

Moltiplicazione di interi

- Algoritmo che usiamo comunemente ha tempo di esecuzione $O(n^2)$, dove n è il numero di cifre di ciascun numero

$$\begin{array}{r} 2345 \times \\ 5382 = \\ \hline 4690 \\ 18760 \\ 7035 \\ 11725 \\ \hline 12620790 \end{array}$$

MOLTIPLICAZIONE VELOCE DI INTERI

Ogni numero intero w di n cifre può essere scritto come $10^{n/2} \times w_s + w_d$

- w_s indica il numero formato dalle $n/2$ cifre più significative di w
- w_d denota il numero formato dalle $n/2$ cifre meno significative.

Ad esempio 124100 può essere scritto come $10^3 \times 124 + 100$

Per moltiplicare due numeri x e y , vale l'uguaglianza

$$\begin{aligned}xy &= (10^{n/2} x_s + x_d)(10^{n/2} y_s + y_d) \\ &= 10^n x_s y_s + 10^{n/2}(x_s y_d + x_d y_s) + x_d y_d\end{aligned}$$

DECOMPOSIZIONE: se x e y hanno almeno due cifre, dividili come numeri x_s , x_d , y_s e y_d aventi ciascuno la metà delle cifre.

RICORSIONE: calcola ricorsivamente le moltiplicazioni $x_s y_s$, $x_s y_d$, $x_d y_s$ e $x_d y_d$.

RICOMBINAZIONE: combina i numeri risultanti usando l'uguaglianza riportata sopra.

MOLTIPLICAZIONE VELOCE DI INTERI

- l'algoritmo esegue quattro moltiplicazioni di due numeri di $n/2$ cifre (ad un costo di $T(n/2)$), e tre somme di numeri di n cifre (a un costo $O(n)$)
- la moltiplicazione per il valore 10^k può essere realizzata spostando le cifre di k posizioni verso sinistra e riempiendo di 0 la parte destra
- il costo della decomposizione e della ricombinazione è cn

Vale la relazione di ricorrenza

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \\ 4T(n/2) + cn & \text{altrimenti} \end{cases}$$

MOLTIPLICAZIONE VELOCE DI INTERI

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \\ 4T(n/2) + cn & \text{altrimenti} \end{cases}$$

Assumiamo per semplicità $n = 2^k$ per un certo k e applichiamo iterativamente la relazione di ricorrenza:

$$\begin{aligned} T(n) &\leq cn + 4T(n/2) \leq cn + 4(cn/2 + 4T(n/2^2)) = cn + 2cn + 4^2 T(n/2^2) \\ &\leq cn + 2cn + 4^2(cn/2^2 + 4T(n/2^3)) = cn + 2cn + 2^2 cn + 4^3 T(n/2^3) \\ &\leq \dots \\ &\leq cn + 2cn + 2^2 cn + \dots + 2^{i-1} cn + 4^i T(n/2^i) \\ &= cn \sum_{j=0}^{i-1} 2^j + 4^i T(n/2^i) = cn2^i - cn + 4^i T(n/2^i) \end{aligned}$$

Ponendo $i = k = \log_2 n$ si ha $T(n) \leq cn^2 - cn + n^2 T(1) = O(n^2)$.

MOLTIPLICAZIONE VELOCE DI INTERI

- È possibile progettare un algoritmo più veloce?
- Abbiamo visto che $x y = 10^n x_s y_s + 10^{n/2}(x_s y_d + x_d y_s) + x_d y_d$.
- Osserviamo che sommando e sottraendo $x_s y_s + x_d y_d$ a $x_s y_d + x_d y_s$ si ha

$$\begin{aligned}x_s y_d + x_d y_s &= x_s y_d + x_d y_s + x_s y_s + x_d y_d - x_s y_s - x_d y_d \\ &= x_s y_s + x_d y_d + (x_s y_d + x_d y_s - x_s y_s - x_d y_d)\end{aligned}$$

