Algoritmi e Strutture di Dati

Ricorsione e complessità

m.patrignani

Sommario

- funzioni e record di attivazione
- ricorsione e record di attivazione
- formule di ricorrenza
  - teorema dell'esperto
- strategie algoritmiche
  - algoritmi divide et impera e merge sort

Effetti di una chiamata a funzione

```
SUM_OF_FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.      sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.      f = f * i
8.  return f
```

- supponiamo di eseguire SUM_OF_FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

SUM_OF_FACT(3)

istruzione	3
variabile sum	0
variabile i	0

Effetti di una chiamata a funzione

```
SUM_OF_FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.       sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.       f = f * i
8.  return f
```

- supponiamo di eseguire SUM_OF_FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT(0)

istruzione	5
variabile f	1
variabile i	

SUM_OF_FACT(3)

istruzione	3
variabile sum	0
variabile i	0

Effetti di una chiamata a funzione

```
SUM_OF_FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.        sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.        f = f * i
8.  return f
```

- supponiamo di eseguire `SUM_OF_FACT(3)`
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT(0)

~~istruzione~~	~~8~~
~~variabile f~~	~~1~~
~~variabile i~~	~~2~~

SUM_OF_FACT(3)

istruzione	3
variabile sum	0
variabile i	0

Effetti di una chiamata a funzione

```
SUM_OF_FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.        sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.        f = f * i
8.  return f
```

- supponiamo di eseguire SUM_OF_FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

SUM_OF_FACT(3)

istruzione	3
variabile sum	1
variabile i	0

Effetti di una chiamata a funzione

```
SUM_OF_FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.        sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.        f = f * i
8.  return f
```

- supponiamo di eseguire SUM_OF_FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

SUM_OF_FACT(3)

istruzione	2
variabile sum	1
variabile i	1

Effetti di una chiamata a funzione

```
SUM_OF_FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.      sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.      f = f * i
8.  return f
```

- supponiamo di eseguire SUM_OF_FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT(1)

~~istruzione~~	~~8~~
~~variabile f~~	~~1~~
~~variabile i~~	~~2~~

SUM_OF_FACT(3)

istruzione	3
variabile sum	1
variabile i	1

Effetti di una chiamata a funzione

```
SUM_OF_FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.      sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.      f = f * i
8.  return f
```

- supponiamo di eseguire `SUM_OF_FACT`(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

SUM_OF_FACT(3)

istruzione	3
variabile sum	2
variabile i	1

Effetti di una chiamata a funzione

```
SUM_OF_FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.        sum = sum + FACT(i)
4. return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.        f = f * i
8.  return f
```

- supponiamo di eseguire SUM_OF_FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

SUM_OF_FACT(3)

istruzione	2
variabile sum	2
variabile i	2

Effetti di una chiamata a funzione

```
SUM_OF_FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.        sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.        f = f * i
8.  return f
```

- supponiamo di eseguire SUM_OF_FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT(2)

istruzione	8
variabile f	2
variabile i	3

SUM_OF_FACT(3)

istruzione	3
variabile sum	2
variabile i	2

Effetti di una chiamata a funzione

```
SUM_OF_FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.       sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.       f = f * i
8.  return f
```

- supponiamo di eseguire SUM_OF_FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

SUM_OF_FACT(3)

istruzione	3
variabile sum	4
variabile i	2

Effetti di una chiamata a funzione

```
SUM_OF_FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.       sum = sum + FACT(i)
4. return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.       f = f * i
8.  return f
```

- supponiamo di eseguire `SUM_OF_FACT`(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

SUM_OF_FACT(3)

istruzione	2
variabile sum	4
variabile i	3

Effetti di una chiamata a funzione

```
SUM_OF_FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.       sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.       f = f * i
8.  return f
```

- supponiamo di eseguire SUM_OF_FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT(3)

istruzione	8
variabile f	6
variabile i	4

SUM_OF_FACT(3)

istruzione	3
variabile sum	4
variabile i	3

Effetti di una chiamata a funzione

```
SUM_OF_FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.        sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.        f = f * i
8.  return f
```

- supponiamo di eseguire `SUM_OF_FACT`(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

SUM_OF_FACT(3)

istruzione	3
variabile sum	10
variabile i	3

Effetti di una chiamata a funzione

```
SUM_OF_FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.        sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.        f = f * i
8.  return f
```

- supponiamo di eseguire `SUM_OF_FACT`(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

SUM_OF_FACT(3)

istruzione	2
variabile sum	10
variabile i	4

Effetti di una chiamata a funzione

```
SUM_OF_FACT(n)
1.  sum = 0
2.  for i = 0 to n
3.          sum = sum + FACT(i)
4.  return sum
```

```
FACT(n)
5.  f = 1
6.  for i = 2 to n
7.          f = f * i
8.  return f
```

- supponiamo di eseguire SUM_OF_FACT(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

SUM_OF_FACT(3)

istruzione	4
variabile sum	10
variabile i	4

Funzioni ricorsive

```
FACT(n)
f = 1
for i = 2 to n
    f = f * i
return f
```

```
FACT_RIC(n)
if n == 0
    f = 1
else
    f = n * FACT_RIC(n-1)
return f
```

- abbiamo già visto che l'algoritmo iterativo FACT per il calcolo del fattoriale ha complessità $\Theta(n)$
- il calcolo del fattoriale può essere facilmente realizzato anche tramite un algoritmo ricorsivo

Esecuzione di funzioni ricorsive

```
FACT_RIC(n)
1.  if n == 0
2.      f = 1
3.  else
4.      f = n * FACT_RIC(n-1)
5.  return f
```

- supponiamo di eseguire `FACT_RIC(3)`
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT_RIC(3)

istruzione	4
variabile f	0

Esecuzione di funzioni ricorsive

```
FACT_RIC(n)
1.  if n == 0
2.      f = 1
3.  else
4.      f = n * FACT-RIC(n-1)
5.  return f
```

- supponiamo di eseguire `FACT_RIC`(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT_RIC(2)

istruzione	4
variabile f	0

FACT_RIC(3)

istruzione	4
variabile f	0

Esecuzione di funzioni ricorsive

```
FACT_RIC(n)
1.  if n == 0
2.      f = 1
3.  else
4.      f = n * FACT-RIC(n-1)
5.  return f
```

