ESERCIZIO 1.

Un numero intero positivo può sempre essere rappresentato come somma di quadrati di altri numeri interi. Ad esempio, il generico numero n può scomporsi come somma di n addendi tutti uguali a 1^2 (vale a dire $n = \sum_{i=1}^{n} 1^2$).
Dato un intero positivo n vogliamo scoprire qual'è il numero minimo di quadrati necessari ad ottenere n.
Ad esempio:

a. per $n = 100$ la risposta è 1 infatti $100 = 10^2$. Nota che vale anche $100 = 5^2 + 5^2 + 5^2 + 5^2$ ma questa scomposizione usa 4 quadrati e non è minimale. Analogamente non è minimale la scomposizione $100 = 8^2 + 6^2$ che usa 2 quadrati.

b. per $n = 6$ la risposta è 3 infatti $6 = 2^2 + 1^2 + 1^2$ e non è difficile vedere che 6 non può esprimersi come somma di due soli quadrati.

Per risolvere il problema viene proposto il seguente algoritmo *greedy*:

```
def esame(n):
    qua=0
    while n>0:
        x=n
        while x**2>n:
            x-=1
        n=n-x**2
        qua+=1
    return qua
```

Provare che l'algoritmo greedy risolve correttamente il problema o fornire un controesempio.

La strategia greedy dell'algoritmo consiste nello scegliere ad ogni passo il quadrato massimo possibile.

L'algoritmo non è corretto! Basta ad esempio portare come controesempio $n = 41$:

- la risposta prodotta dall'algoritmo è 3 infatti nell'ordine vengono selezionati i quadrati $6^2, \ 2^2, \ 1^2$. Si ha dunque $41 = 6^2 + 2^2 + 1$ ma la scomposizione non è minimale.

- la risposta corretta è 2 (infatti 41 non è un quadrato perfetto servono dunque almeno due quadrati per ottenerlo e due quadrati sono sufficienti infatti: $5^2 + 4^2 = 25 + 16 = 41$).

ESERCIZIO 2.

Progettare un algoritmo che dati tre interi positivi n, m e k, con $1 \leq , m, k \leq n$, stampa tutte le stringhe ternarie lunghe n sull'alfabeto $\{0,1,2\}$ dove il numero di occorrenze di 1 non supera m ed il numero di occorrenze di 2 non supera k.
L'algoritmo proposto deve avere complessità $O(n \cdot S(n,m,k))$ dove $S(n,n,k)$ è il numero di stringhe da stampare.

Ad esempio per $n=3$, $m=1$ e $k=2$ l'algoritmo deve stampare, non necessariamente nello stesso ordine, le seguenti 19 stringhe:

000, 001, 002, 010, 012, 020, 021, 022, 100, 102, 120, 122, 200, 201, 202, 210, 212, 220, 221

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

Algoritmo:

Per stampare le stringhe partiamo da un algoritmo di backtracking per la stampa di stringhe ternarie. Modifichiamo quest'algoritmo aggiungendo delle funzioni di taglio che assicurino che se un simbolo viene aggiunto alla soluzione parziale allora quella soluzione potrà poi completarsi in una stringa da stampare.

- Il simbolo 0 può sempre essere aggiunto quindi non c'è bisogno di funzione di taglio
- Il simbolo 1 può essere aggiunto solo se il numero di 1 già inseriti nella stringa non è m. C'è dunque in questo caso bisogno di una funzione di taglio.
- il simbolo 2 può essere aggiunto solo se il numero di 2 già inseriti nella stringa non è k. Anche in questo caso c'è bisogno di una funzione di taglio.

Per far si che le due funzioni di taglio siano calcolabili in tempo $O(1)$ utilizziamo due contatori $tot1$ e $tot2$ che mantengono il conto dei simboli 1 e 2 inseriti in ogni momento nella stringa che si va costruendo.

