TECNICA *DIVIDE ET IMPERA*

1. Introduzione

Data l'istanza di un problema, la strategia del divide-et-impera suggerisce di *partizionarla* in k sotto-istanze in modo da ottenere k nuove istanze per lo stesso problema di partenza ma di dimensioni ridotte. Queste nuove istanze vanno risolte e infine le k soluzioni ottenute vanno ricombinate in modo da avere una soluzione per l'istanza di partenza. Se le istanze ottenute dalla partizione sono ancora relativamente grandi, allora la strategia può essere riapplicata. In questo modo vengono generate istanze sempre più piccole, fino ad ottenere istanze sufficientemente piccole da poter essere risolte senza ulteriori partizionamenti.

Poichè gli algoritmi basati sul divide et impera producono sotto-istanze dello stesso tipo dell'istanza originale è naturale descriverli facendo uso della ricorsione. Uno schema generale è il seguente:

DIVIDE-ET-IMPERA
INPUT l'istanza P del problema
    IF la dimensione di P è piccola THEN risolvi P direttamente e sia SOL la sua soluzione
    ELSE
        Dividi P in k istanze $P_1, P_2, \dots P_k$
        FOR $i = 1$ TO k DO
            invoca DIVIDE-ET-IMPERA(P_i) per ottenere la soluzione SOL_i per P_i
        ENDFOR
        Combina le soluzioni $SOL_1, SOL_2, \dots SOL_k$ per ottenere la soluzione SOL per P
    ENDIF
OUTPUT SOL.

Ovviamente ci sono molti dettagli, tutt'altro che marginali, che lo schema non tenta di specificare:
Come partizionare l'istanza da risolvere? Come ricombinare le soluzioni ottenute? Quando l'istanza può ritenersi sufficientemente piccola da poter essere risolta senza ricorrere ad ulteriori partizionamenti?

Questa tecnica non è legata ad una particolare applicazione e può, sostanzialmente, essere applicata ad ogni tipo di problema (di ricerca, di ottimizzazione ecc.), anche se non sempre fornisce soluzioni efficienti. Nelle sezioni che seguono ne vedremo diverse applicazioni. Possiamo tuttavia fare alcune importanti considerazioni. Un primo aspetto riguarda il metodo da seguire per calcolare la complessità di algoritmi di questo tipo.
Se con $T(n)$ indichiamo il tempo che il procedimento appena illustrato impiega per risolvere un istanza di dimensione n, si avrà la seguente *relazione di ricorrenza*:

$$T(n) = \begin{cases} O(1) & \text{se } n \text{ è piccolo} \\ \sum_{i=1}^{k} T(n_i) + D(n) + C(n) & \text{altrimenti} \end{cases}$$

dove $D(n)$ è il tempo richiesto per partizionare l'istanza mentre $C(n)$ è il tempo necessario a combinare insieme le k soluzioni ottenute per avere la soluzione al problema di partenza.
La risoluzione di una relazione di ricorrenza non è sempre agevole ed anzi, esistono molti casi in cui la trattazione è alquanto difficile. Fortunatamente le relazioni di ricorrenza che risultano dall'applicazione della tecnica del divide et impera appartengono quasi sempre alla stessa classe di relazioni di ricorrenza o sue semplici varianti. Nella stragrande maggioranza degli algoritmi conosciuti, l'istanza viene bipartita (i.e. $k = 2$), inoltre per motivi di efficienza, quando possibile, si tenta di mantenere i sottoproblemi *bilanciati* (i.e. ottenere sottoproblemi approssimativamente della stessa dimensione). Per quanto detto, le equazioni di ricorrenza che più comunemente vengono fuori sono del tipo:

$$T(n) = \begin{cases} O(1) & \text{se } n \text{ è piccolo} \\ 2 \cdot T\left(\frac{n}{2}\right) + D(n) + C(n) & \text{altrimenti} \end{cases}$$

Più in generale, supponendo che un problema di dimensione n generi a sottoproblemi di dimensione $\frac{n}{b}$ e che dividere in sottoproblemi e combinare le soluzioni richiede tempo $O(f(n))$, la relazione di ricorrenza corrispondente a questo scenario è

$$T(n) \leq a \cdot T\left(\frac{n}{b}\right) + O(f(n))$$

Un metodo informale ma efficace per risolvere quest'equazione è dato dall'albero delle chiamate ricorsive. La radice dell'albero ha costo $O(f(n))$ ed ha a figli, ciascuno di costo $O\left(f\left(\frac{n}{b}\right)\right)$. Ciascuno di questi figli ha a sua volta a figli, ciascuno di costo $O\left(f\left(\frac{n}{b^2}\right)\right)$. In generale a livello j dell'albero ci sono a^j nodi e ciascuno di questi ha costo $O\left(f\left(\frac{n}{b^j}\right)\right)$. Il costo di ogni foglia è $\Theta(1)$ e la profondità dell'albero è al più $\log_b n$ poiché $\frac{n}{b^{\log_b n}} = 1$. Si può ottenere la soluzione dell'equazione sommando i costi dei livelli interni dell'albero di ricorsione ed il costo delle al più $a^{\log_b n} = n^{\log_b a}$ foglie. Vale a dire

$$T(n) \leq \sum_{j=0}^{\log_b n} a^j \cdot O\left(f\left(\frac{n}{b^j}\right)\right) + n^{\log_b a}$$

.

