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

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

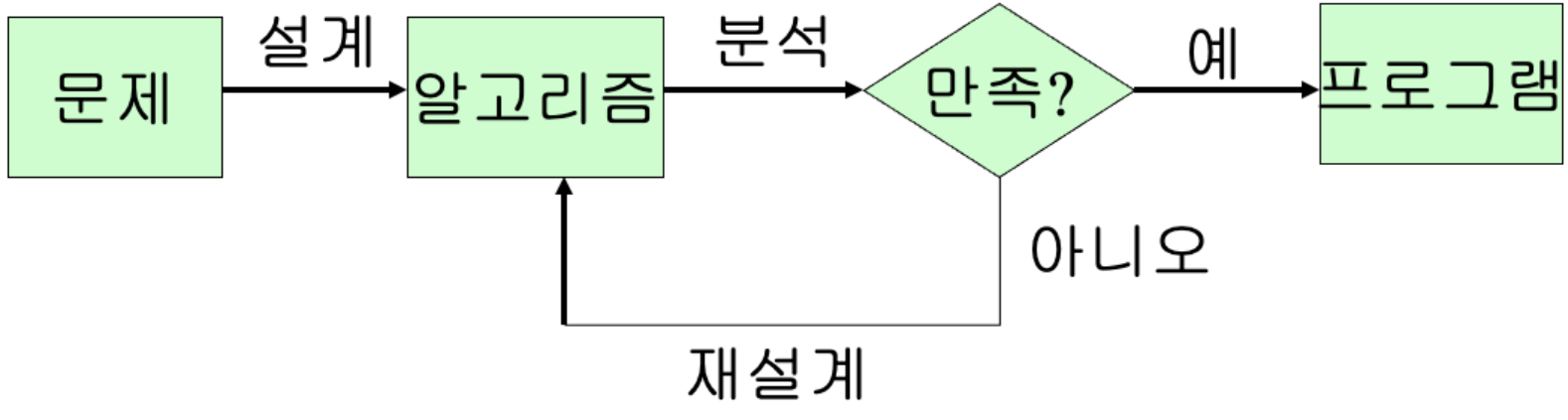
- 7장 계산복잡도 소개: 정렬문제
- 8장 계산복잡도: 검색문제
- 9장 계산복잡도와 문제 난이도: NP 이론 소개

1장 주요 내용

- 1절 알고리즘
- 2절 효율적인 알고리즘 개발 중요성
- 3절 알고리즘 분석
- 4절 차수

1절 알고리즘

프로그램 설계 과정



알고리즘이란?

- 컴퓨터를 이용하여 주어진 문제를 해결하는 기법
- 컴퓨터 프로그램은 여러 방법 중에서 한 가지 방법을 선택하여 구현
- 프로그래밍 언어, 프로그래밍 스타일과 무관

알고리즘과 절차

- 절차: 문제해결 알고리즘 적용 순서

알고리즘 효율성 분석

- 효율성: 문제해결을 위한 필수 요소
    - 컴퓨터 속도가 아무리 빨라져도, 메모리 가격이 아무리 저렴해져도 효율성 문제는 언제나 중요!
    - 수천년, 수만년 동안 실행되어야 끝나는 비효율적 알고리즘이 일반적으로 존재.

- 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 계산복잡도를 기준으로 다양한 카테고리로 분류

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 S에 x가 항목으로 포함되어 있는가?
    - 입력 파라미터: 리스트 S와 값 x
    - 리턴값:
        - x가 S의 항목일 경우: x의 위치 인덱스
        - 항목이 아닐 경우 -1.

- 알고리즘 (자연어):
    - x와 같은 항목을 찾을 때까지 S에 있는 모든 항목을 차례로 검사
    - 만일 x와 같은 항목을 찾으면 항목의 인덱스 내주기
    - S를 모두 검사하고도 찾지 못하면 -1 내주기

In [1]:

```python
# 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 [2]:
```
seq = list(range(30))
val = 5

print(seqsearch(seq, val))
```

```
(5, 5)
```

In [3]:
```
seq = list(range(30))
val = 10

print(seqsearch(seq, val))
```

```
(10, 10)
```

In [4]:
```
seq = list(range(30))
val = 20

print(seqsearch(seq, val))
```

```
(20, 20)
```

In [5]:
```
seq = list(range(30))
val = 29

print(seqsearch(seq, val))
```

```
(29, 29)
```

In [6]:
```
seq = list(range(30))
val = 30

print(seqsearch(seq, val))
```

```
(-1, 30)
```

In [7]:
```
seq = list(range(30))
val = 100

print(seqsearch(seq, val))
```

```
(-1, 30)
```

- 입력값의 위치에 따라 `while` 반복문의 실행횟수가 선형적으로 달라짐.

파이썬튜터 활용: 순차검색

- 위 순차검색 코드를 PythonTutor: 순차검색 (http://pythontutor.com/visualize.html#code=%23%20%EC%88%9C%EC%B0%A8%E
1,%20loop_count%29%0A%0Aseq%20%3D%20list%28range%2830%29%29%0Aval

순차검색 분석

- 특정 값의 위치를 확인하기 위해서 S의 항목 몇 개를 검색해야 하는가?
    - 특정 값과 동일한 항목의 위치에 따라 다름
    - 최악의 경우: S의 길이, 즉, 항목의 개수

- 좀 더 빨리 찾을 수는 없는가?
    - S에 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** S에 x가 항목으로 포함되어 있는가?
    - 입력 파라미터: 리스트 S와 값 x
    - 리턴값:
        - x가 S의 항목일 경우: x의 위치 인덱스
        - 항목이 아닐 경우 -1.

- 알고리즘 (자연어):
    - S의 중간에 위치한 항목과 x를 비교
        - 만일 x와 같으면 해당 항목의 인덱스 내주기
        - 만일 x가 중간에 위치한 값보다 작으면 중간 왼편에 위치한 구간에서 새롭게 검색
        - 만일 x가 중간에 위치한 값보다 크면 중간 오른편에 위치한 구간에서 새롭게 검색
    - x와 같은 항목을 찾거나 검색 구간의 크기가 0이 될 때가지 위 절차 반복

In [8]:

```
# 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 [9]:
```
seq = list(range(30))
val = 5

print(binsearch(seq, val))
```

```
(5, 5)
```

In [10]:
```
seq = list(range(30))
val = 10

print(binsearch(seq, val))
```

```
(10, 3)
```

In [11]:
```
seq = list(range(30))
val = 20

print(binsearch(seq, val))
```

```
(20, 4)
```

In [12]:
```
seq = list(range(30))
val = 29

print(binsearch(seq, val))
```

```
(29, 5)
```

In [13]:
```
seq = list(range(30))
val = 30

print(binsearch(seq, val))
```

```
(-1, 5)
```

In [14]:
```
seq = list(range(30))
val = 100

print(binsearch(seq, val))
```

```
(-1, 5)
```

- 입력값이 달라져도 `while` 반복문의 실행횟수가 거의 변하지 않음.

