Chapter 5 Recursion

信息科学与技术学院　　黄方军

data_structures@163.com

东校区实验中心B502

5.1.1 Stack Frames for Subprograms

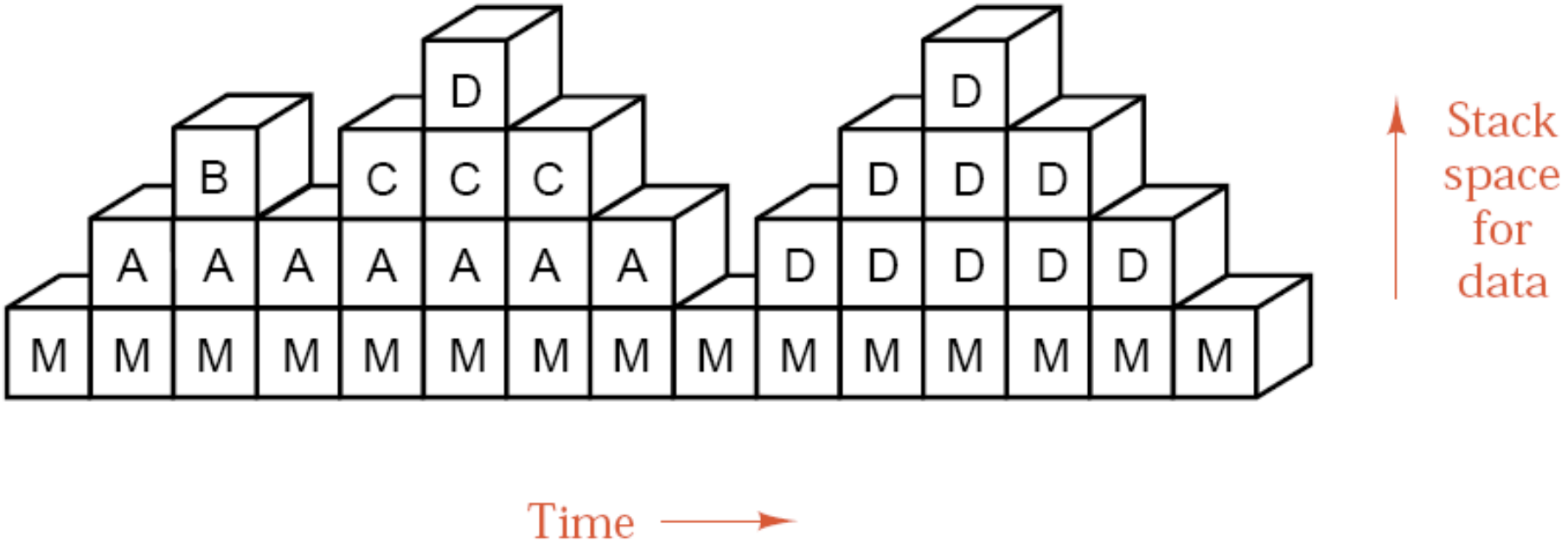


Figure 5.1. Stack frames for function calls

5.1.2 Tree of Subprogram Calls

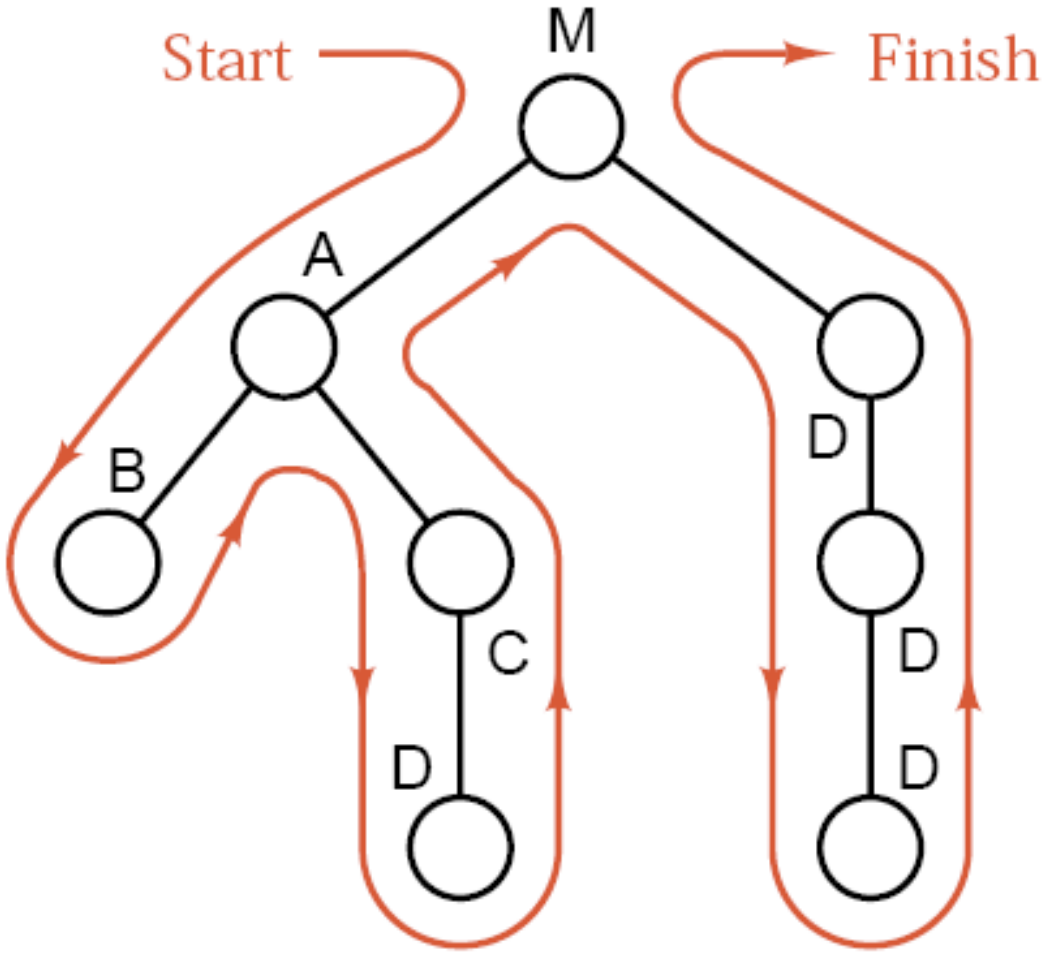
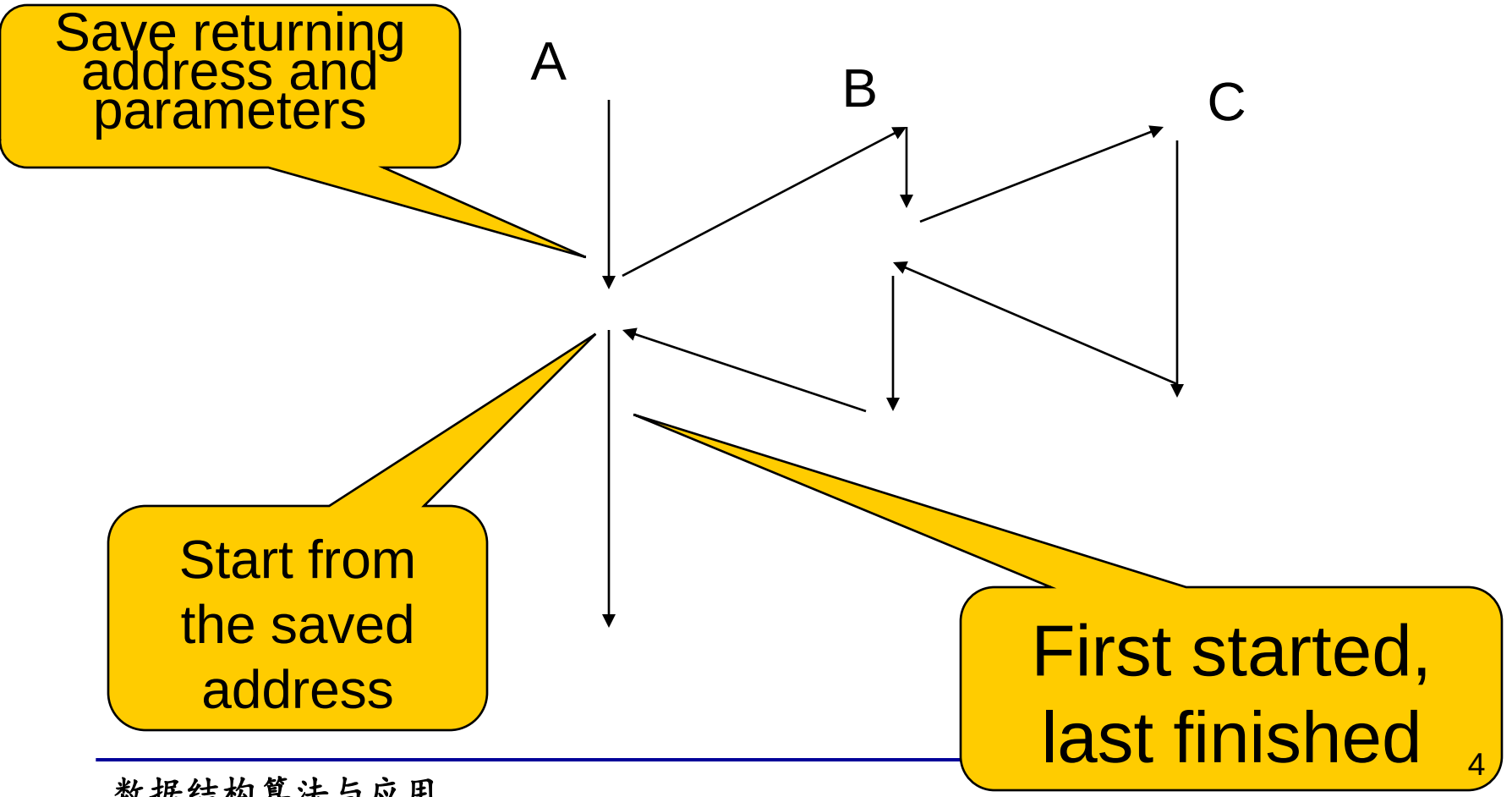


Figure 5.2. Tree of function calls

5.1.2 Tree of Subprogram Calls

Function A calls B, B calls C:



5.1.3 Factorials: A Recursive Definition

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n > 0. \end{cases}$$

$$\begin{aligned} 4! &= 4 \times 3! \\ &= 4 \times (3 \times 2!) \\ &= 4 \times (3 \times (2 \times 1!)) \\ &= 4 \times (3 \times (2 \times (1 \times 0!))) \\ &= 4 \times (3 \times (2 \times (1 \times 1))) \\ &= 4 \times (3 \times (2 \times 1)) \\ &= 4 \times (3 \times 2) \\ &= 4 \times 6 \\ &= 24. \end{aligned}$$

5.1.3 Factorials: A Recursive Definition

```
int factorial(int n)
/* Pre:   n is a nonnegative integer.
   Post:  Return the value of the factorial of n. */
{
   if (n ==  0)
      return 1;
   else
      return n * factorial(n – 1);
}
```