- Poiché $x_s y_d + x_d y_s - x_s y_s - x_d y_d = -(x_s - x_d) \times (y_s - y_d)$ allora possiamo scrivere

$$x_s y_d + x_d y_s = x_s y_s + x_d y_d - (x_s - x_d) \times (y_s - y_d)$$

- quindi il valore $x_s y_d + x_d y_s$ può essere calcolato facendo uso di $x_s y_s$, $x_d y_d$ e $(x_s - x_d) \times (y_s - y_d)$
- Quindi per computare il prodotto xy sono necessarie tre moltiplicazioni e non più quattro come prima

MOLTIPLICAZIONE VELOCE DI INTERI

Si ha quindi la relazione di ricorrenza

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \\ 3T(n/2) + cn & \text{altrimenti} \end{cases}$$

Assumiamo per semplicità $n = 2^k$, per un certo k , e applichiamo iterativamente la relazione di ricorrenza:

$$\begin{aligned} T(n) &\leq cn + 3T(n/2) \leq cn + 3(cn/2 + 3T(n/2^2)) = cn + (3/2)cn + 3^2 T(n/2^2) \\ &\leq cn + (3/2)cn + 3^2(cn/2^2 + 3T(n/2^3)) = cn + (3/2)cn + (3/2)^2 cn + 3^3 T(n/2^3) \\ &\leq \dots \\ &\leq cn + (3/2)cn + (3/2)^2 cn + \dots + (3/2)^{i-1} cn + 3^i T(n/2^i) \\ &= cn \sum_{j=0}^{i-1} (3/2)^j + 3^i T(n/2^i) = cn \left(\frac{(3/2)^i - 1}{3/2 - 1} \right) + 3^i T(n/2^i) \\ &= 2cn((3/2)^i - 1) + 3^i T(n/2^i) = 2cn(3/2)^i - 2cn + 3^i T(n/2^i) \end{aligned}$$

Continua nella prossima slide

Ponendo $i = k = \log_2 n$ si ha

$$\begin{aligned} T(n) &\leq 2cn(3/2)^{\log_2 n} - 2cn + 3^{\log_2 n} T(1) \\ &= 2cn \left(2^{\log_2(3/2)}\right)^{\log_2 n} - 2cn + \left(2^{\log_2 3}\right)^{\log_2 n} T(1) \\ &= 2cn \left(2^{\log_2 n}\right)^{\log_2(3/2)} - 2cn + \left(2^{\log_2 n}\right)^{\log_2 3} T(1) \\ &= 2cn n^{\log_2(3/2)} - 2cn + n^{\log_2 3} T(1) \\ &= 2cn n^{\log_2 3 - 1} - 2cn + n^{\log_2 3} T(1) \\ &= 2cn^{\log_2 3} - 2cn + n^{\log_2 3} T(1) \\ &\leq 2cn^{\log_2 3} - 2cn + n^{\log_2 3} c_0 \\ &= O(n^{\log_2 3}) = O(n^{1,585}) \end{aligned}$$

SOTTOSEQUENZA DI SOMMA MASSIMA DI UN ARRAY DI NUMERI

Dato un array a di n numeri positivi e negativi trovare la sottosequenza di numeri consecutivi la cui somma è massima. N.B. Se l'array contiene solo numeri positivi, il massimo si ottiene banalmente prendendo come sequenza quella di tutti i numeri dell'array; se l'array contiene solo numeri negativi il massimo si ottiene prendendo come sottosequenza quella formata dalla locazione contenente il numero più grande .

- I soluzione: Per ogni coppia di indici (i, j) con $i \leq j$ dell'array computa la somma degli elementi nella sottosequenza degli elementi di indice compreso tra i e j e restituisci la sottosequenza per cui questa somma è max.
- Costo della I soluzione: $O(n^3)$ perché

$$\begin{aligned} \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1) &= \sum_{i=0}^{n-1} \sum_{k=1}^{n-i} k = \sum_{i=0}^{n-1} (n - i + 1)(n - i)/2 \\ &= \sum_{i=0}^{n-1} ((n - i)^2/2 + (n - i)/2) = \sum_{a=1}^n (a^2/2 + a/2) \\ &= \sum_{a=1}^n a^2/2 + \sum_{a=1}^n a/2 \\ &= 1/2(n(n + 1)(2n + 1)/6) + 1/2(n(n + 1)/2) = \Theta(n^3). \end{aligned}$$

