

Discrete Optimization

Constraint Programming: Part X

Goals of the lecture

- ▶ Search in constraint programming
 - introduction
 - active research area

Searching in constraint programming

- ▶ variable/value labeling
- ▶ focusing on the objective
- ▶ value/variable labeling
- ▶ domain splitting
- ▶ symmetry breaking during search
- ▶ randomization and restarts

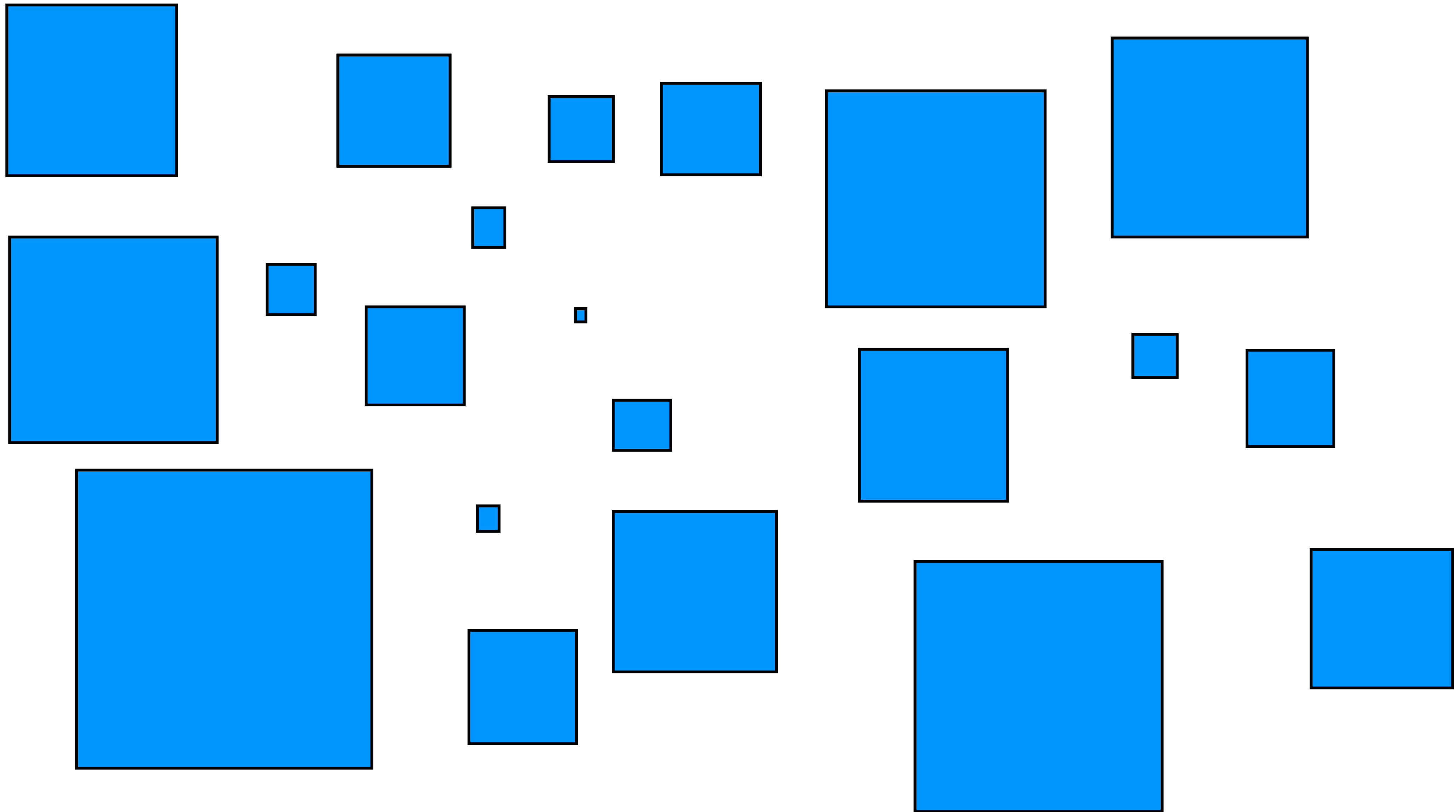
Value/variable labeling

- ▶ Two steps
 - choose the value to assign next
 - choose the variable to assign to this value

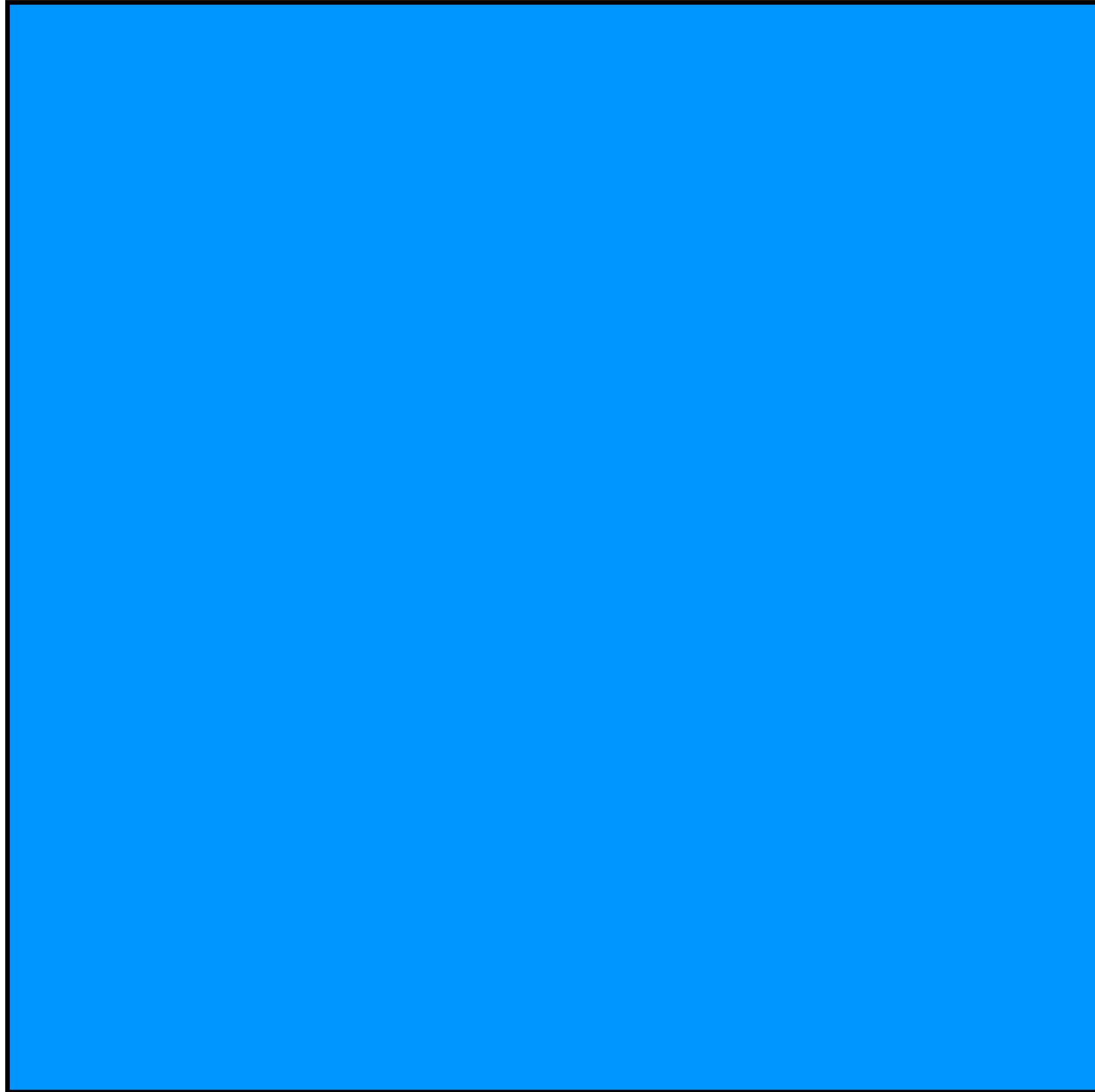
Value/variable labeling

- ▶ Two steps
 - choose the value to assign next
 - choose the variable to assign to this value
- ▶ Why it is useful?
 - you may know that a value must be assigned
 - often the case in scheduling and resource allocation problems

