▼ 3.0 - Notazione asintotica

La **notazione asintotica** permette di analizzare un algoritmo in base al suo tempo di calcolo e alla sua occupazione di memoria tenendo in considerazione solo la dimensione dell'input.

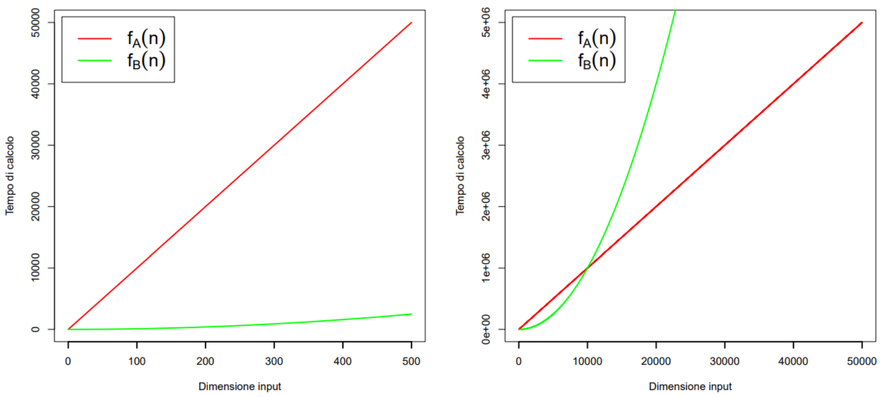
Tenendo in considerazione solamente il comportamento asintotico di un certo algoritmo si è in grado di analizzarlo in maniera corretta senza dover tenere in considerazione ad esempio i secondi o i MB utilizzati da tale algoritmo in quanto questi dipendono da fattori esterni come il linguaggio di programmazione utilizzato per implementarlo e la potenza di calcolo dell'elaboratore utilizzato per eseguirlo.

Il comportamento asintotico di un algoritmo non tiene conto di costanti additive/moltiplicative e termini di ordine inferiore all'interno della formula che mostra l'andamento dell'algoritmo.

Esempio:

- Consideriamo due algoritmi A e B per lo stesso problema.

  Le funzioni che calcolano il tempo di esecuzione dei due algoritmi sono $10^2 n$ per A e $10^{-2} n^2$ per B.



I grafici di tali funzioni assumono la seguente forma.

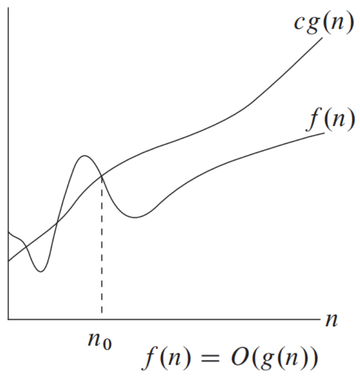
▼ 3.1 - Notazioni

O-grande

> Data una funzione $g(n)$, definiamo l'insieme di funzioni per cui $g(n)$ rappresenta un **limite asintotico superiore** come:
>
> $$O(g(n)) = \{f(n) \mid \exists c > 0, n_0 \geq 0 \text{ tale che } \forall n \geq n_0, f(n) \leq cg(n)\}$$

Dalla definizione capiamo dunque che l'O-grande è riflessivo, in quanto $g(n) = O(g(n))$.

Inoltre, è importante sottolineare che da ora in avanti utilizzeremo l'abuso di notazione $f(n) = O(g(n))$ per indicare che $f(n) \in O(g(n))$.



Esempio grafico di O-grande.

o-piccolo

Data una funzione $g(n)$, definiamo l'insieme di funzioni **dominate asintoticamente** da $g(n)$ come:

$$o(g(n)) = \{f(n) \mid \forall c > 0, \exists n_0 \geq 0 \text{ tale che } \\ \forall n \geq n_0, f(n) < cg(n)\}$$

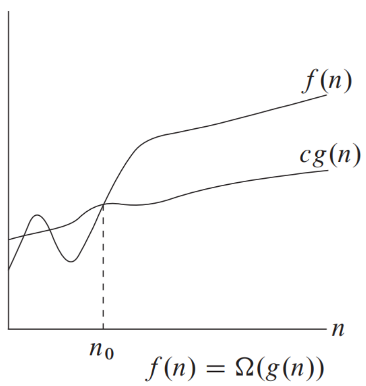
Per definizione $f(n) = o(g(n)) \implies f(n) = O(g(n))$.

