

# Esempi di problemi di ricerca e ottimizzazione



Gianpiero Cabodi Paolo Camurati  
Dip. Automatica e Informatica  
Politecnico di Torino

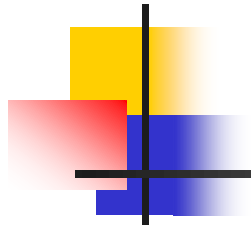


# Controllo di lampadine

---

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  $n \times m$  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



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

mat\_int





	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

A red arrow points from the text box above to the cell at row 0, column 2 (value 0). A green arrow points from the text box below to the cell at row 2, column 3 (value 1).


l'interruttore 2 controlla la lampadina 3

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

mat\_int 

 0

	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:

mat\_int 

 0

	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:

mat\_int 

 0

	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:

mat\_int 

 0

	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:

mat\_int 

 0

	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:

mat\_int 

 0 1 2 3 4

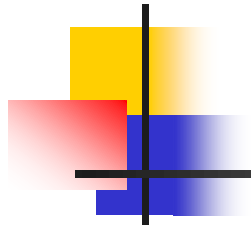
	0	1	2	3	4
0	1	1	0	0	1
1	1	0	1	0	0
2	0	1	1	1	0
3	1	0	0	1	0

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  $n \times m$
- vettori `sol` e `mark` di  $n$  interi
- non serve il vettore `val` (gli interruttori sono numerati da 0 a  $n-1$ )



```
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));  
    int *mark = calloc(n, sizeof(int));
```

cardinalità sottoinsieme  
crescente da 1 a n

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

stop appena trovata soluzione  
a cardinalità minima

```
    }  
    free(sol);  
    free(mark);  
    for (i=0; i < n; i++)  
        free(inter[i]);  
    free(inter);  
    return 0;
```

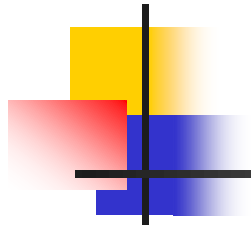
non serve l'insieme vuoto

```
}
```

verifica di validità

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

stop appena trovata  
soluzione valida



## 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.

$\forall$  interruttore del sottoinsieme

```
int verifica(int **mat_int, int n, int m, int k, int *sol) {  
    int i, j, ok = 1, *lampadine;  
    lampadine = calloc(m, sizeof(int));  
    for (i=0; i<k; i++)  
        for(j=0; j<m; j++)  
            lampadine[j] += mat_int[sol[i]][j];  
    for(i=0; i<m; i++) {  
        lampadine[i] = lampadine[i]%2;  
        ok &= lampadine[i];  
    }  
    free(lampadine);  
    return ok;  
}
```

$\forall$  lampadina

conta quanti interruttori del sottoinsieme la controllano

pari o dispari?

OK se tutti dispari





Verifica alternativa:

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

		interruttore	
		non controlla	controlla
lampadina	spenta	spenta	accesa
	accesa	accesa	spenta
		stato della lampadina	

		interruttore	
		0	1
lampadina	0	0	1
	1	1	0
		lampadina EXOR interruttore	

$\forall$  interruttore del sottoinsieme

```
int verifica(int **mat_int, int n, int m, int k, int *sol) {  
    int i, j, ok = 1; lampadine;  
    lampadine = calloc(m, sizeof(int));  
    for (i=0; i<k; i++) {  
        for (j=0; j<m; j++)  
            lampadine[j] ^= mat_int[sol[i]][j];  
    }  
    for (i=0; i<m; i++)  
        ok &= lampadine[i];  
  
    free(lampadine);  
    return ok;  
}
```

