

# 시간 복잡도

# 주요 내용

- 알고리즘 복잡도 분석
- 시간 복잡도와 "Big-O" 표현식
- 최선, 최악, 평균 시간 복잡도

# 알고리즘 복잡도 분석

- 동일한 문제를 해결하는 프로그램 두 개의 성능을 어떻게 비교할 수 있을까?
- 먼저 알고리즘과 프로그램의 차이점을 이해해야 한다.

## 알고리즘 vs. 프로그램

- 알고리즘
  - 주어진 문제를 해결하기 위한 절차의 단계별 설명서.
  - 주어진 문제의 모든 사례를 해결할 수 있어야 함.
  - 예제: 최대공약수 구하기 문제를 해결하는 알고리즘은 임의의 두 정수의 최대공약수를 계산해야 함.
  - 특정 프로그래밍언어 또는 프로그램 구현 방식과 무관함
  - 주어진 문제를 해결하는 여러 종류의 알고리즘 존재 가능
- (컴퓨터) 프로그램
  - 주어진 문제를 해결하기 위해 **특정** 프로그래밍언어로 작성되어 실행이 가능한 코드
  - 사용하는 프로그래밍언어와 작성자에 따른 여러 종류의 프로그램 존재
  - 프로그램의 핵심은 문제해결을 위한 특정 알고리즘!
  - 동일한 알고리즘을 이용하더라도 다르게 보이는 프로그램 구현 가능

## 문제와 알고리즘

- '두 정수의 최대공약수 구하기' 문제를 해결하는 알고리즘
  - 임의의 두 정수에 대해 동일한 방식으로 최대공약수를 구해야 함
  - 문제의 특정 사례에 의존하지 않아야 함.
- 주어진 문제를 해결하는 여러 알고리즘이 존재 가능

## 문제 하나, 알고리즘 여러 개

- 알고리즘 1: 초등학교에서 배운 방식

12와 20의 공약수  $\rightarrow$  2

6과 10의 공약수  $\rightarrow$  2

↑

$2 \times 2$  는

12와 20의 최대공약수

$$\begin{array}{r} 12 \quad 20 \\ \hline 6 \quad 10 \\ \hline 3 \quad 5 \end{array}$$

- 알고리즘 2: 유클리드 호제법 방식

NEXT

유클리드 호제법  $\text{gcd}(a, b) = \text{gcd}(a - qb, b)$ 을 이용하여  $\text{gcd}(2424869, 9509)$ 를 계산해보기로 하자..

$$\begin{aligned} & \text{gcd}(2424869, 9509) \\ &= \text{gcd}(2424869 - 255 \times 9509, 9509) \\ &= \text{gcd}(74, 9509) \\ &= \text{gcd}(74, 9509 - 128 \times 74) \\ &= \text{gcd}(74, 37) \\ &= \text{gcd}(74 - 2 \times 37, 37) \\ &= \text{gcd}(0, 37) \\ &= 37 \end{aligned}$$

따라서  $\text{gcd}(2424869, 9509) = 37$  임을 구할 수 있다.

# 알고리즘 하나, 프로그램 여러 개

## 프로그램 1

```
In [1]: def sum_of_n(n):
    the_sum = 0
    for i in range(1, n + 1):
        the_sum = the_sum + i

    return the_sum
```

## 프로그램 2

```
In [2]: def foo(tom):
    fred = 0
    for bill in range(1, tom + 1):
        barney = bill
        fred = fred + barney

    return fred
```

## 알고리즘 비교

- 두 프로그램 중에서 어떤 프로그램이 보다 좋은 프로그램인가?
  - 가독성, 명료성 등을 기준으로 보면 <프로그램 1>이 보다 좋음
- 알고리즘 복잡도 분석
  - 프로그램 실행에 필요한 컴퓨팅 자원의 양과 활용의 효율성 측정
  - 이 기준에서 보면 위 두 프로그램은 알고리즘 측면에서 동일함.

