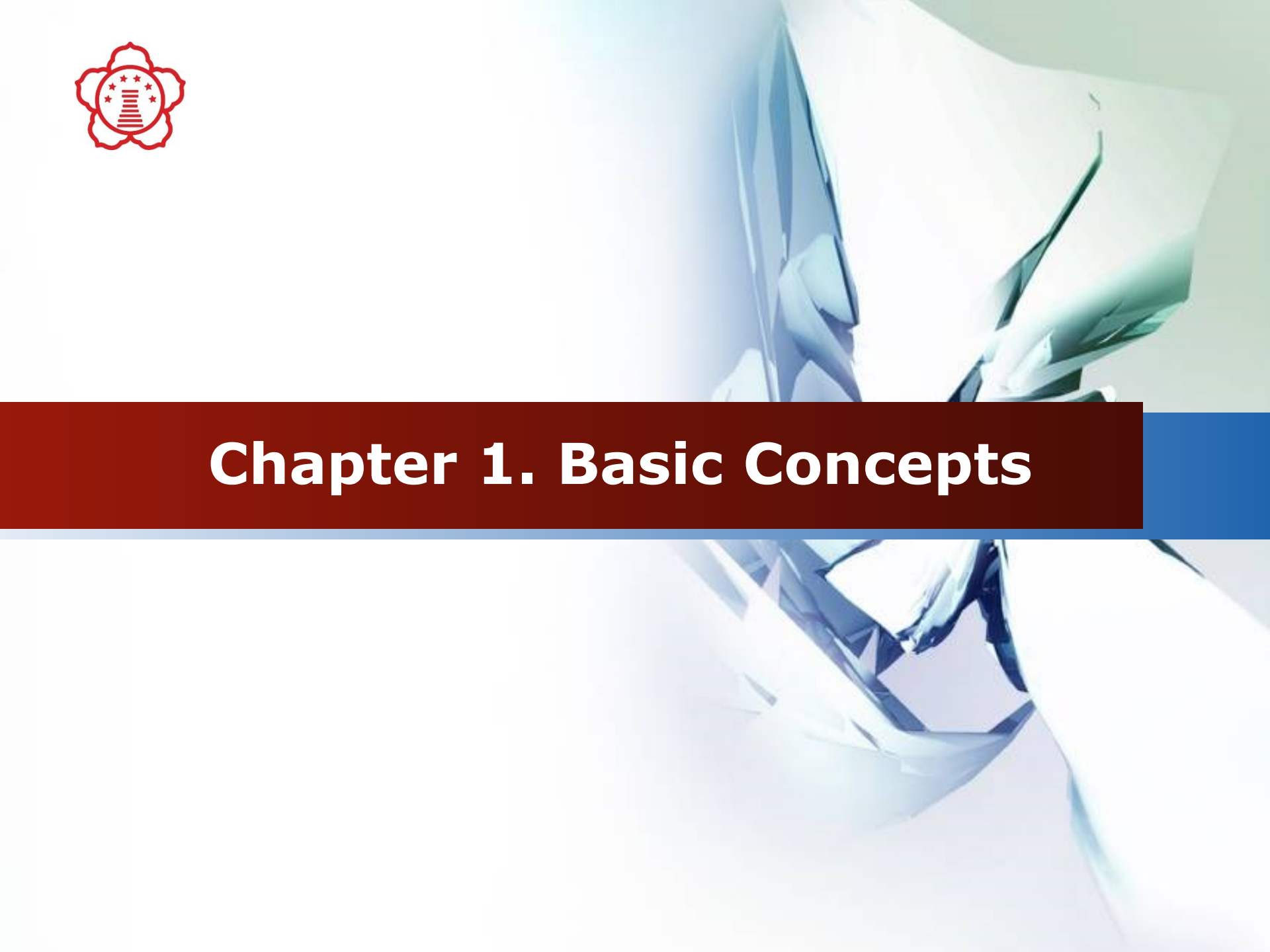


Chapter 1. Basic Concepts

Chapter 1. Basic Concepts

1.1 Overview: System Life Cycle

1.2 Pointers and Dynamic Memory Allocation

1.3 Algorithm Specification

1.4 Data Abstraction

1.5 Performance Analysis

1.6 Performance Measurement

1.5 Performance Analysis

Performance Evaluation

- Performance analysis
  - obtaining estimates of time and space that are *machine independent*

