UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di Ingegneria
"Enzo Ferrari"

Fondamenti di Informatica II

Algoritmi di Backtracking

INTRODUZIONE

Classi di problemi (decisionali, di ricerca, di ottimizzazione)

– Definizioni basate sul concetto di soluzione ammissibile: una soluzione che soddisfa un certo insiemi di criteri

Problemi "tipici"

– Contare le soluzioni ammissibili
– Costruire almeno una o tutte le soluzioni ammissibili
– Trovare le soluzioni ammissibili "più grandi", "più piccole" o in generale "ottimali"

Esempio:

– Elencare tutti i sottoinsiemi di k elementi di un insieme S

TIPOLOGIE DI PROBLEMI (1)

Enumerazione

Elencare tutte le possibili soluzioni ammissibili (spazio di ricerca)

Costruire almeno una soluzione

Si utilizza l'algoritmo per elencare tutte le soluzioni, fermandosi alla prima

Contare le soluzioni

In molti casi, è possibile contare in modo analitico

Esempio: |S| = n, # sottoinsiemi di k elementi: $\frac{n!}{(n-k)!k!}$

In altri casi, si costruiscono tutte le possibili soluzioni e si contano

TIPOLOGIE DI PROBLEMI (2)

Trovare le soluzioni ottimali

Si costruiscono tutte le soluzioni e si valuta una funzione di costo (brute force)

Altre tecniche: Programmazione dinamica, Greedy

Esempio Ciclo hamiltoniano (commesso viaggiatore)

Dato un grafo G=(V,E), si definisce

hamiltoniano un cammino $p=\langle v_1, v_2, \ldots, v_k \rangle$ tale per cui

$$|V| = k + 1, \text{ for all } i,j\ v_i \neq v_j \text{ e } v_1 = v_{k+1}$$

In altre parole, un cammino hamiltoniano tocca tutti i vertici del grafo una ed una sola volta chiudendosi sul nodo di partenza.

MOTIVAZIONE

Approccio "brute-force": esaminare interamente lo spazio delle possibili soluzioni

Perché no:

Spesso e volentieri non è necessario; la soluzione potrebbe essere trovata, ad esempio, tramite una tecnica "greedy".

Perché sì:

A volte è l'unica strada possibile

La potenza dei computer moderni rende "affrontabili" problemi di notevoli dimensioni

- $10! = 3.63 \cdot 10^6$ permutazione di 10 elementi
- $2^{20} = 1.05 \cdot 10^6$ tutti i sottoinsiemi di 20 elementi

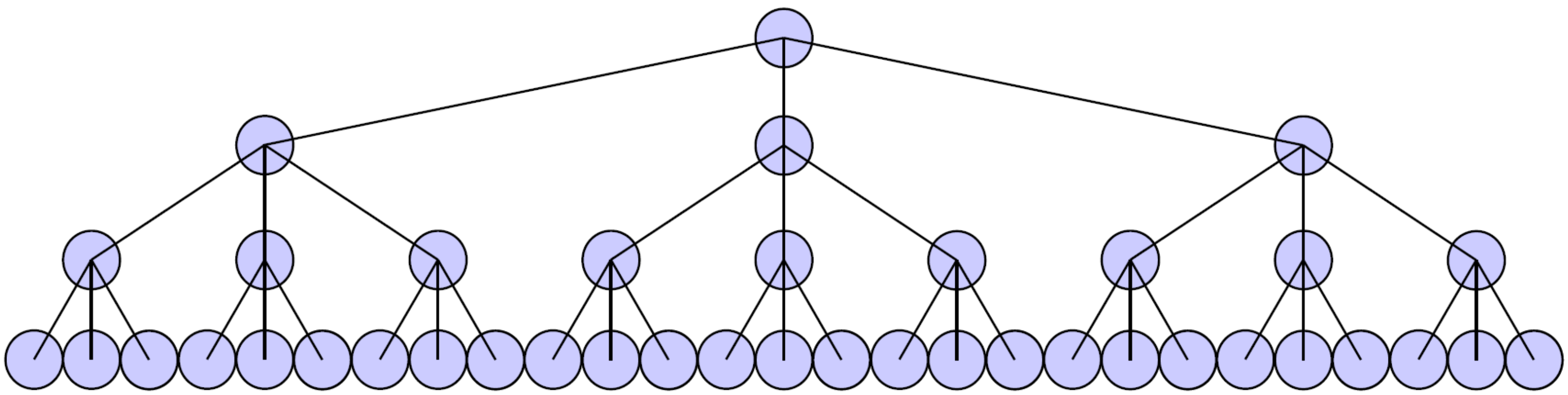
BACKTRACKING (1)

