알고리즘 효율성 분석

IT융합공학부 김진웅

유튜브 주소 : https://youtu.be/ieTFAKrJpTo

한성대학교 HANSUNG UNIVERSITY

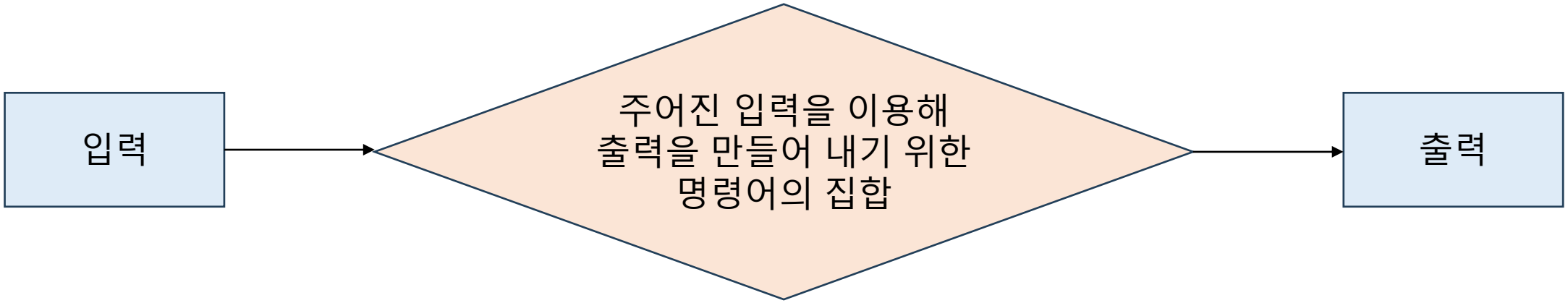
CryptoCraft LAB

- 알고리즘이란?
- 알고리즘의 조건
- 알고리즘 효율성 분석
- 점근적 성능 분석 방법

알고리즘이란?

- 해결해야 할 어떤 문제가 주어졌을 때, 이 문제의 해답을 구하기 위한 절차를 순서대로 명확하게 나타낸 것

- 페르시아의 수학자 **알-콰리즈미(al-Khwarizmi)**의 이름에서 유래되었다.



알고리즘의 조건 (1)

- **입력(well-defined inputs)** : 알고리즘이 입력을 받는다면 모호하지 않고 잘 정의된 입력이어야 한다. 알고리즘은 0개 이상의 입력을 갖는다.

- **출력(well-defined outputs)** : 출력은 명확하게 정의되어야 하며 1개 이상의 출력이 반드시 존재하여야 한다.

- **명확성(clear and unambiguous)** : 각 명령어의 의미는 모호하지 않고 명확해야 한다.

알고리즘의 조건 (2)

- **유한성(finiteness)** : 한정된 수의 단계 후에는 반드시 종료되어야 한다. 즉 무한루프나 이와 유사한 상태로 끝나서는 안 된다.

- **유효성(feasible)** : 명령어들은 현재 실행 가능한 연산이어야 한다. 미래에 개발될 기술 등을 포함해서는 안된다.

알고리즘 효율성 분석

- **시간 효율성(Time efficiency)** : 가장 중요하게 생각되는 평가 요소로 더 빨리 결과를 내는 알고리즘이 더 효율적이다. 실제로 실행시간을 측정할 수도 있지만, 대부분은 이론적으로 복잡도를 분석하는 방법을 사용한다.

- **공간 효율성(Space efficiency)** : 알고리즘이 컴퓨터 메모리를 얼마나 사용하는지를 나타낸다. 더 적은 메모리를 사용하는 알고리즘이 공간 효율성이 더 높고, 더 좋은 알고리즘이다.

시간 효율성 분석

- 알고리즘을 프로그래밍 언어로 구현하고, 실제 컴퓨터상에서 실행시킨 다음 시간을 측정하는 **실행시간 측정 방식**

- 이론적인 복잡도를 기준으로 비교하는 방식(공간 효율성에도 적용 가능)

시간 효율성 분석 – 실행시간 측정 (1)

