

Algo-01-AlgorithmsIntro

2020년 8월 28일 금요일 오후 3:27



Algo-01-AlgorithmsIntro

1장 알고리즘: 효율성, 분석, 차수

책 소개

- 알고리즘 기초(Foundations of Algorithms)
- 리차드 네아폴리탄 저, 도경구 역
- 홍릉과학출판사
- 주요 내용: 컴퓨터로 문제 푸는 기법 배우기

목차

- 1장: 알고리즘: 효율성, 분석, 차수

- 2장 - 6장: 다양한 문제풀이 기법 및 적용 예제
 - 2장 분할정복
 - 3장 동적계획
 - 4장 탐욕 알고리즘
 - 5장 되추적
 - 6장 분기한정법

- 7장 계산복잡도 소개: 정렬문제
- 8장 계산복잡도: 검색문제
- 9장 계산복잡도와 문제 난이도: NP 이론 소개

1장 주요 내용

1. 알고리즘

1. 효율적인 알고리즘 개발 중요성

1. 알고리즘 분석

1. 차수

1절 알고리즘

알고리즘이란?

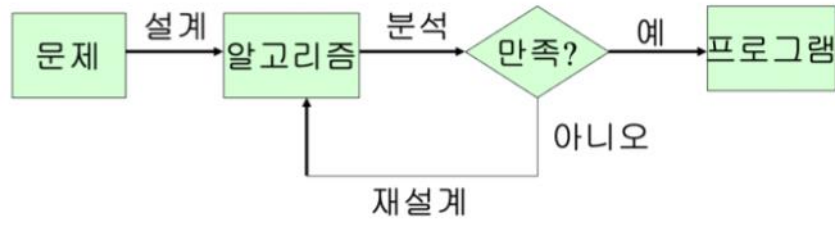
- 컴퓨터를 이용하여 주어진 문제를 해결하는 기법
- 프로그래밍 언어, 프로그래밍 스타일과 무관
- 컴퓨터 프로그램은 여러 방법 중에서 한 가지 방법을 선택하여 구현
- ~~절차: 문제해결 알고리즘 적용 순서~~

↓
다음 슬라이드 내용 중복

알고리즘과 절차

- 절차: 문제해결 알고리즘 적용 순서

프로그램 설계 과정



알고리즘 효율성 분석

- 효율성: 문제해결을 위한 필수 요소
 - 컴퓨터 속도, 메모리 가격과 무관
 - 수천년, 수만년 동안 실행되어야 끝나는 비효율적 알고리즘이 일반적임.

- 분석: 알고리즘의 효율성 판단
 - 효율성 판단 기준: 계산복잡도
 - 계산복잡도
 - 시간복잡도: 특정 연산의 실행 횟수
 - 공간복잡도: 메모리 공간 사용 정도

- 차수: 계산복잡도 판단 기준
 - 계산복잡도 함수의 차수(order) 기준
 - 차수를 이용하여 알고리즘을 계산복잡도별로 분류 가능

알고리즘 효율성 비교 예제

- 문제: 전화번호부에서 '홍길동'의 전화번호 찾기
- 알고리즘 1: 순차검색
 - 첫 쪽부터 '홍길동'이라는 이름이 나올 때까지 순서대로 찾는다.
- 알고리즘 2: 이분검색
 - 전화번호부는 '가나다'순
 - 먼저 'ㅎ'이 있을 만한 곳을 적당히 확인
 - 이후 앞뒤로 뒤적여가며 검색

분석: 어떤 알고리즘이 더 효율적인가?

- 이분검색이 보다 효율적임.

알고리즘 표기법

- 자연어: 한글 또는 영어
 - 단점 1: 복잡한 알고리즘 설명과 전달 어려움
 - 단점 2: 실제로 구현하기 어려움

- 의사코드(Pseudo-code)
 - 실제 프로그래밍 언어와 유사한 언어로 작성된 코드
 - 자연어 사용의 단점 해결
 - 하지만 직접 실행할 수 없음.
 - 교재: C++에 가까운 의사코드 사용

강의에 사용되는 언어: 파이썬3

- 설치: 아나콘다(Anaconda) 패키지 설치 추천
- 주피터 노트북 활용
- 파이썬은 기본패키지만 사용

파이썬 활용의 장점

- 의사코드 수준의 프로그래밍 작성 가능
- 책의 의사코드와 매우 유사하게 구현하여 실행 가능

예제: 순차검색

- 문제: 리스트(배열) S 에 x 가 항목으로 포함되어 있는가?
 - 입력 파라미터: 리스트(배열) S 와 값 x
 - 리턴값: x 가 S 의 항목일 경우 x 의 인덱스, 항목이 아닐 경우 -1.

- 알고리즘 (자연어):
 - x 와 같은 항목을 찾을 때까지 S 에 있는 모든 항목을 차례로 검사
 - 만일 x 와 같은 항목을 찾으면 항목의 인덱스 내주기
 - S 를 모두 검사하고도 찾지 못하면 -1 내주기

In [2]: # 순차검색 알고리즘

```
def seqsearch(S, x):  
    location = 0  
  
    # while 반복문 실행횟수 확인용  
    loop_count = 0  
  
    while location < len(S) and S[location] != x:  
        loop_count += 1  
        location += 1  
  
    if location < len(S):  
        return (location, loop_count)  
    else:  
        return (-1, loop_count)
```

```
In [77]: seq = list(range(30))  
        val = 5  
  
        print(seqsearch(seq, val))  
  
(5, 5)
```

```
In [78]: seq = list(range(30))  
        val = 10  
  
        print(seqsearch(seq, val))  
  
(10, 10)
```

```
In [79]: seq = list(range(30))  
        val = 20  
  
        print(seqsearch(seq, val))  
  
(20, 20)
```

```
In [80]: seq = list(range(30))  
        val = 29  
  
        print(seqsearch(seq, val))  
  
(29, 29)
```

```
In [81]: seq = list(range(30))
        val = 30

        print(seqsearch(seq, val))

(-1, 30)
```

```
In [82]: seq = list(range(30))
        val = 100

        print(seqsearch(seq, val))

(-1, 30)
```

- 입력값의 위치에 따라 while 반복문의 실행횟수가 **선형적** 달라짐.

