<자신의 정리 자료>

〈1장, 자료구조와 알고리즘 (Data Structures and Algorithms), 김성현, 2019107 83〉

〈순서〉

1.1 자료구조와 알고리즘	1.2 추상자료형	1.3 알고리즘의 성능 분석
- 알고리즘 + 자료구조 = 프로그램	- 추상 데이터 타입 (ADT)	- 알고리즘의 성능 분석
- 알고리즘의 특징		- 빅오, 빅오메가, 빅세타 표기법
- 알고리즘의 기술 방법		- 최선, 평균, 최악 경우 분석.

1.1 자료구조와 알고리즘.

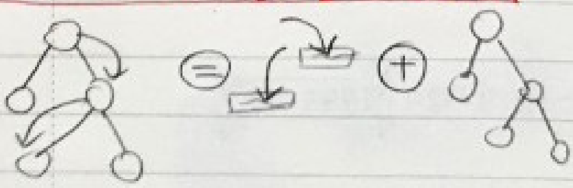
- 알고리즘 + 자료구조 = 프로그램.

(명령어들의 시퀀스) (명령어들)

- **알고리즘**: 어떤 문제를 해결하기 위한 잘 정의된 (well-defined) 유한 (finite) 단계의 컴퓨터명령어들
  - (유한 → 기본적으로 끝이 나야된다는 의미!!!)
- **자료구조**: 효율적으로 자료에 접근하기 (access) 위하여 자료를 조직하고 (organize), 관리하고 (manage), 저장하는 (store) 방법.

- 알고리즘 → 어떤 문제를 풀기 위하여 작성된 컴퓨터 명령어들, 문제를 해결하는 절차
- 자료구조 → 스택, 큐 등등
  - 스택: 자료들을 쌓아서 정리하는 구조, 맨위에서만 제거, 추가 가능
  - 큐: 마트 계산대, 먼저 도착한 놈만 먼저 빠져나간다~.

(컴퓨터) 프로그램 = 알고리즘 + 자료구조



- 알고리즘의 특징.

〈컴퓨터를 통한 문제해결〉
1. 문제의 해결방안 고안.
2. 해결방안에 따라 컴퓨터가 수행해야 할 단계적인 절차를 기술해 가는

⇒ 알고리즘

- **입력 및 출력**: 0개 이상의 입력 and 1개 이상의 출력.
  - (입력 없어도 되지만) (출력은 반드시 1개 이상)
- **명백성**: 알고리즘 구성하는 각 명령어의 의미는 명백.
  - (→ 정의하기 쉽고, 해석을 누구나 쉽게 하고,)
- **수행성**: 알고리즘 구성하는 각 명령어는 수행 가능
  - (실행 가능한 연산!)
- **유한성**: 알고리즘 수행 후 반드시 종료.
  - (유한 개 + 유한 시간 내 끝나야한다!!)

어떤 방식으로 적을 것이냐.

— 알고리즘의 기술 방법.

(입력) → • 자연어 (한국어, 영어 등) — 말로 적을 것이냐?

(그림이 과정) → • 흐름도 (flow chart) — 그림 같은 것으로 그릴 것이냐?

• 의사 코드 (pseudo-code) — code 까지는 아니지만, 수행이 되지는 않는 상태의 '코드'.

(가장 많이 사용) • 프로그래밍 언어 (C, Java 등).

(의사 코드) 논문 등에서 자주 사용, 핵심만 딱 적을 수 있음.

(프로그래밍 언어) 너무 길어서 논문에서 부적합 (너무 길다)

• **자연어**로 기술된 알고리즘.

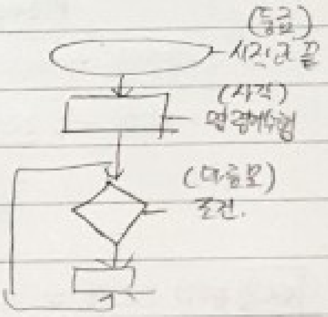
□ 인간이 읽기는 쉽다 → 장점이자 단점

□ 사용되는 자연어의 단어들을 정확하게 정의하지 않으면 의미 전달이 모호해질 우려 有.

ex) 배열에서 최대값 찾기 알고리즘. ArrayMax (list, n) 1. 배열 list[]의 첫번째 요소를 변수 tmp에 복사

⋮

• **흐름도**로 기술된 알고리즘.



