3월1주차 개념 정리

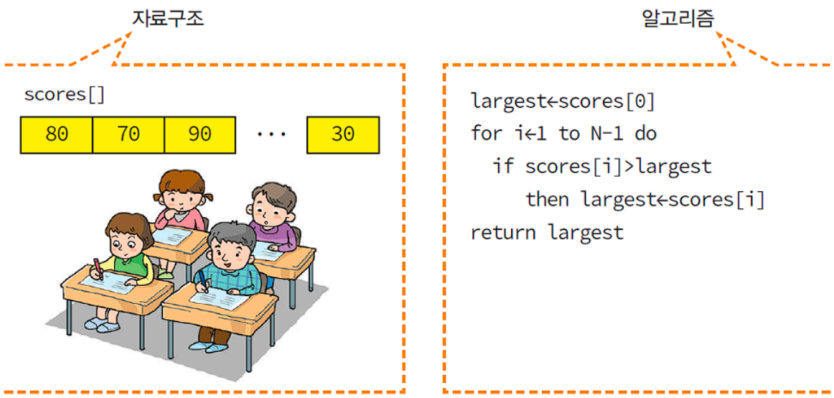
상태	완료
데드라인	@March 27, 2025
PROCESS	💚 데이터구조

▼ 1.1 자료구조와 알고리즘

- 일상과 컴퓨터 자료구조

그릇을 쌓아서 보관하는 것	스택
마트 계산대의 줄	큐
버킷리스트	리스트
영어 사전	사전
지도	그래프
컴퓨터의 디렉토리 구조	트리

- 프로그램 구성
  - 프로그램 = 자료구조 + 알고리즘



▼ 알고리즘의 조건

- 입력 : 0개 이상의 입력이 존재하여야 한다.
- 출력 : 1개 이상의 출력이 존재하여야 한다.
- 명백성 : 각 명령어의 의미는 모호하지 않고 명확해야 한다.
- 유한성 : 한정된 수의 단계 후에는 반드시 종료되어야 한다.
- 유효성 : 각 명령어들은 실행 가능한 연산이어야 한다.

▼ 알고리즘

- algorithm : 컴퓨터로 문제를 풀기 위한 단계적인 절차

▼ 기술방법

▼ 영어나 한국어와 같은 자연어

- 인간이 읽기가 쉽다
- 하지만 자연어의 단어들을 정확하게 정의하지 않으면 전달이 모호해질 우려가 있음

▼ 흐름도 flow chart

- 직관적이고 이해하기 쉬운 알고리즘 기술 방법
- 그러나 복잡한 알고리즘의 경우, 상당히 표현하기 길고 복잡해짐

▼ 의사코드 pseudo-code

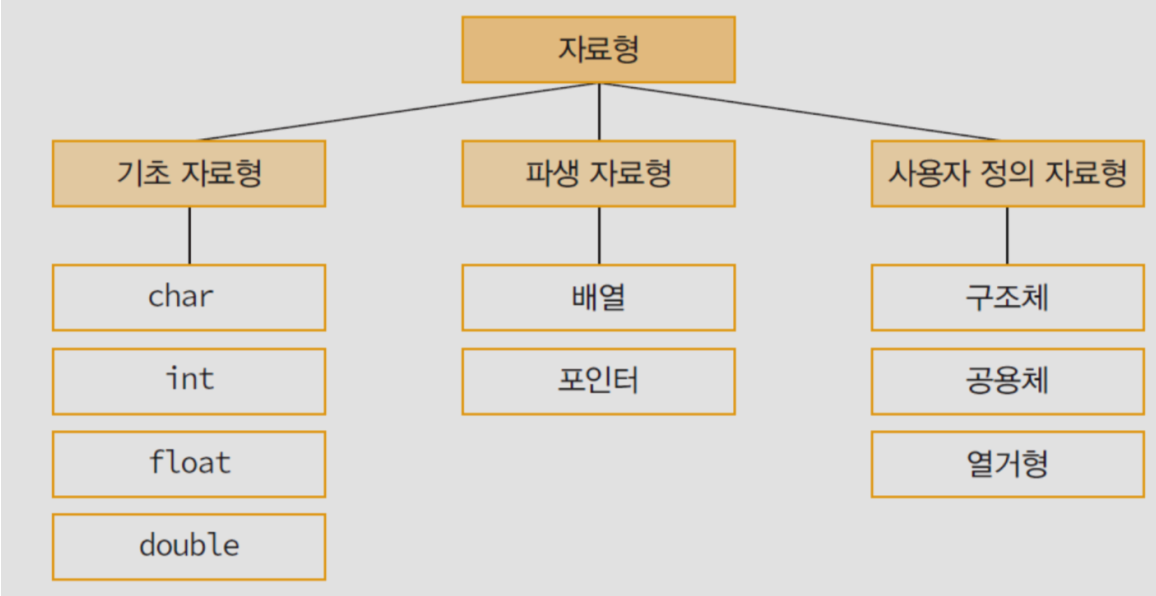
- 알고리즘 기술에 가장 많이 사용됨
- 프로그램을 구현할 때의 여러 가지 문제들을 감출 수 있음. 즉 알고리즘의 핵심적인 내용에만 집중할 수 있음

