Corso di laurea in Informatica
Progettazione d'algoritmi

Tecnica Backtracking 3

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Il problema di ottimizzazione dello Zaino:

Abbiamo n oggetti, ciascuno caratterizzato da un peso p_i ed un valore v_i. Abbiamo inoltre uno zaino di capacità C.
Tra tutti i possibili sottoinsiemi degli n oggetti vogliamo trovare quello il cui peso non supera C e il cui valore sia massimo.
Il peso ed il valore di un insieme di oggetti sono dati dalla somma dei pesi e dalla somma dei valori degli oggetti dell'insieme rispettivamente.

Progettare un algoritmo che, date la lista V dei valori degli n oggetti, la lista P dei pesi degli n oggetti e la capacità C dello zaino, restituisca la soluzione al problema (ovvero il vettore caratteristico degli oggetti da inserire nello zaino per massimizzarne il valore) ed il suo valore.

Esempio: **Considera la seguente istanza con $C = 166$ e $n = 10$ oggetti con peso e valore riportati in tabella**

oggetto	peso	valore
0	23	92
1	31	57
2	29	49
3	89	87
4	44	68
5	53	60
6	38	43
7	63	67
8	85	84
9	82	72

La soluzione ottima pesa 165, vale 309 ed il suo vettore caratteristico è SOL=[1, 1, 1, 0, 1, 0, 1, 0, 0, 0]

Idea1:

- Possiamo risolvere il problema ricorrendo ad un algoritmo esaustivo:

  - generiamo tutti i possibili sottoinsiemi degli n oggetti. Per ognuno verifichiamo se è una soluzione ammissibile (vale a dire se il peso del sottoinsieme di oggetti non eccede la capacità dello zaino) e, in questo caso, ne calcoliamo il valore. Teniamo traccia della soluzione ammissibile di valore massimo via-via trovata. Al termine diamo in outuput il massimo trovato.

Poichè i sottoinsiemi da considerare per n oggetti sono 2^n l'algoritmo esaustivo avrà complessità esponenziale $\Omega(2^n)$.

Algoritmo esaustivo:

la complessità è $O\left(n \cdot 2^n\right)$

```
def es(P, V, c):
    Sol_Ottima = []; valore_Ottimo = -1
    ########
    ########
    def es1(P, V, c, Sol=[], peso = 0, valore = 0):
        n = len(V)
        nonlocal Sol_Ottima, valore_Ottimo
        if  len(Sol) == n:
            #verifico se il sottoinsieme sol è ammissibile
            # ed ha valore ottimo
            if peso <= c and valore > valore_Ottimo:
                    #aggiorno la soluzione ottima e il suo valore
                    Sol_Ottima = Sol.copy(); valore_Ottimo = valore
        else:
            i = len(Sol)
            # scarto l'oggetto
            Sol.append(0)
            es1(P, V, c, Sol, peso, valore)
            Sol.pop()
            # prendo l'oggetto
            Sol.append(1)
            es1(P, V, c, Sol, peso + P[i], valore + V[i])
            Sol.pop()
    ########
    ########
    es1(P, V, c)
    return Sol_Ottima, valore_Ottimo
```

Idea2: trasformiamo l'algoritmo di ricerca esaustiva in un algoritmo di back-traking introducendo funzioni di taglio che permetteranno di potare alcuni dei $\Theta(2^n)$ nodi presenti nell'albero di ricorsione.

- nostro caso procederemo come segue:

  1. potiamo i nodi che non portano a soluzioni ammissibili.
  2. potiamo nodi che se anche portassero a soluzioni ammissibili queste non sarebbero tali da migliorare la soluzione ottima fino a quel momento ottenuta.

- la funzione di taglio del punto 1 verrà applicata al figlio di destra di ogni nodo interno (dove si decide se prendere l'oggetto).
- la funzione di taglio del punto 2 verrà applicata al figlio di sinistra di ogni nodo interno (dove si decide di non prendere l'oggetto).