Spazio di ricerca ≡ albero di decisione

Soluzioni ≡ foglie in un albero di decisione

Soluzioni parziali ≡ nodi interni dell'albero di decisione

Radice ≡ soluzione parziale vuota

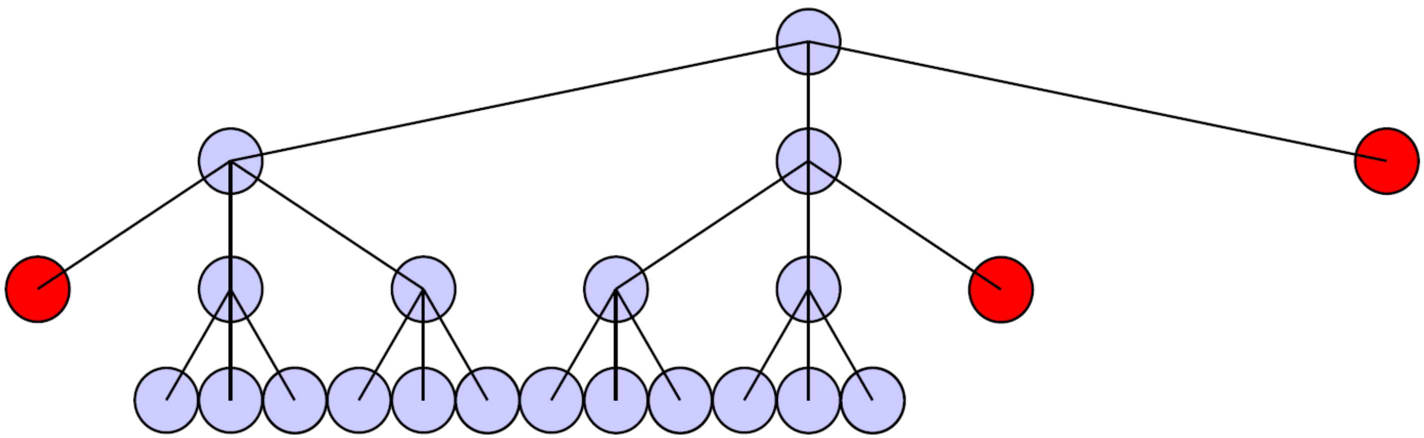


BACKTRACKING (2)

Lo spazio di ricerca può essere ridotto:

“Rami” dell'albero che sicuramente non portano a soluzioni ammissibili possono essere “potati” (**pruned**)

La valutazione viene fatta nelle soluzioni parziali radici del sottoalbero da potare



BACKTRACKING (3)

Cos'è?

Un metodo sistematico per iterare su tutti le possibili istanze di uno spazio di ricerca

Filosofia:

“Prova a fare qualcosa, e se non va bene, disfalo e prova qualcos'altro”

Come funziona?

Lavora tramite una visita in profondità nell'albero delle soluzioni basata su un approccio ricorsivo

E' una tecnica algoritmica che, come altre, deve essere personalizzata per ogni applicazione individuale

BACKTRACKING (4)

Organizzazione generale – struttura dati

- n: lunghezza della sequenza delle soluzioni
- vcurr[0..n-1] Soluzione attuale rappresentata come un vettore
- i: indice intero che esplora tutti gli n elementi
- Il contenuto degli elementi vcurr[i] è preso da un insieme di scelte C dipendente dal problema

Esempi:

- C insieme generico, possibili soluzioni **permutazioni di C**
- C insieme generico, possibili soluzioni **sottoinsiemi di C**
- C mosse di gioco, possibili soluzioni **sequenze di mosse**
- C archi di un grafo, possibili soluzioni **percorsi sul grafo**

BACKTRACKING (5)

