Algoritmi e Strutture Dati – 17/06/2019

Esercizio A1

Limite superiore Andando per tentativi, proviamo con $\Theta(n^2)$. Proviamo quindi a dimostrare che $T(n) = O(n^2)$.

- Ipotesi induttiva: $T(k) \leq ck^2$, per $k < n$
- Passo induttivo:

$$\begin{aligned} T(n) &= 2T(\lfloor n/3 \rfloor) + 3T(\lfloor n/2 \rfloor) + n^2 \\ &\leq 2c\lfloor n/3 \rfloor^2 + 3c\lfloor n/2 \rfloor^2 + n^2 \\ &\leq \frac{2}{9}cn^2 + \frac{3}{4}cn^2 + n^2 \\ &= \frac{35}{36}cn^2 + n^2 \\ &\leq cn^2 \end{aligned}$$

  L'ultima disequazione è vera per $c \geq 36$.

- Casi base:

$$\begin{aligned} T(1) = 1 \leq c \cdot 1^2 \Rightarrow c \geq 1 \\ T(2) = 1 \leq c \cdot 2^2 \Rightarrow c \geq 1/4 \end{aligned}$$

Abbiamo quindi dimostrato che $T(n) = O(n^2)$, con $m = 1$ e $c \geq 36$.
È facile dimostrare che $T(n) = \Omega(n^2)$ per via della componente non ricorsiva n^2.
Abbiamo quindi dimostrato che $T(n) = \Theta(n^2)$.

Esercizio A2

Per risolvere questo esercizio, è possibile utilizzare una singola visita in profondità dell'albero, che però raccoglie due informazioni separate.
Per ogni nodo u, vogliamo misurare:

- il costo più alto per andare da u ad una qualunque delle sue foglie (*highest*);
- il costo massimo fra tutti i cammini semplici contenuti nel sottoalbero radicato in u che uniscono due foglie (*maxpath*).

Calcolare *highest* di un nodo u è semplice: basta prendere il massimo fra il valore *highest* dei figli sinistro e destro, e sommare il peso $u.weight$. Se u è **nil**, si restituisce 0.
Per calcolare *maxpath* di un albero radicato in u, si prende il massimo fra:

- Il *maxpath* del sottoalbero sinistro;
- Il *maxpath* del sottoalbero destro;
- Il costo del cammino massimale che passa attraverso il nodo u, calcolato come il costo più alto per andare da u.left() ad una delle sue foglie, da u.right() ad una delle sue foglie, più il peso del nodo u.

Ci sono alcuni casi particolari:

- Nel caso u == **nil**, si restituisce $-\infty$ per entrambi i valori; si assume che in questo modo, se un nodo u non ha figlio destro oppure sinistro, la somma $highest_L + highest_R + T.weight$) darà comunque $-\infty$ e non verrà selezionata dalla funzione max;

- Nel caso u sia una foglia, si restituisce il peso del nodo sia come *highest* che come *maxpath*.

In ogni caso, il codice può essere reso più chiaro "sviscerando" ognuno dei quattro casi: **nil**, foglia, solo figlio sinistro, solo figlio destro, due figli.

```
int maxPath(TREE T)
    highest, maxpath = maxPathRec(T)
    return maxpath
```

```
(int, int) maxPathRec(TREE T)
    if T == nil then
        return (−∞, −∞);
    if T.left() == nil and T.right() == nil then
        return (T.weight, T.weight);
    highest_L, maxpath_L = maxPath(T.left())
    highest_R, maxpath_R = maxPath(T.right())
    highest = max(highest_L, highest_R) + T.weight
    maxpath = max(maxpath_L, maxpath_R, highest_L + highest_R + T.weight)
    return highest, maxpath
```

La complessità dell'algoritmo proposto è quella di una visita in profondità di un albero, ovvero $\Theta(n)$.

Esercizio A3

In modo simile al compito dell'Agosto 2018, il problema richiede di individuare le componenti connesse nel grafo dove le caselle adiacenti sono connesse da un arco. In questo caso, è sufficiente individuare la sola componenente connessa che contiene la cella di input (r, c).
Per risolvere il problema, viene fatta una visita in profondità a partire dalla cella (r, c), progettata in modo tale che restituisca il numero di celle della linea costiera che sono state incontrate durante la visita.
Quando viene visitata una cella (r, c) valida che non è stata visitata in precedenza, la cella viene marcata come visitata; possono darsi due casi:

- Se è una cella di mare ($M[r][c] < 0$), si visitano tutti i nodi adiacenti a partire da essa (up, down, left, right) e si restituisce la somma delle celle di linea costiera che sono state incontrate
- Se è una cella di terra ($M[r][c] \geq 0$), si restituisce 1 perché è stata visita a partire da una cella di mare e quindi fa parte della linea costiera.

