Progettazione di algoritmi

Discussione dell'esercizio [salto]

Prima di tutto dimostriamo che un vettore V tale che $V[1] < V[n]$ ha almeno un salto. Questo è facile. Se non ci fossero salti, si avrebbe $V[1] \geq V[2] \geq \cdots \geq V[n-1] \geq V[n]$ (perché $1,2,\ldots,n-1$ non sarebbero salti) e quindi $V[1] \geq V[n]$ in contraddizione con l'ipotesi $V[1] < V[n]$.

Pensiamo ad un algoritmo Divide et Impera per trovare un salto di complessità $O(\log n)$. Naturalmente, la complessità $O(\log n)$ fa subito venire in mente la ricerca binaria. Cosa possiamo chiederci relativamente all'elemento centrale $V[n/2]$? Possiamo chiederci se è minore o maggiore di $V[1]$. Se risulta $V[1] < V[n/2]$, in virtù di quello che abbiamo appena dimostrato, sappiamo che esiste un salto nella prima metà del vettore e quindi possiamo limitarci a cercarlo lì. Se invece risulta $V[1] \geq V[n/2]$, allora siccome $V[1] < V[n]$, si ha che $V[n/2] < V[n]$ e quindi possiamo limitarci a cercare il salto nella seconda metà del vettore. In ogni caso scartiamo sempre una metà del vettore e continuiamo la ricerca nell'altra metà in cui siamo certi esiste almeno un salto. Ecco quindi una descrizione dell'algoritmo:

```
SALTO(V: array di interi, a,b: indici di V)
    IF b = a + 1 THEN
        RETURN a
    ELSE
        m = (a + b)/2   /* parte intera inferiore */
        IF V[a] < V[m] THEN
            RETURN SALTO(V, a, m)
        ELSE
            RETURN SALTO(V, m, b)
```

La prima chiamata sarà `SALTO(V, 1, n)` (si assume che $n \geq 2$ e $V[1] < V[n]$). La complessità è uguale a quella della ricerca binaria, cioè $O(\log n)$.

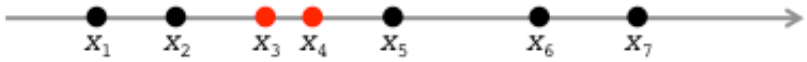
Il problema della coppia di punti più vicini

Il problema che vedremo adesso è uno dei primi problemi studiati, negli anni 70, dalla allora nascente branca della geometria computazionale. È molto semplice da enunciare:

> Dati n punti del piano, trovare la coppia di punti più vicini tra loro.

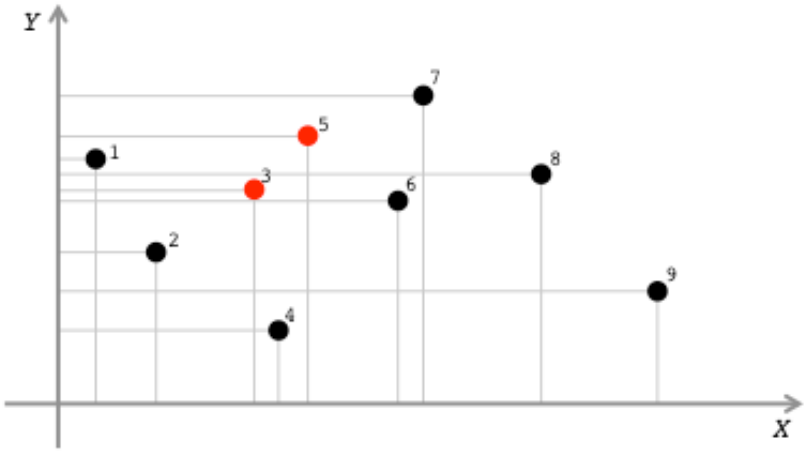
Nella letteratura è conosciuto con il nome di *Closest Pair Problem*. C'è un algoritmo molto semplice che lo risolve in tempo $O(n^2)$: calcola le distanze fra tutte le coppie di punti e ritorna la coppia con la minima distanza. Però non è efficiente. La tecnica Divide et Impera ci permetterà di derivare un algoritmo efficiente ma non sarà così facile. Per prendere un po' di dimestichezza con il problema consideriamo il caso più semplice in cui tutti i punti stanno

sull'asse delle *x*:



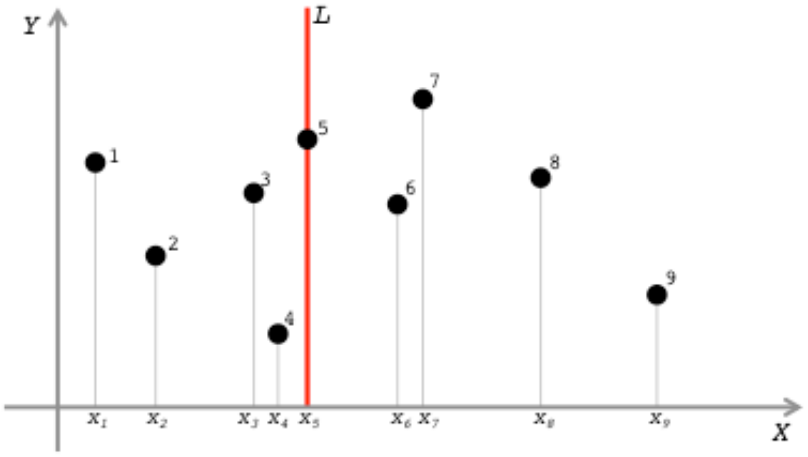
Ordinando i punti in base alla loro unica coordinata, la coppia di punti più vicini è la coppia di punti consecutivi, in tale ordinamento, che minimizza la differenza delle loro coordinate. Questo risolve il problema in tempo O(*n*log*n*).

Se i punti sono sul piano è immediato vedere che, in generale, non è possibile risolvere il problema semplicemente considerando la coppia di punti con le *x* o le *y* più vicine:

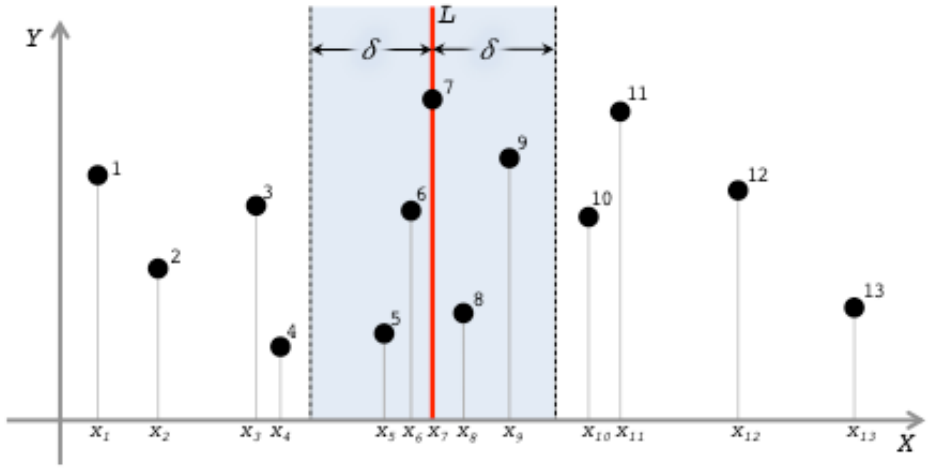


Per semplificare la trattazione assumeremo che tutti i punti abbiano coordinate *x* e *y* distinte e ci concentreremo solamente sul calcolo della distanza della coppia di punti più vicini. Una volta che l'algoritmo è stato descritto non è difficile apportare le modifiche sia per comprendere il caso generale, in cui punti differenti possono avere la stessa coordinata, sia per far sì che ritorni anche la coppia dei punti oltre alla loro distanza.