Come procedere: ad ogni passo, partiamo da una soluzione parziale vcurr[1..k] e cerchiamo di estenderla a vcurr[1..k+1]

Se vcurr[1..k] è una soluzione ammissibile, la “processiamo” in qualche modo e abbiamo finito

Altrimenti:

- Se è possibile, estendiamo vcurr[1..k] con una delle possibili scelte in una soluzione vcurr[1..k+1], e valutiamo la nuova soluzione ricorsivamente
- Altrimenti, “cancelliamo” (ritornando indietro nella ricorsione) l'elemento S[k] (backtrack) e ripartiamo dalla soluzione S[1..k-1]
- Si può fare backtrack su più passi di ricorsione

BACKTRACKING (6)

Due possibili approcci:

Ricorsivo:

- lavora tramite una visita in profondità nell'albero delle scelte, basata su un approccio ricorsivo

Iterativo:

- utilizza un approccio greedy, eventualmente tornando sui propri passi

ALGORITMO GENERALE

```
void BacktrackTutte(int n, int i, int k, int vcurr[], int *nsol){

    if (i == n) { // stampa soluzione
        printf("%d) ", *nsol);
        (*nsol)++;
        for(int j=0;j<n;j++){
            printf("%d ", vcurr[j]); }
        printf("\n");
        return; }

    for(int j=0;j<k;j++){
        vcurr[i] = j; // scelta del valore j (da 0 a k-1)
                  // per il passo i (da 0 a n-1)

        BacktrackTutte(n, i+1, k, vcurr, nsol);}
}
```

vcurr[0 .. n-1] contiene la soluzione parziale vcurr[0 .. i-1]

0 .. k-1 possibili valori per la generica scelta vcurr[i]

ESEMPIO 1: Elencare tutti i sottoinsiemi di un insieme S: 2^n

```
void BacktrackSubset(int n, int i, int vcurr[], int *nsol){

    if (i == n) { // stampa soluzione
        printf("%d) ", *nsol);
        (*nsol)++;
        for(int j=0;j<n;j++){
            printf("%d ", vcurr[j]); }
        printf("\n");
        return; }

    for(int j=0;j<2;j++){
        vcurr[i] = j; // scelta del valore j (da 0 a 1)
                     // per il passo i (da 0 a n-1)

       BacktrackSubset(n, i+1, vcurr, nsol);}
}
```

ESEMPIO 1 (seconda versione): Elencare tutti i sottoinsiemi di un insieme S: 2^n

```
void BacktrackSubset(int n, int i, int vcurr[], int *nsol){

    if (i == n) { // stampa soluzione
        printf("%d) ", *nsol);
        (*nsol)++;
        for(int j=0;j<n;j++){
            printf("%d ", vcurr[j]); }
        printf("\n");
        return; }

    vcurr[i] = 0;
    BacktrackSubset(n, i+1, vcurr, nsol);
    vcurr[i] = 1;
    BacktrackSubset(n, i+1, vcurr, nsol);
}
```

ESEMPIO 2

Problema:

- Elencare tutti i sottoinsiemi di dimensione k di un insieme S

Soluzione basata su algoritmo backtracking generale:

- Possiamo potare?
- Sì, quando al passo i abbiamo scelto k elementi possiamo terminare

ESEMPIO 2

```
void SubsetK(int n, int i, int k, int vcurr[],
                int count, int *nsol){
int j;
     if (count == k) { // stampa soluzione
          printf("%d) ", *nsol);
          (*nsol)++;
          for(int j=0;j<i;j++){
               printf("%d ", vcurr[j]); }
          for(int j=i;j<n;j++){
               printf("0 "); }
     printf("\n");
          return; }
     if (i == n) return;

     vcurr[i] = 0;
     SubsetK(n, i+1, k, vcurr, cont, nsol);

     vcurr[i] = 1;
     count++;
     SubsetK(n, i+1, k, vcurr, cont, nsol);
}
```