ESEMPIO 1. Si consideri il seguente scenario di Divide et Impera:
Dividi l'input in due parti di ugual dimensione, individua una delle parti e risolvila ricorsivamente. Restituisci la soluzione di quella parte come soluzione del problema spendendo tempo costante per l'operazione di divisione e individuazione (questo è lo scenario tipico dell'algoritmo RICERCA BINARIA per la ricerca in insiemi ordinati).
Per quanto detto l'algoritmo corrispondente a questo scenario ha un tempo di esecuzione $T(n)$ che soddisfa la ricorrenza

$$T(n) \leq T\left(\frac{n}{2}\right) + O(1)$$

si ha quindi

$$T(n) = \sum_{j=0}^{\log n} O(1) + O(1) = O\left(\sum_{j=0}^{\log n} 1\right) + O(1) = O(\log n)$$

ESEMPIO 2. Si consideri il seguente scenario di Divide et Impera:
Dividi l'input in due parti di ugual dimensione e risolvile ricorsivamente. Combina i due risultati nella soluzione al problema spendendo tempo lineare per l'operazione di ricombinazione e divisione (questo è lo scenario tipico dell'algoritmo MERGE-SORT per l'ordinamento).
Per quanto detto l'algoritmo corrispondente a questo scenario ha un tempo di esecuzione $T(n)$ che soddisfa la ricorrenza

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

si ha quindi

$$T(n) = \sum_{j=0}^{\log n} 2^j \cdot O\left(\frac{n}{2^j}\right) + O(n) = O\left(\sum_{j=0}^{\log n} n\right) + O(n)O(n \log n)$$

ESEMPIO 3. Si consideri lo stesso scenario dell'esempio 2 ma con tempo $O(n \log n)$ per l'operazione di divisione e ricombinazione. La ricorrenza che viene fuori è allora

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + O(n \log n)$$

si ha pertanto

$$T(n) = \sum_{j=0}^{\log n} 2^j \cdot O\left(\frac{n}{2^j} \log \frac{n}{2^j}\right) + O(n) = O\left(\sum_{j=0}^{\log n} (n \log n - n \cdot j)\right) + O(n) = O\left(n \log^2 n - n\frac{\log n(\log n + 1)}{2}\right) + O(n) = O(n \log^2 n)$$

ESEMPIO 4. Si consideri lo stesso scenario dell'esempio 2 ma con tempo quadratico per l'operazione di divisione e ricombinazione. La ricorrenza che viene fuori è allora

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + O\left(n^2\right)$$

si ha pertanto

$$T(n) = \sum_{j=0}^{\log n} 2^j \cdot O\left(\left(\frac{n}{2^j}\right)^2\right) + O(n) = O\left(\sum_{j=0}^{\log n} \frac{n^2}{2^j}\right) + O(n) = O(n^2)$$

2. Il Problema MASSIMO VALORE DI SOTTOVETTORI

Dato un vettore V di interi, definiamo *sottovettore* di V una qualunque sequenza non vuota di elementi consecutivi del vettore e *valore* del sottovettore la somma dei suoi elementi. Consideriamo quindi il seguente problema:

Dato un vettore V di n interi, determinare il valore massimo tra i valori dei suoi sottovettori.

Ad esempio per il vettore

−4	10	−5	12	−7	−3	8	1

(17) la soluzione è 17.

Un algoritmo di forza bruta restituisce il valore massimo dopo aver calcolato i valori di **tutti** i sottovettori di V . Un vettore di n elementi ha $\binom{n}{2} + n = \frac{n^2}{2} + n$ distinti sottovettori di conseguenza con un simile approccio il meglio che ci si può aspettare è una complessità temporale $O(n^2)$. L' implementazione che segue, calcolando in modo accorto i valori dei diversi sottovettori, garantisce proprio questa prestazione.

```
MAX-VALORE:
INPUT un vettore V di n interi
    SOL ← −∞
    FOR i = 1 TO n DO
        somma ← 0
        FOR j = i TO n DO
            somma ← somma + V[j]
            IF somma > SOL THEN SOL ← somma
        ENDFOR
    ENDFOR
OUTPUT SOL.
```

