



SAPIENZA
UNIVERSITÀ DI ROMA

Algoritmi

Esercizi 5 — Ricorsione

Esercizi svolti sui capitoli 14–18: ricorsione, equazioni di ricorrenza, Master theorem, algoritmi ricorsivi, ricorsivo e iterativo a confronto. Ogni esercizio ha la soluzione completa.

Fabio Furini

fabiofurini.github.io

Dipartimento di Ingegneria informatica, automatica e gestionale — Sapienza Università di Roma

Versione online e materiale scaricabile:

fabiofurini.github.io/algoritmi/
github.com/fabiofurini/algoritmi

Anno accademico 2026–2027

Indice

14	Ricorsione	2
15	Equazioni di ricorrenza	19
16	Il Master theorem	34
17	Algoritmi ricorsivi	48
18	Algoritmi ricorsivi e iterativi a confronto	88



Ricorsione

Negli esercizi di questo capitolo i passi di un algoritmo ricorsivo si contano con le convenzioni dei capitoli «Tempo di esecuzione» e «Ricorsione», e le equazioni di ricorrenza si risolvono per semplice srotolamento.

- Ogni esecuzione di una riga numerata di pseudocodice è **un passo**; una riga che contiene una chiamata ricorsiva conta come un passo **più** tutti i passi eseguiti da quella chiamata.
- $C(n)$ indica il numero di **chiamate** (nodi dell'albero delle chiamate ricorsive) e $S(n)$ il numero di **passi**; la **profondità della ricorsione** è il massimo numero di nodi su un cammino dalla radice a una foglia dell'albero, cioè il massimo numero di stack frame simultaneamente in memoria.
- **Induzione forte**: se $P(n_0)$ è vera e, per ogni $n > n_0$, la verità di $P(k)$ per ogni k con $n_0 \leq k < n$ implica la verità di $P(n)$, allora $P(n)$ è vera per ogni $n \geq n_0$.
- **Srotolamento** di un'equazione di ricorrenza $S(n) = S(n-1) + c$ con c costante:

$$S(n) = S(n-1) + c = S(n-2) + 2c = \dots = S(n-k) + kc,$$

e il caso base si raggiunge per il valore di k che rende $n-k$ un caso base.

Esercizio 14.1 — Ricorsione (E1)

Si consideri il seguente algoritmo ricorsivo.

Algorithm: Mystery**Input** : Un numero naturale $n \geq 1$ **Output**: Un numero naturale

```

1 function Mystery( $n$ )
2   if  $n = 1$  then
3     return 1
4   end
5   return Mystery( $\lfloor n/2 \rfloor$ ) + Mystery( $\lceil n/2 \rceil$ )
6 end

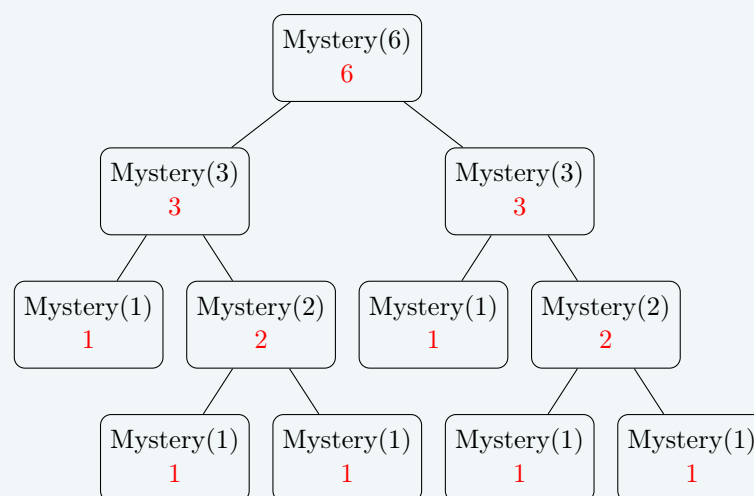
```

Domande

1. Disegnare l'albero delle chiamate ricorsive di **Mystery**(6), scrivendo accanto a ogni chiamata il valore che restituisce.
2. Che cosa calcola l'algoritmo? Dimostrare l'affermazione per induzione forte su n .
3. Sia $C(n)$ il numero di chiamate eseguite da **Mystery**(n). Scrivere un'equazione di ricorrenza per $C(n)$ e dimostrare che $C(n) = 2n - 1$ per ogni $n \geq 1$. Verificare la formula sull'albero della domanda 1.
4. Qual è la profondità della ricorsione di **Mystery**(6), cioè il massimo numero di stack frame simultaneamente in memoria?

Soluzione

1. Si ha $\lfloor 6/2 \rfloor = \lceil 6/2 \rceil = 3$, $\lfloor 3/2 \rfloor = 1$, $\lceil 3/2 \rceil = 2$ e $\lfloor 2/2 \rfloor = \lceil 2/2 \rceil = 1$. L'albero delle chiamate ricorsive (valori restituiti in rosso) è:



`Mystery(2)` restituisce $1 + 1 = 2$, `Mystery(3)` restituisce $1 + 2 = 3$ e `Mystery(6)` restituisce $3 + 3 = 6$.

2. L'algoritmo restituisce n stesso (calcola n come somma di n uni).

Dimostrazione per induzione forte su $n \geq 1$.

- **Caso base** ($n = 1$): l'algoritmo restituisce 1 alla riga 3.
- **Passo induttivo** ($n \geq 2$): supponiamo che `Mystery(k)` restituisca k per ogni $1 \leq k < n$. Per $n \geq 2$ si ha $1 \leq \lfloor n/2 \rfloor \leq \lceil n/2 \rceil < n$, quindi per ipotesi induttiva la riga 5 restituisce

$$\lfloor n/2 \rfloor + \lceil n/2 \rceil = n.$$

L'induzione forte è necessaria perché le chiamate ricorsive sono su $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$, non su $n - 1$.

3. Ogni chiamata con $n \geq 2$ effettua due chiamate ricorsive, quindi

$$C(n) = \begin{cases} 1 & \text{se } n = 1 \\ 1 + C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) & \text{se } n \geq 2 \end{cases}$$

Dimostrazione di $C(n) = 2n - 1$ per induzione forte su n .

- **Caso base** ($n = 1$): $C(1) = 1 = 2 \cdot 1 - 1$.
- **Passo induttivo** ($n \geq 2$): per ipotesi induttiva,

$$C(n) = 1 + (2\lfloor n/2 \rfloor - 1) + (2\lceil n/2 \rceil - 1) = 2(\lfloor n/2 \rfloor + \lceil n/2 \rceil) - 1 = 2n - 1.$$

Per $n = 6$: $C(6) = 11$, e infatti l'albero della domanda 1 ha 11 nodi. (L'albero ha $n = 6$ foglie, una per ciascun 1 della somma, e $n - 1 = 5$ nodi interni, uno per ciascuna addizione.)

4. I cammini più lunghi dalla radice a una foglia sono, per esempio, `Mystery(6)`, `Mystery(3)`, `Mystery(2)`, `Mystery(1)`. Quindi la profondità della ricorsione è 4: al più 4 stack frame sono simultaneamente in memoria, benché vengano eseguite 11 chiamate. (In generale la profondità è $\lceil \log_2 n \rceil + 1$.)

Esercizio 14.2 — Ricorsione (E2)

Dato un array $a[1..n]$ di n numeri interi, si considerino i problemi di calcolare

- la **somma** $a[1] + a[2] + \dots + a[n]$ (uguale a 0 se $n = 0$);
- il valore **massimo** $\max\{a[1], a[2], \dots, a[n]\}$ (per $n \geq 1$).

Per le domande numeriche si usi l'array $a = [3, 8, 1, 6]$.

Domande

1. Scrivere un algoritmo ricorsivo $\text{RecSum}(a, n)$ che calcola la somma di $a[1..n]$ riducendo il problema al sotto-array $a[1..n-1]$ (decremento di uno).
2. Scrivere un algoritmo ricorsivo $\text{RecMax}(a, n)$ che calcola il massimo di $a[1..n]$ riducendo il problema al sotto-array $a[1..n-1]$.
3. Mostrare la sequenza di chiamate e di ritorni di $\text{RecSum}(a, 4)$ e di $\text{RecMax}(a, 4)$ per $a = [3, 8, 1, 6]$.
4. Dimostrare la correttezza di RecMax per induzione su n .
5. Scrivere le equazioni di ricorrenza del numero di passi $S(n)$ dei due algoritmi, risolverle per srotolamento e indicare la profondità della ricorsione.

Soluzione

1. La somma di $a[1..n]$ è la somma di $a[1..n-1]$ più $a[n]$, e la somma di un array vuoto è 0:

Algorithm: RecSum

Input : Un array $a[1..n]$ di $n \geq 0$ numeri interi

Output: La somma $a[1] + \dots + a[n]$

```

1 function RecSum( $a, n$ )
2   if  $n = 0$  then
3     return 0
4   end
5   return RecSum( $a, n - 1$ ) +  $a[n]$ 
6 end
```

2. Il massimo di $a[1..n]$ è il maggiore tra il massimo di $a[1..n-1]$ e $a[n]$; il massimo di un singolo elemento $a[1]$ è $a[1]$:

Algorithm: RecMax**Input** : Un array $a[1..n]$ di $n \geq 1$ numeri interi**Output**: Il valore massimo in $a[1..n]$

```

1 function RecMax( $a, n$ )
2   if  $n = 1$  then
3     return  $a[1]$ 
4   end
5    $m := \text{RecMax}(a, n - 1)$ 
6   if  $a[n] > m$  then
7     return  $a[n]$ 
8   end
9   return  $m$ 
10 end

```

3. Per $a = [3, 8, 1, 6]$:**RecSum**($a, 4$): chiamate
$$\text{RecSum}(a, 4) \rightarrow \text{RecSum}(a, 3) \rightarrow \text{RecSum}(a, 2) \rightarrow \text{RecSum}(a, 1) \rightarrow \text{RecSum}(a, 0),$$

poi i ritorni:

chiamata	restituisce
$\text{RecSum}(a, 0)$	0 (caso base)
$\text{RecSum}(a, 1)$	$0 + a[1] = 0 + 3 = 3$
$\text{RecSum}(a, 2)$	$3 + a[2] = 3 + 8 = 11$
$\text{RecSum}(a, 3)$	$11 + a[3] = 11 + 1 = 12$
$\text{RecSum}(a, 4)$	$12 + a[4] = 12 + 6 = 18$

RecMax($a, 4$): chiamate $\text{RecMax}(a, 4) \rightarrow \text{RecMax}(a, 3) \rightarrow \text{RecMax}(a, 2) \rightarrow \text{RecMax}(a, 1)$, poi i ritorni:

chiamata	confronto	restituisce
$\text{RecMax}(a, 1)$	– (caso base)	$a[1] = 3$
$\text{RecMax}(a, 2)$	$m = 3, a[2] = 8 > 3$	8
$\text{RecMax}(a, 3)$	$m = 8, a[3] = 1 \leq 8$	8
$\text{RecMax}(a, 4)$	$m = 8, a[4] = 6 \leq 8$	8

La somma è 18 e il massimo è 8.

4. **Affermazione**: per ogni $n \geq 1$, **RecMax**(a, n) restituisce $\max\{a[1], \dots, a[n]\}$.

- **Caso base** ($n = 1$): l'algoritmo restituisce $a[1] = \max\{a[1]\}$ alla riga 3.

- **Passo induttivo** ($n \geq 2$): per ipotesi induttiva, la riga 5 assegna

$$m = \max\{a[1], \dots, a[n-1]\}.$$

Se $a[n] > m$, allora $a[n]$ è maggiore di tutti gli elementi di $a[1..n-1]$ e l'algoritmo restituisce $a[n] = \max\{a[1], \dots, a[n]\}$ (riga 7). Altrimenti $a[n] \leq m$ e l'algoritmo restituisce $m = \max\{a[1], \dots, a[n]\}$ (riga 9).

5. **RecSum.** $S(0) = 2$ (righe 2–3) e, per $n \geq 1$, le righe 2 e 5 più la chiamata ricorsiva:

$$S(n) = S(n-1) + 2 = S(n-2) + 2 \cdot 2 = \dots = S(0) + 2n = 2n + 2 = \Theta(n).$$

RecMax. $S(1) = 2$ (righe 2–3) e, per $n \geq 2$, le righe 2, 5, 6 e una delle righe 7, 9, più la chiamata ricorsiva:

$$S(n) = S(n-1) + 4 = S(n-2) + 2 \cdot 4 = \dots = S(1) + 4(n-1) = 4n - 2 = \Theta(n).$$

Per $n = 4$: $S(4) = 10$ per **RecSum** e $S(4) = 14$ per **RecMax**.

La profondità della ricorsione è $n+1$ per **RecSum** (chiamate su $n, n-1, \dots, 0$) e n per **RecMax** (chiamate su $n, \dots, 1$): entrambi usano una memoria $\Theta(n)$ per il call stack, mentre le versioni iterative (come **FindMaximum** nel capitolo «Tempo di esecuzione») usano memoria costante.

Se tutti gli elementi usano un numero costante di bit ($|I| = \Theta(n)$), il tempo di esecuzione nel caso peggiore sia di **RecSum** sia di **RecMax** è $T(|I|) = \Theta(|I|)$.

Esercizio 14.3 — Ricorsione (E3)

Un array $a[1..n]$ è un **palindromo** se si legge allo stesso modo da sinistra a destra e da destra a sinistra, cioè $a[i] = a[n+1-i]$ per ogni $i \in \{1, 2, \dots, n\}$. Si consideri il seguente algoritmo ricorsivo, chiamato sull'intero array come **IsPalindrome**($a, 1, n$).

Algorithm: IsPalindrome**Input** : Un array a e due indici l, r **Output:** **true** se il sotto-array $a[l..r]$ è un palindromo, **false** altrimenti

```

1 function IsPalindrome( $a, l, r$ )
2   if  $l \geq r$  then
3     return true
4   end
5   if  $a[l] \neq a[r]$  then
6     return false
7   end
8   return IsPalindrome( $a, l + 1, r - 1$ )
9 end

```

Domande

1. Mostrare la sequenza di chiamate di $\text{IsPalindrome}(a, 1, 5)$ per gli array $a = [1, 2, 3, 2, 1]$ e $a = [1, 2, 3, 3, 1]$.
2. Dimostrare per induzione sulla lunghezza $k = r - l + 1$ del sotto-array che l'algoritmo è corretto.
3. Determinare il numero di passi $S(n)$ nel caso peggiore di $\text{IsPalindrome}(a, 1, n)$ e la sua profondità di ricorsione.
4. L'algoritmo è ricorsivo in coda? In caso affermativo, scrivere l'algoritmo iterativo equivalente.
5. Scrivere un algoritmo ricorsivo $\text{Reverse}(a, l, r)$ che inverte sul posto il sotto-array $a[l..r]$, e mostrarne l'esecuzione su $a = [5, 7, 2, 9, 4, 6]$ con $l = 1, r = 6$.

Soluzione

1. Per $a = [1, 2, 3, 2, 1]$:

chiamata	$a[l]$	$a[r]$	azione
$\text{IsPalindrome}(a, 1, 5)$	1	1	uguali $\Rightarrow$ richiama $\text{IsPalindrome}(a, 2, 4)$
$\text{IsPalindrome}(a, 2, 4)$	2	2	uguali $\Rightarrow$ richiama $\text{IsPalindrome}(a, 3, 3)$
$\text{IsPalindrome}(a, 3, 3)$	–	–	$l \geq r \Rightarrow$ restituisce true

Il valore **true** viene restituito invariato da tutte le chiamate (3 chiamate).

Per $a = [1, 2, 3, 3, 1]$:

chiamata	$a[l]$	$a[r]$	azione
<code>IsPalindrome(a, 1, 5)</code>	1	1	uguali $\Rightarrow$ richiama <code>IsPalindrome(a, 2, 4)</code>
<code>IsPalindrome(a, 2, 4)</code>	2	3	diversi $\Rightarrow$ restituisce false

Viene restituito il valore **false** (2 chiamate).

2. Sia $k = r - l + 1$ la lunghezza di $a[l..r]$ ($k \leq 0$ significa sotto-array vuoto). Dimostriamo per induzione forte su k che l'algoritmo restituisce **true** se e solo se $a[l..r]$ è un palindromo.

- **Casi base** ($k \leq 1$, cioè $l \geq r$): un array vuoto o con un solo elemento è un palindromo, e l'algoritmo restituisce **true** alla riga 3.
- **Passo induttivo** ($k \geq 2$): $a[l..r]$ è un palindromo se e solo se $a[l] = a[r]$ e il sotto-array interno $a[l + 1..r - 1]$ è un palindromo. Se $a[l] \neq a[r]$, l'algoritmo restituisce correttamente **false** (riga 6). Altrimenti restituisce il risultato della chiamata su $a[l + 1..r - 1]$, la cui lunghezza è $k - 2 < k$; per ipotesi induttiva, questo risultato è **true** se e solo se $a[l + 1..r - 1]$ è un palindromo, quindi se e solo se $a[l..r]$ è un palindromo.

La lunghezza diminuisce di 2 a ogni chiamata, ed è per questo che sia $k = 0$ sia $k = 1$ devono essere casi base.

3. Il caso peggiore si ha quando l'array è un palindromo (la ricorsione non viene mai fermata alla riga 6). Ogni chiamata con $l < r$ esegue le righe 2, 5 e 8 (3 passi), e l'ultima chiamata esegue le righe 2-3 (2 passi). In funzione della lunghezza k :

$$S(k) = S(k - 2) + 3 \quad (k \geq 2), \quad S(0) = S(1) = 2.$$

Srotolando, dopo $\lfloor k/2 \rfloor$ passaggi si raggiunge $S(0)$ oppure $S(1)$:

$$S(n) = 3\lfloor n/2 \rfloor + 2 = \Theta(n).$$

L'algoritmo effettua $\lfloor n/2 \rfloor + 1$ chiamate, che è anche la profondità della ricorsione (l'albero è un cammino). Per $n = 5$: 3 chiamate, come nella domanda 1, e $S(5) = 8$.

4. Sì: la chiamata ricorsiva alla riga 8 è l'ultima operazione e il suo risultato viene restituito invariato. Sostituendo la chiamata in coda con l'aggiornamento $(l, r) \leftarrow (l + 1, r - 1)$ otteniamo:

Algorithm: IterativeIsPalindrome**Input** : Un array $a[1..n]$ **Output:** **true** se a è un palindromo, **false** altrimenti

```

1 function IterativeIsPalindrome( $a, n$ )
2    $l := 1$ 
3    $r := n$ 
4   while  $l < r$  do
5     if  $a[l] \neq a[r]$  then
6       return false
7     end
8      $l \leftarrow l + 1$ 
9      $r \leftarrow r - 1$ 
10  end
11  return true
12 end

```

Usa una memoria costante invece di una pila di profondità $\lfloor n/2 \rfloor + 1$.

5. Per invertire $a[l..r]$: si scambiano i due elementi estremi, poi si inverte il sotto-array interno $a[l + 1..r - 1]$. Un sotto-array con al più un elemento è già invertito (caso base, non c'è nulla da fare).

Algorithm: Reverse**Input** : Un array a e due indici l, r **Output:** Il sotto-array $a[l..r]$ invertito sul posto

```

1 function Reverse( $a, l, r$ )
2   if  $l < r$  then
3     scambia  $a[l]$  e  $a[r]$ 
4     Reverse( $a, l + 1, r - 1$ )
5   end
6 end

```

Esecuzione su $a = [5, 7, 2, 9, 4, 6]$:

chiamata	azione	array dopo lo scambio della chiamata
Reverse($a, 1, 6$)	scambia $a[1] = 5, a[6] = 6$	$[6, 7, 2, 9, 4, 5]$
Reverse($a, 2, 5$)	scambia $a[2] = 7, a[5] = 4$	$[6, 4, 2, 9, 7, 5]$
Reverse($a, 3, 4$)	scambia $a[3] = 2, a[4] = 9$	$[6, 4, 9, 2, 7, 5]$
Reverse($a, 4, 3$)	$l \geq r$: caso base, nulla da fare	$[6, 4, 9, 2, 7, 5]$

L'array finale è $[6, 4, 9, 2, 7, 5]$, dopo $\lfloor 6/2 \rfloor + 1 = 4$ chiamate e 3 scambi. La correttezza segue per induzione su $k = r - l + 1$ esattamente come nella domanda 2.

Esercizio 14.4 — Ricorsione (E4)

Si consideri l'algoritmo ricorsivo ingenuo per i numeri di Fibonacci ($F(0) = 0$, $F(1) = 1$, $F(n) = F(n-1) + F(n-2)$ per $n \geq 2$).

Algorithm: Fib

Input : Un numero naturale n

Output: Il numero di Fibonacci $F(n)$

```

1 function Fib( $n$ )
2   if  $n \leq 1$  then
3     return  $n$ 
4   end
5   return Fib( $n-1$ ) + Fib( $n-2$ )
6 end

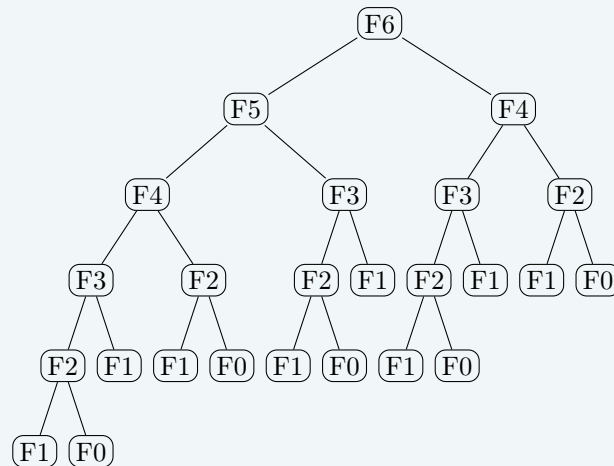
```

Domande

1. Disegnare l'albero delle chiamate ricorsive di **Fib**(6) e calcolare il valore restituito.
2. Contare il numero di chiamate $C(6)$ e verificare la formula $C(n) = 2F(n+1) - 1$.
3. Quante volte viene chiamata **Fib**(k), per ogni $k = 0, 1, \dots, 6$, durante il calcolo di **Fib**(6)? Che cosa si osserva?
4. Confrontare il numero di passi di **Fib** ($S(n) = 4F(n+1) - 2$) con il numero di passi di **IterativeFib** ($S(n) = 5n + 1$) per $n = 6$ e per $n = 20$.
5. Qual è la profondità della ricorsione di **Fib**(6)?

Soluzione

1. L'albero delle chiamate ricorsive di **Fib**(6) (scriviamo Fk per **Fib**(k)):



Risalendo dalle foglie alla radice: $\text{Fib}(2) = 1 + 0 = 1$, $\text{Fib}(3) = 1 + 1 = 2$, $\text{Fib}(4) = 2 + 1 = 3$, $\text{Fib}(5) = 3 + 2 = 5$, $\text{Fib}(6) = 5 + 3 = 8$. Quindi $F(6) = 8$.

2. Contando i nodi dell'albero livello per livello: $1 + 2 + 4 + 8 + 8 + 2 = 25$. La formula dà

$$C(6) = 2F(7) - 1 = 2 \cdot 13 - 1 = 25.$$

Equivalentemente, dall'equazione di ricorrenza $C(n) = C(n-1) + C(n-2) + 1$ con $C(0) = C(1) = 1$: $C(2) = 3$, $C(3) = 5$, $C(4) = 9$, $C(5) = 15$, $C(6) = 15 + 9 + 1 = 25$.

3. Contando i nodi con la stessa etichetta:

k	6	5	4	3	2	1	0	totale
numero di chiamate $\text{Fib}(k)$	1	1	2	3	5	8	5	25

I numeri di chiamate sono essi stessi numeri di Fibonacci: $\text{Fib}(k)$ viene chiamata $F(7-k)$ volte per $k \geq 1$ (e $\text{Fib}(0)$ viene chiamata $F(5)$ volte, una per ogni chiamata $\text{Fib}(2)$). Gli stessi sottoproblemi vengono risolti più e più volte: per esempio $F(2)$ viene calcolato 5 volte, mentre basterebbe calcolarlo una volta e memorizzarlo. Questa è la motivazione della **Programmazione Dinamica**.

4. Il numero di chiamate e di passi dei due algoritmi è:

n	chiamate $C(n) = 2F(n+1) - 1$	passi di Fib : $4F(n+1) - 2$	passi di IterativeFib : $5n + 1$
6	$2 \cdot 13 - 1 = 25$	$4 \cdot 13 - 2 = 50$	31
20	$2 \cdot 10946 - 1 = 21891$	$4 \cdot 10946 - 2 = 43782$	101

Per $n = 6$ i due algoritmi sono confrontabili, ma per $n = 20$ l'algoritmo ingenuo esegue più di 400 volte il numero di passi di quello iterativo: il primo cresce esponenzialmente, il secondo linearmente.

5. Il cammino più lungo dalla radice a una foglia è $\text{Fib}(6)$, $\text{Fib}(5)$, $\text{Fib}(4)$, $\text{Fib}(3)$, $\text{Fib}(2)$, $\text{Fib}(1)$: la profondità della ricorsione è 6. Benché vengano eseguite 25 chiamate, al più 6 stack frame sono simultaneamente in memoria.

Esercizio 14.5 — Ricorsione (E5)

Si considerino le Torri di Hanoi con $n = 4$ dischi, numerati 1, 2, 3, 4 dal più piccolo al più grande, inizialmente sul piolo A , da spostare sul piolo C usando il piolo B come piolo ausiliario, con l'algoritmo ricorsivo:

Algorithm: Hanoi

Input : Il numero di dischi $n \geq 1$, i pioli $from$, to , aux

Output: La sequenza di mosse che trasferisce i dischi 1, 2, ..., n da $from$ a to

```

1 function Hanoi( $n$ ,  $from$ ,  $to$ ,  $aux$ )
2   if  $n = 1$  then
3     |   sposta il disco 1 da  $from$  a  $to$ 
4   else
5     |   Hanoi( $n - 1$ ,  $from$ ,  $aux$ ,  $to$ )
6     |   sposta il disco  $n$  da  $from$  a  $to$ 
7     |   Hanoi( $n - 1$ ,  $aux$ ,  $to$ ,  $from$ )
8   end
9 end

```

Domande

1. Elencare, in ordine, tutte le mosse eseguite da $\text{Hanoi}(4, A, C, B)$.
2. Verificare che il numero di mosse è $M(4) = 2^4 - 1$.
3. Quante volte viene spostato ciascun disco? Dimostrare che, con n dischi, il disco k viene spostato esattamente 2^{n-k} volte ($\forall k \in \{1, 2, \dots, n\}$), e verificare che $\sum_{k=1}^n 2^{n-k} = 2^n - 1$.
4. Quante chiamate di Hanoi vengono eseguite per $n = 4$, e qual è la profondità della ricorsione?
5. Secondo la leggenda, $n = 64$ dischi d'oro vengono spostati alla velocità di una mossa al secondo. Quanto tempo ci vuole?

Soluzione

1. $\text{Hanoi}(4, A, C, B)$ esegue $\text{Hanoi}(3, A, B, C)$, poi sposta il disco 4 da A a C , poi esegue $\text{Hanoi}(3, B, C, A)$. Ogni chiamata con $n = 3$ effettua 7 mosse (secondo lo stesso schema di $\text{Hanoi}(3, A, C, B)$ nel capitolo, con i pioli rinominati):

mossa	disco	da	a	mossa	disco	da	a
1	1	A	B	9	1	B	C
2	2	A	C	10	2	B	A
3	1	B	C	11	1	C	A
4	3	A	B	12	3	B	C
5	1	C	A	13	1	A	B
6	2	C	B	14	2	A	C
7	1	A	B	15	1	B	C
8	4	A	C				

Le mosse 1–7 trasferiscono i dischi 1, 2, 3 da A a B (chiamata $\text{Hanoi}(3, A, B, C)$), la mossa 8 sposta il disco più grande, e le mosse 9–15 trasferiscono i dischi 1, 2, 3 da B a C (chiamata $\text{Hanoi}(3, B, C, A)$).

- Ci sono 15 mosse e $2^4 - 1 = 15$. Equivalentemente, $M(1) = 1$, $M(2) = 2 \cdot 1 + 1 = 3$, $M(3) = 2 \cdot 3 + 1 = 7$, $M(4) = 2 \cdot 7 + 1 = 15$.
- Dalla tabella: il disco 1 viene spostato 8 volte, il disco 2 viene spostato 4 volte, il disco 3 viene spostato 2 volte e il disco 4 viene spostato una volta; sono $2^{4-1}, 2^{4-2}, 2^{4-3}, 2^{4-4}$.

Dimostrazione per induzione su n che, in $\text{Hanoi}(n, \cdot, \cdot, \cdot)$, il disco k viene spostato 2^{n-k} volte per ogni $k \in \{1, 2, \dots, n\}$.

- Caso base** ($n = 1$): il disco 1 viene spostato una volta $= 2^{1-1}$.
- Passo induttivo** ($n \geq 2$): il disco n viene spostato solo dalla chiamata stessa, una volta $= 2^{n-n}$. Un disco $k \leq n - 1$ viene spostato solo all'interno delle due chiamate ricorsive con $n - 1$ dischi, e per ipotesi induttiva viene spostato $2^{(n-1)-k}$ volte in ciascuna di esse: in totale $2 \cdot 2^{n-1-k} = 2^{n-k}$ volte.

Sommando su tutti i dischi (somma geometrica),

$$\sum_{k=1}^n 2^{n-k} = 2^{n-1} + 2^{n-2} + \dots + 2 + 1 = 2^n - 1 = M(n).$$

Per $n = 4$: $8 + 4 + 2 + 1 = 15$.

- Ogni chiamata esegue esattamente una mossa (il caso base sposta il disco 1, le altre chiamate spostano il disco n), quindi il numero di chiamate è uguale al numero di mosse: 15 chiamate. L'albero delle chiamate ricorsive è un albero binario completo con 4 livelli ($1 + 2 + 4 + 8 = 15$ nodi), quindi la profondità della ricorsione è 4.

5. $M(64) = 2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$ mosse, cioè lo stesso numero di secondi. Un anno ha circa $365.25 \cdot 24 \cdot 3600 = 31\,557\,600$ secondi, quindi il tempo è

$$\frac{2^{64} - 1}{31\,557\,600} \approx 5.85 \cdot 10^{11} \text{ anni,}$$

circa 585 miliardi di anni (più di 40 volte l'età dell'universo). Poiché il numero di mosse di Hanoi è ottimo, nessun algoritmo può fare meglio.

Esercizio 14.6 — Ricorsione (E6)

Si considerino i due algoritmi ricorsivi seguenti: **FastPower** calcola x^n per elevamento al quadrato, **ToBinary** stampa la rappresentazione binaria di un numero naturale n .

Algorithm: FastPower

Input : Un numero x e un numero naturale n

Output: La potenza x^n

```

1 function FastPower( $x, n$ )
2   if  $n = 0$  then
3     return 1
4   end
5    $y := \text{FastPower}(x, \lfloor n/2 \rfloor)$ 
6   if  $n \bmod 2 = 0$  then
7     return  $y \cdot y$ 
8   else
9     return  $x \cdot y \cdot y$ 
10  end
11 end
```

Algorithm: ToBinary

Input : Un numero naturale n

Output: Le cifre binarie di n , stampate a partire dalla più significativa

```

1 function ToBinary( $n$ )
2   if  $n \leq 1$  then
3     stampa  $n$ 
4   else
5     ToBinary( $\lfloor n/2 \rfloor$ )
6     stampa  $n \bmod 2$ 
7   end
8 end
```

Domande

1. Mostrare la sequenza di chiamate di $\text{FastPower}(x, 25)$ e contare il numero di moltiplicazioni. Confrontare con l'algoritmo ingenuo, che esegue n moltiplicazioni.
2. Calcolare il numero di moltiplicazioni e il numero di passi $S(n) = 4\lfloor \log_2 n \rfloor + 6$ di FastPower per $n = 31$, $n = 32$ e $n = 100$.
3. Uno studente sostituisce la riga $y := \text{FastPower}(x, \lfloor n/2 \rfloor)$ e le righe seguenti con

$$\text{return FastPower}(x, \lfloor n/2 \rfloor) \cdot \text{FastPower}(x, \lfloor n/2 \rfloor) \quad (\text{per } x \text{ se } n \text{ è dispari}).$$

Scrivere l'equazione di ricorrenza del numero di chiamate $C(n)$ di questa versione, calcolare $C(13)$ e $C(16)$, e mostrare che $C(n) = \Theta(n)$.
4. Mostrare l'esecuzione di $\text{ToBinary}(37)$: chiamate, cifre stampate e numero di chiamate. Perché le cifre vengono stampate nell'ordine corretto, mentre l'algoritmo ChangeOfBase (capitolo «Algoritmi numerici») le produce a partire dalla meno significativa?
5. Dimostrare per induzione forte che $\text{ToBinary}(n)$ stampa la rappresentazione binaria di n , e che esegue $\lfloor \log_2 n \rfloor + 1$ chiamate per $n \geq 1$.