Implementazione:

```
def bt(n,m,k,i=0,tot1=0,tot2=0,sol=[]):
    if i==n:
        print(''.join(sol))
    else:
        sol.append('0')
        bt(n,m,k,i+1,tot1,tot2,sol)
        sol.pop()
        if  tot1< m:
            sol.append('1')
            bt(n,m,k,i+1,tot1+1,tot2,sol)
            sol.pop()
        if  tot2< k:
            sol.append('2')
            bt(n,m,k,i+1,tot1,tot2+1,sol)
            sol.pop()
```

```
>>> bk(3,1,2)
000
001
002
010
012
020
021
022
100
102
120
122
200
201
202
210
212
220
221
```

- 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 $BT(n, m, k)$ richiederà tempo

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

dove:

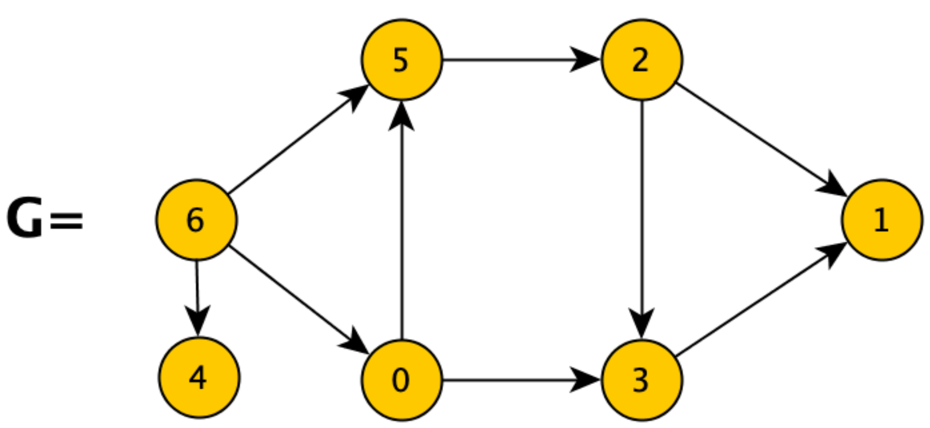
- $h = n$ è l'altezza dell'albero.
- $f(n) = O(1)$ è il lavoro di un nodo interno.
- $g(n) = \Theta(n)$ è il lavoro di una foglia

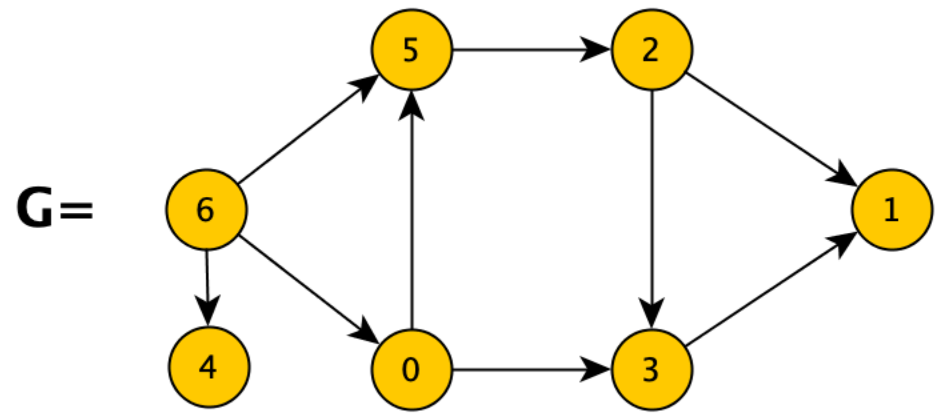
- Quindi il costo di $BT(n, m, k)$ è $\Theta(n \cdot S(n, m, k))$.

ESERCIZIO 3.

