Section 1

자료구조와 알고리즘

동국대학교

AI융합대학 AI소프트웨어융합학부

조교수 선석규

sukkyu.sun@dgu.ac.kr

1. 자료구조의 이해 : 분류

❖ 자료의 형태에 따른 분류와 이 책에서 다루는 세부 주제

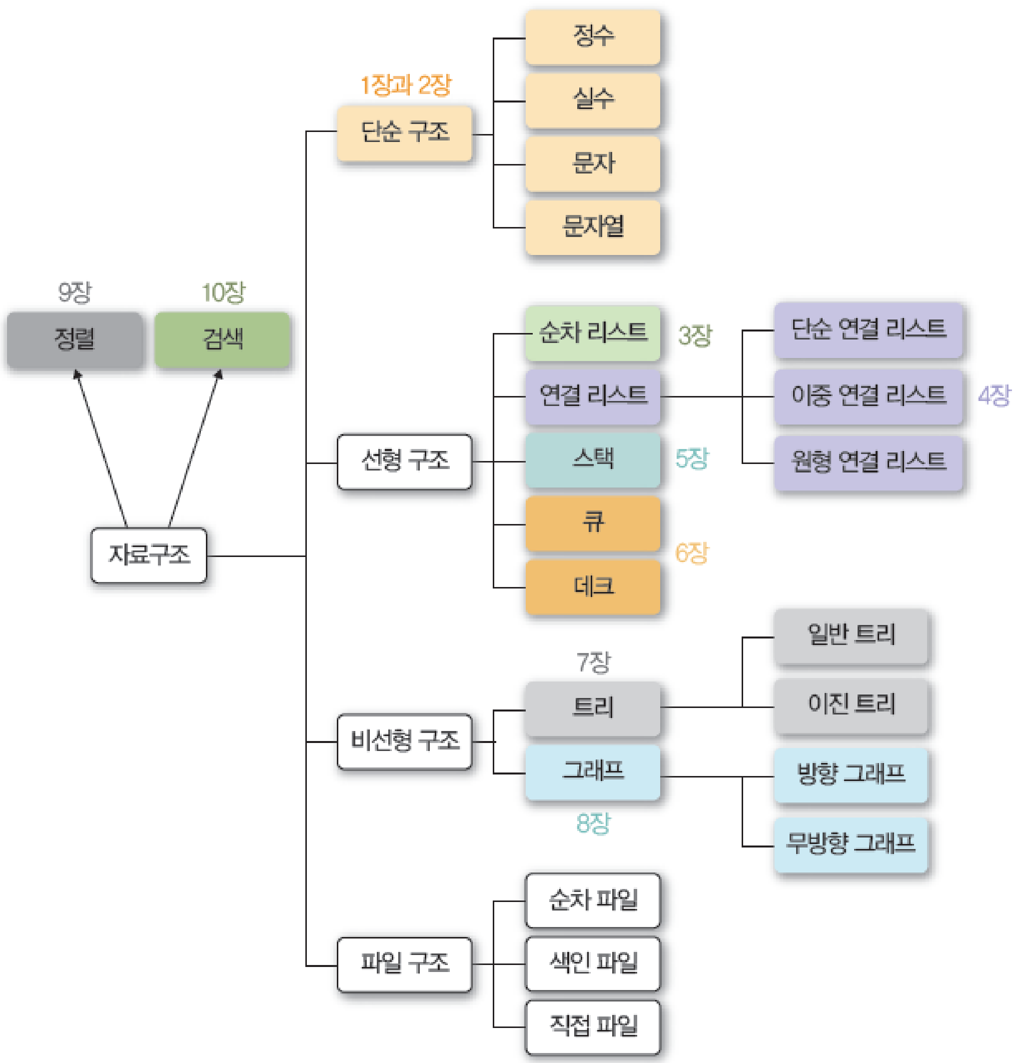
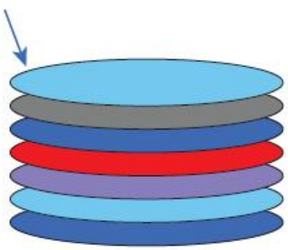


그림 1-4 자료구조의 형태에 따른 분류와 이 책에서 다루는 세부 주제

일상생활에서의 사물의 조직화



큐의 끝 큐의 처음

일상생활과 자료구조의 비교

〈표 1-1〉 일상생활과 자료구조의 유사성

일상생활에서의 예	해당하는 자료구조
그릇을 쌓아서 보관하는 것	스택
마트 계산대의 줄	큐
버킷 리스트	리스트
영어사전	사전
지도	그래프
컴퓨터의 디렉토리 구조	트리

자료구조와 알고리즘

- 프로그램 = 자료구조 + 알고리즘

자료구조

scores[]

80	70	90	· · ·	30

알고리즘

```
largest←scores[0]
for i←1 to N-1 do
  if scores[i]>largest
     then largest←scores[i]
return largest
```

```c
#define MAX_ELEMENTS 100
int scores[MAX_ELEMENTS];           // 자료구조
int get_max_score(int n)   // 학생의 숫자는 n
{
        int i, largest;
        largest = scores[0];              // 알고리즘
        for (i = 1; i<n; i++) {
                if (scores[i] > largest) {
                        largest = scores[i];
                }
        }
        return largest;
}
```

