



SAPIENZA
UNIVERSITÀ DI ROMA

Algoritmi

Volume 5 — Ricorsione

La ricorsione: algoritmi ricorsivi classici, equazioni di ricorrenza, il Master theorem, algoritmi ricorsivi (somma di un array, merge sort, ricerca binaria, sotto-array di somma massima) e il confronto fra versioni ricorsive e iterative.

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	3
14.1	Algoritmi ricorsivi	4
14.2	Albero delle chiamate ricorsive e call stack	6
14.3	Correttezza degli algoritmi ricorsivi	8
14.4	Fattoriale	10
14.5	Numeri di Fibonacci	14
14.6	Massimo comun divisore	19
14.7	Elevamento a potenza veloce	23
14.8	Torri di Hanoi	27
14.9	Riepilogo	32
15	Equazioni di ricorrenza	33
	Introduzione	33
15.1	Equazioni di ricorrenza	34
15.2	Il metodo di espansione	40
15.3	Il metodo di sostituzione	43
15.4	Il metodo dell'albero di ricorsione	48
15.5	Verso il Master theorem	54
15.6	Riepilogo: quale metodo quando	55
16	Il Master theorem	56
	Introduzione	56
16.1	Ricorrenze in forma standard	56
16.2	Il Master theorem	57
16.3	Applicare il Master theorem	60
16.4	Quando il Master theorem non si applica	64
16.5	Dimostrazione del Master theorem	66
16.6	Riepilogo	76
17	Algoritmi ricorsivi	78
	Introduzione	78
17.1	La somma degli elementi di un array	79
17.2	Merge Sort	91



17.3 Ricerca binaria ricorsiva	105
17.4 Il problema del sotto-array di somma massima	116
17.5 Riepilogo	133
18 Algoritmi ricorsivi e iterativi a confronto	134
Introduzione	134
18.1 Ricorsione e iterazione	135
18.2 Confronto in sintesi	137
18.3 Il divide et impera dà il guadagno, non la ricorsione in sé	139
18.4 Quando i sottoproblemi si sovrappongono	144
18.5 Spazio	148
18.6 Implementazione in C++	152
18.7 Esperimento	154
18.8 Linee guida	158

Ricorsione

Nei capitoli precedenti ogni algoritmo era **iterativo**: una sequenza di istruzioni e di cicli, di cui abbiamo contato i passi riga per riga (capitolo «Tempo di esecuzione») e di cui abbiamo dimostrato la correttezza con gli invarianti di ciclo (capitolo «Correttezza»). In questo capitolo studiamo un modo diverso di progettare gli algoritmi: un algoritmo può **richiamare se stesso** su istanze più piccole dello stesso problema.

Il piano del capitolo è il seguente.

- Nella Sezione 14.1 definiamo gli algoritmi ricorsivi, i casi base e i passi ricorsivi, e vediamo perché un algoritmo ricorsivo termina.
- Nella Sezione 14.2 descriviamo come viene eseguito un algoritmo ricorsivo: l'**albero delle chiamate ricorsive** (che determina il numero di passi) e il **call stack** (che determina la memoria).
- Nella Sezione 14.3 mostriamo che la correttezza di un algoritmo ricorsivo si dimostra per **induzione forte**.
- Nelle Sezioni 14.4–14.8 applichiamo questi strumenti a cinque classici algoritmi ricorsivi: il fattoriale, i numeri di Fibonacci, il massimo comun divisore, l'elevamento a potenza veloce e le Torri di Hanoi. Per ciascuno di essi dimostriamo la correttezza, contiamo il numero di passi srotolando un'equazione di ricorrenza e forniamo un'implementazione in C++ eseguita passo per passo.

14.1 Algoritmi ricorsivi

- Molti problemi hanno una struttura **autosimile** naturale: la soluzione di un'istanza si ottiene facilmente dalla soluzione di una o più **istanze più piccole dello stesso problema**.
- Per esempio, per ogni numero naturale $n \geq 1$,

$$n! = n \cdot (n - 1)!$$

cioè il fattoriale di n si ottiene dal fattoriale del numero più piccolo $n - 1$.

- Gli algoritmi che sfruttano questa struttura si chiamano algoritmi ricorsivi.

Definizione 14.1: algoritmo ricorsivo

Un **algoritmo ricorsivo** è un algoritmo che risolve un'istanza di un problema **richiamando se stesso** su una o più **istanze più piccole** dello stesso problema, e combinando poi i risultati di queste chiamate.

- Ogni algoritmo ricorsivo è composto da due parti:
 1. **Caso/i base**: una o più istanze abbastanza piccole da essere risolte direttamente, senza alcuna chiamata ricorsiva (per esempio, $0! = 1$).
 2. **Passo ricorsivo**: su ogni altra istanza, l'algoritmo richiama se stesso su istanze più piccole e combina i risultati restituiti (per esempio, $n! = n \cdot (n - 1)!$).
- Ogni esecuzione dell'algoritmo su un dato input si chiama **chiamata** (o invocazione). Una chiamata effettuata dall'interno di un'altra chiamata è una **chiamata ricorsiva**.
- Per misurare quanto sia “piccola” un'istanza, associamo a ogni input una **dimensione**, che è un numero naturale (per esempio il valore n del numero in input, oppure il numero n di elementi di un array $a[1..n]$).

14.1.1 Terminazione

- Un algoritmo ricorsivo termina soltanto se la catena delle chiamate ricorsive raggiunge prima o poi un caso base.
- Se la dimensione dell'istanza diminuisce strettamente a ogni chiamata ricorsiva, otteniamo una successione strettamente decrescente di numeri naturali

$$n_0 > n_1 > n_2 > \dots$$

- Non esiste alcuna successione infinita strettamente decrescente di numeri naturali (per la proprietà del buon ordinamento di $\mathbb{N}$): dopo al più n_0 chiamate la dimensione non può più diminuire. Quindi, se ogni istanza che non è un caso base viene ridotta a istanze strettamente più piccole, un caso base viene necessariamente raggiunto dopo un numero finito di chiamate.

Idea chiave della ricorsione.

- Risolvere direttamente i **casi base**.
- Ridurre ogni altra istanza a una o più istanze **strettamente più piccole** dello stesso problema, risolverle con chiamate ricorsive e combinare i loro risultati.
- Poiché la dimensione è un numero naturale che **diminuisce strettamente** a ogni chiamata ricorsiva, un caso base viene sempre raggiunto e l'algoritmo **termina**.

Esempio 14.1: Che cosa può andare storto

- **Caso base mancante.** Consideriamo la funzione `WrongFactorial( $n$ )` che restituisce $n \cdot \text{WrongFactorial}(n - 1)$, senza alcun caso base. La chiamata `WrongFactorial(2)` richiama `WrongFactorial(1)`, che richiama `WrongFactorial(0)`, che richiama `WrongFactorial(-1)`, e così via all'infinito: la ricorsione non si ferma mai.
 - **Dimensione che non diminuisce.** Una funzione che, sull'input n , richiama se stessa sull'input n (oppure $n + 1$) non raggiunge mai un caso base $n = 0$ quando $n \geq 1$.
 - In pratica, una ricorsione infinita viene fermata dal calcolatore quando la memoria riservata alle chiamate si esaurisce (un errore di **stack overflow**).
- Per contare il numero di passi di un algoritmo ricorsivo usiamo le stesse convenzioni del capitolo «Tempo di esecuzione»: ogni esecuzione di una riga di pseudocodice è **un passo**. Una riga che contiene una chiamata ricorsiva conta come un passo **più** tutti i passi eseguiti da quella chiamata ricorsiva.
 - Di conseguenza, il numero di passi $S(n)$ di un algoritmo ricorsivo si esprime naturalmente in termini del numero di passi su istanze più piccole, cioè mediante un'**equazione di ricorrenza**, come

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

In questo capitolo risolviamo le equazioni di ricorrenza dei conteggi dei passi per **semplice srotolamento** (detto anche **espansione**): sostituiamo ripetutamente $S(\cdot)$ con la sua definizione fino a raggiungere un caso base.

14.2 Albero delle chiamate ricorsive e call stack

14.2.1 Albero delle chiamate ricorsive

Definizione 14.2: albero delle chiamate ricorsive

L'**albero delle chiamate ricorsive** di un algoritmo ricorsivo su un dato input è l'albero radicato in cui:

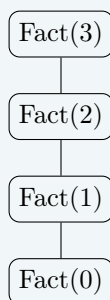
- ogni nodo è una chiamata dell'algoritmo;
- la radice è la chiamata iniziale;
- i figli di un nodo sono le chiamate ricorsive effettuate direttamente da quella chiamata, da sinistra a destra nell'ordine in cui vengono eseguite.

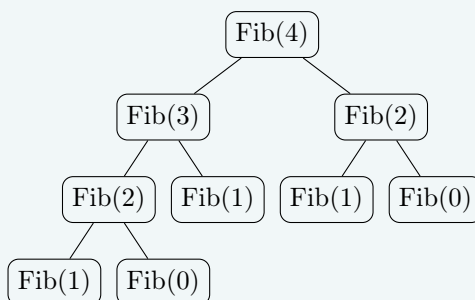
Le foglie dell'albero sono le chiamate che non effettuano chiamate ricorsive (tipicamente i casi base).

- Il numero totale di chiamate eseguite dall'algoritmo è il **numero di nodi** dell'albero.
- Se ogni chiamata esegue un numero costante di passi (esclusi i passi delle sue chiamate ricorsive), allora il numero di passi dell'algoritmo è proporzionale al numero di nodi dell'albero.
- Le chiamate vengono eseguite seguendo una visita **in profondità** dell'albero: una chiamata inizia, esegue completamente la sua prima chiamata ricorsiva (e l'intero sottoalbero sottostante), poi la seconda, e così via; restituisce il suo risultato solo quando tutte le sue chiamate ricorsive sono terminate.

Esempio 14.2: Due forme di alberi delle chiamate ricorsive

- Se ogni chiamata effettua **una** sola chiamata ricorsiva (come per $n! = n \cdot (n-1)!$), l'albero è un **cammino** (a sinistra).
- Se ogni chiamata effettua **due** chiamate ricorsive (come per i numeri di Fibonacci, definiti da $F(n) = F(n-1) + F(n-2)$ con $F(0) = 0$ e $F(1) = 1$), l'albero si **ramifica** e il numero di nodi può crescere molto rapidamente (a destra).





14.2.2 Call stack

- Quando una chiamata inizia, il calcolatore riserva un pezzo di memoria, detto **stack frame** (o record di attivazione), che contiene i parametri della chiamata, le sue variabili locali e il punto del programma in cui l'esecuzione deve proseguire quando la chiamata termina.
- I frame sono memorizzati in un'area di memoria organizzata come una **pila** (lo **stack** delle chiamate, o **call stack**):
 - quando una chiamata inizia, il suo frame viene inserito (**push**) in cima alla pila;
 - quando una chiamata termina, il suo frame viene rimosso (**pop**) dalla cima della pila, e l'esecuzione prosegue nella chiamata il cui frame è ora in cima.
- In ogni momento, i frame nella pila sono esattamente le chiamate sul cammino dalla radice dell'albero delle chiamate ricorsive fino alla chiamata attualmente in esecuzione.

Definizione 14.3: profondità della ricorsione

La **profondità della ricorsione** di un algoritmo ricorsivo su un dato input è il massimo numero di chiamate simultaneamente attive, cioè il massimo numero di nodi su un cammino dalla radice a una foglia dell'albero delle chiamate ricorsive.

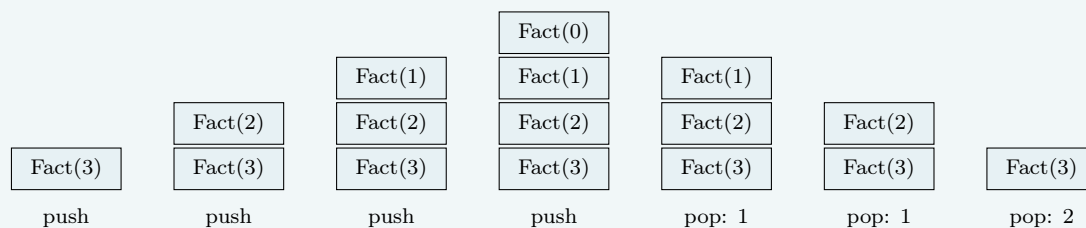
Il massimo numero di stack frame simultaneamente in memoria è uguale alla **profondità della ricorsione**. Se ogni frame usa una quantità costante di memoria, la memoria usata dal call stack è

$$\Theta(\text{profondità della ricorsione}).$$

- Si noti la differenza: il **numero di passi** dipende dal **numero di nodi** dell'albero, mentre la **memoria** usata dal call stack dipende solo dalla sua **profondità**.
- Per l'albero di **Fib(4)** mostrato sopra, ci sono 9 chiamate, ma al più 4 di esse sono contemporaneamente nella pila (per esempio **Fib(4)**, **Fib(3)**, **Fib(2)**, **Fib(1)**).

Esempio 14.3: Evoluzione del call stack per Fact(3)

La figura mostra il call stack (la cima della pila è in alto) dopo ogni push e dopo ogni pop, durante il calcolo di $3!$ con la ricorsione $n! = n \cdot (n-1)!$, $0! = 1$.



- Vengono inseriti quattro frame: **Fact(3)**, **Fact(2)**, **Fact(1)**, **Fact(0)**.
- **Fact(0)** è un caso base: restituisce 1 e il suo frame viene rimosso. Poi **Fact(1)** restituisce $1 \cdot 1 = 1$, **Fact(2)** restituisce $2 \cdot 1 = 2$ e, infine, **Fact(3)** restituisce $3 \cdot 2 = 6$ e la pila diventa vuota.
- La profondità della ricorsione è 4.

- La dimensione del call stack è limitata (tipicamente alcuni megabyte). Una ricorsione molto profonda (per esempio, milioni di chiamate annidate) causa uno **stack overflow**, anche se l'algoritmo è corretto e termina.

14.3 Correttezza degli algoritmi ricorsivi

- Lo strumento naturale per dimostrare la correttezza di un algoritmo ricorsivo è l'**induzione matematica** sulla dimensione dell'input.
- Poiché una chiamata ricorsiva può essere effettuata su una qualunque istanza più piccola (non soltanto sull'istanza di dimensione $n-1$), usiamo l'**induzione forte**.

Induzione forte. Sia $P(n)$ un'affermazione che dipende da $n \in \mathbb{N}$, e sia $n_0 \in \mathbb{N}$. Se

1. **caso base:** $P(n_0)$ è vera, e
2. **passo induttivo:** per ogni $n > n_0$, se $P(k)$ è vera per ogni k con $n_0 \leq k < n$ (**ipotesi induttiva**), allora $P(n)$ è vera,

allora $P(n)$ è vera per ogni numero naturale $n \geq n_0$.

- A volte conviene dimostrare più di un caso base (per esempio $n = 0$ e $n = 1$), e poi il passo induttivo solo per n maggiore di tutti i casi base.
- La struttura di un algoritmo ricorsivo rispecchia la struttura di una dimostrazione per induzione:
 - i **casi base** dell'algoritmo corrispondono ai **casi base** dell'induzione;

- nel **passo ricorsivo** possiamo assumere (ipotesi induttiva) che le chiamate ricorsive, effettuate su input più piccoli, restituiscano risultati corretti, e dobbiamo soltanto mostrare che i risultati vengano **combinati** correttamente.

Osservazione 14.4: correttezza degli algoritmi ricorsivi

Consideriamo un algoritmo ricorsivo in cui ogni input ha una dimensione in $\mathbb{N}$. Supponiamo che:

1. su ogni input che è un caso base, l'algoritmo restituisca l'output corretto;
2. su ogni altro input di dimensione n , l'algoritmo effettui un numero finito di chiamate ricorsive, tutte su input di dimensione strettamente minore di n , e restituisca l'output corretto ogniqualvolta tutte queste chiamate ricorsive restituiscono l'output corretto.

Allora l'algoritmo termina e restituisce l'output corretto su ogni input.

Dimostrazione. Sia $P(n)$ l'affermazione: “su ogni input di dimensione n l'algoritmo termina e restituisce l'output corretto”. Dimostriamo $P(n)$ per ogni $n \in \mathbb{N}$ per induzione forte.

Sia $n \in \mathbb{N}$ e supponiamo che $P(k)$ sia vera per ogni $k < n$ (per $n = 0$ questa ipotesi è vuota). Consideriamo un input di dimensione n .

- Se l'input è un caso base, allora l'algoritmo termina e restituisce l'output corretto per l'ipotesi 1.
- Altrimenti, per l'ipotesi 2, l'algoritmo effettua un numero finito di chiamate ricorsive su input di dimensione $k < n$. Per l'ipotesi induttiva, ciascuna di queste chiamate termina e restituisce l'output corretto. Quindi l'algoritmo termina e, per l'ipotesi 2, restituisce l'output corretto.

Quindi $P(n)$ è vera, e per induzione forte $P(n)$ è vera per ogni $n \in \mathbb{N}$. □

14.3.1 Legame con gli invarianti di ciclo

- Nel capitolo «Correttezza» abbiamo dimostrato la correttezza degli algoritmi iterativi per mezzo degli **invarianti di ciclo**, che hanno anch'essi la struttura di una dimostrazione per induzione.
- Le tre proprietà di un invariante di ciclo corrispondono alle parti della dimostrazione per induzione di un algoritmo ricorsivo:

Invariante di ciclo (algoritmo iterativo)	Induzione (algoritmo ricorsivo)
Inizializzazione: vero prima della prima iterazione	Caso base (o casi base) della ricorsione
Conservazione: preservato da ogni iterazione	Passo induttivo: corretta combinazione dei risultati
Terminazione: il ciclo si ferma	La dimensione diminuisce strettamente verso un caso base

- La differenza principale è la direzione: un algoritmo iterativo costruisce la soluzione “dal basso” (dall’inizializzazione verso l’ultima iterazione), mentre un algoritmo ricorsivo parte “dall’alto” (l’istanza originale) e la riduce a istanze più piccole fino a raggiungere un caso base.

14.4 Fattoriale

Dato un numero naturale n , consideriamo il problema di calcolare il **fattoriale** di n :

$$n! = 1 \cdot 2 \cdot \dots \cdot n \quad \text{con } 0! = 1.$$

- Poiché $n! = n \cdot (n - 1)!$ per ogni $n \geq 1$, il fattoriale ha la seguente definizione ricorsiva:

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

- Questo è lo pseudocodice dell'algoritmo ricorsivo **Factorial**:

Algorithm: Factorial

Input : Un numero naturale n

Output: Il fattoriale $n!$

```

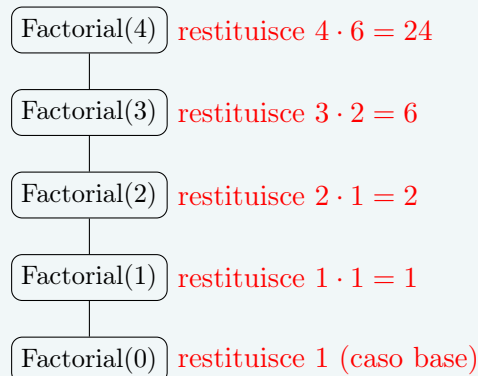
1 function Factorial( $n$ )
2   if  $n = 0$  then
3     return 1                                // caso base
4   end
5   return  $n \cdot \text{Factorial}(n - 1)$           // passo ricorsivo
6 end
```

- Spiegazione dell'algoritmo:
 - Il caso base è $n = 0$, per il quale l'algoritmo restituisce $0! = 1$.
 - Per $n \geq 1$, l'algoritmo richiama se stesso sull'input più piccolo $n - 1$ e moltiplica il risultato per n .

- La dimensione dell'input è il valore n , che diminuisce di uno a ogni chiamata ricorsiva. Quindi il caso base viene raggiunto dopo n chiamate ricorsive.

Esempio 14.4: Albero delle chiamate ricorsive di Factorial(4)

L'albero è un cammino con 5 nodi. Il valore restituito da ciascuna chiamata è scritto in rosso accanto alla chiamata.



Le chiamate vengono effettuate dall'alto verso il basso; i risultati vengono restituiti dal basso verso l'alto. La profondità della ricorsione è 5.

14.4.1 Correttezza

Osservazione 14.5: correttezza di Factorial

Per ogni numero naturale n , la chiamata **Factorial**(n) termina e restituisce $n!$.

Dimostrazione. Procediamo per induzione su n .

- **Caso base** ($n = 0$): la condizione della riga 2 è vera e l'algoritmo restituisce $1 = 0!$ alla riga 3.
- **Passo induttivo** ($n \geq 1$): supponiamo che **Factorial**($n - 1$) termini e restituisca $(n - 1)!$ (ipotesi induttiva). Poiché $n \neq 0$, l'algoritmo esegue la riga 5 e restituisce

$$n \cdot \text{Factorial}(n - 1) = n \cdot (n - 1)! = n!.$$

Quindi **Factorial**(n) restituisce $n!$ per ogni $n \in \mathbb{N}$. □

14.4.2 Numero di passi

- Sia $S(n)$ il numero di passi eseguiti da **Factorial** sull'input n (per ogni valore di n c'è un solo input, quindi questo è anche il caso peggiore).
- Se $n = 0$, le righe 2 e 3 vengono eseguite una volta ciascuna:

$$S(0) = 2.$$

- Se $n \geq 1$, le righe 2 e 5 vengono eseguite una volta ciascuna, e la riga 5 esegue anche tutti i passi della chiamata **Factorial**($n - 1$):

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

- Risolviamo l'equazione di ricorrenza srotolandola:

$$\begin{aligned} S(n) &= S(n - 1) + 2 \\ &= (S(n - 2) + 2) + 2 = S(n - 2) + 2 \cdot 2 \\ &= (S(n - 3) + 2) + 2 \cdot 2 = S(n - 3) + 3 \cdot 2 \\ &\vdots \\ &= S(n - k) + k \cdot 2 \quad \forall k \in \{1, 2, \dots, n\}. \end{aligned}$$

Scegliendo $k = n$ (caso base), otteniamo

$$S(n) = S(0) + 2n = 2n + 2.$$

- Quindi,

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

poiché

$$\lim_{n \rightarrow \infty} \frac{2n + 2}{n} = \lim_{n \rightarrow \infty} \left(2 + \frac{2}{n} \right) = 2 \in \mathbb{R}_{>0}.$$

- La profondità della ricorsione è $n + 1$ (le chiamate con input $n, n - 1, \dots, 0$ sono tutte contemporaneamente nella pila), quindi il call stack usa una memoria $\Theta(n)$. L'algoritmo iterativo che calcola $n!$ con un ciclo **while** usa soltanto una quantità costante di memoria.
- Esprimiamo ora il numero di passi in funzione della dimensione dell'input $|I|$. Come nel capitolo «Tempo di esecuzione», contiamo ogni moltiplicazione come un passo. L'input è il numero naturale n , che richiede $\lceil \log_2 n \rceil + 1$ bit (per $n \geq 1$), quindi

$$|I| = \Theta(\log_2 n).$$

Questo significa che esistono costanti $c_1, c_2 > 0$ tali che, per ogni n sufficientemente grande,

$$c_1 \log_2 n \leq |I| \leq c_2 \log_2 n \quad \Longleftrightarrow \quad 2^{\frac{1}{c_2}|I|} \leq n \leq 2^{\frac{1}{c_1}|I|}.$$

Poiché $S(n) = \Theta(n)$, esistono costanti $a_1, a_2 > 0$ tali che $a_1 n \leq S(n) \leq a_2 n$ per ogni n sufficientemente grande, e quindi

$$a_1 2^{\frac{1}{c_2}|I|} \leq S(n) \leq a_2 2^{\frac{1}{c_1}|I|},$$

cioè $T(|I|) = 2^{\Theta(|I|)}$.

Il numero di passi dell'algoritmo **Factorial** è $S(n) = 2n + 2 = \Theta(n)$, e la sua profondità di ricorsione è $n + 1$. In funzione della dimensione dell'input $|I| = \Theta(\log_2 n)$, il tempo di esecuzione nel caso peggiore è

$$T(|I|) = 2^{\Theta(|I|)},$$

cioè è lineare nel valore numerico n , ma esponenziale nella dimensione dell'input $|I|$.

14.4.3 Implementazione in C++

- Questa è un'implementazione in C++ dell'algoritmo **Factorial**. Usiamo il tipo `unsigned long long` (64 bit): il risultato è corretto solo per $n \leq 20$, poiché $21! > 2^{64} - 1$.

```

1 // Restituisce n! per n >= 0 (corretto per 0 <= n <= 20 con interi a 64 bit)
2 unsigned long long Factorial(int n)
3 {
4     if (n == 0)
5     {
6         return 1;                // caso base
7     }
8     return n * Factorial(n - 1);  // passo ricorsivo
9 }
```

Esempio 14.5: esecuzione dell'implementazione in C++ di Factorial per $n = 4$

Chiamate (lo stack cresce)

1. **Factorial(4)**: $4 \neq 0 \Rightarrow$ deve calcolare $4 \cdot \text{Factorial}(3)$ e attende. stack: [4]
2. **Factorial(3)**: $3 \neq 0 \Rightarrow$ deve calcolare $3 \cdot \text{Factorial}(2)$ e attende. stack: [4, 3]
3. **Factorial(2)**: $2 \neq 0 \Rightarrow$ deve calcolare $2 \cdot \text{Factorial}(1)$ e attende. stack: [4, 3, 2]
4. **Factorial(1)**: $1 \neq 0 \Rightarrow$ deve calcolare $1 \cdot \text{Factorial}(0)$ e attende. stack: [4, 3, 2, 1]
5. **Factorial(0)**: $n = 0 \Rightarrow$ caso base. stack: [4, 3, 2, 1, 0]

Ritorni (lo stack si riduce)

1. **Factorial(0)** restituisce 1. stack: [4, 3, 2, 1]
2. **Factorial(1)** restituisce $1 \cdot 1 = 1$. stack: [4, 3, 2]
3. **Factorial(2)** restituisce $2 \cdot 1 = 2$. stack: [4, 3]
4. **Factorial(3)** restituisce $3 \cdot 2 = 6$. stack: [4]
5. **Factorial(4)** restituisce $4 \cdot 6 = 24$. stack: []

L'algoritmo termina con risultato $24 = 4!$, dopo 5 chiamate.

14.5 Numeri di Fibonacci

Dato un numero naturale n , consideriamo il problema di calcolare l' n -esimo **numero di Fibonacci** $F(n)$, definito come

$$F(n) = \begin{cases} n & \text{se } n \in \{0, 1\} \\ F(n-1) + F(n-2) & \text{se } n \geq 2 \end{cases}$$

- I primi numeri di Fibonacci sono:

n	0	1	2	3	4	5	6	7	8	9	10
$F(n)$	0	1	1	2	3	5	8	13	21	34	55

- La definizione è essa stessa ricorsiva, e fornisce direttamente il seguente algoritmo ricorsivo (ingenuo) `Fib`, con **due** casi base e **due** chiamate ricorsive:

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  $n$                                 // casi base
4   end
5   return Fib( $n-1$ ) + Fib( $n-2$ )                // passo ricorsivo
6 end
```

Osservazione 14.6: correttezza di Fib

Per ogni numero naturale n , la chiamata `Fib( $n$ )` termina e restituisce $F(n)$.

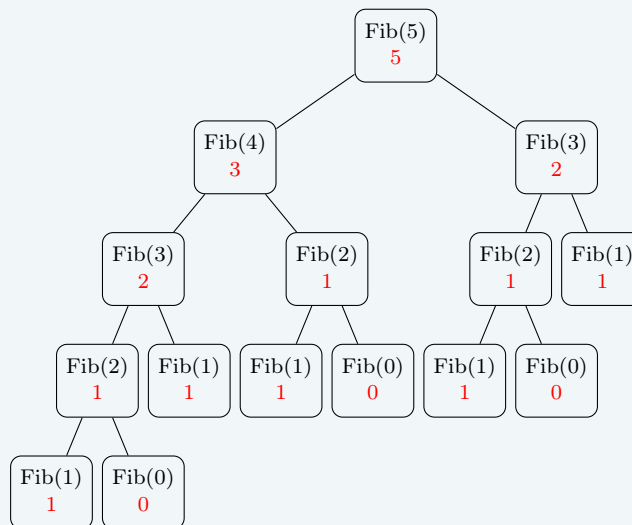
Dimostrazione. Procediamo per induzione forte su n .

- **Casi base** ($n \in \{0, 1\}$): l'algoritmo restituisce $n = F(n)$ alla riga 3.
- **Passo induttivo** ($n \geq 2$): per ipotesi induttiva le due chiamate sugli input più piccoli $n-1$ e $n-2$ restituiscono $F(n-1)$ e $F(n-2)$, e la riga 5 restituisce la loro somma $F(n)$.

Qui servono due casi base, perché il passo induttivo per n usa sia $n-1$ sia $n-2$. $\square$

Esempio 14.6: Albero delle chiamate ricorsive di Fib(5)

Ogni nodo è una chiamata; il valore restituito dalla chiamata è scritto in rosso sotto di essa.



- L'albero ha 15 nodi: l'algoritmo effettua 15 chiamate per calcolare $F(5) = 5$.
- Gli stessi sottoproblemi vengono risolti molte volte: **Fib(3)** viene calcolato 2 volte, **Fib(2)** 3 volte, **Fib(1)** 5 volte e **Fib(0)** 3 volte.
- La profondità della ricorsione è 5 (per esempio, il cammino **Fib(5)**, **Fib(4)**, **Fib(3)**, **Fib(2)**, **Fib(1)**).

14.5.1 Numero di chiamate e numero di passi

- Sia $C(n)$ il numero di chiamate (nodi dell'albero) eseguite da **Fib**(n). Ogni chiamata con $n \geq 2$ genera una chiamata su $n - 1$ e una chiamata su $n - 2$, quindi

$$C(n) = \begin{cases} 1 & \text{se } n \in \{0, 1\} \\ C(n-1) + C(n-2) + 1 & \text{se } n \geq 2 \end{cases}$$

- I primi valori sono $C(0) = 1$, $C(1) = 1$, $C(2) = 3$, $C(3) = 5$, $C(4) = 9$, $C(5) = 15$, $C(6) = 25$. Sono legati ai numeri di Fibonacci stessi:

Osservazione 14.7: numero di chiamate di Fib

Per ogni numero naturale n ,

$$C(n) = 2F(n+1) - 1.$$

Dimostrazione. Procediamo per induzione forte su n .

- **Casi base:** $C(0) = 1 = 2 \cdot 1 - 1 = 2F(1) - 1$ e $C(1) = 1 = 2 \cdot 1 - 1 = 2F(2) - 1$.
- **Passo induttivo** ($n \geq 2$): supponiamo $C(k) = 2F(k+1) - 1$ per ogni $k < n$.

Allora

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

□

- Per esempio, $C(5) = 2F(6) - 1 = 2 \cdot 8 - 1 = 15$, come nell'Esempio 14.6.
- Ogni chiamata esegue esattamente 2 righe (le righe 2-3 per un caso base, le righe 2 e 5 altrimenti), quindi il numero di passi $S(n)$ soddisfa

$$S(n) = S(n-1) + S(n-2) + 2 \quad (n \geq 2), \quad S(0) = S(1) = 2,$$

e perciò $S(n) = 2C(n) = 4F(n+1) - 2$.

- Il numero di chiamate cresce **esponenzialmente** in n :

Osservazione 14.8: numero esponenziale di chiamate di Fib

Per ogni numero naturale $n \geq 2$,

$$2^{n/2} \leq C(n) \leq 2^n.$$

Dimostrazione. Limite inferiore. Procediamo per induzione forte su $n \geq 2$.

- **Casi base:** $C(2) = 3 \geq 2 = 2^{2/2}$ e $C(3) = 5 \geq 2\sqrt{2} = 2^{3/2}$.
- **Passo induttivo** ($n \geq 4$): supponiamo l'affermazione vera per ogni k con $2 \leq k < n$; in particolare per $k = n-2 \geq 2$. Poiché $C(n-1) \geq C(n-2)$ (la successione C è non decrescente),

$$C(n) = C(n-1) + C(n-2) + 1 > 2C(n-2) \geq 2 \cdot 2^{(n-2)/2} = 2^{n/2}.$$

Limite superiore. Anche per induzione forte: $C(2) = 3 \leq 4$, $C(3) = 5 \leq 8$ e, per $n \geq 4$,

$$C(n) = C(n-1) + C(n-2) + 1 \leq 2^{n-1} + 2^{n-2} + 2^{n-2} = 2^n,$$

dove abbiamo usato $1 \leq 2^{n-2}$.

□

- Si noti che $C(1) = 1 < 2^{1/2}$: per questo il limite inferiore è enunciato per $n \geq 2$.
- Quindi $S(n) = 2C(n) = 2^{\Theta(n)}$: l'algoritmo ricorsivo ingenuo è **esponenziale** in n . Per esempio, il calcolo di $F(30) = 832040$ richiede $C(30) = 2F(31) - 1 = 2 \cdot 1346269 - 1 = 2692537$ chiamate.
- (Si può dimostrare che $F(n)$ cresce come $\varphi^n / \sqrt{5}$, dove $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$ è la sezione aurea; quindi $C(n) = \Theta(\varphi^n)$.)

14.5.2 Un algoritmo iterativo

- La ragione del numero esponenziale di chiamate è che l'algoritmo ingenuo **risolve più e più volte gli stessi sottoproblemi** (si veda l'Esempio 14.6). Un algoritmo iterativo calcola ciascun numero di Fibonacci una sola volta, a partire da $F(0)$ e $F(1)$ in su, conservando soltanto gli ultimi due valori:

Algorithm: IterativeFib

Input : Un numero naturale n

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

```

1 function IterativeFib( $n$ )
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     $prev \leftarrow curr$ 
11     $curr \leftarrow next$ 
12     $i \leftarrow i + 1$ 
13  end
14  return  $curr$ 
15 end

```

- Per $n \geq 2$: le righe 2, 5, 6, 7, 14 vengono eseguite una volta, la riga 8 viene eseguita n volte e le righe 9–12 vengono eseguite $n - 1$ volte. Quindi

$$S(n) = 5 + n + 4(n - 1) = 5n + 1 = \Theta(n).$$

- (Invariante di ciclo: prima di ogni verifica della riga 8, $prev = F(i - 2)$ e $curr = F(i - 1)$; alla fine $i = n + 1$ e $curr = F(n)$.)

n	$F(n)$	passi di Fib: $4F(n + 1) - 2$	passi di IterativeFib: $5n + 1$
5	5	30	26
10	55	354	51
20	6765	43782	101
30	832040	5385074	151

L'algoritmo ricorsivo ingenuo **Fib** esegue $S(n) = 4F(n + 1) - 2 = 2^{\Theta(n)}$ passi (esponenziale in n), mentre **IterativeFib** esegue $S(n) = 5n + 1 = \Theta(n)$ passi (lineare in n). Poiché $|I| = \Theta(\log_2 n)$, in funzione della dimensione dell'input l'algoritmo ingenuo è addirittura doppiamente esponenziale, mentre quello iterativo è esponenziale (come **Factorial**).

- La ricorsione in sé non è il problema: il problema è la **ripetizione di sottoproblemi identici**. Memorizzare la soluzione di ciascun sottoproblema la prima volta che viene risolto, e riutilizzarla in seguito, è l'idea chiave della tecnica chiamata **Programmazione Dinamica**.

14.5.3 Implementazione in C++

- Questa è un'implementazione in C++ dell'algoritmo ricorsivo ingenuo e dell'algoritmo iterativo:

```

1 // Algoritmo ricorsivo ingenuo: numero esponenziale di chiamate
2 long long Fib(int n)
3 {
4     if (n <= 1)
5     {
6         return n;                // casi base F(0)=0, F(1)=1
7     }
8     return Fib(n - 1) + Fib(n - 2); // due chiamate ricorsive
9 }
10
11 // Algoritmo iterativo: n-1 iterazioni
12 long long IterativeFib(int n)
13 {
14     if (n <= 1)
15     {
16         return n;
17     }
18     long long prev = 0; // F(i-2)
19     long long curr = 1; // F(i-1)
20     for (int i = 2; i <= n; i++)
21     {
22         long long next = prev + curr;
23         prev = curr;
24         curr = next;
25     }
26     return curr;
27 }

```

Esempio 14.7: esecuzione delle implementazioni in C++ per $n = 4$

Questa è l'esecuzione passo per passo di `Fib(4)`. Le chiamate vengono eseguite nell'ordine della visita in profondità dell'albero dell'Esempio 14.2 (indentazione = profondità nello stack).

chiamata	azione
Fib(4)	richiama Fib(3)
Fib(3)	richiama Fib(2)
Fib(2)	richiama Fib(1)
Fib(1)	caso base, restituisce 1
Fib(2)	richiama Fib(0)
Fib(0)	caso base, restituisce 0
Fib(2)	restituisce $1 + 0 = 1$
Fib(3)	richiama Fib(1)
Fib(1)	caso base, restituisce 1
Fib(3)	restituisce $1 + 1 = 2$
Fib(4)	richiama Fib(2)
Fib(2)	richiama Fib(1)
Fib(1)	caso base, restituisce 1
Fib(2)	richiama Fib(0)
Fib(0)	caso base, restituisce 0
Fib(2)	restituisce $1 + 0 = 1$
Fib(4)	restituisce $2 + 1 = 3$

Il risultato è $F(4) = 3$, dopo $C(4) = 2F(5) - 1 = 9$ chiamate. `IterativeFib(4)`, invece, esegue 3 iterazioni:

iterazione	i	$next = prev + curr$	$prev$	$curr$
inizializzazione	2	–	0	1
1	$2 \rightarrow 3$	$0 + 1 = 1$	1	1
2	$3 \rightarrow 4$	$1 + 1 = 2$	1	2
3	$4 \rightarrow 5$	$1 + 2 = 3$	2	3

Il ciclo termina con $i = 5 > 4$ e la funzione restituisce $curr = 3$.

14.6 Massimo comun divisore

Dati due numeri naturali n, m con $n \geq 1$ e $m \geq 0$, consideriamo il problema di calcolare il loro massimo comun divisore $\gcd(n, m)$ (con $\gcd(n, 0) = n$).

- Nel capitolo «Algoritmi numerici» abbiamo dimostrato che, per tutti i numeri naturali $n \geq 1$ e $m \geq 1$,

$$\gcd(n, m) = \gcd(m, n \bmod m),$$

e abbiamo ottenuto l'algoritmo iterativo `EuclideanAlgorithm`: esso sostituisce la coppia (n, m) con la coppia $(m, n \bmod m)$ più e più volte, finché $m = 0$.

- La stessa proprietà, insieme a $\gcd(n, 0) = n$, è una definizione ricorsiva:

$$\gcd(n, m) = \begin{cases} n & \text{se } m = 0 \\ \gcd(m, n \bmod m) & \text{se } m \geq 1 \end{cases}$$

- Questo dà la versione ricorsiva dell'algoritmo di Euclide:

Algorithm: RecursiveGCD**Input** : Due numeri naturali $n \geq 1$ e $m \geq 0$ **Output**: $\text{gcd}(n, m)$

```

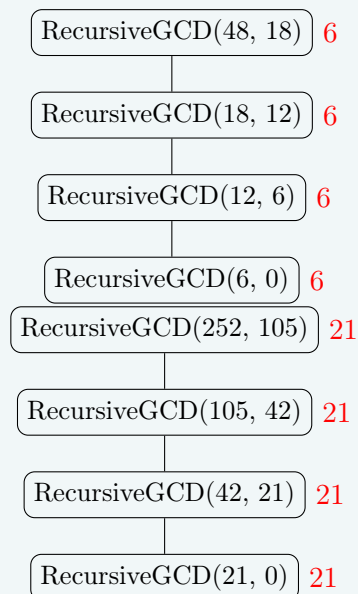
1 function RecursiveGCD( $n, m$ )
2   if  $m = 0$  then
3     return  $n$                                 // caso base
4   end
5   return RecursiveGCD( $m, n \bmod m$ )           // passo ricorsivo
6 end

```

- La dimensione dell'input è il secondo argomento m . Poiché $0 \leq n \bmod m < m$, il secondo argomento diminuisce strettamente a ogni chiamata ricorsiva, quindi il caso base $m = 0$ viene sempre raggiunto.

Esempio 14.8: Alberi delle chiamate ricorsive di RecursiveGCD

Ogni chiamata effettua al più una chiamata ricorsiva, quindi gli alberi sono cammini.



- $48 \bmod 18 = 12$, $18 \bmod 12 = 6$, $12 \bmod 6 = 0$: quindi $\text{gcd}(48, 18) = 6$.
- $252 \bmod 105 = 42$, $105 \bmod 42 = 21$, $42 \bmod 21 = 0$: quindi $\text{gcd}(252, 105) = 21$.
- Il valore trovato dal caso base viene restituito invariato da tutte le chiamate.

14.6.1 Correttezza

Osservazione 14.9: correttezza di RecursiveGCD

Per ogni coppia di numeri naturali $n \geq 1$ e $m \geq 0$, la chiamata `RecursiveGCD( $n, m$ )` termina e restituisce $\text{gcd}(n, m)$.

Dimostrazione. Procediamo per induzione forte su m .

- **Caso base** ($m = 0$): l'algoritmo restituisce $n = \text{gcd}(n, 0)$ alla riga 3.
- **Passo induttivo** ($m \geq 1$): supponiamo l'affermazione vera per ogni secondo argomento $k < m$ e per ogni primo argomento ≥ 1 . L'algoritmo richiama `RecursiveGCD( $m, n \bmod m$ )`, dove $m \geq 1$ e $n \bmod m < m$. Per ipotesi induttiva, questa chiamata restituisce $\text{gcd}(m, n \bmod m)$, che è uguale a $\text{gcd}(n, m)$ per la proprietà $\text{gcd}(n, m) = \text{gcd}(m, n \bmod m)$ richiamata all'inizio di questa sezione.

□

14.6.2 Ricorsione in coda

Definizione 14.10: ricorsione in coda

Una chiamata ricorsiva è una **chiamata in coda** (**tail call**) se è l'ultima operazione eseguita dalla funzione chiamante e il suo risultato viene restituito invariato. Un algoritmo ricorsivo le cui chiamate ricorsive sono tutte chiamate in coda è **ricorsivo in coda**.

- `RecursiveGCD` è ricorsivo in coda: la riga 5 restituisce direttamente il risultato della chiamata ricorsiva, senza alcuna ulteriore operazione.
- `Factorial` **non** è ricorsivo in coda: dopo che la chiamata ricorsiva `Factorial( $n - 1$ )` è terminata, il risultato deve ancora essere moltiplicato per n , quindi il frame del chiamante deve essere conservato nella pila.

Un algoritmo ricorsivo in coda può sempre essere trasformato in un ciclo **while**: la chiamata in coda viene sostituita dall'aggiornamento dei parametri, e il ciclo continua finché non si raggiunge il caso base. Applicando questa trasformazione a `RecursiveGCD`, con l'aggiornamento $(n, m) \leftarrow (m, n \bmod m)$, otteniamo esattamente l'`EuclideanAlgorithm` iterativo, che usa una quantità costante di memoria. Molti compilatori applicano automaticamente questa trasformazione (**tail-call optimization**).

14.6.3 Numero di passi

- Ogni chiamata esegue esattamente 2 righe (le righe 2–3, oppure le righe 2 e 5), quindi il numero di passi è il doppio del numero di chiamate. Limitiamo il numero di chiamate.
- Sia $m_0 = m$ e siano $m_0, m_1, m_2, \dots, m_K = 0$ i secondi argomenti delle chiamate successive. La chiamata $k \geq 1$ è `RecursiveGCD`(m_{k-1}, m_k) e quindi

$$m_{k+1} = m_{k-1} \bmod m_k, \quad m_{k+1} < m_k.$$

Osservazione 14.11: il secondo argomento si dimezza ogni due chiamate

Per ogni k tale che m_{k+2} sia definito,

$$m_{k+2} < \frac{m_k}{2}.$$

Dimostrazione. Abbiamo $m_{k+2} = m_k \bmod m_{k+1}$ con $m_{k+1} < m_k$. Distinguiamo due casi.

- Se $m_{k+1} \leq m_k/2$, allora $m_{k+2} < m_{k+1} \leq m_k/2$.
- Se $m_{k+1} > m_k/2$, allora $m_k < 2m_{k+1}$, quindi il quoziente della divisione di m_k per m_{k+1} è 1 e $m_{k+2} = m_k - m_{k+1} < m_k - m_k/2 = m_k/2$.

□

- Sia $m \geq 1$ e $j = \lfloor \log_2 m \rfloor + 1$, in modo che $2^j > m$. Applicando l'Osservazione j volte, se m_{2j} è definito allora

$$m_{2j} < \frac{m}{2^j} < 1 \implies m_{2j} = 0.$$

Quindi $K \leq 2j$ e il numero di chiamate $(K + 1)$ è al più $2\lfloor \log_2 m \rfloor + 3$. Perciò

$$S(n, m) \leq 2(2\lfloor \log_2 m \rfloor + 3) = O(\log_2 m).$$

- L'input è costituito dai due numeri n e m , quindi $|I| = \Omega(\log_2 m)$ (per $m \geq 2$) e, come per `ChangeToBase2` nel capitolo «Tempo di esecuzione», $S(n, m) = O(|I|)$.

L'algoritmo `RecursiveGCD` effettua al più $2\lfloor \log_2 m \rfloor + 3$ chiamate, e il suo tempo di esecuzione nel caso peggiore è

$$T(|I|) = O(|I|),$$

cioè è al più lineare nella dimensione dell'input. (Si può dimostrare che il caso peggiore si verifica per due numeri di Fibonacci consecutivi: per esempio $\text{gcd}(89, 55)$ richiede 10 chiamate, contro il limite $2\lfloor \log_2 55 \rfloor + 3 = 13$.)

14.6.4 Implementazione in C++

```

1 // Restituisce gcd(n, m), con n >= 1 e m >= 0
2 int RecursiveGCD(int n, int m)
3 {
4     if (m == 0)
5     {
6         return n;                // caso base: gcd(n, 0) = n
7     }
8     return RecursiveGCD(m, n % m); // chiamata in coda
9 }

```

Esempio 14.9: esecuzione dell'implementazione in C++ di RecursiveGCD

Questa è l'esecuzione passo per passo dell'implementazione in C++ per $n = 18$ e $m = 48$ (si noti che $n < m$: la prima chiamata scambia semplicemente i due numeri, poiché $18 \bmod 48 = 18$).

1. RecursiveGCD(18, 48): $m \neq 0$, $18 \% 48 = 18 \Rightarrow$ richiama RecursiveGCD(48, 18)
2. RecursiveGCD(48, 18): $m \neq 0$, $48 \% 18 = 12 \Rightarrow$ richiama RecursiveGCD(18, 12)
3. RecursiveGCD(18, 12): $m \neq 0$, $18 \% 12 = 6 \Rightarrow$ richiama RecursiveGCD(12, 6)
4. RecursiveGCD(12, 6): $m \neq 0$, $12 \% 6 = 0 \Rightarrow$ richiama RecursiveGCD(6, 0)
5. RecursiveGCD(6, 0): $m = 0 \Rightarrow$ caso base, restituisce 6

Il valore 6 viene restituito invariato da tutte le chiamate: $\text{gcd}(18, 48) = 6$, dopo 5 chiamate.

14.7 Elevamento a potenza veloce

Dati un numero x e un numero naturale n , consideriamo il problema di calcolare la potenza x^n (con $x^0 = 1$).

- L'algoritmo ingenuo moltiplica n volte per x :

$$p := 1, \quad p \leftarrow p \cdot x \quad (n \text{ volte}),$$

che richiede n moltiplicazioni e $\Theta(n)$ passi.

- Un algoritmo molto più veloce si ottiene mediante **elevamento al quadrato**. Poiché $n = 2\lfloor n/2 \rfloor$ se n è pari, e $n = 2\lfloor n/2 \rfloor + 1$ se n è dispari,

$$x^n = \begin{cases} 1 & \text{se } n = 0 \\ (x^{\lfloor n/2 \rfloor})^2 & \text{se } n \geq 1 \text{ è pari} \\ x \cdot (x^{\lfloor n/2 \rfloor})^2 & \text{se } n \text{ è dispari} \end{cases}$$

(per n dispari, $\lfloor n/2 \rfloor = (n - 1)/2$). La dimensione dell'esponente si dimezza a ogni chiamata ricorsiva.

Algorithm: FastPower**Input** : Un numero x e un numero naturale n **Output**: La potenza x^n

