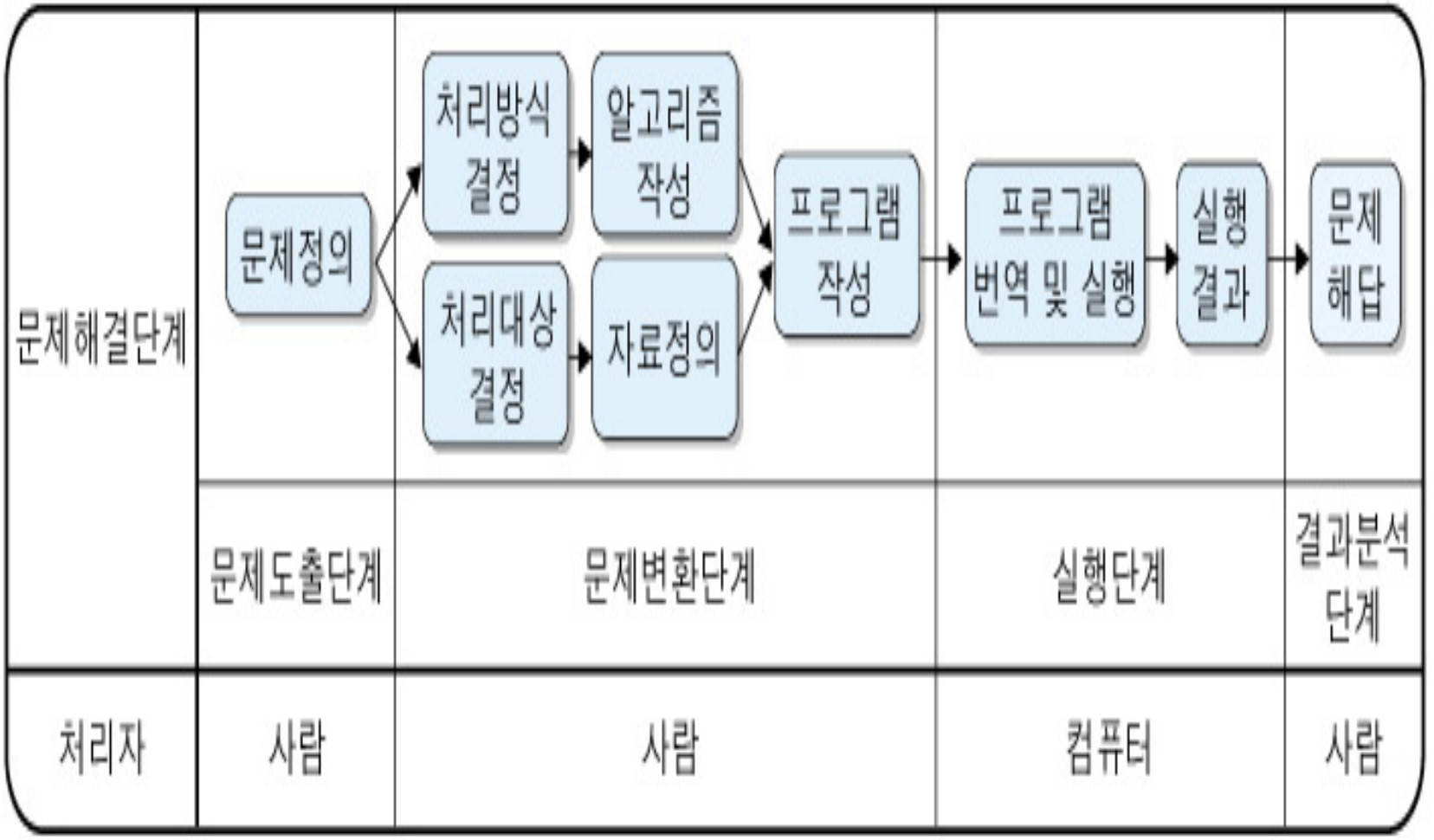
Data Structure & Algorithm

개요

❖ 컴퓨터에 의한 문제 해결 과정



개요

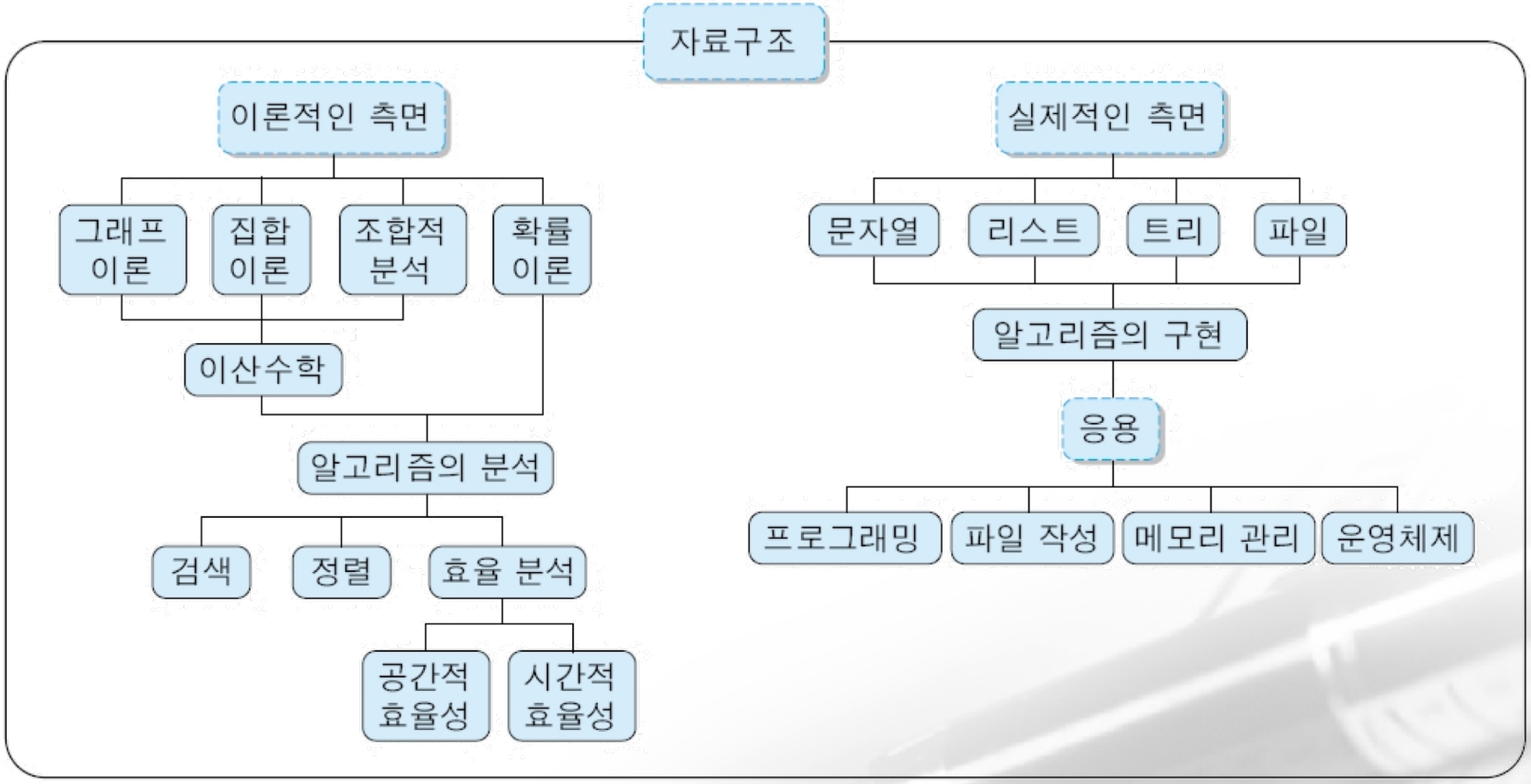
- 자료 구조(Data Structure)
  - ✓ 컴퓨터에 자료를 저장하는 방식
  - ✓ 자료를 효율적으로 사용하기 위해서 자료의 특성에 따라서 분류하고 구성한 후 저장 및 처리하는 모든 작업
  - ✓ 효율적인 자료 구조는 메모리를 절약할 뿐 아니라 프로그램 수행 시간을 최소화하는 기능을 수행

개요

- ❖ 자료 구조(Data Structure) 분류
  - ✓ 단순 구조 – Scala Data
    - 데이터 1개를 저장하기 위한 구조
    - 정수, 실수, 문자, 문자열 등의 기본 자료형
  - ✓ 선형 구조
    - 자료들 간의 앞뒤 관계가 1:1의 선형 관계
    - Dense List (배열, 연접 리스트), Linked List, Stack, Queue, Deque 등
  - ✓ 비선형 구조
    - 자료들 간의 앞뒤 관계가 1:N, 또는 N:N의 관계
    - Tree, Graph 등
  - ✓ 파일 구조
    - 레코드의 집합인 파일에 대한 구조
    - Sequential File, Indexed File, Direct File 등