동국대학교
dongguk university

알고리즘의 조건

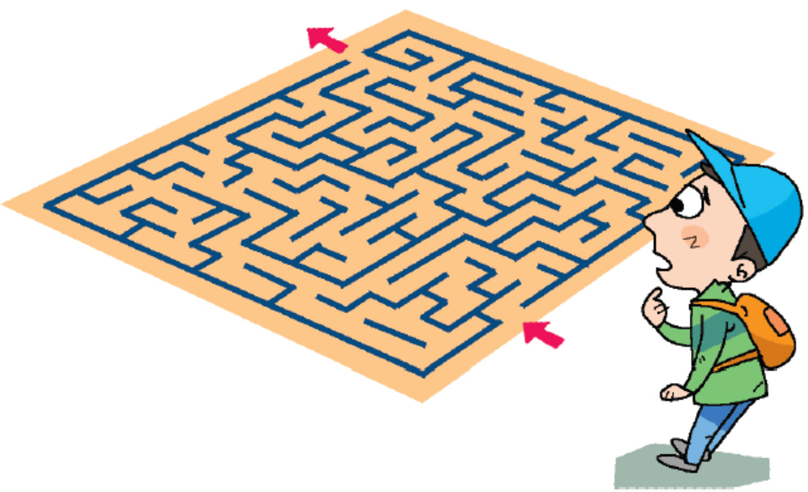
- 알고리즘의 조건
  - 입력 : 0개 이상의 입력이 존재하여야 한다.
  - 출력 : 1개 이상의 출력이 존재하여야 한다.
  - 명백성 : 각 명령어의 의미는 모호하지 않고 명확해야 한다.
  - 유한성 : 한정된 수의 단계 후에는 반드시 종료되어야 한다.
  - 유효성 : 각 명령어들은 실행 가능한 연산이여야 한다.

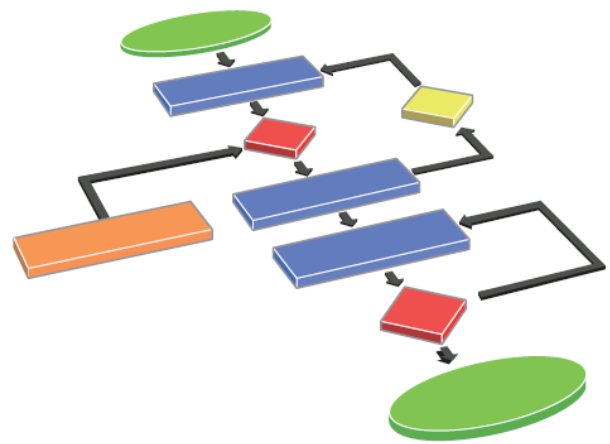
알고리즘

- 알고리즘(algorithm): 컴퓨터로 문제를 풀기 위한 단계적인 절차

알고리즘의 기술 방법

- 영어나 한국어와 같은 자연어
- 흐름도(flow chart)
- 의사 코드(pseudo-code)
- 프로그래밍 언어





자연어로 표기된 알고리즘

- 인간이 읽기가 쉽다.
- 그러나 자연어의 단어들을 정확하게 정의하지 않으면 의미 전달이 모호해질 우려가 있다.

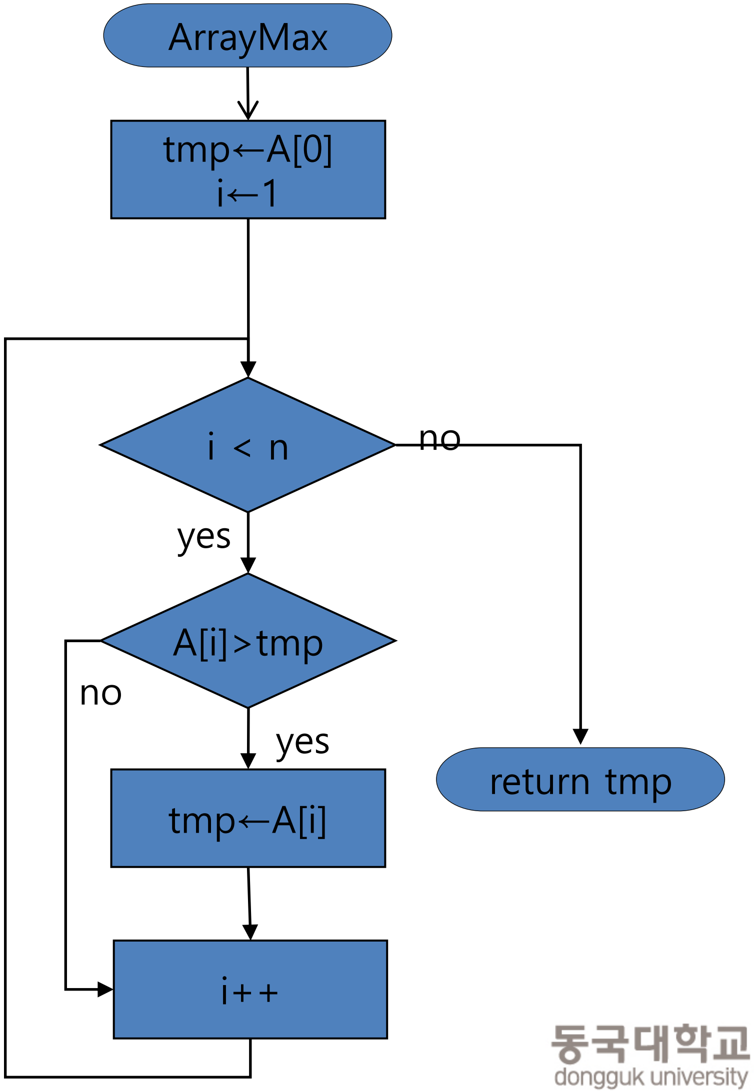
(예) 배열에서 최대값 찾기 알고리즘

```
ArrayMax(list, n)

1.  배열 list의 첫번째 요소를 변수 tmp에 복사
2.  배열 list의 다음 요소들을 차례대로 tmp와 비교하면 더 크면
    tmp로 복사
3.  배열 list의 모든 요소를 비교했으면 tmp를 반환
```