## 컴퓨팅 자원

- 공간량: 알고리즘을 구현한 프로그램이 실행 될 때 요구되는 메모리, 저장 공간 등  
의 양
- 실행 시간: 알고리즘을 구현한 프로그램이 특정 결과를 반환할 때까지 걸리는 실행  
시간

## 입력 크기에 따라 실행 시간이 선형적으로 달라지는 알고리즘

실행 시간 측정

In [3]:

```
import time

def sum_of_n_time(n):

    start = time.time()      # 실행 시작
    sum_of_n(n)               # 1부터 n까지의 합 계산
    end = time.time()         # 실행 종료

    return end - start       # 1부터 n까지의 합 계산에 필요한 시간
```

## 1만까지의 합

```
In [4]: n = 10000
m = 10
time_sum = 0

for i in range(m):
    time_sum += sum_of_n_time(n)

print(f"1부터 {n}까지 더하는데 평균적으로 {time_sum/m:7.5f}초 걸림.")
```

1부터 10000까지 더하는데 평균적으로 0.00021초 걸림.

## 10만까지의 합

```
In [5]: n = 100000
m = 10
time_sum = 0

for i in range(m):
    time_sum += sum_of_n_time(n)

print(f"1부터 {n}까지 더하는데 평균적으로 {time_sum/m:7.5f}초 걸림.")
```

1부터 100000까지 더하는데 평균적으로 0.00239초 걸림.

## 100만까지의 합

In [6]:

```
n = 1000000
m = 10
time_sum = 0

for i in range(m):
    time_sum += sum_of_n_time(n)

print(f"1부터 {n}까지 더하는데 평균적으로 {time_sum/m:7.5f}초 걸림.")
```

1부터 1000000까지 더하는데 평균적으로 0.02529초 걸림.

## 1천만까지의 합

In [7]:

```
n = 10000000
m = 10
time_sum = 0

for i in range(m):
    time_sum += sum_of_n_time(n)

print(f"1부터 {n}까지 더하는데 평균적으로 {time_sum/m:7.5f}초 걸림.")
```

1부터 10000000까지 더하는데 평균적으로 0.24920초 걸림.

## 입력 크기에 상관 없이 실행 시간이 일정한 알고리즘

예제: 1 부터 n 까지 합 구하기 (다른 알고리즘)

`sum_of_n_3()` 함수는 1부터 n까지의 합을 계산하기 위해 아래 식을 이용한다.

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

In [8]:

```
def sum_of_n_2(n):  
    sum = (n * (n + 1)) / 2  
    return sum
```

- 실행 시간 측정

```
In [9]: def sum_of_n_2_time(n):
    start = time.time() # 실행 시작
    sum_of_n_2(n)
    end = time.time() # 실행 종료

    return end - start
```

- 실행시간이 입력값에 의존하지 않음

```
In [10]: m = 10

for n in [10000, 100000, 1000000, 10000000]:
    time_sum = 0

    for i in range(m):
        time_sum += sum_of_n_2_time(n)

    print(f"1부터 {n:8d}까지 더하는데 평균적으로 {time_sum/m:.16f}초 걸림.")
```

1부터 10000까지 더하는데 평균적으로 0.0000003337860107초 걸림.

1부터 100000까지 더하는데 평균적으로 0.0000001668930054초 걸림.

1부터 1000000까지 더하는데 평균적으로 0.0000000000000000초 걸림.

1부터 10000000까지 더하는데 평균적으로 0.000000953674316초 걸림.

# 시간 복잡도

- 실행 시간이 `sum_of_n()` 함수보다 훨씬 빠르지만 실행 시간을 절대적인 기준으로 사용되기 어려움.
- 프로그램 실행시간은 사용되는 컴퓨터, 실행 환경, 컴파일러, 프로그래밍언어 등등에 의존하기 때문임.
- 시간 복잡도를 이용한 알고리즘 복잡도 분석이 요구됨.

## 계산단위

- 특정 연산자 또는 특정 명령문 등의 실행 횟수 확인
- 계산단위: 실행시간을 측정하기 위해 사용되는 연산자 또는 특정 명령문
- 무엇을 계산단위로 지정할 것인가는 알고리즘에 따라 적정하게 지정

## sum\_of\_n() 함수의 일정 시간 복잡도

```
def sum_of_n(n):
    the_sum = 0 ..... # 한 번 할당
    for i in range(1, n + 1):
        the_sum = the_sum + i .. # n 번 할당

    return the_sum
```

