Algoritmi e Strutture Dati - 04/06/2018

Esercizio A1

Stiamo analizzando la seguente funzione di ricorrenza:

$$T_a(n) = \begin{cases} 7T_a(n/3) + T_a(n/a) + n^2 \log n & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Visto che stiamo considerando valori interi di a, la prima cosa da notare è che $a = 1$ non ha senso, in quanto $T(n)$ richiamerebbe $T(n)$.
Consideriamo $a = 2$; proviamo a dimostrare che $T_2(n) = 7T_2(n/3) + T_2(n/2) + n^2 = O(n^2 \log n)$. Dobbiamo quindi dimostrare che:

$$\exists c > 0, \exists m \geq 0 : T_2(n) \leq cn^2 \log n$$

Proviamo:

$$\begin{aligned} T_2(n) &= 7T_2(n/3) + T_2(n/2) + n^2 \log n \\ &\leq 7/9\, cn^2 \log n/3 + 1/4\, cn^2 \log n/2 + n^2 \log n \\ &= 7/9\, cn^2(\log n - \log 3) + 1/4\, cn^2(\log n - \log 2) + n^2 \log n \\ &\leq 7/9\, cn^2 \log n + 1/4\, cn^2 \log n + n^2 \log n \\ &= 37/36\, cn^2 \log n + n^2 \log n \\ &\leq cn^2 \log n \end{aligned}$$

L'ultima disequazione è vera se $37/36\, c + 1 \leq c$, ovvero se $c \leq -36$; ma questo non è accettabile, in quanto c deve essere positivo.
Consideriamo $a = 3$; dobbiamo dimostrare che

$$T_3(n) = 7T_3(n/3) + T_3(n/3) + n^2 \log n = 8T_3(n/3) + n^2 \log n = O(n^2 \log n)$$

Qui si può applicare il Master Theorem, versione estesa; $\alpha = \log_3^8 < 2$; abbiamo quindi che $n^2 \log n = \Omega(n^{\alpha+\epsilon})$ con $\epsilon > 0$ e siamo quindi nel terzo caso. In questo caso, bisogna anche dimostrare che:

$$\exists c < 1, \exists m > 0 : 8(n/3)^2 \log n/3 \leq cn^2 \log n, \forall n \geq m$$

Abbiamo quindi che

$$\begin{aligned} af(n/b) &= 8(n/3)^2 \log n/3 \\ &= 8/9 n^2 \log n/3 \\ &\leq 8/9 n^2 \log n \leq cn^2 \log n \end{aligned}$$

che è vera per ogni $n \geq m = 1$ e per $c \geq 8/9$.
Abbiamo quindi dimostrato che $T_3(n) = \Theta(n^2 \log n)$.
Si poteva fare anche per sostituzione, facendo attenzione ai casi base.
Ora, è possibile notare che:

$$T_a(n) = 7T_a(n/3) + T_a(n/a) \leq 7T_3(n/3) + T_3(n/3) = T_3(n)$$

per $a \geq 3$. Otteniamo quindi che $T_a(n) = O(T_3(n)) = O(n^2 \log n)$ per qualunque valore di $a \geq 3$.

Digressione Dal punto di vista algoritmico, questo esercizio è potenzialmente interessante. Si consideri ad esempio l'algoritmo deterministico per il calcolo delle mediane, che genera un'equazione di ricorrenza simile a quella di questo esercizio. In quel caso, i numeri coinvolti derivano dalla scelta della dimensione dei bucket su cui calcolare la mediana e poi la mediane delle mediane. Uno potrebbe domandarsi se altri numeri danno origine a costi computazionali più alti o più bassi.
Dal punto di vista matematico, ci si potrebbe domandare qual è il più piccolo valore reale di a che fa sì che l'equazione abbia un limite superiore $O(n^2 \log n)$; questo è pari a $\sqrt{9/2} \approx 2.12$.
Questo non avrebbe alcun interesse dal punto di vista algoritmico, tuttavia, perchè difficilmente si va da a dividere il vettore per un valore reale.