L'algoritmo ha costo $O(n^2)$, per via dell'inizializzazione di *visited* e per il fatto che ogni cella viene visitata al più una volta.

```
int coastLen(int[][] M, int n, int r, int c)
    boolean[][] visited = new boolean[0...n − 1][0...n − 1]
    for r = 0 to n − 1 do
        for c = 0 to n − 1 do
            visited[r][c] = false
    return coastDFS(M, visited, n, r, c)
```

```
int coastDFS(int[][] M, boolean[][] visited, int n, int r, int c)
    if r < 0 or r ≥ n or c < 0 or c ≥ n or visited[r][c] then
        return 0
    visited[r][c] = true
    if M[r][c] < 0 then
        int t_U = coastDFS(m, visited, r − 1, c)
        int t_D = coastDFS(m, visited, r + 1, c)
        int t_L = coastDFS(m, visited, r, c − 1)
        int t_R = coastDFS(m, visited, r, c + 1)
        return t_U + t_D + t_L + t_R
    else
        return 1
```

Esercizio B1

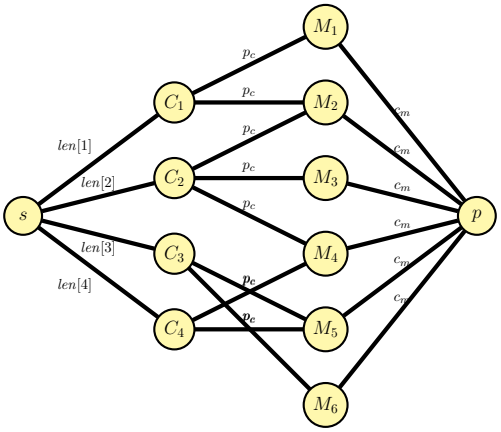
È possibile risolvere questo esercizio costruendo un'opportuna rete di flusso $G = (V, E, s, p, c)$. Per quanto riguarda l'insieme V:

- Si creano un nodo sorgente s e un nodo pozzo p;
- Si creano c nodi, uno per ogni cantiere;
- Si creano m nodi, uno per ogni mese.

Per quanto riguarda l'insieme E:

- Si aggiunge un arco fra la sorgente e ogni cantiere i, con capacità $len[i]$ (ad indicare che il progetto i richiede $len[i]$ mesi/persona).
- Si aggiunge un arco fra ogni progetto i e ogni mese incluso fra $start[i]$ e $end[i]$, con capacità p_c, ad indicare che ogni cantiere può occupare al massimo p_c persone durante un mese;
- Si aggiunge un arco fra ogni mese j e il pozzo, con capacità p_m, ad indicare che ogni mese sono disponibili al più p_m persone.

Sia $L = \sum_i^c len[i]$, ovvero il numero totale di mesi/persona che devono essere impegnati dal progetto. Si calcoli il flusso massimo sulla rete di flusso; se il flusso massimo è pari a L, è possibile completare tutti i cantieri seguendo le regole. Infatti, ogni cantiere riceverà il numero corretto di mesi/persona (primo strato); ad ogni cantiere non potranno lavorare più di p_c persone per mese (secondo strato); in ogni mese non potranno lavorare più di p_m persone (terzo strato).
Poiché $|V| = c+m+2$ e $E = c+m+O(cm)$, e poiché il flusso è limitato superiormente da L, l'algoritmo avrà costo $O((|V| + |E|)L) = O(cmL)$ secondo il limite superiore di Ford-Fulkerson, $O((c + m)(c^2m^2))$ secondo Edmonds e Karp.



Esercizio B2

Si può risolvere il problema utilizzando programmazione dinamica.

Soluzione $\Theta(n^3)$

La domanda su pseudo-polinomialità vi potrebbe indurre a pensare che debba essere risolto con una tabella di programmazione dinamica a tre indici: sia $DP[i][j][k]$ uguale a **true** se e solo se la sottostringa $S[i \dots j]$ è k-palindroma. La tabella DP può essere calcolata nel modo seguente:

$$DP[i][j][k] = \begin{cases} \textbf{false} & k < 0 \\ \textbf{true} & j \leq i \textbf{ and } k \geq 0 \\ S[i] = S[j] \wedge DP[i+1][j-1][k] \textbf{ or} & \\ DP[i+1][j][k-1] \textbf{ or} & \\ DP[i][j+1][k-1] & i < j \textbf{ and } k \geq 0 \end{cases}$$

- Caso 1: se ho effettuato troppe rimozioni e $k < 0$, restituisco **false**.
- Caso 2: Nel caso in cui la stringa sia composta da 1 carattere ($j = i$) o 0 caratteri ($j < i$) e $k \geq 0$, allora la stringa è palindroma.
- Caso 3: Altrimenti, possono darsi tre sottocasi
  - I caratteri $S[i], S[j]$ sono uguali; la stringa $S[i \dots j]$ è k-palindroma se $S[i+1 \dots j-1]$ è k-palindroma;
  - Altrimenti, si elimini il carattere i-esimo e si verifichi se $S[i+1 \dots j]$ è $k-1$-palindroma;
  - Oppure, si elimini il carattere j-esimo e si verifichi se $S[i \dots j-1]$ è $k-1$-palindroma.

