사조참치

박정환 범수 이신행 이창윤 윤태원

-사전 공지-

```
import FinanceDataReader as fdr
```

	Symbol	Name
0	AAPL	Apple Inc
1	MSFT	Microsoft Corp
2	AMZN	Amazon.com Inc
3	NVDA	NVIDIA Corp
4	GOOGL	Alphabet Inc Class A

NASDAQ 미국 주식

종목 개수 3952개

```
df[['Market']].value_counts()
```

```
Market
KOSDAQ          1645
KOSPI            952
KONEX            129
KOSDAQ GLOBAL     50
dtype: int64
```

	Code	Name
0	005930	삼성전자
1	373220	LG에너지솔루션
2	000660	SK하이닉스
3	207940	삼성바이오로직스
4	005935	삼성전자우

KRX 한국 주식

종목 개수 2775개

00680K	미래에셋증권2우B
00088K	한화3우B
00104K	CJ4우(전환)
02826K	삼성물산우B
00279K	아모레G3우(전환)
33626K	두산퓨얼셀1우
03473K	SK우
28513K	SK케미칼우
37550L	DL이앤씨2우(전환)
33637K	솔루스첨단소재1우
37550K	DL이앤씨우
00499K	롯데지주우
33626L	두산퓨얼셀2우B
35320K	대덕전자1우

국내 주식 Ticker에는 영문+숫자 혼합형 Ticker 24개 존재

국내주식 정렬은 이를 제외한 2751개의 종목을 활용

```
df_str.Code.count()
```

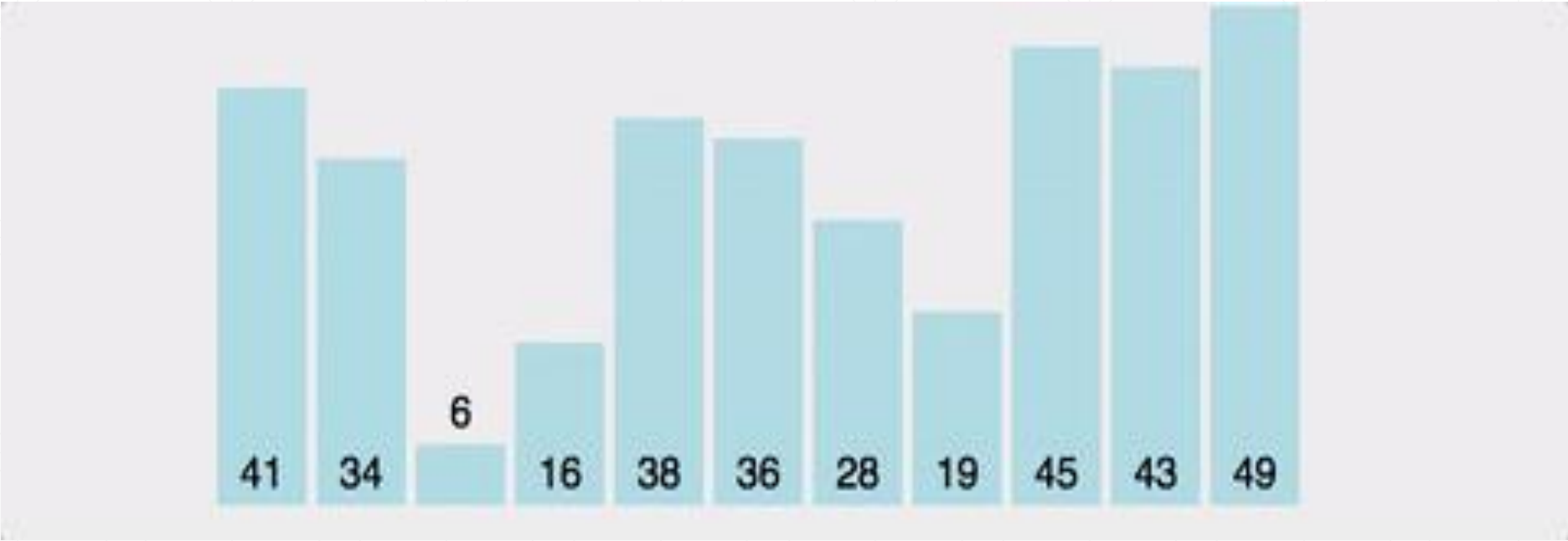
```
24
```

```
[26] def check_list(my_list):
       is_sorted = (sorted(my_list)==my_list)
       print(is_sorted)
```

```
[27] def check_list_r(my_list):
       is_sorted = (sorted(my_list)==my_list)
       return is_sorted
```

check list 함수로 정렬이 제대로 돌아가는지 검증을 진행

Bubble Sort



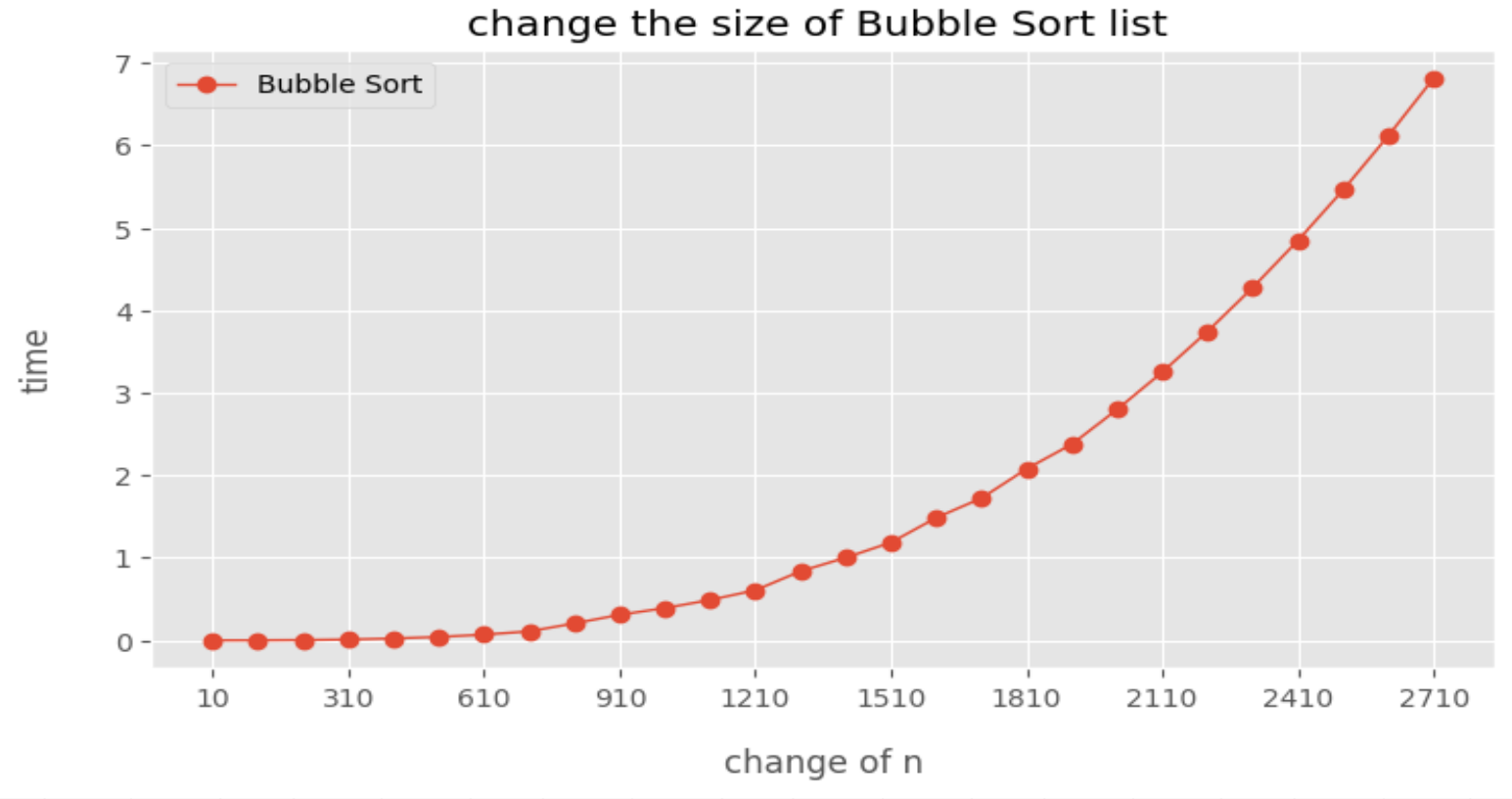
Bubble sort 시간 복잡도

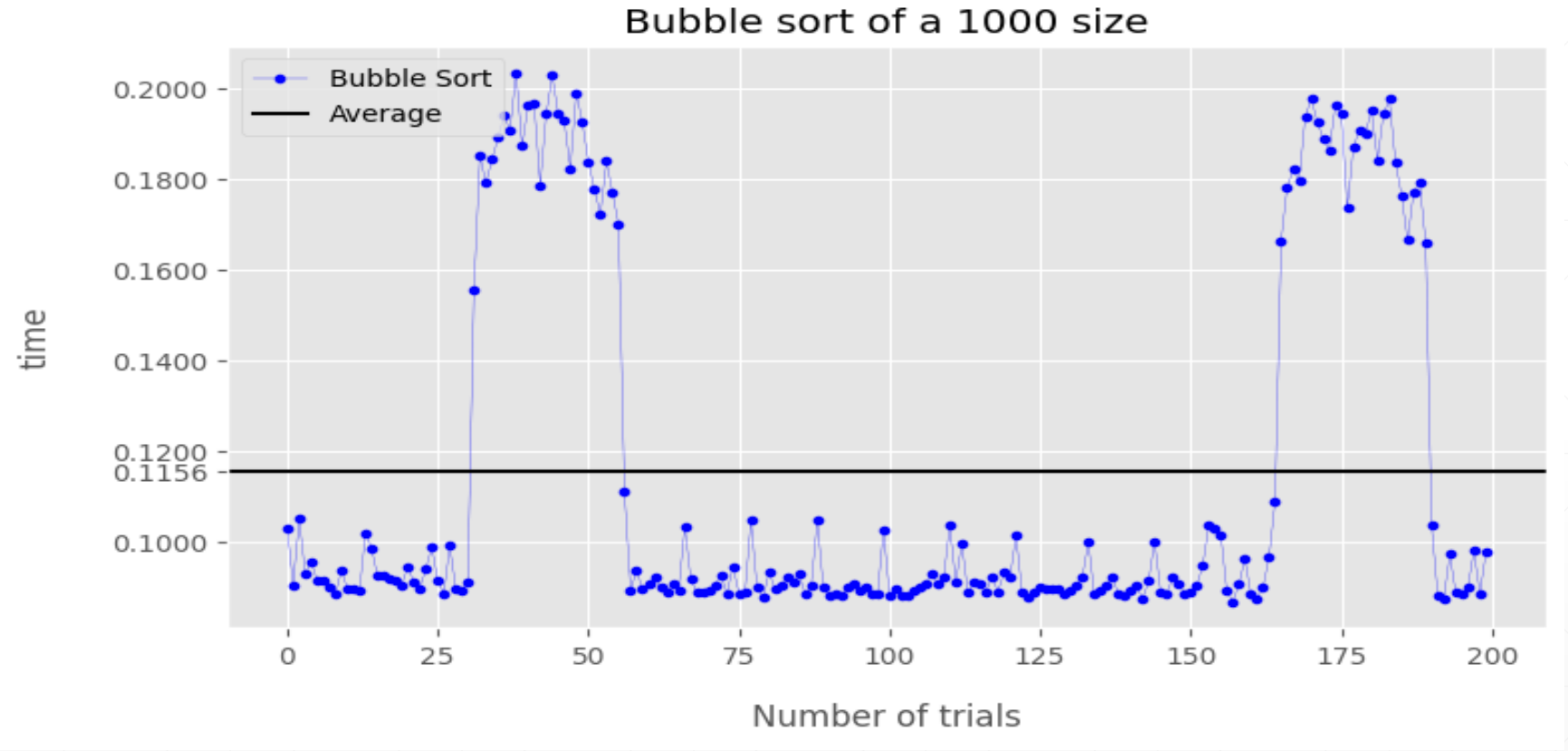
Algorithm	Time complexity		
Bubble sort	Best	Average	Worst
	O(n)	O(n^2)	O(n^2)