Ω-grande

Data una funzione $g(n)$, definiamo l'insieme di funzioni per cui $g(n)$ rappresenta un **limite asintotico inferiore** come:

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0, n_0 \geq 0 \text{ tale che } \forall n \geq n_0, f(n) \geq cg(n)\}$$

Dalla definizione capiamo dunque che l'Ω-grande è riflessivo, in quanto $g(n) = \Omega(g(n))$.



Esempio grafico di Ω-grande.

ω-piccolo

Data una funzione $g(n)$, definiamo l'insieme di funzioni che **dominano asintoticamente** $g(n)$ come:

$$\omega(g(n)) = \{f(n) \mid \forall c > 0, \exists n_0 \geq 0 \text{ tale che } \\ \forall n \geq n_0, f(n) > cg(n)\}$$

Per definizione $f(n) = \omega(g(n)) \implies f(n) = \Omega(g(n))$.

Θ

Data una funzione $g(n)$, definiamo l'insieme di funzioni **asintoticamente equivalenti** a $g(n)$ come:

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

Dalla definizione capiamo dunque che l'Ω-grande è riflessivo, in quanto $g(n) = \Theta(g(n))$.

Teorema: $f(n) \in \Theta(g(n)) \iff f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$.

Notazione asintotica e limiti

L'ordine di crescita asintotico di due funzioni può essere confrontato utilizzando i limiti:

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \implies f(n) = o(g(n)) \implies f(n) = O(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \implies f(n) = \omega(g(n)) \implies f(n) = \Omega(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k > 0 \implies f(n) = \Theta(g(n))$.

Interpretazione intuitiva

Tabella di interpretazione del confronto tra crescita asintotica di funzioni in analogia con il confronto tra numeri reali.

Funzioni	Numeri reali
$f(n) = O(g(n))$	$f \leq g$
$f(n) = o(g(n))$	$f < g$
$f(n) = \Omega(g(n))$	$f \geq g$
$f(n) = \omega(g(n))$	$f > g$
$f(n) = \Theta(g(n))$	$f = g$

Comunque, a differenza di quanto avviene con il confronto tra numeri reali, non tutte le funzioni sono sempre confrontabili in crescita asintotica, ad esempio le due funzioni $f(n) = n$ e $g(n) = n^{\sin(n)+1}$ non sono confrontabili in quanto se $\sin(n) = -1$, $g(n)$ assume valore 1, mentre se $\sin(n) = 1$ assume valore n^2.

Proprietà delle notazioni asintotiche

ϕ: notazione asintotica.

Transitività ($O, o, \Omega, \omega, \Theta$)

$f(n) = \phi(g(n)) \wedge g(n) = \phi(h(n)) \implies f(n) = \phi(h(n))$.

Riflessività (O, Ω, Θ)

$f(n) = \phi(f(n))$.

Simmetria (Θ)

$f(n) = \phi(g(n)) \iff g(n) = \phi(f(n))$.

Simmetria trasposta ($O \iff \Omega, o \iff \omega$)

$f(n) = \phi(g(n)) \iff g(n) = \phi(f(n))$.

Operazioni tra notazioni asintotiche

Somma

$f_1(n) = \phi(g_1(n)) \wedge f_2(n) = \phi(g_2(n)) \implies f_1(n) + f_2(n) = \phi(f_1(n) + f_2(n))$.

Prodotto

$f_1(n) = \phi(g_1(n)) \wedge f_2(n) = \phi(g_2(n)) \implies f_1(n) \times f_2(n) = \phi(f_1(n) \times f_2(n))$.

Moltiplicazione per una costante

$f(n) = \phi(g(n)) \wedge c > 0 \implies c \times f(n) = \phi(g(n))$.

