Imperative Programming

Week 5

Mathematical induction

Let P(n) be a logical expression that is dependent on a natural number n.

To prove that P(n) holds for *all* natural numbers n it suffices to show:

- P(0) holds
- For any natural number n: P(n) implies P(n+1)

Example: math. induction

Define: $S(n)=0+1+2+3+\ldots+n$.

Prove that: $S(n) = n*(n+1)/2$.

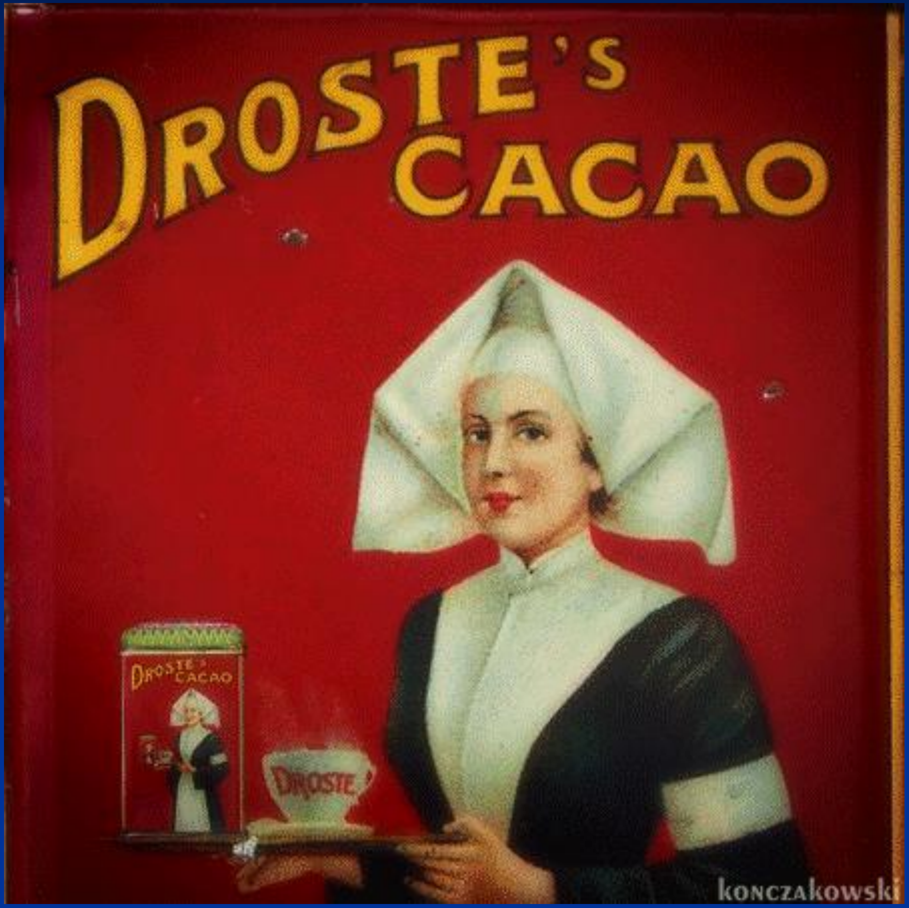
In terms of the previous slide: $P(n)$ == $S(n) = n*(n + 1)/2$

Step 1 $P(0)$: $S(0)=0=0*(0+1)/2$, so $P(0)$ holds.

Step 2 $P(n + 1)$: Assume that $P(n)$ holds. We prove $S(n + 1)=(n + 1)*(n + 2)/2$.

$$
\begin{aligned}
S(n+1) &= 0+1+2+3+\ldots+n+(n+1) \\
&= (0+1+2+3+\ldots+n) + (n+1) \\
&= S(n)+(n+1) \\
&= n*(n+1)/2 + (n+1) \\
&= (n*(n+1) + 2*(n+1))/2 \\
&= ((n+1)*(n+2))/2 \qquad \text{QED.}
\end{aligned}
$$

Recursion is recursion is recursion is recursion …



```
Fac(0)=1

Fac(n)=n*Fac(n-1)
```

Recursion

- **Recursion** = ‘self-reference’
- Recursion is an alternative for repetition (while, for, do-while).
- Often, recursion is a natural and elegant way of solving problems.

Recursion

- Some problems have a recursive character. Recognizing this recursive character usually makes solving them much easier.

- Hardest task: recognize the recursive character
  - solution procedure (argument n)

function call

return

•solution procedure (argument n - 1)

function call

return

•solution procedure(argument n - 2)

return

….

•stop condition

Factorial

`Base case:` `Fac(0)=1`

`Recursive case:` `Fac(n)=n*Fac(n-1)`

Sequence: 1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, ….

```
Fac(4)  = 4*Fac(4-1) = 4*Fac(3)
        = 4*3*Fac(3-1) = 12*Fac(2)
        = 12*2*Fac(2-1) = 24*Fac(1)
        = 24*1*Fac(1-1) = 24*Fac(0)
        = 24*1
        = 24
```

Factorial in C

```
int fac(int n) {
  /* returns n! */
  return (n == 0 ? 1 : n*fac(n-1));
}
```

Exponentiation

$$g^{0} = 1$$

$$g^{m} = g \cdot g^{m-1}$$

```
int power(int g, int m) {
   return (m ==0 ? 1 : g*power(g, m-1));
}
```

Exponentiation (2)

$$g^0 = 1$$
$$g^{2m} = (g^2)^m$$
$$g^{2m+1} = g \cdot g^{2m}$$

Using this recurrence we obtain the answer much faster:

```
int power(int g, int m) {
   if (m==0) {
      return 1;
   }
   if (m%2 == 0) {
      return power(g*g, m/2);
   }
   return g*power(g, m-1);
}
```

Mutual recursion

- If two (or more) functions circularly call each other, then we call this *mutual recursion*.

$f(0) = 1$ and $f(n) = 2*g(n-1)$ (for $n > 0$),
$g(0) = 2$ and $g(n) = f(n-1)$ (for $n > 0$).

$f(0) = 1$ and $f(n) = 2*g(n-1)$ (for $n > 0$),
$g(0) = 2$ and $g(n) = h(n-1)$ (for $n > 0$).
$h(0) = 3$ and $h(n) = f(n/2)$ (for $n > 0$).

Mutual recursion: example

These are silly functions to determine whether their arguments are even or odd. But it nicely shows the concept of mutual recursion.

```
int isOdd(int n); /* forward declaration*/

int isEven(int n) {
  return ((n==0) || (n>0 && isOdd(n-1) || (n<0 && isOdd(n+1)));
}

int isOdd(int n) {
  return ((n==1) || (n>0 && isEven(n-1) || (n<0 && isEven(n+1)));
}
```

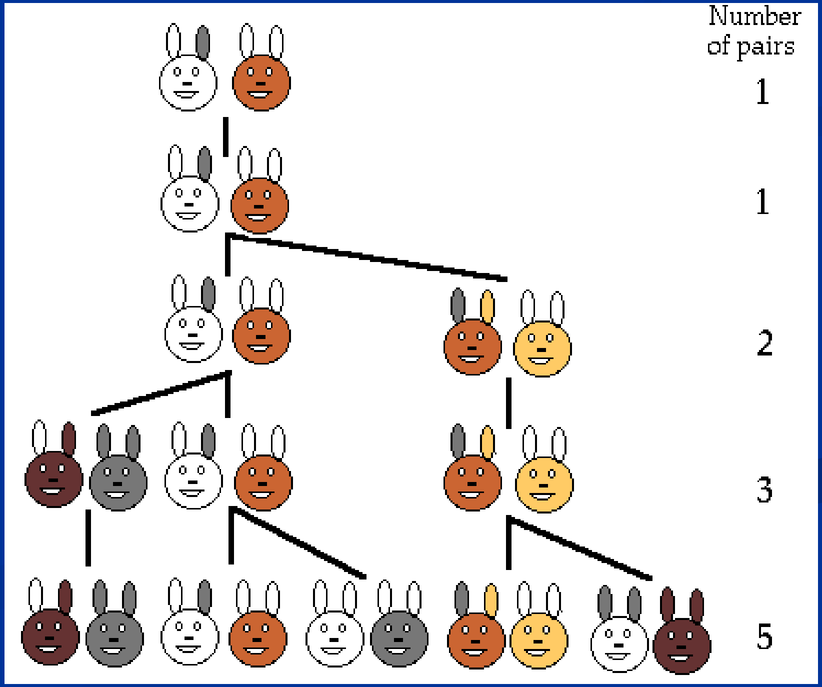
Note that these functions make use of the ‘short circuit evaluation’ of Boolean expressions.

