



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
 - Often one of many possibilities
- 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

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!

Algorithm Representation

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

A photograph of the interior of a large, circular dome, likely a cathedral or church. The dome is covered in intricate stained glass windows, featuring a variety of colors including red, blue, yellow, and green. The light from the windows creates a warm, golden glow throughout the space. The architecture is classical, with a high, vaulted ceiling and a central opening at the top. The overall atmosphere is one of grandeur and historical significance.

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

Logical Thinking

- To solve the problem, you usually need these following two steps:
 - Break down the big problem (logically)
 - Divide
 - Combine the smaller puzzles (logically)
 - Conquer

Take Home Exercise

- Find examples for divide and conquer in your daily life.

The background of the slide is a photograph of a large, circular dome with a complex pattern of colorful stained glass windows. The dome is illuminated from within, creating a warm, golden glow. The windows are arranged in a circular pattern, with some larger windows in the center and smaller ones towards the edges. The colors of the glass include red, blue, yellow, and green. The overall effect is one of architectural grandeur and artistic detail.

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

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

```
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)

⇐ Indentation and parentheses for readability!!

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)
```

Algorithm Primitives and Structures

■ Primitives

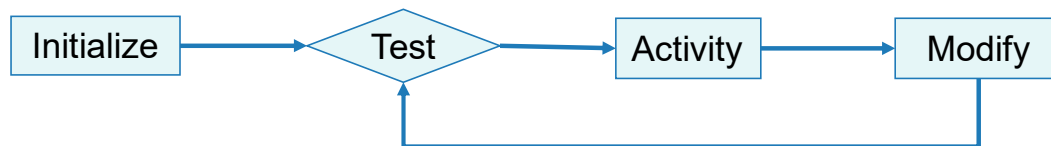
- Assignment *name* \leftarrow *expression*
- Conditional selection **if** *condition* **then** *action*
- Repeated execution **while** *condition* **do** *activity*
- Procedure **procedure** *name* (*generic names*)

■ Repetitive structures used in describing algorithmic processes

- Iterative structures
- Recursive structures

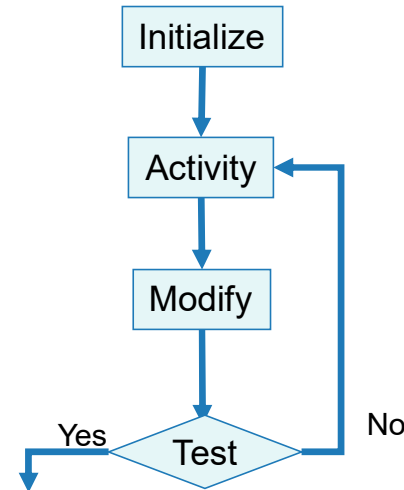
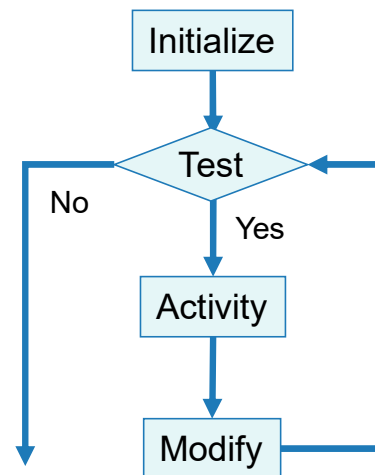
Iterative Structures