```

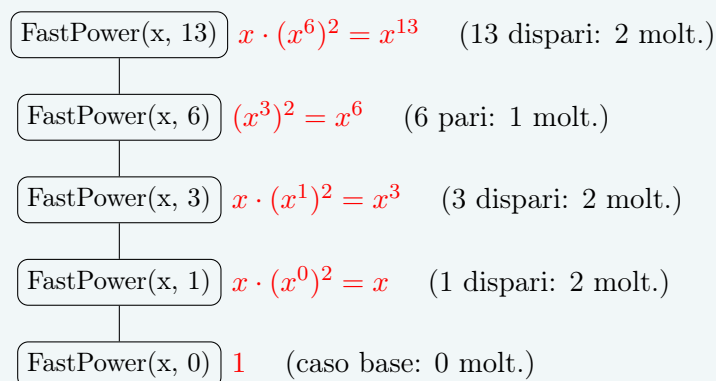
1 function FastPower( $x, n$ )
2   if  $n = 0$  then
3     return 1                                // caso base
4   end
5    $y := \text{FastPower}(x, \lfloor n/2 \rfloor)$           // una chiamata ricorsiva
6   if  $n \bmod 2 = 0$  then
7     return  $y \cdot y$                           //  $n$  pari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

```

- Si noti che il risultato della chiamata ricorsiva è memorizzato nella variabile y e usato due volte: scrivere $\text{FastPower}(x, \lfloor n/2 \rfloor) \cdot \text{FastPower}(x, \lfloor n/2 \rfloor)$ produrrebbe **due** chiamate ricorsive identiche e distruggerebbe l'efficienza dell'algoritmo (questa variante è analizzata negli esercizi del capitolo).

Esempio 14.10: Albero delle chiamate ricorsive di FastPower($x, 13$)

Gli esponenti delle chiamate sono 13, 6, 3, 1, 0 ($\lfloor 13/2 \rfloor = 6$, $\lfloor 6/2 \rfloor = 3$, $\lfloor 3/2 \rfloor = 1$, $\lfloor 1/2 \rfloor = 0$). A destra di ciascuna chiamata: il valore restituito e il numero di moltiplicazioni eseguite da quella chiamata.



- Numero totale di moltiplicazioni: $2 + 1 + 2 + 2 + 0 = 7$, contro le 13 moltiplicazioni dell'algoritmo ingenuo.
- Si noti che $13 = (1101)_2$: gli esponenti dispari/pari incontrati salendo da $\text{FastPower}(x, 1)$ a $\text{FastPower}(x, 13)$ (dispari, dispari, pari, dispari) sono esattamente le cifre binarie 1, 1, 0, 1 di 13 lette a partire dalla più significativa. In generale, il numero di moltiplicazioni è il numero di cifre binarie di n più il numero di cifre uguali a 1: qui $4 + 3 = 7$.

14.7.1 Correttezza

Osservazione 14.12: correttezza di FastPower

Per ogni numero x e ogni numero naturale n , la chiamata $\text{FastPower}(x, n)$ termina e restituisce x^n .

Dimostrazione. Procediamo per induzione forte su n .

- **Caso base** ($n = 0$): l'algoritmo restituisce $1 = x^0$ alla riga 3.
- **Passo induttivo** ($n \geq 1$): supponiamo che $\text{FastPower}(x, k)$ restituisca x^k per ogni $k < n$. Poiché $n \geq 1$, si ha $\lfloor n/2 \rfloor < n$, quindi per ipotesi induttiva la riga 5 assegna $y = x^{\lfloor n/2 \rfloor}$.
 - Se n è pari, allora $n = 2\lfloor n/2 \rfloor$ e la riga 7 restituisce $y \cdot y = x^{2\lfloor n/2 \rfloor} = x^n$.
 - Se n è dispari, allora $n = 2\lfloor n/2 \rfloor + 1$ e la riga 9 restituisce $x \cdot y \cdot y = x^{2\lfloor n/2 \rfloor + 1} = x^n$.

Si noti che serve l'induzione forte: l'ipotesi induttiva viene usata per $k = \lfloor n/2 \rfloor$, non per $k = n - 1$. □

14.7.2 Numero di passi

- Sia $S(n)$ il numero di passi di **FastPower** sull'esponente n . Se $n = 0$, vengono eseguite le righe 2–3: $S(0) = 2$. Se $n \geq 1$, le righe 2, 5, 6 e una delle righe 7, 9 vengono eseguite una volta ciascuna, più i passi della chiamata su $\lfloor n/2 \rfloor$:

$$S(n) = S(\lfloor n/2 \rfloor) + 4 \quad (n \geq 1).$$

- Risolviamo l'equazione di ricorrenza srotolandola. Poiché $\lfloor \lfloor n/2^k \rfloor / 2 \rfloor = \lfloor n/2^{k+1} \rfloor$,

$$\begin{aligned} S(n) &= S(\lfloor n/2 \rfloor) + 4 \\ &= S(\lfloor n/4 \rfloor) + 2 \cdot 4 \\ &\vdots \\ &= S(\lfloor n/2^k \rfloor) + k \cdot 4, \end{aligned}$$

fin tanto che $\lfloor n/2^{k-1} \rfloor \geq 1$. Ora, $\lfloor n/2^k \rfloor \geq 1$ se e solo se $2^k \leq n$, cioè $k \leq \lfloor \log_2 n \rfloor$. Quindi, per $n \geq 1$, gli esponenti $n, \lfloor n/2 \rfloor, \dots, \lfloor n/2^{\lfloor \log_2 n \rfloor} \rfloor = 1$ sono tutti ≥ 1 , e il successivo è 0. Scegliendo $k = \lfloor \log_2 n \rfloor + 1$:

$$S(n) = S(0) + 4(\lfloor \log_2 n \rfloor + 1) = 4\lfloor \log_2 n \rfloor + 6.$$

- Per esempio, $S(13) = 4 \cdot 3 + 6 = 18$. Come mostrato per **ChangeToBase2** nel capitolo «Tempo di esecuzione»,

$$\lim_{n \rightarrow \infty} \frac{4\lfloor \log_2 n \rfloor + 6}{\log_2 n} = 4 \in \mathbb{R}_{>0}, \quad \text{quindi} \quad S(n) = \Theta(\log_2 n).$$

- Per $n \geq 1$, il numero di chiamate, e quindi la profondità della ricorsione, è $\lfloor \log_2 n \rfloor + 2$ (il numero di cifre binarie di n , più la chiamata con esponente 0).
- Il numero di moltiplicazioni è al più $2(\lfloor \log_2 n \rfloor + 1)$, contro le n dell'algoritmo ingenuo. Per esempio, per $n = 1000 = (1111101000)_2$: $10 + 6 = 16$ moltiplicazioni invece di 1000.
- Supponiamo che x e tutti i risultati intermedi usino un numero costante di bit (così ogni moltiplicazione è un passo). La dimensione dell'input è $|I| = \Theta(\log_2 n)$, quindi, come per **ChangeToBase2** nel capitolo «Tempo di esecuzione», da $S(n) = \Theta(\log_2 n)$ segue $T(|I|) = \Theta(|I|)$.

Il numero di passi dell'algoritmo **FastPower** è $S(n) = 4\lfloor \log_2 n \rfloor + 6 = \Theta(\log_2 n)$, e il suo tempo di esecuzione nel caso peggiore è

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

cioè è lineare nella dimensione dell'input. L'algoritmo ingenuo esegue $\Theta(n)$ passi, cioè è esponenziale nella dimensione dell'input.

14.7.3 Implementazione in C++

```

1 // Restituisce x^n (n >= 0) per quadrati successivi
2 long long FastPower(long long x, int n)
3 {
4     if (n == 0)
5     {
6         return 1;                // caso base: x^0 = 1
7     }
8     long long y = FastPower(x, n / 2); // divisione intera = floor(n/2)
9     if (n % 2 == 0)
10    {
11        return y * y;              // n pari
12    }
13    else
14    {
15        return x * y * y;          // n dispari
16    }
17 }
```

Esempio 14.11: esecuzione dell'implementazione in C++ di FastPower

Questa è l'esecuzione passo per passo dell'implementazione in C++ per $x = 3$ e $n = 13$.

Chiamate

1. $\text{FastPower}(3, 13)$: $13 \neq 0 \Rightarrow$ richiama $\text{FastPower}(3, 6)$ $(13/2 = 6)$
2. $\text{FastPower}(3, 6)$: $6 \neq 0 \Rightarrow$ richiama $\text{FastPower}(3, 3)$ $(6/2 = 3)$
3. $\text{FastPower}(3, 3)$: $3 \neq 0 \Rightarrow$ richiama $\text{FastPower}(3, 1)$ $(3/2 = 1)$
4. $\text{FastPower}(3, 1)$: $1 \neq 0 \Rightarrow$ richiama $\text{FastPower}(3, 0)$ $(1/2 = 0)$

5. `FastPower(3, 0)`: caso base, restituisce 1

Ritorni

1. `FastPower(3, 1)`: $y = 1$, 1 è dispari $\Rightarrow$ restituisce $3 \cdot 1 \cdot 1 = 3$

2. `FastPower(3, 3)`: $y = 3$, 3 è dispari $\Rightarrow$ restituisce $3 \cdot 3 \cdot 3 = 27$

3. `FastPower(3, 6)`: $y = 27$, 6 è pari $\Rightarrow$ restituisce $27 \cdot 27 = 729$

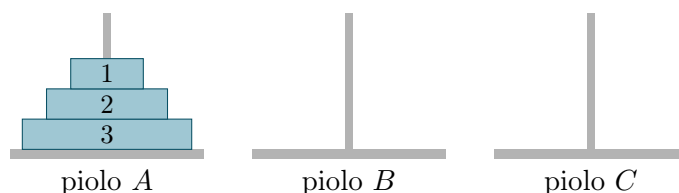
4. `FastPower(3, 13)`: $y = 729$, 13 è dispari $\Rightarrow$ restituisce $3 \cdot 729 \cdot 729 = 1594323$

L'algoritmo termina con risultato $1594323 = 3^{13}$, dopo 5 chiamate e 7 moltiplicazioni.

14.8 Torri di Hanoi

Ci sono tre pioli A , B , C e $n \geq 1$ dischi di dimensioni diverse, numerati $1, 2, \dots, n$ dal più piccolo al più grande. Inizialmente tutti i dischi sono impilati sul piolo A , in ordine decrescente di dimensione dal basso verso l'alto. Consideriamo il problema di spostare tutti i dischi sul piolo C , con le regole seguenti:

- si sposta un solo disco alla volta, prendendo il disco in cima a un piolo e posandolo in cima a un altro piolo;
- un disco non può mai essere posato sopra un disco più piccolo.



- **Idea ricorsiva.** Per spostare n dischi dal piolo A al piolo C usando il piolo B come piolo ausiliario:
 1. spostare gli $n - 1$ dischi più piccoli da A a B (usando C come piolo ausiliario), ricorsivamente;
 2. spostare il disco più grande n da A a C ;
 3. spostare gli $n - 1$ dischi più piccoli da B a C (usando A come piolo ausiliario), ricorsivamente.
- Durante i passi 1 e 3, il disco n si trova in fondo a un piolo ed è più grande di tutti i dischi spostati, quindi non viola mai le regole: gli $n - 1$ dischi possono essere spostati come se il disco n non ci fosse.

Algorithm: Hanoi

Input : Il numero di dischi $n \geq 1$, il piolo di partenza *from*, il piolo di arrivo *to*, il piolo ausiliario *aux*

Output: La sequenza di mosse che trasferisce i dischi $1, 2, \dots, n$ 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                                // caso base
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

```

- La chiamata iniziale è $\text{Hanoi}(n, A, C, B)$.

Osservazione 14.13: correttezza di Hanoi

Per ogni numero naturale $n \geq 1$, se i dischi $1, 2, \dots, n$ sono gli n dischi in cima al piolo *from* e tutti i dischi sui pioli *to* e *aux* sono più grandi di n , allora $\text{Hanoi}(n, \textit{from}, \textit{to}, \textit{aux})$ sposta i dischi $1, 2, \dots, n$ in cima al piolo *to*, rispettando le regole.

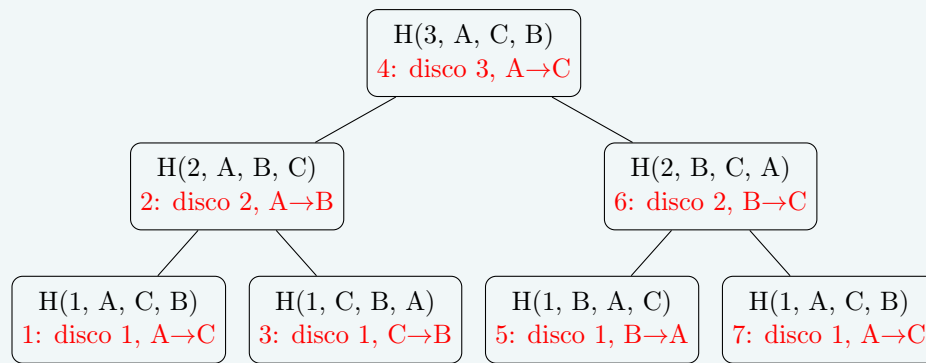
Dimostrazione. Procediamo per induzione su n .

- **Caso base** ($n = 1$): l'unica mossa del disco 1 è valida, poiché il disco 1 è il disco più piccolo.
- **Passo induttivo** ($n \geq 2$): per ipotesi induttiva, la chiamata della riga 5 sposta correttamente la torre degli $n - 1$ dischi più piccoli da *from* a *aux* (il disco n e i dischi sotto di esso sono più grandi, quindi non interferiscono). Allora il disco n è in cima al piolo *from*, e ogni disco sul piolo *to* è più grande di n : la riga 6 sposta correttamente il disco n . Infine, per ipotesi induttiva, la chiamata della riga 7 sposta gli $n - 1$ dischi più piccoli da *aux* in cima al disco n .

□

Esempio 14.12: Albero delle chiamate ricorsive di $\text{Hanoi}(3, A, C, B)$

Ogni nodo mostra la chiamata (con argomenti $n, \textit{from}, \textit{to}, \textit{aux}$) e, in rosso, la mossa eseguita dalla chiamata stessa (riga 3 o riga 6), numerata nell'ordine di esecuzione.



Le mosse vengono eseguite nell'ordine di una visita **simmetrica (in-order)** dell'albero (sottoalbero sinistro, nodo, sottoalbero destro):

mossa	disco	da	a
1	1	A	C
2	2	A	B
3	1	C	B
4	3	A	C
5	1	B	A
6	2	B	C
7	1	A	C

L'albero ha 7 nodi e ogni nodo esegue esattamente una mossa: 7 mosse in totale.

14.8.1 Numero di mosse

- Sia $M(n)$ il numero di mosse eseguite da **Hanoi** con n dischi. Dallo pseudocodice,

$$M(n) = \begin{cases} 1 & \text{se } n = 1 \\ 2M(n-1) + 1 & \text{se } n \geq 2 \end{cases}$$

- Srotolando: $M(n) = 2M(n-1) + 1 = 4M(n-2) + 2 + 1 = 8M(n-3) + 4 + 2 + 1 = \dots = 2^{n-1}M(1) + 2^{n-2} + \dots + 2 + 1 = 2^n - 1$. Lo dimostriamo formalmente:

Osservazione 14.14: numero di mosse di Hanoi

Per ogni numero naturale $n \geq 1$,

$$M(n) = 2^n - 1.$$

Dimostrazione. Procediamo per induzione su n .

- Caso base** ($n = 1$): $M(1) = 1 = 2^1 - 1$.
- Passo induttivo** ($n \geq 2$): supponiamo $M(n-1) = 2^{n-1} - 1$. Allora

$$M(n) = 2M(n-1) + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1.$$



- Il numero di passi soddisfa $S(1) = 2$ (righe 2-3) e $S(n) = 2S(n-1) + 4$ per $n \geq 2$ (righe 2, 5, 6, 7, più le due chiamate ricorsive). Srotolando come sopra,

$$S(n) = 2^{n-1}S(1) + 4(2^{n-2} + \dots + 2 + 1) = 2^n + 4(2^{n-1} - 1) = 3 \cdot 2^n - 4 = \Theta(2^n),$$

poiché $\lim_{n \rightarrow \infty} (3 \cdot 2^n - 4)/2^n = 3 \in \mathbb{R}_{>0}$.

- La profondità della ricorsione è solo n : la memoria usata dal call stack è $\Theta(n)$.

14.8.2 Ottimalità

- Il numero esponenziale di mosse non è una debolezza dell'algoritmo: **nessun** algoritmo può risolvere il problema con meno mosse.

Osservazione 14.15: ottimalità di Hanoi

Sia $M^*(n)$ il minimo numero di mosse necessarie per trasferire una torre di $n \geq 1$ dischi da un piolo a un altro. Allora

$$M^*(n) = 2^n - 1.$$

Traccia della dimostrazione. Poiché l'algoritmo **Hanoi** usa $2^n - 1$ mosse, $M^*(n) \leq 2^n - 1$. Dimostriamo $M^*(n) \geq 2^n - 1$ per induzione su n . Per $n = 1$ serve almeno una mossa.

Sia $n \geq 2$. Il disco più grande n deve essere spostato **almeno una volta**. Consideriamo la sua prima mossa, dal piolo di partenza a un altro piolo P : in quel momento il disco n è l'unico disco sul piolo di partenza e il piolo P è vuoto, quindi gli $n - 1$ dischi più piccoli sono tutti impilati sul terzo piolo. Spostarli lì dal piolo di partenza ha richiesto almeno $M^*(n - 1)$ mosse.

Consideriamo ora l'ultima mossa del disco n , verso il piolo di arrivo. Dopo questa mossa, gli $n - 1$ dischi più piccoli sono tutti impilati su un piolo diverso da quello di arrivo, e devono ancora essere trasferiti in cima al disco n : ciò richiede almeno altre $M^*(n - 1)$ mosse.

Questi due gruppi di mosse sono distinti (uno avviene prima della prima mossa del disco n , l'altro dopo la sua ultima mossa), quindi

$$M^*(n) \geq M^*(n - 1) + 1 + M^*(n - 1) = 2M^*(n - 1) + 1 \geq 2(2^{n-1} - 1) + 1 = 2^n - 1.$$



L'algoritmo **Hanoi** esegue $M(n) = 2^n - 1$ mosse e $S(n) = 3 \cdot 2^n - 4 = \Theta(2^n)$ passi, con profondità di ricorsione n . Il numero di mosse è **ottimo**: ogni algoritmo deve produrre in output $2^n - 1$ mosse, quindi il tempo di esecuzione di qualunque algoritmo per questo problema è esponenziale in n (e quindi doppiamente esponenziale nella dimensione dell'input).

$$|I| = \Theta(\log_2 n).$$

- Con $n = 64$ dischi (la leggenda originale) e una mossa al secondo, $2^{64} - 1$ secondi sono più di $5.8 \cdot 10^{11}$ anni.

14.8.3 Implementazione in C++

```

1 #include <iostream>
2 using namespace std;
3
4 // Stampa le mosse che spostano i dischi 1..n dal piolo from al piolo to
5 void Hanoi(int n, char from, char to, char aux)
6 {
7     if (n == 1)
8     {
9         cout << "sposta il disco 1 da " << from << " a " << to << endl;
10        return;
11    }
12    Hanoi(n - 1, from, aux, to);
13    cout << "sposta il disco " << n << " da " << from << " a " << to << endl;
14    Hanoi(n - 1, aux, to, from);
15 }
16
17 int main()
18 {
19     Hanoi(3, 'A', 'C', 'B');
20     return 0;
21 }

```

Esempio 14.13: esecuzione dell'implementazione in C++ di Hanoi per $n = 3$

Questa è l'esecuzione passo per passo per $n = 3$ (indentazione = profondità nello stack; le righe stampate sono le mosse dell'Esempio 14.12).

chiamata	azione
Hanoi(3,A,C,B)	richiama Hanoi(2,A,B,C)
Hanoi(2,A,B,C)	richiama Hanoi(1,A,C,B)
Hanoi(1,A,C,B)	stampa sposta il disco 1 da A a C , termina
Hanoi(2,A,B,C)	stampa sposta il disco 2 da A a B , richiama Hanoi(1,C,B,A)
Hanoi(1,C,B,A)	stampa sposta il disco 1 da C a B , termina
Hanoi(2,A,B,C)	termina
Hanoi(3,A,C,B)	stampa sposta il disco 3 da A a C , richiama Hanoi(2,B,C,A)
Hanoi(2,B,C,A)	richiama Hanoi(1,B,A,C)
Hanoi(1,B,A,C)	stampa sposta il disco 1 da B a A , termina
Hanoi(2,B,C,A)	stampa sposta il disco 2 da B a C , richiama Hanoi(1,A,C,B)
Hanoi(1,A,C,B)	stampa sposta il disco 1 da A a C , termina
Hanoi(2,B,C,A)	termina
Hanoi(3,A,C,B)	termina

Tutte le 7 mosse vengono stampate e tutti i dischi sono sul piolo C .

14.9 Riepilogo

algoritmo	equazione di ricorrenza per $S(n)$	$S(n)$	chiamate	profondità
Factorial	$S(n) = S(n-1) + 2$	$2n + 2 = \Theta(n)$	$n + 1$	$n + 1$
Fib (naive)	$S(n) = S(n-1) + S(n-2) + 2$	$4F(n+1) - 2 = 2^{\Theta(n)}$	$2F(n+1) - 1$	$\max(n, 1)$
RecursiveGCD	$m_{k+2} < m_k/2$ (Oss. 14.11)	$O(\log_2 m)$	$\leq 2\lfloor \log_2 m \rfloor + 3$	= chiamate
FastPower	$S(n) = S(\lfloor n/2 \rfloor) + 4$	$4\lfloor \log_2 n \rfloor + 6 = \Theta(\log_2 n)$	$\lfloor \log_2 n \rfloor + 2$	= chiamate
Hanoi	$S(n) = 2S(n-1) + 4$	$3 \cdot 2^n - 4 = \Theta(2^n)$	$2^n - 1$	n

- Un **algoritmo ricorsivo** risolve direttamente i **casi base** e riduce ogni altra istanza a istanze **strettamente più piccole** dello stesso problema; questo garantisce la terminazione.
- L'esecuzione è descritta dall'**albero delle chiamate ricorsive**: il numero di passi dipende dal **numero di nodi**, la memoria usata dal **call stack** dalla **profondità** dell'albero.
- La correttezza si dimostra per **induzione (forte)** sulla dimensione dell'input: casi base $\leftrightarrow$ base dell'induzione, passo ricorsivo $\leftrightarrow$ passo induttivo (come inizializzazione e conservazione per gli invarianti di ciclo).
- Il numero di passi è descritto da un'**equazione di ricorrenza** per $S(n)$: una diminuzione di uno con lavoro costante dà $\Theta(n)$ (**Factorial**); un dimezzamento con lavoro costante dà $\Theta(\log_2 n)$ (**FastPower**); due chiamate su $n-1$ danno $\Theta(2^n)$ (**Hanoi**).
- La ricorsione può essere molto inefficiente quando gli **stessi sottoproblemi** vengono risolti molte volte (**Fib ingenuo**): questo motiva la **Programmazione Dinamica**.
- Un algoritmo **ricorsivo in coda** (**RecursiveGCD**) può essere riscritto direttamente come un ciclo.

Equazioni di ricorrenza

Introduzione

Nel capitolo *Ricorsione* abbiamo contato i passi di un algoritmo ricorsivo riga per riga, e abbiamo osservato che il numero di passi eseguiti da una chiamata dipende dal numero di passi eseguiti dalle sue chiamate ricorsive, che sono chiamate dello *stesso* algoritmo su un input *più piccolo*. La quantità che vogliamo calcolare è quindi definita in termini di se stessa: questa è una **equazione di ricorrenza**.

In questo capitolo definiamo prima le equazioni di ricorrenza e mostriamo da dove nascono (Sezione 15.1). Poi studiamo tre metodi per risolverle, cioè per ottenerne la forma chiusa o almeno l'ordine di grandezza:

- il **metodo di espansione** (Sezione 15.2), che srotola l'equazione fino a raggiungere la condizione al contorno;
- il **metodo di sostituzione** (Sezione 15.3), che dimostra per induzione una limitazione ipotizzata;
- il **metodo dell'albero di ricorsione** (Sezione 15.4), che visualizza i costi di tutte le chiamate e li somma livello per livello.

Infine (Sezione 15.5) disegniamo l'albero di ricorsione di una generica equazione divide et impera: i tre comportamenti che vi osserviamo sono il punto di partenza del capitolo *Il Master theorem*.

15.1 Equazioni di ricorrenza

Definizione 15.1: equazione di ricorrenza

Una **equazione di ricorrenza** (o semplicemente **ricorrenza**) è un'equazione (o una disequazione) che descrive una funzione $T : \mathbb{N} \rightarrow \mathbb{R}$ in termini dei suoi valori su input più piccoli.

- Ogni ricorrenza ha delle **condizioni al contorno**, che danno il valore della funzione quando la dimensione dell'input è piccola, cioè quando n raggiunge (o è inferiore a) un valore specifico n_0 . Precisamente, le condizioni al contorno sono

$$T(n) = g(n) \quad \text{se } n \leq n_0,$$

dove $g(n)$ è il valore della funzione per i valori piccoli di n . Tipicamente $n_0 = 1$ e la condizione al contorno è $T(1) = \Theta(1)$.

- **Risolvere una ricorrenza** significa determinare una forma chiusa di $T(n)$ o, almeno, il suo **ordine di grandezza**, cioè una funzione $h(n)$ tale che $T(n) = \Theta(h(n))$ (oppure $T(n) = O(h(n))$, oppure $T(n) = \Omega(h(n))$).

Esempio 15.1: Una prima ricorrenza

Consideriamo la ricorrenza

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

La condizione al contorno è $T(1) = 1$ (qui $n_0 = 1$); per $n > 1$ ogni valore si ottiene dal precedente:

- $T(1) = 1$ (condizione al contorno);
- $T(2) = T(1) + 2 = 1 + 2 = 3$;
- $T(3) = T(2) + 3 = 3 + 3 = 6$;
- $T(4) = T(3) + 4 = 6 + 4 = 10$.

La ricorrenza definisce $T(n)$ per ogni $n \in \mathbb{N}$, $n \geq 1$, ma calcolare $T(n)$ in questo modo richiede $n - 1$ addizioni. Nell'Esempio 15.4 la risolviamo e otteniamo la forma chiusa $T(n) = \frac{n(n+1)}{2}$.

15.1.1 Ricorrenze e algoritmi ricorsivi

Le ricorrenze sono lo strumento naturale per determinare la complessità asintotica degli **algoritmi ricorsivi**.

In questo capitolo le ricorrenze sono oggetti matematici, e usiamo una generica funzione $T(n)$. Quando analizziamo un algoritmo ricorsivo, la funzione descritta dalla ricorrenza è il numero di passi $S(n)$ eseguiti nel caso peggiore su un input di parametro di dimensione n (si veda il capitolo *Ricorsione*): il numero di passi di una chiamata è il numero di passi eseguiti dalla chiamata stessa più il numero di passi eseguiti dalle sue chiamate ricorsive.

- Consideriamo il seguente algoritmo ricorsivo che calcola la somma degli elementi di un array.

Algorithm: RecursiveSum

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

Output: La somma $a[1] + a[2] + \dots + a[n]$

```

1 function RecursiveSum( $a, n$ )
2   if  $n = 1$  then
3     return  $a[1]$ 
4   end
5    $s := \text{RecursiveSum}(a, n - 1)$ 
6   return  $s + a[n]$ 
7 end
```

Esempio 15.2: La ricorrenza di RecursiveSum

Sia $S(n)$ il numero di passi eseguiti da **RecursiveSum** su un array di dimensione n . Contiamo le esecuzioni di ciascuna riga.

- Se $n = 1$, le righe 2 e 3 sono eseguite una volta: $S(1) = 2$.
- Se $n > 1$, le righe 2, 5 e 6 sono eseguite una volta (3 passi), e inoltre la chiamata ricorsiva alla riga 5 esegue $S(n - 1)$ passi.

Quindi

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

Per esempio, $S(2) = 2 + 3 = 5$, $S(3) = 5 + 3 = 8$ e $S(4) = 8 + 3 = 11$. Nell'Esempio 15.3 mostriamo che $S(n) = 3n - 1 = \Theta(n)$.

- Lo stesso algoritmo, scritto in C++, è la seguente funzione del codice del corso. In C++ l'array è indicizzato a partire da 0, quindi i suoi elementi sono $a[0], \dots, a[n-1]$, e l'ultimo elemento è $a[n-1]$.

```

1 // somma di a[0..n-1] per decremento di uno: profondità della ricorsione Theta(n)
2 long long sumRecDecrease(int *a, int n)
3 {
4     if (n == 0)
5     {
6         return 0;
7     }
8
9     return sumRecDecrease(a, n - 1) + a[n - 1];
10 }

```

- La funzione C++ si ferma sull'array **vuoto** ($n = 0$, somma 0) invece che a $n = 1$: accetta anche $n = 0$, ed esegue una chiamata in più rispetto allo pseudocodice. La ricorrenza ha la stessa forma $S(n) = S(n-1) + \text{costante}$, e l'ordine di grandezza non cambia. Sull'array 3, 1, 4, 1, 5, 9, 2, 6 la funzione restituisce 31.

15.1.2 Diverse forme di ricorrenze

La forma della ricorrenza rispecchia il modo in cui l'algoritmo suddivide il suo input. Le forme più comuni sono le seguenti.

- **Divide et impera.** Un algoritmo ricorsivo divide il problema in a sottoproblemi, ciascuno di dimensione n/b , li risolve ricorsivamente e ne combina le soluzioni. Ne risulta una ricorrenza della forma

$$T(n) = a T\left(\frac{n}{b}\right) + f(n).$$

Per esempio, l'algoritmo **MergeSort** (studiato nel capitolo *Algoritmi ricorsivi*) dà luogo a

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

- **Suddivisione non uniforme.** Un algoritmo ricorsivo può dividere il problema in sottoproblemi di dimensioni diverse. Per esempio, se l'input è diviso in una parte $1/3$ e una parte $2/3$, otteniamo la ricorrenza

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + f(n),$$

dove $f(n)$ è il tempo richiesto per dividere il problema e per combinare le soluzioni dei sottoproblemi.

- **Decrementa e conquista.** I sottoproblemi non sono necessariamente una frazione costante del problema originale. Per esempio, un algoritmo ricorsivo può creare un solo sottoproblema contenente un elemento in meno del problema originale (come **RecursiveSum**). Ne risulta la ricorrenza

$$T(n) = T(n-1) + f(n).$$

- **Più sottoproblemi di dimensione $n-1$.** Per esempio, l'algoritmo che risolve il rompicapo della *Torre di Hanoi* con n dischi sposta $n-1$ dischi, poi un disco, poi di nuovo $n-1$ dischi, dando luogo alla ricorrenza

$$T(n) = 2T(n-1) + 1,$$

la cui soluzione è esponenziale in n (si veda l'Esempio 15.5).

Definizione 15.2: ricorrenza in forma standard

Una ricorrenza è in **forma standard** se può essere scritta come

$$T(n) = a T\left(\frac{n}{b}\right) + f(n) \quad (15.1)$$

dove

- n è la dimensione dell'input (tipicamente $n \in \mathbb{N}$);
- $a \geq 1$ è il numero di sottoproblemi da risolvere;
- $\frac{n}{b}$ è la dimensione di ciascun sottoproblema ($b > 1$);
- $f(n)$ è il tempo richiesto per dividere il problema e per combinare le soluzioni dei sottoproblemi.

- Includendo le condizioni al contorno, una ricorrenza in forma standard è

$$T(n) = \begin{cases} g(1) & \text{se } n = 1 \\ a T\left(\frac{n}{b}\right) + f(n) & \text{se } n > 1 \end{cases}$$

dove $g(1)$ è il tempo richiesto per risolvere un sottoproblema di dimensione 1, tipicamente $g(1) = \Theta(1)$.

- Occasionalmente incontreremo ricorrenze che non sono uguaglianze ma disuguaglianze, come

$$T(n) \leq a T\left(\frac{n}{b}\right) + f(n).$$

Poiché una tale ricorrenza fornisce soltanto una **limitazione superiore** per $T(n)$, la sua soluzione è data in notazione O anziché in notazione Θ .

- Analogamente, se la disuguaglianza è invertita,

$$T(n) \geq a T\left(\frac{n}{b}\right) + f(n),$$

la ricorrenza fornisce soltanto una **limitazione inferiore** per $T(n)$, e la sua soluzione è data in notazione Ω .

15.1.3 Metodi per risolvere le ricorrenze

- In questo capitolo studiamo tre metodi per risolvere le ricorrenze:
 1. Il **metodo di espansione** (detto anche di iterazione o di srotolamento): si sostituisce ripetutamente la ricorrenza in se stessa fino a raggiungere la condizione al contorno, e si calcola la sommatoria che ne risulta.
 2. Il **metodo di sostituzione**: si ipotizza una limitazione e poi si usa l'induzione matematica per dimostrare che l'ipotesi è corretta.
 3. Il **metodo dell'albero di ricorsione**: si converte la ricorrenza in un albero i cui nodi rappresentano i costi sostenuti ai vari livelli della ricorsione, e si limitano le sommatorie che ne risultano.
- Nel capitolo *Il Master theorem* studiamo un quarto metodo, il **Master theorem**, che fornisce direttamente la soluzione delle ricorrenze in forma standard (15.1).

Prima di applicare i metodi fissiamo due punti tecnici: quali dettagli di una ricorrenza si possono trascurare, e quali sommatorie compaiono più spesso.

15.1.4 Aspetti tecnici nelle ricorrenze

- Spesso trascuriamo alcuni dettagli tecnici quando enunciamo e risolviamo le ricorrenze.
- **Parti intere inferiori e superiori.** Consideriamo la ricorrenza

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

Quando n è dispari, i due sottoproblemi non possono avere entrambi dimensione $n/2$ (che non è un numero intero): tipicamente, hanno dimensioni $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$. La ricorrenza precisa è quindi

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

Le due ricorrenze hanno la stessa soluzione asintotica. Poiché ci interessa l'ordine di grandezza, le parti intere inferiori e superiori si possono di solito omettere.

- **Potenze esatte.** Per lo stesso motivo, quando risolviamo una ricorrenza in forma standard (15.1) supponiamo spesso che n sia una **potenza esatta di** b , cioè $n = b^k$ per qualche $k \in \mathbb{N}$. In questo modo tutte le dimensioni dei sottoproblemi $n/b, n/b^2, \dots, 1$ sono intere. Nell'esempio precedente supponiamo $n = 2^k$.
- **Condizioni al contorno.** Le condizioni al contorno sono un'altra classe di dettagli che tipicamente ignoriamo. Poiché il tempo di esecuzione di un algoritmo su un input di dimensione costante è una costante, le ricorrenze che nascono dall'analisi degli algoritmi soddisfano in generale

$$T(n) = \Theta(1) \quad \text{per valori piccoli di } n.$$

Di conseguenza, per comodità, ometteremo in generale le condizioni al contorno e supporremo che $T(n)$ sia costante per n piccolo. Cambiare il valore di $T(1)$ cambia tipicamente la soluzione esatta, ma non il suo ordine di grandezza.

Omettere parti intere, condizioni al contorno e supporre che n sia una potenza esatta di b semplifica i calcoli. Quando è richiesta una dimostrazione rigorosa (per esempio nel metodo di sostituzione), questi dettagli devono essere trattati esplicitamente.

15.1.5 Sommatorie utili

Tutti e tre i metodi terminano con una sommatoria: i costi delle singole chiamate vanno sommati. Le formule seguenti sono usate in tutto il capitolo.

Osservazione 15.3: sommatorie e logaritmi utili

- La *serie aritmetica*: per ogni $n \geq 1$

$$\sum_{i=1}^n i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}. \quad (15.2)$$

- La *serie geometrica*: per ogni numero reale $x \neq 1$ e ogni $k \in \mathbb{N}$

$$\sum_{i=0}^k x^i = 1 + x + x^2 + \cdots + x^k = \frac{x^{k+1} - 1}{x - 1}. \quad (15.3)$$

In particolare, per $x = 2$,

$$\sum_{i=0}^{k-1} 2^i = 2^k - 1. \quad (15.4)$$

- La *serie geometrica infinita decrescente*: per ogni numero reale x con $|x| < 1$

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}. \quad (15.5)$$

- Due proprietà dei logaritmi: per ogni $a, b, n > 0$ (con $b \neq 1$)

$$a^{\log_b n} = n^{\log_b a}, \quad \log_b n = \frac{\log_2 n}{\log_2 b}. \quad (15.6)$$

Dimostrazione. Serie aritmetica. Scrivendo la somma due volte, una in ordine crescente e una in ordine decrescente, e sommando termine a termine, otteniamo n coppie, ciascuna con somma $n+1$: il doppio della somma è $n(n+1)$.

Serie geometrica. Moltiplicando la somma per $x-1$, tutti i termini intermedi si semplificano:

$$(x-1) \sum_{i=0}^k x^i = \sum_{i=1}^{k+1} x^i - \sum_{i=0}^k x^i = x^{k+1} - 1,$$

e dividiamo per $x-1 \neq 0$. Per $x=2$ e $k-1$ al posto di k otteniamo $2^k - 1$.

Serie infinita. Se $|x| < 1$, allora $x^{k+1} \rightarrow 0$ per $k \rightarrow \infty$, e la somma finita tende a $\frac{0-1}{x-1} = \frac{1}{1-x}$.

Logaritmi. $a^{\log_b n} = (b^{\log_b a})^{\log_b n} = (b^{\log_b n})^{\log_b a} = n^{\log_b a}$. Inoltre $n = b^{\log_b n}$ dà $\log_2 n = \log_b n \cdot \log_2 b$. $\square$

- Poiché tutti i termini sono positivi quando $0 < x < 1$, ogni somma finita $\sum_{i=0}^k x^i$ è minore di $\frac{1}{1-x}$: la serie infinita dà una **limitazione superiore** per una somma geometrica finita che non dipende da k .
- La seconda proprietà dei logaritmi mostra che logaritmi in basi diverse differiscono per un fattore costante; quindi, dentro Θ , O e Ω la base del logaritmo è irrilevante.

15.2 Il metodo di espansione

Il modo più diretto per risolvere una ricorrenza è applicarla ripetutamente: ogni applicazione sostituisce un valore di T con un valore su un argomento più piccolo, e dopo abbastanza applicazioni resta soltanto la condizione al contorno.

Nel **metodo di espansione** (detto anche **iterazione** o **srotolamento**):

1. si sostituisce la ricorrenza in se stessa alcune volte, fino a riconoscere lo schema generale dopo k sostituzioni;
2. si sceglie k in modo che l'argomento raggiunga la condizione al contorno;
3. si calcola (o si limita) la sommatoria che ne risulta.

- Il metodo fornisce una forma chiusa esatta quando la sommatoria si può calcolare. Lo schema riconosciuto al passo 1 si può dimostrare rigorosamente per induzione su k .

Cominciamo con le ricorrenze di tipo decrementa e conquista, $T(n-1) + f(n)$, e poi passiamo a quelle di tipo divide et impera, $aT(n/2) + f(n)$.

Esempio 15.3: $T(n) = T(n-1) + c$

Siano $c, d > 0$ costanti e

$$T(n) = \begin{cases} d & \text{se } n = 1 \\ T(n-1) + c & \text{se } n > 1 \end{cases}$$

- *Passo 1 (espansione).*

$$\begin{aligned} T(n) &= T(n-1) + c \\ &= (T(n-2) + c) + c = T(n-2) + 2c \\ &= (T(n-3) + c) + 2c = T(n-3) + 3c \\ &\vdots \\ &= T(n-k) + kc. \end{aligned}$$

- *Passo 2 (condizione al contorno).* La condizione al contorno è raggiunta quando $n-k=1$, cioè $k=n-1$.

- *Passo 3 (somma).*

$$T(n) = T(1) + (n-1)c = cn + (d-c) = \Theta(n).$$

Per `RecursiveSum` (Esempio 15.2) abbiamo $c = 3$ e $d = 2$, quindi

$$S(n) = 3(n-1) + 2 = 3n - 1 = \Theta(n),$$

che dà $S(2) = 5$, $S(3) = 8$, $S(4) = 11$, come calcolato direttamente dalla ricorrenza.

Esempio 15.4: $T(n) = T(n-1) + n$

Sia

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

- *Passo 1 (espansione).*

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= T(n-3) + (n-2) + (n-1) + n \\ &\vdots \\ &= T(n-k) + (n-k+1) + \cdots + (n-1) + n. \end{aligned}$$

- *Passo 2 (condizione al contorno).* $n-k=1$, cioè $k=n-1$.
- *Passo 3 (somma).* Usando la serie aritmetica (15.2),

$$T(n) = T(1) + 2 + 3 + \cdots + n = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} = \Theta(n^2).$$

Infatti,

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

La forma chiusa dà $T(1) = 1$, $T(2) = 3$, $T(3) = 6$, $T(4) = 10$, i valori dell'Esempio 15.1.

Esempio 15.5: $T(n) = 2T(n-1) + 1$: la Torre di Hanoi

Sia

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

- *Passo 1 (espansione).*

$$T(n) = 2T(n-1) + 1$$

$$\begin{aligned}
&= 2(2T(n-2) + 1) + 1 = 2^2 T(n-2) + 2 + 1 \\
&= 2^2(2T(n-3) + 1) + 2 + 1 = 2^3 T(n-3) + 2^2 + 2 + 1 \\
&\vdots \\
&= 2^k T(n-k) + \sum_{i=0}^{k-1} 2^i.
\end{aligned}$$

- *Passo 2 (condizione al contorno).* $n - k = 1$, cioè $k = n - 1$.
- *Passo 3 (somma).* Usando (15.4),

$$T(n) = 2^{n-1} T(1) + (2^{n-1} - 1) = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1 = \Theta(2^n).$$

Questo è il numero di mosse necessarie per risolvere il rompicapo della Torre di Hanoi con n dischi: il tempo di esecuzione è **esponenziale** in n . La funzione `hanoi` del codice del corso conta le sue mosse; eseguendola otteniamo

n	1	2	3	4	5	6	7	8	10
mosse	1	3	7	15	31	63	127	255	1023

cioè esattamente $2^n - 1$ mosse.

Le due ricorrenze seguenti dimezzano la dimensione del problema a ogni chiamata. Per evitare le parti intere supponiamo che n sia una potenza esatta di 2.

Esempio 15.6: $T(n) = T(n/2) + c$

Siano $c, d > 0$ costanti, sia $n = 2^k$ una potenza esatta di 2, e

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

- *Passo 1 (espansione).*

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

- *Passo 2 (condizione al contorno).* $\frac{n}{2^j} = 1$, cioè $j = \log_2 n$.
- *Passo 3 (somma).*

$$T(n) = T(1) + c \log_2 n = c \log_2 n + d = \Theta(\log_2 n).$$

Questa ricorrenza descrive, per esempio, la versione ricorsiva di `BinarySearch`: a ogni chiamata la dimensione del sotto-array è dimezzata.

Esempio 15.7: $T(n) = 2T(n/2) + n$

Sia $n = 2^k$ una potenza esatta di 2 e

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

- *Passo 1 (espansione).*

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n \\ &= 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n = 2^2 T\left(\frac{n}{2^2}\right) + 2n \\ &= 2^2\left(2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right) + 2n = 2^3 T\left(\frac{n}{2^3}\right) + 3n \\ &\vdots \\ &= 2^j T\left(\frac{n}{2^j}\right) + jn. \end{aligned}$$

- *Passo 2 (condizione al contorno).* $j = \log_2 n$, in modo che $2^j = n$.
- *Passo 3 (somma).*

$$T(n) = nT(1) + n \log_2 n = n \log_2 n + n = \Theta(n \log_2 n).$$

Verifica: dalla ricorrenza $T(2) = 2 + 2 = 4$, $T(4) = 8 + 4 = 12$, $T(8) = 24 + 8 = 32$, e dalla forma chiusa $T(8) = 8 \cdot 3 + 8 = 32$.

Il metodo di espansione funziona bene quando la ricorrenza ha un **solo termine ricorsivo**. Con più termini ricorsivi di dimensioni diverse (per esempio $T(n/3) + T(2n/3)$) l'espansione diventa rapidamente ingestibile: in questi casi il metodo dell'albero di ricorsione (Sezione 15.4) è più conveniente.

15.3 Il metodo di sostituzione

Il metodo di espansione richiede dimensioni esatte dei sottoproblemi: con le parti intere lo schema non è più pulito. Quando abbiamo già un'idea della soluzione, un approccio più robusto è *dimostrarla* per induzione.

Il **metodo di sostituzione** comprende due passi:

1. **ipotizzare** la forma della soluzione (per esempio $T(n) \leq cn \log_2 n$, con c una costante incognita);
2. usare l'**induzione matematica** per trovare le costanti e mostrare che la soluzione funziona.

- Il nome deriva dal fatto che, nel passo induttivo, *sostituiamo* la limitazione ipotizzata ai valori della funzione sugli input più piccoli.

- Il metodo è molto potente, ma richiede una buona ipotesi. L'ipotesi può venire dall'esperienza (ricorrenze simili), dal metodo di espansione o da un albero di ricorsione.
- Possiamo usare il metodo di sostituzione per dimostrare separatamente limitazioni superiori (O) e inferiori (Ω). Insieme danno Θ .
- Poiché la notazione asintotica richiede la limitazione solo per $n \geq n_0$, siamo liberi di scegliere n_0 : questo è utile quando la limitazione non vale per i valori più piccoli di n .

15.3.1 Un esempio con le parti intere: limitazione superiore

- Consideriamo la ricorrenza

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ 2T(\lfloor n/2 \rfloor) + n & \text{se } n > 1 \end{cases} \quad (15.7)$$

- Dall'Esempio 15.7 (dove n è una potenza di 2) ipotizziamo $T(n) = O(n \log_2 n)$. Vogliamo dimostrare che esistono costanti $c > 0$ e n_0 tali che

$$T(n) \leq cn \log_2 n \quad \text{per ogni } n \geq n_0.$$

Passo induttivo. Supponiamo che la limitazione valga per tutti i valori m con $n_0 \leq m < n$, in particolare per $m = \lfloor n/2 \rfloor$ (questo richiede $\lfloor n/2 \rfloor \geq n_0$):

$$T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor.$$

Sostituendo nella ricorrenza:

$$\begin{aligned} T(n) &\leq 2c \lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor + n \\ &\leq 2c \frac{n}{2} \log_2 \frac{n}{2} + n && (\text{poiché } \lfloor n/2 \rfloor \leq n/2) \\ &= cn \log_2 n - cn \log_2 2 + n \\ &= cn \log_2 n - cn + n \\ &\leq cn \log_2 n, \end{aligned}$$

dove l'ultimo passaggio vale purché $c \geq 1$.

Casi base. L'induzione richiede che la limitazione valga per le condizioni al contorno. Tuttavia, per $n = 1$:

$$T(1) = 1 > c \cdot 1 \cdot \log_2 1 = 0,$$

per ogni c . Quindi la limitazione *non vale* per $n = 1$. Sfruttiamo la libertà di scegliere n_0 : 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 soltanto valori di T su argomenti ≥ 2 .
- Per $n = 2$ e $n = 3$ si ha $\lfloor n/2 \rfloor = 1$, quindi il passo induttivo userebbe $T(1)$, che non soddisfa la limitazione. Pertanto $n = 2$ e $n = 3$ devono essere trattati come **casi base**.

Dalla ricorrenza, $T(2) = 2T(1) + 2 = 4$ e $T(3) = 2T(1) + 3 = 5$. Ci serve

$$4 \leq c \cdot 2 \log_2 2 = 2c \quad \text{e} \quad 5 \leq c \cdot 3 \log_2 3.$$

Entrambe valgono per $c \geq 2$ (poiché $3 \log_2 3 \approx 4.75$, si ha $2 \cdot 3 \log_2 3 \approx 9.51 \geq 5$).

Osservazione 15.4: limitazione superiore per $2T(\lfloor n/2 \rfloor) + n$

La funzione $T(n)$ definita in (15.7) soddisfa $T(n) \leq 2n \log_2 n$ per ogni $n \geq 2$. Quindi $T(n) = O(n \log_2 n)$.

Dimostrazione. Sia $c = 2$. Procediamo per induzione forte su $n \geq 2$.

Casi base $n = 2, 3$: $T(2) = 4 \leq 2 \cdot 2 \cdot \log_2 2 = 4$ e $T(3) = 5 \leq 2 \cdot 3 \log_2 3 \approx 9.51$.

Passo induttivo $n \geq 4$: abbiamo $2 \leq \lfloor n/2 \rfloor < n$, quindi per l'ipotesi induttiva $T(\lfloor n/2 \rfloor) \leq 2\lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor$. Allora

$$T(n) \leq 4\lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor + n \leq 2n \log_2 \frac{n}{2} + n = 2n \log_2 n - 2n + n \leq 2n \log_2 n. \quad \square$$

15.3.2 Un esempio con le parti intere: limitazione inferiore

- Dimostriamo ora $T(n) = \Omega(n \log_2 n)$ per la stessa ricorrenza (15.7). Cerchiamo una costante $d > 0$ tale che $T(n) \geq dn \log_2 n$ per ogni $n \geq 1$.
- Per la limitazione inferiore, la parte intera lavora contro di noi: servono limitazioni inferiori per $\lfloor n/2 \rfloor$. Per ogni $n \geq 2$:

$$2\lfloor n/2 \rfloor \geq n - 1 \quad \text{e} \quad \lfloor n/2 \rfloor \geq \frac{n}{3}.$$

La prima vale perché $\lfloor n/2 \rfloor \geq (n-1)/2$. La seconda vale per $n = 2$ ($1 \geq 2/3$) e, per $n \geq 3$, perché $(n-1)/2 \geq n/3$ è equivalente a $n \geq 3$.

