

Backtrack

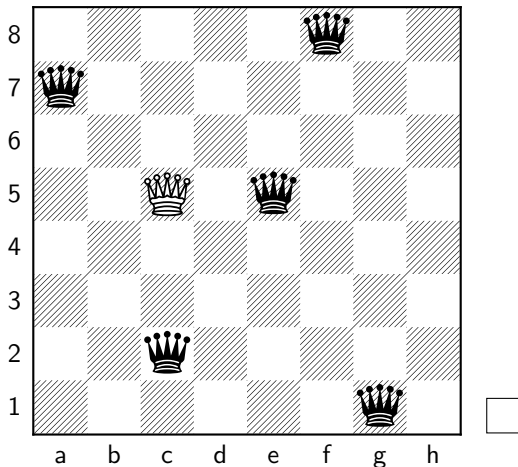
Giuseppe Persiano

Università di Salerno

Novembre, 2021

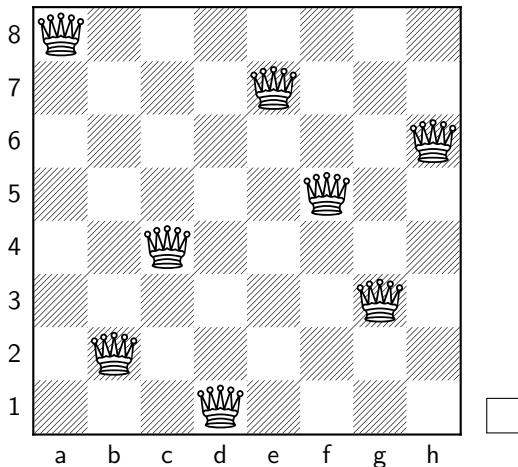
N regine

Piazzare 8 regine su una scacchiera in modo tale che nessuna regina attacchi un'altra regina.



N regine

Piazzare 8 regine su una scacchiera in modo tale che nessuna regina attacchi un'altra regina.



Algoritmo di backtrack

La filosofia del backtrack

- prova tutte le scelte che non sono in conflitto con quelle precedenti
- se non hai alternative, torna indietro a cambia una delle scelte precedenti

8 righe: $0, 1, \dots, 7$

Backtrack applicato al problema delle 8 regine

Prova a piazzare una regina sulla riga R

Regine sulle righe $0, 1, 2, \dots, R - 1$

- se $R = 8$
termina con successo
 - se $R < 8$
per ogni possibile posizione C sulla riga R
 - ▶ se posizione (R, C) non è sotto attacco
 - ★ **annota** dove hai messo la regina e passa ricorsivamente alla riga $R + 1$
 - ★ se la chiamata ricorsiva ha successo, termina con successo
(torna alla chiamata precedente alla riga $R - 1$)
- se hai provato tutte le posizioni, termina con insuccesso
(torna alla chiamata precedente alla riga $R - 1$)

prima chiamata $R = 0$

```
def solve(R, regine):  
    if R==len(regine):  
        return True  
    for C in range(len(regine)):  
        if checkQueen(R, regine, C):  
            regine[R]=C  
            if solve(R+1, regine):  
                return True  
    return False
```

Per risolvere il problema con $N = 8$

```
soluzione=[None]*8  
if solve(R=0, regine=soluzione):  
    print(soluzione)
```

```

##controlla se la posizione della regina nella riga R
##e' compatibile con le regine nelle righe 0,1,...,R-1
def checkQueen(posizioni,R):
    r1=R
    c1=posizioni[R]
    for r in range(R):
        r2=r
        c2=posizioni[r]
        if attackingQ(r1,c1,r2,c2):
            return False
    return True

def attackingQ(r1,c1,r2,c2):
    return samecolumn(r1,c1,r2,c2) or sameMajorD(r1,c1,r2,c2) \
        or sameMinorD(r1,c1,r2,c2)

```

Controllare se due regine si attaccano

Regine a (r_1, c_1) e (r_2, c_2)

- controlla se sono sulla stessa riga:

Controllare se due regine si attaccano

Regine a (r_1, c_1) e (r_2, c_2)

