

Finding all possibilities

— T.09 —

Summary:

- Exhaustive search
- Depth-first search
- Breadth-first search
- Traversing a binary tree in depth-first order and in breadth-first order
- Backtracking
- Pruning
- An example: a chessboard problem
- Two extra examples (sudoku and klotski)

Recommended bibliography for this lecture:

- **Introduction to Algorithms**, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, third edition, The MIT press, 2009.
- **The Algorithm Design Manual**, Steven S. Skiena, second edition, Springer, 2008.

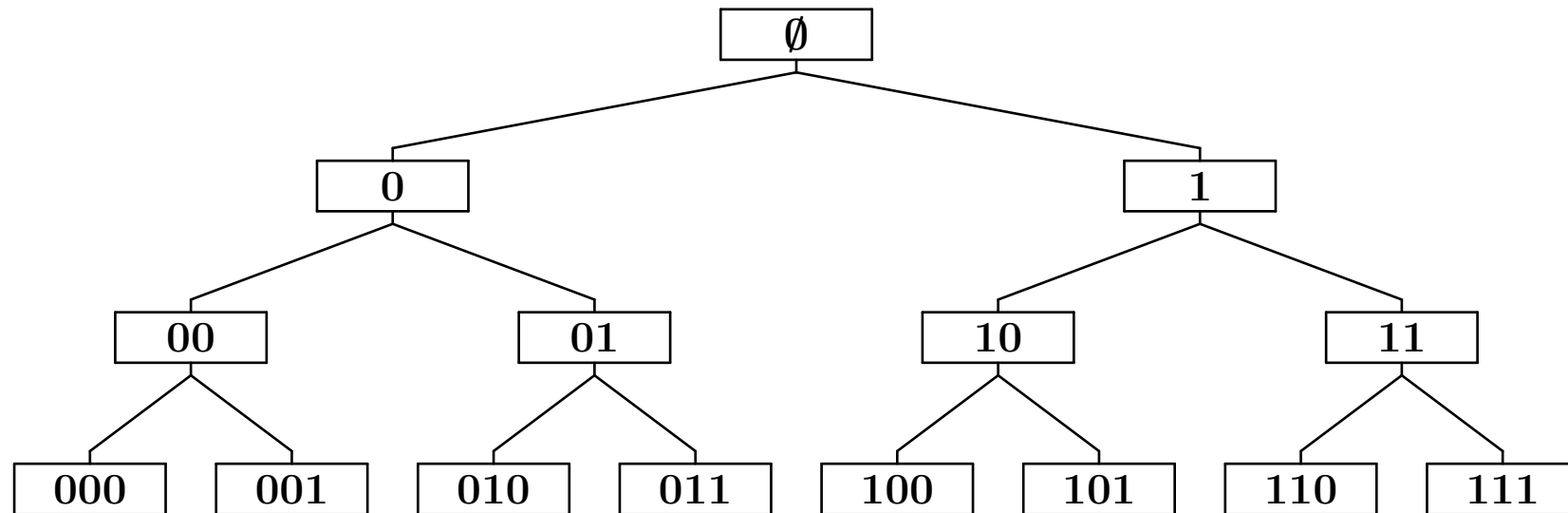
Exhaustive search

Many problems amenable to be solved by a computer require generating all possible configurations of a potential solution and either

- finding one that actually solves the problem or
- finding, among all that solve the problem, the “best one.”

The potential solutions are usually constructed in an incremental way, piece by piece. When that happens it is possible to depict the generation process as building a so-called search tree; adding a node to the tree corresponds to adding another piece to the potential solution. For example, while attempting to solve a sudoku, adding a piece might be placing a specific number in a specific empty square.

We will illustrate the two main strategies of generation of a search tree using the following binary search tree:



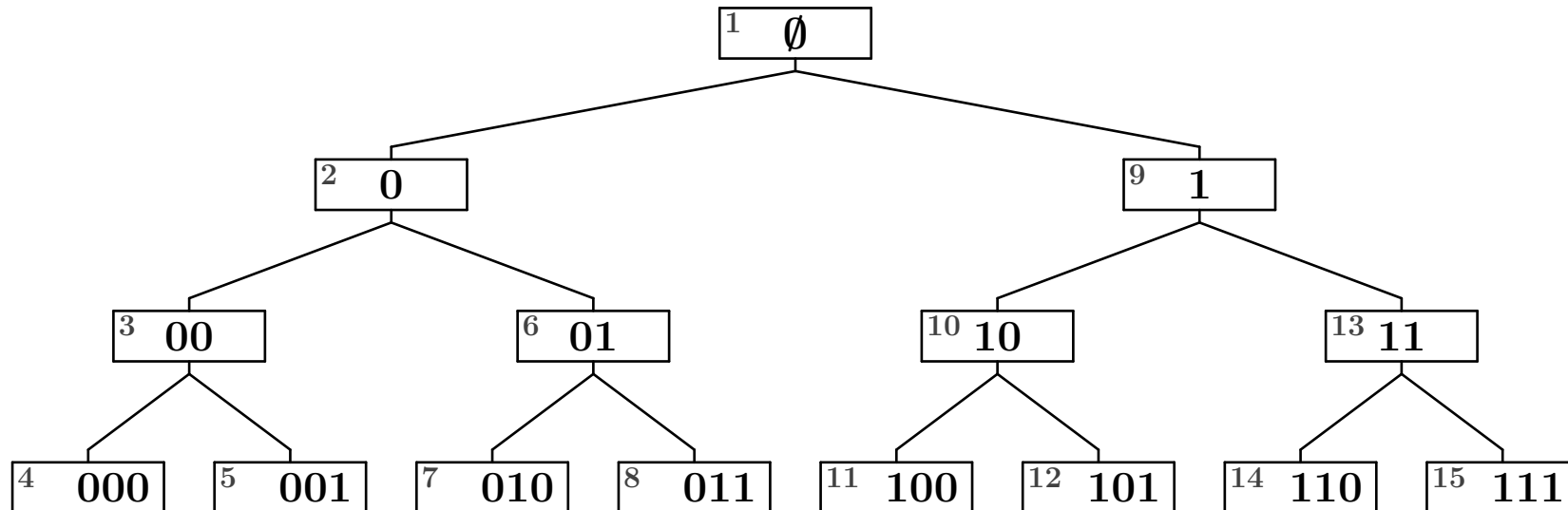
(In this case we may think that the root of the tree corresponds to an empty initial configuration and that at each level we decide to append either a 0 or a 1 to the configuration.)

Depth-first search

The way the search tree is constructed can make a big difference in the execution time of the exhaustive search. For example, placing a piece early in a position that invalidates any possibility of a solution is obviously a bad choice. Deciding how to build the search tree is usually one of the most critical tasks one faces when performing an exhaustive search.

In a **depth-first search** one tries to go as deep as possible as soon as possible. It is the most common strategy used in an exhaustive search, because it can be done quite easily (using recursion, for example), and does not require much memory to manage the search.

