

Data Structure + Data Structure Application

Professor: 박 영철

학기: 2017년도 1학기

Text: Horowitz, Shani, Anderson-Freed,

Fundamentals of data structures in C,(2nd Edition)Silicon-press, 2007.

강의자료: Powerpoint 슬라이드와 판서 이용.

Evaluation:

Mid-Term (30%), Final Exam (30%),

출석 (15%: 출석 1 회하면 0.5% 씩 점수 부여. 단, 중간고사와 기말고사는 각각 1%씩 부여),

과제 (25%: 일일 총점대비 약 1%의 과제부여).

수업진행 : 월요일, 수요일 수업 방법이 동일함. (1시간 30분: 강의, 2시간: 실습)

출석인정: 수업종료 전에 교수 또는 Tutor의 허락 없이 퇴실하면 출석점수를 받지 못함.

실습 tutor: 학부생 tutor(3명(2013학번): 이지원, 김광환, 이규동)

TA: 김동용

과제:

수업시간에 다 한 사람은 tutor에게 검사 받으면 완료됨.

먼저 제출한 학생은 남은 시간 동안 자료구조 관련 개인학습 또는 동료들을 도울 것.

수업시간 내에 완성하지 못할 경우, 다음날 자정까지 ABEEK에 업로드(20% 감점 처리함).

다음날 자정 이후 제출은 0 점 처리함.

선수과목: 기초프로그래밍&실습(C 언어)

Chapter 1. Basic Concepts

1.1 OVERVIEW: System Life Cycle

A **software development process** called the **System Life Cycle** consists of the following phases:

(1) Requirement specification

- 1) The information that we, the programmers, are given (**input**)
- 2) The results that we must produce (**output**).

(2) Problem analysis : break the problem down into manageable pieces.

- Bottom-Up Analysis (X)
- Top-Down Analysis (O)

(3) System design : The **abstract data types** and the **algorithm specifications**

- 1) The **Data Objects** that the program needs
- 2) The **Operations** performed on them.

(4) Refinement and Coding

- 1) Choose representations for our data objects
- 2) Write algorithms for each operation on them.

(5) Verification

- 1) Develop **correctness proofs** for the program
- 2) **Test the program with a variety of input data**
- 3) Remove errors.

1.2 Pointers and Dynamic Memory Allocation

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main()
{
    int i, *pi;
    float f, *pf;
    pi = (int *) malloc(sizeof(int));
    if (pi == NULL) { exit(1); }
    pf = (float *) malloc(sizeof(float));
    if (!pf) { free(pi); exit(1); }
    *pi = 1024;
    *pf = 3.14;
    printf("an integer = %d, a float = %f\n", *pi, *pf);
    free(pi);
    free(pf);
    i = 10; f = 3.141592;
    pi = &i; pf = &f;
    printf("an integer = %d, a float = %f\n", *pi, *pf);
    return 0;
}
```

Program 1.1: Allocation and deallocation of memory

& : the address operator

*** : the dereferencing (or indirection) operator**

```
an integer = 1024, a float = 3.140000
an integer = 10, a float = 3.141592
```

In Program 1.1 if we insert the line:

```
pf = (float *) malloc(sizeof(float));
```

immediately after the printf statement, then the pointer to the storage used to hold the value 3.14 has disappeared. Now there is no way to retrieve this storage. This is an example of a **dangling reference**. Whenever all pointers to a dynamically allocated area of storage are lost, the storage is lost to the program. As we examine programs that make use of pointers and dynamic storage, we will make it a point to always return storage after we no longer need it.

1.3 Algorithm Specification

Definition: An *algorithm* is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

- (1) **Input.** There are zero or more quantities that are externally supplied.
- (2) **Output.** At least one quantity is produced.
- (3) **Definiteness.** Each instruction is clear and unambiguous.
- (4) **Finiteness.** If we trace out the instructions of an algorithm, then for all cases, **the algorithm terminates** after a finite number of steps.
- (5) **Effectiveness.** Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3); it also must be feasible.

In **computational theory**, one distinguishes between an algorithm and a program, the latter of which does not have to satisfy the fourth condition. For example, we can think of an operating system that continues in a wait loop until more jobs are entered. Such a program does not terminate unless the system crashes. Since our programs will always terminate, we will use algorithm and program interchangeably in this text.

Example 1.1 [Selection sort]: Devise a program that sorts a set of $n \geq 1$ integers. A simple solution is given by the following:

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

Problem 1: 배열 list를 아래 그림과 같이 설정한 후 아래와 같이 함수 sort()를 호출한 후 배열 list의 내용을 출력하는 프로그램을 작성하시오.

```
sort (list, 10);
```

	0	1	2	3	4	5	6	7	8	9
list	10	5	8	17	6	9	30	25	20	19

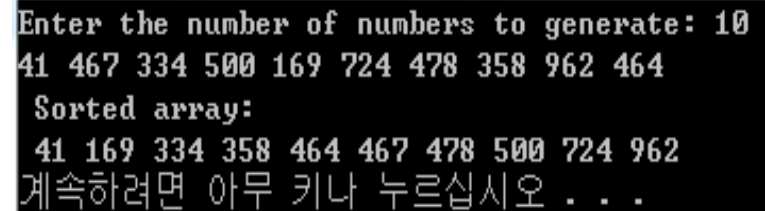
Selection sort (선택 정렬)

// Program 1.4: Selection sort

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define MAX_SIZE 101
#define SWAP(x,y,t) ( (t) = (x), (x) = (y), (y) = (t))
void sort(int [],int);
void main(void)
{
    int i,n;
    int list[MAX_SIZE];
    printf("Enter the number of numbers to generate: ");
    scanf ("%d", &n);
    if( (n < 1) || (n > MAX_SIZE)) {
        fprintf(stderr, "Improper value of n\n");
        exit(EXIT_FAILURE);
    }
    for (i = 0; i < n; i++) { /*randomly generate numbers*/
        list[i] = rand() % 1000;
        printf("%d ", list[i]);
    }
    sort (list, n);
    printf("\n Sorted array:\n ");
    for (i = 0; i < n; i++) /* print out sorted numbers */
        printf("%d ", list[i]);
    printf ( "\n");
}
```

/*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);
    }
}
```



A screenshot of a terminal window showing the output of the selection sort program. The text is as follows:

```
Enter the number of numbers to generate: 10
41 467 334 500 169 724 478 358 962 464
Sorted array:
41 169 334 358 464 467 478 500 724 962
계속하려면 아무 키나 누르십시오 . . .
```

Given

list

0	1	2	3	4	5	6	7	8	9
10	5	8	17	6	9	30	25	20	19

Selection sort

$i = 0$, comparisin = $9 = n-1$

$i = 1$, comparisin = $8 = n-2$

$i = 2$, comparisin = $7 = n-3$

$i = 3$, comparisin = $6 = n-4$

$i = 4$, comparisin = $5 = n-5$

$i = 5$, comparisin = $4 = n-6$

$i = 6$, comparisin = $3 = n-7$

$i = 7$, comparisin = $2 = n-8$

$i = 8$, comparisin = $1 = n-9$

$i = 9$

10	5	8	17	6	9	30	25	20	19
5	10	8	17	6	9	30	25	20	19
5	6	8	17	10	9	30	25	20	19
5	6	8	17	10	9	30	25	20	19
5	6	8	9	10	17	30	25	20	19
5	6	8	9	10	17	30	25	20	19
5	6	8	9	10	17	30	25	20	19
5	6	8	9	10	17	19	25	20	30
5	6	8	9	10	17	19	20	25	30
5	6	8	9	10	17	19	20	25	30
5	6	8	9	10	17	19	20	25	30

total comparisin = $n*(n-1)/2 = 45$

Example 1.2 [Binary search]: Assume that we have $n \geq 1$ distinct integers that are already sorted and stored in the array *list*. That is, $list[0] \leq list[1] \leq \dots \leq list[n-1]$. We must figure out if an integer *searchnum* is in this list. If it is we should return an index, *i*, such that $list[i] = searchnum$. If *searchnum* is not present, we should return -1.

Problem 2: 배열 *list*를 아래 그림과 같이 설정한 후 아래와 같이 함수 *binsearch()*를 호출하여 반환값 *index*를 구하는 프로그램을 작성하시오.

```
index = binsearch(list,30,0,9);
```

	0	1	2	3	4	5	6	7	8	9
list	5	6	8	9	10	17	19	20	25	30

```
#define COMPARE(x, y)  ( ( (x) < (y)) ? -1: ( (x) == (y)) ? 0: 1)
```

```
int binsearch(int list[], int searchnum, int left, int right)  
{  
    /* search list[0] <= list[1] <= ... <= list[n-1] for searchnum.  
        Return its position if found. Otherwise return -1  
    */  
    int middle;  
    while (left <= right) {  
        middle= (left + right)/2;  
        switch (COMPARE(list[middle], searchnum)) {  
            case -1: left= middle + 1; break;  
            case 0: return middle;  
            case 1: right = middle - 1; break;  
        }  
    }  
    return -1;  
}
```

Program 1.7: Searching an ordered list

Given

	0	1	2	3	4	5	6	7	8	9
list	5	6	8	9	10	17	19	20	25	30

Binary search (Iterative Version)

`binsearch(list,30,0,9)`

`middle=(0+9)/2=4`
`list[4] < 30`
`left = middle + 1`

`middle=(5+9)/2=7`
`list[7] < 30`
`left = middle + 1`

`middle=(8+9)/2=8`
`list[8] < 30`
`left = middle + 1`

Comparison: $O(\log_2 n) : \lceil \log_2 n \rceil$
 $\log_2 2^3 = 3 < \lceil \log_2 10 \rceil \leq \log_2 2^4 = 4$

`middle=(9+9)/2=9`
`list[9] == 30`
`return 9; // found!!!`

1.3.2 Recursive Algorithms

Example 1.3 [Binary search]: Program 1.7 gave **the iterative version** of a binary search. To transform this function into **a recursive one**, we must establish **boundary conditions** that terminate the recursive calls, and implement **the recursive calls** so that each call brings us one step closer to a solution.

Problem 3: 배열 `list`를 아래 그림과 같이 설정한 후 아래와 같이 함수 `binsearch()`를 호출하여 반환값 `index`를 구하는 프로그램을 작성하시오.

