Algoritmi e Strutture Dati - 04/02/2020

Esercizio A1

La funzione buildHeap() ha costo lineare nel numero di elementi del vettore. Il vettore viene diviso in due parti e ha quindi costo:

$$T(n) = \begin{cases} 2T(\lfloor n/2 \rfloor) + n & n \geq 2 \\ 1 & n \leq 2 \end{cases}$$

Applicando il Master Theorem - versione normale, si ottiene $T(n) = \Theta(n \log n)$, in quanto $\alpha = \log_a b = 1$, $\beta = 1$ e quindi siamo nel secondo caso.

Esercizio A2

La matrice può essere interpretata come un grafo e può essere risolto con una semplice visita, in profondità o in ampiezza. In questo caso, mostriamo una soluzione basata su visita in profondità.
La funzione hasPathRec() restituisce vero se è possibile raggiungere (n, n) a partire dalla cella i, j. Se la cella $(i, j) = (n, n)$, allora la risposta è banalmente **true**. Se la cella non è contenuta nella griglia o è già stata visitata, la risposta è banalmente **false**. Altrimenti, se una qualunque delle celle a distanza $A[i][j]$ da (i, j) può raggiungere (n, n) allora si restituisce **true**; se non è possbile raggiungere (n, n) in alcun modo, ritorna **false**.
La complessità è quella di una visita di un grafo, $O(|V| + |E|)$; poiché ci sono n^2 nodi e meno di $4n^2$ archi, la complessità è $\Theta(n^2)$.

```
boolean hasPath(int[][] A, int n)
    boolean[][] visited = new boolean[1...n][1...n] = { false }    % Initialized to false
    return hasPathRec(A, visited, n, 1, 1)
```

```
boolean hasPathRec(int[][] A, boolean[][] visited, int n, int i, int j)
    if i == n and j == n then
        return true
    else if 1 ≤ i ≤ n and 1 ≤ j ≤ n and not visited[i][j] then
        visited[i][j] = true
        return  hasPathRec(A, visited, n, i + A[i][j], j) or
                hasPathRec(A, visited, n, i, j + A[i][j]) or
                hasPathRec(A, visited, n, i − A[i][j], j) or
                hasPathRec(A, visited, n, i, j − A[i][j])
    else
        return false
```

Esercizio A3

La soluzione proposta si basa sulla soluzione del problema larghezza() descritto nel libro e risolto negli esercizi dedicati agli alberi. Tutte le volte che tutti i nodi del livello k vengono estratti, la coda contiene tutti e soli i nodi del livello $k + 1$. Quindi, il primo nodo inserito in coda e l'ultimo del livello precedente sono i nodi che vado cercando. Attenzione però al caso in cui esiste un solo nodo nel livello; a seconda di come è scritto il codice, potrebbe capitare di sommare il suo contributo due volte. Nel codice sotto, sommiamo il primo nodo del livello successivo solo se il livello successivo contiene 2 o più nodi. Se ne contiene 1, il suo lavoro verrà sommato quando viene svuotato. Se ne contiene 0, non c'è nulla da sommare.

Essendo una semplice vista in ampiezza, il costo computazionale è pari a $\Theta(n)$.

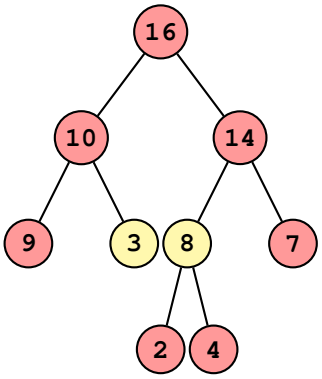
```
int sumAngular(TREE T)
    int count = 1                          % # of nodes to be visited of current level; at the beginning, just the root
    int sum = 0                            % Sum found so far
    QUEUE Q = Queue()
    Q.enqueue(t)
    while not Q.isEmpty() do
        TREE t = Q.dequeue()
        if t.left ≠ nil then
            Q.enqueue(t.left)
        if t.right ≠ nil then
            Q.enqueue(t.right)
        count = count − 1
        if count == 0 then                 % New level
            count = Q.size()               % All the nodes in the queue belong to the next level
            sum = sum + t.value            % Add the last value of the current level
            if count ≥ 2 then
                sum = sum + Q.top().value  % Add the first value of the next level, unless it is the only node in the next level
    return sum
```

Esistono anche altre soluzioni che hanno costo sempre lineare e non sono basata su BFS, ma su DFS. Ad esempio, utilizzando due tabelle hash che associano ad ogni livello il primo e l'ultimo nodo. Queste due tabelle hash possono essere riempite utilizzando una visita in profondità, per poi prendere la minima differenza fra i due nodi.
Purtroppo, molte delle soluzioni proposte non sfruttavano questo approccio, ma hanno usato malamente una visita in profondità. L'idea generale era questa: sul sottoalbero destro della radice, si va a destra, a meno che non ci sia solo un figlio a sinistra, nel qual caso si va a sinistra e poi si torna ad andare a destra. Nel sottoalbero sinistro della radice, si procede in maniera simmetrica.
All'incirca, il codice somiglia a questo schema semplificato:

```
int sumAngular(TREE T)
    return T.value + sumAngularLeft(T.left) + sumAngularRight(T.right)
```

```
int sumAngularLeft(TREE T)
    if T ≠ nil then
        if T.left ≠ nil then
            return T.value + sumAngularLeft(T.left)
        else if T.right ≠ nil then
            return T.value + sumAngularLeft(T.right)
        else
            return T.value
    else
        return 0;
```

La procedura sumAngularRight() è simmetrica. Di varianti di questo schema ne ho viste tante, ma nessuna riesce a gestire una situazione come questa illustrata qui sotto, dove i nodi angolari di un livello sono figli dei nodi centrali.