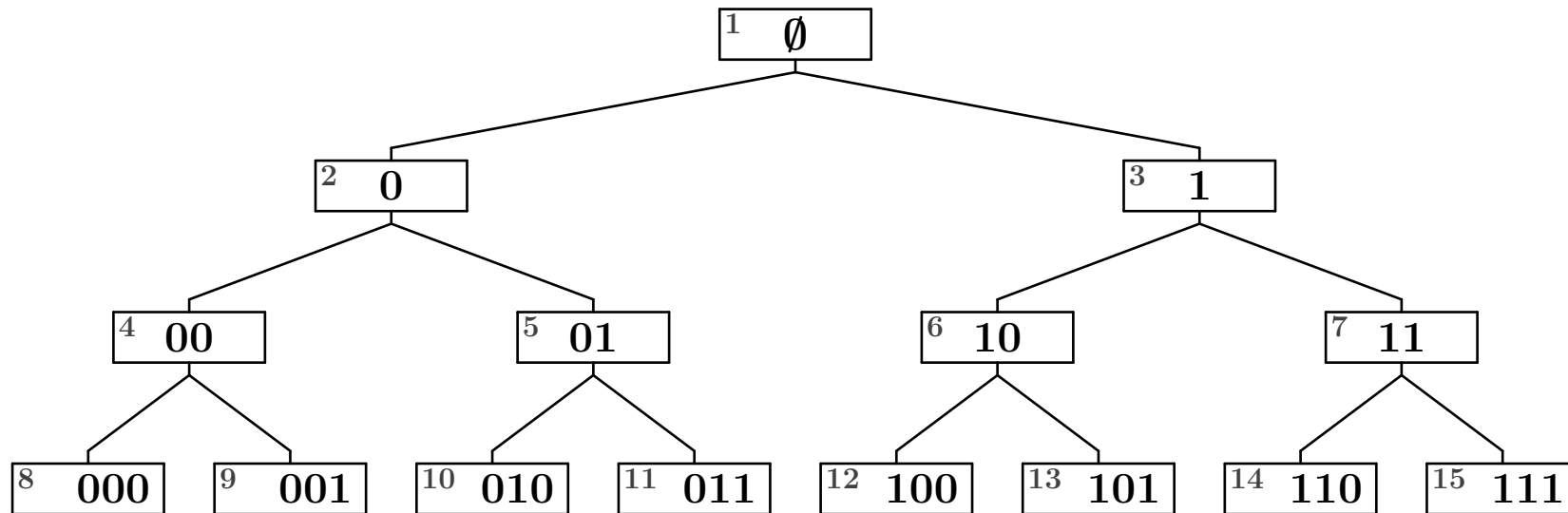
For the search tree we are using as example, in a depth-first search the nodes of the tree are visited in the following order (small numbers in gray on the upper left of each node):



Breadth-first search

In a **breadth-first search** one attempts to avoid going deeper for as long as possible, i.e., one explores the nodes of the search tree one level at a time. It is usually the strategy to use when one wants to first the shallowest solution and when the tree can be very deep. (The problem may ask, for example, for the smallest number of moves to win a game.) Its implementation is more difficult than that of a depth-first search, and it usually requires a significant amount of memory to store the search tree nodes that have not yet been expanded.

For the search tree we are using as example, in a breadth-first search the nodes of the tree are visited in the following order (small numbers in gray on the upper left of each node):



Traversing a binary tree in depth-first order and in breadth-first order

The following two non-recursive functions traverse a binary tree in depth-first and in breadth-first order. Note that they only differ in the data structure used to keep track of the nodes that have not yet been expanded! In both cases each node of the binary tree will be implemented (in C++) as follows:

```
typedef struct tree_node
{
    struct tree_node *left;    // pointer to the left branch (a sub-tree)
    struct tree_node *right;   // pointer to the right branch (a sub-tree)
    int data;                  // the data item (we use an int here, but it can be anything)
}
tree_node;
```

Here go the functions:

```
void depth_first(tree_node *root)
{
    stack<tree_node *> s;
    tree_node *n;

    s.push(root);
    while(s.isEmpty() == 0)
        if((n = s.pop()) != nullptr)
        {
            visit(n);
            s.push(n->left);
            s.push(n->right); // right is done first!
        }
}
```

```
void breadth_first(tree_node *root)
{
    queue<tree_node *> s;
    tree_node *n;

    s.enqueue(root);
    while(s.isEmpty() == 0)
        if((n = s.dequeue()) != nullptr)
        {
            visit(n);
            s.enqueue(n->left); // left is done first!
            s.enqueue(n->right);
        }
}
```

Observe their simplicity and elegance! The first one can also be easily implemented in a recursive way but that cannot be done easily for the second. The maximum queue size can be quite large!

Backtracking

When performing a depth-first search it is quite common to not be able to progress into a deeper search tree level. In that case one has to go retrace our steps and try the next possibility at a lower level (this is called **backtracking**). If the depth-first search is done via recursion, backtracking is done by returning from the function.

A backtracking depth-first search algorithm usually has the following structure:

1. [Initialize.] Initialize all relevant data structures and set the search level to 0. Go to Step 2.
2. [Try the current configuration.] If the current configuration is not acceptable, go to Step 3. Otherwise, if we have a solution to the problem, record it (and terminate the algorithm if the goal was to find one solution) and go to Step 3. Otherwise, update the relevant data structures to reflect the new data in the current configuration, increase the search level by 1, and go to Step 3.
3. [Advance.] Advance to the next configuration. If one exists, go back to Step 2. Otherwise, go to Step 4.
4. [Backtrack.] If we are at search level 0, terminate the algorithm. Otherwise, decrease the search level by 1 and undo the changes made to the relevant data structures in step 2. After that, go back to Step 3.

Depth-first search trees usually have a huge number of nodes, so any gain in speed in its implementation is welcome. Thus, although it is possible to code a depth-first search algorithm using a recursive function, for efficiency reasons it is usually preferable to do it all in the same function, using a custom made stack to store the search history. The author of this document goes one step further: he usually uses goto statements to jump between the different stages of the search, because it makes the code easier to understand! (The use of the more “respectable” control flow statements in algorithms of this nature usually complicates matters if one wants an efficient implementation.)

Pruning

In some problems it is possible to check if continuing to search for a solution by expanding the current search tree node (i.e., going deeper by checking the node's children) can lead to a solution, or to a better solution, of the problem. If it can be determined in advance that a (better) solution is not possible, one can immediately advance to the next configuration at the same search depth. This is called **pruning** the search tree.

If the pruning test is expensive, it may be advantageous to apply it only at some selected search tree depths, say, every tenth level.

The most successful depth-first search algorithms use pruning to reduce, sometimes dramatically, the number of search tree nodes that are visited by the program. In some difficult problems (i.e., very time consuming problems), one may even perform a non-exhaustive search by pruning the search tree with an **heuristic** (a test that keeps only promising search tree branches).

Pruning is usually performed in depth-first searches. It can also be done in a breadth-first search.

An example: a chessboard problem (part 1, problem statement and some code)

Consider the following “toy problem” (actually, not a toy problem, check problem C18 in “Unsolved Problems in Number Theory,” Richard K. Guy, third edition, Springer, 2004): find the maximum number of not attacked squares when Q queens are placed on a $W \times H$ chessboard (squares with queens are considered to be attacked). This problem will be solved using depth-first search, backtracking and pruning.

We begin by declaring the data structures used to solve the problem. In general, this step requires considerable thought. In our case, a few arrays are enough (for legibility, the code uses `max_n_queen` instead of Q , `max_x` instead of W , and `max_y` instead of H):

```
#define true_max_x      10 // max_x cannot be larger than this
#define true_max_y      10 // max_y cannot be larger than this
#define true_max_n_queens 10 // max_n_queens cannot be larger than this

int board[true_max_x][true_max_y];
int queen_x[true_max_n_queens], queen_y[true_max_n_queens];
int max_x, max_y, max_n_queens, n_unattacked;

int best_queen_x[true_max_n_queens], best_queen_y[true_max_n_queens], best_n_unattacked;

void init_board(void)
{
    for(int x = 0; x < max_x; x++) for(int y = 0; y < max_y; y++) board[x][y] = 0; // unattacked
    n_unattacked = max_x * max_y;
    best_n_unattacked = 0;
}
```

The array `board[][]` will record the number of times each square is attacked by the queens, and the arrays `queen_x[]` and `queen_y[]` will record the coordinates of the queens. The variable `n_unattacked` will keep track of the number of not attacked squares. The `best_...` variables will keep track of the best solution (i.e., the one with the largest number of not attacked squares).