```
index = binsearch(list,30,0,9);
```

	0	1	2	3	4	5	6	7	8	9
list	5	6	8	9	10	17	19	20	25	30

```

#define COMPARE(x, y)  ( ( (x) < (y)) ? -1: ( (x) == (y)) ? 0: 1)

int binsearch(int list[], int searchnum, int left, int right)
{
    /* search list[0] <= list[1] <= ... <= list[n-1] for searchnum.
       Return its position if found. Otherwise return -1
    */
    int middle;
    if (left <= right) {
        middle= (left+ right) / 2;
        switch (COMPARE(list[middle], searchnum)) {
            case -1: return binsearch(list, searchnum, middle+ 1, right);
            case 0 : return middle;
            case 1 : return binsearch(list, searchnum, left, middle- 1);
        }
    }
    return -1;
}

```

Given

	0	1	2	3	4	5	6	7	8	9
list	5	6	8	9	10	17	19	20	25	30

Binary search (Recursive Version)

`binsearch(list,30,0,9)`

`middle=(0+9)/2=4`
`list[4] < 30`
`binsearch(list,30,5,9)`

`middle=(5+9)/2=7`
`list[7] < 30`
`binsearch(list,30,8,9)`

`middle=(8+9)/2=8`
`list[8] < 30`
`binsearch(list,30,9,9)`

Comparison: $\log_2 2^3 = 3 < \lceil \log_2 10 \rceil \leq \log_2 2^4 = 4$
: $O(\log_2 n)$

`middle=(9+9)/2=9`
`list[9] == 30`
`return 9; // found!!!`

Example 1.4 [Permutations]: Given a set of $n \geq 1$ elements, print out all possible permutations of this set. For example, if the set is $\{a, b, c\}$, then the set of permutations is $\{(a, b, c), (a, c, b), (b, a, c), (b, c, a), (c, a, b), (c, b, a)\}$. It is easy to see that, given n elements, there are $n!$ permutations. We can obtain a simple algorithm for generating the permutations if we look at the set $\{a, b, c, d\}$. We can construct the set of permutations by printing:

- (1) a followed by all permutations of (b, c, d)
- (2) b followed by all permutations of (a, c, d)
- (3) c followed by all permutations of (a, b, d)
- (4) d followed by all permutations of (a, b, c)

The clue to the recursive solution is the phrase "followed by all permutations." It implies that we can solve the problem for a set with n elements if we have an algorithm that works on $n - 1$ elements. These considerations lead to the development of Program 1.9. We assume that list is a character array. Notice that it recursively generates permutations until $i = n$. The initial function call is *perm* (list, 0, $n - 1$).

Problem 4: 배열 list를 아래 그림과 같이 설정한 후 아래와 같이 함수 perm()을 호출하여 그 리스트에 대한 모든 순열(permutation)을 출력하는 프로그램을 작성하시오.

perm(list, 0, 3)

	0	1	2	3
list	a	b	c	d

```
#define SWAP(x,y,t) ( (t) = (x), (x) = (y), (y) = (t) )
```

```
void perm(char *list, int i, int n)
```

```
{
    /*generate all the permutations of list[i] to list[n] */
    int j, temp;
    if (i == n) {
        for (j = 0; j <= n; j++)
            printf ( "%c", list[j] ) ;
        printf(" ");
    } else {
        /* list[i] to list[n] has more than one permutation,
           generate these recursively */
        for (j = i; j <= n; j++) {
            SWAP(list[i],list[j],temp);
            perm(list,i+1,n);
            SWAP(list[i],list[j],temp);
        }
    }
}
```

Program 1.9: Recursive permutation generator

	0	1	2	3
list	a	b	c	d

perm(list, 0, 3)

<a,b,c,d>

perm(list, 1, 3)

<b,a,c,d>

perm(list, 1, 3)

<c,b,a,d>

perm(list, 1, 3)

<d,b,c,a>

perm(list, 1, 3)

Try to simulate Program 1.9 on the three-element set {a, b, c}. Each recursive call of perm produces new local copies of the parameters *list*, *i*, and *n*. The value of *i* will differ from invocation to invocation, but *n* will not. The parameter *list* is an array pointer and its value also will not vary from call to call.

Given

	0	1	2	3
list	a	b	c	d

Permutations (Recursive Version)

perm(list,0,3) with list[a,b,c,d]

list[a,b,c,d] 0a

perm(list,1,3)

list[a,b,c,d]

list[b,a,c,d] 0b

perm(list,1,3)

list[a,b,c,d]

list[c,b,a,d] 0c

perm(list,1,3)

list[a,b,c,d]

list[d,b,c,a] 0d

perm(list,1,3)

list[a,b,c,d]

perm(list,1,3) with list[a,b,c,d] 0a

list[a,b,c,d] 1b

perm(list,2,3)

list[a,b,c,d]

list[a,c,b,d] 1c

perm(list,2,3)

list[a,b,c,d]

list[a,d,c,b] 1d

perm(list,2,3)

list[a,b,c,d]

perm(list,2,3) with list[a,b,c,d] 0a1b

list[a,b,c,d] 2c

perm(list,3,3)

list[a,b,c,d]

list[a,b,d,c] 2d

perm(list,3,3)

list[a,b,c,d]

perm(list,3,3) with list[a,b,c,d] 3d

Print a b c d

Given

	0	1	2	3
list	a	b	c	d

Permutations (Recursive Version)

(0a 1b 2c 3d) a b c d

(0a 1c 2b 3d) a c b d

(0a 1d 2c 3b) a d c b

(0a 1b 2d 3c) a b d c

(0a 1c 2d 3b) a c d b

(0a 1d 2b 3c) a d b c

(0b 1a 2c 3d) b a c d

(0b 1c 2a 3d) b c a d

(0b 1d 2c 3a) b d c a

(0b 1a 2d 3c) b a d c

(0b 1c 2a 3d) b c a d

(0b 1d 2a 3c) b d a c

(0c 1b 2a 3d) c b a d

(0c 1a 2b 3d) c a b d

(0c 1d 2a 3b) c d a b

(0c 1b 2d 3a) c b d a

(0c 1a 2d 3b) c a d b

(0c 1d 2b 3a) c d b a

(0d 1b 2c 3a) d b c a

(0d 1c 2b 3a) d c b a

(0d 1a 2c 3b) d a c b

(0d 1b 2a 3c) b a d c

(0d 1c 2a 3b) d c a b

(0d 1a 2b 3c) d a b c

Given

	0	1	2
list	a	b	c

Permutations (Recursive Version)

perm(list,0,2) with list[a,b,c]

list[a,b,c] 0a
perm(list,1,2)
list[a,b,c]

list[b,a,c] 0b
perm(list,1,2)
list[a,b,c]

list[c,b,a] 0c
perm(list,2,3)
list[a,b,c]

perm(list,1,2) with list[a,b,c]

list[a,b,c] 1b
perm(list,3,3)
list[a,b,c]

list[a,c,b] 1c
perm(list,3,3)
list[a,b,c]

perm(list,2,2) with list[a,b,c] 2c

Print a b c

(0a 1b 2c) a b c
(0b 1a 2c) b a c
(0c 1b 2a) c b a

(0a 1c 2b) a c b
(0b 1c 2a) b c a
(0c 1a 2b) c a b

EXERCISES (on Pages 16, 17, 18)

3. Given n Boolean variables x_1, \dots, x_n , we wish to print all possible combinations of truth values they can assume. For instance, if $n = 2$, there are four possibilities: $\langle \text{true}, \text{true} \rangle$, $\langle \text{false}, \text{true} \rangle$, $\langle \text{true}, \text{false} \rangle$, and $\langle \text{false}, \text{false} \rangle$. Write a C program to do this.
7. The factorial function $n!$ has value 1 when $n \leq 1$ and value $n \cdot (n-1)!$ when $n > 1$. Write both a recursive and an iterative C function to compute $n!$.
8. The Fibonacci numbers are defined as: $f_0 = 0$, $f_1 = 1$, and $f_i = f_{i-1} + f_{i-2}$ for $i > 1$. Write both a recursive and an iterative C function to compute f_i .
12. If S is a set of n elements the power set of S is the set of all possible subsets of S . For example, if $S = \{a, b, c\}$, then $\text{powerset}(S) = \{ \{\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\} \}$. Write a recursive function to compute $\text{powerset}(S)$.

Exercise 7 and 8 are well known problems in the C language area. You can get and use them.

1.4 Data Abstraction

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

• 자료형 = 객체들의 집합 + 그 객체들에 작용하는 연산들의 집합.

For example, the data type *int* consists of

(1) the *objects* $\{0, +1, -1, +2, -2, \dots, \text{INT-MAX}, \text{INT-MIN}\}$, where INT-MAX and INT-MIN are the largest and smallest integers that can be represented on your machine. (They are defined in limits.h.)

(2) The *operations* on integers are many, and would certainly include the arithmetic operators $+$, $-$, $*$, $/$, and $\%$. There is also testing for equality/inequality and the operation that assigns an integer to a variable. In all of these cases, there is the name of the operation, which may be a prefix operator, such as `atoi`, or an infix operator, such as $+$. Whether an operation is defined in the language or in a library, its name, possible arguments and results must be specified.

(3) In addition to knowing all of the facts about the operations on a data type, we might also want to know about **how the objects of the data type are represented**. For example on most computers a char is represented as a bit string occupying 1 byte of memory, whereas an int might occupy 2 or possibly 4 bytes of memory. If 2 eight-bit bytes are used, then INT-MAX is $2^{15} - 1 = 32,767$.

Knowing the representation of the objects of a data type can be useful and dangerous. By knowing the representation we can often write algorithms that make use of it. However, if we ever want to change the representation of these objects, we also must change the routines that make use of it. It has been observed by many software designers that **hiding the representation of objects of a data type** from its users is a good design strategy. In that case, the user is constrained to manipulate the objects solely through the functions that are provided. The designer may still alter the representation as long as the new implementations of the operations do not change the user interface. This means that users will not have to recode their algorithms.

Definition: An **abstract data type (ADT)** is a data type that is organized in such a way that **the specification of the objects and the specification of the operations on the objects is separated from the representation of the objects and the implementation of the operations.**

- 추상자료형(Abstract Data Type)

(객체들의 명세 + 그 객체들에 작용하는 연산들의 명세)을 함께 선언하며,
그 선언이

(그 객체 표현의 구현 + 그 연산들의 구현)과는 분리되어 있는 것.

- 자료 캡슐화(Data Encapsulation): 캡슐화 구현 종류: class, structure, ...

Some programming languages provide explicit mechanisms to support the distinction between **specification** and **implementation**. For example, **Ada** has a concept called a *package*, and **C++** has a concept called a *class*. Both of these assist the programmer in implementing abstract data types. Although **C** does not have an explicit mechanism for implementing ADTs, it is still possible and desirable to design your data types using the same notion.

How does the specification of the operations of an ADT differ from the implementation of the operations? The specification consists of the names of every function, the type of its arguments, and the type of its result. There should also be a description of what the function does, but without appealing to internal representation or implementation details. This requirement is quite important, and it implies that an abstract data type is implementation-independent. Furthermore, it is possible to classify the functions of a data type into several categories:

- (1) **Creator/constructor**: These functions create a new instance of the designated type.
- (2) **Transformers**: These functions also create an instance of the designated type, generally by using one or more other instances. The difference between constructors and transformers will become more clear with some examples.
- (3) **Observers/reporters**: These functions provide information about an instance of the type, but they do not change the instance.

Typically, an ADT definition will include at least one function from each of these three categories.

Throughout this text, we will emphasize the distinction between **specification** and **implementation**. In order to help us do this, we will typically begin with **an ADT definition of the object** that we intend to study. This will permit the reader to grasp the essential elements of the object, without having the discussion complicated by **the representation of the objects** or by **the actual implementation of the operations**. Once the ADT definition is fully explained we will move on to discussions of representation and implementation. These are quite important in the study of data structures. In order to help us accomplish this goal, we introduce a notation for expressing an ADT.

Example 1.5 [Abstract data type *NaturalNumber*]

ADT 1.1 contains **the ADT definition** of *NaturalNumber*.

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 $x, y \in \textit{NaturalNumber}$, $TRUE, FALSE \in \textit{Boolean}$ 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(x) ::= 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*

For each function, we place the **result type** to the left of the function name and a **definition** of the function to the right. The symbols "**::=**" should be read as "**is defined as**."

- The first function, Zero, has no arguments and returns the natural number zero. This is **a constructor function**.
- The function Successor(x) returns the next natural number in sequence. This is an example of **a transformer function**. Notice that if there is no next number in sequence, that is, if the value of x is already INT_MAX, then we define the action of Successor to return INT_MAX. Some programmers might prefer that in such a case Successor return an error flag. This is also perfectly permissible.
- Other **transformer functions** are Add and Subtract. They might also return an error condition, although here we decided to return an element of the set NaturalNumber.

1.5 Performance Analysis

Performance Evaluation Phases

- **Performance analysis** : a priori estimates
 - obtaining estimates of time and space that are *machine independent*
- **Performance measurement** : a posteriori testing
 - obtaining *machine-dependent* running time

Definition: 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.

1.5.1 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, simple variables, constants, and fixed-size structured variables

- 2) variable space requirements**

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

- The variable space requirement of a program P working on an instance I is denoted $S_P(I)$. $S_P(I)$ is usually given as a function of some *characteristics* of the instance I . Commonly used characteristics include the number, size, and values of the inputs and outputs associated with I .
- For example, if our input is an array containing n numbers then n is an instance characteristic. If n is the only instance characteristic we wish to use when computing $S_P(I)$, we will use $S_P(n)$ to represent $S_P(I)$.
- $S(P) = c + S_P(I)$
 - $S(P)$: *total space requirement of a program P*
 - c : constant for fixed space requirement
 - $S_P(I)$: function of characteristics of the instance I , where characteristics include number, size, values of I/O associated with I .
- When analyzing the space complexity of a program we are usually concerned with only **the variable space requirements**. This is particularly true when we want to compare the space complexity of several programs.

Example 1.6: a function, `abc` (Program 1.10), which accepts three simple variables as input and returns a simple value as output. It has only fixed space requirements. Therefore, $S_{abc}(I) = 0$.

