INT202
Complexity of Algorithms
Introduction

XJTLU/SAT/INT
SEM2 AY2020-2021

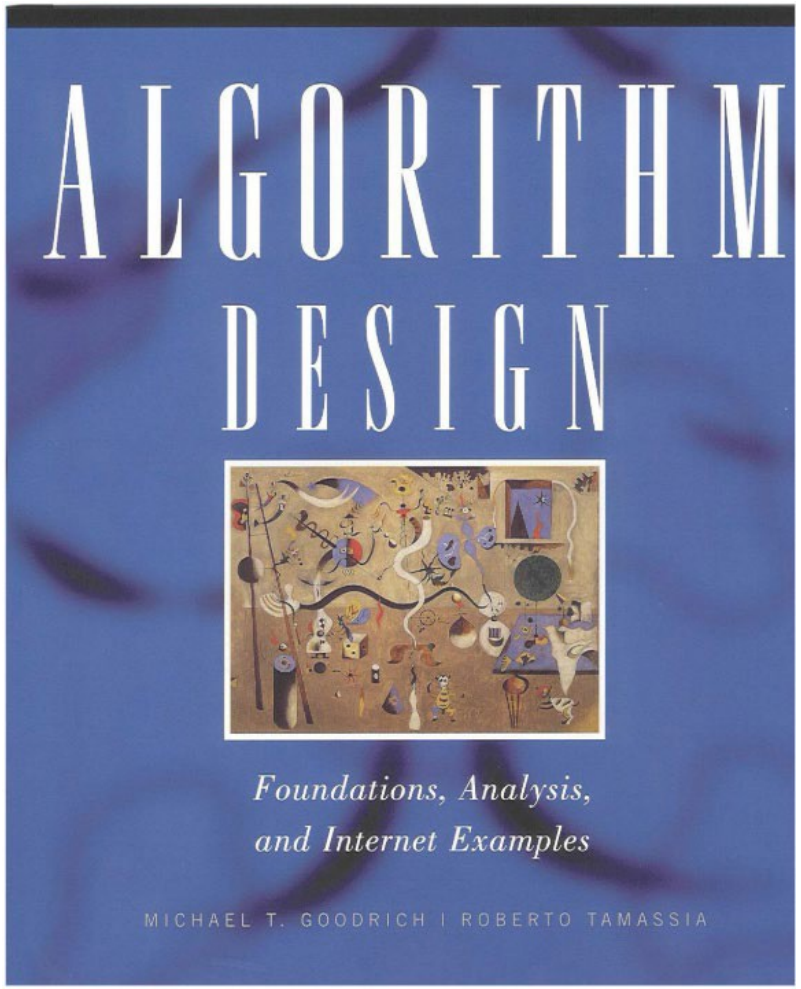
INT202: Complexity of Algorithms

- Lecturer: Rui Yang, email: R.Yang@xjtlu.edu.cn
- Lectures: Wednesday (11:00-13:00)
- Tutorials: Friday (9:00-11:00) (starts from Week 2)
- Assessment:
  - 1 In-Class Test in Week 8 (5%)
  - 1 In-Class Test in Week 14 (5%)
  - 1 Problem Solving Assignment (10%)
  - 1 Final Exam (80%)

Course Texts

- **Module Text Book:** *Algorithm Design*, Author: M.T. Goodrich and R. Tamassia (sometimes referred to as [GT] in these notes).
- **Secondary (reference):** *Introduction to Algorithms*,

  Author: T.H. Cormen, C. E. Leiserson and R.L. Rivest.



Module Outcomes

- By the end of this module a student should

  1. have an appreciation of the diversity of computational fields to which algorithmics has made significant contributions;

  2. have fluency with the structure and utilization of abstract data types such as stacks, queues, lists, and trees in conjunction with classical algorithmic problems such as searching, sorting, graph algorithms, etc;

  3. have a comfortable knowledge of graphs, their role in representing various relationships, and be familiar with several of the algorithms for solving common problems such as finding minimum spanning trees, maximum flows, shortest paths, and Euler tours;

Module Outcomes (cont.)

- By the end of this module a student should

  4. understand the basic concepts from number theory that are used in “public-key” cryptography, and be able to explain the concepts that underlie the RSA encryption/decryption scheme;

  5. be familiar with formal theories providing evidence that many important computational problems are inherently intractable, e.g., NP-completeness.

