



[스파르타코딩클럽] 알고보면 알기쉬운 알고리즘 - 1주차



매 주차 강의자료 시작에 PDF파일을 올려두었어요!

▼ PDF 파일

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/bbc810c4-91c6-4195-ae6c-7e0ca420dab0/_____.pdf

[수업 목표]

1. 개발자들에게 알고리즘 공부에 필요한 이유를 이해한다.
2. 알고리즘을 학습하기 위한 기본 코드 구현력을 높인다.
3. 시간 복잡도, 공간 복잡도에 대해 배운다.

[목차]

- 01. 오늘 배울 것
- 02. 파이썬으로 코딩하기
- 03. 알고리즘과 친해지기 (1)
- 04. 알고리즘과 친해지기 (2)
- 05. 시간 복잡도 판단하기
- 06. 공간 복잡도 판단하기
- 07. 점근 표기법
- 08. 알고리즘 더 풀어보기 (1)
- 09. 알고리즘 더 풀어보기 (2)
- 10. 끝 & 과제 설명



모든 토글을 열고 닫는 단축키

Windows : **Ctrl** + **alt** + **t**

Mac : **⌘** + **⌥** + **t**

01. 오늘 배울 것

▼ 알고리즘이란?



어떤 문제의 해결을 위하여, 입력된 자료를 토대로 하여 원하는 출력을 유도하여 내는 규칙의 집합. 여러 단계의 유한 집합으로 구성되는데, 각 단계는 하나 또는 그 이상의 연산을 필요로 한다. [표준국어대사전]

- 어떤 문제가 있을때, 그것을 해결하기 위한 여러 동작들의 모임입니다.

- 그런데, 하나의 문제를 풀기 위해서는 다양한 방법이 있을 수 있습니다.
예를 들어 엄마가 계란과 돼지고기를 사오라고 하셨다면 여러 방법이 떠오를 수 있습니다.
- 1. 집앞 롯데슈퍼에서 계란과 돼지고기를 사온다.
- 2. 시장 골목에 있는 노점상의 계란과 정육점의 돼지고기를 사오고 남은 돈으로 아이스크림을 사먹는다.
- 3. 시골에 계시는 할머니집에 내려가 농사 일을 하루동안 도와주고 일상으로 계란과 고기를 받는다.
- 4. ...
- 이 방법들 중에 어떤 게 가장 좋은 방법인가요? 시간이 가장 덜 드는 것? 돈이 가장 덜 드는 것? 필요한 계란과 돼지고기의 양에 따라 답이 달라질 수도 있나요?

▼ 알고리즘을 공부해야하는 이유

▼ 1. 좋은 개발자가 되고 싶어요!

- 개발자는 프로그램을 만드는 직업입니다. 즉, 좋은 개발자가 되려면? **좋은 프로그램을 구현할** 줄 알아야 합니다.
- 좋은 프로그램이란? **적은 공간**을 이용해서 **빠른 속도**로 수행되는 프로그램입니다!
- 그런 프로그램을 만들기 위해서는 경우에 따라 **특정 자료구조나 접근방법**을 사용해야 합니다. 즉, 프로그램을 잘하기 위해서는 여러 자료구조와 방법들을 배우고 익혀야 좋은 프로그램을 만들 수 있습니다.
- 막연히 개발만 하다보면 좋은 코드를 만들지 못합니다! 자료구조와 알고리즘에 대해서 배워 더 좋은 프로그램을 만들어 보자구요!

▼ 2. 좋은 회사에 취직하고 싶어요!

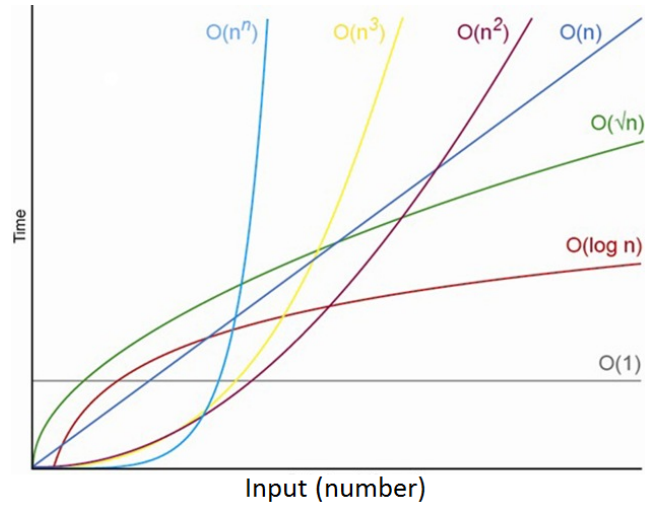
- 수많은 회사들이 코딩테스트를 통해 개발자를 구인하고 있습니다. 카카오, 삼성, 구글 등 국내외 유망 IT 기업들 외에도 많은 스타트업까지 코딩테스트를 개발자의 **필수 관문**으로 만들고 있습니다.
- **그러나**, 엄청나게 어려운 수준의 문제를 출제하진 않습니다.
- 기초적인 지식과 해결책으로 적절한 사고를 할 수 있는지에 대해 검증하기 위함이므로, 5주차 동안 잘 따라오신다면 충분히 해결하실 수 있으리라고 생각합니다.
- (그렇다고 이런 문제를 항상 풀어야 하는 것은 아닙니다. 취업하고부터는 이미 잘 만들어진거 쓰시면 됩니다 😊)

▼ 1~5주차에 배우는 것들!

▼ 1주차: 시간/공간 복잡도, 알고리즘 구현력 기르기



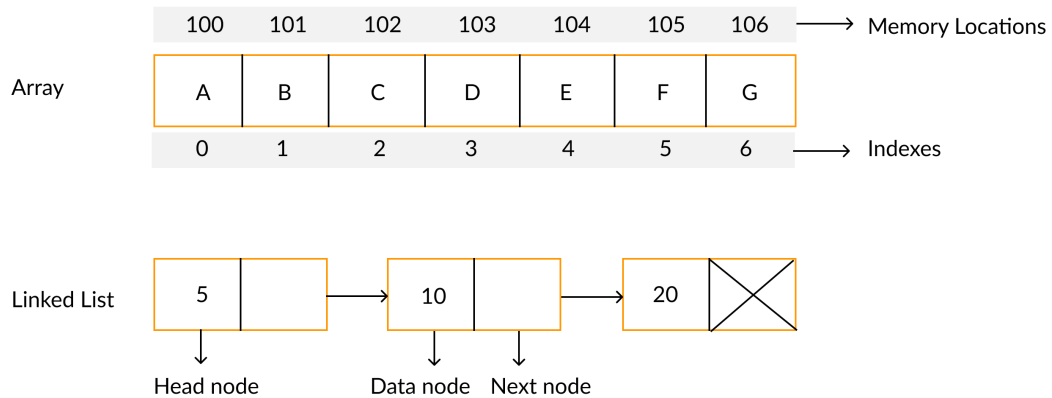
오늘은 시간/공간 복잡도와 알고리즘 구현력 기르기! 기본적인 알고리즘의 개념과 좋은 알고리즘이 무엇인지, 또 알고리즘을 구현하기 위해서 필요한 기본적인 코딩 근육을 늘려볼 거예요



<https://medium.com/@randerson112358/algorithm-analysis-time-complexity-simplified-cd39a81fec71> 출처

▼ 2주차: 어레이, 링크드 리스트, 이분탐색, 재귀

👉 어레이와 링크드 리스트의 개념 학습! 그리고 이 두가지 자료구조를 이용해서 특정 데이터를 탐색하고 삽입, 정렬, 삭제를 해보는 시간을 가질 거예요.