• 직관적이고 이해하기 쉬운 알고리즘 기술 방법.

• **복잡한** 알고리즘의 경우 그림이 복잡해져서 이해도가 떨어짐.

• 의사코드로 기술된 알고리즘.

□ 알고리즘 기술에 **가장 많이 사용**

□ 프로그램 구현 시의 여러 문제들 감출 수 있음. ⟶ 알고리즘의 **핵심만을 기술**

```
ArrayMax(list, N):
    largest ← list[0]
    for i ← 1 to N ← 1 do
        if list[i] > largest
            then largest ← list[i]
    return largest.
```

※ 대입 연산자 => = X
← O

• 프로그래밍 언어로 기술된 알고리즘.

• 알고리즘을 가장 정확하게 기술할 수 있는 방법.

• 프로그램 내 어떤 중요하지 않은 세부사항이 알고리즘의 핵심적인 내용을 가릴 수 있다.

```
#define MAX_ELEMENTS 100

main ()
    int i, tmp;
```

자료형
- 기본 자료형: char / int / float / double
- 파생 자료형: 배열 / 포인터
- 사용자 정의 자료형: 구조체 / 공용체 / 열거형

1.2 추상 자료형 Abstract Data Type

- 추상 데이터 타입(ADT)

□ 데이터 타입(data type) → 데이터 + 연산(함수) (데이터의 종류)

□ 데이터의 집합과 연산의 집합

ex) int 데이터 타입
- 데이터 : { ..., -2, -1, 0, 1, 2, ... }
- 연산 : { +, -, /, *, % } ← 함수

□ 추상 데이터 타입(ADT : Abstract Data Type)

□ 데이터 타입을 추상적(수학적)으로 정의한 것. → 구체적이기보다는 추상적(수학적)으로 산다.

(구체적이지 않으면 불분명하다는 것은 사실!!)

□ 데이터나 연산이 무엇(what)인가는 정의 O / but 어떻게(how) 컴퓨터상 구현할 것인지를 정의 X.

- 데이터(연산) → 무엇인지 정의 O
  - → 어떻게 구현 정의 X

→ 데이터와 연산을 추상적 / 수학적으로 기술한다. 세부적인 실제 어떻게 프로그램의 구현은 쓰지 않는다.

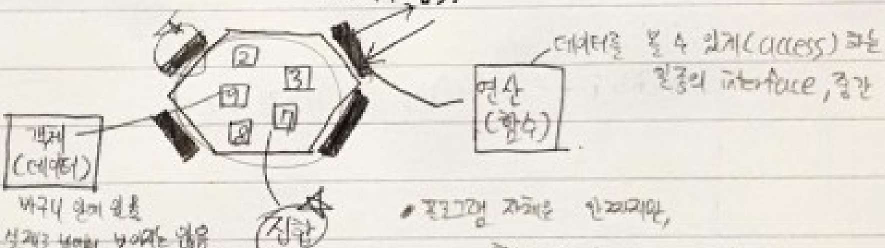
→ 어떻게 ADT를 구현하는가는 구현자의 몫.

★ 정보은닉기법 (information hiding)

⇒ 좋은 추상화, 사용자에게 중요한 정보는 강조 / 중요하지 않은 구현 세부사항은 제거.

□ 객체(데이터, objects) : 추상 데이터 타입(ADT)을 구성하는 데이터.

□ 연산(함수, function, operation) : 객체들을 사용한 연산 (연산은 추상 데이터 타입과 외부를 연결하는 인터페이스임).



• 프로그램 자체는 안 그렇지만, 함수가 무슨 일을 하고 그 기능은 정확히 표시해야 한다.

