이름: 정재윤 , 학번: 201703262 , 학과: Global Business & Technology학과

과제: #1 유일성 문제 – 수행시간 분석

함수 1 : unique_n2(A): #O(n^2)

```
def unique_n2(A):#O(n^2)
    answer="*"#답을 출력할 변수 Answer 지정
    for i in range(len(A)):
        for j in range(len(A)):
            if i!=j:
                if A[i]==A[j]: #list A 에 동일원소가 한쌍이라도 존재 -> No 저장
                    answer="No"
            else: #list A 내의 모든 원소가 다를 경우 -> Yes 저장
                answer="Yes"
    return answer #결과값 출력
```

함수의 수행시간 : O(n^2)

함수 2 : unique_nlogn(A): #O(nlogn)

```
def unique_nlogn(A):#O(nlogn)
    A.sort() #입력받은 리스트 A 의 원소를 정렬
    answer="Yes"#answer 의 기본값은 Yes 로 지정
    for i in range(len(A)): # 각 원소 수만큼의 i 를 호출
        if i < len(A)-1:
            if A[i]==A[i+1]:
                answer="No"
        else:
            break
    return answer
```

함수의 수행시간 : O(nlogn)

함수 3 : unique_n(A): #O(n)

```
def unique_n(A): #O(n)
    B=[] #A 의 원소를 저장할 새로운 list B 생성
    answer="Yes"#기본 answer 값은 Yes 로 설정
    for i in A: # i!=B 인 경우(중복 원소가 없는 경우)
        if i != B:
            B.append(i)
        else: #i==B 인 경우(중복 원소가 있는 경우)
            answer="No" #중복 원소가 1 개라도 존재하면 No 로 answer 의 변수 변경
    return answer
```

함수의 수행시간 : O(n)

이름: 정재윤 , 학번: 201703262 , 학과: Global Business & Technology학과

3개 함수의 실행과 수행시간 분석 :

명령 실행 코드:

```
n = int(input('측정할 n의 값을 입력하시오: ')) # input: 값의 개수 n
A = random.sample(range(-n, n+1), n)
# -n과 n 사이의 서로 다른 값 n 개를 랜덤 선택해 A 구성
###########시간측정&각 함수 실행###########
s1=time.process_time()
result1= unique_n2(A)
e1= time.process_time()

s2=time.process_time()
result2= unique_nlogn(A)
e2= time.process_time()

s3=time.process_time()
result3= unique_n(A)
e3= time.process_time()

#육안으로 쉽게 판별할 수 있도록 각 수행시간에 임의의 수 10000을 더하였습니다.
time1=(e1-s1)+10000
time2=(e2-s2)+10000
time3=(e3-s3)+10000

print(result1 ,result2, result3)
print('<수행시간>')
print('unique_n2(A) = ',time1)
print('unique_nlogn(A) = ', time2)
print('unique_n(A) = ', time3)
```

각 함수 별 수행시간 분석:

함수 별 수행시간

n 값	unique_n2(A)	unique_nlogn(A)	unique_n(A)
10	1.3347999999999832e-05	6.072000000002797e-06	4.367000000001647e-06
20	5.929600000000035e-05	1.2727999999996575e-05	7.484000000020446e-06
50	0.0001833520000000012	1.3610000000000705e-05	8.628999999999581e-06
100	0.0006699459999999977	2.6350999999997377e-05	1.1863000000000845e-05
1000	0.000130734999999999	0.000456799	0.000131856999999999

* time.process_time()함수를 이용하여 함수의 수행시간을 구할 경우, 위와 같이 일반적인 숫자표기로서 표시가 되지 않는 문제가 발생하여, 시간을 비교/측정하여 그래프를 그릴 때는 각 함수의 수행시간+10,000을 처리한 뒤 비교하였습니다.

