CHAPTER 1
BASIC CONCEPTS

Iris Hui-Ru Jiang

Basic Concepts

- **Contents**
  - System life cycle
  - Object-oriented design
  - Data abstraction and encapsulation
  - Algorithm specification
  - Performance analysis and measurement
- **Readings**
  - Chapter 1
  - C++
    - H. Deitel and P. Deitel, C++ How to Program, 5th Ed., Prentice Hall, 2005. ISBN: 0131971093.

System Life Cycle

Hierarchical approach

- **Regard a large-scale computer program as a system**
  - Programming >> coding

1. **Requirements**
   - Define specifications (input and output): vague $\rightarrow$ rigorous
2. **Analysis**
   - Divide into manageable pieces: top-down vs. bottom-up

     preferred
3. **Design**
   - Determine data objects (abstract data types) and operations (algorithms): language-independent
4. **Refinement and coding**
   - Implement: language-dependent
5. **Verification**
   - Correctness proofs: use formal (mathematics) techniques
   - Testing: include all possible scenarios; estimate performance
   - Error removal: debug based on above two; avoid spaghetti codes

4 Object-Oriented Design

Algorithmic vs. OO Decomposition

Divide -and- Conquer

- **Algorithmic (functional) decomposition** views software as a process
  - Decompose the software into steps
  - Data structures are a secondary concern
- **Object-Oriented decomposition** views software as a set of well-defined objects
  - Model entities in the application domain
  - Interact with each other to form a software system
  - Functional decomposition is addressed after the system has been decomposed into objects
- **Advantages of OO decomposition:**
  - Encourage reuse of software
  - Allow software to evolve as system requirements change
  - Is more intuitive: objects naturally model entities in the application domain

Object-Oriented Programming

- **Definition: An object is**
  - An entity that performs computations and has a local state
  - ⇒ Viewed as a combination of data and procedural elements
- **Definition: Object-oriented programming is a method of implementation in which**
  1. Objects are the fundamental building blocks
  2. Each object is an instance of some type (or class)
  3. Classes are related to each other by inheritance relationships
- **Definition: An object-oriented language**
  1. Supports objects
  2. Requires objects to belong to a class
  3. Supports inheritance
- **The first two features are considered as object-based**

7

Data Abstraction and Encapsulation

Example: DVD Players

- **Packaging: Control panel (remote control) of a DVD player:**
  ⏮ ⏪ ⏩ ⏭ ■ ⏸ ▶
  - Interact only through buttons
  - Encapsulation: hide internal representation from users
- **Usage: Instruction manual of the DVD player**
  - Does tell us what the DVD player does
  - Abstraction: describe what to do, not how to do

Data Abstraction & Encapsulation

- **Definition: Data encapsulation (information hiding)**
  - Conceal the implementation details of a data object from the outside world

Make a data object as a black box

- **Definition: Data abstraction**
  - Separate the specification of a data object and its implementation

What to do vs. how to do

Data Type

- **Definition: A data type is**
  1. A collection of objects, and
  2. A set of operations that act on those objects
- **Example: int**
  1. Objects: {0, +1, -1, +2, -2, …, MAXINT, MININT}
     - MAXINT: maximum integer on the computer
     - MININT: minimum integer on the computer
  2. Operations: {+, - , *, /, %, <<, >>, ==, !=, …}

Abstract Data Type (ADT)

- **Definition: An abstract data type (ADT) is**
  - A data type
  - The specification of the objects is separated from the representation of the objects
  - The specification of the operations on the objects is separated from the implementation of the operations

ADT =
abstraction + data type
$\Rightarrow$ Separate specification from implementation:
A specification can have several options of implementation

ADT Example: *NaturalNumber*

ADT *NaturalNumber* is
objects: An ordered subrange of the integers starting at 0, ending at MAXINT on the computer.
functions:
for all *x*, *y* ∊ *NaturalNumber*; TRUE, FALSE ∊ *Boolean*
and where +, -, <, ==, and = are the usual integer operations

Zero(): *NaturalNumber*	::=	0

IsZero(*x*): *Boolean*	::=	**if** (*x* == 0) *IsZero* = TRUE
		else *IsZero* = FALSE

Add(*x*, *y*): *NaturalNumber*	::=	**if** (*x*+*y* <= MAXINT) *Add* = *x* + *y*
		else *Add* = MAXINT

