스택/큐

같은 숫자는 싫어

- 주어진 배열 arr에서 연속적으로 나타나는 숫자는 하나만 남기고 전부 제거한다
  - • arr = [1, 1, 3, 3, 0, 1, 1] 이면 [1, 3, 0, 1] 을 return 합니다.

```
def solution(arr):
    answer = []

    for item in arr:
        if len(answer) != 0 and answer[-1] == item:
            continue
        answer.append(item)

    return answer
```

▼ 파이썬

- 위의 코드에서 answer은 초기에 빈 배열이기 때문에 answer[-1]을 하면 인덱싱오류가 남
- 따라서 answer[-1:] 을 해서(슬라이싱) 리스트를 만들어 비교할 수 있다

```
if answer[-1:] == [item]:
  continue
```

올바른 괄호

- • "()()" 또는 "(())()" 는 올바른 괄호입니다.
- 주어진 문자열 s가 올바른 괄호이면 return true

```
def solution(s):
    answer = True
    temp = []

    for char in s:
        if char == "(":
            temp.append(char)
        if char == ")":
            if temp[-1:] == ["("]:
                temp.pop()
            else:
                return False

    if len(temp) != 0 :
        return False

    return answer
```

▼ 파이썬

- ternary operator (삼항연산자) 에 대해서

```
# 대부분 언어에서는 아래 형태로 지원
[condition] ? [true_value] : [false_value]

# 파이썬
[true_value] if [condition] else [false_value]
```

- ternary operator를 사용하며, 배열을 사용하지 않은 풀이

```
def is_pair(s):
    x = 0
    for w in s:
        if x < 0:
            break
        x = x+1 if w=="(" else x-1 if w==")" else x
    return x==0
```

기능개발

- progresses 는 각 작업의 현재까지 진행도 정보를 담고 있음
  - 작업은 먼저 배포되어야하는 순서대로 담겨있음
- speeds는 각 작업의 진행 속도 정보를 담고 있음
- 각 작업의 개발속도는 모두 다르기 때문에 뒤에 있는 작업이 앞에 있는 작업보다 먼저 개발될 수 있음
  - 이때 뒤에 있는 작업은 앞에 있는 작업이 완료될 때 함께 배포됨
- 각 배포마다 몇개의 작업이 배포되는지에 대한 정보를 담은 배열을 리턴하시오

```
progresses = [93, 30, 55]
speeds = [1, 30, 5]

return = [2,1]
```

```
import math

def solution(progresses, speeds):
    answer = []
    daysleft = []
    temp = []

  # 각 작업 당 걸리는 시간을 계산
    for i in range(0, len(progresses)):
        daysleft.append(math.ceil((100-progresses[i])/speeds[i]))

    crntmax = daysleft[0]

    for days in daysleft:
        # 뒷 작업이 먼저 끝나는 경우를 모두 temp 배열에 넣어놓음
        # 현재 시점에서 가장 오래걸리는 작업(crntmax)보다 더 오래걸리는 작업이 등장했을 경우
        # 그 전까지의 작업들(temp에 저장된 작업들)이 모두 배포될 수 있다는 의미
        if days > crntmax:
            answer.append(len(temp))
            temp.clear()
            crntmax = days
```

```
        temp.append(days)

    if len(temp) > 0:
        answer.append(len(temp))


    return answer
```

▼ 파이썬

- zip

```
q = []
for p, s in zip(progresses, speeds):
```

프린터

```
1. 인쇄 대기목록의 가장 앞에 있는 문서(J)를 대기목록에서 꺼냅니다.
2. 나머지 인쇄 대기목록에서 J보다 중요도가 높은 문서가 한 개라도 존재하면 J를 대기목록의 가장 마지막에 넣습니다.
3. 그렇지 않으면 J를 인쇄합니다.
```

