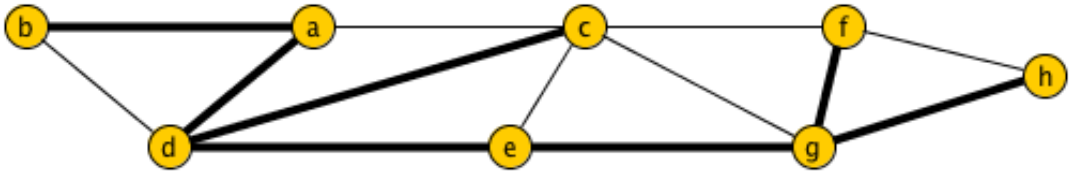
Progettazione di algoritmi - lezione 14

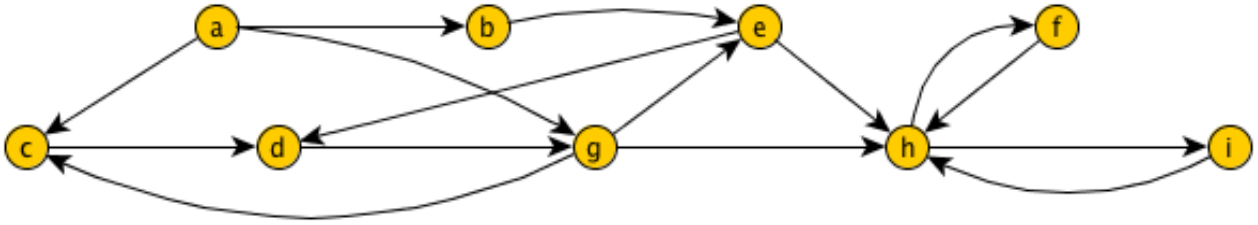
Discussione delle soluzioni della prova intermedia

Esercizio [DFS & BFS] (max 4) Si consideri il grafo *G* nella figura qui sotto e l'albero *T* formato dagli archi marcati. Partendo da quali nodi l'albero *T* può essere stato prodotto da una DFS e/o da una BFS? **Motivare bene la risposta**.



L'albero marcato non può essere stato prodotto da una BFS perchè tutti gli archi incidenti nel nodo di partenza devono appartenere all'albero è questo non si verifica per nessuno dei nodi. Vediamo la DFS nodo per nodo.
Partendo da ***a***: No, avrebbe visitato o *b* da *d* o *d* da *b*.
Partendo da ***b***,***c***,***d***,***e***: No, la visita arrivata a *g* avrebbe visitato o *h* da *f* o *f* da *h*.
Partendo da ***f***: No, la visita arrivata a *d* avrebbe visitato o *a* da *c* o *c* da *a*.
Partendo da ***g***: No, avrebbe visitato o *h* da *f* o *f* da *h*.
Partendo da ***h***: No, la visita arrivata a *d* avrebbe visitato o *a* da *c* o *c* da *a*.

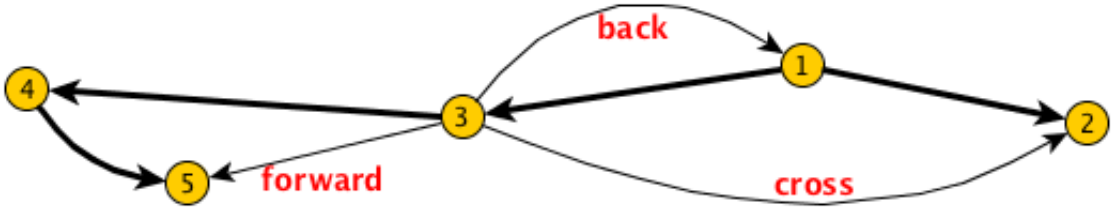
Esercizio [Tarjan] (max 4) Applicando l'algoritmo di Tarjan al grafo nella figura qui sotto, partendo dal nodo *a*, determinare le componenti fortemente connesse numerandole nell'ordine in cui sono trovate dall'algoritmo e per ognuna indicare la sua c-root.



Un possibile ordine con cui sono visitati i nodi dalla DFS dell'algoritmo di Tarjan è: *a*,*b*,*e*,*h*,*f*,*i*,*d*,*g*,*c*, Da questo si ottiene che le componenti fortemente connesse sono, nell'ordine: {***h***,*f*,*i*}, {***e***,*d*,*g*,*c*}, {***b***}, {***a***}; in grassetto sono indicate le c-root.

