Ricerca esaustiva e Backtracking

A.A. 2017-2018

Backtracking: somma di sottinsieme

Somma di sottinsieme: *dati un insieme W e un intero M, trovare tutti i sottoinsiemi di W la somma dei cui elementi sia M.*

I sottoinsiemi sono rappresentati mediante un vettore X i cui elementi possono assumere i valori 0 (presente) o 1 (assente)

Con osservazioni analoghe a quelle fatte per il problema dei sottinsiemi di cardinalità fissata, si può costruire la funzione bounding:

$$B(X[1], \ldots, X[i]) \Leftrightarrow \sum_{k=1}^{i} W[k] \cdot X[k] + \sum_{k=i+1}^{|W|} W[k] >= M$$

Inoltre, se il vettore W è in ordine *non decrescente*, si può definire una seconda funzione bounding:

$$\sum_{k=1}^{i} W[k] \cdot X[k] + W[i+1] <= M$$

Backtracking: somma di sottinsieme

I parametri della procedura seguente sono:

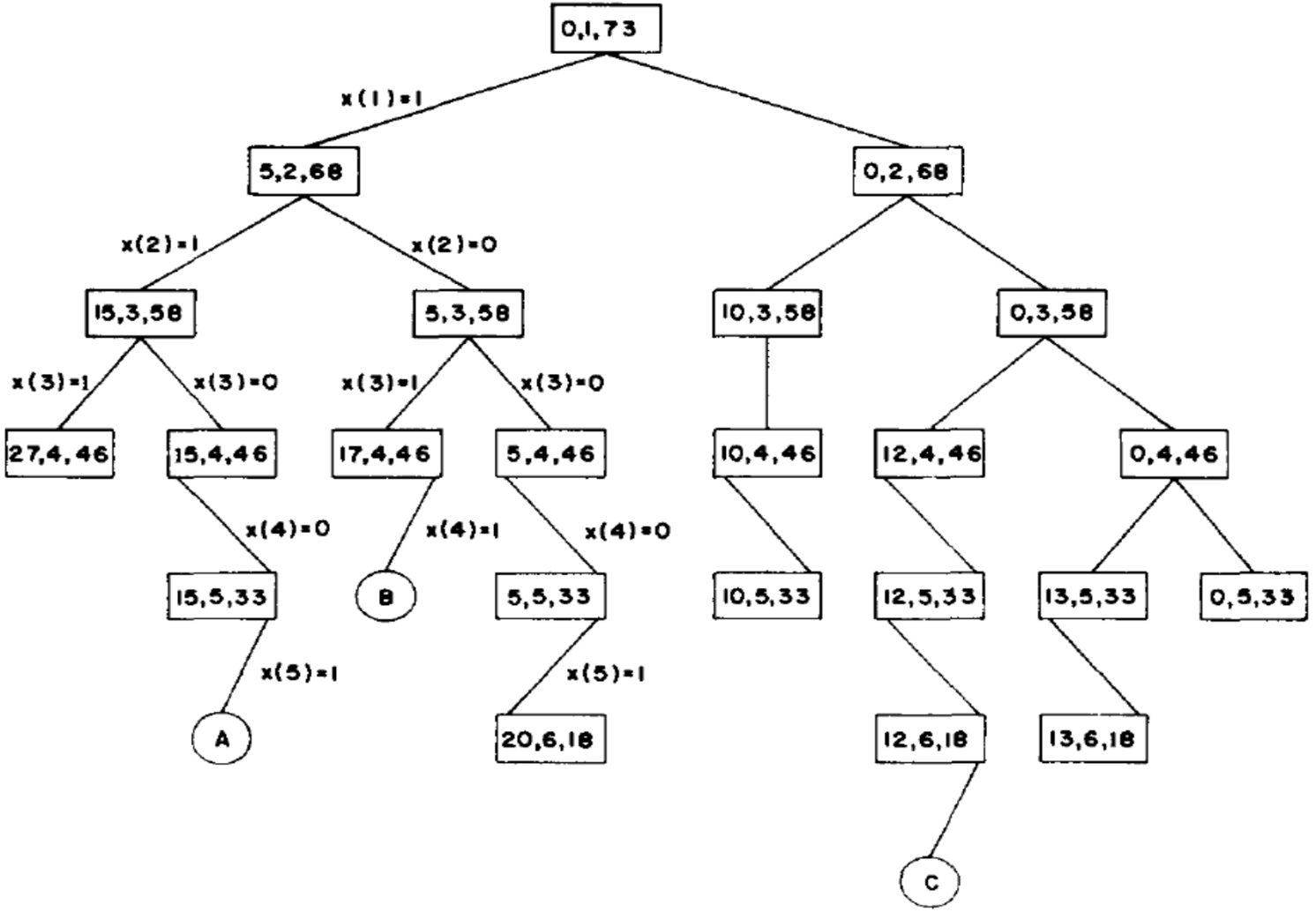
- l'indice i dell'elemento considerato,
- la somma parziale s = $\sum_{k=1}^{i-1} X[k] \cdot S[k]$
- la somma parziale r = $\sum_{k=i}^{n} S[k]$

