```
FUNCTION factorial:
  INPUT: integer n >= 0
  OUTPUT: [n x (n-1) x (n-2) x … x 1]
  USAGE: res = factorial(n)
BEGIN

  IF n is 0, RETURN 1
  ELSE, RETURN [n x factorial(n-1)]

END // factorial
```

ALGORITHMS AND DATA STRUCTURES

RECURSION (I)

UP TO NOW…

- We have control flow structures to solve problems
- **Control flow** corresponds to the order in which statements, instructions, function calls, etc. are evaluated or executed
  - Conditional control (conditional execution)
  - Iterative control (iterative execution)

AND NOW…

- We introduce a new strategy to solve problems
- **Recursion** is a technique that solves problems by reducing them to smaller problems *of the same form*
- It almost look like the definition of stepwise refinement
  - except for the *italicized* part of the definition of recursion
  - however, both (recursion and stepwise refinement) involve decomposition

THE RECURSIVE STRATEGY

- When using recursion to solve a problem we must answer the questions [details following slides]:
  1. What is the base case?
  2. What to do in the base case?
  3. What is the general case?
  4. What is the reduction?
  5. Does the reduction lead to termination?

CONTROL FLOW STATEMENTS

- tell us in what order statements/instructions are executed
- allow to make choices from several “instruction paths”
- offer a way of controlling the way a program flows
  - **Iterative control flow statements**
    - Control flow statements for specifying iteration
    - In C++ we have `for`, `while`, and `do/while`

RECURSION

- Another way of performing a block of instructions repeatedly
- The solution of the problem depends on solutions to smaller instances of the same problem
- A function in said to be recursive if it can call itself within its own definition
- It's another way of controlling structure of functions

RECURSION: EXAMPLE IN KAREL

- Problem: make the Robot face North regardless of initial position

```
DEFINE-NEW-INSTRUCTION face-north AS
  BEGIN
    WHILE not-facing-north DO
      BEGIN
        turnleft
      END
  END;
```

Iterative solution

```
DEFINE-NEW-INSTRUCTION face-north AS
  BEGIN
    IF not-facing-north THEN
      BEGIN
        turnleft;
        face-north
      END
  END;
```

Recursive solution

- What's different? The difference is subtle!

Richard E. Pattis, *Karel the Robot*. Wiley & Sons (1981)

HOW TO WRITE RECURSIVE FUNCTIONS

1. Consider the stopping condition, also called the base case (Karel already facing north)
2. What is needed to do in the base case? (For the facing north problem, nothing in this case)
3. Consider the problem if not in the base case and reduce it to small tasks (Karel not facing north. Reduction: turnleft)
4. Make sure the reduction leads to the base case (By turning left Karel eventually faces north)

DIFFERENCE BETWEEN ITERATION AND RECURSION

- An iterative loop must execute each iteration completely before the next one
- A recursive instruction typically begins a new instance before completing the current one
- Since each progressive instance is supposed to make small progress towards the base case, we should not use loops to control recursive calls
  - We should use instead `if`, `if/else`, `if/elif`, or `if/elif/else` instructions in the body of the recursive function

EXERCISE: MOVE KAREL TO A BEEPER

1. What is the base case? *Karel is on the beeper*
2. What does Karel have to do in the base case? *Nothing*
3. What is the general case? *Karel is not on the beeper*
4. What is the reduction? *Move toward the beeper and make the recursive call*
5. Does the reduction lead to termination? *Yes, assuming the beeper is right in front of Karel*

STRUCTURE OF RECURSIVE FUNCTIONS

- The recursive paradigm

```
FUNCTION recursive_function:
 INPUT: <input>
 OUTPUT: <output>
 USAGE: <output> = recursive_function(<input>)
BEGIN

 IF test-for-simple-case
  compute solution without recursion
 ELSE
  break the problem in small subproblems
  solve each subproblem by calling recursive_function(<input>)
  reassemble the subproblem solutions into the whole solution

END // recursive_function
```

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

EXAMPLE: COLLECTING CONTRIBUTIONS

- Let’s say you want to raise $1,000,000. How would you do it? Ask for small amounts of money to a lot of people!
- Semi-pseudocode

```
int collectContributions(int n)
{
 if (n <= 100)
  collect money from a single donor
 else {
  find 10 volunteers
  get each one to collect n / 10 dollars
  combine the money raised by the 10 volunteers
 }
}
```