An example: a chessboard problem (part 2, support code)

Next, we need a way to register the effects of placing a queen in a square and a way of undoing those effects. The following two simple functions do those jobs:

```
void mark(int x,int y)
{
# define apply_mark(xx,yy)  do if(board[xx][yy]++ == 0) n_unattacked--; while(0)
    for(int i = 0 ;      i < max_x      ; i++) apply_mark(i,y);
    for(int i = 0 ;      i < max_y      ; i++) apply_mark(x,i);
    for(int i = 1 ; x + i < max_x && y + i < max_y ; i++) apply_mark(x + i,y + i);
    for(int i = 1 ; x - i >= 0  && y - i >= 0  ; i++) apply_mark(x - i,y - i);
    for(int i = 1 ; x + i < max_x && y - i >= 0  ; i++) apply_mark(x + i,y - i);
    for(int i = 1 ; x - i >= 0  && y + i < max_y ; i++) apply_mark(x - i,y + i);
# undef apply_mark
}

void unmark(int x,int y)
{
# define remove_mark(xx,yy) do if(--board[xx][yy] == 0) n_unattacked++; while(0)
    for(int i = 0 ;      i < max_x      ; i++) remove_mark(i,y);
    for(int i = 0 ;      i < max_y      ; i++) remove_mark(x,i);
    for(int i = 1 ; x + i < max_x && y + i < max_y ; i++) remove_mark(x + i,y + i);
    for(int i = 1 ; x - i >= 0  && y - i >= 0  ; i++) remove_mark(x - i,y - i);
    for(int i = 1 ; x + i < max_x && y - i >= 0  ; i++) remove_mark(x + i,y - i);
    for(int i = 1 ; x - i >= 0  && y + i < max_y ; i++) remove_mark(x - i,y + i);
# undef remove_mark
}
```

The first two for cycles deal with the squares attacked by a queen in the horizontal and vertical directions, the next two deal (somewhat inefficiently) with squares attacked in the 45 degree direction, and the last two deal (again, somewhat inefficiently) with squares attacked in the -45 degree direction.

An example: a chessboard problem (part 3, more support code)

Next, we need a way to record and present best solutions:

```
void update_best_solution(void)
{
    if(n_unattacked > best_n_unattacked)
    {
        best_n_unattacked = n_unattacked;
        for(int i = 0; i < max_n_queens; i++)
        {
            best_queen_x[i] = queen_x[i];
            best_queen_y[i] = queen_y[i];
        }
    }
}

#include <stdio.h>

void show_best_solution(void)
{
    if(best_n_unattacked == 0)
        return;
    printf("%2d %2d %2d %2d ", max_x, max_y, max_n_queens, best_n_unattacked);
    for(int i = 0; i < max_n_queens; i++)
        printf(" (%d,%d)", best_queen_x[i], best_queen_y[i]);
    printf("\n");
    fflush(stdout);
}
```

An example: a chessboard problem (part 4, the depth-first code)

Now everything is in place to present the depth-first search code in all its glory:

```
void solve(void)
{
    int x,y,n_queens;

    init_board();
    x = y = n_queens = 0;
place_queen: mark(queen_x[n_queens] = x, queen_y[n_queens] = y);
    if(++n_queens == max_n_queens)
    {
        update_best_solution();
        goto backtrack;
    }
/* prune: */ if(n_unattacked <= best_n_unattacked)
    goto backtrack;
next_position: if(++y == max_y)
    {
        y = 0;
        if(++x == max_x)
            goto backtrack;
    }
    goto place_queen;
backtrack: if(--n_queens < 0)
    goto done; // this goto can be easily avoided, the others are not that easy...
    unmark(x = queen_x[n_queens], y = queen_y[n_queens]);
    goto next_position;
done:      show_best_solution();
}
```

In this particular case the goto statements are used in a disciplined way and actually make the program cleaner (at least to the author). If you are not convinced, try doing it using more traditional control flow statements and without recursion (remember, recursion is in this case inefficient); goto statements are not always harmful!

An example: a chessboard problem (part 4, the main code)

It is now time to present the main program. It just requests solutions for all board sizes and number of queens up to the hardcoded limits defined at the beginning of the code:

```
int main(void)
{
    printf(" X  Y  Q  P  coords\n");
    printf("-- -- -- --  -----\n");
    for(max_x = 1;max_x <= true_max_x;max_x++)
        for(max_y = max_x;max_y <= true_max_y;max_y++)
            for(max_n_queens = 1;max_n_queens <= true_max_n_queens;max_n_queens++)
                solve();
    printf("-- -- -- --  -----\n");
    return 0;
}
```

[Research problem: What is the exact maximum number of not attacked squares when N queens are placed on an $N \times N$ chessboard for $N > 14$?]

Two extra examples (sudoku and klotski)

Summary:

- sudoku solver (depth-first search example)
- klotski solver (breadth-first search example)

Remarks: The explanation of the two examples of the following slides follow a **bottom-up** approach: they start by the foundations (data types and low-level functions), and then they present functions that depend on the data types and functions that were previously explained, finishing with the `main()` function. The author of these slides almost always writes his programs in this way. **However**, his approach to the **planning phase** of the program is **top-down**. He usually spends a significant amount of time thinking about the data structures that will be used, and about how to subdivide the program into manageable parts. He only starts writing code when he has convinced himself that his solution will work and is reasonably efficient. While coding, when he has finished a module (a group of functions that address a particular part of the problem), we usually writes one of more test functions to make sure that that part of the program works as planned. The complete code of these two examples can be found in the archive `P09.tgz`.

Highly recommended bibliography:

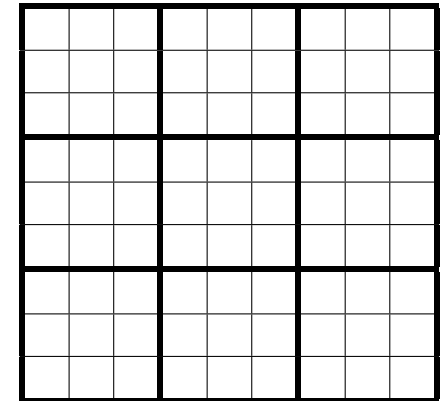
- **Dancing links**, Donald E. Knuth, 2000. [Although the dancing links method is not explained in these slides, reading this paper is definitely worth the time.]

Recommended bibliography:

- **Programming Pearls**, Jon Bentley, second edition, Addison Wesley, 2000.
- **Introduction to Algorithms**, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, third edition, The MIT press, 2009.
- **The Algorithm Design Manual**, Steven S. Skiena, second edition, Springer, 2008.
- **Algorithm Design**, Jon Kleinberg and Éva Tardos, Addison Wesley, 2006.

Sudoku solver (part 1, problem formulation)

In the standard **sudoku puzzle** a 9×9 array of cells is partitioned (independently) into 9 rows, into 9 columns, and into 9 smaller 3×3 arrays of cells (see figure on the right hand side). Given a set of clues (numbers already placed in the array), the objective of the sudoku puzzle is to fill the empty cells with the numbers $1, 2, \dots, 9$ in such a way that



- each row contains only one occurrence of each number,
- each column also contains only one occurrence of each number, and
- each 3×3 smaller square also contains only one occurrence of each number.

Usually, the clues are selected so that the puzzle has only one solution (we can use that to our advantage!). In this lecture we are going to study in detail a program capable of **solving** sudoku puzzles. It has the following characteristics:

- it can deduce cell numbers using some simple methods,
- when no deduction is possible, it can guess cell numbers (and backtrack if the guess was wrong),
- it reports the number of solutions found (0, 1, or more than 1), and
- it can be configured to use only some deduction methods, so it can be the starting point of another program capable of **generating** sudoku puzzles. (One starts with a complete array and then one removes numbers at random and one keeps checking if the puzzle can still be solved and if it still has only one solution.)