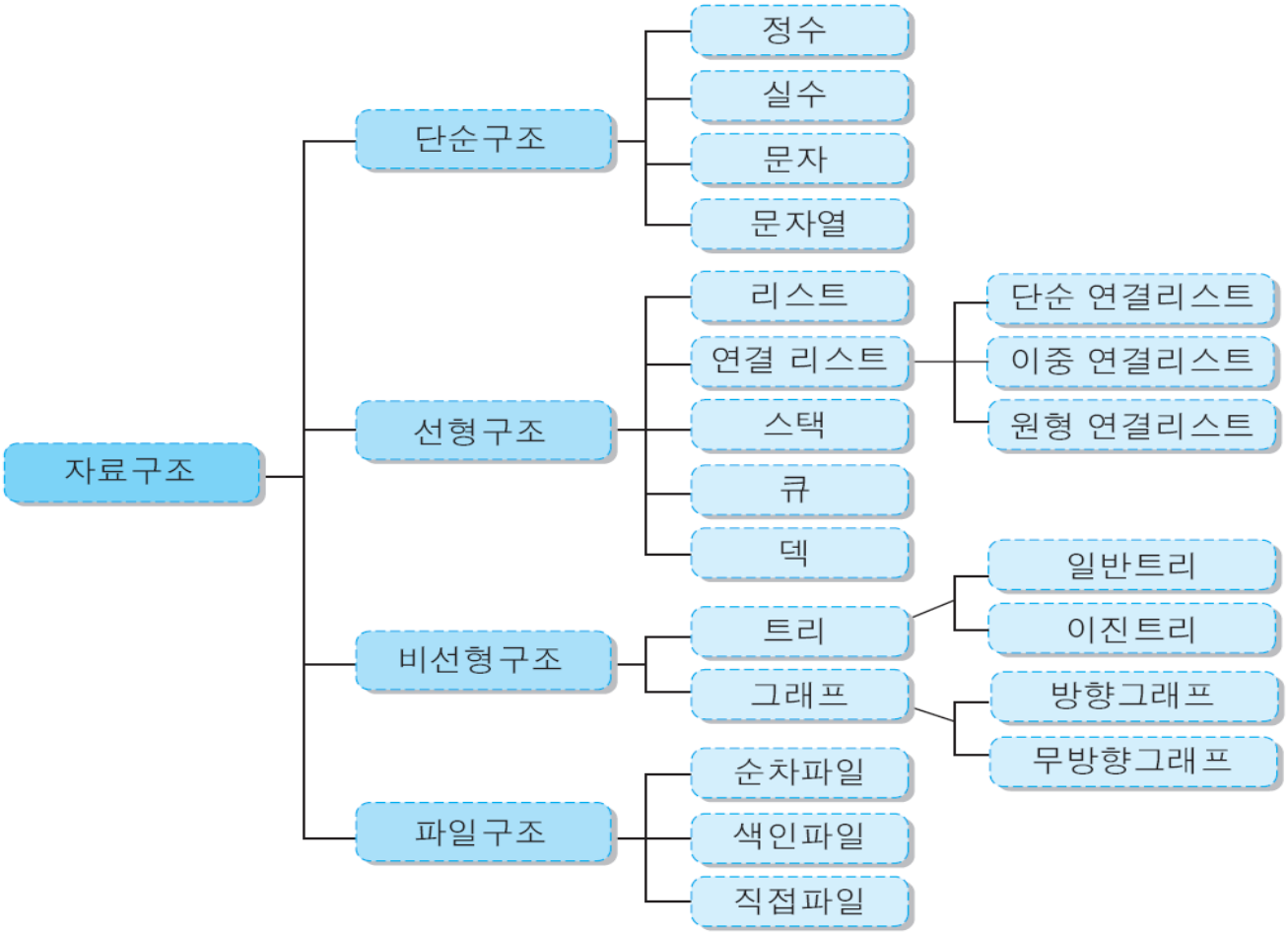
개요

❖ 자료 구조(Data Structure) 분류



개요

❖ 자료 구조(Data Structure) 분류



개요

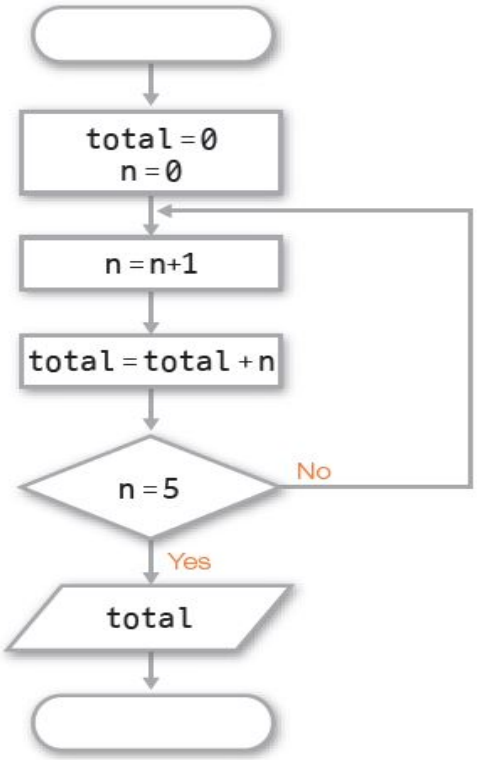
- ❖ 알고리즘
  - ✓ 어떠한 문제를 해결하기 위한 절차
  - ✓ 문제 해결 방법을 추상화하여 각 절차를 논리적으로 기술해 놓은 명세서
  - ✓ 알고리즘의 조건
    - 입력(input): 알고리즘 수행에 필요한 자료가 외부에서 입력으로 제공될 수 있어야 함
    - 출력(output): 알고리즘 수행 후 하나 이상의 결과를 출력해야 함
    - 명확성(definiteness): 수행할 작업의 내용과 순서를 나타내는 알고리즘의 명령어들은 명확하게 명세되어야 함
    - 유한성(finiteness): 알고리즘은 수행 뒤에 반드시 종료되어야 함
    - 효과성(effectiveness): 알고리즘의 모든 명령어들은 기본적인 명령이어야 하며 실행이 가능해야 함
- ✓ 알고리즘과 자료 구조의 관계
  - ○ 같은 자료라 하더라도 어떻게 표현되고 저장 되느냐에 따라 사용 가능한 알고리즘이 달라짐
  - ○ 효율적인 자료 구조의 설계는 알고리즘의 설계에 영향을 끼치기 때문에 프로그램의 성능을 결정짓게 하는 가장 중요한 요소 중 하나
  - ○ 효율적으로 설계되지 않은 자료 구조는 알고리즘을 비효율적으로 만들 뿐 아니라 문제 해결을 위한 알고리즘 자체를 만들어 내기 어려움

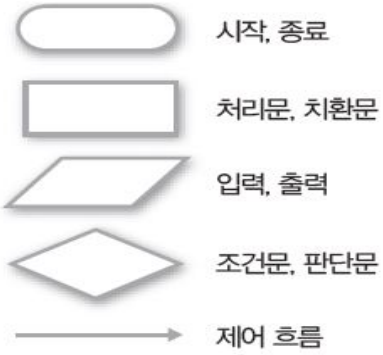
개요

- ❖ 알고리즘
  - ✓ 알고리즘의 표현
    - 자연어를 이용한 서술적 표현 방법: 기술하는 사람에 따라 일관성과 명확성이 달라지기 때문에 알고리즘 표현으로 부적절
    - 순서도(Flow chart)를 이용한 도식화 표현 방법: 알고리즘 각 단계를 직관적으로 표현할 수 있는 장점이 있으나 복잡한 알고리즘을 나타내기에는 비효율적
    - 가상 코드(Pseudo-code)를 이용한 추상화 방법
      - ◆ 프로그래밍 언어보다는 덜 엄격한 문법이나 자연어보다는 더 체계적으로 기술이 가능
      - ◆ 표현하는 개발자 마다 약간씩 구문에 차이가 있음
    - 프로그래밍 언어를 이용한 구체화 방법
      - ◆ 구체적인 구현 소스를 통해 나타내기 때문에 추가 구현 단계가 필요 없음
      - ◆ 너무 엄격하게 기술하기 때문에 비효율적인 경우가 많음

개요

- 알고리즘
  - ✓ 순서도를 이용한 알고리즘 표현
    - 1 부터 5 까지의 합을 구하는 알고리즘





개요

- ❖ 알고리즘
  - ✓ 가상 코드(Pseudo-Code)를 이용한 알고리즘 표현
    - 가상 코드 즉 알고리즘 기술 언어(ADL, Algorithm Description Language)를 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
    - 기본 요소
      - ◆ 기호
        - 변수, 자료형 이름, 프로그램 이름, 레코드 필드 명, 문장의 레이블 등을 나타냄
        - 문자나 숫자의 조합. 첫 문자는 반드시 영문자 사용
      - ◆ 자료형
        - 정수형과 실수형의 수치 자료형, 문자형, 논리형, 포인터, 문자열 등의 모든 자료형 사용
      - ◆ 연산자
        - 산술 연산자, 관계 연산자, 논리 연산자

