Welcome to Data Structures(ECE20010/ITP20001)

Youngsup Kim
idebtor@handong.edu
Handong Global University

Data Structures

Chapter 1

- *overview*
  - *pointers and dynamic memory allocation*
- ***algorithm specification***
  - ***recursive algorithm***
- *data abstraction*
- *performance analysis - time complexity*

1.3 Recursive algorithms

Example: Recursive binary search – revisited

Exercise:

-3	-2	0	0	1	5	5

compare with 0

compare with 5

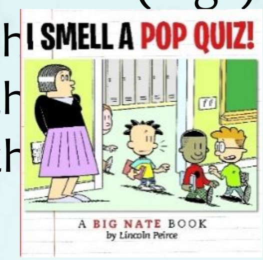
1	5	5

compare with 1

1

int binarySearch(int list[], int wally, int left, int right)

main(args)
binarySearch
binarySearch
binarySearch



int binarySearch(int list[], int wally, int left, int right)

```
if (left > right) return FAILURE;                          // base case

mid = (left + right)/2;
if (wally == list[mid]) return mid;                        // base case
if (wally < list[mid])
    return binarySearch(list, wally,             );        // recursive
else
    return binarySearch(list, wally,             );         // recursive
```

ECE 20010 Data Structures

Data Structures

Chapter 1

- *overview*
  - *pointers and dynamic memory allocation*
- *algorithm specification*
  - *recursive algorithm*
- ***data abstraction***
- *performance analysis - time complexity*

1.4 Data abstraction

A **data type** is a collection of **objects** and a set of **operations** that act on those objects.

Ex. **int** – integer numbers, the operations on integers are + - / * %

Three kind of data types in C;

(1) **primitives** - int, short, long, float, double, char
(2) **arrays** – collections of elements of the same primitive data type
(3) **struct** - collections of elements whose data types may be different (p.19, p59)

```
struct student {
    char lastName[10];
    int studentID;
    char grade;
}
```

```
typedef struct humanBeing {
    char name[10];
    int age;
    int sex;
}
```

1.4 Data abstraction

A **data type** is a collection of **objects** and a set of **operations** that act on those objects.

Why so many data types or many different programming languages?

- facing with large programs in the real world
- new requirements or constraints

Example:

- **stack, queue, search structure:**
  Operations are "insert/delete an item" at least;
  Implementation may be as array, linked list, tree, hash table, ...

1.4 Data abstraction

A **data type** is a collection of **objects** and a set of **operations** that act on those objects.

An **abstract data type (ADT)** is a data type that is organized in such a way that the object **specification** is **separated from** the object **implementation**. **Why?**

The object-oriented languages such as java may assist the programmer in implementing **abstract data types** through **class.** It is called the **encapsulation** which is one of the major characteristics built in Java. .

1.4 Data abstraction

What makes a program good?

(1) It works as specified.

(2) It is easy to understand and modify.

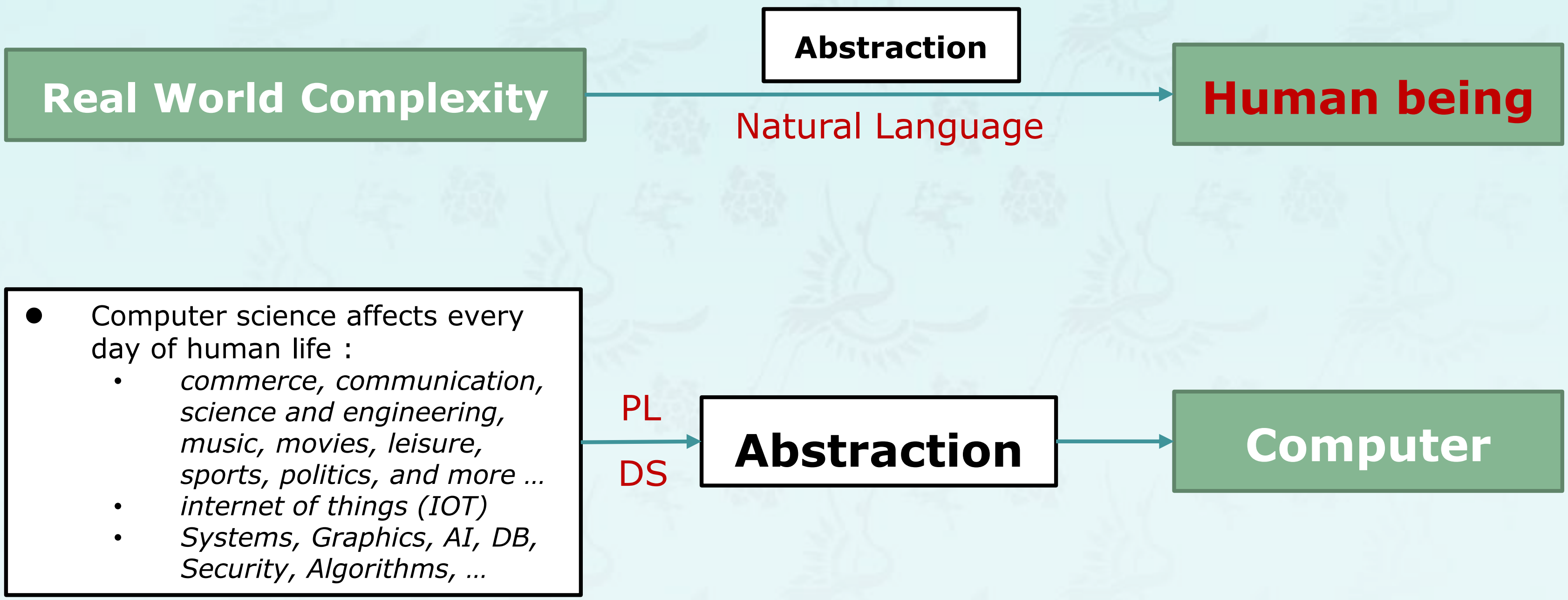
(3) It is reasonably efficient.

