Lecture 2:

Algorithm: Definition & Purposes

Faculty of Computing & Informatics
Multimedia University
Cyberjaya, Malaysia

Learning Objectives

Towards the end of this lesson, you should be able to:

- define algorithm
- represent algorithm
- name examples of algorithm in real-world

Definition of Algorithm

An algorithm is an ordered set of unambiguous, executable steps, defining a terminating process

- must have well-defined order
- each step must have unique & complete interpretation
- each step must be "doable"
  e.g.: "make a list of all positive integers" is not doable
- execution of algorithm must lead to an end
  –e.g.: "divide 1.0 by 3.0" is not a terminating process . . .

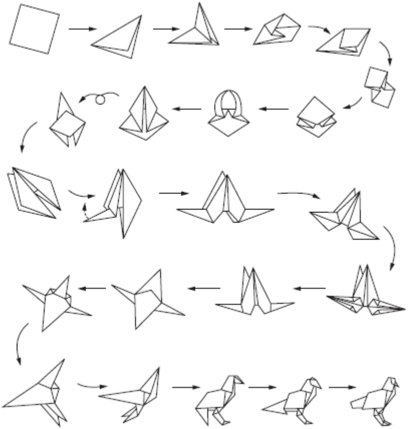
Algorithm Representation

- Algorithm representation requires some form of language
  e.g. natural language: English, Russian, Japanese, ...
  e.g. pictorial form
- Requires well-defined primitives. Primitive is a set of building blocks from which algorithm representations can be constructed
- A collection of primitives constitutes a programming language.

- Building blocks for algorithm construction
  –called: "primitives"
  –if well-defined: primitives can remove ambiguity problems
- Set of primitives plus a set of "rules for combining" constitutes a programming language
- Primitives consist of 2 portions:
  – syntax: *symbolic representation*
  – semantics: *meaning*

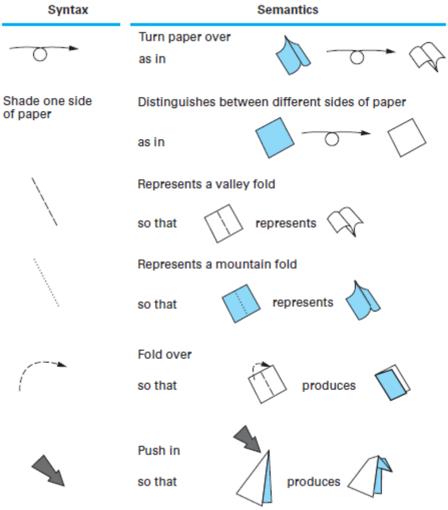
Algorithm for folding a bird

Example



Primitives

Syntax & Semantics



- intuitive/informal notational system
- good starting point for representing algorithms in any high-level programming language

Primitives

- Assignment

  $name \leftarrow expression$

- Conditional selection

  if *condition* **then** *action*

Pseudocode

Primitives

- Assignment of values to descriptive names ('*variables*'):
  - $name \leftarrow expression$
  - e.g.: $Temperature \leftarrow 18$
- Choice between two possible activities:
  - **if** *condition* **then** (*activity1*) **else** (*activity2*)
  - e.g.: **if** *TrafficLight is green* **then** (*drive*) **else** (*stop*)
- One conditional activity:
  - **if** *condition* **then** (*stop*)
  - e.g.: **if** *TrafficLight is red* **then** (*stop*)
- Repetition of one or more activities:
  - **while** (*condition*) **do** *activity*
  - e.g.: **while** (*TrafficLight is green*) **do** *hit the pedal*

Pseudocode

Reusable, encapsulated code

procedure **Greetings**
count $\leftarrow$ 3
while (Count$>$0) do
    (print the message "Hello" and
     count$\leftarrow$count $+$ 1)

To call the procedure,
if (ApproachingPerson is Friend) then (**Greetings**)

Parameterized procedures

– procedure name(parameter list)
– e.g., procedure Sort(list)

Use of Indentation

Which one is easier to read?

