Data Structures (in C++)

- Recursion -

Jinsun Park

Visual Intelligence and Perception Lab., CSE, PNU

Recursion

Recursion

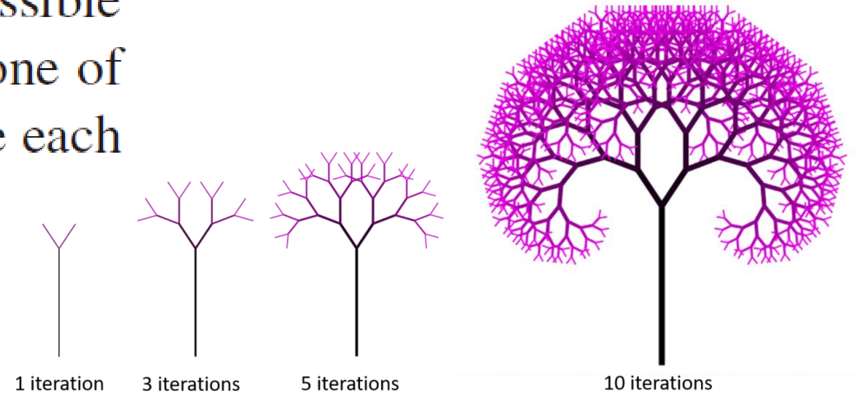
- **Recursive Function**
  - A function is called within the same function
  - Effective for problems with repetitive structures

- ***Test for base cases.*** We begin by testing for a set of base cases (there should be at least one). These base cases should be defined so that every possible chain of recursive calls eventually reaches a base case, and the handling of each base case should not use recursion.
- ***Recur.*** After testing for base cases, we then perform a single recursive call. This recursive step may involve a test that decides which of several possible recursive calls to make, but it should ultimately choose to make just one of these calls each time we perform this step. Moreover, we should define each possible recursive call so that it makes progress towards a base case.

https://medium.com/@wiemzin/getting-started-with-recursion-schemes-using-matryoshka-f5b5ec01bb

Matryoshka (Russian doll)

http://bricault.mit.edu/recursive-drawing



Fractal Trees

Recursion Application: Factorial

- **The Factorial Function**

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$ for any integer $n \geq 0$

Base case (nonrecursive case)

