Algoritmi e Strutture Dati - 22/08/2019

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 due possibili soluzioni, una di complessità $O(n^2)$ molto semplice, e una di complessità $O(n)$ che non mi aspettavo di vedere; e invece, uno degli studenti l'ha proposta. Inoltre, uno studente ha proposto una soluzione $O(n \log n)$.
Per questo motivo ho innalzato il voto massimo a 10 e assegnato 90% a chi ha dato una soluzione $O(n^2)$, in modo da assegnare i 9 punti previsti. A chi è riuscito a fare meglio, ho assegnato 100% (10 punti).

Versione $O(n^2)$ E' 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 contenenuto 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 $O(n)$ Nella soluzione precedente, non viene sfruttato il fatto che l'insieme dei possibili valori è limitato ai valori $0 \dots 9$.
Il fatto che due valori $A[i]$, $A[i+1$ siano "vicini" nel vettore, significa che è possibile andare dal valore $A[i]$ al valore $A[i+1]$ in un passo. In quanti passi è possibile andare dal valore $A[1]$ al valore $A[n]$?
In altre parole, è possibile trasformare il problema in un grafo composto da 10 nodi, dove due nodi x e y sono collegati da un arco non orientato se esiste un indice i tale che $A[i] = x$ e $A[i+1] = y$. Si compie quindi una visita BFS su tale grafo, calcolando le distanze dal nodo $A[1]$ e restituendo le distanze del nodo $A[n]$.

```
int minSteps(int[] A, int n)
    GRAPH G = Graph()
    for i = 0 to 9 do
        G.insertNode(i)
    for i = 1 to n − 1 do
        G.insertEdge(A[i], A[i + 1])
        G.insertEdge(A[i + 1], A[i])
    int[] dist = new int[0 . . . 9] = {−1}        % Initialized to −1
    int[] parent = new int[0 . . . 9]
    erdos(G, A[1], dist, parent)
    return dist[A[n]]
```

Assumendo che il grafo sia implementato tramite una matrice di adiacenza (per evitare il problema che lo stesso arco possa essere inserito più volte), il costo dell'algoritmo é $O(n)$: (n) inserimenti di archi con costo $O(1)$ più una visita BFS in un grafo con 10 nodi (che si realizza in 100 passi = $O(1)$). Il costo dell'algoritmo è quindi $O(n)$.

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.
E' 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.

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 = ⌊(i + j)/2⌋
        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. E' 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. E' 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, $O(4^{nm})$.