Algoritmo di backtracking in cui è stata implementata la prima delle due funzioni di taglio:

la complessità è $O\left(n \cdot 2^n\right)$

```
def es(P, V, c):
    Sol_Ottima = []; valore_Ottimo = -1
    ########
    ########
    def es1(P, V, c, Sol=[], peso=0, valore=0):
        n = len(V)
        nonlocal Sol_Ottima, valore_Ottimo
        if  len(Sol) == n:
            #verifico se il sottoinsieme sol è ammissibile
            # ed ha valore ottimo
            if  valore > valore_Ottimo:
                    #aggiorno la soluzione ottima e il suo valore
                    Sol_Ottima = Sol.copy(); valore_Ottimo = valore
        else:
            i = len(Sol)
            # scarto l'oggetto
            Sol.append(0)
            es1(P, V, c, Sol, peso, valore)
            Sol.pop()
            if peso + P[i]<= c:
                # prendo l'oggetto
                Sol.append(1)
                es1(P, V, c, Sol, peso + P[i], valore + V[i])
                Sol.pop()
    ########
    ########
    es1(P, V, c)
    return Sol_Ottima, valore_Ottimo
```

Resta da implementare la seconda funzione di taglio:

- potare quei nodi che se anche portassero a soluzioni ammissibili queste non sarebbero tali da migliorare la soluzione ottima fino a quel momento ottenuta.

Un modo poco costoso per raggiungere parzialmente quest'obiettivo viene fuori dalla seguente osservazione:

- se a livello i della ricorsione si decide di non inserire l'oggetto i nello zaino, nessuna delle soluzioni che sarà poi possibile ottenere potrà valere più di

$$valore + \sum_{j=i+1}^{n-1} V[j]$$

- Di conseguenza se vale

$$valore + \sum_{j=i+1}^{n-1} V[j] \leq valore\_Ottimo$$

possiamo potare l'albero di ricorsione evitando di porre $sol[i] = 0$.

NOTA: l'obiettivo è raggiunto solo parzialmente perché avere

$$valore + \sum_{j=i+1}^{n-1} V[j] > valore\_Ottimo$$

non garantisce che le soluzioni ammissibili che sarà possibile raggiungere saranno realmente in grado di migliorare la soluzione ottima.

Per rendere efficiente il calcolo di questa funzione di taglio nel codice che segue faremo si che ogni nodo dell'albero a livello i riceva una variabile *valore_Rimanente* con l'informazione del valore totale degli oggetti ancora non considerati.
restanti (vale a dire che il nodo livello i conosce il valore di $\sum_{j=i}^{n-1} v_j$)

Algoritmo di backtracking in cui sono state implementate entrambe le funzioni di taglio:

la complessità è $O\left(n \cdot 2^n\right)$

```
def es(P, V, c):
    Sol_Ottima = []; valore_Ottimo = -1
    ########
    ########
    def es1(P, V, c, valore_Rimanente, Sol=[], peso=0, valore=0):
        n = len(V)
        nonlocal Sol_Ottima, valore_Ottimo
        if  len(Sol) == n:
            #verifico se il sottoinsieme sol è ammissibile
            # ed ha valore ottimo
            if  valore > valore_Ottimo:
                    #aggiorno la soluzione ottima e il suo valore
                    Sol_Ottima = Sol.copy(); valore_Ottimo = valore
        else:
            i = len(Sol)
            if valore + valore_Rimanente - V[i] > valore_Ottimo:
                # scarto l'oggetto
                Sol.append(0)
                es1(P, V, c, valore_Rimanente - V[i], Sol, peso, valore)
                Sol.pop()
            if peso + P[i]<= c:
                # prendo l'oggetto
                Sol.append(1)
                es1(P, V, c, valore_Rimanente - V[i], Sol, peso + P[i], valore + V[i])
                Sol.pop()
    ########
    ########
    valore_Rimanente = sum(V)
    es1(P, V, c, valore_Rimanente)
    return Sol_Ottima, valore_Ottimo
```

```
>>> P = [23, 31, 29, 89, 44, 53, 38, 63, 85, 82]
>>> V = [92, 57, 49, 87, 68, 60, 43, 67, 84, 72]
>>> C = 166
>>> es(P,V,c)
([1, 1, 1, 0, 1, 0, 1, 0, 0, 0], 309)
```

Per il problema dello zaino quindi abbiamo visto 3 algoritmi:

- $Zaino(P, V, C)$**: algoritmo di ricerca esaustiva.**
- $Zaino1(P, V, C)$**: algoritmo di backtraking con una funzione di taglio.**
- $Zaino2(P, V, C)$**: algoritmo di backtraking con due funzioni di taglio.**

L'efficacia delle funzioni di taglio per questo tipo di algoritmi può essere validata sperimentalmente.

Ad esempio: possiamo introdurre nei codici dei 3 algoritmi un contatore dei nodi interni dell'albero di ricorsione effettivamente generati nel corso della loro esecuzione.

