

Esercizio A1

Il costo della prima iterazione del ciclo che chiama InsertionSort varia a seconda dell'input:

- Nel caso il vettore di input sia già ordinato, il costo è pari a $O(n)$.
- Nel caso medio/pessimo, il costo è pari a $O(n^2)$.

In tutte le iterazioni successive, viene alterato un singolo valore che si può ritrovare fuori ordine; nel caso pessimo, il costo di tale ordinamento sarà $O(n)$. Quindi le $n - 1$ iterazioni successive costano in totale $O(n^2)$.

Il costo della parte non ricorsiva è quindi $O(n^2)$.

Il vettore viene quindi diviso in quattro parti approssimativamente uguali, su ognuna delle quali verrà chiamato ricorsivamente l'algoritmo.

Il costo è quindi approssimato dalla seguente ricorrenza:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 4T(\lfloor n/4 \rfloor) + O(n^2) & n > 1 \end{cases}$$

Utilizzando il Master Theorem versione semplice, si ottiene $T(n) = \Theta(n^2)$.

Esercizio A2

Molte soluzioni proposte si basano su un approccio "ingordo", greedy, che è argomento della seconda parte. Si parte dalla casella 1. L'idea è quella di muoversi nella casella che è più vicina alla posizione n e contiene lo stesso valore della casella corrente. Però questo meccanismo non funziona in alcuni casi. Si consideri il vettore seguente:

1	2	3	4	5	6	7	8	9	10	11	12
1	3	4	2	1	5	1	6	7	8	9	2

Il percorso più breve consiste negli indici 1-5-4-12; è quindi possibile raggiungere $n = 12$ in tre passi. Se invece si sceglie il valore 1 più vicino alla posizione $n = 12$, il percorso che si ottiene è 1-7-8-9-10-11-12, che richiede 6 passi.

Esistono almeno tre possibili soluzioni. Quella di complessità $O(n^2)$ era quella che mi aspettavo di vedere; prende 9 punti come previsto. Ne esiste una $O(n \log n)$ che prende 10 punti. Poi ne esiste una $O(n)$, abbastanza lunga da descrivere, non proposta da nessuno.

Versione $O(n^2)$ È possibile trasformare il problema in un problema su grafi e applicare una visita BFS per calcolare le distanze. Per ogni nodo $i < n$, si aggiunge un arco non orientato $(i, i + 1)$. Dopo di che, si confronta il valore contenuto in $A[i]$ con tutti i valori $A[j]$ successivi al successore, e in caso di uguaglianza fra i e j si aggiunge un arco non orientato (i, j) .

Notate che $m = n - 1 + O(n^2)$, quindi sia la costruzione del grafo che la sua visita richiede $O(n^2)$.

```

int minSteps(int[] A, int n)
    GRAPH G = Graph()
    for i = 1 to n do
        G.insertNode(i)
    for i = 1 to n - 1 do
        G.insertEdge(i, i + 1)
        G.insertEdge(i + 1, i)
        for j = i + 2 to n do
            if A[i] == A[j] then
                G.insertEdge(i, j)
                G.insertEdge(j, i)

    int[] dist = new int[1...n] = {-1}                                % Initialized to -1
    int[] parent = new int[1...n]
    erdos(G, 1, dist, parent)
    return dist[n]

```

Versione $O(n \log n)$ Si costruisce un grafo contenente n nodi celle, uno per ogni cella, e 10 nodi valore, uno per ogni valore. Due nodi celle sono uniti da un arco non orientato con peso 1 se sono vicine nel vettore; un nodo cella è unito da un arco non orientato con peso 0.5 ad un nodo valore se la cella contiene tale valore.

È possibile costruire tale grafo in tempo $O(n)$; applicando l'algoritmo di Dijkstra (argomento della seconda parte), è possibile ottenere le distanze pesate minime in tempo $O(n \log n)$. Il codice è lasciato per esercizio.