Soluzione

1. Gli esponenti delle chiamate sono 25, 12, 6, 3, 1, 0. Risalendo dal caso base:

chiamata	y	restituisce	moltiplicazioni
$\text{FastPower}(x, 0)$	–	1 (caso base)	0
$\text{FastPower}(x, 1)$	1	$x \cdot 1 \cdot 1 = x$ (dispari)	2
$\text{FastPower}(x, 3)$	x	$x \cdot x \cdot x = x^3$ (dispari)	2
$\text{FastPower}(x, 6)$	x^3	$x^3 \cdot x^3 = x^6$ (pari)	1
$\text{FastPower}(x, 12)$	x^6	$x^6 \cdot x^6 = x^{12}$ (pari)	1
$\text{FastPower}(x, 25)$	x^{12}	$x \cdot x^{12} \cdot x^{12} = x^{25}$ (dispari)	2

In totale $2 + 2 + 1 + 1 + 2 = 8$ moltiplicazioni, contro 25 dell'algoritmo ingenuo. Si noti che $25 = (11001)_2$: 5 cifre binarie, tre delle quali uguali a 1, e $5 + 3 = 8$.

2. Il numero di moltiplicazioni è il numero di cifre binarie di n più il numero di cifre uguali a 1 (ogni chiamata con $n \geq 1$ esegue 1 moltiplicazione se n è pari, 2 se n è dispari, e la parità degli esponenti successivi dà le cifre binarie di n):

n	binario	moltiplicazioni	$S(n) = 4\lfloor \log_2 n \rfloor + 6$	ingenuo
31	11111	$5 + 5 = 10$	$4 \cdot 4 + 6 = 22$	31
32	100000	$6 + 1 = 7$	$4 \cdot 5 + 6 = 26$	32
100	1100100	$7 + 3 = 10$	$4 \cdot 6 + 6 = 30$	100

Si noti che x^{32} richiede meno moltiplicazioni di x^{31} , benché $32 > 31$.

3. Ogni chiamata con $n \geq 1$ effettua ora **due** chiamate ricorsive su $\lfloor n/2 \rfloor$:

$$C(n) = 2C(\lfloor n/2 \rfloor) + 1 \quad (n \geq 1), \quad C(0) = 1.$$

Quindi $C(1) = 3$, $C(3) = 7$, $C(6) = 15$, $C(13) = 31$, e $C(2) = 7$, $C(4) = 15$, $C(8) = 31$, $C(16) = 63$.

L'albero delle chiamate ricorsive è ora un albero binario completo. I suoi livelli corrispondono agli esponenti $n, \lfloor n/2 \rfloor, \dots, 1, 0$, cioè ci sono $L = \lfloor \log_2 n \rfloor + 2$ livelli, con $1, 2, 4, \dots, 2^{L-1}$ nodi. Quindi

$$C(n) = 1 + 2 + \dots + 2^{L-1} = 2^L - 1 = 4 \cdot 2^{\lfloor \log_2 n \rfloor} - 1.$$

(Verifica: $C(13) = 4 \cdot 8 - 1 = 31$, $C(16) = 4 \cdot 16 - 1 = 63$.) Poiché $n/2 < 2^{\lfloor \log_2 n \rfloor} \leq n$, si ottiene $2n - 1 < C(n) \leq 4n - 1$, cioè $C(n) = \Theta(n)$: la versione con due chiamate ricorsive identiche **non è migliore dell'algoritmo ingenuo**. Calcolare il valore una sola volta e memorizzarlo in y è essenziale.

4. Esecuzione di `ToBinary(37)`:

Chiamate:

$$\begin{aligned} &\text{ToBinary}(37) \rightarrow \text{ToBinary}(18) \rightarrow \text{ToBinary}(9) \rightarrow \\ &\quad \text{ToBinary}(4) \rightarrow \text{ToBinary}(2) \rightarrow \text{ToBinary}(1) \end{aligned}$$

Stampe (nell'ordine in cui le chiamate terminano):

chiamata	azione	stampato finora
<code>ToBinary(1)</code>	caso base, stampa 1	1
<code>ToBinary(2)</code>	stampa $2 \bmod 2 = 0$	10
<code>ToBinary(4)</code>	stampa $4 \bmod 2 = 0$	100
<code>ToBinary(9)</code>	stampa $9 \bmod 2 = 1$	1001
<code>ToBinary(18)</code>	stampa $18 \bmod 2 = 0$	10010
<code>ToBinary(37)</code>	stampa $37 \bmod 2 = 1$	100101

Quindi $37 = (100101)_2$ (infatti $32 + 4 + 1 = 37$), con 6 chiamate.

I resti $n \bmod 2$ vengono calcolati nello stesso ordine di `ChangeOfBase` (divisioni ripetute: 1, 0, 1, 0, 0, 1, a partire dalla cifra meno significativa). Tuttavia, ogni chiamata stampa il proprio resto **dopo** che la sua chiamata ricorsiva è terminata, cioè dopo che tutte le cifre più significative sono state stampate: il call stack inverte l'ordine delle cifre, e non serve alcun array.

5. **Correttezza**, per induzione forte su n .

- **Casi base** ($n \in \{0, 1\}$): la rappresentazione binaria di n è la singola cifra n , che viene stampata.
- **Passo induttivo** ($n \geq 2$): si ha $n = 2\lfloor n/2 \rfloor + (n \bmod 2)$ con $1 \leq \lfloor n/2 \rfloor < n$. Se $\lfloor n/2 \rfloor = (d_{k-1} \dots d_0)_2$, allora

$$n = 2 \sum_{i=0}^{k-1} d_i 2^i + (n \bmod 2) = (d_{k-1} \dots d_0 (n \bmod 2))_2,$$

cioè la rappresentazione binaria di n è quella di $\lfloor n/2 \rfloor$ seguita dalla cifra $n \bmod 2$. Per ipotesi induttiva la chiamata ricorsiva stampa $d_{k-1} \dots d_0$, e poi l'algoritmo stampa $n \bmod 2$.

Numero di chiamate. Sia $C(n)$ il numero di chiamate per $n \geq 1$: $C(1) = 1$ e $C(n) = C(\lfloor n/2 \rfloor) + 1$ per $n \geq 2$. Dimostriamo $C(n) = \lfloor \log_2 n \rfloor + 1$ per induzione forte: $C(1) = 1 = \lfloor \log_2 1 \rfloor + 1$; per $n \geq 2$, usando $\lfloor \log_2 \lfloor n/2 \rfloor \rfloor = \lfloor \log_2 n \rfloor - 1$,

$$C(n) = (\lfloor \log_2 n \rfloor - 1 + 1) + 1 = \lfloor \log_2 n \rfloor + 1.$$

Questo è il numero di cifre binarie di n (una cifra stampata per ogni chiamata). Per $n = 37$: $\lfloor \log_2 37 \rfloor + 1 = 5 + 1 = 6$.

Equazioni di ricorrenza

Negli esercizi di questo capitolo le ricorrenze sono risolte con i tre metodi del capitolo «Equazioni di ricorrenza» (espansione, sostituzione, albero di ricorsione), usando le formule seguenti.

Per ogni $n, m \in \mathbb{N}$, $n, m \geq 1$, ogni $k \in \mathbb{N}$ e ogni numero reale $x \neq 1$:

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} \quad (\text{serie aritmetica})$$

$$\sum_{j=1}^m j^2 = \frac{m(m+1)(2m+1)}{6} \quad (\text{somma dei quadrati})$$

$$\sum_{i=0}^k x^i = \frac{x^{k+1} - 1}{x - 1} = \frac{1 - x^{k+1}}{1 - x} \quad (\text{serie geometrica})$$

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1 - x} \quad \text{se } |x| < 1 \quad (\text{serie geometrica infinita decrescente})$$

Per ogni $a, b, n > 0$ con $b \neq 1$, e ogni numero reale x :

$$a^{\log_b n} = n^{\log_b a}, \quad \log_b n = \frac{\log_2 n}{\log_2 b},$$

$$x - 1 < \lfloor x \rfloor \leq x, \quad 2\lfloor n/2 \rfloor \geq n - 1 \quad (n \in \mathbb{N}).$$

Esercizio 15.1 — Equazioni di ricorrenza (E1)

Consideriamo le seguenti ricorrenze.

- (a) $T(n) = T(n-1) + 2n - 1$ per $n > 1$, con $T(1) = 1$.
- (b) $T(n) = 3T(n-1)$ per $n > 1$, con $T(1) = 2$.
- (c) $T(n) = T\left(\frac{n}{3}\right) + 1$ per $n > 1$, con $T(1) = 1$, dove n è una potenza esatta di 3.
- (d) $T(n) = 2T\left(\frac{n}{2}\right) + n^2$ per $n > 1$, con $T(1) = 1$, dove n è una potenza esatta di 2.

Domande

1. Per ciascuna ricorrenza, calcolare i primi quattro valori di $T(n)$ (per (c) e (d), sui primi quattro valori ammissibili di n).
2. Risolvere ciascuna ricorrenza con il metodo di espansione, cioè determinare una forma chiusa di $T(n)$.
3. Verificare che le forme chiuse siano coerenti con i valori calcolati alla domanda 1.
4. Determinare l'ordine di grandezza di ciascuna ricorrenza (notazione Θ).

Soluzione

1. Primi valori:

- (a) $T(1) = 1$, $T(2) = 1 + 3 = 4$, $T(3) = 4 + 5 = 9$, $T(4) = 9 + 7 = 16$.
- (b) $T(1) = 2$, $T(2) = 6$, $T(3) = 18$, $T(4) = 54$.
- (c) $T(1) = 1$, $T(3) = T(1) + 1 = 2$, $T(9) = T(3) + 1 = 3$, $T(27) = T(9) + 1 = 4$.
- (d) $T(1) = 1$, $T(2) = 2 \cdot 1 + 4 = 6$, $T(4) = 2 \cdot 6 + 16 = 28$, $T(8) = 2 \cdot 28 + 64 = 120$.

2. Espansione.

(a) Espandendo,

$$\begin{aligned}
 T(n) &= T(n-1) + (2n-1) \\
 &= T(n-2) + (2(n-1)-1) + (2n-1) \\
 &\vdots \\
 &= T(n-k) + \sum_{i=n-k+1}^n (2i-1).
 \end{aligned}$$

Con $k = n-1$ e usando la serie aritmetica $\sum_{i=1}^n i = \frac{n(n+1)}{2}$:

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

(b) Espandendo,

$$T(n) = 3T(n-1) = 3^2 T(n-2) = 3^3 T(n-3) = \dots = 3^k T(n-k).$$

Con $k = n-1$: $T(n) = 3^{n-1} T(1) = 2 \cdot 3^{n-1}$.

(c) Sia $n = 3^k$. Espandendo,

$$T(n) = T\left(\frac{n}{3}\right) + 1 = T\left(\frac{n}{3^2}\right) + 2 = \dots = T\left(\frac{n}{3^j}\right) + j.$$

La condizione al contorno è raggiunta quando $\frac{n}{3^j} = 1$, cioè $j = \log_3 n$:

$$T(n) = T(1) + \log_3 n = \log_3 n + 1.$$

(d) Espandendo,

$$\begin{aligned}
 T(n) &= 2T\left(\frac{n}{2}\right) + n^2 \\
 &= 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n^2}{4}\right) + n^2 = 2^2 T\left(\frac{n}{2^2}\right) + n^2 \left(1 + \frac{1}{2}\right) \\
 &= 2^2 \left(2T\left(\frac{n}{2^3}\right) + \frac{n^2}{16}\right) + n^2 \left(1 + \frac{1}{2}\right) = 2^3 T\left(\frac{n}{2^3}\right) + n^2 \left(1 + \frac{1}{2} + \frac{1}{4}\right) \\
 &\vdots \\
 &= 2^j T\left(\frac{n}{2^j}\right) + n^2 \sum_{i=0}^{j-1} \left(\frac{1}{2}\right)^i.
 \end{aligned}$$

Con $j = \log_2 n$ (quindi $2^j = n$) e la serie geometrica $\sum_{i=0}^{j-1} x^i = \frac{1-x^j}{1-x}$:

$$T(n) = nT(1) + n^2 \frac{1 - \left(\frac{1}{2}\right)^{\log_2 n}}{1 - \frac{1}{2}} = n + 2n^2 \left(1 - \frac{1}{n}\right) = 2n^2 - n.$$

3. Verifica:

- (a) n^2 : 1, 4, 9, 16. ✓
- (b) $2 \cdot 3^{n-1}$: 2, 6, 18, 54. ✓
- (c) $\log_3 n + 1$ per $n = 1, 3, 9, 27$: 1, 2, 3, 4. ✓
- (d) $2n^2 - n$ per $n = 1, 2, 4, 8$: 1, $8 - 2 = 6$, $32 - 4 = 28$, $128 - 8 = 120$. ✓

4. Ordini di grandezza:

- (a) $T(n) = n^2 = \Theta(n^2)$.
- (b) $T(n) = 2 \cdot 3^{n-1} = \frac{2}{3} 3^n = \Theta(3^n)$ (esponenziale).
- (c) $T(n) = \log_3 n + 1 = \Theta(\log_2 n)$ (la base del logaritmo è irrilevante dentro Θ).
- (d) $T(n) = 2n^2 - n = \Theta(n^2)$, poiché $\lim_{n \rightarrow \infty} \frac{2n^2 - n}{n^2} = 2 \in \mathbb{R}_{>0}$. Si noti che il costo è dominato dalla radice (n^2).

Esercizio 15.2 — Equazioni di ricorrenza (E2)

Consideriamo le seguenti ricorrenze, definite per ogni $n \in \mathbb{N}$, $n \geq 1$.

$$(A) \quad T(n) = \begin{cases} 1 & \text{se } n = 1 \\ T(n-1) + n & \text{se } n > 1 \end{cases} \quad (B) \quad U(n) = \begin{cases} 1 & \text{se } n = 1 \\ U(\lfloor n/2 \rfloor) + 1 & \text{se } n > 1 \end{cases}$$

Domande

1. Usando il metodo di sostituzione, dimostrare che $T(n) \leq n^2$ per ogni $n \geq 1$.

2. Usando il metodo di sostituzione, dimostrare che $T(n) \geq \frac{1}{2}n^2$ per ogni $n \geq 1$. Concludere che $T(n) = \Theta(n^2)$.
3. Calcolare $U(n)$ per $n = 1, 2, \dots, 8$.
4. Dimostrare che $U(n) = O(\log_2 n)$, mostrando che $U(n) \leq c \log_2 n$ per un'opportuna costante c e ogni $n \geq 2$. Spiegare perché la limitazione non può valere per $n = 1$ e quali casi base servono.
5. Dimostrare che $U(n) \geq \log_2(n+1)$ per ogni $n \geq 1$. Concludere che $U(n) = \Theta(\log_2 n)$.

Soluzione

1. Ipotizziamo $T(n) \leq cn^2$ e cerchiamo c . Supponiamo (ipotesi induttiva) $T(n-1) \leq c(n-1)^2$. Allora

$$T(n) \leq c(n-1)^2 + n = cn^2 - 2cn + c + n = cn^2 - ((2c-1)n - c).$$

Ci serve $(2c-1)n - c \geq 0$. Con $c = 1$ diventa $n - 1 \geq 0$, vero per ogni $n \geq 1$.

Caso base: $T(1) = 1 \leq 1^2$. ✓

Passo induttivo ($n \geq 2$): $T(n) \leq (n-1)^2 + n = n^2 - n + 1 \leq n^2$. ✓

Quindi $T(n) \leq n^2$ per ogni $n \geq 1$, cioè $T(n) = O(n^2)$.

2. Ipotizziamo $T(n) \geq dn^2$. Supponiamo $T(n-1) \geq d(n-1)^2$. Allora

$$T(n) \geq d(n-1)^2 + n = dn^2 + ((1-2d)n + d).$$

Ci serve $(1-2d)n + d \geq 0$: con $d = \frac{1}{2}$ diventa $\frac{1}{2} \geq 0$. ✓

Caso base: $T(1) = 1 \geq \frac{1}{2}$. ✓

Quindi $\frac{1}{2}n^2 \leq T(n) \leq n^2$ per ogni $n \geq 1$, e $T(n) = \Theta(n^2)$. (In effetti, la soluzione esatta è $T(n) = \frac{n(n+1)}{2}$.)

$$3. \begin{array}{c|cccccccc} & n & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline U(n) & & 1 & 2 & 2 & 3 & 3 & 3 & 3 & 4 \end{array}$$

Per esempio, $U(6) = U(3) + 1 = U(1) + 2 = 3$ e $U(8) = U(4) + 1 = U(2) + 2 = U(1) + 3 = 4$. (In generale $U(n) = \lfloor \log_2 n \rfloor + 1$, il numero di bit di n .)

4. Ipotizziamo $U(n) \leq c \log_2 n$.

Passo induttivo. Supponiamo $U(\lfloor n/2 \rfloor) \leq c \log_2 \lfloor n/2 \rfloor$. Allora

$$U(n) \leq c \log_2 \lfloor n/2 \rfloor + 1 \leq c \log_2 \frac{n}{2} + 1 = c \log_2 n - c + 1 \leq c \log_2 n,$$

purché $c \geq 1$.

Casi base. Per $n = 1$: $U(1) = 1 > c \log_2 1 = 0$ per ogni c , quindi la limitazione non può valere per $n = 1$. Dimostriamo la limitazione per $n \geq n_0 = 2$. Per $n \geq 4$ si ha $\lfloor n/2 \rfloor \geq 2$, quindi il passo induttivo usa solo valori ≥ 2 . Per $n = 2, 3$ si ha $\lfloor n/2 \rfloor = 1$: questi sono i casi base. Ci serve

$$U(2) = 2 \leq c \log_2 2 = c, \quad U(3) = 2 \leq c \log_2 3 \approx 1.58 c.$$

Entrambe valgono con $c = 2$.

Quindi $U(n) \leq 2 \log_2 n$ per ogni $n \geq 2$, e $U(n) = O(\log_2 n)$.

5. Per induzione forte su n .

Caso base: $U(1) = 1 = \log_2 2$. ✓

Passo induttivo ($n \geq 2$): poiché $1 \leq \lfloor n/2 \rfloor < n$, l'ipotesi induttiva dà

$$U(\lfloor n/2 \rfloor) \geq \log_2(\lfloor n/2 \rfloor + 1).$$

Allora

$$\begin{aligned} U(n) &\geq \log_2(\lfloor n/2 \rfloor + 1) + 1 = \log_2(2\lfloor n/2 \rfloor + 2) \\ &\geq \log_2((n-1) + 2) = \log_2(n+1), \end{aligned}$$

dove abbiamo usato $2\lfloor n/2 \rfloor \geq n-1$.

Si noti che la scelta naturale $U(n) \geq \log_2 n$ **non** funziona direttamente: per n dispari il passo induttivo dà soltanto

$$U(n) \geq \log_2 \lfloor n/2 \rfloor + 1 = \log_2(2\lfloor n/2 \rfloor) = \log_2(n-1).$$

L'enunciato leggermente più forte $U(n) \geq \log_2(n+1)$ fa funzionare l'induzione.

Poiché $\log_2(n+1) \geq \log_2 n$, si ha $U(n) = \Omega(\log_2 n)$. Insieme alla domanda 4,

$$U(n) = \Theta(\log_2 n).$$

Esercizio 15.3 — Equazioni di ricorrenza (E3)

Consideriamo la ricorrenza

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

dove n è una potenza esatta di 2.

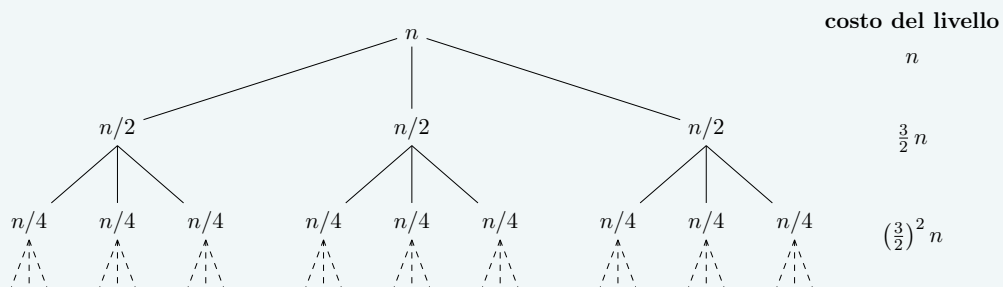
Domande

1. Disegnare i primi tre livelli dell'albero di ricorsione (profondità 0, 1, 2) e indicare il costo di ciascun nodo.

2. Determinare, per una profondità generica i , il numero di nodi, il costo di ciascun nodo e il costo del livello. Determinare la profondità dell'albero e il numero di foglie.
3. Sommare i costi di tutti i livelli e ottenere una forma chiusa di $T(n)$.
4. Determinare l'ordine di grandezza di $T(n)$. Quale parte dell'albero domina il costo totale?
5. Verificare per induzione (metodo di sostituzione) la forma chiusa ottenuta alla domanda 3, e controllarla numericamente per $n = 1, 2, 4, 8$.

Soluzione

1. La radice ha costo n e tre figli, ciascuno con costo $\frac{n}{2}$; ciascuno di essi ha tre figli con costo $\frac{n}{4}$.



2. A profondità i :
 - numero di nodi: 3^i ;
 - dimensione di ciascun sottoproblema: $\frac{n}{2^i}$, quindi costo di ciascun nodo: $\frac{n}{2^i}$;
 - costo del livello: $3^i \frac{n}{2^i} = \left(\frac{3}{2}\right)^i n$.

La dimensione del sottoproblema raggiunge 1 quando $\frac{n}{2^i} = 1$, cioè a profondità $k = \log_2 n$. Il numero di foglie è

$$3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585},$$

ciascuna con costo $T(1) = 1$.

3. Sia $k = \log_2 n$. Usando la serie geometrica $\sum_{i=0}^{k-1} x^i = \frac{x^k - 1}{x - 1}$ con $x = \frac{3}{2}$:

$$\sum_{i=0}^{k-1} \left(\frac{3}{2}\right)^i n = n \frac{\left(\frac{3}{2}\right)^k - 1}{\frac{3}{2} - 1} = 2n \left(\frac{3^k}{2^k} - 1\right) = 2n \left(\frac{3^k}{n} - 1\right) = 2 \cdot 3^k - 2n.$$

Aggiungendo il costo delle foglie, $3^k \cdot 1$:

$$T(n) = 2 \cdot 3^k - 2n + 3^k = 3 \cdot 3^{\log_2 n} - 2n = 3 n^{\log_2 3} - 2n.$$

4. Poiché $\log_2 3 > 1$, il termine $n^{\log_2 3}$ domina n :

$$\lim_{n \rightarrow \infty} \frac{3n^{\log_2 3} - 2n}{n^{\log_2 3}} = \lim_{n \rightarrow \infty} \left(3 - \frac{2}{n^{\log_2 3 - 1}} \right) = 3 \in \mathbb{R}_{>0}.$$

Quindi $T(n) = \Theta(n^{\log_2 3})$. I costi dei livelli crescono geometricamente (ragione $\frac{3}{2} > 1$): il costo totale è dominato dalle **foglie**.

5. Dimostriamo $T(n) = 3n^{\log_2 3} - 2n$ per ogni $n = 2^k$, per induzione su k .

Caso base $k = 0$ ($n = 1$): $3 \cdot 1 - 2 = 1 = T(1)$. ✓

Passo induttivo: supponiamo la formula valida per $\frac{n}{2}$. Poiché $(\frac{n}{2})^{\log_2 3} = \frac{n^{\log_2 3}}{2^{\log_2 3}} = \frac{n^{\log_2 3}}{3}$,

$$T(n) = 3 \left(3 \frac{n^{\log_2 3}}{3} - 2 \frac{n}{2} \right) + n = 3n^{\log_2 3} - 3n + n = 3n^{\log_2 3} - 2n. \quad \checkmark$$

Verifica numerica ($n^{\log_2 3} = 3^k$):

n	$T(n)$ dalla ricorrenza	$3 \cdot 3^k - 2n$
1	1	$3 - 2 = 1$
2	$3 \cdot 1 + 2 = 5$	$9 - 4 = 5$
4	$3 \cdot 5 + 4 = 19$	$27 - 8 = 19$
8	$3 \cdot 19 + 8 = 65$	$81 - 16 = 65$

Esercizio 15.4 — Equazioni di ricorrenza (E4)

Consideriamo le seguenti ricorrenze.

$$(A) \quad T(n) = \begin{cases} 1 & \text{se } 1 \leq n < 4 \\ T(\frac{n}{2}) + T(\frac{n}{4}) + n & \text{se } n \geq 4 \end{cases} \quad (\text{definita per ogni reale } n \geq 1)$$

$$(B) \quad U(n) = \begin{cases} 0 & \text{se } n = 0 \\ U(n-2) + n^2 & \text{se } n \geq 2 \end{cases} \quad (\text{definita per ogni } n \in \mathbb{N} \text{ pari})$$

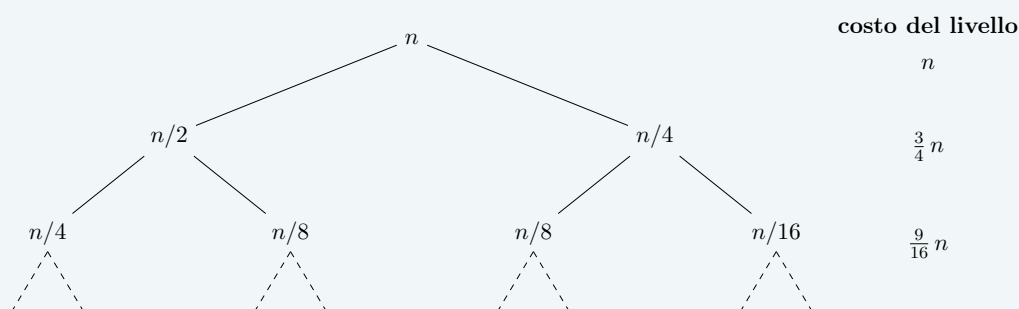
Domande

- Disegnare i primi tre livelli dell'albero di ricorsione di (A) e calcolare il costo di ciascuno di questi livelli.
- Mostrare che il costo del livello a profondità i dell'albero di (A) è al più $(\frac{3}{4})^i n$, e usare questo fatto per ipotizzare una limitazione superiore per $T(n)$.

3. Dimostrare, con il metodo di sostituzione, che $T(n) \leq 4n$ per ogni $n \geq 1$. Concludere che $T(n) = \Theta(n)$.
4. Disegnare l'albero di ricorsione di (B). Che forma ha? Qual è la sua profondità?
5. Determinare una forma chiusa di $U(n)$ e il suo ordine di grandezza. (Si ricordi che $\sum_{j=1}^m j^2 = \frac{m(m+1)(2m+1)}{6}$.)

Soluzione

1. Ogni nodo di dimensione $s \geq 4$ ha due figli, di dimensioni $\frac{s}{2}$ e $\frac{s}{4}$.



I costi dei primi tre livelli (per n abbastanza grande, in modo che tutti questi nodi siano interni) sono:

$$n, \quad \frac{n}{2} + \frac{n}{4} = \frac{3}{4}n, \quad \frac{n}{4} + \frac{n}{8} + \frac{n}{8} + \frac{n}{16} = \frac{9}{16}n.$$

2. Se un nodo ha dimensione s , i suoi figli hanno dimensione totale $\frac{s}{2} + \frac{s}{4} = \frac{3}{4}s$. Quindi, per induzione su i , i nodi a profondità i hanno dimensione totale al più $\left(\frac{3}{4}\right)^i n$ (esattamente, se il livello è completo), e i nodi interni a profondità i hanno costo totale al più $\left(\frac{3}{4}\right)^i n$.

Sommando su tutti i livelli e usando la serie geometrica infinita decrescente $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$:

$$\sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i n = \frac{1}{1-\frac{3}{4}} n = 4n.$$

Poiché le foglie hanno costo costante, **ipotizziamo** $T(n) = O(n)$, più precisamente $T(n) \leq 4n$. (Le foglie non sono facili da contare, quindi dimostriamo l'ipotesi con il metodo di sostituzione.)

3. Dimostriamo $T(n) \leq 4n$ per ogni numero reale $n \geq 1$, per induzione su $k = \lfloor \log_2 n \rfloor$.
Casi base $k \in \{0, 1\}$, cioè $1 \leq n < 4$: $T(n) = 1 \leq 4 \leq 4n$. ✓

Passo induttivo $k \geq 2$, cioè $n \geq 4$: gli argomenti $\frac{n}{2} \geq 2$ e $\frac{n}{4} \geq 1$ sono nel dominio e hanno $\lfloor \log_2(\cdot) \rfloor < k$, quindi per l'ipotesi induttiva

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + n \leq 4\frac{n}{2} + 4\frac{n}{4} + n = 2n + n + n = 4n. \quad \checkmark$$

Quindi $T(n) = O(n)$.

Limitazione inferiore: per $n \geq 4$, tutti i termini sono non negativi e la sola radice costa n , quindi $T(n) \geq n$. Quindi $T(n) = \Omega(n)$ e

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

(Il costo è dominato dalla radice: i costi dei livelli decrescono geometricamente con ragione $\frac{3}{4}$.)

4. Ogni chiamata genera una *sola* chiamata ricorsiva, quindi l'albero di ricorsione è un **cammino**:

$$n^2 \longrightarrow (n-2)^2 \longrightarrow (n-4)^2 \longrightarrow \dots \longrightarrow 2^2 \longrightarrow U(0) = 0.$$

Il nodo a profondità i ha dimensione $n-2i$ e costo $(n-2i)^2$. La dimensione raggiunge 0 a profondità $\frac{n}{2}$, che è la profondità dell'albero.

5. Sommando i costi di tutti i livelli, con $m = \frac{n}{2}$ e $j = m - i$ (in modo che $n - 2i = 2j$):

$$U(n) = \sum_{i=0}^{m-1} (n-2i)^2 = \sum_{j=1}^m (2j)^2 = 4 \sum_{j=1}^m j^2 = 4 \frac{m(m+1)(2m+1)}{6}.$$

Sostituendo $m = \frac{n}{2}$:

$$U(n) = 4 \frac{\frac{n}{2} \cdot \frac{n+2}{2} \cdot (n+1)}{6} = \frac{n(n+1)(n+2)}{6}.$$

Verifica: $U(2) = 4$ e $\frac{2 \cdot 3 \cdot 4}{6} = 4$; $U(4) = 4 + 16 = 20$ e $\frac{4 \cdot 5 \cdot 6}{6} = 20$; $U(6) = 20 + 36 = 56$ e $\frac{6 \cdot 7 \cdot 8}{6} = 56$. $\checkmark$

Ordine di grandezza:

$$\lim_{n \rightarrow \infty} \frac{n(n+1)(n+2)/6}{n^3} = \frac{1}{6} \in \mathbb{R}_{>0} \quad \implies \quad U(n) = \Theta(n^3).$$

Si noti che, a differenza di (A), il costo dei livelli decresce solo lentamente (in modo polinomiale, non geometrico) e la profondità è lineare in n : nessun livello domina, e il totale è $\Theta(n)$ livelli per $\Theta(n^2)$ per livello (almeno metà dei livelli costa almeno $\frac{n^2}{4}$).

