Chapter 5

RECURSION

1. Introduction to Recursion

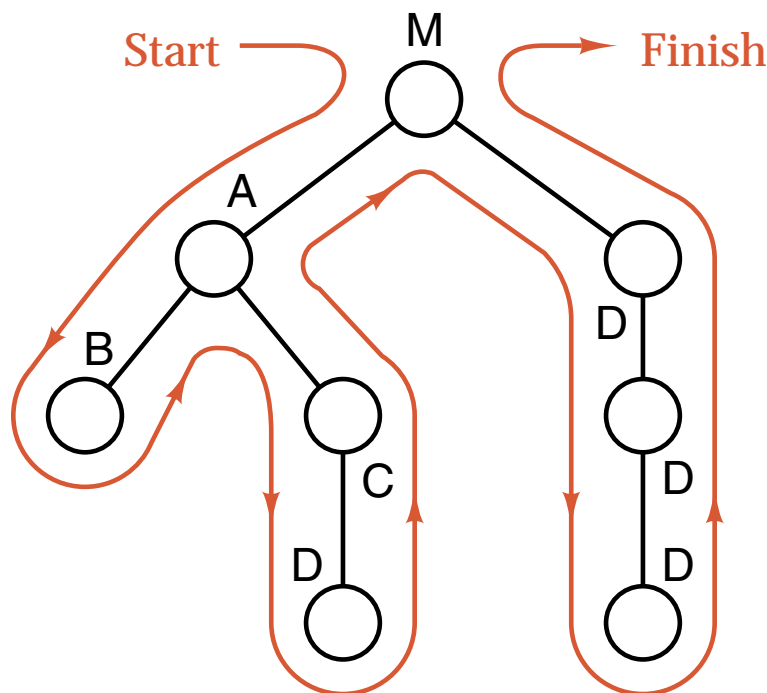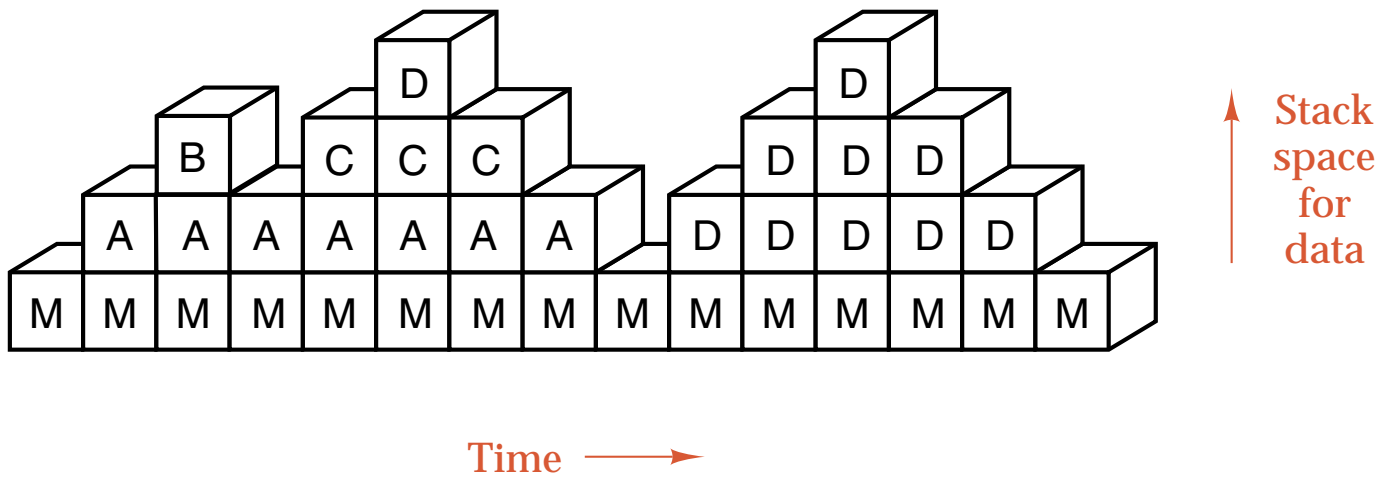   (a) Stack Frames
   (b) Recursion Trees
   (c) Examples

2. Principles of Recursion

3. Backtracking: Postponing the Work

   (a) The Eight-Queens Puzzle
   (b) Review and Refinement
   (c) Analysis

4. Tree-Structured Programs: Look-Ahead in Games

Stacks and Trees

Stack space for data

Time →

Start — M — Finish

THEOREM 5.1 During the traversal of any tree, vertices are added to or deleted from the path back to the root in the fashion of a stack. Given any stack, conversely, a tree can be drawn to portray the life history of the stack, as items are pushed onto or popped from it.

Tree-Diagram Definitions

- The circles in a tree diagram are called ***vertices*** or ***nodes***.

- The top of the tree is called its ***root***.

- The vertices immediately below a given vertex are called the ***children*** of that vertex.

- The (unique) vertex immediately above a given vertex is called its ***parent***. (The root is the only vertex in the tree that has no parent.)

- The line connecting a vertex with one immediately above or below is called a ***branch***.

- ***Siblings*** are vertices with the same parent.

- A vertex with no children is called a ***leaf*** or an ***external vertex***.

- Two branches of a tree are ***adjacent*** if the lower vertex of the first branch is the upper vertex of the second. A sequence of branches in which each is adjacent to its successor is called a ***path***.

- The ***height*** of a tree is the number of vertices on a longest-possible path from the root to a leaf. (Hence a tree containing only one vertex has height 1.)

- The ***depth*** or ***level*** of a vertex is the number of branches on a path from the root to the vertex.

Factorials: A Recursive Definition

Informal definition: The ***factorial*** function of a positive integer is

$$n! = n \times (n-1) \times \cdots \times 1.$$

Formal definition:

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

> Every recursive process consists of two parts:
>
> - A smallest, base case that is processed without recursion; and
>
> - 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.

```
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);
}
```
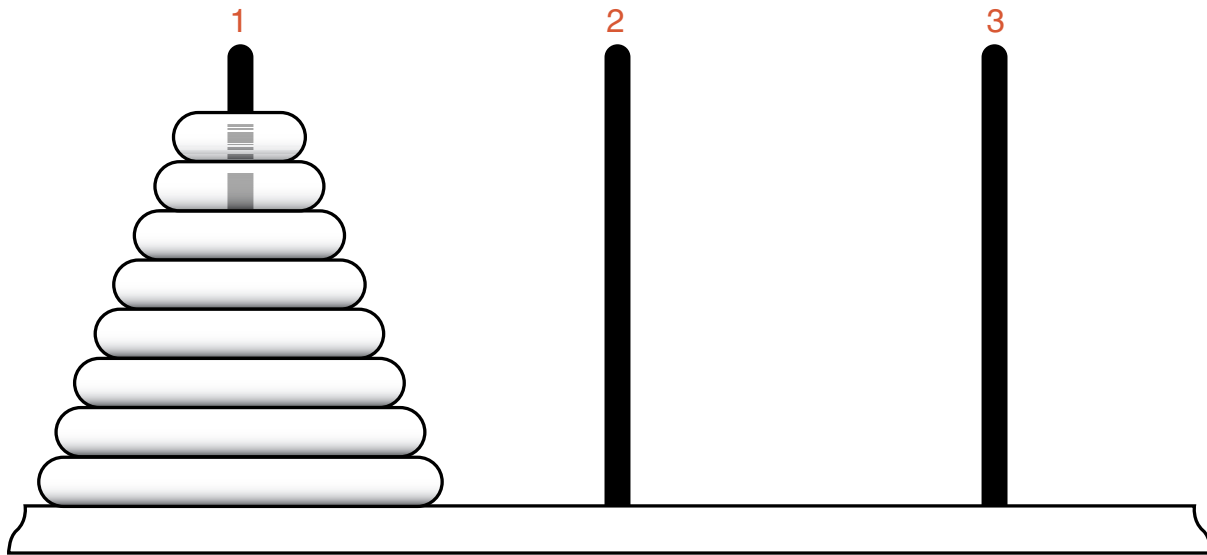
Towers of Hanoi

Rules:

- Move only one disk at a time.

- No larger disk can be on top of a smaller disk.

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

Pre: There are at least count disks on the tower start. The top disk (if any) on each of towers temp and finish is larger than any of the top count disks on tower start.

Post: The top count disks on start have been moved to finish; temp (used for temporary storage) has been returned to its starting position.

