2024-2
초급 스터디

1. 시간 복잡도, 정렬, 그리디

목차

알고리즘과 문제 해결

- **알고리즘** : 컴퓨터가 따라할 수 있도록 **문제를 해결하는 절차, 방법**을 자세히 명시한 것

알고리즘과 문제 해결

- 알고리즘을 왜 공부해야 할까?

알고리즘과 문제 해결

- 개발, 인공지능, 빅데이터, 보안, …

  → 컴퓨터를 활용하여 현실의 **문제를 해결하는 수단**

- 알고리즘 공부는 정형화된 문제를 통해,
  컴퓨터를 활용한 문제 해결 연습을 하는데 좋다고 생각합니다.

- 현실적인 이유라면, **코딩테스트를 준비**하는데 필요합니다.

시간 복잡도

- 알고리즘 문제를 풀 때,
  컴퓨터는 **1초에 1억 번의 연산**을 할 수 있다고 가정한다.
- 따라서 알고리즘의 실행 시간을 평가할 때,
  최대 몇 번의 연산을 해야 하는지 가늠한다.

시간 복잡도

- 우리가 고안한 알고리즘이 답을 구하기까지 **최대 몇 번의 연산**을 하는지 예측할 때 **'시간 복잡도'**라는 개념을 사용한다.

- 시간 복잡도는 N개의 입력으로부터 원하는 결과를 얻는데 걸리는 **최악**의 연산량을 **O(g(N))** 로 표기한다.
  (이를 '빅-O 표기법' 이라고 합니다. 영어로는 'Big-O notation')

Big-O 표기법

- n이라는 입력이 주어졌을 때, 이를 활용하여 어떤 문제를 푸는데 필요한 코드 실행 횟수를 **f(n)** 이라고 하자.

- 이때 어떤 특정 n_1 이후의 모든 n에 대해, (즉, $n_1 \leq n$)
  $f(n) \leq c*g(n)$ 을 만족하도록 하는 **(n_1, c)** 쌍이 존재하면
  (c는 양의 상수)
- 이때의 g(n)에 대해 **O(g(n))** 이라고 쓸 수 있다.

이게 무슨 말이죠..?

- n개 데이터로 구성된 문제를 푸는 연산량이 **f(n)** 일 때,
  일반적으로 n이 증가하면 연산량 **f(n)**도 같이 증가한다.

- 이때 어떤 시점 이후로는 n이 아무리 증가해도
  f(n) 이 c*g(n) 은 안 넘는다고 표현할 수 있다면
  f(n)은 O(g(n))의 시간 복잡도를 갖는다고 말한다.

시간 복잡도 계산 예시

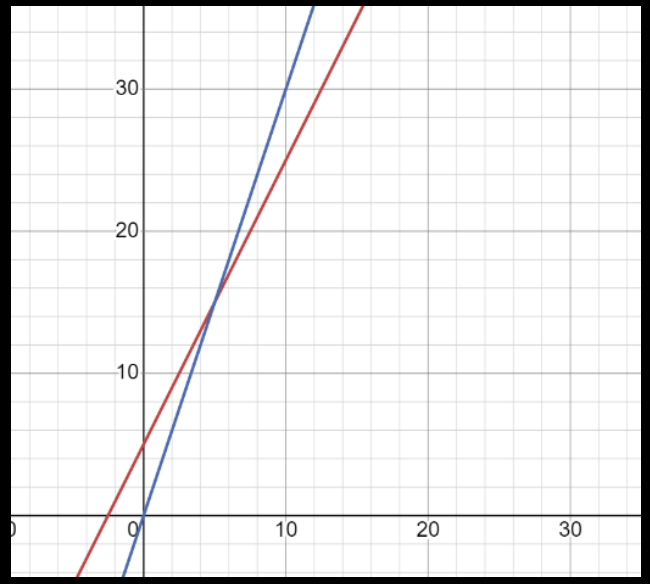
- n이라는 입력이 주어진 어떤 문제를 풀고자 한다.
  이때 필요한 연산량이 2n + 5 라고 하자.

```
N = int(input())
for i in range(2*N+5):
    print(i)
```

예를 들면, 위 코드에서 **print(i)** 의 실행 횟수는 2N + 5 이다.

시간 복잡도 계산 예시

- 실행 횟수 (2n+5)는 n이 증가함에 따라
  어느 순간 (n = 5)부터는 3n보다 항상 작거나 같다.