- supponiamo di eseguire `FACT_RIC`(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT_RIC(1)

istruzione	4
variabile f	0

FACT_RIC(2)

istruzione	4
variabile f	0

FACT_RIC(3)

istruzione	4
variabile f	0

Esecuzione di funzioni ricorsive

```
FACT_RIC(n)
1.  if n == 0
2.      f = 1
3.  else
4.      f = n * FACT-RIC(n-1)
5.  return f
```

- supponiamo di eseguire `FACT_RIC`(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT_RIC(0)

istruzione	2
variabile f	1

FACT_RIC(1)

istruzione	4
variabile f	0

FACT_RIC(2)

istruzione	4
variabile f	0

FACT_RIC(3)

istruzione	4
variabile f	0

Esecuzione di funzioni ricorsive

```
FACT_RIC(n)
1.  if n == 0
2.      f = 1
3.  else
4.      f = n * FACT-RIC(n-1)
5.  return f
```

- supponiamo di eseguire `FACT_RIC`(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT_RIC(1)

~~istruzione~~	~~4~~
~~variabile f~~	~~1~~

FACT_RIC(2)

istruzione	4
variabile f	0

FACT_RIC(3)

istruzione	4
variabile f	0

Esecuzione di funzioni ricorsive

```
FACT_RIC(n)
1.  if n == 0
2.       f = 1
3.  else
4.       f = n * FACT-RIC(n-1)
5.  return f
```

- supponiamo di eseguire `FACT_RIC`(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

FACT_RIC(2)

~~istruzione~~	~~4~~
~~variabile f~~	~~2~~

FACT_RIC(3)

istruzione	4
variabile f	0

Esecuzione di funzioni ricorsive

```
FACT_RIC(n)
1.  if n == 0
2.      f = 1
3.  else
4.      f = n * FACT-RIC(n-1)
5.  return f
```

- supponiamo di eseguire `FACT_RIC`(3)
- seguiamo l'evoluzione dello stack dei record di attivazione

`FACT_RIC(3)`

~~istruzione~~	~~4~~
~~variabile f~~	~~6~~

Costo di FACT_RIC

```
FACT_RIC(n)
1.  if n == 0
2.      f = 1
3.  else
4.      f = n * FACT-RIC(n-1)
5.  return f
```

$T(0) = \Theta(1)$

$T(n) = T(n\text{-}1) + \Theta(1)$

- il costo di FACT_RIC(n) è
  - $\Theta(1)$ quando n è zero
  - pari al costo di FACT_RIC(n-1) + $\Theta(1)$ negli altri casi

Formule di ricorrenza

- equazioni o disequazioni che descrivono una funzione in termini del suo valore su input più piccoli
  - prevedono sempre dei casi base e dei casi induttivi
- esempi

$$T(n) = \begin{cases} a & \text{per } n = 0 \\ T(n\text{-}1) + g(n) & \text{per } n > 0 \end{cases}$$

$$T(n) = \begin{cases} a & \text{per } n = 0 \text{ o } n = 1 \\ 2T(n/2) + f(n) & \text{per } n > 1 \end{cases}$$

Formule di ricorrenza

- le soluzioni delle formule di ricorrenza non sempre sono facili da trovare
- quando esprimono delle complessità asintotiche talvolta i casi base vengono omessi
  - se $T(n)$ esprime il tempo di esecuzione di un algoritmo, $T(n)$ è sempre $\Theta(1)$ per n piccolo
- esempio

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

  - è sottointeso che $T(n) = \Theta(1)$ per n = 0 e n = 1

Soluzione di una equazione di ricorrenza

- dimostriamo che l'equazione di ricorrenza

$$T(n) = \begin{cases} a & \text{per } n = 0 \\ T(n\text{-}1) + g(n) & \text{per } n > 0 \end{cases}$$

- ammette come soluzione

$$T(n) = a + \sum_{k=1}^{n} g(k)$$

- per dimostrarlo sostituiamo la soluzione proposta a destra e sinistra dell'equazione di ricorrenza