```
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

Example 1.7: We want to add a list of numbers (Program 1.11). Although the output is a simple value, the input includes an array. Therefore, the variable space requirement depends on **how the array is passed into the function**.

- Programming languages like **Pascal** may pass arrays by value. This means that the entire array is copied into temporary storage before the function is executed. In these languages the variable space requirement for this program is $S_{sum}(I) = S_{sum}(n) = n$, where n is the size of the array.
- **C passes all parameters by value**. When an array is passed as an argument to a function, C interprets it as passing the address of the first element of the array. C does not copy the array. Therefore, $S_{sum}(n) = 0$.

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

Program 1.11: Iterative function for summing a list of numbers

Example 1.8: Program 1.12 also adds a list of numbers, but this time the summation is handled recursively. This means that the compiler must **save the parameters, the local variables, and the return address for each recursive call.**

In this example, the space needed for one recursive call is **the number of bytes required for the two parameters and the return address.** We can use the *sizeof* function to find the number of bytes required by each type. Figure 1.1 shows the number of bytes required for one recursive call under the assumption that an integer and a pointer each require 4 bytes.

```
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

Type	Name	Number of bytes
parameter: array pointer	<i>list[]</i>	4
parameter: integer	<i>n</i>	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 $n = \text{MAX_SIZE}$ numbers, the total variable space needed for the recursive version is $S_{rsum}(\text{MAX_SIZE}) = 12 * \text{MAX_SIZE}$. If $\text{MAX_SIZE} = 1000$, the variable space needed by the recursive version is $12 * 1000 = 12,000$ bytes. The iterative version has no variable space requirement. As you can see, the recursive version has a far greater overhead than its iterative counterpart.

1.5.2 Time Complexity

The time, $T(P)$, taken by a program, P , is the sum of its *compile time* and its *run (or execution) time*. The compile time is similar to the fixed space component since it does not depend on the instance characteristics. In addition, once we have verified that the program runs correctly, we may run it many times without recompilation. Consequently, we are really concerned only with the program's execution time, T_p .

$$T(P) = c + T_p(n)$$

- $T(P)$: the time taken by a program P .
- c : **compile time**
- T_p : **execution time** (or run time: computation of execution time by using program step)
- n : instance characteristics

Determining T_p is not an easy task because it requires a detailed knowledge of the compiler's attributes. That is, we must know how the compiler translates our source program into object code. For example, suppose we have a simple program that adds and subtracts numbers. Letting n denote the instance characteristic, we might express $T_p(n)$ as:

$$T_p(n) = c_a \text{ADD}(n) + c_s \text{SUB}(n) + c_l \text{LDA}(n) + c_{st} \text{STA}(n)$$

where c_a , c_s , c_l , c_{st} are constants that refer to the time needed to perform each operation, and ADD, SUB, LDA, STA are the number of additions, subtractions, loads, and stores that are performed when the program is run with instance characteristic n .

Obtaining such a detailed estimate of running time is rarely worth the effort. If we must know the running time, the best approach is to use the **system clock** to time the program. We will do this later in the chapter. Alternately, we could count the number of operations the program performs. This gives us a machine-independent estimate, but we must know how to divide the program into distinct steps.

Definition: A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

Note that the amount of computing represented by one program step may be different from that represented by another step. So, for example, we may count a simple assignment statement of the form $a = 2$ as one step and also count a more complex statement such as $a = 2*b + 3*c/d - e + f/g/a/b/c$ as one step. The only requirement is that the time required to execute each statement that is counted as one step be independent of the instance characteristics.

We can determine **the number of steps** that a program or a function needs to solve a particular problem instance by creating a global variable, *count*, which has an initial value of 0 and then inserting statements that increment count by the number of program steps required by each **executable statement**.

Example 1.9: [Iterative summing of a list of numbers]:

We want to **obtain the step count** for the *sum* function discussed earlier (Program 1.11). Program 1.13 shows where to place the *count* statements. Notice that we only need to worry about the executable statements, which automatically eliminates the function header, and the second variable declaration from consideration.

statement	steps for executable statements	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

Example 1.10: [**Recursive summing** of a list of numbers]:

We want to obtain the step count for the recursive version of the summing function. Program 1.15 contains the original function (Program 1.12) with the step counts added.

statement	s/e	freq	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 list[0];	1	1	1
}	0	0	0
total			2n+2

Surprisingly, the recursive function actually has a lower step count than its iterative counterpart. However, we must remember that the step count only tells us how many steps are executed, it does not tell us how much time each step takes. Thus, although the recursive function has fewer steps, it typically runs more slowly than the iterative version as its steps, on average, take more time than those of the iterative version.

Example 1.11: [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

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	$rows \cdot (cols+1)$	$rows \cdot cols + rows$
c[i][j] = a[i][j] + b[i][j];	1	$rows \cdot cols$	$rows \cdot cols$
}	0	0	0
Total			$2rows \cdot cols + 2rows+1$

Figure 1.4: Step count table for matrix addition

Exercise

- ♦ Write the step count table

statement	steps	freq	total steps
<pre>void printMatrix(int matrix[] [MAX-SIZE], int rows, int cols) { int i, j; for (i = 0; i < rows; i++) { for (j = 0; j < cols; j++) printf("%d",matrix[i] [j]); printf (" \n"); } }</pre>			
total			

Exercise

- ♦ Write the step count table

statement	steps	freq	total steps
<pre>void mult(int a[] [MAX-SIZE], int b[] [MAX-SIZE], int c[] [MAX-SIZE]) { int i, j, k; for (i = 0; i < MAX-SIZE; i++) for (j = 0; j < MAX-SIZE; j++) { c[i][j] = 0; for (k = 0; k < MAX-SIZE; k++) c[i][j] += a[i][k]* b[k][j]; } }</pre>			
total			

1.5.3 Asymptotic Notation (O , Ω , Θ)

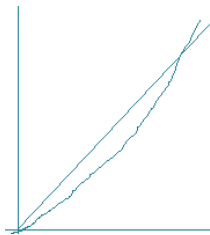
Our motivation to determine **step counts** is

- to be able to **compare** the time complexities of two programs that compute the same function and
- also to **predict the growth** in run time as the instance characteristics change.

Determining the exact step count (either **worst case** or **average**) of a program can prove to be an exceedingly **difficult task**. Expending immense effort to determine the step count exactly **isn't a very worthwhile endeavor** as the notion of a step is itself inexact. (Both the instructions $x = y$ and $x = y + z + (x/y) + (x*y *z - x/z)$ count as one step.) Because of the inexactness of what a step stands for, the exact step count isn't very useful for comparative purposes.

An exception to this is when the difference in the step counts of two programs is very large as in $3n+3$ versus $100n+10$. We might feel quite safe in predicting that the program with step count $3n+3$ will run in less time than the one with step count $100n+10$. **But even in this case, it isn't necessary to know that the exact step count is $100n+10$. Something like, "it's about $80n$, or $85n$, or $75n$," is adequate to arrive at the same conclusion.**

