# *Algoritmi e Strutture Dati – 13/07/2022*

## Esercizio A1 – Complessità – Punti $\geq 8$

Questo esercizio è simile alla funzione di complessità dell'algoritmo deterministico per la selezione che funziona in tempo lineare. Proviamo $\Theta(n)$.

- Visto che la parte induttiva è lineare, la ricorrenza è $\Omega(n)$.
- Limite superiore $O(n)$, dimostrato per sostituzione con induzione. Partendo dal caso base, si ha che:

$$\begin{aligned}
&T(0) = 1 \leq c \cdot 0 \Leftrightarrow 1 \leq 0 \qquad \text{Falso!}\\
&T(1) = 1 \leq c \cdot 1 \Leftrightarrow c \geq 1\\
&T(2) = 2 + T(\lfloor 4/9 \rfloor) + T(\lfloor 10/18 \rfloor) = 2 + T(0) + T(0) = 4 \leq c \cdot 2 \Leftrightarrow c \geq 4/2\\
&T(3) = 3 + T(\lfloor 6/9 \rfloor) + T(\lfloor 15/18 \rfloor) = 3 + T(0) + T(0) = 5 \leq c \cdot 3 \Leftrightarrow c \geq 5/3\\
&T(4) = 4 + T(\lfloor 8/9 \rfloor) + T(\lfloor 20/18 \rfloor) = 4 + T(0) + T(1) = 6 \leq c \cdot 4 \Leftrightarrow c \geq 4/6\\
&T(5) = 5 + T(\lfloor 10/9 \rfloor) + T(\lfloor 25/18 \rfloor) = 5 + T(1) + T(1)
\end{aligned}$$

  Abbiamo calcolato tutti questi casi perché la condizione non viene rispettata per il caso $T(0)$, che poi si ripresenta nei casi $T(2)$, $T(3)$ e $T(4)$. A partire da $T(5)$, tuttavia, la ricorrenza utilizza solo casi già dimostrati; possiamo quindi interrompere il calcolo e utilizzare l'induzione.

  La soluzione corretta non è stata proposta da nessuno dei partecipanti al compito.

  Dimostriamo ora il caso induttivo, assumendo che $T(n') \leq cn'$ per tutti i valori $n' < n$, e volendo dimostrare che $T(n) \leq cn$. Sostituendo abbiamo che

$$\begin{aligned}
T(n) &\leq c\lfloor 2n/9 \rfloor + c\lfloor 5n/18 \rfloor + n\\
&\leq 2/9n + 5/18cn + n\\
&= 9/18cn + n \leq cn\\
&= 1/2cn + n \leq cn
\end{aligned}$$

  L'ultima disequazione è vera per $c \geq 2$. Quindi, per $m = 1, c = \max\{2, 4/2, 5/3, 4/6\} = 2$ abbiamo che $T(n) \leq cn, \forall n \geq m$.

## Esercizio A2 – Lago ricorsivo – Punti $\geq 10$

È possibile risolvere il problema effettuando una qualunque visita a partire dall'angolo in alto a destra (un punto del mare), e visitando tutti i nodi acqua che possono essere raggiunti nella quattro direzioni verticali e orizzontali. Per comodità utilizziamo il vettore stesso e poniamo a -1 i nodi acqua visitati. Se al termine della visita qualche nodo acqua non è stato visitato, esso fa parte di un lago. La complessità è $O(n^2)$, la dimensione dell'input.

```
boolean containsLake(int[][] M, int n)
  visitSea(M, n, 1, 1)
  for i = 1 to n do
      for j = 1 to n do
          if M[i][j] == 0 then
              return true
  return false
```

```
boolean visitSea(int[][] M, int i, int j, int n)
  if 1 ≤ i ≤ n and 1 ≤ j ≤ n and M[i][j] == 0 then
      M[i][j] = −1
      visitSea(M, i − 1, j, n)
      visitSea(M, i, j − 1, n)
      visitSea(M, i + 1, j, n)
      visitSea(M, i, j + 1, n)
```

Per velocità di scrittura, l'algoritmo utilizza una visita DFS ricorsiva; nella realtà, questo porterebbe volocemente all'esaurimento dello stack della chiamete ricorsive, perché nel caso pessimo questo algoritmo visita tutti gli $n^2$ nodi. Una implementazione più corretta sarebbe basata su una BFS, basata su una coda.