- sum\_of\_n() 함수 알고리즘의 기본 계산 단위: 변수 할당
- 변수 할당을 계산단위로 사용할 때의 sum\_of\_n() 함수의 일정 시간 복잡도

$$T(n) = n + 1$$

- $T(n)$ 의 의미: 크기가  $n$ 인 입력값에 대해  $T(n)$ 의 시간이 지나면 해당 알고리즘이 반환값을 계산하고 종료한다

## sum\_of\_n\_2() 함수의 일정 시간 복잡도

```
def sum_of_n_2(n):
    sum = (n * (n + 1)) / 2
    return sum
```

$$T(n) = 1$$

## 예제: 일정 시간 복잡도 계산

In [11]:

```
def fun(n):
    a = 5
    b = 6
    for i in range(n):
        for j in range(n):
            x = i * i
            y = j * j
            z = i * j
    for k in range(n):
        w = a * k + 45
        v = b * b
```

- 계산단위: 변수 할당
- 일정 시간 복잡도

$$T(n) = 3n^2 + 2n + 2$$

## sum\_of\_n() 함수의 Big-O 표현식

- $T(n) = n + 1$ 에서 1은 별로 중요하지 않음
  - 이유: n이 커질 수록  $n+1$  과 n의 차이는 무시될 수 있음.
- $T(n)$ 의 Big-O 표현식

$$T(n) \in O(n)$$

## Big-O 표현식

시간 복잡도 함수를 Big-O 표현식으로 표현하는 일반적인 방법

- 입력 크기  $n$ 이 매우 커질 때 가장 중요한 역할을 수행하는 항 지정
- 그런 항이  $f(n)$  이라 할 때 아래처럼 표기:

$$T(n) \in O(f(n))$$

**예제:**  $T(n) = 3n^2 + 2n + 2$

- 고차항이 가장 중요.
- 상수배는 컴퓨터의 성능에 따라 발생할 수 있는 요인이기에 무시함.

$$T(n) \in O(n^2)$$

**예제:**  $T(n) = \frac{1}{1000}n \log n + 3n + 205$

- $n$  보다  $n \log n$  이 더 큼.

$$T(n) \in O(n \log n)$$

예제:  $T(n) = c$ 의 시간 복잡도 ( $c$ 는 상수)

