POLITECNICO
DI TORINO

Dipartimento
di Automatica e Informatica

Esempi di problemi di ricerca e ottimizzazione

Paolo Camurati

Controllo di lampadine

Specifiche:

- n interruttori e m lampadine
- inizialmente tutte le lampadine sono spente
- ogni interruttore comanda un sottoinsieme delle lampadine:
  - un elemento [i,j] di una matrice di interi n x m indica se vale 1 che l'interruttore i controlla la lampadina j, 0 altrimenti
- se un interruttore è premuto, tutte le lampadine da esso controllate commutano di stato

problema di ottimizzazione

Scopo:

- determinare l’insieme **minimo** di interruttori da premere per accendere tutte le lampadine

Condizione di accensione:

- una lampadina è accesa se e solo se il numero di interruttori premuti tra quelli che la controllano è dispari.

Esempio: n=4 m=5

l'interruttore 0 non controlla la lampadina 2

inter	0	1	2	3	4
0	1	1	0	0	1
1	1	0	1	0	0
2	0	1	1	1	0
3	1	0	0	1	0

l'interruttore 2 controlla la lampadina 3

Effetto degli interruttori 0 e 2 **premuti**:

inter	0	1	2	3	4
0	1	1	0	0	1
1	1	0	1	0	0
2	0	1	1	1	0
3	1	0	0	1	0

int0 controlla lamp0

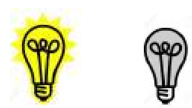
int2 non controlla lamp0

interruttori premuti
che controllano lamp0

1

Effetto degli interruttori 0 e 2 **premuti**:

inter	0	1	2	3	4
0	1	1	0	0	1
1	1	0	1	0	0
2	0	1	1	1	0
3	1	0	0	1	0



int0 controlla lamp1

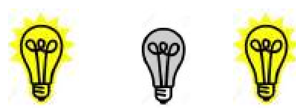
int2 controlla lamp1

interruttori premuti
che controllano lamp1

2

Effetto degli interruttori 0 e 2 **premuti**:

inter	0	1	2	3	4
0	1	1	0	0	1
1	1	0	1	0	0
2	0	1	1	1	0
3	1	0	0	1	0



int0 non controlla lamp2

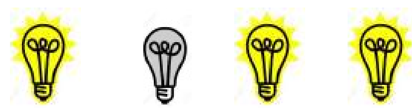
int2 controlla lamp2

interruttori premuti
che controllano lamp2

1

Effetto degli interruttori 0 e 2 **premuti**:

inter	0	1	2	3	4
0	1	1	0	0	1
1	1	0	1	0	0
2	0	1	1	1	0
3	1	0	0	1	0



int0 non controlla lamp3

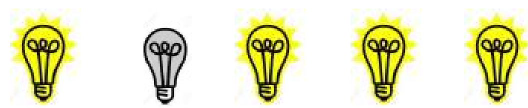
int2 controlla lamp3

interruttori premuti
che controllano lamp3

1

Effetto degli interruttori 0 e 2 **premuti**:

inter	0	1	2	3	4
0	1	1	0	0	1
1	1	0	1	0	0
2	0	1	1	1	0
3	1	0	0	1	0



int0 controlla lamp4

int2 non controlla lamp4

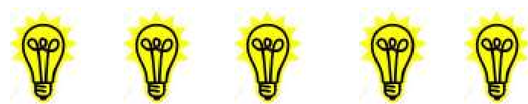
interruttori premuti
che controllano lamp4

1

SOLUZIONE NON VALIDA

Effetto degli interruttori 0, 1 e 3 **premuti**:

inter	0	1	2	3	4
0	1	1	0	0	1
1	1	0	1	0	0
2	0	1	1	1	0
3	1	0	0	1	0



Controllo:
lamp0: 3 interruttori
lamp1: 1 interruttore
lamp2: 1 interruttore
lamp3: 1 interruttore
lamp4: 1 interruttore

SOLUZIONE VALIDA

Algoritmo:

- generare tutti i sottoinsiemi di interruttori (non necessario l’insieme vuoto)
- per ogni sottoinsieme applicare una funzione di verifica di validità
- tra le soluzioni valide, scegliere la prima tra quelle a minima cardinalità.

Modello:

- insieme delle parti generato con combinazioni semplici di n elementi a k a k
- k cresce da 1 a n (non necessario l'insieme vuoto)
- la prima soluzione che si trova è anche quella a cardinalità minima.

Strutture dati:

- matrice inter di interi n x m
- vettore sol di n interi
- non serve il vettore val (gli interruttori sono numerati da 0 a n-1)

```c
int main(void) {
  int n, m, k, i, trovato=0;
  FILE *in = fopen("switches.txt", "r");
  int **inter = leggiFile(in, &n, &m);
  int *sol = calloc(n, sizeof(int));

  printf("Powerset mediante combinazioni semplici\n\n");
  for (k=1; k <= n && trovato==0; k++) {
    if(powerset(0, sol, n, k, 0, inter, m))
      trovato = 1;
  }
  free(sol);
  for (i=0; i < n; i++)
    free(inter[i]);
  free(inter);
  return 0;
}
```

cardinalità sottoinsieme crescente da 1 a n

non serve l’insieme vuoto

stop appena trovata soluzione a cardinalità minima

01interruttori_comb_sempl

```c
int powerset(int pos,int *sol,int n,int k,int start,int **inter,int m) {
  int i;
  if (pos >= k) {
    if (verifica(inter, m, k, sol)) {
      stampa(k, sol);
      return 1;
    }
    return 0;
  }
  for (i = start; i < n; i++) {
    sol[pos] = i;
    if (powerset(pos+1, sol, n, k, i+1, inter, m))
      return 1;
  }
  return 0;
}
```

verifica di validità

stop appena trovata soluzione valida

soluzione non valida

nessuna soluzione valida trovata

Verifica:

- dato un sottoinsieme di k interruttori premuti
  - per ogni lampadina contare quanti interruttori la controllano
  - registrare se pari o dispari (calcolando il resto della divisione intera per 2)
- soluzione valida se per ogni lampadina il numero di interruttori premuti che la controlla è dispari.

```c
int verifica(int **inter, int m, int k, int *sol) {
  int i, j, ok = 1, *lampadine;
  lampadine = calloc(m, sizeof(int));

  for (j=0; j<m && ok; j++)
    for(i=0; i<k; i++)
      lampadine[j] += inter[sol[i]][j];
    if (lampadine[j]%2 == 0)
      ok = 0;
  }
  free(lampadine);
  return ok;
}
```

