Capitolo 3
Divide et impera

Lucidi tratti da
P. Crescenzi · G. Gambosi · R. Grossi · G. Rossi
Strutture di dati e algoritmi
Progettazione, analisi e visualizzazione
Addison-Wesley, 2012
`http://algoritmica.org`

1. I lucidi sono utilizzabili dai soli docenti e se ne sconsiglia la distribuzione agli studenti: oltre al rischio di violare una qualche forma di copyright, il problema principale è che gli studenti studino in modo superficiale la materia senza il necessario approfondimento e la dovuta riflessione che la lettura del libro fornisce
2. Il simbolo [alvie] nei lucidi indica l'uso di ALVIE per visualizzare il corrispettivo algoritmo: per un proficuo rendimento dello strumento, conviene esaminare in anticipo la visualizzazione per determinare i punti salienti da mostrare a lezione (l'intera visualizzazione potrebbe risultare altrimenti noiosa)

Il paradigma "divide et impera"

Strutturato in tre fasi.

Decomposizione: identificazione di un piccolo numero di sotto-problemi dello stesso tipo, ciascuno definito su un insieme dei dati di dimensione inferiore a quello di partenza.

Ricorsione: soluzione ricorsiva di ciascun sotto-problema fino a ottenere sotto-problemi di dimensioni tali da poter essere risolti direttamente.

Ricombinazione: combinazione delle soluzioni dei sotto-problemi per fornire una soluzione al problema di partenza.

Ordinamento per fusione

L'algoritmo di ordinamento per fusione (*mergesort*), opera in tempo $O(n \log n)$ secondo il paradigma divide et impera.

Decomposizione: se la sequenza ha almeno due elementi, viene divisa in due sotto-sequenze uguali (o quasi) in lunghezza.

Ricorsione: le due sotto-sequenze sono ordinate ricorsivamente.

Ricombinazione: le due sotto-sequenze ordinate sono fuse in un'unica sequenza ordinata.

MergeSort

1. Dividi a metà l'array
2. Ordina separatamente le due metà
3. Fondi le due metà

```
1 MergeSort( a, sinistra, destra ):
2   IF (sinistra < destra) {
3     centro = (sinistra+destra)/2;
4     MergeSort( a, sinistra, centro );
5     MergeSort( a, centro+1, destra );
6     Fusione( a, sinistra, centro, destra );
7   }
```

[alvie]

Fusione

- Le due metà possono essere fuse in tempo lineare:
  - due mazzi di carte, ciascun mazzo ordinato
  - confronta la carta in cima di un mazzo con quella dell'altro
  - rimuovi la minima tra le due
  - se un mazzo si svuota, prendi le rimanenti carte nell'altro
- Ad ogni confronto, sistemiamo una carta nella sequenza ordinata: $O(n)$ tempo

Fusione

```
1  Fusione( a, sx, cx, dx ):
2    i = sx; j = cx+1; k = 0;
3    WHILE ((i <= cx) && (j <= dx)) {
4      IF (a[i] <= a[j]) {
5        b[k] = a[i]; i = i+1;
6      } ELSE {
7        b[k] = a[j]; j = j+1;
8      }
9      k = k+1;
10   }
11   FOR ( ; i <= cx; i = i+1, k = k+1)
12     b[k] = a[i];
13   FOR ( ; j <= dx; j = j+1, k = k+1)
14     b[k] = a[j];
15   FOR (i = sx; i <= dx; i = i+1)
16     a[i] = b[i-sx];
```

[alvie]

Ordinamento per fusione

Numero di passi $T(n)$ definito da una relazione di ricorrenza

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

da questo risulta $T(n) = O(\log n)$

Teorema fondamentale delle ricorrenze

Data la relazione di ricorrenza

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq n_0 \\ \alpha T(n/\beta) + cf(n) & \text{altrimenti} \end{cases}$$

dove $f(n)$ è una funzione non decrescente e $\alpha \geq 1$, $\beta > 1$ e $n_0, c_0, c > 0$.

Se esistono due costanti positive γ e n_0' tali che $\alpha\, f(n/\beta) = \gamma\, f(n)$ per ogni $n \geq n_0'$, allora la relazione di ricorrenza ha le seguenti soluzioni per ogni n:

1. $T(n) = O(f(n))$ se $\gamma < 1$;
2. $T(n) = O(f(n) \log_\beta n)$ se $\gamma = 1$;
3. $T(n) = O(n^{\log_\beta \alpha})$ se $\gamma > 1$.

Teorema fondamentale e mergesort

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 $f(n) = n$
- secondo caso del teorema, in quanto $\alpha f(n/\beta) = 2(n/2) = n = f(n)$, e quindi $\gamma = 1$
- il numero di passi è $O(n \log_2 n) = O(n \log n)$.