$\forall$  lampadina

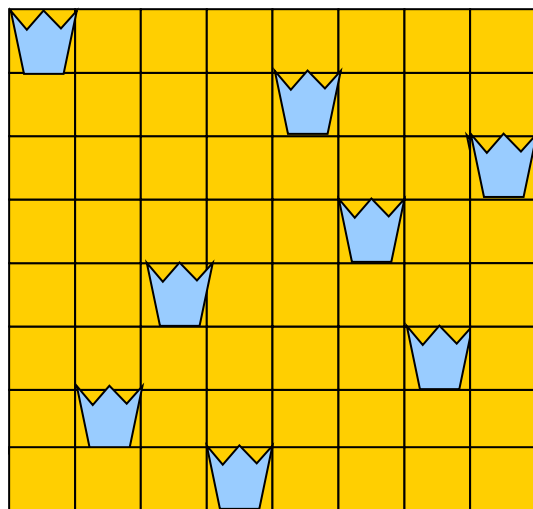
lampadina EXOR interruttore

OK se tutte accese

# 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
- ❑  $D'_{n,k} = 2^{64} \approx 1.84 \cdot 10^{19}$  casi (senza pruning)!
- ❑ variabile globale `scacchiera[N][N]`
- ❑ variabile `q` che svolge il ruolo della variabile `pos`.



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

scacchiera finita!

