

## 4주차 과제 / 최근접 쌍 탐색

### 1. 산점도 그리기

구글링을 통해 matplotlib의 pyplot에 들어 있는 scatter() 함수와 랜덤 수 생성 함수를 이용해서 쉽게 그릴 수 있다는 걸 알았다. 코드와 산점도는 아래와 같다.

```
import matplotlib.pyplot as plt
import numpy as np

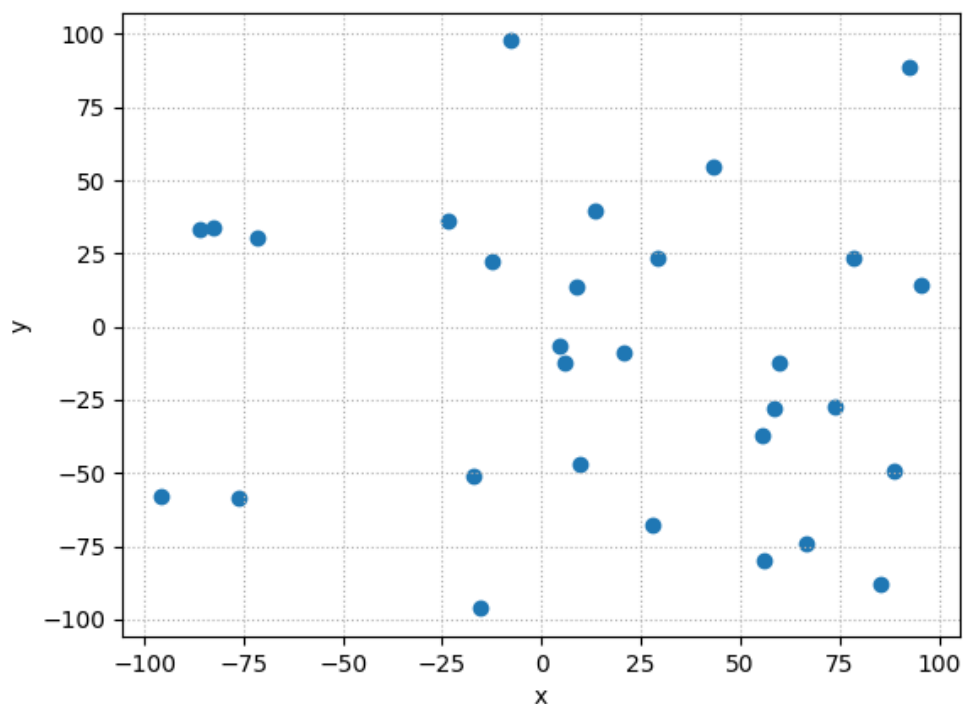
np.random.seed(0) # 무작위 수 생성, c에서 srand(time(null)) 하던거랑 비슷한
역할인듯..?

n = 30
x = np.random.rand(n) * 200 - 100
y = np.random.rand(n) * 200 - 100
# 30 개의 점을 생성

# np.random.rand()는 [0, 1)의 숫자를 생성한다고 한다 -> 200을 곱하고 100을 빼서
범위를 맞췄다.

plt.scatter(x, y) # 산점도를 그려주세요!

plt.xlabel('x')
plt.ylabel('y')
plt.grid(True, linestyle = ':')
plt.show()
```



## 2. 완전탐색과 분할정복을 사용하여 거리 도출하기

우선 컴퓨터가 괴로워하는 완전탐색을 구현해 보았다. 완전탐색에도 종류가 엄청 많았는데, 수업시간에 언급된 브루트포스 방식을 찾아서 작성했다. 정말 무식하게 다 돌아보는 방식이었다.

아래는 코드인데 코드를 보면 아우터 루프로  $n-1$ 번 돌아가는데 이너루프는  $i+1$ 부터  $n$  까지 돌아가고 있다. 시간복잡도는  $O(n^2)$ 이라고 할 수 있다.

```
def bruteforce(p):
    n = len(p)
    min_d = float('inf')
    for i in range(n-1):
        for j in range(i+1, n):
            d=calc_dist(p[i], p[j])
            if d < min_d:
                min_d = d
    return min_d
```

모든 경우의 수를 탐색하기 때문에  $30^2$ , 즉 900 번의 연산을 한다. 코드를 짜기 쉽고 뭐하는 건지 바로바로 이해가 되지만, 성능 면에서는 떨어진다고 볼 수 있겠다.

