**CH5 Algorithms(演算法)**

**Definition**

\*An algorithm is an ordered (照順序) set

of unambiguous (清楚), executable (可

執行) steps that defines a terminating

(可終止) process.

\*Program

 Formal representation of an algorithm

\*Process

 Activity of executing a program

**Ordered Set**

 \*Steps in an algorithm must have a wellestablished structure in terms of the order in

which its steps are executed

 \*Each step must be an executable instruction

 Example: “Making a list of all the positive integers”

is not an executable instruction

 \*May involve more than one thread (parallel

algorithms)

**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

 \*There are, however, many meaningful

application for non-terminating processes

 \*Computer science seeks to distinguish both

 \*Problems whose answers can be obtained

algorithmically

 \*Problems whose answers lie beyond the

capabilities of algorithmic systems

**Algorithm and Its Representation**

\*Like a story and a story book

 \*Different language translation of a textbook

\*Example: converting temperature readings

from Celsius to Fahrenheit

 1. F = (9/5)C + 32

 2. Multiply the temperature reading in Celsius by

9/5 and then add 32 to the product

 3. Implemented by electronic circuit

\*Underlying algorithm is the same, only the

representations differ

**Level of Details**

\*May cause problems in communicating

algorithms

\*Example:

 \*“Convert the Celsius reading to its Fahrenheit equivalent”

 \*This might suffice for 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

**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**

\*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

**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

**Pseudocode**

 \*Less formal, more intuitive than the formalprogramming 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

**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

**Pseudocode Primitives**

\*Assignment

 name <- expression

\*Conditional selection

 if condition then action

\*Repeated execution

 while condition do activity

\*Procedure

 procedure name (generic names)

**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 (模組化) 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)

**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

**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

**Example Problem**

 \*Person A is assigned the task of determining the ages of B’s three children.

 \*B tells A that the product of the children’s ages is X.

 \*A replies that another clue is required.

 \*B tells A the sum of the children’s ages Y.

 \*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.

 \*Come out with an algorithm for person A

**Divide and Conquer Concept**

\*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

**Divide and Conquer**

\*Steps be decomposed into smaller

steps

\*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.

**Iterative Structures**

\*Used in describing algorithmic process

\*A collection of instructions is repeated in a looping manner

**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

**while vs. repeat Structure**

In repeat structure the loop’s body is always performed at least once

(posttest loop)

While in while structure, the body is

never executed if the termination is

satisfied the first time it is tested (pretest

loop)

**A Sort Problem**

 \*Sort a list of names into alphabetical order

 \*The constraint is to sort the list “within itself”

 \*In other words, sort by shuffling its entries as opposed to moving the list to another location

 \*Typical in computer applications to use the storage space efficiently

**5-4**

**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 (來電等候) feature

1.An incomplete telephone conversation is set aside while another incoming call is processed

2.Two conversations are performed

3.But not in a one-after-the-other manner as in the loop structure

4.Instead one is performed within the other

**Recursion**

1.Execution is performed in which each stage of repetition is as a subtask of the previous stage



2.Example: divide-and-conquer in binary search

**Characteristics of Recursion**

1.Existence of multiple copies of itself (or multiple activations of the program)

2.At any given time only one is actively progressing

3.Each of the others waits for another activation to terminate before it can continue

**Recursive Control**

Also involves

1.Initialization

2.Modification

3.Test for termination (degenerative case)



Test for degenerative case

1.Before requesting further activations

2.If not met, assigns another activation to solve a revised problem that is closer to the termination condition



Similar to a loop control

5-6

**Software Efficiency**

1.Measured as number of instructions executed

2.notation for efficiency classes

3.Best, worst, and average case

**Big-theta Notation**

Identification of the shape of the graph representing the resources required with respect to the size of the input data

1.Normally based on the worst-case analysis

2.Insertion sort: Θ(n2)

3.Binary search: Θ(lg n)

(n2-n)/2

lg2(n+1) – 1

**Formal Definition**

Θ(n2): complexity is kn2+o(n2 )

1.f(n)/n2  k, n  ∞



o(n2 ): functions grow slower than n2

1.f(n)/n2  0, n  ∞

(n2-n)/2 = ½ n2 - ½ n

o(n2 )

**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 (the faster, the better)

2.For accuracy

**Software Verification**

1.Evaluate the accuracy of the solution

2.This is not easy

3.The programmer often does not know whether the solution is accurate (enough)

4.Example: Traveler’s gold chain

**Ways to Level the Confidence**

For perfect confidence

1.Prove the correctness of a algorithm

2.Application of formal logic to prove the correctness of a program



For high confidence

1.Exhaustive tests

2.Application specific test generation



For some confidence

1.Program verification

2.Assertions

**Program Conditions**

Preconditions

1.Conditions satisfied at the beginning of the program execution



Propagations

1.The next step is to consider how the consequences of the preconditions propagate through the program

**Program Verification**

Assertions

1.Statements that can be established at various points in the program

2.To check for rightful precondition & propagation



Proof of correctness to some degree

1.Establish a collection of assertions

2.If all assertions pass

3.The program is correct to some degree