실험적 근거

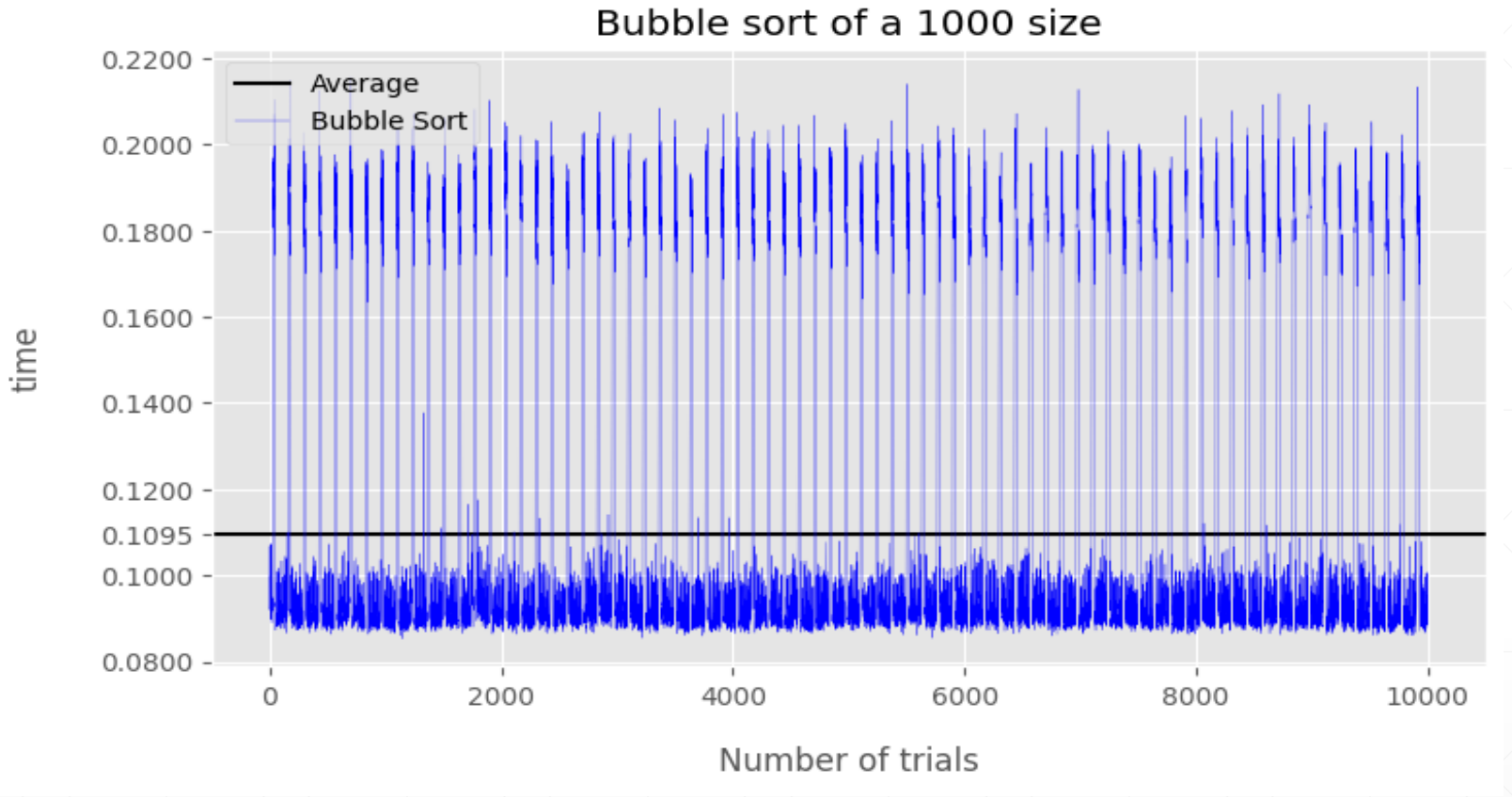
```
#사이즈가 커짐에 따라 변화도

k = 10000  # 회수를 몇번 돌릴지
list_number = np.arange(10, len(non_k_int), 100).tolist() #10에서 100씩 2710까지 올라가는 값
b_run_time = [] #사이즈 당 런타임을 모아두는 임시 리스트
avr_b = [] #사이즈의 평균 값을 모아둔 리스트
i = 0
avr = 0 # 평균값 임시

for j in range(len(list_number)):
    for i in range(0,k):
      n= random.sample(non_k_int,list_number[j])
      start_time = time.time() #시작시간설정
      bubble_sort(n) # 해당코드 삽입 quick_sort(n) / bubble_sort(n) / timSort(n) / radixSort(n) .
      end_time = time.time() #끝나는 시간
      if check_list_r(n) == False: #정렬 에러 잡기코드
        print('ERROR')
      b_run_time.append(end_time - start_time) #두개 빼서 리스트에 추가
    avr = sum(b_run_time) / k #평균계산
    print(avr) # 대략적인 코드 수행 진척도 확인용
    avr_b.append(avr) #평균 리스트에 추가하기
```

표본의 크기에 따른 시간복잡도 변화 그래프





최악의 경우 0.2이상으로 두 배 가량 차이나 성능이 저하되는 모습이 보인다.

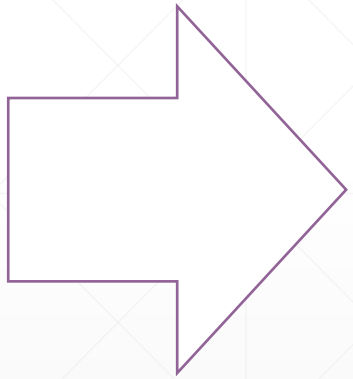


최악의 경우가 빈번하게 나오는 모습을 볼 수 있다.

Tim sort

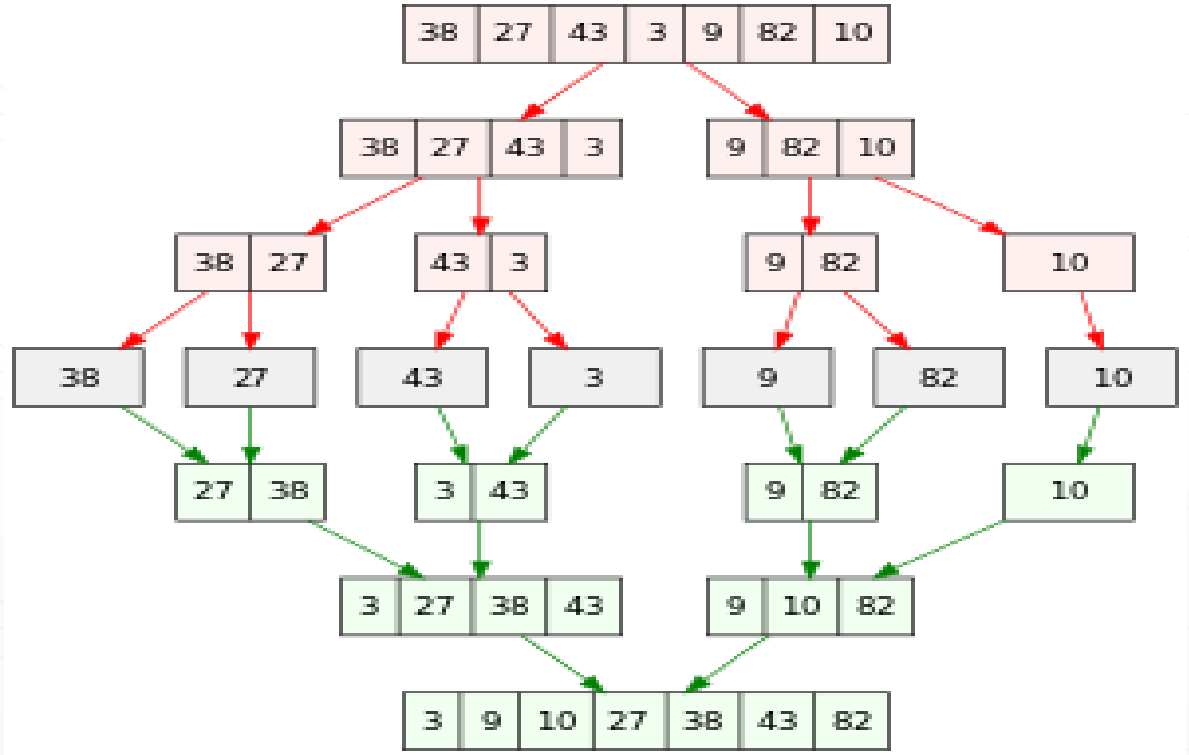
Tim sort 아이디어

현실세계의 데이터들은 어느 정도 정렬된 상태로 배열돼 있는 경우가 많지 않을까?

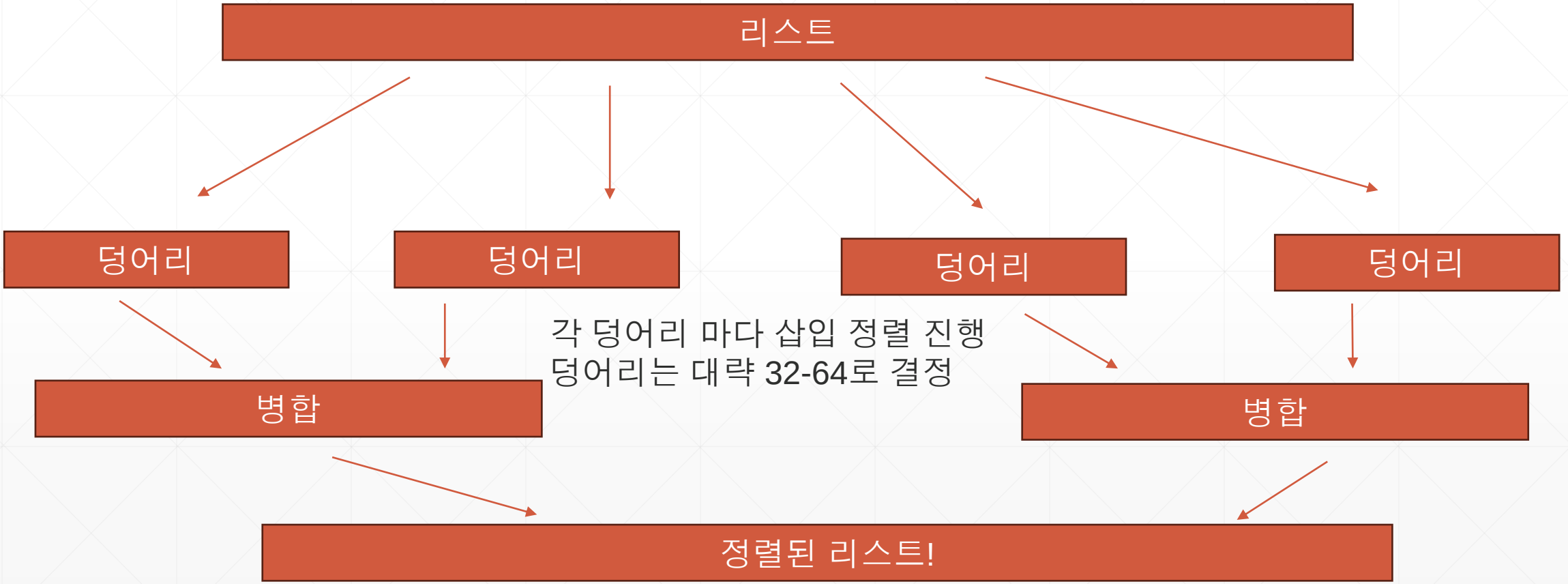


전체 배열을 작은 덩어리들로 잘라 각각의 덩어리를 Insertion sort로 정렬한 뒤, merge sort로 병합하면 더 빠를 거야!

Merge sort


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

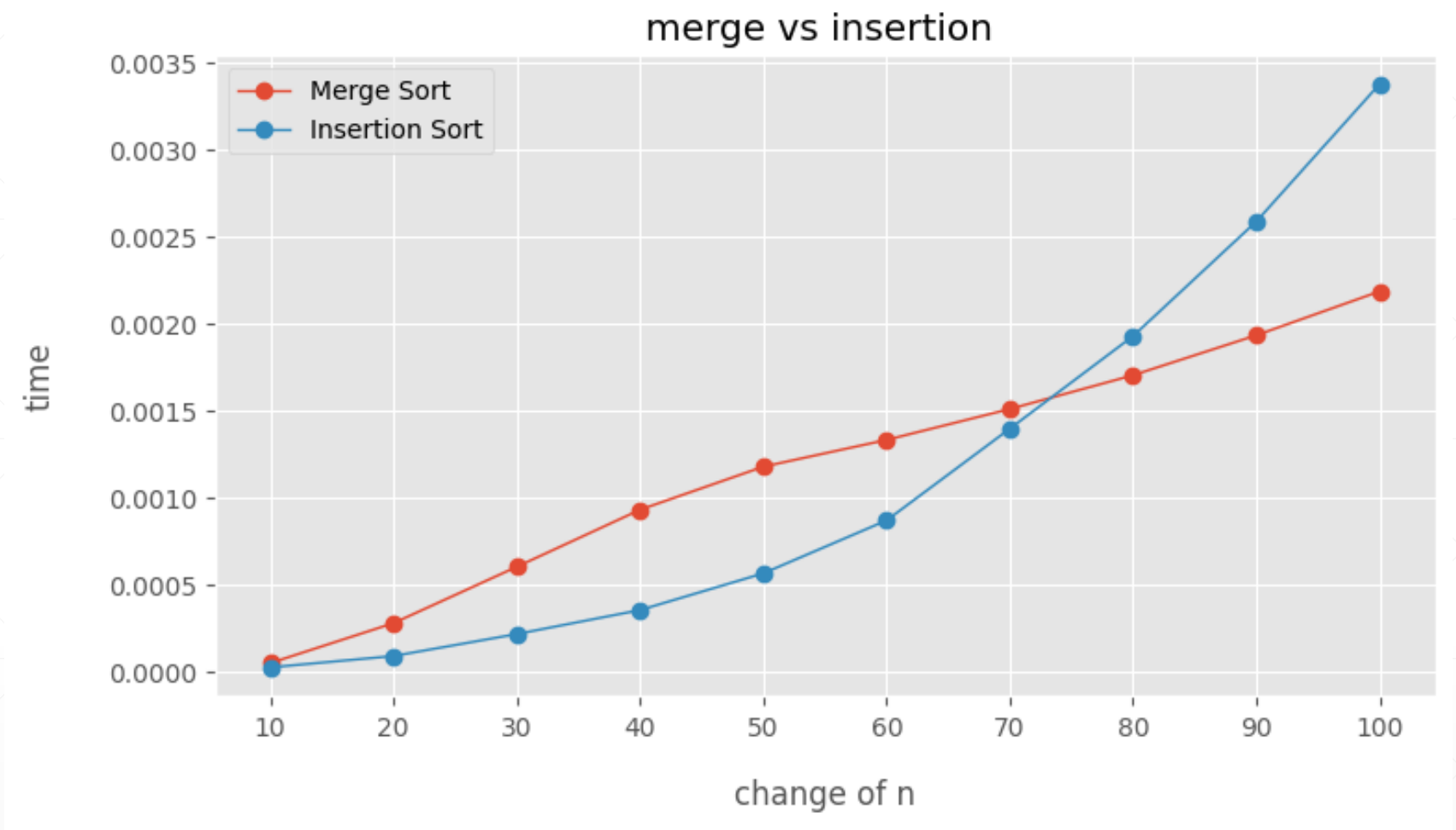
Tim sort


리스트
덩어리
덩어리
덩어리
덩어리
각 덩어리 마다 삽입 정렬 진행
덩어리는 대략 32-64로 결정
병합
병합
정렬된 리스트!