∀ lampadina

∀ interruttore del sottoinsieme

conta quanti interruttori del sottoinsieme la controllano

se pari KO

Verifica alternativa:

- vettore delle lampadine (inizialmente tutte spente)
- per ciascuna delle lampadine
  - per ciascuno degli interruttori del sottoinsieme

lampadina \ interruttore	non controlla	controlla
spenta	spenta	accesa
accesa	accesa	spenta

stato della lampadina

lampadina \ interruttore	0	1
0	0	1
1	1	0

lampadina EXOR interruttore

```c
int verifica(int **inter, int m, int k, int *sol) {
  int i, j, ok = 1, *lampadine;
  lampadine = calloc(m, sizeof(int));

  for (j=0; j<m && ok; j++) {
    for (i=0; i<k; i++)
      lampadine[j] ^= inter[sol[i]][j];
    if (lampadine[j]==0
      ok = 0;
  }

  free(lampadine);
  return ok;
}
```

∀ lampadina

∀ interruttore del sottoinsieme

lampadina EXOR interruttore

se pari KO

Longest Increasing Sequence

Data una sequenza di N interi

$$X = (x_0, x_1, \dots x_{N-1})$$

si definisce **sottosequenza** di X di lunghezza k ($k \leq N$) una qualsiasi n-upla Y di k elementi di X con indici crescenti $i_0, i_1, \cdots, i_{k-1}$ non necessariamente contigui.

Esempio:
X=0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15
Y = 0, 8, 12, 10, 14, 1, 7, 15 è una sottosequenza di lunghezza k=8 di X (indici 0, 1, 3, 5, 7, 8, 14, 15).

Si ricordi:

- sottosequenza: indici non necessariamente contigui
- sottostringa/sottovettore: indici contigui

Problema:

data una sequenza, identificare una qualsiasi delle sue LIS (Longest Increasing Subsequence), cioè una delle sottosequenze:

- strettamente crescenti && a lunghezza massima.

Esempio:

per X=0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

esistono 4 LIS con k=6:

0, 2, 6, 9, 11, 15   0, 4, 6, 9, 11, 15

0, 2, 6, 9, 13, 15   0, 4, 6, 9, 13, 15

Algoritmo:

- generare tutti i sottoinsiemi di elementi di X (non necessario l'insieme vuoto)
- per ogni sottoinsieme applicare una funzione di verifica di validità (controllo di monotonia stretta)
- problema di ottimizzazione: tenere traccia della soluzione ottima corrente e confrontarla con ciascuna soluzione generata
- tra le soluzioni valide, scegliere una tra quelle a massima cardinalità.

Modello:

- insieme delle parti generato con disposizioni ripetute di n elementi a k a k
- k cresce da 1 a n

Strutture dati:

- vettore v di n interi per i valori
- vettore s e bs di n interi per soluzione corrente e soluzione migliore
- interi l e bl per lunghezza corrente e lunghezza migliore.

```
void ps(int pos, int *v, int *s, int k, int *bl, int *bs) {
  int j, l=0, ris;
  if (pos >= k) {
    for (j=0; j<k; j++)
      if (s[j]!=0) l++;
    ris = check(val, k, s, l);
    if (ris==1) {
      if (l >= *bl) {
        for (j=0; j<k; j++) bs[j] = s[j];
        *bl = l;
      }
    }
    return;
  }
  s[pos] = 0; ps(pos+1, v, s, k, bl, bs);
  s[pos] = 1; ps(pos+1, v, s, k, bl, bs);
  return;
}
```

caso terminale

verifica di validità

verifica di ottimalità

02LIS

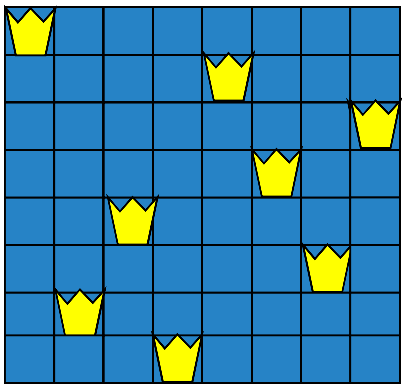
```
int check(int *v, int k, int *s int l) {
  int i=0,j, t, ok=1;

  for (t=0; t<l-1; t++){
    while ((s[i]==0) && (i < k-1))
      i++;
    j=i+1;
    while ((s[j]==0) && (j < k))
      j++;
    if (v[i] >= v[j])
      ok = 0;
    i = j;
   }
   return ok;
 }
```

Le 8 regine (Max Bezzel 1848)

Data una scacchiera 8 x 8, disporvi 8 regine in modo che non si diano scacco reciprocamente:

- 92 soluzioni
- 12 fondamentali (tenendo conto di rotazioni e simmetrie)

Esempio:



Generalizzabile a N regine, con $N \geq 4$:

- N=4: 2 soluzioni
- N=5: 10 soluzioni
- N=6: 4 soluzioni
- etc.

Problema di ricerca per cui si vuole:

- 1 soluzione qualsiasi
- tutte le soluzioni

NB: le regine sono di per sè indistinte. I modelli che le considerano distinte generano soluzioni identiche a meno di permutazioni, rotazioni e simmetrie.

Modello 0:

- ogni cella può contenere o no una regina indistinta (il numero di regine varia da 0 a 64)
- **powerset con disposizioni ripetute**
- pruning opportuno
- filtro le soluzioni imponendo di avere esattamente 8 regine
- $D'_{n,k} = 2^{64} \approx 1.84\ 10^{19}$ casi (senza pruning)!
- variabili globali s[N][N], num_sol
- variabile q che svolge il ruolo della variabile pos.

```c
void powerset (int r, int c, int q) {
  if (c>=N) {
    c=0; r++;
  }
  if (r>=N) {
    if (q!=N)
      return;
    if (controlla())
      stampa();
    return;
  }
  s[r][c] = q+1;
  powerset (r,c+1,q+1);
  s[r][c] = 0;
  powerset (r,c+1,q);
  return;
}
```

colonna finita, passa alla prossima riga

scacchiera finita!