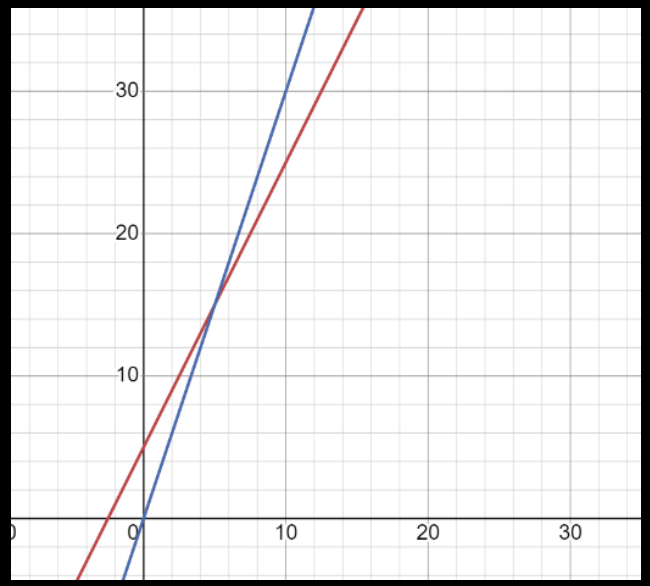


$2n + 5$

$3n$

시간 복잡도 계산 예시

- 따라서 $n_1 = 5$ 일 때, $n_1 \leq n$ 인 모든 n에 대해 $f(n) = 2n + 5 \leq 3n$ 을 만족한다.



$2n + 5$

$3n$

시간 복잡도 계산 예시

어떤 특정 n_1 이후의 모든 n에 대해, ($n_1 \leq n$)
$f(n) \leq c*g(n)$ 을 만족하는 (n_1, c) 쌍이 존재한다!