Eseguendo i tre algoritmi sull'istanza d'esempio con gli $n = 10$ oggetti, vale a dire:
P=$[23, 31, 29, 89, 44, 53, 38, 63, 85, 82]$
V=$[92, 57, 49, 87, 68, 60, 43, 67, 84, 72]$
C=166
per i $2^n - 1 = 1023$ nodi interni dell'albero di ricorsione si ha:

- Per $Zaino(P, V, C)$ vengono visitati 1023 su 1023 nodi interni
- Per $Zaino1(P, V, C)$ vengono visitati 402 su 1023 nodi interni
- Per $Zaino2(P, V, C)$ vengono visitati 371 su 1023 nodi interni

Note:

- la scelta delle funzioni di taglio. In genere migliore è la funzione di taglio più questa risulta computazionalmente onerosa. Considerando che ogni nodo dell'albero di ricorsione dovrà applicarla, non sempre funzioni in grado di tagliare di più producono algoritmi più veloci.

- le soluzioni da cui partire. L'effettività di alcune funzioni di taglio può dipende dalla qualità della soluzione ottima prodotta fino a quel momento. Un esempio di ciò si ha con la seconda funzione di taglio che abbiamo utilizzato per lo zaino.
  In questi casi può essere fruttuoso far partire il backtraking da una soluzione di buona qualità che poi il backtraking tenterà di migliorare.
  Per ottenere *"buone"* soluzioni da cui partire è possibile ricorrere, in una fase di preprocessing, ad algoritmi approssimanti che garantiscono di produrre soluzioni di buona qualità (anche se non ottime) in tempi contenuti.

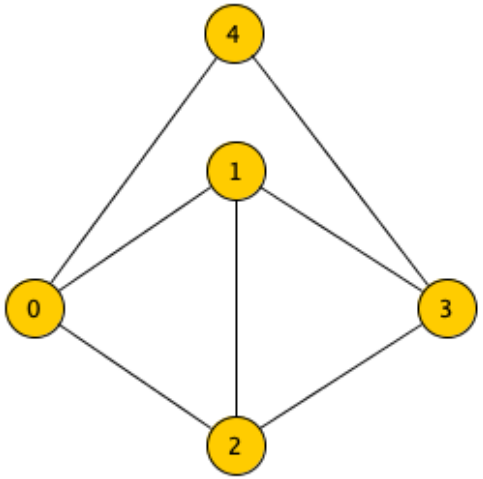
Il problema della 3-colorazione

Sia G un grafo non diretto, una 3-colorazione di G è una colorazione dei suoi nodi con 3-colori $(0, 1$ e $2)$ tale che non risultino nodi adiacenti con lo stesso colore.

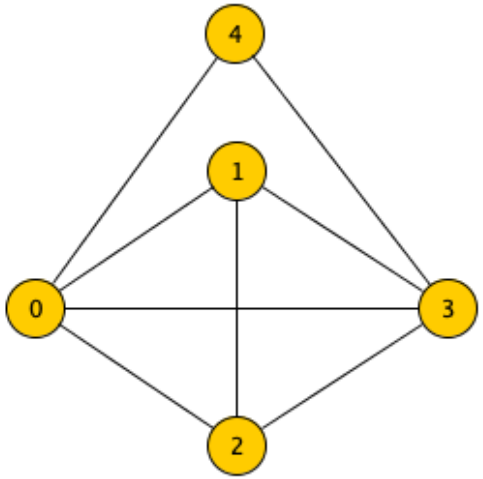
Se una colorazione esiste possiamo rappresentarla con un vettore sol dove $sol[i]$ è il colore assegnato al nodo i.

Ad esempio per il grafo $G1$ in figura esistono ben 12 diverse 3-colorazioni mentre il grafo $G2$ non ha nessuna 3-colorazione.

G1=



G2=



3-colorazioni di G1:

```
['0', '1', '2', '0', '1']
['0', '1', '2', '0', '2']
['0', '2', '1', '0', '1']
['0', '2', '1', '0', '2']
['1', '0', '2', '1', '0']
['1', '0', '2', '1', '2']
['1', '2', '0', '1', '0']
['1', '2', '0', '1', '2']
['2', '0', '1', '2', '0']
['2', '0', '1', '2', '1']
['2', '1', '0', '2', '0']
['2', '1', '0', '2', '1']
```

Progettare un algoritmo che, preso un grafo G, restituisca una sua 3-colorazione nel caso esista, restituisca *None* in caso contrario.

