Algoritmi e Strutture Dati – 16/06/2021

Esercizio A1 – Complessità – Punti ≥ 8

1. $f(n) = n$: si può applicare il teorema base, con $\alpha = 2$, $\beta = 1$, siamo nel caso (1) quindi la complessità è $\Theta(n^2)$.
2. $f(n) = n \log n$: si deve applicare la versione estesa, $\alpha = 2$, $n \log n = O(n^{\alpha-\epsilon})$ per $\epsilon < 1$, siamo nel caso (1) quindi la complessità è $\Theta(n^2)$.
3. $f(n) = n^2$: si può applicare il teorema base, con $\alpha = 2$, $\beta = 2$, siamo nel caso (2) quindi la complessità è $\Theta(n^2 \log n)$.
4. $f(n) = n^2 + \log n$: si deve applicare la versione estesa, ma è facile vedere che $f(n) = \Theta(n^2)$, siamo nel caso (2) quindi la complessità è $\Theta(n^2 \log n)$.
5. $f(n) = n^2 / \log n$: si deve applicare la versione estesa, ma non rientra in nessuno dei tre casi, quindi la risposta è N/A.
6. $f(n) = n^2 \log n$: dobbiamo usare la versione estesa, ma non rientra in nessuno dei tre casi, quindi la risposta è N/A.
7. $f(n) = n^3$: si può applicare il teorema base, con $\alpha = 2$, $\beta = 3$, siamo nel caso (3) quindi la complessità è $\Theta(n^3)$.
8. $f(n) = n^3 \log n$: dobbiamo usare la versione estesa; è facile vedere che $n^3 \log n$ è $\Omega(n^{2+\epsilon})$. Bisogna però anche dimostrare che $\exists c : 0 < c < 1, \exists m \geq 0 : af(n/b) \leq cf(n), \forall n \geq m$. Quindi:

$$\begin{aligned} 16(n/4)^3 \log n/4 = \\ \frac{1}{4}n^3(\log n - \log 4) \leq \\ \frac{1}{4}n^3 \log n \leq cn^3 \log n \end{aligned}$$

L'ultima disequazione è vera per $c = 1/4$. La complessità è quindi $\Theta(n^3 \log n)$.

Esercizio A2 – Trova la coppia – Punti ≥ 10

Questo esercizio è una variante dell'Esercizio 2 del 07/01/13. Invece di essere collocati in un unico vettore ordinato, i valori sono ordinati in due vettori distinti.
Una possibile soluzione consiste quindi nell'unire i due vettori in unico vettore ordinato grande $2n$, utilizzando una procedura simile alla merge() di mergesort() (costo $O(n)$) e poi chiamare la soluzione di quell'esercizio (con costo O(n)), per un costo totale $O(n)$.
Vediamo invece una soluzione ad-hoc per risolvere questa particolare versione del problema; per il principio di funzionamento, fate riferimento alla soluzione del 07/01/13.

```
int closePair(int[] A1, int[] A2, int n, int k)
    int i1 = 1
    int i2 = n
    boolean found = false
    while not found and i1 ≤ n and i2 ≥ 1 do
        sum = A1[i1] + A2[i2]
        if sum == k then
            found = true
        else if sum > k then
            i2 = i2 − 1
        else
            i1 = i1 + 1
    return found
```

La complessità dell'algoritmo è $O(n)$, perché i due indici possono scorrere da 1 a n e da n a 1.

- Una soluzione alternativa consiste nell'inserire tutti gli elementi di A_1 in un insieme implementato con tabella hash; il costo totale di inserimenti é $O(n)$. Dopo di che, per ogni elemento $a \in A_2$, si cerca se $k - a$ è presenta in tabella; costo totale $O(n)$, valutata 100%.
- Una soluzione più inefficiente consiste nell'utilizzare una ricerca dicotomica; per ogni elemento $a \in A_1$, si cerca $k - a$ in A_2 in tempo $O(\log n)$. Costo totale $O(n \log n)$. Questa soluzione è stata valutata 80%.
- Una soluzione ancora più inefficiente consiste nel testare tutte le coppie con due cicli annidati; una simile soluzione ha costo $O(n^2)$. Questa soluzione è stata valutata al 60%.

Esercizio A2 – isComplete – Punti ≥ 12

Sfruttiamo la memorizzazione dell'albero binario tramite vettore heap per verificare che l'albero possa essere memorizzato in un vettore di dimensione n e che tutte le caselle siano piene. Innanzitutto, utilizziamo la procedura count() definita nelle slide per contare il numero di nodi n.

Il meccanismo per calcolare la posizione di un nodo è il seguente: la radice ha posizione 1; un nodo i ha i figli in posizione $2i$ e $2i+1$. Se un nodo ha posizione $i > n$, vuole dire che l'albero non è completo, perchè non è memorizzabile in un heap di n nodi.

La funzione isComplete() chiama la funzione isInside(), che restituisce **true** se il nodo T può essere scritto in posizione i-esima, con $i \leq n$, e se i suoi sottoalberi possono essere contenuti correttamente nello heap; **false** altrimenti.

```
boolean isComplete(TREE T, int n)
    int n = count(T)
    return isInside(T, 1, n)
```

```
boolean isInside(TREE T, int i, int n)
    if T == nil then
        return true
    else if i > n then
        return false
    else
        return isInside(T.left, 2i, n) and isInside(T.right, 2i + 1, n)
```

La complessità è quella di due visite in profondità, quindi $\Theta(n)$.