EXAMPLE: COLLECTING CONTRIBUTIONS

- Let's say you want to raise $1,000,000. How would you do it? Ask for small amounts of money to a lot of people!
- Semi-pseudocode

```
int collectContributions(int n)
{
 if (n <= 100)
  money = collectFromSingleDonor();
 else
  // function calls collectContributions(n / 10)
  money = collectContributionsFrom10Vols(n / 10);

 return money;
}
```

DIVIDE AND CONQUER

- When the solution depends on dividing hard problems into simpler instances of the same problem, recursive solutions of this form are called divide-and-conquer algorithms
- We apply three steps at each level of recursion
  1. **Divide** the problem into smaller instances
  2. **Conquer** the subproblems by solving them recursively
  3. **Combine** the subsolutions into the whole solution

EXAMPLE: FACTORIAL

- Let's follow the callings in these two implementations

```
int factorial(int n)
{
  int result = 1;

  for (int i = 1; i <= n; i++) {
    result *= i;
  }

  return result;
}
```

Iterative solution

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

Recursive solution

THE LEAP OF FAITH

- Take the `factorial(n)` example:
  - analyzing small `n`’s can be done
  - large `n`’s are much harder to follow
- When understanding a recursive program
  - useful to put small details aside
  - focus on a single level of the operation
  - assume that further recursive calls are correct

THE LEAP OF FAITH

- assume that further recursive calls are correct
  - will be true as long as call's arguments of the call are simpler, in some sense
- This strategy is called the **recursive leap of faith**
  - *meaning*: Assume that simpler recursive calls will work correctly
  - it is essential to using recursion effectively

EXAMPLE: FIBONACCI SEQUENCE

- Interest in how rabbit population grows over time
- Model of population growing defined by
  - One pair of fertile rabbits produces one pair per month
  - Rabbits become fertile in their second month
  - Rabbits are immortal
- Results in: 0, 1, 1, 2, 3, 5, …

EXAMPLE: FIBONACCI SEQUENCE

- Important observations:
  - Rabbits never die, then all rabbits from previous moth are still around
  - All fertile rabbits produced a new pair, that number is the number of rabbits alive in month before previous one
  - *Final effect*: new term in the sequence is the sum of rabbits in two previous months