La chiamata esterna sarà *Sum_Sub* (1, 0, $\sum_{k=1}^{n}$ W[k])

```
Sum_Sub (i, s, r)
    if (i > n) return
    X[i] ← 0      \\ elemento i non considerato
    if (s + r - W[i] >= M) and (s + W[i+1] <= M)
            Sum_Sub (i+1, s, r-W[i])  ;  return
    X[i] ← 1   \\ elemento i  considerato
    if (s + W[i] = M)   X[i+1..n] ← [0..0] ;
                        "stampa X[1], …, X[n]" ; return
    else if ((i < n) and (s + W[i] + W[i+1] <= M))
            Sum_Sub (i+1, s+W[i], r-W[i])
    return
```

N.B. Si suppone che i test vengano effettuati con la logica del "corto circuito".

Backtracking: somma di sottinsieme



Porzione dello spazio degli stati con **n**=6, **W** = 5, 10, 12, 13, 15, 18 e **M**=30

I nodi rettangolari listano i valori di s, k, r

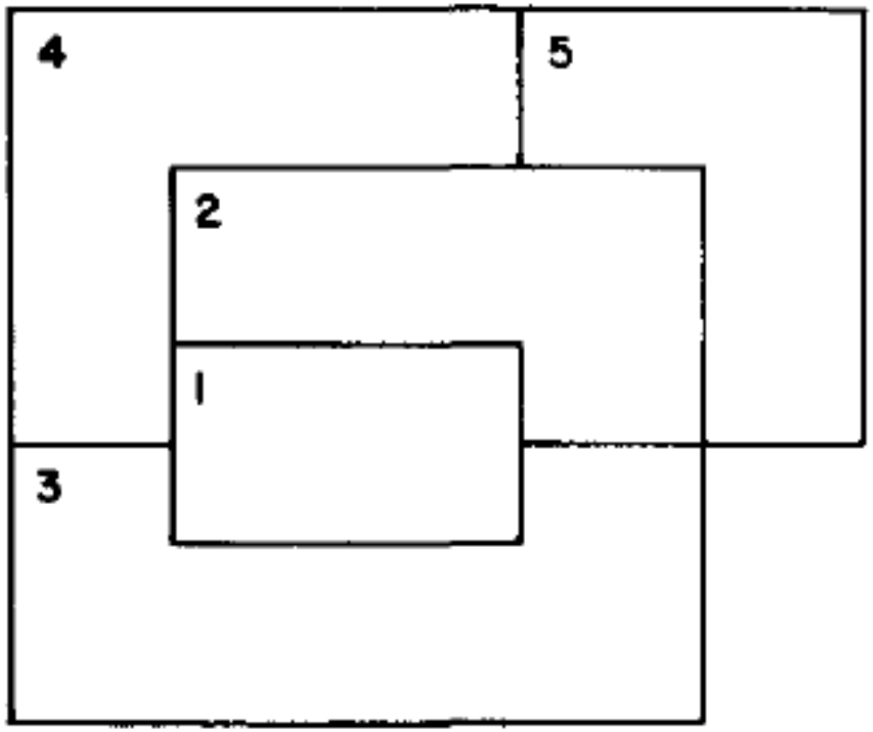
Backtracking: colorabilità

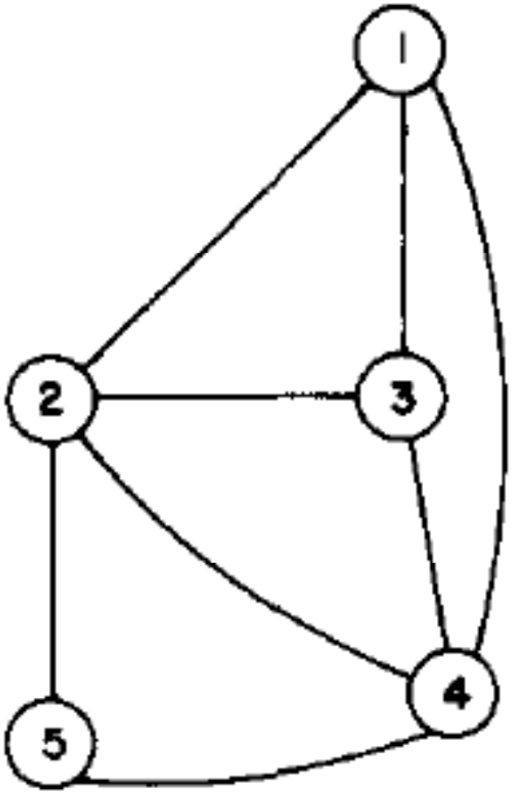
Colorabilità: *dati un grafo **G**, con **n** vertici, e **m** colori, trovare tutti i modi in cui è possibile colorare i vertici di G in modo che i vertici adiacenti abbiano colori diversi.*

```
Color (G, i, X)   \\il vettore X associa colori a vertici
    if (i <= n)
        for j ← 1 to m
            X[i] ← j
            h ← 1
            while (h < i and ((i,h)∉ E or X[i] ≠ X[h]))
                h ← h + 1
            if (h = i)
                if (i = n) "stampa X[1], …, X[n]"
