Estratto delle lezioni su Divide et impera

Annalisa De Bonis

IASD 2014-15

Relazioni di ricorrenza

- Quando un algoritmo contiene una o più chiamate ricorsive a sé stesso, il suo tempo di esecuzione può essere spesso descritto da una *relazione di ricorrenza*.
- Una relazione di ricorrenza consiste in un'uguaglianza o in una disuguaglianza che descrive una funzione in termini dei suoi valori su input più piccoli.
- Esempio:

$$f(n) = \begin{cases} a & \text{se } n \leq 2 \\ 2f(n/3) + 4n & \text{altrimenti} \end{cases}$$

Relazioni di ricorrenza

- Vediamo come si scrive la relazione di ricorrenza che descrive il tempo di esecuzione $T(n)$ di un algorimo basato sulla tecnica del divide et impera per un input di dimensione n.

- Se la dimensione n del problema è minore di una certa costante c, l'algoritmo risolve direttamente il problema (senza effettuare chiamate ricorsive)

$$T(n) \leq c_0, \text{ per una certa costante } c_0 \ .$$

- Per $n > c$, il problema viene suddiviso in sottoproblemi: supponiamo che il problema venga suddiviso in α sottoproblemi, ognuno di dimensione n/β
- L'algoritmo viene invocato ricorsivamente per risolvere ciascuno di questi α sottoproblemi
- Le α soluzioni per questi sottoproblemi vengono ricombinate per ottenere la soluzione al problema originario.

Relazioni di ricorrenza

- Supponiamo che l'algoritmo impieghi al più tempo $d(n)$ per suddividere il problema di partenza in α sottoproblemi.
- Supponiamo che l'algoritmo impieghi al più tempo tempo $r(n)$ per ricombinare le soluzioni degli α sottoproblemi.
- Il tempo di esecuzione $T(n)$ per $n > c$ può essere descritto dalla relazione:

$$T(n) \leq \alpha T(n/\beta) + d(n) + r(n)$$

- Quindi possiamo scrivere la relazione di ricorrenza:

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

Tempo di esecuzione di MergeSort

- L'algoritmo MergeSort decompone il problema in due sottoproblemi di dimensione $n/2$ ciascuno e impiega tempo costante per la decomposizione (deve semplicemente computare gli indici di inizio e fine dei segmenti da ordinare) e tempo lineare per ricombinare le soluzioni dei sue sottoproblemi (deve fondere i due segmenti ordinati).
- Nell'analisi per semplicità assumiamo che n sia una potenza di 2 in modo che ogni chiamata ricorsiva divida il segmento su cui opera in due segmenti di uguale grandezza.
- Quindi

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

Tempo di esecuzione di MergeSort

- Dimostriamo che il tempo di esecuzione è $O(\log n)$.
- Iteriamo la ricorrenza

$$\begin{aligned} T(n) &\leq cn + 2T(n/2) \leq cn + 2(cn/2 + 2T(n/4)) = 2cn + 4T(n/4) \\ &\leq 2cn + 4(cn/4 + 2T(n/8)) = 3cn + 8T(n/8) \\ &\leq \ldots \leq icn + 2^i T\left(\frac{n}{2^i}\right) \end{aligned}$$

- Quante volte dobbiamo iterare la ricorrenza prima di raggiungere il caso base?
- Ogni volta che applichiamo la ricorrenza la dimensione dell'input viene dimezzata per cui l'i-esima volta che applichiamo la ricorrenza l'argomento della funzione T diventa $\frac{n}{2^i}$. Raggiungiamo il caso base quando $\frac{n}{2^i} \leq 1$ e cioè non appena $2^i \geq n$. Ne consegue che ci fermiamo dopo che abbiamo applicato la ricorrenza $\log n$ volte.
- Dopo aver applicato la ricorrenza $\log n$ volte si ha

$$T(n) \leq cn \log n + 2^{\log n} T(1) \leq cn \log n + 2^{\log n} c_0 = cn \log n + c_0 n = O(n \log n).$$

Ricerca binaria: versione ricorsiva

```
1  RicercaBinariaRicorsiva( a,k,sinistra,destra ):
2    IF (sinistra > destra) {
3      RETURN -1;
4    }
5    c = (sinistra+destra)/2;
6    IF (k == a[c]) {
7      RETURN c;
8    }
9    IF (sinistra==destra) {
10     RETURN -1;
11   }
12   IF (k <a[c]) {
13     RETURN RicercaBinariaRicorsiva( a,k,sinistra,c-1 );
14   } ELSE {
15     RETURN RicercaBinariaRicorsiva( a,k,c+1,destra );
16   }
```

Paradigma divide et impera

1. **Caso base**: Il segmento in cui stiamo effettuando la ricerca contiene al più un elemento oppure abbiamo trovato l'elemento al centro del segmento (righe 2–10)
2. **Decomposizione**: per decomporre occorre vedere se k è minore o maggiore di $a[c]$ (riga 11)
3. **Ricorsione e ricombinazione**: di fatto non occorre nessun lavoro di ricombinazione (righe 12–16)

Analisi mediante relazione di ricorrenza

- Se il segmento all'interno del quale stiamo cercando contiene al più un elemento oppure l'elemento cercato è quello centrale, allora l'algoritmo esegue un numero costante di operazioni $\leq c_0$.
- Altrimenti, il tempo richiesto è pari a una costante c più il tempo richiesto dalla ricerca dell'elemento in un segmento di dimensione pari alla metà di quello attuale.
  Il tempo totale di esecuzione $T(n)$ su un array di n elementi verifica la relazione di ricorrenza:

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

- Applicando iterativamente la ricorrenza si ha

$$T(n) \leq T(n/2) + c \leq T(n/4) + c + c \leq \ldots \leq T(\frac{n}{2^i}) + ci$$

- Per $i = \log n$ abbiamo

$$T(n) \leq T(1) + c \log n = c_o + c \log n = O(\log n).$$

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

Paradigma della ricerca binaria

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

1. Chiedi se il numero intero x è $\leq 2^i$ per $i = 1, 2, \ldots$
2. Fermati non appena la risposta è *sì*.
3. Sia h l'indice in corrispondenza del quale otteniamo *sì* come risposta. Ovviamente si ha che $2^{h-1} < x \leq 2^h$ e di conseguenza $\log x \leq h < \log x + 1$
4. Effettua ricerca binaria nell'intervallo $[2^{h-1} + 1, 2^h]$
5. Intervallo contiene 2^{h-1} interi per cui ricerca binaria richiede tempo $O(\log 2^{h-1}) = O(h - 1) = O(\log x)$

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 (l'elemento cercato appare in una delle n posizioni dell'insieme oppure non appare nell'insieme)

