Algoritmi e Strutture di Dati I

Massimo Franceschet
http://www.sci.unich.it/~francesc

Problemi

Un **problema** specifica in termini generali una *relazione* che intercorrere tra dei dati di ingresso (input) e dei dati di uscita (output).

Problema dell'ordinamento *Data in ingresso una sequenza* $\langle a_1, a_2, \ldots, a_n \rangle$ *di* n *numeri, fornire in uscita una permutazione* $\langle a'_1, a'_2, \ldots, a'_n \rangle$ *della sequenza tale che* $a'_1 \leq a'_2, \ldots, \leq a'_n$.

Algoritmi

Un **algoritmo** è una *procedura computazionale* che prende un valore in **ingresso** (**input**) e restituisce un valore in **uscita** (**output**).

Un algoritmo è **corretto** rispetto ad un problema oppure **risolve** un problema se, per ogni ingresso del problema, l'algoritmo termina fornendo l'uscita del problema.

Monte Carlo Sort *Sia S una sequenza di n elementi. L'algoritmo genera casualmente una permutazione S' di S, verifica se S' è ordinata, e, se lo è, la restituisce in uscita terminando la procedura. Dire se l'algoritmo risolve il problema dell'ordinamento nei casi: (A) il generatore casuale di permutazioni ha memoria; (B) il generatore casuale di permutazioni non ha memoria.*

Programmi

Con il termine **programma** si intende la trascrizione (implementazione) dell'algoritmo in un linguaggio formale di programmazione (Pascal, C, Java, ...).

La differenza tra algoritmo e programma è la seguente: un algoritmo è una procedura computazionale intelligibile dall'uomo, un programma è una procedura computazionale comprensibile dalla macchina.

Ne deriva che il linguaggio in cui scriviamo gli algoritmi è più astratto del linguaggio di programmazione.

Decidibilità

Un problema è **decidibile** se esiste almeno un algoritmo che lo risolve. Esistono problemi indecidibili? Quanti?

Problema di corrispondenza di Post *Date due sequenze di* n *parole* $A = \langle w_1, \ldots, w_n \rangle$, $B = \langle v_1, \ldots, v_n \rangle$, *trovare, se esiste, una sequenza di indici* $\langle i_1, \ldots, i_m \rangle$, *con* $m \geq 1$, *tale che*

$$w_{i_1} \ldots w_{i_m} = v_{i_1} \ldots v_{i_m}.$$

Ad esempio, se $A = \langle 1, 10111, 10 \rangle$, e $B = \langle 111, 10, 0 \rangle$, esiste una soluzione. Quale?

Decidibilità

Un problema è **decidibile** se esiste almeno un algoritmo che lo risolve. Esistono problemi indecidibili? Quanti?

Problema di corrispondenza di Post *Date due sequenze di n parole $A = \langle w_1, \dots, w_n \rangle$, $B = \langle v_1, \dots, v_n \rangle$, trovare, se esiste, una sequenza di indici $\langle i_1, \dots, i_m \rangle$, con $m \geq 1$, tale che*

$$w_{i_1} \dots w_{i_m} = v_{i_1} \dots v_{i_m}.$$

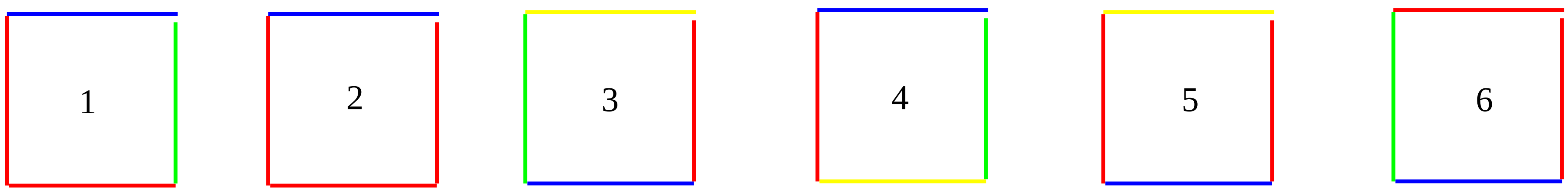
Ad esempio, se $A = \langle 1, 10111, 10 \rangle$, e $B = \langle 111, 10, 0 \rangle$, esiste una soluzione. Quale?

