Unit 03: Recursion

Anthony Estey

CSC 225: Algorithms and Data Structures I

University of Victoria

*Thanks to Dr. Rich Little at the University of Victoria for providing some of the contents of these slides

Unit 03 Overview

- Supplemental Reading:
  - Algorithm Design and Analysis. *Michael Goodrich and Roberto Tamassia*
    - Pages 10, 19-25
- Learning Objectives: (You should be able to…)
  - understand why we will use pseudocode to support or analysis of algorithms and data structures
  - understand the syntax of pseudocode, and how it maps to operations in a programming language like Java, C, or C++
  - understand the methodology we will use in this course to analyze algorithms and data structures, based on the size of the input data, *n*
  - determine the number of operations required to execute an algorithm through an analysis of pseudocode

What is recursion?

- A divide-and-conquer approach to solving problems, where the solution depends on solutions to smaller instances of the *same* problem

- A recursive solution to a problem by using functions/methods that call themselves within their own code

- Recursion is one of the central ideas of computer science; we will see a lot of recursion solutions as we explore different algorithms and data structures throughout this course

Example

factorial(5)

return $5 \ * $ factorial(4)

return $4 \ * $ factorial(3)

return $3 \ * $ factorial(2)

return $2 \ * $ factorial(1)

return 1

Algorithm factorial(n):

Input: An integer $n \geq 0$.

Output: $n!$.

if $n \leq 1$ **then**

return 1

return $n *$ factorial$(n-1)$

Example

Algorithm factorial(n):

Input: An integer $n \geq 0$.

Output: $n!$.

if $n \leq 1$ **then**

return 1

return $n *$ factorial$(n-1)$

factorial(5)