5.1.4 Divide and Conquer: Hanoi

1. The Problem

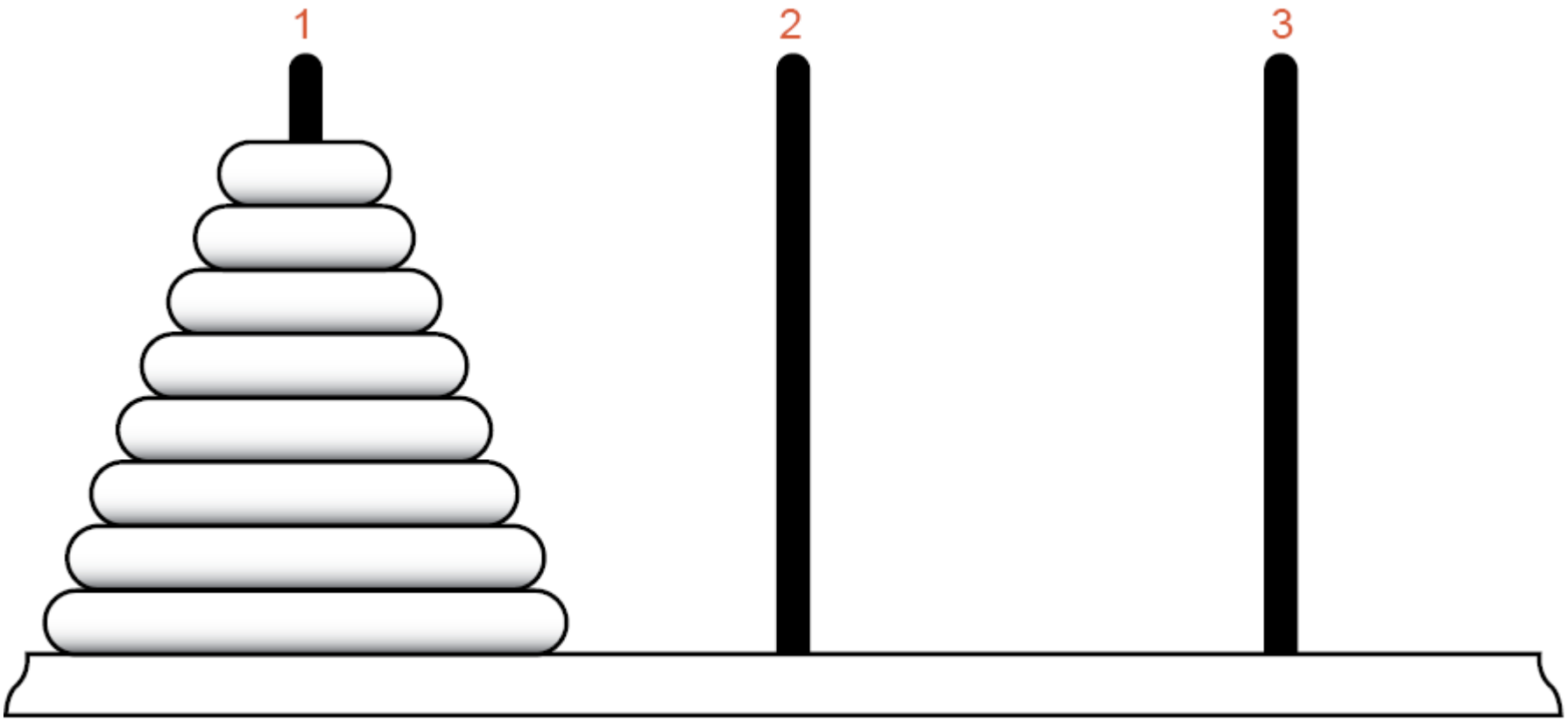
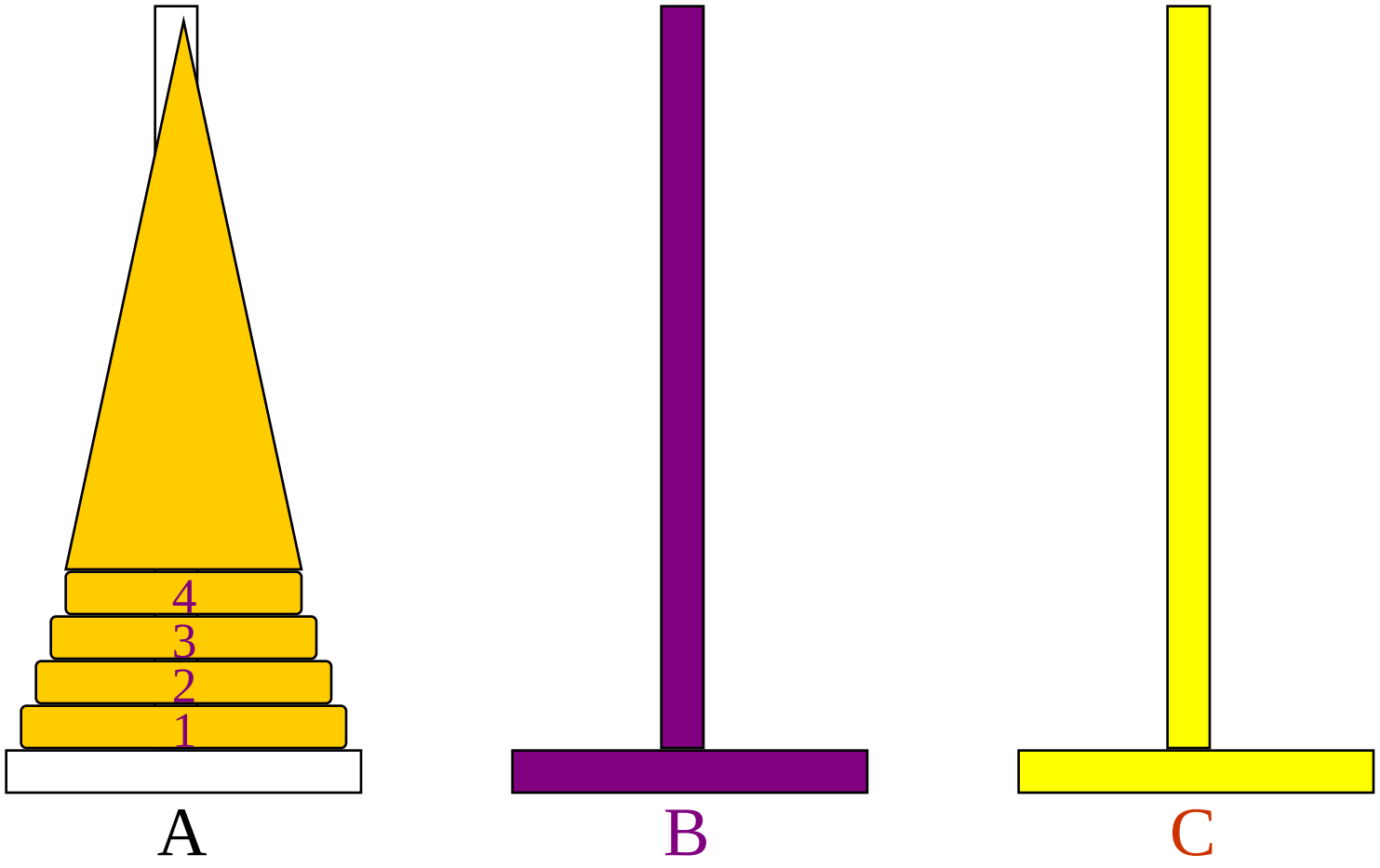


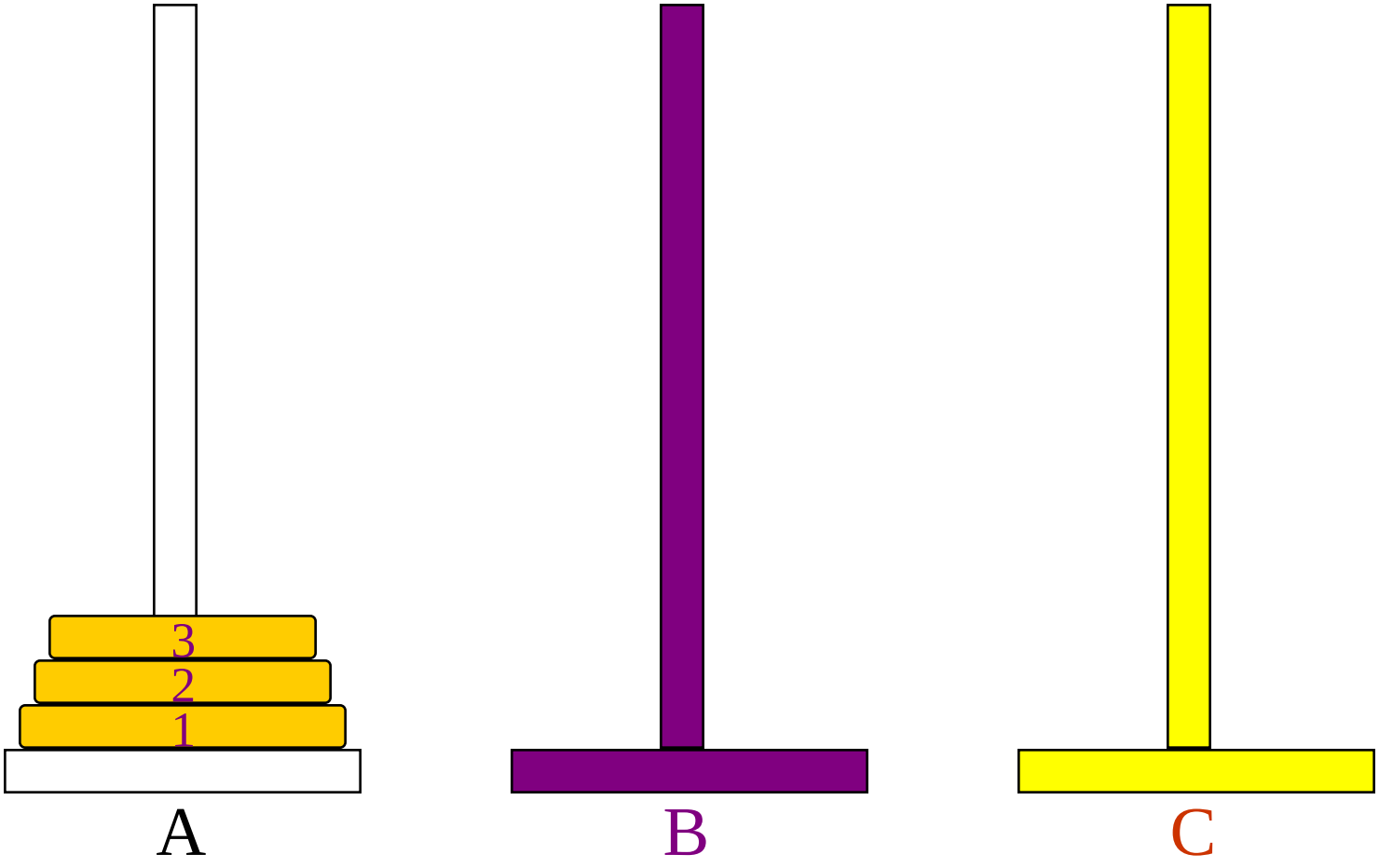
Figure 5.3. The Towers of Hanoi

Towers Of Hanoi/Brahma



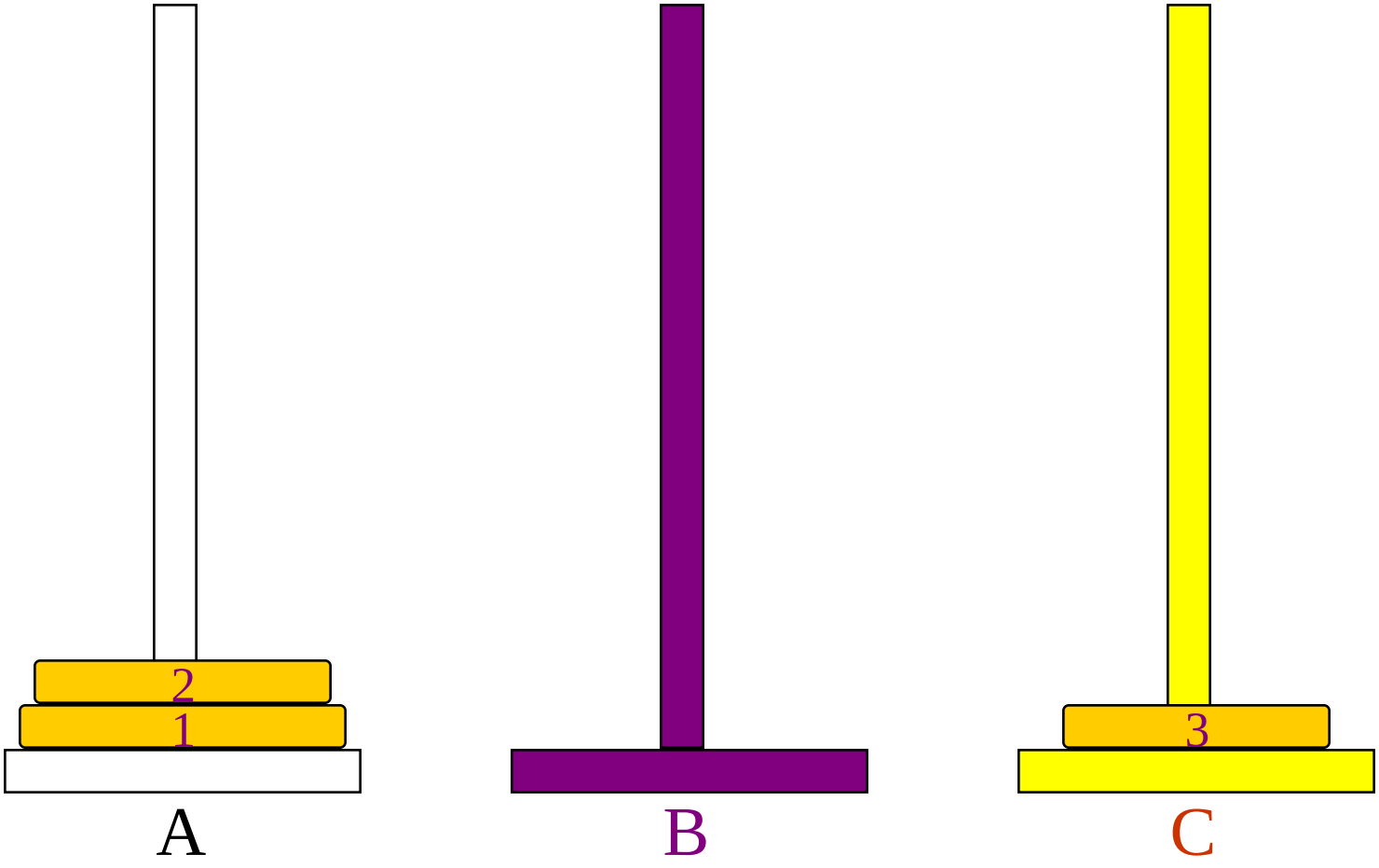
64 gold disks to be moved from tower A to tower ; each tower operates as a stack; cannot place big disk on top of a smaller one

Towers Of Hanoi/Brahma



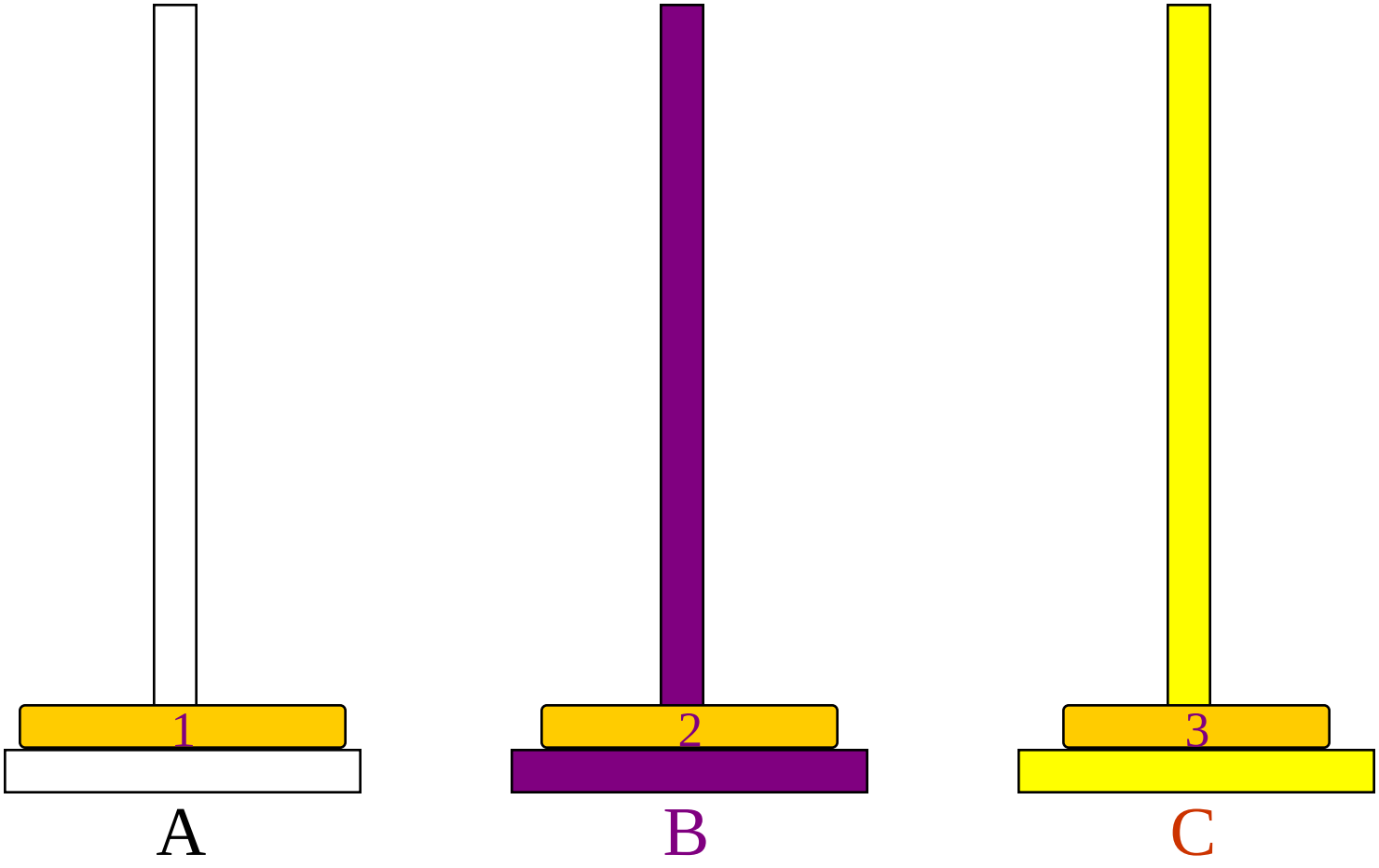
3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



