**ESERCIZIO 1.**

Data una sequenza $S$ crescente di $n$ interi, vogliamo trovare la lunghezza massima per le sottosequenze di $S$ dove ciascun elemento è divisore del successivo.

Ad esempio:
per $S = [3,5,10,20]$ la risposta è $3$ ( la sottosequenza più lunga è $[5,10,20]$).
Per $S = [1,3,6,13,17,18]$ la risposta è $4$ (la sottosequenza più lunga è $[1,3,6,18]$).

Progettare un algoritmo che risolve il problema in tempo $O(n^2)$.

**Motivare BENE la correttezza e la complessità dell'algoritmo proposto.**

Utilizziamo una tabella di dimensioni $n$.

$T[i] =$ **lunghezza massima per sottosequenza lecita che termina in posizione** $i$

La soluzione al problema sarà $\max_{0 \leq i < n} \{T[i]\}$.

Resta solo da definire la ricorrenza per il calcolo delle $\Theta(n)$ celle

**La formula ricorsiva che permette di ricavare $T[i]$ dalle celle precedentemente calcolate è la seguente:**

$$T[i] = \begin{cases} 1 & \text{se } i = 0 \\ 1 & \text{se } S[i] \mod S[j] \neq 0 \text{ per ogni } 0 \leq j < i \\ \max_{0 \leq j < i \text{ per cui } (S[i] \mod S[j]) = 0} \{T[j]\} + 1 & \text{altrimenti} \end{cases}$$

La ricorrenza è corretta per i seguenti motivi:

- se $i = 0$ esiste una sola sottosequenza lecita che termina in $0$ ed ha lunghezza $1$.
- se nessun elemento che precedere $S[i]$ in $S$ lo divide allora l'unica sottosequenza lecita in $S$ che termina in posizione $i$ è il solo elemento $S[i]$.
- se ci sono elementi che dividono $S[i]$ e lo precedono in $S$ posso aggiungere $S[i]$ alle sottosequenze lecite di lunghezza massima che terminano con questi elementi ed ottenere ancora sottosequenze lecite. La sottosequenza di lunghezza massima che termina in $S[i]$ sarà data dal massimo delle sottosequenze lecite che terminano in questi elementi più $1$.

$$T[i] = \begin{cases} 1 & \text{se } i = 0 \\ 1 & \text{se } S[i] \mod S[j] \neq 0 \text{ per ogni } 0 \leq j < i \\ \max_{0 \leq j < i \text{ per cui } (S[i] \mod S[j]) = 0} \{T[j]\} + 1 & \text{altrimenti} \end{cases}$$

```
def es1(S):
    n=len(S)
    T=[0]*n
    for i in range(n):
        T[i]=1
        for j in range(i):
                if S[i] % S[j] == 0 and T[i]<T[j]+1:
                    T[i]=T[j]+1
    return max(T)
```

**Complessità $\Theta(n^2)$**

```
>>> S=[3,5,10,20]
>>> es1(S)
3
>>> S=[1,3,6,13,17,18]
>>> es1(S)
4
```

## ESERCIZIO 2

Progettare un algoritmo che data una stringa $X$ lunga $n$ sull'alfabeto $\{0,1,2\}$ stampa tutte le stringhe lunghe $n$ sull'alfabeto $\{0,1,2\}$ che differiscono da $X$ in ciascuna posizione e non hanno simboli adiacenti uguali.

Ad esempio per $X = 2001$ l'algoritmo deve stampare, non necessariamente nello stesso ordine, le seguenti 5 stringhe:

$$0210 \quad 0212 \quad 0120 \quad 1210 \quad 1212$$

L'algoritmo proposto deve avere complessità $O(nS(n))$ dove $S(n)$ è il numero di stringhe da stampare.

**Motivare BENE la correttezza e la complessità dell'algoritmo proposto.**

## Algoritmo:

- Utilizziamo un algoritmo di backtracking per la stampa di tutte le stringhe ternarie $sol$ lunghe $n$ con l'aggiunta di funzioni di taglio.
- In posizione 0 non può esserci il simbolo $X[0]$ mentre in posizione $i > 0$ non può esserci nè il simbolo $X[i]$ nè il simbolo $sol[i-1]$.
- grazie alla funzione di taglio la soluzione parziale via via costruita è sempre un prefisso di una soluzione da stampare.