Passo induttivo ($n \geq 2$). Supponiamo $T(\lfloor n/2 \rfloor) \geq d\lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor$. Poiché $\lfloor n/2 \rfloor \geq 1$, il fattore $\log_2 \lfloor n/2 \rfloor$ è non negativo, e

$$\begin{aligned} T(n) &\geq 2d\lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor + n \\ &\geq d(n-1) \log_2 \frac{n}{3} + n \\ &= dn \log_2 n - dn \log_2 3 - d \log_2 n + d \log_2 3 + n \\ &\geq dn \log_2 n + (n - dn \log_2 3 - d \log_2 n). \end{aligned}$$

Poiché $\log_2 n \leq n$, il termine fra parentesi è almeno $n(1 - d \log_2 3 - d)$, che è non negativo se

$$d \leq \frac{1}{1 + \log_2 3} \approx 0.387.$$

Scegliamo $d = \frac{1}{4}$.

Caso base ($n = 1$): $T(1) = 1 \geq \frac{1}{4} \cdot 1 \cdot \log_2 1 = 0$. ✓

Osservazione 15.5: limitazione inferiore per $2T(\lfloor n/2 \rfloor) + n$

La funzione $T(n)$ definita in (15.7) soddisfa $T(n) \geq \frac{1}{4}n \log_2 n$ per ogni $n \geq 1$. Quindi $T(n) = \Omega(n \log_2 n)$.

Dimostrazione. Per induzione forte su $n \geq 1$, con $d = \frac{1}{4}$. Per $n = 1$, $T(1) = 1 \geq 0$. Per $n \geq 2$ abbiamo $1 \leq \lfloor n/2 \rfloor < n$, quindi l'ipotesi induttiva vale per $\lfloor n/2 \rfloor$, e il calcolo del passo induttivo precedente dà

$$T(n) \geq \frac{1}{4} n \log_2 n + n \left(1 - \frac{1}{4} \log_2 3 - \frac{1}{4}\right) \geq \frac{1}{4} n \log_2 n,$$

poiché $1 - \frac{1}{4}(1 + \log_2 3) \approx 0.35 > 0$. □

Dalle Osservazioni 15.4 e 15.5, la soluzione di (15.7) è

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

La parte intera non cambia l'ordine di grandezza trovato per n potenza di 2. Numericamente, per ogni n con $2 \leq n < 200\,000$ il rapporto $\frac{T(n)}{n \log_2 n}$ è compreso tra 0.83 e 2.

15.3.3 Quando l'ipotesi va rafforzata

- A volte l'ipotesi è corretta, ma l'induzione non funziona perché l'ipotesi induttiva *non è abbastanza forte*. Consideriamo

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ 2T(\lfloor n/2 \rfloor) + 1 & \text{se } n > 1 \end{cases} \quad (15.8)$$

- Per n potenza di 2 si trova facilmente $T(n) = 2n - 1$ (per esempio $T(2) = 3$, $T(4) = 7$, $T(8) = 15$), quindi ipotizziamo $T(n) = O(n)$, cioè $T(n) \leq cn$.
- **Tentativo ingenuo.** Sostituendo l'ipotesi,

$$T(n) \leq 2c \lfloor n/2 \rfloor + 1 \leq cn + 1,$$

che **non** implica $T(n) \leq cn$ per nessuna scelta di c . L'induzione fallisce, per un termine di ordine inferiore (la costante 1).

- **Ipotesi rafforzata.** È allettante provare un'ipotesi più grande, come cn^2 , ma non serve. Invece *sottraiamo* un termine di ordine inferiore: ipotizziamo

$$T(n) \leq cn - d,$$

con $d \geq 0$ una costante. Allora

$$T(n) \leq 2(c \lfloor n/2 \rfloor - d) + 1 \leq cn - 2d + 1 \leq cn - d,$$

dove l'ultimo passaggio vale purché $d \geq 1$.

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

Osservazione 15.6: ipotesi rafforzata per $2T(\lfloor n/2 \rfloor) + 1$

La funzione $T(n)$ definita in (15.8) soddisfa $n \leq T(n) \leq 2n - 1$ per ogni $n \geq 1$. Quindi $T(n) = \Theta(n)$.

Dimostrazione. Per induzione forte su n . Per $n = 1$: $1 \leq T(1) = 1 \leq 2 \cdot 1 - 1$. Per $n \geq 2$, abbiamo $1 \leq \lfloor n/2 \rfloor < n$ e, per l'ipotesi induttiva, $\lfloor n/2 \rfloor \leq T(\lfloor n/2 \rfloor) \leq 2\lfloor n/2 \rfloor - 1$.

Limitazione superiore: $T(n) \leq 2(2\lfloor n/2 \rfloor - 1) + 1 = 4\lfloor n/2 \rfloor - 1 \leq 2n - 1$.

Limitazione inferiore: $T(n) \geq 2\lfloor n/2 \rfloor + 1 \geq (n - 1) + 1 = n$. □

Quando l'induzione fallisce soltanto a causa di un **termine di ordine inferiore**, non aumentare l'ipotesi: **sottrai un termine di ordine inferiore** da essa (per esempio $cn - d$ invece di cn , oppure $cn^2 - dn$ invece di cn^2). L'ipotesi più forte fa funzionare l'induzione.

- **Evitare le trappole.** Nel tentativo ingenuo sarebbe **sbagliato** concludere “ $T(n) \leq cn + 1 = O(n)$, fatto”. L'induzione deve dimostrare *esattamente* la forma dell'ipotesi induttiva, $T(n) \leq cn$, con la *stessa* costante c ; altrimenti la costante crescerebbe a ogni passo della ricorsione.

15.3.4 Cambio di variabile

A volte una manipolazione algebrica, come un cambio di variabile, trasforma una ricorrenza sconosciuta in una nota, la cui soluzione può allora essere riutilizzata.

Esempio 15.8: $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log_2 n$

Consideriamo

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log_2 n,$$

con condizione al contorno $T(2) = 1$.

- *Passo 1 (ridenominazione).* Ignorando l'arrotondamento (cioè supponendo che $n = 2^{2^k}$, in modo che tutte le radici quadrate siano potenze esatte di 2), poniamo $m = \log_2 n$, cioè $n = 2^m$ e $\sqrt{n} = 2^{m/2}$:

$$T(2^m) = 2T(2^{m/2}) + m.$$

- *Passo 2 (nuova funzione).* Definiamo $R(m) = T(2^m)$ e otteniamo

$$R(m) = 2R\left(\frac{m}{2}\right) + m,$$

che è la ricorrenza dell'Esempio 15.7. Poiché $T(2) = 1$, si ha $R(1) = 1$, e quindi

$$R(m) = m \log_2 m + m.$$

- *Passo 3 (ritorno a T).* Con $m = \log_2 n$:

$$T(n) = R(\log_2 n) = \log_2 n \cdot \log_2 \log_2 n + \log_2 n = \Theta(\log_2 n \cdot \log_2 \log_2 n).$$

Verifica: $T(4) = 2T(2) + 2 = 4$ e $R(2) = 2 \cdot 1 + 2 = 4$; $T(16) = 2T(4) + 4 = 12$ e $R(4) = 4 \cdot 2 + 4 = 12$.

15.4 Il metodo dell'albero di ricorsione

Il metodo di sostituzione dimostra un'ipotesi, ma non la produce. L'albero di ricorsione è lo strumento che produce buone ipotesi, e in molti casi fornisce direttamente la soluzione.

- In un **albero di ricorsione**, ogni nodo rappresenta il costo di un singolo sottoproblema da qualche parte nell'insieme delle invocazioni ricorsive della funzione. I figli di un nodo sono i sottoproblemi da esso generati.
- Può essere usato per risolvere le ricorrenze, cioè per determinarne la complessità asintotica, oppure per produrre una buona ipotesi da verificare con il metodo di sostituzione.
- **Idea principale:** sommiamo i costi all'interno di ciascun livello dell'albero per ottenere un insieme di costi per livello, e poi sommiamo tutti i costi per livello per determinare il costo totale di tutti i livelli della ricorsione.

Nel **metodo dell'albero di ricorsione** determiniamo:

1. la dimensione dei sottoproblemi e il numero di nodi a ciascuna profondità i ;
2. il costo di ciascun livello (numero di nodi $\times$ costo di ciascun nodo);
3. la profondità dell'albero (quando la dimensione del sottoproblema raggiunge la condizione al contorno) e il costo delle foglie;
4. la somma dei costi di tutti i livelli.

Applichiamo il metodo a quattro ricorrenze, che mostrano i tre possibili comportamenti dei costi per livello (decrescente, costante, crescente) e il caso di un albero sbilanciato.

15.4.1 Costi decrescenti per livello: $T(n) = 3T(n/4) + cn^2$

- Consideriamo la ricorrenza ($c > 0$ una costante)

$$T(n) = 3T\left(\frac{n}{4}\right) + cn^2.$$

Per comodità supponiamo che n sia una potenza esatta di 4, in modo che tutte le dimensioni dei sottoproblemi siano intere.

- Il termine cn^2 alla radice rappresenta il costo al livello più alto della ricorsione (divisione e combinazione).
- I tre sottoalberi della radice rappresentano i costi sostenuti dai sottoproblemi di dimensione $\frac{n}{4}$; alla profondità successiva la dimensione diventa $\frac{n}{16}$, e così via.
- Le foglie rappresentano i punti in cui la ricorsione “tocca il fondo”, cioè in cui si raggiunge il *caso base* della ricorsione.

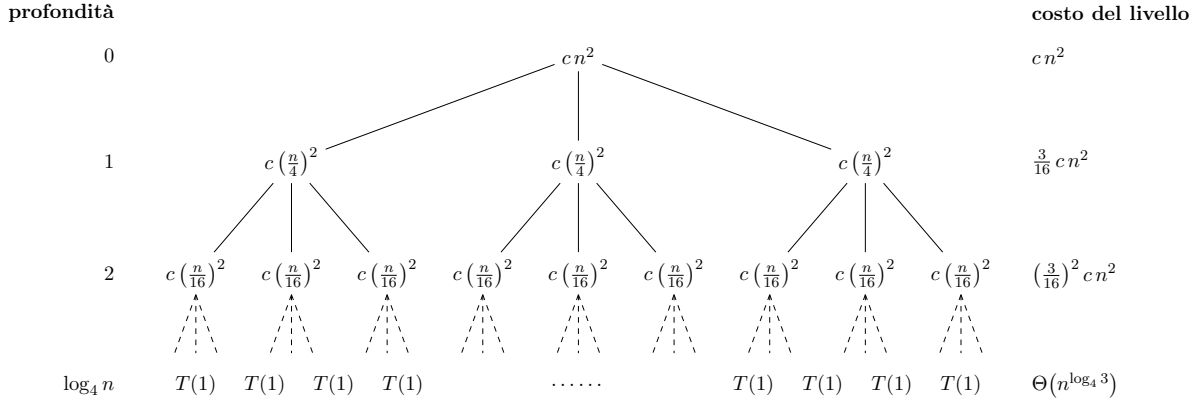


Figura 15.1: Albero di ricorsione di $T(n) = 3T(n/4) + cn^2$: ogni nodo ha tre figli, la dimensione del sottoproblema è divisa per 4 a ogni livello, e le foglie si trovano a profondità $\log_4 n$.

Esempio 15.9: Risolvere $T(n) = 3T(n/4) + cn^2$ con l'albero di ricorsione

- *Passo 1 (dimensioni e nodi).* La dimensione del sottoproblema di ciascun nodo a profondità i è $\frac{n}{4^i}$. Ogni livello ha 3 volte più nodi del livello precedente, quindi il numero di nodi a profondità i è 3^i .
- *Passo 2 (costo di un livello).* Ogni nodo a profondità i ($i = 0, 1, 2, \dots, \log_4 n - 1$) ha un costo di $c \left(\frac{n}{4^i}\right)^2$. Moltiplicando, il costo complessivo di tutti i nodi a profondità i è

$$3^i c \left(\frac{n}{4^i}\right)^2 = \left(\frac{3}{16}\right)^i cn^2.$$

- *Passo 3 (profondità e foglie).* La dimensione del sottoproblema raggiunge 1 quando $\frac{n}{4^i} = 1$, cioè quando $i = \log_4 n$. Di conseguenza, l'albero ha $\log_4 n + 1$ livelli (alle profondità $0, 1, 2, \dots, \log_4 n$). Il livello più basso ha, usando (15.6),

$$3^{\log_4 n} = n^{\log_4 3}$$

nodi (le foglie dell'albero), ciascuno con un costo costante $T(1)$. Il costo totale del livello più basso è

$$n^{\log_4 3} T(1) = \Theta(n^{\log_4 3}).$$

- *Passo 4 (somma).* Sommando i costi di tutti i livelli otteniamo

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16} cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \dots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\ &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}). \end{aligned}$$

Usando la serie geometrica (15.3) otteniamo il valore esatto

$$T(n) = \frac{16}{13} \left(1 - \left(\frac{3}{16}\right)^{\log_4 n}\right) cn^2 + \Theta(n^{\log_4 3}).$$

Limitazione superiore. È più semplice limitare la somma finita con la serie geometrica infinita decrescente (15.5):

$$\begin{aligned} T(n) &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i c n^2 + \Theta\left(n^{\log_4 3}\right) = \frac{1}{1 - \frac{3}{16}} c n^2 + \Theta\left(n^{\log_4 3}\right) \\ &= \frac{16}{13} c n^2 + \Theta\left(n^{\log_4 3}\right) = O(n^2), \end{aligned}$$

poiché $\log_4 3 \approx 0.79 < 2$, quindi $n^{\log_4 3} = O(n^2)$.

Limitazione inferiore. Tutti i termini della somma sono non negativi, e la sola radice costa $c n^2$. Quindi

$$T(n) \geq c n^2 = \Omega(n^2).$$

Verifica numerica ($c = 1$, $T(1) = 1$): la ricorrenza dà $T(4) = 19$, $T(16) = 313$, $T(64) = 5035$, i valori della formula esatta qui sopra, e $\frac{T(n)}{n^2} = 1.1875, 1.2227, 1.2292$, che si avvicinano a $\frac{16}{13} \approx 1.2308$.

La soluzione di $T(n) = 3T(n/4) + c n^2$ è $T(n) = \Theta(n^2)$. I costi dei livelli decrescono geometricamente: il costo totale è dominato dal costo della **radice**.

15.4.2 Costi costanti per livello: $T(n) = 2T(n/2) + c n$

- Consideriamo la ricorrenza ($c > 0$ una costante), che descrive algoritmi come MergeSort:

$$T(n) = 2T\left(\frac{n}{2}\right) + c n,$$

dove n è una potenza esatta di 2.

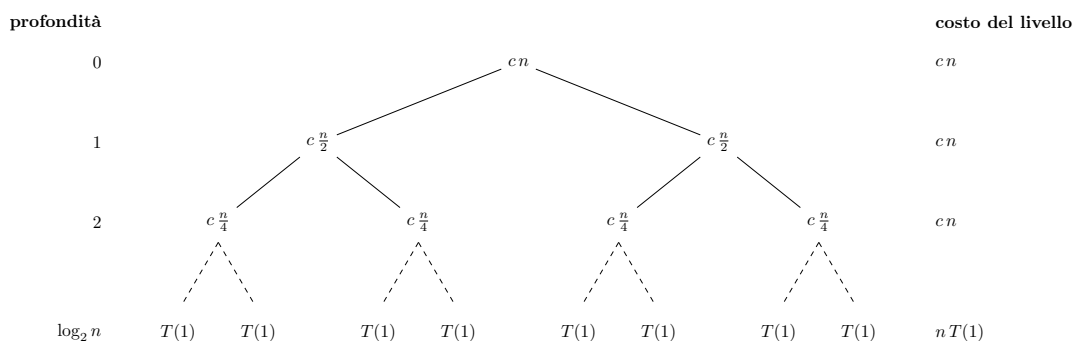


Figura 15.2: Albero di ricorrenza di $T(n) = 2T(n/2) + cn$: ogni livello ha costo totale cn .

Esempio 15.10: Risolvere $T(n) = 2T(n/2) + cn$ con l'albero di ricorrenza

- Passi 1 e 2.** A profondità i ci sono 2^i nodi, ciascuno corrispondente a un sottoproblema di dimensione $\frac{n}{2^i}$ e con costo $c \frac{n}{2^i}$. Il costo di ciascun livello è quindi

$$2^i c \frac{n}{2^i} = cn.$$

- *Passo 3.* La dimensione del sottoproblema raggiunge 1 a profondità $\log_2 n$. Il livello più basso ha $2^{\log_2 n} = n$ foglie, con costo totale $nT(1) = \Theta(n)$.
- *Passo 4.* Sommando i costi di tutti i livelli:

$$T(n) = \sum_{i=0}^{\log_2 n - 1} cn + nT(1) = cn \log_2 n + nT(1) = \Theta(n \log_2 n).$$

Per $c = 1$ e $T(1) = 1$ otteniamo di nuovo $T(n) = n \log_2 n + n$ (Esempio 15.7).

La soluzione di $T(n) = 2T(n/2) + cn$ è $T(n) = \Theta(n \log_2 n)$. Tutti i livelli hanno lo **stesso costo** cn , e ci sono $\Theta(\log_2 n)$ livelli.

15.4.3 Un albero sbilanciato: $T(n) = T(n/3) + T(2n/3) + cn$

- Consideriamo la ricorrenza ($c, d > 0$ costanti), definita per ogni numero reale $n > 0$:

$$T(n) = \begin{cases} d & \text{se } 0 < n < 1 \\ T(n/3) + T(2n/3) + cn & \text{se } n \geq 1 \end{cases}$$

(Lavorare con dimensioni reali ci permette di evitare le parti intere inferiori e superiori.)

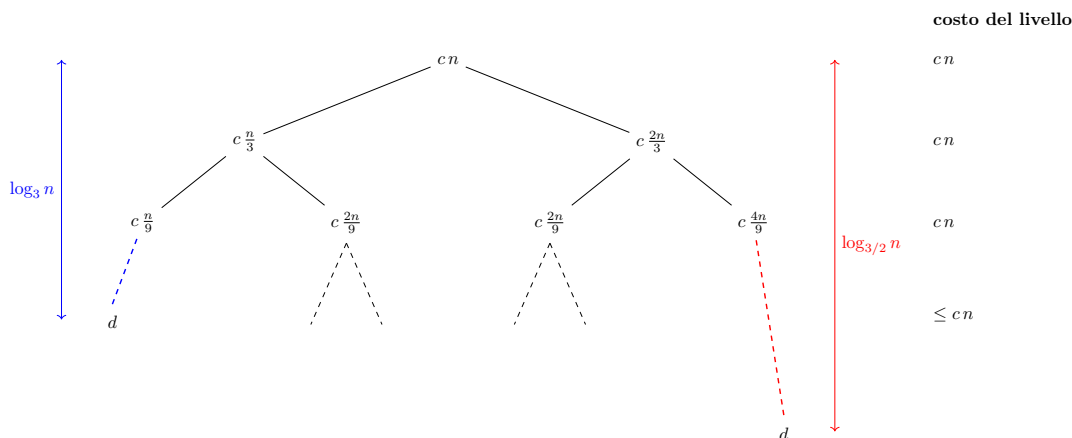


Figura 15.3: Albero di ricorsione di $T(n) = T(n/3) + T(2n/3) + cn$: il cammino più corto dalla radice a una foglia (blu) ha lunghezza circa $\log_3 n$, il più lungo (rosso) circa $\log_{3/2} n$.

- Ogni nodo interno di dimensione s ha due figli di dimensioni $\frac{s}{3}$ e $\frac{2s}{3}$, la cui somma è s . Quindi le dimensioni dei nodi a una data profondità sommano al più a n (esattamente n se il livello è completo).
- Il **cammino più corto** dalla radice a una foglia è quello più a sinistra: $n \rightarrow \frac{n}{3} \rightarrow \frac{n}{9} \rightarrow \dots$. Un nodo a profondità i su questo cammino ha dimensione $\frac{n}{3^i}$, che è ≥ 1 finché $i \leq \log_3 n$.
- Il **cammino più lungo** è quello più a destra: $n \rightarrow \frac{2n}{3} \rightarrow \frac{4n}{9} \rightarrow \dots$. Un nodo a profondità i su questo cammino ha dimensione $(\frac{2}{3})^i n$, che è ≥ 1 finché $i \leq \log_{3/2} n$.

- A differenza degli esempi precedenti, l'albero **non è completo**: le foglie si trovano a profondità diverse, comprese (approssimativamente) tra $\log_3 n$ e $\log_{3/2} n$.

Esempio 15.11: Limitazioni per l'albero sbilanciato

Limitazione superiore.

- *Nodi interni.* Ogni nodo a profondità i ha dimensione al più $(\frac{2}{3})^i n$; un nodo interno ha dimensione ≥ 1 , quindi i nodi interni compaiono solo a profondità $i \leq \log_{3/2} n$. A ciascuna profondità, i nodi interni hanno dimensione totale al più n , quindi il loro costo totale è al più cn . Pertanto il costo dei nodi interni è al più

$$cn \left(\log_{3/2} n + 1 \right).$$

- *Foglie.* Una foglia ha dimensione < 1 , ma il suo genitore ha dimensione ≥ 1 , quindi una foglia ha dimensione $\geq \frac{1}{3}$. Le dimensioni di tutte le foglie sommano esattamente a n (ogni nodo interno ripartisce la sua dimensione tra i suoi figli). Quindi ci sono al più $3n$ foglie, con costo totale al più $3dn = O(n)$.

- Pertanto

$$T(n) \leq cn \left(\log_{3/2} n + 1 \right) + 3dn = O(n \log_2 n),$$

poiché $\log_{3/2} n = \frac{\log_2 n}{\log_2(3/2)}$ differisce da $\log_2 n$ per un fattore costante (15.6).

Limitazione inferiore.

- Ogni nodo a profondità i ha dimensione almeno $\frac{n}{3^i}$. Quindi, per ogni profondità $i \leq \log_3 n$, tutti i nodi a profondità i hanno dimensione ≥ 1 : sono nodi interni, il livello è completo (2^i nodi) e le loro dimensioni sommano esattamente a n . Il costo di ciascuno di questi livelli è esattamente cn .
- Ci sono $\lfloor \log_3 n \rfloor + 1 \geq \log_3 n$ tali livelli (profondità $0, 1, \dots, \lfloor \log_3 n \rfloor$), quindi

$$T(n) \geq cn \log_3 n = \Omega(n \log_2 n).$$

Verifica numerica ($c = d = 1$): calcolando $T(n)$ esattamente dalla ricorrenza, $T(100) \approx 929.2$ e $T(1000) \approx 13\,002.3$; entrambi i valori sono compresi tra la limitazione inferiore $n \log_3 n$ (419.2 e 6\,287.7) e la limitazione superiore $n(\log_{3/2} n + 1) + 3n$ (1\,535.8 e 21\,036.6).

La soluzione di $T(n) = T(n/3) + T(2n/3) + cn$ è $T(n) = \Theta(n \log_2 n)$. Ogni livello costa al più cn ; la limitazione superiore usa il **cammino più lungo** ($\log_{3/2} n$) e quella inferiore usa il **cammino più corto** ($\log_3 n$). Entrambi sono $\Theta(\log_2 n)$.

- **Osservazione.** Se l'albero fosse un albero binario completo di altezza $\log_{3/2} n$, avrebbe $2^{\log_{3/2} n} = n^{\log_{3/2} 2} \approx n^{1.71}$ foglie, che è $\omega(n \log_2 n)$. L'argomento precedente mostra che l'albero è lontano dall'essere completo: il numero effettivo di foglie è $O(n)$. Quando le foglie non sono facili da contare, un approccio sicuro è usare l'albero di ricorsione solo per *ipotizzare* la soluzione e poi dimostrarla con il metodo di sostituzione.

15.4.4 Costi crescenti per livello: $T(n) = 4T(n/2) + n$ **Esempio 15.12:** Risolvere $T(n) = 4T(n/2) + n$ con l'albero di ricorsione

Consideriamo la ricorrenza

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ 4T\left(\frac{n}{2}\right) + n & \text{se } n > 1 \end{cases}$$

dove n è una potenza esatta di 2.

- *Passi 1 e 2.* A profondità i ci sono 4^i nodi, ciascuno di dimensione $\frac{n}{2^i}$ e costo $\frac{n}{2^i}$. Il costo del livello è

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

- *Passo 3.* La dimensione raggiunge 1 a profondità $\log_2 n$, dove ci sono $4^{\log_2 n} = n^2$ foglie:

profondità	numero di nodi	costo di ciascun nodo	costo del livello
0	1	n	n
1	4	$n/2$	$2n$
2	16	$n/4$	$4n$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	4^i	$n/2^i$	$2^i n$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\log_2 n$	$4^{\log_2 n} = n^2$	$T(1) = 1$	n^2

- *Passo 4.* I costi *crescono* geometricamente scendendo lungo l'albero. Usando (15.4),

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

Quindi $T(n) = 2n^2 - n = \Theta(n^2)$: infatti $\lim_{n \rightarrow \infty} \frac{2n^2 - n}{n^2} = 2 \in \mathbb{R}_{>0}$. Verifica: $T(2) = 4 + 2 = 6$, $T(4) = 24 + 4 = 28$, $T(8) = 112 + 8 = 120 = 2 \cdot 64 - 8$.

La soluzione di $T(n) = 4T(n/2) + n$ è $T(n) = \Theta(n^2)$. I costi dei livelli crescono geometricamente: il costo totale è dominato dalle **foglie**, il cui numero è $4^{\log_2 n} = n^{\log_2 4} = n^2$.

15.5 Verso il Master theorem

I tre alberi bilanciati della sezione precedente sono istanze della stessa figura. Disegniamo ora l'albero di ricorsione di una generica ricorrenza in forma standard (15.1)

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

con $a \geq 1$, $b > 1$ costanti e n potenza esatta di b .

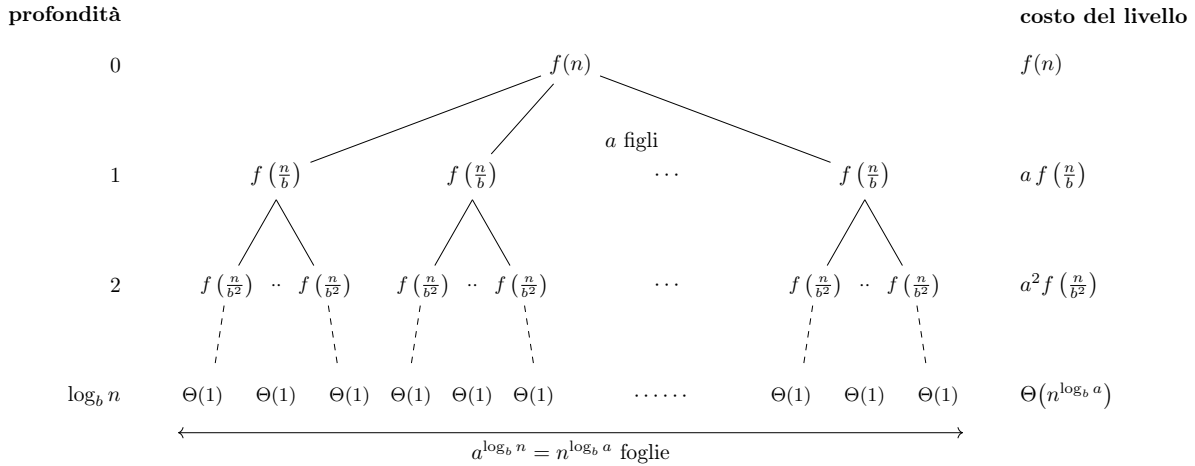


Figura 15.4: Albero di ricorsione di una ricorrenza in forma standard $T(n) = aT(n/b) + f(n)$, per n potenza esatta di b .

- A profondità j ci sono a^j nodi, ciascuno corrispondente a un sottoproblema di dimensione $\frac{n}{b^j}$ con costo $f\left(\frac{n}{b^j}\right)$. Il costo del livello è

$$a^j f\left(\frac{n}{b^j}\right).$$

- La dimensione del sottoproblema raggiunge 1 a profondità $\log_b n$. Il numero di foglie è, per (15.6),

$$a^{\log_b n} = n^{\log_b a},$$

e il loro costo totale è $\Theta(n^{\log_b a})$.

- Sommando i costi di tutti i livelli:

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right). \quad (15.9)$$

- La soluzione dipende da come i costi dei livelli $a^j f(n/b^j)$ evolvono al variare di j , cioè dal confronto tra $f(n)$ (il costo della radice) e $n^{\log_b a}$ (il costo delle foglie). Ci sono tre possibili comportamenti, tutti già incontrati:

1. **Dominano le foglie**: i costi dei livelli crescono geometricamente scendendo lungo l'albero, e $T(n) = \Theta(n^{\log_b a})$.

Esempio: $T(n) = 4T(n/2) + n$, con $n^{\log_2 4} = n^2$ foglie, $T(n) = \Theta(n^2)$ (Esempio 15.12).

2. **Tutti i livelli hanno (circa) lo stesso costo:** il costo di ciascuno dei $\log_b n + 1$ livelli è $\Theta(n^{\log_b a})$, e $T(n) = \Theta(n^{\log_b a} \log_2 n)$.
Esempio: $T(n) = 2T(n/2) + cn$, con $n^{\log_2 2} = n$, $T(n) = \Theta(n \log_2 n)$ (Esempio 15.10).
3. **Domina la radice:** i costi dei livelli decrescono geometricamente scendendo lungo l'albero, e $T(n) = \Theta(f(n))$.
Esempio: $T(n) = 3T(n/4) + cn^2$, con $f(n) = cn^2$, $T(n) = \Theta(n^2)$ (Esempio 15.9).

Il **Master theorem**, studiato nel capitolo *Il Master theorem*, rende precisi questi tre comportamenti. Confronta $f(n)$ con $n^{\log_b a}$ e fornisce direttamente la soluzione di una ricorrenza in forma standard, senza disegnare l'albero di ricorsione.

15.6 Riepilogo: quale metodo quando

- **Metodo di espansione:** ricorrenze con un *solo* termine ricorsivo, per esempio $T(n-1) + f(n)$, $2T(n-1) + 1$, $T(n/2) + c$, $2T(n/2) + n$. Fornisce la forma chiusa esatta quando la sommatoria che ne risulta si può calcolare (serie aritmetica o geometrica).
- **Metodo di sostituzione:** quando abbiamo un'ipotesi e vogliamo una dimostrazione *rigorosa*, in particolare con parti intere e condizioni al contorno esplicite. Si dimostrano O e Ω separatamente; i casi base si trattano scegliendo n_0 ; se l'induzione fallisce per un termine di ordine inferiore, si sottrae un termine di ordine inferiore dall'ipotesi. Si può combinare con un cambio di variabile.
- **Metodo dell'albero di ricorsione:** per visualizzare i costi e ottenere un'ipotesi; ideale quando i termini ricorsivi hanno *dimensioni diverse* (alberi sbilanciati, come $T(n/3) + T(2n/3) + cn$). Si calcolano il costo per livello, la profondità, il numero di foglie, e si somma usando la serie geometrica.
- **Master theorem** (capitolo *Il Master theorem*): ricorrenze in *forma standard* $aT(n/b) + f(n)$; dà la risposta direttamente confrontando $f(n)$ con $n^{\log_b a}$. Non si applica a $T(n-1) + f(n)$ né alle suddivisioni sbilanciate.

Il Master theorem

Introduzione

Nel capitolo *Equazioni di ricorrenza* abbiamo risolto ricorrenze per espansione, per sostituzione e con gli alberi di ricorsione. Ogni metodo richiede qualche calcolo, e per le ricorrenze divide et impera il calcolo è sempre lo stesso: disegnare l'albero, sommare i costi dei livelli, confrontare la radice con le foglie. Alla fine di quel capitolo abbiamo osservato tre possibili comportamenti: dominano le foglie, tutti i livelli costano uguale, oppure domina la radice.

Il **Master theorem** trasforma questi tre comportamenti in un risultato pronto all'uso. In questo capitolo

- richiamiamo le ricorrenze in forma standard e gli algoritmi che le producono (Sezione 16.1);
- enunciamo il teorema e ne spieghiamo i tre casi (Sezione 16.2);
- lo applichiamo con uno schema fisso in cinque passi a molti esempi (Sezione 16.3), e mostriamo le situazioni in cui non può essere applicato (Sezione 16.4);
- lo dimostriamo (Sezione 16.5), prima per potenze esatte di b e poi per ogni numero naturale.

16.1 Ricorrenze in forma standard

- Nel capitolo *Equazioni di ricorrenza* abbiamo visto che il numero di passi eseguiti da un algoritmo ricorsivo è descritto naturalmente da una **equazione di ricorrenza**, cioè un'equazione che esprime il valore di una funzione su un input di dimensione n in termini dei suoi valori su input più piccoli.
- Una grande famiglia di algoritmi *divide et impera* produce ricorrenze di una forma molto specifica, detta **forma standard**:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

dove:

- n è la dimensione dell'input;
- a è il **numero di sottoproblemi** generati da ciascuna chiamata ricorsiva ($a \geq 1$, costante);

- $\frac{n}{b}$ è la **dimensione di ciascun sottoproblema** ($b > 1$, costante); quando n/b non è un numero naturale si intende $\lfloor n/b \rfloor$ oppure $\lceil n/b \rceil$;
- $f(n)$ è il **costo di dividere** il problema negli a sottoproblemi **e di combinare** le loro soluzioni (tutto ciò che viene fatto al di fuori delle chiamate ricorsive). Supponiamo sempre che $f(n)$ sia **asintoticamente positiva**, cioè $f(n) > 0$ per ogni n sufficientemente grande.
- Per n sufficientemente piccolo (per esempio $n = 1$) la ricorsione si ferma e $T(n) = \Theta(1)$: risolvere un problema di dimensione costante richiede un numero costante di passi. Per questo motivo il caso base di solito non viene nemmeno scritto.

In questo capitolo una ricorrenza è un **oggetto matematico**: usiamo una generica funzione $T(n)$. Quando analizziamo un algoritmo ricorsivo, la funzione descritta dalla ricorrenza è il numero di passi $S(n)$ eseguiti nel caso peggiore su un input di parametro di dimensione n ; il tempo di esecuzione nel caso peggiore $T(|I|)$ si ottiene poi da $S(n)$ esattamente come nel capitolo *Tempo di esecuzione*.

- Esempi di ricorrenze in forma standard (saranno tutte risolte nella Sezione 16.3):

Algoritmo	Ricorrenza	a	b	$f(n)$
Ricerca binaria (ricorsiva)	$T(n) = T(n/2) + 1$	1	2	1
Somma di un array (ricorsiva)	$T(n) = 2T(n/2) + 1$	2	2	1
Merge sort	$T(n) = 2T(n/2) + n$	2	2	n
Moltiplicazione di Karatsuba	$T(n) = 3T(n/2) + n$	3	2	n
Moltiplicazione di matrici di Strassen	$T(n) = 7T(n/2) + n^2$	7	2	n^2

- Il **Master theorem** fornisce l'ordine di grandezza di $T(n)$ per la maggior parte delle ricorrenze in forma standard, direttamente da a , b e $f(n)$, senza alcun calcolo sulla ricorsione.

16.2 Il Master theorem

Teorema 16.1: Master theorem

Siano $a \geq 1$ e $b > 1$ costanti, sia $f(n)$ una funzione asintoticamente positiva, e sia $T(n)$ definita sui numeri naturali dalla ricorrenza

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

dove n/b è interpretato come $\lfloor n/b \rfloor$ oppure $\lceil n/b \rceil$, e $T(n) = \Theta(1)$ per n sufficientemente piccolo. Allora $T(n)$ ha le seguenti limitazioni asintotiche.

Caso 1: se $f(n) = O(n^{\log_b a - \epsilon})$ per qualche costante $\epsilon > 0$, allora

$$T(n) = \Theta(n^{\log_b a});$$

Caso 2: se $f(n) = \Theta(n^{\log_b a} \log^k n)$ per qualche costante $k \geq 0$, 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 costante $\epsilon > 0$, e se

$$a f\left(\frac{n}{b}\right) \leq c f(n)$$

per qualche costante $c < 1$ e per ogni n sufficientemente grande (**condizione di regolarità**), allora

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

- Questo è l'enunciato classico del teorema (come nel libro di testo di Cormen, Leiserson, Rivest e Stein, *Introduction to Algorithms*), con il **caso 2 esteso**: per $k = 0$ si ottiene l'enunciato di base "se $f(n) = \Theta(n^{\log_b a})$ allora $T(n) = \Theta(n^{\log_b a} \log n)$ ".
- Le notazioni O , Ω e Θ dell'enunciato sono quelle definite nel capitolo 9 «Notazione asintotica».
- La dimostrazione è data nella Sezione 16.5.

16.2.1 Regola pratica

- In ciascuno dei tre casi confrontiamo la funzione $f(n)$ con la funzione

$$n^{\log_b a},$$

che (come vedremo nel Lemma 16.3) è il numero di foglie dell'albero di ricorsione. Come regola pratica, si controlla:

$$n^{\log_b a} \begin{cases} \text{polinomialmente} & \longrightarrow \text{caso 1: } T(n) = \Theta(n^{\log_b a}) \\ \text{più grande di } f(n) & \\ \text{dello stesso ordine di } f(n) & \longrightarrow \text{caso 2: } T(n) = \Theta(n^{\log_b a} \log^{k+1} n) \\ \text{(a meno di } \log^k n) & \\ \text{polinomialmente} & \longrightarrow \text{caso 3: } T(n) = \Theta(f(n)) \\ \text{più piccolo di } f(n) & \end{cases}$$

e si determini di conseguenza l'ordine di grandezza di $T(n)$ (se possibile).

- **Intuitivamente, la più grande tra le due funzioni determina la soluzione della ricorrenza.**
- **Polinomialmente** più piccola (più grande) significa più piccola (più grande) di un fattore n^ϵ per qualche costante $\epsilon > 0$. Per esempio:
 - n è polinomialmente più piccolo di n^2 (fattore n^1), e n^2 è polinomialmente più piccolo di $n^{2.1}$ (fattore $n^{0.1}$);
 - n è più piccolo di $n \log n$, ma **non** polinomialmente più piccolo: 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 $n \log n$ cresce più lentamente di $n \cdot n^\epsilon$, qualunque $\epsilon > 0$ scegliamo.

16.2.2 Intuizione

- Srotolando la ricorrenza (si veda l'albero di ricorsione nella Figura 16.1) il costo totale è

$$T(n) = \underbrace{\Theta(n^{\log_b a})}_{\text{costo delle foglie}} + \underbrace{\sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right)}_{\text{costo dei livelli interni}}$$

dove il livello j dell'albero contiene a^j sottoproblemi di dimensione n/b^j .

- Consideriamo il caso semplice $f(n) = n^d$. Il costo del livello j è

$$a^j \left(\frac{n}{b^j}\right)^d = n^d \left(\frac{a}{b^d}\right)^j,$$

quindi i costi dei livelli formano una **progressione geometrica di ragione** a/b^d . Poiché $a/b^d > 1 \iff d < \log_b a$:

- **caso 1** ($d < \log_b a$): i costi dei livelli **crescono** geometricamente dalla radice alle foglie; il totale è dominato dalle foglie, cioè da $n^{\log_b a}$;
- **caso 2** ($d = \log_b a$): tutti i $\log_b n$ livelli hanno **lo stesso costo** $n^d = n^{\log_b a}$; il totale è $n^{\log_b a}$ per il numero di livelli, cioè $n^{\log_b a} \log n$;
- **caso 3** ($d > \log_b a$): i costi dei livelli **decrescono** geometricamente; il totale è dominato dalla radice, cioè da $f(n)$.

16.2.3 I vuoti tra i casi

- I tre casi **non coprono tutte le possibili funzioni** $f(n)$.
 - C'è un vuoto tra il caso 1 e il caso 2 quando $f(n)$ è più piccola di $n^{\log_b a}$ ma **non polinomialmente più piccola** (e non della forma $n^{\log_b a} \log^k n$ con $k \geq 0$), per esempio $f(n) = n^{\log_b a} / \log n$.
 - C'è un vuoto tra il caso 2 e il caso 3 quando $f(n)$ è più grande di $n^{\log_b a}$ ma **non polinomialmente più grande** (e non della forma $n^{\log_b a} \log^k n$), per esempio $f(n) = n^{\log_b a} 2^{\sqrt{\log_2 n}}$.
 - Il caso 3 non si applica anche quando $f(n)$ è polinomialmente più grande di $n^{\log_b a}$ ma la **condizione di regolarità** non vale.
- In queste situazioni il Master theorem **non può essere usato**, e la ricorrenza deve essere risolta con altre tecniche (albero di ricorsione, iterazione, metodo di sostituzione), si veda la Sezione 16.4.

16.2.4 Una nota sulla base dei logaritmi

- Poiché $\log_c n = \frac{\log_2 n}{\log_2 c}$ per ogni base $c > 1$, abbiamo

$$\log_c^k n = \frac{1}{\log_2^k c} \log_2^k n = \Theta(\log_2^k n).$$

Quindi, **dentro** Θ , O e Ω la base del logaritmo è irrilevante: nel caso 2 scriviamo semplicemente $\log^k n$.

- Al contrario, la base b nell'esponente $n^{\log_b a}$ è essenziale: per esempio $n^{\log_2 4} = n^2$ mentre $n^{\log_4 4} = n$, e $n^2 \neq \Theta(n)$.

- Il Master theorem confronta il costo della radice, $f(n)$, con il numero di foglie dell'albero di ricorsione, $n^{\log_b a}$: la più grande tra le due determina $T(n)$; quando sono dello stesso ordine (a meno di $\log^k n$) compare un ulteriore fattore $\log n$.
- “Più grande” e “più piccola” devono essere **polinomiali** (di un fattore n^ϵ); nel caso 3 deve valere anche la **condizione di regolarità**.
- Quando $f(n)$ cade in un vuoto tra i casi, il teorema non dà alcuna risposta.

16.3 Applicare il Master theorem

Per risolvere una ricorrenza con il Master theorem seguiamo sempre questo schema:

- identificare** a , b e $f(n)$ (e controllare che $a \geq 1$, $b > 1$ siano costanti e $f(n)$ sia asintoticamente positiva);
- calcolare** $n^{\log_b a}$;
- confrontare** $n^{\log_b a}$ con $f(n)$ e ipotizzare il caso;
- verificare** l'ipotesi del caso: trovare $\epsilon > 0$ (caso 1 o 3), trovare $k \geq 0$ (caso 2), e controllare la condizione di regolarità (caso 3);
- concludere**.

Cominciamo con tre ricorrenze, una per ciascun caso, e poi risolviamo le ricorrenze degli algoritmi elencati nella Sezione 16.1.

Esempio 16.1: $T(n) = 16T(n/4) + n$

- $a = 16$, $b = 4$, $f(n) = n$.
- $n^{\log_b a} = n^{\log_4 16} = n^2$.
- n^2 è polinomialmente più grande di $f(n) = n \rightarrow$ **caso 1**.
- $f(n) = n = O(n^{2-\epsilon})$ con $\epsilon = 1$.

$$5. T(n) = \Theta(n^{\log_4 16}) = \Theta(n^2).$$

Esempio 16.2: $T(n) = 8T(n/2) + n^3$

1. $a = 8, b = 2, f(n) = n^3$.
2. $n^{\log_b a} = n^{\log_2 8} = n^3$.
3. n^3 e $f(n) = n^3$ hanno lo stesso ordine $\rightarrow$ **caso 2**.
4. $f(n) = n^3 = \Theta(n^3 \log^k n)$ con $k = 0$.
5. $T(n) = \Theta(n^3 \log^{k+1} n) = \Theta(n^3 \log n)$.

Esempio 16.3: $T(n) = 8T(n/3) + n^2$

1. $a = 8, b = 3, f(n) = n^2$.
2. $n^{\log_b a} = n^{\log_3 8} \approx n^{1.893}$.
3. $n^{1.893}$ è polinomialmente più piccolo di $f(n) = n^2 \rightarrow$ **caso 3**.
4. $f(n) = n^2 = \Omega(n^{\log_3 8 + \epsilon})$ per ogni $0 < \epsilon \leq 2 - \log_3 8 \approx 0.107$.
Condizione di regolarità: cerchiamo una costante $c < 1$ tale che, per ogni n grande,

$$8f\left(\frac{n}{3}\right) \leq cf(n) \iff 8\left(\frac{n}{3}\right)^2 \leq cn^2 \iff \frac{8}{9}n^2 \leq cn^2,$$

che vale per ogni c con $\frac{8}{9} \leq c < 1$, per esempio $c = \frac{8}{9}$.

$$5. T(n) = \Theta(n^2).$$

Esempio 16.4: Ricerca binaria: $T(n) = T(n/2) + 1$

1. $a = 1, b = 2, f(n) = 1$.
2. $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$.
3. 1 e $f(n) = 1$ hanno lo stesso ordine $\rightarrow$ **caso 2**.
4. $f(n) = 1 = \Theta(n^0 \log^k n)$ con $k = 0$.
5. $T(n) = \Theta(n^0 \log n) = \Theta(\log n)$.

Esempio 16.5: Merge sort (ordinamento per fusione): $T(n) = 2T(n/2) + n$

1. $a = 2, b = 2, f(n) = n$.
2. $n^{\log_b a} = n^{\log_2 2} = n$.

3. n e $f(n) = n$ hanno lo stesso ordine $\rightarrow$ **caso 2**.

4. $f(n) = n = \Theta(n \log^k n)$ con $k = 0$.

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

Esempio 16.6: Somma di un array: $T(n) = 2T(n/2) + 1$

1. $a = 2, b = 2, f(n) = 1$.

2. $n^{\log_b a} = n^{\log_2 2} = n$.

3. n è polinomialmente più grande di $f(n) = 1 \rightarrow$ **caso 1**.

4. $f(n) = 1 = O(n^{1-\epsilon})$ con $\epsilon = 1$ (infatti $n^{1-1} = n^0 = 1$).

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

La funzione C++ `sumRecHalving` del codice del corso è un algoritmo di questo tipo: somma il sotto-array `a[1..r]` (indici a partire da 0) dividendolo in due metà.

```

1 long long sumRecHalving(int *a, int l, int r)
2 {
3     if (l == r)
4     {
5         return a[l];
6     }
7
8     int mid = l + (r - l) / 2;
9
10    return sumRecHalving(a, l, mid) + sumRecHalving(a, mid + 1, r);
11 }
```

Ogni chiamata esegue una quantità costante di lavoro al di fuori delle chiamate ricorsive, e il numero totale di chiamate è $\Theta(n)$: contandole con una copia strumentata della funzione, su $n = 8, 1000$ e $2^{20} = 1\,048\,576$ elementi otteniamo 15, 1999 e 2097151 chiamate, cioè esattamente $2n - 1$.

Esempio 16.7: Karatsuba: $T(n) = 3T(n/2) + n$

1. $a = 3, b = 2, f(n) = n$.

2. $n^{\log_b a} = n^{\log_2 3} \approx n^{1.585}$.

3. $n^{1.585}$ è polinomialmente più grande di $f(n) = n \rightarrow$ **caso 1**.

4. $f(n) = n = O(n^{\log_2 3 - \epsilon})$ con $\epsilon = \log_2 3 - 1 \approx 0.585 > 0$.

5. $T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$: l'algoritmo di Karatsuba è asintoticamente più veloce della moltiplicazione scolastica $\Theta(n^2)$.

Esempio 16.8: Strassen: $T(n) = 7T(n/2) + n^2$

1. $a = 7, b = 2, f(n) = n^2$.
2. $n^{\log_b a} = n^{\log_2 7} \approx n^{2.807}$.
3. $n^{2.807}$ è polinomialmente più grande di $f(n) = n^2 \rightarrow$ **caso 1**.
4. $f(n) = n^2 = O(n^{\log_2 7 - \epsilon})$ con $\epsilon = \log_2 7 - 2 \approx 0.807 > 0$.
5. $T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$, meglio del $\Theta(n^3)$ della moltiplicazione di matrici standard (la cui ricorrenza divide et impera è $T(n) = 8T(n/2) + n^2$).

Esempio 16.9: $T(n) = 2T(n/2) + n \log n$

1. $a = 2, b = 2, f(n) = n \log n$.
2. $n^{\log_b a} = n^{\log_2 2} = n$.
3. $f(n) = n \log n$ è più grande di n , ma solo di un fattore $\log n$, non polinomialmente $\rightarrow$ non è il caso 3; ha la forma $n^{\log_b a} \log^k n \rightarrow$ **caso 2**.
4. $f(n) = n \log n = \Theta(n \log^k n)$ con $k = 1$.
5. $T(n) = \Theta(n \log^2 n)$.

