

# Esercitazione 10: Final Problem Session

Giacomo Paesani

May 30, 2024

**Esercizio 1.** Dato un albero  $T$  di  $n$  nodi, rappresentato tramite il vettore dei padri  $P$ , dare lo pseudo-codice di un algoritmo che produce in tempo  $\mathcal{O}(n)$  la lista dei vertici di  $T_u$ , il sotto-albero radicato in un vertice  $u$  di  $T$ .

**Soluzione 1.** Per risolvere questo esercizio, è necessario costruire una lista dei discendenti di  $u$ , cioè tutti i vertici  $v$  di  $T$  per cui esiste un cammino da  $u$  a  $v$  in  $T$ . La soluzione proposta è data dall'Algoritmo 1. Prima di tutto dal vettore dei padri  $P$  si costruiscono le liste di adiacenza di  $T$  tramite il ciclo **for** di Linea 3. A questo punto, possiamo inizializzare una lista vuota  $S$  che riempiamo con i discendenti di  $u$  tramite la chiamata alla funzione DESC.

---

## Algorithm 1

---

**Input:** vettore dei padri  $P$ , vertice  $u$

**Output:** lista dei discendenti di  $u$

```
1: function TREEDESCENDANT( $P, u$ )
2:    $n = |P| - 1$ 
3:   for  $i = 0, \dots, n$  do
4:     if  $P[i] \neq i$  then
5:        $P[i] = \text{Append.Adj}[i]$ 
6:    $S = \emptyset \leftarrow$  lista vuota
7:    $S = \text{DESC}(S, u)$ 
8:   return  $S$ 
9: function DESC( $S, u$ )
10:   $u = \text{Append.S}$ 
11:  for  $v \in \text{Adj}[u]$  do
12:     $S = \text{DESC}(S, v)$ 
13:  return  $S$ 
```

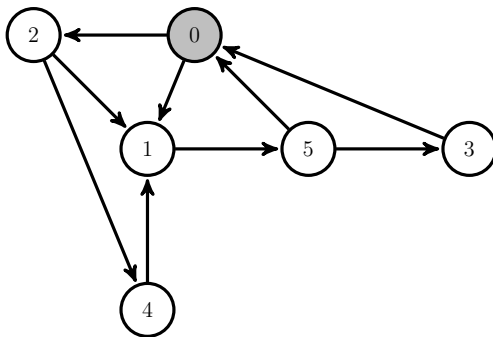
---

La funzione `DESC` è il fulcro della seguente soluzione: prende in input una lista di vertici (che è vuota alla prima chiamata)  $S$ , un vertice  $u$  e restituisce la lista  $S$  in cui sono stati aggiunti tutti i discendenti di  $u$ . Per far questo, si aggiunge  $u$  ad  $S$  (Linea 10) e poi si aggiungono tutti i discendenti  $v$  dei figli di  $u$ , cioè tutti i vertici che hanno  $u$  come padre, e quindi si chiama ricorsivamente la funzione `DESC` con input  $(S, v)$  (Linea 12).

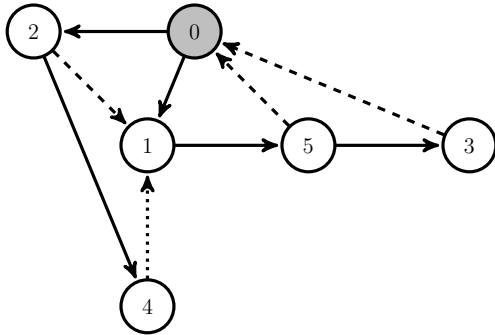
Separando la creazione delle liste di adiacenza dalle chiamate ricorsive alla funzione `DESC` fa sì che questo algoritmo abbia tempo di esecuzione  $\mathcal{O}(n)$ : infatti, per la creazione delle liste di adiacenza usiamo tempo  $\mathcal{O}(n)$  e c'è al più una chiamata ricorsiva della funzione `DESC` per ogni vertice  $v \in V(T)$  e ogni tale chiamata si risolve in  $\mathcal{O}(1)$ .  $\square$