▼ 프로그래밍 언어

- 알고리즘의 가장 정확한 기술이 가능
- 근데 실제 구현 시, 많은 구체적인 사항들이 알고리즘의 핵심적인 내용에 대한 이해를 방해할 수 있음

▼ 1.2 추상자료형 ADT(Abastract Data Type)

- data type : 데이터의 종류 - 정수, 실수, 문자열 등이 기초적인 자료형의 예
- 데이터의 집합과 연산의 집합
  - int 자료형 = 데이터 + 연산

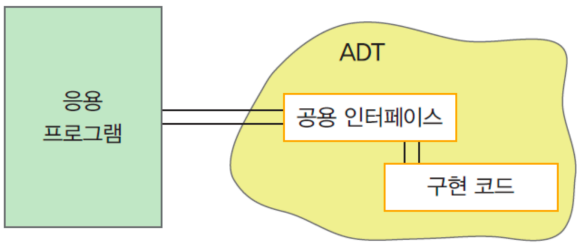


▼ 추상 데이터 타입

- 데이터 타입을 추상적으로 정의한 것
- 데이터나 연산이 무엇인가는 정의되지만 데이터나 연산을 어떻게 컴퓨터 상에서 구현할 것인지는 정의되지 않음

▼ 유래

- abstraction → information hiding → ADT
- 추상화 : 자료형에 대한 간략한 기술, 사용자에게 중요한 정보는 강조되고 반면 중요하지 않은 구현 세부사항은 제거(정보 은닉)하는 것



▼ 추상 데이터 타입의 정의

- 객체 : 추상 데이터 타입에 속하는 객체가 정의됨
- 연산 : 이들 객체들 사이의 연산이 정의. 이 연산은 추상 데이터 타입과 외부를 연결하는 인터페이스의 역할을 함
- 예시 : 자연수, 스택, ...

TV의 인터페이스가 제공하는 특정한 작업만을 할 수 있다.	사용자들은 ADT가 제공하는 연산만을 사용할 수 있다.
사용자는 TV 내부를 볼 수 없다.	사용자들은 ADT 내부의 데이터를 접근할 수 없다.
TV의 내부에서 무엇이 일어나고 있는지를 몰라도 이용할 수 있다.	사용자들은 ADT가 어떻게 구현되닌지 모르더라도 사용할 수 있다.