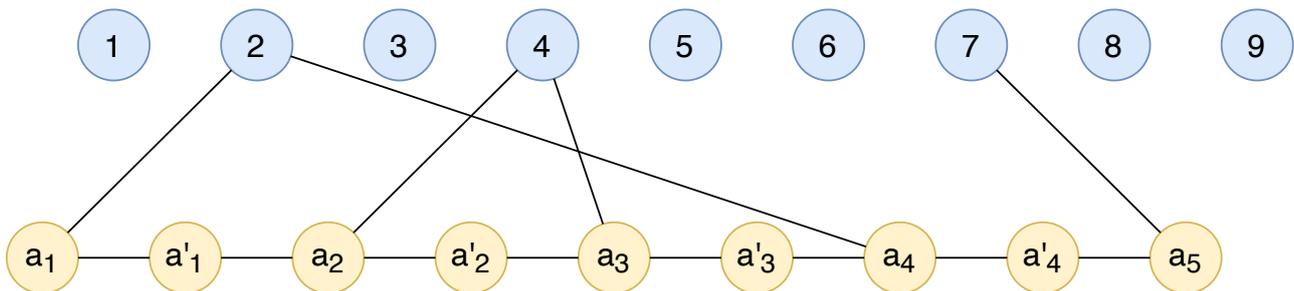
Versione $O(n)$ L'idea è simile a quella precedente, ovvero basata su un grafo non orientato. Si costruiscono nodi celle a_1, \dots, a_n , nodi di passaggio a'_1, \dots, a'_n e nodi "valore" $2n + 1, \dots, 2n + 9$. Si collega ogni nodo a_i con il nodo valore $A[i]$. Quindi, passare da un nodo a_i ad un nodo a_j , se $A[i] = A[j]$, richiederà di passare per due archi. Per questo motivo, si collegheranno i nodi a_i e a_{i+1} con una catena di archi $a_i \leftrightarrow a'_i \leftrightarrow a_{i+1}$, dove a'_i è un nodo fittizio che fa sì che il numero di archi da attraversare per andare da una cella a quella successiva sia 2 . A questo punto, si può utilizzare una visita BFS per calcolare il percorso più breve, passando da una cella a quella precedente/successiva oppure passando attraverso un nodo un valore. La lunghezza del percorso trovato sarà il doppio del valore da restituire. Il numero di nodi è pari a $2n - 1 + 9 = 2n + 8$ (ogni cella è rappresentata da due nodi tranne l'ultima, più 9 nodi valore), il numero di archi è pari a $2n - 2 + n = 3n - 2$, e quindi il costo della visita è $\Theta(n)$.

```

int minSteps(int[] A, int n)
    GRAPH G = Graph()
    for i = 1 to n do
        G.insertNode(2i - 1)                                % Add cell nodes, ai
        if i ≠ n then
            G.insertNode(2i)                                % Add cell nodes, a'i
    for i = 1 to 9 do
        G.insertNode(2n - 1 + i)                            % Add value nodes, 2n ... 2n + 8
    for i = 1 to n do
        G.insertEdge(2i - 1, A[i])                          % Edge (ai, A[i])
        G.insertEdge(A[i], 2i - 1)                          % Edge (A[i], ai)
    for i = 1 to n - 1 do
        G.insertEdge(2i - 1, 2i)                            % Edge (ai, a'i)
        G.insertEdge(2i, 2i - 1)                            % Edge (a'i, ai)
        G.insertEdge(2i, 2i + 1)                            % Edge (a'i, ai+1)
        G.insertEdge(2i + 1, 2i)                            % Edge (ai+1, a'i)
    int[] dist = new int[1 ... 2n + 9] = {-1}              % Initialized to -1
    int[] parent = new int[1 ... 2n + 9]
    erdos(G, 1, dist, parent)
    return dist[2n - 1]/2

```

La figura seguente mostra il grafo generato per il vettore $A = [2, 4, 4, 2, 7]$.



Esercizio A3

Il problema può essere risolto tramite la tecnica divide-et-impera. Inizialmente, passiamo come indici gli estremi del vettore B (più grande). Sia m l'indice mediano del sottovettore considerato. Si possono fare le seguenti osservazioni:

- Se $A[m] = B[m]$, tutti gli elementi compresi fra l'inizio del vettore ed m corrispondono nei due vettori. Quindi l'intruso si troverà a destra del mediano, mediano escluso;
- se $A[m] > B[m]$, il valore si trova sicuramente a sinistra del mediano, mediano incluso;
- non può essere che $A[m] < B[m]$;
- se la dimensione del vettore B si riduce ad 1, abbiamo individuato l'intruso.