Double recursion: Fibonacci's rabbits

Base cases:	f(0) = 0
	f(1) = 1
Recursive case:	f(n) = f(n-1) + f(n-2)

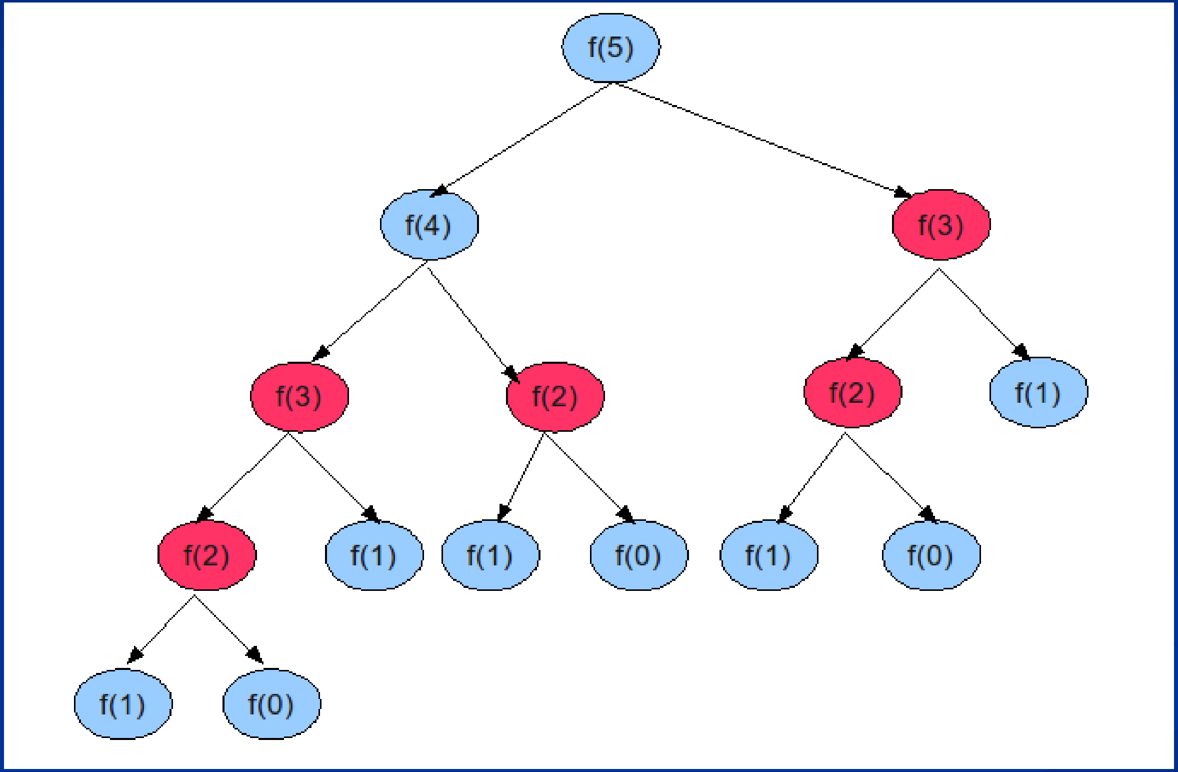
Series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, …

```
int f(int n) {
  if (n <= 1) {
    return n;
  }
  return f(n-1) + f(n-2);
}
```



f(5): Recursion tree

$f(n)=f(n-1) + f(n-2)$



$$f(5)=f(4)+f(3)=f(3)+f(2)+f(3)$$

Memoization / Dynamic pogramming

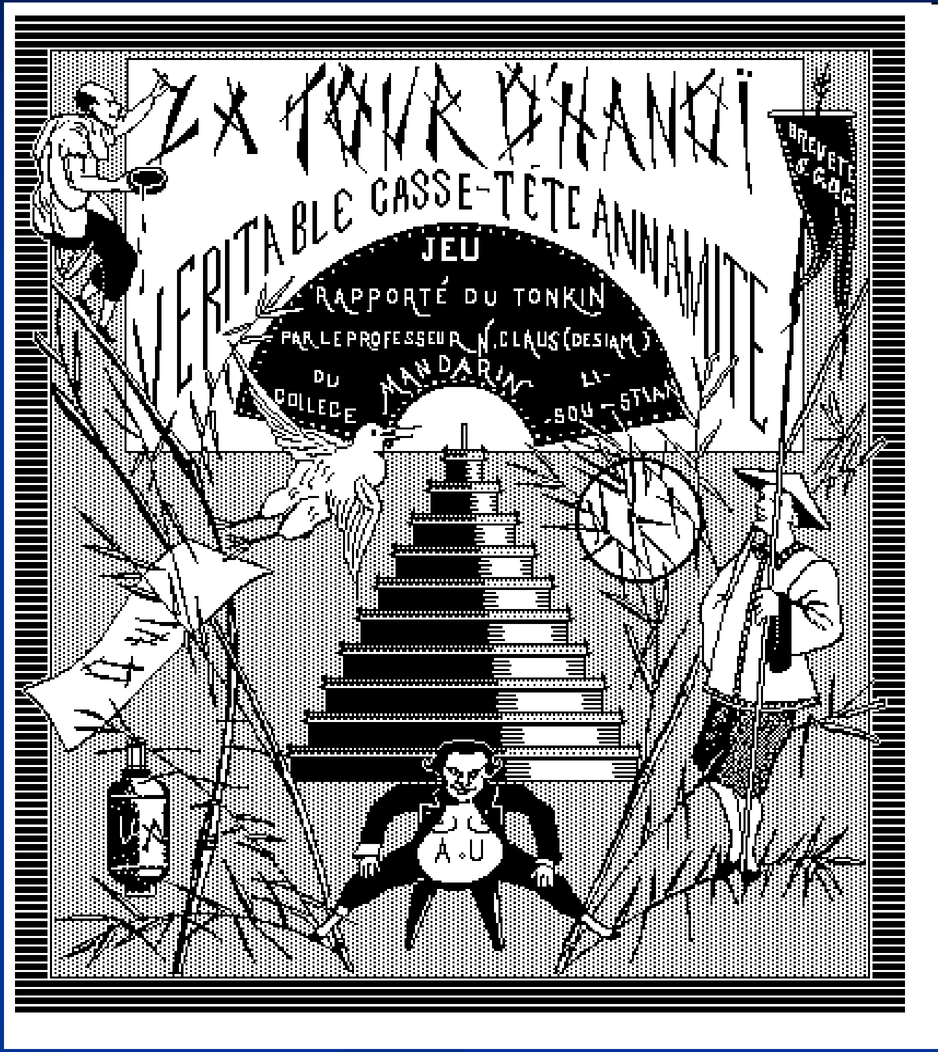
Memoization (a.k.a. **dynamic programming**) is an optimization technique used to speed up a program by storing the results of previous function calls and returning the stored (cached) result when the same function call takes place.

Dynamic programming: fast recursive Fib-function

```
int fibMem[47];  /* f(46)=1836311903, f(47) > maxint */

int fib(int n) {
  /* base cases */
  if (n <= 1) {
    return n;
  }

  /* recursive case */
  if (fibMem[n] == 0) {
    /* fib(n) has not been computed before */
    fibMem[n] = fib(n-1) + fib(n-2);
  }

  return fibMem[n];
}
```

The Tower of Hanoi

- ‘Invented’ by the French mathematician, Edouard Lucas and sold as a game in 1883


LA TOUR D'HANOÏ
VERITABLE CASSE-TÊTE ANNAMITE
JEU
RAPPORTÉ DU TONKIN
PAR LE PROFESSEUR N. CLAUS (DE SIAM)
DU COLLÈGE MANDARIN LI-SOU-STIAN
BREVETÉ S.G.D.G.
A U

The Tower of Hanoi

Inspired by a legend about an Indian temple in which there is a room with three rods surrounded by 64 golden discs. Brahmin priests, acting out the command of an ancient prophecy, are busy moving these discs, in accordance with the rules of the Brahma. According to the legend, when the last move of the puzzle will be completed, the world will end.

Rules of Brahma:

- Only one disc is moved at a time.
- No disc may be placed on top of a smaller disc.

R.O.B.O.T. Comics
WWW.WILLOWGARAGE.COM
JORGE CHAM 2009

"THEY ALL SAY THEY'RE AGNOSTIC, UNTIL IT'S TIME FOR DIAGNOSTICS."

