Algorithm Analysis and Data Structures

CSCI 7432 - Fall 2022

Algorithm Foundations

Dr. Yao XU

Assistant Professor

Department of Computer Science

Georgia Southern University

Email: yxu@georgiasouthern.edu

Table of Contents

Algorithms

What is an Algorithm?

- ***Computational Problem***: Given an input X satisfying..., output Y satisfying...
- ***Algorithm***: A well-defined step-by-step procedure that transforms the input of a problem into the output.
- ***Instance***: A specific input for a problem is called an instance of the problem.
- An algorithm is said to be ***correct*** if, for every input instance, it halts with the correct output.
- We say that a correct algorithm ***solves*** the given computational problem.

Problem Examples:

1. **Input:** A non-negative integer X **Output:** $Y = X!$
2. **Input:** A sequence of n numbers $X = \langle a_1, a_2, \cdots, a_n \rangle$
   Output: A permutation $Y = \langle a'_1, a'_2, \cdots, a'_n \rangle$ of X s.t. $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Argue About An Algorithm

What do we need to argue about an algorithm?

1. Provide an accurate description - **pseudocode**
2. Correctness
3. Amount of resources (**time** and **space**) - analysis
   - For *any* instance?
   - For a *good* instance?
   - For an *average* instance?
4. Can we do better?

Describe an Algorithm in Pseudocode

- *Pseudocode* is designed for expressing algorithms to humans.
- **Name** and a clear indication of the **input** is a must.
- **Short description**, **input/output** or **pre/post-conditions** are optional but strongly encouraged to include.
- **Pseudocode conventions**: *textbook pages 20-22*
- Be consistent and clear!

Example:

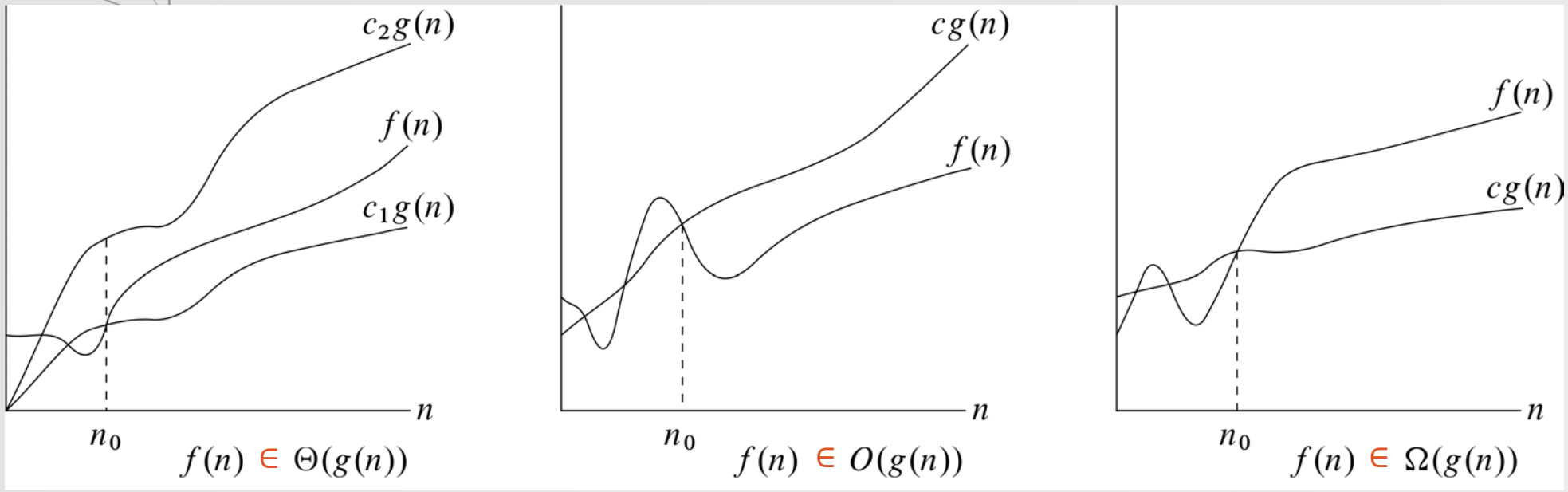
```
INSERTION-SORT(A)
// <Short description>
// Precondition: An array A[1..n] containing a
// sequence of n pair-wise comparable elements.
// Postcondition: The array A contains a sorted
// sequence of n elements.
for j = 2 to A.length // n = A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j - 1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i – 1
    A[i + 1] = key
```