Esempio 16.10: $T(n) = 3T(n/4) + n \log n$

1. $a = 3, b = 4, f(n) = n \log n$ (logaritmo in base 2).
2. $n^{\log_b a} = n^{\log_4 3} \approx n^{0.792}$.
3. $n^{0.792}$ è polinomialmente più piccolo di $f(n) = n \log n \rightarrow$ **caso 3**.
4. Poiché $n \log n \geq n$ per $n \geq 2$, abbiamo $f(n) = \Omega(n^{\log_4 3 + \epsilon})$ con $\epsilon = 1 - \log_4 3 \approx 0.208$.
Condizione di regolarità: per ogni $n \geq 4$,

$$3f\left(\frac{n}{4}\right) = 3 \frac{n}{4} \log \frac{n}{4} \leq \frac{3}{4} n \log n = c f(n) \quad \text{con } c = \frac{3}{4} < 1,$$

perché $\log(n/4) \leq \log n$.

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

Esempio 16.11: $T(n) = T(2n/3) + 1$

1. $a = 1, b = \frac{3}{2}$ (infatti $\frac{2n}{3} = \frac{n}{3/2}$), $f(n) = 1$. Si noti che b non deve necessariamente essere un intero.
2. $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$.
3. 1 e $f(n) = 1$ hanno lo stesso ordine $\rightarrow$ **caso 2**.

4. $f(n) = 1 = \Theta(n^0 \log^k n)$ con $k = 0$.

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

- Una **verifica numerica**. Per ciascuna delle ricorrenze precedenti con b intero abbiamo calcolato $T(n)$ esattamente dalla ricorrenza, con $T(1) = 1$ e $n = b^L$, e l'abbiamo divisa per l'ordine di grandezza $g(n)$ dato dal teorema (logaritmi in base 2). Il rapporto resta compreso tra due costanti positive, come garantisce il teorema, e si avvicina al limite indicato nell'ultima colonna.

Ricorrenza	$g(n)$	$L = 5$	$L = 10$	$L = 20$	limite
$16T(n/4) + n$	n^2	1.3330	1.3333	1.3333	$4/3$
$8T(n/2) + n^3$	$n^3 \log n$	1.2000	1.1000	1.0500	1
$8T(n/3) + n^2$	n^2	4.5606	6.5364	8.2414	9
$T(n/2) + 1$	$\log n$	1.2000	1.1000	1.0500	1
$2T(n/2) + n$	$n \log n$	1.2000	1.1000	1.0500	1
$2T(n/2) + 1$	n	1.9688	1.9990	2.0000	2
$3T(n/2) + n$	$n^{\log_2 3}$	2.7366	2.9653	2.9994	3
$7T(n/2) + n^2$	$n^{\log_2 7}$	2.2521	2.3284	2.3333	$7/3$
$2T(n/2) + n \log n$	$n \log^2 n$	0.6400	0.5600	0.5275	$1/2$
$3T(n/4) + n \log n$	$n \log n$	2.1933	2.8704	3.4020	4

- Per il merge sort, per esempio, $T(2^L) = 2^L(L+1)$, quindi $T(n) = n \log_2 n + n$ e $\frac{T(n)}{n \log_2 n} = 1 + \frac{1}{L}$. Nel caso 3 la convergenza è lenta: il rapporto tende a $\frac{1}{1-c}$ (9 per $c = \frac{8}{9}$, 4 per $c = \frac{3}{4}$), la limitazione che compare nella dimostrazione del teorema.

Il teorema dà l'**ordine di grandezza**, non la costante: nella tabella i rapporti tendono a costanti diverse, ma non crescono né svaniscono mai.

16.4 Quando il Master theorem non si applica

- Prima di applicare il teorema, si controlli sempre che valgano **tutte** le sue ipotesi. Ecco situazioni tipiche in cui il teorema **non può** essere applicato.

Esempio 16.12: Un vuoto tra il caso 1 e il caso 2: $T(n) = 2T(n/2) + n/\log_2 n$

- $a = 2$, $b = 2$, $f(n) = n/\log_2 n$, e $n^{\log_b a} = n$.
- **Non è il caso 1:** $f(n)$ è più piccola di n , ma non polinomialmente più piccola. Per ogni $\epsilon > 0$

$$\lim_{n \rightarrow \infty} \frac{n/\log_2 n}{n^{1-\epsilon}} = \lim_{n \rightarrow \infty} \frac{n^\epsilon}{\log_2 n} = +\infty,$$

quindi $f(n) \neq O(n^{1-\epsilon})$.

- **Non è il caso 2:** $f(n) = n \log^{-1} n$ richiederebbe $k = -1 < 0$.

- **Non è il caso 3:** $f(n)$ è più piccola di n .
- **Soluzione con l'albero di ricorsione** (Lemma 16.3), per $n = 2^L$ e $T(1) = 1$: il livello j costa

$$2^j \frac{n/2^j}{\log_2(n/2^j)} = \frac{n}{L-j}, \quad \text{quindi} \quad T(n) = n + \sum_{j=0}^{L-1} \frac{n}{L-j} = n + n \sum_{i=1}^L \frac{1}{i} = n(1 + H_L),$$

dove $H_L = \sum_{i=1}^L \frac{1}{i}$ è l' L -esimo numero armonico. Poiché $\ln(L+1) \leq H_L \leq 1 + \ln L$ (confronto con $\int \frac{dx}{x}$), $H_L = \Theta(\log L)$ e

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

Esempio 16.13: Non in forma standard: $T(n) = T(n-1) + n$

- Il sottoproblema ha dimensione $n-1$, non n/b con $b > 1$ costante: la ricorrenza **non è in forma standard**.
- Soluzione per iterazione (con $T(1) = 1$):

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

Esempio 16.14: Sottoproblemi di dimensioni diverse: $T(n) = T(n/3) + T(2n/3) + n$

- I due sottoproblemi hanno **dimensioni diverse** $n/3$ e $2n/3$: la ricorrenza non è della forma $aT(n/b) + f(n)$.
- Albero di ricorsione: ogni livello completo ha dimensione totale n , quindi costo n . Il cammino più corto dalla radice a una foglia (sempre $n/3$) ha lunghezza $\log_3 n$, il più lungo (sempre $2n/3$) ha lunghezza $\log_{3/2} n$. Pertanto i primi $\log_3 n$ livelli sono completi (costo $\geq n \log_3 n$), e ci sono al più $\log_{3/2} n + 1$ livelli, ciascuno di costo al più n . Quindi

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

Esempio 16.15: a non è costante: $T(n) = 2^n T(n/2) + n^n$

- Il numero di sottoproblemi $a = 2^n$ **dipende da n** : il Master theorem richiede che a sia una costante.

Esempio 16.16: $a < 1$: $T(n) = 0.5 T(n/2) + 1/n$

- Qui $a = 0.5 < 1$: il teorema richiede $a \geq 1$ (deve esserci almeno un sottoproblema).

Esempio 16.17: La condizione di regolarità non vale: $T(n) = T(n/2) + n(2 - \cos n)$

- $a = 1$, $b = 2$, $n^{\log_b a} = n^0 = 1$, $f(n) = n(2 - \cos n)$. Poiché $1 \leq 2 - \cos n \leq 3$, abbiamo $n \leq f(n) \leq 3n$, quindi

$$f(n) = \Theta(n) = \Omega(n^{0+\epsilon}) \quad \text{con } \epsilon = 1 :$$

$f(n)$ è polinomialmente più grande di $n^{\log_b a}$, e ci aspetteremmo il caso 3.

- **La condizione di regolarità non vale.** Prendiamo $n = 2m$ pari. Allora

$$\frac{af(n/2)}{f(n)} = \frac{m(2 - \cos m)}{2m(2 - \cos 2m)} = \frac{2 - \cos m}{2(2 - \cos 2m)}.$$

Se $\cos m \leq -0.9$, allora $2 - \cos m \geq 2.9$ e $\cos 2m = 2\cos^2 m - 1 \geq 2 \cdot 0.81 - 1 = 0.62$, quindi $2(2 - \cos 2m) \leq 2.76$ e il rapporto è > 1 . Poiché π è irrazionale, i valori $m \bmod 2\pi$ ($m \in \mathbb{N}$) sono densi in $[0, 2\pi)$, quindi esistono **infiniti** m con $\cos m \leq -0.9$ (per esempio $m = 3, 9, 16, 22, 28, \dots$). Per $m = 22$: $\cos 22 \approx -0.99996$, $\cos 44 \approx 0.99984$, e il rapporto è ≈ 1.4997 .

Quindi non esiste alcuna costante $c < 1$ tale che $af(n/2) \leq cf(n)$ per ogni n grande: **il caso 3 non può essere applicato.**

- Ciononostante la ricorrenza può essere risolta direttamente (con $n/2$ che significa $\lfloor n/2 \rfloor$): srotolandola,

$$n \leq f(n) \leq T(n) \leq \sum_{j \geq 0} 3 \frac{n}{2^j} + T(1) = 6n + T(1), \quad \text{quindi} \quad T(n) = \Theta(n).$$

16.5 Dimostrazione del Master theorem

- Seguiamo la dimostrazione classica, che ha due parti:
 1. supponiamo prima che n sia una **potenza esatta di b** , cioè $n = b^L$ per qualche $L \in \mathbb{N}$; allora tutte le dimensioni dei sottoproblemi $n/b, n/b^2, \dots$ sono potenze esatte di b e non servono parti intere inferiori o superiori (Lemmi 16.3, 16.4 e 16.5);
 2. estendiamo poi il risultato a ogni numero naturale n , quando n/b è $\lceil n/b \rceil$ oppure $\lfloor n/b \rfloor$ (Sezione 16.5.5).

16.5.1 Impostazione

- In tutta la dimostrazione poniamo

$$p = \log_b a, \quad \text{in modo che} \quad b^p = a \quad \text{e} \quad p \geq 0 \quad (\text{poiché } a \geq 1).$$

- Per n potenza esatta di b consideriamo la ricorrenza

$$T(n) = \begin{cases} d & \text{se } n = 1, \\ aT(n/b) + f(n) & \text{se } n = b^i, i \geq 1, \end{cases} \quad (16.1)$$

dove $d > 0$ è una costante (questo è il significato di $T(1) = \Theta(1)$).

- Supponiamo $f(m) > 0$ per ogni $m \geq 1$. Questo è sempre vero quando f conta i passi di un algoritmo; in generale, poiché f è asintoticamente positiva, solo un numero finito di valori è coinvolto e cambiano solo le costanti della dimostrazione.
- Le ipotesi dei tre casi valgono “per ogni n sufficientemente grande”, cioè per $n \geq n_0$. La seguente osservazione ci permette di usarle per **ogni** potenza esatta di b almeno uguale a b .

Osservazione 16.2: dalle potenze grandi a tutte le potenze

Siano $f(m) > 0$ e $h(m) > 0$ per ogni potenza esatta $m \geq b$ di b . Se esistono costanti $c > 0$ e n_0 tali che $f(m) \leq ch(m)$ (risp. $f(m) \geq ch(m)$) per tutte le potenze esatte $m \geq n_0$, allora esiste una costante $c' > 0$ tale che $f(m) \leq c'h(m)$ (risp. $f(m) \geq c'h(m)$) per **tutte** le potenze esatte $m \geq b$ di b .

Dimostrazione. Ci sono solo un numero finito di potenze esatte di b nell'intervallo $[b, n_0]$. Sia

$$c' = \max \left\{ c, \max_{b \leq m < n_0} \frac{f(m)}{h(m)} \right\} \quad \left(\text{risp. } c' = \min \left\{ c, \min_{b \leq m < n_0} \frac{f(m)}{h(m)} \right\} > 0 \right),$$

dove m varia tra le potenze esatte di b . Allora $f(m) \leq c'h(m)$ (risp. $\geq$) sia per $m < n_0$, per la scelta di c' , sia per $m \geq n_0$, poiché $c \leq c'$ (risp. $c \geq c'$). $\square$

- La applicheremo con $h(m) = m^{p-\epsilon}$ e $h(m) = m^p \log_b^k m$, che sono positive per $m \geq b$ (si noti che $\log_b m \geq 1$ per $m \geq b$).

16.5.2 L'albero di ricorsione

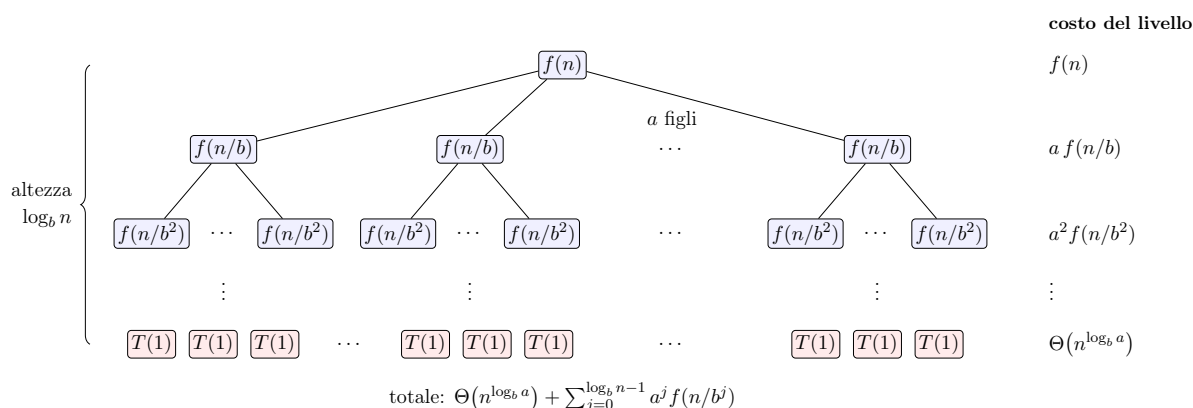


Figura 16.1: Albero di ricorsione di $T(n) = aT(n/b) + f(n)$ per n potenza esatta di b . Ogni nodo interno ha a figli; il livello j ha a^j nodi, ciascuno un sottoproblema di dimensione n/b^j e costo $f(n/b^j)$; le foglie sono a profondità $\log_b n$.

- La radice (livello 0) è un problema di dimensione n ; costa $f(n)$ e genera a sottoproblemi di dimensione n/b .
- Per induzione, il livello j contiene a^j nodi, ciascuno un sottoproblema di dimensione n/b^j e costo $f(n/b^j)$: il costo totale del livello j è $a^j f(n/b^j)$.
- La dimensione del sottoproblema raggiunge 1 quando $n/b^j = 1$, cioè a profondità $j = \log_b n$: le foglie sono gli $a^{\log_b n}$ nodi del livello $\log_b n$, ciascuno di costo $T(1) = d$.

Lemma 16.3: Costo dell'albero di ricorsione

Siano $a \geq 1$ e $b > 1$ costanti e sia $f(n) > 0$ per $n \geq 1$. Per ogni n potenza esatta di b , la soluzione della ricorrenza (16.1) è

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

Dimostrazione. Sia $n = b^L$. Dimostriamo prima per induzione su $i \in \{0, 1, \dots, L\}$ che

$$T(n) = a^i T\left(\frac{n}{b^i}\right) + \sum_{j=0}^{i-1} a^j f\left(\frac{n}{b^j}\right). \quad (16.2)$$

Per $i = 0$ la somma è vuota e (16.2) diventa $T(n) = T(n)$. Supponiamo che (16.2) valga per qualche $i < L$. Poiché $n/b^i = b^{L-i}$ con $L-i \geq 1$, la ricorrenza dà $T(n/b^i) = aT(n/b^{i+1}) + f(n/b^i)$, quindi

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

che è (16.2) per $i + 1$. Per $i = L = \log_b n$ si ha $T(n/b^L) = T(1) = d$, quindi

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

Resta da mostrare che $a^{\log_b n} = n^{\log_b a}$. Scrivendo $a = b^{\log_b a}$ e $n = b^{\log_b n}$,

$$a^{\log_b n} = (b^{\log_b a})^{\log_b n} = b^{\log_b a \cdot \log_b n} = (b^{\log_b n})^{\log_b a} = n^{\log_b a}.$$

Infine, $d > 0$ è una costante, quindi $d n^{\log_b a} = \Theta(n^{\log_b a})$. □

- Il primo termine è il costo delle **foglie**: ci sono esattamente $n^{\log_b a}$ foglie. La somma è il costo dei **nodi interni** (livelli $0, \dots, \log_b n - 1$).

Esempio 16.18: Lemma 16.3 per $T(n) = 2T(n/2) + n$, $T(1) = 1$, $n = 8$

Qui $a = 2$, $b = 2$, $f(n) = n$, $d = 1$, $\log_2 8 = 3$.

livello j	nodi a^j	dimensione n/b^j	costo per nodo	costo del livello
0	1	8	$f(8) = 8$	8
1	2	4	$f(4) = 4$	8
2	4	2	$f(2) = 2$	8
3 (foglie)	$8 = 8^{\log_2 2}$	1	$T(1) = 1$	8
totale				32

Il Lemma 16.3 dà $T(8) = 1 \cdot 8^{\log_2 2} + (8 + 8 + 8) = 8 + 24 = 32$. Direttamente dalla ricorrenza:

$$T(2) = 2 \cdot 1 + 2 = 4, \quad T(4) = 2 \cdot 4 + 4 = 12, \quad T(8) = 2 \cdot 12 + 8 = 32. \quad \checkmark$$

16.5.3 Limitare la somma

- Per il Lemma 16.3, per risolvere la ricorrenza ci serve l'ordine di grandezza di

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

- Un'identità chiave, usata molte volte: poiché $b^p = a$,

$$a^j \left(\frac{n}{b^j}\right)^p = \frac{a^j}{(b^p)^j} n^p = \frac{a^j}{a^j} n^p = n^p \quad \text{per ogni } j. \quad (16.3)$$

- Nella somma, $j \leq \log_b n - 1$, quindi le dimensioni $m = n/b^j$ sono potenze esatte di b con $m \geq b$: per l'Osservazione 16.2, le limitazioni su f dei casi 1 e 2 si possono usare per tutte.

Lemma 16.4: Limitazioni sulla somma

Siano $a \geq 1$ e $b > 1$ costanti, $p = \log_b a$, sia $f(m) > 0$ per $m \geq 1$, e sia $g(n)$ definita, per n potenza esatta di b , come sopra. Allora:

1. se $f(n) = O(n^{p-\epsilon})$ per qualche costante $\epsilon > 0$, allora $g(n) = O(n^p)$;
2. se $f(n) = \Theta(n^p \log^k n)$ per qualche costante $k \geq 0$, allora $g(n) = \Theta(n^p \log^{k+1} n)$;
3. se $f(n) = \Omega(n^{p+\epsilon})$ per qualche costante $\epsilon > 0$, e $a f(n/b) \leq c f(n)$ per qualche costante $c < 1$ e per tutte le potenze esatte $n \geq n_0$, allora $g(n) = \Theta(f(n))$.

Dimostrazione del caso 1. Per l'Osservazione 16.2 esiste una costante $c > 0$ tale che $f(m) \leq c m^{p-\epsilon}$ per ogni potenza esatta $m \geq b$. Quindi, usando (16.3),

$$g(n) \leq c \sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{p-\epsilon} = c \sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^p \left(\frac{b^j}{n}\right)^\epsilon = c n^{p-\epsilon} \sum_{j=0}^{\log_b n - 1} (b^\epsilon)^j.$$

Questa è una serie geometrica finita di ragione $b^\epsilon > 1$ (poiché $b > 1$ e $\epsilon > 0$):

$$\sum_{j=0}^{\log_b n - 1} (b^\epsilon)^j = \frac{(b^\epsilon)^{\log_b n} - 1}{b^\epsilon - 1} = \frac{(b^{\log_b n})^\epsilon - 1}{b^\epsilon - 1} = \frac{n^\epsilon - 1}{b^\epsilon - 1}.$$

Pertanto

$$0 < g(n) \leq c n^{p-\epsilon} \frac{n^\epsilon - 1}{b^\epsilon - 1} \leq \frac{c}{b^\epsilon - 1} n^p,$$

e poiché $\frac{c}{b^\epsilon - 1}$ è una costante positiva, $g(n) = O(n^p)$. □

Esempio 16.19: Caso 1 del Lemma 16.4: $T(n) = 16T(n/4) + n$, $n = 64$

Qui $p = \log_4 16 = 2$, $f(m) = m = m^{2-1}$, quindi $c = 1$, $\epsilon = 1$. I livelli interni costano

$$g(64) = 1 \cdot 64 + 16 \cdot 16 + 256 \cdot 4 = 64 + 256 + 1024 = 1344 :$$

i costi crescono del fattore $b^\epsilon = 4$ da un livello al successivo (sono dominati dai livelli più profondi). La formula della dimostrazione dà esattamente $n^{p-\epsilon} \frac{n^\epsilon - 1}{b^\epsilon - 1} = 64 \cdot \frac{63}{3} = 1344$, e la limitazione $\frac{c}{b^\epsilon - 1} n^p = \frac{4096}{3} \approx 1365.3$ vale. Aggiungendo le foglie, $T(64) = 4096 + 1344 = 5440$ con $T(1) = 1$.

Dimostrazione del caso 2. Poiché $\log^k n$ e $\log_b^k n$ differiscono per un fattore costante, l'ipotesi è $f(n) = \Theta(n^p \log_b^k n)$. Per l'Osservazione 16.2 esistono costanti $c_1, c_2 > 0$ tali che

$$c_1 m^p \log_b^k m \leq f(m) \leq c_2 m^p \log_b^k m \quad \text{per ogni potenza esatta } m \geq b.$$

Sia $L = \log_b n$. Per il termine j della somma, $m = n/b^j$ e $\log_b m = L - j$; per (16.3),

$$a^j \left(\frac{n}{b^j}\right)^p \log_b^k \frac{n}{b^j} = n^p (L - j)^k.$$

Quindi, ponendo $i = L - j$ (al variare di j da 0 a $L - 1$, i va da L a 1),

$$c_1 n^p S_k(L) \leq g(n) \leq c_2 n^p S_k(L), \quad \text{dove} \quad S_k(L) = \sum_{i=1}^L i^k.$$

Dimostriamo ora che $(\frac{L}{2})^{k+1} \leq S_k(L) \leq L^{k+1}$, cioè $S_k(L) = \Theta(L^{k+1})$.

- *Limitazione superiore:* ciascuno dei L termini soddisfa $i^k \leq L^k$ (poiché $k \geq 0$), quindi $S_k(L) \leq L \cdot L^k = L^{k+1}$.
- *Limitazione inferiore:* teniamo solo i termini con $i \geq \lceil L/2 \rceil$ (tutti i termini sono non negativi). Ce ne sono $L - \lceil L/2 \rceil + 1 \geq L/2$ e ciascuno è almeno $(L/2)^k$. Quindi $S_k(L) \geq \frac{L}{2} (\frac{L}{2})^k = (\frac{L}{2})^{k+1}$.

Pertanto

$$\frac{c_1}{2^{k+1}} n^p \log_b^{k+1} n \leq g(n) \leq c_2 n^p \log_b^{k+1} n,$$

cioè $g(n) = \Theta(n^p \log_b^{k+1} n) = \Theta(n^p \log^{k+1} n)$. □

Esempio 16.20: Caso 2 del Lemma 16.4: $2T(n/2) + n$ e $2T(n/2) + n \log_2 n$, $n = 16$

Qui $p = \log_2 2 = 1$ e $L = \log_2 16 = 4$.

- $f(n) = n$ ($k = 0$): ogni livello costa $2^j \cdot \frac{16}{2^j} = 16 = n^p$, quindi $g(16) = 4 \cdot 16 = 64 = n S_0(4)$, con $S_0(4) = 4 = L$.
- $f(n) = n \log_2 n$ ($k = 1$): il livello j costa $n^p(L - j) = 16(4 - j)$:

$$g(16) = 64 + 48 + 32 + 16 = 160 = 16(4 + 3 + 2 + 1) = n S_1(4).$$

Le limitazioni della dimostrazione danno $(\frac{L}{2})^2 = 4 \leq S_1(4) = 10 \leq L^2 = 16$, quindi $64 \leq g(16) \leq 256$.

Dimostrazione del caso 3. Limitazione inferiore. Tutti i termini di $g(n)$ sono positivi e il termine $j = 0$ è $f(n)$: quindi $g(n) \geq f(n)$.

Limitazione superiore. Sia $L = \log_b n$. Dimostriamo prima, per induzione su j , che

$$a^j f\left(\frac{n}{b^j}\right) \leq c^j f(n) \quad \text{per ogni } j \geq 0 \text{ tale che } \frac{n}{b^{j-1}} \geq n_0. \quad (16.4)$$

Per $j = 0$ è un'uguaglianza. Sia $j \geq 1$ con $n/b^{j-1} \geq n_0$. Allora anche $n/b^{j-2} \geq n_0$, quindi l'ipotesi induttiva vale per $j-1$; applicando la condizione di regolarità alla potenza esatta $n/b^{j-1} \geq n_0$,

$$a^j f\left(\frac{n}{b^j}\right) = a^{j-1} \left[a f\left(\frac{n/b^{j-1}}{b}\right) \right] \leq a^{j-1} c f\left(\frac{n}{b^{j-1}}\right) \leq c \cdot c^{j-1} f(n) = c^j f(n).$$

Dividiamo $g(n)$ nei termini che soddisfano $n/b^{j-1} \geq n_0$ (livelli *alti*) e negli altri (livelli *bassi*).

- Livelli alti: per (16.4) e la serie geometrica infinita di ragione $0 < c < 1$,

$$\sum_{\text{alti}} a^j f\left(\frac{n}{b^j}\right) \leq f(n) \sum_{j=0}^{\infty} c^j = \frac{f(n)}{1-c}.$$

- Livelli bassi: se $n/b^{j-1} < n_0$, allora $m = n/b^j < n_0$ è una potenza esatta di b con $m \geq b$, quindi $f(m) \leq M$, dove M è il massimo di f sulle (finite) potenze esatte di b in $[b, n_0]$. Inoltre $a^j \leq a^L = n^p$ (Lemma 16.3), e il numero di livelli bassi è al più $h = \lceil \log_b n_0 \rceil$ (corrispondono a $m = b, b^2, \dots$ con $m < n_0$). Quindi

$$\sum_{\text{bassi}} a^j f\left(\frac{n}{b^j}\right) \leq h M n^p.$$

Infine, $f(n) = \Omega(n^{p+\epsilon})$ implica $n^p = O(f(n))$ (infatti $n^p \leq n^{p+\epsilon}$). Pertanto $g(n) \leq \frac{f(n)}{1-c} + h M n^p = O(f(n))$, e insieme alla limitazione inferiore $g(n) = \Theta(f(n))$. $\square$

Esempio 16.21: Caso 3 del Lemma 16.4: $T(n) = 8T(n/3) + n^2$, $n = 27$

Qui $p = \log_3 8 \approx 1.893$, $c = \frac{8}{9}$ e la condizione di regolarità vale per ogni n (quindi $n_0 = 1$, e non ci sono livelli bassi). Il livello j costa $8^j \left(\frac{27}{3^j}\right)^2 = 729 \left(\frac{8}{9}\right)^j$:

$$g(27) = 729 + 648 + 576 = 1953, \quad f(27) = 729 \leq g(27) \leq \frac{f(27)}{1-8/9} = 6561.$$

I costi dei livelli decrescono geometricamente (sono dominati dalla radice). Con $T(1) = 1$: $T(27) = 8^3 + 1953 = 512 + 1953 = 2465$.

16.5.4 Dimostrazione per potenze esatte di b

Lemma 16.5: Master theorem per potenze esatte

I tre casi del Teorema 16.1 valgono per la ricorrenza (16.1), cioè quando n varia tra le potenze esatte di b .

Dimostrazione. Per il Lemma 16.3, $T(n) = \Theta(n^p) + g(n)$, con $g(n) > 0$. Usiamo il Lemma 16.4.

Caso 1: $g(n) = O(n^p)$, quindi $\Theta(n^p) \leq T(n) \leq \Theta(n^p) + O(n^p)$, cioè $T(n) = \Theta(n^p) = \Theta(n^{\log_b a})$.

Caso 2: $g(n) = \Theta(n^p \log^{k+1} n)$. Poiché $n^p = O(n^p \log^{k+1} n)$, le foglie sono trascurabili: $T(n) = \Theta(n^p \log^{k+1} n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

Caso 3: $g(n) = \Theta(f(n))$. Poiché $f(n) = \Omega(n^{p+\epsilon})$, abbiamo $n^p = O(f(n))$, quindi di nuovo le foglie sono trascurabili: $T(n) = \Theta(f(n))$. $\square$

16.5.5 Parti intere inferiori e superiori

- Consideriamo ora ogni numero naturale n . Trattiamo in dettaglio la ricorrenza con le parti intere superiori,

$$T(n) = aT(\lceil n/b \rceil) + f(n) \quad \text{per } n \geq n', \quad d_1 \leq T(n) \leq d_2 \quad \text{per } 1 \leq n < n', \quad (16.5)$$

dove n' e $0 < d_1 \leq d_2$ sono costanti, e $f(n) > 0$ per ogni $n \geq 1$. Nel caso 3 la condizione di regolarità si legge $a f(\lceil n/b \rceil) \leq c f(n)$ per ogni $n \geq n_0$.

- Aumentare la soglia.** I valori $T(m)$ per $1 \leq m < N$ sono un numero finito di numeri positivi per ogni costante $N \geq n'$. Quindi, sostituendo d_1 con il loro minimo e d_2 con il loro massimo, possiamo supporre che la ricorrenza sia usata solo per $n \geq N$, dove scegliamo

$$N \geq \max \left\{ n', n_0, \frac{2b-1}{b-1} \right\}.$$

Allora le ipotesi su f (e la condizione di regolarità) valgono per ogni argomento $\geq N$.

- Dimensioni dei sottoproblemi.** A profondità j dell'albero di ricorsione tutti i sottoproblemi hanno la stessa dimensione n_j , dove

$$n_0^* = n, \quad n_j = \lceil n_{j-1}/b \rceil \quad (j \geq 1)$$

(scriviamo n_0^* per la dimensione a profondità 0 per evitare confusione con la soglia n_0).

Lemma 16.6: Dimensioni con le parti intere superiori

Per ogni $j \geq 0$:

$$\frac{n}{b^j} \leq n_j < \frac{n}{b^j} + \frac{b}{b-1}.$$

Dimostrazione. Per induzione su j . Per $j = 0$, $n_0^* = n$. Per $j \geq 1$, usando $x \leq \lceil x \rceil < x+1$ e l'ipotesi induttiva,

$$\begin{aligned} n_j &\geq \frac{n_{j-1}}{b} \geq \frac{n}{b^j}, \\ n_j &< \frac{n_{j-1}}{b} + 1 < \frac{1}{b} \left(\frac{n}{b^{j-1}} + \frac{b}{b-1} \right) + 1 = \frac{n}{b^j} + \frac{1}{b-1} + 1 = \frac{n}{b^j} + \frac{b}{b-1}. \quad \square \end{aligned}$$

Lemma 16.7: Master theorem con le parti intere superiori

I tre casi del Teorema 16.1 valgono per la ricorrenza (16.5), per ogni numero naturale n .

Dimostrazione. Sia $n \geq N$ e $L = \log_b n$ (un numero reale, non necessariamente intero).

Passo 1: profondità dell'albero. Sia D il più piccolo j tale che $n_j < N$ (la profondità alla quale la ricorsione si ferma). Per $j \geq L$, il Lemma 16.6 dà $n_j < 1 + \frac{b}{b-1} = \frac{2b-1}{b-1} \leq N$; quindi $D \leq \lceil L \rceil < L+1$. Per $j \leq L - \log_b N$, $n_j \geq n/b^j \geq N$; quindi $D > L - \log_b N$.

Dunque

$$L - \log_b N < D < L + 1.$$

Passo 2: srotolamento. Esattamente come nella dimostrazione del Lemma 16.3 (la ricorrenza vale a ogni dimensione $n_j \geq N$, cioè per $j < D$),

$$T(n) = a^D T(n_D) + \sum_{j=0}^{D-1} a^j f(n_j).$$

Poiché $d_1 \leq T(n_D) \leq d_2$ e, per il Passo 1, $\frac{n^p}{N^p} = a^{L-\log_b N} \leq a^D \leq a^{L+1} = a n^p$, il primo termine è $\Theta(n^p)$.

Passo 3: limitazione di $g(n) = \sum_{j=0}^{D-1} a^j f(n_j)$. Sia $m_j = n/b^j$. Per $0 \leq j \leq D-1$ si ha $j < L$, quindi $m_j > 1$, e per il Lemma 16.6

$$m_j \leq n_j < m_j + \frac{b}{b-1} < C m_j, \quad C = \frac{2b-1}{b-1}, \quad (16.6)$$

mentre $n_j \geq N$, quindi le limitazioni su f valgono in n_j . Ricordiamo che $a^j m_j^p = n^p$ per (16.3).

Caso 1: $f(n_j) \leq c n_j^{p-\epsilon} \leq c C' m_j^{p-\epsilon}$ con $C' = \max\{1, C^{p-\epsilon}\}$ (se $p-\epsilon \geq 0$ si usa $n_j < C m_j$, altrimenti $n_j \geq m_j$). Come nel Lemma 16.4,

$$g(n) \leq c C' n^{p-\epsilon} \sum_{j=0}^{D-1} (b^\epsilon)^j \leq c C' n^{p-\epsilon} \frac{b^{\epsilon D}}{b^\epsilon - 1} \leq c C' n^{p-\epsilon} \frac{b^\epsilon n^\epsilon}{b^\epsilon - 1} = O(n^p).$$

Caso 2: Limitazione superiore: $\log_b n_j < \log_b C + \log_b m_j \leq \log_b C + L$, quindi

$$a^j f(n_j) \leq c_2 a^j C^p m_j^p (L + \log_b C)^k = c_2 C^p n^p (L + \log_b C)^k,$$

e sommando i $D < L + 1$ termini, $g(n) \leq c_2 C^p n^p (L + 1)(L + \log_b C)^k = O(n^p \log^{k+1} n)$.

Limitazione inferiore: $n_j \geq m_j > 1$ dà $a^j f(n_j) \geq c_1 a^j m_j^p \log_b^k m_j = c_1 n^p (L - j)^k$. Per n abbastanza grande $D - 1 \geq L/2$ (perché $D > L - \log_b N$), quindi tenendo solo i termini $0 \leq j \leq L/2$ (almeno $L/2$ termini, ciascuno $\geq (L/2)^k$),

$$g(n) \geq c_1 n^p \left(\frac{L}{2}\right)^{k+1} = \Omega(n^p \log^{k+1} n).$$

Caso 3: $g(n) \geq f(n)$ (termine $j = 0$). Per $1 \leq j \leq D-1$ si ha $n_{j-1} \geq N \geq n_0$, quindi la condizione di regolarità dà $a f(n_j) = a f(\lceil n_{j-1}/b \rceil) \leq c f(n_{j-1})$; per induzione (come in (16.4)) $a^j f(n_j) \leq c^j f(n)$, quindi $g(n) \leq \frac{f(n)}{1-c}$ e $g(n) = \Theta(f(n))$.

Passo 4: conclusione. $T(n) = \Theta(n^p) + g(n)$ con le stesse limitazioni su $g(n)$ del Lemma 16.4; i tre casi seguono esattamente come nella dimostrazione del Lemma 16.5. $\square$

• **Parti intere inferiori.** Per $T(n) = a T(\lfloor n/b \rfloor) + f(n)$ la dimostrazione è la stessa, con

$n_j = \lfloor n_{j-1}/b \rfloor$ e le limitazioni

$$\frac{n}{b^j} - \frac{b}{b-1} < n_j \leq \frac{n}{b^j}$$

(dimostrate come il Lemma 16.6, usando $x-1 < \lfloor x \rfloor \leq x$). Scegliendo anche $N \geq \max\{b, \frac{2b}{b-1}\}$, per $j < D$ si ottiene $m_j \geq n_j \geq N$, quindi $\frac{b}{b-1} \leq \frac{m_j}{2}$ e $\frac{m_j}{2} < n_j \leq m_j$, che sostituisce (16.6); inoltre $L - \log_b(N + \frac{b}{b-1}) \leq D \leq L - \log_b N + 1$. Cambiano solo le costanti.

- Questo completa la dimostrazione del Teorema 16.1. ■

Esempio 16.22: $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$, $T(1) = 1$

Questa è l'esatta ricorrenza del merge sort su un array di lunghezza qualunque n . Non ha la forma $aT(\lceil n/b \rceil) + f(n)$, ma è **compresa** tra due ricorrenze di quella forma:

$$U(n) = 2U(\lceil n/2 \rceil) + n, \quad W(n) = 2W(\lfloor n/2 \rfloor) + n, \quad U(1) = W(1) = 1.$$

- U e W sono non decrescenti (per induzione forte: se $n < n''$ allora $\lceil n/2 \rceil \leq \lceil n''/2 \rceil$ e il termine aggiunto n è minore).
- Per induzione forte, $W(n) \leq T(n) \leq U(n)$: per $n \geq 2$, usando che U è non decrescente e $\lfloor n/2 \rfloor \leq \lceil n/2 \rceil$,

$$T(n) \leq U(\lceil n/2 \rceil) + U(\lfloor n/2 \rfloor) + n \leq 2U(\lceil n/2 \rceil) + n = U(n),$$

e simmetricamente $T(n) \geq 2W(\lfloor n/2 \rfloor) + n = W(n)$.

- Per il Lemma 16.7 (caso 2, $k = 0$) e la sua versione con le parti intere inferiori, $U(n) = \Theta(n \log n)$ e $W(n) = \Theta(n \log n)$. Quindi $T(n) = \Theta(n \log n)$ per **ogni** n , non solo per le potenze di 2.

Numericamente: $T(1000) = 10976$ e $T(10^6) = 20951424$, con $\frac{T(n)}{n \log_2 n} \approx 1.101$ e ≈ 1.051 , rispettivamente.

16.5.6 La condizione di regolarità

- Nel caso 3 l'ipotesi $f(n) = \Omega(n^{p+\epsilon})$ è in realtà implicata dalla condizione di regolarità (per f positiva). Questo spiega perché, in pratica, basta controllare la condizione di regolarità.

Osservazione 16.8: la regolarità implica la crescita polinomiale

Siano $a \geq 1$, $b > 1$, $p = \log_b a$, e sia $f(m) > 0$ per ogni potenza esatta m di b . Se $a f(n/b) \leq c f(n)$ per qualche costante $0 < c < 1$ e per tutte le potenze esatte $n \geq n_0$ di b , allora

$$f(n) = \Omega(n^{p+\epsilon}) \quad \text{con} \quad \epsilon = \log_b \frac{a}{c} - \log_b a = -\log_b c > 0.$$

Dimostrazione. Poiché $0 < c < 1$, $\epsilon = -\log_b c > 0$ e $\frac{a}{c} > a \geq 1$. La condizione di regolarità si può riscrivere come

$$f(n) \geq \frac{a}{c} f\left(\frac{n}{b}\right) \quad \text{per ogni potenza esatta } n \geq n_0.$$

Sia $n \geq n_0$ una potenza esatta di b , e sia $j \geq 0$ il numero di volte in cui possiamo applicare questa disuguaglianza, cioè il più piccolo j tale che $m = n/b^j$ soddisfa $m < bn_0$ (quindi $n_0 \leq m < bn_0$ quando $n \geq n_0$; ogni dimensione intermedia n/b^i , $i < j$, è $\geq bn_0 \geq n_0$). Applicando la disuguaglianza j volte,

$$f(n) \geq \left(\frac{a}{c}\right)^j f\left(\frac{n}{b^j}\right) \geq \left(\frac{a}{c}\right)^j \mu,$$

dove $\mu > 0$ è il minimo di f sulle (finite) potenze esatte di b in $[n_0, bn_0)$. Poiché $b^j = n/m > \frac{n}{bn_0}$ e $\frac{a}{c} = b^{\log_b(a/c)}$ con $\log_b(a/c) > 0$,

$$\left(\frac{a}{c}\right)^j = (b^j)^{\log_b(a/c)} > \left(\frac{n}{bn_0}\right)^{\log_b(a/c)}.$$

Infine $\log_b(a/c) = \log_b a - \log_b c = p + \epsilon$, quindi

$$f(n) \geq \frac{\mu}{(bn_0)^{p+\epsilon}} n^{p+\epsilon} \quad \text{per ogni potenza esatta } n \geq n_0,$$

cioè $f(n) = \Omega(n^{p+\epsilon})$. □

- Per esempio, nell'Esempio 16.3 ($8T(n/3) + n^2$, $c = 8/9$) l'osservazione dà $\epsilon = -\log_3 \frac{8}{9} = 2 - \log_3 8 \approx 0.107$, esattamente il massimo ϵ trovato lì.
- Il viceversa è falso: nell'Esempio 16.17, $f(n) = \Omega(n^{p+1})$ ma la condizione di regolarità non vale.

16.6 Riepilogo

Come risolvere una ricorrenza $T(n)$.

1. È in forma standard $T(n) = aT(n/b) + f(n)$ con costanti $a \geq 1$, $b > 1$ e $f(n)$ asintoticamente positiva?
 - **No** $\rightarrow$ si usi un altro metodo: iterazione, albero di ricorsione, sostituzione.
 - **Sì** $\rightarrow$ si calcoli $n^{\log_b a}$ e lo si confronti con $f(n)$.
2. $f(n) = O(n^{\log_b a - \epsilon})$ per qualche $\epsilon > 0$ (polinomialmente più piccola)?
 $\rightarrow$ **caso 1:** $T(n) = \Theta(n^{\log_b a})$.
3. $f(n) = \Theta(n^{\log_b a} \log^k n)$ per qualche $k \geq 0$?
 $\rightarrow$ **caso 2:** $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
4. $f(n) = \Omega(n^{\log_b a + \epsilon})$ per qualche $\epsilon > 0$ (polinomialmente più grande)? Si controlli la condizione di regolarità $a f(n/b) \leq c f(n)$ con $c < 1$:

- vale $\rightarrow$ **caso 3**: $T(n) = \Theta(f(n))$;
- non vale $\rightarrow$ il teorema non si applica.

5. Nessuno dei precedenti (un **vuoto**, per esempio $f(n) = n^{\log_b a} / \log n \rightarrow$ il teorema non si applica: si usi l'albero di ricorsione $T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$ (Lemma 16.3).

Base dei logaritmi: irrilevante in $\log^k n$, essenziale in $n^{\log_b a}$.

Algoritmi ricorsivi

Introduzione

Nei capitoli «Ricorsione», «Equazioni di ricorrenza» e «Il Master theorem» abbiamo imparato come viene eseguito un algoritmo ricorsivo, come se ne descrive il numero di passi con un'equazione di ricorrenza e come si risolve tale equazione. In questo capitolo mettiamo al lavoro questi strumenti su quattro problemi classici sugli array:

- la **somma** degli elementi di un array, il problema più semplice possibile, su cui confrontiamo un algoritmo iterativo e due algoritmi ricorsivi;
- l'**ordinamento**, con il **Merge Sort**, il primo algoritmo del corso che batte gli algoritmi di ordinamento $\Theta(n^2)$;
- la **ricerca** in un array ordinato, con la **ricerca binaria ricorsiva**;
- il problema del **sotto-array di somma massima**, in cui un algoritmo divide et impera migliora la forza bruta, e un algoritmo iterativo ingegnoso migliora entrambi.

Tutti gli algoritmi sono studiati con lo stesso schema, che è lo schema dell'analisi di ogni algoritmo ricorsivo.

Per ogni algoritmo ricorsivo:

1. scriviamo lo pseudocodice e lo eseguiamo su un piccolo esempio, disegnando l'**albero delle chiamate**;
2. dimostriamo che è **corretto**, per induzione (forte) sulla dimensione del sottoproblema;
3. descriviamo il suo numero di passi con un'**equazione di ricorrenza** e la risolviamo, con l'albero di ricorsione e con il Master theorem;
4. esprimiamo il **tempo di esecuzione** nel caso peggiore in funzione della dimensione $|I|$ dell'input;
5. misuriamo la **memoria aggiuntiva**, che per un algoritmo ricorsivo comprende la pila delle chiamate;
6. diamo un'implementazione in C++ (array indicizzati da 0, mentre nello pseudocodice sono indicizzati da 1).

Tre dei quattro algoritmi seguono il paradigma **divide et impera**: il problema su un sotto-array $a[l..r]$ viene suddiviso in sottoproblemi sulle due metà $a[l..m]$ e $a[m + 1..r]$ (*divide*), i sottoproblemi vengono risolti ricorsivamente (*impera*), e le loro soluzioni vengono messe insieme (*combina*). La ricerca binaria ricorsiva, invece, effettua una sola chiamata ricorsiva su una metà.

17.1 La somma degli elementi di un array

17.1.1 Il problema

Dato un array $a[1..n]$ di $n \geq 1$ numeri interi, consideriamo il problema di calcolare la somma dei suoi elementi:

$$\sum_{i=1}^n a[i] = a[1] + a[2] + \cdots + a[n].$$

- Questo è uno dei problemi più semplici sugli array, ed è il problema ideale per confrontare un algoritmo **iterativo** e un algoritmo **ricorsivo** che risolvono lo stesso problema.
- Scriviamo l'algoritmo ricorsivo, dimostriamo che è corretto, descriviamo il suo numero di passi con un'equazione di ricorrenza e risolviamo l'equazione, usando gli strumenti dei capitoli «Ricorsione», «Equazioni di ricorrenza» e «Il Master theorem».

Esempio 17.1: somma di un array

Consideriamo il seguente array di $n = 7$ numeri interi:

$a[i]$	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$
	9	8	12	3	5	14	6

La somma dei suoi elementi è

$$9 + 8 + 12 + 3 + 5 + 14 + 6 = 57.$$

17.1.2 Algoritmo iterativo

- Partiamo dall'ovvio algoritmo iterativo **SumIterative**, che scorre l'array da sinistra a destra e accumula le somme parziali nella variabile s :

Algorithm: SumIterative**Input** : Un array $a[1..n]$ di n numeri interi**Output**: La somma $a[1] + a[2] + \dots + a[n]$

```

1 function SumIterative( $a, n$ )
2    $s := 0$ 
3    $i := 1$ 
4   while  $i \leq n$  do
5      $s \leftarrow s + a[i]$ 
6      $i \leftarrow i + 1$ 
7   end
8   return  $s$ 
9 end

```

- Sia $S(n)$ il numero di passi eseguiti da **SumIterative** su un array di n elementi (non c'è differenza tra caso migliore e caso peggiore: l'algoritmo esegue sempre le stesse operazioni). Il numero di volte che ciascuna riga di codice viene eseguita è:

- La **riga** 2 viene eseguita 1 volta
- La **riga** 3 viene eseguita 1 volta
- La **riga** 4 viene eseguita $n + 1$ volte
- La **riga** 5 viene eseguita n volte
- La **riga** 6 viene eseguita n volte
- La **riga** 8 viene eseguita 1 volta

Quindi,

$$S(n) = 1 + 1 + (n + 1) + n + n + 1 = 3n + 4 = \Theta(n),$$

$$\text{poiché} \quad \lim_{n \rightarrow \infty} \frac{3n + 4}{n} = \lim_{n \rightarrow \infty} \left(3 + \frac{4}{n} \right) = 3 \in \mathbb{R}_{>0}.$$

- Oltre all'array in ingresso, l'algoritmo usa solo le due variabili s e i : la **memoria aggiuntiva** è $\Theta(1)$.

17.1.3 Algoritmo ricorsivo (divide et impera)

- La stessa somma si può calcolare ricorsivamente. La somma degli elementi di un sotto-array $a[l..r]$ si ottiene con il metodo **divide et impera**:
 1. **Divide**: si suddivide $a[l..r]$ nelle due metà $a[l..m]$ e $a[m+1..r]$, dove $m = \lfloor (l+r)/2 \rfloor$;
 2. **Impera**: si calcolano ricorsivamente la somma s_1 di $a[l..m]$ e la somma s_2 di $a[m+1..r]$;
 3. **Combina**: si restituisce $s_1 + s_2$.
- Il **caso base** è un sotto-array con un solo elemento ($l = r$): la sua somma è l'elemento $a[l]$ stesso.
- Questo è lo pseudocodice dell'algoritmo ricorsivo **RecursiveSum**. Per sommare l'intero array chiamiamo **RecursiveSum**($a, 1, n$).