These Lectures

- Basic Concepts on Algorithm Analysis
- Introduce the complexity analysis of algorithms

- **Reading**: Chap. 1 of the Book *Algorithm Design* by M.T. Goodrich and R. Tamassia

Algorithms and Data Structures

- *A data structure* is a systematic way of organizing and accessing data.
- An *algorithm* is a sequence of steps for performing a task in a finite amount of time.

- Navigate to a place
- Transfer Audio/Video format
- Control solar panels
- Generate an image from 2D/3D model
- Root-finding algorithm
- Compression algorithm
- Optimization algorithm
- Rendering algorithm

Algorithms and Data Structures

- What makes a good algorithm?
- How do you measure efficiency?

Our fundamental interest is to design “good” *algorithms* which utilize efficient *data structures*. In this context “good” means *fast* in a way to be clarified during this course.

Algorithm Analysis

- The analysis of algorithm is the theoretical study of computer program performance and recourse usage.
- **Primary interest**: Running time (*time-complexity*) of the algorithm and the operations on data structures.
- **Secondary interest**: Space (or "memory") usage (*space-complexity*).

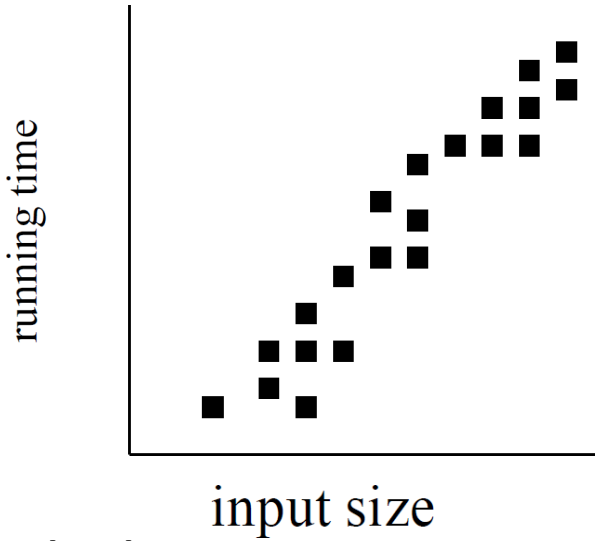
- This course focuses on the *mathematical* design and analysis of algorithms and their associated data structures, utilizing relevant mathematical tools to do so.

Experimental Analysis of Algorithms

- We are interested in *dependency* of the running time or memory requirement on *size* of the input.
- The *size* could mean the number of vertices and edges if we are operating on a graph, the length of a message we're encoding/decoding, and/or the actual length of numbers we're processing in terms of the bits needed to store them in memory.
- To analyze algorithms we sometimes perform *experiments* to empirically observe for example the running time.
- Doing so requires a good choice of sample inputs and an appropriate number of tests (so that we can have *statistical certainty* about our analysis).

Experimental Analysis (cont.)

Running-time depends on both the *size* and *instance* of input and the algorithm used, as well as the software and hardware environment on which it is run.

