3. Algorithm Analysis

2024학년도 가을학기

정보컴퓨터공학부 황원주 교수

부산대학교
PUSAN NATIONAL UNIVERSITY

강의내용

- 자료구조
- 시간복잡도
  - 자료구조와 알고리즘의 성능분석
  - 알고리즘의 시간복잡도
  - Big-Oh 표기법
  - 자주 사용되는 시간복잡도

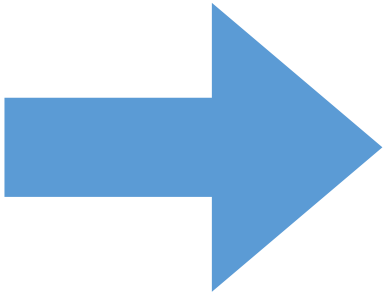
자료구조

알고리즘 (Algorithm)

- 알고리즘은 문제의 **입력(input)**을 수학적이고 논리적으로 정의된 **연산**과정을 거쳐 원하는 **출력(output)**을 계산하는 절차
- 이 절차를 Python과 같은 언어로 표현한 것이 프로그램(program) 또는 코드(code)
  - 입력은 배열(list @Python), 연결리스트(linked list), 트리(tree), 해시테이블(hash table), 그래프(graph)와 같은 자료의 접근과 수정이 빠른 **자료구조**에 저장됨
  - 자료구조에 저장된 입력 값을 기본적인 연산(primitive operation)을 차례로 적용하여 원하는 출력을 계산

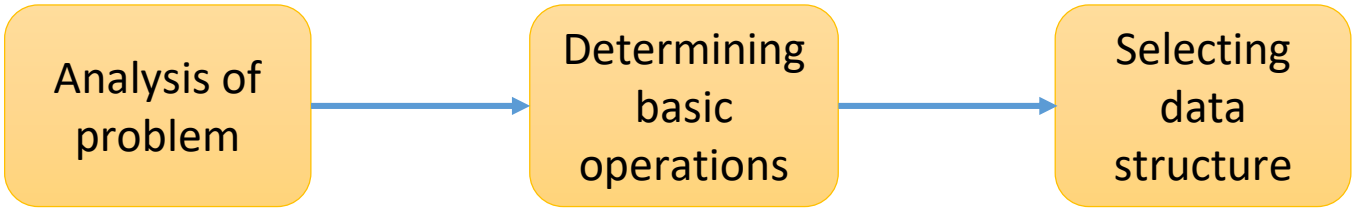
자료구조 (Data Structure)

- 컴퓨터에서 처리할 **데이터를 효율적으로 관리하고 구조화**시키기 위한 방법
- 프로그램에서 저장하는 데이터에 대해 **읽기, 쓰기, 삽입, 삭제, 탐색** 등의 연산을 효율적으로 수행하기 위해서 데이터를 구조화한다.



자료구조 선택

- 자료구조를 선택할 때 어떤 과정을 가질까?
  - 주어진 문제를 분석하여 솔루션이 만족해야 하는 자원(resource)의 제약조건(수행 시간→time complexity, 사용 메모리 양→space complexity)을 확인한다.
  - 솔루션이 사용해야하는 기본 연산을 결정하고, 각 기본 연산들이 자원을 얼마나 필요로 하는지 계산한다.
  - 이러한 제약조건을 가장 잘 충족하는 자료구조를 선택한다



시간복잡도

자료구조와 알고리즘의 성능분석

- 자료구조와 알고리즘의 성능(performance): 수행시간을 나타내는 **시간복잡도(time complexity)**와 알고리즘이 수행되는 동안 사용되는 메모리 공간의 크기를 나타내는 **공간복잡도(space complexity)**에 기반하여 분석
- 주어진 문제를 해결하기 위한 대부분의 알고리즘들이 비슷한 크기의 메모리 공간을 사용하므로 대부분의 경우 **시간복잡도**만을 사용하여 알고리즘의 성능을 분석
- 이를 위해, 실제 코드(C, Java, Python 등)로 구현하여 실제 컴퓨터에서 실행한 후, 수행시간을 측정할 수도 있지만, HW/SW 환경을 하나로 통일해야 하는 어려움이 있다
- 따라서, **가상언어**로 작성된 **가상코드**를 **가상컴퓨터**에서 시뮬레이션하여 HW/SW에 독립적인 계산 환경(computational model)에서 측정해야 한다