## Esercizio A3 – H-index – Punti ≥ 12

Vista la richiesta di risolvere il problema in tempo sub-lineare, utilizziamo la tecnica divide-et-impera. Scriviamo una procedura ricorsiva $\mathsf{hiRec}(A, i, j)$ che restituisce l'h-index, assumendo che sia un valore compreso fra $i$ e $j$. La procedura viene chiamata con $i = 1$ e $j = n$; essendo valori positivi, non dobbiamo gestire il caso particolare in cui l'h-index è pari a zero (in caso di assenza di citazioni).
Se $i = j$, l'h-index è pari a $i$, per assunzione che il valore sia compreso fra $i$ e $j$. Altrimenti, calcoliamo l'elemento centrale $m$. Se $A[m] \geq m$, allora l'h-index è sicuramente compreso fra $m$ e $j$; altrimenti è inferiore a $m$, e quindi compreso fra $i$ e $m - 1$. Notate l'uso dell'operatore di intero superiore: in questo modo, quando il vettore di riduce a due elementi, si prende il secondo come indice $m$; nel caso poi si prenda il sottovettore di sinistra, questo avrà almeno un elemento.

```
int h-index(int[] A, int n)
    return hiRec(A, 1, n)
```

```
int hiRec(int[] A, int i, int j)
    if i == j then
        return i
    else
        int m = ⌈i + j⌉/2
        if A[m] ≥ m then
            return hiRec(A, m, j)
        else
            return hiRec(A, i, m − 1)
```

La complessità è ovviamente $O(\log n)$.

## Esercizio B1 – Prodotto minimo – Punti ≥ 8

**Soluzione corretta, basata su greedy** Questo è un caso in cui un approccio greedy può funzionare.
L'idea è la seguente:

- Se il vettore contiene valori negativi, è possibile sfruttarli per fare in modo che il prodotto finale sia negativo. Sono dati due casi:
  - se il numero di valori negativi è dispari, si moltiplicano tutti i valori non nulli, ottenendo un prodotto negativo, il più basso possibile;
  - se il numero di valori negativi è pari, si multiplicano tutti i valori non nulli, tranne il valore negativo maggiore (quello più piccolo in valore assoluto), per evitare che il prodotto sia positivo.

  In entrambi i casi, l'obiettivo è ottenere il prodotto più alto in valore assoluto, utilizzando però un numero dispari di valori negativi per ottenere un valore finale negativo.
- Se il vettore contiene solo valori positivi o nulli, si restituisce il minimo valore positivo o nullo (che può essere zero).

La soluzione calcola il minimo valore positivo o nullo $min_{\geq 0}$ e il massimo valore negativo $min_{<0}$, oltre al prodotto di tutti i valori diversi da zero. Se esistono valori negativi ($max_{<0} \neq -\infty$), si restituisce il prodotto di tutti valori non nulli, se negativo (i valori negativi sono dispari); se positivo, si divide per il valore negativo più alto, per rimuovere un valore negativo e rendere il prodotto negativo.

```
int minproduct(int[] A, int n)
    int tot = 1
    int min_{≥0} = +∞
    int max_{<0} = −∞
    for i = 1 to n do
        if A[i] < 0 then
            max_{<0} = max(max_{<0}, A[i])
            tot = tot · A[i]
        else if A[i] > 0 then
            min_{≥0} = min(min_{≥0}, A[i])
            tot = tot · A[i]
        else
            min_{≥0} = min(min_{≥0}, A[i])
    if max_{<0} ≠ −∞ then
        if tot > 0 then
            tot = tot/ min_{≥0}
        return tot
    else
        return min_{≥0}
```

Il costo computazionale è ovviamente $O(n)$.

**Soluzione errata, basat su DP** Una soluzione che è comparsa molte volte è una semplice programmazione dinamica – errata.

```
int minproduct(int[] A, int n)
    int[] DP = new int[1 ... n]
    DP[1] = A[1]
    for i = 2 to n do
        DP[i] = min(DP[i − 1], DP[i − 1] · A[i], A[i])
    return DP[n]
```

I tre valori nella funzione min() rappresentano il minimo ottenuto finora, il valore precedente per $A[i]$, o $A[i]$ da solo.
Con un vettore $A = [-2, -2, -2]$, il risultato è un vettore $DP = [-2, -2, -2]$, con un risultato finale $-2$ che è diverso dal valore atteso $-8$. Il problema è che un volta che $DP[i]$ è negativo, moltiplicare per un valore negativo produce un valore positivo che non verrà mai selezionato, ma potrebbe essere utile per il calcolo successivo. In questo caso, la sottostruttura ottima non vale.

