[스파르타코딩클럽] 알고보면 알기쉬운 알고리즘 - 5주차

매 주차 강의자료 시작에 PDF파일을 올려두었어요!

▼ PDF 파일

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/7746b9bd-d346-4407-ad48-6457099cff20/____5_.pdf

[수업 목표]

1. 카카오, 삼성, 라인의 실전 문제를 풀면서 힌트를 찾자.
2. 카카오, 삼성, 라인의 실전 문제를 자세하게 이해해보자.
3. 카카오, 삼성, 라인의 실전 문제를 통해 성장하자.

[목차]

모든 토글을 열고 닫는 단축키
Windows : `Ctrl` + `alt` + `t`
Mac : `⌘` + `⌥` + `t`

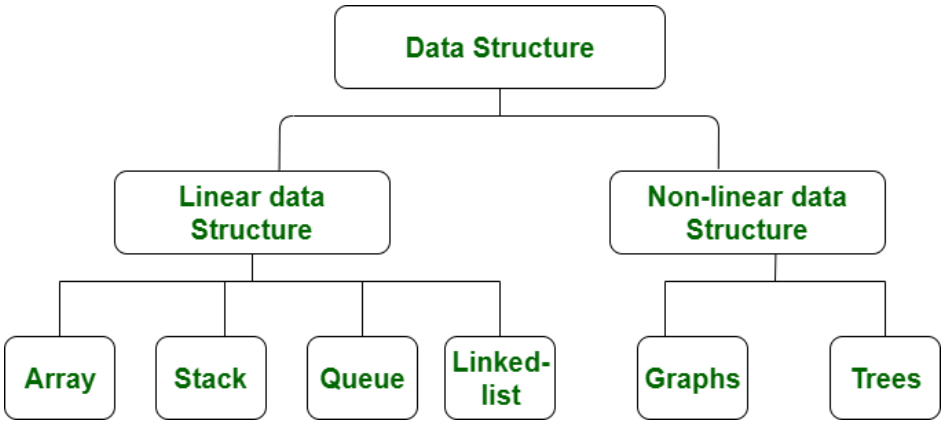
---

01. 오늘 배울 것

▼ 실전 문제 풀이에 앞서서 마음가짐 - 1

👉 실전 알고리즘 문제에서는 이 문제를 어떤 개념으로 풀 건지 알려주지 않고 문제를 냅니다!

즉, 문제를 읽으면서 이 문제는 어떤 자료 구조를 쓰는 게 좋을지,
어떤 알고리즘이 좋을지 고민하셔야 됩니다!
문제의 특성과 입력값, 출력값들을 보시면서 그 힌트들을 찾아가 봅시다!



▼ 실전 문제 풀이에 앞서서 마음가짐 - 2

👉 실전 알고리즘은 문제부터 이해하기 힘든 경우가 많습니다.
그래서 문제의 일부분씩 이해하시거나 예제를 써보시면서 이해하시는 게 좋습니다!
최소 10분~20분은 문제를 자세하게 이해하시고 풀어보시면 좋을 것 같습니다!

▼ 실전 문제 풀이에 앞서서 마음가짐 - 3

👉 앞으로 풀어 볼 문제들은 실제 기업에서 냈던 문제들입니다.
그만큼 어렵겠지만 그만큼 보람찰 것이라고 생각합니다.
물론 모르는 개념도, 처음 보는 풀이 방법도 자주 나올텐데
너무 좌절하지 마세요. 많이 배워서 실제 코딩테스트에서는 전부 풀어버리자구요!
그럼 시작하겠습니다!

02. 2019년 상반기 LINE 인턴 채용 코딩테스트

▼ 1) ✍️ 나 잡아 봐라

❓ Q. 연인 코니와 브라운은 광활한 들판에서 '나 잡아 봐라' 게임을 한다.
이 게임은 브라운이 코니를 잡거나, 코니가 너무 멀리 달아나면 끝난다.
게임이 끝나는데 걸리는 최소 시간을 구하시오.

조건은 다음과 같다.
코니는 처음 위치 C에서 1초 후 1만큼 움직이고,
이후에는 가속이 붙어 매 초마다 이전 이동 거리 + 1만큼 움직인다.
즉 시간에 따른 코니의 위치는 C, C + 1, C + 3, C + 6, ...이다.

브라운은 현재 위치 B에서 다음 순간 B – 1, B + 1, 2 * B 중 하나로 움직일 수 있다.
코니와 브라운의 위치 p는 조건 0 <= x <= 200,000을 만족한다.
브라운은 범위를 벗어나는 위치로는 이동할 수 없고, 코니가 범위를 벗어나면 게임이 끝난다

```
c = 11 # 코니의 처음 위치
b = 2  # 브라운의 처음 위치
# 이렇게 입력된다면
# 이 경우는 어떻게 늘어날지, 고민해보세요!
```

▼ 👏 여기서 잠깐! Tip

▼ Queue 를 실전에서 사용하려면?

여러분, 제가 큐를 파이썬에서 사용하기 위해서는 파이썬의 기본 자료형인 list() 를 사용하면 된다고 했었는데, **코딩테스트에서 큐를 사용할 때는**
collections.deque 를 사용하셔야 합니다.

성능 차이가 많이 나거든요..! 스택은 그대로 list 사용하셔도 됩니다.

따라서 앞으로는 큐가 필요할 때 다음과 같이 사용하겠습니다!

```
>>> from collections import deque
>>> queue = deque()
>>> queue.append(3)
>>> queue.append(4)
>>> print(queue.popleft())
3
```

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! **실전 문제처럼, 20분 이상을 고민해 본 다음**, 아래 방법들을 펼쳐 봅시다!

▼ **[코드스니펫] 나 잡아 봐라**

```
from collections import deque

c = 11
b = 2


def catch_me(cony_loc, brown_loc):
    # 구현해보세요!
    return


print(catch_me(c, b))  # 5가 나와야 합니다!
```

▼ 해결 방법

✅ 문제의 조건들을 다시 살펴보겠습니다.

우선, 코니의 위치 변화부터 파악해볼게요!
코니는 처음 위치에서 1초 후 1만큼, 매 초마다 이전 이동 거리 + 1만큼 움직입니다.
즉 증가하는 속도가 1초마다 1씩 계속 늘어난다는 소리겠군요!

그럼 이제 브라운의 위치 변화를 보면 B - 1, B + 1, 2 * B 입니다.

한 번, 예시에 따른 코니의 위치를 다 적어볼게요
  1 2 3 4
11 → 12 → 14 → 17 → 21 → .... 입니다!

그러면 브라운의 위치는 어떻게 변할까요?

1-1. 2-1 = 1
1-2. 2+1 = 3
1-3. 2×2 = 4

자 그러면, 이제 분기가 다시 나뉘어 집니다.
1-1인 1부터 다시 시작합니다.
1-1-1. 1-1 = 0
1-1-2. 1+1 = 2
1-1-3. 1×2 = 2

1-2인 3도 다시 위치를 반복해야 합니다.
1-2-1. 3-1 = 2
1-2-2. 3+1 = 4
1-2-3. 3×2 = 6
.....

이걸 보는 순간 바로 눈치 채셔야 합니다.
너무 경우의 수가 많을 것 같고, 쉽게 일반화 지어지지 않을 것 같은 문제를 봤다면
모든 경우의 수를 다 나열해야겠구나
즉, **BFS** 를 사용하는 문제겠구나 생각해주셔야 합니다.

그리고 여기 또 중요한 점이 하나 있습니다.
잡았다는 의미는 똑같은 시간에 똑같은 위치에 있어야만 잡은 것입니다.
그러므로 시간과 위치를 저장해놓는 자료구조가 필요합니다.

시간은 +1 씩 규칙적으로 증가하지만,
위치는 코니도 브라운도 그렇고 값이 자유자재로 막 변화합니다.

규칙적으로 증가하는 자료구조는 배열,
자유자재로 저장하기 위한 자료구조는 딕셔너리! 를 사용하면 되겠죠?

그러면 저는 각 시간마다 위치를 저장하기 위한 저장 공간으로
배열안에 딕셔너리를 넣겠습니다!

이 부분은 조금 이따가 다시 설명 드리겠습니다.

자, 그러면 이제 코니와 브라운의 위치를 구하러 가보겠습니다!

```
from collections import deque

c = 11
b = 2


def catch_me(cony_loc, brown_loc):
    time = 0

    while 1:
        cony_loc += time

        # ???? 이 셋 중 뭘로 해야 할까요?
        new_position = brown_loc - 1
        new_position = brown_loc + 1
        new_position = brown_loc * 2

        time += 1


print(catch_me(c, b))
```

✅ 코니의 위치는
저렇게 늘어나는 시간을 규칙적으로 더해주면 구할 수 있을 것 같은데,

브라운의 위치는 너무나 다양합니다!
그러면 모든 경우의 수를 다 구해야 하는데,
모든 경우의 수를 구하기 위해서는? BFS 였죠!

그러면, 경우의 수를 쌓아 놓기 위한 queue 도 하나 만들겠습니다.

이 때! 위치와 함께 시간을 담아주셔야 합니다.
저희가 원하는 것은 위치와 시간이 동일해야 하거든요!

그래서 다음과 같이 새로운 포지션과 위치를 queue 에 담아줍니다.

```
from collections import deque

c = 11
b = 2


def catch_me(cony_loc, brown_loc):
    time = 0
    queue = deque()
    queue.append((brown_loc, 0))  # 위치와 시간을 담아줄게요!

    while 1:
        cony_loc += time

        for i in range(0, len(queue)):  # Q. Queue 인데 while 을 안 쓰는 이유를 고민해보세요!
            current_position, current_time = queue.popleft()

            new_time = current_time + 1
            new_position = current_position - 1
            queue.append((new_position, new_time))

            new_position = current_position + 1
            queue.append((new_position, new_time))
```

```
            new_position = current_position * 2
            queue.append((new_position, new_time))

        time += 1


print(catch_me(c, b))
```

✅ 그런데, 이걸 무제한으로 하나요?
탈출 조건이 없죠!
실패 조건은 다음과 같았습니다.

1. 코니와 브라운의 위치 p는 조건 0 <= x <= 200,000을 만족한다.
→ 따라서 while 문에 cony_loc 의 조건을 추가해줍니다.
2. 브라운은 범위를 벗어나는 위치로는 이동할 수 없고, 코니가 범위를 벗어나면 게임이 끝난다
→ 브라운의 new_position 에 조건을 추가합니다.

그러면, 성공하는 조건은 뭔가요?
코니와 브라운의 위치와 시간이 동일해야 합니다!
그러면 방문한 시간과 위치를 저장하기 위해 visited 라는 배열을 만들겠습니다!

위치마다 브라운이 방문한 시간을 적기 위해 배열 안에 딕셔너리를 넣습니다.
코니는 값이 정해져 있어서 따로 저장할 필요가 없습니다!

예를 들어,
5 in visited[3] 는 5초에 위치 3을 방문했는지 여부를 저장하고,
9 in visited[3] 는 9초에 위치 3을 방문했는지 여부를 저장합니다.

아까 브라운의 위치로 설명을 드려보면,

1. 0초에 갈 수 있는 위치는
2 이므로 visited[2] = { 0: True } 가 됩니다.

2. 1초에 갈 수 있는 위치는
1, 3, 4 이므로
visited[1] = { 1: True }
visited[3] = { 1: True }
visited[4] = { 1: True } 가 됩니다.

3. 2초에 갈 수 있는 위치는
0, 2, 3, 4, 8 이므로
visited[0] = { 2: True }
visited[2] = { 0: True, 2: True }
visited[3] = { 1: True, 2: True}
visited[4] = { 1: True, 2: True}
visited[8] = { 2: True }

그러면 이제 visited[cony 의 위치] 에 현재 시간이 있다?
즉, 브라운이 현재 코니가 방문한 위치에 현재 시간에 들를 수 있다?
그러면 True 를 반환해주시면 됩니다.

```python
from collections import deque

c = 11
b = 2


def catch_me(cony_loc, brown_loc):
    time = 0
    queue = deque()
    queue.append((brown_loc, 0))  # 위치와 시간을 담아줄게요!.
    visited = [{} for _ in range(200001)]

    while cony_loc < 200000:
        cony_loc += time
        if time in visited[cony_loc]:
            return time

        for i in range(0, len(queue)):  # Q. Queue 인데 while 을 안 쓰는 이유를 고민해보세요!
            current_position, current_time = queue.popleft()

            new_position = current_position - 1
            new_time = current_time + 1
            if new_position >= 0 and new_time not in visited[new_position]:
                visited[new_position][new_time] = True
                queue.append((new_position, new_time))

            new_position = current_position + 1
            if new_position < 200001 and new_time not in visited[new_position]:
                visited[new_position][new_time] = True
                queue.append((new_position, new_time))

            new_position = current_position * 2
            if new_position < 200001 and new_time not in visited[new_position]:
                visited[new_position][new_time] = True
                queue.append((new_position, new_time))

        time += 1


print(catch_me(c, b))
```

👍 자 이렇게 문제를 풀어봤습니다!
어떠신가요?

우리가 배운 모든 개념들이 섞여서 나옵니다.
여러분들은 이렇게 문제들의 특징들을 보면서 하나하나 찾아가실 수 있어야 합니다!

수고 많으셨습니다!

03. 2020 카카오 신입 개발자 블라인드 채용 1차 코딩테스트 - 1

▼ 2) ✍️ 문자열 압축

❓ Q. 데이터 처리 전문가가 되고 싶은 어피치는 문자열을 압축하는 방법에 대해 공부를 하고 있습니다.

최근에 대량의 데이터 처리를 위한 간단한 비손실 압축 방법에 대해 공부를 하고 있는데, 문자열에서 같은 값이 연속해서 나타나는 것을 그 문자의 개수와 반복되는 값으로 표현하여 더 짧은 문자열로 줄여서 표현하는 알고리즘을 공부하고 있습니다.