The benefits of using ADTs:

- The program is easier to understand since it is easier to see "high-level" steps being performed, not obscured by low-level code.
- Implementations of ADTs can be changed (e.g., for efficiency) without requiring changes to the program that uses the ADTs.
- ADTs can be reused in future programs.

- **Good programmers who make good programs use ADTs^^**

Course overview



1.4 Data abstraction

Example: ADT Natural Number

ADT Natural Number
Object: an ordered subrange of the integer from 0 to INT_MAX
Functions: for all x, y ∈ Natural number, TRUE, FALSE ∈ Boolean and where +, -, <, and == are the usual integer operations

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

1.4 Data abstraction

Example: ADT in C

1. **.c -** the implementation view
   - declaration of data types, code that implements its operations
2. **.h -** the abstract view
   - declaration for functions, pointer types, and globally accessible data

stack.h

```
#ifnef STACK
#define STACK
// Return a pointer to an empty stack.
extern StackType InitStack ( );

// Push value onto the stack,
// returning success flag.
extern boolean Push (int k);

// Pop value from the stack,
// returning success flag.
extern boolean Pop ( );

// Print the elements of the stack.
extern PrintStack (StackType stack);
#endif
```

stack.c

```
#include "stack.h"
#define STACKSIZE 5
struct StackStructType {
   int stackItems [STACKSIZE];
   int nItems;
};
typedef struct StackStructType *StackType;

// Return a pointer to an empty stack.
StackType InitStack ( ) {
   char *calloc( );
   StackType stack;
   stack = (StackType)
        calloc (1, sizeof (struct StackStructType));
   stack->nItems = 0;
   return (stack);
}
.........
```

ECE 20010 Data Structures

Data Structures

Chapter 1

- *overview*
  pointers and dynamic memory allocation
- *algorithm specification*
  recursive algorithm
- *data abstraction*
- ***performance analysis - time complexity***

1.5 Performance analysis

The program we write should
1. meet the specification.
2. work correctly.
3. be documented properly.
4. run effectively
5. be readable.

6. **use the storage effectively – space**
7. **run timely – time**

space & time complexity

The **space complexity** of a program is the amount of **memory** that it needs to run to completion.

The **time complexity** of a program is the amount of computer **time** that it needs to run to completion.

1.5 Performance analysis

Space complexity:

1. Fixed space requirements : c
   - that do not depend on input size, simple or fixed-size variables
2. Variable space requirements : $S_p(I)$
   - that depend on the instance I, stack, variable

The total space requirement for the program P:

$$\boldsymbol{S(P) = c + S_p(I)}$$

where **c** is a constant for fixed space and variable space for the instance I.

We are concerned about only $\boldsymbol{S_p(I)}$, but not c. **Why?**

Because we usually **compare** the algorithms of the programs.

1.5 Performance analysis

Space complexity: $S(P) = c + S_p(I)$

Example: $S_{sum}(n)$ = ?

Program1.11

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

$\boldsymbol{S_{sum}(n)} = 0$ since the C passes list[] by its address.

1.5 Performance analysis

Space complexity: $S(P) = c + S_p(I)$

Example: S_{sum}(MAX_SIZE) = ?

Program1.12

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

The variable space requirement are for **two** parameters and **one** return address are saved in the system stack **per recursive call**:

$$sizeof(n) + list[]\ address + return\ address = 12$$

$$\boldsymbol{S_{sum}(MAX\_SIZE) = 12 * MAX\_SIZE}$$

1.5 Performance analysis

Time complexity: The time taken by the program P:

$$\boldsymbol{T(P) = compile\ time\ c + execution\ time\ T_p(n)}$$

Similarly, we are concerned about only $\boldsymbol{T_p(n)}$**,** but not c.

Example: $Tp(n) = c_a ADD(n) + c_s SUB(n) + c_l LDA(n) + c_{st} STA(n)$

where n – number of execution, c for constant time for operation

We are not concerned about this, but …

Program step: a meaningful program segment whose execution time is independent of the instance characteristics.

Example:

$$a = 2;$$
$$a = 2 * b + 3 * c/d - e + f/g/a/b/c\ ;$$

⇨ *1 step!!*
⇨ *1 step!!*

1.5 Performance analysis

Example: How many **program steps** required?

