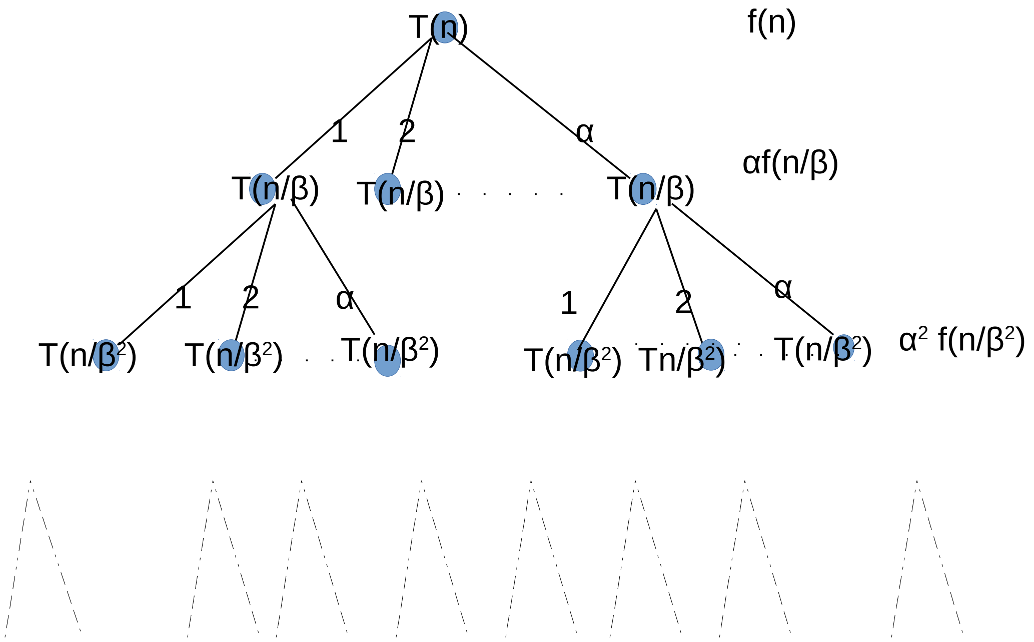
Soluzione delle relazioni di ricorrenza

- Consideriamo la seguente relazione di ricorrenza:

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

  con $\alpha \geq 1$ e $\beta > 1$ costanti.
- Nel caso in cui $T(n)$ sia la funzione che scaturisce dall'analisi di un algoritmo basato sul paradigma del Divide et Impera, $f(n)$ è il tempo per il lavoro di suddivisione e di ricombinazione. In altre parole, $f(n) = d(n) + r(n)$.
- In realtà nella ricorrenza n/β dovrebbe essere $\lceil n/\beta \rceil$ oppure $\lfloor n/\beta \rfloor$. Per stimare $T(n)$, assumiamo per semplicità che n sia una potenza di β in modo da poter omettere le parti intere superiori o inferiori.

Soluzione delle relazioni di ricorrenza



Sia h l'altezza dell'albero (h+1 livelli). Per i<h, Il tempo di esecuzione per tutte le chiamate ricorsive a livello i e` al piu` $\alpha^i f(n/\beta^i)$.

Soluzione delle relazioni di ricorrenza

- L'algoritmo non effettua chiamate ricorsive quando l'input ha dimensione al più c. Quindi la ricorrenza non sarà più applicata quando si arriva al livello i per cui per la prima volta $n/\beta^i \leq c$, cioè $i = \lceil \log_\beta n/c \rceil$.

- Il numero di livelli dell'albero è quindi $\lceil \log_\beta n/c \rceil + 1$ (partiamo dal livello 0) e ciascun nodo sul livello $\lceil \log_\beta n/c \rceil$ corrisponde al tempo $T(n/\beta^{\lceil \log_\beta n/c \rceil}) \leq T(c) \leq c_0$. Il tempo totale per eseguire le $\alpha^{\lceil \log_\beta n/c \rceil}$ chiamate ricorsive in quest'ultimo livello è quindi $\leq \alpha^{\lceil \log_\beta n/c \rceil} c_0$.

- Abbiamo visto che per $i < \lceil \log_\beta n/c \rceil$, il tempo per eseguire tutte le chiamate sul livello i è $\alpha^i f(n/\beta^i)$.

- Sommando su tutti i livelli (compreso l'ultimo) si ha

$$T(n) \leq \alpha^{\lceil \log_\beta n/c \rceil} c_0 + \sum_{i=0}^{\lceil \log_\beta n/c \rceil - 1} \alpha^i f(n/\beta^i).$$

Soluzione delle relazioni di ricorrenza

- Vogliamo stimare la funzione

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

quando la funzione $f(n)$ è limitata da $c'n^k$, dove c' e k sono due costanti tali che $k \geq 0$, $c' > 0$ ($f(n)$ polinomiale).

