

CHAP 1:자료구조와 알고리즘



Contents

■ 학습목표

- 자료구조의 의미와 중요성을 알아본다.
- 자료구조에서 다루는 내용을 알아본다.
- 컴퓨터 내부의 2진수 코드 체계를 알아본다.
- 자료 형태에 따른 자료 표현 형식을 알아본다.
- 자료를 추상화하고 구체화하는 개념을 이해한다.
- 알고리즘의 개념을 이해하고 알고리즘의 표현 방법을 알아본다.
- 알고리즘의 성능 분석 방법을 알아본다.

■ 내용

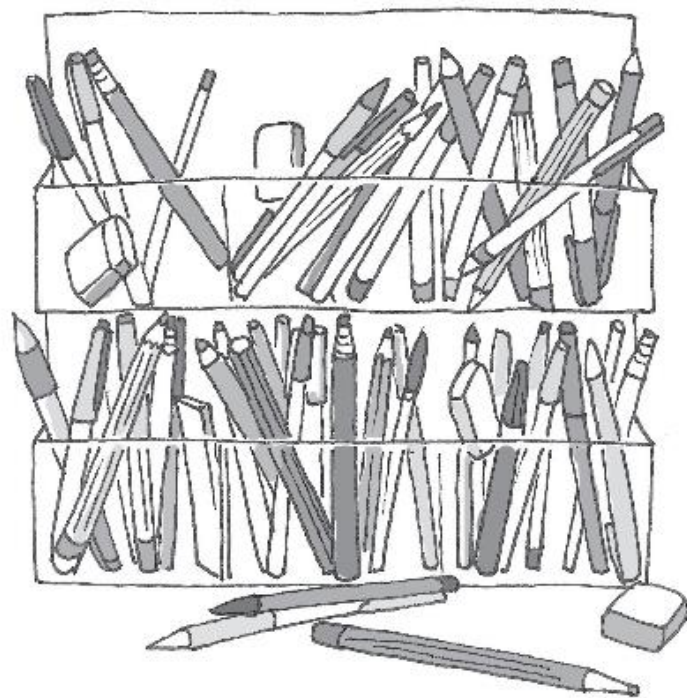
- | | |
|---------------|------------------|
| ■ 01 자료구조의 이해 | ■ 04 알고리즘의 이해 |
| ■ 02 자료의 표현 | ■ 05 알고리즘의 표현 방법 |
| ■ 03 자료의 추상화 | ■ 06 알고리즘의 성능 분석 |

자료구조와 알고리즘의 이해

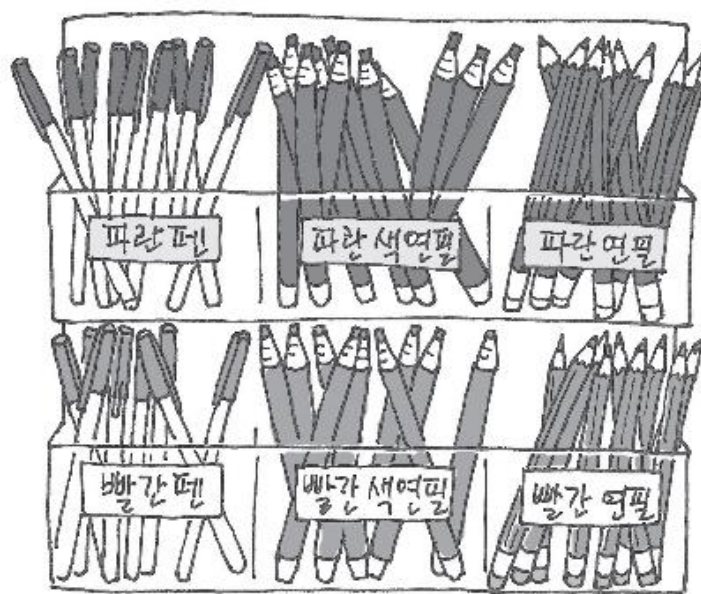
- C언어를 알고 있는가?
 - 구조체를 정의할 줄 안다.
 - 메모리의 동적 할당과 관련하여 이해한다.
 - 포인터와 관련하여 이해와 활용에 부담이 없다.
 - 헤더파일의 정의하고 활용할 줄 안다.
 - 각종 매크로를 이해하고 `#ifndef` ~ `#endif`의 의미를 안다.
 - 다수의 소스파일, 헤더파일로 구성된 프로그램 작성 가능!
 - 재귀함수에 어느 정도 익숙하다.

자료구조의 개념

- 자료구조의 개념
 - 자료를 효율적으로 표현하고 저장하고 처리할 수 있도록 정리하는 것



(a) 자료구조를 적용하기 전



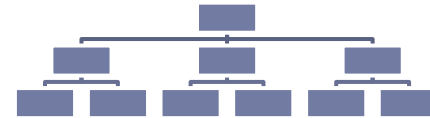
(b) 자료구조를 적용한 후

그림 1-1 생활 속에서 자료구조를 적용한 예

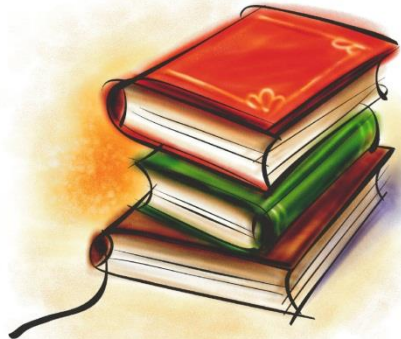
일상생활에서의 사물의 조직화



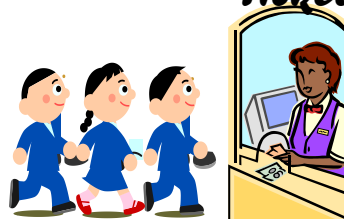
조직도



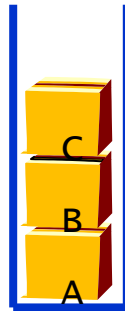
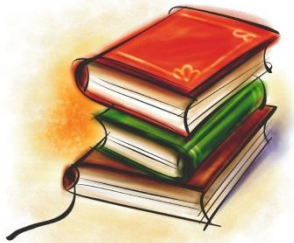
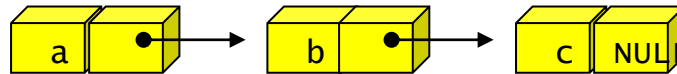
일상생활에서의
사물의 조직화



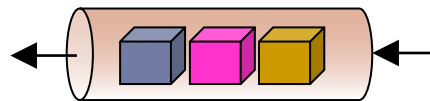
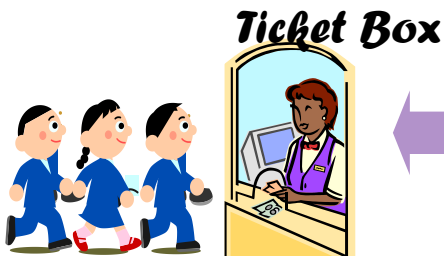
Ticket Box



일상생활과 자료구조의 비교



일상생활에서의 예	자료구조
물건을 쌓아두는 것	스택
영화관 대표소의 줄	큐
할일 리스트	리스트
영어사전	사전, 탐색구조
지도	그래프
조직도	트리



전단(front)

후단(rear)

자료구조란 무엇인가?

자료구조

“프로그램이란 데이터를 표현 하고,

표현에는 저장의 의미가 포함된다!

그렇게 표현된 데이터를 처리 하는 것이다.”

알고리즘



10을 표현하자??

자료구조의 분류

자료구조란 무엇인가?

- 컴퓨터 분야에서 자료구조를 왜 배워야 하는가?
 - 컴퓨터가 효율적으로 문제를 처리하기 위해서는 문제를 정의하고 분석하여 그에 대한 최적의 프로그램을 작성해야 한다.
 - 자료구조에 대한 개념과 활용 능력 필요!

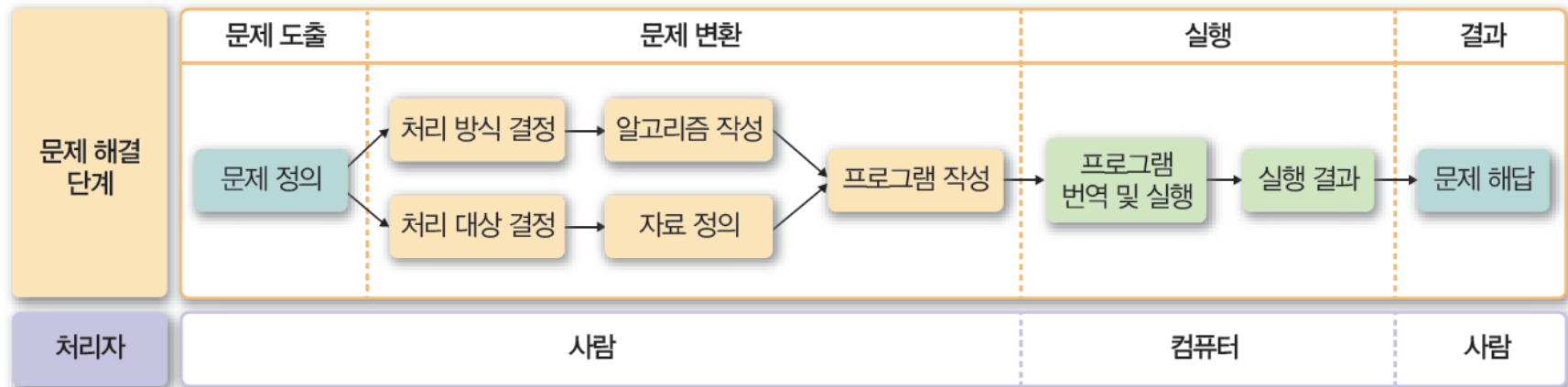


그림 1-2 문제 해결 과정

자료구조와 알고리즘

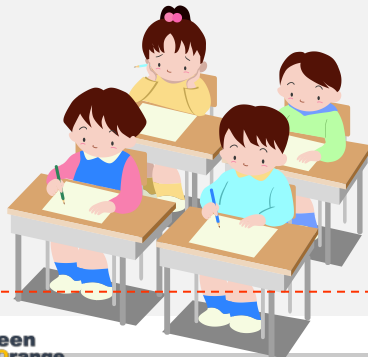
- 프로그램 = 자료구조 + 알고리즘
(예) 최대값 탐색 프로그램 = 배열 + 순차탐색

자료구조

알고리즘

score[]

80	70	90	...	30
----	----	----	-----	----



```
tmp ← score[0];  
for i ← 1 to n do  
    if score[i] > tmp  
        then tmp ← score[i];
```

자료구조와 알고리즘

```
int main(void)
```

```
{
```

```
// 배열의 선언
```

```
int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
...
```

```
// 배열에 저장된 값의 합
```

```
for(idx=0; idx<10; idx++)
```

```
    sum += arr[idx];
```

```
...
```

```
}
```

자료구조

알고리즘

알고리즘은 자료구조에 의존적이다!

자료구조의 분류

- 자료의 형태에 따른 분류
 - 단순 구조
 - 정수, 실수, 문자, 문자열, 등의 기본 자료형
 - 선형 구조
 - 자료들 사이의 관계가 1:1 관계
 - 순차 리스트, 연결 리스트, 스택, 큐, 데크 등
 - 비선형 구조
 - 자료들 사이의 관계가 1:다, 또는 다:다 관계
 - 트리, 그래프 등
 - 파일 구조
 - 서로 관련 있는 필드로 구성된 레코드의 집합인 파일에 대한 구조
 - 순차 파일, 색인 파일, 직접 파일 등

자료구조의 분류

■ 자료의 형태에 따른 분류

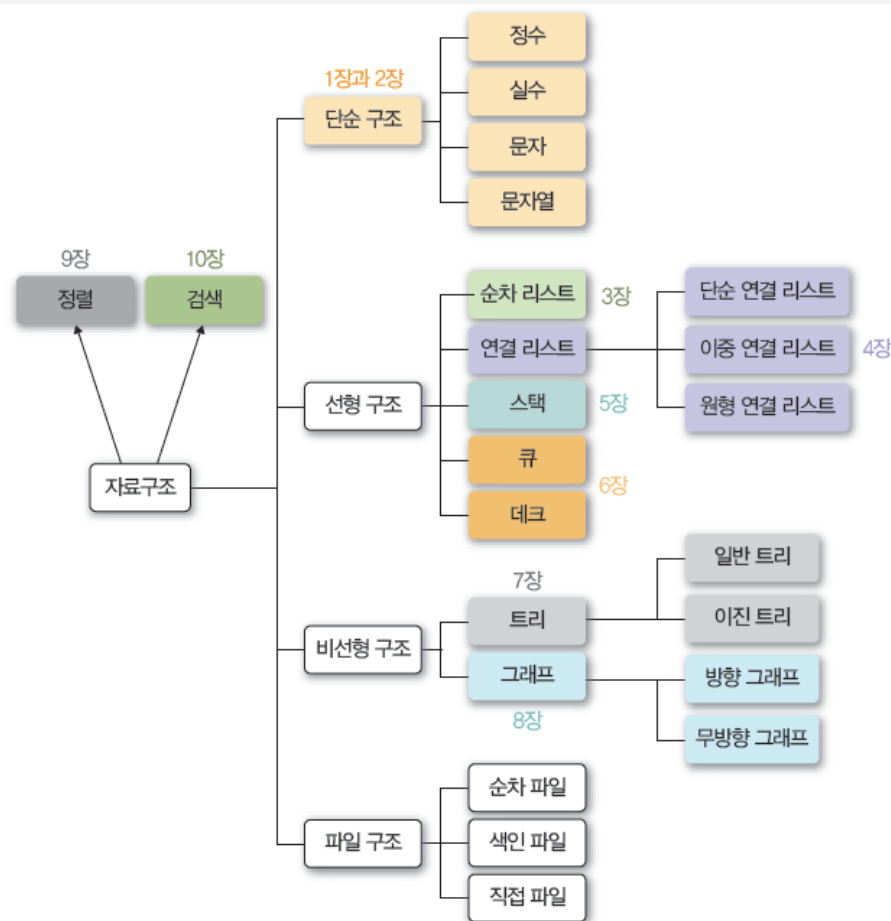


그림 1-4 자료구조의 형태에 따른 분류와 이 책에서 다루는 세부 주제

자료의 표현

- 컴퓨터에서의 자료 표현
 - 숫자, 문자, 그림, 소리, 기호 등 모든 형식의 자료를 2진수 코드로 표현하여 저장 및 처리
 - 2진수 코드란?
 - 1과 0, On과 Off, 참^{True}과 거짓^{False}의 조합
 - 2진수 코드의 단위

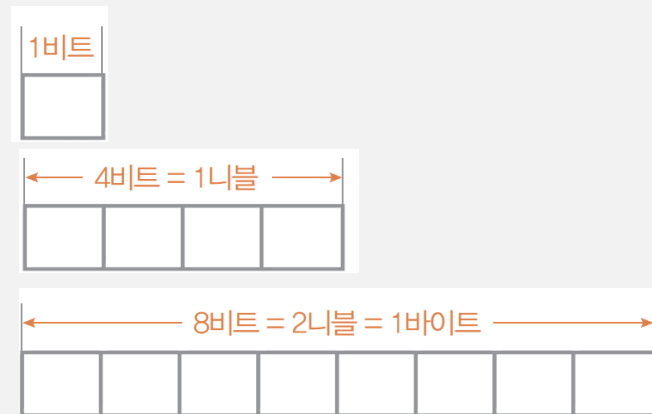
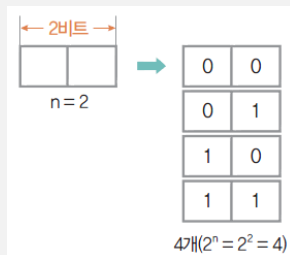