Equal(*x*, *y*): *Boolean*	::=	**if** (*x* == *y*) *Equal* = TRUE
		else *Equal* = FALSE

Successor(*x*): *NaturalNumber*	::=	**if** (*x* == MAXINT) *Successor* = *x*
		else *Successor* = *x* +1

Substract(*x*, *y*): *NaturalNumber*	::=	**if** (x < y) *Substract* = 0
		else *Substract* = *x* – *y*

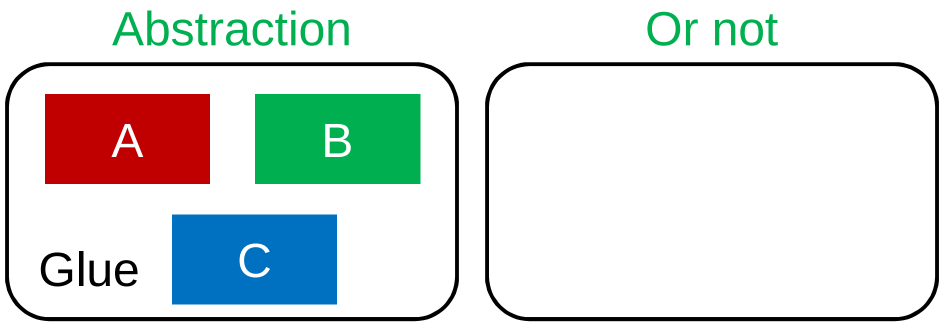
end *NaturalNumber*

::= is defined as

Later we use the syntax of C++ class to express an ADT

Pros of Data Abstraction & Encapsulation

- **Simplification of software development** (data abs)
  - Facilitate the decomposition of a complex task into simpler ones
    - Do not need to know how other portions are implemented
- **Testing and debugging** (data abs)
  - Test and debug separately



- **Reusability** (data abs & enc)
- **Modifications to the representation of a data type** (data enc)
  - Implementation of a data type is invisible to the outside world
  - It can manipulate the data type only through a suite of operations
  - A change in the internal representation of a data type will not affect the rest of the program as long as the operations are kept the same
  - What if without data enc?

14

Algorithm Specification

Algorithm

Problem-solving procedure

- **Definition: An algorithm is**
  - A finite set of instructions that accomplishes a particular task
- **Criteria:**
  - Input: may have
  - Output: must have
  - Definiteness: must be clear and unambiguous
  - Finiteness: terminate after a finite number of steps
  - Effectiveness: must be basic and feasible with pencil and paper

Task (problem)

Procedure (solution)

- **Cf. An algorithm is**
  - A well-defined procedure for transforming some input to a desired output [Cormen et al. Introduction to Algorithms, 2nd Ed.]
  - A well-defined procedure to solve a problem; a programmer turns the algorithm into a computer program [Sci-Tech Encyclopedia]

high-level $\longleftrightarrow$ low-level

Recap System Life Cycle

- **Regard a large-scale computer program as a system**
  - Programming >> coding

1. **Requirements**
   - Define specifications (input and output): vague $\rightarrow$ rigorous

Problem

2. **Analysis**
   - Divide into manageable pieces: top-down vs. bottom-up
3. **Design**
   - Determine data objects (abstract data types) and operations (algorithms): language-independent

Algorithm

4. **Refinement and coding**
   - Implement: language-dependent

Program

5. **Verification**
   - Correctness proofs: use formal (mathematics) techniques
   - Testing: include all possible scenarios; estimate performance
   - Error removal: debug based on above two; avoid spaghetti codes

Selection Sort

System life cycle: requirements

- **Problem: Sorting**
  - Devise a program that sorts a collection of $n \geq 1$ integers
  - Input: a sequence of n integers $<a_1, a_2, \ldots, a_n>$, $n \geq 1$
  - Output: a permutation $<a'_1, a'_2, \ldots, a'_n>$ s.t. $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

System life cycle: analysis & design

- **Solution: Selection sort**
  - From those integers that are currently unsorted, find the smallest and place it next in the sorted list. [in English]
  - Well-defined? Clear? Unambiguous?
    - Where and how are the integers initially sorted?
    - Where to place the results?