- Da quanto ottenuto nella slide precedente si ha che

$$\begin{aligned} T(n) &\leq \alpha^{\lceil \log_\beta n/c \rceil} c_0 + \sum_{i=0}^{\lceil \log_\beta n/c \rceil - 1} \alpha^i f(n/\beta^i) \\ &\leq \alpha^{\lceil \log_\beta n/c \rceil} c_0 + \sum_{i=0}^{\lceil \log_\beta n/c \rceil - 1} \alpha^i c'(n/\beta^i)^k \end{aligned}$$

Soluzione delle relazioni di ricorrenza

- Abbiamo visto che se $f(n) \leq c' n^k$, dove c' e k sono due costanti tali che $k \geq 0$, $c' > 0$, allora

$$T(n) \leq \alpha^{\lceil \log_\beta n/c \rceil} c_0 + c' n^k \sum_{i=0}^{\lceil \log_\beta n/c \rceil - 1} (\alpha/\beta^k)^i.$$

- consideriamo i 2 seguenti casi:
- $\alpha = \beta^k$: In questo caso si ha

$$\alpha^{\lceil \log_\beta n/c \rceil} c_0 = (\beta^k)^{\lceil \log_\beta n/c \rceil} c_0 < (\beta^k)^{\log_\beta (n/c)+1} c_0 = \beta^k (n/c)^k c_0 = O(n^k)$$

e

$$c' n^k \sum_{i=0}^{\lceil \log_\beta n/c \rceil - 1} (\alpha/\beta^k)^i = c' n^k \sum_{i=0}^{\lceil \log_\beta n/c \rceil - 1} 1 = c' n^k \lceil \log_\beta n/c \rceil = O(n^k \log_\beta n).$$

Quindi $T(n) = O(n^k) + O(n^k \log_\beta n) = O(n^k \log_\beta n) = O(n^k \log n)$.

Soluzione delle relazioni di ricorrenza

- $\alpha \neq \beta^k$: In questo caso si ha

$$\begin{aligned}
\alpha^{\lceil \log_\beta n/c \rceil} c_0 &< c_0 \alpha^{\log_\beta n/c+1} = c_0 \alpha \cdot \alpha^{\log_\beta n/c} = c_0 \alpha \cdot \alpha^{\log_\alpha (n/c) \log_\beta \alpha} \\
&= c_0 \alpha (n/c)^{\log_\beta \alpha} = O(n^{\log_\beta \alpha}), \qquad (1)
\end{aligned}$$

e

$$c' n^k \sum_{i=0}^{\lceil \log_\beta n/c \rceil - 1} (\alpha/\beta^k)^i = c' n^k \cdot \frac{(\alpha/\beta^k)^{\lceil \log_\beta n/c \rceil} - 1}{(\alpha/\beta^k) - 1}. \qquad (2)$$

Soluzione delle relazioni di ricorrenza

Consideriamo i due sottocasi di $\alpha \neq \beta^k$: $\alpha < \beta^k$ e $\alpha > \beta^k$

- Caso $\alpha < \beta^k$:

$$\begin{aligned}\frac{(\alpha/\beta^k)^{\lceil \log_\beta (n/c) \rceil} - 1}{(\alpha/\beta^k) - 1} &= \frac{1 - (\alpha/\beta^k)^{\lceil \log_\beta n/c \rceil}}{1 - (\alpha/\beta^k)} \\ &< \frac{1}{1 - (\alpha/\beta^k)} = \frac{\beta^k}{\beta^k - \alpha} = O(1).\end{aligned}$$

Si ha quindi che la suddetta relazione insieme alla (1) e alla (2) della slide precedente implicano:

$$T(n) \leq O(n^{\log_\beta \alpha}) + c' n^k O(1) = O(n^{\log_\beta \alpha} + n^k).$$

Si noti che $\alpha < \beta^k$ implica $\log_\beta \alpha < k$ e di conseguenza si ha

$$T(n) = O(n^{\log_\beta \alpha} + n^k) = O(n^k).$$

Soluzione delle relazioni di ricorrenza

- Caso $\alpha > \beta^k$:

$$\begin{aligned}\frac{(\alpha/\beta^k)^{\lceil \log_\beta (n/c) \rceil} - 1}{(\alpha/\beta^k) - 1} &< \frac{(\alpha/\beta^k)^{\log_\beta (n/c) + 1} - 1}{(\alpha/\beta^k) - 1} = \frac{(\alpha/\beta^k)(\alpha/\beta^k)^{\log_\beta (n/c)} - 1}{(\alpha/\beta^k) - 1} \\ &= O((\alpha/\beta^k)^{\log_\beta (n/c)}) = O((\alpha/\beta^k)^{\log_{(\alpha/\beta^k)} (n/c) \log_\beta (\alpha/\beta^k)}) \\ &= O((n/c)^{\log_\beta (\alpha/\beta^k)}) = O((n/c)^{\log_\beta \alpha - \log_\beta \beta^k}) \\ &= O(n^{\log_\beta (\alpha) - k})\end{aligned}$$

Si ha quindi che la suddetta relazione insieme alla (1) e alla (2) della slide precedente implicano:

$$T(n) \leq O(n^{\log_\beta \alpha}) + c' n^k O(n^{\log_\beta (\alpha) - k}) = O(n^{\log_\beta \alpha} + n^k n^{\log_\beta (\alpha) - k}) = O(n^{\log_\beta \alpha}).$$