                        Color (G, i+1, X)
    return
```

Chiamata iniziale: *Color*(G, 1, X)

Backtracking: colorabilità





Una mappa e la sua rappresentazione come grafo planare.

Backtracking: colorabilità

Complessità.

Si può dare una valutazione grossolana del tempo di esecuzione nel caso peggiore, contando il numero di chiamate ricorsive.

- La procedura *Color*(i) prova ad assegnare al nodo *i* ognuno degli m colori e, supponendo che siano tutti compatibili, si richiama ogni volta sul nodo *i+1.*
- Trascuriamo il lavoro di $\Theta(i)$ per verificare la compatibilità di ogni scelta.
- *Color*(n+1) eventualmente produce una soluzione e poi esce.

Dette C(i) il numero di chiamate ricorsive di Color(i)

$C(1) = m \cdot C(2) = m \cdot m \cdot C(3) = \ldots = m^n \cdot C(n+1) = m^n$

Quindi abbiamo $O(m^n)$

Backtracking: il caso generale

Schema di algoritmo ricorsivo nel caso in cui si vogliano trovare **tutte** le soluzioni al problema dato.

```
BACKTRACK_ric (i, …)
    if (i <= n)
        "determina I_i (i-mo elemento di una soluzione)…."
        for tutti i valori che X[i] può assumere
            if (<X[1]…X[i]> "può diventare una soluzione")
              if ("è una soluzione")
                  "stampa la soluzione"
               ……."lavoro di preparazione"
              BACKTRACK_ric (i+1, …)
    return
```