두번째는 수업시간에 구현하다가 만 분할정복 방식의 코드이다.

혼자 해보려다가 도저히 못하겠어서 무한 티스토리 탐색을 한 후, 로직을 살펴본 뒤에 코드를 작성했다. 그런데 실패했다.

### 1. 실패한 ver

```
def closest_pair(x):
    if len(x) <= 3:
        bruteforce(x)

    mid = x[len(x) // 2] # 중앙점을 mid에 넣어두자

    cpl = closest_pair([s for s in x[0:len(x)//2]])
    cpr = closest_pair([s for s in x[len(x)//2:]] ) # 재귀적으로 중간값을
    기준으로 나누기
    d = min(cpl, cpr)

    # cpc를 찾아보자
    strip = [p for p in x if abs(p[0] - mid[0]) < d]
    # 중간 부분에서 d보다 작거나 같은 애들을 찾기
    # p[0]는 p의 x좌표, abs()는 절댓값 -> x좌표 차이의 절댓값이 d보다 작은 애들만
    넣어두는 리스트

    # strip에 저장된 애들 중 제일 작은거 찾는 함수
    cpc = float('inf')
```

```

strip.sort(key=lambda point: point[1]) # y좌표에 따라 정렬, lambda는 익명
함수(소트 하려고 쓰는 함수)

for i in range(len(strip)): # 아우터 루프는 strip를 돌고
    for j in range(i + 1, len(strip)): # 이너루프는 i+1부터 끝까지 돌면서
        if strip[j][1] - strip[i][1] < d: # x좌표 차이를 찾아본다 -> 그게
d보다 작다면

            distance_ij = calc_dist(strip[i], strip[j]) # y좌표까지 고려한
진짜 거리를 찾아보고

            cpc = min(cpc, distance_ij) # 현재 설정된 중앙 최소거리와 비교하여
작은 값을 업데이트(첫번째는 inf)

return min(d, cpc) # 최종적으로 d와 cpc를 비교하여 리턴

```

내장된 sort 함수를 이용해서 y 좌표를 정렬하고 나름대로 재귀호출도 해보고 했는데, 아래와 같은 오류 메시지가 떴다.

```

Traceback (most recent call last):
  File "C:\computer algorithm\pythonProject\4주차 과제.py", line 60, in <module>
    print(closest_pair(p))
    ~~~~~
  File "C:\computer algorithm\pythonProject\4주차 과제.py", line 38, in closest_pair
    cpl = closest_pair([s for s in x[0:len(x)//2]])
    ~~~~~
  File "C:\computer algorithm\pythonProject\4주차 과제.py", line 38, in closest_pair
    cpl = closest_pair([s for s in x[0:len(x)//2]])
    ~~~~~
  File "C:\computer algorithm\pythonProject\4주차 과제.py", line 38, in closest_pair
    cpl = closest_pair([s for s in x[0:len(x)//2]])
    ~~~~~
[Previous line repeated 2 more times]
  File "C:\computer algorithm\pythonProject\4주차 과제.py", line 36, in closest_pair
    mid = x[len(x) // 2] # 중앙점을 mid에 넣어두자
    ~~~~~
IndexError: list index out of range

```

리스트 레인지가 너무 큰가..., 아무튼 범위를 벗어났다고 했다. 아니 대체 왜? 절반씩 잘라서 잘 넣은 것 같은데 왜 안 되는 건지 모르겠어서 다시 구글링을 무한으로 했다. 로직도 티스토리 글이랑 비슷한 것 같고... 복붙을 해서 돌려야 하나 고민이 되었다. 하지만 양심상 그럴 수가 없었다.

2. 성공한 ver : 챗지피티와 함께

챗지피티는 이걸 깔끔하게 해결해 주었다.



### ChatGPT

업로드하신 스크린샷에 따르면 `IndexError: List index out of range` 오류가 발생했습니다. 이 오류 메시지는 주로 리스트에서 존재하지 않는 인덱스를 참조하려 할 때 발생합니다. 코드에서 `'closest_pair'` 함수를 재귀적으로 호출하는 부분에서 이 문제가 발생하고 있는 것 같습니다.