## Implementazione:

```
def BT(X,i=0,sol=[]):
    if i==len(X):
        print (''.join(sol))
    else:
        for x in ['0','1','2']:
            if x!=X[i] and (i==0 or x!=sol[i-1]):
                sol.append(x)
                BT(X,i+1,sol)
                sol.pop()
```

```
>>> X='2001'
>>> BT(X)
0120
0210
0212
1210
1212
```

- Nota che nell'albero di ricorsione prodotto dall'esecuzione di $BT$ un nodo viene generato solo se porta ad una foglia da stampare.
- Possiamo quindi dire che la complessità dell'esecuzione di $BT(X)$ richiederà tempo

$$O(S(n) \cdot h \cdot f(n) + S(n) \cdot g(n))$$

  dove:

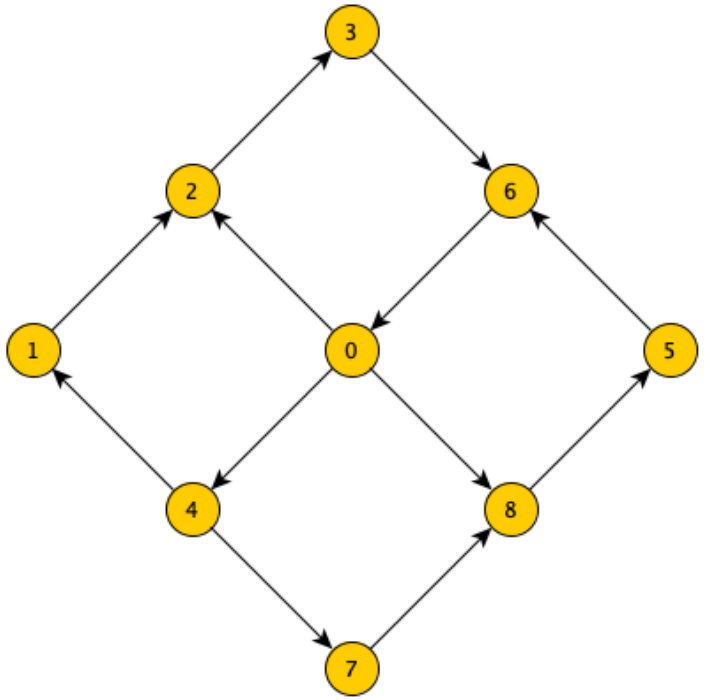
  - $h = n$ è l'altezza dell'albero.
  - $f(n) = O(1)$ è il lavoro di un nodo interno.
  - $g(n) = O(n)$ è il lavoro di una foglia
- Quindi il costo di $BT(X)$ è $O(nS(n))$.

# ESERCIZIO 3

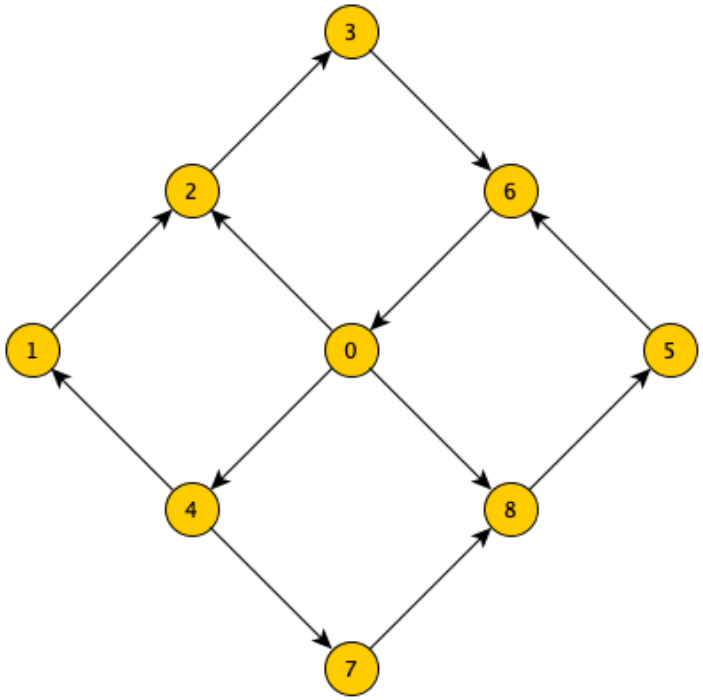
Considerate il grafo $G$ in figura.

