Algoritmi e Strutture Dati - 25/07/2018

Esercizio A1

Le tre equazioni di ricorrenza e i relativi parametri α, β sono riassunti nella tabella seguente. Listiamo inoltre il caso in cui ci troviamo, per quanto riguarda il Teorema delle Ricorrenze Lineari con Partizione Bilanciata.

Funzione	α	β	Caso	Complessità
$T_2(n) = 4T_2(n/2) + n^{1.0}$	2.0	1.0	Caso 1: $n^{1.0} = O(n^{\alpha-\epsilon})$	$\Theta(n^2)$
$T_3(n) = 9T_3(n/3) + n^{1.5}$	2.0	1.5	Caso 1: $n^{1.5} = O(n^{\alpha-\epsilon})$	$\Theta(n^2)$
$T_4(n) = 16T_4(n/4) + n^{2.0}$	2.0	2.0	Caso 2: $n^{2.0} = \Theta(n^{\alpha})$	$\Theta(n^2 \log n)$

Visto che stiamo cercando la classe di complessità previsa di ognuna delle funzioni, sarebbe meglio utilizzare la versione estesa del Teorema. Trovandoci tuttavia nei casi 1 e 2, non è necessario provare condizioni aggiuntive.
Scarteremo quindi T_4, che è più costosa per un fattore logaritmico.
Si noti che utilizzando la versione non estesa, potremmo concludere che $T_2(n) = O(n^2)$, $T_3(n) = O(n^2)$, $T_4(n) = O(n^2 \log n)$; la componenente non ricorsiva ci dice che $T_2(n) = \Omega(n)$, $T_3(n) = \Omega(n^{1.5})$, $T_4(n) = \Omega(n^2)$. Quindi questo ci lascerebbe nell'indecisione, perchè nessuno dei limiti inferiori e superiori identifica univocamente una classe di complessità.

Esercizio A2

E' possibile interpretare l'input come un grafo orientato, dove gli n bastoncini sono i nodi e dove esiste un arco dal bastoncino $X[i]$ al bastoncino $Y[i]$, $\forall i : 1 \leq i \leq n$. A questo punto, è possibile rimuovere tutti i bastoncini se e solo se non esistono cicli.

```
shangai(int[] X, int[] Y, int n, int m)
GRAPH G = Graph()
for i = 1 to n do
    G.addNode(i)
for i = 1 to m do
    G.addEdge(X[i], Y[i])
int clock = 0
int[] dt = new int[1...G.n]
int[] ft = new int[1...G.n]
for u = 1 to G.n do
    dt[u] = ft[u] = 0
for u = 1 to G.n do
    if dt[u] == 0 and hasCycle(G, u, clock, dt, ft) then
        return false
return true
```

Utilizzo qui la versione che si trova nei lucidi, che prende esplicitamente dt, ft e $clock$, al fine di essere precisi. Come spesso avviene nel caso delle visite in profondità, non è detto che una singola visita raggiunga tutti i nodi ed è necessario farla ripartire da ogni nodo che non è già stato visitato.
La complessità della costruzione del grafo è $O(m + n)$, pari al costo della visita in profondità realizzata da hasCycle(). Il costo totale è quindi $O(m + n)$.
Ho visto in alcuni compiti soluzioni come la seguente:

```
shangai(int[] X, int[] Y, int n, int m)
GRAPH G = Graph()
for i = 1 to n do
    G.addNode(i)
for i = 1 to m do
    G.addEdge(X[i], Y[i])
return not ciclico(G)
```

in cui si assumeva che la funzione ciclico(G, u) che si trova nel libro abbia un wrapper che applica la funzione all'intero grafo. Ho ritenuto la soluzione accettabile, in quanto in pochi si sono accorti che il problema era facilmente risolvibile e la mancanza del wrapper nel libro è una mia mancanza (lo aggiungerò ai lucidi).