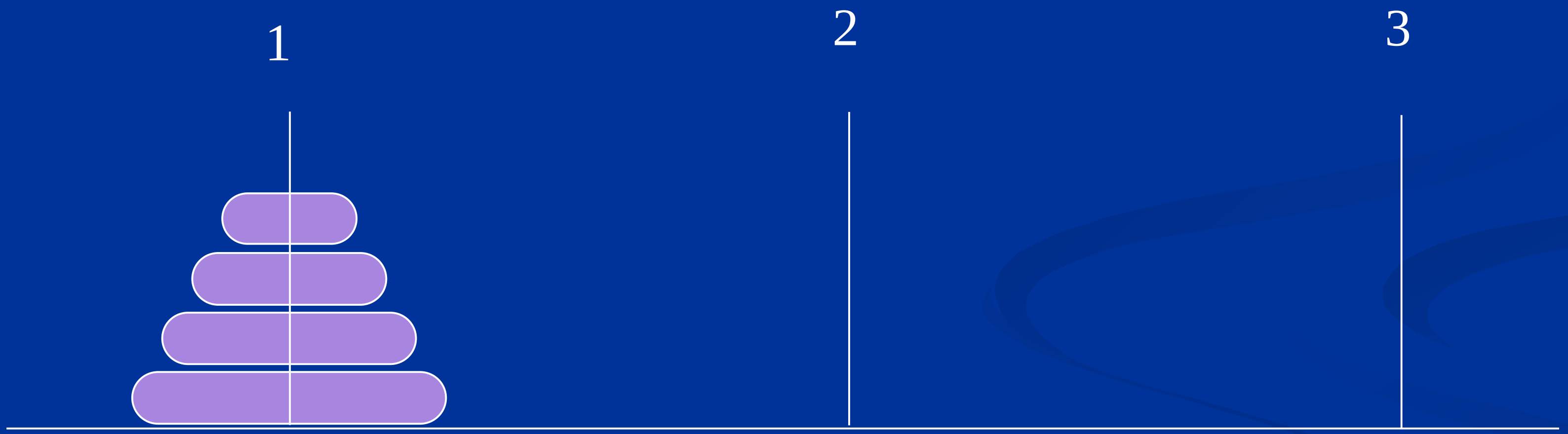
The Tower of Hanoi

- Of course, it is useful information to know when the world ends.
    - Is it before or after my (resit mid)exam?

- How many moves are needed to solve the puzzle with 64 discs?

Recursive property ?

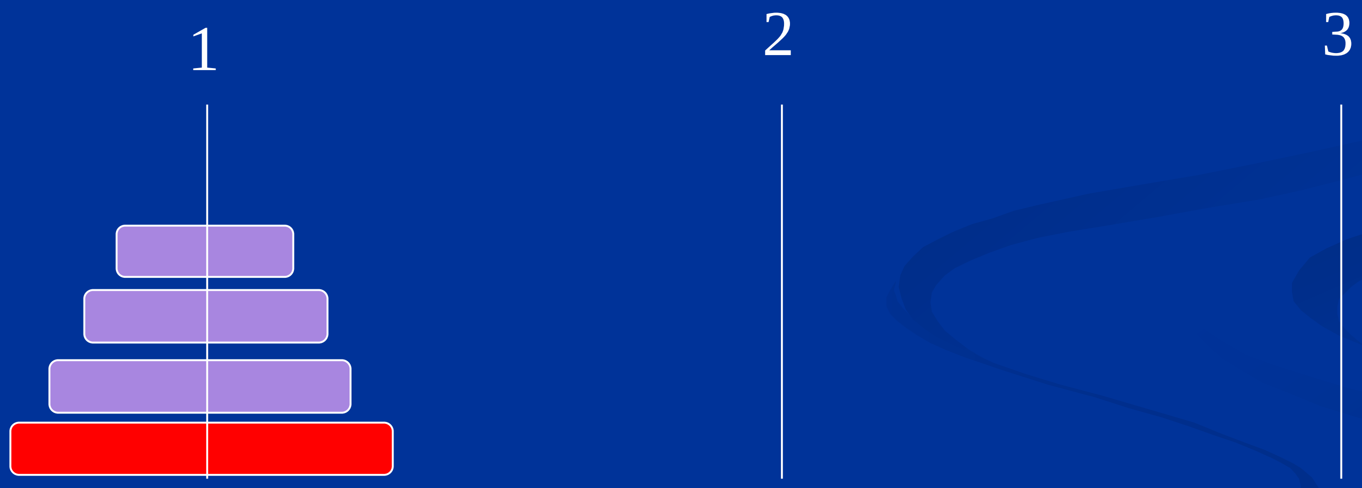
- Which of these 4 discs should be placed first on the target rod 2?



$\longrightarrow$ How can we establish this?

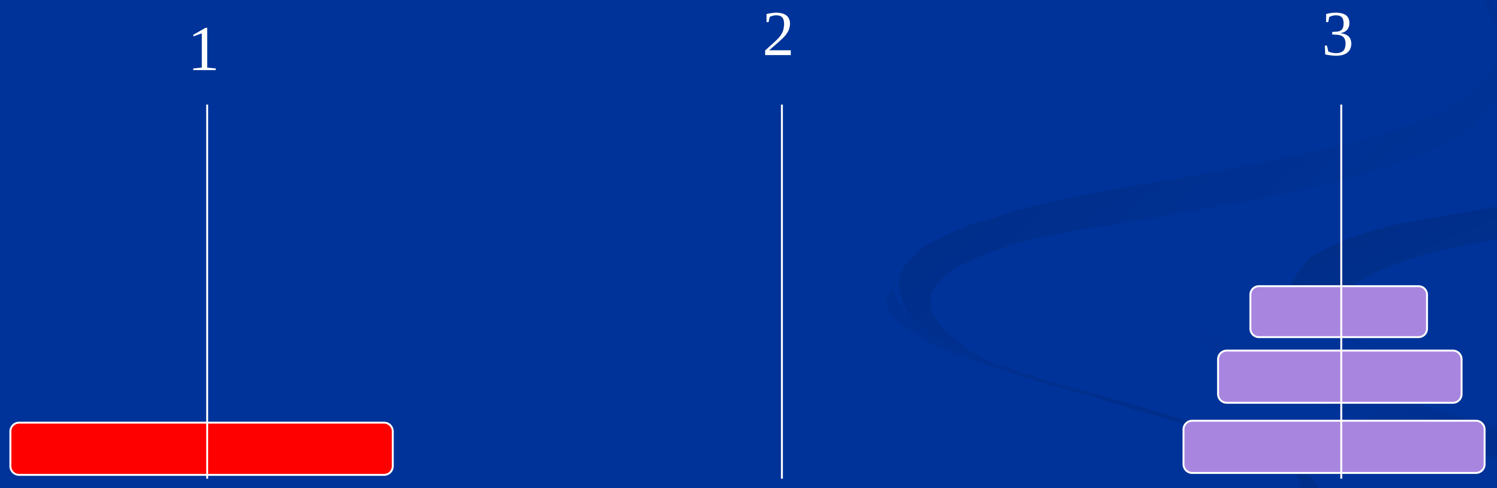
Recursive step

- Step 1: move upper 3 discs to rod 3.
  - How? Recursion!