Esercizio [back-forward-cross] (max 4) Può esistere, in una visita DFS di un grafo diretto *G*, un nodo *v* che è incidente in almeno un arco all'indietro, in almeno un arco in avanti e in almeno un arco di attraversamento? In caso affermativo mostrare un esempio, altrimenti dimostrare l'impossibilità.

Sì può esistere, nel grafo qui sotto il nodo *v* è il nodo 3. I nodi sono numerati nell'ordine in cui sono visitati e gli archi dell'albero di visita sono marcati:



Esercizio [remove] (max 7) Dare lo pseudo-codice di un algoritmo che preso in input un grafo non diretto e connesso *G*, un suo nodo *u*, un vettore dei padri *P* relativo a una BFS da *u* in *G* e un arco {*v*, *w*} di *G*, ritorna *True* se e solo se la rimozione dell'arco {*v*, *w*} non cambia le distanze da *u*. L'algoritmo deve avere complessità O(*n*).

Se l'arco $\{v, w\}$ non appartiene all'albero della BFS, chiaramente la sua rimozione non cambia le distanze. Se invece fa parte dell'albero, allora o v è il figlio di w o w è il figlio di v nell'albero radicato in u dato dal vettore dei padri P. Nel primo caso, le distanze non cambiano se e solo se la distanza di v, dopo la rimozione di $\{v, w\}$, non aumenta e questo accade se e solo se esiste un adiacente $x \neq w$ di v tale che $distanza(x) < distanza(v)$. In modo simmetrico per l'altro caso. Per il calcolo efficiente delle distanze dal vettore dei padri si può usare l'algoritmo visto nella lezione 8:

```
DISTANZE(P: vettore dei padri)                    /* Ritorna l'array delle distanze */
    Dist: array delle distanze, inizializzato a -1
    FOR ogni nodo w DO
        IF Dist[w] = -1 THEN
            Dist[w] <- DIST(P, w, Dist)
    RETURN Dist

DIST(P: vettore dei padri, w: nodo, Dist: array)  /* Procedura di ausilio per DISTANZE */
    IF Dist[w] = -1 THEN
        IF P[w] = w THEN
            Dist[w] <- 0
        ELSE
            Dist[w] <- DIST(P, P[w], Dist) + 1
    RETURN Dist[w]
```

Ed ecco lo pseudo-codice dell'algoritmo:

```
DCHANGE(G: grafo, u: nodo, P: vettore padri, {v, w}: arco di G)
    IF P[v] ≠ w AND P[w] ≠ v THEN        /* {v, w} non fa parte dell'albero */
        RETURN True
    ELSE                                 /* {v, w} fa parte dell'albero */
        IF P[v] = w THEN z = v ELSE z = w
        Dist <- DISTANZE(P)              /* Calcola l'array delle distanze da P */
        FOR ogni adiacente x di z DO
            IF {x, z} ≠ {v, w} AND Dist[x] < Dist[z] THEN
                RETURN True
        RETURN False
```

La complessità di `DISTANZE` è $O(n)$ e la scansione degli adiacenti richiede al più $O(n)$, quindi la complessità dell'algoritmo è $O(n)$.

Esercizio [treni] (max 7) Consideriamo una rete ferroviaria relativa a *n* città che per comodità indichiamo con i numeri 1,2…,*n*. Tra ogni coppia (ordinata) *i*, *j* di città c'è al più un treno e questo impiega tempo t_{ij} per andare da *i* a *j*. Inoltre, per ogni città i c'è un tempo fisso di attesa A_i tra l'arrivo di un qualsiasi treno in *i* e la partenza di un qualsiasi treno da *i*. Dare lo pseudo-codice di un algoritmo che data la descrizione della rete ferroviaria (per ogni città *i* il suo tempo d'attesa A_i e i treni che partono da *i* con i relativi tempi t_{ij}), una città di partenza s e una città di destinazione d, calcola il minimo tempo per andare da s a d. L'algoritmo deve avere complessità $O((n + m)\log n)$ dove *m* è il numero di treni. Discutere la correttezza dell'algoritmo.