prova a mettere la regina su r,c

ricorri

backtrack

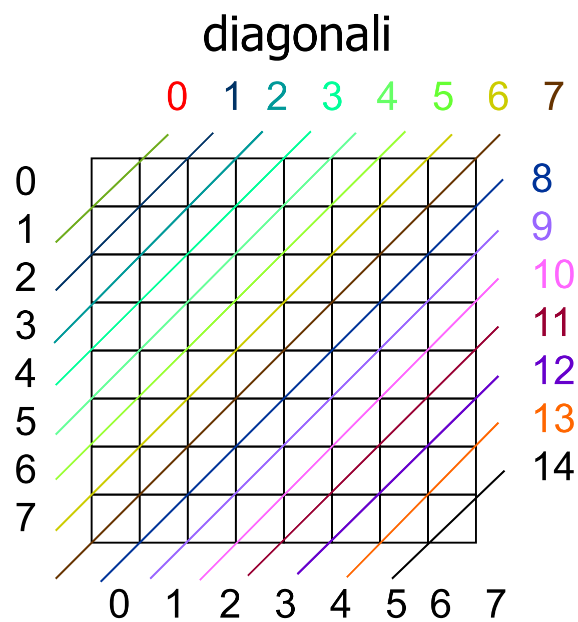
ricorri senza la regina su r,c

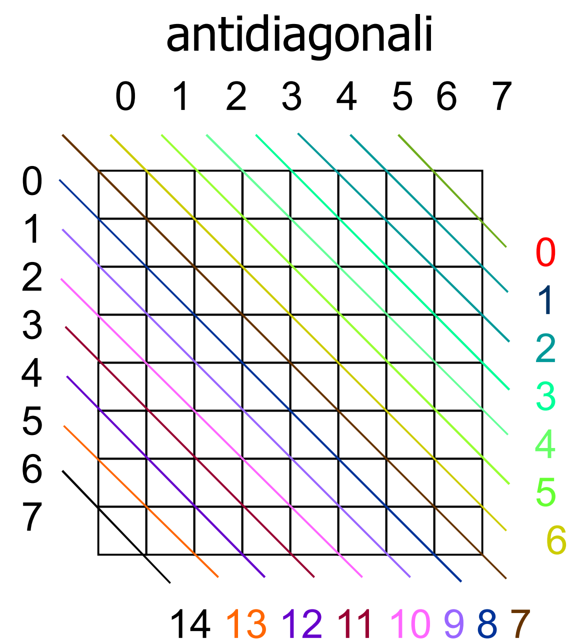
```c
#define N 4
int s[N][N];
int num_sol=0;
```

03otto-regine-powerset

Funzione `controlla`:

- righe , colonne, diagonali e antidiagonali: conteggiare per ognuna il numero di celle della scacchiera diverse da 0. Se tale numero è >1, la soluzione è inaccettabile
- diagonali:
  - 15 diagonali individuate dalla somma degli indici di riga e di colonna
  - 15 antidiagonali individuate dalla differenza degli indici di riga e di colonna (+ 7 per non avere valori negativi)


diagonali
0 1 2 3 4 5 6 7
8 9 10 11 12 13 14
0 1 2 3 4 5 6 7


antidiagonali
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6
14 13 12 11 10 9 8 7

```c
int controlla (void) {
  int r, c, n;
  for (r=0; r<N; r++) {
    for (c=n=0; c<N; c++)
      if (s[r][c]!=0)
        n++;
    if (n>1)
      return 0;
  }
  for (c=0; c<N; c++) {
    for (r=n=0; r<N; r++)
      if (s[r][c]!=0)
        n++;
    if (n>1)
      return 0;
  }
  ...........
```

controlla righe

controlla colonne

controlla diagonali

```
for (d=0; d<2*N-1; d++) {
  n=0;
  for (r=0; r<N; r++) {
    c = d-r;
    if ((c>=0)&& (c<N))
      if (s[r][c]!=0) n++;
  }
  if (n>1) return 0;
}
```

controlla antidiagonali

```
for (d=0; d<2*N-1; d++) {
    n=0;
    for (r=0; r<N; r++) {
      c = r-d+N-1;
      if ((c>=0)&& (c<N))
        if (s[r][c]!=0) n++;
    }
    if (n>1) return 0;
  }
  return 1;
}
```

l'ordinamento conta

Modello 1:

- piazzo 8 distinte regine (k = 8) in 64 caselle (n = 64)
- **disposizioni semplici**
- $D_{n,k} = \frac{n!}{(n-k)!} \approx$ 1,78 10^{14} casi!
- variabili globali `num_sol` e `s[N][N]` che svolge il ruolo del vettore `mark`
- variabile `q` che svolge il ruolo della variabile `pos`.

```
void disp_sempl(int q) {
  int r,c;
  if (q >= N) {
    if(controlla()) {
      num_sol++;
      stampa();
    }
    return;
  }
  for (r=0; r<N; r++)
    for (c=0; c<N; c++)
      if (s[r][c] == 0) {
        s[r][c] = q+1;
        disp_sempl(q+1);
        s[r][c] = 0;
      }
  return;
}
```

piazzate tutte le regine

controllo se cella vuota

prova a mettere la regina su r,c

ricorri

backtrack

```
#define N 4
int s[N][N];
int num_sol=0;
```

04otto-regine-disp-sempl

Modello 2:

- piazzo 8 indistinte regine (k = 8) in 64 caselle (n = 64)
- **combinazioni semplici** l'ordinamento non conta
- $C_{n,k} = \frac{n!}{k!(n-k)!} \approx$ 4,42 10^9 casi!

- variabile globale `s[N][N]` per la scacchiera
- variabile `q` che svolge il ruolo della variabile `pos`
- variabili `r0` e `c0` per forzare un ordinamento.

```
void comb_sempl(int r0, int c0, int q) {
  int r,c;
  if (q >= N) {
    if(controlla()) {
      num_sol++; stampa();
    }
    return;
  }
  for (r=r0; r<N; r++)
    for (c=0; c<N; c++)
      if (((r>r0)||((r==r0)&&(c>=c0)))&&s[r][c]==0) {
        s[r][c] = q+1;
        comb_sempl (r,c,q+1);
        s[r][c] = 0;
      }
  return;
}
```