가상컴퓨터 (virtual machine)

- 현대 컴퓨터 구조는 Turing machine에 기초한 von Neumann 구조를 따른다
- 현재 가장 많이 사용하는 가상컴퓨터 모델은 (real) RAM (Random Access Machine) 모델이다
- (real) RAM 모델은 CPU + 메모리 + **기본연산**으로 정의된다
  - 연산(operation, command)을 수행하는 CPU
  - 임의의 크기의 실수도 저장할 수 있는 무한한 개수의 레지스터(register)로 구성된 메모리
  - 단위 시간에 수행할 수 있는 **기본연산(primitive operation)**의 집합
    - A = B (대입 또는 복사 연산: B의 값을 A 레지스터에 복사)
    - 산술연산: +, − , *, / (나머지 % 연산은 허용 안되나, 본 강의에서는 포함한다)
    - 비교연산: >, >=, <=, ==, !=
    - 논리연산: AND, OR, NOT
    - 비트연산: bit−AND, bit−OR, bit−NOT, bit−XOR, <<, >>

등교 시간 분석

- 집을 나와서 지하철역까지는 5분, 지하철을 타면 학교까지 30분, 강의실까지는 걸어서 10분 걸린다
- **최선경우분석**(best-case analysis): 집을 나와서 5분 후 지하철역에 도착하고, 운이 좋게 바로 열차를 탄 경우를 의미한다. 따라서 최선경우 시간은 5 + 20 + 10 = 35분
- **최악경우분석**(worst-case analysis): 열차에 승차하려는 순간, 열차의 문이 닫혀서 다음 열차를 기다려야 하고 다음 열차가 10분 후에 도착한다면, 최악경우는 5 + 10 + 20 + 10 = 45분
- **평균경우분석**(average-case analysis): 대략 최악과 최선의 중간이라고 가정했을 때, 40분이 된다.

알고리즘의 시간복잡도

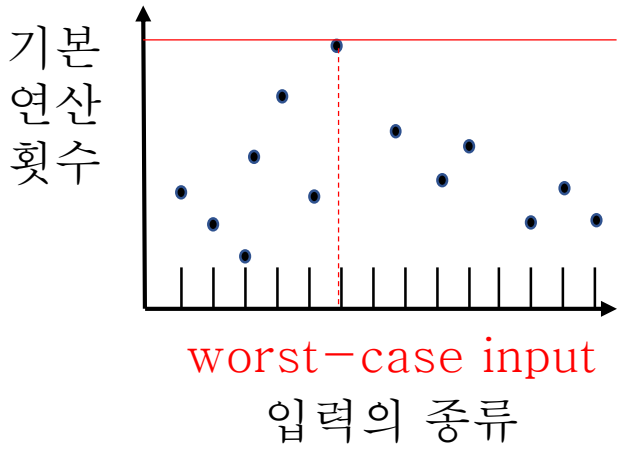
- 가상컴퓨터에서 가상언어로 작성된 가상코드를 실행(시뮬레이션)한다고 가정한다
- 특정 입력에 대해 수행되는 알고리즘의 기본연산의 횟수로 **수행시간**을 정의한다
- 문제는 **입력의 종류가 무한**하므로 모든 입력에 대해 수행시간을 측정하여 평균을 구하는 것은 **현실적으로 가능하지 않다**는 점이다
- 따라서 **최악의 경우의 입력**(worst-case input)을 가정하여, 최악의 경우의 입력에 대한 알고리즘의 수행시간을 측정한다

> 알고리즘의 수행시간 = 최악의 경우의 입력에 대한 기본연산의 수행 횟수

- 최악의 경우의 수행시간은 입력의 크기 n에 대한 함수 T(n)으로 표기된다
- T(n)의 수행시간을 갖는 알고리즘은 어떠한 입력에 대해서도 T(n) 시간 이내에 종료됨을 보장한다