▼ 1.3 알고리즘의 성능 분석

▼ 수행 시간 측정

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

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

int main(){
    clock_t start, stop;
    double duration;
    start = clock();
    for(int i = 0; i < 10000000; i++) ... //의미 없는 반복 루프

    stop = clock();
    duration = (double) (stop - start) / CLOCK_PER_SEC;
    printf("수행 시간은 %f초 입니다.\n", duration);
    return 0;
}
```

▼ 알고리즘의 복잡도 분석

- 직접 구현하지 않고서도 수행 시간을 분석하는 것
  - 시간 복잡도 time complexity
  - 공간 복잡도 space complexity - 워낙 메모리 용량이 적어서 신경 안쓰지만 특수 경우에 ㅇㅇ
- 알고리즘이 수행하는 연산의 횟수를 측정하여 비교
  - **알고리즘을 이루고 있는 연산들이 몇 번이나 수행되는지를 숫자로 표시**
- 일반적으로 연산의 횟수는 n의 함수

왜 프로그램의 효율성이 중요한가?

컴퓨터의 성능이 예전에 비해 월등히 좋아지더라도, 프로그램의 효율성은 시간과 관련있음

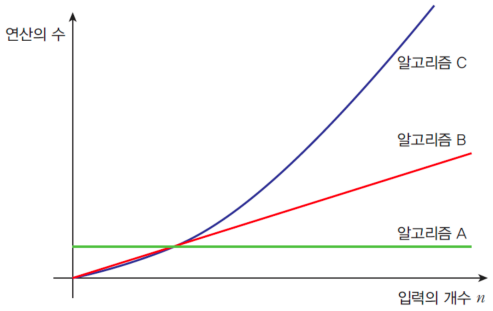
입력 자료의 개수	프로그램 A: n^2	프로그램 B: 2^n
$n = 6$	36초	64초
$n = 100$	10000초	2^{100}초=4×10^{22}년

- ex

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 + 1;

	A	B
대입	1	n
덧셈	1	n
곱셈		
나눗셈		
전체 연산 수	2	2n



▼ 빅오 표기법

- 자료의 개수가 많은 경우에는 차수가 가장 큰 항이 가장 영향을 크게 미침
- 연산의 횟수를 대략적으로 표기한 것

▼ 종류

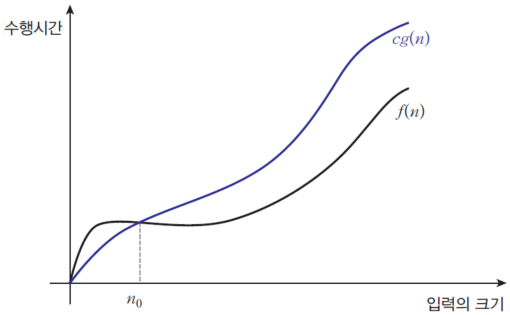
$O(1)$: 상수형
$O(\log n)$: 로그형
$O(n)$: 선형
$O(n\log n)$: 선형로그형
$O(n^2)$: 2차형
$O(n^3)$: 3차형
$O(2^n)$: 지수형
$O(n!)$: 팩토리얼형

