Algorithms

Introduction to Computer

Yu-Ting Wu

(with most slides borrowed from Prof. Tian-Li Yu)

Outline

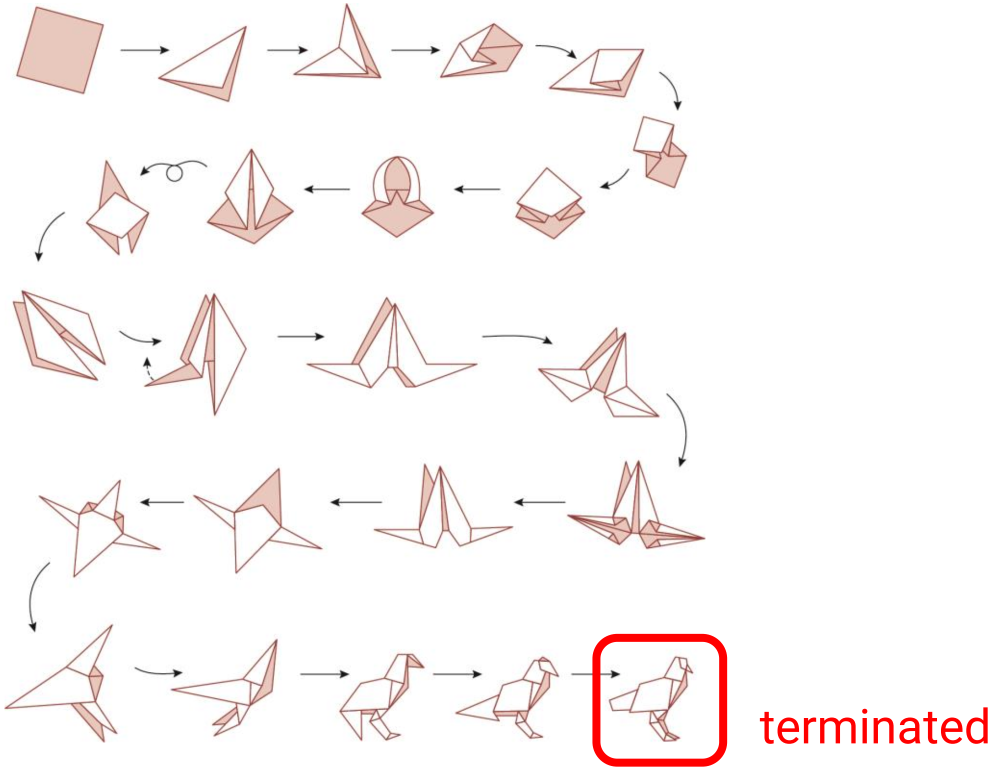
- The concept of an algorithm
- Algorithm representation
- Algorithm discovery and structures
- Efficiency and correctness

Outline

- The concept of an algorithm
- Algorithm representation
- Algorithm discovery and structures
- Efficiency and correctness

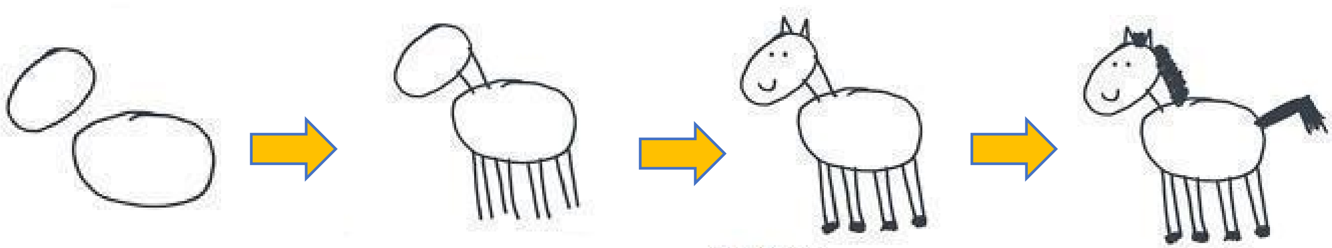
Formal Definition of Algorithm

- An algorithm is an ordered set of **unambiguous**, **executable** steps that define a **terminating** process
- Example: an algorithm for folding a bird

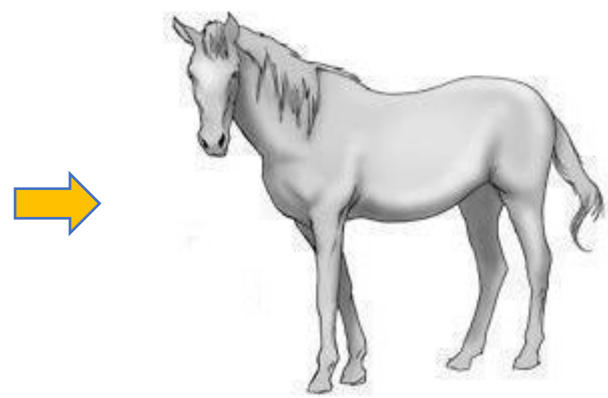


Formal Definition of Algorithm (cont.)

- How to draw a horse in five steps



draw two circles | draw legs | draw face | draw hair and tail



add few details

This is **NOT** an algorithm

Formal Definition of Algorithm (cont.)

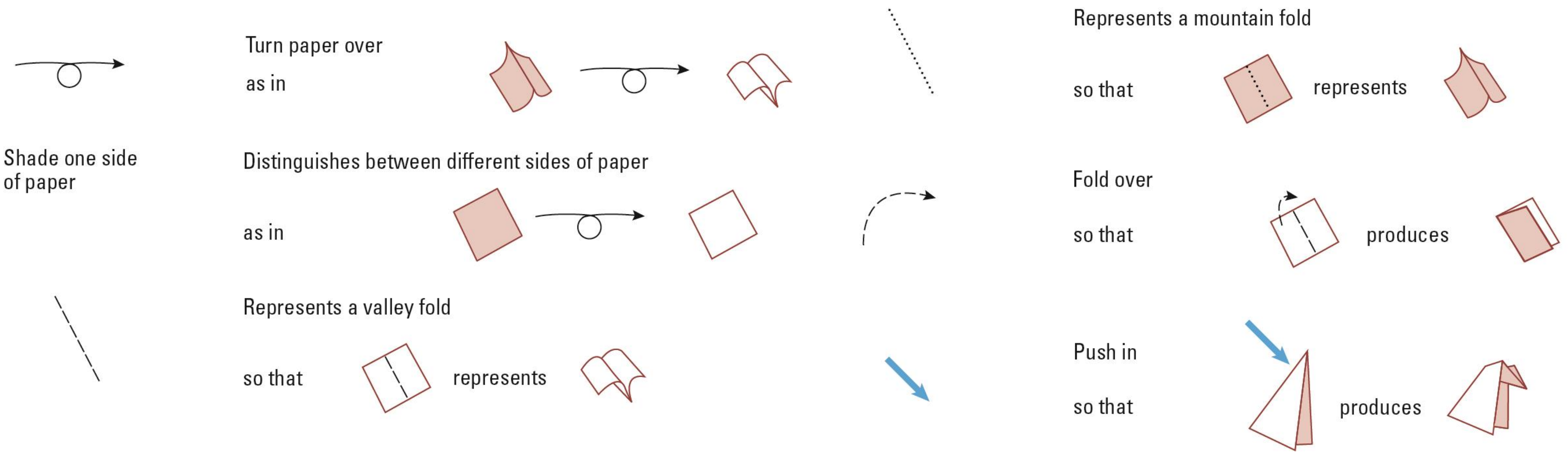
- There is a difference between an algorithm and its representation.
  - Analogy: the difference between a story and a book
- A **program** is a **representation** of an algorithm
- A **process** is the **activity of executing** an algorithm
  - **Terminating process**
    - Finish with a result
  - **Non-terminating process**
    - Do not produce an answer
    - Chapter 12: “Non-deterministic Algorithms”

Outline

- The concept of an algorithm
- Algorithm representation
- Algorithm discovery and structures
- Efficiency and correctness

Algorithm Representation (Program)

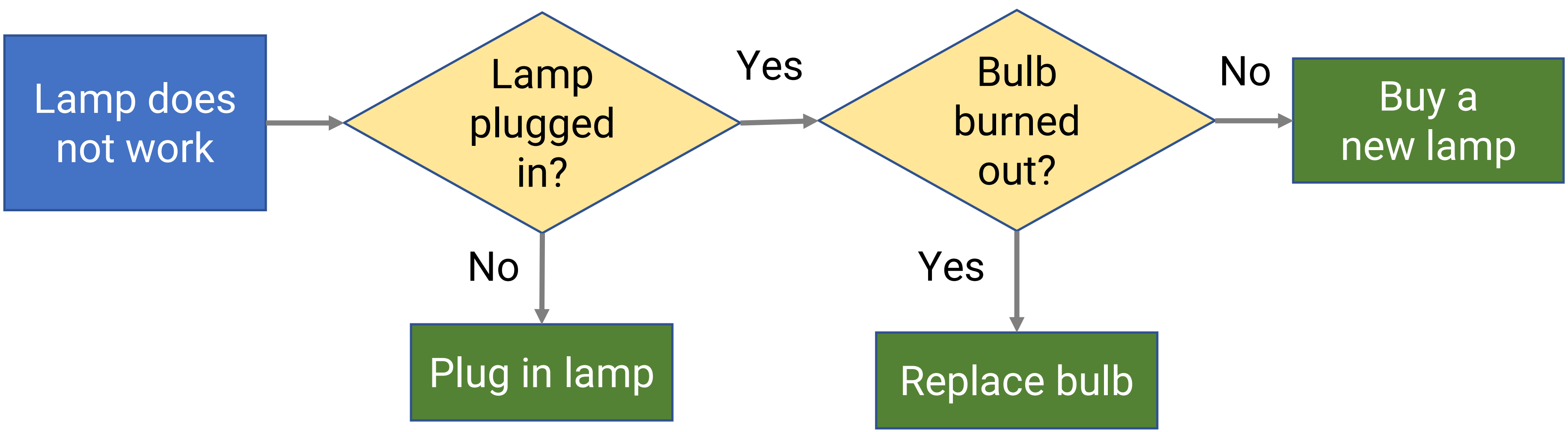
- Formally with **well-defined** **Primitives**



- For programs, a collection of primitives constitutes a **programming language**
  - Value assignment, conditional selection, repeated execution, … etc.

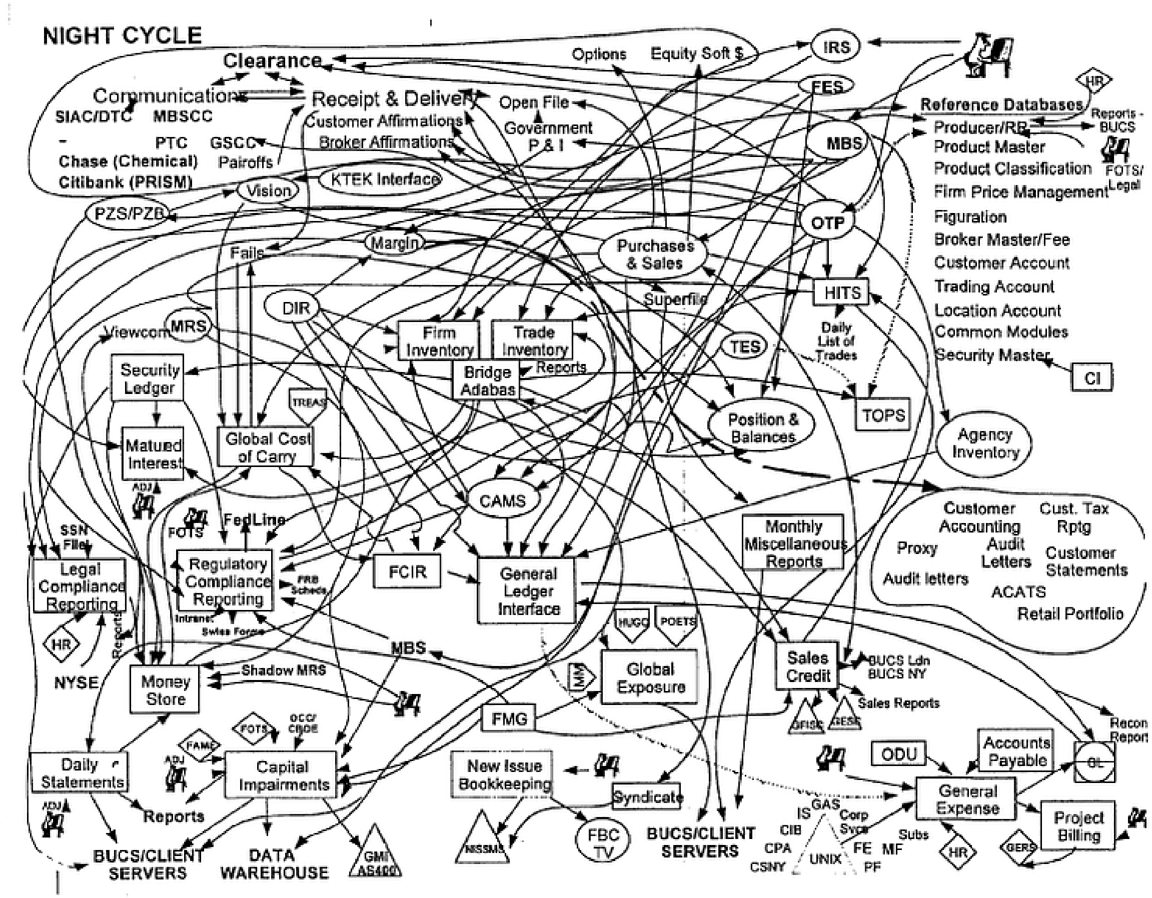
Algorithm Representation (Program)

- Informally with **flowchart** or **pseudocode**
- **Flowchart**
  - Popular in the 50s and 60s
  - Overwhelming for complex algorithms



Algorithm Representation (cont.)