Esercizio 15.5 — Equazioni di ricorrenza (E5)

Consideriamo i seguenti due algoritmi ricorsivi. Supponiamo che ogni moltiplicazione sia un'operazione primitiva (un passo), e che x sia memorizzato con un numero costante di bit.

Algorithm: FastPower

Input : Un numero intero x e un numero naturale n

Output: Un numero intero

```

1 function FastPower( $x, n$ )
2   if  $n = 0$  then
3     return 1
4   end
5    $y := \text{FastPower}(x, \lfloor n/2 \rfloor)$ 
6   if  $n$  è pari then
7     return  $y \cdot y$ 
8   else
9     return  $x \cdot y \cdot y$ 
10  end
11 end

```

Algorithm: SlowPower

Input : Un numero intero x e un numero naturale n

Output: Un numero intero

```

1 function SlowPower( $x, n$ )
2   if  $n = 0$  then
3     return 1
4   end
5   if  $n$  è pari then
6     return  $\text{SlowPower}(x, n/2) \cdot \text{SlowPower}(x, n/2)$ 
7   else
8     return  $x \cdot \text{SlowPower}(x, \lfloor n/2 \rfloor) \cdot \text{SlowPower}(x, \lfloor n/2 \rfloor)$ 
9   end
10 end

```

Domande

1. Che cosa calcolano i due algoritmi? Giustificare la risposta.
2. Sia $S_F(n)$ il numero di passi eseguiti da **FastPower**. Scrivere una ricorrenza per $S_F(n)$, contando le esecuzioni di ciascuna riga, e calcolare $S_F(n)$ per $n = 0, 1, \dots, 8$.
3. Risolvere la ricorrenza per $S_F(n)$ e determinarne l'ordine di grandezza.

4. Sia $S_S(n)$ il numero di passi eseguiti da **SlowPower**. Scrivere una ricorrenza per $S_S(n)$ e risolverla con il metodo di espansione quando n è una potenza esatta di 2. Determinarne l'ordine di grandezza.
5. Determinare il tempo di esecuzione nel caso peggiore dei due algoritmi in funzione della dimensione dell'input $|I|$, e confrontarli.

Soluzione

1. Entrambi gli algoritmi calcolano x^n . Infatti, scrivendo $n = 2\lfloor n/2 \rfloor + r$ con $r \in \{0, 1\}$,

$$x^n = \left(x^{\lfloor n/2 \rfloor}\right)^2 \cdot x^r = \begin{cases} x^{n/2} \cdot x^{n/2} & \text{se } n \text{ è pari} \\ x \cdot x^{\lfloor n/2 \rfloor} \cdot x^{\lfloor n/2 \rfloor} & \text{se } n \text{ è dispari} \end{cases}$$

e $x^0 = 1$. Per induzione forte su n , se le chiamate ricorsive restituiscono $x^{\lfloor n/2 \rfloor}$, entrambi gli algoritmi restituiscono x^n . L'unica differenza è che **FastPower** calcola $x^{\lfloor n/2 \rfloor}$ una volta e lo memorizza in y , mentre **SlowPower** lo calcola due volte.

FastPower è la funzione `powerRec` del codice del corso (in C++, `n / 2` è la divisione intera $\lfloor n/2 \rfloor$ per $n \geq 0$):

```

1 long long powerRec(long long x, int n)
2 {
3     if (n == 0)
4     {
5         return 1;
6     }
7
8     long long h = powerRec(x, n / 2);
9
10    if (n % 2 == 0)
11    {
12        return h * h;
13    }
14
15    return h * h * x;
16 }
```

Per $x = 3$ e $n = 0, 1, \dots, 10$ restituisce 1, 3, 9, 27, 81, 243, 729, 2187, 6561, 19683, 59049, cioè 3^n .

2. **FastPower**.

- Se $n = 0$: le righe 2 e 3 sono eseguite una volta: $S_F(0) = 2$.
- Se $n \geq 1$: le righe 2, 5 e 6 sono eseguite una volta, ed esattamente una tra le righe 7 e 9 è eseguita una volta: 4 passi. Inoltre, la chiamata ricorsiva alla riga 5 esegue $S_F(\lfloor n/2 \rfloor)$ passi.

Quindi

$$S_F(n) = \begin{cases} 2 & \text{se } n = 0 \\ S_F(\lfloor n/2 \rfloor) + 4 & \text{se } n \geq 1 \end{cases}$$

n	0	1	2	3	4	5	6	7	8
$S_F(n)$	2	6	10	10	14	14	14	14	18

3. Espandendo,

$$S_F(n) = S_F(\lfloor n/2 \rfloor) + 4 = S_F(\lfloor n/4 \rfloor) + 8 = \dots = S_F(\lfloor n/2^j \rfloor) + 4j.$$

Per $n \geq 1$, $\lfloor n/2^j \rfloor = 0$ per la prima volta quando $2^j > n$, cioè per $j = \lfloor \log_2 n \rfloor + 1$ (il numero di bit di n). Quindi

$$S_F(n) = 4(\lfloor \log_2 n \rfloor + 1) + 2 \quad (n \geq 1).$$

Verifica: $S_F(1) = 4 + 2 = 6$, $S_F(3) = 8 + 2 = 10$, $S_F(8) = 16 + 2 = 18$. ✓

Poiché $\log_2 n - 1 < \lfloor \log_2 n \rfloor \leq \log_2 n$, si ha $4 \log_2 n + 2 < S_F(n) \leq 4 \log_2 n + 6$, e

$$S_F(n) = \Theta(\log_2 n).$$

4. SlowPower.

- Se $n = 0$: **righe** 2 e 3: $S_S(0) = 2$.
- Se $n \geq 1$: **righe** 2, 5 e una tra 6, 8: 3 passi, più *due* chiamate ricorsive su $\lfloor n/2 \rfloor$.

Quindi

$$S_S(n) = \begin{cases} 2 & \text{se } n = 0 \\ 2 S_S(\lfloor n/2 \rfloor) + 3 & \text{se } n \geq 1 \end{cases}$$

in modo che $S_S(1) = 2 \cdot 2 + 3 = 7$. Sia $n = 2^k$. Espandendo fino a $S_S(1)$:

$$\begin{aligned} S_S(n) &= 2 S_S\left(\frac{n}{2}\right) + 3 \\ &= 2^2 S_S\left(\frac{n}{2^2}\right) + 3(2 + 1) \\ &= 2^3 S_S\left(\frac{n}{2^3}\right) + 3(2^2 + 2 + 1) \\ &\vdots \\ &= 2^k S_S(1) + 3 \sum_{i=0}^{k-1} 2^i = 7 \cdot 2^k + 3(2^k - 1) = 10n - 3. \end{aligned}$$

Verifica: $S_S(1) = 7$, $S_S(2) = 2 \cdot 7 + 3 = 17 = 20 - 3$, $S_S(4) = 2 \cdot 17 + 3 = 37 = 40 - 3$. ✓

Quindi $S_S(n) = \Theta(n)$ per n potenza di 2. (Per un generico $n \geq 1$, $S_S(n)$ dipende solo dal numero di bit di n , e $S_S(n) = 10 \cdot 2^{\lfloor \log_2 n \rfloor} - 3$, che è compreso tra $5n - 3$ e $10n - 3$: ancora $\Theta(n)$.)

5. L'input è costituito da x (un numero costante di bit) e dal numero naturale n , quindi

$$|I| = \Theta(\log_2 n), \quad \text{cioè} \quad n = 2^{\Theta(|I|)}.$$

- **FastPower**: $S_F(n) = \Theta(\log_2 n) = \Theta(|I|)$.
- **SlowPower**: $S_S(n) = \Theta(n) = 2^{\Theta(|I|)}$.

Il tempo di esecuzione nel caso peggiore di **FastPower** è $T(|I|) = \Theta(|I|)$ (lineare nella dimensione dell'input), mentre il tempo di esecuzione nel caso peggiore di **SlowPower** è $T(|I|) = 2^{\Theta(|I|)}$ (esponenziale nella dimensione dell'input).

Calcolare lo stesso sotto-risultato due volte cambia la ricorrenza da $S(n) = S(n/2) + c$ a $S(n) = 2S(n/2) + c$, e il tempo di esecuzione da logaritmico a lineare in n .

Esercizio 15.6 — Equazioni di ricorrenza (E6)

Consideriamo le seguenti ricorrenze, dove n è una potenza esatta di 2.

$$(A) \quad T(n) = \begin{cases} 1 & \text{se } n = 1 \\ 4T\left(\frac{n}{2}\right) + n & \text{se } n > 1 \end{cases} \quad (B) \quad U(n) = \begin{cases} 1 & \text{se } n = 1 \\ 8U\left(\frac{n}{2}\right) + n^2 & \text{se } n > 1 \end{cases}$$

Domande

1. Provare a dimostrare $T(n) \leq cn^2$ con il metodo di sostituzione. Mostrare che l'induzione fallisce per ogni scelta della costante c .
2. Rafforzare l'ipotesi sottraendo un termine di ordine inferiore: dimostrare che $T(n) \leq cn^2 - dn$ per opportune costanti $c, d > 0$, trattando il caso base.
3. Dimostrare che $T(n) \geq n^2$ e concludere che $T(n) = \Theta(n^2)$.
4. Ripetere le domande 1–3 per $U(n)$, con l'ipotesi $U(n) \leq cn^3$.
5. Usando il metodo dell'albero di ricorsione, calcolare il valore esatto di $U(n)$ e confrontarlo con la limitazione ottenuta alla domanda 4.

Soluzione

1. Supponiamo $T\left(\frac{n}{2}\right) \leq c\left(\frac{n}{2}\right)^2$. Allora

$$T(n) \leq 4c \frac{n^2}{4} + n = cn^2 + n.$$

Poiché $n > 0$, $cn^2 + n > cn^2$ per ogni c : non possiamo concludere $T(n) \leq cn^2$. L'induzione fallisce, a causa del termine di ordine inferiore n . (Nota: concludere

“ $T(n) \leq cn^2 + n = O(n^2)$ ” sarebbe **sbagliato**: l'induzione deve dimostrare esattamente l'ipotesi induttiva, con la stessa costante c .)

2. Ipotizziamo $T(n) \leq cn^2 - dn$. Supponiamo $T\left(\frac{n}{2}\right) \leq c\left(\frac{n}{2}\right)^2 - d\frac{n}{2}$. Allora

$$T(n) \leq 4\left(c\frac{n^2}{4} - d\frac{n}{2}\right) + n = cn^2 - 2dn + n = cn^2 - dn - (d-1)n \leq cn^2 - dn,$$

purché $d \geq 1$.

Caso base: $T(1) = 1 \leq c - d$. Con $d = 1$ serve $c \geq 2$.

Scegliendo $c = 2$ e $d = 1$: $T(n) \leq 2n^2 - n$ per ogni n potenza di 2. Quindi $T(n) = O(n^2)$. (In effetti, $T(n) = 2n^2 - n$ esattamente, come si può verificare con il metodo dell'albero di ricorsione.)

3. Supponiamo $T\left(\frac{n}{2}\right) \geq \left(\frac{n}{2}\right)^2$. Allora

$$T(n) \geq 4\frac{n^2}{4} + n = n^2 + n \geq n^2.$$

Caso base: $T(1) = 1 \geq 1$. Quindi $n^2 \leq T(n) \leq 2n^2 - n$, e $T(n) = \Theta(n^2)$.

4. *Ipotesi ingenua*. Supponiamo $U\left(\frac{n}{2}\right) \leq c\left(\frac{n}{2}\right)^3$. Allora

$$U(n) \leq 8c\frac{n^3}{8} + n^2 = cn^3 + n^2,$$

e l'induzione fallisce (a causa del termine di ordine inferiore n^2).

Ipotesi rafforzata. Supponiamo $U\left(\frac{n}{2}\right) \leq c\left(\frac{n}{2}\right)^3 - d\left(\frac{n}{2}\right)^2$. Allora

$$U(n) \leq 8\left(c\frac{n^3}{8} - d\frac{n^2}{4}\right) + n^2 = cn^3 - 2dn^2 + n^2 = cn^3 - dn^2 - (d-1)n^2 \leq cn^3 - dn^2,$$

purché $d \geq 1$. *Caso base*: $U(1) = 1 \leq c - d$, soddisfatto da $c = 2$, $d = 1$. Quindi

$$U(n) \leq 2n^3 - n^2.$$

Limitazione inferiore. Supponiamo $U\left(\frac{n}{2}\right) \geq \left(\frac{n}{2}\right)^3$. Allora $U(n) \geq 8\frac{n^3}{8} + n^2 \geq n^3$; caso base $U(1) = 1 \geq 1$. Quindi $n^3 \leq U(n) \leq 2n^3 - n^2$, e $U(n) = \Theta(n^3)$.

5. A profondità i l'albero di ricorsione di (B) ha 8^i nodi, ciascuno di dimensione $\frac{n}{2^i}$ e costo $\left(\frac{n}{2^i}\right)^2$. Il costo del livello è

$$8^i \frac{n^2}{4^i} = 2^i n^2.$$

La profondità dell'albero è $\log_2 n$, e ci sono $2^{\log_2 n} = n^{\log_2 2} = n^1$ foglie, ciascuna con costo $U(1) = 1$. Usando $\sum_{i=0}^{k-1} 2^i = 2^k - 1$:

$$U(n) = \sum_{i=0}^{\log_2 n - 1} 2^i n^2 + n^3 = n^2 (2^{\log_2 n} - 1) + n^3 = n^2(n - 1) + n^3 = 2n^3 - n^2.$$

La limitazione della domanda 4 è quindi *esatta*: l'ipotesi rafforzata $cn^3 - dn^2$ con $c = 2$, $d = 1$ è esattamente la soluzione della ricorrenza.

Verifica numerica: $U(2) = 8 + 4 = 12 = 16 - 4$, $U(4) = 8 \cdot 12 + 16 = 112 = 128 - 16$, $U(8) = 8 \cdot 112 + 64 = 960 = 1024 - 64$. ✓

Il Master theorem

Negli esercizi di questo capitolo le ricorrenze in forma standard $T(n) = aT(n/b) + f(n)$, con costanti $a \geq 1$, $b > 1$ e $f(n)$ asintoticamente positiva, sono risolte con il Master theorem del capitolo «Il Master theorem», seguendo lo schema in cinque passi (identificare, calcolare $n^{\log_b a}$, confrontare, verificare, concludere). Si usano le formule seguenti.

Attenzione

I tre casi del Master theorem:

- **Caso 1:** se $f(n) = O(n^{\log_b a - \epsilon})$ per qualche $\epsilon > 0$, allora $T(n) = \Theta(n^{\log_b a})$.
- **Caso 2:** se $f(n) = \Theta(n^{\log_b a} \log^k n)$ per qualche $k \geq 0$, allora $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
- **Caso 3:** se $f(n) = \Omega(n^{\log_b a + \epsilon})$ per qualche $\epsilon > 0$ e $a f(n/b) \leq c f(n)$ per qualche $c < 1$ e per ogni n grande (condizione di regolarità), allora $T(n) = \Theta(f(n))$.

Costo dell'albero di ricorsione, per n potenza esatta di b :

$$T(n) = T(1) n^{\log_b a} + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right), \quad a^{\log_b n} = n^{\log_b a}.$$

Numeri armonici: $H_L = \sum_{i=1}^L \frac{1}{i}$ soddisfa $\ln(L+1) \leq H_L \leq 1 + \ln L$; inoltre $\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}$.

Esercizio 16.1 — Il Master theorem (E1)

Consideriamo le seguenti ricorrenze ($\log$ denota $\log_2$):

- (a) $T(n) = 9T(n/3) + n$;
- (b) $T(n) = 2T(n/4) + \sqrt{n}$;
- (c) $T(n) = 2T(n/2) + n^2$;
- (d) $T(n) = 4T(n/2) + n^2 \log n$;

(e) $T(n) = 6T(n/3) + n^2 \log n$.

Domande

1. Per ciascuna ricorrenza, identificare a , b , $f(n)$ e calcolare $n^{\log_b a}$.
2. Per ciascuna ricorrenza, determinare quale caso del Master theorem si applica, verificarne tutte le ipotesi (ϵ , k , condizione di regolarità) e dare l'ordine di grandezza di $T(n)$.
3. Per la ricorrenza (c), calcolare $T(n)$ esattamente per $n = 2^L$ supponendo $T(1) = 1$, e verificare il risultato della domanda precedente.

Soluzione

		a	b	$f(n)$	$n^{\log_b a}$
1.	(a)	9	3	n	$n^{\log_3 9} = n^2$
	(b)	2	4	$\sqrt{n}$	$n^{\log_4 2} = n^{1/2} = \sqrt{n}$
	(c)	2	2	n^2	$n^{\log_2 2} = n$
	(d)	4	2	$n^2 \log n$	$n^{\log_2 4} = n^2$
	(e)	6	3	$n^2 \log n$	$n^{\log_3 6} \approx n^{1.631}$

2. Seguiamo lo schema: identificare $\rightarrow$ calcolare $n^{\log_b a} \rightarrow$ confrontare $\rightarrow$ verificare $\rightarrow$ concludere.

- (a) n^2 è polinomialmente più grande di $f(n) = n \rightarrow$ **caso 1**. Infatti $f(n) = n = O(n^{2-\epsilon})$ con $\epsilon = 1$. Quindi

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

- (b) $n^{1/2}$ e $f(n) = \sqrt{n}$ hanno lo stesso ordine $\rightarrow$ **caso 2**: $f(n) = \Theta(n^{1/2} \log^k n)$ con $k = 0$. Quindi

$$T(n) = \Theta(\sqrt{n} \log n).$$

- (c) n è polinomialmente più piccolo di $f(n) = n^2 \rightarrow$ **caso 3**. $f(n) = n^2 = \Omega(n^{1+\epsilon})$ con $\epsilon = 1$. Condizione di regolarità:

$$2f\left(\frac{n}{2}\right) = 2 \cdot \frac{n^2}{4} = \frac{1}{2} n^2 = c f(n) \quad \text{con } c = \frac{1}{2} < 1,$$

per ogni n . Quindi

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

- (d) $f(n) = n^2 \log n$ è più grande di n^2 solo di un fattore $\log n$ (non polinomialmente) e ha la forma $n^{\log_b a} \log^k n \rightarrow$ **caso 2** con $k = 1$. Quindi

$$T(n) = \Theta(n^2 \log^2 n).$$

- (e) $n^{1.631}$ è polinomialmente più piccolo di $f(n) = n^2 \log n \rightarrow$ **caso 3**. Poiché $n^2 \log n \geq n^2$ per $n \geq 2$, $f(n) = \Omega(n^{\log_3 6 + \epsilon})$ con $\epsilon = 2 - \log_3 6 \approx 0.369 > 0$. Condizione di regolarità: per $n \geq 3$, poiché $\log(n/3) \leq \log n$,

$$6f\left(\frac{n}{3}\right) = 6 \frac{n^2}{9} \log \frac{n}{3} \leq \frac{2}{3} n^2 \log n = c f(n) \quad \text{con } c = \frac{2}{3} < 1.$$

Quindi

$$T(n) = \Theta(n^2 \log n).$$

3. Con l'albero di ricorsione (Lemma del costo dell'albero di ricorsione), con $n = 2^L$, $a = 2$, $T(1) = 1$:

$$\begin{aligned} T(n) &= 2^L \cdot 1 + \sum_{j=0}^{L-1} 2^j \left(\frac{n}{2^j}\right)^2 = n + n^2 \sum_{j=0}^{L-1} \left(\frac{1}{2}\right)^j \\ &= n + n^2 \frac{1 - (1/2)^L}{1 - 1/2} = n + 2n^2 \left(1 - \frac{1}{n}\right) = 2n^2 - n. \end{aligned}$$

Verifica con la ricorrenza: $T(2) = 2 + 4 = 6 = 2 \cdot 4 - 2$, $T(4) = 12 + 16 = 28 = 2 \cdot 16 - 4$, $T(8) = 56 + 64 = 120 = 2 \cdot 64 - 8$. Poiché

$$\lim_{n \rightarrow \infty} \frac{2n^2 - n}{n^2} = 2 \in \mathbb{R}_{>0},$$

si ha $T(n) = \Theta(n^2)$, come trovato con il caso 3. Si noti che i costi dei livelli $n^2(1/2)^j$ decrescono geometricamente: domina la radice.

Esercizio 16.2 — Il Master theorem (E2)

Consideriamo le seguenti ricorrenze ($\log$ denota $\log_2$):

- (a) $T(n) = 4T(n/2) + n^2/\log n$;
- (b) $T(n) = nT(n/2) + n$;
- (c) $T(n) = 3T(n/3) + n/2$;
- (d) $T(n) = T(n-2) + \log n$;
- (e) $T(n) = 64T(n/8) - n^2 \log n$;
- (f) $T(n) = \sqrt{2}T(n/2) + \log n$.

Domande

1. Per ciascuna ricorrenza, dire se il Master theorem può essere applicato. In caso contrario, spiegare con precisione quale ipotesi non vale.
2. Per le ricorrenze a cui il teorema si applica, dare l'ordine di grandezza di $T(n)$, seguendo lo schema consueto.
3. Per la ricorrenza (a), determinare l'ordine di grandezza di $T(n)$ per $n = 2^L$ e $T(1) = 1$ usando l'albero di ricorsione.

Soluzione

1. (a) **Non applicabile.** $a = 4$, $b = 2$, $n^{\log_2 4} = n^2$, $f(n) = n^2/\log n$. $f(n)$ è più piccola di n^2 ma non polinomialmente più piccola: per ogni $\epsilon > 0$

$$\lim_{n \rightarrow \infty} \frac{n^2/\log n}{n^{2-\epsilon}} = \lim_{n \rightarrow \infty} \frac{n^\epsilon}{\log n} = +\infty,$$

quindi il caso 1 non vale; il caso 2 richiederebbe $k = -1 < 0$; il caso 3 non vale poiché $f(n)$ è più piccola di n^2 . Questo è un **vuoto** tra il caso 1 e il caso 2.

- (b) **Non applicabile.** Il numero di sottoproblemi $a = n$ non è una costante.
 - (c) **Applicabile:** $a = 3$, $b = 3$, $f(n) = n/2$ (si veda la domanda 2).
 - (d) **Non applicabile.** Il sottoproblema ha dimensione $n - 2$, non n/b : la ricorrenza non è in forma standard.
 - (e) **Non applicabile.** $f(n) = -n^2 \log n < 0$ per ogni $n \geq 2$: f non è asintoticamente positiva.
 - (f) **Applicabile:** $a = \sqrt{2} \geq 1$, $b = 2$, $f(n) = \log n$ (si veda la domanda 2). Si noti che a non deve necessariamente essere un intero.
2. (c) $n^{\log_b a} = n^{\log_3 3} = n$. $f(n) = n/2$ ha lo stesso ordine di $n \rightarrow$ **caso 2**: $f(n) = \Theta(n \log^k n)$ con $k = 0$ (i fattori costanti non contano). Quindi

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

- (f) $n^{\log_b a} = n^{\log_2 \sqrt{2}} = n^{1/2}$. $f(n) = \log n$ è polinomialmente più piccola di $n^{1/2} \rightarrow$ **caso 1**. Infatti, con $\epsilon = \frac{1}{4}$,

$$\lim_{n \rightarrow \infty} \frac{\log n}{n^{1/2-1/4}} = \lim_{n \rightarrow \infty} \frac{\log n}{n^{1/4}} = 0^+,$$

quindi $f(n) = O(n^{1/2-\epsilon})$. Quindi

$$T(n) = \Theta(\sqrt{n}).$$

Anche se il Master theorem non si applica a (d), questa ricorrenza è facile da risolvere: $T(n)$ è la somma di circa $n/2$ termini $\log n, \log(n-2), \dots$, ciascuno al più $\log n$, e i primi $n/4$ di essi sono almeno $\log(n/2)$; quindi $T(n) = \Theta(n \log n)$.

3. Albero di ricorsione con $n = 2^L$: il livello j ha 4^j nodi di dimensione $n/2^j$, e $\log(n/2^j) = L - j$. Il costo del livello j è

$$4^j \frac{(n/2^j)^2}{L-j} = \frac{n^2}{L-j}.$$

Le foglie sono $4^L = n^2$, ciascuna con costo $T(1) = 1$. Quindi

$$T(n) = n^2 + \sum_{j=0}^{L-1} \frac{n^2}{L-j} = n^2 + n^2 \sum_{i=1}^L \frac{1}{i} = n^2 (1 + H_L),$$

dove H_L è l' L -esimo numero armonico. Confrontando la somma con l'integrale di $1/x$,

$$\ln(L+1) = \int_1^{L+1} \frac{dx}{x} \leq H_L \leq 1 + \int_1^L \frac{dx}{x} = 1 + \ln L,$$

quindi $H_L = \Theta(\log L) = \Theta(\log \log n)$, e

$$T(n) = \Theta(n^2 \log \log n).$$

Per esempio, $T(16) = 256 (1 + H_4) = 256 (1 + \frac{25}{12}) = \frac{2368}{3} \approx 789.33$, che concorda con la ricorrenza: $T(2) = 8$, $T(4) = 40$, $T(8) = \frac{544}{3}$, $T(16) = 4 \cdot \frac{544}{3} + \frac{256}{4} = \frac{2368}{3}$.

Esercizio 16.3 — Il Master theorem (E3)

Consideriamo le seguenti famiglie di ricorrenze, dipendenti da un parametro reale ($\log$ denota $\log_2$):

- (A) $T(n) = aT(n/4) + n^2$, con $a \geq 1$;
- (B) $T(n) = 8T(n/2) + n^d$, con $d \geq 0$;
- (C) $T(n) = 4T(n/b) + n \log n$, con $b > 1$.

Domande

1. Per la famiglia (A), determinare per quali valori di a si applica ciascun caso del Master theorem, e dare il corrispondente ordine di grandezza di $T(n)$. Verificare la condizione di regolarità quando necessario.

2. Stessa domanda per la famiglia (B), in funzione di d .
3. Stessa domanda per la famiglia (C), in funzione di b .
4. Nella famiglia (A), qual è il più grande intero a tale che $T(n) = O(n^2)$? Che cosa succede per $a = 17$?

Soluzione

1. Famiglia (A): $b = 4$, $f(n) = n^2$, $n^{\log_b a} = n^{\log_4 a}$. Confrontiamo gli esponenti $\log_4 a$ e 2; si noti che $\log_4 a = 2 \iff a = 16$.

- $a > 16$: $\log_4 a > 2 \rightarrow$ **caso 1**, poiché $f(n) = n^2 = O(n^{\log_4 a - \epsilon})$ con $\epsilon = \log_4 a - 2 > 0$. Quindi $T(n) = \Theta(n^{\log_4 a})$.
- $a = 16$: $n^{\log_4 16} = n^2 \rightarrow$ **caso 2** con $k = 0$. Quindi $T(n) = \Theta(n^2 \log n)$.
- $1 \leq a < 16$: $\log_4 a < 2 \rightarrow$ **caso 3**, con $f(n) = \Omega(n^{\log_4 a + \epsilon})$, $\epsilon = 2 - \log_4 a > 0$. Condizione di regolarità:

$$a f\left(\frac{n}{4}\right) = a \frac{n^2}{16} = \frac{a}{16} n^2 = c f(n) \quad \text{con } c = \frac{a}{16} < 1.$$

Quindi $T(n) = \Theta(n^2)$.

2. Famiglia (B): $a = 8$, $b = 2$, $n^{\log_2 8} = n^3$, $f(n) = n^d$.

- $0 \leq d < 3$: **caso 1** con $\epsilon = 3 - d > 0$. Quindi $T(n) = \Theta(n^3)$.
- $d = 3$: **caso 2** con $k = 0$. Quindi $T(n) = \Theta(n^3 \log n)$.
- $d > 3$: **caso 3** con $\epsilon = d - 3 > 0$. Condizione di regolarità:

$$8 \left(\frac{n}{2}\right)^d = \frac{8}{2^d} n^d = 2^{3-d} n^d = c f(n) \quad \text{con } c = 2^{3-d} < 1.$$

Quindi $T(n) = \Theta(n^d)$.

3. Famiglia (C): $a = 4$, $f(n) = n \log n$, $n^{\log_b a} = n^{\log_b 4}$. Confrontiamo $\log_b 4$ con l'esponente 1 di n in $f(n)$; si noti che $\log_b 4 = 1 \iff b = 4$, e $\log_b 4 > 1 \iff b < 4$.

- $1 < b < 4$: $p = \log_b 4 > 1 \rightarrow$ **caso 1**. Scegliamo $\epsilon = \frac{p-1}{2} > 0$; allora $p - \epsilon = 1 + \frac{p-1}{2} > 1$ e

$$\lim_{n \rightarrow \infty} \frac{n \log n}{n^{p-\epsilon}} = \lim_{n \rightarrow \infty} \frac{\log n}{n^{(p-1)/2}} = 0^+,$$

quindi $f(n) = O(n^{p-\epsilon})$. Quindi $T(n) = \Theta(n^{\log_b 4})$.

- $b = 4$: $n^{\log_4 4} = n$ e $f(n) = n \log n = \Theta(n \log^k n)$ con $k = 1 \rightarrow$ **caso 2**. Quindi $T(n) = \Theta(n \log^2 n)$.

- $b > 4$: $p = \log_b 4 < 1 \rightarrow$ **caso 3**: poiché $n \log n \geq n$ per $n \geq 2$, $f(n) = \Omega(n^{p+\epsilon})$ con $\epsilon = 1 - p > 0$. Condizione di regolarità, per $n \geq b$:

$$4 \frac{n}{b} \log \frac{n}{b} \leq \frac{4}{b} n \log n = c f(n) \quad \text{con } c = \frac{4}{b} < 1.$$

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

4. Dalla domanda 1, $T(n) = \Theta(n^2)$ per $1 \leq a < 16$, mentre $T(n) = \Theta(n^2 \log n)$ per $a = 16$, che non è $O(n^2)$ perché $\lim_{n \rightarrow \infty} \frac{n^2 \log n}{n^2} = +\infty$. Quindi il più grande intero è $a = 15$.

Per $a = 17$ siamo nel caso 1: $T(n) = \Theta(n^{\log_4 17}) \approx \Theta(n^{2.044})$. Una piccola variazione del numero di sottoproblemi cambia l'esponente del tempo di esecuzione.

Esercizio 16.4 — Il Master theorem (E4)

Consideriamo i seguenti due algoritmi ricorsivi. In **PairSum** la lunghezza $n = r - l + 1$ del sotto-array è una potenza di 2; in **TernaryMax** è una potenza di 3. La prima chiamata è sull'intero array $a[1..n]$, cioè con $l = 1$ e $r = n$.

Algorithm: PairSum

Input : Un array a di numeri interi e due indici $l \leq r$

Output: Un numero intero s

```

1 function PairSum( $a, l, r$ )
2   if  $l = r$  then
3     return 0
4   end
5    $m := \lfloor (l + r) / 2 \rfloor$ 
6    $s := \text{PairSum}(a, l, m) + \text{PairSum}(a, m + 1, r)$ 
7    $i := l$ 
8   while  $i \leq m$  do
9      $j := m + 1$ 
10    while  $j \leq r$  do
11       $s \leftarrow s + a[i] \cdot a[j]$ 
12       $j \leftarrow j + 1$ 
13    end
14     $i \leftarrow i + 1$ 
15  end
16  return  $s$ 
17 end
```

Algorithm: TernaryMax**Input** : Un array a di numeri interi e due indici $l \leq r$ **Output**: Un numero intero

```

1 function TernaryMax( $a, l, r$ )
2   if  $l = r$  then
3     return  $a[l]$ 
4   end
5    $t := (r - l + 1)/3$ 
6    $x := \text{TernaryMax}(a, l, l + t - 1)$ 
7    $y := \text{TernaryMax}(a, l + t, l + 2t - 1)$ 
8    $z := \text{TernaryMax}(a, l + 2t, r)$ 
9   return  $\max\{x, y, z\}$ 
10 end

```

Domande

1. Che cosa calcolano i due algoritmi?
2. Scrivere la ricorrenza per il numero di passi $S(n)$ eseguiti da **PairSum**, contando le esecuzioni di ciascuna riga, e risolverla con il Master theorem. Determinare il tempo di esecuzione nel caso peggiore $T(|I|)$.
3. Stessa domanda per **TernaryMax**. Poi calcolare $S(n)$ esattamente con l'albero di ricorsione e verificarlo per $n = 9$.

