Chapter 6 - Recursion

- Subprogram implementation
- Recursion
- Designing recursive algorithms
- Recursion removal
- Backtracking
- Examples of backtracking and recursive algorithms:
  - Factorial
  - Fibonacci
  - The towers of Hanoi
  - Eight Queens Problem
  - Tree-structured program: Look-ahead in Game

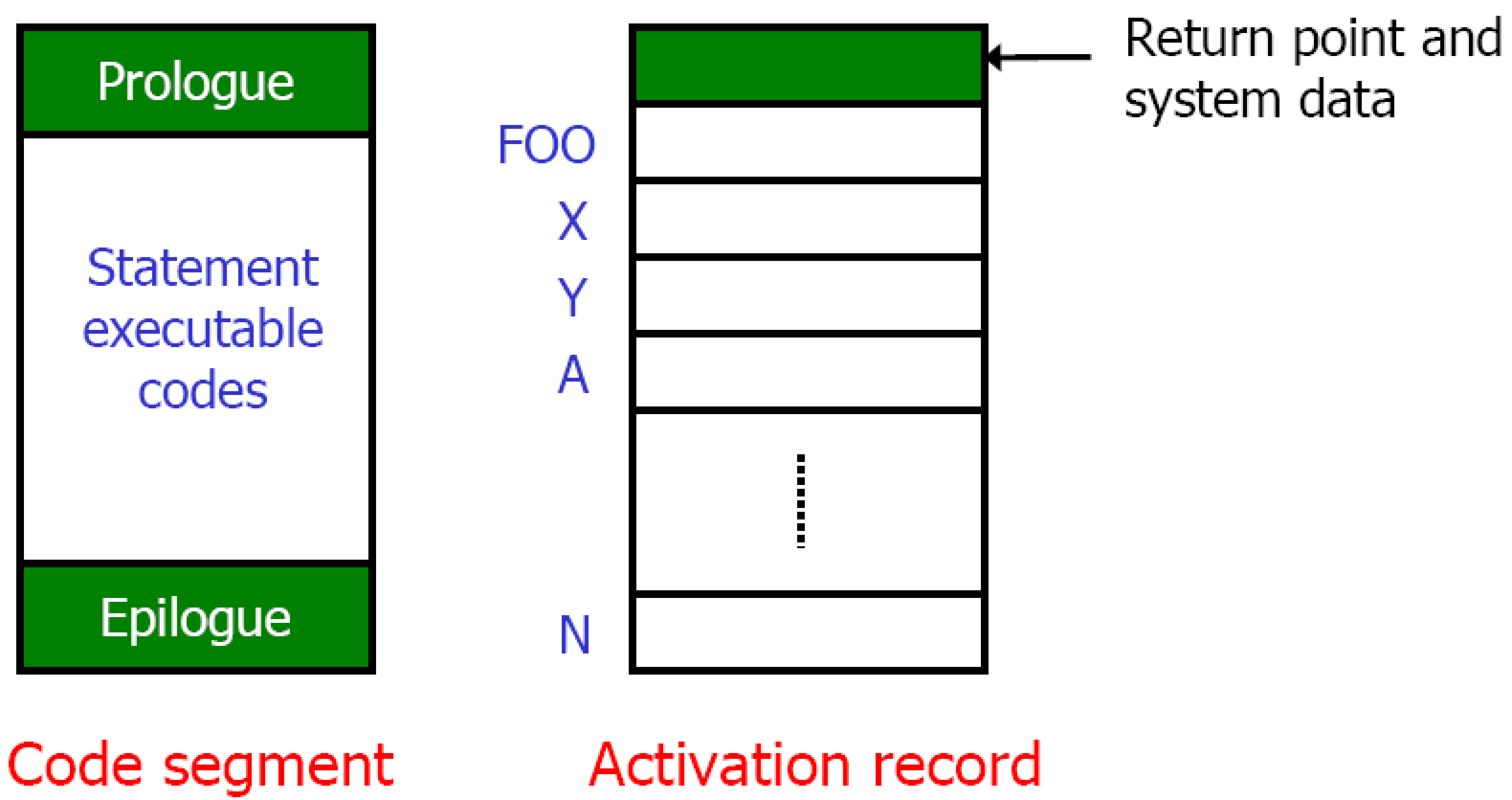
Subprogram implementation

```
function FOO(X: real; Y: integer): real;
    var A: array [1..10] of real;
        N: integer;
    begin
        ...
        N := Y + 1;
        X := A[N] * 2;
        ...
    end;
```

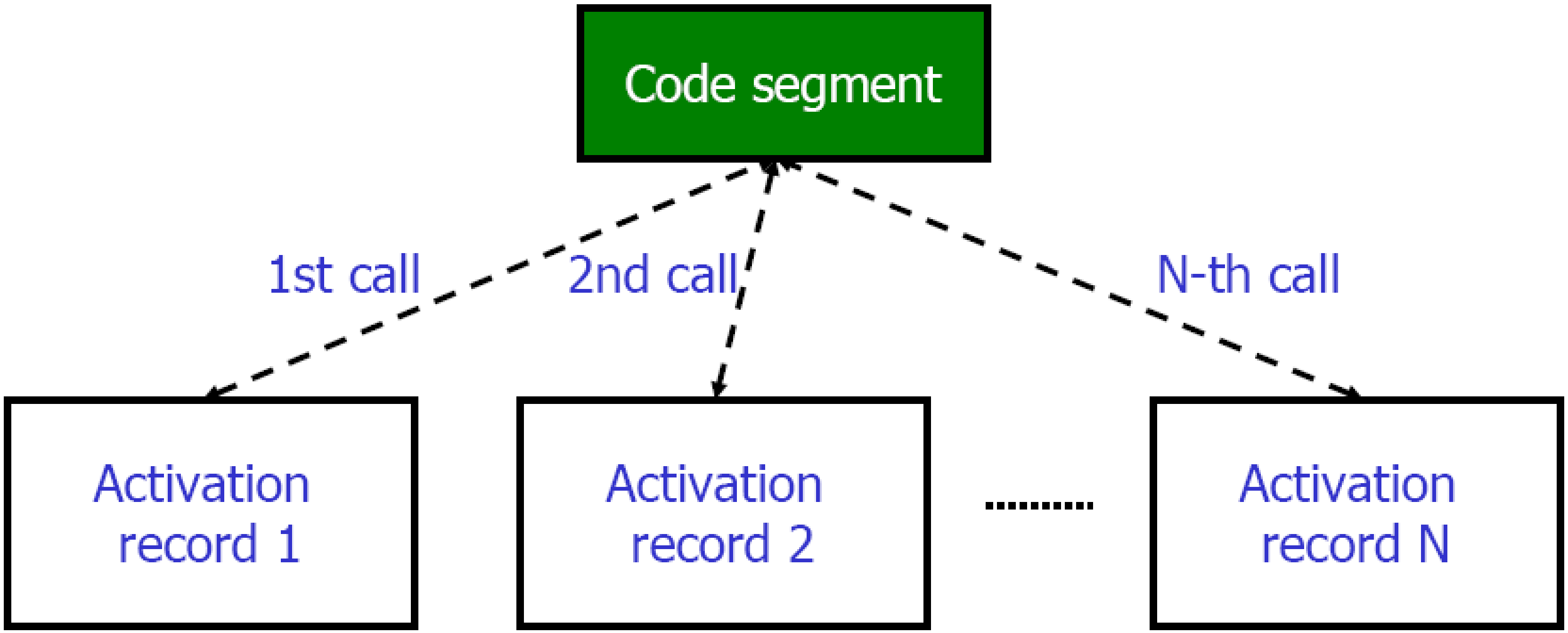
Subprogram implementation

- Code segment (static part)
- Activation record (dynamic part):
  - Parameters
  - Function results
  - Local variables

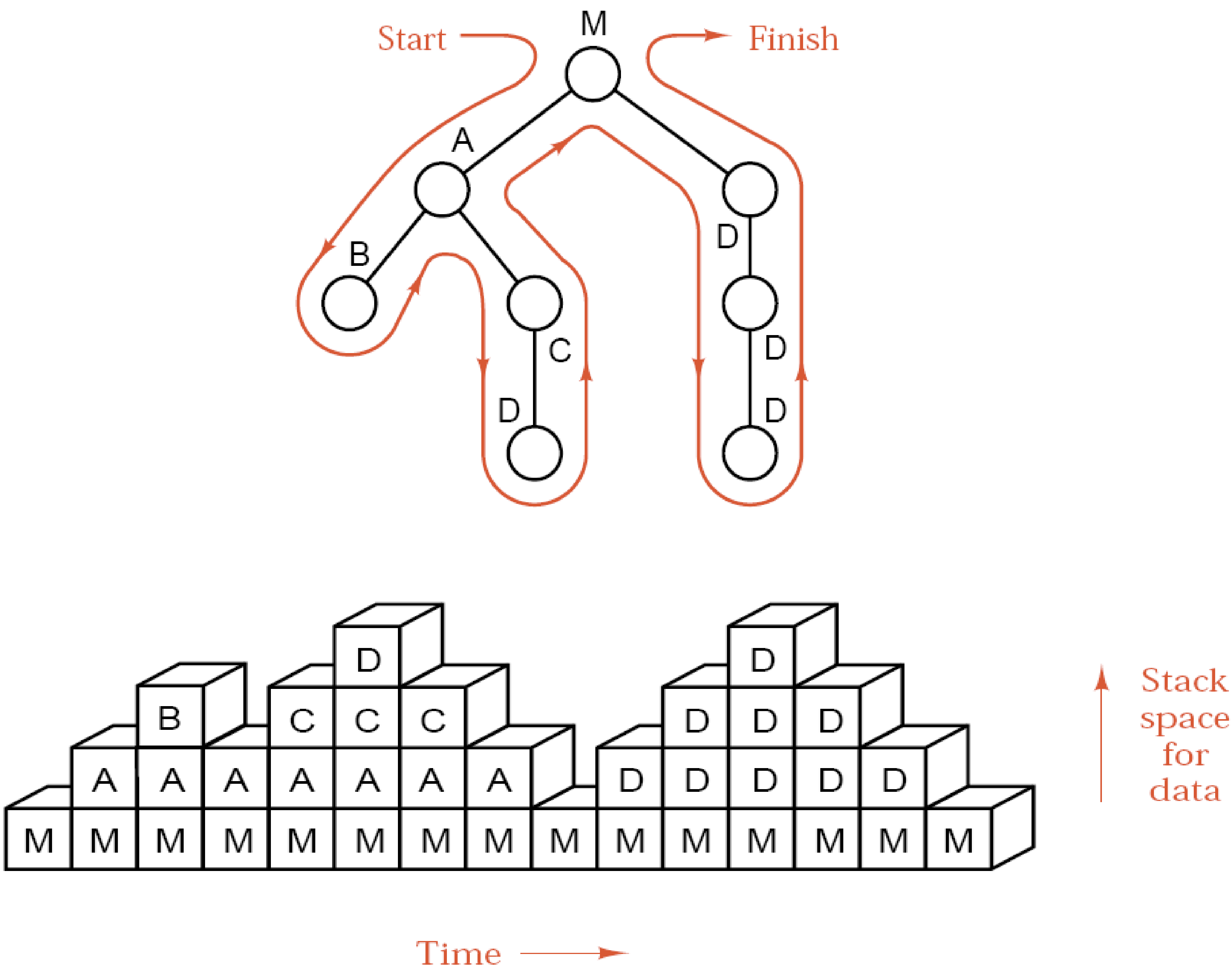
Subprogram implementation



Subprogram implementation



Tree and Stack frames of function calls



Tree and Stack frames of function calls

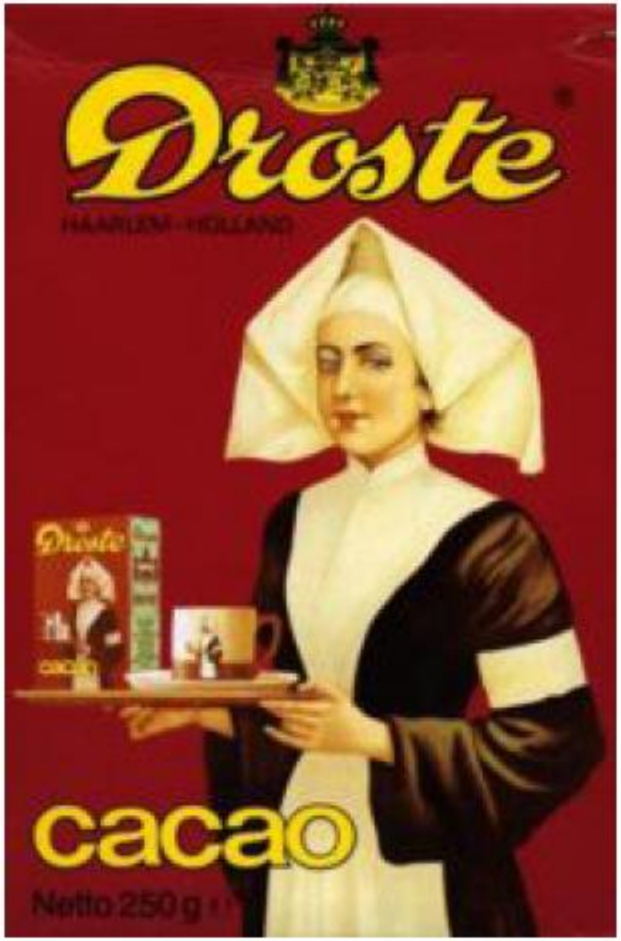
❑ Stack frames:

- ➢ Each vertical column shows the contents of the stack at a given time

- ➢ There is no difference between two cases:
  - o when the temporary storage areas pushed on the stack come from different functions, and
  - o when the temporary storage areas pushed on the stack come from repeated occurrences of the same function.