Soluzione delle relazioni di ricorrenza

- Abbiamo stimato la funzione
$$T(n) \le \begin{cases} c_0 & \text{se } n \le c \\ \alpha T(n/\beta) + c' n^k & \text{altrimenti,} \end{cases}$$
dove c' e k sono due costanti tali che $k \ge 0$, $c' > 0$.
- Abbiamo provato
$$T(n) = \begin{cases} O(n^k) & \text{se } \alpha < \beta^k \\ O(n^k \log n) & \text{se } \alpha = \beta^k \\ O(n^{\log_\beta \alpha}) & \text{se } \alpha > \beta^k \end{cases}$$
- **Esempi:** Nel caso di MergeSort $\alpha = 2$, $\beta = 2$ e $k = 1$. Si ha $\alpha = \beta^k$ e quindi $T(n) = O(n^k \log n) = O(n \log n)$.
Nel caso dell'algoritmo per la ricerca binaria $\alpha = 1$, $\beta = 2$ e $k = 0$. Si ha $\alpha = \beta^k$ e quindi $T(n) = O(n^k \log n) = O(\log n)$.

Soluzione delle relazioni di ricorrenza quando n non è potenza di β

- Consideriamo la seguente relazione di ricorrenza:
$$T(n) = \begin{cases} c_0 & \text{se } n \le c \\ \alpha T(n/\beta) + c' n^k & \text{altrimenti} \end{cases}$$
con $\alpha \ge 1$ e $\beta > 1$ costanti.
- Quando n non è una potenza di β la taglia di ciascun sottoproblema è $\lceil n/\beta \rceil$ oppure $\lfloor n/\beta \rfloor$.
- Siccome vogliamo stabilire un limite superiore per $T(n)$ mettiamoci nel caso peggiore in cui la taglia di ciascun sottoproblema è $\lceil n/\beta \rceil$
Consideriamo quindi la seguente relazione di ricorrenza:
$$T(n) = \begin{cases} c_0 & \text{se } n \le c \\ \alpha T(\lceil n/\beta \rceil) + c' n^k & \text{altrimenti} \end{cases}$$
con $\alpha \ge 1$, $\beta > 1$ e $k \ge 0$ costanti.
- Usando questa nuova relazione di ricorrenza potremmo usare un procedimento simile a quello usato per il caso in cui n è potenza di β per provare le stesse limitazioni superiori viste per quel caso. Nel seguito invece useremo un argomento molto semplice per dedurre che quelle limitazioni valgono anche quando n non è potenza di β.

Soluzione delle relazioni di ricorrenza quando n non è potenza di β

- Sia p il più piccolo intero positivo per cui $n \leq \beta^p$, cioè p è l'intero per cui $\beta^{p-1} < n \leq \beta^p$.
- Osserviamo che siccome $T(n)$ è una funzione non decrescente allora $T(n) \leq T(\beta^p)$.
- Applicando a $T(\beta^p)$ la limitazione asintotica dimostrata per le potenze di β si ha

$$T(\beta^p) = \begin{cases} O((\beta^p)^k) & \text{se } \alpha < \beta^k \\ O((\beta^p)^k \log(\beta^p)) & \text{se } \alpha = \beta^k \\ O((\beta^p)^{\log_\beta \alpha}) & \text{se } \alpha > \beta^k \end{cases} \qquad (3)$$

- Osserviamo che $\beta^p = \beta\beta^{p-1} < \beta n$, dal momento che $\beta^{p-1} < n$. Si ha quindi che

$$O((\beta^p)^k) = O((\beta n)^k) = O(\beta^k n^k) = O(n^k),$$
$$O((\beta^p)^k \log(\beta^p)) = O((\beta n)^k) \log(\beta n)) = O(n^k(\log(\beta) + \log n)) = O(n^k \log n),$$
$$O((\beta^p)^{\log_\beta \alpha}) = O((\beta n)^{\log_\beta \alpha}) = O(\beta^{\log_\beta \alpha} n^{\log_\beta \alpha}) = O(n^{\log_\beta \alpha}).$$

La (3) può essere quindi scritta come segue

$$T(\beta^p) = \begin{cases} O(n^k) & \text{se } \alpha < \beta^k \\ O(n^k \log n) & \text{se } \alpha = \beta^k \\ O(n^{\log_\beta \alpha}) & \text{se } \alpha > \beta^k \end{cases}$$

- Poichè $T(n) \leq T(\beta^p)$ allora le limitazioni appena provate valgono anche per $T(n)$.

Esempi di relazioni di ricorrenza della forma $T(n) \leq \alpha T(n/\beta) + cn^k$

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

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

Una relazione di questo tipo è quella che scaturisce dall'analisi per il caso ottimo' di QuickSelect. Qui tralasciamo il caso in cui l'elemento da selezionare è proprio il pivot.
Quindi,

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

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, decomponi x nei due interi x_s e x_d aventi ciascuno la metà delle cifre di x e decomponi y nei due interi y_s e y_d aventi ciascuno la metà delle cifre di y.