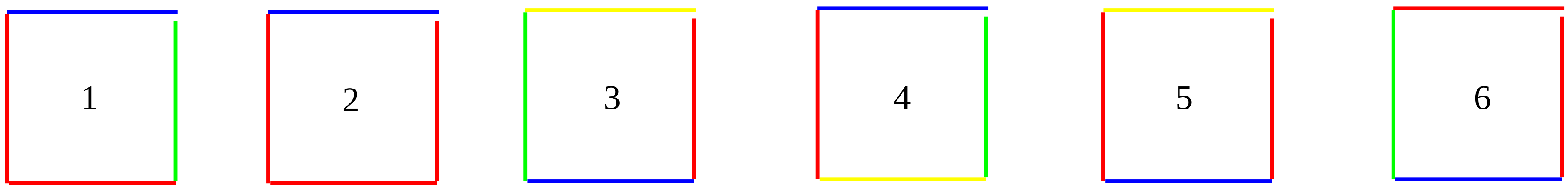
La soluzione è $\langle 2, 1, 1, 3 \rangle$. Infatti $10111\,1\,1\,10 = 10\,111\,111\,0$.

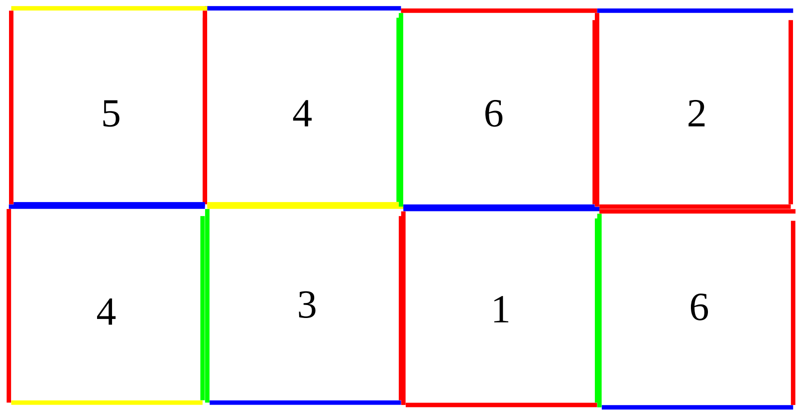
Un altro problema indecidibile

Problema del domino *Abbiamo un insieme finito di tipi di piastrelle quadrate. Ogni tipo è contraddistinto dal colore dei quattro lati delle piastrelle.*

Avendo a disposizione un numero infinito di piastrelle di questo tipo, trovare, se esiste, un modo per piastrellare una parete infinita in lunghezza e in larghezza assicurando che i lati adiacenti delle piastrelle abbiano lo stesso colore.


1
2
3
4
5
6


1
2
3
4
5
6


5
4
6
2
4
3
1
6

Complessità computazionale

In generale, esistono più algoritmi diversi che risolvono un problema decidibile. Quale scegliere?

Complessità computazionale

In generale, esistono più algoritmi diversi che risolvono un problema decidibile. Quale scegliere?

Quello che ha costo o **complessità** inferiore. La complessità computazionale di un algoritmo è la **quantità di risorse** che l'algoritmo richiede per terminare.

Complessità computazionale

In generale, esistono più algoritmi diversi che risolvono un problema decidibile. Quale scegliere?

Quello che ha costo o **complessità** inferiore. La complessità computazionale di un algoritmo è la **quantità di risorse** che l'algoritmo richiede per terminare.

Quali risorse sono significative per il calcolo della complessità?

Complessità computazionale

La complessità di un dato algoritmo è funzione della **dimensione** e della **forma** dei dati in ingresso.

Prendiamo il problema di ordinamento. Ci aspettiamo che più è lunga la sequenza, più risorse l'algoritmo impieghi per terminare.

Inoltre, ci aspettiamo che, a parità di lunghezza, più grande è il numero di coppie ordinate in partenza, meno risorse l'algoritmo impieghi per terminare.

Complessità computazionale

Al fine di rendere univoca la definizione di complessità, esprimiamo la complessità in funzione della **dimensione dell'ingresso**. Inoltre, fissata la dimensione dell'input, consideriamo tre tipi di complessità:

- **complessità ottima**: è la complessità dell'algoritmo nel caso migliore. Cioè, scelgo la più piccola tra tutte le complessità generate dall'algoritmo sugli input di una certa dimensione;
- **complessità media**: è la complessità dell'algoritmo nel caso medio. Cioè, faccio la media tra tutte le complessità generate dall'algoritmo sugli input di una certa dimensione;
- **complessità pessima**: è la complessità dell'algoritmo nel caso peggiore. Cioè, scelgo la più grande tra tutte le complessità generate dall'algoritmo sugli input di una certa dimensione.