Selection Sort
- Algorithm & Program

System life cycle: analysis & design

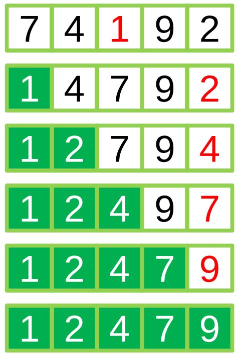
Algorithm

```
void SelectionSort (int *a, const int n)
{// Sort the n integers a[0] to a[n-1] into
      nondecreasing order
   for (int i = 0; i < n; i++)
   {
       examine a[i] to a[n-1] and suppose
           the smallest integer is at a[j];
       interchange a[i] and a[j];
   }
}
```

System life cycle: refinement & coding

Program

```
void SelectionSort (int *a, const int n)
{// Sort the n integers a[0] to a[n-1] into
      nondecreasing order
   for (int i = 0; i < n; i++)
   {
       int j = i;
       // find smallest integer in a[i] to a[n-1]
       for (int k = i + 1; k < n; k++)
           if (a[k] < a[j]) j = k;
       swap(a[i], a[j]);
   }
}
```



High-level
Language independent

$\longleftrightarrow$

Low-level
Language dependent

Theorem: *SelectionSort*(*a*, *n*) correctly sorts a set of $n \geq 1$ integers; the result remains in $a[0] \ldots a[n-1]$ such that $a[0] \leq a[1] \leq \ldots \leq a[n-1]$.

Q: How to prove it?

Binary Search

- **Problem: Searching**
  - Input
    - a sorted array of $n \geq 1$ distinct integers $a[0..n-1]$
    - integer x
  - Output
    - return j, if there exists j such that $x = a[j]$
    - return -1, otherwise
- **Solution:**
  - Compare one by one: Sequential search
    - Correct but slow
  - Better idea?
    - Hint: input is sorted
    - Divide-and-conquer: Binary search

Use known information to improve your solution

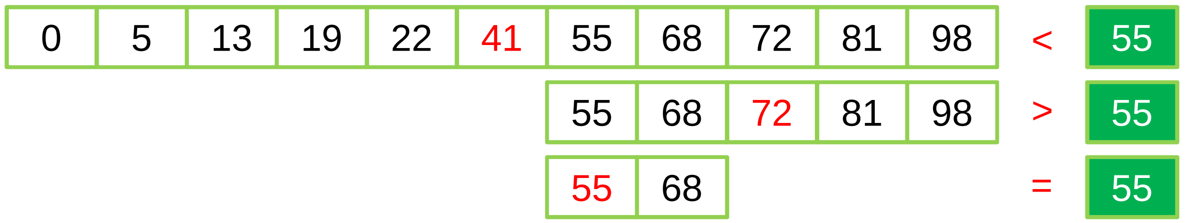
Binary Search
- Divide-and-Conquer

Divide-and-conquer paradigm

- **Divide** the problem into a number of subproblems
  - Similar but easier
- **Conquer** the subproblems
  - Solve them
- **Combine** the subsolutions to get the solution to the original problem

Binary search on a sorted array

- **Divide**: check the middle element
- **Conquer**: search the subarray
- **Combine**: trivial



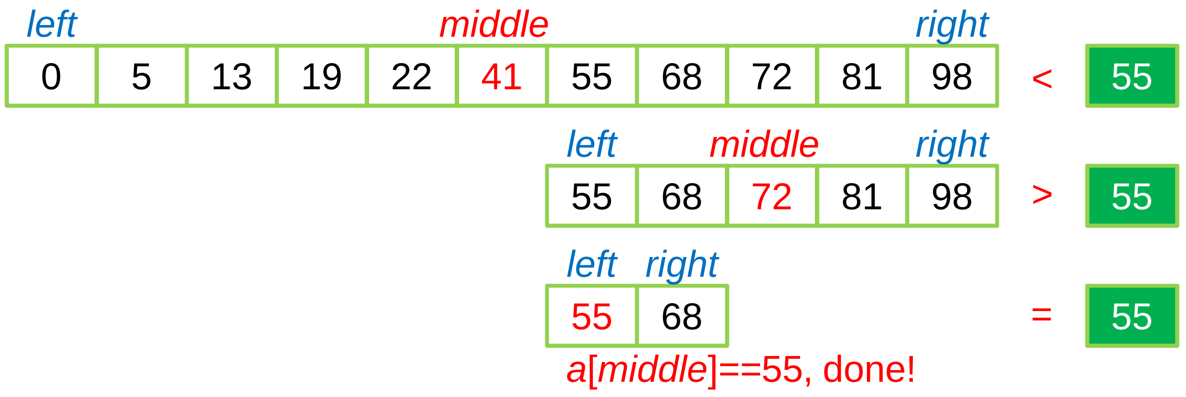
Binary Search
- Algorithm & Program

Algorithm

```
int BinarySearch (int *a, const int x, const int n)
{// Search the sorted array a[0], … , a[n-1] for x
    initialize left and right;
    while (there are more elements)
    {
        let middle be the middle element;
        if (x < a[middle]) set right to middle-1;
        else if (x > a[middle]) set left to middle+1;
        else return middle;
    }
    return -1; //not found
}
```