Recursion

An object
contains itself



Recursion

- A definition contains itself:
  - Sibling(X, Y): X and Y have the same parents
  - Cousin(X, Y): X's and Y's parents are siblings OR cousins

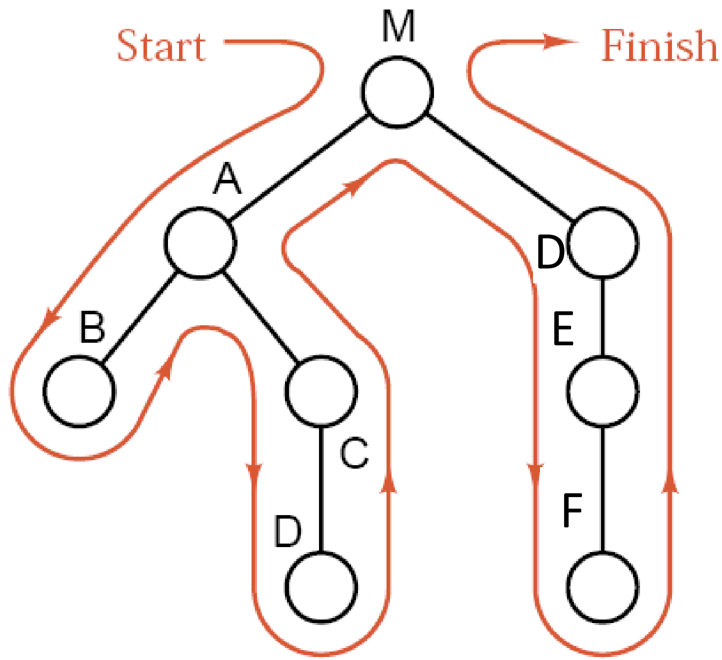
Recursion

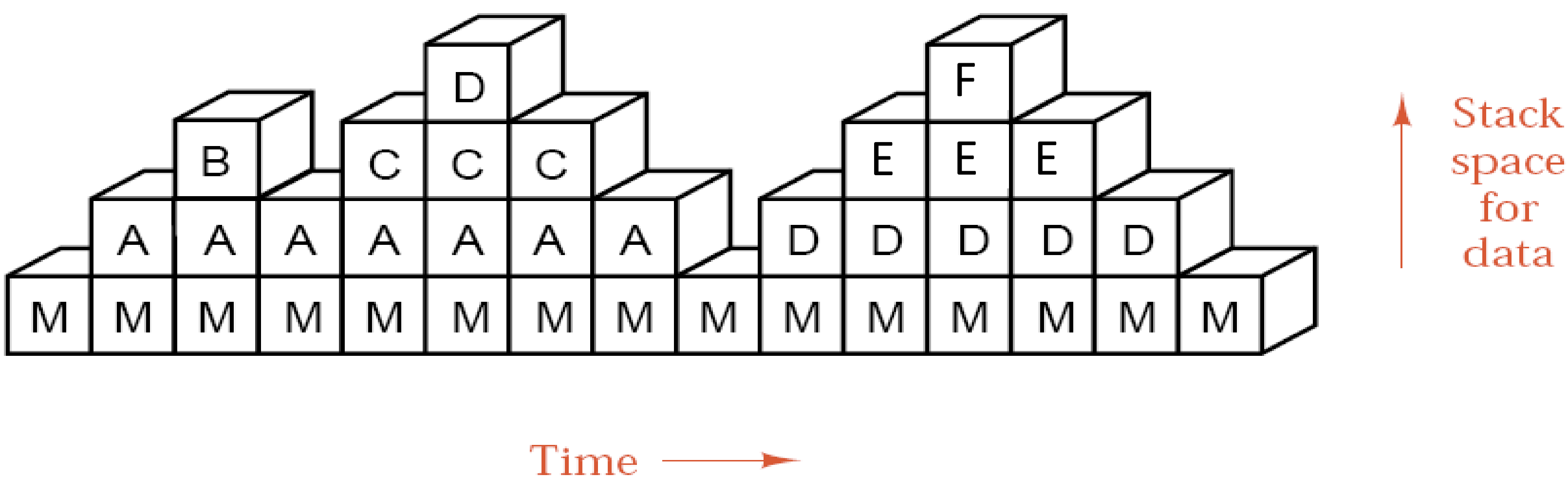
Recursion is the name for the case when:

- A function invokes itself, or
- A function invokes a sequence of other functions, one of which eventually invokes the first function again.

- In regard to stack frames for function calls, recursion is no different from any other function call.
  - Stack frames illustrate the storage requirements for recursion.
  - Separate copies of the variables declared in the function are created for each recursive call.

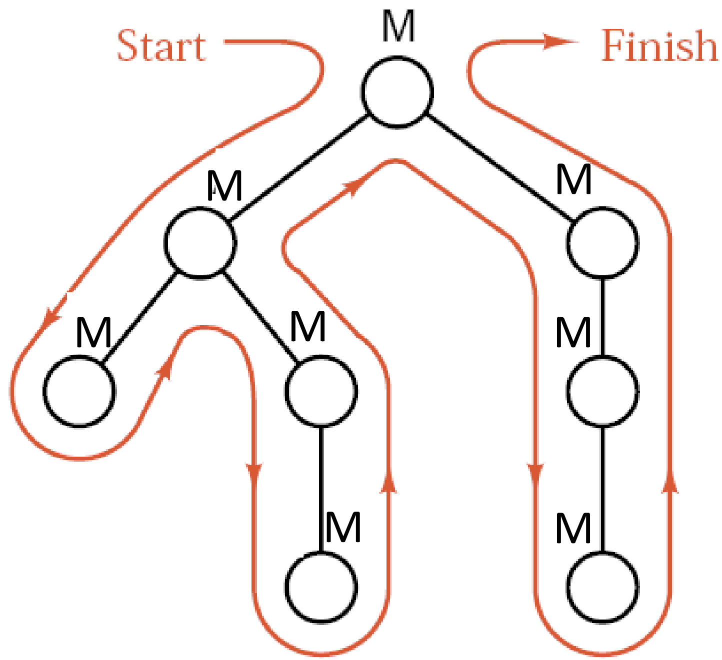
Tree and Stack frames of function calls

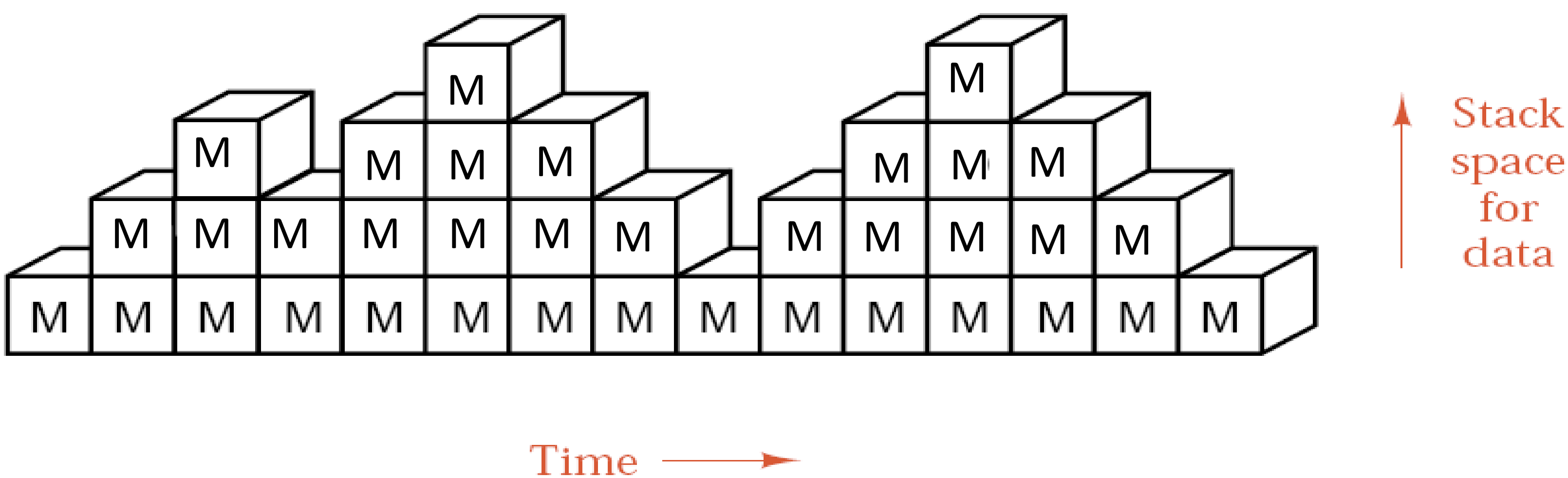




Tree and Stack frames of function calls

Recursive calls:





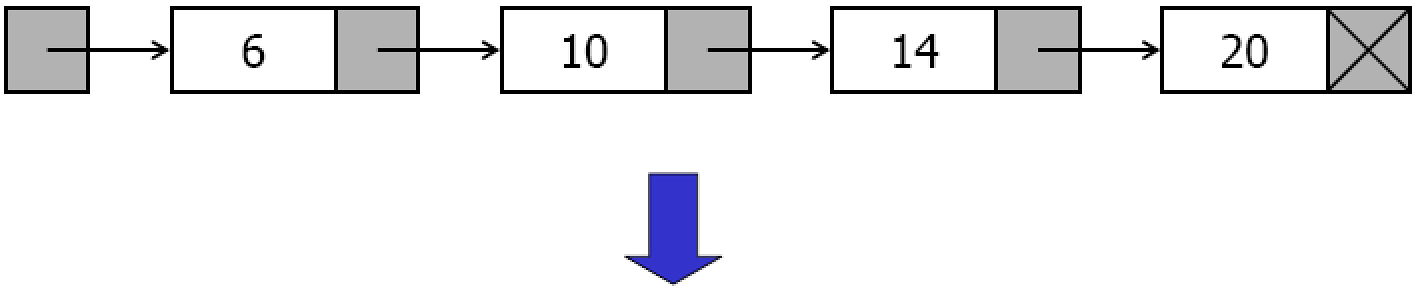
Recursion

- In the usual implementation of recursion, there are kept on a stack.
  - The amount of space needed for this stack depends on the depth of recursion, not on the number of times the function is invoked.
  - The number of times the function is invoked (the number of nodes in recursive tree) determines the amount of running time of the program.

Recursion

- Does human thinking involve recursion?