Complessità computazionale

Complessità di Monte Carlo Sort *Si riconsideri l'algoritmo di ordinamento Monte Carlo. Supponendo che esista una procedura che verifica se una sequenza è ordinata con costo unitario, calcolare nei casi (A) e (B) la complessità ottima, media e pessima dell'algoritmo.*

Limiti asintotici

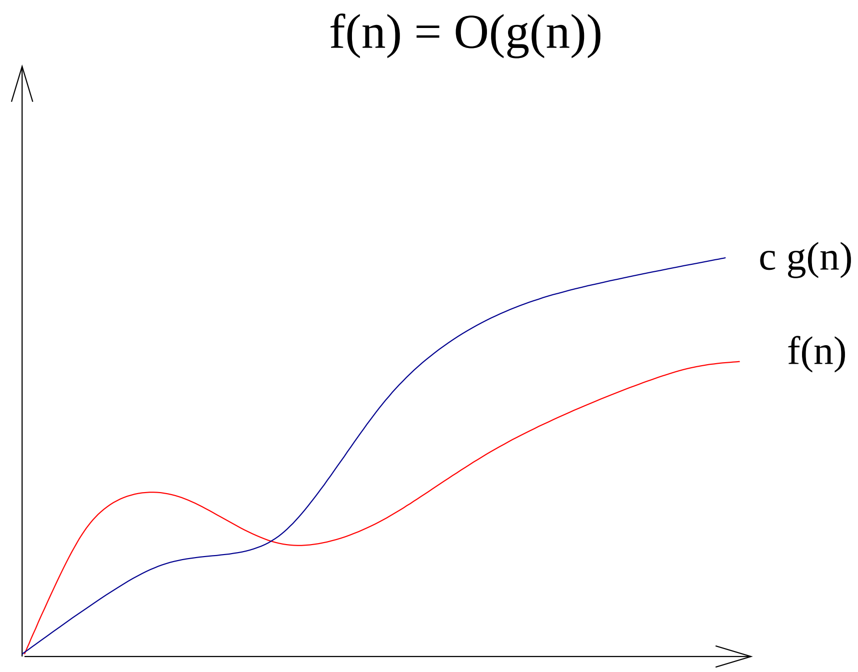
Limite superiore asintotico. Date due funzioni $f(n)$ e $g(n)$, scriveremo

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

qualora esistano $c > 0$ e $n_0 \geq 0$ tali che

$$f(n) \leq cg(n)$$

per ogni $n \geq n_0$.


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

Limiti asintotici

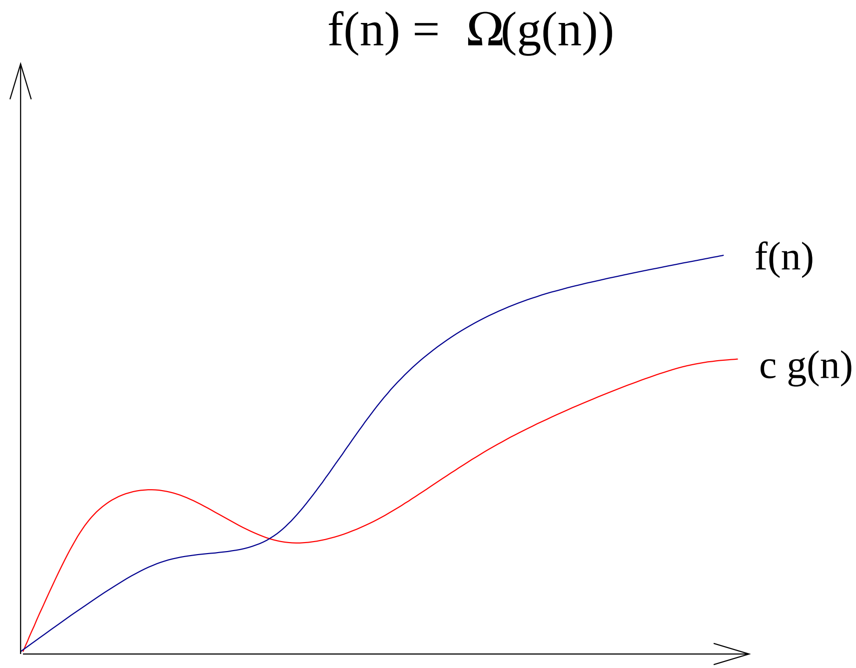
Limite inferiore asintotico Ω. Date due funzioni $f(n)$ e $g(n)$, scriveremo

