CSIEB0100 Data Structures

Lecture01 Basic Concepts

Shiow-yang Wu 吳秀陽
Department of Computer Science and Information Engineering
National Dong Hwa University

Lecture material is mostly home-grown, partly adapted from slides came with the textbook originally prepared by Professor Jiun-Long Huang of NCTU.

What The Course Is About

- Data structures is concerned with the representation and manipulation of data.
- All programs manipulate data.
- So, all programs need to represent data in some way.
- All programs need data structures.
- Data manipulation requires an algorithm.

What The Course Is About

- We shall study structures to represent data and algorithms to manipulate these structures
- The study of data structures is fundamental to Computer Science & Engineering

Prerequisites

- C++
  - Most examples will be presented in C++
- Asymptotic Complexity
  - Big Oh, Theta, and Omega notations
- We will provide short reviews for both topics

Course Web Page

- http://web.csie.ndhu.edu.tw/showyang/DS2023f/index.html
- Handouts, syllabus, textbooks, source codes, exercises, lectures, assignments, TAs, etc.

- **Office**: Sci & Eng Building II C308

Assignments

- All assignments will be given in the class.
- Submit your programs and test data/result to the TA leader. (Detail instruction will be given in the class Web page)

- Do Assignment 0 by next week

Grades

Assignments	35%
Midterm Exam	35%
Final Exam	35%

- Yes! 105% !!
- As long as you work hard, it's hard to fail in this class.

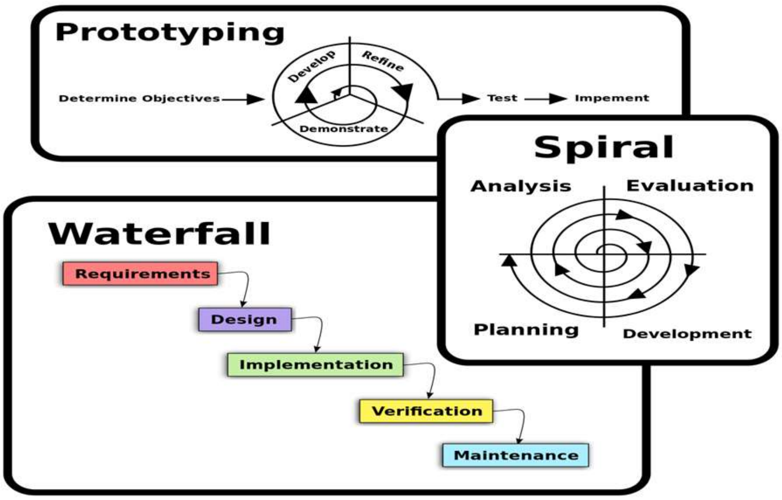
Assignment 0: Getting Started

- Download Code::Blocks and install it on your own machine by following the instructions along with the file.
- Download the source code of the textbook samples from the book's web site.
- Unfortunately, the code doesn't work out of the box.
- Assignment 0 is to make all code of Chapter 1 work correctly.
- Submit any one of the corrected code to the TA.

System Life Cycle

- Large-scale programs are considered as systems with many complex interacting components.
- The development process of such programs is known as the system/software life cycle.
- Different development methodologies lead to different software life cycles.
- The goal is to find repeatable, predictable processes that improve software productivity and quality.
- No de-facto standard yet!

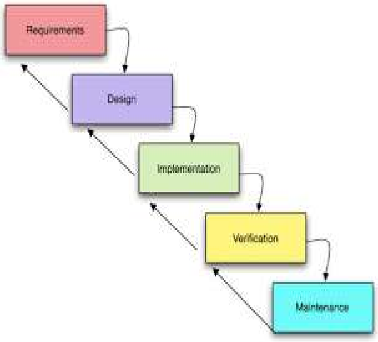
Three Traditional Models



(https://courses.lumenlearning.com/sanjacinto-computerapps/chapter/reading-software-development-process/)

Waterfall Model 1/2

- **Requirements**
  - Project goals
  - Input/Output specification
- **Analysis**
  - Bottom-up
  - Top-down
- **Design**
  - Data objects: abstract data types
  - Operations: specification & design of algorithms