간단한 예로 aabbaccc의 경우 2a2ba3c(문자가 반복되지 않아 한번만 나타난 경우 1은 생략함)와 같이 표현할 수 있는데, 이러한 방식은 반복되는 문자가 적은 경우 압축률이 낮다는 단점이 있습니다. 예를 들면, abcabcdede와 같은 문자열은 전혀 압축되지 않습니다. 어피치는 이러한 단점을 해결하기 위해 문자열을 1개 이상의 단위로 잘라서 압축하여 더 짧은 문자열로 표현할 수 있는지 방법을 찾아보려고 합니다.

예를 들어, ababcdcdababcdcd의 경우 문자를 1개 단위로 자르면 전혀 압축되지 않지만, 2개 단위로 잘라서 압축한다면 2ab2cd2ab2cd로 표현할 수 있습니다. 다른 방법으로 8개 단위로 잘라서 압축한다면 2ababcdcd로 표현할 수 있으며, 이때가 가장 짧게 압축하여 표현할 수 있는 방법입니다.

다른 예로, abcabcdede와 같은 경우, 문자를 2개 단위로 잘라서 압축하면 abcabc2de가 되지만, 3개 단위로 자른다면 2abcdede가 되어 3개 단위가 가장 짧은 압축 방법이 됩니다. 이때 3개 단위로 자르고 마지막에 남는 문자열은 그대로 붙여주면 됩니다.

압축할 문자열 input이 매개변수로 주어질 때, 위에 설명한 방법으로 1개 이상 단위로 문자열을 잘라 압축하여 표현한 문자열 중 가장 짧은 것의 길이를 return 하도록 string_compression 함수를 완성해주세요.

* 문자열의 길이는 1 이상 1,000 이하입니다.
* 문자열은 알파벳 소문자로만 이루어져 있습니다.

이 때, 문자열은 항상 제일 앞부터 정해진 길이만큼 잘라야 합니다.
입출력 예 #5 처럼 xababcdcdababcdcd 이 입력되어도,
문자열을 x / ababcdcd / ababcdcd 로 자르는 것은 **불가능합니다.**
이 경우 어떻게 문자열을 잘라도 압축되지 않으므로 가장 짧은 길이는 17이 됩니다.

```
"aabbaccc"  # -> 7
"ababcdcdababcdcd"  # -> 9
"abcabcdede"  # -> 8
"abcabcabcabcdededededede"  # -> 14
"xababcdcdababcdcd" # -> 17
```

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! **실전 문제처럼, 20분 이상을 고민해 본 다음**, 아래 방법들을 펼쳐 봅시다!

▼ [코드스니펫] 문자열 압축

```
input = "abcabcabcabcdededededede"


def string_compression(string):
    return


print(string_compression(input))  # 14 가 출력되어야 합니다!
```

▼ 해결 방법

✅ 문자를 과연 몇번이나 압축해야 할까요?

길이를 1로 잘라야 할지 2로 잘라야할지 3으로 잘라야할지
한 번에 분석하는 방법이 가능할까요?

어떻게 패턴을 분석하는 게 좋을까요?
뭔가 반복되는 구조를 보기 위해서는 스택을 쓰면 좋겠다고 생각할 수 도 있고,
재귀함수를 떠오르시는 분도 있을 것 같습니다.

그러나, 이 경우는 모든 경우에서 가장 압축을 많이 시킨 문자열의 길이를 반환해야 합니다.
즉, 모든 경우를 다 봐야 한다는 의미이므로 모든 경우의 수를 파악해야 합니다!

문자열의 길이를 n 이라고 한다면,
쪼개는 방법은 1~n 까지 쪼갤 수 있습니다!
그러나 n//2 이상의 길이 부터는, 쪼개도 아무런 이득이 없습니다!
따라서 1부터 n//2 까지만 쪼개보겠습니다!

쪼개기 위해서는 저번에 배웠던 문자열 슬라이싱을 사용하면 됩니다!
split_size 의 크기만큼 0부터 n까지 반복하면 되겠죠!

```
input = "abcabcabcabcdedededededede"


def string_compression(string):
    n = len(string)
    for split_size in range(1, n // 2 + 1):
        splited = [
            string[i:i + split_size] for i in range(0, n, split_size)
        ]
        print(splited)


print(string_compression(input))  # 14 가 출력되어야 합니다!
```

✅ 자 그러면, 이제 이 쪼개진 걸 압축할 수 있는지 아닌지 여부를 알아내야 합니다.

반복을 통해 앞의 문자열과 현재 문자열이 동일한지 확인합니다!
동일하다면 count 를 계속해서 늘려나가고,
다른 순간이 오면 compressed 라는 변수에 추가합니다!
만약 count 가 1이라면 1을 생략하고 더하셔야 합니다!

그리고 반복문이 끝나더라도! 마지막에 남은 꼬다리를 추가해주셔야 됩니다.
그렇게 나온 값들 중에서 가장 최솟값을 반환하시면 됩니다!

```
input = "abcabcabcabcdedededededede"


def string_compression(string):
    n = len(string)
    compression_length_array = []  # 1~len까지 압축했을때 길이 값

    for split_size in range(1, n // 2 + 1):
        compressed = ""
        # string 갯수 단위로 쪼개기 *
        splited = [
```

```
            string[i:i + split_size] for i in range(0, n, split_size)
        ]
        count = 1

        for j in range(1, len(splited)):
            prev, cur = splited[j - 1], splited[j]
            if prev == cur:
                count += 1
            else:  # 이전 문자와 다르다면
                if count > 1:
                    compressed += (str(count) + prev)
                else:  # 문자가 반복되지 않아 한번만 나타난 경우 1은 생략함
                    compressed += prev
                count = 1  # 초기화
        if count > 1:
            compressed += (str(count) + splited[-1])
        else:  # 문자가 반복되지 않아 한번만 나타난 경우 1은 생략함
            compressed += splited[-1]
        compression_length_array.append(len(compressed))

    return min(compression_length_array)  # 최솟값 리턴


print(string_compression(input))  # 14 가 출력되어야 합니다!
```

04. 2020 카카오 신입 개발자 블라인드 채용 1차 코딩테스트 - 2

▼ 3) ✍️ 올바른 괄호 문자열 만들기

문제 설명
카카오에 신입 개발자로 입사한 콘은 선배 개발자로부터 개발역량 강화를 위해 다른 개발자가 작성한 소스 코드를 분석하여 문제점을 발견하고 수정하라는 업무 과제를 받았습니다. 소스를 컴파일하여 로그를 보니 대부분 소스 코드 내 작성된 괄호가 개수는 맞지만 짝이 맞지 않은 형태로 작성되어 오류가 나는 것을 알게 되었습니다.
수정해야 할 소스 파일이 너무 많아서 고민하던 콘은 소스 코드에 작성된 모든 괄호를 뽑아서 올바른 순서대로 배치된 괄호 문자열을 알려주는 프로그램을 다음과 같이 개발하려고 합니다.

용어의 정의
'(' 와 ')' 로만 이루어진 문자열이 있을 경우, '(' 의 개수와 ')' 의 개수가 같다면 이를 균형잡힌 괄호 문자열이라고 부릅니다.
그리고 여기에 '('와 ')'의 괄호의 짝도 모두 맞을 경우에는 이를 올바른 괄호 문자열이라고 부릅니다.
예를 들어, "(()))("와 같은 문자열은 균형잡힌 괄호 문자열 이지만 올바른 괄호 문자열은 아닙니다.
반면에 "(())()"와 같은 문자열은 균형잡힌 괄호 문자열 이면서 동시에 올바른 괄호 문자열 입니다.

'(' 와 ')' 로만 이루어진 문자열 w가 균형잡힌 괄호 문자열 이라면 다음과 같은 과정을 통해 올바른 괄호 문자열로 변환할 수 있습니다.

1. 입력이 빈 문자열인 경우, 빈 문자열을 반환합니다.
2. 문자열 w를 두 "균형잡힌 괄호 문자열" u, v로 분리합니다. 단, u는 "균형잡힌 괄호 문자열"로 더 이상 분리할 수 없어야 하며, v는 빈 문자열이 될 수 있습니다.
3. 문자열 u가 "올바른 괄호 문자열" 이라면 문자열 v에 대해 1단계부터 다시 수행합니다.
  3-1. 수행한 결과 문자열을 u에 이어 붙인 후 반환합니다.
4. 문자열 u가 "올바른 괄호 문자열"이 아니라면 아래 과정을 수행합니다.
  4-1. 빈 문자열에 첫 번째 문자로 '('를 붙입니다.
  4-2. 문자열 v에 대해 1단계부터 재귀적으로 수행한 결과 문자열을 이어 붙입니다.
  4-3. ')'를 다시 붙입니다.
  4-4. u의 첫 번째와 마지막 문자를 제거하고, 나머지 문자열의 괄호 방향을 뒤집어서 뒤에 붙입니다.
  4-5. 생성된 문자열을 반환합니다.

균형잡힌 괄호 문자열 p가 매개변수로 주어질 때, 주어진 알고리즘을 수행해 올바른 괄호 문자열로 변환한 결과를 반환하시오.

```
"(()())()"  # -> "(()())()"
")("        # -> "()"
"(()))(()"  # -> "(()(()))"
```

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! **실전 문제처럼, 20분 이상을 고민해 본 다음**, 아래 방법들을 펼쳐 봅시다!

▼ [코드스니펫] 올바른 괄호 문자열 만들기

```
from collections import deque

balanced_parentheses_string = "(()))(()"


def get_correct_parentheses(balanced_parentheses_string):
    return


print(get_correct_parentheses(balanced_parentheses_string))  # "()(()))()"가 반환 되어야 합니다!
```

▼ 해결 방법

✅ 이 문자는 해결 방법이 다 나와있습니다!
이 해결 방법들을 따라서 구현만 하면 됩니다!

우선, 균형잡힌 괄호 문자열을 올바른 괄호 문자열로 만들기 전에,
어떻게 하면 올바른 괄호 문자열인지 알 수 있었죠?

바로, 예전에 우리가 풀었던 방법 그대로 큐를 사용하시면 됩니다!
이번에는 deque 를 이용해서 구현해봅시다!

```
from collections import deque

balanced_parentheses_string = "()))((()"


def is_correct_parentheses(string):  # 올바른 괄호인지 확인
    stack = []
    for s in string:
        if s == '(':
            stack.append(s)
        elif stack:
            stack.pop()
    return len(stack) == 0


def get_correct_parentheses(balanced_parentheses_string):
    if is_correct_parentheses(balanced_parentheses_string):
        return balanced_parentheses_string
    else:
        return "What?"


print(get_correct_parentheses(balanced_parentheses_string))  # "()(())()"가 반환 되어야 합니다!
```

✅ 자 이렇게, 올바른 괄호 문자열인지 확인하는 함수를 추가했습니다.

그러면, 이제 올바른 괄호 문자열이 아닐 때!
즉, 균형잡힌 괄호 문자열일 때 올바른 괄호 문자열로 바꾸는 방법을
하나하나 구현해보겠습니다.

함수의 이름은 change_to_correct_parentheses 로 하겠습니다.

1. 입력이 빈 문자열인 경우, 빈 문자열을 반환합니다.
→ if string == '' 이라면 빈 문자열을 반환해주면 됩니다.

```
...
def change_to_correct_parentheses(string):
    if string == '':  # 1번
        return ''

def get_correct_parentheses(balanced_parentheses_string):
    if is_correct_parentheses(balanced_parentheses_string):
        return balanced_parentheses_string
    else:
        return change_to_correct_parentheses(balanced_parentheses_string)

...
```

✅ **2. 문자열 w를 두 "균형잡힌 괄호 문자열" u, v로 분리합니다. 단, u는 "균형잡힌 괄호 문자.열"로 더 이상 분리할 수 없어야 하며, v는 빈 문자열이 될 수 있습니다.**
→ 문자열 w를 두 균형잡힌 괄호 문자열로 분리하려면 어떻게 해야 할까요?
균형잡힌 괄호 문자열의 조건은 (와)의 개수가 같아야 합니다!

따라서 w를 큐에 담고, 하나씩 꺼내면서 (와) 의 개수가 같을 때까지 u 에 추가합니다!
여기서 왜 (와)의 개수가 같을 때까지 u에 추가하냐면,
u는 균형잡힌 괄호 문자열로 더 이상 분리할 수 없어야 하기 때문에
더 이상의 괄호 쌍을 넣지 않기 위함이라고 생각해주시면 됩니다.
이제 나머지 v 는 w 가 애초에 ( ) 개수가 맞는 문자열이기 때문에
v 는 자연스레 균형잡힌 괄호 문자열이 됩니다.

```
...
def separate_to_u_v(string):  # u, v로 분리
    queue = deque(string)
    left, right = 0, 0
    u, v = "", ""

    while queue:  # 큐사용
        char = queue.popleft()
        u += char
        if char == '(':
            left += 1
        else:
            right += 1
        if left == right:  # 단, u는 "균형잡힌 괄호 문자열"로 더 이상 분리할 수 없어야 합니다. 즉, 여기서 괄 쌍이 더 생기면 안됩니다.
            break

    v = ''.join(list(queue))
    return u, v

def change_to_correct_parentheses(string):
    if string == '':  # 1번
        return ''
    u, v = separate_to_u_v(string)  # 2번
...
```

✅ **3. 문자열 u가 "올바른 괄호 문자열" 이라면 문자열 v에 대해 1단계부터 다시 수행합니다.**
→ change_to_correct_parentheses 함수로 돌아와서, 조건문과 재귀 함수를 추가합니다.