예 : n개의 정수 중 최대값

```
algorithm arrayMax(A, n)
        input: n개의 정수를 저장한 배열 A
        output: A의 수 중에서 최대값
        currentMax = A[0]
        for i = 1 to n-1 do
                if currentMax < A[i]
                        currentMax = A[i]
        return currentMax
```



- A=[3, −1, 9, 2, 12], n=5 인 경우
  → 기본연산 횟수: 7회 (for문 내부의 기본연산은 무시)
- 입력의 종류의 조합과 입력의 개수가 무한
  : 최악의 경우의 입력을 고려

예 : n개의 정수 중 최대값

```
algorithm arrayMax(A, n)
        input: n개의 정수를 저장한 배열 A
        output: A의 수 중에서 최대값
        currentMax = A[0]
        for i = 1 to n-1 do
                if currentMax < A[i]
                        currentMax = A[i]
        return currentMax
```

- if 문의 결과에 따라 currentMax = A[i]의 실행 여부가 결정
- 최악의 경우의 입력은 무조건 currentMax = A[i]을 실행해야 하므로 if 문을 계속 참(true)이 되도록 해야 함. 이 같은 입력은 A의 저장된 값이 오름차순으로 정렬된 경우이다. (A=[2, 5, 8, 12, 32], n=5 인 경우)
- 즉, **오름차순**으로 정렬된 n개의 값이 저장된 배열 A가 **최악의 경우의 입력**임

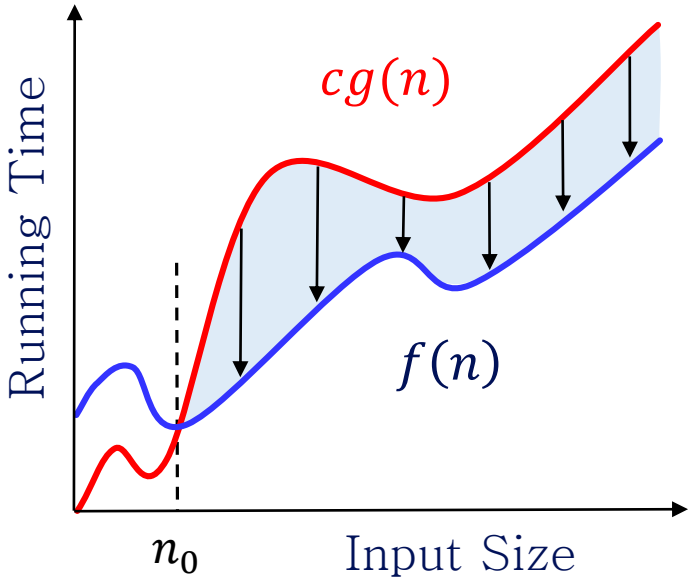
- 최악의 입력에 대한 횟수 분석: $T(n) = 2n - 1$
  - n = 10이면 T(10) = 19가 되어 19번 이내의 기본연산을 수행
  - n = 200이면, T(200) = 399번 이내의 기본연산을 수행한다는 의미

Big-Oh 표기법

- 최악의 입력에 대한 기본연산의 횟수를 정확히 세는 건 일반적으로 귀찮고 까다롭다
- 정확한 횟수보다는 입력의 크기 n이 커질 때, **수행시간의 증가하는 정도(rate of the growth of T(n) as n goes big)**가 훨씬 중요하다
- 수행시간 함수 T(n)이 n에 관한 여러 항(term)의 합으로 표현된다면, 함수 값의 증가율이 가장 큰 항 하나로 간략히 표기하는 게 시간 분석을 간단하게 하는 데 큰 도움이 된다
  - 예를 들어, $T(n) = 2n + 5$이면, 상수항보다는 n의 일차항이 T(n)의 값을 결정하게 되므로 상수항을 생략해도 큰 문제가 없다
  - $T(n) = 3n^2 + 12n - 6$이면, n 값이 커짐에 따라 n^2 항이 T(n)의 값을 결정하게 되므로, 일차항과 상수항을 생략해도 큰 문제가 없다
- 이렇게 최고차 항(가장 빨리 증가하는 항)만을 남기고 나머지는 생략하는 식으로 수행시간을 간략히 표기하는 방법을 점근표기법(asymptotic notation)이라고 부르고, Big-Oh(대문자 O)를 이용하여 다음의 예처럼 표기한다
  - $T(n) = 2n + 5 \rightarrow T(n) = O(n)$
  - $T(n) = 3n^2 + 12n - 6 \rightarrow T(n) = O(n^2)$

