Algoritmi e Strutture Dati - 06/06/2019

Esercizio B1

L'esercizio è molto semplice; con l'esclusione della radice in posizione 1, se l'elemento i del vettore è più piccolo dell'elemento padre in posizione $\lfloor i/2 \rfloor$, allora il vettore non rappresenta un albero min-heap e l'algoritmo deve restituire **false**. Se tutti gli elementi rispettano questa condizione, si deve restituire **true**.
L'algoritmo, riportato sotto, ha complessità lineare $\Theta(n)$.

```
boolean checkPriority(int[] V, int n)
    int i = 2
    boolean ok = true
    while i ≤ n and ok do
        if V[⌊i/2⌋] > V[i] then
            ok = false
        i = i + 1
    return ok
```

Esercizio B2

Per risolvere questo esercizio, è necessario adattare lo schema di backtrack che abbiamo visto a lezione.
L'insieme dei colori selezionabili è pari all'intero insieme di colori, se si tratta della prima striscia; di tutti i colori abbinabili a quello precedente, negli altri casi.
La complessità risultante è $O(n \cdot k^n)$, superpolinomiale. Il termine n deriva dall'operazione di stampa, mentre k^n deriva dal fatto che nel caso pessimo di un matrice con valori **false** solo sulla diagonale, ogni colore è abbinabile a ogni altro colore escluso se stesso.

```
printFlags(int[][] A, int n, int k)
    int[] S = new int[1 ... n]
    printFlagsRec(A, S, 1, n, k)
```

```
printFlagsRec(int[][] A, int[] S, int i, int n, int k)
    if i == n + 1 then
        print S
    else
        for j = 1 to k do
            if i == 1 or A[j][S[i − 1]] then
                S[i] = j
                printFlagsRec(A, S, i + 1, n, k);
```

Esercizio B3

Il problema si risolve tramite programmazione dinamica. Sia $DP[r][c][b]$ il numero di cammini da $(1, 1)$ a (r, c) il cui costo è inferiore o uguale a b.
È possibile calcolare questo valore ricorsivamente:

- Se il budget a disposizione è inferiore a zero ($b < 0$), oppure siamo usciti dalla scacchiera ($r = 0$ **or** $c = 0$), allora il numero di cammini disponibili è 0.
- Se $r = c = 1$ e il budget non è negativo, stiamo contando il numero di cammini da $(1, 1)$ a $(1, 1)$, che ovviamente è 1.
- Altrimenti, il numero di cammini è pari alla somma dei cammini da $(1, 1)$ a $(r-1, c)$ più i cammini da $(1, 1)$ a $(r, c-1)$, sottraendo $P[r][c]$ dal budget. Questo perché il pedone è arrivato in (r, c) o venendo da $(r-1, c)$ oppure venendo da $(r, c-1)$.

$$DP[r][c][b] = \begin{cases} 0 & b < 0 \textbf{ or } r = 0 \textbf{ or } c = 0 \\ 1 & r = 1 \textbf{ and } c = 1 \textbf{ and } b \geq 0 \\ DP[r-1][c][b-P[r][c]] + DP[r][c-1][b-P[r][c]] & \text{altrimenti} \end{cases}$$

Partendo da questa formula ricorsiva, è possibile applicare memoization:

int countPaths(**int**[ ][ ] P, **int** n, **int** B)

```
size = min(10 · (2n − 2), B)
int[ ][ ] DP = new int[1 . . . n][1 . . . n][0 . . . size]
for i = 1 to n do
    for j = 1 to n do
        for k = 0 to size do
            DP[i][j][k] = −1
return countPathsRec(P, DP, n, n, size)
```

int countPathsRec(**int**[ ][ ] P, **int**[ ][ ] DP, **int** r, **int** c, **int** b)

```
if b < 0 or r == 0 or c == 0 then
    return 0
if b ≥ 0 and r == 1 and c == 1 then
    return 1
if DP[r][c][b] < 0 then
    DP[r][c][b] = countPathsRec(P, DP, r − 1, c, b − P[r][c]) + countPathsRec(P, DP, r, c − 1, b − P[r][c])
return DP[r][c][b]
```

Si noti che invece di dimensione la tabella in base al valore B, si utilizza il minimo fra B e $10 \cdot (2n-2)$. Questo perché un cammino ha lunghezza fissa pari a $2n-2$, ogni cella costa al più 10, quindi un budget superiore a $10 \cdot (2n-2)$ non è differente da un budget pari a quel valore.
In questo modo, la complessità dell'algoritmo diventa $O(n^3)$, in quanto $size$ è $O(n)$ e quindi l'algoritmo è polinomiale. Lo spazio occupato è sempre $O(n^3)$, ma può essere ridotto a $O(n^2)$ visto che alla fine non è richiesto di ricostruire una soluzione.