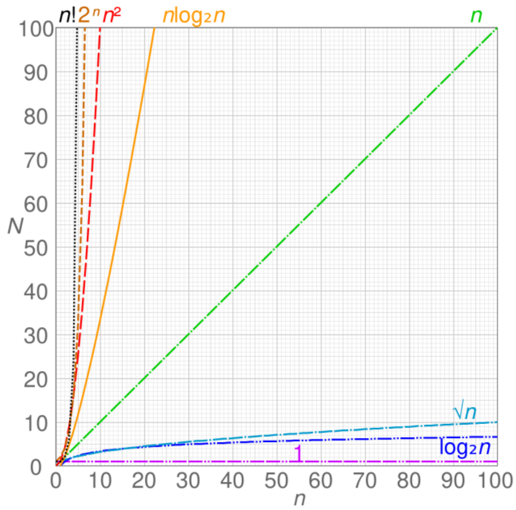
▼ **3.2 - Complessità computazionale**

Ordini di crescita più comuni

Ordine di crescita	Nome
$O(1)$	Constante
$O(\log n)$	Logaritmico
$O(\log^k n)$	Polilogaritmico, $k \geq 1$
$O(n^k)$	Sublineare, $0 < k < 1$
$O(n)$	Lineare
$O(n \log n)$	Pseudolineare
$O(n^k)$	Polinomiale, $k > 1$
$O(n^2)$	Quadratico, per $k = 2$
$O(n^3)$	Cubico, per $k = 3$
$O(c^n)$	Esponenziale, base $c > 1$
$O(n!)$	Fattoriale
$O(n^n)$	Esponenziale, base n

Tabella degli ordini di crescita più comuni.

Nota: viene utilizzata la seguente notazione per i logaritmi: $\log n = \log_2 n$ e $\log^k n = (\log n)^k$.



Confronto tra gli ordini di crescita più comuni.

Complessità computazionale

Complessità computazionale di un algoritmo

> Un **algoritmo** A ha **complessità computazionale** $\phi(f(n))$ rispetto ad una risorsa di calcolo se la quantità di risorse necessarie per eseguirlo su un qualsiasi input di dimensione n è $\phi(f(n))$.

Complessità computazionale di un problema

> Un **problema** P ha **complessità computazionale** $\phi(f(n))$ rispetto ad una risorsa di calcolo se esiste un algoritmo che risolve P con una complessità computazionale $\phi(f(n))$ rispetto a tale risorsa di calcolo.

Analisi del caso ottimo, pessimo e medio

Spesso è necessario analizzare la complessità computazionale per quanto riguarda il caso **ottimo**, **pessimo** e **medio**.

Il caso ottimo descrive il comportamento dell'algoritmo in condizioni ottimali, ad esempio quando l'elemento cercato è il primo all'interno di una lista. Il caso pessimo descrive il comportamento in condizioni sfavorevoli, ad esempio quando l'elemento cercato è l'ultimo di una lista. Il caso medio descrive il comportamento su tutti i possibili input.

Quando si sviluppano algoritmi si è particolarmente interessati a migliorare le prestazioni nel caso pessimo e in quello medio.

Esempio:

- Analizziamo la complessità computazionale di diversi algoritmi per la ricerca di un elemento all'interno di un array.

  Algoritmo 1: ricerca lineare

  ```
  int linsearch(Array A[0 ... n], int x) {
    for (int i = 0; i < n, i++) {
  ```

```
    if A[i] == x then
      return i
  }
  return −1
}
```

- Caso **ottimo**: x è il primo elemento, $O(1)$.
- Caso **pessimo**: x è l'ultimo elemento, $\Theta(n)$.
- Caso **medio**:

  La probabilità che x si trovi in una certa posizione i, compreso anche il caso in cui x non sia presente, è $P_i = \frac{1}{n+1}$.

  Il tempo necessario per ispezionare la posizione i è $T_i = i$.