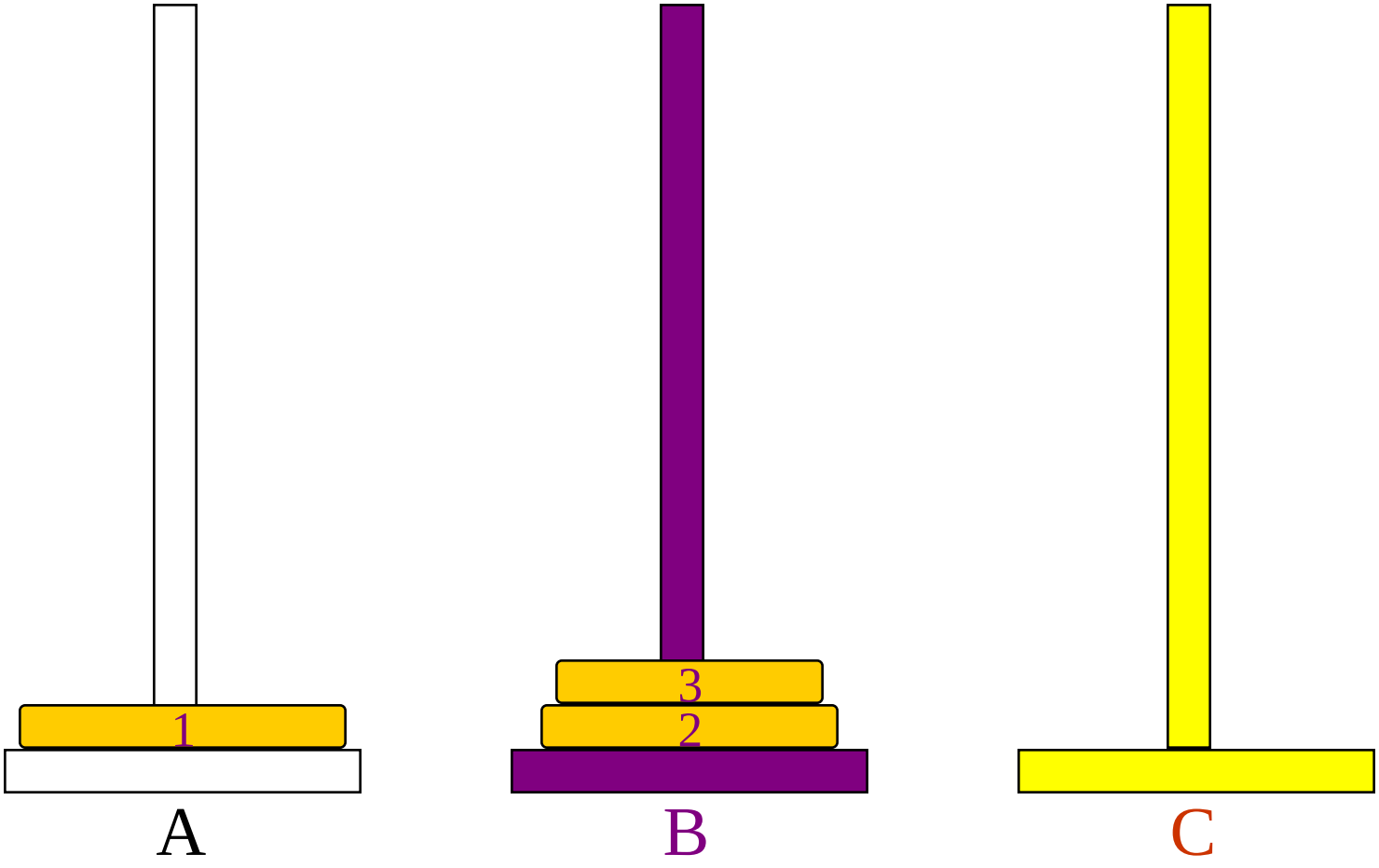
3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



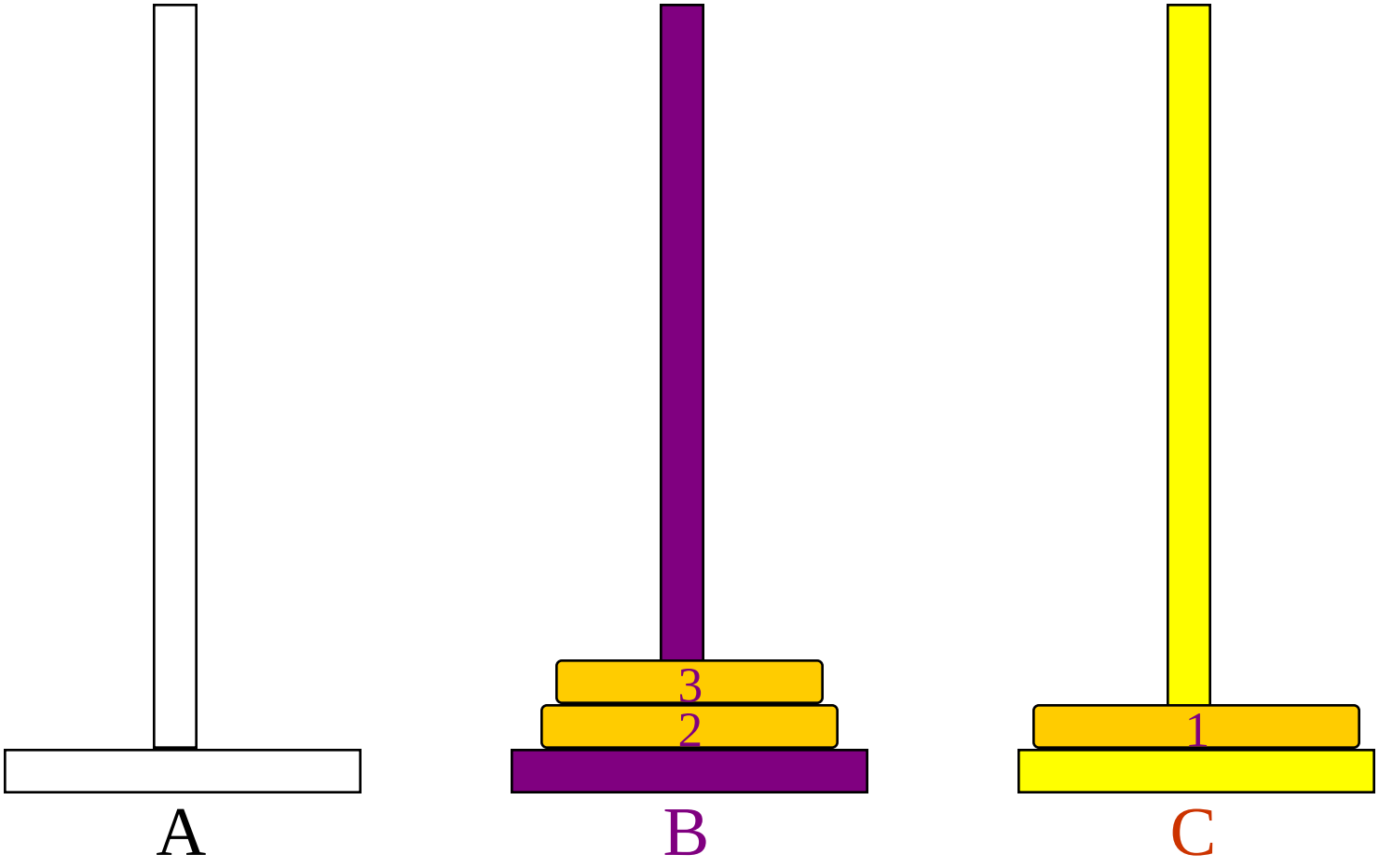
3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



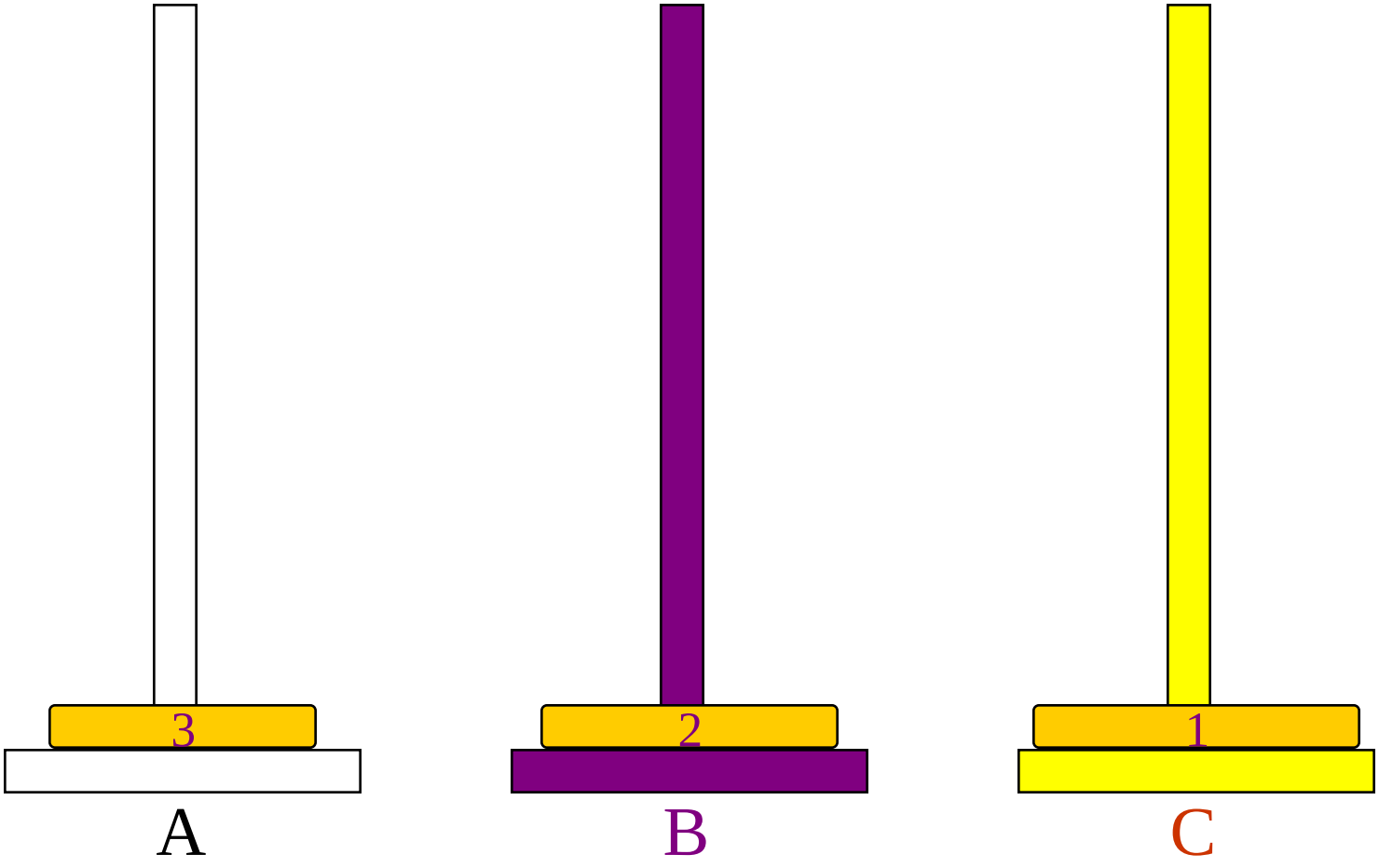
3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



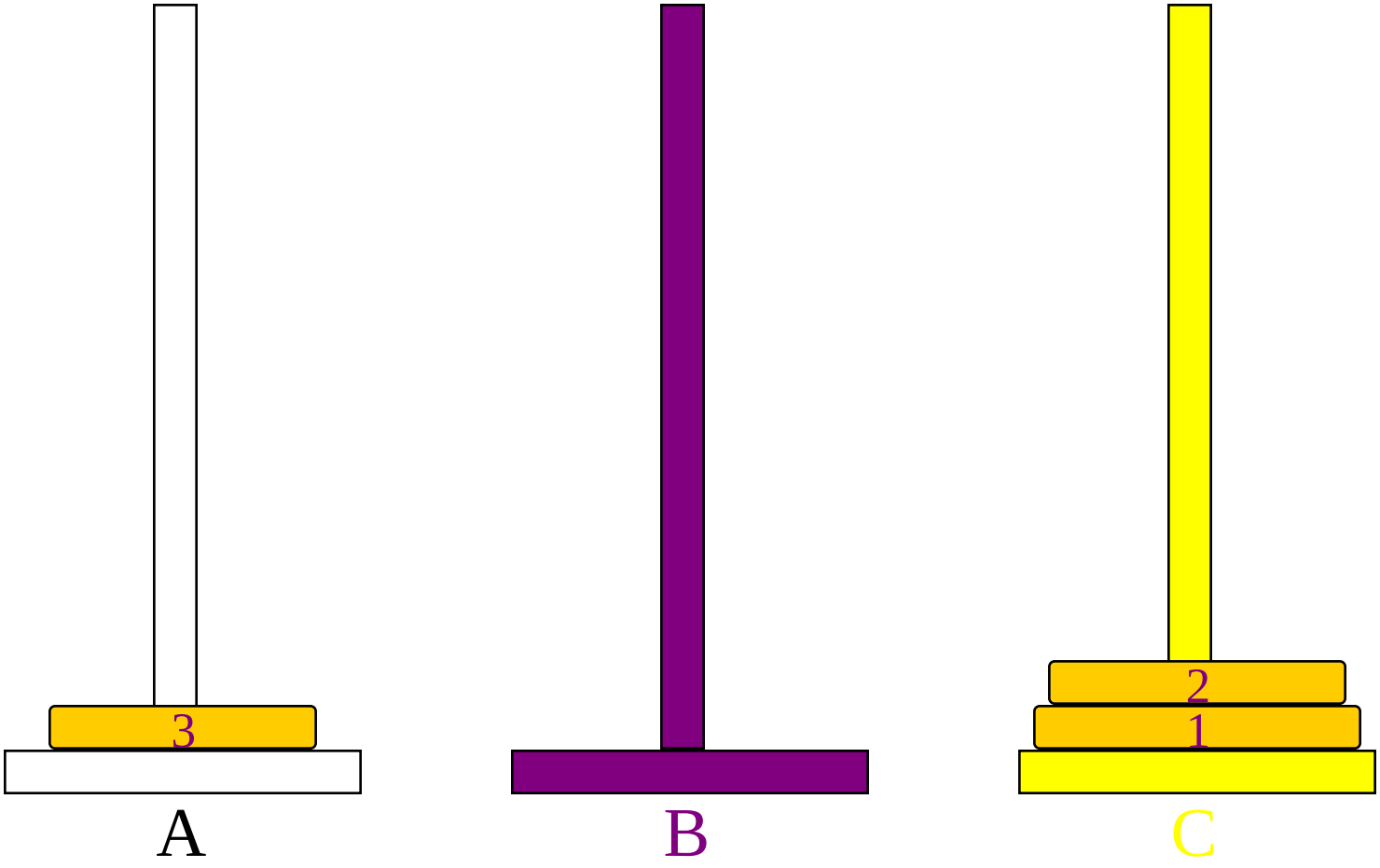
3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