Asymptotic Notations

Asymptotic Notations (1/2)

- $O(g(n))$ is the set of all functions $f(n)$ that
  - roughly, grow no faster than $g(n)$
  - formally, $\exists c > 0$ and $n_0 > 0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$
- $\Omega(g(n))$ is the set of all functions $f(n)$ that
  - roughly, grow no slower than $g(n)$
  - formally, $\exists c > 0$ and $n_0 > 0$ such that $f(n) \geq cg(n)$ for all $n \geq n_0$
- $\Theta(g(n))$ is the set of all functions $f(n)$ that
  - roughly, grow at the same rate as $g(n)$
  - formally, $\exists c_1 > 0, c_2 > 0$ and $n_0 > 0$, such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$
  - $\Theta\big(g(n)\big) = O(g(n)) \cap \Omega(g(n))$

Asymptotic Notations (2/2)



$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \begin{cases} 0 & \Rightarrow f(n) \text{ grows slower than } g(n) \Rightarrow f(n) \in \boldsymbol{o}(g(n)) \\ c > 0 & \Rightarrow f(n) \text{ grows at the same rate as } g(n) \Rightarrow f(n) \in \boldsymbol{\Theta}(g(n)) \\ \infty & \Rightarrow f(n) \text{ grows faster than } g(n) \Rightarrow f(n) \in \boldsymbol{\omega}(g(n)) \end{cases}$$

- We write: $f(n) \in O(g(n))$, instead of "$f(n) = O(g(n))$" in the textbook.

Exercises: Comparing Functions

- **Exercise 1:** Compare the orders of growth of the following two functions:

$$f(n) = \frac{1}{2}n(n-1) \quad \text{and} \quad g(n) = n^2.$$

- **Exercise 2:** Compare the orders of growth of the following two functions:

$$f(n) = \log_2 n \quad \text{and} \quad g(n) = \sqrt{n}.$$

Hint: Use **L'Hôpital's rule**: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f\prime(n)}{g\prime(n)}$.

- **Exercise 3:** Compare the orders of growth of the following two functions:

$$f(n) = n! \quad \text{and} \quad g(n) = 2^n.$$

Hint: Take advantage of **Stirling's formula**: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.

Basic Efficiency Classes

Class	Name	Comments / Algorithm Examples
1	Constant	Usually short of best-case efficiencies.
$\log n$	Logarithmic	Problem's size cut by a constant factor on each iteration.
n	Linear	Scan an array of size n.
$n \log n$	Linearithmic	Many divide-and-conquer algorithms.
n^2	Quadratic	Algorithms with two embedded loops.
n^3	Cubic	Algorithms with three embedded loops.
2^n	Exponential	Generate all subsets of an n-element set.
$n!$	Factorial	Generate all permutations of an n-element set.

- ***Notation:*** $\log n = \lg n = \log_2 n$

Solving Recurrences

The Substitution Method

The Substitution Method Examples (1/4)

Example 1:

$$T(n) = \begin{cases} 1, & \text{for } n = 1 \\ T(n-1) + n, & \text{for } n > 1 \end{cases}$$

Step 1: Guess by either backward substitution or forward substitution.

The Substitution Method Examples (2/4)

Example 1:

$$T(n) = \begin{cases} 1, & \text{for } n = 1 \\ T(n-1) + n, & \text{for } n > 1 \end{cases}$$

Guess: $T(n) = \frac{n(n+1)}{2}$

Step 2: Show that the solution is **correct**.

The Substitution Method Examples (3/4)

Example 2:

$$T(n) = \begin{cases} 0, & \text{for } n = 1 \\ 2T(n-1) + 1, & \text{for } n > 1 \end{cases}$$

Step 1: Guess by either backward substitution or forward substitution.

