Algoritmi e Strutture Dati - 07/02/2019

Esercizio A1

Limite superiore Andando per tentativi, proviamo con $O(n)$. Proviamo quindi a dimostrare che $T(n) = O(n)$.

- Ipotesi induttiva: $T(k) \leq ck$, per $k < n$
- Passo induttivo:

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lfloor n/4 \rfloor) + T(\lfloor n/8 \rfloor) + 2T(\lfloor n/16 \rfloor) + 1 \\ &\leq c\lfloor n/2 \rfloor + c\lfloor n/4 \rfloor + c\lfloor n/8 \rfloor + 2c\lfloor n/16 \rfloor + 1 \\ &\leq \frac{1}{2}cn + \frac{1}{4}cn + \frac{1}{8}cn + \frac{1}{8}cn + 1 \\ &= cn + 1 \leq cn \end{aligned}$$

  L'ultima disequazione è falsa, ma per un termine di ordine inferiore.

Proviamo quindi a dimostrare che

$$\exists b > 0, \exists c > 0, \exists m \geq 0 : T(n) \leq cn - b, \forall n \geq m$$

- Ipotesi induttiva: $T(k) \leq ck - b$, per $k < n$
- Passo induttivo:

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lfloor n/4 \rfloor) + T(\lfloor n/8 \rfloor) + 2T(\lfloor n/16 \rfloor) + 1 \\ &\leq c\lfloor n/2 \rfloor - b + c\lfloor n/4 \rfloor - b + c\lfloor n/8 \rfloor - b + 2 \cdot (c\lfloor n/16 \rfloor - b) + 1 \\ &\leq \frac{1}{2}cn - b + \frac{1}{4}cn - b + \frac{1}{8}cn - b + \frac{2}{16}cn - 2b + 1 \\ &= cn - 5b + 1 \leq cn - b \end{aligned}$$

  L'ultima disequazione è vera per ogni c e per $b \geq 1/4$.

- Caso base: $T(n) = 1 \leq cn - b$, per tutti i valori di n compresi fra 1 e 16, ovvero:

$$c \geq \frac{b+1}{n}, \forall n : 1 \leq n \leq 16$$

  I valori $\frac{b+1}{n}$ sono minori o uguali di $5/4$, per $1 \leq n \leq 16$; quindi tutte queste disequazioni sono soddisfatte da $c \geq 5/4$.

Abbiamo quindi dimostrato che $T(n) = O(n)$, con $m = 1$ e $c \geq 5/4$.

Esercizio A2

L'esercizio è molto semplice, ma bisogna fare attenzione a gestire bene gli indici.
Utilizzeremo una funzione ricorsiva firstRec(**int**[ ] A, **int** i, **int** j, **int** v), richiamata da first(), che assume che il valore sia presente nel sottovettore $A[i \dots j]$ e quindi la dimensione del sottovettore sia almeno 1:

```
int first(int [ ] A, int n, int v)
    return firstRec(A, 1, n, v)
```

Come caso base, se il sottovettore ha dimensione 1 ($i == j$) e tenuto conto che il valore è sicuramente presente, allora bisogna ritornare i stesso.
Altrimenti, sono dati due casi:

- Se il valore cercato è più piccolo o uguale ad $A[m]$, allora la prima occorrenza del valore cercato si troverà sicuramente nel sottovettore di sinistra $A[i \dots m]$ (m incluso, in quanto $A[m]$ può contenere la prima occorrenza di v). Per come è calcolato m (intero inferiore), la dimensione del sottovettore è almeno 1.
- Se il valore se il valore cercato è più grande di $A[m]$, allora la prima occorrenza si troverà sicuramente nel sottovettore di destra, m escluso. Per come è calcolato m (intero inferiore), la dimensione del sottovettore è almeno 1.

```
int firstRec(int [ ] A, int i, int j, int v)
if i == j then
    return i
else
    int m = ⌊(i + j)/2⌋
    if v ≤ A[m] then
        return firstRec(A, i, m, v)
    else
        return firstRec(A, m + 1, j, v)
```