Program 1.11	2n+3
`float sum(float list[], int n) {`	
`float tempsum = 0;`	1
`int i;`	
`for (i=0; i<n; i++)`	n+1
`tempsum += list[i];`	n
`return tempsum;`	1
`}`	

1.5 Performance analysis

Example: How many **program steps** required?

Program 1.11	2n+3
`float sum(float list[], int n) {`	
`    float tempsum = 0;`	1
`    int i;`	
`    for (i=0; i<n; i++)`	n+1
`        tempsum += list[i];`	n
`    return tempsum;`	1
`}`	

Program 1.13	**2n + 3**
`float sum(float list[], int n) {`	
`    float tempsum = 0; count++;    // for assignment`	1
`    int i;`	
`    for(i=0; i<n; i++) {`	
`        count++;                   // for the for loop`	n
`        tempsum += list[i]; count++; // for assignment`	n
`    }`	
`    count++;                       // last execution of for`	1
`    count++;                       // for return`	1
`    return tempsum;`	
`}`	

1.5 Performance analysis

Exercise: How many **program steps** required?

Program 1.12	2n + 2
`float rsum(float list[], int n) {`	
`  if (n)`	n + 1
`    return rsum(list, n-1) + list[n-1];`	n
`  return 0;`	1
`}`	

Program 1.15 **2n + 2**

```
float rsum(float list[], int n) {
  count++;             // for if conditional
  if (n){              // if n == 0, called once, if n>0, n times
    count++;           // for return & rsum invocation n times
    return rsum(list, n-1) + list[n-1];
  }
  count++;
  return list[0];     // if  n == 0, called once
}
```

1.5 Performance analysis

Comparison:

Program 1.11

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

Program 1.12

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

※ *2n + 3 (iterative) > 2n + 2 (recursive)*

⇨ $T_{iterative} > T_{recursive}$

1.5 Performance analysis

Example: How many **program steps** required?

Program 1.16 – sum of matrix	
`void add(int a[][MAX_SIZE], int b[][MAX_SIZE],`	
`         int c[][MAX_SIZE}, int rows, int cols) {`	
`   int i, j;`	
`   for(i=0; i<rows; i++)`	`rows + 1`
`     for(j=0; j<cols; j++)`	`rows * (cols+1)`
`       c[i][j] = a[i][j] + b[i][j];`	`rows * cols`
`}`	

❖ Can we try it without adding "count"?

1.5 Performance analysis

Example: How many **program steps** required?

Program 1.16 – sum of matrix	

`void add(int a[][MAX_SIZE], int b[][MAX_SIZE],`
`int c[][MAX_SIZE}, int rows, int cols) {`
`int i, j;`
`for(i=0; i<rows; i++)`
`for(j=0; j<cols; j++)`
`c[i][j] = a[i][j] + b[i][j];`
`}`
	
	`rows + 1`
	`rows * (cols+1)`
	`rows * cols`

❖ Can we try it without adding "count"?

Program 1.16	

`void add(int a[][MAX_SIZE], int b[][MAX_SIZE],`
`int c[][MAX_SIZE}, int rows, int cols) {`
`int i, j;`
`for(i=0; i<rows; i++)        // including the last loop`
`for(j=0; j<cols; j++)       // including the last loop`
`c[i][j] = a[i][j] + b[i][j];     // rows * cols`
`}`
	
	`rows + 1`
	`rows * (cols + 1)`
	`rows * cols`

// **step count = 2 rows*cols + 2 rows + 1**

1.5 Performance Analysis - Asymptotic notation (O,Ω,Θ) - 점금표기법

Why step count?

It is to compare the **time complexities** of two programs that compute the same function and also to predict the **growth rate** in run time.

Example: Let's compute the step count for three programs and compare their time complexities.

1. $T_{add}(n)$ – adding two numbers
2. $T_{sum}(n)$ – adding list of numbers
3. $T_{mtx}(n)$ – adding two matrix

Chapter 1 – Basic concepts

1.5 Performance Analysis - Asymptotic notation (O,Ω,Θ) - 점금표기법

Program **add**	step count	cost

`float add(int a, int b) {`
`  return     a + b;`
`}`
	1	2

Program 1.11 **sum of list**	step count	cost
`float sum(float list[], int n) {`		
`  float total = 0;`	1	1, c1
`  int i;`		
`  for (i=0; i<n; i++)`	n + 1	2, c2
`    total += list[i];`	n	2, c3
`  return total;`	1	1, c4
`}`		

