Examples of search and optimization problems

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

Light bulbs control

Specifications:

- n switches and m light bulbs
- all light bulbs are initially off
- each switch controls a subset of the light bulbs:
  - an element [i,j] of an n x m integer matrix at 1 indicates that switch i controls light bulb j, if it doesn't control it.
- if a switch is pressed, all light bulbs it controls toggle (if on they become off, if off they become on).

Goal:

- find the minimum size set of switches to press to turn on all the light bulbs.

Condition for turning on a light bulb:

- a light bulb is on if and only if the number of pressed switches among the ones that control it is odd.

Example: n=4 m=5

switch 0 doesn’t control light bulb 2

mat_int

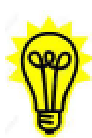
	0	1	2	3	4
0	1	1	0	0	1
1	1	0	1	0	0
2	0	1	1	1	0
3	1	0	0	1	0

switch 2 controls light bulb 3

Effect of **pressing** switches 0 and 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



sw0 controls bulb0

sw2 doesn't control bulb0

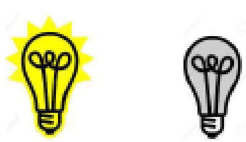
# of pressed switches
that control bulb0

1

Effect of **pressing** switches 0 and 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



sw0 controls bulb1

sw2 controls bulb1

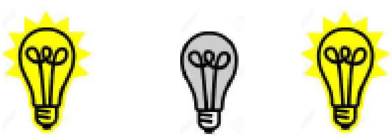
# of pressed switches that control bulb1

2

Effect of **pressing** switches 0 and 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



sw0 doesn’t control bulb2

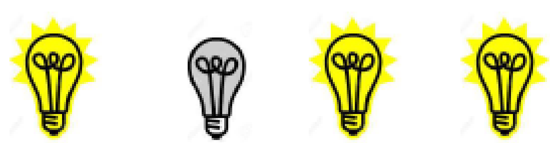
sw2 controls bulb2

# of pressed switches that control bulb2

1

Effect of **pressing** switches 0 and 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



sw0 doesn't control bulb3

sw2 controls bulb3

# of pressed switches that control bulb3

1

Effect of **pressing** switches 0 and 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

sw0 controls bulb4

sw2 doesn't control bulb4

# of pressed switches that control bulb4

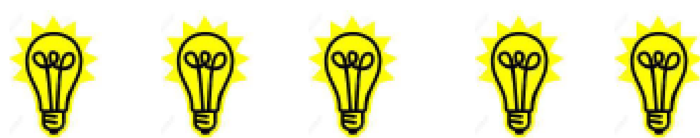
1

INVALID SOLUTION

Effect of **pressing** switches 0, 1 and 3:

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



Control:

bulb0: 3 switches

bulb1: 1 switch

bulb2: 1 switch

bulb3: 1 switch

bulb4: 1 switch

VALID SOLUTION

Algorithm:

- generate all subsets of switches (empty set not needed)
- for each subset apply a validity check function
- among valid solutions, select the first one that exhibits minimum set cardinality.

Model:

- powerset generated with simple n-choose-k combinations
- k grows from 1 to n (empty set not required)
- the first solution found is the minimum size one.

Data structures:

- `n x m` integer matrix `inter`
- n-value integer arrays `sol` and `mark`
- array `val` not required (switches are numbered from 0 to `n-1`)

01interruttori_comb

```
int main(void) {
  int n, m, k, i, found=0;
  FILE *in = fopen("switches.txt", "r");

  int **inter = readFile(in, &n, &m);
  int *sol = calloc(n, sizeof(int));
  int *mark = calloc(n, sizeof(int));

  printf("Powerset with simple combinations\n\n");
  for (k=1; k <= n && found==0; i++) {
    if(powerset(0, sol, n, k, 0, inter, m))
      found = 1;
  }
  free(sol);
  free(mark);
  for (i=0; i < n; i++)
    free(inter[i]);
  free(inter);
  return 0;
}
```

increasing set size from 1 to n

stop as soon as minimum size solution found

empty set not required

validity check

stop as soon as valid solution found

```
int powerset(int pos, int *sol, int n, int k, int start,
             int **mat_int, int m) {
  int i;
  if (pos >= k) {
    if (check(mat_int, n, m, k, sol)) {
      print(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;
}
```

Verification:

- given a subset of k pressed switches
  - for each light bulb count how many switches control it
  - record if even or odd (compute remainder of integer division by 2)
- valid solution if for each light bulb the number of pressed switches that control it is odd.

