

# 자료구조론

## 2장 소프트웨어와 자료구조

## □ 이 장에서 다룰 내용

- ❖ 소프트웨어 생명주기 ← 생략
- ❖ 추상 자료형
- ❖ 알고리즘
- ❖ 성능분석

## ❑ 추상 자료형

### ❖ 추상화(abstraction)

- 자세하고 복잡한 것 대신, 필수적인 중요한 특징만 골라 단순화 시킴

### ❖ 추상화와 구체화

- 추상화 – “무엇(what)인가?”를 논리적으로 정의
- 구체화 – “어떻게(how) 할 것인가?”를 실제로 표현

### ❖ 컴퓨터에서 문제를 해결할 때도 추상화 작업을 적용

- 크고 복잡한 문제를 단순화시켜 좀더 쉽게 해결하는 방법을 찾음

### ❖ 자료형(Data Type)

- 처리할 데이터의 집합과 데이터에 대해 수행할 수 있는 연산자의 집합
- 예) 정수 자료형
  - 데이터는 {..., -1, 0, 1, ...}
  - 연산자는 {+, -, x, ÷, mod}

### ❖ 추상 자료형(ADT: Abstract Data Type)

- 데이터와 연산자의 특성을 논리적으로 추상화하여 정의한 자료형
  - 구현을 포함하지 않음

## □ 추상 자료형

### ❖ 추상 자료형(ADT: Abstract Data Type)

- 예) 스택 추상자료형(7장에서 다룸)

#### ADT Stack

데이터 : 0개 이상의 원소를 가진 유한 순서 리스트

연산 :  $S \in \text{Stack}; \text{item} \in \text{Element};$

**createStack() ::= create an empty Stack;**

// 공백 스택을 생성하는 연산

**isEmpty(S) ::= if (S is empty) then return true  
                  else return false;**

// 스택 S가 공백인지 아닌지를 확인하는 연산

**push(S, item) ::= insert item onto the top of S;**

// 스택 S의 top에 item(원소)을 삽입하는 연산

**pop(S) ::= if (isEmpty(S)) then return error**

**else { delete and return the top item of S };**

// 스택 S의 top에 있는 item(원소)을 스택에서 삭제하고 반환하는 연산

...

**End Stack**

## ❖ 알고리즘(algorithm)이란

- 주어진 문제의 해결 절차를 체계적으로 기술한 것


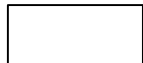
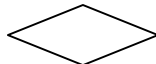


## ❖ 알고리즘의 표현 방법

- 자연어를 이용한 서술적 표현 방법
- 프로그래밍 언어를 이용한 구체화 방법
- 순서도(flow chart)를 이용한 도식화 표현 방법 ←
- 가상코드(pseudo-code)를 이용한 추상화 방법 ←