- $n_1 = 5$
- $c = 3$
- 이때의 $g(n) = n$ → $O(g(n)) = O(n)$

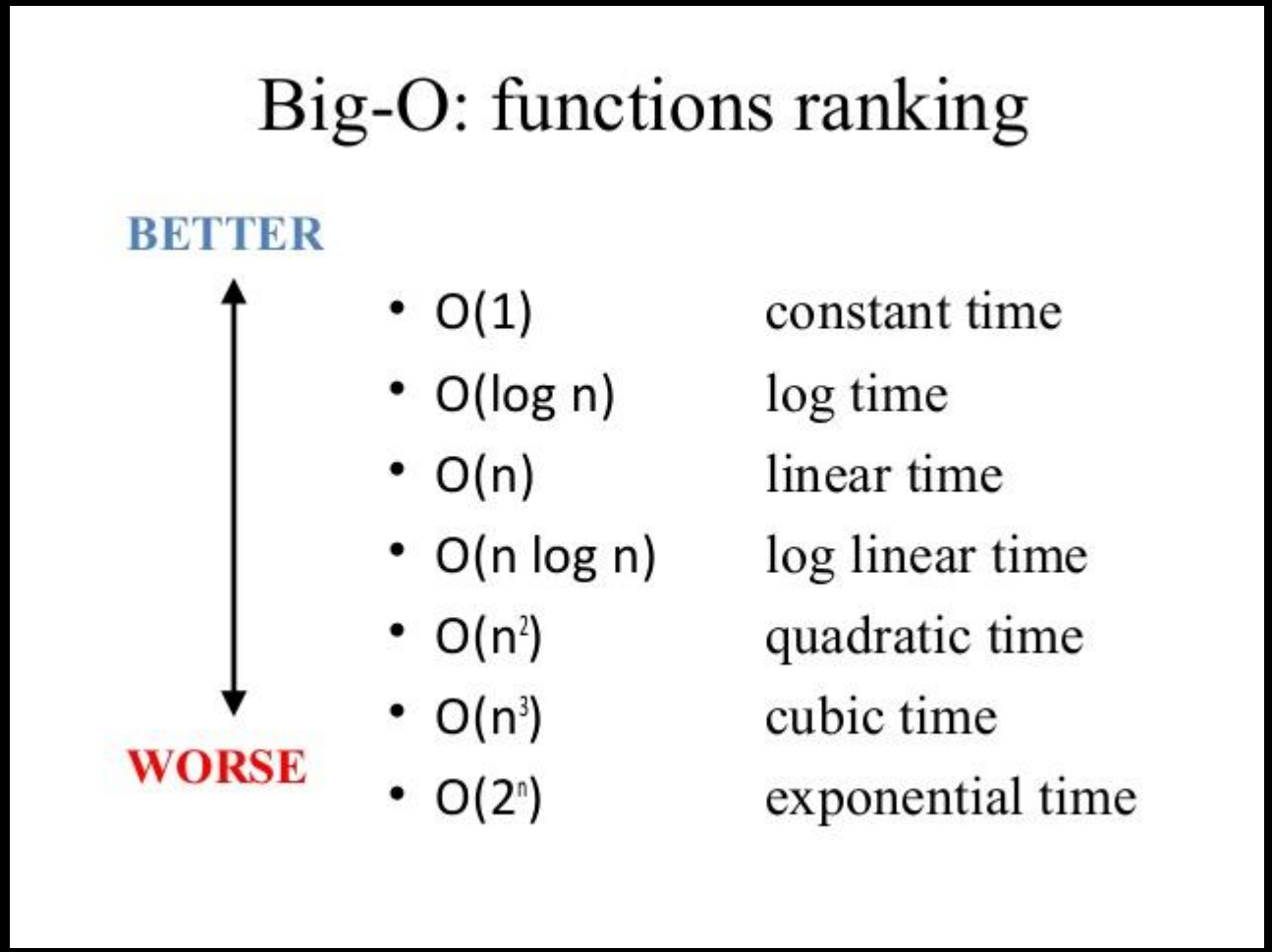
시간 복잡도 계산 예시

- 2n + 5 의 연산량을 갖는 프로그램의 시간복잡도는 O(n) 으로 표현할 수 있다.

시간 복잡도 예시

- 쉽게 정리하면
  어떤 프로그램의 실행 횟수를 입력 데이터 크기 n에 대해서 표현했을 때, 그 식의 **최고차항**을 g(n)으로 보면 됩니다.
- $2n + 50000$ → $O(n)$
- $2n^2 + n + 100$ → $O(n^2)$
- $2 * \log n + 10$ → $O(\log n)$

시간 복잡도 예시


Big-O: functions ranking
BETTER
WORSE
• O(1) constant time
• O(log n) log time
• O(n) linear time
• O(n log n) log linear time
• O(n²) quadratic time
• O(n³) cubic time
• O(2ⁿ) exponential time

시간 복잡도 예시

만약 N개의 데이터가 주어졌을 때,
원하는 결과를 얻기까지 필요한 연산의 횟수가

- N값에 상관없이 **몇 번**만으로 된다 → O(1)
- **N에 비례**한다. → O(N)
- **N^2에 비례**한다. → $O(N^2)$

시간 복잡도 - 연산량 계산

- **Big-O 표기법에 가능한 N의 최댓값을 대입**하면
  최대 연산 횟수를 대략적으로 예측할 수 있다.

- Ex) $O(N^2)$ 시간복잡도를 갖는 알고리즘, $N \leq 10000$
  → 최대 10000^2 = **1억 번의 연산** 수행
  → **최대 1초**의 수행 시간

시간 복잡도

- Q. 프로그램 연산량이 n + 1억 이면요?
  n이 최대 1억이면 최대 2억 번 연산하는 건데,
  O(n) 이라서 1억으로 예측하면 안되는 것 아닌가요?

- A. 맞습니다! 하지만 그렇게 상수가 너무 커서
  시간 복잡도로 예측한 연산량을 크게 벗어나는 경우는
  초급 알고리즘 문제에서 거의 없습니다.

정렬

- 주어진 데이터들을 **특정한 기준**에 따라 나열하는 것
- 대표적인 기준 : 오름차순, 내림차순, 알파벳 사전순

정렬 - 파이썬

- sort() 메서드 → 대상 리스트를 정렬
- sorted() 내장 함수 → 대상 리스트를 정렬한 복사본 반환

```
>>> lst = [1, 4, 3, 2, 8]
>>> lst.sort()
>>> lst
[1, 2, 3, 4, 8]
```

```
>>> lst = [1, 4, 3, 2, 8]
>>> sorted(lst)
[1, 2, 3, 4, 8]
>>> lst
[1, 4, 3, 2, 8]
```

정렬

- 파이썬 정렬 알고리즘의 시간 복잡도는 O(N log N)

정렬 - 연습 문제

- https://www.acmicpc.net/problem/2751

수 정렬하기 2 성공

문제

N개의 수가 주어졌을 때, 이를 오름차순으로 정렬하는 프로그램을 작성하시오.

정렬 - 연습 문제

- 수의 개수는 최대 100만
- 각 수의 절댓값은 최대 100만, 중복되는 숫자는 없음.

정렬 - 연습 문제

- 파이썬 내장 정렬 메서드는 O(N log N) 시간 복잡도를 갖는다.
- N = 100만을 대입하면, 6 * 100만 = **600만**
- 따라서 2초에 안에 충분히 실행 가능!

정렬 - 연습 문제

- 직접 풀어봅시다.
- 빠른 입출력을 사용해야 하는 것에 주의!

정렬 - 연습 문제

- 정답 코드

```python
import sys
input = sys.stdin.readline

n = int(input())
lst = [int(input()) for _ in range(n)]
for num in sorted(lst):
    print(num)
```

정렬 - key

- 정렬은 일반적으로 **두 원소를 비교**하여
  두 원소의 **상대적인 위치를 결정하는 과정**의 반복이다.
- 이때 비교의 기준이 되는 값을 **key** 라고 한다.

정렬 - key

- 일반적으로는 리스트의 원소 자체가 **key** 이지만,
  경우에 따라 다른 **key**를 사용하여 정렬하기도 한다.