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	40326	20922789888000	26313×10^{33}

▼ 그래프 참고

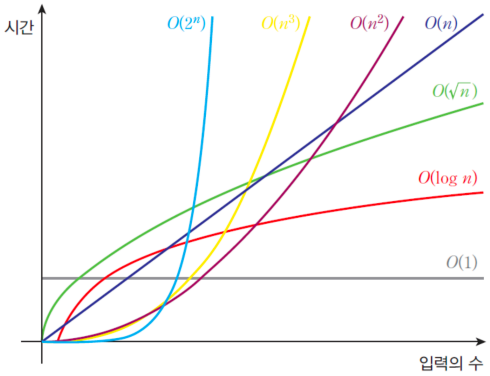
모든 $n \geq n_0$에 대하여 $|f(n)| \leq c|g(n)|$을 만족하는 2개의 상수 c와 n_0가 존재하면 $f(n)=O(g(n))$이다.
- (예) $n \geq 1$ 이면, $f(n)=2n+1 < 6n$ 이므로 $f(n) = O(n)$

① $f(n)=5$ 이면 $O(1)$
($n_0=1$, $c=10$ 일때 n>1 에대해 $5 \leq 10\cdot1$ 이되므로)
② $f(n)=2n+1$ 이면 $O(n)$
($n_0=2$, $c=3$ 일때 $n>2$ 에대해 $2n+1 \leq 3n$ 이 되기 때문)
③ $f(n)=3n^2+100$ 이면 $O(n^2)$
($n_0=100$, $c=5$ 일때 $n>100$ 에대해 $3n^2+100 \leq 5n^2$이 되기 때문)
④ $f(n)=5\cdot2^n+10n^2+100$ 이면 $O(2^n)$
($n_0=1000$, $c=10$ 일때 $n>1000$ 에대해 $5\cdot2^n+10\cdot n^2+100 \leq 10\cdot2^n$ 이 되기때문)



빅오는 함수의 상한을 표시한다.

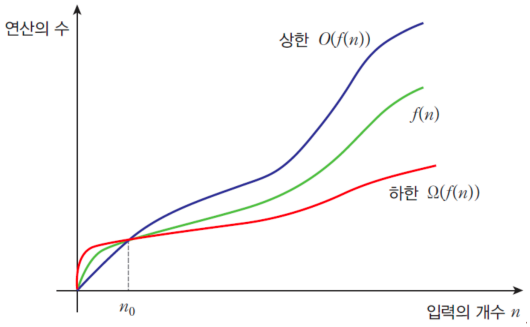
- 복잡도의 상한만 표시함



▼ 빅오메가 표기법

모든 $n \geq n_0$에 대하여 $|f(n)| \geq c|g(n)|$을 만족하는 2개의 상수 c와 n_0가 존재하면, $f(n) = \Omega(g(n))$이다.

- 빅오메가는 함수의 하한을 표시한다.



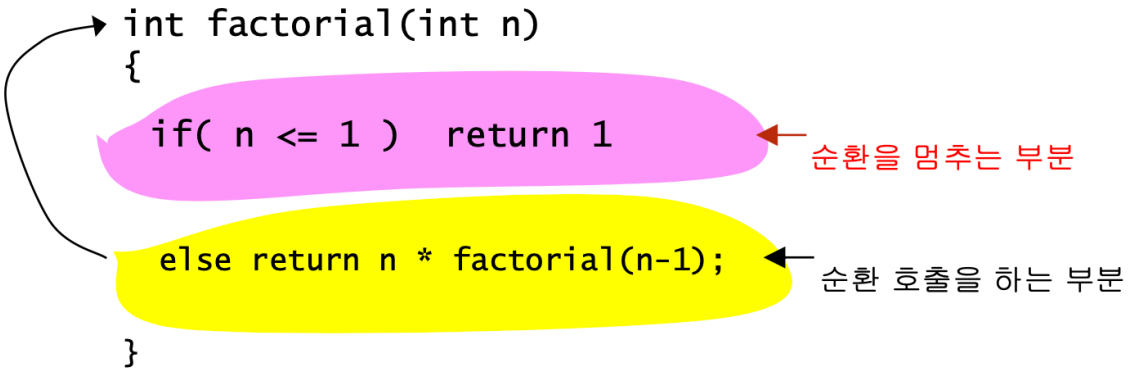
▼ 빅세타 표기법

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_2와 n_0가 존재하면 $f(n)=\theta(g(n))$이다.

- 함수의 하한인 동시에 상한을 표시한다.
- 가장 정밀함

▼ 2.1 순환

- 알고리즘이나 함수가 수행 도중에 자기 자신을 다시 호출하여 문제를 해결
- 정의 자체가 순환적으로 되어 있는 경우에 적합



- 순환 알고리즘을 멈추는 부분이 없다면, 시스템 오류가 발생할 때까지 무한정 호출하게 된다.
- ⇒ 대부분의 순환은 반복으로 바꾸어 작성 가능

▼ 팩토리얼의 반복적 구현