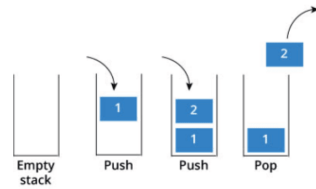


<https://www.faceprep.in/data-structures/linked-list-vs-array/> 출처

▼ 3주차: 정렬, 스택, 큐, 해쉬

👉 코딩 테스트에 나오는 대표적인 자료구조인 **스택, 큐, 해쉬, 힙**에 대해서 배울 거예요. 먼저 해당 자료구조들은 어떤 경우에 쓰는지 좋은지 살펴보고, 직접 문제를 풀며 응용해보겠습니다.

Data Structure Basics



Stack

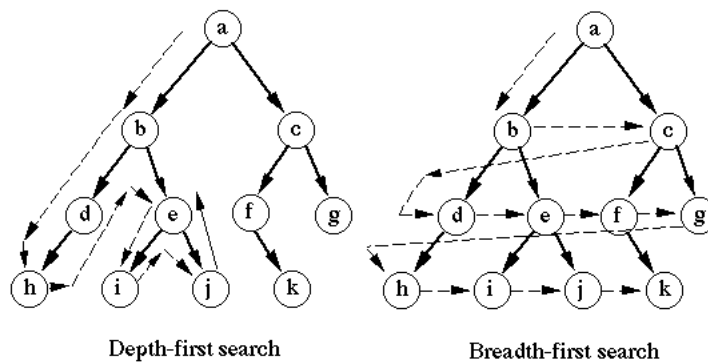


Queue

<https://dev.to/rinsama77/data-structure-stack-and-queue-4ecd> 출처

▼ 4주차: 힙, BFS, DFS, Dynamic Programming

👉 힙, BFS, DFS, Dynamic Programming 배워볼 거예요! 이름만 들으면 되게 무서워보이는데, 사실 지금까지 배운 개념들을 활용해서 해결할 수 있으니 너무 걱정마세요!



▼ 5주차: 종합 알고리즘 문제 풀이

👉 여러분들은 이제 코딩테스트를 풀기 위한 만반의 준비를 마쳤습니다! 실제 기업에서 출제되었던 문제들을 토대로 시험보듯이 풀어봅시다! 여러분의 코딩 근육을 한 단 계 더 업그레이드 시켜줄 거예요!

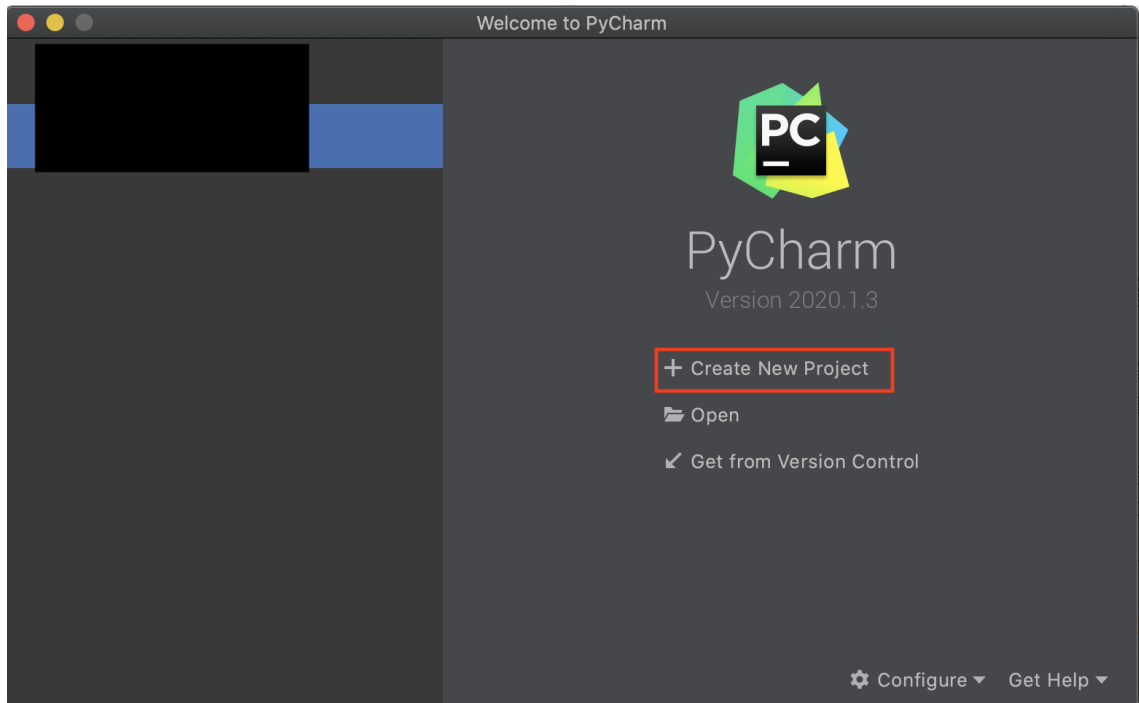


02. 파이참으로 코딩하기

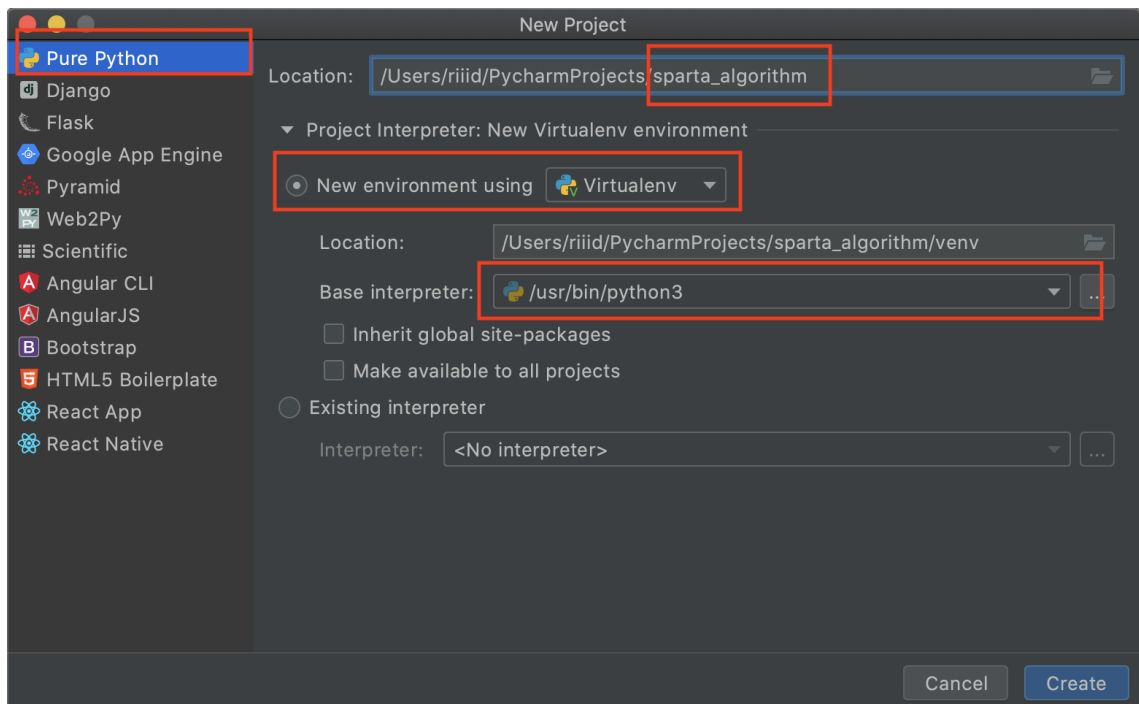
▼ 1) 새 프로젝트 만들기

▼ Pycharm 을 켜고 새 프로젝트를 만들어봅시다!

- 'Create New Project' 클릭!



- Pure Python 프로젝트를 sparta_algorithm 이라는 이름으로 만들어 볼게요! 아래처럼요!