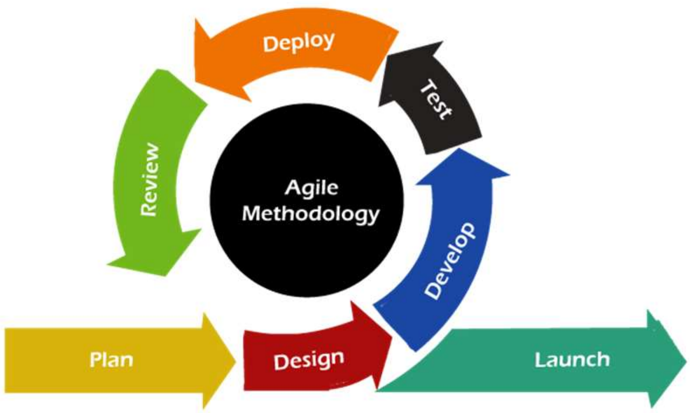


Waterfall Model 2/2

- **Refinement & Implementation**
  - Choose representations for data objects
  - Write algorithms for operations on data objects
  - Write programs that implement the algorithms
- **Verification**
  - Correctness proofs: selecting proved algorithms
  - Testing: correctness & efficiency
  - Error removal (debugging): much easier with well-designed and well-documented code
- **Maintenance**
  - Modification to correct faults, improve performance or other attributes.

Agile Model

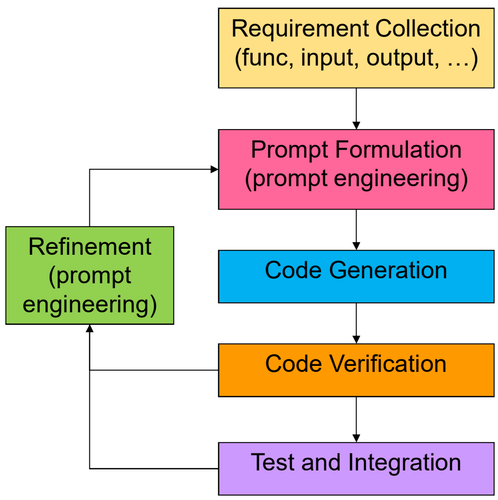
- Flexible and responsive software development for fast adaptation to changes.



(https://www.javatpoint.com/agile-vs-waterfall-model)

S/W Development with AI

- AI-assisted development is a widespread trend.
- No proven and well-accepted model yet.
- Use cases and best practices are accumulating.
- Very popular for generating code snippets.



Quality Evaluation

- Meet the original requirement specification?
- Work correctly?
- Document?
- Use functions to create logical units?
- Code readable?
- Use storage efficiently?
- Running time acceptable?

Data Abstraction & Encapsulation

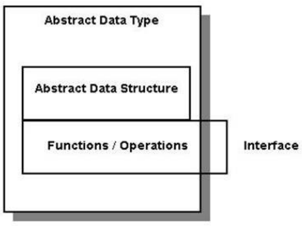
- **Data encapsulation** or **information hiding** is the concealing of the implementation details of a data object from the outside world
- **Data abstraction** is the separation between the specification of a data object and its implementation
- A **data type** is a collection of objects and a set of operations that act on those objects

Specification vs Implementation

- **Specification**: Description of data or function without representation or implementation details.
- **Representation**: Detail description of how data should be represented in a computer program.
  - E.g., char 1 byte, int 4 bytes
- **Implementation**: Detail description of how to realize the specification of data or function with computer programs.
- Separation of specification and implementation is the key to the systematic study of data structures.
- It is possible to have different implementations of the same specification.

Abstract Data Type (ADT)

- An **abstract data type** (**ADT**) is a data type that is organized such that
  - the specification of the objects & operations
  - is separated from the representation of the objects and the implementation of the operations
- ADT is implementation-independent



