Algorithms

National Chiao Tung University

Chun-Jen Tsai

04/13/2012

Algorithm

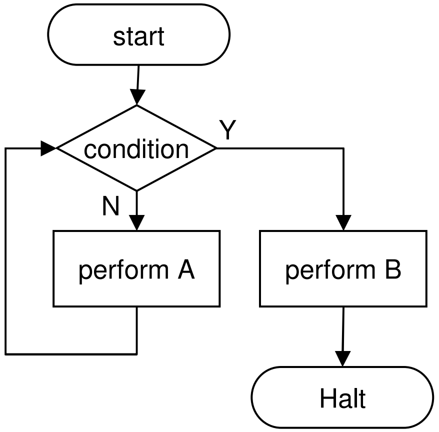
- An algorithm is an **ordered set** of **unambiguous**, **executable steps** that defines a *terminating* process
  - "Ordered" does not imply "sequential" → there are parallel algorithms
- An algorithm is an abstract concept
  - There can be several physical implementations of an algorithm (using same or different programming languages)
  - Given same input, different implementations of an algorithm should produce the same output
  - For example, an algorithm is like a story, and an implementation is like a book on the story

Terminology Clarification

- What is the differences between an algorithm, a program, and a process?
  - Read the last paragraph of Sec 5.1 of the textbook carefully!
  - A program is a formal representation of an algorithm, which can be executed by a computer
  - A process is the activity of executing an algorithm (or equivalently, a program)

Algorithm Representation (1/2)

- ❑ We must find a way to separate an algorithm from the actual implementation, but still present it precisely
- ❑ In 1950's~1970's, flowcharts is very popular in describing an algorithm
  → too cumbersome for
  sophisticated design



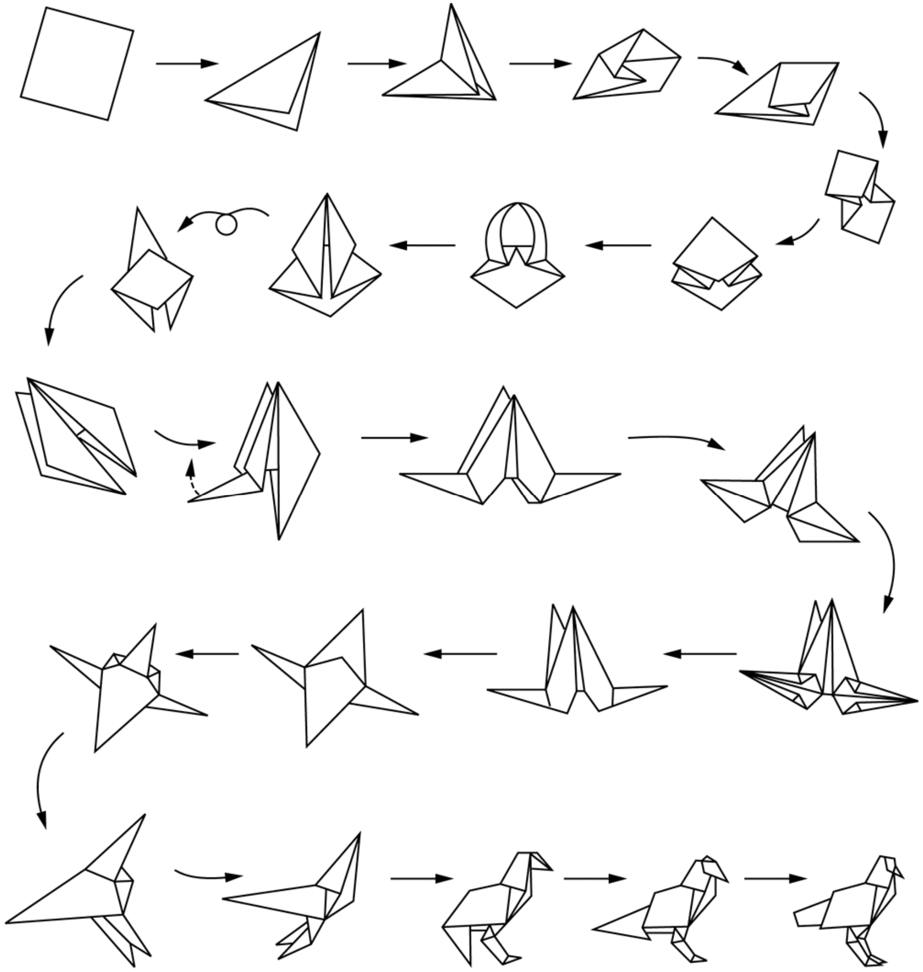
- ❑ Today, we usually use a *pseudocode* language (not a programming language) to describe an algorithm

Algorithm Representation (2/2)

- ❑ Theoretically, any languages (e.g. English) can be used as a pseudocode language to describe algorithms
- ❑ In practice, a good pseudocode language must avoid any ambiguities
  - ■ A language with a small set of well-defined building blocks (called primitives) can removes ambiguity
  - ■ Each primitive is a single operation you can apply to the data