Program (Iterative)

```
int BinarySearch (int *a, const int x, const int n)
{// Search the sorted array a[0], … , a[n-1] for x
    int left = 0, right = n-1;
    while (left <= right)
    {// there are more elements
        int middle = ( left + right)/2;
        if (x < a[middle]) right = middle-1;
        else if (x > a[middle]) left = middle+1;
        else return middle;
    } //end of while
    return -1; //not found
}
```



Recursive Algorithms

Mathematical induction

- **Definition: A recursive function is**
  - A function that invokes itself before it is done
    - Direct (call by itself) vs. indirect (call through others)
- **Why to use recursion?**
  - Often can express a complex process very clearly
- **When to use recursion?**
  - The problem itself is recursively defined
    - e.g., factorials *n*!, Fibonacci sequence $F_n = F_{n-1} + F_{n-2}$
  - assignment+**if-else**+while ⇔ assignment+**if-else**+recursion
  - e.g., divide-and-conquer approach
- **How to develop recursion?**
  1. Terminating condition (basis)
     - Trivial cases; directly computes the output
  2. Recursion (induction hypothesis)
     - If $n=k-1$, the statement holds, check $n=k$?

Binary Search
- Recursion

Faster

More intuitive and elegant

Program (Iterative)

```
int BinarySearch (int *a, const int x, const int n)
{// Search the sorted array a[0], … , a[n-1] for x
    int left = 0, right = n-1;
    while (left <= right)
    {// there are more elements
        int middle = (left + right)/2;
        if (x < a[middle]) right = middle-1;
        else if (x > a[middle]) left = middle+1;
        else return middle;
    } //end of while
    return -1; //not found
}
```

Program (Recursive)

```
int BinarySearch (int *a, const int x, const int
      left, const int right)
{// Search the sorted array a[left], … , a[right] for x
    if (left <= right) {
        int middle = (left + right)/2;
        if (x < a[middle]) {
            return BinarySearch(a, x, left, middle-1);
        } else if (x > a[middle]) {
            return BinarySearch(a, x, middle+1, right);
        } else return middle;
    } //end of if
    return -1; //not found
}
```

Start with *BinarySearch*(*a*, *x*, 0, *n*-1)

Permutation Generator

- **Problem:**
  - Input: a set of $n \geq 1$ elements
  - Output: all possible permutations of this set
  - e.g., input: $\{a,b,c\} \Rightarrow$ output: $\{(a,b,c), (a,c,b), (b,a,c), (b,c,a), (c,a,b), (c,b,a)\}$
- **Solution:**
  - Given $\{a,b,c\}$, the answer can be constructed by writing
    - a followed by all permutations of $\{b, c\}$
    - b followed by all permutations of $\{a, c\}$
    - c followed by all permutations of $\{a, b\}$
  - If we can solve n-1 elements, we then can solve n elements

Permutation Generator

- Recursion

```
void Permutations (char *a, const int k, const int m)
{// Generate all the permutations of a[k], …, a[m]
    if (k == m) { // output permutation
        for (int i = 0; i <= m; i++) cout << a[i] <<" ";
        cout << endl;
    }
    else // a[k:m] has more than one permutation. Generate these recursively.
        for (i = k; i <= m; i++)) {
            swap(a[k], a[i]);
            Permutations(a, k+1, m);
            swap(a[k], a[i]);
        }
}
```

Start with *Permutations*(*a*, 0, *n*-1)

26 Performance Analysis and Measurement

Complexity

- **Definition: The space complexity of a program is**
  - The amount of memory it needs
  - $\Rightarrow$ Storage requirements
- **Definition: The time complexity of a program is**
  - The amount of computer time it needs
  - $\Rightarrow$ Computing time or runtime
- **Performance evaluation**
  - Performance analysis: a priori estimates
  - Performance measurement: a posteriori testing