파이썬튜터 활용: 이분검색

- 위 이분검색 코드를 PythonTutor: 이분검색 (http://pythontutor.com/visualize.html#code=1%0A%20%20%20%20%0A%20%20%20%20%23%20while%20%EB%B0%98%EB%1%3A%0A%20%20%20%20%20%20%20%20loop_count%20%2B%3D%201%0A%2%201%0A%20%20%20%20%20%20%20%20else%3A%0A%20%20%20%20%20%2frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false) 에서 실행하면서

이분검색 분석

- 이분검색으로 특정 값의 위치를 확인하기 위해서 S의 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

- 최악의 경우 확인 항목수

배열 크기	순차 검색	이분 검색
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, 트위터 등등 수백에서 수천만의 회원을 대상으로 검색을 진행하고자 한다면 어떤 알고리즘 선택?

당연히 이분검색!

- 이분 검색은 검색 속도가 사실상 최고로 빠름

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

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, \ldots$$

피보나찌 수 구하기 알고리즘(재귀)

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

In [15]:
```
# 피보나찌 수 구하기 알고리즘(재귀)

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

In [16]:
```
fib(3)
```

Out[16]: 2

In [17]:
```
fib(6)
```

Out[17]: 8

In [18]:
```
fib(10)
```

Out[18]: 55

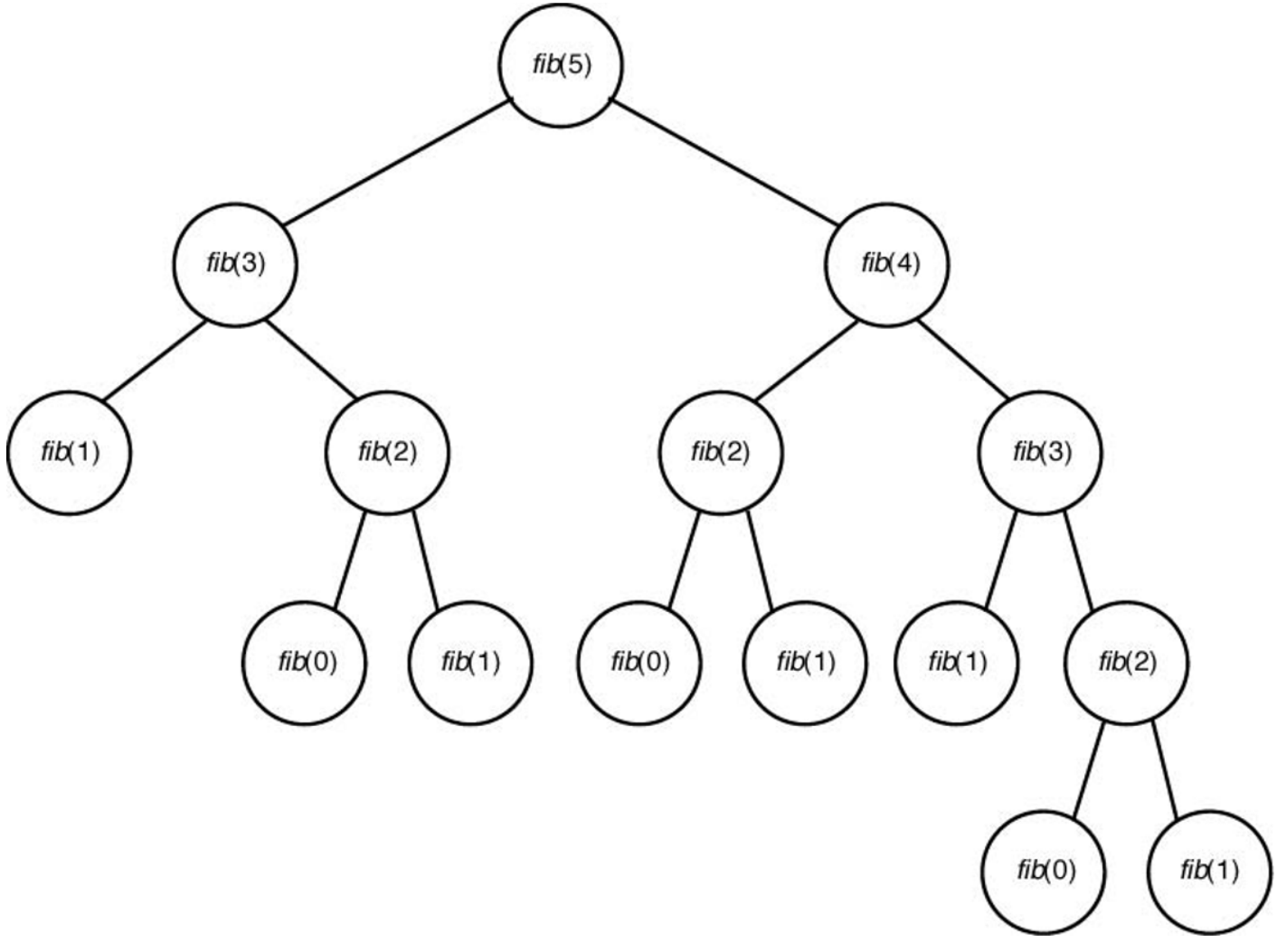
In [19]:
```
fib(13)
```

Out[19]: 233

`fib` 함수 분석

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

- 예를들어, `fib(5)`를 계산하기 위해 `fib(2)`가 세 번 호출됨. 아래 나무구조 그림 참조.



`fib` 함수 호출 횟수

- $T(n)$ = `fib(n)`을 계산하기 위해 `fib` 함수를 호출한 횟수.
    - 즉, `fib(n)`을 위한 재귀 나무구조에 포함된 마디(node)의 개수

- 아래 부등식 성립.

$$
\begin{aligned}
T(0) &= 1 \\
T(1) &= 1 \\
T(n) &= T(n-1) + T(n-2) + 1 \quad (n \geq 2) \\
&> 2 \times T(n-2) \quad (T(n-1) > T(n-2)) \\
&> 2^2 \times T(n-4) \\
&> 2^3 \times T(n-6) \\
&\cdots \\
&> 2^{n/2} \times T(0) = 2^{n/2}
\end{aligned}
$$

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

정리 1.1

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

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

- 증명: (n에 대한 수학적 귀납법으로 증명)

- 귀납시작:
  - $T(2) = T(1) + T(0) + 1 = 3 > 2 = 2^{2/2}$
  - $T(3) = T(2) + T(1) + 1 = 5 > 2.83 \approx 2^{3/2}$

- 귀납가정(IH): $2 \leq m < n$인 모든 m에 대해서 $T(m) > 2^{m/2}$ 이라고 가정.

- 귀납절차: $T(n) > 2^{n/2}$ 임을 보이면 됨.

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + 1 \\
&> 2^{(n-1)/2} + 2^{(n-2)/2} + 1 \quad \text{by (IH)} \\
&> 2^{(n-2)/2} + 2^{(n-2)/2} \\
&= 2 \times 2^{(n/2)-1} \\
&= 2^{n/2}
\end{aligned}
$$

피보나찌 수 구하기 알고리즘 (반복)

- 한 번 계산한 값을 리스트에 저장.
- 중복 계산 없음: 필요할 때 저장된 값 활용

In [20]:
```
# 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 [21]:
```
fib2(3)
```