Verifica della correttezza della soluzione

- caso base per n=0

$$T(n=0) = a + \sum_{k=1}^{0} g(k) = a + 0 = a \qquad \text{(verificato)}$$

- caso induttivo

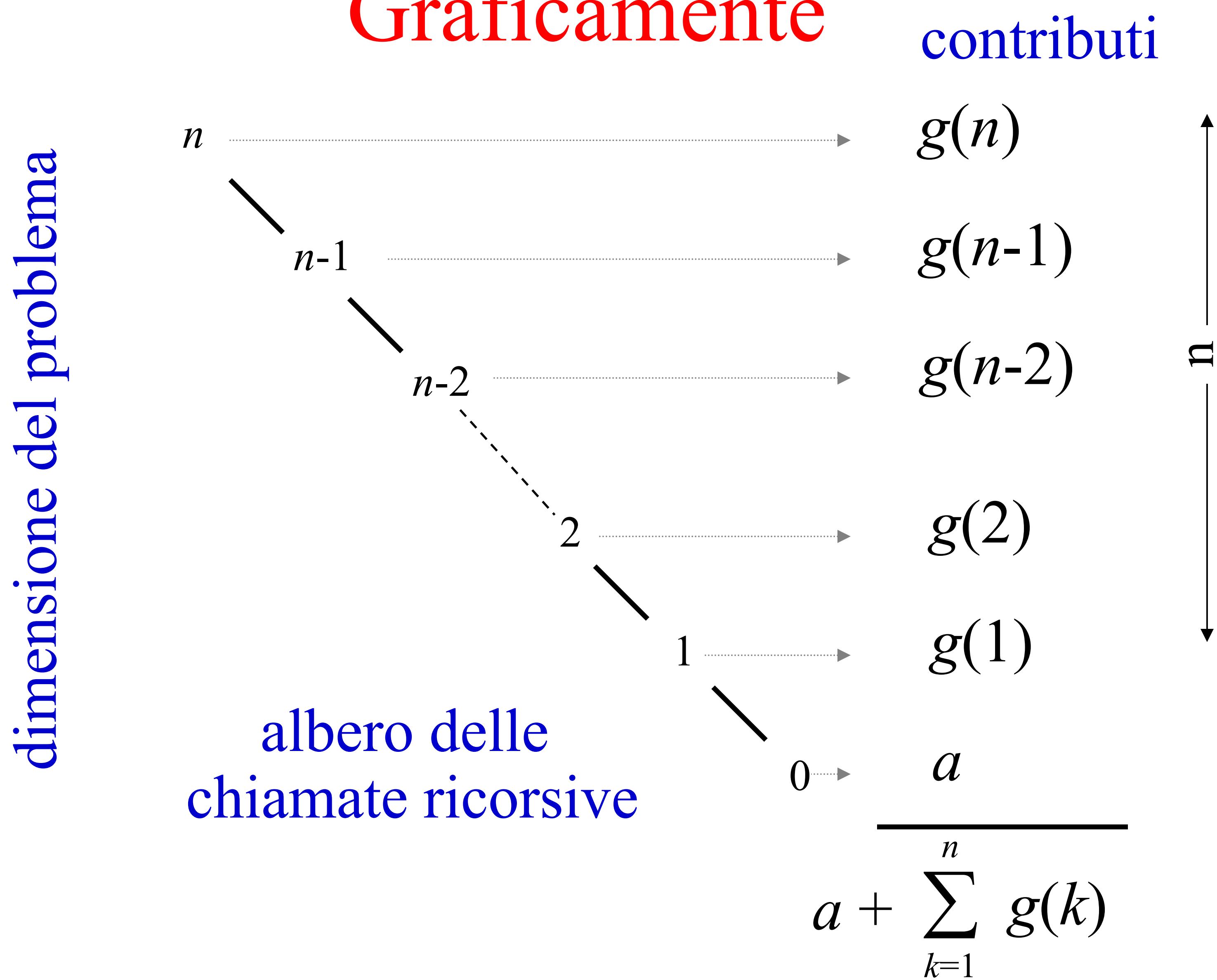
so che $$T(n\text{-}1) = a + \sum_{k=1}^{n-1} g(k) \qquad \text{(ipotesi induttiva)}$$

$$T(n) = T(n\text{-}1) + g(n) \qquad \text{(dalla definizione)}$$

$$T(n) = a + \sum_{k=1}^{n-1} g(k) + g(n)$$

$$T(n) = a + \sum_{k=1}^{n} g(k) \qquad \text{(verificato)}$$

Graficamente



Complessità di `FACT_RIC`

- sappiamo che `FACT_RIC` ha complessità

$$T(n) = \begin{cases} \Theta(1) & \text{per } n = 0 \\ T(n\text{-}1) + \Theta(1) & \text{per } n > 0 \end{cases}$$

- sappiamo che l'equazione di ricorrenza

$$T(n) = \begin{cases} a & \text{per } n = 0 \\ T(n\text{-}1) + g(n) & \text{per } n > 0 \end{cases}$$

- ammette come soluzione $T(n) = a + \sum_{k=1}^{n} g(k)$