Location에서 폴더 눌러서 `sparta_algorithm` 폴더 만들어 선택

Virtualenv 이용해서 새 가상환경 만들기

Base interpreter는 새로 설치한 Python 3.X로! (2.7은 아니되어요~)

- ▼ 가상환경이란(virtual environment)? - 프로젝트별로 패키지들을 담은 공구함!



가상환경(virtual environment)은

같은 시스템에서 실행되는 다른 파이썬 응용 프로그램들의 동작에 영향을 주지 않기 위해, 파이썬 배포 패키지들을 설치하거나 업그레이드하는 것을 가능하게 하는 **격리된 실행 환경**입니다.

출처 : 파이썬 공식 용어집- 가상환경

- 한 프로젝트에서 쓰는 패키지가 한두 개가 아닌데, 다 설치해버리면 되나요? 각 프로젝트마다 필요로 하는 패키지의 버전이 다르다면? 프로젝트 별로 컴퓨터를 새로 사야하나요?
- 이런 상황에 쓸 수 있는 게 바로 가상환경입니다! 각 프로젝트마다 가상환경을 만들고 그 안에 필요한 패키지를 설치하는 것이죠 😊

▼ 2) 디렉토리 관리하기

▼ 개발자처럼 파일 관리하는 방법

- **!** 개발자의 보물은 바로 코드! 내 보물이 어디있는지 모르면 난감하겠죠?
- 1. 어떤 역할을 하는 폴더와 파일인지 한눈에 파악할 수 있게
- 2. 다른 사람들과 협업할 때도 함께 정한 규칙대로 관리하면 유용합니다.

▼ 파일/폴더 이름 짓기(naming) 기본 규칙

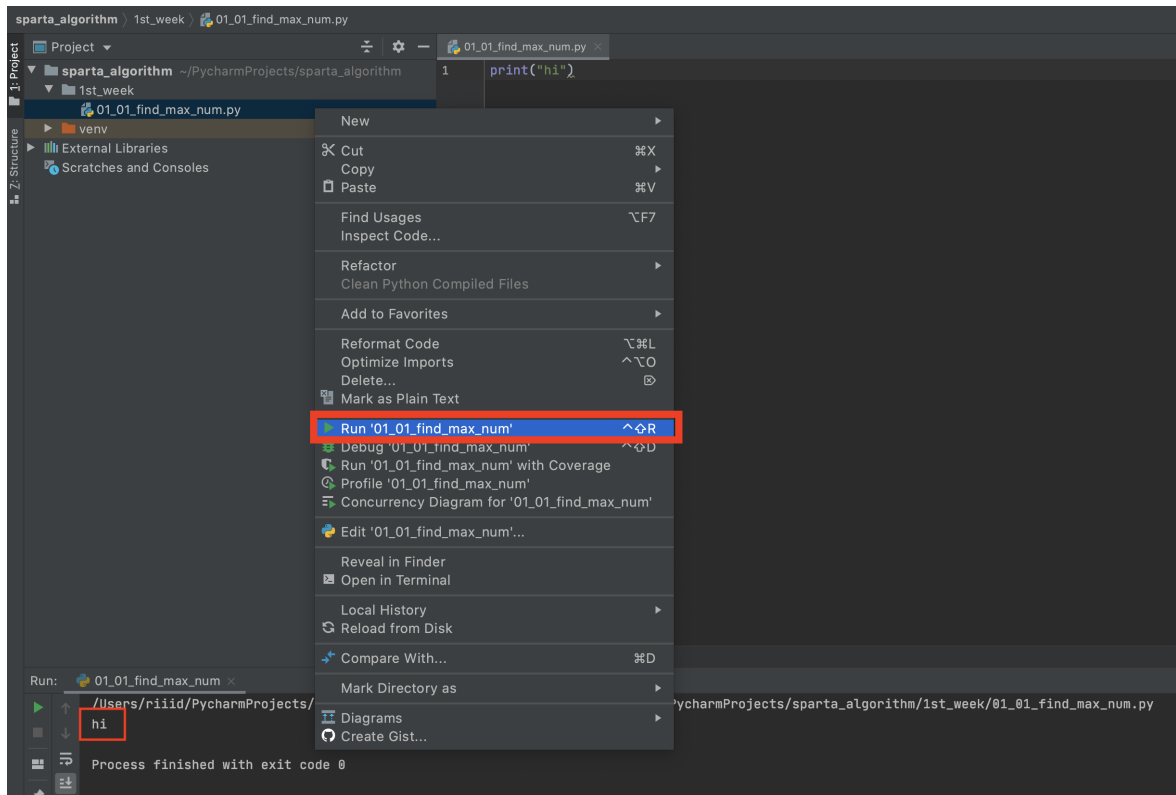
- **폴더/파일이 어떤 내용인지 파악할 수 있게 적기**
 - 개발자는 이름짓기(naming)을 정말 중요하게 생각한답니다. 특히 웹 프로그래밍은 데이터를 주고받는 과정입니다. 어떤 데이터(내용)를 담고 있는지 한 눈에 알 수 있는게 좋아요. 앞으로 배울 내용에도 꾸준히 이름짓기(naming)에 대한 내용이 나올 거예요.
 - 예) a → 무슨 폴더지? / homework → 숙제 폴더구나
- 파일과 폴더 이름은 영어로 : 가끔 컴퓨터가 한글을 인식하지 못하는 경우가 있어요.
- 특수문자는 `_` 만 사용하기 : 다른 특수문자(띄어쓰기 포함)을 컴퓨터가 알아듣게 하려면 조금 수고로워요. 우리는 단어를 연결할 때, `_` 를 사용하겠습니다.
- 그래서 우리는 `1st_week` 이라는 이름으로 폴더를 만들어주겠습니다!
`sparta_algorithm` 폴더 우클릭 > New > Directory를 클릭해서 새 폴더를 만들어주세요 😊

▼ 3) 파이썬 파일 만들고 실행하기

- 이제 `1st_week` 폴더를 다시 우클릭 > New > Python File을 선택해서 새 파이썬 파일을 만들어주겠습니다. 이 파일의 이름은 `01_01_find_max_num.py` 로 하겠습니다. 1주차 1번 최댓값을 찾는 코드라는 뜻이겠죠?
- 새로 생긴 파일에 아래처럼 코드를 넣어볼까요?

```
print("hi")
```

- 이 코드를 실행하고 싶을 때는, 파일 우클릭 > Run '01_01_find_max_num'을 선택하면 끝!



🔥 콘솔창에 hi가 떴다면 공부 준비 완료 😊

03. 알고리즘과 친해지기 (1)

🔥 우선 연습문제를 풀어보면서 알고리즘과 친해져봅시다!

▼ 4) 🛠️ 최댓값 찾기

▼ Q. 문제 설명

❓ Q. 다음과 같이 숫자로 이루어진 배열이 있을 때, 이 배열 내에서 가장 큰 수를 반환하시오.

```
[3, 5, 6, 1, 2, 4]
```

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! 2분 정도 고민해 본 다음, 아래 방법들을 펼쳐 봅시다!

▼ [코드스니펫] 최댓값 찾기 문제

```
input = [3, 5, 6, 1, 2, 4]

def find_max_num(array):
```



```
# 이 부분을 채워보세요!
return 1

result = find_max_num(input)
print(result)
```

▼ A1. 첫 번째 방법

✅ 각 숫자마다 모든 다른 숫자와 비교해서 최대값인지 확인합니다. 만약 다른 모든 값보다 크다면 반복문을 중단합니다. (비교를 위해 조금 이상하게 구현했습니다. 앞으로 이 강의를 들으면 이렇게 작성하지 않으실 거예요!)