Molti studenti hanno proposto questa versione, che fa un controllo in più ma è leggermente più chiara:

```
int firstRec(int [ ] A, int i, int j, int v)
if i == j then
    return i
else
    int m = ⌊(i + j)/2⌋
    if v < A[m] then
        return firstRec(A, i, m − 1, v)
    else if v == A[m] then
        return firstRec(A, i, m, v)
    else
        return firstRec(A, m + 1, j, v)
```

Molti studenti hanno proposto una versione (che non riporto qui) in cui quando $A[m] = v$, si cerca in $A[m-1]$ per vedere se per caso è anche la prima occorrenza. Bisogna però fare attenzione che m non sia uguale a 1 (primo elemento del vettore). Se scritta bene, ha la stessa complessità delle altre, ma la trovo poco elegante.

Uno studente ha proposto questa versione, che modifica il caso base. L'idea è semplice: se il primo elemento del vettore è pari al valore cercato, ritorniamo quel valore perché è sicuramente la prima occorrenza. Altrimenti, sappiamo che la prima occorrenza si trova nel vettore e lavoriamo come in precedenza. Un po' più difficile da spiegare, ma funziona.

```
int firstRec(int [ ] A, int i, int j, int v)
if A[i] == v then
    return i
else
    int m = ⌊(i + j)/2⌋
    if v ≤ A[m] then
        return firstRec(A, i, m, v)
    else
        return firstRec(A, m + 1, j, v)
```

La complessità per tutte queste versioni è $O(\log n)$, come richiesto.

Si noti che se vi avessi chiesto di realizzare last() (la funzione che restituisce l'ultima posizione di un'occorrenza), sarebbe stato facile incorrere in un errore con gli indici. Ve lo lascio come esercizio.

Parecchi studenti hanno utilizzato una ricerca dicotomica normale, trovando un valore v qualunque, e poi hanno cercato linearmente a sinistra di quel valore - vedi sotto. Se questo algoritmo riceve in input un insieme di valori tutti uguali, analizzerà metà del vettore; quindi la complessità nel caso pessimo sarà $O(n)$.

A questi studenti ho assegnato 0% come voto, perché non rispetta le consegne. Altrimenti, scrivere un semplice ciclo **for** sarebbe stato sufficiente (e più rapido).

```
int firstRec(int [ ] A, int i, int j, int v)
if i == j then
    return i
else
    int m = ⌊(i + j)/2⌋
    if v < A[m] then
        return firstRec(A, i, m − 1, v)
    else if v == A[m] then
        while m > 0 and A[m] == v do
            m = m − 1
        return m + 1
    else
        return firstRec(A, m + 1, j, v)
```

Esercizio A3

Il testo è volutamente complicato (ma non ambiguo). Alla fine, è facile vedere che il messaggio m_e (m_p) raggiunge un nodo u al giorno $t_e = d_e[u]$ ($t_p = d_p[u]$), dove $d_e[u]$ ($d_p[u]$) è la minima distanza (in numero di archi) fra u e un nodo inizialmente appartenente a PE (PP).
Per calcolare la distanza di un nodo da un insieme di altri nodi, possiamo prendere ispirazione da un problema simile visto durante le esercitazioni (distanza fra due insiemi di nodi).
Calcolare le distanze è facile, è sufficiente modificare l'algoritmo BFS mettendo inizialmente nella coda tutti i nodi che appartengono ad un partito e settando la loro distanza a zero.
Una volta calcolate le distanze da entrambi i partiti, un nodo resterà indeciso per sempre se e solo se verrà raggiunto dai messaggi nello stesso giorno, ovvero se ha la stessa distanza dai nodi dei due partiti.
Il costo dell'algoritmo così proposto è pari a $O(m + n)$, corrispondente a due visite in ampiezza e al costo di verificare, per ogni nodo, se la distanza dei partiti è uguale.

```
int[] dist(GRAPH G, int[]status, int party)
QUEUE Q = Queue()
int[] dist = new int[1 . . . G.n]
foreach u ∈ G.V() do
    if status[u] == party then
        Q.enqueue(u)
        dist[u] = 0
    else
        dist[u] = ∞
while not Q.isEmpty() do
    NODE u = Q.dequeue()
    foreach v ∈ G.adj(u) do
        if dist[v] == ∞ then
            dist[v] = dist[u] + 1
            Q.enqueue(v)
return dist
```

```
int election(GRAPH G, int[]status)
    int[] distPE = dist(G, status, PE)
    int[] distPP = dist(G, status, PP)
    int tot = 0
    for u = 1 to G.n do
        tot = tot + iif(distPE[u] == distPP[u], 1, 0)
    return tot
```