**Esercizio 2.** Sia  $G$  il grafo in figura in cui le liste di adiacenza sono ordinate in senso crescente degli indici. Allora, determinare:

- l'albero di ricerca ottenuto in seguito ad una ricerca in profondità (DFS) di  $G$  con radice il vertice 0;
- specificare quali sono gli archi di attraversamento, in avanti e all'indietro in seguito a tale DFS.

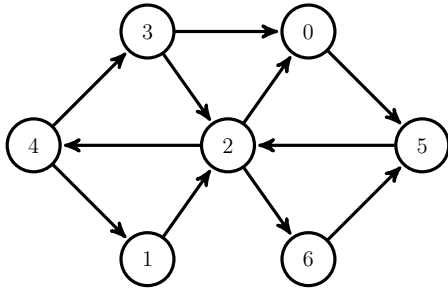


**Soluzione 2.** Allora la soluzione è data dalla seguente figura dove gli archi interi sono quelli dell'albero, quelli tratteggiati sono quelli all'indietro e quelli puntinati sono quelli di attraversamento. Da notare che non ci sono archi in avanti in questo caso.  $\square$



**Esercizio 3** (A. Monti). Sia  $G$  il grafo in figura in cui le liste di adiacenza sono ordinate in senso crescente degli indici.

1. Considerare una visita in profondità (DFS) con radice il vertice 2, allora:
  - 1a. riportare nell'ordine i vertici di  $G$  che vengono effettivamente visitati;
  - 1b. individuare gli archi in avanti, all'indietro, di attraversamento che sono individuati durante la visita.
2. Considerare una visita in ampiezza (BFS) con radice il vertice 2, allora riportare nell'ordine i vertici di  $G$  che vengono effettivamente visitati.
3. Qual'è il minimo numero di archi da eliminare da  $G$  perché il grafo ottenuto risulti avere ordinamenti topologici e quali sono questi archi?
4. Eliminare da  $G$  gli archi ottenuti al punto precedente in modo che il grafo ottenuto  $G'$  risulti avere ordinamenti topologici, allora determinare quanti e quali sono gli ordinamenti topologici di  $G'$ .
5. Eliminare le direzioni degli archi da  $G$  ottenendo un grafo  $G''$  non diretto. Determinare i ponti di  $G''$ .



- Soluzione 3.**
1. Eseguendo la DFS con il vertice 2 come radice si ottiene (1a) l'ordine di visita dei vertici 2, 0, 5, 4, 1, 3, 6; inoltre gli archi all'indietro sono (5, 2), (1, 2) e (3, 2), mentre gli archi di attraversamento sono (3, 0) e (6, 5) e non ci sono archi in avanti;
  2. eseguendo la BFS con il vertice 2 come radice si ottiene l'ordine di visita dei vertici 2, 0, 4, 6, 5, 1, 3;
  3. il grafo  $G$  contiene solo quattro cicli,  $C_1 = (2, 0, 5)$ ,  $C_2 = (2, 6, 5)$ ,  $C_3 = (2, 4, 1)$  e  $C_4 = (2, 4, 3)$ . In particolare,  $C_1$  e  $C_3$  non hanno archi in comune e quindi è necessario eliminare almeno due archi da  $G$  per renderlo aciclico e quindi con ordinamenti topologici. Dimostriamo che è sufficiente eliminare due archi da  $G$  e questi sono (5, 2) e (2, 4): infatti il primo interseca  $C_1$  e  $C_2$  mentre il secondo interseca  $C_3$  e  $C_4$ .
  4. eliminando gli archi (5, 2) e (2, 4) da  $G$  otteniamo i seguenti quattro ordini topologici di  $G'$ : (4, 3, 1, 2, 0, 6, 5), (4, 1, 3, 2, 0, 6, 5), (4, 3, 1, 2, 6, 0, 5) e (4, 1, 3, 2, 6, 0, 5). Questa varietà si ottiene per l'intercambiabilità tra 1 e 3 e tra 0 e 6.
  5. il grafo  $G''$  non ha ponti. Infatti per ogni coppia di vertici del grafo, esistono due cammini, distinti sia negli archi che nei vertici, che li collega: questo certifica che non esistono né punti di articolazione né ponti di  $G$ .  $\square$