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

qualora esistano $c > 0$ e $n_0 \geq 0$ tali che

$$f(n) \geq cg(n)$$

per ogni $n \geq n_0$;


f(n) = Ω(g(n))
f(n)
c g(n)

Limiti asintotici

Limite asintotico stretto Θ. Date due funzioni $f(n)$ e $g(n)$, scriveremo

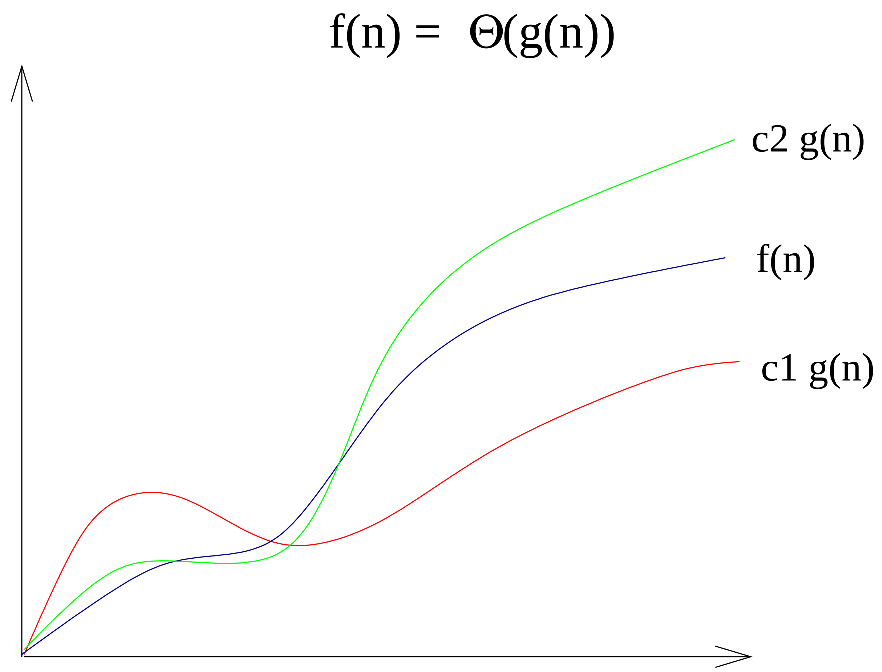
$$f(n) = \Theta(g(n))$$

qualora esistano $c_1, c_2 > 0$ e $n_0 \geq 0$ tali che

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

per ogni $n \geq n_0$.

Si noti che $f(n) = \Theta(g(n))$ se e solo se $f(n) = \Omega(g(n))$ e $f(n) = O(g(n))$.


f(n) = Θ(g(n))
c2 g(n)
f(n)
c1 g(n)

Limiti asintotici

Date due funzioni $f(n)$ e $g(n)$, diremo che le funzioni $f(n)$ e $g(n)$ sono **asintoticamente equivalenti**, scritto $f(n) \sim g(n)$, se $f(n) = \Theta(g(n))$.

Diremo che $f(n)$ è **asintoticamente inferiore** a $g(n)$, scritto $f(n) \prec g(n)$, se $f(n) = O(g(n))$ e $f(n) \neq \Omega(g(n))$.

Esercizio 1 *Dimostrare che:*

- $\sim$ *è una relazione di equivalenza (cioè* $\sim$ *è riflessiva, simmetrica e transitiva); e*
- $\prec$ *è una relazione di ordinamento stretto (cioè* $\prec$ *è irriflessiva e transitiva).*

Limiti asintotici

Di seguito è indicata una scala asintotica crescente di complessità:

$$\log n \prec \sqrt[p]{n} \prec n \log n \prec n^k \prec 2^n \prec n! \prec n^n \prec 2^{2^n},$$

ove $p \geq 1$ e $k > 1$.

Diremo che un algoritmo ha **complessità polinomiale** se

$$f(n) = \Theta(n^k),$$

con $k \geq 1$, ed ha **complessità esponenziale** se

$$f(n) = \Theta(2^n).$$

La **complessità di un problema** è la complessità del miglior algoritmo che lo risolve. Un problema si dice **facile** se ha complessità al più polinomiale, e **difficile** se ha complessità almeno esponenziale.

Uno dei due è difficile...

