1 장. 데이터 구조와 알고리즘

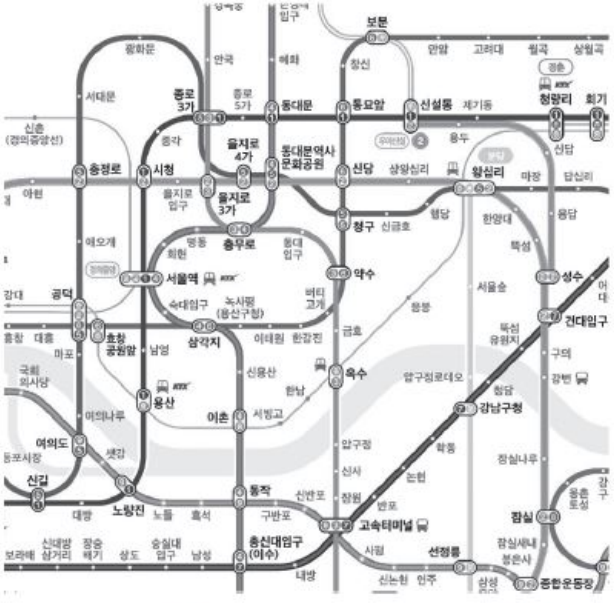
9/3/24

생활 속의 다양한 사물과 관련 자료구조

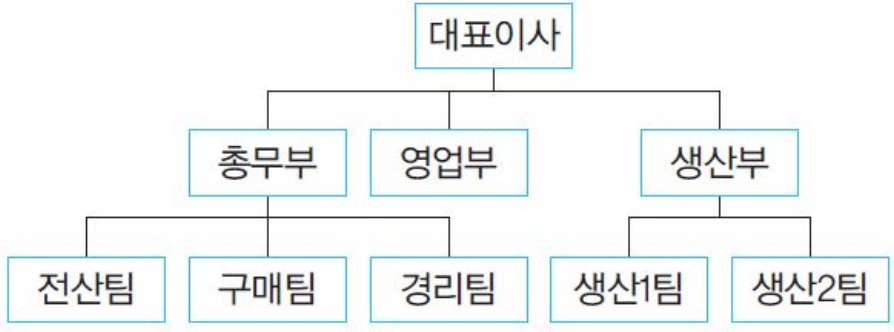
배달할 선물 목록: 리스트(List)

접시나 물건 쌓기: 스택(Stack)

매표소 줄서기: 큐(Queue)



지하철 노선도: 그래프(Graph)



직장의 조직도: 트리(Tree)

[그림 1.1] 생활 속의 다양한 사물과 관련 자료구조의 예

자료구조란?