↓
선형적으로

파이썬튜터 활용: 순차검색

- 위 순차검색 코드를 [PythonTutor: 순차검색](http://pythontutor.com/visualize.html#code=%23%20EC%88%9C%EC%B0%A8%F1,%20loop_count%29%0A%0Aseq%20%3D%20list%28range%2830%29%29%0Aval)
(http://pythontutor.com/visualize.html#code=%23%20EC%88%9C%EC%B0%A8%F1,%20loop_count%29%0A%0Aseq%20%3D%20list%28range%2830%29%29%0Aval)

순차검색 분석

- 특정 값의 위치를 확인하기 위해서 S 의 항목 몇 개를 검색해야 하는가?
 - 특정 값과 동일한 항목의 위치에 따라 다름
 - 최악의 경우: S 의 길이, 즉, 항목의 개수
- 좀 더 빨리 찾을 수는 없는가?
 - S 에 있는 항목에 대한 정보가 없는 한 더 빨리 찾을 수 없음.

2절 효율적 알고리즘 개발 중요성

효율적 검색 알고리즘 예제: 이분검색

- 문제: 항목이 비내림차순(오름차순)으로 정렬된 리스트(배열) S 에 x 가 항목으로 포함되어 있는가?
 - 입력 파라미터: 리스트(배열) S 와 값 x
 - 리턴값: x 가 S 의 항목일 경우 $\overset{\text{↑}}{\text{인덱스}}$, 항목이 아닐 경우 -1 .

- 알고리즘 (자연어):
 - S 의 중간에 위치한 항목과 x 를 비교
 - 만일 x 와 같으면 해당 항목의 인덱스 내주기
 - 만일 x 가 중간에 위치한 값보다 작으면 중간 왼편에 위치한 구간에서 새롭게 검색
 - 만일 x 가 중간에 위치한 값보다 크면 중간 오른편에 위치한 구간에서 새롭게 검색
 - 검색 구간의 크기가 0이 될 때까지 위 절차 반복
 - x와 같은 항목을 찾거나 검색 구간의 크기가 0이 될 때까지 반복

In [3]: # 이분검색 알고리즘

```
def binsearch(S, x):  
    low, high = 0, len(S)-1  
    location = -1  
  
    # while 반복문 실행횟수 확인용  
    loop_count = 0  
  
    while low <= high and location == -1:  
        loop_count += 1  
        mid = (low + high)//2  
  
        if x == S[mid]:  
            location = mid  
        elif x < S[mid]:  
            high = mid - 1  
        else:  
            low = mid + 1  
  
    return (location, loop_count)
```

```
In [70]: seq = list(range(30))  
        val = 5  
  
        print(binsearch(seq, val))  
  
(5, 5)
```

```
In [71]: seq = list(range(30))  
        val = 10  
  
        print(binsearch(seq, val))  
  
(10, 3)
```

```
In [72]: seq = list(range(30))  
        val = 20  
  
        print(binsearch(seq, val))  
  
(20, 4)
```

```
In [73]: seq = list(range(30))  
        val = 29  
  
        print(binsearch(seq, val))  
  
(29, 5)
```

```
In [74]: seq = list(range(30))
        val = 30

        print(binsearch(seq, val))

(-1, 5)
```

```
In [75]: seq = list(range(30))
        val = 100

        print(binsearch(seq, val))

(-1, 5)
```

- 입력값이 달라져도 while 반복문의 실행횟수가 거의 변하지 않음.

파이썬튜터 활용: 이분검색

- [illegible]

이분검색 분석

- 이분검색으로 특정 값의 위치를 확인하기 위해서 S 의 항목 몇 개를 검색해야 하는가?
 - while 반복문이 실행될 때마다 검색 대상의 총 크기가 절반으로 감소됨.
 - 따라서 최악의 경우 $\lg n + 1$ 개의 항목만 검사하면 됨.
 - 여기서 $\lg := \log_2$.

순차검색 vs 이분검색

배열의 크기	순차검색	이분검색
n	n	$\lg n + 1$
128	128	8
1,024	1,024	11
1,048,576	1,048,576	21
4,294,967,296	4,294,967,296	33

이분검색 활용

- 다음, 네이버, 구글, 페이스북, 트위터 등등 수백에서 수천만의 회원을 대상으로 검색을 진행하고자 한다면 어떤 알고리즘 선택?

당연히 이분검색!

- 순차 검색에 비해 이진 검색은 검색 속도가 매우 빠르기 때문에 많은 양의 데이터에서 검색이 필요할 경우 활용됨

예제: 피보나찌 수 구하기 알고리즘

- 피보나치 수열 정의

$$f_0 = 0$$

$$f_1 = 1$$

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

- 피보나찌 수 예제

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

피보나찌 수 구하기 알고리즘(재귀)

- 문제: 피보나찌 수열에서 n 번째 수를 구하라.
 - 입력: 음이 아닌 정수
 - 출력: n 번째 피보나찌 수

```
In [84]: # 피보나찌 수 구하기 알고리즘(재귀)

def fib(n):
    if (n <= 1):
        return n
    else:
        return fib(n-1) + fib(n-2)
```

```
In [55]: fib(3)
```

```
Out[55]: 2
```

```
In [56]: fib(6)
```

```
Out[56]: 8
```

```
In [57]: fib(10)
```

```
Out[57]: 55
```

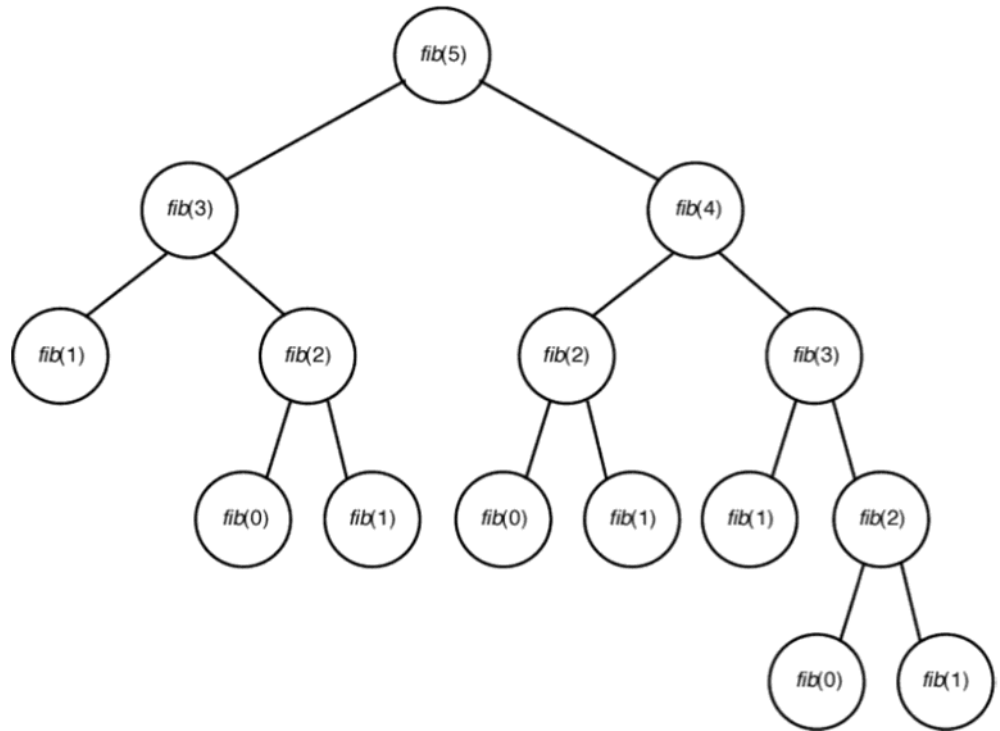
```
In [60]: fib(13)
```

```
Out[60]: 233
```