시간 복잡도가 N제곱인 삽입정렬을 사용하는 이유?

$$C_i \times n^2 < C_q \times n \log n$$

N이 작다면 이를 만족할 수 있다!!



같은 조건에서 돌렸을 때 리스트 개수가 70정도 까지 삽입 정렬이 더 빠른 것을 확인할 수 있다.

따라서 2의 제곱형태인 32와 64를 한 덩어리의 크기로 주로 활용한다.

Tim sort 원리

1. 덩어리 나누기 (대략 32-64)

2. 덩어리 마다 삽입 정렬 진행

3. Merge sort로 병합하기

Tim sort 특징

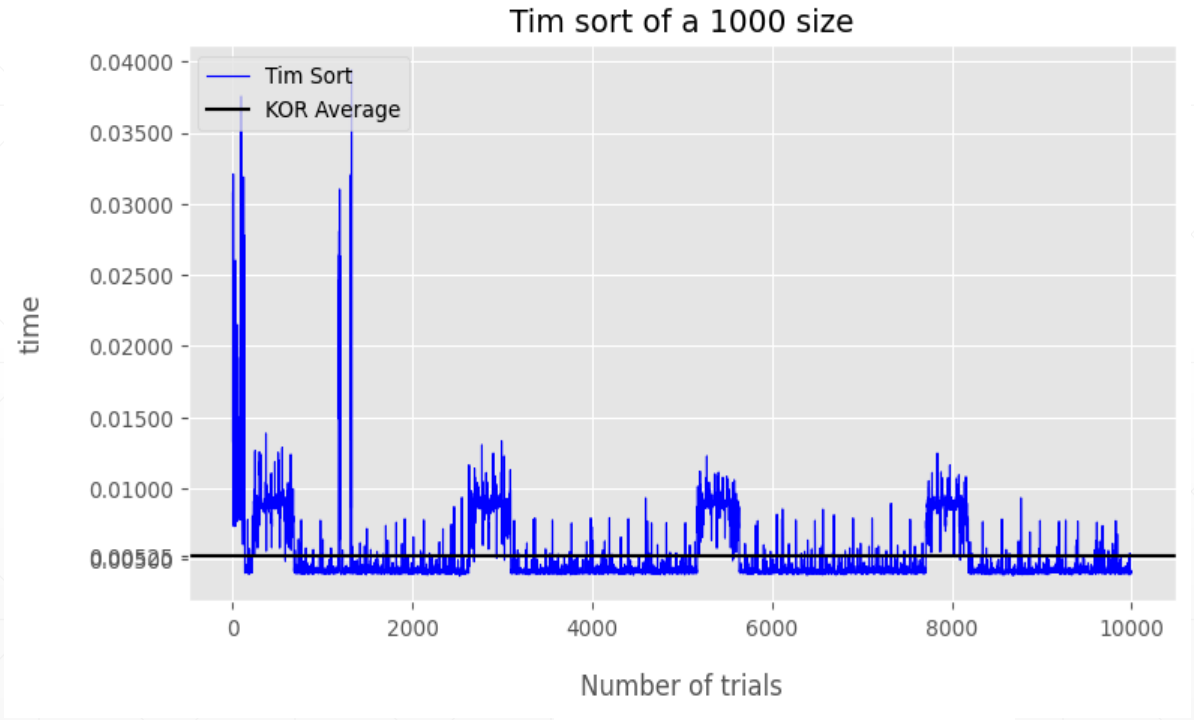
- 창안자인 Tim Peters의 이름을 따서 팀정렬(Tim sort)알고리즘이라 부른다
- 파이썬의 sort()의 내장 함수이다.

Algorithm	Time complexity		
Tim sort	Best	Average	Worst
	O(n)	O(nlogn)	O(nlogn)

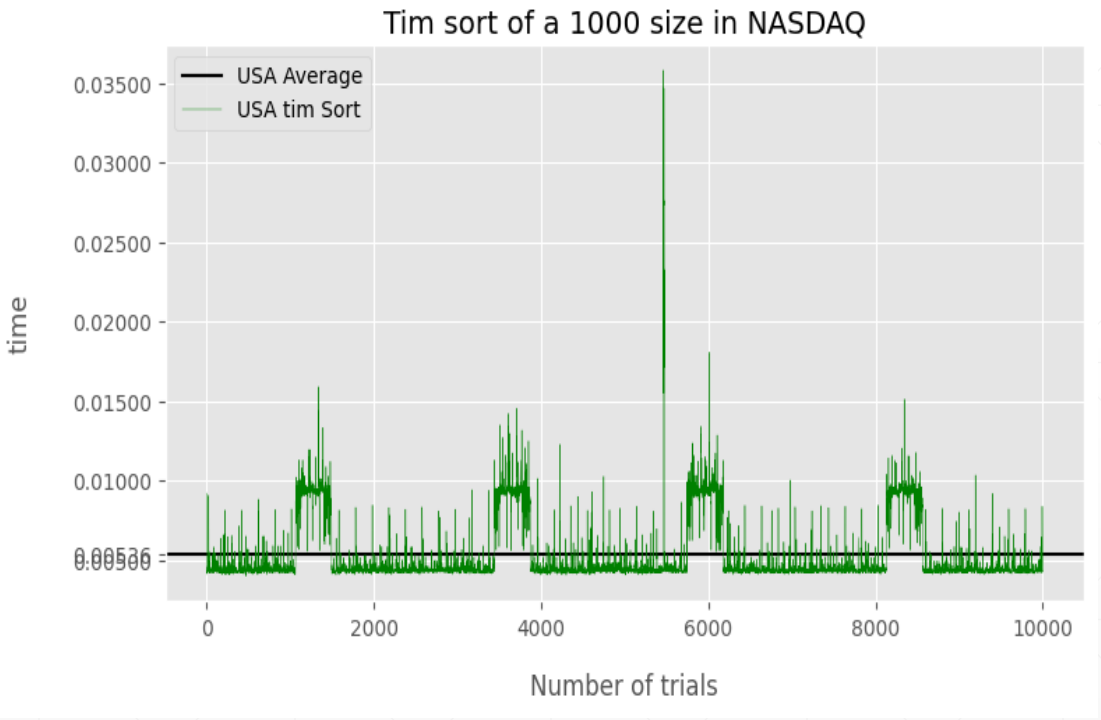
Tim sort code

```python
# Python3 program to perform basic timSort
MIN_MERGE = 32


def calcMinRun(n):
    """Returns the minimum length of a
    run from 23 - 64 so that
    the len(array)/minrun is less than or
    equal to a power of 2.

    e.g. 1=>1, ..., 63=>63, 64=>32, 65=>33,
    ..., 127=>64, 128=>32, ...
    """
    r = 0
    while n >= MIN_MERGE:
        r |= n & 1
        n >>= 1
    return n + r


# This function sorts array from left index to
# to right index which is of size atmost RUN
def insertionSort(arr, left, right):
    for i in range(left + 1, right + 1):
        j = i
        while j > left and arr[j] < arr[j - 1]:
            arr[j], arr[j - 1] = arr[j - 1], arr[j]
            j -= 1


# Merge function merges the sorted runs
def merge(arr, l, m, r):

    # original array is broken in two parts
    # left and right array
    len1, len2 = m - l + 1, r - m
    left, right = [], []
    for i in range(0, len1):
        left.append(arr[l + i])
    for i in range(0, len2):
        right.append(arr[m + 1 + i])

    i, j, k = 0, 0, l

    # after comparing, we merge those two array
    # in larger sub array
    while i < len1 and j < len2:
        if left[i] <= right[j]:
            arr[k] = left[i]
            i += 1

        else:
            arr[k] = right[j]
            j += 1

        k += 1

    # Copy remaining elements of left, if any
    while i < len1:
        arr[k] = left[i]
        k += 1
        i += 1

    # Copy remaining element of right, if any
    while j < len2:
        arr[k] = right[j]
        k += 1
        j += 1


# Iterative Timsort function to sort the
# array[0...n-1] (similar to merge sort)
def timSort(arr):
    n = len(arr)
    minRun = calcMinRun(n)

    # Sort individual subarrays of size RUN
    for start in range(0, n, minRun):
        end = min(start + minRun - 1, n - 1)
        insertionSort(arr, start, end)

    # Start merging from size RUN (or 32). It will merge
    # to form size 64, then 128, 256 and so on ....
    size = minRun
    while size < n:

        # Pick starting point of left sub array. We
        # are going to merge arr[left..left+size-1]
        # and arr[left+size, left+2+size-1]
        # After every merge, we increase left by 2*size
        for left in range(0, n, 2 * size):

            # Find ending point of left sub array
            # mid+1 is starting point of right sub array
            mid = min(n - 1, left + size - 1)
            right = min((left + 2 * size - 1), (n - 1))

            # Merge sub array arr[left.....mid] &
            # arr[mid+1....right]
            if mid < right:
                merge(arr, left, mid, right)

        size = 2 * size
```

Tim sort 분포도

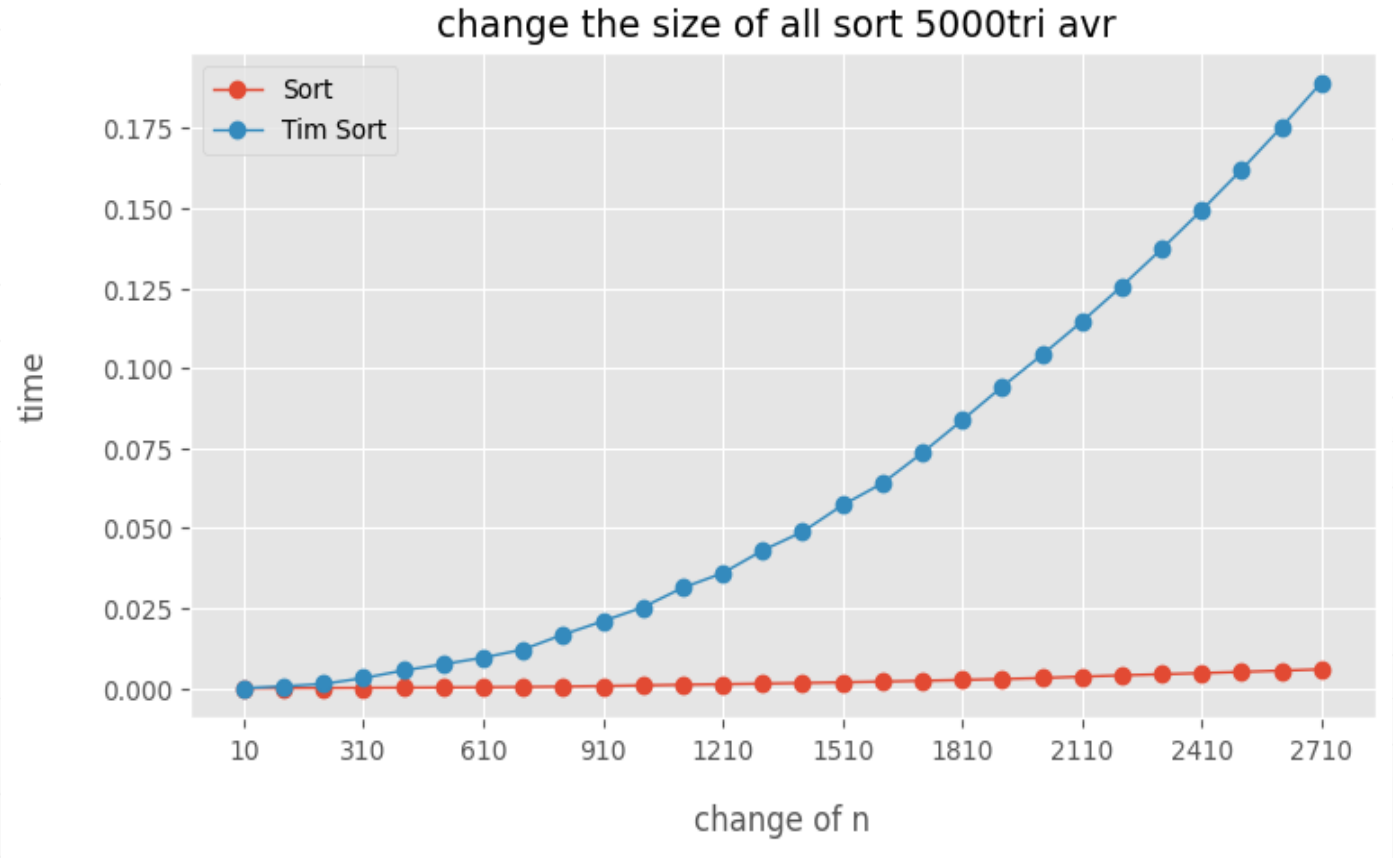


KRX 사이즈 1000개 고정
평균시간- 0.005249804520606994



NASDAQ 사이즈 1000개 고정
평균시간-0.005355568790435791

파이썬 내장된 .sort() vs Tim sort



내장함수로 알려진 sort와 속도 차이가 크게 나는 것을 확인할 수 있다.

Why?

팀 소트 최적화 방안

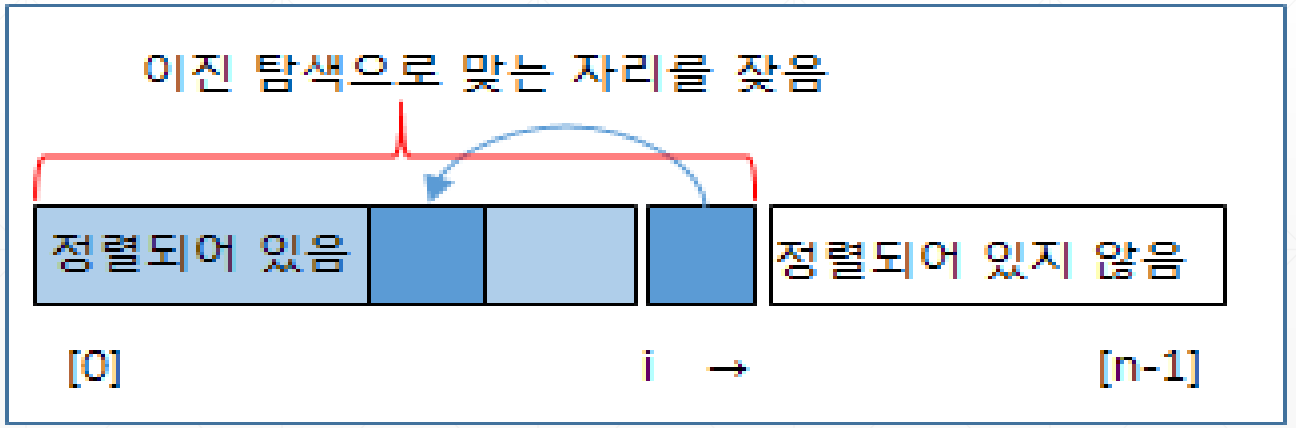
- Binary insertion sort
- 덩어리 똑똑하게 합치기
- 오름차순이면 정렬 안하고 합치기

등등..