- 가장 단순하지만 확실한 방법은 알고리즘을 프로그래밍 언어로 구현하고, 실제 컴퓨터상에서 실행시킨 다음 시간을 측정하는 것

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

int factorial(int n);

int main()
{
    int num;
    int result;

    scanf("%d", &num);

    time_t start = time(NULL);
    result = factorial(num);
    time_t end = time(NULL);

    printf("소요시간 : %lf₩n", (double)(end - start));

    return 0;
}

int factorial(int n)
{
    if (n == 1) return 1;

    return n * factorial(n - 1);
}
```

시간 효율성 분석 – 실행시간 측정 (2)

- 위 방식의 가장 큰 단점은 **같은 조건의 하드웨어**를 사용하지 않을 시에는 알고리즘의 시간 효율성을 정확하게 측정하기 어려움

- 실행시간 측정에는 하드웨어의 성능이 절대적인 영향을 미침

VS

시간 효율성 분석 – 실행시간 측정 (3)

- 프로그래밍 언어나 운영체제와 같은 소프트웨어 환경도 같아야 함. **컴파일 방식**이 **인터프리터 방식**보다 더 빠름

- 성능 비교에 사용했던 데이터가 아닌 다른 데이터에 대해서는 다른 결과가 나올 수 있어 **실험되지 않은 입력에 대해서는 실행시간을 주장할 수 없음**