<ADT 순서>

1. 먼저 ADT의 이름부터 시작
2. 객체 정의 : 주로 집합 개념을 사용.
3. 그 다음, 그 이후 함수 정의.

::= → 으로 정의한다.

• 추상 자료형(ADT)
- 사용자들 → ADT가 제공하는 연산만을 사용.
- 사용자들 → ADT가 제공하는 연산들을 사용하는 방법을 알아야 함.
- 사용자들 → ADT 내부의 데이터를 접근할 수 없음.
- 사용자들 → ADT가 어떻게 구현되는지 모르더라도 ADT 사용할 수 있게 함.
- 만약 다른 사람이 ADT의 구현을 변경하더라도 인터페이스가 변경되지 않으면 사용자들이 여전히 ADT 사용가능.

→ 문제를 푸는 데 걸리는 시간

1.3 알고리즘의 성능 분석

- 알고리즘의 성능 분석.

• 프로그램의 효율성이 (시간) 중요한 이유

1. 처음 프로그램의 규모가 커짐에 따라 걸리는 처리할 자료의 양

2. 사용자들은 빠른 프로그램을 선호.

• 알고리즘의 성능 분석 방법. (시간) (빨리 풀면 성능이 좋다!)

측정 • 수행 시간 측정 (execution time measurement) → 프로그램 실행시켜놓고 시간 측정.

- 알고리즘을 수행 ➡ 수행 시간을 측정하는 방법.
- 알고리즘을 프로그램으로 구현해야 함! (슈도 코드 X, 프로그래밍언어 있어야 함.)
- 프로그램을 수행하는 하드웨어에 따라 수행 시간이 달라짐. (→ 소프트웨어 환경도 중요)

분석 • 시간 복잡도 분석 (time complexity analysis) → 실제 수행은 아니고, 알고리즘을 보고 내가 분석하는 방법 (단, 슈도코드같은 걸로 구현)

- 알고리즘을 분석 ➡ 수행 시간을 예측하는 방법.
- 일반적으로 알고리즘을 보고 주요 연산의 수행 횟수를 분석.
- 시간 복잡도 T(n)는 입력 데이터의 개수인 n의 함수로 주어짐.

○ 수행 시간 측정 방법.

• 알고리즘을 프로그램으로 구현한 후 수행시키면서 시간을 측정 (참고: clock() 등의 도구 사용).

```
#include <stdio.h>
   :
int main(void)
{
    clock_t ... ;
    start = clock();   // 측정 시작
      :
    stop = clock();    // 측정 종료
```

→ 우리는 이걸 공부!! ("알고리즘 사이의 연산을 찾아내는 것!")

☆ 시간 복잡도 분석 방법.

- 시간 복잡도 (time complexity)
  : 알고리즘 수행시간 분석
- 공간 복잡도 (space complexity)
  : 알고리즘이 사용하는 기억공간 분석.

□ 알고리즘을 이루고 있는 연산들이 몇 번이나 수행되는지를 분석하여 숫자로 표시.

□ 분석할 연산 : 산술 연산, 대입 연산, 비교 연산, 이동 연산 등이 분석 대상임.

□ 연산의 수행 횟수는 고정된 숫자가 아니라 입력 데이터의 개수 n에 대한 함수 T(n)으로 표현 함.

n → T(n)

입력데이터 n에 대한 함수 T(n) └ 수행횟수

프로그램A: T(n) = 3n+2

프로그램B: $T(n) = 5n^2+6$.

이만큼의 연산을하고 끝났다.

시간이 더 적게 걸림!! ⇒ 성능이 더 좋다!

☆ 알고리즘을 수행한 연산의 개수, 여기서는 특히 차수가 제일 비교가능!

☆ T(n) ⇒ 시간 복잡도 함수 ☆

<시간복잡도 분석>

☆ 연산 횟수 전부세기!

□ 알고리즘을 분석해보면 수행되는 연산들의 횟수를 입력 크기의 함수로 정의할 수 있음.