Complessità del problema di ordinare un array

Π = problema dell'ordinamento di un array
A = algoritmo `MergeSort`
Limite superiore per Π è $\mathbf{O(n \log n)}$
Limite inferiore per Π è $\mathbf{\Omega(n \log n)}$

1. Sia A^* un generico algoritmo di ordinamento
2. A^* deve discernere tra almeno $n!$ situazioni (le possibili permutazioni che danno luogo a un array ordinato)
3. A^* usa confronti tra coppie di elementi: risultato in $\{<, =, >\}$
4. Dopo t confronti, A^* può distinguere tra al più 3^t situazioni
5. Ne deriva $3^t \geq n! \Rightarrow t \geq \log_3(n!) > \log_3(n/2)^{n/2} = \Omega(n \log n)$ confronti

Ricerca di una chiave in un array

Dato un array a di n elementi e una chiave di ricerca k, trovare un valore `indice` tale che $a[\texttt{indice}] = k$

- Occorre esaminare tutti gli elementi con il metodo della **ricerca sequenziale**. L'algoritmo scandisce, uno dopo l'altro, i valori degli elementi contenuti nell'array: al termine del ciclo, se la chiave k è stata trovata, ne viene restituito l'indice.
- Caso pessimo: la chiave cercata non è tra quelle nella sequenza. Il ciclo scorre tutti gli elementi dell'array, richiedendo un numero di operazioni proporzionale al numero di elementi presenti nella sequenza, e quindi tempo $O(n)$.

```
1  RicercaSequenziale( a, k ):
2    trovato = FALSE;
3    indice = -1;
4    FOR (i = 0; (i<n) && (!trovato); i = i+1) {
5      IF (a[i] == k) {
6        trovato = TRUE;
7        indice = i;
8      }
9    }
10   RETURN indice;
```

Ricerca in un array ordinato

Tempo di ricerca si riduce da $O(n)$ a $O(\log n)$: se $n \approx 10^6$ elementi, poche decine di confronti
Idea: *se in un elenco telefonico prendiamo una pagina a metà, o troviamo il cognome cercato in quella pagina oppure possiamo scartare metà dell'elenco*

1. La chiave k viene confrontata con l'elemento che si trova in posizione centrale nell'array, $a[n/2]$.
2. Se k è minore di tale elemento, il procedimento viene ripetuto nel segmento costituito dagli elementi che precedono $a[n/2]$. Altrimenti, viene ripetuto in quello costituito dagli elementi che lo seguono.
3. Il procedimento termina nel momento in cui k coincide con l'elemento centrale del segmento corrente (con esito positivo) oppure il segmento diventa vuoto (con esito negativo).

Ricerca binaria: versione iterativa

```
1  RicercaBinariaIterativa( a, k ):
2    sinistra = 0;
3    destra = n-1;
4    trovato = FALSE;
5    indice = -1;
6    WHILE ((sinistra <= destra) && (!trovato)) {
7      centro = (sinistra+destra)/2;
8      IF (a[centro] > k) {
9        destra = centro-1;
10     } ELSE IF (a[centro] < k) {
11       sinistra = centro+1;
12     } ELSE {
13       indice = centro;
14       trovato = TRUE;
15     }
16   }
17   RETURN indice;
```

Analisi della complessità

- A ogni iterazione del ciclo `while`, viene **dimezzato** il numero di elementi nel segmento di ricerca $a[\texttt{sinistra}, \texttt{destra}]$: da n, abbiamo $n/2$, $n/4$ e così via
- All'i-esima iterazione, abbiamo al più $n/2^i$ elementi in cui cercare
- Il caso pessimo è per i^* tale che $n/2^{i^*} = 1$, ossia per $i^* = O(\log n)$
- Ogni iterazione `while` richiede $O(1)$ tempo, per cui `RicercaBinariaIterativa` richiede $O(\log n)$ tempo

Paradigma della ricerca binaria

Viene usato in diverse situazioni: per esempio, indovinare un numero positivo con domande del tipo "$a \leq b$?"

1. Chiedi se il numero è $\leq 2^i$ per $i = 1, 2, \ldots$
2. Trova il più piccolo i^* tale che il numero è $\leq 2^{i^*}$
3. Itera il dimezzamento nell'intervallo $[2^{i^*-1}, 2^{i^*}]$

- Per la ricerca in un array ordinato, il paradigma è ottimo come numero di confronti

Ricerca binaria: versione ricorsiva

```
1  RicercaBinariaRicorsiva( a,k,sinistra,destra ):
2    IF (sinistra == destra) {
3      IF (k == a[sinistra]) {
4        RETURN sinistra;
5      } ELSE {
6        RETURN -1;
7      }
8    }
9    c = (sinistra+destra)/2;
10   IF (k <= a[c]) {
11     RETURN RicercaBinariaRicorsiva( a,k,sinistra,c );
12   } ELSE {
13     RETURN RicercaBinariaRicorsiva( a,k,c+1,destra );
14   }
```

Paradigma divide et impera

1. **Caso base**: righe 2–8
2. **Decomposizione**: riga 9
3. **Ricorsione e ricombinazione**: righe 10–14

Analisi mediante relazione di ricorrenza

- Se il segmento all'interno del quale stiamo cercando una chiave è costituito da un solo elemento, allora l'algoritmo esegue un numero costante c_0 di operazioni.
- Altrimenti, il numero di operazioni eseguite è pari a una costante c più il numero di passi richiesto dalla ricerca della chiave in un segmento di dimensione pari alla metà di quello attuale.

Il numero totale $T(n)$ di passi eseguiti su un array di n elementi verifica la relazione di ricorrenza:

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

La ricerca binaria in un array ordinato richiede $O(\log n)$ passi.

Lower bound sulla ricerca

Sia A un qualunque algoritmo di ricerca che usa confronti tra coppie di elementi: A deve discernere tra $n+1$ situazioni (la chiave cercata appare in una delle n posizioni della sequenza oppure non appare nella sequenza stessa)

- A esegue dei confronti, ognuno dei quali dà luogo a tre possibili riposte in $[<, =, >]$.
- Dopo t confronti di chiavi (mai esaminate prima), l'algoritmo A può discernere al più 3^t situazioni.
- Poiché le situazioni da discernere sono $n+1$, deve valere $3^t \geq n+1$.
- Ne deriva che occorrono $t \geq \log_3(n+1) = \Omega(\log n)$ confronti: ciò rappresenta un limite inferiore per il problema della ricerca per confronti.