Possiamo modellare la rete ferroviaria con un grafo F diretto e pesato i cui nodi corrispondono alle città e vi è un arco da i a j se c'è un treno dalla città i alla città j, il peso dell'arco è $A_i + t_{ij}$. Nel grafo F i

cammini da s a d rappresentano tutti i possibili viaggi dalla città s alla città d e il peso di ognuno di essi è proprio uguale alla durata del viaggio, cioè la somma dei tempi dei treni e delle attese. Quindi, per calcolare il minimo tempo possiamo usare l'algoritmo di Dijkstra sul grafo F a partire dal nodo s. Basta modificare opportunamente lo pseudo-codice dell'implementazione dell'algoritmo di Dijkstra dato nella lezione 9:

```
TRENI(R: rete ferroviaria (n città e m treni), s: città di partenza, d: città di
destinazione)
    F <- il grafo costruito dalla rete ferroviaria R come descritto sopra
    Dist: array delle distanze, inizializzato a infinito
    Dist[s] <- 0
    H <- min-heap inizializzato con tutti i nodi di F e le priorità sono i valori di Dist
    v <- s
    WHILE H non è vuoto AND v ≠ d DO
        v <- H.get_min()       /* Preleva il nodo con distanza minima */
        FOR ogni adiacente w di v in F DO
            IF Dist[w] > Dist[v] + p(v, w) THEN
                Dist[w] <- Dist[v] + p(v, w)
                H.decrease(w)  /* Aggiorna l'heap a seguito del decremento */
    RETURN Dist[d]
```

La costruzione del grafo F (definito tramite liste di adiacenza) prende tempo $O(n + m)$ e l'algoritmo di Dijkstra, nell'implementazione data, ha complessità $O((n + m)\log n)$.

Esercizio [acqua] (max 8) Ci sono n case, indicate con gli interi $i = 1,2,\ldots,n$, ognuna delle quali necessita di una fornitura d'acqua. La costruzione di un pozzo nella casa i costa $p[i]$ e la costruzione di una tubazione fra le case i e j costa $c[i, j]$. La fornitura d'acqua per una casa o è un pozzo costruito nella casa o è un cammino di tubazioni dalla casa a qualche pozzo. Dare lo pseudo-codice di un algoritmo che determina le case in cui costruire i pozzi e le tubazioni da costruire per dare una fornitura d'acqua a tutte le case minimizzando il costo totale. L'algoritmo deve avere complessità $O(n^2)$. Discutere la correttezza dell'algoritmo.
Suggerimento: rappresentare il problema tramite un opportuno grafo pesato con n+1 nodi, il nodo in più rappresenta i pozzi…

Seguendo il suggerimento rappresentiamo il problema tramite un grafo A non diretto e pesato con $n+1$ nodi di cui n corrispondono alle case e un nodo z rappresentante i pozzi. Tra due nodi casa i e j c'è un arco di peso $c[i, j]$ e tra ogni nodo casa i e il nodo z c'è un arco di peso $p[i]$. Ogni soluzione ammissibile, che cioè garantisce la fornitura d'acqua ad ogni casa, è un sottoinsieme degli archi del grafo A tale che ogni nodo casa è connesso direttamente (con un arco, cioè la casa ha un pozzo) o indirettamente (con un cammino) al nodo z. Questo è garantito da un qualsiasi albero di copertura del grafo A. Quindi la soluzione al problema è un minimo albero di copertura di A. Siccome il grafo A e completo conviene usare l'implementazione dell'algoritmo di Prim che usa un array invece che un heap:

```
ACQUA(n: numero case, c: matrice costi tubazioni, p: array costi pozzi)
    T: vettore dei padri di dimensione n
    COST: array dei costi di dimensione n
    FOR i <- 1 TO n DO
        T[i] <- n+1                     /* n+1 rappresenta il nodo z */
        COST[i] <- p[i]
    REPEAT n volte DO
        imin <- -1
        FOR i <- 1 TO n DO              /* Trova il nodo di costo minimo */
            IF COST[i] ≠ -1 AND (imin = -1 OR COST[i] < COST[imin]) THEN
                imin <- i
        COST[imin] <- -1                /* I nodi estratti sono marcati con -1 */
        FOR i <- 1 TO n DO
```

```
            IF c[imin, i] < COST[i] THEN
                T[i] <- imin
                COST[i] <- c[imin, i]
    RETURN T
```

Le case i in cui bisogna costruire un pozzo sono quelle per cui $T[i] = n+1$. La Complessità è chiaramente $O(n^2)$.