시간 효율성 분석 – 이론적인 알고리즘 복잡도 분석

- 알고리즘에서 **입력의 크기**는 무엇인가?
- 복잡도에 영향을 미치는 가장 핵심적인 **기본 연산**은 무엇인가?
- 입력의 크기가 증가함에 따라 처리시간은 **어떤 형태로 증가**하는가?
- **입력의 특성에 따라 알고리즘 효율성**에는 어떤 차이가 있는가?

이론적인 알고리즘 복잡도 분석 – 입력의 크기

- 알고리즘의 효율성은 보통 입력 크기의 함수 형태로 표현됨
- 따라서 분석할 알고리즘에서 무엇이 입력의 크기를 나타내는지를 먼저 명확히 결정해야 함
- 입력의 크기는 항목의 값이 아닌 **항목의 수**
- 리스트 A에서 key값을 탐색하는 문제에서 입력의 크기는 key값의 크기가 아닌 $\boldsymbol{len(A)}$**가 입력의 크기**
- X의 거듭제곱 값인 X^n를 구하기 위해 X를 n번 곱하는 거듭제곱 알고리즘에서 입력의 크기는 X가 아닌 n , 이때 X의 값이 크다고 처리시간이 늘어나지는 않음

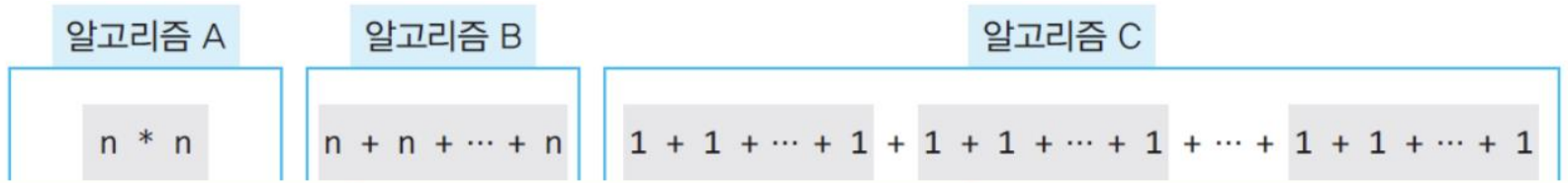
이론적인 알고리즘 복잡도 분석 – 기본 연산 (1)

- 입력의 크기가 결정되면 다음으로 알고리즘의 **실행시간 측정을 위한 단위**를 결정해야 함

- 알고리즘에서 가장 많이 실행되는 연산, 즉 **기본 연산(Basic Operation)**을 찾고, 이 연산이 실행되는 횟수만을 계산하는 것
  - 다중 루프의 경우 가장 안쪽 루프에 있는 연산이 가장 많이 실행됨

이론적인 알고리즘 복잡도 분석 – 기본 연산 (2)

