알고리즘 과제 #1

def evaluate\_n(A, x):

renew = x

R = []

R.append(A[0] \* 1)

for i in range(1, len(A)):

R.append(A[i] \* renew)

renew \*= x

def evaluate\_n2(A, x):

    # code for O(n^2)-time function

    L = []

    R = []

    for i in range(len(A)):

        L.append(x)

    L[0] = 1

    for i in range(2, len(A)):

        for j in range(i, len(A)):

            L[j] \*= L[1]

    for i in range(len(A)):

        R.append(A[i] \* L[i])

random.seed()

n = int(input())

A = []

for i in range(n):

    A.append(random.randint(-999, 999))

before\_n2 = time.clock()

evaluate\_n2(A, n)

after\_n2 = time.clock()

before\_n = time.clock()

evaluate\_n(A, n)

after\_n = time.clock()

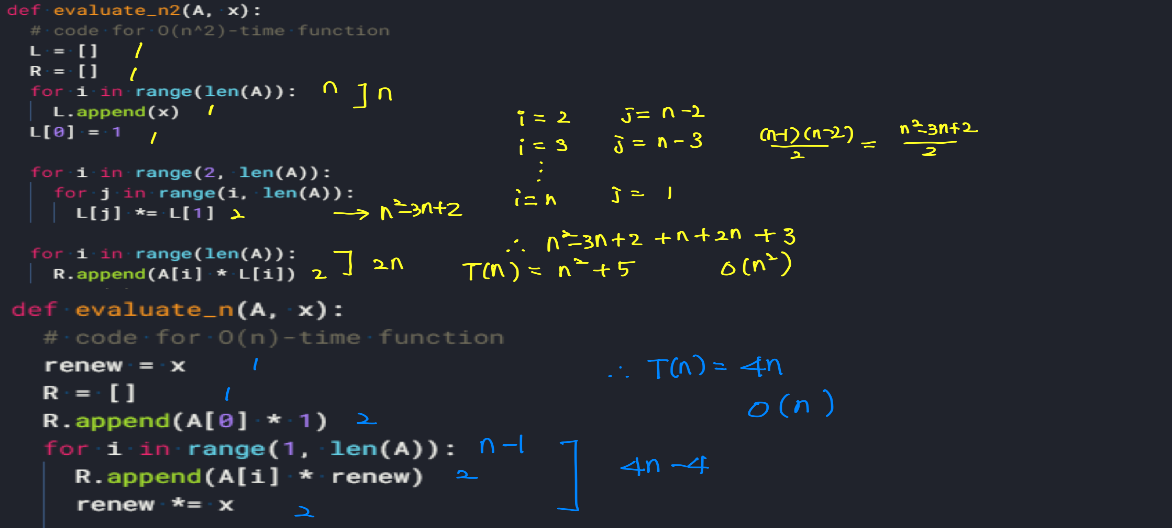
print("evaluate\_n2의 수행 시간: ", (after\_n2 - before\_n2))

print("evaluate\_n의 수행 시간: ", (after\_n - before\_n))

의 시간으로 돌아가는 함수에서 L 리스트에 x값을 n번만큼 반복해서 넣었습니다. 그리고 L의 첫번째 인덱스의 값은 1로 변경했습니다. 어차피 이어서 1로 변경해도 문제없다고 생각했기 때문입니다. 그런 다음 이중 반복문을 사용했습니다. 이중 반복문의 목적은 x의 값을 갱신하기 위해 사용됩니다.

인덱스 2부터 시작하는 이유는 인덱스 0은 이미 1이며, 인덱스 1은 L리스트에 넣은 x값이 그대로 사용되기 때문에 값을 갱신할 필요가 없기 때문입니다. 예를 들어 I = 2 일 때, j = 2 부터 n까지 L[2] \*= L[1] #L[1] = x 를 반복해서 값을 갱신합니다. 그 다음 반복문은 갱신된 x값을 저장해 놓은 L리스트와 계수가 저장 되어있는 A리스트의 값을 곱하는 역할을 합니다.

의 시간으로 돌아가는 함수의 원리는 이전에 곱해지는 x의 값을 기억하는 것입니다. 갱신되는 x의 값을 renew라고 정했습니다. R은 결과값들을 저장해 놓을 리스트입니다. 먼저 R리스트에 A의 첫번째 인덱스에 1을 곱한 값을 추가합니다. 그리고 A의 두번째 리스트부터는 A[i] \* renew를 하며 곱하고 난 다음에 renew의 값을 x를 곱해 갱신합니다.



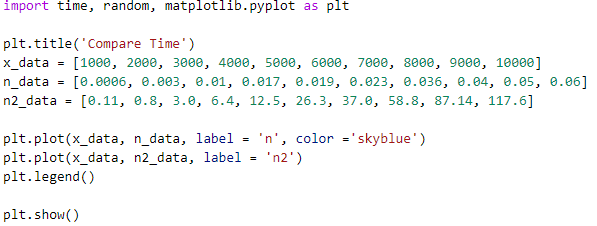
두 함수의 수행시간을 계산해보면 위와 같습니다. Evaluate\_n2의 수행시간은 이며 입니다. Evaluate\_n의 수행시간은 이며 입니다.

밑의 사진은 두 함수의 수행시간을 그래프로 시각화한 자료입니다.

스크린샷이(가) 표시된 사진

자동 생성된 설명

이 자료는 vs에서 matplotlib.pyplot as plt 를 임포트해서 구현했습니다.



X\_data는 1000부터 10000까지 넣었습니다. N\_data와 n2\_data는 각각 x값에 대응하는 y값, 즉 함수의 수행시간입니다. N2와 n모두 x값에 비례하여 값이 증가하지만 증가폭이 n보다 n2가 훨씬 더 높다는 것을 알 수 있습니다.

알고리즘을 어떻게 하느냐에 따라서 수행시간의 차이가 굉장히 크다는 것을 다시 한번 실감하게 되었습니다. 간단히 O(n)으로 만들 수 있는 문제를 엮고 엮어서 O(n2)로 만든다는 것이 조금 복잡했습니다. 알고리즘을 짜면서 어떤 방법을 하나 떠올리면 다른 새로운 방법을 찾기가 어려웠습니다. 새로운 시각으로 바라봐야 하는데 하나에 박혀있어서 그런 것 같습니다. 시점문제는 예전부터 인지하고 있었지만 고치기가 제일 어려운 것 같습니다.