Algorithm 1 MyAlgo

```
if (item is taxable) then
    if (price>limit) then
        pay x
    else
        pay y
    end if
else
    pay z
end if
```

Algorithm 2 MyAlgo

```
if (item is taxable) then (if (price >limit) then (pay x)
else (pay y)) else (pay z)
```

Polya's Problem Solving Steps

The art of problem solving

- Phase 1:
  understand the problem
- Phase 2:
  think of how an algorithmic procedure might solve the problem.
- Phase 3:
  formulate the algorithm and represent it as a program.
- Phase 4:
  evaluate the program for accuracy and for its potential to use it as a tool for solving other problems.

Reconsidering the Problem Solving Phases

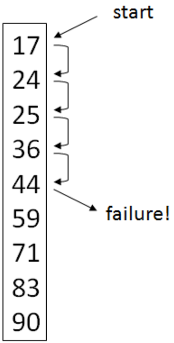
- The phases are not steps to be followed one after another – a deeper understanding of the problem often is gained by trial and error
- A good (often used) approach:
  step-wise refinement – break the problem into smaller pieces, each of which is easier to solve
- However, the art of algorithm discovery can only be mastered over a period of time so: just try . . . and don't be afraid to fail initially . . . !

Several *tools* exist that you will often use in the design of algorithms:

- iterative structures
  repeating a set of instructions in a looping manner
  while (*condition*) **do** (*activity*)
- recursive structures
  repeating a set of instructions as a subtask of itself, e.g.:
  $4!$
  $4 \times (3!) =$
  $4 \times (3 \times (2!)) =$
  $4 \times (3 \times (2 \times (1!))) =$
  $4 \times (3 \times (2 \times (1))) =$
  $4 \times (3 \times (2)) =$
  $4 \times (6)$
  24

Iterative Structures (sequential search)

- Consider the problem of searching an ordered list for a particular target value:
  - TargetValue = 44
  - TargetValue = 39



Sequential Search Pseudo-code

Algorithm 3 Sequential Search

```
procedure SEARCH(List,Target)
    if List is empty then failure!
    else
        TestEntry ← first entry in List
        while TargetValue > TestEntry AND entries remaining do
            TestEntry ← next entry in List
            if TargetValue=TestEntry then Success!
            else failure!
            end if
        end while
    end if
end procedure
```

Loop Control

Repetition by loop structure flexible:

- can be used for fixed number of iterations:

  Number $\leftarrow$ 0
  while (Number $<$ 10) **do**
      Number $\leftarrow$ Number + 1
      doSomethingUseful
  end while

- can be used for unknown number of iterations:

  roomTemperature = measureTemperature(*room*)
  while (roomTemperature $<$ 18) **do**
      let heatingSystem run
      roomTemperature = measureTemperature(*room*)
  end while

Initialize:	Establish an initial state that will be modified towards 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

Example with wrong Loop

What is wrong here...?

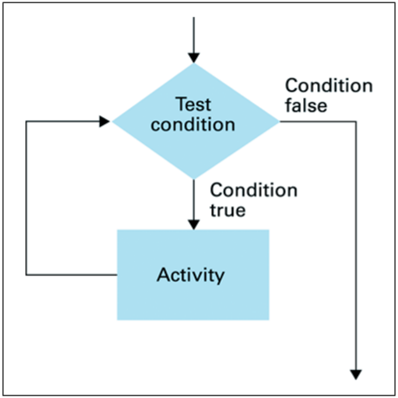
Number $\leftarrow$ 8
while (Number $\neq$ 75) **do**
    Number $\leftarrow$ Number $+$ 3
    doSomethingUseful
end while

Loop never terminates because Number never hits "75". The possible numbers are:
8, 11, 14, ..., 68, 71, 74, 77, ...

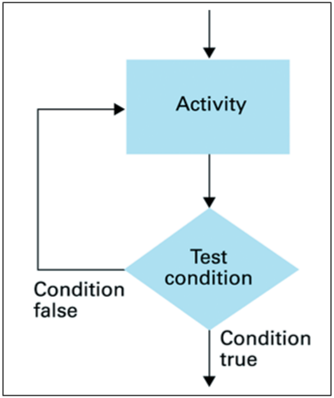
Can you see now algorithm has an objective to meet?