piazzate tutte le regine

iterazione sulle scelte

controllo sulla fattibilità della scelta

scelta

ricorri

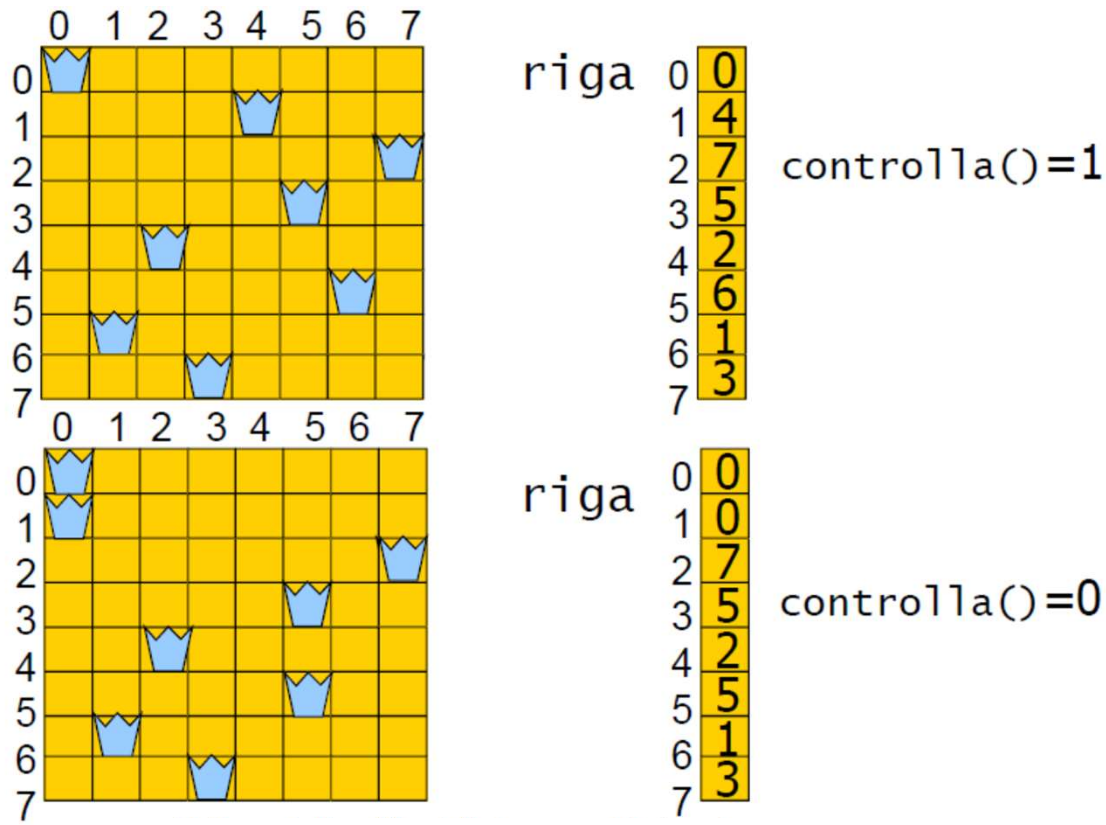
backtrack

05otto-regine-comb-sempl

Modello 3:

- struttura dati monodimensionale:
  - ogni riga contiene una e una sola regina distinta in una delle 8 colonne (n = 8)
- ci sono 8 righe (k = 8)
- **disposizioni con ripetizione**
- $D'_{n,k} = n^k = 8^8 = 16.777.216$ casi!
- non serve più il controllo sulle righe, basta quello su colonne, diagonali e antidiagonali
- variabile `riga[N]`
- variabile `q` che svolge il ruolo della variabile `pos`.

l'ordinamento conta


0 1 2 3 4 5 6 7
riga
0 4 7 5 2 6 1 3
controlla()=1
0 1 2 3 4 5 6 7
riga
0 0 7 5 2 5 1 3
controlla()=0

```
void disp_ripet(int q) {
  int i;
  if (q >= N) {
    if(controlla()) {
      num_sol++;
      stampa();
    }
    return;
  }
  for (i=0; i<N; i++) {
    riga[q] = i;
    disp_ripet(q+1);
  }
  return;
}
```

scacchiera finita!

prova a mettere la regina sulla riga

ricorri

06otto-regine-disp-ripet

```c
int controlla (void) {
  int r, n, d, occ[N];

  for (r=0; r<N; r++) occ[r]=0;

  for (r=0; r<N; r++)
    occ[riga[r]]++;
  for (r=0; r<N; r++)
    if (occ[r]>1)
      return 0;

  for (d=0; d<2*N-1; d++) {
    n=0;
    for (r=0; r<N; r++) {
      if (d==r+riga[r]) n++;
    }
    if (n>1) return 0;
  }
```

vettore delle occorrenze

controlla colonne

controlla diagonali

```
for (d=0; d<2*N-1; d++) {
    n=0;
    for (r=0; r<N; r++) {
      if (d==(r-riga[r]+N-1))
        n++;
    }
    if (n>1) return 0;
  }
  return 1;
}
```

controlla antidiagonali

Modello 4:

- ogni riga e ogni colonna contengono una e una sola regina distinta in una delle 8 colonne (n = 8)
- ci sono 8 righe (k = 8)
- **permutazioni semplici** l'ordinamento conta
- $P_n = D_{n,n}$= n! = 40320 casi possibili!

- variabili globali `riga[N]` e `mark[N]`
- variabile `q` che svolge il ruolo della variabile `pos`
- controllo solo su diagonali e antidiagonali.

```
void perm_sempl(int q) {
  int c;
  if (q >= N) {
    if (controlla()) {
      num_sol++; stampa();
      return;
    }
    return;
  }
  for (c=0; c<N; c++)
    if (mark[c] == 0) {
      mark[c] = 1; riga[q] = c;
      perm_sempl(q+1); mark[c] = 0;
    }
  return;
}
```

scacchiera finita!

prova a mettere la regina sulla riga

backtrack

ricorri

07otto-regine-perm-sempl

```c
int controlla (void) {
  int r, n, d;
  for (d=0; d<2*N-1; d++) {
    n=0;
    for (r=0; r<N; r++)
      if (d==r+riga[r])
        n++;
    if (n>1) return 0;
  }
  for (d=0; d<2*N-1; d++) {
    n=0;
    for (r=0; r<N; r++)
      if (d==(r-riga[r]+N-1))
        n++;
    if (n>1) return 0;
  }
  return 1;
}
```

controlla diagonali

controlla antidiagonali

trade-off tempo/spazio

Modello 4 ottimizzato:

- uso di 2 vettori `d[2*N-1]` e `ad[2*N-1]` per marcare le diagonali e le antidiagonali messe sotto scacco da una regina
- pruning: controllo di ammissibilità prima di procedere ricorsivamente.

```
void perm_sempl(int q) {
  int c;
  if (q >= N) {num_sol++; stampa(); return;}
  for (c=0; c<N; c++)
    if ((mark[c]==0)&&(d[q+c]==0)&&(ad[q-c+(N-1)]==0)){
      mark[c] = 1;
      d[q+c] = 1;
      ad[q-c+(N-1)] = 1;
      riga[q] = c;
      perm_sempl(q+1);
      mark[c] = 0;
      d[q+c] = 0;
      ad[q-c+(N-1)] = 0;
    }
  return;
}
```

scacchiera finita!

controllo

prova a mettere la regina sulla riga

ricorri

backtrack

08otto-regine-perm-sempl-ott

Aritmetica verbale

Specifiche:

input: 3 stringhe, 1 operazione (addizione)

Esempio:

```
   S  E N D +
   M O R E =
 ____________
 M O N E Y
```

Interpretazione:

le stringhe sono interi "criptati", cioè ogni lettera rappresenta una e una sola cifra decimale.