Space Complexity

- **The space complexity $S(P)$ of a program P: $S(P) = c + S_P$**
  - c: constant, including instruction space, space for simple variables and fixed-size component variables, space for constant
  - S_P: instance characteristics, e.g., the input size
    - Referenced variables, recursion stack space

```
float Abc(float a, float b, float c)
{
   return a + b + b*c + (a + b - c)/(a + b) + 4.0;
}
```

$S_{Abc} = 0$

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

$S_{Sum}(n) = 0$

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

$S_{Rsum}(n) = 4*(n+1)$

Each call of *Rsum* requires 4 words for (1) value of n, (2) value of a, (3) returned value, and (4) return address

Time Complexity

- **The time complexity $T(P)$ of a program P: $T(P) = t_P$**
  - t_P: instance characteristics
  - The number of program steps
    - A program step is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.
  - $\Rightarrow$ We wish to know how the runtime increases as the number of inputs increases.
- **How to determine the step count?**
  - Method I: Introduce a new variable, *count*, into the program
  - Method II: Build a step table

Step Count
- Method I (1/2)

Adding *count* into *Sum*

```
float Sum(float *a, const int n)
{
    float s = 0;
    count++; // count is global
    for (int i = 0; i<n; i++) {
        count++; // for for
        s += a[i];
        count++; // for assignment
    }
    count++; // for last time of for
    count++; // for return
    return s;
}
```

Simplified version with *count*

```
void Sum(float *a, const int n)
{
    for (int i = 0; i<n; i++)
        count += 2;
    count += 3;
}
```

$t_{Sum}(n) = 2n + 3$

Step Count
- Method I (2/2)

Adding *count* into *Sum*

```
float Sum(float *a, const int n)
{
    float s = 0;
    count++; // count is global
    for (int i = 0; i<n; i++) {
        count++; // for for
        s += a[i];
        count++; // for assignment
    }
    count++; // for last time of for
    count++; // for return
    return s;
}
```

$t_{Sum}(n) = 2n + 3$

Adding *count* into *Rsum*

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

$t_{Rsum}(n) = 2n + 2$

$$\begin{aligned} t_{Rsum}(n) &= 2 + t_{Rsum}(n\text{-}1),\ t_{Rsum}(0)=2 \\ &= 2 + 2 + t_{Rsum}(n\text{-}2) \\ &= \ldots = 2n + t_{Rsum}(0) \\ &= 2n + 2 \end{aligned}$$

Is *Rsum* faster?

Step Count
- Method II (1/2)

Matrix addition

```
line void Add(int **a, int **b, int **c, int m, int n)
1    {
2        for (int i = 0; i < m; i++)
3            for (int j = 0; j < n; j++)
4                c[i][j] = a[i][j] + b[i][j];
5    }
```

Step table

line	s/e	frequency	total steps
1	0	1	0
2	1	m+1	m+1
3	1	m(n+1)	m(n+1)
4	1	mn	mn
5	0	1	0
total number of steps			2mn+2m+1

s/e: steps per execution
frequency: times executed

Step Count
- Method II (2/2)

Adding *count* into *Sum*

```
float Sum(float *a, const int n)
1   {
2       float s = 0;
3        for (int i = 0; i<n; i++)
4            s += a[i];
5        return s;
6   }
```

Step table

line	s/e	frequency	total steps
1	0	1	0
2	1	1	1
3	1	*n*+1	*n*+1
4	1	*n*	*n*
5	1	1	1
6	0	1	0
total number of steps			2*n*+3

s/e: steps per execution
frequency: times executed

Is It Really that Simple?

- **Time complexity = step count**
  - A function of instance characteristics
- **Each of *Add*, *Sum*, *Rsum* has the same time complexity for all instances of the same *n***
- **How about *BinarySearch*?**
  - Vary with the position of *x* in *a*
- **Typically, consider three kinds**
  - Best-case
  - Worst-case
  - Average-case