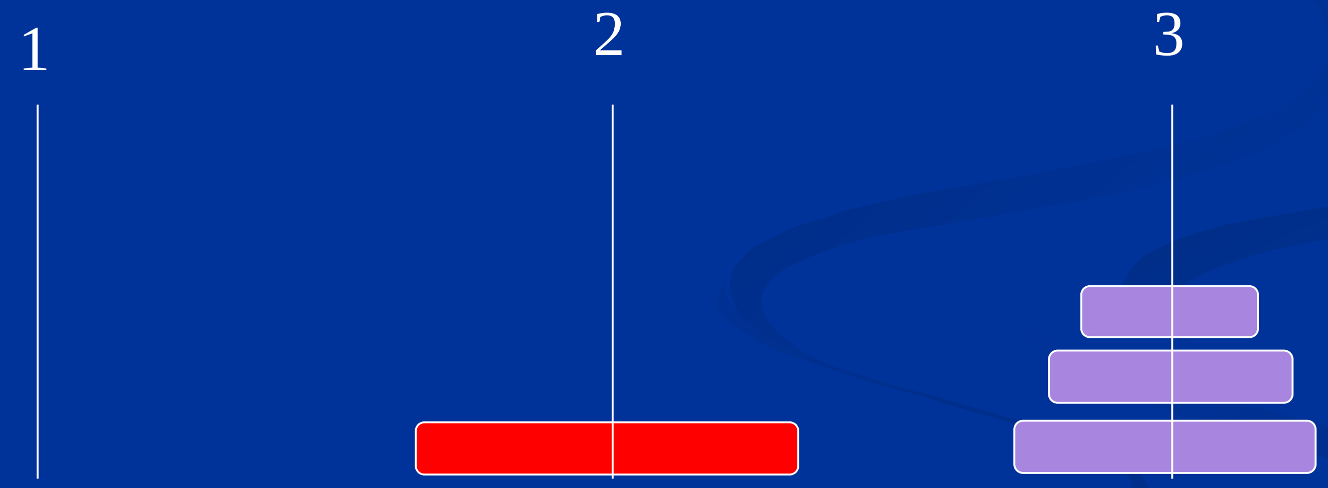
Make a step

- Step 2 : move the largest disc to its destination



Recursive step

- Step 3 : **move upper 3 discs to rod 2.**
    - **How? Recursion!**

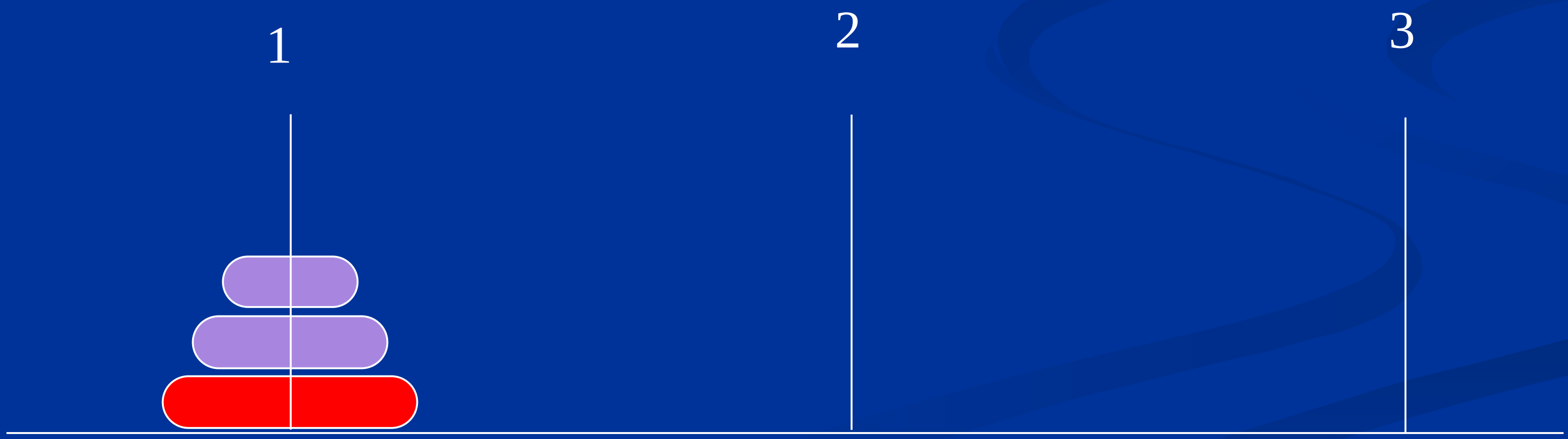


Recursive steps

- Which problems remain?
  - **<u>Step 1</u>** Move 3 discs from rod 1 to rod 3.
  - **<u>Step 3</u>** Move 3 discs from rod 3 to rod 2.

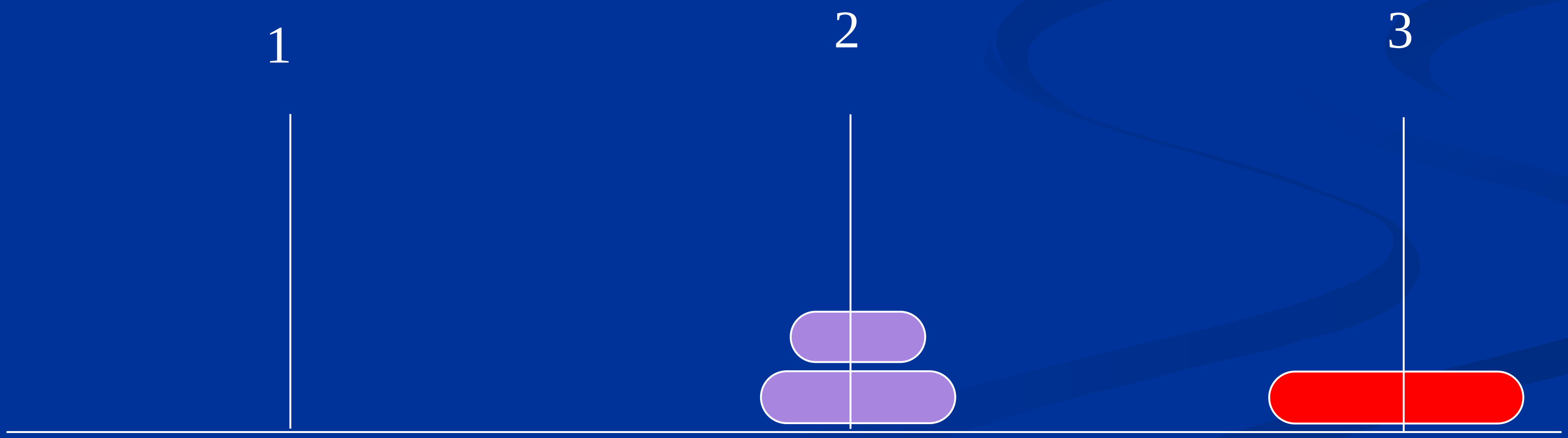
Recursive step 1 (rod 1 -> 3)

- Step 1.1: move upper 2 discs to rod 2.
  - How? Recursion!
- Step 1.2 : move the largest disc to its destination



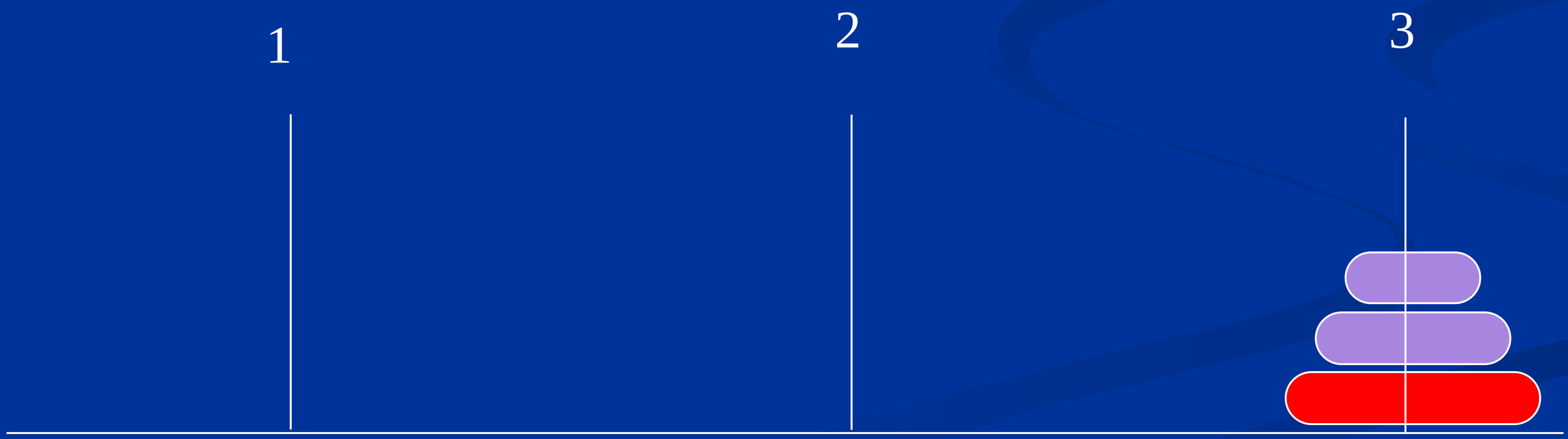
Recursive step 1 (rod 1 -> 3)

- Step 1.3 : move upper 2 discs to rod 3.
  - How? Recursion!



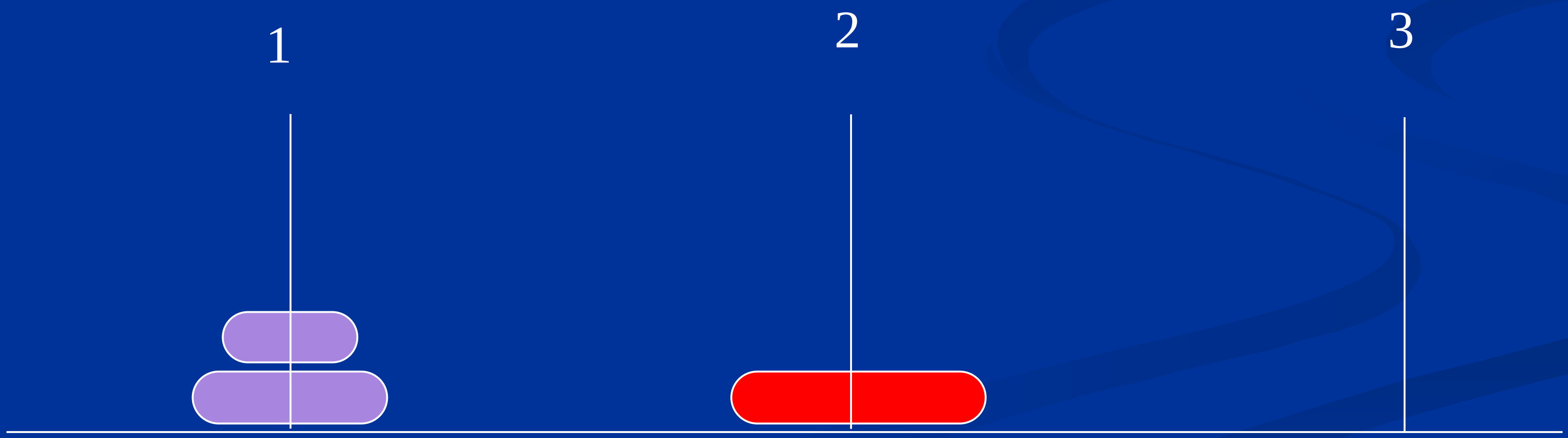
Recursive step 3 (rod 3 -> 2)

- Step 3.1 : move upper 2 discs to rod 1.
  - How? Recursion!
- Step 3.2 : move the largest disc to its destination.