개요

- 알고리즘
  - ✓ 가상 코드(Pseudo-Code)를 이용한 알고리즘 표현
    - 대입

```
변수 ← 값
```

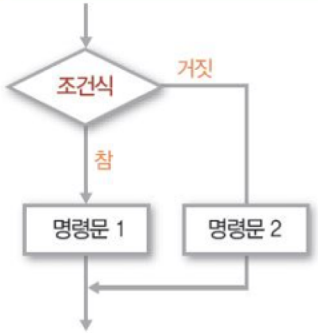
```
a ← 5
a ← 3+2
a ← b;
```

개요

- ❖ 알고리즘
  - ✓ 가상 코드(Pseudo-Code)를 이용한 알고리즘 표현
    - ● 조건에 따른 분기

```
if (조건식) then 명령문 1;
```

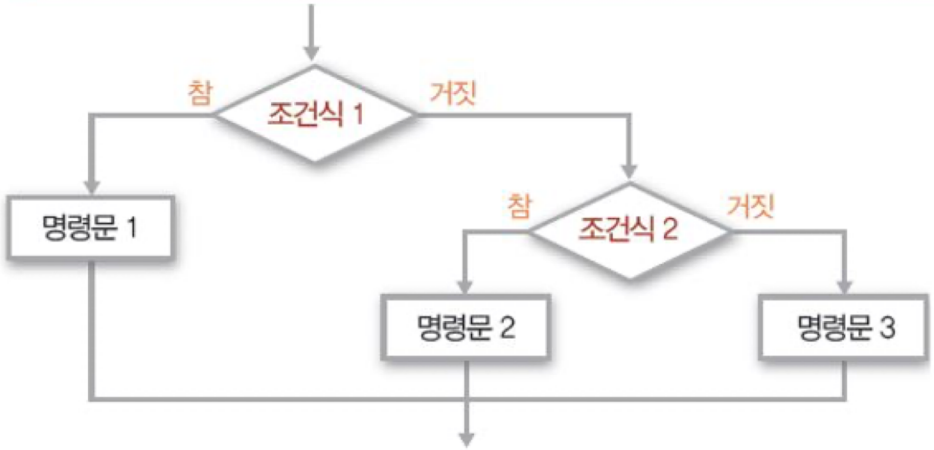

```
if (조건식) then 명령문 1;
else 명령문 2;
```