Backtracking: colorabilità e il caso generale

Colorabilità: *dati un grafo G, con n vertici, e m colori, trovare tutti i modi in cui è possibile colorare i vertici di G in modo che i vertici adiacenti abbiano colori diversi.*

```
Color (i, X)
    if (i <= n)
        for j ← 1 to m
            X[i] ← j
            h ← 1
            while (h < i and ((i,h) ∉ E or X[i] ≠ X[h]))
                h ← h + 1
            if (h = i)
                if (i = n) "stampa X[1], …, X[n]" ; return
            Color (i+1, X)
    return
```

"*determina le scelte per X[i] ….*"

per tutti i valori che X[i] *può assumere*

X[1]…X[i] *può diventare una soluzione?*

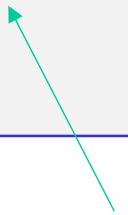
è una soluzione?

Chiamata iniziale: *Color*(1, X)

Backtracking: somma di sottinsieme e il caso generale

```
Sum_Sub (i, s, r)
        if (i > n) return
        X[i] ← 0
        if (s + r - W[i] >= M) and (s + W[i+1] <= M)
                    Sum_Sub (i+1, s, r-W[i])
        X[i] ← 1
        if (s + W[i] = M)  X[i+1..n] ← [0..0]
                           "stampa X[1], …, X[n]"
        else if ((i < n) and (s + W[i] + W[i+1] <= M))
                        Sum_Sub (i+1, s+W[i], r-W[i])
        return
```

```
Sum_Sub (i, s, r)
        if (i <= n)
                X[i] ← 1
                s ← s + W[i]
                r ← r - W[i]
                if (s = M)  X[i+1..n] ← [0..0]
                            "stampa X[1], …, X[n]"
                else if ((i < n) and (s + W[i+1] <= M))
                            Sum_Sub (i+1, s, r)
                X[i] ← 0
                s ← s - W[i]
                if (s + r >= M) and (s + W[i+1] <= M)
                            Sum_Sub (i+1, s, r)
        return
```