3-1. 수행한 결과 문자열을 u에 이어 붙인 후 반환합니다.
→ return u + change_to_correct_parentheses(v)

```
def change_to_correct_parentheses(string):
    if string == '':  # 1번
        return ''
    u, v = separate_to_u_v(string)  # 2번
    if is_correct_parentheses(u):  # 3번
        return u + change_to_correct_parentheses(v)
...
```

✅ **4. 문자열 u가 "올바른 괄호 문자열"이 아니라면 아래 과정을 수행합니다.**
4-1. 빈 문자열에 첫 번째 문자로 '('를 붙입니다.
4-2. 문자열 v에 대해 1단계부터 재귀적으로 수행한 결과 문자열을 이어 붙입니다.
4-3. ')'를 다시 붙입니다.
4-4. u의 첫 번째와 마지막 문자를 제거하고, 나머지 문자열의 괄호 방향을 뒤집어서 뒤에 붙입니다.
4-5. 생성된 문자열을 반환합니다.
→ return '(' + change_to_correct_parentheses(v) + ')' + reverse_parentheses(u[1:-1])

```python
from collections import deque

balanced_parentheses_string = "()))((()"


def is_correct_parentheses(string):  # 올바른 괄호인지 확인
    stack = []
    for s in string:
        if s == '(':
            stack.append(s)
        elif stack:
            stack.pop()
    return len(stack) == 0


def separate_to_u_v(string):  # u, v로 분리
    queue = deque(string)
    left, right = 0, 0
    u, v = "", ""

    while queue:  # 큐사용
        char = queue.popleft()
        u += char
        if char == '(':
            left += 1
        else:
            right += 1
        if left == right:  # 단, u는 "균형잡힌 괄호 문자열"로 더 이상 분리할 수 없어야 합니다. 즉, 여기서 괄 쌍이 더 생기면 안됩니다.
            break

    v = ''.join(list(queue))
    return u, v


def reverse_parentheses(string):  # 뒤집기
    reversed_string = ""
    for char in string:
        if char == '(':
            reversed_string += ")"
        else:
            reversed_string += "("
    return reversed_string


def change_to_correct_parentheses(string):
    if string == '':  # 1번
        return ''
    u, v = separate_to_u_v(string)  # 2번
    if is_correct_parentheses(u):  # 3번
        return u + change_to_correct_parentheses(v)
    else:  # 4번
        return '(' + change_to_correct_parentheses(v) + ')' + reverse_parentheses(u[1:-1])


def get_correct_parentheses(balanced_parentheses_string):
    if is_correct_parentheses(balanced_parentheses_string):
        return balanced_parentheses_string
    else:
        return change_to_correct_parentheses(balanced_parentheses_string)


print(get_correct_parentheses(balanced_parentheses_string))  # "()(())()"가 반환 되어야 합니다!
```

👍 자 이렇게 문제를 풀어봤습니다!
어떠신가요?

문제에서 직접 구현하는 방법을 알려주고 풀도록 하는 방법도 있습니다.
조건에 맞게 잘 구현만 하시면 됩니다!

수고 많으셨습니다!

05. 삼성 역량 테스트 - 1

▼ 4) ✍️ 새로운 게임 2

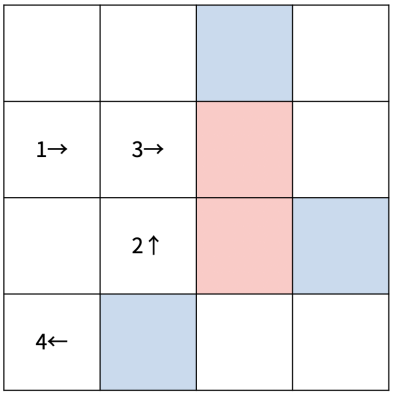
❓ Q. 재현이는 주변을 살펴보던 중 체스판과 말을 이용해서 새로운 게임을 만들기로 했다.
새로운 게임은 크기가 N×N인 체스판에서 진행되고, 사용하는 말의 개수는 K개이다.
말은 원판모양이고, 하나의 말 위에 다른 말을 올릴 수 있다.
체스판의 각 칸은 흰색, 빨간색, 파란색 중 하나로 색칠되어있다.

게임은 체스판 위에 말 K개를 놓고 시작한다. 말은 1번부터 K번까지 번호가 매겨져 있고, 이동 방향도 미리 정해져 있다. 이동 방향은 위, 아래, 왼쪽, 오른쪽 4가지 중 하나이다.

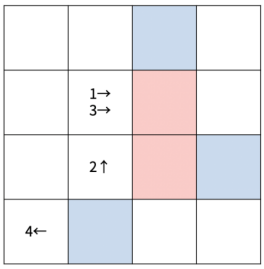
턴 한 번은 1번 말부터 K번 말까지 순서대로 이동시키는 것이다. 한 말이 이동할 때 위에 올려져 있는 말도 함께 이동한다. 말의 이동 방향에 있는 칸에 따라서 말의 이동이 다르며 아래와 같다. 턴이 진행되던 중에 말이 4개 이상 쌓이는 순간 게임이 종료된다.

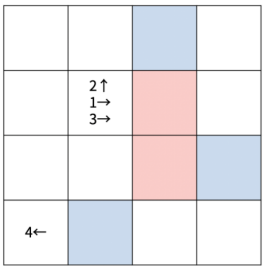
1. A번 말이 이동하려는 칸이
   1) 흰색인 경우에는 그 칸으로 이동한다. 이동하려는 칸에 말이 이미 있는 경우에는 가장 위에 A번 말을 올려놓는다.
      - A번 말의 위에 다른 말이 있는 경우에는 A번 말과 위에 있는 모든 말이 이동한다.
      - 예를 들어, A, B, C로 쌓여있고, 이동하려는 칸에 D, E가 있는 경우에는 A번 말이 이동한 후에는 D, E, A, B, C가 된다.
   2) 빨간색인 경우에는 이동한 후에 A번 말과 그 위에 있는 모든 말의 쌓여있는 순서를 반대로 바꾼다.
      - A, B, C가 이동하고, 이동하려는 칸에 말이 없는 경우에는 C, B, A가 된다.
      - A, D, F, G가 이동하고, 이동하려는 칸에 말이 E, C, B로 있는 경우에는 E, C, B, G, F, D, A가 된다.
   3) 파란색인 경우에는 A번 말의 이동 방향을 반대로 하고 한 칸 이동한다. 방향을 반대로 바꾼 후에 이동하려는 칸이 파란색인 경우에는 이동하지 않고 가만히 있는다.
   4) 체스판을 벗어나는 경우에는 파란색과 같은 경우이다.

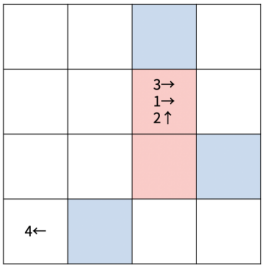
다음은 크기가 4×4인 체스판 위에 말이 4개 있는 경우이다.

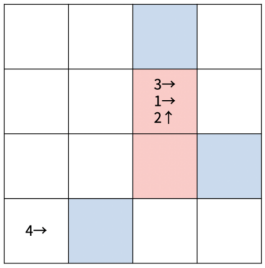


👉 첫 번째 턴은 다음과 같이 진행된다.

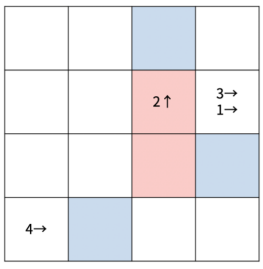


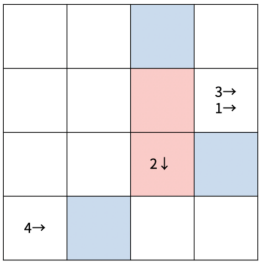


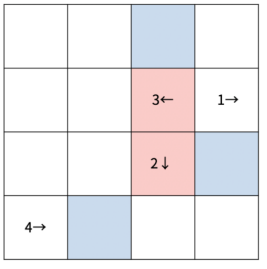


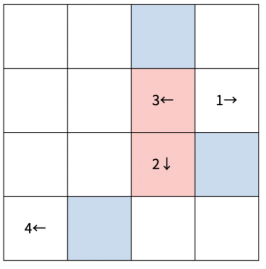


👉 두 번째 턴은 다음과 같이 진행된다.









❓ 체스판의 크기와 말의 위치, 이동 방향이 모두 주어졌을 때,
게임이 종료되는 턴의 번호를 반환하시오.

그 값이 1,000보다 크거나 절대로 게임이 종료되지 않는 경우에는 -1을 반환한다.

입력
각 정수는 칸의 색을 의미한다. 0은 흰색, 1은 빨간색, 2는 파란색이다.
말의 개수와 체스판의 정보, 현재 말의 위치와 방향을 주어진다.
말의 정보는 세 개의 정수로 이루어져 있고,
순서대로 행, 열의 인덱스, 이동 방향이다.
행과 열의 번호는 0부터 시작하고, 이동 방향은 0, 1, 2, 3 이고
0부터 순서대로 →, ←, ↑, ↓의 의미를 갖는다.

```
k = 4  # 말의 개수

chess_map = [
    [0, 0, 2, 0],
    [0, 0, 1, 0],
    [0, 0, 1, 2],
    [0, 2, 0, 0]
]
start_horse_location_and_directions = [
    [1, 0, 0],
    [2, 1, 2],
    [1, 1, 0],
    [3, 0, 1]
]
# 이 경우는 게임이 끝나지 않아 -1 을 반환해야 합니다!
```

```
K = 4  # 말의 개수

chess_map = [
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0]
]
start_horse_location_and_directions = [
    [0, 0, 0],
    [0, 1, 0],
    [0, 2, 0],
    [2, 2, 2]
]
# 이 경우는 2 을 반환해야 합니다!
```

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! **실전 문제처럼, 20분 이상을 고민해 본 다음**, 아래 방법들을 펼쳐 봅시다!

▼ [코드스니펫] 새로운 게임

```
k = 4  # 말의 개수

chess_map = [
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0]
]
start_horse_location_and_directions = [
    [0, 0, 0],
    [0, 1, 0],
    [0, 2, 0],
    [2, 2, 2]
```

```
]
# 이 경우는 게임이 끝나지 않아 -1 을 반환해야 합니다!
# 동 서 북 남
# →, ←, ↑, ↓
dr = [0, 0, -1, 1]
dy = [1, -1, 0, 0]


def get_game_over_turn_count(horse_count, game_map, horse_location_and_directions):
    return


print(get_game_over_turn_count(k, chess_map, start_horse_location_and_directions))  # 2가 반환 되어야합니다
```

▼ 해결 방법

✅ 우선 이 문제를 잘 이해하는 게 중요합니다.

말은 순서대로 이동합니다 → 말의 순서에 따라 **반복문**
말이 쌓일 수 있습니다 → **맵에 말이 쌓이는 걸** 저장해놔야 함.
쌓인 순서대로 이동함 → **스택을** 써야겠구나!

자 우선, 현재 맵에 어떻게 말이 쌓일지를 저장하기 위해서는,
chess_map 이랑 동일하게 만들되, 각 원소에 리스트를 저장해둬야 합니다!

그래서 이를 저장할 변수인
current_stacked_horse_map 을 만들고, 이를 초기화 해두겠습니다!
이제 각 원소가 링크드 리스트이므로,
이 링크드 리스트에 현재 위치한 말들의 위치를 추가해줘야겠죠?

아래와 같이 하면 다음과 같이
current_stacked_horse_map 가 설정됩니다.
[
  [[0], [1], [2], []],
  [[], [], [], []],
  [[], [], [3], []],
  [[], [], [], []]
]

```
k = 4  # 말의 개수

chess_map = [
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0]
]
start_horse_location_and_directions = [
    [0, 0, 0],
    [0, 1, 0],
    [0, 2, 0],
    [2, 2, 2]
]
# 이 경우는 게임이 끝나지 않아 -1 을 반환해야 합니다!
# 동 서 북 남
# →, ←, ↑, ↓
dr = [0, 0, -1, 1]
dc = [1, -1, 0, 0]


def get_game_over_turn_count(horse_count, game_map, horse_location_and_directions):
    n = len(game_map)
```

```
    current_stacked_horse_map = [[[] for _ in range(n)] for _ in range(n)]
    for i in range(horse_count):
        r, c, d = horse_location_and_directions[i]
        current_stacked_horse_map[r][c].append(i)

    return -1


print(get_game_over_turn_count(k, chess_map, start_horse_location_and_directions))  # 2가 반환 되어야합니다
```

✅ 그럼 이제, 각 말들을 규칙에 맞게 움직여보겠습니다!

그 전에, 이 프로그램은 게임이 종료되는 턴의 번호를 반환해야 합니다.
턴의 최대 횟수는 1000입니다. 해당 조건을 반복문에 써줍니다.

그러면 이제 게임을 시작해봐야겠죠!

각 턴이 지날때마다,
말은 자신의 방향에 맞게 위치를 변경합니다!

```
k = 4  # 말의 개수

chess_map = [
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0]
]
start_horse_location_and_directions = [
    [0, 0, 0],
    [0, 1, 0],
    [0, 2, 0],
    [2, 2, 2]
]
# 이 경우는 게임이 끝나지 않아 -1 을 반환해야 합니다!
# 동 서 북 남
# →, ←, ↑, ↓
dr = [0, 0, -1, 1]
dc = [1, -1, 0, 0]


def get_game_over_turn_count(horse_count, game_map, horse_location_and_directions):
    n = len(game_map)
    turn_count = 1
    current_stacked_horse_map = [[[] for _ in range(n)] for _ in range(n)]
    for i in range(horse_count):
        r, c, d = horse_location_and_directions[i]
        current_stacked_horse_map[r][c].append(i)

    while turn_count <= 1000:
        for horse_index in range(horse_count):
            n = len(game_map)
            r, c, d = horse_location_and_directions[horse_index]
            new_r = r + dr[d]
            new_c = c + dc[d]

        turn_count += 1
    return -1


print(get_game_over_turn_count(k, chess_map, start_horse_location_and_directions))  # 2가 반환 되어야합니다
```

✅ 이동할 때 규칙이 있었죠?