Conseguenza: l'algoritmo di ricerca binaria è asintoticamente ottimo.

Ordinamento per distribuzione

L'algoritmo di ordinamento per distribuzione (*quicksort* opera nel modo seguente.

Decomposizione: se la sequenza ha almeno due elementi, scegli un elemento **pivot** e dividi la sequenza in due sotto-sequenze: la prima contiene elementi minori o uguali al pivot e la seconda contiene elementi maggiori o uguali.

Ricorsione: ordina ricorsivamente le due sotto-sequenze.

Ricombinazione: concatena le due sotto-sequenze ordinate in un'unica sequenza ordinata.

```
1  QuickSort( a, sinistra, destra ):
2                                          ⟨pre: 0 ≤ sinistra, destra ≤ n − 1⟩
3    IF (sinistra < destra) {
4      scegli pivot nell'intervallo [sinistra...destra];
5      rango = Distribuzione( a, sinistra, pivot, destra );
6      QuickSort( a, sinistra, rango-1 );
7      QuickSort( a, rango+1, destra );
8    }
```

[alvie]

Distribuzione

- Data la posizione `px` del pivot in un segmento $a[\texttt{sx}, \texttt{dx}]$:
  - scambia gli elementi $a[\texttt{px}]$ e $a[\texttt{dx}]$, se $\texttt{px} \neq \texttt{dx}$
  - doppia scansione che usa due indici cursori `i` e `j`
  - `i` parte da `sx` e va verso destra fino a che $a[\texttt{i}] > \texttt{pivot}$
  - `j` parte da $\texttt{dx} - 1$ e va verso sinistra fino a che $a[\texttt{j}] < \texttt{pivot}$
  - scambia $a[\texttt{i}]$ con $a[\texttt{j}]$ e riparti con la doppia scansione
  - rimetti il pivot nella sua posizione corretta
- Ad ogni confronto, incrementiamo `i` oppure decrementiamo `j`: $O(n)$ tempo

Ordinamento per distribuzione

```
1  Distribuzione( a, sx, px, dx ):
2    IF (px != dx) Scambia( px, dx );
3    i = sx;
4    j = dx-1;
5    WHILE (i <= j) {
6      WHILE ((i <= j) && (A[i] <= A[dx]))
7        i = i+1;
8      WHILE ((i <= j) && (A[j] => A[dx]))
9        j = j-1;
10     IF (i < j) Scambia( i, j );
11   }
12   IF (i != dx) Scambia( i, dx );
13   RETURN i;
```

$\langle$*pre:* $0 \leq \texttt{sx} \leq \texttt{px} \leq \texttt{dx} \leq n-1\rangle$

```
1  Scambia( i, j ):
2    temp = a[j]; a[j] = a[i]; a[i] = temp;
```

$\langle$*pre:* $\texttt{sx} \leq \texttt{i}, \texttt{j} \leq \texttt{dx}\rangle$

Ordinamento per distribuzione

Relazione di ricorrenza per il tempo $T(n)$ di esecuzione dell'algoritmo.

- Caso base: $T(n) \leq c_0$ per $n \leq 1$.
- Passo ricorsivo: sia r il rango dell'elemento pivot. Ci sono $r-1$ elementi a sinistra del pivot e $n-r$ elementi a destra, per cui $T(n) \leq T(r-1) + T(n-r) + cn$.
- caso pessimo:
  - il pivot è tutto a sinistra ($r = 1$) oppure tutto a destra ($r = n$). In entrambi i casi, la relazione diventa $T(n) \leq T(n-1) + T(0) + cn \leq T(n-1) + c'n$ per un'opportuna costante c'
  - la relazione $T(n) \leq T(n-1) + c'n$ fornisce $T(n) = O(n^2)$
  - in questa situazione, il costo è simile a quella dell'ordinamento per selezione o per inserimento.
- caso ottimo:
  - la distribuzione è bilanciata ($r = n/2$)), la ricorsione avviene su ciascuna metà
  - applicando il teorema delle ricorrenze, possiamo mostrare che il costo è di $O(n \log n)$ tempo
  - in questa situazione, il costo è simile a quella dell'ordinamento per fusione.
- caso medio: $O(n \log n)$ passi. La'lgoritmo è veloce in pratica.

Selezione per distribuzione

Problema: selezione dell'elemento con rango r in un array a di n elementi distinti.

- Si vuole evitare di ordinare a (quindi tempo inferiore a $O(n \log n)$)
- Il problema diventa trovare il minimo quando $r = 1$ e il massimo quando $r = n$.

Osservazione: la funzione `Distribuzione` permette di trovare il rango del pivot, posizionando tutti gli elementi di rango inferiore alla sua sinistra e tutti quelli di rango superiore alla sua destra.

Possiamo modificare il codice del quicksort procedendo ricorsivamente nel *solo* segmento dell'array contenente l'elemento da selezionare.

La ricorsione ha termine quando il segmento è composto da un solo elemento.