- Assumiamo che l'elemento cercato non sia nell'insieme (il caso pessimo richiederà **almeno** lo stesso numero di confronti)
- A esegue dei confronti, ognuno dei quali dà luogo a due possibili risposte in $[<, >]$ (dal momento che l'elemento cercato non è nella sequenza, la risposta $=$ non viene mai data).
- Se A effettua t confronti ci sono 2^t possibili sequenze di risposte.
- Dopo t confronti di elementi, l'algoritmo A può discernere al più 2^t situazioni.
- Poiché le situazioni da discernere sono $n+1$, deve valere $2^t \geq n+1$.
- Ne deriva che occorrono $t \geq \log_2(n+1) = \Omega(\log n)$ 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: non occorre fare alcun lavoro.

```
1  QuickSort( a, sinistra, destra ):
2
3    IF (sinistra < destra) {
4      scegli pivot nell'intervallo [sinistra...destra];
5      indiceFinalePivot = Distribuzione( a, sinistra, pivot, destra
       );
6      QuickSort( a, sinistra, indiceFinalePivot-1 );
7      QuickSort( a, indiceFinalePivot+1, destra );
8    }
```

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 $\texttt{px} \neq \texttt{dx}$
  - usa due indici `i` e `j` per scandire il segmento: `i` parte da `sx` e va verso destra e `j` parte da $\texttt{dx} - 1$ e va verso sinistra fino a quando $\texttt{i} \leq \texttt{j}$
  - ogni volta che si ha $a[\texttt{i}] > \texttt{pivot}$ e $a[\texttt{j}] < \texttt{pivot}$, scambia $a[\texttt{i}]$ con $a[\texttt{j}]$ e poi riprende la scansione
  - alla fine della scansione posiziona il pivot nella sua posizione corretta

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;
```

```
1  Scambia( i, j ):                                    ⟨pre: sx ≤ i, j ≤ dx⟩
2    temp = a[j]; a[j] = a[i]; a[i] = temp;
```

Analisi di Distribuzione

- (numero di volte che incrementiamo `i`)+ (numero di volte che decrementiamo `j`)=$n-1$
- ogni confronto di a[`i`] con il pivot porta a incrementare `i` (o prima o dopo aver effettuato lo scambio)
- ogni confronto di a[`j`] con il pivot porta a decrementare `j` (o prima o dopo aver effettuato lo scambio)
- numero di confronti con il pivot è quindi al più $n-1$ (in realtà si può vedere che è proprio $n-1$)
- tempo $O(n)$

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'
  - Applichiamo iterativamente la relazione di ricorrenza:

$$T(n) \leq T(n-1)+c'n \leq T(n-2)+c'(n-1)+c'n \leq \ldots \leq T(n-i)+\sum_{j=0}^{i-1} c'(n-j).$$

  - Sostitutendo $i = n-1$ nell'espressione più a destra, otteniamo

$$T(n) \leq T(1)+\sum_{j=0}^{n-2} c'(n-j) = c_0+\sum_{j=2}^{n} c'j \leq c_0+c'(n+1)n/2-c' = O(n^2),$$

- caso ottimo:
  - la distribuzione è bilanciata ($r = n/2$)), la ricorsione avviene su ciascuna metà
  - in questa situazione, il costo è simile a quella dell'ordinamento per fusione.
  - possiamo dimostrare che il costo è di $O(n \log n)$ tempo
- caso medio: $O(n \log n)$ passi. L'algoritmo 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
- NB: Il problema diventa quello di 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    IF (sinistra == destra) {
3      RETURN a[sinistra];
4    } ELSE {
5      scegli pivot nell'intervallo [sinistra...destra];
6      indiceFinalePivot = Distribuzione( a, sinistra, pivot, destra
       );
7      IF (r-1 == indiceFinalePivot) {
8        RETURN a[indiceFinalePivot];
9      } ELSE IF (r-1 < indiceFinalePivot) {
10       RETURN QuickSelect( a, sinistra, r, indiceFinalePivot-1 );
11     } ELSE {
12       RETURN QuickSelect( a, indiceFinalePivot+1, r, destra );
13     }
14   }
```

Analisi di QuickSelect mediante relazione di ricorrenza

- Caso base:
  Se il segmento sul quale opera l'algoritmo contiene un solo elemento allora l'algoritmo esegue un numero costante di operazioni $\leq c_0$;
  Se nel codice l'indice restituito da *Distribuzione*(a, *sinistra*, *pivot*, *destra*) è uguale a $r-1$ (in altre parole se il pivot ha rango r), l'algoritmo termina e il suo costo è dominato dal costo di Distribuzione che esegue $\leq c_1 n$ operazioni, dove c_1 è una certa costante.
- Passo ricorsivo: il numero di operazioni eseguite è al più pari a cn (c costante) più il numero di operazioni richieste per effettuare la selezione nel segmento degli elementi minori o uguali del pivot **oppure** in quello degli elementi maggiori o uguali del pivot.

Analisi di QuickSelect mediante relazione di ricorrenza

Relazione di ricorrenza per il tempo $T(n)$ di esecuzione dell'algoritmo.
Indichiamo con r_p il rango del pivot (si ha quindi $r_p = \mathit{indiceFinalePivot} + 1$)

- Caso base:
  $T(n) \leq c_0$ per $n \leq 1$ e
  $T(n) \leq c_1 n$ se $r_p = r$.
  In entrambi i casi $T(n) \leq c_1 n$.
- Passo ricorsivo: Ci sono $r_p - 1$ elementi a sinistra del pivot e $n - r_p$ elementi a destra, per cui $T(n) \leq \max\{T(r_p - 1), T(n - r_p)\} + cn$.

$$T(n) \leq \begin{cases} c_1 n & \text{se } n \leq 1 \text{ o } r_p = r - 1 \\ \max\{T(r_p - 1), T(n - r_p)\} + cn & \text{altrimenti} \end{cases}$$

Analisi di QuickSelect mediante relazione di ricorrenza

- caso pessimo:
  - Il pivot è tutto a sinistra ($r_p = 1$) e $r > r_p$ oppure tutto a destra ($r_p = n$) e $r < r_p$. In entrambi i casi, la relazione diventa $T(n) \leq T(n-1) + cn$.
  - Applichiamo iterativamente la relazione di ricorrenza:

$$T(n) \leq T(n-1)+cn \leq T(n-2)+c(n-1)+cn \leq \ldots \leq T(n-i)+\sum_{j=n-i+1}^{n} c\, i.$$

  - Sositutendo $i = n-1$ nell'ultima disequazione, otteniamo

$$T(n) \leq T(1) + \sum_{j=2}^{n} c\, i \leq c_0 + \sum_{j=2}^{n} c\, i = c_0 + c\, n(n+1)/2 - c = O(n^2).$$

- caso ottimo:
  - l'elemento di rango r è proprio il pivot ($r_p = r$), per cui si esce dalla procedura senza effettuare la ricorsione e si ha che $T(n) = O(n)$.
  - il caso ottimo si verifica anche quando ad ogni chiamata ricorsiva viene dimezzata la lunghezza del segmento in cui effettuare la selezione.

$$T(n) \leq T(n/2) + cn \leq T(n/4) + c(n/2) + cn \leq \ldots \leq T(\frac{n}{2^i}) + \sum_{j=0}^{i-1} c\frac{n}{2^j}.$$

  Dopo $\log n$ applicazioni della relazione di ricorrenza otteniamo

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{2^{\log n}}\right) + \sum_{j=0}^{\log n-1} c\frac{n}{2^j} = T(1) + cn \sum_{j=0}^{\log n-1} \frac{1}{2^j} \\ &\leq c_0 + cn\left(\frac{1-1/2^{\log n}}{1/2}\right) = c_0 + 2cn(1-1/n) = O(n) \end{aligned}$$

- caso medio: tempo $O(n)$. L'algoritmo è veloce in pratica.