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

(for ~ tmp ← A[i]; : n-1 번)

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

if 문 안에서는 최대횟수를 따져야함.

총연산수 = 2n (최대) ☆

1번의 대입 + n-1의 비교 + n-1(최대) + 1번의 반환

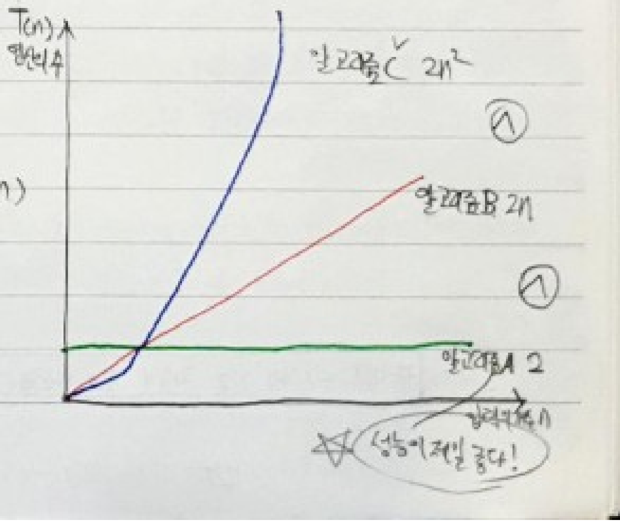
⇒ 2n ☆

알고리즘 A	알고리즘 B	알고리즘 C

sum ← n*n;	for i←1 to n do
	sum ← sum+n;
		for i←1 to n do
		for j←1 to n do
		sum ← sum+j;

	알고리즘A	알고리즘B	알고리즘C
대입연산	1	n	n*n
덧셈연산		n	n*n
곱셈연산	1		
나눗셈연산			
전체연산수	2	2n	$2n^2$

(n안에 n)



- 시간 복잡도의 차수 비교.

- 자료의 개수가 많은 경우 ➡ T(n)에서 차수가 가장 큰 항이 가장 영향을 크게 미치고 다른 항들은 상대적으로 무시.

(ex) - 예를 들어 $T(n) = n^2+n+1$ 이면...

if $n=1000$이면 / $T(n)$의 값은 1,001,001 [첫 번째 항인 n²의 값이 99.9 %
나머지는 0.1%.

(따라서) ⇒ 보통 시간 복잡도 함수 ➡ 가장 영향을 크게 미치는 차수가 가장 큰 항만을 고려하면 충분.

$n=1000$인 경우

$T(n) = n^2 + n+1$

99.9% 0.1%

"식이 복잡하게 나오면 우리는 높은 차수만 보고 얘기하면 된다!!!"

- 빅오, 빅오메가, 빅세타 표기법.

<상한>

<빅오(Big O) 표기법>

- 빅오 표기법 : 연산의 수를 대략적 (점근적)으로 표기한 것.

- 두 개의 함수 $f(n)$과 $g(n)$이 주어졌을 때, 모든 $n \geq n_0$에 대하여, $f(n) \leq c * g(n)$을 만족하는 2개의 상수 c와 n_0가 존재하면, $f(n) = O(g(n))$으로 정의된다.

f(n)은 g(n)에 c라는 어떤 상수를 곱한 것보다 항상 작다!!

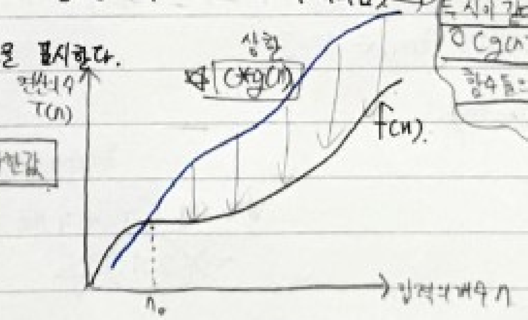
(주의 : 여기서 = 은 같다는 의미가 아니고 ∈ 의 의미임)

등식이 같다는 게 아닌 O(g(n))이라는 함수들의 집합 안에 들어간다는 소리!