개요

- ❖ 알고리즘
  - ✓ 가상 코드(Pseudo-Code)를 이용한 알고리즘 표현
    - 조건에 따른 분기

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



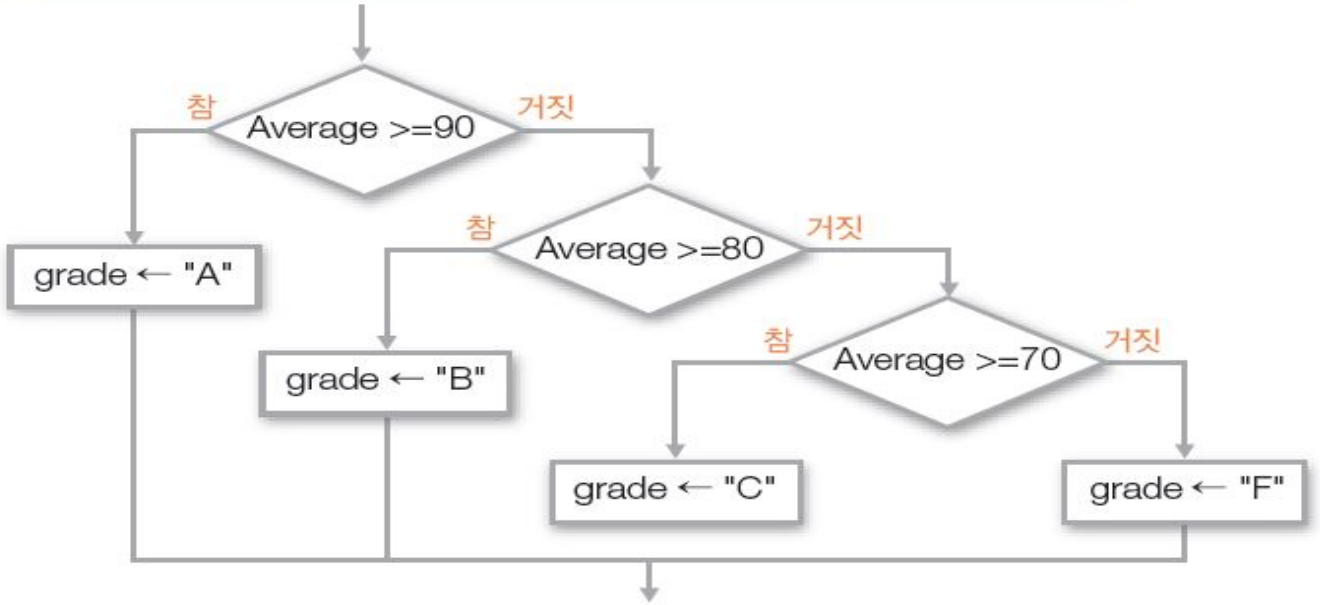
개요

❖ 알고리즘

✓ 가상 코드(Pseudo-Code)를 이용한 알고리즘 표현

● 조건에 따른 분기

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



개요

❖ 알고리즘

✓ 가상 코드(Pseudo-Code)를 이용한 알고리즘 표현

● 값에 따른 분기

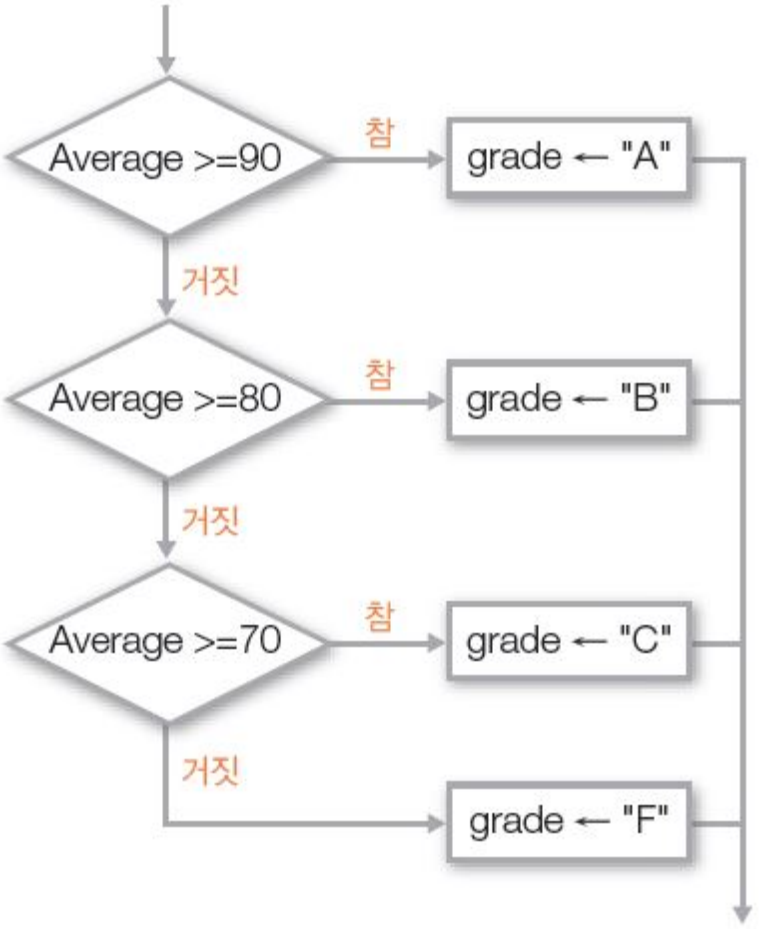
```
case {
    조건식 1 : 명령문 1;
    조건식 2 : 명령문 2;
    ...
    조건식 n : 명령문 n;
    else    : 명령문 n+1;
}
```

개요

❖ 알고리즘

✓ 가상 코드(Pseudo-Code)를 이용한 알고리즘 표현