While loop structure vs. Repeat loop structure

While (condition) do (activity)



repeat (activity) do (condition)



Iterative Structures

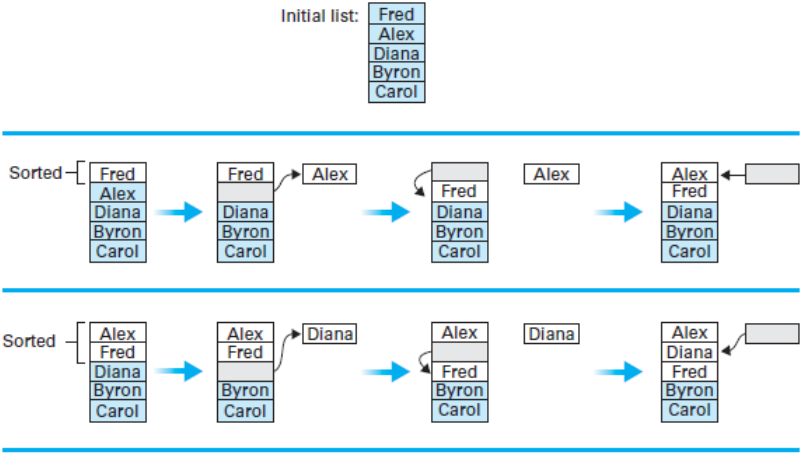
Pretest Loop

while (condition) do (loop body)

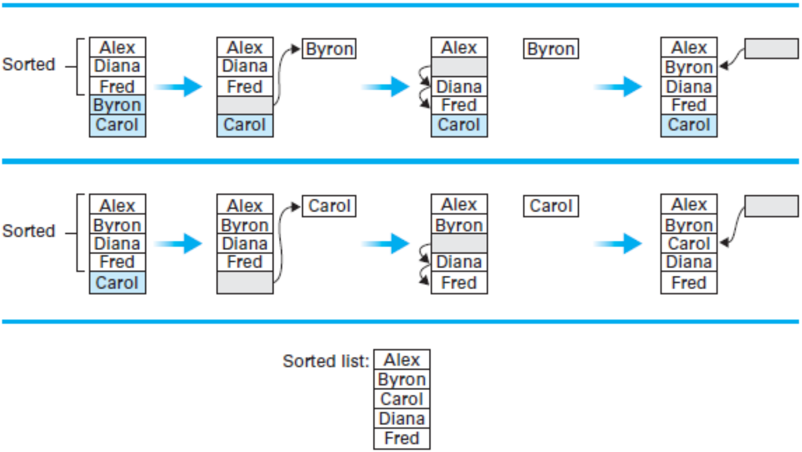
Posttest Loop

repeat (loop body) until (condition)

Example - Insertion Sort Algorithm



Example - Insertion Sort Algorithm



Example - Insertion Sort Algorithm

high-level pseudocode

```
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.
```

Example - Insertion Sort Algorithm

Detailed-level pseudocode

```
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
    )
```

Example - Insertion Sort Algorithm

Worst case situation

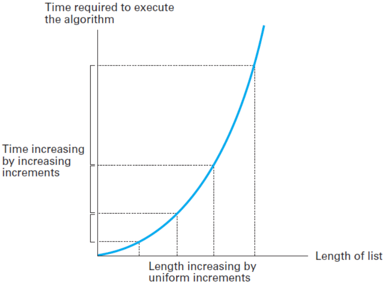
Comparisons made for each pivot

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

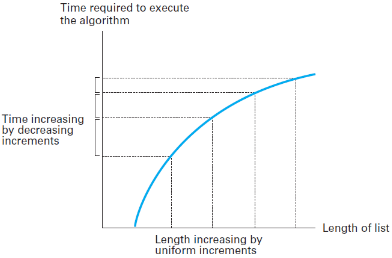
Worst case scenario

The total number of comparisons when sorting a list of n entries is $1 + 2 + 3 + 4 + \ldots + (n\text{-}1)$, which is equivalent to $\frac{1}{2}(n^2 - n)$.