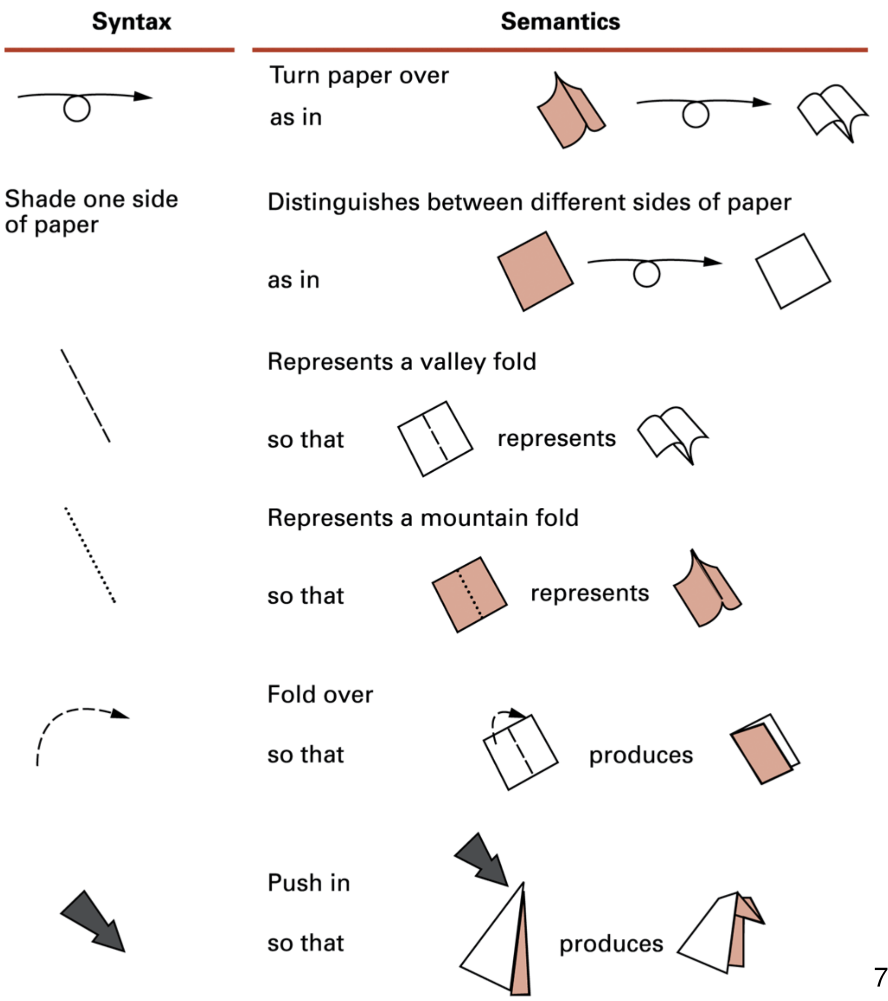
Example: Origami

- ❑ Origami is a complicated procedure
- ❑ Key question: What operations can we perform at each step?



Origami Primitives

- ❑ Primitives defined in our programming language limit our implementation of an algorithm



Syntax

Semantics

Turn paper over as in

Shade one side of paper

Distinguishes between different sides of paper as in

Represents a valley fold so that represents

Represents a mountain fold so that represents

Fold over so that produces

Push in so that produces

Pseudocode Example

❑ Pseudocode primitives

- Assignment: *name* ← *expression*;
- Conditional selection: **if** (*condition*) **then** (*action*)
- Repeated execution: **while** (*condition*) **do** (*action*)
- Procedure: **procedure** *name* (*parameters*)

❑ Example:

```
procedure Greetings
Count ← 3;
while (Count > 0) do
    (
        print the message “Hello”;
        Count ← Count – 1;
    )
```

Algorithm Discovery

- ❑ The techniques to discover an algorithm to solve real world problem often requires specific **domain knowledge** that you won’t learn in Computer Science → to launch a rocket, you must know physics well!
- ❑ However, a big algorithm is usually composed of many small “standard algorithms” which you will learn in Computer Science curriculum
  - ■ For example, “sorting numbers” is a crucial small algorithm for many large problems

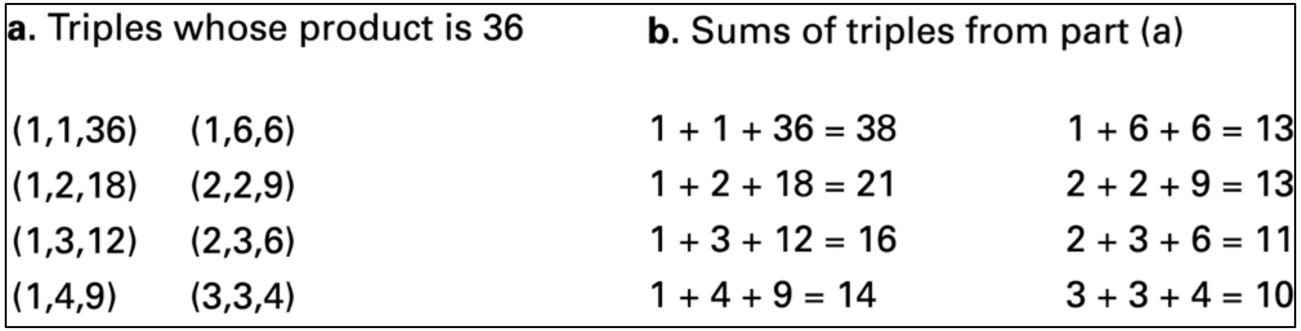
Polya’s Problem Solving Phases

- In 1945, G. Polya defined four problem solving phases
  - Phase 1: Understand the problem
  - Phase 2: Devise a plan for solving the problem
  - Phase 3: Carry out the plan
  - Phase 4: Evaluate the solution for accuracy and for its potential as a tool for solving other problems

A Sample Problem

- ❑ Person A is charged with the task of determining the ages of B’s three children.
  - B tells A that the product of the children’s ages is 36.
  - A replies that another clue is required.
  - B tells A the sum of the children’s ages.
  - A replies that another clue is needed.
  - B tells A that the oldest child plays the piano.
  - A tells B the ages of the three children.