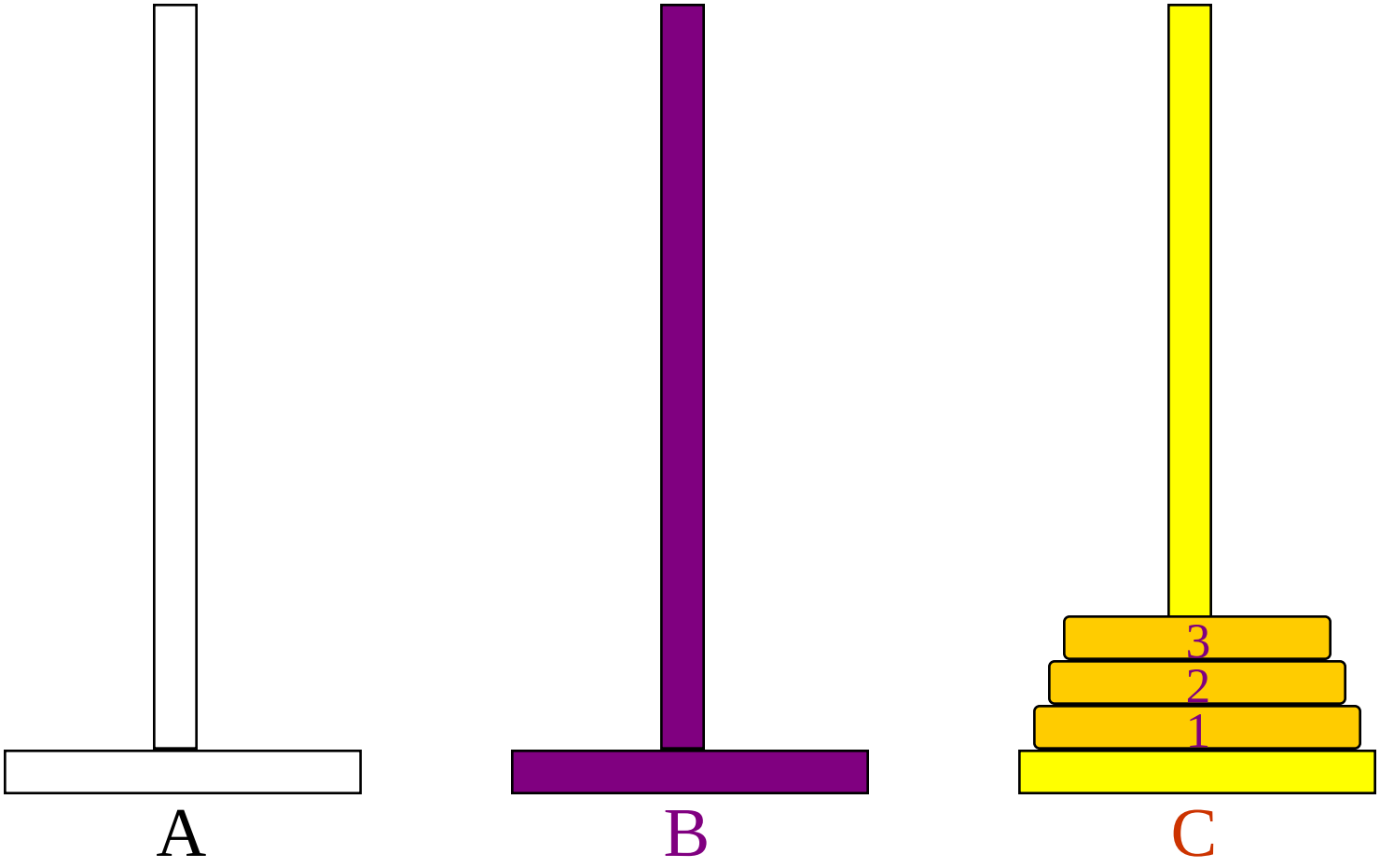
3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



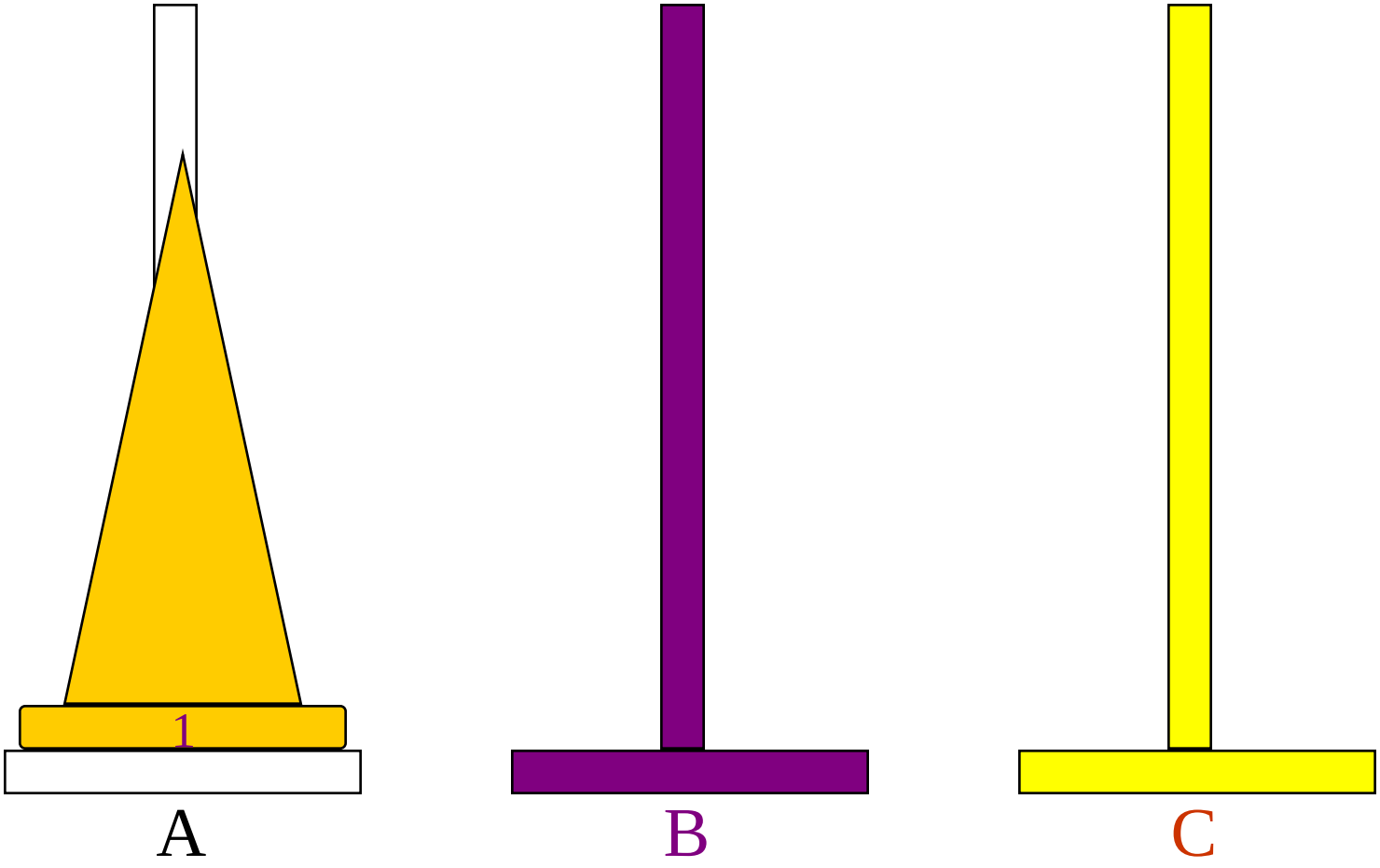
3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



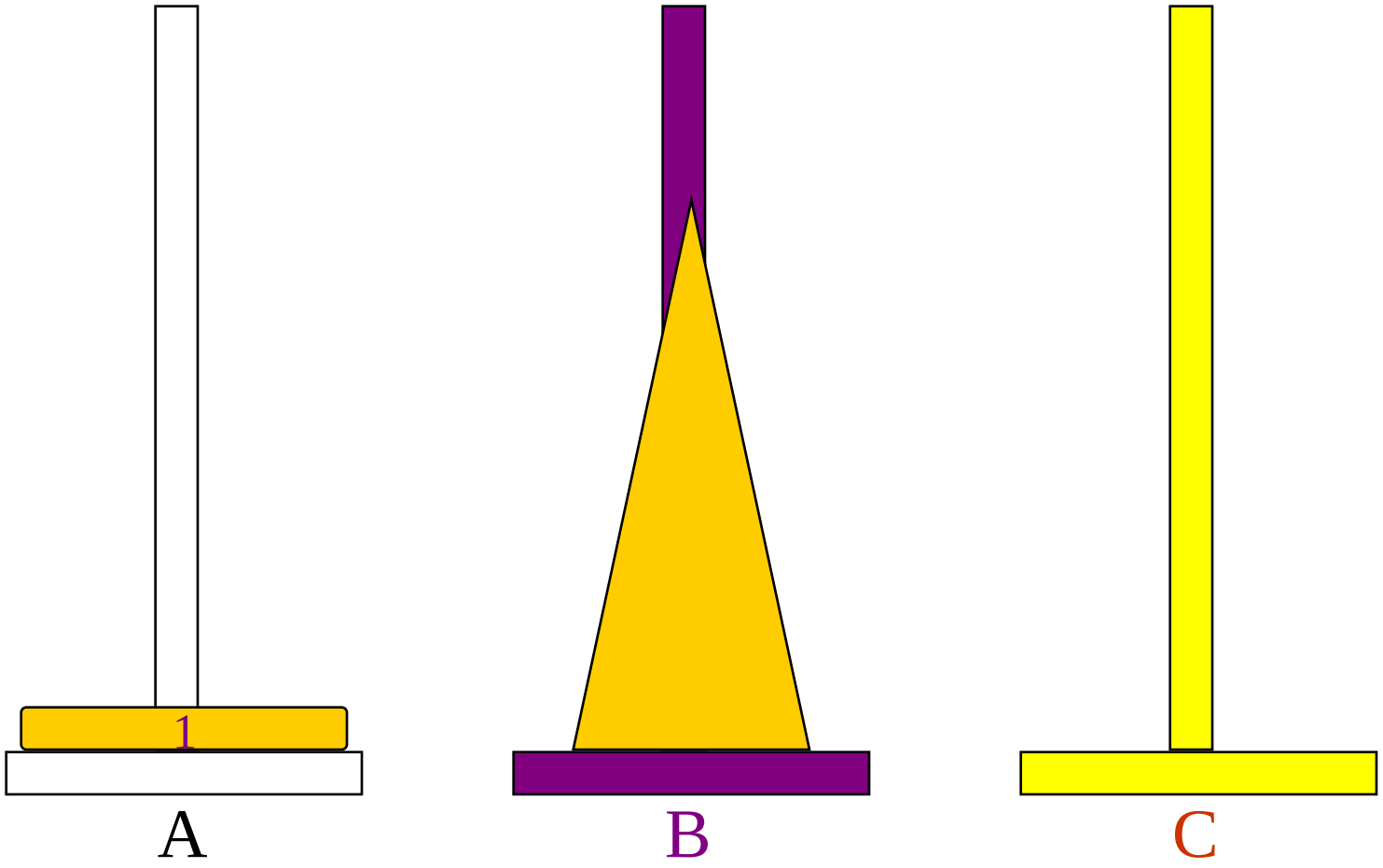
- 3-disk Towers Of Hanoi/Brahma;
- 7 disk moves

Towers Of Hanoi/Brahma



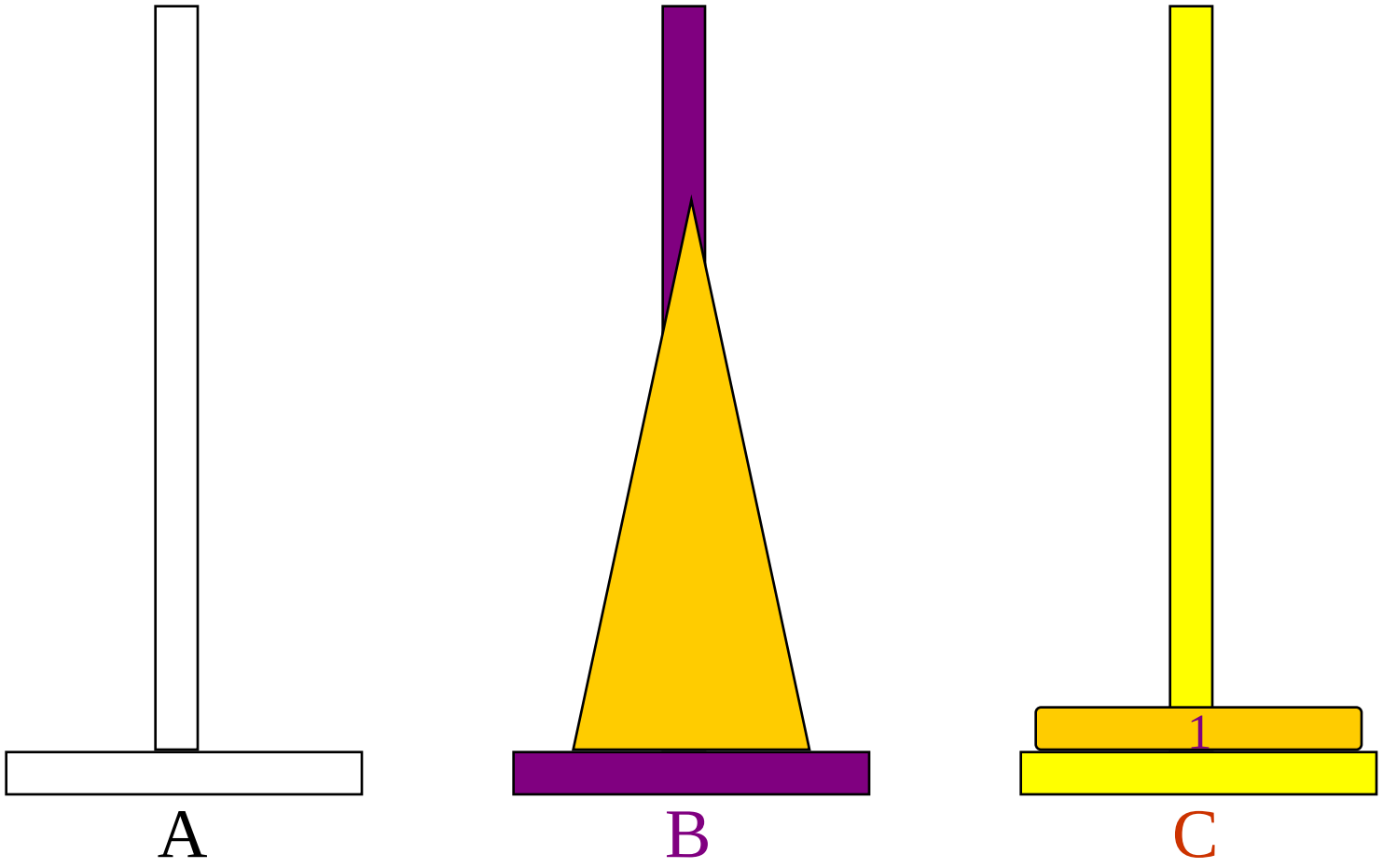
- n > 0 gold disks to be moved from A to C using B
- move top n-1 disks from A to B using C