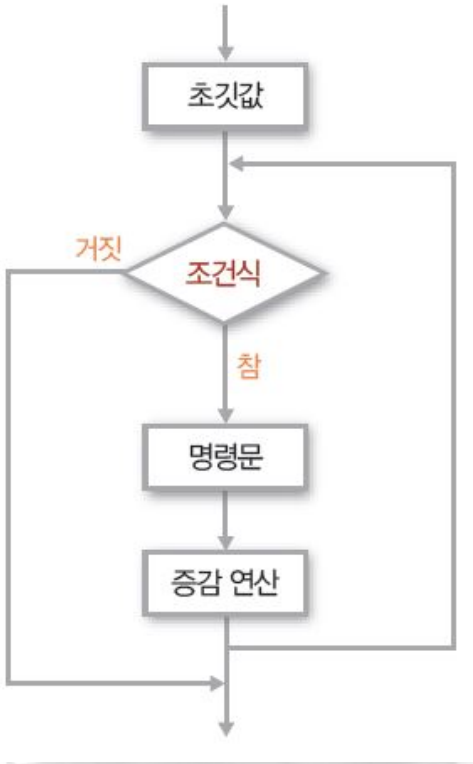
- 값에 따른 분기




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

개요

- ❖ 알고리즘
  - ✓ 가상 코드(Pseudo-Code)를 이용한 알고리즘 표현
    - ● 반복(for)

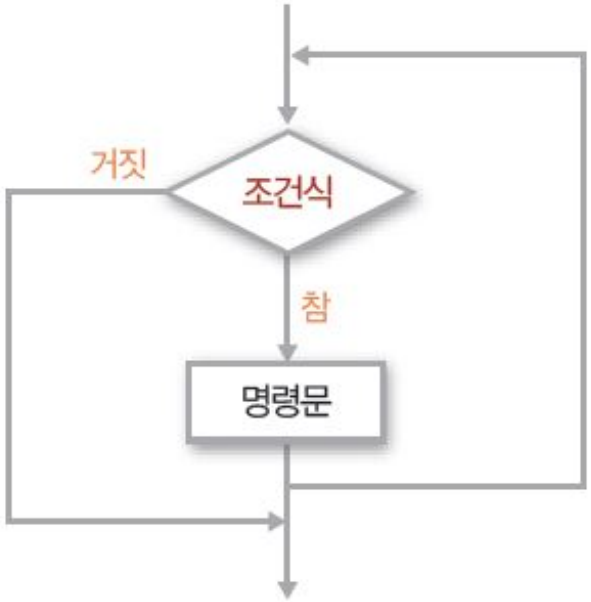



```
for (초깃값; 조건식; 증감값) do 명령문;
```

개요

- 알고리즘
  - ✓ 가상 코드(Pseudo-Code)를 이용한 알고리즘 표현
    - 반복(while)

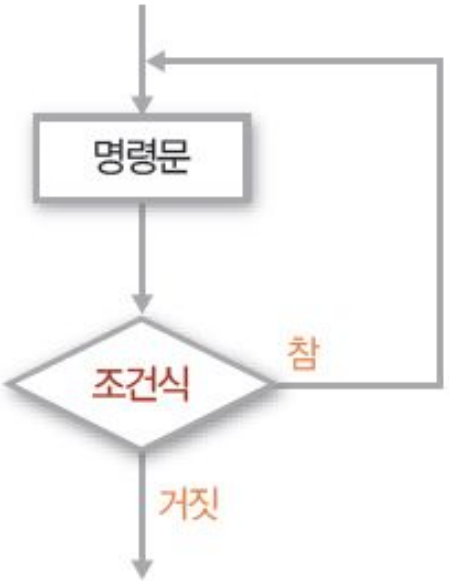
```
while (조건식) do 명령문;
```



개요

- ❖ 알고리즘
  - ✓ 가상 코드(Pseudo-Code)를 이용한 알고리즘 표현
    - ● 반복(do ~ while)

```
do 명령문;
while (조건식);
```