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  $a[l]$                                 // Caso base
4   end
5    $m := \lfloor (l+r)/2 \rfloor$                           // Divide
6    $s_1 := \text{RecursiveSum}(a, l, m)$                 // Impera (metà sinistra)
7    $s_2 := \text{RecursiveSum}(a, m+1, r)$               // Impera (metà destra)
8   return  $s_1 + s_2$                                 // Combina
9 end

```

- Sia $k = r - l + 1$ il numero di elementi di $a[l..r]$. Se $k \geq 2$, le due metà hanno

$$m - l + 1 = \left\lceil \frac{k}{2} \right\rceil \quad \text{e} \quad r - m = \left\lfloor \frac{k}{2} \right\rfloor$$

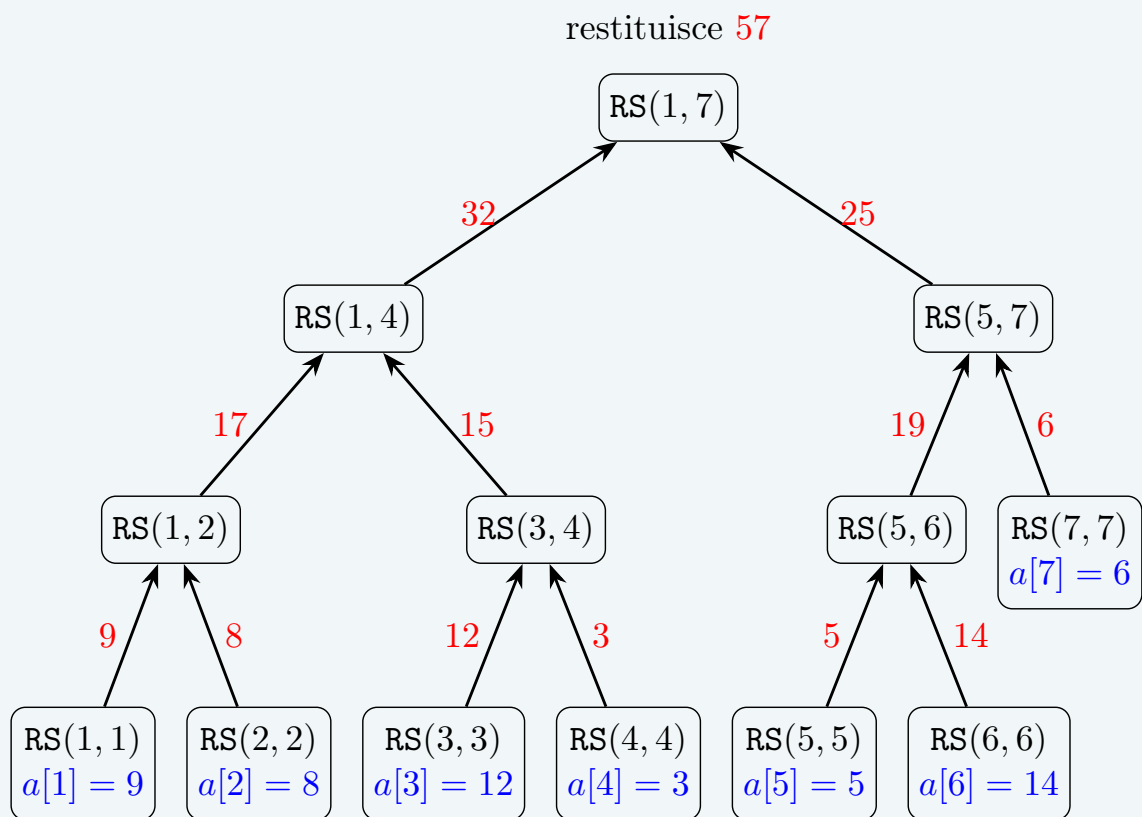
elementi, rispettivamente. Entrambi i numeri sono almeno 1 e strettamente minori di k : ogni chiamata ricorsiva lavora su un sotto-array più piccolo, e la ricorsione raggiunge sempre il caso base.

17.1.4 Albero delle chiamate

- L'esecuzione di un algoritmo ricorsivo è rappresentata dal suo **albero delle chiamate**: ogni nodo è una chiamata, e i figli di un nodo sono le chiamate che esso effettua. Il valore restituito da una chiamata viene rimandato al padre.

Esempio 17.2: albero delle chiamate di RecursiveSum

Consideriamo l'array dell'Esempio 17.1, cioè $a = [9, 8, 12, 3, 5, 14, 6]$ e $n = 7$. Questo è l'albero delle chiamate di `RecursiveSum(a, 1, 7)`, in cui ogni nodo $RS(l, r)$ è una chiamata sul sotto-array $a[l..r]$. In **blu** gli elementi dell'array restituiti dai casi base, in **rosso** i valori restituiti ai padri dalle chiamate (lungo le frecce):



- La radice $RS(1, 7)$ calcola $m = \lfloor 8/2 \rfloor = 4$ e chiama $RS(1, 4)$ e $RS(5, 7)$.
- $RS(1, 4)$ calcola $m = 2$ e restituisce $17 + 15 = 32$; $RS(5, 7)$ calcola $m = 6$ e restituisce $19 + 6 = 25$.
- La radice restituisce $32 + 25 = 57$, che è la somma degli elementi dell'array.
- Ci sono $13 = 2 \cdot 7 - 1$ chiamate: 7 foglie (casi base, una per ogni elemento) e 6 nodi interni, ciascuno dei quali esegue un'addizione, quindi in totale ci sono $6 = n - 1$ addizioni.

- Le chiamate vengono eseguite in ordine di **visita in profondità**: una chiamata termina solo dopo che sono terminati entrambi i suoi figli. Nell'esempio le chiamate terminano nell'ordine

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

17.1.5 Correttezza

- La correttezza di un algoritmo ricorsivo si dimostra per **induzione forte** sulla dimensione del sottoproblema: si assume che le chiamate ricorsive, che lavorano su sotto-array più piccoli, siano corrette, e si mostra che la chiamata combina correttamente i loro risultati.

Osservazione 17.1: correttezza di RecursiveSum

Per ogni coppia di indici $l \leq r$, la chiamata `RecursiveSum(a, l, r)` termina e restituisce

$$\sum_{i=l}^r a[i].$$

Dimostrazione. Usiamo l'induzione forte sul numero $k = r - l + 1 \geq 1$ di elementi del sotto-array.

Caso base ($k = 1$). Allora $l = r$, la condizione della riga 2 è vera e l'algoritmo restituisce $a[l] = \sum_{i=l}^l a[i]$.

Passo induttivo ($k \geq 2$). Supponiamo che l'enunciato valga per tutti i sotto-array con meno di k elementi (*ipotesi induttiva*). Poiché $l < r$, l'algoritmo calcola $m = \lfloor (l + r)/2 \rfloor$, con $l \leq m < r$. I sotto-array $a[l..m]$ 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, le due chiamate ricorsive terminano e

$$s_1 = \sum_{i=l}^m a[i], \quad s_2 = \sum_{i=m+1}^r a[i].$$

Quindi, l'algoritmo restituisce

$$s_1 + s_2 = \sum_{i=l}^m a[i] + \sum_{i=m+1}^r a[i] = \sum_{i=l}^r a[i].$$

□

- Con $l = 1$ e $r = n$, la chiamata `RecursiveSum(a, 1, n)` restituisce la somma di tutti gli elementi dell'array.

17.1.6 Numero di passi di RecursiveSum

17.1.6.1 L'equazione di ricorrenza

- Sia $S(n)$ il numero di passi eseguiti da **RecursiveSum** su un sotto-array di $n = r - l + 1$ elementi. Come per l'algoritmo iterativo, il numero di passi dipende solo da n (e non dai valori degli elementi).
- Contiamo le righe di codice eseguite **da una singola chiamata**. Una riga che contiene una chiamata ricorsiva conta 1 passo (la chiamata stessa) **più** tutti i passi eseguiti dalla funzione chiamata, che esprimiamo ancora con S .
- Se $n = 1$, le **righe 2 e 3** vengono eseguite 1 volta ciascuna, e quindi $S(1) = 2$.
- If $n \geq 2$:
 - La **riga 2** viene eseguita 1 volta
 - La **riga 3** non viene eseguita
 - La **riga 5** viene eseguita 1 volta
 - La **riga 6** viene eseguita 1 volta, più i $S(\lceil n/2 \rceil)$ passi della chiamata su $a[l..m]$
 - La **riga 7** viene eseguita 1 volta, più i $S(\lfloor n/2 \rfloor)$ passi della chiamata su $a[m+1..r]$
 - La **riga 8** viene eseguita 1 volta

e quindi $S(n) = S(\lceil n/2 \rceil) + S(\lfloor n/2 \rfloor) + 5$.

Il numero di passi di **RecursiveSum** è descritto dall'equazione di ricorrenza

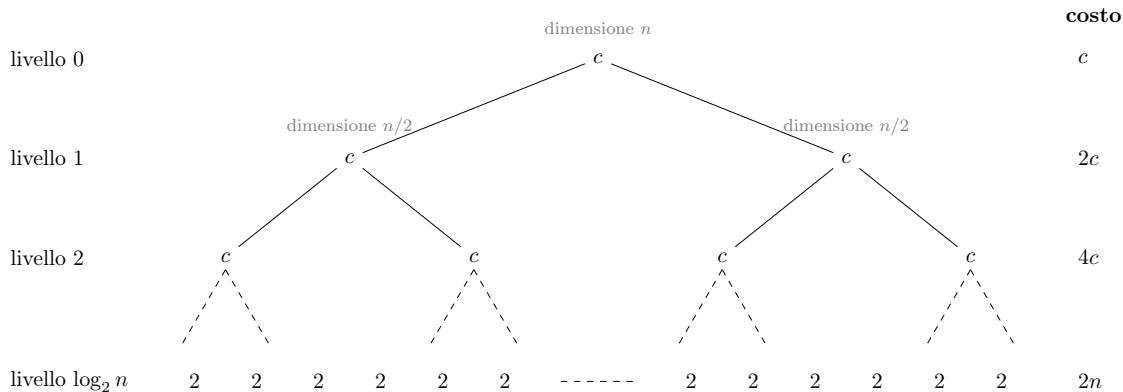
$$S(n) = \begin{cases} 2 & \text{se } n = 1, \\ S(\lceil n/2 \rceil) + S(\lfloor n/2 \rfloor) + c & \text{se } n \geq 2, \end{cases}$$

con $c = 5$. Come di consueto, nell'analisi asintotica la scriviamo semplicemente come

$$S(n) = 2 S(n/2) + c.$$

17.1.6.2 Soluzione con l'albero di ricorsione

- Nell'**albero di ricorsione** ogni nodo rappresenta una chiamata e contiene il numero di passi eseguiti da quella chiamata *esclusi* i passi delle sue chiamate ricorsive: c per un nodo interno, 2 per una foglia. $S(n)$ è la somma dei valori di tutti i nodi.
- Per semplicità, supponiamo dapprima che $n = 2^h$ sia una potenza esatta di 2. Allora ogni sotto-array viene suddiviso in due metà di uguale dimensione, e l'albero è:



- Il livello i contiene 2^i chiamate su sotto-array di $n/2^i$ elementi. Le foglie sono al livello $\log_2 n$, dove i sotto-array hanno $n/2^{\log_2 n} = 1$ elemento: ci sono $2^{\log_2 n} = n$ foglie.
- I livelli interni $0, 1, \dots, \log_2 n - 1$ contengono

$$\sum_{i=0}^{\log_2 n - 1} 2^i = 2^{\log_2 n} - 1 = n - 1$$

nodi (somma di una serie geometrica). Quindi il numero totale di passi è

$$S(n) = c(n - 1) + 2n = 5(n - 1) + 2n = 7n - 5.$$

- Il costo dei livelli **raddoppia** da un livello al successivo: il totale è dominato dalle foglie, che sono n . Questa è esattamente la situazione del caso 1 del Master theorem.
- Lo stesso conteggio vale per **ogni** n , non solo per le potenze di 2: il fatto chiave è che l'albero ha esattamente n foglie e $n - 1$ nodi interni, come mostra la prossima osservazione.

Osservazione 17.2: chiamate e passi di RecursiveSum

Per ogni $n \geq 1$, la chiamata `RecursiveSum(a, l, r)` su un sotto-array di $n = r - l + 1$ elementi genera esattamente

$$2n - 1$$

chiamate (compresa la prima): n di esse sono casi base e $n - 1$ eseguono un'addizione. Inoltre, $S(n) = 7n - 5$.

Dimostrazione. Sia $C(n)$ il numero di chiamate. Abbiamo $C(1) = 1$ e, per $n \geq 2$,

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

Dimostriamo che $C(n) = 2n - 1$ per induzione forte su n . Per $n = 1$: $C(1) = 1 = 2 \cdot 1 - 1$. Per $n \geq 2$, per l'ipotesi induttiva applicata a $\lceil n/2 \rceil < n$ e $\lfloor n/2 \rfloor < n$,

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

perché $\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$ per ogni numero naturale n .

Ogni caso base corrisponde esattamente a un elemento del sotto-array (i sotto-array delle foglie sono disgiunti e ricoprono $a[l..r]$), quindi ci sono n casi base e $2n - 1 - n = n - 1$ chiamate che eseguono la riga 8.

Allo stesso modo, $S(1) = 2 = 7 \cdot 1 - 5$ e, per $n \geq 2$,

$$S(n) = (7\lceil n/2 \rceil - 5) + (7\lfloor n/2 \rfloor - 5) + 5 = 7n - 5.$$

□

- Di conseguenza,

$$S(n) = 7n - 5 = \Theta(n), \quad \text{poiché} \quad \lim_{n \rightarrow \infty} \frac{7n - 5}{n} = 7 \in \mathbb{R}_{>0}.$$

Per l'array dell'Esempio 17.2, $S(7) = 44$: 7 foglie con 2 passi ciascuna e 6 nodi interni con 5 passi ciascuno.

17.1.6.3 Soluzione con il Master theorem

- Possiamo ottenere l'ordine di crescita di $S(n)$ direttamente dal Master theorem, senza calcolare $S(n)$ esattamente. Ricordiamo il caso 1 del Master theorem:

Sia $S(n) = a S(n/b) + f(n)$, con $a \geq 1$ e $b > 1$ costanti e $f(n)$ asintoticamente positiva. Se $f(n) = O(n^{\log_b a - \varepsilon})$ per una qualche costante $\varepsilon > 0$, allora

$$S(n) = \Theta(n^{\log_b a}).$$

- $S(n) = 2 S(n/2) + c \rightarrow a = 2, b = 2, f(n) = c.$
- $n^{\log_b a} = n^{\log_2 2} = n^1 = n.$
- Confronto: $f(n) = c$ cresce polinomialmente più lentamente di $n \rightarrow$ caso 1.
- Ipotesi: $f(n) = c = O(n^{1-\varepsilon})$ con $\varepsilon = 1$, poiché $c = c \cdot n^0$.
- Conclusione: $S(n) = \Theta(n^{\log_2 2}) = \Theta(n).$

Come previsto, il risultato è in accordo con la soluzione esatta $S(n) = 7n - 5$.

17.1.6.4 Tempo di esecuzione nel caso peggiore

- Come nel capitolo «Tempo di esecuzione», facciamo l'ipotesi che ogni elemento dell'array sia memorizzato usando un numero costante fissato di bit. Allora $|I| = \Theta(n)$, cioè esistono costanti $c_1, c_2 > 0$ tali che $c_1 n \leq |I| \leq c_2 n$ per ogni n sufficientemente grande, e

$$\frac{1}{c_2} |I| \leq n \leq \frac{1}{c_1} |I|.$$

Poiché $S(n) = 7n - 5 = \Theta(n)$, esistono costanti $a_1, a_2 > 0$ tali che $a_1 n \leq S(n) \leq a_2 n$ per ogni n sufficientemente grande, e quindi

$$\frac{a_1}{c_2} |I| \leq S(n) \leq \frac{a_2}{c_1} |I|.$$

Se tutti i valori in ingresso usano un numero costante fissato di bit, allora il tempo di esecuzione nel caso peggiore dell'algoritmo **RecursiveSum** (e anche dell'algoritmo **SumIterative**) è

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

Quindi, entrambi gli algoritmi sono lineari nella dimensione dell'input.

- La ricorsione non rende l'algoritmo più veloce: entrambi gli algoritmi leggono ogni elemento una volta ed eseguono $n - 1$ (ricorsivo) oppure n (iterativo) addizioni. La versione ricorsiva esegue anzi più passi ($7n - 5$ contro $3n + 4$), a causa del sovraccarico delle chiamate.

17.1.7 Memoria aggiuntiva: la pila delle chiamate

- Il tempo non è l'unica risorsa. Ogni chiamata attiva di una funzione ricorsiva richiede un proprio **record di attivazione** nella **pila delle chiamate** (*call stack*), che contiene i suoi parametri, le sue variabili locali e l'indirizzo di ritorno. In **RecursiveSum** i parametri e le variabili locali sono l, r, m, s_1, s_2 : ogni record di attivazione ha dimensione costante.
- Una chiamata è attiva dal momento in cui inizia fino a quando termina. Poiché le chiamate vengono eseguite in profondità, le chiamate attive in ogni istante formano un cammino dalla radice dell'albero delle chiamate al nodo corrente. Quindi:

Il massimo numero di chiamate contemporaneamente attive (la **profondità della ricorsione**) è uguale all'altezza dell'albero delle chiamate più uno. La memoria aggiuntiva usata da un algoritmo ricorsivo è (almeno) proporzionale alla profondità della ricorsione.

Osservazione 17.3: profondità della ricorsione di RecursiveSum

Per ogni $n \geq 1$, l'albero delle chiamate di **RecursiveSum** su un sotto-array di n elementi ha altezza $\lceil \log_2 n \rceil$. Quindi la profondità della ricorsione è $\lceil \log_2 n \rceil + 1$ e la memoria aggiuntiva usata da **RecursiveSum** è $\Theta(\log_2 n)$.

Dimostrazione. Sia $h(n)$ l'altezza dell'albero delle chiamate. Allora $h(1) = 0$ e, per $n \geq 2$,

$$h(n) = 1 + \max \{h(\lceil n/2 \rceil), h(\lfloor n/2 \rfloor)\}.$$

Dimostriamo che $h(n) = \lceil \log_2 n \rceil$ per induzione forte. Per $n = 1$, $\lceil \log_2 1 \rceil = 0$. Sia $n \geq 2$. Poiché $\lceil \log_2 x \rceil$ è non decrescente in x , per l'ipotesi induttiva il massimo è raggiunto dalla metà più grande, e $h(n) = 1 + \lceil \log_2 \lceil n/2 \rceil \rceil$.

Sia $k = \lceil \log_2 n \rceil \geq 1$, cioè $2^{k-1} < n \leq 2^k$. Dividendo per 2, otteniamo $2^{k-2} < n/2 \leq 2^{k-1}$; poiché 2^{k-1} è un intero, anche $2^{k-2} < \lceil n/2 \rceil \leq 2^{k-1}$, cioè $\lceil \log_2 \lceil n/2 \rceil \rceil = k - 1$. Quindi $h(n) = 1 + (k - 1) = k = \lceil \log_2 n \rceil$.

Ogni record di attivazione ha dimensione costante, quindi la memoria aggiuntiva è $\Theta(\lceil \log_2 n \rceil + 1) = \Theta(\log_2 n)$. $\square$

- Nell'Esempio 17.2, $n = 7$ e la profondità della ricorsione è $\lceil \log_2 7 \rceil + 1 = 3 + 1 = 4$: per esempio, quando viene eseguita $\text{RS}(1, 1)$, le chiamate attive sono $\text{RS}(1, 7)$, $\text{RS}(1, 4)$, $\text{RS}(1, 2)$ e $\text{RS}(1, 1)$.

algoritmo	numero di passi	memoria aggiuntiva
SumIterative	$3n + 4 = \Theta(n)$	$\Theta(1)$
RecursiveSum	$7n - 5 = \Theta(n)$	$\Theta(\log_2 n)$

17.1.8 Una ricorsione diversa: decremento di uno

- Dimezzare non è l'unico modo di sommare un array ricorsivamente. L'array può anche essere suddiviso in modo **sbilanciato**: la somma di $a[1..n]$ è la somma di $a[1..n-1]$ più $a[n]$. Questa è una ricorsione per *decremento di uno*:

Algorithm: RecursiveSumLinear

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

Output: La somma $a[1] + a[2] + \dots + a[n]$

```

1 function RecursiveSumLinear(a, n)
2   if n = 1 then
3     return a[1]                                // Caso base
4   end
5   s := RecursiveSumLinear(a, n - 1)
6   return s + a[n]
7 end

```

- La correttezza segue per induzione (semplice) su n : per $n = 1$ l'algoritmo restituisce $a[1]$; per $n \geq 2$, per l'ipotesi induttiva $s = a[1] + \dots + a[n-1]$, e l'algoritmo restituisce $s + a[n] = a[1] + \dots + a[n]$.
- Se $n = 1$, le righe 2 e 3 vengono eseguite una volta: $S(1) = 2$. Se $n \geq 2$, le righe 2, 5 e 6 vengono eseguite una volta, più i $S(n-1)$ passi della chiamata ricorsiva:

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

- Il Master theorem **non** si può applicare, perché la dimensione del sottoproblema è $n-1$ e non n/b per una costante $b > 1$. Risolviamo l'equazione per iterazione (srotolamento):

$$S(n) = S(n-1) + 3 = S(n-2) + 2 \cdot 3 = \dots = S(1) + (n-1) \cdot 3 = 3n - 1 = \Theta(n).$$

- L'albero delle chiamate è un **cammino** di n nodi: $\text{RecursiveSumLinear}(a, n)$ chiama $\text{RecursiveSumLinear}(a, n-1)$, che chiama $\text{RecursiveSumLinear}(a, n-2)$, e così via, fino a $n = 1$. Quando si raggiunge il caso base, tutte le n chiamate sono attive.

`RecursiveSumLinear` esegue $S(n) = 3n - 1 = \Theta(n)$ passi, ma la sua profondità di ricorsione è n : richiede $\Theta(n)$ di memoria aggiuntiva per la pila delle chiamate (contro $\Theta(\log_2 n)$ di `RecursiveSum` e $\Theta(1)$ di `SumIterative`). In pratica, per n grande (ad esempio $n = 10^7$) la versione per decremento di uno può esaurire la pila delle chiamate (*stack overflow*), mentre `RecursiveSum` richiede solo $\lceil \log_2 10^7 \rceil + 1 = 25$ chiamate attive.

17.1.9 Implementazione in C++

- Questa è un'implementazione in C++ dell'algoritmo iterativo e di quello ricorsivo. In C++ gli array sono indicizzati da 0: la somma dell'intero array si calcola con la chiamata `RecursiveSum(a, 0, n-1)`.

```

1 // Somma iterativa di a[0..n-1]
2 int SumIterative(int n, int *a)
3 {
4     int s = 0;
5     for (int i = 0; i < n; i++)
6     {
7         s = s + a[i];
8     }
9     return s;
10 }
11
12 // Somma ricorsiva di a[l..r] (indici da 0, l <= r)
13 int RecursiveSum(int *a, int l, int r)
14 {
15     if (l == r)
16     {
17         return a[l]; // Caso base: un solo elemento
18     }
19
20     int m = l + (r - l) / 2; // Equivale a (l + r) / 2
21
22     int s1 = RecursiveSum(a, l, m); // Somma della metà sinistra
23     int s2 = RecursiveSum(a, m + 1, r); // Somma della metà destra
24
25     return s1 + s2;
26 }

```

- L'espressione $l + (r - l) / 2$ calcola $\lfloor (l+r)/2 \rfloor$ (per $l, r \geq 0$) senza il rischio di overflow della somma $l + r$, come nell'algoritmo di ricerca binaria.

Esempio 17.3: esecuzione dell'implementazione in C++ di `RecursiveSum`

Eseguiamo la chiamata `RecursiveSum(a, 0, 4)` sull'array $a = [9, 8, 12, 3, 5]$ ($n = 5$):

$a[j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$l = 0$	$r = 4$
	9	8	12	3	5		

La colonna *pila* riporta le chiamate attive, dalla prima a quella corrente (scriviamo (l, r) per `RecursiveSum(a, l, r)`).

#	pila	azione
1	(0, 4)	$l \neq r, m = 0 + (4 - 0)/2 = 2$; chiamata (0, 2)
2	(0, 4), (0, 2)	$l \neq r, m = 0 + (2 - 0)/2 = 1$; chiamata (0, 1)
3	(0, 4), (0, 2), (0, 1)	$l \neq r, m = 0 + (1 - 0)/2 = 0$; chiamata (0, 0)
4	(0, 4), (0, 2), (0, 1), (0, 0)	$l = r \Rightarrow$ restituisce $a[0] = 9$
5	(0, 4), (0, 2), (0, 1)	$s1 = 9$; chiamata (1, 1)
6	(0, 4), (0, 2), (0, 1), (1, 1)	$l = r \Rightarrow$ restituisce $a[1] = 8$
7	(0, 4), (0, 2), (0, 1)	$s2 = 8$; restituisce $9 + 8 = 17$
8	(0, 4), (0, 2)	$s1 = 17$; chiamata (2, 2)
9	(0, 4), (0, 2), (2, 2)	$l = r \Rightarrow$ restituisce $a[2] = 12$
10	(0, 4), (0, 2)	$s2 = 12$; restituisce $17 + 12 = 29$
11	(0, 4)	$s1 = 29$; chiamata (3, 4)
12	(0, 4), (3, 4)	$l \neq r, m = 3 + (4 - 3)/2 = 3$; chiamata (3, 3)
13	(0, 4), (3, 4), (3, 3)	$l = r \Rightarrow$ restituisce $a[3] = 3$
14	(0, 4), (3, 4)	$s1 = 3$; chiamata (4, 4)
15	(0, 4), (3, 4), (4, 4)	$l = r \Rightarrow$ restituisce $a[4] = 5$
16	(0, 4), (3, 4)	$s2 = 5$; restituisce $3 + 5 = 8$
17	(0, 4)	$s2 = 8$; restituisce $29 + 8 = 37$

L'algoritmo termina con risultato $37 = 9 + 8 + 12 + 3 + 5$. Ci sono $9 = 2 \cdot 5 - 1$ chiamate, e il massimo numero di chiamate attive è $4 = \lceil \log_2 5 \rceil + 1$ (righe 4 e 6).

Punti chiave sulla somma di un array.

- **RecursiveSum** è corretto (induzione forte), effettua $2n - 1$ chiamate ed esegue $7n - 5 = \Theta(n)$ passi: la ricorsione non riduce il lavoro, che è lineare perché ogni elemento deve essere letto.
- L'equazione di ricorrenza $S(n) = 2S(n/2) + c$ ricade nel caso 1 del Master theorem: il costo è dominato dalle n foglie.
- La pila delle chiamate costa memoria: $\Theta(\log_2 n)$ per la ricorsione per dimezzamento, $\Theta(n)$ per la ricorsione per decremento di uno, $\Theta(1)$ per il ciclo.

17.2 Merge Sort

Per la somma di un array, il divide et impera non ha dato alcun guadagno in termini di tempo. Incontriamo ora un problema in cui fa la differenza: l'ordinamento.

17.2.1 Il problema dell'ordinamento

Dato un array $a[1..n]$ di n numeri naturali, consideriamo il problema di riordinare gli elementi dell'array in ordine non decrescente in modo che:

$$a[1] \leq a[2] \leq \dots \leq a[n].$$

- Il problema è stato introdotto nel capitolo «Algoritmi di ordinamento», dove abbiamo studiato Insertion Sort, Exchange Sort e Selection Sort. Nel capitolo «Tempo di esecuzione» abbiamo dimostrato che il numero di passi di **InsertionSort** nel caso peggiore è $S(n) = \frac{3n^2+9n-8}{2} = \Theta(n^2)$.
- In questa sezione studiamo il **Merge Sort**, un algoritmo di ordinamento ricorsivo basato sui confronti e sul paradigma **divide et impera**, il cui numero di passi è $\Theta(n \log_2 n)$ nel caso peggiore.

Esempio 17.4: un'istanza di ordinamento

Consideriamo il seguente array di $n = 7$ numeri naturali:

$a[i]$	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$
	9	12	3	16	19	11	12

L'array ordinato è:

$a[i]$	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$
	3	9	11	12	12	16	19

17.2.2 L'algoritmo Merge Sort

17.2.2.1 Divide et impera

- Il Merge Sort ordina un sotto-array $a[l..r]$ come segue:
 1. **Divide**: si suddivide $a[l..r]$ nelle due metà $a[l..m]$ e $a[m+1..r]$, dove $m = \lfloor (l+r)/2 \rfloor$;
 2. **Impera**: si ordinano ricorsivamente le due metà;
 3. **Combina**: si **fondono** (*merge*) le due metà ordinate in un unico sotto-array ordinato $a[l..r]$.
- Il **caso base** è un sotto-array con un solo elemento ($l = r$), che è già ordinato: non occorre fare nulla.
- Tutto il lavoro è svolto nel passo **combina**, dalla procedura **Merge**. L'osservazione chiave è che due sequenze *ordinate* si possono fondere in modo molto efficiente: il più piccolo elemento del risultato è il minore tra i due primi elementi, e così via.

17.2.2.2 Pseudocodice

- Questo è lo pseudocodice dell'algoritmo **MergeSort**. Per ordinare l'intero array $a[1..n]$, chiamiamo **MergeSort**($a, 1, n$).

Algorithm: MergeSort

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

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

```

1 function MergeSort( $a, l, r$ )
2   if  $l < r$  then
3      $m := \lfloor (l + r) / 2 \rfloor$                                 // Divide
4     MergeSort( $a, l, m$ )                                // Impera (metà sinistra)
5     MergeSort( $a, m + 1, r$ )                            // Impera (metà destra)
6     Merge( $a, l, m, r$ )                                // Combina
7   end
8 end

```

- Se $k = r - l + 1 \geq 2$, 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 $k - 1$: la ricorsione raggiunge sempre il caso base.
- La procedura **Merge**(a, l, m, r) suppone che $a[l..m]$ e $a[m + 1..r]$ siano ordinati. Li copia in due array ausiliari L e R , e poi sposta ripetutamente in a il minore tra i primi due elementi non ancora copiati:

Algorithm: Merge

Input : Un array a e indici $l \leq m < r$ tali che $a[l..m]$ e $a[m+1..r]$ siano ordinati

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

```

1 function Merge( $a, l, m, r$ )
2    $n_1 := m - l + 1$ 
3    $n_2 := r - m$ 
4    $L :=$  nuovo array[1.. $n_1$ ],  $R :=$  nuovo array[1.. $n_2$ ]
5   for  $i := 1$  to  $n_1$  do
6      $L[i] := a[l + i - 1]$ 
7   end
8   for  $j := 1$  to  $n_2$  do
9      $R[j] := a[m + j]$ 
10  end
11   $i := 1$ 
12   $j := 1$ 
13   $k := l$ 
14  while  $i \leq n_1$  and  $j \leq n_2$  do
15    if  $L[i] \leq R[j]$  then
16       $a[k] \leftarrow L[i]$ 
17       $i \leftarrow i + 1$ 
18    else
19       $a[k] \leftarrow R[j]$ 
20       $j \leftarrow j + 1$ 
21    end
22     $k \leftarrow k + 1$ 
23  end
24  while  $i \leq n_1$  do // Copia il resto di L
25     $a[k] \leftarrow L[i]$ 
26     $i \leftarrow i + 1$ 
27     $k \leftarrow k + 1$ 
28  end
29  while  $j \leq n_2$  do // Copia il resto di R
30     $a[k] \leftarrow R[j]$ 
31     $j \leftarrow j + 1$ 
32     $k \leftarrow k + 1$ 
33  end
34 end

```

- Spiegazione di Merge:

- Le righe 2–9 copiano $a[l..m]$ in $L[1..n_1]$ e $a[m+1..r]$ in $R[1..n_2]$. Le copie sono necessarie, perché le posizioni $a[l..r]$ vengono sovrascritte durante la fusione.
- Gli indici i e j puntano ai primi elementi di L e R non ancora ricopiati, e k punta alla prossima posizione di a da riempire.
- Il primo ciclo while (riga 14) confronta $L[i]$ e $R[j]$ e copia il minore in $a[k]$ (in caso di parità, viene copiato per primo l'elemento di L : questo rende il Merge Sort **stabile**).

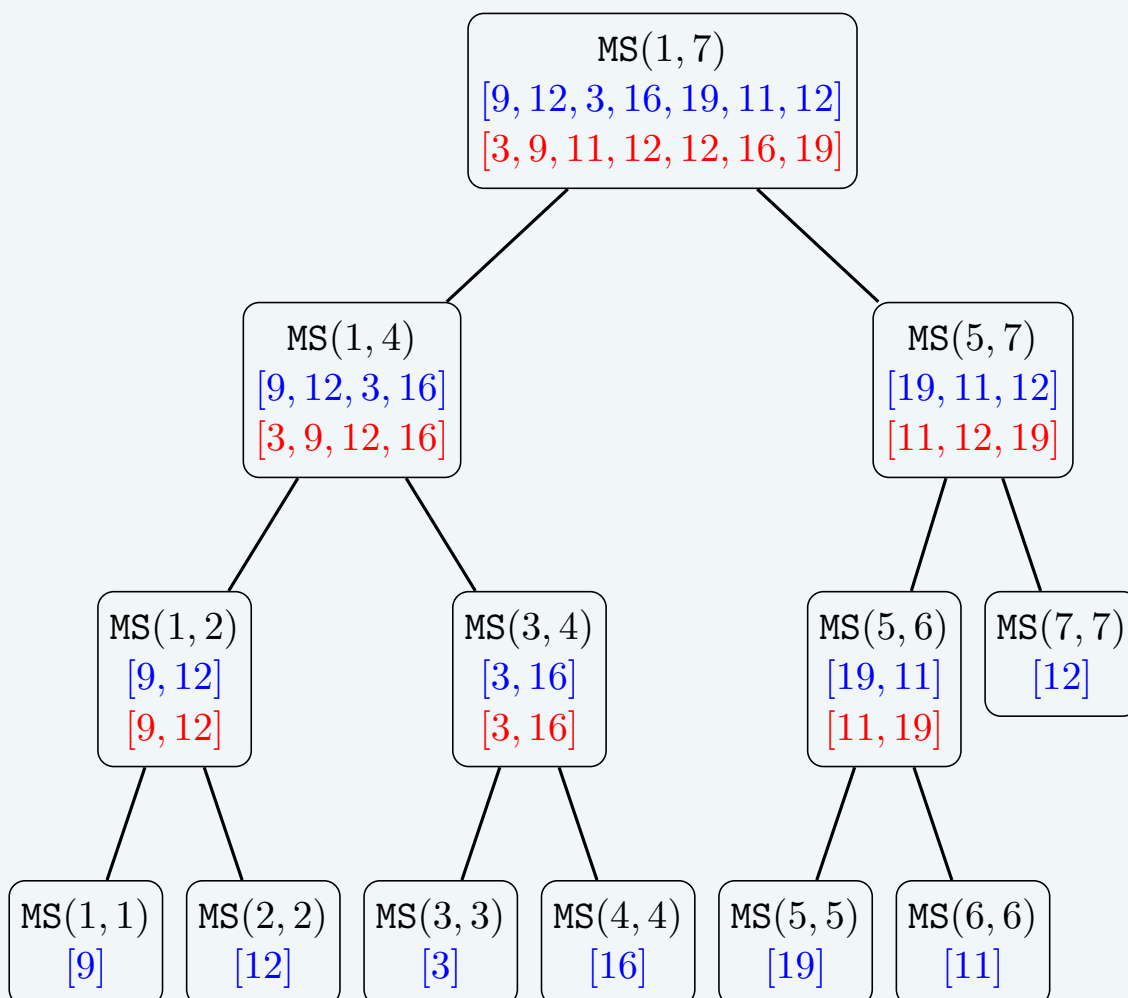
- Non appena uno dei due array si esaurisce, gli elementi rimanenti dell'altro (che sono già ordinati, e non più piccoli di quelli già copiati) vengono copiati in a dal ciclo while della riga 24 o da quello della riga 29. Solo uno di questi due cicli esegue almeno un'iterazione.

17.2.2.3 Esempi

Eseguiamo ora il Merge Sort sull'array dell'Esempio 17.4: guardiamo prima l'intero albero delle chiamate, poi le due fasi dell'algoritmo, e infine una singola chiamata di Merge.

Esempio 17.5: albero delle chiamate di MergeSort

Consideriamo l'array dell'Esempio 17.4, cioè $a = [9, 12, 3, 16, 19, 11, 12]$. Questo è l'albero delle chiamate di `MergeSort`($a, 1, 7$), in cui ogni nodo $MS(l, r)$ è una chiamata sul sotto-array $a[l..r]$. In ogni nodo, in **blu** il sotto-array ricevuto dalla chiamata e in **rosso** il sotto-array ordinato prodotto dalla chiamata (dopo la sua Merge):



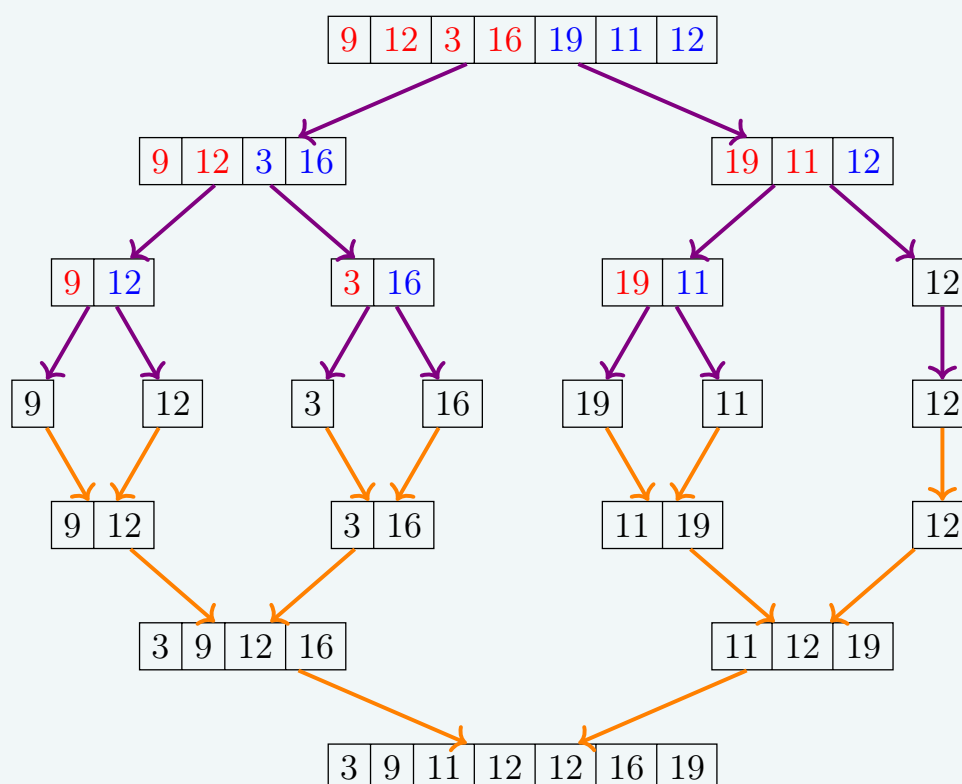
Le chiamate di Merge vengono eseguite nel seguente ordine (in profondità), e questo è il

contenuto dell'array dopo ciascuna di esse (il sotto-array modificato da ciascuna chiamata è in **rosso**):

chiamata	sotto-array fusi	array a dopo la chiamata
Merge(a , 1, 1, 2)	[9] e [12]	[9, 12 , 3, 16, 19, 11, 12]
Merge(a , 3, 3, 4)	[3] e [16]	[9, 12, 3, 16 , 19, 11, 12]
Merge(a , 1, 2, 4)	[9, 12] e [3, 16]	[3, 9, 12, 16 , 19, 11, 12]
Merge(a , 5, 5, 6)	[19] e [11]	[3, 9, 12, 16, 11, 19 , 12]
Merge(a , 5, 6, 7)	[11, 19] e [12]	[3, 9, 12, 16, 11, 12, 19]
Merge(a , 1, 4, 7)	[3, 9, 12, 16] e [11, 12, 19]	[3, 9, 11, 12, 12, 16, 19]

Esempio 17.6: fasi di divisione e di combinazione di MergeSort

Consideriamo di nuovo l'array dell'Esempio 17.4. La parte superiore della figura mostra la fase di **divisione** (freccie **viola**): ogni sotto-array viene suddiviso nella sua metà sinistra (in **rosso**) e nella sua metà destra (in **blu**), fino a ottenere sotto-array di un elemento. La parte inferiore mostra la fase di **combinazione** (freccie **arancioni**): coppie di sotto-array ordinati vengono fuse in un sotto-array ordinato.



Esempio 17.7: traccia di Merge

Consideriamo l'ultima chiamata $\text{Merge}(a, 1, 4, 7)$ dell'Esempio 17.5. Prima della chiamata $a = [3, 9, 12, 16, 11, 12, 19]$, con $a[1..4]$ e $a[5..7]$ ordinati. Allora $n_1 = 4$, $n_2 = 3$ e

$$L = [3, 9, 12, 16], \quad R = [11, 12, 19].$$

Questa è l'esecuzione dei cicli while (una riga per ogni elemento copiato in a):

k	i	j	$L[i]$	$R[j]$	confronto	azione
1	1	1	3	11	$3 \leq 11$	$a[1] \leftarrow 3, i = 2$
2	2	1	9	11	$9 \leq 11$	$a[2] \leftarrow 9, i = 3$
3	3	1	12	11	$12 > 11$	$a[3] \leftarrow 11, j = 2$
4	3	2	12	12	$12 \leq 12$	$a[4] \leftarrow 12$ (da L), $i = 4$
5	4	2	16	12	$16 > 12$	$a[5] \leftarrow 12$ (da R), $j = 3$
6	4	3	16	19	$16 \leq 19$	$a[6] \leftarrow 16, i = 5$
7	5	3	–	19	L esaurito	$a[7] \leftarrow 19, j = 4$

Dopo 6 iterazioni del primo ciclo while (e 6 confronti), $i = 5 > n_1$: l'array L è esaurito, e l'ultimo elemento di R viene copiato dal ciclo while della riga 29. Il risultato è $a = [3, 9, 11, 12, 12, 16, 19]$.

17.2.3 Correttezza

La correttezza del Merge Sort poggia su due fatti: **Merge** fonde correttamente due sotto-array ordinati (una proprietà di un ciclo, dimostrata con un'invariante di ciclo), e la ricorsione è corretta (dimostrata per induzione forte, come per **RecursiveSum**).

17.2.3.1 Correttezza di Merge

- Dimostriamo la correttezza di **Merge** con un'**invariante di ciclo**, cioè una proprietà che vale all'inizio di ogni iterazione di un ciclo. Come nel capitolo «Correttezza», mostriamo che l'invariante vale prima della prima iterazione (*inizializzazione*), che se vale prima di un'iterazione vale anche prima della successiva (*mantenimento*), e che cosa ci fornisce quando il ciclo termina (*terminazione*).

Osservazione 17.4: invariante di ciclo di Merge

Siano $a[l..m]$ e $a[m+1..r]$ ordinati. All'inizio di ogni iterazione del ciclo while della riga 14 di **Merge**:

- (I1) il sotto-array $a[l..k-1]$ contiene i $k-l$ elementi di $L[1..i-1]$ e $R[1..j-1]$, in ordine non decrescente;
- (I2) ogni elemento di $a[l..k-1]$ è minore o uguale a ogni elemento di $L[i..n_1]$ e di $R[j..n_2]$.

Alla fine, **Merge** termina e $a[l..r]$ contiene gli elementi originariamente in $a[l..r]$, ordinati in ordine non decrescente.

Dimostrazione. Dopo le righe 2–9, $L[1..n_1]$ e $R[1..n_2]$ sono copie dei sotto-array ordinati $a[l..m]$ e $a[m+1..r]$, quindi sono ordinati.

Inizializzazione. Prima della prima iterazione $k = l$, $i = 1$, $j = 1$: il sotto-array $a[l..k-1]$ è vuoto, e (I1) e (I2) valgono banalmente.

Mantenimento. Supponiamo (I1) e (I2) all'inizio di un'iterazione, e sia $L[i] \leq R[j]$ (l'altro caso è simmetrico). Poiché L e R sono ordinati, $L[i] \leq L[i+1..n_1]$ e $L[i] \leq R[j] \leq R[j+1..n_2]$: $L[i]$ è il più piccolo elemento non ancora copiato. Per (I2), $L[i]$ è maggiore o uguale a tutti gli elementi di $a[l..k-1]$, quindi dopo $a[k] \leftarrow L[i]$ il sotto-array $a[l..k]$ è ordinato e contiene $L[1..i]$ e $R[1..j-1]$. Dopo $i \leftarrow i+1$ e $k \leftarrow k+1$, (I1) vale di nuovo. Vale anche (I2), perché gli elementi non ancora copiati sono un sottoinsieme dei precedenti, e il nuovo elemento $a[k-1] = L[i-1]$ è minore o uguale a tutti loro.

Terminazione. Ogni iterazione aumenta $i+j$ di uno, quindi il ciclo termina dopo al più $n_1 + n_2$ iterazioni, con $i = n_1 + 1$ oppure $j = n_2 + 1$. Supponiamo $i = n_1 + 1$ (l'altro caso è simmetrico). Il ciclo della riga 24 viene saltato, e il ciclo della riga 29 copia $R[j..n_2]$, in questo ordine, in $a[k..r]$. Poiché $R[j..n_2]$ è ordinato e, per (I2), i suoi elementi non sono più piccoli di quelli di $a[l..k-1]$, l'intero sotto-array $a[l..r]$ è ordinato. Per (I1), contiene esattamente gli elementi di L e R , cioè gli elementi originariamente in $a[l..r]$. $\square$

17.2.3.2 Correttezza di MergeSort

Osservazione 17.5: correttezza di MergeSort

Per ogni coppia di indici $l \leq r$, la chiamata $\text{MergeSort}(a, l, r)$ termina e riordina gli elementi di $a[l..r]$ in ordine non decrescente (senza modificare il resto dell'array).

Dimostrazione. Usiamo l'induzione forte sul numero $k = r - l + 1 \geq 1$ di elementi del sotto-array.

Caso base ($k = 1$). Allora $l = r$, la condizione della riga 2 è falsa e non si fa nulla: un sotto-array di un elemento è ordinato.

Passo induttivo ($k \geq 2$). Supponiamo che l'enunciato valga per tutti i sotto-array con meno di k elementi. Poiché $l < r$, l'algoritmo calcola $m = \lfloor (l+r)/2 \rfloor$, con $l \leq m < r$. I sotto-array $a[l..m]$ 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, dopo le righe 4 e 5 sia $a[l..m]$ sia $a[m+1..r]$ sono ordinati (e contengono gli stessi elementi di prima). Quindi l'ipotesi di **Merge** è soddisfatta, e per l'Osservazione 17.4, dopo la riga 6 il sotto-array $a[l..r]$ è ordinato. $\square$

17.2.4 Numero di passi

17.2.4.1 Numero di passi di Merge

- Sia $n = r - l + 1 = n_1 + n_2$ il numero di elementi da fondere, e sia t il numero di iterazioni del ciclo while della riga 14. Tra queste, t_L copiano un elemento di L e t_R copiano un elemento di R , con $t_L + t_R = t$. L'istestazione di un ciclo for da 1 a n_1 viene eseguita $n_1 + 1$ volte (l'ultimo test fallisce). La riga **else** fa parte dell'**if** della riga 15 e non viene contata separatamente.

Il numero di volte che ciascuna riga di codice viene eseguita è:

- Le **righe** 2, 3, 4 vengono eseguite 1 volta ciascuna
- La **riga** 5 viene eseguita $n_1 + 1$ volte e la **riga** 6 n_1 volte
- La **riga** 8 viene eseguita $n_2 + 1$ volte e la **riga** 9 n_2 volte
- Le **righe** 11, 12, 13 vengono eseguite 1 volta ciascuna
- La **riga** 14 viene eseguita $t + 1$ volte
- La **riga** 15 viene eseguita t volte
- Le **righe** 16, 17 vengono eseguite t_L volte ciascuna
- Le **righe** 19, 20 vengono eseguite t_R volte ciascuna
- La **riga** 22 viene eseguita t volte
- La **riga** 24 viene eseguita $(n_1 - t_L) + 1$ volte e le **righe** 25, 26, 27 $n_1 - t_L$ volte ciascuna
- La **riga** 29 viene eseguita $(n_2 - t_R) + 1$ volte e le **righe** 30, 31, 32 $n_2 - t_R$ volte ciascuna

Quindi, il numero totale di passi di Merge è

$$\begin{aligned} & 3 + (2n_1 + 1) + (2n_2 + 1) + 3 + (t + 1) + t + 2t_L + 2t_R + t \\ & \quad + 4(n_1 - t_L) + 1 + 4(n_2 - t_R) + 1 \\ & = 11 + 2n + 5t + 4(n - t) = 6n + t + 11. \end{aligned}$$

- Ogni iterazione del primo ciclo while copia un elemento in a , quindi $t \leq n$. Più precisamente:
 - il ciclo si ferma non appena uno dei due array si esaurisce, quindi $t \geq \min\{n_1, n_2\}$;
 - quando il ciclo si ferma, l'altro array contiene ancora almeno un elemento, quindi $t \leq n - 1$.

Quindi il numero di passi di Merge su n elementi è compreso tra $6n + 11 + \min\{n_1, n_2\}$ e $7n + 10$; in particolare, è compreso tra $6n + 11$ e $7n + 10$. Il caso peggiore $t = n - 1$ si verifica, per esempio, nell'Esempio 17.7 ($n = 7$, $t = 6$, $6 \cdot 7 + 6 + 11 = 59$ passi).

Il numero di passi di Merge su $n = r - l + 1$ elementi è $6n + t + 11$ e soddisfa

$$6n + 11 \leq S_{\text{Merge}}(n) \leq 7n + 10, \quad \text{quindi} \quad S_{\text{Merge}}(n) = \Theta(n).$$

Il numero di confronti tra elementi (riga 15) è t , con $\min\{n_1, n_2\} \leq t \leq n - 1$.

17.2.4.2 L'equazione di ricorrenza di MergeSort

- Sia $S(n)$ il numero di passi eseguiti da MergeSort nel caso peggiore su un sotto-array di $n = r - l + 1$ elementi. Come per la somma di un array, una riga che contiene una chiamata conta 1 passo più i passi della funzione chiamata.
- Se $n = 1$, viene eseguita solo la riga 2: $S(1) = 1$.
- If $n \geq 2$:
 - La riga 2 viene eseguita 1 volta
 - La riga 3 viene eseguita 1 volta
 - La riga 4 viene eseguita 1 volta, più i $S(\lceil n/2 \rceil)$ passi della chiamata
 - La riga 5 viene eseguita 1 volta, più i $S(\lfloor n/2 \rfloor)$ passi della chiamata
 - La riga 6 viene eseguita 1 volta, più i $S_{\text{Merge}}(n)$ passi di Merge

e quindi

$$S(n) = S(\lceil n/2 \rceil) + S(\lfloor n/2 \rfloor) + f(n), \quad f(n) = 5 + S_{\text{Merge}}(n).$$

- Dall'analisi di Merge, $6n + 16 \leq f(n) \leq 7n + 15$. Poiché $7n + 15 \leq 15n$ per $n \geq 2$, abbiamo

$$6n \leq f(n) \leq 15n \quad \text{per ogni } n \geq 2, \quad \text{cioè} \quad f(n) = \Theta(n).$$

Il numero di passi di MergeSort è descritto dall'equazione di ricorrenza

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

che, ignorando le parti intere inferiori e superiori, scriviamo come

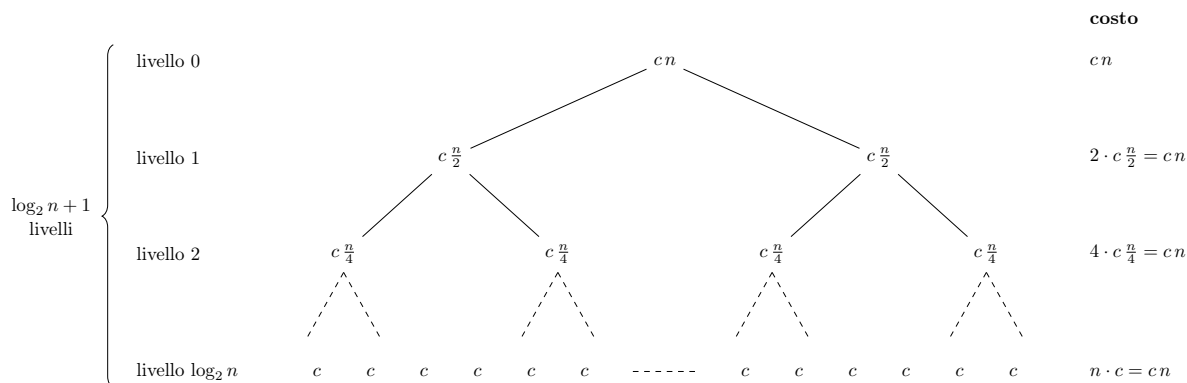
$$S(n) = 2S(n/2) + \Theta(n).$$

17.2.4.3 Soluzione con l'albero di ricorsione

- Supponiamo che $n = 2^h$ sia una potenza esatta di 2, e sia $c = 15$ la costante tale che $f(n) \leq cn$ (con $S(1) = 1 \leq c$). Limitiamo dapprima $S(n)$ dall'alto con l'albero di ricorsione di

$$S(n) \leq 2S(n/2) + cn,$$

dove ogni nodo contiene il costo di una chiamata, escluso il costo delle sue chiamate ricorsive:



- Il livello superiore ha costo cn ; il livello successivo ha costo $c \frac{n}{2} + c \frac{n}{2} = cn$; quello dopo ha costo $4 \cdot c \frac{n}{4} = cn$, e così via. In generale, il livello i ha 2^i nodi, ciascuno corrispondente a un sotto-array di $n/2^i$ elementi e con un contributo di costo al più $cn/2^i$, quindi ogni livello costa al più

$$2^i \cdot c \frac{n}{2^i} = cn.$$

- Il livello più basso $\log_2 n$ ha n foglie (sotto-array di un elemento), ciascuna con costo $S(1) = 1 \leq c$: il suo costo è al più cn .
- I livelli sono $0, 1, \dots, \log_2 n$, cioè $\log_2 n + 1$. Sommando i costi di tutti i livelli:

$$S(n) \leq cn (\log_2 n + 1) = cn \log_2 n + cn.$$

- Allo stesso modo, usando $f(n) \geq 6n$, ogni livello interno costa almeno $6n$, e ci sono $\log_2 n$ livelli interni: $S(n) \geq 6n \log_2 n$.
- Quindi, per n potenza di 2,

$$6n \log_2 n \leq S(n) \leq 15n \log_2 n + 15n, \quad \text{cioè} \quad S(n) = \Theta(n \log_2 n),$$

poiché

$$\lim_{n \rightarrow \infty} \frac{15n \log_2 n + 15n}{n \log_2 n} = \lim_{n \rightarrow \infty} \left(15 + \frac{15}{\log_2 n} \right) = 15 \in \mathbb{R}_{>0}.$$

- A differenza dell'albero di ricorsione della somma di un array, in cui il costo dei livelli raddoppia a ogni livello, qui **tutti i livelli hanno lo stesso costo** cn : il totale è il costo di un livello per il numero di livelli. Questa è esattamente la situazione del caso 2 del Master theorem.

17.2.4.4 Soluzione con il Master theorem

- Ricordiamo il caso 2 del Master theorem:

Sia $S(n) = aS(n/b) + f(n)$, con $a \geq 1$ e $b > 1$ costanti e $f(n)$ asintoticamente positiva. Se $f(n) = \Theta(n^{\log_b a} \log^k n)$ per una qualche costante $k \geq 0$, allora

$$S(n) = \Theta(n^{\log_b a} \log^{k+1} n).$$

- $S(n) = 2S(n/2) + \Theta(n) \rightarrow a = 2, b = 2, f(n) = \Theta(n).$
- $n^{\log_b a} = n^{\log_2 2} = n.$
- Confronto: $f(n)$ e $n^{\log_b a}$ crescono allo stesso modo $\rightarrow$ caso 2.
- Ipotesi: $f(n) = \Theta(n) = \Theta(n^{\log_2 2} \log^0 n)$ con $k = 0.$
- Conclusione: $S(n) = \Theta(n^{\log_2 2} \log^1 n) = \Theta(n \log n).$

La base del logaritmo è irrilevante all'interno di Θ , perché $\log_b n = \frac{\log_2 n}{\log_2 b}$ differisce da $\log_2 n$ per un fattore costante. Quindi $S(n) = \Theta(n \log_2 n)$, in accordo con l'albero di ricorsione.

17.2.4.5 Tempo di esecuzione nel caso peggiore

- Facciamo l'ipotesi che ogni elemento dell'array sia memorizzato usando un numero costante fissato di bit. Allora $|I| = \Theta(n)$: esistono costanti $c_1, c_2 > 0$ tali che, per ogni n sufficientemente grande,

$$\frac{1}{c_2}|I| \leq n \leq \frac{1}{c_1}|I|.$$

Passando ai logaritmi, $\log_2 |I| - \log_2 c_2 \leq \log_2 n \leq \log_2 |I| - \log_2 c_1$, quindi, per $|I|$ sufficientemente grande,

$$\frac{1}{2} \log_2 |I| \leq \log_2 n \leq 2 \log_2 |I|.$$

Moltiplicando le due catene di disuguaglianze (tutti i termini sono positivi),

$$\frac{1}{2c_2} |I| \log_2 |I| \leq n \log_2 n \leq \frac{2}{c_1} |I| \log_2 |I|,$$

cioè $n \log_2 n = \Theta(|I| \log_2 |I|)$. Poiché $S(n) = \Theta(n \log_2 n)$, otteniamo $S(n) = \Theta(|I| \log_2 |I|)$.

Se tutti i valori in ingresso usano un numero costante fissato di bit, allora il tempo di esecuzione nel caso peggiore dell'algoritmo MergeSort è

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

17.2.5 Memoria aggiuntiva

- **Merge**(a, l, m, r) alloca gli array ausiliari L e R , con $n_1 + n_2 = r - l + 1$ elementi in totale, che vengono rilasciati quando **Merge** termina.
- **Merge** viene chiamata solo dopo che entrambe le chiamate ricorsive sono terminate, quindi in ogni istante è attiva al più una **Merge**, e gli array ausiliari più grandi sono quelli della prima chiamata **MergeSort**($a, 1, n$), con n elementi in totale.
- Inoltre, la pila delle chiamate contiene al più $\lceil \log_2 n \rceil + 1$ record di attivazione di dimensione costante (l'albero delle chiamate ha la stessa forma di quello di **RecursiveSum**, Osservazione 17.3): memoria $\Theta(\log_2 n)$.

MergeSort usa $\Theta(n) + \Theta(\log_2 n) = \Theta(n)$ di memoria aggiuntiva: **non** è un algoritmo di ordinamento *in place*. Al contrario, **InsertionSort** ordina sul posto, con $\Theta(1)$ di memoria aggiuntiva.

17.2.6 Confronto con Insertion Sort

- Nel caso peggiore, **InsertionSort** esegue $\Theta(n^2)$ passi, mentre **MergeSort** esegue $\Theta(n \log_2 n)$ passi. Poiché

$$\lim_{n \rightarrow \infty} \frac{n \log_2 n}{n^2} = \lim_{n \rightarrow \infty} \frac{\log_2 n}{n} = 0,$$

il Merge Sort è asintoticamente molto più veloce.

- Per farsi un'idea della differenza, ignoriamo le costanti moltiplicative e supponiamo che un calcolatore esegua 10^9 passi al secondo:

n	n^2	$n \log_2 n$	tempo per n^2 passi	tempo per $n \log_2 n$ passi
10	100	≈ 33	10^{-7} s	$\approx 3.3 \cdot 10^{-8}$ s
10^3	10^6	≈ 9966	10^{-3} s	$\approx 10^{-5}$ s
10^6	10^{12}	$\approx 1.99 \cdot 10^7$	10^3 s ≈ 16.7 min	≈ 0.02 s

- Per array piccoli la differenza è trascurabile (e, a causa delle costanti e del costo delle chiamate ricorsive, **InsertionSort** può anche essere più veloce). Per array grandi, il Merge Sort è enormemente più veloce: con $n = 10^6$ elementi è circa 50 000 volte più veloce.

17.2.7 Implementazione in C++

- Questa è un'implementazione in C++ dell'algoritmo Merge Sort. Gli array sono indicizzati da 0: l'intero array viene ordinato dalla chiamata `MergeSort(a, 0, n-1)`. Gli array ausiliari sono `std::vector`, che vengono rilasciati automaticamente alla fine di `Merge`.