팀 소트 최적화 방안

- Binary insertion sort
- 덩어리 똑똑하게 합치기
- 오름차순이면 정렬 안하고 합치기

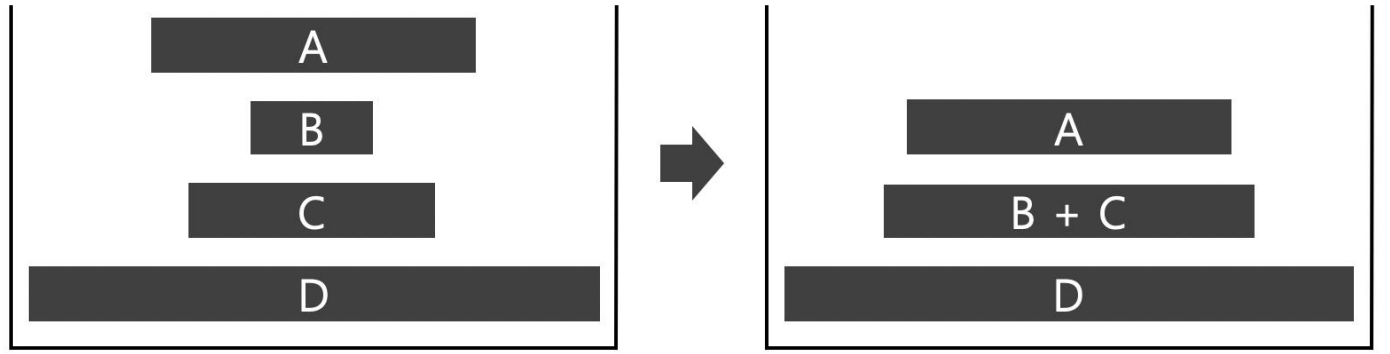
등등..



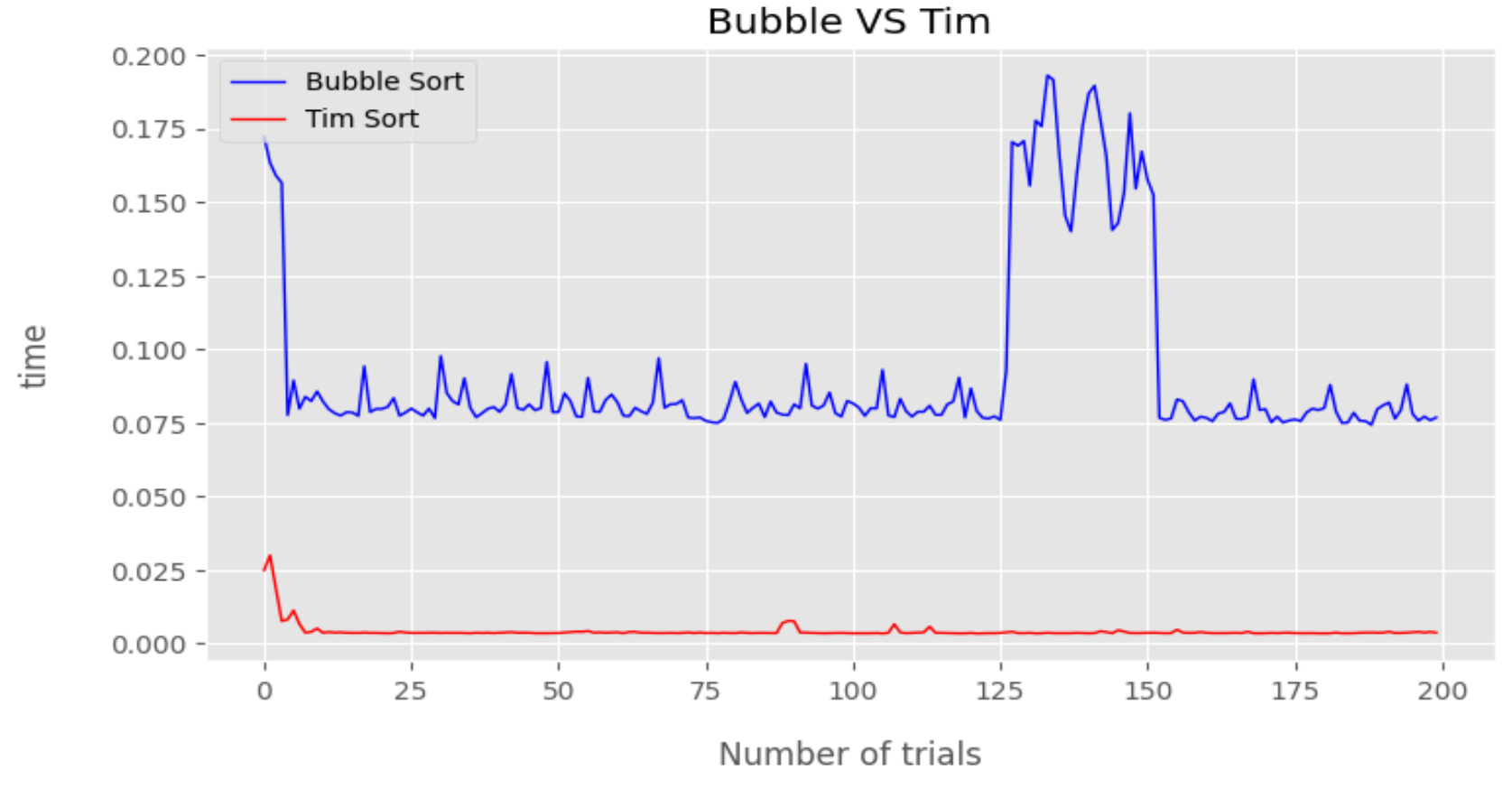
팀 소트 최적화 방안

- Binary insertion sort
- 덩어리 똑똑하게 합치기
- 오름차순이면 정렬 안하고 합치기

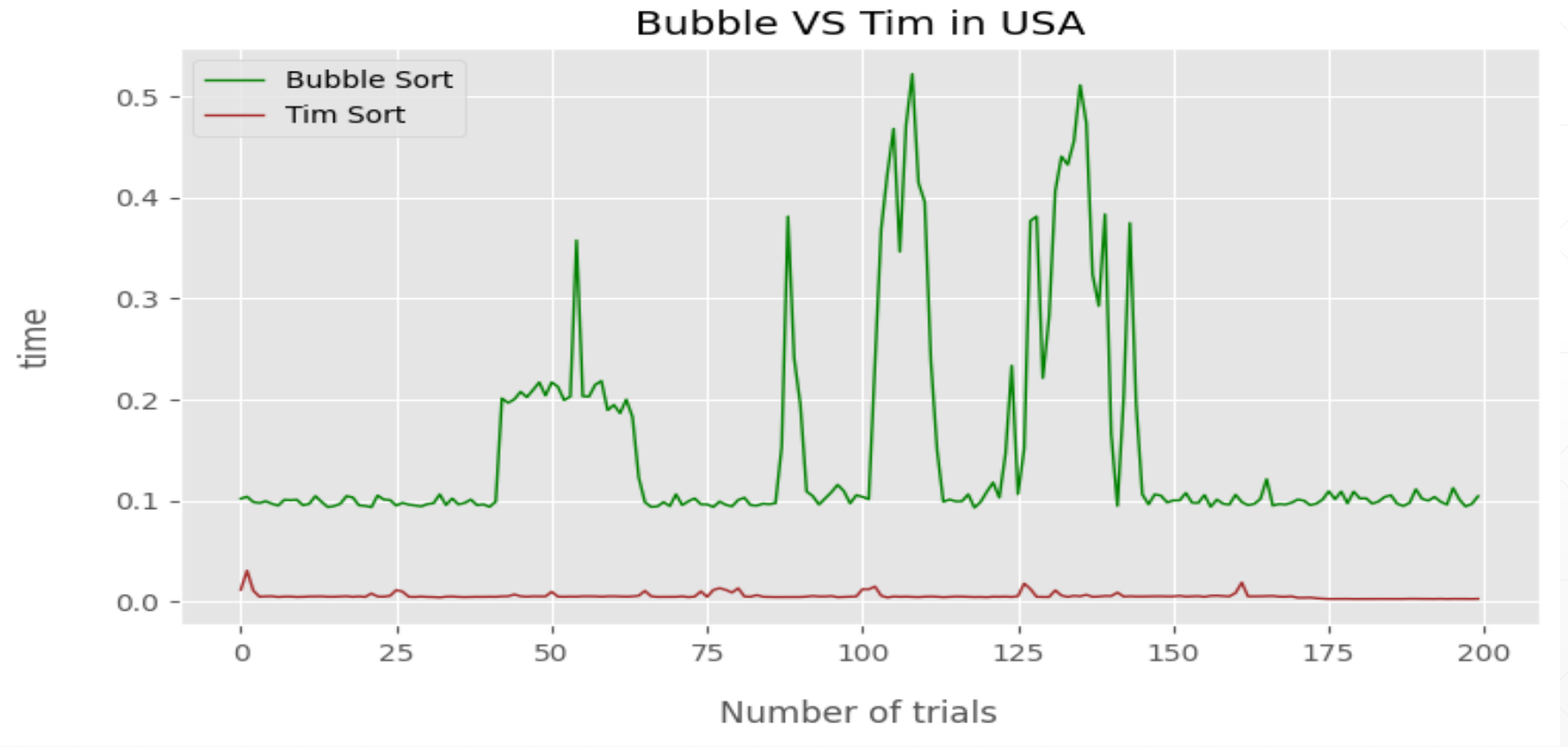
등등..



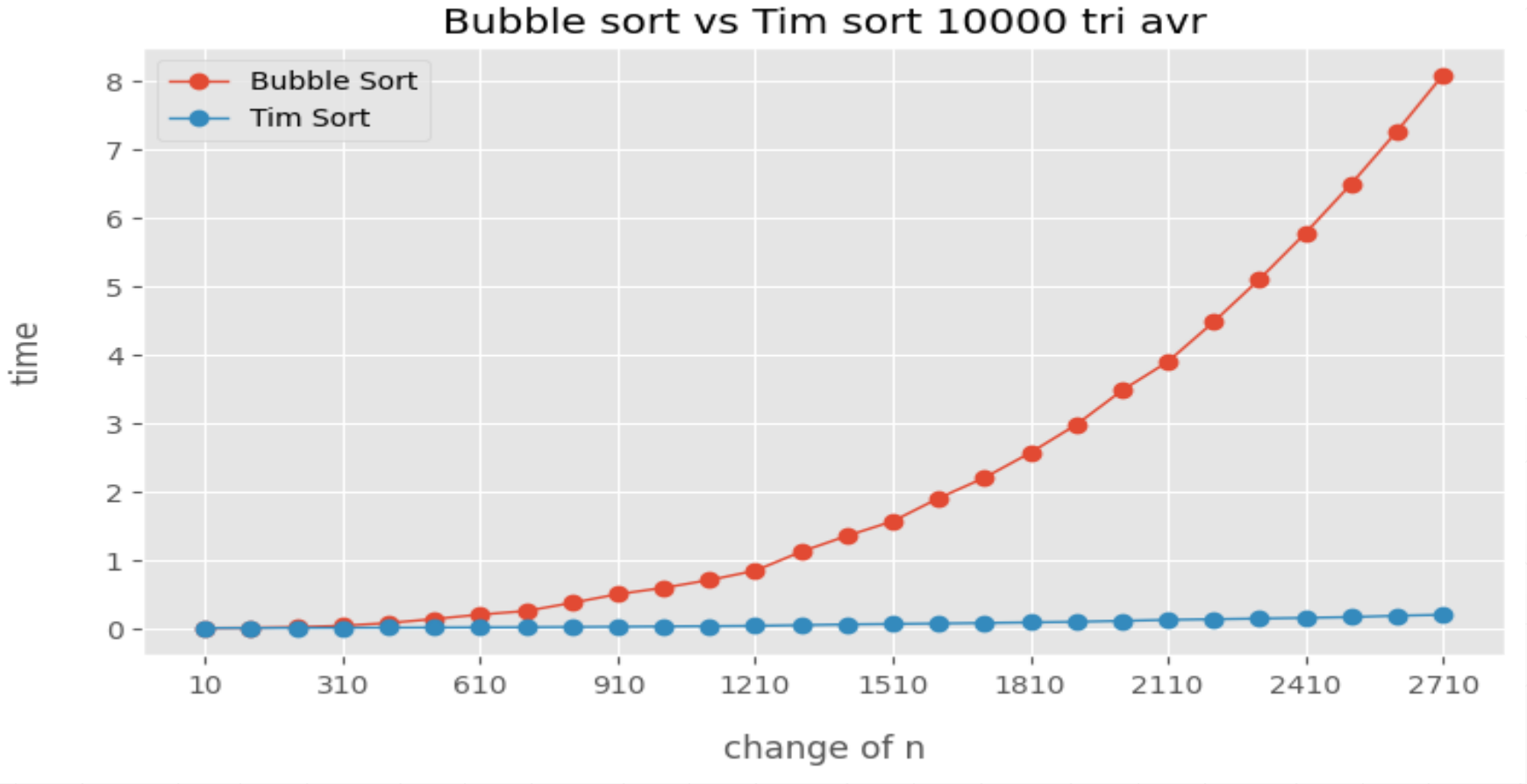
Bubble sort VS Tim sort KRX



Bubble sort VS Tim sort NASDAQ



Bubble sort VS Tim sort in size change



팀소트 사이즈별 평균값

110: 0.0009219670295715333
1010: 0.023523415327072143
2010: 0.1051359236240387

버블 사이즈별 평균값

110: 0.0036975145339965 82
1010: 0.5876614809036255
2010: 3.469430911540985

Tim sort 장단점

장점

1.어느 정도 정렬된 데이터에서 빠른 성능을 보여준다.

