Data Structure

2024 Spring

Ri Yu
Ajou University

What is Data Structure?

What is Data Structure?

- Storage used to store and organize data to access and update the data efficiently

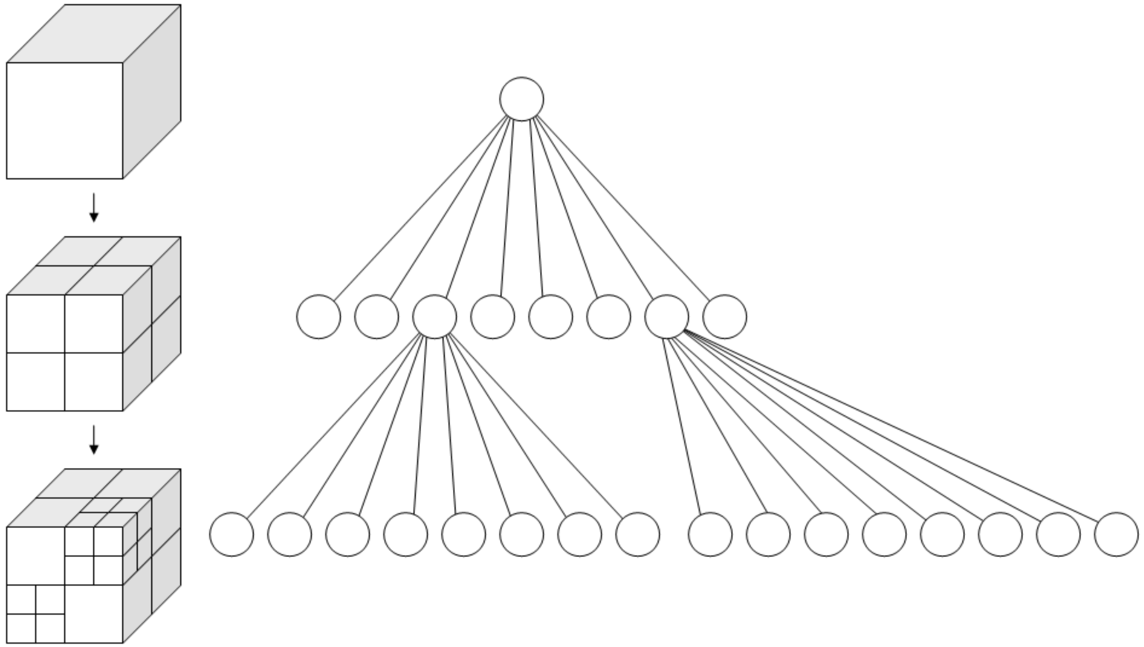
- Array, stack, queue, list, tree, graph, hash table and so on..

