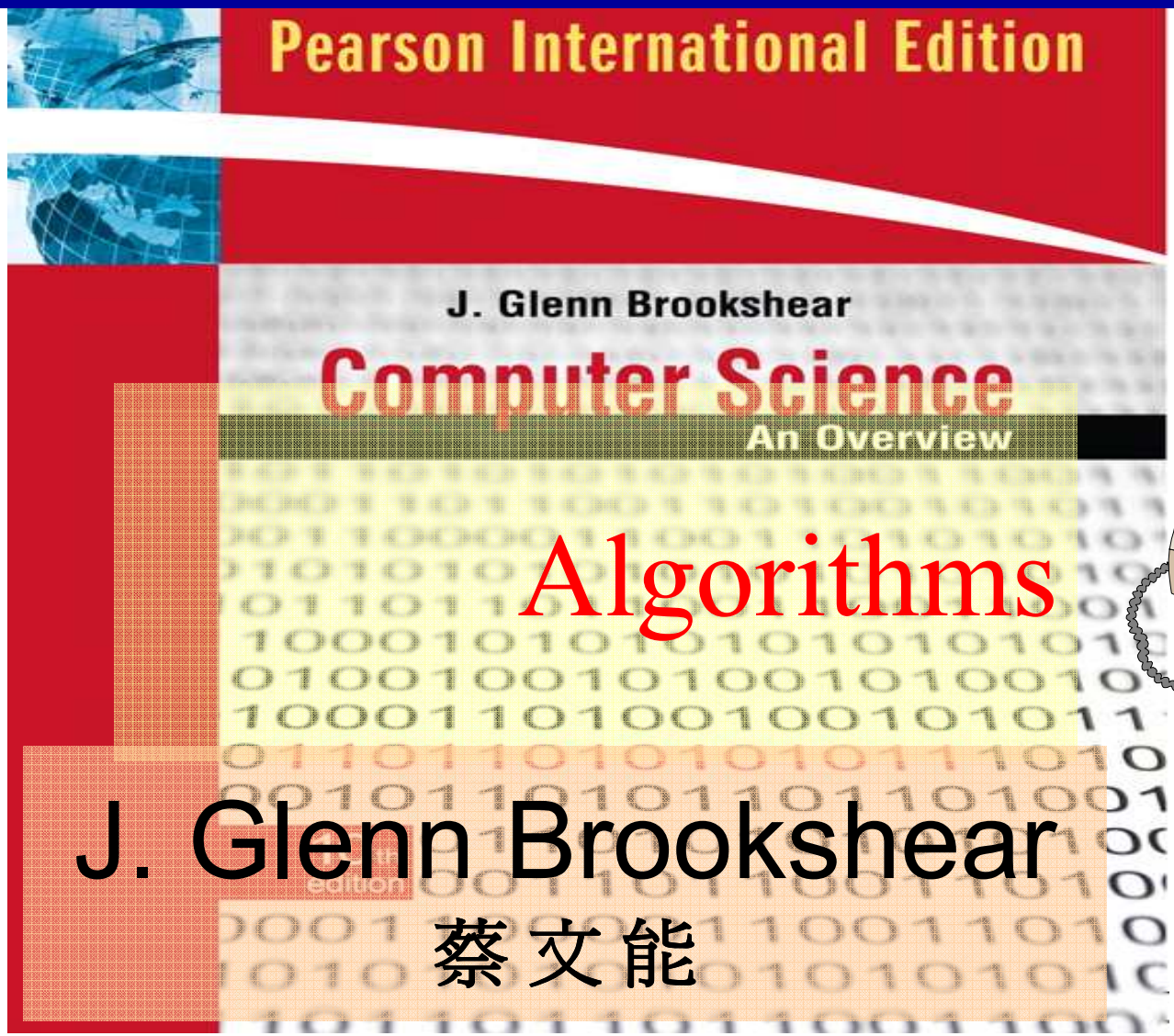
Chapter 5

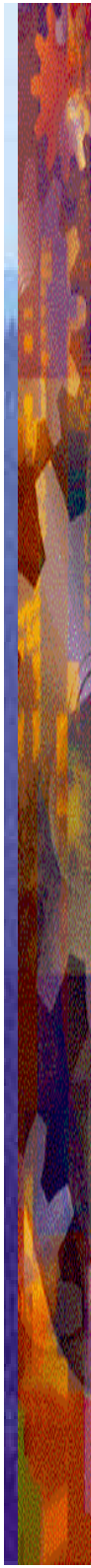


Algorithms

J. Glenn Brookshear
蔡文能

Chapter 5: Algorithms

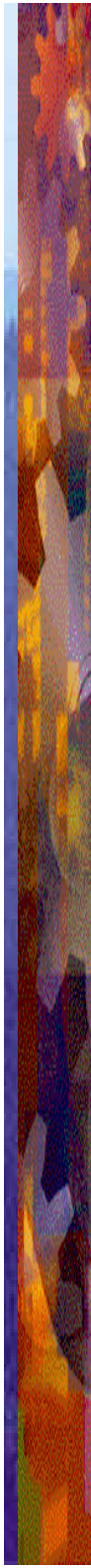
5.1 The Concept of an Algorithm
5.2 Algorithm Representation
5.3 Algorithm Discovery
5.4 Iterative Structures
5.5 Recursive Structures
5.6 Efficiency and Correctness



Definition

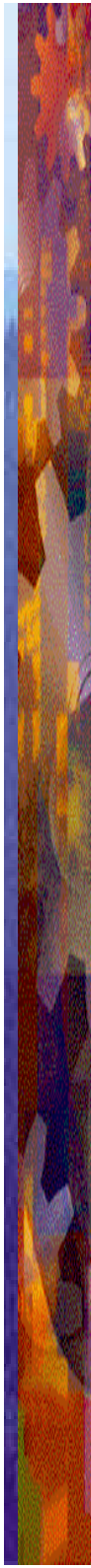
- An algorithm is an **ordered set** of unambiguous, executable steps that defines a terminating process.

- **Program**
  - Formal representation of an **algorithm**
- **Process**
  - Activity of executing a program



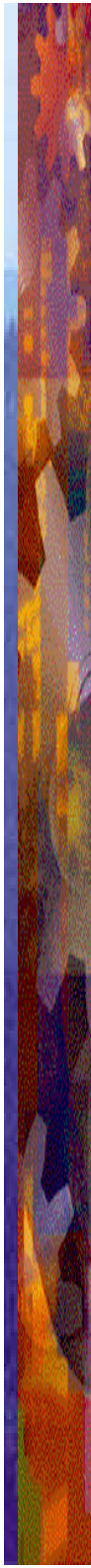
Ordered Set

- Steps in an algorithm must have a well-established structure in terms of the order in which its steps are executed
- May involve more than one thread (parallel algorithms)
- Steps must be an executable instruction
  - Example: Making a list of all the positive integers is not an executable instruction



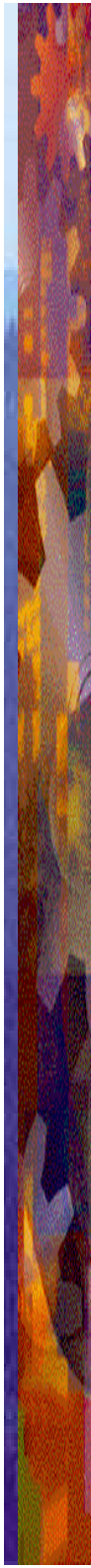
Unambiguous Steps

- During execution of an algorithm, the information in the state of the process must be sufficient to determine uniquely and completely the actions required by each step
- The execution of each step in an algorithm does not require creative skills. Rather, it requires only the ability to follow directions.



Terminating Process

- All execution of an algorithm must lead to an end.
- Computer science seeks to distinguish problems whose answers can be obtained algorithmically and problems whose answers lie beyond the capabilities of algorithmic systems
- There are, however, many meaningful application for non-terminating processes



Chapter 5: Algorithms

5.1 The Concept of an Algorithm

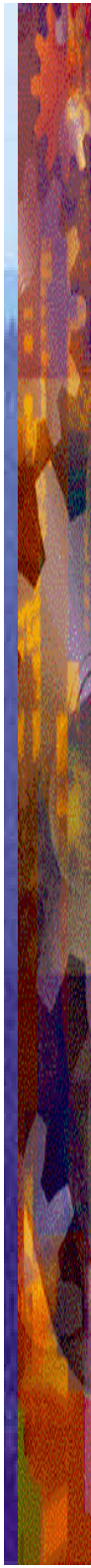
5.2 Algorithm Representation

5.3 Algorithm Discovery

5.4 Iterative Structures

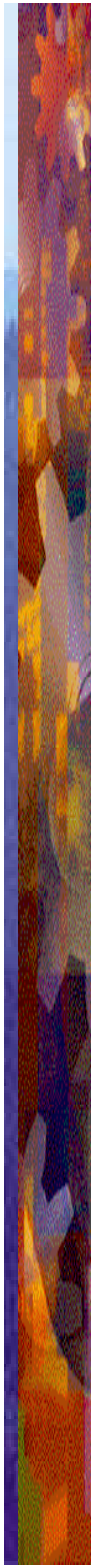
5.5 Recursive Structures

5.6 Efficiency and Correctness



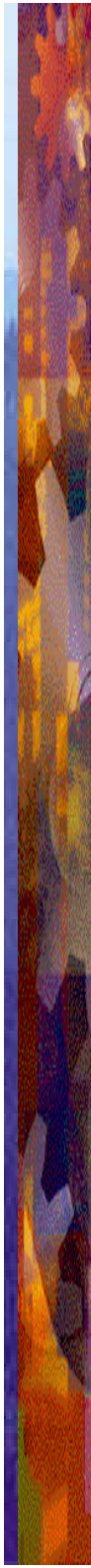
Algorithm and Its Representation

- Like a story and a story book
- Example: converting temperature readings from Celsius to Fahrenheit
  - F = (9/5)C + 32
  - Multiply the temperature reading in Celsius by 9/5 and then add 32 to the product
  - Implemented by electronic circuit
  - Underlying algorithm is the same, only the representation differ

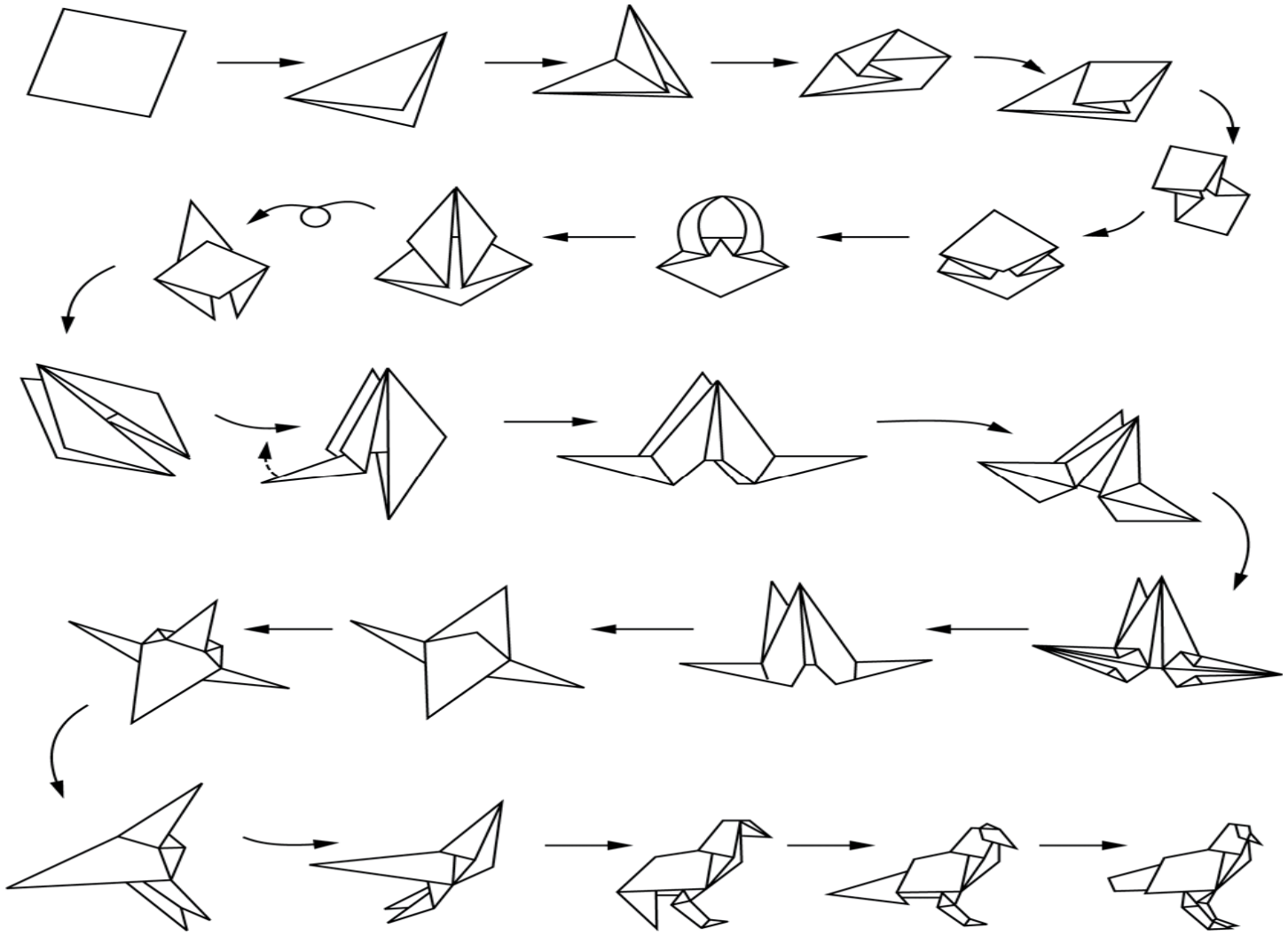


Level of Details

- May cause problems in communicating algorithms
- Example:
  - "Convert the Celsius reading to its Fahrenheit equivalent" suffices among meteorologists
  - But a layperson would argue that this instruction is ambiguous
  - The problem is that the algorithm is not represented in enough detail for the layperson

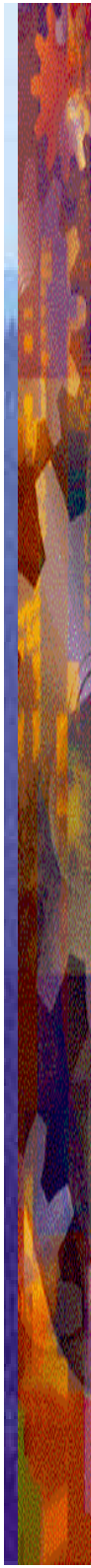


An Example: Origami



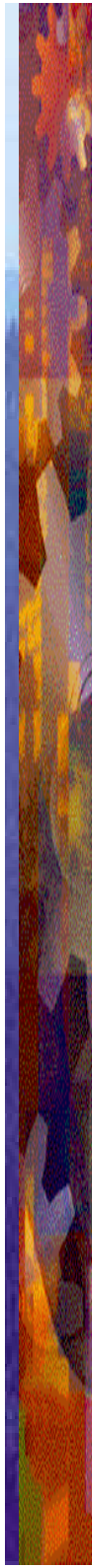
Algorithm Representation

- Primitive
  - Set of building blocks from which algorithm representations can be constructed
- Programming language
  - Collection of primitives
  - Collection of rules stating how the primitives can be combined to represent more complex ideas