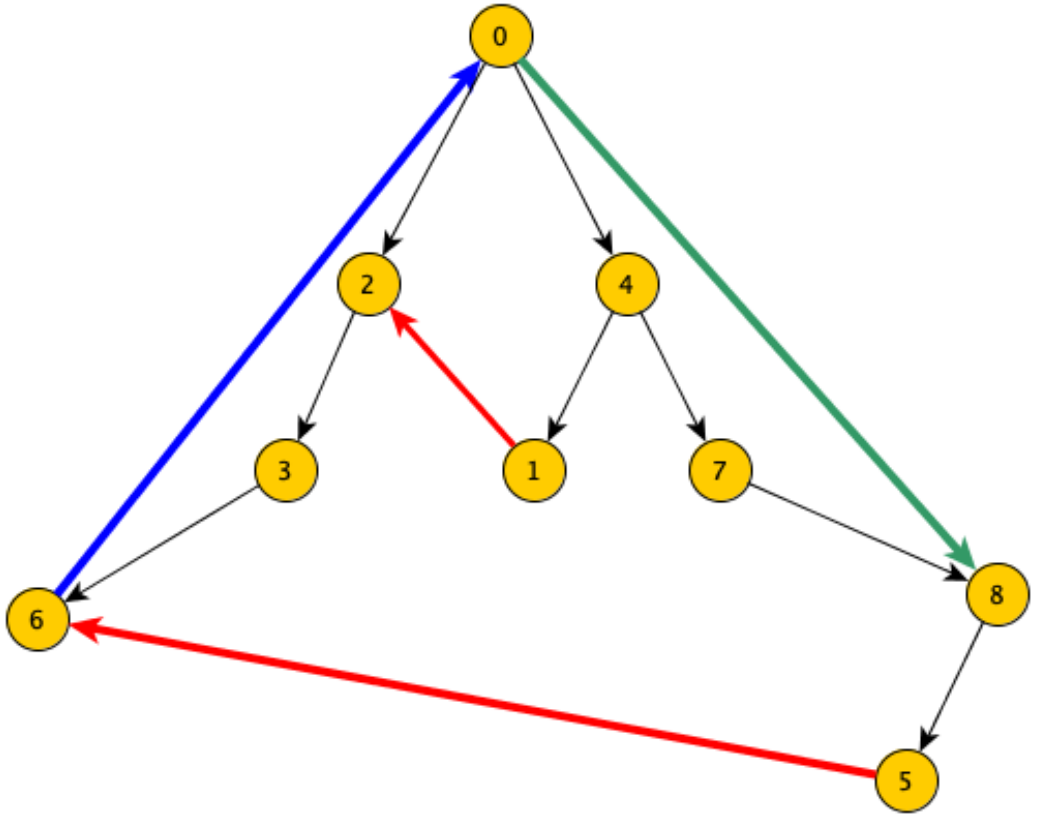
1. Riportate l'albero dfs che si ottiene eseguendo una visita di $G$ a partire dal nodo $0$. Nel corso della visita ogni qualvolta un nodo ha più vicini non visitati scegliete sempre quello di indice minimo (in altre parole assumete il grafo rappresentato tramite liste di adiacenza dove i nodi di ciascuna lista sono ordinati per indice crescente ).
2. dire quali sono gli archi di attraversamento, quali gli archi in avanti e quali gli archi all'indietro che si incontrano nel corso della visita.
3. Quale è il numero minimo di archi da eliminare da $G$ perché il grafo abbia sort topologici? **Motivare BENE la vostra risposta.**
4. Eliminate da grafo $G$ il numero minimo di archi perché questi risulti avere sort topologici e applicate al grafo così ottenuto l'algoritmo per la ricerca del sort topologico basato sulla visita in profondità. Nell'eseguire l'algoritmo qualora risultino più nodi tra cui scegliere per proseguire la visita prendete sempre quella di indice minimo. Qual' è il sort topologico che si ottiene in questo modo?