- Performance measurement
  - obtaining *machine-dependent* running time

1.5 Performance Analysis

❖ **Definition**

- ***Space complexity*** **:**

  the amount of memory that a program needs to run to completion

- ***Time complexity*** **:**

  the amount of computation time that a program needs to run to completion

1.5.1 Space Complexity

- **The space needed by a program**

1) Fixed space requirements

- independent on the number and size of the program's input and output
- space for instruction(code), simple variables, fixed-size structured variables, and constants

2) Variable space requirements

- space needed by structured variables whose size depends on the particular instance *I*, of the problem being solved
- space required when a function uses recursion

1.5.1 Space Complexity

- $S(P) = c + S_P(I)$
  - $S(P)$: total space requirement of a program P
  - c : constant for fixed space requirement
  - $S_P(I)$:
    - the variable space requirement of a program P working on an instance I
    - a function of some *characteristics* of the instance I, where the characteristics include the *number, size, and values of the I/O* associated with I

1.5.1 Space Complexity

- $S_P(I)$
  - Ex)
    - If the input is an array containing *n* numbers, then *n* is an instance characteristic.
    - If *n* is the only characteristic we wish to use, when computing $S_P(I)$, we will use $S_P(n)$ to represent $S_P(I)$.
  - When analyzing the space complexity of a program, we are usually concerned with $S_P(I)$.

1.5.1 Space Complexity

- **Example : abc**
  - has only fixed space requirements
  - $S_{abc}(I) = 0$

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

Program 1.10: Simple arithmetic function

1.5.1 Space Complexity

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

Program 1.11: Iterative function for summing a list of numbers

Example : sum

- Input includes an array with size *n*
- $S_{sum}(I)$
  - Depends on how the array is passed into the function
  - C passes the address of the first element of the array

$$S_{sum}(I) = S_{sum}(n) = 0$$

1.5.1 Space Complexity

- **Example : rsum**
  - for each recursive call
    - the parameters, local variables, the return address for each recursive call

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

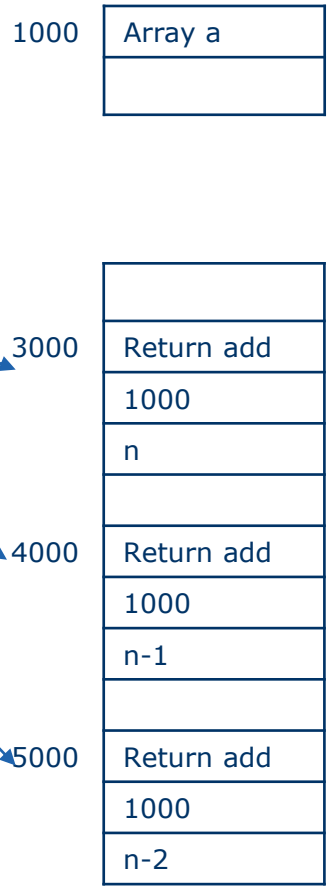
Program 1.12: Recursive function for summing a list of numbers

1.5.1 Space Complexity

resum(a, 10)

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

Program 1.12: Recursive function for summing a list of numbers



1.5.1 Space Complexity

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

Figure 1.1: Space needed for one recursive call of Program 1.12

- If the array has $n = MAX\_SIZE$ numbers, then $S_{rsum}(MAX\_SIZE) = 12 * MAX\_SIZE$
- The recursive version has a far greater overhead than its iterative counterpart!!!

1.5.2 Time Complexity

- **$T(P)$ = compile time + execution time(T_P)**
  - $T(P)$: time taken by a program P
  - compile time : just one

- **We are really concerned only with the T_P !!!**
  - But, determining T_P is not an easy task
  - Alternatively, we could count *program step*.
    - a machine independent estimate

1.5.2 Time Complexity

Definition

- ***program step*** :
  Syntactically or semantically meaningful program segment *whose execution time is independent of the instance characteristics.*

Ex)

a = 2;	*1 step*
*a = 2*b+3*c/d-e+f/g/a/b/c;*	*1 step*

1.5.2 Time Complexity

- **How to count program steps? 2 ways!**
  - using a global variable, *count* with initial value 0
  - using a tabular method
    - Only consider the program steps required by each *executable statement*