SOTTOSEQUENZA DI SOMMA MASSIMA DI UN ARRAY DI NUMERI

- Il soluzione Osserviamo che la somma degli elementi di indice compreso tra i e j può essere ottenuta sommando $a[j]$ alla somma degli elementi di indice compreso tra i e $j - 1$. Di conseguenza, per ogni i , la somma degli elementi in tutte le sottosequenze che partono da i possono essere computate con un costo totale pari a $\Theta(n - i)$. Il costo totale è quindi

$$\sum_{i=0}^{n-1} \Theta(n - i) = \sum_{i=1}^n \Theta(i) = \Theta\left(\sum_{i=1}^n i\right) = \Theta(n^2)$$

SOTTOSEQUENZA DI SOMMA MASSIMA DI UN ARRAY DI NUMERI

- III soluzione: Divide et Impera

Algoritmo A:

- ① Se $i = j$ viene restituita la sottosequenza formata da $a[i]$
- ② Se $i < j$ si invoca ricorsivamente $A(i, (i + j)/2)$ e $A((i + j)/2 + 1, j)$: la sottosequenza cercata o è una di quelle restituite dalle 2 chiamate ricorsive o si trova a cavallo delle due metà dell'array
- ③ La sottosequenza di somma massima tra quelle che intersecano entrambe le metà dell'array si trova nel seguente modo:
 - si scandisce l'array a partire dall'indice $(i + j)/2$ andando a ritroso fino a che si arriva all'inizio dell'array sommando via via gli elementi scanditi: ad ogni iterazione si confronta la somma ottenuta fino a quel momento con il valore max s_1 delle somme ottenute in precedenza e nel caso aggiorna il max s_1 e l'indice in corrispondenza del quale è stato ottenuto.
 - si scandisce l'array a partire dall'indice $(i + j)/2 + 1$ andando in avanti fino a che o si raggiunge la fine dell'array sommando gli elementi scanditi: ad ogni iterazione si confronta la somma ottenuta fino a quel momento con il valore max s_2 delle somme ottenute in precedenza e nel caso aggiorna il max s_2 e l'indice in corrispondenza del quale è stato ottenuto.
 - La sottosequenza di somma massima tra quelle che intersecano le due metà dell'array è quella di somma $s_1 + s_2$.
- ④ L'algoritmo restituisce la sottosequenza massima tra quella restituita dalla prima chiamata ricorsiva, quella restituita dalla seconda chiamata ricorsiva e quella di somma $s_1 + s_2$

SOTTOSEQUENZA DI SOMMA MASSIMA DI UN ARRAY DI NUMERI

- Tempo di esecuzione dell'algoritmo Divide et Impera

$$T(n) \leq \begin{cases} c_0 & \text{se } n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & \text{altrimenti} \end{cases}$$

Il tempo di esecuzione quindi è $O(n \log n)$.

SOTTOSEQUENZA DI SOMMA MASSIMA DI UN ARRAY DI NUMERI

- IV soluzione: Chiamiamo s_j la sottosequenza di somma massima che tra quelle che terminano in j . Si ha $s_{j+1} = \max\{s_j + a[j + 1], a[j + 1]\}$. Questo valore si calcola in tempo costante per ogni j . L'algoritmo calcola questi valori per ogni j e prende il massimo degli n valori computati. Il tempo dell'algoritmo quindi è $O(n)$.

ESEMPI DI RELAZIONI DI RICORRENZA DELLA FORMA

$$T(n) \leq \alpha T(n/\beta) + n^k$$

- Ricerca binaria

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \text{ oppure } k \text{ è l'elemento centrale} \\ T(n/2) + c & \text{altrimenti} \end{cases}$$

Si ha $\alpha = 1, \beta = 2, k = 0$.

Siccome $\alpha = \beta^k$, siamo nel secondo caso e si ha

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

ESEMPI DI RELAZIONI DI RICORRENZA DELLA FORMA

$$T(n) \leq \alpha T(n/\beta) + n^k$$

Nell'ordinamento per fusione,

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \\ 2T(n/2) + cn & \text{altrimenti} \end{cases}$$

Quindi,

- $\alpha = 2$, $\beta = 2$ e $k = 1$
- siamo nel caso $\alpha = \beta^k$ e quindi $T(n) = O(n^k \log n) = O(n \log n)$.