정렬 - key

- 파이썬 sort()는 정렬에 사용할 key를 **key 함수로 재정의**할 수 있다.

```
def key_function(original_key):  # 기본적으로 사용되는 키는 리스트의 원소 값
    new_key = original_key
    return new_key  # 비교에 사용할 새로운 키를 반환


l = list()
l.sort(key=key_function)
```

정렬 - key

- 정렬에 사용할 **key는 여러 개를 사용할 수 있다.**
- 왼쪽 key부터 사용하여 비교하다가,
  같은 key값이 나오면 다음 정의된 key 값을 사용해서 비교한다.

```
def key_function(original_key):  # 기본적으로 사용되는 키는 리스트의 원소 값
    new_key1 = original_key
    new_key2 = original_key
    return (new_key1, new_key2)  # key1이 동일하면, key2를 사용해서 비교


l = list()
l.sort(key=key_function)
```

다중 조건 정렬

- https://www.acmicpc.net/problem/11651

  좌표 정렬하기 2 성공

- (x, y) 좌표가 주어질 때,
  - **y가 증가**하는 순으로 정렬
  - y가 같다면, **x가 증가**하는 순으로 정렬

다중 조건 정렬

- (x, y) 좌표를 리스트에 담은 경우,
  문제 조건에 맞춰 다음과 같이 key 함수를 작성할 수 있다.

```
def compare_function(리스트_원소):    # 리스트 원소 = (x, y) 튜플
    비교기준으로_사용할_값1 = 리스트_원소[1]  # 첫번째 비교 기준은 y 좌표 값 이므로 (x, y) 에서 y에 접근
    비교기준으로_사용할_값2 = 리스트_원소[0]  # 두번째 비교 기준은 x 좌표 값 이므로 (x, y) 에서 x에 접근
    return (비교기준으로_사용할_값1, 비교기준으로_사용할_값2)
```

다중 조건 정렬

- 11651 정답 코드

```python
import sys
input = sys.stdin.readline

def compare_function(리스트_원소):    # 리스트 원소 = (x, y) 튜플
    비교기준으로_사용할_값1 = 리스트_원소[1]  # 첫번째 비교 기준은 y 좌표 값 이므로 (x, y) 에서 y에 접근
    비교기준으로_사용할_값2 = 리스트_원소[0]  # 두번째 비교 기준은 x 좌표 값 이므로 (x, y) 에서 x에 접근
    return (비교기준으로_사용할_값1, 비교기준으로_사용할_값2)


n = int(input())
l = [tuple(map(int, input().split())) for _ in range(n)]
l.sort(key=compare_function)
for x, y in l:
    print(x, y)
```

다중 조건 정렬

- 람다 함수를 이용하면 더 간단하게 작성할 수 있다.

```
import sys
input = sys.stdin.readline

n = int(input())
l = [tuple(map(int, input().split())) for _ in range(n)]
l.sort(key=lambda x: (x[1], x[0]))
for x, y in l:
    print(x, y)
```

다중 조건 정렬

- 숫자형 데이터의 경우 key 값을 원래 값의 반대 부호로 설정하면 내림차순 정렬을 할 수 있다.
- 1 2 3 4 5 를 정렬할 때,
  -1, -2, -3, -4, -5 를 각각의 key로 오름차순 정렬하도록 시키면
  -5, -4, -3, -2, -1 로 정렬되어, 실제로 5 4 3 2 1이 되기 때문이다.

다중 조건 정렬

- y좌표가 증가하는 순으로, y좌표가 같다면 x좌표가 **감소**하는 순으로 정렬

```python
import sys
input = sys.stdin.readline

n = int(input())
l = [tuple(map(int, input().split())) for _ in range(n)]
l.sort(key=lambda x: (x[1], -x[0]))
for x, y in l:
    print(x, y)
```

```
5
0 4
1 2
1 -1
2 2
3 3
```

```
1 -1
2 2
1 2
3 3
0 4
```

그리디

- **탐욕법**이라고도 부른다.
- 어떤 문제의 최적해를 찾을 때,
  눈 앞의 최적을 탐욕적으로 골라서 전체의 최적해를 찾는 기법
- 모든 경우를 확인하지 않고, 최적의 경우를 바로 답으로 쓰기 때문에 빠른 시간에 문제를 풀 수 있다는 장점이 있다.

그리디 - 예시 1

- 동전 문제 https://www.acmicpc.net/problem/11047
- 금액이 배수 관계인 동전들이 **정렬**되어 주어졌을 때,
  K원을 만드는데 필요한 **최소한의 동전 개수**를 구하는 문제

그리디 - 예시 1

- 직관적으로 생각하면,
  현재 쓸 수 있는 가장 큰 금액의 동전을 반복적으로 사용하면 된다.