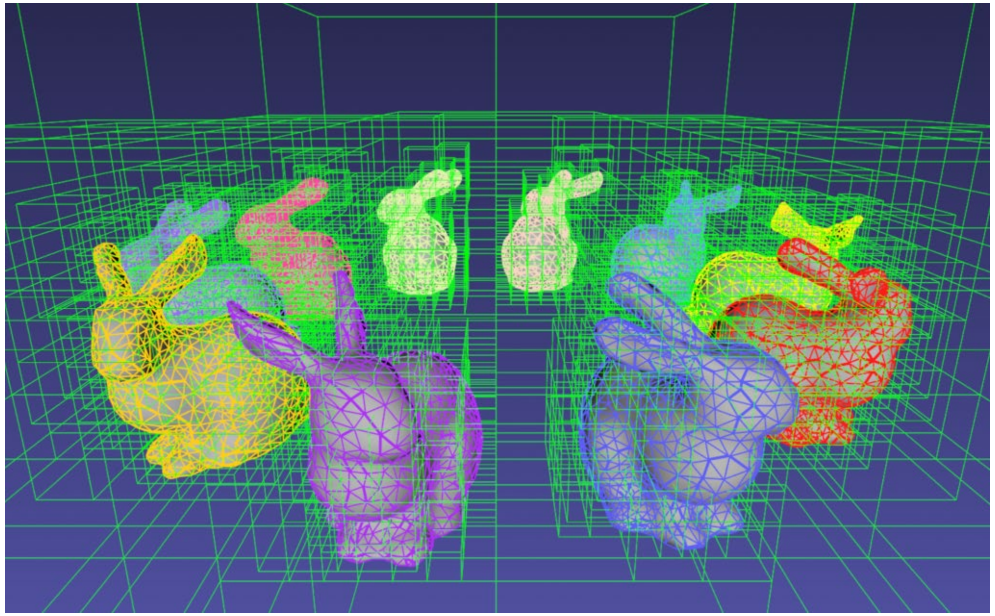
Examples

1) Octree for collision detection
2) Motion graph for character animation

Examples

1) **Octree** for collision detection

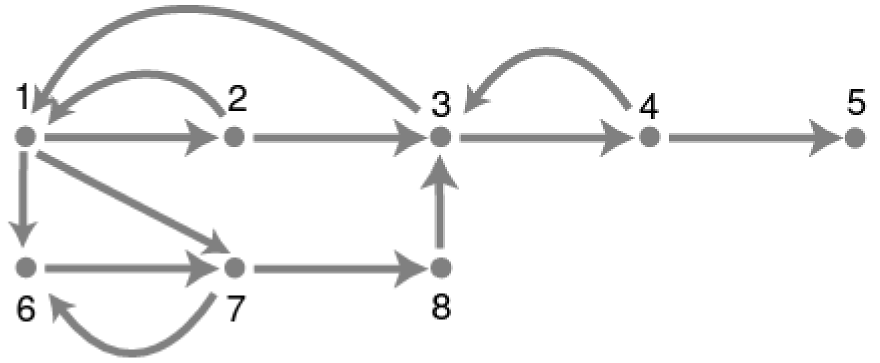


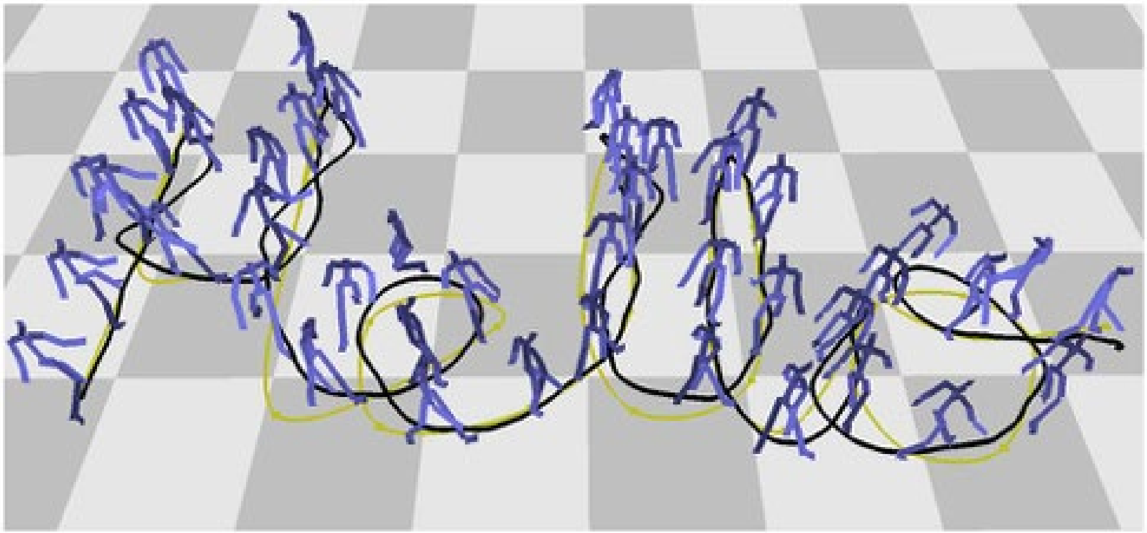


https://www.kitware.com/octree-collision-imstk/

Examples

2) **Motion graph** for character animation





Lucas Kovar, Michael Gleicher, and Frédéric Pighin. 2002. Motion graphs. ACM Trans. Graph. 21, 3 (July 2002), 473–482.