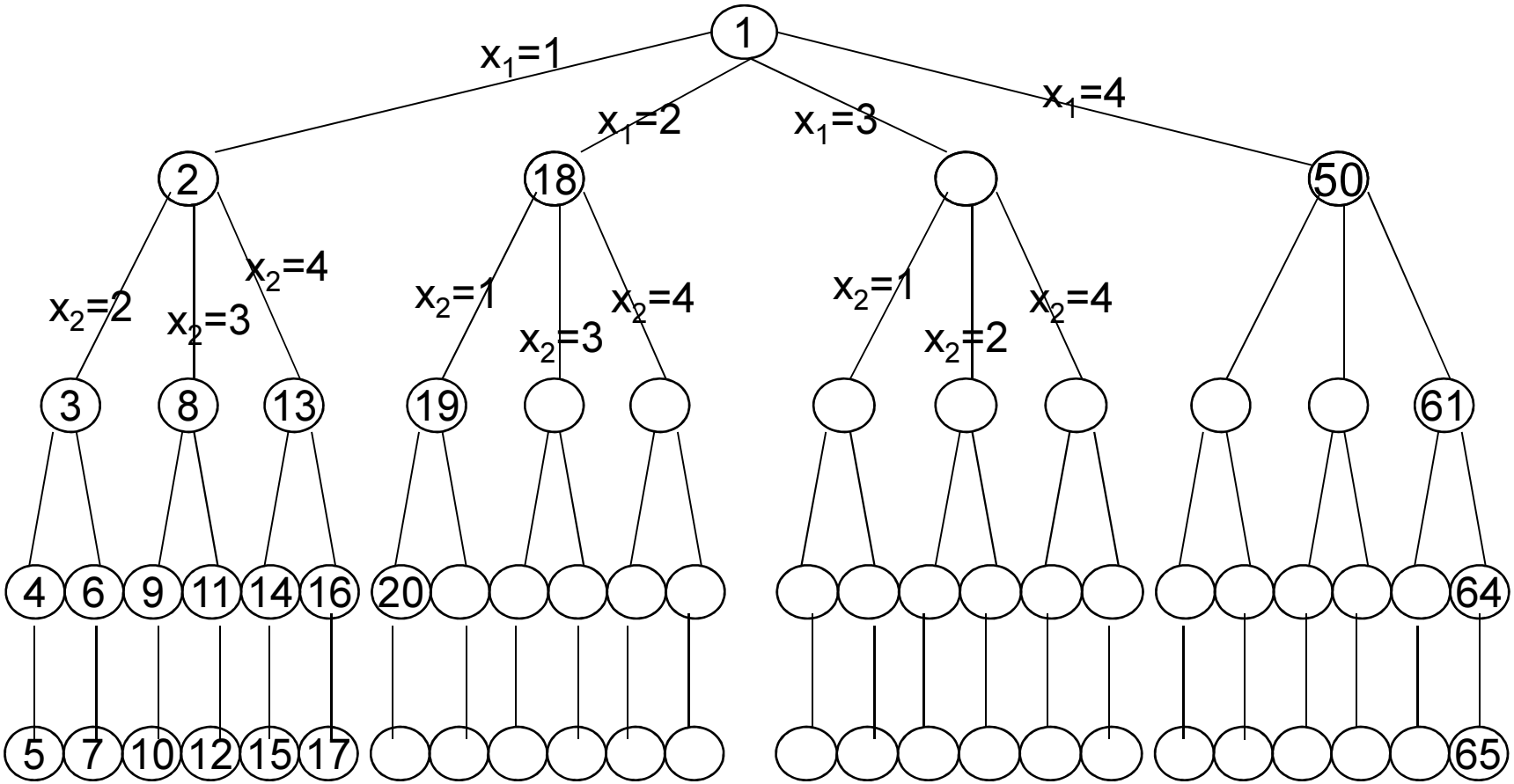
Scema generale

Backtracking: n regine

Problema delle n regine: *Data una scacchiera n x n, posizionare su di essa n regine in modo che ci sia una e una sola regina su ogni riga e nessuna di esse sia sotto scacco, cioè non vi siano mai più di una regina su ogni colonna e su ogni diagonale della scacchiera.*

Possiamo considerare "*una e una sola regina su ogni riga*" come vincolo esplicito, e "nessuna regina è sotto scacco" come vincolo implicito.

Backtracking: n regine



Albero di ricerca esaustiva per il problema delle n-regine, nel caso particolare n = 4.

Adattiamo lo schema ricorsivo al problema delle n regine:

```
{n > 1}
n-regine-ric (n, i)                          {X[1..n] serve a ricordare le soluzioni}
    if  (i <= n)
        "determina I_i ...."                 {l'insieme delle scelte è, per ogni riga i,
        for X[i] ← 1 to n                     l'insieme delle colonne {1,…,n}}
            if ([i, X[i]] "è sicura ")
                "posiziona la regina su [i, X[i]]"
                if  (i = n)   "stampa la soluzione"
                n-regine-ric (n, i+1, …)
    return
```

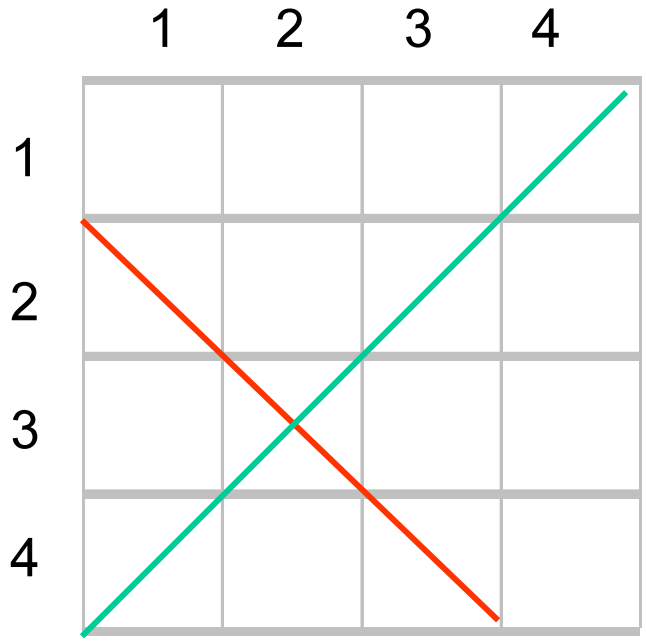
{tutte le disposizioni ammesse sono state generate}