Problema del cammino minimo *Abbiamo un insieme di città, alcune delle quali sono collegate da vie percorribili. Ogni collegamento è associato ad un costo (ad esempio il tempo di percorrenza). Il problema del cammino minimo consiste nel trovare il cammino di costo minimo tra due città.*

Problema del commesso viaggiatore *Abbiamo un insieme di città, collegate a due a due da vie percorribili. Ad ogni collegamento è associato un costo (ad esempio il tempo di percorrenza). Il problema del commesso viaggiatore consiste nel trovare un ciclo di costo minimo che passa esattamente una volta per ogni città.*

Limiti e limiti asintotici

E' possibile usare i limiti matematici per confrontare asintoticamente l'andamento di due funzioni di complessità.

Teorema *Date due funzioni di complessità $f(n)$ e $g(n)$, vale che:*

- *se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, *allora* $f(n) \prec g(n)$;
- *se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, *allora* $g(n) \prec f(n)$;
- *se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = l \neq 0$, *allora* $f(n) \sim g(n)$;
- *se infine* $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ *non esiste, allora* $f(n)$ *e* $g(n)$ *non sono confrontabili.*

Usando i limiti matematici, è facile vedere che:

1. $c = \Theta(1)$;
2. $2n = \Theta(n)$ e $n + 1 = \Theta(n)$;
3. $2^{n+1} = \Theta(2^n)$ e $2^{2n} \neq \Theta(2^n)$.

La **morale** è (ripetere per 3 volte al giorno!):

1. ogni costante è asintotica all'unità;
2. le costanti additive e moltiplicative sono asintoticamente irrilevanti se applicate alla **base della funzione**;
3. le costanti additive ad esponente sono asintoticamente irrilevanti, ma le costanti moltiplicative **ad esponente** sono significative.

Mostriamo che

$$\log(n!) = \Theta(n \log n)$$

Infatti

$$\lim_{n\to\infty} \frac{\log(n!)}{n \log n} = 1,$$

e dunque la tesi per il teorema precedente. Per la **formula di Stirling**, abbiamo che

$$n! = \sqrt{2\pi n}(n/e)^n e^{\alpha_n},$$

ove $1/(12n+1) < \alpha_n < 1/(12n)$. Dunque

$$\log(n!) = \log\sqrt{2\pi n} + n\log(n/e) + \alpha_n \log e.$$

Abbiamo che

$$\lim_{n\to\infty} \frac{\log\sqrt{2\pi n}}{n \log n} = 0, \quad \lim_{n\to\infty} \frac{n\log(n/e)}{n \log n} = 1, \quad \lim_{n\to\infty} \frac{\alpha_n \log e}{n \log n} = 0,$$

e dunque la tesi.

Gli operatori asintotici distribuiscono rispetto alla addizione e alla moltiplicazione, cioè:

1. $\Theta(f(n)) + \Theta(g(n)) = \Theta(f(n) + g(n))$;
2. $\Theta(f(n))\Theta(g(n)) = \Theta(f(n)g(n))$.

Ad esempio:

$\Theta(n) + \Theta(1) = \Theta(n + 1) = \Theta(n)$

$\Theta(n) + \Theta(n) = \Theta(n + n) = \Theta(2n) = \Theta(n)$

$\Theta(n)\Theta(1) = \Theta(n1) = \Theta(n)$

$\Theta(n)\Theta(n) = \Theta(nn) = \Theta(n^2)$.

Equazioni ricorsive

Una **equazione ricorsiva** è una equazione del tipo

$$C(n) = \ldots C(m) \ldots$$

in cui a destra e a sinistra del segno di uguaglianza compare la stessa funzione C. In sostanza, C è definita **implicitamente in termini di sè stessa**.

La funzione di complessità di una **procedura ricorsiva**, cioè di una procedura che richiamano se stessa, è spesso una equazione ricorsiva.

Per capire l'andamento asintotico di una equazione ricorsiva, occorre calcolare la sua **forma esplicita**, cioè eliminare l'occorrenza della funzione alla destra del segno di uguaglianza.

Equazioni ricorsive

Vi sono varie tecniche per risolvere una equazione ricorsiva. Forse la più efficacie consiste nel seguire i seguenti passi:

1. **svolgere l'equazione** un numero sufficiente di volte e **congetturare** una possibile forma esplicita;

2. **dimostrare la congettura** mediante l'uso di strumenti formali.

Quali strumenti formali? Master Theorem, induzione matematica...