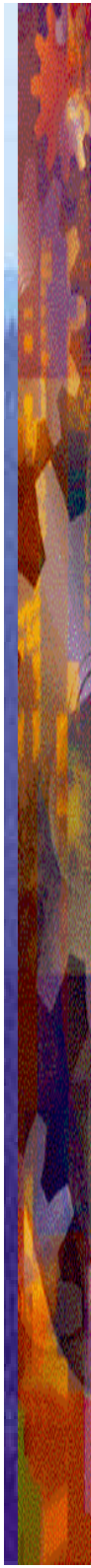
Primitives

- Syntax
  - Symbolic representation
- Semantics
  - Concept represented (meaning of the primitive)
- Levels of abstraction

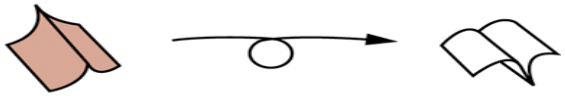
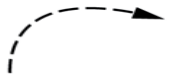


Levels of Abstraction

- Problem
  - Motivation for algorithm
- Algorithm
  - Procedure to solve the problem
  - Often one of many possibilities
- Representation
  - Description of algorithm sufficient to communicate it to the desired audience
  - Always one of many possibilities

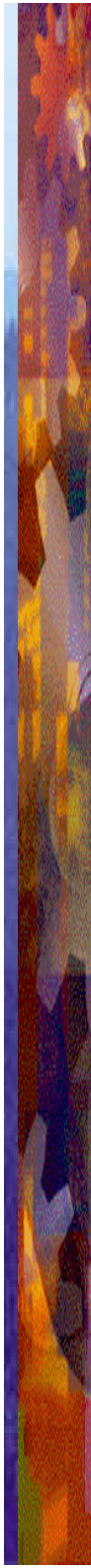


Origami Primitives

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

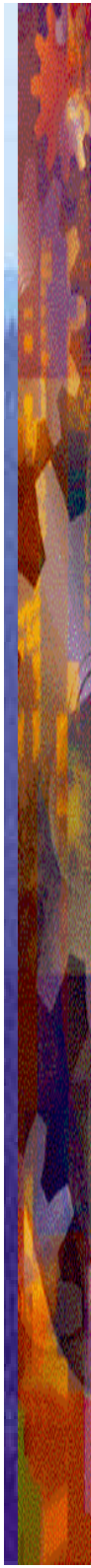
Machine Instructions as Primitives

- Algorithm based on machine instructions is suitable for machine execution
- However, expressing algorithms at this level is tedious
- Normally uses a collection of higher level primitives, each being an abstract tool constructed from the low-level primitives provided in the machine's language – formal programming language



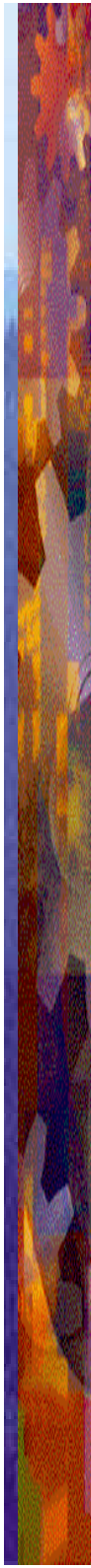
Pseudocode

- Less formal, more intuitive than the formal programming languages
- A notation system in which ideas can be expressed informally during the algorithm development process
- A consistent, concise notation for representing recurring semantic structure
- Comparison with flow chart



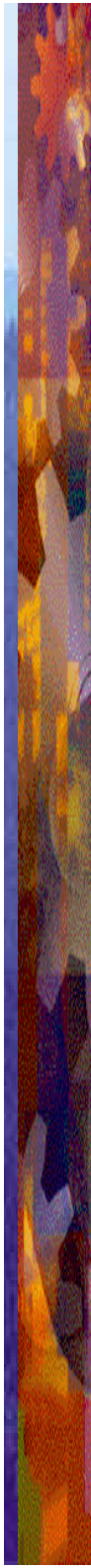
Pseudocode Primitives

- Assignment
  - *name* ← *expression*
- Conditional selection
  - **if** *condition* **then** *action*
- Repeated execution
  - **while** *condition* **do** *activity*
- Procedure
  - **procedure** *name* (*generic names*)



An Example: Greetings

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

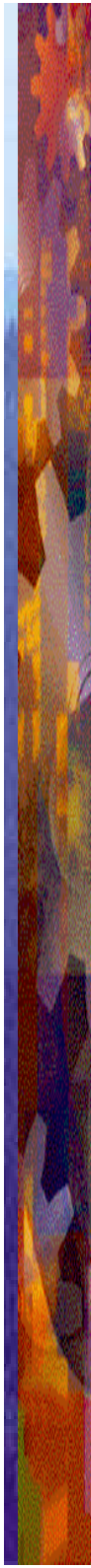


Basic Primitives

- total ← price + tax

- if (sales have decreased)
  then (lower the price by 5%)
- if (year is leap year)
  then (divide total by 366)
  else (divide total by 365)

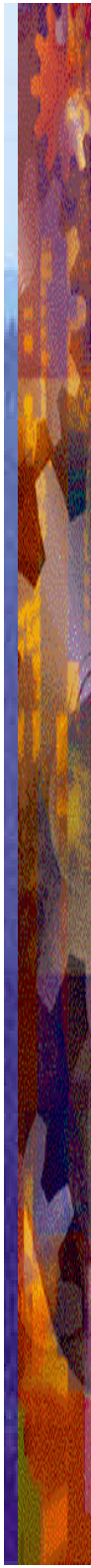
- while(tickets remain to be sold) do
  (sell a ticket)



Procedure Primitive

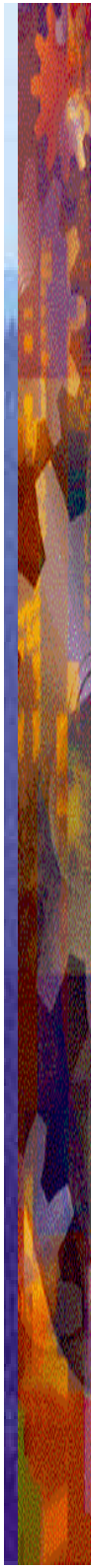
- total ← price + tax

- tax?
- A procedure to calculate tax



Tax as a Procedure

```
Procedure tax
    if (item is taxable)
    then (if (price > limit)
            then (return price*0.1000)
            else (return price*0.0825 )
          )
    else (return 0)
```



Nested Statements

- One statement within another

```
if (item is taxable)
then (if (price > limit)
       then (return price*0.1000)
       else (return price*0.0825 )
      )
else (return 0)
```

Indentations

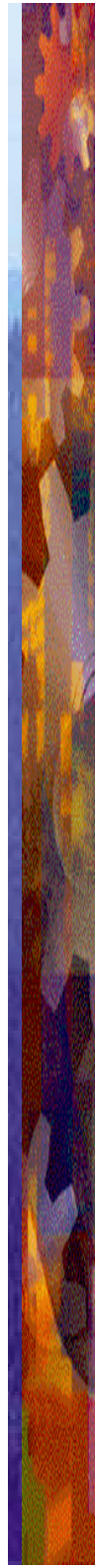
- Easier to tell the levels of nested statements

```
if (item is taxable)
then (if (price > limit)
        then (return price*0.1000)
        else (return price*0.0825 )
      )
else (return 0)
```

Structured Program

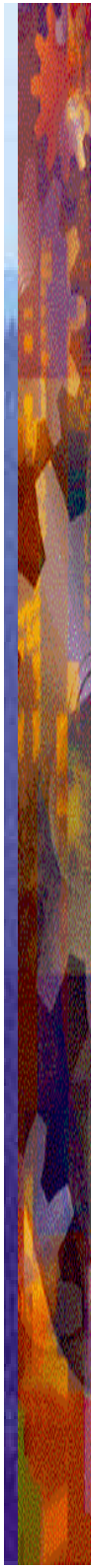
- Divide the long algorithm into smaller tasks
- Write the smaller tasks as procedures
- Call the procedures when needed
- This helps the readers to understand the structure of the algorithm

if (customer credit is good)
then (ProcessLoan)
else (RejectApplication)



The Point of Pseudocode

- To communication the algorithm to the readers
- The algorithm will later turn into program
- Also help the program maintainer or developer to understand the program



Chapter 5: Algorithms

5.1 The Concept of an Algorithm

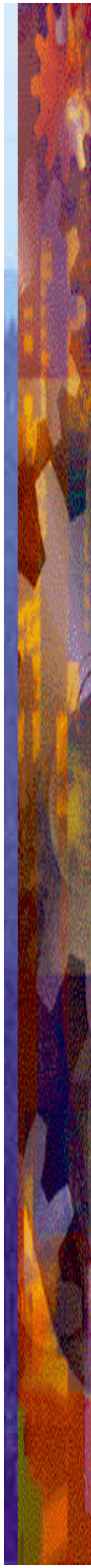
5.2 Algorithm Representation

5.3 Algorithm Discovery

5.4 Iterative Structures

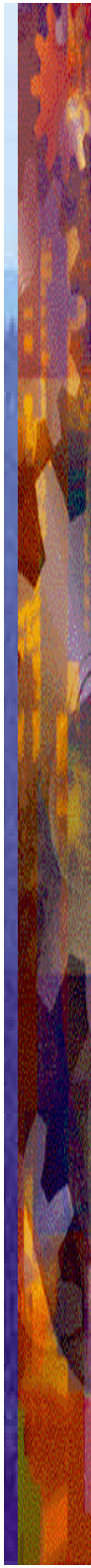
5.5 Recursive Structures

5.6 Efficiency and Correctness



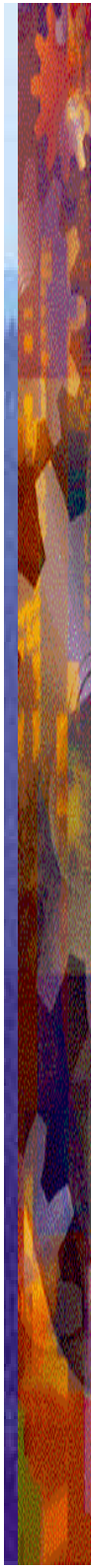
Algorithm Discovery

- Development of a program consists of
  - Discovering the underlying algorithm
  - Representing the algorithm as a program
- Algorithm discovery is usually the more challenging step in the software development process
- Requires finding a method of solving the problem



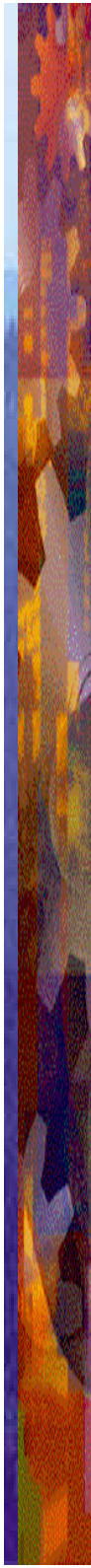
Problem Solving Steps

1. Understand the problem
2. Get an idea
3. Formulate the algorithm and represent it as a program
4. Evaluate the program
    1. For accuracy
    2. For its potential as a tool for solving other problems



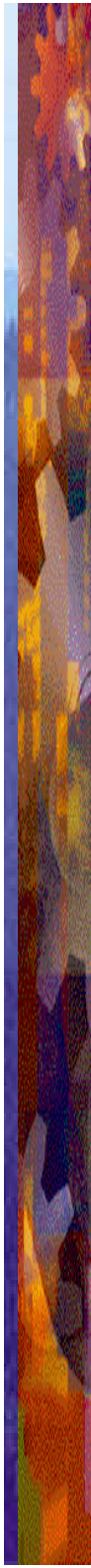
Not Yet Sure What to Do

1. Understand the problem
2. Get an idea
3. Formulate the algorithm and represent it as a program
4. Evaluate the program
    1. For accuracy
    2. For its potential as a tool for solving other problems



Difficulties

- Understanding the problem
  - There are complicated problems and easy problems
  - A complete understanding of the problem before proposing any solutions is somewhat idealistic
- Get an idea
  - Take the ‘Algorithm’ course
  - Mysterious inspiration



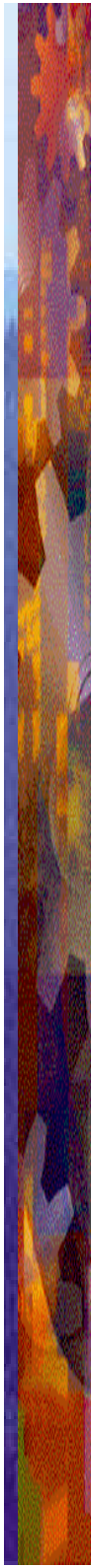
Solving the Problem

a. Triples whose product is 36

(1,1,36) (1,6,6)
(1,2,18) (2,2,9)
(1,3,12) (2,3,6)
(1,4,9) (3,3,4)

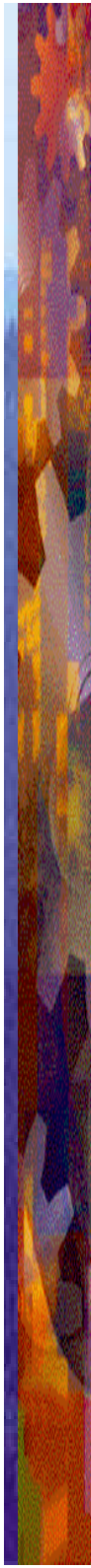
b. Sums of triples from part (a)

1 + 1 + 36 = 38
1 + 2 + 18 = 21
1 + 3 + 12 = 16
1 + 4 + 9 = 14
1 + 6 + 6 = 13
2 + 2 + 9 = 13
2 + 3 + 6 = 11
3 + 3 + 4 = 10



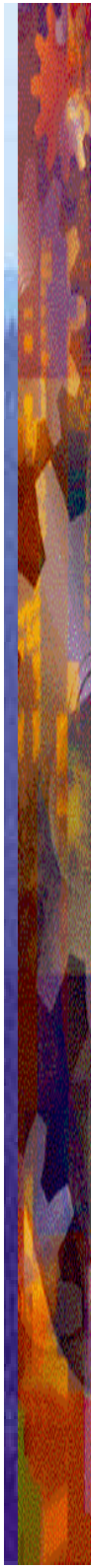
Getting a Foot in the Door

- Work the problem backwards
  - Solve for an example and then generalize
  - Solve an easier related problem
    - Relax some of the problem constraints
- Divide and conquer
  - Stepwise refinement
    - top-down methodology
    - Popular technique because it produces modular programs
  - Solve easy pieces of the problem first
    - bottom up methodology



Work the Problem Backwards

- Simplify the problem
- Build up a scenario for the simplified problem
- Try to solve this scenario
- Generalize the special solution to general scenarios
- Consider a more general problem
- Repeat the process



Divide and Conquer

- Not trying to conquer an entire task at once
- First view the problem at hand in terms of several subproblems
- Approach the overall solution in terms of steps, each of which is easier to solve than the entire original problem
- Steps be decomposed into smaller steps and these smaller steps be broken into still smaller ones until the entire problem has been reduced to a collection of easily solved subproblems
- Solve from the small subproblems and gradually have it all.

