BACKTRACKING

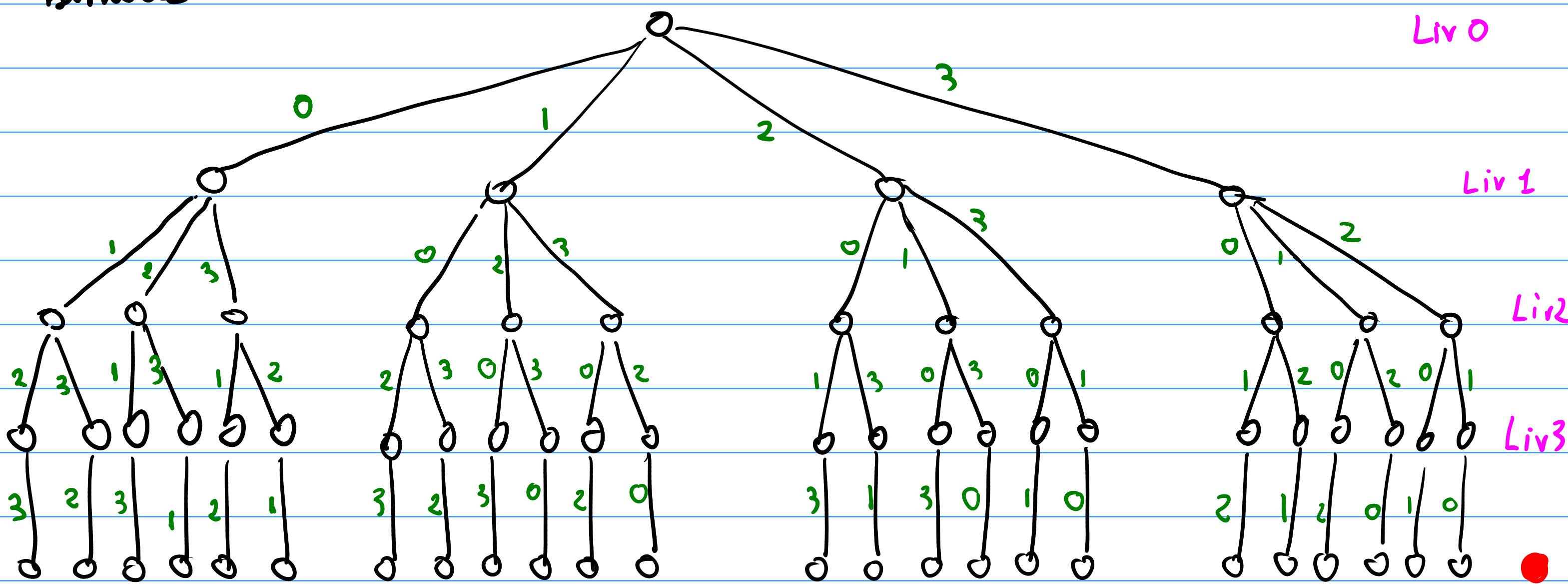
PERMUTAZIONI

- In queste dispense indichiamo come $P(n)$ l'insieme di permutazioni di $\{0,1,2,\dots,n-1\}$
- $|P(n)| = n!$ • $P(3) = \{(0,1,2),(0,2,1),(1,0,2),(1,2,0),(2,0,1)(2,1,0)\}$

Esercizio 1: Descrivere un algoritmo ottimo che stampi $P(n)$

Osserviamo che ci sono $n!$ permutazioni, e ognuna richiede $\Omega(n)$ per essere stampata, quindi $\Omega(n \cdot n!)$

Soluzione Usiamo un meccanismo ricorsivo simile a quello delle stringhe binarie



Per $i=0,\dots,n-1$:
- abbiamo una sequenza $v_0 \dots v_{i-1}$ di valori distinti $\{0,\dots,n-1\}$
  un array SELECTED tale che SELECTED[v] = True se e solo se v è in $v_0 \dots v_{i-1}$
- Scegliamo come v_i uno dei valori per cui SELECTED è falso

- Essenzialmente il nostro algoritmo produrrà:
  - un nodo interno per ogni PREFISSO di una permutazione
  - un nodo foglia per ogni permutazione