1.5.2 Time Complexity

- using a global variable

```
float sum(float list[], int n)  // count 는 전역변수
{
   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;
}
```

Program 1.13: Program 1.11 with count statements

1.5.2 Time Complexity

- The simplification makes easier to express the count arithmetically.
- The final *count* value will be **2*n*+3**

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

Program 1.14: Simplified version of Program 1.13

1.5.2 Time Complexity

- **Example [Recursive summing]**
  - for $n = 0$, *count* is 2.
  - for $n > 0$, n recursive calls, *count* = $2n$
  - The final *count* value will be **$2n+2$**

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

Program 1.15: Program 1.12 with count statements added

1.5.2 Time Complexity

- **Example [Matrix addition]**

```
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];
}
```

Program 1.16: Matrix addition

1.5.2 Time Complexity

❖ Example [Matrix addition]

```
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++) {
      count++;  /* for i for loop */
      for (j = 0; j < cols; j++) {
         count++; /* for j for loop */
         c[i][j] = a[i][j] + b[i][j];
         count++; /*  for assignment statement */
      }
      count++; /* last time of j for loop */
   }
   count++; /* last time of i for loop */
}
```

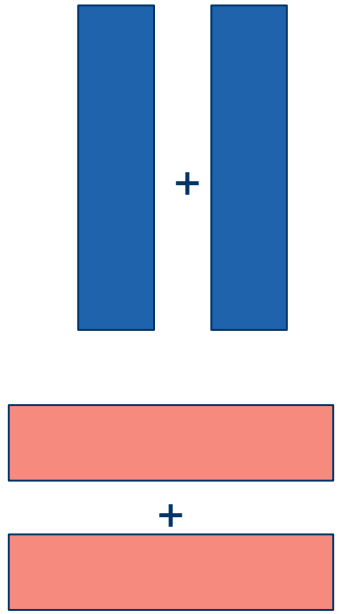
Program 1.17: Matrix addition with count statements

1.5.2 Time Complexity

- The final *count* value will be $2rows \cdot cols + 2rows + 1$
  - We should interchange the matrices if the number of rows is significantly larger than the number of columns.

```
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++)
         count += 2;
      count += 2;
   }
   count++;
}
```

Program 1.18: Simplification of Program 1.17



1.5.2 Time Complexity

❖ using a tabular method

※ s/e : step counts per each statement

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

1.5.2 Time Complexity

❖ using a tabular method

※ s/e : step counts per each statement

Statement	s/e	Frequency	Total steps
float rsum(float list[], int n)	0	0	0
{	0	0	0
if (n)	1	$n+1$	$n+1$
return rsum(list,n−1) + list[n−1];	1	n	n
return 0;	1	1	1
}	0	0	0
Total			$2n+2$

1.5.2 Time Complexity

❖ using a tabular method

※ s/e : step counts per each statement

Statement	s/e	Frequency	Total Steps
void add(int a[][MAX_SIZE] ···)	0	0	0
{	0	0	0
int i, j;	0	0	0
for (i=0; i<rows; i++)	1	*rows*+1	*rows*+1
for (j = 0; j < cols; j++)	1	*rows* · (*cols*+1)	*rows* · *cols* + *rows*
c[i][j] = a[i][j] + b[i][j];	1	*rows* · *cols*	*rows* · *cols*
}	0	0	0
Total			2*rows* · *cols* + 2*rows*+1

1.5.2 Time Complexity

```
for(i=0; i<n; i++)
        if(num == list[i]) {
                printf("find");
                break;
        }
```

- **Three kinds of step counts**
  - Best-case step count
    - The minimum number of steps that can be executed for given parameters
  - Worst-case step count
    - The maximum number of steps that can be executed for given parameters
  - Average step count
    - The average number of steps on instance with the given parameters

1.5.3 Asymptotic Notation (O, Ω, Θ)

- **Motivation to determine *step counts***
  - Compare the time complexities of two programs for the same function
  - Predict the growth in run time as the instance characteristics change

- **However,**
  - Determining the exact step count is very difficult.
    - *The exact step count is not very useful for comparative purposes*.
      - The notion of a step is itself inexact.