Chapter 5: Algorithms

5.1 The Concept of an Algorithm

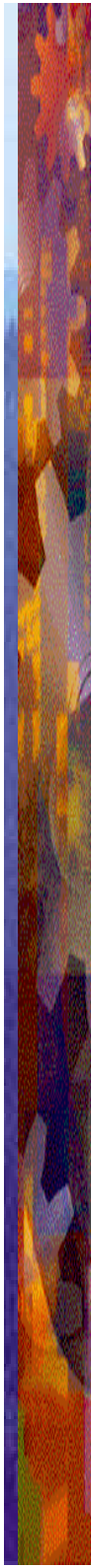
5.2 Algorithm Representation

5.3 Algorithm Discovery

5.4 Iterative Structures

5.5 Recursive Structures

5.6 Efficiency and Correctness



Iterative Structures

- Used in describing algorithmic process
- A collection of instructions is repeated in a looping manner

一直 repeat 到有滿意答案或是無法再改進答案

Example: 用猜測法算 x 的平方根

Example: 用泰勒展開式求 sin(x), cos(x), …

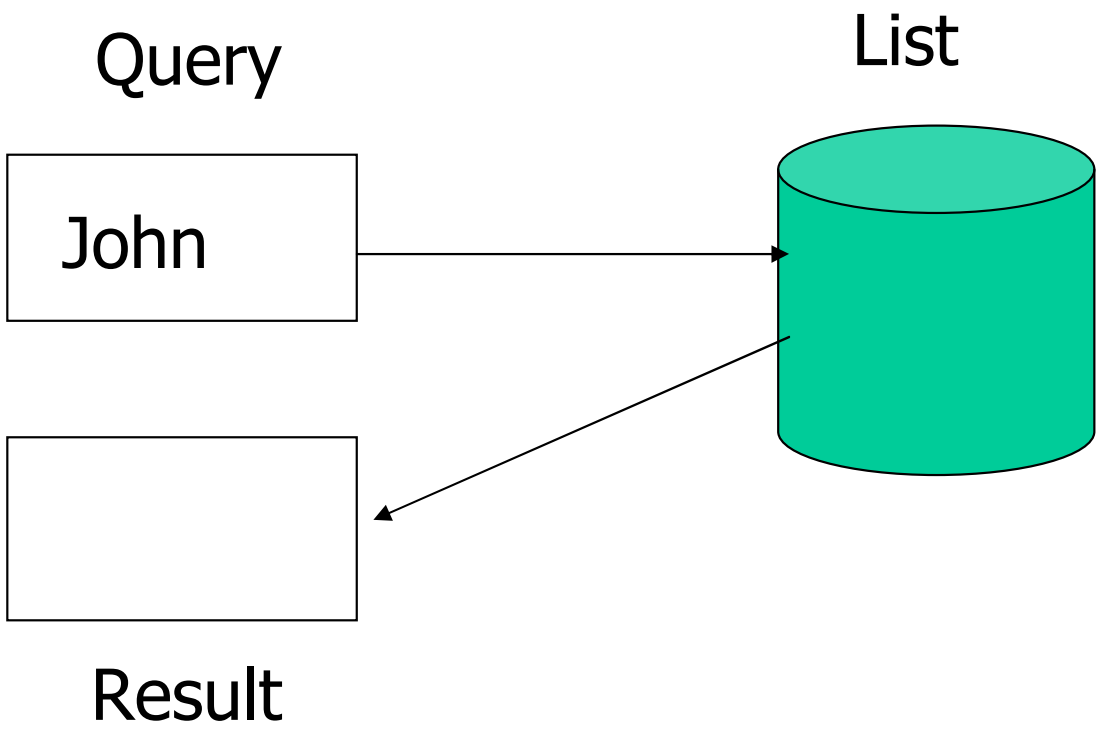
Search Problem

- Search a list for the occurrence of a particular target value
- If the value is in the list, we consider the search a success; otherwise we consider it a failure
- Assume that the list is sorted according to some rule for ordering its entries

Sequential Search vs. Binary Search

Hash ? Store and retrieve data record through hashing?

Search Scenario

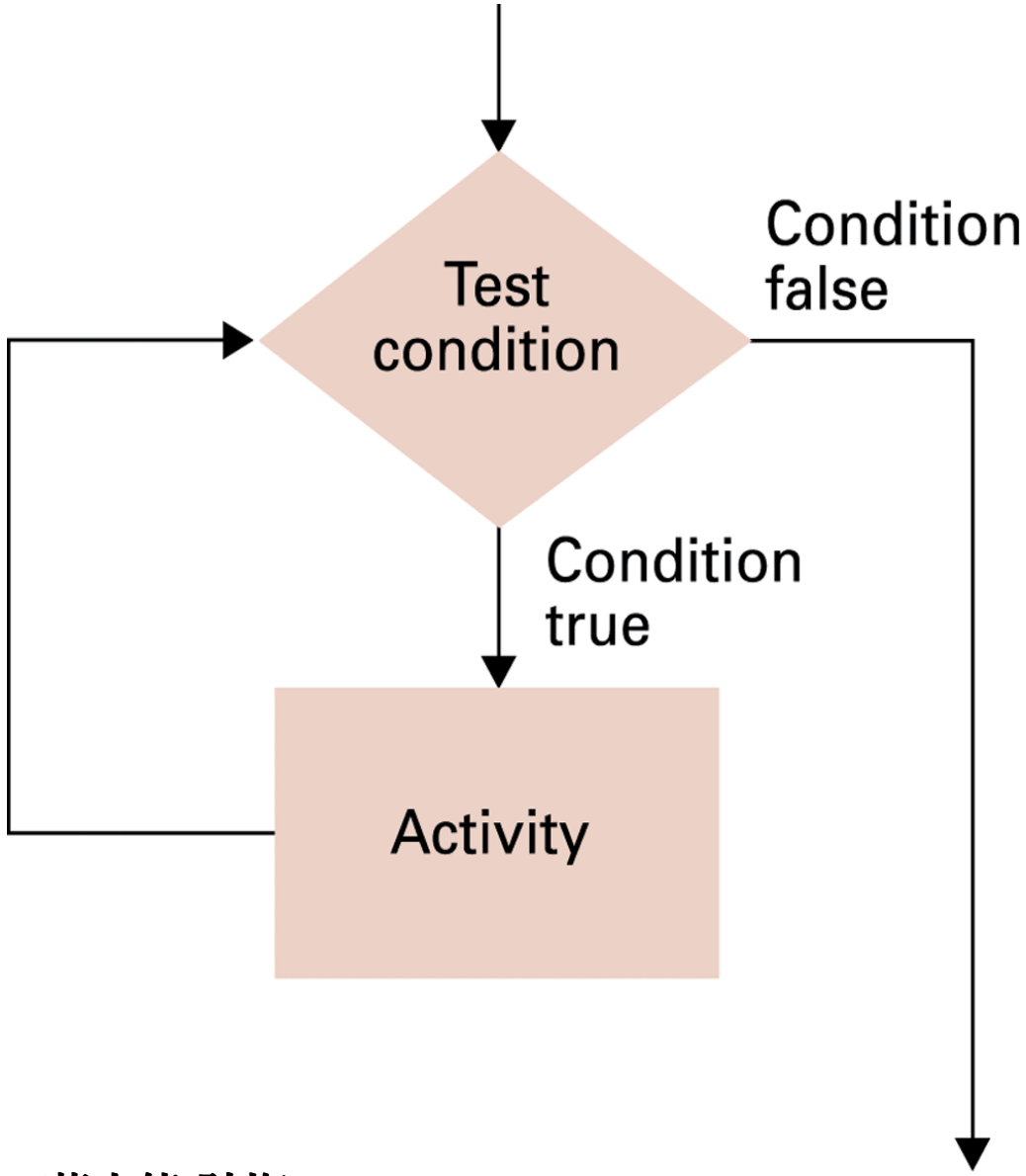


Alice
Bob
Carol
David
Elaine
Fred
George
Harry
Irene
John
Kelly
Larry
Mary
Nancy
Oliver

Sequential Search Algorithm

```
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.)
          ) end if
```

The **while** Loop



C/C++/Java:

```
while(cond) {
    /*…*/
} // while(
```

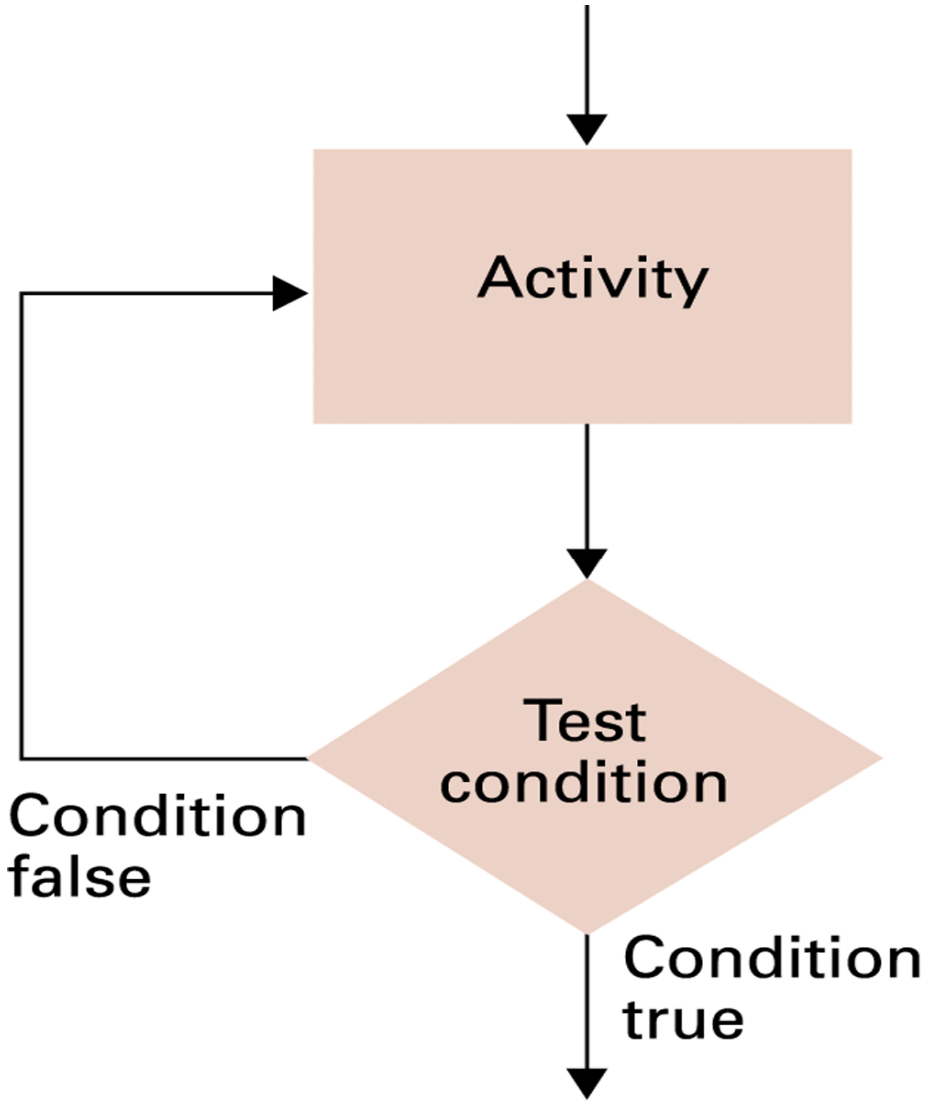
The **repeat** Loop (repeat … until)

C/C++/Java:

```
do {
    /*…*/
 } while(!(cond));
```

Pascal/Delphi:

```
repeat
    /*…*/
until (cond);
```



while vs. repeat Structure

- In **repeat** structure the loop's body is always performed **at least once** (posttest loop) (至少做一次)
  C/C++/Java: do { /*…*/ } while(cond.);

- While in **while** structure, the body is never executed if the termination is satisfied the first time it is tested (pretest loop) (至少做 **0** 次)

Sorting 排序 (排列)

- Take a set of items, order unknown
- Return **ordered** set of the items
  - Ascending order vs. Descending order
- For instance:
  - ➔Sorting **names** alphabetically
  - ➔Sorting by scores in **descending** order
  - ➔Sorting by height in **ascending** order

✓ Issues of interest:

- **Running time in worst case, average/other cases**
- Space requirements (Space complexity)

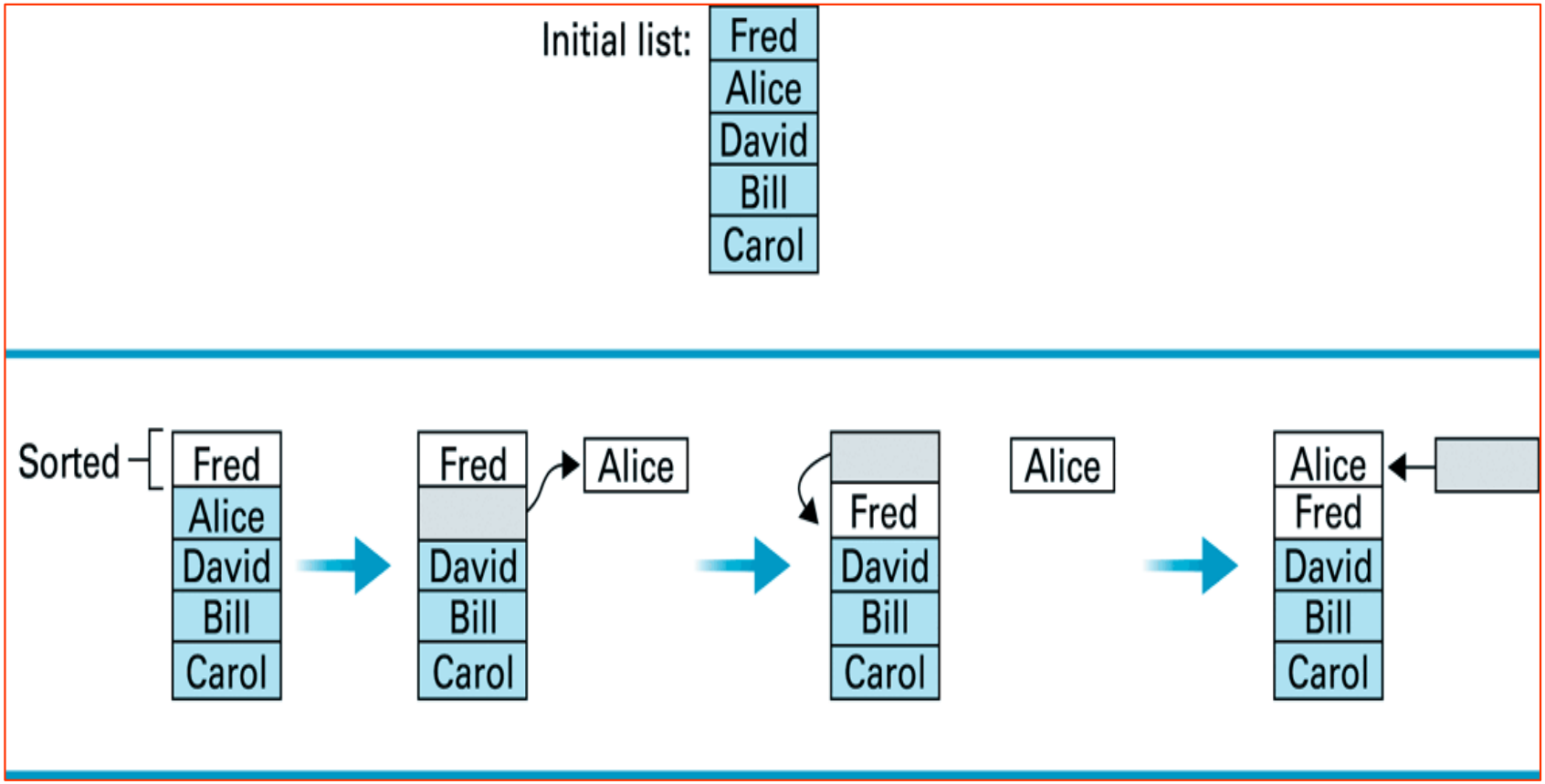
常見簡單 Sorting 技巧

- **Insertion Sort 插入排列法**
- Selection Sort 選擇排列法
- **Bubble** Sort 氣泡排列法
  (Sibling exchange sort; 鄰近比較交換法)