Output: decriptare le stringhe, cioè identificare la corrispondenza lettere – cifre decimali che soddisfa l'addizione data.

Assunzioni:

- solo caratteri alfabetici tutti maiuscoli o tutti minuscoli
- la cifra più significativa diversa da 0
- le prime due stringhe di lunghezza massima 8, non necessariamente uguale
- la terza stringa ha lunghezza coerente con quella delle prime 2
- si opera in base 10: nelle stringhe non compaiono più di 10 lettere distinte (`lett_dist <=10`)

Soluzione:
O=0, M=1, Y=2, E=5, N=6, D=7, R=8 e S=9

```
  S E N D +          9 5 6 7 +
  M O R E =          1 0 8 5 =
-------------       -------------
M O N E Y           1 0 6 5 2
```

A ogni lettera distinta va associata una e una sola cifra decimale 0..9
Modello: disposizioni semplici di n elementi a k a k, dove n = 10 e k = `lett_dist`

Strutture dati

tabella di simboli

- Variabile globale intera `lett_dist`
- Vettore lettere[10] di `struct` di tipo `alpha` con campo `car` (carattere distinto) e `val` (cifra decimale corrispondente)
- Vettore `mark[10]` per marcare le cifre già considerate

Algoritmo

- allocare e inizializzare il vettore `lettere` (funzione `init_alpha`)
- leggere le 3 stringhe
- riempire `lettere` con `lett_dist` caratteri distinti (funzione `setup` che usa la funzione di servizio `trova_indice`)
- calcolare le disposizioni semplici delle n=10 cifre decimali a k a k, dove k=`lett_dist` (funzione `disp`):
  - nella condizione di terminazione sostituire le lettere con le cifre, convertire ad intero (funzione `w2n`) , controllare la validità della soluzione (funzione `c_sol`) e, se valida, stamparla (funzione `stampa`)
  - ricorsione sulla posizione successiva nel vettore `lettere`.

Funzioni

accesso alla tabella di simboli

`int trova_indice(alpha *lettere, char c)`

dato il carattere `c`, trova e ritorna il suo indice nel vettore `lettere`, se non c'è ritorna -1

`alpha * init_alpha()`

alloca `lettere[10]` e inizializzalo (valore = -1, carattere = \0)

`void setup(alpha *lettere, char *str1, char *str2, char *str3)`

date le 3 stringhe, metti i caratteri distinti in `lettere` e conta quanti sono (`lett_dist`)

```
int disp(alpha *lettere, int *mark, int
pos, char *str1, char *str2, char *str3)
```

calcola le disposizioni delle n=10 cifre decimali a k a k, dove k=`lett_dist`

```
int w2n(alpha *lettere, char *str)
```

rimpiazza nella stringa `str` le lettere con le cifre, sulla base della corrispondenza memorizzata in `lettere`, converti a intero, ritornando -1 nei casi in cui la cifra più significativa della stringa è 0

```
int c_sol(alpha *lettere, char *str1,
char *str2, char *str3)
```

controlla che le 3 stringhe convertite a intero soddisfino la somma

```
void stampa(alpha *lettere)
```

stampa la corrispondenza lettere-cifre memorizzata nei campi `car` e `val` di `lettere`

SEND MORE MONEY

`lettere`

`car`	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
`val`	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

dopo `init_alpha`

`lettere`

`car`	S	E	N	D	M	O	R	Y	\0	\0
`val`	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

dopo `setup`

`lett_dist = 8`

`lettere`

`car`	S	E	N	D	M	O	R	Y	\0	\0
`val`	9	5	6	7	1	0	8	2	-1	-1

dopo `disp`:esempio di possibile disposizione

```c
int main(void) {
  char str1[LUN_MAX], str2[LUN_MAX], str3[LUN_MAX+1];
  int mark[base] = {0};
  int i;

  // Lettura delle 3 stringhe

  alpha *lettere = init_alpha();
  setup(lettere, str1, str2, str3);

  disp(lettere, mark, 0, str1, str2, str3);

  free(lettere);
  return 0;
}
```

```c
#define LUN_MAX 8+1
#define n 10
#define base 10
int lett_dist = 0;
```

variabile globale

09aritmetica_verbale

```c
typedef struct { char car; int  val; } alpha;

int trova_indice(alpha *lettere, char c) {
  int i;
  for(i=0; i < lett_dist; i++)
    if (lettere[i].car == c) return i;
  return -1;
}
alpha *init_alpha() {
  int i; alpha *lettere;
  lettere = malloc(n * sizeof(alpha));
  if (lettere == NULL) exit(-1);
  for(i=0; i < n; i++) {
    lettere[i].val = -1; lettere[i].car = '\0';
  }
  return lettere;
}
```

```
void setup(alpha *lettere,char *st1,char *st2,char *st3){
  int i, l1=strlen(st1), l2= strlen(st2), l3=strlen(st3);

  for(i=0; i<l1; i++) {
    if (trova_indice(lettere, st1[i]) == -1)
        lettere[lett_dist++].car = st1[i];
  }
  for(i=0; i<l2; i++) {
    if (trova_indice(lettere, st2[i]) == -1)
        lettere[lett_dist++].car = st2[i];
  }
  for(i=0; i<l3; i++) {
    if (trova_indice(lettere, st3[i]) == -1)
        lettere[lett_dist++].car = st3[i];
  }
}
```

```c
int w2n(alpha *lettere, char *st) {
  int i, v = 0, l=strlen(st);
  if (lettere[trova_indice(lettere, st[0])].val == 0)
    return -1;
  for(i=0; i < l; i++)
    v = v*10 + lettere[trova_indice(lettere, st[i])].val;
  return v;
}
int c_sol(alpha *lettere,char *st1,char *st2,char *st3) {
  int n1, n2, n3;
  n1 = w2n(lettere, st1);
  n2 = w2n(lettere, st2);
  n3 = w2n(lettere, st3);
  if (n1 == -1 || n2 == -1 || n3 == -1)
    return 0;
  return ((n1 + n2) == n3);
}
```

```
int disp(alpha *lettere, int *mark, int pos, char *st1,
         char *st2, char *st3) {
  int i = 0, risolto;
  if (pos == lett_dist) {
    risolto = contr_sol(lettere, st1, st2, st3);
    if (risolto) stampa(lettere);
    return risolto;
  }
  for(i=0;i < base; i++) {
    if (mark[i]==0) {
      lettere[pos].val = i; mark[i] = 1;
      if (disp(lettere, mark, pos+1, st1, st2, st3))
        return 1;
      lettere[pos].val = -1; mark[i] = 0;
    }
  }
  return 0;
}
```