How old are the three children?



a. Triples whose product is 36		**b.** Sums of triples from part (a)	
(1,1,36)	(1,6,6)	1 + 1 + 36 = 38	1 + 6 + 6 = 13
(1,2,18)	(2,2,9)	1 + 2 + 18 = 21	2 + 2 + 9 = 13
(1,3,12)	(2,3,6)	1 + 3 + 12 = 16	2 + 3 + 6 = 11
(1,4,9)	(3,3,4)	1 + 4 + 9 = 14	3 + 3 + 4 = 10

Techniques for Problem Solving

- Work the problem backwards (reverse-engineering)
- Look for solutions of an easier, related problem
- Stepwise refinement (top-down methodology)
  - Popular technique because it produces modular programs
- Breaking a big problem into smaller ones (bottom-up methodology)
  - The solution of each small problem must be unit-tested before integrating it into the big solution

Iterative Structures in Algorithms

- ❑ It often happens that an algorithm contains repeated actions, each action is similar to previous one
- ❑ Example: sequential search

```
procedure Search (List, TargetValue)
if (List empty) then
    ( Declare search a failure; )
else
  (
     Select the first entry in List to be TestEntry;
     while (TargetValue > TestEntry and
            there remain entries to be considered)
     do ( Select the next entry in List as TestEntry; )
     if (TargetValue = TestEntry) then
         ( Declare search a success; )
     else
         ( Declare search a failure; )
   )
```

Two Variants of Iterative Structure

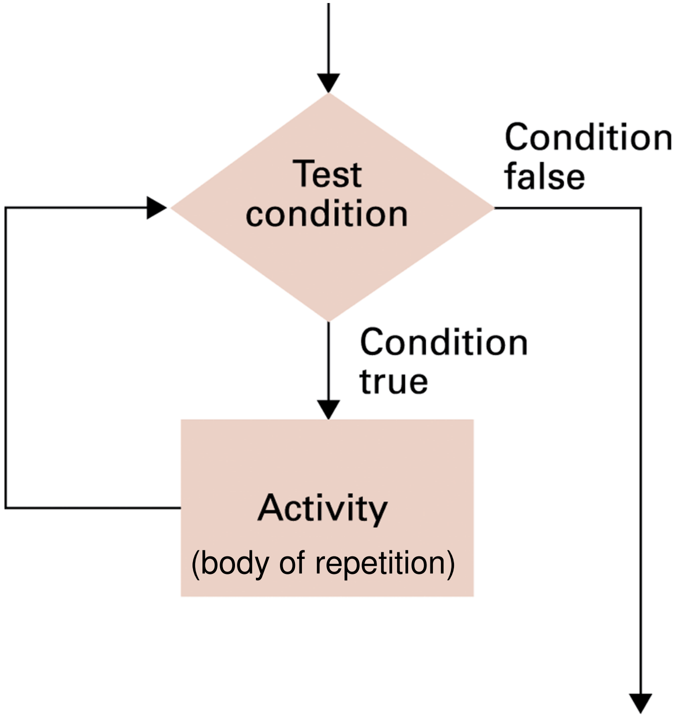
- Iterative structure can be implemented using one of two methods
  - Loop structure
  - Recursive structure
- An iterative structure is composed of two parts
  - Body of repetition
  - Repetitive control

Components of Repetitive Control

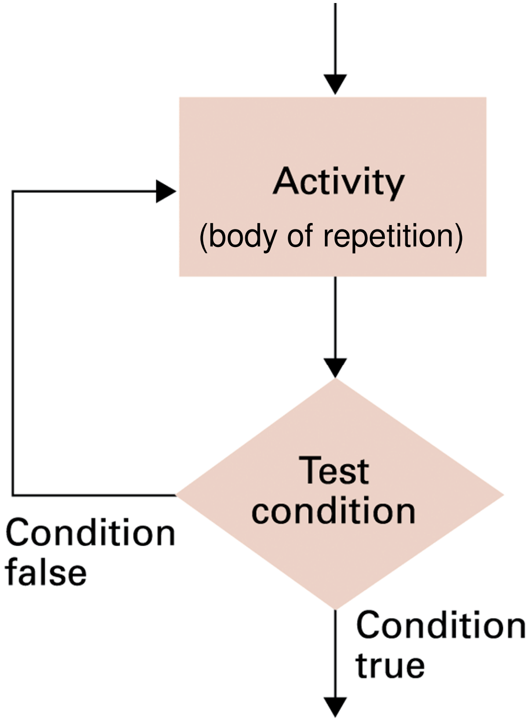
- Initialize:
  - Establish an initial state that will be modified toward the ***termination condition***
- Test:
  - Compare the current state to the termination condition and terminate the repetition if equal
- Modify:
  - Change the state in such a way that it moves toward the termination condition