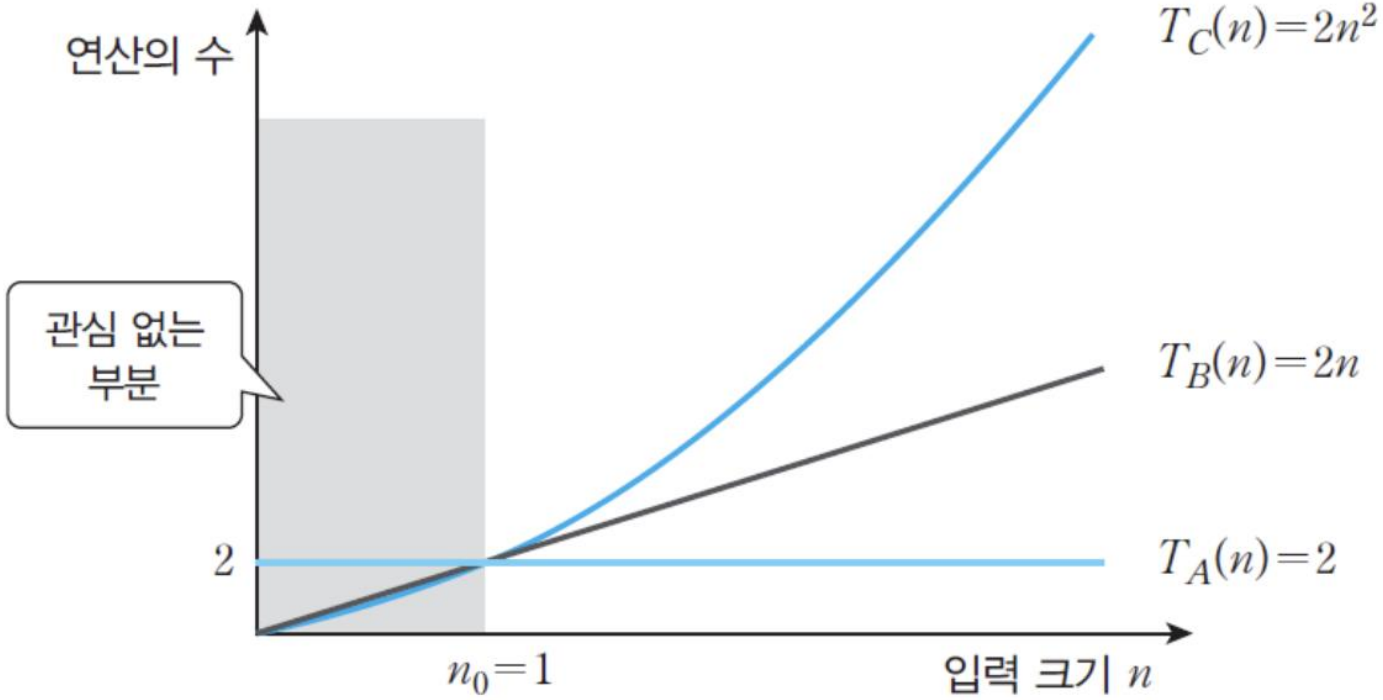
- n의 거듭제곱 알고리즘 3가지 비교



	알고리즘 A	알고리즘 B	알고리즘 C
유사코드	`sum ← n * n`	1. `sum ← 0` 2. `for i←1 to n do` 3. `sum ← sum + n`	1. `sum ← 0` 2. `for i←1 to n do` 3. `for j←1 to n do` 4. `sum ← sum + 1`
전체 연산 횟수	대입 연산: 1 곱셈 연산: 1 전체 횟수: 2	대입 연산: $n+2$ 덧셈 연산: n 전체 횟수: $2n+1$	대입 연산: n^2+n+2 덧셈 연산: n^2 전체 횟수: $2n^2+n+2$

이론적인 알고리즘 복잡도 분석 – 증가 속도

- 알고리즘에서 기본 연산이 실행되는 전체 횟수는 입력의 개수 n의 영향을 받는데, **연산의 수를 n의 함수로 나타낸 것**을 **시간 복잡도 함수** ($T(n)$)이라고 표기함
- 시간 복잡도 함수는 **전체 횟수의 최고 차항**만 표기함
- $T_A(n) = 2,\ T_B(n) = 2n,\ \ T_C(n) = 2n^2$

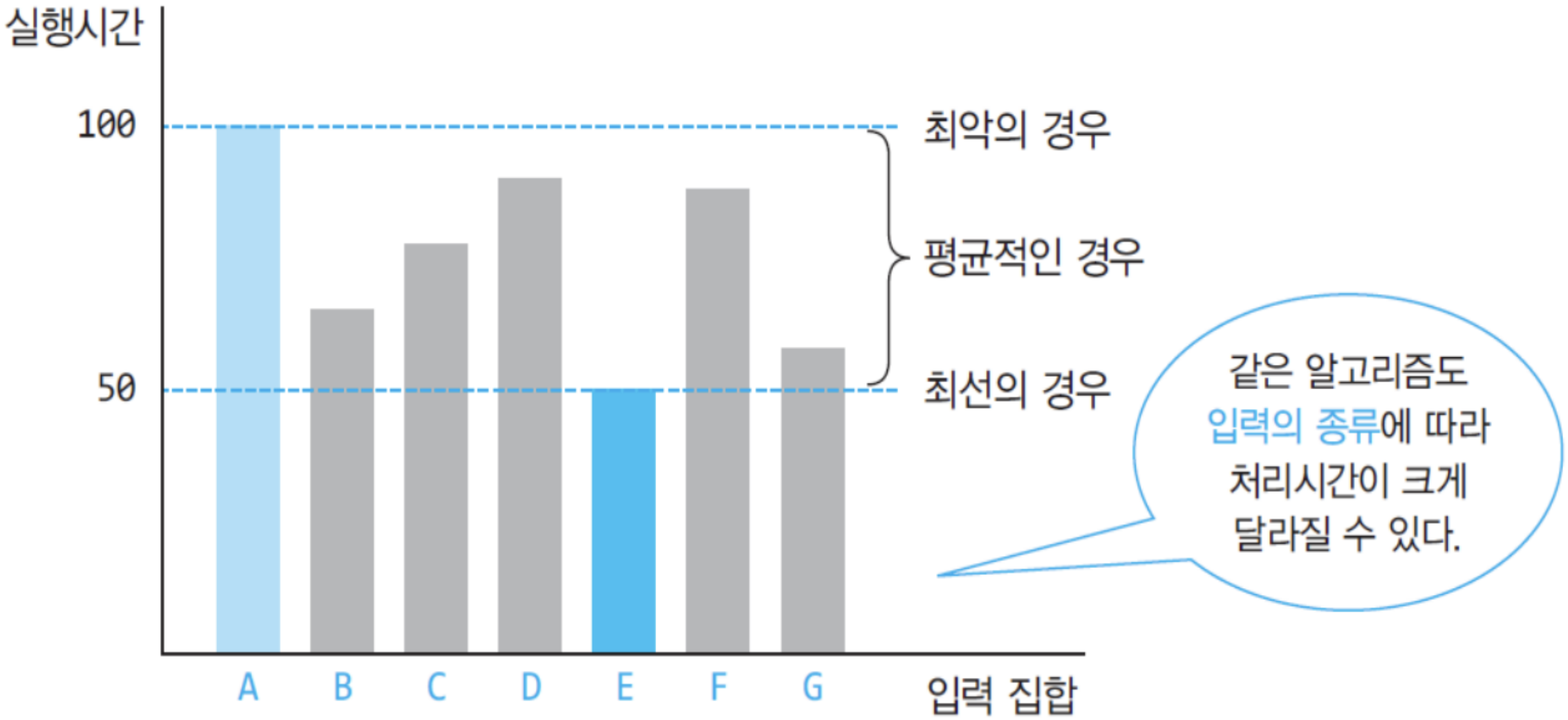


이론적인 알고리즘 복잡도 분석 – 입력의 특성 (1)

- 같은 알고리즘도 입력의 종류 또는 구성에 따라 다른 특성의 실행시간을 보일 수 있음
- 알고리즘의 효율성은 입력의 특징에 따라 3가지로 나누어 평가

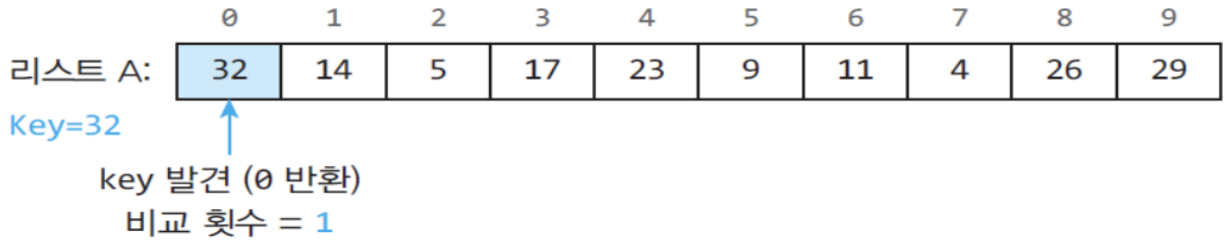