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

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

∀ switch in the subset

∀ light bulb

count how many switches
In the subset control it

even or odd?

OK if all odd

Alternative verification:

- array of light bulbs (initially all turned off)
- for each light bulb
  - for each switch in the subset

light bulb	switch: no control	switch: control
off	off	on
on	on	off

state of light bulb

lampadina \ switch	0	1
0	0	1
1	1	0

light bulb EXOR switch

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

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

∀ switch in the subset

∀ light bulb

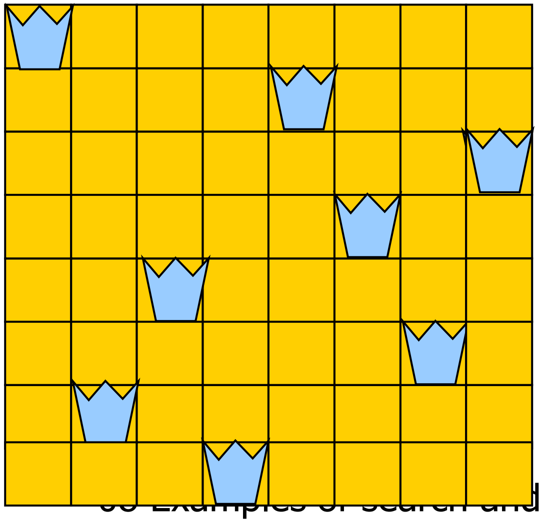
light bulb EXOR switch

OK if all ON

The 8 queens (Max Bezzel 1848)

Given an 8 x 8 checkerboard, place 8 queens so that none is in check:

- 92 solutions
- 12 fundamental solutions (taking into account rotations and simmetries)
- Example:



May be generalized to N queens, with $N \geq 4$:

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

Search problem: find

- 1 solution
- all solutions

NB: queens are not distinguishable. Models that consider them distinct generate identical solutions taking into account permutations, simmetries and rotations.

Model #0:

- Each cell may contain an indistinct queen or not (the number of queens ranges from 0 to 64)
- **Powerset with arragements with repetition**
- pruning necessary
- filter out solutions constraining the number of queens to 8
- $D'_{n,k} = 2^{64} \approx 1.84\ 10^{19}$ cases (without pruning)!
- global variable `board[N][N]`
- variable `q` that plays the role of variable `pos`.

10otto-regine-powerset

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

board completed!