Esercizio A2

Il problema della colorazione per grafi generali è NP-completo, un concetto che abbiamo visto nella seconda parte del corso. Un albero binario, tuttavia, non è un grafo generale. Innanzitutto è bipartito: organizzando l'albero per livelli, è possibile dividere i nodi in due gruppi: nodi di livello pari e nodi di livello dispari. Essendo bipartito, è 2-colorabile. Questi concetti sono stati affrontati invece nella prima parte, parlando delle visite dei grafi.
Poichè l'albero è bi-colorabile, ne viene che qualunque colorazione con più di 2 colori è necessariamente sostituibile con una colorazione di 2 colori, potenzialmente a costo inferiore.
Un modo semplice e facile da realizzare è il seguente: proviamo a colorare la radice con 1 e poi con 2 e restituiamo il costo minore fra i due casi.
Organizziamo il codice tramite due funzioni: una funzione che calcola il costo di colorazione di un albero a partire dal colore della radice, e una funzione wrapper che restituisce il costo della colorazione minima fra le due.
Utilizziamo il trucchetto $3 - color$ per passare da $color = 1$ a 2 e viceversa.
La complessità dell'algoritmo risultante è $O(n)$, in quanto corrisponde ad una visita dell'albero.

int minColoring(TREE T)

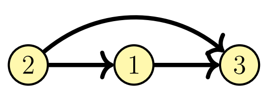
  return min(countRec($T, 1$), countRec($T, 2$))

int countRec(TREE t, **int** $color$)

  if t == **nil then**
    return 0
  return $color$ + countRec(t.left(), $3 - color$) + countRec(t.right(), $3 - color$)

Esercizio A3

A primo acchito, può spaventare il fatto che il grafo contenga cicli. Se 1 batte 2, 2 batte 3 e 3 batte 1, chi vince? Semplice, dipenderà dall'ordine in cui vengono giocate le partite. Supponiamo di voler far vincere 1: giocheremo prima 2 contro 3, eliminiamo 3, poi 1 contro 2, eliminiamo 2 e 1 rimane l'unica squadra in campo.
In quale situazione non è possibile far vincere l'Italia? Sicuramente nel caso in cui il nodo Italia non abbia archi uscenti, ma questa non è l'unica situazione in cui l'Italia non può vincere. Ad esempio, si consideri il grafo seguente:



Il nodo 1 può battere 3, ma non c'è modo che 1 batta 2.
Intuitivamente, è possibile vedere che si può far vincere l'Italia se e solo se ogni altro nodo è raggiungibile dall'Italia nel grafo dei pronostici . Significa che l'Italia può battere qualunque altro nodo o direttamente o "transitivamente".
È possibile dimostrare tale proprietà in maniera formale (tale precisione non è richiesta nel compito):

- Se ogni altro nodo è raggiungibile dall'Italia, utilizziamo questa procedura costruttiva per stabilire l'ordine: effettuiamo una visita DFS posticipata, tale per cui tutte le volte che da un nodo u si scopre un nodo v (tramite un arco (u, v)), si continua ricorsivamente la visita nel nodo v, e poi si aggiunge la partita (u, v) alla lista delle partite. Questo è un ordine in cui l'Italia resta l'unica squadra, perchè è già scoperta all'inizio della visita e non può quindi essere battuta, e tutte le squadre verranno scoperte e battute.

- Se è possibile far vincere l'Italia, allora esiste un ordinamento di $n - 1$ archi in cui l'Italia vince l'ultima partita. Dobbiamo dimostrare che ogni altro nodo u è raggiungibile dall'Italia. Sono date due possibilità: u_1 può essere stato battuto direttamente dal nodo s (Italia), nel qual caso esiste un arco $(s, u_1) \in E$ e quindi u_1 è raggiungibile da s. Oppure u_1 è stato battuto da un'altra squadra u_2 e quindi $(u_2, u_1) \in E$. Il ragionamento viene applicato ricorsivamente su u_2; u_2 è stato battuto dall'Italia, nel qual caso gli archi (s, u_2) e (u_2, u_1) formano un cammino da s a u_1, e quindi u_1 è raggiungibile da u_1; oppure u_2 è stato battuto da u_3. Il ragionamento si ripete, o trovando un nodo intermedio battuto dall'Italia, o arrivando a includere tutti gli $n - 1$ nodi diversi dall'Italia nella catena di archi $(u_{n-1}, u_{n-2}), (u_{n_2}, u_{n-3}, \ldots, (u_2, u_1))$. Poichè u_{n-1} e l'Italia sono gli ultimi due nodi, ed è possibile far vincere l'Italia, (s, u_{n-1}) è l'ultimo arco nell'ordinamento, e ogni nodo u_i è raggiungibile dall'Italia.