- A very complex flowchart



Designing a Pseudocode Language

- Informally with **flowchart** or **pseudocode**
- **Flowchart**
  - Popular in the 50s and 60s
  - Overwhelming for complex algorithms
- **Pseudocode:** a loose version of formal programming languages
  - Choose a common programming language
  - Loosen some of the syntax rules
  - Allow for some natural language
  - Use consistent, concise notation

Pseudocode Primitives

- **Assignment**
  - Name 🡐 expression
- **Conditional selection**
  - if (condition)
    then (activity)
- **Repeated execution**
  - while (condition)
    do (activity)
- **Procedure**
  - Procedure name

Algorithm *Grade*

Input: the numeric score of each student

Output: a letter grade for each student

For (the score S of each student)

  If $S \geq 90$ **then**

    Return grade A

  Endif

  If $S \geq 80$ and $S < 90$ **then**

    Return grade B

  Else

    Return grade C

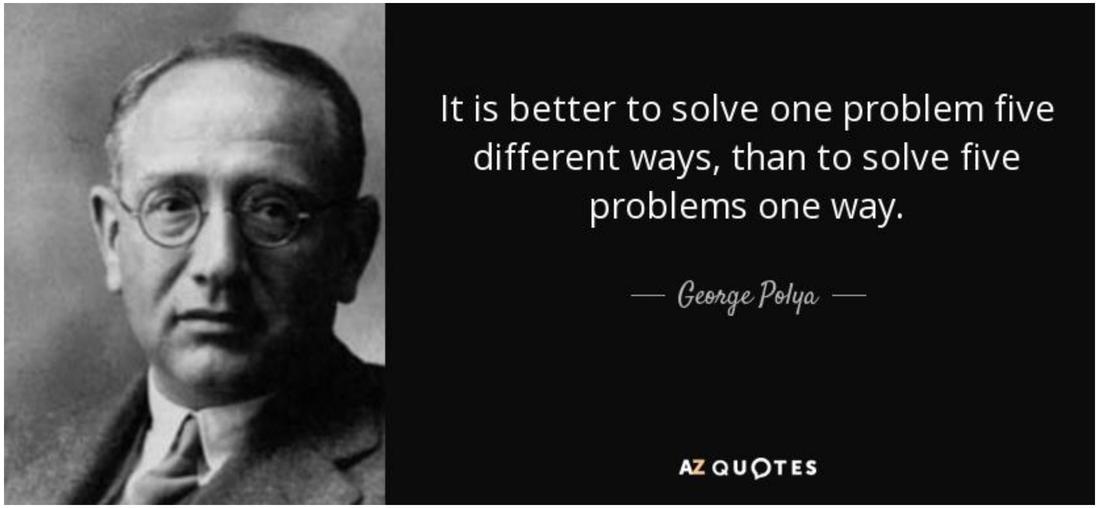
  Endif

Outline

- The concept of an algorithm
- Algorithm representation
- Algorithm discovery and structures
- Efficiency and correctness

Polya's Problem Solving Steps

1. Understand the problem
2. Devise a plan for solving the problem
3. Carry out the plan
4. Evaluate the solution for accuracy and its potential as a tool for solving other problems



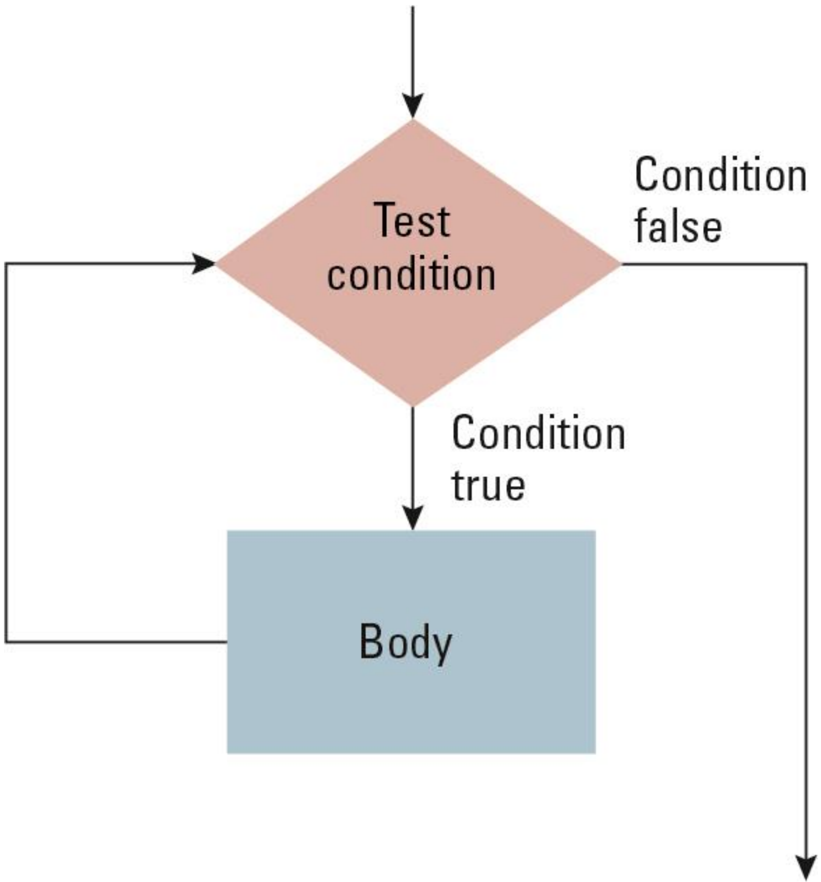
Problem Solving

- Iterative v.s. Recursive
- Top-down v.s. Bottom-up

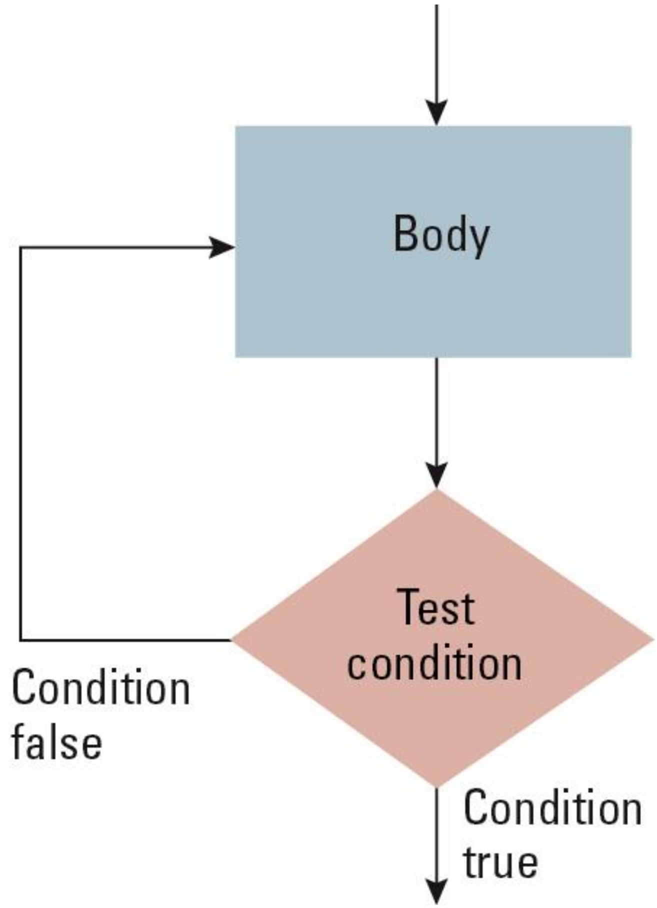
Iterative Structures

- **Loop control**
  - **Initializer**
    - Establish an initial state that will be modified toward the termination condition
  - **Test**
    - Compare the current state to the termination condition and terminate the repetition if equal
  - **Modify**
    - Change the state in such a way that if moves toward the termination condition

Loops



Pre-test (while, for)



Post-test (do...while)