Moltiplicazione di interi

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

```
      2345 x
      5382 =

      ____
      4690
    18760
    7035
  11725

_____________
  12620790
```

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

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

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 (ad un costo di $T(n/2)$), 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

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

Moltiplicazione veloce di interi

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

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

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

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

Moltiplicazione veloce di interi

- È possibile progettare un algoritmo più veloce?
- 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

```
1  MoltiplicazioneVeloce( x, y, n ):
2    IF (n == 1) {
3      prodotto[1] = (x[1] × y[1]) / 10;  prodotto[2] = (x[1] ×
       y[1]) % 10;
4    } ELSE {
5      xs[0] = xd[0] = ys[0] = yd[0] = 1;
6      FOR (i = 1; i <= n/2; i = i + 1)
7        {xs[i] = x[i]; ys[i] = y[i]; xd[i] = x[i + n/2]; yd[i] = y[i
         + n/2];}
8      p1 = MoltiplicazioneVeloce( xs, ys, n/2 );   //Computa x_s · y_s
9      //Computa 10^n x_s y_s e lo memorizza in prodotto
10     FOR (i = 0; i <= n; i = i+1)
11       { prodotto[i] = p1[i]; prodotto[i+n] = 0; }
12     p2 = MoltiplicazioneVeloce( xd, yd, n/2 );   //Computa x_d · y_d
13     xd[0] = yd[0] = -1;
14     //Computa (x_s − x_d) · (y_s − y_d)
15     p3 = MoltiplicazioneVeloce( Somma(xs,xd), Somma(ys,yd), n/2);
16     p3[0] = -p3[0];
17     //Computa x_s y_s + x_d y_d − (x_s − x_d) · (y_s − y_d) = x_s y_d + x_d y_s
18     add = Somma( p1, p2, p3 );
19     parziale[0] = add[0];
20     //Computa 10^{n/2}(x_s y_d + x_d y_s) e mette il risultato in parziale
21     FOR (i = 1; i <= 3 × n/2; i = i+1)
22       { parziale[i] = add[i + n/2]; parziale[i + 3 × n/2] = 0; }
23     //Computa 10^n x_s y_s+10^{n/2}(x_s y_d+x_d y_s)+x_d y_d e lo memorizza in prodotto
24     prodotto = Somma( prodotto, parziale, p2 );
25   }
26   prodotto[0] = x[0] × y[0];
27   RETURN prodotto;
```

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:

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

Continua nella prossima slide

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

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

Soluzione Esercizio 3.1

Per semplificare l'analisi, l'algoritmo effettua la ricorsione su uno o due sotto-segmenti del segmento iniziale, ciascuno dei quali include l'elemento centrale. Ovviamente è possibile modificare l'algoritmo in modo da evitare di includere l'elemento centrale nei due sotto-segmenti.

1. *Conta*(a, s, d, k) conta le occorrenze di k tra gli indici s e d di a.
2. Se $s = d$ allora *Conta* restituisce 1 se $a[s] = k$, e 0 altrimenti.
3. se k è più piccolo dell'elemento centrale, richiama procedura sulla prima metà;
4. se k è più grande dell'elemento centrale, richiama la procedura sulla seconda metà;
5. se k è uguale all'elemento centrale
   - se $k = a[s]$ allora *Conta* restituisce $\lfloor (d - s)/2 \rfloor +$ valore restituito da chiamata su seconda metà (compreso elemento centrale);
   - se $k = a[d]$ allora *Conta* restituisce $\lceil (d - s)/2 \lceil +$ valore restituito da chiamata su prima metà (compreso elemento centrale) ;
   - se k è diverso da $a[s]$ e da $a[d]$, restituisce la somma dei valori restituiti dalle 2 chiamate ricorsive sulle due metà compreso l'elemento centrale -1.

Soluzione Esercizio 3.1

Calcoliamo il tempo di esecuzione.

1. se *Conta* viene invocata con un indice s tale che $a[s] = k$, allora anche le chiamate ricorsive più interne di *Conta* verranno invocate con un valore di s per cui $a[s] = k$;
2. se *Conta* viene invocata con indice di d tale che $a[s] = d$, allora anche le chiamate ricorsive più interne di *Conta* verranno invocate con un valore di s per cui $a[d] = k$;
3. La funzione *Conta* effettua due chiamate ricorsive solo se $a[\lfloor s + d \rfloor)/2] = k$, $k \neq a[s]$ e $k \neq a[d]$ e in questo caso entrambe le chiamate avvengono su segmenti che hanno uno degli estremi uguale a k (l'ultimo estremo nel segmento di sinistra e il primo estremo in quello di destra).
4. I punti 1, 2 e 3 implicano che una volta che uno degli estremi del segmento su cui si invoca *Conta* è uguale k, allora anche tutte le chiamate più interne effettuano al più un'unica chiamata ricorsiva.
5. Nel caso pessimo le due chiamate ricorsive vengono invocate nella prima chiamata ricorsiva (effettuata su tutto l'array)
6. di conseguenza, la funzione che descrive il tempo di esecuzione nel caso pessimo soddisfa la relazione:

continuna nella prossima slide

Soluzione Esercizio 3.1

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \\ 2T'(n/2) + c & \text{altrimenti} \end{cases},$$

dove $T'(n)$ soddisfa 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 relazione soddisfatta da $T'(n)$ è la stessa relazione soddisfatta dalla funzione che descrive il tempo di esecuzione della ricerca binaria per cui $T'(n) = O(\log n)$ e $T(n) \leq 2T'(n/2) + c = O(\log n)$.

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 delle ricorrenze

Dimostrazione. Assumiamo che n sia potenza di β ($n = \beta^k$, per qualche k) e che $n_0 = 1$ e applichiamo iterativamente la relazione di ricorrenza.

$$\begin{aligned}
T(n) &\leq cf(n) + \alpha T(n/\beta) \leq cf(n) + \alpha(cf(n/\beta) + \alpha T(n/\beta^2)) \\
&= cf(n) + \alpha cf(n/\beta) + \alpha^2 T(n/\beta^2) \\
&\leq cf(n) + \alpha cf(n/\beta) + \alpha^2(cf(n/\beta^2) + \alpha T(n/\beta^3)) \\
&= cf(n) + \alpha cf(n/\beta) + \alpha^2 cf(n/\beta^2) + \alpha^3 T(n/\beta^3) \\
&\leq \dots \\
&\leq cf(n) + \alpha cf(n/\beta) + \alpha^2(cf(n/\beta^2) + \dots + \alpha^{i-1}(cf(n/\beta^{i-1}) + \alpha^i T(n/\beta^i))
\end{aligned}$$

Per $i = \log_\beta n$ si ha

$$\begin{aligned}
T(n) &\leq cf(n) + \alpha cf(n/\beta)) + \dots + \alpha^{(\log_\beta n - 1)}(cf(n/\beta^{(\log_\beta n - 1)}) + \alpha^{\log_\beta n} T(n/\beta^{\log_\beta n})) \\
&= \sum_{j=0}^{\log_\beta n - 1} \alpha^j (cf(n/\beta^j) + \alpha^{\log_\beta n} T(1) \leq \sum_{j=0}^{\log_\beta n - 1} \alpha^j (cf(n/\beta^j) + \alpha^{\log_\beta n} c_0.
\end{aligned}$$

L'ipotesi $\alpha\, f(n/\beta) = \gamma\, f(n)$ implica

$$\alpha^i\, f(n/\beta^i) = \gamma^i\, f(n) \tag{1}$$

(quest'ultima uguaglianza è dimostrata per induzione alla fine della dimostrazione del Teorema).