이 오류가 발생한 주된 이유는 리스트를 분할할 때, 분할된 서브리스트의 범위 지정이 올바르지 않아서일 수 있습니다. 예를 들어, 리스트의 길이가 홀수이고 중간 인덱스(`mid`)를 사용하여 리스트를 나눌 때, 중간 인덱스가 한쪽 서브리스트에 포함되지 않아서 리스트의 길이가 예상보다 짧아질 수 있습니다.

그러니까 이 부분이 문제였다.

```
mid = x[len(x) // 2] # 중앙점을 mid 에 넣어두자

cpl = closest_pair([s for s in x[0:len(x)//2]])
cpr = closest_pair([s for s in x[len(x)//2:]] ) # 재귀적으로 중간값을 기준으로
나누기
d = min(cpl, cpr)
```

중앙값을 제대로 잘랐다고 생각했는데 인덱싱에 문제가 있었던 것이다. 그러니까 `mid` 가 리스트가 아니라 `x` 좌표 '값' 이었다. 바보...인가...

```
strip = [p for p in x if abs(p[0] - mid[0]) < d]
```

이게 틀렸던 것이다.

아래는 챗지피티가 수정해준 버전이다.

## 2. 성공한 ver

```
def closest_pair(x):
    if len(x) <= 3:
        return bruteforce(x)

    mid_idx = len(x) // 2 # 중앙점을 mid 에 넣어두자
    mid = x[mid_idx][0]

    cpl = closest_pair([s for s in x[0:len(x)//2]])
    cpr = closest_pair([s for s in x[len(x)//2:]] ) # 재귀적으로 중간값을
기준으로 나누기 -> mid 값이 스택에 착착 쌓임
    d = min(cpl, cpr)

    # cpc 를 찾아보자
    strip = [p for p in x if abs(p[0] - mid) < d]
    # 중간 부분에서 d 보다 작거나 같은 애들을 찾기
    # p[0]는 p 의 x 좌표, abs()는 절댓값 -> x 좌표 차이의 절댓값이 d 보다 작은 애들만
    넣어두는 리스트
```

```

# strip에 저장된 애들 중 제일 작은거 찾는 함수
cpc = float('inf')
strip.sort(key=lambda point: point[1]) # y좌표에 따라 정렬, lambda는 익명
함수(소트하려고 쓰는 함수)

for i in range(len(strip)): # 아우터 루프는 strip를 돌고
    for j in range(i + 1, len(strip)): # 이너루프는 i+1부터 끝까지 돌면서
        if strip[j][1] - strip[i][1] < d: # x좌표 차이를 찾아본다 -> 그게
d보다 작다면

            distance_ij = calc_dist(strip[i], strip[j]) # y좌표까지 고려한
진짜 거리를 찾아보고

            cpc = min(cpc, distance_ij) # 현재 설정된 중앙 최소거리와 비교하여
작은 값을 업데이트(첫번째는 inf)

return min(d, cpc) # 최종적으로 d와 cpc를 비교하여 리턴

```

잘 돌아갔다!

## 출력 결과

```

59 p = np.random.rand(30, 2) * 200 - 100
60
61 print(bruteforce(p))
62 print(closest_pair(p))

```

4주차 x 4주차 과제 x

```

"C:\computer algorithm\pythonProject\venv\Scripts\python.exe" "C:\computer algorithm\pythonProject\4주차 과제.py"
5.9983035551405175
5.9983035551405175
Process finished with exit code 0

```

```

"C:\computer algorithm\pythonProject\venv\Scripts\python.exe" "C:\computer algorithm\pythonProject\4주차 과제.py"
8.20070154075324
8.20070154075324

```

```

"C:\computer algorithm\pythonProject\venv\Scripts\python.exe" "C:\computer algorithm\pythonProject\4주차 과제.py"
5.75040158773412
5.75040158773412

```

그런데 애들은 왜 이런 걸까?

```

5.027364306423113
5.977805986153172

```

```

4.413517986620378
11.054759107742026

```

같은 테스트 케이스에 대해 다른 값이 나왔다. 바로 챗지피티에게로 달려갔다



ChatGPT

`closest\_pair` 함수와 `bruteforce` 함수가 다른 결과를 반환할 수 있는 주된 이유 중 하나는 `closest\_pair` 함수가 정렬된 입력을 기반으로 작동하도록 설계되었기 때문입니다. 하지만, 여기서 주의해야 할 중요한 점이 있습니다.