fib 함수 분석

- 작성하기도 이해하기도 쉽지만, 매우 비효율적임.
- 이유는 동일한 값을 반복적으로 계산하기 때문.

- 예를들어, $\text{fib}(5)$ 를 계산하기 위해 $\text{fib}(2)$ 가 세 번 호출됨. 아래 나무구조 그림 참조.



fib 함수 호출 횟수

- $T(n) = \text{fib}(n)$ 을 계산하기 위해 fib 함수를 호출한 횟수. 즉, fib(n) 을 위한 재귀 나무구조에 포함된 마디(node)의 개수
- 아래 부등식 성립.

$$T(0) = 1$$

$$T(1) = 1$$

$$T(n) = T(n-1) + T(n-2) + 1 \quad (n \geq 2)$$
$$> 2 \times T(n-2) \quad (T(n-1) > T(n-2))$$

$$> 2^2 \times T(n-4)$$

$$> 2^3 \times T(n-6)$$

...

$$> 2^{n/2} \times T(0)$$

$$= 2^{n/2}$$

- 증명
 - 수학적 귀납법 활용
 - 교재 14쪽, 정리 1.1 참조.

정리 11

- 재귀적 알고리즘으로 구성한 재귀 나무구조의 마디의 수를 $T(n)$ 이라고 하면, $n \geq 2$ 인 모든 n 에 대하여 다음이 성립한다.

$$T(n) > 2^{n/2}$$

- 증명: (n 에 대한 수학적 귀납법으로 증명)

- 귀납출발점:

$$\begin{aligned}
 & \blacksquare T \\
 & (2 \\
 &) \\
 & = T \\
 & (1 \\
 &) \\
 & + T \\
 & (0 \\
 &) \\
 & + 1 \\
 & = 3 \\
 & > 2 \\
 & = 2^2
 \end{aligned}$$

↪ 수정 필요

$$\begin{aligned}
 & \blacksquare T \\
 & (3
 \end{aligned}$$

피보나찌 수 구하기 알고리즘 (반복)

- 한 번 계산한 값을 리스트(배열)에 저장해두고 필요할 때 활용.
- 중복 계산 없음.

↪ 한 번 계산한 값을 리스트(배열)에 저장해두면 중복 계산할 필요가 없음

```
In [1]: # 피보나찌 수 구하기 알고리즘 (반복)

def fib2(n):
    f = []

    f.append(0)
    if n > 0:
        f.append(1)
        for i in range(2, n+1):
            fi = f[i-1] + f[i-2]
            f.append(fi)
        return f[n]
```

```
In [2]: fib2(3)
```

```
Out[2]: 2
```

```
In [3]: fib2(6)
```

```
Out[3]: 8
```

```
In [4]: fib2(10)
```

```
Out[4]: 55
```

```
In [5]: fib2(13)
```

```
Out[5]: 233
```

- 중복 계산이 없는 반복 알고리즘은 수행속도가 훨씬 더 빠름.

fib2 함수 분석

- fib2 함수 호출 횟수 $T(n)$
 - $T(n) = n + 1$
 - 즉, $f[0]$ 부터 $f[n]$ 까지 한 번씩만 계산

두 피보나찌 알고리즘의 비교

n	$n+1$	$2^{n/2}$	Iterative	Recursive (Lower bound)
40	41	1,048,576	41 <i>ns</i>	1048 μs
60	61	1.1×10^9	61 <i>ns</i>	1 <i>sec</i>
80	81	1.1×10^{12}	81 <i>ns</i>	18 <i>min</i>
100	101	1.1×10^{15}	101 <i>ns</i>	13 <i>days</i>
120	121	1.2×10^{18}	121 <i>ns</i>	36 <i>years</i>
160	161	1.2×10^{24}	161 <i>ns</i>	3.8×10^7 <i>years</i>
200	201	1.3×10^{30}	201 <i>ns</i>	4×10^{13} <i>years</i>

- $1 \text{ ns} = 10^{-9}$ 초
- $1 \mu s = 10^{-6}$ 초
- 가정: 피보나찌 수 하나를 계산하는 데 걸리는 시간 = 1 ns.

3절 알고리즘 분석

- 설계한 알고리즘의 효율성 분석
- 알고리즘 분석에 사용하는 용어와 표준 분석방법 학습

시간복잡도 분석

- 특정 단위연산이 수행되는 횟수를 입력크기에 대한 함수를 이용한 알고리즘 효율성 분석 기법

↓
입력 크기에 대한 함수로 특정 단위연산이 수행되는 횟수를 측정 하여 알고리즘의 효율성을 분석

입력크기 : 특정 입력값의 크기

- 예제
 - 리스트(배열)의 길이
 - 행렬의 행과 열의 수
 - 나무(트리)의 마디와 이음선의 수
 - 그래프의 정점과 간선의 수
- 주의: 입력과 입력크기는 일반적으로 다름.
 - 피보나찌 함수 `fib`에 사용되는 입력값 n 의 크기는 n 을 이진법으로 표기했을 때의 길이인 $\lg n + 1$ 이다.
 - 예제: $n = 13$ 의 입력크기는 $\lfloor \lg 13 \rfloor + 1 = 4$.