Selezione per distribuzione

```
1  QuickSelect( a, sinistra, r, destra ):
2                                   ⟨pre: 0 ≤ sinistra ≤ r − 1 ≤ destra ≤ n − 1⟩
3    IF (sinistra == destra) {
4      RETURN a[sinistra];
5    } ELSE {
6      scegli pivot nell'intervallo [sinistra...destra];
7      rango = Distribuzione( a, sinistra, pivot, destra );
8      IF (r-1 == rango) {
9        RETURN a[rango];
10     } ELSE IF (r-1 < rango) {
11       RETURN QuickSelect( a, sinistra, r, rango-1 );
12     } ELSE {
13       RETURN QuickSelect( a, rango+1, r, destra );
14     }
15   }
```

Moltiplicazione veloce di interi

- Interi rappresentati come array di cifre
- Per la somma, l'algoritmo che consiste nell'addizionare le singole cifre propagando l'eventuale riporto, richiede $O(n)$ passi ed è quindi ottimo
- Per il prodotto, l'algoritmo elementare richiede tempo $O(n^2)$

Tempo di esecuzione della moltiplicazione riducibile mediante applicazione del "divide et impera"

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.

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

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

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 suddetta.

Moltiplicazione veloce di interi

- l'algoritmo esegue quattro moltiplicazioni di due numeri di $n/2$ cifre (a un costo $T(n/2)$), e tre somme di due 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 $c\,n$

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

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

Applicazione del teorema fondamentale delle ricorrenze

- si ha che $\alpha = 4$, $\beta = 2$ e $f(n) = n$
- $\alpha f(n/\beta) = 4(n/2) = 2n = 2f(n)$, quindi si applica il terzo caso del teorema con $\gamma = 2$
- ne deriva $O(n^{\log 4}) = O(n^2)$, non migliorando quindi le prestazioni

Osserviamo però che il valore $x_s y_d + x_d y_s$ può essere calcolato facendo uso degli altri due valori $x_s y_s$ e $x_d y_d$ nel modo seguente:

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

Quindi sono necessarie tre moltiplicazioni e non quattro.

Moltiplicazione veloce di interi

```
1  MoltiplicazioneVeloce( x, y, n ):                    ⟨pre: x e y interi di n cifre⟩
2    IF (n == 1) {
3      prodotto[1] = (x[1] × y[1]) / 10;
4      prodotto[2] = (x[1] × y[1]) % 10;
5    } ELSE {
6      xs[0] = xd[0] = ys[0] = yd[0] = 1;
7      FOR (i = 1; i <= n/2; i = i + 1) {
8        xs[i] = x[i]; ys[i] = y[i];
9        xd[i] = x[i + n/2]; yd[i] = y[i + n/2];
10     }
11     p1 = MoltiplicazioneVeloce( xs, ys, n/2 );
12     FOR (i = 0; i <= n; i = i+1)
13       { prodotto[i] = p1[i]; prodotto[i+n] = 0; }
14     p2 = MoltiplicazioneVeloce( xd, yd, n/2 );
15     xd[0] = yd[0] = -1;
16     p3 = MoltiplicazioneVeloce( Somma(xs,xd), Somma(ys,yd), n/2);
17     p3[0] = -p3[0];
18     add = Somma( p1, p2, p3 );
19     parziale[0] = add[0];
20     FOR (i = 1; i <= 3 × n/2; i = i+1)
21       { parziale[i] = add[i + n/2]; parziale[i + 3 × n/2] = 0; }
22     prodotto = Somma( prodotto, parziale, p2 );
23   }
24   prodotto[0] = x[0] × y[0];
25   RETURN prodotto;                    ⟨post: prodotto intero di 2n cifre⟩
```

Moltiplicazione veloce di interi

Il numero totale di passi eseguiti è dato da

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

Applicando il teorema fondamentale delle ricorrenze,

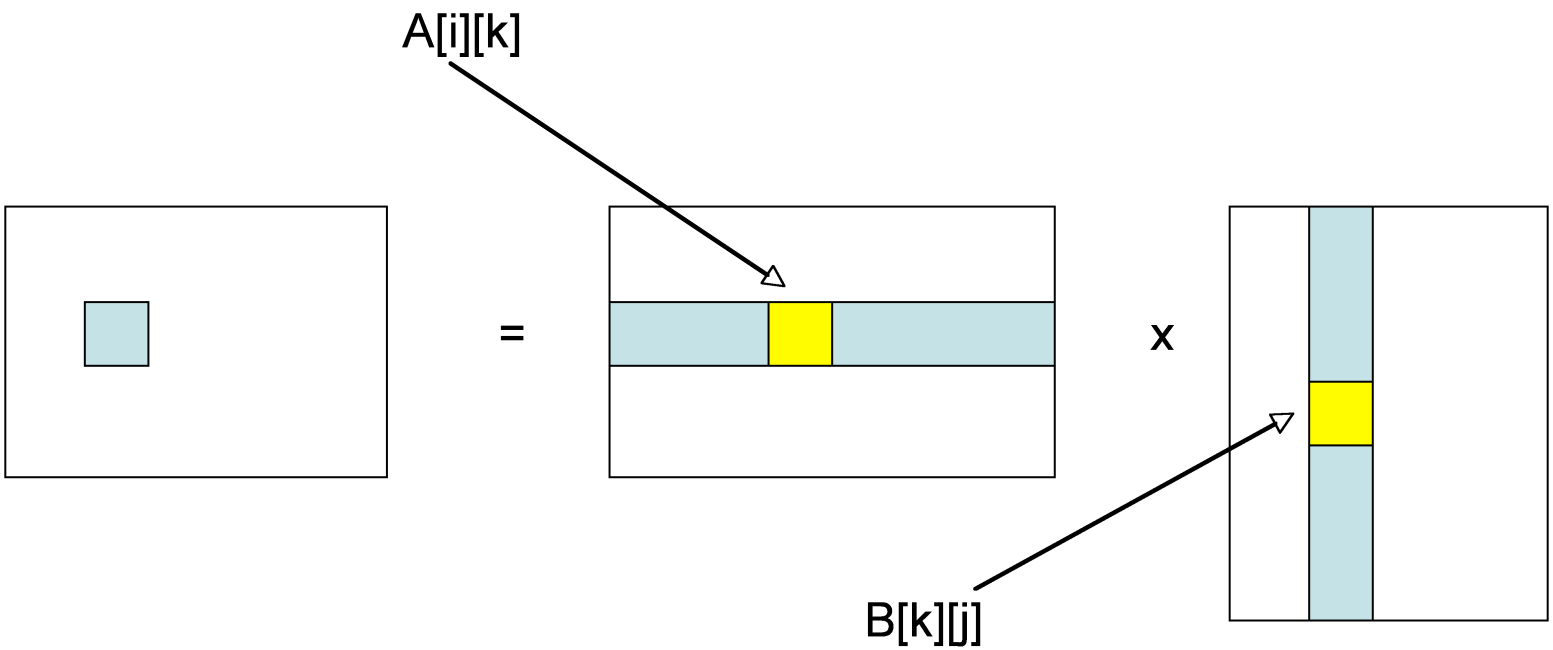
- si ha che $\alpha = 3$, $\beta = 2$ e $f(n) = n$
- si applica il terzo caso del teorema con $\gamma = \frac{3}{2}$
- ne deriva $O(n^{\log_2 3}) = O(n^{1,585})$