- 자료구조 (Data Structure): 컴퓨터에서 자료를 정리하고 조직화하는 다양한 구조

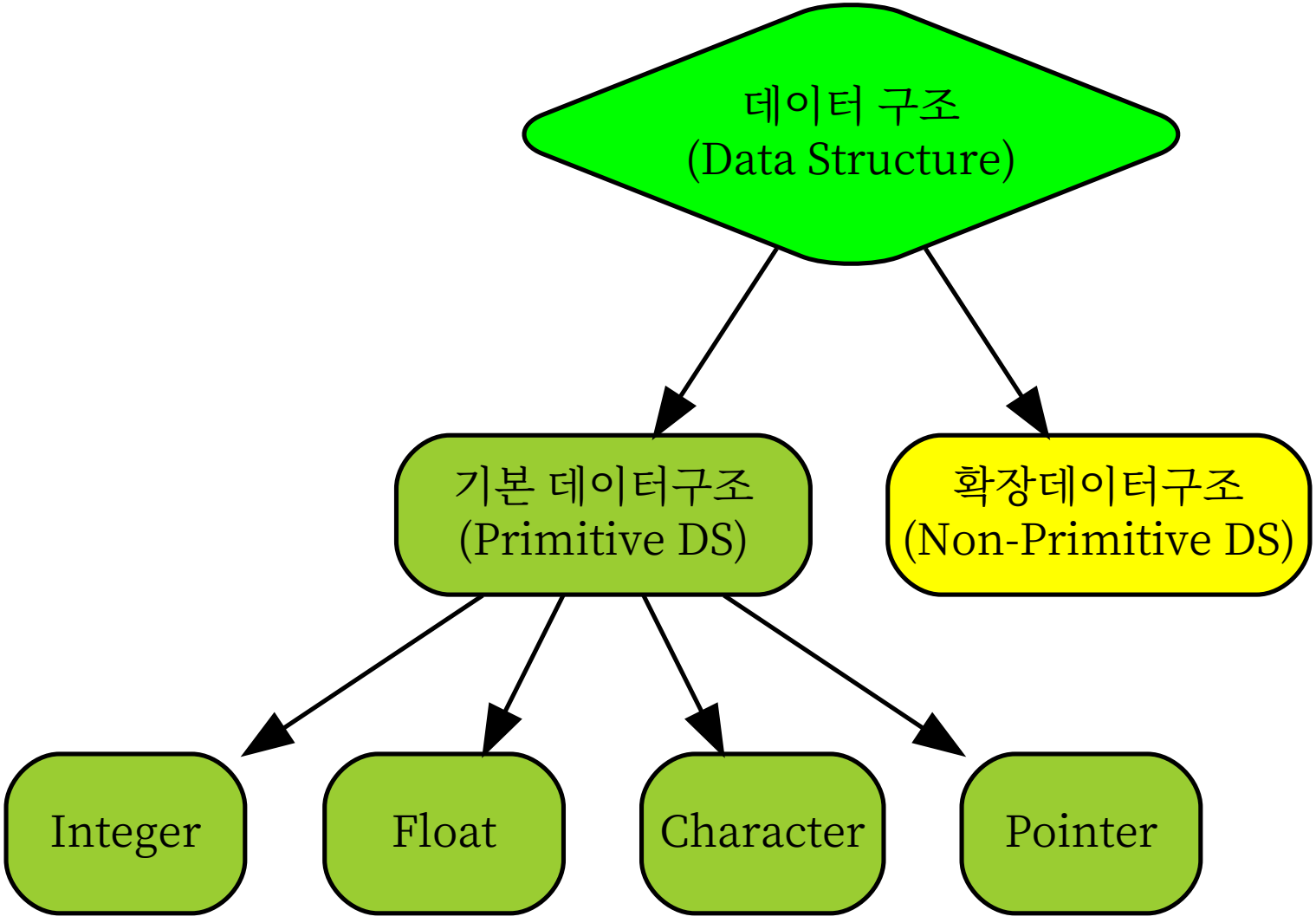
[표 1.1] 일상생활의 자료 정리 방법과 컴퓨터 자료구조의 비교

일상생활의 정리 방법	컴퓨터에서의 자료구조
배달할 선물 목록	리스트
접시나 물건을 쌓아서 보관 하는 것	스택
맛집의 줄서기	큐
직장의 조직도	트리
지하철 노선도	그래프