Insertion Sort



Binary Search



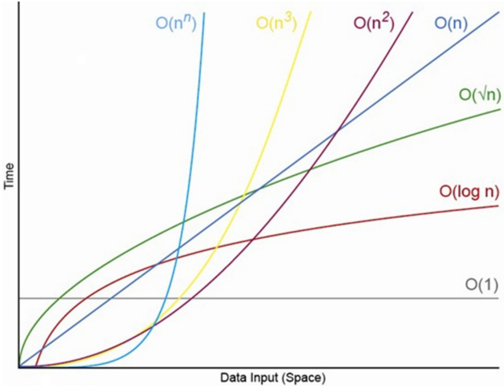
- The shape of the graph is obtained by comparing the *time* required for an algorithm to perform its task to the *size of the input data*.
- Classify algorithms according to the shapes of these graphs, based on worst-case analysis.

Growth Rate	Name
1	Constant
log(n)	Logarithmic
n	Linear
n log(n)	Linearithmic
n^2	Quadratic
n^3	Cubic
2^n	Exponential

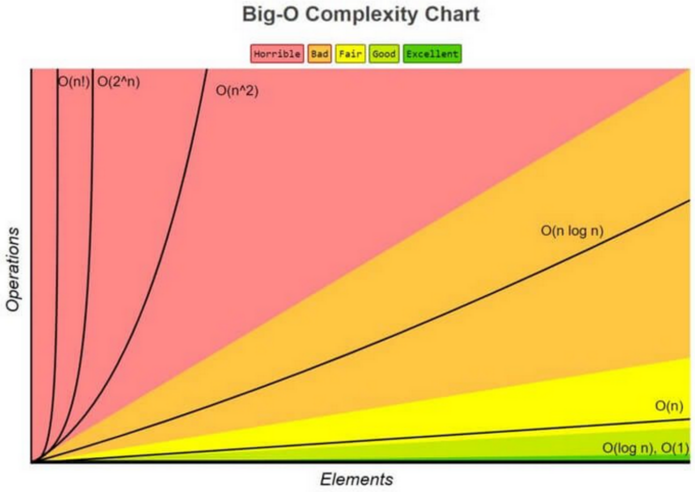
Algorithm Efficiency
Growth Rate

Growth Rate	Name	Code Example	description
1	Constant	`a= b + 1;`	statement (one line of code)
log(n)	Logarithmic	`while(n>1){ n=n/2; }`	Divide in half (binary search)
n	Linear	`for(c=0; c<n; c++){ a+=1; }`	Loop
n*log(n)	Linearithmic	Mergesort, Quicksort, ...	Effective sorting algorithms
n^2	Quadratic	`for(c=0; c<n; c++){ for(i=0; i<n; i++){ a+=1; } }`	Double loop
n^3	Cubic	`for(c=0; c<n; c++){ for(i=0; i<n; i++){ for(x=0; x<n; x++){ a+=1; } } }`	Triple loop
2^n	Exponential	Trying to braeak a password generating all possible combinations	Exhaustive search

$$1 < log_n < \sqrt(n) < n < nlog_n < n^2 < n^3 ... < 2^n < 3^n < 4^n ...$$



Algorithm Efficiency
Complexity Classes



Algorithm Efficiency

Complexity Classes

Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Selection Sort	$\Omega(N^2)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Insertion Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Merge Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$	$O(N)$
Heap Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$	$O(1)$
Quick Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N^2)$	$O(\log N)$
Radix Sort	$\Omega(N\,k)$	$\Theta(N\,k)$	$O(N\,k)$	$O(N + k)$
Count Sort	$\Omega(N + k)$	$\Theta(N + k)$	$O(N + k)$	$O(k)$
Bucket Sort	$\Omega(N + k)$	$\Theta(N + k)$	$O(N^2)$	$O(N)$

Declaration & Acknowledgment

The contents presented in this slide are meant for teaching purpose only. No commercialized component exist. The texts are partially copied from the textbooks and images are downloaded from the internet.