Abstract data type NaturalNumber (p.9)

```
ADT NaturalNumber is
  objects: an ordered subrange of the integers starting at
      zero and ending at the maximum integer (INT_MAX) on
      the computer
  functions:
      for all x, y ∈ Nat_Number; TRUE, FALSE ∈ Boolean
      and where +, -, <, and == are the usual integer
      operations.
      Zero ( ):NaturalNumber          ::=  0
      Is_Zero(x):Boolean              ::= if (x) return FALSE
                                          else return TRUE
      Add(x, y):NaturalNumber         ::= if ((x+y) <= INT_MAX)
                                            return x+y
                                          else return INT_MAX
      Equal(x,y):Boolean              ::= if (x== y) return TRUE
                                          else return FALSE
      Successor(x):NaturalNumber      ::= if (x == INT_MAX)
                                            return x
                                          else return x+1
      Subtract(x,y):NaturalNumber     ::= if (x<y) return 0
                                          else return x-y
end Natural_Number
```

Interface

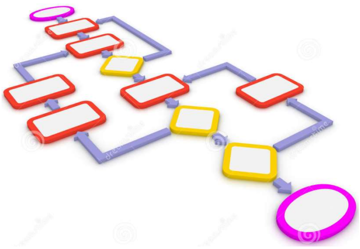
Behavior(semantics)

Algorithm Specification

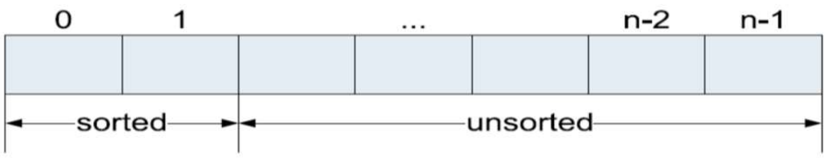
- **Definition**: An algorithm is a finite set of instructions that accomplishes a particular task.
- Criteria that ALL algorithms must satisfy:
  - **Input**: zero or more
  - **Output**: at least one
  - **Definiteness**: clear and unambiguous
  - **Finiteness**: terminate after a finite number of steps
  - **Effectiveness**: instruction is basic enough to be carried out

Algorithm vs Program

- One difference between an algorithm and a program is that the latter does not have to satisfy the fourth condition
  - Program doesn't have to be finite
  - E.g., OS scheduling



Example 1: Selection Sort

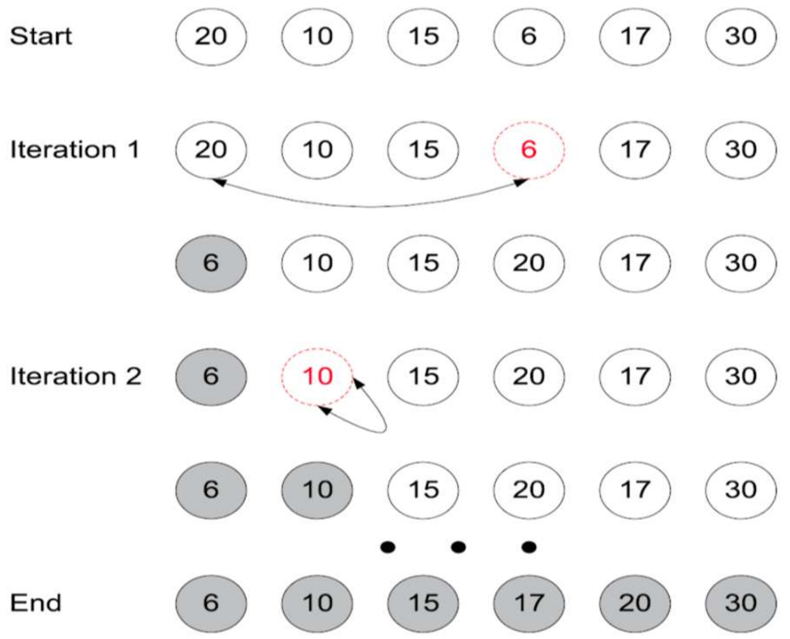


- From those integers that are currently unsorted, find the smallest and place it next in the sorted list.

```
for ( i=0; i<n; i++) {
  Examine list[i] to list[n-1] and suppose
    that smallest integer is list[min]
  Interchange list[i] & list[min]
}
```

Selection Sort

```
void sort (int *a, int n)
// sort n integers a[0] to a[n-1] into nondecreasing order
{
  for ( int i = 0; i < n; i++)
  {
    int j = i;  // a[j] is the current smallest int
    // find smallest KeyType in a[i] to a[n-1]
    for (int k = i+1; k < n; k++)
        if (a[k] < a[j]) j = k;  // update j to the new smallest
    int temp = a[i]; a[i] = a[j]; a[j] = temp;  // swap
  }
}
```

Selection Sort Animation

- http://liveexample.pearsoncmg.com/liang/animation/web/SelectionSort.html
- The site: https://liveexample.pearsoncmg.com/liang/animation/
- Provides many animations for other data structures and algorithms.