Esercizio A3

Anche questo problema è sostanzialmente un problema su grafo, ma non è necessario trasformarlo in grafo per risolverlo; complicherebbe inutilmente le cose.
L'idea è di simulare una visita BFS a partire da tutte le celle che hanno come valore 1, in modo simile a quanto fatto nell'esercizio 3 del 6/6/2011 (visto normalmente a lezione durante le esercitazioni). Si utilizza la struttura della funzione erdos() vista a lezione, modificata per tener conto che lavoriamo su una matrice e non su un grafo.
I vettori dr e dc rappresentano le differenze delle coordinate delle quattro possibili mosse; i valori nr, nc rappresentano le nuove coordinate una volta che vengono applicate le differenze.
La complessità è pari a $O(n^2)$, in quanto ogni casella entra nella coda al massimo una volta.
Nel codice, abbiamo inserito nella coda coppie di elementi, come per esempio è possibile fare in Python senza troppi problemi. Se questo non piace, è possibile semplicemente inserire ed estrarre dalla coda due valori alla volta (riga, colonna).

```
int grid(int[][] M, int n)
    int[] dr = [−1, 0, +1, 0]                          % Mosse possibili sulle righe
    int[] dc = [0, −1, 0, +1]                          % Mosse possibili sulle colonne
    int[] distance = new int[1 . . . n][1 . . . n]
    QUEUE Q = Queue()
    for r = 1 to n do
        for c = 1 to n do
            distance[r][c] = iif(M[r][c] == 1, 0, −1)
            if M[r][c] == 1 then
                Q.enqueue(⟨r, c⟩)
    while not Q.isEmpty() do
        int, int r, c = Q.dequeue()                    % Riga, colonna della cella visitata correntemente
        for i = 1 to 4 do
            nr = r + dr[i]                             % Nuova riga
            nc = c + dc[i]                             % Nuova colonna
            if 1 ≤ nr ≤ n and 1 ≤ nc ≤ n and distance[nr][nc] < 0 then
                distance[nr][nc] = distance[r][c] + 1
                if M[nr][nc] == 3 then
                    return distance[nr][nc]
                else
                    Q.enqueue(⟨nr, nc⟩)
```

Esercizio B1

Ovviamente, è necessario utilizzare la tecnica backtrack. La struttura è quella di una visita in profondità, in cui però si mantiene un contatore per memorizzare il numero di nodi visitati finora.
Quando il numero di nodi è pari a $k + 1$, il numero di archi visitati (e quindi la lunghezza del cammino) è pari a k e quindi si stampa il cammino.
Per elencare tutti i cammini possibili, evitando comunque di ripetere nodi, si utilizza l'accorgimento di marcare il nodo u come visitato all'inizio della sua visita ($visited[u] =$ **true**), e rimuovere il marcatore ($visited[u] =$ **false**) dopo che la visita è terminata. In questo modo potrà essere visitato di nuovo in altri cammini.
Nel caso di un grafo completo, il numero di cammini lunghi k a partire da s (quindi con k archi e $k + 1$ nodi, di cui però il primo è fisso) è pari a $(n-1) \cdot (n-2) \cdot \ldots (n-k) = \frac{(n-1)!}{(n-1-k)!}$. Questo valore è limitato superiormente da $O(n^k)$, in quanto ad ogni passo potrei scegliere n altri nodi (se non fosse che stiamo cercando cammini semplici). Stampare un cammino lungo k richiede tempo $O(k)$.
La complessità dell'algoritmo, nel caso pessimo di un grafo completo, è quindi $O\left(\frac{(n-1)!}{(n-1-k)!}k\right)$, semplificato con $O(kn^k)$.

```
visit(GRAPH G, int k, int s)
    boolean[] visited = new boolean[1...G.n]
    int[] path = new int[1...G.n]
    visitRec(G, k, s, 1, path, visited)
```

```
visitRec(GRAPH G, int k, NODE u, int i, int[] path, boolean[] visited)
    visited[u] = true
    path[i] = v
    if i == k + 1 then
        print path
    foreach v ∈ G.adj(u) do
        if not visited[v] then
            visitRec(G, k, i + 1, s, path)
    visited[u] = false
```

Esercizio B2

Utilizziamo la programmazione dinamica. Sia $DP[i][k]$ il massimo profitto che si può ottenere considerando i giorni che vanno da i ad n, e avendo un numero totale di azioni pari a k già comprate prima del giorno i-esimo. La soluzione si troverà in $DP[1][0]$.
E' possibile esprimere la soluzione tramite la seguente equazione di ricorrenza:

$$DP[i][k] = \begin{cases} k \cdot V[i] & i = n \\ \mathsf{max}(DP[i+1][k], DP[i+1][k+1] - V[i], DP[i+1][0] + k \cdot V[i]) & \text{altrimenti} \end{cases}$$