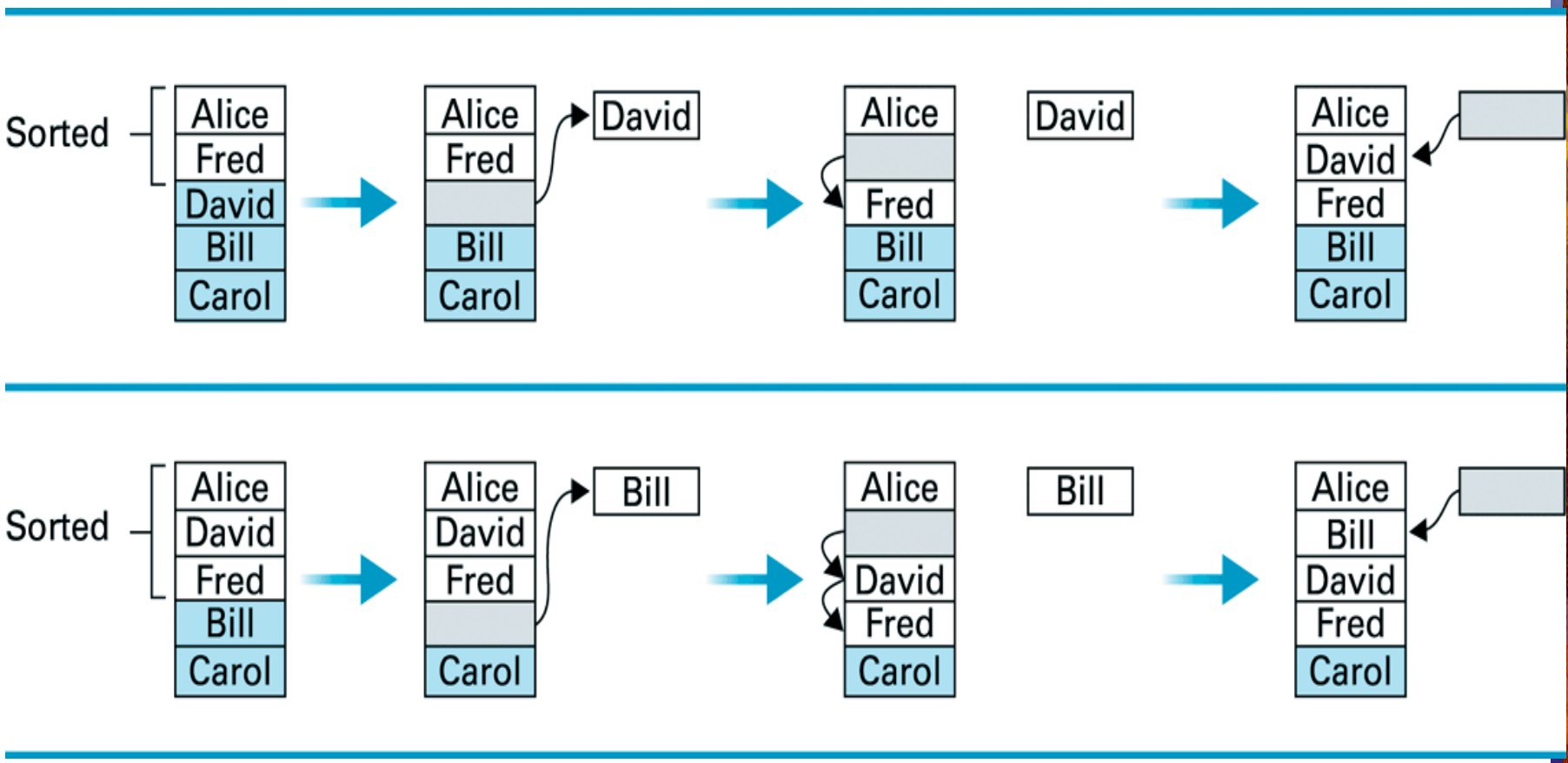
- Other Sorting techniques
  - **Quick Sort, Heap Sort, Merge Sort, ..**
  - Shell Sort, Fibonacci Sort

Sorting the list Fred, Alice, David, Bill, and Carol alphabetically (1/3)

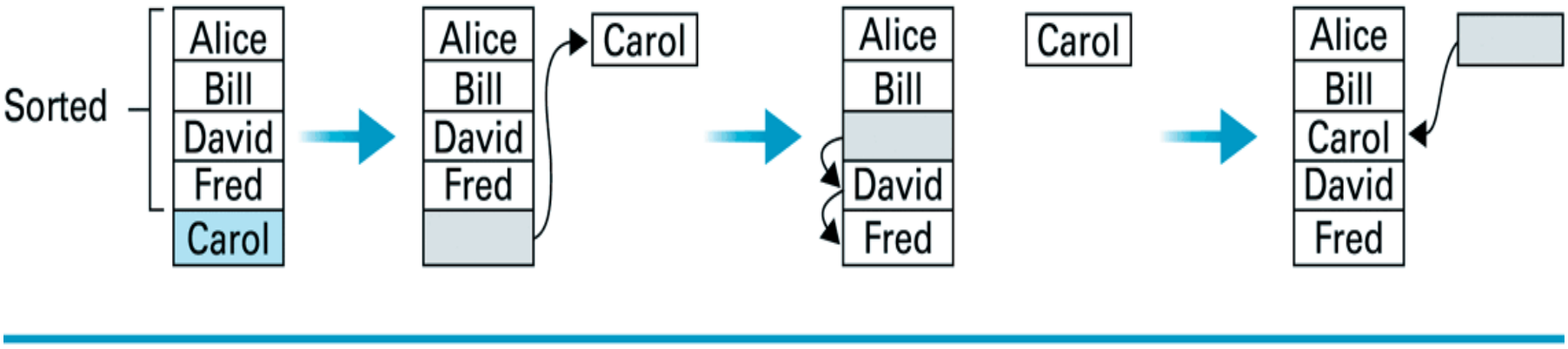
Insertion Sort



Sorting the list Fred, Alice, David, Bill, and Carol alphabetically (2/3)



Sorting the list Fred, Alice, David, Bill, and Carol alphabetically (3/3)



Sorted list:
Alice
Bill
Carol
David
Fred

The insertion sort algorithm expressed in pseudocode

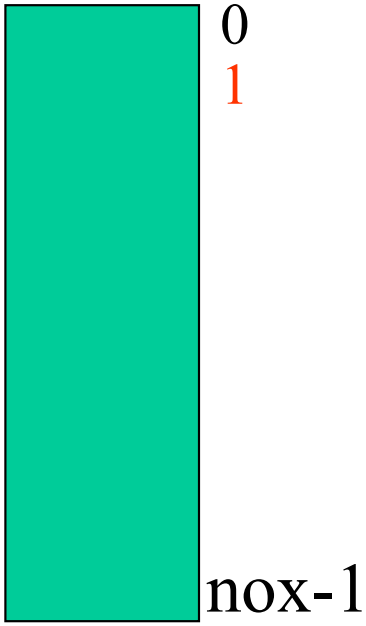
Key idea: Keep part of array always sorted

```
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 in List;
     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
    )
```

Insertion Sort in C Language

Ascending order

```
void sort( double x[ ], int nox) {
    int n = 1, k;  double tmp;   /* C array 從 0 開始*/
    while(n <= nox-1) {
        k=n-1;  tmp = x[n];    /* 我先放到 tmp */
        while( k>=0 && x[k]>tmp){
            x[k+1] = x[k];    /* 前面的 copy 到下一個*/
            --k;
        }
        x[k+1] = tmp;
        ++n;    /* check next element */
    }
}
```



Lazy evaluation
(short-cut evaluation)

Test the insertion Sort Algorithm

```
double y[ ] = {15, 38, 12, 75, 20, 66, 49, 58};
#include<stdio.h>
void pout(double*, int);  void sort(double*, int);
int main( ) {
      printf("Before sort:\n");  pout(y, 8);
      sort(y,  sizeof(y)/sizeof(double) );
      printf(" After sort:\n");  pout(y, 8);
}
void pout(double*p, int n) {
      int i;
      for(i=0; i<=n-1; ++i) {
          printf("%7.2f  ", p[i]);
      } printf(" \n");
}
```

```
Before sort:
  15.00   38.00   12.00   75.00   20.00   66.00   49.00   58.00
After sort:
  12.00   15.00   20.00   38.00   49.00   58.00   66.00   75.00
```

Insertion Sort Summary

- Best case: Already sorted → O(n)
- Worst case:
  - # of comparisons : $O(n^2)$
  - # of exchanges: $O(n^2)$ ： 剛好相反順序時
- Space: No external storage needed
- **Stable**: keep relative order for items have same key
- In practice, good for small sets (<30 items)
- Very efficient on **nearly-sorted** inputs
- 想要減少 data 交換次數： **Selection Sort**

比較的成本 vs. 交換的成本

Selection Sort 選擇排列法

array index 由 0 到 n-1

Ascending order

```
void sort( double x[ ], int nox) {
    int i, k, candt;  double tmp;
    for(i = 0; i < nox-1; ++i) {
        candt = i;   /* assume this is our candidate */
        for( k= i+1; k<=nox-1; ++k) {
            if(x[k] < x[candt]) candt = k;
        }
        tmp=x[i]; x[i]=x[candt]; x[candt]=tmp; /*第i個到定位*/
    } // for(i
}
```

選出剩下中最小的

SelectSort(array A, length n)

array index 由 0 到 n-1

Another version of **selection** sort

```
1.  for i←n-1 to 1   // note we are going down
2.    largest_index ← 0 // assume 0-th is largest
3.    for j← 1 to i   // loop finds max in [1..i]
4.      if A[j] > A[largest_index]
5.         largest_index ← j
6.    next j
7.    swap(A[i],A[largest_index]) //put max in i
8.  Next i
```

選出的放最後(第 i 個)

Selection Sort Summary

- Best case: Already sorted
  - Passes: n-1
  - Comparisons each pass: (n-k) where k pass number
  - # of comparisons: (n-1)+(n-2)+…+1 = $O(n^2)$
- Worst case: $O(n^2)$
- Space: No external storage needed
- Very few exchanges:
  - Always n-1 (better than Bubble Sort)
- Not Stable (select then swap)
  - Can be implemented as **stable** version: select then insert to correct position

Bubble Sort-v1 氣泡排列法

array index 由 0 到 n-1

Ascending order

```
void sort( double x[ ], int nox) {
    int i, k;  double tmp;
    for(i = nox-1; i >=1; --i) {
        for( k= 0; k< i; ++k) {
            if(x[k] > x[k+1]) {    /* 左大右小, 需要調換 */
                tmp= x[k]; x[k]=x[k+1]; x[k+1]=tmp;
            }
        } // for k
    } // for i
}
```

Sibling exchange sort

鄰近比較交換法

Bubble Sort-v1 example(1/2)

15	38	12	75	20	66	49	58
15	38	12	75	20	66	49	58
15	**12**	**38**	75	20	66	49	58
15	12	38	75	20	66	49	58
15	12	38	**20**	**75**	66	49	58
15	12	38	20	**66**	75	49	58
15	12	38	20	66	**49**	75	58
15	12	38	20	66	49	**58**	75

第一回合 (pass 1) : 7 次比較

第一回合後 75 到定位

Bubble Sort-v1 example (2/2)

15 38 12 75 20 66 49 58 (original)

第一回合後 75 到定位:

15 12 38 20 66 49 58 | 75

第二回合後 66 到定位:

12 15 20 38 49 **58** | 66 75

sorted

第三回合 (pass 3) ?

12 15 20 38 49 | **58** 66 75

剛剛都沒換; 還需要再做下一回合(pass)嗎 ?