ESEMPIO 3: stampare tutte le permutazioni di un insieme A

```
void Permutazioni(int n, int i, int vcurr[], int *nsol){
int j, tmp;
    if (i == n) { // stampa soluzione
        printf("%d) ", *nsol);
        (*nsol)++;
        for(int j=0;j<n;j++){
            printf("%d ", vcurr[j]); }
        printf("\n");
        return; }
    for(int j=i;j<n;j++){
        tmp = vcurr[i]; vcurr[i] = vcurr[j]; vcurr[j]= tmp;
        Permutazioni(n, i+1, vcurr, nsol);
        tmp = vcurr[i]; vcurr[i] = vcurr[j]; vcurr[j]= tmp; }
}
```

ESEMPIO 4: Subset Sum

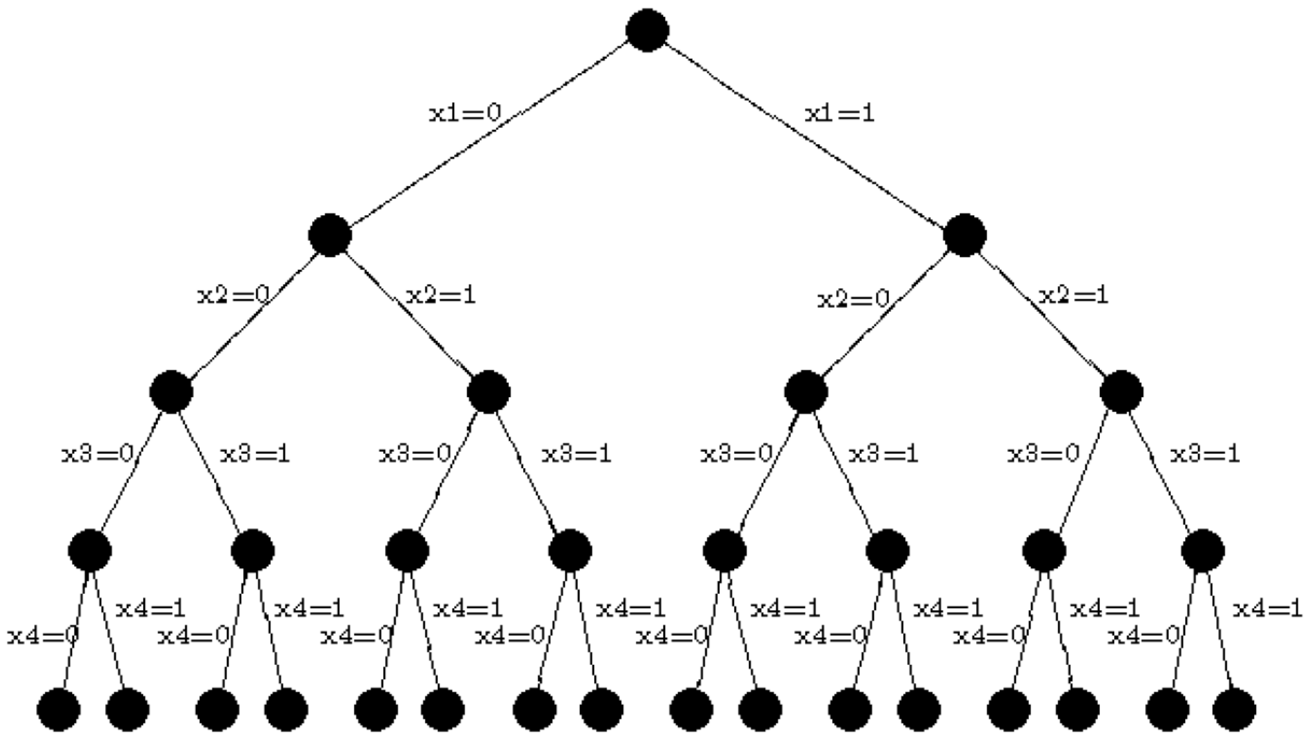
Problema: Sono dati in input n interi positivi $w_1 \dots w_n$ ed un intero target M. Si vuole visualizzare tutti i sottoinsiemi degli interi w_i la cui somma è esattamente M

La soluzione è descritta da vcurr[] binario: se l'i-esima componente è uguale a 0, l'i-esimo elemento non appartiene al sottoinsieme soluzione; se invece l'i-esima componente e uguale a 1, l'i-esimo elemento appartiene al sottoinsieme soluzione.