Soluzione

1. **PairSum**($a, 1, n$) restituisce la somma dei prodotti di tutte le coppie di elementi distinti,

$$\sum_{1 \leq i < j \leq n} a[i] a[j] :$$

ogni coppia è o dentro la metà sinistra, o dentro la metà destra (chiamate ricorsive), oppure ha i nella metà sinistra e j nella metà destra (i due cicli annidati).

TernaryMax($a, 1, n$) restituisce il valore massimo dell'array: divide l'array in tre parti di uguale lunghezza, calcola ricorsivamente il massimo di ciascuna parte e restituisce il più grande dei tre.

2. Sia $n = r - l + 1 \geq 2$ e $h = n/2$ (la lunghezza di ciascuna metà). Il numero di volte in cui ciascuna riga è eseguita, **esclusi i passi eseguiti all'interno delle chiamate ricorsive**, è:

- La riga 2 è eseguita 1 volta
- La riga 3 è eseguita 0 volte
- La riga 5 è eseguita 1 volta
- La riga 6 è eseguita 1 volta (più le due chiamate ricorsive su h elementi)
- La riga 7 è eseguita 1 volta
- La riga 8 è eseguita $h + 1$ volte
- La riga 9 è eseguita h volte
- La riga 10 è eseguita $h(h + 1)$ volte
- La riga 11 è eseguita h^2 volte
- La riga 12 è eseguita h^2 volte
- La riga 14 è eseguita h volte
- La riga 16 è eseguita 1 volta

In totale $1 + 1 + 1 + 1 + (h + 1) + h + h(h + 1) + h^2 + h^2 + h + 1 = 3h^2 + 4h + 6$. Con $h = n/2$, e poiché per $n = 1$ sono eseguite solo le righe 2 e 3,

$$S(n) = \begin{cases} 2 & \text{se } n = 1, \\ 2S(n/2) + \frac{3}{4}n^2 + 2n + 6 & \text{se } n \geq 2. \end{cases}$$

(Per esempio $S(2) = 2 \cdot 2 + \frac{3}{4} \cdot 4 + 4 + 6 = 17$.) Applichiamo il Master theorem:

- (a) $a = 2$, $b = 2$, $f(n) = \frac{3}{4}n^2 + 2n + 6$.
 (b) $n^{\log_b a} = n^{\log_2 2} = n$.
 (c) $f(n) = \Theta(n^2)$ è polinomialmente più grande di $n \rightarrow$ **caso 3**.
 (d) $f(n) \geq \frac{3}{4}n^2$, quindi $f(n) = \Omega(n^{1+\epsilon})$ con $\epsilon = 1$. Condizione di regolarità con $c = \frac{3}{4}$:

$$2f\left(\frac{n}{2}\right) = \frac{3}{8}n^2 + 2n + 12 \leq \frac{3}{4}\left(\frac{3}{4}n^2 + 2n + 6\right) = \frac{9}{16}n^2 + \frac{3}{2}n + \frac{9}{2}$$

$$\iff 0 \leq \frac{3}{16}n^2 - \frac{1}{2}n - \frac{15}{2} \iff 3n^2 - 8n - 120 \geq 0,$$

che vale per ogni $n \geq 8$ (la radice positiva è $\frac{8+\sqrt{1504}}{6} \approx 7.8$).

- (e) $S(n) = \Theta(f(n)) = \Theta(n^2)$.

Nell'ipotesi che ogni elemento usi un numero costante di bit, $|I| = \Theta(n)$, quindi $n = \Theta(|I|)$.

Il tempo di esecuzione nel caso peggiore di **PairSum**, in funzione della dimensione dell'input $|I|$, è

$$T(|I|) = \Theta(|I|^2).$$

3. Per **TernaryMax** con $n \geq 3$ (potenza di 3), escluse le chiamate ricorsive, le righe 2, 5, 6, 7, 8, 9 sono eseguite una volta ciascuna e la riga 3 non è eseguita: 6 passi. Per $n = 1$, sono eseguite le righe 2 e 3: 2 passi. Ciascuna delle tre chiamate è su $t = n/3$ elementi. Quindi

$$S(n) = \begin{cases} 2 & \text{se } n = 1, \\ 3S(n/3) + 6 & \text{se } n \geq 3. \end{cases}$$

- (a) $a = 3, b = 3, f(n) = 6$.
- (b) $n^{\log_b a} = n^{\log_3 3} = n$.
- (c) n è polinomialmente più grande di $f(n) = 6 \rightarrow$ **caso 1**.
- (d) $f(n) = 6 = O(n^{1-\epsilon})$ con $\epsilon = 1$.
- (e) $S(n) = \Theta(n)$.

Il tempo di esecuzione nel caso peggiore di **TernaryMax**, in funzione della dimensione dell'input $|I|$, è

$$T(|I|) = \Theta(|I|).$$

Valore esatto con l'albero di ricorsione, $n = 3^L$: il livello j ha 3^j nodi interni di costo 6, e ci sono $3^L = n$ foglie di costo 2:

$$S(n) = 2n + \sum_{j=0}^{L-1} 6 \cdot 3^j = 2n + 6 \frac{3^L - 1}{3 - 1} = 2n + 3(n - 1) = 5n - 3.$$

Verifica per $n = 9$: $S(1) = 2$, $S(3) = 3 \cdot 2 + 6 = 12$, $S(9) = 3 \cdot 12 + 6 = 42 = 5 \cdot 9 - 3$.
✓

Esercizio 16.5 — Il Master theorem (E5)

Uno studente risolve tre ricorrenze come segue ($\log$ denota $\log_2$, $T(1) = 1$):

- (a) $T(n) = 2T(n/2) + n \log n$. “ $f(n) = n \log n$ è più grande di $n^{\log_2 2} = n$, quindi per il caso 3 $T(n) = \Theta(n \log n)$.”
- (b) $T(n) = 2T(n/2) + n/\log n$. “ $f(n) = n/\log n$ è più piccola di n , quindi per il caso 1 $T(n) = \Theta(n)$.”
- (c) $T(n) = 2T(n/2) + n/\log^2 n$. “ $f(n) = n/\log^2 n$ è più piccola di n , quindi per il caso 1 $T(n) = \Theta(n)$.”

Domande

1. Trovare l'errore in ciascuna delle tre soluzioni.

2. Risolvere correttamente (a) con il Master theorem.
3. Risolvere (b) e (c) per $n = 2^L$ con l'albero di ricorsione. La risposta finale dello studente è corretta per (b)? E per (c)?
4. Verificare le formule per (b) e (c) calcolando $T(16)$ direttamente dalle ricorrenze.

Soluzione

1. Lo studente confronta $f(n)$ con $n^{\log_b a} = n$ ma dimentica che il confronto deve essere **polinomiale**.

(a) Il caso 3 richiede $f(n) = \Omega(n^{1+\epsilon})$ per qualche $\epsilon > 0$. Ma per ogni $\epsilon > 0$

$$\lim_{n \rightarrow \infty} \frac{n \log n}{n^{1+\epsilon}} = \lim_{n \rightarrow \infty} \frac{\log n}{n^\epsilon} = 0^+,$$

quindi $f(n) \neq \Omega(n^{1+\epsilon})$: il caso 3 non si applica.

(b) e (c) Il caso 1 richiede $f(n) = O(n^{1-\epsilon})$ per qualche $\epsilon > 0$. Ma per ogni $\epsilon > 0$

$$\lim_{n \rightarrow \infty} \frac{n / \log n}{n^{1-\epsilon}} = \lim_{n \rightarrow \infty} \frac{n^\epsilon}{\log n} = +\infty, \quad \lim_{n \rightarrow \infty} \frac{n / \log^2 n}{n^{1-\epsilon}} = \lim_{n \rightarrow \infty} \frac{n^\epsilon}{\log^2 n} = +\infty,$$

quindi il caso 1 non si applica. Il caso 2 richiederebbe $k = -1$ e $k = -2$, rispettivamente, ma $k \geq 0$. Entrambe le ricorrenze cadono nel **vuoto** tra il caso 1 e il caso 2: il Master theorem non può essere usato.

2. (a) $a = 2, b = 2, f(n) = n \log n, n^{\log_2 2} = n. f(n) = \Theta(n \log^k n)$ con $k = 1 \rightarrow$ **caso 2:**

$$T(n) = \Theta(n \log^2 n).$$

3. Per $n = 2^L$: il livello j ha 2^j nodi di dimensione $n/2^j$, con $\log(n/2^j) = L - j$; ci sono $2^L = n$ foglie di costo $T(1) = 1$.

(b) Il livello j costa $2^j \frac{n/2^j}{L-j} = \frac{n}{L-j}$, quindi

$$T(n) = n + \sum_{j=0}^{L-1} \frac{n}{L-j} = n + n \sum_{i=1}^L \frac{1}{i} = n(1 + H_L).$$

Poiché $\ln(L+1) \leq H_L \leq 1 + \ln L, H_L = \Theta(\log L)$ e

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

La risposta dello studente $\Theta(n)$ è **sbagliata**: $\lim_{n \rightarrow \infty} \frac{n \log \log n}{n} = +\infty$.

(c) Il livello j costa $2^j \frac{n/2^j}{(L-j)^2} = \frac{n}{(L-j)^2}$, quindi

$$T(n) = n + n \sum_{i=1}^L \frac{1}{i^2}.$$

La serie $\sum_{i \geq 1} \frac{1}{i^2}$ converge (a $\frac{\pi^2}{6} \approx 1.645$), quindi

$$n \leq T(n) \leq n \left(1 + \frac{\pi^2}{6}\right), \quad \text{cioè} \quad T(n) = \Theta(n).$$

La **risposta** dello studente è **corretta**, ma il **ragionamento è sbagliato**: il Master theorem non si applica, e il risultato deve essere dimostrato con l'albero di ricorsione. Qui i costi dei livelli decrescono così rapidamente verso le foglie che dominano le foglie (che sono n).

4. Per $n = 16$ ($L = 4$):

(b) $T(2) = 2 + \frac{2}{1} = 4$, $T(4) = 8 + \frac{4}{2} = 10$, $T(8) = 20 + \frac{8}{3} = \frac{68}{3}$, $T(16) = \frac{136}{3} + \frac{16}{4} = \frac{148}{3}$.
Formula: $16(1 + H_4) = 16(1 + \frac{25}{12}) = \frac{148}{3} \approx 49.33$. ✓

(c) $T(2) = 2 + \frac{2}{1} = 4$, $T(4) = 8 + \frac{4}{4} = 9$, $T(8) = 18 + \frac{8}{9} = \frac{170}{9}$, $T(16) = \frac{340}{9} + \frac{16}{16} = \frac{349}{9}$.
Formula: $16(1 + 1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16}) = 16(2 + \frac{61}{144}) = \frac{349}{9} \approx 38.78$. ✓

Esercizio 16.6 — Il Master theorem (E6)

Consideriamo le ricorrenze, definite per n potenza di 2:

$$T_1(n) = 3T_1(n/2) + n^2, \quad T_2(n) = 4T_2(n/2) + n, \quad T_1(1) = T_2(1) = 1.$$

Domande

1. Per $n = 8$, disegnare la tabella dell'albero di ricorsione di T_1 (livello, numero di nodi, dimensione, costo del livello) e verificare il lemma, confrontando con il valore di $T_1(8)$ calcolato direttamente dalla ricorrenza.
2. Stessa domanda per T_2 e $n = 8$.
3. Usando il lemma, trovare una formula chiusa per $T_1(n)$ e $T_2(n)$, $n = 2^L$.

4. Risolvere entrambe le ricorrenze con il Master theorem e controllare il risultato con le formule chiuse.
5. Per T_1 , controllare numericamente, per $n = 8$, la limitazione $f(n) \leq g(n) \leq \frac{f(n)}{1-c}$ del caso 3 del lemma sulla somma $g(n)$.

Soluzione

1. T_1 : $a = 3$, $b = 2$, $f(n) = n^2$, $\log_2 8 = 3$.

livello j	nodi 3^j	dimensione $8/2^j$	costo per nodo	costo del livello
0	1	8	64	64
1	3	4	16	48
2	9	2	4	36
3 (foglie)	27	1	$T_1(1) = 1$	27
totale				175

Il numero di foglie è $3^3 = 27 = 8^{\log_2 3}$ (infatti $8^{\log_2 3} = (2^3)^{\log_2 3} = (2^{\log_2 3})^3 = 3^3$).
Direttamente:

$$T_1(2) = 3 + 4 = 7, \quad T_1(4) = 21 + 16 = 37, \quad T_1(8) = 111 + 64 = 175. \quad \checkmark$$

2. T_2 : $a = 4$, $b = 2$, $f(n) = n$.

livello j	nodi 4^j	dimensione $8/2^j$	costo per nodo	costo del livello
0	1	8	8	8
1	4	4	4	16
2	16	2	2	32
3 (foglie)	$64 = 8^{\log_2 4}$	1	$T_2(1) = 1$	64
totale				120

Direttamente: $T_2(2) = 4 + 2 = 6$, $T_2(4) = 24 + 4 = 28$, $T_2(8) = 112 + 8 = 120$. $\checkmark$

3. Con $n = 2^L$:

$$T_1(n) = n^{\log_2 3} + \sum_{j=0}^{L-1} 3^j \frac{n^2}{4^j} = n^{\log_2 3} + n^2 \frac{1 - (3/4)^L}{1 - 3/4} = n^{\log_2 3} + 4n^2 - 4n^2 \frac{3^L}{4^L}.$$

Poiché $4^L = n^2$ e $3^L = n^{\log_2 3}$,

$$T_1(n) = 4n^2 - 3n^{\log_2 3}.$$

(Verifica: $4 \cdot 64 - 3 \cdot 27 = 256 - 81 = 175$.) Analogamente

$$T_2(n) = n^2 + \sum_{j=0}^{L-1} 4^j \frac{n}{2^j} = n^2 + n(2^L - 1) = 2n^2 - n.$$

(Verifica: $128 - 8 = 120$.)

4. • T_1 : $n^{\log_2 3} \approx n^{1.585}$ è polinomialmente più piccolo di $f(n) = n^2 \rightarrow$ **caso 3**, con $\epsilon = 2 - \log_2 3 \approx 0.415$. Regolarità: $3 \left(\frac{n}{2}\right)^2 = \frac{3}{4}n^2$, quindi $c = \frac{3}{4} < 1$. Quindi $T_1(n) = \Theta(n^2)$. Formula chiusa:

$$\lim_{n \rightarrow \infty} \frac{4n^2 - 3n^{\log_2 3}}{n^2} = \lim_{n \rightarrow \infty} \left(4 - \frac{3}{n^{2-\log_2 3}}\right) = 4 \in \mathbb{R}_{>0}. \quad \checkmark$$

- T_2 : $n^{\log_2 4} = n^2$ è polinomialmente più grande di $f(n) = n \rightarrow$ **caso 1**, con $\epsilon = 1$. Quindi $T_2(n) = \Theta(n^2)$. Formula chiusa: $\lim_{n \rightarrow \infty} \frac{2n^2 - n}{n^2} = 2 \in \mathbb{R}_{>0}. \quad \checkmark$

Entrambe le ricorrenze sono $\Theta(n^2)$, ma per ragioni diverse: in T_1 il costo è dominato dalla radice (i costi dei livelli $n^2(3/4)^j$ decrescono), in T_2 dalle foglie (i costi dei livelli $n 2^j$ crescono).

5. Per T_1 e $n = 8$: $g(8) = 64 + 48 + 36 = 148$, $f(8) = 64$, $c = \frac{3}{4}$:

$$64 \leq 148 \leq \frac{64}{1 - 3/4} = 256. \quad \checkmark$$

Algoritmi ricorsivi

Negli esercizi di questo capitolo i passi si contano come nel capitolo «Algoritmi ricorsivi»: ogni riga di pseudocodice eseguita conta come un passo, e una riga che contiene una chiamata conta un passo più i passi della funzione chiamata. Si usano i seguenti risultati del capitolo.

Attenzione

Numero di passi su un array (o sotto-array) di n elementi:

<code>SumIterative(a, n)</code>	$3n + 4$
<code>RecursiveSum($a, 1, n$)</code>	$7n - 5$, con $2n - 1$ chiamate e profondità $\lceil \log_2 n \rceil + 1$
<code>RecursiveSumLinear(a, n)</code>	$3n - 1$, con n chiamate e profondità n
<code>Merge(a, l, m, r), $n = r - l + 1$</code>	$6n + t + 11$, dove t è il numero di confronti e $\min\{n_1, n_2\} \leq t \leq n - 1$
<code>RecursiveBinarySearch($a, 1, n, m$)</code>	$S(0) = 2, S(n) = S(\lfloor n/2 \rfloor) + 5$ per $n \geq 1$
<code>BruteForceMaximumSubarray(a, n)</code>	$2n^2 + 7n + 6 + 3U$, dove U è il numero di aggiornamenti di <i>best</i>

Attenzione

Master theorem. Sia $S(n) = a S(n/b) + f(n)$, con $a \geq 1$ e $b > 1$ costanti e $f(n)$ asintoticamente positiva.

1. Se $f(n) = O(n^{\log_b a - \varepsilon})$ per una qualche costante $\varepsilon > 0$, allora $S(n) = \Theta(n^{\log_b a})$.
2. Se $f(n) = \Theta(n^{\log_b a} \log^k n)$ per una qualche costante $k \geq 0$, allora $S(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. Se $f(n) = \Omega(n^{\log_b a + \varepsilon})$ per una qualche costante $\varepsilon > 0$, e $a f(n/b) \leq c f(n)$ per una qualche costante $c < 1$ e per ogni n sufficientemente grande, allora $S(n) = \Theta(f(n))$.

Esercizio 17.1 — Algoritmi ricorsivi (E1)

Consideriamo l'algoritmo ricorsivo `RecursiveSum`:

Algorithm: RecursiveSum**Input** : Un array a e due indici $l \leq r$ **Output**: La somma $a[l] + a[l+1] + \dots + a[r]$

```

1 function RecursiveSum( $a, l, r$ )
2   if  $l = r$  then
3     return  $a[l]$ 
4   end
5    $m := \lfloor (l + r)/2 \rfloor$ 
6    $s_1 := \text{RecursiveSum}(a, l, m)$ 
7    $s_2 := \text{RecursiveSum}(a, m + 1, r)$ 
8   return  $s_1 + s_2$ 
9 end

```

e l'array di $n = 9$ elementi

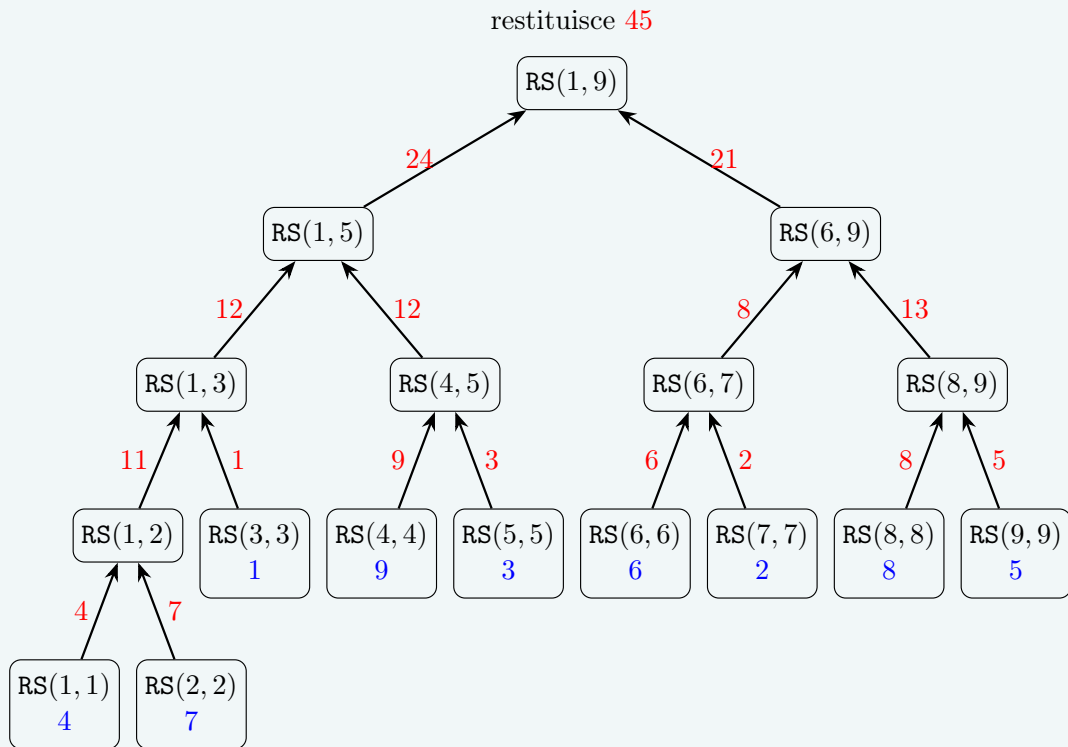
$$a = [4, 7, 1, 9, 3, 6, 2, 8, 5].$$

Domande

1. Disegnare l'albero delle chiamate di **RecursiveSum**($a, 1, 9$), indicando per ogni chiamata gli indici (l, r) e il valore restituito al padre.
2. Elencare le chiamate nell'ordine in cui **terminano**.
3. Quante chiamate vengono eseguite in totale? Quante addizioni? Quanti passi (contando ogni riga eseguita come un passo)? Confrontare con le formule generali $2n - 1$, $n - 1$ e $7n - 5$.
4. Qual è il massimo numero di chiamate contemporaneamente attive (la profondità della ricorsione)? Quali chiamate sono attive in quel momento? Confrontare con la formula $\lceil \log_2 n \rceil + 1$.

Soluzione

1. La radice **RS**(1, 9) calcola $m = \lfloor 10/2 \rfloor = 5$; la metà sinistra $a[1..5]$ ha $\lceil 9/2 \rceil = 5$ elementi e la metà destra $a[6..9]$ ha $\lfloor 9/2 \rfloor = 4$ elementi. Procedendo allo stesso modo per ogni chiamata, otteniamo il seguente albero delle chiamate (in **blu** gli elementi restituiti dai casi base, in **rosso** i valori restituiti ai padri):



I valori di m calcolati dalle chiamate interne sono:

chiamata	(1, 9)	(1, 5)	(1, 3)	(1, 2)	(4, 5)	(6, 9)	(6, 7)	(8, 9)
m	5	3	2	1	4	7	6	8
valore restituito	45	24	12	11	12	21	8	13

L'algoritmo restituisce $45 = 4 + 7 + 1 + 9 + 3 + 6 + 2 + 8 + 5$.

- Le chiamate vengono eseguite in profondità, e una chiamata termina solo dopo che sono terminati entrambi i suoi figli. L'ordine di terminazione è:

(1, 1), (2, 2), (1, 2), (3, 3), (1, 3), (4, 4), (5, 5), (4, 5), (1, 5),
(6, 6), (7, 7), (6, 7), (8, 8), (9, 9), (8, 9), (6, 9), (1, 9).

- L'albero delle chiamate ha 9 foglie (una per ogni elemento) e 8 nodi interni:

- numero di chiamate: $17 = 2 \cdot 9 - 1$;
- numero di addizioni: una per ogni nodo interno, cioè $8 = 9 - 1$;
- numero di passi: ogni foglia esegue 2 righe e ogni nodo interno 5 righe, quindi il totale è $9 \cdot 2 + 8 \cdot 5 = 58 = 7 \cdot 9 - 5$.

- L'albero delle chiamate ha altezza 4 (le foglie RS(1,1) e RS(2,2) sono al livello 4), quindi il massimo numero di chiamate contemporaneamente attive è 5. Per esempio, quando viene eseguita RS(1,1), le chiamate attive sono

RS(1,9), RS(1,5), RS(1,3), RS(1,2), RS(1,1).

Questo è in accordo con la formula: $\lceil \log_2 9 \rceil + 1 = 4 + 1 = 5$, poiché $2^3 = 8 < 9 \leq 16 = 2^4$.

Si noti che le foglie non sono tutte allo stesso livello: $\text{RS}(3, 3)$, per esempio, è al livello 3. Le foglie più profonde si raggiungono seguendo sempre la metà più grande (quella sinistra): $9 \rightarrow 5 \rightarrow 3 \rightarrow 2 \rightarrow 1$.

Esercizio 17.2 — Algoritmi ricorsivi (E2)

Consideriamo i tre algoritmi che calcolano la somma degli elementi di un array $a[1..n]$, $n \geq 1$, studiati nel capitolo «Algoritmi ricorsivi»:

- $\text{SumIterative}(a, n)$ (un ciclo);
- $\text{RecursiveSum}(a, l, n)$ (divide et impera, $m = \lfloor (l + r)/2 \rfloor$, caso base $l = r$);
- $\text{RecursiveSumLinear}(a, n)$ (decremento di uno: restituisce $\text{RecursiveSumLinear}(a, n - 1) + a[n]$, caso base $n = 1$).

Il loro numero di passi è richiamato nell'introduzione agli esercizi.

Domande

1. Per RecursiveSum , sia $A(n)$ il numero di addizioni eseguite su un array di n elementi. Scrivere un'equazione di ricorrenza per $A(n)$ e dimostrare per induzione forte che $A(n) = n - 1$.
2. Completare la seguente tabella, dove *chiamate* è il numero totale di chiamate (per SumIterative scrivere $-$) e *profondità* è il massimo numero di chiamate contemporaneamente attive:

	$n = 10$			$n = 100$			$n = 1000$		
	passi	chiamate	profondità	passi	chiamate	profondità	passi	chiamate	profondità
SumIterative									
RecursiveSum									
$\text{RecursiveSumLinear}$									

3. Consideriamo la seguente variante di RecursiveSum , che tratta come caso base anche i sotto-array di due elementi: se $r - l + 1 \leq 2$ restituisce $a[l]$ (se $l = r$) oppure $a[l] + a[r]$ (se $r = l + 1$); altrimenti procede come RecursiveSum . Quante chiamate e quante addizioni esegue per $n = 8$? E per $n = 2^k$, $k \geq 1$?
4. Spiegare perché *ogni* algoritmo che calcola la somma di n numeri sommandoli due alla volta esegue esattamente $n - 1$ addizioni.

Soluzione

1. Una chiamata su un elemento non esegue alcuna addizione; una chiamata su $n \geq 2$ elementi esegue un'addizione ($s_1 + s_2$) più le addizioni delle sue due chiamate ricorsive:

$$A(n) = \begin{cases} 0 & \text{se } n = 1, \\ A(\lceil n/2 \rceil) + A(\lfloor n/2 \rfloor) + 1 & \text{se } n \geq 2. \end{cases}$$

Caso base ($n = 1$): $A(1) = 0 = 1 - 1$.

Passo induttivo ($n \geq 2$): supponiamo $A(k) = k - 1$ per ogni $1 \leq k < n$. Poiché $1 \leq \lfloor n/2 \rfloor \leq \lceil n/2 \rceil < n$,

$$A(n) = (\lceil n/2 \rceil - 1) + (\lfloor n/2 \rfloor - 1) + 1 = \lceil n/2 \rceil + \lfloor n/2 \rfloor - 1 = n - 1.$$

2. Usiamo le formule del capitolo «Algoritmi ricorsivi» (si veda anche l'introduzione agli esercizi):

- **SumIterative**: $3n + 4$ passi, nessuna chiamata ricorsiva;
- **RecursiveSum**: $7n - 5$ passi, $2n - 1$ chiamate, profondità $\lceil \log_2 n \rceil + 1$;
- **RecursiveSumLinear**: $3n - 1$ passi, n chiamate (su $n, n - 1, \dots, 1$ elementi), profondità n (tutte le chiamate sono attive quando si raggiunge il caso base).

Poiché $2^3 < 10 \leq 2^4$, $2^6 < 100 \leq 2^7$ e $2^9 < 1000 \leq 2^{10}$, si ha $\lceil \log_2 10 \rceil = 4$, $\lceil \log_2 100 \rceil = 7$ e $\lceil \log_2 1000 \rceil = 10$. Quindi:

	$n = 10$			$n = 100$			$n = 1000$		
	passi	chiamate	profondità	passi	chiamate	profondità	passi	chiamate	profondità
SumIterative	34	—	—	304	—	—	3004	—	—
RecursiveSum	65	19	5	695	199	8	6995	1999	11
RecursiveSumLinear	29	10	10	299	100	100	2999	1000	1000

Tutti e tre gli algoritmi eseguono $\Theta(n)$ passi, ma la memoria aggiuntiva (proporzionale alla profondità) è rispettivamente $\Theta(1)$, $\Theta(\log_2 n)$ e $\Theta(n)$.

3. Per $n = 8$ i sotto-array hanno dimensioni $8 \rightarrow 4, 4 \rightarrow 2, 2, 2, 2$: le chiamate su 2 elementi sono casi base. L'albero delle chiamate ha $1 + 2 + 4 = 7$ chiamate: 3 chiamate interne (una addizione ciascuna) e 4 casi base su due elementi (una addizione ciascuno), per un totale di $3 + 4 = 7$ addizioni.

Per $n = 2^k$, l'albero ha livelli $0, 1, \dots, k - 1$, con 2^i chiamate al livello i su sotto-array di 2^{k-i} elementi; le chiamate al livello $k - 1$ sono i casi base su 2 elementi. Il numero di chiamate è

$$\sum_{i=0}^{k-1} 2^i = 2^k - 1 = n - 1$$

(circa la metà delle $2n - 1$ chiamate di **RecursiveSum**). Il numero di addizioni è ancora $(2^{k-1} - 1) + 2^{k-1} = 2^k - 1 = n - 1$: una per ciascuna delle $2^{k-1} - 1$ chiamate interne e una per ciascuno dei 2^{k-1} casi base.

4. Ogni addizione prende due numeri (elementi dell'array o somme parziali) e li sostituisce con un unico numero (la loro somma). All'inizio ci sono n numeri, alla fine ce n'è uno solo (la somma totale), e ogni addizione diminuisce il conteggio esattamente di uno. Quindi servono esattamente $n - 1$ addizioni, indipendentemente dall'ordine in cui vengono eseguite (iterativo, divide et impera, decremento di uno, ...).

Esercizio 17.3 — Algoritmi ricorsivi (E3)

Dato un array $a[1..n]$ di $n \geq 1$ numeri interi, consideriamo il problema di determinare il valore massimo nell'array. Vogliamo risolverlo con un algoritmo ricorsivo **divide et impera** $\text{RecursiveMax}(a, l, r)$, seguendo lo stesso schema di $\text{RecursiveSum}(a, l, r)$, e applicarlo all'array

$$a = [5, 17, 2, 11, 17, 8].$$

Domande

1. Scrivere lo pseudocodice di $\text{RecursiveMax}(a, l, r)$, che restituisce il massimo di $a[l..r]$ (caso base $l = r$, indice centrale $m = \lfloor (l + r)/2 \rfloor$).
2. Disegnare l'albero delle chiamate di $\text{RecursiveMax}(a, 1, 6)$ per l'array dato, con il valore restituito da ogni chiamata. Quante chiamate e quanti confronti tra elementi vengono eseguiti?
3. Dimostrare per induzione forte che $\text{RecursiveMax}(a, l, r)$ restituisce $\max\{a[l], \dots, a[r]\}$.
4. Scrivere l'equazione di ricorrenza per il numero di passi $S(n)$ di RecursiveMax e risolverla con il Master theorem. Poi calcolare $S(n)$ esattamente.
5. Determinare il tempo di esecuzione nel caso peggiore $T(|I|)$, supponendo che ogni elemento usi un numero costante di bit, e confrontare l'algoritmo con l'algoritmo iterativo FindMaximum del capitolo «Tempo di esecuzione».