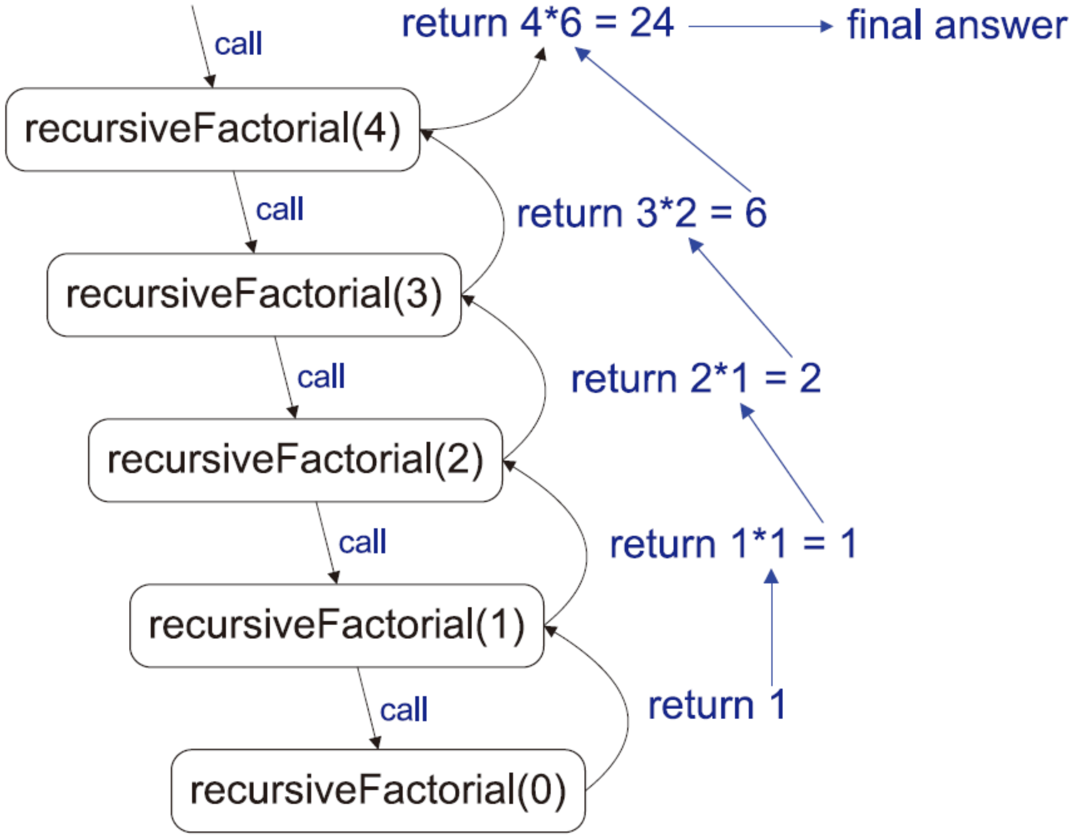
Recursive case

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{factorial}(n-1) & \text{if } n \geq 1. \end{cases}$$

Recursive Definition

Recursion Application: Factorial

```
int recursiveFactorial(int n) {              // recursive factorial function
  if (n == 0) return 1;                      // basis case
  else return n * recursiveFactorial(n−1);   // recursive case
}
```



Recursion Application: Drawing a Ruler

- How to draw the markings of a typical ruler?
- Recursive pattern exists
  - An interval with a central tick length $L-1$
  - A single tick of length L
  - An interval with a central tick length $L-1$

```
---- 0          ----- 0         --- 0
-               -               -
--              --              --
-               -               -
---             ---             --- 1
-               -               -
--              --              --
-               -               -
---- 1          ----            --- 2
-               -               -
--              --              --
-               -               -
---             ---             --- 3
-               -
--              --
-               -
---- 2          ----- 1

(a)             (b)             (c)
```