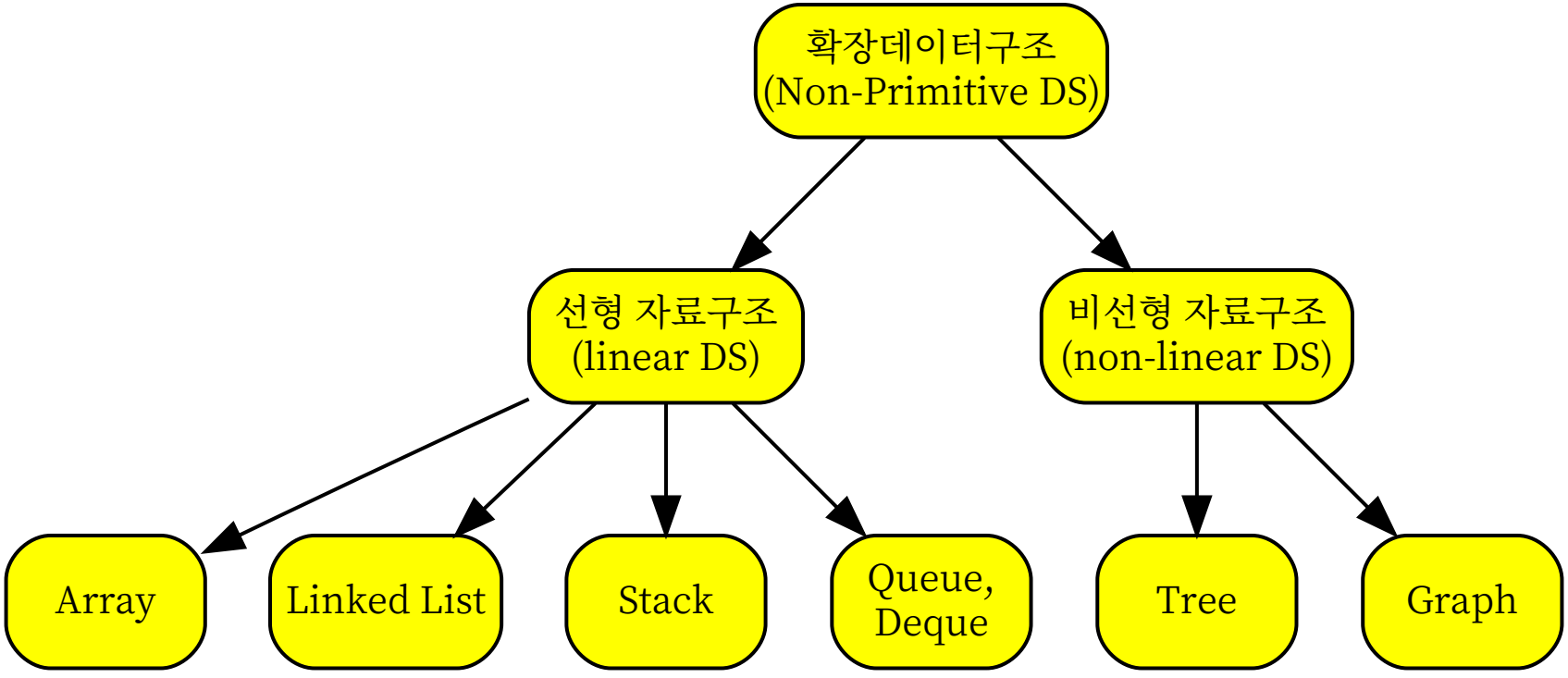
자료구조

- 데이터 구조: 데이터에 효율적으로 접근하기 위한 데이터의 구성과 저장 형식
  - 데이터 값
  - 데이터간의 관계
  - 적용 함수와 연산

자료구조의 분류

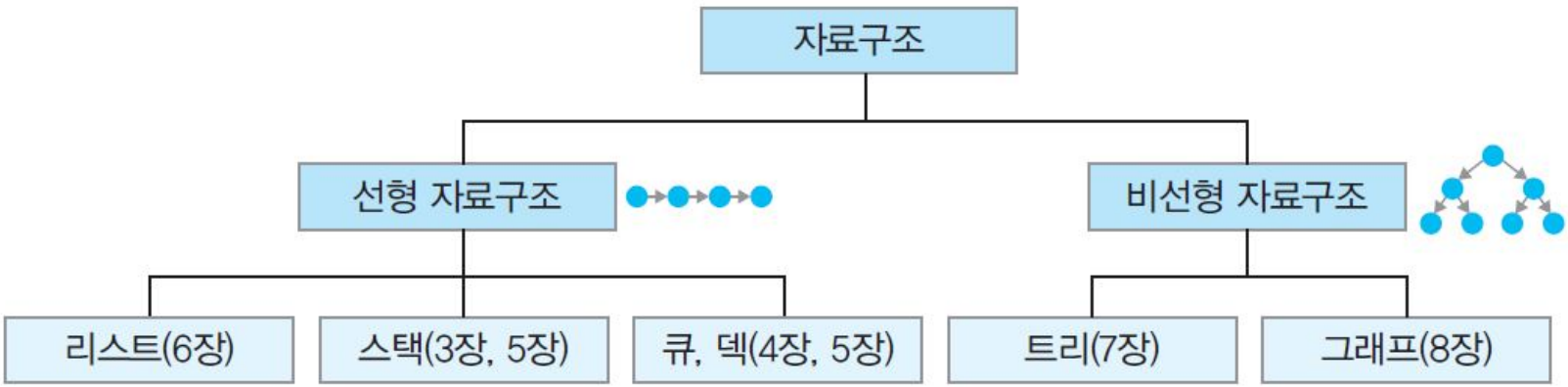

데이터 구조
(Data Structure)
기본 데이터구조
(Primitive DS)
확장데이터구조
(Non-Primitive DS)
Integer
Float
Character
Pointer

확장데이터구조


확장데이터구조
(Non-Primitive DS)
선형 자료구조
(linear DS)
비선형 자료구조
(non-linear DS)
Array
Linked List
Stack
Queue,
Deque
Tree
Graph

자료구조의 분류

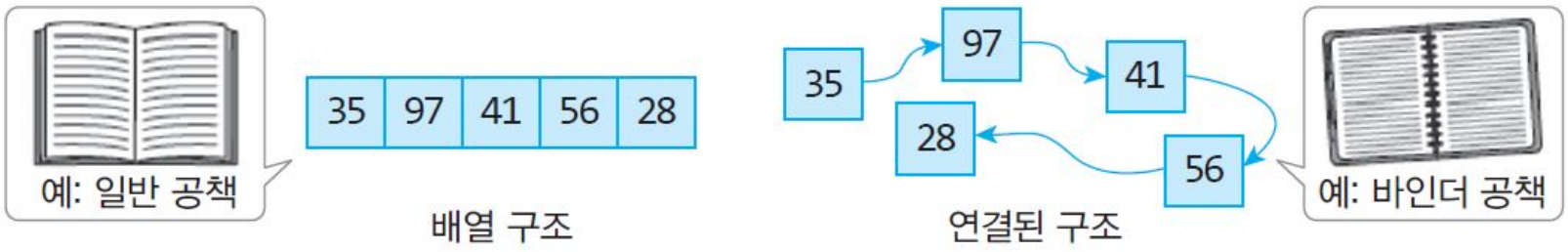
1. 선형 자료구조 (linear data structure)
   - 자료를 일렬로 나열할 수 있는 구조
   - 자료들 사이에는 반드시 순서
   - 예: 스택, 큐, 덱, 리스트 등
2. 비선형 자료구조 (non-linear data structure)
   - 한 줄로 나열하기 어려운 복잡한 관계의 자료들을 표현
   - 예: 트리, 그래프, 집합 등