- 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

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
```


Binary Search Algorithm

Line	
1	procedure Search (List, TargetValue)
2	if (List empty)
3	then
4	(Report that the search failed.)
5	else
6	[Select the “middle” entry in List to be the TestEntry;
7	Execute the block of instructions below that is
8	associated with the appropriate case.
9	case 1: TargetValue = TestEntry
10	(Report that the search succeeded.)
11	case 2: TargetValue < TestEntry
12	(Apply the procedure Search to see if TargetValue
13	is in the portion of the List preceding TestEntry,
14	and report the result of that search.)
15	case 3: TargetValue > TestEntry
16	(Apply the procedure Search to see if TargetValue
17	is in the portion of List following TestEntry,
18	and report the result of that search.)
19] end if

The background of the slide is a photograph of a large, circular dome interior, likely a cathedral or a grand hall. The dome is covered in numerous stained glass windows of various sizes, each featuring intricate designs in shades of blue, red, yellow, and green. The lighting is dim, with the primary light source being the glow from the stained glass, creating a dramatic and atmospheric effect. The word "Performance" is centered over the lower half of the dome.

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

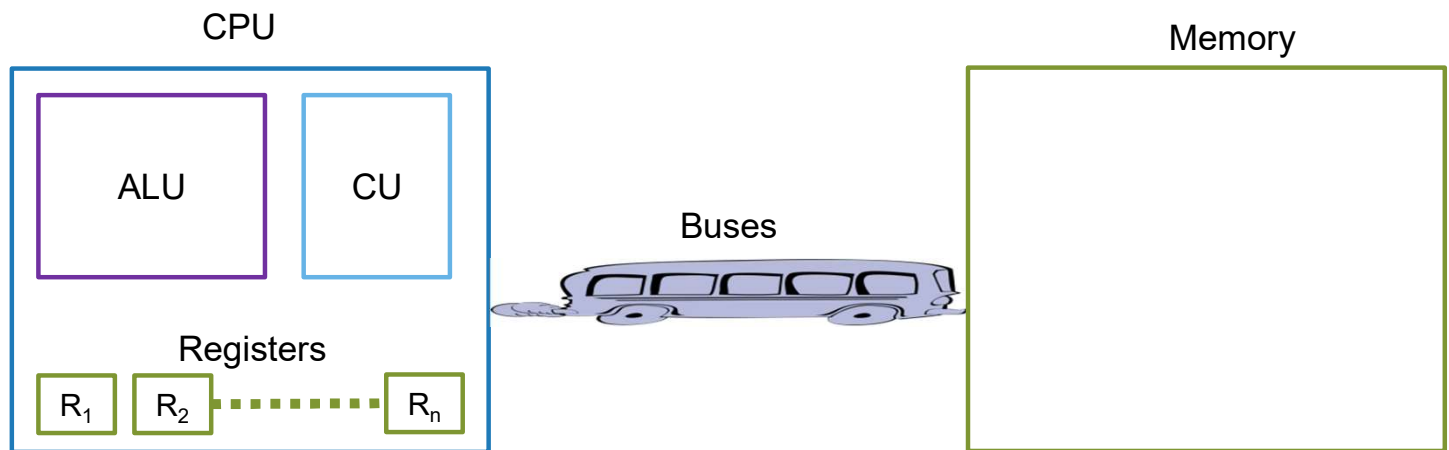
- How much memory space is used?

■ Time Complexity

- How many running time is needed?

Von Neumann Architecture

- Central Processing Unit (CPU)
 - Arithmetic and Logic Unit (ALU)
 - Control Unit (CU)
 - Registers
- Memory Unit
- Buses



Performance Evaluation

- Space Complexity
- Time Complexity

Iteration vs. Recursive

- Reminder: To solve the problem, you usually need these following two steps:
 - Break down the big problem (logically)
 - Divide
 - Combine the smaller puzzles (logically)
 - Conquer
- What are the differences?

Performance Evaluation

- Two criteria:
 - Space Complexity
 - How much memory space is used?
 - Time Complexity
 - How many running time is needed?
- Two approaches to evaluate:
 - Performance Analysis
 - Machine independent
 - A prior estimate
 - Performance Measurement
 - Machine dependent
 - A posterior testing

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**

The background of the slide is a photograph of a large, circular dome interior, likely a cathedral or a grand hall. The dome is covered in numerous stained glass windows of various sizes, each featuring intricate designs in shades of blue, red, yellow, and green. The lighting is warm and ambient, highlighting the architectural details and the vibrant colors of the glass. The text "Performance Evaluation" is centered over the lower half of the image.

Performance Evaluation

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
- $S_{Abc}(I) = 0$
- $S(Abc) = C + S_{Abc}(I) = \text{constant}$

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

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
 - Number of calls: $Rsum(A, n) \rightarrow Rsum(A, n-1) \rightarrow \dots \rightarrow Rsum(A, 0) \Rightarrow n+1$ calls
- $S(Rsum) = C + S_{Rsum}(n) =$

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).

Time 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;
}
```

- $l = n$ (number of elements to be summed)
- $T_{\text{Sum}}(l) =$
- $T(\text{Sum}) = C + T_{\text{Sum}}(n) =$

Time Complexity : Recursive Summing

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

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

Observation on Step Counts

- In the previous examples :

$$T_{\text{Sum}}(n) = 2n + 3 \text{ steps}$$

$$T_{\text{Rsum}}(n) = 2n + 2 \text{ steps}$$

- Can we say that **Rsum** is faster than **Sum** ?