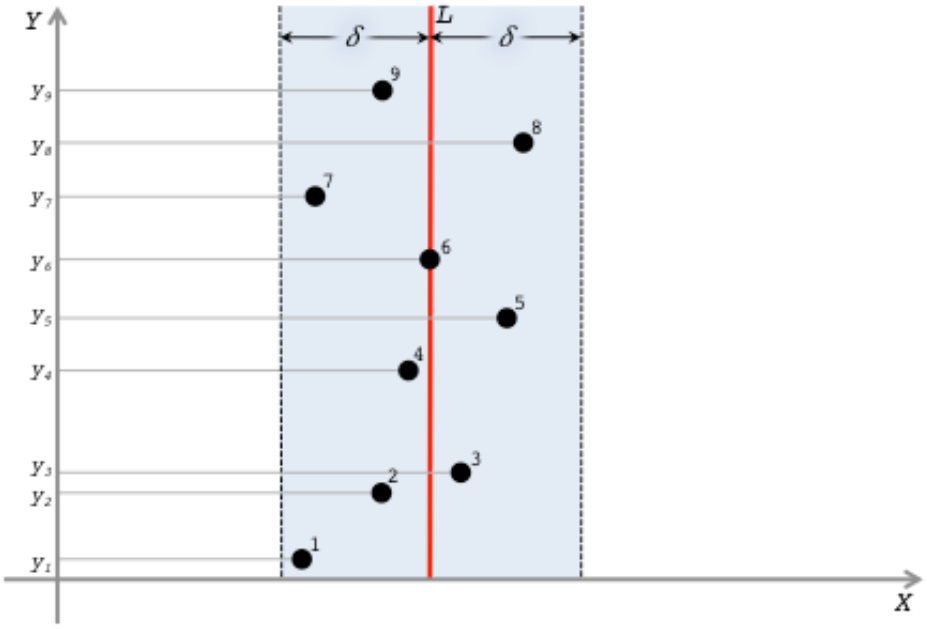
Volendo applicare la tecnica Divide et Impera, dividiamo in due parti l'insieme dei punti $P = \{p_1,\ldots,p_n\}$. Un modo semplice per farlo è tracciare una retta verticale *L* (cioè, parallela all'asse delle *y*) e considerare l'insieme dei punti P_S a sinistra di *L* e l'insieme dei punti P_D a destra di *L*. Per far sì che la divisione sia bilanciata, ordiniamo i punti in base alle *x*, cosicché $x_1 < x_2 < \cdots < x_n$, e scegliamo la coordinata *x* della retta *L* uguale a $x_{\lceil n/2 \rceil}$. Così i punti in P_S (cioè, i punti con la $x \leq x_{\lceil n/2 \rceil}$) sono $\lceil n/2 \rceil$ e quelli in P_D (cioè, i punti con la $x > x_{\lceil n/2 \rceil}$) sono $\lfloor n/2 \rfloor$.



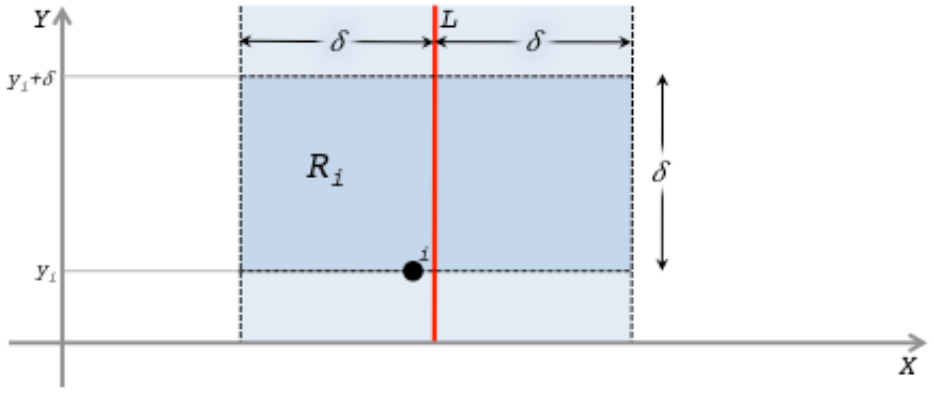
Se $n \leq 3$, possiamo risolvere il problema direttamente. Altrimenti, risolviamo il problema relativamente ad entrambe le parti P_S e P_D, ottenendo le corrispondenti distanze minime d_S e d_D. Sia $\delta = \min\{d_S, d_D\}$. In generale, δ non è la distanza minima dell'insieme di punti P perchè non abbiamo considerato le coppie di punti in cui uno è a sinistra di L e l'altro è a destra di L. Però, conoscendo δ, non è necessario considerare tutte le coppie di punti a cavallo di L. Infatti, se un punto p è a distanza da L maggiore o uguale a δ, una qualsiasi coppia di punti a cavallo di L che comprende p avrà distanza almeno δ. Quindi possiamo limitarci a considerare le coppie di punti p, p' tali che p è in P_S, p' è in P_D ed entrambi, p e p', non distano da L più di δ.