ESEMPI DI RELAZIONI DI RICORRENZA DELLA FORMA

$$T(n) \leq \alpha T(n/\beta) + n^k$$

- Moltiplicazione veloce di interi: primo algoritmo

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \\ 4T(n/2) + cn & \text{altrimenti} \end{cases}$$

Applicazione del risultato provato:

- si ha che $\alpha = 4$, $\beta = 2$ e $k = 1$
- $\alpha > \beta^k$, quindi si applica il terzo caso e si ha $T(n) = O(n^{\log_2 4}) = O(n^2)$

- Moltiplicazione veloce di interi: secondo algoritmo

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \\ 3T(n/2) + cn & \text{altrimenti} \end{cases}$$

Applicando il risultato dimostrato,

- si ha che $\alpha = 3$, $\beta = 2$ e $k = 1$
- $\alpha > \beta^k$, quindi si applica il terzo caso e si ha $T(n) = O(n^{\log_2 3}) = O(n^{1,585})$

DIVIDE ET IMPERA SU ALBERI

- **Caso base:** per $u = \text{null}$ o una foglia
- **Decomposizione:** riformula il problema per i sottoalberi radicati nei figli di u .
- **Ricombinazione:** ottieni il risultato con Ricombina

```
1 Decomponibile(u):
2   IF (u == null) {
3     RETURN valore base;
4   } ELSE {
5     i=0;
6     FOR( ciascun figlio f di u ){
7
8       risultatiFigli[i] = Decomponibile(f);
9       i=i+1 }
10    RETURN Ricombina(risultatiFigli);
11  }
```

La ricombinazione dei risultati delle chiamate ricorsive sui figli potrebbe essere effettuata anche nel for man mano che vengono ottenuti i risultati delle chiamate sui figli.

DIVIDE ET IMPERA SU ALBERI BINARI

- **Caso base:** per $u = \text{null}$ o una foglia
- **Decomposizione:** riformula il problema per i sottoalberi radicati nei figli $u.\text{sx}$ e $u.\text{dx}$
- **Ricombinazione:** ottieni il risultato con Ricombina

```
1 Decomponibile(u):
2   IF (u == null) {
3     RETURN valore base;
4   } ELSE {
5     risultatoSX = Decomponibile(u.sx);
6     risultatoDx = Decomponibile(u.dx);
7     RETURN Ricombina(risultatoSX, risultatoDx);
8   }
```

Analisi dell'algoritmo Decomponibile

- Assumiamo che il tempo per la decomposizione e la ricombinazione sia costante
- Se escludiamo il tempo impiegato per le chiamate ricorsive, l'algoritmo impiega tempo $O(1 + c_v)$, dove c_v è il numero di figli di v
- Se cominciamo la visita dal nodo w , l'algoritmo viene invocato su tutti i discendenti di w

→ **Tempo totale** = $\sum_{v \in T_w} O(c_v + 1) = O(|T_w|)$

- La visita di tutto l'albero richiede tempo $O(|T|)$
- Se l'albero ha n nodi la visita richiede tempo $T(n) = O(n)$

ALGORITMI RICORSIVI SU ALBERI: DIMENSIONE

Calcolo della dimensione $d =$ numero di nodi

- Caso base: albero vuoto $\Rightarrow d = 0$
- Caso induttivo: $d = 1 +$ dimensione del sottoalbero sinistro $+$ dimensione del sottoalbero destro

```
1 Dimensione( u ):
2   IF (u == null) {
3     RETURN 0;
4   } ELSE {
5     dimensioneSX = Dimensione( u.sx );
6     dimensioneDX = Dimensione( u.dx );
7     RETURN dimensioneSX + dimensioneDX + 1;
8   }
```

Se si vuole conoscere la dimensione di tutto l'albero, si invoca Dimensione con u uguale alla radice

ALGORITMI RICORSIVI SU ALBERI: ALTEZZA

Calcolo dell'altezza h di un nodo:

- caso base per null $\Rightarrow h = -1$
- passo induttivo: $h = 1 +$ massima altezza dei figli

```
1 Altezza( u ):
2   IF (u == null) {
3     RETURN -1;
4   } ELSE {
5     altezzaSX = Altezza( u.sx );
6     altezzaDX = Altezza( u.dx );
7     RETURN max( altezzaSX, altezzaDX ) + 1;
8   }
```