- controlla se sono sulla stessa riga:
 - ▶ $r_1 == r_2$

Controllare se due regine si attaccano

Regine a (r_1, c_1) e (r_2, c_2)

- controlla se sono sulla stessa riga:
 - ▶ $r_1 == r_2$
- controlla se sono sulla stessa colonna:

Controllare se due regine si attaccano

Regine a (r_1, c_1) e (r_2, c_2)

- controlla se sono sulla stessa riga:
 - ▶ $r_1 == r_2$
- controlla se sono sulla stessa colonna:
 - ▶ $c_1 == c_2$

Controllare se due regine si attaccano

Regine a (r_1, c_1) e (r_2, c_2)

- controlla se sono sulla stessa riga:
 - ▶ $r_1 == r_2$
- controlla se sono sulla stessa colonna:
 - ▶ $c_1 == c_2$
- controlla se sono sulla stessa diagonale maggiore:

Controllare se due regine si attaccano

Regine a (r_1, c_1) e (r_2, c_2)

- controlla se sono sulla stessa riga:
 - ▶ $r_1 == r_2$
- controlla se sono sulla stessa colonna:
 - ▶ $c_1 == c_2$
- controlla se sono sulla stessa diagonale maggiore:
 - ▶ $c_1 - r_1 == c_2 - r_2$

Controllare se due regine si attaccano

Regine a (r_1, c_1) e (r_2, c_2)

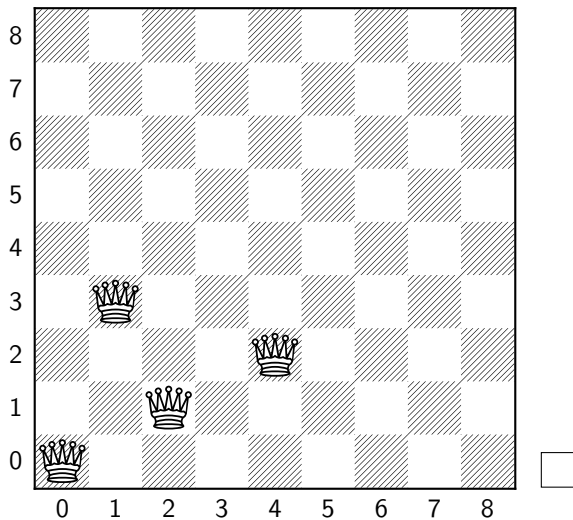
- controlla se sono sulla stessa riga:
 - ▶ $r_1 == r_2$
- controlla se sono sulla stessa colonna:
 - ▶ $c_1 == c_2$
- controlla se sono sulla stessa diagonale maggiore:
 - ▶ $c_1 - r_1 == c_2 - r_2$
- controlla se sono sulla stessa diagonale minore:

Controllare se due regine si attaccano

Regine a (r_1, c_1) e (r_2, c_2)

- controlla se sono sulla stessa riga:
 - ▶ $r_1 == r_2$
- controlla se sono sulla stessa colonna:
 - ▶ $c_1 == c_2$
- controlla se sono sulla stessa diagonale maggiore:
 - ▶ $c_1 - r_1 == c_2 - r_2$
- controlla se sono sulla stessa diagonale minore:
 - ▶ $c_1 + r_1 == c_2 + r_2$

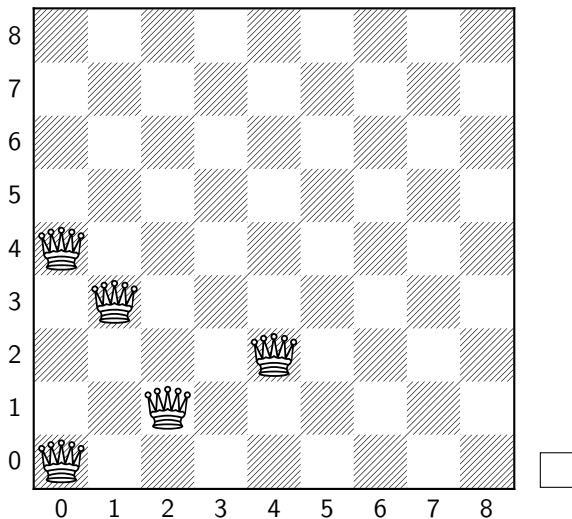
$R = 4$ $\text{regime} = [0, 2, 4, 1, \text{None}, \text{None}, \text{None}, \text{None}, \text{None}]$ $N = 9$