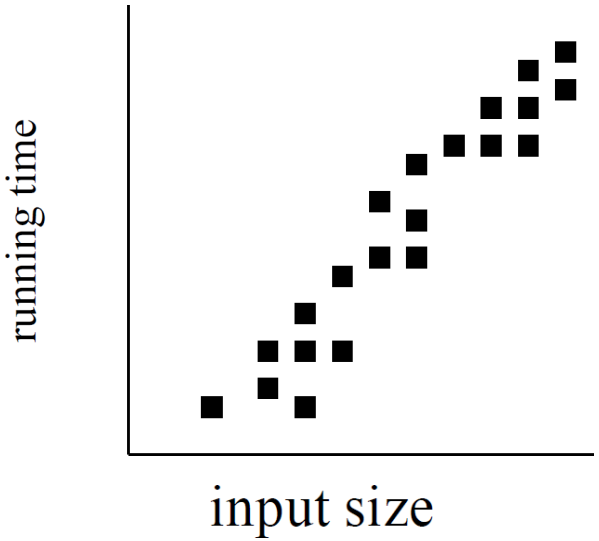


input size

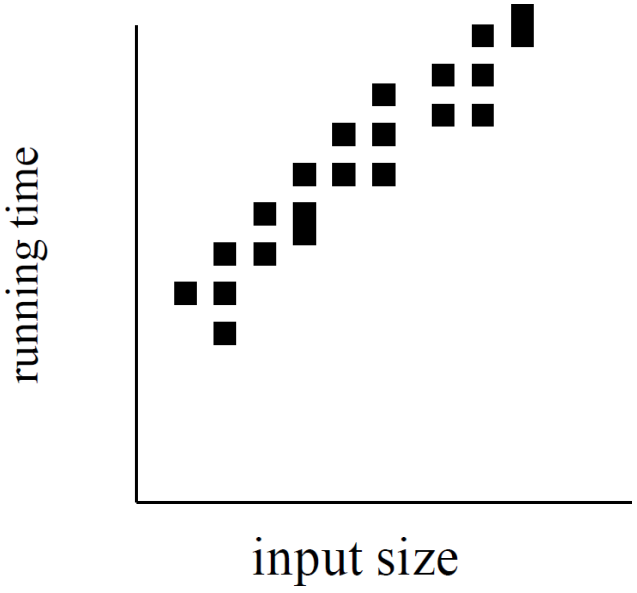
In general, the running time of an algorithm or data structure method increases with the input size

Experimental Analysis (cont.)

Running-time depends on both the *size* and *instance* of input and the algorithm used, as well as the software and hardware environment on which it is run.



a fast computer



a slow computer

Limitations of Experimental Analysis

1. Experiments are performed on a limited set of test inputs.
2. Requires all tests to be performed using same hardware and software.
3. Requires implementation and execution of algorithm.

Theoretical Analysis

Benefits over experimental analysis:

1. Can take all possible inputs into account.
2. Can compare efficiency of two (or more) algorithms, independent of hardware/software environment.
3. Involves studying high-level descriptions of algorithms (*pseudo-code*).

Theoretical Analysis (cont.)

- When we analyze an algorithm in this fashion, we aim to associate a function $f(n)$ to each algorithm, where $f(n)$ characterizes the running-time in terms of some measure of the *input-size*, n.

Typical functions include: n, $\log n$, n^2, $n \log n$, 2^n,...

'Algorithm A runs in time proportional to n'

Theoretical Analysis (requirements)

For formal theoretical analysis, we need:

1. A *language for* describing algorithms.
2. *Computational model* in which algorithms are executed.
3. *Metric* for measuring performance.
4. A way of characterizing performance.

Pseudo-code

- Pseudo-code is a high-level description language for algorithms.

Pseudo-code provides more structured description of algorithms.

Allows high-level analysis of algorithms to determine their running time (and memory requirements).

Task: Give a Pseudo-code for the function *Minimum-Element*, which finds the minimum element of a set of numbers.

Pseudo-code (cont.)

- Pseudo-code is a mixture of natural language and high-level programming language (e.g., Java, C, etc.).

Describes a generic implementation of data structures or algorithms.

- Pseudo-code includes: expressions, declarations, variable initialization and assignments, conditionals, loops, arrays, method calls, etc.

In here, we won't formally define a strict method for giving pseudo-code, but when using it we aim to describe an algorithm in a manner that would allow a competent programmer to translate the pseudo-code into program code without misinterpretation of the algorithm.

Pseudo-code (cont.)

We define a set of high-level primitive operations that are largely independent from the programming language used.

Primitive operations include the following:

- Assigning a value to a variable
- Calling a method
- Performing an arithmetic operation (for example, adding two numbers)
- Comparing two numbers
- Indexing into an array
- Following an object reference
- Returning from a method.

❖To analyze the running time of an algorithm we *count* the number of operations executed during the course of the algorithm.

Counting Primitive Operations

Algorithm arrayMax(A,n):
input: array *A*
output: maximum element *currentMax*

```
1    currentMax ← A[0]
2    for j ← 1 to n-1 do
3            if currentMax<A[j]
4                then currentMax ← A[j]
5    return currentMax
```

How many primitive operations?

Random Access Machine

This approach of simple counting primitive options gives rise to a computational model called the Random Access Machine.