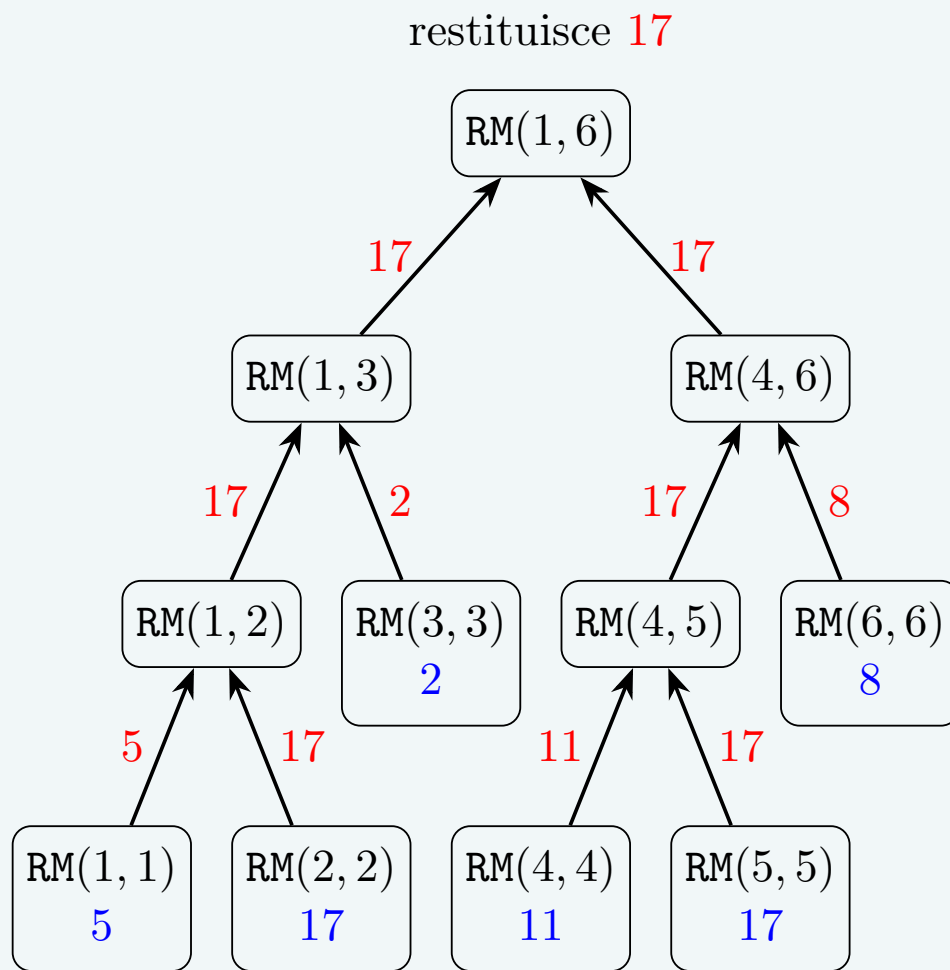
Soluzione

1. Il massimo di $a[l..r]$ è il maggiore tra i massimi delle due metà:

Algorithm: RecursiveMax**Input** : Un array a e due indici $l \leq r$ **Output:** Il valore massimo in $a[l..r]$

```
1 function RecursiveMax( $a, l, r$ )
2   if  $l = r$  then
3     return  $a[l]$ 
4   end
5    $m := \lfloor (l + r) / 2 \rfloor$ 
6    $x_1 := \text{RecursiveMax}(a, l, m)$ 
7    $x_2 := \text{RecursiveMax}(a, m + 1, r)$ 
8   if  $x_1 \geq x_2$  then
9     return  $x_1$ 
10  else
11    return  $x_2$ 
12  end
13 end
```

2. La radice calcola $m = \lfloor 7/2 \rfloor = 3$. Questo è l'albero delle chiamate, in cui $\text{RM}(l, r)$ indica la chiamata $\text{RecursiveMax}(a, l, r)$ e i valori restituiti sono in rosso:



I confronti eseguiti alla riga 8 sono:

chiamata	x_1	x_2	valore restituito
RM(1, 2)	5	17	$5 < 17 \Rightarrow 17$
RM(1, 3)	17	2	$17 \geq 2 \Rightarrow 17$
RM(4, 5)	11	17	$11 < 17 \Rightarrow 17$
RM(4, 6)	17	8	$17 \geq 8 \Rightarrow 17$
RM(1, 6)	17	17	$17 \geq 17 \Rightarrow 17$

Ci sono $11 = 2 \cdot 6 - 1$ chiamate e $5 = 6 - 1$ confronti (uno per ogni chiamata interna). Il massimo è 17.

3. Usiamo l'induzione forte su $k = r - l + 1 \geq 1$.

Caso base ($k = 1$): $l = r$ e l'algoritmo restituisce $a[l] = \max\{a[l]\}$.

Passo induttivo ($k \geq 2$): supponiamo che l'enunciato valga per tutti i sotto-array con meno di k elementi. Le due metà $a[l..m]$ e $a[m+1..r]$ hanno $\lceil k/2 \rceil$ e $\lfloor k/2 \rfloor$ elementi, entrambi compresi tra 1 e $k-1$; per l'ipotesi induttiva

$$x_1 = \max\{a[l], \dots, a[m]\}, \quad x_2 = \max\{a[m+1], \dots, a[r]\}.$$

Ogni elemento di $a[l..r]$ appartiene a una delle due metà, quindi è al più $\max\{x_1, x_2\}$; inoltre $\max\{x_1, x_2\}$ è un elemento di $a[l..r]$. Quindi $\max\{a[l], \dots, a[r]\} = \max\{x_1, x_2\}$, che è esattamente il valore restituito dalle righe 8–11.

4. Se $n = 1$, vengono eseguite le righe 2 e 3: $S(1) = 2$. Se $n \geq 2$:

- Le righe 2, 5, 8 vengono eseguite 1 volta ciascuna
- La riga 6 viene eseguita 1 volta, più i $S(\lceil n/2 \rceil)$ passi della chiamata
- La riga 7 viene eseguita 1 volta, più i $S(\lfloor n/2 \rfloor)$ passi della chiamata
- una sola delle righe 9 e 11 viene eseguita 1 volta

Quindi

$$S(n) = \begin{cases} 2 & \text{se } n = 1, \\ S(\lceil n/2 \rceil) + S(\lfloor n/2 \rfloor) + 6 & \text{se } n \geq 2, \end{cases} \quad \text{cioè} \quad S(n) = 2S(n/2) + c.$$

Master theorem:

- $a = 2$, $b = 2$, $f(n) = c$;
- $n^{\log_b a} = n^{\log_2 2} = n$;
- $f(n) = c$ cresce polinomialmente più lentamente di $n \rightarrow$ caso 1;
- $f(n) = c = O(n^{1-\varepsilon})$ con $\varepsilon = 1$;
- $S(n) = \Theta(n^{\log_2 2}) = \Theta(n)$.

Valore esatto: come per **RecursiveSum**, l'albero delle chiamate ha n foglie (costo 2 ciascuna) e $n - 1$ nodi interni (costo 6 ciascuno), quindi

$$S(n) = 2n + 6(n - 1) = 8n - 6.$$

(Si può verificare per induzione forte: $S(1) = 2 = 8 - 6$ e $S(n) = (8\lceil n/2 \rceil - 6) + (8\lfloor n/2 \rfloor - 6) + 6 = 8n - 6$.) Per l'array dato, $S(6) = 42$.

5. Poiché $|I| = \Theta(n)$ e $S(n) = 8n - 6 = \Theta(n)$, esattamente come nel capitolo «Algoritmi ricorsivi» otteniamo

Il tempo di esecuzione nel caso peggiore di **RecursiveMax** è $T(|I|) = \Theta(|I|)$.

FindMaximum esegue $4n - 1$ passi nel caso peggiore: entrambi gli algoritmi sono lineari ed eseguono $n - 1$ confronti tra elementi. L'algoritmo ricorsivo esegue più passi ($8n - 6$) e usa $\Theta(\log_2 n)$ di memoria aggiuntiva per la pila delle chiamate, contro $\Theta(1)$ di **FindMaximum**.

Esercizio 17.4 — Algoritmi ricorsivi (E4)

Consideriamo il seguente algoritmo ricorsivo, che suddivide il sotto-array $a[l..r]$ in **tre** parti:

Algorithm: RecursiveSum3**Input** : Un array a e due indici $l \leq r$ **Output**: La somma $a[l] + a[l+1] + \dots + a[r]$

```

1 function RecursiveSum3( $a, l, r$ )
2   if  $r - l + 1 \leq 2$  then
3     if  $l = r$  then return  $a[l]$ 
4     return  $a[l] + a[r]$ 
5   end
6    $s := \lfloor (r - l + 1)/3 \rfloor$ 
7    $m_1 := l + s - 1$ 
8    $m_2 := m_1 + s$ 
9    $x_1 := \text{RecursiveSum3}(a, l, m_1)$ 
10   $x_2 := \text{RecursiveSum3}(a, m_1 + 1, m_2)$ 
11   $x_3 := \text{RecursiveSum3}(a, m_2 + 1, r)$ 
12  return  $x_1 + x_2 + x_3$ 
13 end

```

e l'array di $n = 9$ elementi $a = [2, 7, 1, 8, 2, 8, 1, 8, 2]$.

Domande

1. Mostrare che, per $n = r - l + 1 \geq 3$, le tre parti sono non vuote e hanno meno di n elementi. Quali sono le loro dimensioni?
2. Disegnare l'albero delle chiamate di $\text{RecursiveSum3}(a, 1, 9)$ con il valore restituito da ogni chiamata.
3. Scrivere l'equazione di ricorrenza per il numero di passi $S(n)$ e risolverla con il Master theorem.
4. Supponiamo $n = 3^k$, $k \geq 1$. Calcolare il numero di chiamate, il numero di addizioni e la profondità della ricorsione, e confrontarli con quelli di RecursiveSum .

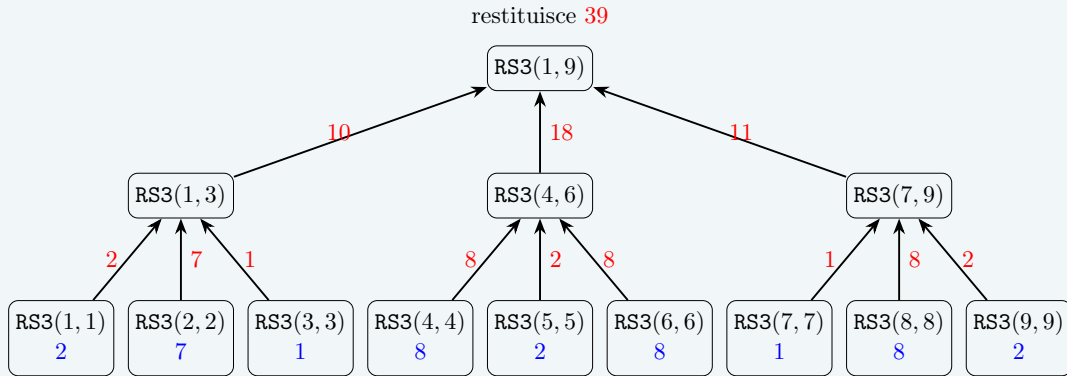
Soluzione

1. Siano $n = r - l + 1 \geq 3$ e $s = \lfloor n/3 \rfloor \geq 1$. Le tre parti sono

$$a[l..m_1] \text{ (} s \text{ elementi),} \quad a[m_1+1..m_2] \text{ (} s \text{ elementi),} \quad a[m_2+1..r] \text{ (} n-2s \text{ elementi).}$$

Poiché $s \geq 1$, le prime due parti sono non vuote e hanno $s \leq n/3 < n$ elementi. Scrivendo $n = 3s + \rho$ con $\rho \in \{0, 1, 2\}$, la terza parte ha $n - 2s = s + \rho$ elementi, con $1 \leq s \leq s + \rho < 3s + \rho = n$. Quindi anche la terza parte è non vuota e più piccola di n , e tutte le dimensioni sono circa $n/3$ (differiscono al più di 2).

2. La radice ha $n = 9$, $s = 3$, $m_1 = 3$, $m_2 = 6$; ogni parte ha 3 elementi, che vengono suddivisi ancora in parti di un elemento:



Per esempio, $RS3(4,6)$ calcola $s = 1$, $m_1 = 4$, $m_2 = 5$ e restituisce $8 + 2 + 8 = 18$. L'algoritmo restituisce $10 + 18 + 11 = 39$, che è la somma degli elementi. Ci sono 13 chiamate e $4 \cdot 2 = 8$ addizioni.

3. Per $n \leq 2$, vengono eseguite le righe 2, 3 e (se $n = 2$) 4: $S(1) = 2$, $S(2) = 3$. Per $n \geq 3$, le righe 2, 6, 7, 8, 12 vengono eseguite una volta, e le righe 9, 10, 11 una volta ciascuna più i passi delle tre chiamate:

$$S(n) = S(s) + S(s) + S(n - 2s) + 8, \quad s = \lfloor n/3 \rfloor,$$

che, ignorando le parti intere inferiori, scriviamo come

$$S(n) = 3 S(n/3) + c.$$

Master theorem:

- $a = 3$, $b = 3$, $f(n) = c$;
- $n^{\log_b a} = n^{\log_3 3} = n$;
- $f(n) = c$ cresce polinomialmente più lentamente di $n \rightarrow$ caso 1;
- $f(n) = c = O(n^{1-\varepsilon})$ con $\varepsilon = 1$;
- $S(n) = \Theta(n^{\log_3 3}) = \Theta(n)$.

Come previsto, suddividere in tre parti invece che in due non cambia l'ordine di crescita: ogni elemento deve essere letto almeno una volta.

4. Per $n = 3^k$, ogni chiamata su 3^j elementi, $j \geq 1$, effettua tre chiamate su 3^{j-1} elementi, e le chiamate su $3^0 = 1$ elemento sono casi base. Il livello i dell'albero delle chiamate contiene 3^i chiamate, per $i = 0, 1, \dots, k$:

- **chiamate:** $\sum_{i=0}^k 3^i = \frac{3^{k+1} - 1}{2} = \frac{3n - 1}{2}$ (per $n = 9$: 13, come nella domanda 2), contro le $2n - 1$ di `RecursiveSum`;
- **addizioni:** ciascuna delle $\frac{3n-1}{2} - n = \frac{n-1}{2}$ chiamate interne esegue 2 addizioni, per un totale di $n-1$, come `RecursiveSum` (ogni algoritmo che somma due numeri alla volta richiede $n - 1$ addizioni);

- **profondità della ricorsione:** l'altezza dell'albero è $k = \log_3 n$, quindi la profondità è $\log_3 n + 1$ (per $n = 9$: 3), contro $\lceil \log_2 n \rceil + 1$ di `RecursiveSum`. Poiché $\log_3 n = \frac{\log_2 n}{\log_2 3} \approx 0.63 \log_2 n$, la profondità è minore di un fattore costante, ma è comunque $\Theta(\log n)$.

Esercizio 17.5 — Algoritmi ricorsivi (E5)

Consideriamo il seguente array di $n = 8$ numeri naturali:

$$a = [38, 27, 43, 3, 9, 82, 10, 15].$$

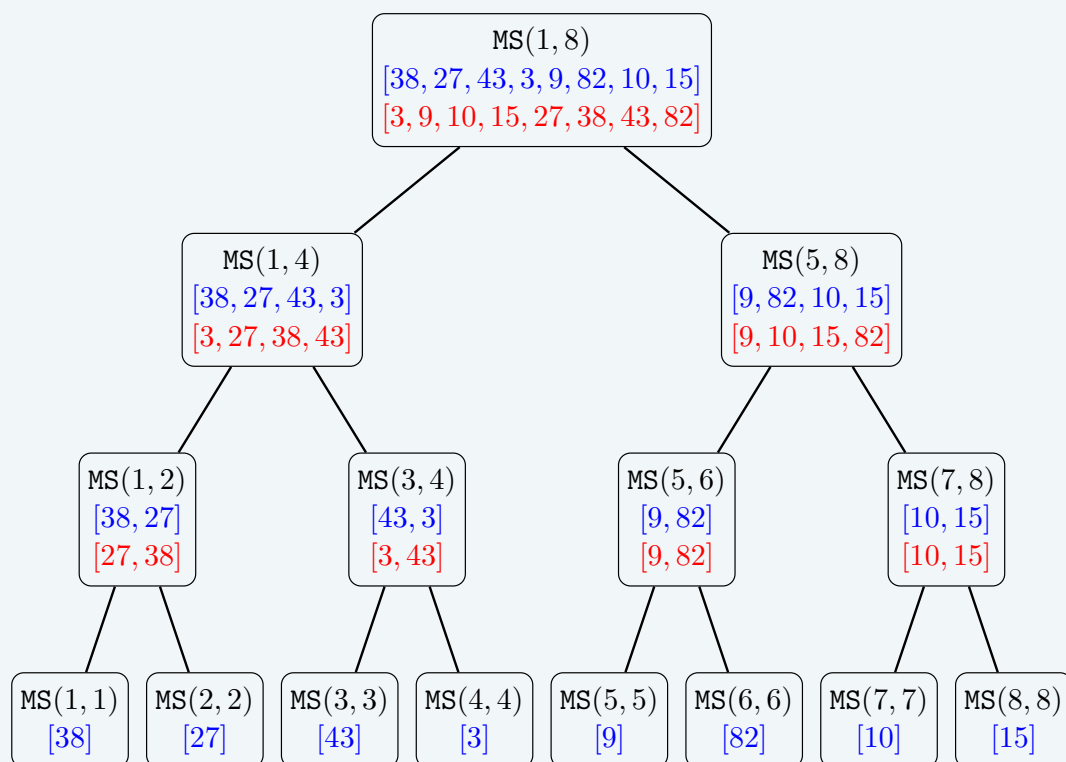
L'array viene ordinato dalla chiamata `MergeSort(a, 1, 8)` dell'algoritmo studiato nel capitolo «Algoritmi ricorsivi» (con $m = \lfloor (l + r)/2 \rfloor$ e la procedura `Merge(a, l, m, r)`).

Domande

1. Disegnare l'albero delle chiamate di `MergeSort(a, 1, 8)`, indicando per ogni chiamata gli indici (l, r) , il sotto-array ricevuto e il sotto-array ordinato prodotto.
2. Elencare le chiamate di `Merge` nell'ordine in cui vengono eseguite, e dare il contenuto dell'array a dopo ciascuna di esse.
3. Per ogni chiamata di `Merge`, dare il numero di confronti tra elementi (riga $L[i] \leq R[j]$). Qual è il numero totale di confronti?
4. Quante chiamate di `MergeSort` e quante chiamate di `Merge` vengono eseguite? Generalizzare a un array di $n = 2^k$ elementi.

Soluzione

1. Poiché $n = 8$ è una potenza di 2, ogni sotto-array viene suddiviso in due metà della stessa dimensione. In ogni nodo, in **blu** il sotto-array ricevuto e in **rosso** il sotto-array ordinato prodotto:



2. Le chiamate di **Merge** vengono eseguite in profondità, dopo le due chiamate ricorsive del corrispondente MergeSort (in **rosso** il sotto-array modificato):

chiamata	sotto-array fusi	array a dopo la chiamata
Merge(a , 1, 1, 2)	[38], [27]	[27, 38 , 43, 3, 9, 82, 10, 15]
Merge(a , 3, 3, 4)	[43], [3]	[27, 38, 3, 43 , 9, 82, 10, 15]
Merge(a , 1, 2, 4)	[27, 38], [3, 43]	[3, 27, 38, 43 , 9, 82, 10, 15]
Merge(a , 5, 5, 6)	[9], [82]	[3, 27, 38, 43, 9, 82 , 10, 15]
Merge(a , 7, 7, 8)	[10], [15]	[3, 27, 38, 43, 9, 82, 10, 15]
Merge(a , 5, 6, 8)	[9, 82], [10, 15]	[3, 27, 38, 43, 9, 10, 15, 82]
Merge(a , 1, 4, 8)	[3, 27, 38, 43], [9, 10, 15, 82]	[3, 9, 10, 15, 27, 38, 43, 82]

3. I confronti di ciascuna Merge sono:

- Merge(a , 1, 1, 2): $38 \leq 27$? No. Poi si copia L . **1** confronto.
- Merge(a , 3, 3, 4): $43 \leq 3$? No. **1** confronto.
- Merge(a , 1, 2, 4), $L = [27, 38]$, $R = [3, 43]$: $27 \leq 3$? No; $27 \leq 43$? Sì; $38 \leq 43$? Sì; poi L è esaurito e si copia 43. **3** confronti.
- Merge(a , 5, 5, 6): $9 \leq 82$? Sì. **1** confronto.
- Merge(a , 7, 7, 8): $10 \leq 15$? Sì. **1** confronto.
- Merge(a , 5, 6, 8), $L = [9, 82]$, $R = [10, 15]$: $9 \leq 10$? Sì; $82 \leq 10$? No; $82 \leq 15$? No; poi R è esaurito e si copia 82. **3** confronti.
- Merge(a , 1, 4, 8), $L = [3, 27, 38, 43]$, $R = [9, 10, 15, 82]$:

k	i	j	$L[i]$	$R[j]$	azione
1	1	1	3	9	$3 \leq 9$: $a[1] \leftarrow 3$
2	2	1	27	9	$27 > 9$: $a[2] \leftarrow 9$
3	2	2	27	10	$27 > 10$: $a[3] \leftarrow 10$
4	2	3	27	15	$27 > 15$: $a[4] \leftarrow 15$
5	2	4	27	82	$27 \leq 82$: $a[5] \leftarrow 27$
6	3	4	38	82	$38 \leq 82$: $a[6] \leftarrow 38$
7	4	4	43	82	$43 \leq 82$: $a[7] \leftarrow 43$
8	5	4	–	82	L esaurito: $a[8] \leftarrow 82$

7 confronti.

Il numero totale di confronti è $1 + 1 + 3 + 1 + 1 + 3 + 7 = 17$. Ogni **Merge** di n' elementi esegue al più $n' - 1$ confronti; qui ogni **Merge** raggiunge questo massimo $(1, 1, 3, 1, 1, 3, 7)$, quindi questo array è un **caso peggiore** per il numero di confronti del Merge Sort con $n = 8$: $17 = 8 \log_2 8 - 8 + 1$.

4. L'albero delle chiamate ha 8 foglie e 7 nodi interni: ci sono $15 = 2 \cdot 8 - 1$ chiamate di **MergeSort** e $7 = 8 - 1$ chiamate di **Merge** (una per ogni nodo interno).

Per $n = 2^k$, il livello i dell'albero delle chiamate contiene 2^i chiamate, per $i = 0, 1, \dots, k$, quindi il numero di chiamate di **MergeSort** è $\sum_{i=0}^k 2^i = 2^{k+1} - 1 = 2n - 1$. Le foglie sono le n chiamate su un elemento (che non chiamano **Merge**), quindi il numero di chiamate di **Merge** è $2n - 1 - n = n - 1$.

Esercizio 17.6 — Algoritmi ricorsivi (E6)

Consideriamo l'array di $n = 9$ elementi

$$a = [2, 5, 8, 13, 20, 3, 4, 9, 21],$$

in cui $a[1..5]$ e $a[6..9]$ sono ordinati, e la chiamata **Merge**($a, 1, 5, 9$) della procedura **Merge** studiata nel capitolo «Algoritmi ricorsivi». Il numero di passi di **Merge** su n elementi è richiamato nell'introduzione agli esercizi (t è il numero di iterazioni del primo ciclo while).

Domande

1. Dare i valori di n_1 , n_2 , L e R .
2. Tracciare l'esecuzione di **Merge**($a, 1, 5, 9$) in una tabella con colonne k , i , j , $L[i]$, $R[j]$, confronto e azione.
3. Quanti confronti tra elementi vengono eseguiti? Quanti elementi vengono copiati dal primo ciclo while e quanti dagli altri due cicli? Quanti passi vengono eseguiti?
4. Ripetere la domanda 3 (senza la tabella) se la parte destra viene sostituita da $[22, 23, 24, 25]$ e da $[0, 1, 1, 1]$, cioè per $a = [2, 5, 8, 13, 20, 22, 23, 24, 25]$ e per $a = [2, 5, 8, 13, 20, 0, 1, 1, 1]$.

Soluzione

1. $n_1 = m - l + 1 = 5 - 1 + 1 = 5$ e $n_2 = r - m = 9 - 5 = 4$. I due cicli for di Merge copiano le due parti ordinate:

$$L = [2, 5, 8, 13, 20], \quad R = [3, 4, 9, 21].$$

2. Inizialmente $i = 1, j = 1, k = 1$. Una riga per ogni elemento copiato in a :

k	i	j	$L[i]$	$R[j]$	confronto	azione
1	1	1	2	3	$2 \leq 3$	$a[1] \leftarrow 2, i = 2$
2	2	1	5	3	$5 > 3$	$a[2] \leftarrow 3, j = 2$
3	2	2	5	4	$5 > 4$	$a[3] \leftarrow 4, j = 3$
4	2	3	5	9	$5 \leq 9$	$a[4] \leftarrow 5, i = 3$
5	3	3	8	9	$8 \leq 9$	$a[5] \leftarrow 8, i = 4$
6	4	3	13	9	$13 > 9$	$a[6] \leftarrow 9, j = 4$
7	4	4	13	21	$13 \leq 21$	$a[7] \leftarrow 13, i = 5$
8	5	4	20	21	$20 \leq 21$	$a[8] \leftarrow 20, i = 6$
9	6	4	–	21	L esaurito	$a[9] \leftarrow 21, j = 5$

Dopo l'8-a iterazione $i = 6 > n_1$: il primo ciclo while termina, il ciclo che copia il resto di L viene saltato, e il ciclo che copia il resto di R copia $R[4] = 21$. Il risultato è

$$a = [2, 3, 4, 5, 8, 9, 13, 20, 21].$$

3. Ci sono $t = 8$ confronti, che è il massimo valore possibile $n - 1 = 8$. Il primo ciclo while copia 8 elementi, il ciclo su L copia 0 elementi e il ciclo su R copia 1 elemento. Il numero di passi è

$$6n + t + 11 = 6 \cdot 9 + 8 + 11 = 73.$$

4. **Parte destra** $[22, 23, 24, 25]$. Tutti gli elementi di L sono minori di $R[1] = 22$: il primo ciclo while copia 2, 5, 8, 13, 20 con i confronti $2 \leq 22, 5 \leq 22, 8 \leq 22, 13 \leq 22, 20 \leq 22$, e si ferma perché L è esaurito. Il ciclo su R copia 22, 23, 24, 25. Quindi $t = 5$ confronti (5 elementi copiati dal primo ciclo, 4 dal ciclo su R) e $6 \cdot 9 + 5 + 11 = 70$ passi.

Parte destra $[0, 1, 1, 1]$. Tutti gli elementi di R sono minori di $L[1] = 2$: il primo ciclo while copia 0, 1, 1, 1 con i confronti $2 \leq 0, 2 \leq 1, 2 \leq 1, 2 \leq 1$ (tutti falsi), e si ferma perché R è esaurito. Il ciclo su L copia 2, 5, 8, 13, 20. Quindi $t = 4$ confronti (4 elementi copiati dal primo ciclo, 5 dal ciclo su L) e $6 \cdot 9 + 4 + 11 = 69$ passi. Questo è il minimo numero possibile di confronti, $\min\{n_1, n_2\} = 4$.

In tutti i casi il numero di passi è compreso tra $6n + 11 = 65$ e $7n + 10 = 73$: Merge è lineare nel numero di elementi, qualunque siano i loro valori.

Esercizio 17.7 — Algoritmi ricorsivi (E7)

Consideriamo la procedura $\text{Merge}(a, l, m, r)$ studiata nel capitolo «Algoritmi ricorsivi», che fonde i sotto-array ordinati $a[l..m]$ (n_1 elementi) e $a[m+1..r]$ (n_2 elementi), con $n_1, n_2 \geq 1$. Sia C il numero di **confronti tra elementi** ($L[i] \leq R[j]$) eseguiti da Merge , e sia $C_{\text{MS}}(n)$ il numero totale di confronti eseguiti da MergeSort su un array di n elementi.

Domande

1. Dimostrare che $\min\{n_1, n_2\} \leq C \leq n_1 + n_2 - 1$.
2. Mostrare che entrambi i limiti sono raggiunti: per $n_1 = n_2 = 4$, dare due sotto-array ordinati per i quali Merge effettua 4 confronti e due per i quali ne effettua 7.
3. Sia $n = 2^k$. Scrivere equazioni di ricorrenza per il minimo $B(n)$ e per il massimo $W(n)$ di $C_{\text{MS}}(n)$ su tutti gli array di n elementi, e risolverle.
4. Per $n = 8$, verificare le formule con gli array $[1, 2, 3, 4, 5, 6, 7, 8]$ e $[1, 5, 3, 7, 2, 6, 4, 8]$. Che cosa succede con l'array $[8, 7, 6, 5, 4, 3, 2, 1]$?

Soluzione

1. Ogni iterazione del primo ciclo while esegue esattamente un confronto e copia esattamente un elemento in a ; gli altri due cicli non eseguono confronti. Quindi $C = t$, il numero di iterazioni del primo ciclo while.
 - **Limite inferiore.** Il ciclo termina solo quando $i > n_1$ oppure $j > n_2$, cioè quando tutti gli elementi di L o tutti gli elementi di R sono stati copiati dal ciclo. Quindi $t \geq n_1$ oppure $t \geq n_2$, e in entrambi i casi $t \geq \min\{n_1, n_2\}$.
 - **Limite superiore.** Ogni iterazione aumenta i o j di uno, e prima dell'ultima iterazione valgono sia $i \leq n_1$ sia $j \leq n_2$. Quando il ciclo termina, esattamente uno dei due array è esaurito, mentre l'altro contiene ancora almeno un elemento (non era esaurito prima dell'ultima iterazione, e l'ultima iterazione ha copiato un elemento del primo array). Quindi il ciclo copia al più $n_1 + n_2 - 1$ elementi: $t \leq n_1 + n_2 - 1$.
2. **4 confronti:** tutti gli elementi di una parte sono minori di tutti gli elementi dell'altra, per esempio $L = [1, 2, 3, 4]$ e $R = [5, 6, 7, 8]$. I confronti sono $1 \leq 5, 2 \leq 5, 3 \leq 5, 4 \leq 5$; poi L è esaurito e $5, 6, 7, 8$ vengono copiati senza confronti.
7 confronti: gli elementi delle due parti si alternano, per esempio $L = [1, 3, 5, 7]$ e $R = [2, 4, 6, 8]$. I confronti sono

$$1 \leq 2, \quad 3 \leq 2, \quad 3 \leq 4, \quad 5 \leq 4, \quad 5 \leq 6, \quad 7 \leq 6, \quad 7 \leq 8,$$
 dopodiché L è esaurito e viene copiato senza confronti solo 8.
3. Sia $n = 2^k$. MergeSort esegue i confronti delle due chiamate ricorsive su $n/2$ elementi più quelli della Merge finale con $n_1 = n_2 = n/2$, e le chiamate su un elemento non eseguono alcun confronto.

- **Minimo.** Per la domanda 1, la **Merge** finale esegue almeno $n/2$ confronti, ed esattamente $n/2$ se le due metà sono, per esempio, una interamente minore dell'altra (cosa che accade se l'array è ordinato, a tutti i livelli della ricorsione). Quindi

$$B(1) = 0, \quad B(n) = 2B(n/2) + \frac{n}{2}.$$

- **Massimo.** La **Merge** finale esegue al più $n - 1$ confronti, ed esattamente $n - 1$ se gli elementi delle due metà ordinate si alternano. Quindi

$$W(1) = 0, \quad W(n) = 2W(n/2) + n - 1.$$

(Che il massimo venga raggiunto contemporaneamente a tutti i livelli è mostrato dall'esempio della domanda 4.)

Risolviamo le equazioni di ricorrenza con l'albero di ricorsione: il livello i , $i = 0, \dots, k-1$, ha 2^i chiamate su $n/2^i$ elementi.

$$B(n) = \sum_{i=0}^{k-1} 2^i \cdot \frac{n}{2^{i+1}} = \sum_{i=0}^{k-1} \frac{n}{2} = \frac{n}{2} \log_2 n,$$

$$W(n) = \sum_{i=0}^{k-1} 2^i \left(\frac{n}{2^i} - 1 \right) = kn - \sum_{i=0}^{k-1} 2^i = n \log_2 n - (2^k - 1) = n \log_2 n - n + 1.$$

In entrambi i casi $C_{MS}(n) = \Theta(n \log_2 n)$: anche il caso migliore di **MergeSort** è $\Theta(n \log n)$ (al contrario, il caso migliore di **InsertionSort** è lineare).