Example: Insertion Sort

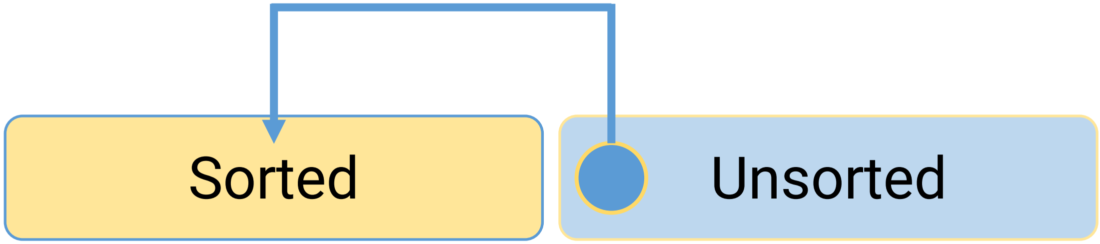
AQJKK

QAJKK

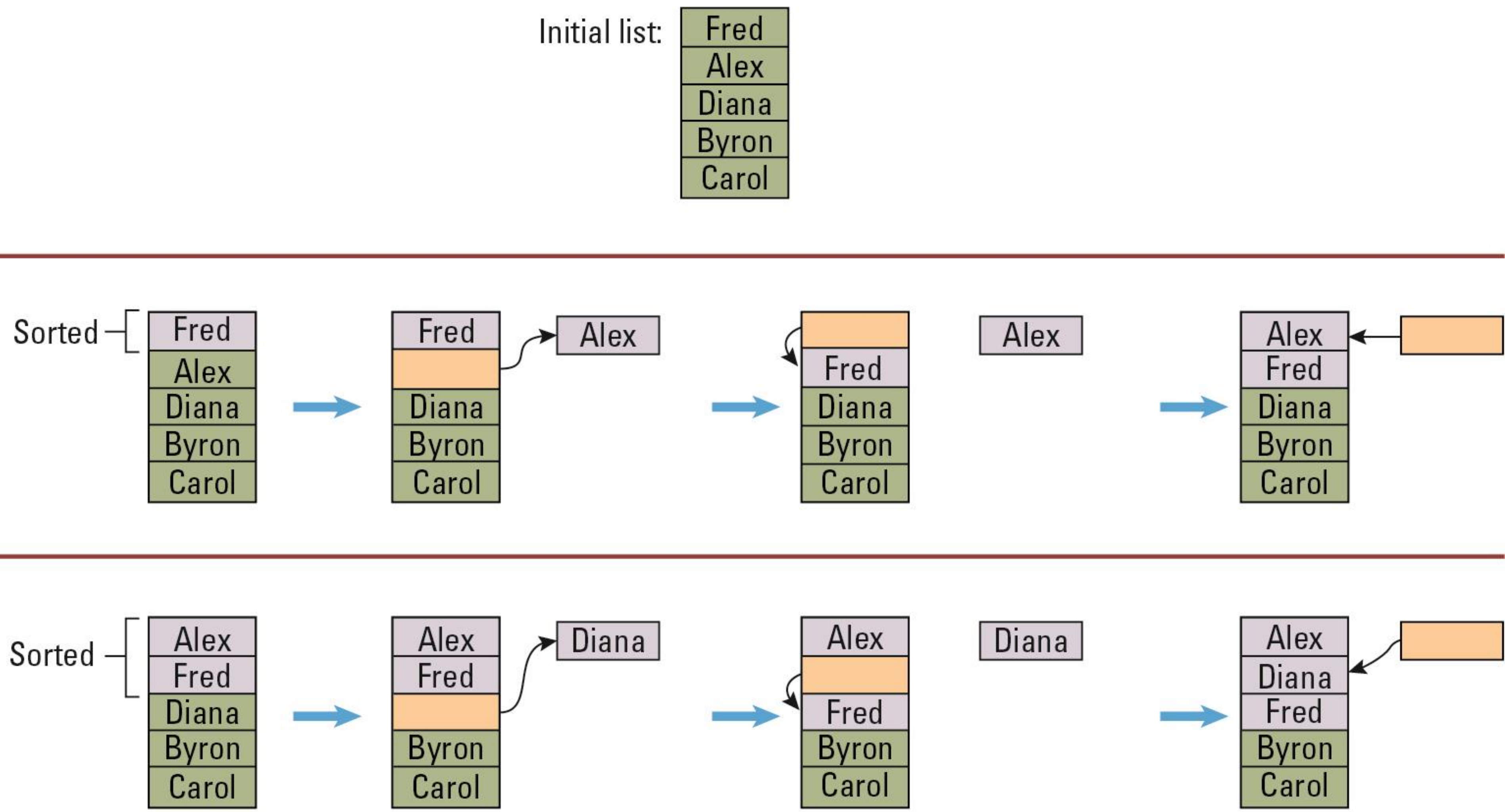
JQAKK

JQKAK

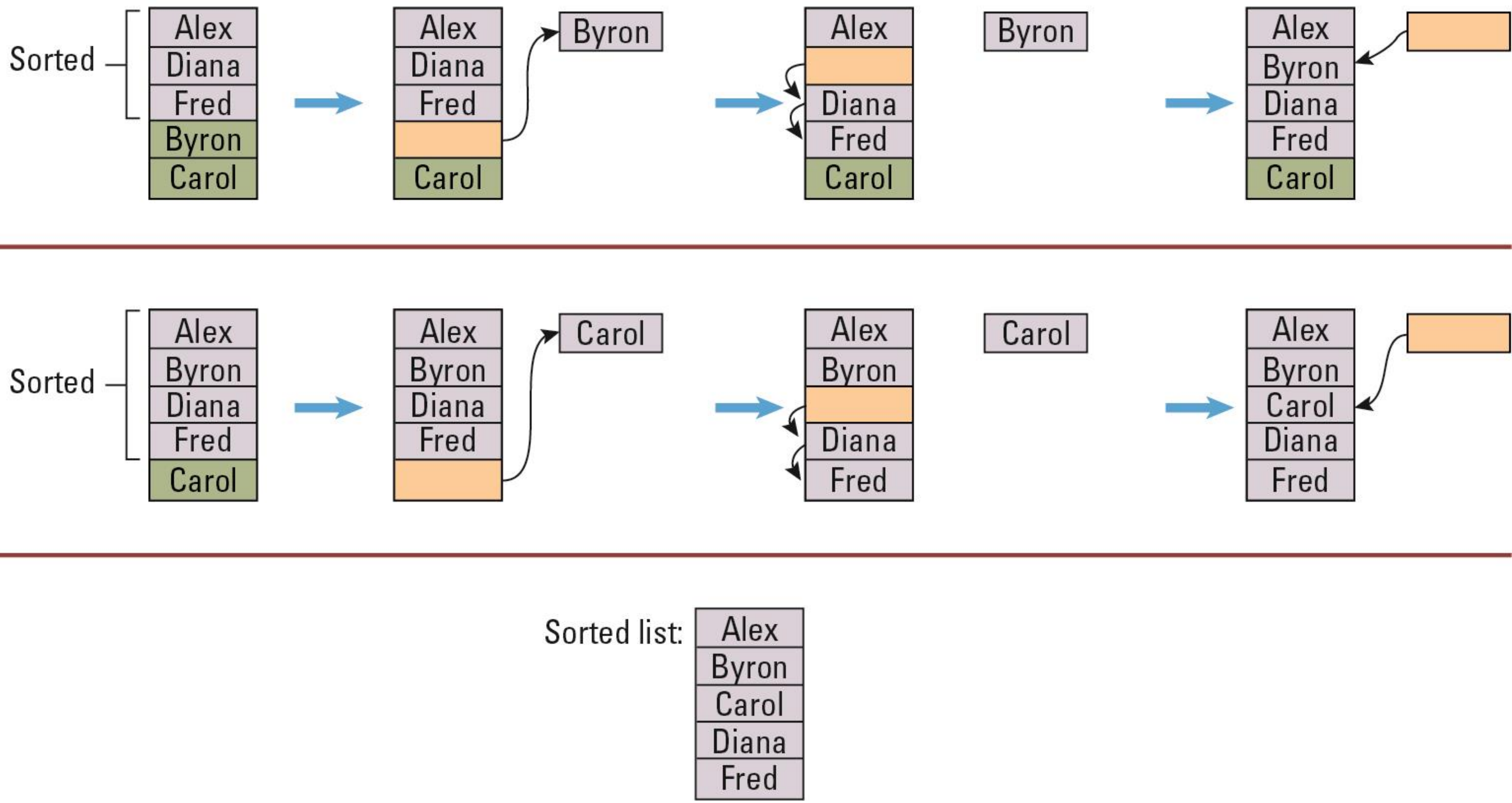
JQKKA



Example: Insertion Sort (cont.)



Example: Insertion Sort (cont.)



Example: Insertion Sort (cont.)

Procedure *InsertionSort (List)*

N ← 2

while (the value of *N* does not exceed the length of *List*) **do**

  Select the *N-th* entry in *List* as the pivot entry

  while (there is a name above the hole and that name is

    greater than the pivot) **do**

    Move the name above the hole down into the hole,

    leaving a hole above the name

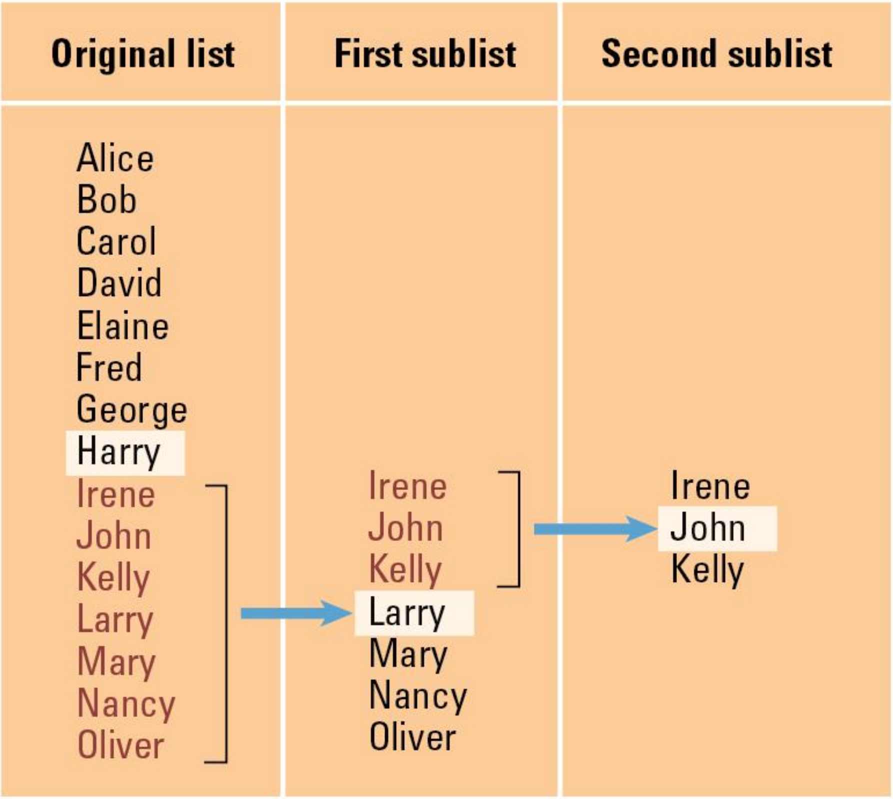
  Move the pivot entry into the hole in *List*

  N ← N + 1

Recursive Structures

- Repeating the set of instructions as a **subtask** of itself
- A classic example: the binary search algorithm

Is John in the array?



Binary Search Pseudo Code

Procedure *BinarySearch (List, TargetValue)*

if (*List* empty) **then**

    Report that the search failed

else

    Select the middle in *List* to be the *TestEntry*

    Execute the instructions below based on different cases

        case 1: *TargetValue* == *TestEntry*

            Report that the search succeeded

        case 2: *TargetValue* < *TestEntry*

            Search the portion of *List* preceding *TestEntry*

BinarySearch(
  FirstHalfList*,*
  TargetValue
)

        case 3: *TargetValue* > *TestEntry*

            Search the portion of *List* succeeding *TestEntry*

*BinarySearch(****SecondHalfList****, TargetValue)*

endif

Recursive Problem Solving

- Do not abuse recursion!
  - Calling functions takes a long time
    - Memory allocation, parameters passing … etc.
  - Example: Factorial

```
int factorial (int x) {
    if (x == 0) return 1;
    return x * factorial(x – 1);
}
```

recursive

factorial(3) =
*3 * factorial(2) =*
*3 * 2 * factorial(1) =*
*3 * 2 * 1 * factorial(0) =*
*3 * 2 * 1 * 1*

```
int factorial (int x) {
    int product = 1;
    for (int i = 1; i <= x ; ++i)
        product *= i;
    return product;
}
```

iterative

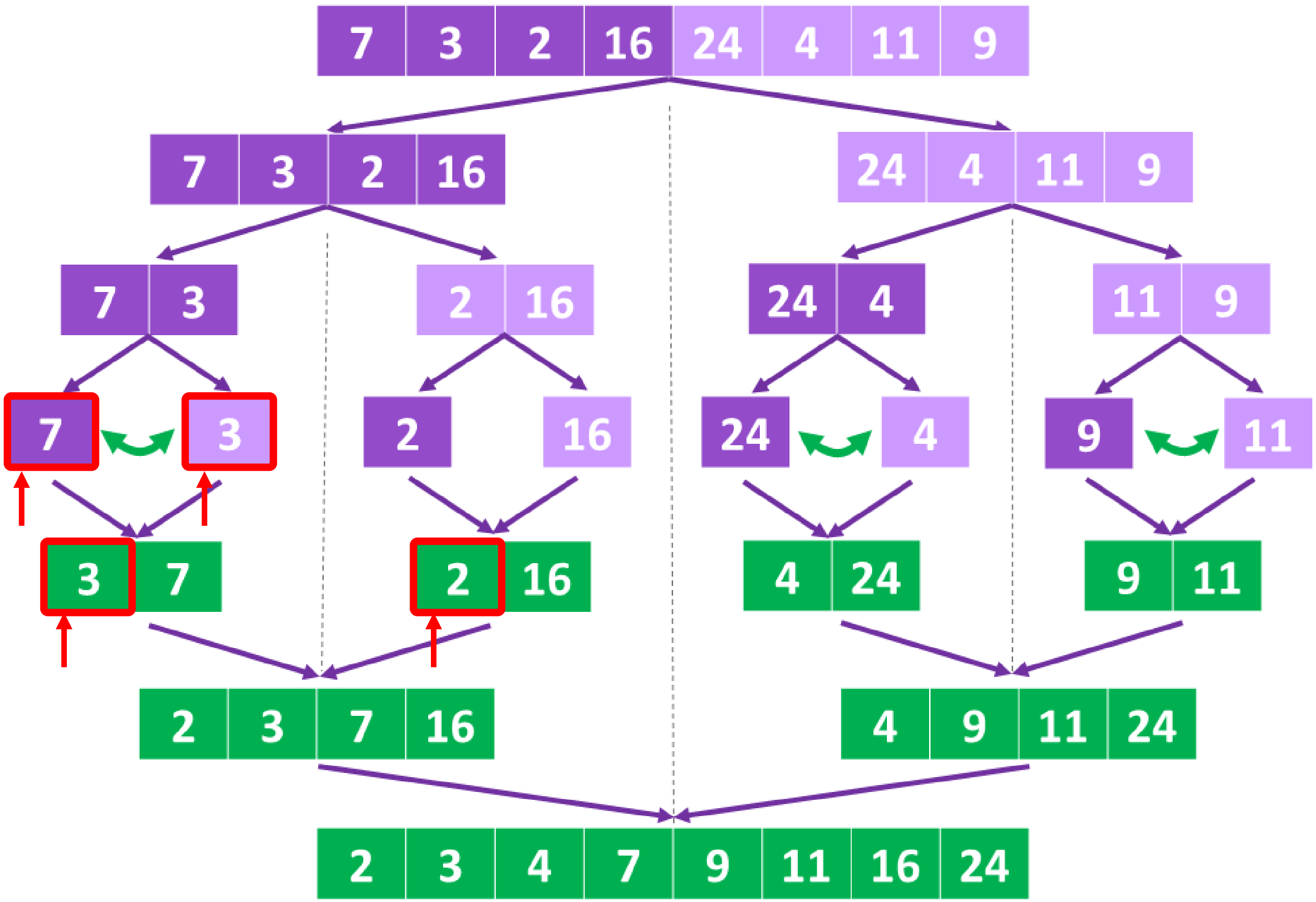
Problem Solving (cont.)

- Iterative v.s. Recursive
- Top-down v.s. Bottom-up

Top-down Approach

- Stepwise refinement
- **Divide and conquer (problem decomposition)**
- Examples
    - Binary search
    - Merge sort

Merge Sort



Step 1:
Split sub-lists in two until you reach pair of values.

