

Algorithms



Yi-Shin Chen

Institute of Information Systems and Applications

Department of Computer Science

National Tsing Hua University

yishin@gmail.com



The Concept of an Algorithm



The Concept of an Algorithm

- Formal Definition: An algorithm is an **ordered** set of **unambiguous, executable** steps that defines a **terminating** process
- Problem = motivation for algorithm
- Algorithm = procedure to solve the problem
 -
- Program – a formal and executable representation of an algorithm
- Process – activity of executing a program

Algorithm Criteria

- Input
 - Zero/more quantities are externally supplied
- Output
 - At least one quantity is produced
- Definiteness
 - Each instruction is clear and unambiguous
- Finiteness
 - Terminate after a finite number of steps
- Effectiveness:
 - Every instruction must be basic and easy to be computed

Example: Party Hosting

■ Input: Budget \$1000, #People: 50

■ Output: A Happy Party

■ Instructions

- Invite guests and receive confirmations
- Buy foods and presents (from Carrefour)
- Shopping for the party dresses (at Sogo)
- Buy dancing shoes (at La New)
- Dress up
- Welcome guests and have fun

Representation

■ Description of algorithm sufficient to communicate it to the desired audience

■ Natural languages

- English, Chinese, ...etc.
- A lot of sentences...

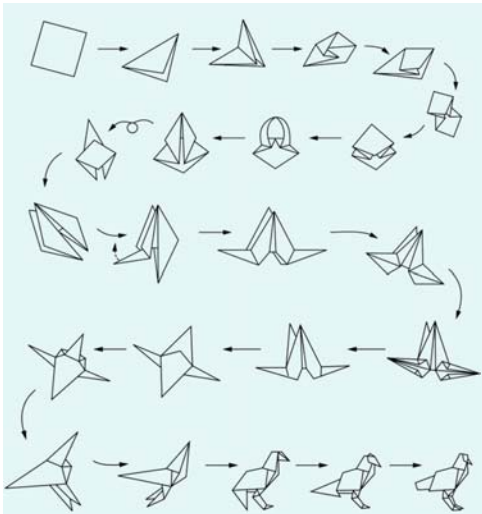
■ Graphic representation

- Flowchart.
- Feasible only if the algorithm is small and simple

■ Programming language + few English

- C++
- Concise and effective!

Example for Representation



Folding a bird from a square piece of paper

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 a mountain fold so that
	Fold over so that
	Push in so that

Origami primitives

Algorithm Representation

- Primitives— a well-defined set of building blocks from which algorithm representations can be constructed.
 - syntax: symbolic representation
 - semantics: concept represented

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

Pseudocode

Pseudocode

- A formal programming language in favor of a less formal, more intuitive notational system
- A notational system in which ideas can be expressed **informally** during the algorithm development process
 - Focus more on the numerous interrelated concepts and criteria
 - Researches show that human minds is capable of manipulating only about 7 details at a time
 - **Flowcharts** and **graphical representation** techniques are two other useful tools

Pseudocode Primitives

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

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

Conditional Branch

- **if** (condition) **then** (activity 1) **else** (activity 2)

- Divide the total by 366 or 365 dependent on the year is a leap year or not
- E.g., **if** (year is leap year) **then** (divide total by 366) **else** (divide total by 365)
- E.g.,
 if (year is leap year)
 then (divide total by 366)
 else (divide total by 365)

Conditional Loop

- **while** (condition) **do** (activity)

- While there are tickets to sell, keep selling tickets
- E.g.,
 while (tickets remain to be sold) **do**
 (sell tickets)

Procedure

- The set of activities to be used later

- **procedure** name
- E.g.,
 procedure Greetings (var)
 assign Count **the value** var+ 6;
 while Count > 0 **do**
 (print the message "Hello" and
 assign Count **the value** Count -1)

The Sequential Search Algorithm In Pseudocode

```
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
```

Thinking Processes

Algorithm Discovery

- The development of a program consists:
 - Discovering the underlying algorithm
 - Representing that algorithm as a program