I 36 ufficiali di Eulero

Ci sono 36 ufficiali provenienti da 6 reggimenti (colori) e appartenenti a 6 ranghi diversi (pezzi degli scacchi).

Disporre gli ufficiali in un quadrato 6 x 6 in modo che in ogni riga e in ogni colonna compaia un ufficiale di ogni rango e un ufficiale di ogni reggimento.

Il Quadrato Latino

Quadrato latino di ordine n: quadrato n per n in cui le n^2 caselle sono occupate da n simboli distinti in modo che ogni simbolo compaia una e una sola volta in ogni riga e in ogni colonna del quadrato.

A	B	C	D
B	A	D	C
C	D	A	B
D	C	B	A

Il Quadrato Greco-latino

Quadrato greco-latino di ordine n su 2 insiemi S e T di n elementi: quadrato n per n in cui le n^2 caselle sono occupate da n^2 coppie ordinate di simboli distinti di S e T in modo che ogni coppia compaia una e una sola volta in ogni riga e in ogni colonna del quadrato.

Numero di coppie ordinate: principio di moltiplicazione nxn.

A	B	C	D
B	A	D	C
C	D	A	B
D	C	B	A

α	β	γ	δ
γ	δ	α	β
δ	γ	β	α
β	α	δ	γ

A;α	B;β	C;γ	D;δ
B;γ	A;δ	D;α	C;β
C;δ	D;γ	A;β	B;α
D;β	C;α	B;δ	A;γ

36 ufficiali di Eulero: esiste un quadrato greco-latino con n=6?

I quadrati greco-latini esistono per ordine n ≥ 3 con l’eccezione di n=6.

Eulero aveva ipotizzato che non esistessero per n=6 e per tutti i numeri pari che, divisi per 2, danno un numero dispari.

Falsa in generale la congettura di Eulero, ma vera per n=6.

Il Sudoku

Deriva dai quadrati latini di Eulero.

Input:

- griglia di 9×9 celle
- cella o vuota o con numero da 1 a 9
- 9 righe orizzontali, 9 colonne verticali
- da bordi doppi 9 regioni, di 3×3 celle contigue
- inizialmente da 20 a 35 celle riempite

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

Scopo del gioco è quello di riempire le caselle bianche con numeri da 1 a 9, in modo tale che in ogni riga, colonna e regione siano presenti tutte le cifre da 1 a 9 senza ripetizioni.

Una soluzione:

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

Si può generalizzare il Sudoku a griglie n x n purché n sia un quadrato perfetto (un numero intero che può essere espresso come il quadrato di un altro numero intero).

Modello:

- disposizioni con ripetizione
- k = numero di celle non preassegnate, n è il numero di scelte
- dimensione dello spazio: n^k .

Ricerca di tutte o di una soluzione.

Ricorsione di 2 tipi:

- casella già piena: non c'è scelta
- casella vuota: c'è scelta. In fase di ritorno si annulla la scelta (altrimenti si ricadrebbe nel caso precedente)
- pruning: controllo prima di ricorrere.

```c
int main() {
char nomefile[20];
  printf("Inserire il nome del file: ");   scanf("%s", nomefile);
  acquisisci(nomefile);
  disp_ripet(0);
  printf("\n Numero di soluzioni = %d\n", num_sol);
return 0;
}
void acquisisci(char *nomefile) {
  int i, j;  FILE *fp;
  fp = fopen(nomefile, "r");   /* inserire controllo di errore */
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) fscanf(fp, "%d", &schema[i][j]);
  }
  fclose(fp);
  return;
}
```

variabili globali

```c
#define n 9
int schema[n][n], num_sol=0;
```

```c
void disp_ripet(int pos) {
  int i, j, k;
  if (pos >= n*n) {
    num_sol++; stampa(schema); return;
  }
  i = pos / n; j = pos % n;
  if (schema[i][j] != 0) {
    disp_ripet(pos+1);
    return;
  }
  for (k=1; k<=n; k++) {
    schema[i][j] = k;
    if (controlla(pos, k))
      disp_ripet(pos+1);
    schema[i][j] = 0;
  }
  return;
}
```