return 5 $* \boxed{24}$

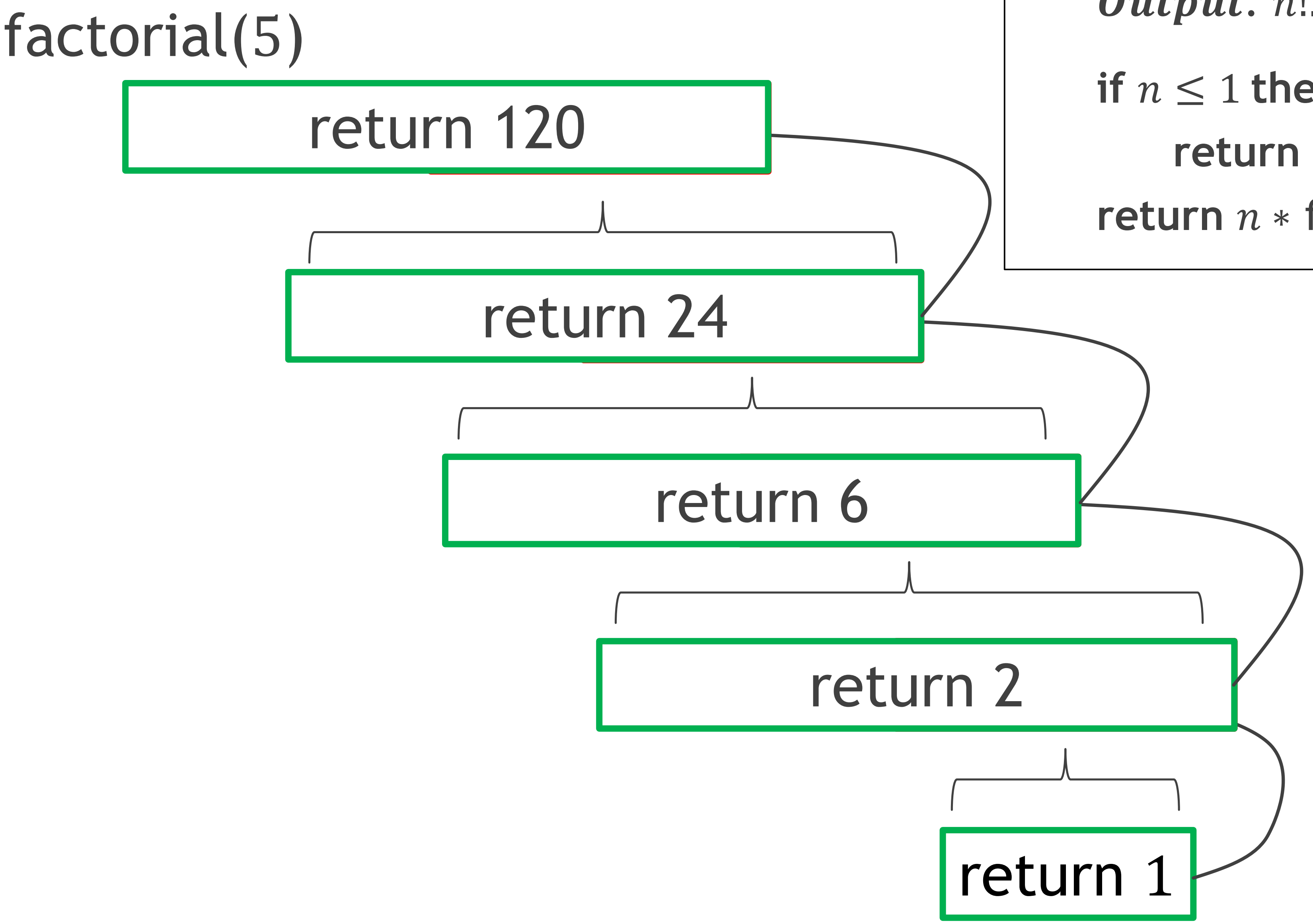
return 4 $* \boxed{6}$

return 3 $* \boxed{2}$

return 2 $* \boxed{1}$

$\boxed{\text{return } 1}$

Example



Algorithm factorial(n):

> ***Input***: An integer $n \geq 0$.
>
> ***Output***: $n!$.
>
> **if** $n \leq 1$ **then**
>
>   **return** 1
>
> **return** $n * \text{factorial}(n-1)$

Rules of Recursion

Algorithm factorial(n):

Input: An integer $n \geq 0$.

Output: $n!$.

if $n \leq 1$ **then**

  return 1

return $n * \boxed{\text{factorial}(n-1)}$

- Three important properties of a recursive algorithm:

1. The recursive algorithm must have a base case
2. The recursive algorithm must call itself (called a *recursive call*)
3. The recursive calls must converge to the base case

Runtime analysis (recursion)

- To determine the runtime of a recursive solution, we need to first determine the recurrence equation
  - We do this by counting the number of operations
  - But now there are multiple cases to account for: the base case, and the inductive step

Structure of a recursive algorithm

Algorithm recursiveAlgorithm(n):

    $\boldsymbol{Input}$: An integer $n \geq 0$.

    $\boldsymbol{Output}$: A solution to the problem

    if $n = 1$ **then**

        $base\ case$

    else

        $inductive\ step$

        recursiveAlgorithm($n - 1$)

    end

- Then: $\mathrm{T}(n) = \begin{cases} c_1, & if\ n = 1 \\ T(n-1) + c_2, & \text{otherwise} \end{cases}$

Recall: Runtime analysis of arrayMax

Algorithm arrayMax(A, n)

$\boldsymbol{Input}$: An array A storing $n \geq 1$ integers

$\boldsymbol{Output}$: The maximum element in A

$currentMax \leftarrow A[0]$

for $\mathrm{k} \leftarrow 1$ to $\mathrm{n} - 1$ **do**

    if $currentMax < A[k]$ **then**

        $currentMax \leftarrow A[k]$

    end

end

return $currentMax$

$$\boxed{T(n) = 7n - 2}$$

Recursive arrayMax example

Algorithm recursiveMax(A, n)

$\boldsymbol{Input}$: An array A storing $n \geq 1$ integers

$\boldsymbol{Output}$: The maximum element in A

if $n = 1$ **then**

  return $A[0]$

return $\max\{$recursiveMax$(A, n-1), \mathrm{A}[\mathrm{n}-1]\}$

Base case: 3 operations
($n = 1, A[0]$, return)

Induction step: $T(n-1) + 7$ operations
$(n = 1, n-1, n-1, \text{call}, \max, \text{return})$

$$\mathrm{T}(n) = \begin{cases} c_1, & if\ n = 1 \\ T(n-1) + c_2, & \text{otherwise} \end{cases}$$

$$\boxed{\mathrm{T}(n) = \begin{cases} 3, & if\ n = 1 \\ T(n-1) + 7, & \text{otherwise} \end{cases}}$$

Runtime analysis (recursion)

- To determine the runtime of a recursive solution, we need to first determine the recurrence equation
  - We do this by counting the number of operations
  - But now there are multiple cases to account for: the base case, and the inductive step

- We then need to solve the recurrence equation, by expressing the equation in **closed form**
  - **closed form**: no references to the function T appear on the righthand side

Solving recurrence equations (recursiveMax example)

Starting with $T(n) = \begin{cases} 3, & if\ n = 1 \\ T(n-1) + 7, & \text{otherwise} \end{cases}$

- Solving by repeated substitution:

$$T(n) = T(n-1) + 7$$
$$T(n-1) = T(n-2) + 7$$
$$T(n-2) = T(n-3) + 7$$
$$\ldots$$
$$T(2) = T(1) + 7$$
$$T(1) = 3$$

$$T(n) = T(n-1) + 7$$
$$T(n) = (T(n-2) + 7) + 7 = T(n-2) + 2(7)$$
$$T(n) = (T(n-3) + 7) + 2(7) = T(n-3) + 3(7)$$
$$\ldots$$
$$T(n) = T(n-i) + 7i$$
$$\ldots$$
$$T(n) = T\big(n - (n-1)\big) + 7(n-1)$$
$$T(n) = T(1) + 7n - 7$$
$$T(n) = 3 + 7n - 7 = 7n - 4$$