R = 4
C = 0

regine = [0, 2, 4, 1, None, None, None, None]

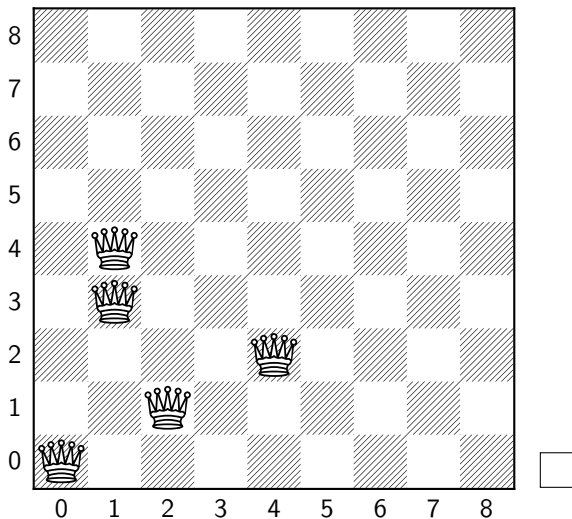
N = 9



R = 4
C = 1

regime = [0, 2, 4, 1, None, None, None, None]

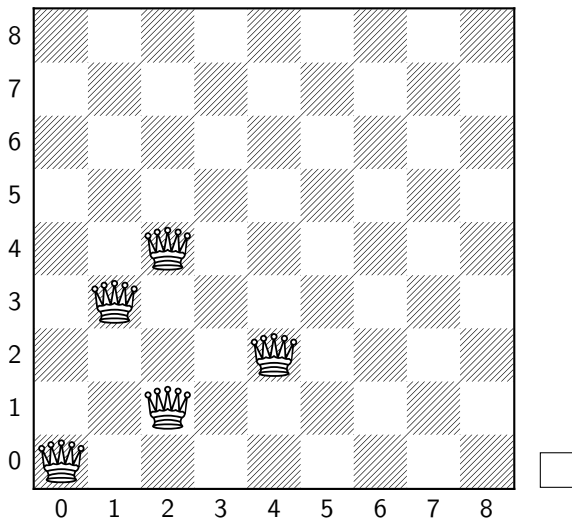
N = 9



R = 4
C = 2

regine = [0, 2, 4, 1, None, None, None, None]

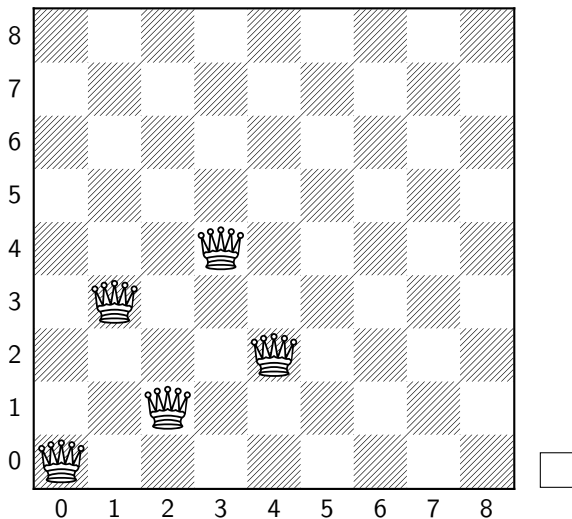
N = 9



R = 4
C = 3

regine = [0, 2, 4, 1, None, None, None, None]