Nulla ci dice che debbano necessariamente essere calcolati i valori di **tutti** i sottovettori di V.
Quello che descriveremo ora è un algoritmo basato sulla tecnica del divide et impera che risolve il problema in $O(n \log n)$ tempo.
Qualora il vettore V abbia un unico elemento allora il problema è banalmente risolvibile (i.e. la soluzione è l'unico valore V[1]). Assumiamo quindi $n \geq 2$. In base allo schema della tecnica partizioniamo il vettore V in due sottovettori VS e VD di dimensioni $\lceil \frac{n}{2} \rceil$ e $\lfloor \frac{n}{2} \rfloor$ rispettivamente e risolviamo i due sottoproblemi così ottenuti. Siano $SOL1$ e $SOL2$ le soluzioni ai due sottoproblemi (i.e. $SOL1$ è il valore massimo per i sottovettori di VS mentre $SOL2$ è il valore massimo per i sottovettori di VD). Al fine di risolvere il problema non basta calcolare il valore $\max\{SOL1, SOL2\}$ in quanto, così facendo, si trascurerebbe il contributo che possono dare alla soluzione i valori di tutti quei sottovettori che hanno parte dei loro elementi in VS e parte in VD e nulla vieta che il sottovettore di valore massimo per V sia proprio tra questi. L'algoritmo deve quindi restituire $\max\{SOL1, SOL2, SOL3\}$, dove $SOL3$ è il valore massimo per quei sottovettori che contengono sia elementi di VS che elementi di VD. Allo scopo di calcolare $SOL3$ notiamo che $SOL3 = maxsin + maxdes$ dove $maxsin$ è il valore massimo per quei sottovettori di VS che terminano con l'elemento finale di VS mentre $maxdes$ è il valore massimo per quei sottovettori di VD che iniziano con l'elemento iniziale di VD. I valori degli $\lceil \frac{n}{2} \rceil$ sottovettori di VS che ci interessano possono essere calcolati facilmente scorrendo il vettore VS da destra verso sinistra, in modo analogo, scorrendo il vettore VD da sinistra verso destra possiamo calcolare gli $\lceil \frac{n}{2} \rceil$ valori dei sottovettori di VD.
Lo pseudo-codice dell'algoritmo appena descritto è il seguente:

L'algoritmo MAX-VALORE1, essendo ricorsivo, prende in input il vettore V degli interi e il primo e l'ultimo indice del sottovettore di V su cui vogliamo risolvere il problema. Quindi la prima chiamata sarà MAX-VALORE1(V, 1, n).

```
MAX-VALORE1
INPUT il vettore V con n interi, primo indice i e l'ultimo indice j
    IF i = j THEN SOL ← V[i]
    ELSE
        m ← ⌊(i+j)/2⌋
        SOL1 ← MAX-VALORE1(V, i, m)
        SOL2 ← MAX-VALORE1(V, m + 1, j)
        maxsin ← V[m]
        sin ← V[m]
        FOR k = m − 1 TO i (step −1) DO
            sin ← sin + V[k]
```

```
            IF sin > maxsin THEN maxsin ← sin
        ENDFOR
        maxdes ← V[m + 1]
        des ← V[m + 1]
        FOR k = m + 2 TO j DO
            des ← des + V[k]
            IF des > maxdes THEN maxdes ← des
        ENDFOR
        SOL3 ← maxsin + maxdes
        SOL ← max{SOL1, SOL2, SOL3}
    ENDIF
OUTPUT SOL.
```

Per calcolare il tempo $T(n)$ richiesto dall'algoritmo per risolvere un'istanza di dimensione n possiamo procedere come segue: il tempo $D(n)$ richiesto per partizionare l'istanza di dimensione n è $\Theta(1)$ mentre il tempo $C(n)$ necessario a ricombinare insieme le soluzioni ottenute è $\Theta(n)$. Di conseguenza:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n) & \text{altrimenti} \end{cases}$$

che risolta da

$$T(n) = \Theta(n \log n)$$

Per il problema in esame è in realtà noto un semplice algoritmo che lo risolve in $O(n)$ tempo, l'algoritmo usa una tecnica differente: la programmazione dinamica. Pensare di poter risolvere il problema senza esaminare tutti gli interi presenti nel vettore è ovviamente impossibile, $\Omega(n)$ è quindi un limite inferiore alla complessità temporale del problema. L'algoritmo basato sulla la programmazione dinamica è quindi ottimo.

3. Il Problema DISTANZA MINIMA TRA PUNTI

Descriveremo ora un altro problema che può essere risolto efficientemente con un algoritmo basato sullo stile di programmazione che stiamo descrivendo, dove il modo giusto di "fondere" le soluzioni dei sottoproblemi che vengono via-via generati richiede un certo ingegno.
Il problema che consideriamo è molto semplice da enunciare:

Dati n punti sul piano, determinare la coppia a distanza minima.

Cominciamo con un pò di notazione. Denoteremo l'insieme di punti con $P = \{p_1, p_2, \ldots p_n\}$ dove p_i ha coordinate (x_i, y_i) e per due punti p_i e p_j in P con $dist(p_i, p_j)$ indicheremo la distanza euclidea tra i due (vale a dire $dist(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$).