흐름도로 표기된 알고리즘

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



유사코드로 표현된 알고리즘

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

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

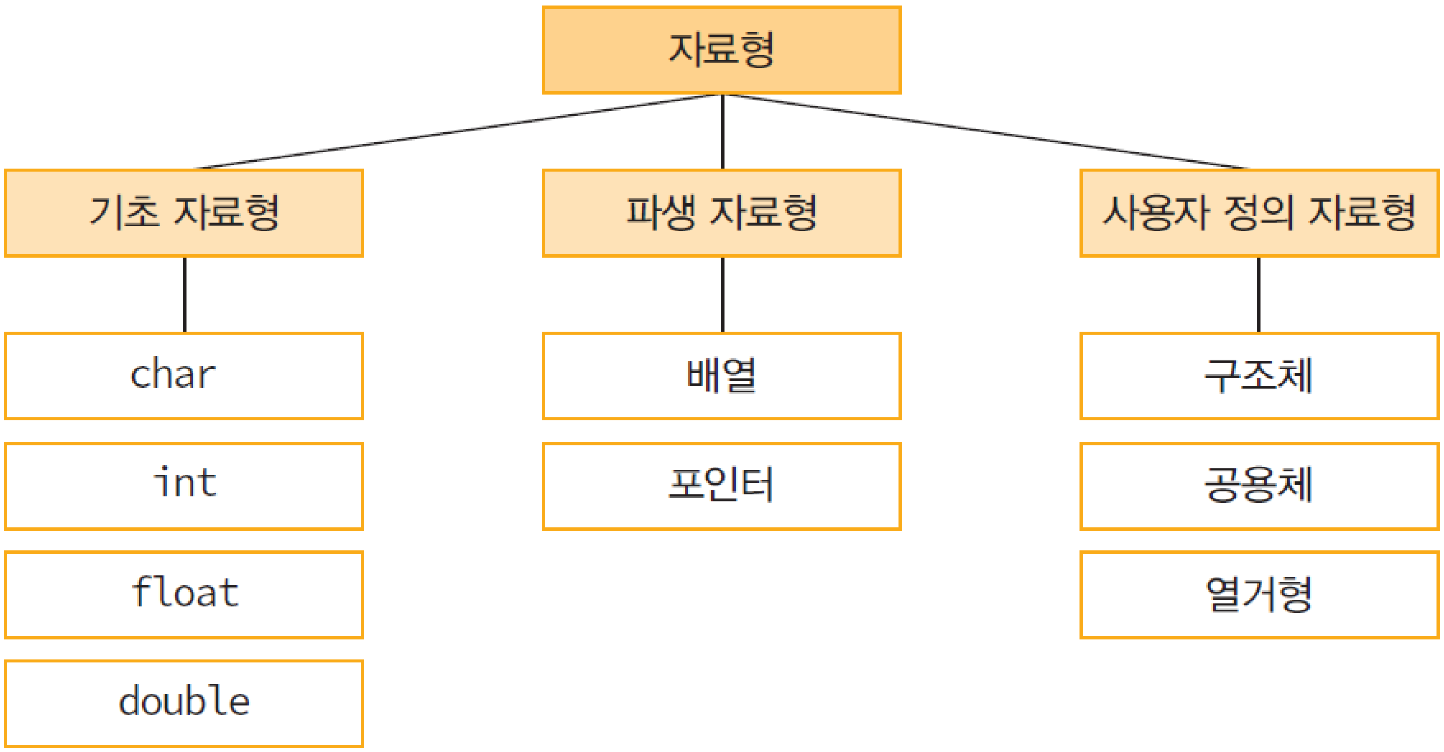
C로 표현된 알고리즘

- 알고리즘의 가장 정확한 기술이 가능
- 반면 실제 구현 시, 많은 구체적인 사항들이 알고리즘의 핵심적인 내용에 대한 이해를 방해할 수 있다.

```
#define MAX_ELEMENTS 100
int score[MAX_ELEMENTS];
int find_max_score(int n)
{
        int i, tmp;
        tmp=score[0];
        for(i=1;i<n;i++){
                if( score[i] > tmp ){
                        tmp = score[i];
                }
        }
        return tmp;
}
```