[그림 1.2] 자료구조의 분류와 이 책의 구성

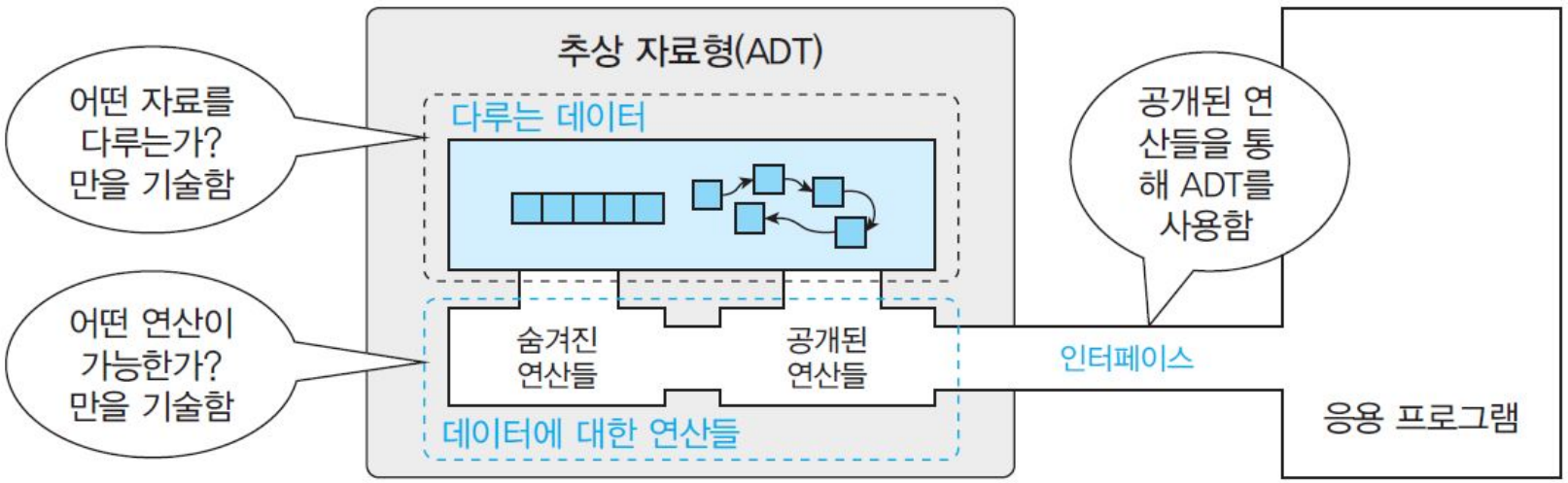
자료구조의 구현 방법

1. 배열 구조: 모든 자료가 인접한 메모리 공간에 저장 (3 장, 4 장)
2. 연결 구조: 흩어져 있는 자료들을 연결하여 하나로 관리 (5 장)



추상 자료형 (ADT: Abstract Data Type)

- 추상 자료형의 구조와 응용 프로그램과의 관계



[그림 1.5] 추상 자료형의 구조와 응용 프로그램과의 관계

추상 자료형 (ADT: Abstract Data Type)

- 구체적인 데이터 구조를 구현하기 위해 사용자의 관점에서 가능한 값과 연산을 정의
  - 자료형 (data type)
    - C 언어의 자료형: char, int, float, double 등
    - 새로운 자료형 정의: arrays, linked lists, hash tables, queue, tree, graph, ...
  - 추상 자료형 (ADT: Abstract Data Type)
    - 데이터 타입을 추상적 (수학적) 으로 정의한 것
    - 자료형에서 필수적이고 중요한 특징만 골라서 단순화 (데이터, 함수)
    - List, Stack, Queue, Priority Queue, Tree, Heap, Set, Map, Graph
- 추상 자료형 (ADT) 은 새로운 데이터 구조를 구현 (implementation) 하기 전 데이터 구조에 대한 상세 설명 (specification)

예제: 자연수의 추상 자료형

- **데이터:** 1~INT_MAX 까지의 정수의 부분 범위
- **연산:**
  1. 산술 연산 add(a,b), substract(a,b), ...
  2. 비교 연산 is_equal(a,b), is_greater(a,b), is_zero(a) 등