단위연산: 명령문 또는 명령문 덩어리(군) → 알고리즘에 의해 수행된 총 작업은 이 명령문 혹은 명령문 덩어리의 횟수에 비례

- 예제
 - 비교문(comparison)
 - 지정문(assignment)
 - 반복문
 - 모든 기계적 명령문 각각의 실행
 - 예제: PythonTutor의 Step 계산
- 순차검색과 이분검색 알고리즘에서는 비교 while 반복문에 실행되는 명령문들의 덩어리를 단위 연산으로 보았음.
- 피보나치 함수의 경우 함수 본체 전체를 단위연산으로 사용됨.
- 주의의
 - 단위연산을 지정하는 일반적인 규칙 없음.
 - 경우에 따라 두 개의 다른 단위연산을 고려해야 할 수도 있음.
 - 예제: 키를 비교하여 정렬하는 경우, 비교와 지정이 서로 다른 비율로 발생하여, 서로 독립적인 단위연산으로 간주해야 함.
 - 단위연산의 실행횟수가 입력크기뿐만 아니라 입력에도 의존함.

따라서 알고리즘의 시간 복잡도 분석은 입력 크기의 각 값에 대해
단위 연산이 수행되는 횟수를 측정한 것

시간복잡도 종류

- 단위연산 실행횟수가 입력값에 상관없이 입력크기에만 의존하는 경우
 - 일정 시간복잡도: $T(n)$
- 단위연산 실행횟수가 입력값과 입력크기 모두에 의존하는 경우
 - 최악 시간복잡도: $W(n)$
 - 평균 시간복잡도: $A(n)$
 - 최선 시간복잡도: $B(n)$

일정 시간복잡도

- 일정 시간복잡도 $T(n)$: 입력크기 n 에 대한 단위연산 실행횟수
- 예제
 - 리스트(배열)의 원소 모두 더하기
 - 교환정렬
 - 행렬곱셈

최악 시간복잡도

- 최악 시간복잡도 $W(n)$: 입력크기 n 에 대한 단위연산의 최대 실행횟수
- 핵발전소 시스템의 경우처럼 나쁜 사례에 대한 최악의 반응시간이 중요한 경우 활용

평균 시간복잡도

위

- 평균 시간복잡도 $A(n)$: 입력크기 n 에 대한 단위연산의 실행횟수 기대치(평균)
- 평균 단위연산 실행횟수가 중요한 경우 활용
- 각 입력값에 대해 확률 할당 가능
- 최악의 경우 분석보다 ~~보통~~ 계산이 보다 복잡함

최선 시간복잡도

- 최선 시간복잡도 $B(n)$: 입력크기 n 에 대한 단위연산의 최소 실행횟수
- 잘 사용되지 않음.

시간복잡도 특성

- $T(n)$ 이 존재하는 경우:

$$T(n) = W(n) = A(n) = B(n)$$

- 일반적으로:

$$B(n) \leq A(n) \leq W(n)$$

일정 시간복잡도를 구할 수 없는 경우

- 최선의 경우 보다 최악 또는 평균의 경우 분석을 일반적으로 진행
- 평균 시간복잡도 분석
 - 다른 입력을 여러 번 사용할 때 평균적으로 걸리는 시간 알려줌.
 - 예를 들어, 속도가 느린 정렬 알고리즘이라도 평균적으로 시간이 좋게 나오는 경우 사용 가능.
- 최악 시간복잡도 분석
 - 핵발전소 감시시스템 경우처럼 단 한 번의 사고가 치명적인 경우 활용.

공간(메모리)복잡도

- 알고리즘이 메모리 사용량으로 얼마나 효율적인지 분석
- 책에서는 시간복잡도에 집중.
- 필요한 경우 공간복잡도 분석 활용.

예제: 일정 시간복잡도 분석

알고리즘: 리스트(배열) 항목더하기

- 문제: 크기가 n 인 리스트(배열) S 의 모든 항목을 더하라.
- 입력: 리스트(배열) S
- 출력: 리스트(배열) S 에 있는 항목의 합

In [7]: *# 리스트(배열)의 항목 모두 더하기*

```
def sum(S):  
    result = 0  
  
    for i in range(len(S)):  
        result = result + S[i]  
    return result
```

In [9]: seq = list(range(11))

```
sum(seq)
```

Out[9]: 55

리스트(배열) 항목더하기 알고리즘의 $T(n)$ 구하기: 덧셈 기준

- 단위연산: 덧셈
- 입력크기: 리스트(배열)의 크기 n

- 모든 경우 분석:
 - 리스트(배열) 내용에 상관없이 for-반복문 n 번 실행.
 - 반복마다 덧셈 1회 실행.
 - 따라서 $T(n) = n$.

알고리즘: 교환정렬

- 문제: 리스트(배열)의 항목을 비내림차순(오름차순)으로 정렬하기
- 입력: 리스트(배열) S
- 출력: 비내림차순으로 정렬된 리스트(배열)

In [10]: # 교환정렬

```
def exchangesort(S):  
    for i in range(len(S)):  
        for j in range(i+1, len(S)):  
            if (S[j] < S[i]):  
                S[i], S[j] = S[j], S[i]
```

In [11]: seq = [1, 4, 5, 2, 7]
exchangesort(seq)
print(seq)

[1, 2, 4, 5, 7]

교환정렬 알고리즘의 $T(n)$ 구하기: 조건문 기준

- 단위연산: 조건문 ($s[j]$ 와 $s[i]$ 의 비교)
- 입력크기: 리스트(배열)의 길이 n

- 모든 경우 분석:
 - j-반복문이 실행할 때마다 조건문 한 번씩 실행
 - 조건문의 총 실행횟수
 - $i = 1$ 일 때: $n - 1$ 번
 - $i = 2$ 일 때: $n - 2$ 번
 - $i = 3$ 일 때: $n - 3$ 번
 - ...
 - $i = n - 1$ 일 때: 1 번
 - 따라서 다음이 성립한다.

$$T(n) = (n - 1) + (n - 2) + \dots + 1 = \frac{(n - 1)n}{2}$$

예제: 최악 시간복잡도 분석

교환정렬 알고리즘의 $W(n)$ 구하기: 교환 기준

- 단위연산: 교환하는 연산 ($s[j]$ 와 $s[i]$ 의 교환)
- 입력크기: 정렬할 항목의 수 n

- 최악의 경우 분석:
 - 조건문의 결과에 따라서 교환 연산의 실행여부 결정
 - 최악의 경우
 - 조건문이 항상 참(true)이 되는 경우
 - 즉, 입력 배열이 ~~꺼~~꾸로 정렬되어 있는 경우
 - 이때, 조건문 실행 횟수와 동일하게 실행됨.

$$W(n) = \frac{(n-1)n}{2}$$

순차검색 알고리즘의 $W(n)$ 구하기: 항목 비교 연산 기준

- 단위연산: 리스트(배열) s 의 항목과 값 x 와의 비교연산
 - $S[\text{location}] \neq x$
- 입력크기: 리스트(배열) 크기 n

- 최악의 경우 분석:

- x 가 리스트(배열)의 마지막 항목이거나, 리스트(배열)에 포함되지 않은 경우, 단위연산이 n 번 수행된다. 즉,

$$W(n) = n$$

- **주의:** 입력(s 와 x)에 따라서 검색횟수가 달라지므로, 일정 시간복잡도 분석 불가능.