Out[21]: 2

In [22]:
```
fib2(6)
```

Out[22]: 8

In [23]:
```
fib2(10)
```

Out[23]: 55

In [24]:
```
fib2(13)
```

Out[24]: 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

- 가정: 피보나찌 수 하나를 계산하는 데 걸리는 시간 = 1 ns.
  - 1 ns = 10^{-9} 초
  - 1 μs = 10^{-6} 초

n	$n+1$	$2^{n/2}$	반복	재귀
40	41	$1,048,576$	41 ns	1048 μs
60	61	1.1×10^{9}	61 ns	1 초
80	81	1.1×10^{12}	81 ns	18 분
100	101	1.1×10^{15}	101 ns	13 일
120	121	1.2×10^{18}	121 ns	36 년
160	161	1.2×10^{24}	161 ns	3.8×10^{7} 년
200	201	1.3×10^{30}	201 ns	4×10^{13} 년

3절 알고리즘 분석

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

시간복잡도 분석

- 알고리즘 효율성 분석 기법
- 기준: 입력크기에 대해 **특정 단위연산이 수행되는 횟수**

입력크기 : 입력값의 크기

- 예제
    - 리스트의 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 n의 크기는 n을 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)

- 단위연산의 실행 횟수가 알고리즘의 실행 시간 결정

- 예제
    - 비교문(comparison)
    - 지정문(assignment)
    - 반복문
    - 모든 기계적 명령문 각각의 실행
        - 예제: PythonTutor의 Step 계산

- 순차검색과 이분검색 알고리즘의 단위 연산: while 반복문 전체

- 피보나찌 함수의 단위 연산: 함수 본체 전체

주의사항

- 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 발생하기에 서로 독립적인 단위연산으로 다룸
- 단위연산의 실행횟수가 입력크기뿐만 아니라 입력값에도 의존할 수 있음.

알고리즘의 시간복잡도

- 입력값의 입력크기 n에 대해 지정된 단위연산이 수행되는 횟수 $f(n)$을 계산하는 함수 f로 표현
- 시간복잡도 함수: 시간복잡도를 표현하는 함수

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 $T(n)$
    - 입력값에 상관없이 입력크기 n에만 의존하는 단위연산 실행횟수

- 예제
    - 리스트의 원소 모두 더하기
    - 교환정렬
    - 행렬곱셈(2장)

최악 시간복잡도

- 최악 시간복잡도 $W(n)$: 입력크기 n에 대한 단위연산의 최대 실행횟수

- 예제
    - 핵발전소 시스템의 경우처럼 나쁜 사례에 대한 최악의 반응시간이 중요한 경우 활용

평균 시간복잡도

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

최선 시간복잡도

- 최선 시간복잡도 $B(n)$: 입력크기 n에 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

- 알고리즘의 메모리 사용 효율성 분석
- 책에서는 시간복잡도에 집중.
- 필요한 경우 공간복잡도 분석 활용.

예제: 일정 시간복잡도 분석

알고리즘: 리스트 항목더하기

- 문제: 크기가 n인 리스트 S의 모든 항목을 더하라.
- 입력: 리스트 S
- 출력: 리스트 S에 있는 항목의 합

In [25]:
```
# 리스트의 항목 모두 더하기

def sum(S):
    result = 0

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

In [26]:
```
seq = list(range(11))

sum(seq)
```

Out[26]: 55

리스트 항목더하기 알고리즘의 $T(n)$ 구하기: 덧셈 기준

- 단위연산: 덧셈
- 입력크기: 리스트의 크기 n

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

알고리즘: 교환정렬

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

In [27]:
```
# 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 [28]:
```
seq = [1, 4, 5, 2, 7, 4]
exchangesort(seq)
print(seq)
```

```
[1, 2, 4, 4, 5, 7]
```

교환정렬 알고리즘의 $T(n)$ 구하기 : 조건문 기준

- 단위연산: 조건문 (`S[j]`와 `S[i]`의 비교)
- 입력크기: 리스트의 길이 n

교환정렬 알고리즘 일정 시간복잡도 분석

- `j`-반복문이 실행할 때마다 조건문 한 번씩 실행

- 조건문의 총 실행횟수
  - $i = 1$인 경우: $(n-1)$ 번
  - $i = 2$인 경우: $(n-2)$ 번
  - $i = 3$인 경우: $(n-3)$ 번
  - ...
  - $i = (n$인 경우: 1 번
    $-1)$

- 그러므로 다음 성립:

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

- 확인

In [29]:
```
# 교환정렬

def exchangesort_1(S):
    count = 0
    for i in range(len(S)):
        for j in range(i+1, len(S)):
            count += 1
            if (S[j] < S[i]):
                S[i], S[j] = S[j], S[i]
    return count
```

In [30]:
```
seq = [1, 4, 5, 2, 7, 4]
print(exchangesort_1(seq))
```

```
15
```

- 실제로

$$15 = \frac{6 \cdot 5}{2}$$

예제: 최악 시간복잡도 분석

교환정렬 알고리즘의 $W(n)$ 구하기 : 교환 기준

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

- 최악의 경우 분석:
    - 조건문의 결과에 따라서 교환 연산의 실행여부 결정
    - 최악의 경우
        - 조건문이 항상 참(true)인 경우
        - 즉, 입력 배열이 거꾸로 정렬되어 있는 경우
        - 이때, 조건문 실행 횟수와 동일하게 실행됨. 즉, 일정 시간복잡도와 동일.