💡 python의 `for ~ else` 문은 “`for` 문에서 `break` 가 발생하지 않았을 경우”의 동작을 `else` 문에 적어주는 것이다.

```
input = [3, 5, 6, 1, 2, 4]

def find_max_num(array):
    for num in array:
        for compare_num in array:
            if num < compare_num:
                break
        else:
            return num

result = find_max_num(input)
print(result)
```

▼ A2. 두 번째 방법

✅ 배열 내에서 가장 큰 수를 찾아야 합니다. 그러면, 가장 큰 수를 저장할 변수를 만들고, 배열을 돌아가면서 그 변수와 비교합니다! 만약 값이 더 크다면, 그 변수에 대입해주면 됩니다!

```
input = [3, 5, 6, 1, 2, 4]

def find_max_num(array):
    max_num = array[0]
    for num in array:
        if num > max_num:
            max_num = num
    return max_num

result = find_max_num(input)
print(result)
```

? 여러분은 위 두 가지 풀이 방법 중 어떤 게 효율적인 함수인 것 같나요?

04. 알고리즘과 친해지기 (2)

▼ 5) 🚧 최빈값 찾기

▼ Q. 문제 설명

? Q. 다음과 같은 문자열을 입력받았을 때, 어떤 알파벳이 가장 많이 포함되어 있는지 반환하시오

"hello my name is sparta"

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! 2분 정도 고민해 본 다음, 아래 방법들을 펼쳐 봅시다!

▼ [코드스니펫] 최빈값 찾기

```
input = "hello my name is sparta"

def find_max_occurred_alphabet(string):
    # 이 부분을 채워보세요!
    return "a"

result = find_max_occurred_alphabet(input)
print(result)
```

▼ 🌟 여기서 잠깐! Tip

▼ 문자인지 확인하는 방법

파이썬의 내장 함수 `str.isalpha()` 를 이용하면 해당 문자열이 알파벳인지 확인할 수 있습니다!

```
print("a".isalpha()) # True
print("1".isalpha()) # False

s = "abcdefg"
print(s[0].isalpha()) # True
```

▼ 알파벳 별로 빈도수를 리스트에 저장하기

- 우선 알파벳 별 빈도수를 저장하기 위한 길이가 26인 0으로 초기화된 배열을 만듭니다.

```
alphabet_occurrence_array = [0] * 26
```

- 이제 이 배열의 각 원소에 알파벳마다 빈도수를 추가해줘야 합니다. a일 때는 0번째 원소에 1을 추가하고, b일 때는 1번째 원소에 1을 추가해줘야 하는데, 이를 어떻게 해줄 수 있을까요?
- 바로 **아스키 (ASCII) 코드**를 사용해야 합니다. 컴퓨터는 0과 1 숫자 밖에 모르기 때문에 문자도 숫자로 기억합니다. 이 때, 어떤 숫자와 어떤 문자를 대응시키는가에 따라 여러 가지 인코딩 방식이 있는데 통상 **아스키 코드** 방식을 많이 사용합니다.

제어 문자			공백 문자			구두점			숫자			알파벳		
10진	16진	문자	10진	16진	문자	10진	16진	문자	10진	16진	문자	10진	16진	문자
0	0x00	NUL	32	0x20	SP	64	0x40	@	96	0x60				
1	0x01	SOH	33	0x21	!	65	0x41	A	97	0x61	a			
2	0x02	STX	34	0x22	"	66	0x42	B	98	0x62	b			
3	0x03	ETX	35	0x23	#	67	0x43	C	99	0x63	c			
4	0x04	EOT	36	0x24	\$	68	0x44	D	100	0x64	d			
5	0x05	ENQ	37	0x25	%	69	0x45	E	101	0x65	e			
6	0x06	ACK	38	0x26	&	70	0x46	F	102	0x66	f			
7	0x07	BEL	39	0x27	'	71	0x47	G	103	0x67	g			
8	0x08	BS	40	0x28	(72	0x48	H	104	0x68	h			

- 참고로, 이 아스키 코드를 다 외울 필요는 전—혀 없습니다! 그냥 이런 게 있고, 문자를 숫자로 바꿀 때 필요하구나, 정도만 생각하고 나중에 찾아보시면 됩니다. 저도 하나도 몰라요 ㅎㅎ
- 자, 그러면 문자를 아스키코드 변환시켜야 하는데, 어떻게 할까요?
- 구글링 해보겠습니다! "**python char to ascii code**" 라고 검색하면, ord 함수를 사용하라고 나옵니다! 그러면 앞으로 ord 함수를 이용해봅시다!

```
# 내장 함수 ord() 이용해서 아스키 값 받기
print(ord('a'))          # 97
print(ord('a') - ord('a')) # 97-97 -> 0
print(ord('b') - ord('a')) # 98-97 -> 1
```

- 파이썬에는 이런 수많은 함수들이 있습니다. 어떻게 외우냐구요? 전.혀. 외울 필요 없습니다. 필요한 지식만 있다면 필요할 때마다 구글링해서 방법을 찾아오시면 됩니다!
- 이제 각 알파벳의 빈도수를 세어볼까요?

▼ [코드스니펫] 알파벳 빈도수 세기

```
def find_alphabet_occurrence_array(string):
    alphabet_occurrence_array = [0] * 26

    # 이 부분을 채워보세요!

    return alphabet_occurrence_array

print(find_alphabet_occurrence_array("hello my name is sparta"))
```

▼ 정답 보기

```
def find_alphabet_occurrence_array(string):
    alphabet_occurrence_array = [0] * 26

    for char in string:
        if not char.isalpha():
            continue
        arr_index = ord(char) - ord('a')
        alphabet_occurrence_array[arr_index] += 1

    return alphabet_occurrence_array
```

▼ A1. 첫 번째 방법

- ✓ 각 알파벳마다 문자열을 돌면서 몇 글자 나왔는지 확인합니다. 만약 그 숫자가 저장한 알파벳 빈도 수보다 크다면, 그 값을 저장하고 제일 큰 알파벳으로 저장합니다. 이 과정을 반복하다보면 가장 많이 나왔던 알파벳을 알 수 있습니다.

```
input = "hello my name is sparta"

def find_max_occurred_alphabet(string):
    alphabet_array = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "x", "y", "z"]
    max_occurrence = 0
    max_alphabet = alphabet_array[0]

    for alphabet in alphabet_array:
        occurrence = 0
        for char in string:
            if char == alphabet:
                occurrence += 1

        if occurrence > max_occurrence:
            max_alphabet = alphabet
```

```

        max_occurrence = occurrence

    return max_alphabet

result = find_max_occurred_alphabet(input)
print(result)

```

▼ A2. 두 번째 방법

✓ 각 알파벳의 빈도수를 `alphabet_occurrence_list` 라는 변수에 저장합니다. 그리고 각 문자열을 돌면서 해당 문자가 알파벳인지 확인하고, 알파벳을 인덱스화 시켜 각 알파벳의 빈도수를 업데이트 합니다.

이후, 알파벳의 빈도수가 가장 높은 인덱스를 찾습니다.

```

input = "hello my name is sparta"

def find_max_occurred_alphabet(string):
    alphabet_occurrence_array = [0] * 26

    for char in string:
        if not char.isalpha():
            continue
        arr_index = ord(char) - ord('a')
        alphabet_occurrence_array[arr_index] += 1

    max_occurrence = 0
    max_alphabet_index = 0
    for index in range(len(alphabet_occurrence_array)):
        alphabet_occurrence = alphabet_occurrence_array[index]
        if alphabet_occurrence > max_occurrence:
            max_occurrence = alphabet_occurrence
            max_alphabet_index = index

```

✓ 여기까지 했으면, 가장 높은 빈도수의 인덱스를 알아냈습니다.

그러면 이 문제에서는 **max_alphabet_index** 가 0 입니다.
그러면, 이번에는 인덱스를 문자로 변경하려면 어떻게 할까요?