Program for Hanoi

Main program:

```
const int disks = 64;      //    Make this constant much smaller to run program.
void move(int count, int start, int finish, int temp);


/* Pre:   None.
   Post:  The simulation of the Towers of Hanoi has terminated.  */

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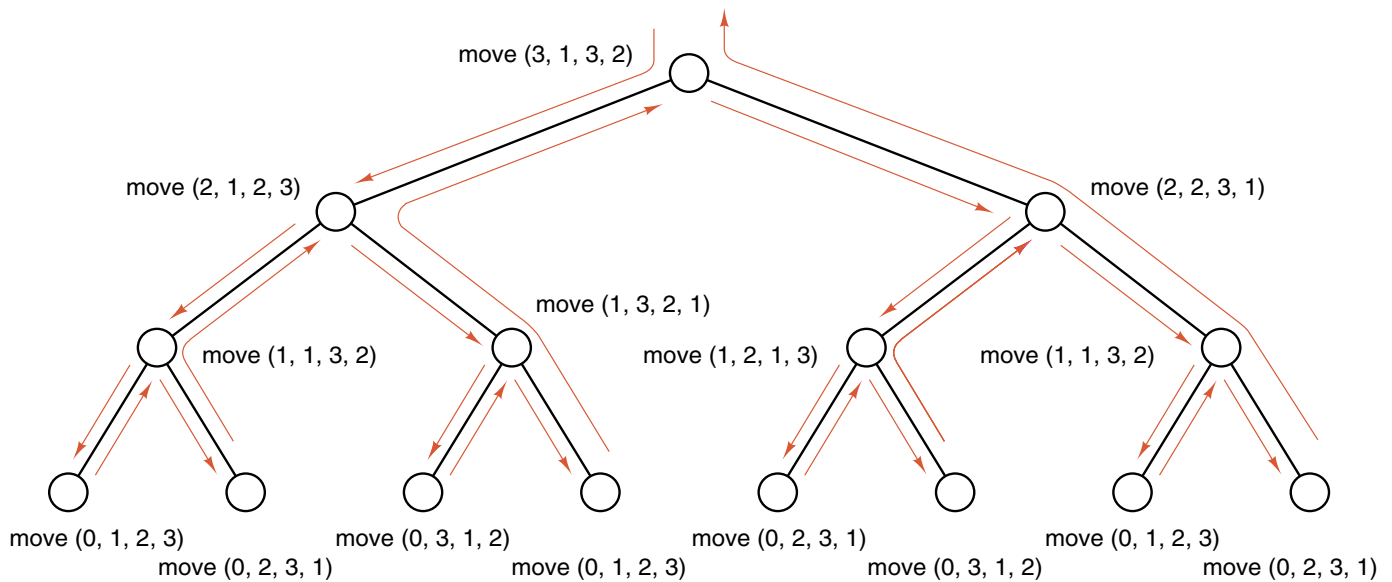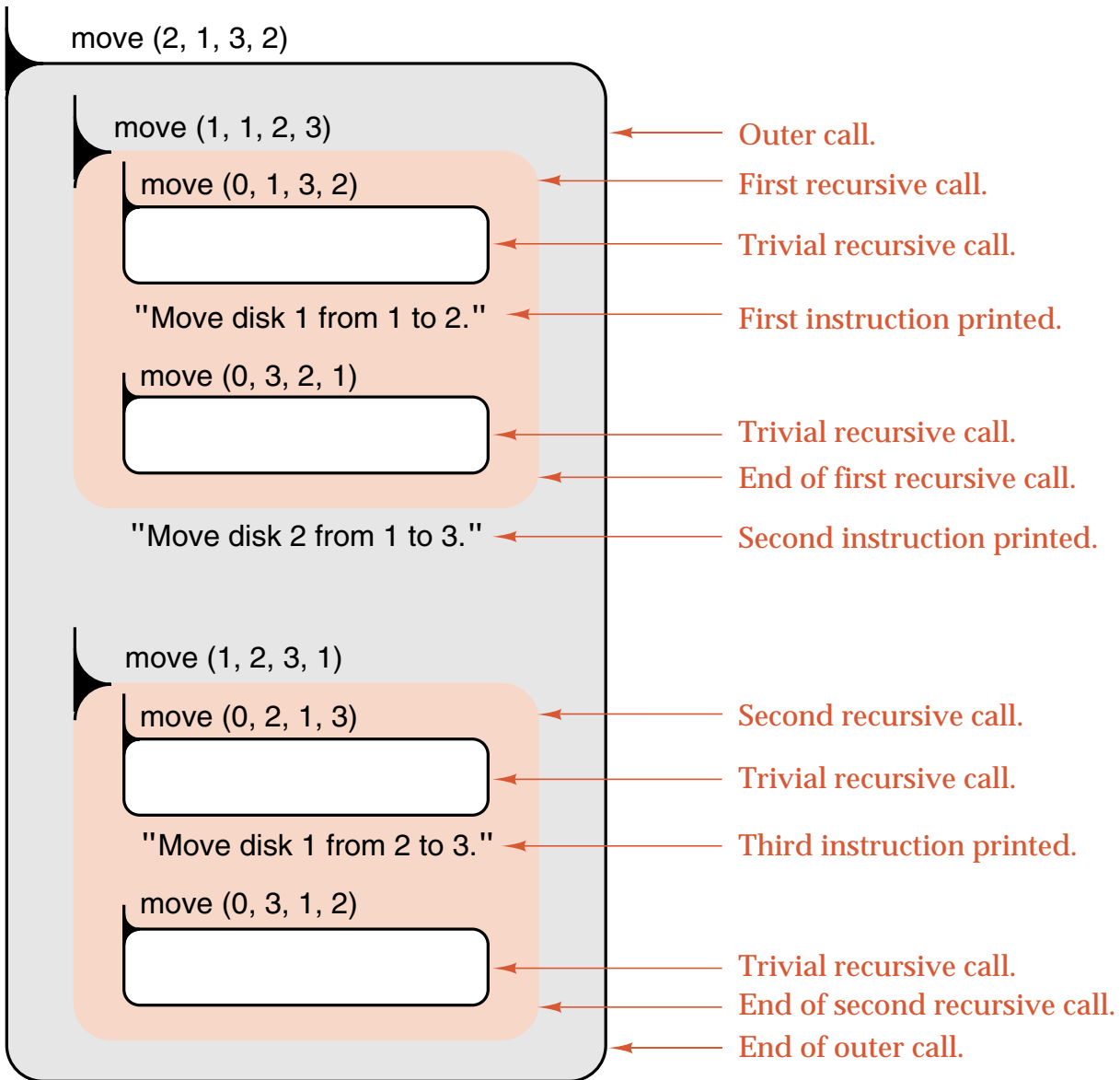
Recursive function:

```
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);
  }
}
```

move (2, 1, 3, 2)

move (1, 1, 2, 3)

move (0, 1, 3, 2)

"Move disk 1 from 1 to 2."

move (0, 3, 2, 1)

"Move disk 2 from 1 to 3."

move (1, 2, 3, 1)

move (0, 2, 1, 3)

"Move disk 1 from 2 to 3."

move (0, 3, 1, 2)

Outer call.

First recursive call.

Trivial recursive call.

First instruction printed.

Trivial recursive call.

End of first recursive call.

Second instruction printed.

Second recursive call.

Trivial recursive call.

Third instruction printed.

Trivial recursive call.

End of second recursive call.

End of outer call.

move (3, 1, 3, 2)

move (2, 1, 2, 3)

move (2, 2, 3, 1)

move (1, 3, 2, 1)

move (1, 1, 3, 2)

move (1, 2, 1, 3)

move (1, 1, 3, 2)

move (0, 1, 2, 3)

move (0, 2, 3, 1)

move (0, 3, 1, 2)

move (0, 1, 2, 3)

move (0, 2, 3, 1)

move (0, 3, 1, 2)

move (0, 1, 2, 3)

move (0, 2, 3, 1)

Designing Recursive Algorithms

- **Find the key step.** Begin by asking yourself, "How can this problem be divided into parts?" or "How will the key step in the middle be done?"

- **Find a stopping rule.** This stopping rule is usually the small, special case that is trivial or easy to handle without recursion.

- **Outline your algorithm.** Combine the stopping rule and the key step, using an **if** statement to select between them.

- **Check termination.** Verify that the recursion will always terminate. Be sure that your algorithm correctly handles extreme cases.

- **Draw a recursion tree.** The height of the tree is closely related to the amount of memory that the program will require, and the total size of the tree reflects the number of times the key step will be done.
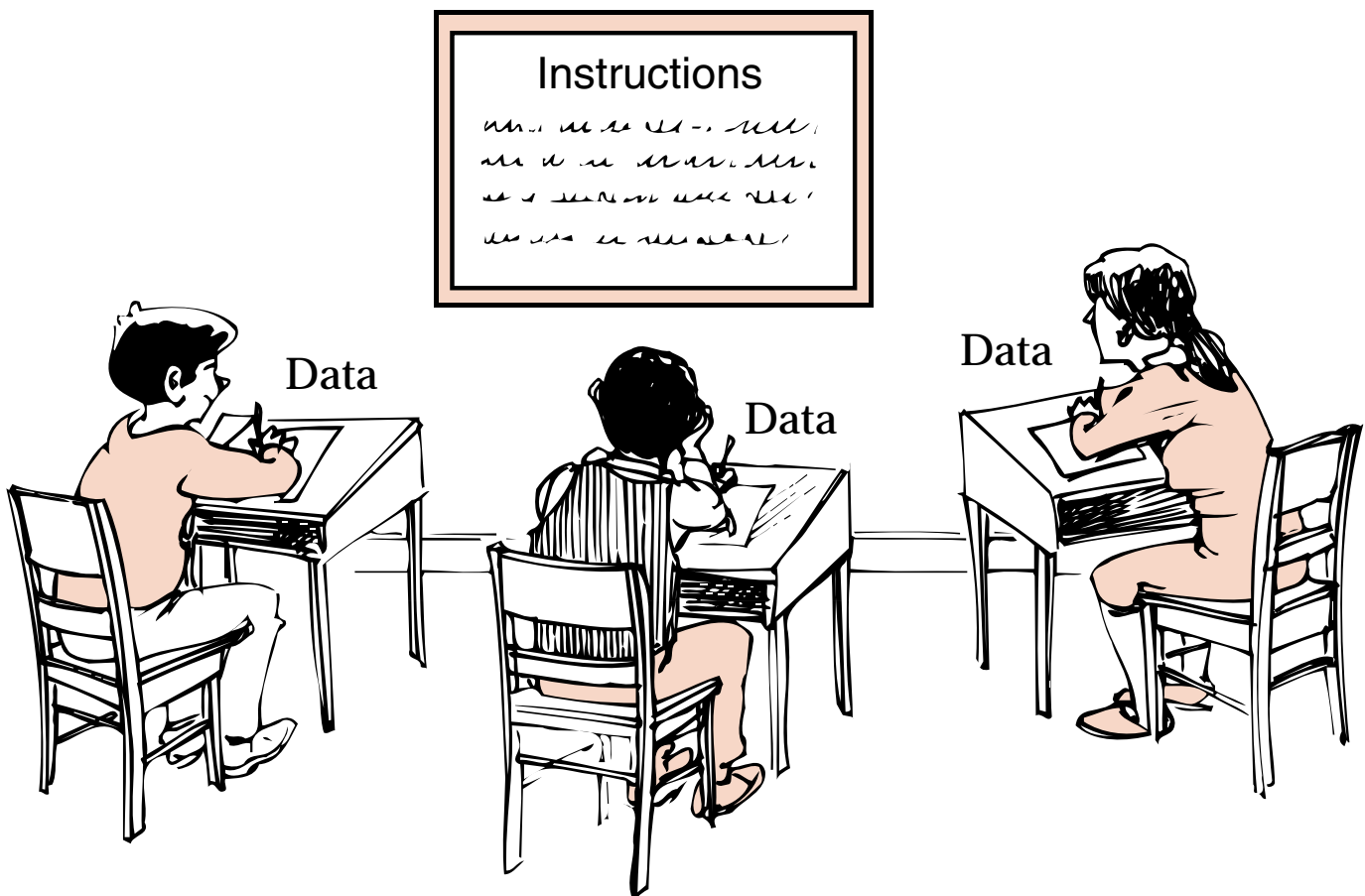
Implementation of Recursion

Multiple Processors: Concurrency:

Processes that take place simultaneously are called ***concurrent***. Recursion can run on multiple processors concurrently.
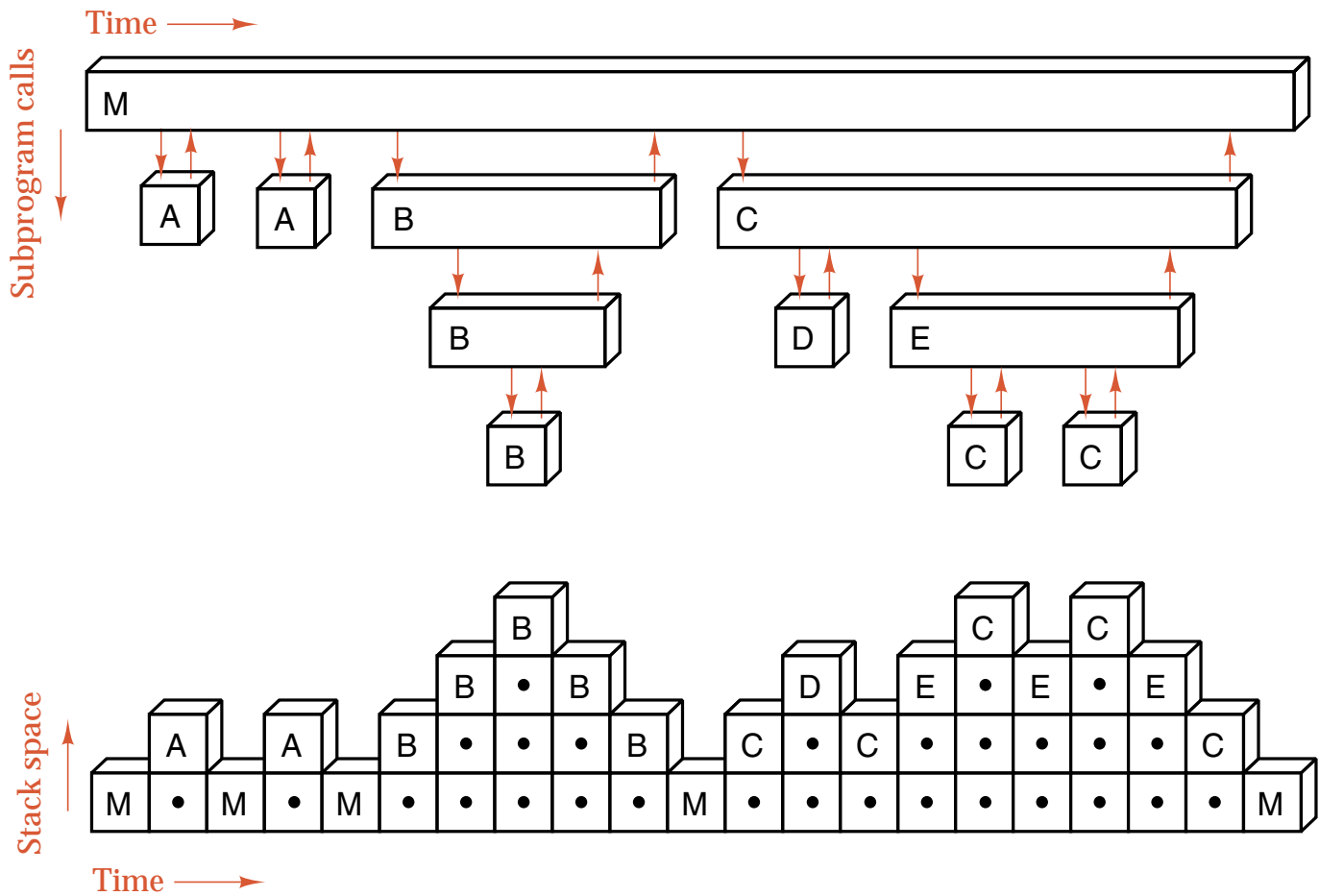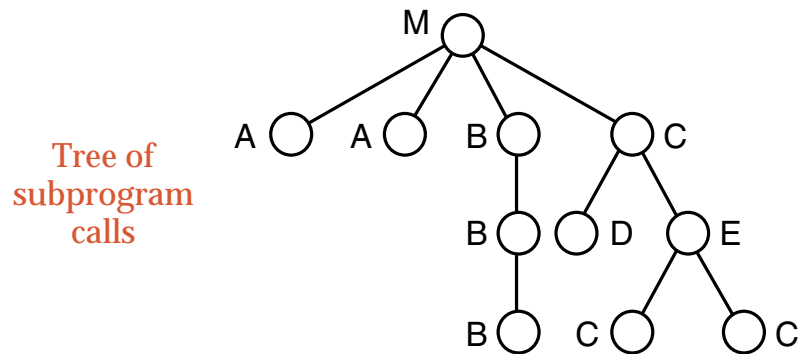
Single Processor Implementation:

Recursion can use multiple storage areas with a single processor.

Example of Re-entrant Processes:

Execution Tree and Stack Frames

Tree of
subprogram
calls

Time →

Subprogram calls

Stack space

Time →

Tail Recursion

DEFINITION **Tail recursion** occurs when the last-executed statement of a function is a recursive call to itself.
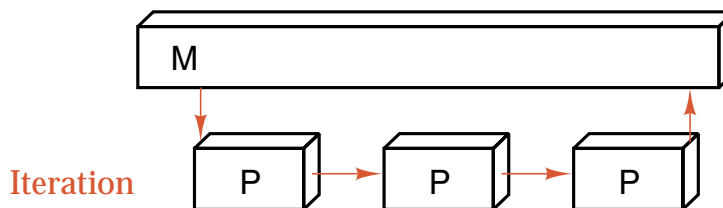
If the last-executed statement of a function is a recursive call to the function itself, then this call can be eliminated by reassigning the calling parameters to the values specified in the recursive call, and then repeating the whole function.

M

P

P

P

Recursion

(a)

M

P

P

P

Tail
recursion

(b)

M

P → P → P
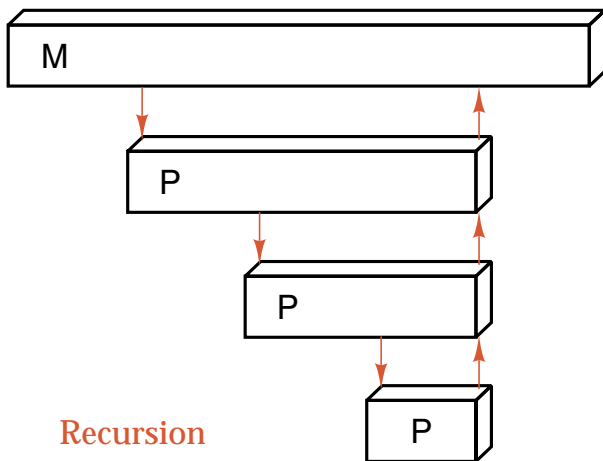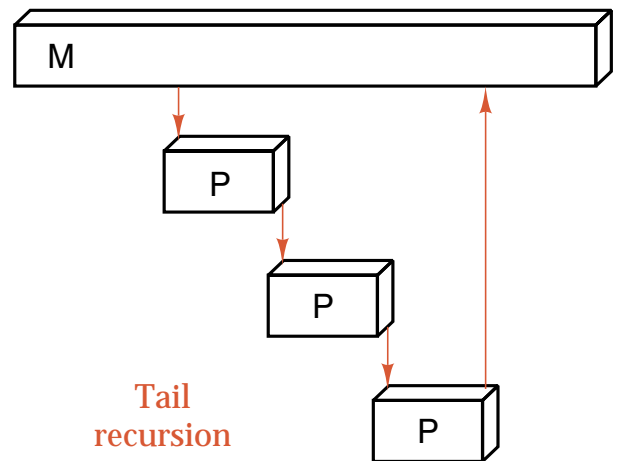
Iteration

(c)

Hanoi Without 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;
  }
}
```