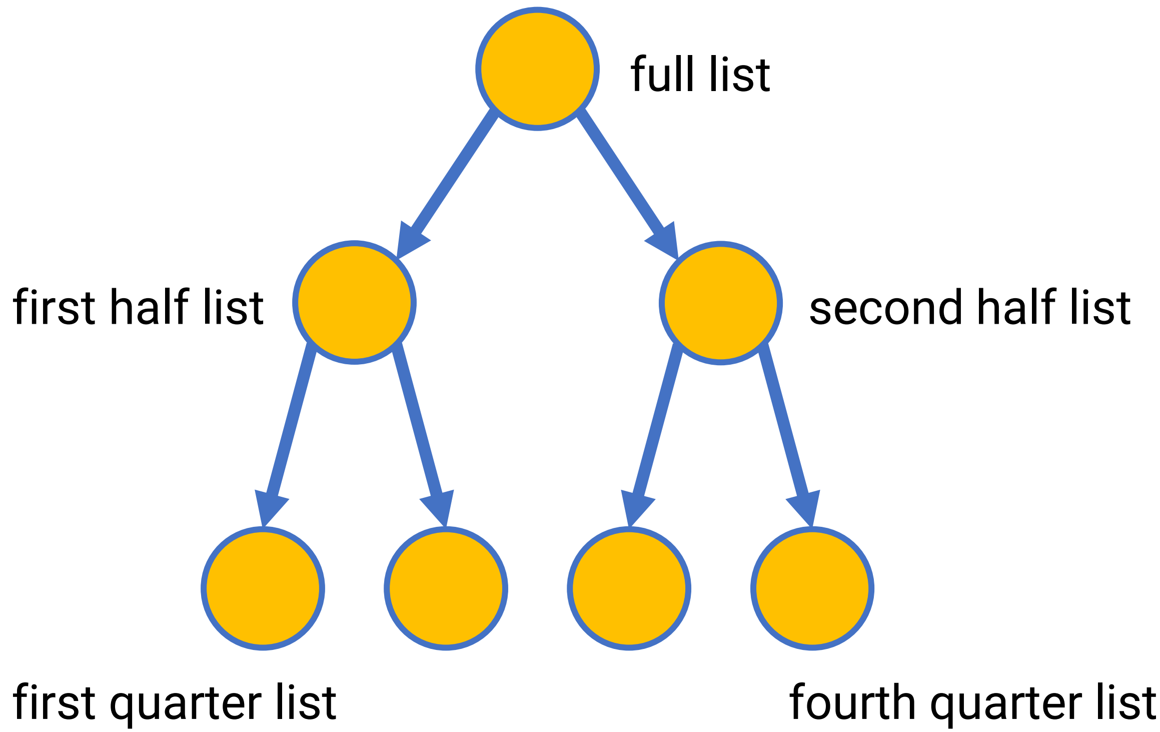
Step 3:
Sort/swap pair of values if needed.

Step 4:
Merge and sort sub-lists and repeat process till you merge to the full list.

from 101 Computing.net: https://www.101computing.net/merge-sort-algorithm/

Top-down Approach Review

- Stepwise refinement
- **Divide and conquer (problem decomposition)**
- Examples
  - Binary search
  - Merge sort

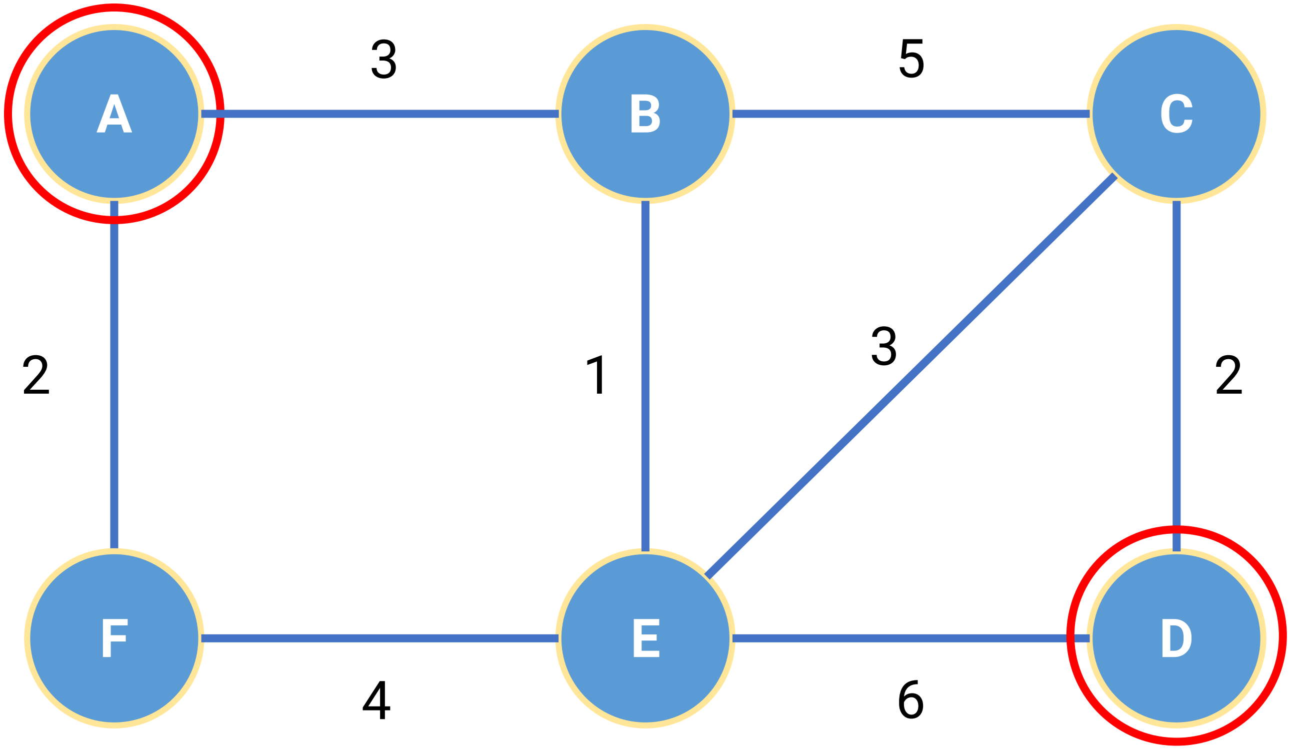


Bottom-up Approach

- Solve pieces of the problem first
- Relax some of the problem constraints
- **Dynamic programming (DP)**
- Example
  - Shortest path

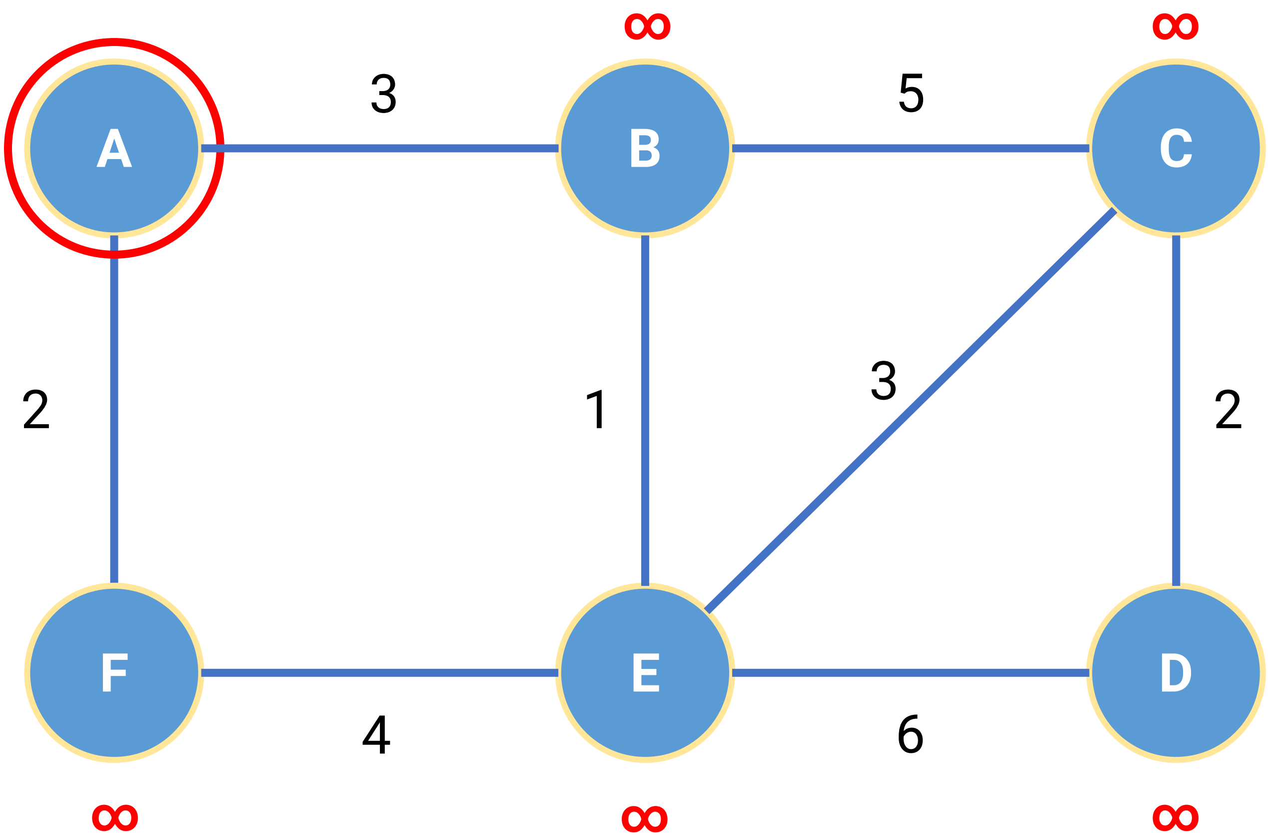
Shortest Path

$$Shortest_{\boldsymbol{AD}} = \mathbf{min}_{\ i\, \in \{A,\, B,\, C,\, D,\, E,\, F\}} \left(Shortest_{\boldsymbol{Ai}} + Shortest_{\boldsymbol{iD}}\right)$$



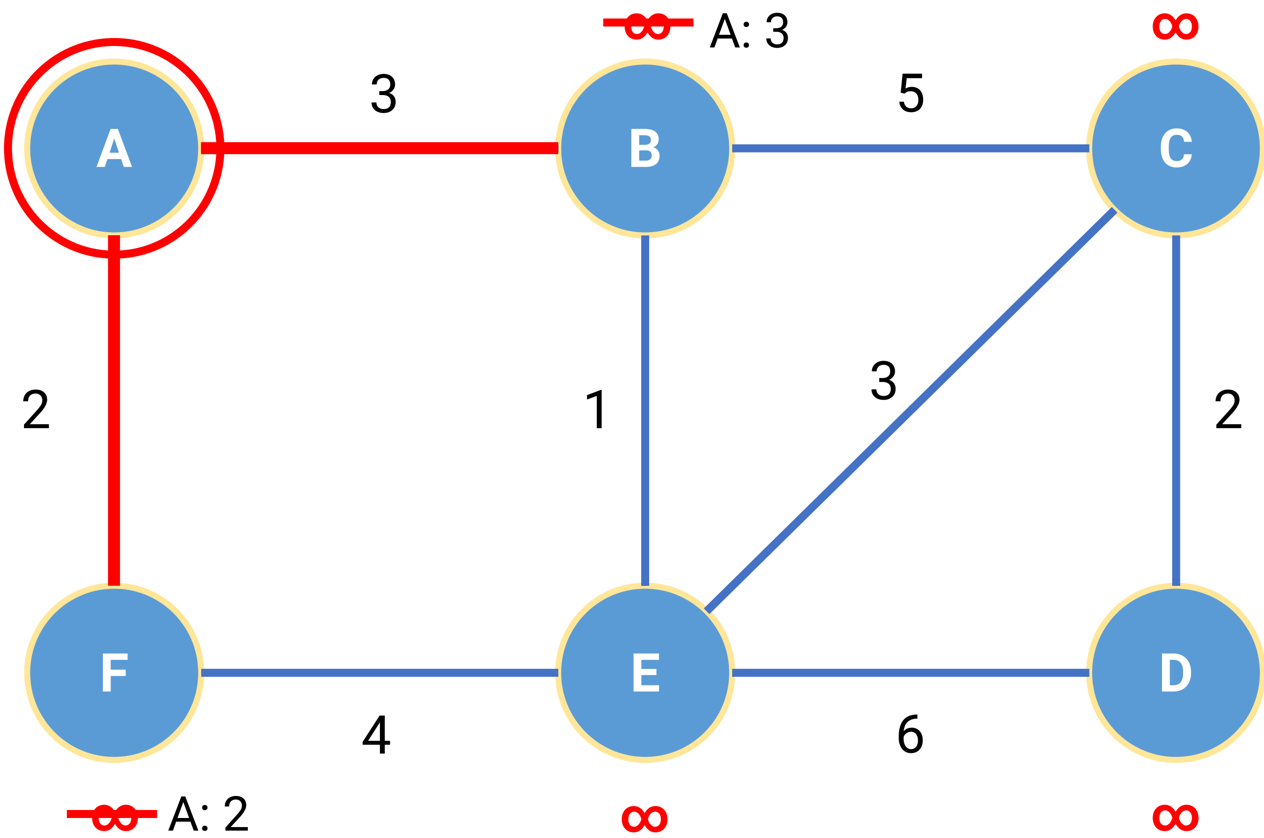
Shortest Path (cont.)