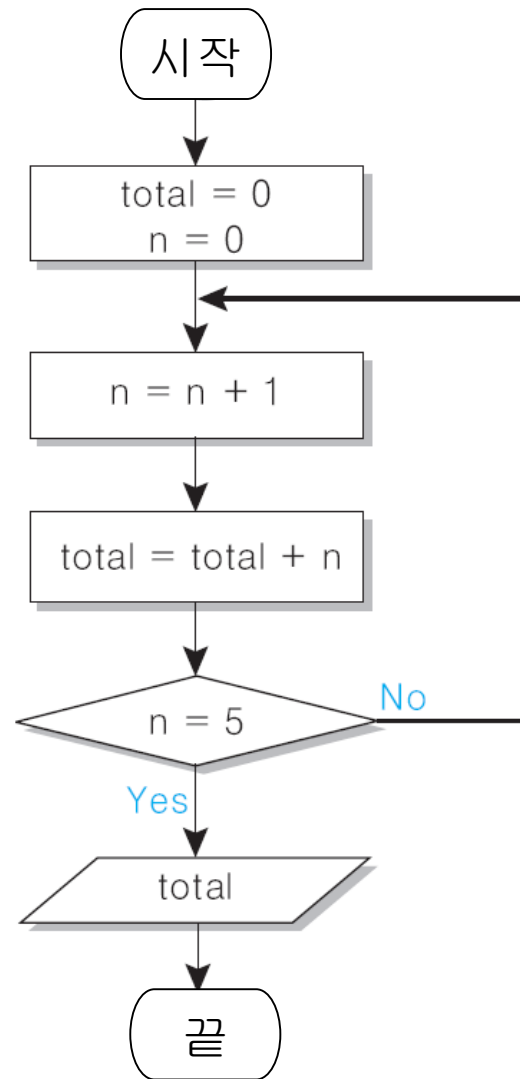
## ❖ 순서도를 이용한 알고리즘의 표현

- 예) 1부터 5까지의 합을 구하는 순서도

- 순서도에서 사용하는 기호

- 연결자 ○
- 시작, 종료 
- 처리문, 치환문 
- 조건문, 판단문 
- 입력, 출력 
- 제어의 흐름 

- 장점 : 알고리즘의 흐름 파악이 용이함
- 단점 : 복잡한 알고리즘의 표현이 어려움



## ❖ 가상코드를 이용한 알고리즘의 표현

- 가상코드를 사용하여 프로그래밍 언어와 유사하게 알고리즘을 표현
- 특정 프로그래밍 언어가 아니므로 직접 실행은 불가능
- 원하는 특정 프로그래밍 언어로의 변환 용이
- 예) 1부터 5까지의 합을 구하여 반환하는 알고리즘

```
sum()
{
    total ← 0 ;
    for (n←1 ; n ≤ 5; n←n+1) do {
        total ← total + n ;
    }
    return total;
}
```

## ❖ 알고리즘 분석

- 하나의 문제에 대해 알고리즘은 여러 개일 수 있으므로, 그 중 가장 효율적이고 사용 환경에 최적인 알고리즘을 결정하기 위해서는 알고리즘 분석이 필요하다.
- 분석 기준
  - 정확성 : 올바른 입력이 들어왔을 때 정해진 시간 내에 올바른 결과를 출력하는가
  - 명확성 : 알고리즘의 표현이 이해하기 쉽게 명확한가
  - 수행량 : 알고리즘 특성을 나타내는 중요 연산이 얼마나 여러 번 수행되는가
  - 메모리 사용량 : 알고리즘에 의해 사용되는 메모리 양이 어느 정도인가
  - 최적성 : 알고리즘을 적용할 시스템의 사용 환경에 적합한가



## ❖ 알고리즘의 성능 분석 방법

### ■ 공간 복잡도(space complexity)

- 알고리즘을 프로그램으로 실행하여 완료하는 데 필요한 총 저장 공간
- 고정 공간 + 가변 공간

### ■ 시간 복잡도(time complexity)

- 알고리즘을 프로그램으로 실행하여 완료하는 데 걸리는 시간
- 같은 프로그램이라도 실행하는 컴퓨터 성능에 따라 달라지므로 실제 실행시간을 측정하는 것은 의미 적음
- 따라서 명령문의 실행 빈도수를 계산하여 실행 시간을 구함
- 실행 빈도수 계산 방법
  - 지정문, 조건문, 반환문, 반복문 내의 제어문 등도 하나의 단위시간을 갖는 기본 명령문으로 취급하여 빈도수를 계산

## ❖ 시간 복잡도 예

- 피보나치 수열 알고리즘의 각 라인 실행 빈도수 구하기
- 피보나치 수열 정의

$$f_0 = 0$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2} \quad (n \geq 2)$$

```
fibonacci(n)  
01  if (n<0) then  
02      stop ;  
03  if (n≤1) then  
04      return n ;  
05  f1 ← 0 ;  
06  f2 ← 1 ;  
07  for (i←2 ; i≤n ; i←i+1) do {  
08      fn←f1+f2 ;  
09      f1←f2 ;  
10      f2←fn ;  
11  }  
12  return fn ;  
13 end
```

## □ 성능분석

- $n < 0$ ,  $n = 0$ ,  $n = 1$ 인 경우 실행 빈도수
  - for 반복문이 수행되지 않기 때문에 실행 빈도수가 작다.

