Algoritmi e Strutture Dati – 27/07/2021

Esercizio A1 – Complessità – Punti ≥ 8

È facile vedere che la funzione è $\Omega(n^2)$, per via della parte non ricorsiva. Proviamo quindi a vedere se la funzione è anche $O(n^2)$.

- **Caso base**: per $n = 1, 2, 3$, si deve dimostrare che $T(n) = 1 \leq cn^2$; questa disequazione è vera per qualunque $c \geq 1, c \geq 1/4, c \geq 1/9$ rispettivamente; è quindi vera per ogni $c \geq 1$.
- **Ipotesi induttiva**: ipotizziamo che per ogni valore k, $1 \leq k < n$: $T(k) \leq k^2$;
- **Passo induttivo**: consideriamo tutti i casi con $n \geq 4$, tali per cui $\lfloor n/2 \rfloor$ e $\lfloor n/4 \rfloor$ sono maggiori o uguali a 1 (il primo caso base); applichiamo la sostituzione e otteniamo:

$$
\begin{aligned}
T(n) &= 3T(\lfloor n/2 \rfloor) + 3T(\lfloor n/4 \rfloor) + n^2 & \\
&\leq 3c\lfloor n/2 \rfloor^2 + 3c\lfloor n/4 \rfloor^2 + n^2 & \text{Sostituzione} \\
&\leq 3cn^2/4 + 3cn^2/16 + n^2 & \text{Eliminazione } \lfloor\ \rfloor \\
&= \frac{15}{16}cn^2 + n^2 \overset{?}{\leq} cn^2 & \text{Passaggio algebrico}
\end{aligned}
$$

L'ultima disequazione è vera per $c \geq 16$.

Abbiamo quindi dimostrato che $T(n) = O(n^2)$, per $m = 1$ e $c = 16$. Poiché $T(n) = \Omega(n^2)$, abbiamo che $T(n^2)$ è $\Theta(n^2)$.

Esercizio A2 – Vettori k-ordinati – Punti ≥ 10

Una soluzione banale è ovviamente ordinare usando un algoritmo come MergeSort, che ha costo $\Theta(n \log n)$, una soluzione che non viene considerata.
È possibile vedere che una semplice applicazione di InsertionSort ha costo $O(nk)$. Infatti, quando InsertionSort incontra un valore, inizia a spostarlo indietro fino a quando non incontra un valore più piccolo. Poiché ogni valore può essere al più k posizioni in avanti, potranno essere effettuati al più k di questi spostamenti. Quindi il ciclo interno di InsertionSort ha costo $O(k)$, e il costo finale é $O(nk)$ (ma può essere $O(n)$ nel caso ottimo di un vettore ordinato).
Un approccio diverso, sempre $O(nk)$, anzi $\Theta(nk)$, parte da questo principio: l'elemento minimo si trova nel sottovettore $A[1 \dots k+1]$; applichiamo quindi una versione modificata del SelectionSort, che invece di cercare il minimo ovunque, lo cerca in $A[i \dots i+k]$.

```
sortK(Item[] A, int n, int k)
    for i = 1 to n − 1 do
        int min = min(A, i, i + k)
        A[i] ↔ A[min]
```

Infine, una volta resosi conto che se abbiamo ordinato i primi $i-1$ elementi, l'elemento i-esimo nell'ordinamento si trova nell'intervallo $A[i \dots i+k]$, è possibile utilizzare una Min-PriorityQueue contenente i prossimi $k+1$ valori per identificare il minimo, volta per volta. Una volta estratto il minimo, si inserisce l'elemento successivo e si ricomincia.

```
sortK(Item[] A, int n, int k)
    PriorityQueue Q = PriorityQueue(n)
    for i = 1 to k + 1 do
        Q.insert(A[i])
    for i = 1 to n do
        A[i] = Q.deleteMin()
        if i + k + 1 ≤ n then
            Q.insert(A[i + k + 1])
```

Ogni elemento viene inserito e rimosso dalla coda al massimo una volta; ogni rimozione ha costo $O(\log k)$. Il costo computazionale di questa soluzione è quindi $O(n \log k)$.