Backtracking: n regine

È facile verificare se una colonna è libera o no.

Più complicato è esprimere che una diagonale è “libera” o “occupata”

Nella scacchiera sono presenti due tipi di diagonali: *ascendenti* (es. diagonale verde) e *discendenti* (es. diagonale rossa)



Backtracking: n regine

Osservazioni:

- la differenza degli indici di riga e di colonna degli elementi di una diagonale *discendente* si mantiene costante perciò, per due elementi (i, j) e (h, k), si ha i - j = h – k

- la somma degli indici di riga e di colonna degli elementi di una diagonale *ascendente* si mantiene costante perciò, per due elementi (i, j) e (h, k) si ha i + j = h + k

$$i - j = h - k \rightarrow i - h = j - k \qquad i + j = h + k \rightarrow i - h = k - j$$

$$\Downarrow$$

$$|i - h| = |k - j|$$

L'uguaglianza |i - h| = |k - j| permette di verificare se la posizione (i, j) è su una diagonale già occupata.

Backtracking: n regine

Otteniamo così l'algoritmo:

```
n-regine-ric-1 (n, i)
  if (i <= n)
     for X[i] ← 1 to n
          if (P-sicura (i, X[i]))
               if  i = n  print "la disposizione"
               n-regine-ric (n, i+1)
  return
```

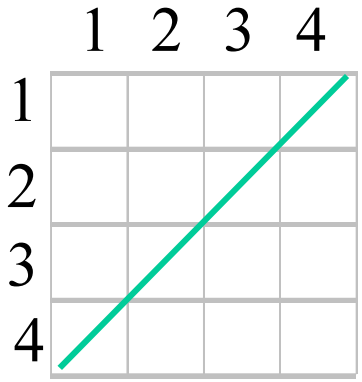
Nuova regina

```
P-sicura (r, c)
   for h ← 1 to r - 1
          if ((X[h] = c) or (|X[h] – c| = |h – r|))
               return false
   return true
```

Proposta alternativa:

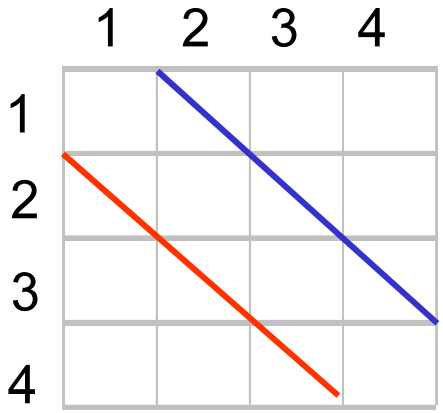
Per verificare se una colonna è libera usiamo un vettore C[1..n] di booleani, con *true* che significa colonna “libera” e *false* che indica colonna “occupata”.

Ogni diagonale ascendente può essere identificata in modo univoco da un intero; in particolare, poichè la somma degli indici di riga e di colonna dei suoi elementi si mantiene costante, si può usare il numero i + j - 1.
Alla diagonale evidenziata viene associato il numero 4.



In tal modo ogni diagonale *ascendente* viene identificata da un numero da 1 a 2n-1.