Foglie $= n!$ costo per foglia $O(n)$

nodi interni $= 1 + n + n(n-1) + n(n-1)(n-2) + \cdots =$

$$= \sum_{i=0}^{n-1} \frac{n!}{(n-i)!} < n! \sum_{i=0}^{n-1} \frac{1}{(n-i)!} = n! \sum_{i=1}^{n} \frac{1}{i!} < n! \sum_{i=1}^{\infty} \frac{1}{i!} =$$

usando $i! \geq \frac{2^i}{2}$ per $i \geq 1$ e $\sum_{i=1}^{\infty} x^i = \frac{x}{1-x}$ per $x < 1$

$$= n! \sum_{i=1}^{\infty} \frac{2}{2^i} = 2 \cdot n! \cdot \frac{1/2}{1-1/2} = 2 \cdot n! \rightarrow \text{\# nodi interni } O(n!)$$

costo per nodo interno $O(n)$

Complessità: $O(n \cdot n!)$

```
57 def permutations(n):
58     selected=[False]*n
59     perm_rec(n,0,[],selected)
60
61
62 def perm_rec(n,i,p,sel):
63
64     if i >= n:
65         print(p)
66         return
67
68     for nextvalue in range(n):
69         if sel[nextvalue]:
70             continue
71         p.append(nextvalue  )
72         sel[nextvalue]=True
73
74         perm_rec(n,i+1,p,sel)
75
76         sel[nextvalue]=False
77         p.pop()
78
```

- `selected=[False]*n` → NESSUN VALORE È SELEZIONATO
- `perm_rec(n,0,[],selected)` → IL PREFISSO DI PERMUTAZIONE È VUOTO
- `if sel[nextvalue]: continue` → TAGLIO I RAMI CORRISPONDENTI A VALORI GIÀ SELEZIONATI, PERCHÉ LE PERMUTAZIONI NON HANNO RIPETIZIONI
- `p.append(nextvalue)`, `sel[nextvalue]=True` → AGGIUNGO UN NUOVO VALORE
- `perm_rec(n,i+1,p,sel)` → RICORSIONE CON UN VALORE IN PIÙ
- `sel[nextvalue]=False`, `p.pop()` → ELIMINO IL NUOVO VALORE

- In effetti abbiamo generato le permutazioni andando a costruire tutte le sequenze lunghe n di valori in $\{0,1,2,\dots,n-1\}$ ma assicurandoci CON UNA FUNZIONE DI TAGLIO che non si inserissero nella sequenza valori ripetuti.

ESERCIZIO 2 Progettare un algoritmo che stampi tutte e sole le permutazioni che contengono NUMERI PARI, nelle POSIZIONI PARI

- Se $S(n)$ è il numero di queste permutazioni, allora l'algoritmo deve avere complessità $O(n^2 S(n))$

Indizi
- Aggiungere una funzione di taglio alle righe 69-70 dell'algoritmo a pag. 2
- Osservare che, se la funzione di taglio è corretta, le foglie corrispondono esattamente alle permutazioni desiderate
- Quindi $\#\text{foglie} = S(n)$
- Osservare che $\#\text{nodi} = \#\text{foglie} + \#\text{nodi interni} \leq \#\text{foglie} \cdot (n+1)$
- Osservare che il lavoro compiuto in ogni nodo è $O(n)$

Esempio $n=5$. ci sono $5! = 720$ permutazioni ma solo 12 hanno le caratteristiche richieste

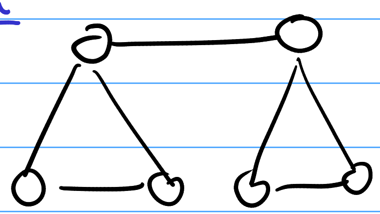
[0,1,2,3,4] [0,1,4,3,2] [2,1,0,3,4]
[2,1,4,3,0] [4,1,0,3,2] [4,1,2,3,0]
[0,3,2,1,4] [0,3,4,1,2] [2,3,0,1,4]
[2,3,4,1,0] [4,3,0,1,2] [4,3,2,1,0]

Esercizio 3

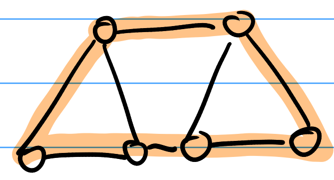
- Questo esercizio chiede di determinare se un grafo $G=(V,E)$ non orientato ha un CICLO HAMILTONIANO

- Un CICLO HAMILTONIANO in un grafo G è un ciclo di G che contiene TUTTI i vertici del grafo.