Calculating Factorials
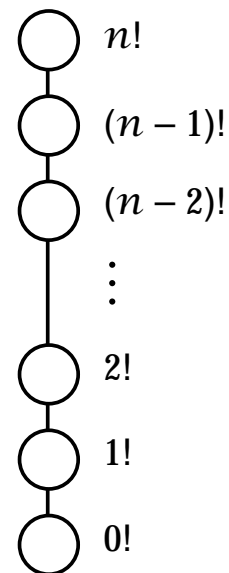
Recursive:

```
int factorial(int n)
{
    if (n ==  0) return 1;
    else         return n * factorial(n − 1);
}
```

Nonrecursive:

```
int factorial(int n)
{
    int count, product = 1;
    for (count = 1;  count <= n;  count++)
        product *= count;
    return product;
}
```

Example:

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

$n!$

$(n − 1)!$

$(n − 2)!$

$\vdots$

$2!$

$1!$

$0!$

Fibonacci Numbers

Definition:

Fibonacci numbers are defined by the recurrence relation

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

Recursive function:

```
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);
}
```

Nonrecursive function:

int fibonacci(**int** n)

/* fibonacci: *iterative version*
 **Pre:**   *The parameter* n *is a nonnegative integer.*
  Post: *The function returns the* n*th 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;
   }
}
```
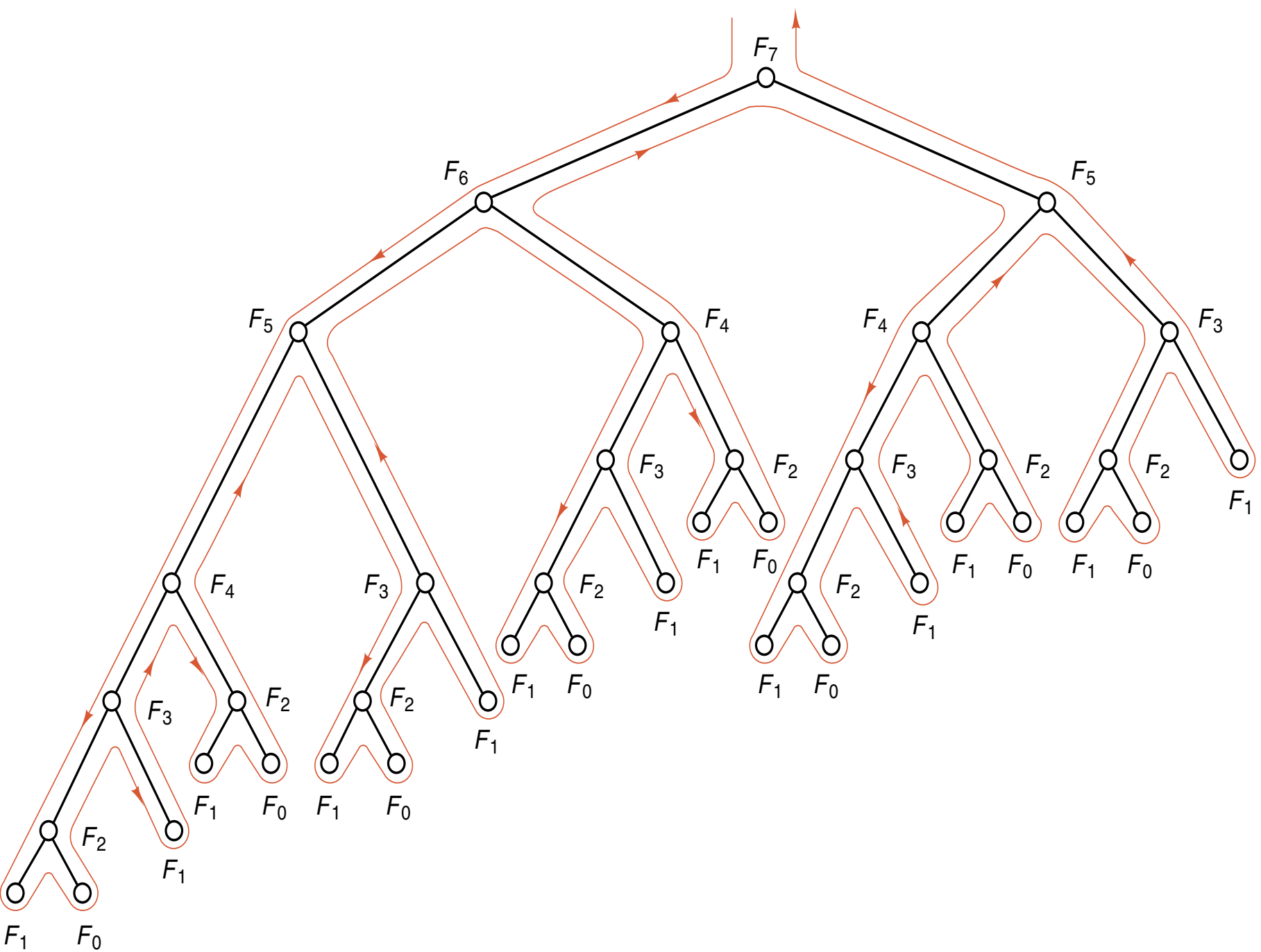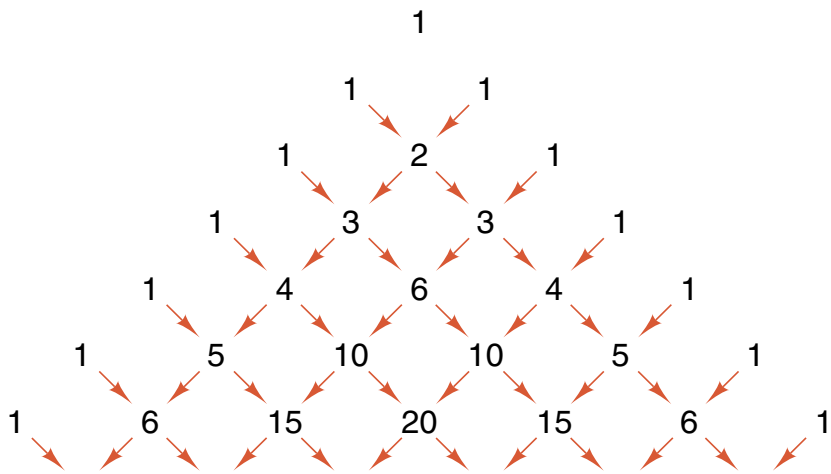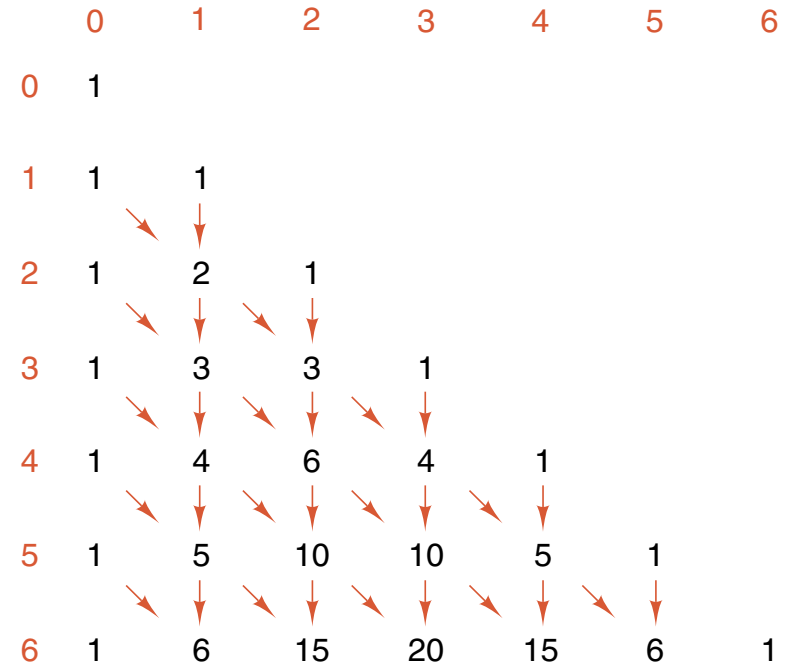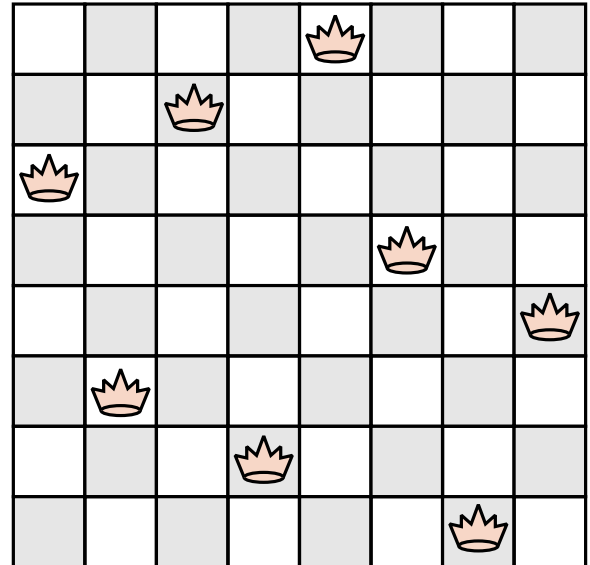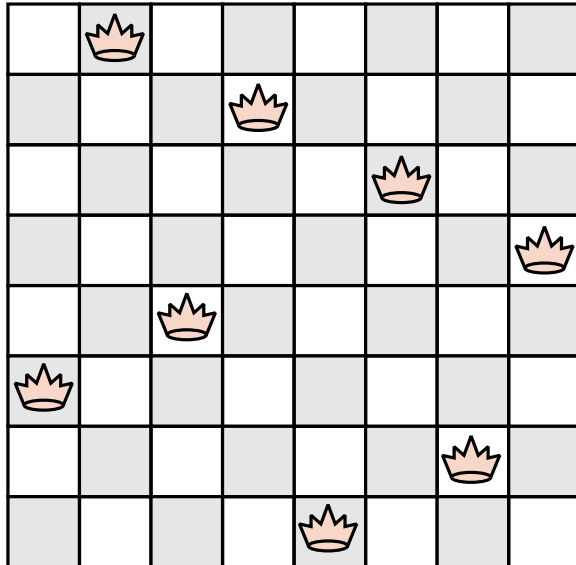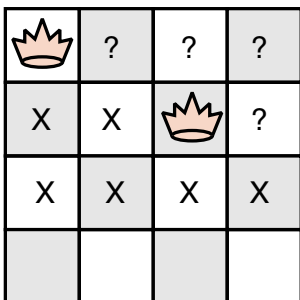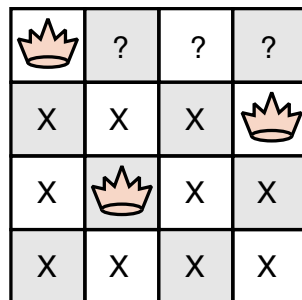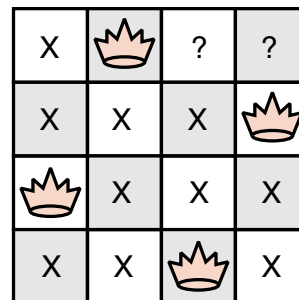
(a) Symmetric form

(b) In square array
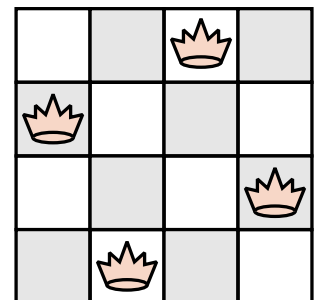
Eight Queens Puzzle

Four Queens Solution

Dead end	Dead end	Solution	Solution
(a)	(b)	(c)	(d)

Program Outline

Recursive function:

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;
    }

Main program:

```
int main( )
/* Pre:    The user enters a valid board size.
   Post:  All solutions to the n-queens puzzle for the selected board size are printed.
   Uses:  The class Queens and the recursive function solve_from.  */
{
   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.
   }
}
```

Methods in the Queens Class

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.

---

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.

---

void Queens :: remove(**int** col);

Pre: There is a queen in the square in row count − 1 and column col.

Post: The above queen has been removed; count has been decremented by 1.

---

bool Queens :: is_solved( ) **const**;

Post: The function returns **true** if the number of queens already placed equals board_size; otherwise, it returns **false**.

The Backtracking Function

void solve_from(Queens &configuration)
/* **Pre:**   *The* Queens configuration *represents a partially completed arrangement of*
                nonattacking queens on a chessboard.

   Post: *All n-queens solutions that extend the given* configuration *are printed. The*
                configuration is restored to its initial state.

   Uses: *The* **class** Queens *and the function* solve_from*, recursively.* */

```
{
  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);   //    Recursively continue to add queens.
        configuration.remove(col);
      }
}
```

The First Data Structure

This implementation uses a square array with **bool** entries.