행 번호	$n < 0$	$n = 0$	$n = 1$
1	1	1	1
2	1	0	0
3	0	1	1
4	0	1	1
5~13	0	0	0

```
fibonacci(n)
01  if (n<0) then
02      stop ;
03  if (n≤1) then
04      return n ;
05  f1 ← 0 ;
06  f2 ← 1 ;
07  for (i←2 ; i≤n ; i←i+1) do {
08      fn←f1+f2 ;
09      f1←f2 ;
10      f2←fn ;
11  }
12  return fn ;
13 end
```

## □ 성능분석

- 일반적인 경우( $n > 1$ ) 실행 빈도수
  - $n$ 에 따라 for 반복문 수행

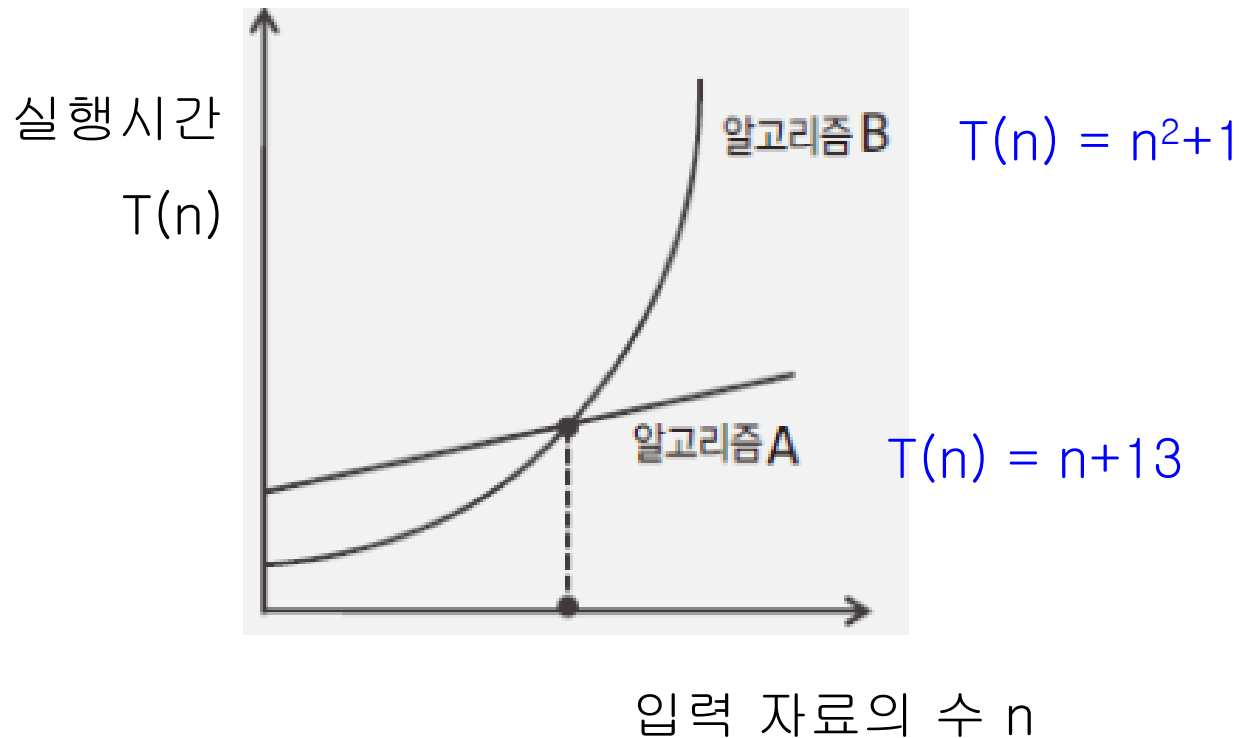
행 번호	실행 빈도수	행 번호	실행 빈도수
1	1	8	$n-1$
2	0	9	$n-1$
3	1	10	$n-1$
4	0	11	0
5	1	12	1
6	1	13	0
7	$n$		

- 총 실행 빈도수  
 $= 1+0+1+0+1+1+n+(n-1)+(n-1)+(n-1)+0+1+0 = 4n+2$   
즉, 문제의 크기가  $n$ 일 때 실행 시간  $T(n) = 4n+2$

```
fibonacci(n)
01  if (n<0) then
02      stop ;
03  if (n≤1) then
04      return n ;
05  f1 ← 0 ;
06  f2 ← 1 ;
07  for (i←2 ; i≤n ; i←i+1) do {
08      fn←f1+f2 ;
09      f1←f2 ;
10      f2←fn ;
11  }
12  return fn ;
13 end
```

## ❖ 서로 다른 알고리즘의 실행시간 비교

- 입력 자료의 수가 작을 때 알고리즘들을 비교하는 것은 큰 의미가 없다.
- 입력 자료의 수가 커질 때 실행시간이 증가하는 정도에 초점을 맞추어 알고리즘들을 비교한다.
- 자료 수(문제의 크기)가  $n$ 일 때 실행 시간을  $T(n)$ 이라고 하자.