Program 1.16 sum of matrix

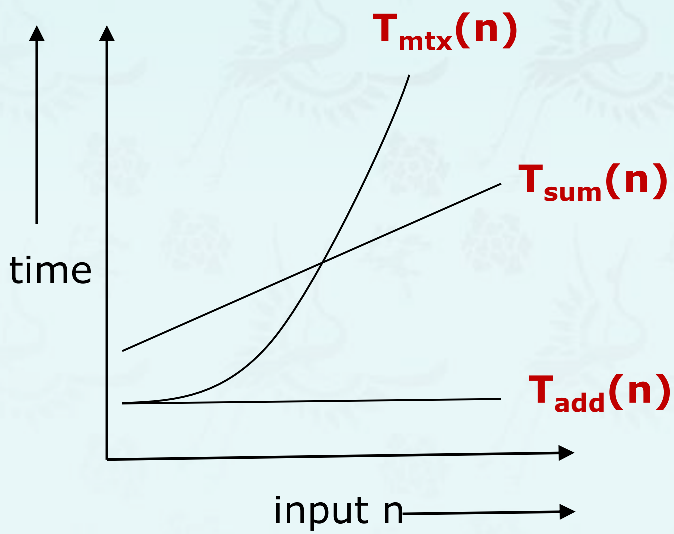
```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
         int c[][MAX_SIZE}, int rows, int cols) {
  for(int i=0; i<rows; i++)          // (rows + 1) include last loop
    for(int j=0; j<cols; j++)        // rows*(cols+1) include last loop
      c[i][j] = a[i][j] + b[i][j];   // rows * cols
}
// step count = 2 rows*cols + 2 rows + 1
```

1.5 Performance Analysis - Asymptotic notation (O,Ω,Θ) - 점금표기법

$$T_{add}(n) = 2 \quad \rightarrow O(1)$$

$$T_{sum}(n) = 1 + 2(n+1) + 2n + 1 = 4n + 4$$
$$= c * n + c' \quad \rightarrow O(n)$$

$$T_{mtx}(n) = 2\,rows * cols + 2\,rows + 1$$
$$= a * n^2 + b * n + c \quad \rightarrow O(n^2)$$

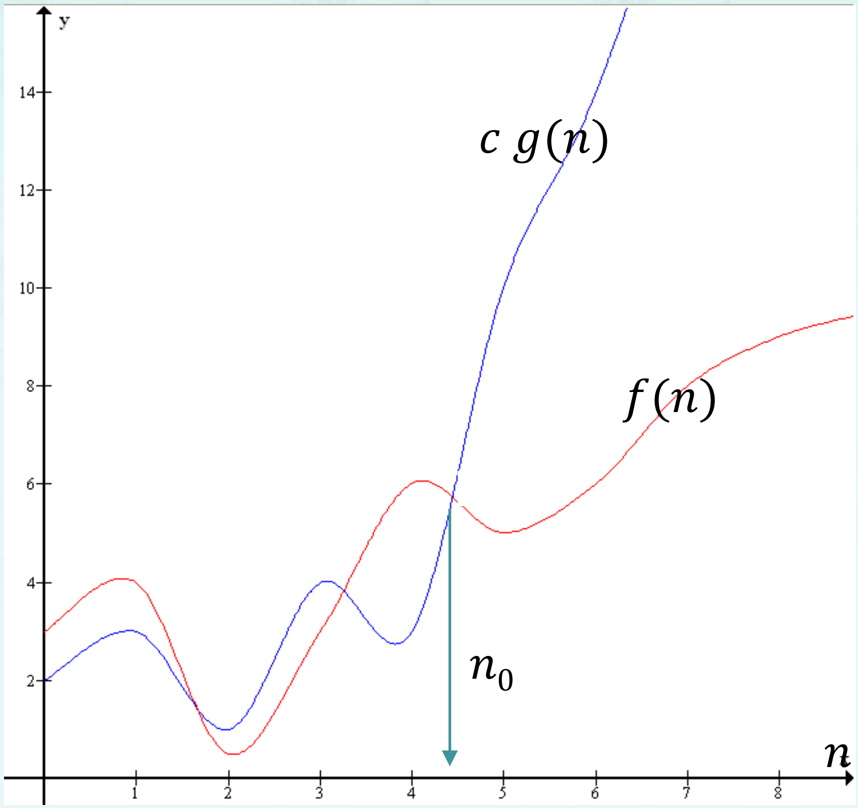


1.5 Performance Analysis - Asymptotic notation (O,Ω,Θ) - 점금표기법

The "Big-Oh" Notation:

Let f(n) and g(n) be functions mapping nonnegative integers to real numbers. We say that ***f(n) is O(g(n))*** iff there are positive constants c and n_o such that

$$\boldsymbol{f}(\boldsymbol{n}) \leq \boldsymbol{c}\ \boldsymbol{g}(\boldsymbol{n}), for\ n \geq n_0.$$



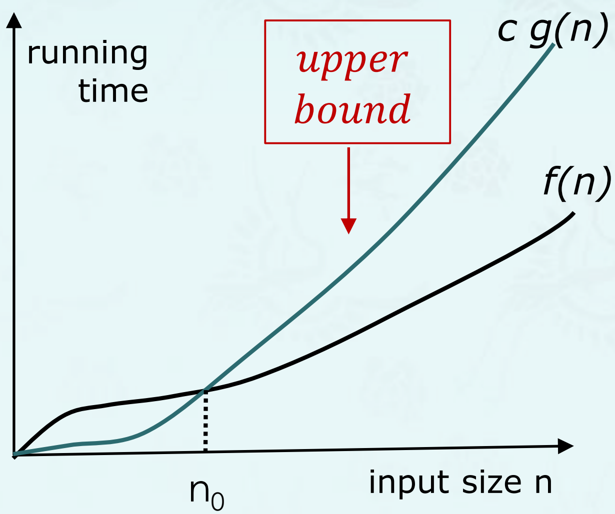
1.5 Performance Analysis - Asymptotic notation (O,Ω,Θ) - 점금표기법

The "Big-Oh" Notation:

Let f(n) and g(n) be functions mapping nonnegative integers to real numbers.
We say that ***f(n) is O(g(n))*** iff there are positive constants c and n_0 such that

$$\boldsymbol{f(n)} \le \boldsymbol{c\, g(n)}, for\ n \ge n_0.$$

Then it is pronounced as "$f(n)$ ***is*** *big Oh of* $g(n)$ *or* $f(n) = O(g(n))$".