try to put the queen on r,c

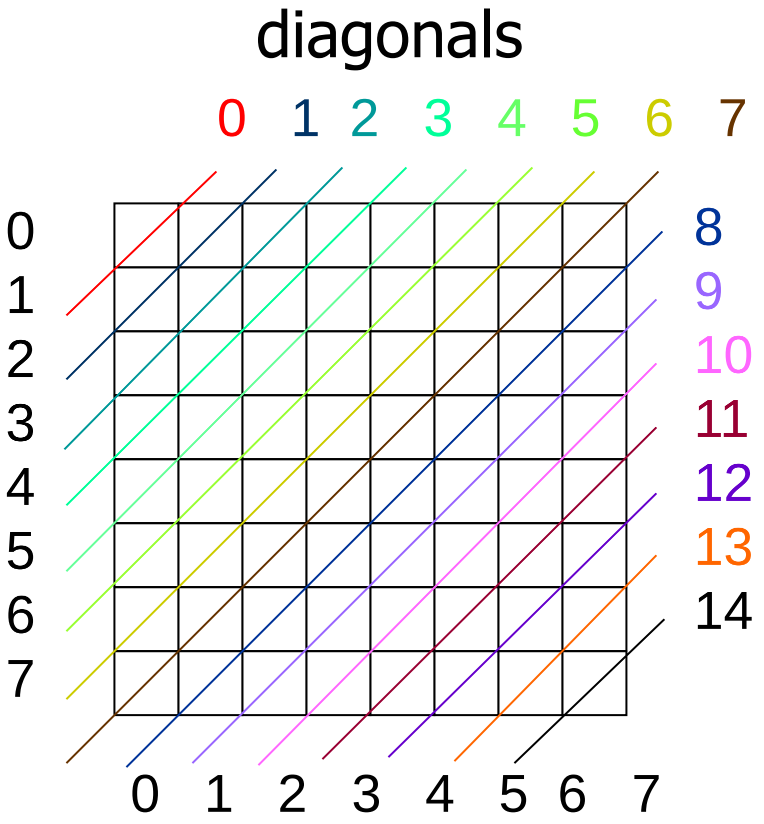
recur

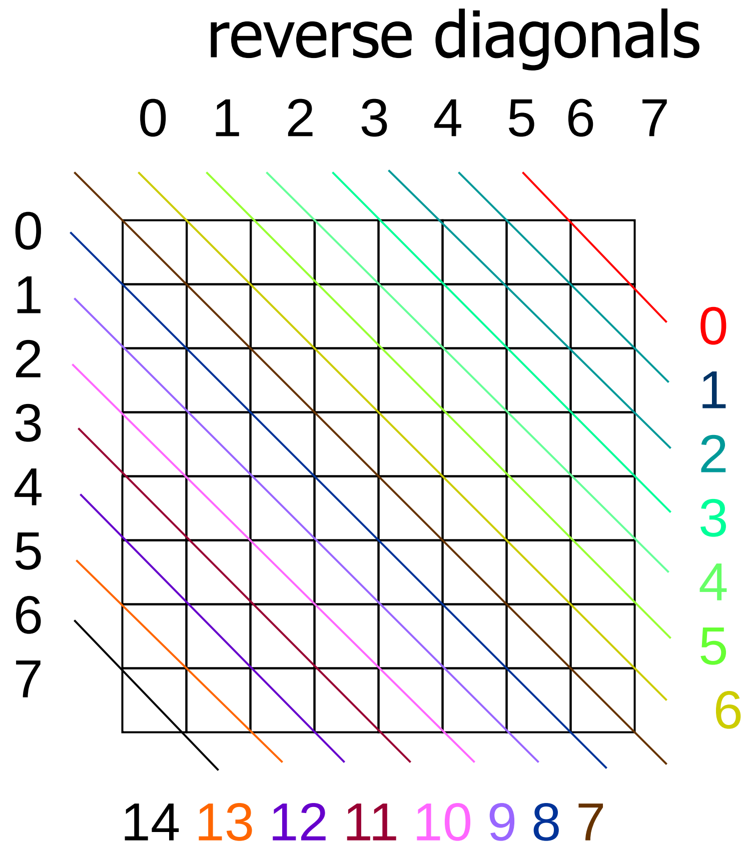
backtrack

recur without the queen on r,c

Function `check`:

- rows , columns, diagonals and reverse diagonals: count for each the number of cells $\neq$ 0. If such value is >1, the solution is unacceptable
- diagonals:
  - 15 diagonals identified by the sum of row and colum indices
  - 15 reverse diagonals identified by the difference of row and column indices (+ 7 to avoid negative values)


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


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

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

check rows

check colums

check diagonals

check reverse diagonals

```
  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 (board[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 (board[r][c]!=0) n++;
    }
    if (n>1) return 0;
  }
  return 1;
}
```

Model #1:

- Place 8 distinct queens (k = 8) in 64 cells (n = 64)
- **simple arrangements**

order matters

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

- global variable `board[N][N]` that plays the role of array `mark`
- variable `q` that plays the role of variable `pos`.

```
void simpl_arr(int q) {
  int r,c;
  if (q >= N) {
    if(check()) {
      num_sol++;
      print();
    }
    return;
  }
  for (r=0; r<N; r++)
    for (c=0; c<N; c++)
    if (board[r][c] == 0) {
      board[r][c] = q+1;
      simpl_arr(q+1);
      board[r][c] = 0;
    }
  return;
}
```

all queens placed

check if cell empty

put queen on r,c

recur

backtrack

Model 2:

- Place 8 undistinct queens (k = 8) in 64 cells (n = 64)
- **simple combinations**

order doesn't matter

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

- global variable `s[N][N]` for the board
- variable `q` that plays the role of variable `pos`.
- variables `r0` and `c0` to force an ordering.

```c
void simpl_comb(int r0, int c0, int q) {
  int r,c;
  if (q >= N) {
    if(check()) {
      num_sol++; print();
    }
    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;
        simpl_comb(r,c,q+1);
        s[r][c] = 0;
      }
  return;
}
```

all queens placed

iteration on choices

check feasibility of choice

choice

recur

backtrack

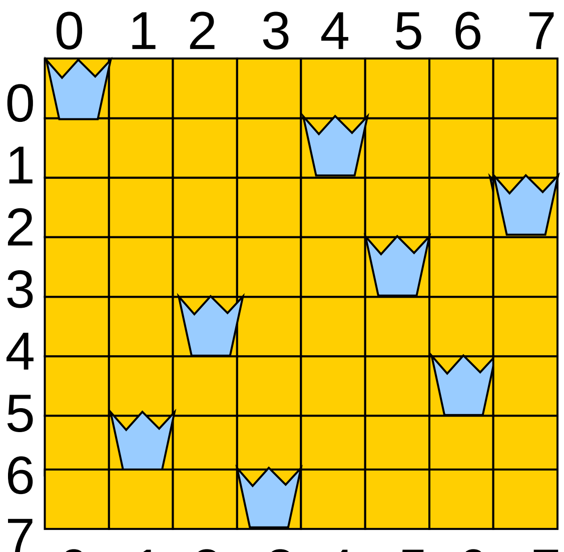
order matters

Model 3:

- one-dimensional data structure:
  - Each row contains 1 and only 1 <u>distinct</u> queen in one of the 8 columns (n = 8)
- there are 8 rows (k = 8)
- **arrangements with repetitions**

$D'_{n,k} = n^k = 8^8 = 16.777.216$ cases!

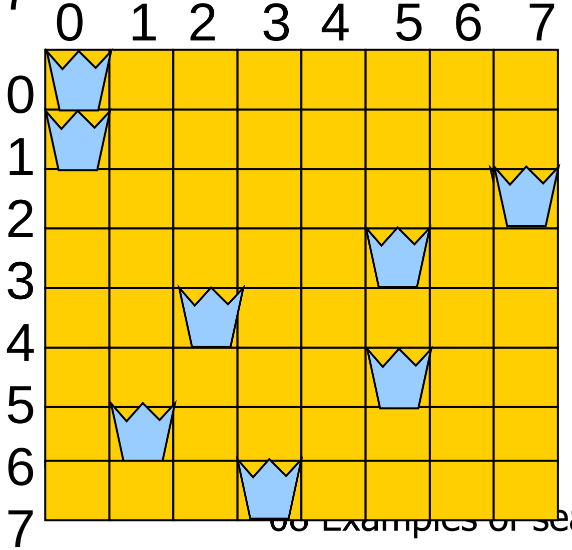
- no row check needed, enough check on columns, diagonals and reverse diagonals
- variable `row[N]`
- variable `q` that plays the role of variable `pos`.



sol

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

check()=1



sol

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

check()=0

```c
void rep_arr(int q) {
  int i;
  if (q >= N) {
    if(check()) {
      num_sol++;
      print();
    }
    return;
  }
  for (i=0; i<N; i++) {
    riga[q] = i;
    rep_arr(q+1);
  }
  return;
}
```

board completed!

put queen on row

recur

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

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

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

occurrence array

check columns

check diagonals

check reverse diagonals

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

order matters

Model 4:

- Each row and each column contain 1 and only 1 distinct queen in 1 of the 8 columns (n = 8)
- there are 8 rows (k = 8)
- **simple permutations**

$P_n = D_{n,n} =$ n! = 40320 cases!