Loop Structure

The while loop:

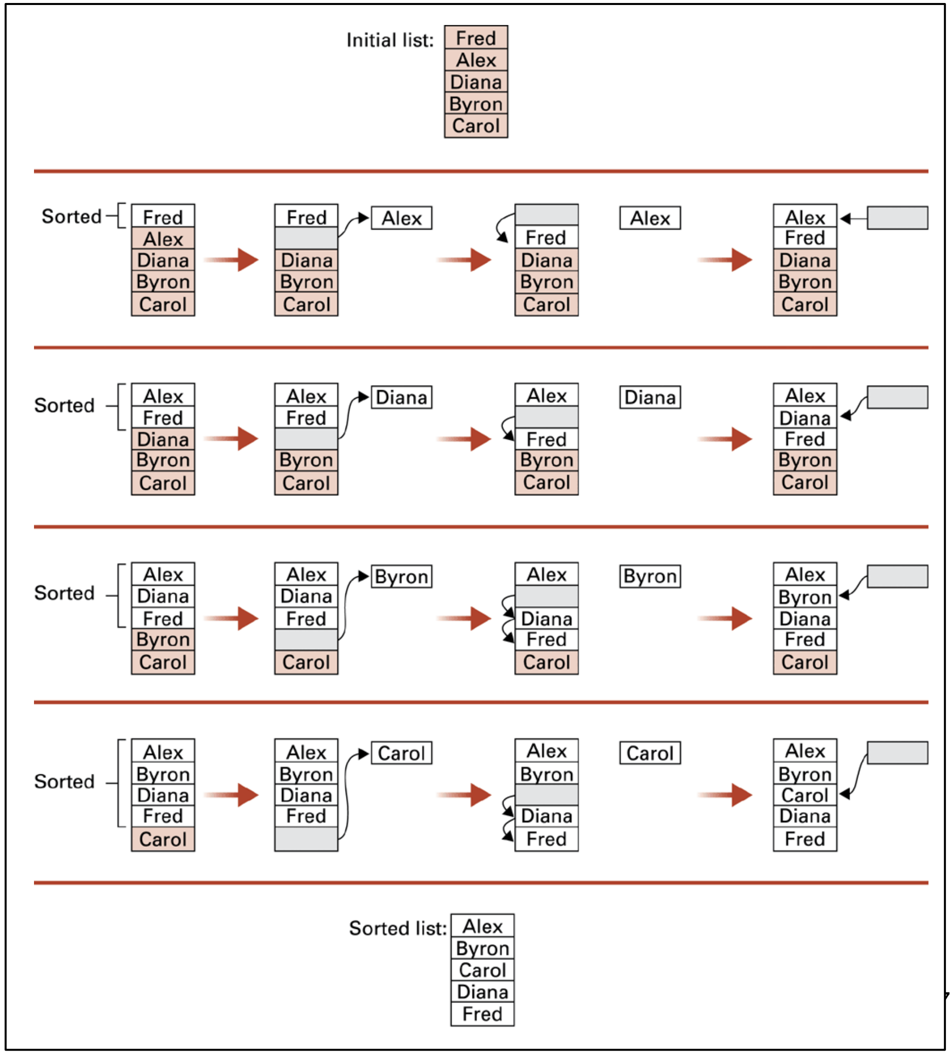


The repeat loop:



Example: Sorting (1/2)

- ❑ Sorting the list: Fred, Alex, Diana, Byron, and Carol alphabetically
- ❑ The example given here uses ***insertion sort***



Initial list: Fred Alex Diana Byron Carol

Sorted: Fred | Alex Diana Byron Carol → Fred, Alex, Diana Byron Carol → Fred Diana Byron Carol, Alex → Alex Fred Diana Byron Carol

Sorted: Alex Fred | Diana Byron Carol → Alex Fred, Diana, Byron Carol → Alex Fred Byron Carol, Diana → Alex Diana Fred Byron Carol

Sorted: Alex Diana Fred | Byron Carol → Alex Diana Fred, Byron, Carol → Alex Diana Fred Carol, Byron → Alex Byron Diana Fred Carol

Sorted: Alex Byron Diana Fred | Carol → Alex Byron Diana Fred, Carol → Alex Byron Diana Fred, Carol → Alex Byron Carol Diana Fred

Sorted list: Alex Byron Carol Diana Fred

Example: Sorting (2/2)