È possibile realizzare una versione in cui viene effettuata una sola visita, invece di due, aggiornando due vettori di distanze contemporaneamente e partendo dall'unione dei nodi PE e PP. Ma la complessità resta uguale, $O(m + n)$.
Esiste anche la possibilità di "simulare" il processo, con un ciclo che scorre tutti i nodi e tutti gli archi, ripetuto n volte. La complessità di questo algoritmo è $O(mn)$.

Esercizio B1

Il problema è una leggera variante del problema di Hateville, in cui bisogna contare invece che cercare il massimo.
Sia $DP[n]$ il numero di stringhe di n bit che non contengono due 1 consecutivi; può essere calcolato ricorsivamente nel modo seguente:

$$DP[n] = \begin{cases} 1 & n = 0 \\ 2 & n = 1 \\ DP[n-1] + DP[n-2] & n \geq 2 \end{cases}$$

Nel caso generale, se piazzo un `1` come ultimo bit, poi dovrò piazzare un `0`; conto quindi le combinazioni con $n-2$ bit. Se piazzo uno `0` come ultimo bit, non ho vincoli e quindi conto le combinazioni con $n-1$ bit. Questi due valori vanno sommati insieme, essendo esclusivi.

```
int bits(int n)
    int[] DP = new int[0...n]
    DP[0] = 1
    DP[1] = 2
    for i = 2 to n do
        DP[i] = DP[i-1] + DP[i-2]
    return DP[n]
```

Ancora una volta, quello che viene generata è la successione di Fibonacci.
L'algoritmo richiede $O(n)$ somme. Si noti tuttavia che n non è la dimensione del problema, è l'input del problema. Secondo il modello di costo logaritmico, il vero costo è $O(n^2)$, in quanto il numero di bit per rappresentare bits(n) (e quindi il numero di operazioni per effettuare le somme) cresce linearmente con n.

Esercizio B2

Il problema si risolve tramite backtrack. Oltre alle stringhe T e P, viene passato uno stack.
Vengono passati due indici i e j, che vengono utilizzati per indicare il prefisso i-esimo di T e il prefisso j-esimo di T. Se $j = 0$, allora non ci sono più caratteri nel pattern, e quindi bisogna stampare lo stack (in ordine LIFO, in modo da stampare l'ultimo elemento inserito per primo).
Altrimenti, se il numero di caratteri nel prefisso i-esimo di T è maggiore o uguale al numero di caratteri nel prefisso j-esimo di P, allora la computazione si biforca ricorsivamente: o semplicemente ignoriamo l'ultimo carattere di T (considerando il prefisso $i-1$-esimo), oppure, se i caratteri sono uguali, inseriamo l'indice i nello stack e chiamiamo la funzione ricorsivamente con $i-1$ e $j-1$.

```
printAll(ITEM[] T, int n, ITEM[] P, int m)
    STACK S = Stack()
    printAllRec(T, P, n, m, S)
```

```
printAll(ITEM[] T, ITEM[] P, int i, int j, STACK S)
    if j == 0 then
        print S                                  % Stampa in ordine LIFO
    else if i ≥ j then
        { Caso in cui si ignora l'ultimo carattere di P }
        printAllRec(T, P, i-1, j, S)
        { Caso in cui si considera l'ultimo carattere di P, se sono uguali }
        if T[i] == P[j] then
            S.push(i)
            printAllRec(T, P, i-1, j-1, S)
            S.pop
```

Nel caso le due stringhe siano composte da n ed m caratteri tutti uguali ad un carattere c, ogni chiamata ricorsiva può generare due chiamate ricorsive, con un costo limitato superiormente da $O(2^n)$. È importante notare tuttavia che tramite il meccanismo di pruning e tenuto conto che in input "normali" i caratteri saranno diversi, il numero di chiamate può essere molto inferiore, specialmente per valori di m vicini a 1 ed n.