Nel seguito assumeremo che in P non ci sono due punti con la stessa x-coordinata o la stessa y-coordinata. Ciò renderà la trattazione più chiara ed è facile poi estendere l'algoritmo che descriveremo in modo da eliminare questa assunzione.
Un algoritmo di forza bruta calcola la distanza per **tutte** le coppie di punti e prende la minima. Per n punti ci sono $\binom{n}{2}$ coppie distinte e quindi $\Theta(n^2)$ distanze da calcolare. L'algoritmo in pseudo-codice è il seguente:

```
COPPIA-MIN
INPUT il vettore P con gli n punti.
    min ← +∞
    FOR i = 1 TO n − 1 DO
        FOR j = i + 1 TO n DO
            δ ← dist(p_i, p_j)
            IF δ < min THEN
                min ← δ
                SOL ← (p_i, p_j)
            ENDIF
        ENDFOR
    ENDFOR
```

OUTPUT SOL

Al fine di risolvere il problema, è veramente necessario calcolare le distanze per tutte le $\binom{n}{2}$ coppie di punti o esistono algoritmi più efficienti di COPPIA-MIN?

È istruttivo considerare per un attimo la versione modo-dimensionale del problema. Come possiamo trovare la coppia più vicina tra i punti su di una retta? Prima li ordiniamo per coordinata crescente e poi scorrendo la lista ordinata calcoliamo la distanza di ciascun punto con quello che lo segue nell'ordinamento. È facile vedere che una di queste distanze deve essere quella minima.

COPPIA-MIN-SU-RETTA
INPUT il vettore P con gli n punti.
  ordina gli elementi di P rispetto alla coordinata x sia $\{p_{j_1}, p_{j_2}, \dots p_{j_n}\}$ la sequenza ottenuta
  $min \leftarrow +\infty$
  FOR $i = 1$ TO $n-1$ DO
    $\delta \leftarrow \text{dist}(p_{j_i}, p_{j_{i+1}})$
    IF $\delta < min$ THEN
      $min \leftarrow \delta$
      $SOL \leftarrow (p_{j_i}, p_{j_{i+1}})$
    ENDIF
  ENDFOR
OUTPUT SOL

L' algoritmo COPPIA-MIN-SU-RETTA permette di risolvere il problema calcolando solo $\Theta(n)$ distanze con tempo di esecuzione che, utilizzando un efficiente algoritmo di ordinamento, è $O(n \log n)$.

Nelle due dimensioni potremmo cercare di ordinare i punti rispetto alla loro x-coordinata (o la loro y-coordinata) e sperare che i due punti più vicini risultino adiacenti l'uno all'altro in uno dei due ordinamenti. Tuttavia è facile costruire esempi in cui risultano al contrario molto lontani in entrambi gli ordinamenti, precludendo così la possibilità di adattare l'approccio mono-dimensionale al caso bidimensionale.

Descriveremo ora un algoritmo basato sulla tecnica divide-et-impera. L'idea è quella di trovare la coppia più vicina tra i punti nel "lato di sinistra" di P e la coppia più vicina tra i punti nel "lato di destra" di P e poi utilizzare queste informazioni per ottenere la soluzione in tempo lineare. Se sviluppiamo un algoritmo con questa struttura allora la complessità di tempo sarà data da $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$ che, come visto nell'esempio 2, ha come soluzione $O(n \log n)$,

Se l'insieme P è composto da soli 3 punti il problema viene risolto in modo esaustivo calcolando le distanze delle 3 distinte coppie di punti e restituendo quella minima. Assumiamo quindi che il numero n di punti sia superiore a 3. In base allo schema della tecnica divide-et impera partizioniamo l'insieme P in due sottoinsiemi P_S e P_D contenenti $\lceil \frac{n}{2} \rceil$ punti e $\lfloor \frac{n}{2} \rfloor$ punti, rispettivamente, con la proprietà che nessun punto di P_D sia alla sinistra dei punti di P_S. Risolviamo ora i due sottoproblemi e siano $SOL1$ e $SOL2$ le loro soluzioni(vale a dire $SOL1$ è la coppia a distanza minima per i punti in P_S e $SOL2$ è la coppia a distanza minima per i punti in P_D).
Sia ora $\delta = \min\{dist(SOL1), dist(SOL2)\}$, al fine di risolvere il problema per l'insieme P non è corretto restituire delle due coppie $SOL1$ e $SOL2$ quella i cui punti sono a distanza δ (così facendo, non si terrebbe conto delle distanze tra coppie di punti caratterizzate dall'avere un estremo in P_S e l'altro in P_D e proprio tra coppie di questo tipo.
Sia ora x^* la x-coordinata del punto più a destra in P_S ed L la linea verticale descritta dall'equazione $x = x^*$, questa linea "separa" P_S da P_D. Ovviamente una coppia con un estremo in P_S e l'altro in P_D può eventualmente avere distanza inferiore a δ solo se ciascuno dei suoi punti dista per meno di δ da L. Sia quindi Q il sottoinsieme dei punti in P che hanno coordinata x appartenente all'intervallo $]L-\delta, L+\delta[$. Per quanto detto, alla ricerca di eventuali coppie di punti di P a distanza inferiore a δ, basterà considerare le coppie di punti in Q. Indicando quindi con $SOL3$ la coppia a distanza minima per i punti di Q, delle tre coppie ottenute è soluzione al problema quella con distanza pari a $\min\{dist(SOL1), dist(SOL2), dist(SOL3)\}$. Lo pseudocodice di quanto appena descritto è