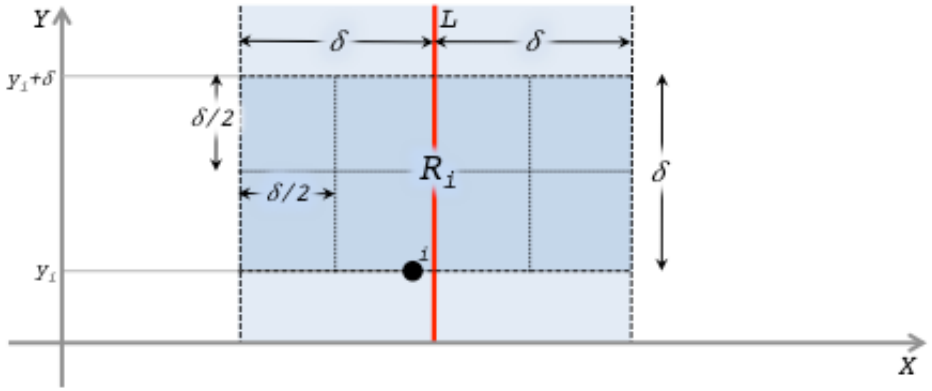
Sia quindi P_L l'insieme dei punti che si trovano a distanza inferiore a δ dalla retta L. Per trovare la distanza minima tra coppie di punti in P_L conviene che questi siano ordinati rispetto alle y. Siano allora $q_1,\ldots,q_k$ i punti in P_L ordinati rispetto alle y cosicché $y_1 < y_2 < \cdots < y_k$.



Chiaramente, per esaminare tutte le coppie in P_L possiamo per ogni punto q_i considerare le coppie (q_i, q_j) con $j > i$, anche se in questo modo considereremo anche coppie già esaminate perché contenute o in P_S o in P_D. Se ci fermiamo qui non avremo guadagnato nulla rispetto all'algoritmo semplice di complessità $O(n^2)$ perchè il numero di coppie da esaminare in P_L può essere dell'ordine di n^2. Ma c'è un'osservazione cruciale che possiamo fare. Quando esaminiamo le coppie (q_i, q_j), se $y_j \geq y_i + \delta$, è inutile calcolare la distanza della coppia perché siamo certi che risulterà maggiore od uguale a δ. Quindi a partire da q_i dobbiamo solamente considerare quei punti q_j, con $j > i$, tali che $y_j < y_i + \delta$. Sia allora R_i il rettangolo delimitato dalle rette parallele a L a distanza δ da L e dalle rette orizzontali con coordinate y uguali a y_i e $y_i + \delta$:



Sappiamo che due punti in P_L che si trovano o entrambi a sinistra di L o entrambi a destra di L sono a distanza maggiore o uguale a δ. Questo ci fa intuire che non ci possono essere troppi punti all'interno del rettangolo R_i. Infatti, partizioniamo R_i con una griglia di quadratini di lato δ/2:



Ogni quadratino può contenere al più un punto di P_L perché la massima distanza di due punti contenuti in un quadratino è uguale alla lunghezza della diagonale, cioè $\sqrt{2}(\delta/2) = \delta/\sqrt{2} < \delta$. Ne segue che il rettangolo R_i può contenere al più 8 punti compreso il punto q_i. Quindi le coppie che saranno esaminate a partire da q_i sono al più 7 e il numero totale di coppie che saranno esaminate in P_L è lineare in n. Ciò ci porterà sicuramente ad un algoritmo con complessità inferiore a $O(n^2)$. Ritorneremo sulla complessità dopo aver descritto con precisione l'algoritmo.

Ricapitoliamo l'algoritmo:

- Preliminarmente, ordina i punti di P rispetto alle x.

- Dividi i punti nelle due parti P_S e P_D
- Chiama ricorsivamente l'algoritmo con input P_S e con input P_D e calcola il minimo δ dei due output
- Determina l'insieme P_L e ordinalo rispetto alle y
- Esamina le coppie di P_L nel modo limitato sudescritto e ritorna il minimo tra le distanze esaminate e δ.