  Per calcolare la complessità computazione media è dunque necessario sommare la probabilità di ispezione moltiplicata per il relativo tempo di ispezione:

$$\sum_{i=1}^{n} P_i T_i = \frac{1}{n+1} \sum_{i=1}^{n} i = \frac{1}{n+1} \frac{(n+1)(n+2)}{2} = \frac{n+2}{2} = \Theta(n).$$

Algoritmo 2: ricerca binaria

```
int binsearch(Array A[0 ... n], int x) {
  i = 1, j = n

  while (i ≤ j) {
    m = (i + j) / 2
    if A[m] == x then
      return m
    else if A[m] < x then
      i = m + 1
    else
      j = m − 1
  }

  return −1
}
```

- Caso **ottimo**: x si trova in posizione centrale, $O(1)$.
- Caso **pessimo**: x non è presente nell'array.

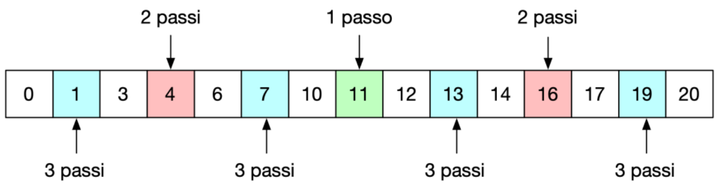
  Dopo ogni iterazione lo spazio di ricerca viene dimezzato, dunque alla i-esima iterazione lo spazio di ricerca equivale a $\frac{n}{2^{i-1}}$.

  Il ciclo while termina quando lo spazio di ricerca è < 1, dunque $\frac{n}{2^{i-1}} < 1 \implies n < 2^{i-1} \implies \log_2 n < \log_2 2^{i-1} \implies i > \log_2 n + 1$.

  Il costo nel caso pessimo è dunque $\Theta(\log_2 n)$.
- Caso **medio**:

  La probabilità che x si trovi in una certa posizione i, escluso il caso in cui x non sia presente per semplificare i calcoli, è $P_i = \frac{1}{n}$.

  Il costo di accesso ad una certa posizione i dipende dalla sua posizione:



Siccome eseguiamo al massimo $\log_2 n$ passi, il costo medio è equivalente a:

$$\frac{1}{n}\sum_{i=1}^{\log_2 n} i2^{i-1}$$

Tale formula è semplificabile nel seguente modo:

$$\frac{1}{n}\sum_{i=1}^{\log_2 n} i2^{i-1} \leq \frac{\log_2 n}{n}\sum_{i=1}^{\log_2 n} 2^i = \frac{\log_2 n}{n} \times \frac{2^{\log_2 n+1}-2}{2-1} = O(\log n).$$

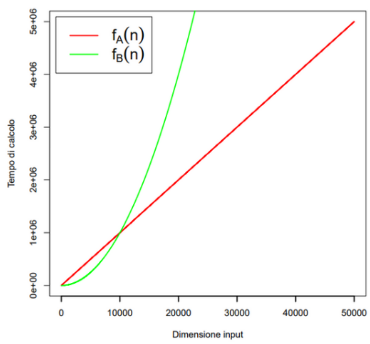
- Analizziamo la complessità computazionale di un algoritmo di ricerca del valore minimo all'interno di un array.

```
int min(Array A[0 ... n]) {
  m = 0
  for (int i = 1; i < n; i++) {
      if (A[i] < A[m]) m = i
  }
  return m
}
```

Il loop viene sempre eseguito $n-1$ volte, dunque i casi ottimo, pessimo e medio coincidono e sono tutti $\Theta(n)$.

Scelta dell'algoritmo

Utilizziamo come esempio due algoritmi A e B che hanno complessità computazionale f_A e f_B, visibili nella seguente figura:



Rappresentazione grafica della complessità computazionale degli algoritmi A e B.

Nella **pratica** spesso viene utilizzato un algoritmo che controlla la dimensione dell'input e, in base a questa, sceglie se utilizzare il primo o il secondo al fine massimizzare le prestazioni in tutti i casi.

Nella **teoria** invece viene preso in considerazione solamente il comportamento asintotico della complessità computazionale, dunque nell'esempio appena fatto l'algoritmo A a discapito di quello B in quanto ha un costo lineare invece che esponenziale.