Esempi



NESSUN CICLO HAMILTONIANO



HA UN CICLO HAMILTONIANO

Progettiamo un algoritmo che, dato $G=(V,E)$ con $|V|=n$

- Restituisca None, se G non ha un ciclo Hamiltoniano
- Restituisca una sequenza $(v_0, v_1, v_2 \dots v_{n-1})$ tale che
  - $v_0 = 0$
  - $\{v_i, v_{i+1}\} \in E$ per ogni $0 \leq i \leq n-2$ e $\{v_0, v_{n-1}\} \in E$
  - $v_i \neq v_j \quad \forall \ 0 \leq i < j < n$

- In sostanza l'algoritmo deve trovare una permutazione dei vertici tale che vengano soddisfatte le condizioni

- Utilizzeremo una procedura simile a quella delle permutazioni
  - al passo i abbiamo la sequenza $v_0, \dots v_{i-1}$ e apriamo un ramo nella ricorsione per ogni vertice vicino di v_{i-1}, non visitato
  - al passo n verifichiamo che v_0 sia vicino di v_{n-1}

Algoritmo di Backtracking per ciclo Hamiltoniano

```python
26 def hamiltonian(G):
27     selected=[False]*len(G)
28     selected[0]=True
29     return hamc_rec(G,1,[0],selected)
30
31
32 def hamc_rec(G,i,hamc,sel):
33
34     if i >= len(G):
35         if hamc[0] in G[hamc[-1]]:
36             return hamc
37         else:
38             return None
39
40     for vi in G[hamc[-1]]:
41
42         if sel[vi]:
43             continue
44
45         hamc.append(vi)
46         sel[vi]=True
47
48         if hamc_rec(G,i+1,hamc,sel) is not None:
49             return hamc
50
51         sel[vi]=False
52         hamc.pop()
53
54     return None
```

→ Partiamo dalla soluzione $V_0 = \emptyset$

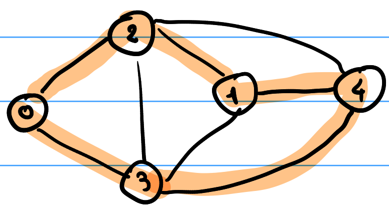
→ Verifica che V_0 sia un vicino di V_{n-1}

→ Prolunghiamo la sequenza solo con i vicini di V_{i-1}

→ Fermiamo la ricorsione appena troviamo una soluzione

Esempi

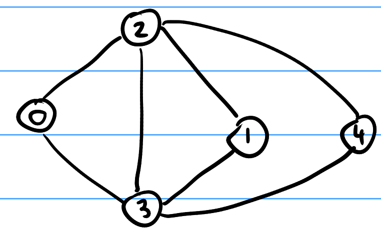
G: 0 → 2,3
1 → 2,3,4
2 → 0,1,3,4
3 → 0,1,2,4
4 → 1,2,3



L'algoritmo trova il ciclo 0,2,1,4,3

Altre soluzioni sono (0,2,4,1,3) (0,3,1,4,2) (0,3,4,1,2)

G: 0 → 2,3
1 → 2,3
2 → 0,1,3,4
3 → 0,1,2,4
4 → 2,3



L'algoritmo non restituisce cicli

Complessità: Visto che il primo vertice è fissato, vengono tentate al più $(n-1)!$ sequenze, che corrispondono alle permutazioni di $\{1,...,n-1\}$

Quindi (se non ci sono tagli ulteriori) i nodi dell'albero sono $O((n-1)!)$ come visto per l'esercizio 1

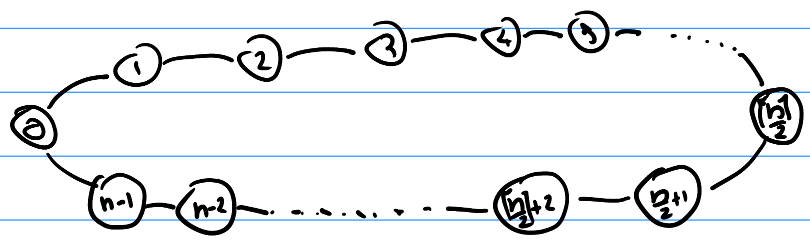
Il lavoro per ogni nodo è $O(n)$ quindi la complessità totale è $O(n!)$

