotesi più favorevole (caso ottimo) l'elemento cercato è proprio quello che occupa posizione centrale; in tal caso $T(n) = O(1)$.
- Nel caso meno favorevole (caso pessimo) l'elemento cercato non esiste. Quanto vale $T(n)$ in tale situazione?

Analisi dell'algoritmo di ricerca binaria

Metodo dell'iterazione

Possiamo definire $T(n)$ per ricorrenza, come segue.

$$T(n) = \begin{cases} c_1 & \text{se } n = 0 \\ T(\lfloor n/2 \rfloor) + c_2 & \text{se } n > 0 \end{cases}$$

Il metodo dell'iterazione consiste nello sviluppare l'equazione di ricorrenza, per intuirne la soluzione:

$$T(n) = T(n/2)+c_2 = T(n/4)+2c_2 = T(n/8)+3c_2 = \ldots = T(n/2^i)+i\times c_2$$

Supponendo che n sia una potenza di 2, ci fermiamo quando $n/2^i = 1$, ossia $i = \log n$. Alla fine abbiamo

$$T(n) = c_1 + c_2 \log n = O(\log n)$$

Teorema fondamentale della ricorrenza
Master Theorem

Teorema

La relazione di ricorrenza:

$$T(n) = \begin{cases} aT(n/b) + f(n) & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases} \tag{3}$$

ha soluzione:

1. $T(n) = \Theta(n^{\log_b a})$ *se* $f(n) = O(n^{\log_b a - \epsilon})$ *per* $\epsilon > 0$;
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* $f(n) = \Omega(n^{\log_b a + \epsilon})$ *per* $\epsilon > 0$ *e* $af(n/b) \leq cf(n)$ *per* $c < 1$ *e* n *sufficientemente grande.*

Applicare il Master Theorem:

a) CALCOLARE a, b, f(n)

b) CALCOLARE n^{log} b^a

c) CONFRONTARE Asintoticamente f(n) con n^{log} b^a

d) Applicare opportunamente il MT

$T(n) = 2T(n/2) + n$ Caso?

$T(n) = 9T(n/3) + n$ Caso?

$T(n) = 3T(n/4) + n\log(n)$ Caso ?

Esempio
Applicazione del teorema fondamentale

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

1. Nel caso della ricerca binaria, abbiamo $T(n) = T(n/2) + O(1)$. Da cui $a = 1$, $b = 2$, $f(n) = O(1)$; siamo nel secondo caso del teorema, da cui $T(n) = \Theta(\log n)$.
2. Consideriamo $T(n) = 9T(n/3) + n$; in questo caso $a = 9$, $b = 3$ e $f(n) = O(n)$. Siamo nel primo caso, $f(n) = O(n^{\log_b a - \epsilon})$ con $\epsilon = 1$, da cui $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$.

Teorema delle ricorrenze lineari di ordine costante

Teorema

Siano

$$a_1, a_2 \ldots, a_h$$

costanti intere non negative, c, d e β costanti reali tale che: $c > 0$, $d > 0$ *e* $\beta \geq 0$*:*

$$T(n) = \begin{cases} d & \text{se } n \leq m \leq h \\ \sum_{i=1}^{h} a_i T(n-i) + c(n)^{\beta} & \text{se } n > m \end{cases} \tag{4}$$

posto: $a = \sum_{i=1}^{h} a_i$

1. $T(n) = O(n^{\beta+1})$ *se* $a = 1$*;*
2. $T(n) = O(n^{\beta} a^n)$ *se* $a \geq 2$*;*

Teorema delle ricorrenze lineari con partizioni bilanciate

Teorema

Siano: $a \geq 1$ *e* $b \geq 2$ *interi;* c, d *e* β *costanti reali tale che:* $c > 0$, $d \geq 0$ *e* $\beta \geq 0$*:*

$$T(n) = \begin{cases} aT(n/b) + c(n^{\beta}) & \text{se } n > 1 \\ d & \text{se } n = 1 \end{cases} \tag{5}$$

posto: $\alpha = loga/logb$

1. $T(n) = O(n^{\alpha})$ *se* $\alpha > \beta$;
2. $T(n) = O(n^{\alpha} \log n)$ *se* $\alpha = \beta$;
3. $T(n) = O(n^{\beta})$ *se* $\alpha < \beta$.

Esempio
Applicazione del teorema fondamentale

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

1. Nel caso della ricerca binaria, abbiamo $T(n) = T(n/2) + O(1)$. Da cui $a = 1$, $b = 2$, $f(n) = O(1)$; siamo nel secondo caso del teorema, da cui $T(n) = \Theta(\log n)$.
2. Consideriamo $T(n) = 9T(n/3) + n$; in questo caso $a = 9$, $b = 3$ e $f(n) = O(n)$. Siamo nel primo caso, $f(n) = O(n^{\log_b a - \epsilon})$ con $\epsilon = 1$, da cui $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$.

Analisi di algoritmi ricorsivi
Numeri di Fibonacci

Ricordiamo la definizione della sequenza di Fibonacci:

$$F_n = \begin{cases} 1 & \text{se } n = 1,\ 2 \\ F_{n-1} + F_{n-2} & \text{se } n > 2 \end{cases}$$

Consideriamo nuovamente il tempo di esecuzione dell'algoritmo ricorsivo banale per calcolare F_n, il cui tempo di esecuzione $T(n)$ soddisfa la relazione di ricorrenza

$$T(n) = \begin{cases} c_1 & \text{se } n = 1,\ 2 \\ T(n-1) + T(n-2) + c_2 & \text{se } n > 2 \end{cases}$$

Vogliamo produrre un limite inferiore e superiore a $T(n)$

Analisi di algoritmi ricorsivi
Numeri di Fibonacci–limite superiore

Limite superiore. Sfruttiamo il fatto che $T(n)$ è una funzione non decrescente:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + c_2 \\ &\leq 2T(n-1) + c_2 \\ &\leq 4T(n-2) + 2c_2 + c_2 \\ &\leq 8T(n-3) + 2^2 c_2 + 2c_2 + c_2 \\ &\leq \dots \\ &\leq 2^k T(n-k) + c_2 \sum_{i=0}^{k-1} 2^i \\ &\leq \dots \\ &\leq 2^{n-1} c_3 \end{aligned}$$

per una opportuna costante c_3. Quindi $T(n) = O(2^n)$.

Analisi di algoritmi ricorsivi
Numeri di Fibonacci–limite inferiore

Limite inferiore. Sfruttiamo ancora il fatto che $T(n)$ è una funzione non decrescente:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + c_2 \\ &\geq 2T(n-2) + c_2 \\ &\geq 4T(n-4) + 2c_2 + c_2 \\ &\geq 8T(n-6) + 2^2 c_2 + 2c_2 + c_2 \\ &\geq \dots \\ &\geq 2^k T(n-2k) + c_2 \sum_{i=0}^{k-1} 2^i \\ &\geq \dots \\ &\geq 2^{\lfloor n/2 \rfloor} c_4 \end{aligned}$$

per una opportuna costante c_4. Quindi $T(n) = \Omega(2^{\lfloor n/2 \rfloor})$.