# *Algoritmi e Strutture Dati – 24/07/2020*

## Esercizio A1

Data la forma della ricorrenza, è possibile utilizzare il Master Theorem, versione base.

- Per $a = 1$, abbiamo $\alpha = \log_2 1 = 0$, $\beta = 0$; siamo nel secondo caso, $T(n) = \Theta(\log n)$.
- Per $a = 2$, abbiamo $\alpha = \log_2 2 = 1$, $\beta = 1$; siamo nel secondo caso, $T(n) = \Theta(n \log n)$.
- Per $a = 3$, abbiamo $\alpha = \log_2 3 < 2$, $\beta = 2$; siamo nel terzo caso, $T(n) = \Theta(n^2)$.
- In generale, è possibile dimostrare che $\log_2 a < a - 1$ per tutti i valori interi $a \geq 3$; siamo nel terzo caso e si ottiene $T(n) = \Theta(n^{a-1})$.

## Esercizio A2

È possibile risolvere questo tipo di problemi con un meccanismo di "sliding window" ed un insieme implementato tramite tabella hash. L'insieme viene utilizzato per contenere i precedenti $k$ valori, per verificare la presenza di duplicati. Una volta che l'insieme contiene i primi $k + 1$ valori (ovvero tutti gli elementi a distanza $\leq k$ rispetto all'elemento in posizione $k + 1$), nei passi successivi si dovrà rimuovere l'elemento a distanza $k + 1$ e verificare che i nuovi elementi inseriti non siano già presenti. Se si arriva in fondo senza incontrare duplicati, si ritorna **false**.

```
boolean closeDuplicates(int[] A, int n, int k)
    SET S = Set()
    for i = 1 to n do
        if i > k + 1 then                          % Remove the element which is k position away
            S.remove(A[i − (k + 1)])
        if S.contains(A[i]) then % Check if the element is already present in the last k − 1 elements
            return true
        else
            S.insert(A[i])
    return false                                   % No duplicate found
```

Il costo computazionale della soluzione proposta è $\Theta(n)$, assumendo che le operazioni sull'insieme siano eseguibili in $O(1)$. È possibile pensare a soluzioni in tempo $O(nk)$ (due cicli annidati), $O(n \log n)$ (creando delle coppie (valore, indice originale) e ordinando per valore e in caso di parità per indice, per poi controllare la distanza fra valori uguali) e infine $O(n \log k)$ (implementazione dell'insieme basato alberi red-black).

## Esercizio A3

Un sottalbero è simmetrico se i suoi sottoalberi destro e sinistro sono speculari. Due sottoalberi $t_1$, $t_2$ sono speculari se il sottoalbero sinistro di $t_1$ è speculare al sottoalbero destro di $t_2$ e il sottoalbero destro di $t_1$ è speculare al sottoalbero sinistro di $t_1$. Il caso base è dato due nodi **nil** (si ritorna **true**) o da uno nodo **nil** e un nodo non **nil** (si ritorna **false**).
La procedura effettua una visita su entrambi gli alberi, e quindi ha complessità $\Theta(n)$.

```
boolean isSymmetric(TREE T)
    return isMirror(T.left, T.right)
```

```
boolean isMirror(TREE t_L, TREE t_R)
  if t_L == nil and t_R == nil then
  |  return true
  else if t_L ≠ nil and t_R ≠ nil then
  |  return isMirror(t_L.right, t_R.left) and isMirror(t_L.left, t_R.right)
  else
  |  return false
```

**Errori tipici**

- È difficile identificare un approccio seguito da molti, ma direi che ho visto molti tentativi sbagliati basati sul tentativo di contare il numero di foglie "destre" e "sinistre" nei sottoalberi destro e sinistro, e confrontarli in maniera incrociata. Sfortunatamente, è possibile che gli alberi rispettino questo controllo e non siano simmetrici nei nodi interni.

## Esercizio B1

L'esercizio può essere risolto estendendo l'approccio per LCS a due stringhe, tenendo conto di tre stringhe.
Sia $DP[i][j][k]$ la lunghezza della longest common subsequence dei prefissi $X(i)$, $Y(j)$, $Z(k)$. Il suo valore può essere calcolato ricorsivamente come segue:

$$DP[i][j][k] = \begin{cases} 0 & i = 0 \vee j = 0 \vee k = 0 \\ DP[i-1][j-1][k-1] + 1 & i > 0 \wedge j > 0 \wedge k > 0 \wedge X[i] = Y[j] = Z[k] \\ \max\{DP[i-1][j][k], DP[i][j-1][k], DP[i][j][k-1]\} & \text{altrimenti} \end{cases}$$