Bubble Sort-v1 Features

- Time complexity in **Worst case**: Inverse sorting
  - Passes: need n-1 passes
  - Comparisons each pass: (n-k) where k is pass number
  - Total number of comparisons:

    $(n-1)+(n-2)+(n-3)+\ldots+1 = n(n-1)/2=n^2/2-n/2 = O(n^2)$
- Space: No auxilary storage needed
- Best case: already sorted
  - $O(n^2)$ Still: Many redundant passes with no swaps
  - Can be improved by using a **Flag**

Bubble Sort-v2 氣泡排列法

改良式氣泡排序法

array index 由 0 到 n-1

```
void sort( double x[ ], int nox) {
    int i, k, flag;  double tmp;
    for(i = nox-1; i >=1; --i) {
        flag = 0; /* assume no exchange  in this pass */
        for( k= 0; k< i; ++k) {
            if(x[k] > x[k+1]) {   /* 需要調換 */
                tmp= x[k]; x[k]=x[k+1]; x[k+1]=tmp; flag=1;
            } // if
        } // for k
        if(flag==0) break;  /* 剛剛這回合沒交換, 不用再做 */
    } // for i
}
```

Bubble Sort –v2 Features

- Best case: Already sorted
  - O(n) – one pass
- Total number of exchanges
  - Best case: 0
  - ***Worst*** case: $O(n^2)$ (資料*相反順序*時)
  - ➢ **Lots of exchanges:**

    A problem with large **data** items

Another version of **Bubble** sort (in pseudo code)

```
BubbleSort(array A[ ], int n)
```

array index 由 0 到 n-1

```
1.  i←n-1
2.  quit←false
3.  while(i>0 AND NOT quit)// note: going down
4.    quit←true
5.    for j=1 to i // loop does swaps in [1..i]
6.      if (A[j-1] > A[j]) {
7.         swap(A[j-1],A[j]) // put max in I
8.         quit←false     }
9.    next j
10.   i←i-1
11. wend
```

Selection Sort vs. Bubble Sort

- Selection sort:
  - more comparisons than bubble sort in best case
    - Always $O(n^2)$ comparisons : **$n(n-1)/2$**
  - But fewer exchanges : $O(n)$
  - Good for small sets/cheap comparisons, large items
- Bubble sort-v2:
  - Many exchanges : $O(n^2)$ in worst case
  - $O(n)$ on sorted input (best case) : only one pass

Chapter 5: Algorithms

5.1 The Concept of an Algorithm

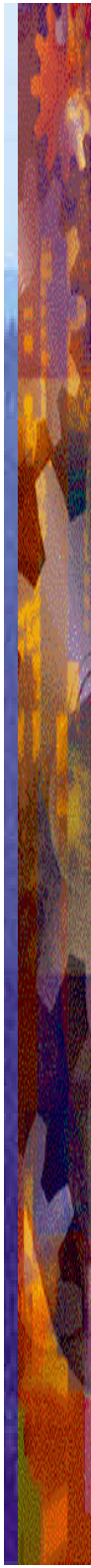
5.2 Algorithm Representation

5.3 Algorithm Discovery

5.4 Iterative Structures

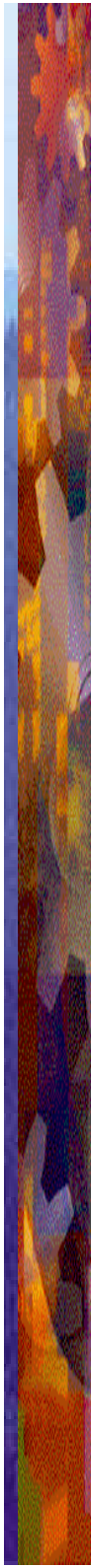
5.5 Recursive Structures

5.6 Efficiency and Correctness



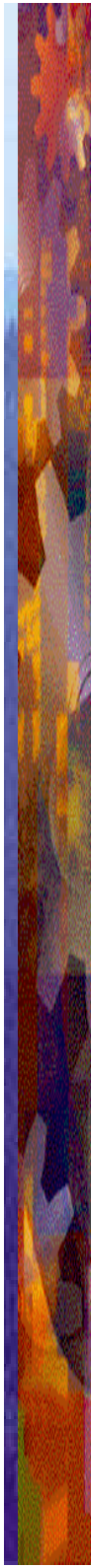
Recursive Structures

- Involves repeating the set of instructions as a subtask of itself
- An example is in processing incoming telephone calls using the call-waiting feature
  - An incomplete telephone conversation is set aside while another incoming call is processed
  - Two conversations are performed
  - But not in a one-after-the-other manner as in the loop structure
  - Instead one is performed within the other



Recursion

- Execution is performed in which each stage of repetition is as a subtask of the previous stage
- Examples:
  - divide-and-conquer in **binary search**
  - **Quick sort**
  - **Hanoi tower**
  - **Factorial(n) = n * Factorial(n-1)**
  - **GCD(m, n) = GCD(n, m%n) if n != 0**
  - **Fibonacci series formula (Fibonacci Rabbit Problem)**



Characteristics of Recursion

- Existence of multiple copies of itself (or multiple activations of the program)
- At any given time only one is actively progressing
- Each of the others waits for another activation to terminate before it can continue

要我算 N!
請你先幫我算出 (N-1) !
然後我就再乘上 N 阿就得到 N! (N階乘)

Recursive Control

所有的 Recursive 函數至少要有一個 if 或類似的測試語句以便決定是否該結束！

- Also involves
  - Initialization
  - Modification
  - **Test for termination (degenerative case)**
- Test for degenerative case
  - Before requesting further activations
  - If not met, assigns another activation to solve a revised problem that is closer to the termination condition
- Similar to a **loop** control

```
if( n == 0 ) return 1;   // 0 ! 是  1
```

```
if( n == 0 ) return m;   // GCD(m, 0) 答案是 m
```

qsort( ) in C Library

站在巨人肩膀上

- There is a library function for quick sort in C Language: qsort( ). (**unstable**)
- #include <stdlib.h>

void qsort(void *base, size_t num, size_t size,
int (*comp_func)(const void *, const void *))

void * base --- a pointer to the array to be sorted

size_t num --- the number of elements

size_t size --- the element size

int (*cf) (…) --- is a pointer to a **function** **used to compare**

int comp_func( ) 必須傳回 -1, 0, 1 代表 <， ==， >

C++ STL <algorithm>

站在巨人肩膀上

```
#include <algorithm>
using namespace std;
int x[ ] = { 38, 49, 15, 158, 25, 58, 88,66 };    // array of primitive data
#define  n  (sizeof(x)/sizeof(x[0]))
//…
sort(x, x+n);    // ascending order
   // what if we want to sort into descending order
sort(x, x+n, sortfun);  // with compare function
sort(y, y+k, sortComparatorObject);  // with Comparator Object
```

Comparison function? Default: bool operator<(first, second)

C++ Comparison function 爲bool

須傳回 true 代表 第一參數 < 第二參數 : ascending

Comparator 內要有 bool operator( ) (Obj a, Obj b) { /*…*/ }

http://www.cplusplus.com/reference/algorithm/sort/

Java 的 java.util.Arrays.sort( )

站在巨人肩膀上

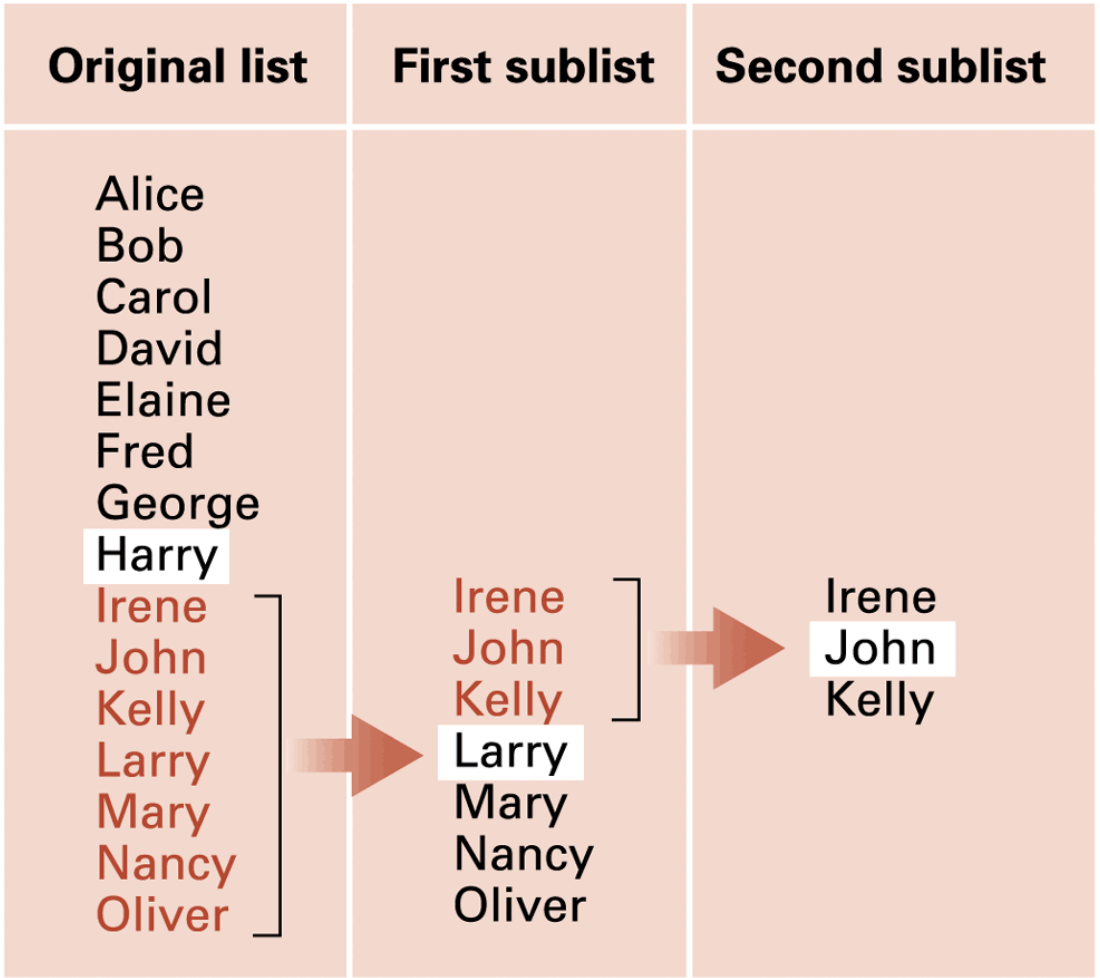
- 可透過該物件的 compareTo( ), 當然該 class 須 implements **java.lang.Comparable**
- 也可透過傳給 sort 一個 Comparator, 就是某個有 implement **java.util.Comparator** 之 class 的實體物件; 注意 Java 沒辦法把函數當作參數傳!!!

Java 不可以把函數當作參數!
java.util.Arrays.sort( ) 要傳 Comparator 物件?
不傳則會用要被 sort 之 array 的物件內的 compareTo( )

不論是 compareTo( ), 還是**Comparator 內的 compare( )**
都是類似 C 的 qsort 用的比較函數, int, 要傳回 -1, 0, 1

➢**注意 C++ STL 的是須傳回 true 與 false (seeprevious slide)**

Binary Search



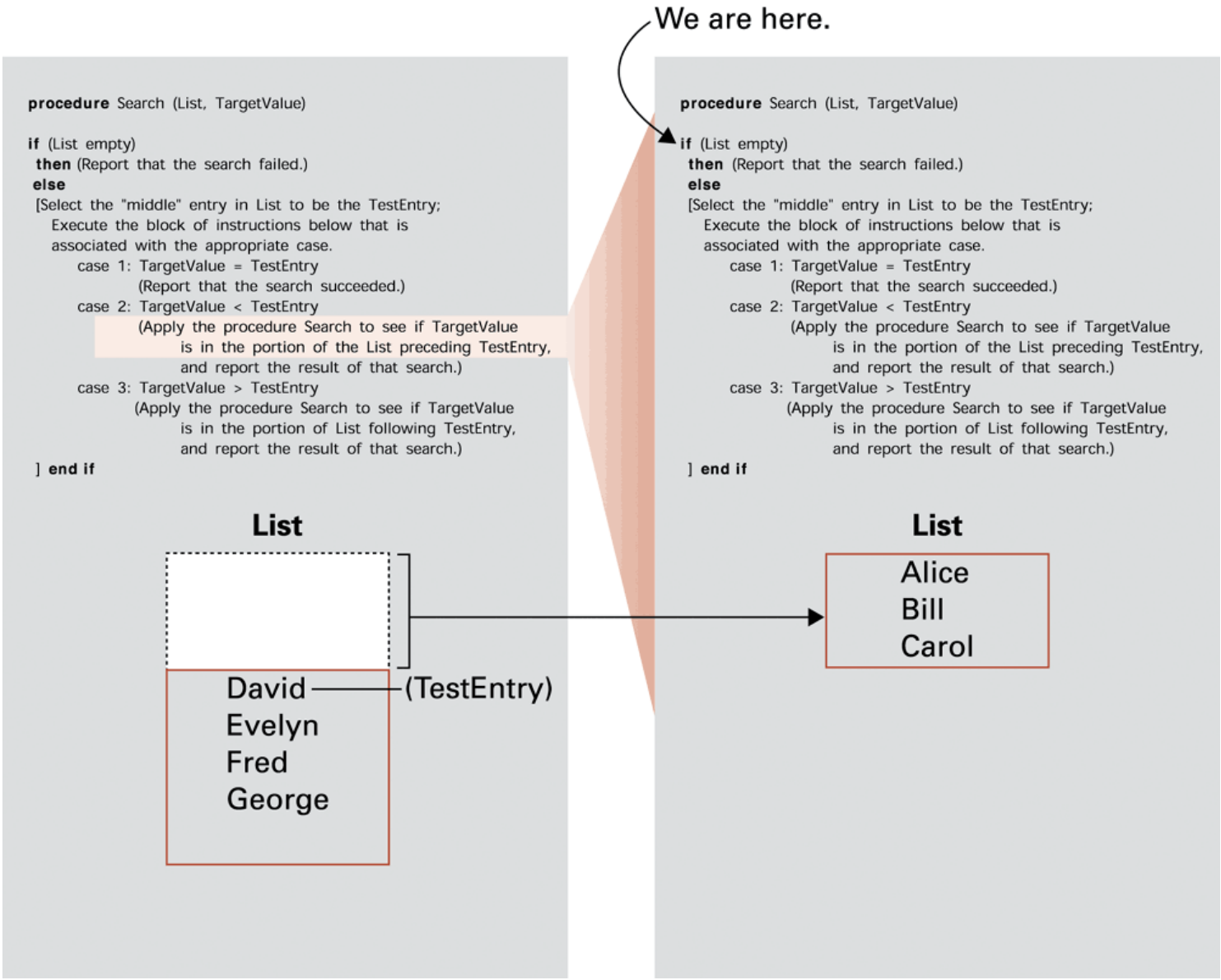
Binary Search Algorithm

```
if (List empty)
 then
  (Report that the search failed.)
 else
    [Select the "middle" entry in the List to be the TestEntry;
     Execute the block of instructions below that is
        associated with the appropriate case.
          case 1: TargetValue = TestEntry
                  (Report that the search succeeded.)
          case 2: TargetValue < TestEntry
                  (Search the portion of List preceding TestEntry for
                        TargetValue, and report the result of that search.)
          case 3: TargetValue > TestEntry
                  (Search the portion of List following TestEntry for
                        TargetValue, and report the result of that search.)
    ] end if
```