Equazioni ricorsive

$$
\begin{aligned}
C(1) &= 1 \\
C(n) &= C(n/2) + 1
\end{aligned}
$$

Equazioni ricorsive

$$
\begin{aligned}
C(n) &= 1 + C(n/2) \\
&= 1 + 1 + C(n/4) \\
&= 1 + 1 + 1 + C(n/8) \\
&= \ldots \\
&= \sum_{i=0}^{\log n} 1 = \log n + 1 \sim \log n
\end{aligned}
$$

Equazioni ricorsive

$$
\begin{aligned}
C(1) &= 1 \\
C(n) &= C(n/2) + n
\end{aligned}
$$

Equazioni ricorsive

$$\begin{aligned} C(n) &= n + C(n/2) \\ &= n + n/2 + C(n/4) \\ &= n + n/2 + n/4 + C(n/8) \\ &= \ldots \\ &= \textstyle\sum_{i=0}^{\log n} n/2^i = n \sum_{i=0}^{\log n} 1/2^i \sim n \end{aligned}$$

Infatti

$$1 \leq \sum_{i=0}^{\log n} 1/2^i < 2$$

Equazioni ricorsive

$$\begin{aligned} C(1) &= 1 \\ C(n) &= 2C(n/2) + 1 \end{aligned}$$

Equazioni ricorsive

$$
\begin{aligned}
C(n) &= 1 + 2C(n/2) \\
&= 1 + 2[1 + 2C(n/4)] \\
&= 1 + 2 + 4C(n/4) \\
&= 1 + 2 + 4[1 + 2C(n/8)] \\
&= 1 + 2 + 4 + 8C(n/8) \\
&= \ldots \\
&= \sum_{i=0}^{\log n} 2^i = 2^{\log n + 1} - 1 = 2n - 1 \sim n
\end{aligned}
$$

Equazioni ricorsive

$$
\begin{aligned}
C(1) &= 1 \\
C(n) &= 2C(n/2) + n
\end{aligned}
$$

Equazioni ricorsive

$$
\begin{aligned}
C(n) &= n + 2C(n/2) \\
&= n + 2[n/2 + 2C(n/4)] \\
&= n + n + 4C(n/4) \\
&= n + n + 4[n/4 + 2C(n/8)] \\
&= n + n + n + 8C(n/8) \\
&= \ldots \\
&= \textstyle\sum_{i=0}^{\log n} n = n(\log n + 1) \sim n \log n
\end{aligned}
$$

Equazioni ricorsive

Master Theorem *Siano* $a \geq 1$ *e* $b > 1$ *due costanti intere, e* $f(n)$ *una funzione di complessità. Sia* $C(n) = aC(n/b) + f(n)$ *una equazione ricorsiva definita sui naturali, ove per* n/b *si intende* $\lfloor n/b \rfloor$ *oppure* $\lceil n/b \rceil$*. Allora:*

1. *Se* $f(n) = O(n^{\log_b a - \epsilon})$ *per qualche* $\epsilon > 0$*, allora* $C(n) = \Theta(n^{\log_b a})$.
2. *Se* $f(n) = \Theta(n^{\log_b a})$*, allora* $C(n) = \Theta(n^{\log_b a} \log n)$.
3. *Se* $f(n) = \Omega(n^{\log_b a + \epsilon})$ *per qualche* $\epsilon > 0$*, e se esiste una costante* $c < 1$ *tale che* $af(n/b) \leq cf(n)$ *per tutti gli* n *sufficientemente grandi, allora* $C(n) = \Theta(f(n))$.

Equazioni ricorsive

Sia

$$C(n) = C(n/2) + 1.$$

Allora $a = 1$, $b = 2$, $f(n) = 1$. Dunque $f(n) = \Theta(n^{\log_2 1}) = \Theta(1)$. Il caso 2 del Master Theorem si applica, e quindi $C(n) = \Theta(\log n)$.

Sia

$$C(n) = 2C(n/2) + 1.$$

Allora $a = 2$, $b = 2$, $f(n) = 1$. Dunque $f(n) = \Theta(n^{\log_2 2 - \epsilon}) = \Theta(1)$, ponendo $\epsilon = 1$. Il caso 1 del Master Theorem si applica, e quindi $C(n) = \Theta(n)$.

Equazioni ricorsive

Alcune volte il Master Theorem non è applicabile. Ad esempio nei seguenti casi:

1. $C(1) = 1$ e, per $n > 0$, $C(n) = C(n-1) + 1$;
2. $C(1) = 0$ e, per $n > 0$, $C(n) = C(n-1) + \log n$;
3. $C(1) = 1$ e, per $n > 0$, $C(n) = C(n-1) + n$;

Risolviamo la terza. **Congetturiamo** una forma esplicita srotolando l'equazione...

$$\begin{aligned} C(n) &= n + C(n-1) \\ &= n + n - 1 + C(n-2) \\ &= n + n - 1 + n - 2 + \ldots 1 \\ &= \textstyle\sum_{i=1}^{n} i = n(n+1)/2 \end{aligned}$$

Dimostriamo la congettura per induzione. Se $n = 1$, allora $C(1) = 1 \cdot 2/2 = 1$. Sia $n > 1$. Allora $C(n) = n + C(n-1)$ per definizione. Applicando l'ipotesi induttiva, abbiamo che

$$C(n) = n + (n-1)n/2 = (n^2 + 2n - n)/2 = n(n+1)/2$$

Pseudocodice

Una **variabile** è una locazione di memoria che può essere letta e scritta. Una **costante** è una locazione di memoria che può essere solo letta.