Esercizio [museo] (max 8) In un museo c'è un lungo corridoio rettilineo in cui sono esposti n quadri nelle posizioni $0 \leq q_1 < q_2 < q_3 < \ldots < q_n$. Un custode può sorvegliare i quadri che si trovano ad al più distanza 5 dalla sua posizione. Vogliamo sorvegliare tutti i quadri con un numero minimo di custodi. Dare lo pseudo-codice di un algoritmo che trova le posizioni di un numero minimo di custodi che sorvegliano gli n quadri. La complessità dell'algoritmo deve essere $O(n)$. Dare la dimostrazione di correttezza dell'algoritmo proposto.

Un algoritmo greedy naturale consiste nel considerare i quadri in ordine di posizione crescente e se ci sono quadri non ancora sorvegliati, posizionare un nuovo custode in modo tale che l'estremo minore del suo intervallo di sorveglianza (di lunghezza 10) cada nella posizione del primo quadro non ancora sorvegliato:

```
MUSEO(q: array delle posizioni degli n quadri)
    POS <- lista vuota delle posizioni dei custodi
    i = 1
    WHILE i <= n DO
        p <- q[i]+5                        /* Posizione custode */
        WHILE i <= n AND q[i] <= p+5 DO    /* Trova il primo quadro non sorvegliato */
            i <- i + 1
        POS.append(p)
    RETURN POS
```

Ad ogni iterazione del `WHILE` esterno almeno un nuovo quadro viene sorvegliato e l'indice `i` è incrementato di almeno 1 nel `WHILE` interno. Ne segue che sono eseguite al più n iterazioni del `WHILE` esterno e la complessità totale è $O(n)$. Per dimostrare la correttezza seguiamo lo schema generale per gli algoritmi greedy. Sia POS_h la lista `POS` al passo h e sia k il numero di iterazioni del `WHILE` esterno. Per completare la dimostrazione di correttezza è sufficiente dimostrare:

Per ogni $h = 1, 2, \ldots, k$, esiste una soluzione ottima POS^* che estende POS_h.
Dimostrazione. Per $h = 0$, POS_0 è una lista vuota e quindi è estesa da una qualsiasi soluzione ottima. Supponiamo che sia vero per h e dimostriamolo per $h + 1$. Sia POS^* una soluzione ottima che per ipotesi induttiva estende POS_h. Se POS^* estende anche POS_{h+1}, abbiamo fatto. Altrimenti, la posizione p del custode scelta al passo $h + 1$ non è contenuta in POS^*. Il quadro i che è il primo a non essere sorvegliato al passo $h + 1$ deve essere sorvegliato da un qualche custode in POS^*. Sia p^* la posizione di un tale custode. Siccome $p = q[i] + 5$, deve essere $p^* \leq p$ (perchè altrimenti un custode in p^* non potrebbe sorvegliare il quadro i). Chiaramente p^* non può essere in POS_h perchè il quadro i non è sorvegliato dalle posizioni in POS_h. Allora, possiamo sostituire p^* con p in POS^* definendo $POS^@ = (POS^* - p^*) + p$. Tutti i quadri con posizione $\geq q[i]$ sorvegliati dalla posizione p^* in POS^* sono anche sorvegliati dalla posizione p. Quindi $POS^@$ è una soluzione ottima che estende POS_{h+1}.

Divide et Impera

Una delle prime tecniche algoritmiche che si incontrano è *Divide et Impera*. La si vede all'opera in algoritmi di ordinamento, come *quick-sort* e *merge-sort*, e nell'algoritmo di *ricerca binaria*. La tecnica quindi dovrebbe essere già abbastanza conosciuta ed anche per questo la trattazione sarà più breve rispetto a quella di altre tecniche.

La tecnica Divide et Impera cerca di "spezzare" l'istanza di un problema in istanze più piccole in modo tale che le soluzioni per queste sotto-istanze agevolino la costruzione della soluzione dell'istanza originale. Gli algoritmi che usano questa tecnica si prestano naturalmente ad essere implementati in modo ricorsivo. Ad esempio, ricordiamo che nell'algoritmo quick-sort l'array di input è diviso in due parti spostando nella prima parte tutti gli elementi minori di un elemento perno e nella seconda parte tutti quelli maggiori, poi l'algoritmo è chiamato ricorsivamente su entrambe le parti per ordinarle ottenendo così l'ordinamento dell'intero array. In questo caso, l'algoritmo fa un lavoro (lo spostamento degli elementi a seconda che siano minori o maggiori del perno) prima delle chiamate ricorsive. In altri casi, come nel merge-sort, viene fatto un lavoro dopo aver ottenuto le soluzioni delle sotto-istanze per costruire a partire da quest'ultime la soluzione dell'istanza di input. In questi algoritmi l'istanza di input è partizionata in due parti (in generale, in due o più parti) e l'algoritmo è poi applicato ricorsivamente su ognuna delle parti, ma ci sono casi, come la ricerca binaria, in cui è sufficiente conoscere la soluzione di una sola delle parti e quindi l'algoritmo è applicato ricorsivamente solo su una di queste.