4. Per $n = 8$ ($k = 3$) le formule danno $B(8) = 4 \cdot 3 = 12$ e $W(8) = 24 - 8 + 1 = 17$.

- $[1, 2, 3, 4, 5, 6, 7, 8]$ (ordinato): le quattro fusioni di coppie eseguono 1 confronto ciascuna, le due fusioni di quadruple $([1, 2]$ con $[3, 4]$, $[5, 6]$ con $[7, 8])$ 2 ciascuna, e la fusione finale di $[1, 2, 3, 4]$ con $[5, 6, 7, 8]$ esegue 4 confronti: $4 \cdot 1 + 2 \cdot 2 + 4 = 12 = B(8)$.
- $[1, 5, 3, 7, 2, 6, 4, 8]$: le coppie $[1, 5]$, $[3, 7]$, $[2, 6]$, $[4, 8]$ sono già ordinate (1 confronto ciascuna); poi $[1, 5]$ e $[3, 7]$ si alternano, dando $[1, 3, 5, 7]$ con 3 confronti, e $[2, 6]$ e $[4, 8]$ si alternano, dando $[2, 4, 6, 8]$ con 3 confronti; infine $[1, 3, 5, 7]$ e $[2, 4, 6, 8]$ si alternano (7 confronti, si veda la domanda 2). Totale: $4 \cdot 1 + 2 \cdot 3 + 7 = 17 = W(8)$.
- $[8, 7, 6, 5, 4, 3, 2, 1]$ (invertito): ogni fusione combina due metà in cui tutti gli elementi di R sono minori di tutti gli elementi di L . Ogni fusione esegue $\min\{n_1, n_2\}$ confronti, esattamente come per l'array ordinato: $4 \cdot 1 + 2 \cdot 2 + 4 = 12 = B(8)$. L'array invertito, che è il caso peggiore di **InsertionSort**, è un caso migliore per il numero di confronti di **MergeSort**.

Esercizio 17.8 — Algoritmi ricorsivi (E8)

Consideriamo la seguente variante **a 3 vie** del Merge Sort, che suddivide $a[l..r]$ in tre parti e usa due volte la procedura **Merge** del capitolo «Algoritmi ricorsivi» (**Merge** funziona

correttamente anche quando la sua parte destra è vuota, cioè $n_2 = 0$):

Algorithm: MergeSort3

Input : Un array a e due indici l, r

Output: Il sotto-array $a[l..r]$ ordinato in ordine non decrescente

```

1 function MergeSort3( $a, l, r$ )
2   if  $l < r$  then
3      $m_1 := \lfloor (2l + r)/3 \rfloor$ 
4      $m_2 := \lfloor (l + 2r)/3 \rfloor$ 
5     MergeSort3( $a, l, m_1$ )
6     MergeSort3( $a, m_1 + 1, m_2$ )
7     MergeSort3( $a, m_2 + 1, r$ )
8     Merge( $a, l, m_1, m_2$ )
9     Merge( $a, l, m_2, r$ )
10  end
11 end
  
```

Consideriamo inoltre l'array $a = [24, 5, 17, 9, 31, 2, 12, 20, 7]$ di $n = 9$ elementi.

Domande

1. Applicare **MergeSort3**($a, 1, 9$) all'array dato: dare le tre parti della prima chiamata, le parti ordinate restituite dalle tre chiamate ricorsive, e il risultato di ciascuna delle due chiamate finali di **Merge**.
2. Spiegare che cosa fanno le righe 8 e 9, e mostrare che insieme eseguono $\Theta(n)$ passi, dove $n = r - l + 1$.
3. Scrivere l'equazione di ricorrenza per il numero di passi $S(n)$ di **MergeSort3** e risolverla con il Master theorem. **MergeSort3** è asintoticamente più veloce di **MergeSort**?
4. Sia $n = 3^k$. Confrontare il massimo numero di confronti tra elementi eseguiti a ciascun livello della ricorsione e il numero di livelli con quelli di **MergeSort**. Quale algoritmo esegue meno confronti nel caso peggiore, per n grande?

Soluzione

1. Nella prima chiamata $l = 1, r = 9$: $m_1 = \lfloor 11/3 \rfloor = 3$ e $m_2 = \lfloor 19/3 \rfloor = 6$. Le tre parti sono

$$a[1..3] = [24, 5, 17], \quad a[4..6] = [9, 31, 2], \quad a[7..9] = [12, 20, 7].$$

Ogni parte ha 3 elementi; per esempio, per $a[1..3]$: $m_1 = \lfloor 5/3 \rfloor = 1$ e $m_2 = \lfloor 7/3 \rfloor = 2$, quindi viene suddivisa in tre parti di un elemento. Le tre chiamate ricorsive restituiscono

$$a[1..3] = [5, 17, 24], \quad a[4..6] = [2, 9, 31], \quad a[7..9] = [7, 12, 20],$$

cioè $a = [5, 17, 24, 2, 9, 31, 7, 12, 20]$. Poi:

- $\text{Merge}(a, 1, 3, 6)$ fonde $[5, 17, 24]$ e $[2, 9, 31]$ in $a[1..6] = [2, 5, 9, 17, 24, 31]$; ora $a = [2, 5, 9, 17, 24, 31, 7, 12, 20]$;
 - $\text{Merge}(a, 1, 6, 9)$ fonde $[2, 5, 9, 17, 24, 31]$ e $[7, 12, 20]$: $a = [2, 5, 7, 9, 12, 17, 20, 24, 31]$.
2. Dopo le tre chiamate ricorsive, $a[l..m_1]$, $a[m_1 + 1..m_2]$ e $a[m_2 + 1..r]$ sono ordinati. La riga 8 fonde le prime due parti, in modo che $a[l..m_2]$ sia ordinato; la riga 9 fonde $a[l..m_2]$ con la terza parte, in modo che $a[l..r]$ sia ordinato.
- Merge su n' elementi esegue tra $6n' + 11$ e $7n' + 10$ passi. La prima chiamata lavora su $n' = m_2 - l + 1 \leq n$ elementi e la seconda su $n' = n$ elementi, quindi le due chiamate insieme eseguono almeno $6n + 11$ e al più $2(7n + 10) = 14n + 20$ passi: $\Theta(n)$.
3. Per $n \geq 2$, le righe 2–9 vengono eseguite una volta ciascuna (8 passi), più i passi di tre chiamate ricorsive su circa $n/3$ elementi, più i $\Theta(n)$ passi delle due chiamate di Merge :

$$S(n) = 3S(n/3) + \Theta(n).$$

Master theorem:

- $a = 3, b = 3, f(n) = \Theta(n)$;
- $n^{\log_b a} = n^{\log_3 3} = n$;
- $f(n)$ e $n^{\log_b a}$ crescono allo stesso modo $\rightarrow$ caso 2;
- $f(n) = \Theta(n) = \Theta(n^{\log_3 3} \log^0 n)$ con $k = 0$;
- $S(n) = \Theta(n^{\log_3 3} \log n) = \Theta(n \log n)$.

MergeSort3 non è asintoticamente più veloce di MergeSort: entrambi eseguono $\Theta(n \log n)$ passi. L'albero di ricorsione di **MergeSort3** ha solo $\log_3 n + 1$ livelli invece di $\log_2 n + 1$, ma

$$\log_3 n = \frac{\log_2 n}{\log_2 3},$$

e i due logaritmi differiscono solo per un fattore costante, che scompare all'interno di Θ .

4. Per $n = 3^k$, ogni chiamata su $n' = 3^j \geq 3$ elementi li suddivide in tre parti di $n'/3$ elementi. La prima **Merge** lavora su $2n'/3$ elementi ed esegue al più $2n'/3 - 1$ confronti, la seconda lavora su n' elementi ed esegue al più $n' - 1$ confronti: al più $\frac{5}{3}n' - 2$ confronti per chiamata. Sommando sulle 3^i chiamate del livello i (con $n' = n/3^i$), ogni livello esegue meno di $\frac{5}{3}n$ confronti, e ci sono $k = \log_3 n$ livelli con **Merge**. Nel caso peggiore:

- **MergeSort3:** circa $\frac{5}{3}n \log_3 n = \frac{5}{3 \log_2 3} n \log_2 n \approx 1.05 n \log_2 n$ confronti;
- **MergeSort:** circa $n \log_2 n$ confronti

(il valore esatto per **MergeSort** è $n \log_2 n - n + 1$ quando n è una potenza di 2). I livelli sono meno, ma ogni livello è più costoso: per n grande, **MergeSort3** esegue circa il 5% di confronti **in più** nel caso peggiore. Entrambi sono $\Theta(n \log n)$.

Esercizio 17.9 — Algoritmi ricorsivi (E9)

Per sotto-array piccoli **InsertionSort** è in pratica più veloce di **MergeSort**. Consideriamo il seguente algoritmo **ibrido**, in cui $k \geq 1$ è un parametro e **InsertionSort**(a, l, r) è l'Insertion Sort applicato al sotto-array $a[l..r]$ (esegue $\Theta(p^2)$ passi nel caso peggiore su p elementi):

Algorithm: HybridMergeSort

Input : Un array a , due indici $l \leq r$ e un parametro $k \geq 1$

Output: Il sotto-array $a[l..r]$ ordinato in ordine non decrescente

```

1 function HybridMergeSort( $a, l, r, k$ )
2   if  $r - l + 1 \leq k$  then
3     InsertionSort( $a, l, r$ )
4   else
5      $m := \lfloor (l + r) / 2 \rfloor$ 
6     HybridMergeSort( $a, l, m, k$ )
7     HybridMergeSort( $a, m + 1, r, k$ )
8     Merge( $a, l, m, r$ )
9   end
10 end

```

Domande

1. Applicare **HybridMergeSort**($a, 1, 8, 2$) all'array $a = [7, 2, 9, 4, 3, 8, 6, 1]$: elencare, in ordine, le chiamate delle procedure **InsertionSort** e **Merge**, con i loro risultati.
2. Sia $n = k \cdot 2^h$ per un numero naturale h . Mostrare che le chiamate di **InsertionSort** vengono eseguite su n/k sotto-array di esattamente k elementi, e che eseguono $\Theta(nk)$ passi in totale nel caso peggiore.
3. Mostrare che le chiamate di **Merge** eseguono $\Theta(n \log_2(n/k))$ passi in totale.
4. Concludere che il numero di passi di **HybridMergeSort** nel caso peggiore è

$$S(n) = \Theta(nk + n \log(n/k)),$$

e discutere i casi estremi $k = 1$ e $k = n$.

5. Qual è il massimo ordine di crescita di k , in funzione di n , per cui **HybridMergeSort** esegue ancora $\Theta(n \log n)$ passi?

Soluzione

1. Con $k = 2$, i sotto-array con al più 2 elementi vengono ordinati da **InsertionSort**. La chiamata su $a[1..8]$ ($m = 4$) chiama $a[1..4]$ ($m = 2$) e $a[5..8]$ ($m = 6$), che chiamano i sotto-array di due elementi. In ordine:

chiamata	sotto-array	risultato
InsertionSort ($a, 1, 2$)	[7, 2]	[2, 7]
InsertionSort ($a, 3, 4$)	[9, 4]	[4, 9]
Merge ($a, 1, 2, 4$)	[2, 7], [4, 9]	[2, 4, 7, 9]
InsertionSort ($a, 5, 6$)	[3, 8]	[3, 8]
InsertionSort ($a, 7, 8$)	[6, 1]	[1, 6]
Merge ($a, 5, 6, 8$)	[3, 8], [1, 6]	[1, 3, 6, 8]
Merge ($a, 1, 4, 8$)	[2, 4, 7, 9], [1, 3, 6, 8]	[1, 2, 3, 4, 6, 7, 8, 9]

Ci sono $4 = n/k$ chiamate di **InsertionSort** e 3 chiamate di **Merge**, contro le 7 chiamate di **Merge** di **MergeSort** su 8 elementi.

2. Sia $n = k \cdot 2^h$. Una chiamata su $k \cdot 2^j$ elementi con $j \geq 1$ ha più di k elementi, quindi suddivide il sotto-array in due metà di $k \cdot 2^{j-1}$ elementi ciascuna. Quindi l'albero di ricorsione ha livelli $0, 1, \dots, h$: il livello i contiene 2^i chiamate su $n/2^i = k \cdot 2^{h-i}$ elementi, e le chiamate del livello h , su esattamente k elementi, sono le foglie, che chiamano **InsertionSort**. Ci sono $2^h = n/k$ foglie.

Nel caso peggiore **InsertionSort** su k elementi esegue $\frac{3k^2+9k-8}{2} = \Theta(k^2)$ passi (si veda il capitolo «Tempo di esecuzione»). Questo accade su ogni foglia contemporaneamente, per esempio se l'intero array è ordinato in ordine decrescente (allora ogni sotto-array delle foglie è in ordine decrescente). In totale:

$$\frac{n}{k} \cdot \Theta(k^2) = \Theta(nk).$$

3. Le chiamate di **Merge** sono eseguite dai nodi interni, cioè ai livelli $0, 1, \dots, h-1$. Al livello i ci sono 2^i chiamate di **Merge**, ciascuna su $n/2^i$ elementi. Poiché **Merge** su n' elementi esegue tra $6n' + 11$ e $7n' + 10$ passi, le chiamate al livello i eseguono tra

$$2^i \left(6 \frac{n}{2^i} + 11 \right) \geq 6n \quad \text{e} \quad 2^i \left(7 \frac{n}{2^i} + 10 \right) = 7n + 10 \cdot 2^i \leq 17n$$

passi (poiché $2^i \leq 2^h = n/k \leq n$). Ogni livello costa $\Theta(n)$ e ci sono $h = \log_2(n/k)$ livelli con **Merge**, quindi il totale è

$$\Theta(n \log_2(n/k)).$$

4. I passi di **HybridMergeSort** sono quelli delle foglie (domanda 2), più quelli dei nodi interni (domanda 3), più un numero costante di passi (righe 2–8) per ciascuna delle $2^{h+1} - 1 < 2n$ chiamate, che è $O(n)$ e viene assorbito dagli altri termini. Nel caso peggiore,

$$S(n) = \Theta(nk) + \Theta(n \log(n/k)) = \Theta(nk + n \log(n/k)).$$

- $k = 1$: le foglie contengono un elemento, **InsertionSort** non fa nulla di significativo e l'algoritmo è **MergeSort**: $S(n) = \Theta(n + n \log n) = \Theta(n \log n)$.
- $k = n$: c'è una sola foglia, l'intero array viene ordinato da **InsertionSort** e non c'è alcuna fusione: $S(n) = \Theta(n^2 + n \log 1) = \Theta(n^2)$.

5. Poiché $\log(n/k) \leq \log n$, si ha sempre $n \log(n/k) = O(n \log n)$. Quindi $S(n) = \Theta(n \log n)$ se e solo se, inoltre, $nk = O(n \log n)$, cioè

$$k = O(\log n).$$

Infatti, se $k = \Theta(\log n)$, allora $nk = \Theta(n \log n)$ e $n \log(n/k) = n \log n - n \log k \leq n \log n$, quindi $S(n) = \Theta(n \log n)$. Se k cresce più velocemente di $\log n$ (per esempio $k = \sqrt{n}$), il termine nk domina e $S(n)$ cresce più velocemente di $n \log n$.

Quindi il massimo ordine di crescita ammissibile è $k = \Theta(\log n)$. In pratica, k si sceglie sperimentalmente (valori tipici sono tra 8 e 32), come la dimensione massima per cui, sul calcolatore dato, l'Insertion Sort è più veloce del Merge Sort.

Esercizio 17.10 — Algoritmi ricorsivi (E10)

Consideriamo il seguente array ordinato di $n = 10$ elementi:

$$a = [1, 4, 7, 10, 13, 17, 21, 25, 30, 34]$$

e l'algoritmo `RecursiveBinarySearch(a, left, right, m)`, con prima chiamata `RecursiveBinarySearch(a, 1, 10, m)` e indice centrale

$$mid = left + \left\lfloor \frac{right - left}{2} \right\rfloor.$$

Domande

1. Applicare l'algoritmo di ricerca binaria ricorsiva per i seguenti valori di m :

- (a) $m = 21$
- (b) $m = 1$
- (c) $m = 34$
- (d) $m = 15$
- (e) $m = 40$

Per ogni ricerca, riportare una riga per ogni chiamata, indicando i valori di $left$, $right$, mid , $a[mid]$ e l'azione eseguita, e dare il valore restituito.

2. Disegnare l'albero delle chiamate per $m = 15$, indicando i valori restituiti da ogni chiamata.
3. Per ogni ricerca, contare il numero di chiamate su un intervallo non vuoto e il numero totale di chiamate. Confrontare questi numeri con il limite $\lfloor \log_2 n \rfloor + 1$ sul numero di chiamate su un intervallo non vuoto.

Soluzione

1. (a) Ricerca di $m = 21$, prima chiamata `RecursiveBinarySearch(a, 1, 10, 21)`:

chiamata	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	azione
1	1	10	5	13	$a[mid] < m \Rightarrow$ chiama $left = 6, right = 10$
2	6	10	8	25	$a[mid] > m \Rightarrow$ chiama $left = 6, right = 7$
3	6	7	6	17	$a[mid] < m \Rightarrow$ chiama $left = 7, right = 7$
4	7	7	7	21	$a[mid] = m \Rightarrow$ Trovato! Ritorna 7

Il valore 7 viene restituito lungo le chiamate: l'elemento 21 viene trovato all'indice 7.

- (b) Ricerca di $m = 1$, prima chiamata `RecursiveBinarySearch(a, 1, 10, 1)`:

chiamata	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	azione
1	1	10	5	13	$a[mid] > m \Rightarrow$ chiama $left = 1, right = 4$
2	1	4	2	4	$a[mid] > m \Rightarrow$ chiama $left = 1, right = 1$
3	1	1	1	1	$a[mid] = m \Rightarrow$ Trovato! Ritorna 1

L'elemento 1 viene trovato all'indice 1.

- (c) Ricerca di $m = 34$, prima chiamata `RecursiveBinarySearch(a, 1, 10, 34)`:

chiamata	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	azione
1	1	10	5	13	$a[mid] < m \Rightarrow$ chiama $left = 6, right = 10$
2	6	10	8	25	$a[mid] < m \Rightarrow$ chiama $left = 9, right = 10$
3	9	10	9	30	$a[mid] < m \Rightarrow$ chiama $left = 10, right = 10$
4	10	10	10	34	$a[mid] = m \Rightarrow$ Trovato! Ritorna 10

L'elemento 34 viene trovato all'indice 10.

- (d) Ricerca di $m = 15$, prima chiamata `RecursiveBinarySearch(a, 1, 10, 15)`:

chiamata	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	azione
1	1	10	5	13	$a[mid] < m \Rightarrow$ chiama $left = 6, right = 10$
2	6	10	8	25	$a[mid] > m \Rightarrow$ chiama $left = 6, right = 7$
3	6	7	6	17	$a[mid] > m \Rightarrow$ chiama $left = 6, right = 5$
4	6	5	-	-	$left > right \Rightarrow$ Non trovato! Ritorna -1

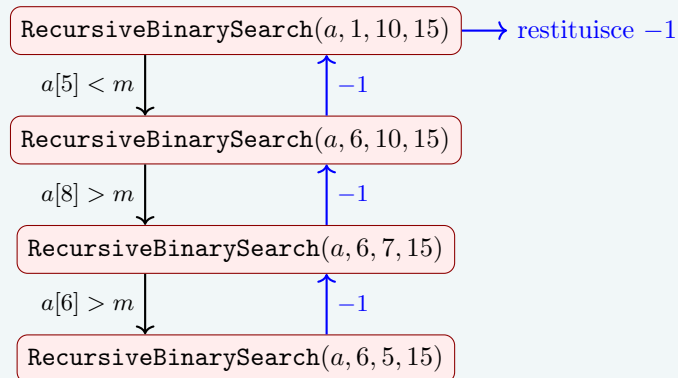
L'elemento 15 non è nell'array (restituisce -1).

- (e) Ricerca di $m = 40$, prima chiamata `RecursiveBinarySearch(a, 1, 10, 40)`:

chiamata	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	azione
1	1	10	5	13	$a[mid] < m \Rightarrow$ chiama $left = 6, right = 10$
2	6	10	8	25	$a[mid] < m \Rightarrow$ chiama $left = 9, right = 10$
3	9	10	9	30	$a[mid] < m \Rightarrow$ chiama $left = 10, right = 10$
4	10	10	10	34	$a[mid] < m \Rightarrow$ chiama $left = 11, right = 10$
5	11	10	-	-	$left > right \Rightarrow$ Non trovato! Ritorna -1

L'elemento 40 non è nell'array (restituisce -1).

2. Ogni chiamata effettua al più una chiamata ricorsiva, quindi l'albero delle chiamate è un cammino. Per $m = 15$ (freccie nere: chiamate ricorsive; frecce blu: valori restituiti):



L'ultima chiamata è sull'intervallo vuoto $left = 6$, $right = 5$ e restituisce -1 ; ogni altra chiamata restituisce, invariato, il valore restituito dalla sua chiamata ricorsiva.

3. Per $n = 10$ si ha $\lfloor \log_2 10 \rfloor + 1 = 3 + 1 = 4$, poiché $2^3 = 8 \leq 10 < 16 = 2^4$.

m	chiamate su un intervallo non vuoto	chiamate totali
21	4	4
1	3	3
34	4	4
15	3	4
40	4	5

In ogni ricerca il numero di chiamate su un intervallo non vuoto è al più 4, come previsto dal limite. Per $m = 40 > a[10]$ l'algoritmo prosegue sempre nella metà destra, il limite 4 viene raggiunto, e segue ancora una chiamata sull'intervallo vuoto: il numero totale di chiamate è $\lfloor \log_2 10 \rfloor + 2 = 5$, che è il massimo possibile per $n = 10$.

Esercizio 17.11 — Algoritmi ricorsivi (E11)

Consideriamo l'algoritmo `RecursiveBinarySearch(a, left, right, m)` applicato a un array ordinato $a[1..n]$, con prima chiamata `RecursiveBinarySearch(a, 1, n, m)`. Il suo numero di passi nel caso peggiore $S(n)$ soddisfa l'equazione di ricorrenza richiamata nell'introduzione agli esercizi.

Domande

1. Determinare il massimo numero di chiamate su un intervallo non vuoto e il massimo numero totale di chiamate (compresa l'ultima chiamata su un intervallo vuoto) per

un array di $n = 1000$ elementi. Dare un esempio di valore m per cui il massimo viene raggiunto, se $a[i] = 2i$ per $i = 1, \dots, 1000$.

2. Rispondere alle stesse domande per $n = 10^6$.
3. Calcolare $S(1000)$ e $S(10^6)$.
4. Determinare il più piccolo n per cui, nel caso peggiore, l'algoritmo effettua 21 chiamate su un intervallo non vuoto.
5. Qual è il massimo numero di chiamate contemporaneamente presenti nella pila delle chiamate per $n = 10^6$? Confrontare con il numero di elementi esaminati da una ricerca lineare nel caso peggiore.

Soluzione

1. Il massimo numero di chiamate su un intervallo non vuoto è $\lfloor \log_2 n \rfloor + 1$, e il massimo numero totale di chiamate è $\lfloor \log_2 n \rfloor + 2$. Per $n = 1000$:

$$2^9 = 512 \leq 1000 < 1024 = 2^{10} \implies \lfloor \log_2 1000 \rfloor = 9.$$

Quindi:

- massimo numero di chiamate su un intervallo non vuoto: $9 + 1 = 10$;
- massimo numero totale di chiamate: $9 + 2 = 11$.

Il massimo viene raggiunto, per esempio, da ogni $m > a[1000] = 2000$, ad esempio $m = 2001$: l'algoritmo prosegue sempre nella metà destra, le cui dimensioni sono

$$1000, 500, 250, 125, 62, 31, 15, 7, 3, 1, 0,$$

cioè 10 intervalli non vuoti seguiti da quello vuoto.

2. Per $n = 10^6$:

$$2^{19} = 524\,288 \leq 10^6 < 1\,048\,576 = 2^{20} \implies \lfloor \log_2 10^6 \rfloor = 19.$$

Quindi:

- massimo numero di chiamate su un intervallo non vuoto: $19 + 1 = 20$;
- massimo numero totale di chiamate: $19 + 2 = 21$.

Anche qui, il massimo viene raggiunto da ogni $m > a[n]$.

3. Per espansione, $S(n) = 5(\lfloor \log_2 n \rfloor + 1) + 2$ per $n \geq 1$. Quindi

$$S(1000) = 5 \cdot 10 + 2 = 52, \quad S(10^6) = 5 \cdot 20 + 2 = 102.$$

Moltiplicare per 1000 la dimensione dell'array fa solo raddoppiare (approssimativamente) il numero di passi.

4. Serve $\lfloor \log_2 n \rfloor + 1 = 21$, cioè $\lfloor \log_2 n \rfloor = 20$, cioè

$$2^{20} \leq n < 2^{21}.$$

Il più piccolo n di questo tipo è $n = 2^{20} = 1\,048\,576$. Infatti, per $n = 2^{20} - 1$ si ha $\lfloor \log_2 n \rfloor = 19$ e solo 20 chiamate su un intervallo non vuoto.

5. Nessuna chiamata termina prima che sia terminata la sua chiamata ricorsiva; quindi, quando è in esecuzione la chiamata più profonda, tutte le chiamate del cammino sono nella pila delle chiamate. Per $n = 10^6$ ci sono al più 21 chiamate contemporaneamente nella pila delle chiamate (memoria aggiuntiva $\Theta(\log n)$).

Una ricerca lineare esamina, nel caso peggiore, tutti i 10^6 elementi dell'array, mentre la ricerca binaria ricorsiva ne esamina al più 20 (uno per ogni chiamata su un intervallo non vuoto).

Esercizio 17.12 — Algoritmi ricorsivi (E12)

Consideriamo il seguente algoritmo di **ricerca ternaria**, che suddivide l'intervallo $a[\textit{left}..\textit{right}]$ in tre parti usando due indici m_1 e m_2 .

Algorithm: TernarySearch

Input : Un array $a[1..n]$ ordinato in ordine non decrescente, due indici $left$, $right$, un valore da cercare m

Output: Un indice i con $a[i] = m$, oppure -1 se m non è in $a[left..right]$

```

1 function TernarySearch( $a, left, right, m$ )
2   if  $left > right$  then
3     | return  $-1$ 
4   end
5    $d := \lfloor (right - left) / 3 \rfloor$ 
6    $m_1 := left + d$ 
7    $m_2 := right - d$ 
8   if  $a[m_1] = m$  then
9     | return  $m_1$ 
10  end
11  if  $a[m_2] = m$  then
12    | return  $m_2$ 
13  end
14  if  $m < a[m_1]$  then
15    | return TernarySearch( $a, left, m_1 - 1, m$ )
16  end
17  if  $m > a[m_2]$  then
18    | return TernarySearch( $a, m_2 + 1, right, m$ )
19  end
20  return TernarySearch( $a, m_1 + 1, m_2 - 1, m$ )
21 end

```

Domande

1. Sia $n = right - left + 1 \geq 1$. Mostrare che ciascuno dei tre sotto-intervalli su cui si può effettuare una chiamata ricorsiva ha al più $\lfloor n/3 \rfloor$ elementi, e scrivere un'equazione di ricorrenza per il numero di passi nel caso peggiore $S(n)$.
2. Risolvere l'equazione di ricorrenza $S(n) = S(n/3) + c$ con il Master theorem.
3. Mostrare che il numero di chiamate su un intervallo non vuoto è al più $\lfloor \log_3 n \rfloor + 1$. Calcolare questo limite per $n = 10^6$ e confrontarlo con il limite corrispondente per la ricerca binaria ricorsiva.
4. Ogni chiamata della ricerca binaria ricorsiva confronta m con elementi dell'array al più 2 volte ($a[mid] = m$ e $a[mid] < m$), mentre ogni chiamata della ricerca ternaria lo fa al più 4 volte. Confrontare il massimo numero di confronti dei due algoritmi per $n = 10^6$ e per $n \rightarrow \infty$. La ricerca ternaria è più veloce della ricerca binaria?

Soluzione

1. Siano $n = \text{right} - \text{left} + 1 \geq 1$ e $n - 1 = \text{right} - \text{left} = 3d + s$ con $d = \lfloor (n - 1)/3 \rfloor$ e $s \in \{0, 1, 2\}$. I tre sotto-intervalli hanno le seguenti dimensioni:

- $a[\text{left}..m_1 - 1]$: $m_1 - \text{left} = d$ elementi;
- $a[m_2 + 1..\text{right}]$: $\text{right} - m_2 = d$ elementi;
- $a[m_1 + 1..m_2 - 1]$: $m_2 - m_1 - 1 = (n - 1) - 2d - 1 = d + s - 1$ elementi (se questo numero è negativo, l'intervallo è vuoto).

Poiché $d = \lfloor (n - 1)/3 \rfloor \leq \lfloor n/3 \rfloor$, le prime due dimensioni sono al più $\lfloor n/3 \rfloor$. Per l'intervallo centrale:

- se $s \in \{0, 1\}$, allora $d + s - 1 \leq d \leq \lfloor n/3 \rfloor$;
- se $s = 2$, allora $n = 3d + 3$, quindi $\lfloor n/3 \rfloor = d + 1 = d + s - 1$.

Quindi ogni sotto-intervallo ha al più $\lfloor n/3 \rfloor$ elementi. Per esempio, per $n = 9$: $d = 2$, $s = 2$, e i tre sotto-intervalli hanno 2, 3, 2 elementi, con $\lfloor 9/3 \rfloor = 3$.

Ogni chiamata esegue un numero costante di passi oltre alla chiamata ricorsiva, e effettua al più una chiamata ricorsiva. Quindi, il numero di passi nel caso peggiore soddisfa

$$S(n) = S(\lfloor n/3 \rfloor) + c \quad (n \geq 1), \quad S(0) = c_0,$$

per certe costanti $c, c_0 > 0$.

2. Applichiamo il Master theorem a $S(n) = S(n/3) + c$:

- **Identificare:** $a = 1$, $b = 3$, $f(n) = c$.
- **Calcolare:** $n^{\log_b a} = n^{\log_3 1} = n^0 = 1$.
- **Confrontare:** $f(n) = c = \Theta(1) = \Theta(n^{\log_3 1} \log^0 n)$.
- **Verificare:** vale il caso 2 del Master theorem con $k = 0$.
- **Concludere:**

$$S(n) = \Theta(n^{\log_3 1} \log^1 n) = \Theta(\log n).$$

La ricerca ternaria ha lo stesso ordine di crescita della ricerca binaria: la base del logaritmo è irrilevante all'interno di Θ , poiché $\log_3 n = \log_2 n / \log_2 3$.

3. Sia n_i la dimensione dell'intervallo della i -esima chiamata ($n_1 = n$). Per la domanda 1, $n_{i+1} \leq \lfloor n_i/3 \rfloor$, e per induzione $n_i \leq \lfloor n/3^{i-1} \rfloor$ (usando $\lfloor \lfloor x/3^{i-1} \rfloor / 3 \rfloor = \lfloor x/3^i \rfloor$). Se la i -esima chiamata è su un intervallo non vuoto, allora $n_i \geq 1$, quindi $3^{i-1} \leq n$, cioè $i - 1 \leq \log_3 n$ e, poiché $i - 1$ è un intero,

$$i \leq \lfloor \log_3 n \rfloor + 1.$$

Per $n = 10^6$: $3^{12} = 531\,441 \leq 10^6 < 1\,594\,323 = 3^{13}$, quindi $\lfloor \log_3 10^6 \rfloor = 12$ e la ricerca ternaria effettua al più

$$12 + 1 = 13$$

chiamate su un intervallo non vuoto (e questo valore viene raggiunto per $n = 10^6$). La ricerca binaria ricorsiva effettua al più $\lfloor \log_2 10^6 \rfloor + 1 = 19 + 1 = 20$ di tali chiamate. La ricerca ternaria effettua meno chiamate.