The Substitution Method Examples (4/4)

Example 2:

$$T(n) = \begin{cases} 0, & \text{for } n = 1 \\ 2T(n-1) + 1, & \text{for } n > 1 \end{cases}$$

Guess: $T(n) = 2^n - 1$

Step 2: Show that the solution is **correct**.

Solving Recurrences

The Recursion-Tree Method

The Recursion-Tree Method Examples (1/2)

- **Example 1:**

$$T(n) = \begin{cases} 1, & \text{for } n = 1 \\ 2T(n/2) + n, & \text{for } n > 1 \end{cases}$$

The Recursion-Tree Method Examples (2/2)

- **Example 2:**

$$T(n) = \begin{cases} 1, & \text{for } n = 1 \\ T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n, & \text{for } n > 2 \end{cases}$$

Solving Recurrences

The Master Theorem Method

The Master Theorem Method

- Used for many divide-and-conquer recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

where $a \geq 1$, $b > 1$, and $f(n) > 0$.

- **Master Theorem:** Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n)$. Then $T(n)$ has the following asymptotic bounds:
  1. If $f(n) \in O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) \in \Theta(n^{\log_b a})$.
  2. If $f(n) \in \Theta(n^{\log_b a} \log^k n)$ for some $k \geq 0$, then $T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$.
  3. If $f(n) \in \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) \in \Theta(f(n))$.

The Master Theorem Method Examples (1/3)

Master Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n)$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) \in O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) \in \Theta(n^{\log_b a})$.

Example:

$$T(n) = \begin{cases} 1, & \text{for } n = 1 \\ 5T(n/2) + n^2 \log n, & \text{for } n \geq 2 \end{cases}$$

The Master Theorem Method Examples (2/3)

Master Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n)$. Then $T(n)$ has the following asymptotic bounds:

2. If $f(n) \in \Theta(n^{\log_b a} \log^k n)$ for $k \geq 0$, then $T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$.

Example:

$$T(n) = \begin{cases} 1, & \text{for } n = 1 \\ 9T(n/3) + n^2 \log^2 n, & \text{for } n \geq 2 \end{cases}$$

The Master Theorem Method Examples (3/3)

Master Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n)$. Then $T(n)$ has the following asymptotic bounds:

3. If $f(n) \in \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) \in \Theta\big(f(n)\big)$.

Example:

$$T(n) = \begin{cases} 1, & \text{for } n = 1 \\ 3T(n/4) + n, & \text{for } n \geq 2 \end{cases}$$

Algorithm Analysis

Algorithm Analysis

- **Analyze an algorithm**: predict the resources that the algorithm requires. We will measure
  - ***Time efficiency/complexity*** – how fast an algorithm runs
  - ***Space efficiency/complexity*** – the amount of memory units required in addition to the space needed for its input and output
- When running time depends not only on an input size but also on the specifics of a particular input (instance), we discuss
  - ***Best case*** – gives a lower bound on the algorithm's running time
  - ***Worst case*** – gives an upper bound on the algorithm's running time
  - ***Average case*** – by forcing a distribution over the instances (we are making a huge assumption)

Algorithm Analysis

Non-Recursive Algorithm Analysis

INSERTION-SORT Time and Space Complexity

```
INSERTION-SORT(A)
for j = 2 to A.length // n = A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i – 1
    A[i + 1] = key
```

- **Worst-case** running time: $\Theta(n^2)$
- **Best-case** running time: $\Theta(n)$
- Space complexity: $\Theta(1)$

Algorithm Analysis

Recursive Algorithm Analysis

MERGE-SORT Time and Space Complexity

```
MERGE-SORT(A, p, r)
if p < r                                  // check for base case
    q = ⌊(p + r)/2⌋                       // divide
    MERGE-SORT(A, p, q)                   // conquer
    MERGE-SORT(A, q + 1, r)               // conquer
    MERGE(A, p, q, r)                     // combine
```