Algoritmo:

Possiamo risolvere il problema con un algoritmo di backtracking che va alla ricerca delle eventuali 3-colorazioni del grafo e appena ne trova una termina.
Al passo ricorsivo i si sceglie un colore per il nodo i tra quelli non utilizzati per i nodi a lui adiacenti già colorati e se questo colore esiste lo si utilizza per colorare il nodo $i+1$. Se si raggiunge $i=n$ l'algoritmo restituisce la colorazione ottenuta.

```python
def col3(G, i=0, sol=[]):
    if i== len(G):
        return sol
    # in C inseriamo i colori gia utilizzati
    # per i nodi adiacenti al nodo i
    C=[]
    for j in G[i]:
        if j < len(sol) and sol[j] not in C:
            C.append(sol[j])
    for c in range(3):
        if c not in C:
            sol.append(c)
            if col3(G,i+1,sol): return sol
            sol.pop()
```

```python
G1=[
    [1, 2, 4],
    [0, 2, 3],
    [0, 1, 3],
    [1, 2, 4],
    [0, 3]
]

>>> col3(G1)
[0, 1, 2, 0, 1]
```

Complessità dell'algoritmo proposto:

La funzione di taglio introdotta (vale a dire la scelta di non colorare un vertice intermedio con un colore uguale a quello di un vertice adiacente già colorato) non garantisce che quella colorazione porterà ad una foglia che rappresenta una colorazione. La complessità quindi non sarà necessariamente proporzionale al numero di 3-colorazioni esistenti.

la Complessità dell'algoritmo è $O\left(3^n\right)$

- l'albero di ricorsione è un albero ternario di altezza n che ha dunque al più $\frac{3^n-1}{3-1} = \Theta(3^n)$ nodi (ricorda infatti che un albero t-ario completo, con $t > 1$, ha esattamente $\sum_{i=0}^{n} t^i = \frac{t^n-1}{t-1}$ nodi)
- ogni nodo interno richiede tempo $O(d)$ dove d è il grado massimo del grafo. Inoltre al più una foglia verrà raggiunta e richiederà tempo $O(1)$.
- il tempo totale è $O\left(d \cdot 3^n\right)$

- Dare una buona stima della complessità è piuttosto difficile proprio perché dipende fortemente dal grafo di input.
- Possiamo però vedere come l'algoritmo si comporta su alcuni input.
- Consideriamo il grafo G mostrato nella figura qui sotto (con la relativa numerazione dei nodi):

- È chiaro che G non è 3-colorabile ma il programma *col*3 quanto tempo impiegherà per accorgersene?
- Per i primi $n-3$ nodi l'algoritmo trova sempre due colori leciti (per il nodo 1 ne trova tre) quindi la ricorsione genera almeno un albero binario completo di profondità $n-3$. Un tale albero contiene almeno 2^{n-3} nodi e dunque la complessità sarà almeno pari a $\Omega(2^n)$ che è esponenziale in n.

- Modifichiamo ora la numerazione dei nodi del grafo G come in figura:

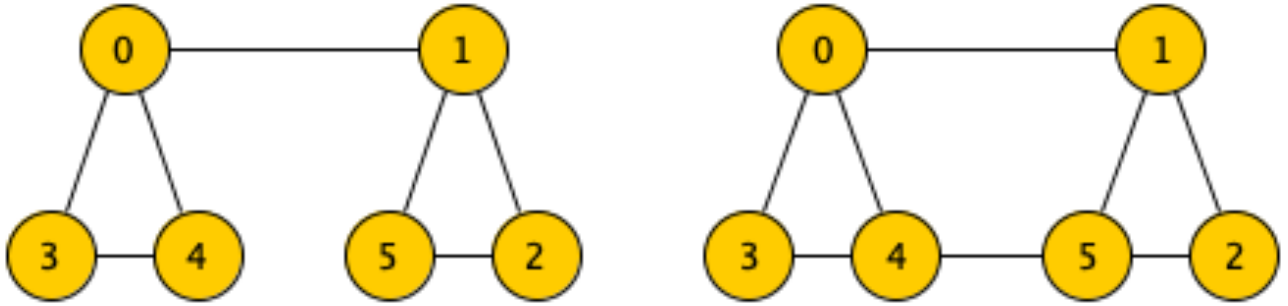
- Quanto tempo impiegherà ora l'algoritmo per accorgersi che G non ha una 3-colorazione lecita?
  La complessità si riduce enormemente perché quando l'algoritmo arriva al nodo 4 non trova mai colori leciti per cui la complessità è pari ad $O(1)$.