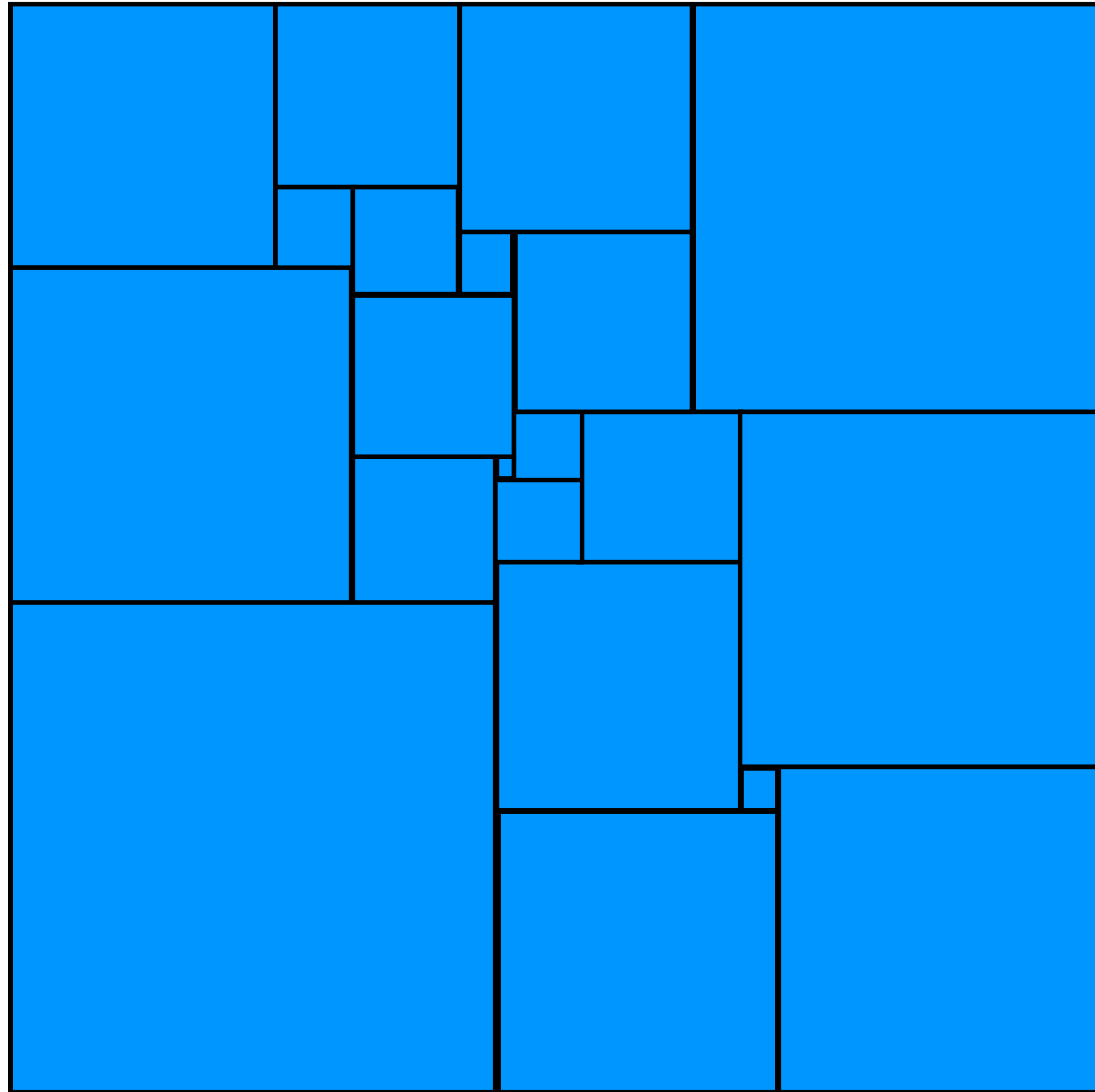
The perfect square problem



The perfect square problem



The perfect square problem



The perfect square problem

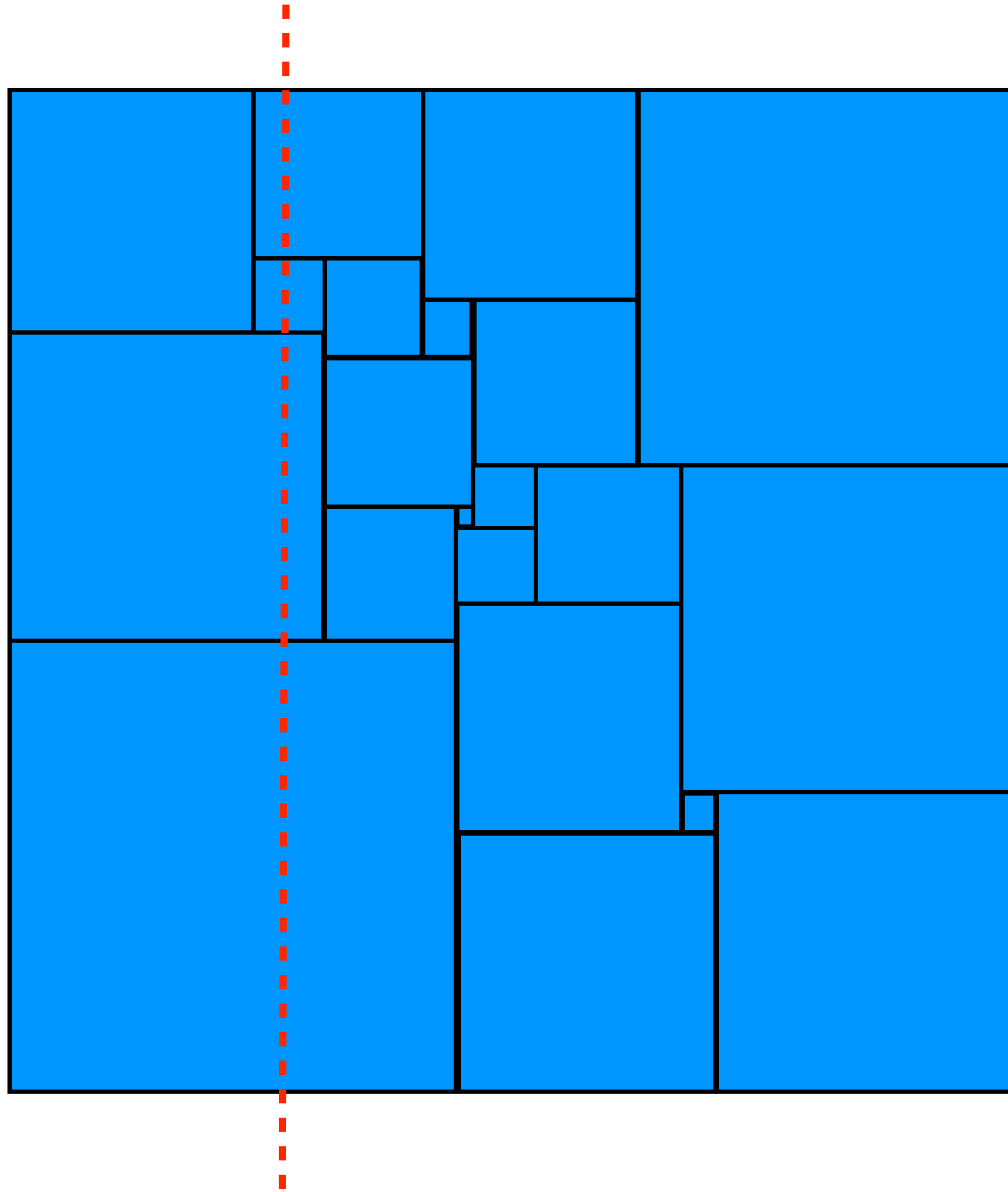
The perfect square problem

- ▶ What are the decision variables?
 - x and y -coordinates of the bottom-left corner of every square
- ▶ What are the constraints?
 - the squares fit in the larger square
 - the squares do not overlap

The perfect square problem

- ▶ What are the decision variables?
 - x and y -coordinates of the bottom-left corner of every square
- ▶ What are the constraints?
 - the squares fit in the larger square
 - the squares do not overlap
- ▶ There are also redundant constraints
 - here is the intuition

The perfect square problem



The perfect square problem

```
range R = 1..8;
int s = 122; range Side = 1..s; range Square = 1..21
int side[Square] = [50,42,37,35,33,29,27,25,24,19,18,17,16,15,11,9,8,7,6,4,2];
var{int} x[Square] in Side;
var{int} y[Square] in Side;

solveall {
  forall(i in Square) {
    x[i]<=s-side[i]+1;
    y[i]<=s-side[i]+1;
  }
  forall(i in Square,j in Square: i<j)
    x[i]+side[i]<= x[j] || x[j]+side[j]<=x[i] || y[i]+side[i]<=y[j] || y[j]+side[j]<=y[i];

  forall(p in Side) {
    sum(i in Square) side[i]*((x[i]<=p) && (x[i]>=p-side[i]+1)) = s;
    sum(i in Square) side[i]*((y[i]<=p) && (y[i]>=p-side[i]+1)) = s;
  }
}
```


