

**Esercizio 1**

L'equazione di ricorrenza della funzione `crazy()` è la seguente:

$$T(n) = \begin{cases} 2T(\lfloor n/4 \rfloor) + T(\lfloor n/2 \rfloor) + n^2 & n > 1 \\ 1 & n \leq 1 \end{cases}$$

È facile vedere che  $T(n) = \Omega(n^2)$ ; proviamo a dimostrare che  $T(n) = O(n^2)$ .

- Caso base:  $n = 1, T(n) = 1 \leq cn^2 = c$ , ovvero  $c \geq 1$ .
- Ipotesi induttiva:  $\forall n' < n : T(n') \leq c(n')^2$
- Passo induttivo:

$$\begin{aligned} T(n) &= 2T(\lfloor n/4 \rfloor) + T(\lfloor n/2 \rfloor) + n^2 \\ &\leq 2c\lfloor n/4 \rfloor^2 + c\lfloor n/2 \rfloor^2 + n^2 \\ &\leq 2cn^2/16 + cn^2/4 + n^2 \\ &= 3/8cn^2 + n^2 \leq cn^2 \end{aligned}$$

L'ultima disequazione è vera per  $c \geq 8/5$ .

Abbiamo quindi dimostrato che  $T(n) = \Theta(n^2)$ , con  $c \geq 8/5$  e  $m = 1$ .

**Esercizio 2**

Il problema proposto è quello della bi-colorazione di un grafo, che è possibile se e solo se il grafo è bipartito. La bi-colorazione può essere ottenuta facilmente tramite una visita DFS:

---

```

boolean good-bad-guys(GRAPH G)
  int[] C = new int[1...G.n] = {-1}                                     % Initialized to -1
  foreach u ∈ G.V() do
    if C[u] < 0 then
      if not dfsVisit(G, u, 0, C) then
        return false
  return true

```

---

```

boolean dfsVisit(GRAPH G, int u, int c, int[] C)
  C[u] = c
  foreach v ∈ G.adj(u) do
    if C[v] < 0 then
      if not dfsVisit(G, v, 1 - c, C) then
        return false
    else if C[v] = c then
      return false
  return true