Example: Justify that the function $8n - 2$ is $O(n)$.

Given $f(n) = 8n - 2, g(n) = n$,
we need to find c and n_0 such that
$8n - 2 \le c\,n$ for every integer $n \ge n_0$.

An easy choice among many is $c = 8$ and $n_0 = 1$.
Therefore, $f(n) = 8n - 2\ is\ O(n)$.

$g(n) = n$

1.5 Performance Analysis - Asymptotic notation (O,Ω,Θ) - 점금표기법

[Big 'Oh'] ***f(n) = O(g(n))*** iff there are positive constants c and n_o such that

$$\boldsymbol{f}(\boldsymbol{n}) \;\leq\; \boldsymbol{c}\,\boldsymbol{g}(\boldsymbol{n}), for\ n \;\geq\; n_0.$$

Examples:

1) $3n + 2\ =$
2) $3n + 3\ =$
3) $100n + 6 =$
4) $10n^2 + 4n + 2\ =$
5) $6 * 2^n + n^2\ =$
6) $3n + 3\ =$
7) $10n^2 + 4n + 2\ =$
8) $3n + 2 \neq O(1)$
9) $10n^2 + 4n + 2\ \neq O(n)$

1.5 Performance Analysis - Asymptotic notation (O,Ω,Θ) - 점금표기법

[Big 'Oh'] $f(n) = O(g(n))$ iff there are positive constants c and n_0 such that

$$f(n) \leq c\, g(n), for\ n \geq n_0.$$

Preferred Big-Oh usage:

- **Pick the tightest bound.** If $f(N) = 5N$, then:

  $f(N) = O(N^5)$
  $f(N) = O(N^3)$
  $f(N) = O(N \log N)$
  $f(N) = O(N)$ ← preferred or right!

- **Ignore constant factors and low order terms:**

  $f(N) = O(N)$, *not* $f(N) = O(5N)$
  $f(N) = O(N^3)$, *not* $f(N) = O(N^3 + N^2 + 15)$

  - Wrong: $f(N) \leq O(g(N))$
  - Wrong: $f(N) \geq O(g(N))$
  - Right: $f(N) = O(g(N))$

1.5 Performance Analysis - Asymptotic notation (O,Ω,Θ) - 점금표기법

[Big 'Oh'] *f(n)* = O(*g*(*n*)) iff there are positive constants *c* and n_0 such that

$$f(n) \le c\, g(n), for\; n \ge n_0.$$

Suppose two algorithms, A and B, solving the same problem have the running time of $O(n)$ and $O(n^2)$, respectively.
Then algorithm A is asymptotically better than algorithm B.
※ $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

constant — 정수함수
logarithmic — 대수
linear — 선형함수
linearithmic — 선형대수 loglinear
quadratic, cubic — 2/3승함수
exponential — 지수함수

1.5 Performance Analysis - Asymptotic notation (O,Ω,Θ) - 점금표기법

[Omega] ***f(n)* = Ω (*g(n)*)** iff there exist positive constants *c* and n_0 such that

$$f(n) \geq c\, g(n), for\ n \geq n_0.$$

Example: Let's suppose we have

$f(n) = 5n^2 + 2n + 1$

$g(n) = n^2$

For all $n \geq 0$, this $(2n + 1)$ will be ≥ to 1, **if** we have $c = 5$ and $n_0 = 0$.

Then, 5 $n^2 \leq f(n)$, for all $n \geq 0$

Therefore, we can say that the time complexity of $f(n)$ is Ω ($\boldsymbol{n^2}$)**;**

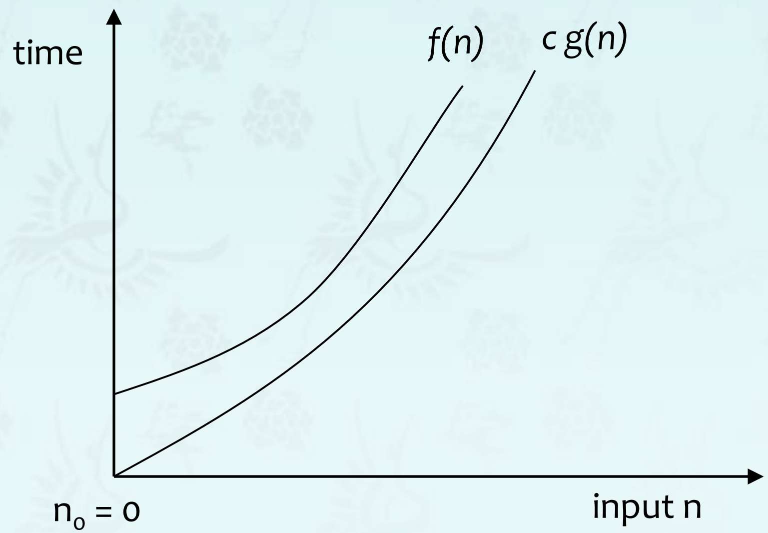
1.5 Performance Analysis - Asymptotic notation (O,Ω,Θ) - 점금표기법