The perfect square problem

```
range R = 1..8;
int s = 122; range Side = 1..s; range Square = 1..21
int side[Square] = [50,42,37,35,33,29,27,25,24,19,18,17,16,15,11,9,8,7,6,4,2];
var{int} x[Square] in Side;
var{int} y[Square] in Side;

solveall {
  forall(i in Square) {
    x[i]<=s-side[i]+1;
    y[i]<=s-side[i]+1;
  }
  forall(i in Square,j in Square: i<j)
    x[i]+side[i]<= x[j] || x[j]+side[j]<=x[i] || y[i]+side[i]<=y[j] || y[j]+side[j]<=y[i];

  forall(p in Side) {
    sum(i in Square) side[i]*((x[i]<=p) && (x[i]>=p-side[i]+1)) = s;
    sum(i in Square) side[i]*((y[i]<=p) && (y[i]>=p-side[i]+1)) = s;
  }
}
```

redundant
constraints

The perfect square problem

```
range R = 1..8;
int s = 122; range Side = 1..s; range Square = 1..21
int side[Square] = [50,42,37,35,33,29,27,25,24,19,18,17,16,15,11,9,8,7,6,4,2];
var{int} x[Square] in Side;
var{int} y[Square] in Side;

solveall {
  forall(i in Square) {
    x[i]<=s-side[i]+1;
    y[i]<=s-side[i]+1;
  }
  forall(i in Square,j in Square: i<j)
    x[i]+side[i]<= x[j] || x[j]+side[j]<=x[i] || y[i]+side[i]<=y[j] || y[j]+side[j]<=y[i];

  forall(p in Side) {
    sum(i in Square) side[i]*((x[i]<=p) && (x[i]>=p-side[i]+1)) = s;
    sum(i in Square) side[i]*((y[i]<=p) && (y[i]>=p-side[i]+1)) = s;
  }
}
```

non-overlapping constraints

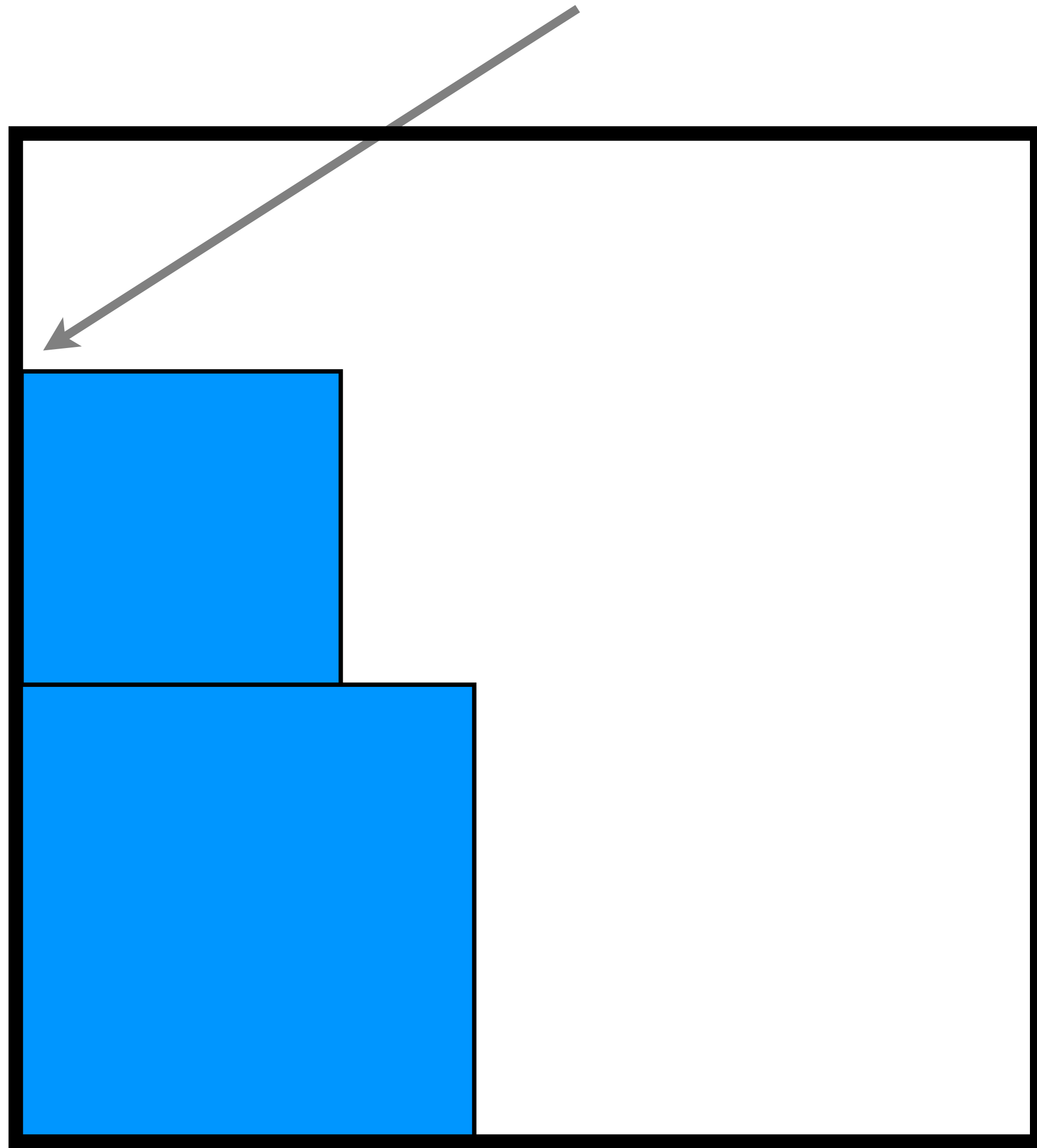
redundant constraints

The value/variable labeling

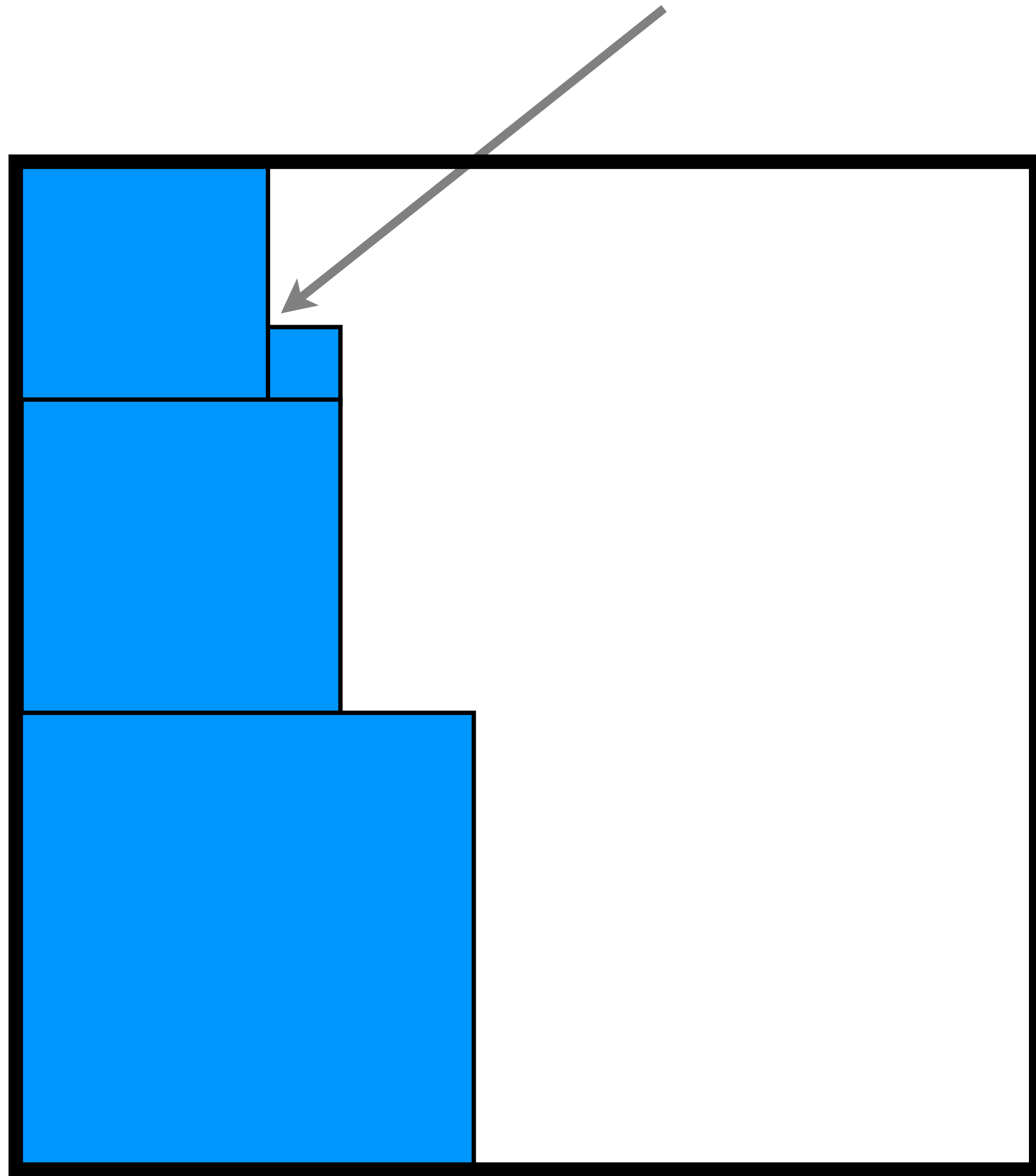
The value/variable labeling

- ▶ Why a value/variable labeling
 - we know that there is no empty space in the square to be filled.