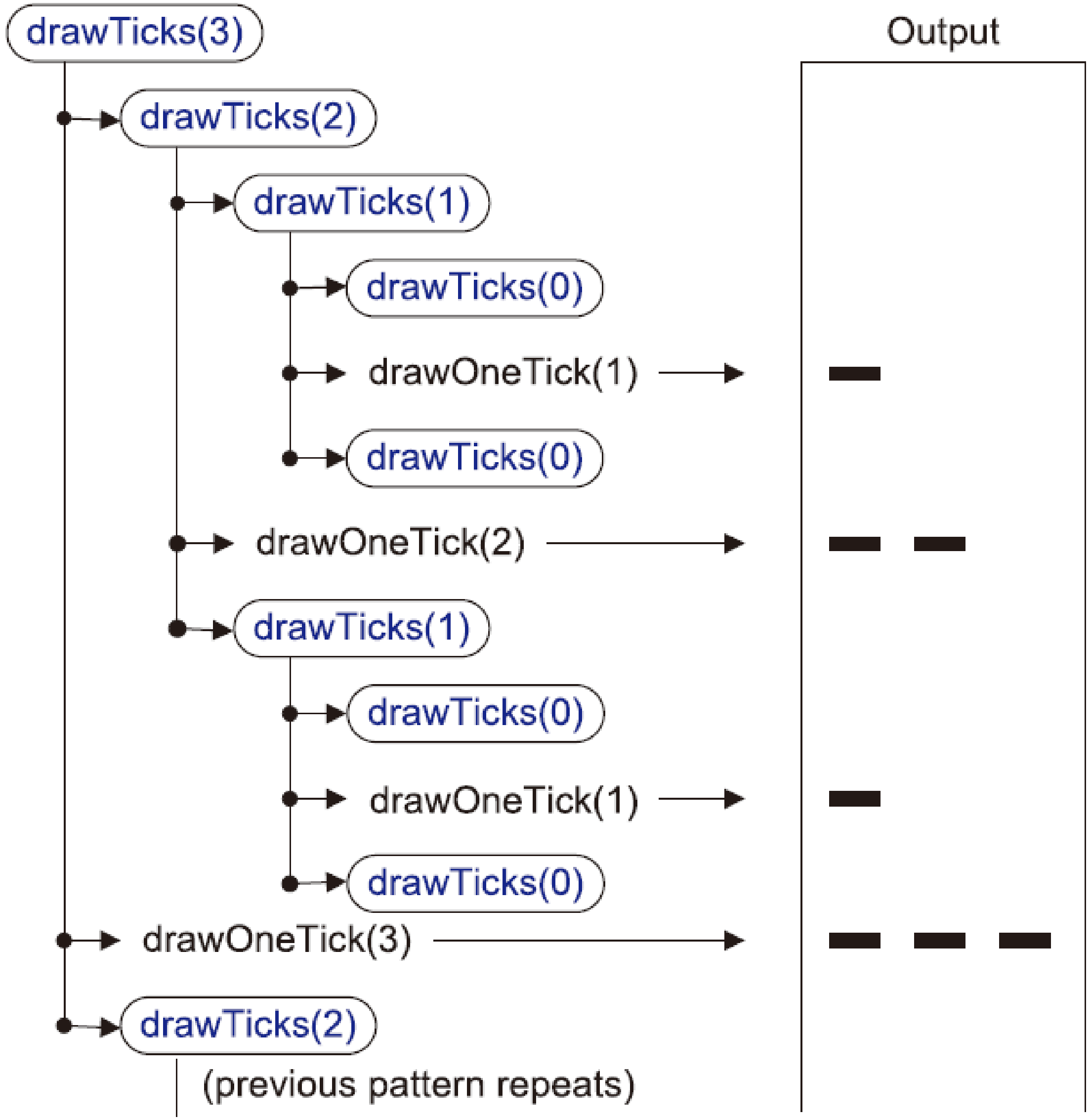
Figure 3.17: Three sample outputs of an English ruler drawing: (a) a 2-inch ruler with major tick length 4; (b) a 1-inch ruler with major tick length 5; (c) a 3-inch ruler with major tick length 3.

```
----- 0
-
--
-
---
-
--
-
----
-
--
-
---
-
--
-
----- 1
```

Ticks

Major Tick Length

Recursion Application: Drawing a Ruler



Recursion Application: Drawing a Ruler

```
                                        // one tick with optional label
void drawOneTick(int tickLength, int tickLabel = −1) {
  for (int i = 0; i < tickLength; i++)
    cout << "-";
  if (tickLabel >= 0) cout << " " << tickLabel;
  cout << "\n";
}
void drawTicks(int tickLength) {        // draw ticks of given length
  if (tickLength > 0) {                 // stop when length drops to 0
    drawTicks(tickLength−1);            // recursively draw left ticks
    drawOneTick(tickLength);            // draw center tick
    drawTicks(tickLength−1);            // recursively draw right ticks
  }
}
void drawRuler(int nInches, int majorLength) {// draw the entire ruler
  drawOneTick(majorLength, 0);          // draw tick 0 and its label
  for (int i = 1; i <= nInches; i++) {
    drawTicks(majorLength−1);           // draw ticks for this inch
    drawOneTick(majorLength, i);        // draw tick i and its label
  }
}
```

Linear Recursion

- A function makes at most one recursive call each time

- **Summing the Elements of an Array Recursively**

Algorithm $\mathsf{LinearSum}(A, n)$:

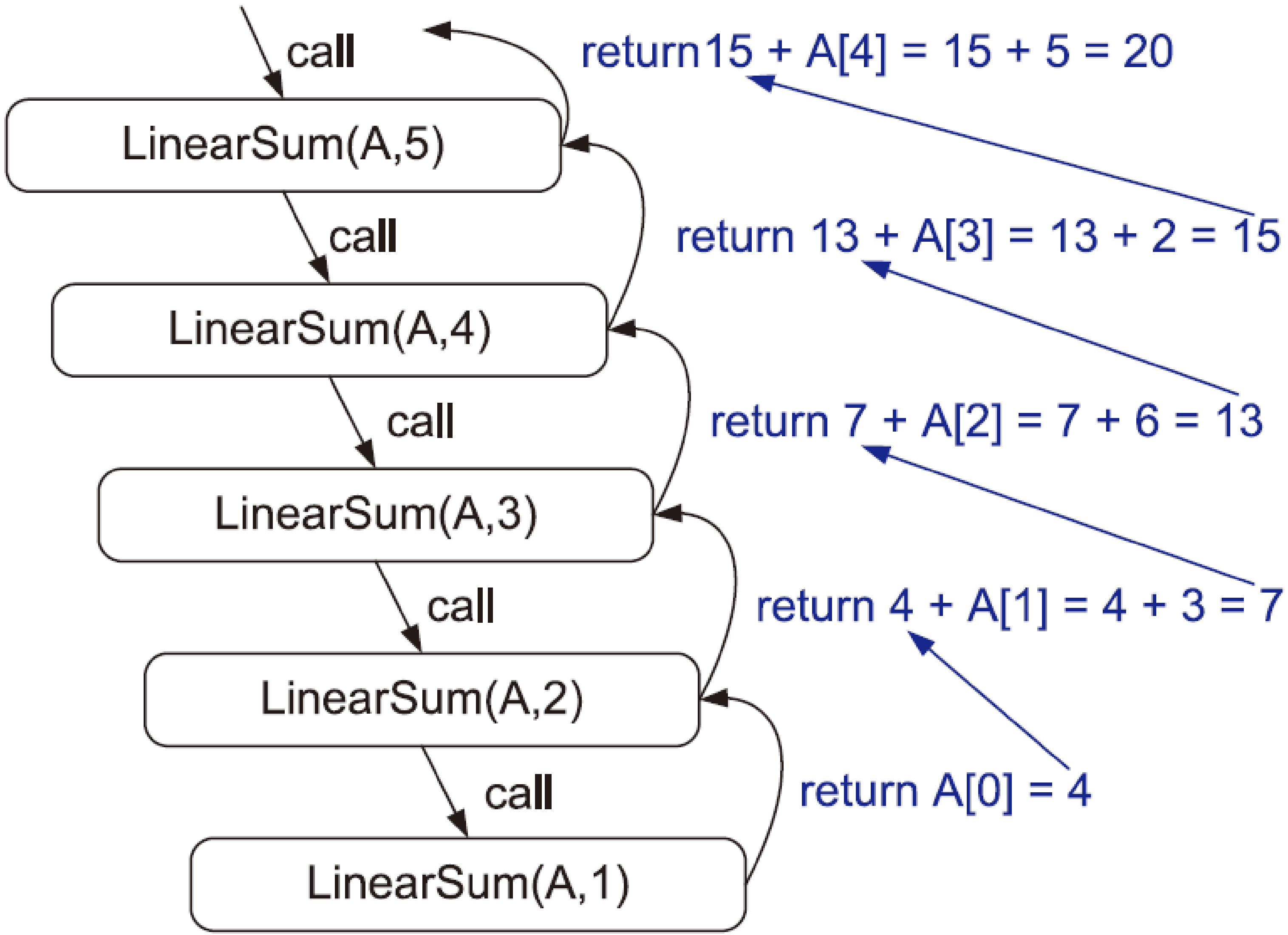
Input: A integer array A and an integer $n \geq 1$, such that A has at least n elements

Output: The sum of the first n integers in A

if $n = 1$ **then**
    return $A[0]$
else
    return $\mathsf{LinearSum}(A, n-1) + A[n-1]$

Linear Recursion: Array Summation

- **Recursion Trace**
  - *LinearSum* makes *n* calls
  - Computation time is (roughly) proportional to *n*
  - Memory usage is (roughly) proportional to *n*