- Quindi la complessità del programma *col*3 non solo dipende dal grafo ma può dipendere fortemente anche dalla numerazione dei nodi.

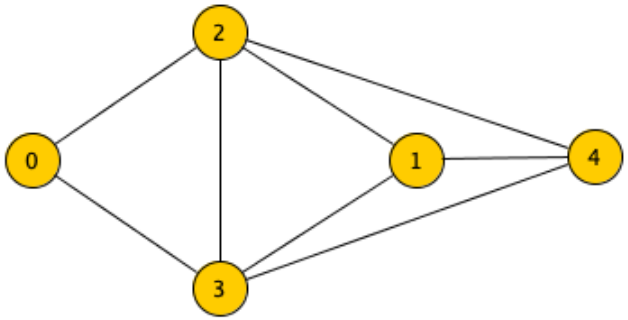
In un grafo un ciclo si dice *Hamiltoniano* se tocca tutti i nodi del grafo.

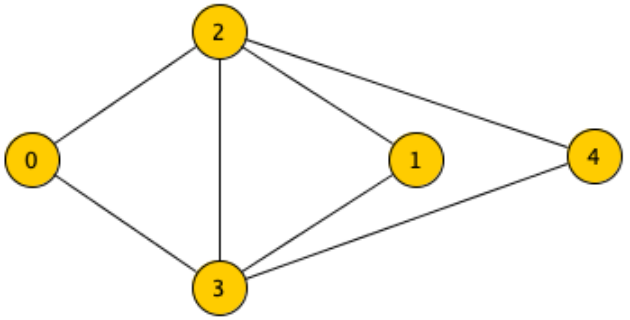
Il problema del ciclo Hamiltoniano: Dato un grafo G voglimo sapere se G ha o meno un ciclo Hamiltoniano.

Ad esempio Il grafo a sinistra non ha cicli hamiltoniani, il grafo di destra si.



Progettare un algoritmo che, dato un grafo G, restituisca un ciclo hamiltoniano se nel grafo è presente, restituisca *None* in caso contrario.

$G1 =$ 

$G2 =$ 

- per il grafo $G1$ l'algoritmo deve restituire uno dei seguenti cicli hamiltoniani del grafo:

  $(0, 2, 1, 4, 3)$ $(0, 2, 4, 1, 3)$ $(0, 3, 1, 4, 2)$ $(0, 3, 4, 1, 2)$

- il grafo $G2$ non ha cicli hamiltoniani e l'algoritmo deve restituire $None$.

Nota che:

- ogni nodo del grafo viene toccato una sola volta dal ciclo quindi un ciclo hamiltoniano è una permutazione P degli n vertici del grafo con la proprietà che esistono nel grafo tutti gli archi:

$$(P[0], P[1]), (P[1], P[2]), \ldots, (P[n-2], P[n-1]), (P[n-1], P[0]).$$

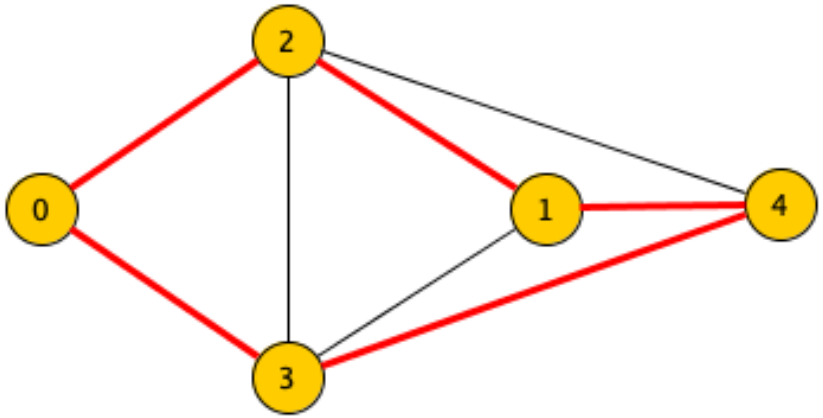
- per risolvere il problema possiamo quindi andare alla ricerca di una tale permutazione.

- Per potare lo spazio di ricerca nell'albero delle permutazioni possiamo
  - imporre che la permutazione inizi con il vertice 0
  - imporre che i prefissi della permutazione rappresentino cammini del grafo.