The perfect square problem



The perfect square problem



The value/variable labeling

The value/variable labeling

- ▶ What is the labeling doing?
 - choose a x-coordinate p
 - for all square i , decide whether to place i at coordinate p
 - that is, whether the bottom-left corner of i has x-coordinate p
 - repeat for all x-coordinates
 - repeat for all y-coordinates

The value/variable labeling

The value/variable labeling

- ▶ What is the labeling doing?
 - choose a x-coordinate p
 - for all square i , decide whether to place i at coordinate p
 - that is, whether the bottom-left corner of i has x-coordinate p
 - repeat for all x-coordinates
 - repeat for all y-coordinates

The perfect square problem

```
using {  
  
    forall(p in Side)  
        forall(i in Square)  
            try  
                x[i] = p;  
            |  
                x[i] != p;  
  
    forall(p in Side)  
        forall(i in Square)  
            try  
                y[i] = p;  
            |  
                y[i] != p;  
  
}
```

The perfect square problem

```
using {
```

```
  forall(p in Side)
    forall(i in Square)
      try
        x[i] = p;
      |
        x[i] != p;
```

```
  forall(p in Side)
    forall(i in Square)
      try
        y[i] = p;
      |
        y[i] != p;
```

```
}
```

choose a x-coordinate p

The perfect square problem

```
using {
```

```
  forall(p in Side)
```

```
    forall(i in Square)
```

```
      try
```

```
        x[i] = p;
```

```
      |
```

```
        x[i] != p;
```

```
  forall(p in Side)
```

```
    forall(i in Square)
```

```
      try
```

```
        y[i] = p;
```

```
      |
```

```
        y[i] != p;
```

```
}
```

choose a x-coordinate p

consider a square i

The perfect square problem

```
using {
```

```
  forall(p in Side)
```

```
    forall(i in Square)
```

```
      try
```

```
        x[i] = p;
```

```
      |
```

```
        x[i] != p;
```

```
  forall(p in Side)
```

```
    forall(i in Square)
```

```
      try
```

```
        y[i] = p;
```

```
      |
```

```
        y[i] != p;
```

```
}
```

choose a x-coordinate p

consider a square i

decide whether to place
i at position p

Searching in constraint programming

- ▶ variable/value labeling
- ▶ focusing on the objective
- ▶ value/variable labeling
- ▶ domain splitting
- ▶ symmetry breaking during search
- ▶ randomization and restarts

The magic square problem

- ▶ Place numbers in a square
 - all numbers are different
 - all rows, columns, and diagonals sum to the same number