COPPIA-DISTANZA-MIN
INPUT un insieme P con n punti
  IF $n \leq 3$ THEN $SOL \leftarrow$ la coppia di punti a distanza minima (trovata esaustivamente)
  ELSE

$P_S \leftarrow$ gli $\lfloor \frac{n}{2} \rfloor$ punti di P più a sinistra nel piano
$P_D \leftarrow P - P_S$
$x^* \leftarrow$ la coordinata x del punto più a destra in P_S
$SOL1 \leftarrow$ DISTANZA-MIN(P_S)
$SOL2 \leftarrow$ DISTANZA-MIN(P_D)
$SOL \leftarrow$ la coppia a distanza minima tra le due coppie $SOL1$ e $SOL2$
$\delta \leftarrow dist(SOL)$
$Q \leftarrow$ i punti di P con coordinata x nell'intervallo $[x^* - \delta, x^* + \delta]$
IF $|Q| > 1$ THEN
    $SOL3 \leftarrow$ la coppia a distanza minima tra i punti in Q
    $SOL \leftarrow$ la coppia a distanza minima tra le due coppie $SOL3$ e SOL
ENDIF
ENDIF
OUTPUT SOL.

Se assumiamo i punti in P ordinati rispetto alla coordinata x, allora il tempo $D(n)$ richiesto per partizionare l'istanza è $O(1)$, basterà infatti richiamare il problema prima sui primi $\lfloor \frac{n}{2} \rfloor$ punti e poi sui rimanenti. Consideriamo quindi il tempo $C(n)$ richiesto per combinare le soluzioni. Questo tempo è dominato dal costo della ricerca del valore $SOL3$ nell'insieme Q. Ricercare la distanza minima in Q calcolando tutte le $\begin{pmatrix} |Q| \\ 2 \end{pmatrix}$ distanze per le coppie in Q determinerebbe un costo $O(|Q|^2)$ e poiché in Q potrebbero esserci punti dell'ordine di n. La complessità della procedura sarebbe data dalla ricorrenza

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n^2)$$

e quindi (vedi esempio 4) si avrebbe un tempo di calcolo $O(n^2)$ senza alcun vantaggio rispetto alla ricerca esaustiva.

Fortunatamente per la ricerca in Q di coppie di punti a distanza inferiore a δ non è affatto necessario controllare le distanze di tutte le $\begin{pmatrix} |Q| \\ 2 \end{pmatrix}$ coppie di punti come si deduce dalla seguente proprietà

Si assumano i punti di Q ordinati rispetto alla coordinata y, allora due punti di Q con distanza inferiore a δ distano al più 7 posizioni.

Dimostrazione. Sia $q = (q_x, q_y)$ un punto di Q e consideriamo la zona di piano individuata dal rettangolo di dimensione $2\delta \times \delta$ contenente punti con coordinata x in $[x^* - \delta, x^* + \delta]$ e coordinata y in $[q_y, q_y + \delta]$. Partizioniamo il rettangolo in 8 quadrati di ugual dimensione $\frac{\delta}{2} \times \frac{\delta}{2}$.
Supponiamo che due punti di Q siano situati all'interno di uno stesso quadrato. Poiché tutti i punti di uno stesso quadrato risiedono da una stessa parte della retta L questi due punti o apparterrebbero entrambi a P_S o entrambi a P_D e disterebbero al più $\sqrt{(\frac{\delta}{2})^2 + (\frac{\delta}{2})^2} = \frac{\delta}{\sqrt{2}} < \delta$. Il che contraddice la definizione di δ come distanza minima tra coppie di punti in P_S o in P_D. Quindi ciascun quadrato contiene al più un punto di Q.
Un qualsiasi punto p' in Q con coordinata y superiore a q_y al di fuori di questi 8 quadrati dista da p in verticale di almeno δ. quindi se anche tutti i quadrati contenessero un punto di Q qualsiasi punto p' che dista da p nell'ordinamento rispetto alla coordinata y più di 7 posizioni dista da p di almeno δ.