L'ultima disuguaglianza nella slide precedente e la (1) implicano che

$$T(n) \leq c \sum_{j=0}^{\log_\beta n - 1} \alpha^j (f(n/\beta^j) + \alpha^{\log_\beta n} c_0 = c \sum_{j=0}^{\log_\beta n - 1} \gamma^j\, f(n) + \alpha^{\log_\beta n} c_0$$

Nota che per $i = \log_\beta n$ si ha $\alpha^{\log_\beta n}\, f(1) = \alpha^{\log_\beta n}\, f(n/\beta^{\log_\beta n}) = \gamma^{\log_\beta n}\, f(n)$ e quindi $\alpha^{\log_\beta n} = \frac{\gamma^{\log_\beta n}\, f(n)}{f(1)} = O(\gamma^{\log_\beta n}\, f(n))$.
Si ha quindi

$$\begin{aligned} T(n) &\leq c \sum_{j=0}^{\log_\beta n - 1} \gamma^j\, f(n) + \alpha^{\log_\beta n} c_0 = c \sum_{j=0}^{\log_\beta n - 1} \gamma^j\, f(n) + O(\gamma^{\log_\beta n})\, f(n) \\ &= O\left(\sum_{j=0}^{\log_\beta n} \gamma^j\, f(n) \right) \end{aligned} \tag{2}$$

Esaminiamo i tre casi:

1. Caso 1: $\gamma < 1$

$$\sum_{j=0}^{\log_\beta n} \gamma^j = \frac{\gamma^{\log_\beta n+1} - 1}{\gamma - 1} = \frac{1 - \gamma^{\log_\beta n+1}}{1 - \gamma} < \frac{1}{1 - \gamma} = O(1)$$

   per cui dalla (2) si ha che

$$T(n) = O(f(n))$$

2. Caso 2: $\gamma = 1$

$$\sum_{j=0}^{\log_\beta n} \gamma^j = 1 + \log_\beta n$$

   per cui dalla (2) si ha che

$$T(n) = O(f(n) \log_\beta n)).$$

1. Caso 3: $\gamma > 1$.

$$\sum_{j=0}^{\log_\beta n} \gamma^j = \frac{\gamma^{\log_\beta n+1} - 1}{\gamma - 1}$$

Si noti che $\frac{\gamma^{\log_\beta n+1}-1}{\gamma-1} < \frac{\gamma^{\log_\beta n+1}}{\gamma-1} = \frac{\gamma}{\gamma-1}\gamma^{\log_\beta n+1} = O(\gamma^{\log_\beta n})$, dal momento che $\frac{\gamma}{\gamma-1}$ è una costante positiva.
Si ha quindi

$$\sum_{j=0}^{\log_\beta n} \gamma^j = \frac{\gamma^{\log_\beta n+1} - 1}{\gamma - 1} = O(\gamma^{\log_\beta n})$$

per cui dalla (2) si ha che

$$T(n) = O(\gamma^{\log_\beta n} f(n)).$$

Inoltre dalla (1) si ha che

$$\gamma^{\log_\beta n} f(n) = \alpha^{\log_\beta n} f(n/\beta^{\log_\beta n}) = \alpha^{\log_\beta n} f(1)$$

e quindi

$$T(n) \quad = \quad O(\alpha^{\log_\beta n} f(1)) = O(\alpha^{\log_\beta n}) = O(\alpha^{\log_\alpha n \log_\beta \alpha}) = O(n^{\log_\beta \alpha}).$$

Dimostrazione dell'ugualglianza $\alpha^i f(n/\beta^i) = \gamma^i f(n)$

- Base dell'induzione $0 \leq i \leq 1$: per $i = 0$ l'uguaglianza vale banalmente; per $i = 1$ discende immediatamente dall'ipotesi $\alpha f(n/\beta) = \gamma f(n)$.
- Passo Induttivo: assumiamo che l'uguaglianza valga per i, con $i \geq 1$, e dimostriamo che vale per $i + 1$:

  $$\alpha^{i+1} f(n/\beta^{i+1}) = \alpha^i \alpha f((n/\beta^i)/\beta).$$

  Applicando l'ipotesi del teorema $\alpha f(n/\beta) = \gamma f(n)$ con n/β^i al posto di n si ha $\alpha f((n/\beta^i)/\beta) = \gamma f(n/\beta^i)$, da cui si ha

  $$\alpha^{i+1} f(n/\beta^{i+1}) = \alpha^i \alpha f((n/\beta^i)/\beta) = \alpha^i \gamma f(n/\beta^i)$$

e applicando l'ipotesi induttiva si ha $\alpha^i f(n/\beta^i) = \gamma^i f(n)$ per cui $\alpha^i \gamma f(n/\beta^i) = \gamma \gamma^i f(n) = \gamma^{i+1} f(n)$.

Teorema fondamentale: esempi

- Ricerca binaria

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

Caso $\gamma = 1$ ($\alpha = 1, \beta = 2$, $f(n) = 1$) per cui
$T(n) = O(f(n) \log_\beta n)) = O(\log n)$

Teorema fondamentale: esempi

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 $O(n \log_2 n) = O(n \log n)$.

Teorema fondamentale: esempi

- 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 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 $O(n^{\log 4}) = O(n^2)$

- 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 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 $O(n^{\log_2 3}) = O(n^{1,585})$

Il problema della coppia più vicina

Problema: vogliamo trovare la coppia di punti più vicina tra un insieme di punti del piano.
La distanza tra due punti $p_1 = (x_1, y_1)$ e $p_2 = (x_2, y_2)$ si calcola con la formula $\sqrt{((x_1 - x_2)^2 + (y_1 - y_2)^2}$ in tempo $O(1)$