- la complessità di `FACT_RIC` è dunque

$$T(n) = \Theta(1) + \sum_{k=1}^{n} \Theta(1) = \Theta(n)$$

Versione ricorsiva del selection sort

```
SELECTION(A)                    ▷ ordina A
   SELECTION_RIC(A,0)           ▷ ordina A da 0 in poi
```

```
SELECTION_RIC(A,i)              ▷ ordina A da i a A.length-1
   if i < A.length-1            ▷ altrimenti è già ordinato
      min = i          ▷ indice elemento minimo in A[i..n-1]
      for j = i + 1 to A.length-1     ▷ scorro l'array
          if A[j] < A[min]            ▷ devo aggiornare min
             min = j
      temp = A[i]                  ▷ scambio A[i] con A[min]
      A[i] = A[min]
      A[min] = temp
      SELECTION-RIC(A,i+1)
```


Complessità di `SELECTION_RIC`

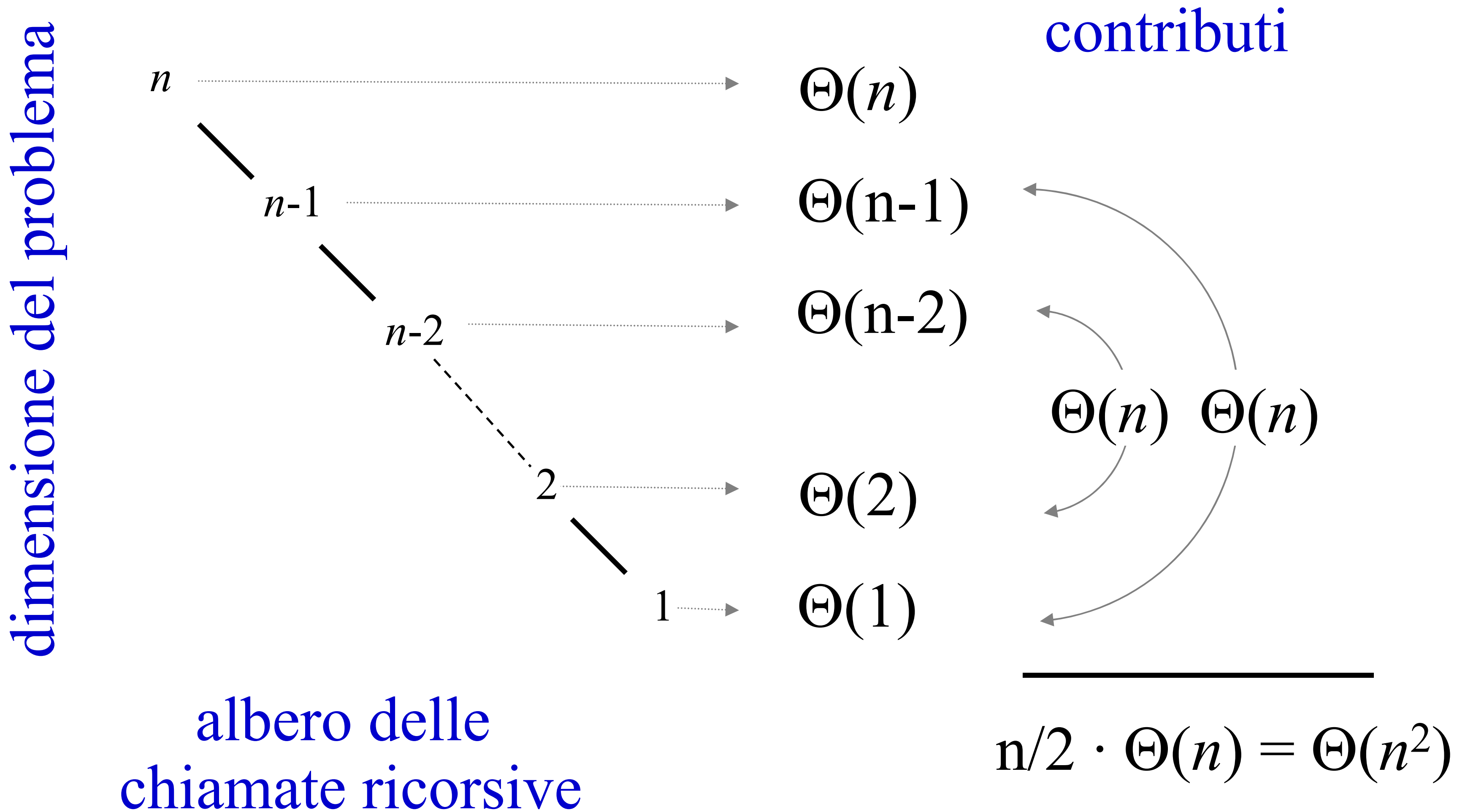
- possiamo scrivere la seguente equazione di ricorrenza, in cui n è il numero degli elementi di A ancora da ordinare