L'ordinamento iniziale prende tempo $O(n\log n)$. Il tempo $T(n)$ del resto dell'algoritmo soddisfa la relazione di ricorrenza:

$$T(n) = 2T(n/2) + O(n\log n)$$

dove il termine addittivo $O(n\log n)$ è dovuto all'ordinamento di P_L. Dal secondo caso del Master Theorem con $a = b = 2$ e $k = 1$, otteniamo che $T(n) = \Theta(n\log^2 n)$. Quindi la complessità

dell'algoritmo è $\Theta(n\log^2 n)$. Tuttavia è possibile migliorare ulteriormente l'algoritmo osservando che possiamo ordinare l'insieme P_L in modo ricorsivo. Ovvero, le chiamate ricorsive possono ritornare le parti P_S e P_D ordinate rispetto alle *y* e poi possiamo ordinare *P* facendo la fusione delle due parti già ordinate, come nel merge-sort. Ecco quindi lo pseudo-codice:

```
Ordina preliminarmente l'array dei punti P rispetto alle x

CP(P: array di n punti ordinati rispetto alle x)
    IF n <= 3 THEN
        Ordina P rispetto alle y
        RETURN minima distanza, calcolata esaustivamente
    ELSE
        m <- n/2            /* parte intera superiore */
        PS <- array dei punti di P da 1 a m
        PD <- array dei punti di P da m + 1 a n
        L <- P[m].x
        delta <- CP(PS)
        d <- CP(PD)
        IF d < delta THEN delta <- d
        P <- MERGE(PS, PD)  /* fonde le parti ordinate rispetto alle y */
        PL <- array dei punti in P con x in (L - delta, L + delta)
        k <- numero punti in PL
        FOR i <- 1 TO k DO
            j <- i + 1
            WHILE j <= k AND PL[j].y < PL[i].y + delta DO
                d <- dist(PL[j], PL[i])
                IF d < delta THEN delta <- d
                j <- j + 1
        RETURN delta
```

Il tempo di calcolo $T(n)$ di questa versione migliorata soddisfa

$$T(n) = 2T(n/2) + \mathrm{O}(n)$$

quindi la complessità è $\mathrm{O}(n\log n)$.

Il problema del calcolo della mediana

La *mediana* (o valore mediano) di una sequenza di valori è il valore centrale, cioè il valore *v* per cui ci sono almeno metà dei valori della sequenza minori o uguali a *v* e almeno metà valori della sequenza maggiori o uguali a *v*. La mediana è usata in statistica come riassunto di una distribuzione come la media, ma ha anche altri utilizzi. Ad esempio, nell'algoritmo sopra descritto per effettuare la divisione dei punti in due parti sarebbe stato sufficiente calcolare la mediana delle *x* dei punti.

Naturalmente, c'è un modo semplicissimo di calcolare la mediana di una sequenza di n numeri, basta ordinare la sequenza e prendere il valore centrale, cioè quello in posizione $\lceil n/2 \rceil$ (se n è pari potremmo anche prendere quello in posizione $\lceil n/2 \rceil + 1$). Questo algoritmo ha complessità $O(n \log n)$. Possiamo fare di meglio? Sappiamo calcolare i valori minimo e massimo di una sequenza in tempo lineare, perchè non potremmo fare lo stesso per la mediana? Intuitivamente il calcolo della mediana è più complicato. Ma, proviamo ad applicare l'approccio Divide et Impera al problema. Dovremmo cercare di dividere la sequenza o array in due parti, ma come? La mediana stessa permetterebbe di fare la divisione, ma è proprio ciò che vogliamo calcolare. Questo ci ricorda il quick-sort in cui l'array di input è diviso in due parti in modo tale che la prima parte contiene valori minori o uguali a un valore perno e la seconda valori maggiori o uguali al valore perno. In generale, il valore perno non è la mediana dell'array, anche se la mediana sarebbe il valore ottimale, quello cioè che divide l'array in due parti di uguale dimensione o quasi. Il quick-sort è molto efficiente nella pratica nonostante la divisione effettuata non è garantita essere ottimale. Forse una situazione simile vale anche per il calcolo della mediana. Proviamo a fare una divisione dell'array di input come nel quick-sort: scegliamo in modo "casuale" un valore dell'array, detto perno, e poi dividiamo l'array in una parte contenente valori minori o uguali al perno e l'altra di valori maggiori o uguali al perno. Però la presenza di valori uguali nell'array è importante per la mediana, conviene precisare meglio la divisione: dato un array A e un suo valore v, siano $A_<$, $A_=$ e $A_>$ array tali che