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:

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

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

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:

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

Continua nella prossima slide

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

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

ESEMPI DI RELAZIONI DI RICORRENZA DELLA FORMA $T(n) \leq \alpha T(n/\beta) + cn^k$

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

Sommatorie utili

- $$\sum_{i=1}^{n} i = n(n+1)/2$$
- $$\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$$
- $$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1} \text{ per } a \neq 1$$
- $$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a} \text{ per } 0 < a < 1.$$

Divide et impera su alberi

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

```
Decomponibile(u):
  IF (u == null) {
    RETURN valore base;
  } ELSE {
    i=0;
    FOR( ciascun figlio f di u ){

    risultatiFigli[i] = Decomponibile(f);
    i=i+1 }
    RETURN Ricombina(risultatiFigli);
  }
```

La ricombinazione del risultati della 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 $u = \texttt{null}$ o una foglia
- **Decomposizione**: riformula il problema per i sottoalberi radicati nei figli `u.sx` e `u.dx`
- **Ricombinazione**: ottieni il risultato con `Ricombina`

```
Decomponibile(u):
  IF (u == null) {
    RETURN valore base;
  } ELSE {
    risultatoSX = Decomponibile(u.sx);
    risultatoDx = Decomponibile(u.dx);
    RETURN Ricombina(risultatoSX, risultatoDx);
  }
```

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)

Analisi dell'algoritmo Decomponibile

- Nell'analisi precedente abbiamo usato il fatto che $\sum_{v \in T_w} c_v = |T_w| - 1$.
- È facile vedere che vale questa uguaglianza in quanto ogni nodo di T_w, eccezion fatta per la radice w, è figlio di un unico nodo v dell'albero T_w e quindi viene contato esattamente una volta in quella sommatoria.

Analisi dell'algoritmo Decomponibile per un albero binari mediante relazione di ricorrenza

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 $r - 1 \geq 0$ è il numero di nodi del sottoalbero sinistro.

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

Dimostriamo per induzione che $T(n) \leq c'n$ per $n \geq 1$ e per una costante $c' > 0$. In altre parole $T(n) = O(n)$.

- Base: $T(1) \leq c_0$ implica $T(1) \leq c'$ se si sceglie $c' \geq c_0$.
- Passo induttivo: Assumiano $T(m) \leq c'm$ per ogni $1 \leq m < n$ e dimostriamo $T(n) \leq c'n$.
  Applichiamo relazione di ricorrenza: $T(n) \leq T(r-1) + T(n-r) + c$.
  L'ipotesi induttiva implica $T(r-1) \leq c'(r-1)$ e $T(n-r) \leq c'(n-r)$.
  Si ha quindi $T(n) \leq c'(r-1) + c'(n-r) + c = c'n - c'r + c$.
  Affinché risulti $T(n) \leq c'n$ basta scegliere c' in modo che $c'r \geq c$ cioè $c' \geq c/r$. Non sappiamo quanto vale r ma sappiamo che $r \geq 1$ per cui basta scegliere $c' \geq c$.
- Dalla base dell'induzione e dal passo induttivo, sappiamo che basta scegliere $c' = \max\{c_0, c\}$ affinché valga $T(n) \leq c'n$ per $n \geq 1$.

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

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

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

```
Altezza( u ):
  IF (u == null) {
    RETURN -1;
  } ELSE {
    altezzaSX = Altezza( u.sx );
    altezzaDX = Altezza( u.dx );
    RETURN max( altezzaSX, altezzaDX ) + 1;
  }
```

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

Visita di un albero binario: inorder

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

```
Simmetrica( u ):
  IF (u != null) {
    Simmetrica( u.sx );
    elabora(u);
    Simmetrica( u.dx );
  }
```

$O(n)$ tempo per n nodi

Visita di un albero binario: preorder

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

```
Anticipata( u ):
  IF (u != null) {
    elabora(u);
    Antiticipata( u.sx );
    Antiticipata( u.dx );
  }
```

$O(n)$ tempo per n nodi

Visita di un albero binario: postorder

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

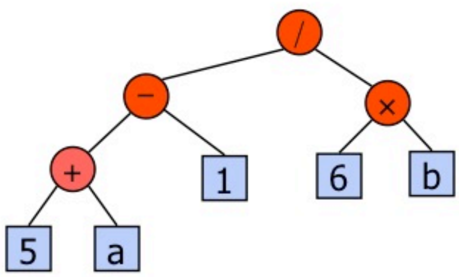
```
Posticipata( u ):
  IF (u != null) {
    Posticipata( u.sx );
    Posticipata( u.dx );
    elabora(u);
  }
```

$O(n)$ 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

```
Valuta( u ):
  IF (u==null) {
    RETURN null;
  }
  IF (u.sx == null && u.dx==null) {
    RETURN u.dato;
  } ELSE {
    valSinistra=Valuta( u.sx );
    valDestra= Valuta( u.dx );
    ris= Calcola(u.dato,valSinistra ,valDestra);
    RETURN ris;
  }
```

- 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.