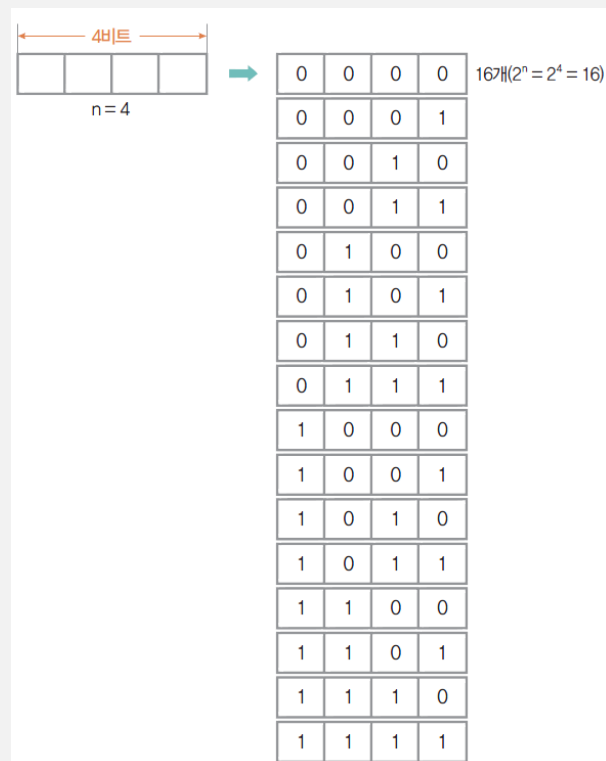
그림 1-5 컴퓨터의 자료 표현 : 비트, 니블, 바이트

자료의 표현

- 디지털 시스템에서의 자료 표현
 - n 개의 비트로 2^n 개의 상태 표현 가능



(a) $n=2$ 인 경우 : 2^2 개의 상태 표현



(b) $n=4$ 인 경우 : 2^4 개의 상태 표현

그림 1-6 자료 표현 예 : n 개의 비트로 2^n 개의 상태 표현

자료의 표현

■ 컴퓨터 내부에서 표현할 수 있는 자료의 종류

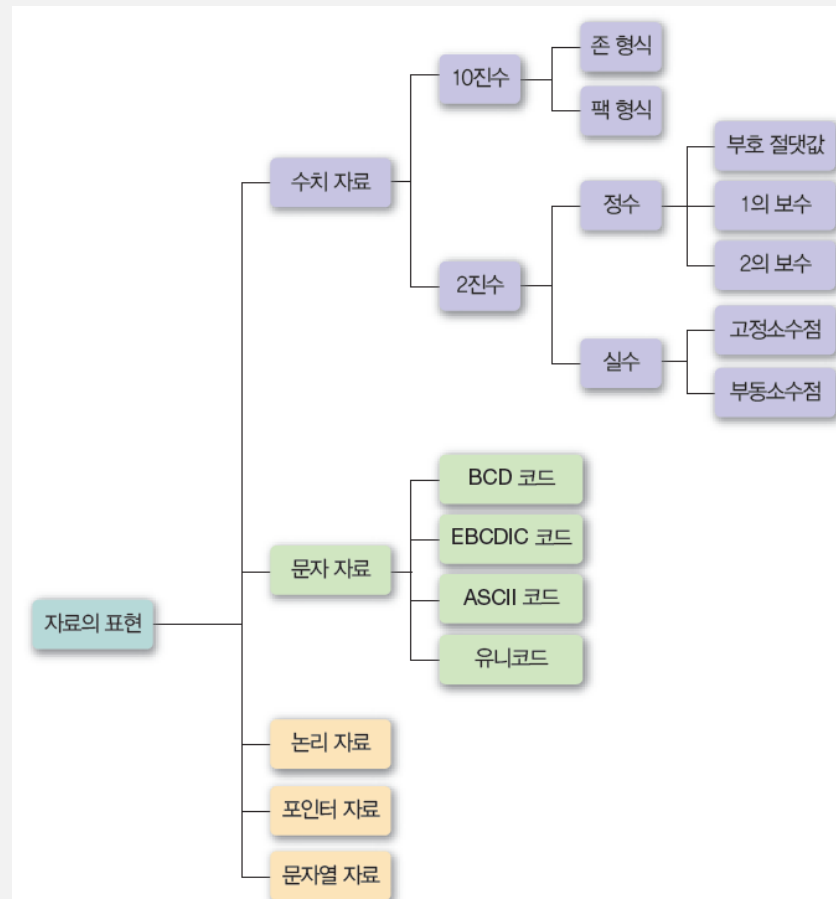


그림 1-7 컴퓨터 내부에서 자료를 표현하는 방법

자료의 표현 : 수치 자료의 표현

■ 10진수의 표현

■ 존Zone 형식의 표현

- 10진수 한 자리를 표현하기 위해서 1바이트(8비트)를 사용하는 형식
- 존 영역
 - 상위 4비트
 - 1111로 표현
- 수치 영역
 - 하위 4비트
 - 표현하고자 하는 10진수 한 자리 값에 대한 2진수 값을 표시

■ 존 형식의 구조

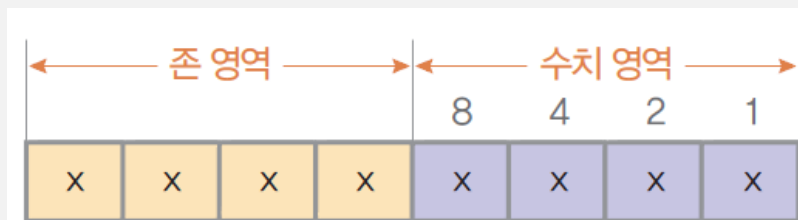


그림 1-8 존 형식의 구조

자료의 표현 : 수치 자료의 표현

- 10진수의 표현
 - 수치 영역의 값 표현 : [표 1-1]

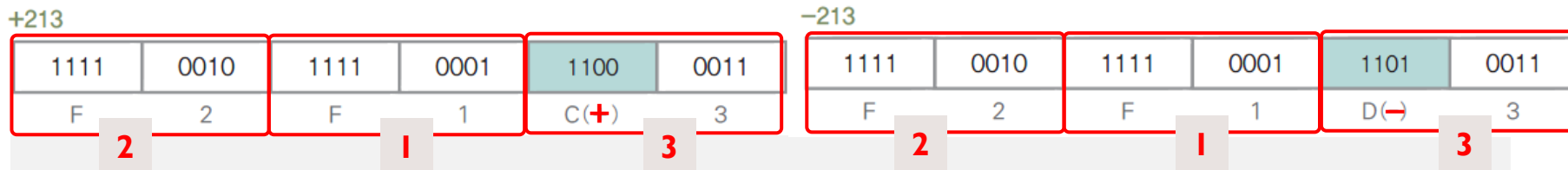
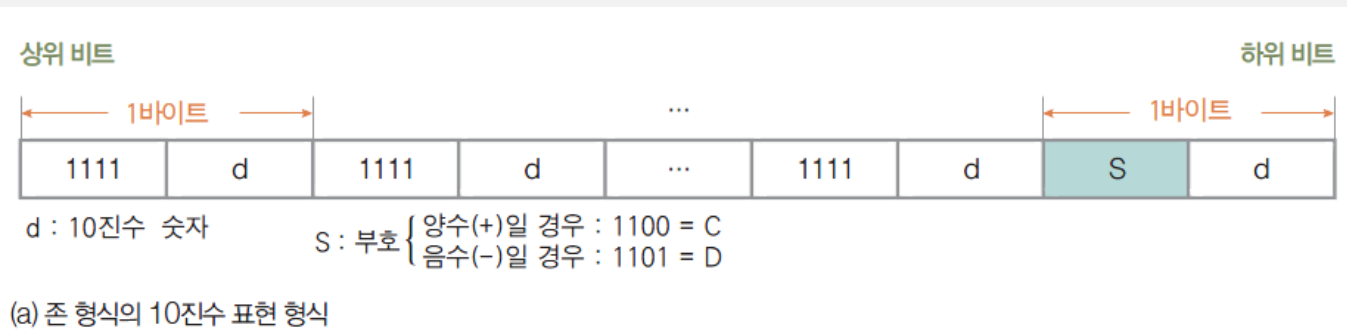
표 1-1 4비트의 2진수에 대한 10진수 표현

4비트의 2진수				10진수 변환	10진수
0	0	0	0	$0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$	0
0	0	0	1	$0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$	1
0	0	1	0	$0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$	2
0	0	1	1	$0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$	3
0	1	0	0	$0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$	4
0	1	0	1	$0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$	5
0	1	1	0	$0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$	6
0	1	1	1	$0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$	7
1	0	0	0	$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$	8
1	0	0	1	$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$	9
1	0	1	0	$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$	10 = A
1	0	1	1	$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$	11 = B
1	1	0	0	$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$	12 = C
1	1	0	1	$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$	13 = D
1	1	1	0	$1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$	14 = E
1	1	1	1	$1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$	15 = F

자료의 표현 : 수치 자료의 표현

■ 10진수의 표현

- 여러 자리의 10진수를 표현하는 방법
 - 10진수의 자릿수만큼 존 형식을 연결하여 사용
 - 마지막 자리의 존 영역에 부호를 표시
 - 양수(+) : 1100
 - 음수(-) : 1101

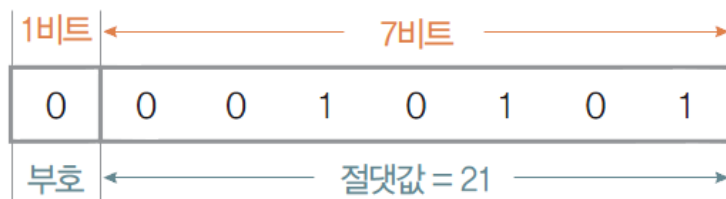


(b) 존 형식의 10진수 표현 예

자료의 표현 : 수치 자료의 표현

- 2진수의 정수 표현
 - n비트의 부호 절댓값 형식
 - 최상위 1비트 : 부호 표시
 - 양수(+) : 0
 - 음수(-) : 1
 - 나머지 n-1 비트 : 이진수 표시
 - 1바이트를 사용하는 부호 절댓값 표현 예

+21



-21



그림 1-11 +21과 -21의 부호와 절댓값 표현 예

자료의 표현 : 수치 자료의 표현

■ 2진수의 정수 표현

■ 1의 보수¹⁾ Complement 형식

- 음수 표현에서 부호 비트를 사용하는 대신 1의 보수를 사용하는 방법
- n비트의 2진수를 1의 보수로 만드는 방법
 - n비트를 모두 1로 만든 이진수에서 변환하고자 하는 이진수를 뺌
 - 예) 10진수 21을 1의 보수로 만듦(1바이트 사용)

$$(2^8 - 1) - |-21| = (2^8 - 1) - 21$$

$$= (1\ 0000\ 0000 - 0000\ 0001) - 0001\ 0101$$

$$= (1111\ 1111) - 0001\ 0101$$

$$= 1110\ 1010$$

$$\begin{array}{r} 11111111 \\ -00010101 \\ \hline 11101010 \end{array} \quad \begin{array}{l} \leftarrow -21\text{의 절댓값} \\ \leftarrow 21\text{의 1의 보수} = -21 \end{array}$$

그림 1-12 1의 보수 형식 표현 과정

- 1바이트를 사용하는 1의 보수 표현 예

+21

0	0	0	1	0	1	0	1
부호	← 21의 절댓값 →						

-21

1	1	1	0	1	0	1	1
부호	← 21의 2의 보수						

그림 1-15 +21과 -21의 2의 보수 형식 표현 예

자료의 표현 : 수치 자료의 표현

■ 2진수의 정수 표현

■ 2의 보수^{2'} Complement 형식

- 음수의 표현에서 부호 비트를 사용하는 대신 2의 보수를 사용하는 방법
- n비트의 2진수를 2의 보수로 만드는 방법
 - 1의 보수에 1을 더함
 - 예) 10진수 21을 2의 보수로 만들기(1바이트 사용)

$$\begin{aligned} 2^8 - |-21| &= 2^8 - 21 \\ &= 1\ 0000\ 0000 - 0001\ 0101 \\ &= 1110\ 1011 \end{aligned}$$

$$\begin{array}{r} 11111111 \\ - 00010101 \quad \leftarrow -21\text{의 절댓값} \\ \hline 11101010 \quad \leftarrow 21\text{의 1의 보수} \\ + \quad \quad \quad 1 \\ \hline 11101011 \quad \leftarrow 21\text{의 2의 보수} = -21 \end{array}$$

그림 1-14 2의 보수 형식 표현 과정

자료의 표현 : 수치 자료의 표현

- 2진수의 정수 표현
 - 2의 보수^{2'} Complement 형식
 - 1바이트를 사용하는 2진 보수 형식의 예



- 2진수 정수의 세 가지 표현 방법에서 양수의 표현은 같고 **음수의 표현만 다르다.**

자료의 표현 : 수치 자료의 표현

표 1-3 2진수를 표현하는 세 가지 방법 비교

표현 방법	특징
부호와 절댓값 형식	<ul style="list-style-type: none"> • MSB값을 바꿔 음수를 간단히 표현할 수 있다. • 가산기와 감산기가 모두 필요하므로 하드웨어 구성 비용이 많이 든다. • +0(00000000)과 -0(10000000)이 존재하므로 논리적으로 맞지 않다. • n비트로 $-(2^{n-1}-1) \sim +(2^{n-1}-1)$의 범위를 표현할 수 있다.
1의 보수 형식	<ul style="list-style-type: none"> • $(A-B)$ 뺄셈을 $(A+(B \text{의 } 1 \text{의 보수}))$로 변환하여 계산할 수 있으므로, 가산기 회로로 감산을 수행할 수 있다. • +0(00000000)과 -0(11111111)이 존재하므로 논리적으로 맞지 않다. • n비트로 $-(2^{n-1}-1) \sim +(2^{n-1}-1)$의 범위를 표현할 수 있다.
2의 보수 형식	<ul style="list-style-type: none"> • $(A-B)$ 뺄셈을 $(A+(B \text{의 } 2 \text{의 보수}))$로 변환하여 계산할 수 있으므로 가산기 회로로 감산을 수행할 수 있다. • 덧셈 연산에서 발생하는 오버플로 처리가 1의 보수 형식보다 간단하다. • 컴퓨터 시스템에서 실제로 사용하는 형식이다. • n비트로 $-2^{n-1} \sim +(2^{n-1}-1)$의 범위를 표현할 수 있다.

자료의 표현 : 수치 자료의 표현

- 2진수의 실수 표현

- 고정 소수점 표현

- 소수점이 항상 최상위 비트의 왼쪽 밖에 고정되어 있는 것으로 취급하는 방법
 - 고정 소수점 표현의 **00010101**은 **0.00010101**의 실수 값을 의미

- 부동 소수점 형식의 표현

- 고정 소수점 형식에 비해서 표현 가능한 값의 범위가 넓음
 - 실수를 구분하여 표현

$$213 = 0.213 \times 10^3 \xrightarrow{\text{지수}}$$

↓ ↓
소수부 밑수 base, radix

그림 1-16 실수의 과학적 표기

자료의 표현 : 수치 자료의 표현

- 2진수의 실수 표현
 - 부동 소수점 형식의 표현



그림 1-17 부동소수점 표현 형식

자료의 표현 : 문자 자료의 표현

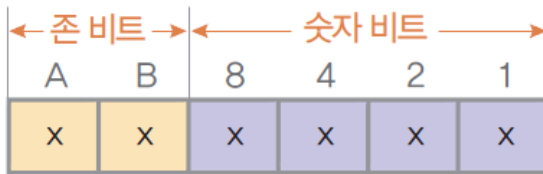
- 문자 자료의 표현
 - 문자에 대한 이진수 코드를 정의하여 사용
 - 문자에 대한 이진수 코드표
 - BCD 코드
 - EBCDIC 코드
 - ASCII 코드

자료의 표현 : 문자 자료의 표현

- BCD 코드

- 6비트를 사용하여 문자 표현

- 상위 2비트 : 존 비트
 - 하위 4비트 : 2진수 비트
 - 존 비트와 2진수 비트를 조합하여 10진수 0~9와 영어 대문자, 특수 문자를 표현



존 비트 AB의 값 {

- 00 : 숫자 0, 1~9(1010, 0001~1001)
- 01 : 문자 A~I(0001~1001)
- 10 : 문자 J~R(0001~1001)
- 11 : 문자 S~Z(0010~1001)

그림 1-19 BCD 코드의 구성

자료의 표현 : 문자 자료의 표현

■ BCD 코드

존 비트 01과 숫자 비트 0001을 연결한 01 0001이 A에 대한 BCD 코드

표 1-4 BCD 코드표

존 비트	숫자 비트	표현 문자	존 비트	숫자 비트	표현 문자	존 비트	숫자 비트	표현 문자	존 비트	숫자 비트	표현 문자
00	0001	1	01	0001	A	10	0001	J			
00	0010	2	01	0010	B	10	0010	K	11	0010	S
00	0011	3	01	0011	C	10	0011	L	11	0011	T
00	0100	4	01	0100	D	10	0100	M	11	0100	U
00	0101	5	01	0101	E	10	0101	N	11	0101	V
00	0110	6	01	0110	F	10	0110	O	11	0110	W
00	0111	7	01	0111	G	10	0111	P	11	0111	X
00	1000	8	01	1000	H	10	1000	Q	11	1000	Y
00	1001	9	01	1001	I	10	1001	R	11	1001	Z
00	1010	0									

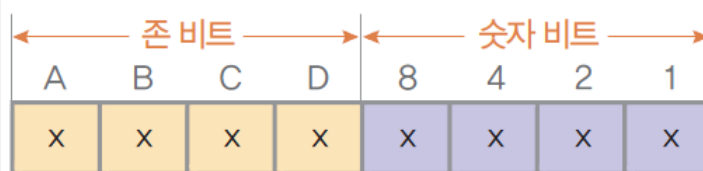
자료의 표현 : 문자 자료의 표현

■ EBCDIC 코드

■ 8비트를 사용하여 문자 표현

- 상위 4비트 : 존 비트
- 하위 4비트 : 2진수 비트
- 존 비트와 2진수 비트를 조합하여 10진수 0~9와 영어 대문자/소문자와 특수문자를 표현

■ EBCDIC 코드의 구성



존 비트 AB의 값 {
00 : 여분
01 : 특수문자
10 : 영어 소문자
11 : 영어 대문자

존 비트 CD의 값 {
00 : 문자 A~I(0001~1001)
01 : 문자 J~R(0001~1001)
10 : 문자 S~Z(0010~1001)
11 : 숫자 0~9(0000~1001)