Recursive step 1 (rod 3 -> 2)

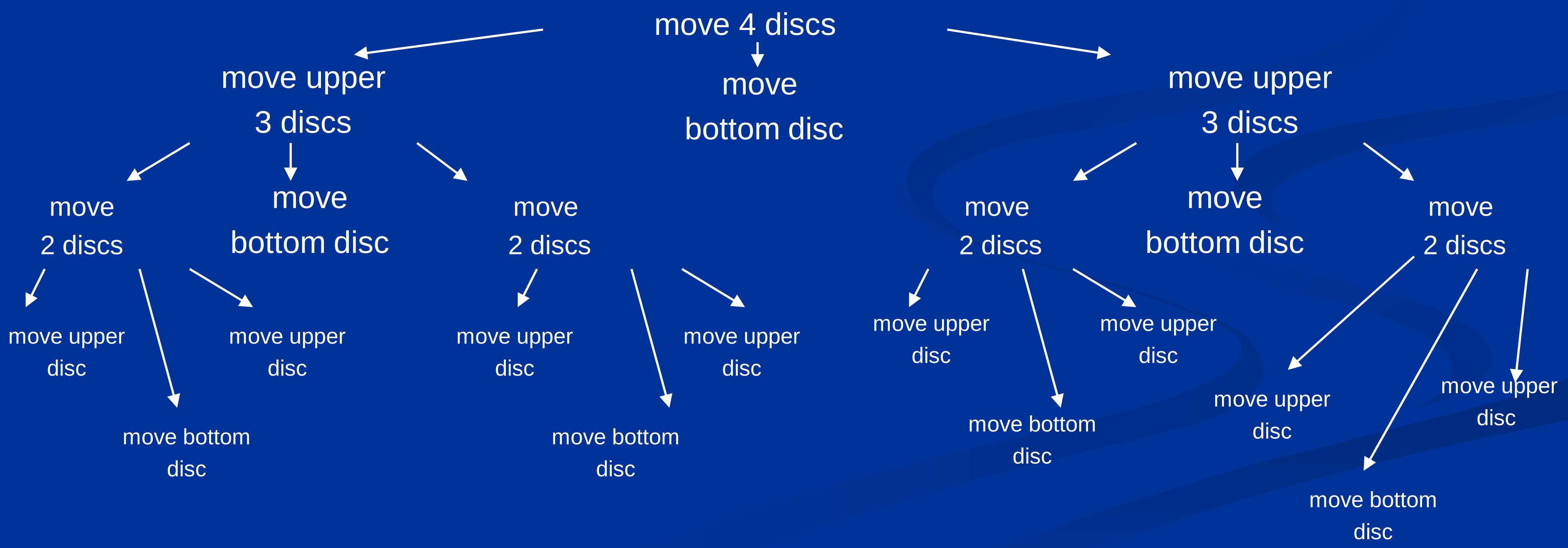
- Step 2.3 : move upper 2 discs to rod 2.



Recursive strategy

- We reduced to problem of moving n discs to moving n-1 discs twice, i.e. two smaller instances of the orginal problem!

  ➔ RECURSION



Resume: Tower of Hanoi

1. Assume that we know how to solve the problem for a pyramid of n discs. It takes $T(n)$ moves.

2. Then we know how to solve the problem for a pyramid of $n + 1$ discs. We need three steps for this:

   - step 1:Move the upper n discs to another rod. This takes $T(n)$ moves.

   - step 2:Move the bottom discs to the final destination. This takes 1 move.

   - step 3:Move the other n discs to their final destination. This again takes $T(n)$ moves.

   - B.T.W. You can prove (but we won't) that this solution strategy is optimal!

Resume: Tower of Hanoi

- So, we find the recurrence:
  - T(n+1)=T(n) + 1 + T(n)=2T(n)+1

- Clearly: T(1)=1
- Or even better: T(0)=0
- Sequence: 0, 1, 3, 7, 15, 31, 63, …

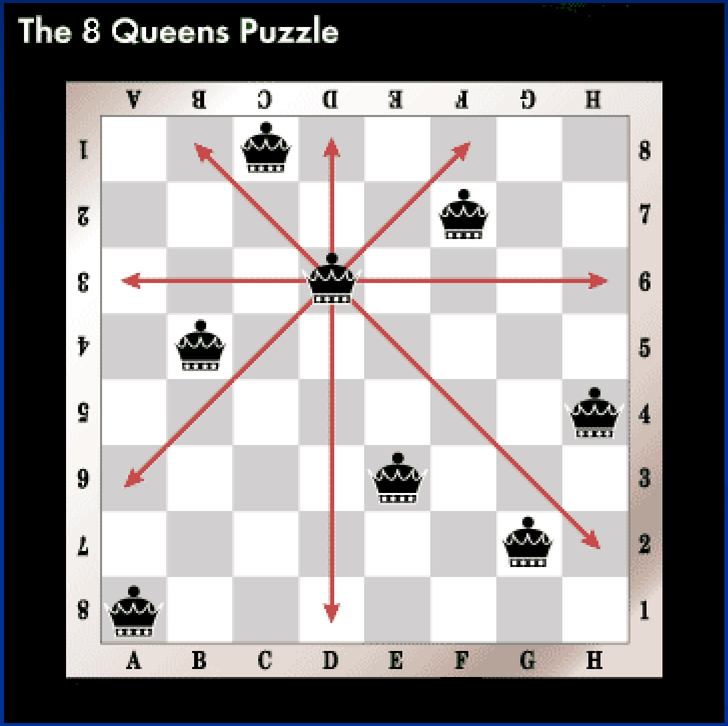
- We observe: $T(n) = 2^n - 1$ (proof math.ind.)

- So, $T(64) = 2^{64} - 1 = 18446744073709551615$
- Really fast priests may be able to perform one move per second, which means that the world will end in approximately 585,000,000,000 year!
  - My personal advice: Do not worry about it. Study for the (resit mid)exam!

Tower of Hanoi- program

```
#include <stdio.h>
#include <stdlib.h>

void hanoi(int numberOfDiscs, int src, int dest) {
  /* base case: numberOfDiscs == 0, there is nothing to do */
  if (numberOfDiscs != 0) {
    /* recursive case */
    int via = 6 - src - dest;
    hanoi(numberOfDiscs - 1, src, via);
    printf ("Move disc from rod %d to rod %d.\n", src, dest);
    hanoi(numberOfDiscs - 1, via, dest);
  }
}

int main(int argc, char *argv[]) {
  hanoi(atoi(argv[1]), 1, 3);
  return 0;
}
```

Eight queens problem



Write a program that prints all board configurations with 8 queens such that no queen attacks another queen.

Eight queens problem

- Naive solution: generate all possible configurations, and check validity
  - Huge number of configurations!
  - $\frac{64!}{8!(64-8)!} = \frac{64*63*62*61*60*59*58*57}{40320} = 4{,}426{,}165{,}368$
- But, most of these generations need not be generated in the first place!
  - Example: place the first queen at A1, and the second at B2. These queens are in conflict, whatever the locations of the other queens are!