Print List

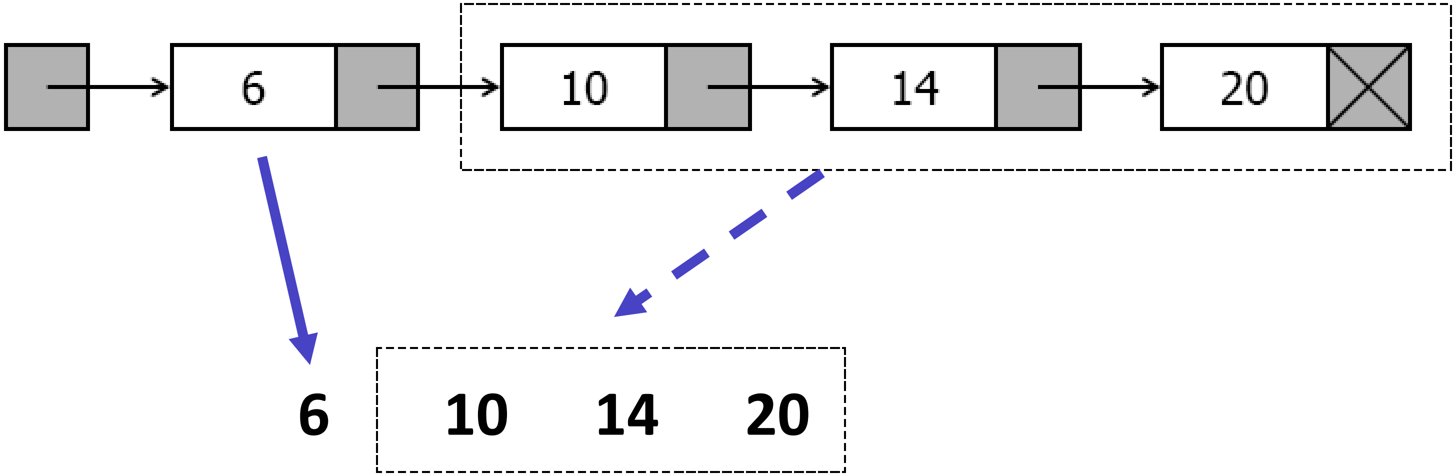


6 10 14 20

Print List

- A list is
  - empty, or
  - consists of an element and a sublist, where sublist is a list.

Print List



Print List

Algorithm **Print**(val head <pointer>)

Prints Singly Linked List.

Pre head points to the first element of the list needs to be printed.

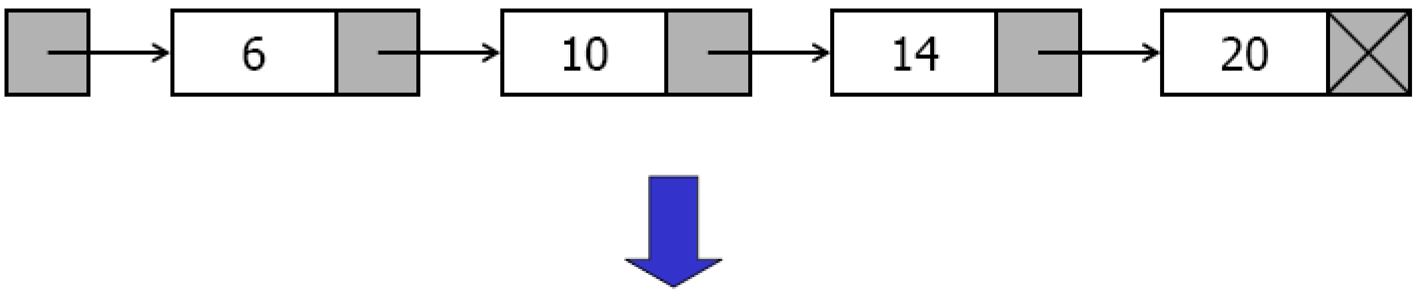
Post Elements in the list have been printed.

Uses recursive function Print.

1. **if** (head = NULL) *// stopping case*
   1. return
2. write (head->data)
3. Print(head->link) *// recursive case*

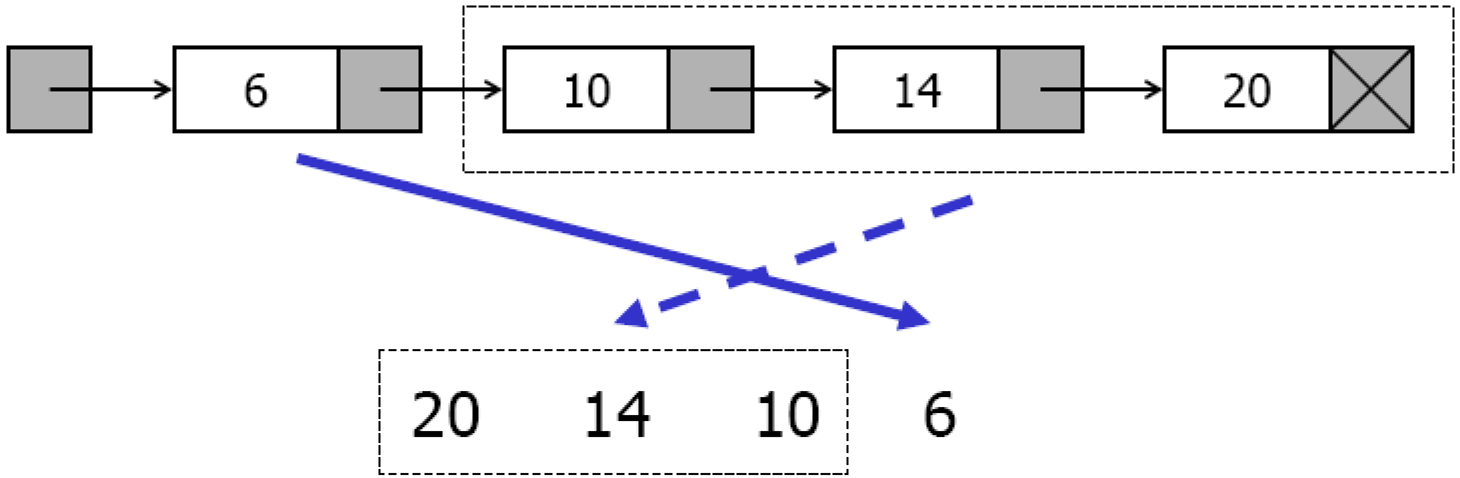
End Print

Print List in Reverse



20 14 10 6

Print List in Reverse



Print List in Reverse

Algorithm **PrintReverse**(val head <pointer>)

Prints Singly Linked List in reverse.

Pre head points to the first element of the list needs to be printed.

Post Elements in the list have been printed in reverse.

Uses recursive function PrintReverse.

1. **if** (head = NULL) *// stopping case*
    1. return
2. PrintReverse(head->link) *// recursive case*
3. write (head->data)

End PrintReverse

Factorial: A recursive Definition

Iterative algorithm

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 & \text{if } n > 0 \end{cases}$$

Recursive algorithm

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

Iterative Solution

Algorithm **IterativeFactorial** (val n <integer>)

Calaculates the factorial of a number using a loop.

Pre n is the number to be raised factorial, n >= 0.

Post n! is returned.

1. i = 1
2. factN = 1
3. **loop** (i <= n)
   1. factN = factN * i
   2. i = i + 1
4. return factN

End IterativeFactorial

Recursive Solution

Algorithm **RecursiveFactorial** (val n <integer>)

Caculates the factorial of a number using recursion.

Pre n is the number to be raised factorial, n >= 0

Post n! is returned.

Uses recursive function RecursiveFactorial

1. **if** (n = 0)
    1. factN = 1 *// stopping case*
2. **else**
    1. factN = n * RecursiveFactorial(n-1) *// recursive case*
3. return factN

End RecursiveFactorial

Recursive Solution

$$\text{Factorial}(3) = 3 \times \text{Factorial}(2)$$

$$\text{Factorial}(2) = 2 \times \text{Factorial}(1)$$

$$\text{Factorial}(1) = 1 \times \text{Factorial}(0)$$

$$\text{Factorial}(0) = 1$$

$$\text{Factorial}(1) = 1 \times 1 = 1$$

$$\text{Factorial}(2) = 2 \times 1 = 2$$

$$\text{Factorial}(3) = 3 \times 2 = 6$$

- The recursive definition and recursive solution can be both concise and elegant.

- The computational details can require keeping track of many partial computations before the process is complete.

Recursive Solution

Algorithm **RecursiveFactorial** (val n <integer>)

Calaculates the factorial of a number using recursion.

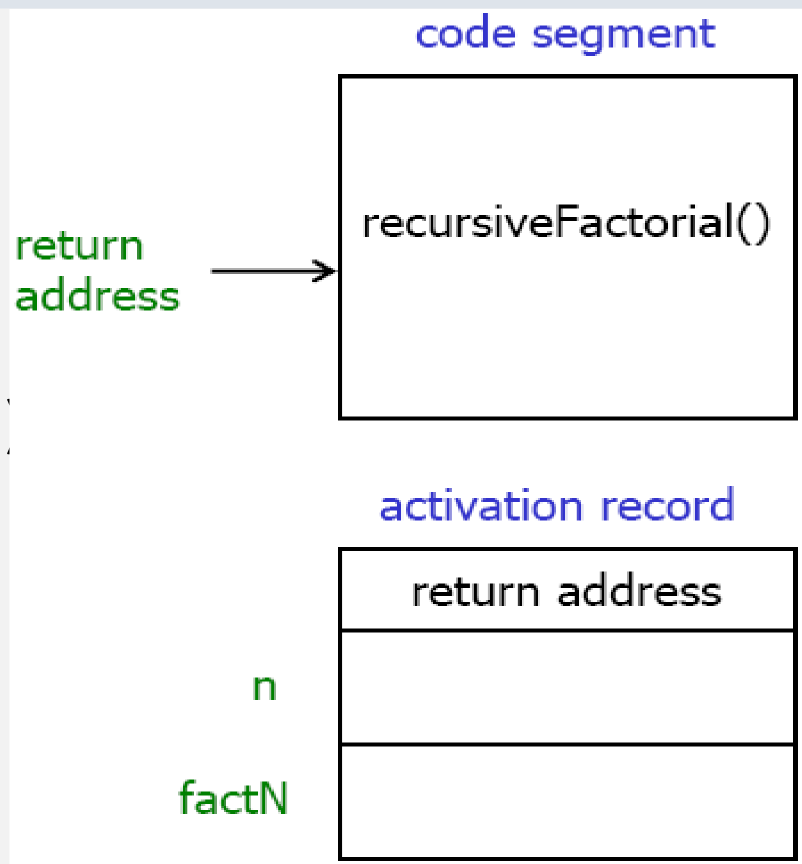
Pre n is the number to be raise factorial, n >= 0.

Post n! is returned

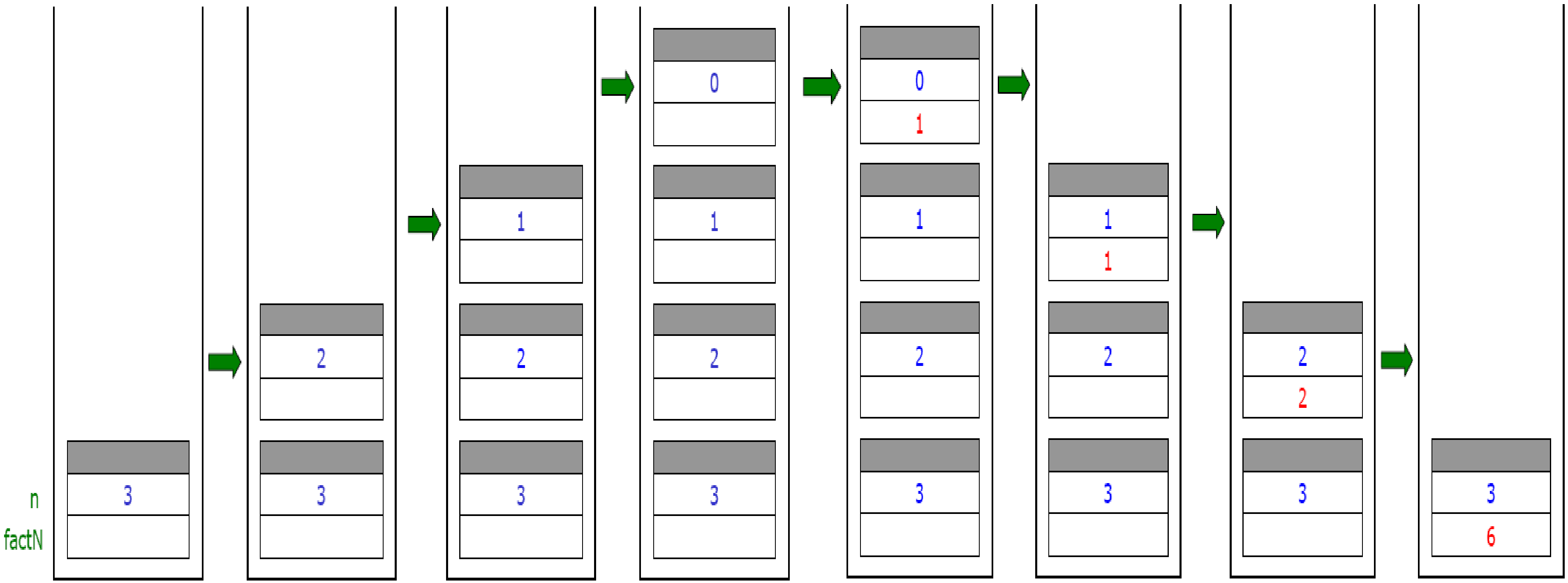
Uses recursive function RecursiveFactorial.