Nelle **espressioni aritmetiche** useremo le operazioni: $+$, $-$, $*$, $/$, *div* (divisione intera), *mod* (resto). Useremo i seguenti operatori di confronto: $<$, $\leq$, $>$, $\geq$, $=$, $\neq$.

Nelle **formule logiche Booleane** useremo i **valori di verità** TRUE e FALSE e i **connettivi logici** *and*, *or* e *not*.

- *Assegnamento*. Sia x una variabile e y una variabile o una costante. La **sintassi** dell'assegnamento è:

  1: $x \leftarrow y$

  La **semantica** è la seguente: assegno alla variabile x il contenuto di y; x viene scritta, mentre y viene letta.

- *Condizionale.* Siano C una condizione Booleana, B_1 e B_2 sottoprogrammi. La sintassi del condizionale è:

  if C **then**
  $\quad B_1$
  else
  $\quad B_2$
  end if

  Le semantica è la seguente: verifico la condizione C. Se C è vera, eseguo B_1, altrimenti eseguo B_2. Il ramo *else* può essere omesso.

```
x ← 0
y ← 1
x ← y
y ← x
if x > y then
    print "Mulder"
else
    print "Scully"
end if
```

$x \leftarrow 0$
$y \leftarrow 1$
$z \leftarrow x$
$x \leftarrow y$
$y \leftarrow z$
if $x > y$ **then**
    print $''Mulder''$
else
    print $''Scully''$
end if

- *While*. Siano C una condizione Booleana e B un sottoprogramma. La sintassi del *while* è la seguente:

  1: **while** C **do**
  2: $\quad B$
  3: **end while**

  La semantica è la seguente: eseguo il blocco B fino a che la condizione C diventa falsa.

```
x ← 0
y ← 1
while x < y do
    x ← x + 1
    y ← y + 1
end while
```

$x \leftarrow 1$
$y \leftarrow 0$
while $x < y$ **do**
    $x \leftarrow x + 1$
    $y \leftarrow y + 1$
end while

- *Repeat*. Siano C una condizione Booleana e B un sottoprogramma. La sintassi del *repeat* è la seguente:

```
1: repeat
2:     B
3: until C
```

  La semantica è la seguente: eseguo una volta il blocco B e poi lo eseguo ancora fino a che la condizione C diventa vera.

$x \leftarrow 0$
$y \leftarrow 1$
repeat
    $x \leftarrow x + 1$
until $x = y$

- *For*. Siano x una variabile, n e m due numeri interi, e B un sottoprogramma. La sintassi del *for* è la seguente:

```
for x ← n to m do
    B
end for
```

  La semantica è la seguente: x assume i valori da n a m in ordine crescente e per ogni valore assunto da x eseguo una volta il blocco B. Se sostituiamo il **to** con un **downto**, la variabile x assume i valori da n a m in ordine decrescente.

$n \leftarrow 1$
$m \leftarrow n + 1$
for $x \leftarrow n$ **to** m **do**
    print x
end for

```
n ← 1
m ← n
for x ← n to m do
    print x
end for
```

$n \leftarrow 1$
$m \leftarrow n - 1$
for $x \leftarrow n$ **to** m **do**
    print x
end for

Inoltre, useremo le seguenti ulteriori istruzioni:

- *Ritorno.* Sia x una variabile o una costante. L'istruzione

  return x

  ha l'effetto di **terminare la procedura** in corso e restituire alla procedura chiamante il contenuto di x.

- *Errore.* Sia x una variabile o una costante. L'istruzione

  error x

  ha l'effetto di **terminare l'intero programma** e restituire al sistema operativo il contenuto di x.

- *Stampa.* Sia x una variabile o una costante. L'istruzione

  print x

  ha l'effetto di stampare il contenuto di x.

Procedure iterative e ricorsive

- Una **procedura ricorsiva** è una procedura che richiama sè stessa.
- Una **procedura iterativa** può chiamare altre procedure ma non può invocare sè stessa.