- Theory of problem solving
 - The algorithm to generate an algorithm for any particular problem is purely imaginary
 - There are certain problems that are **unsolvable!!**
 - The ability to solve problems is more like an **artistic skill** to be developed

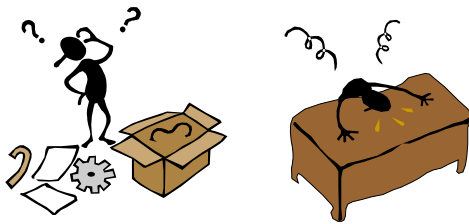
Problem Solving Phases

1. Understand the problem
 2. Get an idea how an algorithmic procedure might solve the problem.
 3. Formulate the algorithm and represent it as a program
 4. Evaluate the program for accuracy and its potential as a tool for solving other problems
- ⇒ Not necessarily completed in sequence

Incubation Periods

■ Between conscious work and the sudden inspiration

- Reflect a process
 - A subconscious part of the mind appears to continue working
 - Forces the solution into the conscious mind



Techniques For “Getting A Foot In The Door”

- Work the problem backwards
- Solve an easier related problem
 - Relax some of the problem constraints
 - Solve pieces of the problem first = bottom up methodology
- Stepwise refinement = top-down methodology
 - Popular technique because it produces modular programs

Art of Algorithm Discovery

- Paradigm shifts - "The Structure of Scientific Revolutions" by Thomas Kuhn
 - First a break with tradition, with old ways of thinking, with old paradigms
 - Suddenly, everything takes on a different interpretation

Algorithm Primitives and Structures

■ Primitives

- | | |
|-------------------------|---|
| ■ Assignment | <i>name</i> \leftarrow <i>expression</i> |
| ■ Conditional selection | if <i>condition</i> then <i>action</i> |
| ■ Repeated execution | while <i>condition</i> do <i>activity</i> |
| ■ Procedure | procedure <i>name</i> (<i>generic names</i>) |

■ Repetitive structures used in describing algorithmic processes

- Iterative structures
- Recursive structures

Iterative Structures

- Repeat collections of instructions in a looping manner
- Four kinds of code blocks:
 - Initialize: establish an initial state to be modified
 - Test: compare the current state with the termination condition
 - Statement: the block repeated in each iteration
 - Modify: change the state toward the termination condition.

While-loop vs. Repeat-loop

- While-loop: `initialize; while(test) { activity; modify; }`
- Repeat-loop: `initialize; repeat (activity; modify;) until (test)`

- For-loop: `for(initialize; test; modify) { statement; }`

Recursive Structures

- Another loop paradigm for repetitive structures (by invoking itself)
- Divide-and-Conquer
 - The execution creates multiple instances (children)
 - Each child is born to conquer revised smaller problems and return the results back to the parent
 - Only one instance is actively progressing

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

Binary Search Algorithm

```
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
```

Performance

Efficiency and Correctness

- One problem can have a variety of algorithms
- The choice between efficient and inefficient algorithms can make the difference
 - Time and storage complexity of the algorithm

Performance Evaluation

■ Two criteria:

- Space Complexity
 - How much memory space is used?
- Time Complexity
 - How many running time is needed?

■ Two approaches:

- Performance Analysis
 - Machine independent
 - A prior estimate
- Performance Measurement
 - Machine dependent
 - A posterior testing

29

Space Complexity

■ $S(P) = C + S_p(I)$

■ C is a **fixed** part:

- Independent of the inputs and outputs.
- Including: Instruction space, space for simple variables, fixed-size structured variables, constants

■ $S_p(I)$ is a **variable** part:

- Depends on the particular problem instance
- Space of referenced variable and recursion stack space (**Instance Characteristics**)
 - Include the number and magnitude of the **input** and **output**

Space Complexity: Simple Function

```
float Abc(float a, b, c)
{
    return a+b*b*c+(a+b-c)/(a+b)+4.00;
}
```

- $I = a, b, c$
- C = space for the program + space for variables a, b, c , Abc = constant
- $S_{Abc}(I) = 0$
- $S(Abc) = C + S_{Abc}(I) = \text{constant}$