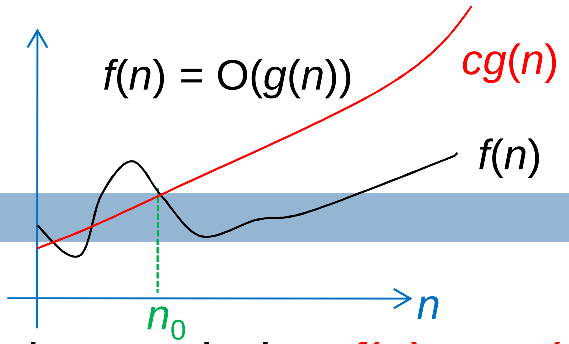
Asymptotic Notation

- **Motivation:**
  - To compare the time complexities of two programs that solve the same problem
  - To predict the growth in runtime as the instance characteristics change
- **For the same problem, which is faster?**
  - $T_A(n) = 100n+10$; $T_B(n) = 3n+3 \Rightarrow$ It depends!
    - Since the notion of a step count is itself inexact…
  - $T_A(n) = 100n+10$; $T_B(n) = 3n^2+3 \Rightarrow A$ is faster for large n

Asymptotic Notation

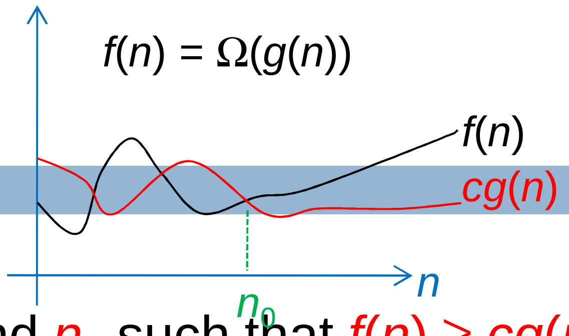
- **Types of bounding functions:**
  - O: upper bound
  - Ω: lower bound
  - Θ: tight bound

Asymptotic Notation
- Big Oh



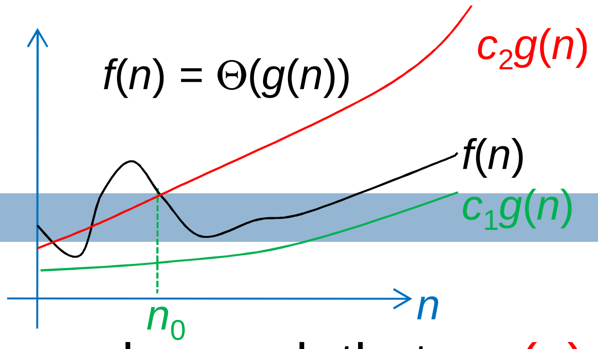
- **Definition: $f(n) = O(g(n))$ iff**
  - there exist positive constants c and n_0 such that $f(n) \le cg(n)$ for all n, $n \ge n_0$
  - $\Rightarrow$ $g(n)$ is an upper bound of $f(n)$ if we ignore c and small n
- **Example**
  - $3n+2=O(n)$ $\Rightarrow$ $3n+2 \le 4n$ for $n \ge 2$
  - $6*2^n+n^2=O(2^n)$ $\Rightarrow$ $6*2^n+n^2 \le 7*2^n$ for $n \ge 4$
  - $3n+3=O(n^2)$ $\Rightarrow$ $3n+3 \le 3n^2$ for $n \ge 2$
    - Correct but...
    - To be informative, $g(n)$ should be as small as possible
  - Q: $3n^2 + n = O(n^2)$? Yes
  - Q: $3n^2 + n = O(n)$? No
  - Q: $3n^2 + n = O(n^3)$? Yes

Asymptotic Notation
- Omega



- **Definition: $f(n) = \Omega(g(n))$ iff**
  - there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all n, $n \geq n_0$
  - $\Rightarrow g(n)$ is a lower bound of $f(n)$ if we ignore c and small n
  - To be informative, $g(n)$ should be as large as possible
- **Example:**
  - Q: $3n^2 + n = \Omega(n^2)$? Yes
  - Q: $3n^2 + n = \Omega(n)$? Yes
  - Q: $3n^2 + n = \Omega(n^3)$? No

Asymptotic Notation
- Theta



- **Definition: $f(n) = \Theta(g(n))$ iff**
  - there exist positive constants c_1, c_2 and n_0 such that $c_1g(n) \leq f(n) \leq c_2g(n)$ for all n, $n \geq n_0$
  - $\Rightarrow$ $g(n)$ is a tight bound of $f(n)$ if we ignore c_1, c_2 and small n
- **Example**
  - Q: $3n^2 + n = \Theta(n^2)$? Yes
  - Q: $3n^2 + n = \Theta(n)$? No
  - Q: $3n^2 + n = \Theta(n^3)$? No