$$Shortest_{AD} = \mathbf{min}_{\ i \in \{A, B, C, D, E, F\}} \left(Shortest_{Ai} + Shortest_{iD}\right)$$



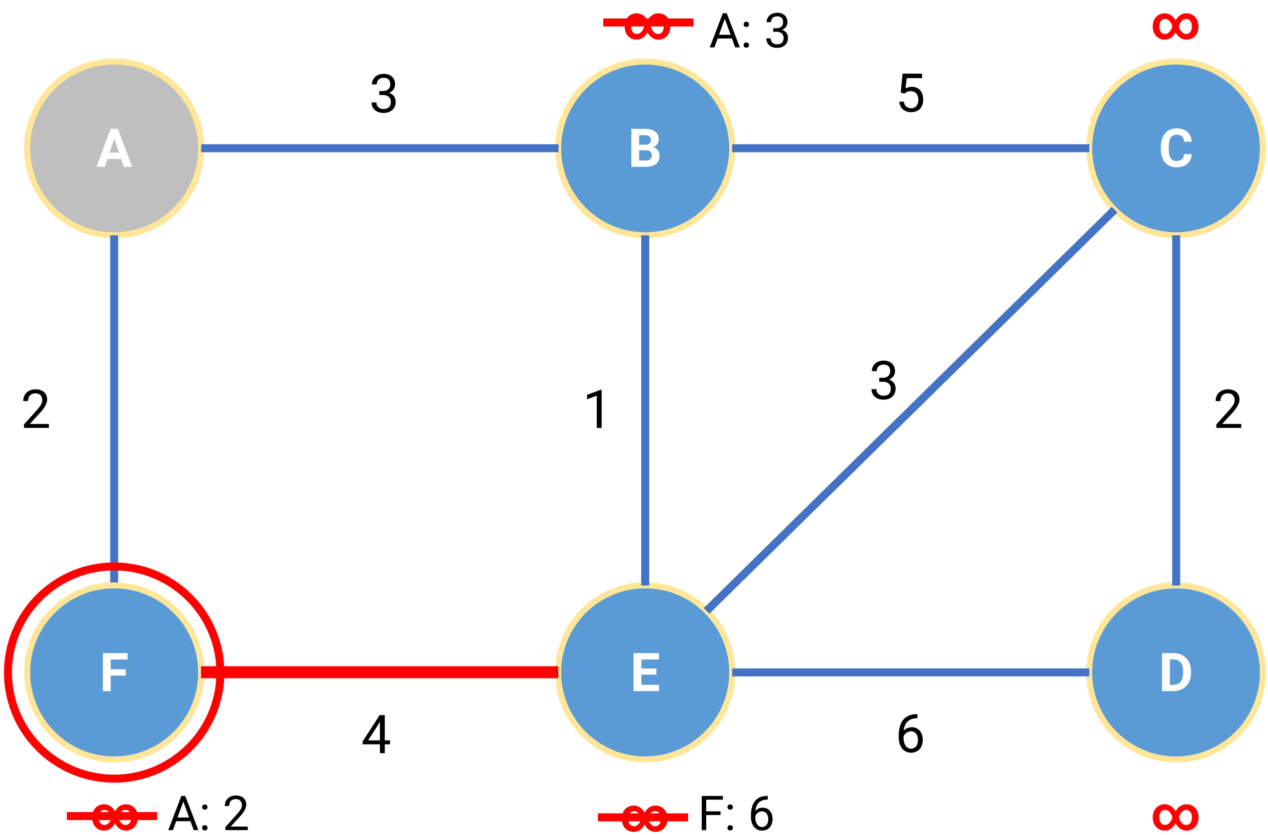
Shortest Path (cont.)

$$Shortest_{AD} = \mathbf{min}_{\ i \in \{A,\ B,\ C,\ D,\ E,\ F\}} (Shortest_{Ai} + Shortest_{iD})$$



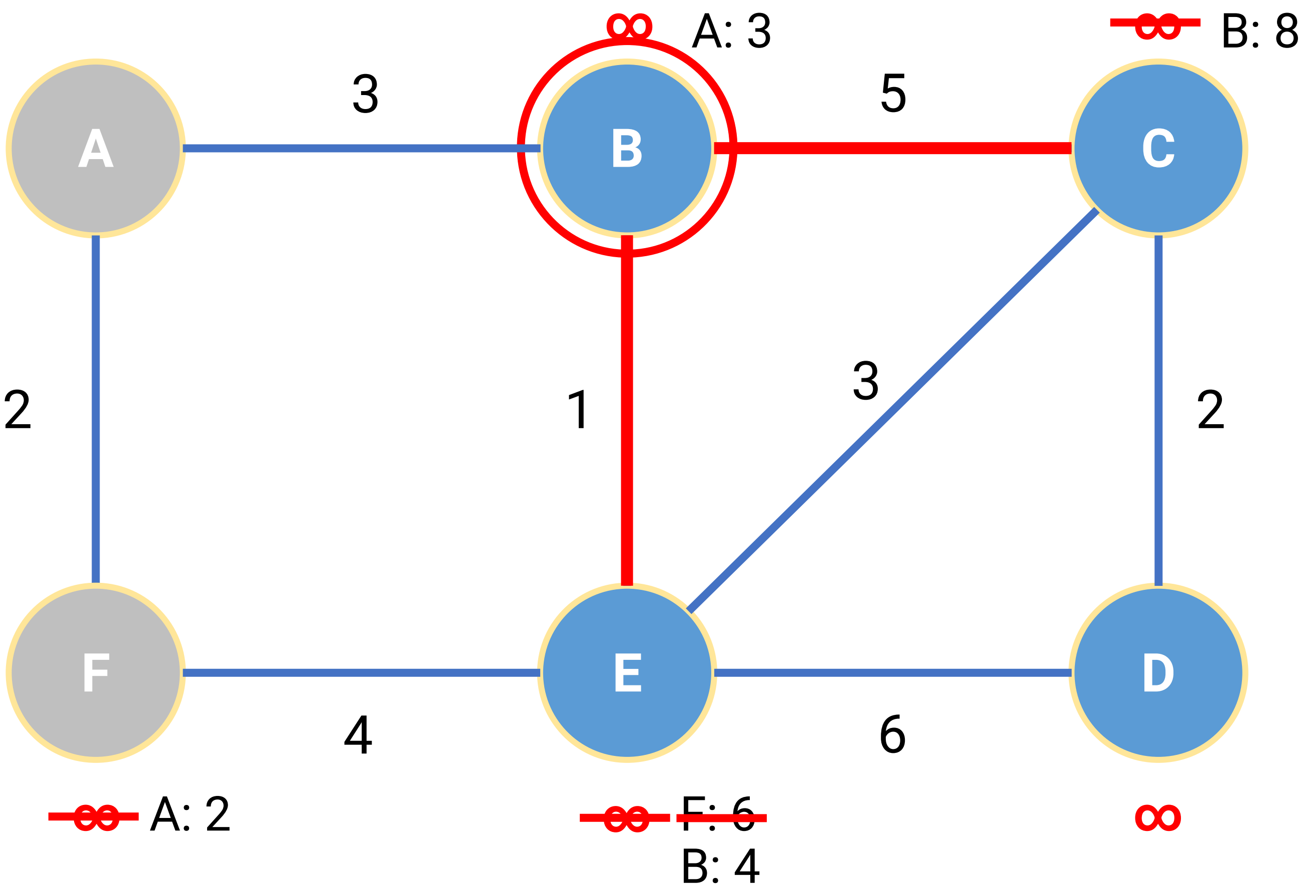
Shortest Path (cont.)

$$Shortest_{AD} = \textbf{min}_{\ i \in \{A,\ B,\ C,\ D,\ E,\ F\}} (Shortest_{Ai} + Shortest_{iD})$$



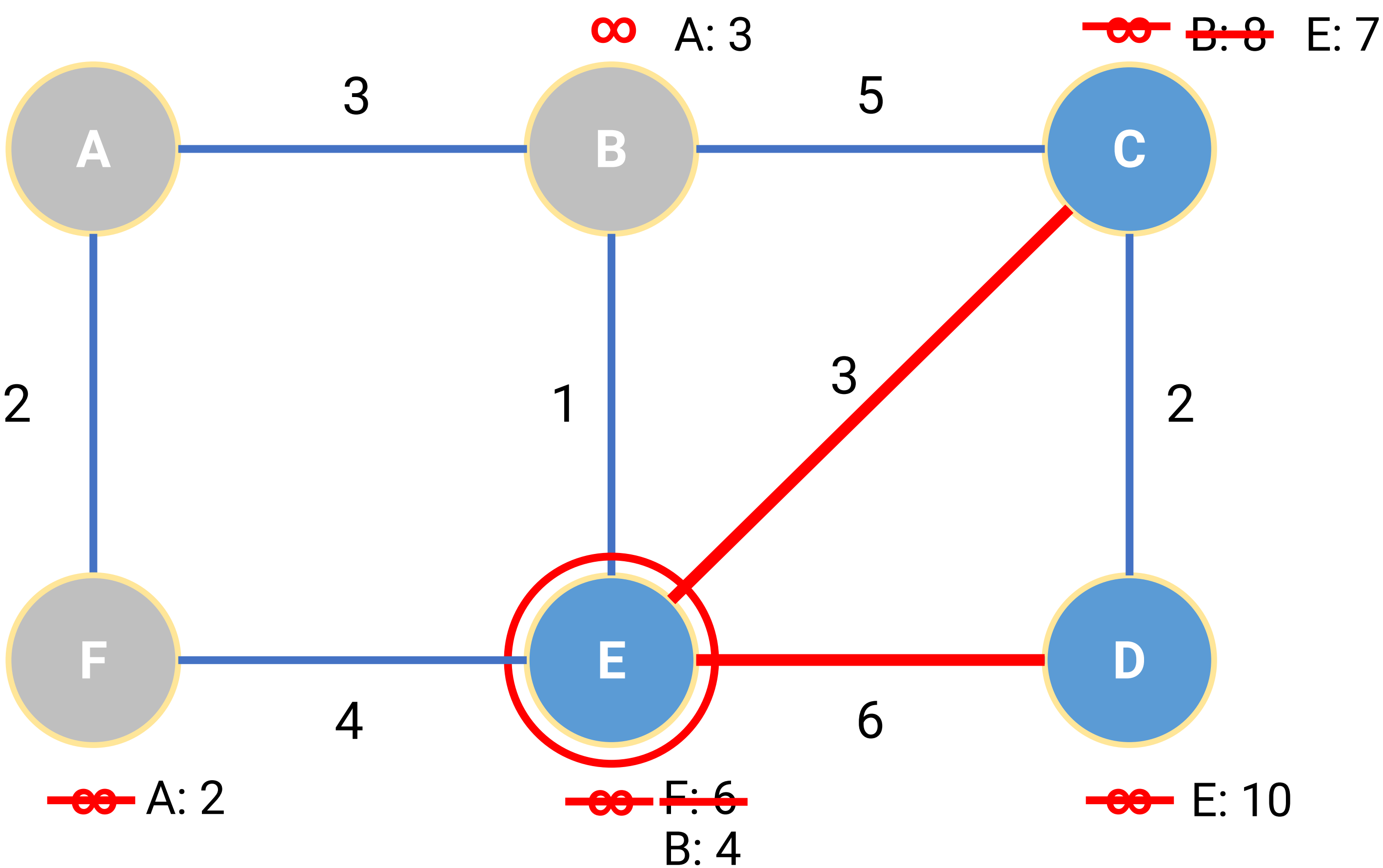
Shortest Path (cont.)

$$Shortest_{AD} = \textbf{min}_{\,i \in \{A, B, C, D, E, F\}} (Shortest_{Ai} + Shortest_{iD})$$