Grazie alla proprietà appena dimostrata, per la ricerca di punti in Q a distanza inferiore a δ, basterà ordinare i punti di Q per coordinata y non decrescente e poi, per ciascun punto, calcolare la distanza tra questo e i 7 che lo seguono nell'ordinamento, per un totale di $O(|Q|)$ confronti.
Stando così le cose, il tempo $C(n)$ richiesto per combinare le soluzioni è dato dal tempo richiesto per ordinare i punti in Q rispetto alla coordinata y, tempo che , con un efficiente algoritmo d'ordinamento, ammonta a $O(|Q| \log |Q|)$. La complessità della procedura è ora descritta dalla ricorrenza

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 3 \\ 2 \cdot T\left(\frac{n}{2}\right) + O(n \log n) & \text{altrimenti} \end{cases}$$

Che risolta (vedi esempio 3) dà $T(n) = O(n \log^2 n)$. Il tempo richiesto per ordinare preliminarmente rispetto alla coordinata x l'insieme di punti P è $O(n \log n)$. La complessità di tempo dell'implementazione appena descritta è quindi $O(n \log^2 n)$.

Veniamo ora a determinare un'implementazione dell'algoritmo che garantisca tempi di calcolo dell'ordine $O(n \log n)$. Un modo più efficiente di implementare l'algoritmo è di far uso di una procedura che, oltre a restituire la soluzione all'istanza del problema, si occupa anche di riordinare i punti dell'istanza rispetto alla coordinata y. Così facendo, dopo aver risolto i due sottoproblemi per P_S e P_D ho anche i punti dei due sottovettori $P[1, \ldots, \lceil \frac{n}{2} \rceil]$ e $P[\lceil \frac{n}{2} \rceil + 1, \ldots, n]$ ordinati rispetto alla coordinata y e posso efficientemente riordinare l'intero vettore P in tempo $O(n)$ di modo che, scorrendo poi i punti di P da sinistra a destra ricaviamo i punti di Q ordinati rispetto alla coordinata y.

Il tempo $D(n)$ richiesto per partizionare l'istanza di dimensione n è $\Theta(1)$ mentre il tempo $C(n)$ necessario a ricombinare le soluzioni è ora $O(n)$. Quindi il tempo di calcolo $T(n)$ richiesto dalla procedura per risolvere un istanza di dimensione n è dato dall' equazione di ricorrenza

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 3 \\ 2 \cdot T\left(\frac{n}{2}\right) + O(n) & \text{altrimenti} \end{cases}$$

Da cui $T(n) = O(n \log n)$.

Quello che segue è lo pseudocodice dell' implementazione appena descritta. DISTANZA-MIN1 prende in input il vettore P i cui punti sono stati preliminarmente ordinati rispetto alla coordinata x
L'algoritmo fa uso di una sottoprocedura FONDI. Scopo della chiamata a FONDI(P_S, P_D) è quello di riordinare i punti del sottovettore P rispetto alla coordinata y in tempo $O(n)$ sfruttando il fatto che i punti dei sottovettori P_S e P_D sono gia ordinati.

COPPIA-DISTANZA-MIN1
INPUT un vettore P con $n \geq 3$ punti
  IF $n \leq 3$ THEN
    riordina i punti rispetto alla coordinata y
    $SOL \leftarrow$ la coppia di punti a distanza minima (trovata esaustivamente)
  ELSE
    $P_S \leftarrow$ i primi $\lfloor \frac{n}{2} \rfloor$ punti di P
    $P_D \leftarrow$ gli ultimi $\lceil \frac{n}{2} \rceil$ punti di P
    $x^* \leftarrow$ la coordinata x dell'ultimo punto in P_S
    $SOL1 \leftarrow$ DISTANZA-MIN(P_S)
    $SOL2 \leftarrow$ DISTANZA-MIN(P_D)
    $SOL \leftarrow$ la coppia a distanza minima tra le due coppie $SOL1$ e $SOL2$
    $\delta \leftarrow dist(SOL)$
    $P \leftarrow$ FONDI(P_S, P_D)
    $Q \leftarrow$ i punti di P con coordinata x nell'intervallo $[x^* - \delta, x^* + \delta]$ (ottenuti scorrendo il vettore P)
    IF $|Q| > 1$ THEN
      $SOL3 \leftarrow$ la coppia a distanza minima tra i punti in Q che distano tra loro al più 7 posizioni
      $SOL \leftarrow$ la coppia a distanza minima tra le due coppie $SOL3$ e SOL
    ENDIF
  ENDIF
OUTPUT SOL.