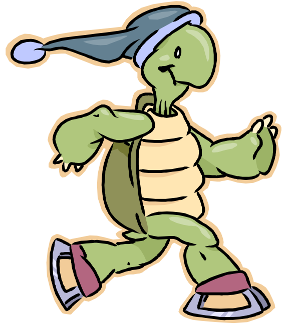
Polynomial-Time Complexity

- **Polynomial-time complexity O(p(n))**
  - n is the input size
  - $p(n)$ is a polynomial function of n ($p(n) = n^{O(1)}$)
- **Order**
  - O(1): constant
  - O(log$n$): logarithmic
  - O($n^{0.5}$): sublinear
  - O(n): linear
  - O($n$log$n$): loglinear
  - O(n^2): quadratic
  - O(n^3): cubic
  - O(n^4): quartic
  - O(2^n): exponential
  - O(n!): factorial
  - O(n^n)

Faster

$\updownarrow$

Slower

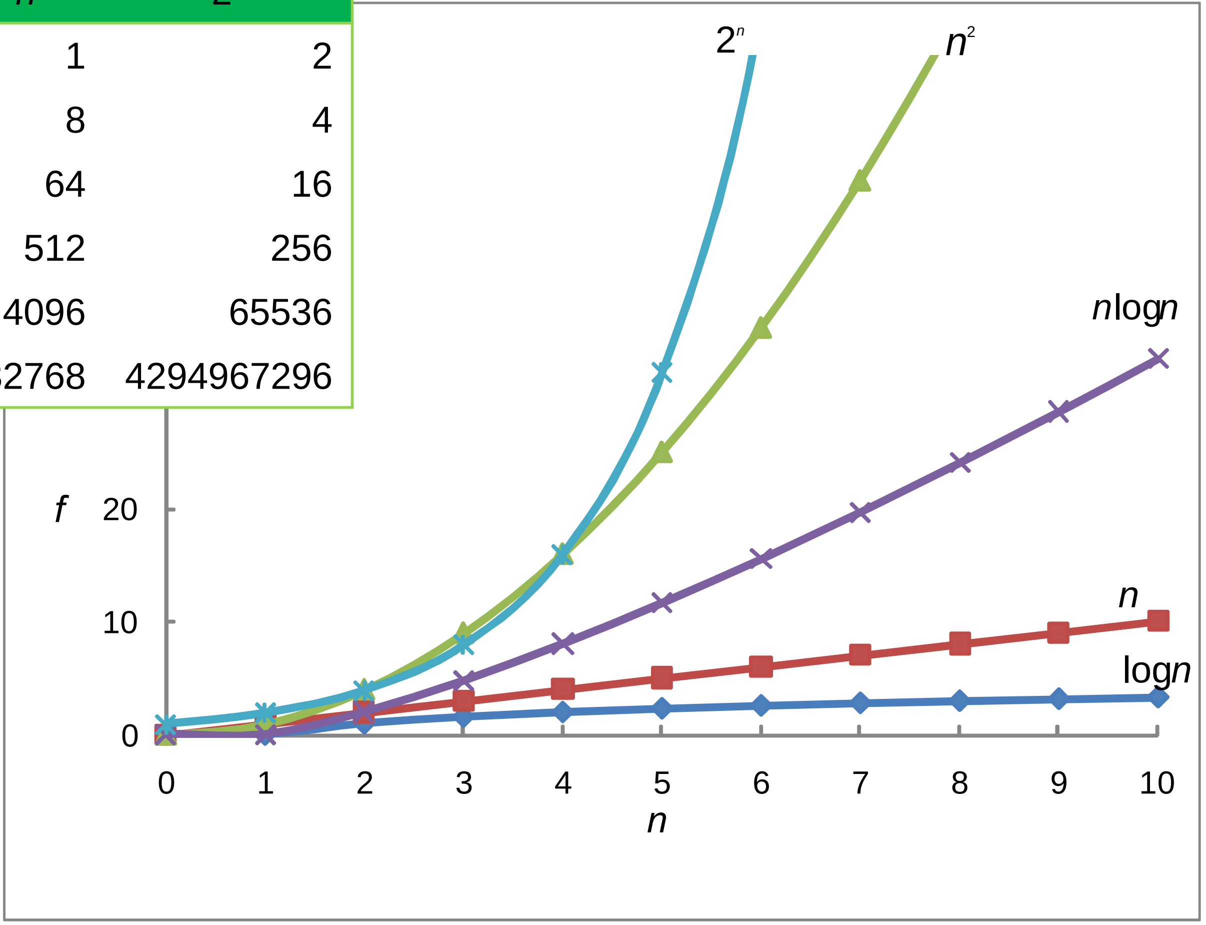


Function Growth

Function values

Predicted curves

log n	n	n log n	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296



log = $\log_{10}$
lg = $\log_2$
ln = $\log_e$

Time on a 1-billion-steps-per-sec Computer

	$f(n)$						
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	13d
50	.05 μs	.28 μs	2.5 μs	125 μs	6.25ms	3.1y	$4*10^{13}$y