- $A_<$ contiene gli elementi x di A con $x < v$
- $A_=$ contiene gli elementi x di A con $x = v$
- $A_>$ contiene gli elementi x di A con $x > v$

Quindi ripartiamo tutti gli elementi di A nei tre array $A_<$, $A_=$ e $A_>$. Per ogni array X, sia len(X) il numero di elementi contenuti nell'array X. Sia $m = \lceil n/2 \rceil$, dove n = len(A). Se len($A_<$) $\geq m$, siamo certi che la mediana si trova nell'array $A_<$. Se invece len($A_<$) $< m$, siamo certi che la mediana si trova o in $A_=$ o in $A_>$, più precisamente se len($A_<$) + len($A_=$) $\geq m$ la mediana è in $A_=$ (cioè è uguale al valore perno v), altrimenti è in $A_>$. Però per cercare la mediana in $A_<$ come facciamo? La mediana di A non è uguale alla mediana di $A_<$. La mediana è l'elemento di $A_<$ in posizione m dopo che l'array $A_<$ è stato ordinato. Se invece dobbiamo cercarla in $A_>$, la mediana di A è l'elemento in posizione len($A_>$) - (n - m) dopo che $A_>$ è stato ordinato. Quindi per trovare la mediana dovremmo saper trovare, più in generale, il k-esimo valore di un array per un qualsiasi k con $1 \leq k \leq n$. Per k-esimo valore di un array intendiamo il valore che si trova in posizione k dopo che gli elementi dell'array sono stati ordinati. La mediana è un caso particolare: è l'm-esimo valore con $m = \lceil n/2 \rceil$. Questo però non crea problemi perchè quello che abbiamo detto riguardo alla mediana, cioè l'm-esimo valore, vale anche per un qualsiasi k. Possiamo allora definire il seguente algoritmo per trovare il k-esimo valore di un array:

```
KTH(A: array di n numeri, k: intero compreso tra 1 e n)
    Scegli in modo "casuale" un indice i compreso tra 1 e n
    v <- A[i]
```

```
    Dividi A in tre array:
        ALT <- gli elementi x di A con x < v
        AEQ <- gli elementi x di A con x = v
        AGT <- gli elementi x di A con x > v
    IF len(ALT) >= k THEN
        RETURN KTH(ALT, k)
    ELSE IF len(ALT) + len(AEQ) >= k THEN
        RETURN v
    ELSE
        RETURN KTH(AGT, len(AGT) - (n - k))
```

Chiaramente l'algoritmo per la mediana è

```
MEDIANA(A: array di n numeri)
    RETURN KTH(A, ⌈n/2⌉)
```

Il tempo per effettuare la divisione nei tre array è lineare in n similmente all'analoga procedura del quick-sort. L'analisi delle complessità al pari di quella del quick-sort dovrebbe essere fatta nel caso medio (ovvero valutando il tempo atteso) dato che l'algoritmo compie scelte "casuali". Per fare tale analisi in modo rigoroso è necessaria una dimestichezza con il calcolo delle probabilità che esula da questo corso. Un'argomentazione che si può fare ad un livello intuitivo è la seguente. Se indichiamo con $T(n)$ il tempo di calcolo dell'algoritmo `KTH` abbiamo che

$$T(n) = T(h) + \mathrm{O}(n)$$

dove h è la lunghezza dell'array sul quale viene effettuata la chiamata ricorsiva e dipende dalla scelta "casuale". Un caso buono è $h = n/2$ che in base al terzo caso del Master Theorem con $a = 1$, $b = 2$ e $c = 1$ ci dà $T(n) = \Theta(n)$. Ma anche se h è molto peggiore, diciamo $h = (99/100)n = n/(100/99)$, il Master Theorem ci dà sempre $T(n) = \Theta(n)$. Ciò ci fa intuire che anche se la scelta "casuale" non produce una divisione bilanciata, la complessità rimane lineare in n. Ovviamente nel caso peggiore, $h = n - 1$ e la complessità diventa $\Theta(n^2)$. Però questo accade con probabilità molto piccola e un'analisi rigorosa permette di dimostrare che il tempo di calcolo dell'algoritmo `KTH` è lineare in n con alta probabilità.

Esercizio [mediana]

Dati due array A e B di n interi, ordinati in senso non decrescente, descrivere un algoritmo che trovi la mediana dei $2n$ elementi in tempo $\mathrm{O}(\log n)$.