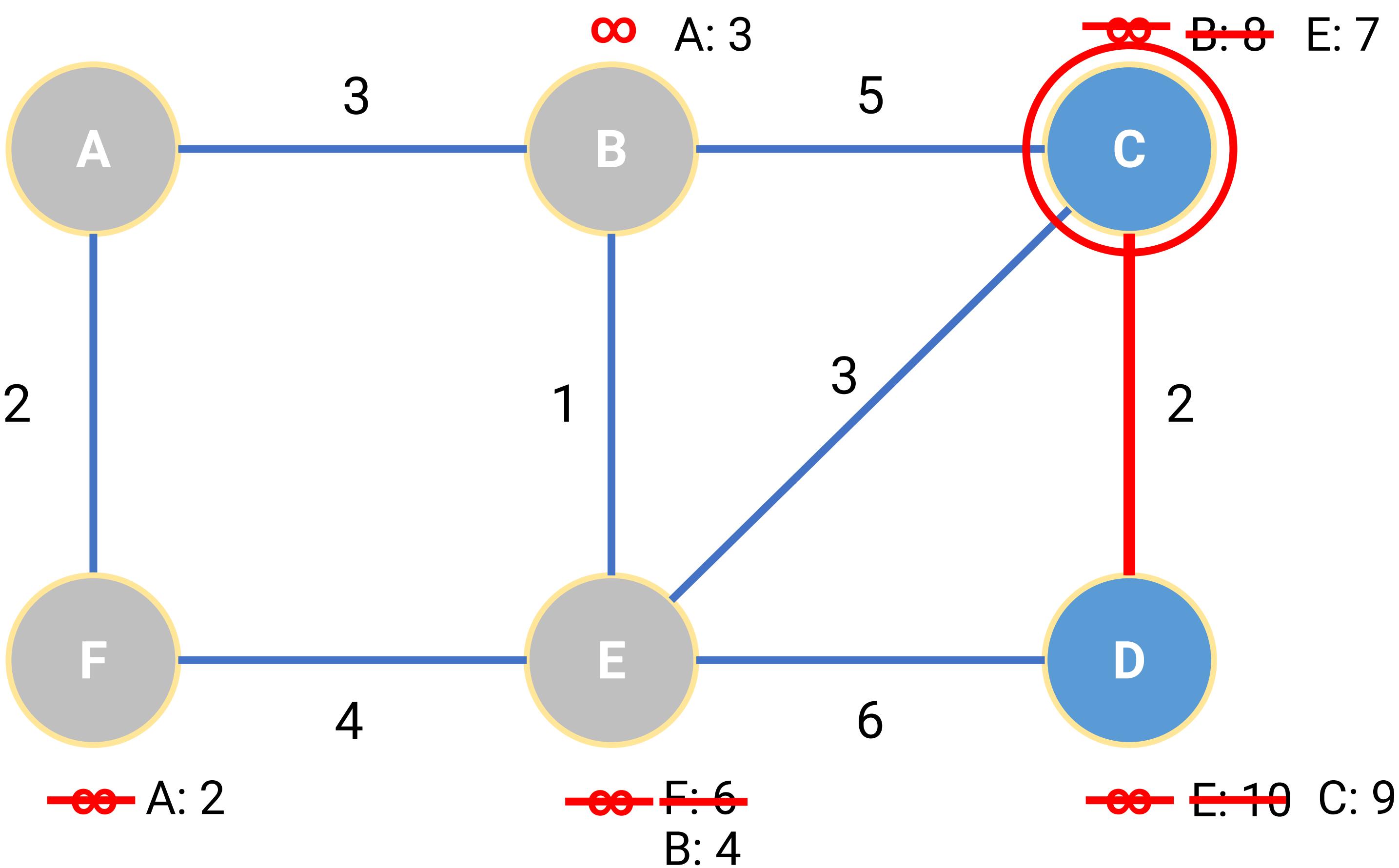
Shortest Path (cont.)

$$Shortest_{AD} = \mathbf{min}_{\ i \in \{A, B, C, D, E, F\}} (Shortest_{Ai} + Shortest_{iD})$$



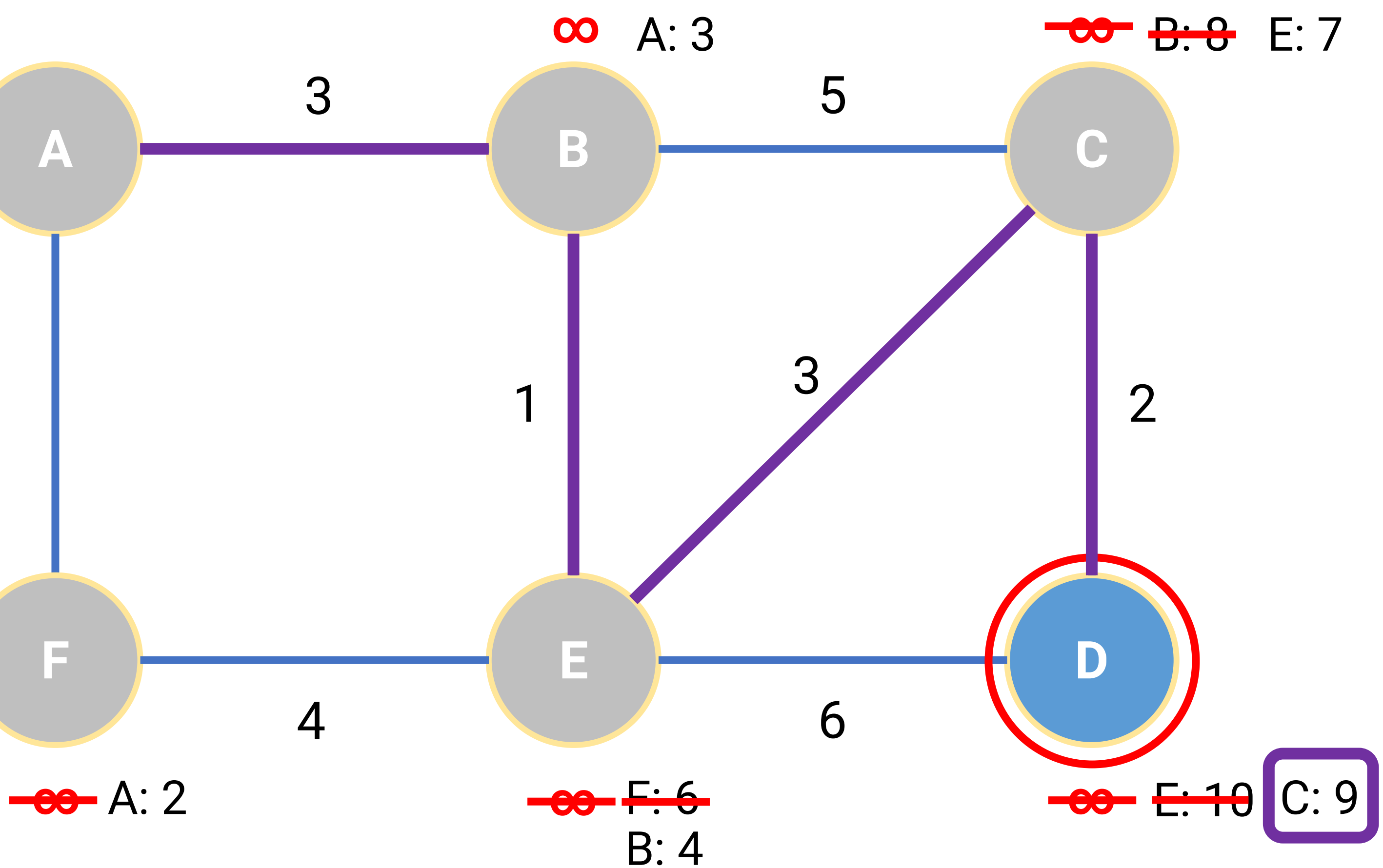
Shortest Path (cont.)

$$Shortest_{AD} = \textbf{min}_{\ i \in \{A, B, C, D, E, F\}} \left(Shortest_{Ai} + Shortest_{iD}\right)$$



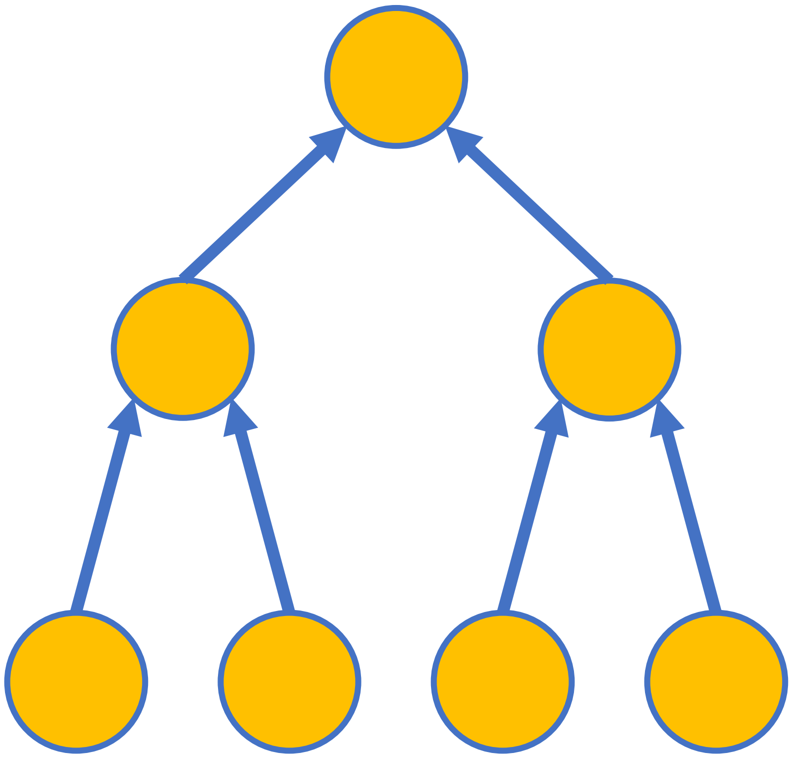
Shortest Path (cont.)

$$Shortest_{AD} = \mathbf{min}_{\ i \in \{A, B, C, D, E, F\}} (Shortest_{Ai} + Shortest_{iD})$$



Bottom-up Approach

- Solve pieces of the problem first
- Relax some of the problem constraints
- **Dynamic programming (DP)**
- Example
  - Shortest path



Outline

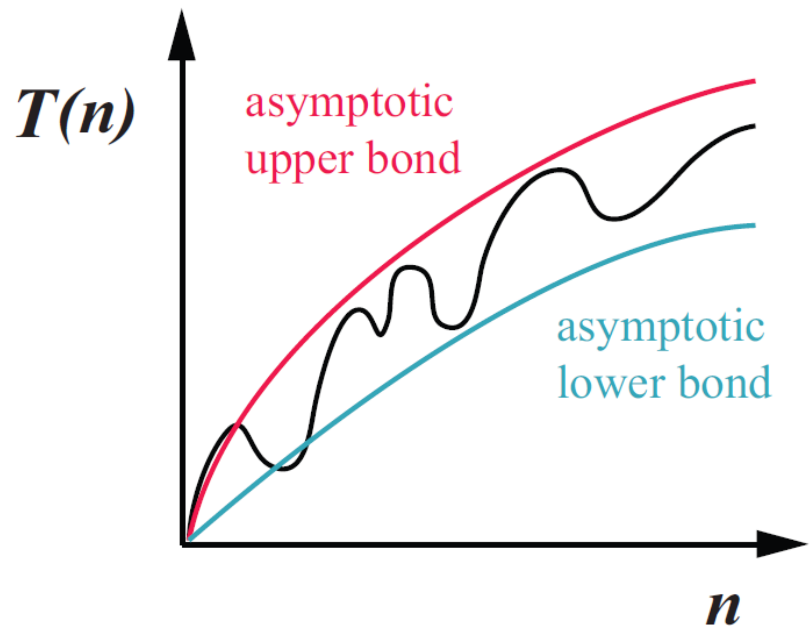
- The concept of an algorithm
- Algorithm representation
- Algorithm discovery and structures
- Efficiency and correctness

Efficiency

- The choice between efficient and inefficient algorithms can make the difference between a practical solution and an impractical one
- Measured as the **number of instructions** executed
  - Why not use the execution time
    - What about on different machines?