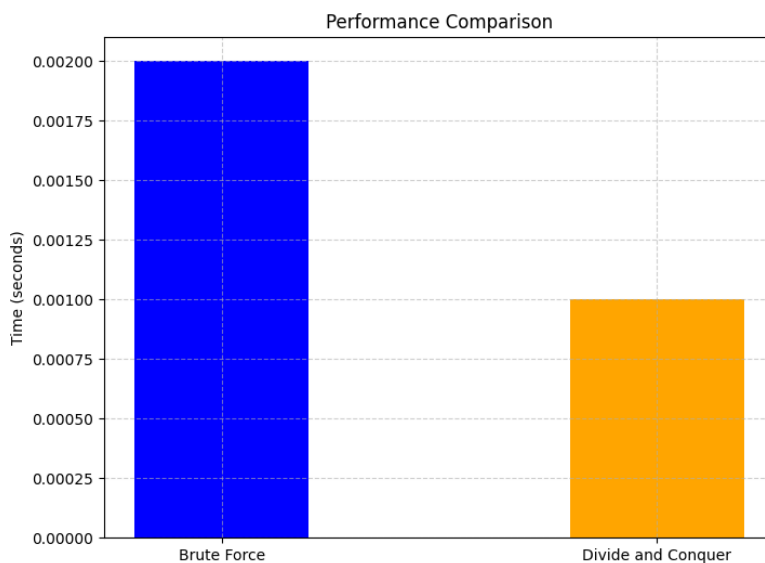
- `np.sort(p, axis=0)`를 사용하여 `closest\_pair` 함수에 전달하기 전에 `p`를 정렬할 때, 이는 `p`의 모든 점을 **x 축과 y 축에 따라 독립적으로 정렬**합니다. 이는 `closest\_pair` 알고리즘에 필요한 것과 다릅니다. `closest\_pair` 알고리즘은 **x 좌표에 따라만 정렬된** 입력을 요구합니다. 이로 인해 `closest\_pair` 함수 내에서 불일치가 발생할 수 있습니다.

`closest\_pair` 알고리즘은 분할 정복 방식으로 점들 사이의 최소 거리를 찾으려고 할 때, **x 축에 따라 정렬된 상태**를 기반으로 가정합니다. 그래서 `np.sort(p, axis=0)` 대신에, 우리는 `p`를 **x 축에 따라 정렬**해야 합니다. 그렇게 하려면 다음과 같이 해야 합니다:

사실... 무슨 말인지 모르겠다... 수정해준 코드를 쓰니까 해결되긴 했는데 왜 되는지 모르겠다. ㄱㄱ...

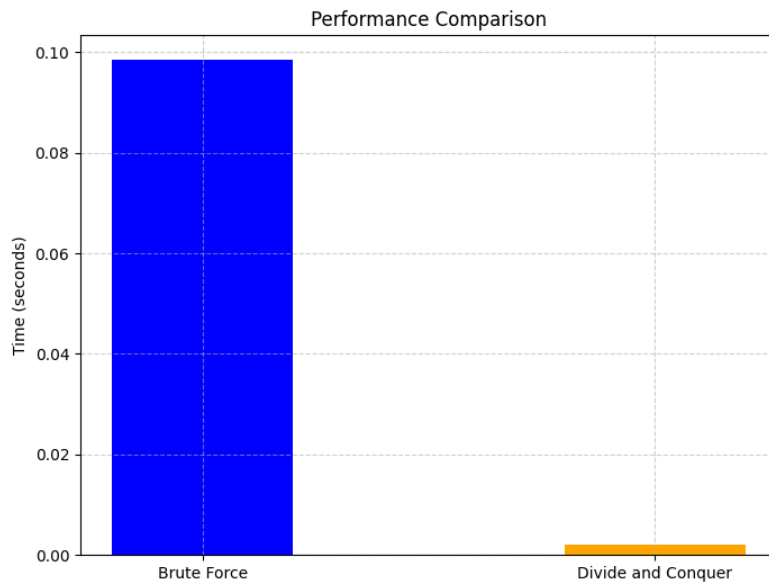
### 성능 비교(속도 비교)

점의 개수가 늘어나거나 하지는 않다보니 라인그래프보다는 막대 그래프가 좋겠다 싶었다. 그리는 방법은 비슷했다. 아래는 성능 비교 그래프이다.