The magic square problem

2	9	4
7	5	3
6	1	8

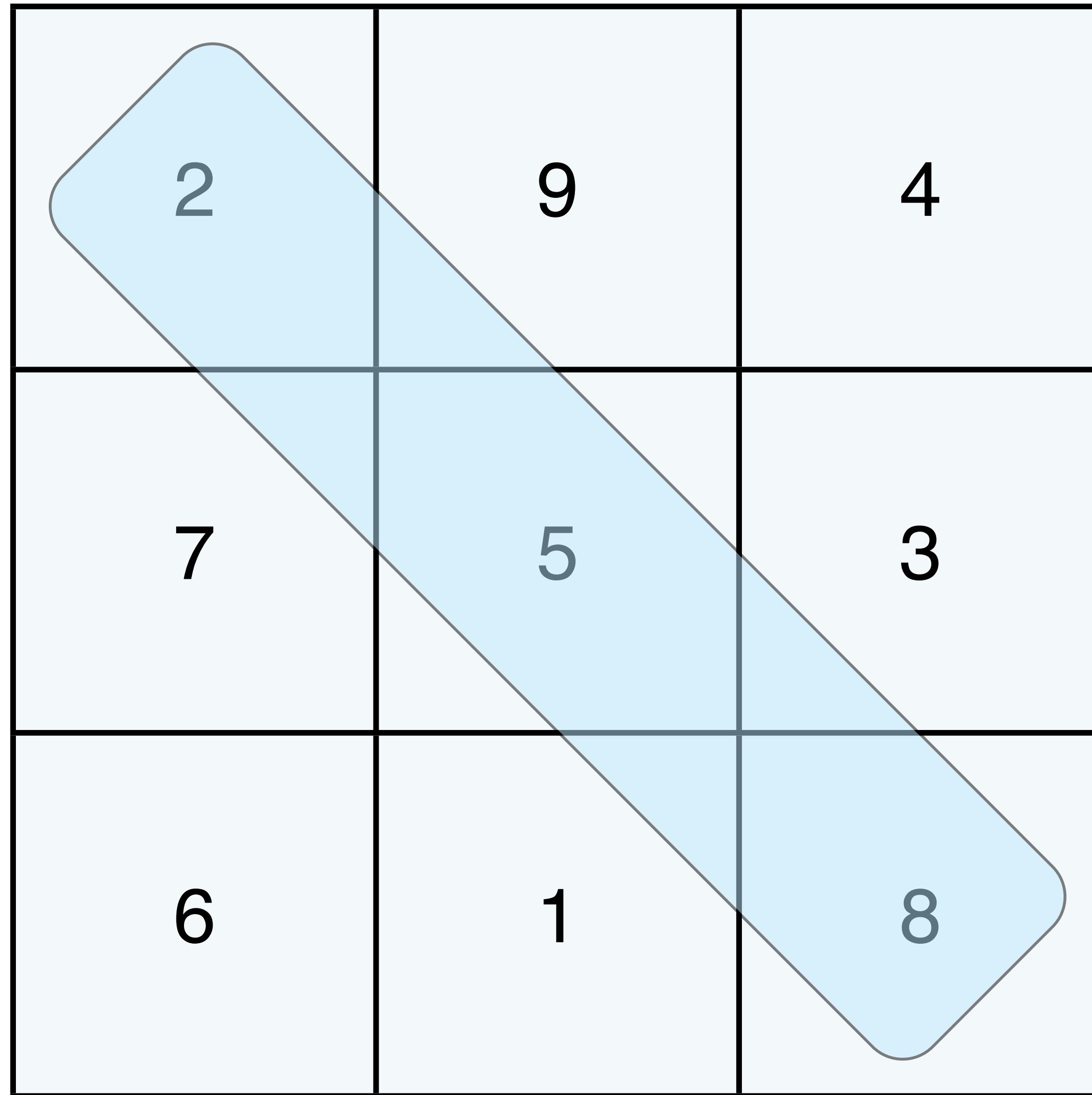
The magic square problem

2	9	4
7	5	3
6	1	8

The magic square problem

2	9	4
7	5	3
6	1	8

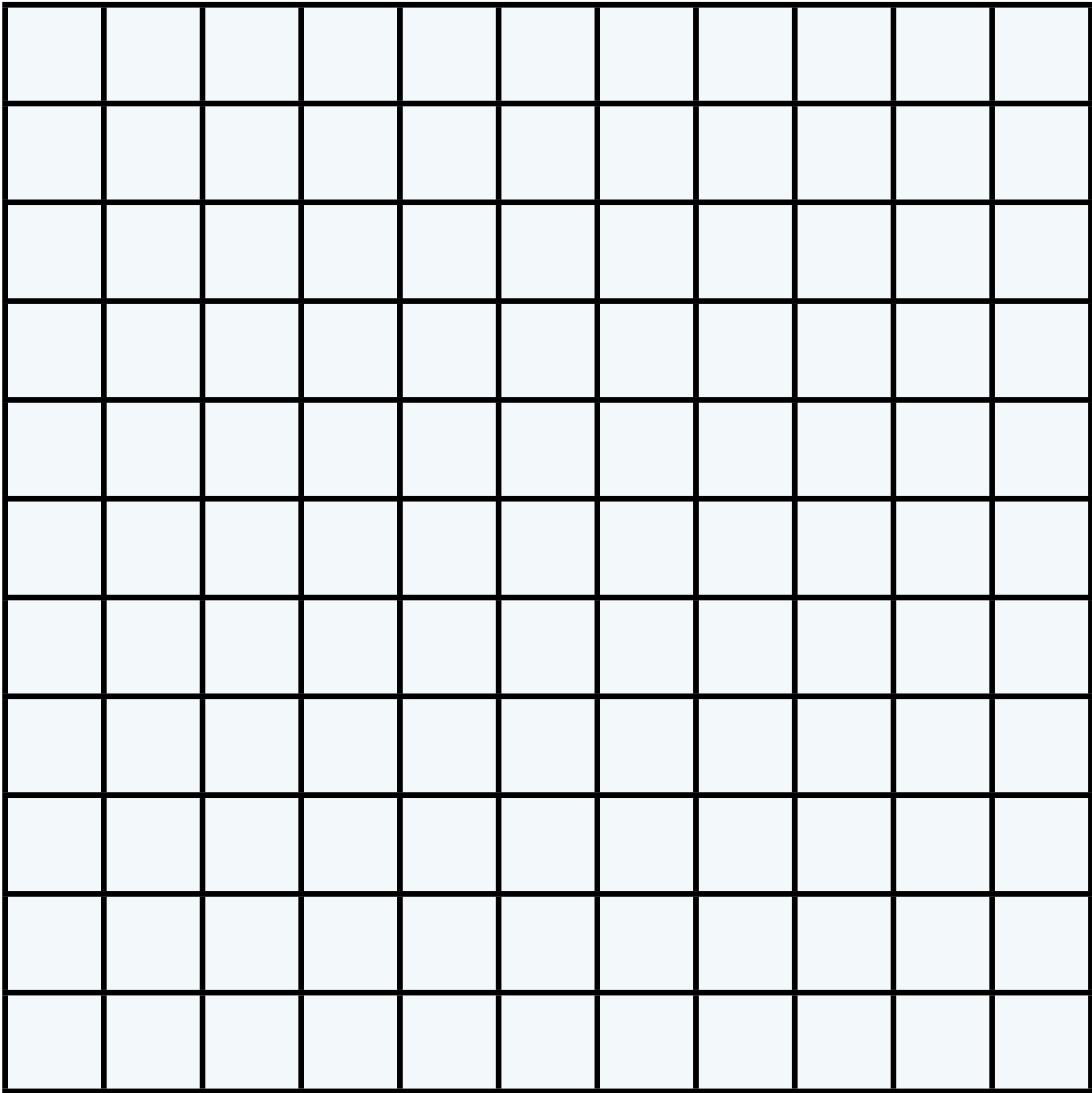
The magic square problem



A 3x3 grid representing a magic square. The grid is divided into three rows and three columns. A blue diagonal band, consisting of three rounded squares, runs from the top-left to the bottom-right. The numbers 2, 5, and 8 are placed within this band. The other cells of the grid contain the numbers 9, 4, 7, 3, 6, and 1.

2	9	4
7	5	3
6	1	8

The magic square problem



The magic square problem

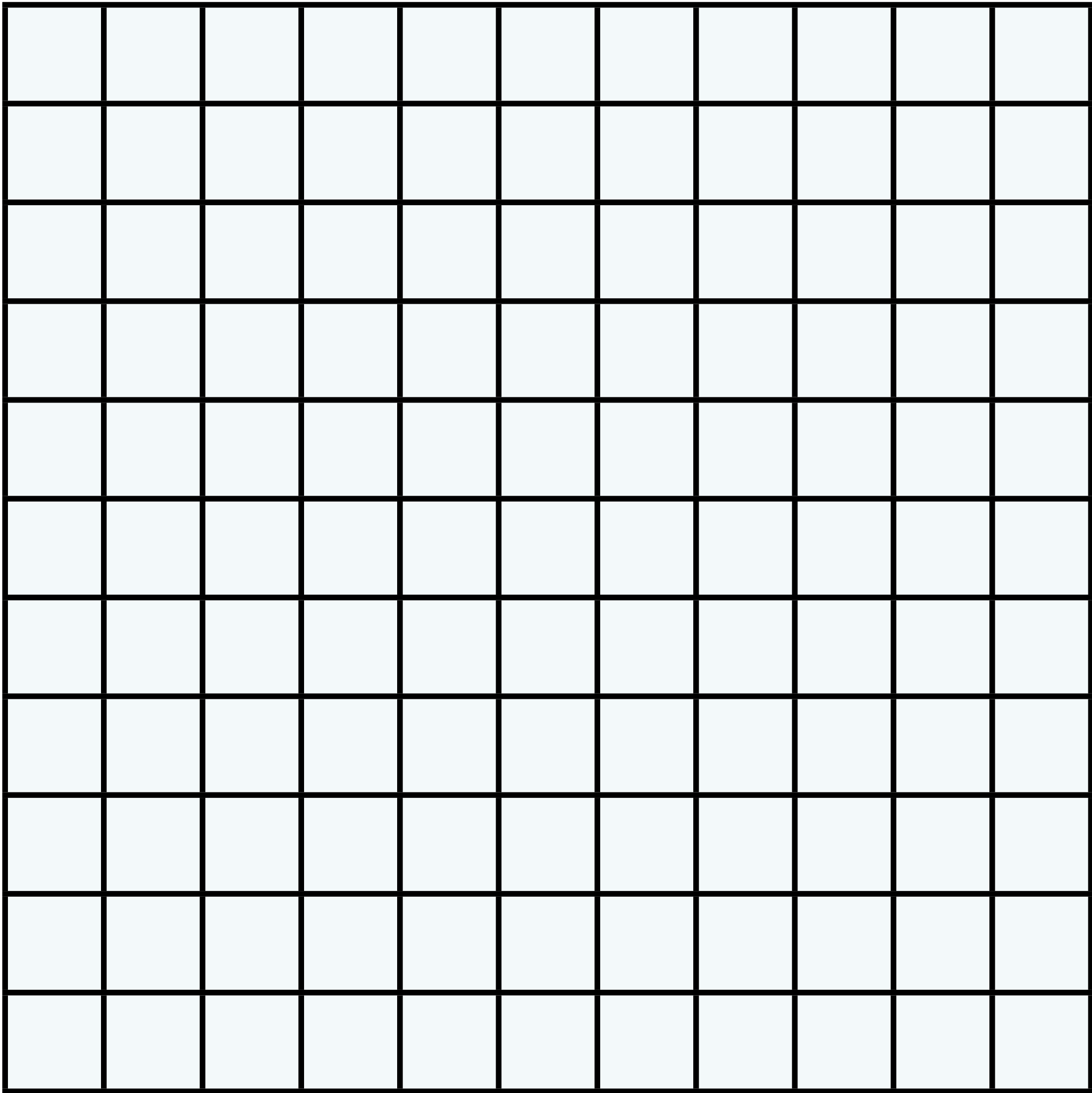
1	90	18	14	119	112	117	118	20	54	8
93	2	99	16	29	92	94	31	68	56	91
39	75	3	5	12	87	95	101	57	97	100
21	83	103	4	86	78	84	58	89	15	50
79	70	98	107	55	51	59	28	25	76	23
24	106	63	32	109	60	9	108	104	34	22
69	49	105	110	61	26	82	7	13	72	77
73	46	102	62	40	36	38	113	81	33	47
71	42	64	114	41	27	44	53	115	52	48
80	65	6	111	74	67	19	37	11	116	85
121	43	10	96	45	35	30	17	88	66	120