4. Massimo numero di confronti con elementi dell'array per $n = 10^6$:

$$\text{ricerca binaria: } 2 \cdot 20 = 40, \quad \text{ricerca ternaria: } 4 \cdot 13 = 52.$$

Per $n \rightarrow \infty$ i due limiti si comportano come $2 \log_2 n$ e $4 \log_3 n$, e

$$\lim_{n \rightarrow \infty} \frac{4 \log_3 n}{2 \log_2 n} = \frac{4}{2 \log_2 3} = \frac{2}{\log_2 3} \approx 1.26.$$

Sebbene la ricerca ternaria effettui meno chiamate, ogni chiamata è più costosa, e in totale esegue circa il 26% di confronti in più della ricerca binaria nel caso peggiore. Entrambi gli algoritmi sono $\Theta(\log n)$: la ricerca ternaria **non** è asintoticamente più veloce, e in pratica è leggermente più lenta.

Esercizio 17.13 — Algoritmi ricorsivi (E13)

Consideriamo l'array ordinato con duplicati

$$a = [2, 4, 4, 4, 4, 7, 9, 9, 9, 12]$$

e il seguente algoritmo, la cui prima chiamata è `FirstOccurrence(a, 1, n, m)`.

Algorithm: FirstOccurrence

Input : Un array $a[1..n]$ ordinato in ordine non decrescente, due indici $left$, $right$, un valore da cercare m

Output: Il più piccolo indice $i \in \{left, \dots, right\}$ con $a[i] = m$, oppure -1 se m non è in $a[left..right]$

```

1 function FirstOccurrence(a, left, right, m)
2   if left > right then
3     return -1
4   end
5   mid := left + ⌊(right - left)/2⌋
6   if a[mid] < m then
7     return FirstOccurrence(a, mid + 1, right, m)
8   end
9   if a[mid] > m then
10    return FirstOccurrence(a, left, mid - 1, m)
11  end
12  r := FirstOccurrence(a, left, mid - 1, m)           // a[mid] = m
13  if r = -1 then
14    return mid
15  end
16  return r
17 end

```

Domande

1. Applicare `RecursiveBinarySearch(a, 1, 10, m)` per $m = 4$ e $m = 9$. Restituisce la prima occorrenza di m ?
2. Applicare `FirstOccurrence(a, 1, 10, m)` per $m = 4$, $m = 9$ e $m = 5$. Per ogni ricerca, riportare una riga per ogni chiamata e il valore restituito da ciascuna chiamata.
3. Dimostrare per induzione forte su $k = right - left + 1$ che `FirstOccurrence` è corretto.
4. Scrivere l'equazione di ricorrenza per il numero di passi nel caso peggiore $S(n)$ e risolverla.
5. Uno studente propone un algoritmo diverso: trovare un'occorrenza qualsiasi i di m con `RecursiveBinarySearch`, poi spostarsi a sinistra finché $i > 1$ e $a[i - 1] = m$. Determinare il suo numero di passi nel caso peggiore.
6. `FirstOccurrence` è ricorsivo in coda?

Soluzione

1. Ricerca di $m = 4$ con `RecursiveBinarySearch(a, 1, 10, 4)`:

chiamata	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	azione
1	1	10	5	4	$a[mid] = m \Rightarrow$ Trovato! Ritorna 5

Ricerca di $m = 9$ con `RecursiveBinarySearch(a, 1, 10, 9)`:

chiamata	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	azione
1	1	10	5	4	$a[mid] < m \Rightarrow$ chiama $left = 6$, $right = 10$
2	6	10	8	9	$a[mid] = m \Rightarrow$ Trovato! Ritorna 8

Gli indici restituiti sono 5 e 8, mentre le prime occorrenze sono all'indice 2 (per $m = 4$) e 7 (per $m = 9$). `RecursiveBinarySearch` restituisce **un'**occorrenza, non necessariamente la prima.

2. Ricerca di $m = 4$ con `FirstOccurrence(a, 1, 10, 4)`:

chiamata	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	azione	restituisce
1	1	10	5	4	$a[mid] = m \Rightarrow r :=$ chiama 1, 4	$r = 2 \Rightarrow 2$
2	1	4	2	4	$a[mid] = m \Rightarrow r :=$ chiama 1, 1	$r = -1 \Rightarrow mid = 2$
3	1	1	1	2	$a[mid] < m \Rightarrow$ chiama 2, 1	-1
4	2	1	-	-	$left > right$	-1

I valori restituiti si calcolano dal basso verso l'alto: la chiamata 4 restituisce -1, la chiamata 3 restituisce -1, la chiamata 2 riceve $r = -1$ e restituisce il proprio $mid = 2$, la chiamata 1 riceve $r = 2 \neq -1$ e restituisce 2. La prima occorrenza di 4 è all'indice 2.

Ricerca di $m = 9$ con `FirstOccurrence(a, 1, 10, 9)`:

chiamata	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	azione	restituisce
1	1	10	5	4	$a[mid] < m \Rightarrow$ chiama 6, 10	7
2	6	10	8	9	$a[mid] = m \Rightarrow r :=$ chiama 6, 7	$r = 7 \Rightarrow 7$
3	6	7	6	7	$a[mid] < m \Rightarrow$ chiama 7, 7	7
4	7	7	7	9	$a[mid] = m \Rightarrow r :=$ chiama 7, 6	$r = -1 \Rightarrow mid = 7$
5	7	6	-	-	$left > right$	-1

La prima occorrenza di 9 è all'indice 7.

Ricerca di $m = 5$ con `FirstOccurrence(a, 1, 10, 5)`:

chiamata	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	azione	restituisce
1	1	10	5	4	$a[mid] < m \Rightarrow$ chiama 6, 10	-1
2	6	10	8	9	$a[mid] > m \Rightarrow$ chiama 6, 7	-1
3	6	7	6	7	$a[mid] > m \Rightarrow$ chiama 6, 5	-1
4	6	5	-	-	$left > right$	-1

L'elemento 5 non è nell'array (restituisce -1).

3. Dimostriamo che, per $1 \leq left \leq right + 1 \leq n + 1$, `FirstOccurrence`($a, left, right, m$) restituisce il più piccolo indice $i \in \{left, \dots, right\}$ con $a[i] = m$, oppure -1 se non ce ne sono. Induzione forte su $k = right - left + 1$.

Caso base $k = 0$: l'intervallo è vuoto, m non compare, e l'algoritmo restituisce -1 .

Passo induttivo $k \geq 1$: supponiamo l'enunciato vero per tutte le dimensioni $k' < k$. Si ha $left \leq mid \leq right$, e ogni chiamata ricorsiva è su un intervallo di dimensione $right - mid$ oppure $mid - left$, entrambe minori di k .

- Se $a[mid] < m$: tutti gli elementi $a[left..mid]$ sono $< m$ (array ordinato), quindi le occorrenze di m in $a[left..right]$ sono esattamente quelle in $a[mid + 1..right]$, e la più piccola viene restituita dalla chiamata ricorsiva (ipotesi induttiva).
 - Se $a[mid] > m$: tutti gli elementi $a[mid..right]$ sono $> m$, quindi le occorrenze sono esattamente quelle in $a[left..mid - 1]$, e concludiamo come sopra.
 - Se $a[mid] = m$: mid è un'occorrenza, quindi la prima occorrenza è in $a[left..mid - 1]$ oppure è mid . Per l'ipotesi induttiva, r è la prima occorrenza in $a[left..mid - 1]$, oppure -1 se non ce ne sono. Nel primo caso l'algoritmo restituisce r (riga 16), nel secondo caso restituisce mid (riga 14). Entrambi sono corretti.
4. Sia $n = right - left + 1$. Ogni chiamata esegue un numero costante di passi (al più 7: righe 2, 5, 6, 9, 12, 13 e 14 oppure 16) e effettua **al più una** chiamata ricorsiva, su un intervallo di $\lfloor (n - 1)/2 \rfloor$ oppure $\lfloor n/2 \rfloor$ elementi. Quindi

$$S(n) = S(\lfloor n/2 \rfloor) + c \quad (n \geq 1), \quad S(0) = c_0.$$

Per il caso 2 del Master theorem ($a = 1$, $b = 2$, $f(n) = c = \Theta(n^{\log_2 1} \log^0 n)$, $k = 0$):

$$S(n) = \Theta(\log n).$$

Come per `RecursiveBinarySearch`, ci sono al più $\lfloor \log_2 n \rfloor + 1$ chiamate su un intervallo non vuoto (qui $\lfloor \log_2 10 \rfloor + 1 = 4$; la ricerca di $m = 9$ lo raggiunge). Si noti che `FirstOccurrence` non si ferma mai quando trova m : la ricorsione termina sempre con una chiamata su un intervallo vuoto.

5. Consideriamo l'array $a[1..n]$ con tutti gli elementi uguali a m . `RecursiveBinarySearch` restituisce immediatamente $i = mid = 1 + \lfloor (n - 1)/2 \rfloor$. Poi l'algoritmo si sposta a sinistra $i - 1 = \lfloor (n - 1)/2 \rfloor$ volte, fino a $i = 1$. Ogni spostamento costa un numero costante di passi, quindi il numero di passi nel caso peggiore è

$$\Theta(\log n) + \Theta(n) = \Theta(n).$$

L'algoritmo proposto è lineare nel caso peggiore, mentre `FirstOccurrence` è logaritmico.

6. No. Le chiamate ricorsive delle righe 7 e 10 sono chiamate in coda, ma la chiamata della riga 12 non lo è: dopo che termina, l'algoritmo deve ancora verificare $r = -1$ (riga 13). L'algoritmo può essere reso ricorsivo in coda passando il miglior candidato trovato finora come parametro aggiuntivo *best* (inizialmente -1): quando $a[mid] = m$, chiama `FirstOccurrence(a, left, mid-1, m, mid)`, e l'intervallo vuoto restituisce *best* invece di -1 .

Esercizio 17.14 — Algoritmi ricorsivi (E14)

Consideriamo l'array di $n = 5$ numeri interi

$$a = [2, -4, 3, -1, 2]$$

e l'algoritmo `BruteForceMaximumSubarray(a, n)`, che esamina tutte le coppie (l, r) con $1 \leq l \leq r \leq n$ nell'ordine $(1, 1), (1, 2), \dots, (1, n), (2, 2), \dots, (n, n)$, partendo da $best = a[1]$ e aggiornando *best*, *bl*, *br* solo quando $sum > best$. Il suo numero di passi su un array di n elementi, in funzione del numero U di aggiornamenti di *best*, è richiamato nell'introduzione agli esercizi.

Domande

1. Quanti sotto-array ha a ?
2. Applicare l'algoritmo di forza bruta ad a . Per ogni coppia (l, r) , riportare la somma di $a[l..r]$ e se *best* viene aggiornato. Quale terna $(bl, br, best)$ viene restituita?
3. Calcolare il numero esatto di passi eseguiti dall'algoritmo su a .
4. Per $n = 1000$, dare un limite inferiore e un limite superiore al numero di passi eseguiti dall'algoritmo su un qualunque input di dimensione n .

Soluzione

1. I sotto-array sono le coppie (l, r) con $1 \leq l \leq r \leq 5$:

$$\frac{n(n+1)}{2} = \frac{5 \cdot 6}{2} = 15.$$

2. Inizialmente $best = a[1] = 2$, $bl = br = 1$.

l	r	sum di $a[l..r]$	azione
1	1	2	$sum \leq best = 2$
1	2	-2	
1	3	1	
1	4	0	
1	5	2	$sum \leq best = 2$
2	2	-4	
2	3	-1	
2	4	-2	
2	5	0	
3	3	3	$sum > best \Rightarrow best = 3, bl = 3, br = 3$
3	4	2	
3	5	4	$sum > best \Rightarrow best = 4, bl = 3, br = 5$
4	4	-1	
4	5	1	
5	5	2	

L'algoritmo restituisce (3, 5, 4): il sotto-array di somma massima è $a[3..5] = [3, -1, 2]$ con somma 4. Si noti che l'intero array ha somma 2 e il sotto-array $a[1..1]$ ha somma 2: poiché il confronto è stretto, non causano mai un aggiornamento.

3. Ci sono $U = 2$ aggiornamenti (coppie (3, 3) e (3, 5)). Con $n = 5$:

$$2 \cdot 5^2 + 7 \cdot 5 + 6 + 3 \cdot 2 = 50 + 35 + 6 + 6 = 97 \text{ passi.}$$

4. Il numero di aggiornamenti soddisfa $0 \leq U \leq n(n+1)/2$. Quindi il numero di passi è almeno

$$2 \cdot 1000^2 + 7 \cdot 1000 + 6 = 2\,007\,006$$

e al più

$$2\,007\,006 + 3 \cdot \frac{1000 \cdot 1001}{2} = 2\,007\,006 + 1\,501\,500 = 3\,508\,506.$$

In entrambi i casi il numero di passi è circa 2–3.5 milioni: l'algoritmo è $\Theta(n^2)$ su ogni input, non solo nel caso peggiore.

Esercizio 17.15 — Algoritmi ricorsivi (E15)

Consideriamo l'array di $n = 8$ numeri interi

$$a = [-1, 3, -2, 4, -6, 1, 5, -3]$$

e l'algoritmo $\text{FindMaxCrossingSubarray}(a, l, mid, r)$, che scorre $i = mid, mid - 1, \dots, l$ calcolando le somme di $a[i..mid]$, e $j = mid + 1, \dots, r$ calcolando le somme di $a[mid + 1..j]$, tenendo la migliore di ciascuna (confronto stretto, valori iniziali $-\infty$).

Domande

1. Calcolare $\text{FindMaxCrossingSubarray}(a, 1, 4, 8)$ passo per passo.
2. Calcolare $\text{FindMaxCrossingSubarray}(a, 5, 6, 8)$ passo per passo.
3. Il sotto-array trovato nella domanda 1 è un sotto-array di somma massima di a ? Giustificare la risposta.
4. Quante iterazioni eseguono i cicli nelle domande 1 e 2? Perché il sotto-array restituito nella domanda 1 è obbligato a contenere $a[4]$ e $a[5]$, anche se $a[5] = -6$?

Soluzione

1. $l = 1, mid = 4, r = 8$.

Ciclo sinistro (i da 4 a 1 in discesa):

i	$a[i]$	sum	$leftsum$	$maxleft$
4	4	4	4	4
3	-2	2	4	4
2	3	5	5	2
1	-1	4	5	2

Ciclo destro (j da 5 a 8):

j	$a[j]$	sum	$rightsum$	$maxright$
5	-6	-6	-6	5
6	1	-5	-5	6
7	5	0	0	7
8	-3	-3	0	7

L'algoritmo restituisce $(maxleft, maxright, leftsum + rightsum) = (2, 7, 5 + 0) = (2, 7, 5)$: il sotto-array $a[2..7] = [3, -2, 4, -6, 1, 5]$ con somma 5.

2. $l = 5, mid = 6, r = 8$.

i	$a[i]$	sum	$leftsum$	$maxleft$
6	1	1	1	6
5	-6	-5	1	6
j	$a[j]$	sum	$rightsum$	$maxright$
7	5	5	5	7
8	-3	2	5	7

L'algoritmo restituisce $(6, 7, 1 + 5) = (6, 7, 6)$: il sotto-array $a[6..7] = [1, 5]$ con somma 6.

3. No. Il sotto-array $a[6..7]$ trovato nella domanda 2 ha somma $6 > 5$. In effetti, $a[6..7]$ è un sotto-array di somma massima di a : il sotto-array di somma massima si trova interamente nella metà destra $a[5..8]$ e **non** attraversa il punto medio $mid = 4$.

FindMaxCrossingSubarray trova solo il miglior sotto-array **tra quelli che attraversano il punto medio**. Ecco perché **FindMaximumSubarray** risolve ricorsivamente anche le due metà e restituisce il migliore dei tre candidati: qui la metà sinistra $a[1..4]$ dà $a[2..4]$ con somma $3 - 2 + 4 = 5$, la metà destra $a[5..8]$ dà $a[6..7]$ con somma 6, il sotto-array che attraversa il punto medio ha somma 5, e la risposta è $a[6..7]$ con somma 6.

4. Nella domanda 1 i cicli eseguono $(mid - l + 1) + (r - mid) = 4 + 4 = 8 = r - l + 1$ iterazioni; nella domanda 2 eseguono $2 + 2 = 4 = r - l + 1$ iterazioni. In generale, **FindMaxCrossingSubarray** esegue esattamente $n = r - l + 1$ iterazioni, quindi $\Theta(n)$ passi.

Per definizione, un sotto-array che attraversa il punto medio è $a[i..j]$ con $i \leq mid < j$: contiene sempre sia $a[mid]$ sia $a[mid + 1]$. Infatti, il ciclo sinistro parte da $i = mid$ e il ciclo destro da $j = mid + 1$, e sia *leftsum* sia *rightsum* vengono aggiornati alla prima iterazione (valori iniziali $-\infty$). I sotto-array che non contengono $a[5] = -6$ sono considerati dalle chiamate ricorsive sulle due metà.

Esercizio 17.16 — Algoritmi ricorsivi (E16)

Consideriamo l'array di $n = 8$ numeri interi

$$a = [-2, 4, -1, 3, -5, 2, 6, -4]$$

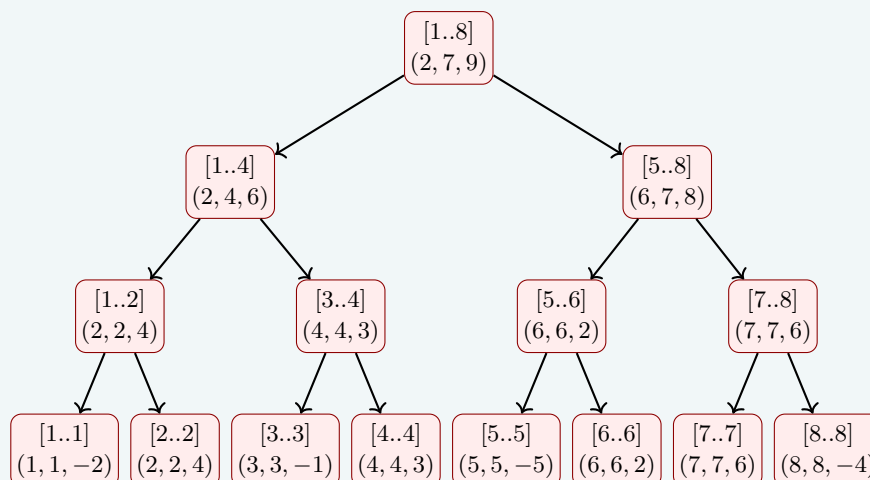
e l'algoritmo divide et impera **FindMaximumSubarray**(a, l, r), con prima chiamata **FindMaximumSubarray**($a, 1, 8$), $mid = \lfloor (l+r)/2 \rfloor$, e parità risolte a favore della metà sinistra, poi della metà destra.

Domande

1. Disegnare l'albero delle chiamate, indicando per ogni chiamata $[l..r]$ la terna (i, j, s) che restituisce.
2. Per ogni chiamata su almeno due elementi, riportare le terne restituite dalle due chiamate ricorsive e da **FindMaxCrossingSubarray**, e la terna restituita.
3. Mostrare il calcolo di **FindMaxCrossingSubarray** nella prima chiamata.
4. Quante chiamate di **FindMaximumSubarray** vengono eseguite? Quante iterazioni vengono eseguite, in totale, dai cicli di tutte le chiamate di **FindMaxCrossingSubarray**? Confrontare con $n \log_2 n$.

Soluzione

1. Albero delle chiamate (ogni nodo: il sotto-array $[l..r]$ e la terna restituita (i, j, s)):



2. Risultati delle chiamate su almeno due elementi (candidati sinistro, destro e che attraversa il punto medio, e terna restituita):

chiamata	<i>mid</i>	sinistra	destra	attraversante	restituita
[1..8]	4	(2, 4, 6)	(6, 7, 8)	(2, 7, 9)	(2, 7, 9)
[1..4]	2	(2, 2, 4)	(4, 4, 3)	(2, 4, 6)	(2, 4, 6)
[5..8]	6	(6, 6, 2)	(7, 7, 6)	(6, 7, 8)	(6, 7, 8)
[1..2]	1	(1, 1, -2)	(2, 2, 4)	(1, 2, 2)	(2, 2, 4)
[3..4]	3	(3, 3, -1)	(4, 4, 3)	(3, 4, 2)	(4, 4, 3)
[5..6]	5	(5, 5, -5)	(6, 6, 2)	(5, 6, -3)	(6, 6, 2)
[7..8]	7	(7, 7, 6)	(8, 8, -4)	(7, 8, 2)	(7, 7, 6)

Per esempio, nella chiamata $[1..4]$ il sotto-array che attraversa il punto medio si calcola così: ciclo sinistro $i = 2$: $sum = 4$; $i = 1$: $sum = 2$, quindi $leftsum = 4$, $maxleft = 2$; ciclo destro $j = 3$: $sum = -1$; $j = 4$: $sum = 2$, quindi $rightsum = 2$, $maxright = 4$; risultato $(2, 4, 6)$. Nella chiamata $[5..8]$: $leftsum = 2$ ($maxleft = 6$), $rightsum = 6$ ($maxright = 7$), risultato $(6, 7, 8)$.

3. Prima chiamata: $l = 1$, $mid = 4$, $r = 8$.

<i>i</i>	$a[i]$	<i>sum</i>	<i>leftsum</i>	<i>maxleft</i>	<i>j</i>	$a[j]$	<i>sum</i>	<i>rightsum</i>	<i>maxright</i>
4	3	3	3	4	5	-5	-5	-5	5
3	-1	2	3	4	6	2	-3	-3	6
2	4	6	6	2	7	6	3	3	7
1	-2	4	6	2	8	-4	-1	3	7

Il sotto-array che attraversa il punto medio è $a[2..7] = [4, -1, 3, -5, 2, 6]$ con somma $6 + 3 = 9$. Poiché $9 > 6$ (sinistra) e $9 > 8$ (destra), l'algoritmo restituisce $(2, 7, 9)$: il sotto-array di somma massima è $a[2..7]$ con somma 9. Si noti che contiene l'elemento negativo $a[5] = -5$.

4. L'albero delle chiamate ha $n = 8$ foglie e $n - 1 = 7$ nodi interni: $2n - 1 = 15$ chiamate di `FindMaximumSubarray`, e 7 chiamate di `FindMaxCrossingSubarray`.

Una chiamata di `FindMaxCrossingSubarray` su $a[l..r]$ esegue $r - l + 1$ iterazioni. Le chiamate di ciascun livello dell'albero ricoprono l'intero array una volta:

- livello 0: $[1..8] \Rightarrow 8$ iterazioni;
- livello 1: $[1..4], [5..8] \Rightarrow 4 + 4 = 8$ iterazioni;
- livello 2: $[1..2], [3..4], [5..6], [7..8] \Rightarrow 2 + 2 + 2 + 2 = 8$ iterazioni.

In totale $3 \cdot 8 = 24$ iterazioni, che è esattamente $n \log_2 n = 8 \cdot 3 = 24$. Questo è il termine $cn \log_2 n$ dell'albero di ricorsione, che dà $S(n) = \Theta(n \log n)$.

Esercizio 17.17 — Algoritmi ricorsivi (E17)

Consideriamo i due array

$$a = [-2, 1, -3, 4, -1, 2, 1, -5, 4] \quad \text{e} \quad b = [-8, -3, -6, -2, -5, -4]$$

e l'algoritmo iterativo `Kadane(a, n)`: parte con $best := a[1]$, $bl := br := 1$, $cur := a[1]$, $cl := 1$ e, per $i = 2, \dots, n$, esegue

- se $cur < 0$: $cur \leftarrow a[i]$, $cl \leftarrow i$ (ripartenza); altrimenti $cur \leftarrow cur + a[i]$ (estensione);
- se $cur > best$: $best \leftarrow cur$, $bl \leftarrow cl$, $br \leftarrow i$.

Restituisce $(bl, br, best)$.

Domande

1. Applicare `Kadane` all'array a , riportando per ogni i l'azione eseguita e i valori di cur , cl , $best$, bl , br .
2. Applicare `Kadane` all'array b , i cui elementi sono tutti negativi.
3. Che cosa restituiscono su b l'algoritmo di forza bruta e l'algoritmo divide et impera `FindMaximumSubarray`? (Per l'algoritmo divide et impera, riportare i tre candidati della prima chiamata.) Dimostrare che, se tutti gli elementi di un array sono negativi, ogni sotto-array di somma massima è formato da un solo elemento.
4. Una variante diffusa dell'algoritmo inizializza $cur := 0$ e $best := 0$ e, per $i = 1, \dots, n$, esegue $cur \leftarrow \max\{0, cur + a[i]\}$ e $best \leftarrow \max\{best, cur\}$. Che cosa restituisce su a e su b ? Quale problema risolve?

Soluzione

1. Esecuzione su $a = [-2, 1, -3, 4, -1, 2, 1, -5, 4]$ (riga $i = 1$: inizializzazione; le altre righe riportano i valori alla fine dell'iterazione con indice i):

i	$a[i]$	azione	cur	cl	$best$	(bl, br)
1	-2	inizializzazione	-2	1	-2	(1, 1)
2	1	ripartenza ($cur = -2 < 0$)	1	2	1	(2, 2)
3	-3	estensione	-2	2	1	(2, 2)
4	4	ripartenza ($cur = -2 < 0$)	4	4	4	(4, 4)
5	-1	estensione	3	4	4	(4, 4)
6	2	estensione	5	4	5	(4, 6)
7	1	estensione	6	4	6	(4, 7)
8	-5	estensione	1	4	6	(4, 7)
9	4	estensione	5	4	6	(4, 7)

L'algoritmo restituisce (4, 7, 6): il sotto-array di somma massima è $a[4..7] = [4, -1, 2, 1]$ con somma 6.

2. Esecuzione su $b = [-8, -3, -6, -2, -5, -4]$:

i	$b[i]$	azione	cur	cl	$best$	(bl, br)
1	-8	inizializzazione	-8	1	-8	(1, 1)
2	-3	ripartenza	-3	2	-3	(2, 2)
3	-6	ripartenza	-6	3	-3	(2, 2)
4	-2	ripartenza	-2	4	-2	(4, 4)
5	-5	ripartenza	-5	5	-2	(4, 4)
6	-4	ripartenza	-4	6	-2	(4, 4)

Poiché cur è sempre negativo, l'algoritmo riparte a ogni iterazione: $cur = b[i]$, e $best$ è semplicemente il massimo elemento visto finora. L'algoritmo restituisce (4, 4, -2): il sotto-array di somma massima è il singolo elemento $b[4] = -2$.

3. L'algoritmo di forza bruta esamina tutti i sotto-array e restituisce anch'esso (4, 4, -2).

L'algoritmo divide et impera, nella prima chiamata $[1..6]$ con $mid = 3$, ottiene:

- metà sinistra $[1..3]$: (2, 2, -3);
- metà destra $[4..6]$: (4, 4, -2);
- sotto-array che attraversa il punto medio: ciclo sinistro $i = 3$: $sum = -6$, $i = 2$: -9 , $i = 1$: -17 , quindi $leftsum = -6$ ($maxleft = 3$); ciclo destro $j = 4$: $sum = -2$, $j = 5$: -7 , $j = 6$: -11 , quindi $rightsum = -2$ ($maxright = 4$); risultato (3, 4, -8).

Il migliore dei tre è $(4, 4, -2)$. Tutti e tre gli algoritmi restituiscono $b[4..4]$ con somma -2 .

Dimostrazione della proprietà. Siano tutti gli elementi negativi, e sia $a[l..r]$ un sotto-array con almeno due elementi ($l < r$). Allora

$$\sum_{k=l}^r a[k] = a[l] + \sum_{k=l+1}^r a[k] < a[l],$$

perché $\sum_{k=l+1}^r a[k]$ è una somma di numeri negativi. Quindi il sotto-array di un solo elemento $a[l..l]$ ha somma maggiore, e $a[l..r]$ non è un sotto-array di somma massima. Quindi ogni sotto-array di somma massima ha un elemento, ed è un elemento massimo dell'array.

4. La variante calcola la massima somma di un sotto-array **ammettendo il sotto-array vuoto** (la cui somma è 0): riportare *cur* a 0 significa «iniziare un nuovo sotto-array, vuoto».

- Su *a*: *cur* assume i valori 0, 1, 0, 4, 3, 5, 6, 1, 5, e l'algoritmo restituisce $best = 6$, come prima, poiché il sotto-array di somma massima ha somma positiva.
- Su *b*: *cur* viene riportato a 0 a ogni iterazione, e l'algoritmo restituisce $best = 0$, che è la somma del sotto-array vuoto. Questa **non** è una risposta corretta al nostro problema, in cui un sotto-array deve essere non vuoto ($l \leq r$): la risposta corretta è -2 .

Nell'interpretazione sull'azione, la variante risponde alla domanda «qual è il massimo profitto se possiamo anche **non comprare** affatto?»: se il prezzo scende ogni giorno, la scelta migliore è non comprare (profitto 0). Le due versioni coincidono ogni volta che l'array contiene almeno un elemento non negativo.

Algoritmi ricorsivi e iterativi a confronto

Esercizio 18.1 — Algoritmi ricorsivi contro iterativi (E1)

Consideriamo i seguenti due algoritmi ricorsivi.

Algorithm: GCDRec

Input : Due numeri naturali a e b , non entrambi uguali a 0

Output: Il massimo comune divisore di a e b

```
1 function GCDRec( $a, b$ )
2   if  $b = 0$  then
3     return  $a$ 
4   end
5   return GCDRec( $b, a \bmod b$ )
6 end
```

Algorithm: RecBinarySearch**Input** : Un sotto-array ordinato $a[l..r]$ e un valore da cercare m **Output**: Un indice di m in $a[l..r]$, oppure -1 se m non è presente

```

1 function RecBS( $a, l, r, m$ )
2   if  $l > r$  then
3     return  $-1$ 
4   end
5    $mid := l + \lfloor (r - l) / 2 \rfloor$ 
6   if  $a[mid] = m$  then
7     return  $mid$ 
8   end
9   if  $a[mid] < m$  then
10    return RecBS( $a, mid + 1, r, m$ )
11  else
12    return RecBS( $a, l, mid - 1, m$ )
13  end
14 end

```

Domande

1. GCDRec e RecBS sono ricorsivi in coda? Giustificare la risposta.
2. Trasformare GCDRec in un algoritmo iterativo equivalente GCDIt, applicando passo per passo l'eliminazione meccanica della ricorsione in coda. Poi eseguire entrambi gli algoritmi con $(a, b) = (1071, 462)$, elencando i valori dei parametri in ogni chiamata e in ogni iterazione.
3. Trasformare RecBS in un algoritmo iterativo equivalente ItBS. Poi eseguire entrambi gli algoritmi sull'array $a = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]$ (con $l = 1, r = 10$) per $m = 23$ e per $m = 4$.
4. Confrontare le versioni ricorsiva e iterativa dei due algoritmi in termini di tempo e spazio. In particolare, determinare la profondità della ricorsione di GCDRec(89, 55) e la massima profondità della ricorsione di RecBS su un array di $n = 10^6$ elementi.

Soluzione

1. Entrambi gli algoritmi sono ricorsivi in coda.
 - In GCDRec l'unica chiamata ricorsiva è alla riga 5: il suo risultato viene restituito così com'è, senza alcuna ulteriore operazione.
 - In RecBS le chiamate ricorsive sono alle righe 10 e 12: in entrambi i casi il risultato della chiamata viene restituito così com'è.

Quindi, dopo ogni chiamata ricorsiva, nella funzione chiamante non resta nulla da fare:

il suo frame non serve più, e la chiamata può essere sostituita da un aggiornamento dei parametri.

2. Applichiamo le tre regole dell'eliminazione meccanica della ricorsione in coda.

- La condizione del caso base è $b = 0$ (riga 2): la condizione del ciclo è $b \neq 0$.
- La chiamata in coda $\text{GCDRec}(b, a \bmod b)$ diventa l'assegnazione simultanea $(a, b) \leftarrow (b, a \bmod b)$. Poiché il nuovo valore di b deve essere calcolato con il vecchio valore di a , usiamo una variabile temporanea r .
- Il return del caso base (**return** a , riga 3) viene spostato dopo il ciclo.