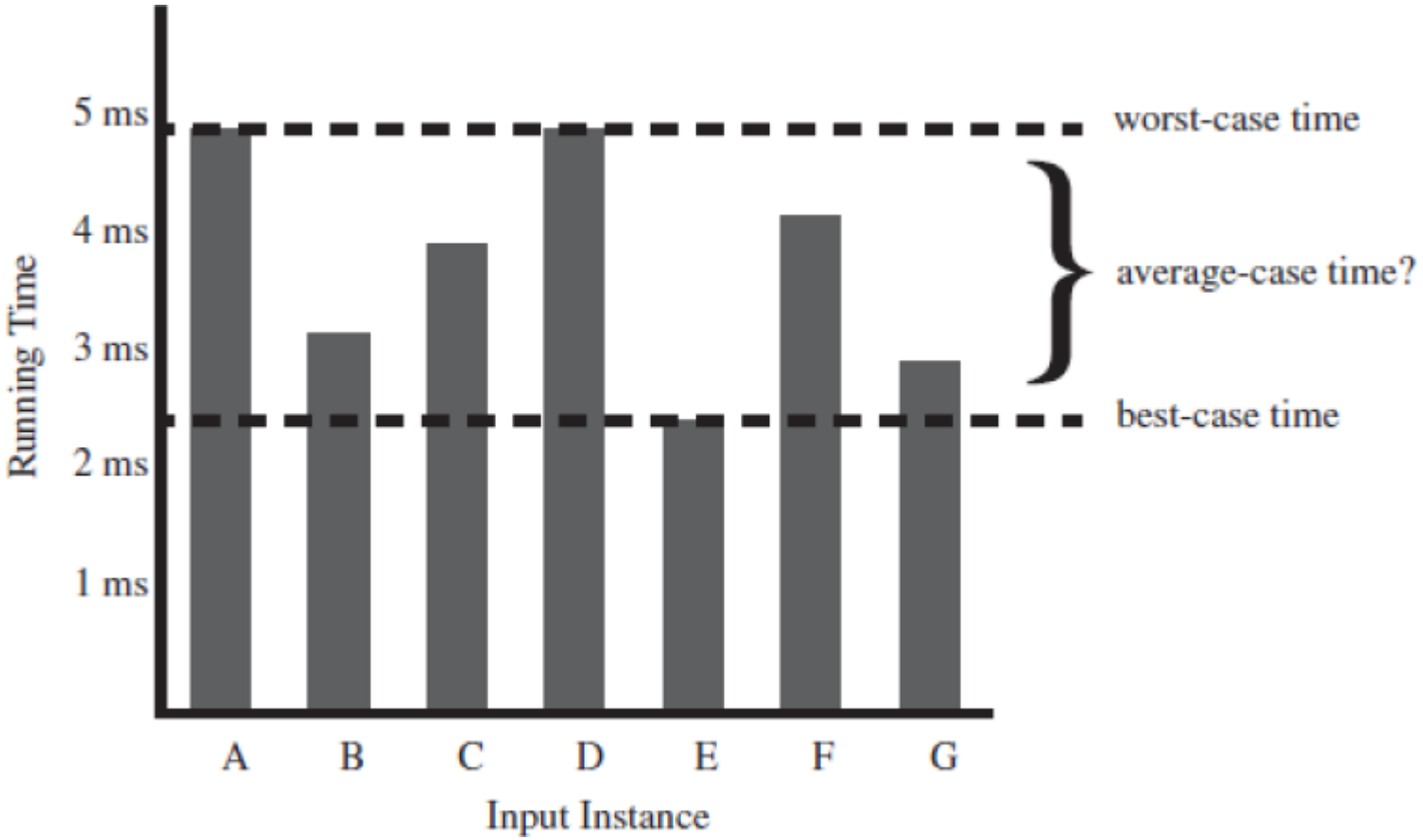
- CPU connected to a bank of *memory cells*.
- Each memory cell can store a number, character, or address.

Assumption: primitive operations (like a single addition, multiplication, or comparison) require *constant time* to perform.

(Not necessarily true, e.g. multiplication takes about four times as long to perform on a computer than addition.)

Average- vs. Worst-Case Complexity

An algorithm may run faster on some inputs compared to others.