Example 2: Binary Search


```
while (there are more integers to check)
{
  middle = (left + right) /2;
  if (searchnum < list[middle])
    right = middle -1;
  else if (searchnum == list[middle])
    return middle;
  else
    left = middle+1;
}
```

Binary Search in C++

```
char compare (int x, int y)
{
  if (x > y) return '>';
  else if (x < y) return '<';
       else return '=';
}
```



```
int BinarySearch (int *a, int x, const int n)
// Search the sorted array a[0], ..., a[n-1] for x
{
 for (int left = 0,  right = n - 1; left <= right;) { // more elements
   int middle = (left + right)/2;
   switch (compare (x, a[middle])){
      case '>': left = middle + 1; break; // x > a[middle]
      case '<': right = middle - 1; break; // x < a[middle]
      case '=': return middle; // x == a[middle]
   } // end of switch
 } // end of for
 return -1; // not found
} // end of BinarySearch
```

Binary Search Examples

- Input
  - 1 3 7 9 13 20 31
- Search for 7 (next slide)
- Search for 16 (exercise)

Search for 7

Binary Search Animation

- http://liveexample.pearsoncmg.com/liang/animation/web/BinarySearch.html

Binary vs Sequential Search

- Comparison between sequential search and binary search
  - Binary search is faster than sequential search
  - However, binary search requires the input to be sorted in advance
- Should we always use binary search?
  - Not necessary.

Example 3: Selection Problem

- Selection problem: select the k-th largest among N numbers
- Approach 1
  - Read N numbers into an array
  - Sort the array in decreasing order
  - Return the element in position k

Example 3: Selection Problem

- Approach 2
  - Read k elements into an array
  - Sort them in decreasing order
  - For each remaining elements, read one by one
  - Ignored if it is smaller than the k-th element
  - Otherwise, place in correct place and kick one out of the array (which one?)
  - Return the last (kth) element of the array after all elements have been processed

Example of Approach 2

- Input
  - 20 9 15 6 17 30
- Find the 3rd largest number
- Read three numbers and sort them in descending order
  - 20 15 9
- Read next: “6”
  - 20 15 9

Example of Approach 2

- Read next: “17”
  - 20 17 15
  - 9 is out
- Read next: “30”
  - 30 20 17
  - 15 is out
- Finish processing of all numbers.
- The third largest number is 17.

- Does it work all the time?

Comparison of Approach 1 & 2

- Which one is better?
  - Implementation difficulty
  - Efficiency
    - Time complexity analysis
- Remember that time complexity is not the only yardstick
  - Space complexity
  - Easy to implement
  - …

Recursive Algorithms

- Recursion is usually used to solve a problem in a "divided-and-conquer" manner
- Direct recursion
  - Functions that call themselves
- Indirect recursion
  - Functions that call other functions that invoke calling function again
- C(n,m) = n!/[m!(n-m)!]
  - C(n,m)=C(n-1,m)+C(n-1,m-1) // why?
- Boundary condition for recursion

Recursive Summation

- sum(1, n)=sum(1, n-1)+n
- sum(1, 1)=1

```
int sum(int n)
{
  if (n==1)
    return (1);
  else
    return(sum(n-1)+n);
}
```


Recursive Factorial

- n!=n×(n-1)!
- fact(n)=n×fact(n-1)
- 0!=1

```
int fact(int n)
{
  if ( n== 0)
    return (1);
  else
    return (n * fact(n-1) );
}
```

Recursive Multiplication

- a×b=a×(b-1)+a
- a×1=a

```
int mult(int a, int b)
{
  if ( b==1)
    return (a);
  else
    return( mult(a,b-1) + a);
}
```