Esercizio 4

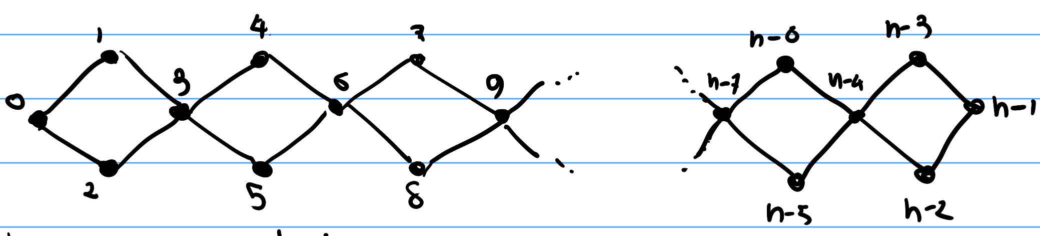
Questo esercizio serve a far vedere che le prestazioni dell'algoritmo dipendono fortemente dal grafo

- Dare una stima (anche grossolana) della complessità dell'algoritmo a pag 5 sui due grafi

CICLO di n vertici



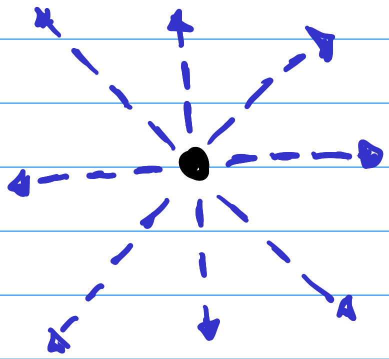
BIVI CONCATENATI



il grafo ha $3k+1$ vertici

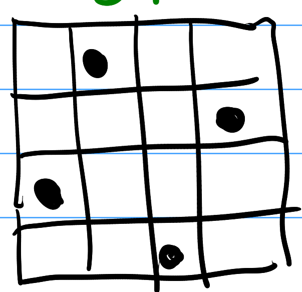
Esercizio 5 Il problema delle n regine

Data una scacchiera $n \times n$, quanti modi ci sono per posizionare n regine in modo tale che nessuna possa mangiarne un'altra?

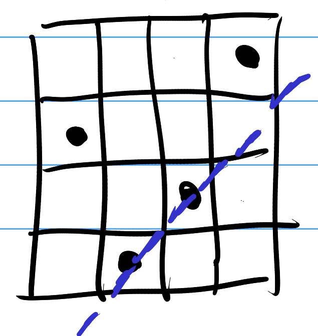


- Ogni regina muove nelle 8 direzioni ad una distanza arbitraria

SI



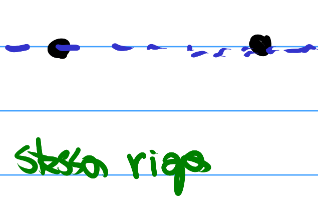
NO

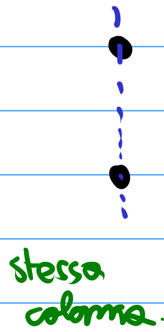


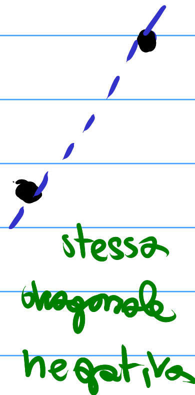
Domanda: è possibile mettere più di n regine su una scacchiera $n \times n$?

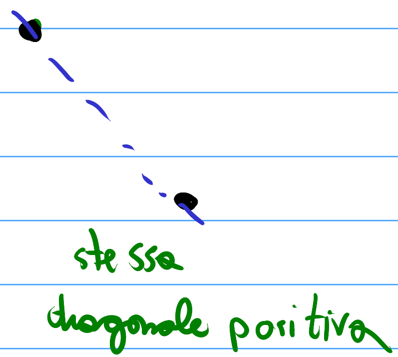
Domanda: quante configurazioni possibili ci sono per una scacchiera 3×3? E una 4×4?