Linear Recursion: Array Reversal

- **Reversing an Array by Recursion**
  - The first element becomes the last, the second one becomes the penultimate, …

Algorithm ReverseArray(A, i, j):

    Input: An array A and nonnegative integer indices i and j

    Output: The reversal of the elements in A starting at index i and ending at j

```
if i < j then
    Swap A[i] and A[j]
    ReverseArray(A, i+1, j−1)
return
```

Linear Recursion: Array Reversal

- **Case 1. The Number of Elements is Even**

T	C	U	R	T	S	A	T	A	D

Linear Recursion: Array Reversal

- **Case 2. The Number of Elements is Odd**

A	L	A	S	R	E	V	E	R

Tail Recursion

- **Recursive vs. Nonrecursive Implementations of the Array Reversal**
  - Can convert a recursive algorithm into a nonrecursive one explicitly

Algorithm ReverseArray(A, i, j):

Input: An array A and nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

if $i < j$ **then**
  Swap $A[i]$ and $A[j]$
  ReverseArray$(A, i+1, j-1)$
return

Algorithm IterativeReverseArray(A, i, j):

Input: An array A and nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

while $i < j$ **do**
  Swap $A[i]$ and $A[j]$
  $i \leftarrow i+1$
  $j \leftarrow j-1$
return

No need to store activation records in the stack

Tail Recursion

- Recursion is simple to implement
- Memory cost can be increased drastically

- **Tail Call Optimization**
  - Linear recursion
  - The very last operation must be a recursive
  - Supported by some languages

Language support [edit]

- Clojure - Clojure has `recur` special form.[22]
- Common Lisp - Some implementations perform tail-call optimization during compilation if optimizing for speed
- Elixir - Elixir implements tail-call optimization[23] As do all languages currently targeting the BEAM VM.
- Elm - Yes[24]
- Erlang - Yes
- F#- F# implements TCO by default where possible [25]
- Go - No support[26]
- Haskell - Yes[27]
- JavaScript - ECMAScript 6.0 compliant engines should have tail calls[28] which is now implemented on Safari/WebKit[29] but rejected l
- Kotlin - Has `tailrec` modifier for functions[30]
- Lua - Tail recursion is required by the language definition[31]
- Objective-C - Compiler optimizes tail calls when -O1 (or higher) option specified but it is easily disturbed by calls added by Automat
- OCaml - Yes
- Perl - Explicit with a variant of the "goto" statement that takes a function name: `goto &NAME;` [32]
- PureScript - Yes
- Python - Stock Python implementations do not perform tail-call optimization, though a third-party module is available to do this.[33]
- Racket - Yes[35]
- Rust - tail-call optimization may be done in limited circumstances, but is not guaranteed[36]
- Scala - Tail-recursive functions are automatically optimized by the compiler. Such functions can also optionally be marked with a `@ta`
- Scheme - Required by the language definition[38][39]
- Tcl - Since Tcl 8.6, Tcl has a tailcall command[40]

https://en.wikipedia.org/wiki/Tail_call

Algorithm $\mathsf{LinearSum}(A,n)$:

Input: A integer array A and an integer $n \geq 1$, such that A has at least n elements

Output: The sum of the first n integers in A

if $n = 1$ **then**
  return $A[0]$
else
  return $\mathsf{LinearSum}(A, n-1) + A[n-1]$