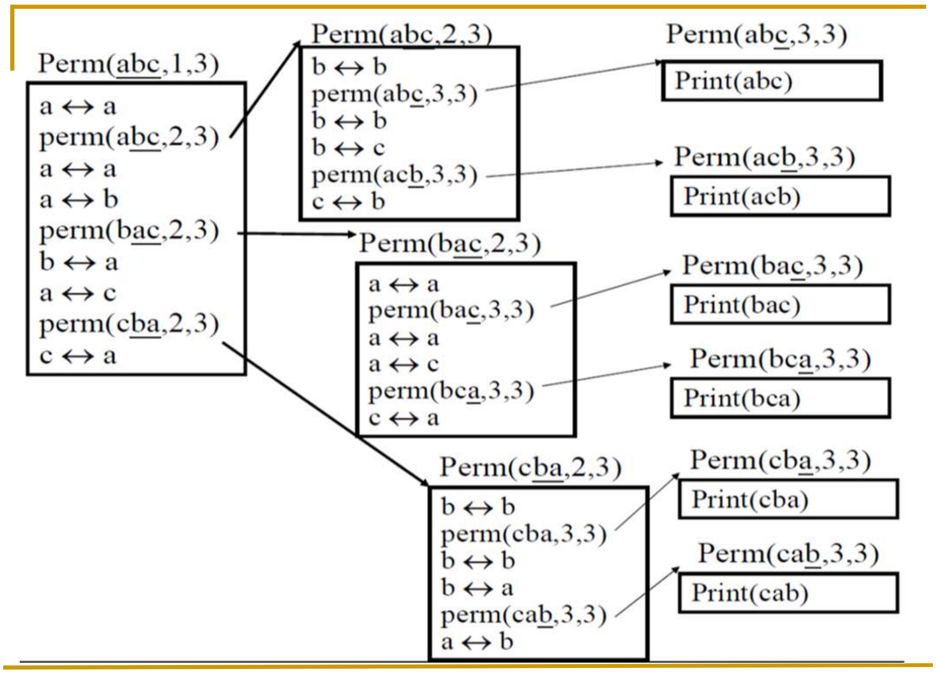

Recursive Binary Search

```
int BinarySearch (int *a, int x, const int left, const int right)
//Search the sorted array a[left], ..., a[right] for x
{
  if (left <= right) {
    int mid = (left + right)/2;
    switch (compare (x, a[mid])){
       case '>': return BinarySearch(a, x, mid+1, right); // x > a[mid]
       case '<': return BinarySearch(a, x, left, mid-1);  // x < a[mid]
       case '=': return mid; // x == a[mid]
    } // end of switch
  } // end of if
  return -1;// not found
}// end of BinarySearch
```

Recursive Permutation

- Permutation of {a, b, c}
  - (a, b, c), (a, c, b)
  - (b, a, c), (b, c, a)
  - (c, a, b), (c, b, a)
- Recursion?
  - a+Perm({b,c})
  - b+Perm({a,c})
  - c+Perm({a,b})


```
void perm (char *a, const int k, const int n) // n is the size of a
// Generate all the permutations of a[k], ..., a[n-1].
{
   if (k == n-1) {  // output permutation
      for (int i = 0; i < n; i++) cout << a[i] << " ";
      cout << endl;
   }
   else { // a[k], ..., a[n-1] has more than one permutation.
             // Generate these recursively
      for (int i = k; i < n; i++) {
         // swap a[k] and a[i]
         char temp = a[k]; a[k] = a[i]; a[i] = temp;
         perm(a, k+1, n); // all permutations of a[k+1], ..., a[n-1]
         // swap a[k] and a[i] back to original
         temp = a[k]; a[k] = a[i]; a[i] = temp;
      }
   } // end of else
} // end of perm
// Can we improve the code above?
```

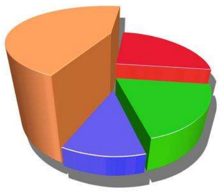


Performance Evaluation

- Criteria
  - Is it correct?
  - Is it efficient?
  - Is it readable?
- Performance analysis
  - Machine independent
- Performance measurement
  - Machine dependent

Performance Analysis

- Complexity theory
- Space complexity
  - Amount of memory
- Time complexity
  - Amount of computing time