- ❖ 시간 복잡도는  $O$ -notation (“빅-오” 표기법)으로 표현할 수 있다.
  - $O$ -notation 구하는 순서
    - ① 실행 빈도수를 구하여 입력의 크기  $n$ 에 대한 실행시간 함수  $T(n)$ 을 구함
    - ②  $T(n)$ 의 증가에 가장 큰 영향을 주는, 가장 차수가 높은  $n$ 에 대한 항을 선택
    - ③ 선택한 항에서 계수는 생략하고  $O$  기호의 오른쪽의 괄호 안에 표시
  - 예) 피보나치 수열 알고리즘의 시간 복잡도를 구하는 과정
    - ① 실행시간 함수 :  $T(n) = 4n+2$
    - ②  $n$ 에 대한 최고차 항을 선택 :  $4n$
    - ③ 계수 4는 생략하고  $O$  오른쪽의 괄호 안에 표시 :  $O(n)$즉, 피보나치 수열 알고리즘의 시간 복잡도 =  $O(n)$

## ❖ O-표기법 예

- 다음  $n$ 에 대한 함수  $f(n)$ 를 O-표기법으로 나타내시오.

$$f(n) = 3n^2 + 100n + 1000 =$$

$$f(n) = 5\log_2 n + 2n =$$

$$f(n) = 500 =$$

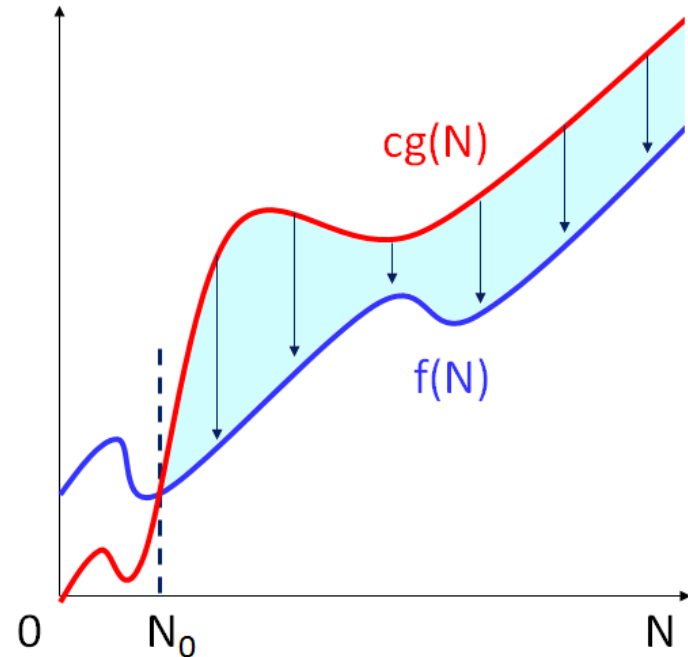
$$f(n) = 5n^3 + 5n^2 =$$

$$f(n) = 2n \log_2 n + 10n =$$

$$f(n) = 2n^2 \log_2 n + 10n^3 =$$

## ❖ O-표기법은 점근적 표기법(asymptotic notation)의 하나

- 점근적 표기법은 입력의 크기( $n$ )가 증가함에 따라 어느 정도 속도로 실행시간이 증가하는가(order of growth)를 표현
- 정의: 모든  $N \geq N_0$ 에 대해서  $f(N) \leq cg(N)$ 이 성립하는 양의 상수  $c$ 와  $N_0$ 가 존재하면,  $f(N) = O(g(N))$ 이다.