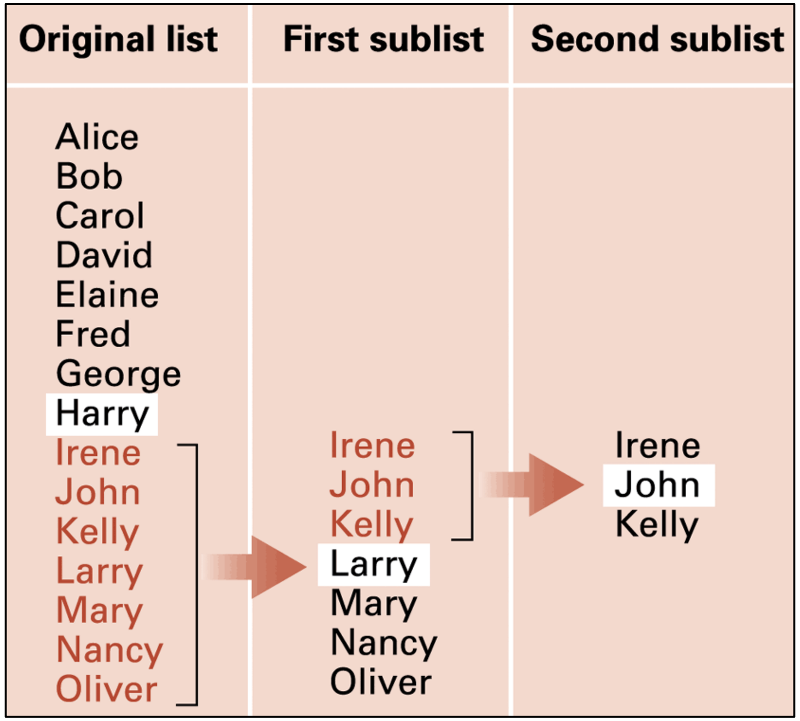
- The pseudo code of insertion sort

```
procedure Sort(List)
N ← 2;
while (the value of N does not exceed the length of List) do
  (
     Select the Nth entry in List as the pivot entry;
     Move the pivot entry to a temporary location, leaving a hole;
     while (there is a name above the hole 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

- ❑ Recursive structures provides an alternative to the iterative structures
- ❑ Example: binary search

Goal: find John



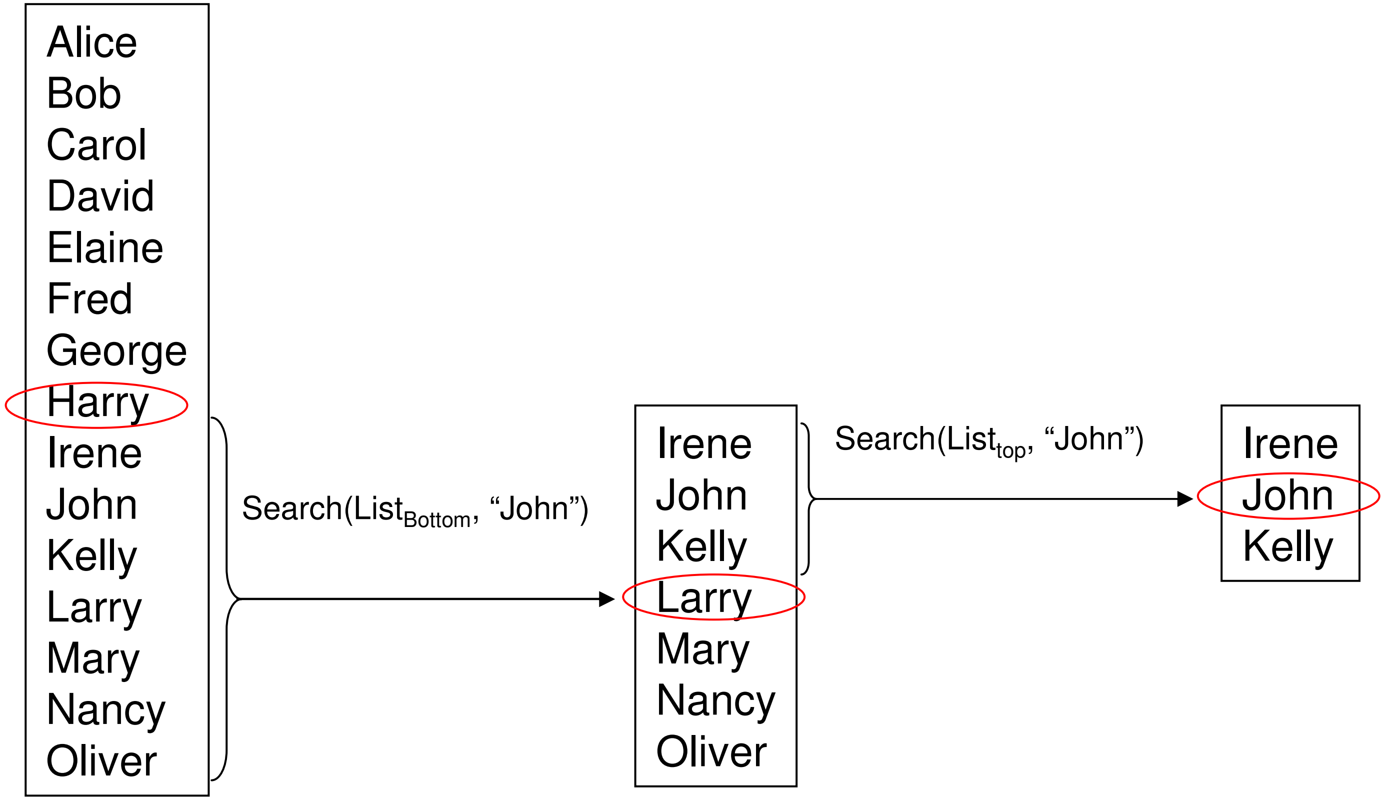
Original list	First sublist	Second sublist
Alice		
Bob		
Carol		
David		
Elaine		
Fred		
George		
Harry		
Irene	Irene	Irene
John	John	John
Kelly	Kelly	Kelly
Larry	Larry	
Mary	Mary	
Nancy	Nancy	
Oliver	Oliver	

Pseudo Code of Binary Search (1/2)

- ❑ The pseudo code of binary search can be described elegantly using recursive structure:

```
procedure Search(List, TargetValue)
if (List empty) then
    ( Report that the search failed; )
else
  (
     Select the middle entry in List to be the first TestEntry;
     if (TargetValue = TestEntry) then
        ( Report that the search succeeded; )
     else
       (
            if (TargetValue < TestEntry) then
                ( Search($List_{top}$, TargetValue); )
            else
                ( Search($List_{bottom}$, TargetValue); )
       )
   )
```

Pseudo Code of Binary Search (2/2)



Recursive Control

- Note that, in a recursive structure, repetitive control is achieved by first testing the termination condition before recursive calls

```
procedure my_function(parameters)
if (termination condition is true) then
   (
         Ends body of repetition
   )
else
   (
         Perform recursive calls
   )
```

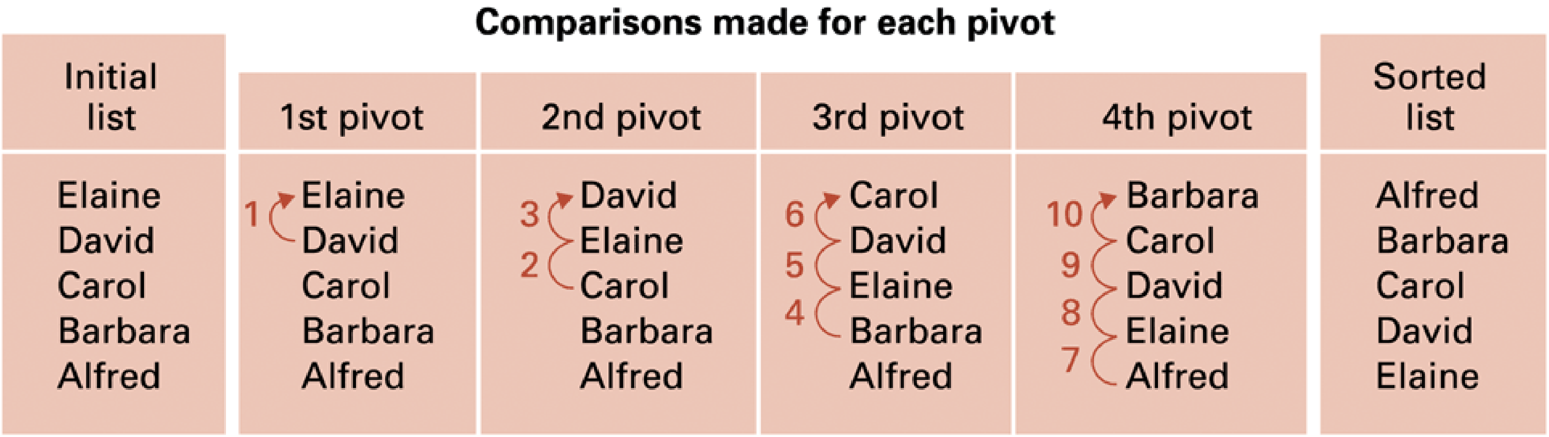
Algorithm Efficiency

- ❑ Question: when the number of input data grows linearly, what is the growth of the number of operations of an algorithm?
  → The answer to this question is the complexity of the algorithm

- ❑ Since the number of operations for processing a set of data depends on both the size of the data and the data pattern, an algorithm’s best case, worst case, and average case complexity can be quite different

Example: Insertion Sort (1/2)

❑ Applying the insertion sort in a worst-case situation



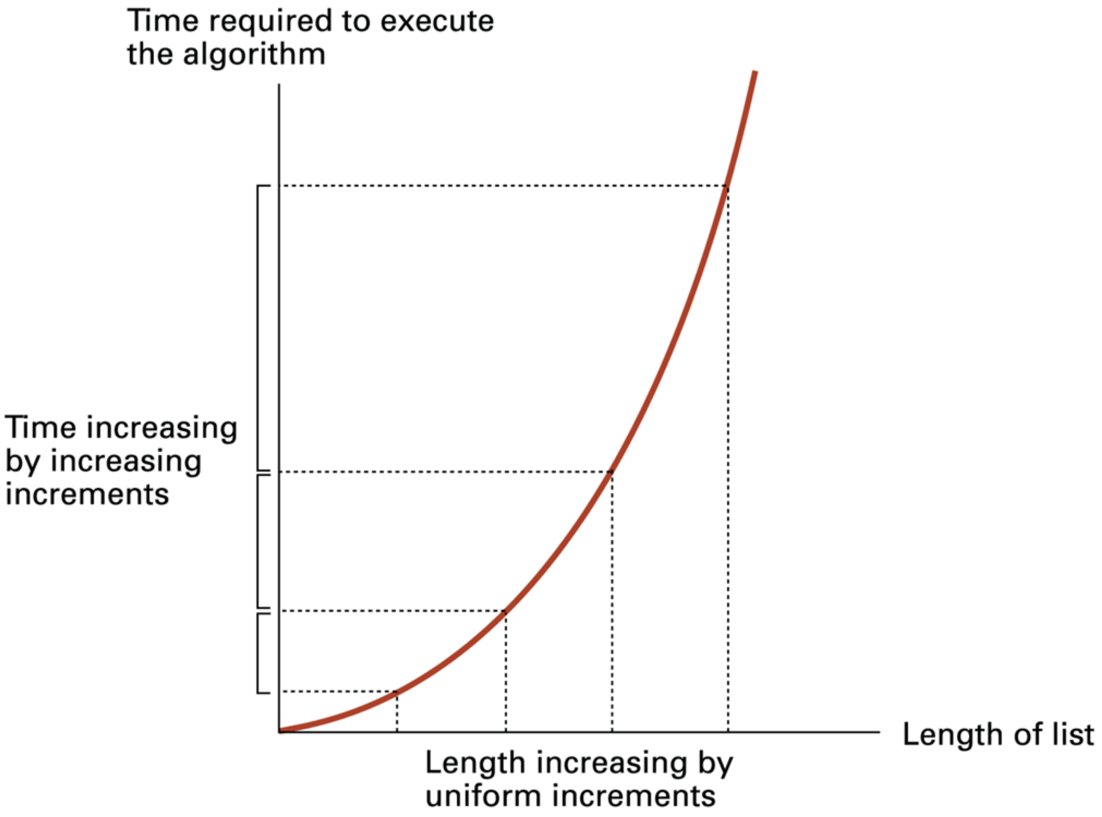
Comparisons made for each pivot

Initial list	1st pivot	2nd pivot	3rd pivot	4th pivot	Sorted list
Elaine	Elaine	David	Carol	Barbara	Alfred
David	David	Elaine	David	Carol	Barbara