prova a mettere la regina su r,c

ricorri

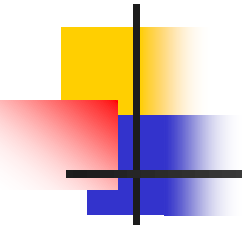
backtrack

ricorri senza la regina su r,c

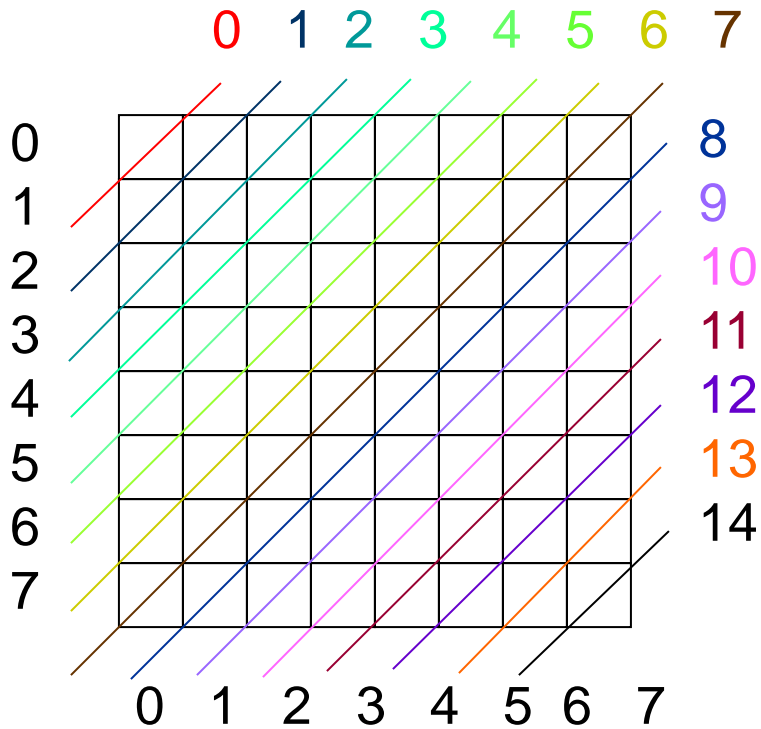


## Funzione `controlla`:

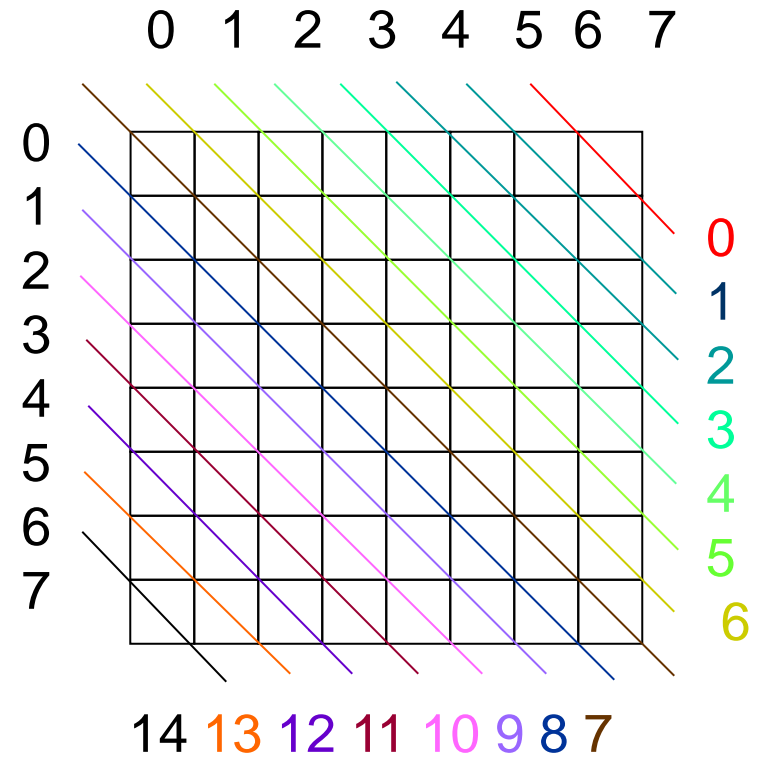
- ❑ righe , colonne, diagonali e antidiagonali:  
conteggiare per ognuna il numero di celle della scacchiera diverse da 0. Se tale numero è  $>1$ , la soluzione è inaccettabile
- ❑ diagonali:
  - 15 diagonali individuate dalla somma degli indici di riga e di colonna
  - 15 antidiagonali individuate dalla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



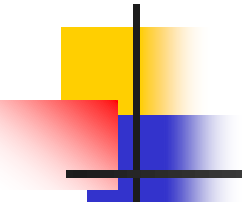
## diagonali



## antidiagonali







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

controlla antidiagonali



## Modello 1:

- piazza 8 distinte regine ( $k = 8$ ) in 64 caselle ( $n = 64$ )

- **disposizioni semplici**

l'ordinamento conta

$$D_{n,k} = \frac{n!}{(n-k)!} \approx 1,78 \cdot 10^{14} \text{ casi!}$$

- variabile globale `scacchiera[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) {
```

piazzate tutte le regine

```
        if(controlla()) {
```

```
            num_sol++;
```

```
            stampa();
```

```
        }
```

```
    } return;
```

```
}
```

```
for (r=0; r<N; r++)
```

controllo se cella vuota

```
    for (c=0; c<N; c++)
```

prova a mettere la regina su r,c

```
        if (scacchiera[r][c] == 0) {
```

```
            scacchiera[r][c] = q+1;
```

```
            disp_sempl(q+1);
```

```
            scacchiera[r][c] = 0;
```

ricorri

```
        }
```

```
    } return;
```

backtrack

```
}
```



## Modello 2:

- piazza 8 indistinte regine ( $k = 8$ ) in 64 caselle ( $n = 64$ )

- **combinazioni semplici**

$$C_{n,k} = \frac{n!}{k!(n-k)!} \approx 4,42 \cdot 10^9 \text{ casi!}$$

l'ordinamento non conta

- 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.



piazzate tutte le regine