L'algoritmo può essere scritto utilizzando memoization. Utilizziamo il valore -1 per indicare una casella non ancora calcolata, 0 e 1 per indicare vero e falso, rispettivamente.

```
boolean k-palindrome(ITEM[] S, int n, int k)
    DP = new int[1...n][1...n][1...k]
    k = min(n, k)
    for i = 1 to n do
        for j = 1 to n do
            for ℓ = 1 to k do
                DP[i][j][ℓ] = −1
    return kpalRec(S, 1, n, k) == 1
```

```
int kpalRec(ITEM[] S, int i, int j, int k)
    if k < 0 then
        return 0
    if k ≥ 0 and j ≤ i then
        return 1
    if DP[i][j][k] < 0 then
        DP[i][j][k] = (S[i] == S[j] and kpalRec(S, i + 1, j − 1, k)) or
                      kpalRec(S, i + 1, j, k − 1) or
                      kpalRec(S, i, j − 1, k − 1)
    return DP[i][j][k]
```

La complessità è $O(kn^2)$; essendo k limitato superiormente da n, possiamo dire che l'algoritmo è $O(n^3)$, il che lo rende polinomiale.

Soluzione $\Theta(n^2)$

Ma si può fare meglio di così: sia $DP[i][j]$ il numero minimo di caratteri che devono essere cancellati per rendere la stringa palindroma. La tabella DP può essere calcolata nel modo seguente:

$$DP[i][j] = \begin{cases} 0 & j \leq i \\ min(DP[i+1][j], DP[i][j-1]) + 1 & i < j \wedge S[i] \neq S[j] \\ DP[i+1][j-1] & i < j \wedge S[i] = S[j] \end{cases}$$

- Caso 1: Nel caso in cui la stringa sia composta da 1 carattere ($j = i$) o 0 caratteri ($j < i$), allora la stringa è palindroma e il numero minimo di cancellazioni è 0.
- Casi 2,3: Altrimenti, possono darsi due sottocasi

– I caratteri $S[i], S[j]$ sono uguali; il numero minimo di caratteri è pari al minimo fra il numero minimo di caratteri per rendere palindroma $S[i \dots j-1]$ o $S[i+1 \dots j]$.
– I caratteri $S[i], S[j]$ sono uguali; il numero minimo di caratteri da cancellare per rendere palindroma la stringa $S[i \dots j]$ è pari al numero di caratteri per rendere palindroma la stringa $S[i+1 \dots j-1]$;

```
boolean k-palindrome(ITEM[] S, int n, int k)
    DP = new int[1...n][1...n]
    for i = 1 to n do
        for j = 1 to n do
            DP[i][j] = −1
    return kpalRec(S, 1, n) ≤ k
```

```
int kpalRec(ITEM[] S, int i, int j)
    if j ≤ i then
        return 0
    if DP[i][j] < 0 then
        if S[i] == S[j] then
            DP[i][j] = kpalRec(S, i + 1, j − 1)
        else
            DP[i][j] = min (kpalRec(S, i + 1, j), kpalRec(S, i, j − 1))) + 1
    return DP[i][j]
```

Questo algoritmo ha complessità $O(n^2)$.

Soluzione $O(2^n)$

Notate che mi capita spesso di vedere codice come questo:

```
int kpalRec(ITEM[] S, int i, int j)
    if j ≤ i then
        return 0
    if S[i] == S[j] then
        return kpalRec(S, i + 1, j − 1)
    else
        return min(kpalRec(S, i + 1, j), kpalRec(S, i, j − 1)) + 1
```

Notate che manca il controllo sul valore -1 necessario per la memoization? Questo trasforma la soluzione da polinomiale a superpolinomiale, perché lo stesso problema può essere calcolato più volte; è formalmente corretta, ma estremamente inefficiente. Per questo prende normalmente 60%.

Esercizio B3

Il problema corrisponde a identificare almeno un cammino hamiltoniano che inizia in un certo nodo, e si risolve tramite un algoritmo di backtrack.
L'algoritmo sfrutta il template degli algoritmi basati su backtrack visti a lezione. Utilizza un vettore *visited* per memorizzare i nodi già visitati nel cammino attuale; a differenza di una semplice visita, quando si effettua un'operazione di backtrack si cancella la visita dal vettore.

```
int[] safeLine(GRAPH G, int u)
    boolean[] visited = new int[1...G.n]
    for i = 1 to n do
        visited[i] = false
    int[] S = new int[1...G.n]
    return safeLineRec(G, u, visited, S, 1)
```

```
int[] safeLineRec(GRAPH G, int u, boolean[] visited, int[] S, int i)
    S[i] = u
    if i == G.n then
        return S
    else
        visited[u] = true
        foreach v ∈ G.adj(u) : visited[v] == false do
            int[] res = safeLineRec(G, v, visited, S, i + 1)
            if res ≠ nil then
                return res
        visited[u] = false
        return nil
```

La complessità è $O(n!)$, in quanto potenzialmente bisogna considerare tutte le le $(n-1)!$ permutazioni di $n-1$ nodi (il primo è fisso).