Asymptotic Analysis

- Exact analysis is often difficult and tedious
- **Asymptotic analysis** emphasizes the behavior of the algorithm when ***n*** tends to **infinity**
- **Asymptotic**
  - Upper bound (***O***)
  - Lower bound (***Ω***)
  - Tight bound (***Θ***)

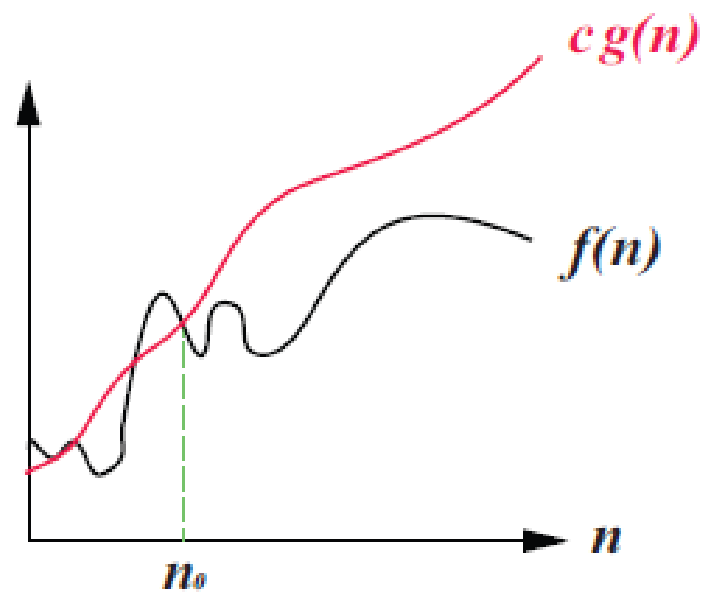


Big-O

$$O(g(n)) = \{f(n) \mid \underbrace{\exists}_{\text{exist}} c > 0, n_0 > 0 \underbrace{\text{s.t.}}_{\text{such that}} \underbrace{\forall}_{\text{for each}} n \geq n_0, 0 \leq f(n) \leq cg(n)\}$$

- **Asymptotic upper bound**

- Examples
  - $500n = O(n^2)$
  - $n^{10} = O(2^n)$
  - $5n + 10000 = O(n)$



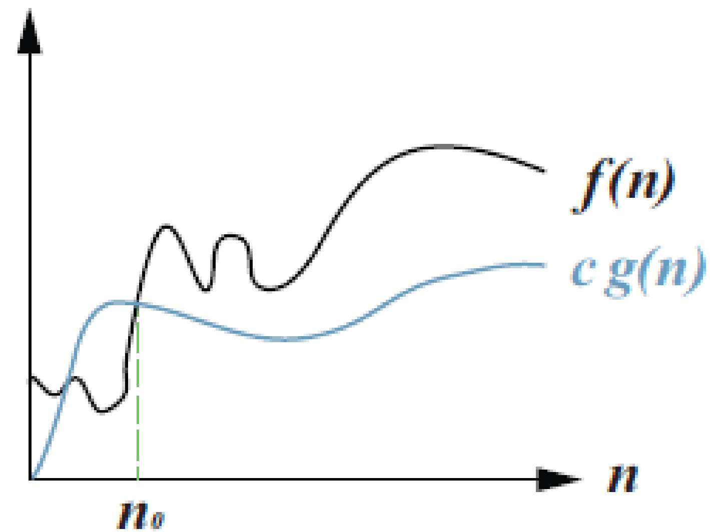
$f(n) = O(g(n))$

Big-Ω

$$\Omega(g(n)) = \{f(n) \mid \underbrace{\exists}_{\text{exist}} c > 0, n_0 > 0 \underbrace{\text{s.t.}}_{\text{such that}} \underbrace{\forall}_{\text{for each}} n \geq n_0, 0 \leq cg(n) \leq f(n)\}$$

- **Asymptotic lower bound**

- Examples
  - $0.001n^2 = \Omega(n)$
  - $2^n = \Omega(n^{10})$
  - $5n + 10000 = \Omega(n)$



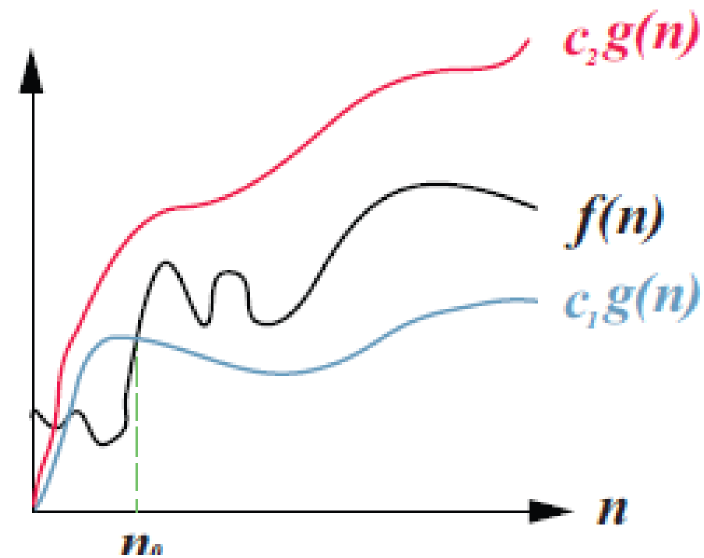
$f(n) = \Omega(g(n))$

Big-Θ

$$\Theta(g(n)) = \{f(n) \mid \underset{\text{exist}}{\exists} c_1, c_2, n_0 > 0 \ \underset{\text{such that}}{\text{s.t.}} \ \underset{\text{for each}}{\forall} n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

- **Asymptotic tight bound**

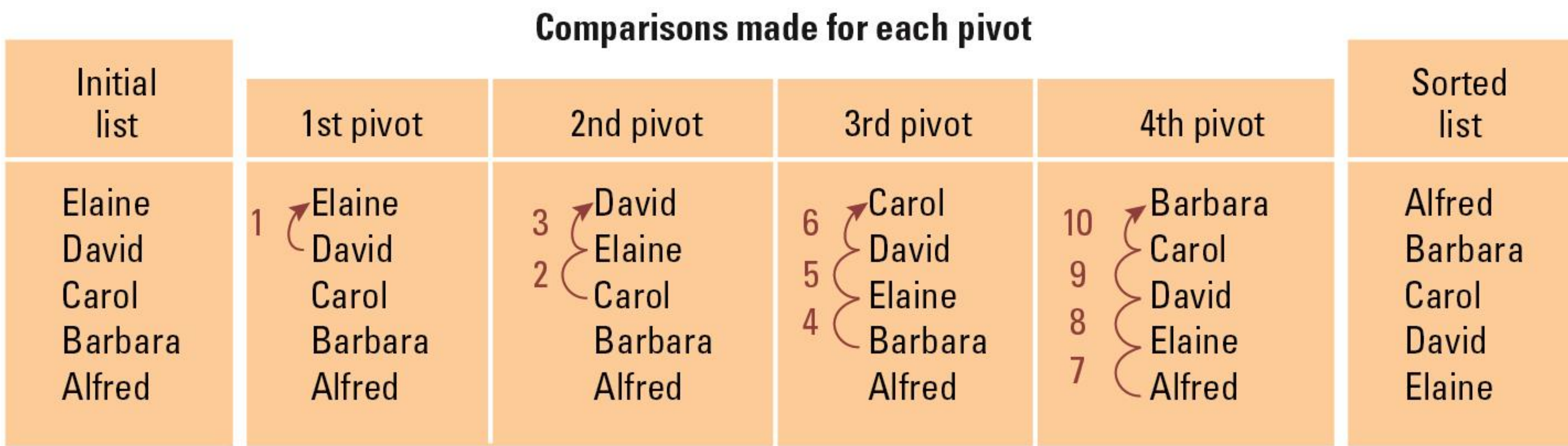
- Examples
  - $0.001n^2 = \Theta(n^2)$
  - $n + \log n = \Theta(n)$
  - $5n + 10000 = \Theta(n)$



$f(n) = \Theta(g(n))$

Efficiency (cont.)

- Incorporates **best**, **worst**, and **average** case analysis
- Example: worst case for insertion sort: $O(n^2)$



Worst: $(n^2 - n) / 2$ Best: $(n - 1)$ Average: $\theta(n^2)$

Recap: Insertion Sort

Procedure *InsertionSort (List)*

N ← 2

while (the value of *N* does not exceed the length of *List*) **do**

    Select the *N-th* entry in *List* as the pivot entry

    while (there is a name above the hole and that name is greater than the pivot) **do**

        Move the name above the hole down into the hole, leaving a hole above the name

    Move the pivot entry into the hole in *List*

    N ← N + 1

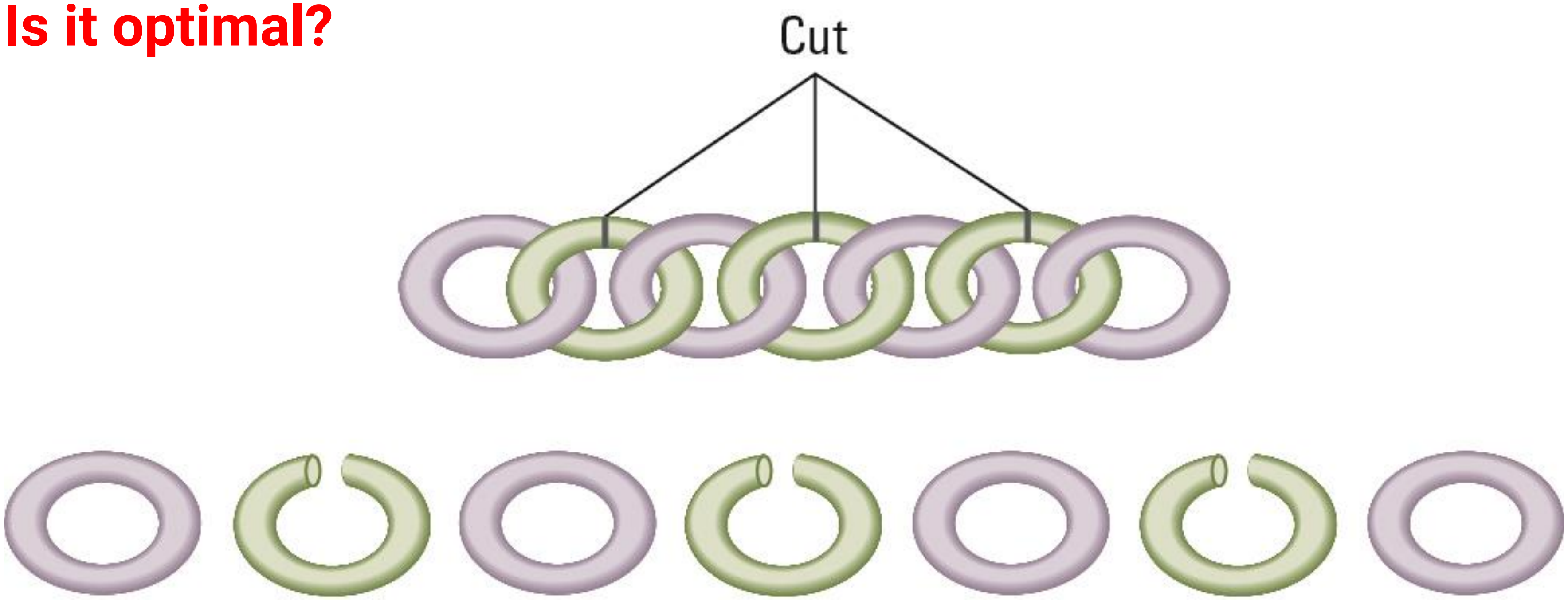
Correctness

- The correctness of an algorithm is determined by **reasoning** formally about the algorithm, not by testing its implementation

Traveler's Gold Chain Problem

A traveler with a gold chain of seven links must stay in an isolated hotel for seven nights. The rent each night consists of one link from the chain. What is the fewest number of links that must be cut so that the traveler can pay the hotel one link of the chain each morning without paying for lodging in advance?

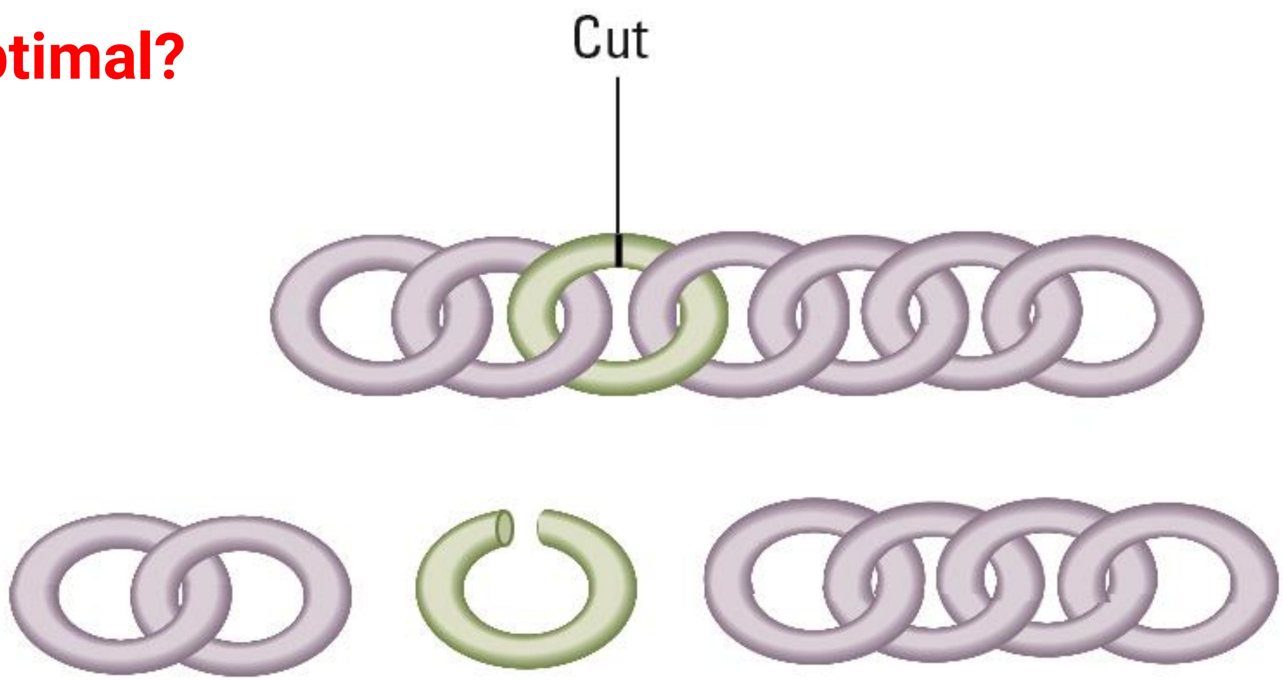
Is it optimal?