31

Space Complexity : Iterative Summing

```
float Sum(float *A, const int n)
{
    float s = 0;
    for(int i=0; i<n; i++)
        s += A[i];
    return s;
}
```

- $I = n$ (number of elements to be summed)
- C = constant
- $S_{Sum}(I) = 0$ (a stores only the address of array)
- $S(Sum) = C + S_{Sum}(I) = \text{constant.}$

32

Space Complexity : Recursive Summing

```
float Rsum(float *A, const int n)
{
    if (n<=0) return 0;
    else return (Rsum(A, n-1) + A[n-1]);
}
```

- C = constant
- $I = n$ (number of elements to be summed)
 - Each recursive call “Rsum” requires $4 \cdot (1 + 1 + 1) = 12$ bytes
 - Number of calls: $Rsum(A, n) \rightarrow Rsum(A, n-1) \rightarrow \dots \rightarrow Rsum(A, 0)$
 $\Rightarrow n+1$ calls
- $S(Rsum) = C + S_{Rsum}(n) = \text{const} + 12 \cdot (n+1)$

33

Time Complexity

- $T(P) = C + T_P(I)$
- C is a **constant** part:
 - Compile time
- $T_P(I)$ is a **variable** part:
 - Running time
 - Use “**program step**” to estimate $T_P(I)$
 - “program step” = a statement whose execution time is **independent** of instance characteristics(I).

$abc = a + b + b * c;$ \rightarrow one program step
 $a = 2;$ \rightarrow one program step

Time Complexity : Iterative Summing

```
float Sum(float *A, const int n)
{ float s = 0;           // 1 step
  for(int i=0; i<n; i++) // n+1 steps
    s += A[i];           // n steps
  return s;              // 1 step
}
```

- $I = n$ (number of elements to be summed)
- $T_{\text{Sum}}(I) = 1 + n + 1 + n + 1 = 2n + 3$
- $T(\text{Sum}) = C + T_{\text{Sum}}(n) = \text{constant} + (2n + 3)$

35

Time Complexity : Recursive Summing

```
float Rsum(float *A, const int n)
{
  if (n <= 0)           // 1 step
    return 0;           // 1 step
  else return (Rsum(A, n-1) + A[n-1]); // 1 step
}
```

- $I = n$ (number of elements to be summed)
- $T_{\text{Rsum}}(n) = ?$

36

Time Complexity : Recursive Summing

```
float Rsum(float *A, const int n)
{
    if (n<=0) // 1 step
        return 0; // 1 step
    else return (Rsum(A, n-1) + A[n-1]); // 1 step
}
```

- $l = n$ (number of elements to be summed)
- Relation for $T_{\text{Rsum}}(n)$:
- $T_{\text{Rsum}}(0) = 2$
- $T_{\text{Rsum}}(n) = 2 + T_{\text{Rsum}}(n-1)$
 $= 2 + (2 + T_{\text{Rsum}}(n-2))$
 $= \dots$
 $= 2n + T_{\text{Rsum}}(0) = 2n + 2$

37

Observation on Step Counts

- In the previous examples :
 $T_{\text{Sum}}(n) =$
 $T_{\text{Rsum}}(n) =$
- Can we say that **Rsum** is faster than **Sum** ?
 -
 - The execution time of each step is different.
- Instead, we are interesting in “**Growth Rate**” of the program
 - “How the running time changes with changes in the instance characteristics?”

38

Program Growth Rate

- $T_{\text{Sum}}(n) = 2n + 3$ means
 - When n is tenfold(10X)
 - The running time $T_{\text{Sum}}(n)$ is tenfold(10X).
 - Runs in **linear time**.
- $T_{\text{Rsum}}(n) = 2n + 2$
 - Runs in **linear time**.
- $T_{\text{Sum}}(n)$ and $T_{\text{Rsum}}(n)$
 - The same growth rate
 - Equal in time complexity

39

Asymptotic Notation

- Predict the growth rate
 - Scenario 1: $c_1 = 1$, $c_2 = 2$, and $c_3 = 100$
 - P1:
 - P2:
 - Scenario 2: $c_1 = 1$, $c_2 = 2$, and $c_3 = 1000$
 - P1:
 - P2:

Notation: Big-O (O)

■ Definition:

- Let $f(n) = O(g(n))$
- iff these exist $c, n_0 > 0$ such that $f(n) \leq c g(n)$ for all $n \geq n_0$

■ Examples

- $3n + 2 =$
 -
- $100n + 6 =$
 -
- $10n^2 + 4n + 2 =$
 -

41

Theorem 1.2

■ Theorem 1.2:

If $f(n) = a_m n^m + \dots + a_1 n + a_0$, then $f(n) = O(n^m)$

■ Proof:

$$\begin{aligned} f(n) &= a_m n^m + \dots + a_1 n + a_0 \\ &\leq |a_m| n^m + \dots + |a_1| n + |a_0| \\ &\leq n^m (|a_m| + \dots + |a_1| + |a_0|) \\ &\leq n^m c \text{ for } n \geq 1 \\ \text{So, } f(n) &= O(n^m) \end{aligned}$$

- *Leading constants and lower-order terms do not matter*

Practices

■ $n^2 - 10n - 6 =$

■ $n + \log n =$

■ $n + n \log n =$

■ $n^2 + \log n =$

■ $2^n + n^{10000} =$

■ $n^4 + 1000 n^3 + n^2 = O(n^4)$, True or False?

■ $n^4 + 1000 n^3 + n^2 = O(n^5)$, True or False?

43

Properties of Big-O

■ $f(n) = O(g(n))$

■ $g(n)$ is an **upper bound** of $f(n)$.

■ $n = O(n) = O(n^{2.5}) = O(n^3)$

■ However, we want $g(n)$ as small as possible

■ Big-O: **worst-case running time** of a program

■ $f(n) = O(g(n)) \rightarrow g(n) = O(f(n))$

44

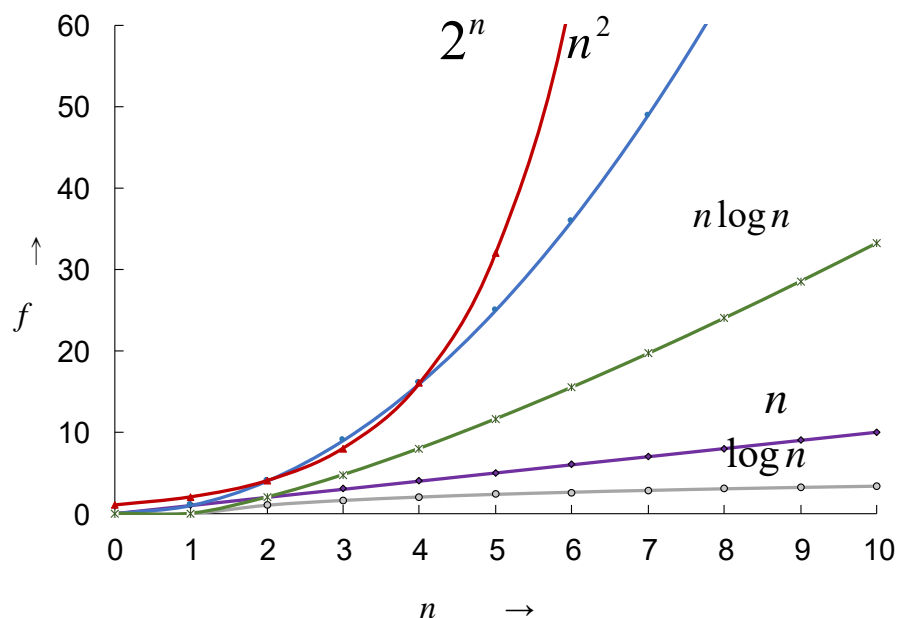
Naming Common Functions

Complexity	Naming
$O(1)$	Constant time
$O(\log n)$	Logarithmic time
$O(n \log n)$	$O(\log n) \leq . \leq O(n^2)$
$O(n^2)$	Quadratic time
$O(n^3)$	Cubic time
$O(n^{100})$	Polynomial time
$O(2^n)$	Exponential time