```
void comb_semp1(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_semp1 (r,c,q+1);  
                s[r][c] = 0;  
            }  
    return;  
}
```

iterazione sulle scelte

controllo sulla fattibilità della scelta

scelta

ricorri

backtrack



## l'ordinamento conta

### 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 ( $k = 8$ )

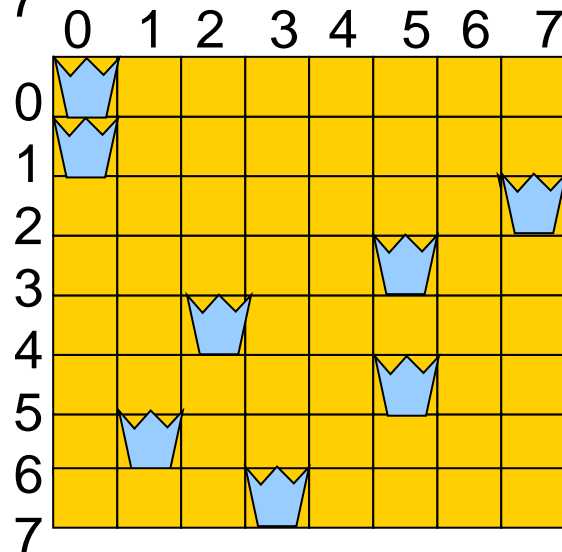
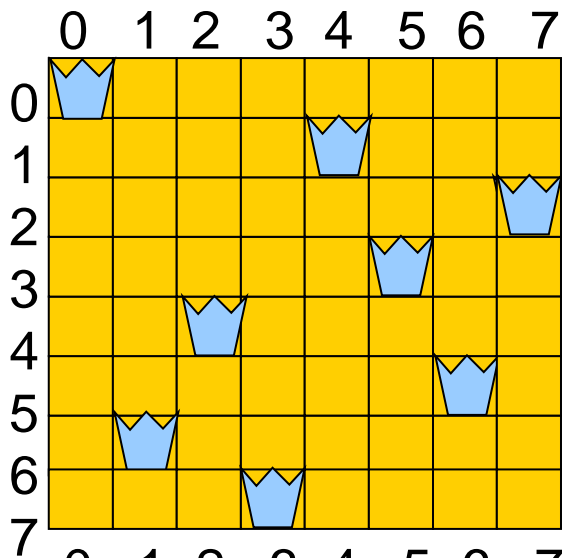
- **disposizioni con ripetizione**

$$D'_{n,k} = n^k = 8^8 = 16.777.216 \text{ casi!}$$

- 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



sol

0	0
1	4
2	7
3	5
4	2
5	6
6	1
7	3

controlla()=1

sol

0	0
1	0
2	7
3	5
4	2
5	5
6	1
7	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



vettore delle occorrenze

```
int controlla (void) {  
    int r, n, d, occ[N];  
  
    for (r=0; r<N; r++) occ[r]=0;
```

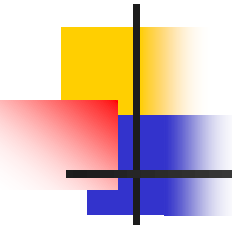
```
    for (r=0; r<N; r++)  
        occ[riga[r]]++;
```

controlla colonne

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

controlla diagonali

```
    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;  
    }  
}
```



controlla antidiagonali

```
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;  
}
```



l'ordinamento conta

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**

$$P_n = D_{n,n} = n! = 40320 \text{ casi possibili!}$$

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



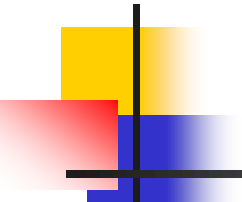
```
void perm_sempl(int q)
{
    int c;
    if (q >= N) {
        if (controlla()) {
            num_sol++; stampa();
            return;
        }
        return;
    }
    for (c=0; c<N; c++)
        if (mark[c] == 0) {
            mark[c] = 1; riga[q] = c;
            perm_sempl(q+1); mark[c] = 0;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

scacchiera finita!

prova a mettere la regina sulla riga

ricorri

backtrack



```
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+sol[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-sol[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 diagonalì e le antidiagonalì 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





# Aritmetica Verbale

---

Specifiche:

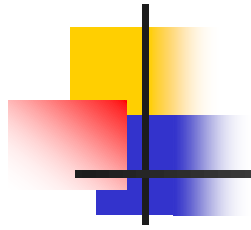
input: 3 stringhe, 1 operazione (addizione)

Esempio:

$$\begin{array}{r} S \ E \ N \ D \ + \\ M \ O \ R \ E \ = \\ \hline M \ O \ N \ E \ Y \end{array}$$

interpretazione:

le stringhe sono interi "criptati", cioè ogni lettera rappresenta 1 e 1 sola cifra decimale.



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

Considerare solo i casi in cui la lettera più significativa non corrisponde allo 0.



Soluzione:

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

S E N D +

M O R E =

---

M O N E Y

9 5 6 7 +

1 0 8 5 =

---

1 0 6 5 2

Le stringhe hanno `lett_dist` ( $\leq 10$ ) lettere distinte, a ognuna delle quali va associata 1 e 1 sola cifra decimale 0..9

Modello: disposizioni semplici di  $n$  elementi a  $k$  a  $k$ , dove  $n = 10$  e  $k = \text{lett\_dist}$



# 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

---

- ❑ leggere le 3 stringhe
- ❑ riempire `lettere` con `lett_dist` caratteri distinti
- ❑ calcolare le disposizioni delle  $n=10$  cifre decimali a  $k$  a  $k$ , dove  $k=\text{lett\_dist}$  :
  - nella condizione di terminazione sostituire le lettere con le cifre, converti ad intero, controllare la validità della soluzione e, se valida, stamparla.

# Funzioni

**int** trova\_indice(alpha \*lettere, **char** c)

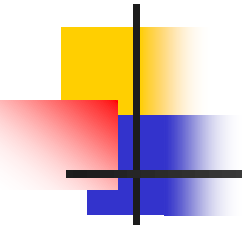
dato il carattere c, trova e ritorna il suo indice nel vettore lettere, se non c'è ritorna -1

**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  $k=\text{lett\_dist}$