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

순차검색 알고리즘의 $W(n)$ 구하기: 항목 비교 연산 기준

- 단위연산: 리스트 S의 항목과 값 x와의 비교연산
    - `S[location] != x`
- 입력크기: 리스트 크기 n

- 최악의 경우 분석:
    - `x` 가 리스트의 마지막 항목이거나, 리스트에 포함되지 않은 경우, 단위연산이 n번 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 `S`의 항목과 값 `x`와의 비교연산
    - `S[location] != x`
- 입력크기: 리스트 크기 n

- 경우 1
    - 가정
        - x가 리스트 S안에 있음
        - 리스트의 항목이 모두 다름.
        - x가 리스트의 특정 위치에 있을 확률 동일, 즉 $1/n$. 단, n은 리스트 S의 길이.
    - x 가 리스트의 k 번째 있다면, S 를 찾기 위해서 수행하는 단위연산의 횟수는 k.

- 경우 2
    - 가정
        - x가 리스트 s 안에 없을 수도 있음.
        - x가 리스트 s 안에 있을 확률: p
    - x가 배열에 없을 확률: $1 - p$
    - x가 리스트의 k 번째 항목일 확률: p/n

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

- p 일 때: $A(n$
  $= 1$)
  $= \frac{n+1}{2}$

예제: 최선 시간복잡도 분석

교환정렬 알고리즘의 $B(n)$ 구하기 : 교환 기준

- 단위연산: 교환하는 연산 (`S[i]`와 `S[j]`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 $B(n) = 0$.

- 확인

In [31]:
```
# 교환정렬

def exchangesort_2(S):
    count = 0
    for i in range(len(S)):
        for j in range(i+1, len(S)):
            if (S[j] < S[i]):
                count += 1
                S[i], S[j] = S[j], S[i]
    return count
```

In [32]:
```
seq = [1, 2, 4, 4, 5, 7]
print(exchangesort_2(seq))
```

```
0
```

순차검색 알고리즘의 $B(n)$ 구하기: 항목 비교 연산 기준

- 단위연산: 리스트 S의 항목과 값 x와의 비교연산
    - `S[location] != x`
- 입력크기: 리스트 크기 n

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

복잡도 함수 예제

$$f(n) = 1$$
$$f(n) = \lg n$$
$$f(n) = n$$
$$f(n) = 1000n$$
$$f(n) = n^2$$
$$f(n) = \frac{n(n-1)}{2}$$
$$f(n) = 3n^2 + 4n^2$$

복잡도 함수와 실행시간

예제

- 아래 두 알고리즘 중에서 어떤 알고리즘 선택?
  - 알고리즘 A의 시간복잡도: $1000n$
  - 알고리즘 B의 시간복잡도: n^2

- n^2이 $1000n$ 보다 복잡도가 커보임. 하지만...

- 정답: n의 크기에 따라 달라짐.
  - $n \leq 1,000$: 알고리즘 B 선택
  - $n > 1,000$: 알고리즘 A 선택

- 이유:

$$n^2 > 1000n \quad \Longleftrightarrow \quad n > 1000$$

4절 차수

- 아래 두 알고리즘 중에서 어떤 알고리즘 선택?
  - 알고리즘 A의 시간 복잡도: $100n$
  - 알고리즘 B의 시간 복잡도: $0.01n^2$

- $0.01n^2$ 과 $100n$ 중에 누구의 복잡도가 더 커보임?

- 정답: n의 크기에 따라 달라짐.
  - $n \leq 10,000$: 알고리즘 B 선택
  - $n > 10,000$: 알고리즘 A 선택

- 이유:

$$0.01n^2 > 100n \iff n^2 > 10000n$$
$$\iff n > 10000$$

"궁극적으로 더 빠름"

- '$n > 10,000$인 임의의 양의 정수 n에 대해 $0.01n^2$ 이 $100n$ 보다 크다'를 다르게 표현하면 다음과 같음.

  <div style="text-align:center";>$0.01n^2$ 이 $100n$ 보다 궁극적으로 크다

- 다음 성질을 갖는 정수 $N \geq 0$이 존재할 때 $f(n)$이 $g(n)$ 보다 궁극적으로 크다고 말함:

  <div style="text-align:center";>$n > N$인 임의의 양의 정수 n에 대해 $f(n) > g(n)$

- 시간 복잡도의 기준으로 볼 경우:

<div style="text-align:center";>$g(n)$이 $f(n)$ 보다 궁극적으로 빠르다 $\iff$ $f(n)$이 $g(n)$ 보다 궁극적으로 크다

차수(Θ, 쎄타)의 직관적 이해

- $\Theta(n)$: 1차 시간 복잡도
  - $100n$
  - $0.001n + 10000$
  - 등등

- $\Theta(n^2)$: 2차 시간 복잡도
  - $5n^2$
  - $0.1n^2 + n + 100$
  - 등등

고차항의 지배력

- 예제: $0.1n^2 + n + 100$에서 2차 항 $0.1n^2$이 함수 전체를 지배함

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

복잡도 카테고리의 직관적 이해

- 1차, 2차, 3차 등의 시간복잡도를 갖는 함수들의 집합을 복잡도 카테고리라고 함.

매우 효율적인 알고리즘의 복잡도 예제

- $\Theta(1)$: 상수 복잡도
    - 0
    - 1
    - 1000
    - 1억 등등 모든 상수

- $\Theta(\lg n)$: 로그 복잡도
    - $\lg n$
    - $2 \lg n$
    - $1/2 \cdot \lg n + 3$
    - ...

- $\Theta(n)$: 1차 복잡도
    - n
    - $100n$
    - $0.001n + 10000$
    - ...

- $\Theta(n \lg n)$: 엔 로그 엔(n log n) 복잡도
    - $n \lg n$
    - $2n \lg n$
    - $\frac{1}{2}n \lg n + \lg n + 3$
    - ...