한 말이 이동할 때 위에 올려져 있는 말도 함께 이동한다.
1. A번 말이 이동하려는 칸이
1) 흰색인 경우에는 그 칸으로 이동한다. 이동하려는 칸에 말이 이미 있는 경우에는 가장 위에 A번 말을 올려놓는다.
- A번 말의 위에 다른 말이 있는 경우에는 A번 말과 위에 있는 모든 말이 이동한다.
- 예를 들어, A, B, C로 쌓여있고, 이동하려는 칸에 D, E가 있는 경우에는 A번 말이 이동한 후에는 D, E, A, B, C가 된다.
→ 따라서, 지금 같이 이동할 말을 저장하길 위한 변수인
moving_horse_index_array 을 빈 배열로 만듭니다.

그리고 현재 위치에 쌓인 말들, current_stacked_horse_map[r][c] 을 이용해서
자신을 포함해 같이 이동해야 하는 말들을 추가합니다.

```
k = 4  # 말의 개수

chess_map = [
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0]
]
start_horse_location_and_directions = [
    [0, 0, 0],
    [0, 1, 0],
    [0, 2, 0],
    [2, 2, 2]
]
# 이 경우는 게임이 끝나지 않아 -1 을 반환해야 합니다!
# 동 서 북 남
# →, ←, ↑, ↓
dr = [0, 0, -1, 1]
dc = [1, -1, 0, 0]


def get_game_over_turn_count(horse_count, game_map, horse_location_and_directions):
    n = len(game_map)
    turn_count = 1
    current_stacked_horse_map = [[[] for _ in range(n)] for _ in range(n)]
    for i in range(horse_count):
        r, c, d = horse_location_and_directions[i]
        current_stacked_horse_map[r][c].append(i)

    while turn_count <= 1000:
        for horse_index in range(horse_count):
            n = len(game_map)
            r, c, d = horse_location_and_directions[horse_index]
            new_r = r + dr[d]
            new_c = c + dc[d]

            moving_horse_index_array = []
            for i in range(len(current_stacked_horse_map[r][c])):
                current_stacked_horse_index = current_stacked_horse_map[r][c][i]
                # 여기서 이동해야 하는 애들은 현재 옮기는 말 위의!!! 말들이다.
                if horse_index == current_stacked_horse_index:
                    moving_horse_index_array = current_stacked_horse_map[r][c][i:]
                    current_stacked_horse_map[r][c] = current_stacked_horse_map[r][c][:i]
                    break

        turn_count += 1
    return -1


print(get_game_over_turn_count(k, chess_map, start_horse_location_and_directions))  # 2가 반환 되어야합니다
```

✅ 자 그런데,

2) 빨간색인 경우에는 이동한 후에 A번 말과 그 위에 있는 모든 말의 쌓여있는 순서를 반대로 바꾼다.

- A, B, C가 이동하고, 이동하려는 칸에 말이 없는 경우에는 C, B, A가 된다.

- A, D, F, G가 이동하고, 이동하려는 칸에 말이 E, C, B로 있는 경우에는 E, C, B, G, F, D, A가 된다.

→ 이런 조건이 있었습니다!

따라서, game_map[new_r][new_c] == 1 라면 아예 배열을 뒤집어버립니다!

이 때, reversed 라는 함수를 쓰면 쉽게 뒤집을 수 있습니다!

```
k = 4  # 말의 개수

chess_map = [
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0]
]
start_horse_location_and_directions = [
    [0, 0, 0],
    [0, 1, 0],
    [0, 2, 0],
    [2, 2, 2]
]
# 이 경우는 게임이 끝나지 않아 -1 을 반환해야 합니다!
# 동 서 북 남
# →, ←, ↑, ↓
dr = [0, 0, -1, 1]
dc = [1, -1, 0, 0]


def get_game_over_turn_count(horse_count, game_map, horse_location_and_directions):
    n = len(game_map)
    turn_count = 1
    current_stacked_horse_map = [[[] for _ in range(n)] for _ in range(n)]
    for i in range(horse_count):
        r, c, d = horse_location_and_directions[i]
        current_stacked_horse_map[r][c].append(i)

    while turn_count <= 1000:
        for horse_index in range(horse_count):
            n = len(game_map)
            r, c, d = horse_location_and_directions[horse_index]
            new_r = r + dr[d]
            new_c = c + dc[d]

            moving_horse_index_array = []
            for i in range(len(current_stacked_horse_map[r][c])):
                current_stacked_horse_index = current_stacked_horse_map[r][c][i]
                # 여기서 이동해야 하는 애들은 현재 옮기는 말 위의!!! 말들이다.
                if horse_index == current_stacked_horse_index:
                    moving_horse_index_array = current_stacked_horse_map[r][c][i:]
                    current_stacked_horse_map[r][c] = current_stacked_horse_map[r][c][:i]
                    break

            # 2) 빨간색인 경우에는 이동한 후에 A번 말과 그 위에 있는 모든 말의 쌓여있는 순서를 반대로 바꾼다.
            if game_map[new_r][new_c] == 1:
                moving_horse_index_array = reversed(moving_horse_index_array)

        turn_count += 1
    return -1


print(get_game_over_turn_count(k, chess_map, start_horse_location_and_directions))  # 2가 반환 되어야합니다
```

✅ 그러면 이제 moving_horse_index_array를 반복하면서

새롭게 이동할 current_stacked_horse_map[new_r][new_c] 에 말들을 쌓습니다.

또한 현재 말들의 위치인 horse_location_and_directions 도 업데이트 해줍니다.

```python
k = 4  # 말의 개수

chess_map = [
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0]
]
start_horse_location_and_directions = [
    [0, 0, 0],
    [0, 1, 0],
    [0, 2, 0],
    [2, 2, 2]
]
# 이 경우는 게임이 끝나지 않아 -1 을 반환해야 합니다!
# 동 서 북 남
# →, ←, ↑, ↓
dr = [0, 0, -1, 1]
dc = [1, -1, 0, 0]


def get_game_over_turn_count(horse_count, game_map, horse_location_and_directions):
    n = len(game_map)
    turn_count = 1
    current_stacked_horse_map = [[[] for _ in range(n)] for _ in range(n)]
    for i in range(horse_count):
        r, c, d = horse_location_and_directions[i]
        current_stacked_horse_map[r][c].append(i)

    while turn_count <= 1000:
        for horse_index in range(horse_count):
            n = len(game_map)
            r, c, d = horse_location_and_directions[horse_index]
            new_r = r + dr[d]
            new_c = c + dc[d]

            moving_horse_index_array = []
            for i in range(len(current_stacked_horse_map[r][c])):
                current_stacked_horse_index = current_stacked_horse_map[r][c][i]
                # 여기서 이동해야 하는 애들은 현재 옮기는 말 위의!!! 말들이다.
                if horse_index == current_stacked_horse_index:
                    moving_horse_index_array = current_stacked_horse_map[r][c][i:]
                    current_stacked_horse_map[r][c] = current_stacked_horse_map[r][c][:i]
                    break

            # 2) 빨간색인 경우에는 이동한 후에 A번 말과 그 위에 있는 모든 말의 쌓여있는 순서를 반대로 바꾼다.
            if game_map[new_r][new_c] == 1:
                moving_horse_index_array = reversed(moving_horse_index_array)

            for moving_horse_index in moving_horse_index_array:
                current_stacked_horse_map[new_r][new_c].append(moving_horse_index)
                # horse_location_and_directions 에 이동한 말들의 위치를 업데이트한다.
                horse_location_and_directions[moving_horse_index][0], horse_location_and_directions[moving_horse_index][
                    1] = new_r, new_c

        turn_count += 1
    return -1


print(get_game_over_turn_count(k, chess_map, start_horse_location_and_directions))  # 2가 반환 되어야합니다
```

✅ 아! 그런데 깜빡한 조건이 있습니다.

3) 파란색인 경우에는 A번 말의 이동 방향을 반대로 하고 한 칸 이동한다. 방향을 반대로 바꾼 후에 이동하려는 칸이 파란색인 경우에는 이동하지 않고 가만히 있는다.

4) 체스판을 벗어나는 경우에는 파란색과 같은 경우이다.

→ 이에 대한 조건문을 추가합니다.
그런데 이 조건은, 사실 맨 뒤에 넣는 것보다 앞에 넣는 게 좋아요! 왜냐면 위에서 했던 것처럼
배열에 말의 인덱스를 쌓는 연산을 굳이 하지 않아도 되기 때문입니다!

그리고, 턴이 진행되던 중에 말이 4개 이상 쌓이는 순간 게임이 종료된다.
라고 했으니 해당 조건문도 추가해줍니다!

이를 구현하면 다음과 같습니다.

```
k = 4  # 말의 개수

chess_map = [
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0]
]
start_horse_location_and_directions = [
    [0, 0, 0],
    [0, 1, 0],
    [0, 2, 0],
    [2, 2, 2]
]
# 이 경우는 게임이 끝나지 않아 -1 을 반환해야 합니다!
# 동 서 북 남
# →, ←, ↑, ↓
dr = [0, 0, -1, 1]
dc = [1, -1, 0, 0]

def get_d_index_when_go_back(d):
    if d % 2 == 0:
        return d + 1
    else:
        return d - 1


def get_game_over_turn_count(horse_count, game_map, horse_location_and_directions):
    n = len(game_map)
    turn_count = 1
    current_stacked_horse_map = [[[] for _ in range(n)] for _ in range(n)]
    for i in range(horse_count):
        r, c, d = horse_location_and_directions[i]
        current_stacked_horse_map[r][c].append(i)

    while turn_count <= 1000:
        for horse_index in range(horse_count):
            n = len(game_map)
            r, c, d = horse_location_and_directions[horse_index]
            new_r = r + dr[d]
            new_c = c + dc[d]

            #  3) 파란색인 경우에는 A번 말의 이동 방향을 반대로 하고 한 칸 이동한다.
            if not 0 <= new_r < n or not 0 <= new_c < n or game_map[new_r][new_c] == 2:
                new_d = get_d_index_when_go_back(d)

                horse_location_and_directions[horse_index][2] = new_d
                new_r = r + dr[new_d]
                new_c = c + dc[new_d]
                # 3) 방향을 반대로 바꾼 후에 이동하려는 칸이 파란색인 경우에는 이동하지 않고 가만히 있는다.
                if not 0 <= new_r < n or not 0 <= new_c < n or game_map[new_r][new_c] == 2:
                    continue

            moving_horse_index_array = []
```

```
            for i in range(len(current_stacked_horse_map[r][c])):
                current_stacked_horse_index = current_stacked_horse_map[r][c][i]
                # 여기서 이동해야 하는 애들은 현재 옮기는 말 위의!!! 말들이다.
                if horse_index == current_stacked_horse_index:
                    moving_horse_index_array = current_stacked_horse_map[r][c][i:]
                    current_stacked_horse_map[r][c] = current_stacked_horse_map[r][c][:i]
                    break

            # 2) 빨간색인 경우에는 이동한 후에 A번 말과 그 위에 있는 모든 말의 쌓여있는 순서를 반대로 바꾼다.
            if game_map[new_r][new_c] == 1:
                moving_horse_index_array = reversed(moving_horse_index_array)

            for moving_horse_index in moving_horse_index_array:
                current_stacked_horse_map[new_r][new_c].append(moving_horse_index)
                # horse_location_and_directions 에 이동한 말들의 위치를 업데이트한다.
                horse_location_and_directions[moving_horse_index][0], horse_location_and_directions[moving_horse_index][
                    1] = new_r, new_c
            if len(current_stacked_horse_map[new_r][new_c]) >= 4:
                return turn_count

        turn_count += 1
    return -1


print(get_game_over_turn_count(k, chess_map, start_horse_location_and_directions))  # 2가 반환 되어야합니다
```

👍 자 이렇게 문제를 풀어봤습니다!
어떠신가요?

문제를 해결하기 위한 어느 정도의 아이디어를 떠올리면,
문제에 있는 구현사항만 가지고 쉽게 구현할 수 있습니다!
힌트를 주워서, 코드로 구현하라! 였습니다.

수고 많으셨습니다!

06. 삼성 역량 테스트 - 2

▼ 5) ✍️ 구슬 탈출

❓ 스타트링크에서 판매하는 어린이용 장난감 중에서 가장 인기가 많은 제품은 구슬 탈출이다. 구슬 탈출은 직사각형 보드에 빨간 구슬과 파란 구슬을 하나씩 넣은 다음, 빨간 구슬을 구멍을 통해 빼내는 게임이다.

보드의 세로 크기는 N, 가로 크기는 M이고, 편의상 1×1크기의 칸으로 나누어져 있다. 가장 바깥 행과 열은 모두 막혀져 있고, 보드에는 구멍이 하나 있다. 빨간 구슬과 파란 구슬의 크기는 보드에서 1×1크기의 칸을 가득 채우는 사이즈이고, 각각 하나씩 들어가 있다. 게임의 목표는 빨간 구슬을 구멍을 통해서 빼내는 것이다. 이때, 파란 구슬이 구멍에 들어가면 안 된다.

이때, 구슬을 손으로 건드릴 수는 없고, 중력을 이용해서 이리 저리 굴려야 한다. 왼쪽으로 기울이기, 오른쪽으로 기울이기, 위쪽으로 기울이기, 아래쪽으로 기울이기와 같은 네 가지 동작이 가능하다.

각각의 동작에서 공은 동시에 움직인다. 빨간 구슬이 구멍에 빠지면 성공이지만, 파란 구슬이 구멍에 빠지면 실패이다. 빨간 구슬과 파란 구슬이 동시에 구멍에 빠져도 실패이다. 빨간 구슬과 파란 구슬은 동시에 같은 칸에 있을 수 없다. 또, 빨간 구슬과 파란 구슬의 크기는 한 칸을 모두 차지한다. 기울이는 동작을 그만하는 것은 더 이상 구슬이 움직이지 않을 때 까지이다.

보드의 상태가 주어졌을 때, 10번 이하로 빨간 구슬을 구멍을 통해 빼낼 수 있는지 구하는 프로그램을 작성하시오.
입력
보드를 나타내는 2차원 배열 game_map이 주어진다.
이 때, 보드의 행, 열의 길이는 3이상 10 이하다.