- priorities에는 문서의 중요도가 대기 순서대로 정렬되어있음
- location은 priorities에서 내가 인쇄를 요청한 문서의 위치를 나타냄
- 리턴값 : 내가 인쇄를 요청한 문서가 몇번째로 인쇄되는지

```
from collections import deque
def solution(priorities, location):
    answer = 0

    deq = deque()
    for p in priorities:
        deq.append(p)

    answer += 1
    while len(deq) != 0 :
            if max(deq) == deq[0]: // 현재 문서의 중요도가 가장 높음(현재 문서 인쇄)
                if location == 0: // 현재 문서가 내가 요청한 문서
                    return answer
                else: // 현재 문서가 내가 요청한 문서가 아닌 다른 문서
                    deq.popleft()
                    answer += 1
                    location -= 1
                    continue
            elif max(deq) > deq[0]: // 현재 문서(deq[0])를 보류시키고 deq의 마지막에 넣음
                deq.append(deq[0])
                deq.popleft()
                if location == 0: // 현재 문서가 내가 요청한 문서였다면 내가 요청한 문서의 location을 가장 마지막 인덱스로 수정
                    location = len(deq) - 1
                else: // 현재 문서가 내가 요청한 문서가 아니라면 내가 요청한 문서의 location이 1 당겨짐
                    location -= 1


    return answer
```

- deque를 사용해서 리스트의 가장 왼쪽 아이템을 계속 popleft()할 것임으로 for문으로 전체 아이템을 훑을 필요없이 deq[0](현재 문서) 값만 가지고 max(deq)와 계속 비교하면 된다
- 문서를 인쇄할 때마다 answer += 1

다리를 지나는 트럭

- 이거 좀 잘 푼듯

```
bridge_length : 다리에 올라갈 수 있는 트럭의 최대 개수
                트럭이 다리를 건너는데 bridge_length만큼의 term이 소요된다
weight : 다리가 최대로 견딜 수 있는 트럭무게 합
truck_weights : 트럭 무게 리스트 (배열된 순서대로 다리를 건너야 함)
answer : 모든 트럭이 다리를 건너는데 걸리는 시간(풀이에서는 term으로 표기)
```

- 내가 생각하는 핵심은 다리(bridge_q)에 트럭을 올릴 수 없는 term에는 bridge_q에 popleft() 및 0을 append해서 이미 올라가 있는 **트럭이 앞으로 이동하는 것과 같은 효과**를 내야 한다는 것이다. (머릿속에 그림을 잘 그려보기!)
  - 시작할 때는 bridge_q에 bridge_length만큼 0을 채워놓은 상태로 시작
  - 매 term마다 popleft() & append() 수행
  - wait_q에 더 이상 트럭이 없다면(마지막 트럭이 건너는 중) bridge_q에 0을 append하지 않고, 이미 다리에 올라가 있는 트럭(및 0)을 pop한다
    - bridge_q가 비워지는 시점 = 모든 트럭이 다리를 건너는 시점

```
from collections import deque
def solution(bridge_length, weight, truck_weights):
    answer = 0
    bridge_q = deque()
    wait_q = deque()

    for i in range(0, bridge_length):
        bridge_q.append(0)

    for truck in truck_weights:
        wait_q.append(truck)

    crntweight = 0


    while len(bridge_q) != 0:
        crntweight -= bridge_q[0]
        bridge_q.popleft()

        if len(wait_q) > 0:
            if crntweight + wait_q[0] > weight:
                bridge_q.append(0)
            else:
                bridge_q.append(wait_q[0])
                crntweight += wait_q[0]
                wait_q.popleft()

        answer += 1


    return answer
```

주식 가격

```
prices : 초 단위로 기록된 주식 가격이 담긴 배열
return : 각 초에 해당하는 가격이 떨어지지 않은 기간

ex)
prices = [1,2,3,2,3] 이면
return = [4,3,1,1,0]

- 1초 시점의 가격 1은 끝까지 가격이 떨어지지 않음
- 2초 시점의 가격 2도 ''
- 3초 시점의 가격 3은 1초 뒤에 가격이 떨어짐
```

- prices를 deque에 넣고 popleft()하면서 q[0] 의 값을 그 뒤 값들과 비교한다

```
from collections import deque
def solution(prices):
    answer = []
    q = deque()
    for price in prices:
        q.append(price)

    while len(q) != 0:
        crntprice = q[0]
        q.popleft()

        time = 0
        for price in q:
            time += 1
            if price < crntprice:
                break
        answer.append(time)


    return answer
```