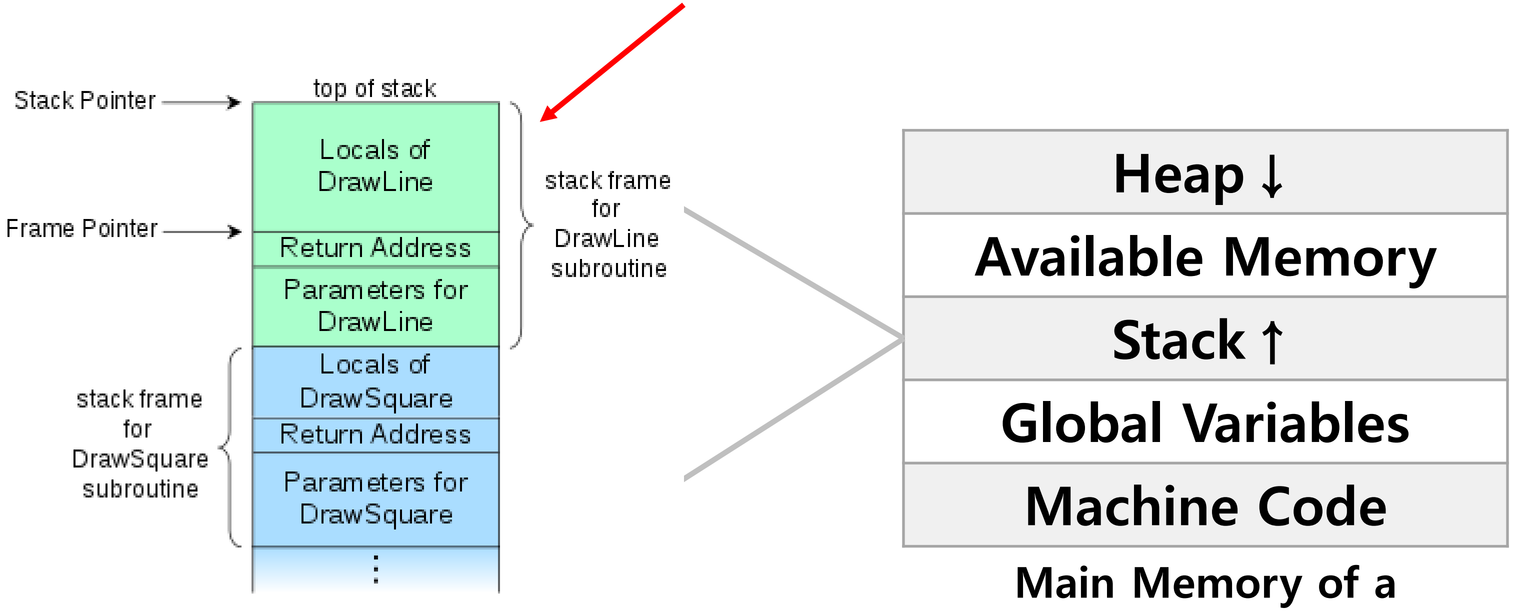
The last operation is the addition not the recursion

Appendix: Call Stack

- A stack that stores information about the active subroutines of a program
- Composed of stack frames (*i.e.*, activation records) containing subroutine state information

```
void DrawSquare(){
    // Do something
    DrawLine();
    // Do something and return
}
```

Return value, value parameter, return address, local variables



Call Stack Layout

Main Memory of a Program

Binary Recursion

- A function makes two recursive calls

- **Array Summation revisited**
  - i) Recursively sum the elements in the first half of an array
  - ii) Do the same for the second half of an array
  - iii) Add two values together

Algorithm BinarySum(A, i, n):