$$T(n) = \begin{cases} \Theta(1) & \text{per } n = 1 \\ T(n\text{-}1) + \Theta(n) & \text{per } n > 1 \end{cases}$$

- la complessità di `SELECTION_RIC` è dunque

$$T(n) = \Theta(1) + \sum_{k=1}^{n} \Theta(k) = \Theta(n^2)$$

Graficamente



La tecnica divide et impera

- detta anche “divide and conquer”
- consiste nel suddividere il problema in diversi sottoproblemi
  - i sottoproblemi sono dello stesso tipo del problema originale
    - ma di dimensioni più piccole
  - i sottoproblemi possono essere risolti in maniera ricorsiva
    - suddividendoli a loro volta
  - caso base
    - quando i sottoproblemi sono di dimensioni ridottissime la loro soluzione è banale

Ricorsione del divide et impera

- a ciascun passo della ricorsione
  - divide
    - l’istanza corrente viene divisa in due o più istanze più piccole
  - impera
    - l’algoritmo viene lanciato sulle istanze più piccole
  - combina
    - le soluzioni delle istanze più piccole vengono utilizzate per produrre una soluzione dell’istanza corrente

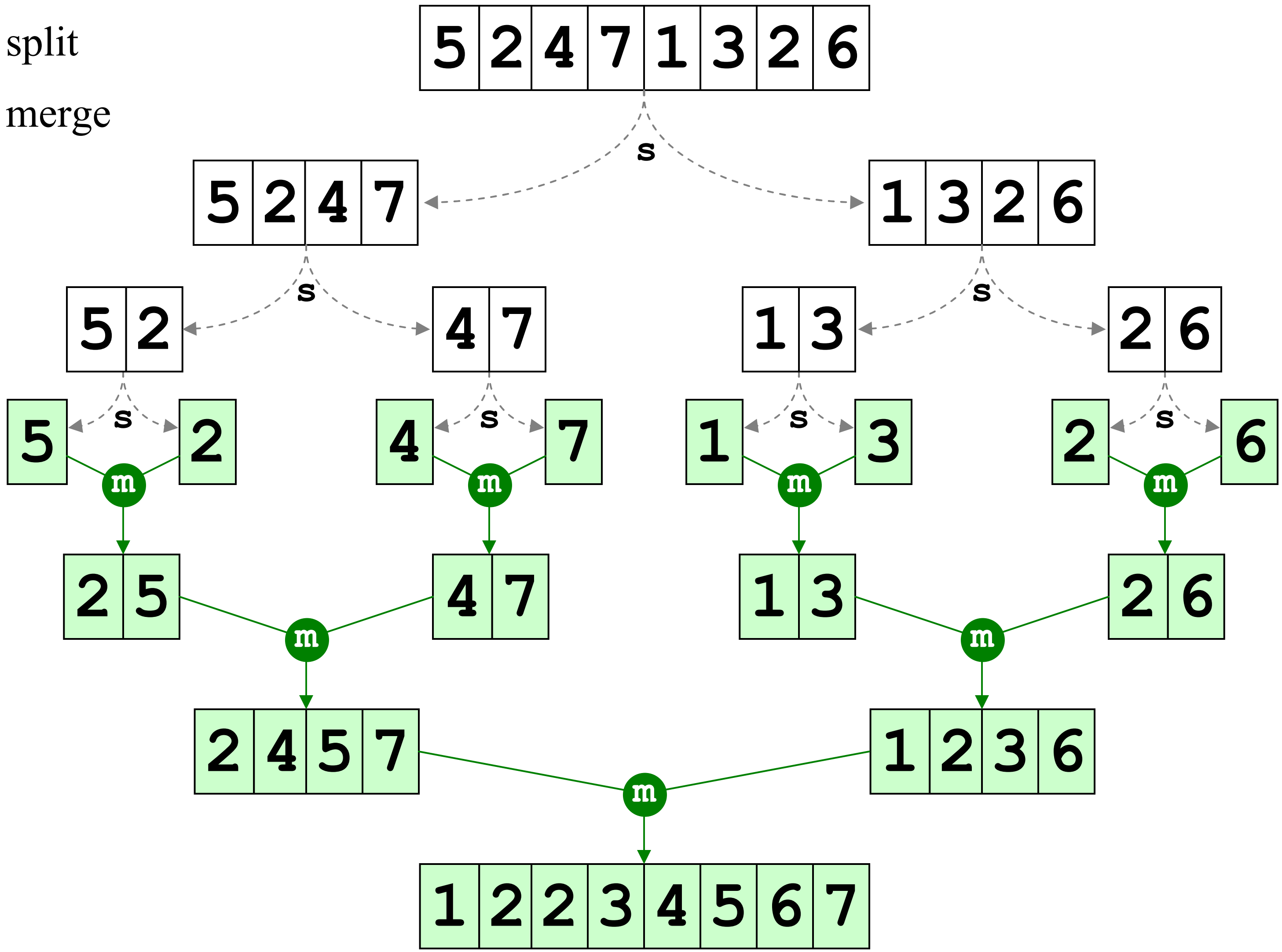
Merge sort

- introdotto da John von Neumann nel 1945
- osservazione elementare
  – due sequenze ordinate possono essere fuse in un’unica sequenza ordinata molto facilmente
- un possibile algoritmo
  – dividere la sequenza di input in due sottosequenze
  – ordinare le due sottosequenze
    - tramite lo stesso merge sort
  – fondere le due sottosequenze ordinate
- caso base
  – un array di un solo elemento è ordinato per definizione

Merge sort con input 5 2 4 7 1 3 2 6

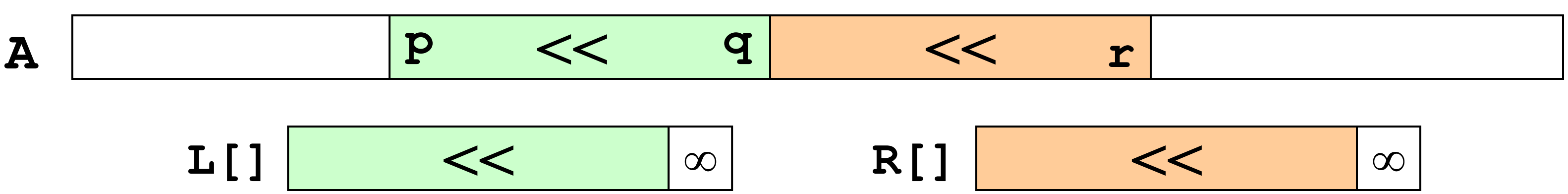
s = split

m = merge



Fusione: l'algoritmo MERGE

```
MERGE(A,p,q,r)
n1 = q - p + 1          ▷ lunghezza della prima sequenza
n2 = r - q              ▷ lunghezza della seconda sequenza
        ▷ creo array L[0…n1] e R[0…n2] (con una casella in +)
for i = 0 to n1-1
        L[i] = A[p+i]       ▷ copio la 1a sequenza
for j = 0 to n2-1
        R[j] = A[q+j+1]     ▷ copio la 2a sequenza
L[n1] = ∞       ▷ chiudo con "infinito"
R[n2] = ∞       ▷ chiudo con "infinito"
                …(continua nella prossima slide)…
```



Fusione (continua)

```
…(dalla slide precedente)…
10. i = 0                ▷ iteratore per array L
11. j = 0                ▷ iteratore per array R
12. for k = p to r
13.     if L[i] ≤ R[j] then
14.         A[k] = L[i]        ▷ pesco da L
15.         i = i + 1
16.     else
17.         A[k] = R[j]        ▷ pesco da R
18.         j = j + 1
```

- il confronto con "≤" sulla riga 13 garantisce la stabilità dell'algoritmo
  - se `L[i]=R[j]` allora `L[i]` ha la precedenza

L'algoritmo MERGE_SORT

- l'algoritmo MERGE_SORT esegue la parte "divide", risolve i sottoproblemi ed esegue la parte "combine"

```
MERGE_SORT(A,p,r)
1.  if p < r then             ▷ nel caso base esco subito
2.         q = ⌊(p+r)/2⌋      ▷ divido l'array in due
3.         MERGE_SORT(A,p,q)
4.         MERGE_SORT(A,q+1,r)
5.         MERGE(A,p,q,r)
```

- all'inizio della computazione lanciamo

```
MERGE(A)                ▷ ordina A
1.  MERGE_SORT(A,0,A.length-1)
```


Tempo di esecuzione di merge sort

- calcoliamo il costo $T(n)$ di esecuzione del merge sort su un'istanza con n elementi
- caso base
  - costo $\Theta(1)$
- divide
  - calcolo di $n/2$: costo $D(n) = \Theta(1)$
- impera
  - ogni sottoproblema ha dimensione $n/2$
  - i sottoproblemi sono 2
  - costo: $2 \cdot T(n/2)$
- combina
  - l'algoritmo `MERGE` ha costo lineare: $C(n) = \Theta(n)$

Tempo di esecuzione di merge sort

- complessivamente

$$T(n) = \begin{cases} \Theta(1) & \text{per } n = 0 \text{ o } n = 1 \\ 2 \cdot T(n/2) + D(n) + C(n) & \text{per } n > 1 \end{cases}$$

- poiché $D(n) + C(n) = \Theta(1) + \Theta(n) = \Theta(n)$ si ha

$$T(n) = \begin{cases} \Theta(1) & \text{per } n = 0 \text{ o } n = 1 \\ 2 \cdot T(n/2) + \Theta(n) & \text{per } n > 1 \end{cases}$$

- dimostreremo che questa particolare equazione di ricorrenza ammette come soluzione