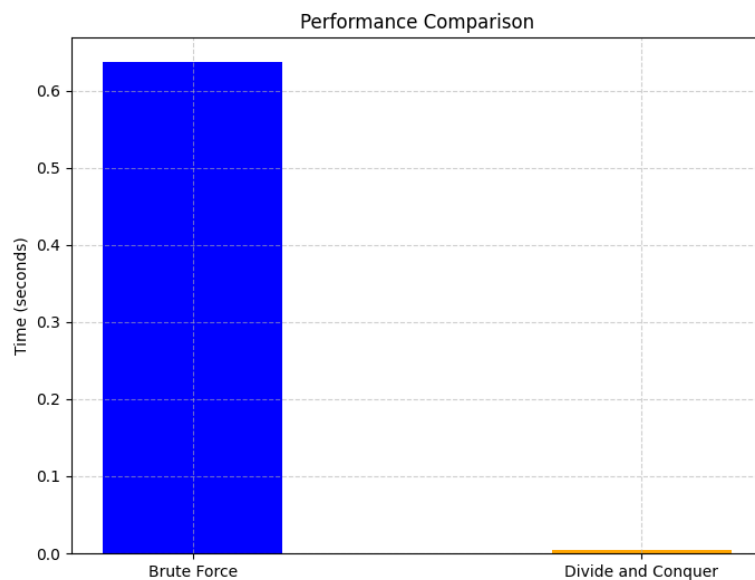


30 개로 하니까 시간이 엄청 짧아서 그런지 빈 그래프가 뜨길래 150 개로 바꿨다.

개인적으로 궁금해서 점 개수를 400 개로 올려보았다. 당연하게도 브루트포스와 분할정복의 차이가 두드러지게 나타났다.  $O(n^2)$ 과  $O(n\log n)$ 이기 때문이다.



1000 개

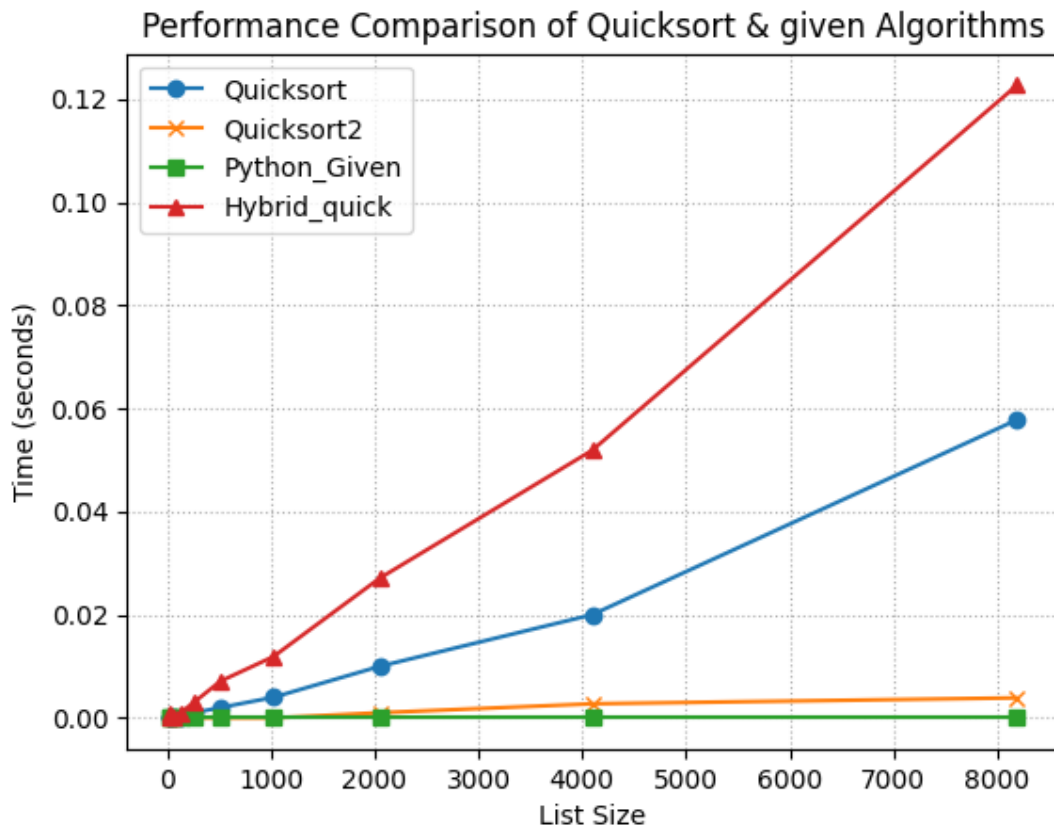


더 올려보기도 했는데 분할정복방식의 그래프가 사라졌었다. 그리고 브루트포스 때문인지 노트북 쿨링팬이 미친듯이 돌고 있다.