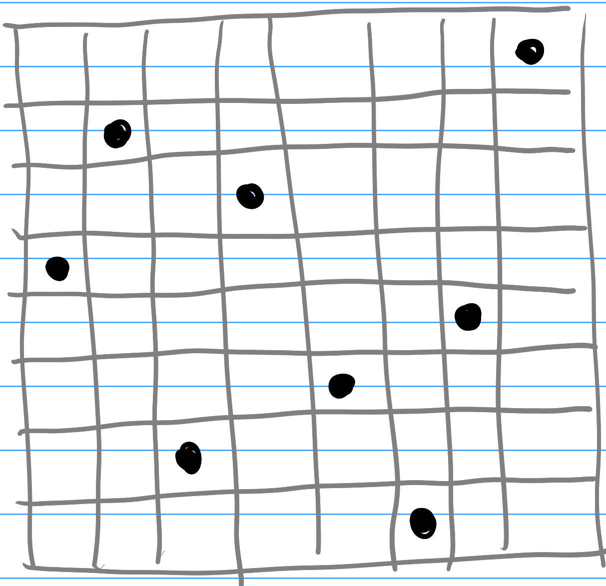
Precisiamo l'esercizio Dato n un programma deve calcolare il numero di posizionamenti di n regine in una scacchiera $n \times n$ tali che nessuna coppia di regine sia in una delle seguenti configurazioni

