Lecture 2: Algorithm analysis

Algorithm

Taegyeom Lee

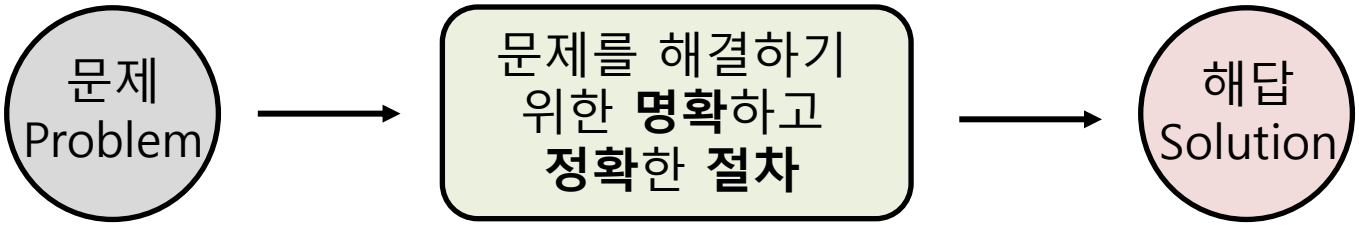
Table of Contents

Part 1

PRELIMINARIES

Preliminaries

❖ 알고리즘(algorithm) 이란?



사과 6개를
마트에서 사서
집으로 돌아오자

1.마트를 간다
2.사과를 산다
3.집으로 온다

Preliminaries

❖ 문제(Problem)란?

- 답을 찾기 위한 질문 (a question to which we seek an answer)

❖ 문제 예시

임의의 x, y ∈ I (실수)에 대해서, x ≤ y이면 f(x) ≤ f(y)일 경우 증가함수라고 하고 f가 단조증가 한다고 한다.

- 문제 #1: 단조증가 순서로 n개의 정수를 가진 리스트 S를 정렬 하세요.
  - 답: 오름차순으로 정렬된 리스트S

- 문제 #2: n개의 정수가 포함된 리스트 S에 정수 x가 존재하는지를 확인하라. 존재하면 yes로, 존재하지 않으면 no로 답하라.
  - 답: YES or NO

Preliminaries

- 매개변수 (parameter)란?
  - 문제에서 언급된 할당되지 않은 (**구체적인 값이 없는) 변수들**

- 문제 예시
  - 문제 #1: 단조증가 순서로 n개의 정수를 가진 리스트 S를 정렬 하세요.
    - 매개변수: n, S
    - 답: 오름차순으로 정렬된 리스트S
  - 문제 #2: 정수 x가 n개 정수 리스트 S에 존재하는지를 결정하라.
    존재하면 yes로, 존재하지 않으면 no로 답하라.
    - 매개변수: n, S, x
    - 답: YES or NO

Preliminaries

- 실체란 (instance)?
  - 매개변수에 실제로 할당된 (구체적인 값이 있는) 값

- 문제 예시
  - 문제 #1: 단조증가 순서로 n개 정수의 리스트 S를 정렬 하시오.
    - 매개변수 및 실체: n=6, S=[10,7,11,5,13,8]
    - 답: 오름차순으로 정렬된 리스트S
  - 문제 #2: 정수 x가 n개 정수 리스트 S에 존재하는지를 결정하라. 존재하면 yes로, 존재하지 않으면 no로 답하라.
    - 매개변수 및 실체: n =6, S=[10,7,11,5,13,8], x = 5
    - 답: YES or NO

Preliminaries

- 알고리즘 (Algorithm) 이란?
  - 어떤 문제를 풀기 위한 유한한 절차와 방법 (유한->방법이 한정적이다(x))
  - 여기서 문제는 보통 '수학적으로 엄밀히' 정의 된 문제에 한정 됨.

- 절차와 방법?
  - 계산 문제를 풀기 위해서는?
    - 숫자와 사칙연산을 이용해서 풀이
  - 컴퓨터로 계산 문제를 풀기 위해서는?
    - 컴퓨터에 주어진 명령어 집합(instruction set)을 이용

Example of Algorithm #1

- 문제:
  - n개의 정수로 구성된 리스트 S의 모든 요소를 합을 구하시오
- 입력:
  - 양의 정수 n, 첨자 1에서 n 까지로 구성된 정수목록 S
- 출력:
  - 리스트 S에 존재하는 n개의 정수들의 합
- Algorithm:

```
void Algorithm1(int n, const int S[]){
    int location, sum = 0;    // index
    for (location = 0; location < n; location ++)
        sum = sum+S[location];
    return sum;
}
```

의사코드(Pseudo-code)
- 의사(疑似: 비교할 의, 비슷할 사)
- Pseudo: 가짜의~

알고리즘으로 수행할 절차를
다양한 언어로 간략하게
서술해 놓은 것

Example of Algorithm #2

- 문제:
  - n개의 정수로 구성된 리스트 S에 정수 x가 어디에 위치해 있는가?
- 입력:
  - 양의 정수 n, 첨자 1에서 n 까지로 구성된 정수목록 S, 정수 x
- 출력:
  - x가 S에 존재하지 않으면 -1, 존재하면 location 반환 (location: S의 인덱스)
- Algorithm:

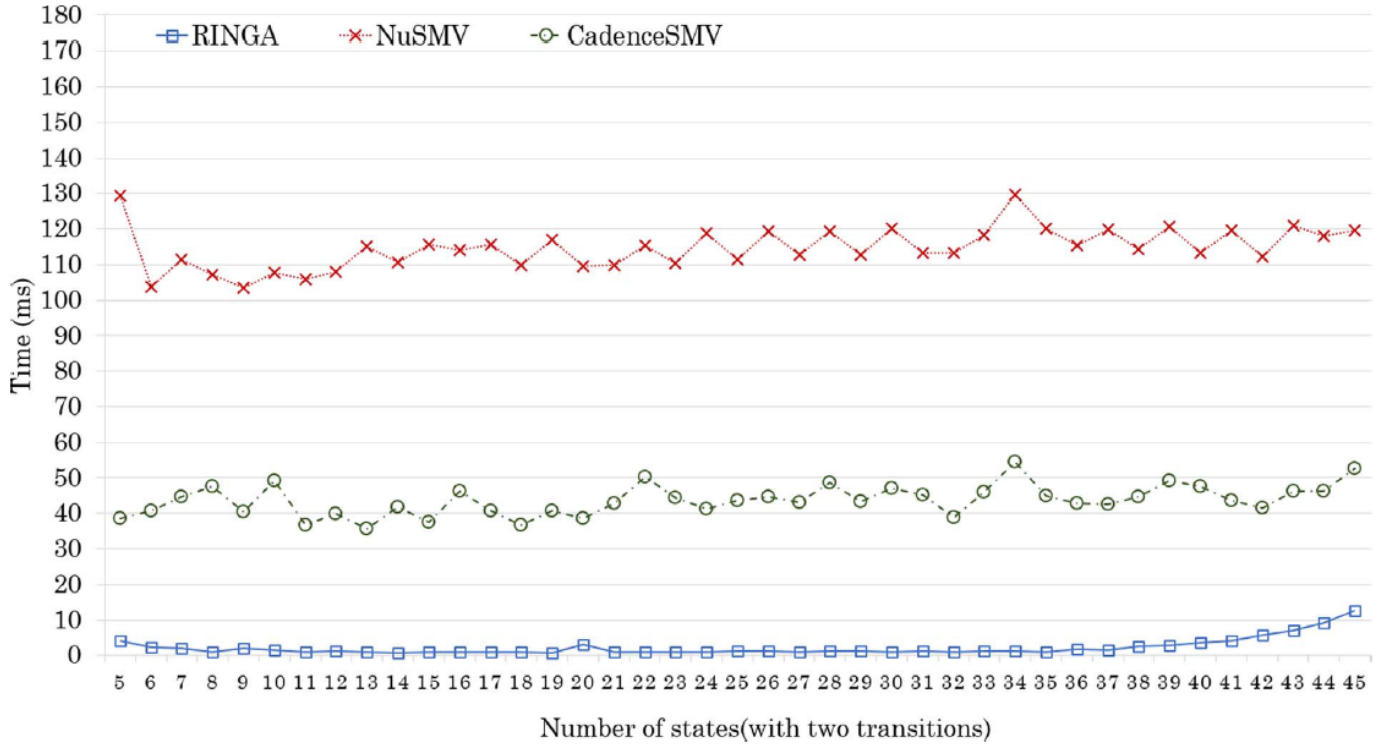
```
void Algorithm2(int n, const int S[], int x, index &location)
{
        int location;    // index

        for (location = 0; location< n; location++)
                if (S[location] == x)
                        break;
        if (location < n)
                return location;
        else
                return -1;
}
```

Part 2

THEORETICAL ANALYSIS VS. EXPERIMENTAL ANALYSIS

Experimental vs. Theoretical Analysis

- 알고리즘의 성능을 분석하는 방법
  - 실험적 분석 (Experimental analysis)
    - 주어진 알고리즘을 소스코드로 구현한 다음, 실제 환경에서 동작 시켜 실제 실행 시간을 측정



Experimental vs. Theoretical Analysis

❖ 알고리즘의 성능을 분석하는 방법

- 실험적 분석 (Experimental analysis)
  - C 언어에서는 clock(), time() 함수를 이용해서 알고리즘의 동작 시간을 측정 가능
    - clock()은 ms단위로 시간을 측정
    - time()은 1초 단위로 측정

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

int main() {
    clock_t start = clock();

    int sum = 0;
    for (int i = 0; i < 10000; ++i)
        for(int j = 0; j < 10000; ++j)
            sum += 1;
    clock_t end = clock();
    printf("소요 시간: %lf₩n", (double)(end - start) / CLOCKS_PER_SEC);
}
```

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

int main() {
    int sum = 0;
    time_t start = time(NULL);

    for (int i = 0; i < 10000; i++)
        for (int j = 0; j < 10000; j++)
            sum += i * j;

    time_t end = time(NULL);
    printf("소요시간: %lf₩n", (double)(end - start));
}
```

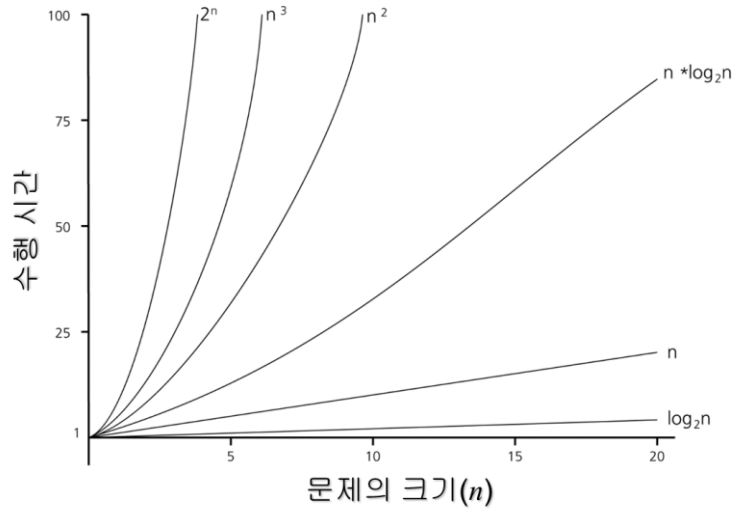
Experimental vs. Theoretical Analysis

- 알고리즘의 성능을 분석하는 방법
  - **실험적 분석의 문제점**
    - 알고리즘을 실제 구현하기까지 시간과 노력이 소모.
    - 기본적으로 명시된 것들 (하드웨어 사양), 측정할 수 없는 다양한 외부 요인들 (코딩 스타일, 현재 컴퓨터 외부 환경 상태) 로 인해 정확한 비교가 힘듦.

Experimental vs. Theoretical Analysis

❖ 알고리즘의 성능을 분석하는 방법

- **이론적 분석 (Theoretical analysis)** ← *수업에서 주로 사용할 방법*
  - 알고리즘 수행 시간 (= 성능) 을 실제 구현을 통해서가 아닌, high-level 에서 이론적으로 기술하는 방법.
  - 사용하는 하드웨어 및 소프트웨어와 무관하게 알고리즘의 성능을 표현 가능.
  - 시간은 보통 입력 사이즈 (보통 n으로 표기) 에 관한 함수로 표현됨.
  - 이론적 분석을 통해 구한 알고리즘 수행 시간을 시간 복잡도 (time complexity) 라고 함.



알고리즘의 실행시간이 $(n, \log_2 n, n * \log_2 n \ldots)$이고, 입력사이즈가 5, 10, 15,… 일 때, 어떤 알고리즘이 가장 좋을까?

Experimental vs. Theoretical Analysis

- 이론적 분석에서 고려하는 기본 연산들
  - 숫자를 변수에 대입
  - 함수를 호출, 함수의 결과값을 반환
  - 두 (작은) 정수 사이의 사칙 연산
  - Array 의 Get, Set 연산
  - 기타 컴퓨터 하드웨어에서 정의 된 단일 명령어

- 위의 각각의 연산들이 걸리는 시간은 하드웨어 및 소프트웨어와 관계가 있으므로 정확한 시간은 알 수 없음.

- 하지만 각각의 단일 연산들은 입력 크기와는 무관하다는 것을 알 수 있음.

- 이 경우 이론적으로 이 연산들은 상수 시간 (constant time) 이 소모된다고 함.
  - 상수시간: 입력 크기 n과 상관없이 항상 같은 시간이 걸리는 연산

Example of Algorithm Theoretical Analysis #1

- 문제:
  - n개의 정수로 구성된 리스트 S의 모든 요소를 합을 구하시오
- 입력:
  - 양의 정수 n, 첨자 1에서 n 까지로 구성된 정수목록 S
- 출력:
  - 리스트 S에 존재하는 n개의 정수들의 합 (location: S의 인덱스)
- Algorithm:

```
void Algorithm1(int n, const int S[], int location){
    int location // index                          ◄····· 0
    int sum = 0;                                   ◄····· 1
    for (location = 0; location < n; location ++)  ◄····· 1+(n+1)+n
        sum = sum+S[location];                     ◄····· n(1+1+1)
    return sum;                                    ◄····· 1
}
```

$1 + 1 + (n + 1) + n + 3n + 1 = \mathbf{5n + 4}$

Example of Algorithm Theoretical Analysis #2

- 문제:
  - n개의 정수로 구성된 리스트 S에 정수 x가 있는가?
- 입력:
  - 양의 정수 n, 첨자 1에서 n 까지로 구성된 정수목록 S, 정수 x
- 출력:
  - x가 S에 존재하지 않으면 0, 존재하면 1 반환 (location: S의 인덱스)
- Algorithm:

```
void Algorithm2(int n, const int S[] , int x, int location)
{
      int location;   // index                                  ◄ 0

      for (location = 0; location < n; location++)             ◄ 1 + (n + 1) + n
            if (S[location] == x)                              ◄ n(1 + 1)
                  break;
      if (location < n)                                        ◄ 1
            return location;                                   ◄ 1
      else
            return -1;                                         ◄ 1
}
```

$1 + (n + 1) + n + 2n + 3 = \mathbf{4n + 5}$

Part 3

ASYMPTOTIC NOTATION

이론적인 분석은 어떻게 수행해야 할까?

- 이론적 분석에서는 걸리는 시간을 표현한 함수 자체보다 입력 사이즈에 비례해서 해당 함수가 어느 정도로 증가하는가에 더 관심이 많음
  - e.g., 입력이 10배 커지면 시간은 얼마나 늘어 날까?
- 앞선 상수시간(constant time) 연산들은 입력 사이즈와 무관.
- 예 : **컴퓨터 1** 보다 모든 기본 연산들이 10배 빠른 **컴퓨터 2**가 있다고 했을 때,
  - 실제 성능에서도 복잡한 알고리즘일수록 기본 연산들의 속도차이는 거의 무의미 해짐

시간 복잡도 (n= 입력 크기)	**컴퓨터 1이 k개 의 자료를 처리할 동안 컴퓨터 2가 처리할 수 있는 자료 수**
n	10k 개
n^3	약 2k 개
2^n	약 k+3 개

점근적 분석(Asymptotic Analysis)

- 이론적인 분석을 통해 알고리즘을 평가할 때, 하드웨어의 성능보다는 알고리즘의 복잡도가 더 큰 영향을 미침
- 따라서, 입력크기 n이 매우 커질 때 **(→∞)** 알고리즘의 성능이 어떤 형태로 변하는지 분석할 필요가 있음
- 이미 알고있는 점근적 개념의 예 $\lim_{n\to\infty} f(n)$
- $\lim_{n\to\infty}\left(\frac{2n^2+3n}{n}\right)$
- 알고리즘 A의 시간 복잡도가 $\frac{2n^2+3n}{n}$이고 입력크기 n이 무한히 커질 때, 어떤 항이 가장 큰 영향력을 행사하는가?
- 시간 복잡도 ≈ n
  - 알고리즘 A의 시간복잡도는 최대 n에 비례하는 시간이 걸림
  - n 보다는 느릴 수 없음
- 점근적 분석이란?
  - 시간복잡도에 가장 큰 영향을 미치는 **대표항만을 남겨서 나타내는 방법**

빅-오 표기법(Big-Oh notation)

❖ **Big-Oh notation (Big-Oh 표기법)은?**

- 알고리즘의 소요시간이 입력크기 n에 대해서 기껏해야 걸리는 시간
  - e.g., 시간복잡도가 $\approx n^2$ 면 그 알고리즘은 **최악의 경우에** n^2 시간이 소요되며, $O(n^2)$ 와 같이 표기할 수 있음
- 최악이어도 이 정도 성능은 나와!
- $O(f(n))$은 점근적 증가율이 $f(n)$ **을 넘지 않는 모든 함수의 집합**
  - $5n^2+4n = O(n^2)$
  - $7n, 5n^2+1, n\log n+3, n^2, 500 = O(n^2)$

빅-오 표기법(Big-Oh notation)

❖ **Big-Oh notation (Big-Oh 표기법)의 수학적 정의**

- O(g(n))은 점근적 증가율이 g(n) **<u>을 넘지 않는 모든 함수의 집합</u>**

- f(n)과 g(n)를 자연수에서 실수로의 함수라고 하고, 모든 n > n_0 에 대하여 f(n) ≤ c•g(n)
  이 조건을 만족하는 어떤 실수 c>0와 자연수 n_0>0이 존재하면, f(n) = O(g(n))이라고 한다.

- $O(g(n)) = \{ f(n) \mid \exists c > 0, n_0 \geq 0 \text{ s.t.} \forall n \geq n_0, cg(n) \geq f(n) \}$
- $O(g(n)) = \{ f(n) \mid$ 모든 $n \geq n_0$에 대하여 $cg(n) \geq f(n)$ 인 양의 상수 c와 n_0가 존재한다$\}$
- $O(g(n)) = \{ f(n) \mid$ 충분히 큰 모든 n 에 대하여 $cg(n) \geq f(n)$ 인 양의 상수 c가 존재한다$\}$

빅-오 표기법(Big-Oh notation)

❖ **Big-Oh notation (Big-Oh 표기법)의 표기법 예시 #1[상수 함수]**

- f(n)과 g(n)를 자연수에서 실수로의 함수라고 하고, 모든 $n > n_0$ 에 대하여 $f(n) \leq c \cdot g(n)$ 이 조건을 만족하는 어떤 실수 $c>0$와 자연수 $n_0>0$이 존재하면, $f(n) = O(g(n))$이라고 한다.
- f(n) = 30 → 상수 함수 (constant function) : 입력 크기 (n) 과 상관 없이 항상 일정한 값을 가지고 있음.
- **f (n) = O(1)**
  - 증명 : 앞선 정의에서 c=31 로 두면 됨.

빅-오 표기법(Big-Oh notation)

❖ **Big-Oh notation (Big-Oh 표기법)의 표기법 예시 #2[일차 함수]**

- f(n)과 g(n)를 자연수에서 실수로의 함수라고 하고, 모든 $n > n_0$ 에 대하여 $f(n) \leq c \cdot g(n)$ 이 조건을 만족하는 어떤 실수 $c>0$와 자연수 $n_0>0$이 존재하면, f(n) = O(g(n))이라고 한다.

- f(n) = 3n+1000
- **f(n) = O(n)**
  - 증명 : c=4 로 두면, 모든 n >1000에 대하여

빅-오 표기법(Big-Oh notation)

❖ **Big-Oh notation (Big-Oh 표기법)의 표기법 예시 #3[압축표현]**

Q: f(n) = O(3n+1000) 도 맞나요?

A: 정의에 의하면 맞음. 다만 Big-oh notation 은 반드시 제일 간략한 형태 (모든 계수 (coefficient) 가 1인 함수들 중 가능한 가장 (증가율이) 작은 함수) 로 표현해야 함.

빅-오 표기법(Big-Oh notation)

❖ **Big-Oh notation (Big-Oh 표기법)의 표기법을 위한 몇가지 규칙**

- 함수의 각 term의 **계수는 생략 가능 ex : $13n^2 = O(n^2)$**

- **a > b 인 경우** n^a 와 n^b term 이 같이 있으면 **n^a term 만 남길 수 있음**
  - ✓ (이 경우 n^a **dominates** n^b 라 함) **ex: $f(n) = n^4 + n^3 = O(n^4)$**

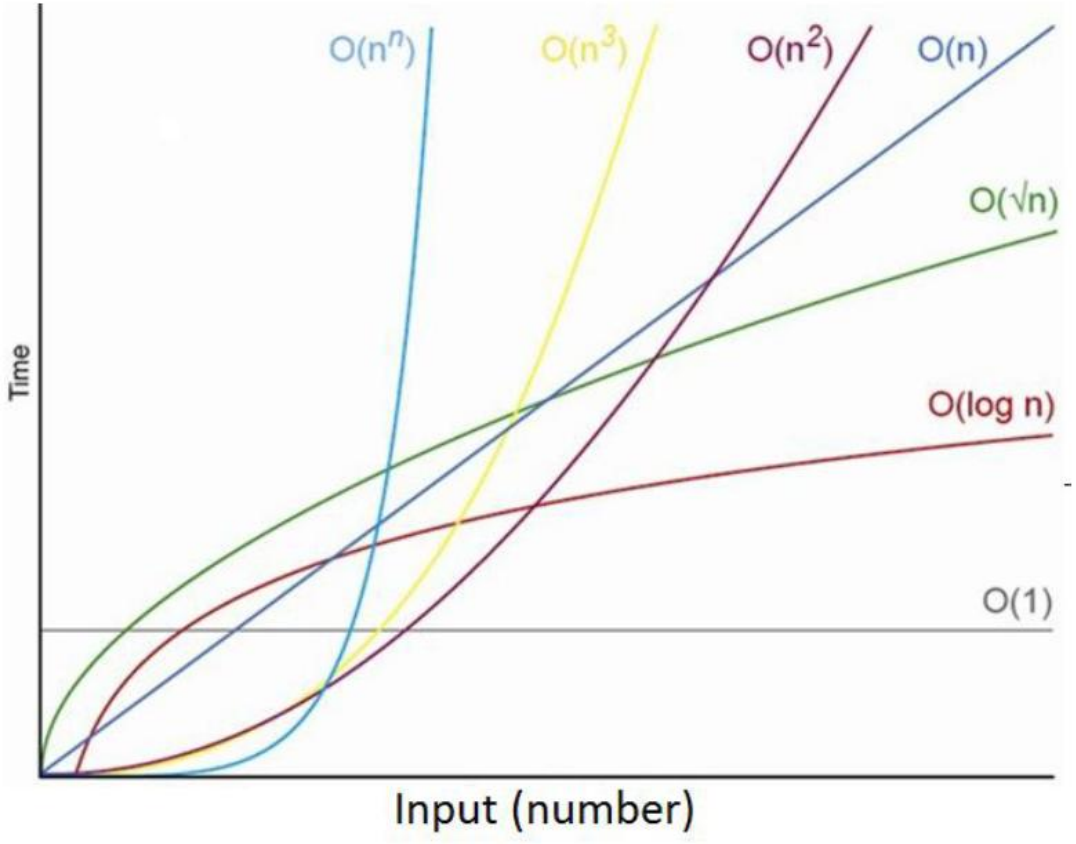
- 어떠한 지수함수(exponential) term도 다항식(polynomial term) 을 **dominate** 함
  - ✓ 지수함수는 다항식보다 큼 **ex: $f(n) = 1.00001^n + n^{100000000} = O(1.00001^n)$**

- 어떠한 다항식도 로그 함수(logarithm) term 을 **dominate** 함.
  - ✓ 다항식은 로그함수보다 큼 **ex: $f(n) = n + (\log n)^{100000000} = O(n)$**

빅-오 표기법(Big-Oh notation)

❖ **Big-Oh notation (Big-Oh 표기법)의 표기법을 위한 규칙 정리**

- 상수< 로그 < 선형< 로그선형 < 제곱< 세제곱 .. < 지수



BETTER

WORSE

• O(1)	constant time
• O(log n)	log time
• O(n)	linear time
• O(n log n)	log linear time
• $O(n^2)$	quadratic time
• $O(n^3)$	cubic time
• $O(2^n)$	exponential time

빅-오 표기법(Big-Oh notation)

❖ **Big-Oh notation (Big-Oh 표기법)의 추가 예시**

1. $f(n) = 3n^2 - 4n + 2$ is $\boldsymbol{O(n^2)}$
2. $f(n) = 100n^3 + 10n\log n + 2$ is $\boldsymbol{O(n^3)}$
3. $f(n) = \log n + 2\log(\log n) - 3$ is $\boldsymbol{O(log\, n)}$
4. $f(n) = 2^{n+2}$ is $\boldsymbol{O(2^n)}$
5. $f(n) = 6 \cdot 2^n + n^2$ is $\boldsymbol{O}(2^n)$

두 알고리즘을 이론적으로 비교하는 방법

1. 두 알고리즘의 기본 연산의 수를 입력 크기에 대한 함수로 표현.

2. 1번의 함수들을 **Big-Oh notation** 으로 표현.

3. Big-Oh notation 안의 두 함수를 비교하여**, 더 느리게 증가하는 (작은) 함수로 표현 가능한 알고리즘이 이론적으로 더 좋은 알고리즘.**

두 알고리즘을 이론적으로 비교하는 방법

❖ Example

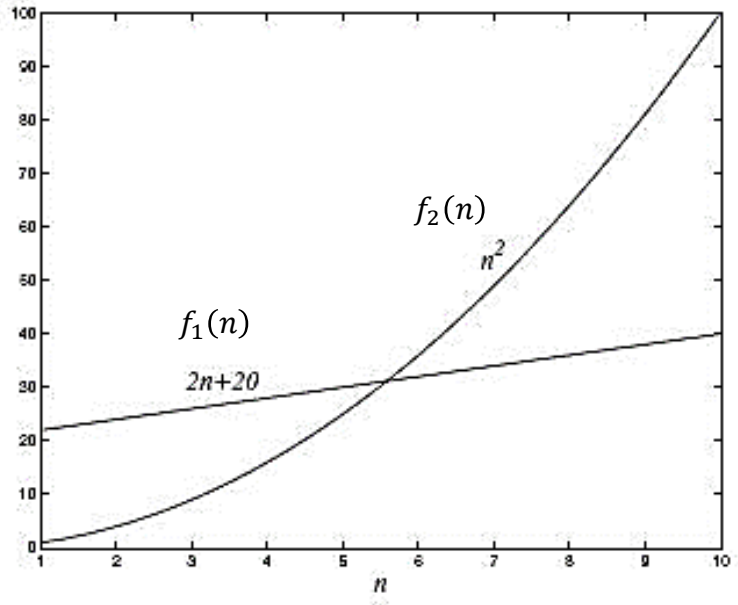
Algorithm 1: $f_1(n) = 2n + 20$

Algorithm 2: $f_2(n) = n^2$

$f_1(n) = 2n + 20 \rightarrow O(n)$

$f_2(n) = n^2 \rightarrow O(n^2)$

- n이 매우 작은 특수한 경우에는 f_2 시간이 걸리는 알고리즘이 f_1 시간이 걸리는 알고리즘보다 더 우수하지만, 일반적으로는 후자가 전자보다 이론적으로 더 우수한 알고리즘.



다른 점근적 분석 방법들

❖ **Theta notation (세타 표기법)**

- $\Theta(f(n))$은 점근적 증가율이 $f(n)$**과 일치하는 모든 함수의 집합**
  - $5n^2+4n = \Theta(n^2)$
  - $5n^2+1,\ n^2,\ 7n^3,\ 7n = \Theta(n^2)$
- $\Theta(f(n))$은 최고차항의 차수가 $f(n)$**과 일치하는 함수의 집합**

❖ **Omega notation (오메가 표기법)**

- $\Omega(f(n))$은 점근적 증가율이 **적어도** $f(n)$ **이 되는 모든 함수의 집합**
  - $5n^2+4n = \Omega(n^2)$
  - $5n^2+1,\ n^2,\ 7n^3,\ 7n = \Omega(n^2)$
- $\Omega(f(n))$은 최고차항의 차수가 $f(n)$ **과 일치하거나 더 큰 함수의 집합**

다른 점근적 분석 방법들

❖ **Theta notation (세타 표기법), Omega notation (오메가 표기법)**

1. $f(n) = 3n^2 - 4n + 2$, Theta: , Omega:

2. $f(n) = 100n^3 + 10n \log n + 2$, Theta: , Omega:

3. $f(n) = \log n + 2 \log(\log n) - 3$, Theta: , Omega:

4. $f(n) = 2^{n+2}$, Theta: , Omega:

5. $f(n) = n!$, Theta: , Omega:

다른 점근적 분석 방법들

O(Big-Oh) vs Ω (Omega) vs Θ (Theta)

그럼 셋 중에 어떤 방법을 써서 알고리즘을 분석하지?