Opus libri: grafica e moltiplicazione di matrici

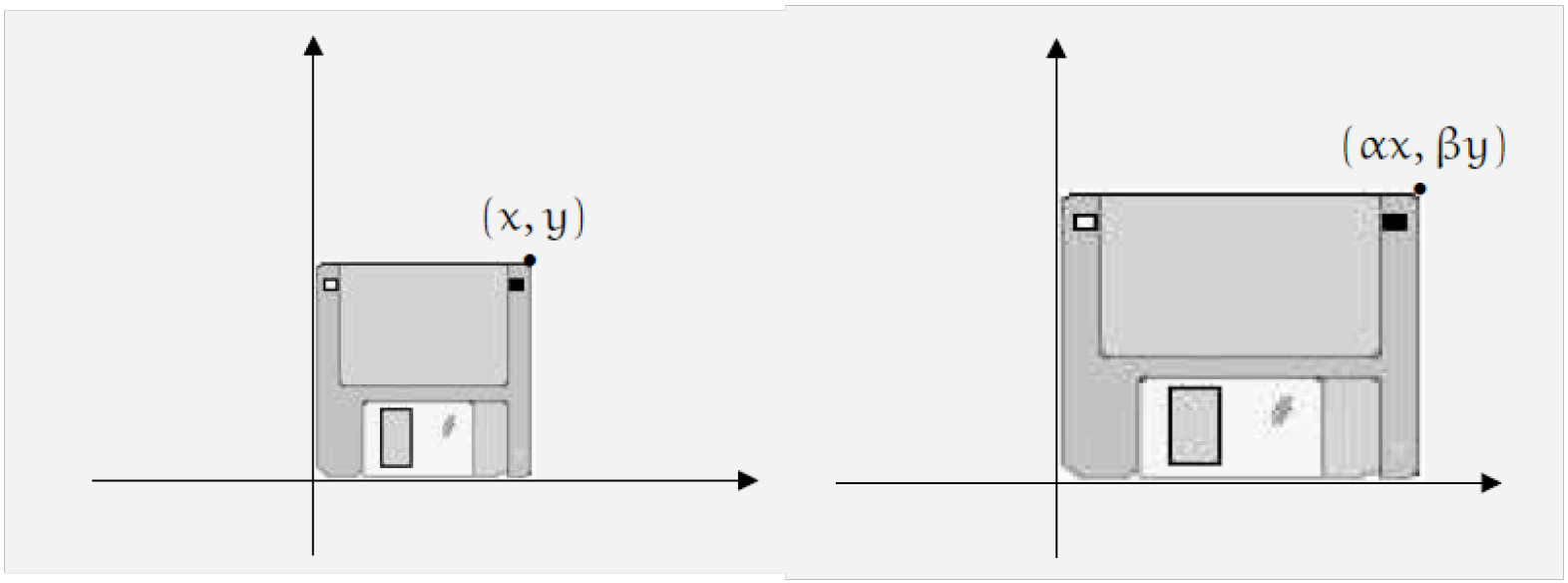
Matrice *frame buffer* (memoria video)
$A[i][j]$ = informazione per il *pixel* in riga i e colonna j (colore e luminosità)
Scene grafiche (ad esempio, di un videogioco):
modello 3-dimensionale "proiettato" sul frame buffer 2-dimensionale (*rendering*)
modello 3-dimensionale: superfici 3-D "triangolarizzate"
I vertici (x, y, z) di ogni triangolo sono array o vettori $[x, y, z, 1]$ (la quarta dimensione serve per la traslazione)