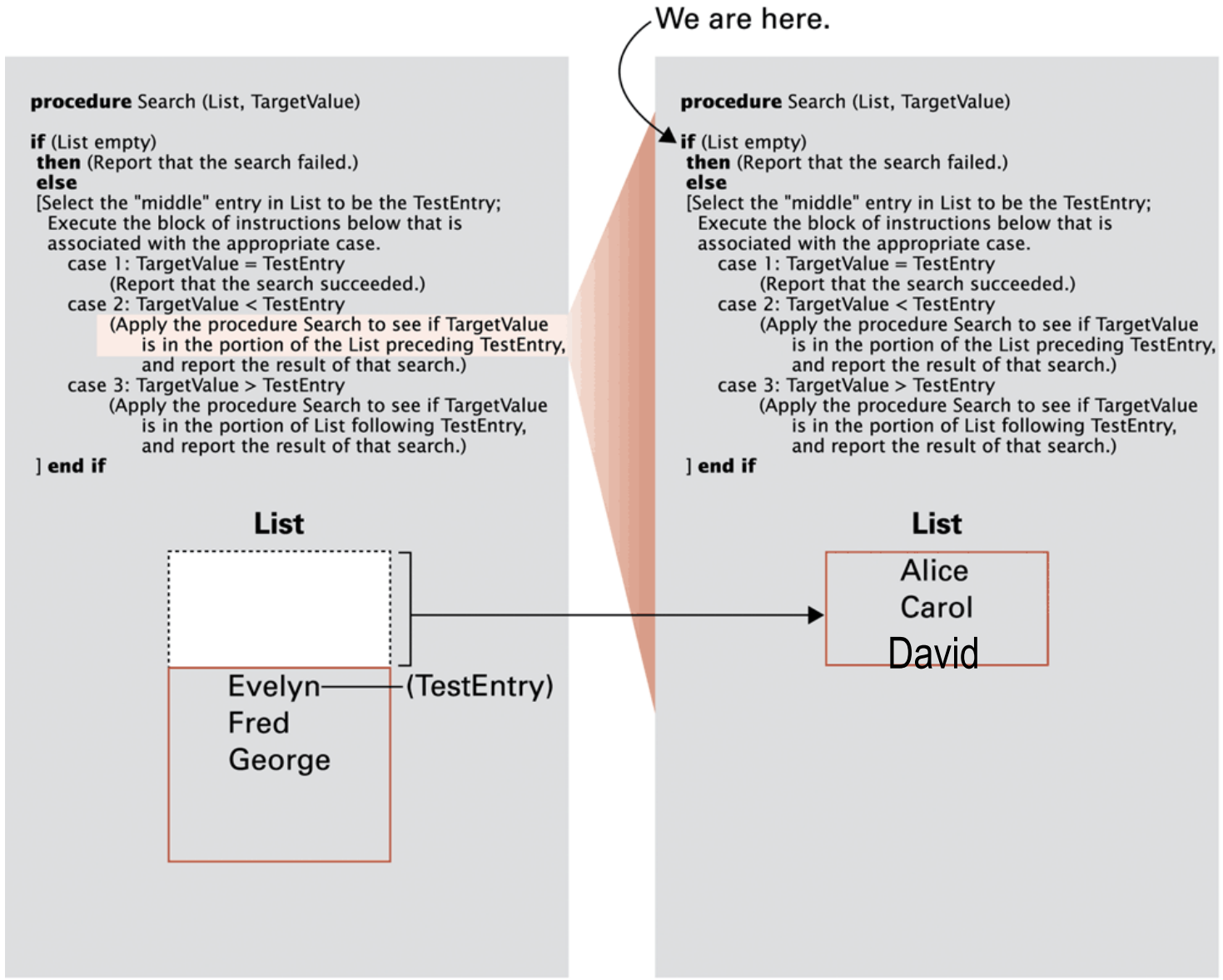
Binary Search Algorithm in Pseudocode

```
procedure Search (List, TargetValue)
if (List empty)
  then
     (Report that the search failed.)
   else
     [Select the "middle" entry in List to be the TestEntry;
      Execute the block of instructions below that is
          associated with the appropriate case.
             case 1: TargetValue = TestEntry
                      (Report that the search succeeded.)
             case 2: TargetValue < TestEntry
                      (Apply the procedure Search to see if TargetValue
                            is in the portion of the List preceding TestEntry,
                            and report the result of that search.)
             case 3: TargetValue > TestEntry
                     (Apply the procedure Search to see if TargetValue
                           is in the portion of List following TestEntry,
                           and report the result of that search.)
     ] end if
```

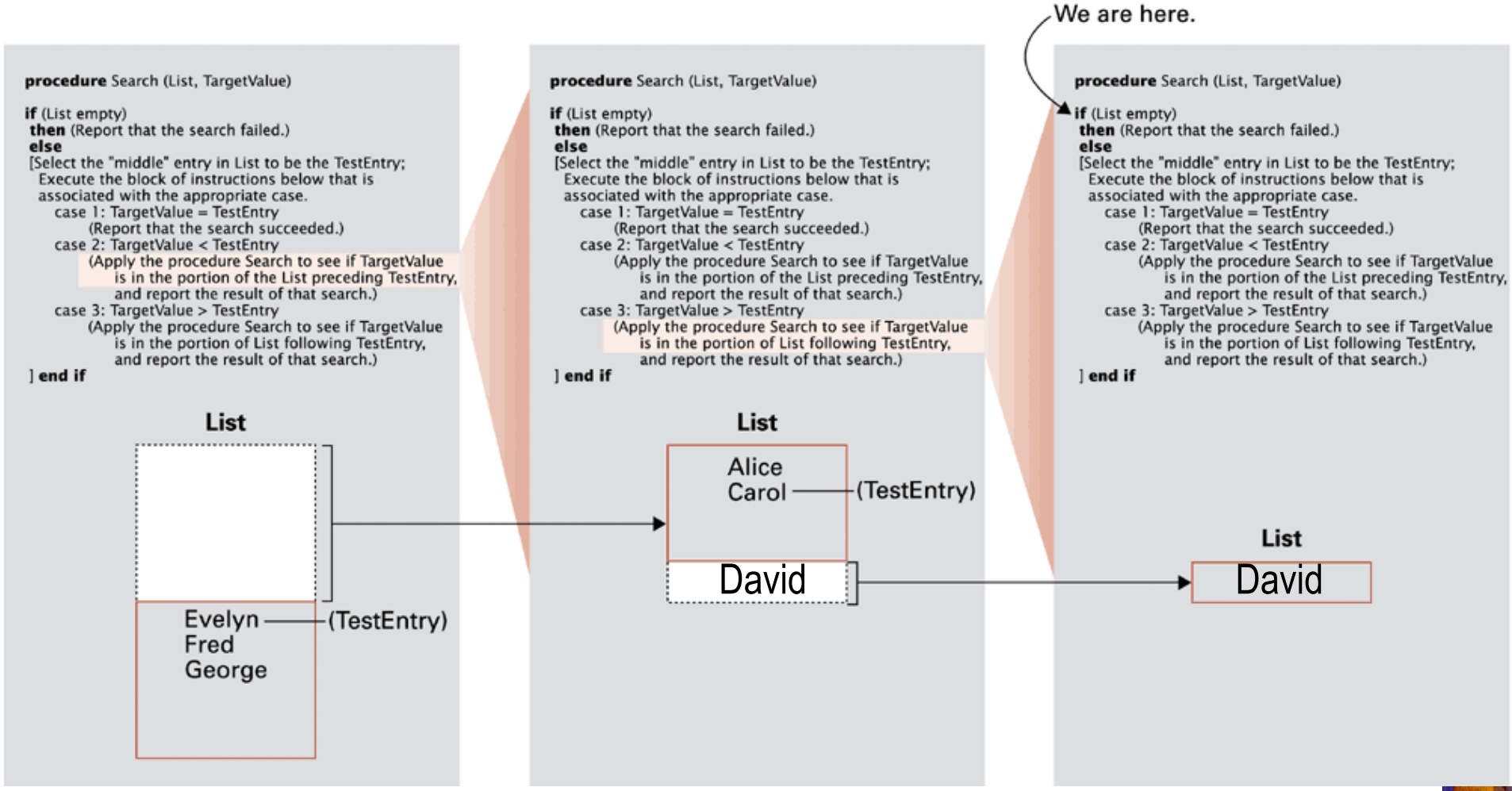
Searching for Bill



Searching for David



Searching for David



Chapter 5: Algorithms

5.1 The Concept of an Algorithm

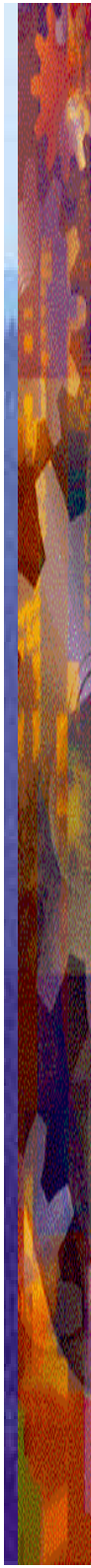
5.2 Algorithm Representation

5.3 Algorithm Discovery

5.4 Iterative Structures

5.5 Recursive Structures

5.6 Efficiency and Correctness



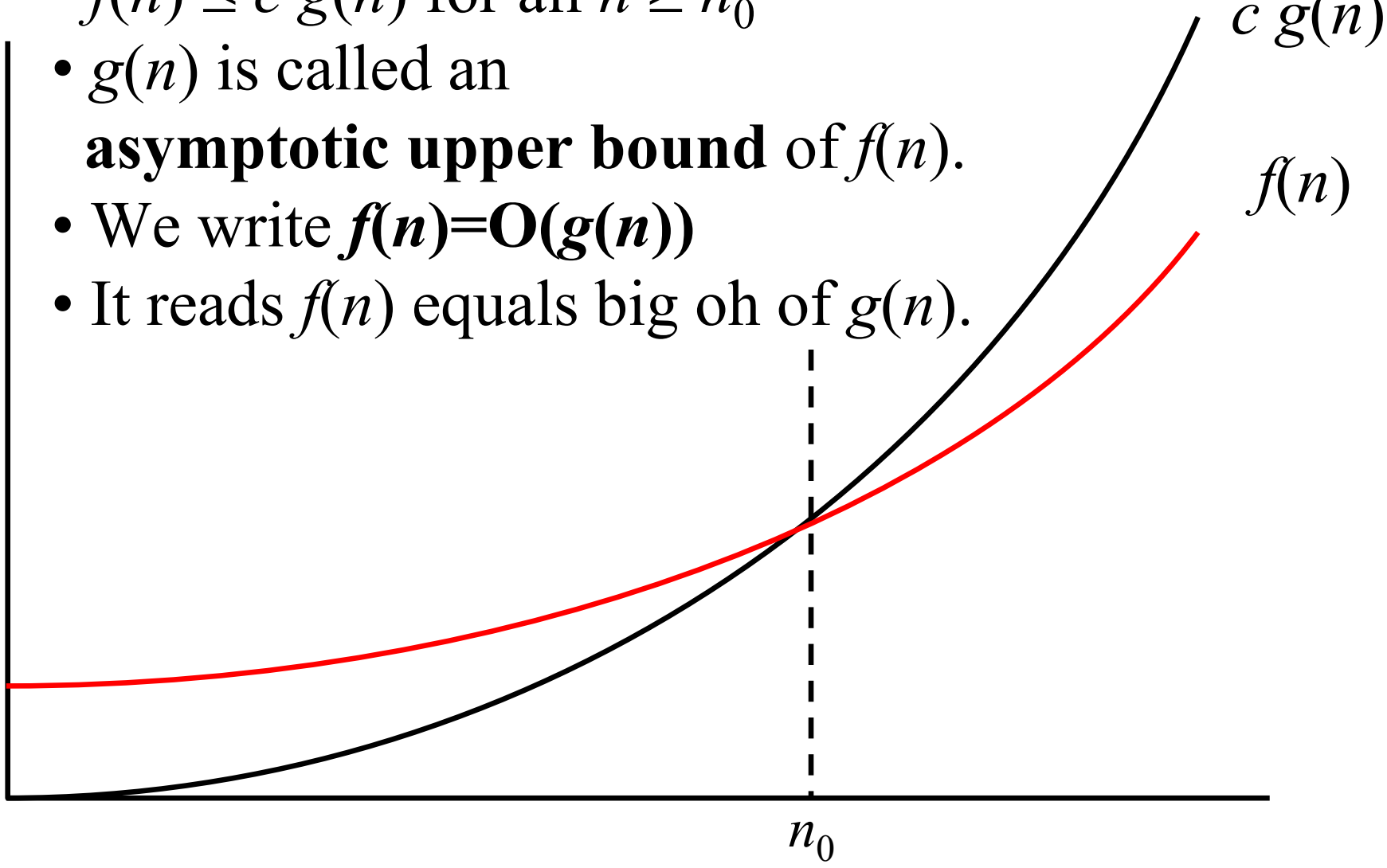
Software Efficiency

- Measured as number of instructions executed
- notation for efficiency classes
  - **O(?)**
  - **Ω(?)**
  - Θ (?)
- Best, worst, and average case

Time Complexity vs. Space Complexity

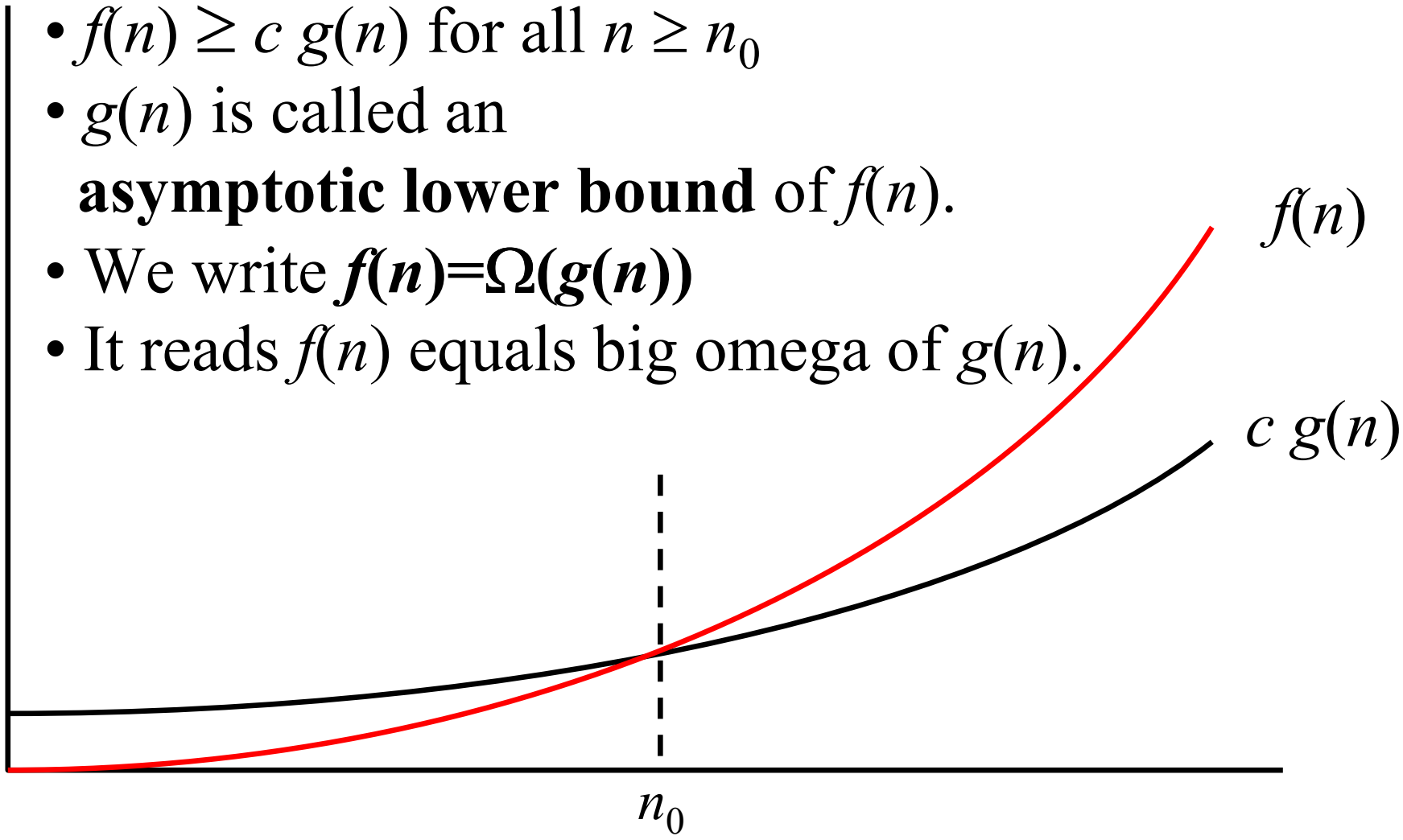
Asymptotic Upper Bound (Big O)

- $f(n) \leq c\ g(n)$ for all $n \geq n_0$
- $g(n)$ is called an **asymptotic upper bound** of $f(n)$.
- We write $\boldsymbol{f(n)=O(g(n))}$
- It reads $f(n)$ equals big oh of $g(n)$.