- global variables `sol[N]` and `mark[N]`
- variable `q` that plays the role of variable `pos`.
- check diagonals and reverse diagonals only.

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

board completed!

put queen on row

recur

backtrack

```c
int check (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;
}
```

check diagonals

check reverse diagonals

time for space trade-off

Model 4 (optimized version):

- use 2 arrays `d[2*N-1]` and `ad[2*N-1]` to mark diagonals and reverse diagonals in check by a queen
- pruning: check if choice feasible before recursive descent.

```c
void simpl_perm(int q) {
  int c;
  if (q >= N) {num_sol++; check(); 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;
      simpl_perm(q+1);
      mark[c] = 0;
      d[q+c] = 0;
      ad[q-c+(N-1)] = 0;
    }
  return;
}
```

board completed!

check

put queen on row

recur

backtrack

Cryptoarithmetic

Specifications:

input: 3 strings, 1 operation (addition)

Example:

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

interpretation:

strings are “encrypted” integers, i.e., each letter represents 1 and only 1 decimal digit.

Output: decrypt strings, i.e., identify the matching letters – decimal digits that satisfies the addition.

Disreagard solutions where the most significant letter (leftmost letter) corresponds to 0.

Solution:

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
```

Strings have `dist_lett` (<=10) letters, each being associated to 1 and only 1 decimal digit 0..9

Model: simple arrangements of n-choose-k elements, where n = 10 and k = `dist_lett`.

Data structures

symbol table

- Integer global variable `dist_lett`
- Array `letters[10]` of `struct` of type `alpha` with a `car` (distinct character) and `val` (corresponding decimal digit)
- Array `mark[10]` to mark already considered digits.

Algorithm

- Read the 3 strings
- Fill in array `letters`
- Compute simple arrangements:
  - when in terminal case, replace letters with digits, convert to integer, check the validity of the solution and, if valid, print it.

Functions

access symbol table

`int find_index(alpha *letters, char c)`

give character `c`, find and return its index in the array `letters`, if not present return -1

`alpha * init_alpha()`

allocate `letters[10]` and initialize it (value = -1, character = \0)

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

given the 3 strings, put the distinct characters in `letters` and count them (`dist_lett`)

```
void print(alpha *letters)
```

Print the matching letters-digits stored in fields `car` and `val` of `lettere`

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

in string `str` replace letters with digits, based on the correspondence stored in `letters`, convert to integer, returning -1 when the string’s leftmost digit is 0.

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

check that the 3 strings converted to integers satisfy the sum

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

compute simple arrangements of $n=10$ decimal digits k by k, where k=`dist_lett`

SEND MORE MONEY

`letters`

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

after `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

after `setup`

`dist_lett = 8`

`lettere`

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

after `arr`: example of possibile arrangement

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

global variable

08aritmetica_verbale

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

  // Read 3 strings

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

  arr(letters, mark, 0, str1, str2, str3);

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

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

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

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

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

```
int arr(alpha *letters, int *mark, int pos, char *st1,
         char *st2, char *st3) {
  int i = 0;
  if (pos == dist_lett) {
    int solved = check_sol(letters, st1, st2, st3);
    if (solved) print(letters);
    return solved;
  }
  for(i=0;i < base; i++) {
    if (mark[i]==0) {
        letters[pos].val = i; mark[i] = 1;
        if (arr(letters, mark, pos+1, st1, st2, st3))
            return 1;
        letters[pos].val = -1; mark[i] = 0;
    }
  }
  return 0;
}
```

Sudoku

Input:

- 9×9 cell grid
- cell either empty or contains a digit from 1 to 9
- 9 horizontal rows, 9 vertical columns
- thicker lines identify 9 regions, each with 3×3 contiguous cells
- initially between 20 to 35 non empty cells

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

Goal: fill all empty cells with digits in the range from 1 to 9, so that in each row, column and region appear the digits from 1 to 9 without repetitions.

Possible solution:

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

Model:

- arrangements with repetitions
- n = number of empty cells, k =9
- space size: n^9

Search for all the solutions.

Two kinds of recursion:

- cell already filled: no choice
- empty cell: there is a choice. Upon return, cancel previous choice (otherwise one would revert to the previous case)
- pruning: check before recurring.

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

global variable

09sudoku