Algoritmo di backtracking per la ricerca del ciclo hamiltoniano

```
def es(G):
    presi = [0]*len(G)
    presi[0] = 1
    sol = [0]
    if es1(G, sol, presi):
        return sol

def es1(G,  sol, presi):
    if len(sol) == len(G):
        if 0 in G[ sol[-1] ]:
            return True
        return False
    for j in G[ sol[-1] ]:
        if presi[j] == 0:
            sol.append(j)
            presi[j] = 1
            if es1(G, sol, presi):
                return True
            presi[j] = 0
            sol.pop()
    return False
```




```
>>>G = [
      [2, 3],
      [2, 3, 4],
      [0, 1, 3, 4],
      [0, 1, 2, 4],
      [1, 2, 3]
      ]
>>> es(G)
[0, 2, 1, 4, 3]
```

Complessità dell'algoritmo proposto:

La funzione di taglio introdotta (vale a dire la scelta di non generare solo permutazioni che cominciano con 0 e i cui prefissi siano cammini del grafo) non garantisce che i nodi generati porteranno ad una foglia che rappresenta un ciclo hamiltoniano. La complessità quindi non sarà necessariamente proporzionale al numero di cicli hamiltoniani esistenti.

la Complessità dell'algoritmo è $O(n!)$

- l'albero di ricorsione è un albero delle permutazioni di altezza $n-1$ che ha dunque $\Theta((n-1)!)$ nodi.
- ogni nodo interno richiede tempo $\Theta(n)$ mentre al più $(n-1)!$ foglie verranno raggiunte e ciascuna richiederà tempo $O(1)$.
- il tempo totale è $O((n-1)! \cdot n) = O(n!)$

- La complessità del programma dipende fortemente dal grafo di input.
- Ci sono grafi per i quali trova subito un ciclo Hamiltoniano (ad es. se il grafo è un ciclo) o determina che non ce ne sono (ad es. se il grafo è un albero).
- Ma ci sono tantissimi grafi per i quali impiega tempo esponenziale per dare una risposta. Basti pensare che se il grafo G non ha cicli Hamiltoniani, tutti i cammini dal nodo 1 saranno comunque visitati e se questi sono tanti la complessità sarà molto elevata.

- si consideri ad esempio il seguente grafo:

- Il grafo è la concatenazione di $k \geq 1$ cicli ognuno di 4 nodi, ha dunque un totale di $n = 3k + 1$ nodi.
- Ogni ciclo può essere attraversato in due modi diversi quindi il numero di cammini dal nodo 1 al nodo n è 2^k e questi saranno tutti esaminati dal programma ham.
- Quindi il programma impiegherà $\Omega\left(2^{\frac{n}{3}}\right)$ tempo per determinare che il grafo non ha cicli Hamiltioniani.

Esercizio:

Il problema delle n regine. Data una scacchiera $n \times n$ vogliamo calcolare quanti diversi modi ci sono di disporre sulla scacchiera n regine in modo che nessuna di esse sia in grado di catturarne un'altra in base alle regole degli scacchi. Perciò una disposizione dovrà prevedere che nessuna delle n regine abbia colonna, riga o diagonale in comune con un altra.

Ad esempio, ecco due delle possibile disposizioni per $n = 8$:

							x
	x						
			x				
x							
						x	
				x			
		x					
					x		

			x				
	x						
						x	
		x					
					x		
							x
x							
				x			

Bisogna quindi progettare una funzione di backtracking che prende come parametro l'intero n e restituisce il numero di disposizioni lecite possibili.

- Non è difficile convincersi che per $n = 2$ ed $n = 3$ non esiste alcuna disposizione lecita.
- Per i primi valori di n la funzione deve dare le seguenti risposte:

n	disposizioni
1	1
2	0
3	0
4	2
5	10
6	4
7	40
8	92
9	352
10	724
11	2680
12	14200
13	73712
14	365596
15	2279184

Le disposizioni che vogliamo contare soddisfano 3 vincoli:

1. **le n regine sono su righe distinte**
2. **le n regine sono su colonne distinte**
3. **le n regine sono su diagonali distinte**

- I primi due vincoli fanno si che in ciascuna delle disposizioni da contare in ogni riga della scacchiera ci sarà esattamente una regina e lo stesso varrà per ogni colonna.
- Questo significa che posso vedere ciascuna delle disposizioni da contare come una permutazione dei numeri da 0 a $n-1$ dove in posizione i compare la colonna in cui è stata sistemata la regina della i-esima riga.

							x	7
	x							1
			x					3
x								0
						x		6
				x				4
		x						2
					x			5

$= [7, 1, 3, 0, 6, 4, 2, 5]$

Una disposizione che soddisfa i primi due vincoli del problema (vale a dire regine in righe e colonne distinte) può dunque essere vista come una permutazione dei primi n interi.