Per ottimizzare la soluzione, ipotizziamo di ordinare in modo crescente gli elementi w_i

ESEMPIO 4: Subset Sum

Se abbiamo n = 4 interi positivi (7; 11; 13; 24) ed M = 31, il problema Subset Sum ammette 2 soluzioni(7; 11; 13) e (7; 24).



ESEMPIO 4: Subset Sum

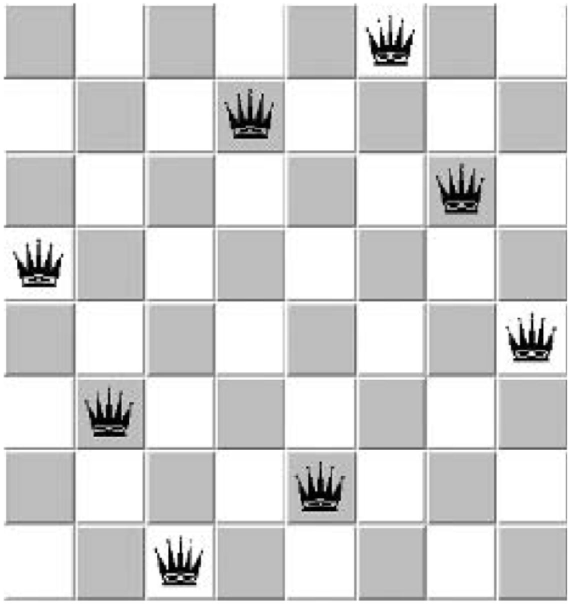
```
void BacktrackSomma(int n, int i, int vcurr[], int w[], int obiettivo,
                int sommacurr, int rimanenza, int *nsol){
      if (sommacurr == obiettivo) { // stampa soluzione
            printf("%d) ", *nsol);
            (*nsol)++;
            for(int j=0;j<i;j++){
                  printf("%d ", vcurr[j]); }
            for(int j=i;j<n;j++){
                  printf("0 "); }
            printf("\n");
            return; }
      if (i == n) return;
      rimanenza -= w[i];
      if ((sommacurr + rimanenza >= obiettivo) &&
           (i == n-1 || (i < n-1 && sommacurr + w[i+1] <= obiettivo))){
            vcurr[i] = 0;
            BacktrackSomma(n, i+1, vcurr, w, obiettivo, sommacurr,
                          rimanenza, nsol);
      }
      if (sommacurr + w[i] <= obiettivo) {
            vcurr[i] = 1;
            BacktrackSomma(n, i+1, vcurr, w, obiettivo, sommacurr + w[i],
                          rimanenza, nsol);
      }
}
```

Problema n Regine

Problema: posizionare n regine in una scacchiera n x n, in modo tale che nessuna regina ne "minacci" un'altra.

Commenti storici:

- Il problema classico, con n=8, è stato studiato, fra gli altri, anche da Gauss (che trovò 72 soluzioni)
- Non esiste una soluzione analitica (92 per n=8)



Metodo

Partiamo dall'approccio più semplice,

e mano a mano raffiniamo la soluzione.

Problema n Regine

Idea: ci sono n^2 caselle dove piazzare una regina

Algoritmo:

- vcurr[1..n^2] array binario vcurr[i] = true ⇒ "regina in vcurr[i]"
- Soluzione trovata se $i = n^2$
- Scelta(vcurr, n, i) ritorna {true}
  se regina in vcurr[i] non minaccia
  le regine in vcurr[1..i-1],
  {false} altrimenti
- # scelte per n=8 $2^{64} \sim 1.84 \cdot 10^{19}$

Commenti:

Problema di rappresentazione

Matrice binaria molto sparsa

Problema n Regine

Idea: Dobbiamo piazzare n regine, ci sono n^2 caselle

Algoritmo:

- vcurr[1..n] coordinate in $1..n^2$ — vcurr[i] = coordinata della regina i
- Soluzione trovata — se i = n
- Scelta(vcurr, n, i) — ritorna le posizioni legali
- # scelte per n=8 — $(n^2)^n = 64^8 = 2^{48} \sim 2.81 \cdot 10^{14}$

Commenti:

C'è un miglioramento, ma lo spazio è ancora grande...