Chapter 1
Basic Concepts

Contents

Overview: System Life Cycle
Pointers and Dynamic Memory Allocation
Algorithm Specification
Data Abstraction
Performance Analysis
Performance Measurement

Overview: System Life Cycle

- View a large-scale computer program as a system with many complex interacting parts
- Systems undergo development process called system life cycle

Overview: System Life Cycle

1) Requirements (요구사항)
- ➢ set of specifications that define the purpose of the project
- ➢ describe input and output

2) Analysis (분석)
- ➢ break the problem down into manageable pieces; bottom-up, top-down
- ➢ bottom-up: old and naïve, early emphasis on the coding fine points
- ➢ top-down: begins with the purpose of the end-product, divide the problem into segments, generates diagrams used to design the system

Overview: System Life Cycle

3) Design (설계)
   - consider data objects and operations performed on them
   - create abstract data types, specification of algorithms
   - language independent; postpone implementation decisions

4) Refinement and Coding (정제 및 코딩)
   - choose representations for data objects
   - write algorithms for each operation on them

Overview: System Life Cycle

5) Verification (검증)

- Correctness proofs: mathematically prove the correctness of the program
  - difficult to develop for large projects
  - select algorithms proven correct to reduce errors
- Testing: requires working code and sets of test data
  - test data should include all possible cases
  - besides correctness, testing running time is also important
- Error removal: remove discovered errors
  - compared to programs written in "spaghetti" code, debugging well-documented program that is divided into autonomous units interacting through parameters is far easier
  - especially true if each unit is tested separately and integrated into system

Pointers and Dynamic Memory Allocation

- Pointer
  - Actual value of a pointer type is an address of memory
  - Two operators
    - &: address operator (주소 연산자)
    - *: dereferencing (or indirection) operator (역참조 연산자)
  - Special value
    - NULL(0): points to no object or function

- Dynamic memory allocation
  - malloc: allocate the amount of memory you may need
    - Ex) int *pi = (int*)malloc(sizeof(int));
  - free: return the area of memory to the system
    - ex) free(pi);

Algorithm

❖ Definition

- An algorithm is a finite set of instructions that, if followed, accomplishes a particular task

❖ Requirements

- Input (입력): zero or more quantities that are externally supplied
- Output (출력): at least one quantity is produced
- Definiteness (명확성): clear and unambiguous instructions
- Finiteness (유한성): terminated after finite steps
- Effectiveness (유효성): instruction is basic enough to be carried out

Example : Selection Sort

- ❖ Devise a program that sorts a set of n≥1 integers
  - Simple solution: from those integers that are currently unsorted, find the smallest and place it next in the sorted list

i	[0]	[1]	[2]	[3]	[4]
-	30	10	50	40	20
0	10	30	50	40	20
1	10	20	50	40	30
2	10	20	30	40	50
3	10	20	30	40	50

Example : Selection Sort

❖ Algorithm

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


  Interchange list[i] and list[min];
}
```

Two subtasks

- Finding the smallest integer
- Interchanging it with list[i]

Example : Selection Sort

```
void sort(int list[], int n)
{
        int i, j, min, temp;
        for (i = 0; i < n-1; i++) {
                min = i;
                for (j = i+1; j < n; j++)
                        if (list[j] < list[min])
                                min = j;
                SWAP(list[i], list[min], temp);
        }
}
```

Two subtasks
- Finding the smallest integer
- Interchanging it with list[i]

Recursive Algorithm 순환 알고리즘

❖ Recursion: A common method of simplification is to divide a problem into sub-problems of the same type

E.g.: $n! = n \times (n\text{-}1)!$, $\sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1} i$

❖ The function calls itself recursively on a smaller version of the input (n - 1) until reaching the base case