```
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
val	-1	-1	-1	-1	-1	-1	-1	-1	-1

dopo `init_alpha`

lettere

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

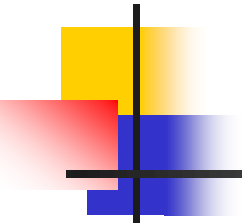
dopo `setup`

`lett_dist = 8`

lettere

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

dopo `disp`: esempio  
di possibile disposizione



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



08aritmetica\_verbale

variabile globale

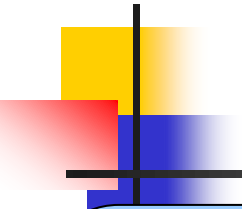
```
int main(void) {
    char str1[LUN_MAX], str2[LUN_MAX], str3[LUN_MAX+1];
    int mark[base] = {0};
    int i;

    // Lettura delle 3 stringhe

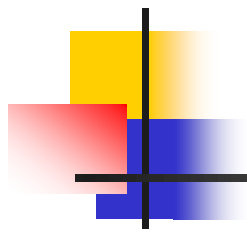
    alpha *lettere = init_alpha();
    setup(lettere, str1, str2, str3);

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

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

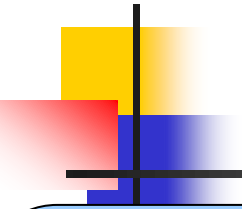


```
typedef struct {
    char car; int val;
} alpha;
int trova_indice(alpha *lettere, char c) {
    int i;
    for(i=0; i < lett_dist; i++)
        if (lettere[i].car == c) return i;
    return -1;
}
alpha *init_alpha() {
    int i; alpha *lettere;
    lettere = malloc(n * sizeof(alpha));
    if (lettere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

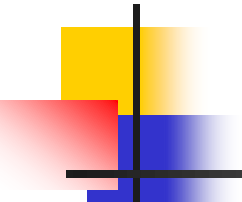


```

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;
}
```

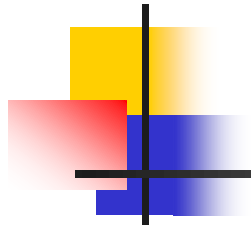


# Sudoku

Input:

- ❑ griglia di  $9 \times 9$  celle
- ❑ cella o vuota o con numero da 1 a 9
- ❑ 9 righe orizzontali, 9 colonne verticali
- ❑ da bordi doppi 9 regioni, di  $3 \times 3$  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





---

Modello:

- ❑ disposizioni con ripetizione
- ❑  $n$  = numero di celle non preassegnate,  $k = 9$
- ❑ dimensione dello spazio:  $n^9$

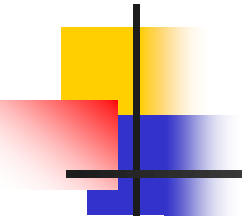
Ricerca di tutte le soluzioni.