자료형

- 자료형(data type): "데이터의 종류"
- 정수, 실수, 문자열 등이 기초적인 자료형의 예



자료형

- 자료형(data type)
  - 데이터의 집합과 연산의 집합

int 자료형

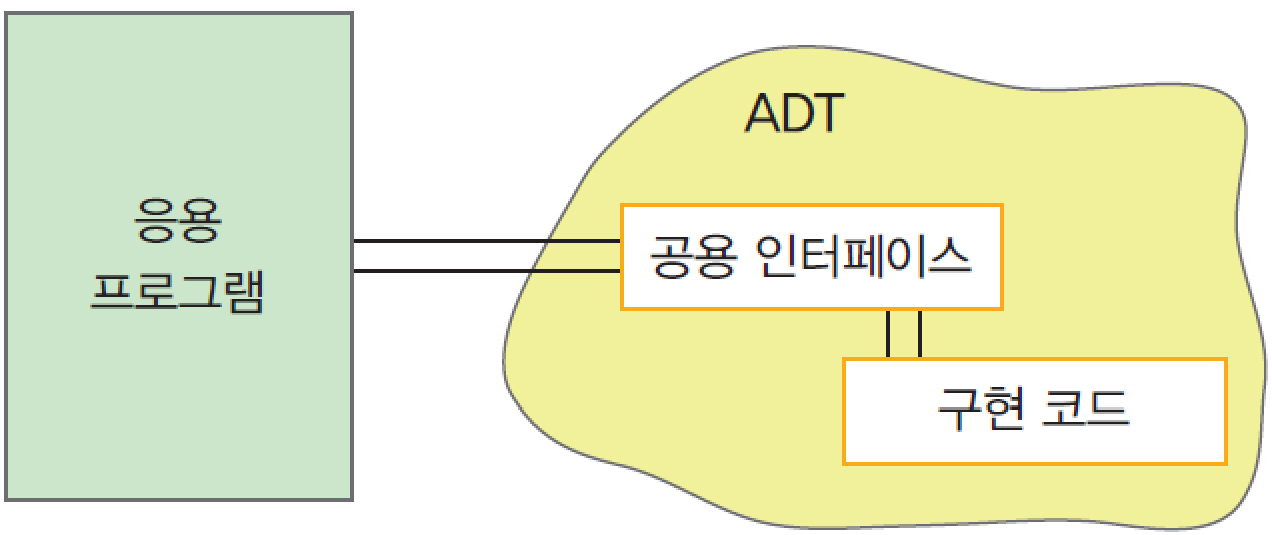
- 데이터: {-INT_MIN, ..., -2, -1. 0, 1, 2, ..., INT_MAX }
- 연산: +, -, *, /, %, ==, >, <

추상 데이터 타입

- 추상 데이터 타입(ADT: Abstract Data Type)
  - 데이터 타입을 추상적(수학적)으로 정의한 것
  - 데이터나 연산이 무엇(what)인가는 정의되지만 데이터나 연산을 어떻게(how) 컴퓨터 상에서 구현할 것인지는 정의되지 않는다.

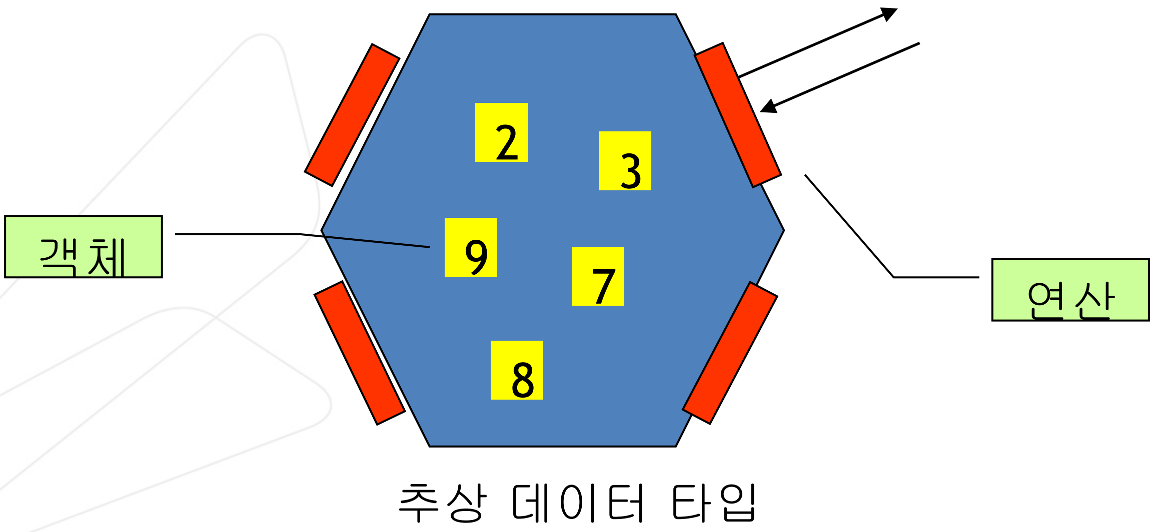
추상데이터타입의 유래

- 추상화(abstraction)-> 정보은닉기법(information hiding)-> 추상자료형(ADT)
- 추상화란 사용자에게 중요한 정보는 강조되고 반면 중요하지 않은 구현 세부 사항은 제거하는 것



추상 데이터 타입의 정의

- **객체**: 추상 데이터 타입에 속하는 객체가 정의된다.
- **연산**: 이들 객체들 사이의 연산이 정의된다. 이 연산은 추상 데이터 타입과 외부를 연결하는 인터페이스의 역할을 한다.