Carol	Carol	Carol	Elaine	David	Carol
Barbara	Barbara	Barbara	Barbara	Elaine	David
Alfred	Alfred	Alfred	Alfred	Alfred	Elaine

1
3
2
6
5
4
10
9
8
7

Example: Insertion Sort (2/2)

- ❑ Graph of the worst-case analysis of the insertion sort algorithm



Complexity Measure

- The complexity of an algorithm can be classified using the bit-theta notation, Θ, of the input data size n
- $\Theta(f(n))$ means that if the input data size is n, the number of operations, $c(n)$, of the algorithm grows at the same speed of $f(n)$, within constant factors. That is, given constants $k_1 \geq k_2 > 0$, as $n \longrightarrow \infty$,

$$k_2 \cdot f(n) \leq c(n) \leq k_1 \cdot f(n)$$

- For example, the complexity of insertion sort is $\Theta(n^2)$ and the complexity of binary search is $\Theta(\log_2 n)$

Software Verification

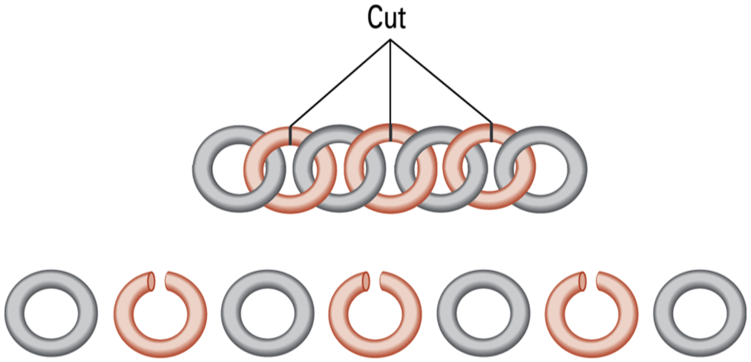
- ❑ Once we have designed an algorithm, the next logical question to ask is:

  How do I prove the correctness of the algorithm?

- ❑ Software verification’s goal is to prove that an algorithm works by a ***formal procedure*** instead of ***intuitive arguments***

Example: Chain Separating (1/3)

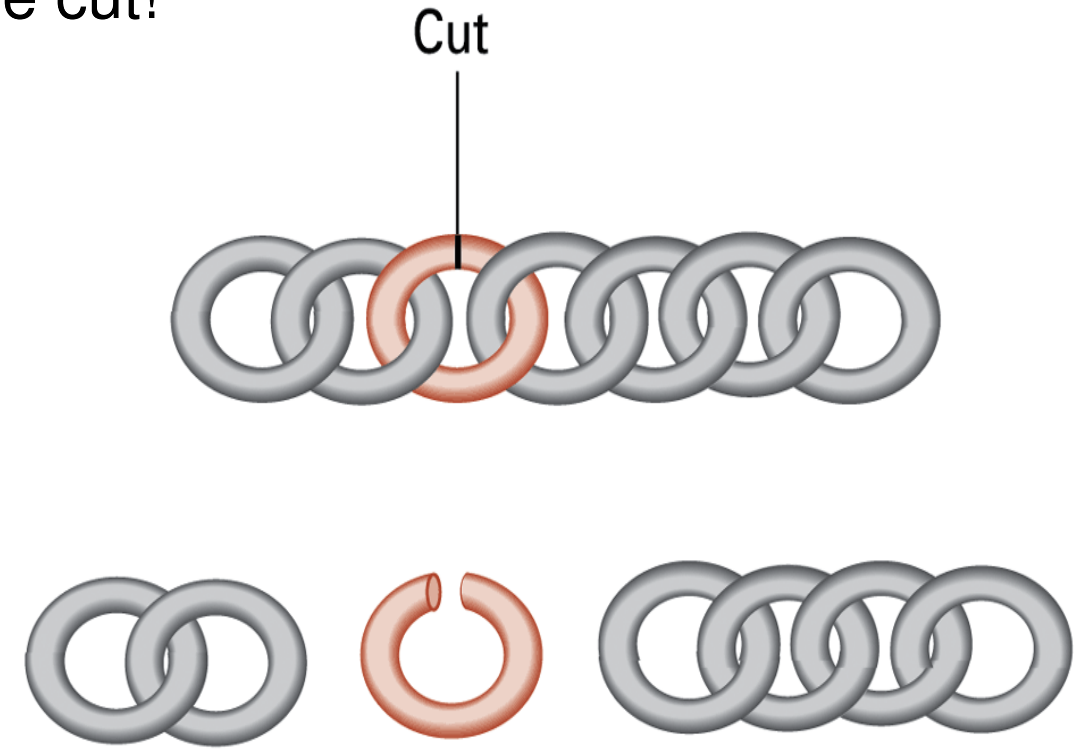
- A traveler has a gold chain of seven links. He must stay at an isolated hotel for seven nights. The rent each night consists of one link from the chain.
  - What is the fewest number of links that must be cut so that the traveler can pay the hotel one link of the chain each morning without paying for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