```

1  #include <vector>
2
3  // Fonde i sotto-array ordinati a[l..m] e a[m+1..r] (indici da 0)
4  void Merge(int *a, int l, int m, int r)
5  {
6      int n1 = m - l + 1; // Dimensione della parte sinistra
7      int n2 = r - m;      // Dimensione della parte destra
8
9      std::vector<int> L(n1), R(n2); // Array ausiliari
10
11     for (int i = 0; i < n1; i++)
12     {
13         L[i] = a[l + i];
14     }
15     for (int j = 0; j < n2; j++)
16     {
17         R[j] = a[m + 1 + j];
18     }
19
20     int i = 0, j = 0, k = l;
21
22     while (i < n1 && j < n2) // Fonde L e R di nuovo in a[l..r]
23     {
24         if (L[i] <= R[j])
25         {
26             a[k] = L[i];
27             i++;
28         }
29         else
30         {
31             a[k] = R[j];
32             j++;
33         }
34         k++;
35     }
36
37     while (i < n1) // Copia gli elementi rimanenti di L
38     {
39         a[k] = L[i];
40         i++;
41         k++;
42     }
43
44     while (j < n2) // Copia gli elementi rimanenti di R
45     {
46         a[k] = R[j];
47         j++;
48         k++;
49     }
50 }
51

```

```

52 // Ordina a[l..r] in ordine non decrescente (indici da 0)
53 void MergeSort(int *a, int l, int r)
54 {
55     if (l < r)
56     {
57         int m = l + (r - l) / 2; // Equivale a (l + r) / 2
58
59         MergeSort(a, l, m);
60         MergeSort(a, m + 1, r);
61         Merge(a, l, m, r);
62     }
63 }

```

Esempio 17.8: esecuzione dell'implementazione in C++ di Merge

Eseguiamo la chiamata `Merge(a, 0, 1, 3)` sull'array $a = [5, 8, 2, 7]$: i sotto-array $a[0..1] = [5, 8]$ e $a[2..3] = [2, 7]$ sono ordinati.

$a[j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$l = 0$	$m = 1$	$r = 3$
	5	8	2	7			

Dopo le copie,

$$n1 = 1 - 0 + 1 = 2, \quad n2 = 3 - 1 = 2, \quad L = [5, 8], \quad R = [2, 7], \quad i = 0, \quad j = 0, \quad k = 0.$$

- **Iterazione 1** (primo ciclo while). $L[0] = 5 \leq R[0] = 2$? No $\Rightarrow a[0] = R[0] = 2, j = 1, k = 1$; ora $a = [2, 8, 2, 7]$.
- **Iterazione 2** (primo ciclo while). $L[0] = 5 \leq R[1] = 7$? Sì $\Rightarrow a[1] = L[0] = 5, i = 1, k = 2$; ora $a = [2, 5, 2, 7]$.
- **Iterazione 3** (primo ciclo while). $L[1] = 8 \leq R[1] = 7$? No $\Rightarrow a[2] = R[1] = 7, j = 2, k = 3$; ora $a = [2, 5, 7, 7]$.
- La condizione $j < n2$ è falsa ($j = 2$): R è esaurito e il primo ciclo while termina.
- **Iterazione 1** (copia del resto di L). $a[3] = L[1] = 8, i = 2, k = 4$; ora $a = [2, 5, 7, 8]$.
- Le condizioni $i < n1$ e $j < n2$ sono entrambe false: **Merge** termina.

L'algoritmo termina con $a = [2, 5, 7, 8]$, dopo $3 = n - 1$ confronti. Si noti che i valori originali $a[0] = 5$ e $a[1] = 8$ vengono sovrascritti durante l'esecuzione: ecco perché le copie L e R sono necessarie.

Punti chiave sul Merge Sort.

- Tutto il lavoro è svolto da **Merge**, che fonde due sotto-array ordinati di dimensione totale n in $6n + t + 11 = \Theta(n)$ passi.
- L'equazione di ricorrenza $S(n) = 2S(n/2) + \Theta(n)$ ricade nel caso 2 del Master theorem: $\log_2 n + 1$ livelli di costo $\Theta(n)$ ciascuno, $S(n) = \Theta(n \log_2 n)$.
- Il prezzo è la memoria: $\Theta(n)$ per gli array ausiliari.

algoritmo	numero di passi nel caso peggiore	memoria aggiuntiva
InsertionSort	$\Theta(n^2)$	$\Theta(1)$
MergeSort	$\Theta(n \log_2 n)$	$\Theta(n)$

17.3 Ricerca binaria ricorsiva

Il Merge Sort effettua due chiamate ricorsive sulle due metà. Il prossimo algoritmo ne effettua soltanto **una**: dopo un solo confronto, una delle due metà può essere scartata.

17.3.1 L'algoritmo

Dato un array $a[1..n]$ di n numeri naturali ordinati in ordine non decrescente e un numero naturale m da cercare, consideriamo il problema di determinare se m è uno degli elementi dell'array.

- Nel capitolo «Ricerca lineare e binaria» abbiamo risolto questo problema con l'algoritmo **iterativo BinarySearch**: l'algoritmo mantiene un intervallo di ricerca $a[left..right]$, inizialmente $a[1..n]$, e a ogni iterazione confronta il valore cercato m con l'elemento centrale

$$a[mid], \quad mid = left + \left\lfloor \frac{right - left}{2} \right\rfloor.$$

Se $a[mid] = m$ l'elemento è stato trovato; altrimenti, poiché l'array è ordinato, metà dell'intervallo può essere scartata.

- L'osservazione chiave è che, dopo il confronto con $a[mid]$, il problema che resta da risolvere è **lo stesso problema** (cercare m in un array ordinato) su un sotto-array **più piccolo**:
 - se $a[mid] < m$, allora m può trovarsi solo in $a[mid + 1..right]$;
 - se $a[mid] > m$, allora m può trovarsi solo in $a[left..mid - 1]$.
- Questa è esattamente la struttura richiesta da un algoritmo ricorsivo: risolviamo il problema su $a[left..right]$ risolvendo lo stesso problema su un sotto-array di dimensione (circa) metà. I **casi base** sono l'intervallo vuoto ($left > right$), in cui m certamente non è presente, e il caso $a[mid] = m$, in cui m viene trovato.
- A differenza della versione iterativa, i limiti $left$ e $right$ sono ora **parametri** della funzione. Per cercare m nell'intero array, chiamiamo

RecursiveBinarySearch($a, 1, n, m$).

- Questo è lo pseudocodice dell'algoritmo di ricerca binaria ricorsiva:

Algorithm: RecursiveBinarySearch

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

Output: Un indice $i \in \{left, \dots, right\}$ tale che $a[i] = m$, oppure -1 se m non è in $a[left..right]$

```

1 function RecursiveBinarySearch(a, left, right, m)
2   if left > right then
3     | return -1                                // Intervallo vuoto: non trovato
4   end
5   mid := left + ⌊(right - left)/2⌋              // Indice centrale
6   if a[mid] = m then
7     | return mid                               // Elemento trovato
8   end
9   if a[mid] < m then
10    | return RecursiveBinarySearch(a, mid + 1, right, m)
11  end
12  return RecursiveBinarySearch(a, left, mid - 1, m)
13 end

```

- Spiegazione dell'algoritmo:

- **Righe 2–3**: se l'intervallo è vuoto ($left > right$), l'elemento non è presente e viene restituito -1 (**caso base**).
- **Riga 5**: l'indice centrale mid viene calcolato con la stessa formula dell'algoritmo iterativo.
- **Righe 6–7**: se $a[mid] = m$, l'elemento è stato trovato e il suo indice viene restituito (**caso base**).
- **Righe 9–10**: se $a[mid] < m$, la ricerca prosegue **ricorsivamente** nella metà destra $a[mid + 1..right]$.
- **Riga 12**: altrimenti $a[mid] > m$, e la ricerca prosegue **ricorsivamente** nella metà sinistra $a[left..mid - 1]$.
- Il valore restituito dalla chiamata ricorsiva viene restituito invariato al chiamante.

- Ogni chiamata di RecursiveBinarySearch effettua **al più una** chiamata ricorsiva. Quindi, l'**albero delle chiamate** è un **cammino**: non ha ramificazioni. Questo è molto diverso dal Merge Sort, in cui ogni chiamata su un array con almeno due elementi genera due chiamate ricorsive.

17.3.2 Esempi

- Consideriamo le stesse ricerche eseguite nel capitolo «Ricerca lineare e binaria» con l'algoritmo iterativo. Per ogni ricerca riportiamo una riga per ogni chiamata: i valori di *left*, *right*, *mid*, $a[mid]$ e l'azione eseguita. Poi disegniamo l'albero delle chiamate: una freccia nera è una chiamata ricorsiva, una freccia blu è il valore restituito al chiamante.

Esempio 17.9: ricerca binaria ricorsiva su un array ordinato

Cerchiamo quattro valori nell'array ordinato $a = [3, 9, 11, 12, 12, 16, 19]$ ($n = 7$).

- Ricerca di $m = 12$.** Prima chiamata `RecursiveBinarySearch(a, 1, 7, 12)`:

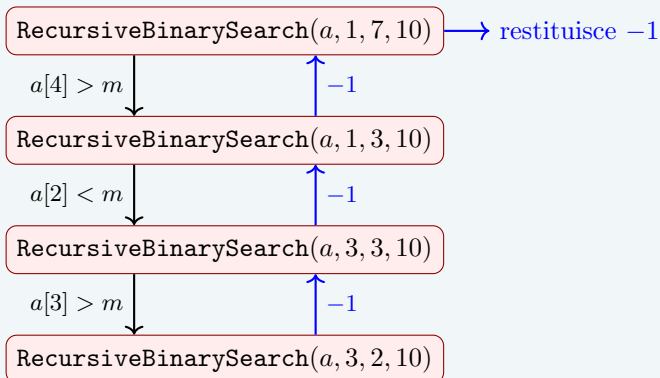
chiamata	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	azione
1	1	7	4	12	$a[mid] = m \Rightarrow$ Trovato! Ritorna 4

`RecursiveBinarySearch(a, 1, 7, 12)` → restituisce 4

L'elemento 12 viene trovato all'indice 4 con una sola chiamata (nessuna ricorsione).

- Ricerca di $m = 10$.** Prima chiamata `RecursiveBinarySearch(a, 1, 7, 10)`:

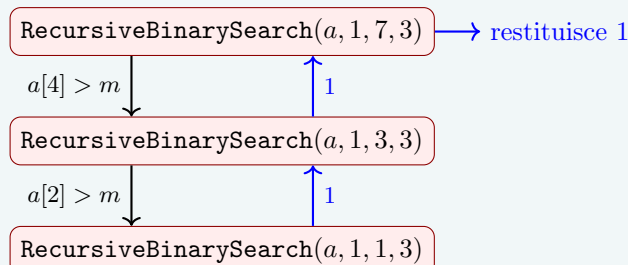
chiamata	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	azione
1	1	7	4	12	$a[mid] > m \Rightarrow$ chiama $left = 1, right = 3$
2	1	3	2	9	$a[mid] < m \Rightarrow$ chiama $left = 3, right = 3$
3	3	3	3	11	$a[mid] > m \Rightarrow$ chiama $left = 3, right = 2$
4	3	2	–	–	$left > right \Rightarrow$ Non trovato! Ritorna –1



L'elemento 10 non è nell'array: l'ultima chiamata è su un intervallo vuoto e restituisce –1, e questo valore viene propagato fino alla prima chiamata.

- Ricerca di $m = 3$.** Prima chiamata `RecursiveBinarySearch(a, 1, 7, 3)`:

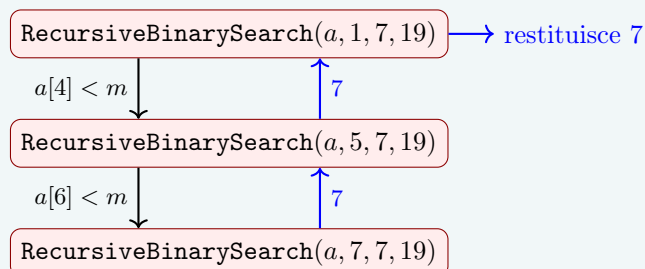
chiamata	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	azione
1	1	7	4	12	$a[mid] > m \Rightarrow$ chiama $left = 1, right = 3$
2	1	3	2	9	$a[mid] > m \Rightarrow$ chiama $left = 1, right = 1$
3	1	1	1	3	$a[mid] = m \Rightarrow$ Trovato! Ritorna 1



L'elemento 3 viene trovato all'indice 1.

- **Ricerca di $m = 19$.** Prima chiamata `RecursiveBinarySearch(a, 1, 7, 19)`:

chiamata	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	azione
1	1	7	4	12	$a[mid] < m \Rightarrow$ chiama $left = 5, right = 7$
2	5	7	6	16	$a[mid] < m \Rightarrow$ chiama $left = 7, right = 7$
3	7	7	7	19	$a[mid] = m \Rightarrow$ Trovato! Ritorna 7



L'elemento 19 viene trovato all'indice 7.

- La successione dei valori ($left, right, mid$) è **esattamente la stessa** prodotta dall'algoritmo iterativo: ogni chiamata dell'algoritmo ricorsivo corrisponde a un'iterazione del ciclo `while` dell'algoritmo iterativo.
- Si noti che la seconda occorrenza di 12 (indice 5) non viene mai trovata cercando $m = 12$: come nella versione iterativa, l'algoritmo restituisce **un** indice i con $a[i] = m$, non necessariamente il primo.

17.3.3 Correttezza

- Come per gli algoritmi precedenti, la correttezza si dimostra per **induzione forte** sulla dimensione dell'input della chiamata. La dimensione della chiamata $\text{RecursiveBinarySearch}(a, left, right, m)$ è il numero di elementi dell'intervallo:

$$k = right - left + 1.$$

Per un intervallo vuoto ($left = right + 1$) si ha $k = 0$.

Teorema 17.6: correttezza della ricerca binaria ricorsiva

Sia $a[1..n]$ ordinato in ordine non decrescente e siano $1 \leq left \leq right + 1 \leq n + 1$. Allora $\text{RecursiveBinarySearch}(a, left, right, m)$ termina e restituisce:

- un indice $i \in \{left, \dots, right\}$ tale che $a[i] = m$, se m è uno degli elementi di $a[left..right]$;
- -1 , altrimenti.

Dimostrazione. Procediamo per induzione forte su $k = right - left + 1 \geq 0$.

Caso base $k = 0$. Allora $left > right$, l'intervallo è vuoto e m non è uno dei suoi elementi. L'algoritmo esegue le righe 2–3 e restituisce -1 , il che è corretto.

Passo induttivo. Sia $k \geq 1$ e supponiamo (ipotesi induttiva) che l'enunciato valga per ogni chiamata il cui intervallo ha dimensione $k' < k$. Poiché $left \leq right$, l'algoritmo calcola

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

- Se $a[mid] = m$, l'algoritmo restituisce $mid \in \{left, \dots, right\}$, il che è corretto.
- Se $a[mid] < m$, allora, poiché l'array è ordinato, $a[i] \leq a[mid] < m$ per ogni $i \in \{left, \dots, mid\}$. Quindi m è in $a[left..right]$ se e solo se è in $a[mid + 1..right]$. Questo intervallo ha dimensione $right - mid < k$ (perché $mid \geq left$), quindi per l'ipotesi induttiva la chiamata $\text{RecursiveBinarySearch}(a, mid + 1, right, m)$ termina e restituisce una risposta corretta, che viene restituita invariata.
- Se $a[mid] > m$, allora $a[i] \geq a[mid] > m$ per ogni $i \in \{mid, \dots, right\}$. Quindi m è in $a[left..right]$ se e solo se è in $a[left..mid - 1]$, un intervallo di dimensione $mid - left < k$ (perché $mid \leq right$). Per l'ipotesi induttiva, la chiamata ricorsiva restituisce una risposta corretta.

In tutti i casi la chiamata termina e restituisce una risposta corretta. $\square$

- Applicando il Teorema 17.6 con $left = 1$ e $right = n$, otteniamo che $\text{RecursiveBinarySearch}(a, 1, n, m)$ risolve il problema.

17.3.4 Tempo di esecuzione nel caso peggiore

17.3.4.1 L'equazione di ricorrenza

- Sia $S(n)$ il numero di passi eseguiti da `RecursiveBinarySearch` nel caso peggiore su un intervallo di $n = right - left + 1$ elementi (compresi i passi eseguiti da tutte le chiamate ricorsive).
- Il numero di passi eseguiti da una singola chiamata, **senza** contare i passi della chiamata ricorsiva, è:
 - intervallo vuoto: righe 2 e 3 $\Rightarrow$ 2 passi;
 - elemento trovato: righe 2, 5, 6, 7 $\Rightarrow$ 4 passi;
 - chiamata ricorsiva: righe 2, 5, 6, 9 e riga 10 (oppure 12) $\Rightarrow$ 5 passi.
- **Dimensione dei sottoproblemi.** Sia $n = right - left + 1 \geq 1$. Poiché $mid - left = \lfloor (n-1)/2 \rfloor$:
 - l'intervallo sinistro $a[left..mid-1]$ ha $mid - left = \lfloor \frac{n-1}{2} \rfloor$ elementi;
 - l'intervallo destro $a[mid+1..right]$ ha $right - mid = (n-1) - \lfloor \frac{n-1}{2} \rfloor = \lceil \frac{n-1}{2} \rceil = \lfloor \frac{n}{2} \rfloor$ elementi.

Per esempio, per $n = 7$ entrambe le metà hanno 3 elementi; per $n = 8$ la metà sinistra ha 3 elementi e la metà destra ne ha 4.

- Nel caso peggiore l'elemento non viene trovato prima che l'intervallo diventi vuoto, e la ricorsione prosegue sulla metà più grande, che ha $\lfloor n/2 \rfloor$ elementi (si verifica facilmente per induzione che S è non decrescente). Quindi, $S(n)$ soddisfa l'equazione di ricorrenza

$$S(n) = \begin{cases} 2 & \text{se } n = 0, \\ S(\lfloor n/2 \rfloor) + 5 & \text{se } n \geq 1. \end{cases}$$

Questo caso peggiore si verifica, per esempio, quando $m > a[n]$: l'algoritmo prosegue sempre nella metà destra, finché l'intervallo diventa vuoto.

- In generale, scriviamo

$$S(n) = S(\lfloor n/2 \rfloor) + c,$$

dove c è una costante (qui $c = 5$): un algoritmo ricorsivo che riduce il problema a **un** sottoproblema di dimensione **metà**, svolgendo una quantità **costante** di lavoro aggiuntivo.

17.3.4.2 Soluzione per espansione

- Espandiamo l'equazione di ricorrenza, usando la proprietà $\lfloor \lfloor n/2^i \rfloor / 2 \rfloor = \lfloor n/2^{i+1} \rfloor$:

$$\begin{aligned} S(n) &= S(\lfloor n/2 \rfloor) + 5 \\ &= S(\lfloor n/4 \rfloor) + 5 + 5 \\ &= S(\lfloor n/8 \rfloor) + 5 \cdot 3 \\ &\vdots \\ &= S(\lfloor n/2^j \rfloor) + 5j. \end{aligned}$$

- L'espansione si ferma al primo j tale che $\lfloor n/2^j \rfloor = 0$, cioè $2^j > n$. Per $n \geq 1$, questo è

$$j = \lfloor \log_2 n \rfloor + 1,$$

perché $2^{\lfloor \log_2 n \rfloor} \leq n < 2^{\lfloor \log_2 n \rfloor + 1}$. Quindi, per ogni $n \geq 1$,

$$S(n) = 5(\lfloor \log_2 n \rfloor + 1) + 2.$$

- Per esempio, $S(7) = 5 \cdot (2 + 1) + 2 = 17$: nella ricerca di $m = 10$ dell'Esempio 17.9, le tre chiamate su intervalli non vuoti eseguono 5 passi ciascuna e l'ultima chiamata (intervallo vuoto) ne esegue 2.
- Poiché $\log_2 n - 1 < \lfloor \log_2 n \rfloor \leq \log_2 n$, abbiamo

$$\lim_{n \rightarrow \infty} \frac{S(n)}{\log_2 n} = \lim_{n \rightarrow \infty} \frac{5\lfloor \log_2 n \rfloor + 7}{\log_2 n} = 5 \in \mathbb{R}_{>0}, \quad \text{quindi} \quad S(n) = \Theta(\log_2 n).$$

17.3.4.3 Soluzione con il Master theorem

- Lo stesso risultato si ottiene immediatamente con il Master theorem, applicato a

$$S(n) = S(n/2) + c$$

(come di consueto, le parti intere inferiori non cambiano la soluzione asintotica).

- **Identificare** $a, b, f(n)$: $a = 1, b = 2, f(n) = c$.
- **Calcolare** $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$.
- **Confrontare**: $f(n) = c = \Theta(1) = \Theta(n^{\log_2 1} \log^0 n)$.
- **Verificare l'ipotesi**: siamo nel caso 2 del Master theorem con $k = 0$.
- **Concludere**:

$$S(n) = \Theta(n^{\log_2 1} \log^{0+1} n) = \Theta(\log n) = \Theta(\log_2 n).$$

17.3.4.4 Numero di chiamate ricorsive

- Ogni chiamata su un intervallo non vuoto confronta m con esattamente un elemento $a[mid]$. Quindi, il numero di chiamate su un intervallo non vuoto è il numero di elementi dell'array esaminati dall'algoritmo (esattamente come il numero di iterazioni del ciclo `while` dell'algoritmo iterativo).

Osservazione 17.7: numero di chiamate della ricerca binaria ricorsiva

Per $n \geq 1$, la chiamata `RecursiveBinarySearch(a, 1, n, m)` esegue al più

$$\lfloor \log_2 n \rfloor + 1$$

chiamate su un intervallo non vuoto, e questo limite viene raggiunto (per esempio quando $m > a[n]$). Contando anche l'ultima chiamata su un intervallo vuoto, il numero totale di chiamate è al più $\lfloor \log_2 n \rfloor + 2$.

Dimostrazione. Sia n_i la dimensione dell'intervallo della i -esima chiamata ($i = 1$ è la prima chiamata, quindi $n_1 = n$). Per l'analisi dei sottoproblemi, $n_{i+1} \leq \lfloor n_i/2 \rfloor$. Per induzione su i ,

$$n_i \leq \left\lfloor \frac{n}{2^{i-1}} \right\rfloor :$$

vale per $i = 1$, e se vale per i allora $n_{i+1} \leq \lfloor n_i/2 \rfloor \leq \lfloor \lfloor n/2^{i-1} \rfloor / 2 \rfloor = \lfloor n/2^i \rfloor$.

Se la i -esima chiamata è su un intervallo non vuoto, allora $n_i \geq 1$, quindi $n/2^{i-1} \geq 1$, cioè $2^{i-1} \leq n$ e $i - 1 \leq \log_2 n$. Poiché $i - 1$ è un intero, $i - 1 \leq \lfloor \log_2 n \rfloor$, cioè

$$i \leq \lfloor \log_2 n \rfloor + 1.$$

Quindi, al più $\lfloor \log_2 n \rfloor + 1$ chiamate sono su un intervallo non vuoto, e può seguirne al più un'altra (su un intervallo vuoto, che non effettua chiamate ricorsive).

Se $m > a[n]$, l'algoritmo prosegue sempre nella metà destra, la cui dimensione è esattamente $\lfloor n_i/2 \rfloor$. Quindi $n_i = \lfloor n/2^{i-1} \rfloor$, che è ≥ 1 per $i = \lfloor \log_2 n \rfloor + 1$: il limite viene raggiunto, ed è seguito da una chiamata su un intervallo vuoto. $\square$

- Per esempio, per $n = 7$ si ha $\lfloor \log_2 7 \rfloor + 1 = 3$: nella ricerca di $m = 10$ ci sono 3 chiamate su intervalli non vuoti, più l'ultima chiamata sull'intervallo vuoto $left = 3$, $right = 2$.
- Lo stesso ragionamento mostra che anche l'algoritmo iterativo esegue al più $\lfloor \log_2 n \rfloor + 1$ iterazioni del ciclo **while**.

17.3.4.5 Tempo di esecuzione nel caso peggiore in funzione della dimensione dell'input

- L'input dell'algoritmo è costituito dall'array $a[1..n]$ e dal valore cercato m . Sotto l'ipotesi che ogni valore usi un numero costante di bit, la dimensione dell'input soddisfa

$$|I| = \Theta(n),$$

cioè esistono costanti $c_1, c_2 > 0$ tali che $c_1 n \leq |I| \leq c_2 n$ per ogni n sufficientemente grande. Passando ai logaritmi,

$$\log_2 |I| - \log_2 c_2 \leq \log_2 n \leq \log_2 |I| - \log_2 c_1,$$

quindi $\log_2 n = \Theta(\log_2 |I|)$ (le costanti additive sono trascurabili rispetto a $\log_2 |I|$).

- Poiché $S(n) = \Theta(\log_2 n)$, otteniamo $S(n) = \Theta(\log_2 |I|)$.

Se tutti i valori in ingresso usano un numero costante fissato di bit, allora il tempo di esecuzione nel caso peggiore dell'algoritmo **RecursiveBinarySearch** è

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

Quindi, come la versione iterativa, l'algoritmo è logaritmico nella dimensione dell'input: **non** è necessario leggere l'intero input per risolvere il problema, perché l'array è ordinato.

17.3.5 Spazio: ricerca binaria ricorsiva contro iterativa

- L'algoritmo ricorsivo e quello iterativo eseguono gli stessi confronti e hanno lo stesso tempo di esecuzione nel caso peggiore. Differiscono per la **memoria** usata.
- Ogni chiamata che non è ancora terminata mantiene in memoria una propria copia dei parametri e delle variabili locali (*left*, *right*, *m*, *mid*, un riferimento all'array e l'indirizzo di ritorno). Queste informazioni sono memorizzate nella **pila delle chiamate**, e occupano una quantità costante di memoria per chiamata.
- In `RecursiveBinarySearch` nessuna chiamata termina prima che sia terminata la sua chiamata ricorsiva. Quindi, quando è in esecuzione l'ultima chiamata, **tutte** le chiamate del cammino sono nella pila delle chiamate. Per l'Osservazione 17.7, nel caso peggiore sono $\lfloor \log_2 n \rfloor + 2$, e la memoria aggiuntiva usata è

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

- L'algoritmo iterativo usa solo le variabili *left*, *right* e *mid*: la sua memoria aggiuntiva è

$$\Theta(1).$$

Definizione 17.8: ricorsione in coda

Una chiamata ricorsiva è una **chiamata in coda** (*tail call*) se è l'ultima operazione eseguita dalla funzione chiamante, cioè il valore che restituisce viene restituito direttamente dal chiamante senza ulteriori calcoli. Una funzione le cui chiamate ricorsive sono tutte chiamate in coda si dice **ricorsiva in coda** (*tail recursive*).

- `RecursiveBinarySearch` è ricorsiva in coda: nelle **righe** 10 e 12 il valore della chiamata ricorsiva viene restituito così com'è.
- Una chiamata in coda si può sostituire aggiornando i parametri e tornando all'inizio della funzione:
 - la **riga** 10 diventa $left \leftarrow mid + 1$;
 - la **riga** 12 diventa $right \leftarrow mid - 1$;
 - il test della **riga** 2 diventa la condizione di un ciclo **while** «**while** $left \leq right$ ».

Il risultato di questa trasformazione è **esattamente** l'algoritmo iterativo `BinarySearch`. Ecco perché i due algoritmi producono la stessa successione di valori di *left*, *right* e *mid*.

- Molti compilatori eseguono automaticamente questa trasformazione (**eliminazione delle chiamate in coda**, *tail-call elimination*), in modo che la versione ricorsiva usi solo $\Theta(1)$ di memoria aggiuntiva. Per esempio, il compilatore `g++` 15.2 elimina le chiamate ricorsive dell'implementazione `C++` qui sotto con l'opzione `-O2`, ma non con `-O0`. Tuttavia, questo non è garantito dallo standard `C++`.

La ricerca binaria ricorsiva e quella iterativa hanno lo stesso tempo di esecuzione nel caso peggiore $\Theta(\log n)$, ma la versione ricorsiva usa $\Theta(\log n)$ di memoria aggiuntiva per la pila

delle chiamate, mentre la versione iterativa usa $\Theta(1)$ di memoria aggiuntiva. Ogni algoritmo ricorsivo in coda può essere trasformato in uno iterativo.

17.3.6 Implementazione in C++

- Questa è un'implementazione in C++ dell'algoritmo di ricerca binaria ricorsiva. Come di consueto in C++, l'array è indicizzato da 0: per cercare m nell'intero array di n elementi chiamiamo `RecursiveBinarySearch(a, 0, n-1, m)`.

```

1 // Restituisce un indice di m in a[left..right], oppure -1 se m non c'è
2 // L'array a deve essere ordinato in ordine non decrescente
3
4 int RecursiveBinarySearch(int *a, int left, int right, int m)
5 {
6     if (left > right)
7     {
8         return -1; // Intervallo vuoto: elemento non trovato
9     }
10
11     int mid = left + (right - left) / 2;
12
13     if (a[mid] == m)
14     {
15         return mid; // Elemento trovato all'indice mid
16     }
17
18     if (a[mid] < m)
19     {
20         return RecursiveBinarySearch(a, mid + 1, right, m); // Metà destra
21     }
22
23     return RecursiveBinarySearch(a, left, mid - 1, m); // Metà sinistra
24 }
25
26 int main()
27 {
28     int a[] = {3, 9, 11, 12, 12, 16, 19};
29     int n = 7;
30     int result = RecursiveBinarySearch(a, 0, n - 1, 10); // result = -1
31     return 0;
32 }
```

Esempio 17.10: esecuzione dell'implementazione in C++ di RecursiveBinarySearch

Cerchiamo $m = 10$ e $m = 19$ nell'array ordinato $a = [3, 9, 11, 12, 12, 16, 19]$:

$a[j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$
	3	9	11	12	12	16	19

Ricerca di $m = 10$.

- Chiamata 1**, `RecursiveBinarySearch(a, 0, 6, 10)`: $left = 0 \leq right = 6$; $mid = 0 + (6 - 0)/2 = 3$; $a[3] = 12 > 10 \Rightarrow$ chiamata `RecursiveBinarySearch(a, 0, 2, 10)`.

- **Chiamata 2**, `RecursiveBinarySearch(a, 0, 2, 10)`: $left = 0 \leq right = 2$; $mid = 0 + (2 - 0) / 2 = 1$; $a[1] = 9 < 10 \Rightarrow$ chiamata `RecursiveBinarySearch(a, 2, 2, 10)`.
- **Chiamata 3**, `RecursiveBinarySearch(a, 2, 2, 10)`: $left = 2 \leq right = 2$; $mid = 2 + (2 - 2) / 2 = 2$; $a[2] = 11 > 10 \Rightarrow$ chiamata `RecursiveBinarySearch(a, 2, 1, 10)`.
- **Chiamata 4**, `RecursiveBinarySearch(a, 2, 1, 10)`: $left = 2 > right = 1$: l'intervallo è vuoto $\Rightarrow$ restituisce -1 .

Il valore -1 viene restituito alla chiamata 3, che lo restituisce alla chiamata 2, che lo restituisce alla chiamata 1, che lo restituisce a `main`. Nel momento della chiamata più profonda, 4 chiamate sono contemporaneamente nella pila delle chiamate.

Ricerca di $m = 19$.

- **Chiamata 1**, `RecursiveBinarySearch(a, 0, 6, 19)`: $mid = 3$; $a[3] = 12 < 19 \Rightarrow$ chiamata `RecursiveBinarySearch(a, 4, 6, 19)`.
- **Chiamata 2**, `RecursiveBinarySearch(a, 4, 6, 19)`: $mid = 4 + (6 - 4) / 2 = 5$; $a[5] = 16 < 19 \Rightarrow$ chiamata `RecursiveBinarySearch(a, 6, 6, 19)`.
- **Chiamata 3**, `RecursiveBinarySearch(a, 6, 6, 19)`: $mid = 6$; $a[6] = 19 = m \Rightarrow$ restituisce 6.

Il valore 6 viene restituito alla chiamata 2, che lo restituisce alla chiamata 1, che lo restituisce a `main`. L'elemento viene trovato all'indice 6 (posizione 7 nello pseudocodice, dove l'array è indicizzato da 1).

Punti chiave sulla ricerca binaria ricorsiva.

- Una chiamata ricorsiva su (al più) metà dell'intervallo: l'albero delle chiamate è un cammino, con al più $\lfloor \log_2 n \rfloor + 2$ chiamate.
- $S(n) = S(\lfloor n/2 \rfloor) + 5$, cioè $S(n) = 5(\lfloor \log_2 n \rfloor + 1) + 2 = \Theta(\log n)$ (caso 2 del Master theorem con $a = 1$).
- L'algoritmo è ricorsivo in coda: eliminando la ricorsione si ottiene l'algoritmo iterativo `BinarySearch`, con $\Theta(1)$ di memoria aggiuntiva invece di $\Theta(\log n)$.

17.4 Il problema del sotto-array di somma massima

L'ultimo problema del capitolo mostra il divide et impera al lavoro su un problema meno ovvio, in cui il passo *combina* è la parte interessante. Mostra anche che un algoritmo ricorsivo che migliora la forza bruta non è necessariamente il migliore possibile.

17.4.1 Motivazione: comprare e vendere un'azione

- Supponiamo di conoscere in anticipo il prezzo di un'azione per i giorni successivi. Possiamo comprare un'unità di azione **una volta** e venderla **una volta**, in un giorno successivo. Vogliamo massimizzare il profitto (prezzo di vendita meno prezzo di acquisto).
- Questo è un esempio¹: $p[d]$ è il prezzo al giorno d , e $a[d] = p[d] - p[d-1]$ è la **variazione giornaliera** del prezzo.

giorno d	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
prezzo $p[d]$	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
variazione $a[d]$		13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

- Se compriamo al giorno $l-1$ e vendiamo al giorno $r \geq l$, il profitto è

$$p[r] - p[l-1] = (p[l] - p[l-1]) + (p[l+1] - p[l]) + \dots + (p[r] - p[r-1]) = a[l] + a[l+1] + \dots + a[r],$$

cioè la somma delle variazioni giornaliere consecutive $a[l..r]$. Quindi, massimizzare il profitto equivale a trovare la porzione consecutiva dell'array $a[1..16]$ con la somma più grande.

- Si noti che **non** basta comprare al prezzo più basso o vendere al prezzo più alto: il prezzo più basso (63, giorno 7) viene dopo il prezzo più alto (113, giorno 1). In questo esempio la scelta migliore è comprare al giorno 7 (prezzo 63) e vendere al giorno 11 (prezzo 106), con profitto

$$106 - 63 = 43 = a[8] + a[9] + a[10] + a[11] = 18 + 20 - 7 + 12.$$

¹L'esempio è tratto dal libro di testo di T. H. Cormen, C. E. Leiserson, R. L. Rivest e C. Stein, *Introduction to Algorithms*, Sezione 4.1.

17.4.2 Il problema

Dato un array $a[1..n]$ di n numeri interi, consideriamo il problema di trovare due indici $l \leq r$ tali che la somma

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

sia massima. Il sotto-array $a[l..r]$ si chiama **sotto-array di somma massima** di a .

- Un sotto-array è sempre **non vuoto** ($l \leq r$) e formato da elementi **consecutivi**.
- Il problema è interessante solo se l'array contiene numeri negativi: se tutti gli elementi sono non negativi, l'intero array $a[1..n]$ è un sotto-array di somma massima. Se tutti gli elementi sono negativi, un sotto-array di somma massima è formato da un solo elemento, il più grande.
- Un sotto-array di somma massima non è necessariamente unico: ne cerchiamo **uno**.
- Il numero di sotto-array di $a[1..n]$ è il numero di coppie (l, r) con $1 \leq l \leq r \leq n$:

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

17.4.3 Un algoritmo di forza bruta

- L'algoritmo più semplice considera tutti gli $n(n+1)/2$ sotto-array $a[l..r]$ e tiene il migliore. Per un l fissato, le somme di $a[l..r]$ per $r = l, l+1, \dots, n$ si calcolano in modo incrementale: la somma di $a[l..r]$ è la somma di $a[l..r-1]$ più $a[r]$.
- Questo è lo pseudocodice dell'algoritmo **BruteForceMaximumSubarray**:

Algorithm: BruteForceMaximumSubarray**Input** : Un array $a[1..n]$ di n numeri interi**Output**: Indici $bl \leq br$ di un sotto-array di somma massima e la sua somma $best$