**Esercizio 4.** Dato un intero  $n$ , sia  $c_n$  il numero di stringhe binarie lunghe  $n$  in cui non compaiono due zeri consecutivi. Fornire uno pseudo-codice che descrive un algoritmo, che dato  $n \geq 1$ , calcola il valore di  $c_n$  in tempo  $\mathcal{O}(n)$ . E se invece volessi calcolare  $d_n$ , cioè il numero di stringhe binarie lunghe  $n$  in cui non compaiono tre zeri consecutivi?

**Soluzione 4.** Una delle possibili soluzioni è esposta nell'Algoritmo 2. L'idea è di costruire il caso base  $c_1$  e poi ricorsivamente  $c_{n+1}$  a partire da  $c_n$ . Per semplificare la spiegazione di questo esercizio definiamo come  $a_n$  e  $b_n$  come il numero di stringhe binarie lunghe  $n$  in cui non compaiono due zeri consecutivi in cui l'ultimo elemento è uno 0 e un 1, rispettivamente. E' quindi chiaro che  $c_n = a_n + b_n$ .

---

### Algorithm 2

---

**Input:** intero  $n$

**Output:** numero di stringhe binarie di lunghezza  $n$  senza due 0 consecutivi

```

1: function NOCONSECUTIVEZEROS( $n$ )
2:    $i = 1$ 
3:    $a = 1$ 
4:    $b = 1$ 
5:    $c = 1$ 
6:   while  $i < n$  do
7:      $b = a$ 
8:      $a = c$ 
9:      $c = a + b$ 
10:     $i = i + 1$ 

```

---

Allora, una stringa rappresentata da  $a_{n+1}$ , e che quindi termina con 1, ha il prefisso lungo  $n$  in cui l'ultimo termine può essere sia uno 0 che un 1: abbiamo che  $a_{n+1} = a_n + b_n = c_n$ . Una stringa rappresentata da  $b_{n+1}$ , e che quindi termina con 0, ha il prefisso lungo  $n$  in cui l'ultimo termine deve essere necessariamente uno 1: abbiamo che  $b_{n+1} = a_n$ . Si osserva facilmente che vengono calcolati esattamente  $n$  valori  $c_i$  e ognuno di questi calcoli si svolge in tempo  $\mathcal{O}(1)$ : l'algoritmo si risolve in  $\mathcal{O}(n)$ .

Per implementare la soluzione prima descritta è necessario occupare uno spazio pari a  $\mathcal{O}(n)$  di memoria, cioè il valori  $a_i$ ,  $b_i$  e  $c_i$ , per ogni  $1 \leq i \leq n$ . In realtà molti di questi dati diventano superflui una volta utilizzati: allora la variabile  $a$  contiene il valore di  $a_i$ ,  $b$  contiene il valore di  $b_i$  e  $c$  contiene il valore di  $c_i$  e quindi per calcolare  $a_{i+1}$ ,  $b_{i+1}$  e  $c_{i+1}$  è sufficiente aggiornare come prima descritto di valore di  $a$ ,  $b$  e  $c$  e quindi usando  $\mathcal{O}(1)$  spazio.