Il problema base richiede una risposta **true/false** e non di stabilire l'ordine; scriviamo quindi una soluzione che verifichi se tutti i nodi sono raggiungibili dal nodo s (Italia). Per semplificare il codice, possiamo utilizzare la funzione ccdfs() definita nel Capitolo 9, che effettua una visita e scrive l'identificatore (s) che gli viene passato. Possiamo poi utilizzare la somma parziale definita nel primo set di slide, e verificare se tutti i nodi sono stati visitati. Il costo computazionale è $O(m + n)$, corrispondente ad una visita del grafo.

```
canItalyWin(GRAPH G, int s)
  int[] visited = new int[1 ... G.n]
  foreach u ∈ G.V() do
    visited[u] = 0
  ccdfs(G, s, 1, visited)
  return sum(visited, 1, G.n) == G.n
```

Esercizio A3 - Bonus

Assumendo che tale ordinamento esista, ci viene chiesto di stampare l'ordine delle partite. Questo può essere risolto con il metodo suggerito nella dimostrazione costruttiva qui sopra. La complessità è ancora una volta $O(m + n)$.

```
printOrderRec(GRAPH G, int s)
  int[] visited = new boolean[1 ... G.n]
  foreach u ∈ G.V() do
    visited[u] = false
  printOrderRec(G, s, visited)
```

```
printOrderRec(GRAPH G, NODE u, boolean[] visited)
  visited[u] = true
  foreach v ∈ G.adj(u) do
    if not visited[v] then
      printOrderRec(G, v, visited)
      print (u, v)
```

Esercizio B1

Questo esercizio è simile ad un esercizio che abbiamo visto in passato. Sia $DP[i][j]$ il numero di caratteri necessari per rendere palindroma la stringa $S[i \ldots j]$.
Ovviamente, se $S[i \ldots j]$ ha zero o un carattere ($i \geq j$), non è necessario alcuna operazione per rendere la stringa palindroma. Altrimenti, possiamo

- rimuovere il carattere i e considerare la sottostringa $S[i + 1 \ldots j]$ (al costo di una operazione);
- rimuovere il carattere j e considerare la sottostringa $S[i \ldots j - 1]$ (al costo di una operazione);
- Se $S[i] = S[j]$, ignoriamo questi due caratteri e consideriamo la stringa $S[i + 1 \ldots j - 1]$ (senza costi aggiuntivi)
- Se $S[i] \neq S[j]$, cambiamo $S[i]$ in $S[j]$ e consideriamo la stringa $S[i + 1 \ldots j - 1]$ (al costo di una operazione)

È possibile esprimere in maniera compatta questa idea tramite la seguente formula ricorsiva:

$$DP[i][j] = \begin{cases} 0 & i \geq j \\ \min\{DP[i+1][j] + 1, & i < j \\ \qquad DP[i][j-1] + 1, & \\ \qquad DP[i+1][j-1] + \mathsf{iif}(S[i] = S[j], 0, 1)\} & \end{cases}$$

Tradotta in codice tramite memoization, abbiamo:

```
minPalindrome(ITEM[] S, 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 minPalRec(S, 1, n, DP)
```

```
minPalRec(ITEM[] S, int i, int j, int[][] DP)
    if i ≥ j then
        return 0
    if DP[i][j] < 0 then
        DP[i][j] = min(minPalRec(S, i + 1, j, DP) + 1,
                       minPalRec(S, i, j − 1, DP) + 1,
                       minPalRec(S, i + 1, j − 1, DP) + iif(S[i] == S[j], 0, 1))
    return DP[i][j]
```

La complessità è pari a $O(n^2)$, il tempo necessario per inizializzare e riempire una matrice $n \times n$.

Esercizio B2

L'esercizio va affrontato con la tecnica del backtrack. L'idea è la seguente: Alternativamente, il segno "+" può apparire, oppure no, prima di ogni carattere, tranne il primo. È quindi necessario generare tutte le possibili sequenze composte da $n-1$ valori booleani, e utilizzarle per inframezzare i valori numerici con i segni $+$.

```
printAllRec(int[] S, int n, boolean[] stop, int i)
    if i == 1 then
        print S[1]                          % Print the sum
        for i = 2 to n do
            if stop[i] then
                print "+"
            print S[i]
        println
    else
        foreach c ∈ {true, false} do
            stop[i] = c
            printAllRec(S, n, stop, i − 1)
```

```
printAll(int[] S, int n)
    int[] stop = new boolean[1...n]
    printAllRec(S, n, stop, n)
```

La complessità dell'algoritmo proposto è pari a $\Theta(n \cdot 2^n)$, che deriva da 2^{n-1} possibili combinazioni più il tempo $O(n)$ per stampare ognuna di esse.