Big-Oh 표기법의 정의

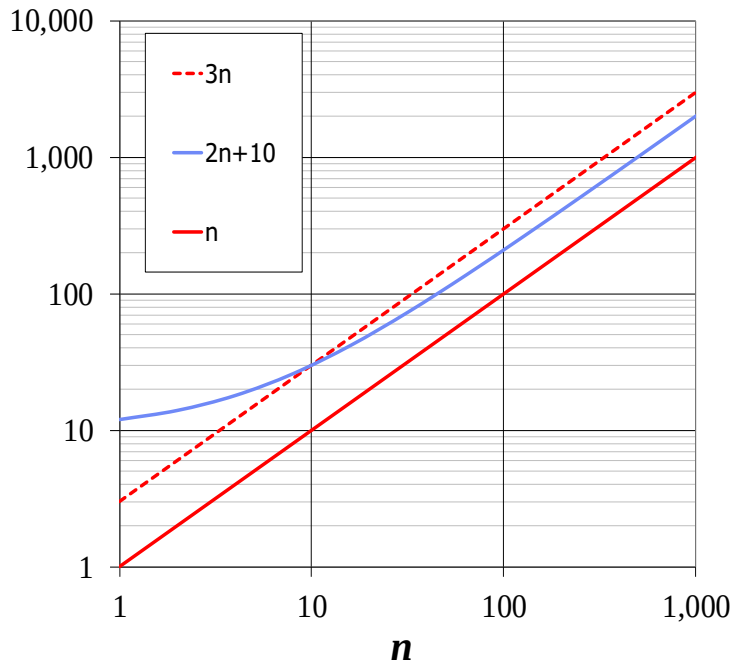
- 모든 $n \geq n_0$에 대해서 $f(n) \leq cg(n)$이 성립하는 양의 상수 c와 n_0 이 존재하면, $f(n) = O(g(n))$이다.
  - Big-Oh 표기법의 의미 : n_0 과 같거나 큰 모든 n (즉, n_0 이후의 모든 n) 대해서 $f(n)$ 이 $cg(n)$ 보다 크지 않다는 것
  - $f(n) = O(g(n))$ 은 n_0 보다 큰 모든 n 대해서 $f(n)$이 양의 상수를 곱한 $g(n)$에 미치지 못한다는 뜻
  - $g(n)$을 $f(n)$의 **상한(Upper Bound)**이라고 한다