결론: 성능은 분할정복방식이 압도적이다. 하지만 구현하기는 너무 어려웠다... ㅏㅏ...

#### 4주차 과제 / 개인 공부

과제 후에 조금 성의가 없나 싶기도 하고 아쉬워서 나름대로 퀵정렬에 대해 공부해 보았다.



빨간게 퀵+삽입 버전인데(코드는 깃허브에서 주웠다.) 무슨 일인지 모르겠지만 효율이 안 좋게 나왔다. 왜 그런가를 알아보기 위해 구글의 도움을 받았다.

우선 이 현상에 대해 적어둔 깃허브 글을 찾았다.

\* [https://github.com/yarnabrina/Hybrid-Sort/blob/master/report\\_in\\_markdown.md](https://github.com/yarnabrina/Hybrid-Sort/blob/master/report_in_markdown.md)

퀵+삽입 정렬 알고리즘에 대한 성능 분석 그래프를 볼 수 있었다. x축을 임계값으로 설정하여 배열 사이즈별로 그래프 몇 개가 나와 있는데, 이에 대해선 U자 그래프가 나온다.

분석글을 복붙하자면,

Firstly, we see that there is a sharp fall in all the graphs initially. This proves the effectiveness of the hybrid algorithm over quick sort, as it should be noted that for the choice of cut-off as 1, we are essentially applying quick sort over the entire array. So that steep fall helps us to conclude with confidence that combining the two algorithms is not at all worthless. This is because of the fact that as quick sort is a



recursive algorithm, it has a too much of overhead cost for calling itself repeatedly for small arrays.

Secondly, we note that after a certain point, average run-time has a steadily increasing trend, which is due to the fact that insertion sort is effective only for "small" arrays. As we are increasing the cut-off size, insertion sort needs to be applied on larger partially sorted sub-arrays and hence the sorting of the entire array becomes slower.

Finally, we observe that the trade-off between these two opposite effects on run-time is balanced in the lower part of the skewed U-shaped pattern, which is revealed in all the graphs, in more or less extent.

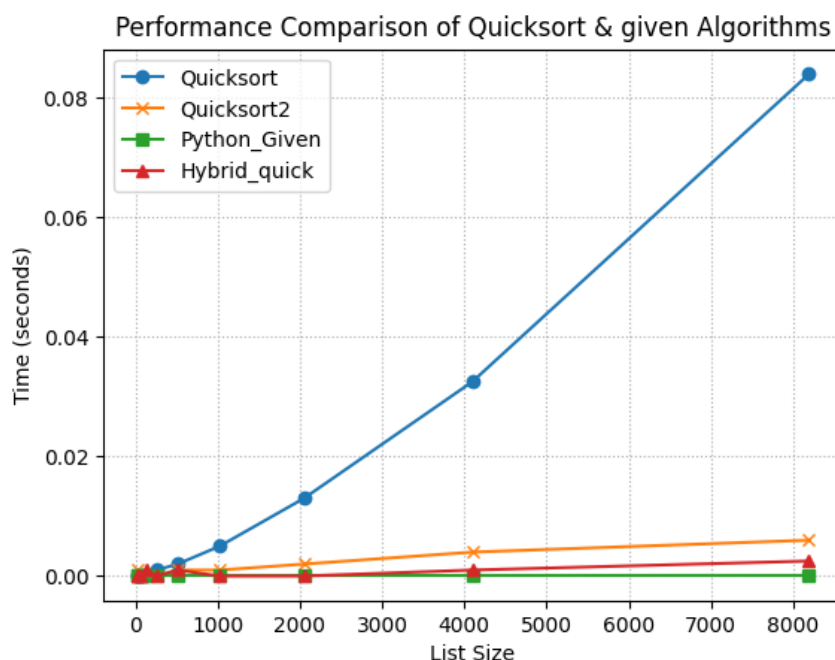
1. 임계값이 1로 설정되었을 때, 그러니까 퀵소트와 다른 없을 때보다, 임계값을 올렸을 때 성능이 향상된다.