Asymptotic

- 사람의 수에 대한 예상(근사적 추정)
  - 최대 0000 명 정도 (식사준비 할 때)
  - 최소 0000 명 정도 (회비 걷을 때 예상)

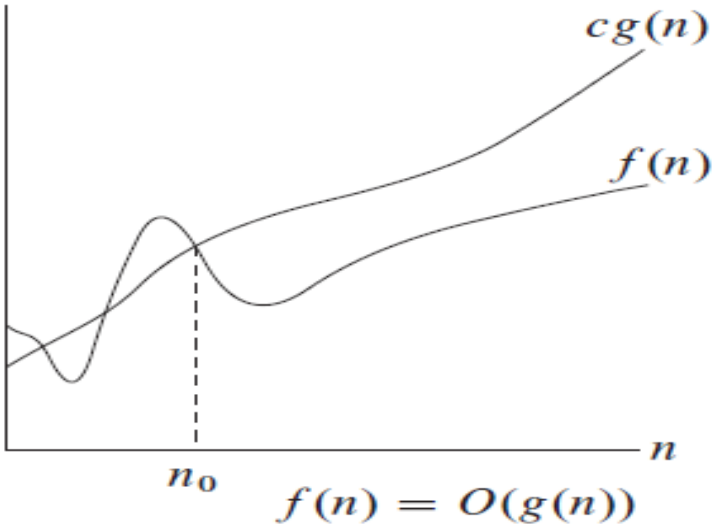
1.5.3 Asymptotic Notation (O, Ω, Θ)

- **Time complexity**
  - Depends on input size.
  - Maximum (O)
  - Minimum (Ω)
  - Maximum and Minimum (Θ)

1.5.3 Asymptotic Notation (O, Ω, Θ)

❖ **Definition [*Big "oh"*]**

- *f(n)=O(g(n)) (read as "f of n is big oh of g of n") iff* $\exists c, n_0 > 0$, *s.t.* $f(n) \leq cg(n) \ \forall n, n \geq n_0$



1.5.3 Asymptotic Notation (O)

- **Examples**

$n \geq 2$, $3n + 2 \leq 4n$ $\Rightarrow 3n + 2 = O(n)$

$n \geq 3$, $3n + 3 \leq 4n$ $\Rightarrow 3n + 3 = O(n)$

$n \geq 10$, $100n + 6 \leq 101n$ $\Rightarrow 100n + 6 = O(n)$

$n \geq 5$, $10n^2 + 4n + 2 \leq 11n^2$ $\Rightarrow 10n^2 + 4n + 2 = O(n^2)$

$n \geq 4$, $6*2^n + n^2 \leq 7*2^n$ $\Rightarrow 6*2^n + n^2 = O(2^n)$

$n \geq 2$, $3n + 3 \leq 3n^2$ $\Rightarrow 3n + 3 = O(n^2)$

$3n+2 \neq O(1)$

- for any constant c and all n, $n \geq n_0$, $3n + 2 \leq c$ is false.

$10n^2 + 4n + 2 \neq O(n)$

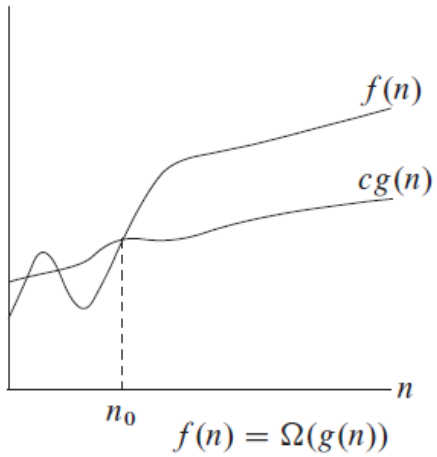
- O(1) means a computing time is a constant.

1.5.3 Asymptotic Notation (O)

- **$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$**
- ***f(n) = O(g(n)) ('=' means 'is' not 'equal')***
  - $\forall n, n \geq n_0$, $g(n)$ is an upper bound on $f(n)$
  - $g(n)$ should be as small as one can come up with $f(n)$
    - in order for the statement $f(n) = O(g(n))$ to be informative

1.5.3 Asymptotic Notation (Ω)

- **Definition [*Omega*]**
- $f(n)=\Omega(g(n))$ *iff*
- $\exists c, n_0 > 0$, s.t. $f(n) \geq cg(n) \quad \forall n, n \geq n_0$