Non tutte le permutazioni corrispondono alle disposizioni lecite che vogliamo contare.

Esempio di permutazione che non corrisponde ad una disposizione lecita per il problema delle n regine (la regina sulla riga 2 e quella sulla riga 5 condividono la stessa diagonale)

$[7, 1, 3, 0, 4, 6, 2, 5] =$

7								x
1		x						
3				x				
0	x							
4					x			
6							x	
2			x					
5						x		

Per risolvere il problema possiamo modificare il programma che stampa tutte le permutazioni dei primi n interi in modo che, anzicché stamparle, conti quelle che soddisfano i vincoli richiesti.
Cominciamo a scrivere il programma per il conteggio delle "soluzioni" che soddisfanno solo i primi due vincoli quando si richiede alle regine di non essere mai nelle stesse righe o stesse colonne (in pratica conta le permutazioni lunghe n restituendo n!), aggiungeremo poi a questa versione del programma delle funzioni di taglio in modo che vengano generate (e quindi contate) solo permutazioni corrispondenti a disposizioni lecite.

```python
def es(n):
    presi = [0]*n
    return es1(n, [], presi)


def es1(n, sol, presi):
    if len(sol) == n:
        return 1
    tot = 0
    for j in range(n):
        if presi[j] == 0:
            sol.append(j)
            presi[j] = 1
            tot += es1(n, sol, presi)
            sol.pop()
            presi[j] = 0
    return tot
```

```
>>> es(3)
6
>>> es(5)
120
```

Bisogna ora aggiungere al codice i vincoli attinenti alle diagonali di modo che **la posizione che si vuole assegnare ad una nuova regina non deve essere su una diagonale già occupata**.

Ogni cella della scacchiera appartiene a due diagonali: una diagonale che va da sinistra a destra (diagonale principale DP) ed una diagonale che va da destra a sinistra (diagonale secondaria DS).

Ecco due proprietà che permettono di ricavare da una cella la diagonale a cui questa appartiene.

- **due celle** $(i,\ j)$ **e** $(i',\ j')$ **sono sulla stessa diagonale principale DP se e solo se** $i - j = i' - j'$
- **due celle** $(i,\ j)$ **e** $(i',\ j')$ **sono sulla stessa diagonale secondaria DS se e solo se** $i + j = i' + j'$

Ad esempio di seguito per $n = 8$ vengono evidenziate:

- in blu la diagonale destra-sinista di valore $i + j = 4$
- in rosso la diagonale sinistra-destra di valore $i - j = -2$

		$(0,2)$		$(0,4)$			
			$(1,3)$				
		$(2,2)$		$(2,4)$			
	$(3,1)$				$(3,5)$		
$(4,0)$						$(4,6)$	
							$(5,7)$

- **due celle (i, j) e (i', j') sono nella stessa diagonale da sinistra a destra se e solo se $i - j = i' - j'$**
- **due celle (i, j) e (i', j') sono nella stessa diagonale da destra a sinistra se e solo se $i + j = i' + j'$**

In un insieme DP posso tener traccia delle diagonali già occupate che vanno da sinistra a destra mentre in un insieme DS posso tener traccia delle diagonali già occupate che vanno da destra a sinistra.

Per sapere se la diagonale DP della cella (i, j) è stata già occupata basta vedere se $i - j$ è in DP, analogamente, per sapere se la diagonale DS della cella (i, j) è stata già occupata basta vedere se $i + j$ è in DS.

```python
def es(n):
    presi = [0]*n
    return es1(n, [], presi, set(), set())



def es1(n, sol, presi, DP, DS):
    if len(sol) == n:
        return 1
    tot = 0
    i = len(sol)
    for j in range(n):
        if presi[j]==0 and (i-j not in DP) and (i+j not in DS):
            sol.append(j)
            presi[j]=1
            DP.add(i-j)
            DS.add(i+j)
            tot += es1(n, sol, presi, DP, DS)
            sol.pop()
            presi[j]=0
            DP.remove(i-j)
            DS.remove(i+j)
    return tot
```

Corso di laurea in Informatica
Introduzione agli Algoritmi

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

Esercizio

Ho una lista di interi A ed un intero positivo k. Voglio sapere se nella lista esiste un insieme di interi la cui somma dia k .