```
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];
};
```

Sample Results

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

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;** }

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;
}
```

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.* */

```
{
  int i;
  bool ok = true;              //    turns false if we find a queen in column or diagonal
  for (i = 0;  ok && i < count;  i++)
    ok = ! queen_square [i] [col];   //   Check upper part of column
  for (i = 1;  ok && count − i >= 0 && col − i >= 0;  i++)
    ok = ! queen_square [count − i] [col − i];
                                     //   Check upper-left diagonal
  for (i = 1;  ok && count − i >= 0 && col + i < board_size;  i++)
    ok = ! queen_square [count − i] [col + i];
                                     //   Check upper-right diagonal
  return ok;
}
```
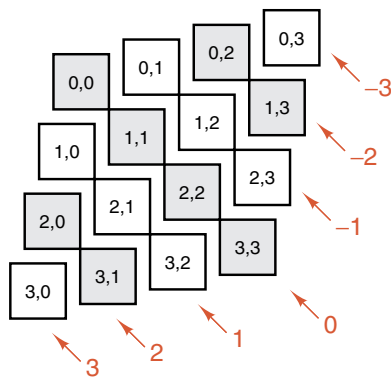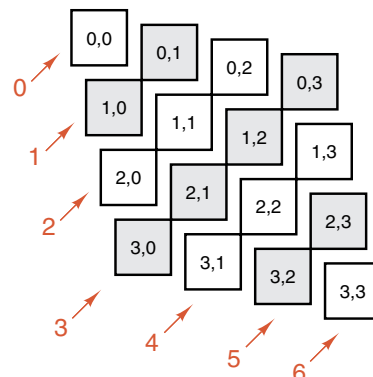
Components of a Chessboard
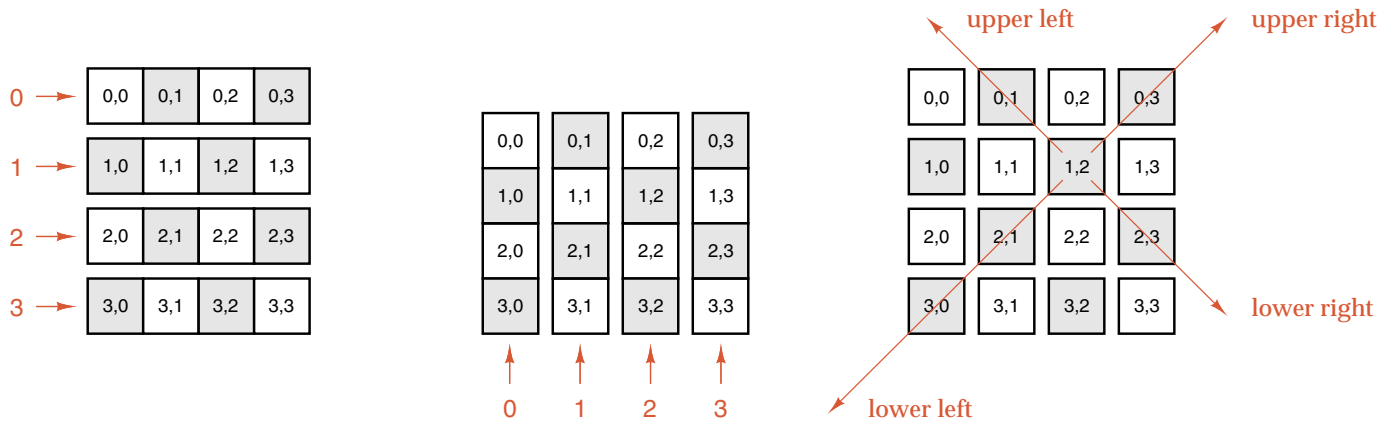
(a) Rows

(b) Columns

(c) Diagonal directions

upper left   upper right

lower left   lower right

difference = row – column

(d) Downward diagonals

sum = row + column

(e) Upward diagonals

Revised Data Structures

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

```cpp
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
};
```

Sample Results

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

Class Declaration

```
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
};
```

Constructor

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;
}
```

Insertion

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++;
}
```