1. **최선의 경우(best case)** : 실행시간이 가장 적은 경우를 말하는데, 알고리즘 분석에서는 큰 의미가 없음
2. **평균적인 경우(average case)** : 알고리즘의 모든 입력을 고려하고 각 입력이 발생할 확률을 고려한 평균적인 실행시간을 의미
3. **최악의 경우(worst case)** : 입력의 구성이 알고리즘의 실행시간을 가장 많이 요구하는 경우를 말하는데, 가장 중요하게 사용됨

이론적인 알고리즘 복잡도 분석 – 입력의 특성 (2)

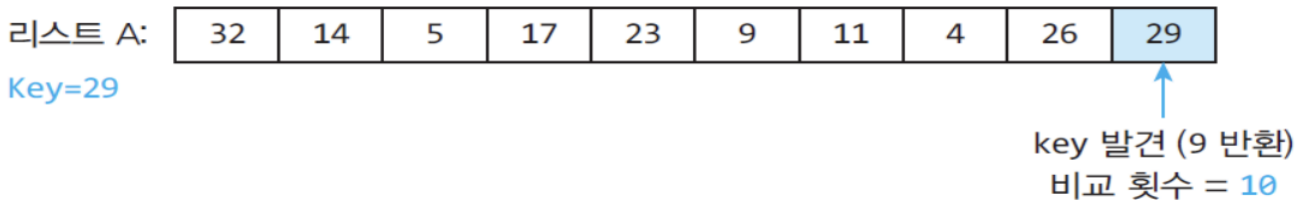


이론적인 알고리즘 복잡도 분석 – 입력의 특성 (3)

- **최선의 경우** : 리스트 A의 첫 번째 항목이 key와 같은 경우로 항상 한번만에 탐색이 완료되고, 시간 복잡도 함수 $T_{best}(n) = 1$ 이다.



- **최악의 경우** : 탐색키 key가 리스트에 없거나 맨 뒤에 있는 경우로 시간 복잡도 함수 $T_{worst}(n) = n$ 이다.



이론적인 알고리즘 복잡도 분석 – 입력의 특성 (4)

- **평균적인 경우** : 각 숫자가 key로 사용될 가능성이 $\frac{1}{n}$로 모두 같다고 가정한 후, 모든 숫자를 탐색했을 때 비교 연산의 횟수를 모두 더한 다음, 이것을 전체 숫자의 개수로 나누어주면 평균적인 경우의 비교 연산 수행 횟수를 알 수 있음

$$T_{avg}(n) = \frac{1 + 2 + \cdots + n}{n} = \frac{n(n+1)/2}{n} = \frac{n+1}{2}$$

- 평균적인 경우는 계산하기 어려운 경우가 많고, 특히 최악의 상황에 대한 시간을 보장하지 못한다는 문제가 있음