Towers Of Hanoi/Brahma



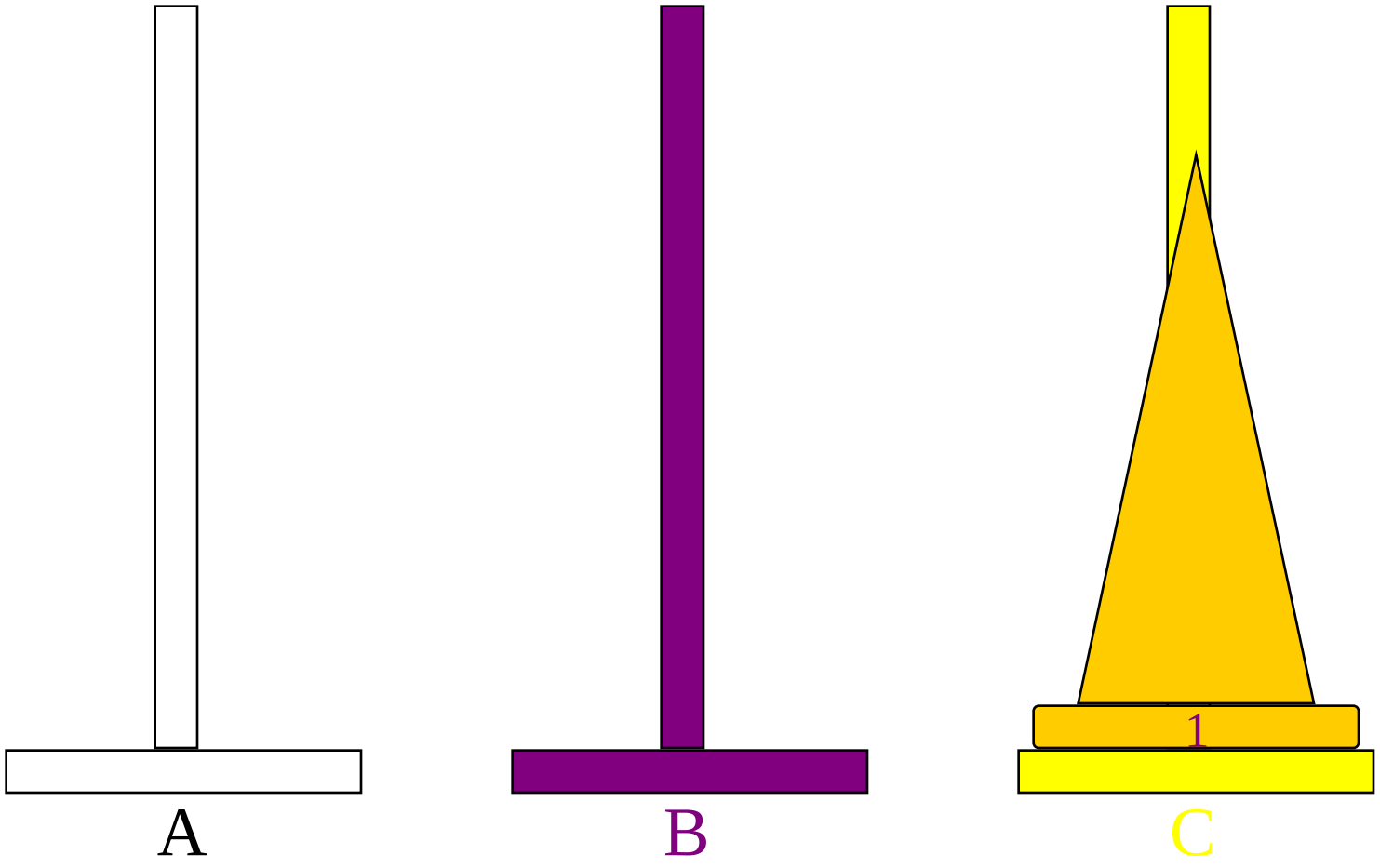
- move top disk from A to C

Towers Of Hanoi/Brahma



- move top n-1 disks from B to C using A

Towers Of Hanoi/Brahma



- moves(n) = 0 when n = 0
- moves(n) = 2*moves(n-1) + 1 = 2^n-1 when n > 0

5.1.4 Divide and Conquer: Hanoi

2. The Solution

```
move(64, 1, 3, 2)

    //    Move 63 disks from tower 1 to 2 (tower 3 temporary).
    move(63, 1, 2, 3);

    cout << "Move disk 64 from tower 1 to tower 3." << endl;

//    Move 63 disks from tower 2 to 3 (tower 1 temporary).
move(63, 2, 3, 1);
```

函数说明

```
void move(int count, int start, int finish, int temp);
```

5.1.4 Divide and Conquer: Hanoi

3. Refinement

```
const int disks = 64;

void move(int count, int start, int finish, int temp);

main()
{
   move(disks, 1, 3, 2);
}
```

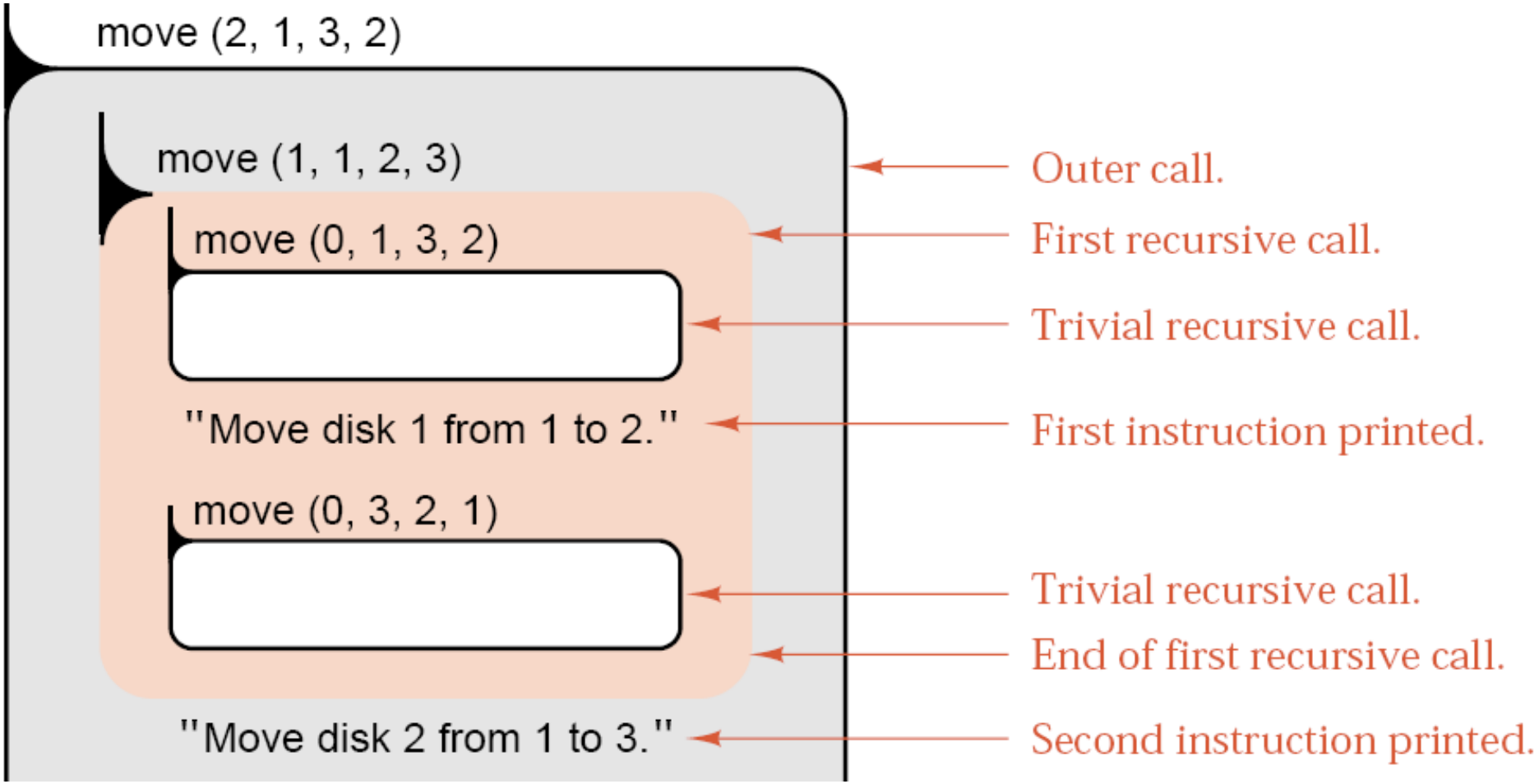
5.1.4 Divide and Conquer: Hanoi

3. Refinement

```
void move(int count, int start, int finish, int temp)
{
   if (count > 0) {
      move(count – 1, start, temp, finish);
      cout  ≪ "Move disk " ≪ count ≪ " from " ≪ start
            ≪ " to " ≪ finish ≪ "." ≪ endl;
      move(count – 1, temp, finish, start);
   }
}
```