- For most situations, it is adequate to be able to make a statement like $c_1n^2 \leq Tp(n) \leq c_2n^2$ or $T_Q(n,m) = c_1n + c_2m$ where c_1 and c_2 are nonnegative constants.
- This is so because if we have two programs with a complexity of $c_1n^2 + c_2n$ and c_3n , respectively, then we know that the one with complexity c_3n will be faster than the one with complexity $c_1n^2 + c_2n$ for sufficiently large values of n . For small values of n , either program could be faster (depending on c_1 , c_2 , and c_3).
- If $c_1 = 1$, $c_2 = 2$, and $c_3 = 100$ then $c_1n^2 + c_2n \leq c_3n$ for $n \leq 98$ and $c_1n^2 + c_2n > c_3n$ for $n > 98$.



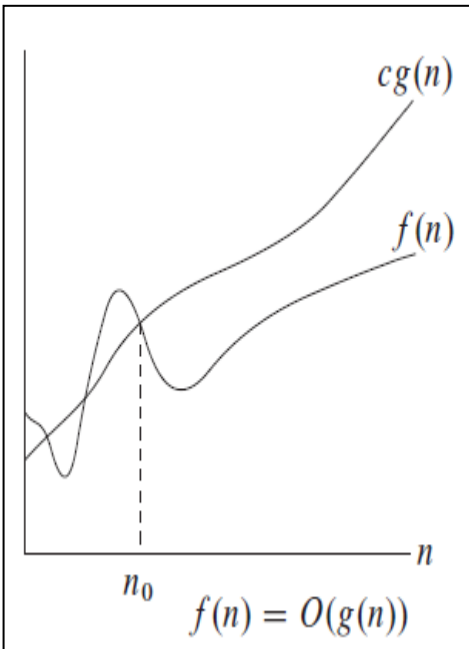
break even point : $n = 98$

- If $c_1 = 1$, $c_2 = 2$, and $c_3 = 1000$, then $c_1n^2 + c_2n \leq c_3n$ for $n \leq 998$.

In the remainder of this chapter, the functions f and g are **nonnegative functions**.

Definition: [*Big “oh”*]

$f(n)=O(g(n))$ (read as “ f of n *is* big oh of g of n ”) iff (if and only if) $\exists c, n_0 > 0$, such that $f(n) \leq cg(n) \quad \forall n, n \geq n_0$.



- $f(n) = O(g(n))$ ('=' means '*is*' not 'equal')
 - $g(n)$ is **an upper bound on** the value of $f(n)$ for all $n, n \geq n_0$.
 - In order for the statement $f(n) = O(g(n))$ to be informative, $g(n)$ **should be as small a function of n as one can** come up with for which $f(n) = O(g(n))$.

Some comments:

- So, while **we shall often say** $3n + 3 = O(n)$, **we shall almost never say** $3n + 3 = O(n^2)$ even though this latter statement is correct.
- From the definition of O , it should be clear that $f(n) = O(g(n))$ is not the same as $O(g(n)) = f(n)$. In fact, it is meaningless to say that $O(g(n)) = f(n)$. The use of the symbol "=" is unfortunate as this symbol commonly denotes the "equals" relation.

Example 1.15:

- $3n + 2 = O(n)$ as $3n + 2 \leq 4n$ for all $n \geq 2$.
- $3n + 3 = O(n)$ as $3n + 3 \leq 4n$ for all $n \geq 3$.
- $100n + 6 = O(n)$ as $100n + 6 \leq 101n$ for all $n \geq 10$.
- $10n^2 + 4n + 2 = O(n^2)$ as $10n^2 + 4n + 2 \leq 11n^2$ for all $n \geq 5$.
- $6 \cdot 2^n + n^2 = O(2^n)$ as $6 \cdot 2^n + n^2 \leq 7 \cdot 2^n$ for all $n \geq 4$.
- $3n + 3 = O(n^2)$ as $3n + 3 \leq 3n^2$ for all $n \geq 2$.
- $10n^2 + 4n + 2 = O(n^4)$ as $10n^2 + 4n + 2 \leq 10n^4$ for all $n \geq 2$.
- $3n + 2 \neq O(1)$ as $3n + 2$ is not less than or equal to c for any constant c and all n , $n \geq n_0$.
- $10n^2 + 4n + 2 \neq O(n)$.

Seven computing times that we will see most often in this book.

$$\mathbf{O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)}$$

- $O(1)$: constant
- $O(\log n)$: logarithmic
- $O(n)$: linear
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(2^n)$: exponential

Theorem 1.2 obtains a very useful result concerning the order of $f(n)$ (i.e., the $g(n)$ in $f(n) = O(g(n))$) when $f(n)$ is a polynomial in n .

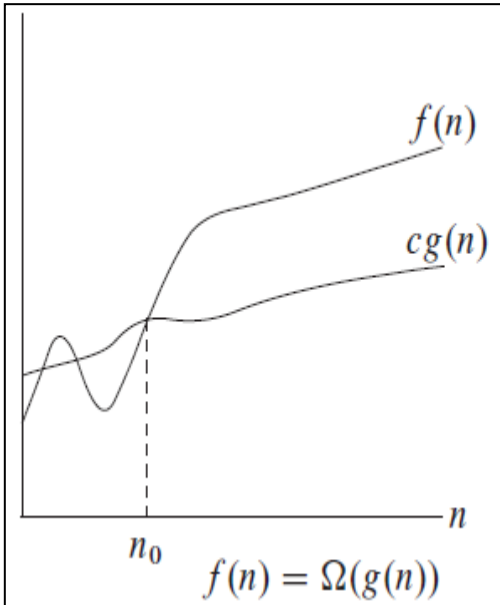
Theorem 1.2: If $f(n) = a_m n^m + \dots + a_1 n + a_0$, then $f(n) = O(n^m)$.

$$\begin{aligned}\text{Proof: } f(n) &\leq \sum_{i=0}^m |a_i| n^i \\ &\leq n^m \sum_{i=0}^m |a_i| n^{i-m} \\ &\leq n^m \sum_{i=0}^m |a_i|, \text{ for } n \geq 1\end{aligned}$$

So, $f(n) = O(n^m)$.

Definition: [*Omega*]

$f(n) = \Omega(g(n))$ (read as “ f of n *is* omega of g of n ”)
iff there exist positive constant c and n_0 , such that $f(n) \geq cg(n)$ for all $n, n \geq n_0$.



- $f(n) = \Omega(g(n))$
 - $g(n)$ is only **a lower bound on** the value of $f(n)$ for all $n, n \geq n_0$.
 - For the statement $f(n) = \Omega(g(n))$ to be informative, $g(n)$ **should be as large a function of n as possible** for which the statement $f(n) = \Omega(g(n))$ is true.

Some comments:

- So, while **we shall say** that $3n + 3 = \Omega(n)$ and that $6 \cdot 2^n + n^2 = \Omega(2^n)$, we **shall almost never say** that $3n + 3 = \Omega(1)$ or that $6 \cdot 2^n + n^2 = \Omega(1)$ even though both these statements are correct.

Example 1.17:

- $3n + 2 = \Omega(n)$ as $3n + 2 \geq 3n$ for $n \geq 1$ (actually the inequality holds for $n \geq 0$ but the definition of Ω requires an $n_0 > 0$).
- $3n + 3 = \Omega(n)$ as $3n + 3 \geq 3n$ for $n \geq 1$.
- $100n + 6 = \Omega(n)$ as $100n + 6 \geq 100n$ for $n \geq 1$.
- $10n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \geq n^2$ for $n \geq 1$.
- $6 \cdot 2^n + n^2 = \Omega(2^n)$ as $6 \cdot 2^n + n^2 \geq 2^n$ for $n \geq 1$.
- $6 \cdot 2^n + n^2 = \Omega(n)$.
- $6 \cdot 2^n + n^2 = \Omega(1)$.

Observe also that

- $3n + 3 = \Omega(1)$;
- $10n^2 + 4n + 2 = \Omega(n)$;
- $10n^2 + 4n + 2 = \Omega(1)$;
- $6 \cdot 2^n + n^2 = \Omega(n^{100})$;
- $6 \cdot 2^n + n^2 = \Omega(n^{50.2})$;
- $6 \cdot 2^n + n^2 = \Omega(n^2)$;

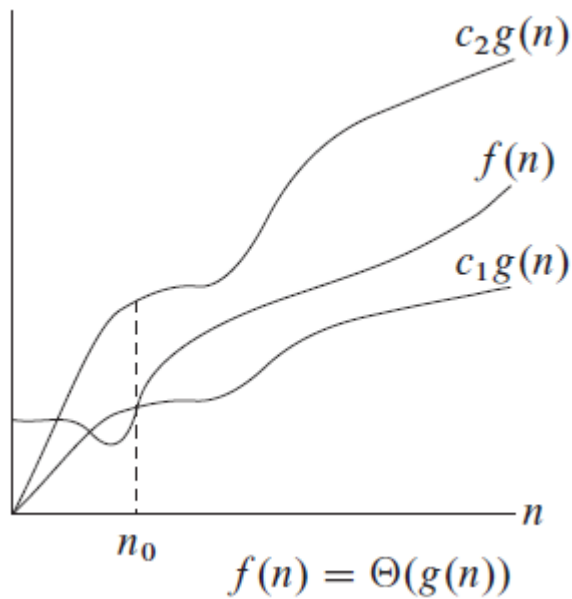
Theorem 1.3 is the analogue of Theorem 1.2 for the omega notation.

Theorem 1.3: If $f(n) = a_m n^m + \dots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$.

Proof: Left as an exercise.

Definition: [*Theta*]

$f(n) = \Theta(g(n))$ (read as “ f of n *is* theta of g of 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 , $n \geq n_0$.



- The theta notation is more precise than both the "big oh" and omega notations.
- $f(n) = \Theta(g(n))$
 - $g(n)$ is **both an upper and lower bound on** the values of $f(n)$ for all n , $n \geq n_0$.

Graphic Examples of the O , Ω , Θ Notations

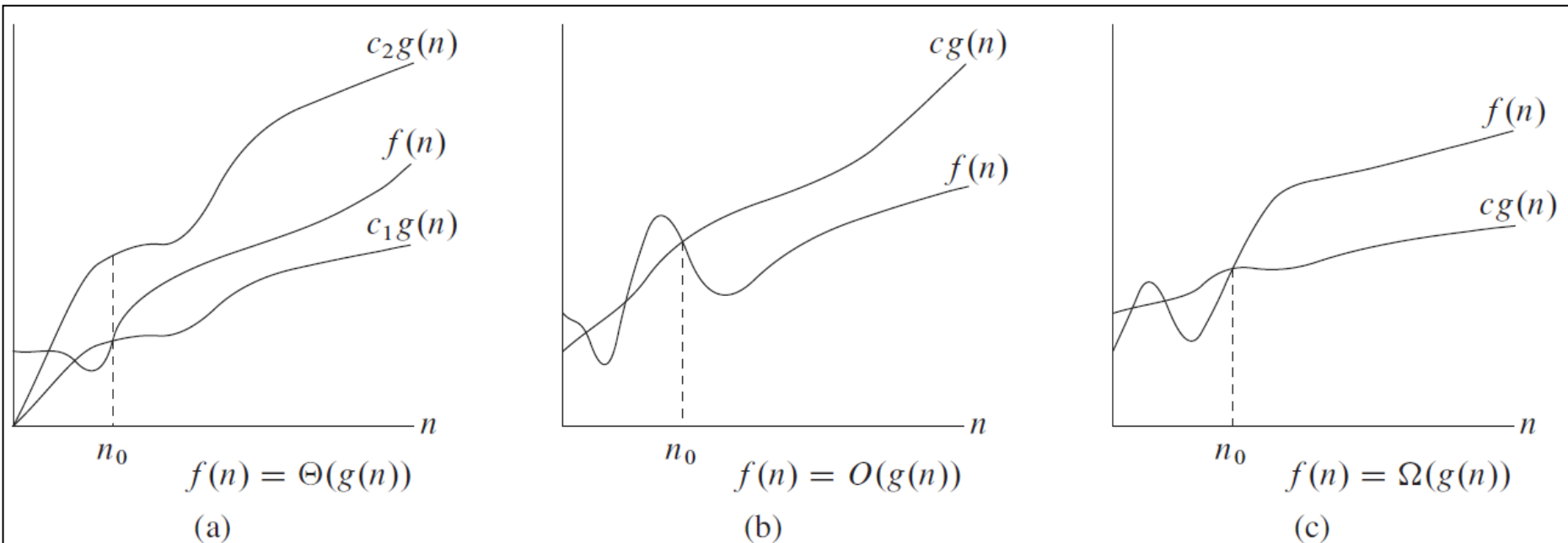


Figure 3.1 Graphic examples of the Θ , O , and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. **(a)** Θ -notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0 , c_1 , and c_2 such that at and to the right of n_0 , the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive. **(b)** O -notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0 , the value of $f(n)$ always lies on or below $cg(n)$. **(c)** Ω -notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0 , the value of $f(n)$ always lies on or above $cg(n)$.

Example 1.17:

- $3n + 2 = \Theta(n)$ as $3n + 2 \geq 3n$ for all $n \geq 2$ and $3n + 2 \leq 4n$ for all $n \geq 2$, so $c_1 = 3$, $c_2 = 4$, and $n_0 = 2$.
- $3n + 3 = \Theta(n)$
- $10n^2 + 4n + 2 = \Theta(n^2)$
- $6 \cdot 2^n + n^2 = \Theta(2^n)$
- $10 \cdot \log n + 4 = \Theta(\log n)$
- $2 \cdot n \cdot m + 2 \cdot n + 1 = \Theta(n \cdot m)$.

- $3n + 2 \neq \Theta(1)$;
- $3n + 3 \neq \Theta(n^2)$;
- $10n^2 + 4n + 2 \neq \Theta(1)$
- $6 \cdot 2^n + n^2 \neq \Theta(n^2)$
- $6 \cdot 2^n + n^2 \neq \Theta(n^{100})$, and
- $6 \cdot 2^n + n^2 \neq \Theta(1)$.

- Notice that the coefficients in all of the $g(n)$'s used in the preceding three examples (examples 1.15, 1.16, 1.17) has been 1. This is in accordance with practice. We shall almost never find ourselves saying that $3n + 3 = O(3n)$, or that $10 = O(100)$, or that $10n^2 + 4n + 2 = \Omega(4n^2)$, or that $6 \cdot 2^n + n^2 = \Omega(6 \cdot 2^n)$, or that $6 \cdot 2^n + n^2 = \Theta(4 \cdot 2^n)$, even though each of these statements is true.

Theorem 1.4: If $f(n) = a_m n^m + \dots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Theta(n^m)$.

Proof: Left as an exercise.

Let us reexamine the time complexity analyses of the previous section. For function *sum* (Program 1.12) we had determined that $T_{sum}(n) = 2n + 3$. So, $T_{sum}(n) = \Theta(n)$. $T_{rsum}(n) = 2n + 2 = \Theta(n)$ and $T_{add}(rows, cols) = 2rows.cols + 2rows + 1 = \Theta(rows.cols)$.

While we might all see that the O , Ω , and Θ notations have been used correctly in the preceding paragraphs, we are still left with the question: "Of what use are these notations if one has to first determine the step count exactly?" The answer to this question is that the asymptotic complexity (i.e., the complexity in terms of O , Ω , and Θ) can be determined quite easily without determining the exact step count. This is usually done by first determining the asymptotic complexity of each statement (or group of statements) in the program and then adding up these complexities.

Example 1.18 [Complexity of matrix addition]

statement	asymptotic complexity
<pre>void add(int a[][MAX_SIZE] ...) { int i, j; for (i=0; i < rows; i++) for (j=0; j < cols; j++) c[i][j] = a[i][j] + b[i][j]; }</pre>	<pre>0 0 0 $\Theta(\text{rows})$ $\Theta(\text{rows} \cdot \text{cols})$ $\Theta(\text{rows} \cdot \text{cols})$ 0</pre>
total	$\Theta(\text{rows} \cdot \text{cols})$

Figure 1.5: Time complexity of matrix addition

Example 1.19 [Binary search]