그 반대로, "아스키 코드 번호"를 "실제 문자"로 변환하려면 `chr()` 함수를 사용합니다.

```

chr(97) == 'a'
chr(0 + ord('a')) == 'a'
chr(0 + 97) == 'a'
chr(1 + 97) == 'b'

```

✓ 즉, 다음과 같이 해결할 수 있습니다.

```

input = "hello my name is sparta"

def find_max_occurred_alphabet(string):
    alphabet_occurrence_array = [0] * 26

    for char in string:
        if not char.isalpha():
            continue
        arr_index = ord(char) - ord('a')

```

```

        alphabet_occurrence_array[arr_index] += 1

    max_occurrence = 0
    max_alphabet_index = 0
    for index in range(len(alphabet_occurrence_array)):
        alphabet_occurrence = alphabet_occurrence_array[index]
        if alphabet_occurrence > max_occurrence:
            max_occurrence = alphabet_occurrence
            max_alphabet_index = index

    return chr(max_alphabet_index + ord('a'))

result = find_max_occurred_alphabet(input)
print(result)

```

? Q. 위 두 가지 풀이 방법 중에는 어떤 게 효율적인 함수인 것 같나요?

05. 시간 복잡도 판단하기

▼ 6) 시간 복잡도란?



입력값과 문제를 해결하는 데 걸리는 시간과의 상관관계를 말합니다! 입력값이 2배로 늘어났을 때 문제를 해결하는 데 걸리는 시간은 몇 배로 늘어나는지를 보는 것이죠.
우리는 시간이 적게 걸리는 알고리즘을 좋아하니 입력값이 늘어나도 걸리는 시간이 덜 늘어나는 알고리즘이 좋은 알고리즘이겠죠?

▼ 7) 최댓값 찾기 알고리즘의 시간 복잡도 판단해보기

▼ 첫 번째 방법

```

input = [3, 5, 6, 1, 2, 4]

def find_max_num(array):
    for num in array:
        for compare_num in array:
            if num < compare_num:
                break
        else:
            return num

result = find_max_num(input)
print(result)

```

- 이 해결 방법은 각 숫자마다 모든 다른 숫자와 비교해서 최대값인지 확인합니다. 만약 다른 모든 값보다 크다면 반복문을 중단합니다.
- 이 함수가 시간이 얼마나 걸리는지 어떻게 분석할 수 있을까요?
- 바로, **각 줄이 실행되는 걸 1번의 연산이 된다고** 생각하고 계산하시면 됩니다. 아래와 같이 계산할 수 있습니다.

```

for num in array:           # array 의 길이만큼 아래 연산이 실행
    for compare_num in array: # array 의 길이만큼 아래 연산이 실행
        if num < compare_num: # 비교 연산 1번 실행
            break
    else:
        return max_num

```

- 위에서 연산된 것들을 더해보면,

1. array의 길이 X array의 길이 X 비교 연산 1번

만큼의 시간이 필요합니다. 여기서 array(입력값)의 길이는 보통 N 이라고 표현합니다. 그러면 위의 시간을 다음과 같이 표현할 수 있습니다.

$$N \times N$$

그러면 우리는 이제 이 함수는 N^2 만큼의 **시간이 걸렸겠구나!** 라고 말할 수 있습니다.



Q. 선생님 여기서 입력값이 뭔가요?

A. 함수에서 크기가 변경될 수 있는 값이라고 보시면 됩니다!
배열을 받고 있으니 이 함수에서는 배열이 입력값입니다.

Q. 선생님 그러면 여기서 N 이 6이니까, 36이라고 말하면 안되나요?

A. N 의 크기에 따른 시간의 상관관계를 시간복잡도라고 하는 것이라 수식으로 표현하셔야 합니다!

▼ 두 번째 방법

```
input = [3, 5, 6, 1, 2, 4]

def find_max_num(array):
    max_num = array[0]
    for num in array:
        if num > max_num:
            max_num = num
    return max_num

result = find_max_num(input)
print(result)
```

- 이 해결 방법은 리스트를 하나씩 돌면서 num 과 max_num 값을 비교하는 함수입니다.
- 다시 한 번 시간복잡도를 분석해볼까요?

```
max_num = array[0] # 연산 1번 실행
```

```
for num in array:      # array의 길이만큼 아래 연산이 실행
    if num > max_num:   # 비교 연산 1번 실행
        max_num = num  # 대입 연산 1번 실행
```

- 위에서 연산된 것들을 더해보면,
 1. max_num 대입 연산 1번
 2. array의 길이 X (비교 연산 1번 + 대입 연산 1번)

만큼의 시간이 필요합니다. 첫 번째 방법에서 했던 것처럼 array의 길이를 N 이라고 하면, 다음과 같이 표현할 수 있겠죠?

$$1 + 2 \times N$$

그러면 우리는 이제 이 함수는 $2N + 1$ 만큼의 **시간이 걸렸겠구나!** 라고 말할 수 있습니다.

▼ 비교하기

- 코드만 봐도 두 번째 방법이 좋을 것 같다고 어렵듯이 생각은 했겠지만, 이렇게 수치화시키니 얼마나 효율적인지 정량적으로 분석할 수 있습니다

- 그러면, 이를 수학적으로 표현해보면 첫 번째 방법은 N^2 , 두 번째 방법은 $2N + 1$ 이라는 식이 나온다는 걸 알 수 있습니다. 그러면 N 의 길이가 길어질수록, 다음과 같이 연산량이 변화합니다.

3N^2 + 1 vs 2N+1

Aa N의 길이	# N^2	# 2N + 1
1	1	3
10	100	21
100	10000	201
1000	1000000	2001
10000	100000000	20001

- 이 표를 보면, 두 가지를 깨달을 수 있습니다.
 1. N 과 N^2 은 N 이 커질수록 더 큰 차이가 나는구나!
 2. N 의 지수를 먼저 비교하면 되겠구나.
- 그러나, 저희가 매번 코드를 매 실행 단위로 이렇게 몇 번의 연산이 발생하는지 확인하는 건 불가능합니다. 따라서 상수는 신경 쓰지 말고, 입력값에 비례해서 어느 정도로 증가하는지만 파악하면 됩니다.



즉, $2N + 1$ 의 연산량이 나온 첫번째 풀이 방법은 N 만큼의 연산량이 필요하다
 N^2 의 연산량이 나온 두번째 풀이 방법은 N^2 만큼의 연산량이 필요하다.

참고로, 만약 상수의 연산량이 필요하다면, 1 만큼의 연산량이 필요하다고 말하면 됩니다.

06. 공간 복잡도 판단하기

▼ 8) 공간 복잡도란?



입력값과 문제를 해결하는 데 걸리는 공간과의 상관관계를 말합니다! 입력값이 2배로 늘어났을 때 문제를 해결하는 데 걸리는 공간은 몇 배로 늘어나는지를 보는 것이죠.
 우리는 공간이 적게 걸리는 알고리즘을 좋아하니 입력값이 늘어나도 걸리는 공간이 덜 늘어나는 알고리즘이 좋은 알고리즘이겠죠?

▼ 9) 최빈값 찾기 알고리즘의 공간 복잡도 판단해보기

▼ 첫 번째 방법

```
input = "hello my name is sparta"

def find_max_occurred_alphabet(string):
    alphabet_array = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u"]
    max_occurrence = 0
    max_alphabet = alphabet_array[0]

    for alphabet in alphabet_array:
        occurrence = 0
        for char in string:
            if char == alphabet:
                occurrence += 1

        if occurrence > max_occurrence:
            max_alphabet = alphabet
            max_occurrence = occurrence

    return max_alphabet
```

```
result = find_max_occurred_alphabet(input)
print(result)
```

- 이 해결 방법은 각 알파벳마다 문자열을 돌면서 몇 글자 나왔는지 확인합니다. 만약 그 숫자가 저장한 알파벳 빈도 수보다 크다면, 그 값을 저장하고 제일 큰 알파벳으로 저장합니다. 이 함수가 공간을 얼마나 사용하는지 어떻게 분석할 수 있을까요?
- 바로 **저장하는 데이터의 양이 1개의 공간을 사용한다고** 계산하시면 됩니다. 아래와 같이 계산할 수 있습니다.

```
alphabet_array = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"]
# -> 26 개의 공간을 사용합니다
max_occurrence = 0 # 1개의 공간을 사용합니다
max_alphabet = alphabet_array[0] # 1개의 공간을 사용합니다.

for alphabet in alphabet_array:
    occurrence = 0 # 1개의 공간을 사용합니다
```

- 위에서 사용된 공간들을 더해보면,
 1. alphabet_array 의 길이 = 26
 2. max_occurrence, max_alphabet, occurrence 변수 = 3

29

만큼의 공간이 필요합니다.