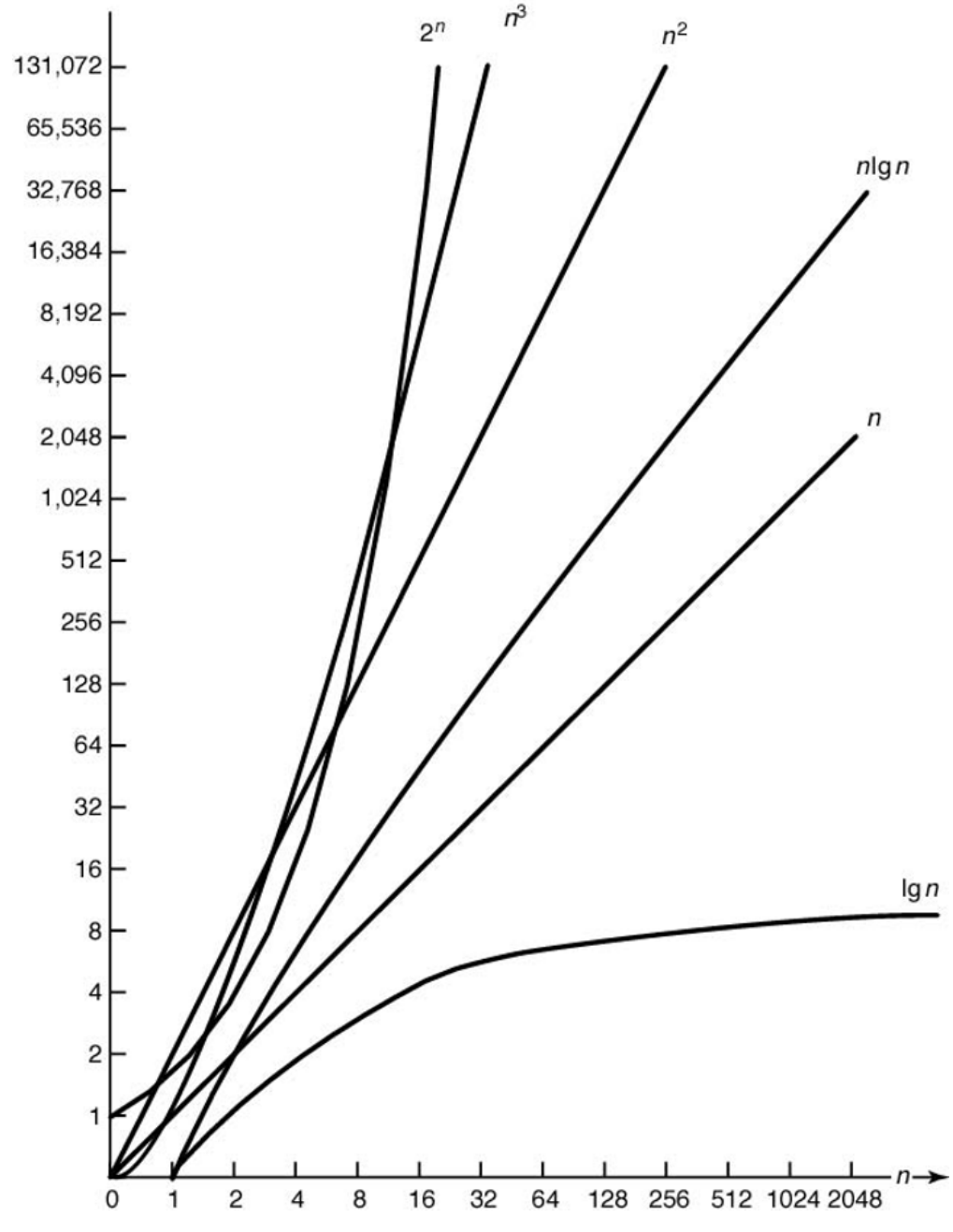
경우에 따라 괜찮은 알고리즘의 복잡도 예제

- $\Theta(n^2)$: 2차 복잡도
    - n^2
    - $5n^2$
    - $0.1n^2 + n + 100$
    - ...

사실상 사용할 수 없는 알고리즘의 복잡도 예제

- $\Theta(2^n)$: 지수 복잡도
    - 2^n
    - $0.001 \cdot 2^n + 5n^3 + 2n + 7$
    - $3 \cdot 2^n + 100n^3 + n + 100$
    - ...

복잡도 함수의 증가율


2^n
n^3
n^2
$n \lg n$
n
$\lg n$
131,072
65,536
32,768
16,384
8,192
4,096
2,048
1,024
512
256
128
64
32
16
8
4
2
1
0 1 2 4 8 16 32 64 128 256 512 1024 2048
n

시간복잡도별 실행시간 비교

- 가정: 단위연산 실행시간 = 1 ns

n	$\lg n$	n	$n \lg n$	n^2	n^3	
10	0.003 μs	0.01 μs	0.033 μs	0.10 μs	1.0 μs	
20	0.004 μs	0.02 μs	0.086 μs	0.40 μs	8.0 μs	
30	0.005 μs	0.03 μs	0.147 μs	0.90 μs	27.0 μs	
40	0.005 μs	0.04 μs	0.213 μs	1.60 μs	64.0 μs	
50	0.006 μs	0.05 μs	0.282 μs	2.50 μs	125.0 μs	
10^2	0.007 μs	0.10 μs	0.664 μs	10.00 μs	1.0 ms	4 ×
10^3	0.010 μs	1.00 μs	9.966 μs	1.00 ms	1.0 초	
10^4	0.013 μs	10.00 μs	130.000 μs	100.00 ms	16.7 분	
10^5	0.017 μs	0.10 ms	1.670 ms	10.00 초	11.6 일	
10^6	0.020 μs	1.00 ms	19.930 ms	16.70 초	31.7 년	
10^7	0.023 μs	0.01 초	0.230 초	1.16 일	31, 709 년	
10^8	0.027 μs	0.10 초	2.660 초	115.70 일	3.17×10^7 년	
10^9	0.030 μs	1.00 초	29.900 초	31.70 년		

- 원서 오류 주의: 0.230 초 (추정치)

차수 정의

- 차수(Θ)를 엄밀하게 정의하려면 "큰 O(big O)"와 "Ω(Omega, 오메가)" 개념 필요

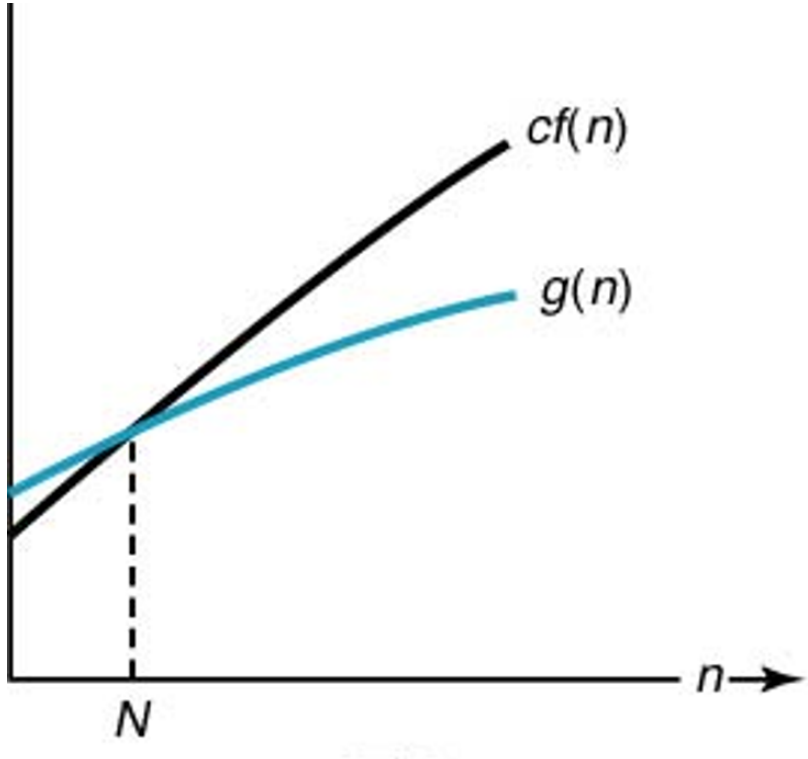
'큰 O' 표기법

- 다음 성질을 갖는 양의 실수 c와 음이 아닌 정수 N이 존재할 때 $g(n) \in O(f(n))$ 성립:

  <div style="text-align:center";>$n \geq N$인 임의의 정수 n에 대해 $g(n) \leq c \cdot f(n)$

- $g(n) \in O(f(n))$ 읽는 방법:
  - $g(n)$은 $f(n)$의 큰 O이다.
  - $g(n)$의 점근적 상한은 $f(n)$이다.

- 의미: 입력크기 n에 대해 시간 복잡도 $g(n)$의 수행시간은 궁극적으로 $f(n)$보다 나쁘지는 않다.

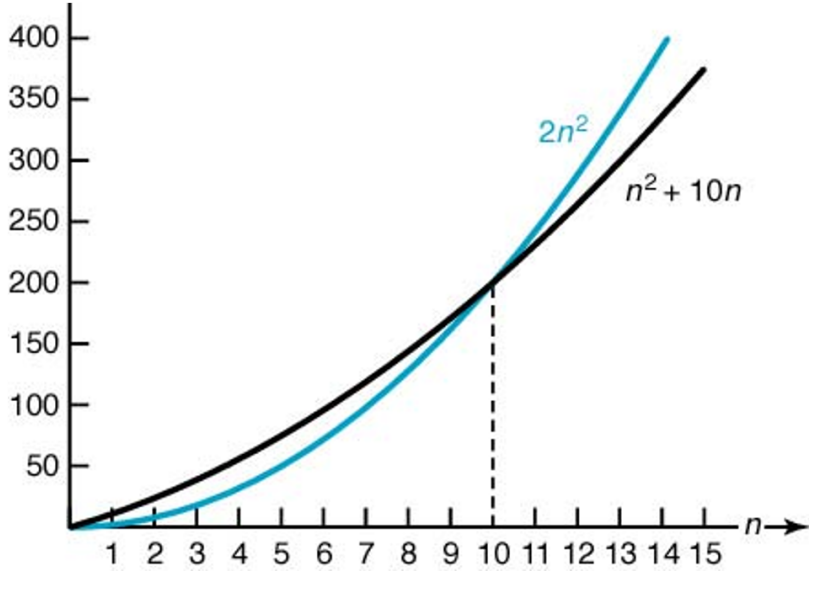


(a) $g(n) \in O(f(n))$

'큰 O' 표기법 예제

- $n^2 + 10n \in O(n^2)$
  - $n \geq 10$인 경우: $n^2 + 10n \leq 2n^2$
    - 그러므로 $c = 2$와 $N = 10$ 선택
  - $n \geq 1$인 경우: $n^2 + 10n \leq n^2 + 10n^2 = 11n^2$
    - 그러므로 $c = 11$와 $N = 1$ 선택

- $2n^2$과 $n^2 + 10n$의 비교



- $5n^2 \in O(n^2)$
  - $n \geq 0$인 경우: $5n^2 \leq 5n^2$
  - 그러므로 $c = 5$와 $N = 0$ 선택

- $T(n) = n(n-1)/2$

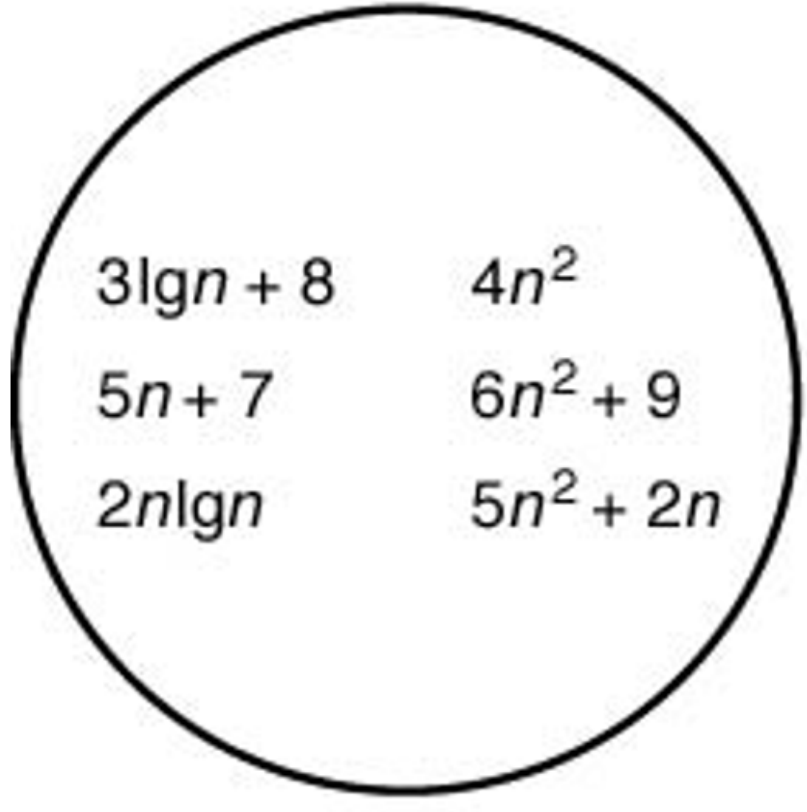
  $\in O(n^2)$
  - $n \geq 0$인 경우: $n(n-1)/2 \leq n^2/2$
  - 그러므로 $c = 1/2$과 $N = 0$ 선택

- $n^2 \in O(n^2 + 10n)$
  - $n \geq 0$인 경우: $n^2 \leq 1 \times (n^2 + 10n)$
  - 그러므로 $c = 1$과 $N = 0$ 선택

- $n \in O(n^2)$
  - $n \geq 1$인 경우: $n \leq 1 \times n^2$ 이 성립한다.
  - 그러므로 $c = 1$과 $N = 1$ 선택

- $n^3 \notin O(n^2)$
  - c와 N을 아무리 크게 지정하더라도, N 보다 큰 어떤 수 n에 대해 $n^3 > c \cdot n^2$ 이 성립한다.
  - 예를 들어, $n > c$로 잡으면 됨.