```

1 function BruteForceMaximumSubarray( $a, n$ )
2    $best := a[1]$ 
3    $bl := 1$ 
4    $br := 1$ 
5    $l := 1$ 
6   while  $l \leq n$  do
7      $sum := 0$ 
8      $r := l$ 
9     while  $r \leq n$  do
10       $sum \leftarrow sum + a[r]$  // somma di  $a[l..r]$ 
11      if  $sum > best$  then
12         $best \leftarrow sum$ 
13         $bl \leftarrow l$ 
14         $br \leftarrow r$ 
15      end
16       $r \leftarrow r + 1$ 
17    end
18     $l \leftarrow l + 1$ 
19  end
20  return ( $bl, br, best$ )
21 end

```

Esempio 17.11: algoritmo di forza bruta

Consideriamo $a = [-2, 5, -1, 3]$. L'algoritmo parte con $best = a[1] = -2$ ed esamina i $4 \cdot 5/2 = 10$ sotto-array in quest'ordine:

l	r	sum	azione
1	1	-2	$sum \leq best = -2$
1	2	3	$sum > best \Rightarrow best = 3, bl = 1, br = 2$
1	3	2	
1	4	5	$sum > best \Rightarrow best = 5, bl = 1, br = 4$
2	2	5	$sum \leq best = 5$
2	3	4	
2	4	7	$sum > best \Rightarrow best = 7, bl = 2, br = 4$
3	3	-1	
3	4	2	
4	4	3	

L'algoritmo restituisce (2, 4, 7): il sotto-array di somma massima è $a[2..4] = [5, -1, 3]$ con somma 7.

17.4.3.1 Tempo di esecuzione nel caso peggiore

- Sia $S(n)$ il numero di passi eseguiti da `BruteForceMaximumSubarray` nel caso peggiore, dove n è la dimensione dell'array.
- L'esecuzione delle righe 12–14 dipende dall'input. Definiamo una funzione $t(l, r) \in \{0, 1\}$, per $1 \leq l \leq r \leq n$, dove $t(l, r) = 1$ se le righe 12–14 vengono eseguite quando il ciclo interno esamina la coppia (l, r) , e $t(l, r) = 0$ altrimenti.
- Per un l fissato, il ciclo interno esegue $n - l + 1$ iterazioni, e la sua condizione viene verificata $n - l + 2$ volte. Quindi

$$\sum_{l=1}^n (n - l + 1) = \frac{n(n+1)}{2}, \quad \sum_{l=1}^n (n - l + 2) = \frac{n(n+1)}{2} + n.$$

- Il numero di volte che ciascuna riga di codice viene eseguita è:
 - Le righe 2, 3, 4, 5 vengono eseguite 1 volta ciascuna
 - La riga 6 viene eseguita $n + 1$ volte
 - Le righe 7 e 8 vengono eseguite n volte ciascuna
 - La riga 9 viene eseguita $\frac{n(n+1)}{2} + n$ volte
 - Le righe 10, 11 e 16 vengono eseguite $\frac{n(n+1)}{2}$ volte ciascuna
 - Le righe 12, 13, 14 vengono eseguite $\sum_{l=1}^n \sum_{r=l}^n t(l, r)$ volte ciascuna
 - La riga 18 viene eseguita n volte
 - La riga 20 viene eseguita 1 volta
- Quindi, il numero totale di passi è

$$\begin{aligned} & 4 + (n + 1) + 2n + \frac{n(n+1)}{2} + n + 3 \frac{n(n+1)}{2} + n + 1 + 3 \sum_{l=1}^n \sum_{r=l}^n t(l, r) \\ &= 2n^2 + 7n + 6 + 3 \sum_{l=1}^n \sum_{r=l}^n t(l, r). \end{aligned}$$

Nell'Esempio 17.11 ci sono 3 aggiornamenti e l'algoritmo esegue $2 \cdot 16 + 7 \cdot 4 + 6 + 3 \cdot 3 = 75$ passi.

- Poiché $0 \leq t(l, r) \leq 1$, la somma doppia è compresa tra 0 e $n(n+1)/2$. Quindi, per ogni input di dimensione n , il numero di passi è compreso tra

$$2n^2 + 7n + 6 \quad \text{e} \quad 2n^2 + 7n + 6 + \frac{3n(n+1)}{2} = \frac{7}{2}n^2 + \frac{17}{2}n + 6,$$

e lo stesso vale per il numero di passi nel caso peggiore $S(n)$. Entrambi i limiti sono $\Theta(n^2)$; per esempio,

$$\lim_{n \rightarrow \infty} \frac{2n^2 + 7n + 6}{n^2} = 2 \in \mathbb{R}_{>0}, \quad \lim_{n \rightarrow \infty} \frac{\frac{7}{2}n^2 + \frac{17}{2}n + 6}{n^2} = \frac{7}{2} \in \mathbb{R}_{>0}.$$

Quindi,

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

Si noti che, per **ogni** input (non solo nel caso peggiore), l'algoritmo esegue $\Theta(n^2)$ passi, poiché esamina sempre tutti gli $n(n+1)/2$ sotto-array.

- Sotto l'ipotesi che ogni elemento usi un numero costante di bit, $|I| = \Theta(n)$, e quindi $S(n) = \Theta(|I|^2)$.

Se tutti i valori in ingresso usano un numero costante fissato di bit, allora il tempo di esecuzione nel caso peggiore dell'algoritmo `BruteForceMaximumSubarray` è

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

Quindi, l'algoritmo di forza bruta è quadratico nella dimensione dell'input.

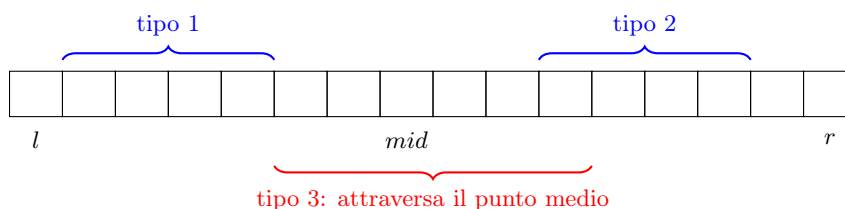
17.4.4 Un algoritmo divide et impera

- Progettiamo ora un algoritmo ricorsivo basato sull'approccio **divide et impera**, come per il Merge Sort. Risolviamo il problema su un sotto-array $a[l..r]$, e lo chiamiamo inizialmente con $l = 1$ e $r = n$.

- **Divide**: si suddivide $a[l..r]$ in due metà $a[l..mid]$ e $a[mid + 1..r]$, dove

$$mid = \left\lfloor \frac{l+r}{2} \right\rfloor \quad \left(= l + \left\lfloor \frac{r-l}{2} \right\rfloor \right).$$

- Ogni sotto-array $a[i..j]$ di $a[l..r]$ (con $l \leq i \leq j \leq r$) si trova in una e una sola di queste tre situazioni:
 1. si trova interamente nella metà sinistra: $l \leq i \leq j \leq mid$;
 2. si trova interamente nella metà destra: $mid + 1 \leq i \leq j \leq r$;
 3. **attraversa il punto medio**: $l \leq i \leq mid < j \leq r$.
- **Impera**: un sotto-array di somma massima di tipo 1 e uno di tipo 2 si trovano **ricorsivamente**, poiché sono sotto-array di somma massima di $a[l..mid]$ e $a[mid + 1..r]$, rispettivamente.
- **Combina**: un sotto-array di somma massima di tipo 3 si trova con una procedura non ricorsiva, `FindMaxCrossingSubarray`, descritta qui sotto. La risposta è il migliore dei tre sotto-array.



17.4.4.1 Sotto-array di somma massima che attraversa il punto medio

- Un sotto-array che attraversa il punto medio è formato da due parti: $a[i..mid]$, che **termina** in mid , e $a[mid + 1..j]$, che **inizia** in $mid + 1$:

$$\sum_{k=i}^j a[k] = \sum_{k=i}^{mid} a[k] + \sum_{k=mid+1}^j a[k].$$

Il primo termine dipende solo da i , il secondo solo da j : le due parti si possono scegliere **indipendentemente**. Quindi, basta trovare il miglior i (scorrendo la metà sinistra da mid fino a l) e il miglior j (scorrendo la metà destra da $mid + 1$ fino a r).

- Questo è lo pseudocodice dell'algoritmo `FindMaxCrossingSubarray` (dallo stesso libro di testo dell'esempio sull'azione):

Algorithm: FindMaxCrossingSubarray

Input : Un array a e tre indici $l \leq mid < r$

Output: Indici $maxleft \leq mid < maxright$ di un sotto-array di somma massima che attraversa il punto medio, e la sua somma

```

1 function FindMaxCrossingSubarray( $a, l, mid, r$ )
2   leftsum :=  $-\infty$ 
3   sum := 0
4   for  $i := mid$  downto  $l$  do
5     sum  $\leftarrow$  sum +  $a[i]$                                 // somma di  $a[i..mid]$ 
6     if sum > leftsum then
7       leftsum  $\leftarrow$  sum
8       maxleft  $\leftarrow$   $i$ 
9   end
10  end
11  rightsum :=  $-\infty$ 
12  sum  $\leftarrow$  0
13  for  $j := mid + 1$  to  $r$  do
14    sum  $\leftarrow$  sum +  $a[j]$                                 // somma di  $a[mid + 1..j]$ 
15    if sum > rightsum then
16      rightsum  $\leftarrow$  sum
17      maxright  $\leftarrow$   $j$ 
18    end
19  end
20  return ( $maxleft, maxright, leftsum + rightsum$ )
21 end
  
```

Lemma 17.9: sotto-array che attraversa il punto medio

Sia $l \leq mid < r$. `FindMaxCrossingSubarray( $a, l, mid, r$ )` restituisce $(maxleft, maxright, s)$ con $l \leq maxleft \leq mid < maxright \leq r$ e

$$s = \sum_{k=maxleft}^{maxright} a[k] = \max \left\{ \sum_{k=i}^j a[k] : l \leq i \leq mid < j \leq r \right\}.$$

Inoltre, esegue $\Theta(n)$ passi, dove $n = r - l + 1$.

Dimostrazione. Ciclo sinistro. Alla fine dell'iterazione con indice i , la variabile sum è uguale a $L(i) = \sum_{k=i}^{mid} a[k]$, e $leftsum$ è il massimo tra $L(mid), L(mid-1), \dots, L(i)$, raggiunto in $maxleft$ (per induzione sulle iterazioni: a ogni iterazione sum viene confrontato con il miglior valore trovato finora, che inizialmente è $-\infty$, quindi il primo confronto ha sempre successo). Alla fine del ciclo,

$$leftsum = \max_{l \leq i \leq mid} L(i) = L(maxleft).$$

Ciclo destro. Allo stesso modo, con $R(j) = \sum_{k=mid+1}^j a[k]$,

$$rightsum = \max_{mid+1 \leq j \leq r} R(j) = R(maxright).$$

Combinazione. Per ogni sotto-array che attraversa il punto medio, $\sum_{k=i}^j a[k] = L(i) + R(j)$. Poiché i e j si possono scegliere indipendentemente,

$$\max_{l \leq i \leq mid < j \leq r} (L(i) + R(j)) = \max_{l \leq i \leq mid} L(i) + \max_{mid < j \leq r} R(j) = leftsum + rightsum,$$

che è il valore restituito, raggiunto da $a[maxleft..maxright]$.

Tempo di esecuzione. I due cicli eseguono $(mid - l + 1) + (r - mid) = r - l + 1 = n$ iterazioni in totale, ciascuna con un numero costante di passi (tra 3 e 5 righe, contando il test del ciclo), più un numero costante di passi fuori dai cicli. Quindi il numero di passi è compreso tra $c_1 n$ e $c_2 n$ per due costanti $c_1, c_2 > 0$, cioè $\Theta(n)$. $\square$

- Si noti che il sotto-array che attraversa il punto medio contiene sempre sia $a[mid]$ sia $a[mid + 1]$, anche se sono negativi: i sotto-array di tipo 1 e 2 si occupano degli altri casi.

17.4.4.2 L'algoritmo ricorsivo

- Questo è lo pseudocodice dell'algoritmo `FindMaximumSubarray`. Per risolvere il problema sull'intero array chiamiamo `FindMaximumSubarray(a, 1, n)`.

Algorithm: FindMaximumSubarray**Input** : Un array a e due indici $l \leq r$ **Output**: Indici $i \leq j$ di un sotto-array di somma massima di $a[l..r]$ e la sua somma

```

1 function FindMaximumSubarray( $a, l, r$ )
2   if  $l = r$  then
3     | return ( $l, r, a[l]$ )           // Caso base: un elemento
4   end
5    $mid := \lfloor (l + r) / 2 \rfloor$ 
6   ( $ll, lr, lsum$ ) := FindMaximumSubarray( $a, l, mid$ )
7   ( $rl, rr, rsum$ ) := FindMaximumSubarray( $a, mid + 1, r$ )
8   ( $cl, cr, csum$ ) := FindMaxCrossingSubarray( $a, l, mid, r$ )
9   if  $lsum \geq rsum$  and  $lsum \geq csum$  then
10    | return ( $ll, lr, lsum$ )
11  end
12  if  $rsum \geq lsum$  and  $rsum \geq csum$  then
13    | return ( $rl, rr, rsum$ )
14  end
15  return ( $cl, cr, csum$ )
16 end

```

- Spiegazione dell'algoritmo:
 - **Righe 2–3**: se il sotto-array ha un solo elemento, il suo unico sotto-array è $a[l..l]$ (**caso base**).
 - **Riga 5**: **divide** il sotto-array in $a[l..mid]$ e $a[mid + 1..r]$.
 - **Righe 6–7**: **impera**, cioè trova ricorsivamente un sotto-array di somma massima in ciascuna metà.
 - **Riga 8**: trova un sotto-array di somma massima che attraversa il punto medio.
 - **Righe 9–15**: **combina**, cioè restituisce il migliore dei tre (in caso di parità, si preferisce quello di sinistra, poi quello di destra).
- A differenza della ricerca binaria ricorsiva, ogni chiamata su almeno due elementi effettua **due** chiamate ricorsive: l'albero delle chiamate è un albero binario con n foglie (una per ogni elemento) e $n - 1$ nodi interni, cioè $2n - 1$ chiamate in totale, esattamente come per RecursiveSum (Osservazione 17.2).

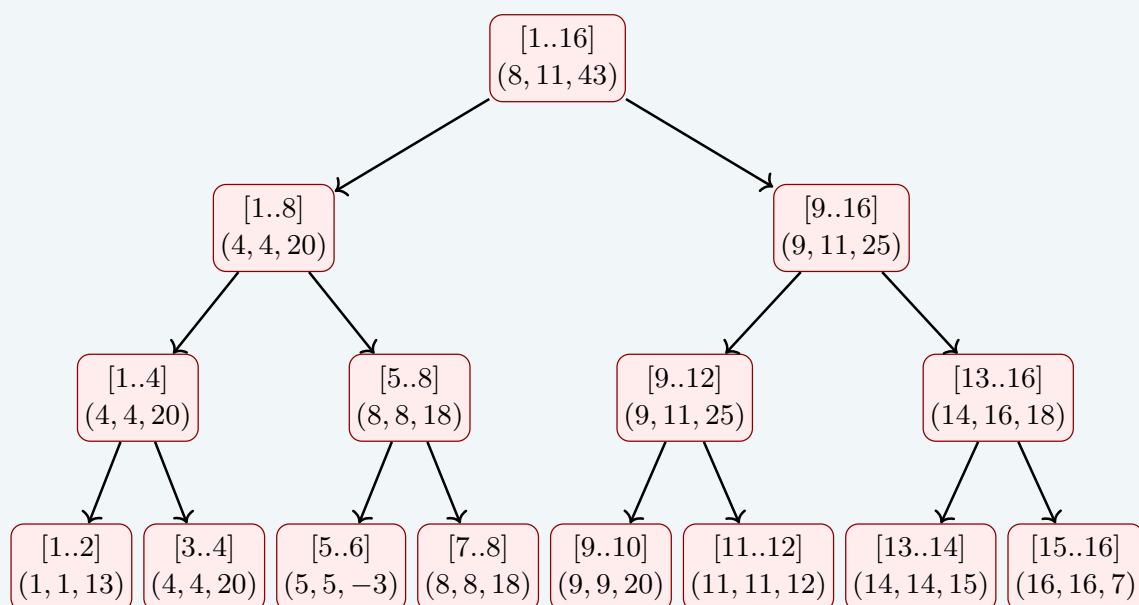
17.4.4.3 Esempio**Esempio 17.12: divide et impera sull'esempio dell'azione in borsa**

Consideriamo l'array delle variazioni giornaliere dell'esempio dell'azione:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$a[i]$	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

- **Passo 1: l'albero delle chiamate.** Chiamiamo FindMaximumSubarray($a, 1, 16$). Nel seguente albero, ogni nodo è una chiamata su $a[l..r]$, scritta come $[l..r]$, insieme

alla terna (i, j, s) che restituisce. Le 16 foglie (chiamate su un elemento $[k..k]$, che restituiscono $(k, k, a[k])$) non sono disegnate.



- **Passo 2: i risultati delle chiamate.** Per ogni chiamata su almeno due elementi, riportiamo le terne restituite dalla chiamata ricorsiva sinistra, dalla chiamata ricorsiva destra e da FindMaxCrossingSubarray, e la terna restituita (la migliore delle tre).

chiamata	<i>mid</i>	sinistra	destra	attraversante	restituita
[1..16]	8	(4, 4, 20)	(9, 11, 25)	(8, 11, 43)	(8, 11, 43)
[1..8]	4	(4, 4, 20)	(8, 8, 18)	(4, 5, 17)	(4, 4, 20)
[9..16]	12	(9, 11, 25)	(14, 16, 18)	(9, 16, 16)	(9, 11, 25)
[1..4]	2	(1, 1, 13)	(4, 4, 20)	(1, 4, 5)	(4, 4, 20)
[5..8]	6	(5, 5, -3)	(8, 8, 18)	(6, 8, -21)	(8, 8, 18)
[9..12]	10	(9, 9, 20)	(11, 11, 12)	(9, 11, 25)	(9, 11, 25)
[13..16]	14	(14, 14, 15)	(16, 16, 7)	(14, 16, 18)	(14, 16, 18)
[1..2]	1	(1, 1, 13)	(2, 2, -3)	(1, 2, 10)	(1, 1, 13)
[3..4]	3	(3, 3, -25)	(4, 4, 20)	(3, 4, -5)	(4, 4, 20)
[5..6]	5	(5, 5, -3)	(6, 6, -16)	(5, 6, -19)	(5, 5, -3)
[7..8]	7	(7, 7, -23)	(8, 8, 18)	(7, 8, -5)	(8, 8, 18)
[9..10]	9	(9, 9, 20)	(10, 10, -7)	(9, 10, 13)	(9, 9, 20)
[11..12]	11	(11, 11, 12)	(12, 12, -5)	(11, 12, 7)	(11, 11, 12)
[13..14]	13	(13, 13, -22)	(14, 14, 15)	(13, 14, -7)	(14, 14, 15)
[15..16]	15	(15, 15, -4)	(16, 16, 7)	(15, 16, 3)	(16, 16, 7)

- **Passo 3: i calcoli dei sotto-array che attraversano il punto medio per le due chiamate del livello 1.** Chiamata [1..8], *mid* = 4:

i	$a[i]$	sum	$leftsum$	$maxleft$
4	20	20	20	4
3	-25	-5	20	4
2	-3	-8	20	4
1	13	5	20	4

j	$a[j]$	sum	$rightsum$	$maxright$
5	-3	-3	-3	5
6	-16	-19	-3	5
7	-23	-42	-3	5
8	18	-24	-3	5

Il sotto-array che attraversa il punto medio è $a[4..5]$ con somma $20 + (-3) = 17$. Chiamata $[9..16]$, $mid = 12$:

i	$a[i]$	sum	$leftsum$	$maxleft$
12	-5	-5	-5	12
11	12	7	7	11
10	-7	0	7	11
9	20	20	20	9

j	$a[j]$	sum	$rightsum$	$maxright$
13	-22	-22	-22	13
14	15	-7	-7	14
15	-4	-11	-7	14
16	7	-4	-4	16

Il sotto-array che attraversa il punto medio è $a[9..16]$ con somma $20 + (-4) = 16$.

- **Passo 4: il calcolo del sotto-array che attraversa il punto medio per la prima chiamata $[1..16]$, $mid = 8$:**

i	$a[i]$	sum	$leftsum$	$maxleft$
8	18	18	18	8
7	-23	-5	18	8
6	-16	-21	18	8
5	-3	-24	18	8
4	20	-4	18	8
3	-25	-29	18	8
2	-3	-32	18	8
1	13	-19	18	8

j	$a[j]$	sum	$rightsum$	$maxright$
9	20	20	20	9
10	-7	13	20	9
11	12	25	25	11
12	-5	20	25	11
13	-22	-2	25	11
14	15	13	25	11
15	-4	9	25	11
16	7	16	25	11

Il sotto-array che attraversa il punto medio è $a[8..11]$ con somma $18 + 25 = 43$.

Nella prima chiamata, la metà sinistra dà 20, la metà destra dà 25 e il sotto-array che attraversa il punto medio dà 43: l'algoritmo restituisce $(8, 11, 43)$, cioè si compra al giorno 7 e si vende al giorno 11, con profitto 43. Si noti che il sotto-array ottimo viene trovato solo all'ultimo passo, da **FindMaxCrossingSubarray**: non si trova né nella metà sinistra né nella metà destra.

17.4.5 Correttezza

Teorema 17.10: correttezza di FindMaximumSubarray

Per ogni $l \leq r$, FindMaximumSubarray(a, l, r) restituisce (i, j, s) con $l \leq i \leq j \leq r$ e

$$s = \sum_{k=i}^j a[k] = \max \left\{ \sum_{k=i'}^{j'} a[k] : l \leq i' \leq j' \leq r \right\}.$$

Dimostrazione. Procediamo per induzione forte sul numero di elementi $n = r - l + 1 \geq 1$.

Caso base $n = 1$. Allora $l = r$, l'unico sotto-array è $a[l..l]$, e l'algoritmo restituisce $(l, l, a[l])$, il che è corretto.

Passo induttivo. Sia $n \geq 2$ e supponiamo (ipotesi induttiva) che l'enunciato valga per ogni sotto-array con $n' < n$ elementi. Poiché $l < r$, si ha $l \leq mid < r$, e le due metà $a[l..mid]$ e $a[mid+1..r]$ hanno $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$ elementi, entrambi almeno 1 e minori di n . Per l'ipotesi induttiva:

- $lsum$ è la somma massima di un sotto-array di tipo 1 (contenuto in $a[l..mid]$), raggiunta da $a[l..lr]$;
- $rsum$ è la somma massima di un sotto-array di tipo 2 (contenuto in $a[mid+1..r]$), raggiunta da $a[r..rr]$.

Per il Lemma 17.9, $csum$ è la somma massima di un sotto-array di tipo 3 (che attraversa il punto medio), raggiunta da $a[cl..cr]$.

Ogni sotto-array di $a[l..r]$ è di tipo 1, 2 o 3. Quindi la somma massima di un sotto-array di $a[l..r]$ è

$$\max\{lsum, rsum, csum\},$$

e le righe 9–15 restituiscono esattamente una terna che raggiunge questo massimo. $\square$

17.4.6 Tempo di esecuzione nel caso peggiore

17.4.6.1 L'equazione di ricorrenza

- Sia $S(n)$ il numero di passi eseguiti da FindMaximumSubarray nel caso peggiore su un sotto-array di $n = r - l + 1$ elementi.
- Se $n = 1$, vengono eseguite solo le righe 2–3: $S(1) = c_1$, una costante.
- If $n \geq 2$:
 - le righe 2, 5 e 9–15 eseguono un numero costante di passi;
 - le chiamate ricorsive delle righe 6 e 7 eseguono $S(\lceil n/2 \rceil)$ e $S(\lfloor n/2 \rfloor)$ passi;
 - la riga 8 esegue $\Theta(n)$ passi (Lemma 17.9).

- Quindi,

$$S(n) = \begin{cases} c_1 & \text{if } n = 1, \\ S(\lceil n/2 \rceil) + S(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n \geq 2. \end{cases}$$

Ignorando le parti intere inferiori e superiori (che non cambiano la soluzione asintotica), otteniamo la stessa equazione di ricorrenza del Merge Sort:

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

- Si noti che il tempo di esecuzione non dipende dai valori nell'array, ma solo da n : ogni input di dimensione n è un caso peggiore.

17.4.6.2 Soluzione con il Master theorem

- **Identificare $a, b, f(n)$:** $a = 2, b = 2, f(n) = \Theta(n)$.
- **Calcolare $n^{\log_b a} = n^{\log_2 2} = n$.**
- **Confrontare:** $f(n) = \Theta(n) = \Theta(n^{\log_2 2} \log^0 n)$.
- **Verificare l'ipotesi:** siamo nel caso 2 del Master theorem con $k = 0$.
- **Concludere:**

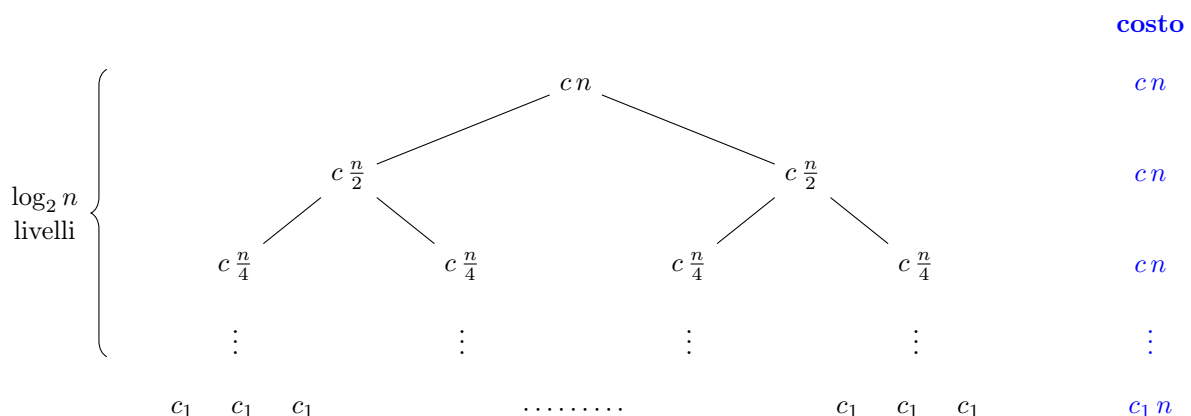
$$S(n) = \Theta(n^{\log_2 2} \log^{0+1} n) = \Theta(n \log n).$$

17.4.6.3 Soluzione con l'albero di ricorsione

- Sia $n = 2^k$ una potenza di 2, e sia cn il costo della parte non ricorsiva di una chiamata su $n \geq 2$ elementi. Nell'albero di ricorsione, il livello h (per $h = 0, 1, \dots, k-1$) contiene 2^h chiamate su $n/2^h$ elementi ciascuna, con costo totale

$$2^h \cdot c \frac{n}{2^h} = cn.$$

L'ultimo livello ($h = k = \log_2 n$) contiene le n foglie, con costo totale $c_1 n$.



- Sommando su tutti i livelli:

$$S(n) = \underbrace{cn + cn + \dots + cn}_{\log_2 n \text{ volte}} + c_1 n = cn \log_2 n + c_1 n = \Theta(n \log n).$$

- L'albero di ricorsione spiega anche il risultato: ci sono $\log_2 n$ livelli e ogni livello costa $\Theta(n)$, perché i calcoli dei sotto-array che attraversano il punto medio, per le chiamate di un livello, scorrono tutti insieme l'intero array una sola volta.

17.4.6.4 Tempo di esecuzione nel caso peggiore in funzione della dimensione dell'input

- Sotto l'ipotesi che ogni elemento usi un numero costante di bit, $|I| = \Theta(n)$, e anche $\log n = \Theta(\log |I|)$ (come mostrato per la ricerca binaria ricorsiva). Quindi

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

Se tutti i valori in ingresso usano un numero costante fissato di bit, allora il tempo di esecuzione nel caso peggiore dell'algoritmo `FindMaximumSubarray` è

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

Quindi, l'algoritmo divide et impera è asintoticamente più veloce dell'algoritmo di forza bruta, il cui tempo di esecuzione nel caso peggiore è $\Theta(|I|^2)$.

17.4.7 Osservazione: un algoritmo iterativo lineare

- L'algoritmo divide et impera è molto meglio dell'algoritmo di forza bruta. Tuttavia, il problema del sotto-array di somma massima si può risolvere ancora più velocemente, con un semplice algoritmo **iterativo** dovuto a J. Kadane, che scorre l'array una **sola** volta.
- **Idea:** per ogni posizione i , sia $M(i)$ la somma massima di un sotto-array che **termina in** posizione i . Un sotto-array che termina in i è o il solo $a[i]$, oppure un sotto-array che termina in $i - 1$ esteso con $a[i]$. Quindi,

$$M(1) = a[1], \quad M(i) = a[i] + \max\{0, M(i-1)\} \quad (i \geq 2).$$

A parole: se il miglior sotto-array che termina in $i - 1$ ha somma negativa, conviene **ripartire** da $a[i]$; altrimenti, conviene **estenderlo**. La risposta al problema è il massimo tra $M(1), \dots, M(n)$.

- Questo è lo pseudocodice dell'algoritmo **Kadane**. La variabile *cur* memorizza $M(i)$ e *cl* il primo indice del sotto-array corrispondente.

Algorithm: Kadane**Input** : Un array $a[1..n]$ di n numeri interi**Output**: Indici $bl \leq br$ di un sotto-array di somma massima e la sua somma $best$

```

1 function Kadane( $a, n$ )
2    $best := a[1]; bl := 1; br := 1$ 
3    $cur := a[1]; cl := 1$ 
4    $i := 2$ 
5   while  $i \leq n$  do
6     if  $cur < 0$  then
7        $cur \leftarrow a[i]; cl \leftarrow i$            // ripartire
8     else
9        $cur \leftarrow cur + a[i]$                  // estendere
10    end
11    if  $cur > best$  then
12       $best \leftarrow cur; bl \leftarrow cl; br \leftarrow i$ 
13    end
14     $i \leftarrow i + 1$ 
15  end
16  return ( $bl, br, best$ )
17 end

```

Lemma 17.11: invariante di ciclo di Kadane

All'inizio di ogni iterazione del ciclo **while** (e quando il ciclo termina), con indice corrente i :

1. $cur = \sum_{k=cl}^{i-1} a[k]$ è la somma massima di un sotto-array che **termina in** posizione $i - 1$;
2. $best = \sum_{k=bl}^{br} a[k]$ è la somma massima di un sotto-array di $a[1..i - 1]$.

Dimostrazione. **Inizializzazione** ($i = 2$). L'unico sotto-array che termina in 1 è $a[1..1]$, ed è anche l'unico sotto-array di $a[1..1]$: $cur = best = a[1]$ con $cl = bl = br = 1$.

Mantenimento. Supponiamo che l'invariante valga all'inizio dell'iterazione con indice i . Un sotto-array che termina in i è o $a[i..i]$, con somma $a[i]$, oppure $a[k..i]$ con $k < i$, la cui somma è $\sum_{h=k}^{i-1} a[h] + a[i] \leq cur + a[i]$ (per la parte 1), con uguaglianza per $k = cl$. Quindi la somma massima di un sotto-array che termina in i è $\max\{a[i], cur + a[i]\}$, che è $a[i]$ se $cur < 0$ e $cur + a[i]$ altrimenti: è esattamente ciò che calcolano le righe 6–9. Quindi la parte 1 vale per $i + 1$.

Un sotto-array di $a[1..i]$ o è un sotto-array di $a[1..i - 1]$ (somma migliore: $best$, per la parte 2) oppure termina in i (somma migliore: il nuovo cur). Le righe 11–12 mantengono il maggiore dei due, quindi la parte 2 vale per $i + 1$.

Terminazione. Il ciclo termina con $i = n + 1$: per la parte 2, $best$ è la somma massima di un sotto-array di $a[1..n]$, raggiunta da $a[bl..br]$. $\square$

- **Tempo di esecuzione.** Il ciclo `while` esegue $n - 1$ iterazioni, ciascuna con un numero costante di passi. Quindi $S(n) = \Theta(n)$ e, se ogni elemento usa un numero costante di bit, $T(|I|) = \Theta(|I|)$. Poiché qualunque algoritmo deve leggere tutti gli elementi dell'array, nessun algoritmo può essere asintoticamente più veloce.

Esempio 17.13: algoritmo di Kadane sull'esempio dell'azione

Esecuzione di Kadane su

$$a = [13, -3, -25, 20, -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4, 7].$$

Ogni riga riporta i valori alla fine dell'iterazione con indice i (riga $i = 1$: inizializzazione).

i	$a[i]$	azione	cur	cl	$best$	(bl, br)
1	13	inizializzazione	13	1	13	(1, 1)
2	-3	estensione	10	1	13	(1, 1)
3	-25	estensione	-15	1	13	(1, 1)
4	20	ripartenza ($cur < 0$)	20	4	20	(4, 4)
5	-3	estensione	17	4	20	(4, 4)
6	-16	estensione	1	4	20	(4, 4)
7	-23	estensione	-22	4	20	(4, 4)
8	18	ripartenza ($cur < 0$)	18	8	20	(4, 4)
9	20	estensione	38	8	38	(8, 9)
10	-7	estensione	31	8	38	(8, 9)
11	12	estensione	43	8	43	(8, 11)
12	-5	estensione	38	8	43	(8, 11)
13	-22	estensione	16	8	43	(8, 11)
14	15	estensione	31	8	43	(8, 11)
15	-4	estensione	27	8	43	(8, 11)
16	7	estensione	34	8	43	(8, 11)

L'algoritmo restituisce (8, 11, 43), lo stesso sotto-array di somma massima trovato dall'algoritmo divide et impera, dopo un'unica scansione dell'array.

17.4.8 Implementazione in C++

- Questa è un'implementazione in C++ dell'algoritmo divide et impera. L'array è indicizzato da 0, e le tre quantità restituite dalle funzioni sono memorizzate in una `struct`. Il valore $-\infty$ è sostituito da `INT_MIN`, il più piccolo `int`. Per risolvere il problema su un array di n elementi chiamiamo `FindMaximumSubarray(a, 0, n-1)`.

```

1 #include <climits>
2
3 struct Subarray
4 {
5     int low;    // primo indice
6     int high;   // ultimo indice
7     int sum;    // somma di a[low..high]
8 };
9
10 Subarray FindMaxCrossingSubarray(int *a, int l, int mid, int r)
11 {

```

```

12  int leftSum = INT_MIN, sum = 0, maxLeft = mid;
13  for (int i = mid; i >= l; i--)
14  {
15      sum = sum + a[i];
16      if (sum > leftSum)
17      {
18          leftSum = sum;
19          maxLeft = i;
20      }
21  }
22
23  int rightSum = INT_MIN, maxRight = mid + 1;
24  sum = 0;
25  for (int j = mid + 1; j <= r; j++)
26  {
27      sum = sum + a[j];
28      if (sum > rightSum)
29      {
30          rightSum = sum;
31          maxRight = j;
32      }
33  }
34
35  Subarray result = {maxLeft, maxRight, leftSum + rightSum};
36  return result;
37 }
38
39 Subarray FindMaximumSubarray(int *a, int l, int r)
40 {
41     if (l == r)
42     {
43         Subarray result = {l, r, a[l]}; // Caso base: un solo elemento
44         return result;
45     }
46
47     int mid = (l + r) / 2;
48
49     Subarray left = FindMaximumSubarray(a, l, mid);
50     Subarray right = FindMaximumSubarray(a, mid + 1, r);
51     Subarray cross = FindMaxCrossingSubarray(a, l, mid, r);
52
53     if (left.sum >= right.sum && left.sum >= cross.sum)
54     {
55         return left;
56     }
57     if (right.sum >= left.sum && right.sum >= cross.sum)
58     {
59         return right;
60     }
61     return cross;
62 }

```

Esempio 17.14: esecuzione dell'implementazione in C++ di FindMaximumSubarray

Eseguiamo la prima chiamata FindMaximumSubarray(a , 0, 3) sull'array $a = [-2, 5, -1, 3]$:

$a[j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$
	-2	5	-1	3

- **Chiamata FindMaximumSubarray(a , 0, 3):** $l = 0 \neq r = 3$, $mid = (0 + 3)/2 = 1$. Chiamata sulla metà sinistra.
- **Chiamata FindMaximumSubarray(a , 0, 1):** $l = 0 \neq r = 1$, $mid = (0 + 1)/2 = 0$.
 - FindMaximumSubarray(a , 0, 0) restituisce $\{0, 0, -2\}$ (caso base)
 - FindMaximumSubarray(a , 1, 1) restituisce $\{1, 1, 5\}$ (caso base)
 - FindMaxCrossingSubarray(a , 0, 0, 1): ciclo sinistro $i = 0$: $sum = -2$, $leftSum = -2$, $maxLeft = 0$; ciclo destro $j = 1$: $sum = 5$, $rightSum = 5$, $maxRight = 1$. Restituisce $\{0, 1, 3\}$
 - Confronto: $left.sum = -2$, $right.sum = 5$, $cross.sum = 3 \Rightarrow$ **restituisce** $\{1, 1, 5\}$
- **Chiamata FindMaximumSubarray(a , 2, 3):** $l = 2 \neq r = 3$, $mid = (2 + 3)/2 = 2$.
 - FindMaximumSubarray(a , 2, 2) restituisce $\{2, 2, -1\}$ (caso base)
 - FindMaximumSubarray(a , 3, 3) restituisce $\{3, 3, 3\}$ (caso base)
 - FindMaxCrossingSubarray(a , 2, 2, 3): ciclo sinistro $i = 2$: $sum = -1$, $leftSum = -1$, $maxLeft = 2$; ciclo destro $j = 3$: $sum = 3$, $rightSum = 3$, $maxRight = 3$. Restituisce $\{2, 3, 2\}$
 - Confronto: $left.sum = -1$, $right.sum = 3$, $cross.sum = 2 \Rightarrow$ **restituisce** $\{3, 3, 3\}$
- **Ritorno alla chiamata FindMaximumSubarray(a , 0, 3):** $left = \{1, 1, 5\}$, $right = \{3, 3, 3\}$.
 - FindMaxCrossingSubarray(a , 0, 1, 3): ciclo sinistro $i = 1$: $sum = 5$, $leftSum = 5$, $maxLeft = 1$; $i = 0$: $sum = 3 \leq 5$, nessun aggiornamento; ciclo destro $j = 2$: $sum = -1$, $rightSum = -1$, $maxRight = 2$; $j = 3$: $sum = 2 > -1$, $rightSum = 2$, $maxRight = 3$. Restituisce $\{1, 3, 5 + 2\} = \{1, 3, 7\}$
 - Confronto: $left.sum = 5$, $right.sum = 3$, $cross.sum = 7 \Rightarrow$ **restituisce** $\{1, 3, 7\}$

L'algoritmo termina con risultato $\{1, 3, 7\}$: il sotto-array di somma massima è $a[1..3] = [5, -1, 3]$ (indici 2..4 nello pseudocodice, dove l'array è indicizzato da 1), con somma 7. In totale vengono eseguite $2n - 1 = 7$ chiamate di FindMaximumSubarray.

Punti chiave sul problema del sotto-array di somma massima.

- algoritmo di forza bruta: $\Theta(n^2)$;

- algoritmo divide et impera (ricorsivo): $\Theta(n \log n)$, con la stessa equazione di ricorrenza del Merge Sort;
- algoritmo di Kadane (iterativo): $\Theta(n)$.

Un algoritmo ricorsivo divide et impera non è necessariamente il migliore possibile: qui, un semplice algoritmo iterativo che sfrutta la struttura del problema ($M(i)$ dipende solo da $M(i-1)$) è asintoticamente più veloce. Torneremo su questa lezione nel capitolo «Algoritmi ricorsivi contro iterativi».

17.5 Riepilogo

I quattro problemi del capitolo coprono le tre forme tipiche di un algoritmo ricorsivo sugli array: due chiamate sulle metà con lavoro di combinazione costante (somma), due chiamate sulle metà con lavoro di combinazione lineare (Merge Sort, sotto-array di somma massima), una chiamata su una metà (ricerca binaria).

algoritmo	ricorrenza	numero di passi	memoria extra
RecursiveSum	$S(n) = 2S(n/2) + c$	$7n - 5 = \Theta(n)$	$\Theta(\log_2 n)$
RecursiveSumLinear	$S(n) = S(n-1) + 3$	$3n - 1 = \Theta(n)$	$\Theta(n)$
MergeSort	$S(n) = 2S(n/2) + \Theta(n)$	$\Theta(n \log_2 n)$	$\Theta(n)$
RecursiveBinarySearch	$S(n) = S(\lfloor n/2 \rfloor) + 5$	$5(\lfloor \log_2 n \rfloor + 1) + 2 = \Theta(\log_2 n)$	$\Theta(\log_2 n)$
FindMaximumSubarray	$S(n) = 2S(n/2) + \Theta(n)$	$\Theta(n \log_2 n)$	$\Theta(\log_2 n)$

- Correttezza: sempre per induzione forte sulla dimensione del sottoproblema; le parti non ricorsive (Merge, FindMaxCrossingSubarray, Kadane) con invarianti di ciclo.
- La ricorsione dà un guadagno asintotico solo quando il lavoro del passo di combinazione, sommato su un livello, è piccolo rispetto a quello che fa un algoritmo iterativo (il Merge Sort rispetto agli algoritmi di ordinamento $\Theta(n^2)$). Per la somma e per la ricerca non dà alcun guadagno in tempo e costa memoria; per il sotto-array di somma massima il miglior algoritmo è iterativo.

Algoritmi ricorsivi e iterativi a confronto

Introduzione

Nei capitoli precedenti abbiamo studiato diversi algoritmi ricorsivi (capitolo «Ricorsione»: fattoriale, numeri di Fibonacci, MCD ricorsivo, potenza veloce, Torre di Hanoi; capitolo «Algoritmi ricorsivi»: somma di un array, Merge Sort, ricerca binaria ricorsiva, sotto-array di somma massima) e li abbiamo analizzati con le equazioni di ricorrenza e con il Master theorem. In questo capitolo rispondiamo a una domanda molto pratica:

quando conviene progettare un algoritmo ricorsivo, e quando no?

Vedremo che la risposta non dipende dalla ricorsione in sé, ma da **come** il problema viene scomposto in sottoproblemi:

- se il problema viene suddiviso in pochi sottoproblemi **indipendenti** di dimensione n/b (divide et impera), la ricorsione può dare un grande guadagno asintotico;
- se il problema viene solo ridotto dalla dimensione n alla dimensione $n - 1$, la ricorsione non dà alcun guadagno in tempo e costa memoria;
- se i sottoproblemi si **sovrappongono**, un algoritmo ricorsivo ingenuo può essere esponenzialmente più lento di uno iterativo.

Il capitolo spiega innanzitutto i costi che ogni chiamata ricorsiva comporta, poi confronta algoritmi ricorsivi e iterativi per i problemi del corso, studia una a una le tre situazioni precedenti, e si chiude con i tempi di esecuzione misurati delle implementazioni in C++ e con alcune linee guida.

18.1 Ricorsione e iterazione

Ogni algoritmo ricorsivo può essere trasformato in uno iterativo (che usa una **pila esplicita** per memorizzare le chiamate in sospenso), e ogni algoritmo iterativo può essere scritto in modo ricorsivo (ogni ciclo può essere sostituito da una funzione che chiama se stessa).

Ricorsione e iterazione hanno la **stessa potenza espressiva**: ciò che cambia dall'una all'altra è il **numero di passi** e la **quantità di memoria** necessari per risolvere il problema.

18.1.1 I costi nascosti della ricorsione

- Ogni volta che una funzione viene chiamata, il calcolatore crea un **frame** (o *record di attivazione*) nella **pila delle chiamate** (*call stack*), un'area di memoria riservata al programma. Il frame contiene:
 - i parametri della chiamata;
 - le variabili locali della chiamata;
 - l'indirizzo di ritorno, cioè il punto della funzione chiamante da cui riprende l'esecuzione quando la chiamata termina.
- Il frame viene rimosso dalla pila solo quando la chiamata termina. Di conseguenza, durante l'esecuzione di un algoritmo ricorsivo, tutte le chiamate che sono iniziate e non ancora terminate mantengono il loro frame nella pila.
- Questo produce tre costi nascosti, che non compaiono in un ciclo:
 1. **Sovraccarico delle chiamate**: creare e rimuovere un frame, copiare i parametri e saltare alla funzione e tornare indietro richiedono un numero costante di passi aggiuntivi per ogni chiamata. Questo non cambia l'ordine di crescita del tempo di esecuzione, ma rende un algoritmo ricorsivo più lento di un ciclo equivalente di un fattore costante.
 2. **Memoria**: la memoria usata dalla pila delle chiamate è proporzionale al massimo numero di chiamate *contemporaneamente* attive, cioè alla **profondità** della ricorsione (l'altezza dell'albero di ricorsione più uno, studiata in dettaglio più avanti in questo capitolo).
 3. **Stack overflow**: la pila delle chiamate ha una dimensione fissa e piuttosto piccola. Su Linux la dimensione predefinita è tipicamente 8 MB (comando `ulimit -s` $\Rightarrow$ 8192 KB), su Windows è tipicamente 1 MB. Se la profondità della ricorsione è troppo grande, la pila si esaurisce e il programma va in **crash** (in C++ con un *segmentation fault*). In Python il limite predefinito è di sole 1000 chiamate annidate.

Esempio 18.1: una somma lineare ricorsiva che va in crash

- Consideriamo l'algoritmo ricorsivo per decremento di uno che calcola la somma di un array $a[1..n]$ come $\text{SumDecrease}(a, n) = \text{SumDecrease}(a, n - 1) + a[n]$, con caso base $\text{SumDecrease}(a, 0) = 0$ (la somma per decremento di uno del capitolo «Algoritmi ricorsivi», con caso base $n = 0$ invece di $n = 1$).

- Le chiamate $\text{SumDecrease}(a, n), \text{SumDecrease}(a, n - 1), \dots, \text{SumDecrease}(a, 0)$ sono tutte attive contemporaneamente: la profondità della ricorsione è $n + 1$.
- Con $n = 10^7$ e un frame di circa 32 byte, la pila dovrebbe contenere circa

$$10^7 \cdot 32 = 3.2 \cdot 10^8 \text{ byte} \approx 320 \text{ MB},$$

cioè 40 volte gli 8 MB predefiniti.

- Il programma va in crash, benché l'array stesso ($4 \cdot 10^7$ byte = 40 MB) stia comodamente in memoria e la somma iterativa calcoli il risultato in circa 3 millisecondi. Nell'esperimento alla fine del capitolo misuriamo che, con una pila di 8 MB, il crash avviene già per $n \approx 7 \cdot 10^5$.

18.1.2 Dalla ricorsione all'iterazione con una pila esplicita

- Un algoritmo ricorsivo può sempre essere trasformato in uno iterativo gestendo **esplicitamente** una pila che contiene i parametri delle chiamate ancora da eseguire.
- Come esempio, questa è una versione iterativa della somma ricorsiva per dimezzamento `RecursiveSum` del capitolo «Algoritmi ricorsivi», che calcola la somma di $a[l..r]$ come la somma di $a[l..mid]$ più la somma di $a[mid + 1..r]$:

Algorithm: SumExplicitStack

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

Output: La somma $a[1] + a[2] + \dots + a[n]$