그림 1-20 EBCDIC 코드의 구성

자료의 표현 : 문자 자료의 표현

■ EBCDIC 코드

A가 있는 자리의 알(존 비트) '1100'과 맹(숫자 비트) '0001'을 연결한 '11000001'이 A에 대한 EBCDIC 코드

표 1-5 EBCDIC 코드 표

상위 하위	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	NUL	DLE	DS		SP	&	-						{	}	W(V)	0
0001	SOH	DC1	SOS				/		a	j	~		A	J		1
0010	STX	DC2	FS	SYN					b	k	s		B	K	S	2
0011	ETX	TM							c	l	t		C	L	T	3
0100	PF	RES	BYP	PN					d	m	u		D	M	U	4
0101	HT	NL	LF	RS					e	n	v		E	N	V	5
0110	LC	BS	ETB	UC					f	o	w		F	O	W	6
0111	DEL	IL	ESC	EOT					g	p	x		G	P	X	7
1000	GE	CAN							h	q	y		H	Q	Y	8
1001	RLF	EM							i	r	z		I	R	Z	9
1010	SMM	CC	SM		¢	!		:								
1011	VT	CU1	CU2	CU3	.	\$.	#								
1100	FF	IFS		DC4	<	*	%	@								
1101	CR	IGS	ENQ	NAK	()	-	'								
1110	SO	IRS	ACK		+	:	>	=								
1111	SI	IUS	BEL	SUB		└	?	"								

- EOT End Of Transmission
- ESC ESCape
- ETB End of Transmission Block
- ETX End of Text
- FF From Feed
- FS Field Separator
- HT Horizontal Tab
- IFS Interchange File Separator
- PN Punch on

- RES REStore
- RS Reader Stop
- SI Shift In
- SM Set Mode
- SMM Start of Manual Message
- SO Shift Out
- SOH Start Of Heading
- IGS Interchange Group Separator
- IL Idle

- ACK ACKnowledge
- BEL BELl
- BS BackSpace
- BYP BYPass
- CAN CAnceL
- CC Cursor Control
- CR Carriage Return
- CU1 Customer Use 1
- CU3 Customer Use 3
- IRS Interchange Record Separator
- IUS Interchange Unit Separator
- LC Lower Case
- LF Line Feed
- NAK Negative ACKnowledge
- NL New Line
- NUL NULl
- PF Punch ofF
- SOS Start Of Significance

- DC1 Device Control1
- DC2 Device Control2
- DC4 Device Control4
- DEL DELeTe
- DLE Data Link Escape
- CU2 Customer Use 2
- DS Digital Select
- EM End of Medium
- ENQ ENQuire
- SP SPace
- STX Start of Text
- SUB SUBStitute
- SYN SYNchronous
- TM Tape Mark
- UC Upper Case
- VT Vertical Tab
- ¢ Cent Sign
- └ Logical NOT

자료의 표현 : 문자 자료의 표현

- ASCII 코드

- 7비트를 사용하여 문자 표현

- 상위 3비트 : 존 비트
 - 하위 4비트 : 2진수 비트
 - 존 비트와 2진수 비트를 조합하여 10진수 0~9와 영어 대문자/소문자, 특수문자를 표현

- ASCII 코드의 구성

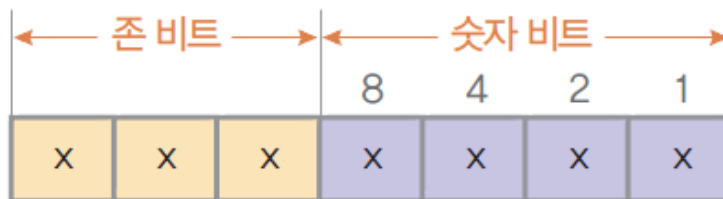


그림 1-21 ASCII 코드의 구성

자료의 표현 : 문자 자료의 표현

■ ASCII 코드

A가 있는 자리의 열(존 비트) '100'과 행(숫자 비트) '0001'을 연결한 '1000001'이 A에 대한 ASCII 코드

표 1-6 ASCII 코드 표

상위 하위	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	\$	4	D	T	d	t
0101	END	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(8	H	X	h	x
1001	HT	EM)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[k	{
1100	FF	FS	,	<	L	W(\)	l	
1101	CR	GS	-	=	M]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

- GS Group Separator
- RS Record Separator
- US Unit Separator

자료의 표현 : 논리자료의 표현

■ 유니코드

- EBCDIC 코드나 ASCII 코드는 최대 8비트로 숫자, 몇 가지 특수문자, 알파벳 정의하므로 문자 코드 표에 정의되어 있지 않은 문자 표현 불가능
- 이런 문제 해결 위해 세계 여러 나라의 언어를 통일된 방법으로 표현할 수 있도록 정의한 국제 표준 코드(ISO/IEC 10646)
- 2바이트를 조합하여 하나의 글자를 표현하기 때문에 1바이트 코드로 표현할 수 없었던 다양한 언어를 표현.
- 유니코드 표는 <http://www.unicode.org/>에서 확인 가능
- 초기 IBM 컴퓨터 시스템에서는 BCD 코드를 사용하다가 더 많은 문자 코드를 표현할 수 있는 EBCDIC코드로 대체, 그러다 미국 표준 코드인 ASCII 코드 일반화, 현재는 표현의 한계를 극복한 유니코드가 일반화
- XML, Java, CORBA 3.0, WML 등 인터넷 기반 프로그램과 제품에 사용

자료의 표현 : 논리자료의 표현

- 논리자료
 - 논리값을 표현하기 위한 자료 형식
 - 논리값
 - 참(True)와 거짓(False), 1과 0
 - 1바이트를 사용하여 논리자료를 표현하는 방법
 - 방법 1)
 - 참 : 최하위 비트를 1로 표시 **00000001**
 - 거짓 : 전체 비트를 0으로 표시. **00000000**
 - 방법2)
 - 참 : 전체 비트를 1로 표시. **11111111**
 - 거짓 : 전체 비트를 0으로 표시. **00000000**
 - 방법3)
 - 참 : 하나 이상의 비트를 1로 표시 **00000001 or 00000100**
 - 거짓 : 전체 비트를 0으로 표시. **00000000**

자료의 표현 : 포인터 자료의 표현

- 포인터 자료
 - 메모리의 주소를 표현하기 위한 자료 형식
 - 변수의 주소나 메모리의 특정 위치에 대한 주소를 저장하고 주소연산하기 위해 사용

자료의 표현 : 문자열 자료의 표현

- 문자열 String 자료
 - 여러 문자로 이루어진 문자의 그룹을 하나의 자료로 취급하여 메모리에 연속적으로 저장하는 자료 형식
 - 하나의 문자열 자료에 포함된 부분문자열을 표현하는 방법
 - 방법 1 : 부분 문자열 사이에 구분자를 사용하여 저장한다.
 - 방법 2 : 가장 긴 문자열의 길이에 맞춰 고정 길이로 저장한다.
 - 방법 3 : 부분 문자열을 연속하여 저장하고 각 부분 문자열에 대한 포인터를 사용한다.

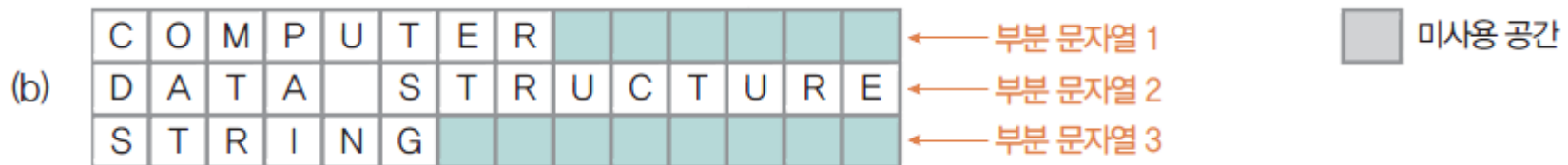
자료의 표현 : 문자열 자료의 표현

■ 문자열 String 자료

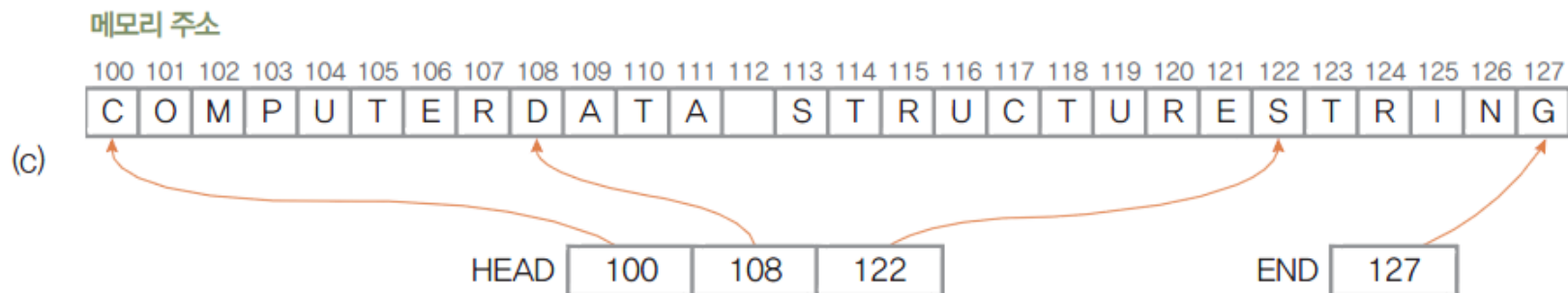
- 방법 1. 구분자를 사용하는 표현 : 구분자로 세미콜론(;) 사용



- 방법 2. 고정길이를 사용하는 표현



- 방법 3. 포인터를 사용하는 표현



자료의 표현 : 문자열 자료의 표현

■ 문자열 String 자료

표 1-7 문자열 표현 방법 비교

비교 항목 방법	메모리 이용률	부분 문자열 탐색 시간
구분자를 사용하는 방법	문자열 길이 + 구분자 길이 → 효율적	문자 비교 연산 시간 + 구분자 식별 시간 → 비효율적
고정 길이로 저장하는 방법	가장 긴 부분 문자열 길이 × 부분 문자열의 개수 → 비효율적	문자 비교 연산 시간 → 효율적
포인터를 사용하는 방법	문자열 길이 + 포인터 저장 공간 → 효율적	문자 비교 연산 시간 + 포인터 주소 연산 시간 → 효율적

알고리즘

요리 재료



케이크 시트(20cm×20cm) 1개, 크림치즈 무스(크림치즈 200g, 달걀 2알, 설탕 3큰술, 레몬즙 1큰술, 바닐라 에센스 1큰술), 딸기 시럽(딸기 500g, 설탕 1½컵, 레몬즙 1작은술), 딸기 1개, 플레인 요구르트 2큰술

요리법

- 1 케이크 틀에 유산지를 깔고 케이크 시트를 놓는다.
- 2 달걀 2알을 잘 푼다. 볼에 크림치즈를 넣고 거품기로 젓는다. 달걀 푼 물과 설탕 3큰술을 세 차례로 나누어 넣으면서 크림 상태가 되도록 거품기로 젓는다.
- 3 2에 레몬즙과 바닐라 에센스를 넣고 살짝 저은 다음 1에 붓는다. 180℃로 예열된 오븐에 전체를 넣고 20분 정도 굽는다.
- 4 딸기를 얇게 자르고 냄비에 넣은 다음 설탕 1½컵을 넣고 약한 불로 끓인다. 끓어붙지 않도록 계속해서 젓고 거품이 생기면 건어 낸다. 되직해지면 레몬즙을 넣고 차갑게 식힌다.
- 5 치즈케이크 한 조각을 접시에 담고 4를 뿌린 다음 플레인 요구르트와 딸기를 얹는다.

그림 1-26 딸기 시럽을 얹은 치즈케이크 만들기

알고리즘의 이해

자료



[요리 재료]

스펀지케이크(20×20cm) 1개, 크림치즈 200g, 달걀 푼 물 2개 분량, 설탕 3큰술, 레몬즙·바닐라에센스 1큰술씩, 딸기시럽(딸기 500g, 설탕 1½ 컵, 레몬즙 1작은술), 딸기 1개, 플레인 요구르트 2큰술

[요리법] >> 알고리즘

- ① 케이크 틀의 가장자리에 필름을 돌린 다음 스펀지케이크를 놓는다.
- ② 볼에 크림치즈를 넣고 거품기로 젓다가 달걀 푼 물과 설탕 3큰술을 세번에 나누어 넣으면서 크림 상태로 만든다.
- ③ ②에 레몬즙과 바닐라에센스를 넣고 살짝 저은 다음 ①에 붓는다. 이것을 180℃의 오븐에 넣고 20분 정도 굽는다.
- ④ 냄비에 슬라이스한 딸기와 설탕 1½ 컵을 넣고 끓이다가 약한 불에서 눌어붙지 않도록 저으면서 거품을 건어낸다. 되직해지면 레몬즙을 넣고 차게 식힌다.
- ⑤ 접시에 치즈케이크를 한 조각 담고 ④의 시럽을 뿌린 다음 플레인 요구르트와 딸기를 엮어낸다

절차

연산

알고리즘

박철수의 전화번호
호는 바로 브부근
으로 넘기면 찾을
수 있겠군

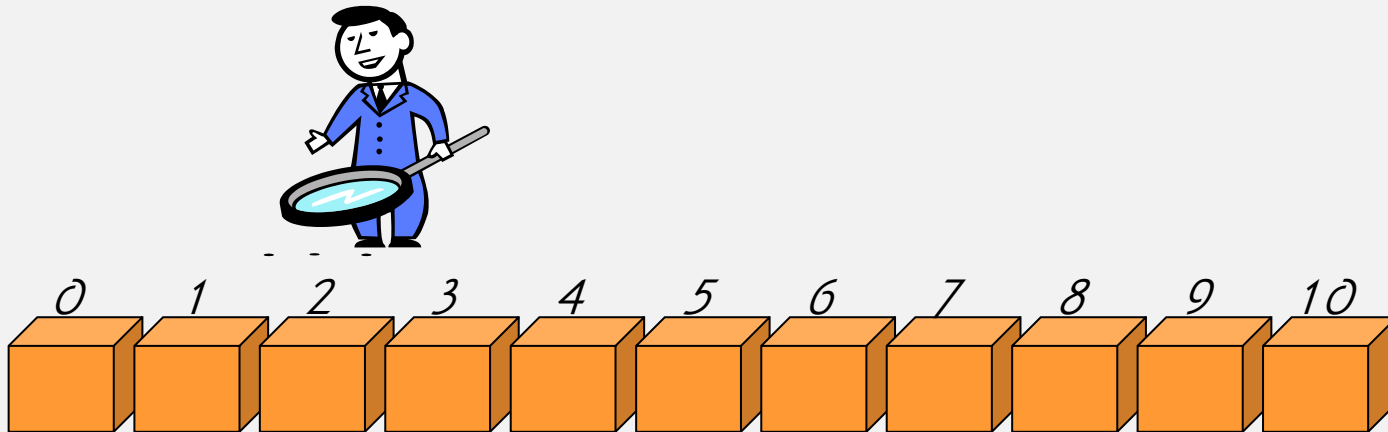
- **알고리즘(algorithm):** 컴퓨터로 문제를 풀기 위한 단계적인 절차
- 알고리즘의 조건
 - 입력 :
 - 알고리즘 수행에 필요한 자료가 외부에서 입력으로 제공될 수 있어야 한다.
 - 0개 이상의 입력이 존재하여야 한다.
 - 출력 : 1개 이상의 출력이 존재하여야 한다.
 - 명백성 :
 - 수행할 작업의 내용과 순서를 나타내는 알고리즘의 명령어들은 명확하게 명세되어야 한다.
 - 각 명령어의 의미는 모호하지 않고 명확해야 한다.
 - 유한성 : 한정된 수의 단계 후에는 반드시 종료되어야 한다.
 - 유효성 : 각 명령어들은 실행 가능한 연산이어야 한다.

알고리즘의 기술 방법

- 영어나 한국어와 같은 자연어
- 흐름도(flow chart)
- 유사 코드(pseudo-code)
- C와 같은 프로그래밍 언어



(예) 배열에서 최대값 찾기 알고리즘



자연어로 표기된 알고리즘

- 인간이 읽기가 쉽다.
- 그러나 자연어의 단어들을 정확하게 정의하지 않으면 의미 전달이 모호해질 우려가 있다.

(예) 배열에서 최대값 찾기 알고리즘

ArrayMax(A,n)

1. 배열 A의 첫번째 요소를 변수 tmp에 복사
2. 배열 A의 다음 요소들을 차례대로 tmp와 비교하면 더 크면 tmp로 복사
3. 배열 A의 모든 요소를 비교했으면 tmp를 반환

흐름도로 표기된 알고리즘

- 순서도를 이용한 도식화
 - 직관적이고 이해하기 쉬운 알고리즘 기술 방법
 - 그러나 복잡한 알고리즘의 경우, 상당히 복잡해짐.
 - 순서도의 예) 1부터 5까지의 합을 구하는 알고리즘

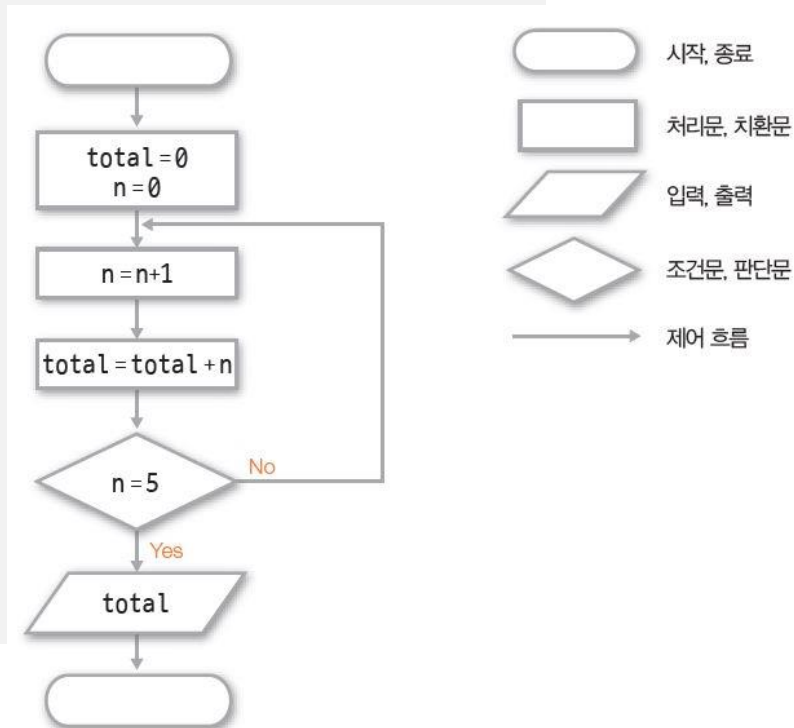


그림 1-27 순서도의 예

유사코드로 표현된 알고리즘

- 알고리즘의 고수준 기술 방법
- 자연어보다는 더 구조적인 표현 방법
- 프로그래밍 언어보다는 덜 구체적인 표현방법
- 가상코드, 즉 알고리즘 기술언어 ADL, Algorithm Description Language를 사용하여 프로그래밍 언어의 일반적인 형태와 유사하게 알고리즘을 표현
- 특정 프로그래밍 언어가 아니므로 직접 실행은 불가능
- 일반적인 프로그래밍 언어의 형태이므로 원하는 특정 프로그래밍 언어로의 변환 용이
- 알고리즘 기술에 가장 많이 사용
- 프로그램을 구현할 때의 여러가지 문제들을 감출 수 있다. 즉 알고리즘의 핵심적인 내용에만 집중할 수 있다.