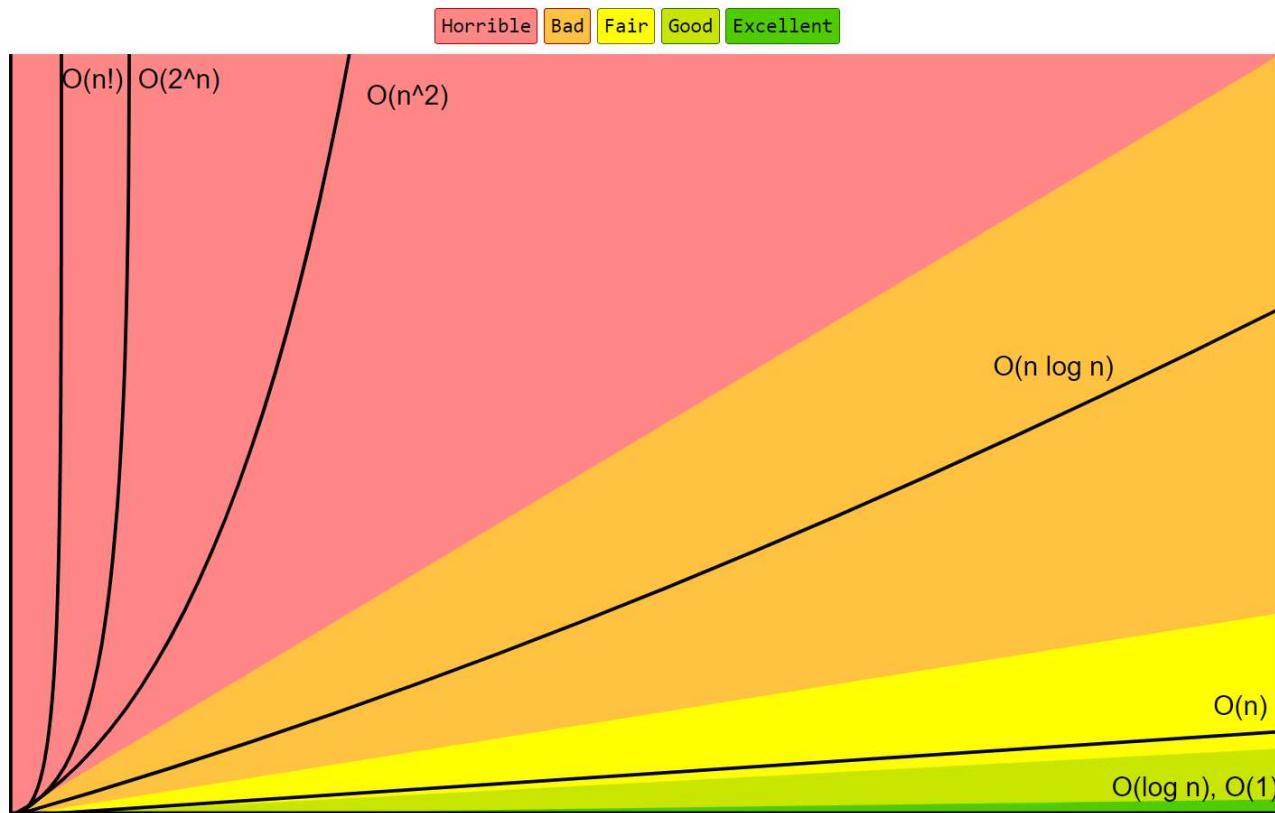
- 상수배는 무시

$$T(n) \in O(1)$$

## 주요 시간 복잡도 함수

시간 복잡도 의미	
1	상수 시간
$\log n$	로그 시간
$n$	선형 시간
$n \log n$	로그선형 시간
$n^2$	2차 시간
$2^n$	지수 시간
$n!$	계승 시간

# 시간 복잡도 함수의 그래프



## 시간 복잡도와 실행시간

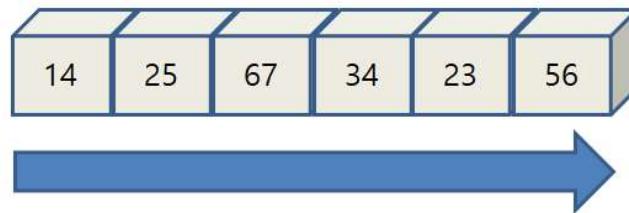
- 가정: 계산 단위 실행 시간 = 1 ns(나노 초, 10억 분의 1 초)
- ms(밀리 초): 천 분의 1초,  $\mu$ s(마이크로 초): 100만 분의 1초
- $n$ : 입력 크기

$n$	$\lg n$	$n$	$n \lg n$	$n^2$	$2^n$
10	0.003 $\mu$ s	0.01 $\mu$ s	0.033 $\mu$ s	0.10 $\mu$ s	1 $\mu$ s
20	0.004 $\mu$ s	0.02 $\mu$ s	0.086 $\mu$ s	0.40 $\mu$ s	1 ms
30	0.005 $\mu$ s	0.03 $\mu$ s	0.147 $\mu$ s	0.90 $\mu$ s	1 초
40	0.005 $\mu$ s	0.04 $\mu$ s	0.213 $\mu$ s	1.60 $\mu$ s	18.3 분
50	0.006 $\mu$ s	0.05 $\mu$ s	0.282 $\mu$ s	2.50 $\mu$ s	13 일
$10^2$	0.007 $\mu$ s	0.10 $\mu$ s	0.664 $\mu$ s	10.00 $\mu$ s	$4 \times 10^{13}$ 년
$10^3$	0.010 $\mu$ s	1.00 $\mu$ s	9.966 $\mu$ s	1.00 ms	
$10^4$	0.013 $\mu$ s	10.00 $\mu$ s	130.000 $\mu$ s	100.00 ms	
$10^5$	0.017 $\mu$ s	0.10 ms	1.670 ms	10.00 초	
$10^6$	0.020 $\mu$ s	1.00 ms	19.930 ms	16.70 초	
$10^7$	0.023 $\mu$ s	0.01 초	0.230 초	1.16 일	
$10^8$	0.027 $\mu$ s	0.10 초	2.660 초	115.70 일	
$10^9$	0.030 $\mu$ s	1.00 초	29.900 초	31.70 년	

# 최선, 최악, 평균 시간 복잡도

- 알고리즘의 시간 복잡도가 입력 크기뿐만 아니라 입력값 자체에 의존할 수도 있음.
- 일정 시간 복잡도  $T(n)$  계산 불가능
- 최선, 최악, 평균 시간 복잡도를 계산 필요

## 순차 탐색



```
In [19]: def seqSearch(sequence, value):
    for item in sequence:
        if item == value:    # 계산 단위
            return True
    return found
```

```
In [20]: seq = [54, 26, 93, 17, 77, 31, 44, 55, 20, 65]
seq_sorted = [17, 20, 26, 31, 44, 54, 55, 65, 77, 93]
print(seqSearch(seq, 93))
print(seqSearch(seq_sorted, 93))
```