1. if (n = 0)
    1. factN = 1
2. else
    1. factN = n * RecursiveFactorial(n-1)
3. return factN

End RecursiveFactorial



Recursive Solution



Stack

Thinking of Recursion

- Remembering partial computations: computers can easily keep track of such partial computations with a stack, but human mind can not.

- It is exceedingly difficult for a person to remember a long chain of partial results and then go back through it to complete the work.

- Ex.:

> As I was going to St. Ives,
> I met a man with seven wives.
> Each wife had seven sacks,
> Each sack had seven cats,
> Each cat had seven kits:
> Kits, cats, sacks and wives,
> How many were there going to St. Ives?

- When we use recursion, we need to think in somewhat difference terms than with other programming methods.

- Programmers must look at the big picture and leave the detailed computations to the computer.

Recursion

❑ The essence of the way recursion works:

- To obtain the answer to a large problem, a general method is used that reduces the large problem to one or more problems of a similar nature but a smaller size.
- The same general method is then used for these subproblems.
- Recursion continues until the size of the subproblems is reduced to some smallest, base case.
- Base case: the solution is given directly without using further recursion.

Recursion

Every recursive process consists of two parts:

1. Some smallest base cases that are processed without recursion.
2. A general method that reduces a particular case to one or more of the smaller cases, thereby making progress toward eventually reducing the problem all the way to the base case.

Designing Recursive Algorithms

Recursive algorithm = recursive case + stopping case

$n \times \text{factorial}(n-1)$ factorial(0)

- Every recursive call must solve a part of the problem or reduce the size of the problem.

Designing Recursive Algorithms

- Determine the recursive case
- Determine the stopping case
- Combine the recursive and stopping cases

Designing Recursive Algorithms

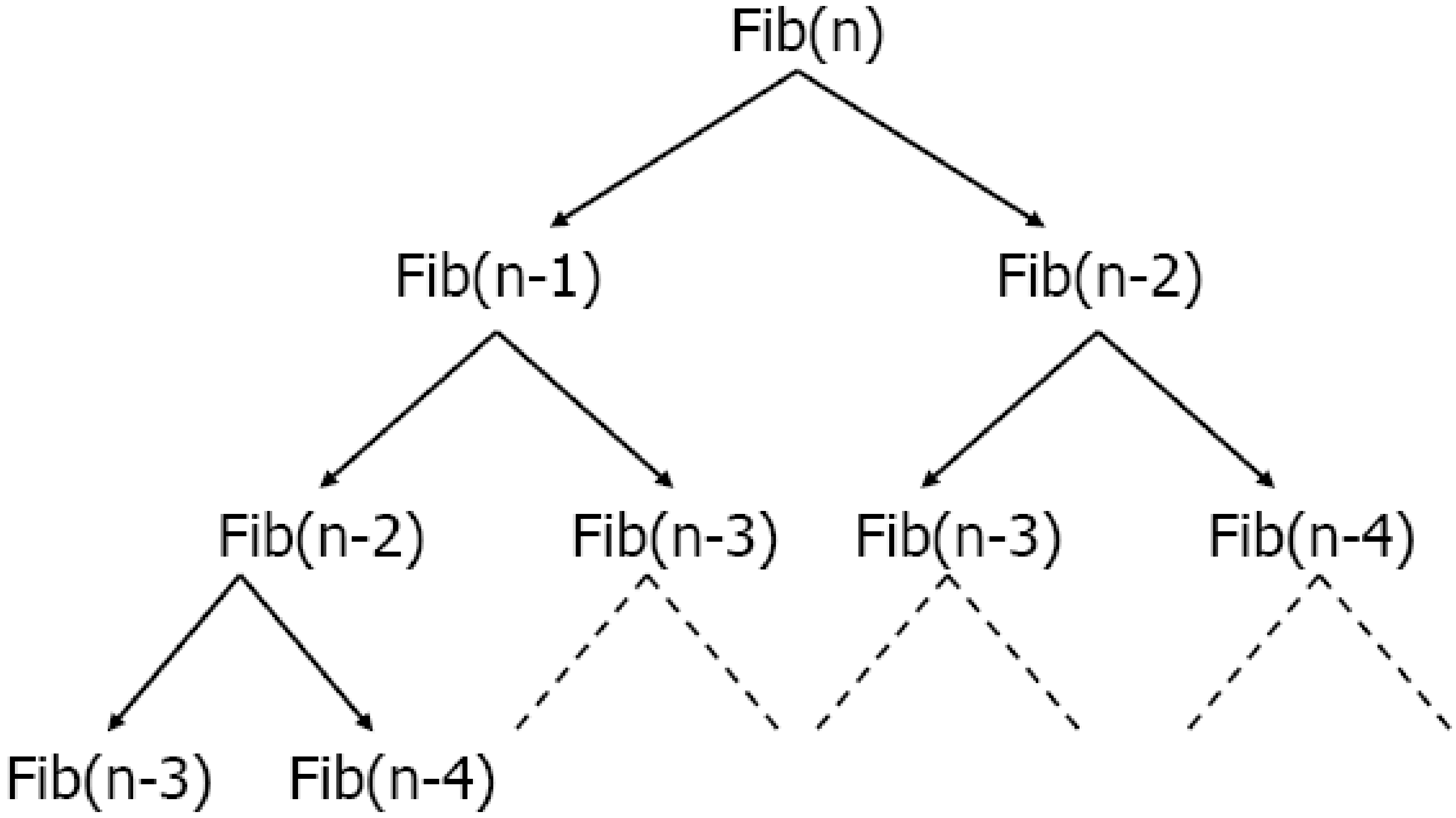
- Is the algorithm or data structures naturally suited to recursion?
- Is the recursive solution shorter and more understandable?
- Does the recursive solution run in acceptable time and space limits?

Fibonacci Numbers

0 1 1 2 3 5 8 13 21 34

- Recursive case: Fib(n) = Fib(n – 1) + Fib(n – 2)
- Stopping case: Fib(0) = 0 Fib(1) = 1

Fibonacci Numbers



Fibonacci Numbers

Algorithm **Fibonacci** (val n <integer>)

Calculates the n^{th} Fibonacci number.

Pre n is the ordinal of the Fibonacci number.

Post returns the n^{th} Fibonacci number

Uses Recusive function Fibonacci

1. if (n = 0) OR (n = 1) *// stopping case*
    1. return n
2. return (Fibonacci(n -1)+Fibonacci(n -2)) *// recursive case*

End Fibonacci

Fibonacci Numbers

No	Calls	Time	No	Calls	Time
1	1	< 1 sec.	11	287	< 1 sec.
2	3	< 1 sec.	12	465	< 1 sec.
3	5	< 1 sec.	13	753	< 1 sec.
4	9	< 1 sec.	14	1,219	< 1 sec.
5	15	< 1 sec.	15	1,973	< 1 sec.
6	25	< 1 sec.	20	21,891	< 1 sec.
7	41	< 1 sec.	25	242,785	1 sec.
8	67	< 1 sec.	30	2,692,573	7 sec.
9	109	< 1 sec.	35	29,860,703	1 min.
10	177	< 1 sec.	40	331,160,281	< 13 min.

Fibonacci Numbers

- Is the algorithm or data structures naturally suited to recursion?
- Is the recursive solution shorter and more understandable?
- Does the recursive solution run in acceptable time and space limits?

Fibonacci Numbers

- The recursive program needlessly repeats the same calculations over and over.

- The amount of time used by the recursive function to calculate F_n grows exponentially with n.

- Simple iterative program: starts at 0 and keep only three variables, the current Fibonacci number and its two predecessors.

Fibonacci Numbers (Iterative version)

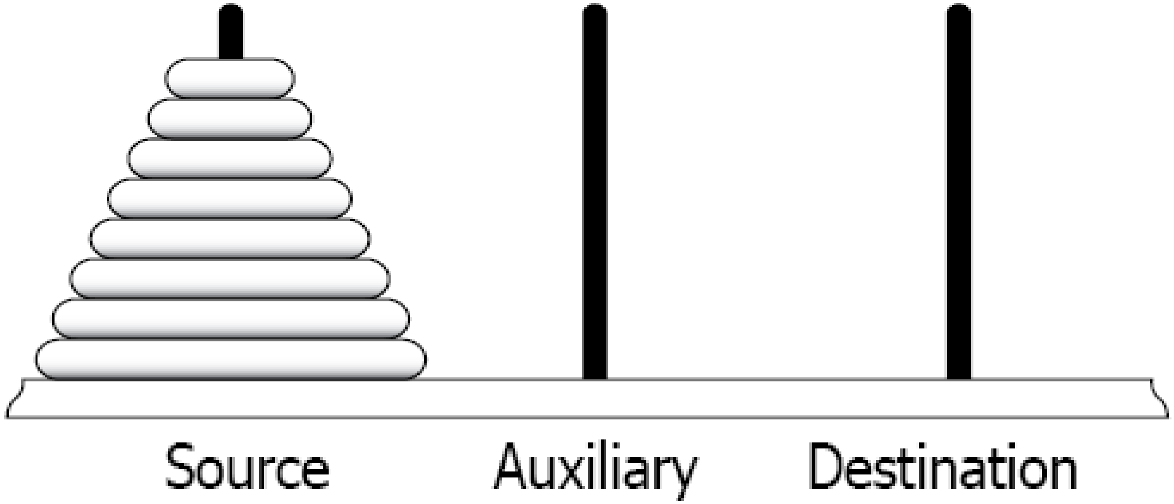
```
int fibonacci(int n)

/* 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;
   }
}
```

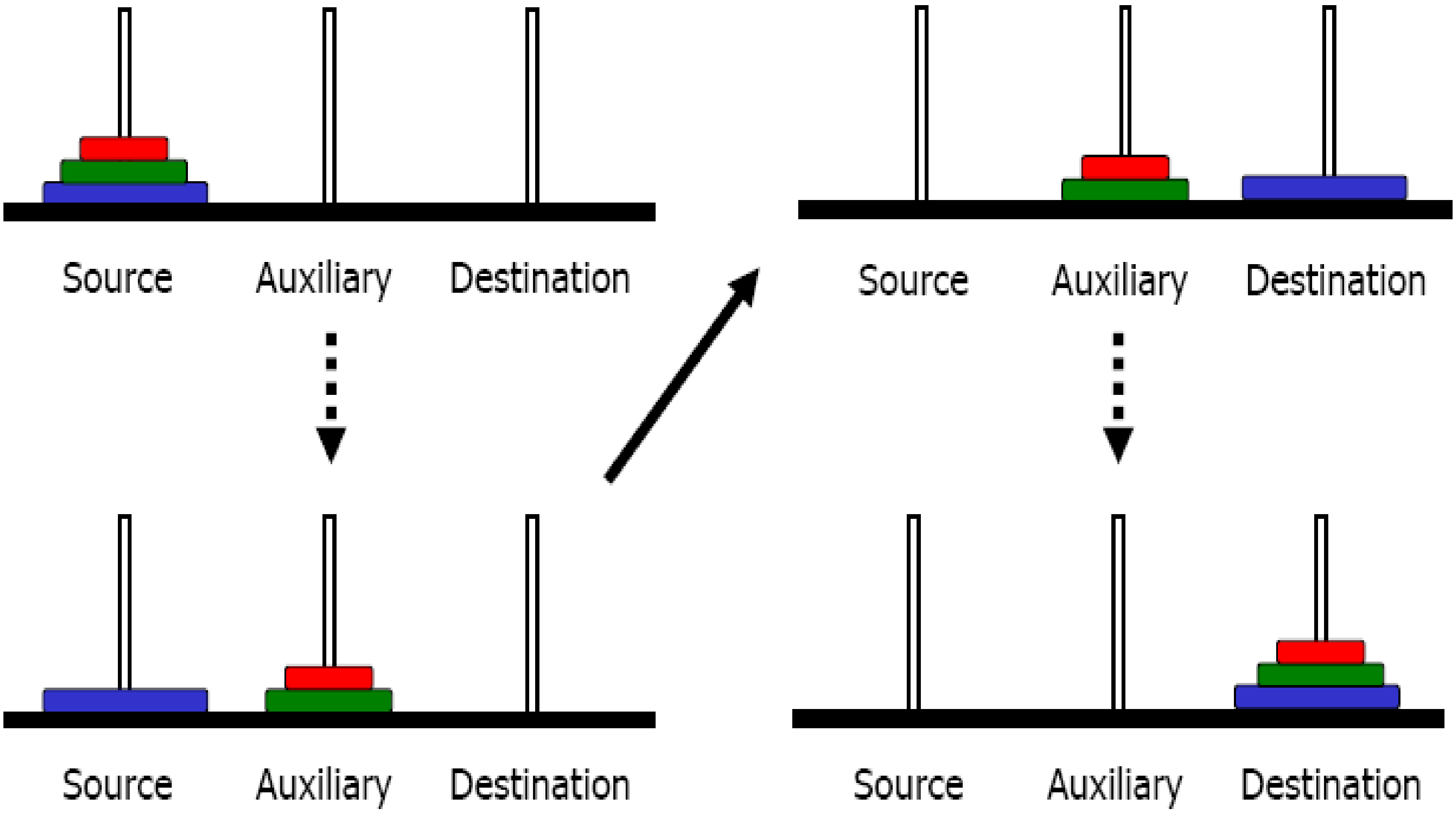
The Towers of Hanoi

Move disks from Source to Destination using Auxiliary:

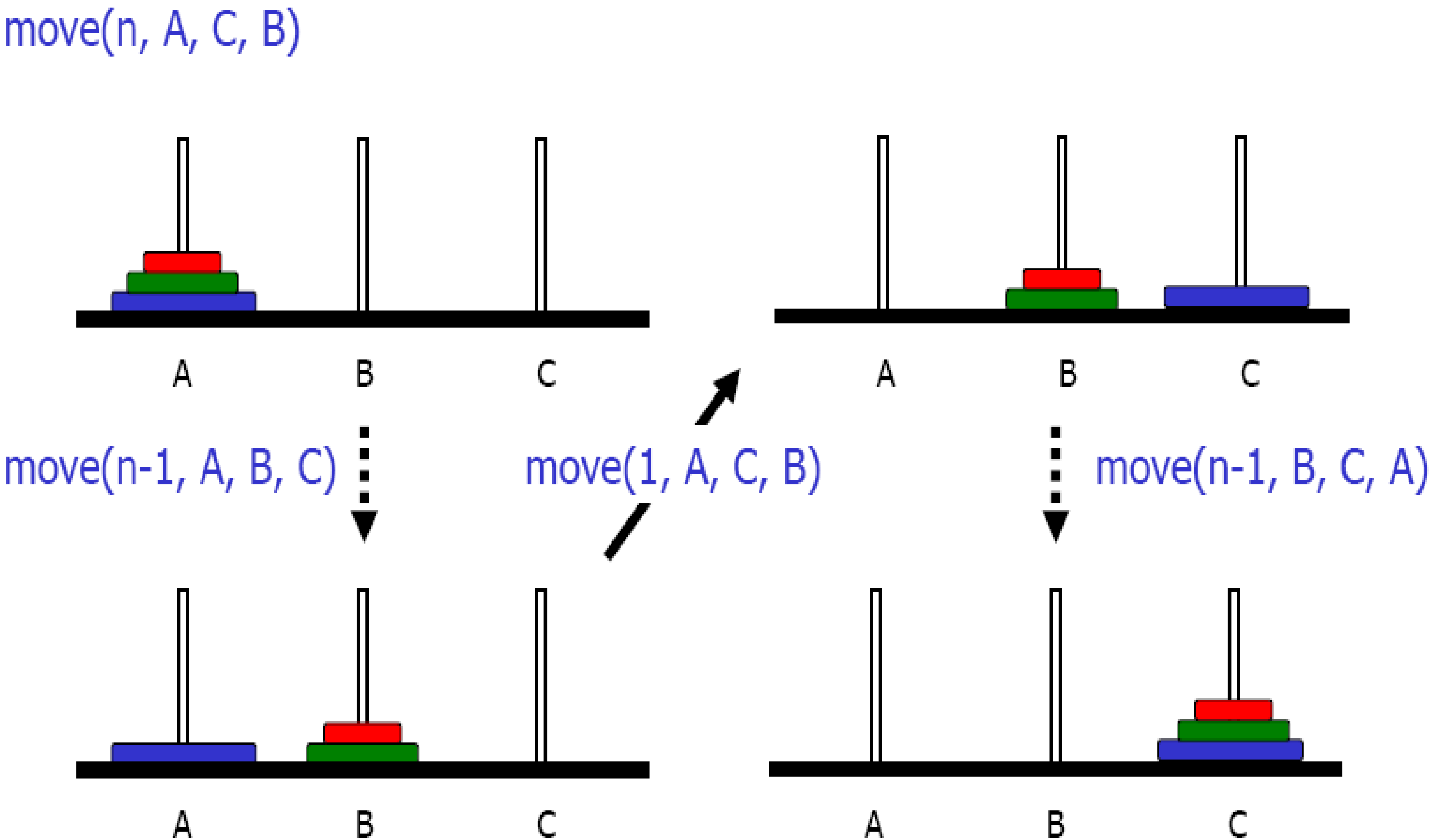
1. Only one disk could be moved at a time.
2. A larger disk must never be stacked above a smaller one.
3. Only one auxiliary needle could be used for the intermediate storage of disks.



The Towers of Hanoi



The Towers of Hanoi



The Towers of Hanoi

Algorithm **Move** (val count <integer>,val source <integer>,
val destination <integer>, val auxiliary <integer>)

Moves count disks from source to destination using auxiliary.

Pre There are at least count disks on the tower source.
The top disk (if any) on each of towers auxiliary and destination is larger than any of the top count disks on tower source.

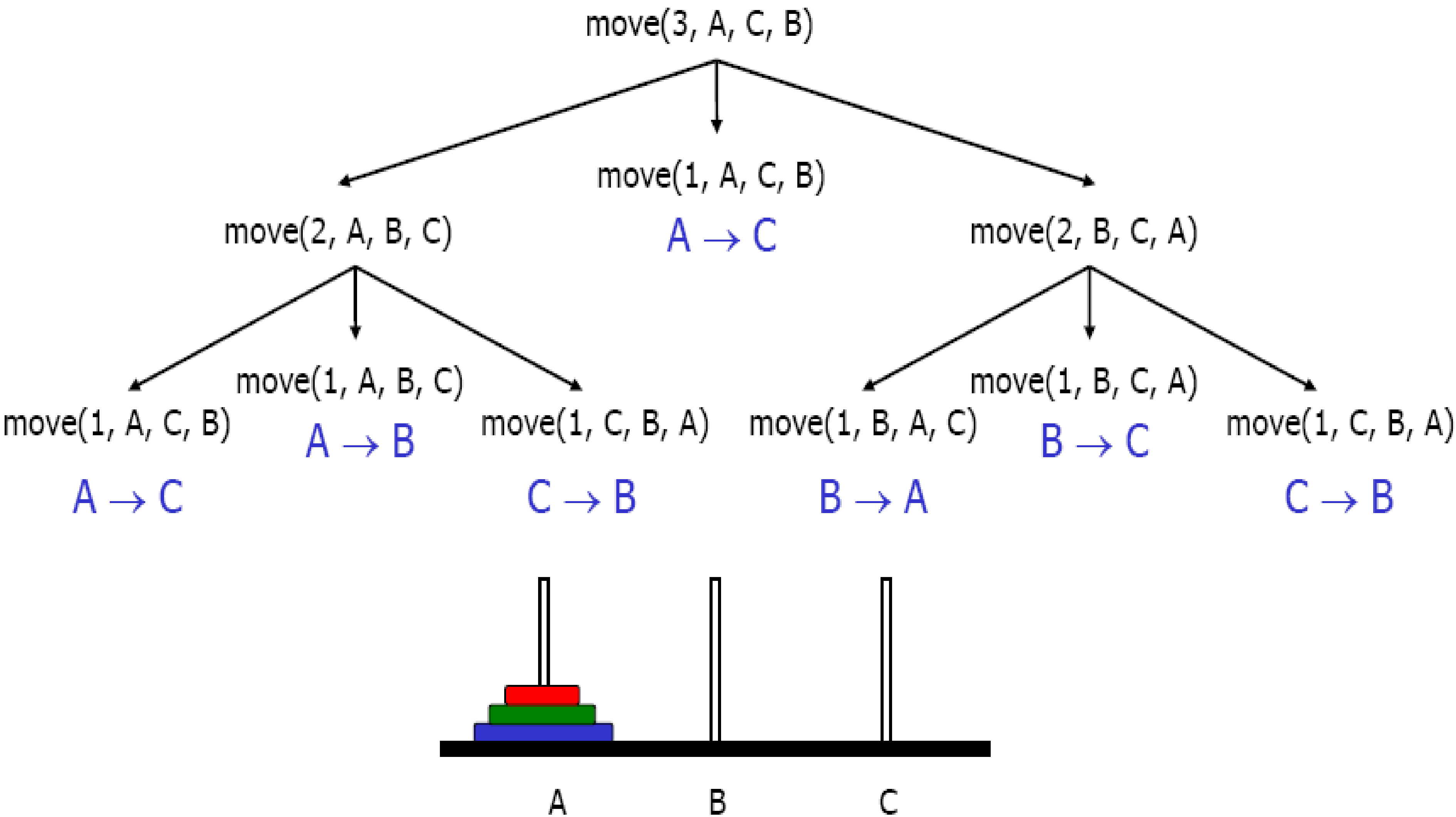
Post The top count disks on source have been moved to destination;
auxiliary (used for temporary storage) has been returned to its starting position.

Uses recursive function Move.

1. **if** (count > 0)
   1. Move (count -1, source, auxiliary, finish)
   2. move a disk from source to finish
   3. Move (count -1, auxiliary, finish, source)

End Move

The Towers of Hanoi



The Towers of Hanoi

- Does the program for the Towers of Hanoi produce the best possible solution? (possibly include redundant and useless sequences of instruction sush as:
  - Move disk 1 from tower 1 to tower 2.
  - Move disk 1 from tower 2 to tower 3.
  - Move disk 1 from tower 3 to tower 1.)
- How about the depth of recursion tree?
- How many instructions are needed to move 64 disks? (One instruction is printed for each vertex in the tree, except for the leaves with count = 0)

The Towers of Hanoi

- The depth of recursion is 64, not include the call with count = 0 which do nothing.

- Total number of moves:

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

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

- If one move takes 1s, 2^{64} moves take about 5×10^{11} years!

- Recursive program for the Towers of Hanoi would fail for lack of time, but not for lack of space.

The Towers of Hanoi

Complexity:

$$T(n) = 2 \times T(n - 1) + 1$$

$$\Rightarrow T(n) = O(2^n)$$

Recursion

When recursion should or should not be used?

Recursion or Not?

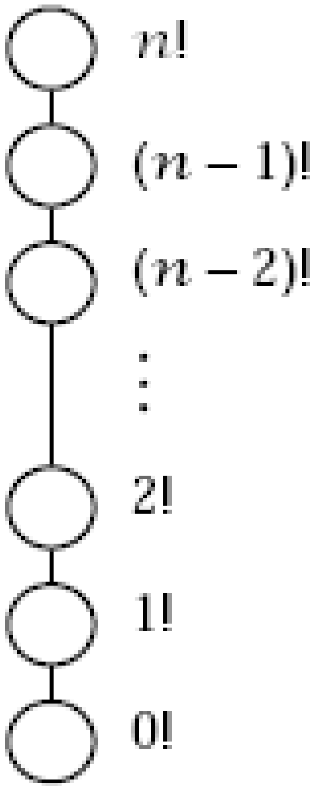
❑ Chain:

➢ Recursive function makes only one recursive call to itself.

➢ Recursion tree does reduce to a chain: each vertex has only one child.

➢ By reading the recursion tree from bottom to top instead of top to bottom, we obtain the iterative program.

➢ We save both space and time.



Recursion tree for calculating factorials

Recursion or Not?