Data Structures and Program Design In C++
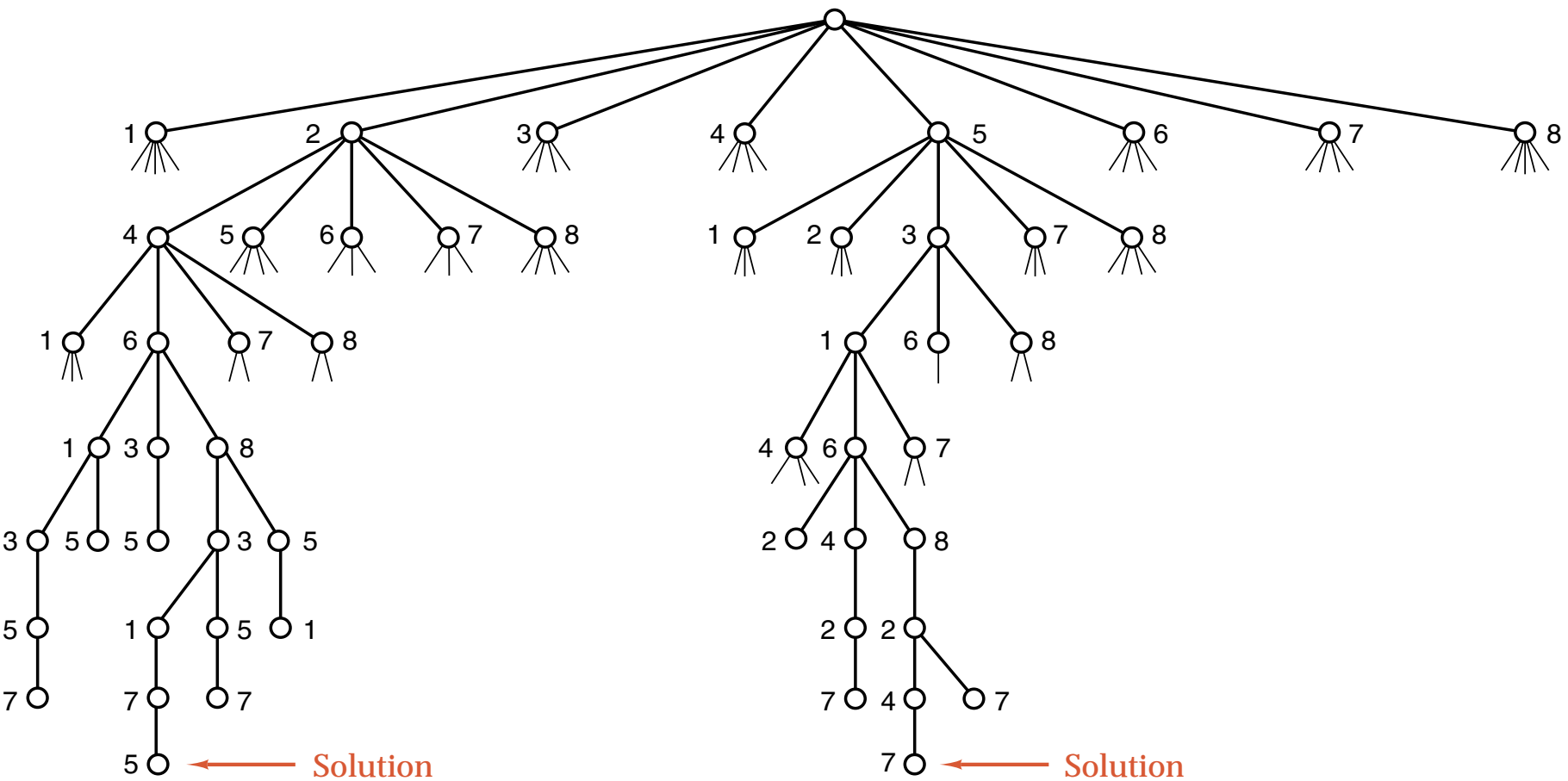
Game Trees
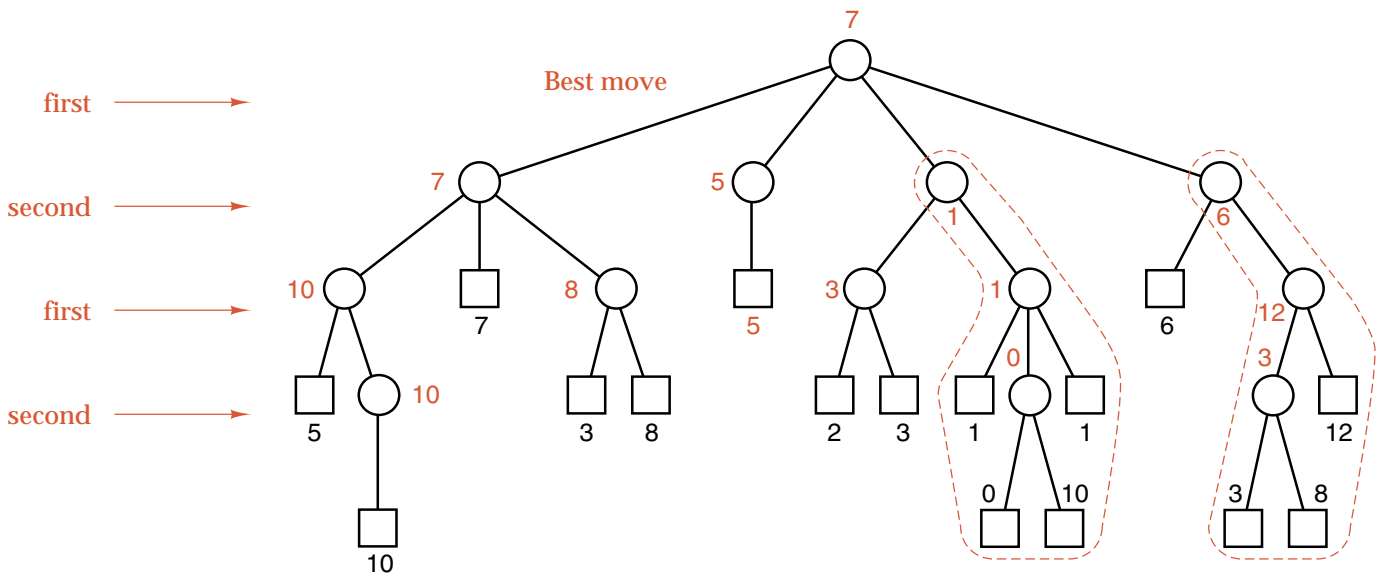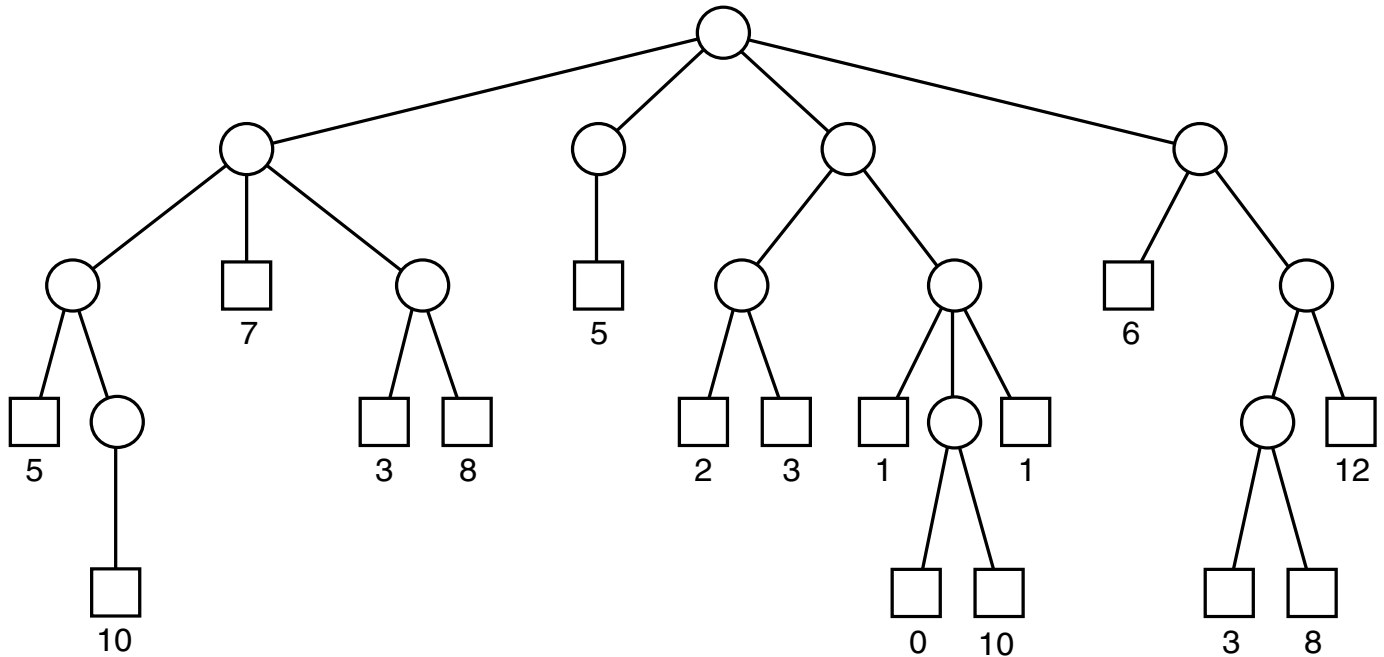
Algorithm Development

Use two classes called Move and Board. A Move object represents a single game move. A Board object represents a single game position.

The **class** Move requires only constructor methods.