The magic square problem

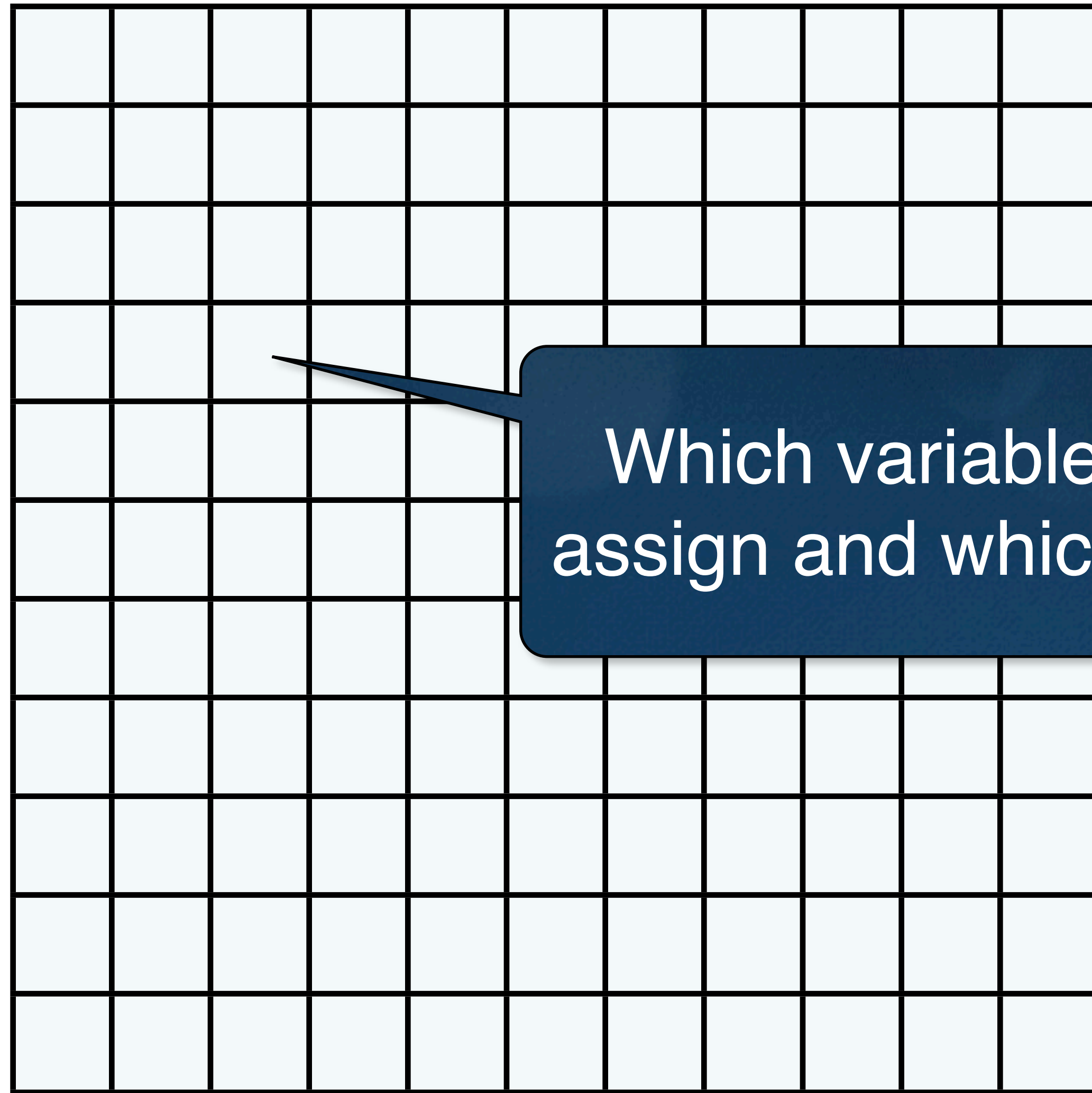
The magic square problem

```
range R = 1..n;  
range D = 1..n^2;  
int T = n*(n^2+1)/2;  
var{int} s[R,R] in D;  
solve {  
    forall(i in R) {  
        sum(j in R) s[i,j] = T;  
        sum(j in R) s[j,i] = T;  
    }  
    sum(i in R) s[i,i] = T;  
    sum(i in R) s[i,n-i+1] = T;  
    alldifferent(all(i in R,j in R) s[i,j]);  
}
```


The magic square problem

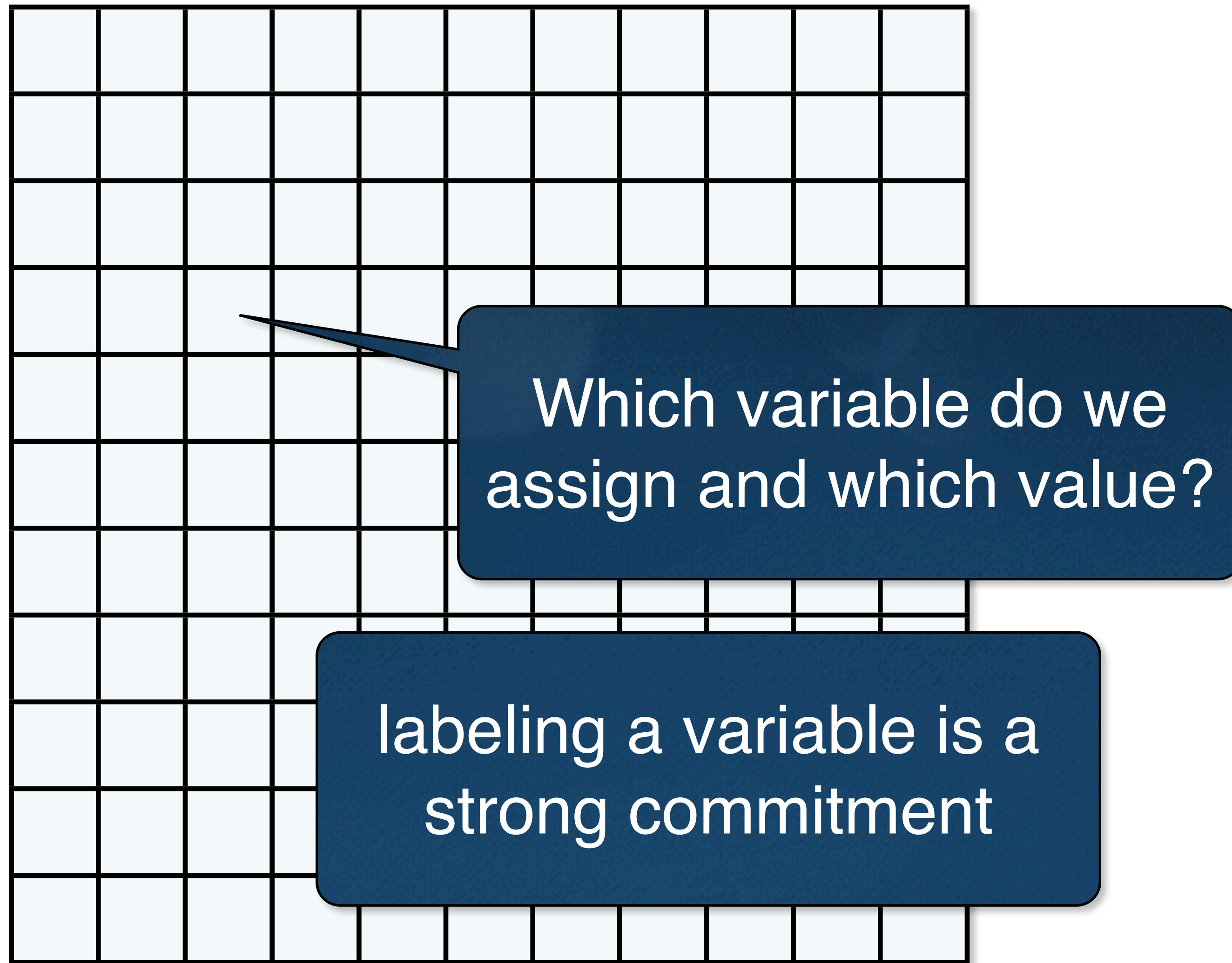


The magic square problem



Which variable do we
assign and which value?