- Prodotto $A \times B$ di due matrici (di taglia $r \times s$ e $s \times t$) è la matrice C (di taglia $r \times t$):

$$C[i][j] = \sum_{k=0}^{s-1} A[i][k]B[k][j]$$

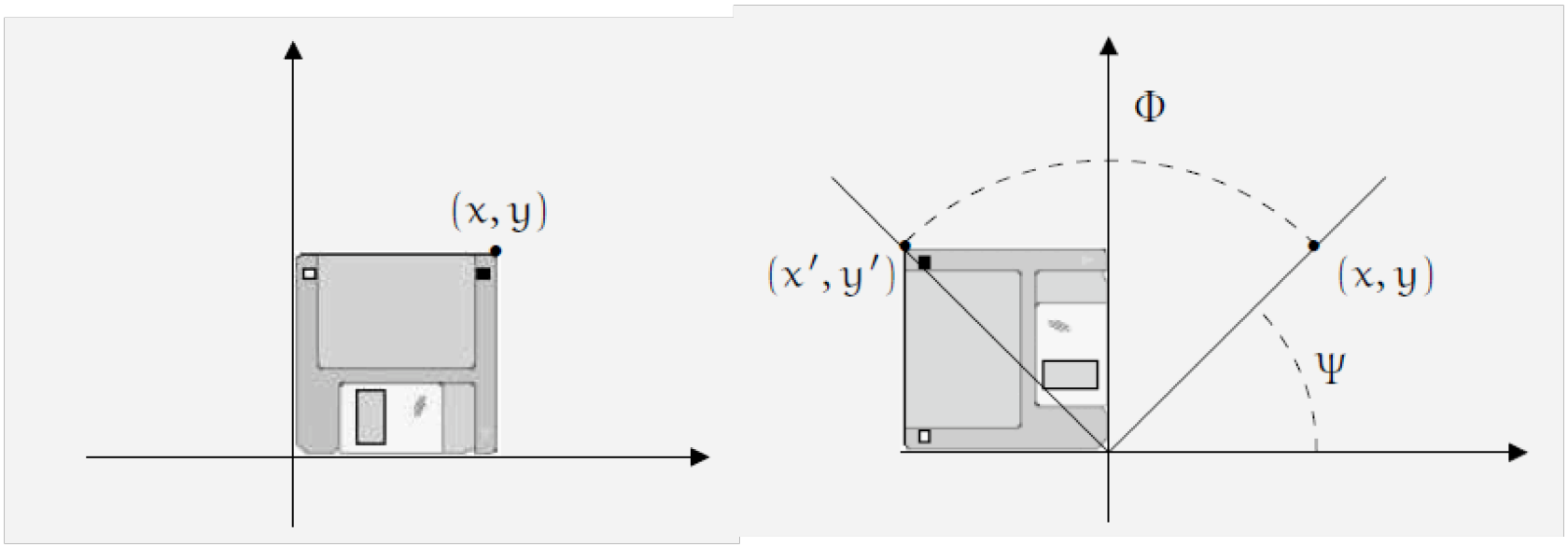


Operazioni su matrici e grafica: scalare (dimensione 2)



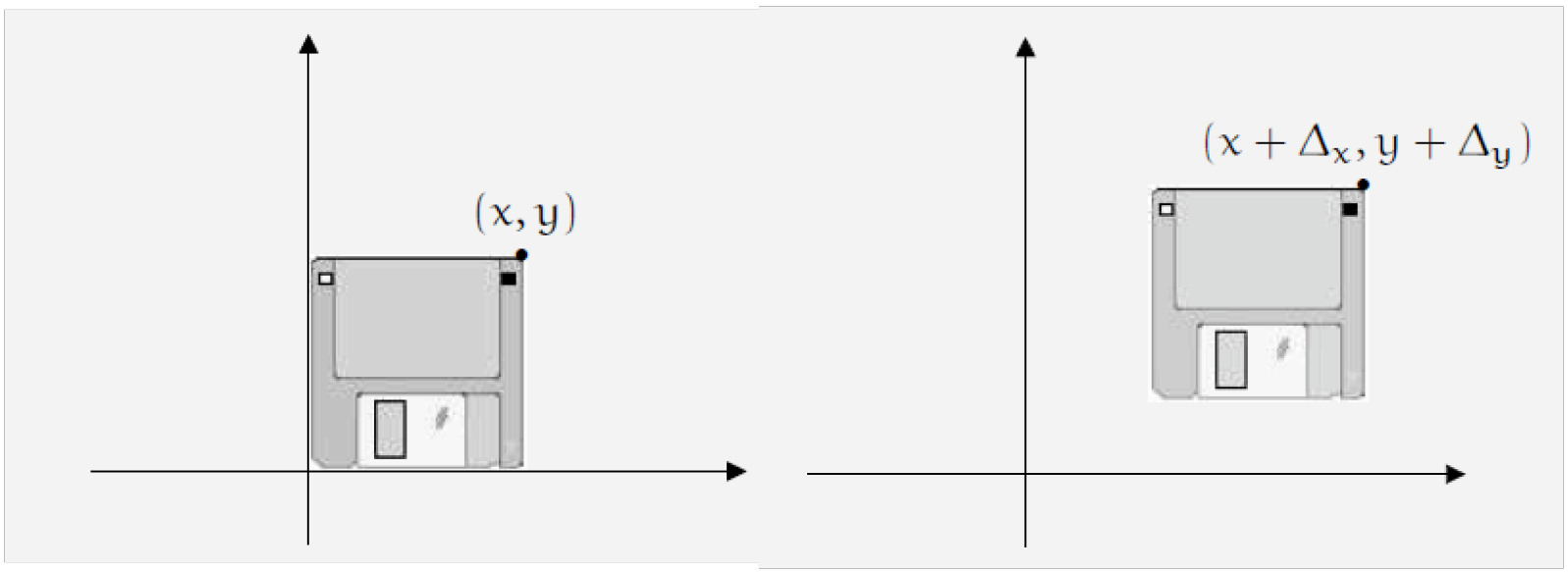
$$[x, y, 1] \times \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & 1 \end{bmatrix} = [\alpha x, \beta y, 1]$$