알고리즘 분석에는 O가 가장 많이 쓰임

대부분의 관심은 가장 최악일 경우에 어떻게 되는지가 중요

최선의 시간보다는 최악의 시간이 알고리즘의 성능을 평가하기에 더 적합

Summary

- 알고리즘의 요소
  - 문제, 매개변수, 인스턴스
- 알고리즘의 성능 분석
  - 실험적 분석, 이론적 분석
  - 대표적인 점근적 표현 방법(Big-Oh / Omega / Theta notation)
  - 알고리즘 평가는 최악의 경우를 완화하는 것이 중요! -> Big-Oh notation 이 쓰이는 이유!
- 알고리즘 비교 방법
  - 입력값 (n)에 따른 알고리즘의 연산 횟수 함수
  - 함수의 점근적 표현 O($f(n)$) / Ω ($f(n)$) / Θ ($f(n)$)
  - 점근적 표현값 비교(작은 값이 좋은 알고리즘)

이해를 위한 추가 예시 #1

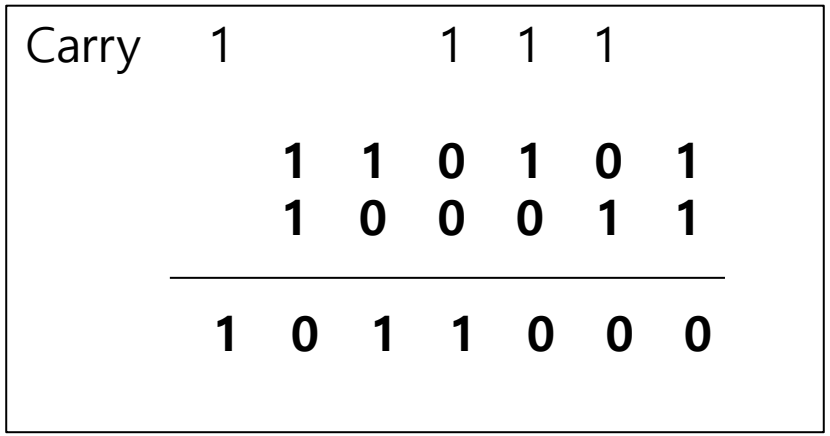
❖ 문제.

- $2n + 20$, n^2 , $100n^3 + 10n\log n + 2$, $3n^2 - 4n + 2$, 2^{n+2}, $\log n + 2\log(\log n) - 3$
- 위 함수들이 포함될 수 있는 집합을 찾아 해당 칸에 모두 작성 하세요.
  - O (n^2) :
  - Ω (2^{n+2}) :
  - Θ ($\log n$) :
- $f_1(n) = 2n+20$, $f_1(n) = n^2$

- 다음 식이 옳은 지 판단하고, 그 이유를 설명 하세요.
- $f_1(n) = O(f_2(n))$
  - 답변:

이해를 위한 추가 예시 #2

- 문제.
  - 단일 bit 에 대해 기본적인 bit 연산에 O(1) 시간이 소모된다 하자.
  - Problem 1: 두 n-bit의 이진수 덧셈



  - 문제의 시간 복잡도는?

이해를 위한 추가 예시 #3

- 문제.
  - 단일 bit 에 대해 기본적인 bit 연산에 O(1) 시간이 소모된다 하자.
  - Problem 1: 두 n-bit 이진수 곱

```
                    1  1  0  1     x
                    1  0  1  1     y
x
-------------------------------------
                 1  1  0  1
              1  1  0  1        (왼쪽으로 한번 Shift)
           0  0  0  0           (왼쪽으로 두번 Shift)
        1  1  0  1              (왼쪽으로 세번 Shift)
-------------------------------------
     1  0  0  0  1  1  1  1
```

  - 문제의 시간 복잡도는?

Questions?

SEE YOU NEXT TIME!