2.참조 지역성의 원리를 잘 만족한다.

3.안정적이다.

단점

1.무작위된 배열에서는 성능이 느리다.

Radix sort

Radix sort

1. Radix sort(기수 정렬)의 이론적 개념과 예시
2. 시간 복잡도
3. 실험적 근거
4. 장단점

Radix sort란?

자릿수를 크기대로 배열하여 큐(Queue)방식으로 정렬하는 알고리즘
  cf. 큐(Queue) : 들어간 값 순서대로 나오는 구조

값들 간의 비교 연산을 하지 않고 정렬할 수 있는 빠르고 안정적인 정렬 알고리즘

Radix sort의 이론적 개념

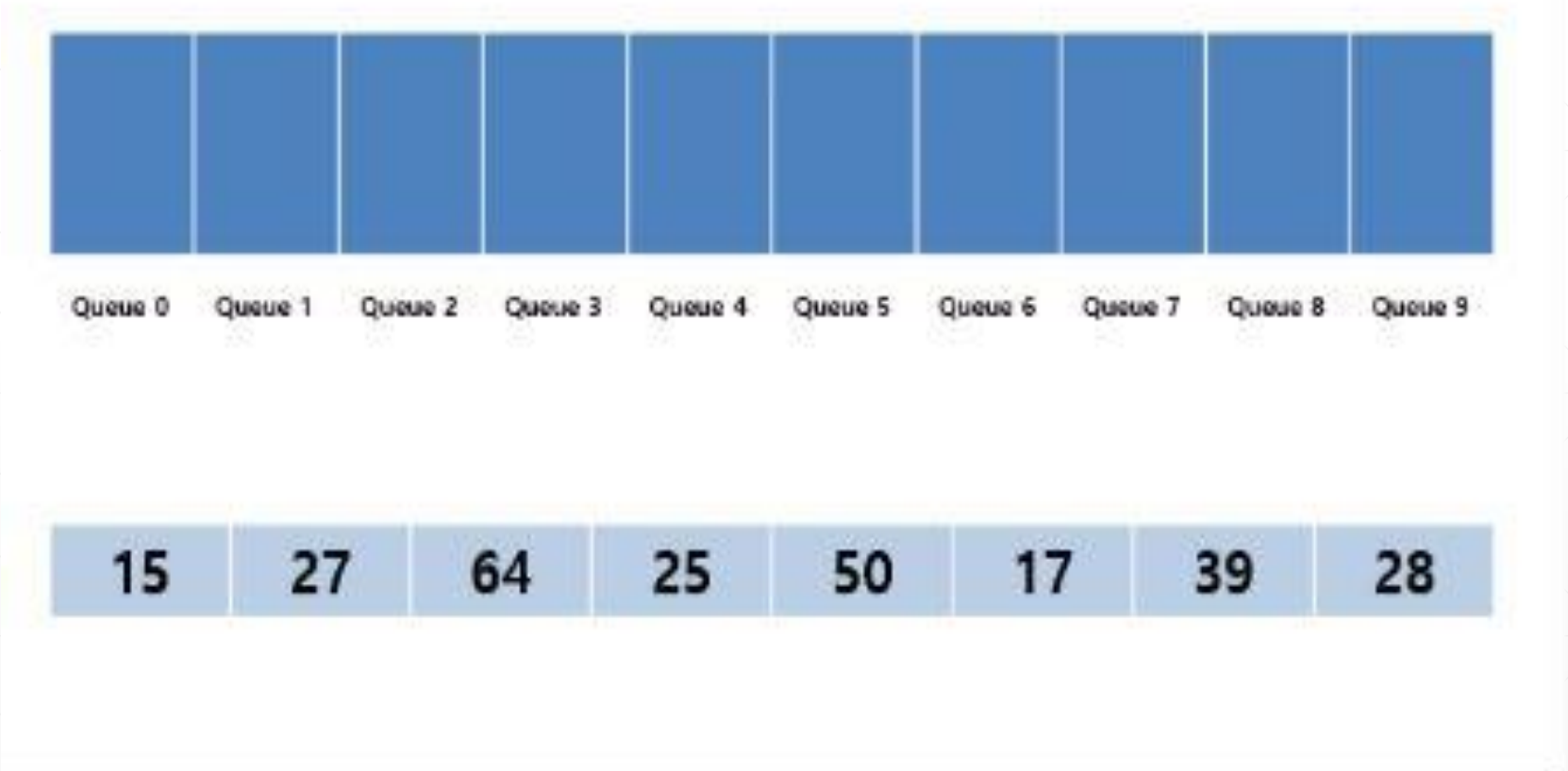
1. 먼저 0~9 까지의 Bucket을 준비한다.

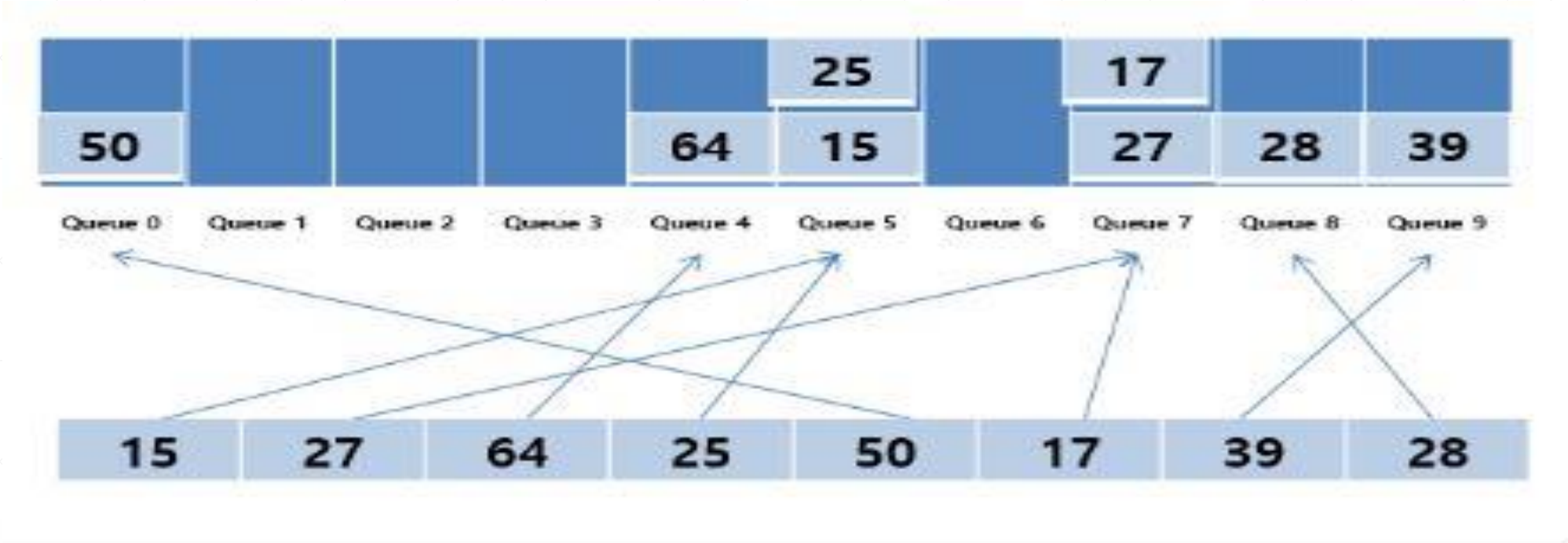
2. 모든 데이터의 가장 낮은 자릿수에 해당하는 값을 Bucket에 차례대로 위치시킨다.

3. Bucket에서 0부터 큐(Queue) 방식으로 데이터를 정렬시킨다.

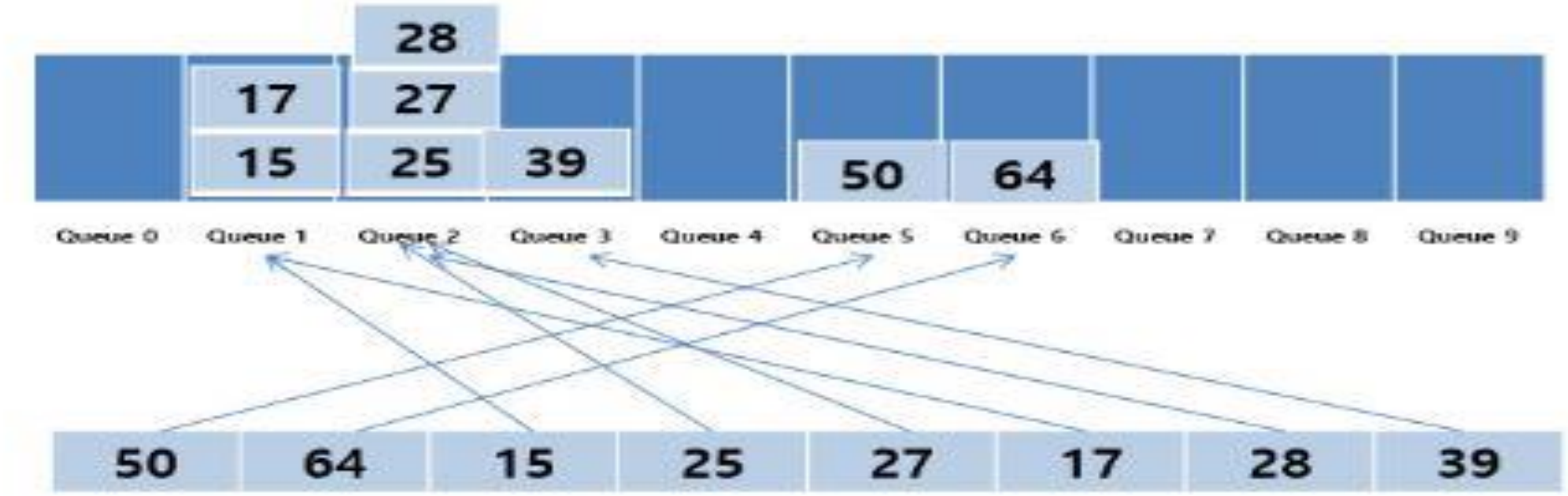
4. 가장 낮은 자릿수부터 가장 높은 자릿수까지 2번 3번 과정을 반복한다.

Radix sort의 예시





50	64	15	25	27	17	28	39



15	17	25	27	28	39	50	64

Radix sort의 시간복잡도

Time complexity	Radix sort	다른 정렬 방법
Best	X	O
Worst	X	O

Algorithm	Time complexity
Radix sort	O(n)

ticker의 개수를 n개라 하고, 가장 큰 수의 자릿수를 d라 하면
1의 자릿수부터 d의 자릿수까지 n번 시행 하기때문에

T(n) = O(d*n) = O(n)의 시간복잡도를 가진다.

실험적 근거

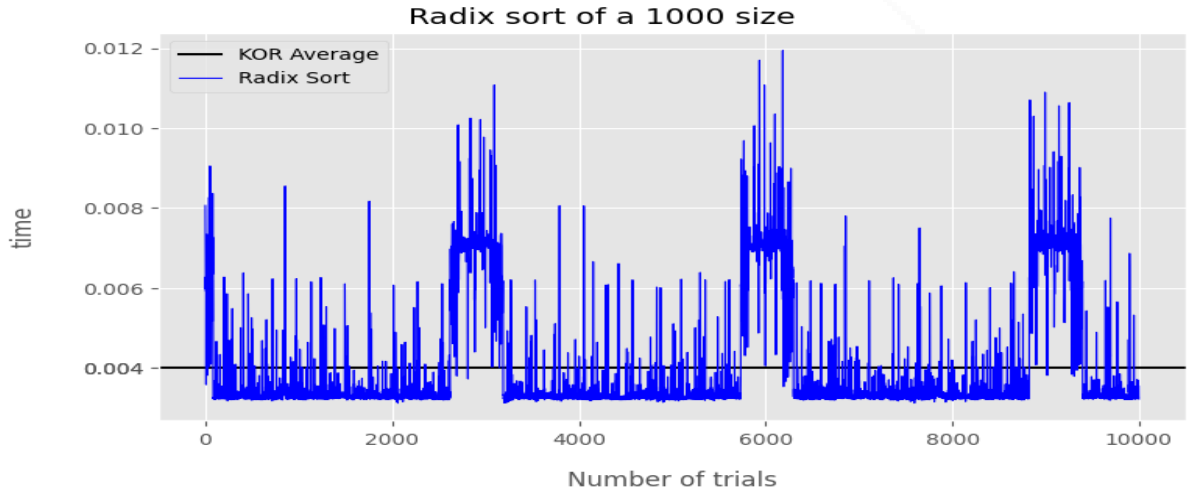
Radix sort

```python
# Using counting sort to sort the elements in the basis of significant places
def countingSort(array, place):
    size = len(array)
    output = [0] * size
    count = [0] * 10

    # Calculate count of elements
    for i in range(0, size):
        index = array[i] // place
        count[index % 10] += 1

    # Calculate cumulative count
    for i in range(1, 10):
        count[i] += count[i - 1]

    # Place the elements in sorted order
    i = size - 1
    while i >= 0:
        index = array[i] // place
        output[count[index % 10] - 1] = array[i]
        count[index % 10] -= 1
        i -= 1

    for i in range(0, size):
        array[i] = output[i]


# Main function to implement radix sort
def radixSort(array):
    # Get maximum element
    max_element = max(array)

    # Apply counting sort to sort elements based on place value.
    place = 1
    while max_element // place > 0:
        countingSort(array, place)
        place *= 10
```