```
Example:  Power_of_2 (natural number k)

   Input: k, a natural number
   Output: k-th power of 2
   Algorithm (recursive):
     if k = 0, then return 1;
     else return 2*Power_of_2(k - 1)
```

```
int a = Power_of_2(4);   ←16
   2*Power_of_2(3)   ←2*8
     2*Power_of_2(2)   ←2*4
       2*Power_of_2(1)   ←2*2
         2*Power_of_2(0)   ←2*1
```

Recursive Algorithm

- Pros and Cons
  - Pros: More natural to express the functions
  - Cons: More memory is required than the iterative counterpart

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

```
int fibbonacci(int n) {
   if(n == 0){ return 0;}
   else if(n == 1) { return 1; }
   else {
    return (fibbonacci(n-1) + fibbonacci(n-2));
   }
}
```

Data Abstraction

- Data type
  - A collection of **objects** and a set of **operations** that act on those objects
  - For example, the data type **int** consists of the objects {0, +1, -1, +2, -2, …, INT_MAX, INT_MIN} and the operations +, -, *, /, and %
- The data types in C
  - The basic data types: char, int, float and double
  - The group data types:
    - Arrays: collections of elements of the same basic data type
    - Struct: collections of elements whose data types need not be the same
- Knowing the representation of the object of a data type
  - Can be useful and dangerous
  - Hiding the representation of objects of a data type from its users is a good design strategy: solely through the provided functions

Abstract Data Type (ADT) 추상 데이터 타입

❖ Definition

- A data type that is organized in such a way that the **specification** of the **objects** and the **operation**s on the objects is **separated** from the **representation** of the objects and the **implementation** of the operations
- ex) Ada: package, C++ and Java: class

❖ Specification vs implementation

- Operation specification
- Function name, arguments types, results type, what the function does
- not including internal representation or implementation details

Example : NaturalNumber ADT

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 $x, y \in$ *NaturalNumber*, *TRUE*, *FALSE* $\in$ *Boolean*
    and where +, -, <, and == are the usual integer operations.

Result type

NaturalNumber	Zero ( )	::= 0
Boolean	Is_Zero(x)	::= **if** (x) **return** *FALSE* **else return** *TRUE*
Boolean	Equal(x, y)	::= **if** (x== y) **return** *TRUE* **else return** *FALSE*
NaturalNumber	Add(x, y)	::= **if** ((x+y) <= *INT_MAX*) **return** $x+y$ **else return** *INT_MAX*
NaturalNumber	Successor(x)	::= **if** (x == *INT_MAX*) **return** x **else return** $x+1$
NaturalNumber	Subtract(x, y)	::= **if** (x<y) **return** 0 **else return** x-y

Transformers

end *NaturalNumber*

"is defined as"

Performance Analysis

- Criteria upon which we can judge a program
  - Does the program meet the original specifications of the task?
  - Does it work correctly?
  - Does the program contain documentation that shows how to use it and how it works?
  - Does the program effectively use functions to create logical units?
  - Is the program's code readable?
  - Does the program efficiently use primary and secondary storage?
  - Is the program's running time acceptable for the task?

Associated with the development of of a good programming style

Focus on performance evaluation

- Two distinct fields of performance evaluation
  - Performance analysis (machine independent)
    - **Space and time complexity**
  - Performance measurement (machine dependent)
    - Used to identify inefficient code segments

Space Complexity 공간 복잡도

❖ Definition

- The amount of memory that it needs to run to completion

$$S(P) = c + S_P(I)$$

- Fixed space requirements (c) 고정 공간 요구
  - Independent of the number and size of the inputs and outputs
    - Instruction space (to store the code)
    - Space for simple variables, fixed-size structured variable, constants
- Variable space requirements ($S_P(I)$) 가변 공간 요구
  - Depend on the particular instance I of the problem being solved
    - Usually given as a function of some characteristics of the instance I
      - number, size, and values of the inputs and outputs associated with I
      - ex) array containing n numbers -> n is an instance characteristic
    - Recursion function
      - Recursive stack space, parameters, local variables, return address

Examples

❖ Simple arithmetic function

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

$S_{abc}(I) = 0$

❖ Iterative function

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

$S_{sum}(I) = 0$

Examples

Required space
1. parameters
2. local variables
3. return address

❖ Recursive function

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

$$S_{rsum}(n) = 12n$$

Type	Name	Number of bytes
parameter : array pointer parameter : integer return address : (used internally)	list [ ] n	4 4 4
TOTAL per recursive call		12

Time Complexity

❖ Definition

- The amount of computer time that it needs to run to completion

$$T(P) = c + T_P(I)$$

$$= \text{compile time} + \text{run time}$$

$$\approx \text{run time}$$

$$\approx \text{number of the operations}$$

- The time, $T(P)$, taken by a program, P, is the sum of its *compile time* c and its *run* (or execution) time, $T_P(I)$
- Fixed time requirements
  - Compile time (c), independent of instance characteristics
- Variable time requirements
  - Run (execution) time T_P

Program Step

- Definition
  - A syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics
- Iterative summing of a list of numbers