Ad esempio: sia $A = [61,\ 20,\ 1,\ 33,\ 20,\ 2,\ 4,\ 1,\ 1,\ 33]$

- Per $k = 50$ la risposta è negativa
- Per $k = 70$ la risposta è positiva (esistono diversi insiemi soluzione uno è dato dalla sottolista $[33,\ 4,\ 33]$)

Progettare un algoritmo che dati A e k restituisce una sottolista di A che risolve il problema se esiste, $None$ altrimenti.

Esercizio

Un cavallo è piazzato nella prima cella in alto a sinistra di una scacchiera di lato n vogliamo trovare una sequenza di mosse del cavallo che lo portano a toccare una e una sola volta tutte le celle della scacchiera.

Ad esempio per $n = 5$ una possibile soluzione è la seguente dove in ogni cella è riportato il numero di passi eseguiti dal cavallo per arrivare a quella cella.

0	5	14	9	20
13	8	19	4	15
18	1	6	21	10
7	12	23	16	3
24	17	2	11	22

Progettare un algoritmo che dato n restituisce una tale sequenza di mosse se ne esistono, $None$ altrimenti.

Esercizio

Un *quadrato magico* di ordine n è una matrice $n \times n$ contenente tutti gli interi in $\{1, 2, \ldots, n^2\}$ assegnati in modo tale che la somma di ogni riga, di ogni colonna e delle due diagonali sia la stessa.
Ad esempio per $n = 5$ un possibile quadrato magico è

1	15	24	8	17
23	7	16	5	14
20	4	13	22	6
12	21	10	19	3
9	18	2	11	25

Scrivere in pseudo-codice una procedura che, preso in input un intero n, stampi i diversi quadrati magici di ordine n (nota che la somma di tutti gli elementi della matrice è

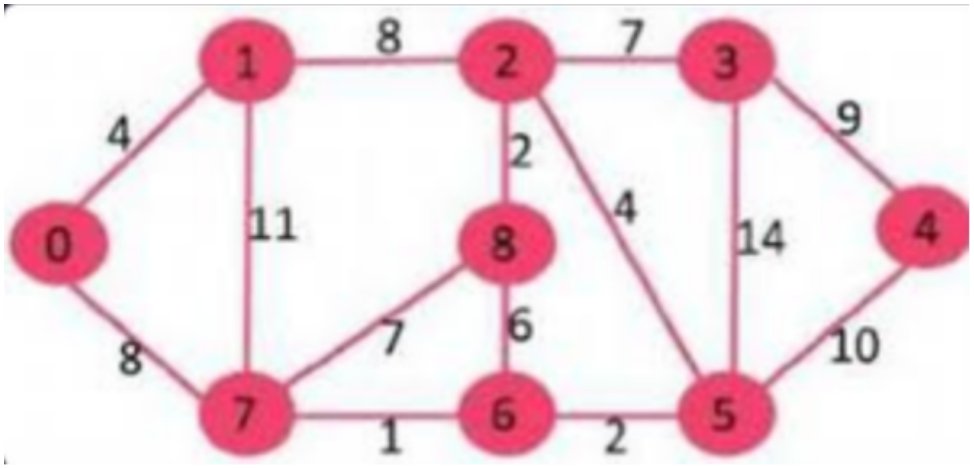
$$\sum_{i=1}^{n^2} i = \frac{n^2(n^2+1)}{2}$$

quindi la somma comune alle righe, alle colonne ed alle due diagonali deve essere $\frac{n(n^2+1)}{2}$).

Esercizio

Dato un grafo diretto e pesato G ed un intero k vogliamo sapere se in G esiste un cammino semplice (vale a dire un cammino che non tocca due volte uno stesso nodo) di costo almeno k che parte dal nodo 0.
Progettare un algoritmo che, dati G e k risolve il problema restituendo il cammino se esiste, *None* altrimenti.

Ad esempio per il grafo in figura e $k = 58$ esiste un cammino semplice di costo 60 e la procedura deve restituire dunque la sequenza di nodi $[0, 7, 1, 2, 8, 6, 5, 3, 4]$,



Esercizio

L'ufficio postale di uno stato permette l'uso di francobolli di n valori interi e diversi e proibisce l'uso di più di m francobolli per lettera. Scrivere in pseudo-codice una procedura che, presi in input m ed n, calcola il massimo intervallo a partire da 1 di affrancature possibili e tutti gli insiemi di valori che permettono di realizzarlo.

Ad esempio, per $n = 4$ e $m = 5$ l'intervallo massimo è $\{1, 2 \ldots, 71\}$ che si può ottenere solamente con i seguenti due insiemi di valori:

$\{1, 4, 12, 21\}$

$\{1, 5, 12, 28\}$.