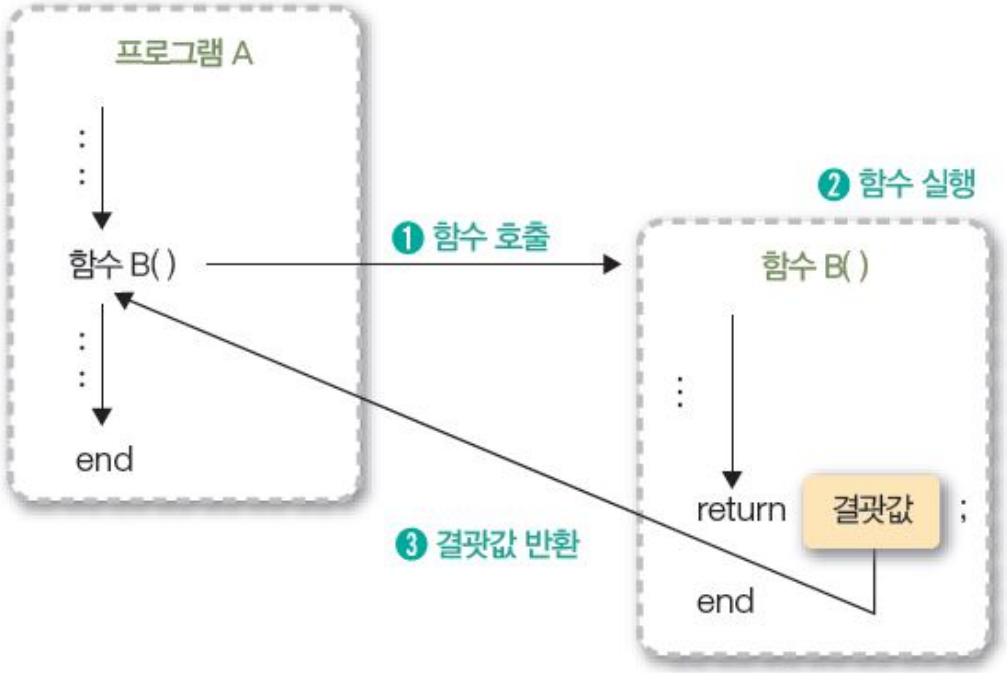


개요

- 알고리즘
  - ✓ 가상 코드(Pseudo-Code)를 이용한 알고리즘 표현
    - 함수 호출

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

(a) 함수의 형식



(b) 함수의 호출과 실행 및 결괏값 반환의 예

개요

- ❖ 알고리즘 분석
  - ✓ 알고리즘 성능 분석 기준
    - 기준에는 정확성, 명확성, 수행량, 메모리 사용량, 최적성 등 있음
    - 정확성 : 올바른 자료 입력 시 유한한 시간 내에 올바른 결과 출력 여부
    - 명확성 : 알고리즘이 얼마나 이해하기 쉽고 명확하게 작성되었는가
    - 수행량 : 일반적인 연산 제외, 알고리즘 특성 나타내는 중요 연산 모두 분석
    - 메모리 사용량
    - 최적성

개요

- 알고리즘 분석
  - ✓ 알고리즘 성능 분석 방법
    - 공간 복잡도
      - 알고리즘을 프로그램으로 실행하여 완료하기까지 필요한 총 저장 공간의 양
      - 공간 복잡도 = 고정 공간 + 가변 공간
    - 시간 복잡도
      - 알고리즘을 프로그램으로 실행하여 완료하기까지의 총 소요 시간
      - 시간 복잡도 = 컴파일 시간 + 실행 시간
      - 컴파일 시간: 프로그램마다 거의 고정적인 시간 소요
      - 실행 시간: 컴퓨터의 성능에 따라 달라질 수 있으므로 실제 실행시간 보다는 명령문의 실행 빈도수에 따라 계산
      - 실행 빈도수의 계산
      - 지정문, 조건문, 반복문 내의 제어문과 반환문은 실행시간 차이가 거의 없으므로 하나의 단위 시간을 갖는 기본 명령문으로 취급
  - ✓ 알고리즘의 성능 분석에 시간 복잡도가 공간 복잡도보다 더 중요한 평가 기준이기 때문에 알고리즘의 성능 분석은 대부분 시간 복잡도를 대상으로 함

개요

❖ 알고리즘 분석

✓ 시간 복잡도

피보나치 수열

```
00  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
```

개요

- ❖ 알고리즘 분석
  - ✓ 시간 복잡도
    - ● 입력 값에 따른 실행 연산의 빈도수

피보나치 수열 알고리즘의 실행 빈도수

(a) n<0, n=0, n=1인 경우

행 번호	n<0	n=0	n=1
01	1	1	1
02	1	0	0
03	0	1	1
04	0	1	1
05~13	0	0	0

(b) n>1 경우

행 번호	n>1	행 번호	n>1
01	1	08	n-1
02	0	09	n-1
03	1	10	n-1
04	0	11	0
05	1	12	1
06	1	13	0
07	n		

개요

- ❖ 알고리즘 분석
  - ✓ 시간 복잡도
    - ● 빅-오(O) 표기법
      - ◆ O(f(n))과 같이 표기 - Big Oh of f(n) 으로 읽음
      - ◆ 수학적 정의: 함수 f(n)과 g(n)이 주어졌을 때, 모든 n ≥ n0에 대하여 |f(n)| ≤ c |g(n)| 을 만족하는 상수 c 와 n0 가 존재하면 f(n) = O(g(n))
      - ◆ 함수의 상한을 나타내기 위한 표기법: 최악의 경우에도 g(n)의 수행 시간 안에는 알고리즘 수행 완료 보장
      - ◆ 먼저 실행 빈도수를 구하여 실행 시간 함수를 찾고 이 함수 값에 가장 큰 영향을 주는 n에 대한 항을 한 개 선택하여 계수는 생략하고 O의 오른쪽 괄호 안에 표시
      - ◆ 피보나치 수열에서 실행 시간 함수는 4n+2이고 가장 영향이 큰 항은 4n인데 여기에서 계수 4를 생략하여 O(n) 으로 표기