The class Board requires methods to *initialize* the Board, to detect whether the game is *over*, to *play* a move that is passed as a parameter, to *evaluate* a position, and to supply a list of all current *legal* moves.
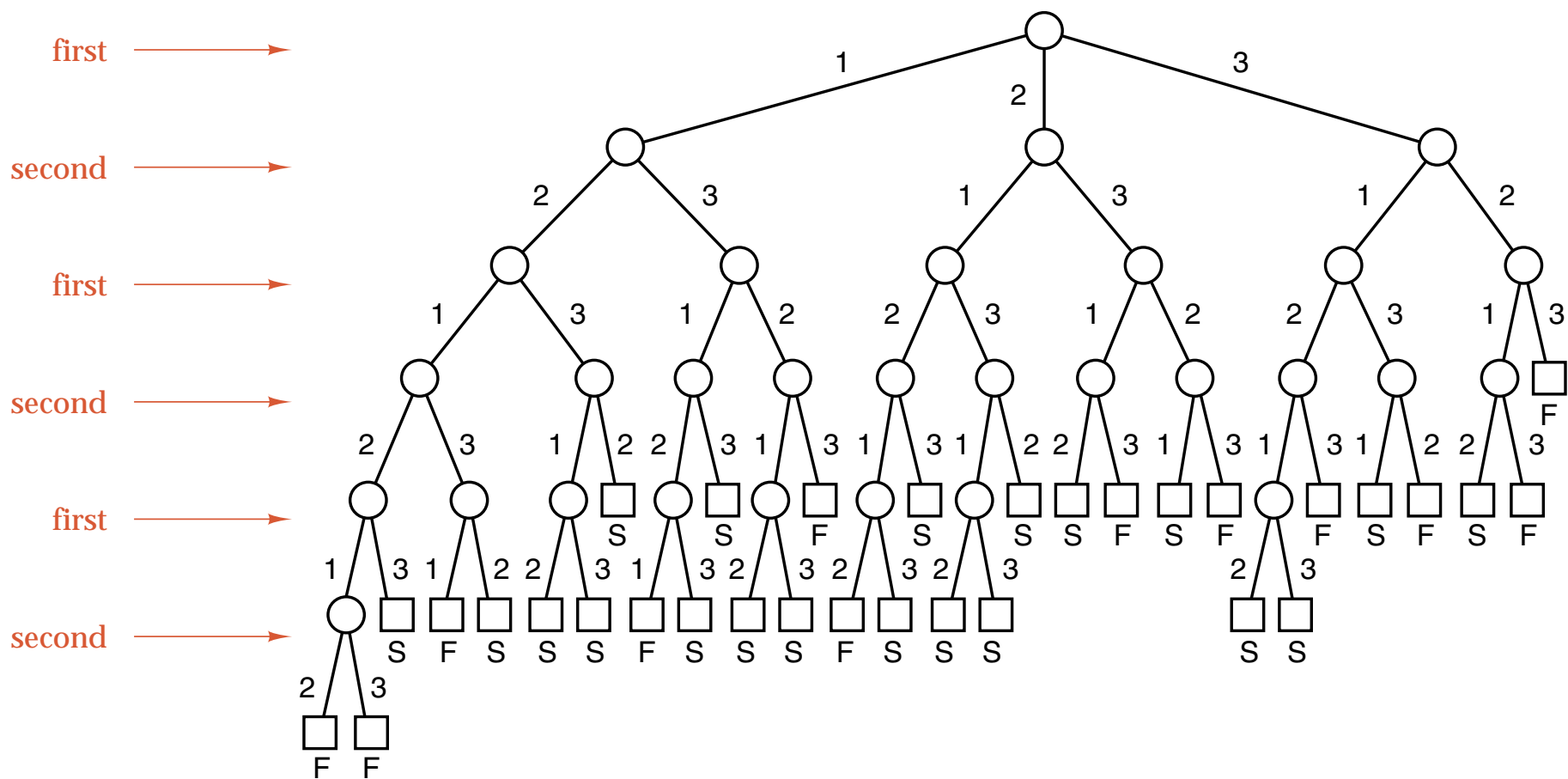
The method legal_moves uses a stack to return the current move options.