Esercizio A3 – Alberi ben bilanciati – Punti ≥ 12

Questo problema è molto simile al Problema A2 del 24/04/2014. Come in quell'esercizio, è possibile risolverlo con una semplice post-visita dell'albero, di costo $O(n)$. In quell'esercizio trovate anche varianti della soluzione che possono essere adattate a questo problema.

```
(int, boolean) wbRic(TREE t)
    if t == nil then
        return (0, true)
    n_L, balanced_L = wbRic(t.left)
    n_R, balanced_R = wbRic(t.right)
    return (n_L + n_R + 1, balanced_L and balanced_R and |n_L − n_R| ≤ 1)
```

```
boolean wellBalanced(TREE T)
    n, balanced = wbRic(T)
    return balanced
```

Nella funzione ricorsiva viene restituita una coppia di valori, dove il primo valore è un intero che rappresenta il numero di nodi compresi nel sottoalbero, mentre il secondo è un booleano uguale a **true** se il sottoalbero è ben-bilanciato.

Esercizio B1 – Punto d'incontro – Punti ≥ 8

Il problema è simile all'Esercizio 1 del 14/07/15 - quello di Alice e Bob che si devono incontrare, solo che non c'è Carl e la metrica per scegliere un nodo è leggermente diversa. Notate che la discussione sul minimo è complicata dal fatto che non volevo scrivere direttamente "cammini minimi", quindi ci ho girato un po' intorno.
È sufficiente utilizzare due ricerche dei cammini minimi, a partire dai nodi s e t.
Una volta ottenuti i pesi dei cammini minimi, è sufficiente cercare il nodo in cui pesi sono uguali. La procedura shortestPath() implementa l'algoritmo di Johnson con complessità $O(m \log n)$ e restituisce il vettore delle distanze (oltre che all'albero dei cammini minimi, che viene ignorato). Ora, nella mia testa i pesi erano positivi, ma ho scritto che sono in $\mathbb{R}$, quindi qui bisognerebbe applicare Bellman-Ford con costo $O(mn)$ (thanks Loris Lorenzini).

```
int meetingPoint(GRAPH G, int[][] w), NODE s, NODE t)
    int[], int[] d_s, T_s = shortestPath(G, s)
    int[], int[] d_t, T_t = shortestPath(G, t)
    foreach u ∈ G.V() do
        if d_s[u] == d_t[u] and d_s[u] ≠ ∞ then
            return true
    return false
```

La complessità è dominata da $O(m \log n)$, in quanto la ricerca del nodo con peso uguale ha costo lineare in n.

Esercizio B2 – PrintBits – Punti ≥ 10

Il problema può essere risolto con un approccio basato su backtrack, ovviamente. È sufficiente adattare l'algoritmo che genera tutti i sottoinsiemi di un elemento, con l'accorgimento di non selezionare un valore 1 se il valore precedente è un 1.

```
printBitsRec(int[] S, int n, int i)
    if i == n + 1 then
        print S
    else
        if i == 1 or S[i − 1] ≠ 1 then
            S[i] = 1
            printBitsRec(S, n, i + 1)
        S[i] = 0
        printBitsRec(S, n, i + 1)
```

```
printBits(int n)
    int[] S = new int[1...n]
    printBitsRec(S, n, 1)
```

È sufficiente dire che la complessità è $O(n \cdot 2^n)$, perché ovviamente le stringhe di n bit sono 2^n, ma non le stamperemo tutte. Più precisamente, abbiamo visto che il numero di stringhe di n bit senza due bit consecutivi cresce come l'$n+2$-esimo numero di fibonacci, quindi la complessità é $O(nf(n))$ (si veda l'esercizio del 7 Febbraio 2019).