True

True

## seqSearch() 함수의 일정 시간 복잡도 $T(n)$ 계산 가능?

- 입력 크기  $n$ : 첫째인자로 사용되는 리스트의 길이
- 계산 단위: 비교 연산  $==$  명령문
- `seqSearch(testList, 93)` 호출: 비교를 세 번 실행
- `seqSearch(testList_sorted, 93)` 호출: 비교 열 번 실행
- 동일한 길이의 리스트를 사용하더라도 리스트에 포함된 항목들의 순서에 따라 실행시간이 달라짐.
- 일정 시간 복잡도  $T(n)$ 을 계산 불가능

## seqSearch() 함수의 최선, 최악, 평균 시간 복잡도

- 입력 크기  $n$ 에 의존하는 시간 복잡도의 최솟값, 최댓값, 평균값 계산 가능
- 알고리즘의 **최선**(best), **최악**(worst), **평균**(average) 시간 복잡도 계산
- 입력 크기는 리스트의 길이로 지정

	최선	최악	평균
항목인 경우	1	$n$	$n/2$
항목이 아닌 경우	$n$	$n$	$n$

## $B(n), W(n), A(n)$

- $B(n), W(n), A(n)$ : 각각 최선, 최악, 평균 시간 복잡도 함수를 가리킴
- 일정 시간 복잡도  $T(n)$ 이 존재할 때:

$$T(n) = B(n) = A(n) = W(n)$$

- 일정 시간 복잡도  $T(n)$ 이 존재하지 않을 때:

$$B(n) \leq A(n) \leq W(n)$$

## 어구전철

- 단어를 구성하는 문자의 순서를 바꾸어 새로운 단어 생성하기
- 영어로 **애너그램**<sup>anagram</sup>
- 예제:
  - "국왕", "왕국"
  - "다들 힘내", "힘내 다들", "내 힘들다"
  - "heart", "earth"
  - "python", "typhon"

## 어구전철 확인 알고리즘 1: 일일이 확인하기

```
In [1]: def anagram_solution_1(s1, s2):
    if len(s1) != len(s2):
        return False

    n = len(s1)
    s2_list = list(s2)

    for idx1 in range(n):
        found = False
        for idx2 in range(n):
            if s1[idx1] == s2_list[idx2]:
                found = True
                s2_list[idx2] = None
                break

        if not found:
            return False

    return True
```

## 두 문자열이 서로 어구전철일 때

- 계산단위: 두 문자열 항목들 사이의 비교 연산
- 입력 크기: 문자열의 길이
- 두 문자열이 서로 어구전철일 때

$$\begin{aligned} T(n) &= \sum_{i=1}^n i \\ &= \frac{n(n+1)}{2} \\ &= \frac{1}{2}n^2 + \frac{1}{2}n \in O(n^2) \end{aligned}$$

## 두 문자열의 길이가 같지만 서로 어구전철 관계가 아닐 때

- 최선: 문자열 `s1`의 첫째 문자가 리스트 `s2_list`에 포함되지 않은 경우에 어구전  
철이 아니라고 판단

$$B(n) = n$$

- 최악: 첫째 문자열 `s1`의 마지막 문자가 리스트 `s2_list`에 포함되지 않은 경우에 어구전철이 아니라고 판단

- 처음  $(n - 1)$ 개의 문자를 확인하는데 걸리는 최악 시간:

$$2 + 3 + \cdots + n$$

- 마지막 문자가 `s2_list`에 없는 것을 확인하는 데에  $n$  번의 비교 필요

$$\begin{aligned}W(n) &= 2 + 3 + \cdots + n + n \\&= \frac{n(n+1)}{2} - 1 + n \\&= \frac{1}{2}n^2 + \frac{3}{2}n - 1 \\&\in O(n^2)\end{aligned}$$

## 어구전철 확인 알고리즘 2: 정렬 후 비교

- 두 문자열이 서로 어구전철인 경우: 비교 연산자가  $n$  번 실행
- `sort()` 함수의 시간 복잡도: 알고리즘에 따라  $O(n^2)$  또는  $O(n \log n)$
- 따라서 위 알고리즘이 시간 복잡도는 사용되는 정렬 알고리즘의 시간 복잡도와 동일

```
In [15]: def anagram_solution_2(s1, s2):  
    a_list_1 = list(s1)  
    a_list_2 = list(s2)  
    a_list_1.sort()  
    a_list_2.sort()  
    pos = 0  
    matches = True  
    while pos < len(s1) and matches:  
        if a_list_1[pos] == a_list_2[pos]:  
            pos = pos + 1  
        else:  
            matches = False  
    return matches
```