동국대학교 dongguk university

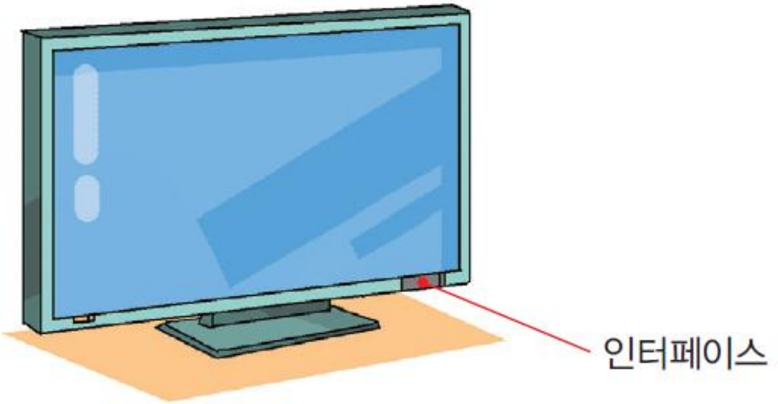
추상 데이터 타입의 예: 자연수

```
Nat_No

객체: 0에서 시작하여 INT_MAX까지의 순서화된 정수의 부분범위

함수:
 Nat_Number zero()         ::= 0
 Nat_Number successor(x) ::= if( x==INT_MAX ) return x
                             else return x+1
 Boolean is_zero(x)         ::= if (x) return FALSE
                              else return TRUE
 Boolean equal(x,y)         ::= if( x==y ) return TRUE
                             else return FALSE
 Nat_Number add(x,y)       ::= if( (x+y) <= INT_MAX )
                              return x+y
                            else return INT_MAX
 Nat_Number sub(x,y)       ::= if ( x<y ) return 0
                             else return x-y;
```

추상 데이터 타입과 TV



▪TV의 인터페이스가 제공하는 특정한 작업만을 할 수 있다.

▪사용자는 TV의 내부를 볼 수 없다.

▪TV의 내부에서 무엇이 일어나고 있는지를 몰라도 이용할 수 있다.

▪사용자들은 ADT가 제공하는 연산만을 사용할 수 있다.

▪사용자들은 ADT 내부의 데이터를 접근할 수 없다.

▪사용자들은 ADT가 어떻게 구현되는지 모르더라도 ADT를 사용할 수 있다.

알고리즘의 성능분석

- 알고리즘의 성능 분석 기법
  - 수행 시간 측정
    - 두개의 알고리즘의 실제 수행 시간을 측정하는 것
    - 실제로 구현하는 것이 필요
    - 동일한 하드웨어를 사용하여야 함
  - 알고리즘의 복잡도 분석
    - 직접 구현하지 않고서도 수행 시간을 분석하는 것
    - 알고리즘이 수행하는 연산의 횟수를 측정하여 비교
    - 일반적으로 연산의 횟수는 n의 함수

왜 프로그램의 효율성이 중요한가?

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

수행시간측정

- 알고리즘을 프로그래밍 언어로 작성하여 실제 컴퓨터상에서 실행시킨 다음, 그 수행시간을 측정

수행시간 측정 2가지 방법

방법 #1	방법 #2

`#include <time.h>`
 	
`start = clock();`
`...`
`stop = clock();`
`double duration = (double)(stop - start) / CLOCKS_PER_SEC;`
	`#include <time.h>`
	
	`start = time(NULL);`
	`...`
	`stop = time(NULL);`
	`double duration = (double) difftime(stop, start);`