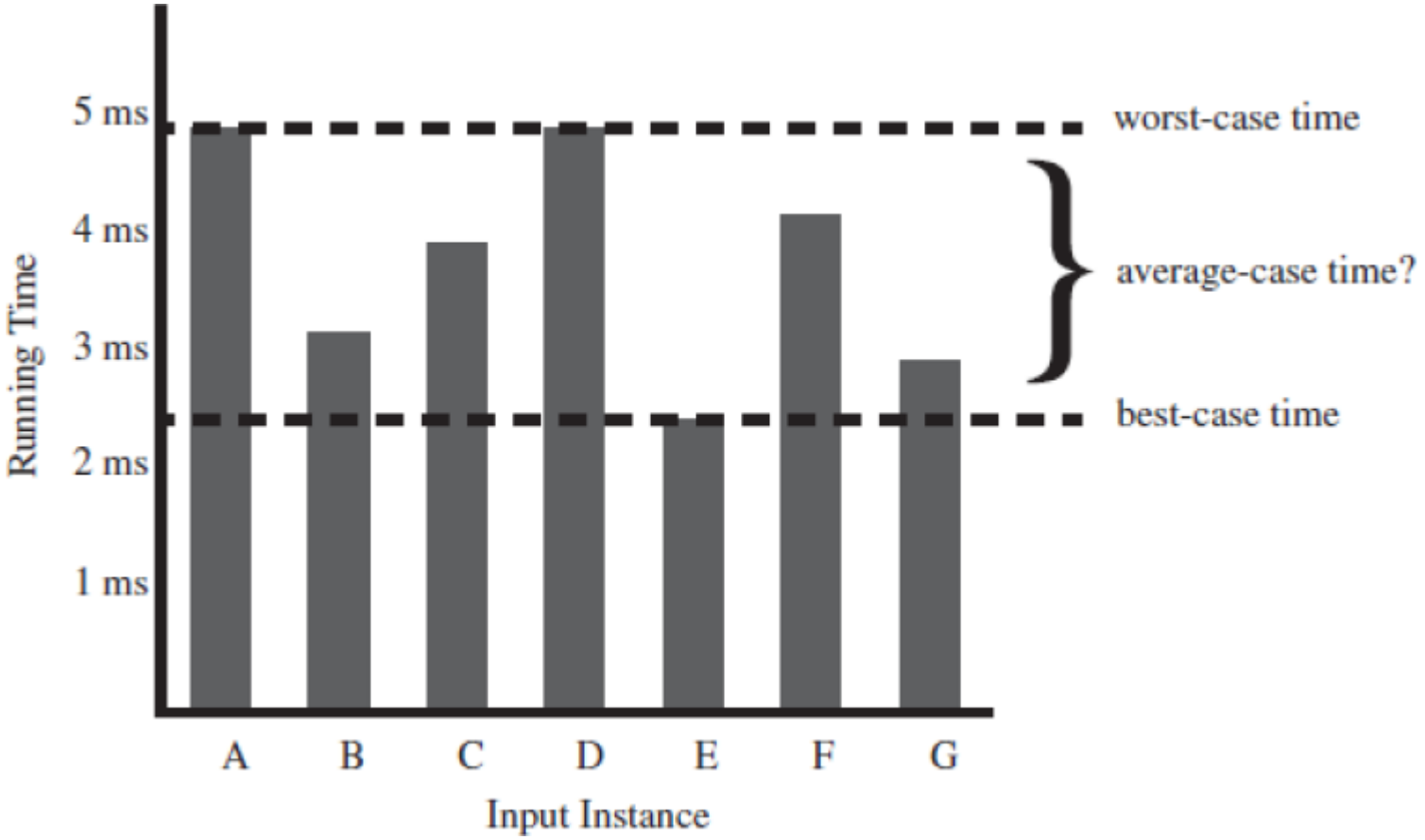
Average- vs. Worst-Case Complexity

An algorithm may run faster on some inputs compared to others.

- *Average-case* complexity refers to running time as an *average taken over all inputs* of the same size.
- *Worst-case* complexity refers to running time as the *maximum taken over all inputs* of the same size.

Usually, we're most interested in worst-case complexity.

Average- vs. Worst-Case Complexity (cont.)