- 빅오는 함수의 상한을 표시한다.

- f(n)은 n이 매우 커지게 되면 결국은 g(n)보다 작거나 같게 된다.

g(n) ➡ f(n)의 상한값



예제 1.1 빅오 표기법 차수가 없다 ⇒ O(1)

- c 값은 달라져도 일은 일어나도 무방하다.

- $f(n)=5$이면 $O(1)$이다. 왜냐하면 $n_0=1$, $c=10$일 때, $n>1$에 대하여 $5 \leq 10 \cdot 1$이 되기 때문이다.
- $f(n)=2n+1$이면 $O(n)$이다. 왜냐하면 $n_0=2$, $c=3$일 때, $n>2$에 대하여 $2n+1 \leq 3n$이 되기 때문이다.
- $f(n)=3n^2+100$이면 $O(n^2)$이다. 왜냐하면 $n_0=100$, $c=5$일 때, $n>100$에 대하여 $3n^2+100 \leq 5n^2$이 되기 때문이다.
- $f(n)=5 \cdot 2^n+10n^2+100$이면 $O(2^n)$이다. 왜냐하면 $n_0=1000$, $c=10$일 때, $n>1000$에 대하여 $5 \cdot 2^n+10n^2+100 \leq 10 \cdot 2^n$이 되기 때문이다.

- 상수 지워버리기.

n_0 ⇒ 일종의 기준 입력값?

$2^n > n^2$

✩ 빅오 ⇒ 가장 높은 차수의 좋기 자료형 (1, n, n^2, 2^n, ...)

- 최고차항의 계수도 떼고 단지 최고차항의 차수만을 사용!
- 주의할 것!! ⇒ logn은 살려야 한다 → logn도 차수를 가지고 있기 때문.

<빅오 종류별 시간>

T(n)	1	2	4	8	16	32
O(1)	1	1	1	1	1	1
O(lgn)	0	1	2	3	4	5
O(n)	1	2	4	8	16	32
O(n*lgn)	0	2	8	24	64	160
$O(n^2)$	1	4	16	64	256	1024
$O(n^3)$	1	8	64	512	4096	32768
$O(2^n)$	2	4	16	256	65536	4294967296
O(n!)	1	2	24	40326	20922789888000	26313×10^{33}

n의 증가에 따른 T(n)의 증가는 작음.

P — 여기에 들어가는 알고리즘을 쓰기 편함.

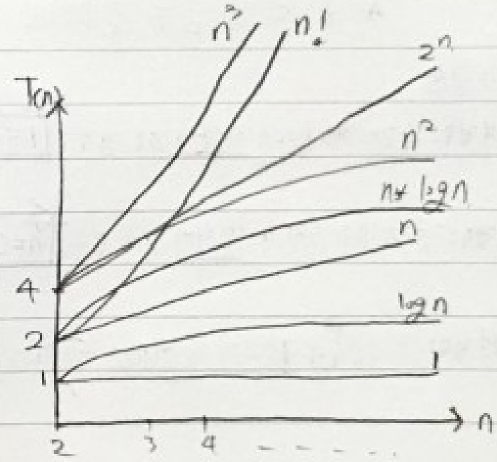
e — 여기에 들어가는 알고리즘은 쓰기 어려움.

n의 증가에 따른 T(n)의 증가가 폭발적.

- O(1) : 상수형
- O(log n) : 로그형
- O(n) : 선형
- O(n*logn) : 로그선형
- $O(n^2)$: 2차형
- $O(n^3)$: 3차형
- $O(2^n)$: 지수형
- O(n!) = 팩토리얼형

P: O(1) ~ $O(n^3)$

e (지수형): $O(2^n)$, O(n!)