Space Complexity

- $S(P) = c + S_p(I)$
  - c: fixed space (instruction, simple variables, constants)
  - $S_p(I)$: depends on characteristics of instance I
  - Characteristics: number, size, values of I/O associated with I
- If n is the only characteristic, $S_p(I) = S_p(n)$

Space Complexity Examples

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

$S_{abc}(I) = 0$

```
float sum(float list[ ], int n)
{
  float tempsum = 0;
  int i;
  for (i=0; i<n; i++)
     tempsum += list [i];
  return tempsum;
}
```

$S_{sum}(I) = 0$
Recall: pass the address of the first element of the array & pass by value

Space Complexity Examples

```
float rsum(float list[ ], int n)
{
  if (n)
    return rsum(list, n-1) + list[n-1];
  return 0;
 }
```

$S_{sum}(I)=S_{sum}(n)=6n$

Assumptions:
Space needed for one recursive call of the program

Type	Name	Number of bytes
Parameter: float Parameter: integer Return address: (used internally)	list[ ] n	2 2 2 (unless a far address)
Total		6

Time Complexity

- $T(P) = c + T_p(I)$
  - c: compile time
  - $T_p(I)$: program execution time
    - Depends on characteristics of instance I
- Predict the growth in run time as the instance characteristics change

Time Complexity

- Compile time (c)
  - Independent of instance characteristics
- Run (execution) time T_P
- A program step is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.
- Time complexity can be measured by counting the total number of steps required.

Methods to Compute Step Count

- Introduce variable count into programs
- Tabular method
- Determine the total number of steps contributed by each statement

  step per execution × frequency
- Add up the contribution of all statements

Step Count with count Variable

```
float sum(float list[ ], int n)
{
  float tempsum = 0;
  count++;        /* for assignment */
  int i;
  for (i=0; i<n; i++) {
    count++;      /* for the for loop */
    tempsum += list[i];
    count++;      /* for assignment */
  }
  count++;        /* last execution of for */
  return tempsum;
  count++;        /* for return */
}
```

2n+3 steps

Step Count with count Variable

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

T(n)
=2+T(n-1)
=2+2+T(n-2)
...
=2n+T(0)
=2n+2

Tabular Method

Statement	s/e	Frequency	Total steps
`float sum(float list[ ], int n)`			
`{`	0	1	0
`float tempsum = 0;`	1	1	1
`for(int i=0; i <n; i++)`	1	n+1	n+1
`tempsum += list[i];`	1	n	n
`return tempsum;`	1	1	1
`}`	0	1	0
Total			2n+3

s/e: steps per execution

Time Complexity

- Cases
  - Worst case
  - Best case
  - Average case
- Worst case and average case analysis is much more useful in practice

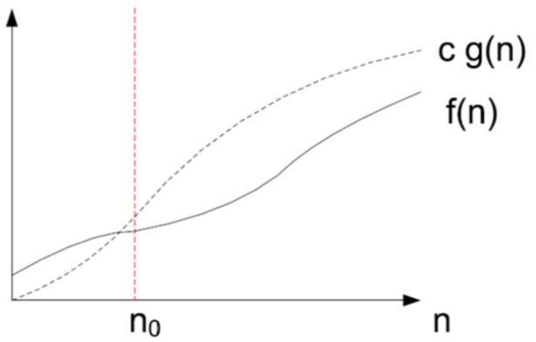
Time Complexity

- Difficult to determine the exact step counts
- What a step stands for is inexact
  - e.g. x := y v.s. x := y + z + (x/y) + …
- Exact step count is not useful for comparison
- Step count doesn't tell how much time a step takes
- Just consider the growth in run time as the instance characteristics change

Asymptotic Notation – Big O

- f(n)=O(g(n)) iff

∃ a real constant $c > 0$ and an integer constant $n_0 \geqq 1$, ∋ $f(n) \leq cg(n)$ ∀n, $n \geq n_0$