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

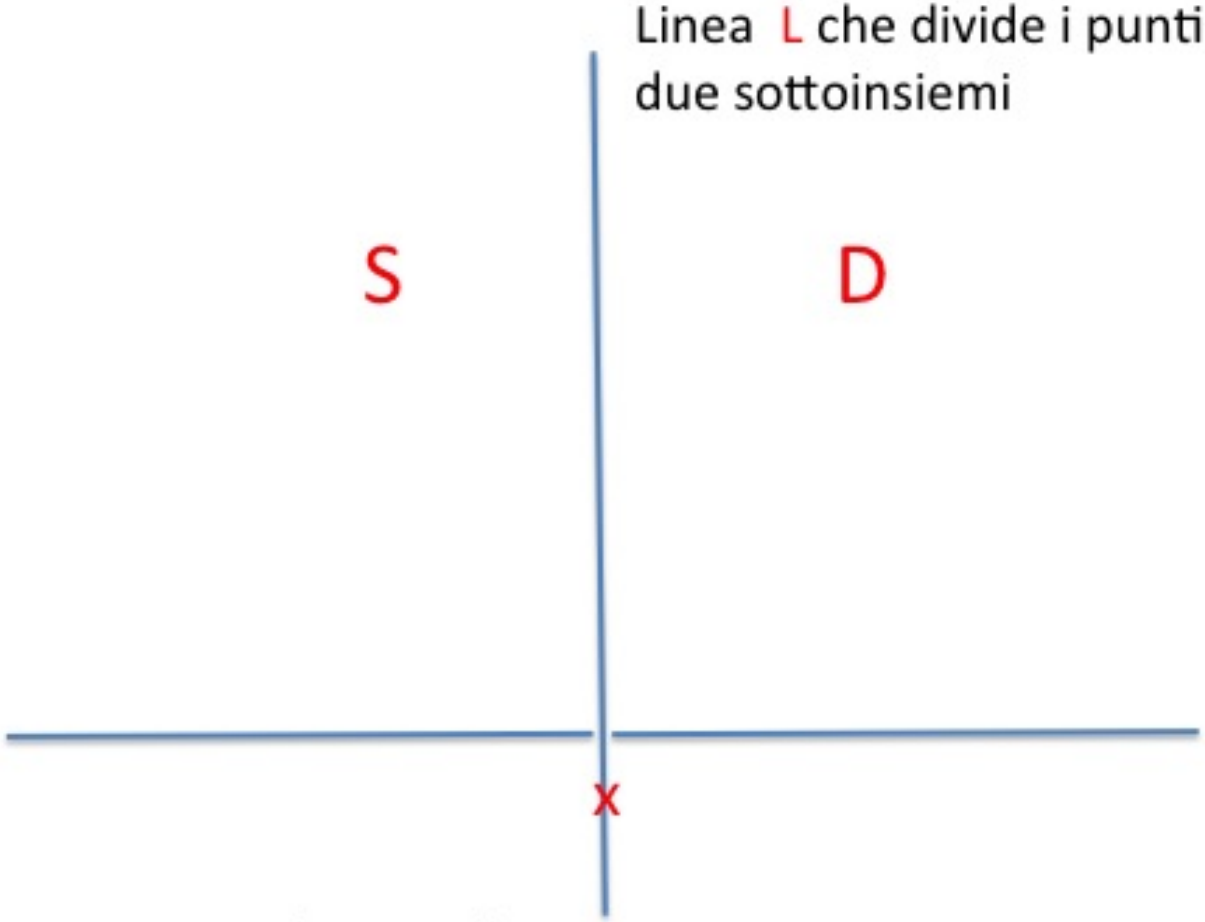
Utilizzando la tecnica del divide et impera, il problema può essere risolto in tempo $O(n \log n)$.

Il problema della coppia più vicina

Idea intuitiva.

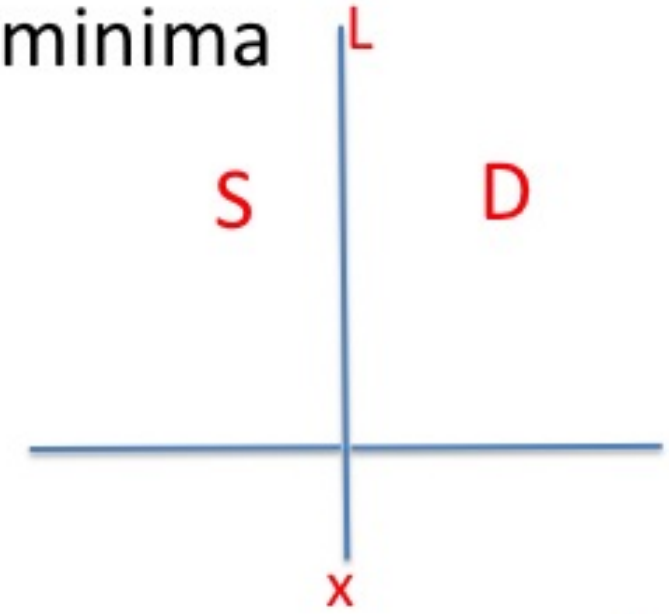
- l'insieme ha cardinalià costante: usiamo la ricerca esaustiva.
- altrimenti: lo dividiamo in due parti uguali S e D, per esempio quelli a sinistra e quelli a destra di una fissata linea verticale
  - troviamo ricorsivamente le soluzioni per l'istanza per S e quella per D individuando due coppie di 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 $\min\{d_{SD}, d_S, d_D\}$.

Partizione in due sottoinsiemi di n/2 punti ciascuno



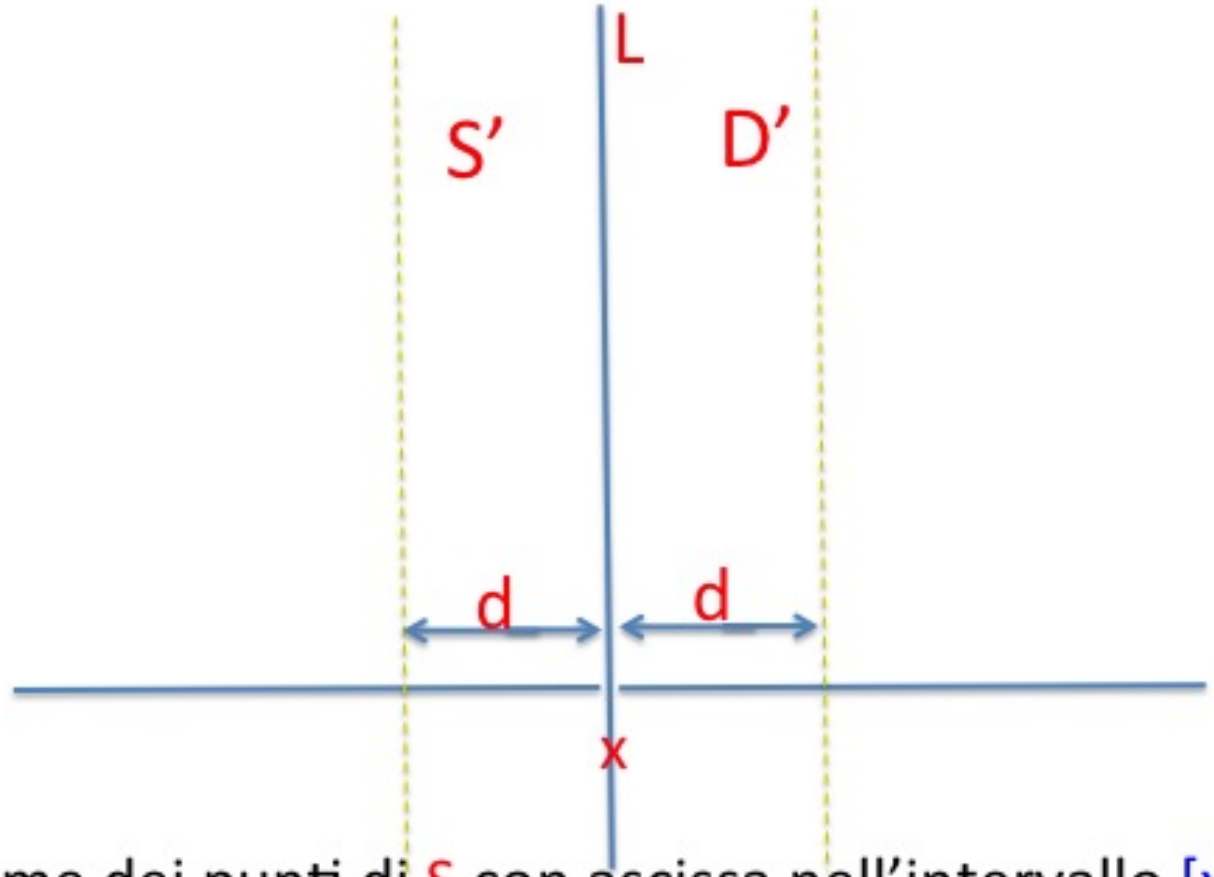
- Ordina i punti in base alle ascisse
- x= ascissa punto **pc** centrale nell'ordinamento
- S= insieme dei punti a sinistra di **pc** nell'ordinamento
- D= insieme dei punti a destra di **pc** nell'ordinamento

Individuazione della coppia di punti a distanza minima



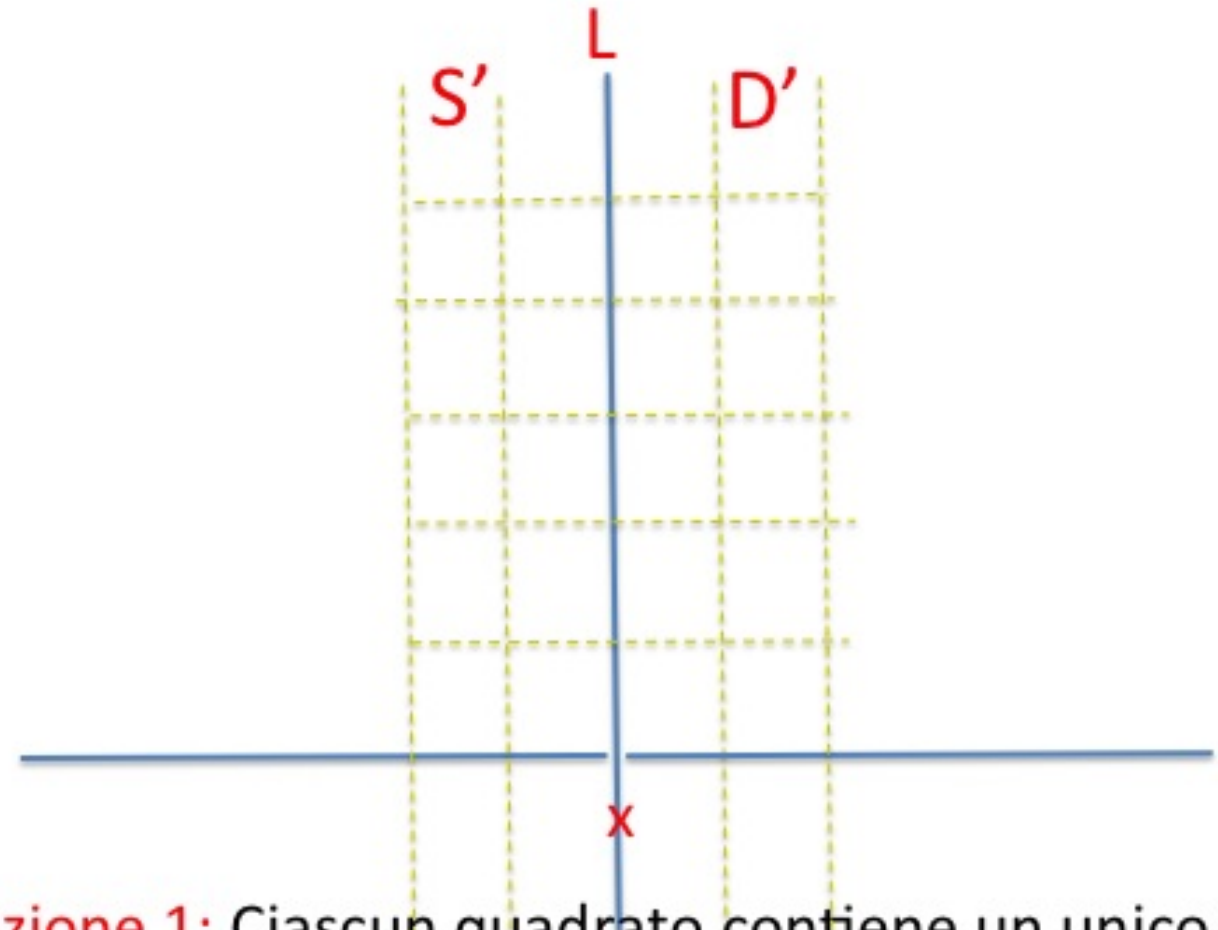
- I 2 punti a distanza minima o sono entrambi in S, o sono entrambi in D, o uno dei due si trova in S e l'altro in D
- Divide et impera:
- Decomposizione: partiziona l'insieme di punti in S e D
- Soluzione sottoproblemi: cerca la coppia a distanza minima d_S in S e la coppia a distanza minima d_D in D. $d=\min\{d_S, d_D\}$
- Ricombinazione: Cerca tra le coppie (p,q) con p in S e q in D quella a distanza minima d_{SD} e resituisce $\min\{d, d_{SD}\}$

Ricerca della coppia (p,q) a distanza minima con p in S e q in D



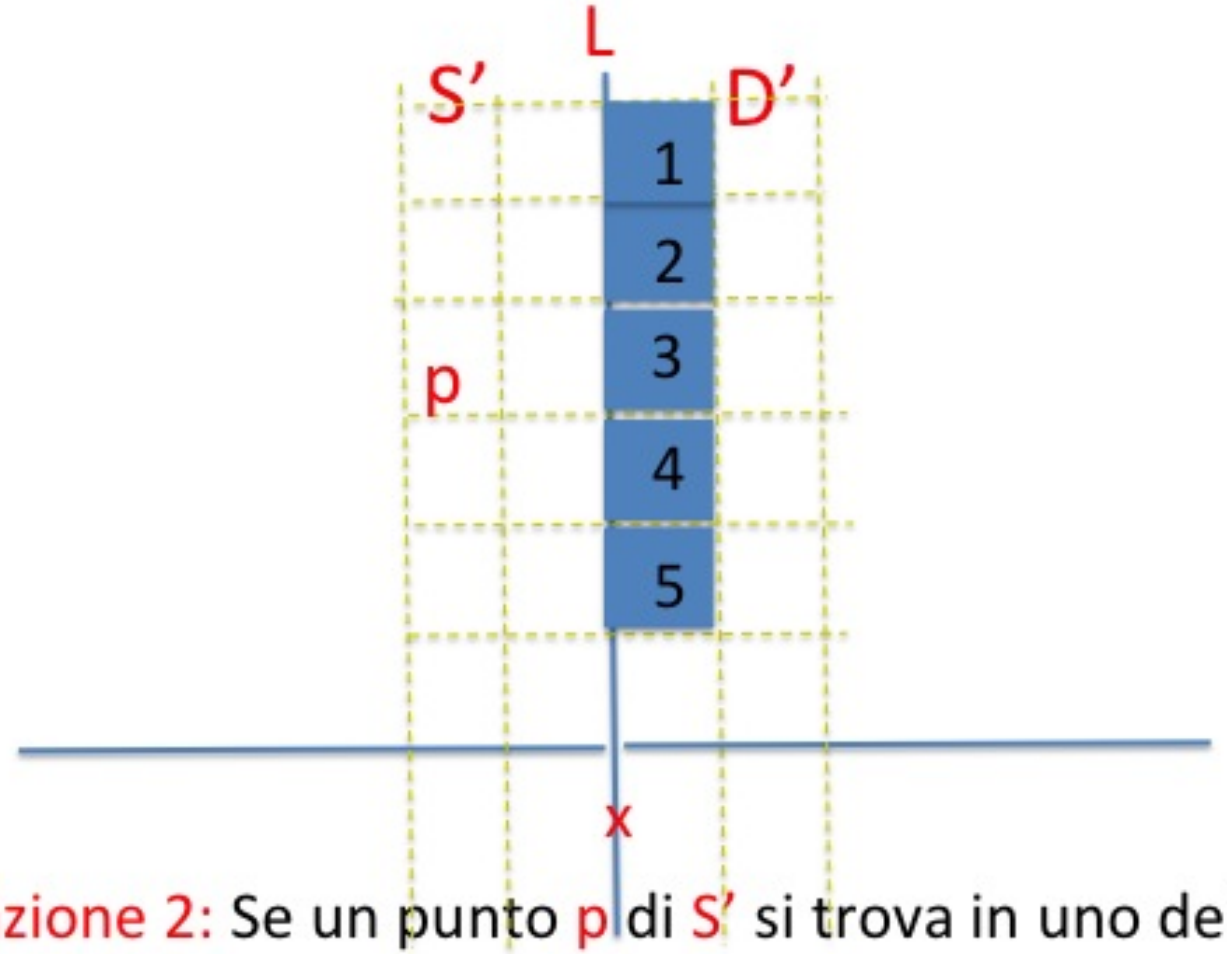
- S'= insieme dei punti di S con ascissa nell'intervallo [x-d,x]
- D'= insieme dei punti di D con ascissa nell'intervallo [x,x+d]
- E` sufficiente considerare coppie (p,q) con p in S' e q in D' ' in quanto le altre coppie (p,q) con p in S e q in D sono a distanza maggiore di d

Dividiamo S’ e D’ in tanti quadrati di lato uguale a d/2



- Osservazione 1: Ciascun quadrato contiene un unico punto altrimenti esisterebbe una coppia di punti entrambi in S’ o entrambi in D’, a distanza minore di d

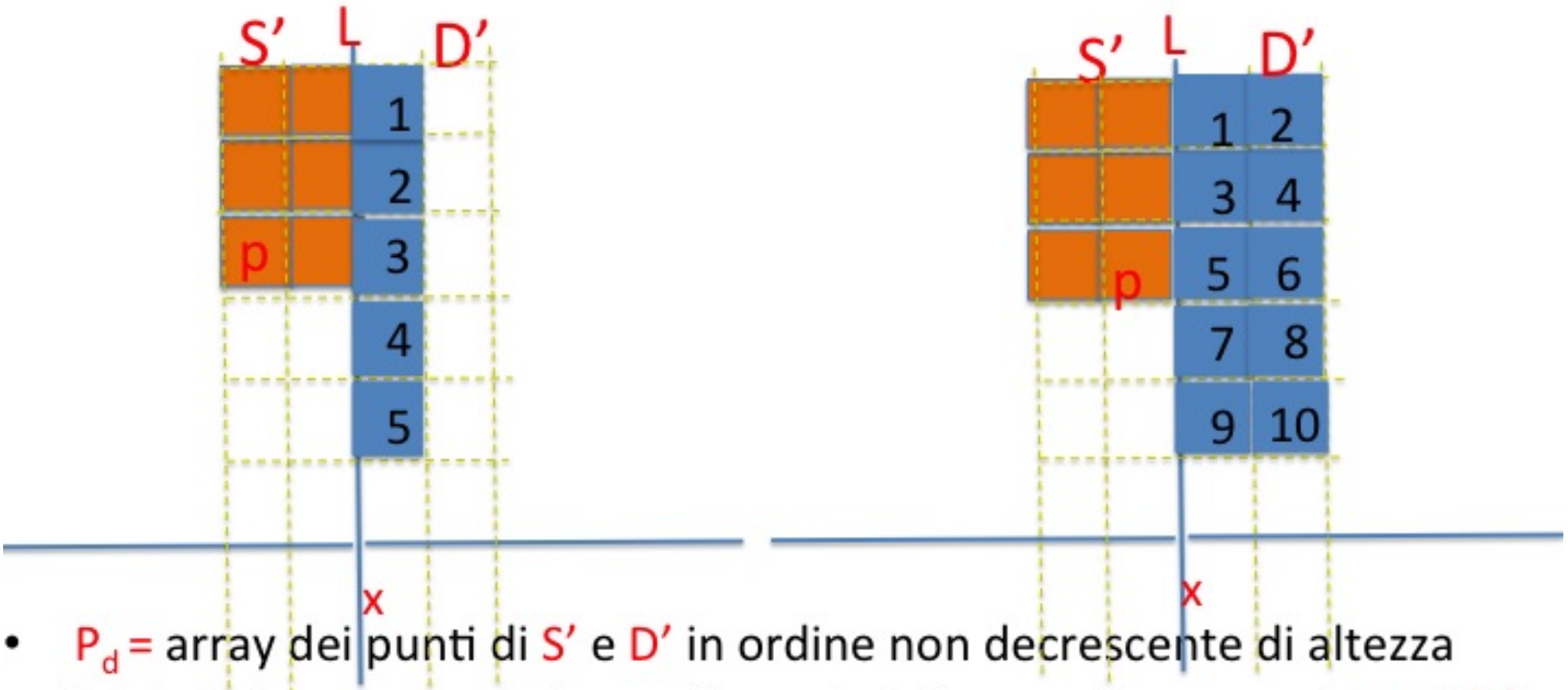
Dividiamo S’ e D’ in una griglia di quadrati di lato uguale a d/2



- Osservazione 2: Se un punto p di S’ si trova in uno dei quadrati piu` a sinistra allora un punto q di D’ a distanza minore di d da p puo` trovarsi solo in uno dei quadrati in D’ colorati di azzurro
  - Se q e` piu` in alto rispetto a p allora q si trova in uno dei quadrati 1, 2, 3; altrimenti si trova in uno dei quadrati 3, 4, 5.

Dividiamo S’ e D’ in tanti quadrati di lato uguale a d/2

- Osservazione 3: Se un punto p di S’ si trova in uno dei quadrati confinati con L allora un punto q di D’ a distanza minore di d da p puo` trovarsi solo in uno dei quadrati in D’ colorati di azzurro
  - Se q e` piu` in alto rispetto a p allora q si trova in uno dei quadrati 1-6; altrimenti q si trova in uno dei quadrati 5-10



- P_d = array dei punti di S' e D' in ordine non decrescente di altezza
- Poiche` ciascun quadrato contiene al piu` un punto, un punto q di D' a distanza al piu` d da p si trova a distanza al piu` 10 da p in P_d
  - tra p e q possono esserci al piu` 5 punti di D' e 5 punti di S' (II figura ): ad esempio se q e` piu` in alto rispetto a q allora tra p e q puo` esserci al piu` un punto di D' per ciascuno dei quadrati 1-6 (meno quello contente q) e un punto di S' per ciascuno dei quadrati arancioni (meno quello contente p)

L'algoritmo che trova la coppia più vicina (Per lo pseudocodice si veda il libro.)

Input: P_x = array dei punti ordinato in modo non decrescente rispetto alle ascisse; P_y = array dei punti ordinato in modo non decrescente rispetto alle ordinate, n dimensione degli array P_x e P_y

1. Se $n <= 3$, calcola le distanze tra le tre coppie di punti per trovare la coppia a distanza minima.
2. Se $n > 3$, esegue i seguenti passi:
3. Inserisce nell'array S_x i primi $\lfloor n/2 \rfloor$ punti di P_x e nell'array D_x gli ultimi $\lceil n/2 \rceil$ punti di P_x
4. Inserisce nell'array S_y i primi $\lfloor n/2 \rfloor$ punti di P_x nell'ordine in cui appaiono in P_y e nell'array D_y gli ultimi $\lceil n/2 \rceil$ punti di P_x nell'ordine in cui appaiono in P_y
5. Effettua una chiamata ricorsiva con input S_x, S_y e $\lfloor n/2 \rfloor$ e una chiamata ricorsiva con input D_x, D_y e $\lceil n/2 \rceil$. Siano d_S e d_D i valori delle distanze delle coppie di punti restituite dalla prima e dalla seconda chiamata rispettivamente. Pone $d = \min\{d_S, d_D\}$ e (p, q) uguale alla coppia a distanza d.
6. Copia in P_d i punti a distanza minore di d dalla retta verticale passante per l'elemento centrale di P_x nello stesso ordine in cui appaiono in P_y
7. Per ciascun punto p' in P_d esamina gli 11 punti che seguono p' in P_d; per ciascun punto q' (tra questi 11) computa la sua distanza da p' e se questa risulta minore di d, aggiorna il valore di d e pone $(p, q) = (p', q')$
8. Restituisce la coppia (p, q)

Analisi del costo dell'algoritmo che trova coppia più vicina

Assumiamo per semplicità che n sia un potenza di 2

1. Se n è $<= 3$, il costo è limitato superiormente da una certa costante c_0
2. Se n è > 3, il costo dell'algoritmo è così computato:
3. il costo del passo 3 è $O(n)$
4. il costo del passo 4 è $O(n)$: i punti di P_y vengono scanditi a partire dalla prima locazione.e vengono man mano inseriti in S_y o in D_y a seconda che si trovino in locazioni di P_x di indice minore di $\lfloor n/2 \rfloor$ oppure in locazioni di P_x di indice maggiore o uguale di $\lfloor n/2 \rfloor$
5. Il costo delle due chiamate ricorsive è $2T(n/2)$; il costo delle altre operazioni eseguite al passo 5 è costante
6. Il passo 6 richiede tempo $O(n)$: i punti di P_y vengono scanditi a partire dalla prima locazione e quelli la cui ascissa differisce al più d dall'ascissa dell'elemento centrale di S_x vengono man mano inseriti in P_d
7. il passo 7 richiede tempo $O(n)$ perchè P_d contiene al più n punti e per ciascuno di essi vengono computate al più 11 distanze, 11 confronti e 11 aggiornamenti di d, p e q.
8. il passo 8 richiede tempo $O(1)$

Costo computazionale dell'algoritmo per la coppia più vicina definito mediante 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 $T(n) = O(n \log n)$.

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 conente 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: $O(n^3)$ perchè

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

Sottosequenza di somma massima di un array di numeri

- II 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. 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

$$\sum_{i=0}^{n-1} \theta(n-i) = \sum_{i=1}^{n} \theta(i) = \theta(\sum_{i=1}^{n} i) = \theta(n^2)$$

Sottosequenza di somma massima di un array di numeri

- III soluzione: Divide et Impera

  Algoritmo A:

  1. Se $i = j$ restituisci la sottosequenza formata da $a[i]$
  2. Se $i < j$ invoca ricorsivamente $A(i, (i+j)/2)$ e $A((i+j)/2+1, j)$: la sottosequenza cercata o è una di quelle resituite dalle 2 chiamate ricorsive o si trova a cavallo delle due metà dell'array
  3. La sottosequenza di somma massima tra quelle che interesecano entrambe le metà dell'array si trova nel seguente modo:
     - 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 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.
     - scandisce l'array a partire dal'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 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 $s_1 + s_2$

  Tempo di esecuzione:

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

  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'lagoritmo calcola questi valori per ogni j e prende il massimo degli n valori computati. Il tempo dell'algoritmo quindi è $O(n)$.

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: 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'algoritmo per calcolare la profondita' 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'algoritmo per calcolare la profondita' di un nodo u

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

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

Divide et impera su alberi

- **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 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 $O(|\mathbf{T}|)$
- Se l'albero ha n nodi la visita richiede tempo $T(n)= O(n)$

Analisi dell'algoritmo Decomponibile

La funzione $T(n)$ che esprime il tempo di esecuzione dell'algoritmo Decomponibile su un albero binario con n nodi può essere descritta dalla seguente relazione di ricorrenza, dove $r-1$ è il numero di nodi del sottoalbero sinistro.

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

SI simostra per induzione che $T(n) = O(n)$ (si veda libro)

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      Antiticipata( u.sx );
5      Antiticipata( u.dx );
6    }
```

  $O(n)$ tempo per n nodi

Visita di un albero binario: postorder

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

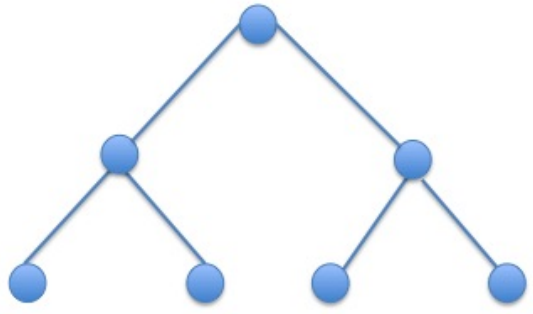
```
1  Posticipata( u ):
2    IF (u != null) {
3      Posticipata( u.sx );
4      Posticipata( u.dx );
5      elabora(u);
6    }
```

  $O(n)$ tempo per n nodi

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

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   }
```

Trasmissione dell'informazione tra chiamate ricorsive

- **postorder** : l'informazione è trasferita dalle foglie alla radice
  – la soluzione del problema per $T(\mathtt{u})$ può essere ottenuta dalla soluzioni dei sottoproblemi per $T(\mathtt{u.sx})$ e $T(\mathtt{u.dx})$
- **passaggio dei parametri** : informazione passata attraverso i parametri dalla radice alle foglie
  – la soluzione del problema per $T(\mathtt{u})$ può esserre 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   }
```