▼ **3.3 - Analisi ammortizzata**

Molti algoritmi hanno un costo che dipende dallo stato attuale dell'input, dunque una data operazione può essere molto costosa in alcune situazione e molto efficienti in altre, rendendo così molto difficile effettuare analisi probabilistiche per ricavarne la complessità. In questi casi occorre utilizzare l'**analisi ammortizzata** al fine di valutare il costo medio di una sequenza di operazioni.

Esistono due metodi utilizzati per l'analisi ammortizzata: il **metodo dell'aggregazione** e il **metodo degli accantonamenti**.

Vediamo come si comportano nell'analizzare il comportamento di un algoritmo di incremento che consente di incrementare un numero binario di 1 unità utilizzando un array per rappresentare il numero in input. Tale algoritmo è il seguente:

```
void increment(Array A[1 ... k])
  i = k
  while (i ≥ 1 and A[i] == 1) {
    A[i] = 0
    i = i - 1
  }
  if (i ≥ 1) // counter overflow
    A[i] = 1
```

Possiamo inoltre visualizzare in maniera grafica diverse operazioni di incremento di un numero binario il quale costo dipende dall'attuale stato del numero in input:

Valore	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	Costo
0	0	0	0	0	0	0
1	0	0	0	0	1	1
2	0	0	0	1	0	2
3	0	0	0	1	1	1
4	0	0	1	0	0	3
5	0	0	1	0	1	1
6	0	0	1	1	0	2
7	0	0	1	1	1	1
8	0	1	0	0	0	4
9	0	1	0	0	1	1
10	0	1	0	1	0	2

Operazioni di incremento di numeri binari e relativo costo.

Metodo dell'aggregazione

Il **metodo dell'aggregazione** consiste nel determinare un limite superiore al costo totale di una sequenza di n operazioni per poi dividere tutto per n ed ottenere il costo medio di una singola operazione.

Notiamo dalla tabella sopra che il k-esimo bit viene cambiato ad ogni incremento, il k-1-esimo bit ogni due incrementi, k-2-esimo bit ogni 4 incrementi e così via. Tramite questa osservazione possiamo dunque costruire una formula per calcolare il costo totale di n operazioni: $n + \frac{n}{2} + ... + \frac{n}{2^{k-1}} = \sum_{i=0}^{k-1} \frac{n}{2^i} \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = n \frac{1}{1 - \frac{1}{2}} = 2n.$

Abbiamo dunque trovato che il costo totale di n operazioni è $O(n)$, e da questo possiamo trovare il costo ammortizzato per operazione dividendo per n: $\frac{O(n)}{n} = O(1)$.

Metodo degli accantonamenti

Il **metodo degli accantonamenti** è un metodo basato sulla contabilità economica. Consiste nel fare una previsione assegnando un **costo "ammortizzato"** per svolgere una singola operazione. In questo modo addebitiamo ogni operazione con il suo costo ammortizzato, salvando in un credito la differenza tra il suo costo ammortizzato e il costo reale. Il credito servirà poi in futuro per pagare operazioni che hanno un costo reale maggiore di quello ammortizzato. Capiamo che il costo ammortizzato scelto inizialmente è quello corretto se il credito non è mai negativo.

Utilizziamo tale metodo per calcolare il costo di una singola operazione della funzione increment. Assegniamo un costo ammortizzato di 2€ per cambiare ad 1 un bit con valore 0. Scegliamo proprio 2 per via del fatto che qualunque bit che viene trasformato da 0 a 1 verrà poi ritrasformato da 1 a 0 prima o poi, dunque ci servirà 1€ di credito per farlo. In questo modo, ogni volta che un bit viene trasformato da 1 a 0 viene accumulato un credito di 1€, dunque il credito residuo per una certa operazione è equivalente al numero di 1 presenti nel numero in input. Siccome il numero di 1 presenti in input non è mai negativo, allora neanche il credito sarà mai negativo, dunque abbiamo dimostrato che il costo ammortizzato di 2€ è esatto.

Siccome a ogni operazione abbiamo assegnato un costo di 2€, allora il costo totale è equivalente a $2n$€, e il costo ammortizzato è uguale a $\frac{2n}{n} = O(1)$.