Esercizio B3

La definizione è complessa, la soluzione è molto semplice. È possibile ottenere un albero di copertura k-minimale ignorando tutti gli archi con peso inferiore a k e applicando uno degli algoritmi per l'albero di copertura minima visti a lezione, ad esempio Kruskal. Invece di mantenere informazioni sull'albero, l'algoritmo memorizza semplicemente il valore massimo e minimo degli archi che verrebbero aggiunti. Al termine ritorna la differenza fra questi due valori.

La versione proposta qui assume che A sia già ordinato, prende k in input e restituisce la snellezza dell'albero k-minimale.

```
int kMimimal(EDGE[] A, int k, int n, int m, int k)
    { ordina A[1,...,m] in modo che A[1].peso ≤ ··· ≤ A[m].peso }
    int min = +∞
    int max = −∞
    MFSET M = Mfset(n)
    int c = 0
    while c < n − 1 and i ≤ m do                      % Termina quando l'albero è costruito
        if A[i].peso ≥ k and M.find(A[i].u) ≠ M.find(A[i].v) then
            M.merge(A[i].u, A[i].v)
            max = max(max, A[i].peso)
            min = min(min, A[i].peso)
            c = c + 1
        i = i + 1
    return iif(i > m, +∞, max − min)
```

La complessità di tale algoritmo è assolutamente identica a quella di Kruskal, ovvero $O(m \log n)$.

Esercizio B3 - Bonus

È molto semplice scrivere una versione che chiama l'algoritmo precedente su tutti i possibili valori di k, fino a quando kMinimal() non restituisce $+\infty$, segno che gli archi che sono rimasti non possono formare un albero di copertura.

```
int thinnest(EDGE[] A, int n, int m)
    { ordina A[1,...,m] in modo che A[1].peso ≤ ··· ≤ A[m].peso }
    int k = 1
    int last = 0
    int minSoFar = +∞
    while last ≠ +∞ do
        last = kruskal(A, n, m, k)
        minSoFar = min(minSoFar, last)
    return minSoFar
```

L'ordinamento degli archi può essere fatto una volta sola, all'esterno di kMinimal(). Se M è il peso massimo, il costo di tale algoritmo è $O(Mm + m \log n)$. Tale algoritmo è pseudo-polinomiale, in quanto M non è la dimensione dell'input, ma fa parte dell'input.
Vediamo invece una versione basata sull'algoritmo di Kruskal, che elimina mano a mano gli archi di peso crescente e ricalcola l'albero di copertura minima.
Per prima cosa, ordiniamo gli archi, così come viene fatto da Kruskal. Questo ha costo $O(m \log n)$. L'indice i viene fatto scorrere da 1 a $m - n + 1$ e costituisce il primo arco che viene considerato. Si cercano tutti i valori distinti dei pesi degli archi, utilizzando la variabile *prev* per evitare di ripetere più volte il calcolo per gli stessi valori di k.

```
int thinnest(EDGE[] A, int n, int m)
    { ordina A[1,...,m] in modo che A[1].peso ≤ ··· ≤ A[m].peso }
    int minSoFar = +∞
    int prev = −1
    for i = 1 to m − n + 1 do
        if prev ≠ A[i] then
            minSoFar = min(minSoFar, kruskal(A, n, m, i))
            prev = A[i]
    return minSoFar
```

La versione di Kruskal che segue prende in input l'indice i da cui iniziare a guardare gli archi, e restituisce la snellezza invece che restituire l'albero di copertura minima.

```
int kruskal(EDGE[] A, int n, int m, int i)
    int min = A[i].peso
    int max = 0
    MFSET M = Mfset(n)
    int c = 0
    while c < n − 1 and i ≤ m do                    % Termina quando l'albero è costruito
        if M.find(A[i].u) ≠ M.find(A[i].v) then
            M.merge(A[i].u, A[i].v)
            max = A[i].peso
            c = c + 1
        i = i + 1
    return iif(i > m, +∞, max − min)
```

Dando l'ordinamento per scontanto, Kruskal ha costo $O(m)$. Poichè viene ripetuto m volte, il costo dell'algoritmo è pari $O(m^2)$, un costo comunque molto alto.