예제: 평균 시간복잡도 분석

순차검색 알고리즘의 $A(n)$ 구하기: 항목 비교 연산 기준

- 단위연산: 리스트(배열) s 의 항목과 값 x 와의 비교연산
 - $S[\text{location}] \neq x$
- 입력크기: 리스트(배열) 크기 n

- 평균의 경우 분석 (경우 1)

- 가정

- x 가 리스트(배열) s 안에 있음
 - 리스트(배열)의 항목이 모두 다름.
 - x 가 리스트(배열)의 특정 위치에 있을 확률 동일, 즉 $1/n$. 단, n 은 리스트(배열) s 의 길이.

- x 가 리스트(배열)의 k 번째 있다면, s 를 찾기 위해서 수행하는 단위연산의 횟수는 k .

- 따라서 다음이 성립

$$\begin{aligned}
 A(n) &= \sum_{k=1}^n \left(k \times \frac{1}{n} \right) \\
 &= \frac{1}{n} \times \sum_{k=1}^n k \\
 &= \frac{1}{n} \times \frac{n(n+1)}{2} \\
 &= \frac{n+1}{2}
 \end{aligned}$$

순차검색 알고리즘의 $A(n)$ 구하기 (경우 2)

- 가정
 - x 가 리스트(배열) s 안에 없을 수도 있음.
 - x 가 리스트(배열) s 안에 있을 확률: p
- x 가 배열에 없을 확률: $1 - p$
- x 가 리스트(배열)의 k 번째 항목일 확률: p/n
- 따라서 다음이 성립.

$$\begin{aligned} A(n) &= \sum_{k=1}^n \left(k \times \frac{p}{n} \right) + n(1 - p) \\ &= \frac{p}{n} \times \frac{n(n+1)}{2} + n(1 - p) \\ &= n \left(1 - \frac{p}{2} \right) + \frac{p}{2} \end{aligned}$$

- $p = 1$ 일 때:
= 1

$$A(n) = (n + 1)/2$$

예제: 최선 시간복잡도 분석

교환정렬 알고리즘의 $B(n)$ 구하기: 교환 기준

- 단위연산: 교환하는 연산 ($s[j]$ 와 $s[i]$ 의 교환)
- 입력크기: 정렬할 항목의 수 n

- 최선의 경우 분석:
 - 조건문의 결과에 따라서 교환 연산의 실행여부 결정
 - 최선의 경우
 - 조건문이 항상 거짓(false)이 되는 경우
 - 즉, 입력 배열이 이미 오름차순(비내림차순)으로 정렬되어 있는 경우
 - 이때, 교환이 전혀 발생하지 않음.
 - 따라서 $B(n) = 0$.

순차검색 알고리즘의 $B(n)$ 구하기: 항목 비교 연산 기준

- 단위연산: 리스트(배열) s 의 항목과 값 x 와의 비교연산
 - $S[\text{location}] \neq x$
- 입력크기: 리스트(배열) 크기 n

- 최선의 경우 분석:
 - x 가 $s[0]$ 일 때, 입력의 크기에 상관없이 단위연산이 한 번 수행
 - 따라서 $B(n) = 1$.

4절 차수

복잡도의 표기법

- O
 - Big O , asymptotic upper bound
- Ω
 - Omega, asymptotic lower bound
- Θ
 - Theta, order, asymptotic tight bound ($O \cap \Omega$)

대표적인 복잡도 카테고리

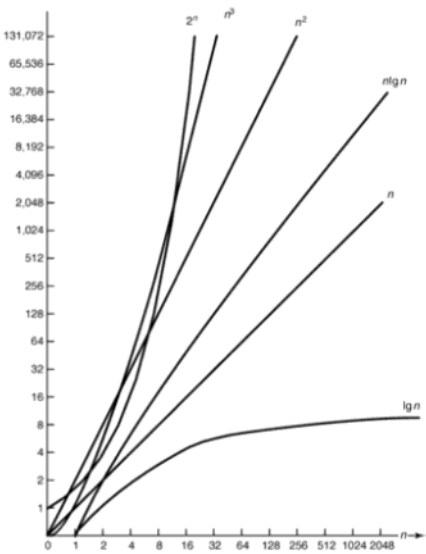
- $\Theta(\lg n)$
- $\Theta(n)$: 1차 (linear time algorithm)
- $\Theta(n \lg n)$
- $\Theta(n^2)$: 2차 (quadratic time)
- $\Theta(n^3)$: 3차 (cubic time)
- $\Theta(2^n)$: 지수 (exponential time)
- $\Theta(n!)$

최고차 항이 궁극적으로 지배한다

n	$0.1n^2$	$0.1n^2+n+100$
10	10	120
20	40	160
50	250	400
100	1,000	1,200
1,000	100,000	101,100

- $g(n)$ order of n^2
 $= 5n^2$
 $+ 100n$
 $+ 20$
 $\in \theta$
 $(n^2) \equiv$

복잡도 함수의 증가율



시간복잡도별 실행시간 비교

n	$f(n) = \lg n$	$f(n) = n$	$f(n) = n \lg n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0,003 μs^*	0,01 μs	0,033 μs	0,10 μs	1,0 μs	1 μs
20	0,004 μs	0,02 μs	0,086 μs	0,40 μs	8,0 μs	1 ms [†]
30	0,005 μs	0,03 μs	0,147 μs	0,90 μs	27,0 μs	1 s
40	0,005 μs	0,04 μs	0,213 μs	1,60 μs	64,0 μs	18,3 min
50	0,006 μs	0,05 μs	0,282 μs	2,50 μs	125,0 μs	13 days
10^2	0,007 μs	0,10 μs	0,664 μs	10,00 μs	1,0 ms	4×10^{13} years
10^3	0,010 μs	1,00 μs	9,966 μs	1,00 ms	1,0 s	
10^4	0,013 μs	10,00 μs	130,000 μs	100,00 ms	16,7 min	
10^5	0,017 μs	0,10 ms	1,670 ms	10,00 s	11,6 days	
10^6	0,020 μs	1,00 ms	19,930 ms	16,70 min	31,7 years	
10^7	0,023 μs	0,01 s	2,660 s	1,16 days	31,709 years	
10^8	0,027 μs	0,10 s	2,660 s	115,70 days	$3,17 \times 10^7$ years	
10^9	0,030 μs	1,00 s	29,900 s	31,70 years		