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

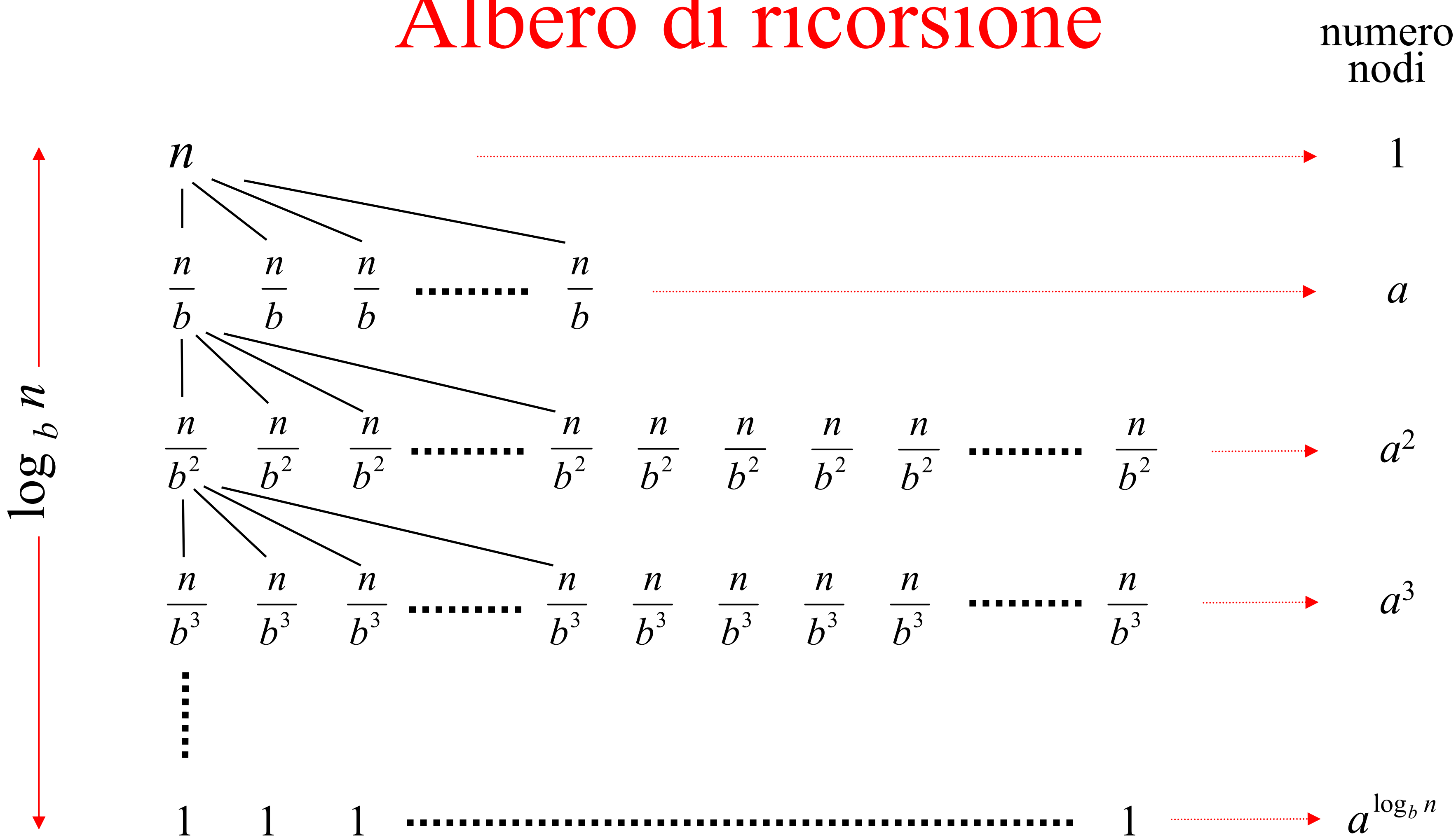
Master theorem (teorema dell'esperto)

- siano $a, b \geq 1$
- il master theorem considera l'equazione di ricorrenza seguente

$$T(n) = \begin{cases} \Theta(1) & \text{per } n = 0 \\ a \cdot T(n/\mathrm{b}) + O(n^k) & \text{per } n > 0 \end{cases}$$

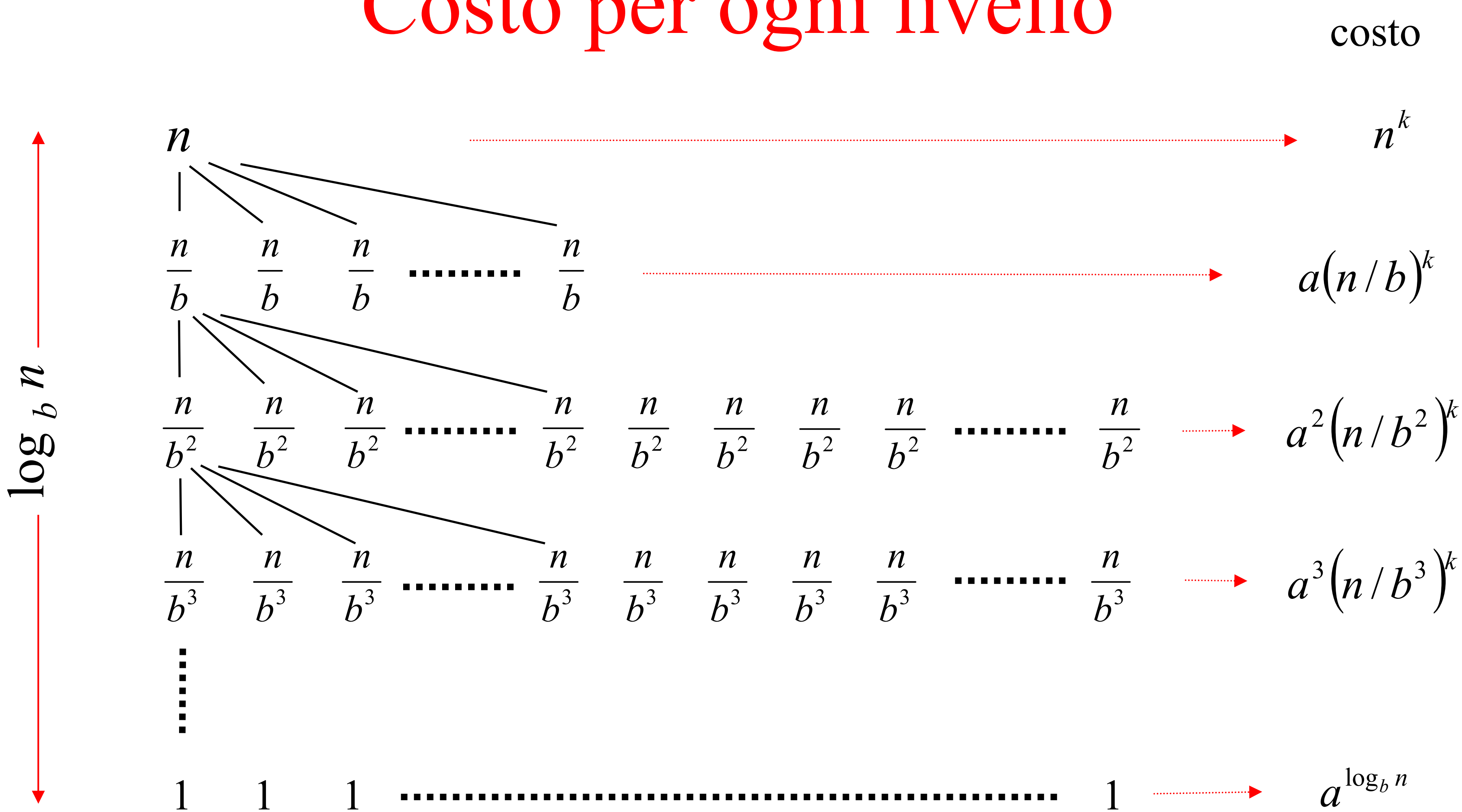
- il master theorem afferma che tale equazione di ricorrenza ammette le soluzioni seguenti
  1. se $a < b^k$ allora $T(n) = \Theta(n^k)$
  2. se $a = b^k$ allora $T(n) = \Theta(n^k \log n)$
  3. se $a > b^k$ allora $T(n) = \Theta(n^{\log_b a})$