**Soluzione corretta, basata su DP** Una soluzione basata su programmazione dinamica richiede il calcolo sia del valore massimo che del valore minimo ottenibile con i primi $i$ valori; in questo modo, moltiplicando il massimo per un valore negativo, si ottiene un valore negativo molto basso.

```
int minproduct(int[] A, int n)
    int max = A[1]
    int min = A[1]
    for i = 2 to n do
        int tmp1 = A[i] · max
        int tmp2 = A[i] · min
        max = max(max, A[i], tmp1, tmp2)
        min = min(min, A[i], tmp1, tmp2)
    return min
```

## Esercizio B2 – Stampa con spazi – Punti $\geq 10$

Il problema può essere risolto tramite un algoritmo basato su backtrack. L'idea è la seguente: ogni carattere tranne il primo può essere preceduto da uno spazio oppure no. Creiamo quindi una funzione che genera tutte le sequenze di $n-1$ valori booleani e scriviamo una funzione di stampa che stampi il primo carattere in ogni caso, seguito dai caratteri successivi preceduto da spazi o meno a seconda del valore booleano corrispondente.
Il numero di stringhe effettivamente stampate è $2^{n-1}$. Il costo computazionale è quindi $O(n \cdot 2^n)$, dove il termine $n$ deriva dalla stampa.

```
printWithSpaces(ITEM[] A, int n)
    boolean[] S = new boolean[1...n]
    printRec(A, S, n)
```

```
printRec(ITEM[] A, ITEM[] S, int i)
    if i == 1 then
        print A[1]
        for i = 2 to n do
            if S[i] then
                print " "
            print A[i]
    else
        A[i] = true
        printRec(A, S, i − 1)
        A[i] = false
        printRec(A, S, i − 1)
```

## Esercizio B3 – Sequenze crescenti – Punti $\geq 12$

Il problema può essere risolto con programmazione dinamica.
Costruiamo una tabella bidimensionale $DP$ di dimensione $n \times k$, dove $DP[i][j]$ rappresenta il numero di sequenze crescenti che terminano con il valore $A[i]$ e hanno dimensione $j$. $DP[i][j]$ può essere calcolato ricorsivamente in questo modo:

$$DP[i][j] = \begin{cases} 1 & j = 1 \\ \sum_{1 \leq m < i : A[m] < A[i]} DP[m][j-1] & j > 1 \wedge i > 0 \\ 0 & i = 0 \end{cases}$$

L'equazione ricorsiva può essere intepretata in questo modo:

- Se $j = 1$, il numero di sequenze crescenti lunghe 1 che terminano in posizione $A[i]$ (con $i > 0$) è pari a una, ovvero la sequenza composta dal solo valore $A[i]$.
- Altrimenti, consideriamo le sequenze crescenti lunghe $j - 1$ che terminano in un valore $A[m]$, con $m < i$ e $A[m] < A[i]$.
- La sezione $i = 0$ viene attivata nel caso in cui si è preso un valore $A[m]$ con un indice $m$ troppo piccolo e non c'è spazio per selezionare abbastanza elementi per fare una sequenza lunga $j$.

Il risultato desiderato si trova sommando i valori nell'ultima colonna: $\sum_{i=1}^{n} A[i][k]$, che corrispondono al numero di sequenze crescenti di $k$ elementi che termina in posizione $i$-esima, in quanto ogni valore può essere l'ultimo della sequenza. È possibile notare tuttavia, che per indici $i < k$, il valore è sempre zero (è impossibile ottenere una sequenza di $k$ elementi che termina in $A[i]$ se $i < k$), quindi è possibile utilizzare questa formula: $\sum_{i=k}^{n} A[i][k]$.
Notate che rimuovendo il controllo su $i > 0$ nella seconda riga, si può ottenere una versione più semplice che accorpa il caso $i = 0$ in quello precedente:

$$DP[i][j] = \begin{cases} 1 & j = 1 \\ \sum_{1 \leq m < i : A[m] < A[i]} DP[m][j-1] & j > 1 \end{cases}$$

Quando $i = 1$ e $j > 1$, si ricade nella sommatoria; ma non esistono indici $m$ compresi fra 1 e $i - 1 = 0$, quindi la sommatoria darà 0. Usiamo quest'ultima versione per scrivere il codice, basato su programmazione dinamica.

```
int countIncreasing(int[] A, INTEGER n, int k)
    int[][] DP = new int[1...n][1...k] = { 1 }
    for i = 1 to n do
        for j = 2 to k do
            DP[i][j] = 0
            for m = 1 to i − 1 do
                if A[m] < A[i] then
                    DP[i][j] = DP[i][j] + DP[m][j − 1]
    tot = 0
    for i = k to n do
        tot = tot + A[i][k]
    return tot
```

Il costo computazionale è pari a $O(kn^2)$, per via dei tre cicli **for**.