Uso della visita inorder per stampare l'espressione aritmetica rappresentata da un albero binario

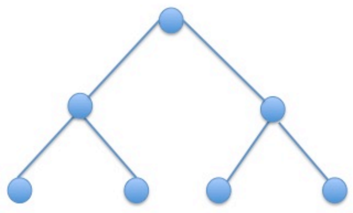
- Il seguente algoritmo stampa l'espressione l'espressione aritmetica rappresentata da un albero binario.
- L'algoritmo deve effettuare una visita inorder in modo che per ogni nodo interno *u*, stampi prima la sottoespressione a sinistra dell'operatore contenuto in u, poi l'operatore contenuto in u e infine la sottoespressione a destra dell'operatore.
- Per ciascun nodo interno *u*, la sottoespressione rappresentata dal sottoalbero radicato in *u* viene stampata tra una coppia di parentesi tonde:
  - la parentesi sinistra viene aperta prima di invocare l'algoritmo ricorsivamente sul figlio sinistro di *u*
  - la parentesi destra viene chiusa dopo aver invocato l'algoritmo ricorsivamente sul figlio destro di *u*

```
Stampa( u ):
  IF (u==null) {
    print("");
  }
  IF (u.sx == null && u.dx==null) {
    print(u.dato);
  } ELSE {
    print("(");
    Stampa(u.sx);
    print(u.dato);
    Stampa(u.dx);
    print(")");
  }
```

Algoritmo per verificare se un albero binario è completamente bilanciato

Definizioni:

- Albero binario **proprio**: ogni nodo interno ha sempre due figli non vuoti
- Albero **completamente bilanciato**: albero proprio 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

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

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 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'algoritmo per calcolare la profondità di un nodo `u`

```
Profondita( u ):
  IF (u.padre==null) {
    RETURN 0;
  }
  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 $T(\mathtt{u})$ 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 essere ottenuta utilizzando l'informazione raccolta dalla radice fino al nodo `u`

Esempio: stampa la profondità di tutti i nodi

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

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 alla 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 $\text{profondita}(\mathtt{u}) = \text{altezza}(T(\mathtt{u}))$