- $O(n^2)$: 특정 양의 실수 c에 대해 $c\,n^2$ 보다 궁극적으로 작은 값을 가지는 함수들의 집합



(a) $O(n^2)$

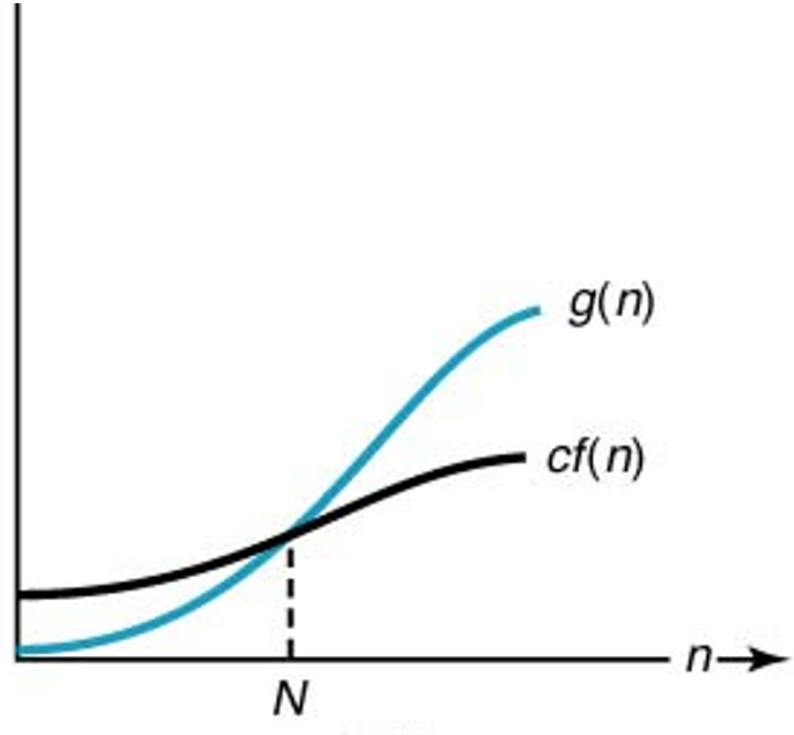
Ω 표기법

- 다음 성질을 갖는 양의 실수 c와 음이 아닌 정수 N이 존재할 때 $g(n) \in \Omega(f(n))$ 성립:

  <div style="text-align:center";>$n \geq N$인 임의의 정수 n에 대해 $g(n) \geq c \cdot f(n)$

- $g(n) \in \Omega(f(n))$ 읽는 방법:
  - $g(n)$은 $f(n)$의 오메가이다.
  - $g(n)$의 점근적 하한은 $f(n)$이다.

- 의미: 입력크기 n에 대해 시간 복잡도 $g(n)$의 수행시간은 궁극적으로 $f(n)$보다 효율적이지 못하다.



(b) $g(n) \in \Omega(f(n))$

Ω 표기법 예제

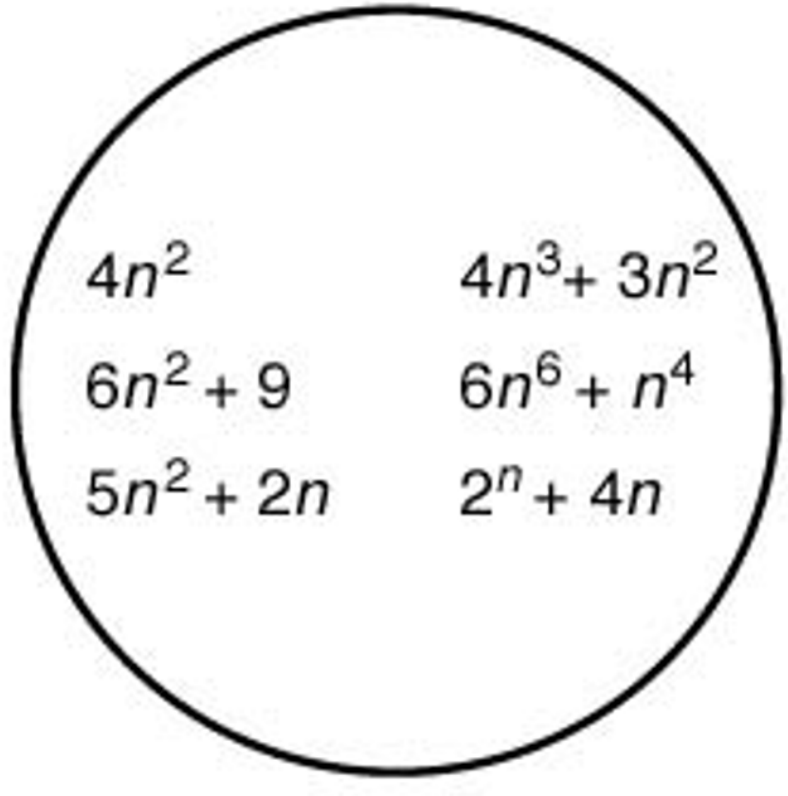
- $n^2 + 10n \in \Omega(n^2)$
  - $n \geq 0$인 경우: $n^2 + 10n \geq n^2$
  - 그러므로 $c = 1$과 $N = 0$ 선택

- $5n^2 \in \Omega(n^2)$
  - $n \geq 0$인 경우: $5n^2 \geq 1 \cdot n^2$
  - 그러므로, $c = 1$과 $N = 0$ 선택

- $T(n) = n(n-1)/2$
  $\in \Omega(n^2)$
  - $n \geq 2$인 경우: $\frac{n(n-1)}{2} \geq \frac{1}{4}n^2$
  - 그러므로 $c = 1/4$과 $N = 2$ 선택

- $n \notin \Omega(n^2)$
  - c를 아무리 작게, N을 아무리 크게 지정하더라도, $n \leq c \cdot n^2$을 만족시키는 $n \geq N$이 존재.
  - 예를 들어, $n \geq 1/c$로 잡으면 됨.