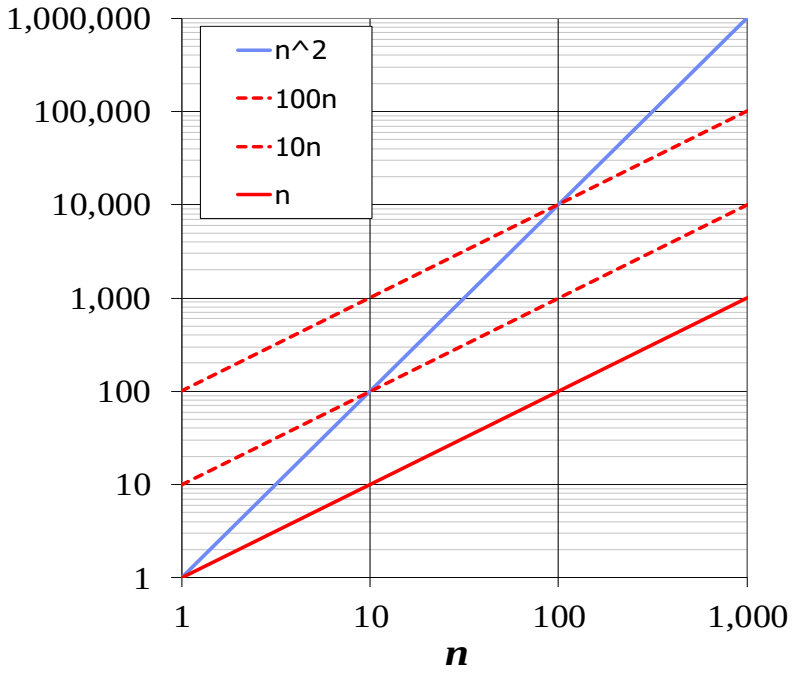


$f(n) = O(g(n))$

Big-Oh 표기법의 예

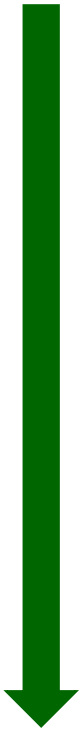
- 예: $T(n) = 2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\ n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$
- 예: $T(n) = n^2$ is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since c must be a constant





자주 사용되는 시간복잡도

- O(1) 상수시간(constant time)
- O(logN) 로그(대수)시간(logarithmic time)
- O(N) 선형시간(linear time)
- O(NlogN) 로그선형시간(N-log-N time)
- $O(N^2)$ 제곱시간(quadratic time)
- $O(N^3)$ 세제곱시간(cubic time)
- $O(2^N)$ 지수시간(exponential time)



시간복잡도 증가

예: python으로 기술한 코드의 수행시간

- O(1) 시간 알고리즘(constant time algorithm): 값을 1 증가시킨 후 리턴

```
def increment_one(a):
    return a+1
```

- O(log n) 시간 알고리즘(logarithmic time algorithm); log의 밑은 2: 십진수 n을 이진수로 표현할 때 필요한 비트 개수 계산 알고리즘

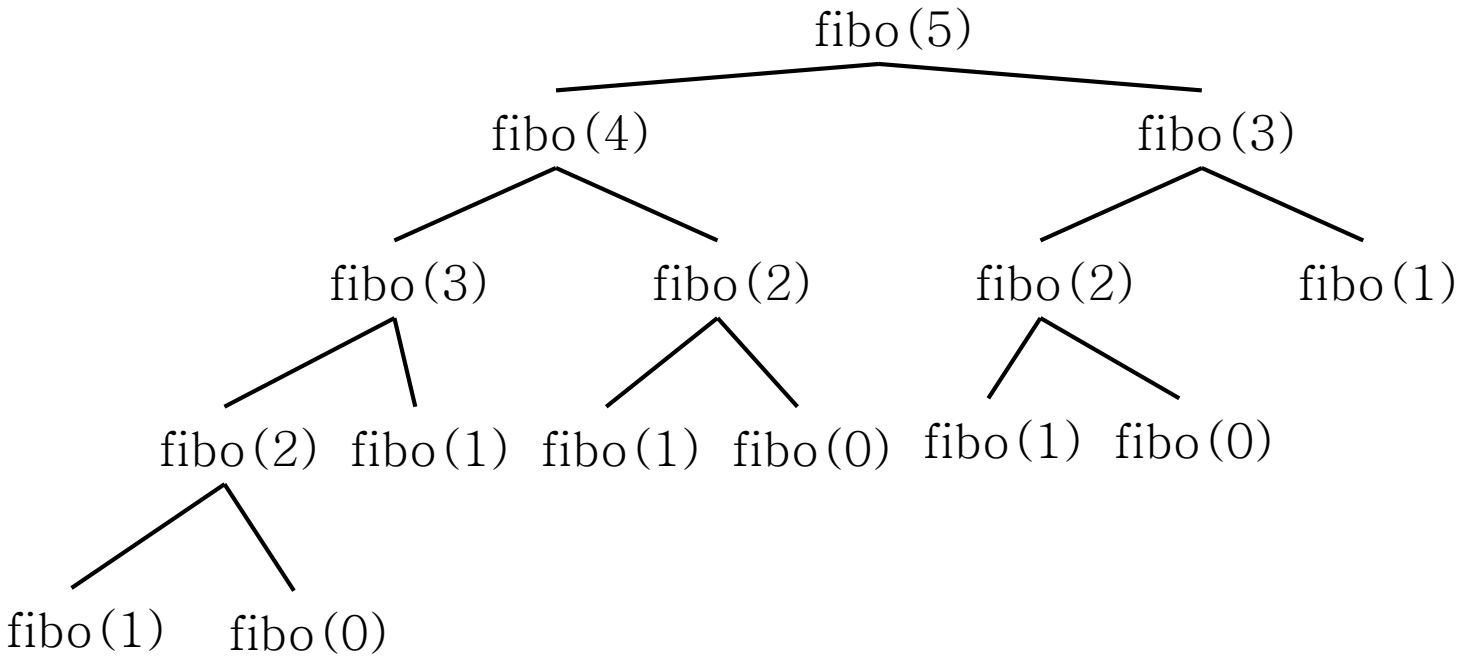
```
def number_of_bits(n):
    count = 0
    while n > 0:
        n = n // 2
        count += 1
    return count
```