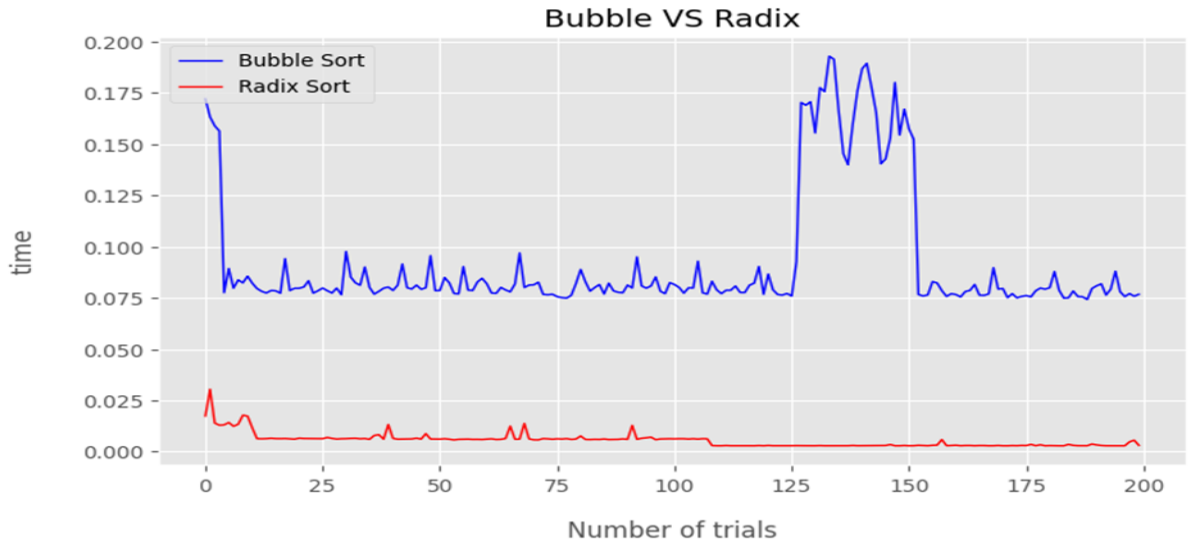
```python
#래딕스 그래프 만들기용

k = 200 # 회수를 몇번 돌릴지
list_number = 1000 # 몇개의 리스트를 뽑을지
radix_run_time = [] #런타임을 모아두는 리스트
i = 0

while i < k:
  n= random.sample(non_k_int,list_number)
  n = list(n)
  start_time = time.time() #시작시간설정
  radixSort(n) # 해당코드 삽입 quick_sort(n) / bubble_sort(n) / timSort(n) / radixSort(n)
  end_time = time.time() #끝나는 시간
  radix_run_time.append(end_time - start_time) #두개 빼서 리스트에 추가
  i += 1 #횟수 세기
radix_avr_run_time = sum(radix_run_time) / k

print(radix_avr_run_time)
```

Bubble sort와 Radix sort의 시간 복잡도 비교



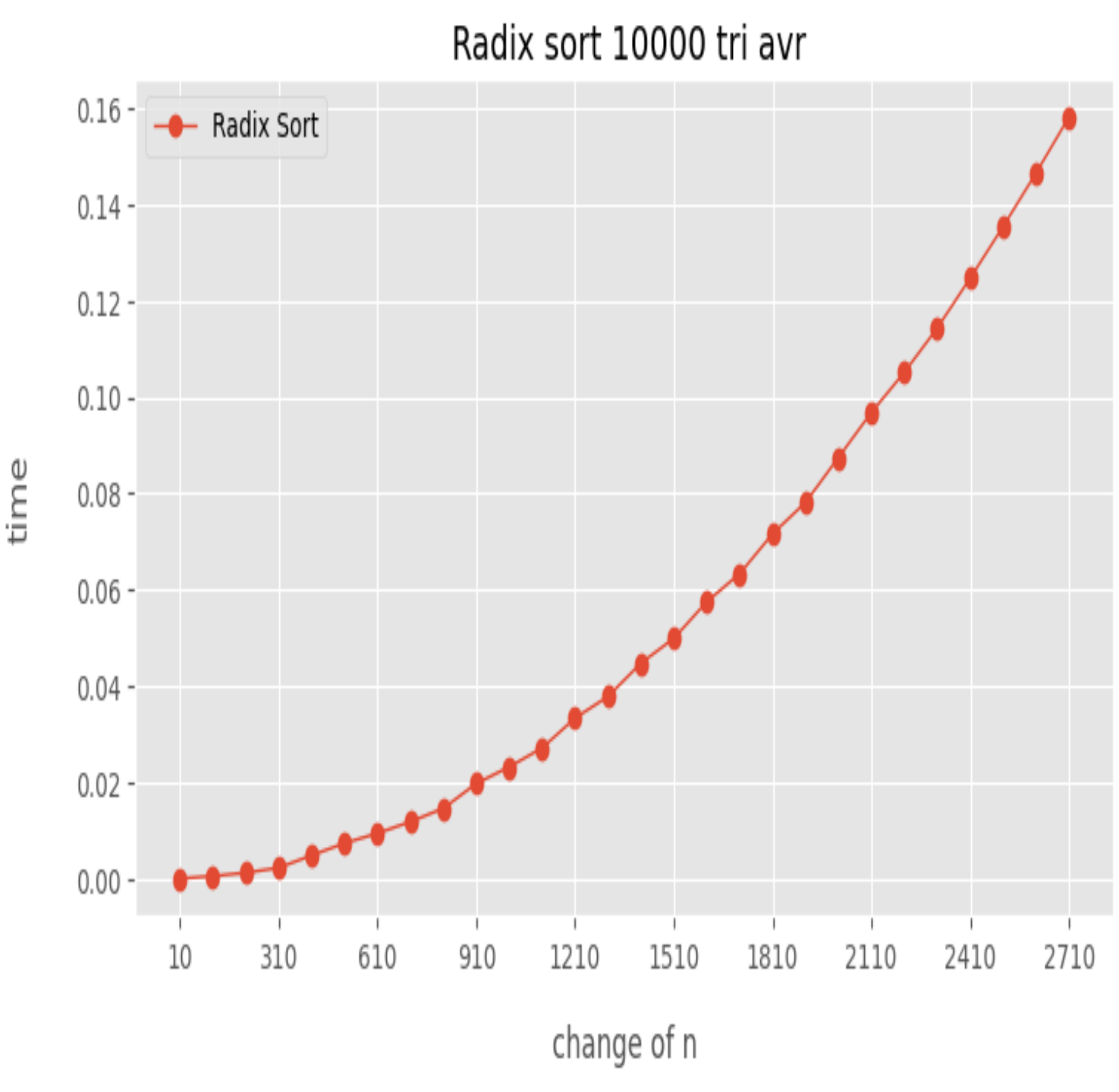
Ticker 1000개를 10000번 정렬할 때 걸리는 시간값의 평균값

Radix sort의 평균 값 : 0.00400798530578613 3



Ticker 1000개를 200번 돌린 값의 평균값

육안으로 봐도 Radix sort가 Bubble sort보다 훨씬 빠르다는 것을 알 수 있다



Radix sort 10000 tri avr

n=110

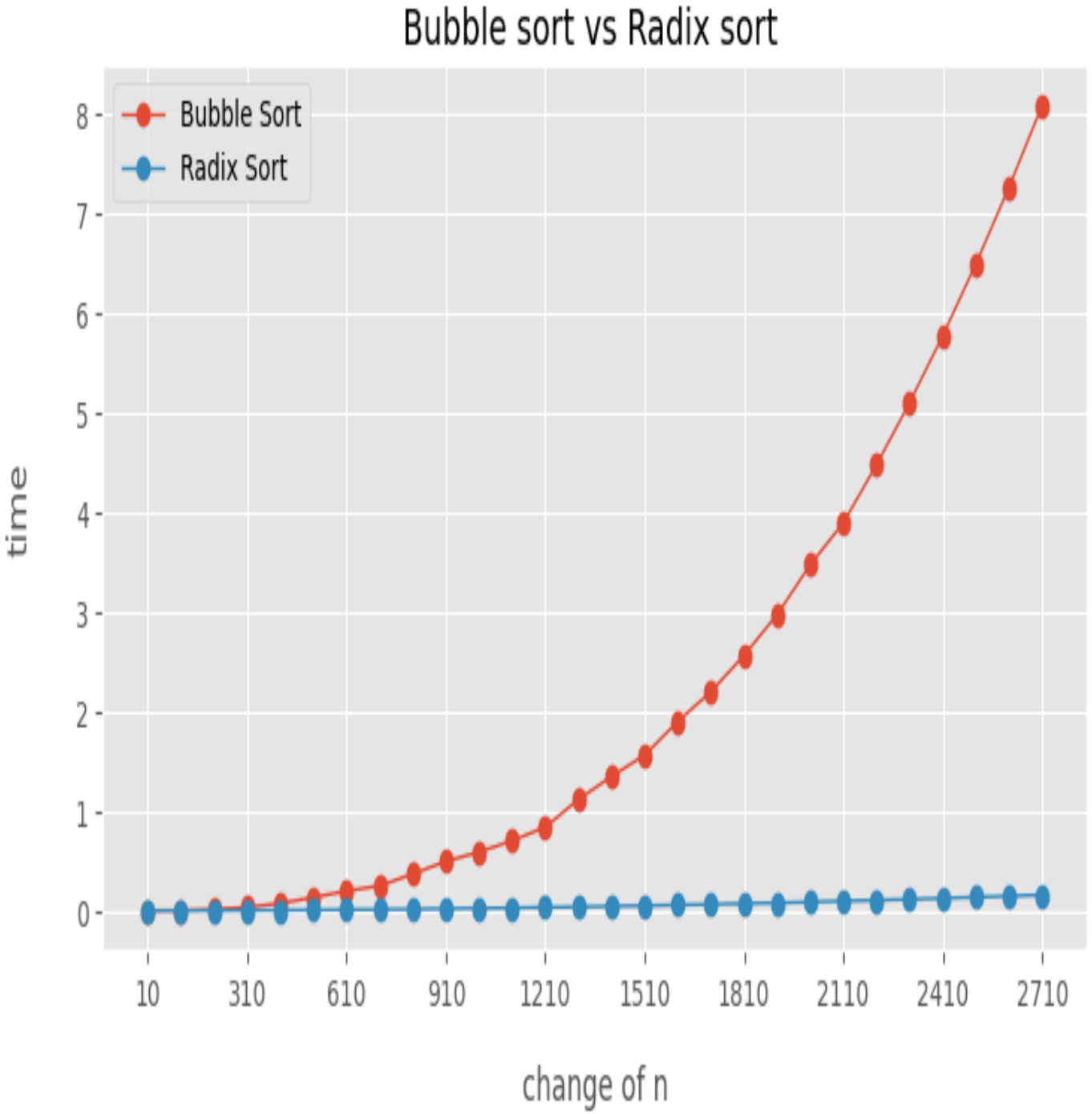
0.000700736999511718 7

n=1010

0.023322611570358278

n=2010

0.0874816219806671 1



Bubble sort 10000 tri avr

n=110

0.003697514533996582

n=1010

0.587661480903625 5

n=2010

3.469430911540985

sort / n	Radix sort	Bubble sort	배수
110	0.000700736999511718 7	0.00369751453399658 2	약 5배
1010	0.02332261157035827 8	0.587661480903625 5	약 29배
2010	0.08748162198066711	3.469430911540985	약 43배

Radix sort의 장단점

장점

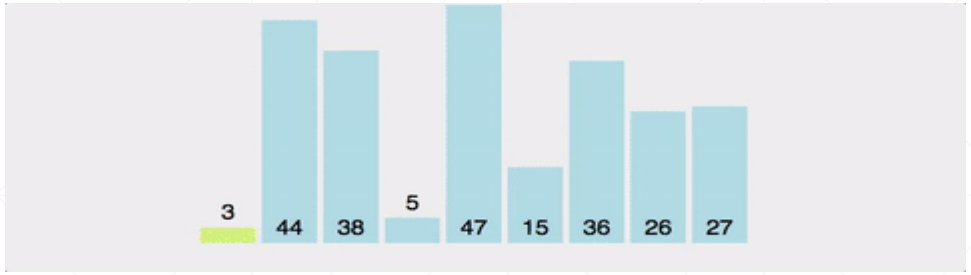
- 비교연산 없이 정렬을 수행하기 때문에 빠른 시간 복잡도를 가진다.
- 자릿수가 일정한 값들을 비교하는데 효과적이다.

단점

- Bucket을 생성하기 때문에 추가적인 메모리가 많이 필요하다.
- 버킷의 종류가 고정적이지 않다.
- 음수, 부동소수점 같은 자릿수가 없는 데이터는 정렬할 수 없다.

Quick sort

Quick sort 원리



분할(Divide): 입력 배열을 피벗을 기준으로 비균등하게 2개의 부분 배열로 분할한다.

정복(Conquer): 분할한 양쪽 각각 재귀로 퀵 정렬한다.

Quick sort 원리

- 첫 번째 단계 Pivot 정하기

Quick sort 원리

좌 -> 우로 탐색하며 pivot보다 큰 값을 찾고

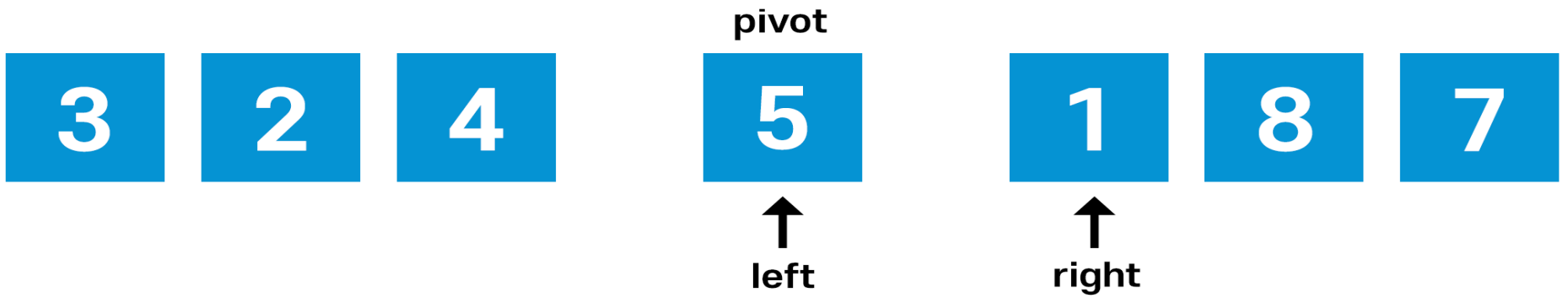
우 -> 좌로 탐색하며 pivot보다 작은 값을 찾는다.