2. 임계값을 올리다보면 어느 순간부터는 성능이 나빠진다. 인서션 소트 자체가 큰 배열에는 나쁘기 때문이다.

3. U자 그래프의 낮은 쪽(임계값 크기가 작은 쪽을 말하는 듯)에서 최적화된 포인트를 찾을 수 있다.

-> 결론은 좋은 정렬방법이지만, 적절한 임계값을 잘 찾아야 한다 이거인듯.

임계값을 잘 찾으면 해결될까 싶어서 해봤다. 좋은 임계값을 찾기 위해 컴퓨터를 고문했다.



성능이 꽤나 괜찮다. 근데 배열 크기를 줄이거나 늘리면 다시 가파른 우상향 그래프를 그렸다. 길어 질까봐 첨부은 안 하지만, 임계값을 비례하여 줄이고 늘려주면 다시  $O(n \log n)$  모양의 일자 그래프를

그렸다. 신기! 재밌구만.

그래서 초록 그래프는 뭐길래 상태가 제일 좋냐?

나는 파이썬에서 제공하는 sort가 퀵소트인줄 알았다. 그래서 수업시간에 배운 두 가지의 퀵소트와 성능을 비교해보려고 했다. 하지만 알아보니 파이썬 내장인 sort와 sorted는 timsort라고 불리는 머지소트+인서션소트 였다.

여기서 촉발된 내 궁금증은 퀵이나 머지나  $O(n \log n)$ 인 분할정렬인건 똑같지 않나? 였다. 그래서 역시 구글링을 해 보았다.

1. Quora에서 찾은 질문 : 두개 이상의 정렬을 사용하는 하이브리드 소트를 이용할 때, 머지와 퀵을 어떻게 이용하시나요

\* <https://www.quora.com/How-do-you-implement-a-hybrid-sort-of-QuickSort-and-MergeSort-proletariat-c-algorithm-assembly-mergesort-quicksort-solutions>

답변을 조금 발췌하자면

Mergesort is most useful when your data is in a linked list (where quicksort is not directly applicable), or in a file and is too large to fit into memory. I have dealt with both situations. (...) However, If you take the original list (or file) and sort 1024 records at a time using quicksort, then you can avoid 10 of the 30 passes, and would see a significant speed improvement. The larger the block that you pre-sort, the faster your overall sort will be.

요약하자면 머지소트는 큰 데이터에서도 괜찮은 퍼포먼스를 낸다는 거고, 퀵소트는 리스트 크기가 작을 때(위에 깃허브 글과 같이), 혹은 어느정도 정렬이 된 상태의 배열에서 엄청난(significant) 속도차이를 보여준다는 거다.

2. **geeksforgeeks**에서 찾은 퀵 vs 머지

하이브리드 말고 퀵과 머지만의 차이가 궁금해서 알아봤다.

\* <https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/>

사이트에 나와 있는 테이블을 보면 차이가 엄청 많다. 내가 궁금했던 것만 요약하자면,

1. 퀵소트는 작은 배열을 정렬할 때 효율적, 반대로 머지소트는 어느 크기나 상관없이 없다.

-> 최악의 경우를 비교할 때, 머지소트가 압도적임

2. 퀵소트는 인플레이스 방식, 머지소트는 아니다. 추가적인 스토리지가 필요하다.

아무튼 원래대로 돌아와서, timsort는 머지소트가 상대적으로 느린 점을 보완하기 위해 인서션 소트를 결합한 형태의 알고리즘이다. 때문에 가장 좋은 퍼포먼스를 보이며, 파이썬뿐만이 아니라 자바나 다른 언어에서도 내장 메소드로 자주 사용이 된다.

### 퀵 vs 퀵의 성능차이

이건 왜 그러는건지 모르겠다. 수업시간에 배운 퀵 2가지 버전을 테스트해본 건데, 왜 첫번째건 저렇게 구리고 두번째건 꽤 괜찮게 나오는 걸까? 재귀적인것도 똑같고, 표현의 차이이지 반복문 도는 것도 똑같은데... 음...

어쩌다가보니 길어졌는데 퀵과 인서션을 결합하려면 임계값을 잘 움직여야 한다는 걸 알게 된게 전부인것 같다.