4. Il problema della SELEZIONE

Si consideri il seguente problema noto come SELEZIONE:

> *Dato un insieme di n interi distinti ed un intero k, con* $1 \leq k \leq n$*, trovare l'elemento che occuperebbe la k-ma posizione se l'insieme fosse ordinato.*

Se $k = 1$ il problema si riduce alla ricerca dell'elemento minimo dell'insieme, mentre per $k = n$ quel che si vuole è l'elemento massimo. Per questi casi particolari il problema può essere banalmente risolto in $O(n)$ tempo scandendo gli elementi

dell'insieme. In generale il problema della SELEZIONE può essere facilmente risolto in $O(n \log n)$ tempo: infatti si possono ordinare gli elementi dell'insieme con un efficiente algoritmo d'ordinamento e poi selezionare quello che compare nella k-esima posizione.
Vedremo ora un algoritmo basato sulla tecnica del divide-et-impera. che risolve il problema in tempo $O(n)$.

Sia S un insieme di n numeri e definiamo *mediano* quel numero di S che finirebbe nella posizione di mezzo se i numeri venissero ordinati (per risolvere il caso di n pari dove non c'è "posizione di mezzo" più formalmente il mediano è definito come l'elemento che dopo l'ordinamento finisce nella posizione $\lceil \frac{n}{2} \rceil$). Consideriamo per cominciare il seguente algoritmo ricorsivo

SELEZIONA1
INPUT l'insieme S e l'intero k, con $1 \leq k \leq |S|$
  IF $|S| = 1$ THEN $SOL \leftarrow$ l'unico elemento in S
  ELSE
    scegli elemento x mediano di S
    $S1 \leftarrow$ gli elementi di S a valore inferiore ad x
    $S2 \leftarrow$ gli elementi di S a valore superiore ad x
    CASE
      $k \leq |S1|$: $SOL \leftarrow$ SELEZIONA1$(S1, k)$
      $k = |S1| + 1$: $SOL \leftarrow x$
      $k > |S1| + 1$: $SOL \leftarrow$ SELEZIONA1$(S2, k - (|S1| + 1))$
    ENDCASE
  ENDIF
OUTPUT SOL

Poichè ogni chiamata ricorsiva come minimo dimezza la dimensione del problema e il tempo necessario a ricombinare la soluzione ottenuta è $O(1)$ (un semplice assegnamento), qualora l'individuazione del mediano richiedesse $O(n)$, il tempo $D(n)$ richiesto per scomporre un'istanza di dimensione n sarebbe $O(n)$ (una semplice scansione dell'insieme S permette di costruire i due insiemi $S1$ ed $S2$), otterremmo la relazione di ricorrenza

$$T\left(\frac{n}{2}\right) + O(n)$$

che ha soluzione $O(n)$

Sperare di poter, in modo efficiente (vale a dire in tempo $O(n)$), individuare il il mediano dell'insieme è un compito forse troppo ambizioso. Rilassiamo le nostre pretese e cerchiamo di selezionare come perno della suddivisione qualcosa che risulti poi essere non troppo lontano dal mediano di S, ad esempio qualcosa che disti da esso al più $\frac{|S|}{4}$. In questo modo ogni chiamata ricorsiva sarebbe su di una frazione dell'istanza al più grande $\frac{3|S|}{4}$ e quindi la dimensione dell'istanza verrebbe ridotta di almeno $\frac{1}{4}$ ad ogni chiamata ricorsiva. Come ottenere efficientemente un perno con questa proprietà? L'idea qui è quella di cercare il perno non tra tutti gli elementi di S ma su un opportuno campione di questi. E come scegliere il campione? Facendo in modo che ciascuno degli elementi del campione risulti essere mediano di un certo numero di elementi di S. In pseudocodice l'algorimo è il seguente

SELEZIONA
INPUT l'insieme S e l'intero k, con $1 \leq k \leq |S|$
  IF $|S| \leq 60$ THEN
    ordina S
    $SOL \leftarrow$ il k-mo elemento di S
  ELSE
    ordina S a gruppi di 5 elementi a parte l'ultimo che potrebbe contenere un numero inferiore di elementi
    $A \leftarrow$ i mediani di ciascuno degli $\lceil \frac{n}{5} \rceil$ gruppi ordinati di S
    $m \leftarrow$ SELEZIONA$(A, \lceil \frac{n}{10} \rceil)$
    $S1 \leftarrow$ gli elementi di S a valore inferiore ad m
    $S2 \leftarrow$ gli elementi di S a valore superiore ad m
    CASE

$k \leq |S1|$: $SOL \leftarrow$ SELEZIONA$(S1, k)$
$k = |S1| + 1$: $SOL \leftarrow m$
$k > |S1| + 1$: $SOL \leftarrow$ SELEZIONA$(S2, k - (|S1| + 1))$
ENDCASE
ENDIF
OUTPUT SOL

Cominciamo col dimostrare che effettivamente l'elemento m scelto come perno dall'algoritmo ha la proprietà di generare sottoinsiemi $S1$ ed $S2$ di S di cardinalità limitata:

Gli insiemi $S1$ ed $S2$ costruiti dall'algoritmo a partire da un insieme S di cardinalità n hanno cardinalità inferiore a $\frac{3n}{4}$.