terminazione

indici casella corrente

cella già piena

ricorri su cella successiva

scelta

controlla

ricorri su cella successiva

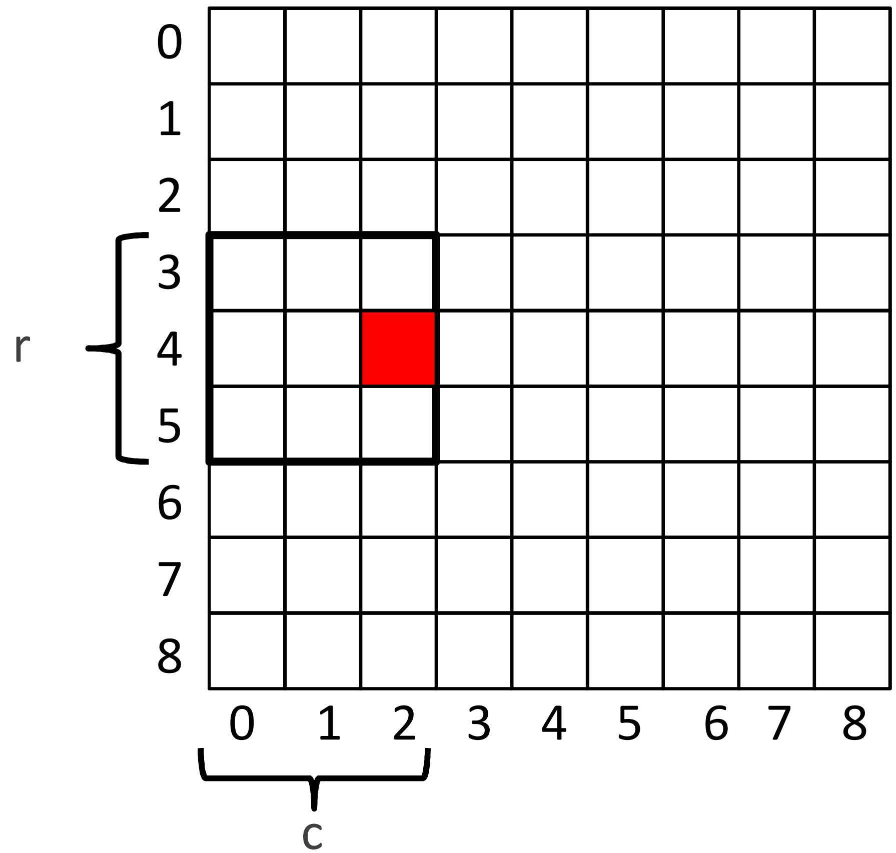
smarca la cella, altrimenti si considera fissata a priori

```c
int controlla(int pos, int val){
  int i, j, r, c, dim=floor(sqrt(n));

  i = pos/n;
  j = pos % n;

  for (c=0; c<n; c++) {
    if (c!=j)
      if (schema[i][c]==val)
        return 0;
  }
  for (r=0; r<n; r++) {
    if (r!=i)
      if (schema[r][j]==val)
        return 0;
  }
```

indici casella corrente

data la riga i, ciclo sulle colonne: controllo che il valore val inserito in i, j non sia già presente, ad esclusione della colonna j

data la colonna j, ciclo sulle righe: controllo che il valore val inserito in i, j non sia già presente, ad esclusione della riga i



Identificazione del blocco:

dim= 3

n = 9

i = 4

j = 2

$(i/dim)*dim \le r < (i/dim)*dim+dim$

$3 \le r < 6$

$(j/dim)*dim \le c < (j/dim)*dim+dim$

$0 \le c < 3$

```
  for (r=(i/dim)*dim; r<(i/dim)*dim+dim; r++)
    for (c=(j/dim)*dim; c<(j/dim)*dim+dim; c++) {
      if ((r!=i) || (c!=j))
        if (schema[r][c]==val)
          return 0;
  }
  return 1;
}
```

ciclo sui blocchi: controllo che il valore val inserito in i, j non sia presente nel blocco ad esclusione della cella i,j

Ricerca di una sola soluzione

```
int disp_ripet(int pos) {
  int i, j, k;
  if (pos >= n*n) {stampa(schema); return 1;}
  i = pos / n;
  j = pos % n;
  if (schema[i][j] != 0)
    return (disp_ripet(pos+1));

  for (k=1; k<=n; k++) {
    schema[i][j] = k;
    if (controlla(pos, k))
      if (disp_ripet(pos+1)==1)
        return 1;
    schema[i][j] = 0;
  }
  return 0;
}
```

terminazione

cella già piena

ricorri su cella successiva

scelta

controlla

ricorri su cella successiva

successo

smarca

fallimento

11sudoku1soluz

Scacchiere e grafi

Una scacchiera NxN può essere interpretata come *grafo implicito* dove:

- i vertici sono le caselle
- gli archi rappresentano la raggiungibilità di coppie di vertici. La definizione di raggiungibilità dipende dal problema in esame.

Per questa tipologia di grafo non è necessaria una rappresentazione esplicita di vertici ed archi, in quanto li si può ricavare direttamente dalla scacchiera.

In generale i problemi si riconducono alla ricerca di cammini per la quale non sono necessarie conoscenze di Teoria dei Grafi.

Il Tour del cavallo

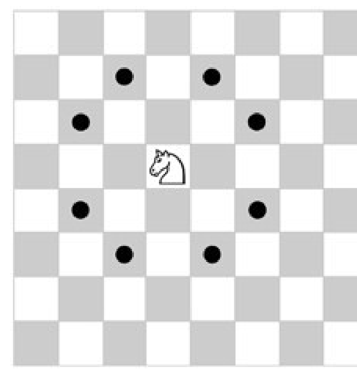
Si consideri una scacchiera NxN , trovare un *“giro di cavallo”,* cioè una sequenza di mosse valide del cavallo tale che ogni casella venga visitata una sola volta (visitata = casella su cui il cavallo si ferma, non casella attraverso cui transita).

Se il cavallo si ferma su una casella da cui si può raggiungere con una mossa la casella da cui si è partiti, il tour si dice chiuso, altrimenti si dice aperto.

Il tour del cavallo è un caso particolare di cammino di Hamilton se aperto o ciclo di Hamilton se chiuso.

Il più antico riferimento al tour del cavallo si trova in un testo poetico sanscrito del XI secolo dC. Fu studiato da Eulero nel XVIII secolo. La prima soluzione (euristica) risale al 1823 ed è la regola di von Warnsdorff.

Le mosse del cavallo lecite sono al massimo 8 a partire da ogni casella:



Interpretando la scacchiera come grafo non orientato:

- vertici = caselle
- archi: tra coppie di vertici mutuamente raggiungibili con le mosse lecite del cavallo

il problema si riconduce al Cammino di Hamilton (cammino semplice che tocca tutti i vertici).

Modello: principio di moltiplicazione: a partire da ogni cella si considerano le 8 possibili mosse. La dimensione dello spazio di ricerca è quindi $O(8^{NxN})$.

Pruning:

- si vincola la discesa ricorsiva a quelle, tra le 8 mosse possibili, che portano a caselle nella scacchiera. I vincoli sono quindi statici.

Si attribuisce a ogni cella un numero di mossa. Si termina quando sono state etichettate tutte le N x N caselle.