An average-case analysis also typically requires that we calculate expected running times based on a given input distribution.

Counting Primitive Operations

Algorithm arrayMax(A,n):

input: array *A*

output: maximum element *currentMax*

1 *currentMax* ← *A*[0]

2 **for** *j* ← 1 **to** *n-1* ***do***

3 **if** *currentMax*<*A*[*j*]

4 **then** *currentMax* ← *A*[*j*]

5 **return** *currentMax*

How many primitive operations?

1. Line 1*:*
2. Line 2*:*
3. Line 5*:*

Counting Primitive Operations

Algorithm arrayMax(A,n):

input: array *A*

output: maximum element *currentMax*

```
1    currentMax ← A[0]
2    for j ← 1 to n-1 do
3            if currentMax<A[j]
4                then currentMax ← A[j]
5    return currentMax
```

How many primitive operations?

4. Line 3:
5. Line 4:

Counting Primitive Operations

Algorithm arrayMax(A,n):

input: array *A*

output: maximum element *currentMax*

```
1    currentMax ← A[0]
2    for j ← 1 to n-1 do
3            if currentMax<A[j]
4                then currentMax ← A[j]
5    return currentMax
```

Worst case

How many primitive operations?

4. Line 3
5. Line 4

Counting Primitive Operations

Algorithm arrayMax(A,n):

input: array *A*

output: maximum element *currentMax*

```
1    currentMax ← A[0]
2    for j ← 1 to n-1 do
3            if currentMax<A[j]
4                then currentMax ← A[j]
5    return currentMax
```

Best case ?

How many primitive operations?

4. Line 3
5. Line 4

Recursive Algorithms

- Recursion involves a procedure calling itself to solve *sub-problems* of a smaller size. These smaller subproblems can then be combined in some way to get a solution to a larger problem.
- Recursive procedures require a *base case* that can be solved directly without using recursion.

Recurrence Relations

- *Recurrence relations* sometimes allow us to define the running-time of an algorithm in the form of an equation.

Suppose that $T(n)$ denotes the running time of algorithm on input of size n. Then we might be able to characterize $T(n)$ in terms of, say, $T(n-1)$. For example, we might be able to show that

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ T(n-1) + 7 & \text{for } n \geq 2 \end{cases}$$

Ideally, given such a relationship we would then want to express this recurrence relation in a *closed form*.

In the example, we can show that

$$T(n) = 7(n-1) + 3 = 7n - 4.$$

3 10 17 24 …

Recurrence Relations (cont.)

- Recurrence relations may appear in many forms. Some examples include:

1. $C(n) = 3 \cdot C(n-1) + 2 \cdot C(n-2) + C(n-3)$ where

   $C(1) = 1, C(2) = 3, C(3) = 5$

2. The Fibonacci numbers

Recursion example: Fibonacci Numbers

The Fibonacci numbers are defined as the sequence

$f_1 = f_2 = 1$, and $f_n = f_n - 1 + f_n - 2$, for $n \geq 3$. They can be found using the following pseudo-code that computes them recursively.

Problem: Write a piece of pseudocode to compute Fibonacci numbers, n=50.

The terms of the Fibonacci sequence are: 1,1,2,3,5,8,13,21,34.

