

### Esercizio 1

E' possibile osservare che  $T(n) = \Omega(n^3)$ , visto il termine  $n^3$  che compare nell'equazione di ricorrenza. Verifichiamo se è anche  $O(n^3)$ , nel qual caso il limite è stretto e abbiamo terminato.

- Caso base:  $T(1) = 1 \leq c \cdot 1^3 \Rightarrow c \geq 1$
- Ipotesi induttiva:  $\forall k < n : T(k) \leq ck^3$
- Passo induttivo:

$$\begin{aligned} T(n) &= T\left(\left\lfloor \frac{n}{\sqrt[3]{2}} \right\rfloor\right) + T\left(\left\lfloor \frac{n}{\sqrt[3]{5}} \right\rfloor\right) + T\left(\left\lfloor \frac{n}{\sqrt[3]{7}} \right\rfloor\right) + n^3 \\ &\leq T\left(\frac{n}{\sqrt[3]{2}}\right) + T\left(\frac{n}{\sqrt[3]{5}}\right) + T\left(\frac{n}{\sqrt[3]{7}}\right) + n^3 \\ &\leq cn^3/2 + cn^3/5 + cn^3/7 + n^3 \\ &= \frac{59}{70}cn^3 + n^3 \leq cn^3 \end{aligned}$$

L'ultima disequazione è vera per se  $59/70c + 1 \leq c$ , ovvero se  $c \geq 70/11$ . Abbiamo quindi trovato due valori  $c = 70/11$  e  $m = 1$  per cui il limite asintotico superiore è soddisfatto. L'equazione di ricorrenza ha quindi complessità  $\Theta(n^3)$ .

### Esercizio 2

E' sufficiente fare una visita in profondità, restituendo i nodi per tempo di fine crescente. Nel codice seguente, ho preso il codice dell'ordinamento topologico e ho semplicemente cambiato la struttura dati da una pila ad una coda.

---

connected-order(GRAPH  $G$ , QUEUE  $Q$ )

---

**boolean**[] *visitato*  $\leftarrow$  **boolean**[1... $G.n$ ]  
**foreach**  $u \in G.V()$  **do** *visitato*[ $u$ ]  $\leftarrow$  **false**

**foreach**  $u \in G.V()$  **do**  
    **if not** *visitato*[ $u$ ] **then**  
        ts-dfs( $G, u, \text{visitato}, Q$ )

ts-dfs(GRAPH  $G$ , NODE  $u$ , **boolean**[] *visitato*, QUEUE  $S$ )  
    *visitato*[ $u$ ]  $\leftarrow$  **true**  
    **foreach**  $v \in G.adj(u)$  **do**  
        **if not** *visitato*[ $v$ ] **then**  
            ts-dfs( $G, v, \text{visitato}, Q$ )  
     $Q.enqueue(u)$

---

La complessità è  $O(m + n)$ ; la correttezza deriva dalle seguenti considerazioni. Considerate il nodo  $u$  con tempo di fine più basso; possono darsi due casi; se  $u$  non ha archi uscenti, la sua eliminazione non comporta disconnessione del grafo; se ha archi uscenti, questi portano a nodi già visitati, e quindi  $u$  fa parte di un ciclo; rimuoverlo lascia il grafo connesso. Una volta rimosso, si considera il nodo con tempo di fine più basso e si ragiona per induzione.

### Esercizio 3

E' possibile risolvere il problema definendo un'equazione di ricorrenza per determinare se è possibile risolvere il problema  $P[i, r, j]$  con le prime  $i$  monete, dovendo dare un resto  $r$  e potendo utilizzare al più  $j$  monete:

$$P[i, r, j] = \begin{cases} r < 0 & \text{false} \\ r = 0 & \text{true} \\ r > 0 \text{ and } (i = 0 \text{ or } j = 0) & \text{false} \\ T(i, r - v[i], j - 1) \text{ or } T(i - 1, r, j) & \text{altrimenti} \end{cases}$$

E' possibile trasformare questo algoritmo in un algoritmo basato su memoization nel modo seguente:

---

```
resto-limitato(integer[] v, integer i, integer r, integer j, integer[][][] P)
  if r < 0 then
    return false
  if r = 0 then
    return true
  if i = 0 or j = 0 then
    return false
  if P[i, r, j] = ⊥ then
    P[i, r, j] = resto-limitato(v, i, r - v[i], j - 1, P) or resto-limitato(v, i - 1, r, j, P)
  return P[i, r, j]
```

---

La complessità è pari a  $O(nRk)$ .

#### Esercizio 4

Una possibile metodo è scrivere una funzione che determina se  $B^k$  è sottosequenza di  $A$ ; per farlo, si può utilizzare un approccio greedy che cerca di individuare tutte le lettere di  $B$ , ripetute  $k$  volte, mano a mano che si incontrano. Un modo compatto per scrivere tale funzione è il seguente (indici a partire da 0)

---

```
isSubsequence(ITEM[] A, ITEM[] B, integer k, integer n, integer m)
  if mk > n then
    return false
  integer i ← 0
  integer j ← 0
  while i < n and j < mk do
    if A[i] = B[j/k] then
      j ← j + 1
    i ← i + 1
  return j = mk
```

---

Questa procedura ha costo  $O(n)$ , in quanto  $mk$  deve essere inferiore o uguale ad  $n$ .

Per individuare il valore massimo di  $k$ , bisognerà ripetere l'operazione con valori crescenti di  $k$ :

---

```
maxSubsequence(ITEM[] A, ITEM[] B, integer n, integer m)
  integer k ← 1
  while isSubsequence(A, B, k, m, n) do
    k ← k + 1
  return k - 1
```

---

Questo ha complessità  $O(n \cdot n/m)$ , ovvero  $O(n^2/m)$ , in quanto al limite verrà ripetuta  $O(n/m)$  volte.