Esempio di tour del cavallo aperto per una scacchiera 8 x 8 a partire dalla cella (0,0):

0	59	38	33	30	17	8	63
37	34	31	60	9	62	29	16
58	1	36	39	32	27	18	7
35	48	41	26	61	10	15	28
42	57	2	49	40	23	6	19
47	50	45	54	25	20	11	14
56	43	52	3	22	13	24	5
51	46	55	44	53	4	21	12

```c
int main(void) {
  int dx[8], dy[8], **scacc, x, y, N;
  dx[0]=2; dy[0]=1; dx[1]=1; dy[1]=2;
  dx[2]=-1;dy[2]=2; dx[3]=-2;dy[3]=1;
  dx[4]=-2;dy[4]=-1;dx[5]=-1;dy[5]=-2;
  dx[6]=1; dy[6]=-2;dx[7]=2; dy[7]=-1;
  printf("Dimensione: "); scanf("%d", &N);
  scacc = malloc2d(N);
  /* inizializzazione a -1 delle celle di scacc */
  printf("Partenza: "); scanf("%d %d", &x, &y);
  scacc[x][y] = 0;
  if (mv(1, x, y, dx, dy, scacc, N)==1) {
    printf("Mosse del cavallo\n"); stampa(scacc, N);
  } else
    printf("Soluzione non trovata\n");
return 0;
}
```

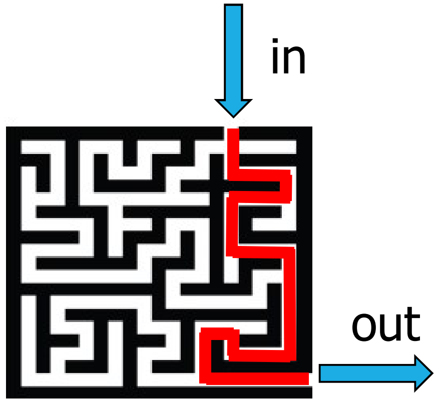
```
int mv(int m,int x,int y,int *dx,int *dy,int **s,int N){
  int i, newx, newy;
  if (m == N*N)
    return 1;
  for (i=0; i<8; i++) {
    newx = x + dx[i];
    newy = y + dy[i];
    if ((newx<N) && (newx>=0) && (newy<N)&&(newy>=0)) {
      if (s[newx][newy] == 0) {
        s[newx][newy] = m;
        if (mv(m+1, newx, newy, dx, dy, s, N) == 1)
          return 1;
        s[newx][newy] = 0;
      }
    }
  }
  return 0;
}
```

Il Labirinto

Un labirinto può essere rappresentato come una scacchiera NxM dove ogni cella o è vuota o è piena per rappresentare un muro.

Data una cella di ingresso e una di uscita, il problema consiste nel trovare, se esiste, un cammino semplice che le connette.

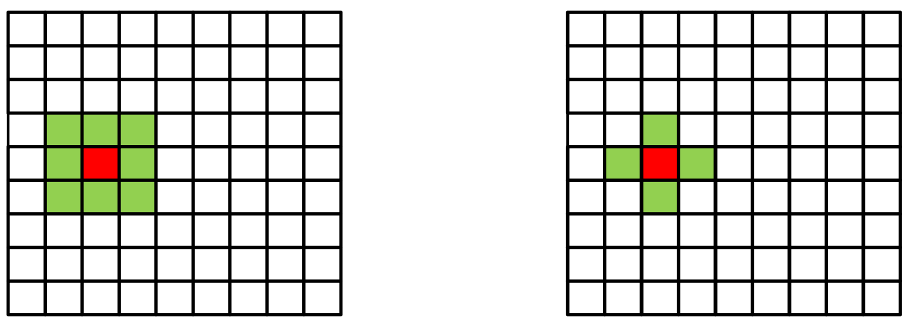
Una cella piena non può mai essere attraversata.



Relazione di raggiungibilità:

- da ogni cella si possono raggiungere al massimo le 8 celle a distanza 1 che appartengono alla scacchiera. Il movimento è secondo le 4 direzioni N, S, E, W e in obliquo
- da ogni cella si possono raggiungere al massimo le 4 celle che appartengono alla scacchiera in direzione N, S, E, W ma non in obliquo.

Nel problema in esame si considera la seconda scelta.



Interpretando la scacchiera come grafo non orientato:

- vertici = caselle
- archi: tra vertice corrente e vertici adiacenti secondo le direzioni N, S, E, W

il problema si riconduce all'enumerazione dei cammini semplici a partire dalla cella di ingresso con condizione di terminazione di aver raggiunto la cella di uscita.

Modello: principio di moltiplicazione: a partire da ogni cella si considerano le 4 possibili mosse. La dimensione dello spazio di ricerca è quindi $O(4^{N \times N})$.

Un punto viene rappresentato mediante il tipo `punto_t`, una `struct` avente come campi le coordinate di riga e colonna.

Pruning:

- si vincola la discesa ricorsiva a quelle, tra le 4 mosse possibili, che portano a caselle nella scacchiera, che non siano muri e che siano diverse dalla casella da cui si parte.

```c
int main (int argc, char *argv[]) {
  /* dichiarazioni varie,tra cui punto_t ingresso, uscita; */
  /* apertura del file e lettura di labirinto e ingresso/uscita */

  L[ingresso.r][ingresso.c] = 'I';
  L[uscita.r][uscita.c] = 'U';
  printf("configurazione iniziale\n");
  stampa();

  if (mossa(ingresso, uscita)){
    printf("soluzione trovata\n");
    stampa();
  }
  else
    printf("soluzione NON trovata\n");
  return 0;
}
```

12cavallo

```
int mossa (punto_t corrente, punto_t uscita) {
  int i;
  punto_t nuovo;
  if (corrente.r == uscita.r && corrente.c == uscita.c){
    L[corrente.r][corrente.c] = 'U';
    return 1;
  }
  for (i=0; i < 4; i++) {
    nuovo = sposta(corrente,i);
    if (nuovo.r!=corrente.r || nuovo.c!=corrente.c) {
      L[nuovo.r][nuovo.c] = '*';
      if (mossa(nuovo,uscita)==1)
        return 1;
      L[nuovo.r][nuovo.c] = '.';
    }
  }
  return 0;
}
```

backtrack

```
punto_t sposta(punto_t punto, int i) {
  int r, c;
  int spr[4] = { 0,-1, 0, 1};
  int spc[4] = {-1, 0, 1, 0};

  r = punto.r+spr[i];
  c = punto.c+spc[i];
  if (r >= 0 && c >= 0 && r < nr && c < nc)
    if ((L[r][c])=='.' || (L[r][c])=='U'){
      punto.r = r;
      punto.c = c;
    }
  return punto;
}
```