그러면 우리는 이제 이 함수는 29 만큼의 **공간이 사용되었구나!** 라고 말할 수 있습니다.

▼ 두 번째 방법

```
input = "hello my name is sparta"

def find_max_occurred_alphabet(string):
    alphabet_occurrence_list = [0] * 26

    for char in string:
        if not char.isalpha():
            continue
        arr_index = ord(char) - ord('a')
        alphabet_occurrence_list[arr_index] += 1

    max_occurrence = 0
    max_alphabet_index = 0
    for index in range(len(alphabet_occurrence_list)):
        alphabet_occurrence = alphabet_occurrence_list[index]
        if alphabet_occurrence > max_occurrence:
            max_occurrence = alphabet_occurrence
            max_alphabet_index = index

    return chr(max_alphabet_index + ord('a'))

result = find_max_occurred_alphabet(input)
print(result)
```

- 이 해결 방법은 각 알파벳의 빈도수를 저장한 다음에, 빈도 수 중 가장 많은 알파벳의 인덱스 값을 반환합니다. 다시 한 번 공간 복잡도를 분석해볼까요?

```
alphabet_occurrence_list = [0] * 26 # -> 26 개의 공간을 사용합니다

for char in string:
    if not char.isalpha():
        continue
    arr_index = ord(char) - ord('a') # 1개의 공간을 사용합니다
    alphabet_occurrence_list[arr_index] += 1

max_occurrence = 0 # 1개의 공간을 사용합니다
max_alphabet_index = 0 # 1개의 공간을 사용합니다
```



```

for index in range(26):
    alphabet_occurrence = alphabet_occurrence_list[index] # 1개의 공간을 사용합니다
    if alphabet_occurrence > max_occurrence:
        max_occurrence = alphabet_occurrence
        max_alphabet_index = index


```


- 위에서 사용된 공간들을 더해보면,
 1. alphabet_array 의 길이 = 26
 2. arr_index, max_occurrence, max_alphabet_index, alphabet_occurrence 변수 = 4

30

만큼의 공간이 필요합니다.

그러면 우리는 이제 이 함수는 30 만큼의 **공간이 사용되었구나!** 라고 말할 수 있습니다.

 그러면, 공간을 더 적게 쓴 첫 번째 방법이 더 효율적인걸까요?

 그렇지 않습니다.

29와 30 모두 상수라 큰 상관이 없습니다.

그러면 뭘로 비교하는 게 좋을까요?

바로 시간 복잡도로 비교하시면 됩니다.

이처럼, 대부분의 문제에서는 알고리즘의 성능이 공간에 의해서 결정되지 않습니다. **따라서 공간 복잡도보다는 시간 복잡도를 더 신경 써야 합니다.**

그러면, 시간 복잡도는 차이가 있는지 분석해볼까요?

▼ 첫 번째 방법 - 시간 복잡도

```

for alphabet in alphabet_array:      # alphabet_array 의 길이(26)만큼 아래 연산이 실행
    occurrence = 0                    # 대입 연산 1번 실행
    for char in string:                # string 의 길이만큼 아래 연산이 실행
        if char == alphabet:          # 비교 연산 1번 실행
            occurrence += 1           # 대입 연산 1번 실행

    if occurrence > max_occurrence:    # 비교 연산 1번 실행
        max_alphabet = alphabet       # 대입 연산 1번 실행
        max_occurrence = number       # 대입 연산 1번 실행

```

- 위에서 연산된 것들을 더해보면,
 1. alphabet_array 의 길이 X 대입 연산 1번
 2. alphabet_array 의 길이 X string의 길이 X (비교 연산 1번 + 대입 연산 1번)
 3. alphabet_array 의 길이 X (비교 연산 1번 + 대입 연산 2번)

만큼의 시간이 필요합니다. 여기서 string 의 길이는 보통 N이라고 표현합니다. 그러면 위의 시간을 다음과 같이 표현할 수 있습니다.

$$26 * (1 + 2 * N + 3) = 52N + 104$$

그러면 우리는 이제 이 함수는 $52N + 104$ 만큼의 **시간이 걸렸겠구나!** 라고 말할 수 있습니다.

그런데, 이제 다른 상수 시간의 연산은 이제 계산하지 않아도 된다고 말씀드렸죠?

그러므로 N 만큼의 시간이 필요하다고 생각하시면 됩니다.

그럼 이제, 두 번째 방법은 얼마나 시간이 사용되었는지 보러 가보죠!

▼ 두 번째 방법 - 시간 복잡도

```
for char in string:          # string 의 길이만큼 아래 연산이 실행
    if not char.isalpha():    # 비교 연산 1번 실행
        continue
    arr_index = ord(char) - ord('a')    # 대입 연산 1번 실행
    alphabet_occurrence_list[arr_index] += 1 # 대입 연산 1번 실행

max_occurrence = 0          # 대입 연산 1번 실행
max_alphabet_index = 0      # 대입 연산 1번 실행
for index in range(len(alphabet_occurrence_list)):    # alphabet_array 의 길이(26)만큼 아래 연산이 실행
    alphabet_occurrence = alphabet_occurrence_list[index] # 대입 연산 1번 실행
    if alphabet_occurrence > max_occurrence: # 비교 연산 1번 실행
        max_occurrence = alphabet_occurrence # 대입 연산 1번 실행
        max_alphabet_index = index          # 대입 연산 1번 실행
```

- 위에서 연산된 것들을 더해보면,
 - string의 길이 X (비교 연산 1번 + 대입 연산 2번)
 - 대입 연산 2번
 - alphabet_array 의 길이 X (비교 연산 1번 + 대입 연산 3번)

만큼의 시간이 필요합니다. 여기서 string 의 길이는 보통 N 이라고 표현합니다. 그러면 위의 시간을 다음과 같이 표현할 수 있습니다.

$$N * (1 + 2) + 2 + 26 * (1 + 3) = 3N + 106$$

그러면 우리는 이제 이 함수는 $3N + 106$ 만큼의 **시간이 걸렸겠구나!** 라고 말할 수 있습니다.

그런데, 이제 다른 상수 시간의 연산은 이제 계산하지 않아도 된다고 말씀드렸죠?

그러므로 N 만큼의 시간이 필요하다고 생각하시면 됩니다.

▼ 비교하기

- 그러면, 이를 수학적으로 표현해보면 첫 번째 방법은 $52N + 104$, 두 번째 방법은 $3N + 106$ 이라는 식이 나온다는 걸 알 수 있습니다. 그러면 N 의 길이가 길어질수록, 다음과 같이 연산량이 변화합니다. 비교를 위해서 N^2 도 넣어봤습니다!

52N + 104 vs 3N+106 vs N^2

Aa N의 길이	# 52N + 104	# 3N + 106	# N^2
1	156	109	1
10	624	136	100
100	5304	406	10000
1000	52104	3106	1000000
10000	520104	30106	100000000

- 이 표를 보면, 두 가지를 깨달을 수 있습니다.

1. $52N + 104$ 든 $3N + 106$ 이든 N^2 에 비해서 아무것도 아니구나.
2. 공간 복잡도보다는 시간 복잡도를 더 신경 써야 한다!

07. 점근 표기법

▼ 10) 점근 표기법이란?