- 과연 논리적으로도 타당할까? 한번 증명해보자.

그리디 - 예시 1

- 사용 가능한 가장 큰 금액의 동전이 **A원**이라고 하자.
- K ≥ A 일 때, 최적해에 A원 동전이 사용되지 않았다고 가정해보자.
- 즉, A > B인, B원 금액의 동전들로 K원을 만드는 최적해가 존재한다고 가정해보자.

그리디 - 예시 1

- A는 B의 배수이므로, $A = c * B$ $(c > 1)$ 로 표현할 수 있다.
- $K \geq A = c * B$ 이므로, 최소한 c개의 B원 동전을 사용해야 한다.
- 이때 A원 동전을 c개의 B원 동전 대신 사용하면,
  총 c-1 개의 동전을 더 적게 사용할 수 있으므로 최적이다.
- 따라서 최적해에 A원 동전이 사용되지 않았다는 가정은 모순이므로,
  최적해에는 항상 A원 동전이 포함되어 있어야 한다.

그리디 - 예시 1

- 직접 풀어봅시다.

그리디 - 예시 1

- 정답 코드 - **정렬**이 되어 있기 때문에 **O(n)** 에 풀 수 있다.

```python
n, k = map(int, input().split())
coins = reversed([int(input()) for _ in range(n)])
answer = 0

for coin in coins:
    if coin <= k:
        answer += k // coin
        k %= coin

print(answer)
```

그리디

그리디 알고리즘을 사용하는 문제는 다음과 같은 특징이 있다.

1. **탐욕적 선택 속성**
   현재의 선택이 다음의 선택에 영향을 주지 않는다.
2. **최적 부분 구조**
   전체 문제의 최적해가, 부분 문제의 최적해로 구성된다.

그리디

동전 문제의 경우

1. **탐욕적 선택 속성**
   K원이 남아도, K-A원이 남아도 '가장 큰 금액의 동전을 고르면 된다' 는 선택 기준을 계속 적용할 수 있다.
2. **최적 부분 구조**
   K원에 대한 동전 문제의 최적해는 A원 동전을 n개 사용한 뒤 남은 K-A*n원에 대한 동전 문제의 최적해로 구성된다.

그리디

- 그리디는 문제를 부분 문제로 나눈 뒤
  최적해 선택 기준을 모든 **부분 문제에 일관되게 사용**하여 답을 구하는 방법
- 최적해 선택 기준을 직관적으로 잘 떠올리고,
  이 기준을 일관되게 적용할 수 있음을 증명하는 것이 그리디 풀이의 핵심

그리디

- 최적해 선택 기준은 **'가장 ~ 한 것을 선택'** 하는 경우가 많다.
- 따라서 최적해를 선택하는 과정에서 보통 **정렬**을 함께 활용한다.

그리디 - 예시 2

- 회의실 배정 (https://www.acmicpc.net/problem/1931)
- N개의 회의에 대해 (회의 시작, 끝 시간) 이 주어질 때,
  하나의 회의실에 배정할 수 있는 **회의 개수의 최댓값** 구하기

그리디 - 예시 2

- **N ≤ 100,000** 이므로,
  각각의 회의를 배정하는 모든 경우의 수를 고려할 수 는 없다. (2^10만 가지)

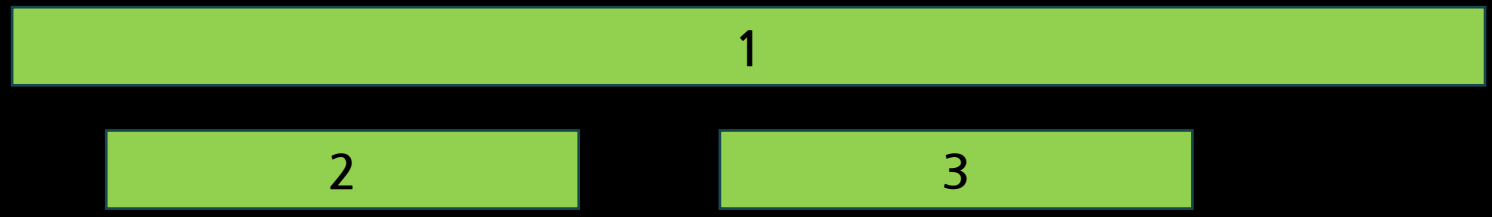
그리디 - 예시 2

- 탐욕적으로 회의를 배치하여 빠르게 답을 구해보자.

- 즉, 일관되게 적용할 **회의를 배치하는 기준**을 정하고,
- **회의를 선택하는 각각의 부분 문제**에 대해 이 기준을 일관되게 적용해보자.