- Notice of the chain:

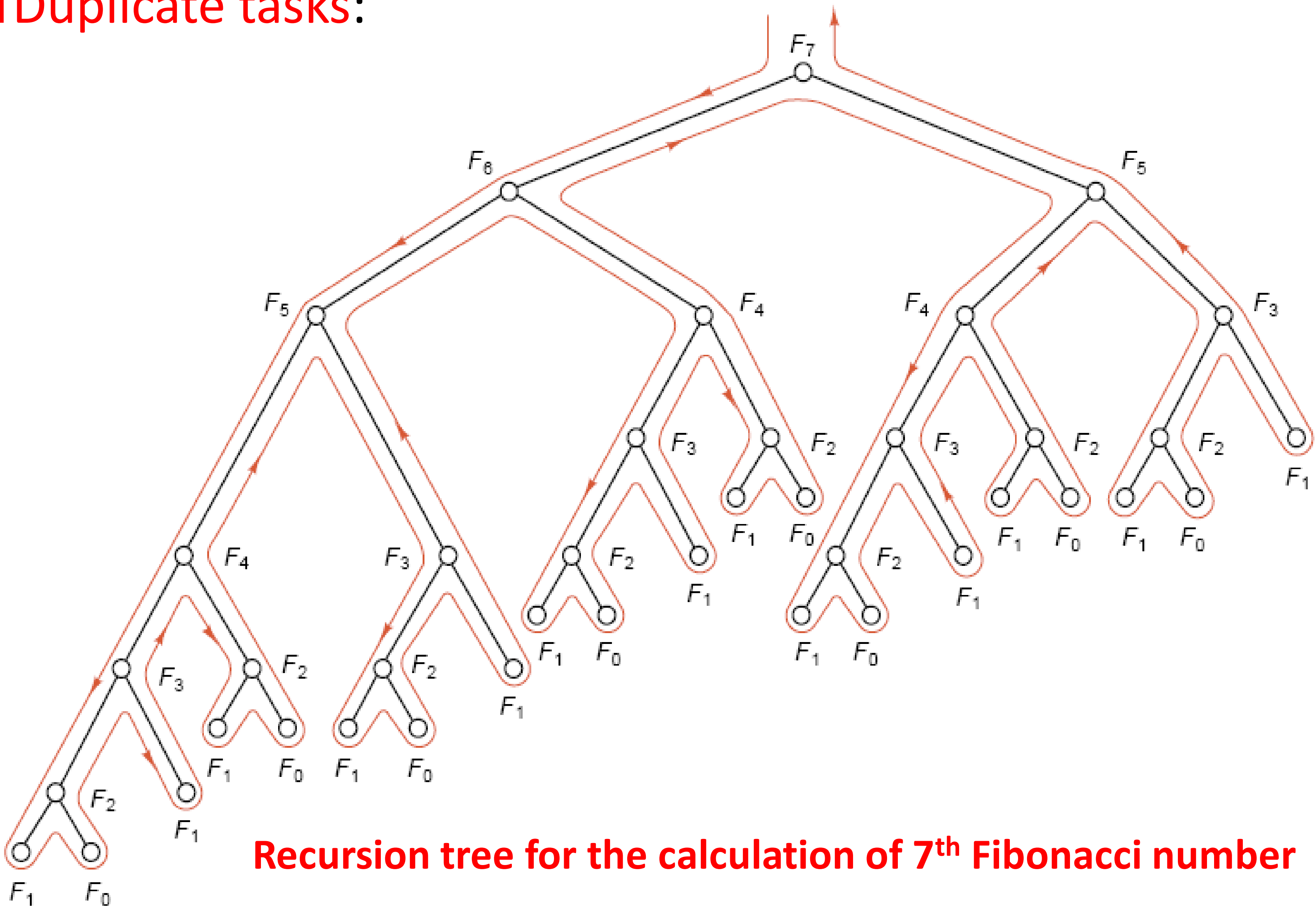
  - A chain: recursive function makes only one recursive call to itself.

  - Recursive call appears at only one place in the function, but might be inside a loop: more than one recursive calls!

  - Recursive call appears at more than one place in the function but only one call can actually occur (conditional statement)

Recursion or Not?

- Duplicate tasks:



Recursion tree for the calculation of 7th Fibonacci number

Recursion or Not?

❑ Duplicate tasks:

➢ Recursion tree for calculating Fibonacci numbers contains many vertices signifying duplicate tasks.

➢ The results stored in the stack, used by the recursive program, are discarded rather than kept in some other data structure for future use.

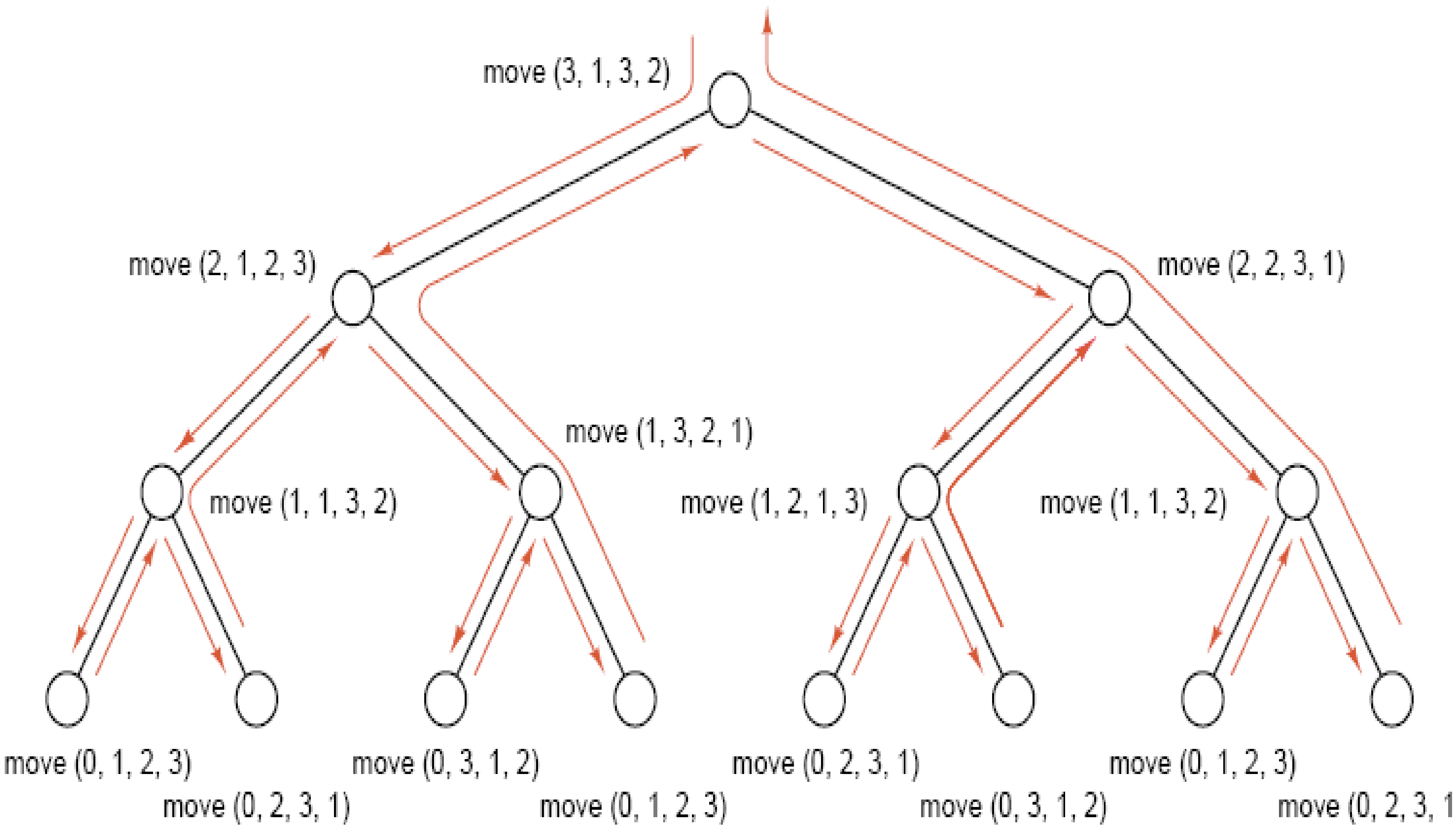
➢ It is preferable to substitute another data structure that allows references to locations other than the top of the stack.

Recursion or Not?

❑Comparison of Fibonacci and Hanoi:

- Both have a very similar divide-and-conquer form.
- Each consists of two recursive calls.
- Hanoi recursive program is very efficient.
- Fibonacci recursive program is very inefficient.
- Why?
- The answer comes from the size of the output!
- Fibonacci calculates only one number, while
- For Hanoi, the size of the output is the number of instructions to be printed, which increases exponentially with the number of disks.

Recursion or Not?



Recursion tree for three disks

Recursion or Not?

- If the recursion tree has a simple form: iterative version may be better.
- If the recursion tree involves duplicate tasks: data structures other than stacks will be appropriate, the need for recursion may disappear.
- If the recursion tree appears quite bushy, with little duplication of tasks: recursion is likely the natural method.

Recursion or Not?

- Top-down design:
  - Recursion is something of a top-down approach to problem solving.
  - Iteration is more of a bottom-up approach.
- Stacks or recursion?
  - Any recursion can be replaced by iteration stack.
  - When recursion should be removes?
  - When a program involves stacks, the introduction of recursion might produce a more natural and understandable program.

Recursion Removal

- Recursion can be removed using stacks and iteration.

Recursion Removal

Algorithm P (val n <integer>)

1. **if** (n = 0)
   1. print("Stop")
2. else
   1. Q(n)
   2. P(n - 1)
   3. R(n)

End P

Execution:

Q(n)
Q(n-1)
...
Q(1)
Stop
R(1)
R(2)
...
R(n)

Recursion Removal

```
Algorithm P (val n <integer>)
1.  if (n = 0)
    1.  print("Stop")
2.  else
    1.  Q(n )
    2.  P(n - 1)
    3.  R(n)
End P
```

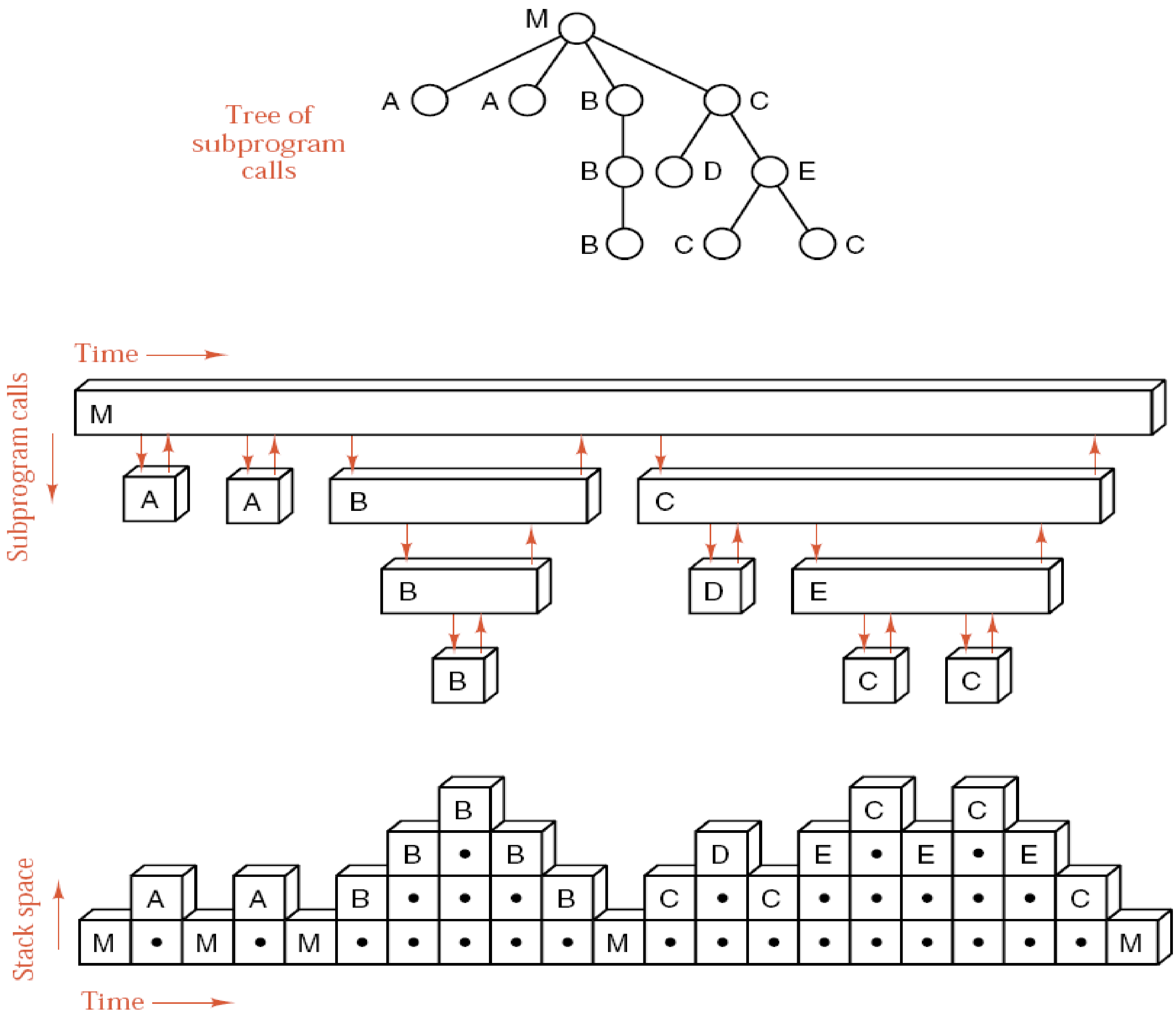
```
Algorithm P (val n <integer>)
1.  stackObj <Stack>
2.  loop (n >0)
    1.  Q(n )
    2.  stackObj.Push(n)
    3.  n = n – 1
3.  print("Stop")
4.  loop (NOT stackObj.isEmpty())
    1.  stackObjTop(n )
    2.  stackObj.Pop()
    3.  R(n )
End P
```

Tail Recursion

DEFINITION : Tail recursion occurs when the last-executed statement of a function is a recursive call to itself.