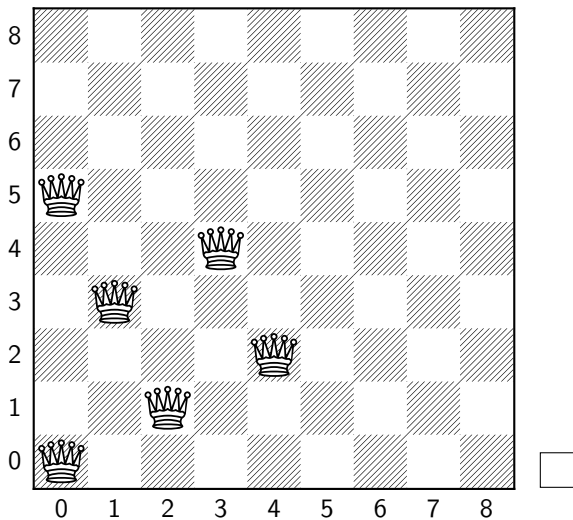
N = 9



R = 5
C = 0

regine = [0, 2, 4, 1, 3, None, None, None, None]

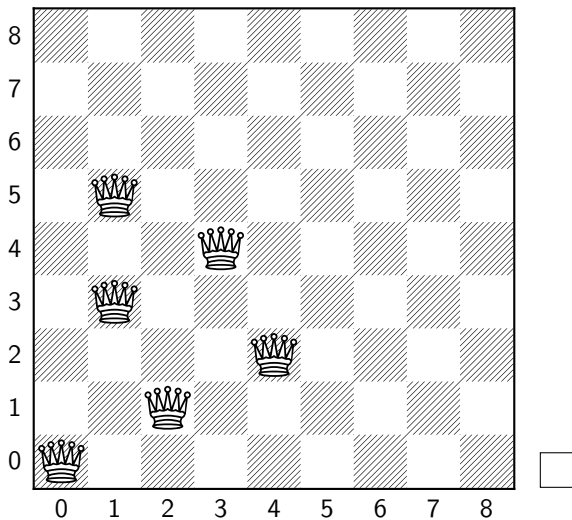
N = 9



R = 5
C = 1

regine = [0, 2, 4, 1, 3, None, None, None, None]

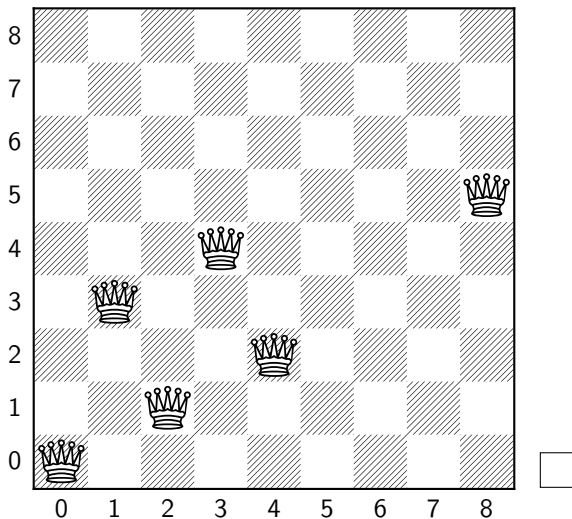
N = 9



R = 5
C = 8

regine = [0, 2, 4, 1, 3, None, None, None]

N = 9

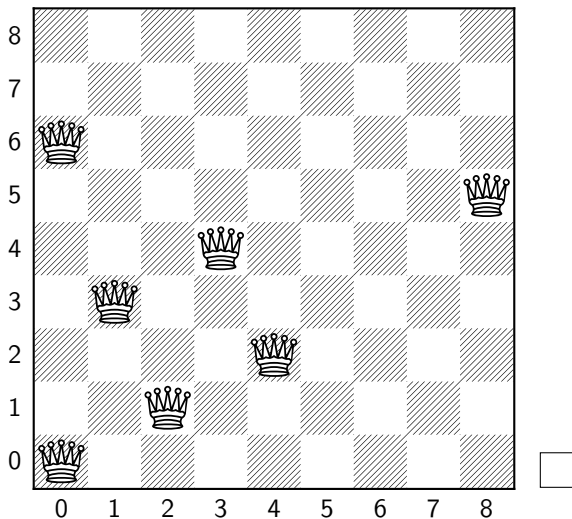


R = 6

regine = [0, 2, 4, 1, 3, 8, None, None, None]