Backtracking

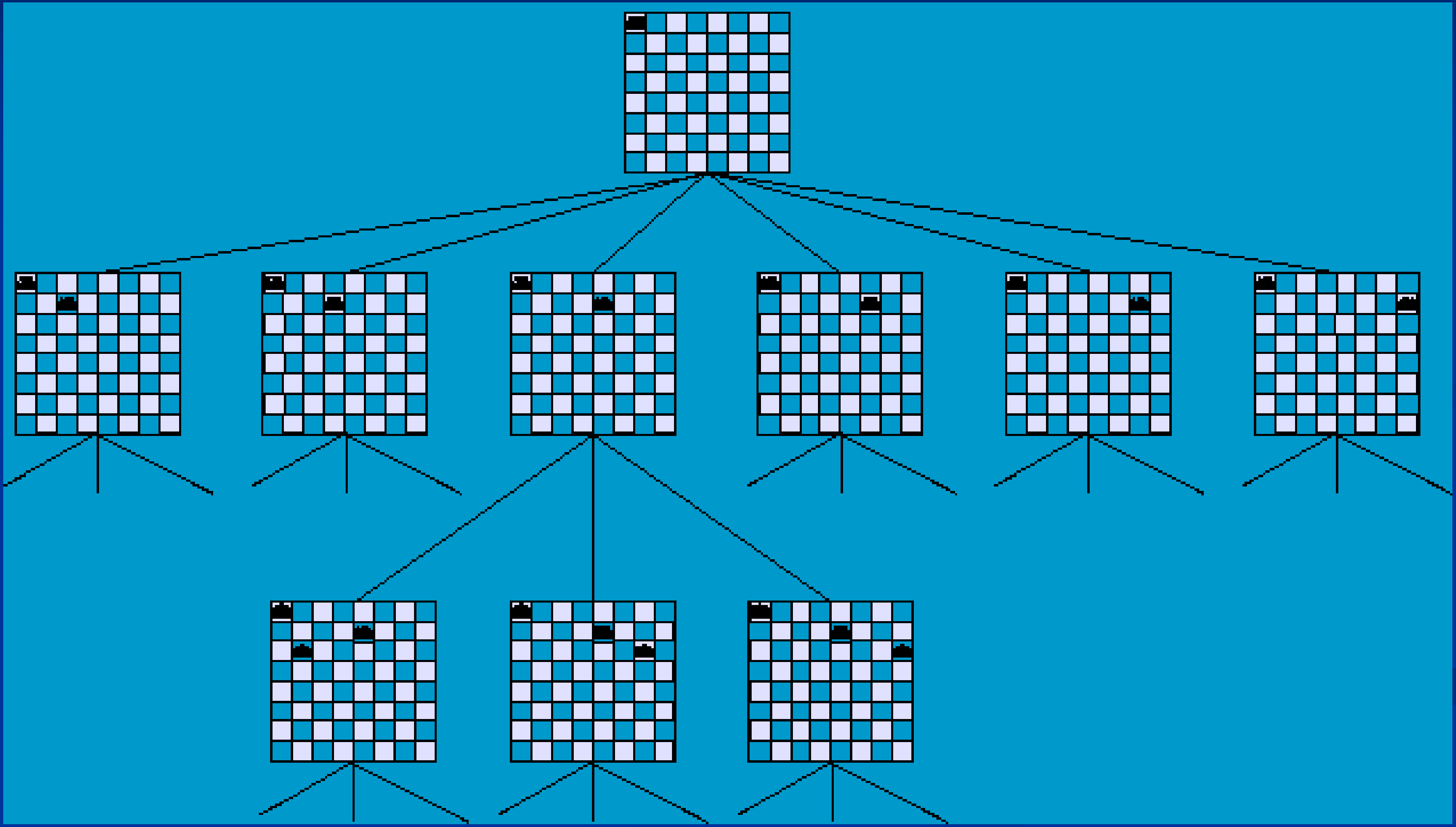
- ‘Solving a problem by searching’ stategy.
- Follow a ‘route’ in the ‘direction’ of a solution until the solutions is found, or until we cannot go on (we get stuck).
  - In the latter case, we backtrack to the last decision state, and make at that point another decision (and move forward again).

Eight queens problem

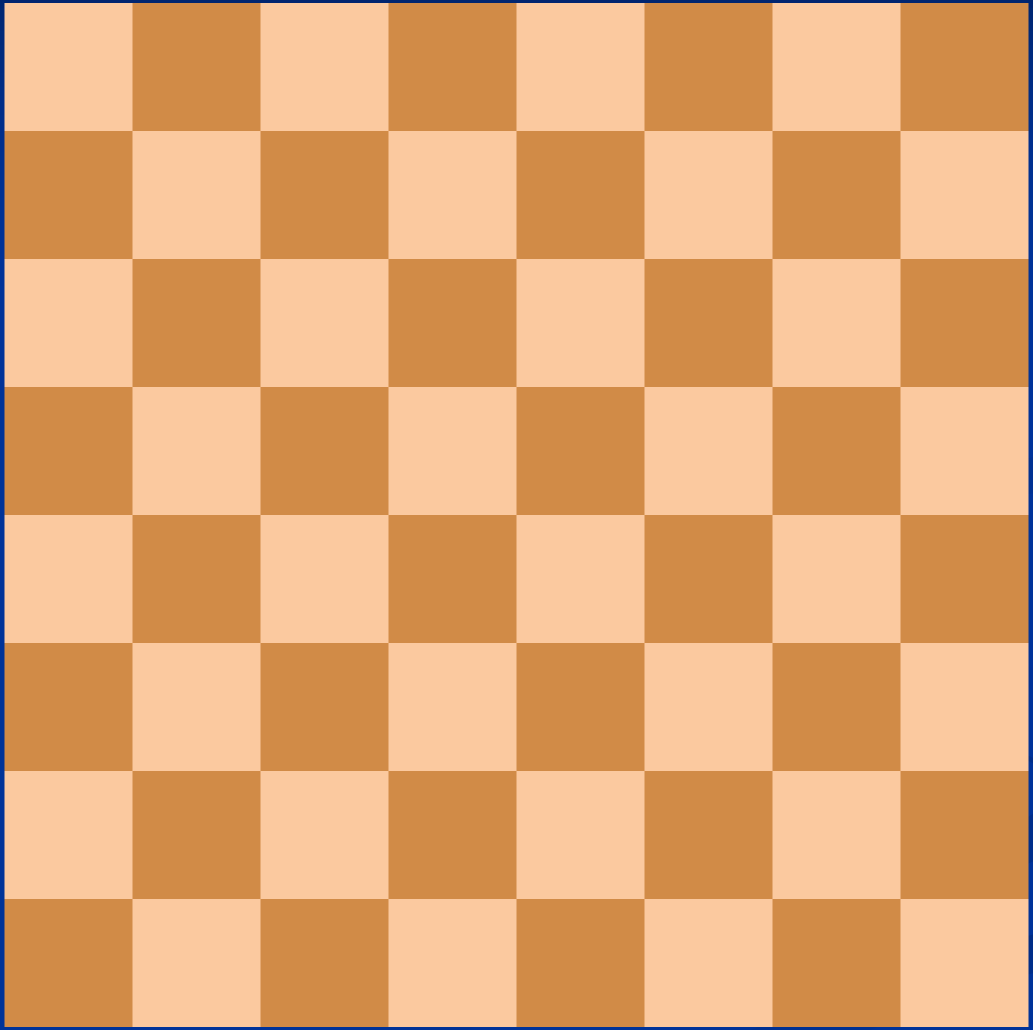
- Problem reformulation: *Place on each row a queen such that no two queens share the same column, or diagonal.*

- Recursive backtracking strategy:
  - In the nth recursive call we try to place a queen in row n.
  - In each recursive step n, we know the locations of the previously placed $n-1$ queens.
  - If we find a suitable column for the queen in row n, place it and recurse with row $n+1$.
  - If all possible positions for the nth queen have been tried, we backtrack and try to make other decisions for one or more queens k (with $k < n$).
  - This method inspects less than 8!=8*7*6*5*4*3*2= 40320 configurations (about 100,000 times better than the naive method).

Eight queens problem



Eight queens problem



Eight queens problem

- In each column we need to place exactly one queen.
- So, we can represent the chess board as follows:

```
int pos[8];
```

Interpretation: `pos[n]==c` means that the queen in row n is in column c, i.e. its location is (n,pos[n]).

Eight queens problem

The chosen board representation already ensures that we cannot place two queens in the same row.