The program performs a depth-first search because if a second guess is necessary it does it without trying first the other possible choices for the first guess.

Sudoku solver (part 2a, deduction methods)

The program will use the following deduction methods:

- **Method number 1: (elementary)** If only one number can be placed in a cell, place it. (Example on the left below.)
- **Method number 2: (elementary)** If in a region (row, column, or small 3×3 square) a number can only be placed in one cell, place it in that cell. (Example on the middle below.)
- **Method number 3: (intermediate/advanced)** If in a region there exist n cells in which only the same n numbers can be placed, then these numbers cannot be placed in the other cells of that region. Furthermore, if these cells also all belong to another region, then these numbers cannot be placed in the other cells of that other region. (Example on the right below.)

These methods should be reapplied until they do not produce any more changes in the puzzle state. The following examples illustrate how these deduction methods work. On the first two, the number inside the gray square is forced. On the last, the gray squares cannot contain the numbers in the red squares (the red squares must contain the numbers 1, 3, and 7, the other two cells of the central small square must contain the numbers 2 and 8).

	3				
1		4			
	7			2	8
	5				9
	6				

1					
				1	
		1			
7	2	9			
	1				

			8		
				6	
	1			4	5
				9	3
			2		

Stop and think! How can these tasks be accomplished in an efficient way?

Sudoku solver (part 2b, data structures for the deduction methods)

Deduction method number 1 is implemented by keeping for each cell a list of the numbers that can still be placed there. Given that there are only 9 possible numbers for each cell, the list can be stored in a single integer (in the code this integer is called a `mask`), one bit per possible number. If bit number i of the mask is set (equal to 1) then the number $i + 1$ can be placed in the cell without violating the sudoku rules. To apply this deduction method it will be necessary to find out if a mask has only one bit set. This is accomplished by storing in the array `n_bits[]` the number of bits equal to 1 for all possible masks. If the mask has only one bit set, it will be necessary to find out that bit number. This is accomplished by storing in the array `lsb_number[]` the number of the least significant bit that is not zero (if the mask is all zeros, -1 is stored in that array).

Deduction method number 2 is implemented by providing a way to expand a mask, inserting 3 zero bits between the bits of the original mask (and three more on the left). For example, the 9-bit mask **100110110** when expanded becomes the 36-bit mask **000100000000000100010000000100010000**. By adding up the expanded masks one is counting **in parallel** the number of times each digit appears in the masks. (Given that a digit can appear in all 9 masks, at least 4 bits are necessary to do this for each number.) In the program, the array `expand_mask[]` stores the expanded masks.

Deduction method number 3 uses the arrays described above and some bit manipulation tricks to do its job. In particular, code such as the following is used to identify and extract one at a time the bits set to one of a mask:

```
for(mask_copy = mask; ((bit_number = lsb_number[mask_copy]) >= 0; mask_copy ^= 1 << bit_number)
{
    // do some stuff here (this is done n_bits[mask] times)
}
```


Sudoku solver (part 3, bit stuff)

The following code initializes the arrays `n_bits[]`, `lsb_number[]`, and `expand_mask[]` described in the previous slide. Note that there are $2^9 = 512$ possible masks and that the data type of the `expand_mask[]` array must be able to hold at least 36 bits.

```
static int n_bits[512];           // number of bits set to 1
static int lsb_number[512];       // number of the least significant bit that is set to 1
static long long expand_mask[512]; // transform each mask bit into four bits (insert three zeros)

static void init_mask_data(void)
{
    int m,i,j;

    for(m = 0;m < 512;m++)        // for each mask ,,,
    {
        n_bits[m] = 0;           // compute its number of bits set to one
        lsb_number[m] = -1;      // compute also the bit number of its least significant bit set to one
        expand_mask[m] = 011;    // and compute its expanded mask ...
        i = -1;
        for(j = m;j != 0;j >>= 1)
        {
            i++;
            if((j & 1) != 0)      // is the i-th bit of m one?
            {                    // yes!
                n_bits[m]++;      // one more bit
                if(lsb_number[m] < 0) // is this the first bit set to one?
                    lsb_number[m] = i; // yes! set the least significant bit number
                expand_mask[m] |= 111 << (4 * i); // update the expanded mask
            }
        }
    }
}
```

Sudoku solver (part 4a, puzzle regions)

The 9×9 array of cells is partitioned into rows, columns, and smaller 3×3 arrays of cells (called regions in the code). It will be necessary to know which cells belong to which regions and vice versa. That information is stored in the arrays `cell_regions[] []` and `region_cells[] []`. To apply deduction method number 3 it will also be necessary to discover if a group of cells belongs to more than one region. Sufficient information to do this is stored in the array `region_intersections[] []`.

The following code initializes the arrays mentioned above. Note that there are 81 cells and 27 regions, that each cell belongs to 3 regions, that each region has 9 cells, and that there are 54 relevant region intersections.

```
#include <assert.h>
```

```
static int cell_regions[81][3];           // the numbers of the regions each cell belongs to
static int region_cells[27][9];           // the cells that are part of each region
static int region_intersections[54][5];   // intersection data

void init_regions(void)
{
    int c,r,n,x,y,r1,r2,i,j,C[9],ni;

    for(c = 0; c < 81; c++)                // for each cell ...
    {
        x = c % 9;                         // cell x coordinate
        y = c / 9;                         // cell y coordinate
        cell_regions[c][0] = x;             // vertical region formed by the same x value
        cell_regions[c][1] = 9 + y;         // horizontal region formed by the same y value
        x /= 3;                            // square region x coordinate
        y /= 3;                            // square region y coordinate
        cell_regions[c][2] = 18 + 3 * y + x; // square region
    }
}
```

— the code continues on the next slide —

— continuation of the code of the previous slide —

```
for(r = 0;r < 27;r++)           // for each region ...
{
    for(n = c = 0;c < 81;c++)    // find the cells that belong to this region ...
        if(cell_regions[c][0] == r || cell_regions[c][1] == r || cell_regions[c][2] == r)
            region_cells[r][n++] = c;
    assert(n == 9);              // must be 9 for each region
}
ni = 0;                          // count the number of valid region intersections
for(r1 = 18;r1 < 27;r1++)       // for all square regions ...
    for(r2 = 0;r2 < 18;r2++)    // for all non-square regions ...
    {
        n = i = j = 0;          // compute intersection of the two regions
        while(i < 9 && j < 9)    // find and count the number of common elements of two sorted arrays
            if(region_cells[r1][i] == region_cells[r2][j])
            {
                C[n++] = region_cells[r1][i++];
                j++;
            }
            else if(region_cells[r1][i] < region_cells[r2][j])
                i++;
            else
                j++;
        if(n == 3)              // if the intersection has 3 cells ...
        {
            region_intersections[ni][0] = C[0]; // record first cell number
            region_intersections[ni][1] = C[1]; // record second cell number
            region_intersections[ni][2] = C[2]; // record third cell number
            region_intersections[ni][3] = r1;   // record first region number
            region_intersections[ni][4] = r2;   // record second region number
            ni++;
        }
    }
assert(ni == 54);               // must be 54
}
```

Sudoku solver (part 4b, find a puzzle region)

To apply deduction method number 3 it is necessary to find if a given group of cells belong to two regions, and if so it is necessary to find the number of one of those regions given the number of the other. The following code does precisely that.

```
static int find_region(int r,int nc,int *c)
{
    int i,j,k,n;

    if(nc < 2 || nc > 3)    // only 2 or 3 cells are possible
        return -1;        // no valid region
    for(i = 0;i < 54;i++)  // for each possible intersection
    {
        n = j = k = 0;    // count the number of common elements of two sorted arrays
        while(j < 3 && k < nc)
            if(region_intersections[i][j] == c[k])
            {
                n++;
                j++;
                k++;
            }
            else if(region_intersections[i][j] < c[k])
                j++;
            else
                k++;
        assert(n != nc || region_intersections[i][3] == r || region_intersections[i][4] == r);
        if(n == nc)        // found it! return the number of the other region
            return (region_intersections[i][3] == r) ? region_intersections[i][4] : region_intersections[i][3];
    }
    return -1;            // no valid region
}
```

[Homework: Study how the number of common elements of two sorted arrays is counted.]