1.5.3 Asymptotic Notation (Ω)

- **Examples**

  $n \geq 1, \quad 3n + 2 \geq 3n \quad \Rightarrow 3n + 2 = \Omega(n)$

  $n \geq 1, \quad 3n + 3 \geq 3n \quad \Rightarrow 3n + 3 = \Omega(n)$

  $n \geq 1, \quad 100n + 6 \geq 100n \quad \Rightarrow 100n + 6 = \Omega(n)$

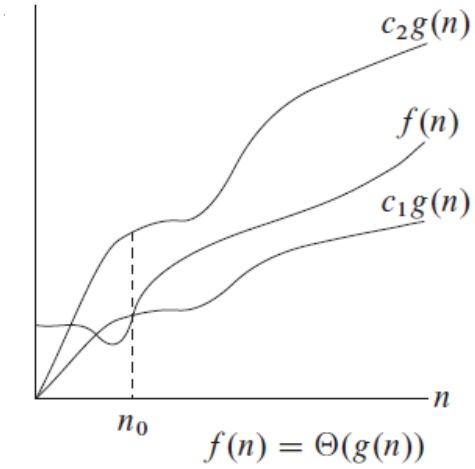
  $n \geq 1, \quad 10n^2 + 4n + 2 \geq n^2 \Rightarrow 100n^2 + 4n + 2 = \Omega(n^2)$

  $n \geq 1, \quad 6*2^n + n^2 \geq 2^n \quad \Rightarrow 6*2^n + n^2 = \Omega(2^n)$

- ***f(n) = Ω(g(n))***
  - for all n, $n \geq n_0$, $g(n)$ is a lower bound on $f(n)$
  - $g(n)$ should be as large as one can come up with

1.5.3 Asymptotic Notation (Θ)

- **Definition [*Theta*]**
- $f(n)=\Theta(g(n))$ *iff*
- $\exists c_1, c_2, n_0 > 0$, *s.t.* $c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n,$



1.5.3 Asymptotic Notation (Θ)

- **Examples**

  $n \geq 2, \quad 3n \leq 3n + 2 \leq 4n \quad \Rightarrow 3n + 2 = \Theta(n)$,
  where $c_1 = 3$, $c_2 = 4$, $n_0 = 2$

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

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

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

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

  $2*n*m + 2*n + 1 = \Theta(n*m)$

- ***f(n) = Θ(g(n))***
  - *g*(*n*) is both an upper and lower bound on *f*(*n*)

1.5.3 Asymptotic Notation (O, Ω, Θ)

- **Coefficients of all of the *g*(*n*)'s is 1 !!!**
  - We do not write O(3n), $\Omega(4n^2)$, Θ(32n)
    - but O(n), $\Omega(n^2)$, Θ(n)

1.5.3 Asymptotic Notation (O, Ω, Θ)

Theorem 1.2: If $f(n) = a_m n^m + \ldots + a_1 n + a_0$, then $f(n) = O(n^m)$.

Proof: $f(n) \leq \sum_{i=0}^{m} |a_i| n^i$

$$\leq n^m \sum_{0}^{m} |a_i| n^{i-m}$$

$$\leq n^m \sum_{0}^{m} |a_i|, \text{ for } n \geq 1$$

So, $f(n) = O(n^m)$. □

1.5.3 Asymptotic Notation (O, Ω, Θ)

- **Asymptotic complexity (O,Ω,Θ) is determined easily *without determining the exact step count*.**
  - Add up asymptotic complexities for each statement(or group of statements)
- **Example**

Statement	Asymptotic complexity
void add(int a[][MAX_SIZE] ···)	0
{	0
int i, j;	0
for (i=0; i<rows; i++)	Θ(*rows*)
for (j = 0; j < cols; j++)	Θ(*rows.cols*)
c[i][j] = a[i][j] + b[i][j];	Θ(*rows.cols*)
}	0
Total	Θ(*rows.cols*)

Figure 1.5: Time complexity of matrix addition

1.5.4 Practical Complexities

	$\log n$	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	4096	65,536
32	5	32	160	1024	32,768	4,294,967,296