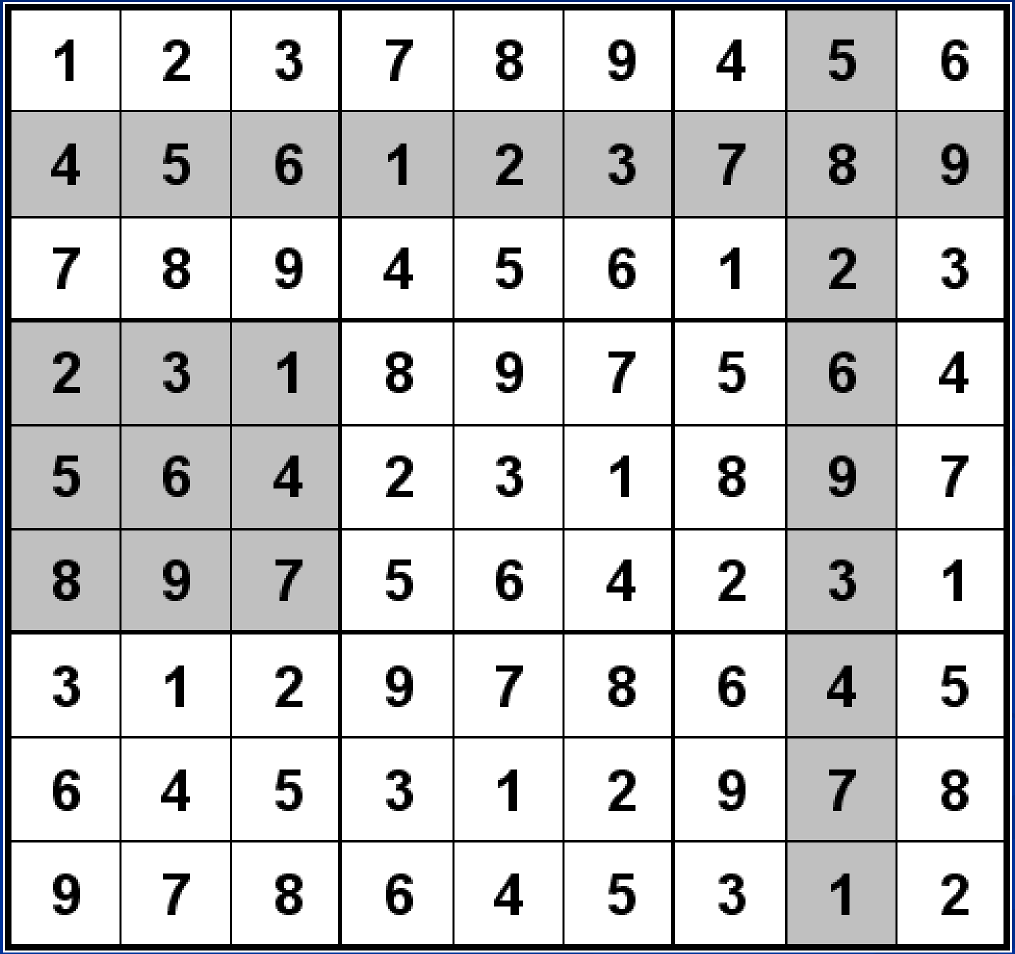
If a queen is placed at position (r, c), then she 'attacks':

- **the row** r
- **the column** c
- **the ascending** diagonal through (r,c), i.e. c - r = constant.
- **the descending** diagonal through (r,c), i.e. c + r = constant.

Eight queens problem: program

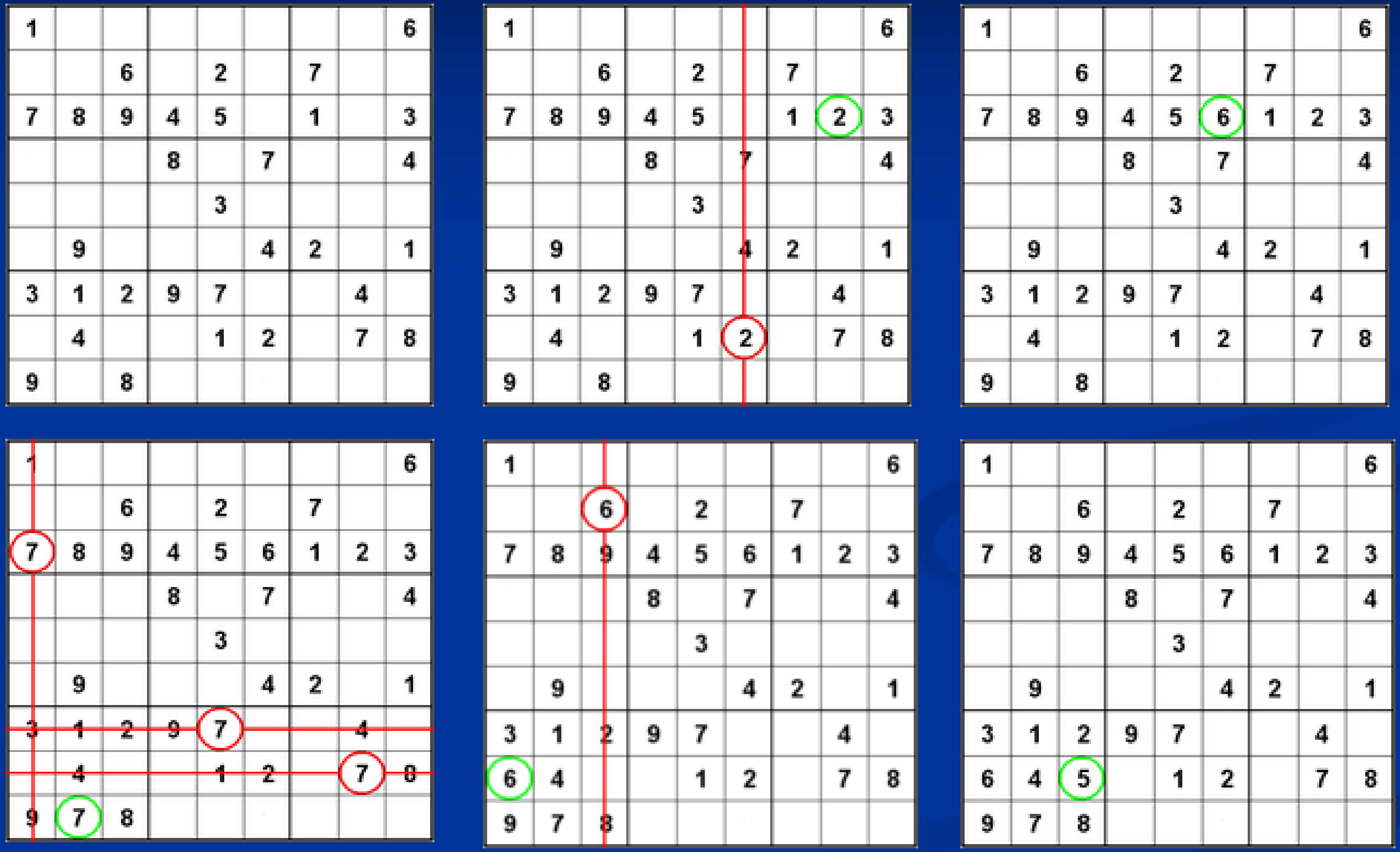
```
#define ABS(a) ((a)<0 ? (-(a)) : (a))

void placeQueen(int row, int pos[8]) {
  /* base case */
  if (row == 8) {
    printPosition(pos);
  } else {
    /* recursive case */
    int column;
    for (column=0; column < 8; column++) {
      int r;
      for (r=0; r < row; r++) { /* loop over previous placed queens */
        if ((pos[r] == column)  /* queen in this column? */
            || (ABS(pos[r]-column) == row-r))  /* queen in this diagonal? */ {
          break;
        }
      }
      if (r==row) { /* square is not attacked => place queen */
        pos[row] = column;
        placeQueen(row+1, pos);
      }
    }
  }
}

int main(int argc, char *argv[]) {
  int pos[8];
  placeQueen(0, pos);
  return 0;
}
```

Sudoku



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

Solving a sudoku by hand



Sudoku: computer solution strategy

- For some empty square, choose a digit.

- Check whether this digit is allowed by the rules.
  - If yes, fill it in.
  - We now have a new puzzle with fewer empty squares.
    - So, a smaller instance of the original problem! Recursion!
    - Detail: smaller solution might not be solvable: backtracking.

  - If no, choose another digit. If no digits ‘fits’, we are stuck.
    - Empty the square again, and rely on backtracking.

```
void solveSudoku(int row, int column, int sudoku[9][9]) {
  int digit, r, c;
  /* base case */
  if (row == 9) {
    printSudoku(sudoku);
    return;
  }
  /* recursive case: compute next square (r,c) */
  if (column < 8) {
    /* Next square is (row,column+1) */
    r = row;
    c = column+1;
  } else {
    /* Next square is (row+1,0) */
    r = row+1;
    c = 0;
  }
  if (sudoku[row][column] != 0) {
    /* There is already a digit here, continue */
    solveSudoku(r, c, sudoku);
  } else { /* Empty square. Try all possible digits */
    for (digit=1; digit<10; digit++) {
      if (digitPossible(row, column, digit, sudoku)) {
        sudoku[row][column] = digit; /* fill in digit, and recurse */
        solveSudoku(r, c, sudoku);
        sudoku[row][column] = 0; /* clear square */
      }
    }
  }
}
```

```c
#define FALSE 0
#define TRUE 1

int digitPossible(int row, int column, int digit, int sudoku[9][9]) {
  int i, r, r0, r1, c, c0, c1;
  /* digit in same row/column? */
  for (i=0; i < 9; i++) {
    if ((sudoku[row][i] == digit) || (sudoku[i][column] == digit)) {
      return FALSE;
    }
  }
  /* digit in same 3x3 block? */
  r0 = row - row%3;
  r1 = r0 + 3;
  c0 = column - column%3;
  c1 = c0 + 3;
  for (r=r0; r<r1; r++) {
    for (c=c0; c<c1; c++) {
      if (sudoku[r][c] == digit) {
        return FALSE;
      }
    }
  }
  return TRUE;
}
```

Sudoku: main program

```
int main(int argc, char *argv[]) {
  int sudoku[9][9];
  readSudoku(sudoku);
  printSudoku(sudoku);
  solveSudoku(0, 0, sudoku);
  return 0;
}
```

As an aside: How many sudokus?