## ❖ 그 밖의 점근적 표기법

- $\Omega$  (Big-Omega)-표기법,  $\Theta$  (Theta)-표기법, ...
- 알고리즘의 수행시간은 주로 O-표기를 사용하며, 보다 정확히 표현하기 위해  $\Theta$ -표기를 사용하기도 한다.
- 자세한 내용은 알고리즘 수업에서 다룸



### ❖ 자주 사용되는 O-표기와 이름

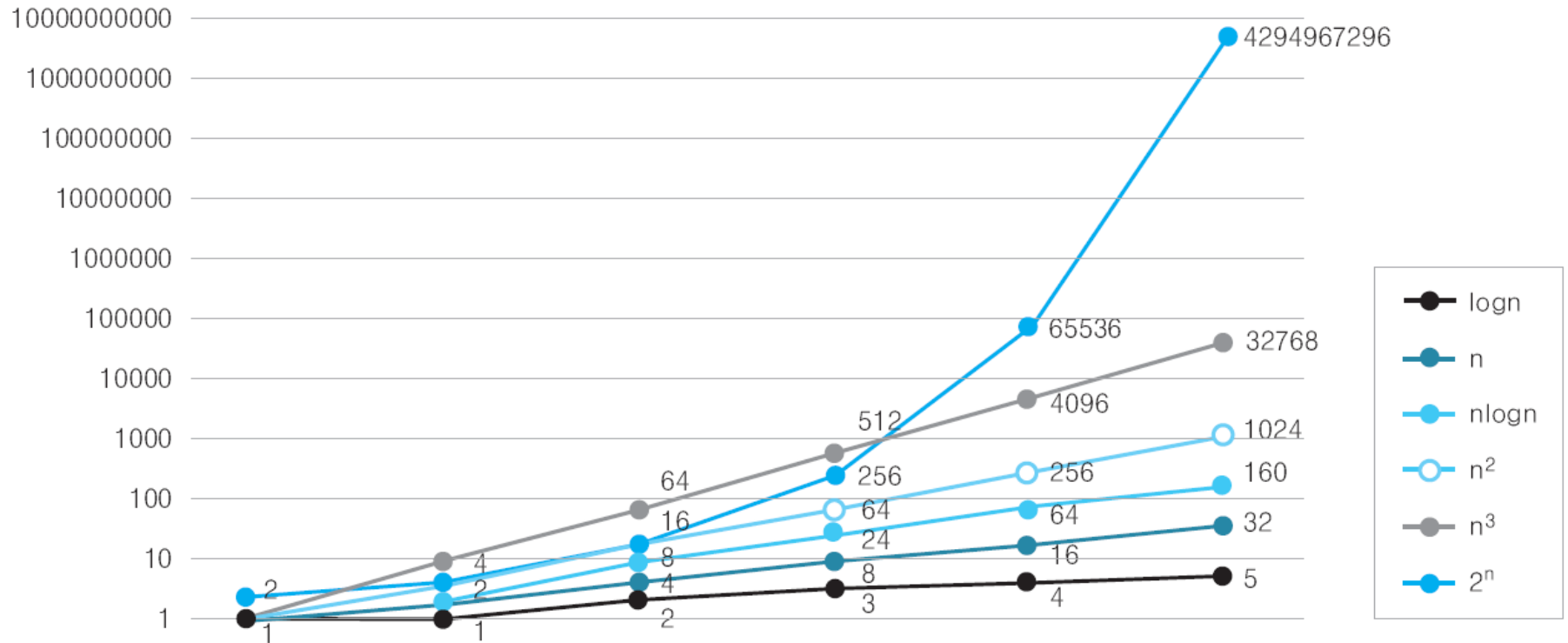
- $O(1)$  상수시간(Constant Time)
- $O(\log n)$  로그(대수)시간(Logarithmic Time)
- $O(n)$  선형시간(Linear Time)
- $O(n \log n)$  로그선형시간(Log-linear Time)
- $O(n^2)$  제곱시간(Quadratic Time)
- $O(n^3)$  세제곱시간(Cubic Time)
- $O(2^n)$  지수시간(Exponential Time)

## ❖ $n$ 값의 변화에 따른 각 실행시간 함수의 실행 빈도수 비교

1	<	$\log n$	<	$n$	<	$n \log n$	<	$n^2$	<	$n^3$	<	$2^n$
1		0		1		0		1		1		2
1		1		2		2		4		8		4
1		2		4		8		16		64		16
1		3		8		24		64		512		256
1		4		16		64		256		4096		65536
1		5		32		160		1024		32768		4294967296

## ❖ 알고리즘의 시간 복잡도를 낮은 것부터 나열하면 다음과 같다.

$O(1)$ ,  $O(\log n)$ ,  $O(n)$ ,  $O(n \log n)$ ,  $O(n^2)$ ,  $O(n^3)$ ,  $O(2^n)$ , ...



[그림 2-17]  $n$ 값에 대한 실행 시간 함수 그래프