Sudoku solver (part 5a, puzzle state initialization)

The state of the sudoku puzzle is essentially the state of each one of its cells. For each cell it was decided to store the number it holds (digit in the code below), to store a mask (mask) of the possible numbers it can hold, to record the method number (method) used to place the number, and to use a flag (frozen) to simplify the implementation of the deduction method number 3. The following function initializes the puzzle state.

```
typedef struct
{
    int n_known_digits;          // number of digits already known
    struct
    {
        int digit;              // number placed in this cell (-1 means none, 0..8 means 1..9)
        int mask;               // bit-mask of the digit values that can be placed in this cell
        int method;             // method number used to place a digit in this cell (-1 means none)
        int frozen;             // if non-zero, do not make changes to the mask
    }
    cells[81];                  // the state of each of the 9x9 cells
}
state;

void init_state(state *s)
{
    int c;

    s->n_known_digits = 0;      // no known digits
    for(c = 0; c < 81; c++)    // for each cell ...
    {
        s->cells[c].digit = -1; // no digit
        s->cells[c].mask = 0xFF; // all digits are possible
        s->cells[c].method = -1; // no placement method
        s->cells[c].frozen = 0; // not frozen
    }
}
```

Sudoku solver (part 5b, display puzzle state)

The following function outputs the current puzzle state.

```
#include <stdio.h>

static void show_state(state *s)
{
    int c,x,y,t0,t1,t2,t4,tx;

    t0 = t1 = t2 = t4 = tx = 0;
    for(c = 0; c < 81; c++)
    {
        x = c % 9;
        y = c / 9;
        if(x == 0)
            printf(" ");
        if(s->cells[c].digit < 0)
            printf("[?]");
        else
            printf("[%d]", s->cells[c].digit + 1);
        switch(s->cells[c].method)
        {
            case 0: printf("F"); t0++; break;
            case 1: printf("1"); t1++; break;
            case 2: printf("2"); t2++; break;
            case 4: printf("g"); t4++; break;
            default: printf("?"); tx++; break;
        }
        if(x < 8)
            printf((x % 3 == 2) ? " " : " ");
        else
            printf((y % 3 == 2) ? "\n\n" : "\n");
    }
    printf(" --- F:%d 1:%d 2:%d g:%d ?:%d\n", t0, t1, t2, t4, tx);
}
```

Sudoku solver (part 6, place a number)

The following function places a number in a given cell. It returns 0 if no solution is possible (that happens when a mask becomes 0), and returns 1 otherwise.

```
static int place_digit(state *s,int cell,int digit,int method)
{
    int ir,r,ic,c,m;

    assert(0 <= cell && cell < 81 && digit >= 0 && digit <= 8);
    assert(s->cells[cell].digit == -1 && (s->cells[cell].mask & (1 << digit)) != 0);
    s->n_known_digits++;
    s->cells[cell].digit = digit;
    s->cells[cell].mask = 0;
    s->cells[cell].method = method;
    s->cells[cell].frozen = 1;
    m = 1 << digit;                // digit mask
    for(ir = 0;ir < 3;ir++)        // for each of the three cell regions ...
    {
        r = cell_regions[cell][ir];
        for(ic = 0;ic < 9;ic++)    // for each of the 9 cells belonging to the region ...
        {
            c = region_cells[r][ic];
            if((s->cells[c].mask & m) != 0)    // if the digit can be place here ...
                if((s->cells[c].mask &= ~m) == 0)    // make that impossible, and say no solution is
                    return 0;                        // possible if the new mask is 0
        }
    }
    return 1;                        // say that a solution is possible
}
```

Sudoku solver (part 7, parse a sudoku puzzle)

The following function extract the clues of a sudoku puzzle from an initialization string and places them on an initially empty puzzle state. It returns 0 if the initialization string is not valid, or if no solution is possible, and returns 1 otherwise.

```
static int init_sudoku(state *s, char *data)
{
    int c;

    init_state(s);
    for(c = 0; c < 81 && *data != '\0'; data++)
        if(*data == ' ' || *data == '.' || *data == '?')
            c++;
        else if(*data >= '1' && *data <= '9' && place_digit(s, c++, (int)(*data) - '1', 0) == 0)
            return 0;
    return (*data == '\0' && c == 81) ? 1 : 0;
}
```


Sudoku solver (part 8, deduction method number 1)

Everything is now in place to start presenting code to actually solve the sudoku puzzle. The functions that implement the number deduction methods return 0 if they did change the state of the puzzle and if because of that a solution is no longer possible, return 1 if they did not change the state of the puzzle (so a solution is still possible), and return 2 if they did change the state of the puzzle and if a solution is still possible.

The following code deals with cells that can have only one possible value.

```
static int do_method_1(state *s)
{
    int rv,c,d;

    rv = 1;
    for(c = 0;c < 81;c++)          // for each cell ...
        if(n_bits[s->cells[c].mask] == 1)    // only one possible digit?
        {                                  // yes!
            d = lsb_number[s->cells[c].mask]; // find digit
            if(place_digit(s,c,d,1) == 0)     // place it
                return 0;                    // say no solution is possible if place_digit() says so
            rv = 2;                          // say at least one digit was placed
        }
    return rv;
}
```

Sudoku solver (part 9, deduction method number 2)

The following code deals with regions in which a digit can only be placed in one cell.

```
static int do_method_2(state *s)
{
    int rv,r,ic,c,d;
    long long sum;

    rv = 1;
    for(r = 0;r < 27;r++)           // for each region ...
    {
        sum = 0ll;
        for(ic = 0;ic < 9;ic++)     // for each cell ...
        {
            c = region_cells[r][ic];
            sum += expand_mask[s->cells[c].mask];
        }
        for(d = 0;d < 9;d++)         // for each digit ...
            if(((sum >> (4 * d)) & 15ll) == 1ull) // if the digit appears in only one mask ...
                for(ic = 0;ic < 9;ic++)         // find cell! For each cell ...
                {
                    c = region_cells[r][ic];
                    if(((s->cells[c].mask >> d) & 1) != 0) // is this cell the one?
                    {
                        // yes!
                        if(place_digit(s,c,d,2) == 0) // place the digit in the cell
                            return 0;                // say no solution is possible if place_digit() says so
                        rv = 2;                        // say at least one digit was placed
                    }
                }
    }
    return rv;
}
```

Sudoku solver (part 10a, deduction method number 3, trim masks)

The following function adjusts the masks of all unfrozen cells in a region so that they no longer can hold a given set of numbers. It returns 0 if a solution becomes impossible, returns 1 if no change was made to the puzzle state, and returns 2 otherwise.

```
static int trim_masks(state *s,int r,int mask)
{ // 0 -> no solution, 1 -> no change, 2 -> change
  int rv,ic,c,new_mask;

  rv = 1;
  if(r >= 0 && r <= 26)                // if the region number is valid ...
    for(ic = 0;ic < 9;ic++)              // for each cell ...
    {
      c = region_cells[r][ic];
      if(s->cells[c].frozen == 0)         // if the mask can be modified ...
      {
        new_mask = s->cells[c].mask & ~mask; // trim mask
        if(new_mask == 0)
          return 0;                       // say no solution is possible
        if(new_mask != s->cells[c].mask)   // if the new mask is different from the old one ...
        {
          s->cells[c].mask = new_mask;     // update mask
          rv = 2;                          // and say that at least one mask changed
        }
      }
    }
  return rv;
}
```