Traveler’s Gold Chain Problem (cont.)

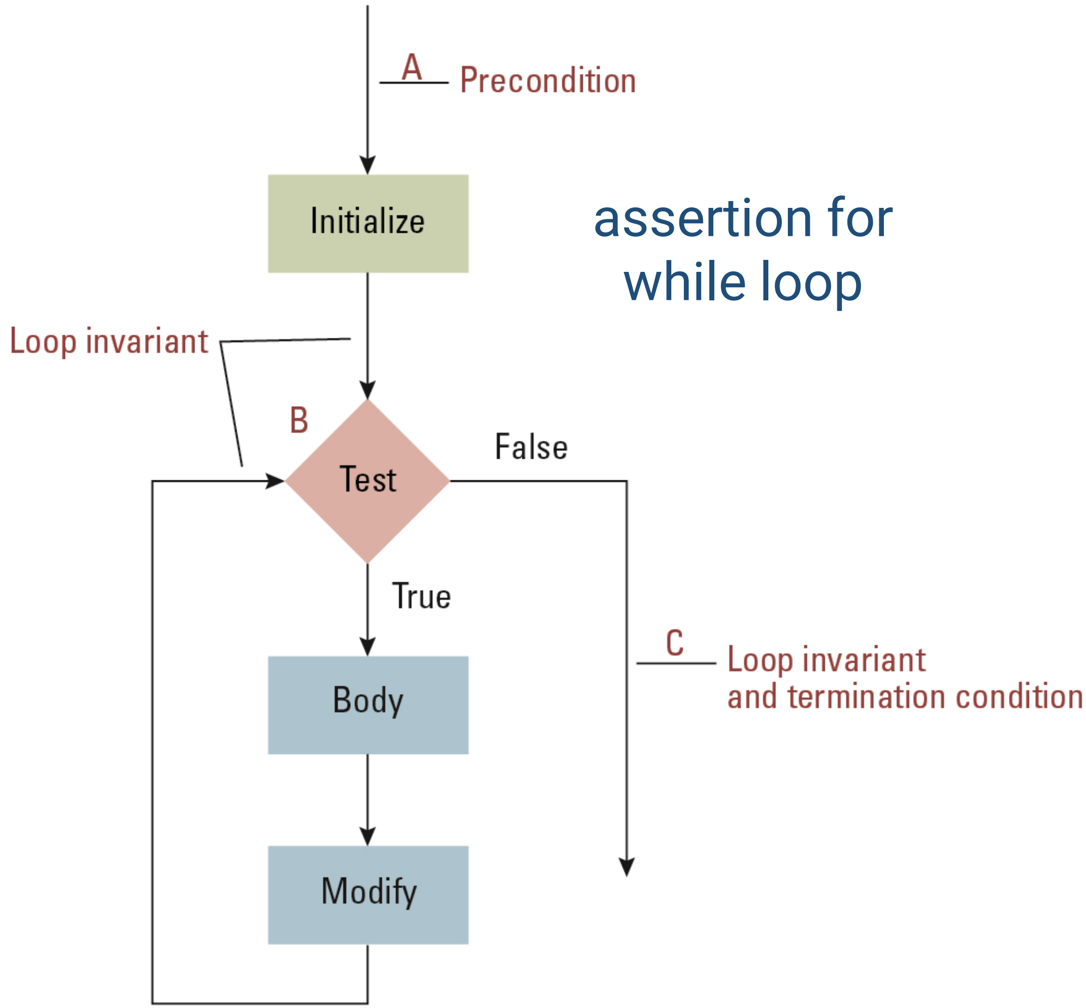
A traveler with a gold chain of seven links must stay in an isolated hotel for seven nights. The rent each night consists of one link from the chain. What is the fewest number of links that must be cut so that the traveler can pay the hotel one link of the chain each morning without paying for lodging in advance?

Is it optimal?



Software Verification

- Proof of correctness (with formal logic)
  - Assertions
    - **Preconditions**
    - **Loop invariants**
    - **Termination condition**



assertion for while loop

Example for Assertion

Procedure *FindQuotient*

Count ← 0

Remainder ← Dividend

do

  Remainder ← Remainder – Divisor

  Count ← Count + 1

while (Remainder < Divisor)

Quotient ← Count

Correct?

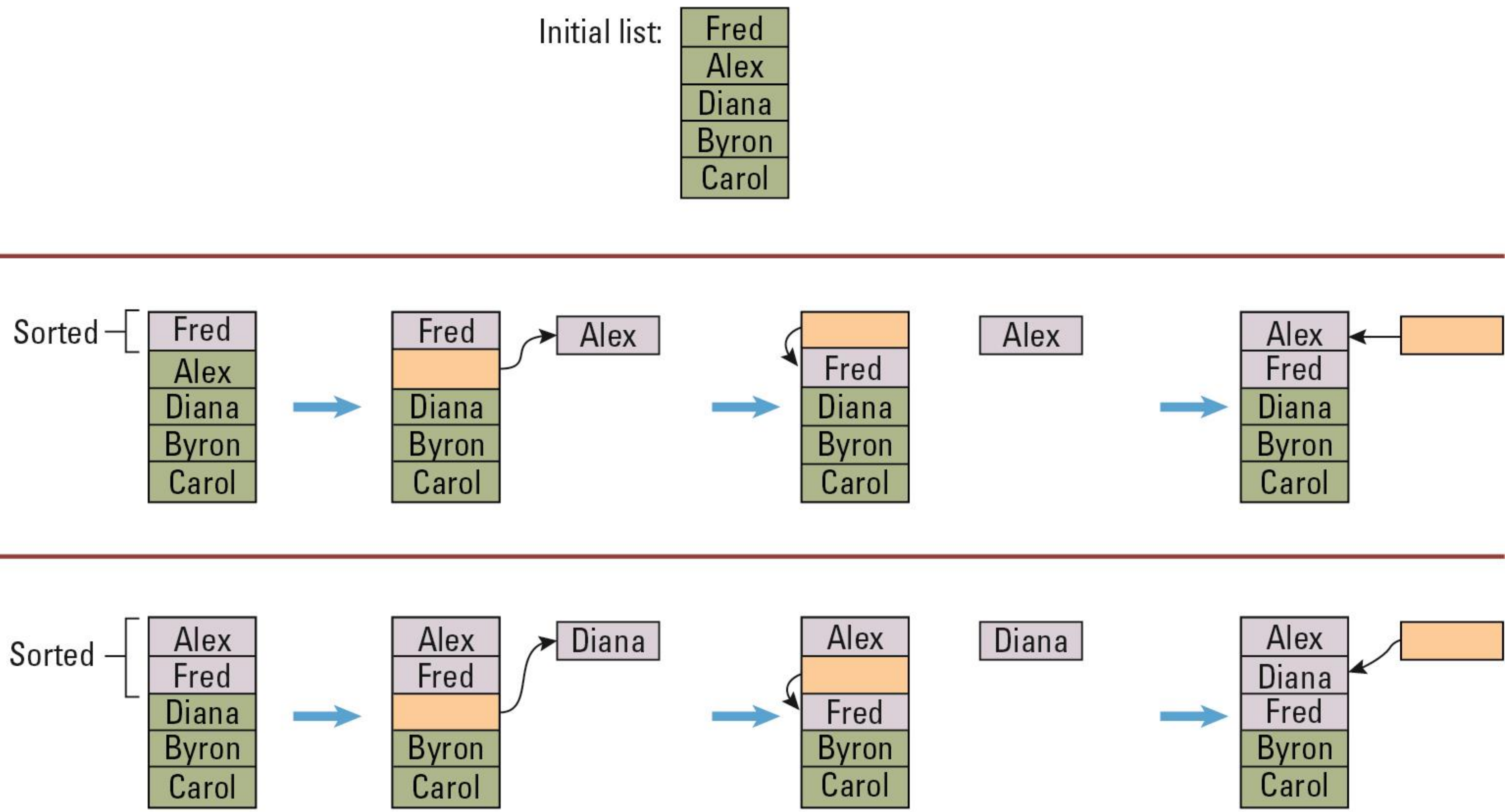
Remainder > 0?

- **Preconditions**
  - Dividend > 0
  - Divisor > 0
  - Count = 0
  - Remainder = Dividend
- **Loop invariants**
  - Dividend > 0
  - Divisor > 0
  - Dividend = Count * Divisor + Remainder
- **Termination condition**
  - Remainder < Divisor

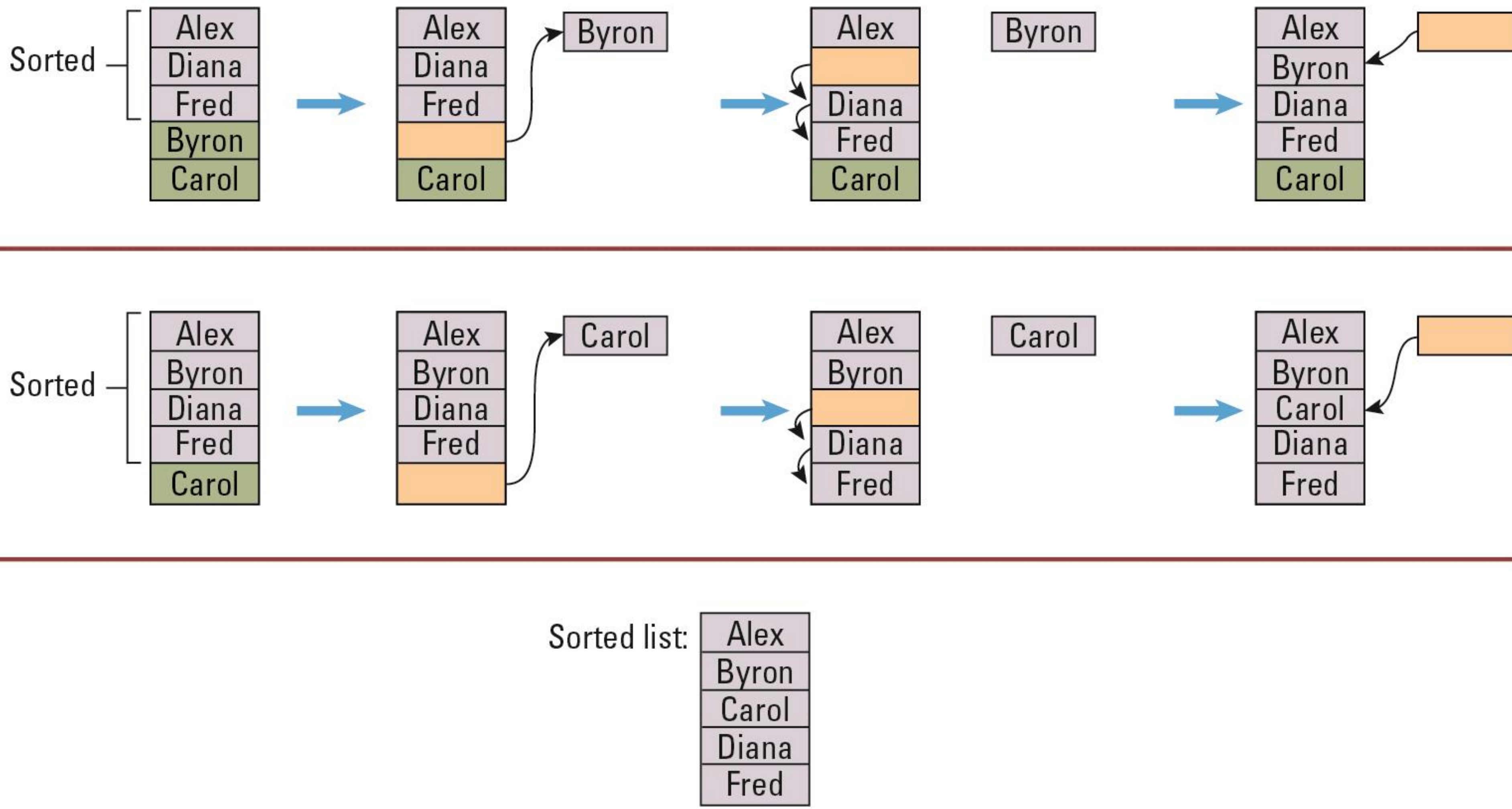
Verification of Insertion Sort

- Loop invariant of the outer loop
  - Each time the test for termination is performed, the name preceding the ***N***-th entry from a sorted list
- Termination condition
  - The value of N is greater than the length of the list
- If the loop terminates, the list is sorted

Recap: Insertion Sort



Recap: Insertion Sort (cont.)



Summary of Software Verification

- Software verification is not easy
- Can be easier with a formal programming language with better properties

- In practice, testing is more commonly used to verify software
  - However, testing only proves that the program is correct for the test cases used

Any Questions?