## 어구전철 확인 알고리즘 3: 빈도 활용

- 모든 알파벳에 대해 각 문자열에 포함된 빈도를 측정 후 비교
- 길이가  $n$ 인 문자열에 포함된 문자들의 빈도를 확인하는 시간 복잡도

$$B(n) = 2n + 1 \in O(n)$$

$$W(n) = 2n + 26 \in O(n)$$

```
In [16]: def anagram_solution_3(s1, s2):
    alphabets = "abcdefghijklmnopqrstuvwxyz"
    alpha_dict1 = dict()
    alpha_dict2 = dict()
    for s in alphabets:
        alpha_dict1[s] = 0
        alpha_dict2[s] = 0
    for s in s1.lower():
        alpha_dict1[s] += 1
    for s in s2.lower():
        alpha_dict2[s] += 1
    for alpha in alphabets:
        if alpha_dict1[alpha] == alpha_dict2[alpha]:
            continue
        else:
            return False
    return True
```

## 알고리즘 2 vs. 알고리즘 3

- 입력 문자열의 길이: 2백60만, 2천600만, 2억6천만
- 평가에 사용된 PC의 성능
  - 프로세서: 11th Gen Intel(R) Core(TM) i9-11950H @ 2.60GHz 2.61 GHz
  - RAM: 32GB

## 입력값 생성

영어 알파벳 26개로 구성된 문자열을 10만번, 100만번, 천만번 이어붙이기

In [22]: `from time import time`

```
s1 = alphabets  
len(s1)
```

Out[22]: 26

## 알고리즘 2: 입력 크기 2백60만

```
In [23]: start = time()  
  
k = 100_000  
anagram_solution_2(s1*k, s1*k)  
  
print(time()-start)
```

0.35188865661621094

## 알고리즘 2: 입력 크기 2천600만

```
In [24]: start = time()  
  
k = 1_000_000  
anagram_solution_2(s1*k, s1*k)  
  
print(time()-start)
```

3.6589527130126953

## 알고리즘 2: 입력 크기 2억6천만

```
In [25]: start = time()  
  
k = 10_000_000  
anagram_solution_2(s1*k, s1*k)  
  
print(time()-start)
```

39.13177156448364

## 알고리즘 3: 입력 크기 2백60만

```
In [26]: start = time()  
  
k = 100_000  
anagram_solution_3(s1*k, s1*k)  
  
print(time()-start)
```

0.19489121437072754

## 알고리즘 3: 입력 크기 2천600만

```
In [27]: start = time()  
  
k = 1_000_000  
anagram_solution_3(s1*k, s1*k)  
  
print(time()-start)
```

2.007293939590454

## 알고리즘 3: 입력 크기 2억6천만

```
In [28]: start = time()  
  
k = 10_000_000  
anagram_solution_3(s1*k, s1*k)  
  
print(time()-start)
```

19.824604749679565

## 실험 결과

- 실행 시간 자체는 별로 중요하지 않음. 실험에 사용된 컴퓨터 환경에 따라 실행 시간이 조금씩 다를 수 있기 때문.
- 아래 사실이 중요
  - 알고리즘 2: 입력 크기가 10배일 때 실행시간은 10배보다 조금 더 깊
  - 알고리즘 3: 입력 크기가 10배일 때 실행시간은 10배 정도 오래 걸림
- 참고: 알고리즘 2의 경우 정렬 알고리즘의 시간 복잡도가  $O(n \ln n)$  정도

입력 크기	알고리즘 2	알고리즘 3
2백60만	0.363136초	0.208275초
2천600만	3.745986초	1.935389초
2억6천만	39.513536초	19.316656초

# 공간 복잡도 문제

- `anagram_solution_3()` 의 시간 복잡도는 이전 세 알고리즘에 비해 훨씬 좋음.
- 하지만 그 대신에 빈도 리스트를 새로 생성하기 위해 보다 많은 메모리를 사용
  - `anagram_solution_1()`: 문자열 `s2`를 리스트로 변환한 값
  - `anagram_solution_2()`: 두 문자열을 리스트로 형변환한 후 정렬. 그리고 정렬 과정에서 추가 메모리 사용 가능.