Quick sort 원리

7 > 5 이고 2 < 5이기 때문에 7과 2의 위치를 스왑

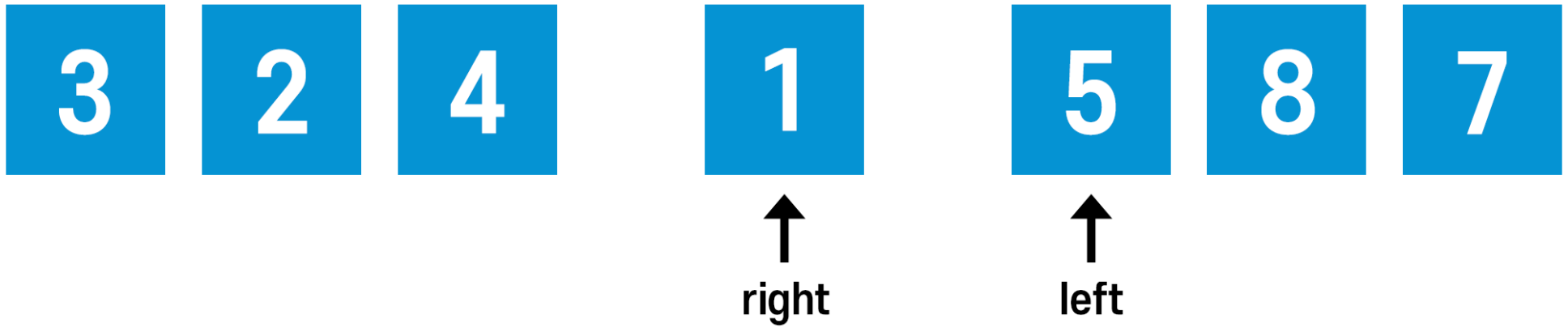
Quick sort 원리

Quick sort 원리



5 = 5 이고 1 < 5이기 때문에
5와 1의 위치를 스왑

Quick sort 원리

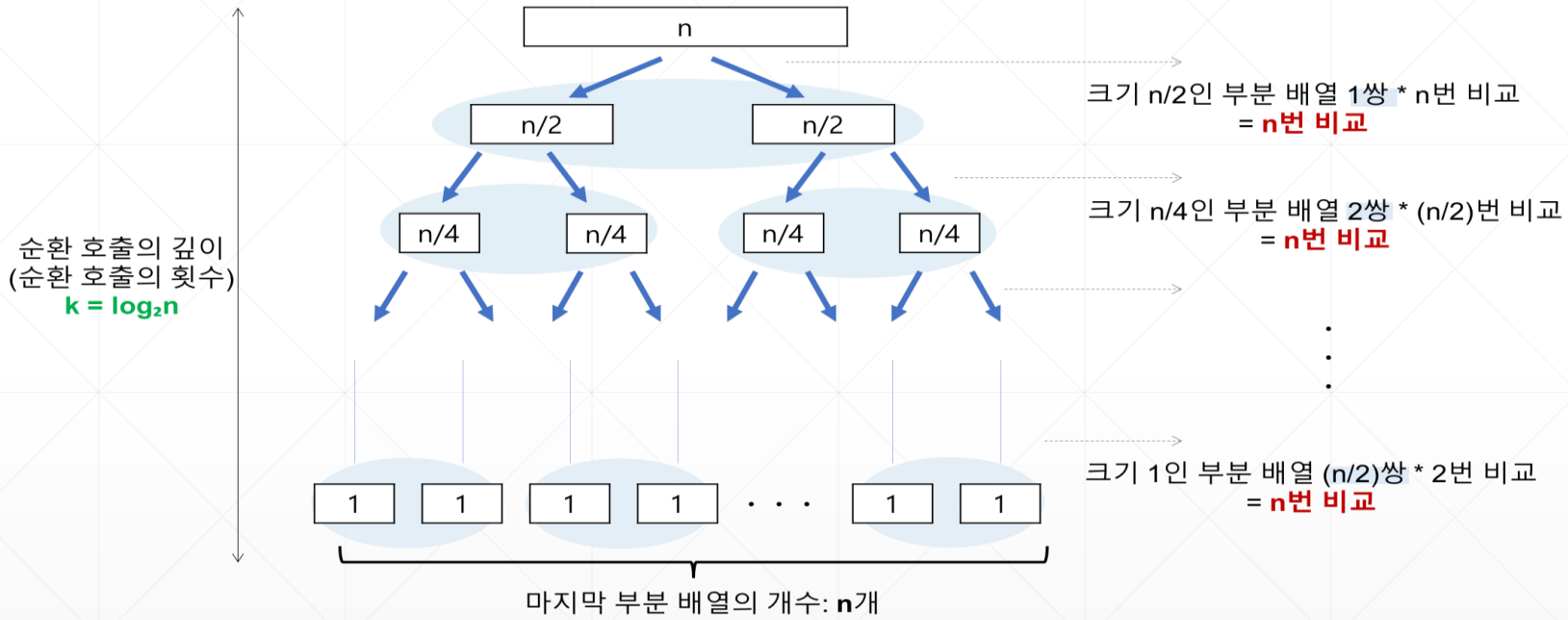


PIVOT을 기준으로 작은 수들은 왼쪽
큰 수 들은 오른쪽으로 분할이 된다.

Bubble sort 시간 복잡도

Algorithm	Time complexity		
Quick sort	Best	Average	Worst
	O(nlogn)	O(nlogn)	O(n^2)

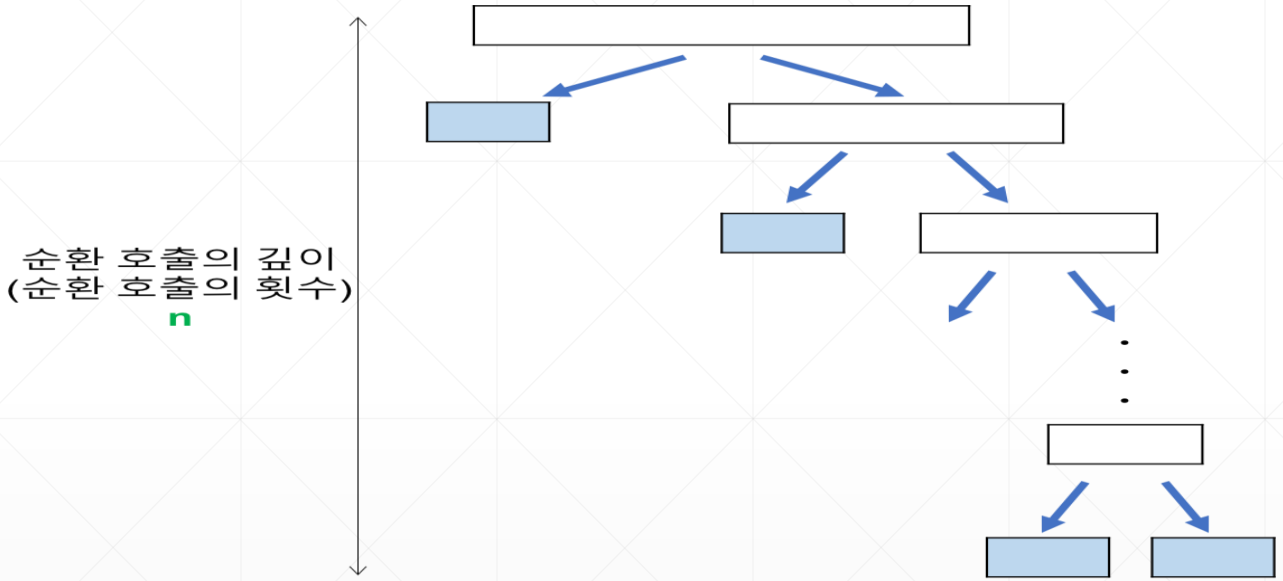
Quick sort의 시간 복잡도 (최선의 경우)



순환 호출의 횟수 x 각 순환 호출 단계의 비교 연산 = **nlog(n)**

최선의 경우: **T(n) = O(nlog(n))**

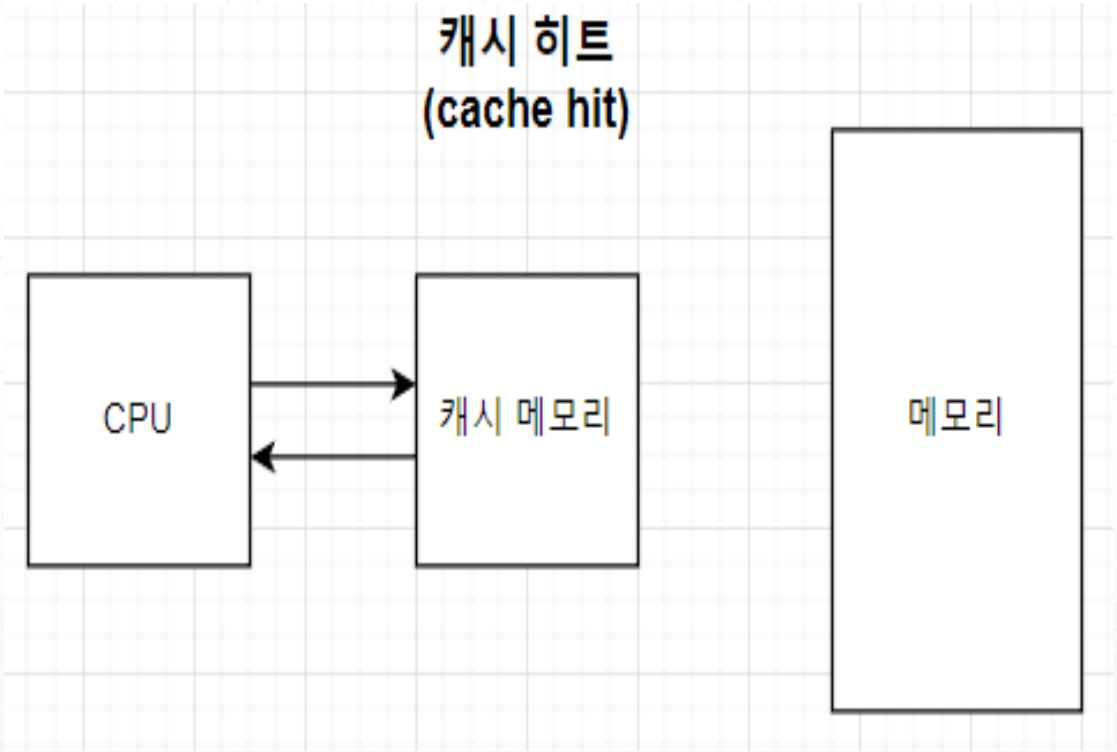
Quick sort의 시간 복잡도 (최악의 경우)