알고리즘의 성능을 수학적으로 표기하는 방법입니다. 알고리즘의 “효율성”을 평가하는 방법입니다. 점근 표기법 (asymptotic notation)은 어떤 함수의 증가 양상을 다른 함수와의 비교로 표현하는 수론과 해석학의 방법이다. 저희가 지금까지 이 함수는 N 정도의 시간과 공간이 걸리겠구나 하면서 분석했던 게 점근 표기법의 일종이라고 말할 수 있습니다.



점근 표기법의 종류에는
빅오(Big-O)표기법, 빅 오메가(Big-Ω) 표기법이 있습니다.

빅오 표기법은 최악의 성능이 나올 때 어느 정도의 연산량이 걸릴것인지,
빅오메가 표기법은 최선의 성능이 나올 때 어느 정도의 연산량이 걸릴것인지에 대해 표기합니다.

예를 들어

빅오 표기법으로 표시하면 $O(N)$,

빅 오메가 표기법으로 표시하면 $\Omega(1)$ 의 시간복잡도를 가진 알고리즘이다!

라고 표현할 수 있습니다.

한 번 문제를 풀면서 알아보도록 하겠습니다!

▼ 11) 🐞 배열에서 특정 요소 찾기

▼ Q. 문제 설명



Q. 다음과 같은 숫자로 이루어진 배열이 있을 때, 이 배열 내에 특정 숫자가 존재한다면 True, 존재하지 않다면 False 를 반환하시오.

```
[3, 5, 6, 1, 2, 4]
```

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! 2분 정도 고민해 본 다음, 아래 방법들을 펼쳐 봅시다!

▼ [코드스니펫] 배열에서 특정 요소 찾기

```
input = [3, 5, 6, 1, 2, 4]

def is_number_exist(number, array):
    # 이 부분을 채워보세요!
    return True

result = is_number_exist(3, input)
print(result)
```

▼ A. 해결 방법



배열을 돌면서 배열의 원소가 찾고자하는 숫자와 같은지 비교합니다. 만약 같다면 True 를 반환하고, 끝까지 없다면 False 를 반환합니다.

```
input = [3, 5, 6, 1, 2, 4]

def is_number_exist(number, array):
    for element in array:
        if number == element:
            return True
    return False

result = is_number_exist(3, input)
print(result)
```



그러면 위에서 배웠던 걸 한 번 적용해볼까요?!
이 함수의 시간 복잡도는 얼마나 걸릴까요?

```
for element in array:      # array 의 길이만큼 아래 연산이 실행
    if number == element:  # 비교 연산 1번 실행
        return True
return False
```



위에서 연산된 것들을 더해보면, 이렇게 총 $N * 1$ 의 시간 복잡도를 가진다는 걸 볼 수 있습니다.

그런데 여기서, 모든 경우에 N 만큼의 시간이 걸릴까요?



`input = [3, 5, 6, 1, 2, 4]` 이라는 입력값에 대해서 3을 찾으면,

첫 번째 원소를 비교하는 순간!! 바로 결과값을 반환하게 됩니다.
즉, 운이 좋으면 1번의 연산 이후에 답을 찾을 수 있다는 의미입니다.



그러나, `input = [4, 5, 6, 1, 2, 3]` 이라는 입력값에 대해서 3을 찾으면 어떻게 될까요?

마지막 원소 끝까지 찾다가 겨우 마지막에 3을 찾아 결과값을 반환하게 됩니다.
즉, 운이 좋지 않으면 input의 길이(N) 만큼 연산 이후에 답을 찾을 수 있습니다.



이처럼, 알고리즘의 성능은 항상 동일한 게 아니라 입력값에 따라서 달라질 수 있습니다.
어떤 값을 가지고 있는지, 어떤 패턴을 이루는 데이터인지에 따라서 뭐가 좋은 알고리즘인지 달라질 수 있다는 의미입니다.

- 이를 표로 표기하면 다음과 같습니다.

입력값에 따라 달라지는 연산량

Aa 입력값이	≡ 소요되는 연산량
좋은 때	1
안 좋은 때	N

👉 즉 위의 경우에는
빅오 표기법으로 표시하면 $O(N)$,
빅 오메가 표기법으로 표시하면 $\Omega(1)$ 의 시간복잡도를 가진 알고리즘이다!
라고 말할 수 있습니다.

❓ 그런데, 지금까지 우리는 왜 항상 모든 반복문이 돌 것이라고 생각하고 계산했을까요?
항상 최악의 경우, 빅오 표기법으로만 구한 거 아닌가요?

✅ 알고리즘에서는 거의 모든 알고리즘을 **빅오 표기법**으로 분석합니다.

왜냐면 대부분의 입력값이 최선의 경우일 가능성은 굉장히 적을 뿐더러,
우리는 최악의 경우를 대비해야 하기 때문입니다.

👏 여기까지 어려운 이론 배우느라 고생 많으셨습니다!!

우리는 다음만 기억하면 됩니다.

1. 입력값에 비례해서 얼마나 늘어날지 파악해보자. 1 ? N ? N^2 ?
2. 공간복잡도 보다는 시간 복잡도를 더 줄이기 위해 고민하자.
3. 최악의 경우에 시간이 얼마나 소요될지(빅오 표기법)에 대해 고민하자

08. 알고리즘 더 풀어보기 (1)

▼ 12) 🧐 곱하기 or 더하기

🔥 여러분 위에서 어려운 개념들을 배우시느라 고생 많으셨습니다.
이제부터는 문제를 풀어보자구요!

▼ Q. 문제 설명



Q. 다음과 같이 0 혹은 양의 정수로만 이루어진 배열이 있을 때, 왼쪽부터 오른쪽으로 하나씩 모든 숫자를 확인하며 숫자 사이에 'x' 혹은 '+' 연산자를 넣어 결과적으로 가장 큰 수를 구하는 프로그램을 작성하시오.

단, '+' 보다 'x' 를 먼저 계산하는 일반적인 방식과는 달리, 모든 연산은 왼쪽에서 순서대로 이루어진다.

[0, 3, 5, 6, 1, 2, 4]

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! 2분 정도 고민해 본 다음, 아래 방법들을 펼쳐 봅시다!

▼ [코드스니펫] 곱하기 or 더하기

```
input = [0, 3, 5, 6, 1, 2, 4]

def find_max_plus_or_multiply(array):
    # 이 부분을 채워보세요!
    return 1

result = find_max_plus_or_multiply(input)
print(result)
```

▼ A. 같이 풀어보기



곱하기 혹은 더하기를 해서 가장 큰 수를 반환해라! 라고 하면 $4 + 9 = 13$, $4 \times 9 = 36$ 이니까 당연히 \times 를 항상 해야 하는 거 아니야?!

라고 해서 모든 값을 \times 해버리면 최대의 값이 나오지 않습니다. 왜냐면 1 혹은 0 같은 경우는 곱하는 것보다 더하는 게 좋기 때문입니다!

$3 \times 1 = 3$ 보다, $3 + 1 = 4$ 를 하는 게 더 큰 수를 가질 수 있으니까요!

자, 그러면 이제 총 계산된 값을 구하기 위한 합계 변수를 두겠습니다.

이제 반복문을 돌면서 합계 변수에 더하거나 곱해 나갈텐데,

마찬가지로 합계가 1 이하일 때 더하는 게 좋습니다. (1×2 보다는 $1 + 2$ 가 더 크기 때문입니다!)

다시 정리하면, 합계와 현재 숫자가 1보다 작거나 같다면 더합니다. 그 외에는 전부 곱하면 되구요!

```
input = [0, 3, 5, 6, 1, 2, 4]

def find_max_plus_or_multiply(array):
    multiply_sum = 0
    for number in array:
        if number <= 1 or multiply_sum <= 1:
            multiply_sum += number
        else:
            multiply_sum *= number
    return multiply_sum

result = find_max_plus_or_multiply(input)
print(result)
```

▼ 시간 복잡도

? Q. 여기서 퀴즈, 이 함수의 시간 복잡도는 얼마나 걸릴까요?