Per calcolare l'altezza dell'albero, si invoca `Altezza` con `u` uguale alla radice

VISITA DI UN ALBERO BINARIO: INORDER

- **simmetrica** (*inorder*):

```
1 Simmetrica( u ):
2   IF (u != null) {
3     Simmetrica( u.sx );
4     elabora(u);
5     Simmetrica( u.dx );
6   }
```

$O(n)$ tempo per n nodi

VISITA DI UN ALBERO BINARIO: PREORDER

- **anticipata** (*preorder*):

```
1 Anticipata( u ):
2   IF (u != null) {
3     elabora(u);
4     Anticipata( u.sx );
5     Anticipata( u.dx );
6   }
```

$O(n)$ tempo per n nodi

VISITA DI UN ALBERO BINARIO: POSTORDER

- **posticipata** (*postorder*):

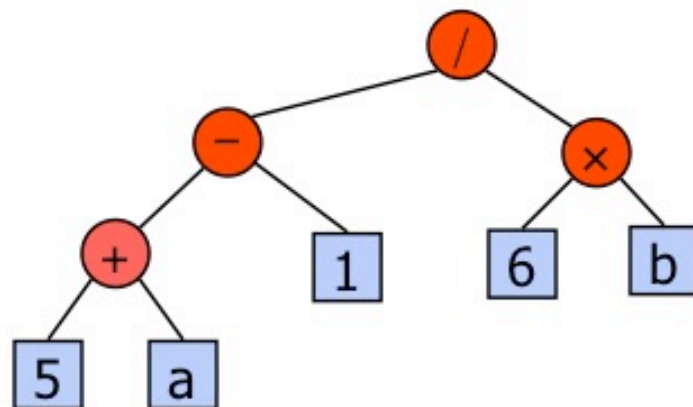
```
1 Posticipata( u ):
2   IF (u != null) {
3     Posticipata( u.sx );
4     Posticipata( u.dx );
5     elabora(u);
6   }
```

$O(n)$ tempo per n nodi

ESEMPIO DELL'USO DELLE VISITE

Esempio dell'uso delle visite: valutazione dell'espressione aritmetica rappresentata da un albero binario

- Albero binario associato ad una espressione:
 - Nodi interni: operatori
 - Nodi esterni: operandi
- Esempio: $((5 + a) - 1) / (6 \times b)$



USO DELLA VISITA POSTORDER PER VALUTARE L'ESPRESSIONE ARITMETICA RAPPRESENTATA DA UN ALBERO BINARIO

```
1 Valuta( u ) :
2   IF (u==null) {
3     RETURN null;
4   }
5   IF (u.sx == null && u.dx==null) {
6     RETURN u.dato;
7   } ELSE {
8     valSinistra=Valuta( u.sx );
9     valDestra= Valuta( u.dx );
10    ris= Calcola(u.dato,valSinistra ,valDestra);
11    RETURN ris;
12  }
```

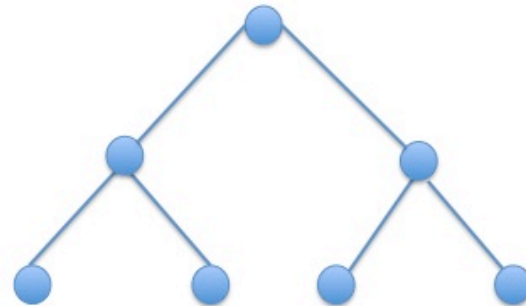
- La funzione *Calcola* invocata su *u.dato*, *valSinistra* e *valDestra*, applica l'operatore memorizzato nel nodo interno *u* ai valori *valSinistra* e *valDestra*.
- N.B.: la condizione del primo if è soddisfatta (*u* è null) solo se inizialmente la funzione *Valuta* è invocata su null. Se inizialmente *Valuta* è invocata su un nodo $u \neq null$ allora la condizione del primo if non sarà mai soddisfatta perché quando è invocata su una foglia, la funzione restituisce il contenuto della foglia.