그리디 - 예시 2

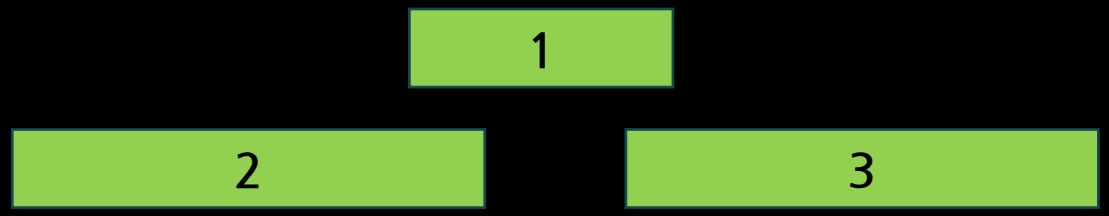
- 회의가 빠르게 시작하는 순서대로 배치



- 1번 회의 1개를 고르는 것보다, (2, 3) 2개 회의를 고르는 것이 최적이다.

그리디 - 예시 2

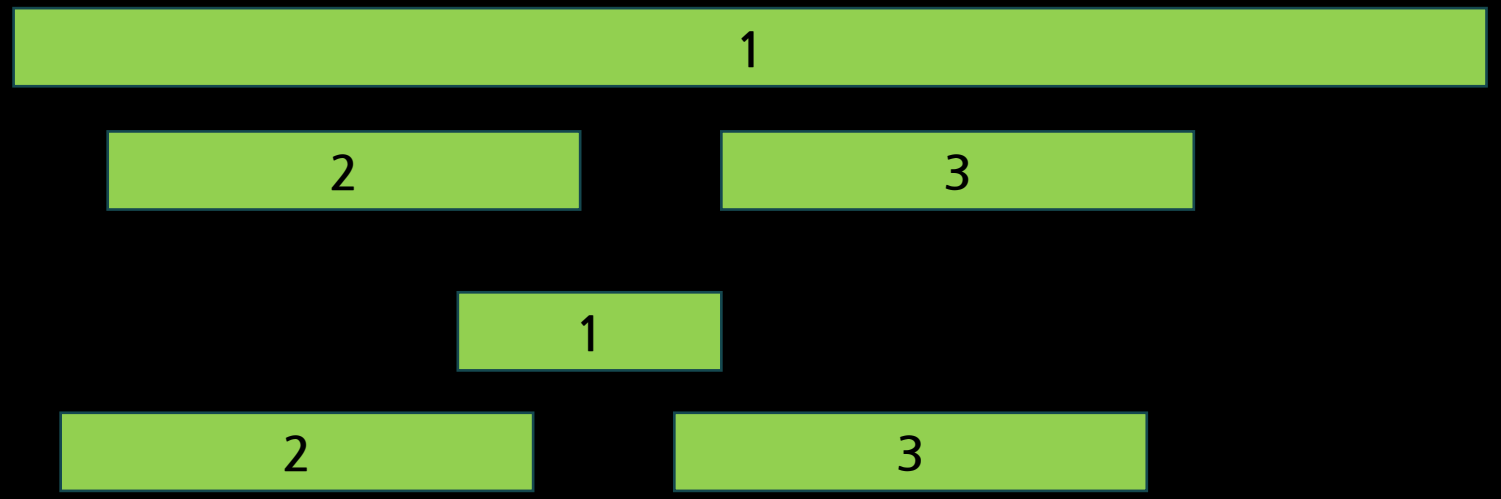
- 회의 시간이 짧은 순으로 배치



- 1번 회의 1개를 고르는 것보다, (2, 3) 2개 회의를 고르는 것이 최적이다.

그리디 - 예시 2

- 회의가 빨리 끝나는 순으로 배치



- 두 경우 모두 문제가 없다.

그리디 - 예시 2

- **회의가 빨리 끝나는 순**으로 배치하는 것이 최적임을 증명해보자.
- 회의가 제일 빠르게 끝나는 회의를 선택하지 않았을 때,
  더 많은 회의를 배정할 수 있다고 가정해보자.

그리디 - 예시 2

- 다음과 같은 상황을 고려해보자.

그리디 - 예시 2

- A를 고른 경우

그리디 - 예시 2

- B를 고른 경우

그리디 - 예시 2

- 가장 빨리 끝나는 회의 A를 선택하지 않았을 때도
  B를 선택하면 같은 최적해를 얻을 수 있다.

그리디 - 예시 2

- 하지만 B를 선택한 상황에서, B 대신 A를 선택하더라도
  최소한 B를 선택했을 때 만큼의 회의는 선택할 수 있다.
- 이를 통해, 지금 최선의 선택을 하지 않은 경우, 최소한 최선의 경우를 선택했을 때보다 더 나은 상황이 되지는 않음을 보일 수 있다.