N = 9

C=0

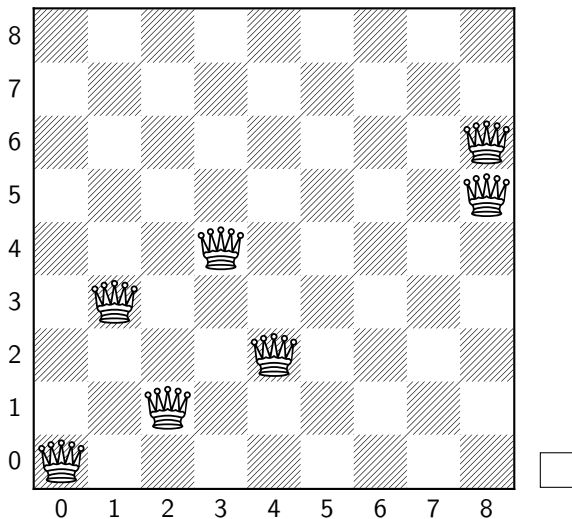


R = 6

regine = [0, 2, 4, 1, 3, 8, None, None, None]

N = 9

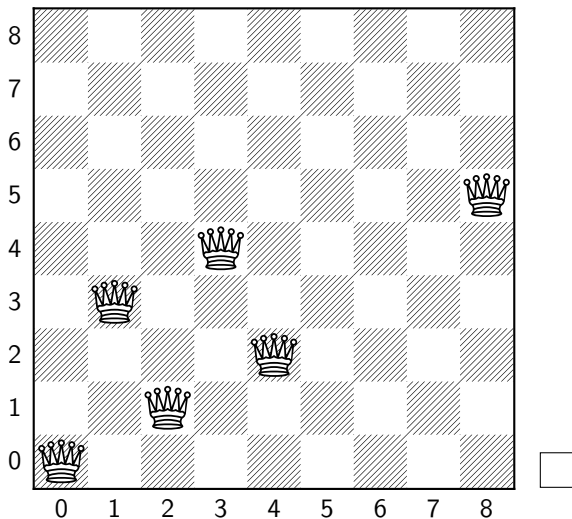
C=8



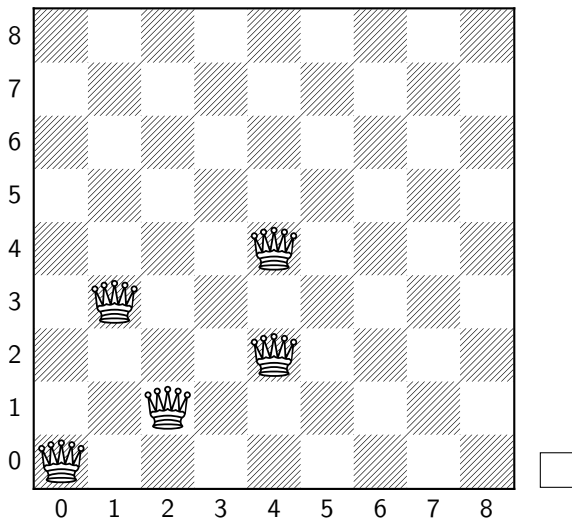
R = 5
C = 8

regine = [0, 2, 4, 1, 3, None, None, None]

N = 9



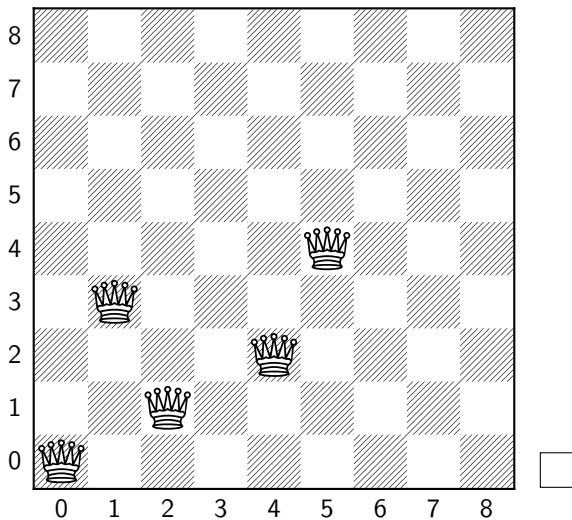
$R = 4$ $\text{regine} = [0, 2, 4, 1, \text{None}, \text{None}, \text{None}, \text{None}, \text{None}]$ $N = 9$
 $C=3 \Rightarrow C=4$