-

-

- Instead, we are interesting in “**Growth Rate**” of the program

- *“How the running time changes with changes in the instance characteristics?”*

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

Asymptotic Notation

■ Predict the growth rate

- Scenario 1: $c_1 = 1$, $c_2 = 2$, and $c_3 = 100$

- P1: $c_1 n^2 + c_2 n = n^2 + 2n$

- P2: $c_3 n = 100n$

- Scenario 2: $c_1 = 1$, $c_2 = 2$, and $c_3 = 1000$

- P1: $c_1 n^2 + c_2 n = n^2 + 2n$

- P2: $c_3 n = 1000n$

-

- Compare the complexity for a ***sufficiently large value*** of n

Notation: Big-O (O)

■ Definition:

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

■ Examples

- $3n + 2 =$
 - $3n+2 \leq 4n$ for all $n \geq 2$
- $100n + 6 =$
 - $100n+6 \leq 101n$ for all $n \geq 6$
- $10n^2 + 4n + 2 =$
 - $10n^2 + 4n + 2 \leq 11 n^2$ for all $n \geq 5$

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:

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

So, $f(n) = O(n^m)$

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

Practices

■ $n^2 - 10n - 6 =$

■ $n + \log n =$

■ $n + n \log n =$

■ $n^2 + \log n =$

■ $2^n + n^{10000} =$

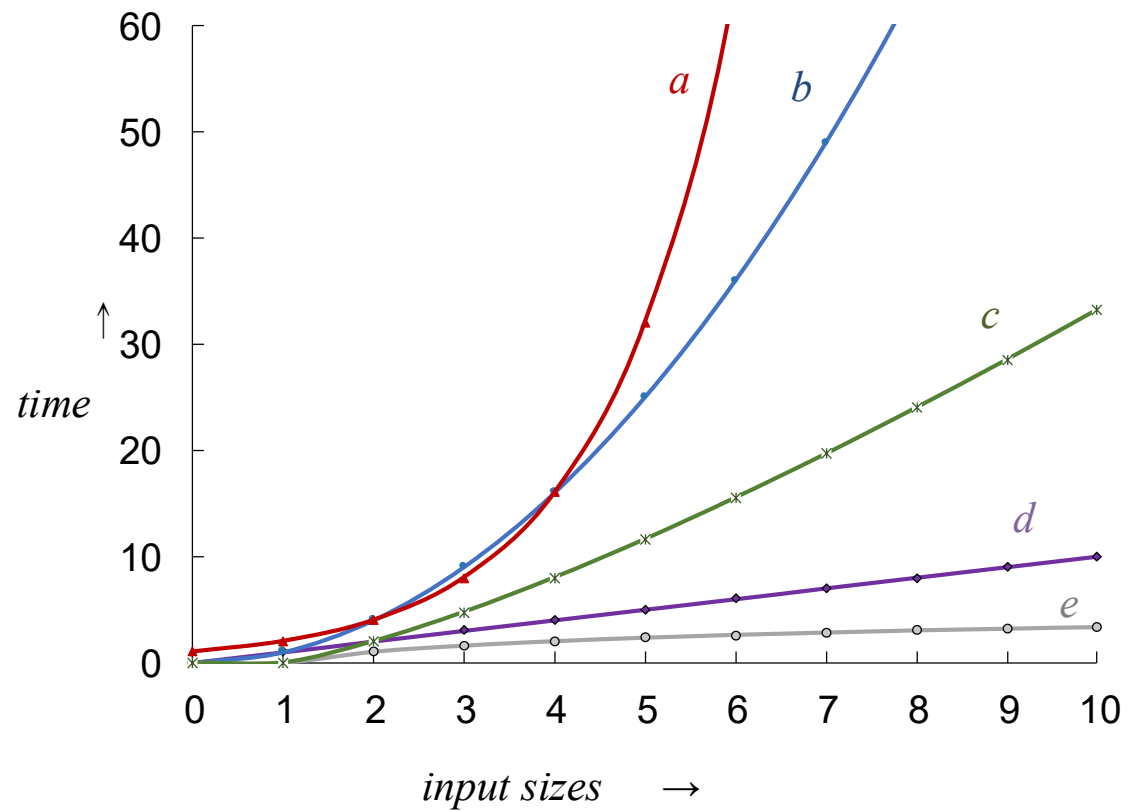
■ $n^4 + 1000 n^3 + n^2 = O(n^4), \text{ True}$

■ $n^4 + 1000 n^3 + n^2 = O(n^5), \text{ True}$

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

What Kinds of Growth Rates?



Running Times on Computers

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

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

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

?

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:

- $f_1(n) = O(n), f_2(n) = O(n^2)$

- $f_1(n) + f_2(n) =$

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

- $f_1(n) + f_2(n) =$

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)
}
```

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 $O(\log_2 n)$

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:

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

Notation: Theta(Θ)

■ Definition

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

■ Examples

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

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.