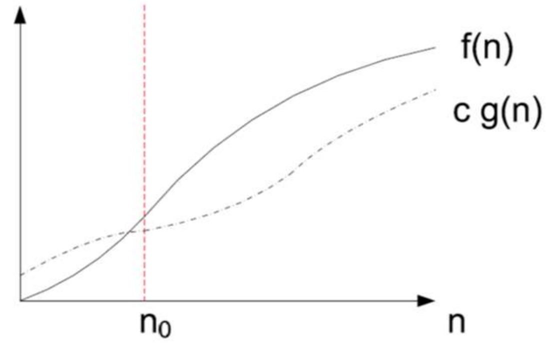
Big O Examples

- Examples
  - $3n+2 = O(n)$
    $3n+2 \leq 4n$ for all $n \geq 2$
  - $10n^2+4n+2=O(n^2)$
    $10n^2+4n+2 \leq 11n^2$ for all $n \geq 10$
  - $3n+2 = O(n^2)$
    $3n+2 \leq n^2$ for all $n \geq 4$ (Not tight enough!)
- g(n) should be a least upper bound

Asymptotic Notation – Big Ω

- f(n)=Ω(g(n)) iff

∃ a real constant c > 0 and an integer constant $n_0 \geqq 1$, ∋ f(n) ≥ cg(n) ∀n, n ≥ n_0



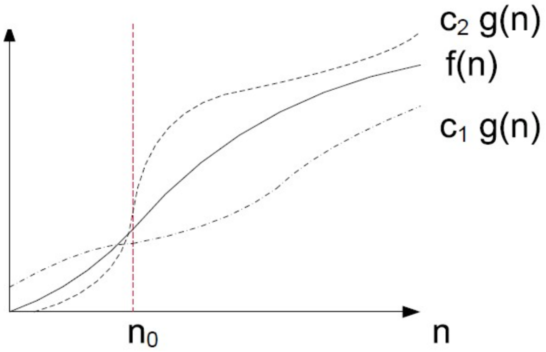
Big Ω Examples

- Examples
  - 3n+3=Ω(n)
    3n+3≥3n for all n≥1
  - $6*2^n+n^2=\Omega(2^n)$
    $6*2^n+n^2 \geq 2^n$ for all n≥1
  - 3n+3=Ω(1)
    3n+3≥3 for all n≥1 (Not tight enough!)
- g(n) should be a greatest lower bound

Asymptotic Notation – Big Θ

- f(n)=Θ(g(n)) iff

∃ two positive real constants c_1,c_2 > 0, and an integer constant $n_0 \geqq 1$, ∋ $c_1 g(n) \le f(n) \le c_2 g(n)$, ∀ n, $n \ge n_0$



Big Θ Examples

- Examples
  - 3n+2= Θ(n)

    $3n \le 3n+2 \le 4n$, for all $n \ge 2$
  - $10n^2+4n+2= \Theta(n^2)$

    $10n^2 \le 10n^2+4n+2 \le 11n^2$, for all $n \ge 5$
- g(n) should be both lower bound & upper bound

Some Rules

- **Rule 1**:
  If $T_1(N)=O(f(N))$ and $T_2(N)=O(g(N))$ Then
  (a) $T_1(N)+T_2(N) = \max (O(f(N)), O(g(N)))$
  (b) $T_1(N)\times T_2(N) = O(f(N)\times g(N))$
- **Rule 2**:
  If T(N) is a polynomial of degree k, then
  $T(N)= \Theta(N^k)$

Running Time Calculation

- For loop

```
for (i=0; i<n; i++)
{
    x++;
    y++;
    z++;
}
```

- $n\times 3=O(n)$

Running Time Calculation

- Nested for loops

```
for (i=0; i <n; i++)
    for (j=0; j<n; j++)
        k++;
```

- $n \times n = O(n^2)$

Running Time Calculation

- Consecutive statements

```
for (i=0; i<n; i++)
    A[i]=0;
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
     A[i]+=A[j]+i+j
```

- $\max(1 \times n,\ 1 \times n \times n) = 1 \times n \times n = O(n^2)$

Running Time Calculation

- If/Else

```
if (i>0) {
    i++;
    j++;
}
else {
    for (j=0; j<n; j++)
     k++;
}
```

- max(2, 1×n)=n

CSIEB0100 Data Structures Basic Concepts 69

Running Time Calculation – Recursion

```
long int F(int N)
{
    if (N<=1)
        return 1;
    else
        return N*F(N-1);
}
```