수행시간측정

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void)
{
    clock_t start, stop;
    double duration;
    start = clock(); // 측정 시작
    for (int i = 0; i < 1000000; i++)       // 의미 없는 반복 루프
          ;
    stop = clock(); // 측정 종료
    duration = (double)(stop - start) / CLOCKS_PER_SEC;
    printf("수행시간은 %f초입니다.\n", duration);
    return 0;
}
```

복잡도 분석

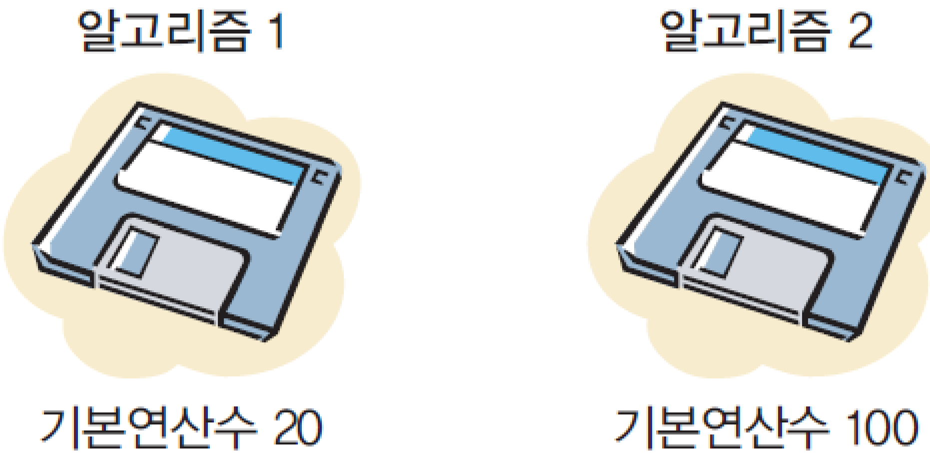
- 시간 복잡도는 알고리즘을 이루고 있는 연산들이 몇 번이나 수행되는지를 숫자로 표시

```
largest←scores[0]
for i←1 to N-1 do
    if scores[i]>largest
        then largest←scores[i]
return largest
```

동국대학교 dongguk university

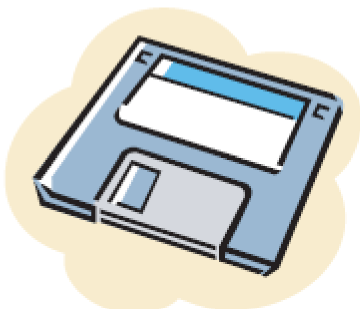
복잡도 분석의 종류

- 시간 복잡도(time complexity)
- 공간 복잡도(space complexity)



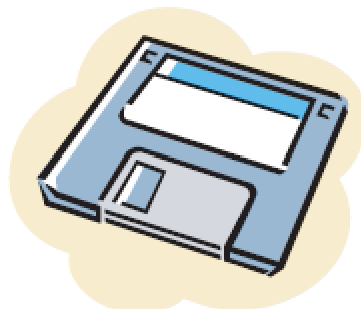
입력의 개수 고려

알고리즘 1



$3n+2$

알고리즘 2



$5n^2+6$

복잡도 분석의 예

- 양의 정수 을 번 더하는 문제를 생각하여 보자.

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;

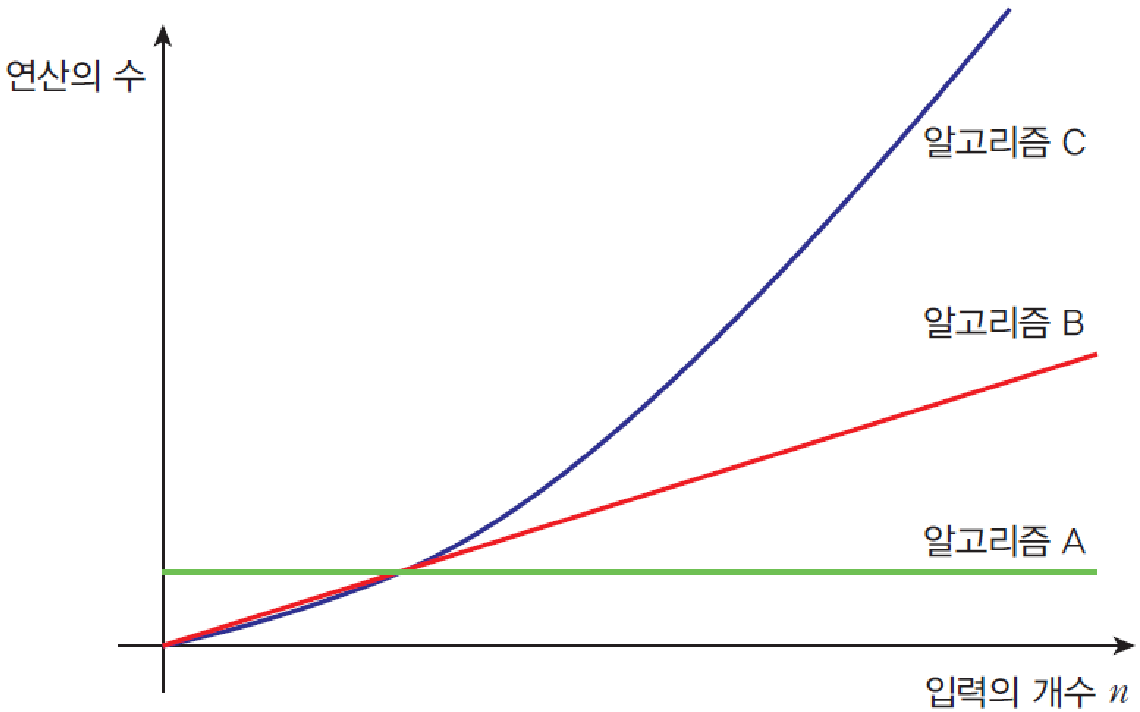
알고리즘의 비교

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;

	알고리즘 A	알고리즘 B	알고리즘 C
대입연산	1	n	$n * n$
덧셈연산		n	$n * n$
곱셈연산	1		
나눗셈연산			
전체연산수	2	$2n$	$2n^2$

연산의 횟수를 그래프로 표현



빅오 표기법

- **자료의 개수가 많은 경우에는 차수가 가장 큰 항이 가장 영향을 크게 미치고 다른 항들은 상대적으로 무시될 수 있다.**

n=1000인 경우

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

99.9%

0.1%

입력의 개수 n

빅오 표기법

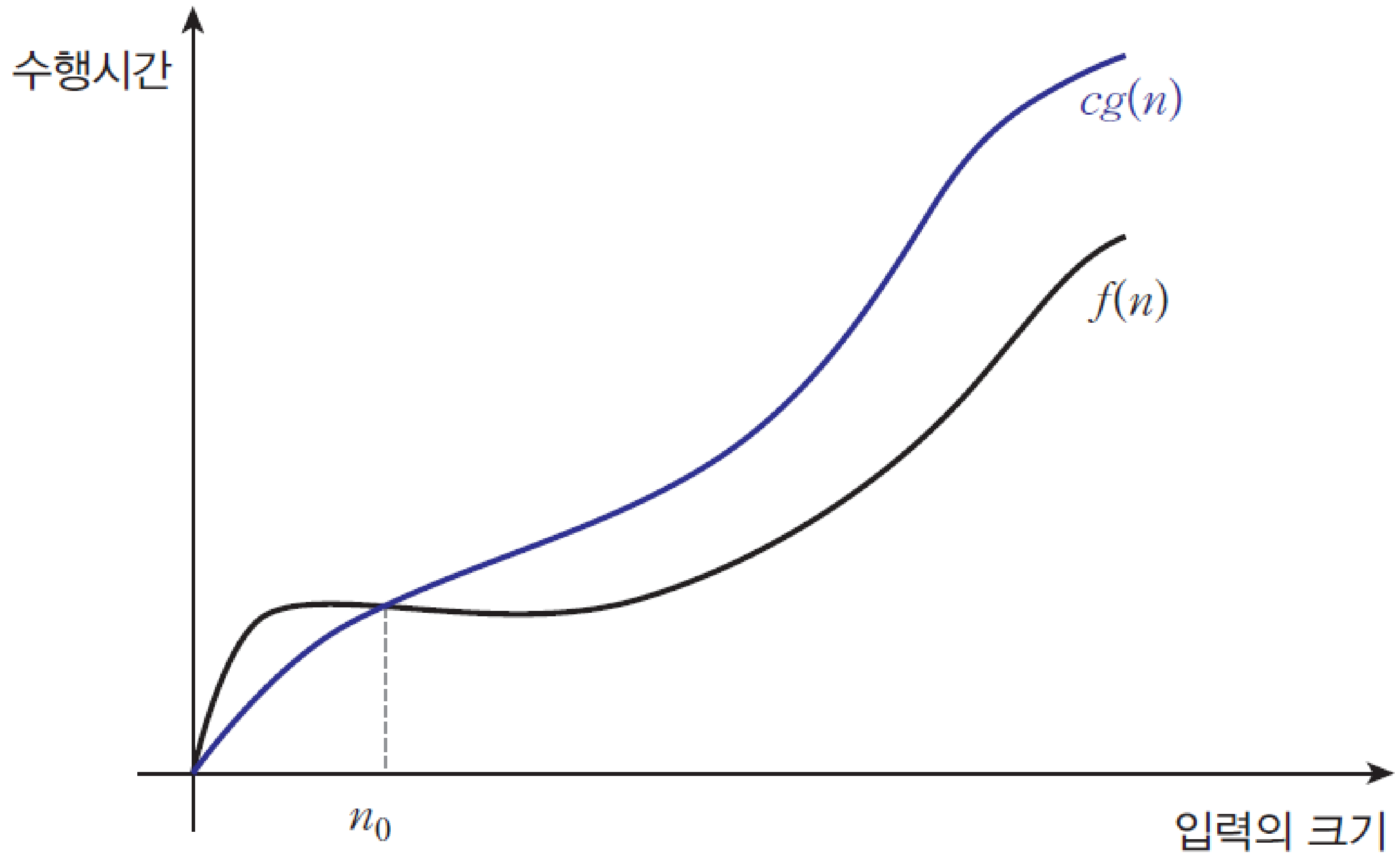
- **빅오표기법**: 연산의 횟수를 대략적(점근적)으로 표기한 것

- 두개의 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 $f(n)=O(g(n))$이다.

- 빅오는 **함수의 상한**을 표시한다.
  - (예) $n \geq 5$ 이면 $2n+1 < 10n$ 이므로 $2n+1 = O(n)$

입력의 개수 n

빅오 표기법



빅오 표기법의 예

예제 1.1 빅오 표기법

- $f(n)=5$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.

빅오 표기법의 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

$f(n)$	$O(f(n))$
10	$O(1)$
$5n^2+6$	$O(n^2)$
$2n^3+1$	$O(n^3)$
$2n^3+5n^2+6$	$O(n^3)$

계산 복잡도 참고 사이트

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

https://www.bigocheatsheet.com/

빅오 표기법의 종류

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}

Big-O 예시

```
int doNothing(int N)
{
   int i, total = 0;
   if (N < 0)
      return N;
   else
   {
      for (i = 1; i <= N; i++)
            total += i;
      return total;
   }
```

O(N)

If~else문은 if나 else 구문 중 실행 시간이 많이 걸리는 시간 복잡도를 갖는다.