- O(n) 시간 알고리즘(linear time algorithm): n개의 수 중에서 최대값 찾는 알고리즘
- $O(n^2)$ 시간 알고리즘(quadratic time algorithm): 두 배열 A, B의 모든 정수 쌍의 곱의 합을 계산하는 알고리즘

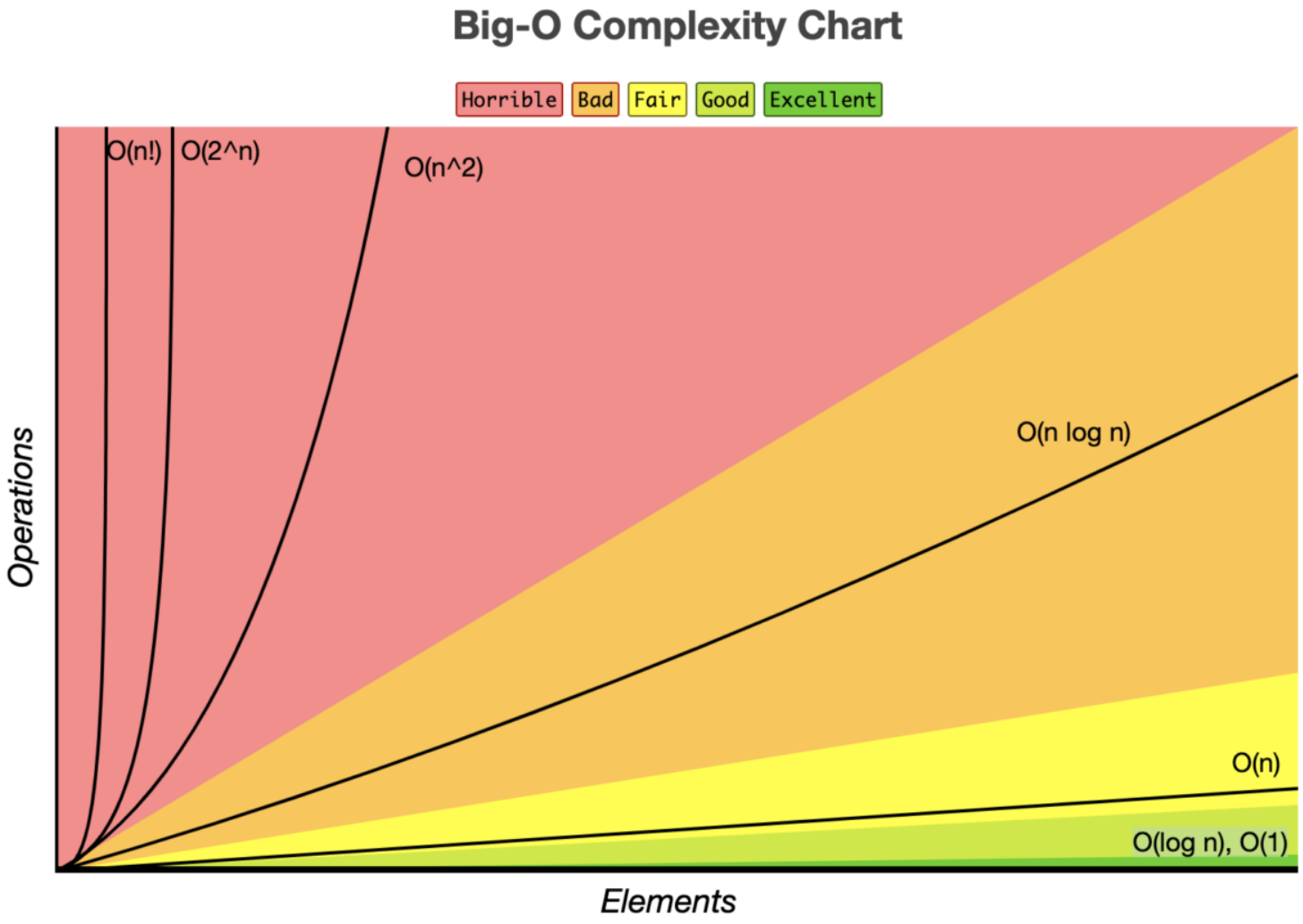
```
algorithm array_sum(A, B, n)
    sum = 0
    for i = 0 to n - 1 do
                for j = 0 to n - 1 do
                            sum += A[i]*B[j]
    return sum
```