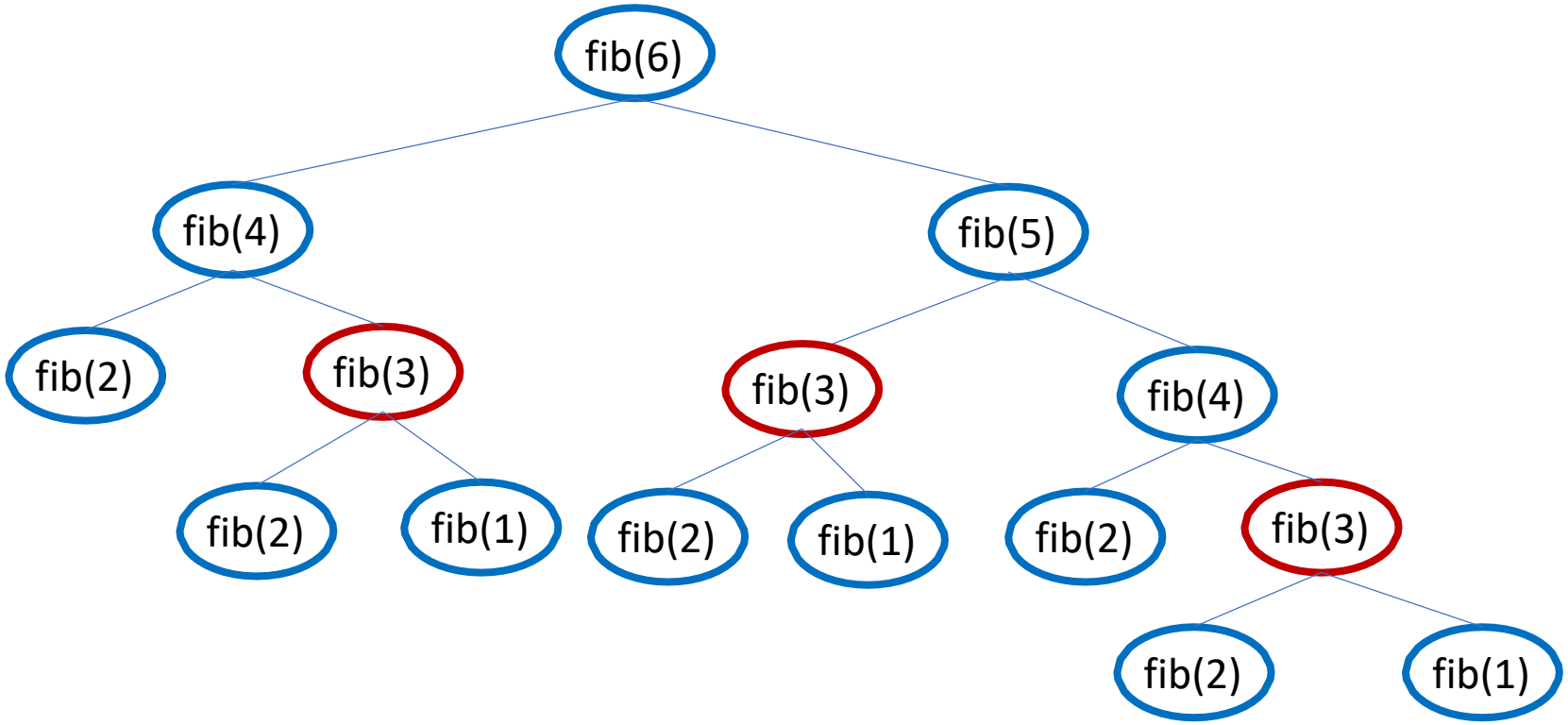
Recursive Algorithms: A Word of Caution…

While recursive algorithms are often “simpler” to write than a non-recursive version, there are often reasons to avoid them.

- In many situations, the smaller subproblems might be solved *repeatedly* during execution of the recursive algorithm.

Recursive Algorithms: A Word of Caution…

To compute Fibonacci(*n*), we must compute Fibonacci(*n* − 1) and Fibonacci(*n* − 2). **Both** of these function calls must then compute Fibonacci(*n* − 3) and Fibonacci(*n* − 4), etc. This repetition of work can massively increase the overall running time of the algorithm.



Recursive Algorithms: A Word of Caution…

To compute Fibonacci(*n*), we must compute Fibonacci(*n* − 1) and Fibonacci(*n* − 2). **Both** of these function calls must then compute Fibonacci(*n* − 3) and Fibonacci(*n* − 4), etc. This repetition of work can massively increase the overall running time of the algorithm.

- Because of the above phenomena, a recursive algorithm could also be impossible to perform on a computer (for large input values) because the repeated function calls might exhaust the memory of the machine.

Recursive Algorithms

- Recursion involves a procedure calling itself to solve *sub-problems* of a smaller size. These smaller subproblems can then be combined in some way to get a solution to a larger problem.
- Recursive procedures require a *base case* that can be solved directly without using recursion.

Exercise

- Write pseudo-code for a non-recursive method to compute *Fibonacci*(*n*), assuming that *n* is a positive integer.

Hint: Avoid the repeated computations mentioned above, by starting from the beginning of the sequence and “working up” to get the term you want.