Input: An array A and integers i and n

Output: The sum of the n integers in A starting at index i

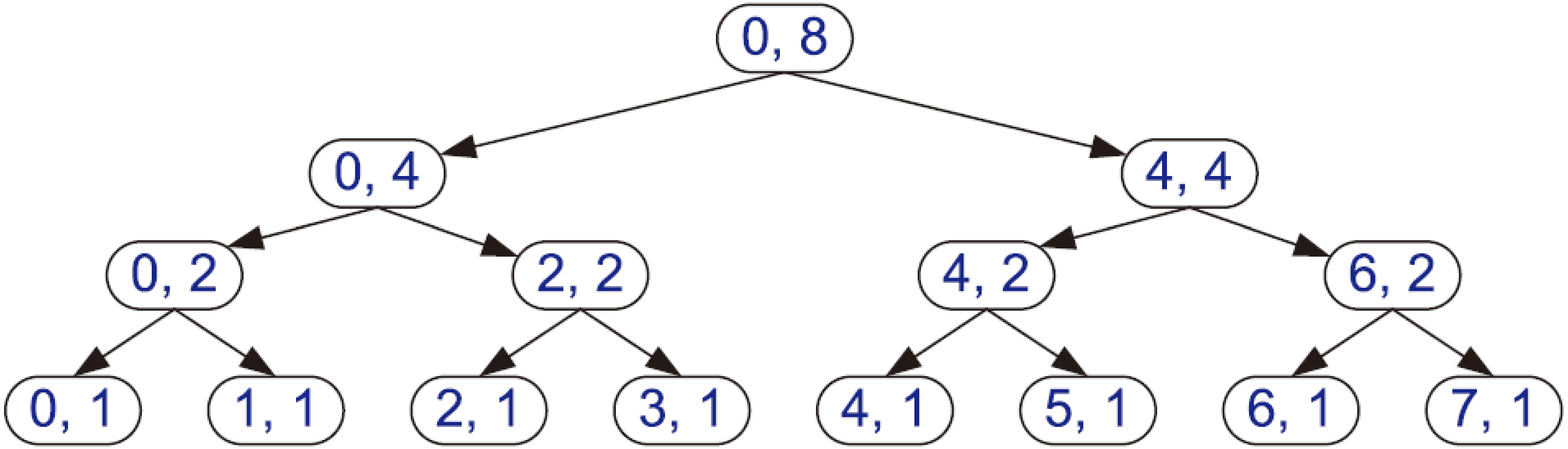
if $n = 1$ **then**

  return $A[i]$

return BinarySum$(A, i, \lceil n/2 \rceil)$ + BinarySum$(A, i + \lceil n/2 \rceil, \lfloor n/2 \rfloor)$

Binary Recursion: Array Summation

- The depth of the recursion is $1 + \log_2 n$
- Uses $O(\log n)$ additional space
  - LinearSum uses $O(n)$ additional space
- The running time is $O(n)$ because there are $2n - 1$ method calls, each requiring constant time



Recursion Trace for the Execution of *BinarySum(0,8)*

Binary Recursion: Fibonacci Numbers

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_i &= F_{i-1} + F_{i-2} \quad \text{for } i > 1
\end{aligned}
$$

Definition of Fibonacci Numbers

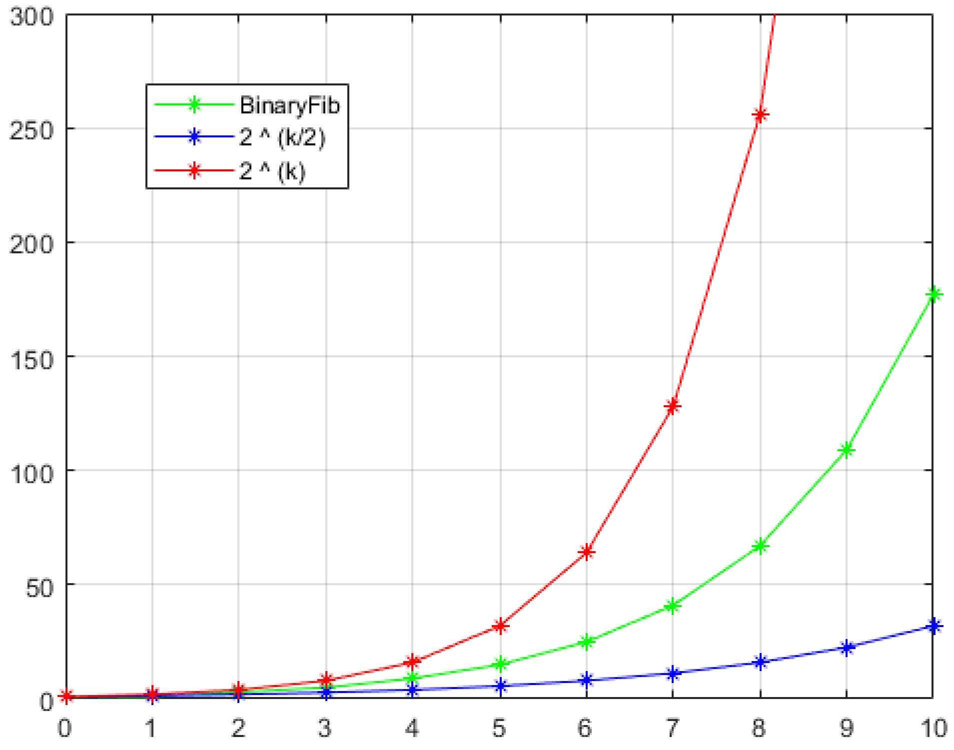
Algorithm BinaryFib(k):
    Input: Nonnegative integer k
    Output: The kth Fibonacci number F_k
    if $k \leq 1$ **then**
        return k
    else
        return BinaryFib($k-1$) + BinaryFib($k-2$)