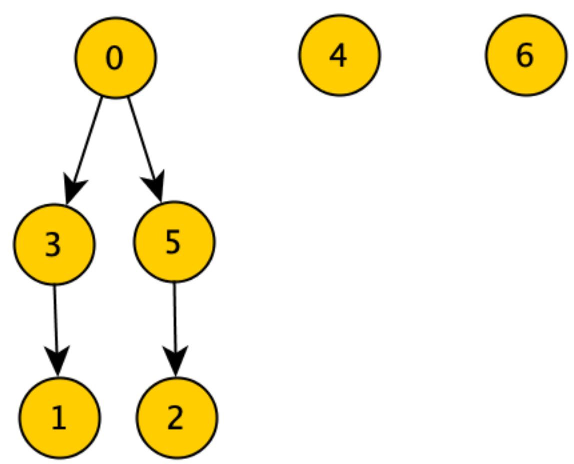
Considerate il grafo aciclico G in figura. Alla ricerca di un ordinamento topologico per G vogliamo applicare l'algoritmo visto a lezione basato sulla visita in profondità di G.

1. Assumiamo che, durante l'esecuzione dell'algoritmo, quando ci sono più nodi tra cui scegliere per proseguire la visita, viene sempre scelto quello di indice minimo
   Riportate l'ordinamento topologico che l'algoritmo produce.
2. Assumiamo ora che, durante l'esecuzione dell'algoritmo, quando ci sono più nodi tra cui scegliere per proseguire la visita, viene sempre scelto quello di indice massimo.
   Riportate l'ordinamento topologico che l'algoritmo produce.



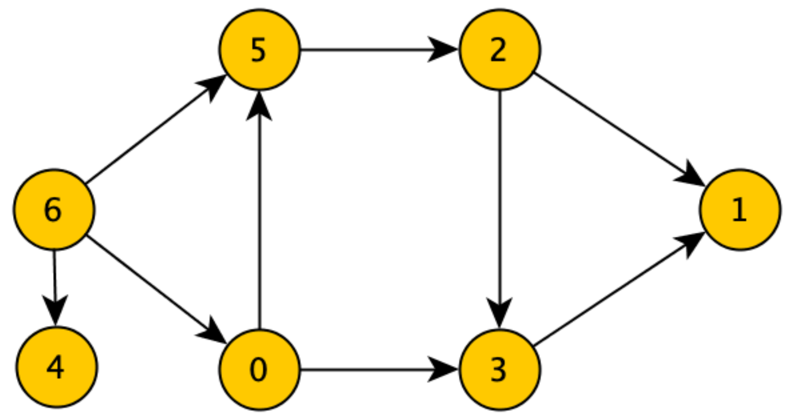


Scegliendo sempre il nodo di indice minimo la visita dei nodi del grafo G richiederà 3 diverse DFS:

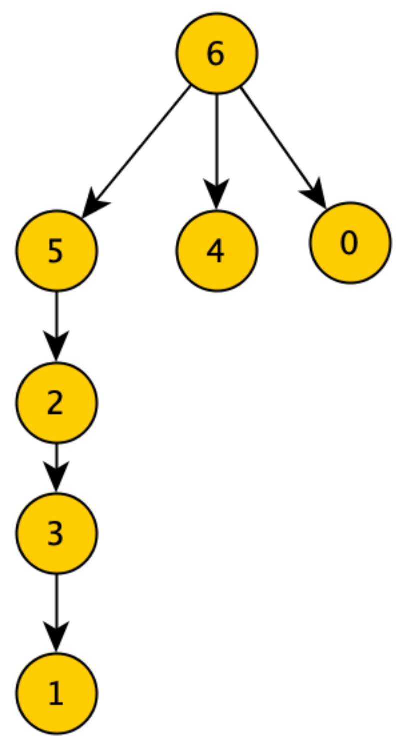


- inserendo in coda ad una lista i nodi man mano che termina la loro visita si ha: **[1,3,2,5,0,4,6]**

- L'ordinamento topologico è dato dal reverse dei nodi in lista: **6,4,0,5,2,3,1**

G= 

Scegliendo sempre il nodo di indice massimo la visita dei nodi del grafo G richiederà una sola DFS:



- inserendo in coda ad una lista i nodi man mano che termina la loro visita si ha: **[1,3,2,5,4,0,6]**

- L'ordinamento topologico è dato dal reverse dei nodi in lista: **6,0,4,5,2,3,1**