Possiamo visualizzare in maniera grafica il metodo appena utilizzato:

Valore	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	Credito residuo	Costo totale
0	0	0	0	0	0	0	0
1	0	0	0	0	1€ 1	1	2
2	0	0	0	1€ 1	0€ 0	1	4
3	0	0	0	1€ 1	1€ 1	2	6
4	0	0	1€ 1	0€ 0	0€ 0	1	8
5	0	0	1€ 1	0€ 0	1€ 1	2	10
6	0	0	1€ 1	1€ 1	0€ 0	2	12
7	0	0	1€ 1	1€ 1	1€ 1	3	14
8	0	1€ 1	0€ 0	0€ 0	0€ 0	1	16
9	0	1€ 1	0€ 0	0€ 0	1€ 1	2	18
10	0	1€ 1	0€ 0	1€ 1	0€ 0	2	20

Utilizzo del metodo degli accantonamenti nel costo ammortizzato dell'algoritmo increment.

▼ 3.4 - Equazioni di ricorrenza

Le **equazioni di ricorrenza** descrivono ogni elemento in una sequenza in termini degli elementi precedenti. Per questo motivo è possibile utilizzarle per determinare la crescita asintotica degli algoritmi ricorsivi.

Vedremo **3 modi** per risolvere equazioni di ricorrenza:

- Metodo dell'**iterazione**
- Metodo della **sostituzione**
- **Master Theorem**

Le equazioni di ricorrenza di algoritmi ricorsivi sono solitamente del tipo:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(n/3) + O(n) & n > 1 \end{cases}$$

Tipicamente vengono sostituite le notazioni asintotiche con espressioni positive:

$$T(n) = \begin{cases} d & n = 1 \\ 2T(n/3) + cn & n > 1 \end{cases}$$

Inoltre viene tipicamente utilizzata la costante 1 al posto delle costanti simboliche:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n/3) + n & n > 1 \end{cases}$$

Metodo dell'iterazione

Il **metodo dell'iterazione** consiste nel sostituire in modo iterativo la parte ricorsiva dell'equazione finchè non appare uno schema sintetizzabile tramite una formula. A questo punto occorre chiedersi quando la ricorrenza termina e in questo modo si riesce a determinare la crescita asintotica dell'agoritmo ricorsivo.

Esempio:

- Utilizziamo il metodo dell'iterazione nell'equazione di ricorrenza:

  $$T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + c & n > 1 \end{cases}$$

  Sostituendo la parte ricorsiva in modo iterativo otteniamo:

  $T(n)$
  $= T(n/2) + c$
  $= T(n/4) + c + c$
  $= T(n/8) + c + c$
  $\dots$
  $= T(n/2^i) + c \cdot i$

  La ricorsione termina quando $n/2^i = 1 \implies i = \log_2 n$.

  Quindi $T(n) = T(1) + c \cdot \log_2 n = 1 + c \cdot \log_2 n = \Theta(\log n)$.

  Come abbiamo visto in precedenza possiamo sostituire la costante c con 1, il risultato non cambierebbe.

Metodo della sostituzione

Il **metodo della sostituzione** consiste nell'ipotizzare una soluzione e nel cercare di validare tale ipotesi effettuando una dimostrazione tramite induzione.

Esempio:

- Utilizziamo il metodo della sostituzione nell'equazione di ricorrenza:

  $$T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + n & n > 1 \end{cases}$$

  Ipotizziamo che $T(n) = O(n)$, ovvero che $\exists c > 0, n_0 \geq 0$ tale che $\forall n \geq n_0 .\ T(n) \leq cn$.

  - $n = 1$

    $T(1) \leq c \cdot 1 \implies 1 \leq c$, vero $\forall c \geq 1$.

- $n > 1$

  Assumiamo per ipotesi induttiva che l'ipotesi fatta sia vera per $T(n/2)$, quindi $T(n/2) \leq c \cdot (n/2)$.

  Dobbiamo provare che $T(n) \leq c \cdot n \implies T(n/2) + n \leq c \cdot n$.

  Per ipotesi induttiva abbiamo quindi che $c \cdot (n/2) + n \leq c \cdot n \implies (c/2 + 1) \cdot n \leq c \cdot n \implies c/2 + 1 \leq c$, vero $\forall c \geq 2$.