개요

- ❖ 알고리즘 분석
  - ✓ 시간 복잡도
    - ● 빅-오(O) 표기법
      - ◆ O(1)
        - 입력 값이 아무리 커도 실행 시간은 일정
        - 최고의 알고리즘이라 할 수 있음
        - 상수 시간을 갖는 알고리즘은 성배와 같아서 찾을 수만 있다면 놀라울 정도로 가치가 있지만 그 성배를 찾기 위해 일평생을 보내야 할지도 모름
        - 상수 시간에 실행된다 해도 상수 값이 상상을 넘어설 정도로 매우 크다면 사실상 일정한 시간의 의미가 없음
        - O(1)에 실행되는 알고리즘: 해시 테이블의 조회 및 삽입이 이에 해당

개요

- ❖ 알고리즘 분석
  - ✓ 시간 복잡도
    - ● 빅-오(O) 표기법
      - ◆ O(log n)
        - 실행 시간은 입력 값에 영향을 받지만 로그는 매우 큰 입력 값에도 크게 영향을 받지 않는 편으로 웬만한 n의 크기에 대해서도 매우 견고
        - 대표적으로 이진 검색이 이에 해당

개요

- ❖ 알고리즘 분석
  - ✓ 시간 복잡도
    - ● 빅-오(O) 표기법
      - ◆ O(n)
        - 입력 값만큼 실행 시간에 영향을 받으며 알고리즘을 수행하는 데 걸리는 시간은 입력 값에 비례
        - 선형 시간(Linear Time) 알고리즘
        - 정렬되지 않은 리스트에서 최댓값 또는 최솟값을 찾는 경우가 이에 해당하는데 이 값을 찾기 위해서는 모든 입력 값을 적어도 한 번 이상은 확인

개요

- ❖ 알고리즘 분석
  - ✓ 시간 복잡도
    - ● 빅-오(O) 표기법
      - ◆ O(n log n)
        - 병합 정렬을 비롯한 대부분의 효율 좋은 정렬 알고리즘이 이에 해당
        - 적어도 모든 수에 대해 한 번 이상은 비교해야 하는 비교 기반 정렬 알고리즘은 아무리 좋은 알고리즘도 O(n log n) 보다 빠를 수 없음
        - 입력 값이 최선인 경우 비교를 건너뛰어 O(n)이 될 수 있으며 Timsort가 이런 로직을 갖고 있음
        - 팀소트(Tim Sort)
          - 수 많은 종류의 실 세계 데이터에 잘 수행하도록 설계된 하이브리드형 안정 정렬 알고리즘의 하나로 합병 정렬과 삽입 정렬이 기원
          - 2002년 파이썬 프로그래밍에 사용하기 위해 팀 피터스가 구현
          - 팀소트는 버전 2.3부터 파이썬의 표준 정렬 알고리즘
          - 자바 SE 7, 안드로이드, GNU 옥타브, V8, 스위프트, 러스트에서 프리미티브가 아닌 타입의 배열을 정렬하기 위해서도 사용

개요

- ❖ 알고리즘 분석
  - ✓ 시간 복잡도
    - ● 빅-오(O) 표기법
      - ◆ $O(n^2)$: 버블 정렬 같은 비효율적인 정렬 알고리즘이 이에 해당
      - ◆ $O(2^n)$
        - 피보나치 수를 재귀로 계산하는 알고리즘이 이에 해당
        - $O(n^2)$ 과 혼동하는 경우가 있는데 처음에는 비슷해 보이지만 2"이 훨씬 큼
      - ◆ O(n!)
        - 각 도시를 방문하고 돌아오는 가장 짧은 경로를 찾는 외판원 문제(Ravelling Salesman Problem - TSP)를 브루트 푸스(brute force)로 풀이할 때가 이에 해당
        - 가장 느린 알고리즘으로 입력 값이 조금만 커져도 웬만한 다항 시간 내에는 계산이 어려움

개요

- ❖ 알고리즘 분석
  - ✓ 시간 복잡도
    - ● 빅-오메가 표기법
      - ◆ Ω(f(n))과 같이 표기 - Big Omega of f (n) 으로 읽음
      - ◆ 수학적 정의: 함수 f(n)과 g(n)이 주어졌을 때, 모든 n≥n0에 대하여 |f(n)| ≥ c |g(n)|을 만족하는 상수 c 와 n0 가 존재하면 f(n) = Ω(g(n))
      - ◆ 함수의 하한을 나타내기 위한 표기법: 어떤 알고리즘의 시간 복잡도가 Ω(g(n)) 으로 분석되었다면 이 알고리즘 수행에는 적어도 g(n)의 수행 시간이 필요함을 의미

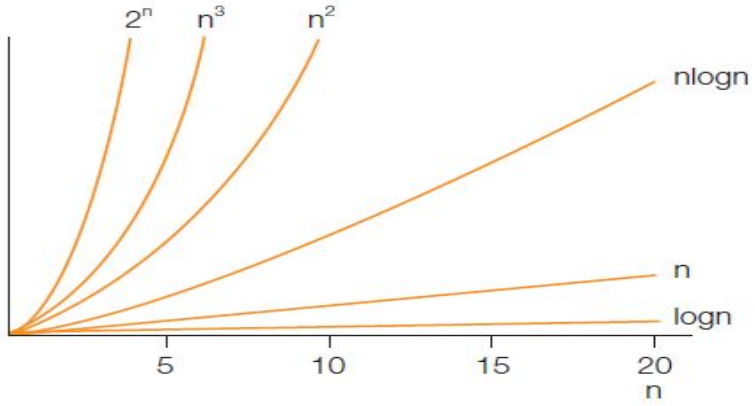
개요

- 알고리즘 분석
  - ✓ 시간 복잡도
    - 빅-세타 표기법
      - θ(f(n))과 같이 표기 - Big Theta of f (n) 으로 읽음
      - 수학적 정의: f(n)과 g(n)이 주어졌을 때, 모든 n≥n0에 대하여 c1|g(n)| ≤ f(n) ≤ c2 |g(n)| 을 만족하는 상수 c1, c2 와 n0 가 존재하면, f(n) = θ(g(n))
      - 상한과 하한이 같은 정확한 차수를 표현하기 위한 표기법: f(n)= θ (g(n))이 되려면 f(n)= O(g(n))이면서 f(n)= Ω(g(n))이어야 함