Esercizio B1 – Stampa percorsi – Punti ≥ 8

L'esercizio è praticamente identico all'esercizio "Prima elementare" (Compito 01/02/2012), quello in cui si chiedeva di stampare n caratteri R e m caratteri G (per indicare palline gialle e palline rosse). Il punto è che per arrivare in posizione (n, n), bisognerà produrre n passi "D" e n passi R.
Si utilizza la tecnica del backtrack; se non esistono più lettere da collocare, abbiamo terminato la nostra stringa e la stampiamo. Altrimenti, se esiste la possibilità di aggiungere una lettera "D" si prova ad aggiungerla e si chiama la funziona ricorsiva con una lettera D in meno ($n_d - 1$); in modo simile si opera sulla lettera "R". Il numero di stringhe da stampare (ognuna lunga $2n$) è pari a $\frac{(2n)!}{n!n!}$. La complessità è quindi $O\left(\frac{n\cdot(2n)!}{n!n!}\right)$. È comunque corretto dire che la complessità è $O(n2^{2n})$, perché vengono fatti $2n$ scelte, ognuna delle quali ha due possibilità. Non è corretto dire che la complessità è $O(n2^n)$, perché non si può semplificare dicendo che $O(2^n) = O(2^{2n})$, in quanto $O(2^{2n}) = O(4^n)$.

```
printPaths(int n)
    S = new ITEM[1 ... 2n]
    printRec(S, 1, n, n)
```

```
printRec(char[] S, int i, int n_d, int n_r)
    if n_d == 0 and n_r == 0 then
        print S
    else
        if n_d > 0 then
            S[i] = "D"
            printRec(S, i + 1, n_d − 1, n_r)
        if n_r > 0 then
            S[i] = "R"
            printRec(S, i + 1, n_d, n_r − 1)
```

Esercizio B2 – removeSum – Punti ≥ 10

Questo esercizio è praticamente il duale dell'Esercizio B2 del 08/02/2021 (Zaino leggero). Invece di individuare il sottoinsieme che la cui somma è C e ha dimensione più piccola in assoluto, si deve individuare il sottoinsieme di elementi da rimuovere di dimensione più grande per cui la somma restante è la più grande possibile. È quindi possibile riutilizzare la soluzione

dell'esercizio Zaino Leggero, ottenendo la dimensione dell'insieme più piccolo e quindi sottrando tale valore dalla dimensione n.

```
int removeSum(int[] A, int n, int C)
    return n − smallSum(A, n, C)
```

Si noti che l'algoritmo proposto nel compito del 08/02/2021 restituisce $+\infty$ se non è possibile ottenere C; in questo caso, restituirà $-\infty$ ottenuto dalla sottrazione di $+\infty$ da n. La complessità è quella dell'esercizio del 08/02/2021, ovvero $O(nC)$
Volendo riscrivere invece la soluzione da zero, ripropongo solo la formula:

$$DP[i][c] = \begin{cases} i & c = 0 \\ -\infty & c < 0 \\ -\infty & c > 0 \wedge i = 0 \\ \max(DP[i-1][c] + 1, DP[i-1][c - W[i]]) & c > 0 \wedge i > 0 \end{cases}$$

Notate che se $c = 0$, significa che si è ottenuto il valore C senza utilizzare gli elementi negli indici compresi fra 1 e i; per questo motivo, è come se venissero rimossi e quindi $DP[i][c] = i$ in quel caso.

Esercizio B3 – Massima somma crescente – Punti $\geq$ 12

Questo esercizio è simile all'Esercizio 1.5 del file di esercizi sulla programmazione dinamica, con la differenza che non consideriamo la lunghezza delle sequenza crescente più lunga, ma la somma dei suoi valori. Nella definizione ricorsiva, basta sostituire tutte le occorrenze di 1 con $A[i]$. La versione presentata qui sotto semplifica ulteriori aspetti.
Denotiamo con $DP[i]$ il valore massimo fra tutte le sottosequenza crescenti che terminano nella posizione i-esima. È possibile calcolare $DP[i]$ in maniera ricorsiva. Si consideri l'indice i e si considerino gli elementi minori di $A[i]$ nel sottovettore $A[1 \dots i-1]$; $A[i]$ può essere utilizzato per estendere la sottosequenza di valore massimo che termina in uno di questi elementi. Se non esistono elementi minori, allora dobbiamo "ricominciare da capo", ovvero considerare la sequenza composta dal singolo valore $A[i]$.
Un modo per esprimerlo è il seguente:

$$DP[i] = \begin{cases} A[i] & \forall j, 1 \leq j < i : A[i] < A[j] \\ \max\{DP[j] : 1 \leq j \leq i-1 \wedge A[j] < A[i]\} + A[i] & \text{altrimenti} \end{cases}$$

Per risolvere il problema, utilizziamo programmazione dinamica. La funzione calcola tutti i valori $DP[i]$ con $i = 1 \dots n$, in ordine. Al termine, si restituisce il massimo fra i valori memorizzati in DP. Il costo della procedura è $O(n^2)$.

```
int maxSumIncreasing(int[] A, int n)
    int[] DP = new int[1...n]
    for i = 1 to n do
        DP[i] = A[i]                    % Se non troviamo valori minori, consideriamo solo questo elemento
        for j = 1 to i − 1 do
            if A[j] < A[i] and DP[j] + A[i] > DP[i] then
                DP[i] = DP[j] + A[i]
    return max(DP)
```

In più, è stato anche proposto come esercizio B3 del 18/07/19.