Problema: una soluzione "1-7-...." non si distingue da "7-1-...."

Problema n Regine

Idea: la regina i non può stare in una casella "precedente" alla regina i-1 (vcurr[i-1] < vcurr[i])

Algoritmo:

- vcurr[1..n] coordinate in 1..n^2 vcurr[i] = coordinata della regina i
- Soluzione trovata se i = n
- Scelta(vcurr, n, i) ritorna le posizioni legali vcurr[i-1] < vcurr[i]
- # scelte per n=8 $(n^2)^n / n! = 64^8 / 8! \sim 6.98 \cdot 10^9$

Commenti:

Ottimo, abbiamo ridotto molto, ma si può ancora fare qualcosa

Problema n Regine

Idea: ogni riga (colonna) della scacchiera deve contenere esattamente una regina. Infatti non ne può contenere 2 o più, e se ne contenesse 0 un'altra riga (colonna) dovrebbe contenerne 2

Algoritmo:

- vcurr[1..n] valori in 1..n — vcurr[i] = colonna della regina i
- Soluzione trovata — se i = n
- Scelta(vcurr, n, i) — ritorna le colonne legali
- # scelte per n=8 — $n^n = 8^8 \sim 1.67 \cdot 10^7$

Commenti:

Quasi alla fine

Problema n Regine

Idea: anche ogni colonna deve contenere esattamente una regina. I numeri 1..n devono apparire in vcurr[0..n-1] come permutazione

Algoritmo:

- Modifichiamo l'algoritmo delle permutazioni per verificare anche le diagonali
- # scelte per n=8 $n! = 8! = 40.320 \sim 4.03 \cdot 10^4$
- # soluzioni effettivamente visitate: 15.720