  - Key question: is this the correct answer?

Example: Chain Separating (2/3)

- In fact, the problem can be solved by using one single cut!



Example: Chain Separating (3/3)

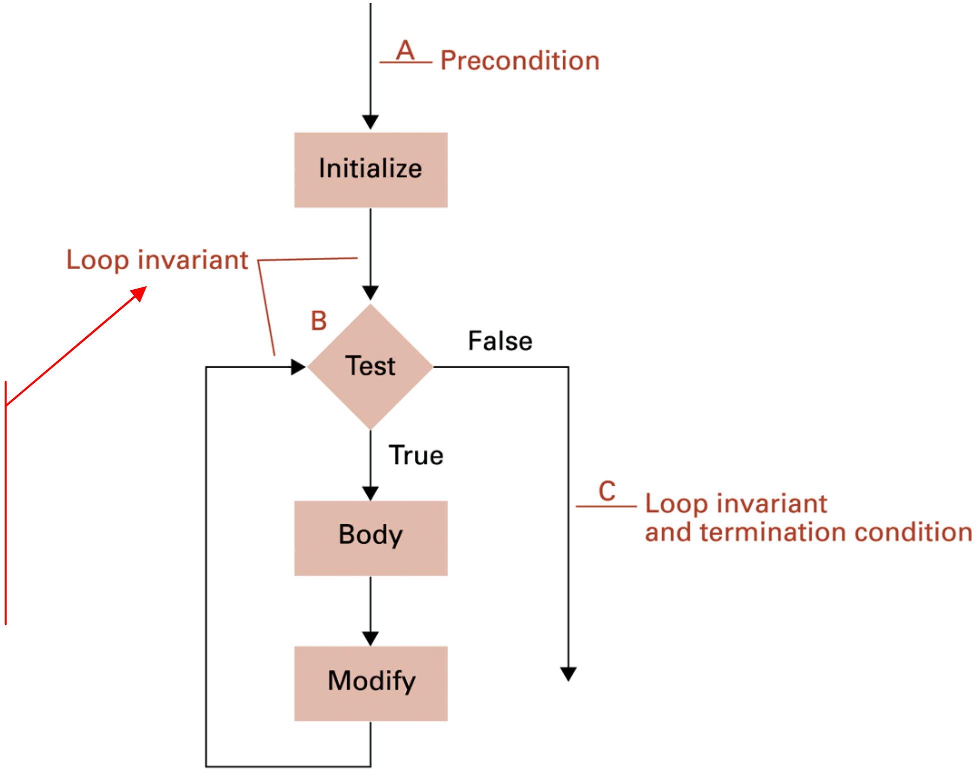
- ❑ How do we prove that the 2nd answer is optimal?
- ❑ Proof:
  - On the first morning, a single link must be given to the hotel; at least one cut is needed
  - Since our solution uses only one cut and there can be no other solution using less than one cut, we have found the optimal solution

Software Verification

- Proof of correctness of algorithm is composed of several steps
    - First, some conditions are true before the execution of the algorithm; these conditions are called preconditions
    - Then, some statements, called assertions, throughout the algorithm must be established
    - At the end, an assertion must specify the desired output of the algorithm
    - If, given the preconditions, each identified assertion is true when the execution reaches that particular point, then the algorithm is correct

Example of Assertions

- ❑ An assertion at a point in a loop that is true every time that point in the loop is reached is known as ***loop invariant***



Loop invariant for the insertion sort:

"Every time we reach here, the first N-1 position are sorted"

Software Testing

- ❑ Formal algorithm verification techniques have not been powerful enough to apply to general algorithms
- ❑ Most programs today are “verified” by testing under various conditions (called ***test points***), nevertheless, there is no guarantee that a “tested” program is correct under any circumstances