Sudoku solver (part 10c, deduction method number 3)

The following function implements the deduction method number 3. For each region it first records all cells without numbers (that is the code presented in this page), and then for each possible subset of `nc` cells it checks if their masks force `nc` numbers to be placed in them; if so these numbers are removed from the masks of the other cells of the region and of a possible “intersecting” region (that is the code presented on the next page). For example, the last two of the following five masks $m_1 = 000000101$, $m_2 = 001000100$, $m_3 = 001000001$, $m_4 = 010000111$, and $m_5 = 011000011$ can be trimmed to $m_4 = m_5 = 010000010$ because the mask m_1 or m_2 or $m_3 = 001000101$ has 3 bits set to 1 and so three numbers (1, 3, and 7) must be placed in the cells corresponding to these masks. Here is its code:

```
static int do_method_3(state *s)
{
    int rv,r,ic,c,nm,M[9],C[9],nc,CC[9],i,j,cm,b,rr;

    rv = 1;
    for(r = 0;r < 27;r++)                // for each region ...
    {
        nm = 0;                          // count the number of cells and masks that are relevant
        for(ic = 0;ic < 9;ic++)          // for each cell ...
        {
            c = region_cells[r][ic];
            if(s->cells[c].digit < 0)    // if it does not yet have a digit ...
            {
                C[nm] = c;                // record the cell
                M[nm] = s->cells[c].mask; // record the mask
                nm++;
            }
        }
    }
}
```

— the code continues on the next slide —

— continuation of the code of the previous slide —

```
for(i = 3; i < (1 << nm); i++)          // for each possible subset with at least 2 elements
{
    if(n_bits[i] >= 2)
    {
        nc = 0;                          // count the cells of the subset and combine (logical or) their masks
        for(cm = 0, j = i; (b = lsb_number[j]) >= 0; j ^= 1 << b)
        {
            CC[nc++] = C[b];              // record the cell
            cm |= M[b];                   // combine masks
        }
        assert(n_bits[i] == nc);          // not needed, but we check this anyway
        if(nc == n_bits[cm])              // if the number of cells is equal to the number of possible digits
        {
            for(j = 0; j < nc; j++)
                s->cells[CC[j]].frozen = 1; // freeze the cells of the subset
            switch(trim_masks(s, r, cm))    // trim all unfrozen cells of the region
            {
                case 0: return 0;          // say no solution is possible if trim_masks() says so
                case 2: rv = 2;            // say that at least one mask changed if trim_masks() says so
            }
            rr = find_region(r, nc, CC);    // find another region (if any) that has the frozen cell of the current one
            if(rr >= 0)                    // if it exists ...
                switch(trim_masks(s, rr, cm)) // trim all unfrozen cells of that region
                {
                    case 0: return 0;      // say no solution is possible if trim_masks() says so
                    case 2: rv = 2;        // say that at least one mask changed if trim_masks() says so
                }
            for(j = 0; j < nc; j++)
                s->cells[CC[j]].frozen = 0; // unfreeze the cells of the subset
        }
    }
}
return rv;
}
```

Sudoku solver (part 11, solve sudoku)

It is now time to present the function that solves a sudoku puzzle. It returns -1 when a solution was not found, returns 0 when the puzzle does not have a solution, returns 1 when it has only one solution, and returns 2 when it has two or more solutions. It records one solution in the sol argument. While attempting to find a solution it does not use methods with numbers larger than max_method (guessing a number is method number 4). Here is its code:

```
static int solve_sudoku(state *s, state *sol, int max_method)
{
    int rv1, rv2, rv3, c, bc, nb, m, d, ns;
    state ss;

    rv1 = rv2 = rv3 = 2;
    while(s->n_known_digits < 81 && (rv1 > 1 || rv2 > 1 || rv3 > 1)) // deduce digits
    {
        rv1 = rv2 = rv3 = 1;
        if(max_method >= 1 && (rv1 = do_method_1(s)) == 0) return 0;
        if(max_method >= 2 && (rv2 = do_method_2(s)) == 0) return 0;
        if(max_method >= 3 && (rv3 = do_method_3(s)) == 0) return 0;
    }
    if(s->n_known_digits == 81) { *sol = *s; return 1; } // solved!
    if(max_method < 4) return -1; // not solved!
    for(nb = 10, bc = 0, c = 0; c < 81; c++) // find the "best" cell
        if(s->cells[c].digit < 0 && n_bits[s->cells[c].mask] < nb)
            nb = n_bits[s->cells[bc = c].mask];
    for(ns = 0, m = s->cells[bc].mask; ns < 2 && m != 0; m ^= 1 << d) // try all digits
    {
        d = lsb_number[m]; // digit to try
        ss = *s; // clone the state
        if(place_digit(&ss, bc, d, 4) != 0) // place digit
            ns += solve_sudoku(&ss, sol, max_method); // recurse if the digit placement allows solutions
    }
    return (ns > 1) ? 2 : ns;
}
```

Sudoku solver (part 12a, main function)

To exercise the sudoku solver the main function uses it to solve a few interesting puzzles:

```
int main(void)
{
    static struct
    {
        char *data;
        char *author;
    }
    puzzles[] =
    {
        { "7.19..3.6.....6.71...1..29..3....6.99.4...8.76.8....3..19..5...57.4.....4.3..91.8", "solo (trivial)" },
        { "..462.5.....75...6.....3.....5..3284.9.6....1.8...9.4..623.....7.4..", "solo (unreasonable)" },
        { "8.....36.....7..9.2...5...7.....457.....1...3...1....68..85...1..9....4..", "Arto Inkala" },
        { "6....894.9....61...7..4....2..61.....2...89..2.....6...5.....3.8....16..", "David Filmer" },
        { "...8.1.....435.....7.8.....1...2..3....6.....75..34.....2..6..", "McGuire, Tugemann, Civario" }
    };
    state s,sol;
    int i,ns;

    init_mask_data();
    init_regions();
    for(i = 0; i < (int)(sizeof(puzzles) / sizeof(puzzles[0])); i++)
        if(init_sudoku(&s,puzzles[i].data) != 0)
        {
            ns = solve_sudoku(&s,&sol,9);
            printf("Sudoku by %s\n\n",puzzles[i].author);
            if(ns > 0)
                show_state(&sol);
            printf(" --- number of solutions: %d%s\n\n",ns,(ns == 2) ? " or more" : "");
        }
    return 0;
}
```

Sudoku solver (part 12b, main function output)

It is interesting to observe that one of the possible sudoku puzzles with the least number of clues (17) can be solved by entirely elementary methods:

Sudoku by McGuire, Tugemann, Civario

```
[2]1 [3]1 [7]1 [8]F [4]1 [1]F [5]1 [6]2 [9]2
[1]2 [8]2 [6]2 [7]1 [9]1 [5]2 [2]1 [4]F [3]F
[5]F [9]1 [4]1 [3]2 [2]1 [6]1 [7]1 [1]2 [8]2
[3]2 [1]1 [5]2 [6]2 [7]F [4]2 [8]F [9]2 [2]1
[4]2 [6]1 [9]2 [5]1 [8]2 [2]2 [1]F [3]2 [7]1
[7]1 [2]F [8]1 [1]1 [3]F [9]1 [4]1 [5]1 [6]1
[6]F [4]1 [2]2 [9]2 [1]2 [8]2 [3]2 [7]F [5]F
[8]2 [5]1 [3]F [4]F [6]1 [7]2 [9]1 [2]2 [1]2
[9]1 [7]1 [1]1 [2]F [5]1 [3]2 [6]F [8]1 [4]2
```

```
--- F:17 1:34 2:30 g:0 ?:0
--- number of solutions: 1
```