Albero di ricorsione



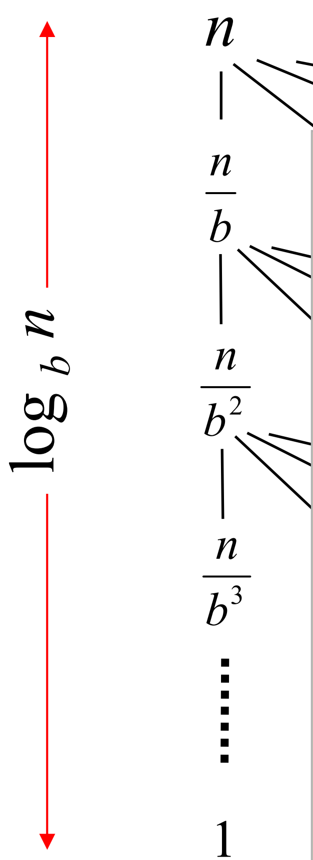
$$n/b^h = 1 \quad \Rightarrow \quad n = b^h \quad \Rightarrow \quad h = \log_b n$$

Costo per ogni livello



Costo per ogni livello

costo



n^k

$a(n/b)^k$

$a^2(n/b^2)^k$

$a^3(n/b^3)^k$

$a^{\log_b n}$

Master Theorem:

$$T(n) = a \cdot T(n/b) + O(n^k)$$

1. se $a < b^k$ allora $T(n) = O(n^k)$

In pratica se $a < b^k$ la complessità è dominata dal primo livello dell'albero di ricorsione

Costo per ogni livello

costo

$\log_b n$

	costo
n	n^k
$\frac{n}{b}$	$a(n/b)^k$
$\frac{n}{b^2}$	$a^2(n/b^2)^k$
$\frac{n}{b^3}$	$a^3(n/b^3)^k$
⋮	
1	$a^{\log_b n}$

Master Theorem:

$$T(n) = a \cdot T(n/b) + O(n^k)$$

2. se $a = b^k$ allora $T(n) = O(n^k \log n)$

Se $a = b^k$ la complessità è uguale per tutti i livelli

- infatti $a^i(n/b^i)^k = b^{ki}(n/b^i)^k = n^k$

Costo per ogni livello

costo

$n \quad\rightarrow\quad n^k$

$\dfrac{n}{b} \quad\rightarrow\quad a(n/b)^k$

$\dfrac{n}{b^2} \quad\rightarrow\quad a^2(n/b^2)^k$

$\dfrac{n}{b^3} \quad\rightarrow\quad a^3(n/b^3)^k$

$\vdots$

$1 \quad\rightarrow\quad a^{\log_b n}$

$\log_b n$

Master Theorem:

$$T(n) = a \cdot T(n/b) + O(n^k)$$

3. se $a > b^k$ allora $T(n) = O(n^{\log_b a})$

Se $a > b^k$ la complessità è dominata dall'ultimo livello
- infatti $a^{\log_b n} = n^{\log_b a}$

Dimostriamo che $x^{\log y} = y^{\log x}$

- Partiamo da

$$x^{\log y} = y^{\log x}$$

- Facciamo il logaritmo da entrambe le parti

$$\log(x^{\log y}) = \log(y^{\log x})$$

- Ricordando che

$$\log a^b = b \log a$$

- Otteniamo

$$(\log y)(\log x) = (\log x)(\log y)$$

- Che è vera per la proprietà commutativa del prodotto

Dimostrazione del primo caso ($a < b^k$)

- La somma del costo di tutti i livelli è

$$T(n) = \sum_{i=0}^{h} a^i \left(\frac{n}{b^i}\right)^k = \sum_{i=0}^{h} a^i \frac{n^k}{b^{ik}} = n^k \sum_{i=0}^{h} \left(\frac{a}{b^k}\right)^i$$

- $\sum_{i=0}^{h} \left(\frac{a}{b^k}\right)^i$ è una serie geometrica con ragione $r = \frac{a}{b^k}$
- Se $r < 1$, cioè se $a < b^k$, la sommatoria, anche se avesse infiniti termini, sarebbe comunque una costante $1/(1-r)$
- Dunque $T(n) = O(n^k)$

Dimostrazione del terzo caso ($a > b^k$)