```

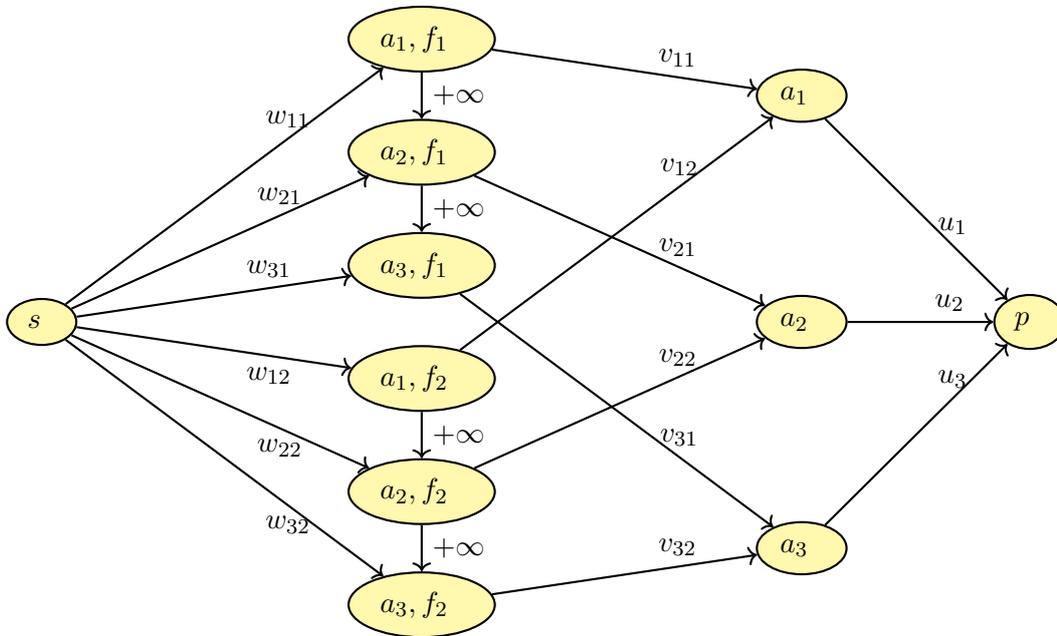
---

### Esercizio 3

Creiamo una rete di flusso contenente una sorgente  $s$ , un nodo  $(a_i, f_j)$  per ogni coppia (anno  $i$ -esimo, foresta  $j$ -esima), un nodo  $a_i$  per ogni anno, e un nodo pozzo  $p$ .

- Aggiungiamo un arco fra la sorgente e ogni nodo  $(a_i, f_j)$ , con capacità  $w_{ij}$  ad indicare gli alberi che maturano in un anno.
- Per ogni foresta  $j$ , creiamo un arco fra ogni nodo  $(a_i, f_j)$  e ogni nodo  $(a_{i+1}, f_j)$ , per indicare che gli alberi maturati nell'anno  $i$ -esimo possono essere usati nell'anno successivo; la capacità può essere messa a  $+\infty$ , oppure può essere pari alla somma degli alberi maturati in quell'anno e in quelli precedenti.
- Aggiungiamo un arco ogni coppia  $(a_i, f_j)$  e l'anno  $a_i$ , con capacità  $v_{ij}$ , ad indicare il numero massimo di archi che possono essere tagliati nell'anno  $i$ -esimo nella foresta  $j$ -esima
- Aggiungiamo un arco fra ogni anno  $a_i$  e il pozzo  $p$ , con capacità  $u_i$ , ad indicare il numero massimo di alberi che possono essere tagliati in un dato anno

Un esempio per  $n = 2$  foreste e  $m = 3$  anni è presente in figura:



Il numero di nodi è  $|V| = nm + m + 2$ , il numero di archi è  $|E| = 2nm + m$ ; un limite superiore al flusso massimo è  $O(U)$ , dove  $U = \sum_{i=1}^m a_i$ . Il costo computazionale è quindi  $O(nmU)$ .

Il numero di alberi che devono essere tagliati per ogni anno e per ogni foresta si trova sugli archi  $(a_i, f_j) \rightarrow a_i$ .

### Esercizio 4

Data una stringa di input  $S$ , l'esercizio può essere risolto in tempo  $\Theta(n^3)$  utilizzando la programmazione dinamica o memoization, utilizzando questa formulazione ricorsiva:

$$DP[i][j] = \begin{cases} 1 & i \geq j \\ 1 & i < j \wedge DP[i+1][j-1] = 1 \wedge S[i] = S[j] \\ \min_{i \leq k < j} \{DP[i][k] + DP[k+1][j]\} & \text{altrimenti} \end{cases}$$

$DP[i][j]$  contiene il numero di stringhe palindrome che costituiscono la sottostringa  $S[i \dots j]$ . Nel caso la sottostringa sia vuota oppure lunga 1 carattere, è palindroma e quindi il numero di stringhe palindrome

è pari a 1. Se la sottostringa  $S[i + 1 \dots j - 1]$  è palindroma e  $S[i] = S[j]$ , allora anche la stringa  $S[i, j]$  è palindroma e il numero di stringhe palindrome è pari a 1. Altrimenti, si spezza la stringa in un punto qualsiasi e si restituisce il minimo numero di palindrome dato dalla somma delle palindrome contenuta nelle due sottostringhe.