Quali sono i vantaggi offerti dalla tecnica Divide et Impera? Di solito, quando la tecnica è applicabile, non è difficile vedere *come* applicarla. Questo perchè, tipicamente, è facile intuire come si potrebbe spezzare l'istanza. Inoltre, è abbastanza facile, una volta che un algoritmo di Divide et Impera è stato ideato, analizzarne sia la correttezza che la complessità. In particolare, la complessità di un algoritmo Divide et Impera, essendo un algoritmo ricorsivo, può essere determinata risolvendo un'opportuna relazione di ricorrenza relativa alla funzione $T(n)$ che dà il massimo tempo di calcolo dell'algoritmo su istanze di dimensione n. Dovrebbero essere già noti i metodi più comuni per risolvere tali ricorrenze, tuttavia ricordiamo uno degli strumenti più utili, il cosidetto *Master Theorem*, che risolve relazioni del seguente tipo:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \quad \text{dove } a \geqslant 1 \text{ e } b > 1$$

- Se $f(n) = \Theta(n^c)$ con $c < \log_b a$, allora

$$T(n) = \Theta(n^{\log_b a})$$

- Se $f(n) = \Theta(n^{\log_b a} \log^k n)$ per $k \geqslant 0$, allora

$$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$$

- Se $f(n) = \Theta(n^c)$ con $c > \log_b a$, allora

$$T(n) = \Theta(n^c)$$

Ad esempio, per il merge-sort la relazione di ricorrenza è

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

dove n è la dimensione dell'array di input. Dal Master Theorem si ottiene che $T(n) = \Theta(n \log n)$ (secondo caso con $a = b = 2$ e $k = 0$). Per la ricerca binaria la ricorrenza è

$$T(n) = T(n/2) + O(1)$$

e dal Master Theorem concludiamo che $T(n) = \Theta(\log n)$ (secondo caso con $a = 1$, $b = 2$, $k = 0$).

Il problema del massimo sottovettore

Consideriamo un problema che è una versione molto semplificata di un problema di analisi statistica di immagini che fu studiato negli anni '70 del secolo scorso. Il problema è conosciuto con il nome *Maximum subarray problem* e consiste nel trovare, dato un vettore A di numeri reali, un sottovettore (una sequenza di elementi consecutivi del vettore) la cui somma degli elementi è massima. Si assume che il sottovettore vuoto abbia somma zero. Il problema è interessante perché pur essendo molto semplice ci permetterà di vedere un'evoluzione di algoritmi via via più efficienti che passando per la tecnica Divide et Impera arriveranno fino alla Programmazione dinamica. Lo vedremo infatti anche quando parleremo di quest'ultima tecnica.

È chiaro che il problema è banale se i valori sono tutti nonnegativi o sono tutti nonpositivi. Il problema non è banale solo se il vettore di input ha valori sia positivi che negativi. Ecco un esempio:

Un algoritmo diretto per il problema consiste nel considerare le somme di tutti i possibili sottovettori e prenderne il massimo (per semplicità descriviamo l'algoritmo che ritorna solamente il massimo senza indicazione del sottovettore):

```
MS1(A: vettore di n numeri reali)
    max <- 0            /* Somma di un sottovettore vuoto */
    FOR i <- 1 TO n DO
        FOR j <- i TO n DO
            sum <- 0
            FOR k <- i TO j DO     /* Calcola la somma del sottovettore da i a j */
                sum <- sum + A[k]
            IF sum > max THEN max <- sum
    RETURN max
```