Ogni procedura ricorsiva può essere riscritta in forma iterativa.

Fattoriale

$$n! = \begin{cases} 1 & \text{se} \quad n = 0 \\ \prod_{i=1}^{n} i & \text{se} \quad n > 0 \end{cases}$$

F(n)

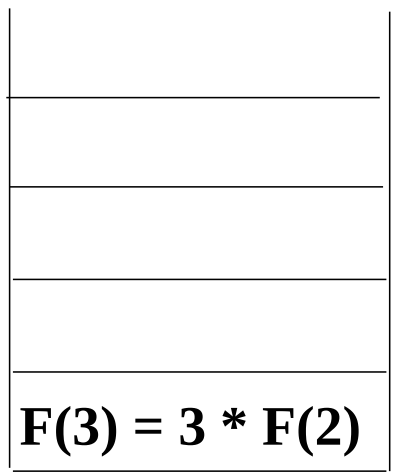
```
if n = 0 then
    return 1
end if
Result ← 1
for i ← 2 to n do
    Result ← Result * i
end for
return Result
```

Fattoriale – Versione ricorsiva

$$n! = \begin{cases} 1 & \text{se} \quad n = 0 \\ n(n-1)! & \text{se} \quad n > 0 \end{cases}$$

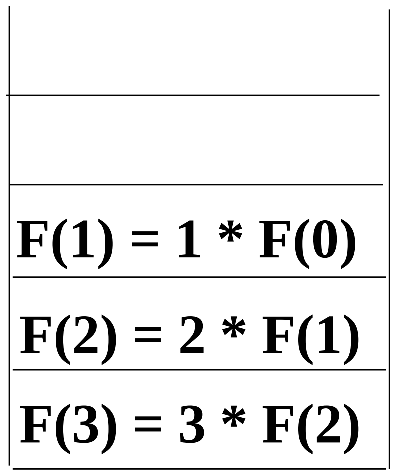
F(n)

```
if n = 0 then
    return 1
else
    return n * F(n − 1)
end if
```

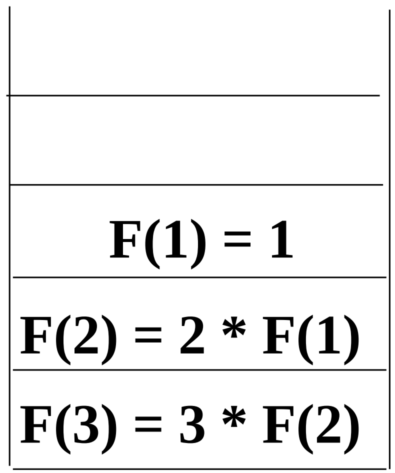

F(3) = 3 * F(2)

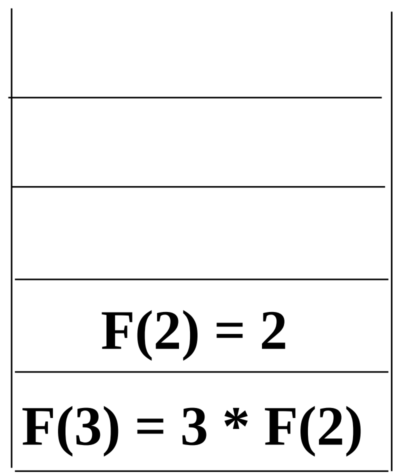
F(2) = 2 * F(1)

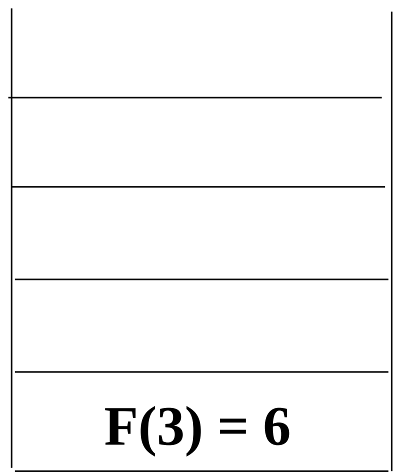
F(3) = 3 * F(2)


F(1) = 1 * F(0)
F(2) = 2 * F(1)
F(3) = 3 * F(2)

$F(0) = 1$
$F(1) = 1 * F(0)$
$F(2) = 2 * F(1)$
$F(3) = 3 * F(2)$


F(1) = 1
F(2) = 2 * F(1)
F(3) = 3 * F(2)


F(2) = 2
F(3) = 3 * F(2)


F(3) = 6