Essendo un algoritmo di ricerca dicotomica, la complessità è pari a $O(\log n)$.

```

int intruder(int[] A, int[] B, int n)
    return intruderRec(A, B, 1, n + 1)

```

```
int intruderRec(int[] A, int[] B, int i, int j)
```

```
if i == j then
|   return B[i]
else
|   m =  $\lfloor (i + j) / 2 \rfloor$ 
|   if A[m] == B[m] then
|   |   return intruderRec(A, B, m + 1, j)
|   else
|   |   return intruderRec(A, B, i, m)
```

Esercizio B1

Questo problema non è altro che una versione ri-elaborata del problema dell'Insieme Indipendente di Intervalli. È sufficiente richiamare la funzione `independentSet()` presentata a lezione e restituire la dimensione dell'insieme così ottenuto. La complessità è $\Theta(n \log n)$, in quanto l'insieme di intervalli va ordinato per tempo di fine.

```
int longestChain(int[] X, int[] Y, int n)
```

```
SETS = independentSet(X, Y, n)
return S.size()
```

Esercizio B2

Sia $DP[i]$ il valore del più grande sottovettore di lunghezza pari che termina in posizione i . È possibile fare le seguenti osservazioni:

- Non esistono sottovettori di dimensione pari che terminano in posizione $i = 1$; quindi il valore è pari a 0.
- Nel caso di $i = 0$, non ho elementi, quindi il valore è di nuovo pari a zero.
- Nel caso generale, prendo il valore del più grande sottovettore che termina nella posizione $i - 2$ e sommo i valori $A[i - 1]$, $A[i]$. Se il risultato è positivo, vuol dire che posso estendere il sottovettore che termina in $i - 2$ con altri due elementi e il valore risultante è il più grande sottovettore che termina in posizione i ; se è negativo, il più grande sottovettore che termina in i è vuoto e il suo valore è 0.

Questo conduce alla seguente formulazione ricorsiva:

$$DP[i] = \begin{cases} 0 & i \leq 1 \\ \max\{DP[i - 2] + A[i - 1] + A[i], 0\} & i > 1 \end{cases}$$

A questo punto, il valore cercato è il valore massimo contenuto nella tabella DP – il più grande sottovettore che termina in qualsiasi indice del vettore.

Utilizziamo programmazione dinamica per tradurre questo formulazione ricorsiva in codice.

```
int maxSumEven(int[] A, int n)
```

```
int[] DP = new int[0...n]
DP[0] = DP[1] = 0
for i = 2 to n do
|   DP[i] = max(DP[i - 2] + A[i - 1] + A[i], 0)
return max(DP, n)
```

La complessità della soluzione è ovviamente lineare – $\theta(n)$.

Esercizio B3

Questo problema si risolve tramite backtrack - in effetti, non è altro che un problema di cammino più lungo su grafi, la cui versione decisionale è NP-completa.

Utilizziamo la matrice M sia come input, sia per memorizzare il fatto che una certa casella 1 è già stata visita, cambiando il valore associato da 1 a -1 . Si effettua una visita in profondità, aumentando di 1 la lunghezza ℓ del percorso trovato ad ogni passo successivo e confrontandolo con la variabile *maxSoFar*, passata per riferimento (è possibile risolvere il problema senza passare la variabile per riferimento, ma al prezzo di rendere l'algoritmo molto più difficile da leggere).

Un ulteriore trucco è quello di chiamare comunque la funzione ricorsivamente sulle quattro celle adiacenti, verificando, come primo step della chiamata, che la cella sia valida (i compreso fra 1 ed n , j compreso fra 1 e m) e che contenga un valore 1.

```
int longestPath(int[][] M, int n, int m)
```

```
    return longestPathRec(M, m, n, 1, 1)
```

```
int longestPathRec(int[][] M, int n, int m, int i, int j)
```

```
    if 1 ≤ i ≤ n and 1 ≤ j ≤ m and M[i][j] == 1 then
```

```
        M[i][j] = -1
```

```
        % Interpreted as "visited"
```

```
        int max = 1 + max(    longestPathRec(M, n, m, i + 1, j),
                            longestPathRec(M, n, m, i, j + 1),
                            longestPathRec(M, n, m, i - 1, j),
                            longestPathRec(M, n, m, i, j - 1) )
```

```
        M[i][j] = 1
```

```
        % Not "visited" any more
```

```
        return max
```

```
    else
```

```
        return 0
```

La complessità dell'algoritmo è molto alta, $O(4^{nm})$.