```

1 function SumExplicitStack(a, n)
2   s := 0
3   P := pila vuota                                // sotto-array in sospenso
4   inserisci (1, n) in P
5   while P non è vuota do
6     (l, r) := estrai da P
7     if l = r then
8       s ← s + a[l]                                // caso base
9     else
10      mid := ⌊(l + r)/2⌋
11      inserisci (mid + 1, r) in P
12      inserisci (l, mid) in P
13    end
14  end
15  return s
16 end

```

- La pila P svolge esattamente il ruolo della pila delle chiamate: contiene i sotto-array ancora da sommare. La sua dimensione massima è dello stesso ordine della profondità della ricorsione, cioè $\Theta(\log_2 n)$.

Esempio 18.2: SumExplicitStack su quattro elementi

Eseguiamo `SumExplicitStack(a, 4)` con $a = [4, 7, 1, 9]$. Ogni elemento riporta la coppia estratta da P , ciò che viene fatto, e il contenuto di P dopo (la cima della pila è a destra):

- estrai (1, 4): $mid = 2$, inserisci (3, 4) e (1, 2) $\Rightarrow P = [(3, 4), (1, 2)]$, $s = 0$;
- estrai (1, 2): $mid = 1$, inserisci (2, 2) e (1, 1) $\Rightarrow P = [(3, 4), (2, 2), (1, 1)]$, $s = 0$;
- estrai (1, 1): caso base, $s = 4 \Rightarrow P = [(3, 4), (2, 2)]$;
- estrai (2, 2): caso base, $s = 11 \Rightarrow P = [(3, 4)]$;
- estrai (3, 4): $mid = 3$, inserisci (4, 4) e (3, 3) $\Rightarrow P = [(4, 4), (3, 3)]$, $s = 11$;
- estrai (3, 3): caso base, $s = 12 \Rightarrow P = [(4, 4)]$;
- estrai (4, 4): caso base, $s = 21 \Rightarrow P$ è vuota.

L'algoritmo restituisce $21 = 4 + 7 + 1 + 9$. I sotto-array vengono elaborati nello stesso ordine delle chiamate di `RecursiveSum(a, 1, 4)`, e P non contiene mai più di 3 coppie.

Simulare la ricorsione con una pila esplicita **non** cambia l'ordine di crescita del numero di passi e della memoria: elimina solo il sovraccarico delle chiamate e sposta la pila dalla (piccola) pila delle chiamate alla (grande) memoria principale.

Un vero guadagno in tempo o in memoria può venire solo da un **algoritmo diverso**, non dalla sostituzione meccanica della ricorsione con l'iterazione (o viceversa).

18.2 Confronto in sintesi

- La tabella seguente confronta, per i problemi studiati nel corso, il miglior algoritmo iterativo del corso con quello ricorsivo.
- Il **tempo** è il numero di passi $S(n)$ nel caso peggiore (le operazioni aritmetiche sui numeri contano come un passo, come nel capitolo «Tempo di esecuzione»); lo **spazio** è la memoria usata **in aggiunta all'input**, compresa la pila delle chiamate.
- L'esito si riferisce all'algoritmo ricorsivo: **GUADAGNO** se è asintoticamente più veloce, **UGUALE** se ha lo stesso tempo di esecuzione, **PERDITA** se è peggiore in tempo o in spazio.

Problema	Iterativo	tempo	spazio	Ricorsivo	tempo	spazio	Esito
Ordinamento	Insertion, Selection, Exchange Sort	$\Theta(n^2)$	$\Theta(1)$	Merge Sort	$\Theta(n \log_2 n)$	$\Theta(n)$	GUADAGNO in tempo (divide et impera), pagato con $\Theta(n)$ di memoria
Potenza x^n	n moltiplicazioni	$\Theta(n)$	$\Theta(1)$	Potenza veloce	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$	GUADAGNO , che deriva dal dimezzamento di n : anche l'algoritmo iterativo square-and-multiply è $\Theta(\log_2 n)$, con spazio $\Theta(1)$
Sotto-array di somma massima	Forza bruta	$\Theta(n^2)$	$\Theta(1)$	Divide et impera	$\Theta(n \log_2 n)$	$\Theta(\log_2 n)$	GUADAGNO rispetto alla forza bruta, ma l'algoritmo iterativo di Kadane, con tempo $\Theta(n)$ e spazio $\Theta(1)$, batte entrambi
Prodotto di interi di n cifre	Metodo scolastico	$\Theta(n^2)$	$\Theta(n)$	Karatsuba	$\Theta(n^{\log_2 3})$	$\Theta(n)$	GUADAGNO : $S(n) = 3S(n/2) + \Theta(n)$, $n^{\log_2 3} \approx n^{1.585}$
Somma di un array	Ciclo	$\Theta(n)$	$\Theta(1)$	Dimezzamento Decr. di uno	$\Theta(n)$ $\Theta(n)$	$\Theta(\log_2 n)$ $\Theta(n)$	UGUALE tempo, PERDITA in spazio
Ricerca binaria	Ciclo	$\Theta(\log_2 n)$	$\Theta(1)$	Ricerca binaria ricorsiva	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$	UGUALE tempo, PERDITA in spazio; ricorsione in coda, si può eliminare
Fattoriale, ricerca lineare	Ciclo	$\Theta(n)$	$\Theta(1)$	Decremento di uno	$\Theta(n)$	$\Theta(n)$	UGUALE tempo, PERDITA in spazio, rischio di stack overflow
MCD (Euclide)	Ciclo	$\Theta(\log m)$	$\Theta(1)$	MCD ricorsivo	$\Theta(\log m)$	$\Theta(\log m)$	UGUALE : ricorsione in coda, la profondità è solo logaritmica
Numeri di Fibonacci	Ciclo con due variabili	$\Theta(n)$	$\Theta(1)$	Ricorsione ingenua	$\Theta(\varphi^n)$	$\Theta(n)$	enorme PERDITA : sottoproblemi sovrapposti; la memoizzazione lo riporta a $\Theta(n)$
Torre di Hanoi	Esistono versioni iterative	$\Theta(2^n)$	$\Theta(n)$	Hanoi ricorsivo	$\Theta(2^n)$	$\Theta(n)$	UGUALE : servono comunque $2^n - 1$ mosse; la ricorsione è la descrizione naturale

- Nella tabella, $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$ è la sezione aurea e $m = \min(a, b)$ per l'MCD di a e b .
- Le quattro righe del primo blocco sono i casi in cui la ricorsione **conviene**: in tutti il problema viene suddiviso in sottoproblemi di dimensione $n/2$ (sezione successiva).
- Le righe del secondo blocco sono i casi in cui la ricorsione **sostituisce** soltanto un ciclo: nessun guadagno in tempo, una perdita in memoria (sezione «Spazio»).
- La riga di Fibonacci è il caso in cui la ricorsione ingenua è un **disastro**, perché gli stessi sottoproblemi vengono risolti più e più volte (sezione «Quando i sottoproblemi si sovrappongono»).

18.3 Il divide et impera dà il guadagno, non la ricorsione in sé

- Il numero di passi di un algoritmo ricorsivo è descritto da un'equazione di ricorrenza. Nel corso compaiono due famiglie di equazioni di ricorrenza:

- **decremento di uno**: il problema di dimensione n viene ridotto a **un** problema di dimensione $n - 1$,

$$S(n) = S(n - 1) + f(n);$$

- **divide et impera**: il problema di dimensione n viene suddiviso in a problemi di dimensione n/b ,

$$S(n) = a S(n/b) + f(n), \quad a \geq 1, b > 1.$$

18.3.1 Decremento di uno: la ricorsione è un ciclo sotto mentite spoglie

Osservazione 18.1: equazioni di ricorrenza per decremento di uno

Sia $S(n) = S(n - 1) + f(n)$ per $n \geq 1$, con $S(0) = d$. Allora

$$S(n) = d + \sum_{i=1}^n f(i).$$

In particolare, se $f(n) = c$ per una costante $c > 0$, allora $S(n) = cn + d = \Theta(n)$.

Dimostrazione. Per induzione su n . Per $n = 0$ la somma è vuota e $S(0) = d$. Se la formula vale per $n - 1$, allora

$$S(n) = S(n - 1) + f(n) = d + \sum_{i=1}^{n-1} f(i) + f(n) = d + \sum_{i=1}^n f(i).$$

Se $f(i) = c$ per ogni i , allora $\sum_{i=1}^n c = cn$ e $S(n) = cn + d$. Poiché $\lim_{n \rightarrow \infty} \frac{cn+d}{n} = c \in \mathbb{R}_{>0}$, si ha $S(n) = \Theta(n)$. $\square$

- La formula $S(n) = d + \sum_{i=1}^n f(i)$ è **esattamente** il numero di passi di un ciclo che esegue, per $i = 1, 2, \dots, n$, un corpo di costo $f(i)$. La ricorsione per decremento di uno svolge lo stesso lavoro di un ciclo:
 - fattoriale, ricerca lineare e somma di un array per decremento di uno: $S(n) = S(n - 1) + \Theta(1) = \Theta(n)$, come il ciclo corrispondente;
 - un insertion sort ricorsivo (ordinare $a[1..n - 1]$ ricorsivamente, poi inserire $a[n]$): $S(n) = S(n - 1) + \Theta(n) = \Theta(n^2)$, come l'**InsertionSort** iterativo.
- In tutti questi casi l'algoritmo ricorsivo ha lo stesso ordine di crescita del tempo di esecuzione di quello iterativo, più il sovraccarico delle chiamate e una profondità di ricorsione $\Theta(n)$.

18.3.2 Divide et impera: da dove viene il guadagno

- Il Merge Sort suddivide l'array in due metà, le ordina ricorsivamente e le fonde in $\Theta(n)$ passi:

$$S(n) = 2S(n/2) + \Theta(n) = \Theta(n \log_2 n)$$

per il caso 2 del Master theorem. Il guadagno rispetto a $\Theta(n^2)$ deriva dal fatto che le due metà sono **bilanciate** e **indipendenti**: l'albero di ricorsione ha solo $\Theta(\log_2 n)$ livelli, e ogni livello costa $\Theta(n)$.

- La stessa suddivisione si può organizzare **senza ricorsione**: si fondono prima coppie di sotto-array di lunghezza 1, poi coppie di sotto-array di lunghezza 2, poi di lunghezza 4, e così via. Questo è il Merge Sort **bottom-up**. Usa la procedura $\text{Merge}(a, l, m, r)$ del capitolo «Algoritmi ricorsivi», che fonde i sotto-array ordinati $a[l..m]$ e $a[m + 1..r]$ in $\Theta(r - l + 1)$ passi.

Algorithm: MergeSortBottomUp

Input : Un array $a[1..n]$ di n numeri naturali

Output: L'array a ordinato in ordine non decrescente

```

1 function MergeSortBottomUp(a, n)
2   width := 1
3   while width < n do
4     l := 1
5     while l + width ≤ n do
6       m := l + width - 1
7       r := min(l + 2 · width - 1, n)
8       Merge(a, l, m, r)
9       l ← l + 2 · width
10    end
11    width ← 2 · width
12  end
13 end
```

Esempio 18.3: Merge Sort bottom-up

Consideriamo $a = [5, 2, 4, 6, 1, 3]$ ($n = 6$).

- $width = 1$: si fondono $a[1..1]$ con $a[2..2]$, $a[3..3]$ con $a[4..4]$, $a[5..5]$ con $a[6..6] \Rightarrow a = [2, 5, 4, 6, 1, 3]$.
- $width = 2$: si fondono $a[1..2]$ con $a[3..4]$; per $l = 5$ si ha $l + width = 7 > 6$, quindi $a[5..6]$ resta com'è $\Rightarrow a = [2, 4, 5, 6, 1, 3]$.
- $width = 4$: si fondono $a[1..4]$ con $a[5..6] \Rightarrow a = [1, 2, 3, 4, 5, 6]$.
- $width = 8 \geq 6$: ci si ferma.

L'algoritmo esegue $3 = \lceil \log_2 6 \rceil$ passate.

Osservazione 18.2: Merge Sort bottom-up

Per $n \geq 2$, `MergeSortBottomUp` esegue $\lceil \log_2 n \rceil$ passate del ciclo esterno e il suo numero di passi nel caso peggiore è

$$S(n) = \Theta(n \log_2 n),$$

lo stesso del Merge Sort ricorsivo. Inoltre, non usa la pila delle chiamate.

Dimostrazione. Sia $n \geq 2$. Nel ciclo esterno $width$ assume i valori $1, 2, 4, \dots, 2^{p-1}$, dove p è il più piccolo intero con $2^p \geq n$, cioè $p = \lceil \log_2 n \rceil$.

Limite superiore. In una passata, le chiamate a `Merge` (riga 8) lavorano su sotto-array disgiunti, quindi la loro lunghezza totale è al più n e il loro costo totale è $O(n)$. Il ciclo interno viene eseguito al più $\frac{n}{2 \cdot width} + 1 \leq n$ volte, ogni volta con un numero costante di altri passi. Quindi ogni passata costa $O(n)$ e $S(n) = O(n \lceil \log_2 n \rceil) = O(n \log_2 n)$.

Limite inferiore. Il ciclo interno si ferma quando $l + width > n$, quindi gli elementi che non vengono fusi in una passata sono al più $width$. Nelle passate con $width \leq n/2$, vengono fusi almeno $n - width \geq n/2$ elementi, con $\Omega(n)$ passi. Queste passate sono quelle con $width = 2^t$, $t = 0, 1, \dots, \lfloor \log_2 n \rfloor - 1$, cioè $\lfloor \log_2 n \rfloor$ passate. Quindi $S(n) = \Omega(n \lfloor \log_2 n \rfloor) = \Omega(n \log_2 n)$.

Quindi $S(n) = \Theta(n \log_2 n)$. L'algoritmo non contiene chiamate ricorsive, quindi l'unica memoria aggiuntiva è quella usata da `Merge`, cioè $\Theta(n)$. $\square$

Il guadagno del Merge Sort rispetto agli algoritmi di ordinamento $\Theta(n^2)$ è una conseguenza della strategia **divide et impera** (metà bilanciate, fuse in tempo lineare), non della ricorsione: la versione iterativa bottom-up ha lo stesso tempo di esecuzione $\Theta(n \log_2 n)$.

18.3.3 Potenza veloce: il guadagno viene dal dimezzamento

- La potenza veloce ricorsiva del capitolo «Ricorsione» calcola $x^n = (x^{\lfloor n/2 \rfloor})^2$ se n è pari, e $x^n = (x^{\lfloor n/2 \rfloor})^2 \cdot x$ se n è dispari, con $S(n) = S(\lfloor n/2 \rfloor) + \Theta(1) = \Theta(\log_2 n)$.
- Il dimezzamento dell'esponente si può realizzare anche con un ciclo, leggendo i bit di n dal meno significativo (**square-and-multiply**):

Algorithm: PowerIt**Input** : Un numero x e un numero naturale n **Output**: La potenza x^n

```

1 function PowerIt( $x, n$ )
2    $p := 1$ 
3    $y := x$ 
4    $k := n$ 
5   while  $k > 0$  do
6     if  $k$  è dispari then
7        $p \leftarrow p \cdot y$ 
8     end
9      $y \leftarrow y \cdot y$ 
10     $k \leftarrow \lfloor k/2 \rfloor$ 
11  end
12  return  $p$ 
13 end

```

- All'inizio di ogni iterazione si ha $p \cdot y^k = x^n$ (vale all'inizio con $p = 1, y = x, k = n$, ed è preservata da ogni iterazione); quando $k = 0$ si ottiene $p = x^n$.

Esempio 18.4: square-and-multiply con $n = 13$

Eseguiamo **PowerIt**($x, 13$), dove $13 = (1101)_2$. Ogni elemento riporta i valori alla fine di un'iterazione:

- $k = 13$ è dispari: $p = x, y = x^2, k = 6$;
- $k = 6$ è pari: $p = x, y = x^4, k = 3$;
- $k = 3$ è dispari: $p = x \cdot x^4 = x^5, y = x^8, k = 1$;
- $k = 1$ è dispari: $p = x^5 \cdot x^8 = x^{13}, y = x^{16}, k = 0$.

Le iterazioni moltiplicano p per x, x^4 e x^8 (i bit di 13 uguali a 1), e $p = x^{1+4+8} = x^{13}$ dopo $4 = \lfloor \log_2 13 \rfloor + 1$ iterazioni.

Osservazione 18.3: iterazioni di PowerIt

Per $n \geq 1$, **PowerIt** esegue esattamente $\lfloor \log_2 n \rfloor + 1$ iterazioni, quindi $S(n) = \Theta(\log_2 n)$, e usa $\Theta(1)$ di memoria aggiuntiva.

Dimostrazione. Ogni iterazione sostituisce k con $\lfloor k/2 \rfloor$, cioè elimina il bit meno significativo della rappresentazione binaria di k . Quindi il numero di iterazioni è il numero di cifre binarie di n , che è $\lfloor \log_2 n \rfloor + 1$ (capitolo «Rappresentazione dei numeri»). Ogni iterazione esegue un numero costante di passi, quindi $S(n) = \Theta(\log_2 n)$. Vengono usate solo le tre variabili p, y, k . □

18.3.4 Sotto-array di somma massima: un algoritmo iterativo migliore

- Per il problema del sotto-array di somma massima (capitolo «Algoritmi ricorsivi») l'algoritmo divide et impera, con $S(n) = 2S(n/2) + \Theta(n) = \Theta(n \log_2 n)$, migliora l'algoritmo di forza bruta $\Theta(n^2)$.
- Tuttavia, un algoritmo **iterativo** più astuto, dovuto a Kadane, risolve il problema in $\Theta(n)$ passi. Per ogni posizione i mantiene la somma massima cur di un sotto-array che **termina** in posizione i : tale sotto-array o è il solo $a[i]$, oppure estende il miglior sotto-array che termina in $i - 1$. Questa è una versione compatta dell'algoritmo **Kadane** del capitolo «Algoritmi ricorsivi», che restituisce solo la somma massima:

Algorithm: Kadane

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

Output: La somma massima di un sotto-array non vuoto $a[l..r]$

```

1 function Kadane(a, n)
2   best := a[1]
3   cur := a[1]          // miglior somma di un sotto-array che termina in i
4   i := 2
5   while i ≤ n do
6     cur ← max(a[i], cur + a[i])
7     best ← max(best, cur)
8     i ← i + 1
9   end
10  return best
11 end
  
```

- Il ciclo viene eseguito $n - 1$ volte con un numero costante di passi: $S(n) = \Theta(n)$, con $\Theta(1)$ di memoria aggiuntiva.
- Il divide et impera è una potente **tecnica di progetto**, ma non è sempre la migliore: un guadagno rispetto alla forza bruta non significa che l'algoritmo ricorsivo sia ottimo.

18.3.5 Moltiplicazione di interi: Karatsuba

- Due interi di n cifre si moltiplicano con il metodo scolastico in $\Theta(n^2)$ passi sulle cifre.
- Suddividendo ciascun numero in due metà di $n/2$ cifre, il prodotto richiede quattro prodotti di numeri di $n/2$ cifre. Karatsuba ha osservato che bastano **tre** prodotti (più addizioni e scorrimenti, che costano $\Theta(n)$):

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

- Qui $a = 3$, $b = 2$, $f(n) = \Theta(n)$ e $n^{\log_2 3} \approx n^{1.585}$; poiché $f(n) = O(n^{\log_2 3 - \varepsilon})$ per $\varepsilon = 0.5$, il caso 1 del Master theorem dà $S(n) = \Theta(n^{\log_2 3})$, asintoticamente migliore di $\Theta(n^2)$.

Punti chiave. Una ricorsione per decremento di uno svolge lo stesso lavoro di un ciclo. Una ricorsione divide et impera può essere asintoticamente più veloce dei semplici algoritmi iterativi (Merge Sort, potenza veloce, Karatsuba), ma il guadagno deriva dalla scomposizione in sottoproblemi indipendenti e bilanciati, e la stessa scomposizione si può spesso organizzare con un ciclo (Merge Sort bottom-up, square-and-multiply). Il miglior algoritmo per un problema può addirittura essere iterativo (Kadane).

18.4 Quando i sottoproblemi si sovrappongono

- Nel divide et impera i sottoproblemi sono **disgiunti**: le due metà del Merge Sort non hanno alcun elemento in comune. Quando i sottoproblemi si **sovrappongono**, un algoritmo ricorsivo ingenuo risolve molte volte lo stesso sottoproblema.
- I numeri di Fibonacci sono definiti da $F(0) = 0$, $F(1) = 1$ e $F(n) = F(n-1) + F(n-2)$ per $n \geq 2$. La traduzione diretta della definizione è:

Algorithm: FibRec

Input : Un numero naturale n

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

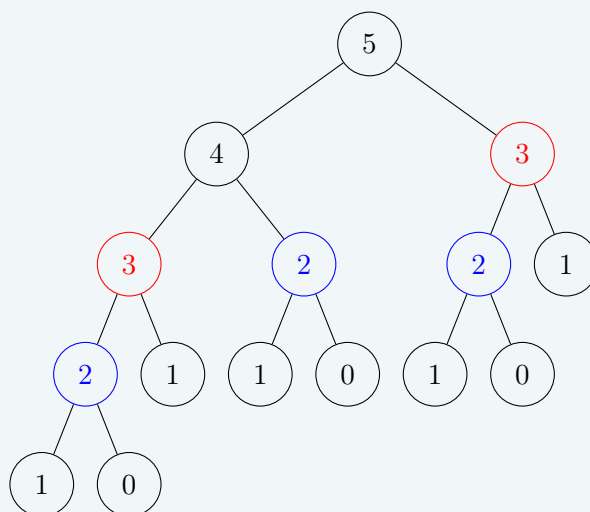
```

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

```

Esempio 18.5: l'albero di ricorsione di FibRec(5)

Questo è l'albero di ricorsione di FibRec(5) (ogni nodo è una chiamata, etichettata con il suo parametro). Il sotto-albero di FibRec(3) viene calcolato **due volte**, la chiamata FibRec(2) **tre volte**:



L'albero ha 15 nodi, mentre servono solo 6 valori diversi $F(0), F(1), \dots, F(5)$.

Osservazione 18.4: numero di chiamate di FibRec

Per ogni $n \geq 0$, il numero di chiamate eseguite da **FibRec**(n) è

$$C(n) = 2F(n+1) - 1.$$

Dimostrazione. Le chiamate soddisfano $C(0) = C(1) = 1$ e $C(n) = 1 + C(n-1) + C(n-2)$ per $n \geq 2$. Dimostriamo la formula per induzione forte. Casi base: $C(0) = 1 = 2F(1) - 1$ e $C(1) = 1 = 2F(2) - 1$. Se la formula vale per $n-1$ e $n-2$, allora

$$C(n) = 1 + (2F(n) - 1) + (2F(n-1) - 1) = 2(F(n) + F(n-1)) - 1 = 2F(n+1) - 1. \quad \square$$

Osservazione 18.5: crescita dei numeri di Fibonacci

Sia $\varphi = \frac{1+\sqrt{5}}{2}$. Per ogni $n \geq 1$,

$$\varphi^{n-2} \leq F(n) \leq \varphi^{n-1}.$$

Quindi $F(n) = \Theta(\varphi^n)$ e il numero di passi di **FibRec** è $S(n) = \Theta(C(n)) = \Theta(\varphi^n)$.

Dimostrazione. Il numero φ soddisfa $\varphi^2 = \varphi + 1$. Casi base: $\varphi^{-1} \approx 0.618 \leq F(1) = 1 \leq \varphi^0 = 1$ e $\varphi^0 = 1 \leq F(2) = 1 \leq \varphi \approx 1.618$. Per $n \geq 3$, per induzione forte,

$$F(n) = F(n-1) + F(n-2) \geq \varphi^{n-3} + \varphi^{n-4} = \varphi^{n-4}(\varphi + 1) = \varphi^{n-4}\varphi^2 = \varphi^{n-2},$$

e allo stesso modo $F(n) \leq \varphi^{n-2} + \varphi^{n-3} = \varphi^{n-3}(\varphi + 1) = \varphi^{n-1}$. Di conseguenza, $\varphi^{-2}\varphi^n \leq F(n) \leq \varphi^{-1}\varphi^n$, cioè $F(n) = \Theta(\varphi^n)$. Ogni chiamata esegue un numero costante di passi (oltre alle sue sotto-chiamate), quindi $S(n) = \Theta(C(n)) = \Theta(2F(n+1) - 1) = \Theta(\varphi^n)$. $\square$

- Per esempio, **FibRec**(40) esegue $2F(41) - 1 = 331\,160\,281$ chiamate per calcolare $F(40) = 102\,334\,155$, e **FibRec**(50) esegue $2F(51) - 1 = 40\,730\,022\,147$ chiamate.

18.4.1 Memoizzazione

- Lo spreco di `FibRec` si elimina memorizzando in una tabella `memo[0..n]` i valori già calcolati (**memoizzazione**): prima di calcolare $F(k)$ si verifica se è già nella tabella.

Algorithm: `FibMemo`

Input : Un numero naturale n e una tabella `memo[0..n]` le cui voci sono tutte *non definite*

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

```

1 function FibMemo( $n$ ,  $memo$ )
2   if  $n \leq 1$  then
3     return  $n$ 
4   end
5   if  $memo[n]$  è non definita then
6      $memo[n] \leftarrow \text{FibMemo}(n-1, memo) + \text{FibMemo}(n-2, memo)$ 
7   end
8   return  $memo[n]$ 
9 end

```

Osservazione 18.6: numero di chiamate di `FibMemo`

Per ogni $n \geq 2$, `FibMemo`($n, memo$) (con tutte le voci di `memo` non definite) esegue esattamente $2n - 1$ chiamate. Quindi $S(n) = \Theta(n)$; la memoria usata è $\Theta(n)$ (la tabella e la pila delle chiamate).

Dimostrazione. Sia $M(k)$ il numero di chiamate eseguite da `FibMemo`($k, memo$) quando le voci `memo[0..k]` non sono definite. Per $k = 2$ la chiamata ne effettua due, entrambe casi base: $M(2) = 3$. Per $k \geq 3$ la chiamata effettua la chiamata `FibMemo`($k - 1, memo$), che esegue $M(k - 1)$ chiamate e riempie `memo[2..k - 1]`; poi effettua la chiamata `FibMemo`($k - 2, memo$), che trova `memo[k - 2]` già definita (oppure è un caso base se $k - 2 \leq 1$) e termina subito. Quindi $M(k) = 1 + M(k - 1) + 1 = M(k - 1) + 2$, e per l'Osservazione 18.1 $M(n) = M(2) + 2(n - 2) = 2n - 1$. Ogni chiamata esegue un numero costante di passi, quindi $S(n) = \Theta(n)$. La tabella ha $n + 1$ voci e la profondità della ricorsione è n . $\square$

18.4.2 L'algoritmo iterativo

- Calcolando i valori $F(2), F(3), \dots, F(n)$ in ordine crescente, ogni valore richiede solo i due precedenti. Non serve nemmeno la tabella: bastano due variabili.

Algorithm: FibIt

Input : Un numero naturale n

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

```

1 function FibIt( $n$ )
2   if  $n \leq 1$  then
3     return  $n$ 
4   end
5   prev := 0                                //  $F(i - 2)$ 
6   curr := 1                                //  $F(i - 1)$ 
7    $i := 2$ 
8   while  $i \leq n$  do
9     next := prev + curr
10    prev ← curr
11    curr ← next
12     $i \leftarrow i + 1$ 
13  end
14  return curr
15 end
  
```

- Il ciclo viene eseguito $n - 1$ volte: $S(n) = \Theta(n)$, con $\Theta(1)$ di memoria aggiuntiva.

Algoritmo	tempo	spazio
FibRec (ricorsione ingenua)	$\Theta(\varphi^n)$	$\Theta(n)$
FibMemo (ricorsione + memoizzazione)	$\Theta(n)$	$\Theta(n)$
FibIt (iterazione, bottom-up)	$\Theta(n)$	$\Theta(1)$

Quando i sottoproblemi si sovrappongono, la tecnica giusta è risolvere ogni sottoproblema **una sola volta** e memorizzarne il valore: top-down con la memoizzazione, oppure bottom-up con un ciclo. Questa è l'idea della **programmazione dinamica**, che verrà studiata più avanti nel corso.

18.5 Spazio

Le sezioni precedenti hanno confrontato i numeri di passi. Passiamo ora alla seconda risorsa, la memoria, che per un algoritmo ricorsivo è governata dalla profondità della ricorsione.

18.5.1 Profondità della ricorsione

Definizione 18.7: profondità della ricorsione

La **profondità della ricorsione** $D(n)$ di un algoritmo ricorsivo su un input con parametro di dimensione n è il massimo numero di chiamate attive contemporaneamente durante l'esecuzione. È il numero di nodi del cammino più lungo dalla radice a una foglia dell'albero di ricorsione, cioè l'altezza dell'albero più uno.

- Quando le chiamate vengono eseguite una dopo l'altra, sono attive solo le chiamate sul cammino dalla radice alla chiamata corrente. Quindi, se ogni frame ha dimensione costante, la memoria usata dalla pila delle chiamate è $\Theta(D(n))$.
- La profondità **non** è il numero di chiamate: **FibRec**(n) esegue $\Theta(\varphi^n)$ chiamate, ma al più n di esse sono attive contemporaneamente.

Algoritmo	caso base	profondità $D(n)$	$D(10^6)$
Fattoriale, somma lineare (decr. di uno)	$n = 0$	$n + 1$	1 000 001
Torre di Hanoi con n dischi	$n = 0$	$n + 1$	1 000 001
FibRec (n), $n \geq 1$	$n \leq 1$	n	1 000 000
Somma, Merge Sort, sotto-array (dimezzam.)	$l = r$	$\lceil \log_2 n \rceil + 1$	21
Ricerca binaria ricorsiva (caso peggiore)	$l > r$	$\lfloor \log_2 n \rfloor + 2$	21
Potenza veloce x^n , $n \geq 1$	$n = 0$	$\lfloor \log_2 n \rfloor + 2$	21
MCD ricorsivo(a, b)	$b = 0$	$\Theta(\log \min(a, b))$	–

- Per gli algoritmi per dimezzamento, il sotto-array più grande al livello t dell'albero di ricorsione ha $\lceil n/2^t \rceil$ elementi, e raggiunge la dimensione 1 per $t = \lceil \log_2 n \rceil$; per la potenza veloce, l'esponente assume i valori $n, \lfloor n/2 \rfloor, \dots, 1, 0$, che sono $\lfloor \log_2 n \rfloor + 2$.
- Una profondità $\Theta(\log_2 n)$ non è mai un problema in pratica (21 frame per $n = 10^6$, 31 per $n = 10^9$). Una profondità $\Theta(n)$ è un vero rischio di stack overflow per n grande.

18.5.2 Ricorsione in coda e sua eliminazione

Definizione 18.8: ricorsione in coda

Una chiamata ricorsiva è una **chiamata in coda** (*tail call*) se è l'ultima operazione eseguita dalla funzione, e il suo risultato viene restituito così com'è, senza ulteriori calcoli. Un algoritmo ricorsivo è **ricorsivo in coda** (*tail recursive*) se tutte le sue chiamate ricorsive sono chiamate in coda.

- L'MCD ricorsivo (restituisce $\text{GCD}(b, a \bmod b)$) e la ricerca binaria ricorsiva (restituisce il risultato della ricerca in una metà) sono ricorsivi in coda.
- Il fattoriale ($n \cdot \text{Fact}(n-1)$), la somma per decremento di uno ($\text{SumDecrease}(a, n-1) + a[n]$) e **FibRec non** sono ricorsivi in coda: dopo che la chiamata ricorsiva termina resta ancora una moltiplicazione o un'addizione da eseguire, quindi il frame deve essere mantenuto.
- In una chiamata in coda, il frame della funzione chiamante non serve più: dopo la chiamata non resta nulla da fare. Quindi la chiamata può essere sostituita da un **aggiornamento dei parametri** seguito da un salto all'inizio della funzione, cioè da un ciclo.

Eliminazione meccanica della ricorsione in coda

1. Racchiudere il corpo della funzione in un ciclo **while** la cui condizione è la negazione della condizione del caso base.
2. Sostituire ogni chiamata in coda $f(e_1, \dots, e_k)$ con l'assegnazione **simultanea** dei nuovi valori $e_1, \dots, e_k$ ai parametri (usare variabili temporanee quando un nuovo valore dipende da un parametro che viene modificato).
3. Spostare il return del caso base dopo il ciclo.

Esempio 18.6: MCD ricorsivo $\Rightarrow$ MCD iterativo**Algorithm: GCDRec**

```

1 function GCDRec(a, b)
2   if b = 0 then
3     return a
4   end
5   return GCDRec(b, a mod b)
6 end

```

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

```

- Condizione di base $b = 0 \Rightarrow$ condizione del ciclo $b \neq 0$.
- Chiamata in coda $\text{GCDRec}(b, a \bmod b) \Rightarrow (a, b) \leftarrow (b, a \bmod b)$; la variabile temporanea r è necessaria perché $a \bmod b$ deve essere calcolato con il *vecchio* valore di a .
- Con $(a, b) = (252, 105)$ entrambi gli algoritmi passano per le coppie $(252, 105)$, $(105, 42)$, $(42, 21)$, $(21, 0)$ e restituiscono 21: quattro chiamate contro tre iterazioni.

Esempio 18.7: ricerca binaria ricorsiva $\Rightarrow$ ricerca binaria iterativa**Algorithm: RecBinarySearch**

```

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

```

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

```

- Condizione di base $l > r \Rightarrow$ condizione del ciclo $l \leq r$.
- Il secondo caso base ($a[mid] = m$) resta dentro il ciclo come **return**.
- Chiamata in coda $\text{RecBS}(a, mid + 1, r, m) \Rightarrow l \leftarrow mid + 1$; chiamata in coda $\text{RecBS}(a, l, mid - 1, m) \Rightarrow r \leftarrow mid - 1$ (i parametri che non cambiano non vengono assegnati).
- Chiamando $\text{ItBS}(a, 1, n, m)$ otteniamo esattamente la **BinarySearch** iterativa del capitolo «Ricerca lineare e binaria».

Osservazione 18.9: eliminazione della ricorsione in coda

Sia f un algoritmo ricorsivo in coda le cui chiamate eseguono un numero costante di passi (oltre alla chiamata ricorsiva) e usano frame di dimensione costante, e sia g l'algoritmo iterativo ottenuto con l'eliminazione meccanica della ricorsione in coda. Allora g restituisce lo stesso risultato di f , il suo numero di passi è dello stesso ordine di quello di f , e usa $\Theta(1)$ di memoria aggiuntiva invece di $\Theta(D(n))$.

Dimostrazione. Siano $p_1, p_2, \dots, p_k$ i valori dei parametri nelle successive chiamate di f , dove p_k soddisfa una condizione di base. La j -esima iterazione del ciclo di g inizia con i parametri uguali a p_j ed esegue le stesse operazioni della chiamata di f con parametri p_j , tranne che la chiamata in coda è sostituita dall'assegnazione di p_{j+1} . Quindi g passa per gli stessi valori $p_1, \dots, p_k$ e restituisce, alla fine, ciò che restituisce la chiamata con parametri p_k ; poiché ogni chiamata di f restituisce così com'è il valore della successiva, questo è anche il risultato di f . Le k chiamate di f e le $k - 1$ iterazioni di g (più il test finale e il return) eseguono gli stessi passi a meno di una costante per chiamata, quindi i due numeri di passi sono dello stesso ordine. Infine, g usa solo i parametri e un numero costante di variabili temporanee. $\square$

- Una funzione che non è ricorsiva in coda può talvolta essere resa ricorsiva in coda con

un parametro aggiuntivo (un **accumulatore**) che porta il risultato parziale: per esempio $\text{FactAcc}(n, acc)$ restituisce acc se $n = 0$ e $\text{FactAcc}(n - 1, n \cdot acc)$ altrimenti, e $n! = \text{FactAcc}(n, 1)$. La sua eliminazione dà il solito ciclo $acc \leftarrow acc \cdot n, n \leftarrow n - 1$.

- I compilatori ottimizzanti spesso eliminano automaticamente le chiamate in coda: per esempio, g++ 15.2 con l'opzione `-O2` elimina le chiamate ricorsive delle funzioni C++ `gcdRec` e `binarySearchRec` del codice del corso, mentre con `-O0` le mantiene. Tuttavia, lo standard C++ non lo garantisce, e Python e Java non lo fanno mai. Per essere sicuri che la memoria sia $\Theta(1)$, si scriva il ciclo.

Punti chiave sullo spazio. La pila delle chiamate usa una memoria proporzionale alla profondità della ricorsione, non al numero di chiamate. Una profondità logaritmica è innocua; una profondità lineare (decremento di uno) rischia lo stack overflow. La ricorsione in coda può essere eliminata meccanicamente, ottenendo un ciclo con $\Theta(1)$ di memoria aggiuntiva.

18.6 Implementazione in C++

- Tutti gli algoritmi di questo capitolo sono implementati nel codice del corso, nel file `Recursion.cpp` (con `Recursion.h`); gli esperimenti della sezione successiva sono nel file `Experiments.cpp`, e gli algoritmi di ordinamento `insertionSort` e `mergeSort` sono nel file `Sortings.cpp`. Gli array sono indicizzati da 0.
- Queste sono le tre implementazioni della somma di un array:

```

1 long long sumIt(int *a, int n)
2 {
3     long long s = 0;
4     for (int i = 0; i < n; i++)
5     {
6         s = s + a[i];
7     }
8     return s;
9 }
10
11 // somma di a[l..r] per dimezzamento: profondità Theta(log n)
12 long long sumRecHalving(int *a, int l, int r)
13 {
14     if (l == r)
15     {
16         return a[l];
17     }
18     int mid = l + (r - l) / 2;
19     return sumRecHalving(a, l, mid) + sumRecHalving(a, mid + 1, r);
20 }
21
22 // somma di a[0..n-1] per decremento di uno: profondità Theta(n)
23 long long sumRecDecrease(int *a, int n)
24 {
25     if (n == 0)
26     {
27         return 0;

```

```

28     }
29     return sumRecDecrease(a, n - 1) + a[n - 1];
30 }

```

Esempio 18.8: la pila delle chiamate di sumRecDecrease

Questa è l'esecuzione passo per passo di `sumRecDecrease(a, 3)` sull'array $a = [4, 7, 1]$, che mostra il contenuto della pila delle chiamate (la cima della pila è a destra; `s` sta per `sumRecDecrease` con l'array `a` omissa).

$a[j]$	$j = 0$	$j = 1$	$j = 2$
	4	7	1

passo	pila delle chiamate (chiamate attive)	azione
1	s(3)	$n = 3 \neq 0$: chiama s(2), poi somma $a[2]$
2	s(3) s(2)	$n = 2 \neq 0$: chiama s(1), poi somma $a[1]$
3	s(3) s(2) s(1)	$n = 1 \neq 0$: chiama s(0), poi somma $a[0]$
4	s(3) s(2) s(1) s(0)	$n = 0$: restituisce 0
5	s(3) s(2) s(1)	restituisce $0 + a[0] = 4$
6	s(3) s(2)	restituisce $4 + a[1] = 11$
7	s(3)	restituisce $11 + a[2] = 12$

Al passo 4 ci sono $4 = n + 1$ chiamate attive: la profondità della ricorsione è $n + 1$. Le addizioni vengono eseguite solo **dopo** che si è raggiunto il caso base, mentre la pila si svuota. Per confronto, `sumIt(a, 3)` usa solo le due variabili `s` e `i`: $s = 0 \rightarrow 4 \rightarrow 11 \rightarrow 12$.

- Queste sono le tre implementazioni dei numeri di Fibonacci:

```

1 long long fibonacciRec(int n)
2 {
3     if (n <= 1)
4     {
5         return n;
6     }
7     return fibonacciRec(n - 1) + fibonacciRec(n - 2);
8 }
9
10 // memo deve avere n+1 posizioni, tutte inizializzate a -1
11 long long fibonacciMemo(int n, long long *memo)
12 {
13     if (n <= 1)
14     {
15         return n;
16     }
17     if (memo[n] == -1)
18     {
19         memo[n] = fibonacciMemo(n - 1, memo) + fibonacciMemo(n - 2, memo);
20     }
21     return memo[n];

```

```

22 }
23
24 long long fibonaccilt(int n)
25 {
26     if (n <= 1)
27     {
28         return n;
29     }
30     long long prev = 0; // F(i-2)
31     long long curr = 1; // F(i-1)
32     for (int i = 2; i <= n; i++)
33     {
34         long long next = prev + curr;
35         prev = curr;
36         curr = next;
37     }
38     return curr;
39 }

```

18.7 Esperimento

L'analisi asintotica prevede come cresce il numero di passi con n . Verifichiamo ora queste previsioni su un calcolatore reale.

18.7.1 Impostazione sperimentale

- I tempi di esecuzione sono stati misurati l'11 ottobre 2026 su un calcolatore con processore Intel Core i5-14500T (20 CPU logiche, frequenza massima 4.8 GHz), con sistema operativo Ubuntu 26.04 LTS (kernel Linux 7.0), usando il codice C++ del corso compilato con g++ 15.2.0 e l'opzione -O2. La dimensione predefinita della pila è 8 MB (`ulimit -s ⇒ 8192 KB`).
- La funzione `runRecursionExperiments` del file `Experiments.cpp` misura il tempo reale di ogni calcolo e lo ripete finché non sono trascorsi almeno 0.2 secondi (e almeno 3 ripetizioni); ogni tempo nelle tabelle è la media sulle ripetizioni.
 - **Ordinamento:** array di $n = 10^3$, 10^4 e 10^5 interi casuali tra 1 e 10^6 (seme 12345), ordinati con `insertionSort`, con il `mergeSort` ricorsivo e con `mergeSortBottomUp`.
 - **Fibonacci:** `fibonacciRec(n)` e `fibonaccilt(n)` per $n = 20, 25, 30, 35, 40$; le chiamate di `fibonacciRec` sono contate da un contatore nel codice; `fibonaccilt` viene ripetuta a blocchi di 10^5 chiamate.
 - **Somma di un array:** `sumIt`, `sumRecHalving` e `sumRecDecrease` su array di $n = 10^5$, 10^6 e 10^7 interi casuali tra 1 e 100, con il limite della pila rimosso (`ulimit -s unlimited`); un'esecuzione separata verifica se `sumRecDecrease` termina con la pila predefinita di 8 MB.

I tempi misurati dipendono dalla macchina, dal compilatore e dalle sue opzioni, dagli altri programmi in esecuzione e dalla memoria cache. Non vanno confrontati in termini assoluti: ciò che conta è **come crescono** con n , che è previsto dall'analisi asintotica ed è lo stesso su ogni macchina.

18.7.2 Ordinamento: Insertion Sort contro Merge Sort

n	insertionSort	mergeSort (ricorsivo)	mergeSortBottomUp
10^3	0.0695 ms	0.0361 ms	0.0332 ms
10^4	5.38 ms	0.654 ms	0.629 ms
10^5	525 ms	7.76 ms	7.43 ms

- Da $n = 10^4$ a $n = 10^5$ il tempo dell'Insertion Sort viene moltiplicato per circa 98, vicino al fattore 100 previsto da $\Theta(n^2)$, mentre il tempo del Merge Sort viene moltiplicato per circa 12, vicino al fattore previsto da $\Theta(n \log_2 n)$:

$$10 \cdot \frac{\log_2 10^5}{\log_2 10^4} = 12.5.$$

(Da $n = 10^3$ a $n = 10^4$ i fattori sono più piccoli, circa 77 e 18: per array piccoli i costi costanti e la memoria cache pesano ancora sulla misura.)

- Per $n = 10^5$ il Merge Sort è circa 70 volte più veloce. La versione ricorsiva e quella bottom-up hanno praticamente lo stesso tempo: il guadagno è dovuto al divide et impera, non alla ricorsione.

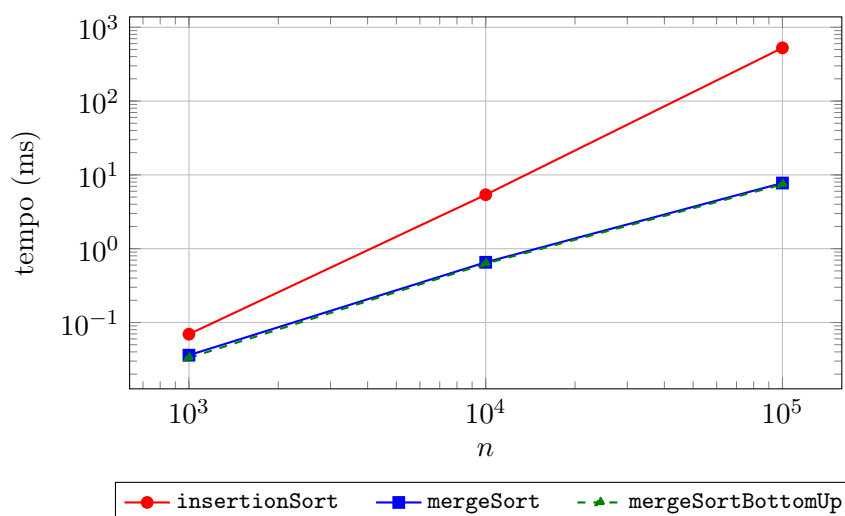


Figura 18.1: Tempi di esecuzione misurati dei tre algoritmi di ordinamento (scale logaritmiche su entrambi gli assi).

- In un grafico con scale logaritmiche su entrambi gli assi una potenza n^k è una retta di pendenza k : la retta dell'Insertion Sort ha pendenza vicina a 2, le rette del Merge Sort una pendenza leggermente maggiore di 1.

18.7.3 Fibonacci: ricorsione ingenua contro iterazione

n	$F(n)$	chiamate di <code>fibonacciRec</code>	<code>fibonacciRec</code>	<code>fibonacciIt</code>
20	6 765	21 891	0.00808 ms	4.0 ns
25	75 025	242 785	0.0699 ms	4.8 ns
30	832 040	2 692 537	0.764 ms	7.4 ns
35	9 227 465	29 860 703	9.15 ms	8.4 ns
40	102 334 155	331 160 281	93.8 ms	9.5 ns

- Il numero di chiamate è stato contato con un contatore nel codice e coincide con $2F(n+1) - 1$ (Osservazione 18.4).
- Quando n aumenta di 5, il tempo di `fibonacciRec` viene moltiplicato per un fattore compreso tra circa 9 e 12, in accordo con $\varphi^5 \approx 11.09$. Il tempo di `fibonacciIt` resta di pochi nanosecondi.
- Estrapolando con il fattore φ per ogni unità di n dal tempo misurato per $n = 40$:
 - `fibonacciRec`(50) richiede circa $93.8 \cdot \varphi^{10} \approx 11\,500$ ms, cioè circa 12 secondi;
 - `fibonacciRec`(60) richiede circa $93.8 \cdot \varphi^{20}$ ms, cioè circa 24 minuti;
 - `fibonacciRec`(100) richiede circa $93.8 \cdot \varphi^{60}$ ms, cioè circa 10 000 anni.
- Il compilatore ottimizza il codice (con `-O2` fonde più livelli della ricorsione in una singola chiamata di funzione), quindi il tempo per chiamata, circa $93.8 \text{ ms} / 331\,160\,281 \approx 0.28$ ns, è minore di quanto ci si aspetterebbe: questo cambia la costante, non la crescita esponenziale.

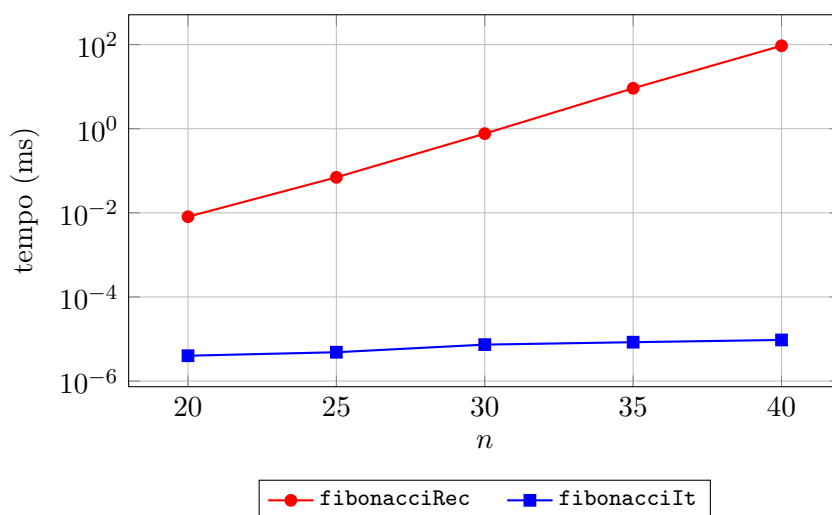


Figura 18.2: Tempi di esecuzione misurati degli algoritmi di Fibonacci ricorsivo e iterativo (scala logaritmica sull'asse verticale).

- In un grafico semi-logaritmico una funzione esponenziale è una retta: la pendenza della retta rossa è circa $\log_{10} \varphi \approx 0.21$ per ogni unità di n .

18.7.4 Somma di un array: tempo e pila

n	sumIt	sumRecHalving	sumRecDecrease	sumRecDecrease, pila 8 MB
10^5	0.0215 ms	0.0960 ms	0.207 ms	OK
10^6	0.210 ms	0.910 ms	2.24 ms	crash
10^7	2.69 ms	9.01 ms	34.9 ms	crash

- I tre algoritmi hanno lo stesso ordine di crescita $\Theta(n)$: quando n viene moltiplicato per 10, tutti i tempi vengono moltiplicati per circa 10 (tra 9.5 e 16; da $n = 10^6$ a $n = 10^7$ l'array non sta più nella memoria cache e i fattori aumentano).
- Le versioni ricorsive sono più lente di un fattore costante (circa 3–4.5 per la versione per dimezzamento, circa 10–13 per la versione per decremento di uno): questo è il sovraccarico delle chiamate.
- Con la pila predefinita di 8 MB, **sumRecDecrease** va in crash (segmentation fault) per $n = 10^6$ e $n = 10^7$; i tempi della tabella sono stati misurati rimuovendo il limite con il comando `ulimit -s unlimited`.
- Per bisezione su n (con una precisione di 10^3) abbiamo trovato che, con 8 MB, il massimo n che non va in crash è circa $6.97 \cdot 10^5$ con `-O2`, e circa $2.62 \cdot 10^5$ senza ottimizzazione (`-O0`). Senza ottimizzazione ogni chiamata usa un frame di 32 byte:

$$\frac{8 \cdot 2^{20}}{32} = 262\,144.$$

Con `-O2` il compilatore fonde quattro livelli della ricorsione in una singola chiamata con un frame di 48 byte, quindi

$$\frac{8 \cdot 2^{20}}{48} \cdot 4 \approx 6.99 \cdot 10^5 :$$

l'ottimizzazione cambia la costante, non la profondità $\Theta(n)$.

- **sumRecHalving** usa solo $\lceil \log_2 n \rceil + 1 \leq 25$ frame per $n \leq 10^7$ e non va mai in crash; **sumIt** non usa alcun frame.

18.8 Linee guida

Usare la ricorsione quando:

- il problema si suddivide in pochi sottoproblemi **indipendenti** di dimensione n/b (divide et impera), in modo che il Master theorem dia un tempo di esecuzione migliore di quello degli algoritmi iterativi (Merge Sort, potenza veloce, Karatsuba);
- la descrizione ricorsiva è quella naturale e comunque non è possibile alcun guadagno asintotico (Torre di Hanoi, e più avanti le visite di alberi e grafi);
- la profondità della ricorsione è piccola, tipicamente $\Theta(\log_2 n)$.

Evitare la ricorsione quando:

- il problema diminuisce solo da n a $n - 1$ con lavoro aggiuntivo costante: la ricorsione è un ciclo sotto mentite spoglie, con lo stesso tempo di esecuzione, un sovraccarico delle chiamate, $\Theta(n)$ di memoria di pila e un rischio di stack overflow (fattoriale, somma lineare, ricerca lineare);
- i sottoproblemi si **sovrappongono**: la ricorsione ingenua ripete gli stessi calcoli e può essere esponenziale (Fibonacci); si usi la memoizzazione o, meglio, un ciclo bottom-up (programmazione dinamica).

Considerare sempre lo spazio: la pila delle chiamate usa una memoria proporzionale alla profondità della ricorsione. La ricorsione in coda può essere eliminata meccanicamente, ottenendo un ciclo con $\Theta(1)$ di memoria aggiuntiva.

Considerare sempre il miglior algoritmo iterativo: un algoritmo ricorsivo può migliorare la forza bruta ed essere comunque battuto da un ciclo più astuto (l'algoritmo di Kadane per il sotto-array di somma massima).