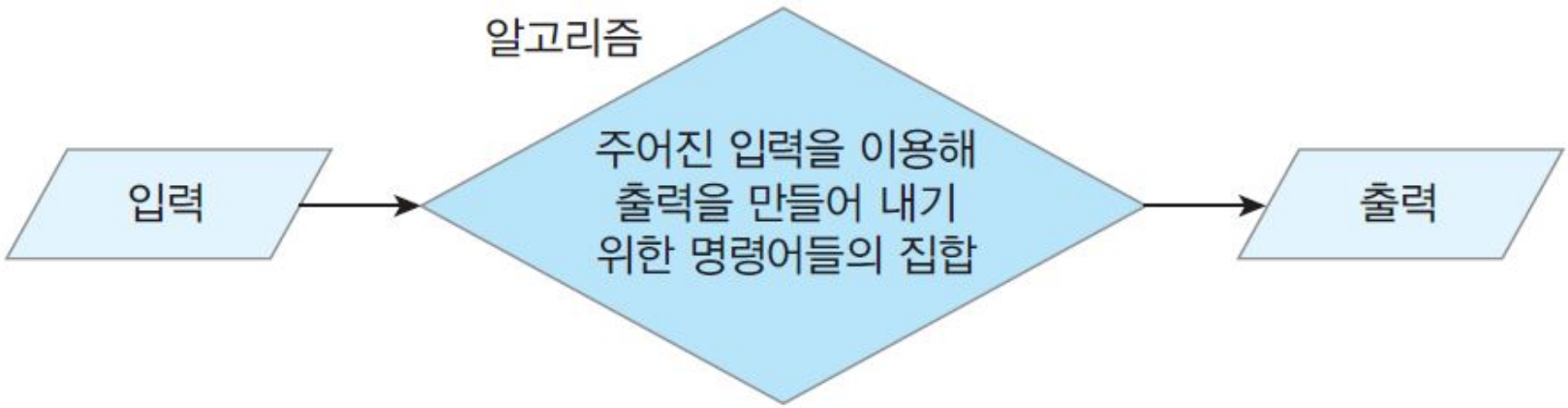
예제: 다항식의 추상 자료형

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0.$$

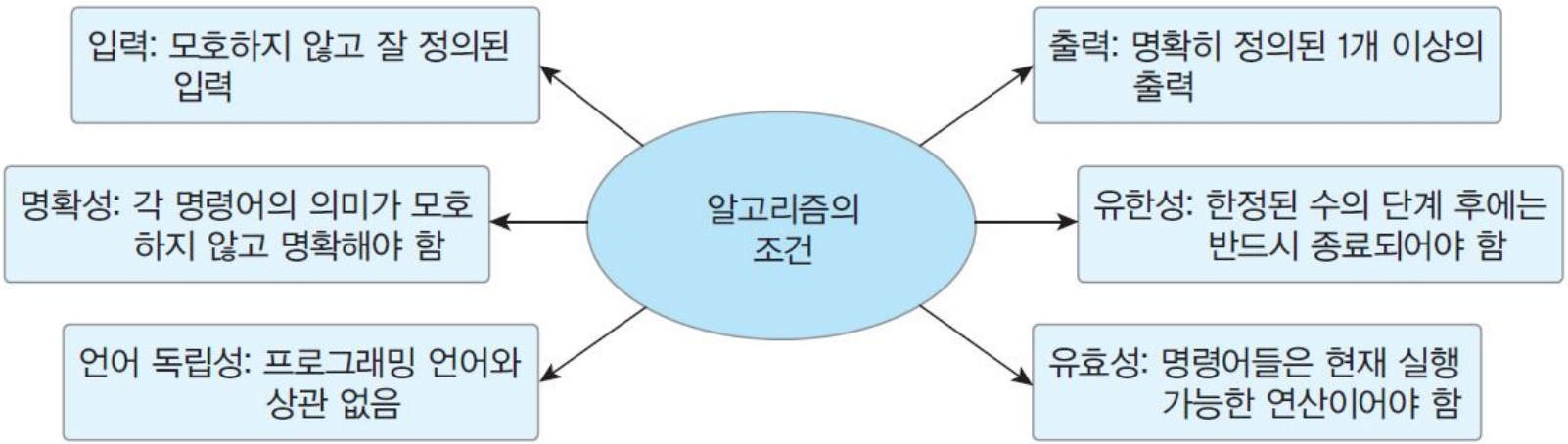
- **데이터**: (지수, 계수) 의 순서쌍의 집합: (i, a_i), $i = \{0, 1, \ldots, n\}$
- **연산**
  1. 상태를 검사
     - 다항식의 차수 (degree) 를 반환: `degree()`
     - 지수가 i 인 항의 계수를 반환하는 연산: `coefficient(i)`
  2. 다항식 계산
     - 미지수 (x) 에 특정값을 대입할 때, 다항식의 결과를 반환
     - `evaluate(x)`
  3. 두 다항식간의 연산들
     - 두 다항식의 합, 차, 곱 등
     - `add(B)`, `sub(B)`, `muliply(B)`

알고리즘이란?

- 주어진 문제를 해결하기 위한 단계적인 절차 (계산절차 또는 처리과정)
- 프로그래밍 언어와 상관없이 문제 해결 절차를 나타내는 명령어의 집합



알고리즘의 조건



[그림 1.3] 알고리즘의 조건

알고리즘의 기술 방법

- 영어나 한국어와 같은 자연어
- 흐름도 (flow chart)
- 유사 코드 (pseudo-code)
- 특정한 프로그래밍 언어 (C 언어, C++, java 등)

3 개의 숫자에서 가장 큰 수를 찾는 알고리즘

1. 자연어 표현:

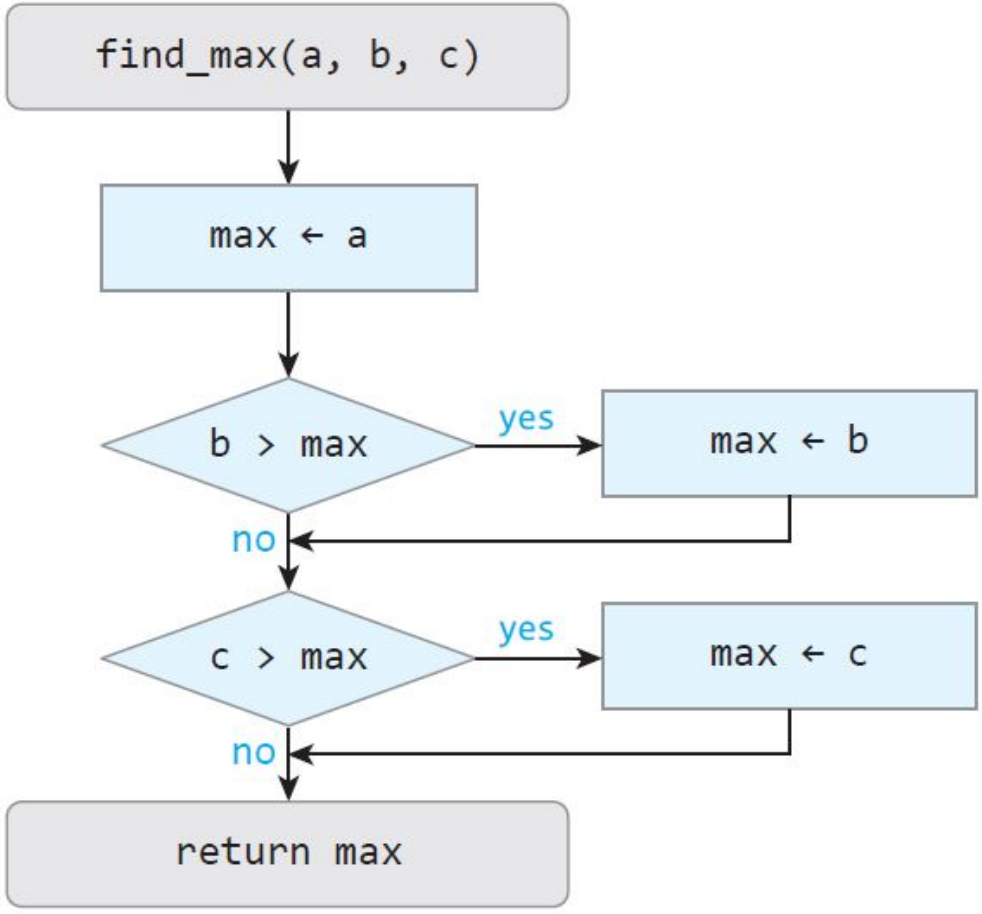
- 표현이 자유롭고 편리하지만 자칫 문장의 의미가 애매해질 수 있음

알고리즘 1.1 세 개의 숫자에서 최댓값을 찾는 알고리즘(자연어 표현)

```
01  find_max( a, b, c )
02      a를 최댓값을 저장하는 변수 max에 복사한다.
03      만약 b가 max보다 크면 b를 max에 복사한다.
04      만약 c가 max보다 크면 c를 max에 복사한다.
05      max를 반환한다.
```

2. 흐름도 (flowchart) 표현

- 직관적이고 이해하기 쉽지만, 복잡한 알고리즘의 경우, 상당히 복잡해짐



[그림 1.4] 흐름도로 표시한 알고리즘 예(세 개의 수에서 최댓값 찾기)

3. 유사 코드 (pseudo code) 표현

- 자연어보다는 체계적이지만 프로그래밍 언어보다는 덜 엄격
- 프로그래밍 언어에서 발생하는 많은 불필요한 표현을 생략
- 알고리즘의 핵심적인 내용에만 집중할 수 있음

알고리즘 1.2 세 개의 숫자에서 최댓값을 찾는 알고리즘(유사 코드 표현)

```
01  find_max( a, b, c )
02      max ← a                 // ← 는 대입 연산을 의미함
03      if b > max :            // :는 새로운 블록의 시작을 나타냄
04          max ← b             // 새로운 블록은 들여쓰기로 표시함
05      if c > max :
06          max ← c
07      return max
```

4. 특정 프로그래밍 언어 표현

- 바로 실행해 볼 수 있음
- 실제 구현시의 많은 구체적인 사항들이 알고리즘의 핵심적인 내용들의 이해를 방해할 수 있음

```
int find_max(int a, int b, int c){
  int max = a;
  if(b > max) max = b;
  if(c > max) max = c;
  return(max);
}
```

알고리즘의 성능 분석

- 효율적인 알고리즘이란?
  - 실행 시간이 짧으면서 컴퓨터의 자원들을 적게 사용
  - 시간 효율성, 공간 효율성

입력자료의 수	프로그램 A	프로그램 B
6	36 sec	64 sec
100	10,000 sec	2^{100} sec =

- 알고리즘의 성능 분석 기법
  1. 실행 시간을 측정
  2. 알고리즘 복잡도 분석 (complexity analysis): 연산횟수를 개략적으로 계산

알고리즘의 성능 분석 (1) 실행시간 측정

- 가장 단순하지만 확실한 방법
  - 실행시간 = 종료 시각 – 현재 시각
  - 시각 측정: C 언어에서 clock() 함수
    - 시스템의 현재 시각 반환 (CLOCKS_PER_SEC 단위)
- 문제점
  - 반드시 구현해야 함
  - 같은 조건의 하드웨어, 소프트웨어 환경 사용
  - 동일한 프로그래밍 언어 (컴파일 방식/ 인터프리트 방식)
  - 성능 평가를 한 데이터에 대해서만 유효

예제: 1 부터 1 억까지의 합을 구하는 데 걸리는 시간 측정

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

int calc_sum(int n){
    int i, sum = 0;
    for (i = 1; i < n; i++)
        sum = sum + i;
    return sum;
}
```

예제: 1 부터 1 억까지의 합 (conti.)

```c
int main(){
    clock_t start, finish;
    double  duration;

    printf("1 부터 10 까지의 합은 %d 입니다.\n", calc_sum(10));

    start = clock();         // 시작 시각
    calc_sum(100000000);// 실행시간을 측정코드를 넣는 부분
    finish = clock();        // 종료 시각

    duration = (double)(finish - start) / CLOCKS_PER_SEC;
    // CLOCKS_PER_SEC 초당 클락수
    printf("1 부터 1 억까지의 합을 구하는데 걸리 시간: %f 초\n",
           duration);
    return 0;
}
```

예제: 1 부터 1 억까지의 합을 구하는 데 걸리는 시간 측정 (conti.)