Esercizio B3

La soluzione è basata su programmazione dinamica. Si costruiscono quattro matrici di appoggio, chiamate L, R, U, D (left, right, up, down). Le caselle $L[i][j], R[i][j], U[i][j], D[i][j]$ contiene la più lunga fila di valori 1 che si estende a partire dalla cella (i, j) (compresa) nella direzione sinistra, destra, alto, basso.
Questi valori possono essere calcolati nel modo seguente:

$$L[i][j] = \begin{cases} M[i][j] & j = 1 \\ L[i][j-1] + 1 & j > 1 \wedge M[i][j] = 1 \\ 0 & j > 1 \wedge M[i][j] = 0 \end{cases}$$

$$R[i][j] = \begin{cases} M[i][j] & j = n \\ R[i][j+1] + 1 & j < n \wedge M[i][j] = 1 \\ 0 & j < n \wedge M[i][j] = 0 \end{cases}$$

$$D[i][j] = \begin{cases} M[i][j] & i = n \\ D[i+1][j] + 1 & i < n \wedge M[i][j] = 1 \\ 0 & i < n \wedge M[i][j] = 0 \end{cases}$$

$$U[i][j] = \begin{cases} M[i][j] & i = 1 \\ U[i-1][j] + 1 & i > 1 \wedge M[i][j] = 1 \\ 0 & i > 1 \wedge M[i][j] = 0 \end{cases}$$

Una volta calcolati questi valori, la più grande croce centrata in i, j è data da $\mathsf{min}(L[i][j], R[i][j], U[i][j], D[i][j]) * 4 - 3$ (i quattro bracci, tenuto conto che l'elemento centrale è contato quattro volte invece di 1, da cui il valore -3).

```
int cross(int[][] M, int n)
    int[][] L = new int[1...n][1...n]
    int[][] R = new int[1...n][1...n]
    int[][] U = new int[1...n][1...n]
    int[][] D = new int[1...n][1...n]
    for j = 1 to n do
        U[1][j] = M[1][j]
        D[n][j] = M[n][j]
    for i = 1 to n do
        L[i][1] = M[i][1]
        R[i][n] = M[i][n]
    for i = 2 to n do
        for j = 2 to n do
            L[i][j] = iif(M[i][j] == 0, 0, L[i][j-1] + 1)
            U[i][j] = iif(M[i][j] == 0, 0, U[i-1][j] + 1)
    for i = n - 1 downto 1 do
        for j = n - 1 downto 1 do
            R[i][j] = iif(M[i][j] == 0, 0, R[i][j+1] + 1)
            D[i][j] = iif(M[i][j] == 0, 0, D[i+1][j] + 1)
    int maxsofar = 0
    for i = 1 to n do
        for j = 1 to n do
            int size = min(L[i][j], R[i][j], U[i][j], D[i][j]) · 4 - 3
            maxsofar = max(maxsofar, size)
    return maxsofar
```

Questa procedura ha costo $\Theta(n^2)$. Tuttavia, esiste anche una soluzione molto semplice che, partendo da ogni posizione (i, j), cerca la croce più grande, senza utilizzare strutture di appoggio.

```
int cross(int[][] M, int n)
    int maxsofar = 0
    for i = 1 to n do
        for j = 1 to n do
            if M[i][j] == 1 then
                int dim = largestCross(M, i, j, n)
                maxsofar = max(maxsofar, dim)
    return maxsofar
```

La procedura largestCross() parte dalla posizione i, j e cerca di espandere la croce in tutte le direzioni, controllando di non uscire dalla matrice e che tutti i quattro bracci abbiano un bit attivo a distanza k.

```
int largestCross(int[][] M, int i, int j, int n)
    int k = 1                                                    % Lunghezza del braccio
    while i + k ≤ n and j + k ≤ n and i − k ≥ 1 and j − k ≥ 1 and M[i][j − k] + M[i][j + k] + M[i − k][j] + M[i + k][j] == 4
     do
        k = k + 1
    return k − 1
```

Questa versione ha costo $\Theta(n^3)$.