- **Worst-case/Best-case** running time:

$$T(n) = \begin{cases} c, & \text{for } n = 1 \\ 2T(n/2) + cn, & \text{for } n \geq 2 \end{cases}$$

$$\Longrightarrow T(n) \in \Theta(n \log n)$$

- Space complexity: $O(n)$

```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
let L[1 . . n1 + 1] and R[1 . . n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Prove Algorithm Correctness

Prove the Correctness of An Algorithm

- **Claim:** For any instance I (satisfying...), Algorithm-Name(I) returns...
- **Example claim:** For any array A containing a sequence of pair-wise comparable elements, INSERTION-SORT(A) correctly sorts A.
- To prove the correctness of an algorithm,
  - When **recursion** is involved, use **mathematical induction**
  - When **loop** is involved, use ***loop invariant*** (and **induction**)
- Anything that can be computed using a recursion can be computed using loops, and vice-versa.

Convert Recursion to Loops (1/2)

Example 1:

Factorial(n)

if $n = 0$

    return 1

else

    return $n \times$ Factorial(n - 1)

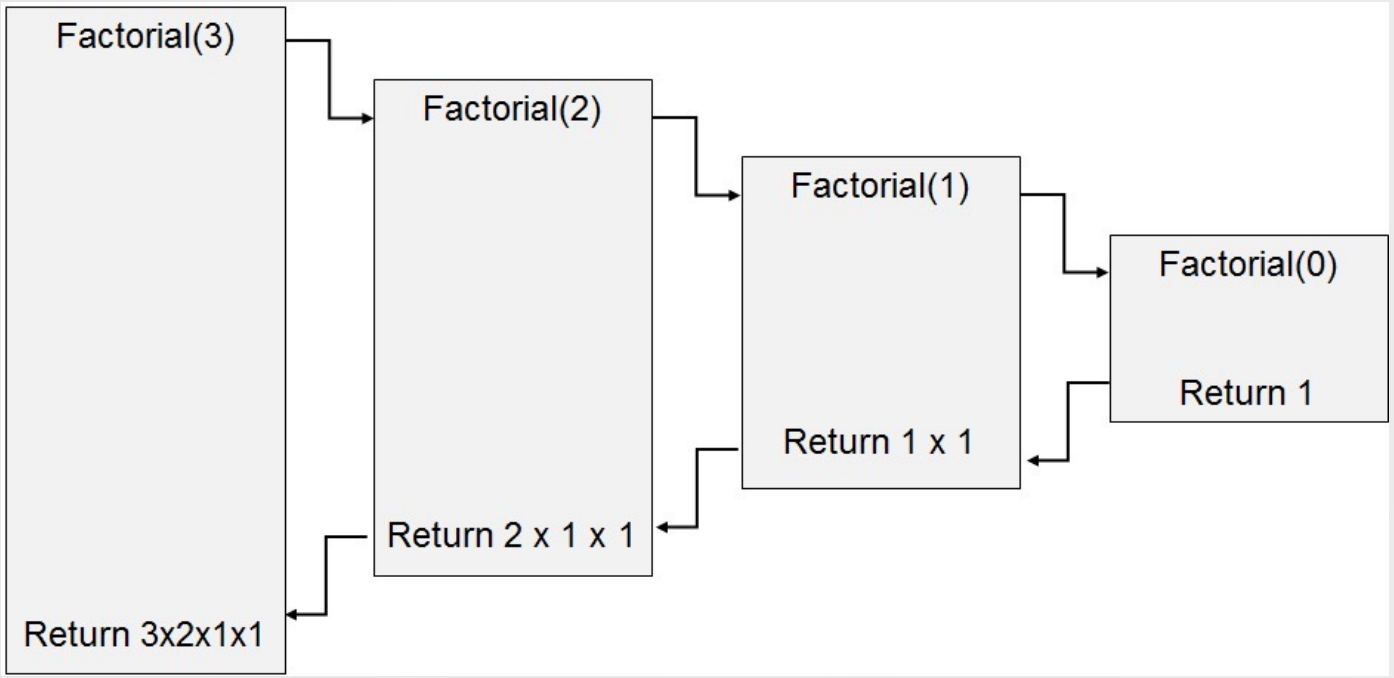
Factorial-Loop(n)

$res = 1$

for $i = 1$ **to** n

    $res = i \times res$

return res

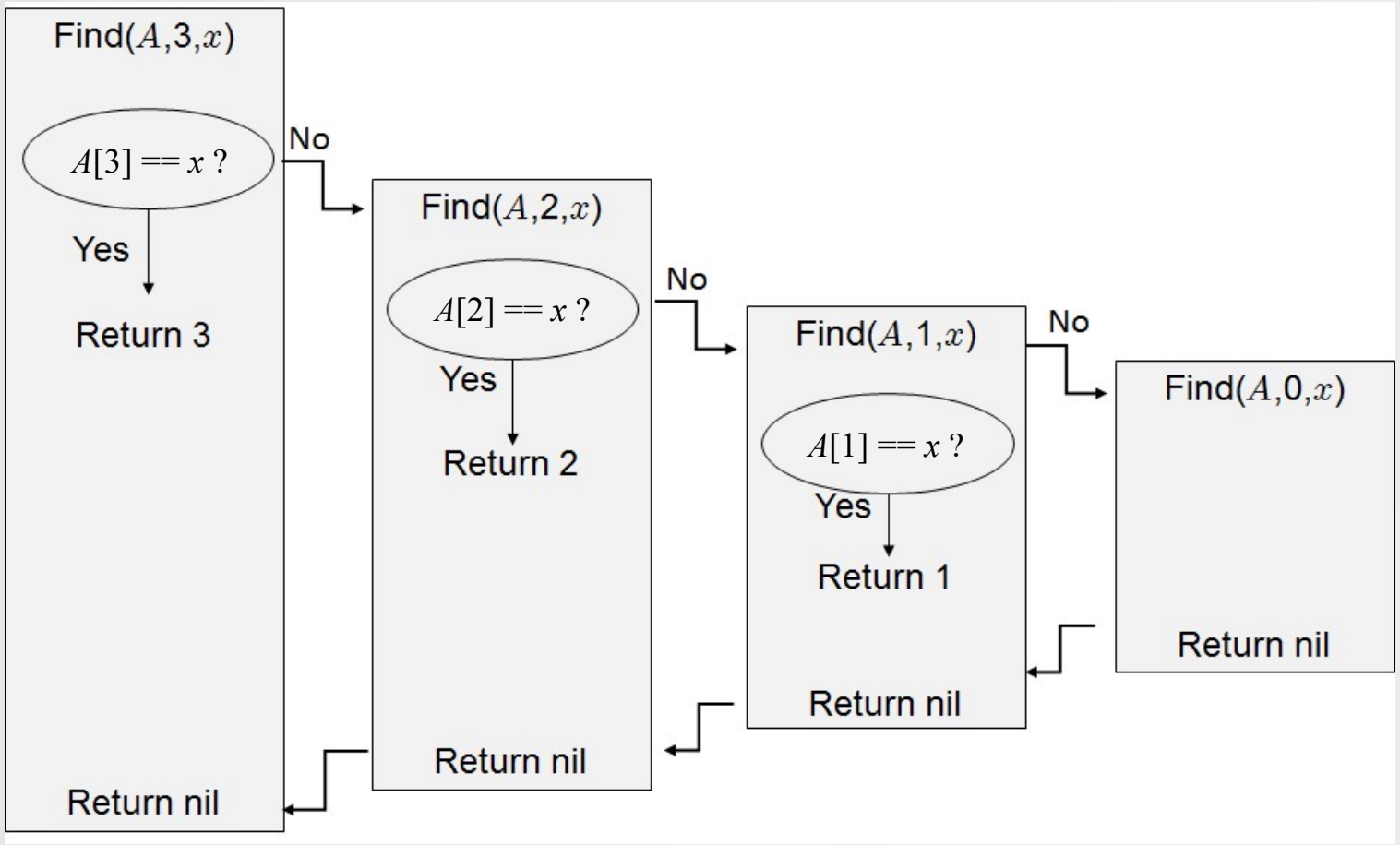


Convert Recursion to Loops (2/2)

Example 2:

```
Find(A, n, x)
if n == 0
    return NIL
else if A[n] == x
    return n
else
    return Find(A, n-1, x)

Find-Loop(A, n, x)
for i = n downto 1
    if A[i] == x
        return i
return NIL
```



Convert Loops to Recursion

Example:

Sum(A, n)

$s = A[1]$

for $i = 2$ **to** n

    $s = s + A[i]$

return s

Sum-Recursion(A, n)

if $n \leq 1$

    return $A[1]$

else

    return $A[n]$ + Sum-Recursion(A, n - 1)

Prove Correctness using Loop Invariants (1/4)

```
INSERTION-SORT(A)
for j = 2 to A.length  // n = A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j-1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i – 1
    A[i + 1] = key
```

Example: $A = \langle 5, 2, 4, 6, 1, 3 \rangle$

- A ***loop invariant (LI)*** is a statement/assertion/predicate about the state of the code that is always true at the beginning of each loop-iteration.
- What type of assertion?

  It should accurately describe the cumulative effect of repeatedly iterating through the loop.
- **Q:** What is the **LI** of the **for loop**?
- **A:** At the start of each iteration of the **for loop**, the subarray $A[1..j\text{-}1]$ consists of the elements originally in $A[1..j\text{-}1]$, but in sorted order.

Prove Correctness using Loop Invariants (2/4)

```
INSERTION-SORT(A)
for j = 2 to A.length  // n = A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j-1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i – 1
    A[i + 1] = key
```

LI of the **for loop**:

At the start of each iteration, the subarray $A[1..j\text{-}1]$ consists of the elements originally in $A[1..j\text{-}1]$, but in sorted order.

To prove correctness using **LI**, we must show:

1. **Initialization: LI** is true prior to the first iteration of the loop.
2. **Maintenance:** If **LI** is true before an iteration of the loop, it remains true before the next iteration.
3. **Termination:** When the loop terminates, **LI** gives us a useful property that helps show the correctness of the overall algorithm.

Prove Correctness using Loop Invariants (3/4)

```
INSERTION-SORT(A)
for j = 2 to A.length  // n = A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j-1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i – 1
    A[i + 1] = key
```

LI of the **for loop**:

At the start of each iteration, the subarray $A[1..j\text{-}1]$ consists of the elements originally in $A[1..j\text{-}1]$, but in sorted order.

Prove the correctness of the **for loop** (correctness of INSERTION-SORT):

1. **Initialization:** $j = 2$ and $A[1]$ is trivially sorted.
2. **Maintenance:** Show that if **LI** is true before an iteration of the loop, it remains true before the next iteration. (see next slide)
   - **Q:** What is the **while loop** doing?
3. **Termination:** The **for loop** terminates when $j = n + 1$. **LI** indicates that $A[1..n]$ is sorted.

Prove Correctness using Loop Invariants (4/4)

```
INSERTION-SORT(A)
for j = 2 to A.length  // n = A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j-1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i – 1
    A[i + 1] = key
```

LI of the **for loop**:

At the start of each iteration, the subarray $A[1..j\text{-}1]$ consists of the elements originally in $A[1..j\text{-}1]$, but in sorted order.

2. **Maintenance:** Show that if **LI** is true before an iteration of the loop, it remains true before the next iteration.
   - **Q:** What is the **while loop** doing?
- Assume **LI** is true for $j = k$. That is, subarray $A[1..k\text{-}1]$ is sorted.
- Need to prove **LI** is true for $j = k + 1$ before the next iteration. That is, subarray $A[1..k]$ is sorted.

Exercise: Prove Correctness of Merge-Sort

```
MERGE-SORT(A, p, r)
if p < r                                  // check for base case
    q = ⌊(p + r)/2⌋                       // divide
    MERGE-SORT(A, p, q)                   // conquer
    MERGE-SORT(A, q + 1, r)               // conquer
    MERGE(A, p, q, r)                     // combine
```

1. Prove the correctness of the major **for loop** of the MERGE(A, p, q, r) procedure.
   - Use **LI**. (*p.32-33 of the textbook*)
2. Prove the correctness of algorithm MERGE-SORT(A, p, r).
   - Prove by **induction**.

```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Thank you!
Questions?