효율성 분석 요약

- 알고리즘의 시간 효율성과 공간 효율성은 모두 **입력의 크기(개수) n**의 함수로 나타낸다.
- **시간 효율성**은 알고리즘의 기본 연산의 실행 횟수를 이용해 구하고, **공간 효율성**은 알고리즘에 의해 사용되는 추가적인 메모리양을 이용해 구한다.
- 어떤 알고리즘은 입력의 크기가 같더라도 입력의 구성에 따라 효율이 매우 다르게 나타날 수 있다. 이런 알고리즘에서는 **최선**, **평균**, 그리고 **최악의 경우에 대한 분석**이 추가적으로 **필요**하다.
- 알고리즘의 효율성 분석은 입력의 크기가 무한대로 커짐에 따라 **실행시간(또는 추가적인 메모리 사용량)이 어떤 속도로 증가하는지에만 관심이 있다**.

점근적 성능 분석 방법

- 알고리즘의 복잡도 함수 $T(n)$은 입력의 크기 n에 대한 수식으로, 보통 여러 개의 항을 가진 다항식의 형태가 됨
- n이 커질수록 **차수가 최대인 항의 영향이 절대적**이고 나머지 항들은 무시할 수 있을 정도가 됨
- 예를 들어, 복잡도 함수 $T(n)=n^2+n+1$에서 최고 차항인 n^2의 영향을 계산해보면
- $n=1$ 일 때 : $T(n) = \mathbf{1}+1+1=3$ (n^2항이 33.3%)
- $n=10$ 일 때 : $T(n) = \mathbf{100}+10+1=111$ (n^2항이 90%)
- $n=100$ 일 때 : $T(n) = \mathbf{10000}+100+1=10101$ (n^2항이 99%)