In modo analogo si possono identificare con numeri da 1 a 2n-1 le diagonali *discendenti,* associando ad ognuna di esse il numero *n* - *j* + *i* (per le diagonali discendenti è costante la differenza tra gli indici di riga e di colonna).
Alla diagonale evidenziata in rosso viene associato il numero 5, il numero 3 alla diagonale in blu.



Allora i vincoli

"Una e una sola regina su ogni diagonale ascendente e su ogni diagonale discendente"

si possono esprimere usando due vettori *booleani*:

A[1..2n-1] e D[1..2n-1]

La posizione [3, 5] sulla scacchiera (di lato 8) sarà libera per la regina sulla riga 3 se sono "true" gli elementi C[5], A[7] e D[6].

Backtracking: n regine

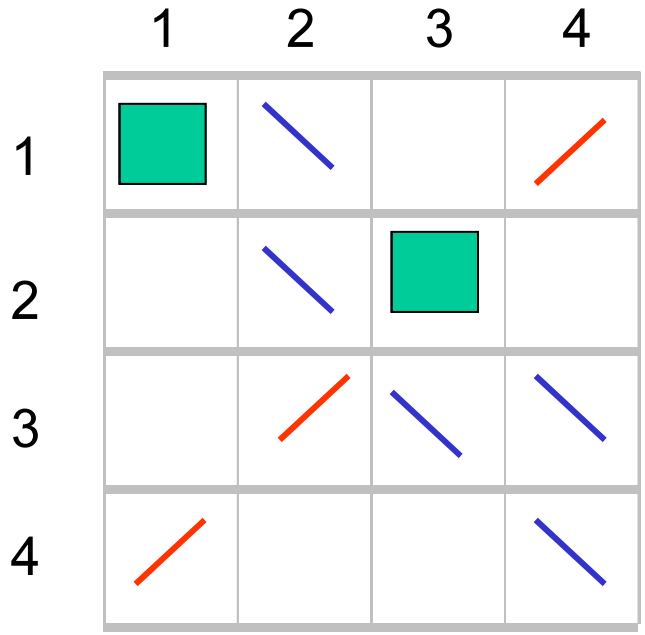
```
n-regine-ric (n, i)
    if (i <= n)
        for j ← 1 to n
            if (C[j] and A[i+j-1] and D[n-j+i])
                X[i] ← j
                C[j] ← A[i+j-1] ← D[n-j+i] ← false
                if  i = n  stampa "la disposizione"
                n-regine-ric (n, i+1)
                C[j] ← A[i+j-1] ← D[n-j+i] ← true
    return
```

Attenzione: si crea un side-effect negativo

Si deve rimediare prima di provare un'altra posizione

Backtracking: n regine

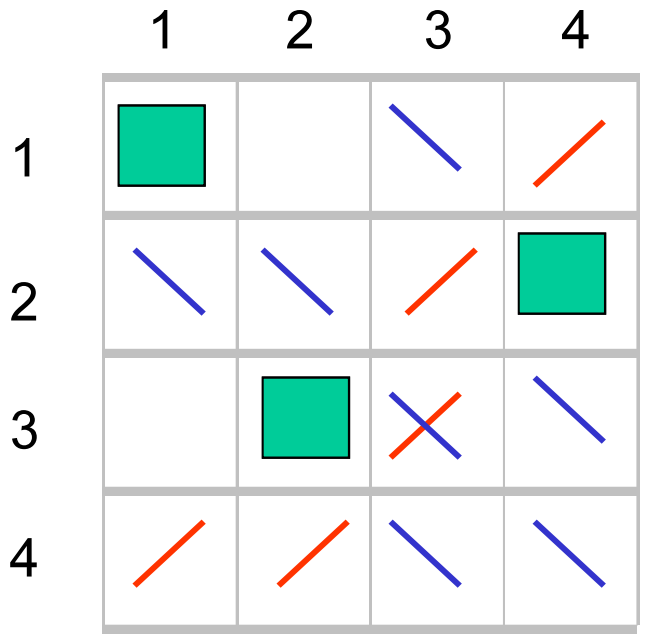
Applichiamo l'algoritmo all'istanza del problema per n = 4:



Con le regine nelle posizioni [1, 1] e [2, 3] non si ha nessuna posizione sicura per la terza regina.