```
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 */
    count++ ;                        /* for return */
    return tempsum;

}
```

Explicitly count
the number of operations

$2n + 3$ steps

Program Step

❖ Tabular Method : another way to obtain step counts

1. Determine the total number of steps contributed by each statement
    - Steps per execution (s/e) x Frequency
2. Add up the contribution of all statements

the number of times that each statement is executed

Step count table for Program sum

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

Asymptotic Notation 점근 표기법

❖ Motivation

- To determine practicality of algorithm
- Ex) Comparing the time complexity of $c_1n^2+c_2n$ and c_3n
  - For sufficiently large values of n, c_3n is faster than $c_1n^2+c_2n$
  - For small values of n, either could be faster
    - $c_1=1,\ c_2=2,\ c_3=100 \rightarrow c_1n^2+c_2n \leq c_3n$ *for* $n \leq 98$
    - $c_1=1,\ c_2=2,\ c_3=1000 \rightarrow c_1n^2+c_2n \leq c_3n$ *for* $n \leq 998$
  - Break even point
    - No matter what the values of c_1, c_2, and c_3, there will be an n beyond which c_3n is always faster than $c_1n^2+c_2n$
- Introducing three kinds of step counts
  - Best, worst and average cases

Asymptotic Notation

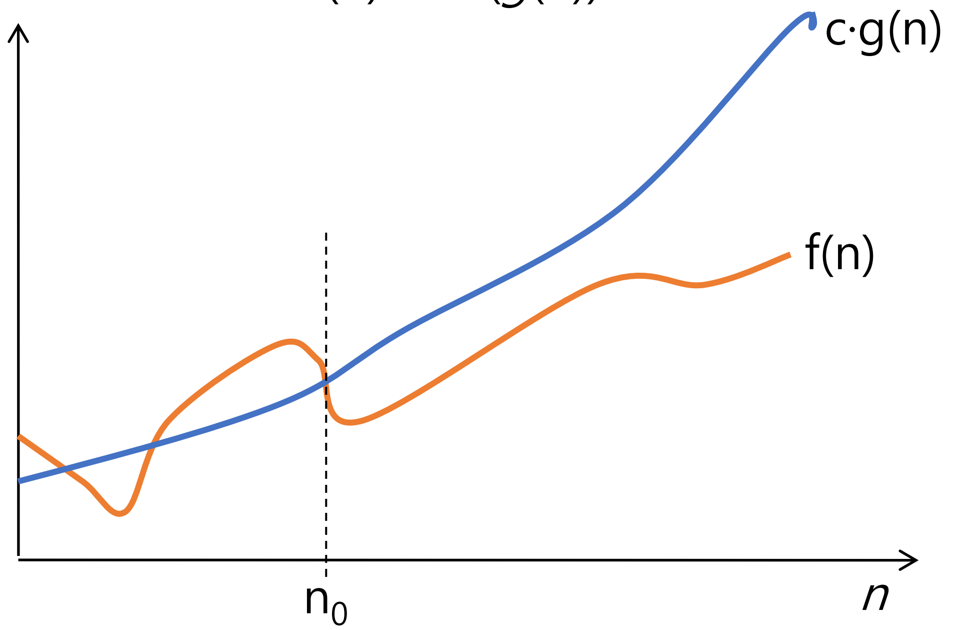
- O (Big-Oh) $f(n) = O(g(n))$
  “less than or equal to" relation
- Ω (Big-Omega) $f(n) = \Omega\ (g(n))$
  "greater than or equal to" relation
- Θ (Big-Theta) $f(n) = \Theta\ (g(n))$
  "equal to" relation

Big-Oh

❖ Definition: $f(n) = O(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all n, $n \geq n_0$

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

$f(n) = O(g(n))$



Examples

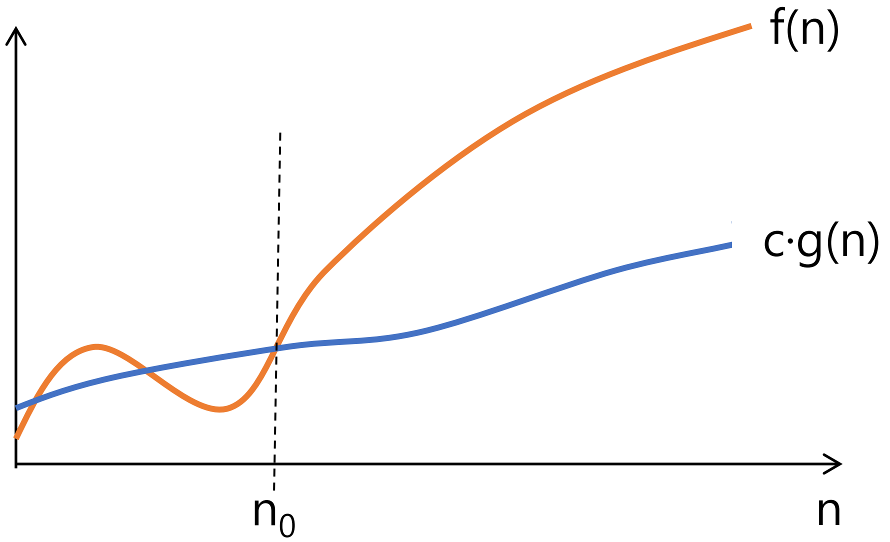
$3n+2=O(n)$	: $3n+2<=4n$ for $n>=2$
$3n+3=O(n)$	: $3n+3<=4n$ for $n>=3$
$100n+6=O(n)$	: $100n+6<=101n$ for $n>=10$
$10n^2+4n+2=O(n^2)$	: $10n^2+4n+2<=11n^2$ for $n>=5$
$6*2^n+n^2=O(2^n)$	: $6*2^n+n^2 <=7*2^n$ for $n>=4$

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, $n \geq n_0$

If $f(n) = a_m n^m + \ldots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$

$f(n) = \Omega(g(n))$



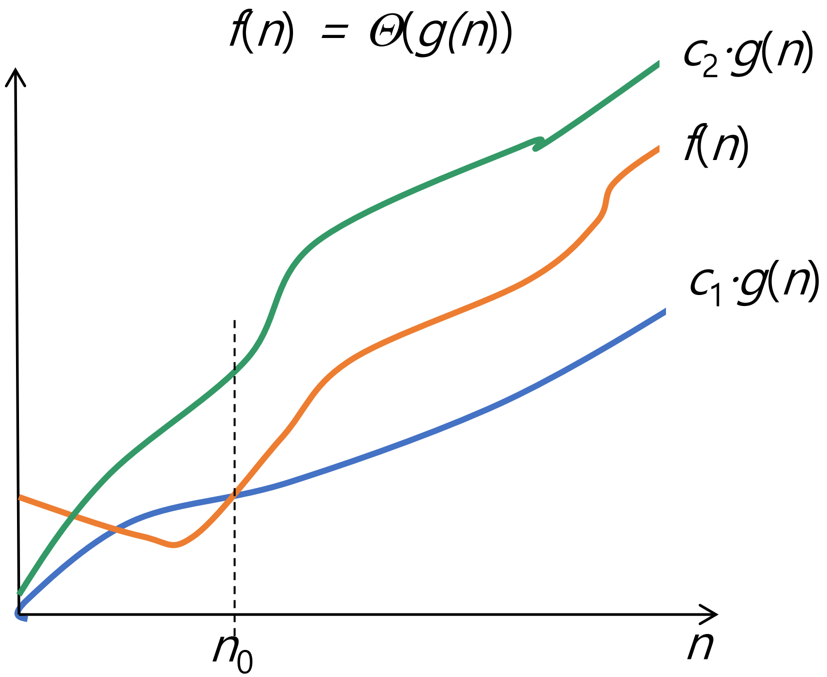