```
(base) jskim@bigdata:~/www/lectures/DataStructure/c/ch01$ ll
total 44
drwxrwxr-x  2 jskim jskim  4096  8월 25 17:27 ./
drwxrwxr-x 12 jskim jskim  4096  8월 24 16:34 ../
-rwxrw-r--  1 jskim jskim   888  8월 25 16:18 elapsed.c*
-rwxr-xr-x  1 jskim jskim 16736  8월 24 16:35 find_max*
-rwxrw-r--  1 jskim jskim   416  8월 24 16:48 find_max.c*
-rwxrw-r--  1 jskim jskim   899  8월 24 14:49 has_duplicate_elem.c*
-rwxrw-r--  1 jskim jskim   820  8월 24 14:49 sequential_search.c*
(base) jskim@bigdata:~/www/lectures/DataStructure/c/ch01$ gcc elapsed.c -o test
(base) jskim@bigdata:~/www/lectures/DataStructure/c/ch01$ ./test
1부터 10까지의 합은 45입니다.
1부터 1억까지의 합을 구하는데 걸리 시간: 0.336890 초
(base) jskim@bigdata:~/www/lectures/DataStructure/c/ch01$
```

알고리즘 복잡성 **(big O)**

함수 $f(x)$ 에 대하여, 아래의 식을 만족하는 양의 실수 M 과 x_0 가 존재하면

$$|f(x)| \leq Mg(x) \quad \text{for all } x \geq x_0.$$

$$f(x) = O(g(x)).$$

로 표기하고, 여기서 O 를 **big-O** 라고 부른다

- 예제

$$f(n) = 8n^3 + 5n^2 + 7, n \geq 1$$

- 해답

$$f(n) = 8n^3 + 5n^2 + 7 \leq 8n^3 + 5n^3 + 7n^3 = 19n^3$$

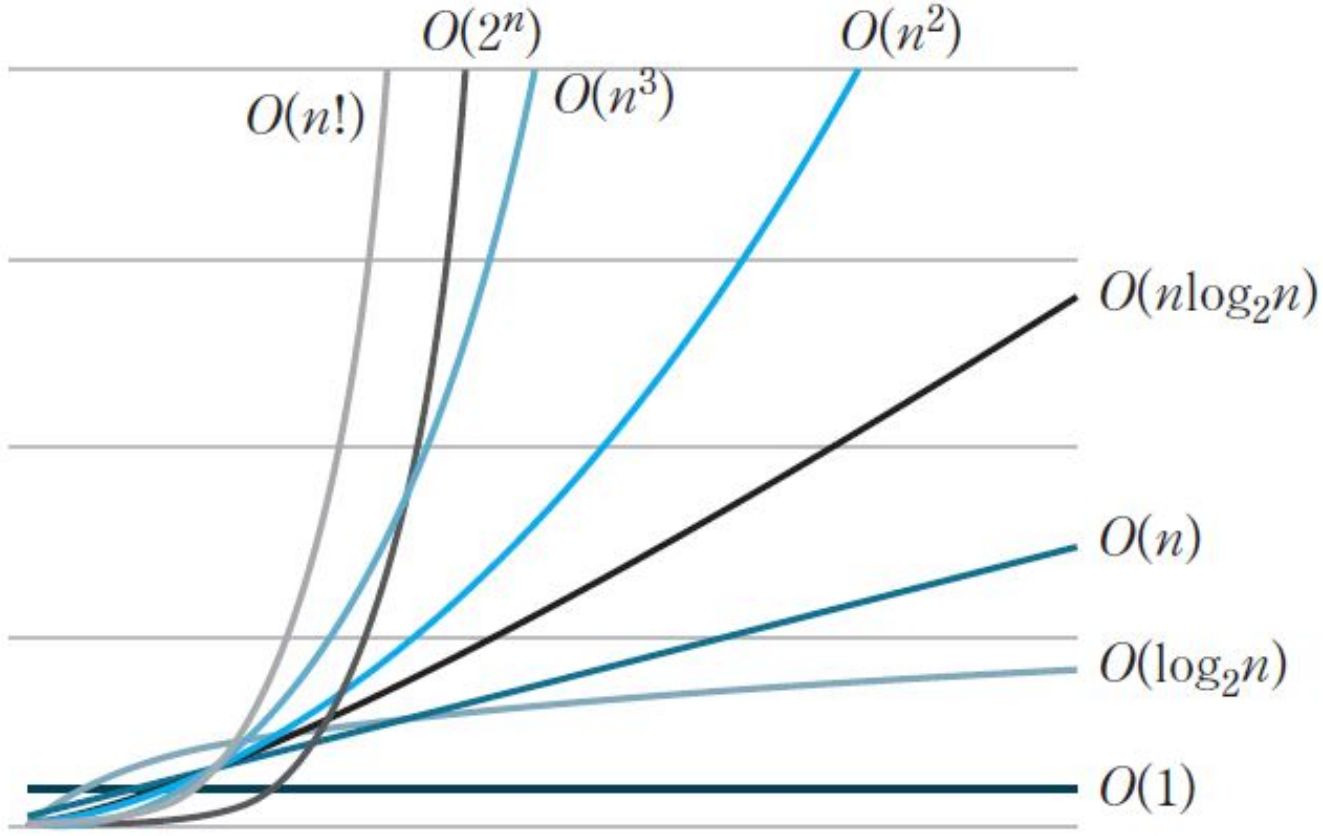
따라서 $f(n) = O(n^3)$

알고리즘 복잡도의 표기법에 따른 이름

이름	표기법
Constant(상수)	$O(1)$
Logarithmic time(로그)	$O(\log_2 n)$