보드 내에 문자열은 '.', '#', 'O', 'R', 'B' 로 이루어져 있다.
'.'은 빈 칸을 의미하고,
'#'은 공이 이동할 수 없는 장애물 또는 벽을 의미하며,
'O'는 구멍의 위치를 의미한다.
'R'은 빨간 구슬의 위치,
'B'는 파란 구슬의 위치이다.

입력되는 모든 보드의 가장자리에는 모두 '#'이 있다. 구멍의 개수는 한 개 이며, 빨간 구슬과 파란 구슬은 항상 1개가 주어진다.

출력
파란 구슬을 구멍에 넣지 않으면서 빨간 구슬을 10번 이하로 움직여서 빼낼 수 있으면 True, 없으면 False를 반환한다.

```
game_map = [
    ["#", "#", "#", "#", "#"],
    ["#", ".", ".", "B", "#"],
    ["#", ".", "#", ".", "#"],
    ["#", "R", "O", ".", "#"],
    ["#", "#", "#", "#", "#"],
]  # -> True를 반환해야 한다.

game_map = [
    ["#", "#", "#", "#", "#", "#", "#", "#", "#", "#"],
    ["#", "R", "#", ".", ".", ".", "#", "#", "B", "#"],
    ["#", ".", ".", ".", "#", ".", "#", "#", ".", "#"],
    ["#", "#", "#", "#", "#", ".", "#", "#", ".", "#"],
    ["#", ".", ".", ".", ".", ".", ".", "#", ".", "#"],
    ["#", ".", "#", "#", "#", "#", "#", "#", ".", "#"],
    ["#", ".", "#", ".", ".", ".", ".", "#", ".", "#"],
    ["#", ".", "#", ".", "#", ".", "#", ".", ".", "#"]
    ["#", ".", ".", ".", "#", ".", "O", "#", ".", "#"],
    ["#", "#", "#", "#", "#", "#", "#", "#", "#", "#"]
]  # -> False 를 반환해야 한다
```

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! **실전 문제처럼, 20분 이상을 고민해 본 다음**, 아래 방법들을 펼쳐 봅시다!

▼ [코드스니펫] 구슬 탈출

```
from collections import deque

game_map = [
    ["#", "#", "#", "#", "#"],
    ["#", ".", ".", "B", "#"],
    ["#", ".", "#", ".", "#"],
    ["#", "R", "O", ".", "#"],
    ["#", "#", "#", "#", "#"],
]


def is_available_to_take_out_only_red_marble(game_map):
    # 구현해보세요!
    return


print(is_available_to_take_out_only_red_marble(game_map))  # True 를 반환해야 합니다
```

▼ 해결 방법

✅ 파란 구슬과 빨간 구슬이 동시에 굴러다니며,
끝까지 굴러가는 상황들이 겹쳐서 문제 해결의 규칙성을 찾기 쉽지 않습니다.

따라서 탐색하는 방법을 탐색해나가야 하는데, 이동할 수 있는 방향이 4개나 있습니다!
즉 모든 경우를 시도해보면서 탈출하는 경우를 찾아야 합니다!
→ 모든 경우를 탐색해보는 BFS 로 구현하면 되겠군요!

자 그러면, queue 를 사용해야 겠군요!

자, 이제 파란 구슬은 넣지 않되 빨간 구슬을 나가게 해봅시다!
BFS 를 구현하려면 방문 여부를 저장할 visited 가 필요한데...
어? 공이 2개나 있는데, 어떻게 방문했는지 아닌지를 저장하면 되나요?
원래는 1차원 배열 혹은 2차원 배열에만 visited 를 저장했는데....

바로, 4차원 배열을 사용하면 됩니다.

헐?? 너무 공간을 낭비하는 거 아닐까요?? 라고 걱정하실 수 있는데,
보드의 행과 열의 길이가 고작 3 ≤ x ≤ 10 입니다.
이 경우에는 만드셔도 아~~무 상관없습니다.
따라서, 4차원 배열로 visited 를 초기화 시켜줍니다.

```
from collections import deque

game_map = [
    ["#", "#", "#", "#", "#"],
    ["#", ".", ".", "B", "#"],
    ["#", ".", "#", ".", "#"],
    ["#", "R", "O", ".", "#"],
    ["#", "#", "#", "#", "#"],
]


def is_available_to_take_out_only_red_marble(game_map):
    n, m = len(game_map), len(game_map[0])
    visited = [[[[False] * m for _ in range(n)] for _ in range(m)] for _ in range(n)]


print(is_available_to_take_out_only_red_marble(game_map))  # True 를 반환해야 합니다
```

✅ 그러면 이제,
BFS에서 탐색을 원할하게 하기 위해 queue 에 탐색할 데이터를 쌓아줍시다.

우리는 빨간 구슬과 파란구슬을 모두 신경써야 하므로
둘의 위치를 넣어줘야 합니다.
둘의 위치를 이중 반복문을 이용해서 찾아줍니다.

또한, 우리는 최대 10회까지만 탐색할 수 있습니다!
따라서, 탐색횟수 까지 큐에 넣어줍니다.
그리고 visited 에 현재 빨간 구슬과 파란 구슬의 위치를 방문했다고 처리합니다!
이렇게요!

visited[red_row][red_col][blue_row][blue_col] = True

```
from collections import deque

game_map = [
    ["#", "#", "#", "#", "#"],
    ["#", ".", ".", "B", "#"],
    ["#", ".", "#", ".", "#"],
    ["#", "R", "O", ".", "#"],
    ["#", "#", "#", "#", "#"],
]


def is_available_to_take_out_only_red_marble(game_map):
    n, m = len(game_map), len(game_map[0])
    visited = [[[[False] * m for _ in range(n)] for _ in range(m)] for _ in range(n)]
    queue = deque()
    red_row, red_col, blue_row, blue_col = -1, -1, -1, -1
    for i in range(n):
        for j in range(m):
            if game_map[i][j] == "R":
                red_row, red_col = i, j
            elif game_map[i][j] == "B":
                blue_row, blue_col = i, j

    queue.append((red_row, red_col, blue_row, blue_col, 1))
    visited[red_row][red_col][blue_row][blue_col] = True
```

그럼 이제 큐를 이용해 탐색해야겠죠?

큐에서 꺼낸 빨간 구슬과 파랑 구슬의 위치를 가지고
매 방향으로 이동하는 것을 계산해야 합니다!

그런데, 각 방향으로 벽 혹은 구멍이 나올 때까지 쭉 ~! 이동하는 걸 어떻게 구하면 될까요?

move_until_wall_or_hole 라는 함수로 한 번 구현해보겠습니다.
이동하는 방향에 따른 변화량을 받아서 계속 움직입니다.
그리고 다음 칸이 구슬이거나 이동한 칸이 구슬이면 멈추고 해당 값을 반환하면 됩니다!

이 때, 파란 구슬이 구멍에 도착했다면? 실패했으니 continue,
빨간 구슬이 구멍에 도착했다면? True 를 반환하면 됩니다.

그런데 여기서 중요한 점이 있습니다.
아래처럼 되어 있을 때, 왼쪽으로 이동하면 어떻게 될까요?

* . . B R
* B R . .
처럼 되어야겠죠? 이를 구현하기 위해서는, 뭐가 방향에 따라 앞섰는지를 알아야 합니다.
그래서 이를 위해 끝까지 이동하는 횟수를 구합니다.
이동하는 횟수가 적은 친구가 끝으로 부터 한 칸 떨어져있도록 만들면 됩니다.

```
def move_until_wall_or_hole(r, c, diff_r, diff_c, game_map):
    move_count = 0  # 이동한 칸 수
    # 다음 이동이 벽이거나 구멍이 아닐 때까지
    while game_map[r + diff_r][c + diff_c] != '#' and game_map[r][c] != 'O':
        r += diff_r
        c += diff_c
        move_count += 1
    return r, c, move_count

...
    while queue:
        red_row, red_col, blue_row, blue_col, try_count = queue.popleft()  # FIFO
        if try_count > 10:  # 10 이하여야 한다.
            break

        for i in range(4):
            next_red_row, next_red_col, r_count = move_until_wall_or_hole(red_row, red_col, dr[i], dc[i], game_map)
            next_blue_row, next_blue_col, b_count = move_until_wall_or_hole(blue_row, blue_col, dr[i], dc[i], game_map)

            if game_map[next_blue_row][next_blue_col] == 'O':  # 파란 구슬이 구멍에 떨어지지 않으면(실패 X)
                continue
            if game_map[next_red_row][next_red_col] == 'O':  # 빨간 구슬이 구멍에 떨어진다면(성공)
                return True
            if next_red_row == next_blue_row and next_red_col == next_blue_col:  # 빨간 구슬과 파란 구슬이 동시에 같은 칸에 있을 수 없다.
                if r_count > b_count:  # 이동 거리가 많은 구슬을 한칸 뒤로
                    next_red_row -= dr[i]
                    next_red_col -= dc[i]
                else:
                    next_blue_row -= dr[i]
                    next_blue_col -= dc[i]
...
```

자 그리고 마지막으로
만약 방문하지 않았다면 해당 위치를 다시 queue 에 넣고 이를 반복해주시면 됩니다!

```
...
    while queue:
        red_row, red_col, blue_row, blue_col, try_count = queue.popleft()  # FIFO
        if try_count > 10:  # 10 이하여야 한다.
            break

        for i in range(4):
            next_red_row, next_red_col, r_count = move_until_wall_or_hole(red_row, red_col, dr[i], dc[i], game_map)
            next_blue_row, next_blue_col, b_count = move_until_wall_or_hole(blue_row, blue_col, dr[i], dc[i], game_map)

            if game_map[next_blue_row][next_blue_col] == 'O':  # 파란 구슬이 구멍에 떨어지지 않으면(실패 X)
                continue
            if game_map[next_red_row][next_red_col] == 'O':  # 빨간 구슬이 구멍에 떨어진다면(성공)
                return True
            if next_red_row == next_blue_row and next_red_col == next_blue_col:  # 빨간 구슬과 파란 구슬이 동시에 같은 칸에 있을 수 없다.
                if r_count > b_count:  # 이동 거리가 많은 구슬을 한칸 뒤로
                    next_red_row -= dr[i]
                    next_red_col -= dc[i]
                else:
                    next_blue_row -= dr[i]
                    next_blue_col -= dc[i]
            # BFS 탐색을 마치고, 방문 여부 확인
            if not visited[next_red_row][next_red_col][next_blue_row][next_blue_col]:
                visited[next_red_row][next_red_col][next_blue_row][next_blue_col] = True
                queue.append((next_red_row, next_red_col, next_blue_row, next_blue_col, try_count + 1))
...
```

✅ 최종 코드는 다음과 같습니다

```
from collections import deque

game_map = [
    ["#", "#", "#", "#", "#"],
    ["#", ".", ".", "B", "#"],
    ["#", ".", "#", ".", "#"],
    ["#", "R", "O", ".", "#"],
    ["#", "#", "#", "#", "#"],
]

dr = [-1, 0, 1, 0]
dc = [0, 1, 0, -1]


def move_until_wall_or_hole(r, c, diff_r, diff_c, game_map):
    move_count = 0  # 이동한 칸 수
    # 다음 이동이 벽이거나 구멍이 아닐 때까지
    while game_map[r + diff_r][c + diff_c] != '#' and game_map[r][c] != 'O':
        r += diff_r
        c += diff_c
        move_count += 1
    return r, c, move_count


def is_available_to_take_out_only_red_marble(game_map):
    n, m = len(game_map), len(game_map[0])
    visited = [[[[False] * m for _ in range(n)] for _ in range(m)] for _ in range(n)]
    queue = deque()
    red_row, red_col, blue_row, blue_col = -1, -1, -1, -1
    for i in range(n):
        for j in range(m):
            if game_map[i][j] == "R":
                red_row, red_col = i, j
            elif game_map[i][j] == "B":
                blue_row, blue_col = i, j

    queue.append((red_row, red_col, blue_row, blue_col, 1))
    visited[red_row][red_col][blue_row][blue_col] = True

    while queue:
        red_row, red_col, blue_row, blue_col, try_count = queue.popleft()  # FIFO
        if try_count > 10:  # 10 이하여야 한다.
```

```
            break

        for i in range(4):
            next_red_row, next_red_col, r_count = move_until_wall_or_hole(red_row, red_col, dr[i], dc[i], game_map)
            next_blue_row, next_blue_col, b_count = move_until_wall_or_hole(blue_row, blue_col, dr[i], dc[i], game_map)

            if game_map[next_blue_row][next_blue_col] == 'O':  # 파란 구슬이 구멍에 떨어지지 않으면(실패 X)
                continue
            if game_map[next_red_row][next_red_col] == 'O':  # 빨간 구슬이 구멍에 떨어진다면(성공)
                return True
            if next_red_row == next_blue_row and next_red_col == next_blue_col:  # 빨간 구슬과 파란 구슬이 동시에 같은 칸에 있을 수 없다.
                if r_count > b_count:  # 이동 거리가 많은 구슬을 한칸 뒤로
                    next_red_row -= dr[i]
                    next_red_col -= dc[i]
                else:
                    next_blue_row -= dr[i]
                    next_blue_col -= dc[i]
            # BFS 탐색을 마치고, 방문 여부 확인
            if not visited[next_red_row][next_red_col][next_blue_row][next_blue_col]:
                visited[next_red_row][next_red_col][next_blue_row][next_blue_col] = True
                queue.append((next_red_row, next_red_col, next_blue_row, next_blue_col, try_count + 1))

    return False


print(is_available_to_take_out_only_red_marble(game_map))  # True 를 반환해야 합니다
```

👍 자 이렇게 문제를 풀어봤습니다!
어떠신가요?

BFS 를 4차원 배열을 이용해서 해결한다니, 신기하죠?
입력값의 범위에 따라서 사용 가능한 자료구조도 달라질 수도 있다는 점
생각해주셨으면 좋겠습니다!