[Omega] *f(n)* = Ω (*g(n)*) iff there exist positive constants c and n_0 such that

$$f(n) \geq c\, g(n), for\ n \geq n_0.$$

Example: Let's suppose we have

$f(n) = 5n^2 + 2n + 1$
$g(n) = n^2$



❖ **Omega** notation gives us the **lower bound** of the growth rate of a function.

1.5 Performance Analysis - Asymptotic notation (O,Ω,Θ) - 점금표기법

[Omega] *f(n)* = Ω (*g(n)*) iff there exist positive constants c and n_0 such that

$$f(n) \ge c\, g(n), for\ n \ge n_0.$$

Example:

1) $3n + 2 = \Omega(n)\ since\ \mathbf{3n + 2} \ge \mathbf{3n}\ for\ n \ge 1$

2) $3n + 3 = \Omega(n)\ since\ 3n + 3 \ge 3n\ for\ n \ge 1$

3) $100n + 6 = \Omega(n)\ since\ 100n + 6 \ge 100n\ for\ n \ge 1$

4) $100n^2 + 4n + 2 = \Omega(n^2)\ since\ 100n^2 + 4n + 2 \ge n^2\ for\ n \ge 1$

5) $6 * 2^n + n^2 = \Omega(2^n)\ since\ 6 * 2^n + n^2 \ge 2^n\ for\ n \ge 1$

1.5 Performance Analysis - Asymptotic notation (O,Ω,Θ) - 점금표기법

[Theta] ***f(n) = Θ (g(n))*** iff there exist positive constants *c* and n_0 such that

$$c_1 g(n) \le f(n) \le c_2 g(n), for\ n \ge n_0.$$

Example: Let's suppose we have

$f(n) = 5n^2 + 2n + 1$
$g(n) = n^2$

Then, we can choose $c_1 = 5, c_2 = 8$, and $n_0 = 1$; and our inequality will hold.
Therefore we can say that the time complexity of

$$f(n) = 5n^2 + 2n + 1 = \Theta\,(n^2)$$

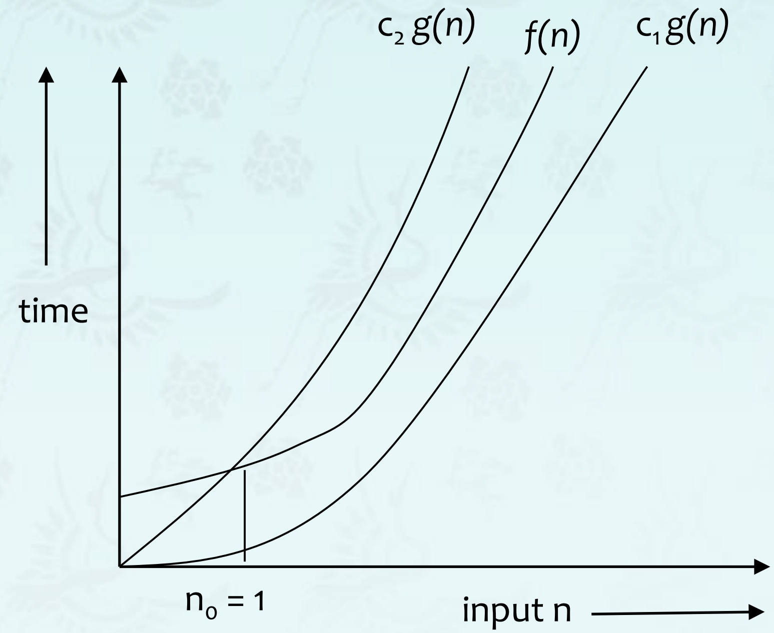
1.5 Performance Analysis - Asymptotic notation (O,Ω,Θ) - 점금표기법

[Theta] *f(n)* = **Θ** *(g(n))* iff there exist positive constants *c* and n_0 such that

$$c_1 g(n) \le f(n) \le c_2 g(n), for\ n \ge n_0.$$

Example: Let's suppose we have

$$f(n) = 5n^2 + 2n + 1$$
$$g(n) = n^2$$



❖ **Θ notation** best describes or give the best idea about the growth rate of the function because it gives us a **tight bound** unlike **O and Ω** which give us **upper bound** and **lower bound,** respectively.