Problema n Regine

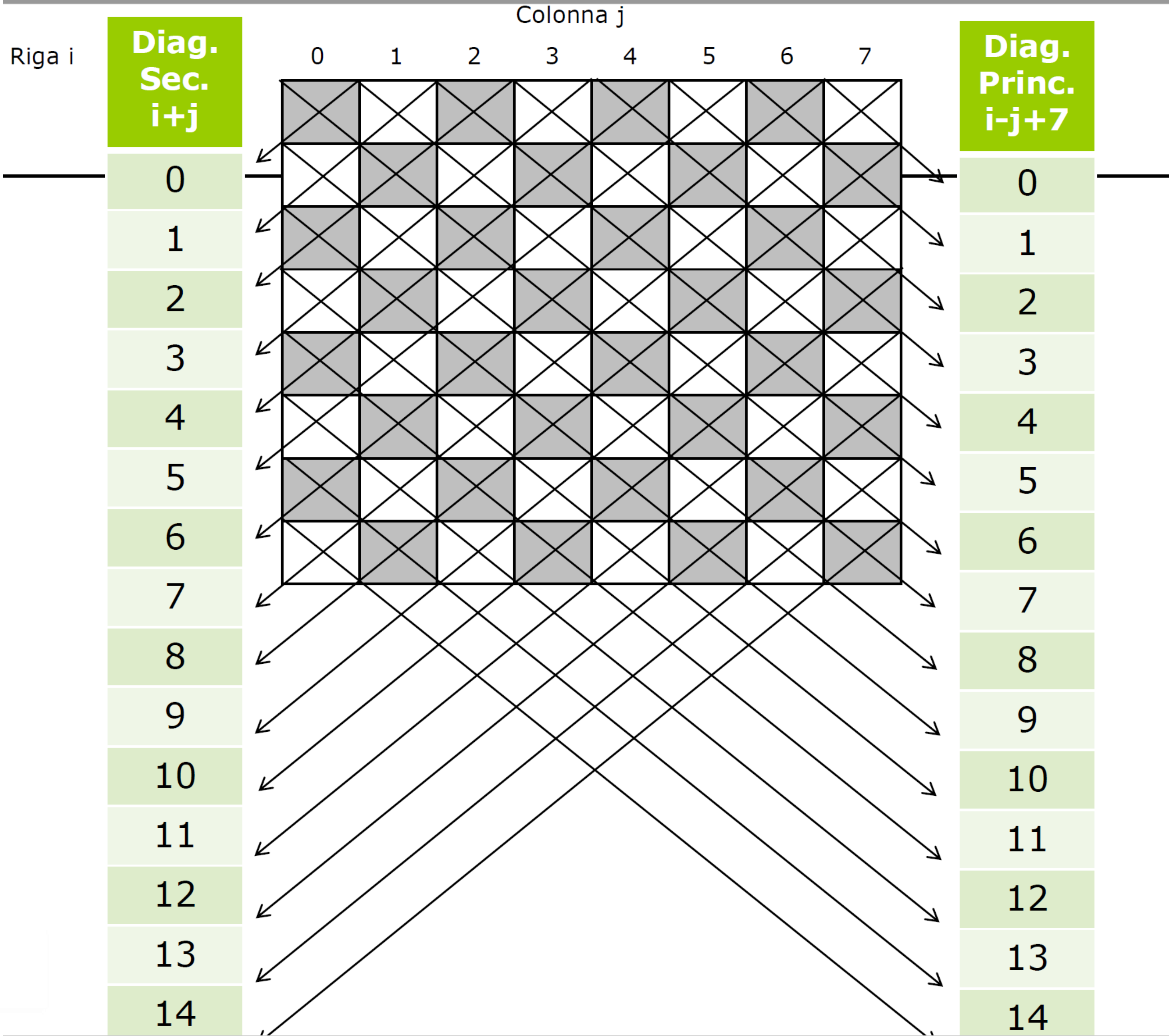
Matrice quadrata

- La differenza tra gli indici delle celle della diagonale principale è costante **i - j + (n-1)**

- La somma tra gli indici delle celle della diagonale secondaria è costante **i + j**

Verifica della scelta:

- Fissata la riga i, la cella a_{ij} è sotto scacco **sse**
  - La colonna j è occupata o
  - La diagonale principale i – j + (n-1) è occupata o
  - La diagonale secondaria i + j è occupata


Riga i
Diag. Sec. i+j
Colonna j
0 1 2 3 4 5 6 7
Diag. Princ. i-j+7
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14

Problema n Regine

```
#define DIM 8
void main(void){
    int riga[DIM];
    int diag1[2*DIM-1];
    int diag2[2*DIM-1];
    int colonna[DIM];
    int sol = 0, num = 0, chiama = 0;
    for(int i=0;i<DIM;i++) riga[i]=0;
    for(int i=0;i<2*DIM-1;i++) {
        diag1[i]=0; diag2[i]=0;
    }
    verifica(0, riga, colonna, diag1, diag2,
                                &sol, &num, &chiama);
            printf("\nNumero totale soluzioni= %d,
                         su %d tentativi, %d chiamate",
                         sol, num, chiama); }
```

Problema n Regine

```
void verifica(int j, int *riga, int *colonna, int *diag1, int
*diag2, int *nsol, int *ntent, int *ncall)
{ char invio;
    for(int i=0;i<DIM;i++){
        (*ntent)++;
        if((riga[i] == 0) &&
           (diag1[i+j]==0) && (diag2[i-j+7]==0)){
            /* regina nella casella i,j */
            colonna[j] = i;
            /* riga i sotto scacco */
            riga[i] = 1;
            /* diagonale secondaria i+j sotto scacco */
            diag1[i+j] = 1;
            /* diagonale secondaria i-j+7 sotto scacco */
            diag2[i-j+7] = 1;
        if (j<DIM-1){ (*ncall)++;
            verifica(j+1, riga, colonna, diag1,
                    diag2, nsol, ntent, ncall); }
```

Problema n Regine

```
     else { /* trovata 1 soluzione */
          (*nsol)++;
          visualizza(colonna, *nsol);
          scanf("%c", &invio); }
     /* libero le righe e le diagonali esaminate */
     riga[i] = 0;
     diag1[i+j] = 0;
     diag2[i-j+7] = 0; }}}

void visualizza(int *colonna, int nsol){
     printf("\n Soluzione numero: %d\n", nsol);
     for (int k=0; k<DIM; k++){
               for (int i=0; i<DIM; i++)
               if(i==colonna[k])
                  printf("1 ");
               else
                  printf("0 ");
          printf("\n");
     }
}
```