- Torniamo alla serie geometrica con $r = \dfrac{a}{b^k}$

$$T(n) = n^k \sum_{i=0}^{h} \left(\frac{a}{b^k} \right)^i$$

- Se $a > b^k$, cioè se $r > 1$, la sommatoria vale

$$\frac{1-r^h}{1-r} = \frac{r^h - 1}{r-1} \in O\left(r^h\right)$$

$$r^h = \left(\frac{a}{b^k} \right)^{\log_b n} = \frac{a^{\log_b n}}{b^{k \log_b n}} = \frac{a^{\log_b n}}{(b^{\log_b n})^k} = \frac{a^{\log_b n}}{n^k}$$

Dimostrazione del terzo caso ($a > b^k$)

- Dunque

$$T(n) = O(n^k) \cdot O(r^h) = O(n^k) \cdot O\left(\frac{a^{\log_b n}}{n^k}\right)$$

- Da cui

$$T(n) = O\left(a^{\log_b n}\right) = O\left(n^{\log_b a}\right)$$

Esempi di applicazione del master theorem

- $T(n) = 9T(n/3) + n$
  - abbiamo: $a = 9$; $b = 3$; $p(n^k) = n$; $k = 1$
  - quindi $a > b^k$
  - si ha $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_3 9}) = \Theta(n^2)$
- $T(n) = T(2n/3) + 1$
  - abbiamo: $a = 1$; $b = 3/2$; $p(n^k) = 1$; $k = 0$
  - quindi $a = b^k$
  - si ha $T(n) = \Theta(n^k \log n) = \Theta(n^0 \log n) = \Theta(\log n)$

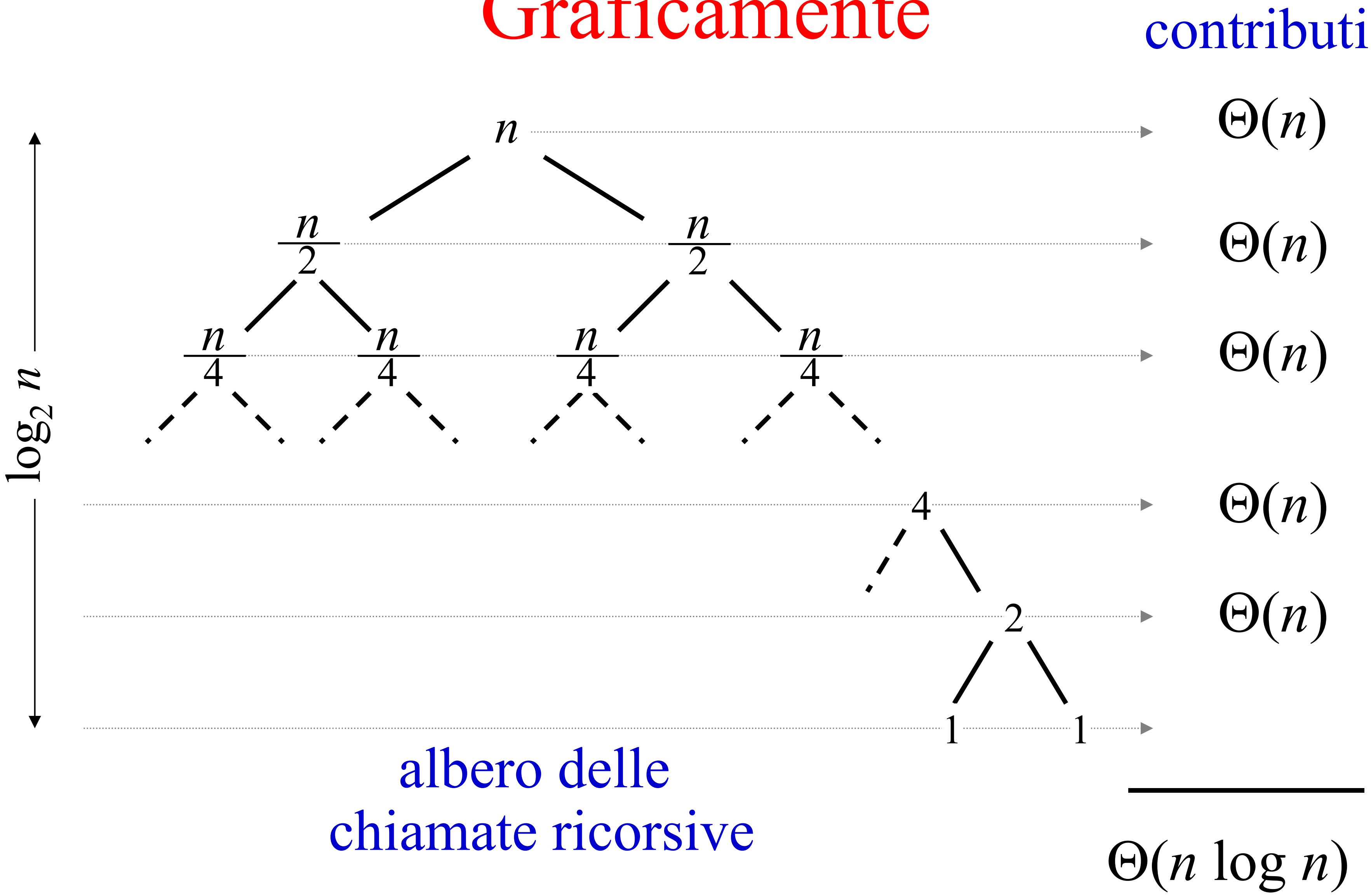
Complessità del merge sort

- la complessità del merge sort è data dalla formula di ricorrenza

  $T(n) = 2 \cdot T(n/2) + \Theta(n)$

- applichiamo il teorema dell'esperto
  - abbiamo: $a = 2$; $b = 2$; $p(n^k) = n$; $k = 1$
  - quindi $a = b^k$
  - si ha $T(n) = \Theta(n^k \log n) = \Theta(n \log n)$

Graficamente



Algoritmi di ordinamento visti finora

	caso migliore	caso medio	caso peggiore	in loco	stabile
SELECTION-SORT	$\Theta(n^2)$			*si*	*si*
INSERTION-SORT	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	*si*	*si*
MERGE-SORT	$\Theta(n \log n)$			*no*	*si*