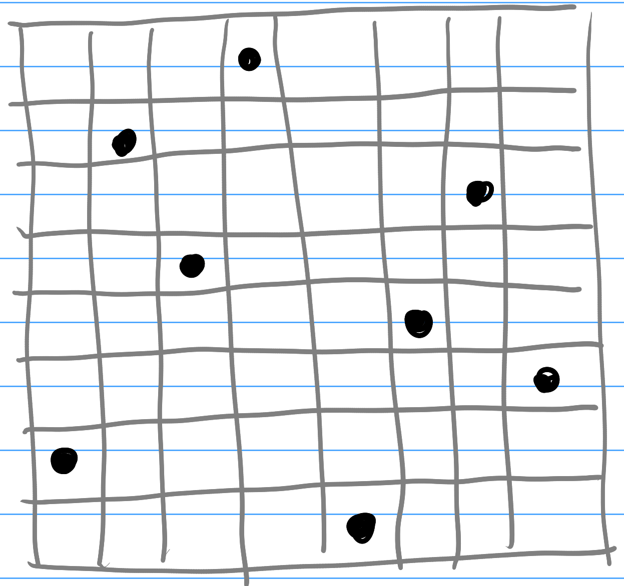












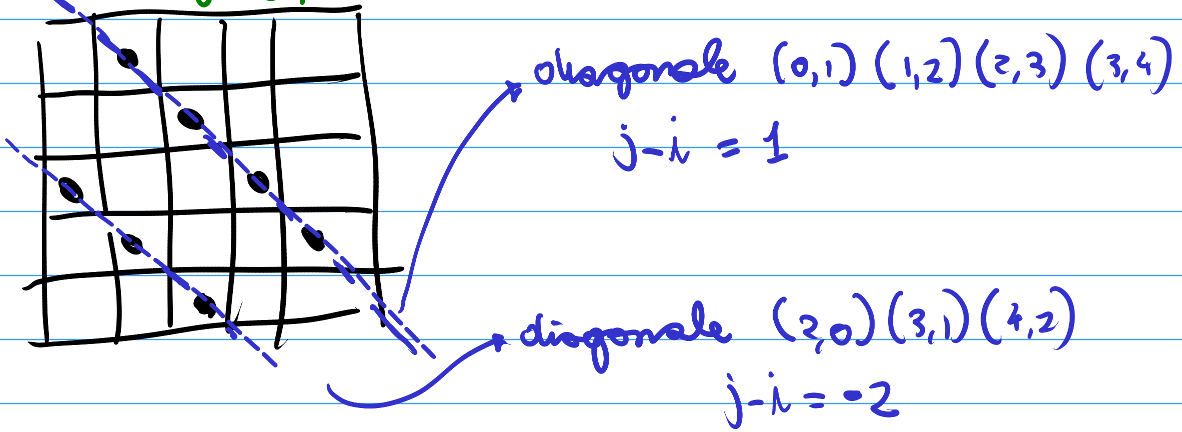
n	posizionamenti
1	1
2	0
3	0
4	2
5	10

n	posizionamenti
6	4
7	40
8	92
9	352
10	724

n	posizionamenti
11	2680
12	14200
13	73712
14	365596
15	2279184

Come testo se due regine in posizioni (i_1, j_1) e (i_2, j_2) sono in conflitto

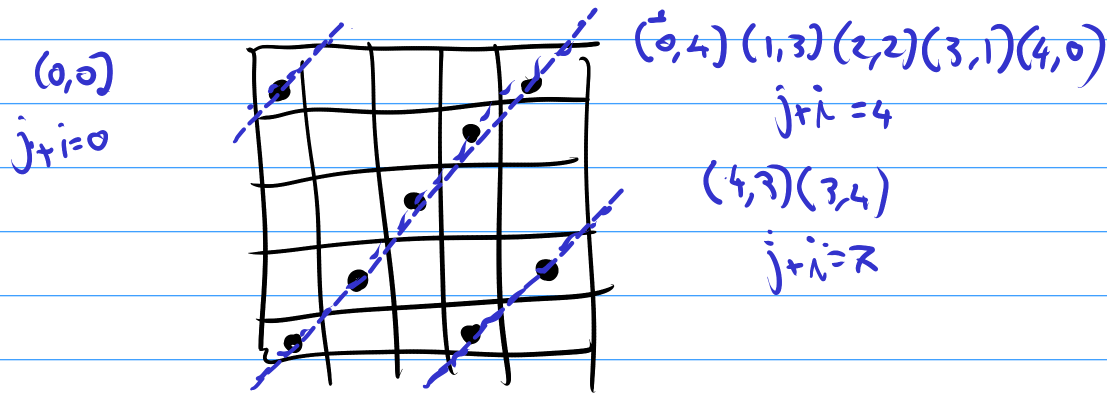
- Stessa riga: se e solo se $i_1 = i_2$
- Stessa colonna: se e solo se $j_1 = j_2$
- Stessa diagonale positiva



Due regine (i_1, j_1) e (i_2, j_2) sono sulla stessa diagonale positiva sse

$$j_1 - i_1 = j_2 - i_2$$

- Stessa diagonale negativa



Due regine (i_1, j_1) (i_2, j_2) sono sulla stessa diagonale negativa sse

$$j_1 + i_1 = j_2 + i_2$$

- Strategia banale:

  tentare tutte le 2^{n^2} matrici $n \times n$ binarie e verificare che non ci siano conflitti

Tuttavia è ovvio osservare che l'unico modo possibile di mettere n regine è (tanto per cominciare)

Ⓐ – esattamente una per riga
Ⓑ – esattamente una per colonna

- Se le posizioni delle regine sono $(0,j_0)(1,j_1)(2,j_2) \cdots (n-1,j_{n-1})$

allora $j_0 \cdots j_{n-1}$ è una permutazione di $\{0,1,2,\dots,n-1\}$

- Usiamo uno schema simile a quello dell'algoritmo per la generazione delle permutazioni.

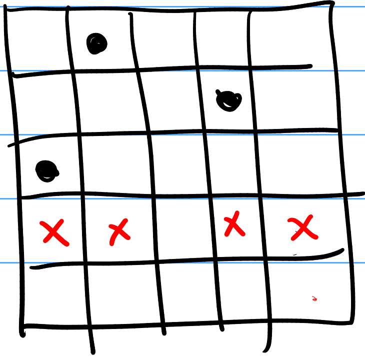
RICORSIONE

- Al passo t abbiamo la sequenza $j_0 j_1 j_2 j_3 \cdots j_{t-1}$
  che indica che le regine sono posizionate SENZA CONFLITTI nelle posizioni

$$(0,j_0) \quad (1,j_1) \quad (2,j_2) \quad (3,j_3) \quad \dots \quad (t-1, j_{(t-1)})$$

dobbiamo decidere quale valore $j \in \{0,\dots,n-1\}$ può essere scelto come j_t.

Es $(0,1)(1,3)(2,0)(3,?)$



$j_3 \neq 0$ perché $(2,0)$
$j_3 \neq 1$ perché $(0,1)$
$j_3 = 2$ ricorsione
$j_3 \neq 3$ perché $(1,3)$
$j_3 \neq 4$ perché $(0,1)$ stessa diag positiva

- Durante la ricorsione manteniamo tre array booleani
  - columns di dimensione n

    columns[j] == True indica che una regina è nella colonna j
  - posdiag di dimensione $2n-1$

    posdiag[t] == True indica che una regina è in posizione (i,j) con $j-i=t$

    [osservate che posdiag ha indici $-(n-1), -(n-1)+1, \dots, 0, 1, 2, \dots (n-1)$]
  - negdiag di dimensione $2n-1$

    negdiag[t] == True indica che una regina è in (i,j) con $j+i=t$

    [negdiag ha indici $0, 1, 2, \dots, 2n-2$]