G=



G=




**La risposta alla seconda domanda è:**

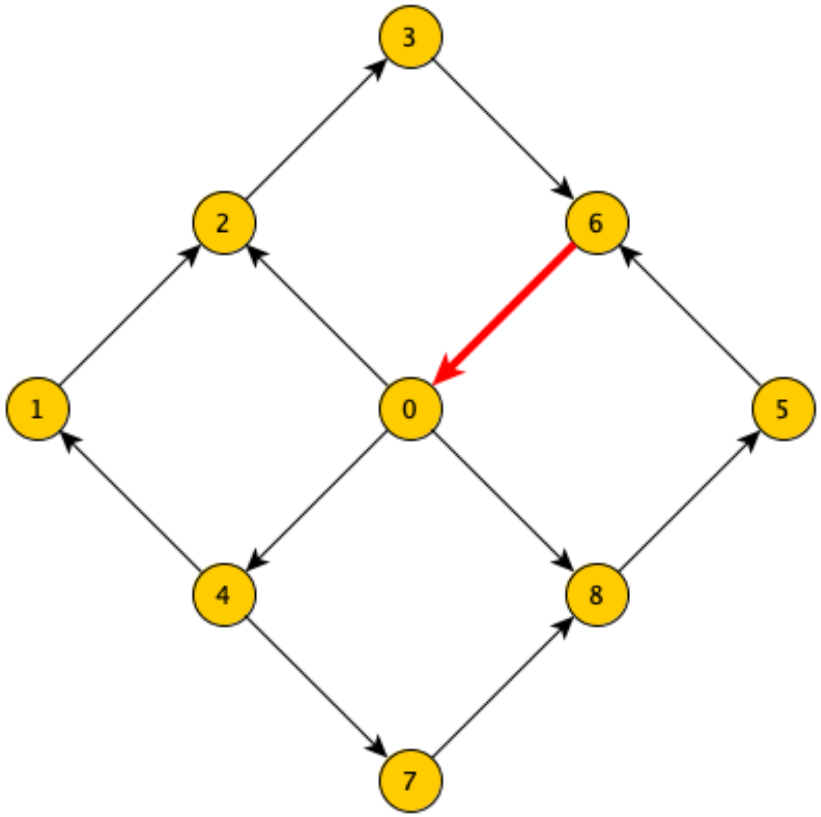
**archi di attraversamento: 1—2 e 5—6**

**archi in avanti: 0-8**

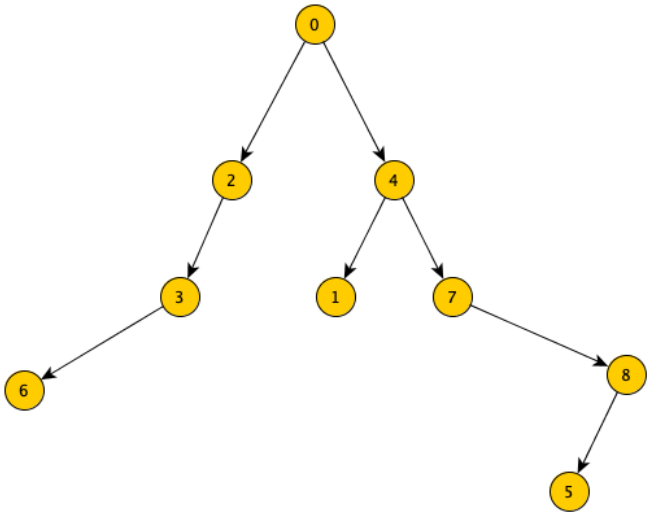
**archi all'indietro: 6—0**

Un grafo ha sort topologici se e solo se è aciclico. Nel grafo sono presenti diversi cicli: (1,2,3,6,0,4) , (0,2,3,6) , (0,8,5,6) , (0,4,7,8,5,6) Per avere sort topologici devo quindi eliminarli dal grafo, i cicli non sono disgiunti ma hanno tutti in comune l'arco (6,0) basterà quindi eliminare quell'arco per rendere il grafo aciclico

G=




**La risposta alla terza domanda è quindi 1.**




Applicando ora l'algoritmo della visita in profondità per la ricerca del sort topologico la visita dei nodi termina secondo quest'ordine:

6-3-2-1-5-8-7-4-0

**La risposta alla quarta domanda è quindi : 0-4-7-8-5-1-2-3-6**