1.5 Performance Analysis - Asymptotic notation (O,Ω,Θ) - 점금표기법

[Theta] *f(n)* = *Θ* (*g(n)*) iff there exist positive constants *c* and n_0 such that

$$c_1 g(n) \le f(n) \le c_2 g(n), for\ n \ge n_0.$$

Example:

1) $3n + 2 = \Theta(n)$

since $\mathbf{3n} \le \mathbf{3n+2} \le \mathbf{4n}\ for\ all\ n \ge 2, c_1 = 3, c_2 = 4, and\ n_0 = 2$

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

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

4) $6 * 2^n + n^2 = \Theta(2^n)$

5) $10 * \log n + 4 = \Theta(\log n)$

1.5 Performance Analysis

Recurrence Relations is an equation that recursively defines a sequence or multidimensional array of values, once one or more initial terms are given: each further term of the sequence or array is defined as a function of the preceding terms.

For example:

$$T(1) = c$$
$$T(n) = T(n-1) + c$$

Useful formulas:

$1 + 2 + 3 + \ldots + N = N(N+1)/2$

$1 + 2 + 4 + 8 + \ldots + 2^n = 2^{n+1} - 1$

Recurrence equation

1.5 Performance Analysis – Linear search

The time complexity of the linear search:

- **Best Case:** **Find at first place - one comparison**
- **Worst Case:** **Find at nth place or not at all - n comparisons**
- **Average Case:** **It is shown below that this case takes - (n+1)/2 comparisons**

- In considering the average case there are n cases that can occur, i.e. find at the first place, the second place, the third place and so on up to the *n*th place. If found at the *i*th place then *i* comparisons are required. Hence the average number of comparisons over these n cases is:

  average = (1 + 2 + 3 … + n) / n
  = (n + 1) / n,
  where (1 + 2 + 3 + … + n) is equal to (n + 1)n/2.

Hence linear search is $an\ order(n)\ process\ or\ T(n) = O(n).$

Recurrence equation

1.5 Performance Analysis – Linear search

We may describe that the time complexity of the linear search is

$$T(1) = c$$
$$T(n) = T(n-1) + c$$

- **The cost of searching n elements is the cost of looking at 1 element, plus the cost of searching n – 1 elements.**
- Let's "**telescoping**" a few of these…

$$T(n) = T(n-1) + c$$
$$T(n-1) = T(n-2) + c$$
$$T(n-2) = T(n-3) + c$$
$$\ldots$$
$$T(2) = T(1) + c$$

- Then add each side,

$$T(n) = T(1) + (n-1)c$$
$$T(n) = c + nc - c$$
$$T(n) = O(n)$$

Recurrence equation

1.5 Performance Analysis – Selection sort

$$T(1) = 1$$
$$T(n) = n + T(n-1)$$

- **Unfolding** makes repeated substitutions applying the recursive rule until the base case is reached.

Substitute n-1 everywhere we see an n in the recurrence relation:

$$T(n-1) = (n-1) + T(n-2)$$
$$T(n) = n + (n-1) + T(n-2)$$

Making this substitution one more time we get

$$T(n) = n + (n-1) + (n-2) + T(n-3)$$

We repeat this process until we reaches T(1), base case

$$= n + (n-1) + \ldots + (n-(n-1)) + T(n-(n-1))$$
$$= n + (n-1) + \ldots + 2 + T(1)$$
$$= \frac{n(n+1)}{2} - 1 + T(1)$$
$$= O(n^2)$$

Recurrence equation

1.5 Performance Analysis – Selection sort

$T(1) = 1$

$T(n) = n + T(n-1)$

- **Telescoping**

$$
\begin{aligned}
T(n) &= n + T(n-1) \\
T(n-1) &= n - 1 + T(n-2) \\
T(n-2) &= n - 2 + T(n-3) \\
&\ldots \\
T(2) &= 2 + T(1)
\end{aligned}
$$

- **Add all terms in each side and cancel the equal terms, then it becomes**

$$
\begin{aligned}
T(n) &= n + (n - 1) + \ldots + 2 + T(1) \\
&= \frac{n(n+1)}{2} - 1 + T(1) \\
&= O(n^2)
\end{aligned}
$$

1.5 Performance Analysis – Binary search

Recurrence equation

Best case: Let suppose that 3 steps $= T(1) = O(1) = 1$

Worst case: Let suppose that $T(n) \; = 1 + T$ where n is $hi - lo$

- $O(\log n)$ where n is $array.length$
- Solve *recurrence equation* to know that…

```
// requires array a is sorted
// returns whether k is in array
boolean find(int[]a, int k){
   return binarySearch(a,k,0,a.length-1);// Java syntax
}

boolean binarySearch(int[]a, int k, int lo, int hi) {
   int mid = (hi+lo)/2;
   if (lo==hi)    return false;
   if (a[mid]==k) return true;
   if (a[mid]< k) return binarySearch(a,k,mid+1,hi);
   else           return binarySearch(a,k,lo,mid-1);
}
```

1.5 Performance Analysis – Binary search

Best case: Let suppose that 3 steps, $\mathrm{T}(1) = O(1) = 1$

Worst case: Let suppose that $\boldsymbol{T(n) = 1 + T(n/2)}$ where n is $hi - lo$

- Show $O(\log n)$ where n is $array.length$

1. Determine the recurrence relation. What is the base case?

$$T(n) = 1 + T(n/2)$$
$$T(n/2) = 1 + T(n/4)$$
$$T(n/4) = 1 + T(n/8)$$
$$\ldots$$
$$T(4) = 1 + T(2)$$
$$T(2) = 1 + T(1)$$

telescoping →

2. Sum up the left and right sides of the equations above:

$$T(n) + T(n/2) + T(n/4) + \ldots + T(2) = (1 + 1 + \ldots + 1) + T(n/2) \ldots + T(2) + T(1)$$