In altre parole, nel caso l'ultimo carattere di tutte e tre le stringhe sia uguale, sia conta un carattere in comune e si "arretra" su tutte le stringhe. Altrimenti, si seleziona il massimo fra eliminare un carattere da una delle stringhe e tenere le altre inalterate.

```
int lcs(ITEM[] X, ITEM[] Y, ITEM[] Z, int n_x, int n_y, int n_z)
  int[][] DP = new int[0...n_x][0...n_y][0...n_z] = { 0 }
  for i = 1 to n_x do
  |  for j = 1 to n_y do
  |  |  for k = 1 to n_z do
  |  |  |  if X[i] == Y[j] == Z[k] then
  |  |  |  |  DP[i][j][k] = DP[i-1][j-1][k-1] + 1
  |  |  |  else
  |  |  |  |  DP[i][j][k] = max(DP[i-1][j][k], DP[i][j-1][k], DP[i][j][k-1])
  return DP[n_x][n_y][n_z]
```

Il costo computazionale è $\Theta(n_x \cdot n_y \cdot n_z)$, per via dei tre cicli annidati.

**Errori tipici**

- Molti hanno inizializzato la matrice con dei cicli che scorrevano la tabella $DP$ in questo modo: $DP[i][0][0]$ (con due indici a zero). In realtà, bisogna scrivere tre cicli annidati, ognuno dei quali ha un indice pari a zero e due indici scorrono per tutti i valori possibili. Per semplificare, pur mantenendo il costo computazionale pari a $\Theta(n_x \cdot n_y \cdot n_z)$, ho preferito inizializzare a zero l'intera matrice.

- Alcuni studenti hanno utilizzato una versione di LCS che ritorna una sottosequenza comune massimale, e poi hanno chiamato la funzione LCS classica sulle prime due stringhe, per poi richiamare LCS sulla stringa così ottenuta e la terza stringa. Questo approccio non va bene perché LCS restituisce una delle sequenze comuni massimali, e tale sottosequenza potrebbe non avere niente in comune con la terza stringa.

  Ad esempio, $LCS(LCS(\texttt{"ccabbacc"}, \texttt{"ccccabba"}), \texttt{"cccc"})) = LCS(\texttt{"ccabba"}, \texttt{"cccc"}) = \texttt{"cc"}$ mentre $LCS(\texttt{"ccabbacc""}, \texttt{"ccccabba"}, \texttt{"cccc"}) = \texttt{"cccc"}$.

- Non va nemmeno bene prendere min(LCS(X,Y), LCS(Y,Z), LCS(X,Z)). Ad esempio, con $X =$ "$ABC$", $Y =$ "$BCD$", $Z =$ "$CDE$", la LCS di ogni coppia di stringhe ha lungheza 2, mentre la lunghezza della LCS di tutte e tre le stringhe è pari a 1 ("C").

### Esercizio B2

Il problema può essere risolto con una rete di flusso.
Si crea il seguente insieme di nodi:

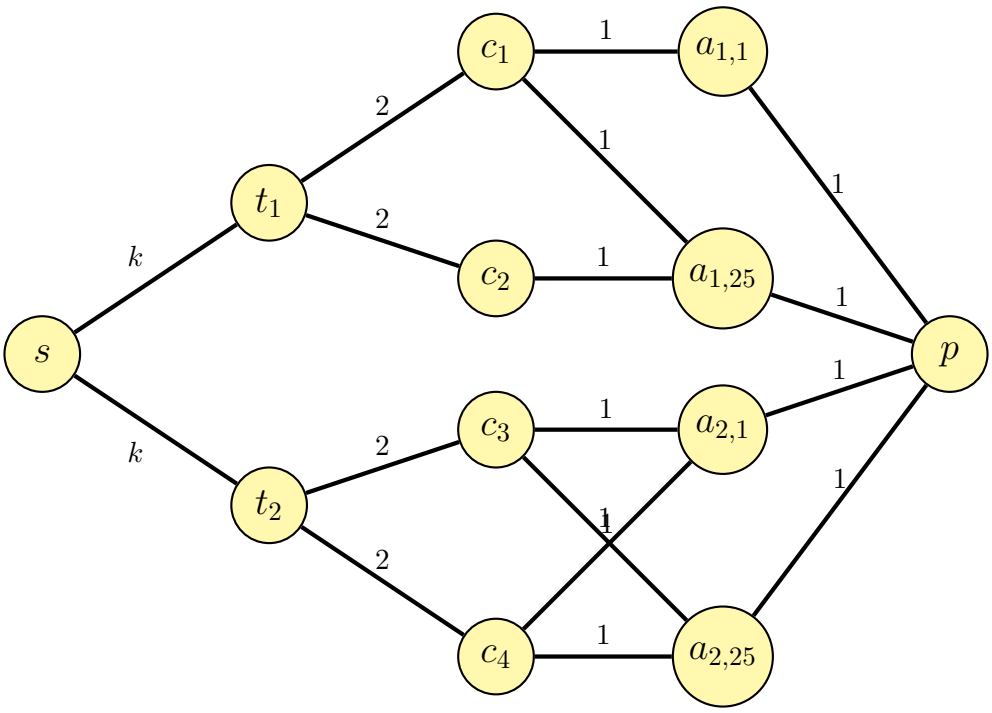
- Nodo sorgente $s$ (1 nodo)
- Per ogni triennale $t_k \in \mathcal{T}$, si crea un nodo $t_k$ ($t$ nodi)
- Per ogni corso $c_i \in \mathcal{C}$, si crea un nodo $c_i$ ($m$ nodi)
- Per ogni coppia aula $a_x$, slot $s_u$, si crea un nodo $(a_x, s_y)$ ($25n$ nodi) corrispondente ad uno slot in una certa aula
- Nodo pozzo $p$ (1 nodo)