Supponiamo vogliamo calcolare  $d_n$ , allora definiamo  $a_n$ ,  $b_n$  e  $c_n$  come il numero di stringhe di lunghezza  $n$  che terminano con 1, 10 e con 00, rispettivamente. Allora abbiamo necessariamente che  $d_n = a_n + b_n + c_n$  e che  $a_{n+1} = d_n$ ,  $b_{n+1} = a_n$  e  $c_{n+1} = b_n$ . con il caso base:  $d_1 = 2$ ,  $a_2 = 2$ ,  $b_2 = 1$  e

$c_2 = 1$  e  $d_2 = 4$ . □

**Esercizio 5.** Dati due interi  $k$  e  $n$ , con  $1 \leq k \leq n$ , definiamo  $P(k, n)$  come il numero di differenti partizioni dei numeri da 1 a  $n$  in  $k$  sotto-insiemi non vuoti. Fornire in pseudo-codice un algoritmo che calcola  $P(k, n)$  in tempo  $\mathcal{O}(k \cdot n)$ .

**Soluzione 5.** Per risolvere questo esercizio si suggerisce di adottare un approccio di programmazione dinamica. Costruiamo una matrice  $P$  con  $k$  righe ed  $n$  colonne, dove all'elemento  $p_{ij}$  sarà assegnato il valore  $P(i, j)$ . Come caso base, è facile osservare che  $P(1, j) = 1$ : se è possibile avere un solo sotto-insieme nella partizione, allora ogni numero deve essere necessariamente assegnato nell'unico sotto-insieme a disposizione.

Si considera una qualsiasi partizione dei numeri da 1 a  $j$  in  $i$  sotto-insiemi non vuoti e abbiamo due casi: o (1) il numero  $j$  è in un singleton della partizione oppure (2) il numero  $j$  non è in un singleton della partizione. Allora, il numero delle partizioni che soddisfano il caso (1) è pari alle partizioni dei numeri da 1 a  $j - 1$  in  $i - 1$  sotto-insiemi non vuoti e quindi  $P(i - 1, j - 1)$ . Infine, il numero delle partizioni che soddisfano il caso (2) è pari alle partizioni dei numeri da 1 a  $j - 1$  in  $i$  sotto-insiemi non vuoti moltiplicato  $i$  cioè il numero di diversi elementi della partizione in cui il numero  $j$  può essere posizionato, e quindi  $i \cdot P(i, j - 1)$ . Allora in totale abbiamo che per  $2 \leq i \leq j \leq n$  si ha che  $P(i, j) = P(i - 1, j - 1) + i \cdot P(i, j - 1)$ .

---

### Algorithm 3

---

**Input:** interi  $k$  e  $n$

**Output:** numero di partizioni in  $k$  sotto-insiemi non vuoti dei numeri da 1 a  $n$

```
1: function COUNTPARTITION( $k, n$ )
2:   if  $k > n$  then
3:     return 0
4:   for  $i = 1, \dots, k$  do
5:     for  $j = i \dots n$  do
6:       if  $i = 1$  then
7:          $P(i, j) = 1$ 
8:       else
9:          $P(i, j) = P(i - 1, j - 1) + i \cdot P(i, j - 1)$ 
10:  return  $P(k, n)$ 
```

---

La soluzione proposta nell'Algoritmo 3 implementa proprio l'idea sopra

descritta. Si osserva che il tempo di esecuzione dell'algoritmo è di  $\mathcal{O}(k \cdot n)$ : infatti ogni iterazione dei cicli **for** la computazione del numero  $P(i, j)$  si esegue in tempo costante, cioè  $\mathcal{O}(1)$ .  $\square$

**Esercizio 6.** Dato un intero  $n \geq 2$ , definiamo con  $x_n$  il minimo numero di operazioni con cui è possibile ottenere  $n$  partendo dal numero 2 e potendo effettuare le sole 3 operazioni di incremento di 1, prodotto per due e prodotto per tre; e con  $y_n$  il numero totale di modi (non per forza con un numero minimo di operazioni) per ottenere  $n$  con tali operazioni. Fornire un algoritmo che dato un intero  $n$  calcola sia  $x_n$  che  $y_n$  in tempo  $\mathcal{O}(n)$ .