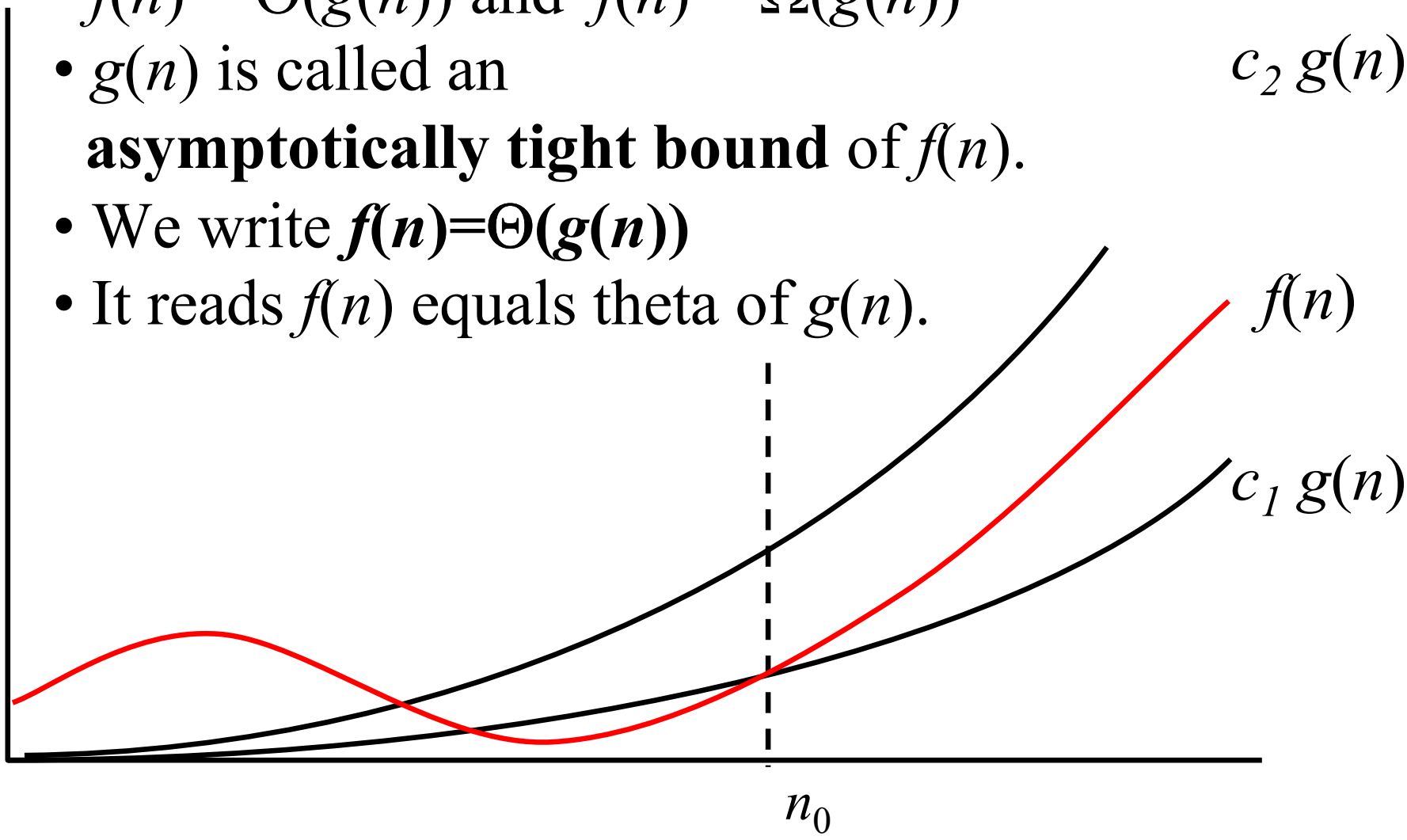
Asymptotic Lower Bound (Big Omega)

- $f(n) \geq c\ g(n)$ for all $n \geq n_0$
- $g(n)$ is called an **asymptotic lower bound** of $f(n)$.
- We write $\boldsymbol{f(n)=\Omega(g(n))}$
- It reads $f(n)$ equals big omega of $g(n)$.



Asymptotically Tight Bound (Big Theta)

- $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- $g(n)$ is called an **asymptotically tight bound** of $f(n)$.
- We write $\boldsymbol{f(n)=\Theta(g(n))}$
- It reads $f(n)$ equals theta of $g(n)$.

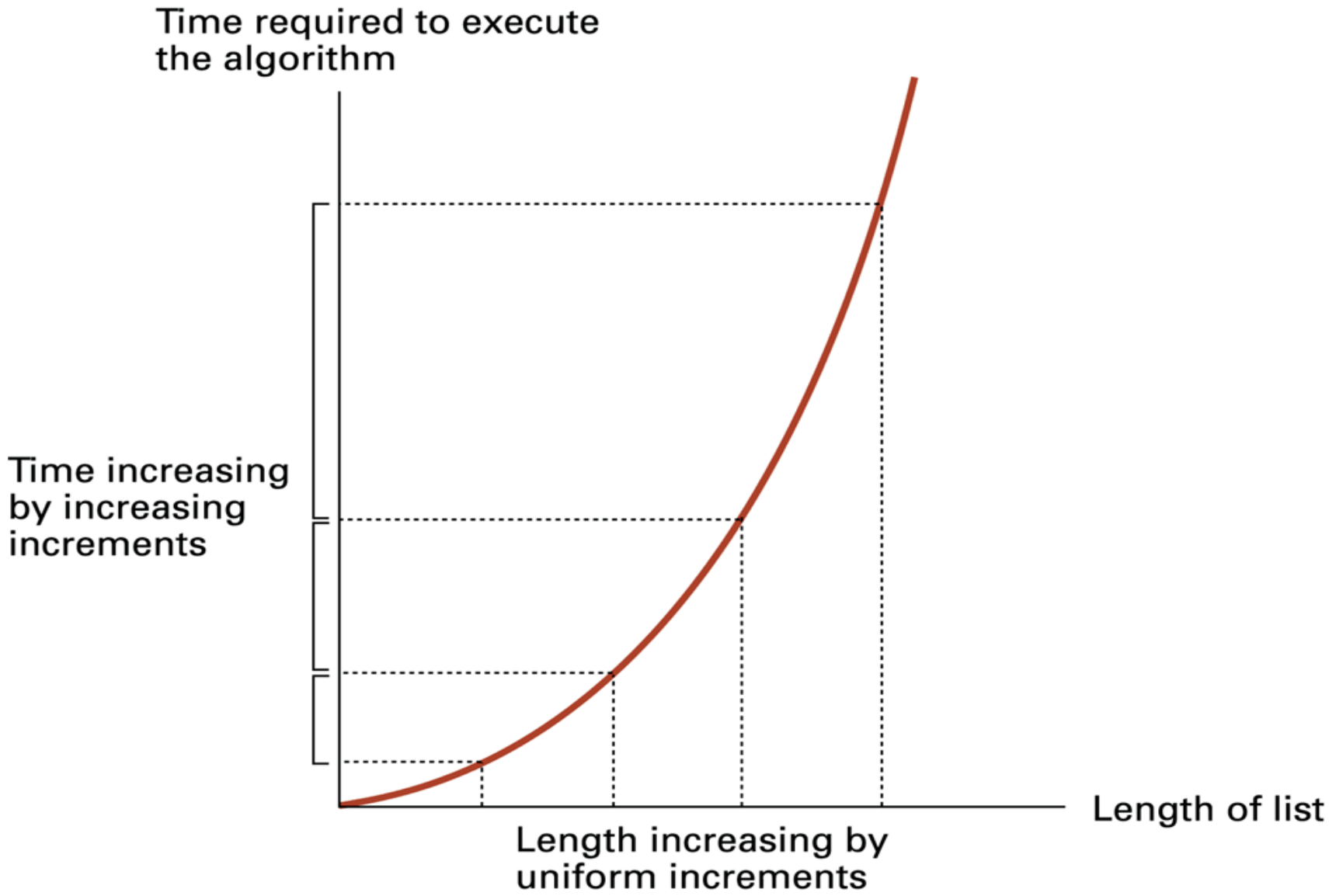


Insertion Sort in Worst Case

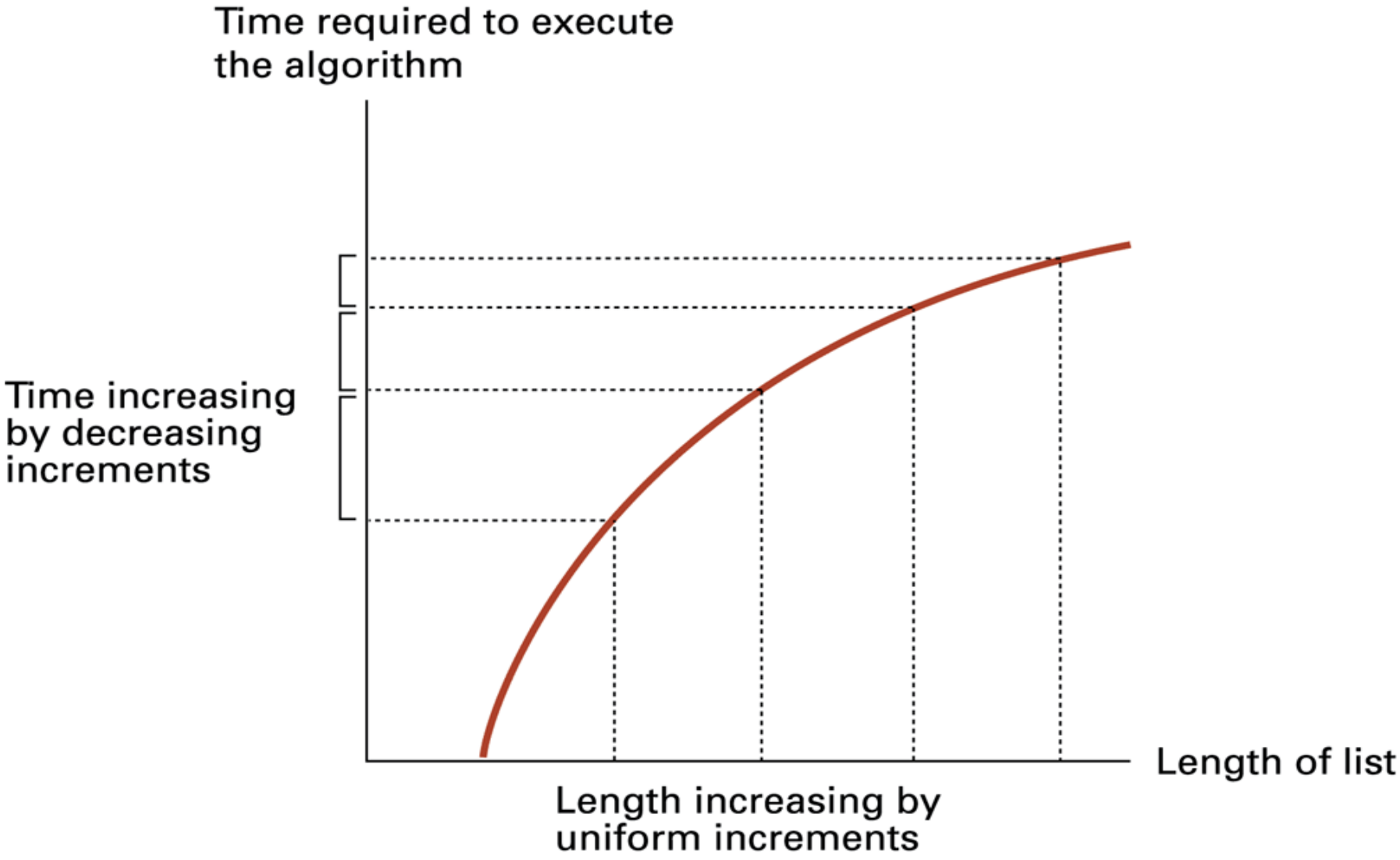
Comparisons made for each pivot

Initial list	1st pivot	2nd pivot	3rd pivot	4th pivot	Sorted list
Elaine	1 Elaine	3 David	6 Carol	10 Barbara	Alfred
David	David	2 Elaine	5 David	9 Carol	Barbara
Carol	Carol	Carol	4 Elaine	8 David	Carol
Barbara	Barbara	Barbara	Barbara	7 Elaine	David
Alfred	Alfred	Alfred	Alfred	Alfred	Elaine

Worst-Case Analysis Insertion Sort

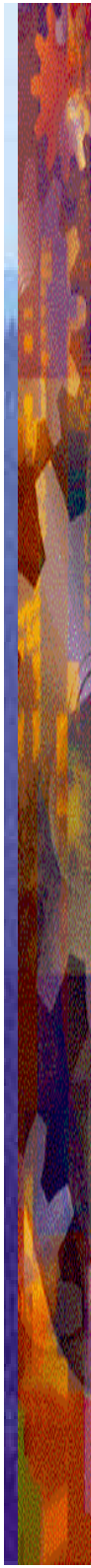


Worst-Case Analysis Binary Search



Big-Theta Notation

- Identification of the **shape** of the graph representing the resources required with respect to the size of the input data
  - Normally based on the worst-case analysis
  - Insertion sort: $\Theta(n^2)$
  - Binary search: $\Theta(\log n)$



Formal Definition

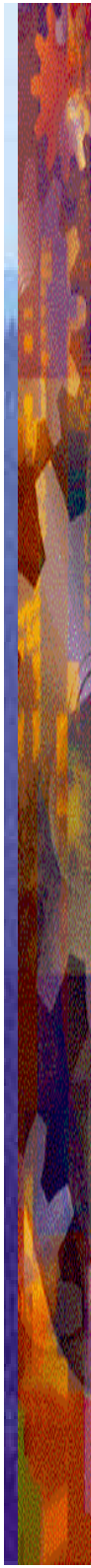
- $\Theta(n^2)$: complexity is $kn^2+o(n^2)$
  - $f(n)/n^2 \rightarrow k$, $n \rightarrow \infty$
- $o(n^2)$: functions grow slower than n^2
  - $f(n)/n^2 \rightarrow 0$, $n \rightarrow \infty$

Problem Solving Steps

1. Understand the problem
2. Get an idea
3. Formulate the algorithm and represent it as a program
4. Evaluate the program
    1. For its potential as a tool for solving other problems
    2. For accuracy

Step-wise refinement is a technique for software development using a top-down, structured approach to solving a problem.