- Tail recursion can be eliminated by reassigning the calling parameters to the values specified in the recursive call, and then repeating the whole function.

Subprogram calls

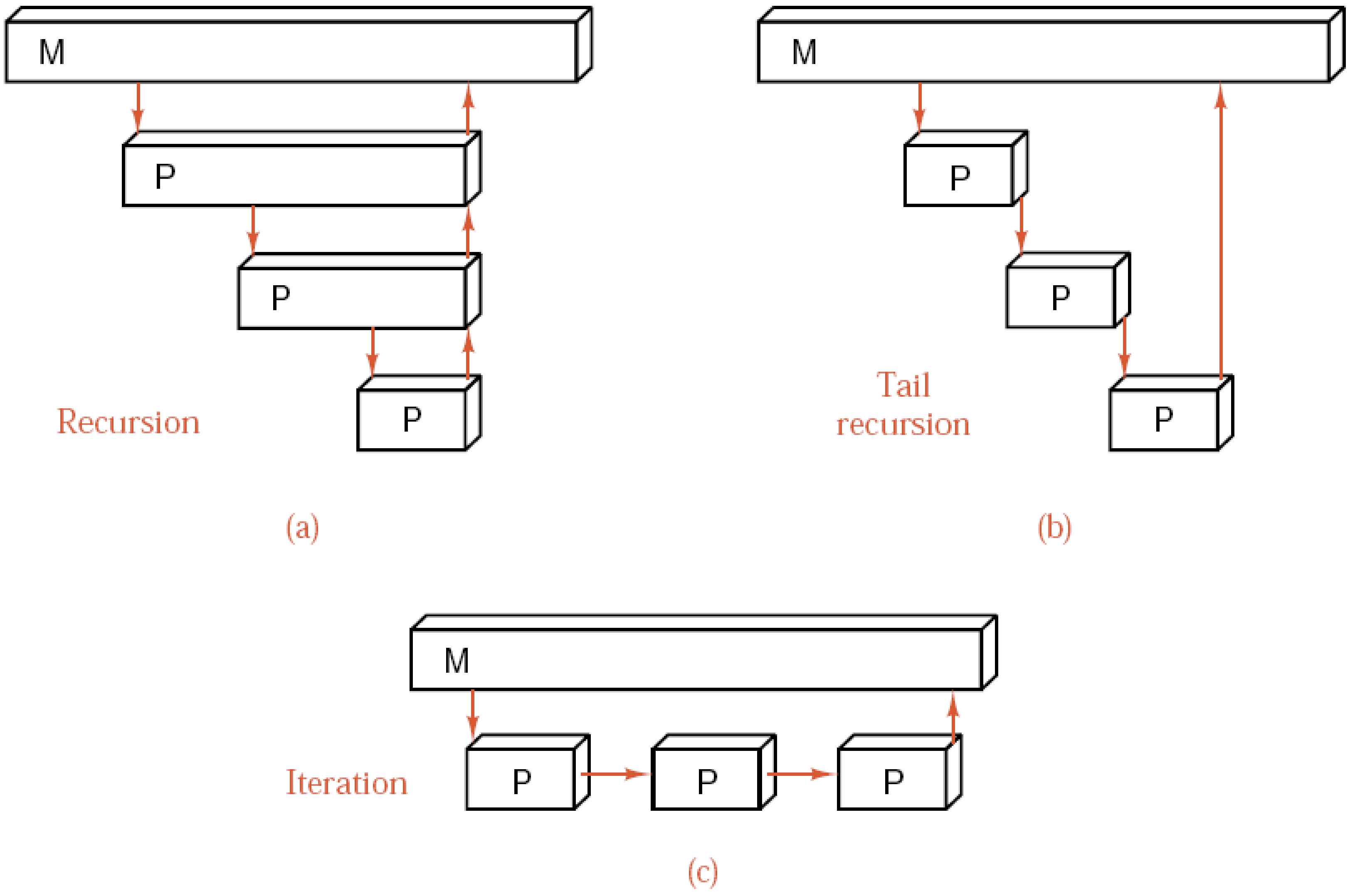


Tail Recursion

- When the recursive call is initiated, the local variables of the calling function are pushed onto the stack.
- When the recursive call terminates, these local variables are popped from the stack and used for the calling function.
- But there is no any task the calling function must do.

⇒ If space considerations are important, tail recursion should be removed.

Tail Recursion



Tail Recursion

```
Algorithm  P (val n <integer>)
1   if (n = 0)
    1   print("Stop")
2   else
    1   Q(n)
    2   P(n - 1)
End  P
```

Execution:

Q(n)

Q(n - 1)

...

Q(1)

Stop

Tail Recursion

```
Algorithm  P (val n <integer>)
1   if (n = 0)
    1   print("Stop")
2   else
    1   Q(n)
    2   P(n - 1)
End  P
```

```
Algorithm  P (val n <integer>)
1   loop (n > 0)
    1   Q(n)
    2   n = n - 1
2   print("Stop")
End  P
```

Backtracking

- A process to go back to previous steps to try unexplored alternatives.

Eight Queens problem

PROBLEM: Place eight queens on the chess board in such a way that no queen can capture another.

Algorithm **EightQueens**

Finds all solutions of Eight Queens problem.

Pre ChessBoard contains no Queen.

Post All solutions of Eight Queens problem are printed.

Uses Recursive function RecursiveQueens.

1. row = 0
2. RecursiveQueens(ChessBoard, row)

Eight Queens problem

Algorithm **RecursiveQueens**(val ChessBoard <BoardType>,

val r <integer>)

Pre ChessBoard represents a partially completed arrangement of nonattacking queens on the rows above row r of the chessboard

Post All solutions that extend the given ChessBoard are printed. The ChessBoard is restored to its initial state.

Uses recursive function RecursiveQueens.

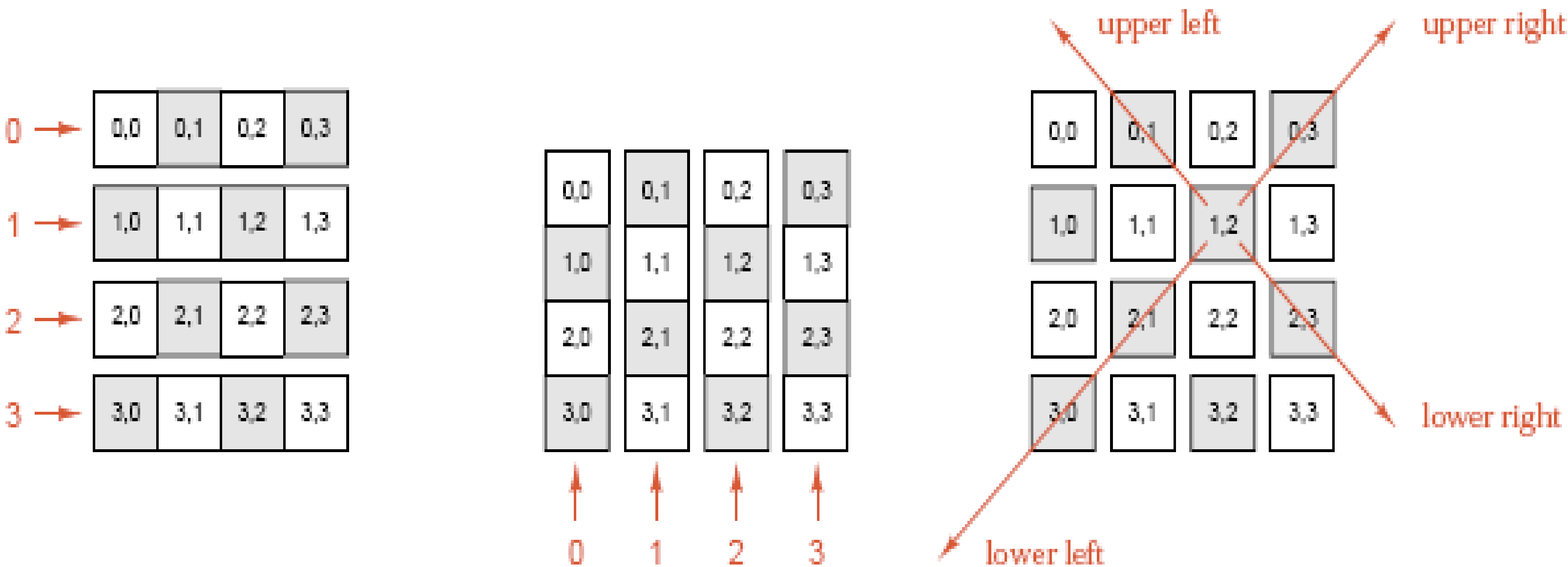
Eight Queens problem

Algorithm **RecursiveQueens**(val chessBoard <BoardType>,
val r <integer>)

1. **if** (ChessBoard already contains eight queens)
   1. print one solution.
2. **else**
   1. **loop** (more column c that cell[r, c] is unguarded)
      1. add a queen on cell[r, c]
      2. RecursiveQueens(chessBoard) *// recursivelly continue to*
         // add queens to the next rows.
      3. remove a queen from cell[r, c]

End RecursiveQueens

Eight Queens problem



(a) Rows (b) Columns (c) Diagonal directions

- Data structure for version 1 of class Board

Eight Queens problem

```
class Board_ver1 {

  private:
    int count;                // current number of queens = first unoccupied row
    bool queen_square[max_board][max_board];

};
```

Eight Queens problem

Algorithm **RecursiveQueens**(val chessBoard <BoardType>,

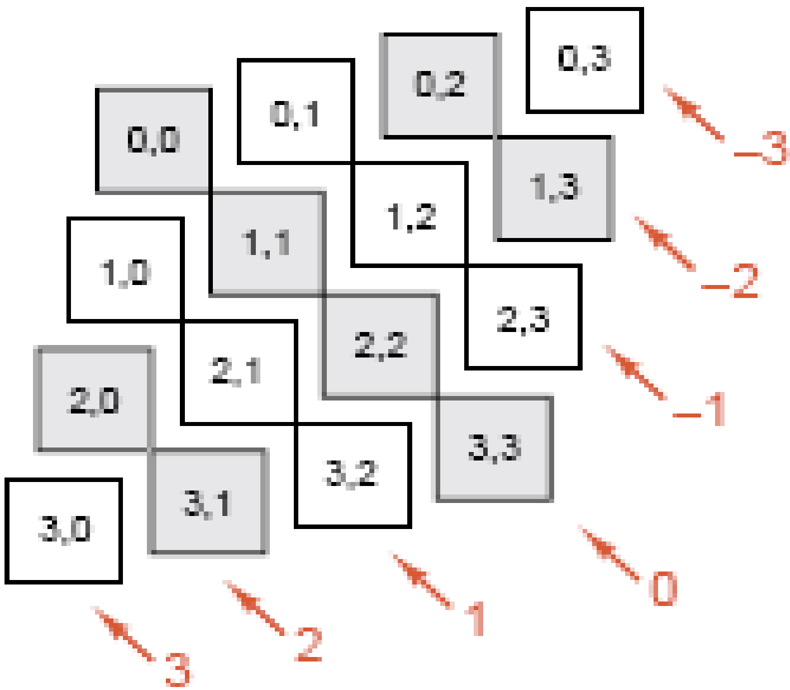
val r <integer>)

1. **if** (ChessBoard already contains eight queens)
   1. print one solution.
2. **else**
   1. **loop** (more column c that cell[r, c] is unguarded)
      1. add a queen on cell[r, c]
      2. RecursiveQueens(chessBoard) *// recursivelly continue to add*
         // queens to the next rows.
      3. remove a queen from cell[r, c]

End RecursiveQueens

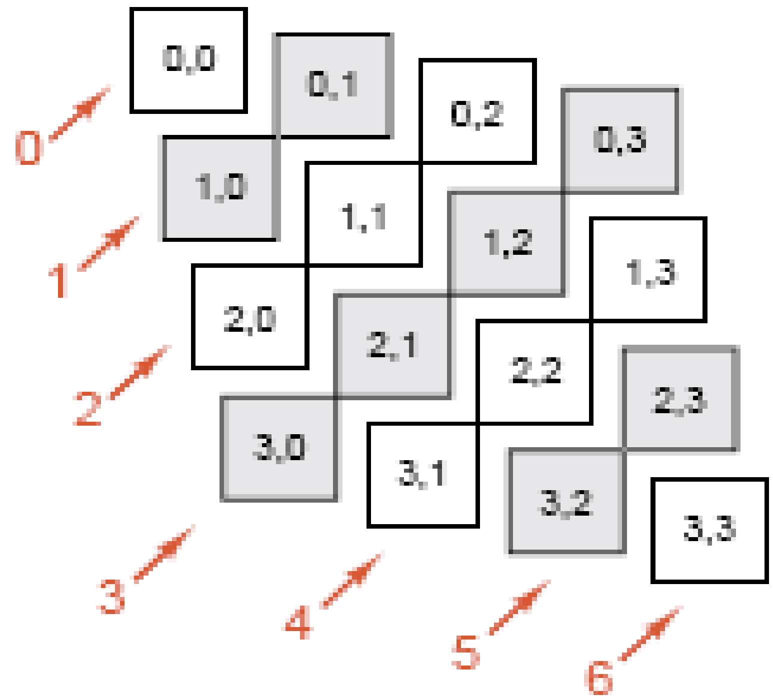
board[r][c]=1

Eight Queens problem



difference = row − column

(d) Downward diagonals



sum = row + column

(e) Upward diagonals

- Data structure for version 2 of class Board

Eight Queens problem

Refinement:

```
class Board_ver2 {


  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
};
```

This implementation keeps arrays to remember which components of the chessboard are free or are guarded.