---

```

int countPalyndrome(ITEM[] S, int n)
  int[][] DP = new int[0...n][0...n] = {+∞}           % Initialized to +∞
  return cpRec(S, 1, n, DP)

```

---

```

int cpRec(ITEM[] S, int i, int j, int[][] DP)
  if i ≥ j then
    | return 1
  if cpRec(S, i + 1, j - 1, DP) = 1 and S[i] == S[j] then
    | return 1
  if DP[i][j] == +∞ then
    | for k = i to j - 1 do
    | | DP[i][j] = min(DP[i][j], cpRec(S, i, k, DP) + cpRec(S, k + 1, j, DP))
  return DP[i][j]

```

---

La matrice  $DP$  viene inizializzata con valori  $+\infty$ , ad indicare che non sono stati calcolati e pronti per essere utilizzati come valori iniziali per il calcolo del minimo. La chiamata iniziale è  $\text{countPalyndrome}(S, 1, n, D)$  e restituisce il numero minimo di stringhe palindrome che compongono il codice.

Esiste una soluzione alternativa, che utilizza un vettore  $DP$  invece che una matrice.  $DP[j]$  contiene il minimo numero di sottostringhe palindrome necessarie per costituire il prefisso  $j$ -esimo di  $S$ , ovvero la sottostringa  $S[1 \dots j]$ .  $DP[j]$  può essere calcolato nel modo seguente:

$$DP[j] = \begin{cases} 0 & i = 0 \\ \min_{1 \leq i \leq j \wedge \text{palyndrome}(S, i, j) = \text{true}} \{DP[i - 1] + 1\} & i > 0 \end{cases}$$

dove  $\text{palyndrome}(S, i, j)$  è una funzione che restituisce **true** se la stringa contenuta in  $S[i \dots j]$  è palindroma, di costo  $O(n)$ .

L'idea è la seguente: si considerano tutte le sottostringhe palindrome che terminano in  $j$ . Queste sottostringhe contano per uno nella somma finale. Per ognuna di esse, si considera il prefisso restante una volta rimosse, e si prende il numero minimo di palindrome che costituiscono uno di tali prefissi.

---

```

int countPalyndrome(ITEM[] S, int n)
  int[] DP = new int[0...n]
  DP[0] = 0
  for j = 1 to n do
    | DP[j] = +∞
    | for i = 1 to j do
    | | if palyndrome(S, i, j) then
    | | | DP[j] = max(DP[i - 1] + 1, DP[j])
  return DP[n]

```

---

La complessità resta  $O(n^3)$ , in quanto per ognuno degli  $n$  elementi del vettore, bisogna considerare un numero  $O(n)$  di possibili suddivisioni della stringa, per ognuna delle quali bisogna eseguire  $\text{palyndrome}$  di costo  $O(n)$ .

Ma è possibile fare di meglio, con un po' di pre-elaborazione. Sia  $P$  una matrice booleana tale che  $P[i][j] = \text{true}$  se la sottostringa  $S[i \dots j]$  è palindroma.  $P$  può essere calcolata in tempo in tempo  $\Theta(n^2)$  nel modo seguente:

$$DP[i][j] = \begin{cases} \text{true} & i \geq j \\ DP[i+1][j-1] \wedge S[i] = S[j] & i < j \end{cases}$$

Tradotto in codice, l'algoritmo è il seguente:

---

```

int countPalyndrome(ITEM[] S, int n)
  int[][] P = new int[1...n][1...n] = {-1}                                % Initialized
  int[] DP = new int[0...n]
  DP[0] = 0
  for j = 1 to n do
    DP[j] = +∞
    for i = 1 to j do
      if palyndrome(S, i, j, P) then
        DP[j] = min(DP[i-1] + 1, DP[j])
  return DP[n]

```

---

```

int palyndrome(ITEM[] S, int i, int j, int[][] P)
  if i ≥ j then
    return 1
  if P[i][j] < 0 then
    P[i][j] = iif(palyndrome(S, i+1, j-1, P) == 1 and S[i] == S[j], 1, 0)
  return P[i][j]

```

---

Questo algoritmo ha complessità  $\Theta(n^2)$ . Si noti il fatto che gli indici  $i$  devono essere calcolati dal più grande al più piccolo, per assicurarsi che i valori richiesti siano già calcolati.