Algorithm: GCDIt

```

1 function GCDIt( $a, b$ )
2   while  $b \neq 0$  do
3      $r := a \bmod b$ 
4      $a \leftarrow b$ 
5      $b \leftarrow r$ 
6   end
7   return  $a$ 
8 end

```

Esecuzione con $(a, b) = (1071, 462)$:

GCDRec				GCDIt			
chiamata	a	b	$a \bmod b$	iterazione	a	b	r
1	1071	462	147	1	1071	462	147
2	462	147	21	2	462	147	21
3	147	21	0	3	147	21	0
4	21	0	–	fine	21	0	–

Entrambi gli algoritmi passano per le stesse coppie $(1071, 462)$, $(462, 147)$, $(147, 21)$, $(21, 0)$ e restituiscono 21. **GCDRec** esegue 4 chiamate (tutte attive contemporaneamente quando si raggiunge il caso base); **GCDIt** esegue 3 iterazioni.

3. La condizione del caso base $l > r$ diventa la condizione del ciclo $l \leq r$; il secondo caso base ($a[\text{mid}] = m$) resta dentro il ciclo come un return; la chiamata in coda alla riga 10 diventa $l \leftarrow \text{mid} + 1$ e la chiamata in coda alla riga 12 diventa $r \leftarrow \text{mid} - 1$ (i parametri a e m non cambiano); il return di -1 viene spostato dopo il ciclo.

Algorithm: ItBinarySearch

```

1 function ItBS( $a, l, r, m$ )
2   while  $l \leq r$  do
3      $mid := l + \lfloor (r - l) / 2 \rfloor$ 
4     if  $a[mid] = m$  then
5       return  $mid$ 
6     end
7     if  $a[mid] < m$  then
8        $l \leftarrow mid + 1$ 
9     else
10       $r \leftarrow mid - 1$ 
11    end
12  end
13  return  $-1$ 
14 end

```

I due algoritmi passano per gli stessi valori di (l, r) ; per **RecBS** ogni riga è una chiamata, per **ItBS** ogni riga è un'iterazione (l'ultima riga con $l > r$ è il test finale del ciclo).

- $m = 23$:

chiamata / iterazione	l	r	mid	$a[mid]$
1	1	10	5	$16 < 23 \Rightarrow l = 6$
2	6	10	8	$56 > 23 \Rightarrow r = 7$
3	6	7	6	$23 = m \Rightarrow$ restituisce 6

- $m = 4$:

chiamata / iterazione	l	r	mid	$a[mid]$
1	1	10	5	$16 > 4 \Rightarrow r = 4$
2	1	4	2	$5 > 4 \Rightarrow r = 1$
3	1	1	1	$2 < 4 \Rightarrow l = 2$
4	2	1	–	$l > r \Rightarrow$ restituisce -1

Per $m = 23$ **RecBS** esegue 3 chiamate e **ItBS** 3 iterazioni; per $m = 4$ **RecBS** esegue 4 chiamate e **ItBS** 3 iterazioni più il test finale.

4. **Tempo.** Le versioni iterative passano per esattamente gli stessi valori dei parametri di quelle ricorsive, con un numero costante di passi per ogni chiamata / iterazione. Quindi il numero di passi ha lo stesso ordine di crescita: $\Theta(\log \min(a, b))$ nel caso peggiore per l'MCD e $\Theta(\log_2 n)$ nel caso peggiore per la ricerca binaria. Le versioni ricorsive pagano soltanto il sovraccarico (costante) delle chiamate.

Spazio. Le versioni iterative usano $\Theta(1)$ di memoria aggiuntiva (i parametri e le variabili r o mid). Nelle versioni ricorsive tutte le chiamate restano attive finché non

si raggiunge il caso base, quindi la pila contiene tanti frame quanti sono la profondità della ricorsione.

- $\text{GCDRec}(89, 55)$ passa per le coppie

$(89, 55), (55, 34), (34, 21), (21, 13), (13, 8), (8, 5), (5, 3), (3, 2), (2, 1), (1, 0),$

quindi la profondità della ricorsione è 10. Numeri di Fibonacci consecutivi sono il caso peggiore dell'algoritmo di Euclide: la profondità è $\Theta(\log \min(a, b))$.

- Per RecBS su n elementi la profondità massima è $\lfloor \log_2 n \rfloor + 2$ (le dimensioni dei sotto-array sono al più $n, \lfloor n/2 \rfloor, \dots, 1$, più l'ultima chiamata sul sotto-array vuoto). Per $n = 10^6$: $\lfloor \log_2 10^6 \rfloor + 2 = 19 + 2 = 21$.

Per entrambi gli algoritmi, la ricorsione non dà **alcun guadagno** in tempo e una (piccola, logaritmica) **perdita** in spazio. Poiché sono ricorsivi in coda, la ricorsione può essere eliminata meccanicamente, ottenendo cicli con $\Theta(1)$ di memoria aggiuntiva.

Esercizio 18.2 — Algoritmi ricorsivi contro iterativi (E2)

Consideriamo i seguenti algoritmi ricorsivi e dimensioni dell'input.

- $\text{Fact}(n)$: restituisce 1 se $n = 0$, altrimenti $n \cdot \text{Fact}(n - 1)$; con $n = 1000$.
- $\text{SumDecrease}(a, n)$: restituisce 0 se $n = 0$, altrimenti $\text{SumDecrease}(a, n - 1) + a[n]$; con $n = 10^6$.
- $\text{SumHalving}(a, l, r)$: restituisce $a[l]$ se $l = r$, altrimenti $\text{SumHalving}(a, l, \text{mid}) + \text{SumHalving}(a, \text{mid} + 1, r)$ con $\text{mid} = \lfloor (l + r)/2 \rfloor$; su un array di $n = 10^6$ elementi.
- Merge Sort su un array di $n = 1000$ elementi (caso base: sotto-array di un elemento).
- Torre di Hanoi con $n = 20$ dischi (caso base: 0 dischi).
- $\text{FibRec}(n)$: restituisce n se $n \leq 1$, altrimenti $\text{FibRec}(n - 1) + \text{FibRec}(n - 2)$; con $n = 30$.
- Potenza veloce ricorsiva x^n (caso base: $n = 0$; chiamata ricorsiva su $\lfloor n/2 \rfloor$); con $n = 10^9$.

Supponiamo che ogni frame della pila occupi 48 byte e che la pila delle chiamate abbia una dimensione di $8 \text{ MB} = 8 \cdot 2^{20} = 8\,388\,608$ byte.

Domande

1. Determinare la profondità della ricorsione di ciascun algoritmo per la dimensione dell'input data.

2. Determinare la memoria occupata dalla pila delle chiamate al suo massimo, e dire quali esecuzioni causano uno stack overflow.
3. Determinare il massimo n per cui **SumDecrease**(a, n) non causa uno stack overflow, e la profondità di **SumHalving** per lo stesso n .
4. Per **FibRec**(30), confrontare la profondità della ricorsione con il numero totale di chiamate. Il problema di questo algoritmo è lo spazio o il tempo?

Soluzione

1. La profondità della ricorsione è il massimo numero di chiamate attive contemporaneamente, cioè il numero di chiamate sul cammino più lungo dalla radice a una foglia dell'albero di ricorsione.
 - (a) Le chiamate attive sono **Fact**(1000), **Fact**(999), ..., **Fact**(0): profondità $n + 1 = 1001$.
 - (b) Le chiamate attive sono **SumDecrease**($a, 10^6$), ..., **SumDecrease**($a, 0$): profondità $n + 1 = 1\,000\,001$.
 - (c) Il sotto-array più grande al livello t dell'albero ha $\lceil n/2^t \rceil$ elementi, che diventa 1 per $t = \lceil \log_2 n \rceil$. Profondità $\lceil \log_2 n \rceil + 1$; poiché $2^{19} = 524\,288 < 10^6 \leq 2^{20} = 1\,048\,576$, si ha $\lceil \log_2 10^6 \rceil = 20$ e la profondità è 21.
 - (d) Come in (c): $\lceil \log_2 1000 \rceil + 1 = 10 + 1 = 11$ (poiché $2^9 = 512 < 1000 \leq 2^{10} = 1024$).
 - (e) Le chiamate attive sono **Hanoi**(20), **Hanoi**(19), ..., **Hanoi**(0): profondità $n + 1 = 21$.
 - (f) Il cammino più lungo è **FibRec**(30), **FibRec**(29), ..., **FibRec**(1): profondità $n = 30$.
 - (g) L'esponente assume i valori $n, \lfloor n/2 \rfloor, \dots, 1, 0$, cioè la profondità è $\lfloor \log_2 n \rfloor + 2$. Poiché $2^{29} = 536\,870\,912 \leq 10^9 < 2^{30}$, si ha $\lfloor \log_2 10^9 \rfloor = 29$ e la profondità è 31.
2. La memoria della pila al suo massimo è 48 byte per la profondità:

	algoritmo	profondità	pila (byte)	overflow?
(a)	Fact (1000)	1 001	48 048	no
(b)	SumDecrease , $n = 10^6$	1 000 001	48 000 048	sì
(c)	SumHalving , $n = 10^6$	21	1 008	no
(d)	Merge Sort, $n = 1000$	11	528	no
(e)	Hanoi, $n = 20$	21	1 008	no
(f)	FibRec (30)	30	1 440	no
(g)	potenza veloce, $n = 10^9$	31	1 488	no

Solo (b) causa uno stack overflow: $48\,000\,048 > 8\,388\,608$ byte (circa 46 MB contro 8 MB).

3. La pila contiene al più $\lfloor 8\,388\,608/48 \rfloor = 174\,762$ frame. `SumDecrease`(a, n) richiede $n + 1$ frame, quindi non va in overflow se e solo se

$$n + 1 \leq 174\,762 \iff n \leq 174\,761.$$

Per $n = 174\,761$, poiché $2^{17} = 131\,072 < n \leq 2^{18} = 262\,144$, `SumHalving` ha profondità $\lceil \log_2 n \rceil + 1 = 18 + 1 = 19$, cioè 912 byte di pila. Entrambi gli algoritmi eseguono $\Theta(n)$ passi: la versione per dimezzamento usa una pila esponenzialmente più piccola.

4. `FibRec`(30) ha profondità 30 (solo 1 440 byte di pila), ma esegue

$$2F(31) - 1 = 2 \cdot 1\,346\,269 - 1 = 2\,692\,537$$

chiamate. Il problema di `FibRec` è il **tempo**, $\Theta(\varphi^n)$, non lo spazio, $\Theta(n)$: la profondità della ricorsione e il numero di chiamate sono due quantità molto diverse.

Una profondità della ricorsione $\Theta(\log_2 n)$ non è mai un problema in pratica; una profondità $\Theta(n)$ (decremento di uno) causa uno stack overflow già per n dell'ordine di 10^5 – 10^6 , mentre il ciclo corrispondente usa $\Theta(1)$ di memoria.

Esercizio 18.3 — Algoritmi ricorsivi contro iterativi (E3)

Consideriamo i seguenti quattro algoritmi ricorsivi.

Algorithm: A: MaxDecrease

```

1 function MaxDec(a, n)
2   if n = 1 then
3     return a[1]
4   end
5   return max(MaxDec(a, n - 1), a[n])
6 end
```

Algorithm: B: MaxHalving

```

1 function MaxHalf(a, l, r)
2   if l = r then
3     return a[l]
4   end
5   mid := (l + r) / 2
6   return max(MaxHalf(a, l, mid),
              MaxHalf(a, mid + 1, r))
7 end
```

Algorithm: C: PowerDecrease

```

1 function PowDec( $x, n$ )
2   if  $n = 0$  then
3     return 1
4   end
5   return  $x \cdot \text{PowDec}(x, n - 1)$ 
6 end

```

Algorithm: D: Tribonacci

```

1 function Trib( $n$ )
2   if  $n \leq 2$  then
3     return  $\lfloor n/2 \rfloor$ 
4   end
5   return Trib( $n - 1$ ) + Trib( $n - 2$ )
        + Trib( $n - 3$ )
6 end

```

MaxDec e MaxHalf calcolano il massimo di un array $a[1..n]$ (MaxHalf si chiama come MaxHalf($a, 1, n$)); PowDec calcola x^n ; Trib calcola i numeri di Tribonacci $T(0) = 0$, $T(1) = 0$, $T(2) = 1$, $T(n) = T(n - 1) + T(n - 2) + T(n - 3)$.

Domande Per ciascuno dei quattro algoritmi:

1. scrivere l'equazione di ricorrenza del numero di passi $S(n)$ e determinarne l'ordine di crescita;
2. determinare la profondità della ricorsione e la memoria aggiuntiva usata;
3. indicare il miglior algoritmo iterativo visto nel corso per lo stesso problema, con il suo tempo e il suo spazio;
4. decidere se la ricorsione guadagna, è equivalente o perde, giustificando la risposta.

Soluzione**A: MaxDecrease**

1. Ogni chiamata esegue un numero costante c di passi oltre alla chiamata ricorsiva su $n - 1$ elementi:

$$S(n) = S(n - 1) + c, \quad S(1) = d.$$

Srotolando, $S(n) = d + c(n - 1) = \Theta(n)$.

2. Le chiamate MaxDec(a, n), ..., MaxDec($a, 1$) sono tutte attive contemporaneamente: profondità n , memoria aggiuntiva $\Theta(n)$. L'algoritmo non è ricorsivo in coda (il massimo viene calcolato dopo che la chiamata termina).

3. FindMaximum (capitolo «Tempo di esecuzione»): un ciclo con $S(n) = 4n - 1 = \Theta(n)$ passi e $\Theta(1)$ di memoria aggiuntiva.
4. **UGUALE** tempo, **PERDITA** in spazio: la ricorsione per decremento di uno è un ciclo sotto mentite spoglie, con un sovraccarico delle chiamate e un rischio di stack overflow per n grande.

B: MaxHalving

1. Due chiamate ricorsive sulle metà, più un numero costante di passi:

$$S(n) = 2S(n/2) + \Theta(1).$$

Master theorem: $a = 2$, $b = 2$, $f(n) = \Theta(1)$; $n^{\log_b a} = n^{\log_2 2} = n$. Poiché $f(n) = \Theta(1) = O(n^{1-\varepsilon})$ con $\varepsilon = 1$, il caso 1 dà $S(n) = \Theta(n)$.

2. Profondità $\lceil \log_2 n \rceil + 1$, memoria aggiuntiva $\Theta(\log_2 n)$.
3. Di nuovo FindMaximum: tempo $\Theta(n)$, spazio $\Theta(1)$.
4. **UGUALE** tempo, piccola **PERDITA** in spazio ($\Theta(\log_2 n)$). Il divide et impera qui non aiuta: l'albero di ricorsione ha $2n - 1$ nodi, ciascuno con lavoro costante, quindi il lavoro totale è ancora $\Theta(n)$. Nessun algoritmo può fare di meglio, perché ogni elemento deve essere esaminato almeno una volta. Il divide et impera dà un guadagno solo quando riduce il lavoro totale, come nel Merge Sort, in cui la combinazione delle metà costa $\Theta(n)$ invece dei $\Theta(n^2)$ degli algoritmi iterativi.

C: PowerDecrease

1. $S(n) = S(n - 1) + c$, $S(0) = d \Rightarrow S(n) = cn + d = \Theta(n)$.
2. Profondità $n + 1$ (chiamate con esponente $n, n - 1, \dots, 0$), memoria aggiuntiva $\Theta(n)$.
3. Il ciclo con n moltiplicazioni ha tempo $\Theta(n)$ e spazio $\Theta(1)$; l'algoritmo iterativo square-and-multiply **PowerIt** ha tempo $\Theta(\log_2 n)$ e spazio $\Theta(1)$.
4. **PERDITA**: stesso tempo del semplice ciclo e $\Theta(n)$ di memoria; rispetto al ciclo square-and-multiply è addirittura esponenzialmente più lento in termini di $\log_2 n$. La ricorsione conviene per la potenza solo quando **dimezza** l'esponente (potenza veloce): $S(n) = S(n/2) + \Theta(1) = \Theta(\log_2 n)$.

D: Tribonacci

1. Sia $C(n)$ il numero di chiamate; allora $C(0) = C(1) = C(2) = 1$ e

$$C(n) = 1 + C(n - 1) + C(n - 2) + C(n - 3), \quad n \geq 3,$$

e $S(n) = \Theta(C(n))$, poiché ogni chiamata esegue un numero costante di passi. La funzione C è non decrescente, quindi per $n \geq 3$

$$C(n) \geq 3C(n - 3) \geq 3^2C(n - 6) \geq \dots \geq 3^{\lfloor n/3 \rfloor}.$$

Quindi $S(n) = \Omega(3^{n/3}) = \Omega(1.442^n)$: l'algoritmo è esponenziale. (Per esempio $C(10) = 289$, $C(20) = 128\,287$, $C(25) = 2\,700\,421$.) Allo stesso modo, $C(n) \leq 3C(n-1) + 1$ dà $S(n) = O(3^n)$.

2. Il cammino più lungo nell'albero di ricorsione è $\text{Trib}(n), \text{Trib}(n-1), \dots, \text{Trib}(2)$: profondità $n-1$, memoria aggiuntiva $\Theta(n)$.
3. Un ciclo che calcola $T(3), T(4), \dots, T(n)$ tenendo solo gli ultimi tre valori in tre variabili: tempo $\Theta(n)$ e spazio $\Theta(1)$.
4. Enorme **PERDITA**: i sottoproblemi si **sovrappongono** (per esempio $\text{Trib}(n-1)$ e $\text{Trib}(n-2)$ chiamano entrambe $\text{Trib}(n-3)$), quindi gli stessi valori vengono ricalcolati un numero esponenziale di volte. La memoizzazione o il ciclo bottom-up (programmazione dinamica) riportano il tempo a $\Theta(n)$.

	ricorsivo		miglior iterativo	esito
	tempo	spazio	tempo, spazio	
A	$\Theta(n)$	$\Theta(n)$	$\Theta(n), \Theta(1)$	stesso tempo, perdita in spazio
B	$\Theta(n)$	$\Theta(\log_2 n)$	$\Theta(n), \Theta(1)$	stesso tempo, piccola perdita
C	$\Theta(n)$	$\Theta(n)$	$\Theta(\log_2 n), \Theta(1)$	perdita
D	esponenziale	$\Theta(n)$	$\Theta(n), \Theta(1)$	enorme perdita

Esercizio 18.4 — Algoritmi ricorsivi contro iterativi (E4)

I coefficienti binomiali soddisfano, per $0 < k < n$, l'equazione di ricorrenza (regola di Pascal)

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \quad \binom{n}{0} = \binom{n}{n} = 1.$$

Consideriamo il seguente algoritmo ricorsivo.

Algorithm: BinomRec

Input : Due numeri naturali n e k con $0 \leq k \leq n$

Output: Il coefficiente binomiale $\binom{n}{k}$

```

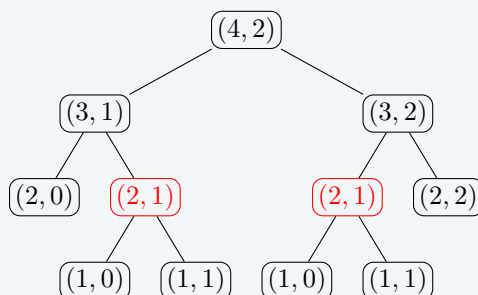
1 function BinomRec( $n, k$ )
2   if  $k = 0$  or  $k = n$  then
3     return 1
4   end
5   return BinomRec( $n-1, k-1$ ) + BinomRec( $n-1, k$ )
6 end
```

Domande

1. Disegnare l'albero di ricorsione di $\text{BinomRec}(4, 2)$, contare le chiamate e indicare i sottoproblemi che vengono risolti più di una volta.
2. Dimostrare che il numero di chiamate eseguite da $\text{BinomRec}(n, k)$ è $N(n, k) = 2\binom{n}{k} - 1$.
3. Calcolare $N(10, 5)$, $N(30, 15)$ e $N(40, 20)$, e stimare il tempo richiesto da $\text{BinomRec}(40, 20)$ su un calcolatore che esegue 10^9 chiamate al secondo.
4. Scrivere una versione con memoizzazione BinomMemo e dimostrare che, per $0 < k < n$, esegue esattamente $2k(n - k) + 1$ chiamate. Calcolare questo numero per i tre casi della domanda 3.
5. Scrivere un algoritmo iterativo che calcola $\binom{n}{k}$ riga per riga (triangolo di Pascal), e darne il tempo e lo spazio.

Soluzione

1. Ogni nodo è una chiamata, etichettata con (n, k) ; le foglie sono casi base ($k = 0$ oppure $k = n$) e restituiscono 1.



L'albero ha 11 nodi, cioè 11 chiamate, e $6 = \binom{4}{2}$ foglie. Il sottoproblema $(2, 1)$ viene risolto **due volte** (e così le sue sotto-chiamate $(1, 0)$ e $(1, 1)$).

2. Ogni chiamata è o una foglia (caso base, restituisce 1) o un nodo interno con esattamente due figli. Il valore restituito da una chiamata è la somma dei valori restituiti dalle foglie del suo sotto-albero, cioè il numero delle sue foglie. Quindi l'albero di $\text{BinomRec}(n, k)$ ha $\binom{n}{k}$ foglie.

In un albero in cui ogni nodo interno ha esattamente due figli, il numero di nodi interni è il numero di foglie meno uno (per induzione: una singola foglia ha 0 nodi interni; unendo due alberi di questo tipo con L_1 e L_2 foglie sotto una nuova radice si ottengono $(L_1 - 1) + (L_2 - 1) + 1 = (L_1 + L_2) - 1$ nodi interni). Quindi

$$N(n, k) = \binom{n}{k} + \left(\binom{n}{k} - 1 \right) = 2\binom{n}{k} - 1.$$

Per esempio $N(4, 2) = 2 \cdot 6 - 1 = 11$, come nella domanda 1.

3.

$$N(10, 5) = 2 \cdot 252 - 1 = 503, \quad N(30, 15) = 2 \cdot 155\,117\,520 - 1 = 310\,235\,039,$$

$$N(40, 20) = 2 \cdot 137\,846\,528\,820 - 1 = 275\,693\,057\,639.$$

A 10^9 chiamate al secondo, `BinomRec(40, 20)` richiede circa 275.7 secondi, cioè più di 4.5 minuti, per calcolare un numero che si può ottenere con qualche centinaio di addizioni (si vedano le domande 4 e 5). Poiché $\binom{2m}{m} \geq \frac{4^m}{2m+1}$, il numero di chiamate cresce in modo esponenziale.

4. Memorizziamo i valori già calcolati in una tabella *memo*, indicizzata dalla coppia (i, j) :

Algorithm: BinomMemo

Input : Numeri naturali $0 \leq k \leq n$ e una tabella *memo* le cui voci sono tutte *non definite*

Output: Il coefficiente binomiale $\binom{n}{k}$

```

1 function BinomMemo(n, k, memo)
2   if k = 0 or k = n then
3     return 1
4   end
5   if memo[n][k] è non definita then
6     memo[n][k] ←
7       BinomMemo(n − 1, k − 1, memo) + BinomMemo(n − 1, k, memo)
8   end
9   return memo[n][k]
10 end
```

Numero di chiamate. A partire da (n, k) , una chiamata raggiunge le coppie $(n-s, k-t)$, dove s è il numero di passi verso il basso e $t \leq s$ è il numero di questi passi in cui k diminuisce. Sia $u = s - t$ il numero di passi in cui k non diminuisce. Una coppia è **interna** (non è un caso base) se $k - t \geq 1$ e $(n-s) - (k-t) \geq 1$, cioè

$$0 \leq t \leq k-1 \quad \text{e} \quad 0 \leq u \leq n-k-1,$$

quindi ci sono esattamente $k(n-k)$ sottoproblemi interni, e tutti vengono raggiunti. Il corpo di ciascun sottoproblema interno (le due chiamate ricorsive) viene eseguito solo la **prima** volta che viene chiamato; le volte successive il valore viene letto da *memo*. Ogni chiamata, tranne la prima, è effettuata dal corpo di un sottoproblema interno, che effettua esattamente 2 chiamate. Quindi il numero di chiamate è

$$1 + 2k(n-k).$$

Per i tre casi:

$$(10, 5) : 1 + 2 \cdot 25 = 51, \quad (30, 15) : 1 + 2 \cdot 225 = 451, \quad (40, 20) : 1 + 2 \cdot 400 = 801,$$

contro le 503, 310 235 039 e 275 693 057 639 chiamate di **BinomRec**. Il tempo è $\Theta(k(n-k))$ e la memoria è $\Theta(k(n-k))$ per la tabella più $\Theta(n)$ per la pila (profondità $\leq n$).

5. La riga i del triangolo di Pascal si calcola dalla riga $i-1$. Tenendo un solo array $c[0..k]$ e aggiornandolo da destra a sinistra (in modo che $c[j-1]$ contenga ancora il valore della riga precedente quando si aggiorna $c[j]$):

Algorithm: BinomIt

Input : Due numeri naturali n e k con $0 \leq k \leq n$

Output: Il coefficiente binomiale $\binom{n}{k}$

```

1 function BinomIt( $n, k$ )
2    $c[0] := 1; c[1..k] := 0$ 
3    $i := 1$ 
4   while  $i \leq n$  do
5      $j := \min(i, k)$ 
6     while  $j \geq 1$  do
7        $c[j] \leftarrow c[j] + c[j-1]$ 
8        $j \leftarrow j-1$ 
9     end
10     $i \leftarrow i+1$ 
11  end
12  return  $c[k]$ 
13 end
```

Dopo la i -esima iterazione del ciclo esterno, $c[j] = \binom{i}{j}$ per $0 \leq j \leq \min(i, k)$. Il ciclo interno viene eseguito al più k volte per ciascuna delle n righe: tempo $O(nk)$ (e $\Theta(nk)$ quando $k \leq n/2$), memoria aggiuntiva $\Theta(k)$, nessuna ricorsione. Per $(40, 20)$ l'algoritmo esegue $\sum_{i=1}^{40} \min(i, 20) = 210 + 400 = 610$ addizioni.

La ricorsione ingenua è esponenziale perché i sottoproblemi si sovrappongono; la memoizzazione (top-down) e il triangolo di Pascal iterativo (bottom-up) risolvono ogni sottoproblema una sola volta, con un numero polinomiale di passi. Questa è l'idea della programmazione dinamica.

Esercizio 18.5 — Algoritmi ricorsivi contro iterativi (E5)

Consideriamo i due seguenti algoritmi per i numeri di Fibonacci ($F(0) = 0$, $F(1) = 1$, $F(n) = F(n-1) + F(n-2)$).

Algorithm: FibRec

```

1 function FibRec(n)
2   if  $n \leq 1$  then
3     return n
4   end
5   return FibRec( $n - 1$ )
      + FibRec( $n - 2$ )
6 end

```

Algorithm: FibIt

```

1 function FibIt(n)
2   if  $n \leq 1$  then
3     return n
4   end
5   prev := 0
6   curr := 1
7   i := 2
8   while  $i \leq n$  do
9     next := prev + curr
10    prev ← curr
11    curr ← next
12    i ← i + 1
13  end
14  return curr
15 end

```

Supponiamo che ogni riga eseguita conti come un passo (la riga 5 conta come un passo, oltre ai passi delle due sotto-chiamate) e che il calcolatore esegua 10^9 passi al secondo.

Domande

1. Determinare il numero esatto di passi $S_{it}(n)$ eseguiti da **FibIt** per $n \geq 2$.
2. Determinare il numero esatto di passi $S_{rec}(n)$ eseguiti da **FibRec**, usando il fatto che **FibRec**(n) esegue $2F(n + 1) - 1$ chiamate.

3. Stimare i tempi di esecuzione dei due algoritmi per $n = 50$, sapendo che

$$F(51) = 20\,365\,011\,074.$$

4. Stimare il tempo di esecuzione di **FibRec** per $n = 60$ e $n = 100$, sapendo che

$$F(61) = 2\,504\,730\,781\,961 \quad \text{e} \quad F(101) \approx 5.73 \cdot 10^{20}.$$

5. Determinare il massimo n per cui **FibRec** termina entro un'ora, sapendo che

$$F(58) = 591\,286\,729\,879 \quad \text{e} \quad F(59) = 956\,722\,026\,041.$$

Soluzione

1. Per $n \geq 2$ il numero di volte che ciascuna riga di **FibIt** viene eseguita è:

- La **riga** 2 viene eseguita 1 volta (il return nella riga successiva non viene eseguito)
- Le **righe** 5, 6, 7 vengono eseguite 1 volta ciascuna
- La **riga** 8 viene eseguita n volte ($i = 2, 3, \dots, n+1$)
- Le **righe** 9, 10, 11, 12 vengono eseguite $n-1$ volte ciascuna
- La **riga** 14 viene eseguita 1 volta

Quindi

$$S_{it}(n) = 1 + 3 + n + 4(n-1) + 1 = 5n + 1 = \Theta(n).$$

2. Ogni chiamata di **FibRec** esegue esattamente 2 passi propri: la riga 2 e poi o la riga 3 (caso base) o la riga 5. Poiché **FibRec**(n) esegue $2F(n+1) - 1$ chiamate,

$$S_{rec}(n) = 2(2F(n+1) - 1) = 4F(n+1) - 2 = \Theta(\varphi^n).$$

3. Per $n = 50$:

$$S_{it}(50) = 5 \cdot 50 + 1 = 251 \text{ passi} \implies \frac{251}{10^9} \text{ s} = 251 \text{ ns},$$

$$S_{rec}(50) = 4 \cdot 20\,365\,011\,074 - 2 = 81\,460\,044\,294 \text{ passi} \implies \approx 81.5 \text{ s}.$$

L'algoritmo ricorsivo è circa $3 \cdot 10^8$ volte più lento.

4. Per $n = 60$:

$$\begin{aligned} S_{rec}(60) &= 4 \cdot 2\,504\,730\,781\,961 - 2 = 10\,018\,923\,127\,842 \text{ passi} \\ &\implies \approx 10\,019 \text{ s} \approx 2.8 \text{ ore}. \end{aligned}$$

Per $n = 100$:

$$S_{rec}(100) \approx 4 \cdot 5.73 \cdot 10^{20} \approx 2.29 \cdot 10^{21} \text{ passi} \implies \approx 2.29 \cdot 10^{12} \text{ s}.$$

Poiché un anno ha circa $3.156 \cdot 10^7$ secondi, questo corrisponde a circa $7.3 \cdot 10^4$ anni. Nel frattempo, **FibIt**(100) esegue 501 passi, cioè circa mezzo microsecondo.

5. Un'ora corrisponde a $3600 \cdot 10^9 = 3.6 \cdot 10^{12}$ passi. Servono

$$4F(n+1) - 2 \leq 3.6 \cdot 10^{12} \iff F(n+1) \leq 9 \cdot 10^{11} + 0.5.$$

Poiché $F(58) = 591\,286\,729\,879 \leq 9 \cdot 10^{11} < F(59) = 956\,722\,026\,041$, la condizione vale per $n+1 \leq 58$. Il massimo n è $n = 57$, con

$$S_{rec}(57) = 4 \cdot 591\,286\,729\,879 - 2 = 2\,365\,146\,919\,514 \text{ passi} \approx 39 \text{ minuti},$$

mentre $n = 58$ richiede già $4F(59) - 2 \approx 3.83 \cdot 10^{12}$ passi, cioè circa 64 minuti. Ogni volta che n aumenta di 1, il tempo di **FibRec** viene moltiplicato per circa $\varphi \approx 1.618$: un calcolatore 10 volte più veloce permetterebbe di aumentare n solo di $\log_{\varphi} 10 \approx 4.8$.

Con un algoritmo esponenziale, calcolatori più veloci non aiutano: la dimensione dei problemi risolvibili cresce solo di una costante additiva. L'algoritmo iterativo, con $5n + 1$ passi, calcola $F(57)$ in meno di 0.3 microsecondi.