Esercizio B3 – Sottosequenze k-contigue – Punti ≥ 12

s L'esercizio è simile all'Esercizio 2 del 17/12/2015, solo che invece di chiedere di non poter "saltare" più di k elementi, in quel problema non si potevano "prendere" più di k elementi consecutivi. Basta quindi adattare le formule per risolvere il problema. Sia $DP[i][j]$ il valore che posso ottenere dai primi i elementi, avendo ancora la possibilità di "cancellare" j elementi consecutivi. In altre parole, per via di selezioni precedenti, posso continuare a cancellare fino a j elementi, ma poi dovrò includere un elemento. La soluzione del problema originale si trova in $DP[n][k]$.
Una formulazione ricorsiva per DP è la seguente:

$$DP[i][j] = \begin{cases} 0 & i = 0 \\ DP[i-1][k] + V[i] & i > 0 \wedge j = 0 \\ \max\{DP[i-1][j-1], DP[i-1][k] + V[i]\} & i > 0 \wedge j > 0 \end{cases}$$

In pratica, se $j = 0$ si è costretti a prendere l'elemento, mentre se $j > 0$ si può scegliere di saltare o non saltare un elemento; se si decide di non saltare, il numero di elementi consecutivi selezionabili si resetta a k e bisogna sommare il valore selezionato; se si decide di saltare, restano a disposizione $j - 1$ elementi consecutivi da cancellare.
È possibile risolvere il problema con memoization nel modo seguente:

```
int maxSumK(int[] V, int n, int k)
    int[][] DP = new int[1...n][1...k] = {−1}        % Initialized to −1
    return maxSumKRec(V, n, k, DP)
```

```
int maxSumKRec(int[] V, int i, int j, int[][] DP)
    if i ≤ 0 then
        return 0
    if DP[i][j] < 0 then
        if j == 0 then
            DP[i][j] = maxSumKRec(V, i − 1, k, DP) + V[i]
        else
            DP[i][j] = max(maxSumKRec(V, i − 1, j − 1, DP), maxSumKRec(V, i − 1, k, DP) + V[i])
    return DP[i][j]
```

La complessità è pari a $O(nk)$, in quanto è necessario riempire tutta la tabella.
Una soluzione alternativa, abbozzata da alcuni studenti, è la seguente. Sia $DP[i]$ il valore della sottosequenza k-contigua massimale per i primi i elementi che termina nella posizione i-esima, cioè che include $V[i]$. Una formulazione ricorsiva per DP è la seguente:

$$DP[i] = \begin{cases} 0 & i \leq 0 \\ \max_{i-k \leq j < i}\{DP[j]\} + V[i] \end{cases}$$

L'idea è questa: se prendo l'elemento i-esimo, posso saltare 0 elementi (e quindi prendere anche $i-1$); saltare un elemento $(i-1)$ e prendere $i-2$; saltare due elementi ($i-1$, $i-2$ e prendere $i-3$); fino a quando non raggiungo $i-k-1$. Oltre non posso andare, perchè avrei saltato più di k elementi. E' quindi sufficiente prendere il massimo fra i valori $DP[j]$, con $i-k-1 \leq j < i$ e sommare il valore di $V[i]$. Ovviamente in presenza di 0 elementi il valore massimale è 0.
Attenzione però: il valore che cerchiamo non si trova in $DP[n]$ (che è la sottosequenza k-contigua massimale che termina nell'ultima posizione, anche se tale valore è negativo); non è nemmeno $\max(DP)$, perché il vettore DP include sequenze contigue k-massimali che terminano anche molto prima di n. Dobbiamo quindi prendere il massimo incluso nel sottovettore $DP[n-k \ldots n]$.
Il codice per risolvere il problema è il seguente:

```
int maxSumK(int[] V, int n, int k)
    int[] DP = new int[0...n]
    DP[0] = 0
    for i = 1 to n do
        DP[i] = −∞
        for j = max(0, i − k) to i − 1 do
            DP[i] = max(DP[i], DP[j] + V[i])
    return max(DP, n − k, n)
```

La complessità è $O(nk)$, come la soluzione precedente. La complessità spaziale è però $O(n)$, un miglioramento rispetto alla soluzione precedente che aveva complessità spaziale $O(nk)$.