ALGORITMO PER VERIFICARE SE UN ALBERO BINARIO È COMPLETAMENTE BILANCIATO

Definizioni:

- Albero binario **completo**: ogni nodo interno ha sempre due figli non vuoti
- Albero **completamente bilanciato**: albero completo con tutte le **foglie** alla **stessa profondità**

Esempio:



ALGORITMO PER VERIFICARE SE UN ALBERO BINARIO È COMPLETAMENTE BILANCIATO

- Def. ricorsiva di albero completamente bilanciato:
 - Un albero binario vuoto è completamente bilanciato
 - Una albero binario con almeno un nodo è completamente bilanciato se e solo se il sottoalbero destro e il sottoalbero sinistro della radice sono completamente bilanciati e hanno la stessa altezza (per convenzione, un albero vuoto ha altezza -1)
- N.B. In un albero completamente bilanciato l'altezza dell'albero corrisponde alla profondità di tutte le foglie
- Indichiamo con $T(u)$ il sottoalbero di T radicato in u
- Risolviamo un problema più generale per $T(u)$, calcolandone anche l'altezza oltre che a dire se è completamente bilanciato o meno
- La ricorsione restituisce una coppia (booleano, intero)
- Tempo di risoluzione: $O(n)$ tempo per n nodi

```
1 CompletamenteBilanciato( u ):
2   IF (u == null) {
3     RETURN <TRUE, -1>;
4   } ELSE {
5     <bilSX,altSX> = CompletamenteBilanciato( u.sx );
6     <bilDX,altDX> = CompletamenteBilanciato( u.dx );
7     bil = bilSX && bilDX && (altSX == altDX);
8     altezza = max(altSX, altDX) + 1;
9     RETURN <bil,altezza>;
10  }
```

ALGORITMI RICORSIVI SU ALBERI: PROFONDITÀ DI UN NODO

- La radice ha profondità 0
- I figli della radice hanno profondità pari a 1, e così via
- Un nodo ha profondità p ha i figli a profondità $p + 1$

Versione iterativa dell'algorithmo per calcolare la profondità di un nodo u

```
p = 0;
WHILE (u.padre != null) {
    p = p + 1;
    u = u.padre;
}
```

Definizione ricorsiva di profondità di un nodo:

- La radice ha profondità 0
- I nodi diversi dalla radice hanno profondità pari alla profondità del padre + 1

Versione ricorsiva dell'algorithmo per calcolare la profondità di un nodo u

```
1 Profondita( u ):
2     IF (u.padre==null) {
3         RETURN 0;
4     }
5     RETURN profondita(u.padre)+1;
```

TRASMISSIONE DELL'INFORMAZIONE TRA CHIAMATE RICORSIVE

- **postorder** : l'informazione è trasferita dalle foglie alla radice
 - la soluzione del problema per $T(u)$ può essere ottenuta dalla soluzioni dei sottoproblemi per $T(u.sx)$ e $T(u.dx)$
- **passaggio dei parametri** : informazione passata attraverso i parametri dalla radice alle foglie
 - la soluzione del problema per $T(u)$ può essere ottenuta utilizzando l'informazione raccolta dalla radice fino al nodo u

Esempio: stampa la profondità di tutti i nodi

```
1 Profondita( u, p ):
2   IF (u != null) {
3     PRINT profondità di u è pari a p;
4     Profondita( u.sx, p+1 );
5     Profondita( u.dx, p+1 );
6   }
```

Il parametro p indica la profondità del nodo u . Se vogliamo stampare le profondità di tutti i nodi dobbiamo invocare la funzione con u uguale all'indirizzo della radice dell'albero e $p = 0$.

ALGORITMO PER TROVARE I NODI CARDINE

Trasferiamo informazione simultaneamente dalle foglie alla radice e dalla radice verso le foglie combinando i due approcci della slide precedente

- Nodo u è cardine se e solo se $\text{profondita}(u) = \text{altezza}(T(u))$

```
1 Cardine( u, p ):
2   IF (u == null) {
3     RETURN -1;
4   } ELSE {
5     altezzaSX = Cardine( u.sx, p+1 );
6     altezzaDX = Cardine( u.dx, p+1 );
7     altezza = max( altezzaSX, altezzaDX ) + 1;
8     IF (p == altezza) PRINT u.dato;
9     RETURN altezza;
10  }
```