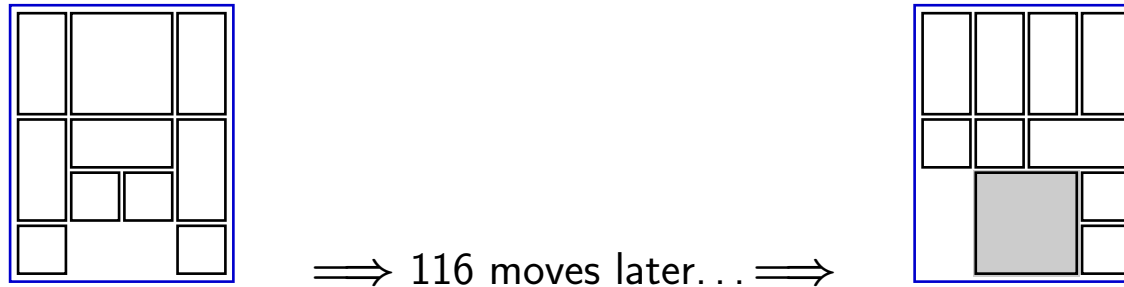
(The clues have a gray background.)

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

Using only the simple deduction methods used by the program, hard sudoku puzzles (Arto Inkala, David Filmer) require several guesses. A more sophisticated program would use more (and more complex) deduction methods.

Klotski solver (part 1, problem formulation)

The so-called **klotski puzzle** is a sliding block puzzle. Given an initial set of blocks placed inside an enclosure, the player has to move the blocks one at a time without lifting them (i.e., sliding them) with the goal of reaching a certain final configuration. In most cases the final configuration consists of placing one of the pieces, usually the largest one, in a certain position. The following figure presents one very popular puzzle of this kind.





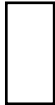

The following slides describe a program capable of solving small puzzles of this kind. It has the following characteristics:

- the shape of the puzzle pieces (1×1 , 2×1 , 1×2 and 2×2) is hardwired in the code
- the puzzle enclosure shape is rectangular, with a width and a height that are at most 8,
- it reports a solution with the smallest number of moves (one move is a piece movement to an adjacent location).

The program performs a breadth-first search because we are interested in a solution with the smallest number of moves. It starts with the initial configuration (generation 0) and produces all possible configurations obtained from it by performing one movement of one piece (this gives rise to generation 1). From that point onward, generation n is obtained by considering all possible piece movements (to an adjacent location) for each of the generation $n - 1$ configurations; the generation n configurations are the **new** ones (those not observed before). The search terminates either when a solution is found or when there are no more new configurations.

Klotski solver (part 2, fundamental data types, defines, and some global variables)

To simplify things, it was decided to make available only the following four piece shapes:

shape 0:  shape 1:  shape 2:  shape 3: 

The shape number can be encoded using only two bits. It was also decided to restrict coordinates to the values 0, 1, ..., 7, so that each of the two coordinates can be encoded in only 3 bits. The following declarations of data types and some macro definitions reflect these choices. (Some global variables are also declared.)

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

typedef unsigned int  u32;      // for hash table indices
typedef unsigned char u08;      // piece info: shape in bits 7..6, x coordinate in bits 5..3, y coordinate in bits 2..0

#define info(s,x,y) ((u08)(((s) << 6) | ((x) << 3) | ((y) << 0)))
#define s_info(p)    (((int)(p) >> 6) & 3)
#define x_info(p)    (((int)(p) >> 3) & 7)
#define y_info(p)    (((int)(p) >> 0) & 7)

#define hash_table_size 50000u // the size of the hash table
#define max_width      8      // maximum puzzle width (HARD CODED, do not change)
#define max_height      8      // maximum puzzle height (HARD CODED, do not change)
#define max_n_pieces    16     // maximum number of pieces a puzzle can have

static u32 width;           // actual puzzle width
static u32 height;          // actual puzzle height
static u32 n_pieces;        // actual number of pieces of the puzzle
static u08 goal;            // puzzle goal (piece shape and position)
```

Klotski solver (part 3a, hash table node data type)

The puzzle configurations will be stored in a hash table. Assuming a sufficiently large hash table size, this makes checking if a configuration is new or not an $O(1)$ operation (on average).

Each hash table node stores one puzzle configuration. It has the following fields:

- the `next_hash_node` field keeps a pointer to a possible other hash table node with an equal key (our hash table will use chaining),
- the `pieces[]` field keeps the compacted information of each piece shape and coordinates,
- the `parent` field keeps a pointer to the configuration that gave rise to this one, and
- the `next_configuration` field keeps a pointer to the next hash table node of the singly-linked link that implements a queue (for the breadth-first search).

The data in the `pieces[]` field is stored in sorted order, so that there exists only one representation for a given puzzle configuration.

Given, the above description, the hash table node data type is as follows:

```
typedef struct hash_node
{
    struct hash_node *next_hash_node;    // pointer to the next hash table node with the same key
    struct hash_node *parent;           // pointer to the configuration that generated this one
    struct hash_node *next_configuration; // pointer to the next configuration to try (queue)
    u08 pieces[max_n_pieces];           // the actual configuration data
}
hash_node;
```

[Question: Which field is “the key”? Which field, or fields, is “the value”?

Klotski solver (part 3b, more global variables and allocation of a hash table node)

The following four variables keep all necessary information about the hash table and about the breadth-first search queue:

```
static hash_node *hash_table[hash_table_size];           // the hash table
static hash_node *free_hash_nodes = NULL;                // linked list of tree hash table nodes
static hash_node *first_untried_configuration = NULL;    // head of the queue linked list
static hash_node *last_untried_configuration = NULL;    // tail of the queue linked list
```

The following function is used to allocate a new hash table node. To reduce memory allocation overheads (in both time and space), it allocates nodes 1000 at a time and it manages itself the free nodes (they are kept in a linked list).

```
static hash_node *allocate_hash_node(void)
{
    hash_node *hn;
    int i;

    if(free_hash_nodes == NULL)
    {
        free_hash_nodes = (hash_node *)malloc((size_t)1000 * sizeof(hash_node));
        for(i = 0; i < 999; i++)
            free_hash_nodes[i].next_hash_node = &free_hash_nodes[i + 1];
        free_hash_nodes[i].next_hash_node = NULL;
    }
    hn = free_hash_nodes;
    free_hash_nodes = free_hash_nodes->next_hash_node;
    return hn;
}
```

Klotski solver (part 3c, hash table initialization and hash function)

The following function initializes the hash table array and the breadth-first search queue.

```
void init_hash_table(void)
{
    u32 i;

    for(i = 0u; i < hash_table_size; i++)
        hash_table[i] = NULL;
    free_hash_nodes = NULL;
    first_untried_configuration = NULL;
    last_untried_configuration = NULL;
}
```

The next function computes the hash function of a given puzzle configuration (in canonical form, i.e., it is assumed that the pieces[] array is already sorted in increasing order).