개요

- ❖ 알고리즘 분석
  - ✓ 시간 복잡도
    - 각 실행 시간 함수에서 n 값의 변화에 따른 실행 빈도수 비교

logn	<	n	<	nlogn	<	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



개요

❖ 알고리즘 분석

✓ 시간 복잡도

● 시간 복잡도에 따른 알고리즘 수행 시간 비교

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	

개요

- ❖ 알고리즘 분석
  - ✓ 시간 복잡도
    - ● 시간 복잡도에 따른 알고리즘 수행 시간 비교
      - ◆ 입력 조건에서 명시되어 있는 최악의 경우를 시간 복잡도에 대입해보았을 대 1억이 넘지 않는다면 실행 시간이 1초보다 빠른 충분히 효율적인 코드일 가능성이 높음
      - ◆ 문제에서 주어진 조건에 따라 N 값이 최대 1만이라고 가정하면 시간 복잡도 O(N logN) 을 갖는 코드는 N을 대입했을 때 그 값이 약 13만으로 계산되기 때문에 충분히 효율적인 코드라고 할 수 있지만 작성한 코드가 시간 복잡도 $O(N^2)$ 을 갖는다면 N을 대입했을 때 1억이 되므로 시간 제한을 맞추지 못할 가능성이 있음
      - ◆ 길이가 N인 배열의 반만 사용하는 알고리즘의 경우 반복 횟수는 N/2 인데 이 경우는 O(N) 으로 간주하지만 배열을 M번 반복해야 한다면 이 경우에는 M을 무시하면 안되고 O(NM)으로 간주해서 N의 최댓값도 고려해야 하고 길이 N짜리 배열을 순회하고 그 다음에 M짜리 배열을 순회하는 경우에는 O(N + M)으로 표기하고 N 과 M의 최댓값을 구해서 계산

개요

- ❖ 알고리즘 분석
  - ✓ 시간 복잡도
    - ● 시간 복잡도에 따른 알고리즘 수행 시간 비교
      - ◆ 정렬된 배열에서 원소의 위치를 찾는 문제를 생각해보면 배열의 모든 원소를 순회한다면 이는 O(N)의 시간 복잡도가 소요되지만 정렬되어 있다는 조건을 생각해보면 이진 탐색을 적용할 수 있는데 이진 탐색의 시간 복잡도는 O(logN) 이므로 훨씬 효율적인 탐색이 됨
      - ◆ 배열에서 중복을 제거한 원소를 찾고자 하는 경우 원소 별로 배열 전체를 순회하면 $O(N^2)$의 시간 복잡도가 소요되지만 이 때 자료구조를 Set을 이용하면 O(N) 으로 해결
      - ◆ 시간 제한이 1초일 때 일반적인 유추 가능한 시간 복잡도 와 알고리즘

N	시간 복잡도	알고리즘
10	O(N!)	순열
20	$O(2^N)$	조합
1,000	$O(N^3)$, $O(N^3 \log N)$	완전 탐색, 이진 탐색
10,000	O(N logN)	정렬, 이진 탐색

개요

- ❖ 알고리즘 분석
  - ✓ 시간 복잡도
    - ● 분할 상환 분석
      - ◆ 시간 또는 메모리를 분석하는 알고리즘의 복잡도를 계산할 때 알고리즘 전체를 보지 않고 최악의 경우만을 살펴보는 것은 지나치게 비관적이라는 이유로 분할 상환 분석 방법이 등장
      - ◆ 분할 상환 분석(Amortized Analysis)은 빅오 와 함께 함수의 동작을 설명할 때 중요한 분석 방법 중 하나
      - ◆ 분할 상환 분석이 유용한 대표적인 예로 동적 배열을 들 수 있는데 동적 배열에서 더블링이 일어나는 일은 어쩌다 한 번 뿐 이지만 이로 인해 아이템 삽입 시 시간 복잡도는 O(n)이다 라고 얘기하는 건 지나치게 비관적이고 정확하지도 않기 때문에 이 경우 분할 상환 또는 상각이라고 표현하는 최악의 경우를 여러 번에 걸쳐 골고루 나눠주는 형태로 알고리즘의 시간 복잡도를 계산할 수 있는데 이렇게 할 경우 동적 배열의 삽입 시 시간 복잡도는 O(1)
      - ◆ 이 방법은 1985년 미국의 수학자이자 컴퓨터과학자인 로버트 타잔이 상각된 계산 복잡도(Amortized Computational Complexity)라는 논문에서 처음으로 소개했으며 그 유용함 덕분에 최근에는 시간 복잡도 를 분석할 때 매우 보편적으로 널리 사용되는 방법

개요

- ❖ 알고리즘 분석
  - ✓ 시간 복잡도
    - ● 병렬화
      - ◆ 일부 알고리즘들은 병렬화로 실행 속도를 높일 수 있음
      - ◆ 최근에는 딥 러닝의 인기와 함께 병렬화가 큰 주목을 받고 있으며 GPU는 병렬 연산을 위한 대표적인 장치
      - ◆ GPU 각각의 코어는 CPU보다 훨씬 더 느리지만 GPU의 코어는 수천여 개로 구성되어 있어 많아 봐야 수십여 개에 불과한 CPU 보다 수백 배 더 많은 연산을 동시에 수행할 수 있는데 CPU가 비행기로 짐을 여러 번 나르는 것이라면 GPU는 배로 수많은 짐을 한 번에 나르는 것에 비유할 수 있음
      - ◆ 각 배의 운반 속도는 훨씬 느리지만 비행기보다 더 많은 짐을 동시에 나를 수 있는 것처럼 GPU는 결국 같은 시간에 목적지에 훨씬 더 많은 짐을 나를 수 있음
      - ◆ 딥 러닝 알고리즘을 비롯해 병렬 연산이 가능한 알고리즘들은 최근에 큰 주목을 받고 있는데 알고리즘 자체의 시간 복잡도 외에도 알고리즘이 병렬화가 가능한지는 근래에 알고리즘의 우수성을 평가하는 매우 중요한 척도 중 하나이기도 함