Algoritmi e Strutture Dati – 24/08/2022

Esercizio A1 – Complessità – Punti ≥ 8

È facile vedere che $T(n)$ è $\Omega(n)$, per via della parte non ricorsiva. È possibile fare un passo in più e osservare che

$$T(n) = 2T(n/2) + 2T(n/3) + n \geq 2T(n/2) + n = \Theta(n \log n)$$

Per via del $\geq$, possiamo dedurne che $T(n) = \Omega(n \log n)$.
In alternativa, possiamo osservare che

$$T(n) = 2T(n/2) + 2T(n/3) + n \geq 4T(n/3) + n = \Theta(n^{\log_3 4})$$

Per via del $\geq$, possiamo dedurne che $T(n) = \Omega(n^{\log_3 4}) \approx \Omega(n^{1.26})$.
Per via dei limiti inferiori, non perdiamo tempo a provare che $T(n) = O(n)$. Notiamo invece che

$$T(n) = 2T(n/2) + 2T(n/3) + n \leq 4T(n/2) + n = \Theta(n^2)$$

Per via del $\geq$, possiamo dedurne che $T(n) = O(n^2)$.
Questo è perfettamente dimostrabile anche con il metodo di sostituzione.

Esercizio A2 – DAG massimale – Punti ≥ 11

Si consideri un ordinamento topologico del grafo. Qualunque esso sia, il primo nodo può avere al massimo $n-1$ archi; il secondo $n-2$, fino all'ultimo, che ne ha 0. Quindi, il numero massimo di archi che possono essere contenuti in un grafo orientato che sia anche aciclico, è pari a:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Il grafo in input ha m archi già presenti; quindi il numero totale di archi che possono essere aggiunti è pari a:

$$\frac{n(n-1)}{2} - m$$

Poichè l'API del libro non fornisce il numero di archi del grafo, siamo costretti a contarli in tempo $O(n)$; se questa informazione fosse memorizzata nella struttura dati, l'algoritmo opererebbe in tempo costante.

```
int maxDag(Graph G)
    int n = 0
    int m = 0
    for u ∈ G.V() do
        n = n + 1
        m = m + |G.adj(u)|
    return (n · (n − 1)/2) − m
```

Esercizio A3 – Somma su albero – Punti ≥ 11

Il problema può essere risolto utilizzando un insieme di supporto S, implementato tramite tabella hash. Si effettua una visita in profondità dell'albero, e per ogni nodo con valore x si verifica se $k - x$ è presente nell'insieme. Se non è presente, si inserisce x in S.

```
boolean containsSum(Tree T, int k)
    Set S = Set()
    return visit(T, S, k)
```

```
boolean visit(Tree T, Set S, int k)
    if T == nil then
        return false
    else
        if S.contains(k − T.value) then
            return true
        else
            S.insert(T.value)
            return visit(T.left) or visit(T.right)
```

La complessità è quella di una visita in profondità, $O(n)$.

Esercizio B1 – Esami – Punti ≥ 8

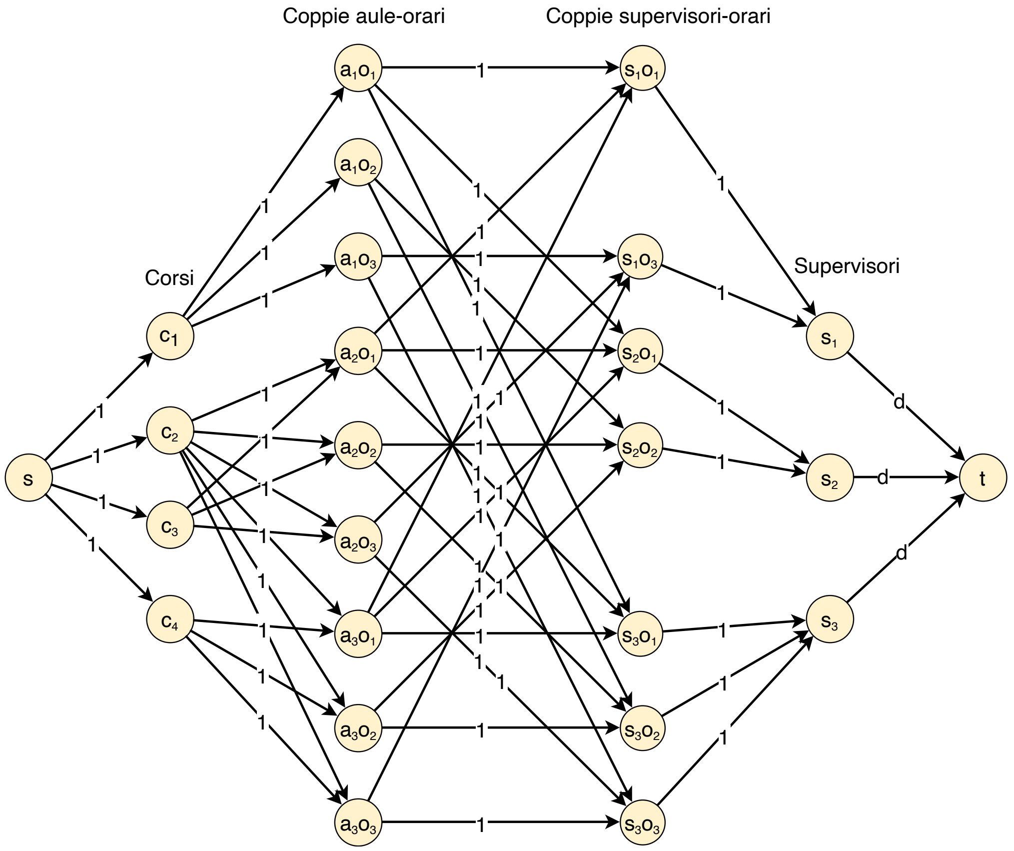
È possibile risolvere il problema tramite una rete di flusso appositamente progettata.
Costruiamo un insieme V di nodi così organizzato:

- Una sorgente s;
- C nodi corsi;
- $A \cdot O$ nodi, uno per ogni coppia aula-orario (a_j, o_k); questi nodi rappresentano i vari slot prenotabili nelle aule;
- $S \cdot O$ nodi, uno per ogni coppia supervisore-orario (s_l, o_k), se $o_k \in T_l$; questi nodi rappresentano le disponibilità dei supervisori nei vari orari;
- S nodi supervisori;
- un pozzo t.

La dimensione dell'insieme dei nodi è $|V| = C + A \cdot (O + S) + S + 2$.
Costruiamo un insieme E così organizzato:

- per ogni corso c_i, un arco (s, c_i), per un totale di C archi, con peso 1; in questo modo, ad ogni corso verrà assegnato un esame;
- per ogni coppia corso c_i e aula-orario (a_j, o_k), si crea un arco $(c_i, (a_j, o_k))$ se $cs_i \geq as_j$, con peso 1; vengono creati al più $A \cdot C \cdot O$ archi. Poichè ogni corso riceve una capacità 1 dalla sorgente, ad ogni corso potrà essere assegnato al massimo uno slot aula-orario;
- per ogni coppia aula-orario (a_j, o_k) e supervisore-orario (s_l, o_k), si crea un arco $((a_j, o_k), (s_l, o_k))$; vengono creati al più $A \cdot O \cdot S$ archi, con peso 1. Questi archi servono ad associare gli slot delle aule agli slot dei supervisori.
- per ogni coppia supervisore-orario (s_l, o_k) e supervisore s_l, si crea un arco $((s_l, o_k), s_l)$; vengono creati al più $S \cdot O$ archi, con peso 1. Questi archi servono a limitare il numero di supervisioni che possono essere fatte in un singolo orario da parte del supervisore;
- Per ogni supervisore s_l, si crea un arco (s_l, t), per un totale di S archi, con peso d; questi archi servono a limitare il numero di supervisioni da un singolo supervisore.

La dimensione dell'insieme degli archi è $|E| = O(C + ACO + AOS + SO + S) = O(AO(C + S) + SO)$.
Tutti gli archi hanno peso 1 tranne quelli fra supervisori e pozzo, che hanno peso d.
Poichè il flusso massimo è limitato da C, il costo computazionale totale è $O(AOC(C + S) + CSO)$, utilizzando il limite di Ford-Fulkerson.
La figura seguente illustra la costruzione del grafo, con 4 corsi, 3 aule, 3 slot, 3 supervisori.



Esercizio B2 – Permutazioni eleganti – Punti $\geq$ 10

È possibile risolvere il problema utilizzando la tecnica backtrack, modificando opportunamente l'algoritmo di generazione delle permutazioni che abbiamo visto a lezione.

```
printElegant(int n)
    Set A = Set()
    for i = 1 to n do
        A.insert(i)
    int[] S = newint[1 ... n]
    return printElegantRec(A, S, 1)
```

```
printElegantRec(Set A, Item[] S, int i)
    % Se A è vuoto, S è ammissibile
    if A.isEmpty() then
        print S
    else
        % Copia A per il ciclo foreach
        Set C = copy(A)
        foreach c ∈ C do
            if c mod i == 0 or i mod c == 0 then
                S[i] = c
                A.remove(c)
                permRec(A, S, i + 1)
                A.insert(c)
```

Il costo computazionale è $O(n^2 n!)$, per via delle copie del vettore.
È possibile anche modificare la versione più efficiente, sempre presentata a lezione, ottenendo una complessità $O(nn!)$.

```
printElegantRec(Item[] S, int i)
    % Caso base, un elemento
    if i == 1 then
        print S
    else
        for j = 1 to i do
            if (S[i] mod i == 0 or i mod S[i] == 0) and (S[j] mod i == 0 or i mod S[j] == 0) then
                swap(S, i, j)
                permRec(S, i − 1)
                swap(S, i, j)
```

Esercizio B3 – Connessioni – Punti $\geq$ 12

Il problema non è altro che il problema delle sottosequenze comuni massimali, dove una connessione rappresenta un'associazione fra caratteri e la mancanza di intersezioni significa che l'ordine dei caratteri deve essere rispettato nella sottosequenza.

```
int maxConnections(int[] X, int[] Y, int n)
    int[][] DP = new int[0 ... n][0 ... n]
    for i = 0 to n do
        DP[i][0] = 0
        DP[0][i] = 0
    for i = 1 to n do
        for j = 1 to m do
            if X[i] == Y[j] then
                DP[i][j] = DP[i − 1][j − 1] + 1
            else
                DP[i][j] = max(DP[i − 1][j], DP[i][j − 1])
    return DP[n][m]
```

La complessità è quella della LCS, $O(n^2)$.