The magic square problem



Domain splitting

- ▶ Select a variable

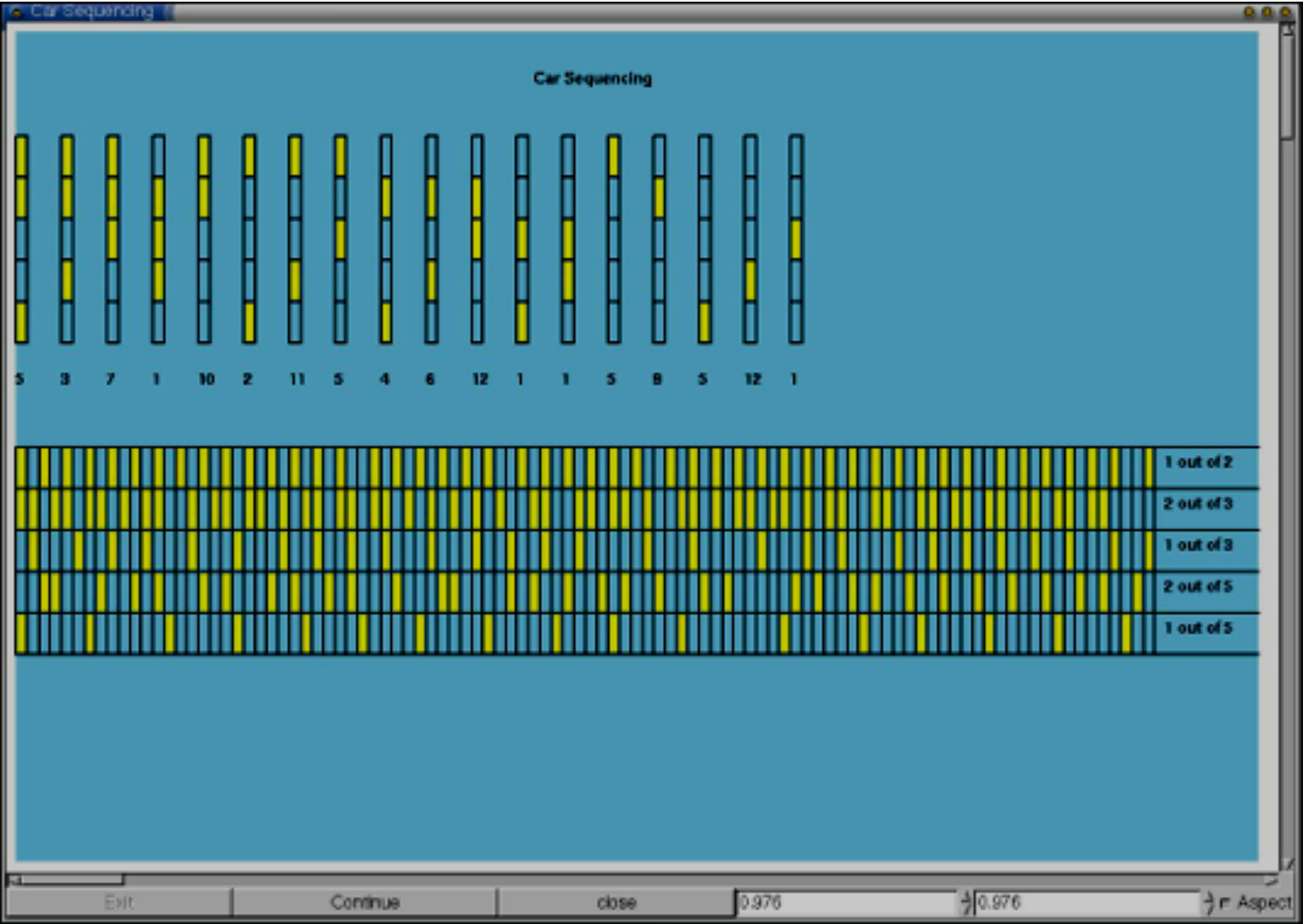
Domain splitting

- ▶ Select a variable
- ▶ Split its domain in two or more sets
 - much weaker commitment

The magic square problem

```
using {
  var{int}[] x = all(i in R,j in R) s[i,j];
  range V = x.getRange();
  while (!bound(x)) {
    selectMin(i in V: !x[i].bound()) (x[i].getSize()) {
      int mid = (x[i].getMin()+x[i].getMax())/2;
      try
        x[i] <= mid;
      |
        x[i] > mid;
    }
  }
}
```

Car sequencing



Domain splitting in car sequencing

► Motivation

- focus on difficult options
- decide which slots take these options

► Domain splitting

- do not assign configurations to slots
- decide whether the slot take the option

Domain splitting in car sequencing

```
using {  
  forall(o in Options) by (slack[o])  
    forall(i in Slots)  
      try  
        line[i] in options[o];  
      |  
        line[i] notin options[o]);  
}
```

Domain splitting in car sequencing

```
using {  
  forall(o in Options) by (slack[o])  
    forall(i in Slots)  
      try  
        line[i] in options[o];  
      |  
        line[i] notin options[o]);  
}
```

start with the
option that has
the least slack

Domain splitting in car sequencing

```
using {  
  forall(o in Options) by (slack[o])  
    forall(i in Slots)  
      try  
        line[i] in options[o];  
      |  
        line[i] not in options[o];  
}
```

start with the
option that has
the least slack

Force the slot to
take the option

Domain splitting in car sequencing

```
using {  
  forall(o in Options) by (slack[o])  
    forall(i in Slots)  
      try  
        line[i] in options[o];  
      |  
        line[i] not in options[o];  
}
```

start with the
option that has
the least slack

Force the slot to
take the option

Force the slot
not to take the
option

Searching in constraint programming

- ▶ variable/value labeling
- ▶ focusing on the objective
- ▶ value/variable labeling
- ▶ domain splitting
- ▶ **symmetry breaking during search**
- ▶ randomization and restarts

Scene allocation

► How can we eliminate these symmetries?

- consider the first scene s_1 . What are the days that we consider for this scene?

Only one day, say day 1

- consider the second scene s_2 . Which days must be considered for s_2 ?

Only two days, day 1 and day 2

- In general the already used days and one additional new day

existing days

new day

$$D(s_k) = \{1, 2, \dots, \max(s_1, \dots, s_{k-1}) + 1\}$$

Scene allocation

```
range Scenes = 1..n;
range Days   = 1..m;
range Actor  = ...;
int fee[Actor] = ...;
set{Actor} appears[Scenes] = ...;
set{int} which[a in Actor] = setof(i in Scenes) member(a,appears[i]);
var{int} shoot[Scenes] in Days;

minimize
    sum(a in Actor) sum(d in Days)
        fee[a] * or(s in which[a]) (shoot[s]=d)
subject to {
    atmost(all(i in Days) 5,Days,shoot);
    scene[1] = 1;
    forall(s in Scenes: s > 1)
        scene[s] <= max(k in 1..s-1) scene[k] + 1;
}
```


Symmetry-breaking constraints

Symmetry-breaking constraints

- ▶ Side effect
 - interferences with the search heuristic

Symmetry-breaking constraints

- ▶ Side effect
 - interferences with the search heuristic
- ▶ Can we avoid this?
 - symmetry-breaking during search
 - dynamically impose the symmetry-breaking constraints

Symmetry-breaking constraints

- ▶ Side effect
 - interferences with the search heuristic
- ▶ Can we avoid this?
 - symmetry-breaking during search
 - dynamically impose the symmetry-breaking constraints
- ▶ How?
 - same constraints
 - the order is different and discovered dynamically

Symmetry-breaking during search

Symmetry-breaking during search

- ▶ Choose a scene to shoot
 - use a good heuristic
 - first-fail
 - expensive scene

Symmetry-breaking during search

- ▶ Choose a scene to shoot
 - use a good heuristic
 - first-fail
 - expensive scene
- ▶ Consider existing days + 1 new day
 - to label the scene

Symmetry-breaking during search

- ▶ Choose a scene to shoot
 - use a good heuristic
 - first-fail
 - expensive scene
- ▶ Consider existing days + 1 new day
 - to label the scene
- ▶ Advantages
 - break symmetries
 - does not interfere with the search heuristic

Symmetry breaking during search

```
using {  
  while (!bound(shoot)) {  
    int eday = max(-1,maxBound(shoot));  
    selectMin(s in Scenes: !shoot[s].bound())  
      (shoot[s].getSize(),-sum(a in appears[s]) fee[a])  
    tryall(d in 0..eday + 1)  
      shoot[s] = d;  
  }  
}
```

Symmetry breaking during search

```
using {  
  while (!bound(shoot)) {  
    int eday = max(-1,maxBound(shoot));  
    selectMin(s in Scenes: !shoot[s].bound())  
      (shoot[s].getSize(),-sum(a in appears[s]) fee[a])  
    tryall(d in 0..eday + 1)  
      shoot[s] = d;  
  }  
}
```

existing days

Symmetry breaking during search

```
using {  
  while (!bound(shoot)) {  
    int eday = max(-1, maxBound(shoot));  
    selectMin(s in Scenes: !shoot[s].bound())  
      (shoot[s].getSize(), -sum(a in appears[s]) fee[a])  
    tryall(d in 0..eday + 1)  
      shoot[s] = d;  
  }  
}
```

existing days

new day

Symmetry breaking during search

dynamic ordering

```
using {  
  while (!bound(shoot)) {  
    int eday = max(-1, maxBound(shoot));  
    selectMin(s in Scenes: !shoot[s].bound())  
      (shoot[s].getSize(), -sum(a in appears[s]) fee[a])  
    tryall(d in 0..eday + 1)  
      shoot[s] = d;  
  }  
}
```

existing days

new day

Searching in constraint programming

- ▶ variable/value labeling
- ▶ focusing on the objective
- ▶ value/variable labeling
- ▶ domain splitting
- ▶ symmetry breaking during search
- ▶ randomization and restarts

Randomization and restarts

Randomization and restarts

- ▶ Sometimes there is no (obvious) search ordering
 - but there exists some good ones