Stepwise refinement
http://en.wikipedia.org/wiki/Program_refinement



Software Verification

- Evaluate the accuracy of the solution
- This is not easy
- The programmer often does not know whether the solution is accurate (enough)
- Example: Traveler's gold chain

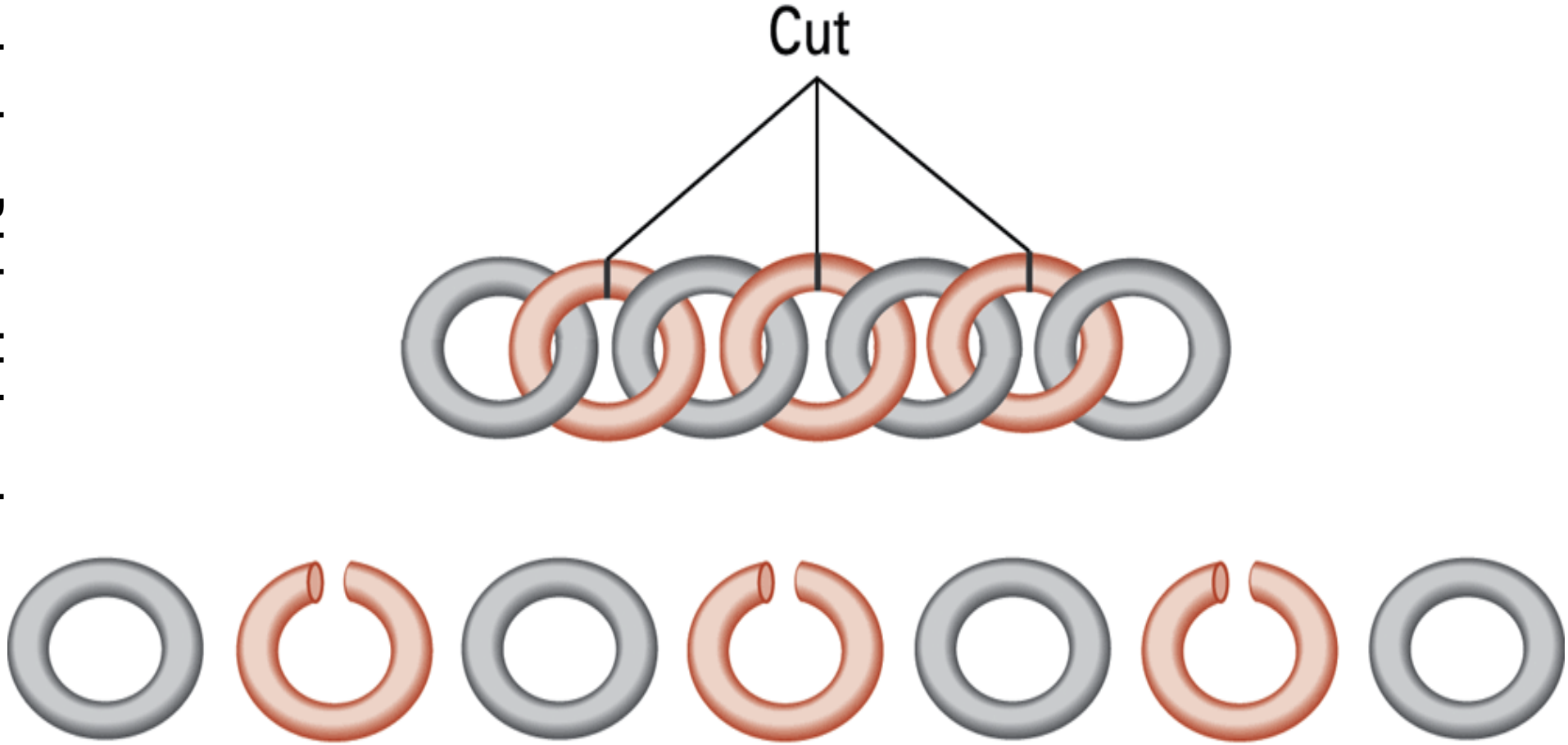
http://en.wikipedia.org/wiki/Software_verification

http://en.wikipedia.org/wiki/Software_engineering

Example: Chain Separating

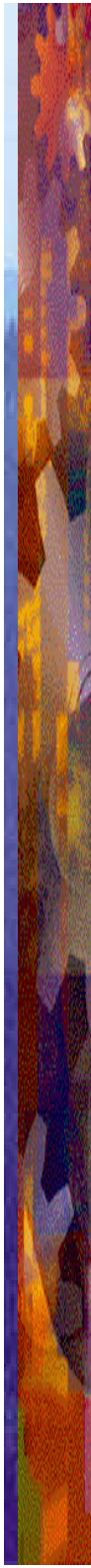
- A traveler has a **gold** chain of **seven** links.
- He must stay at an isolated hotel for seven nights.
- The rent each night consists of one link from the chain.
- What is the **fewest** number of links that **must be cut** so that the traveler can pay the hotel one link of the chain each morning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

Separating the chain using only three cuts

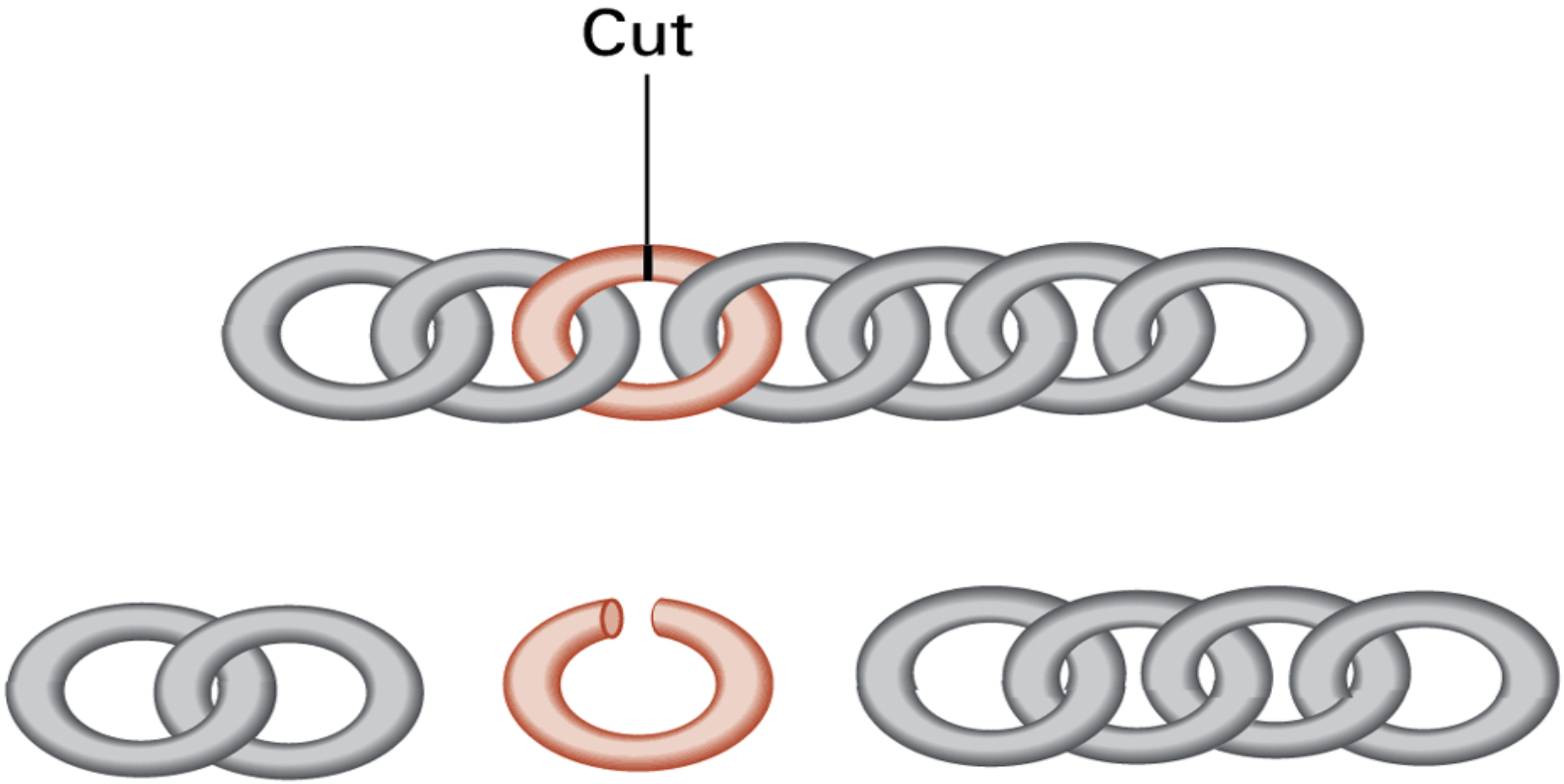


Moral of the Story

- You thought there is no better way
- You thought it is accurate enough
- But really? who knows?



Solving the problem with only one cut



有一大堆橘子要如何用最少種包裝?

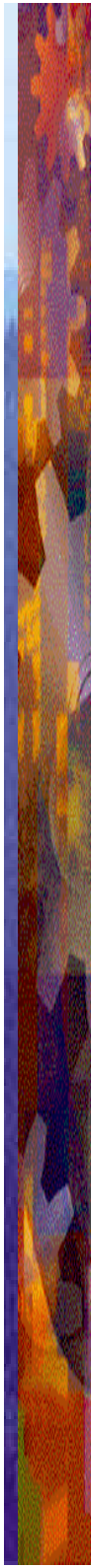
才能應付任何要買 1 ~ 1023 個橘子之客戶但不用拆包?

Ways to Level the Confidence

- For perfect confidence
  - Prove the correctness of a algorithm
  - Application of formal logic to prove the correctness of a program
- For high confidence
  - **Exhaustive** tests (全面測試)
  - Application specific test generation
- For some confidence
  - **Program verification** (程式驗證或證明)
  - Assertions

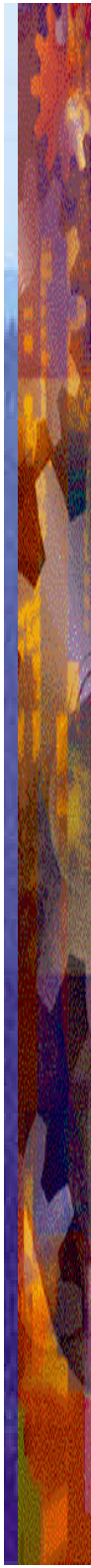
Program Verification (1/2)

- Preconditions
  - Conditions satisfied at the beginning of the program execution
- The next step is to consider how the consequences of the preconditions propagate through the program
- Assertions
  - Statements that can be established at various points in the program

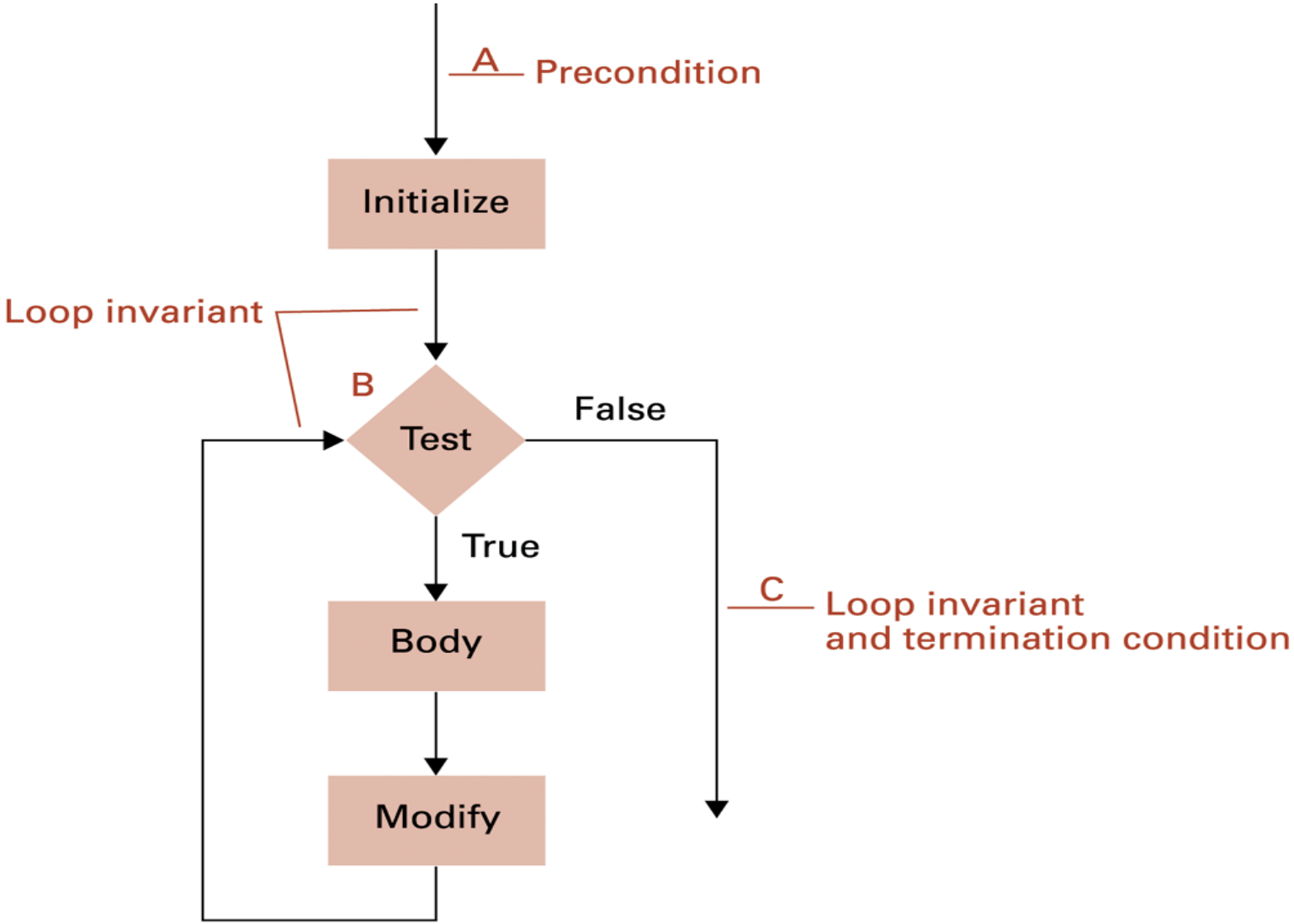


Program Verification (2/2)

- Proof of correctness to some degree
  - Establish a collection of assertions
  - If the assertion at the end of the program corresponds to the desired output specifications, we can conclude that the program is correct



In a *while* Structure

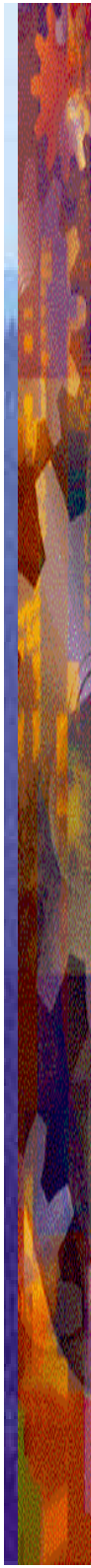


Insertion Sort Algorithm

```
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 in List;
      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
    )
```

Asserting of Insertion Sort

- Loop invariant of the outer loop
  - Each time the test for termination is performed, the names preceding the Nth entry form 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



Thank You!

CHAPTER 5

Algorithms

謝謝捧場

tsaiwn@csie.nctu.edu.tw

蔡文能