Dimostrazione. $\lceil \frac{n}{10} \rceil$ elementi in A hanno valore minore o uguale ad m, di questi almeno $\lceil \frac{n}{10} \rceil - 1$ sono mediani di gruppi di 5 elementi presenti in S, quindi in S ci sono almeno $3\left(\lceil \frac{n}{10} \rceil - 1\right) \geq \frac{3n}{10} - 3$ elementi a valore minore o uguale ad m. Otteniamo quindi che

$$|S2| \leq n - \left(\frac{3n}{10} - 3\right) = \frac{7n}{10} + 3 < \frac{3n}{4}$$

dove l'ultima diseguaglianza segue dal fatto che $n > 60$. Con analogo ragionamento si dimostra che $|S1| < \frac{3n}{4}$.

La complessità di tempo $T(n)$ dell'algoritmo può essere trovata risolvendo la seguente relazione di ricorrenza

$$T(n) < T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{3n}{4}\right) + O(n)$$

che può essere giustificata dalle seguenti considerazioni

- la costruzione dell'insieme A con gli $\lceil \frac{n}{5} \rceil$ mediani richiede $O(n)$ tempo (basta ordinare l'insieme S a gruppi di cinque e selezionare poi da S gli $\lceil \frac{n}{5} \rceil$ interi nelle posizioni $3 + 5 \cdot j$, con $0 \leq j < \lceil \frac{n}{5} \rceil$)
- il calcolo del mediano in A è effettuato eseguendo una chiamata ricorsiva della procedura su di un insieme di cardinalità $\lceil \frac{n}{5} \rceil$. Ciò richiede $T\left(\lceil \frac{n}{5} \rceil\right)$ tempo.
- la costruzione degli insiemi $S1$ ed $S2$ richiede $O(n)$ tempo (una semplice scansione di S permette di ottenere $S1$ ed $S2$).
- la chiamata ricorsiva che restituisce la soluzione viene eventualmente eseguita su uno degli insiemi $S1$ ed $S2$, insiemi che, per la proprietà prima dimostrata, hanno cardinalità limitata da $\frac{3n}{4}$. Ciò richiede $T\left(\frac{3n}{4}\right)$ tempo.

La relazione di ricorrenza appena individuata è tipicamente generata da algoritmi di tipo divide et impera che dividono il problema originario in due sottoproblemi in maniera non bilanciata. Possiamo considerare questa ricorrenza come un caso particolare della seguente relazione

$$T(n) = T(\alpha \cdot n) + T(\beta \cdot n) + cn \text{ con } \alpha + \beta < 1$$

mostreremo ora col procedimento informale basato sull'analisi dell'albero di ricorsione che tale relazione ha soluzione $O(n)$. Il fatto che la somma delle parti, $\alpha \cdot n + \beta \cdot n$, sia strettamente minore della dimensione n originaria giocherà un ruolo determinante nell'analisi.

La relazione $T(n) = T(\alpha \cdot n) + T(\beta \cdot n) + cn$ con $\alpha + \beta < 1$ ha soluzione $T(n) < \frac{cn}{1-(\alpha+\beta)} = O(n)$.

Dimostrazione. Consideriamo l'albero delle chiamate ricorsive generato dalla ricorrenza e mostrato in figura 4, analizziamo il suo costo per livelli. Al primo livello abbiamo un costo $(\alpha + \beta) \cdot n$, al secondo un costo $(\alpha + \beta)^2 \cdot n$ e così via. Il tempo di esecuzione totale è quindi

$$T(n) < c \cdot n + c \cdot (\alpha + \beta) \cdot n + c \cdot (\alpha + \beta)^2 \cdot n \ldots \leq c \cdot n \cdot \sum_{i=0}^{\infty} (\alpha + \beta)^i = c \cdot n \frac{1}{1 - (\alpha + \beta)}$$

dove l'ultimo passaggio segue dal fatto che $\alpha + \beta < 1$ e la serie geometrica $\sum_{i=0}^{\infty} x^i$, con $x < 1$, converge a $\frac{1}{1-x}$,

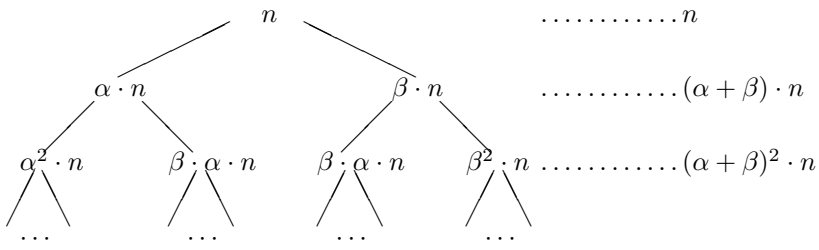

FIGURE 1. Albero di ricorsione corrispondente alla relazione $T(n) = T(\alpha \cdot n) + T(\beta \cdot n) + cn$.