Examples

$3n+2 = \Omega(n)$	: $3n+2 >= 3n$ for $n>=1$
$3n+3 = \Omega(n)$	: $3n+3 >= 3n$ for $n>=1$
$100n+6 = \Omega(n)$	: $100n+6 >= 100n$ for $n>=1$
$10n^2+4n+2 = \Omega(n^2)$	: $10n^2+4n+2 >= n^2$ for $n>=1$
$6*2^n+n^2 = \Omega(2^n)$	: $6*2^n+n^2 >= 2^n$ for $n>=1$

Theta

❖ Definition: $f(n) = \Theta(g(n))$ iff there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$

If $f(n) = a_m n^m + \ldots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Theta(n^m)$



Examples

$3n+2 = \Theta(n)$

$10n^2+4n+2 = \Theta(n^2)$

$6 * 2^n+n^2 = \Theta(2^n)$

$3n+2 = O(n^2)$, but $3n+2 \neq \Theta(n^2)$

Practical Complexities

Instance characteristic n								
Time	Name	1	2	4	8	16	32
1	Constant	1	1	1	1	1	1
$\log n$	Logarithmic	0	1	2	3	4	5
n	Linear	1	2	4	8	16	32
$n \log n$	Log linear	0	2	8	24	64	160
n^2	Quadratic	1	4	16	64	256	1024
n^3	Cubic	1	8	64	512	4096	32768
2^n	Exponential	2	4	16	256	65536	4294967296
$n!$	Factorial	1	2	24	40326	20922789888000	26313×10^{33}

Figure 1.7 Function values

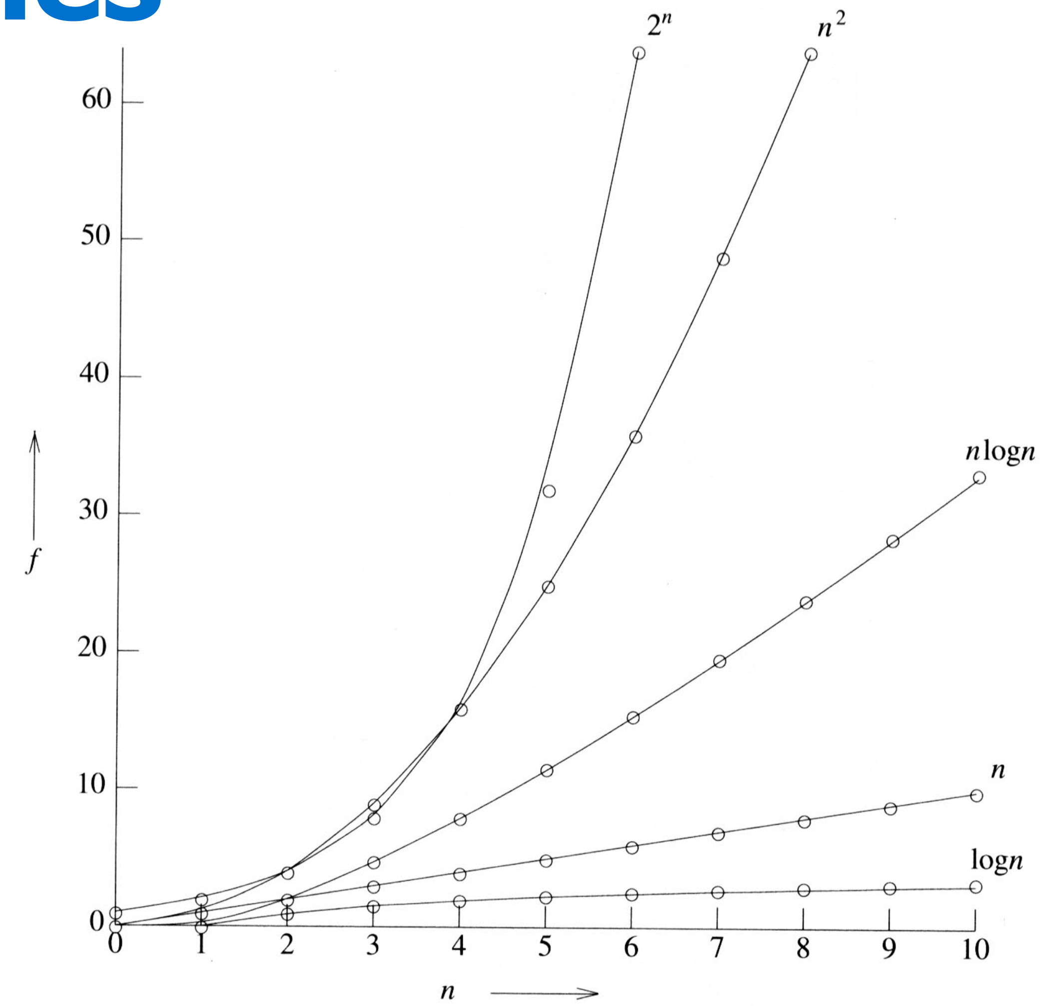


Figure 1.8 Plot of function values

Performance Measurement 성능 측정

❖ Clocking

- Although performance analysis gives us a powerful tool for assessing an algorithm's space and time complexity, at some point we also must consider how the algorithm executes on our machine
  - This consideration moves us from the realm of analysis to that of measurement

C언어 표준 라이브러리
#include <time.h>

Event timing in C

	Method 1	Method 2
Start timing	start = clock();	start = time(NULL);
Stop timing	stop = clock();	stop = time(NULL);
Type returned	clock_t	time_t
Result in seconds	duration = ((double) (stop−start)) / CLK_TCK;	duration = (double) difftime(stop,start);

Summary

- Algorithm Specification
  -Definition, 5 requirements, and recursion.

- Abstract data type (ADT) = manual of an object & its operation w/o implementation details

- Performance analysis (machine independent)
  - Space & time complexity
  - Asymptotic notation: Big-oh(upper bound), Omega(lower bound), Theta(both)

- Performance measurement (machine dependent)
  -C function :clock(), time(NULL)

Next Topic

- Arrays and structures