Es $n=7$ e abbiamo scelto le regine $(0,1)(1,3)(2,0)(3,2)(4,4)$

	•					
			•			
•						
		•				
				•		

columns =

0	1	2	3	4	5	6
V	V	V	V	V		

posdiag =

0	1	2	3	4	5	6	-6	-5	-4	-3	-2	-1
V	V	V									V	V

negdiag =

0	1	2	3	4	5	6	7	8	9	10	11	12
	V	V		V	V			V				

- Non ci sono scelte possibili per la 6ª regina

Il programma principale

```
120 def queens(n):
121     if n==1:
122         return 1
123     columns=[False]*n         # colonne occupate
124     posdiag=[False]*(2*n-1)  # diagonali pos. occupate
125     negdiag=[False]*(2*n-1)  # diagonali neg. occupate
126     conf=[]
127     count = queens_rec(n,0,conf,
128                        columns,posdiag,negdiag)
129     return count
```

MEMORIZZEREMO SOLO $j_0\ j_1\ j_2\ j_3 \cdots j_{(n-1)}$

```
131 def queens_rec(n,i,conf,cols,pdiag,ndiag):
132     if i >= n:
133         return 1
134
135     cnt=0
136     for j in range(n):
137         if cols[j] or pdiag[j-i] or ndiag[j+i]:
138             continue
139
140         conf.append(j)
141         cols[j]    =True
142         pdiag[j-i]=True
143         ndiag[j+i]=True
144
145         cnt += queens_rec(n,i+1,conf,
146                           cols,pdiag,ndiag)
147
148         conf.pop()
149         cols[j]    =False
150         pdiag[j-i]=False
151         ndiag[j+i]=False
152
153     return cnt
154
```

- cnt=0 → Contiamo le soluzioni trovate
- righe 137–138 → LA POSIZIONE (i,j) NON È DISPONIBILE
- righe 140–143 → POSIZIONIAMO LA REGINA SU (i,j)
- righe 148–151 → RIMUOVIAMO LA REGINA DA (i,j)

Solutore di Sudoku

Puzzle Source: https://sandiway.arizona.edu/sudoku/examples.html

			2	6		7		1
6	8			7			9	
1	9				4	5		
8	2		1				4	
		4	6		2	9		
	5				3		2	8
		9	3				7	4
	4			5			3	6
7		3		1	8			

→

4	3	5	2	6	9	7	8	1
6	8	2	5	7	1	4	9	3
1	9	7	8	3	4	5	6	2
8	2	6	1	9	5	3	4	7
3	7	4	6	8	2	9	1	5
9	5	1	7	4	3	6	2	8
5	1	9	3	2	6	8	7	4
2	4	8	9	5	7	1	3	6
7	6	3	4	1	8	2	5	9

- Data una matrice 9×9 con delle celle vuote e delle celle contenenti cifre in $1, 2, \dots, 9$. Completate la matrice in modo che

  Le cifre $1, 2, 3, \dots, 8, 9$ appaiono tutte,

  - in ogni riga
  - in ogni colonna
  - in ogni settore di 3×3 celle

Codifica del problema:

- un problema di sudoku può essere rappresentato come una matrice codificata come una lista contenente 9 liste contenenti 9 numeri.
- ogni cella contiene 0 per marcare che è vuota, altrimenti contiene un numero da 1 a 9

ESERCIZIO Scrivere un programma che dato un problema di Sudoku codificato come sopra,

- Segnali errore se l'input non rispetta i vincoli (e.s. cifra ripetuta su una riga)
- Determini se
  - non ha soluzione
  - ha UNA SOLA soluzione (e in quel caso la emetta)
  - ha più di una soluzione (questo non accade nei buoni SUDOKU)

1			4	8	9			6
7	3						4	
					1	2	9	5
		7	1	2		6		
5			7		3			8
		6		9	5	7		
9	1	4	6					
	2						3	7
8			5	1	2			4

	2		6		8			
5	8				9	7		
				4				
3	7					5		
6								4
		8					1	3
				2				
		9	8				3	6
			3		6		9	

			6			4		
7					3	6		
				9	1		8	
	5		1	8				3
			3		6		4	5
	4		2				6	
9		3						
	2					1		