Operazioni su matrici e grafica: ruotare (dimensione 2)



$$[x, y, 1] \times \begin{bmatrix} \cos\Phi & \sin\Phi & 0 \\ -\sin\Phi & \cos\Phi & 0 \\ 0 & 0 & 1 \end{bmatrix} = [x', y', 1]$$

Operazioni su matrici e grafica: traslare (dimensione 2)



$$[x, y, 1] \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \Delta_x & \Delta_y & 1 \end{bmatrix} = [x + \Delta_x, y + \Delta_y, 1]$$

Sequenza di trasformazioni in una scena

Per ogni vertice (dei numerosi triangoli), applica

$$[x, y, z, 1] \times A_0 \times A_1 \times \cdots \times A_{n-1}$$

Trucco: calcola prima

$$A^* = A_0 \times A_1 \times \cdots \times A_{n-1}$$

e applica in parallelo l'operazione

$$[x, y, z, 1] \times A^*$$

(le schede grafiche eseguono queste operazioni in parallelo con prestazioni notevoli)

Moltiplicazione veloce tra matrici

- Algoritmo immediato, $O(r \times s \times t)$ operazioni:

```
1  ProdottoMatrici( A, B ):
2    FOR (i = 0; i < r; i = i+1)
3      FOR (j = 0; j < t; j = j+1) {
4        C[i][j] = 0;
5        FOR (k = 0; k < s; k = k+1)
6          C[i][j] = C[i][j] + A[i][k] × B[k][j];
7      }
8    RETURN C;
```

- Studiamo il caso $r = t = s = n$: costo $O(n^3)$
- Idea della moltiplicazione veloce tra interi, estesa a matrici: costo $O(n^\epsilon)$ dove $2 \leq \epsilon < 3$

Moltiplicazione veloce tra matrici

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

usando le equazioni

$$\begin{aligned} v_0 &= (b - d)(g + h) & v_4 &= a(f - h) \\ v_1 &= (a + d)(e + h) & v_5 &= d(g - e) \\ v_2 &= (a - c)(e + f) & v_6 &= e(c + d) \\ v_3 &= h(a + b) \end{aligned}$$

possiamo verificare che

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} v_0 + v_1 - v_3 + v_5 & v_3 + v_4 \\ v_5 + v_6 & v_1 - v_2 + v_4 - v_6 \end{bmatrix}$$

Analisi della moltiplicazione

- Risparmiamo una moltiplicazione ad ogni passo ricorsivo

$$T(n) = 7T(n/2) + O(n^2)$$

  Soluzione: $T(n) = O(n^{\log_2 7}) = O(n^{2,807\ldots})$
  teorema fondamentale: caso $\gamma > 1$

Congettura: complessità del problema della moltiplicazione tra matrici è $\Theta(n^\epsilon)$ dove $2 \leq \epsilon \leq 2,376\ldots$

Opus libri: il problema della coppia più vicina

Problema: trovare la coppia di punti più vicina tra un insieme di punti del piano.

Il problema può essere risolto in tempo $O(n^2)$ calcolando le distanze tra tutti i punti.

Utilizzando la tecnica del divide et impera, il problema può essere risolto in tempo $O(n \log n)$.

Il problema della coppia più vicina

Idea intuitiva.

- l'insieme ha cardinalià costante: usiamo la ricerca esaustiva.
- altrimenti: lo dividiamo in due parti uguali S e D, per esempio quelli a sinistra e quelli a destra di una fissata linea verticale
  - troviamo ricorsivamente le soluzioni per l'istanza per S e quella per D individuando due coppie di punti a distanza minima, d_S e d_D
- soluzione finale: o una delle due coppie già individuate oppure può essere formata da un punto in S e uno in D
- se d_{SD} è la minima distanza tra punti aventi estremi in S e D, la soluzione finale è data dalla coppia di punti a distanza $\min\{d_{SD}, d_S, d_D\}$.

Il problema della coppia più vicina

Costo computazionale definito mediante la relazione di ricorrenza

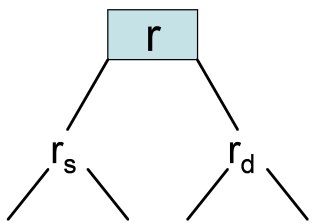
$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 2 \\ 2T\left(\frac{n}{2}\right) + cn & \text{altrimenti} \end{cases}$$

dove c_0, c sono costanti.

Per il teorema fondamentale, abbiamo $T(n) = O(n \log n)$.