✓ 바로 **O(N)** 만큼 걸립니다. 함수 구문 하나하나를 보지 않더라도, 1차 반복문이 나왔고, array 의 길이 만큼 반복한다? 그러면 O(N) 이겠구나! 생각해주시면 됩니다. 다른 계수는 다 버려버리자구요~!

09. 알고리즘 더 풀어보기 (2)

▼ 13) 🐞 반복되지 않는 문자

▼ Q. 문제 설명

? Q. 다음과 같이 영어로 되어 있는 문자열이 있을 때, 이 문자열에서 반복되지 않는 첫번째 문자를 반환하시오. 만약 그런 문자가 없다면 _ 를 반환하시오.

"abadaabac" # 반복되지 않는 문자는 d, c 가 있지만 "첫번째" 문자니까 d를 반환해주면 됩니다!

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! 2분 정도 고민해 본 다음, 아래 방법들을 펼쳐 보시다!

▼ [코드스니펫] 반복되지 않는 문자

```
input = "abadaabac"

def find_not_repeating_first_character(string):
    # 이 부분을 채워보세요!
    return "_"

result = find_not_repeating_first_character(input)
print(result)
```

▼ A. 같이 풀어보기

✓ 반복되는지 아닌지를 확인하기 위해서는, 각 알파벳 별로 출현하는 횟수를 저장해야 합니다. 그러면, 가장 많이 나온 알파벳 찾기에서 사용했던 방법을 다시 사용하면 됩니다!

alphabet_occurrence_array 배열에 각 빈도수를 저장하고, 그 배열을 돌면서 not_repeating_character_array 라는 배열에 반복되지 않는 문자를 다 집어넣습니다.

그리고 다시 한 번 문자열을 돌면서 해당 문자가 발견된다면 그 값을 반환하면 됩니다!

이 때, 1의 빈도수를 가진 인덱스를 알파벳으로 변환해서 not_repeating_character_array 에 저장하면 됩니다.

그러면 not_repeating_character_array 에는 ["c", "d"]가 담기게 되는데, 그 중 첫 번째인 "c" 를 반환하면 될까요? 그렇지 않습니다!

입력된 문자열 내에서 반복되지 않는 첫번째 문자를 찾아서 반환해줘야 하기 때문에 string 을 다시 반복해서 돌면서 첫 번째 반복되지 않는 문자를 찾아 반환해주시면 됩니다!

이를 이용해서 해결해봅시다!

```

input = "abadabac"

def find_not_repeating_first_character(string):
    alphabet_occurrence_array = [0] * 26

    for char in string:
        if not char.isalpha():
            continue
        arr_index = ord(char) - ord("a")
        alphabet_occurrence_array[arr_index] += 1

    not_repeating_character_array = []
    for index in range(len(alphabet_occurrence_array)):
        alphabet_occurrence = alphabet_occurrence_array[index]

        if alphabet_occurrence == 1:
            not_repeating_character_array.append(chr(index + ord("a")))


    for char in string:
        if char in not_repeating_character_array:
            return char


    return "_"

result = find_not_repeating_first_character(input)
print(result)


```

▼ 시간 복잡도


 Q. 여기서 퀴즈, 이 함수의 시간 복잡도는 얼마나 걸릴까요?

 이것도 바로 **O(N)** 만큼 걸립니다. 함수 구문 하나하나를 보지 않더라도, 1차 반복문이 나왔고, array 의 길이 만큼 반복한다? 그러면 O(N) 이겠구나! 생각해주시면 됩니다. 다른 계수는 다 버려버리자구요~!

10. 끝 & 숙제 설명

 문제 직접 풀어보기!

▼ Q1. 🐟 소수 나열하기

 Q. 정수를 입력 했을 때, 그 정수 이하의 소수를 모두 반환하시오.

소수는 자신보다 작은 두 개의 자연수를 곱하여 만들 수 없는 1보다 큰 자연수이다.

20이 입력된다면, 아래와 같이 반환해야 합니다!
[2, 3, 5, 7, 11, 13, 17, 19]

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! 2분 정도 고민해 본 다음, 아래 방법들을 펼쳐 봅시다!

▼ [코드스니펫] 소수 나열하기


```
input = 20

def find_prime_list_under_number(number):
    # 이 부분을 채워보세요!
    return []

result = find_prime_list_under_number(input)
print(result)
```

▼ Q2. 📌 문자열 뒤집기



Q.

0과 1로만 이루어진 문자열이 주어졌을 때, 이 문자열에 있는 모든 숫자를 전부 같게 만들려고 한다. 할 수 있는 행동은 문자열에서 연속된 하나 이상의 숫자를 잡고 모두 뒤집는 것이다. 뒤집는 것은 1을 0으로, 0을 1로 바꾸는 것을 의미한다.

예를 들어 S=0001100 일 때,

전체를 뒤집으면 1110011이 된다.

4번째 문자부터 5번째 문자까지 뒤집으면 1111111이 되어서 2번 만에 모두 같은 숫자로 만들 수 있다.

하지만, 처음부터 4번째 문자부터 5번째 문자까지 문자를 뒤집으면 한 번에 0000000이 되어서 1번 만에 모두 같은 숫자로 만들 수 있다.

주어진 문자열을 모두 0 혹은 모두 1로 같게 만드는 최소 횟수를 반환하시오.

```
"0001100"
```

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! 2분 정도 고민해 본 다음, 아래 방법들을 펼쳐 보시다!

▼ [코드스니펫] 문자열 뒤집기

```
input = "011110"

def find_count_to_turn_out_to_all_zero_or_all_one(string):
    # 이 부분을 채워보세요!
    return 1

result = find_count_to_turn_out_to_all_zero_or_all_one(input)
print(result)
```



알고리즘 문제를 풀다보면, 문제 자체를 이해하기 힘들 때가 있습니다.

그럴 때는 다음과 같이 해보세요!

1. 바로 코드를 작성하지 말고, 문제의 다른 예시들을 떠올리면서 규칙성을 생각해보세요.

ex) 00000 은 최소 횟수를 어떻게 구할까?

2. 배웠던 자료구조를 활용하면 어떨지 생각해보세요!

ex) (추후에 배울)스택, 큐를 활용하면 어떨까?

3. 문제의 특징들을 하나하나 글로 써보세요!

ex) 문자열을 뒤집어야 하는데, 0으로 할지 1으로 할지 고민 된다. 뒤집는 걸 감지할만한 시점은 0에서 1로, 1에서 0으로 바뀌는 시점인데, 초기에 0인지 1인지도 횟수에 연관이 있다.

이 과정을 통해 문제에 대해 더 깊게 파악한다면, 금방 풀이방법을 떠올릴 수 있을거예요!

▼ 🌞 문제가 이해가지 않는다면?

"0001100"



이 문자열을 모두 0 혹은 1로 만들기 위해서는 두가지 방법이 있습니다.

1. 모두 0으로 만드는 방법

1) 4번째 원소와 5번째 원소를 잡고 뒤집으면? 0000000 이 됩니다.

문자열을 순서대로 탐색하다보면

뒤집는 시점은 바로 0에서 1로 변할 때 뒤집어야 하는 걸 감지할 수 있습니다!

2. 모두 1으로 만드는 방법

1) 1번째 원소와 3번째 원소를 잡고 뒤집으면? 1111100 이 됩니다.

2) 6번째 원소와 7번째 원소를 잡고 뒤집으면? 1111111 이 됩니다.

문자열을 순서대로 탐색하다보면

뒤집는 시점은 1에서 0으로 변할 때 뒤집어야 하는 걸 감지할 수 있습니다.

즉, 모두 0으로 만드는 방법이 1회이므로 최소 횟수입니다!

Copyright © TeamSparta All rights reserved.

Copyright © TeamSparta All rights reserved.