```
int main() {
  int **grid, dim, i, j, result;
  char filename[20];

  printf("Input file name: ");
  scanf("%s", filename);
  grid = readFile(filename, &dim);

  rep_arr(grid, dim, 0);

  printf("\n Number of solutions = %d\n", num_sol);

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

```
void rep_arr(int **grid, int size, int pos) {
  int i, j, k;
  if (pos >= size*size) {
    num_sol++; print(grid, size); return;
  }
  i = pos / size; j = pos % size;
  if (grid[i][j] != 0) {
    rep_arr(grid, size, pos+1);
    return;
  }
for (k=1; k<=size; k++) {
    grid[i][j] = k;
    if (check(grid, size, pos))
      rep_arr(grid, size, pos+1);
    grid[i][j] = 0;
  }
  return;
}
```

termination

indices for current cell

already filled cell

recur on next cell

choice

check

recur on next cell

unmark cell

```
int check(int **grid, int size, int step) {
  int r,c, rb, cb, i, j, )), occ[dim];
  r = step/ size;
  c = step% size;
  for(r=0; r<size; r++) {
    for(c=0; c<size; c++)
      occ[c] = 0;
    for(c=0; c<size; c++)
      occ[grid[r][c]-1]++;
    for(c=0; c<size; c++)
      if(occ[c] > 1)
        return 0;
  }

  // check column (similar to row control)
```

indices for current cell

row loop

clear occurrences

compute occurrences

check if more than 1 occurrence

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

index of initial row in block

index of initial control in block

clear occurrences

compute occurrences in block

check if more than 1 occurrence

```
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;
}
```

termination

already filled cell

recur on next cell

choice

controll

recur on next cell

unmark cell

success

failure

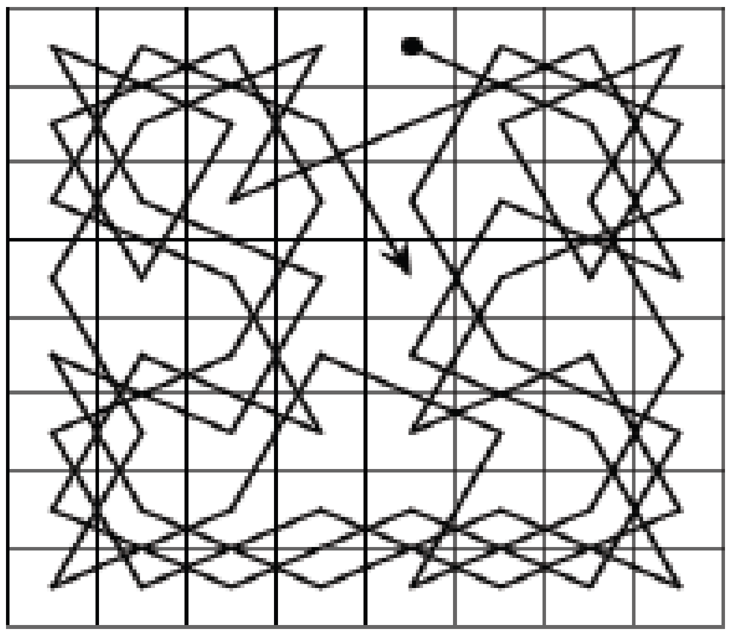
Knight's Tour

Given an n x n board, find a *"knight's tour"*, that is a sequence of valid moves such that each cell is visited at most once (visited = cell on which the knight stops, not just traversed cell).

Model: principle of multiplucation with dynamic set of choices.

Hamiltonian path: simple path on an undirected graph that contains all the vertices.

Solution:



The Labyrinth

Given a labyrinth, find a path from entrance to exit (from cell to cell).

At each step there are at most 4 choices:

- go N
- go S
- go E
- go W

Model: principle of multiplication