→ 즉, 매 순간 가장 빨리 끝나는 회의를 선택하는 것이 이득이다.
(탐욕적 선택 속성)

그리디 - 예시 2

- 회의가 빠르게 끝나는 순으로 **정렬**하여 문제를 풀어보자.
  (회의 시작 시간과 끝 시간이 같은 경우를 주의하자!)

그리디 - 예시 2

- 정답 코드

```python
n = int(input())
schedules = sorted([tuple(map(int, input().split())) for _ in range(n)], key=lambda x: (x[1], x[0]))
last_meeting_time = 0
answer = 0
for start_time, end_time in schedules:
    if start_time >= last_meeting_time:
        answer += 1
        last_meeting_time = end_time
print(answer)
```

그리디 - 접근 팁

- 그리디 기법을 사용하려면, 당장의 최적해를 바로 가져가는 것이 올바른 문제 풀이 과정임을 **증명**할 수 있어야 한다.
- 하지만 코딩 테스트, 대회에서 수학 증명을 엄밀히 하기는 힘들다..
- 실전에서는 이 문제가 그리디라는 **강한 믿음**을 갖고 풀자.
- 그러고 나서 틀렸다면 과감하게 그 문제는 넘기자.
  (그리디로 푸는 문제가 아닐 수도 때문이다.)

이번주 연습 문제

정렬

- 2750 (버블소트 풀이 추천)
- 1181
- 1026
- 1427
- 10814
- 10989

그리디

- 14659
- 13305
- 1541
- 11399
- 2812

부록 : 정렬 - stable, in-place

- Stable Sort : 같은 기준 값에 대해 기존의 상대적 위치가 보존되는 정렬

  1 3 2 1 을 정렬한 1 1 2 3 에 대해서,
  1의 상대적 위치가 정렬 후에도 동일함을 보장한다.

- In-place sort : 데이터가 기존에 저장된 공간 안에서만 진행되는 정렬
  정렬 중에 추가적인 메모리 공간이 필요하지 않다.

부록 : 정렬 - stable, in-place

- 참고) 파이썬 정렬은 in-place 방식이며, stable 하다.

```
def sort(self, *args, **kwargs): # real signature unknown
    """
    Sort the list in ascending order and return None.

    The sort is in-place (i.e. the list itself is modified) and stable (i.e. the
    order of two equal elements is maintained).

    If a key function is given, apply it once to each list item and sort them,
    ascending or descending, according to their function values.

    The reverse flag can be set to sort in descending order.
    """
    pass
```

부록 : 정렬 - 구현 알고리즘

- 버블 소트, 삽입 정렬, 병합 정렬, 힙 소트, 퀵 소트 …

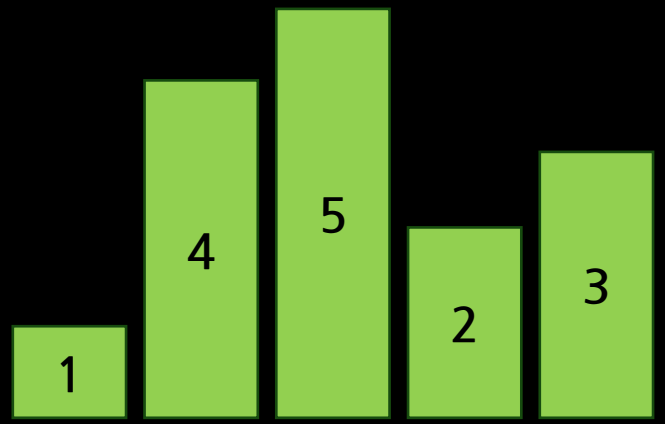
- 시간 복잡도, in-place, stable sort 개념 예시를 확인하는 차원에서 버블 소트 알고리즘만 살펴보자.

버블 소트

- 인접한 원소 2개를 반복적으로 비교하는 정렬 방법
- 각 단계를 거칠 때마다 **제일 큰 값**이 거품이 떠오르듯 정렬된다.