5.1.4 Divide and Conquer: Hanoi

4. Program Tracing



5.1.4 Divide and Conquer: Hanoi

4. Program Tracing

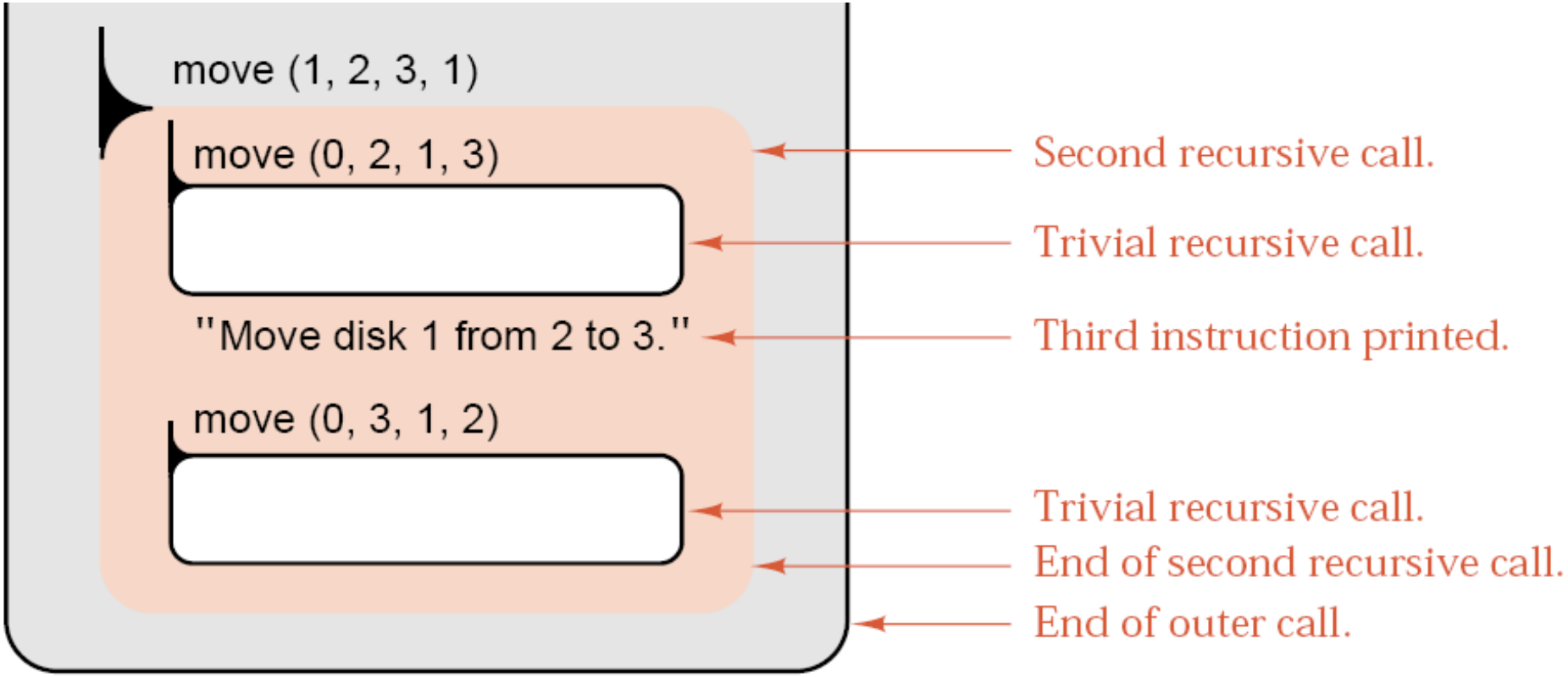


Figure 5.4. Trace of Hanoi for disks == 2

5.1.4 Divide and Conquer: Hanoi

5. Analysis

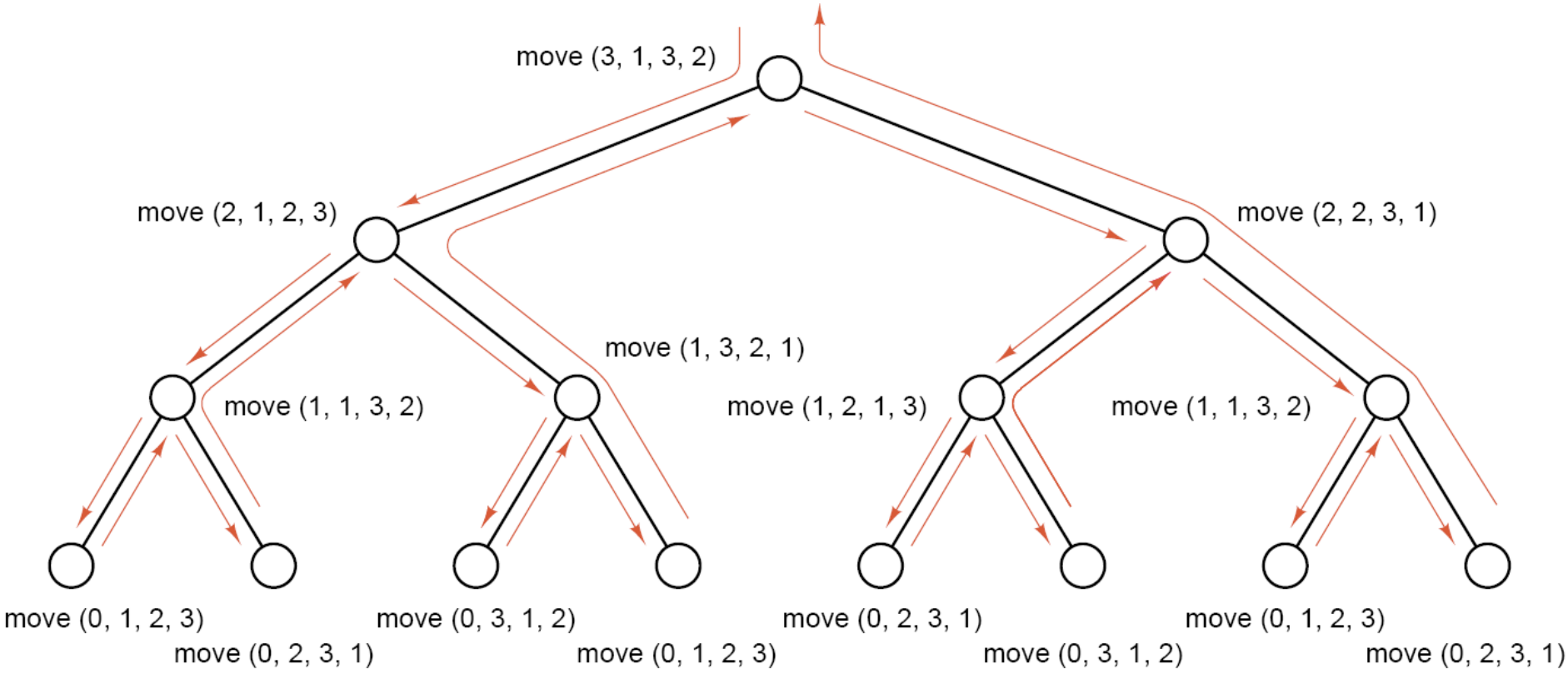


Figure 5.5. Recursion tree for three disks

5.1.4 Divide and Conquer: Hanoi

5. Analysis

$$1 + 2 + 4 + \cdots + 2^{63} = 2^0 + 2^1 + 2^2 + \cdots + 2^{63} = 2^{64} - 1.$$

$$10^3 = 1000 \approx 1024 = 2^{10}.$$

$$2^{64} = 2^4 \times 2^{60} \approx 2^4 \times 10^{18} = 1.6 \times 10^{19}.$$

There are about 3.2×10^7 seconds in one year. Suppose that the instructions could be carried out at the rather frenetic rate of one every second. (The priests have plenty of practice.) The total task will then take about 5×10^{11} years. Astronomers estimate the age of the universe at less than 20 billion (2×10^{10}) years, so, according to this story, the world will indeed endure a long time—25 times as long as it already has!

5.2.1 Designing Recursive Algorithms

- Find the key step
- Find a stopping rule
- Outline your algorithm
- Check termination
- Draw a recursion tree.

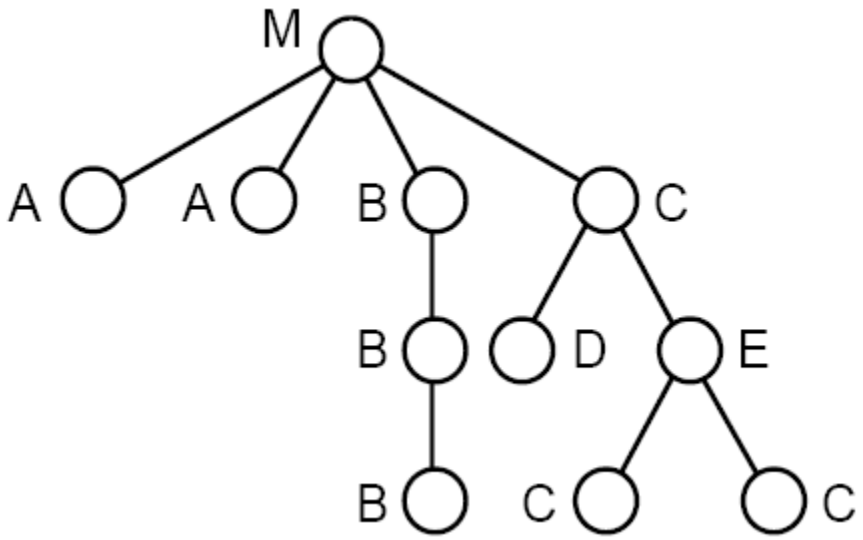
5.2.2 How Recursion Works

- Multiple Processors: Concurrency
- Single-Processor Implementation: Storage Areas
- Re-Entrant Programs
- Data Structures: Stacks and Trees

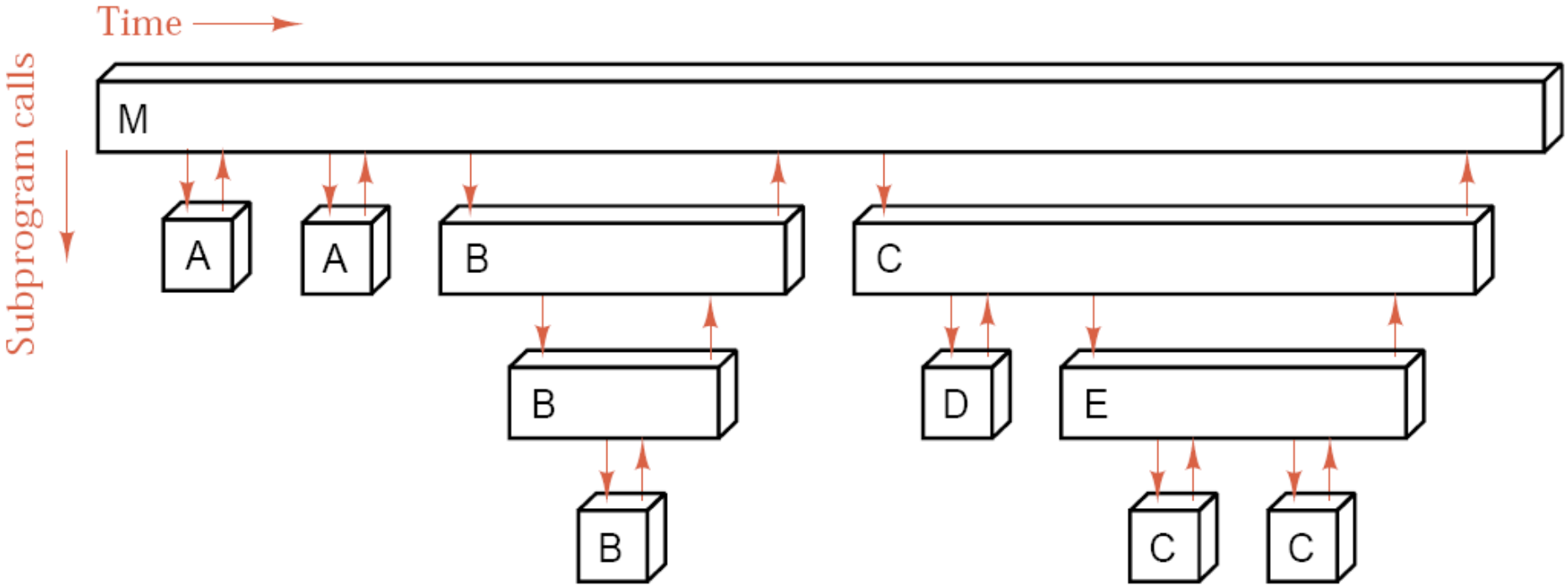
5.2.2 How Recursion Works

The *time* requirement of the program is related to the number of times functions are done, and therefore to the total number of vertices in the tree, but the *space* requirement is only that of the storage areas on the path from a single vertex back to the root. Thus the space requirement is reflected in the height of the tree. A well-balanced, bushy recursion tree signifies a recursive process that can do much work with little need for extra space.

Tree of subprogram calls



5.2.2 How Recursion Works



5.2.2 How Recursion Works

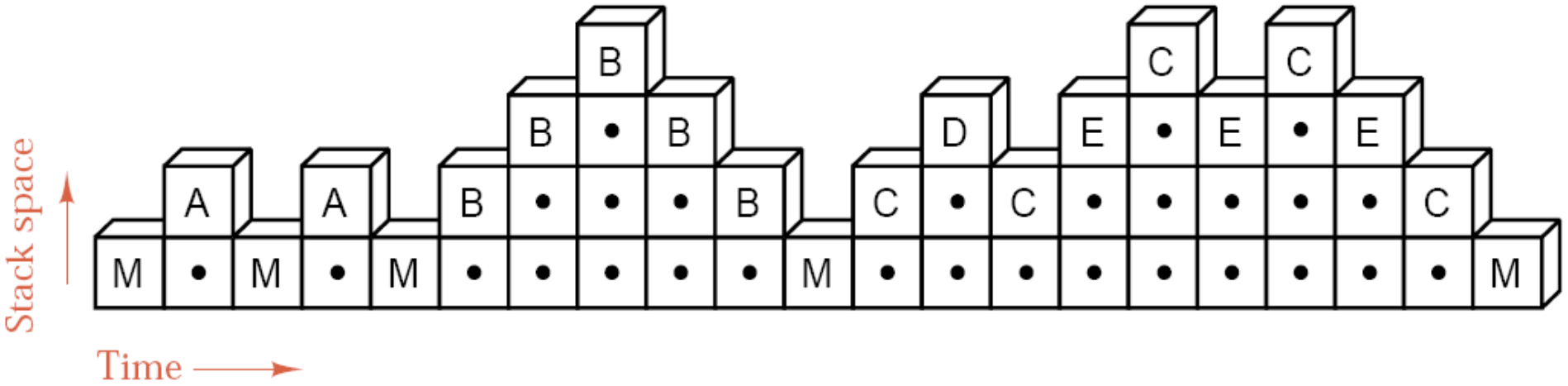


Figure 5.7. A tree of function calls and the associated stack frames

5.2.3 Tail Recursion

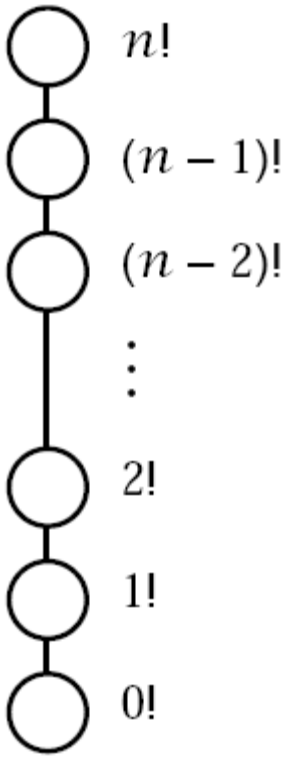
```
void move(int count, int start, int finish, int temp)
{
   if (count > 0) {
      move(count – 1, start, temp, finish);
      cout  ≪ "Move disk " ≪ count ≪ " from " ≪ start
            ≪ " to " ≪ finish ≪ "." ≪ endl;
      move(count – 1, temp, finish, start);
   }
}
```

5.2.3 Tail Recursion

```
void move(int count, int start, int finish, int temp)
/* move: iterative version
   Pre:  Disk count is a valid disk to be moved.
   Post: Moves count disks from start to finish using temp for temporary storage. */
{
   int swap;                          //   temporary storage to swap towers
   while (count > 0) {                //   Replace the if statement with a loop.
      move(count - 1, start, temp, finish);   //   first recursive call
      cout << "Move disk " << count << " from " << start
           << " to " << finish << "." << endl;
      count--;                        //   Change parameters to mimic the second recursive call.
      swap = start;
      start = temp;
      temp = swap;
   }
}
```

5.2.4 When Not to Use Recursion

1. Factorials




```
int factorial(int n)
/* factorial: recursive version
   Pre:  n is a nonnegative integer.
   Post: Return the value of the factorial of n. */
{
   if (n ==  0) return 1;
   else         return n * factorial(n – 1);
}
```

5.2.4 When Not to Use Recursion

$$
\begin{aligned}
\text{factorial}(5) &= 5 * \text{factorial}(4) \\
&= 5 * (4 * \text{factorial}(3)) \\
&= 5 * (4 * (3 * \text{factorial}(2))) \\
&= 5 * (4 * (3 * (2 * \text{factorial}(1)))) \\
&= 5 * (4 * (3 * (2 * (1 * \text{factorial}(0))))) \\
&= 5 * (4 * (3 * (2 * (1 * 1)))) \\
&= 5 * (4 * (3 * (2 * 1))) \\
&= 5 * (4 * (3 * 6)) \\
&= 5 * (4 * 6) \\
&= 5 * 24 \\
&= 120.
\end{aligned}
$$

5.2.4 When Not to Use Recursion

```
int factorial(int n)
/* factorial: iterative version
   Pre:  n is a nonnegative integer.
   Post: Return the value of the factorial of n. */
{
   int count, product = 1;
   for (count = 1; count <= n; count++)
      product *= count;
   return product;
}
```