The method better uses two integer parameters and returns a nonzero result if the (current) mover would prefer a game value given by the first rather than the second parameter.

The method worst_case returns a constant value that the mover would like less than the value of any possible game position.

```
class Board {
public:
  Board( );                         //    constructor for initialization
  int done( ) const;                //    Test whether the game is over.
  void play(Move try_it);
  int evaluate( ) const;
  int legal_moves(Stack &moves) const;
  int worst_case( ) const;
  int better(int value, int old_value) const;
                                    //    Which parameter does the mover prefer?
  void print( ) const;
  void instructions( ) const;
  /* Additional methods, functions, and data will depend on the game under considera-
      tion. */
};
```

Look-Ahead Algorithm

int look_ahead(**const** Board &game, **int** depth, Move &recommended)
/∗ **Pre:**   Board game *represents a legal game position.*
    Post: *An evaluation of the* game, *based on looking ahead* depth *moves, is re-*
        turned. The best move that can be found for the mover is recorded as Move
        recommended.
    Uses: *The classes* Stack, Board, *and* Move, *together with function* look_ahead
        recursively. ∗/

```
{
  if (game.done( ) || depth ==  0)
    return game.evaluate( );

  else {
    Stack moves;
    game.legal_moves(moves);
    int value, best_value = game.worst_case( );

    while ( !moves.empty( )) {
      Move try_it, reply;
      moves.top(try_it);
      Board new_game = game;
      new_game.play(try_it);
      value = look_ahead(new_game, depth − 1, reply);
      if (game.better(value, best_value)) {
                              //    try_it is the best move yet found
        best_value = value;
        recommended = try_it;
      }
      moves.pop( );
    }
    return best_value;
  }
}
```

Tic-Tac-Toe

- The tic-tac-toe grid is a 3×3 array of integers, with 0 to denote an empty square and the values 1 and 2 to denote squares occupied by the first and second players, respectively.

- A Move object stores the coordinates of a square on the grid.

- The Board class contains **private** data members to record the current game state in a 3×3 array and to record how many moves have been played.

- The game is finished either after nine moves have been played or when one or the other player has actually won.

- The legal moves available for a player are just the squares with a value of 0.

- Evaluate a Board position as 0 if neither player has yet won; however, if one or other player has won, we shall evaluate the position according to the rule that quick wins are considered very good, and quick losses are considered very bad.

- A program that sets the depth of look-ahead to a value of 9 or more will play a perfect game, since it will always be able to look ahead to a situation where its evaluation of the position is exact. A program with shallower depth can make mistakes, because it might finish its look-ahead with a collection of positions that misleadingly evaluate as zero.

The Move Class

```
//    class for a tic-tac-toe move
class Move {
public:
   Move( );
   Move(int r, int c);
   int row;
   int col;
};
```

```
Move :: Move( )
/* Post:  The Move is initialized to an illegal, default value.  */
{
   row = 3;
   col = 3;
}
```

```
Move :: Move(int r, int c)
/* Post:  The Move is initialized to the given coordinates.  */
{
   row = r;
   col = c;
}
```

The Board Class

```cpp
class Board {
public:
   Board( );
   bool done( ) const;
   void print( ) const;
   void instructions( ) const;
   bool better(int value, int old_value) const;
   void play(Move try_it);
   int worst_case( ) const;
   int evaluate( ) const;
   int legal_moves(Stack &moves) const;
private:
   int squares[3][3];
   int moves_done;
   int the_winner( ) const;
};
```

Constructor

```cpp
Board :: Board( )
/* Post:  The Board is initialized as empty.  */

{
   for (int i = 0;  i < 3;  i++)
      for (int j = 0;  j < 3;  j++)
         squares[i][j] = 0;
   moves_done = 0;
}
```

```
void Board :: play(Move try_it)
/* Post:  The Move try_it is played on the Board.  */

{
   squares[try_it.row][try_it.col] = moves_done % 2 + 1;
   moves_done++;
}


int Board :: the_winner() const
/* Post:  Return either a value of 0 for a position where neither player has won, a value
          of 1 if the first player has won, or a value of 2 if the second player has won.  */

{
   int i;
   for (i = 0; i < 3; i++)
      if (squares[i][0]  && squares[i][0] == squares[i][1]
                         && squares[i][0] == squares[i][2])
            return squares[i][0];

   for (i = 0; i < 3; i++)
      if (squares[0][i]  && squares[0][i] == squares[1][i]
                         && squares[0][i] == squares[2][i])
            return squares[0][i];

   if (squares[0][0]  && squares[0][0] == squares[1][1]
                      && squares[0][0] == squares[2][2])
         return squares[0][0];

   if (squares[0][2]  && squares[0][2] == squares[1][1]
                      && squares[2][0] == squares[0][2])
         return squares[0][2];
   return 0;
}
```

bool Board :: done( ) **const**
/* **Post:** *Return* **true** *if the game is over; either because a player has already won or because all nine squares have been filled.* */

```
{
  return moves_done ==  9 || the_winner( ) > 0;
}
```

int Board :: legal_moves(Stack &moves) **const**
/* **Post:** *The parameter* Stack moves *is set up to contain all possible legal moves on the* Board. */

```
{
  int count = 0;
  while ( ! moves.empty( )) moves.pop( );
  for (int i = 0;  i < 3;  i++)
    for (int j = 0;  j < 3;  j++)
      if (squares [i] [j] ==  0) {
        Move can_play(i, j);
        moves.push(can_play);
        count++;
      }
  return count;
}
```

int Board :: evaluate( ) **const**
/* **Post:** *Return either a value of 0 for a position where neither player has won, a positive value between 1 and 9 if the first player has won, or a negative value between* − 1 *and* − 9 *if the second player has won,* */

```
{
  int winner = the_winner( );
  if (winner ==  1) return 10 − moves_done;
  else if (winner ==  2) return moves_done − 10;
  else return 0;
}
```

Pointers and Pitfalls

1. Recursion should be used freely in the initial design of algorithms. It is especially appropriate where the main step toward solution consists of reducing a problem to one or more smaller cases.

2. Study several simple examples to see whether or not recursion should be used and how it will work.

3. Attempt to formulate a method that will work more generally. Ask, "How can this problem be divided into parts?" or "How will the key step in the middle be done?"

4. Ask whether the remainder of the problem can be done in the same or a similar way, and modify your method if necessary so that it will be sufficiently general.

5. Find a stopping rule that will indicate that the problem or a suitable part of it is done.

6. Be very careful that your algorithm always terminates and handles trivial cases correctly.

7. The key tool for the analysis of recursive algorithms is the recursion tree. Draw the recursion tree for one or two simple examples appropriate to your problem.

8. The recursion tree should be studied to see whether the recursion is needlessly repeating work, or if the tree represents an efficient division of the work into pieces.

9. A recursive function can accomplish exactly the same tasks as an iterative function using a stack. Consider carefully whether recursion or iteration with a stack will lead to a clearer program and give more insight into the problem.

10. Tail recursion may be removed if space considerations are important.

11. Recursion can always be translated into iteration, but the general rules will often produce a result that greatly obscures the structure of the program. Such obscurity should be tolerated only when the programming language makes it unavoidable, and even then it should be well documented.

12. Study your problem to see if it fits one of the standard paradigms for recursive algorithms, such as divide and conquer, backtracking, or tree-structured algorithms.

13. Let the use of recursion fit the structure of the problem. When the conditions of the problem are thoroughly understood, the structure of the required algorithm will be easier to see.

14. Always be careful of the extreme cases. Be sure that your algorithm terminates gracefully when it reaches the end of its task.

15. Do as thorough error checking as possible. Be sure that every condition that a function requires is stated in its preconditions, and, even so, defend your function from as many violations of its preconditions as conveniently possible.