Esercizio B1

Enumerare tutti i percorsi è un problema di natura potenzialmente superpolinomiale; va quindi risolto tramite backtrack.
L'idea è simile a quanto visto in altri compiti: utilizziamo la ricorsione per visitare, in profondità, la prossima mossa sulla matrice. Utilizziamo una matrice $visited$ per segnare le caselle già visitate; $visited[i][j]$ sarà posta a **true** quando una cella viene visitata, per poi ritornare a **false** al completamento della visita.
Se arriviamo alla casella finale (n, n), ritorniamo il valore 1, per contare un cammino semplice che termina in (n, n). Altrimenti, contiamo quanti cammini possono essere ottenuti andando nelle quattro direzioni permesse.
Se (i, j) non è un cella valida oppure è già stata visita, ritorniamo 0.
La complessità dell'algoritmo proposto è limitata superiormente da $O(4^{n^2})$; è possibile notare che a meno di valori tutti pari a 1, gran parte dell'albero della visita viene potato dal fatto che molti salti non sono possibili.

```
int countPaths(int[][] A, int n)
    boolean[][] visited = new boolean[1...n][1...n] = { false }      % Initialized to false
    return countRec(A, visited, n, 1, 1)
```

```
int countRec(int[][] A, boolean[][] visited, int n, int i, int j)
    if i == n and j == n then
        return 1
    else if 1 ≤ i ≤ n and 1 ≤ j ≤ n and not visited[i][j] then
        visited[i][j] = true
        int ret = countRec(A, visited, n, i + A[i][j], j) +
                  countRec(A, visited, n, i, j + A[i][j]) +
                  countRec(A, visited, n, i − A[i][j], j) +
                  countRec(A, visited, n, i, j − A[i][j])
        visited[i][j] = false
        return ret
    else
        return 0
```

Ho visto alcuni tentativi di risolvere il problema tramite programmazione dinamica. Senza entrare nei dettagli, molte di queste soluzioni (i) si "muovevano" avanti e indietro nello spazio della matrice e (ii) non tenevano conto delle celle già visitate lungo un cammino. In altre parole, non avevano la proprietà di sottostruttura ottima necessaria per poter applicare la programmazione dinamica.

Esercizio B2

Il problema può essere risolto trasformandolo in un'istanza di Somma Massimale, il primo algoritmo visto a lezione. Si crea un vettore di supporto che contiene 1 per ogni valore 0, -1 per ogni valore 1. In questo modo, la somma massimale in un vettore corrisponde allo zero-sbilanciamento dello stesso.

Il costo dell'algoritmo è quindi $\Theta(n)$.

```
int zeroUnbalance(int[] A, int n)
    int[] B = new int[1...n]
    for i = 1 to n do
        B[i] = iif(A[i] == 0, 1, -1)
    return maxsum(B, n)
```

Vorrei comunque far notare che qualunque degli approcci visti per somma massimale possono essere utilizzati per creare una soluzione ad-hoc per il problema. Ad esempio, la soluzione seguente ha costo $\Theta(n^3)$ e dovrebbe essere proposta da chiunque abbia completato con successo un corso di Programmazione 1.

```
int zeroUnbalance(int[] A, int n)
    int maxSoFar = 0
    for i = 1 to n do
        for j = i to n do
            int n_1 = sum(A, i, j)        % Number of ones in A[i...j], computational cost O(n)
            int n_0 = (j - i + 1) - n_1   % Number of zeros in A[i...j]
            int unbalance = n_0 - n_1
            maxSoFar = max(maxSoFar, unbalance)
    return maxSoFar
```

Meglio scrivere una soluzione di questo tipo (che non era esplicitamente vietata) che non scrivere nulla.

Esercizio B3

Anche in questo caso, è possibile scrivere una soluzione molto semplice in $\Theta(n^4)$. Si può fare (leggermente) meglio di così.
Si costruiscano due matrici di supporto R, C di dimensione $n \times n$ tali per cui

- $R[i][j]$ contiene la lunghezza della più lunga sequenza orizzontale di 1 che inizia a sinistra di (i, j) e termina in (i, j);
- $C[i][j]$ contiene la lunghezza della più lunga sequenza orizzontale di 1 che inizia sopra di (i, j) e termina in (i, j).

Questi due vettori vengono facilmente calcolati in tempo $O(n^2)$.
A questo punto, possono essere utilizzati per verificare se esiste un quadrato di dimensione $k \in [1, \ldots, \min([R[i][j], C[i][j])]$, utilizzando le matrici R, C per verificare in tempo $O(1)$ se esistono tutti i lati.
Sono necessari tre cicli annidati e la complessità è quindi $O(n^3)$. Tuttavia, è possibile migliorare leggermente l'algoritmo notando che è inutile verificare quadrati di dimensione $maxSoFar$ o inferiori, dove $maxSoFar$ è il quadrato più grande trovato finora.

```
int largestSquare(int[][] A, int n)
    int[][] R = new int[1...n][1...n]
    int[][] C = new int[1...n][1...n]
    for i = 1 to n do
        for j = 1 to n do
            R[i][j] = iif(A[i][j] == 0, 0, R[i][j-1] + 1)
            C[i][j] = iif(A[i][j] == 0, 0, C[i-1][j] + 1)
    int maxSoFar = 0                      % Max found so far
    for i = 1 to n do
        for j = 1 to n do
            int m = min(R[i][j], C[i][j])
            for k = maxSoFar + 1 to m do
                if R[i-k+1][j] ≥ k and C[i][j-k+1] ≥ k then
                    maxSoFar = max(maxSoFar, k)
    return maxSoFar
```