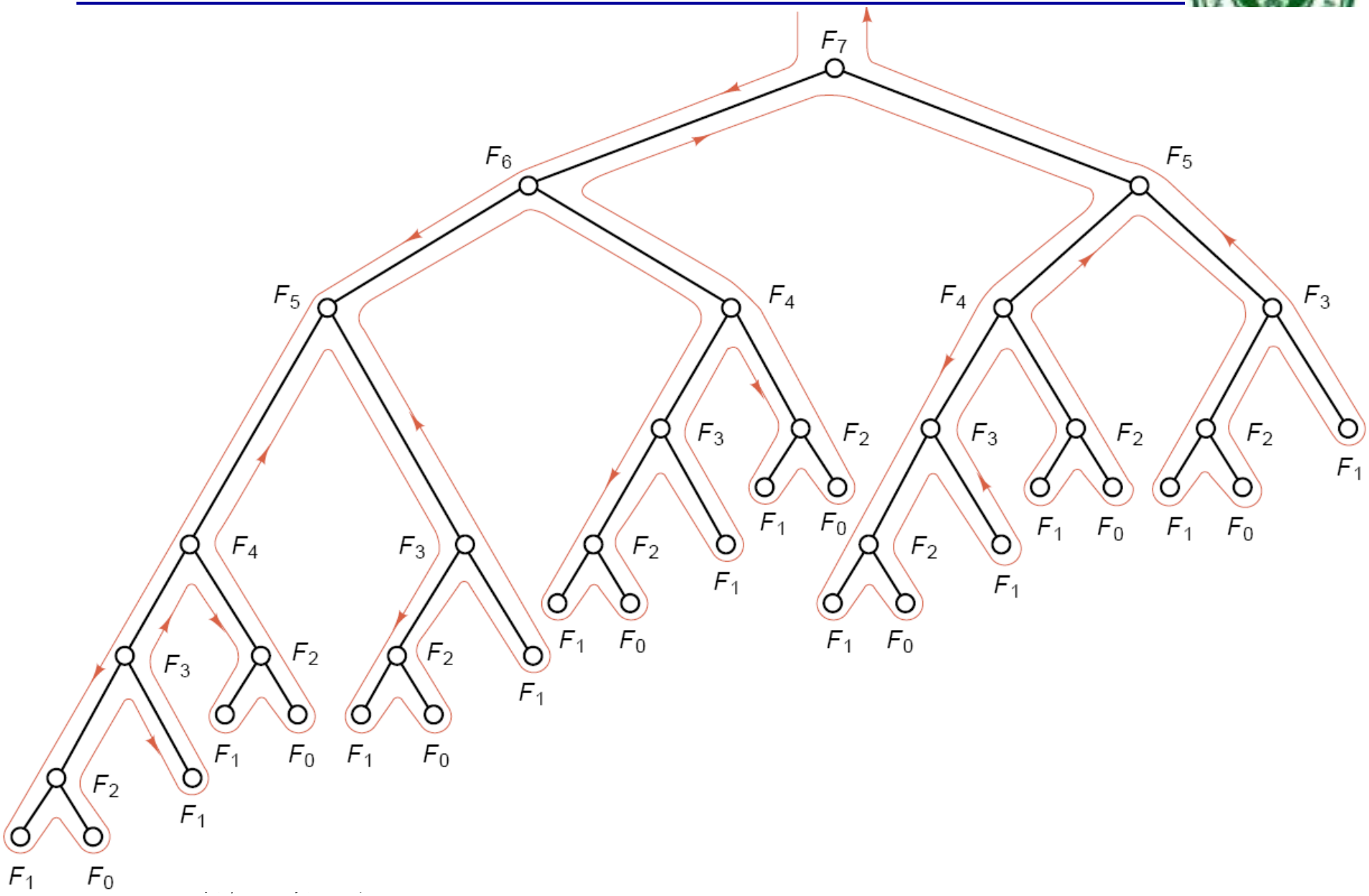
5.2.4 When Not to Use Recursion

2. Fibonacci Numbers

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2.$$

```
int fibonacci(int n)
/* fibonacci: recursive version
   Pre:  The parameter n is a nonnegative integer.
   Post: The function returns the nth Fibonacci number. */
{
   if (n <= 0)          return 0;
   else if (n == 1)     return 1;
   else                 return fibonacci(n - 1) + fibonacci(n - 2);
}
```

5.2.4 When Not to Use Recursion



5.2.4 When Not to Use Recursion

```
int fibonacci(int n)

/* fibonacci: iterative version
   Pre:  The parameter n is a nonnegative integer.
   Post: The function returns the nth Fibonacci number. */
{
   int last_but_one;          //  second previous Fibonacci number, F_{i-2}
   int last_value;            //  previous Fibonacci number, F_{i-1}
   int current;               //  current Fibonacci number F_i
   if (n <= 0) return 0;
   else if (n ==  1) return 1;
   else {
      last_but_one = 0;
      last_value = 1;
      for (int i = 2;  i <= n;  i++) {
         current = last_but_one + last_value;
         last_but_one = last_value;
         last_value = current;
      }
      return current;
   }
}
```

5.3 Backtracking: Postponing the work

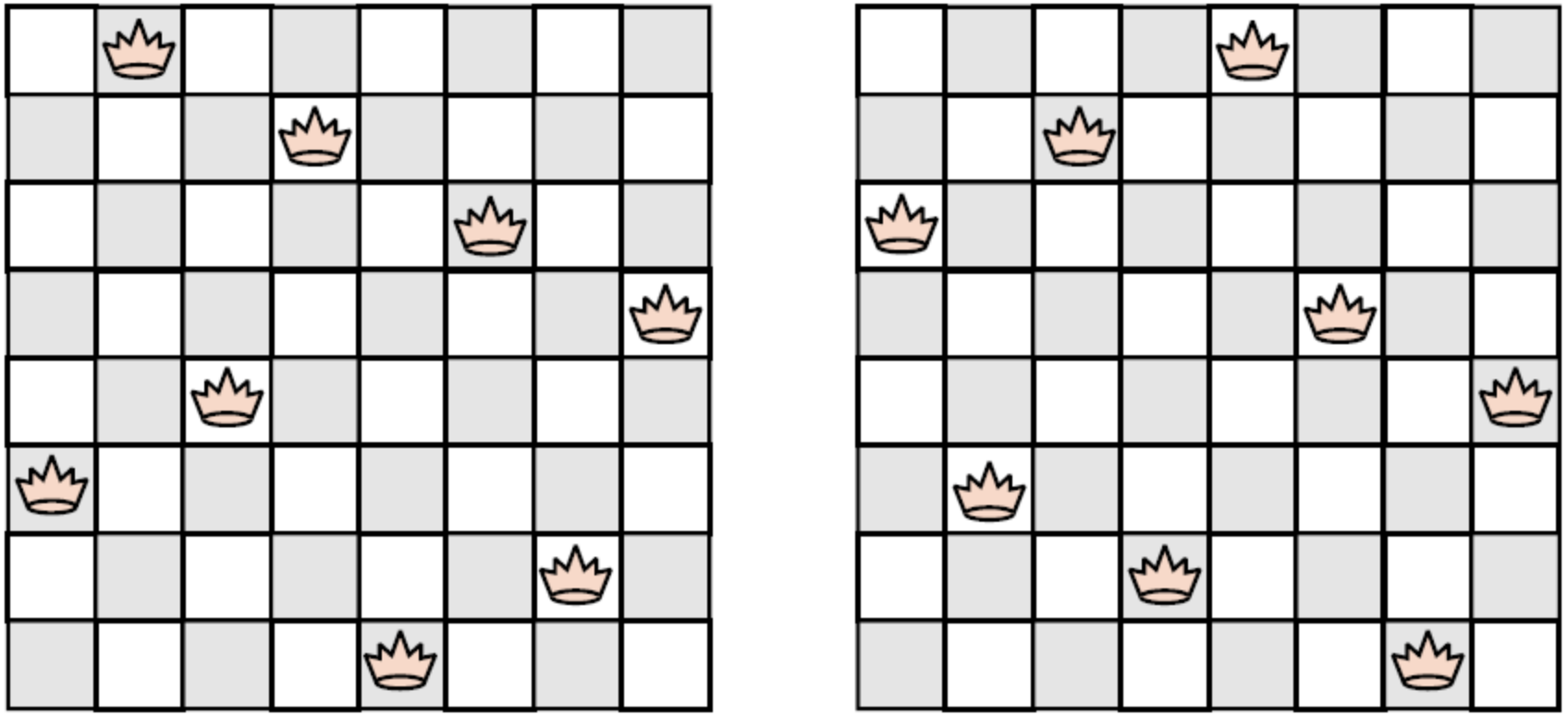


Figure 5.12. Two configurations showing eight nonattacking queens

5.3.1 Solving the Eight-Queens Puzzle

```
solve_from (Queens configuration)
  if Queens configuration already contains eight queens
      print configuration
  else
      for every chessboard square p that is unguarded by configuration {
          add a queen on square p to configuration;
          solve_from(configuration);
          remove the queen from square p of configuration;
       }
```

5.3.2 Example: Four Queens

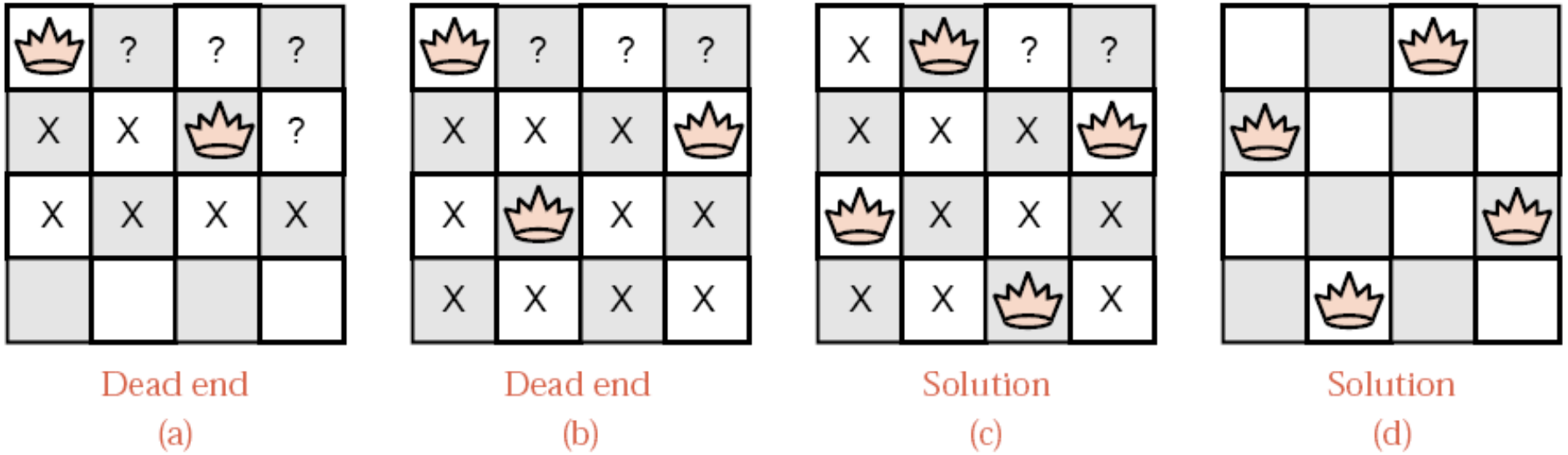


Figure 5.13. Solution to the four-queens problem

5.3.2 Example: Four Queens

```
const int max_board = 30;

class Queens {
public:
   Queens(int size);
   bool is_solved( ) const;
   void print( ) const;
   bool unguarded(int col) const;
   void insert(int col);
   void remove(int col);
   int board_size;     //    dimension of board = maximum number of queens
private:
   int count;     //    current number of queens = first unoccupied row
   bool queen_square[max_board] [max_board];
};
```

5.3.2 Example: Four Queens

```
Queens :: Queens(int size)
/* Post: The Queens object is set up as an empty configuration on a chessboard
         with size squares in each row and column. */
{
   board_size = size;
   count = 0;
   for (int row = 0; row < board_size; row++)
      for (int col = 0; col < board_size; col++)
         queen_square[row][col] = false;
}

void Queens :: insert(int col)
/* Pre:  The square in the first unoccupied row (row count) and column col is not
         guarded by any queen.
   Post: A queen has been inserted into the square at row count and column col;
         count has been incremented by 1. */
{
   queen_square[count++][col] = true;
}
```

5.3.2 Example: Four Queens

```
bool Queens::unguarded(int col) const
{
   int i;
   bool ok = true;
   for (i = 0; ok && i < count; i++)
      ok = !queen_square[i][col];
   for (i = 1; ok && count - i >= 0 && col - i >= 0; i++)
      ok = !queen_square[count - i][col - i];
   for (i = 1; ok && count - i >= 0 && col + i < board_size; i++)
      ok = !queen_square[count - i][col + i];
   return ok;
}
```

Upper part

Upper-left diagonal

Upper-right diagonal

5.3.2 Example: Four Queens

```
int main( )
{
   int board_size;
   print_information( );
   cout  ≪ "What is the size of the board? " ≪ flush;
   cin    ≫ board_size;
   if (board_size < 0 || board_size > max_board)
      cout ≪ "The number must be between 0 and " ≪ max_board ≪ endl;
   else {
      Queens configuration(board_size);   //    Initialize empty configuration.
      solve_from(configuration); //    Find all solutions extending configuration.
   }
}
```

5.3.2 Example: Four Queens

```
void solve_from(Queens &configuration)

{
   if (configuration.is_solved( )) configuration.print( );
   else
      for (int col = 0;  col < configuration.board_size;  col++)
         if (configuration.unguarded(col)) {
            configuration.insert(col);
            solve_from(configuration);
            configuration.remove(col);
         }
}
```

5.3.6 Review and Refinement

改进前

Size	8	9	10	11	12	13
Number of solutions	92	352	724	2680	14200	73712
Time (seconds)	0.05	0.21	1.17	6.62	39.11	243.05
Time per solution (ms.)	0.54	0.60	1.62	2.47	2.75	3.30

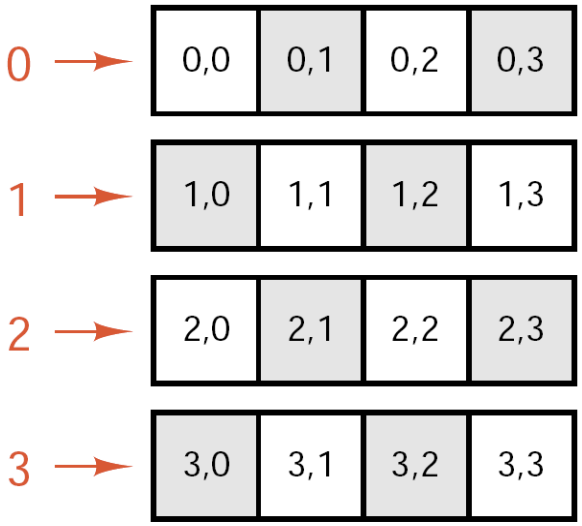
改进后

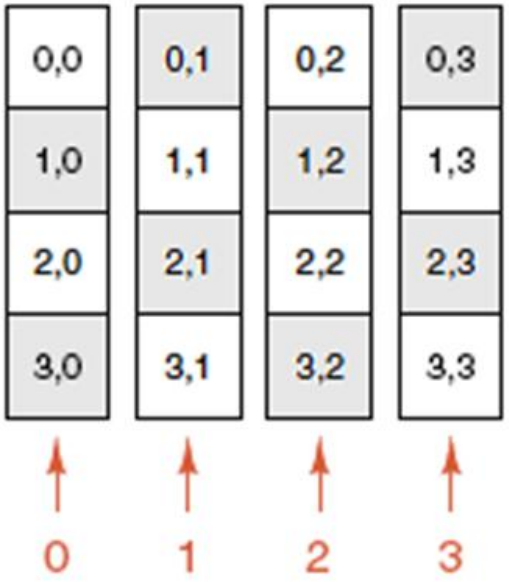
Size	8	9	10	11	12	13
Number of solutions	92	352	724	2680	14200	73712
Time (seconds)	0.01	0.05	0.22	1.06	5.94	34.44
Time per solution (ms.)	0.11	0.14	0.30	0.39	0.42	0.47