ArrayMax(A,n)

```
tmp ← A[0];  
for i ← 1 to n-1 do  
    if tmp < A[i] then  
        tmp ← A[i];  
return tmp;
```

대입 연산자가
←임을 유의

C로 표현된 알고리즘

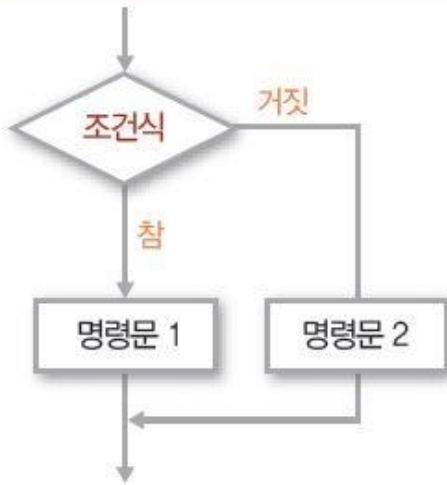
- 알고리즘의 가장 정확한 기술이 가능
- 반면 실제 구현시의 많은 구체적인 사항들이 알고리즘의 핵심적인 내용들의 이해를 방해할 수 있다.

```
#define MAX_ELEMENTS 100
int score[MAX_ELEMENTS];
int find_max_score(int n)
{
    int i, tmp;
    tmp=score[0];
    for(i=1;i<n;i++){
        if( score[i] > tmp ){
            tmp = score[i];
        }
    }
    return tmp;
}
```

알고리즘의 표현 방식

- 조건문
 - 조건에 따라 실행할 명령문이 결정되는 선택적 제어구조를 만든다.
 - if 문의 형식과 제어흐름

```
if (조건식) then 명령문 1;  
else 명령문 2;
```



(a) if - then - else 형

```
if (조건식) then 명령문 1;
```



(b) if - then 형

그림 1-29 기본 if 문의 형식과 제어 흐름

알고리즘의 표현 방식

- 다단계 조건문
 - 중첩 if 문의 형식과 제어 흐름

```
if (조건식 1) then 명령문 1;  
else if (조건식 2) then 명령문 2;  
else 명령문 3;
```

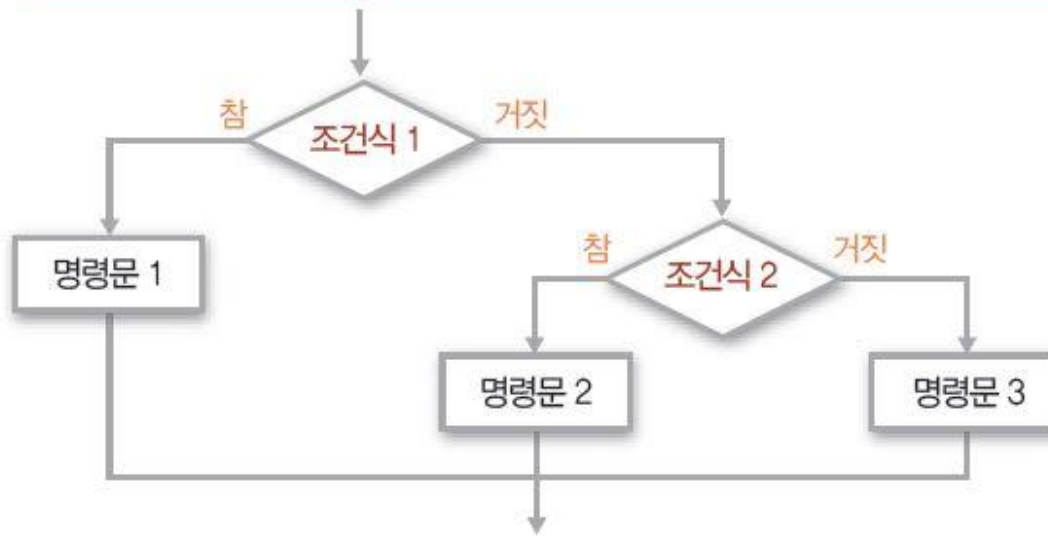


그림 1-30 중첩 if 문의 형식과 제어 흐름

알고리즘의 표현 방식

- 중첩 if문 사용 예) 평균 점수에 따른 등급 계산하기

```
if Average >= 90 then grade ← "A";  
else if Average >= 80 then grade ← "B";  
else if Average >= 70 then grade ← "C";  
else grade ← "F";
```

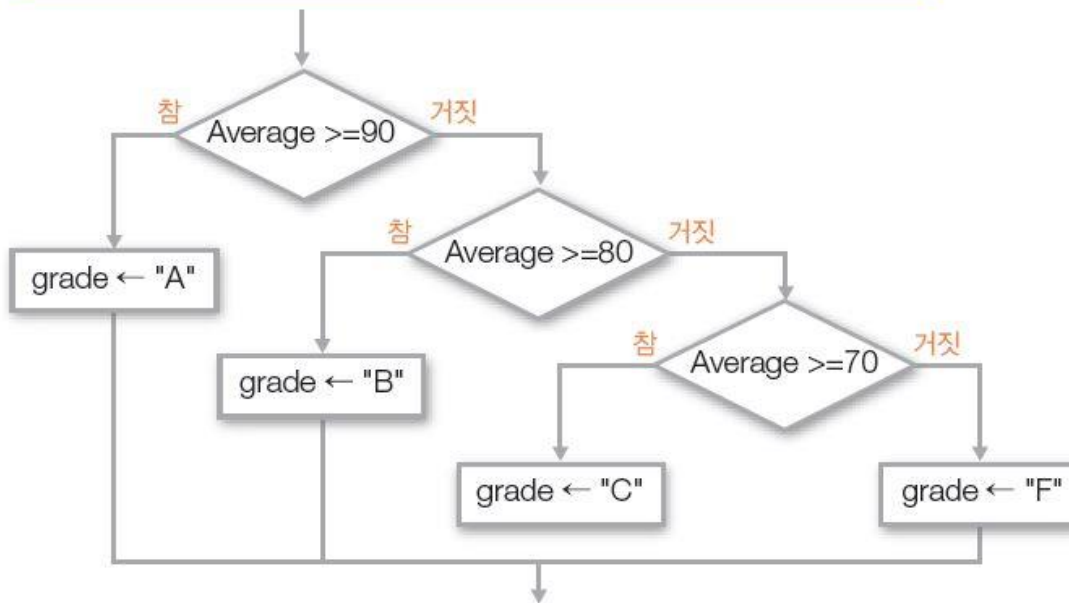


그림 1-31 중첩 if 문의 예

알고리즘의 표현 방식

■ case 문

- 여러 조건식 중에서 해당 조건을 찾아서 그에 대한 명령문을 수행
- 중첩 if 문으로 표현 가능
- 형식과 제어흐름

```
case {  
    조건식 1 : 명령문 1;  
    조건식 2 : 명령문 2;  
    ...  
    조건식 n : 명령문 n;  
    else : 명령문 n+1;  
}
```



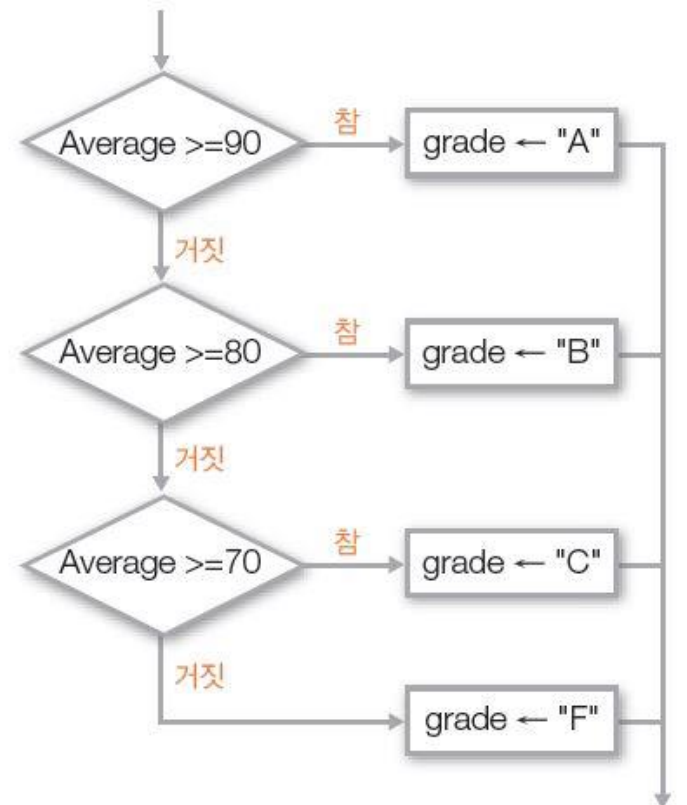
그림 1-32 case 문의 형식과 제어 흐름

알고리즘의 표현 방식

- case 문 예) 평균 점수에 따른 등급 계산하기

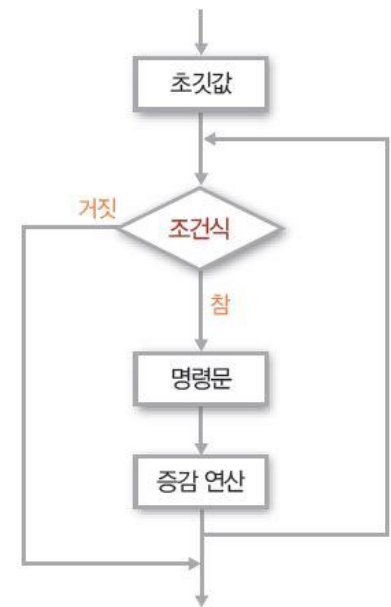
```
case {  
    Average >= 90 : grade ← "A";  
    Average >= 80 : grade ← "B";  
    Average >= 70 : grade ← "C";  
    else :  
        grade ← "F";  
}
```

그림 1-33 case 문의 예



알고리즘의 표현 방식

- 반복문
 - 일정한 명령을 반복 수행하는 루프(loop) 형태의 제어구조
 - for 문
 - 형식과 제어흐름



for (초깃값; 조건식; 증감값) do 명령문;

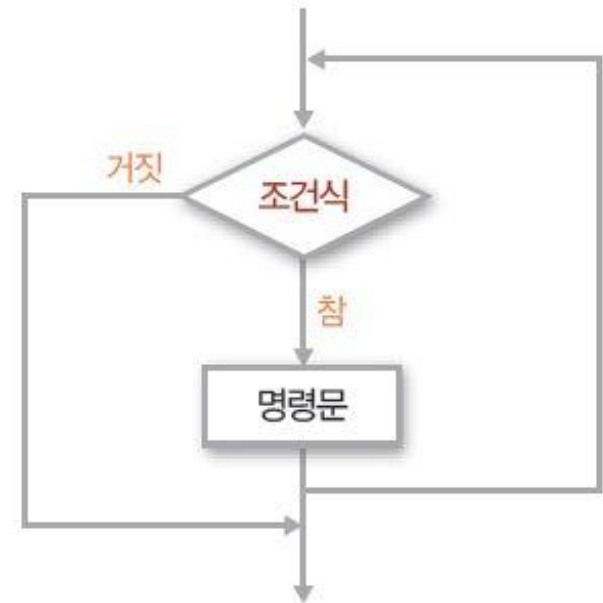
그림 1-34 for 문의 형식과 제어 흐름

알고리즘의 표현 방식

- while – do 문
 - 형식과 제어흐름

```
while (조건식) do 명령문;
```

그림 1-35 while – do 문의 형식과 제어 흐름



알고리즘의 표현 방식

- do-while 문
 - 형식과 제어흐름

```
do 명령문;  
while (조건식);
```

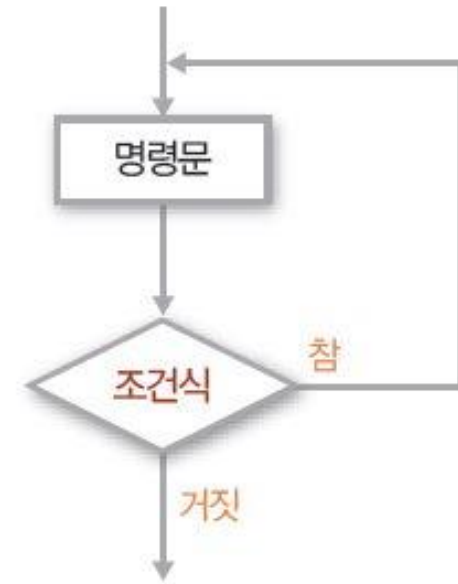


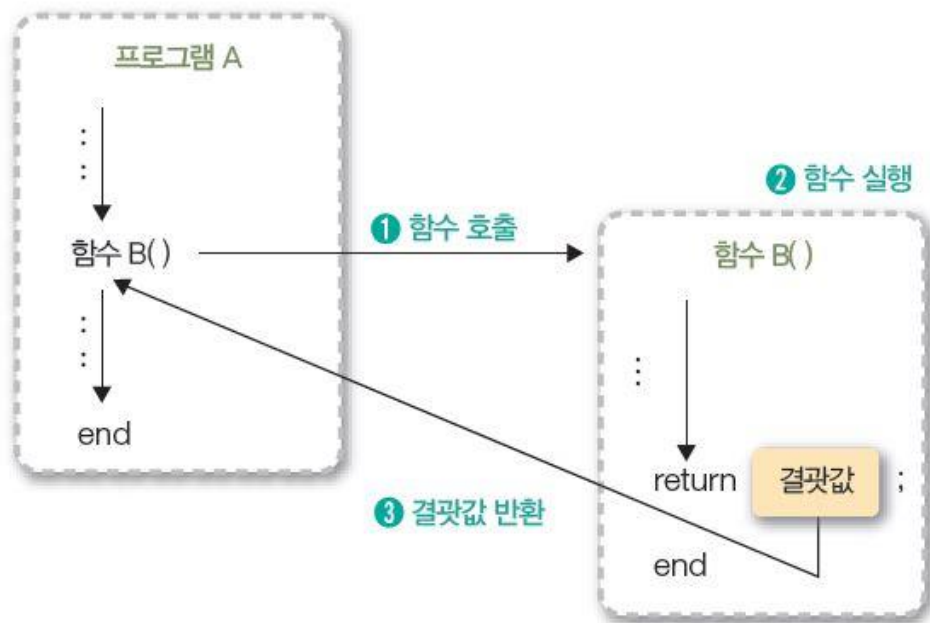
그림 1-36 do-while 문의 형식과 제어 흐름

알고리즘의 표현 방식

- 함수문
 - 처리작업 별로 모듈화하여 만든 부프로그램
 - 형식과 예

```
함수 이름 (매개변수)  
명령문;  
...  
return 결과값;  
end
```

(a) 함수의 형식



(b) 함수의 호출과 실행 및 결과값 반환의 예

그림 1-37 함수의 형식과 예

자료의 추상화

- 뇌의 추상화 기능

- 기억할 대상의 구별되는 특징만을 단순화하여 기억하는 기능

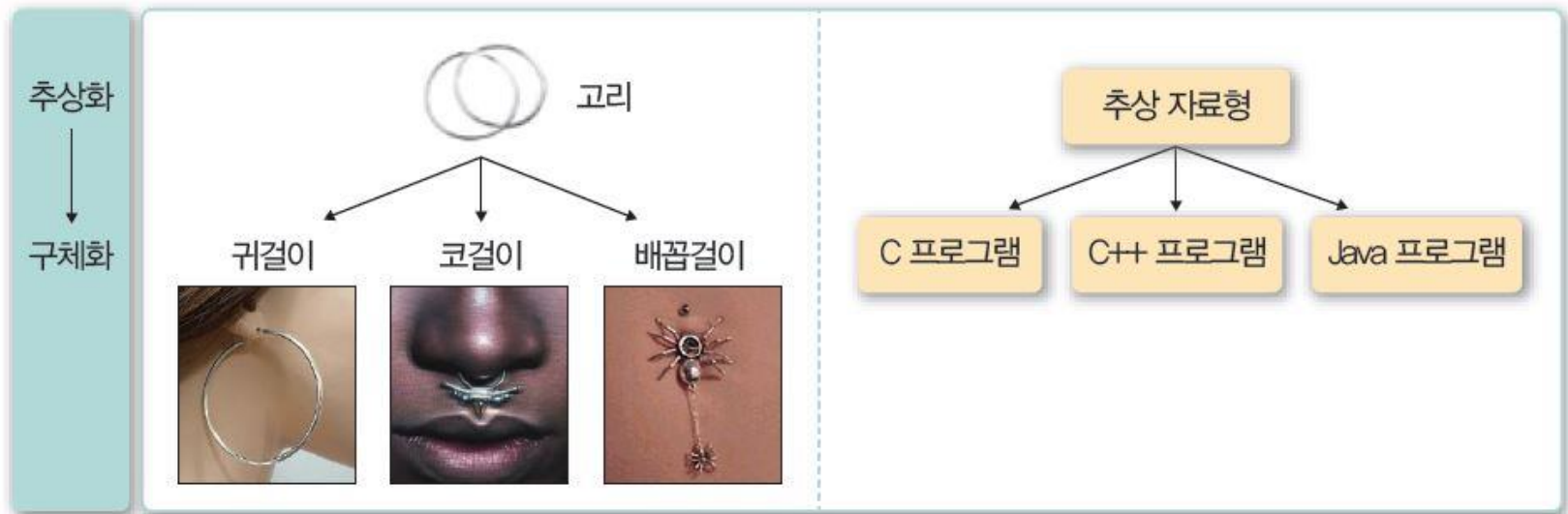


자료의 추상화

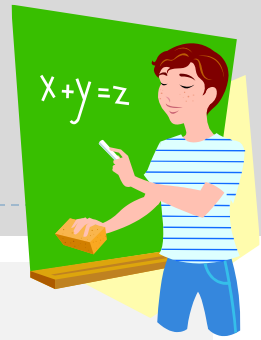
- 컴퓨터를 이용한 문제해결에서의 추상화
 - 크고 복잡한 문제를 단순화시켜 쉽게 해결하기 위한 방법
 - 자료 추상화(Data Abstraction)
 - 처리할 자료, 연산, 자료형에 대한 추상화 표현
 - 자료 : 프로그램의 처리 대상이 되는 모든 것을 의미
 - 연산
 - 어떤 일을 처리하는 과정. 연산자에 의해 수행
 - 예) 더하기 연산은 + 연산자에 의해 수행
 - 자료형
 - 처리할 자료의 집합과 자료에 대해 수행할 연산자의 집합
 - 예) 정수 자료형
자료 : 정수의 집합. {..., -1, 0, 1, ...}
연산자 : 정수에 대한 연산자 집합. {+, -, x, ÷, mod}

자료의 추상화 : 개념

- 추상 자료형(ADT, Abstract Data Type)
 - 자료와 연산자의 특성을 논리적으로 추상화하여 정의한 자료형
- 추상화와 구체화
 - 추상화 – “무엇(what)인가?”를 논리적으로 정의
 - 구체화 – “어떻게(how) 할 것인가?”를 실제적으로 표현



데이터 타입, 추상 데이터 타입



- 데이터 타입(data type)

- 데이터의 집합과 연산의 집합

(예)

int 데이터 타입

{ 데이터: $\{\dots, -2, -1, 0, 1, 2, \dots\}$
연산: $+, -, /, *, \%$

- 추상 데이터 타입(ADT: Abstract Data Type)

- 데이터 타입을 추상적(수학적)으로 정의한 것
 - 데이터나 연산이 **무엇(what)**인가는 정의되지만 데이터나 연산을 **어떻게(how)** 컴퓨터 상에서 구현할 것인지는 정의되지 않는다.

자료의 추상화

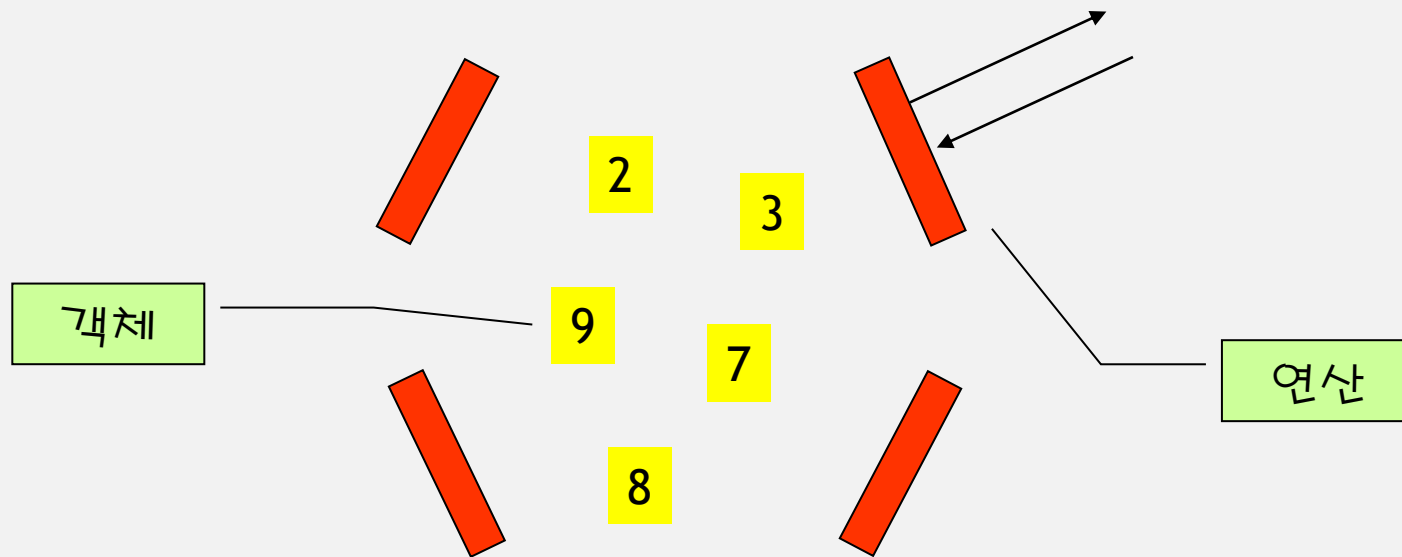
- 추상화와 구체화
 - 자료와 연산에 있어서의 추상화와 구체화의 관계

표 1-8 자료와 연산의 추상화와 구체화

구분	자료	연산
추상화	추상 자료형	알고리즘 정의
구체화	자료형	프로그램 구현

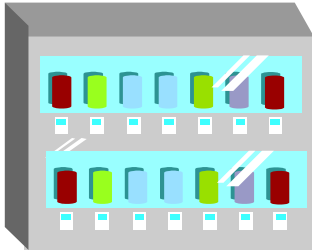
추상 데이터 타입의 정의

- **객체**: 추상 데이터 타입에 속하는 객체가 정의된다.
- **연산**: 이들 객체들 사이의 연산이 정의된다. 이 연산은 추상 데이터 타입과 외부로 연결하는 인터페이스의 역할을 한다.

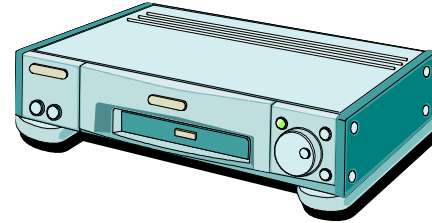


추상 데이터 타입

추상 데이터 타입과 VTR

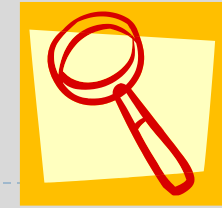


- 사용자들은 추상 데이터 타입이 제공하는 연산만을 사용할 수 있다.
- 사용자들은 추상 데이터 타입을 어떻게 사용하는지를 알아야 한다.
- 사용자들은 추상 데이터 타입 내부의 데이터를 접근할 수 없다.
- 사용자들은 어떻게 구현되었는지 몰라도 이용할 수 있다.
- 만약 다른 사람이 추상 데이터 타입의 구현을 변경하더라도 인터페이스가 변경되지 않으면 사용할 수 있다.



- VCR의 인터페이스가 제공하는 특정한 작업만을 할 수 있다.
- 사용자는 이러한 작업들을 이해해야 한다. 즉 비디오를 시청하기 위해서는 무엇을 해야 하는지를 알아야 한다.
- VCR의 내부를 볼 수는 없다.
- VCR의 내부에서 무엇이 일어나고 있는지를 몰라도 이용할 수 있다.
- 누군가가 VCR의 내부의 기계장치를 교환한다고 하더라도 인터페이스만 바뀌지 않는 한 그대로 사용이 가능하다.

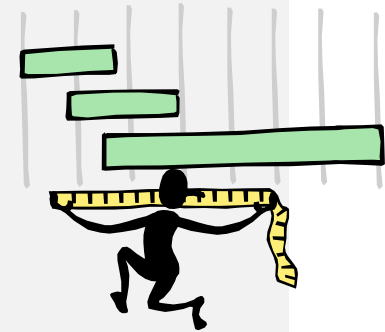
알고리즘의 성능분석



■ 알고리즘의 성능 분석 기법

■ 수행 시간 측정

- 두개의 알고리즘의 실제 수행 시간을 측정하는 것
- 실제로 구현하는 것이 필요
- 동일한 하드웨어를 사용하여야 함



■ 알고리즘의 복잡도 분석

- 직접 구현하지 않고서도 수행 시간을 분석하는 것
- 알고리즘이 수행하는 연산의 횟수를 측정하여 비교
- 일반적으로 연산의 횟수는 n 의 함수
- **시간 복잡도 분석**: 수행 시간 분석
- **공간 복잡도 분석**: 수행시 필요로 하는 메모리 공간 분석



복잡도 분석

- 시간 복잡도는 알고리즘을 이루고 있는 연산들이 몇 번이나 수행되는지를 숫자로 표시
- 산술 연산, 대입 연산, 비교 연산, 이동 연산의 기본적인 연산: 수행시간이 입력의 크기에 따라 변하면 안됨: 기본적인 연산만
- 알고리즘이 수행하는 연산의 개수를 계산하여 두개의 알고리즘을 비교할 수 있다.
- 연산의 수행횟수는 고정된 숫자가 아니라 입력의 개수 n 에 대한 함수->**시간복잡도 함수**라고 하고 $T(n)$ 이라고 표기한다.

프로그램 A



연산의 수 = 8
 $3n+2$

프로그램 B



연산의 수 = 26
 $5n^2 + 6$

수행시간측정

- 컴퓨터에서 수행시간을 측정하는 방법에는 주로 clock 함수가 사용된다.
- clock_t clock(void);
 - clock 함수는 호출되었을 때의 시스템 시각을 CLOCKS_PER_SEC 단위로 반환
- 수행시간을 측정하는 전형적인 프로그램

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void main( void )
{
    clock_t start, finish;
    double duration;
    start = clock();
    // 수행시간을 측정하고 하는 코드....
    // ....
    finish = clock();
    duration = (double)(finish - start) / CLOCKS_PER_SEC;
    printf("%f 초입니다.\n", duration);
}
```



복잡도 분석의 예

- n 을 n 번 더하는 문제:

각 알고리즘이 수행하는 연산의 개수를 세어 본다,
단 for 루프 제어 연산은 고려하지 않음.

알고리즘 A	알고리즘 B	알고리즘 C
sum $\leftarrow n*n$;	sum $\leftarrow 0$; for i $\leftarrow 1$ to n do sum \leftarrow sum + n;	sum $\leftarrow 0$; for i $\leftarrow 1$ to n do for $\leftarrow 1$ to n do sum \leftarrow sum + 1;

	알고리즘 A	알고리즘 B	알고리즘 C
대입연산	1	$n + 1$	$n*n + 1$
덧셈연산		n	$n*n$
곱셈연산	1		
나눗셈연산			
전체연산수	2	$2n + 1$	$2n^2 + 1$

시간복잡도 함수 계산 예

- 코드를 분석해보면 수행되는 연산들의 횟수를 입력 크기의 함수로 만들 수 있다.

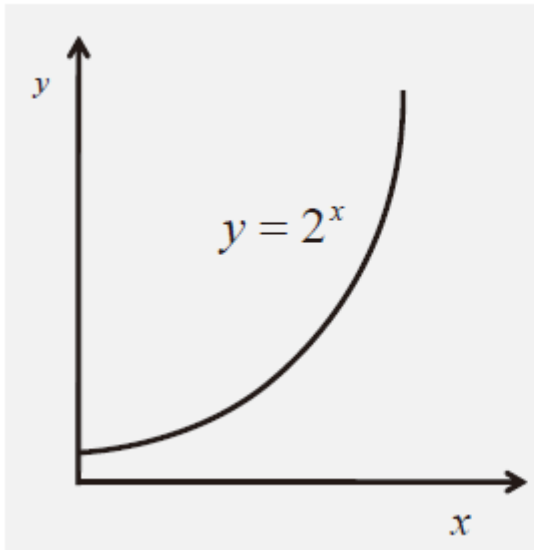
ArrayMax(A,n)

```
tmp ← A[0];  
for i ← 1 to n-1 do  
    if tmp < A[i] then  
        tmp ← A[i];  
return tmp;
```

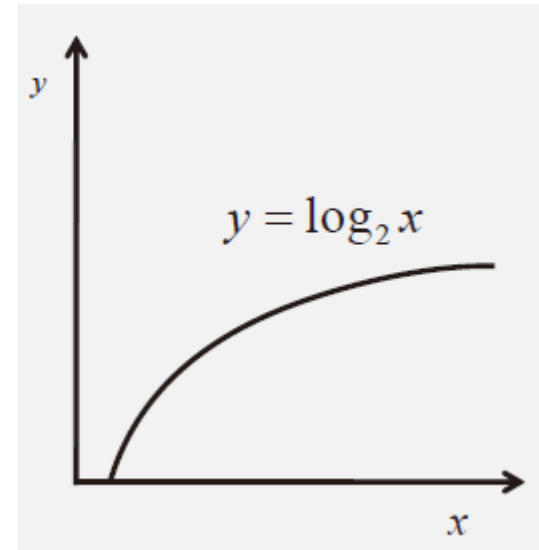
1번의 대입 연산
루프 제어 연산은 제외
n-1번의 비교 연산
n-1번의 대입 연산(최대)
1번의 반환 연산

총 연산수 = $2n$ (최대)

수학과 관련해서 알고 있다고 가정하는 것



지수식



로그식

X 축이 데이터의 수! Y축이 연산의 횟수를 의미한다면?

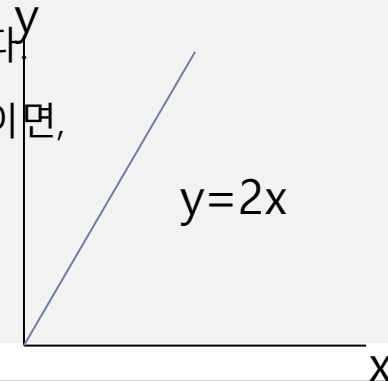
시간 복잡도 & 공간 복잡도 1

■ 알고리즘을 평가하는 두 가지 요소

- 시간 복잡도(time complexity) → 얼마나 빠른가? (CPU)
- 공간 복잡도(space complexity) → 얼마나 메모리를 적게 쓰는가? (MEM)
- **시간 복잡도를 더 중요시 한다. 왜?? 그냥!??**

■ 시간 복잡도의 평가 방법

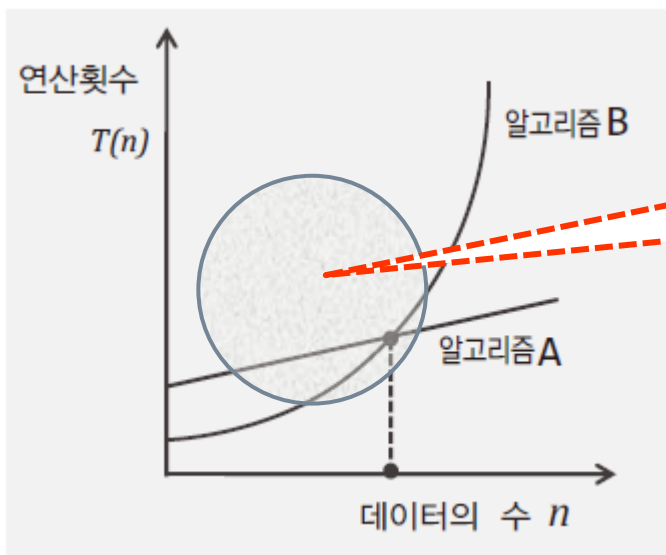
- **중심이 되는 특정 연산**의 횟수를 세어서 평가를 한다.
- 데이터의 수에 대한 연산횟수의 함수 $T(n)$ 을 구한다.
 - 함수로 만들면 그래프를 그리기가 쉬워진다
 - x 는 데이터 수, y 는 처리 횟수 일때, $y=2x$ 이면,
 - $y = T(n)$
 - $x = n$



시간 복잡도 & 공간 복잡도 2

■ 알고리즘의 수행 속도 비교 기준

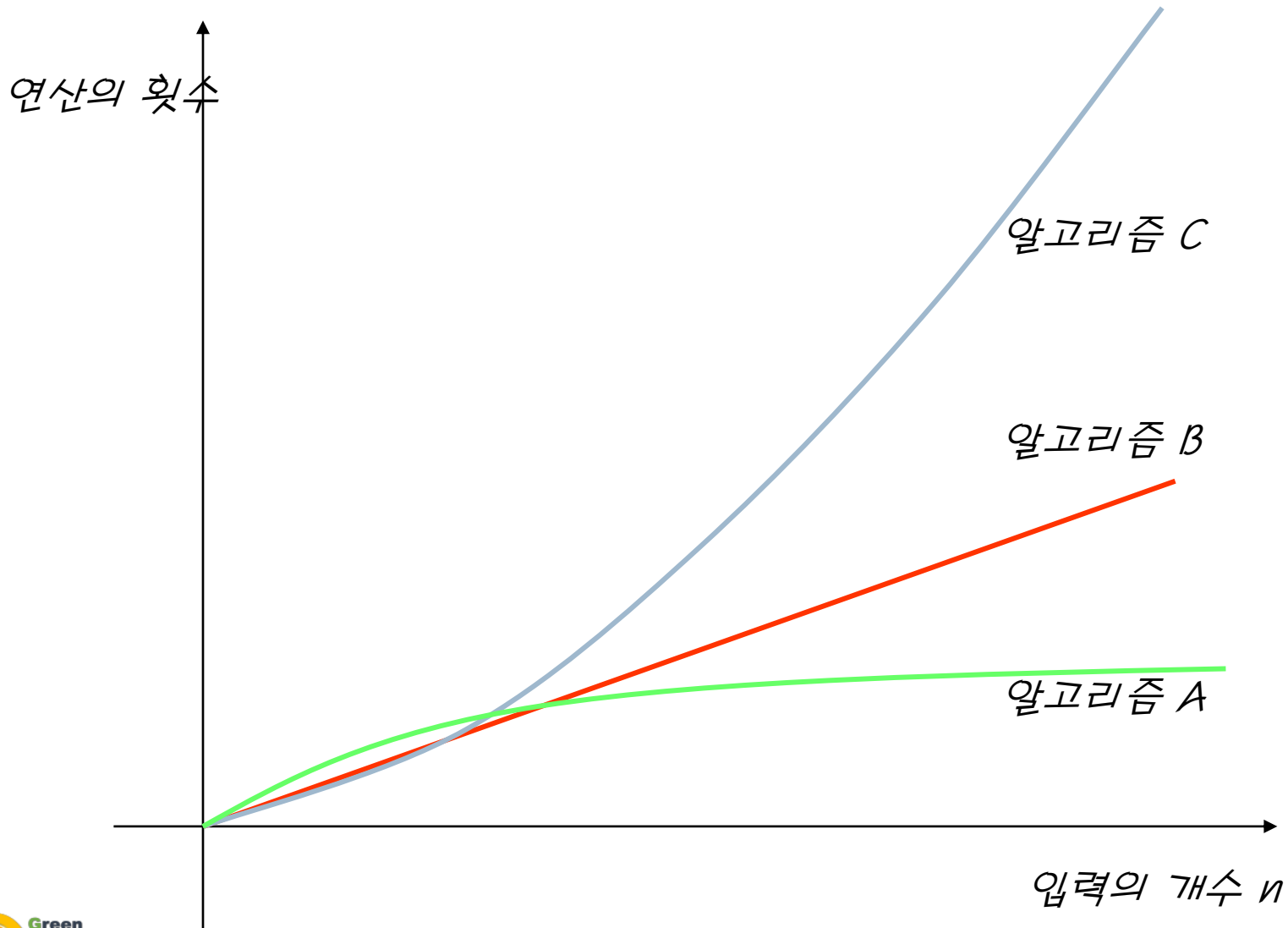
- 데이터의 수가 적은 경우의 수행 속도는 큰 의미가 없다.
- 데이터의 수에 따른 수행 속도의 변화 정도를 기준으로 한다.
 - 데이터 수가 늘어날때 실제 연산의 횟수가 어떤 유형의 패턴을 보이며 변화하는가를 관찰하는 것이 중요
 - 때문에 $T(n)$ 의 함수를 구하고, 그래프를 이용하여 성능을 분석, 평가한다.



이 영역만큼의
데이터 수에 대한
처리 속도는
그냥 신경 쓸 필요가 없어!!!

두 알고리즘을 비교 평가해보면?
어떻게 더 좋은가????

연산의 횟수를 그래프로 표현



순차 탐색 알고리즘과 시간 복잡도

- 순차 탐색 알고리즘??
 - 배열의 처음부터 찾는 요소가 있을 때까지 순차적으로 비교해서 요소를 찾는 방법

순차 탐색 알고리즘과 시간 복잡도

```
// 순차 탐색 알고리즘 적용된 함수
int LSearch(int ar[], int len, int target)
{
    int i;
    for(i=0; i<len; i++)
    {
        if(ar[i] == target)
            return i;    // 찾은 대상의 인덱스 값 반환
    }
    return -1;    // 찾지 못했음을 의미하는 값 반환
}
```

중심 연산자 선정할 수 있어야 함.

어떠한 연산자의 연산 횟수를 세어야 하겠는가?

- 일반적으로..주변 연산자의 연산 횟수는 중심이 되는 연산자의 연산횟수에 의존적이다.
- `<`, `++`는 `==` 가 true를 반환할 때까지 수행된다.

시간 복잡도 계산을 위한 중심 연산자는 무엇일까?

최악의 경우와 최상의 경우

- 순차 탐색 상황 하나: **운이 좋은 경우** -> $T(n)=1$
 - 배열의 맨 앞에서 대상을 찾는 경우
 - 만족스러운 상황이므로 성능평가의 주 관심이 아니다!
 - '**최상의 경우(best case)**'라 한다.

시간 복잡도와 공간 복잡도 각각에 대해서 최악의 경우와 최상의 경우를 구할 수 있다.

- 순차 탐색 상황 둘: **운이 좋지 않은 경우** -> $T(n)=n$
 - 배열의 끝에서 찾거나 대상이 저장되지 않은 경우
 - 만족스럽지 못한 상황이므로 성능평가의 주 관심이다!
 - '**최악의 경우(worst case)**'라 한다.
 - **그런데, 최악의 경우도 빈번한 경우는 아닐 수 있다... 고로, 다음 장~**

평균적인 경우(Average Case)

- 가장 현실적인 경우에 해당한다.
 - 일반적으로 등장하는 상황에 대한 경우의 수이다.
 - 최상의 경우와 달리 알고리즘 평가에 도움이 된다.
 - 하지만 계산하기가 어렵다. 객관적 평가가 쉽지 않다.
- 평균적인 경우의 복잡도 계산이 어려운 이유
 - '평균적인 경우'의 연출이 어렵다.
 - '평균적인 경우'임을 증명하기 어렵다.
 - '평균적인 경우'는 상황에 따라 달라진다.
 - '평균적인 경우'를 구하는 것 자체가 무의미 할 수 있다 – **“최악의 경우는 항상 동일하다.”**

순차 탐색 최악의 경우 시간 복잡도

“데이터의 수가 n 개일 때,

최악의 경우에 해당하는 연산횟수는(비교연산의 횟수는) n 이다.”

$T(n) = n$ 최악의 경우를 대상으로 정의한 함수 $T(n)$

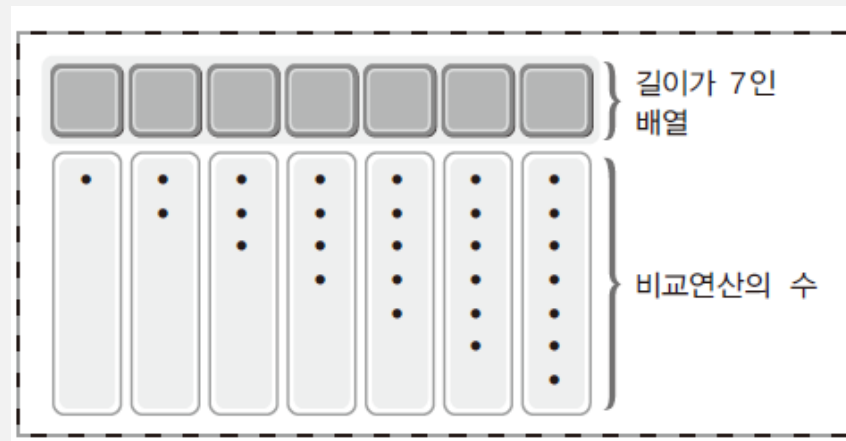
최선, 최악, 평균에 대한 언급이 없으면 최악의 경우가 기본이 된다.

순차 탐색 평균적 경우 시간 복잡도

- 순차 탐색의 평균적 경우가 진짜 별로 인가?

- 평균적 경우에 대한 가정

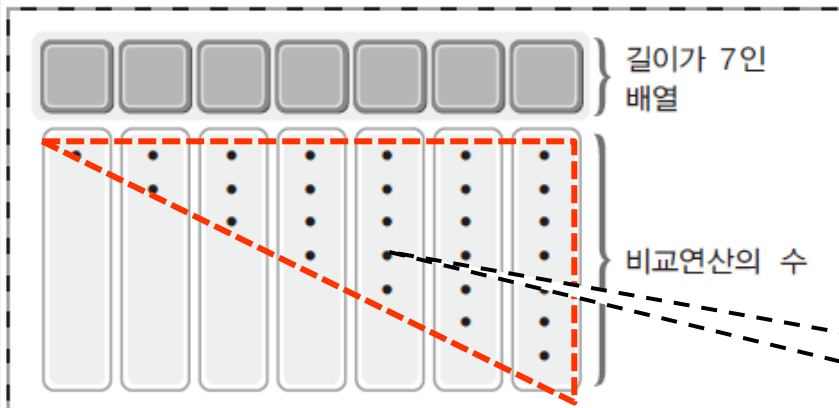
- 가정 1 탐색 대상이 배열에 존재하지 않을 확률 50% = 존재할 확률 50%
 - 가정 2 배열 첫 요소부터 마지막 요소까지 탐색 대상 존재 확률 동일!
= 찾고자 하는 모든 대상이 배열에 존재한다. 50%



가정 1과 2는 평균적인 경우라 할 수 있겠는가? 그렇다면 무엇이 평균적인 경우이겠는가?

순차 탐색 평균적 경우 시간 복잡도

- 길이 7인 배열에서 타겟을 찾기 위해 각 배열에 대한 순차 탐색을 할 경우 평균값은 $\frac{7}{2}$ 즉, $\frac{n}{2}$
 - 탐색 대상이 존재하지 않는 경우 연산횟수는 n
 - 확률이 50%이므로 $n * \frac{1}{2}$
 - 가정 2에서 탐색 대상이 존재하는 경우의 연산 횟수 $n/2$
 - 확률이 50%이므로 $\frac{n}{2} * \frac{1}{2}$



$$T(n) = n \times \frac{1}{2} + \frac{n}{2} \times \frac{1}{2} = \frac{3}{4}n$$

결론!

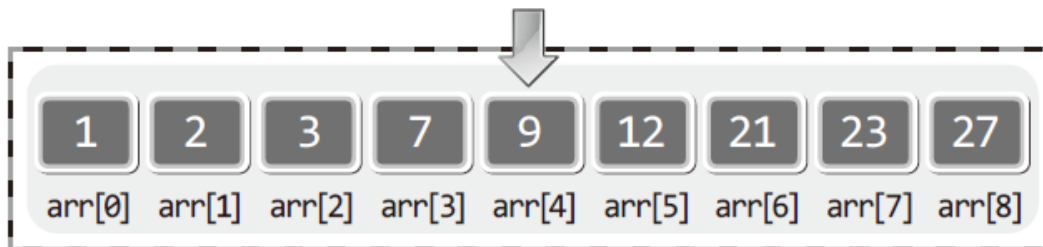
대상이 존재할때
총 연산 횟수

이진 탐색 알고리즘의 소개 1

- 순차 탐색보다 훨씬 좋은 성능을 보이지만, 배열이 정렬되어 있어야 한다는 제약이 따른다.
- 정렬의 기준이 정해져 있는것은 아니다.. (오름이든 내림이든..)

👉 이진 탐색 알고리즘의 1st Step:

1. 배열 인덱스의 시작과 끝은 각각 0과 8이다.
2. 0과 8을 합하여 그 결과를 2로 나눈다. $(0+8)/2=4$
3. 2로 나뉘서 얻은 결과 4를 인덱스 값으로 하여 `arr[4]`에 저장된 값과 찾고자 하는 3과 비교.



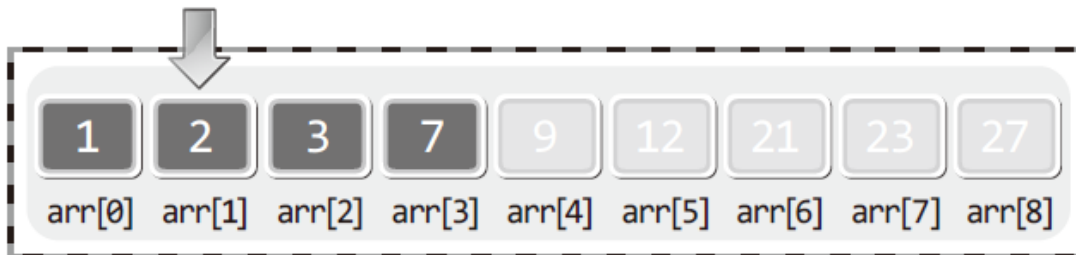
3이 저장되어 있는지
를 탐색한다!

이진 탐색 알고리즘의 소개 2

- 일단, 탐색의 대상이 첫 번째 시도 이후 반으로 줄었다!

👉 이진 탐색 알고리즘의 2nd Step:

- arr[4]에 저장된 값 9와 탐색 대상인 3의 대소를 비교한다. – **방향 결정**
- 대소 비교결과는 $\text{arr}[4] > 3$ 이므로 탐색 범위를 인덱스 기준 0~3으로 제한!
- 0과 3을 더하여 그 결과를 2로 나눈다. 이때 나머지는 버린다.
- 2로 나눠서 얻은 결과가 1이니 arr[1]에 저장된 값과 찾는 값 3을 비교한다.

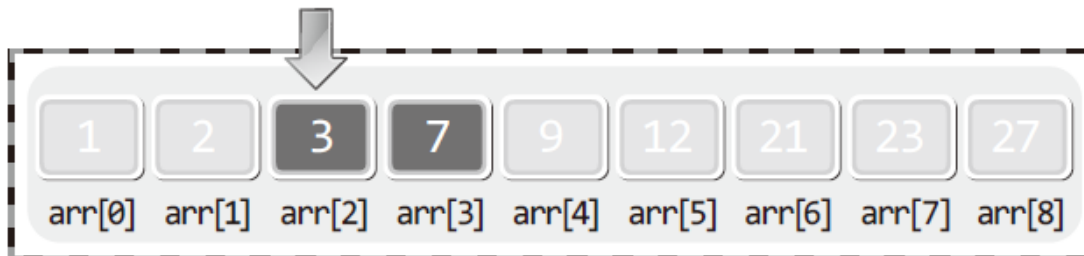


이진 탐색 알고리즘의 소개 3

- 탐색의 대상이 다시 반으로 줄었다!

■ 이진 탐색 알고리즘의 세 번째 시도:

1. `arr[1]`에 저장된 값 2와 탐색 대상인 3의 대소를 비교한다. – **방향 결정**
2. 대소 비교결과 `arr[1] < 3`이므로 탐색의 범위를 인덱스 기준 2~3으로 제한!
3. 2와 3을 더하여 그 결과를 2로 나눈다. 이때 나머지는 버린다.
4. 2로 나뉘서 얻은 결과가 2이니 `arr[2]`에 저장된 값과 찾는 값 3을 비교한다.



이진 탐색 알고리즘의 소개 4

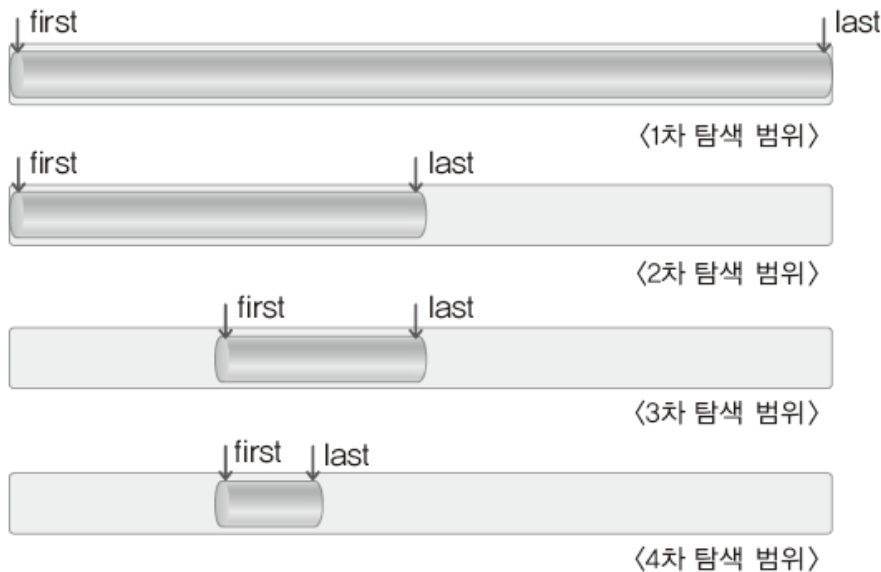
- 이진 탐색의 매 과정마다 탐색의 대상을 반씩 줄여나가기 때문에 순차 탐색보다 비교적 좋은 성능을 보인다.



이진 탐색 알고리즘의 핵심!

이진 탐색 알고리즘의 구현 1

여기서 문제!! 탐색대상이 없는 경우는 언제 판단하게 될까?



- first – 탐색 대상의 첫번째 요소
- Last – 탐색 대상의 마지막 요소
- 2진 탐색의 특성상 first는 오른쪽으로 이동하고, last는 왼쪽으로 이동한다.
- 점점 좁혀져서 **first와 last가 만났다는 것은 탐색 대상이 하나 남았다**는 것을 뜻함!
 - 검사가 안끝났다!!!
- 따라서 **first와 last가 역전될때까지** 탐색의 과정을 계속! – 탐색 대상이 없다!!!

이진 탐색 알고리즘의 구현 1

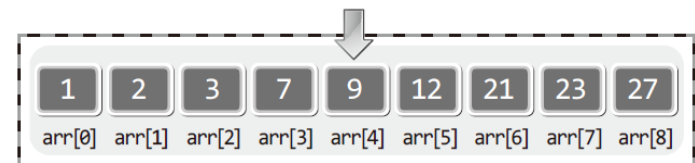
이진 탐색의 기본 골격

```
while(first <= last)    // first <= last 가 반복의 조건임에 주의!  
{  
    // 이진 탐색 알고리즘의 진행  
}
```

이진 탐색 알고리즘의 구현 2

```
int BSearch(int ar[], int len, int target)
{
    int first = 0;    // 탐색 대상의 시작 인덱스 값
    int last = len-1; // 탐색 대상의 마지막 인덱스 값
    int mid;

    while(first <= last)
    {
        mid = (first+last) / 2;    // 탐색 대상의 중앙을 찾는다.
        if(target == ar[mid])    // 중앙에 저장된 것이 타겟이라면
        {
            return mid; // 탐색 완료!
        }
        else    // 타겟이 아니면 탐색 대상을 반으로 줄인다.
        {
            if(target < ar[mid])
                last = mid-1;    // 왜 -1을 하였을까?
            else
                first = mid+1;    // 왜 +1을 하였을까?
        }
    }
    return -1;    // 찾지 못했을 때 반환되는 값 -1
}
```



Target=3

미드값 9를 취한후 오른쪽을 버렸을

last=

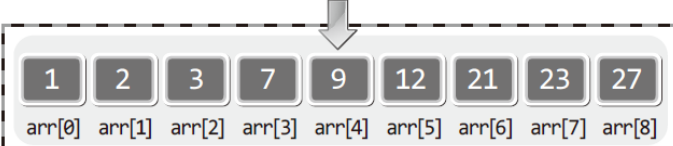
-1 혹은 +1을 추가하지 않으면

first <= mid <= last가 항상 성립이 되어,
탐색 대상이 존재하지 않는 경우 first와 last의
역전 현상이 발생하지 않는다!

이진 탐색 알고리즘의 구현 2

```
int BSearch(int ar[], int len, int target)
{
    int first = 0;    // 탐색 대상의 시작 인덱스 값
    int last = len-1; // 탐색 대상의 마지막 인덱스 값
    int mid;

    while(first <= last)
    {
        mid = (first+last) / 2;    // 탐색 대상의 중앙을 찾는다.
        if(target == ar[mid])    // 중앙에 저장된 것이 타겟이라면
        {
            return mid; // 탐색 완료!
        }
        else    // 타겟이 아니라면 탐색 대상을 반으로 줄인다.
        {
            if(target < ar[mid])
                last = mid-1;    // 왜 -1을 하였을까?
            else
                first = mid+1;    // 왜 +1을 하였을까?
        }
    }
    return -1;    // 찾지 못했을 때 반환되는 값 -1
}
```



1	2	3	7	9	12	21	23	27
arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]	arr[6]	arr[7]	arr[8]

- Target=3
- 미드값 9를 취한후 오른쪽을 버렸을때, last=mid-1을 수행해야 하는 이유??
단순히 9를 한번 더 탐색하는게 문제가 되는가???

-1 혹은 +1을 추가하지 않으면
first <= mid <= last가 항상 성립이 되어,
탐색 대상이 존재하지 않는 경우 first와 last의
역전 현상이 발생하지 않는다!

이진 탐색 알고리즘 최악의 경우 시간 복잡도1

```
while(first <= last)
{
    mid = (first+last) / 2;
    if(target == ar[mid])
    {
        return mid;
    }
    else    // 타겟이 아니라면
    {
        ....
    }
}
```

시간 복잡도 계산을 위한 핵심 연산은?

== 연산자!

따라서 == 연산자의 연산 횟수를 기준으로
시간 복잡도를 결정할 수 있다.

데이터의 수가 n 개일 때, 최악의 경우에 발생하는 비교연산의 횟수는?

이진 탐색 알고리즘 최악의 경우 시간 복잡도2

☞ 데이터의 수가 n 개일 때 비교 연산의 횟수

- 처음에 데이터 수가 n 개일 때의 탐색과정에서 1회의 비교연산 진행
- 데이터 수를 반으로 줄여서 그 수가 $n/2$ 개일 때의 탐색과정에서 1회 비교연산 진행
- 데이터 수를 반으로 줄여서 그 수가 $n/4$ 개일 때의 탐색과정에서 1회 비교연산 진행
- 데이터 수를 반으로 줄여서 그 수가 $n/8$ 개일 때의 탐색과정에서 1회 비교연산 진행

..... n 이 얼마인지 결정되지 않았으니

이 사이에 도대체 몇 번의 비교연산이 진행되는지 알 수가 없다!.....

- 데이터 수를 반으로 줄여서 그 수가 1개일 때의 탐색과정에서 1회의 비교연산 진행

• 이러한 표현 방법으로는 객관적 성능 비교 불가!

언제까지 인지는 대답 가능하나 몇번인지는 알 수 없다.

이진 탐색 알고리즘 최악의 경우 시간 복잡도3

Needs... Math...

- 비교 연산의 횟수의 예 – 데이터 개수가 8일때
 - 8일때 1번, 4일때 1번, 2일때 1번, 1일때 1번
 - 8이 1이 되기까지 2로 나눈 횟수 3회, 따라서 비교연산 3회 진행
 - 데이터가 1개 남았을 때, 이때 마지막으로 비교연산 1회 진행
 - 즉, 3+1회
- 비교 연산의 횟수의 일반화
 - n 이 1이 되기까지 2로 나눈 횟수 k 회, 따라서 비교연산은 k 회 진행
 - 데이터가 1개 남았을 때, 이때 마지막으로 비교연산 1회 진행
 - 고로, $k+1$ 회
- 비교 연산의 횟수의 1차 결론!
 - 최악의 경우에 대한 시간 복잡도 함수 $T(n)=k+1$
 - k 는 ???

이진 탐색 알고리즘 최악의 경우 시간 복잡도4

$$n \times \left(\frac{1}{2}\right)^k = 1$$

$n = 8$ 일때 k 는?? 3
고로, $T(8) = 3 + 1 = 4$

- $k = 1$
 n 을 2로 1번 나눔
- $k = 2$
 n 을 2로 2번 나눔
- $k = 3...$
- 결국, n 을 몇번 반으로 나눠야 1이 되는가? 를 결정한 수식
- k 가 나눠야 하는 횟수를 말한다.

👉 k 에 관한 식으로...

$$\begin{aligned} n \times \left(\frac{1}{2}\right)^k = 1 & \xrightarrow{\frac{1}{2} = 2^{-1}} n \times 2^{-k} = 1 \xrightarrow{} \underline{n = 2^k} \\ n = 2^k & \xrightarrow{} \log_2 n = \log_2 2^k \xrightarrow{} \log_2 n = k \log_2 2 \xrightarrow{} \boxed{\log_2 n = k} \\ & \text{밑이 2일 log를 취하면} \quad = 1 \end{aligned}$$

이진 탐색 알고리즘 최악의 경우 시간 복잡도5

■ 결론

- 함수 $T(n) = k + 1$ 에서
- $k = \log_2 n$ 이므로
- $T(n) = \log_2 n + 1$
- +1은 생략하고
- $T(n) = \log_2 n$

■ 왜 +1은 생략하는가?

- 얻고자 하는것이 알고리즘에서 n의 증가에 따른 수행 횟수 변화에 대한 패턴을 구하고자 하는 것임으로.. +1의 의미가 없다!!

이진 탐색 알고리즘 최악의 경우 시간 복잡도6

그렇다면 $+1$ 이 아닌 $+200$ 인 경우도 생략이 가능한가?

빅-오에 대한 이해를 통해서 이에 대한 답을 얻자!

- $T(n) = n^2 + 100$ 을 $T(n) = n^2$ 이라 표현하는 것이 맞는가??
 - 맞다... 시간 복잡도는 최악의 경우를 본다
 - 즉, 데이터의 수가 급격히 증가하는 경우의 알고리즘의 성능을 평가하기 위한 것이므로... n 이 중요하다.
- $T(n) = n^2 + 100,000,000$ 을 $T(n) = n^2$ 으로 표현하는 것도 맞는가?
 - 맞다!!!!

빅-오 표기법(Big-Oh Notation)

$$T(n) = 17 + 2n$$

↓ 1차 근사치 식

$$T(n) = 17 + 2n$$

↓ 2차 근사치 식

$$T(n) = 17$$

↓ $T(n)$ 의 빅-오

$$O(n)$$

빅-오의 표기 방법

$T(n)$ 에서 실제로 영향력을 끼치는 부분을 큰 오로 감싸서 표현해 놓은 것을 가리켜 빅-오(Big-Oh)라 한다.

☞ n 의 변화에 따른 $T(n)$ 의 변화 정도를 판단하는 것이 목적이니 $+1$ 은 무시할 수 있다.

n	n^2	$2n$	$T(n)$	n^2 의 비율
10	100	20	120	83.33%
100	10,000	200	10,200	98.04%
1,000	1,000,000	2,000	1,002,000	99.80%
10,000	100,000,000	20,000	100,020,000	99.98%
100,000	10,000,000,000	200,000	10,000,200,000	99.99%

☞ $2n$ 도 근사치 식의 구성에서 제외시킬수 있음을 보인 결과!
 n 이 증가함에 따라 $2n+1$ 이 차지하는 비율은 미미해진다.

단순하게 빅-오 구하기

👉 빅-오는?

- $T(n)$ 이 다항식으로 표현이 된 경우, **최고차항의 차수만**가 빅-오가 된다.

👉 빅-오 결정의 예

최고차항의 차수를 빅-오로 결정!

$$\bullet T(n) = \underline{n^2} + 2n + 9 \quad \blacktriangleright \quad O(n^2)$$

$$\bullet T(n) = \underline{n^4} + n^3 + n^2 + 1 \quad \blacktriangleright \quad O(n^4)$$

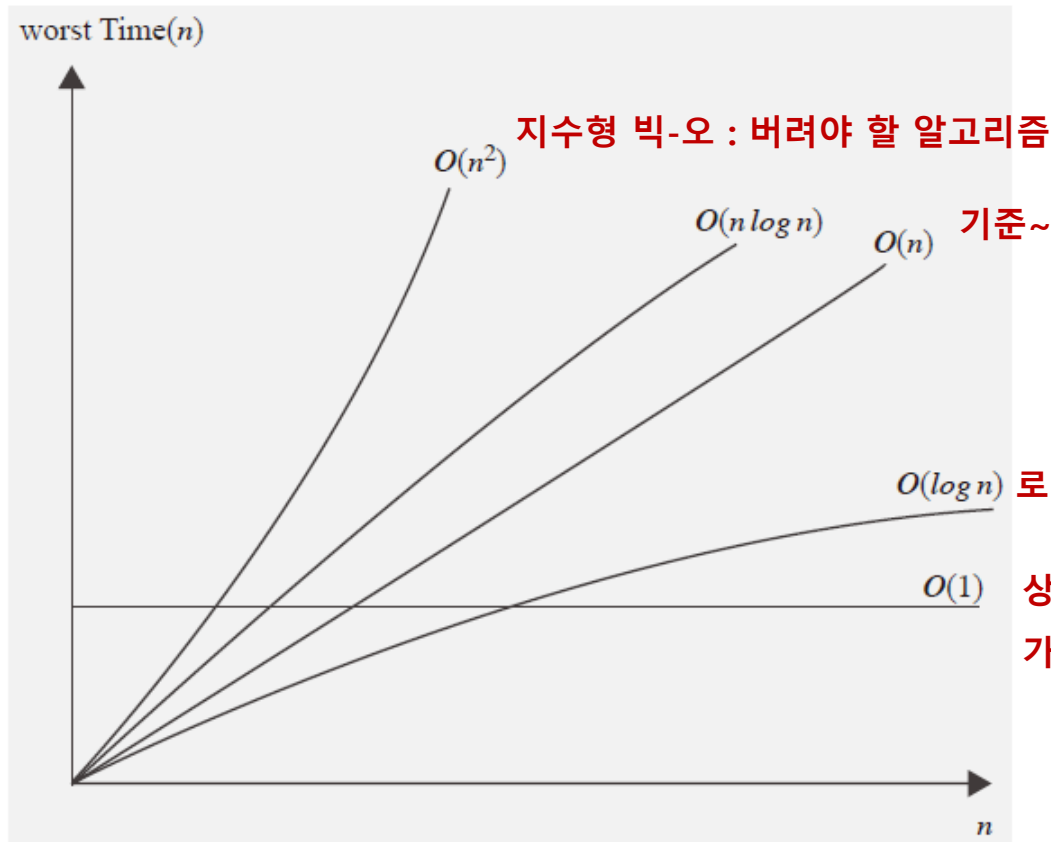
$$\bullet T(n) = \underline{5n^3} + 3n^2 + 2n + 1 \quad \blacktriangleright \quad O(n^3)$$

결국 그래프의 기울기 패턴은 비슷해지므로 $5n^3$ 의 5도 생략 가능

👉 빅-오 결정의 일반화

$$\bullet T(n) = \cancel{a_m n^m} + a_{m-1} n^{m-1} + \dots + a_1 n^1 + a_0 \quad \blacktriangleright \quad O(n^m)$$

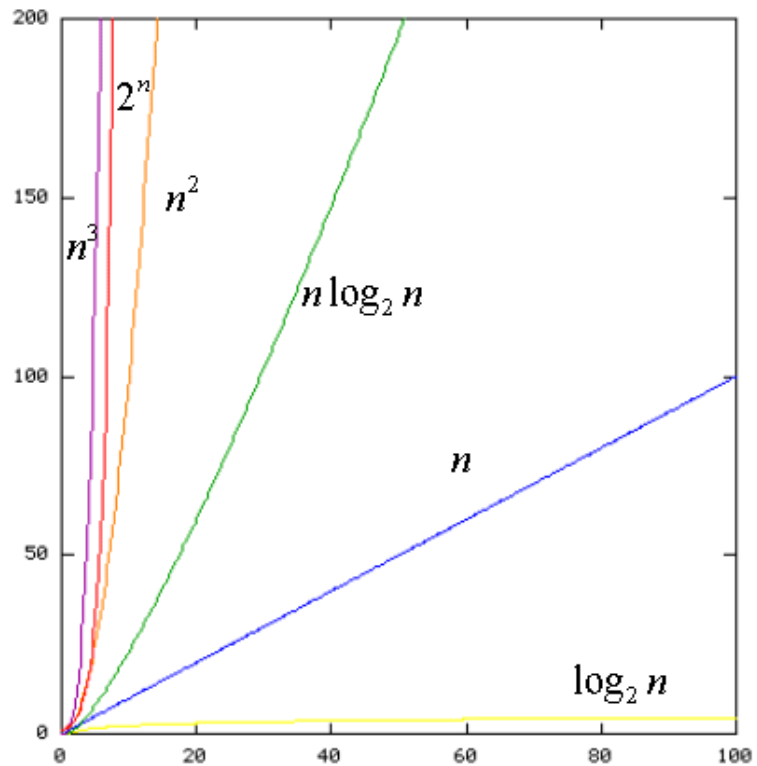
대표적인 빅-오



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

빅오 표기법의 종류

- $O(1)$: 상수형
- $O(\log n)$: 로그형
- $O(n)$: 선형
- $O(n \log n)$: 로그선형
- $O(n^2)$: 2차형
- $O(n^3)$: 3차형
- $O(n^k)$: k차형
- $O(2^n)$: 지수형
- $O(n!)$: 팩토리얼형



빅오 표기법의 종류

시간복잡도	n					
	1	2	4	8	16	32
1	1	1	1	1	1	1
$\log n$	0	1	2	3	4	5
n	1	2	4	8	16	32
$n \log n$	0	2	8	24	64	160
n^2	1	4	16	64	256	1024
n^3	1	8	64	512	4096	32768
2^n	2	4	16	256	65536	4294967296
$n!$	1	2	24	40320	20922789888000	26313×10^{33}

6. 알고리즘의 성능분석

$\log n$	<	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		65536
5		32		160		1024		32768		4294967296

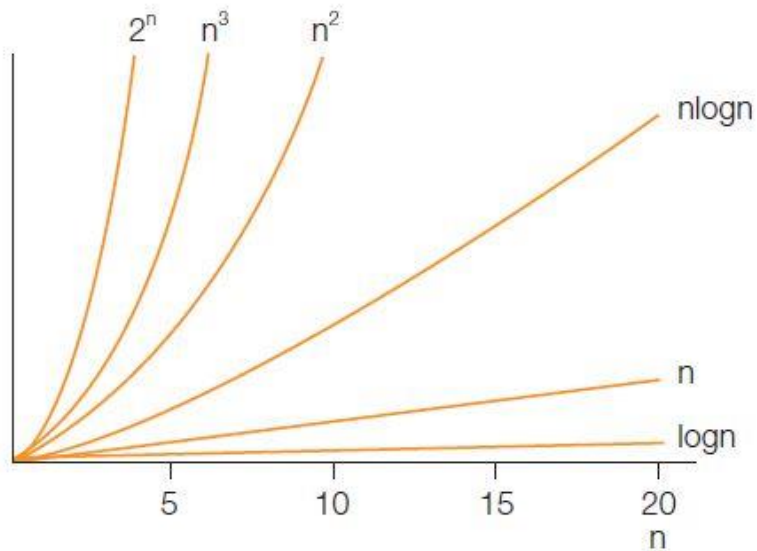


그림 1-38 실행 시간 함수에서 n 값의 변화에 따른 실행 빈도수 비교

6. 알고리즘의 성능분석

표 1-10 시간 복잡도에 따른 알고리즘 수행 시간 비교 예

입력 크기 n	알고리즘 수행 시간				
	n	nlogn	n ²	n ³	2 ⁿ
10	10 ⁻⁸ 초	3×10 ⁻⁸ 초	10 ⁻⁷ 초	10 ⁻⁶ 초	10 ⁻⁶ 초
30	3×10 ⁻⁸ 초	2×10 ⁻⁷ 초	9×10 ⁻⁷ 초	3×10 ⁻⁵ 초	1초
50	5×10 ⁻⁸ 초	3×10 ⁻⁷ 초	3×10 ⁻⁶ 초	10 ⁻⁴ 초	13일
100	10 ⁻⁷ 초	7×10 ⁻⁷ 초	10 ⁻⁵ 초	10 ⁻³ 초	4×10 ¹³ 년
1,000	10 ⁻⁶ 초	10 ⁻⁵ 초	10 ⁻³ 초	1초	3×10 ²⁸³ 년
10,000	10 ⁻⁵ 초	10 ⁻⁴ 초	10 ⁻¹ 초	17분	
100,000	10 ⁻⁴ 초	2×10 ⁻³ 초	10초	12일	
1,000,000	10 ⁻³ 초	2×10 ⁻² 초	17분	32년	

순차 탐색 알고리즘 vs. 이진 탐색 알고리즘

$$\bullet T(n) = n$$

순차 탐색의 최악의 경우 시간 복잡도



$$O(n)$$

순차 탐색의 빅-오

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

이진 탐색의 최악의 경우 시간 복잡도



$$O(\log n)$$

이진 탐색의 빅-오

n	순차 탐색 연산횟수	이진 탐색 연산횟수
500	500	9
5,000	5,000	13
50,000	50,000	16

이진 탐색의 연산횟수는
예제 BSWorstOpCount.c
에서 확인한 결과

최악의 경우에 진행이 되는 연산의 횟수

BSWorstOpCount.c

```
#include <stdio.h>

int BSearch(int ar[], int len, int target)
{
    int first=0;
    int last=len-1;
    int mid;
    int opCount=0;    // 비교연산의 횟수를 기록

    while(first<=last)
    {
        mid=(first+last)/2;

        if(target==ar[mid])
        {
            return mid;
        }
        else
        {
            if(target<ar[mid])
                last=mid-1;
            else
                first=mid+1;
        }
        opCount+=1;    // 비교연산의 횟수 1 증가
    }
    printf("비교연산 횟수: %d \n", opCount);    // 탐색 실패 시 연산횟수 출력
    return -1;
}
```

BSWorstOpCount.c

```
int main(void)
{
    int arr1[500]={0,}; // 모든 요소 0으로 초기화
    int arr2[5000]={0,}; // 모든 요소 0으로 초기화
    int arr3[50000]={0,}; // 모든 요소 0으로 초기화
    int idx;

    // 저장되지 않은 정수 1을 찾으라고 명령
    idx=BSearch(arr1, sizeof(arr1)/sizeof(int), 1);
    if(idx==-1)
        printf("탐색 실패 \n\n");
    else
        printf("타겟 저장 인덱스: %d \n", idx);

    // 저장되지 않은 정수 2를 찾으라고 명령
    idx=BSearch(arr2, sizeof(arr2)/sizeof(int), 2);
    if(idx==-1)
        printf("탐색 실패 \n\n");
    else
        printf("타겟 저장 인덱스: %d \n", idx);

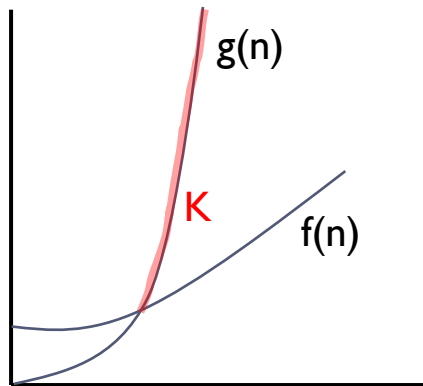
    // 저장되지 않은 정수 3을 찾으라고 명령
    idx=BSearch(arr3, sizeof(arr3)/sizeof(int), 3);
    if(idx==-1)
        printf("탐색 실패 \n\n");
    else
        printf("타겟 저장 인덱스: %d \n", idx);

    return 0;
}
```

빅-오에 대한 수학적 접근1

👉 빅-오의 수학적 정의

두 개의 함수 $f(n)$ 과 $g(n)$ 이 주어졌을 때, 모든 $n \geq K$ 에 대하여 $f(n) \leq Cg(n)$ 을 만족하는 두 개의 상수 C 와 K 가 존재하면, $f(n)$ 의 빅-오는 $O(g(n))$ 이다.



- n^2 은 n^3 보다 클수 없다.
- 즉, $2,000n^2$, $1000n^2+1,000$ 뭐가 됐든 n^2 의 패턴을 벗어나지 못한다.

K: 두 알고리즘을 이용해 그래프를 그릴 때 일정 횟수 이후의 그래프가 역전되는 지점..

C: $2,000n^2$ 의 2,000 $\rightarrow Cg(n)$

빅-오에 대한 수학적 접근1

👉 빅-오의 실질적인 의미

$n+1$ 의 빅-오는 (n) 이

다 $n+1$ 이 아무리 연산횟수의 증가율이 크다 한들 증가율의 패턴이 n 을 넘지 못한다!

빅-오에 대한 수학적 접근2

👉 빅-오 판별의 예

"두 개의 함수 $f(n)$ 과 $g(n)$ 이 주어졌을 때"

➡ 두 개의 함수 $f(n)=5n^2+100$, $g(n)=n^2$ 이 주어졌을 때

"모든 $n \geq K$ 에 대하여"

➡ 모든 $n \geq 12$ 에 대하여,

n 의 값이 점차 증가하여 어느 순간 이후부터

$f(n) \leq Cg(n)$ 을 만족하게 하는 두 개의 상수 C 와 K 가 존재한다면,

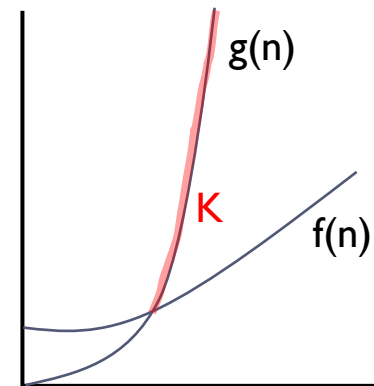
" $f(n) \leq Cg(n)$ 을 만족하는 두 개의 상수 C 와 K 가 존재하면"

➡ $5n^2+100 \leq 3500n^2$ 을 만족하는 $3500(C)$ 과 $12(K)$ 가 존재하니,

C는 임의로 붙여 보면...

" $f(n)$ 의 빅-오는 $O(g(n))$ 이다."

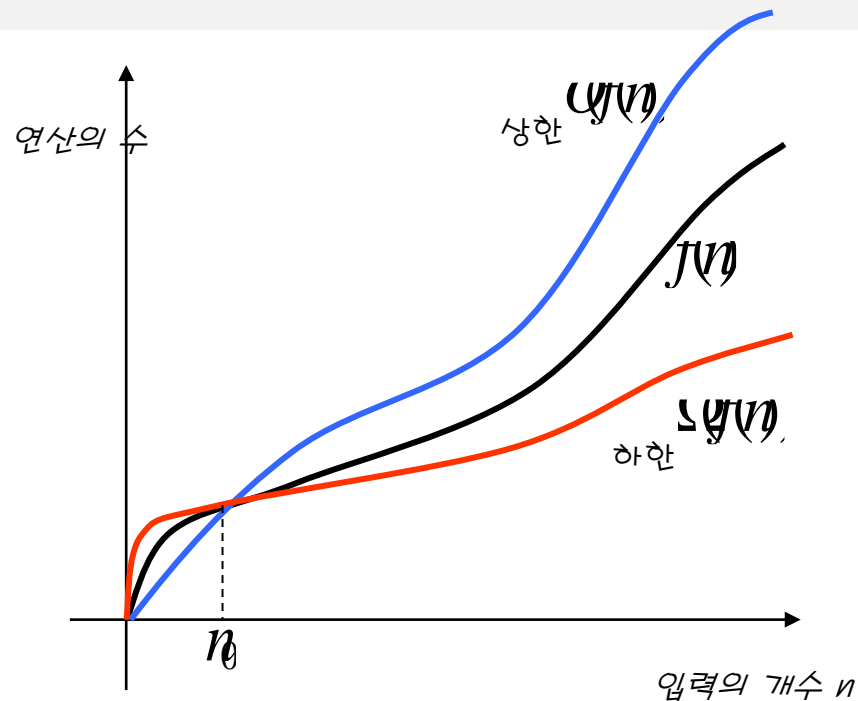
➡ $5n^2+100$ 의 빅-오는 $O(n^2)$ 이다.



빅오 표기법 이외의 표기법

■ 빅오메가 표기법

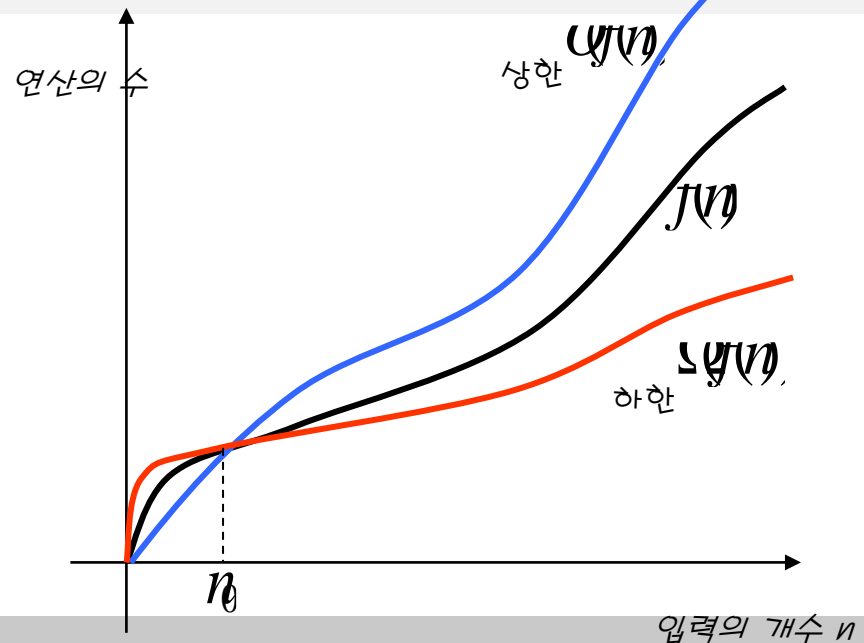
- 모든 $n \geq n_0$ 에 대하여 $|f(n)| \geq c|g(n)|$ 을 만족하는 2개의 상수 c 와 n_0 가 존재하면 $f(n) = \Omega(g(n))$ 이다.
- 빅오메가는 함수의 하한을 표시한다.
- (예) $n \geq 5$ 이면 $2n+1 < 10n$ 이므로 $n = \Omega(n)$



빅오 표기법 이외의 표기법

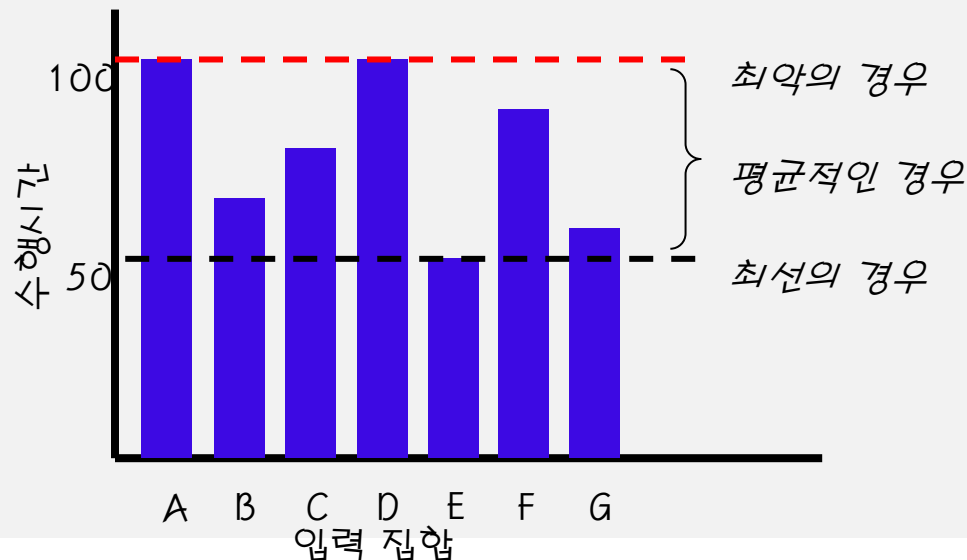
■ 빅세타 표기법

- 모든 $n \geq n_0$ 에 대하여 $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$ 을 만족하는 3개의 상수 c_1 , c_2 와 n_0 가 존재하면 $f(n) = \theta(g(n))$ 이다.
- 빅세타는 함수의 하한인 동시에 상한을 표시한다.
- $f(n) = O(g(n))$ 이면서 $f(n) = \Omega(g(n))$ 이면 $f(n) = \theta(n)$ 이다.
- (예) $n \geq 1$ 이면 $n \leq 2n+1 \leq 3n$ 이므로 $2n+1 = \theta(n)$



최선, 평균, 최악의 경우

- 알고리즘의 수행시간은 입력 자료 집합에 따라 다를 수 있다.
(예) 정렬 알고리즘의 수행 시간은 입력 집합에 따라 다를 수 있다.
- 최선의 경우(best case):** 수행 시간이 가장 빠른 경우
- 평균의 경우(average case):** 수행 시간이 평균적인 경우
- 최악의 경우(worst case):** 수행 시간이 가장 늦은 경우



최선, 평균, 최악의 경우

- 최선의 경우: 의미가 없는 경우가 많다.
- 평균적인 경우: 계산하기가 상당히 어려움.
- 최악의 경우: 가장 널리 사용된다. 계산하기 쉽고 응용에 따라서 중요한 의미를 가질 수도 있다.

(예) 비행기 관제업무, 게임, 로봇틱스

최선, 평균, 최악의 경우

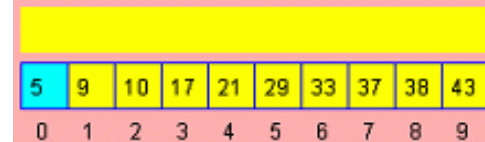
- (예) 순차탐색
- **최선의 경우**: 찾고자 하는 숫자가 맨앞에 있는 경우
 $\therefore O(1)$
- **최악의 경우**: 찾고자 하는 숫자가 맨뒤에 있는 경우
 $\therefore O(n)$
- **평균적인 경우**: 각 요소들이 균일하게 탐색된다고 가정하면

$$(1+2+\dots+n)/n=(n+1)/2$$

$$\therefore O(n)$$

인덱스 0에서 값 5 발견

숫자 비교 횟수 = 1



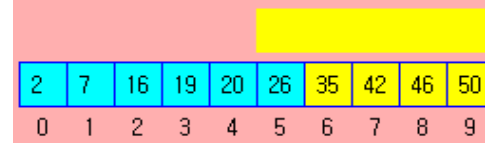
인덱스 9에서 값 43 발견

숫자 비교 횟수 = 10



인덱스 5에서 값 26 발견

숫자 비교 횟수 = 6



자료 구조의 C언어 표현방법

- 자료구조와 관련된 데이터들을 구조체로 정의
- 연산을 호출할 경우, 이 구조체를 함수의 파라미터로 전달
(예)

```
// 자료구조 스택과 관련된 자료들을 정의
typedef int element;
typedef struct {
    int top;
    element stack[MAX_STACK_SIZE];
} StackType;
```

자료구조의 요소

관련된 데이터를 구조체
로 정의

```
// 자료구조 스택과 관련된 연산들을 정의
void push(StackType *s, element item)
{
    if( s->top >= (MAX_STACK_SIZE - 1)){
        stack_full();
        return;
    }
    s->stack[++(s->top)] = item;
}
```

연산을 호출할때 구조체를
함수의 파라미터로 전달

자료구조 기술규칙

■ 상수

- 대문자로 표기
- (예) `#define MAX_ELEMENT 100`



■ 변수의 이름

- 소문자를 사용하였으며 언더라인을 사용하여 단어와 단어를 분리
- (예) `int increment;`
- `int new_node;`

■ 함수의 이름

- 동사를 이용하여 함수가 하는 작업을 표기
- (예) `int add(ListNode *node)` // 혼동이 없는 경우
- `int list_add(ListNode *node)` // 혼동이 생길 우려가 있는 경우

자료구조 기술규칙

- **typedef의 사용**
- C언어에서 사용자 정의 데이터 타입을 만드는 경우에 쓰이는 키워드

```
typedef <새로운 타입의 정의> <새로운 타입 이름>;
```

(예)

```
typedef int element;
typedef struct ListNode {
    element data;
    struct ListNode *link;
} ListNode;
```