L'ipotesi fatta è dunque corretta, quindi $T(n) = O(n)$.

- Utilizziamo il metodo della sostituzione nell'equazione di ricorrenza di Fibonacci:

$$T(n) = \begin{cases} 1 & n \leq 2 \\ T(n-1) + T(n-2) + 1 & n > 2 \end{cases}$$

Ipotizziamo che $T(n) = O(2^n)$, ovvero che $\exists c > 0, n_0 \geq 0$ tale che $\forall n \geq n_0 .\ T(n) \leq c \cdot 2^n$.

- $n \leq 2$

  $T(1) \leq c \cdot 2^1 \implies 1 \leq c \cdot 2$, vero $\forall c \geq \frac{1}{2}$.

  $T(2) \leq c \cdot 2^2 \implies 1 \leq c \cdot 4$, vero $\forall c \geq \frac{1}{4}$.

- $n > 2$

  Assumiamo per ipotesi induttiva che l'ipotesi fatta sia vera per $T(n-1)$ e $T(n-2)$, quindi $T(n-1) \leq c \cdot 2^{n-1}$ e $T(n-2) \leq c \cdot 2^{n-2}$.

  Dobbiamo provare che $T(n) \leq c \cdot 2^n \implies T(n-1) + T(n-2) + 1 \leq c \cdot 2^n$.

  Per ipotesi induttiva abbiamo quindi che $c \cdot 2^{n-1} + c \cdot 2^{n-2} + 1 \leq c \cdot 2^n \implies c \cdot 2 \cdot 2^{n-2} + c \cdot 2^{n-2} + 1 \leq c \cdot 2^n \implies c \cdot 2^{n-2}(2+1) + 1 \leq c \cdot 2^n$.

  $\forall n \geq 2 - \log_2 c$ possiamo arrivare alla forma $c \cdot 2^{n-2}(2+2) \leq c \cdot 2^n \implies c \cdot 2^n \leq c \cdot 2^n$, vero $\forall c \in \mathbb{R}$.

L'ipotesi fatta è quindi corretta, dunque $T(n) = O(2^n)$.

È possibile inoltre dimostrare tramite lo stesso metodo che la ricorrenza di Fibonacci è limitata inferiormente da $\Omega(\sqrt{2}^n)$, ma è difficile trovare dei limiti più stretti andando a tentativi con il metodo della sostituzione.

Master Theorem

Il Master Theorem è un approccio per risolvere ricorrenze della forma

⁍

con $a \geq 1$ e $b > 1$ constanti e $f(n)$ asintoticamente positiva. Il costo di ogni chiamata ricorsiva è dato da $f(n)$.

Si consideri la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} d & n = 1 \\ aT(n/b) + cn^\beta & n > 1 \end{cases}$$

dove $a \geq 1, b > 1$ e c, d costanti.

Sia $\alpha = \log_b a = \frac{\log a}{\log b}$. Allora

- Se $\alpha > \beta$ allora $T(n) = \Theta(n^\alpha)$
- Se $\alpha = \beta$ allora $T(n) = \Theta(n^\alpha \log n)$
- Se $\alpha < \beta$ allora $T(n) = \Theta(n^\beta)$

Esempio:

- Utilizziamo il Master Theorem per individuare la crescita asintotica del seguente algoritmo di ricerca binaria:

```
int binsearch(Array A[1 ... n], int x, int i, int j)
  if (i > j) return −1
  else {
    m = (i + j) / 2
    if (A[m] == x)
      return m
    else if (A[m] > x)
      return search(A, x, i, m − 1)
    else
      return search(A, x, m + 1, j)
  }
```

Dallo pseudocodice ricaviamo che la funzione di ricorrenza è la seguente:

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n/2) + 1 & n > 0 \ \text{(ricerca su } 1/2 \text{ dello spazio)} \end{cases}$$

$\alpha = \log_2 1 = 0$ e $\beta = 0 \implies \alpha = \beta \implies T(n) = \Theta(n^0 \log n) = \Theta(\log n)$.