```
static u32 hash_function(const u08 *pieces)
{
    static u32 table[256];
    u32 crc, i, j;

    if(table[1] == 0u) // do we need to initialize the table[] array?
        for(i = 0u; i < 256u; i++)
            for(table[i] = i, j = 0u; j < 8u; j++)
                if(table[i] & 1u)
                    table[i] = (table[i] >> 1) ^ 0xAED00022u; // "magic" constant
                else
                    table[i] >>= 1;
    crc = 0xAED02016u; // initial value (chosen arbitrarily)
    for(i = 0u; i < (u32)n_pieces; i++)
        crc = (crc >> 8) ^ table[crc & 0xFFu] ^ ((u32)pieces[i] << 24);
    return crc % hash_table_size;
}
```

Klotski solver (part 3d, insert a configuration)

Now comes one of the most important functions: it checks if the configuration it is given is new or not, and if it is it inserts it in the hash table and in the breadth-first search queue.

```
static int insert_configuration(u08 *pieces, hash_node *parent)
{
    hash_node *hn; int i, j; u32 idx;

    // sort the pieces[] array using insertion sort (canonical representation!)
    for(i = 1; i < n_pieces; i++)
    {
        u08 tmp = pieces[i];
        for(j = i; j > 0 && tmp < pieces[j - 1]; j--)
            pieces[j] = pieces[j - 1];
        pieces[j] = tmp;
    }
    // check if this is a new configuration
    idx = hash_function(pieces);
    for(hn = hash_table[idx]; hn != NULL && memcmp(pieces, hn->pieces, n_pieces) != 0; hn = hn->next_hash_node)
        ;
    if(hn != NULL) return 0; // this configuration is already in the hash table, do nothing
    // it is a new configuration, insert it in the hash table and in the breadth-first search queue
    hn = allocate_hash_node();
    hn->next_hash_node = hash_table[idx];
    hash_table[idx] = hn;
    hn->parent = parent;
    if(first_untried_configuration == NULL)
        first_untried_configuration = last_untried_configuration = hn;
    else
        last_untried_configuration = last_untried_configuration->next_configuration = hn;
    hn->next_configuration = NULL;
    memcpy(hn->pieces, pieces, n_pieces);
    return 1;
}
```

Klotski solver (part 4a, piece map)

It is now time to arrange a way to check if a piece can be moved to an adjacent square in a relatively efficient way. This is going to be done by constructing a map of the puzzle.

```
static char map[max_width + 2][max_height + 2];
```

The border of the map will be marked with a plus sign ('+', chosen more or less arbitrarily), an empty square will be marked with a space (' ', also chosen more or less arbitrarily), and the square, or squares, where each piece lies will be marked with the piece's number.

```
static void init_map(u08 *pieces)
{
    int i,x,y,s;

    // initialize map (with a border)
    for(x = 0;x < width + 2;x++)
        for(y = 0;y < height + 2;y++)
            map[x][y] = (x == 0 || x == width + 1 || y == 0 || y == height + 1) ? '+' : ' ';
    // put the pieces on the map
    for(i = 0;i < n_pieces;i++)
    {
        s = s_info(pieces[i]);
        x = x_info(pieces[i]) + 1;
        y = y_info(pieces[i]) + 1;
        switch(s)
        {
            case 0: map[x][y] = (char)i; break;
            case 1: map[x][y] = map[x + 1][y] = (char)i; break;
            case 2: map[x][y] = map[x][y + 1] = (char)i; break;
            case 3: map[x][y] = map[x][y + 1] = map[x + 1][y] = map[x + 1][y + 1] = (char)i; break;
        }
    }
}
```

Klotski solver (part 4b, piece map output)

The following function outputs a puzzle configuration. Each piece is represented by a lower case letter. The output could be beautified but that is not worth the trouble.

```
static void print_map(void)
{
    int x,y;

    for(y = height;y >= 1;y--)
    {
        for(x = 1;x <= width;x++)
            putchar((map[x][y] < n_pieces) ? 'a' + map[x][y] : map[x][y]);
        putchar('\n');
    }
    putchar('\n');
}
```


Klotski solver (part 4c, attempt to move a piece)

The following function receives the index (*i*) of a piece and a movement displacement (*dx* and *dy*) and attempts to move that piece to its new position. If the new configuration solves the puzzle it returns 1. Otherwise it return 0.

```
static int try_move(u08 *pieces,int i,int dx,int dy,hash_node *parent)
{
    u08 new_pieces[max_n_pieces];
    int x,y,s,j;

    s = s_info(pieces[i]);
    x = x_info(pieces[i]) + 1 + dx;
    y = y_info(pieces[i]) + 1 + dy;
    // can we move in this direction?
    if(
        map[x][y] != (char)i && map[x][y] != ' ' ) return 0;
    if((s == 2 || s == 3) && map[x][y + 1] != (char)i && map[x][y + 1] != ' ' ) return 0;
    if((s == 1 || s == 3) && map[x + 1][y] != (char)i && map[x + 1][y] != ' ' ) return 0;
    if(s == 3 && map[x + 1][y + 1] != (char)i && map[x + 1][y + 1] != ' ' ) return 0;
    // yes we can!
    for(j = 0;j < n_pieces;j++)
        new_pieces[j] = pieces[j];
    new_pieces[i] = info(s,x - 1,y - 1);
    if(insert_configuration(new_pieces,parent) == 0)
        return 0;
    for(j = 0;j < n_pieces && new_pieces[j] != goal;j++)
        ;
    return (j < n_pieces) ? 1 : 0;
}
```

Note that the functions `init_map()` and `try_move()` are the ones where the shape of the pieces is hard-coded. It is not difficult to lift that restriction. As stated before, that was not done to simplify the code (and to make it shorter).

Klotski solver (part 5, solve the puzzle)

It is now time to present the function that actually solves the puzzle. It performs a breadth-first search by removing configurations from the queue until the solution is found or until there are no more configurations to consider. When this function returns, the variable `last_untried_configuration` points to the first solution; if it is `NULL` no solution exists.

```
static void solve_puzzle(u08 *initial_configuration)
{
    hash_node *hn;
    int i;

    assert(width <= max_width && height <= max_height && n_pieces <= max_n_pieces);
    init_hash_table();
    insert_configuration(initial_configuration, NULL);
    // do a breadth-first search
    while(first_untried_configuration != NULL)
    {
        hn = first_untried_configuration;
        first_untried_configuration = first_untried_configuration->next_configuration;
        if(hn == last_untried_configuration)
            last_untried_configuration = NULL;
        init_map(hn->pieces);
        for(i = 0; i < n_pieces; i++)
        {
            if(try_move(hn->pieces, i, 1, 0, hn) != 0) return; // return early if solved
            if(try_move(hn->pieces, i, -1, 0, hn) != 0) return; // return early if solved
            if(try_move(hn->pieces, i, 0, 1, hn) != 0) return; // return early if solved
            if(try_move(hn->pieces, i, 0, -1, hn) != 0) return; // return early if solved
        }
    }
    // not solved
    assert(last_untried_configuration == NULL);
}
```

Klotski solver (part 6, main function)

We have finally reached the main function. If the puzzle has a solution, it prints it backwards (stating from the solution until the initial configuration is reached). This is done by following the parent pointers.

```
int main(void)
{
    u08 pieces[max_n_pieces];
    hash_node *hn;
    int i;

    width = 4;
    height = 5;
    n_pieces = 10;
    pieces[0] = info(0,0,0);
    pieces[1] = info(0,3,0);
    pieces[2] = info(2,0,1);
    pieces[3] = info(0,1,1);
    pieces[4] = info(0,2,1);
    pieces[5] = info(2,3,1);
    pieces[6] = info(1,1,2);
    pieces[7] = info(2,0,3);
    pieces[8] = info(3,1,3);
    pieces[9] = info(2,3,3);
    goal      = info(3,1,0);
    solve_puzzle(pieces);
    for(i = 0, hn = last_untried_configuration; hn != NULL; i++, hn = hn->parent)
    {
        init_map(hn->pieces);
        printf("move -%d\n", i);
        print_map();
    }
    printf("%d moves\n", i - 1);
    return 0;
}
```