R = 4
C = 5

regine = [0, 2, 4, 1, None, None, None, None]

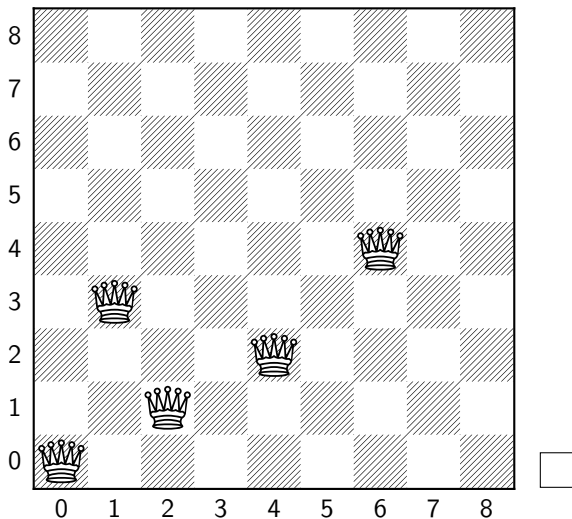
N = 9



R = 4
C = 6

regine = [0, 2, 4, 1, None, None, None, None, None]

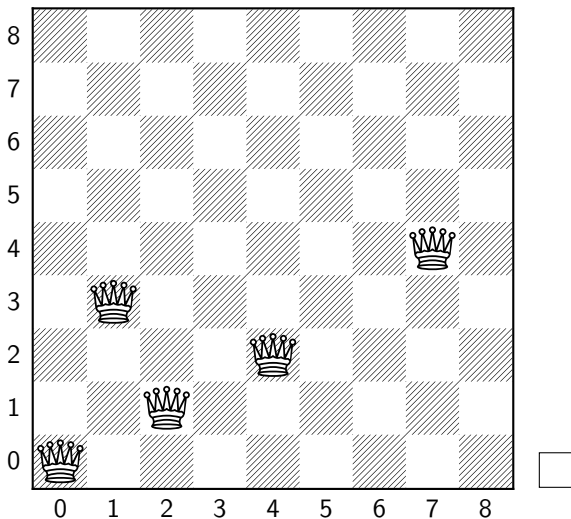
N = 9



R = 4
C = 7

regine = [0, 2, 4, 1, None, None, None, None]

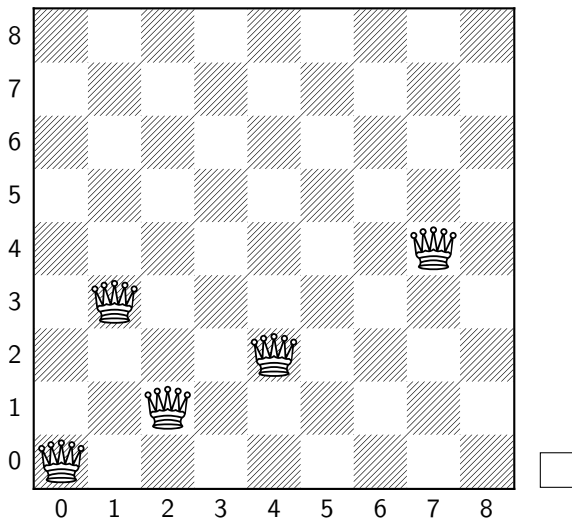
N = 9



R = 5

regine = [0, 2, 4, 1, 7, None, None, None]