수고 많으셨습니다!

07. 삼성 역량 테스트 - 3

▼ 6) ✍️ 치킨 배달

❓ Q. 크기가 N×N인 도시가 있다. 도시는 1×1크기의 칸으로 나누어져 있다. 도시의 각 칸은 빈 칸, 치킨집, 집 중 하나이다. 도시의 칸은 (r, c)와 같은 형태로 나타내고, r행 c열 또는 위에서부터 r번째 칸, 왼쪽에서부터 c번째 칸을 의미한다. r과 c는 1부터 시작한다.

이 도시에 사는 사람들은 치킨을 매우 좋아한다. 따라서, 사람들은 "치킨 거리"라는 말을 주로 사용한다. 치킨 거리는 집과 가장 가까운 치킨집 사이의 거리이다. 즉, 치킨 거리는 집을 기준으로 정해지며, 각각의 집은 치킨 거리를 가지고 있다. 도시의 치킨 거리는 모든 집의 치킨 거리의 합이다.

임의의 두 칸 (r1, c1)과 (r2, c2) 사이의 거리는 |r1-r2| + |c1-c2|로 구한다.

예를 들어, 아래와 같은 지도를 갖는 도시를 살펴보자.

```
0 2 0 1 0
1 0 1 0 0
0 0 0 0 0
0 0 0 1 1
0 0 0 1 2
```

❓ 0은 빈 칸, 1은 집, 2는 치킨집이다.

(2, 1)에 있는 집과 (1, 2)에 있는 치킨집과의 거리는 |2-1| + |1-2| = 2, (5, 5)에 있는 치킨집과의 거리는 |2-5| + |1-5| = 7이다. 따라서, (2, 1)에 있는 집의 치킨 거리는 2이다.

(5, 4)에 있는 집과 (1, 2)에 있는 치킨집과의 거리는 |5-1| + |4-2| = 6, (5, 5)에 있는 치킨집과의 거리는 |5-5| + |4-5| = 1이다. 따라서, (5, 4)에 있는 집의 치킨 거리는 1이다.

이 도시에 있는 치킨집은 모두 같은 프랜차이즈이다. 프렌차이즈 본사에서는 수익을 증가시키기 위해 일부 치킨집을 폐업시키려고 한다. 오랜 연구 끝에 이 도시에서 가장 수익을 많이 낼 수 있는 치킨집의 개수는 최대 M개라는 사실을 알아내었다.

도시에 있는 치킨집 중에서 최대 M개를 고르고, 나머지 치킨집은 모두 폐업시켜야 한다. 어떻게 고르면, 도시의 치킨 거리가 가장 작게 될지 반환하시오.

입력
N(2 ≤ N ≤ 50)과 M(1 ≤ M ≤ 13)이 주어진다.
또한 도시의 정보가 주어진다.

도시의 정보는 0, 1, 2로 이루어져 있고, 0은 빈 칸, 1은 집, 2는 치킨집을 의미한다. 집의 개수는 2N개를 넘지 않으며, 적어도 1개는 존재한다. 치킨집의 개수는 M보다 크거나 같고, 13보다 작거나 같다.

출력
첫째 줄에 폐업시키지 않을 치킨집을 최대 M개를 골랐을 때, 도시의 치킨 거리의 최솟값을 출력한다.

```
n = 5
m = 3

city_map = [
    [0, 0, 1, 0, 0],
    [0, 0, 2, 0, 1],
    [0, 1, 2, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 0, 0, 2],
]  # 5 가 출력되어야 합니다
```

▼ 👏 여기서 잠깐! Tip

▼ 조합을 얻으려면?

모든 경우의 수를 구해야 하는 문제를 풀기 위해서는
특정 숫자들의 모든 조합을 얻어야 할 때가 있습니다.
그럴 때는 아래와 같이 itertools 모듈의 combinations 을 이용하시면 됩니다!

```
>>> from itertools import combinations
>>> itertools.combinations([1,2,3,4,5], 2) # [1,2,3,4,5] 에서 2개씩 뽑을 수 있는 조합을 다 가져와라
<itertools.combinations object at 0x10453acc0>
>>> list(itertools.combinations([1,2,3,4,5], 2))
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
>>> list(itertools.combinations([1,2,3,4,5], 3))
[(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5), (1, 4, 5), (2, 3, 4), (2, 3, 5), (2, 4, 5), (3, 4, 5)]
```

▼ 숫자의 최댓값을 얻으려면?

최솟값의 초깃값을 잡기 애매할 때가 있습니다.
예를 들어 내가 원소를 비교하면서 최솟값을 찾고 싶은데,

min = 99999 이라고 뒀습니다.
그런데 모든 원소들이 막 200만 이상이에요.
그러면 최솟값이 99999로 오는 실수를 범할 수 있어서,
시스템상 최댓값을 min 의 초깃값으로 설정하는 게 좋습니다.
그럴 때 쓰는 게 sys.maxsize 입니다

```
>>> from sys
>>> min_value = sys.maxsize
>>> min_value
9223372036854775807
```

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! **실전 문제처럼, 20분 이상을 고민해 본 다음**, 아래 방법들을 펼쳐 봅시다!

▼ **[코드스니펫] 치킨 배달**

```
import itertools, sys

n = 5
m = 3

city_map = [
    [0, 0, 1, 0, 0],
    [0, 0, 2, 0, 1],
    [0, 1, 2, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 0, 0, 2],
]


def get_min_city_chicken_distance(n, m, city_map):
    return


# 출력
print(get_min_city_chicken_distance(n, m, city_map))  # 5 가 반환되어야 합니다!
```

▼ 해결 방법

✅ **여러 개 중에서 M 개를 고른 뒤**, 그 치킨 거리가 가장 작게 되는 경우를 반환하시오.
→ 여러 개 중에서 특정 개수를 뽑는 경우의 수

여러 개 중에서 M 개를 고른 뒤, 그 치킨 거리가 **가장 작게 되는 경우**를 반환하시오.
→ 모든 경우의 수를 다 봐야 함.

이 두 가지 요구사항이 있으면 조합을 사용하셔야 합니다!
M개를 뽑는 모든 경우의 수를 다 봐서 최소 값을 가지는 경우를 반환하시면 됩니다!

한 번, 같이 구현해보겠습니다.
우선 지도를 가지고 집의 위치들과 치킨의 위치들을 가져와야 합니다.
이걸 가지고 최소 거리를 계산 할 수 있으니까요!

```
import itertools, sys

n = 5
m = 3

city_map = [
    [0, 0, 1, 0, 0],
    [0, 0, 2, 0, 1],
```

```
    [0, 1, 2, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 0, 0, 2],
]


def get_min_city_chicken_distance(n, m, city_map):
    chicken_location_list = []
    home_location_list = []
    for i in range(n):
        for j in range(n):
            if city_map[i][j] == 1:
                home_location_list.append([i, j])
            if city_map[i][j] == 2:
                chicken_location_list.append([i, j])


# 출력
print(get_min_city_chicken_distance(n, m, city_map))  # 5 가 반환되어야 합니다!
```

그리고 이제 치킨집의 조합을 구해봅시다.
itertools.combinations(chicken_location_list, m) 를 이용하시면 됩니다!

그리고 저희가 원하는 것은 이 조합들을 시도해봤을 때 가지는 최소 도시 치킨 거리 입니다!
이의 초깃값을 시스템의 최댓값으로 설정합니다.

그리고 각 조합을 반복하면서 집의 위치와 치킨 집들의 거리를 비교하면서
각 집의 최소 치킨 거리를 구합니다.
그리고 최종적으로 모든 조합의 최소 도시 치킨 거리를 구해서 반환하시면 됩니다!

```
import itertools, sys

n = 5
m = 3

city_map = [
    [0, 0, 1, 0, 0],
    [0, 0, 2, 0, 1],
    [0, 1, 2, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 0, 0, 2],
]


def get_min_city_chicken_distance(n, m, city_map):
    chicken_location_list = []
    home_location_list = []
    for i in range(n):
        for j in range(n):
            if city_map[i][j] == 1:
                home_location_list.append([i, j])
            if city_map[i][j] == 2:
                chicken_location_list.append([i, j])

    # 치킨집 중에 M개 고르기(조합)
    chicken_location_m_combinations = list(itertools.combinations(chicken_location_list, m))
    min_distance_of_m_combinations = sys.maxsize
    for chicken_location_m_combination in chicken_location_m_combinations:
        distance = 0
        for home_r, home_c in home_location_list:
            min_home_chicken_distance = sys.maxsize
            for chicken_location in chicken_location_m_combination:
                min_home_chicken_distance = min(
                    min_home_chicken_distance,
                    abs(home_r - chicken_location[0]) + abs(home_c - chicken_location[1])
                )
            distance += min_home_chicken_distance
        min_distance_of_m_combinations = min(min_distance_of_m_combinations, distance)
    return min_distance_of_m_combinations
```

```
# 출력
print(get_min_city_chicken_distance(n, m, city_map))  # 5 가 반환되어야 합니다!
```

👍 자 이렇게 문제를 풀어봤습니다!

모든 경우의 수를 다 해봐야하지만, 비교적 단순하게 문제가 풀리는 문제였습니다!
for 문이 3번이나 나와서 걱정하셨을텐데, 이렇게 무식하게 풀어야 풀리는 문제도 있습니다.

수고 많으셨습니다!

08. 나만의 알고리즘 노트 만들기

▼ 7) Git & Github이란

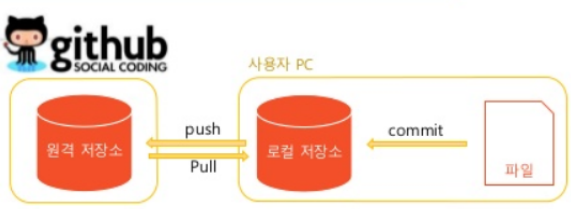
- 버전 관리가 안되어 힘든 경험 다들 있지 않으신가요? 아래처럼 말이죠!

- **Git** 은 작업 기록을 남기고 이력을 **추적**해서 코드를 손쉽게 관리하도록 도와줍니다. 버전 관리도 매우 간편합니다.
  - 어떤 버전부터 버그가 생긴 걸까? 어떤 내용이 변경되었는지 한 눈에 보고 싶다!
  - 여러 사람 / 컴퓨터에서 코드 작업을 동시에 하고 있다면 변경된 내역을 어떻게 합칠까?
  - 이력 추적하기 예시 👉헌법 개정안 비교
- **Github** 은 작업 기록과 파일을 저장하는 Git 원격 저장소를 지원해주는 사이트입니다.
  - 그 외에도 최근 트렌드인 오픈소스 찾기, 프로젝트 issue 관리처럼 다양한 부가 기능을 제공하고 있어요. (비슷한 곳으로는 Gitlab, bitbucket 등이 있습니다.)

▼ 8) Git의 commit / pull / push 란

혼자 개발 할 때에도 git을 쓰면 무척 편리합니다.
만약 작성 중이던 코드가 한참 잘못돼서 일주일 전 버전으로 돌리고 싶을 때, git이 없다면 난감하겠죠?

- 우리는 앞으로 주로 '내 컴퓨터(로컬 저장소)에 작업 내역을 반영하고, 원격 저장소(예. Github)에 작업 내역을 올리기' 를 많이 사용해볼거에요. 이제 Git 에서 가장 많이 쓰이는 commit / pull / push 를 살펴볼게요!



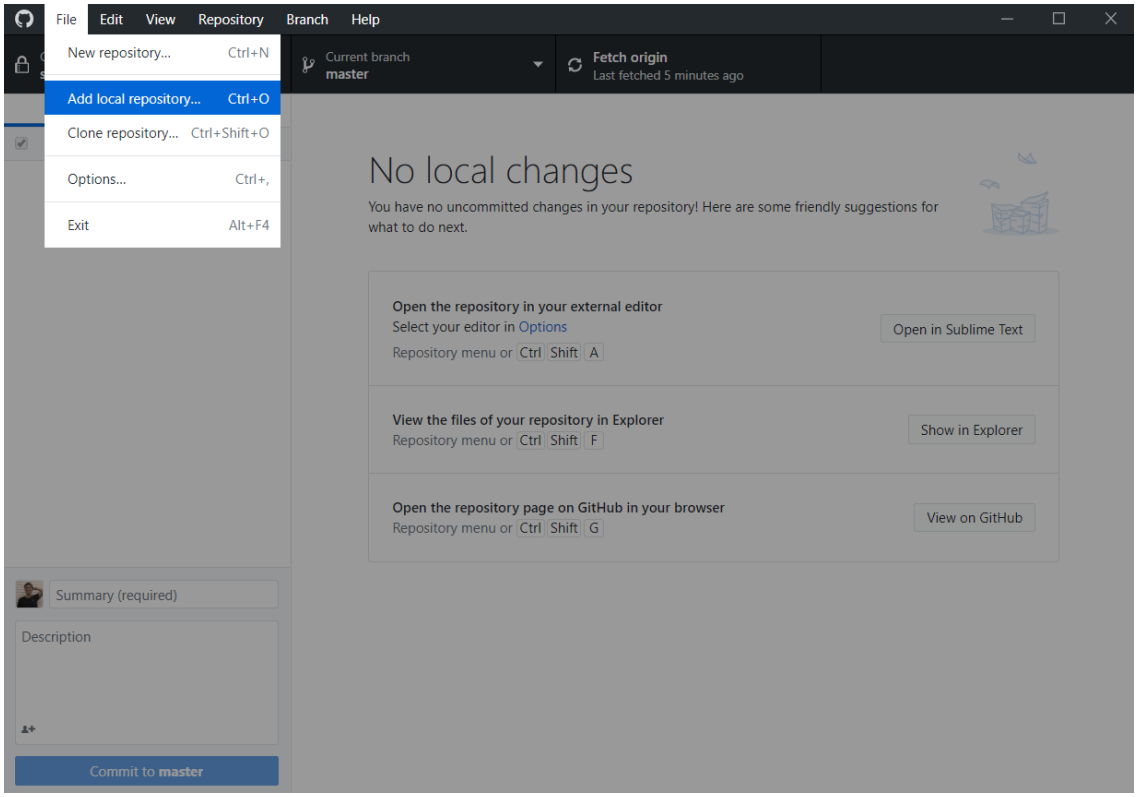
1. 내 로컬 저장소에 작업 내역을 반영하고 → `commit`
    - commit 앞에 파일을 새로 추적하는 `add` 과정이 있습니다. 우리가 쓰는 Github Desktop 처럼 GUI 툴을 사용하게 되면, 보통 commit 작업에 포함되어 있습니다.
2. 원격 저장소에 작업 내역을 업로드하는 것 → `push`
    - 보통은 여러 사용자가 올린 것이라도 작업 내역에 겹치는 게 없다면 자동으로 합쳐줍니다.
    - 여러 사용자가 하나의 파일의 동일한 부분을 고쳤다면, 어떤 부분을 어떻게 반영해야할까요? Git이 충돌(conflict)났다고 알려주고 사용자에게 직접 수정하라고 알려줍니다.
3. 원격 저장소 작업 내역을 내 로컬 저장소로 가져오는 것 → `pull`
    - 다른 사람의 작업내역을 가져오거나, 다른 컴퓨터에서 작업한 내용을 가져올 수 있습니다.
    - 보통 나의 작업 내역을 `commit` 하고 , `pull` 해오면 좋습니다. 그렇지 않으면 다른 사람의 작업 내역으로 내 작업 내역이 덮어쓰기가 되어버립니다.
        - 임시로 파일의 작업 내역을 보관해두는 `stash` 도 있습니다. 하지만 우리는 처음 배우니까 요건 나중에 git에 익숙해지면 써 보세요!