- $O(2^n)$ 이상의 시간이 필요한 알고리즘(exponential time algorithm): k번째 피보나치 수 계산하는 재귀 알고리즘

```
def fibonacci(k):
    if k <= 1: return k
    return fibonacci(k-1) + fibonacci(k-2)
```



함수의 증가율 비교


Big-O Complexity Chart
Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

코딩테스트

LeetCode 문제

[비트 조작, Math]

- 136. Single Number
- 231. Power of Two
- 268. Missing Number
- 204. Count Primes

136. Single Number

136. Single Number

Easy | Topics | Companies | Hint

Given a **non-empty** array of integers `nums`, every element appears *twice* except for one. Find that single one.

You must implement a solution with a linear runtime complexity and use only constant extra space.

Example 1:

```
Input: nums = [2,2,1]
Output: 1
```

Example 2:

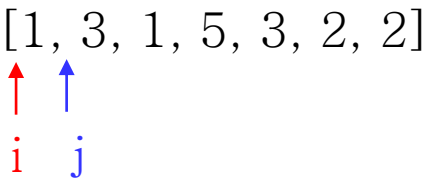
```
Input: nums = [4,1,2,1,2]
Output: 4
```

Example 3:

```
Input: nums = [1]
Output: 1
```

관찰 1: Brute force

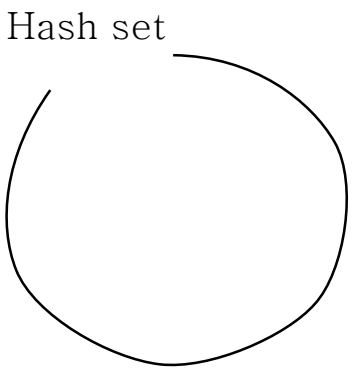
- 포인터 i를 고정하고, 다른 포인터 j를 iteration하면서 값을 비교
- i와 같은 값이 없으면 그 i가 정답



- 시간복잡도: 숫자를 순서대로 하나씩 잡고 배열의 끝까지 iteration하는데 필요한 시간 $O(n^2)$

관찰 2: Hash set 사용

- [1, 3, 1, 5, 3, 2, 2]