3. Cross out the equal terms to simplify. How many 1's on the right side?

$$T(n) = \log_2 n + T(1)$$
$$= \log_2 n + 1$$

Therefore the time complexity of binary search is $T(n)$ is $O(\log n)$

1.5 Performance Analysis – Binary search

Best case: Let suppose that 3 steps, $\mathrm{T}(1) = O(1) = 1$

Worst case: Let suppose that $\boldsymbol{T(n) = 1 + T(n/2)}$ where n is $hi - lo$

- Show $O(\log n)$ where n is $array.length$

1. Determine the recurrence relation. What is the base case?

$$T(n) = 1 + T\left(\frac{n}{2}\right) \qquad\qquad T(1) = 1$$

2. "**Unfolding**" the original relation to find an equivalent general expression *in terms of the number of expansions.*

$$\begin{aligned} T(n) &= 1 + 1 + T(n/4) \\ &= 1 + 1 + 1 + T(n/8) \\ &= 1 + \ldots + 1 + T(n/n) \end{aligned}$$

How many 1's here?

1.5 Performance Analysis – Binary search

Best case: Let suppose that 3 steps, $\mathrm{T}(1) = O(1) = 1$

Worst case: Let suppose that $\boldsymbol{T(n) = 1 + T(n/2)}$ where n is $hi - lo$

- Show $O(\log n)$ where n is $array.length$

1. Determine the recurrence relation. What is the base case?

$$T(n) = 1 + T\left(\frac{n}{2}\right) \qquad\qquad T(1) = 1$$

2. "**Unfolding**" the original relation to find an equivalent general expression *in terms of the number of expansions.*

$$\begin{aligned} T(n) &= 1 + 1 + T(n/4) \\ &= 1 + 1 + 1 + T(n/8) \\ &= 1 + \ldots + 1 + T(n/n) \\ &= \mathbf{1}k + T\left(\frac{\boldsymbol{n}}{2^k}\right) \end{aligned}$$

1.5 Performance Analysis – Binary search

Best case: Let suppose that 3 steps, $\mathrm{T}(1) = O(1) = 1$

Worst case: Let suppose that $\boldsymbol{T(n) = 1 + T(n/2)}$ where n is $hi - lo$

- Show $O(\log n)$ where n is $array.length$

1. Determine the recurrence relation. What is the base case?

$$T(n) = 1 + T\left(\frac{n}{2}\right) \qquad T(1) = 1$$

2. "**Unfolding**" the original relation to find an equivalent general expression *in terms of the number of expansions.*

$$\begin{aligned} T(n) &= 1 + 1 + T(n/4) \\ &= 1 + 1 + 1 + T(n/8) \\ &= 1 + \ldots + 1 + T(n/n) \\ &= \mathbf{1}k + T\left(\frac{\boldsymbol{n}}{2^k}\right) \end{aligned}$$

Find a closed-form expression by setting the number of expansions to a value which reduces the problem to a base case

$n/(2^k) = 1$ means $n = 2^k \rightarrow k = \log_2 n$

So $T(n) = 1 \log_2 n + 1$ (get to base case and do it)

So $T(n)$ is $O(\log n)$

1.5 Performance Analysis - Asymptotic notation (O, Ω, Θ) - 점금표기법

Asymptotic Analysis:

Suppose that two algorithms, A and B, solving the same problem have the running time of $O(n)$ and $O(n^2)$, respectively. Then this implies that algorithm A is **asymptotically better** than algorithm B.

We can use the **big-Oh** notation to order classes of functions by **asymptotic growth rate**.

Seven functions below are often used and ordered by increasing growth rate.

※ $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

n	log n	n	n log n	n^2	n^3	2^n
1	0	1	0	1	1	2
2	1	2	2	4	8	4
4	2	4	8	16	64	16
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84 x 10^19
128	7	128	896	16,384	2,097,152	3.40 x 10^38
256	8	256	2,048	65,536	16,777,216	1.15 x 10^77

※ Even if we achieve a dramatic speed-up in hardware, we still cannot overcome the handicap of an asymptotically slow program.

1.5 Performance Analysis - Asymptotic notation (O,Ω,Θ) - 점금표기법

Example: Running time estimates - empirical analysis

- Laptop executes 10^8 compares/second
- Supercomputer executes 10^{12} compares/second

use a reasonable time unit

	Insertion sort (N^2)			Merge sort (N lg N)		
N	Thousand	Million	Billion	Thousand	Million	Billion
Laptop	Instant	2.8 hours		Instant	1 sec	
Super Com	Instant	1 sec		Instant	Instant	Instant

※ **Bottom line:** Good algorithms are better than supercomputers.

ECE 20010 Data Structures

Data Structures

Chapter 1

- *overview*
  - *pointers and dynamic memory allocation*
- *algorithm specification*
  - *homework02*
  - *recursive algorithm*
- *data abstraction*
- *performance analysis - time complexity*