In altre parole,

- Se è l'ultimo giorno ($i = n$), l'unica cosa che posso fare è vendere tutte le mie azioni;
- Altrimenti, devo scegliere il massimo fra gli esiti di tre azioni:
  - Posso non fare nulla, quindi nè comprare nè vendere, ottenendo così il guadagno del giorno successivo ($i + 1$) con lo stesso numero di azioni (k);
  - Posso comprare un'azione, riducendo il mio guadagno di un fattore $V[i]$ ma aumentando le azioni in mio possesso in $DP[i + 1][k]$
  - Posso vendere tutte le mie azioni, al prezzo $V[i]$ per ciascuna.

```
int profit(int[] V, int n)
    int[][]DP = new int[1...n][1...n]
    for i = 1 to n do
        for j = 1 to n do
            DP[i][j] = −1
    return profitRec(V, n, 1, 0)
```

```
int profitRec(int[] V, int n, int i, int k, int[][] DP)
    if i == n then
        return V[i] · k
    if DP[i][k] < 0 then
        DP[i][k] = max(profitRec(V, n, i + 1, k, DP)),
                       profitRec(V, i + 1, k + 1, DP) − V[i],
                       profitRec(V, i + 1, 0, DP) + k ∗ V[i])
    return DP[i][k]
```

Poichè è possibile comprare al più $n-1$ azioni, al limite è necessario riempire tutta la tabella e il costo è pari a $O(n^2)$.

Esercizio B3

Anche in questo caso, utilizziamo programmazione dinamica. Sia $DP[i][r]$ il numero di modi con cui è possibile ottenere r utilizzando le prime i cifre dell'input. Tale valore può essere calcolato ricorsivamente come segue:

$$DP[i][r] = \begin{cases} 1 & i = 0 \wedge r = 0 \\ 0 & i > 0 \wedge r \leq 0 \\ 0 & i = 0 \wedge r > 0 \\ \sum_{s=1}^{i} DP[s-1][r - \mathsf{value}(V, s, i)] & \text{altrimenti} \end{cases}$$

In altre parole:

- Il numero di modi per ottenere un valore nullo non avendo termini da sommare è 1;
- Il numero di modi per ottenere un valore negativo o nullo sommando termini positivi è 0;
- Il numero di modi per ottenere un valore positivo non avendo termini da sommare è 0;
- Altrimenti, spezziamo le i cifre rimanenti in due parti: $V[1 \dots s-1]$ e $V[s \dots i]$. Interpretiamo le cifre fra s ed i come un addendo e applichiamo ricorsivamente la sommatoria alle cifre fra 1 ed $s-1$. Dobbiamo sommare su tutti i possibili valori di s, da 1 ad i.

Il risultato che stiamo cercando si trova nella posizione $DP[n][k]$.

Traduciamo questa formula ricorsiva in algoritmo basato su memoization.

```
countSum(int[] V, int k)
    int[][] DP = new int[1...n][1...k]
    for i = 1 to n do
        for r = 1 to k do
            DP[i][r] = −1
    return countRec(V, n, k, DP)
```

```
countSum(int[] V, int i, int r, int[][] DP)
    if i == 0 and r == 0 then
        return 1
    if (i == 0 and r ≤ 0) or (i == 0 and r > 0) then
        return 0
    if DP[i][r] < 0 then
        DP[i][r] = 0
        for s = 1 to i do
            DP[i][r] = DP[i][r] + countRec(V, s − 1, r − value(V, s, i))
    return DP[i][r]
```

Il costo è molto elevato. La tabella ha dimensione $n \times k$, dove $k = O(n)$. Per riempire ogni cella della tabella, sono necessari $O(n^2)$ passi. Quindi il costo totale è $O(n^4)$. E' possibile pre-calcolare tutti gli $O(n^2)$ possibili addendi in tempo $O(n^2)$, riducendo il costo a $O(n^3)$.
Si noti comunque che la complessità è inferiore a quella ottenibile da un algoritmo di backtrack, vista nel compito scorso, che è esponenziale nel numero delle cifre.

Brutte cose

Cose che non si possono vedere:

```
foreach v ∈ G.adj(u) do
    if visited[u] then
        continue
    [...]
```

al posto di:

```
foreach v ∈ G.adj(u) do
    if not visited[u] then
        [...]
```

Se scrivete qualcosa del genere, io non abbasso il voto, ma sappiate che Sebastiani possiede una mannaia....