```
int factorial_iter(int n){
   int k, v=1;
   for(k=n;k>0;k--)
      v *= k;

   return v;
}
```

▼ 순환과 반복의 비교

- 일반적으로 순환 프로그래밍 < - > 반복 프로그래밍 변환 가능
- 문제 정의가 순환적인 경우
  - 순환적 프로그래밍이 더 쉽고, 이해하기 편함
  - 반복문 작성 시에 훨씬 코드도 길어지고 이해하기도 힘듦
- 프로그램 효율적인 측면에서는 반복문 유리
  - 순환 프로그래밍 - 함수 호출의 부담 : 활성 레코드 생성/보존, 실행 코드 점프 등
  - 둘다 가능하고 코드 길이가 커지지 않는다면 반복문 ㅇㅇ

▼ 활성 레코드 activation record

- 함수 호출 순서 f1 → f2 → f3
- 호출되는 함수의 각 활성 레코드가 스택 형태로 저장
- 활성 레코드 항목
  - return address : 자신을 호출한 함수의 복귀 주소
  - 레지스터 값들 : 호출된 함수가 실행되기 전에 보존해야 할 CPU 레지스터 값
  - 함수 매개변수 : 해당 함수 호출 시에 전달된 인수
  - 지역 변수 : 함수 내에 사용되는 지역변수

▼ 거듭제곱 값 프로그래밍

- 순환적인 방법이 더 효율적인 예제

```
double slow_power(double x, int n){
   double result = 1.0;
   for(int i = 0; i < n ; i++)
      result = result * x;
```

```
    return result;
}
```

▼ 2.2 순환 프로그래밍 작성 방법

- 선언적Declarative 프로그래밍
  - 프로그래밍 목표를 명시. 구체적인 알고리즘을 명시하지 않음
  - 명령형 프로그래밍에 익숙한 개발자는 어렵댕
- 명령형Imperative 프로그래밍
  - 프로그래밍의 목표보다 구체적인 알고리즘을 명시
- 순환 프로그래밍 : 정의를 이해하고, 이를 재귀적으로 분석
  - 종료 조건
  - 문제의 정의

▼ 거듭제곱 pr - 1, 2

```
power(x,n)

if n==0
   then return 1;
else
   then return x*power(x, n-1);

double power(double x, int n){
   if(n == 0)
      return 1;
   else
      return x*power(x, n-1);
}
```

```
// n = 2 * n/2 이용
power(x,n)

if n==0
   then return 1;
else if n%2 == 0
   then return power(x^2, n/2);
else if n%2 != 0
   then return x*power(x^2, (n-1)/2);

double power(double x, int n){
   if n==0
      return 1;
   else if(n%2 == 0)
      return power(x*x, n/2);
   else
      return x*power(x*x, (n-1)/2);
   }
```

▼ 피보나치 수열 계산

```
int fib(int n){
   if n==0 return 0;
```

- 같은 항 중복 계산
- 비효율 순환!

```
    if n==1 return 1;
    return (fib(n-1) + fib(n-2));
}
```

```
int fib_iter(int n){
    int pp = 0, p=1, result = 0;
    if n==0 return 0;
    if n==1 return 1;

    for int i=2; i<n; i++{
        result = p+pp;
        pp = p;
        p = result;
    }
    return result;
}
```

- 복잡도 함수
  - *T(n) = T(n-1) + T(n-2) + O(1)*
- 시간 복잡도: $O(2^n)$
  - 하나의 피보나치 함수 fib()는 매 스텝 2개의 fib() 함수를 호출
  - 1 -> 2 -> 4 -> 8 -> 16 -> 32 -> 64 -> …

▼ 2.3 하노이탑

```
#include <stdio.h>

void hanoi_tower(int n, char from, char tmp, char to){
    if(n==1)
        printf("원판 1을 %c에서 %c로 이동함 \n", from, to);
    else{
        hanoi_tower(n-1, from, to, tmp);
        printf("원판 %d를 %c에서 %c로 이동함 \n", n, from, to);
        hanoi_tower(n-1, tmp, from, to);
    }
    return ;
}

int main(){
    int n;
    printf("탑 높이를 입력하세요 : ");
    scanf("%d", &n);
    printf("\n\n");

    hanoi_tower(n, 'A', 'B', 'C');
```

```
    return 0;
}
```