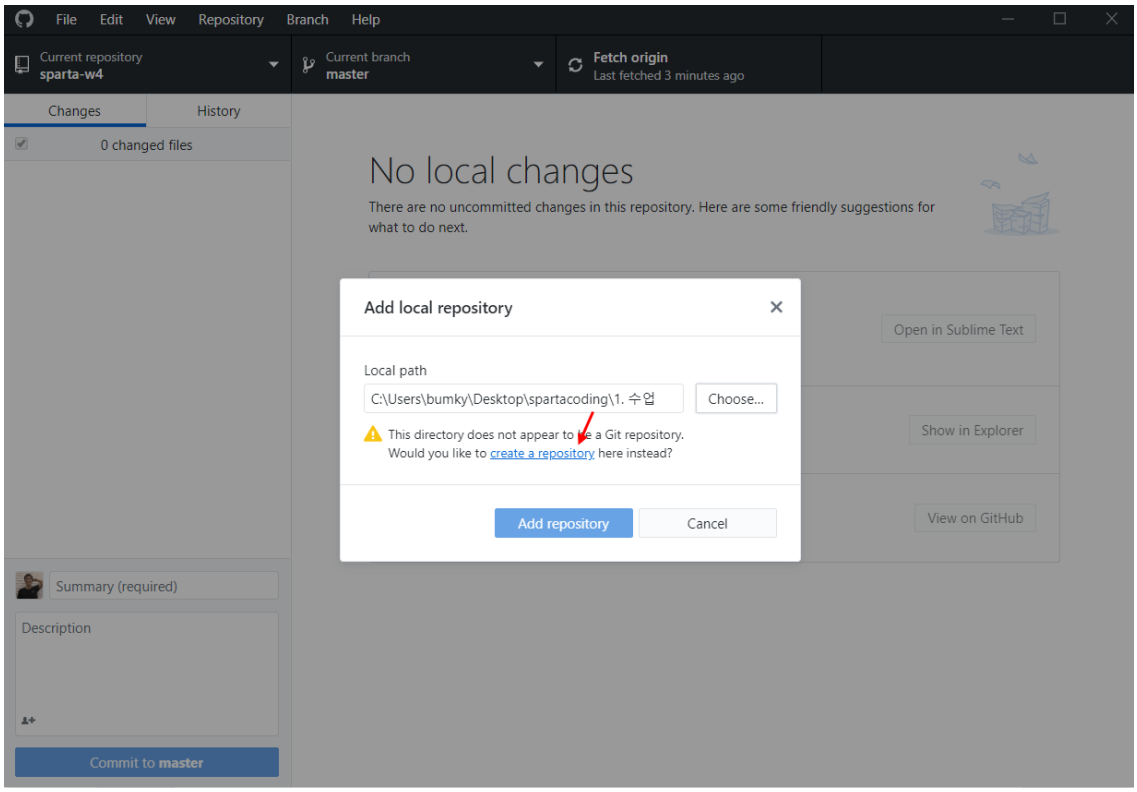
▼ 9) **알고리즘 폴더를** git에 올려보기 - Github Desktop 사용

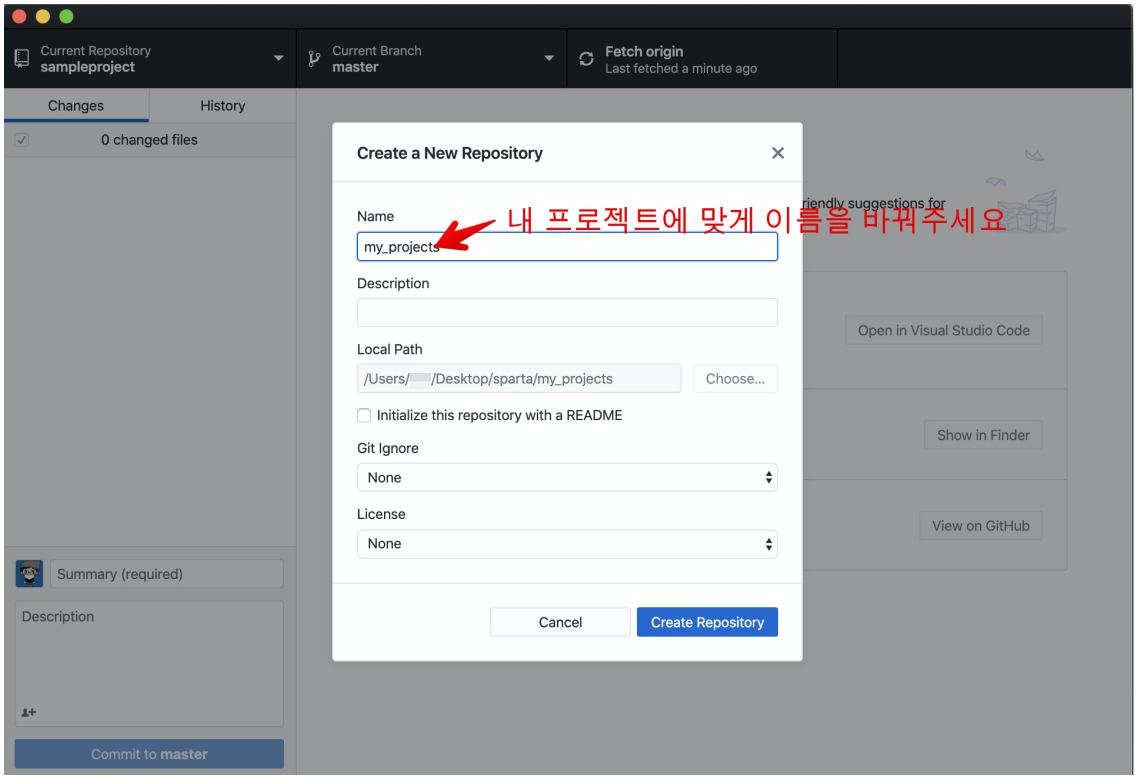
- Github Desktop이란? 그래픽 인터페이스(클릭, 클릭 / GUI)로 Git을 쓸 수 있게 하는 프로그램!

1. 내 로컬 저장소(Repository) 만들기
    - Github Desktop 을 켜고 로그인 후, add local repository 클릭, 여러분들의 sparta_algorithm 폴더를 클릭하고, 파란색 글자/버튼을 따라 누르면 repository가 완성됩니다.
    - 기존에 존재하던 폴더(homework)를 git 이 추적(tracking)할 수 있게 설정했다고 생각하면 됩니다. (sparta_algorithm 폴더 안에 git 설정 내용을 담고 있는 `.git` 이라는 폴더가 생성되었을 거에요.)

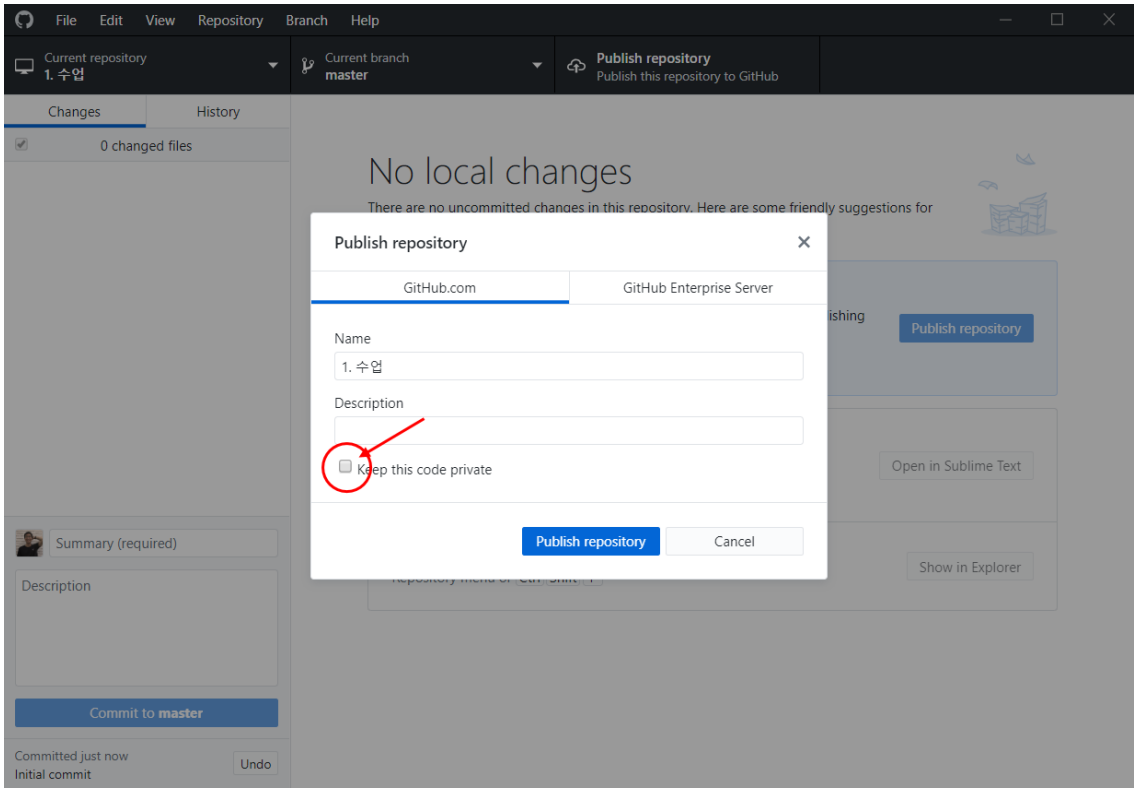
    ▼ 화면


File
Edit
View
Repository
Branch
Help
New repository... Ctrl+N
Add local repository... Ctrl+O
Clone repository... Ctrl+Shift+O
Options... Ctrl+,
Exit Alt+F4
Current branch
master
Fetch origin
Last fetched 5 minutes ago
No local changes
You have no uncommitted changes in your repository! Here are some friendly suggestions for what to do next.
Open the repository in your external editor
Select your editor in Options
Repository menu or Ctrl Shift A
Open in Sublime Text
View the files of your repository in Explorer
Repository menu or Ctrl Shift F
Show in Explorer
Open the repository page on GitHub in your browser
Repository menu or Ctrl Shift G
View on GitHub
Summary (required)
Description
Commit to master

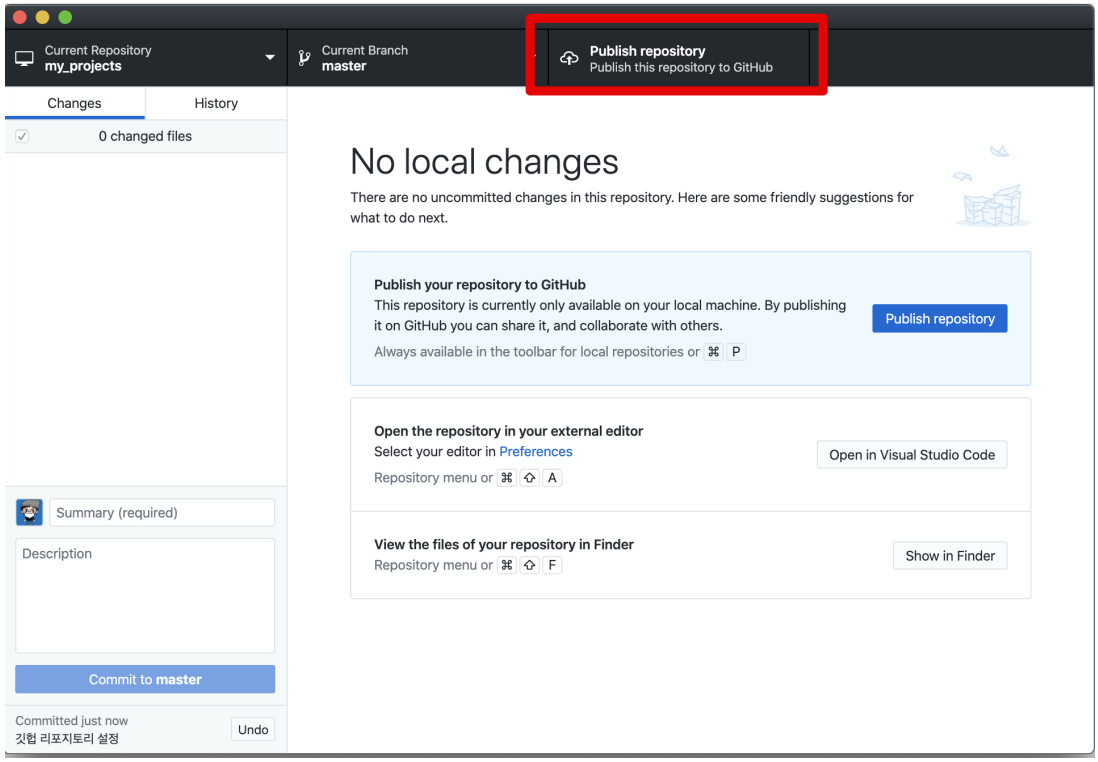

File
Edit
View
Repository
Branch
Help
Current repository
sparta-w4
Current branch
master
Fetch origin
Last fetched 3 minutes ago
Changes
History
0 changed files
No local changes
There are no uncommitted changes in this repository. Here are some friendly suggestions for what to do next.
Open in Sublime Text
Show in Explorer
View on GitHub
Add local repository
Local path
C:\Users\bumky\Desktop\spartacoding\1. 수업
Choose...
This directory does not appear to be a Git repository.
Would you like to create a repository here instead?
Add repository
Cancel
Summary (required)
Description
Commit to master



반드시 'keep this code private'을 해제해주세요!
공개되어 있어야 튜터도 숙제 확인이 가능하답니다.