Definizione ricorsiva di albero binario

1. Scegli uno degli elementi come **radice** r
2. Dividi i **rimanenti** elementi in **due gruppi**
3. Etichetta un gruppo come **sinistro** e l'altro come **destro**
4. Ricorsivamente organizza **ciascun gruppo** in un **sottoalbero** (potenzialmente vuoto)
5. r_s = radice del sottoalbero sinistro (`null` se vuoto)
   r_d = radice del sottoalbero destro (`null` se vuoto)
6. r è **padre** di r_s e r_d
   r_s è **figlio sinistro** di r
   r_d è **figlio destro** di r



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    }
```

Invocare con u uguale alla radice
[alvie]

Profondità di un nodo

- Radice ha profondità 0
- I suoi figli hanno profondità pari a 1, e così via
- Un nodo ha profondità pari a $p \Rightarrow$ i figli hanno profondità pari a $p + 1$

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

Algoritmi ricorsivi su alberi: altezza

Altezza = massima profondità raggiunta dalle foglie
Calcolo efficiente dell'altezza h:

- caso base: foglia ha $h = 0$ (ma non lo usiamo...)
- passo induttivo: $h = 1+$ massima altezza dei figli
- caso base per `null` $\Rightarrow h = -1$ (usiamo questo!)

```
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    }
```

Invocare con `u` uguale alla radice
[alvie]

Problema decomponibile (divide et impera su alberi)

Codici precedenti: stessa struttura!

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

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

Problemi decomponibili (divide et impera su alberi)

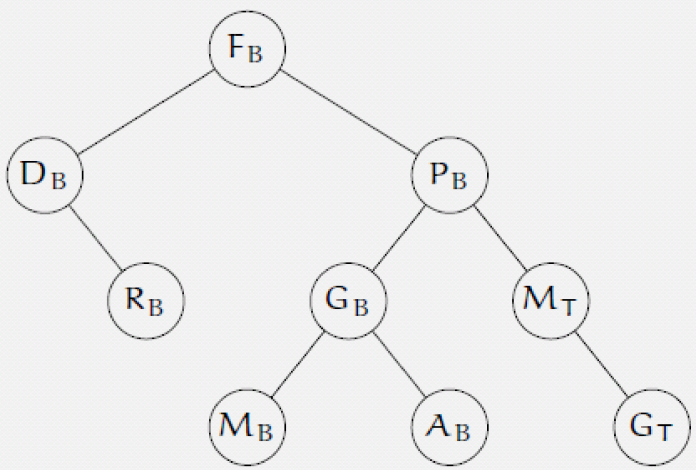
Analisi di complessità per un albero di n nodi:

> *se* `Ricombina` *richiede* R *tempo, allora* `Decomponibile` *richiede* $\mathbf{O(R \times n)}$ **tempo** *(nei nostri esempi,* $R = \text{costante}$*)*

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

Visite di alberi

Attraversamento sistematico di tutti i nodi (analogamente alla scansione di liste)



anticipata:

F_B D_B R_B P_B G_B M_B A_B M_T G_T

simmetrica:

D_B R_B F_B M_B G_B A_B P_B M_T G_T

posticipata:

R_B D_B M_B A_B G_B G_T M_T P_B F_B

ampiezza:

F_B D_B P_B R_B G_B M_T M_B A_B G_T

`visita(u)` = PROBLEMA DECOMPONIBILE

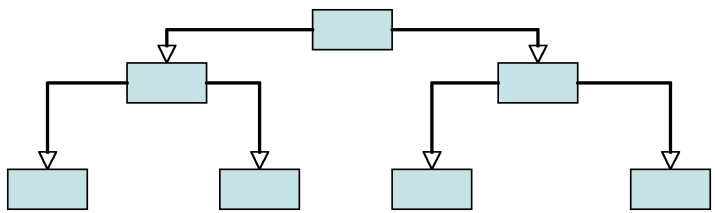
- **anticipata** (*preorder*):
  `elabora(u)`, `visita(u.sx)`, `visita(u.dx)`

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

  $O(n)$ tempo per n nodi
- **simmetrica** (*inorder*):
  `visita(u.sx)`, `elabora(u)`, `visita(u.dx)`
- **posticipata** (*postorder*):
  `visita(u.sx)`, `visita(u.dx)`, `elabora(u)`

Problema decomponibile: albero completamente bilanciato

- Albero binario **completo**: ogni nodo interno ha sempre due figli non vuoti
- Albero **completamente bilanciato**: completo e tutte le **foglie** hanno la **stessa profondità**
  Esempio:



- $T(u) =$ sottoalbero di T radicato in u

Problema decomponibile: bilanciato

- 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   }
```

[alvie]

Trasmissione dell'informazione tra chiamate ricorsive

- **postorder** $\Rightarrow$ dalle foglie alla radice
  la soluzione del problema per $T(\texttt{u})$ può utilizzare l'informazione raccolta in $T(\texttt{u.sx})$ e $T(\texttt{u.dx})$
- **passaggio dei parametri** $\Rightarrow$ dalla radice alle foglie
  la soluzione del problema per $T(\texttt{u})$ può utilizzare l'informazione raccolta dalla radice fino al nodo u

Esempio: calcolo della profondità rivisitato

```
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   }
```

Problema giocattolo: nodi cardine

- Passiamo informazione simultaneamente da foglie a radice e viceversa, combinano i due approcci
- Nodo u è cardine se e solo se $\text{profondita}(\mathtt{u}) = \text{altezza}(T(\mathtt{u}))$
- Calcoliamo simultaneamente la profondità e l'altezza

```
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   }
```

Attenzione: richiede più di $O(n)$ tempo calcolare la profondità o l'altezza con un'ulteriore chiamata ricorsiva in u!

Inserimento e cancellazione

- Cambiano $O(1)$ riferimenti tra i nodi
- Estensione delle omologhe operazioni descritte per le liste
- Vedremo l'uso specifico nel capitolo 5 (sui dizionari)