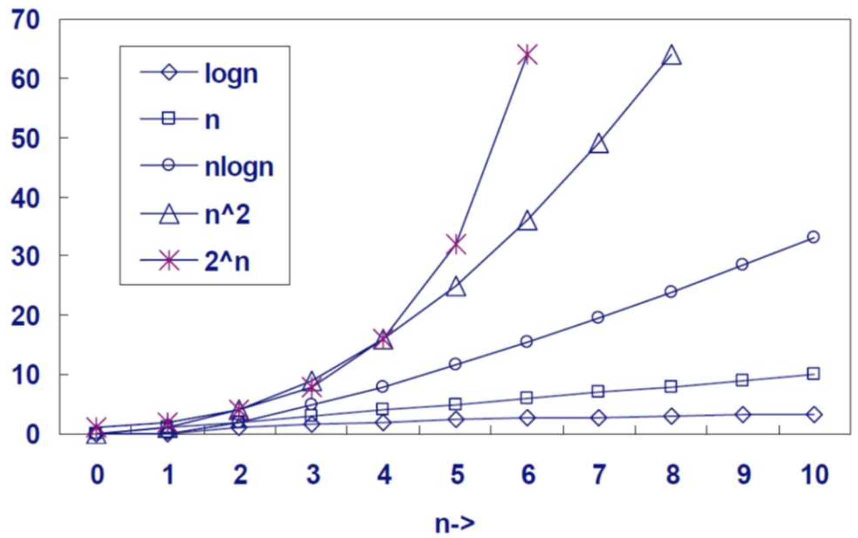
T(N)
=T(N-1)+c
=T(N-2)+2c
…
=T(1)+(N-1)c
=cN-c+1
=O(N)

CSIEB0100 Data Structures Basic Concepts 70

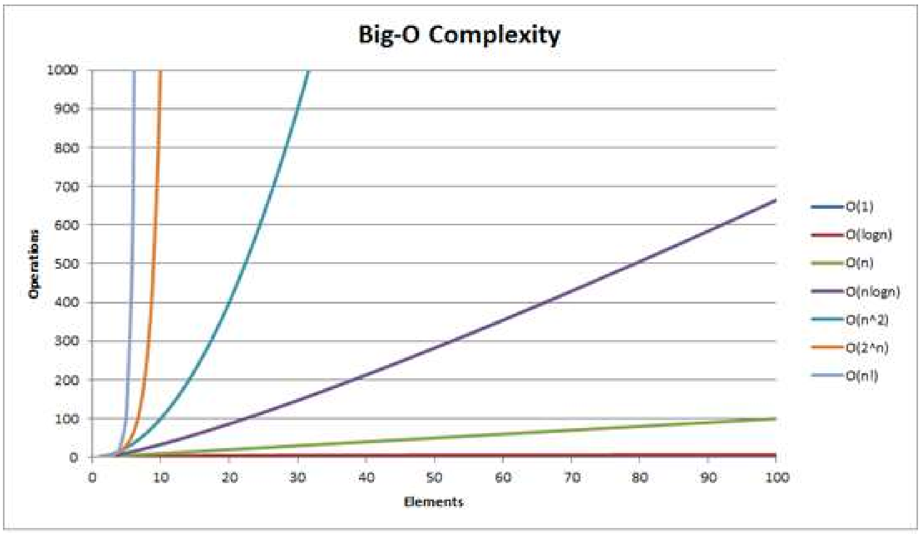
Typical Growth Rate

- c: constant
- log N: logarithmic
- $\log^2 N$: Log-squared
- N: Linear
- NlogN:
- N^2: Quadratic
- N^3: Cubic
- 2^N, c^N : Exponential

Comparison of Growth Rate



Colorful Growth Rate Comparison



Practical Complexities

10^9 instructions/second

n	*n*	*nlogn*	n^2	n^3
1000	1mic	10mic	1milli	1sec
10000	10mic	130mic	100milli	17min
10^6	1milli	20milli	17min	32years

Impractical Complexities

10^9 instructions/second

n	n^4	n^{10}	2^n
1000	17min	3.2×10^{13} years	3.2×10^{283} years
10000	116 days	???	???
10^6	3×10^7 years	??????	??????

Faster Computer vs Better Algorithm

Algorithmic improvement more useful than hardware improvement.

E.g. 2^n to n^3