---

## 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.



```
#define MAXBUFFER 128
int num_sol=0;
```

variabile globale



09sudoku

```
int main() {
    int **schema, dim, i, j, risultato;
    char nomefile[20];

    printf("Inserire il nome del file: ");
    scanf("%s", nomefile);
    schema = acquisisci(nomefile, &dim);

    disp_ripet(schema, dim, 0);

    printf("\n Numero di soluzioni = %d\n", num_sol);

    for (i=0; i<dim; i++)
        free(schema[i]);
    free(schema);
    return risultato;
}
```

terminazione

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

indici casella corrente

cella già piena


ricorri su cella successiva

scelta

controlla

ricorri su cella successiva

smarca la cella




```

int controlla(int **schema, int dim, int passo) {
    int r, c, rb, cb, i, j, indici casella corrente), occ[dim];
    r = passo / dim;
    c = passo % dim;
    ciclo sulle righe
    for(r=0; r<dim; r++) {
        azzerare occorrenze
        for(c=0; c<dim; c++)
            occ[c] = 0;
        calcolo occorrenze
        for(c=0; c<dim; c++)
            occ[schema[r][c]-1]++;
        controllo se c'è più di 1 occorrenza
        for(c=0; c<dim; c++)
            if(occ[c] > 1)
                return 0;
    }
}

```

// controllo la colonna (analogo a controllo riga)



```

for(r=0; r<dim; r=r+n) {
    rb = (r/n) * n;
    for(c=0; c<dim; c=c+n) {
        cb = (c/n) * n;
        for(r=0; r<dim; r++)
            occ[r] = 0;
        for(i=rb; i<rb+n; i++)
            for(j=cb; j<cb+n; j++)
                occ[schema[i][j]-1]++;
        for(r=0; r<dim; r++)
            if(occ[r] > 1)
                return 0;
    }
}
return 1;
}

```

indice riga iniziale del blocco

indice colonna iniziale del blocco

azzerare occorrenze

calcolo occorrenze nel blocco

controllo se c'è più di 1 occorrenza

# Ricerca di una sola soluzione



10sudoku1soluz

terminazione

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

cella già piena

ricorri su cella successiva

scelta

controlla

ricorri su cella successiva

smarca la cella

successo

fallimento



# Tour del cavallo

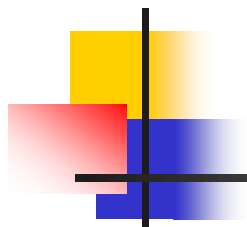
---

Si consideri una scacchiera  $n \times n$ , trovare un "*giro di cavallo*", cioè una sequenza di mosse valide del cavallo tale che ogni casella venga visitata al più una volta (visitata = casella su cui il cavallo si ferma, non casella attraverso cui transita).

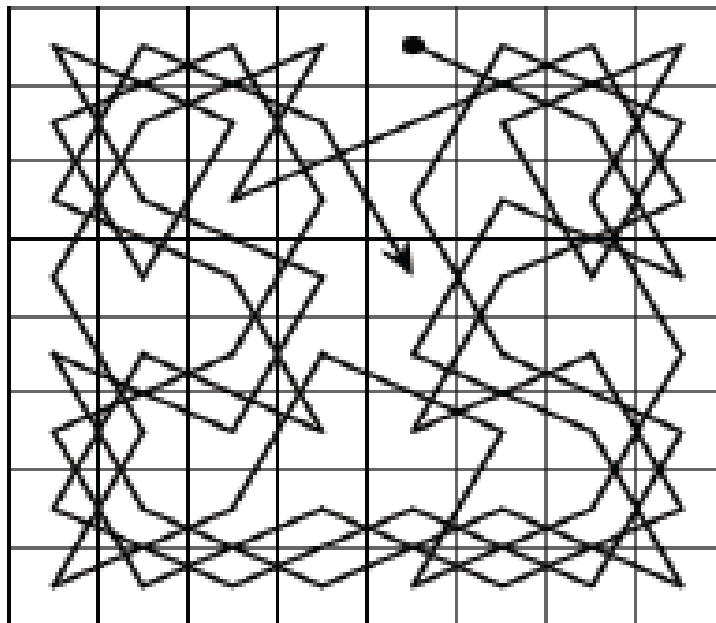
Modello: principio di moltiplicazione con cardinalità dell'insieme scelte dinamica.

Cammino di Hamilton: dato un grafo non orientato, cammino semplice che contiene tutti i vertici.





Soluzione:



# Il labirinto

Dato un labirinto, trovare un cammino dall'ingresso all'uscita (da cella a cella).

Ad ogni passo ci sono al più 4 scelte:

- andare a N
- andare a S
- andare a E
- andare a W

Modello: principio di moltiplicazione