- $n=1000$ 일 때 : $T(n) = \mathbf{1000000}+1000+1=1001001$ (n^2항이 99.9%)

점근적 표기(Asymptotic notation) (1)

- 입력의 크기 n이 무한대로 커질 때의 복잡도를 간단히 표현하기 위해 사용하는 방법으로 여러 항을 갖는 복잡도 함수를 **최고차항만을 계수 없이 취해 단순화 시킨다**.

- **n이 충분히 크다면** $3n^2 - 100n + 10000$는 n^2으로, $2n! + 100^n$은 $n!$로 단순화 됨

- 효율성 분석에서 중요한 것은 n에 대해 “연산이 정확히 몇 번 필요한가?” 가 아니라 n이 증가함에 따라 “**무엇에 비례하는 수의 연산이 필요한가?**” 이다.

점근적 표기(Asymptotic notation) (2)

- 점근적 표기는 함수에 **상한(Upper bound)**과 **하한(Lower bound)** 및 **동일 등급(Tight bound)**과 같은 개념을 적용하여 **빅오(big O)**, **빅오메가(Big omega)**와 **빅세타(Big theta)** 표기로 나눌 수 있다.

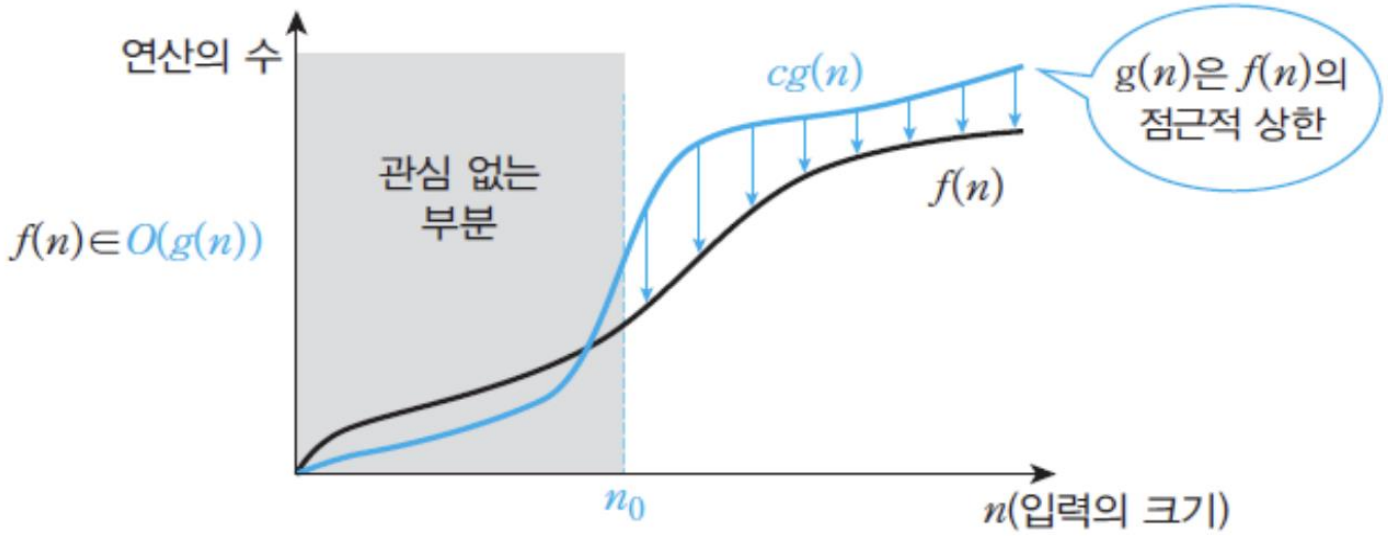
- **빅오 표기법**은 복잡도 함수의 상한을 나타냄.

- **빅오메가 표기법**은 복잡도 함수의 하한을 나타냄.

- **빅세타 표기법**은 복잡도 함수의 점근적 상한인 동시에 점근적 하한임을 나타냄

점근적 표기(Asymptotic notation) – 빅오 표기법

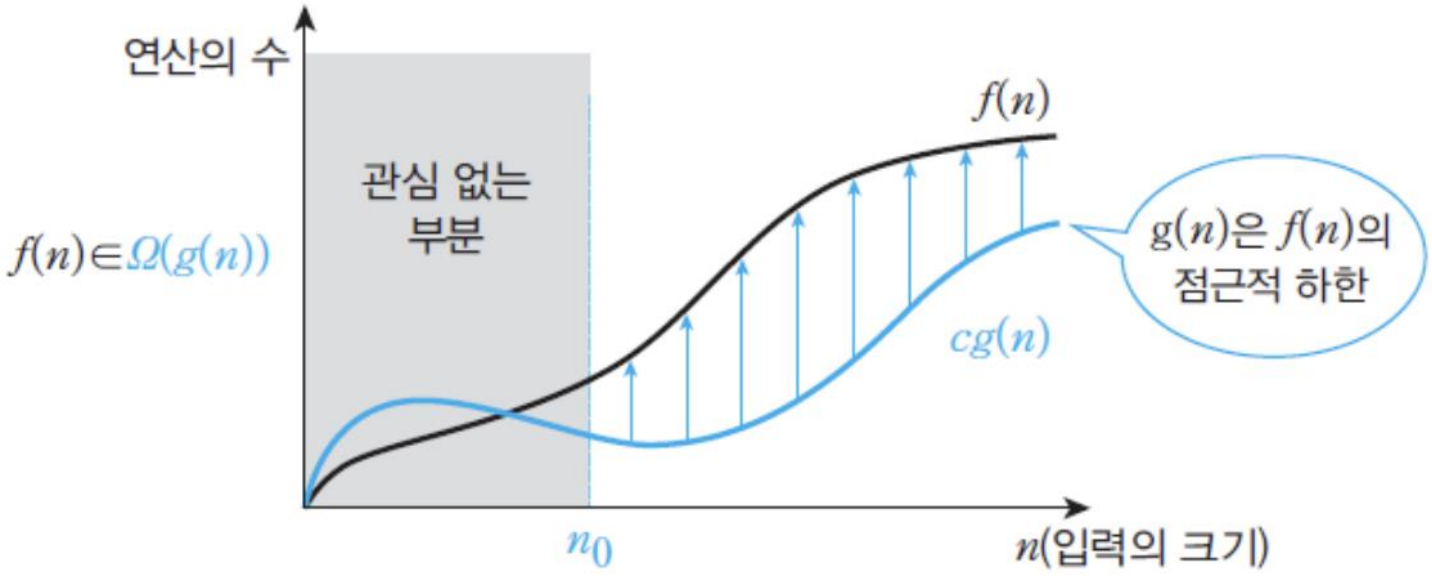