Linear time (선형)	$O(n)$
linearithmic time(선형 로그)	$O(n \log_2 n)$
Quadratic time (2 차)	$O(n^2)$
Exponential time(지수)	$O(2^n)$
Factorial time(팩토리얼)	$O(n!)$

알고리즘 복잡성의 순서

$$O(1) < O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < \\ < O(n^3) < \ldots < O(2^n) < O(n!)$$



[그림 1.11] n이 증가함에 따른 시간 복잡도 함수들의 증가 속도

알고리즘 복잡성 **(big Omega)**

아래의 식을 만족하는 양의 실수 M 과 x_0 가 존재하면

$$|f(x)| \geq Mg(x) \quad \text{for all } x \geq x_0.$$

다음과 같이 표기한다. 여기서 Ω 를 **big-Omega** 라고 부른다

$$f(x) = \Omega(g(x)).$$

- 함수 $f(x) = 8x^3 + 5x^2 + 7$ 는

$$f(x) = \Omega(x^3).$$

- 해답

$$f(n) = 8n^3 + 5n^2 + 7 \geq 8n^3$$

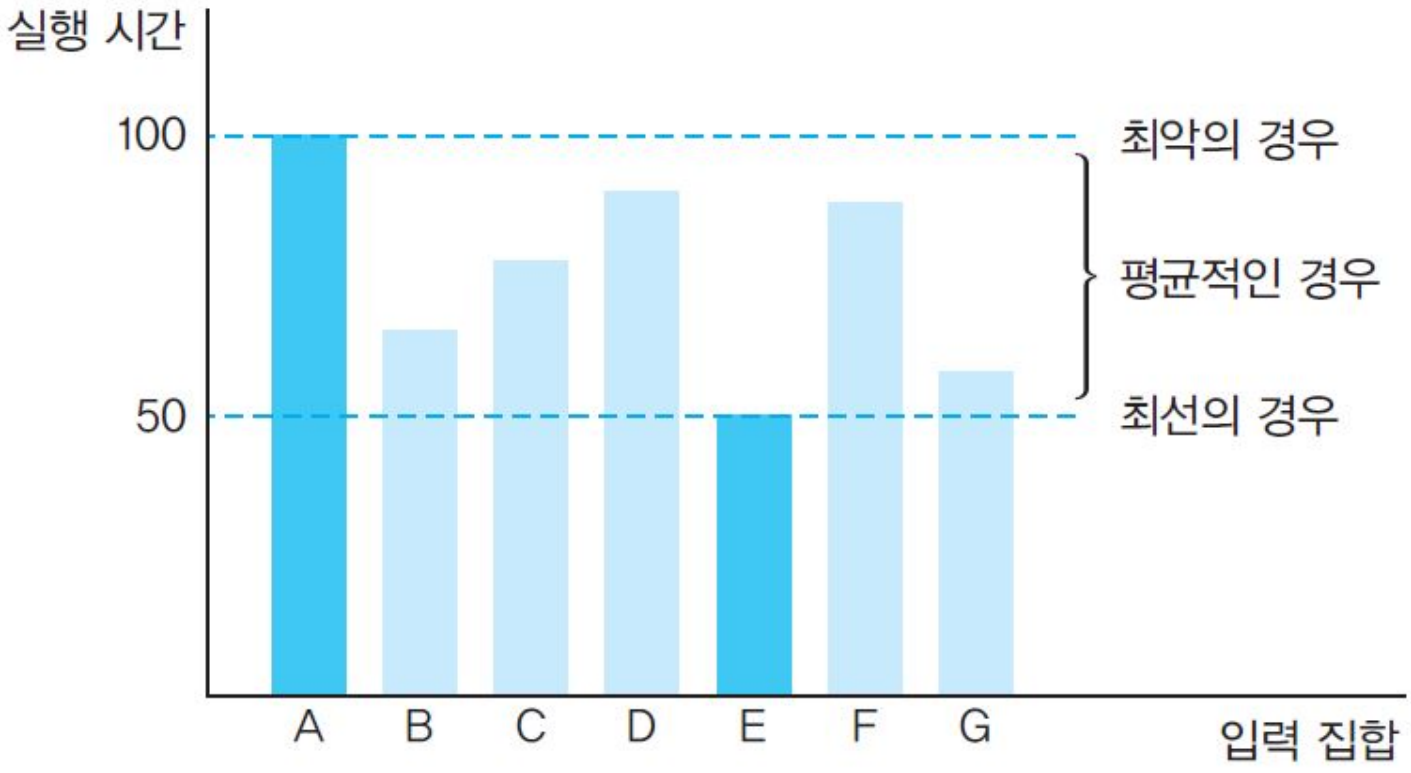
따라서 $f(n) = \Omega(n^3)$

연습문제

- 1.11, P1.17
- 각자 계산해 볼 것

알고리즘 분석에서 최선, 평균, 최악의 경우

- 동일한 알고리즘이라 할지라도 어떤 값을 입력하느냐에 따라 실행시간이 달라질 수 있음



[그림 1.12] 최선, 최악 및 평균적인 경우의 예

알고리즘 분석에서 최선, 평균, 최악의 경우 (순차 탐색)

```c
#include <stdio.h>
void main()
{
    int A[10] = { 5, 9, 10, 17, 21, 29, 33, 37, 38, 43 };
    int index;
    int key = 5;

    for (int i = 0; i<10; i++)
        if (A[i] == key)
                printf(" %d의 위치 = %d\n", key, index);
}
```

- key 가 어떤 값이냐에 따라 실행시간이 서로 다름
- 평균시간은?

$$\frac{1+2+\ldots+n}{n} = \frac{n+1}{2} = O(n).$$