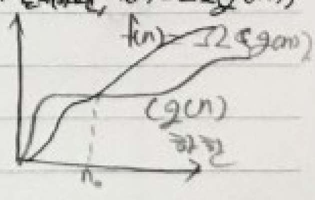


☆ $O(1) < O(\log n) < O(n) < O(n*\log n) < O(n^2) < O(2^n) < O(n!)$

- 빅오 표기법은 오직 최소 함수로 표기되었을 경우에만 의미가 있다는 문제점이 있다.

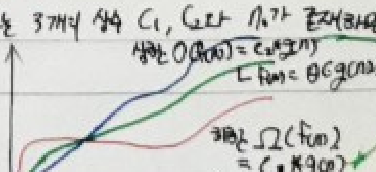
<빅오 표기법 이외의 표기법> — 하한 / 중간(상한+하한)

- 빅오메가 (Big Ω) 표기법
  - 모든 $n \ge n_0$에 대하여, $f(n) \ge c * g(n)$을 만족하는 2개의 상수 c와 n_0가 존재하면, $f(n) = \Omega(g(n))$으로 정의한다. (주의: 여기서 =은 같다는 의미가 아닌 ∈의 의미임).
  - 빅오메가는 함수의 하한을 표시한다.



- 3개의 표기법 중에서 가장 정밀한 것은 빅세타.
- 그러나 일반적으로 빅오 표기법을 많이 사용 (최악의 경우를 표시하기 때문)

- 빅세타 (Big θ) 표기법. — 중간 (상한+하한)
  - 모든 $n \ge n_0$에 대하여, $c_1 * g(n) \le f(n) \le c_2 * g(n)$을 만족하는 3개의 상수 c_1, c_2와 n_0가 존재하면, $f(n) = \theta(g(n))$으로 정의한다. (주의: 여기서 =은 같다.)
  - 빅세타는 함수의 하한과 상한을 동시에 표시한다.



-최선, 평균, 최악 경우 분석.

<최선, 평균, 최악의 경우>

- 알고리즘의 수행 시간은 입력 크기에 따라 달라지고 입력 데이터의 특성에 따라 매우 다를 수 있다.

입력의 집합

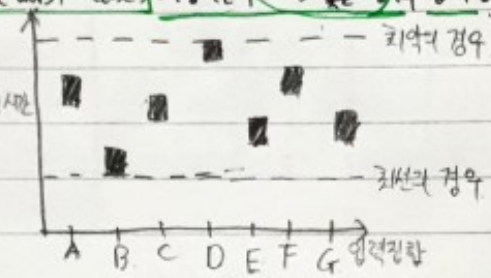
데이터의 특성이 중요!!

ex) 순차 탐색 알고리즘의 수행 시간은 입력 집합의 특성에 따라 다름.

- 최선의 경우 (best case): 수행 시간이 가장 빠른 경우의 입력 집합
- 평균의 경우 (average case): 수행 시간이 평균적인 경우의 입력 집합
- 최악의 경우 (worst case): 수행 시간이 가장 느린 경우의 입력 집합. → 우리는 이 경우를 생각!!

=> 가장 오래 걸릴 때도 보고 그걸 꼭 보장되기 때문.

입력 자료 집합을 알고리즘에 적용한 결과로 얻어지는 컴퓨터의 시간이 소모되는 것을 분석!!



ex) 순차 탐색

- 최선의 경우: 찾고자 하는 숫자가 맨 앞에 있는 경우 : 1 = O(1)

인덱스 0에서 5번째 찾고 있음 = 1

- 최악의 경우: 찾고자 하는 숫자가 맨 뒤에 있는 경우 : n = O(n)

인덱스 9에서 43번째 찾음 = 10

- 평균적인 경우: 각 요소들이 균등하게 탐색된다고 가정하면: (1+2+ ··· +n)/n = (n+1)/2 = O(n)

- 최선의 경우 분석: 크게 의미가 없음.
- 평균적인 경우 분석: 계산하기가 상당히 어려움. ~ 정형화된 자료 집합에 대하여 알고리즘을 적용시키기 때문.
- 최악의 경우 분석: 알고리즘 분석에 가장 널리 사용, 계산하기도 쉽고 중요한 의미를 가짐.