per un totale di $m + 25n + t + 2$ nodi.
Si crea il seguente insieme di archi:

- Si collega la sorgente ad ogni triennale $t_k$ ($t$ archi), con peso $k$ ($t$ archi)
- Si collega ogni triennale $c_k$ a tutti i corsi $c_i$ tali per cui $t[i] = k$, con peso 2 ($m$ archi)
- Si collega ogni corso $c_i$ con le coppie $(a, s)$ tali per cui $a \in A_i$ e $s \in S_i$, con peso 1 ($O(mn)$).
- Si collega ogni coppia $(a, s)$ con il pozzo, peso 1 ($25n$ archi)

per un totale di $t + m + mn + 25n$ archi.
In questo modo, ogni coppia aula,slot (peso 1 sull'arco verso il pozzo) viene assegnato ad un solo corso (peso 1 sull'arco dal corso alla coppia. Ogni corso può ricevere al massimo due slot (peso 2 sull'arco dalla triennale al corso), e ogni triennale può ricevere al massimo $k$ slot (peso $k$ sull'arco dalla sorgente alle triennali).
Il numero di slot occupati è limitato superioremente da $2m$, ovvero ad ogni corso vengono allocati al massimo 2 slot.
La complessità risultante è quindi:

$$O(2m(m + 25n + t + 2 + t + m + mn + 25n)) = O(m(mn + t))$$




Notate che nella specifica non era richiesto che il corso non venisse assegnato due volte nello stesso slot temporale. Da questo punto di vista, il testo era mal progettato. Se ci fosse stata anche questa richiesta, avremmo dovuto aggiungere 25 nodi gadget per ogni corso, e da questi nodi andare verso i nodi coppia (aula, slot) rappresentando le possibili associazioni. La complessità del problema non sarebbe cambiata.

**Errori tipici**

- Molti studenti hanno creato una "colonna" di $n$ aule, seguito da una colonna di 25 slot, collegati da archi con pesi decisi in vario modo, ma per lo più con peso 1. Questo non va bene, perchè alla fine ogni aula e ogni slot potranno essere usati al massimo una volta. Questo significa che il flusso massimo è pari a 25, il che significa che posso tenere al massimo 25 lezioni alla settimana e il numero di aule è ininfluente.

## Esercizio B3

Si utilizza un approccio basato su backtrack basato su una visita DFS. Innanzitutto, è possibile notare che è possibile definire i cammini hamiltoniani come una permutazione dei nodi del grafo. Tuttavia, ogni cammino da origine a $n$ permutazioni, ognuna delle quali ha un diverso nodo di partenza. È quindi possibile partire un nodo arbitrario (tutti i cammini hamiltoniani dovranno attraversare tale nodo) e percorrere una visita in profondità. Se si percorrono tutti gli $n$ nodi e dall'ultimo nodo si può raggiungere il primo, si stampa il ciclo.
Nel caso di un grafo completo, il numero di cicli fissato un nodo di partenza è $(n-1)!$, ognuno dei quali richiederà un costo $O(n)$ per la stampa, per un costo finale $O(n!)$.

```
printHamilton(GRAPH G,)
boolean[] visited = new boolean[1...G.n] = { false }        % Init to false
int[] path = new int[1...n]
visitRec(G, 1, 1, path, visited)
```

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

**Errori tipici**

- Molti algoritmi partivano da tutti i possibili nodi. In questo modo, tuttavia, tutti i cicli hamiltoniani vengono stampati $n$ volte.
- Alcuni algoritmi stampavano non appena incontravano un nodo già visitato. Questi algoritmi stampano tutti i cicli e non solo quelli hamiltoniani.
- Una distrazione molto comune è dimenticarsi di controllare che esista l'arco di ritorno dall'ultimo nodo al primo.
- Un errore grave è utilizzare un insieme al posto di visited e del vettore di visita; in questo modo si perde totalmente controllo dell'ordine di visita, e poichè si stampa quando sono stati visitati tutti i nodi, si finisce per stampare l'insieme $V$ senza un particolare ordine.