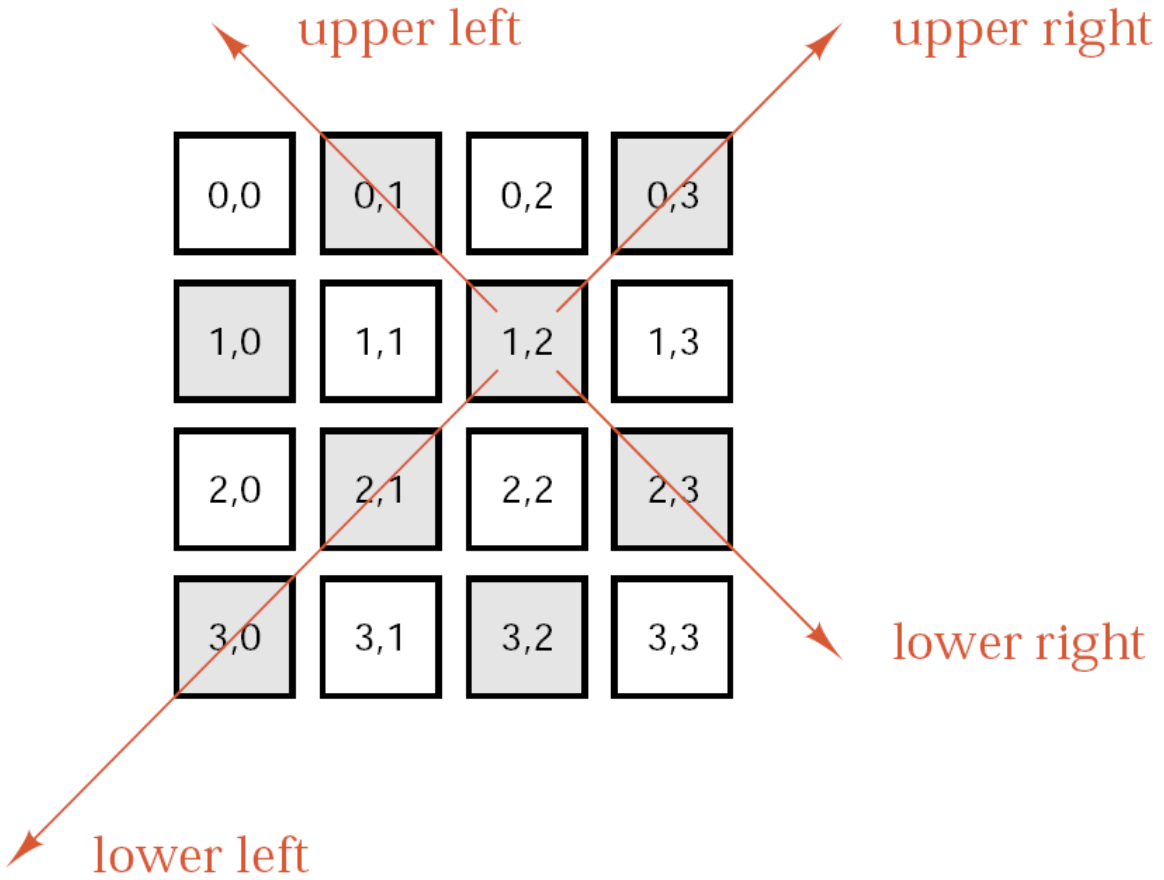
5.3.6 Review and Refinement

```
class Queens {
public:
    Queens(int size);
    bool is_solved( ) const;
    void print( ) const;
    bool unguarded(int col) const;
    void insert(int col);
    void remove(int col);
    int board_size;
private:
    int count;
    bool col_free[max_board];
    bool upward_free[2 * max_board – 1];
    bool downward_free[2 * max_board – 1];
    int queen_in_row[max_board]; //    column number of queen in each row
};
```

5.3.6 Review and Refinement

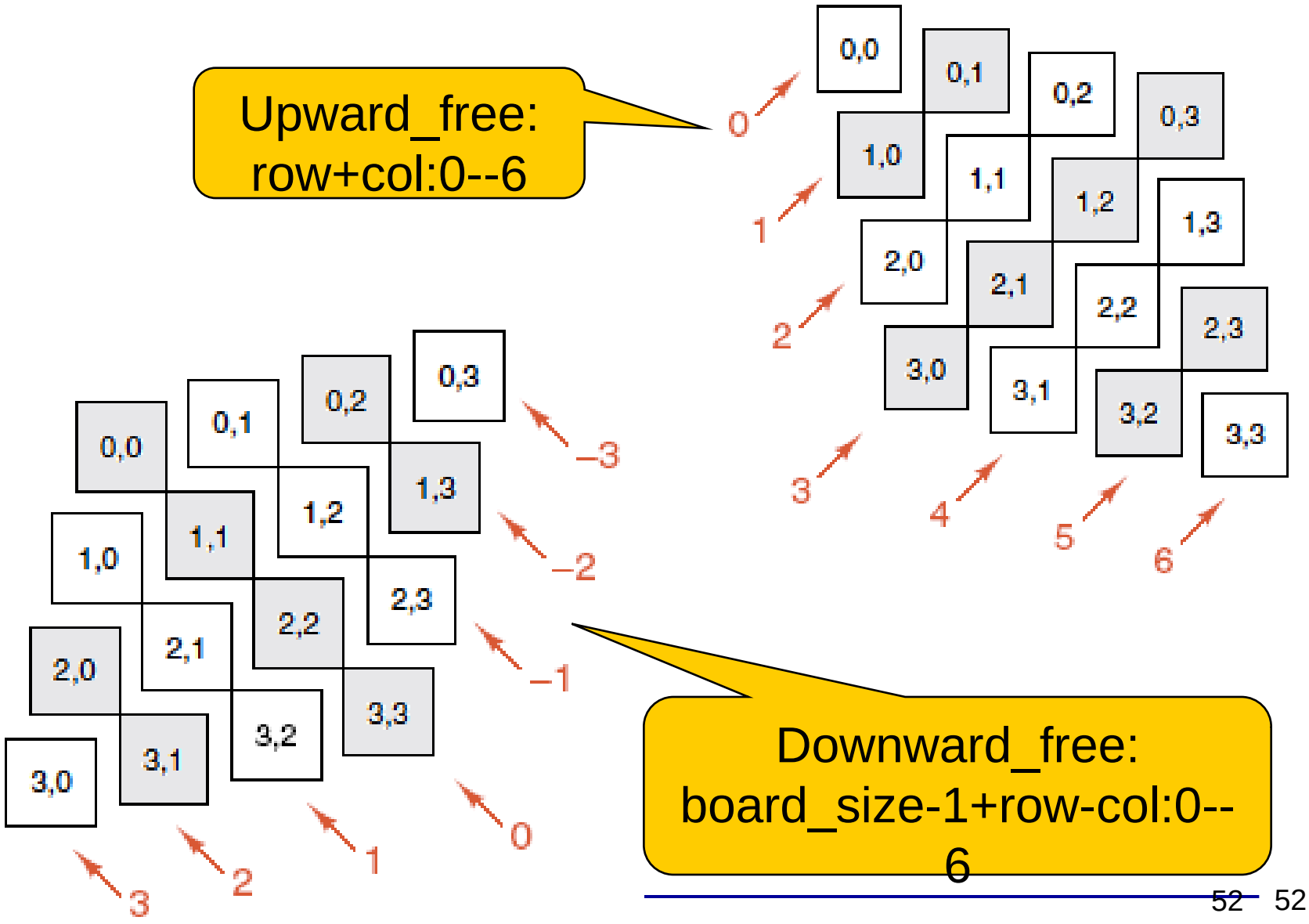






(c) Diagonal directions

5.3.6 Review and Refinement



5.3.6 Review and Refinement

```
Queens :: Queens(int size)
/* Post: The Queens object is set up as an empty configuration on a chessboard
         with size squares in each row and column. */
{
   board_size = size;
   count = 0;
   for (int i = 0; i < board_size; i++) col_free[i] = true;
   for (int j = 0; j < (2 * board_size - 1); j++) upward_free[j] = true;
   for (int k = 0; k < (2 * board_size - 1); k++) downward_free[k] = true;
}
```

5.3.6 Review and Refinement

```
void Queens :: insert(int col)
/* Pre:  The square in the first unoccupied row (row count) and column col is not
         guarded by any queen.
   Post: A queen has been inserted into the square at row count and column col;
         count has been incremented by 1. */
{
   queen_in_row[count] = col;
   col_free[col] = false;
   upward_free[count + col] = false;
   downward_free[count - col + board_size - 1] = false;
   count++;
}
```

5.3.6 Review and Refinement

```
bool Queens :: unguarded(int col) const
/* Post:  Returns true or false according as the square in the first unoccupied row
          (row count) and column col is not guarded by any queen. */
{
   return col_free[col]
          && upward_free[count + col]
          && downward_free[count - col + board_size - 1];
}
```

5.3.7 Analysis of Backtracking

1. Effectiveness of Backtracking

$$\binom{64}{8} = 4{,}426{,}165{,}368.$$

The observation that there can be only one queen in each row immediately cuts this number to

$$8^8 = 16{,}777{,}216.$$

This number is still large, but our program will not investigate nearly this many squares. Instead, it rejects squares whose column or diagonals are guarded. The requirement that there be only one queen in each column reduces the number to

$$8! = 40{,}320,$$

5.3.7 Analysis of Backtracking

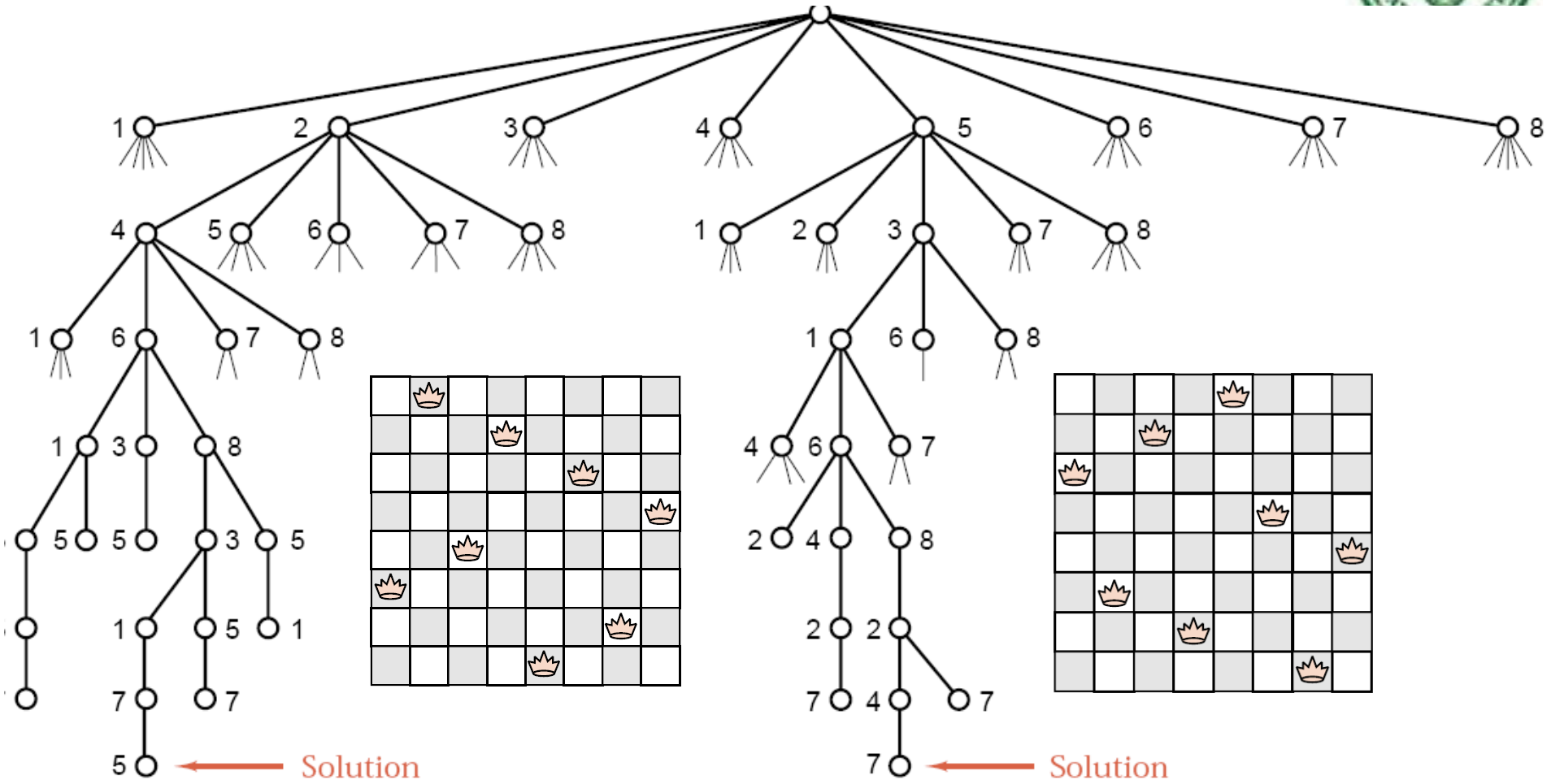


Figure 5.15. Part of the recursion tree, eight-queens problem

5.3.7 Analysis of Backtracking

2. Lower Bounds

For the second row it must investigate at least $n-3$ positions, for the third row $n-6$, and so on. Hence, to place a queen in each of the first $n/4$ rows, backtracking investigates a minimum of

$$n(n-3)(n-6)\ldots(n-3n/4) > (n/4)^{n/4}$$

1×3　　2×3　　(*n*/4)×3

But

$$\log((n/4)^{n/4}) / \log(2^n) = \log(n/4)/4\log(2).$$