100	.10 μs	.66 μs	10 μs	1ms	100ms	3171y	$32*10^{283}$y
10^3	1 μs	9.96 μs	1 ms	1s	115.7d	$3.17*10^{13}$y	
10^4	10 μs	130 μs	100 ms	11.57d	3171y	$3.17*10^{23}$y	
10^5	100 μs	1.66 ms	10s	31.71y	$3.17*10^7$y	$3.17*10^{33}$y	
10^6	1ms	19.92ms				$3.17*10^{43}$y	

$\mu = 10^{-6}$ m = 10^{-3}

s = sec m = min h = hour d = day y = year

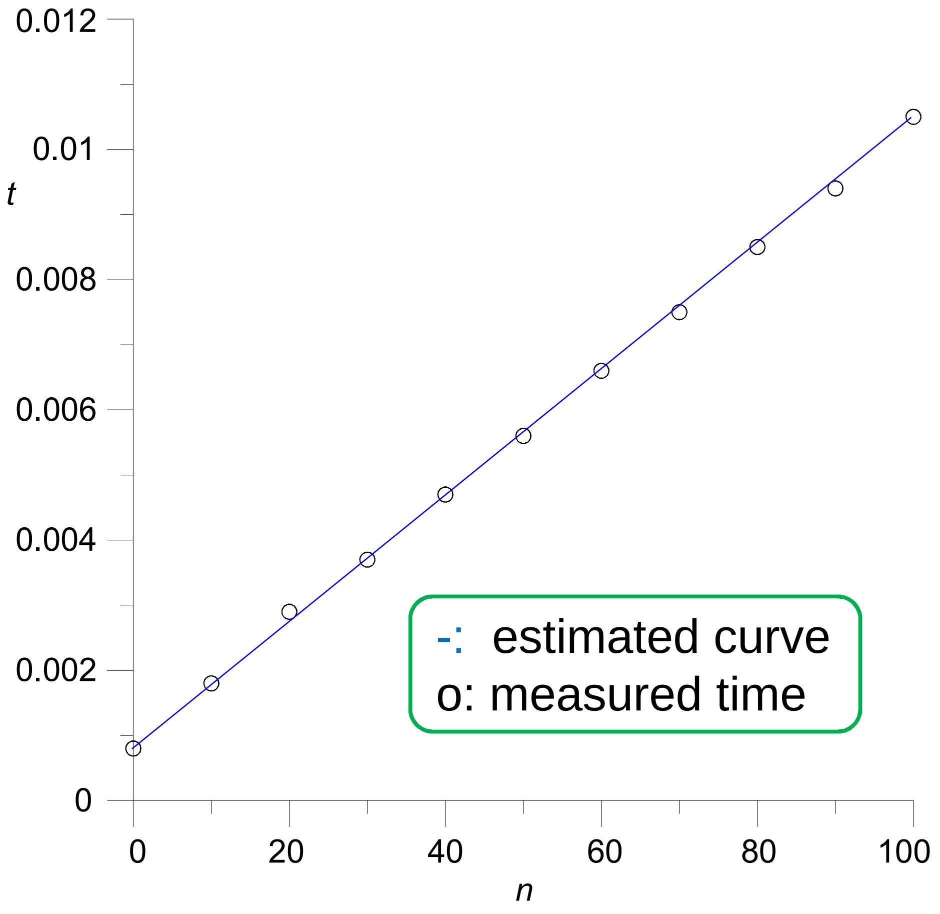
Performance Measurement

Analysis vs. measurement

- **Sequential search**
  1. Time complexity: Θ(n)
  2. Real runtime can be measured by *time()* in C++

1. **Asymptotic analysis**
   - Works only for sufficiently large values of *n*
   - Forgets coefficients
2. **The measured time plot**
   - May not lie exactly on the predicted curve due to the effects of low-order terms

Time plot of sequential search


t
n
-: estimated curve
o: measured time