Figure 1.7: Function values

1.5.4 Practical Complexities

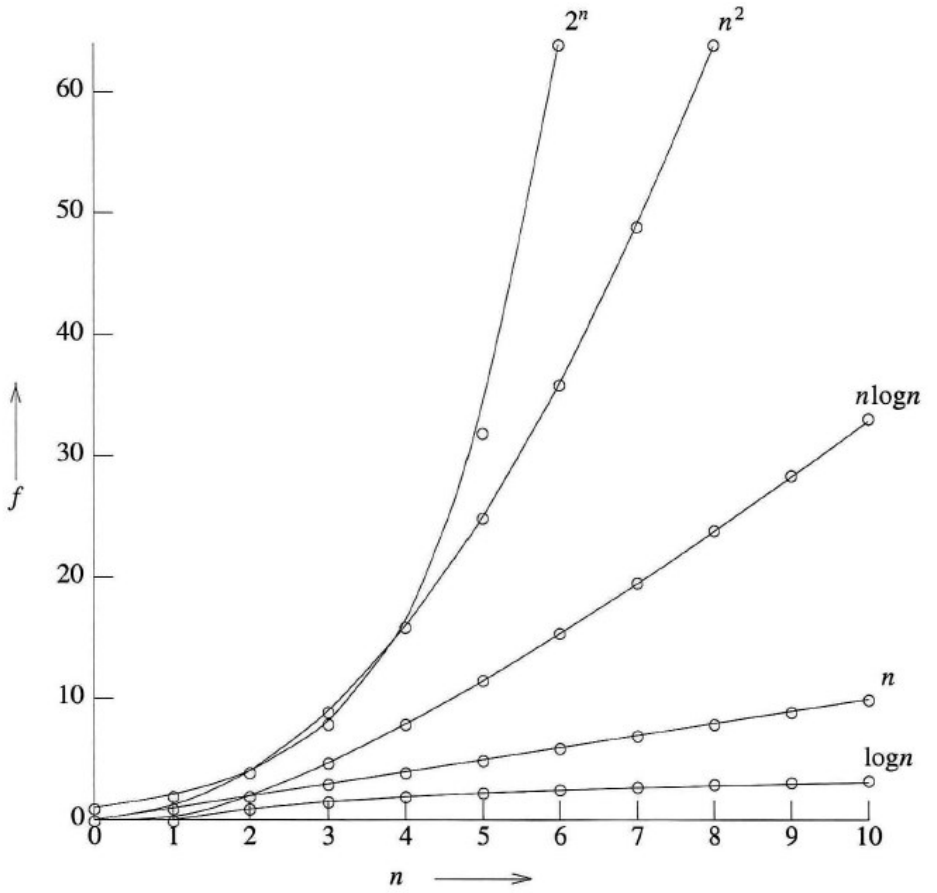


Figure 1.8 Plot of function values

1.5.4 Practical Complexities

	$f(n)$						
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	10 s	1 μs
20	.02 μs	.09 μs	.4 μs	8 μs	160 μs	2.84 h	1 ms
30	.03 μs	.15 μs	.9 μs	27 μs	810 μs	6.83 d	1 s
40	.04 μs	.21 μs	1.6 μs	64 μs	2.56 ms	121 d	18 m
50	.05 μs	.28 μs	2.5 μs	125 μs	6.25 ms	3.1 y	13 d
100	.10 μs	.66 μs	10 μs	1 ms	100 ms	3171 y	$4*10^{13}$ y
10^3	1 μs	9.96 μs	1 ms	1 s	16.67 m	$3.17*10^{13}$ y	$32*10^{283}$ y
10^4	10 μs	130 μs	100 ms	16.67 m	115.7 d	$3.17*10^{23}$ y	
10^5	100 μs	1.66 ms	10 s	11.57 d	3171 y	$3.17*10^{33}$ y	
10^6	1 ms	19.92 ms	16.67 m	31.71 y	$3.17*10^{7}$ y	$3.17*10^{43}$ y	

μs = microsecond = 10^{-6} seconds; ms = milliseconds = 10^{-3} seconds
s = seconds; m = minutes; h = hours; d = days; y = years

Figure 1.9: Times on a 1-billion-steps-per-second computer **(10^9)**

Super computer(Summit) 148.6 PF, 1PF(Peta Flops) = 10^{15}
Korea Super computer(Nurion) 25.7 PF (15^{th} rank)

KUN

1.6 Performance Measurement

- **Measure actual time on an actual computer.**
- **Data to use for measurement**
  - worst-case data
  - best-case data
  - average-case data