When n is large enough, the latter terms take more time than the former ones

45

Plot of Common Function Values



46

Running Times on Computers

	f (n)							
	n	n	$n \log_2 n$	n^2	n^3	n^4	n^{10}	2^n
	10	.01 μs	.03 μs	.1 μs	1 μs	10 μs	10s	1μs
	20	.02 μs	.09 μs	.4 μs	8 μs	160 μs	2.84h	1ms
	30	.03 μs	.15 μs	.9 μs	27 μs	810 μs	6.83d	1s
	40	.04 μs	.21 μs	1.6 μs	64 μs	2.56ms	121d	18m
	50	.05 μs	.28 μs	2.5 μs	125 μs	6.25ms	3.1y	13d
	100	.10 μs	.66 μs	10 μs	1ms	100ms	3171y	$4 \cdot 10^{13}y$
	10^3	1 μs	9.96 μs	1 ms	1s	16.67m	$3.17 \cdot 10^{13}y$	$32 \cdot 10^{283}y$
	10^4	10 μs	130 μs	100 ms	16.67m	115.7d	$3.17 \cdot 10^{23}y$	
	10^5	100 μs	1.66 ms	10s	11.57d	3171y	$3.17 \cdot 10^{33}y$	
	10^6	1ms	19.92ms	16.67m	31.71y	$3.17 \cdot 10^7y$	$3.17 \cdot 10^{43}y$	

μs = microsecond = 10^{-6} second; ms =milliseconds = 10^{-3} seconds

s = seconds; m = minutes; h = hours; d = days; y = years;

47

Rule of Sum

- To compute the sequential statements in a program
- $f_1(n) = O(g_1(n))$, $f_2(n) = O(g_2(n))$
 - $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$
- Examples:
 - $f_1(n) = O(n)$, $f_2(n) = O(n^2)$
 - $f_1(n) + f_2(n) =$
 - $f_1(n) = O(n)$, $f_2(n) = O(n)$
 - $f_1(n) + f_2(n) =$

48

Rule of Product

- Used in time analysis of **nested loops**

- $f_1(n) = O(g_1(n)), f_2(n) = O(g_2(n))$

- $f_1(n) \times f_2(n) = O(g_1(n) \times g_2(n))$

- Examples:

- $f_1(n) = O(n), f_2(n) = O(n)$

- $f_1(n) \times f_2(n) = O(n^2)$.

```
for (i=0; i<n; i++) {           // O(n)
    for (j=0; j<n; j++)         // O(n)
        sum := sum + 1;        // O(1)
}
```

49

Complexity of Binary Search

- Analysis of the while loop:

- Iteration 1: n values to be searched

- Iteration 2: $n/2$ left for searching

- Iteration 3: $n/4$ left for searching

- ...

- Iteration $k+1$: $n/(2^k)$ left for searching

- When $n/(2^k) = 1$, searching must finish.

- $n = 2^k$

- $k = \log_2 n$

- Hence, **worst-case running time** of binary search is $O(\log_2 n)$

50

Notation: Omega (Ω)

■ Definition

- $f(n) = \Omega(g(n))$
- iff there exist $c, n_0 > 0$ such that $f(n) \geq c g(n)$ for all $n \geq n_0$

■ Examples:

- $3n + 2 = \Omega(n)$
 - $3n + 2 \geq$
- $100n + 6 = \Omega(n)$
 - $100n + 6 \geq$
- $10n^2 + 4n + 2 = \Omega(n^2)$
 - $10n^2 + 4n + 2 \geq$

51

Notation: Theta (Θ)

■ Definition

- $f(n) = \Theta(g(n))$
- iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

■ Examples

- $3n + 2 = \Theta(n)$
- $100n + 6 = \Theta(n)$
- $10n^2 + 4n + 2 = \Theta(n^2)$

52

Performance Measurement

- Obtain actual space and time requirement when running a program.
- How to do time measurement in codes ?
 - Method 1: Use `clock()`, measured in `clock ticks`
 - Method 2: Use `time()`, measured in `seconds`
- To time a short program
 - Repeat it many times
 - Take the average.