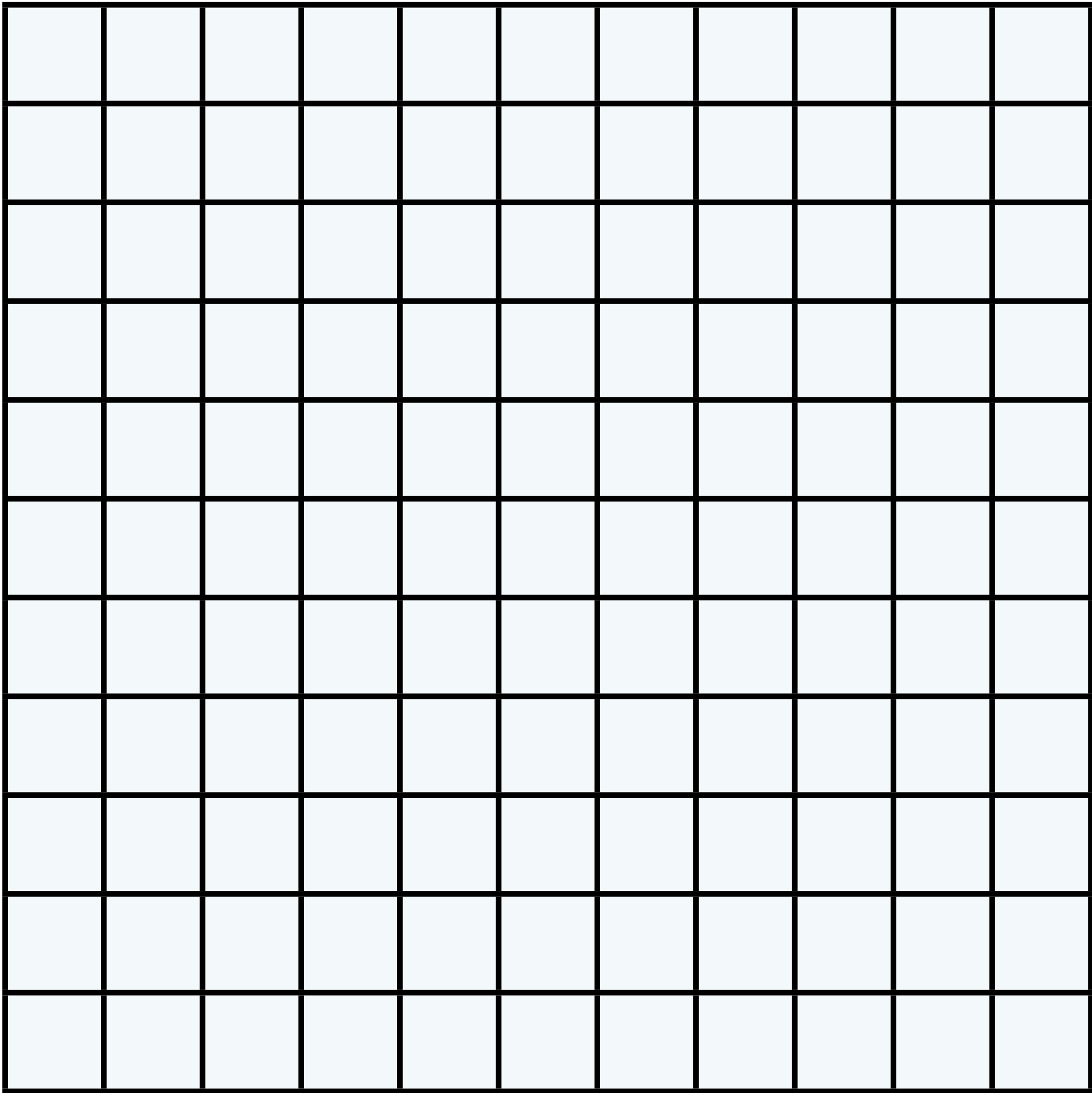
Randomization and restarts

- ▶ Sometimes there is no (obvious) search ordering
 - but there exists some good ones
- ▶ How to find them?
 - brute force
 - randomization and restarts

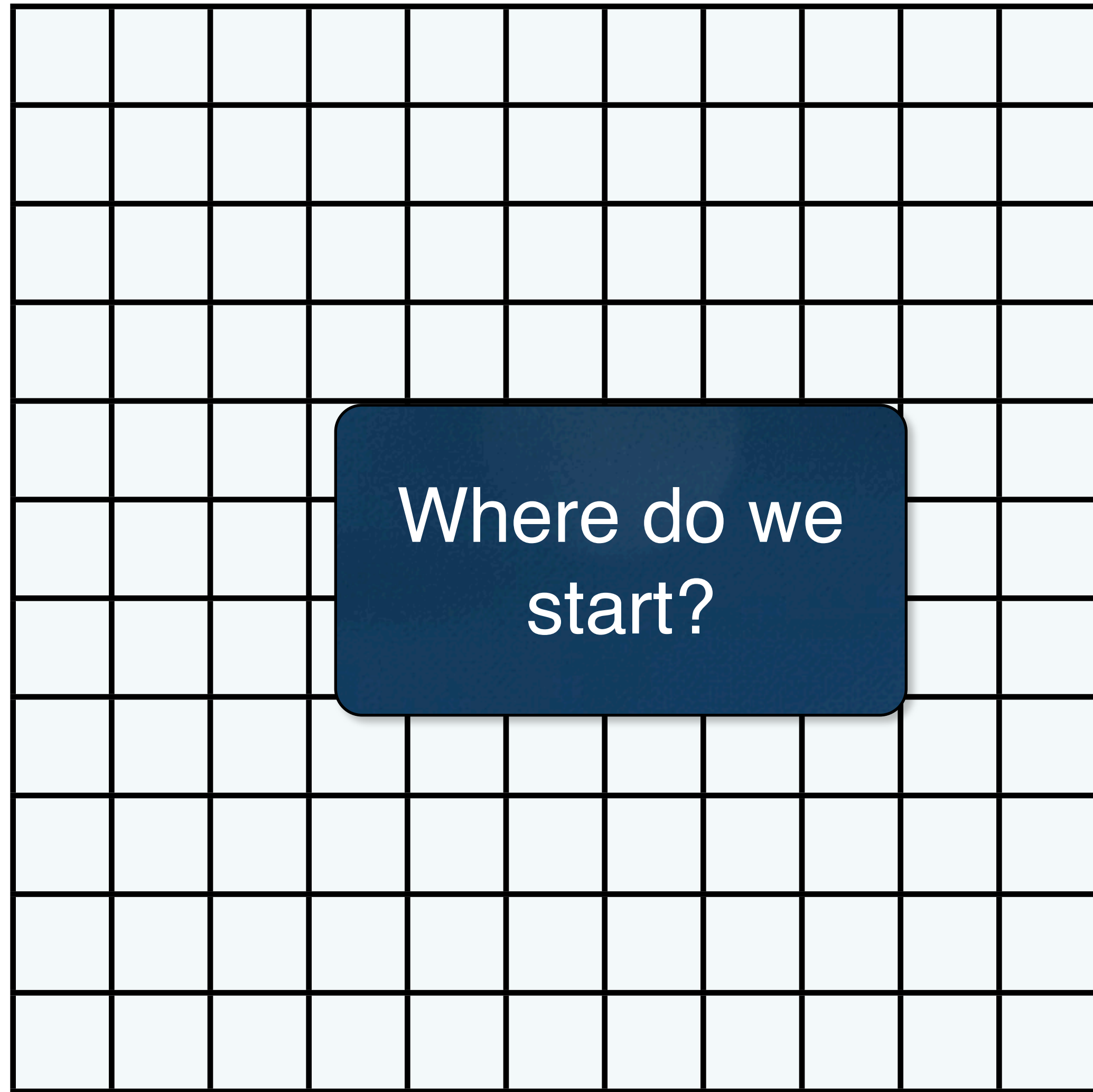
Randomization and restarts

- ▶ Sometimes there is no (obvious) search ordering
 - but there exists some good ones
- ▶ How to find them?
 - brute force
 - randomization and restarts
- ▶ Key idea
 - try a random ordering
 - if no solution is found after some limit, restart the search

The magic square problem



The magic square problem



Randomization and restarts

Randomization and restarts

► Key idea

- apply a heuristic but with randomization
 - e.g., one of the three best variables
- limit the time in the search
- if the limit is reached, restart and possibly increase the limit

The magic square problem

```
using {
  timeLimit = 10;
  repeat {
    limitTime(timeLimit) {
      var{int}[] x = all(i in R,j in R) s[i,j];
      range V = x.getRange();
      while (!bound(x)) {
        selectMin[3] (i in V: !x[i].bound()) (x[i].getSize()) {
          int mid = (x[i].getMin()+x[i].getMax())/2;
          try x[i] <= mid; | x[i] > mid;
        }
      }
    }
  }
  onFailure {
    timeLimit = 1.1 * timeLimit;
  }
}
```