Eight Queens problem

Algorithm **RecursiveQueens**(val chessBoard <BoardType>,

val r <integer>)

1. **if** (ChessBoard already contains eight queens)
    1. print one solution.
2. **else**
    1. **loop** (more column c that cell[r, c] is unguarded)
        1. add a queen on cell[r, c]
        2. RecursiveQueens(chessBoard) *// recursivelly continue to add*
           // queens to the next rows.
        3. remove a queen from cell[r, c]

End RecursiveQueens

col_free[c] = 0
upward_free[r+c]=0
downward_free[r-c]=0

Eight Queens problem

Algorithm **RecursiveQueens**(val chessBoard <BoardType>,

val r <integer>)

1. **if** (ChessBoard already contains eight queens)
    1. print one solution.
2. **else**
    1. **loop** (more column c that cell[r, c] is unguarded)
        1. add a queen on cell[r, c]
        2. RecursiveQueens(chessBoard) *// recursivelly continue to add*
           // queens to the next rows.
        3. remove a queen from cell[r, c]

End RecursiveQueens

col_free[c] = 1
upward_free[r+c]=1
downward_free[r-c]=1
queen_in_row[r] = c

col_free[c] = 0
upward_free[r+c]=0
downward_free[r-c]=0

Eight Queens problem

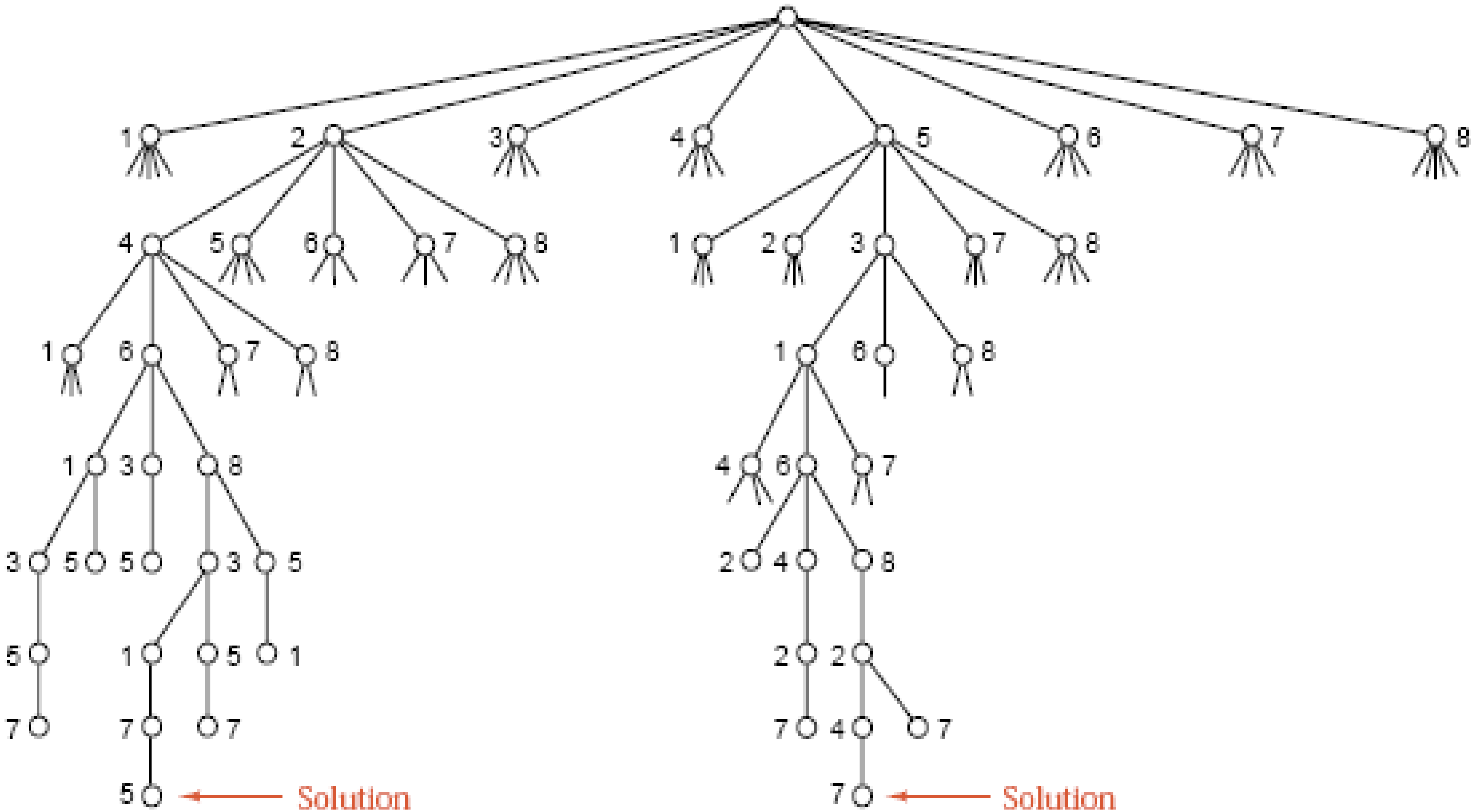
Class Queens_ver1

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

Class Queens_ver2

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

Eight Queens problem



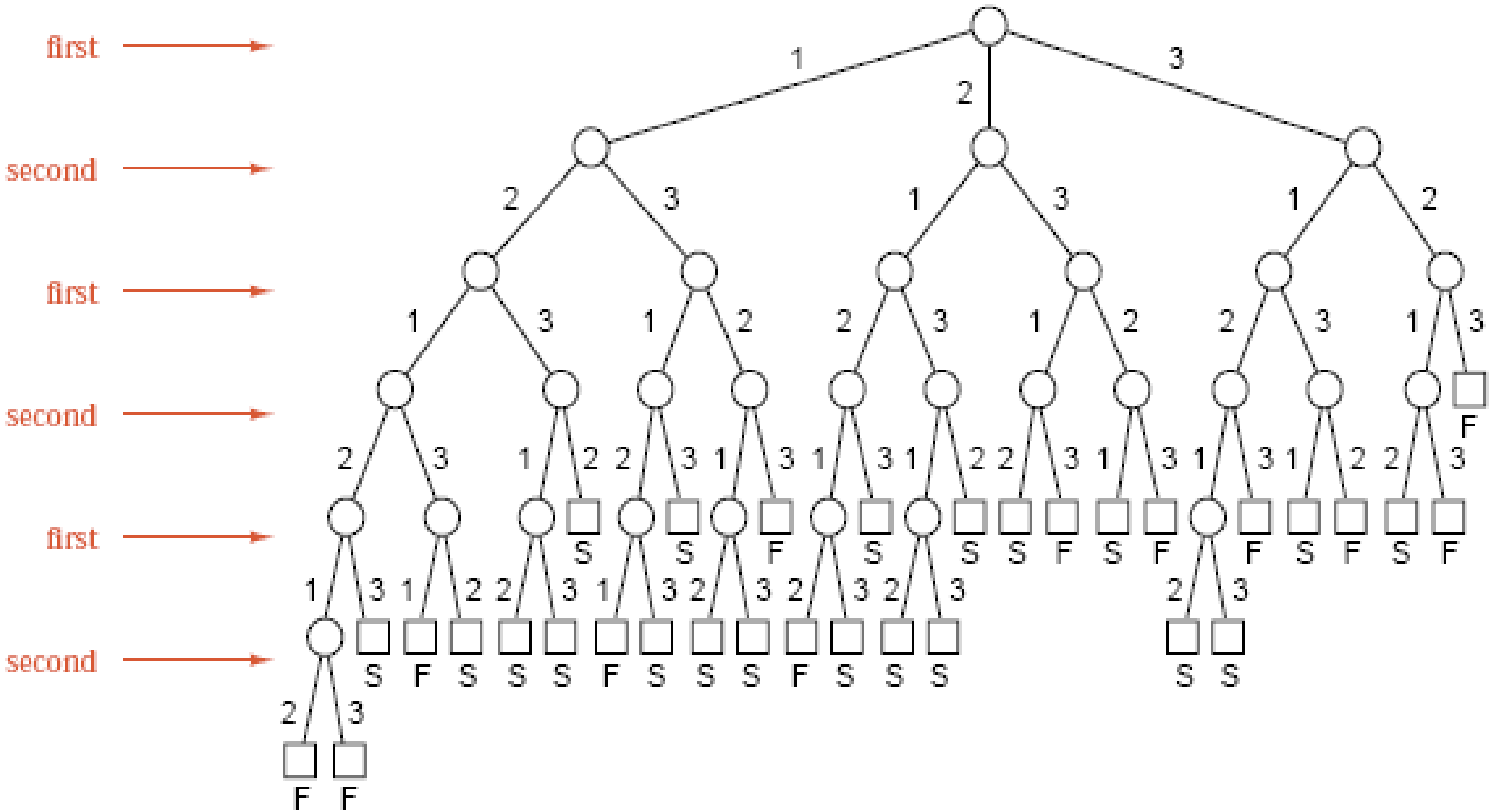
Part of the recursion tree, eight-queens problem

Eight Queens problem

❑Recursion tree for Eight Queens problem

- Each node in the tree might have up to eight children (recusive calls for the eight possible values of column).
- Most of the branches are found to be impossible.
- Backtracking is a most effective tool to prune a recursion tree to manageable size.

Tree-Structured Program



. Tree for the game of Eight

Tree-Structured Program

❑ Look-ahead in Game

Computer algorithm plays a game by looking at moves several steps in advance

❑ Evaluation function

- Examine the current situation and return an integer assessing its benefits.

Tree-Structured Program

❑ Minimax method

- At each node of the tree, we take the evaluation function from the perspective of player who must make the first move.

- First player wishes to maximize the value.

- Second player wishes to maximize the value for oneself, i.e. minimize value for the first player.

- In evaluating a game tree we alternately take minima and maxima, this process called a minimax method

Look-ahead in Game

<integer> **Look_ahead**(val board <BoardType>,
val depth <integer>, ref recommended_move <MoveType>)

Pre board represents a legal game position.

Post An evaluation of the game, based on looking ahead depth moves, is returned. recommended_move contains the best move that can be found for the mover.

Uses Recursive function Look_ahead

```
<integer> Look_ahead(val board <BoardType>,
                    val depth <integer>, ref recommended_move <MoveType>)
1.  if ( (game is over) OR (depth=0) )
    1.  return board.evaluate()          // return an evaluation of the position.
2.  else
    1.  list <ListType>
    2.  Insert into list all legal moves
    3.  best_value = WORST_VALUE //initiate with the value of the worst case
    4.  loop (more data in list) // Select the best option for the mover among
        1.  list.retrieve(tried_move)              // values found in the loop.
        2.  board.play(tried_move)
        3.  value = Look_ahead (board, depth-1, reply) // the returned value
        4.  if (value > best_value)             // of reply is not used at this step.
            1.  best_value = value
            2.  recommended_move = tried_move
        5.  board.unplay(tried_move)
    5.  return best_value
3.  End Look_ahead
```