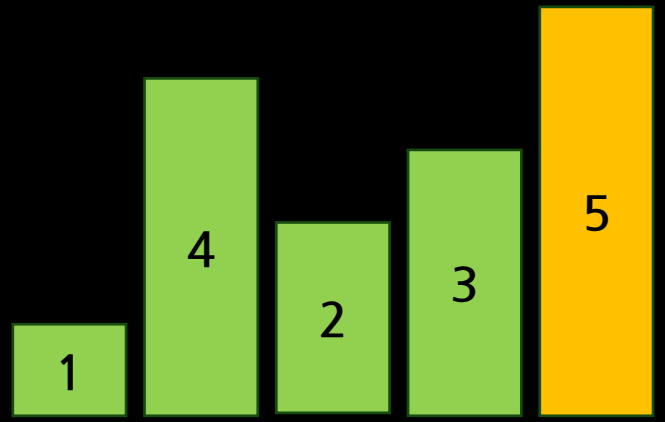
버블 소트

- 제일 왼쪽에서부터 2개씩 비교해서 **왼쪽 > 오른쪽** 이면 swap
- 1단계 - 제일 큰 수인 5 정렬 완료



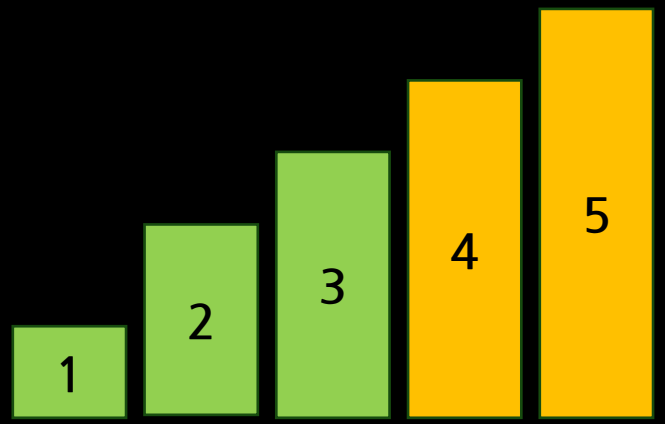
버블 소트

- 제일 왼쪽에서부터 2개씩 비교해서 **왼쪽 > 오른쪽** 이면 swap
- 2단계 - 그 다음으로 큰 수인 4 정렬 완료



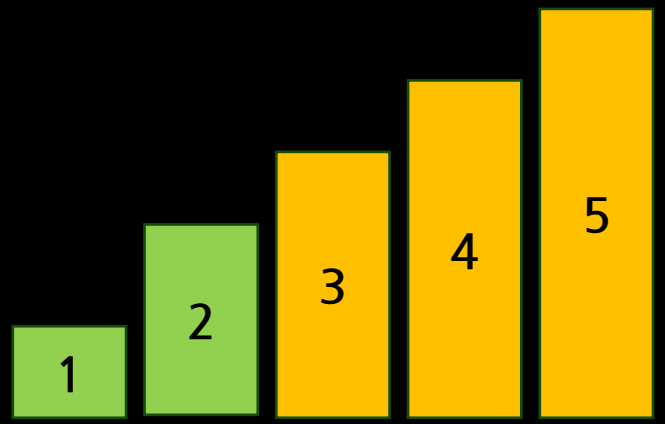
버블 소트

- 제일 왼쪽에서부터 2개씩 비교해서 **왼쪽 > 오른쪽** 이면 swap
- 3단계 - 그 다음으로 큰 수인 3 정렬 완료



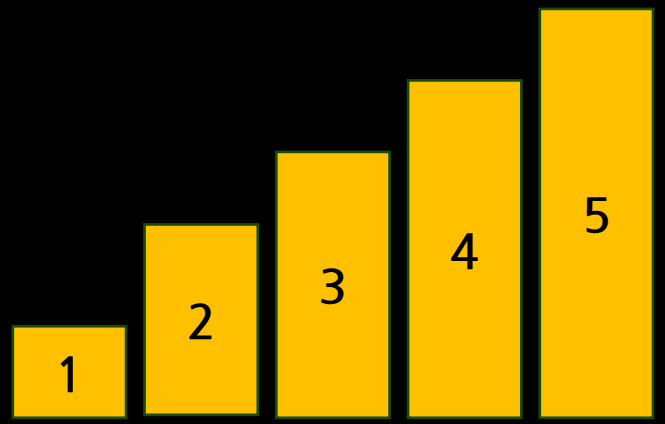
버블 소트

- 제일 왼쪽에서부터 2개씩 비교해서 **왼쪽 > 오른쪽** 이면 swap
- 4단계 - 그 다음으로 큰 수인 2 정렬 완료



버블 소트

- 정렬 완료



버블 소트

버블 소트의 비교 횟수

- 1단계 : 4번
- 2단계 : 3번
- 3단계 : 2번
- 4단계 : 1번

버블 소트

- N개의 원소가 있을 때, i번째 단계에서 N-i 번 비교
- 총 비교 횟수는 1부터 N-1까지 합과 같으므로 **(N-1) * N / 2**
- 최고 차항을 고려하면 시간복잡도는 **$O(N^2)$**

버블 소트

- 구현

```
lst = [1, 4, 5, 2, 3]
for i in range(len(lst)-1, -1, -1):
    for j in range(i):
        if lst[j] > lst[j+1]:
            lst[j], lst[j+1] = lst[j+1], lst[j]
    print(lst)
```

```
[1, 4, 2, 3, 5]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

버블 소트

- N개의 원소 리스트 안에서만 정렬을 진행하므로 in-place 정렬
- 같은 값에 대해서는 swap하지 않으면 상대적 위치가 바뀌지 않으므로 stable 정렬