```
Cardine( u, p ):
  IF (u == null) {
    RETURN -1;
  } ELSE {
    altezzaSX = Cardine( u.sx, p+1 );
    altezzaDX = Cardine( u.dx, p+1 );
    altezza = max( altezzaSX, altezzaDX ) + 1;
    IF (p == altezza) PRINT u.dato;
    RETURN altezza;
  }
```

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.

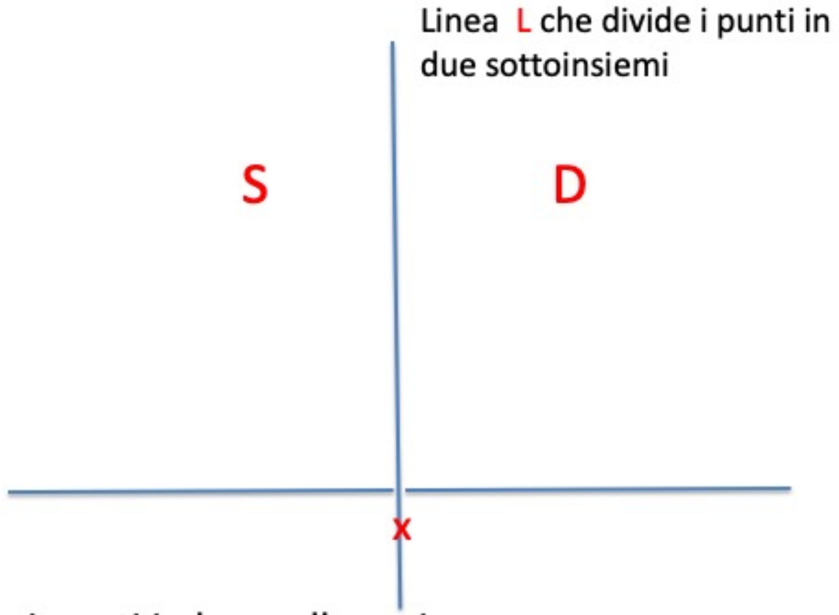
Utilizzando la tecnica del divide et impera, il problema può essere risolto in tempo $O(n \log n)$.

Il problema della coppia più vicina

Idea intuitiva.

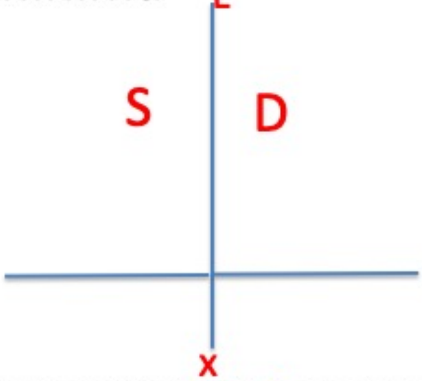
- l'insieme ha cardinalià minore o uguale di una certa costante: usiamo la ricerca esaustiva.
- altrimenti: lo dividiamo in due parti uguali S e D, per esempio quelli a sinistra e quelli a destra di una fissata linea verticale
  - troviamo ricorsivamente le soluzioni per S e quella per D individuando 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



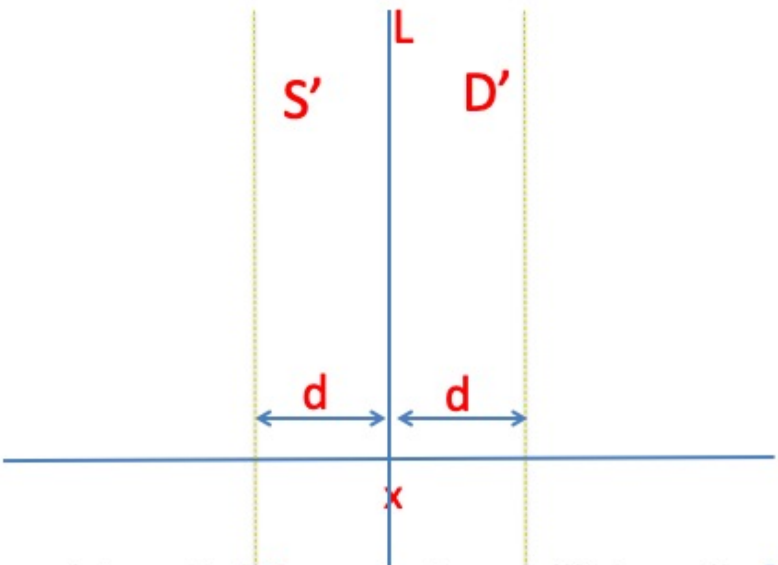
- Ordina i punti in base alle ascisse
- **x**= ascissa punto **pc** centrale nell'ordinamento
- **S**= insieme dei punti a sinistra di **pc** nell'ordinamento
- **D**= insieme dei punti a destra di **pc** nell'ordinamento

Individuazione della coppia di punti a distanza minima



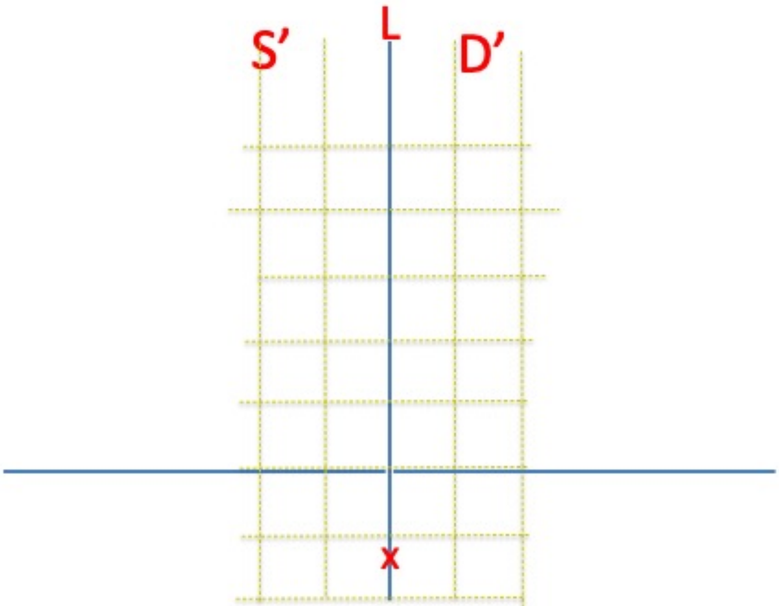
- I 2 punti a distanza minima o sono entrambi in S, o sono entrambi in D, o uno dei due si trova in S e l'altro in D
- Divide et impera:
- Decomposizione: partiziona l'insieme di punti in S e D
- Soluzione sottoproblemi: cerca la coppia a distanza minima d_S in S e la coppia a distanza minima d_D in 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 nel far questo ignora le coppie che evidentemente sono a distanza maggiore di d. Alla fine restituisce la coppia con distanza pari a $\min\{d, d_{SD}\}$.

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



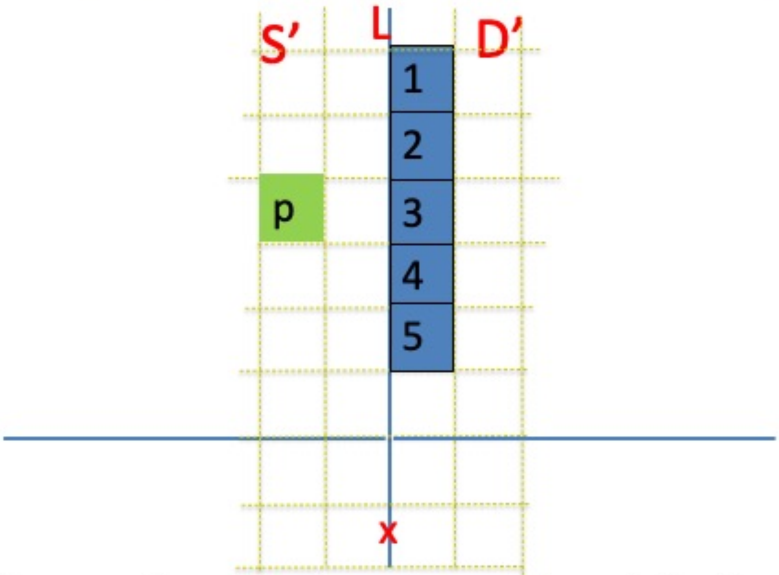
- S'= insieme dei punti di S con ascissa nell'intervallo [x-d,x]
- D'= insieme dei punti di D con ascissa nell'intervallo [x,x+d]
- è sufficiente considerare coppie (p,q) con p in S' e q in D' ' in quanto le altre coppie (p,q) con p in S e q in D sono a distanza maggiore di d

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



- **Osservazione 1:** Ciascun quadrato contiene al piu` 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 tanti quadrati di lato uguale a d/2