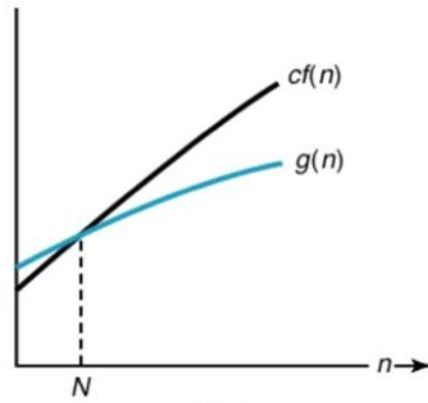
*1 $\mu s = 10^{-6}$ second,

†1 ms = 10^{-3} second,

Big O 표기법

- 정의 : 점근적 상한 (Asymptotic Upper Bound)
 - 분석된 복잡도함수 $g(n)$ 이 어떤 함수 $f(n)$ 에 대해서 $g(n) \in O(f(n))$.
 - $n \geq N$ 인 모든 정수 n 에 대해서 $g(n) \leq c \times f(n)$ 이 성립하는 실수 $c > 0$ 와 음이 아닌 정수 N 이 존재한다.
- $g(n) \in O(f(n))$ 읽는 방법:
 - $g(n)$ 의 점근적 상한은 $f(n)$ 이다.
 - Asymptotic upper bound of $g(n)$ is $f(n)$.
- 의미:
 - 입력 크기 n 에 대해서 이 알고리즘의 수행시간은 궁극적으로 $f(n)$ 보다 나쁘지는 않다.

Big O 표기법 (Cont)



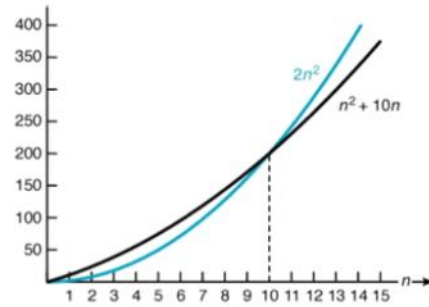
(a) $g(n) \in O(f(n))$

Big O 표기법 (예)

- 어떤 함수 $g(n)$ 이 $O(n^2)$ 에 속한다는 말은
 - 함수 $g(n)$ 은 궁극에 가서는 (즉, 어떤 N 값 이후부터는) 어떤 2차 함수 cn^2 보다는 작은 값을 가지게 된다는 것을 뜻한다. (그래프 상에서는 아래에 위치)
- $n^2 + 10n \in O(n^2)$?
 - $n \geq 10$ 인 모든 정수 n 에 대해서 $n^2 + 10n \leq 2n^2$ 이 성립한다. 그러므로 $c = 2$ 와 $N = 10$ 을 선택하면, 'Big O'의 정의에 의해서 $n^2 + 10n \in O(n^2)$ 이라고 결론지을 수 있다.
 - $n \geq 1$ 인 모든 정수 n 에 대해서 $n^2 + 10n \leq n^2 + 10n^2 = 11n^2$ 이 성립한다. 그러므로 $c = 11$ 와 $N = 1$ 을 선택하면, '큰 O'의 정의에 의해서 $n^2 + 10n \in O(n^2)$ 이라고 결론지을 수 있다.

Big O 표기법 (예) (Cont)

- $2n^2$ 과 $n^2 + 10n$ 의 비교



Big O 표기법 (예) (Cont)

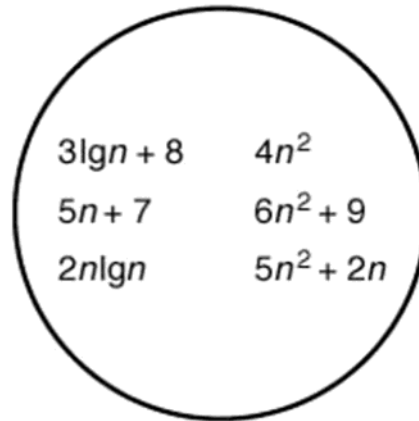
- $5n^2 \in O(n^2)$?
 - $c = 5$ 와 $N = 0$ 을 선택하면, $n \geq 0$ 인 모든 정수 n 에 대해서 $5n^2 \leq 5n^2$ 이 성립한다.
- $T(n) = n(n - 1)/2$?
 - $n \geq 0$ 인 모든 정수 n 에 대해서 $n(n - 1)/2 \leq n^2/2$ 이 성립한다. 그러므로 $c = 1/2$ 과 $N = 0$ 을 선택하면, $T(n) \in O(n^2)$ 이라고 결론지을 수 있다.
- $n^2 \in O(n^2 + 10n)$?
 - $n \geq 0$ 인 모든 정수 n 에 대해서, $n^2 \leq 1 \times (n^2 + 10n)$ 이 성립한다. 그러므로, $c = 1$ 와 $N = 0$ 을 선택하면, $n^2 \in O(n^2 + 10n)$ 이라고 결론지을 수 있다.

Big O 표기법 (예) (Cont)

- $n \in O(n^2)$?
 - $n \geq 1$ 인 모든 정수 n 에 대해서, $n \leq 1 \times n^2$ 이 성립한다. 그러므로, $c = 1$ 와 $N = 1$ 을 선택하면, $n \in O(n^2)$ 이라고 결론지을 수 있다.
- $n^3 \in O(n^2)$?
 - $n \geq N$ 인 모든 n 에 대해서 $n^3 \leq c \cdot n^2$ 이 성립하는 c 와 N 값은 존재하지 않는다. 즉, 양변을 n^2 으로 나누면, $n \leq c$ 가 되는데, c 를 아무리 크게 잡더라도 그 보다 더 큰 n 이 존재한다. (성립하지 않음)

Big O 표기법 (예) (Cont)

- $O(n^2)$: cn^2 보다 작은 값을 가지는 모든 함수.

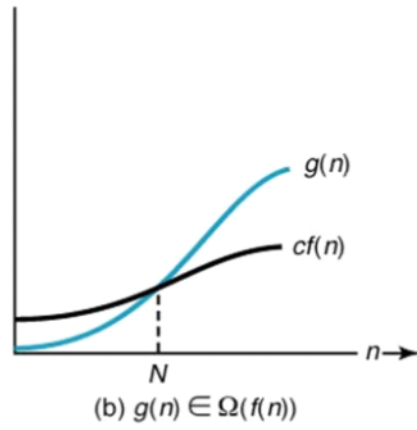


(a) $O(n^2)$

Ω 표기법

- 정의: 점근적 하한 (Asymptotic Lower Bound)
 - 분석된 복잡도함수 $g(n)$ 이 어떤 함수 $f(n)$ 에 대해서 $g(n) \in \Omega(f(n))$
 - $n \geq N$ 인 모든 정수 n 에 대해서 $g(n) \geq c \cdot f(n)$ 이 성립하는 실수 $c > 0$ 와 음이 아닌 정수 N 이 존재한다.
- $g(n) \in \Omega(f(n))$ 읽는 방법:
 - $g(n)$ 의 점근적 하한은 $f(n)$ 이다.
 - Asymptotic lower bound of $g(n)$ is $f(n)$.
- 의미:
 - 입력 크기 n 에 대해서 이 알고리즘의 수행시간은 궁극적으로 $f(n)$ 보다 효율적이지는 못하다.

Ω 표기법



Ω 표기법 : 예

- 어떤 함수 $g(n)$ 이 $\Omega(n^2)$ 에 속한다는 말은
 - 그 함수는 궁극에 가서는 (즉 어떤 N 값 이후부터는) 어떤 2차 함수 $c \cdot n^2$ 의 값보다는 큰 값을 가지게 된다는 것을 뜻한다(그래프 상에서는 위에 위치).
- $n^2 + 10n \in \Omega(n^2)$?
 - $n \geq 0$ 인 모든 정수 n 에 대해서 $n^2 + 10n \geq n^2$ 이 성립한다. 그러므로 $c = 1$ 와 $N = 0$ 을 선택하면, $n^2 + 10n \in \Omega(n^2)$ 이라고 결론지을 수 있다.
- $5n^2 \in \Omega(n^2)$?
 - $n \geq 0$ 인 모든 정수 n 에 대해서, $5n^2 \geq 1 \cdot n^2$ 이 성립한다. 그러므로, $c = 1$ 와 $N = 0$ 을 선택하면, $5n^2 \in \Omega(n^2)$ 이라고 결론지을 수 있다.

Ω 표기법 : 예 (계속)

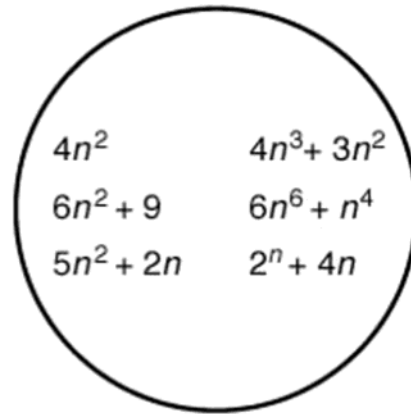
- $T(n) = n(n - 1)/2$?
 - $n \geq 2$ 인 모든 n 에 대해서 $n - 1 \geq n/2$ 이 성립한다. 그러므로, $n \geq 2$ 인 모든 n 에 대해서 $n(n - 1)/2 \geq n/2 \cdot n/2 = 1/4n^2$ 이 성립한다. 따라서 $c = 1/4$ 과 $N = 2$ 를 선택하면, $T(n) \in \Omega(n^2)$ 이라고 결론지을 수 있다.
- $n^3 \in \Omega(n^2)$?
 - $n \geq 1$ 인 모든 정수 n 에 대해서, $n^3 \geq 1 \cdot n^2$ 이 성립한다. 그러므로, $c = 1$ 과 $N = 1$ 을 선택하면, $n^3 \in \Omega(n^2)$ 이라고 결론지을 수 있다.

Ω 표기법 : 예 (계속)

- $n \in \Omega(n^2)$?
 - 모순유도에 의한 증명(Proof by contradiction)
 - $n \in \Omega(n^2)$ 이라고 가정. 그러면 $n \geq N$ 인 모든 정수 n 에 대해서, $n \geq c \cdot n^2$ 이 성립하는 실수 $c > 0$, 그리고 음이 아닌 정수 N 이 존재한다. 위의 부등식의 양변을 $c \cdot n$ 으로 나누면 $1/c \geq n$ 이 된다. 그러나 이 부등식은 절대로 성립할 수 없다. 따라서 위의 가정은 모순이다.

Ω 표기법: 예 (Cont)

- $\Omega(n^2)$

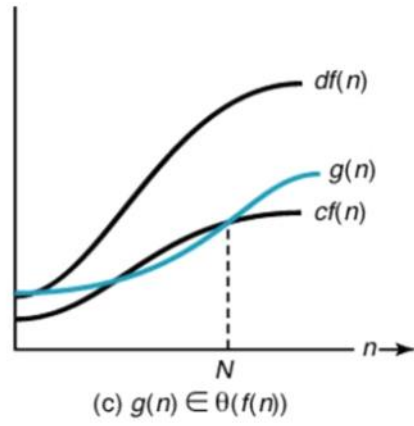


(b) $\Omega(n^2)$

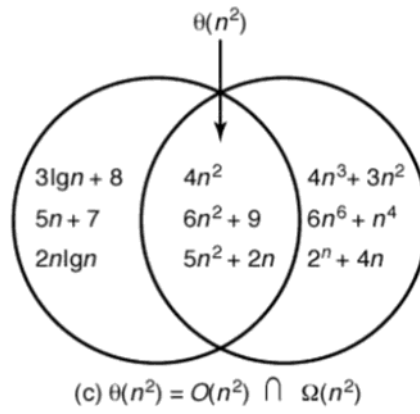
Θ 표기법

- 정의 : Asymptotic Tight Bound
 - 분석된 복잡도함수 $g(n)$ 이 어떤 함수 $f(n)$ 에 대해서 $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$.
 - $n \geq N$ 인 모든 정수 n 에 대해서 $c \cdot f(n) \geq g(n) \leq d \cdot f(n)$ 이 성립하는 실수 $c > 0$ 와 $d > 0$, 그리고 음이 아닌 정수 N 이 존재한다.
- $g(n) \in \Theta(f(n))$ 읽는 방법:
 - $g(n)$ 의 차수(order=asymptotic tight bound)는 $f(n)$ 이다.
 - Asymptotic tight bound of $g(n)$ is $f(n)$.
- 예 : $T(n) = n(n - 1)/2$ 은 $O(n^2)$ 이면서 $\Omega(n^2)$ 이다. 따라서 $T(n) = \Theta(n^2)$.

⊕ 표기법



$\Theta(n^2)$



작은(Small) o 표기법

- 정의: 작은 o
 - 분석된 복잡도 함수 $g(n)$ 이 어떤 함수 $f(n)$ 에 대해서 $g(n) \in o(f(n))$
 - 어떤 N 값 이후부터는 모든 실수 $c > 0$ 에 대해서 $g(n) \leq c \cdot f(n)$
- 참고: $g(n) \in o(f(n))$ 은 " $g(n)$ 은 $f(n)$ 의 작은 오(o)"라고 한다.

큰 O vs 작은 o

- 큰 O 와의 차이점
 - 큰 O : 실수 $c > 0$ 중에서 하나만 성립하여도 됨
 - 작은 o : 모든 실수 $c > 0$ 에 대해서 성립하여야 함
- $g(n) \in o(f(n))$ 은 쉽게 설명하자면
 - $g(n)$ 이 궁극적으로 $f(n)$ 보다 '훨씬' 낮다(좋다)는 의미이다.

작은 o 표기법 : 예

- $n \in o(n^2)$?

- 증명:

$c > 0$ 이라고 하자. $n \geq N$ 인 모든 n 에 대해서 $n \leq c \cdot n^2$ 이 성립하는 N 을 찾아야 한다. 이 부등식의 양변을 cn 으로 나누면 $1/c \leq n$ 을 얻는다. 따라서 $N \geq 1/c$ 가 되는 어떤 N 을 찾으면 된다. 여기서 N 의 값은 c 에 의해 좌우된다.

예를 들어 만약 $c = 0.0001$ 이라고 하면, N 의 값은 최소한 10,000이 되어야 한다. 즉, $n \geq 10,000$ 인 모든 n 에 대해서 $n \leq 0.0001 \cdot n^2$ 이 성립한다.

작은 o 표기법 : 예 (계속)

- n 이 $o(5n)$?
- 모순 유도에 의한 증명: $c = 1/6$ 이라고 하자.

$n \in o(5n)$ 이라고 가정하면, $n \geq N$ 인 모든 정수 n 에 대해서, $n \leq 1/6 \cdot 5 \cdot n \leq 5/6 \cdot n$ 이 성립하는 음이 아닌 정수 N 이 존재해야 한다.

그러나 그런 N 은 절대로 있을 수 없다. 따라서 위의 가정은 모순이다.

극한(limit)을 이용하여 차수를 구하는 방법

- 정의:

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \begin{cases} c & \text{for some } c > 0 \text{ if } g(n) \in \Theta(f(n)), \\ 0 & \text{if } g(n) \in o(f(n)) = O(f(n)) \setminus \Theta(f(n)), \\ \infty & \text{if } g(n) \in \Omega(f(n)) \setminus \Theta(f(n)). \end{cases}$$

- 예: 다음이 성립함을 보이시오.

- $\frac{n^2}{2} \in o(n^3)$
 - 이유: $\lim_{n \rightarrow \infty} \frac{n^2/2}{n^3} = \lim_{n \rightarrow \infty} \frac{1}{2n} = 0$
- $b > a > 0$ 일 때, $a^n \in o(b^n)$
 - 이유: $0 < \frac{a}{b} < 1$. 따라서 $\lim_{n \rightarrow \infty} \frac{a^n}{b^n} = \lim_{n \rightarrow \infty} \left(\frac{a}{b}\right)^n = 0$.

로피탈(L'Hopital)의 법칙

- 정리: 로피탈(L'Hopital)의 법칙:

$\lim_{n \rightarrow \infty} f(n) = \lim_{n \rightarrow \infty} g(n) = \infty$ 이면 $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \lim_{n \rightarrow \infty} \frac{g'(n)}{f'(n)}$ 이다.

- 예: 다음이 성립함을 보이시오.

- $\lg n \in o(n)$, 이유는

$$\lim_{n \rightarrow \infty} \frac{\lg n}{n} = \lim_{n \rightarrow \infty} \left(\frac{\frac{1}{n \ln 2}}{1} \right) = 0$$

- $\log_a n \in \Theta(\log_b n)$, 이유는

$$\lim_{n \rightarrow \infty} \frac{\log_a n}{\log_b n} = \lim_{n \rightarrow \infty} \left(\frac{\frac{1}{n \ln a}}{\frac{1}{n \ln b}} \right) = \frac{\log b}{\log a} > 0$$

차수의 주요 성질 I

$$1. \begin{aligned} g(n) &\in O(f(n)) \text{ iff } f(n) \in \Omega(g(n)). \\ &\Leftrightarrow \begin{aligned} &\exists c > 0, n_0 \in \mathbb{N} \text{ such that } \\ &0 \leq f(n) \leq c g(n) \text{ for all } n \geq n_0. \end{aligned} \end{aligned}$$

- $g(n) \in \Theta(f(n))$ iff $f(n) \in \Theta(g(n))$.
- $b > 1$ 이고 $a > 1$ 이면, $\log_a n \in \Theta(\log_b n)$ 은 항상 성립. 다시 말하면 로그(logarithm) 복잡도 함수는 모두 같은 카테고리에 속한다. 따라서 통상 $\Theta(\lg n)$ 으로 표시한다.
- $b > a > 0$ 이면, $a^n \in o(b^n)$. 다시 말하면, 지수(exponential) 복잡도 함수가 모두 같은 카테고리 안에 있는 것은 아니다.

차수의 주요 성질 II

1. $a > 0$ 인 모든 a 에 대해서, $a^n \in o(n!)$. 다시 말하면, $n!$ 은 어떤 지수 복잡도 함수보다도 나쁘다.
2. 복잡도 함수를 다음 순으로 나열해 보자.

$$\Theta(\lg n), \Theta(n), \Theta(n \lg n), \Theta(n^2), \Theta(n^j), \Theta(n^k), \Theta(a^n), \Theta(b^n), \Theta(n!)$$

여기서 $k > j > 2$ 이고 $b > a > 1$ 이다.

복잡도 함수 $g(n)$ 이 $f(n)$ 을 포함한 카테고리의 왼쪽에 위치하면, $g(n) \in o(f(n))$.

3. $c \geq 0, d > 0, g(n) \in O(f(n))$, 그리고 $h(n) \in \Theta(f(n))$ 이면,

$$c \cdot g(n) + d \cdot h(n) \in \Theta(f(n))$$

- ex) $5n + 3 \lg n + 10n \lg n + 7n^2 \in \Theta(n^2)$.