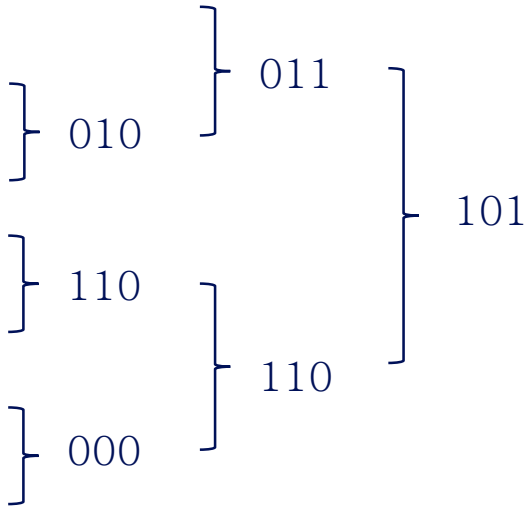
- Hash set 안에
  - 해당 숫자가 없으면 추가
  - 이미 해당 숫자가 있으면 해당 숫자를 삭제
  - 마지막에 hash set 안에는 남는 숫자가 정답
- 시간복잡도: 숫자 배열을 끝까지 iteration하는데 필요한 시간 O(n)
- 공간복잡도: Hash set을 유지하는 데 필요한 공간 O(n)

→ 공간복잡도를 O(n)에서 O(1)로 줄이라고 하면?

관찰 3: 비트 조작

```
>>> bin(~0b0101)
'-0b110'
>>> bin(0b0101 & 0b0011)
'0b1'
>>> bin(0b0101 | 0b0011)
'0b111'
>>> bin(0b0101 ^ 0b0011)
'0b110'
```

- 기본적인 부울 연산(Boolean operator)
  - NOT, AND, OR, XOR
- 비트연산자(bitwise operator)
  - ~, &, |, ^
  - ~0101 → 1010 (1010을 2의 보수로 표현하면 -(0101+1)=-0110)
  - 0101 & 0011 → 0001; 0101 | 0011 → 0111; 0101 ^ 0011 → 0110
- [1, 3, 1, 5, 3, 2, 2]
  - 1: 001
  - 3: 011
  - 1: 001
  - 5: 101
  - 3: 011
  - 2: 010
  - 2: 010



- 시간복잡도: 총 n−1개의 XOR 연산이 필요했기 때문에 O(n)
- 공간복잡도: Hash set을 유지하는 공간이 필요 없으므로 O(1)

231. Power of Two

231. Power of Two

Easy | Topics | Companies

Given an integer `n`, return `true` *if it is a power of two. Otherwise, return* `false`.

An integer `n` is a power of two, if there exists an integer `x` such that $n == 2^x$.

Example 1:

> **Input:** n = 1
> **Output:** true
> **Explanation:** $2^0 = 1$

Example 2:

> **Input:** n = 16
> **Output:** true
> **Explanation:** $2^4 = 16$

Example 3:

> **Input:** n = 3
> **Output:** false

관찰 1: 직관적인 방법

- 주어진 숫자를 2로 나눈 나머지가 1이 나올 때까지 계속해서 2로 나눔.
- 마지막 피젯수(나누어지는 수)가 1이면 true를 1이 아니면 false를 반환

- 16

나머지	몫
0	8
0	4
0	2
0	1
1	

- 10

나머지	몫
0	5
1	

- 8

나머지	몫
0	4
0	2
0	1
1	

- 1

나머지	몫
	1
1	

- 시간복잡도: 주어진 숫자를 절반씩 계속 나눠 가므로 O(log n)

→ 시간복잡도를 O(log n)에서 O(1)로 줄이라고 하면?

관찰 2: 비트 조작

- 비트로 표현하면 2의 제곱이 되는 수들은 중에서 1은 하나만 있음
- (생각하기 어려움) 주어진 숫자에서 1 빼고, AND 연산하면 2의 제곱수는 0이 됨

- 16

1	0	0	0	0
& 0	1	1	1	1
0	0	0	0	0

- 10

0	1	0	1	0
& 0	1	0	0	1
0	1	0	0	0

- 8

0	1	0	0	0
& 0	0	1	1	1
0	0	0	0	0

- 1

0	0	0	0	1

- 7

0	0	1	1	1
& 0	1	1	1	0
0	0	1	1	0

수고 하셨습니다!

부산대학교
PUSAN NATIONAL UNIVERSITY