- **정의** : 복잡도 함수 $f(n)$이 주어졌을 때 $n \geq n_0$인 모든 정수 n에 대해 $|f(n)| \leq c|g(n)|$을 만족하는 양의 상수 c와 자연수 n_0가 존재하면 $f(n) \in O(g(n))$이다.



- $n < n_0$ 인 모든 n에 대해서는 전혀 관심이 없다

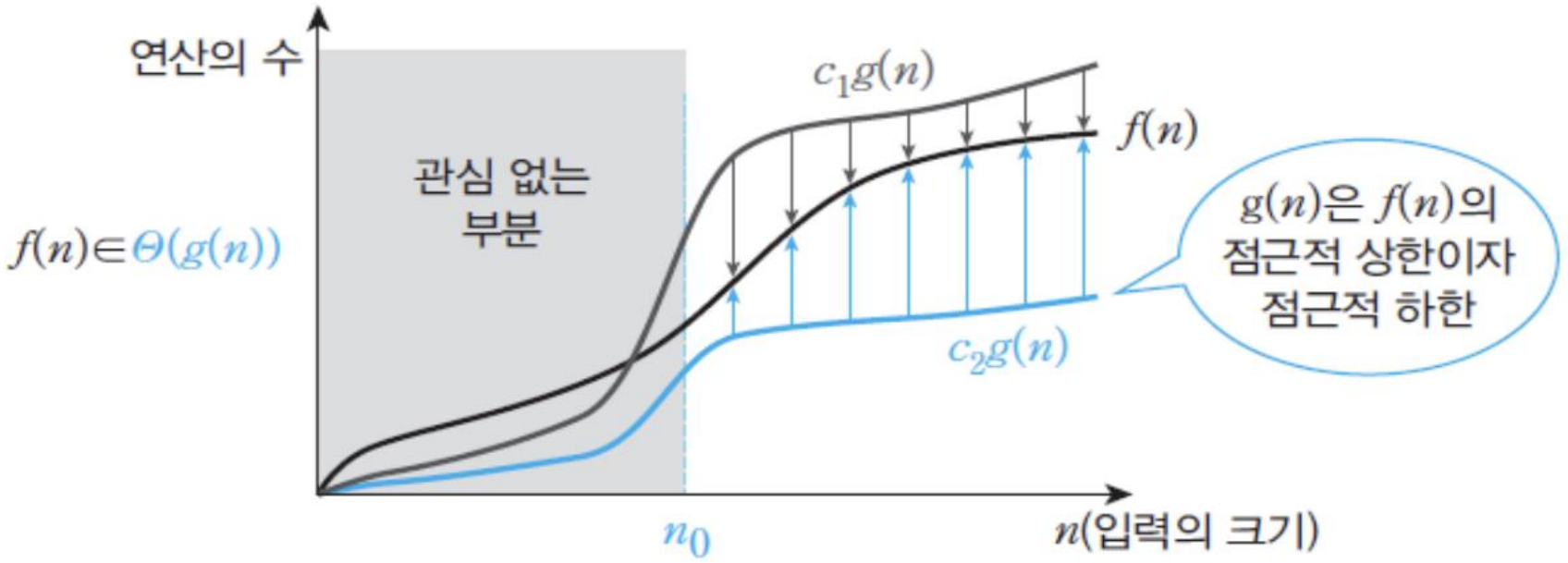
점근적 표기(Asymptotic notation) – 빅오메가 표기법

- **정의** : 복잡도 함수 $f(n)$이 주어졌을 때 $n \geq n_0$인 모든 정수 n에 대해 $|f(n)| \geq c|g(n)|$을 만족하는 양의 상수 c와 자연수 n_0가 존재하면 $f(n) \in \Omega(g(n))$이다.



점근적 표기(Asymptotic notation) – 빅세타 표기법

- **정의** : 복잡도 함수 $f(n)$이 주어졌을 때 $n \geq n_0$인 모든 정수 n에 대해 $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$을 만족하는 양의 상수 c와 자연수 n_0가 존재하면 $f(n) \in \Omega(g(n))$이다.



알고리즘의 점근적 성능 클래스들

$O(1)$: 상수형

$O(\log n)$: 로그형

$O(n)$: 선형

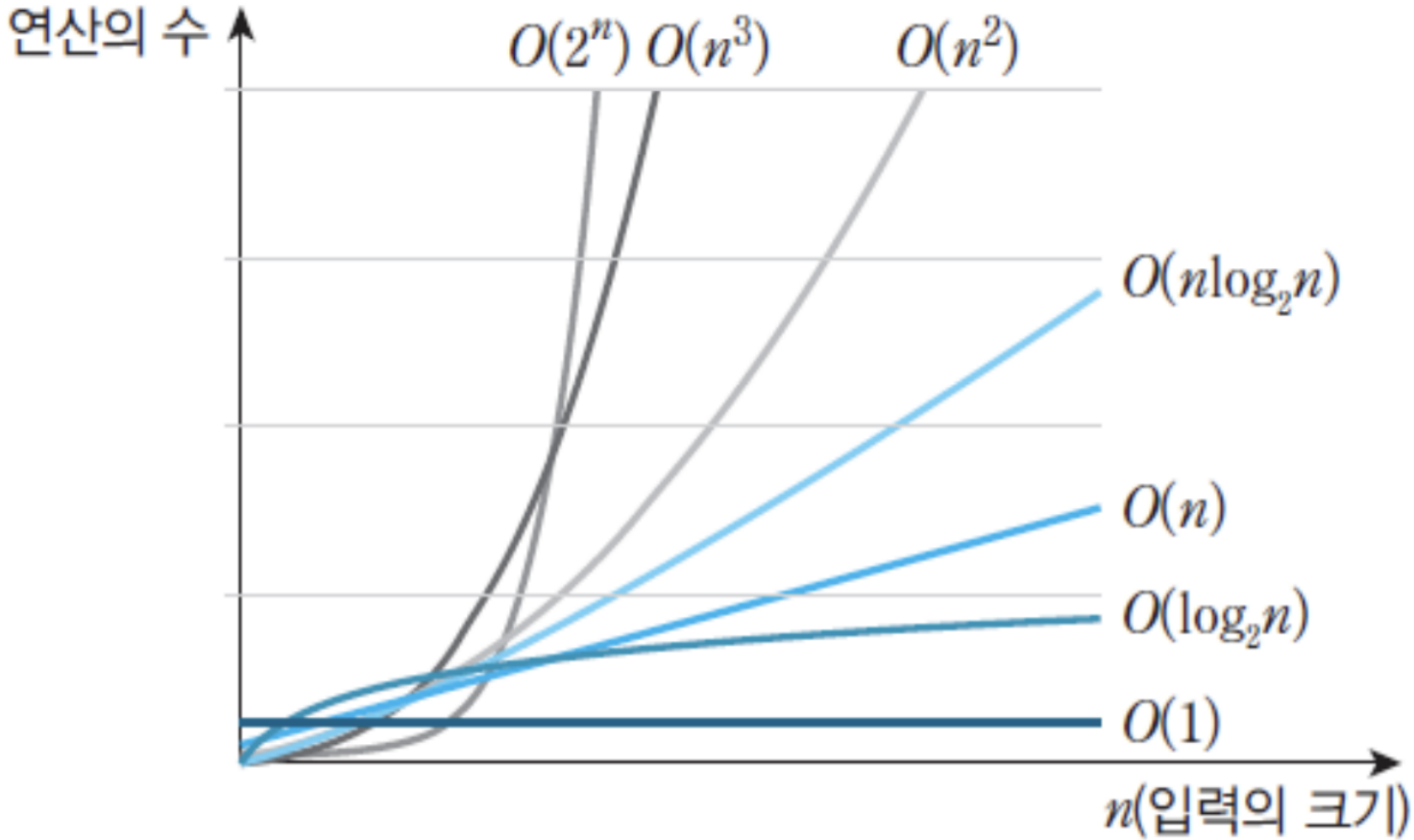
$O(n\log n)$: 선형로그형

$O(n^2)$: 2차형

$O(n^3)$: 3차형

$O(2^n)$: 지수형

$O(n!)$: 팩토리얼형



Q & A