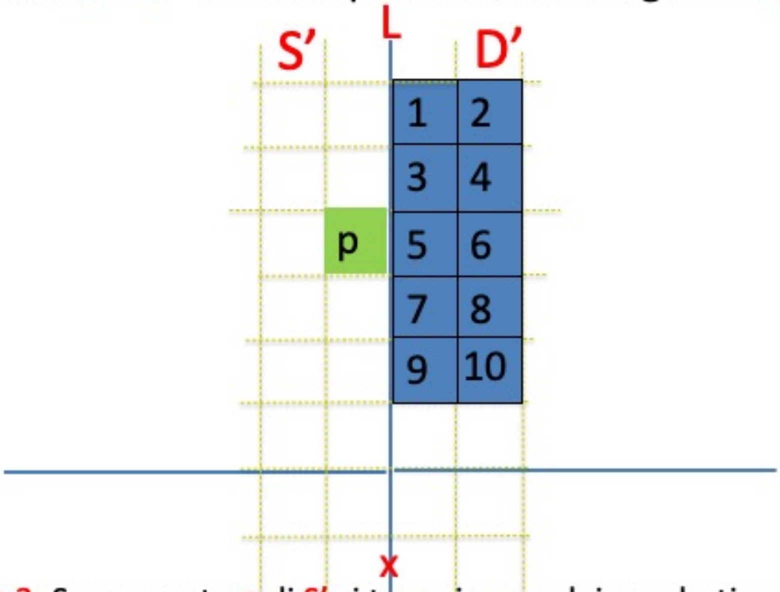


Osservazione 2: Se un punto p di S’ si trova in uno dei quadrati più a sinistra allora i punti di D’ a distanza minore di d da p possono trovarsi solo nei quadrati di D’ confinanti con L e in particolare in 5 di questi quadrati, in quello alla stessa altezza del quadrato contenente p, nei 2 quadrati al di sopra di questo e nei due al di sotto.

Ad esempio se p è nel quadrato verde, allora un punto q di D’ a distanza minore di d da p può trovarsi solo in uno dei quadrati in D’ colorati di azzurro

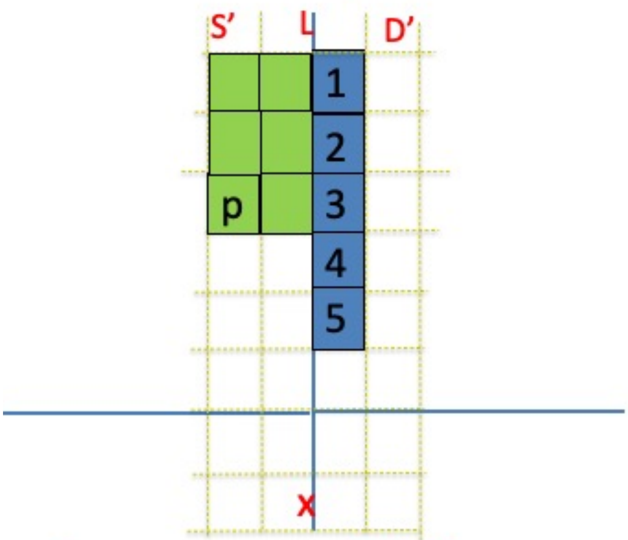
Se q è più in alto rispetto a p allora q si trova in uno dei quadrati 1, 2, 3; altrimenti si trova ir uno dei quadrati 3, 4, 5.

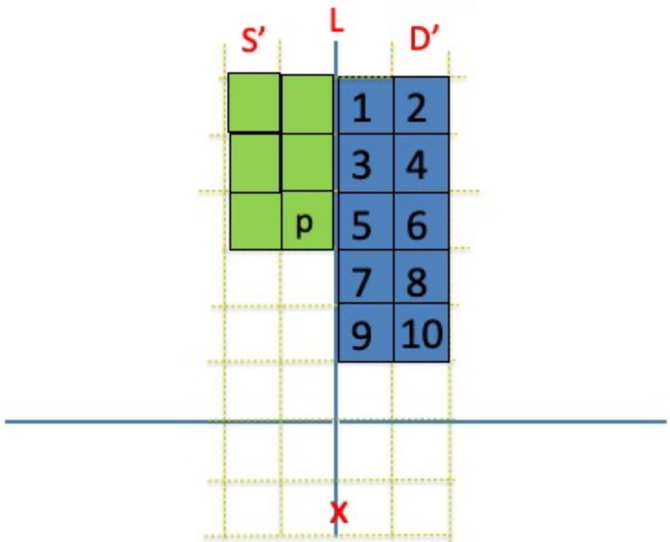
Dividiamo S’ e D’ in tanti quadrati di lato uguale a d/2



- Osservazione 3: Se un punto p di S’ si trova in uno dei quadrati confinati con L allora i punti di D’ a distanza minore di d da p possono trovarisi solo nei due quadrati di D’ alla stessa altezza di quello contenente p o nei quattro quadrati al di sopra di questi due quadrati o nei quattro al di sotto.
- Se p si trova nel quadrato verde allora un punto q di D’ a distanza minore di d da p può trovarsi solo in uno dei 10 quadrati in D’ colorati di azzurro
    - Se q è più 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

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





- P_d = array dei punti di S’ e D’ in ordine non decrescente di altezza
- per ogni punto p in P_d cerchiamo il punto a distanza minima da p tra quelli più in alto di p
- Ciascun quadrato contiene al più 1 punto →un punto q di D’ a distanza al più d da p si trova al più 11 locazioni in avanti nell’array P_d rispetto a p
    - tra p e q possono esserci infatti al più 5 punti di D’ e 5 punti di S’ (II figura): ad esempio se q è più in alto rispetto a p allora tra p e q può esserci al più un punto di D’ per ciascuno dei quadrati 1-6 (meno quello contente q) e un punto di S’ per ciascuno dei quadrati verdi (meno quello contente p)

L'algoritmo che trova la coppia più vicina

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 $O(n)$ perché 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.

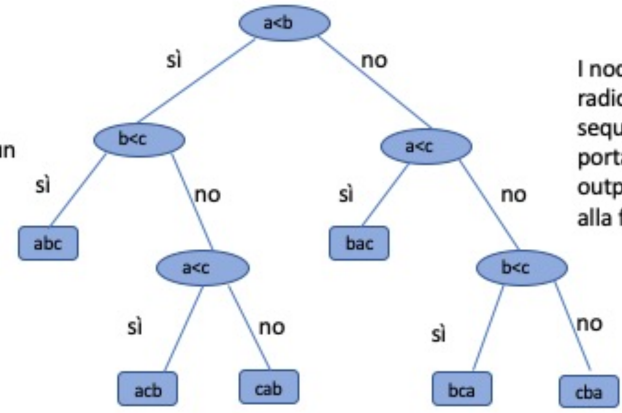
Abbiamo $T(n) = O(n \log n)$.

```
CoppiaPiuVicina(Px,Py,n):
IF(n<=3){Return RicercaEsaustiva(Px,Py,n);}
 ELSE{ p=Px[n/2];}
j=k=0;
FOR(i=0;i<n/2;i=i+1){
Sx[i]=Px[i]; Dx[i]=Px[i+n/2];}
if n%2==1 Dx[n-1]=Px[n-1];
FOR(i=0;i<n;i=i+1){
IF(Py[i].x<=p.x){Sy[j]=Py[i]; j=j+1;}
ELSE {Dy[k]=Py[i]; k=k+1;}
}
(ps,qs)= CoppiaPiuVicina(Sx,Sy,n/2);
(pd,qd)=CoppiaPiuVicina(Dx,Dy,(n+1)/2);
IF(Dist(ps,qs)<Dist(pd,qd)){d=Dist(ps,qs); (p,q)=(ps,qs);}
ELSE {d=Dist(pd,qd); (p,q)=(pd,qd);}
FOR(i=m=0;i<n;i=i+1){
 IF(|Py[i].x-p.x|<=d){Pd[m]=Py[i]; m=m+1;}
}
FOR(i=0;i<m;i=i+1){
  FOR(j=i+1;j<=min{i+11,m};j=j+1){
     IF(Dist(Pd[i],Pd[j])<d){ d=Dist(Pd[i],Pd[j]); (p,q)=(Pd[i],Pd[j]);}
 }
}
RETURN(p,q);
}
```

Albero di decisione di un particolare algoritmo basato sui confronti per ordinare 3 numeri

I nodi interni sono associati ai confronti

Ciascuna foglia e` associata ad un possible ordinamento (permutazione)



I nodi lungo un percorso dalla radice ad una foglia forniscono la sequenza di confronti che hanno portato l'algoritmo a dare in output l'ordinamento associato alla foglia

Altezza dell'albero = massimo numero di confronti fatti in un'esecuzione dall'algoritmo

Limite inferiore per gli algoritmi di ordinamento basati su confronti

- L'albero di decisione di un qualsiasi algoritmo di ordinamento deve avere una foglia per ogni possibile ordinamento dell'input
- Se l'input consiste di n numeri allora l'albero di decisione deve contenere almeno n! foglie. Sia h l'altezza dell'albero.
- Il massimo numero di foglie in un albero binario di altezza h e` 2^h
- Ne consegue che l'altezza h dell'albero deve essere tale che

  $2^h \geq n! \rightarrow h \geq \log n! \geq \log (n/2)^{n/2} = n/2 \log(n) - n/2$

  $\rightarrow h = \Omega(n\log n)$

- Siccome h=numero confronti fatti nel caso pessimo dall'algoritmo allora abbiamo dimostrato che il numero di confronti effettuati nel caso pessimo di da qualsiasi algoritmo di ordinamento basato su confronti e` $\Omega(n\log n)$.