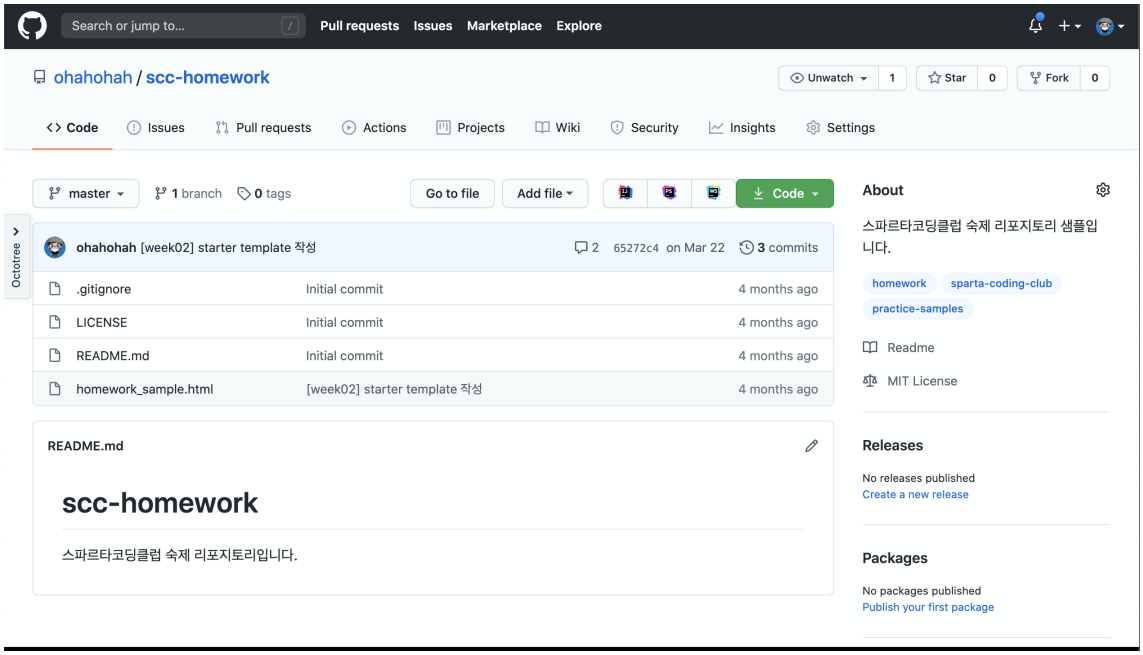


- Github 에 Publish(초기 공개) 합니다



2. github.com에 로그인 후, 원격 저장소(repository)가 생성된 것 확인

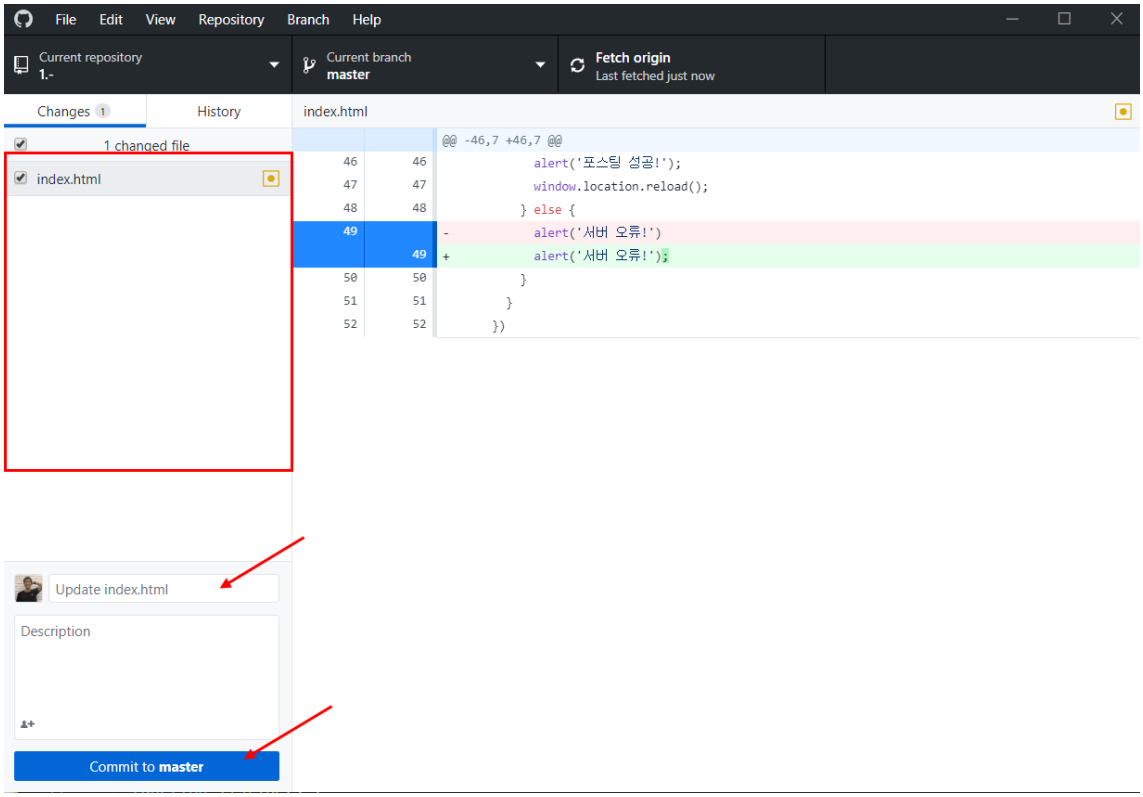
▼ 화면

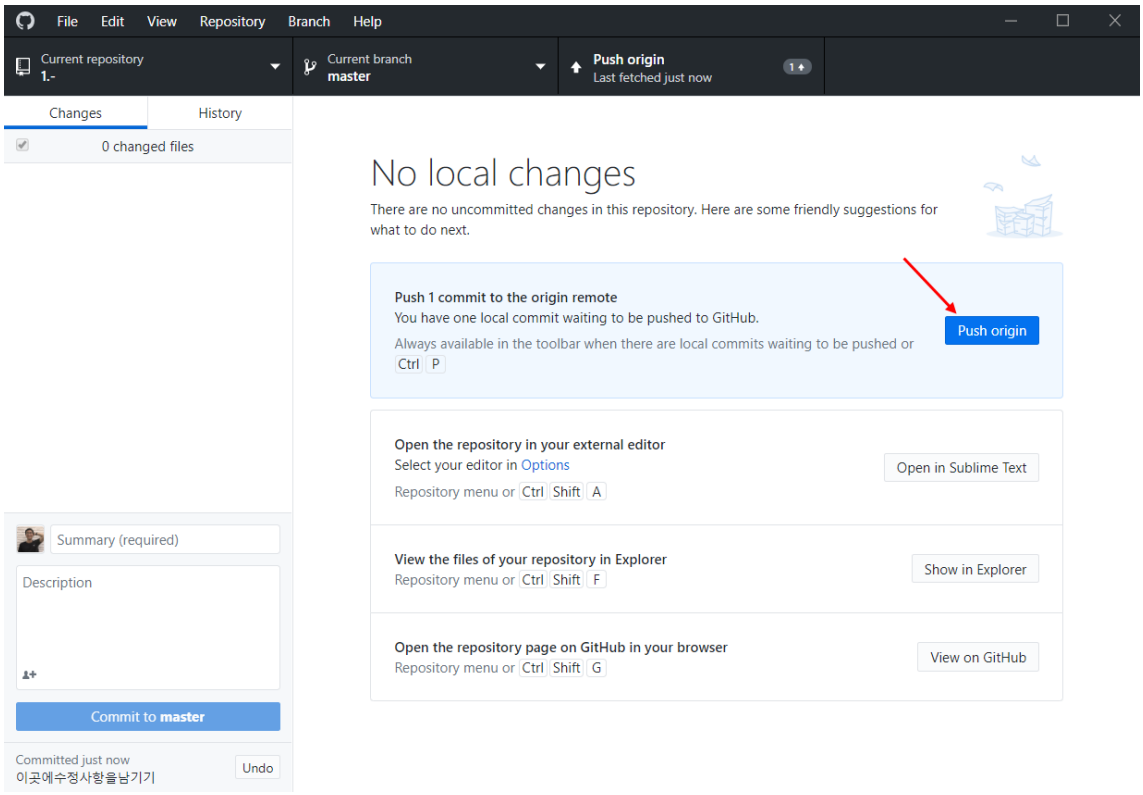


3. 코드를 수정하고, `commit` 하고 `push` 해보기

우리는 작업 내역을 잘 관리하기 위해 Git을 사용하는 거니까,
작업 내역을 알기 쉽게 commit message 를 적어주어야겠죠? (변수/함수 이름 짓기(naming) 처럼 중요하답니다!)

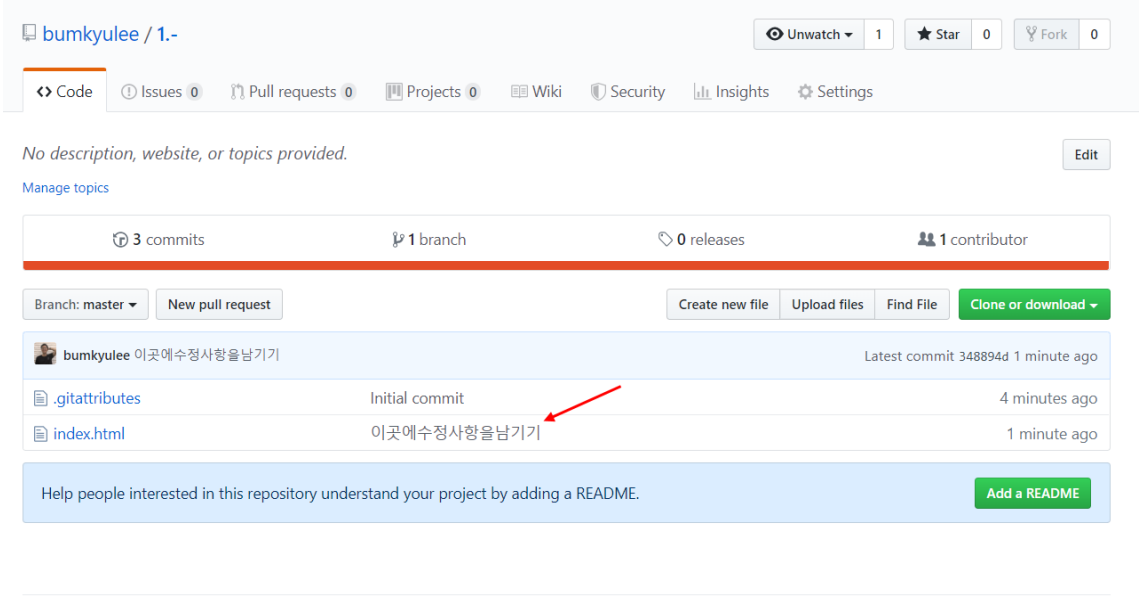
▼ 화면





4. github.com 에 내 원격 저장소 페이지에서 코드가 바뀐 것 확인

    ▼ 화면



09. 강의 끝 & 앞으로의 숙제!!! 고생 많으셨습니다 💝

▼ 10) 앞으로의 숙제

👍 여러분, 정말 고생 많으셨습니다!!

여기까지 오신 여러분에게 박수 한 번 쳐드리고 싶습니다.

5주차 과정동안 어려운 적도, 힘든 적도 많았을텐데

이렇게 훌륭하게 완수해내신 것 만으로도 본인에게 자랑스러우셨으면 좋겠습니다.

앞으로, 여러분들은 혼자 알고리즘을 헤쳐나갈 수 있는 자격증이 생겼습니다!

이제 여러분들이 알고리즘을 풀어나가실 수 있는 사이트들을 추천드리겠습니다.

아래 무료 알고리즘 사이트들을 직접 풀어보시면서,

풀이 방법을 우리가 만든 깃허브에 조금씩 올려보자구요!

1. 프로그래머스(https://programmers.co.kr/)

2. 코드시그널(https://app.codesignal.com/interview-practice)

3. 해커랭크(https://www.hackerrank.com/dashboard)

여러분들께 이 수업이 알고리즘 공부를 위한 하나의 통로가 되었으면 합니다.

지금까지 들어주셔서 감사합니다!

- 스파르타 알기쉬운 알고리즘 튜터 박현준 올림  -