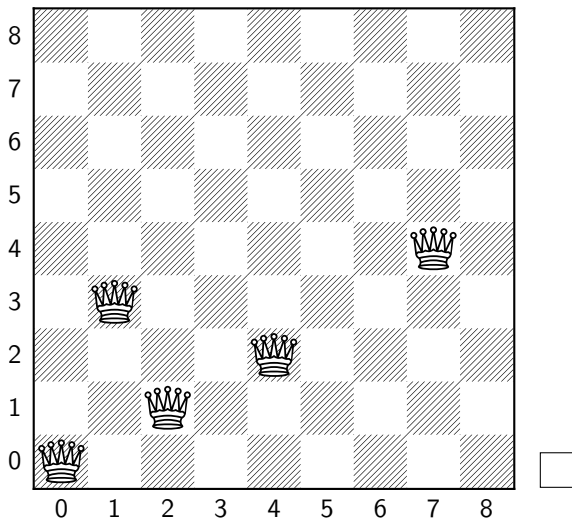
N = 9



R = 5
C = 0

regine = [0, 2, 4, 1, 7, None, None, None, None]

N = 9



Cosa abbiamo fatto?

```
def solve(R, regine):  
    if R==len(regine):  
        return True  
    for C in range(len(regine)):  
        if checkQueen(R, regine, C):  
            regine[R]=C  
            if solve(R+1, regine):  
                return True  
    return False
```

- ad ogni passo dell'algoritmo, abbiamo uno **stato**
 - ▶ `stato = [R, regine]`
- controlliamo se lo stato è **finale**
 - ▶ `R == N`
- per **ogni mossa C** controlliamo se la mossa è **ammissibile** nello **stato**
 - ▶ `for C in range(N)`
 - ▶ `checkQueen(R, regine, C)`
- se è ammissibile **costruiamo** lo stato che si ottiene applicando la mossa ed effettuiamo la chiama ricorsiva

Approccio generale

I 4+1 ingredienti per un Backtrack perfetto

Definizione dello stato

- 1 funzione che controlla se uno stato è finale
- 2 funzione che restituisce la lista di tutte le mosse per uno stato
- 3 funzione che controlla se una mossa è ammissibile per uno stato
- 4 funzione che costruisce il nuovo stato

```
##restituisce una lista
##all'indice 0 True/False
##se l'indice 0 e' uguale a True, l'indice 1 contiene uno stato finale
def _Solve(self,stato):

    if self.isFinal(stato):
        return [True,stato]

    for m in self.allMoves(stato):
        if not self.isAdm(stato,m):
            continue
        newStato=self.newStato(stato,m)
        risultato=self._Solve(newStato,verbose)
        if risultato[0]:
            return risultato

    return [False]
```

Approccio generale

- 1 Costruisco una classe **BackTrack** con un solo metodo

```
class BackTrack:

    #la classe derivata deve implementare
    # allMoves(stato) -- restituisce la lista di mosse nello stato
    # isAdm(stato,m) -- True sse mossa m e' ammissibile in stato
    # newStato(stato,m) -- restituisce lo stato che si ottiene applicando m
    # isFinal(stato) -- Ture sse lo stato e' finale

    ##restituisce una lista
    ##all'indice 0 True/False
    ##se l'indice 0 e' uguale a True, l'indice 1 contiene uno stato finale
    def _Solve(self,stato):
```

- 2 Derivo la classe **Queen** da **BackTrack** e fornisco l'implementazione dei metodi mancanti

Approccio generale

```
from back import BackTrack

##lo stato per questo problema consiste di una lista di due elementi
##all'indice 0 abbiamo una lista che contiene le posizioni delle regine
##gia' decise
##all'indice 1 abbiamo l'indice della prossima regina da considerare
##stato iniziale
##0 --> [None]*n nessuna posizione decisa
##1 --> 0 prossima regina da considerare
## e' la regina in riga 0
```

```
class Queen(BackTrack):
```

```
    def allMoves(self,stato):
        return list(range(self.n))
```

```
    def isAdm(self,stato,m):
        R=stato[1]
        return self._checkQueen(stato[0],R,m)
```

```
    def newStato(self,stato,m):
        newStato=[]
        newStato.append(stato[0].copy())
        R=stato[1]
        newStato[0][R]=m
        newStato.append(R+1)
        return newStato
```

```
    def isFinal(self,stato):
        return stato[1]==self.n
```

Perché?

Vantaggi dell'approccio generale

- scriviamo **BackTrack** una sola volta
- utilizziamo la stessa classe **BackTrack** per più problemi
- basta solo
 - 1 definire lo **stato**
 - 2 implementare le 4 funzioni