```
int binsearch(int list[], int searchnum, int left, int right)
{
    int middle;
    while (left <= right) {
        middle = (left + right) / 2;
        switch (COMPARE(list[middle], searchnum)) {
            case -1: left = middle + 1;
                    break;
            case 0: return middle;
            case 1: right = middle - 1;
        }
    }
    return -1;
}
```

Program 1.7: Searching an ordered list

- Each iteration of the while loop takes $\Theta(1)$ time.
- the while loop is iterated at most $\lceil \log_2(n+1) \rceil$ times.
- this loop is iterated $\Theta(\log n)$ times in the worst case.
- As each iteration takes $\Theta(1)$ time, the overall worst-case complexity of binsearch is $\Theta(\log n)$.
- Notice that the best case complexity is $\Theta(1)$ as in the best case searchnum is found in the first iteration of the while loop.

Figure 1.5: Time complexity of matrix addition

Example 1.20 [Permutations]

```
void perm(char *list, int i, int n)
{
    int j, temp;
    if (i == n) {
        for (j = 0; j <= n; j++) printf ("%c", list[j]);
        printf(" ");
    } else {
        for (j = i; j <= n; j++) {
            SWAP(list[i],list[j],temp);
            perm(list,i+1,n);
            SWAP(list[i],list[j],temp);
        }
    }
}
```

Program 1.9: Recursive permutation generator

- When $i = n$, the time taken is $\Theta(n)$.
- When $i < n$, the else clause is entered. The for loop of this clause is entered $n - i + 1$ times. Each iteration of this loop takes $\Theta(n + T_{perm}(i + 1, n))$ time.
- So, $T_{perm}(i, n) = \Theta((n - i + 1)(n + T_{perm}(i + 1, n)))$ when $i < n$. Since, $T_{perm}(i + 1, n)$, is at least n when $i + 1 \leq n$, we get $T_{perm}(i, n) = \Theta((n - i + 1) T_{perm}(i + 1, n))$ for $i < n$. Solving this recurrence, we obtain $T_{perm}(1, n) = \Theta(n(n!))$, $n \geq 1$.
- 경우의 수 = $n!$
- 각 경우의 수를 출력 = n
- So, $\Theta(n(n!))$

Example 1.21 [Magic square]

방진: 가로와 세로의 크기가 같은 정사각형의 칸에 1부터 시작하는 숫자를 차례로 배열하되 모든 행, 열 그리고 두 대각선상의 숫자의 합이 모두 같게 한 것. 예를 들어 4차의 방진은 다음과 같다.

```
#include <stdio.h>
#define MAX_SIZE 15 /* maximum size of square */
void main(void)
{ /* construct a magic square, iteratively */
    int square[MAX_SIZE][MAX_SIZE];
    int i, j, row, column; /* indexes */
    int count; /* counter */
    int size; /* square size */
    printf("Enter the size of the square: ");
    scanf("%d", &size);
    /* check for input errors */
    if (size < 1 || size > MAX_SIZE + 1) {
        fprintf(stderr, "Error! Size is out of range \ n");
        exit(EXIT_FAILURE);
    }
    if (!(size%2)) {
        fprintf(stderr, "Error! Size is even\n");
        exit(EXIT_FAILURE);
    }
    for (i = 0; i < size; i++)
        for (j = 0; j < size; j++)
            square [ i ] [ j ] = 0;
    square[0] [(size-1) / 2] = 1; /*middle of first row*/
```

```
/* i and j are current position */
i = 0;
j = (size - 1) / 2;
for (count = 2; count <= size* size; count++) {
    row = (i-1 < 0) ? (size - 1) : (i- 1); /*up*/
    column = (j-1 < 0) ? (size - 1) : (j- 1); /*left*/
    if (square[row] [column]) /*down*/
        i = (++i) % size;
    else { /* square is unoccupied */
        i = row;
        j = (j-1 < 0) ? (size - 1) : --j;
    }
    square[i] [j] =count;
}
/* output the magic square */
printf(" Magic Square of size %d : \n\n", size);
for (i = 0; i < size; i++) {
    for (j = 0; j < size; j++)
        printf("%5d", square[i] [j]);
    printf("\n");
}
printf("\n\n");
}
```

Program 1.23: Magic square program

방진

1	15	8	10
6	12	3	13
11	5	14	4
16	2	9	7

As our last example of complexity analysis, we use a problem from recreational mathematics, the creation of a magic square. A magic square is an $n \times n$ matrix of the integers from 1 to n^2 such that the sum of each row and column and the two major diagonals is the same. Figure 1.6 shows a magic square for the case $n = 5$. In this example, the common sum is 65.

Coxeter has given the following rule for generating a magic square when n is odd:

- Put a one in the middle box of the top row.
- Go up and left assigning numbers in increasing order to empty boxes.
- If your move causes you to jump off the square (that is, you go beyond the square's boundaries), figure out where you would be if you landed on a box on the opposite side of the square.
- Continue with this box.
- If a box is occupied, go down instead of up and continue.

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

Figure 1.6: Magic square for $n = 5$

We created Figure 1.6 using Coxeter's rule. Program 1.23 contains the coded algorithm. Let n denote the size of the magic square (i.e., the value of the variable size in Program 1.23. The if statements that check for errors in the value of n take $\Theta(1)$ time. The two nested for loops have a complexity $\Theta(n^2)$. Each iteration of the next for loop takes $\Theta(1)$ time. This loop is iterated $\Theta(n^2)$ time. So, its complexity is $\Theta(n^2)$. The nested for loops that output the magic square also take $\Theta(n^2)$ time. So, the asymptotic complexity of Program 1.23 is $\Theta(n^2)$.

1.5.4 Practical Complexities

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

Figure 1.7: Function values

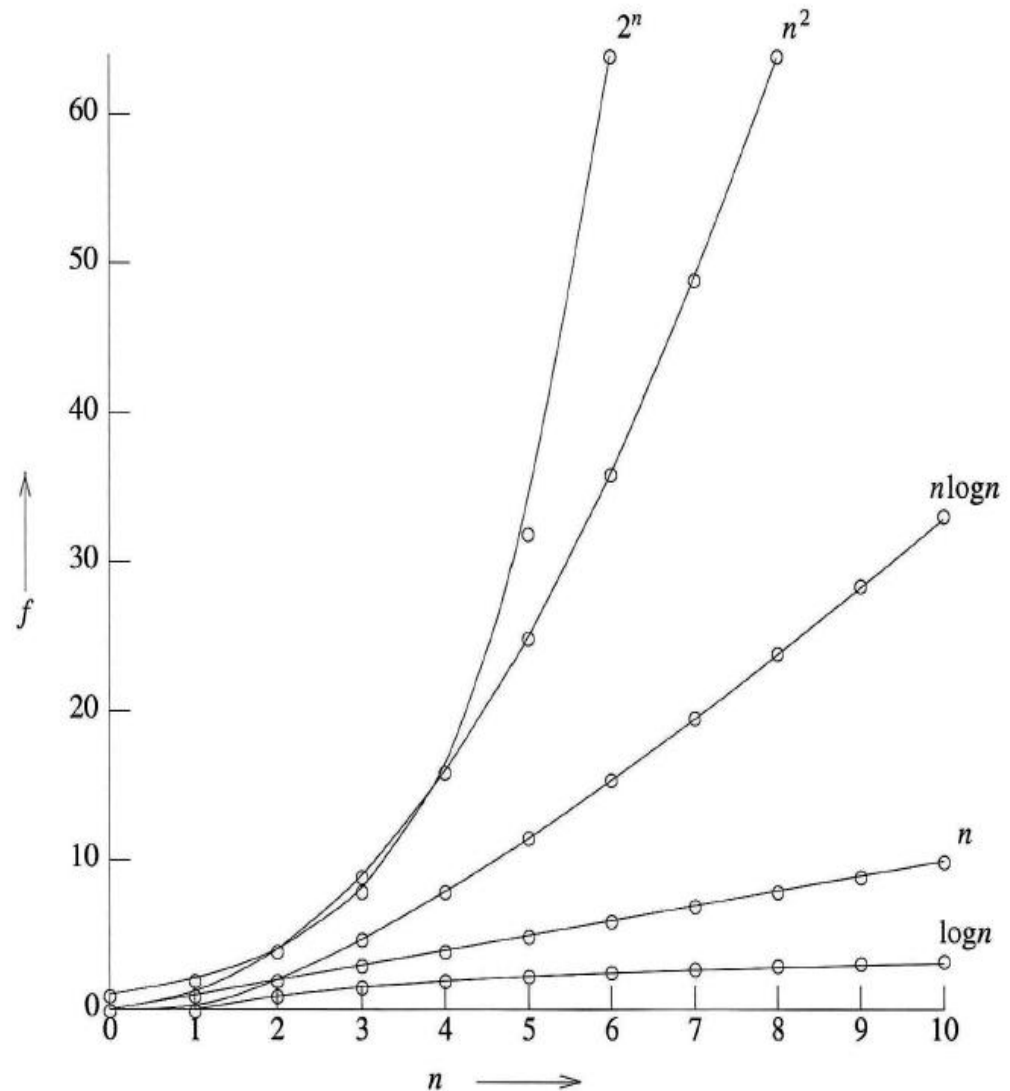


Figure 1.8 Plot of function values

	$f(n)$						
n	n	$n \log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	.01 μ s	.03 μ s	.1 μ s	1 μ s	10 μ s	10 s	1 μ s
20	.02 μ s	.09 μ s	.4 μ s	8 μ s	160 μ s	2.84 h	1 ms
30	.03 μ	.15 μ	.9 μ	27 μ	810 μ	6.83 d	1 s
40	.04 μ s	.21 μ s	1.6 μ s	64 μ s	2.56 ms	121 d	18 m
50	.05 μ s	.28 μ s	2.5 μ s	125 μ s	6.25 ms	3.1 y	13 d
100	.10 μ s	.66 μ s	10 μ s	1 ms	100 ms	3171 y	4×10^{13} y
10^3	1 μ s	9.96 μ s	1 ms	1 s	16.67 m	3.17×10^{13} y	32×10^{283} y
10^4	10 μ s	130 μ s	100 ms	16.67 m	115.7 d	3.17×10^{23} y	
10^5	100 μ s	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	

μ s = 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

1 billion = 10 억 = 1000000000 = 10^9 .

1.6 Performance Measurement

1.6.1 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**. We will concentrate our discussion on **measuring time**.

The functions we need to **time events** are part of C's standard library, and are accessed through the statement: **#include <time.h>**. There are actually two different methods for timing events in C. Figure 1.10 shows the major differences between these two methods.

	Method 1	Method 2
Start timing	<code>start = clock();</code>	<code>start = time(NULL);</code>
Stop timing	<code>stop = clock();</code>	<code>stop = time(NULL);</code>
Type returned	<code>clock_t</code>	<code>time_t</code>
Result in seconds	<code>duration = ((double) (stop - start)) / CLOCKS_PER_SEC;</code>	<code>duration = (double) difftime(stop, start);</code>

Data to use for measurement

- **worst-case data**
- **best-case data**
- **average-case data**

Figure 1.10: Event timing in C

Method 1 uses *clock* to time events. This function gives **the amount of processor time that has elapsed since the program began running**. To time an event we use *clock* twice, once at the start of the event and once at the end. The time is returned as a built-in type, *clock_t*. The total time required by an event is its start time subtracted from its stop time. Since this result could be any legitimate numeric type, we type cast it to **double**. In addition, since this result is measured as **internal processor time**, we must divide it by **the number of clock ticks per second** to obtain the result in seconds. In ANSI C, the ticks per second is held in the built-in constant, *CLOCKS_PER_SEC*. We found that this method was far more accurate on our machine. However, the second method does not require a knowledge of the ticks per second, which is why we also present it here.

Method 2 uses *time*. This function returns **the time, measured in seconds**, as the built-in type *time_t*. Unlike *clock*, *time* has one parameter, which specifies a location to hold the time. Since we do not want to keep the time, we pass in a NULL value for this parameter. As was true of Method 1, we use *time* at the start and the end of the event we want to time. We then pass these two times into *difftime*, which returns the difference between two times measured in **seconds**. Since the type of this result is *time_t*, we type cast it to **double** before printing it out.