알고리즘(월123)

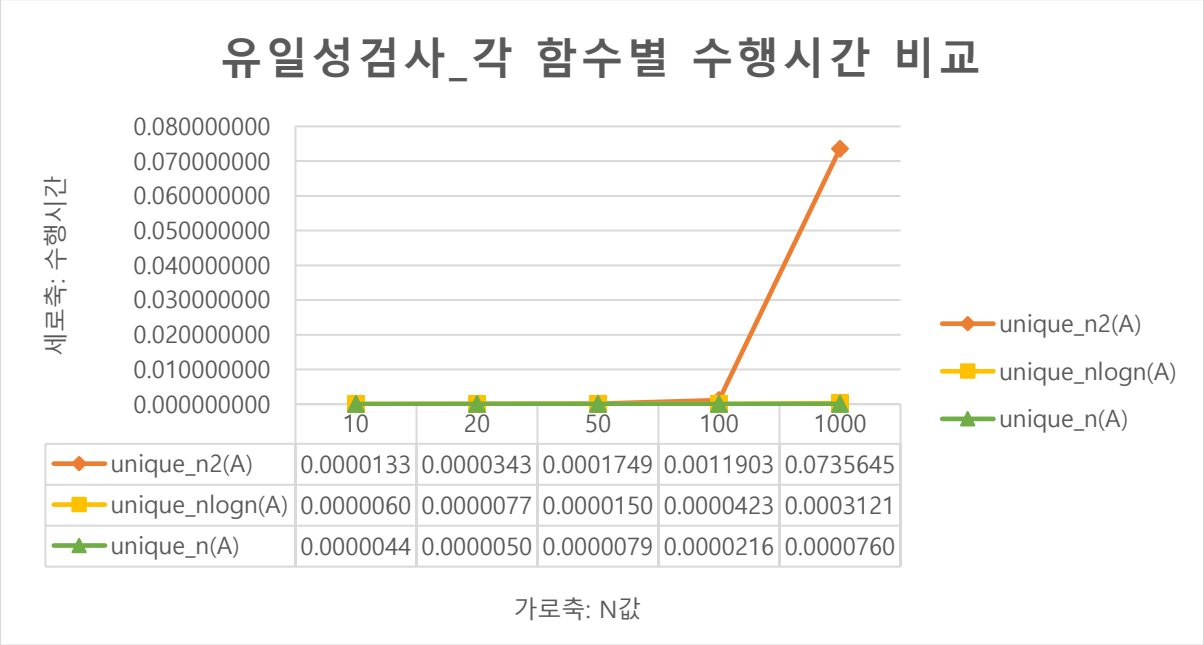
이름: 정재윤 , 학번: 201703262 , 학과: Global Business & Technology학과

육안으로 쉬운 비교를 위한 대안으로 각 함수 별 수행시간+10000

함수 별 수행시간

n 값	unique_n2(A)	unique_nlogn(A)	unique_n(A)
10	10000.000013382	10000.000006050	10000.000004422
20	10000.000034377	10000.000007713	10000.000005051
50	10000.000174967	10000.000015021	10000.000007996
100	10000.001190382	10000.000042395	10000.000021624
1000	10000.073564579	10000.000312137	10000.000076046

(함수 별 수행시간 비교 그래프는 +10,000수치를 제외 처리하였습니다.)



결론:

unique_n2(A)는 big-O가 n의 제곱만큼 증가하는 함수의 수행시간을 갖습니다. 따라서 동일한 n 값을 갖더라도 수가 커짐에 따라 타 함수(unique_nlogn(A) 또는 unique_n(A))에 비해 수행시간이 급격하게 증가하는 모습을 확인할 수 있습니다. 이와 달리 unique_nlogn(A)과 unique_n(A)은 비교적 n의 값에 따른 수행시간의 증가율이 완만한 모습을 보이며, Big-O의 값이 nlogn을 갖는 unique_nlogn(A)이 Big-O의 값이 n을 갖는 unique_n(A)보다 근소하게 수행시간이 더 큰 모습을 보였으며 증가 폭 또한 비교적 근소한 차이로 높다는 것을 확인할 수 있었습니다.

동일한 기능을 수행하는 함수에 대해 3가지 다른 수행시간을 갖는 구조의 알고리즘을 구현해보고 이를 그래프로 도출해본 결과, 같은 기능을 수행하더라도 입력 값(input값으로 지정 받은 n)이 커질 수록 Big-O에 따른 효율적인 알고리즘과 비효율적인 알고리즘 간 코드 동작 속도에 차이가 발생한다는 점을 확인할 수 있었으며 효율적코드의 중요성을 체감할 수 있었습니다.