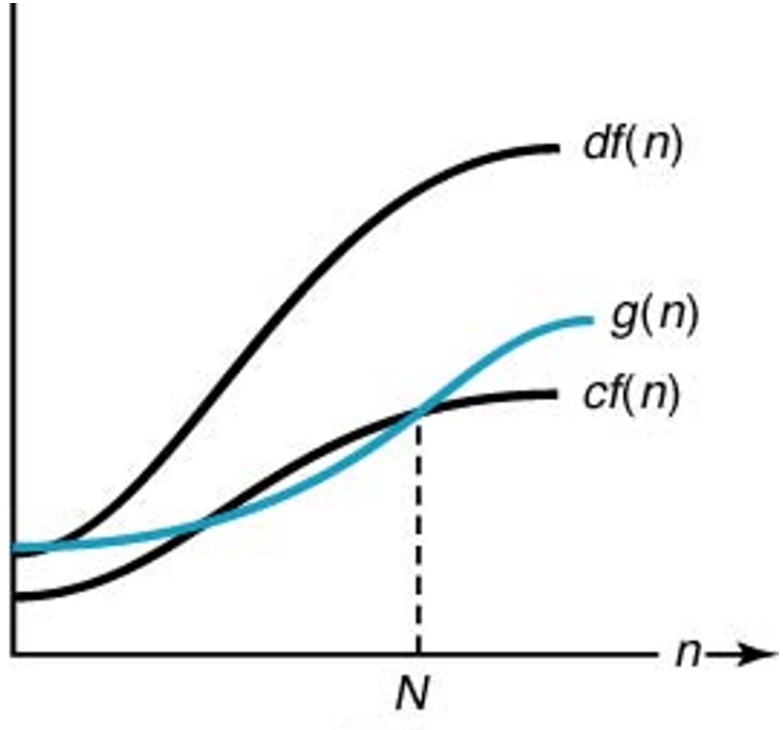
- $\Omega(n^2)$: 특정 양의 실수 c에 대해 $c\,n^2$ 보다 궁극적으로 큰 값을 가지는 함수들의 집합



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

ϴ 표기법

- $g(n) \in \Theta(f(n))$ 읽는 방법:
  - $g(n)$의 $f(n)$의 **차수**이다.



(c) $g(n) \in \theta(f(n))$

Θ 표기법 예제

- $T(n) = n(n-1)/2$:

  $\in \Theta(n^2)$
  - $n \geq 2$인 경우: $n(n-1)/2 \geq \frac{1}{4}n^2$
  - $n \geq 0$인 경우: $n(n-1)/2 \leq \frac{1}{2}n^2$
  - 그러므로, $c = \frac{1}{4}, d = \frac{1}{2}, N = 2.$

작은 *o*(small *o*) 표기법

- 임의의 양의 실수 c에 대해 다음 성질을 갖는 음이 아닌 정수 N이 존재할 때 $g(n) \in o(f(n))$ 성립:

  <div style="text-align:center";>$n \geq N$인 임의의 정수 n에 대해 $g(n) \leq c \cdot f(n)$

- $g(n) \in o(f(n))$ 읽는 방법:
  - $g(n)$은 $f(n)$의 '작은 오(small *o*)이다.

- 의미
    - $g(n)$이 $f(n)$에 비해 궁극적으로 하찮을 만큼 작다.
    - 알고리즘 분석적 측면: 복잡도 $g(n)$이 복잡도 $f(n)$ 보다 궁극적으로 훨씬 좋다.
        - 이유: $c > 0$이 아무리 작더라도, n이 충분히 크면 $g(n) < f(n)$ 성립하기 때문.

큰 O vs 작은 o

- 큰 O: 하나의 양의 실수 c에 대해서 부등식 성립
- 작은 o: 모든 양의 실수 c에 대해서 부등식 성립

작은 o 표기법 예제

- $n \in o(n^2)$
  - $c > 0$가 주어졌을 때, $n \geq 1/c$인 모든 n에 대해 $n \leq c \cdot n^2$ 성립.
- $n \notin o(5n)$
  - $c < 1/5$인 경우, 임의의 음이 아닌 정수 n에 대해 $n > c \cdot 5n$ 성립.
- $n^2 \notin o(5n)$
  - n 이기 때문. $\notin o(5n$)

작은 o 특성

- $g(n)$ $\in o(f(n))$ 이면 다음도 성립:

$$g(n) \in O(f(n)) - \Omega(f(n))$$

- 증명: 생략.

차수의 특성

- $g(n) \in O(f(n)) \quad \Longleftrightarrow \quad f(n) \in \Omega(g(n))$
- $g(n) \in \Theta(f(n)) \quad \Longleftrightarrow \quad f(n) \in \Theta(g(n))$

- 임의의 $a, b > 1$에 대해

$$\log_a n \in \Theta(\log_b n)$$

즉, 로그 함수는 모두 동일한 복잡도 카테고리에 속함.

- $b > a > 0$이면 다음 성립:

$$a^n \in o(b^n)$$

  즉, 지수 함수는 밑수가 다르면 다른 복잡도 카테고리에 속함.

- 임의의 양의 실수 a에 대해 다음 성립:

$$a^n \in o(n!)$$

  즉, $n!$은 어떠한 지수 복잡도함수보다 더 나쁘다(느리다).

- 많이 언급되는 복잡도 카테고리를 순서대로 나열하면 다음과 같음:

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

  - 단, $k > j > 2$이고 $b > a > 1$임.
  - $g(n)$이 $f(n)$의 카테고리 보다 왼편에 위치한 카테고리에 속한 경우 다음 성립:

$$g(n) \in o(f(n))$$

- $c \geq 0, d > 0, g(n) \in O(f(n)), h(n) \in \Theta(f(n))$ 인 경우 다음 성립:

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

예제

- $\Theta(\log_4 n) \in \Theta(\lg n)$
- $\lg n \in o(n)$
- $n^{10} \in o(2^n)$

극한(limit)을 이용하여 차수를 구하는 방법

정리

- $\lim_{n\to\infty} \frac{g(n)}{f(n)}$의 값이
    - 만약 $c > 0$이면, $g(n) \in \Theta(f(n))$,
    - 만약 0 이면, $g(n) \in o(f(n))$,
    - 만약 ∞, 즉, 발산하면, $f(n) \in o(g(n))$.

예제

- $\frac{n^2}{2}$:
  $\in o$
  $(n^3$
  $)$

$$\lim_{n\to\infty} \frac{n^2/2}{n^3} = \lim_{n\to\infty} \frac{1}{2n} = 0$$

- b 일 때, a^n :
  $> a$ $\in o$
  > 0 (b^n)

$$\lim_{n\to\infty} \frac{a^n}{b^n} = \lim_{n\to\infty} \left(\frac{a}{b}\right)^n = 0$$

로피탈(L'Hopital)의 법칙

- $\lim_{n\to\infty} f$ 이 성립하면:
  $(n) =$
  $\lim_{n\to\infty} g(n)$
  $= \infty$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{g'(n)}{f'(n)}$$

예제

- $\lg n \in o(n)$:

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

- $\log_a n \in \Theta(\log_b n)$: