Algoritmi ricorsivi su alberi binari

Gli alberi binari, definiti ricorsivamente, si prestano naturalmente alla progettazione di algoritmi attraverso la metodologia divide et impera.

Dimensione di un albero $\triangleq$ numero di nodi in esso contenuti

Un albero con n nodi ha $n-1$ archi.

Dimensione di un albero binario:

- un albero vuoto ha dimensione 0
- non vuoto, somma delle dimensioni dei due sottoalberi, incrementata di uno.

Algoritmo divide et impera per il calcolo della dimensione.

Decomposizione: se l'albero è vuoto, restituisci zero, altrimenti suddividilo nei due sottoalberi radicati nei figli

Ricorsione: calcola ricorsivamente le dimensioni dei sottoalberi

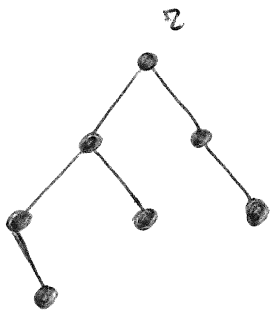
Ricombinazione: restituisci la somma delle dimensioni dei sottoalberi $+1$.

```
Dimensione (u)                          → radice dell'albero

IF (u == NULL) {
   RETURN 0;
}
   ELSE {
      dimensioneSX = Dimensione (u.sx);
      dimensioneDX = Dimensione (u.dx);
      RETURN dimensioneSX + dimensioneDX + 1;
   }
```

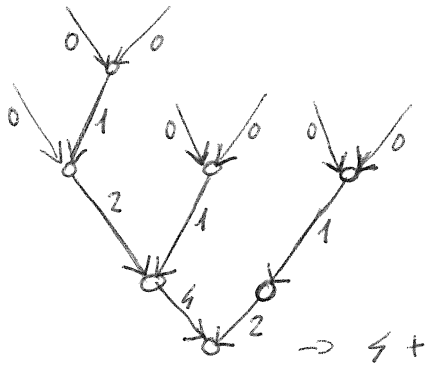
Proviamo a visualizzare l'esecuzione dell'algoritmo

T



Albero in Input

R



→ 4 + 2 + 1 = 7

Albero delle invocazioni ricorsive

La dimensione di T si legge nell'etichetta della radice di R

I nodi di T e di R sono in corrispondenza 1 - a - 1

L'etichetta numerica sull'arco ← indica il valore restituito dalla procedura chiamata a quella chiamante.

In R c'è un arco diretto → tra il nodo u e il nodo v se Dimensione(v) ha invocato Dimensione(u)



Teorema 3.6 La dimensione n di un albero binario può essere calcolata in tempo $O(n)$.

Osservazione preliminare: il problema della dimensione rientra nella famiglia di problemi su alberi risolvibili per "decomposizione", detti appunto "problemi decomponibili".

```
Decomponibile (u):
IF (u == NULL) {
    RETURN Decomponibile (NULL);
}
ELSE {
    risultatoSX = Decomponibile (u.sx)
    risultatoDX = Decomponibile (u.dx)
    RETURN Ricombina (risultatoSX, risultatoDX);
}
```

La complessità delle soluzioni che ricadono in questo schema divide et impera può essere espressa immediatamente attraverso una relazione di ricorrenza.

Ipotizziamo che il costo di divisione e ricombinazione sia costante.

Possiamo allora scrivere

costo della base (foglia o NULL)

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \\ T(z-1) + T(n-z) + c & \end{cases} \qquad (1)$$

$T(n)$: # di nodi del sottoalbero radicato in n (nodo n compreso)

$T(z-1)$: # nodi che discendono dal figlio sinistro del nodo n

$T(n-z)$: # nodi che discendono dal figlio destro del nodo n

c: costo decomposizione e ricombinazione

Teorema (generale). La relazione di ricorrenza (1) ha soluzione $O(n)$.

Dim. Il Master Theorem non è applicabile.

Utilizziamo il metodo della "sostituzione" (con prova per induzione). Scegliendo c' tale che $T(0) \leq c_0 \leq c'$, mostriamo per induzione che $T(n) \leq 3c' \cdot n \quad \forall n \geq 1$, dove $c' = \max\{c_0, c\}$.

Base dell'induzione. Sia $n = 1$. Allora $z = 1$ e, quindi,

$$T(1) \leq 2 \cdot T(0) + c \leq 2c_0 + c \leq 3c' \cdot 1$$

Ipotesi induttiva. Sia $T(n') \leq 3c'n'$, $1 \leq n' < n$.

Passo induttivo. Distinguiamo tre casi: $r=1$, $1<r<n$, $r=n$.

Nel caso in cui $1<r<n$, si ha:

$$T(n) \leq T(r-1) + T(n-r) + c$$

che, per l'ipotesi induttiva, risulta:

$$\begin{aligned} T(n) &\leq 3c'(r-1) + 3c'(n-r) + c \\ &\leq 3c'r + \underbrace{3c'}_{>3c} + 3c'n - 3c'r + c \\ &\leq 3c'n - 2c < 3c'n \end{aligned}$$

Se, invece, fosse $r=1$, si avrebbe

$$\begin{aligned} T(n) &\leq T(\phi) + T(n-1) + c \\ &\leq c_0 + 3c'(n-1) + c \\ &\leq c' + 3c'n + 3c' + c \\ &\leq 3c'n - \underbrace{2c'}_{>2c} + c \leq 3c'n - c \\ &< 3c'n \end{aligned}$$

Analogamente si prova il risultato per $r=n$.

Pertanto, $T(n) = O(n)$.

Esercizio svolto 3.4.

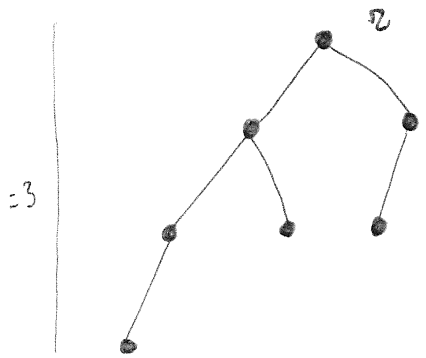
Altezza di un albero = massima distanza di una foglia dalla radice dell'albero.

Progettare un algoritmo ricorsivo che calcoli l'altezza di un albero binario in tempo $O(n)$, dove n denota la dimensione dell'albero.

Osservazioni sull'altezza:
- Albero composto da una sola foglia → altezza = 0
- Albero con almeno due nodi → altezza del suo sottoalbero più alto + 1.

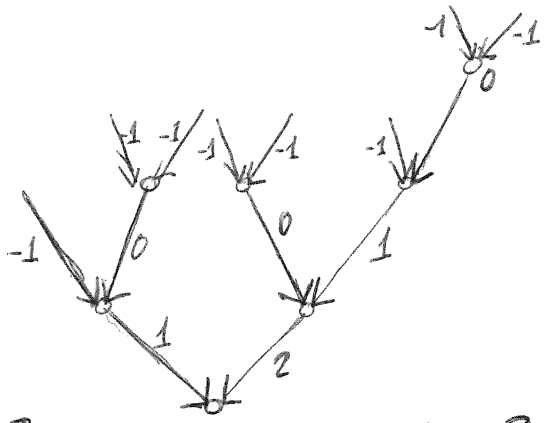
```
Altezza (u)
  IF (u == NULL) {                    /* l'altezza delle
                                         foglie risulta pari a 0
    RETURN -1;
  }
  ELSE {
    altezzaSX = Altezza (u.sx);
    altezzaDX = Altezza (u.dx);
    RETURN max (altezzaSX, altezzaDX) + 1;
  }
```

Si noti che l'algoritmo è conforme alla soluzione generica per i problemi decomponibili.



T

Albero in input



R $\longrightarrow$ 2 + 1 = 3

Albero delle ricorsioni

Visite di alberi

La visita di un albero equivale ad esaminare tutti i nodi in modo sistematico, una e una sola volta.

Corrisponde all'operazione di scansione di una lista

Facciamo corrispondere all'esame del nodo la stampa del suo contenuto.

Visita anticipata (preorder):

- stampa l'elemento contenuto nel nodo
- visita il sottoalbero sinistro
- visita il sottoalbero destro

Visita simmetrica (inorder):

- visita il sottoalbero sinistro
- stampa l'elemento contenuto nel nodo
- visita il sottoalbero destro

Visita posticipata (postorder):

- visita il sottoalbero sinistro
- visita il sottoalbero destro
- stampa l'elemento contenuto nel nodo.

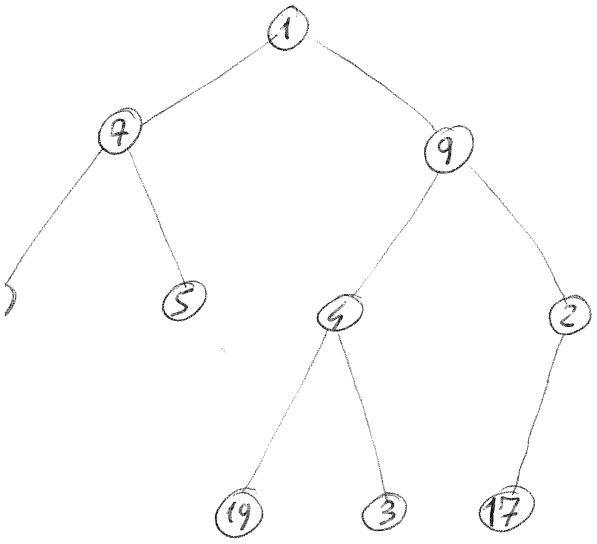
In codice:

```
Anticipata (u):
IF (u != NULL){
   PRINT u.dato;
   Anticipata (u.sx);
   Anticipata (u.dx);
}
```

```
Simmetrica (u):
IF (u != NULL){
   Simmetrica(u.sx)
   PRINT u.dato
   Simmetrica (u.dx)
}
```

```
Posticipata (u):
IF (u != NULL) {
   Posticipata (u.sx)
   Posticipata (u.dx)
   PRINT u.dato;
}
```

Esempio di stampa



ANTICIPATA: 1, 7, 11, 5, 9, 4, 19, 3, 2, 17

SIMMETRICA: 11, 7, 5, 1, 19, 4, 3, 9, 17, 2

POSTICIPATA: 11, 5, 7, 19, 3, 4, 17, 2, 9, 1

Alberi completamente bilanciati

Ricordiamo che:

Albero binario completo = ogni nodo interno ha due figli non vuoti

Albero completamente bilanciato = completo, tutte le foglie hanno la stessa profondità.

Albero completamente bilanciato di altezza h: 2^h foglie, $2^h - 1$ nodi interni

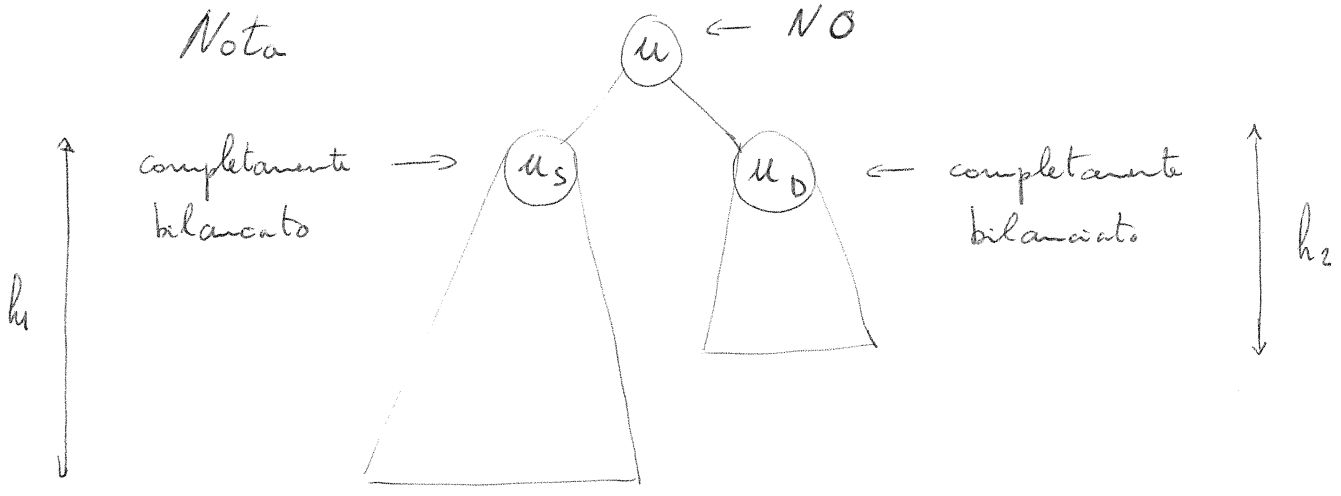
$$n = 2 \cdot 2^h - 1 = 2^{h+1} - 1$$

$$\text{L'altezza } h = \log(n+1) - 1 = O(\log n)$$

$$\text{Albero bilanciato} \triangleq \text{albero tale che } h = O(\log n)$$

Vogliamo progettare un algoritmo che verifica se un albero binario risulta completamente bilanciato.

Nota



$T(u)$ = valore booleano

$T(u_S) = T(u_d) = \text{TRUE} \not\Rightarrow T(u) = \text{TRUE}$

albero radicato in u completamente bilanciato

Occorre pertanto considerare anche le altezze dei sottoalberi

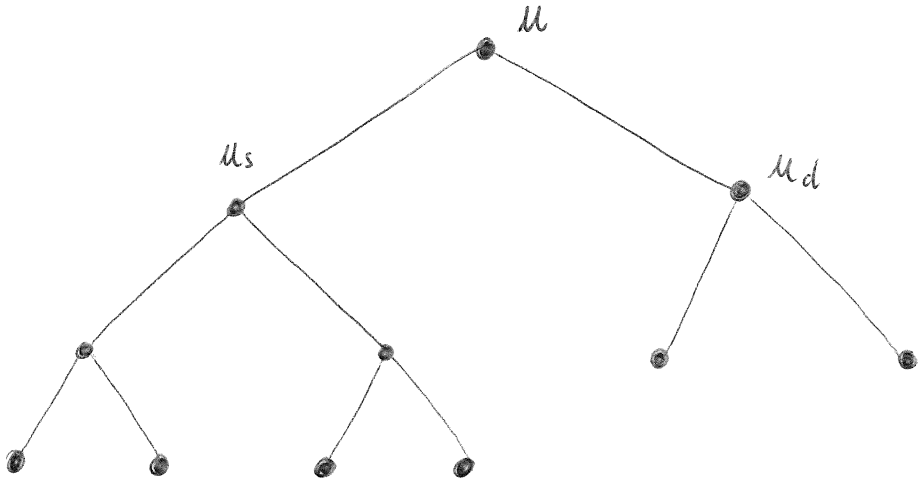
```
CompletamenteBilanciato (u):
    IF (u == NULL) {
        RETURN < TRUE, -1 >
    } ELSE {
        < bilSX, altSX > = CompletamenteBilanciato (u.sx);
        < bilDX, altDX > = CompletamenteBilanciato (u.dx);
①       completamenteBil = bilSX && bilDX && (altSX == altDX);
②       altezza = max (altSX, altDX) + 1;
        RETURN < completamenteBil, altezza >;
    }
```

Restituisce una coppia di valori: $< V, H >$

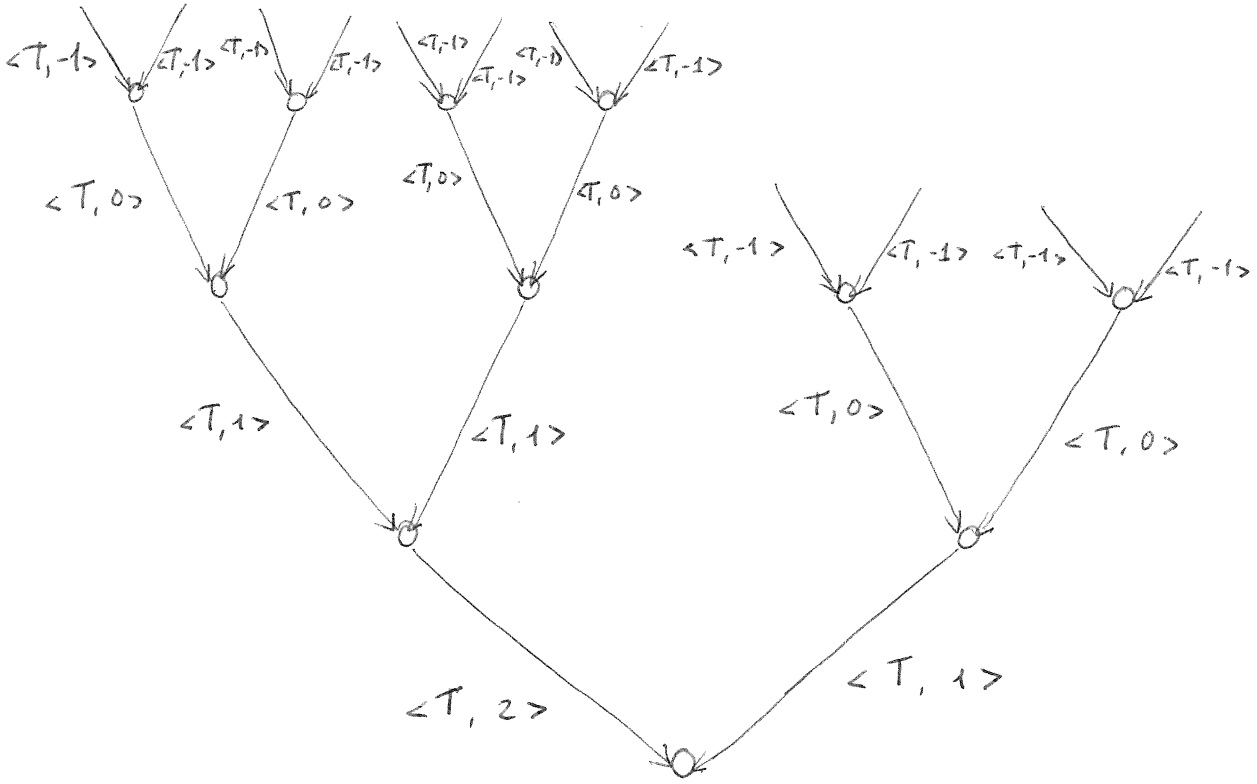
- La prima ① componente è TRUE se e solo se u_s e u_d sono radici di sottoalberi completamente bilanciati e hanno la stessa altezza

- La seconda ② è uguale al massimo tra le altezze dei sottoalberi sinistro e destro + 1.

Esempio di esecuzione



Albero in input



L'algoritmo restituisce FALSE perché pur essendo bilanciati i sottoalberi sinistro e destro, hanno altezza diverse.

Nodi cardine di un albero binario

L'algoritmo che verifica se un albero binario è completamente bilanciato è un esempio di un algoritmo che raccoglie informazioni dai sottoalberi.

Vediamo ora un esempio di algoritmi che non solo raccolgono informazioni dai sottoalberi ma propagano anche informazioni dagli antenati.

Dato un nodo u

- p_u indica la sua profondità
- h_u indica l'altezza del sottoalbero $T(u)$.

Diciamo che u è un nodo cardine se e solo se $p_u = h_u$.

Vogliamo progettare un algoritmo ricorsivo che stampi il contenuto di tutti i nodi cardine presenti in un albero binario.

```
Cardine (u, p):
IF (u == NULL) {
    RETURN -1;
}
ELSE {
    alteraSX = Cardine(u.sx, p+1);
    alteraDX = Cardine(u.dx, p+1);
    altezza = max(alteraSX, alteraDX) + 1;
    IF (p == altezza) print u.dato;
    RETURN altezza;
}
```

La chiamata iniziale è Cardine(r, 0). (r: radice dell'albero; 0: profondità radice)

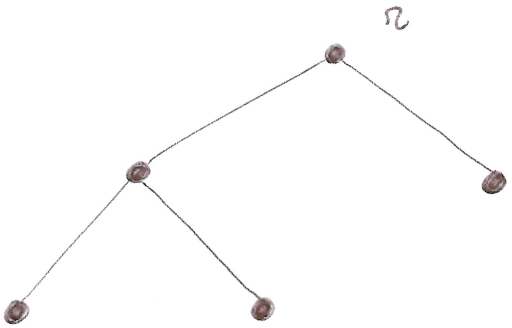
La funzione restituisce l'altezza h_u del nodo u.

Se il nodo corrente u ha profondità p, i figli sono a p+1.

Da notare che:

- la funzione rientra nello schema generale degli algoritmi per problemi decomponibili
- la complessità resta $O(n)$.

Esempio di esecuzione:

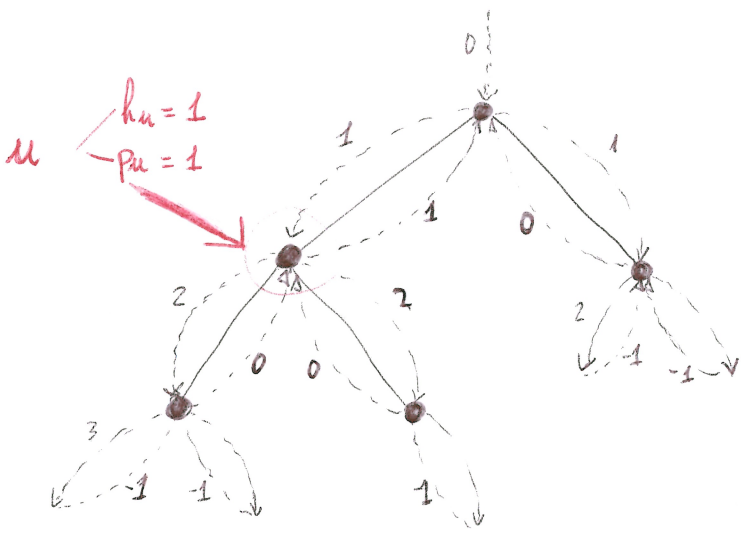


Albero in input

Con archi tratteggiati indichiamo le chiamate ricorsive.

Le etichette denotano la profondità dei nodi (in discesa ↓)

(in salita ↑ il valore dell'altezza del nodo sottostante)



L'istanza della funzione Cardine (u, 1), dopo che le due istanze Cardine (u.sx, 2) e Cardine (u.dx, 2) hanno restituito il valore 0 per le altezze dei figli, stampa il contenuto del nodo, essendo $p_u = h_u = 1$