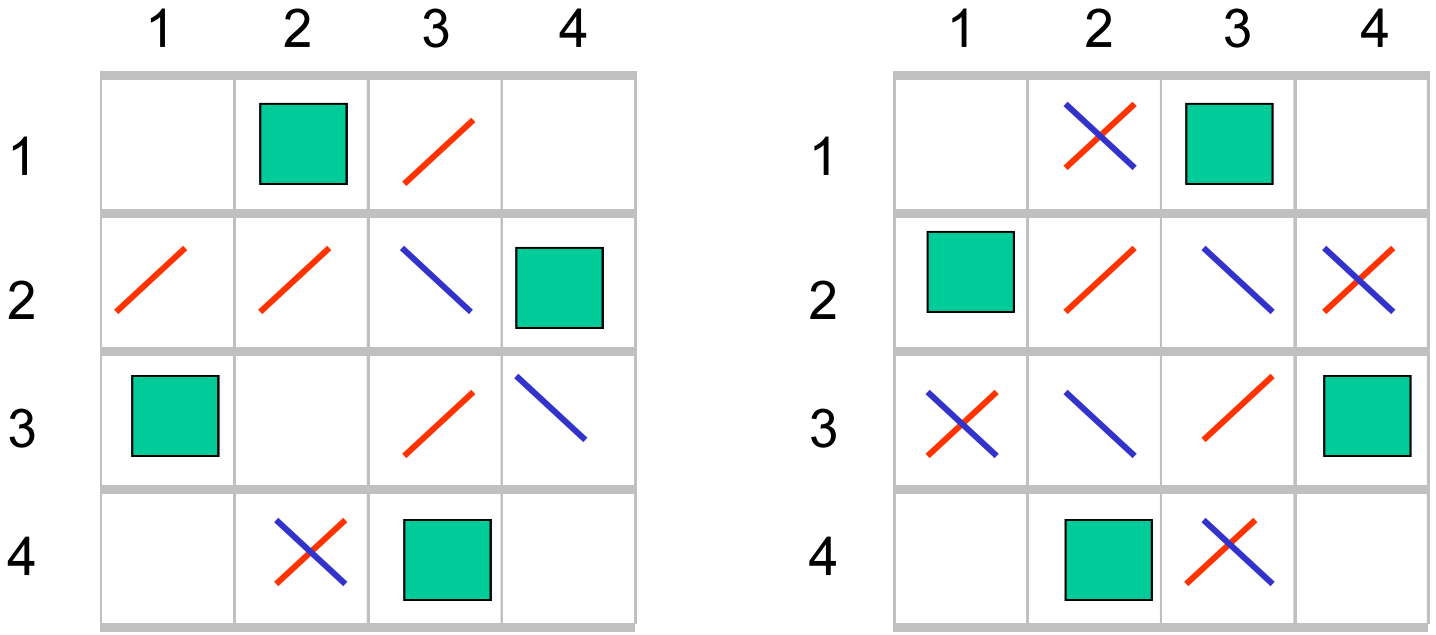
Backtracking: n regine

Con la regina nella posizione [1, 1] non si hanno soluzioni.



Backtracking: n regine

Soluzioni però esistono:



La corrispondente versione iterativa:

```
n-regine_it (n)
    i ← 1
    X[1] ← 0
    while (i > 0)
            X[i] ← X[i]+1
            while (X[i] <= n) and not (P-sicura (i, X[i]))  do
                    X[i] ← X[i] + 1
                    \\trova la prima posizione sicura per reg. i
             if i <= n then
                    if  i = n  print "la disposizione"
                          else i ← i + 1   \\sistema un'altra regina
                                X[i] ← 0
                 else i  ← i-1  \\ tenta un'altra possibilità
return
```