EXAMPLE: FIBONACCI SEQUENCE

- The Fibonacci sequence is defined in terms of the recurrence relation

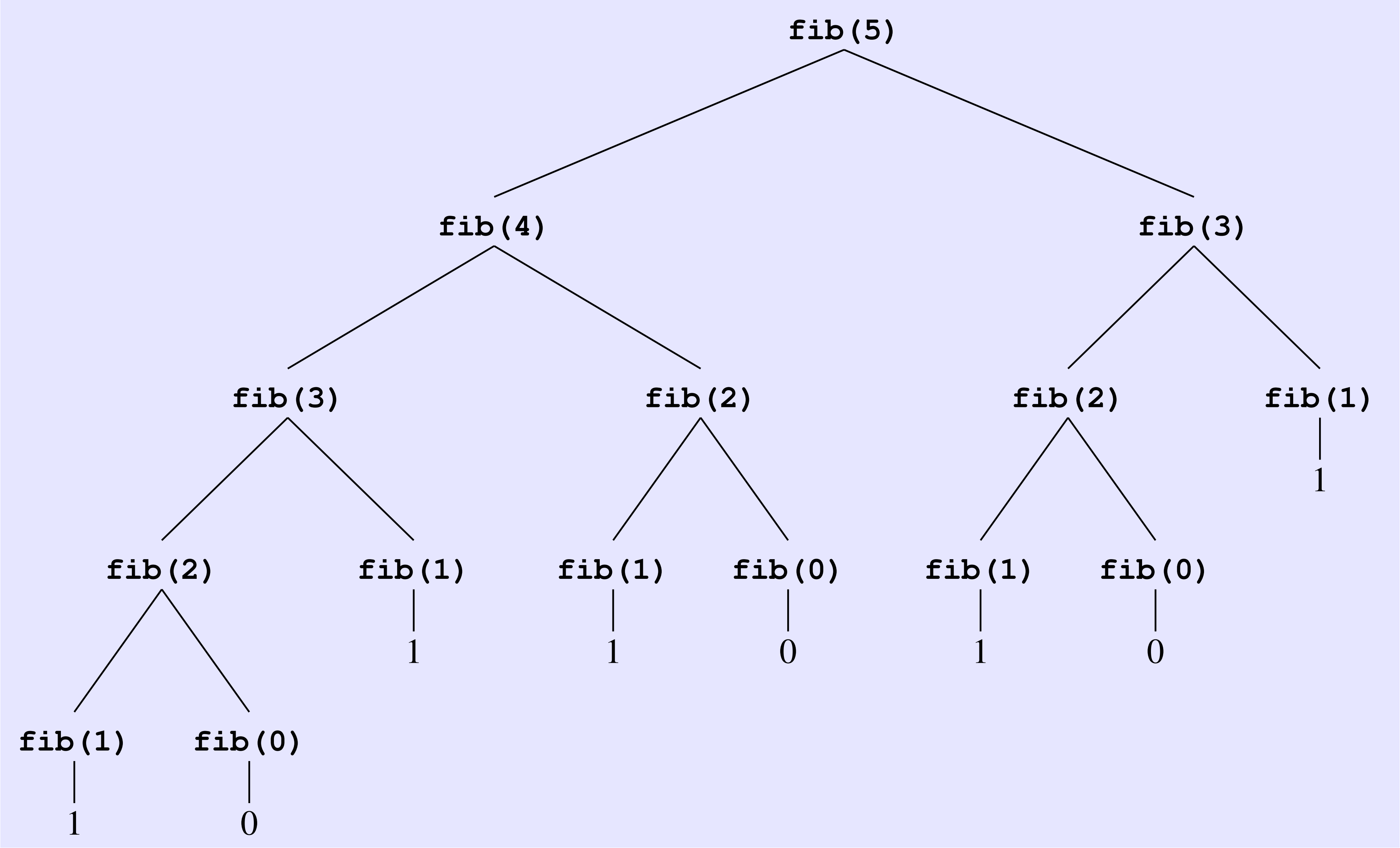
```
f[n] = 1,                 n = 0, 1
f[n] = f[n-1] + f[n-2], n > 1
```

- Code:

```
int fibonacci(int n)
{
 if (n < 2)
   return n;
 else
   return fibonacci(n-1) + fibonacci(n-2);
}
```

EFFICIENCY OF THE RECURSION IMPLEMENTATION

- Extremely inefficient way of getting f[n]



Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.

EFFICIENCY OF THE RECURSION IMPLEMENTATION

- Recursion is not to blame. It is the way the sequence is defined and used
- Way out: adopt a more general approach to the problem
  - usually the case when using recursion
- Fibonacci sequence is a subset of more general sequences called *additive sequences*
- Turn the problem into finding the *n*-th term in an additive sequence with given starting points

ADDITIVE SEQUENCES

- are a class of sequences that satisfy the same recurrence relation, but differ in the initial conditions
- are Fibonacci sequences that only differ in the initial terms `f[0]` and `f[1]`
  - Example:

```
f[n] = -1,                    n = 0
f[n] = 2,                     n = 1
f[n] = f[n-1] + f[n-2], n > 1
```

```
-1, 2, 1, 3, 4, 7, 11, 18, 29, 47, 76, …
```

HOW TO IMPLEMENT AN ADDITIVE SEQUENCE

- The key is to realize that the n−th term is the $(n-1)$−th term with the next initial conditions
- Example for the Fibonacci sequence

$$
\begin{array}{ccccccccccc}
t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & t_9 & \\
3 & 7 & 10 & 17 & 27 & 44 & 71 & 115 & 186 & 301 & \ldots
\end{array}
$$

$$
\begin{array}{cccccccccc}
t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & \\
7 & 10 & 17 & 27 & 44 & 71 & 115 & 186 & 301 & \ldots
\end{array}
$$

- The terms get shifted by one place to the left!

RECURSIVE IMPLEMENTATION OF THE ADDITIVE SEQUENCE

- applied to the Fibonacci sequence

```
int additiveSequence(int n, int f0, int f1)
{
 if (n == 0) return f0;
 if (n == 1) return f1
 return additiveSequence(n - 1, f1, f0 + f1);
}

// wrapper function
int fibonacci(int n)
{
 return additiveSequence(n, 0, 1);
}
```

- No unnecessary computations are performed!

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.