```

#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

Event timing in C : more accurate timing

```
#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)    {
        long repetitions = 0;
        clock_t start= clock( );
        do {
            repetitions++;
            for (i = 0; i < n; i++) a[i] = n - i;        /* initialize with worst-case data */
            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

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)

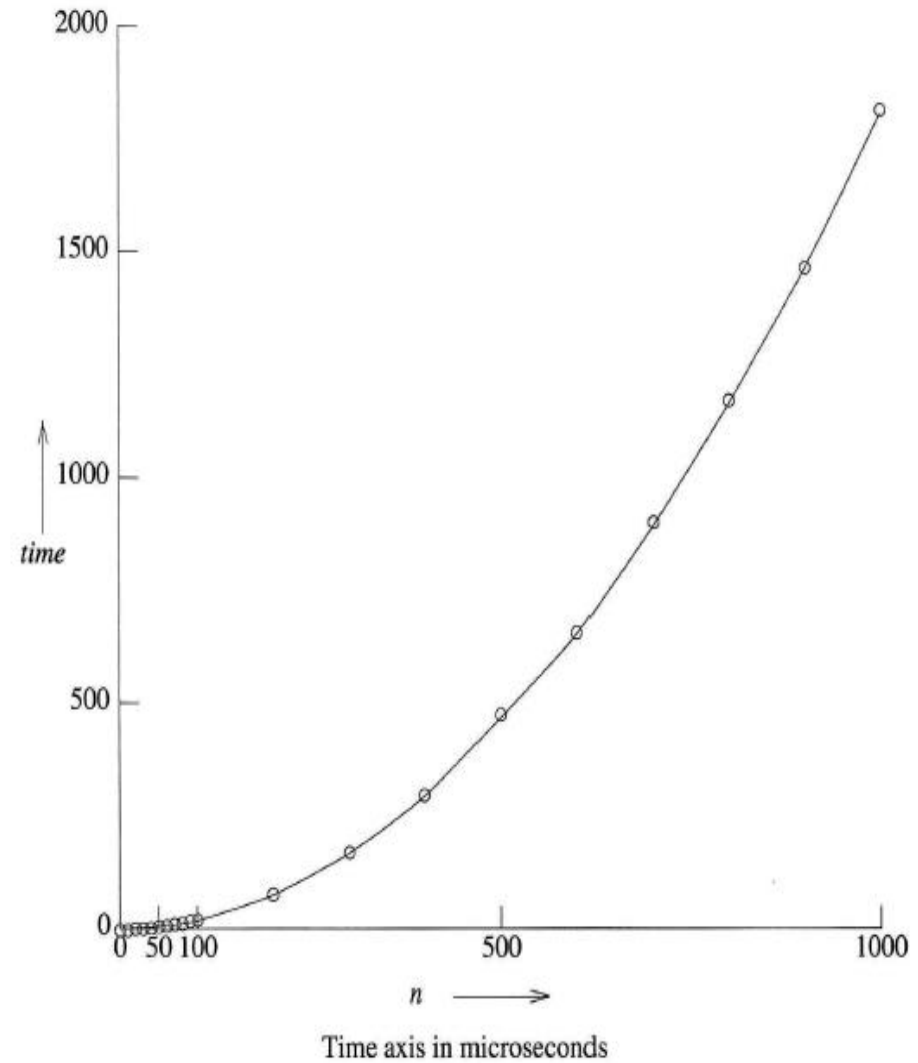


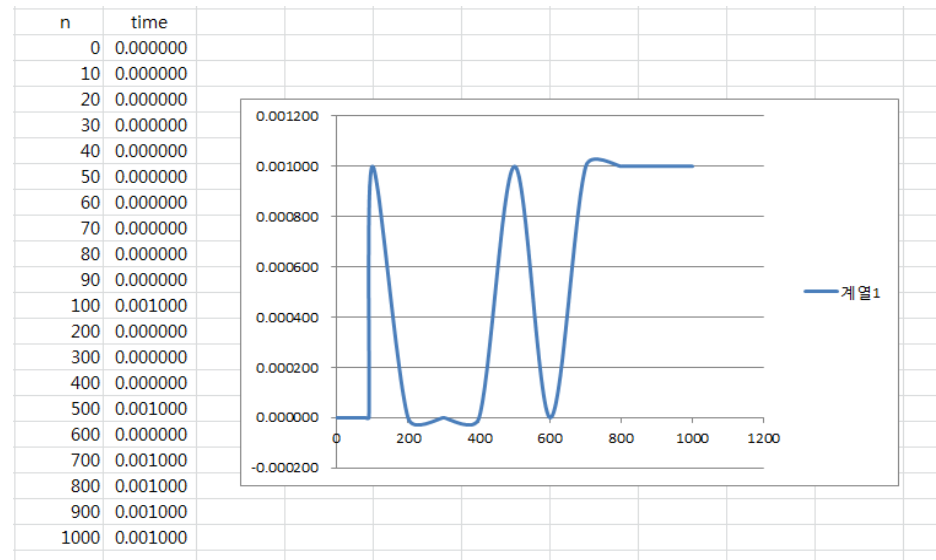
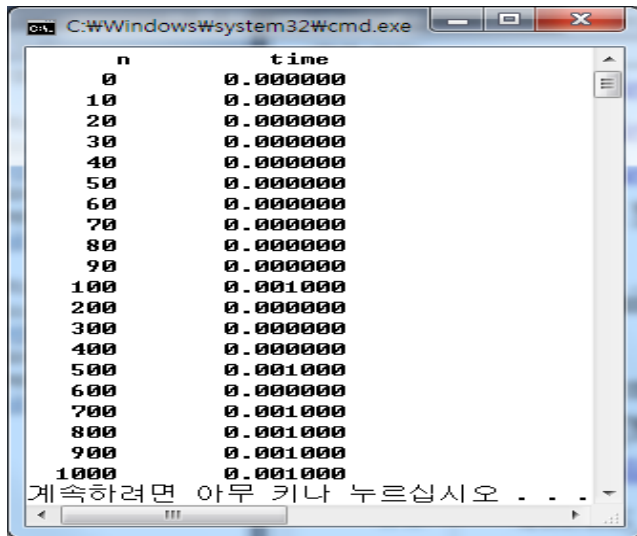
Figure 1.12: Graph of worst-case performance of selection sort

실습 2: 성능측정(Performance Measurement)

1. Program 1.24를 실행하여 각자의 시스템에 대한 실제 성능측정을 하여라. 프로그램의 실행결과를 이용하여 Figure 1.11 ~ 12와 같은 표와 그래프로 나타내어라. (2점)

2. Program 1.25를 실행하여 출력결과로부터 표와 그래프를 작성하라. (2점)

실행결과에 대한 표와 그래프 작성 예
1번.

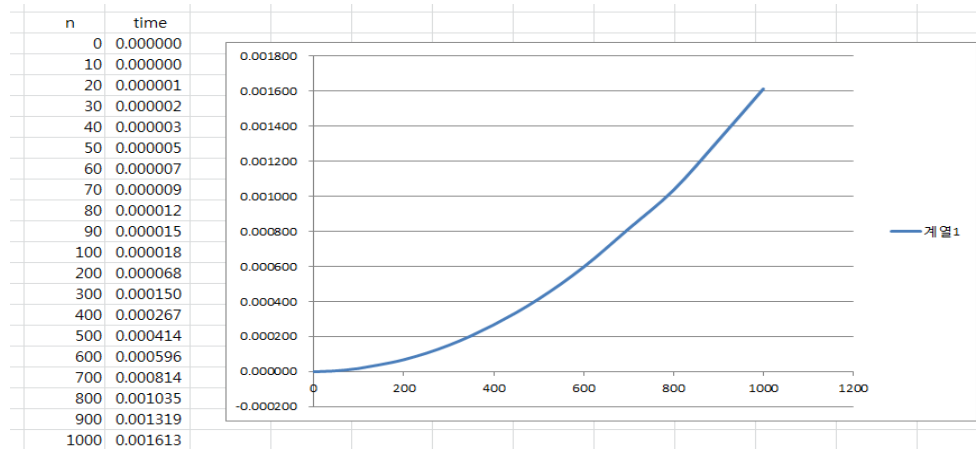


실습 2: 성능측정(Performance Measurement)

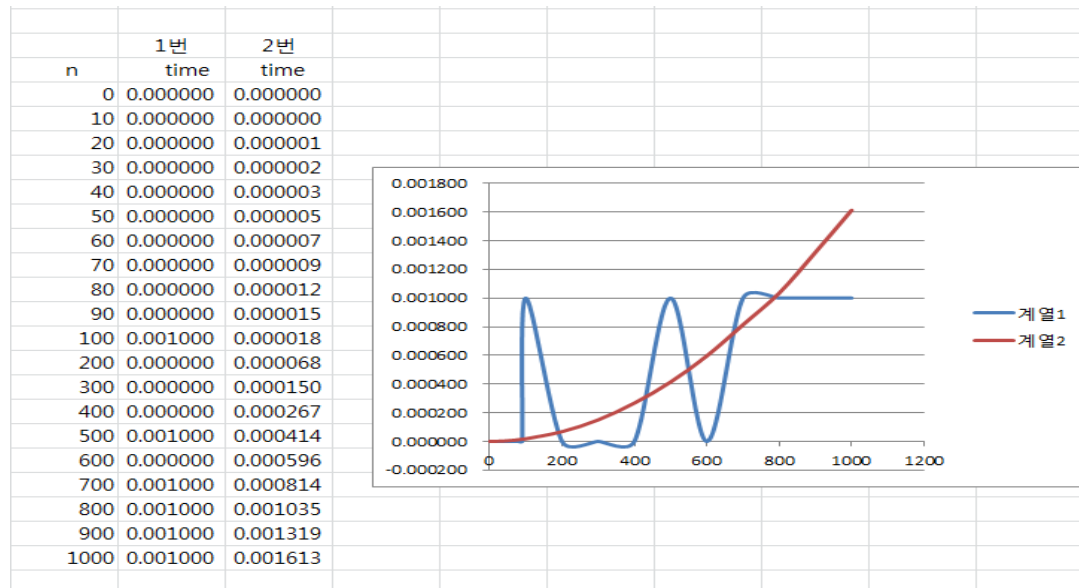
2번.

n	repetitions	time
0	16876035	0.000000
10	626215	0.000003
20	680128	0.000001
30	433536	0.000002
40	288027	0.000003
50	203398	0.000005
60	144730	0.000007
70	108047	0.000009
80	85275	0.000012
90	68952	0.000015
100	56022	0.000018
200	14658	0.000068
300	6614	0.000151
400	3774	0.000265
500	2443	0.000409
600	1706	0.000586
700	1251	0.000799
800	959	0.001043
900	761	0.001314
1000	615	0.001628

계속하려면 아무 키나 누르십시오 . . .



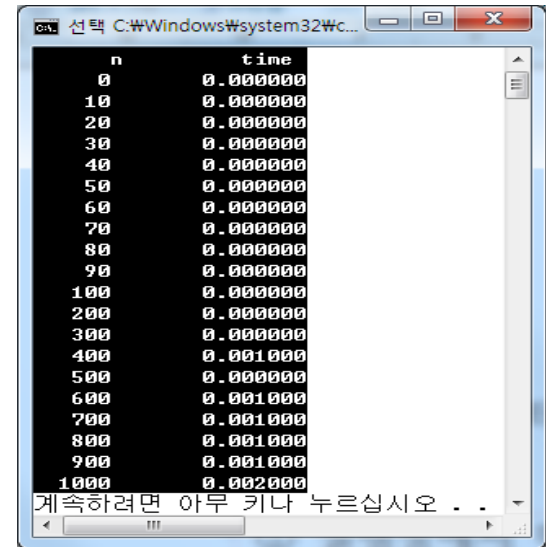
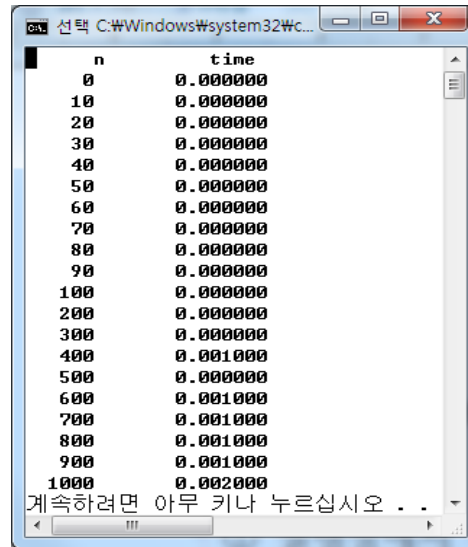
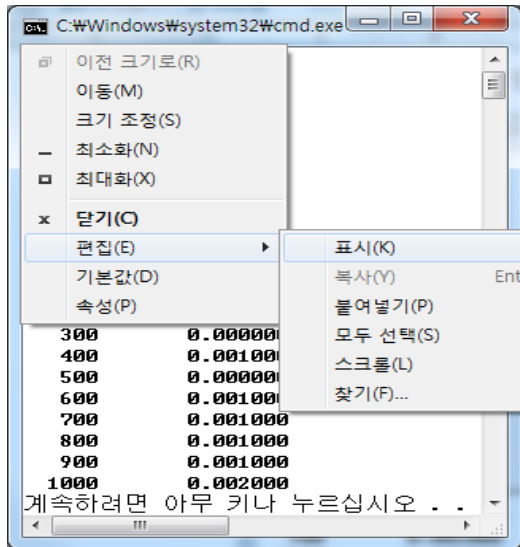
1번 + 2번.



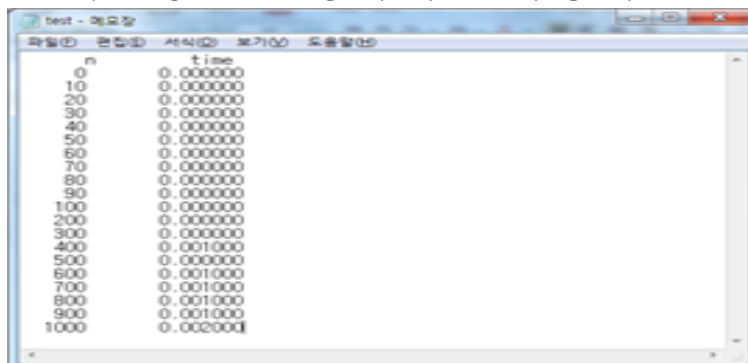
실습 2: 성능측정(Performance Measurement)

■ 실행결과로부터 표와 그래프 만들기

- ① 실행창에서 메뉴 선택 ② 마우스로 클릭&드래그 ③ 블록지정 후 엔터키

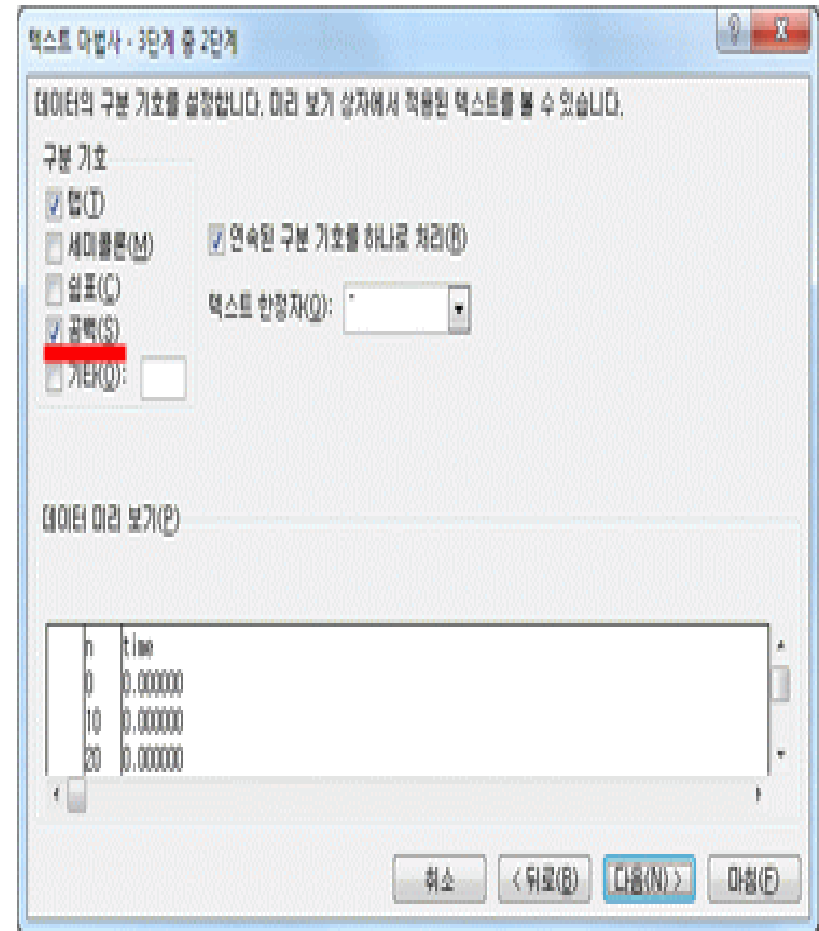
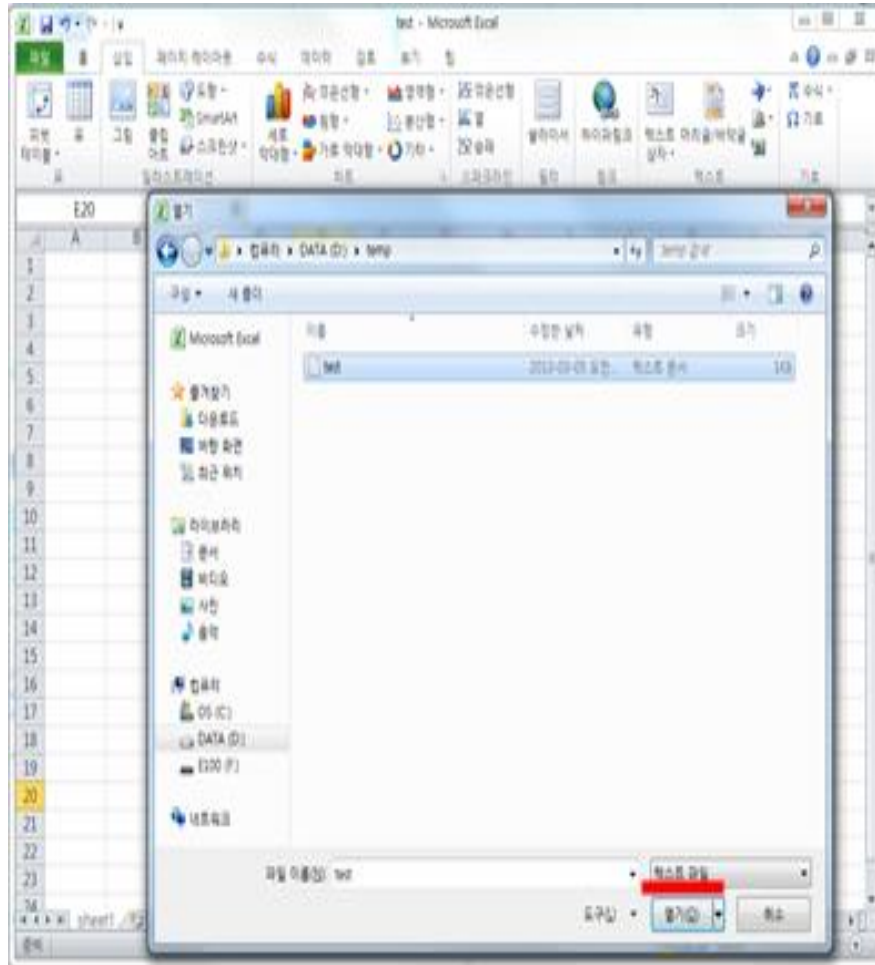


- ④ 메모장을 실행하여 붙여넣기를 한 후 저장하기



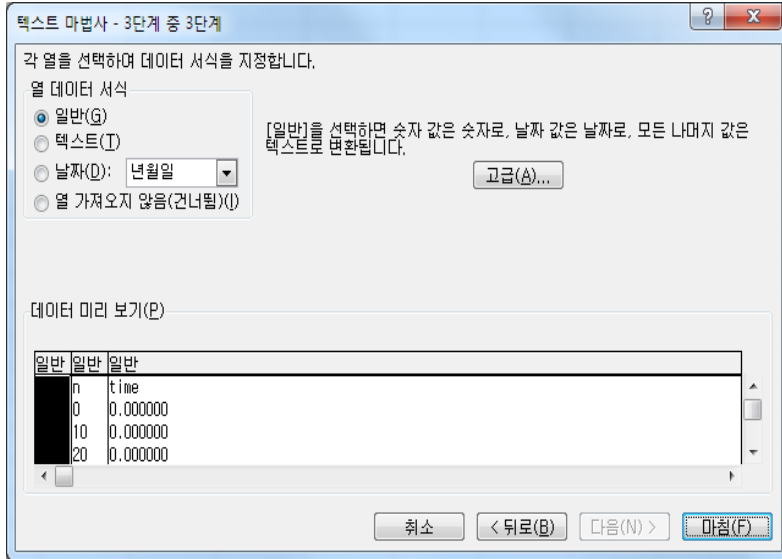
실습 2: 성능측정(Performance Measurement)

⑤ 엑셀을 실행한 후 ④에서 저장한 파일 열기

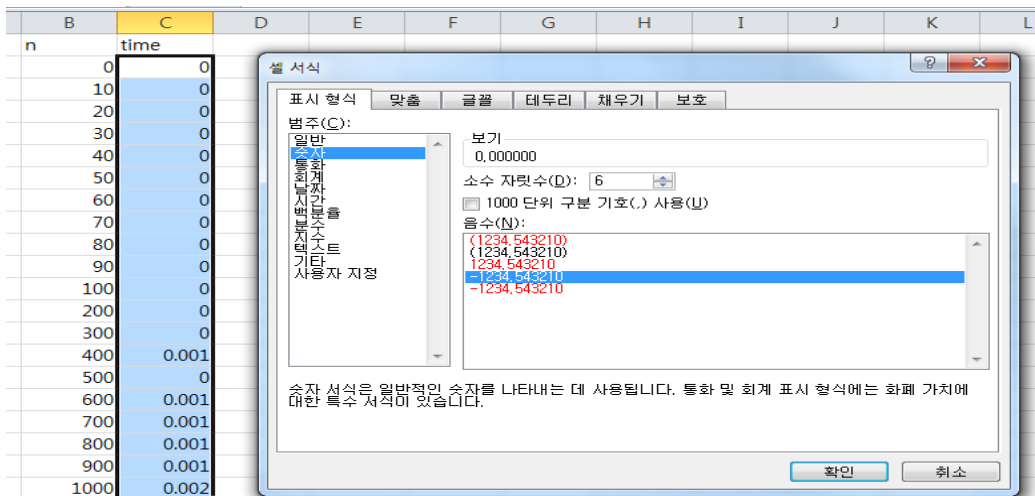


실습 2: 성능측정(Performance Measurement)

⑤ 엑셀을 실행한 후 ④에서 저장한 파일 열기

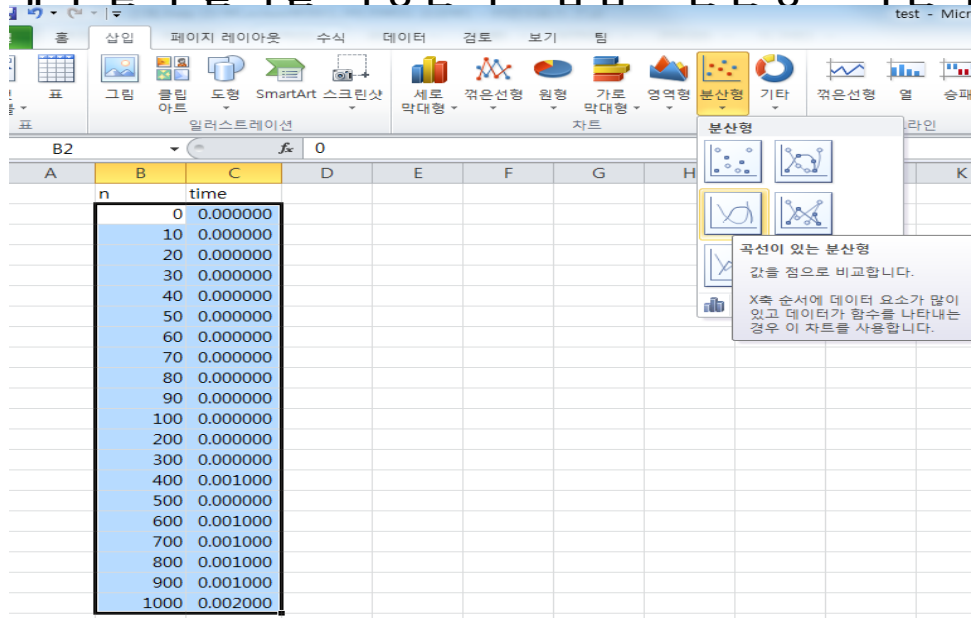


A	B	C	D
	n	time	
	0	0	
	10	0	
	20	0	
	30	0	
	40	0	
	50	0	
	60	0	
	70	0	
	80	0	
	90	0	
	100	0	
	200	0	
	300	0	
	400	0.001	
	500	0	
	600	0.001	
	700	0.001	
	800	0.001	
	900	0.001	
	1000	0.002	

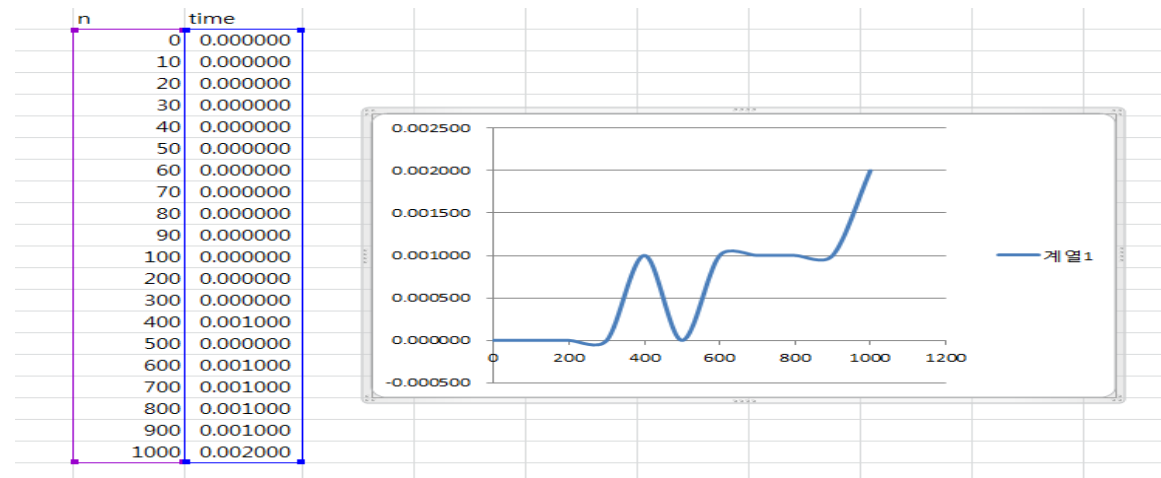


실습 2: 성능측정(Performance Measurement)

⑥ 아래와 같이 블록을 지정한 후 “삽입 > 분산형 > 곡선이 있는 분산형” 아이콘을 선택하기

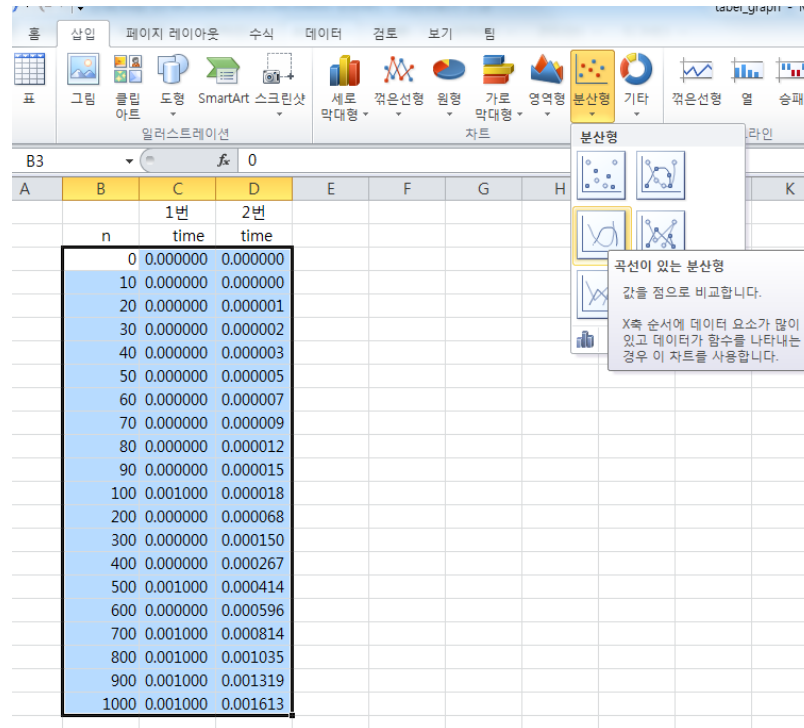


⑦ 그래프 생성 결과



실습 2: 성능측정(Performance Measurement)

※ 그래프를 겹쳐서 그리는 방법



The End