È molto inefficiente, i tre cicli `FOR` nidificati implicano che la complessità è $\Theta(n^3)$. Possiamo migliorarlo osservando che la somma di un sottovettore `[i…j]` è uguale alla somma del sottovettore `[i…j-1]` più `A[j]`. Quindi l'algoritmo migliorato è:

```
MS2(A: vettore di n numeri reali)
    max <- 0
    FOR i <- 1 TO n DO
        sum <- 0
        FOR j <- i TO n DO
            sum <- sum + A[j]    /* Somma del sottovettore da i a j */
            IF sum > max THEN max <- sum
    RETURN max
```

La complessità di `MS2` è $\Theta(n^2)$. Possiamo fare di meglio? Cerchiamo di applicare la tecnica Divide et Impera. La

prima cosa che viene in mente è di dividere a metà il vettore, trovare il massimo in entrambe le metà e poi calcolare il massimo delle somme dei sottovettori a cavallo delle due metà. Alla fine ritorniamo il massimo dei tre massimi. Sicuramente l'algoritmo è corretto perché considera tutti i possibili sottovettori. Infatti, un qualsiasi sottovettore o è contenuto nella prima metà del vettore di input o nella seconda o è cavallo delle due. Ma, è più efficiente di `MS2` ? La risposta dipende da come effettuiamo il calcolo delle somme dei sottovettori a cavallo. Se lo facciamo in modo diretto cioè, per ogni coppia di indici `i` , `j` con `i` nella prima metà e `j` nella seconda, ci calcoliamo la somma del sottovettore `[i…j]` , questo richiede tempo almeno $O(n^2)$, anche se usiamo la tecnica dell'algoritmo `MS2` dato che il numero di sottovettori a cavallo è dell'ordine di n^2. Ovviamente questo implica che il nostro algoritmo Divide et Impera avrebbe complessità $O(n^2)$ (terzo caso del Master Theorem con $a = b = 2$ e $c = 2$).

Possiamo migliorare il calcolo del massimo delle somme dei sottovettori a cavallo delle due metà? Osserviamo che ogni sottovettore a cavallo è formato da una parte nella prima metà del vettore, che chiamiamo il *prefisso*, e un'altra nella seconda metà, che chiamiamo il *suffisso*:

Un sottovettore di somma massima è tale che il suo prefisso ha somma massima tra quelle di tutti i prefissi e il suo suffisso ha somma massima tra quelle di tutti i suffissi. Quindi basterà calcolare la somma massima dei prefissi e quella dei suffissi, la somma delle due sarà la somma massima tra tutti i sottovettori a cavallo.

```
MS3(A: vettore di numeri reali, a,b: indici di A)
    IF a = b THEN
        IF A[a] > 0 THEN RETURN A[a]
        ELSE RETURN 0
    ELSE
        m <- (a + b)/2       /* Parte intera inferiore */
        max1 <- MS3(A, a, m)
        max2 <- MS3(A, m+1, b)
        pre <- 0
        sum <- 0
        FOR i <- m DOWNTO a DO     /* Calcola la massima somma dei prefissi */
            sum <- sum + A[i]
            IF sum > pre THEN pre <- sum
        suf <- 0
        sum <- 0
        FOR i <- m+1 TO b DO       /* Calcola la massima somma dei suffissi */
            sum <- sum + A[j]
            IF sum > suf THEN suf <- sum
        m <- max(max1, max2, pre + suf)
        RETURN m
```

La prima chiamata sarà `MS3(A, 1, n)` . La relazione di ricorrenza per il tempo di calcolo di `MS3` è $T(n) = 2T(n/2) + O(n)$, infatti ora il calcolo del sottovettore massimo tra quelli a cavallo richiede solamente tempo lineare. Il secondo caso del Master Theorem con $a = b = 2$ e $k = 0$, ci dice che la complessità dell'algoritmo basato sulla tecnica Divide et Impera è $\Theta(n\log n)$. Un bel miglioramento rispetto a $\Theta(n^2)$ di `MS2` .

Esercizio svolto

Dare lo pseudo-codice di un algoritmo che preso in input un array ordinato di n interi A e un intero x, calcola il numero di occorrenze di x in A in tempo $O(\log n)$.

Esercizio [salto]

In un vettore V di n interi, chiamiamo *salto* un indice i, $1 \leq i < n$, tale che $V[i] < V[i+1]$.

- Dato un vettore V di $n \geq 2$ interi tale che $V[1] < V[n]$, provare che V ha almeno un salto.
- Descrivere un algoritmo che, dato un vettore V di $n \geq 2$ interi tale che $V[1] < V[n]$, trovia un salto in tempo $O(\log n)$.