```
int doSomething(int N)
{
   int total = 0;

   for (int i = 1; i <= N; i++)
      for(int j = 1; j <= N; j++)
            total += i;

   return total;
}
```

O(N²)

- 버블정렬(bubble sort)
- 선택정렬(seletion sort)
- 삽입정렬(insertion sort)

Big-O 예시

```
int doSomething(int N, int M)
{
   int total = 0;

   for (int i = 1; i <= N; i++)
      for(int j = 1; j <= M; j++)
         total += (i + j);

   return total;
}
```

O(NM)

N,M 이 다른경우 NM의 복잡도를 갖기도한다.

Big-O 예시

```
int doSomething(int N)
{
    int total = 0;

    for (int i = N; i > 0; i /= 2)
    {
        total += i;
    }

    return total;
}
```

O(logN)

- 이진 검색(binary search)

빅오표기법

```
int doSomething(int N)
{
   int total = 0;

   for (int i = 1; i <= N; i++)
      for(int j = 1; j <= N; j++)
         total += i;

   return total;
}
```

$O(N^2)$

- 버블정렬(bubble sort)
- 선택정렬(seletion sort)
- 삽입정렬(insertion sort)

빅오 표기법이외의 표기법

- 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.
  - (예) $n \geq 5$ 이면 $2n+1 < 10n$ 이므로 $n = \Omega(n)$

빅오 표기법이외의 표기법

- 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.

  - 빅세타는 함수의 하한인 동시에 상한을 표시한다.
  - $f(n)=O(g(n))$이면서 $f(n)= \Omega(g(n))$이면 $f(n)= \theta(n)$이다.
  - (예) $n \geq 1$이면 $n \leq 2n+1 \leq 3n$이므로 $2n+1 = \theta(n)$

동국대학교 dongguk university

```c
int doNothing(int N)
{
   int i, total = 0;
   if (N < 0)
      return N;
   else
   {
      for (i = 1; i <= N; i++)
            total += i;
      return total;
   }
}
```

O(N)

If~else문은 if나 else 구문 중 실행 시간이 많이 걸리는 시간 복잡도를 갖는다.

최선, 평균, 최악의 경우

- 알고리즘의 수행시간은 입력 자료 집합에 따라 다를 수 있다.

- 최선의 경우(best case): 수행 시간이 가장 빠른 경우
- 평균의 경우(average case): 수행시간이 평균적인 경우
- 최악의 경우(worst case): 수행 시간이 가장 늦은 경우

```
int doSomething(int N, int M)
{
   int total = 0;

   for (int i = 1; i <= N; i++)
      for(int j = 1; j <= M; j++)
         total += (i + j);

   return total;
}
```

O(NM)

동국대학교 dongguk university

(예) 최선, 평균, 최악의 경우

- (예) 순차탐색
- 최선의 경우: 찾고자 하는 숫자가 맨 앞에 있는 경우

  ∴ O(1)

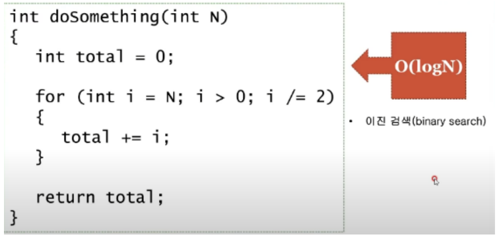
- 최악의 경우: 찾고자 하는 숫자가 맨 뒤에 있는 경우

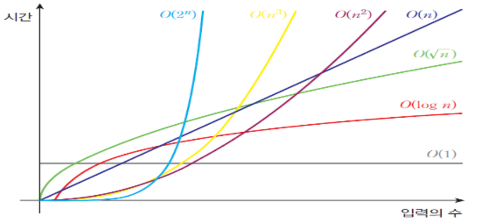
  ∴ O(n)

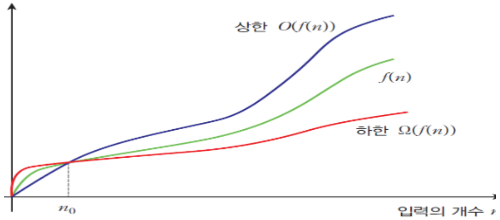
- 평균적인 경우: 각 요소들이 균일하게 탐색된다고 가정하

  (1+2+⋯+n)/n=(n+1)/2

  ∴ O(n)







최선, 평균, 최악의 경우

- 최선의 경우: 의미가 없는 경우가 많다.
- 평균적인 경우: 계산하기가 상당히 어려움.
- 최악의 경우: 가장 널리 사용된다. 계산하기 쉽고 응용에 따라서 중요한 의미를 가질 수도 있다.
  - (예) 비행기 관제업무, 게임, 로보틱스

Q & A