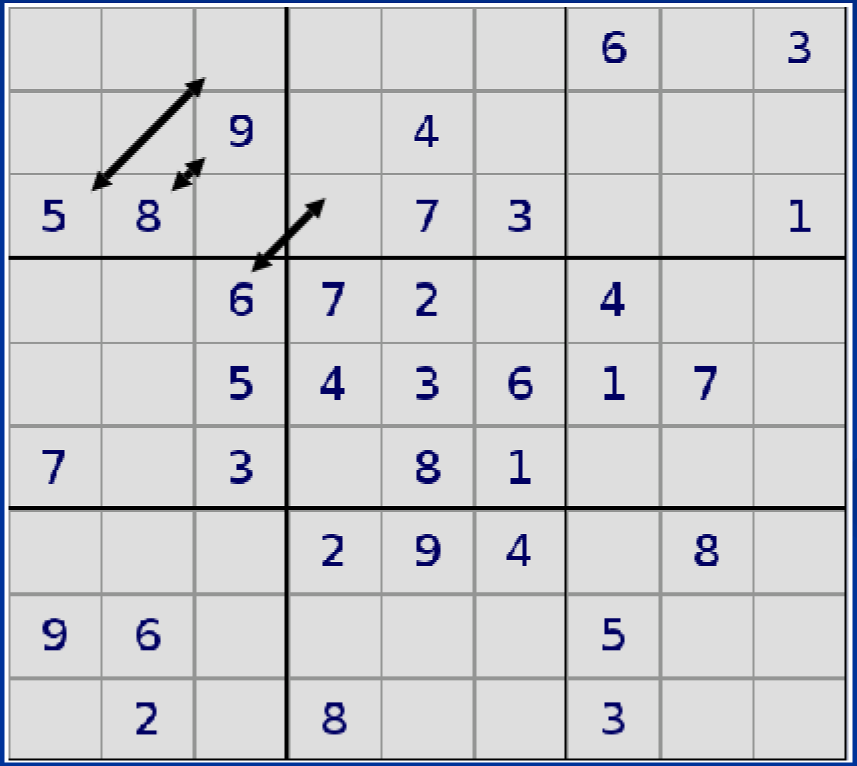
- Felgenhauer and Jarvis were the first to compute the number of valid solved (i.e. completely filled in) sudokus.
- For a 9 x 9-sudoku the answer is 6,670,903,752,021,072,936,960.
- This number equals $9! * 72^2 * 27^7 * 27{,}704{,}267{,}971$.
- Actually, there are ‘only’ about 5.5 million unique sudokus due to the existence of isomorphisms.

Isomorphisms

For each sudoku there are $2 * 9! * 6^8$= 1218998108160 isomorphisms.

- The factor 9! is easy to see.
  - Simply, swap the role of digits!

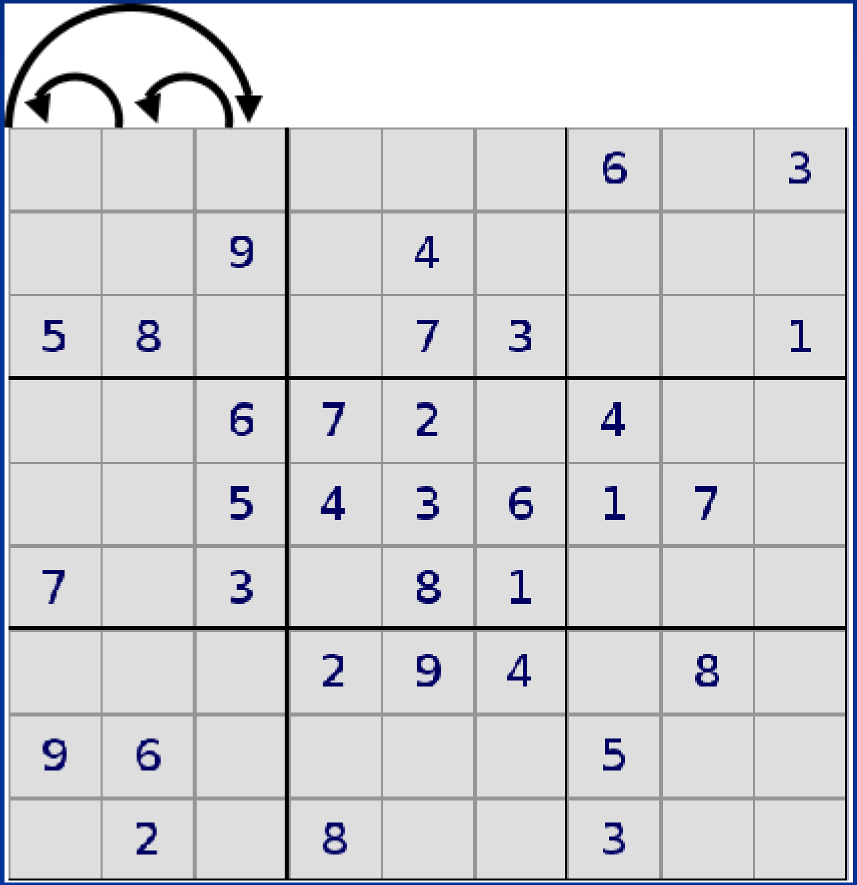
Isomorphisms: transposition



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

2 possibilities

Isomorphisms: swap rows/columns

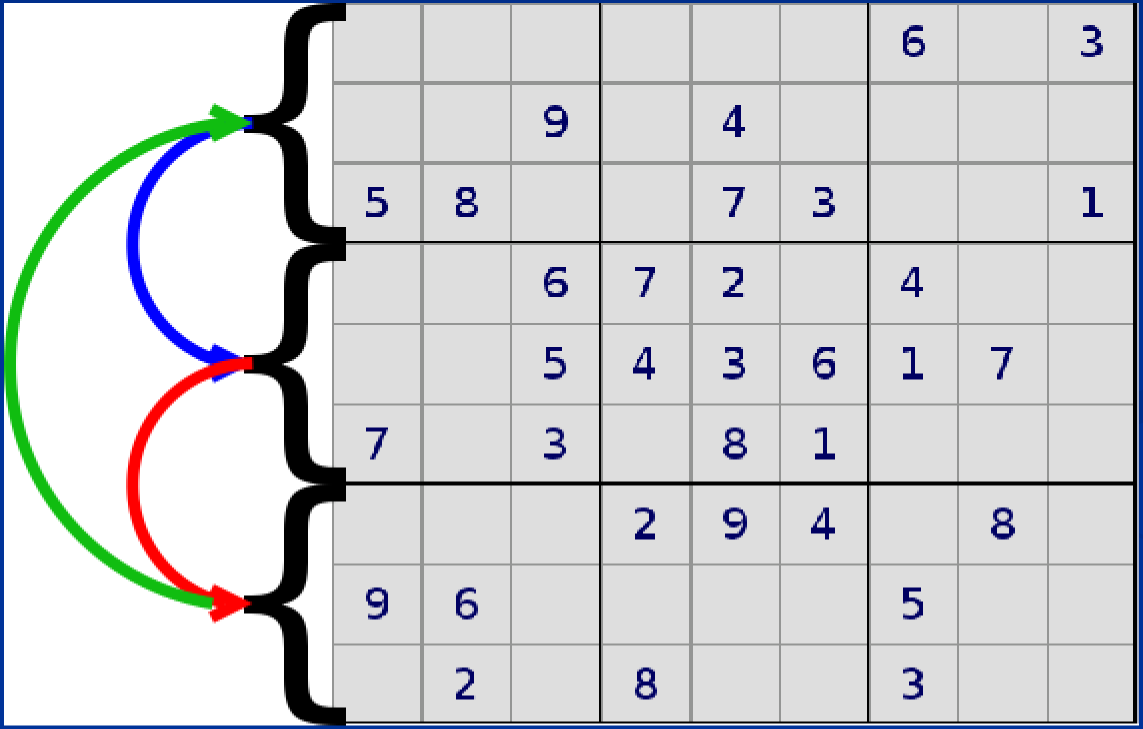


						6		3
		9		4				
5	8			7	3			1
		6	7	2		4		
		5	4	3	6	1	7	
7		3		8	1			
			2	9	4		8	
9	6					5		
	2		8			3		

Within a block you can swap rows/columns. This yields 3!=6 possibilities.

In total $6^6 = 46656$ for the complete sudoku.

Isomorphisms: swap blocks



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

You can swap blocks as well.

This yields 36 possibilities.

End week 5