Binary Recursion: Fibonacci Numbers

- **Number of function calls**

n_k : The number of calls performed by *BinaryFib(k)*

$$
\begin{aligned}
n_0 &= 1 \\
n_1 &= 1 \\
n_2 &= n_1 + n_0 + 1 = 1 + 1 + 1 = 3 \\
n_3 &= n_2 + n_1 + 1 = 3 + 1 + 1 = 5 \\
n_4 &= n_3 + n_2 + 1 = 5 + 3 + 1 = 9 \\
n_5 &= n_4 + n_3 + 1 = 9 + 5 + 1 = 15 \\
n_6 &= n_5 + n_4 + 1 = 15 + 9 + 1 = 25 \\
n_7 &= n_6 + n_5 + 1 = 25 + 15 + 1 = 41 \\
n_8 &= n_7 + n_6 + 1 = 41 + 25 + 1 = 67
\end{aligned}
$$



$$\boldsymbol{BinaryFib(k) = O(2^k)}$$

Binary Recursion: Fibonacci Numbers

- **Fibonacci: Linearly Recursive Problem**
  - Computes a pair of consecutive Fibonacci numbers

Algorithm LinearFibonacci(k):

Input: A nonnegative integer k

Output: Pair of Fibonacci numbers (F_k, F_{k-1})

if $k \leq 1$ **then**

  return $(k, 0)$

else

  $(i, j) \leftarrow$ LinearFibonacci($k-1$)

  return $(i + j, i)$

k	n_k
0	1
1	1
2	1+1 = 2
3	2 + 1 = 3
4	3 + 1 = 4
5	4 + 1 = 5
6	5 + 1 = 6
7	6 + 1 = 7

$$LinearFibonacci(k) = O(k)$$

Multiple Recursion

- A function may make multiple recursive calls

- **Summation Puzzles**
  - Each alphabet corresponds to an integer
  - Enumerate all the possible combinations

$$
\begin{array}{cccccc}
 & & S & E & N & D \\
+ & & M & O & R & E \\
\hline
= & M & O & N & E & Y
\end{array}
$$

O = 0, M = 1, Y = 2, E = 5, N = 6, D = 7, R = 8, and S = 9.

Multiple Recursion: Summation Puzzles

Algorithm PuzzleSolve(k, S, U):
  Input: An integer k, sequence S, and set U
  Output: An enumeration of all k-length extensions to S using elements in U without repetitions

```
for each e in U do
    Remove e from U {e is now being used}
    Add e to the end of S
    if k = 1 then
        Test whether S is a configuration that solves the puzzle
        if S solves the puzzle then
            return "Solution found: " S
    else
        PuzzleSolve(k − 1, S, U)
    Add e back to U {e is now unused}
    Remove e from the end of S
```

$S = \{a, b, c, \ldots\} = \{1, 2, 3, \ldots\}$

Each position corresponds to an alphabet

$U = \{5, 6, 7, \ldots\}$

A set of unused numbers

Multiple Recursion: Summation Puzzles