**Soluzione 6.** Una delle possibili soluzioni per questo esercizio è presentata nell'Algoritmo 4.

---

#### Algorithm 4

---

**Input:** numero  $n$

**Output:** interi  $x_n$  e  $y_n$

```

1: global variables
2:    $X \leftarrow$  array di  $n + 1$  elementi
3:    $Y \leftarrow$  array di  $n + 1$  elementi inizializzato a  $+\infty$ 
4: end global variables
5: function FROM2TON( $n$ )
6:   if  $Y(n) \neq +\infty$  then
7:     return ( $X(n), Y(n)$ )
8:   else if  $n = 0$  or  $n = 1$  or  $n \notin \mathbb{N}$  then
9:      $x_n = +\infty$ 
10:     $y_n = 0$ 
11:   else if  $n = 2$  then
12:      $x_n = 0$ 
13:      $y_n = 1$ 
14:   else
15:      $(a_{-1}, b_{-1}) = \text{FROM2TON}(n - 1)$ 
16:      $(a_2, b_2) = \text{FROM2TON}(n/2)$ 
17:      $(a_3, b_3) = \text{FROM2TON}(n/3)$ 
18:      $x_n = \min\{a_{-1}, a_2, a_3\} + 1$ 
19:      $y_n = b_{-1} + b_2 + b_3$ 
20:      $X(n) = x_n$ 
21:      $Y(n) = y_n$ 
22:   return ( $x_n, y_n$ )

```

---

La spiegazione inizia calcolando  $x_n$ : per ottenere  $n$  col minor numero di operazioni è sufficiente confrontare il minor numero di operazioni per ottenere rispettivamente  $n - 1$ ,  $n/2$  e  $n/3$ , prendere la minima ed aggiungere 1. Infatti, qual'è l'ultima operazione che necessaria per ottenere  $n$ ? Se tale operazione è l'incremento di 1 allora si considera  $x_{n-1}$ , se tale operazione è la moltiplicazione per 2 allora si considera  $x_{n/2}$  e, similmente, se tale operazione è la moltiplicazione per 3 allora si considera  $x_{n/3}$ . Allora, necessariamente abbiamo che  $x_n = \min\{x_{n-1}, x_{n/2}, x_{n/3}\} + 1$ .

Per calcolare  $y_n$ , la situazione è leggermente diversa: infatti è necessario contare tutte le possibili combinazioni di operazioni disponibili che hanno come risultato  $n$ . Per far ciò, si sommano i modi per ottenere  $n - 1$ ,  $n/2$  e  $n/3$  con tali operazioni e quindi si ha che  $y_n = y_{n-1} + y_{n/2} + y_{n/3}$ .

L'algoritmo proposto usa due vettori  $X$  e  $Y$  che permettono di salvare i valori *non banali* di  $x_n$  e  $y_n$  (Linea 20 e 21 e di calcolarli al più una volta (Linea 6), riducendo la complessità dell'algoritmo. Se il valore  $n$  dato in input è pari a 0 o 1 o non è intero allora è chiaro che partendo da 2 è impossibile ottenere  $n$  eseguendo solo le operazioni a disposizione. Allo stesso modo, se  $n = 2$  sono necessarie 0 operazioni per ottenere tale numero a partire da 2 e c'è solo una sequenza (quella banale).

La soluzione qui descritta ha un tempo di esecuzione pari a  $\mathcal{O}(n)$ . Infatti, per ogni  $i$  da 0 a  $n$  si calcolano i valori di  $x_i$  e  $y_i$  sfruttando i valori di  $x_j$  e  $y_j$  con  $0 \leq j < i$  in tempo  $\mathcal{O}(1)$ .  $\square$