- **Timing mechanism – clock(), time()**

1.6 Performance Measurement

Timing events in C

- Use clock() or time() function in the C standard library.
- #include <time.h>

	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)) / CLOCKS_PER_SEC;	duration = (double) difftime(stop,start);

1.6 Performance Measurement

```
#include <stdio.h>
#include <time.h>
#include "selectionSort.h"
#define MAX_SIZE 1001
void main(void)
{
   int i, n, step = 10;
   int a[MAX_SIZE];
   double duration;
   clock_t start;

   /* times for n = 0, 10, ..., 100, 200, ..., 1000 */
   printf("    n     time\n");
   for (n = 0; n <= 1000; n += step)
   {/* get time for size n */

      /* initialize with worst-case data */
      for (i = 0; i < n; i++)
         a[i] = n - i;

      start = clock( );
      sort(a, n);
      duration = ((double) (clock() - start))
                          / CLOCKS_PER_SEC;
      printf("%6d   %f\n", n, duration);
      if (n == 100) step = 100;
   }
}
```

Program 1.24: First timing program for selection sort

1.6 Performance Measurement

```
#include <stdio.h>
#include <time.h>
#include "selectionSort.h"
#define MAX_SIZE 1001
void main(void)
{
   int i, n, step = 10;
   int a[MAX_SIZE];
   double duration;

   /* times for n = 0, 10, ..., 100, 200, ..., 1000 */
   printf("    n    repetitions     time\n");
   for (n = 0; n <= 1000; n += step)
   {
      /* get time for size n */
      long repetitions = 0;
      clock_t start = clock( );
      do
      {
         repetitions++;

         /* initialize with worst-case data */
         for (i = 0; i < n; i++)
            a[i] = n - i;

         sort(a, n);
      } while (clock( ) - start < 1000);
           /* repeat until enough time has elapsed */

      duration = ((double) (clock() - start))
                              / CLOCKS_PER_SEC;
      duration /= repetitions;
      printf("%6d  %9d   %f\n", n, repetitions, duration);
      if (n == 100) step = 100;
   }
}
```

Program 1.25: More accurate timing program for selection sort

1.6 Performance Measurement

n	repetitions	time
0	8690714	0.000000
10	2370915	0.000000
20	604948	0.000002
30	329505	0.000003
40	205605	0.000005
50	145353	0.000007
60	110206	0.000009
70	85037	0.000012
80	65751	0.000015
90	54012	0.000019
100	44058	0.000023
200	12582	0.000079
300	5780	0.000173
400	3344	0.000299
500	2096	0.000477
600	1516	0.000660
700	1106	0.000904
800	852	0.001174
900	681	0.001468
1000	550	0.001818

Figure 1.11: Worst-case performance of selection sort (seconds)

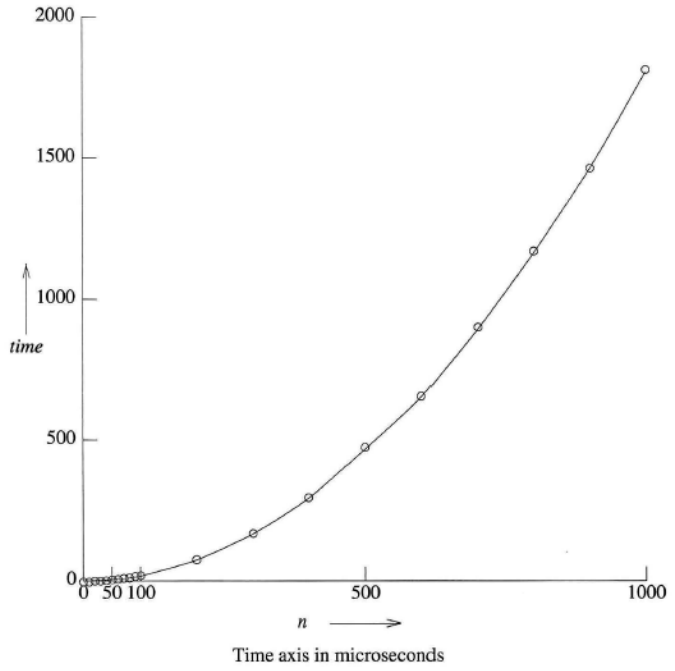


Figure 1.12: Graph of worst-case performance of selection sort

1.6 Performance Measurement

- Performance Analysis
- Performance Measurement


Have a nice day !!