순환 호출의 횟수 x 각 순환 호출 단계의 비교 연산 = **n^2**

최악의 경우 : **T(n) = O(n^2)**

Quick sort가 더 빠른 이유 – locality



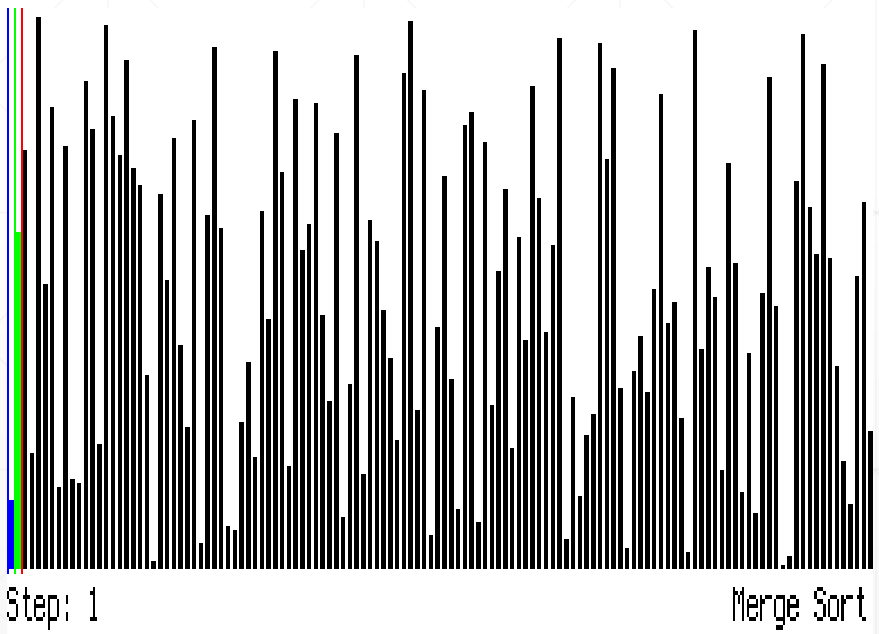
Locality : 메모리 내의 정보를 균일하게 엑세스 하는게 아닌

짧은 시간내에 특정 부분을

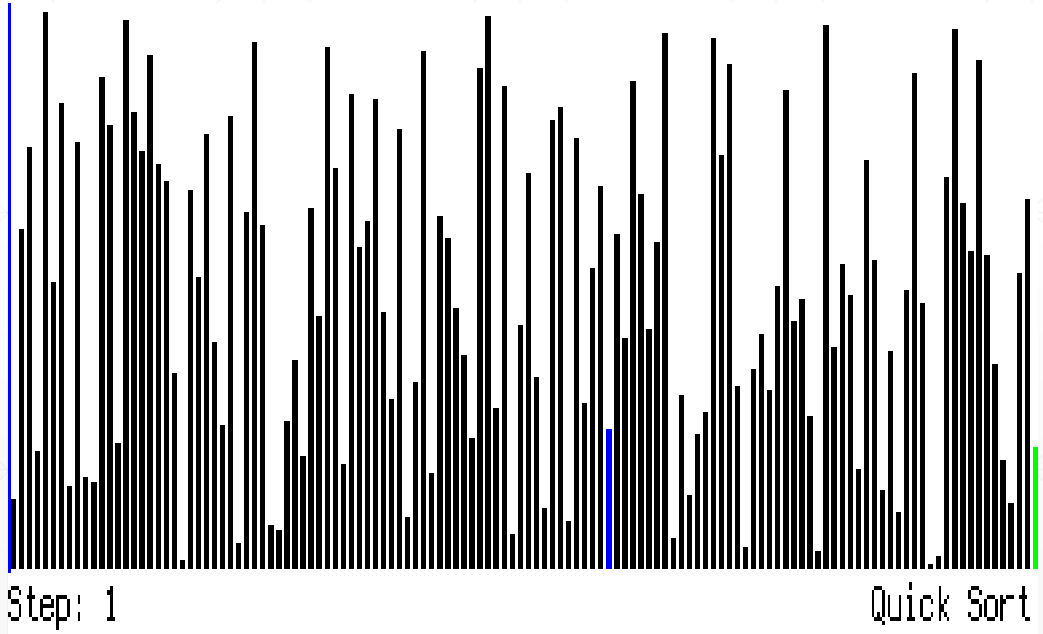
집중적으로 참조하는 특성

Quick sort가 더 빠른 이유 – locality

Merge sort



Quick sort

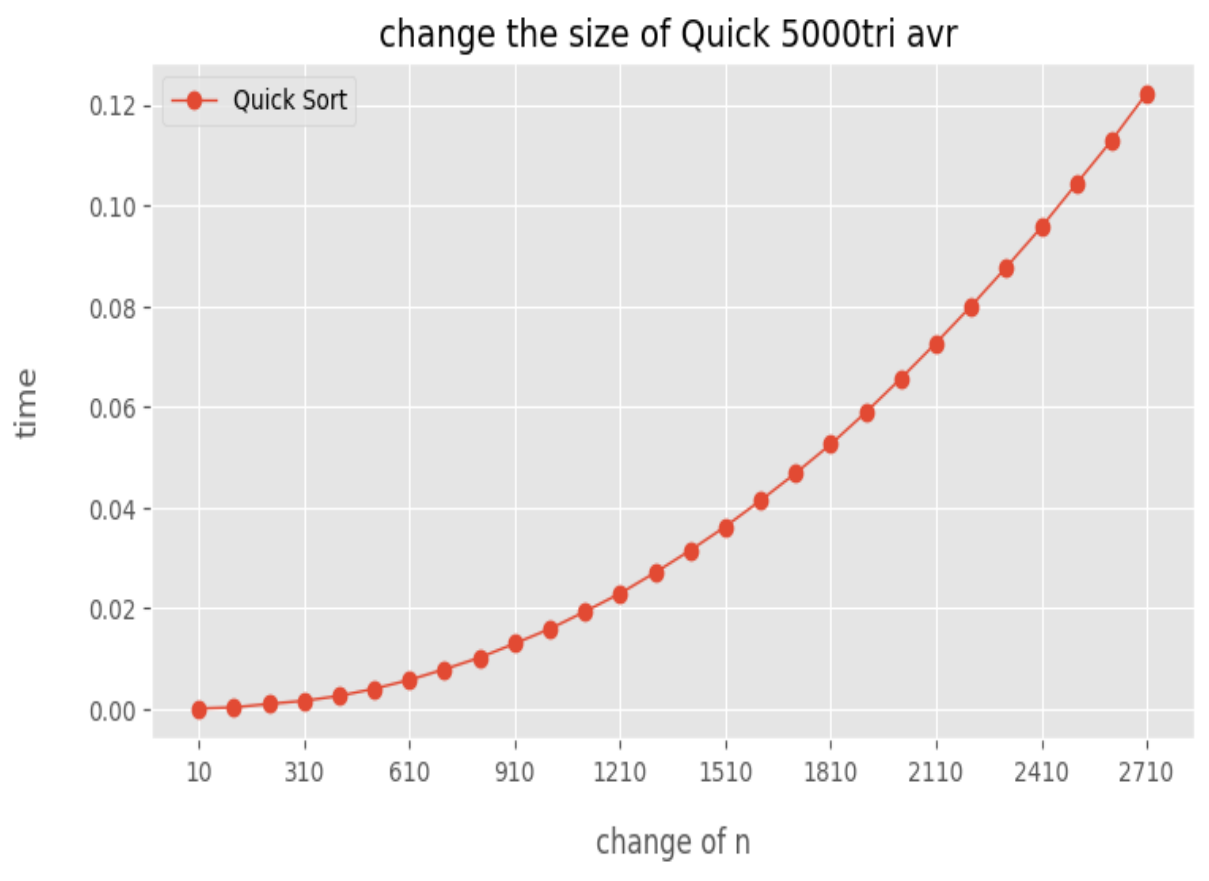


실험적 근거

```
[64]  #퀵소트 그래프 만들기용

      k = 200 # 회수를 몇번 돌릴지
      list_number = 1000 # 몇개의 리스트를 뽑을지
      Quick_run_time = [] #런타임을 모아두는 리스트
      i = 0

      while i < k:
        n= random.sample(non_k_int,list_number)
        n = list(n)
        start_time = time.time() #시작시간설정
        quick_sort(n) # 해당코드 삽입 quick_sort(n) / bubble_sort(n) / timSort(n) / radixSort(n)
        end_time = time.time() #끝나는 시간
        Quick_run_time.append(end_time - start_time) #두개 빼서 리스트에 추가
        i += 1 #횟수 세기
      quick_avr_run_time = sum(Quick_run_time) / k

      print(quick_avr_run_time)
```

```
0.005754336118698l2
```

표본의 크기에 따른 시간 변화량



Quick sort 5000 tri avr

n=110

0.0003191864490509033 4 (약 0.0003초)

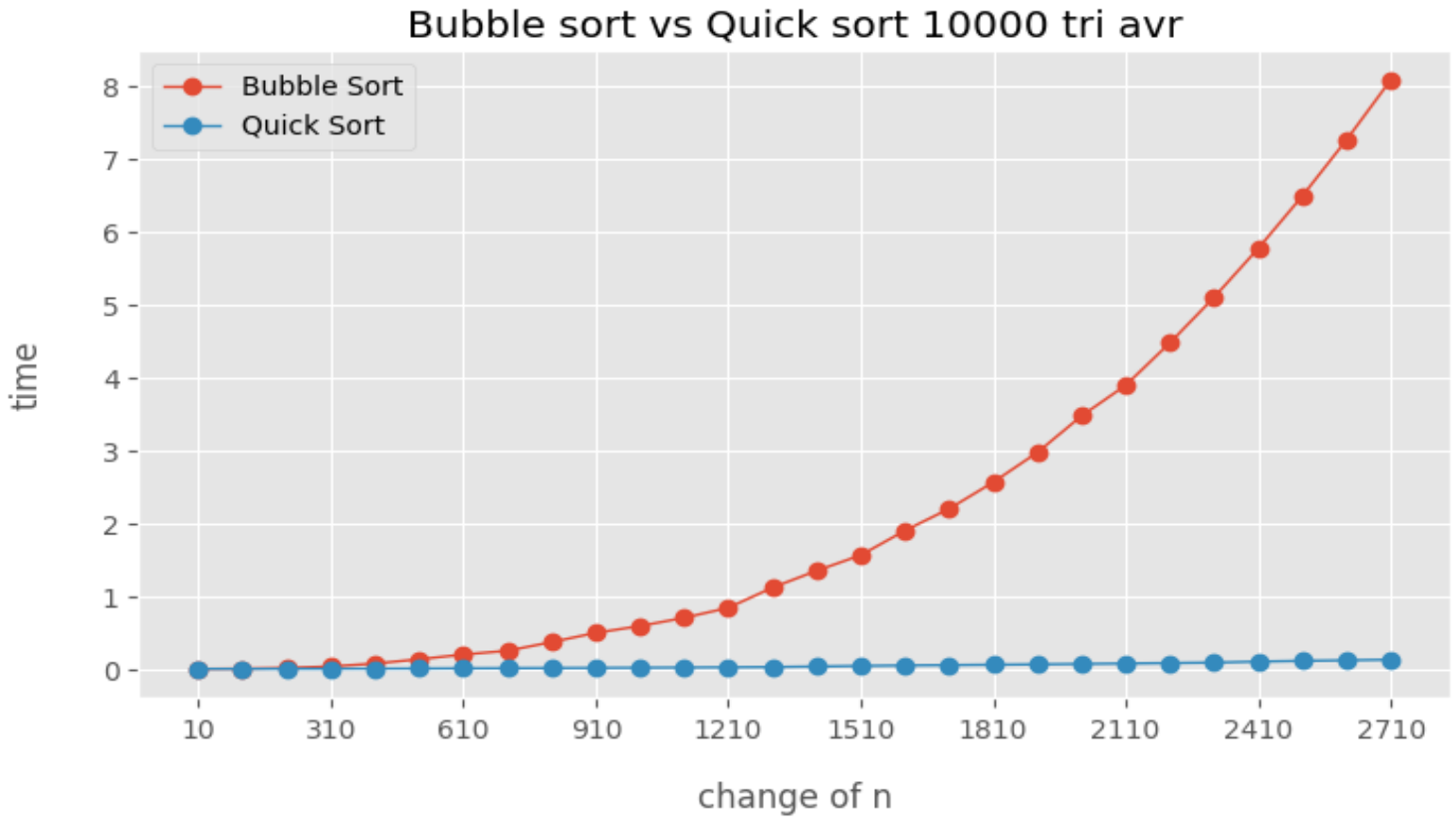
n=1010

0.016221110820770265 (약 0.016초)

n=2010

0.06844034433364868 (약 0.07초)

Bubble sort VS Quick sort KRX



Bubble vs Quick sort

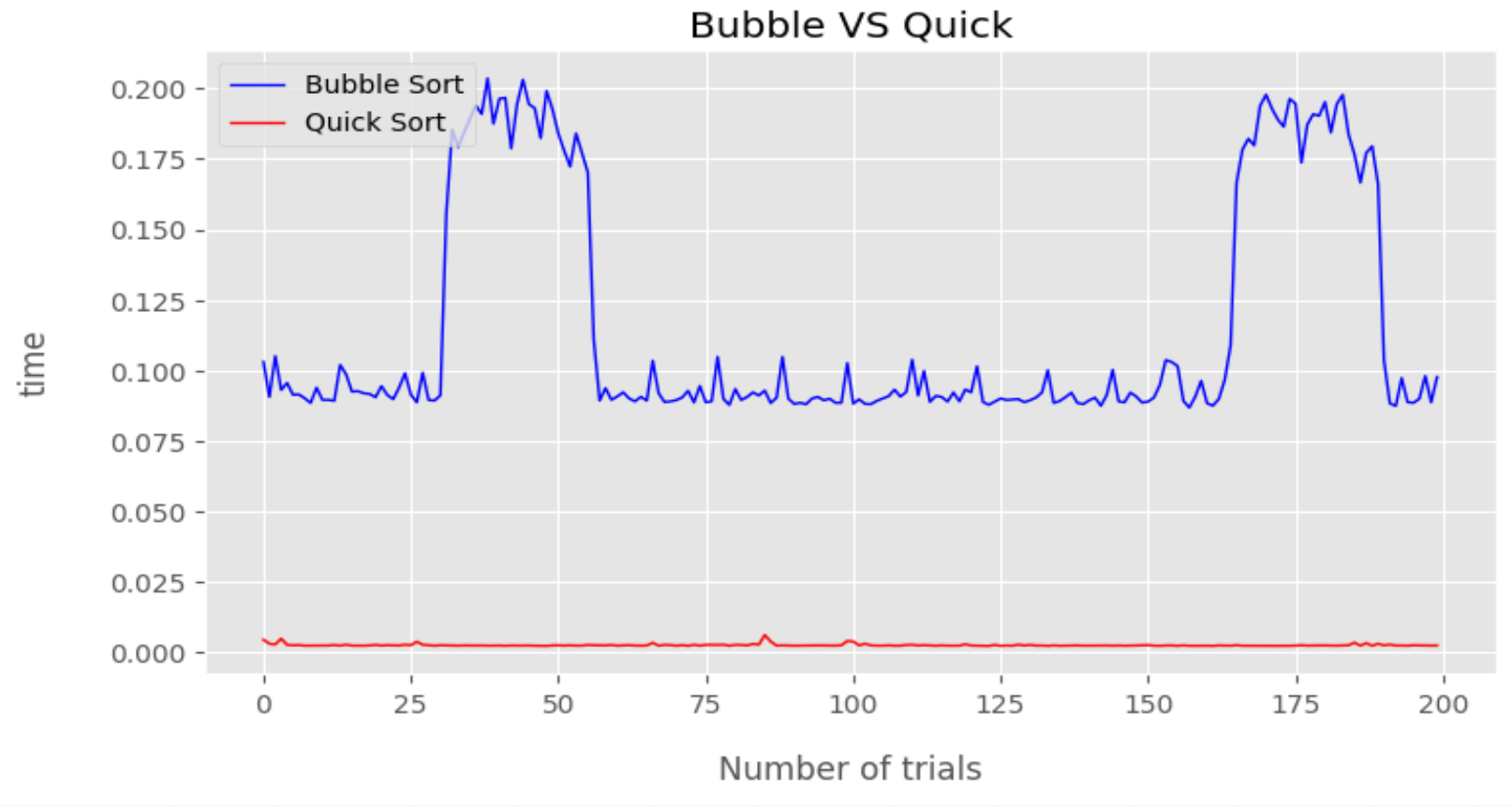
n=1010

Quick (약 0.016초)
Bubble (약 0.6초)

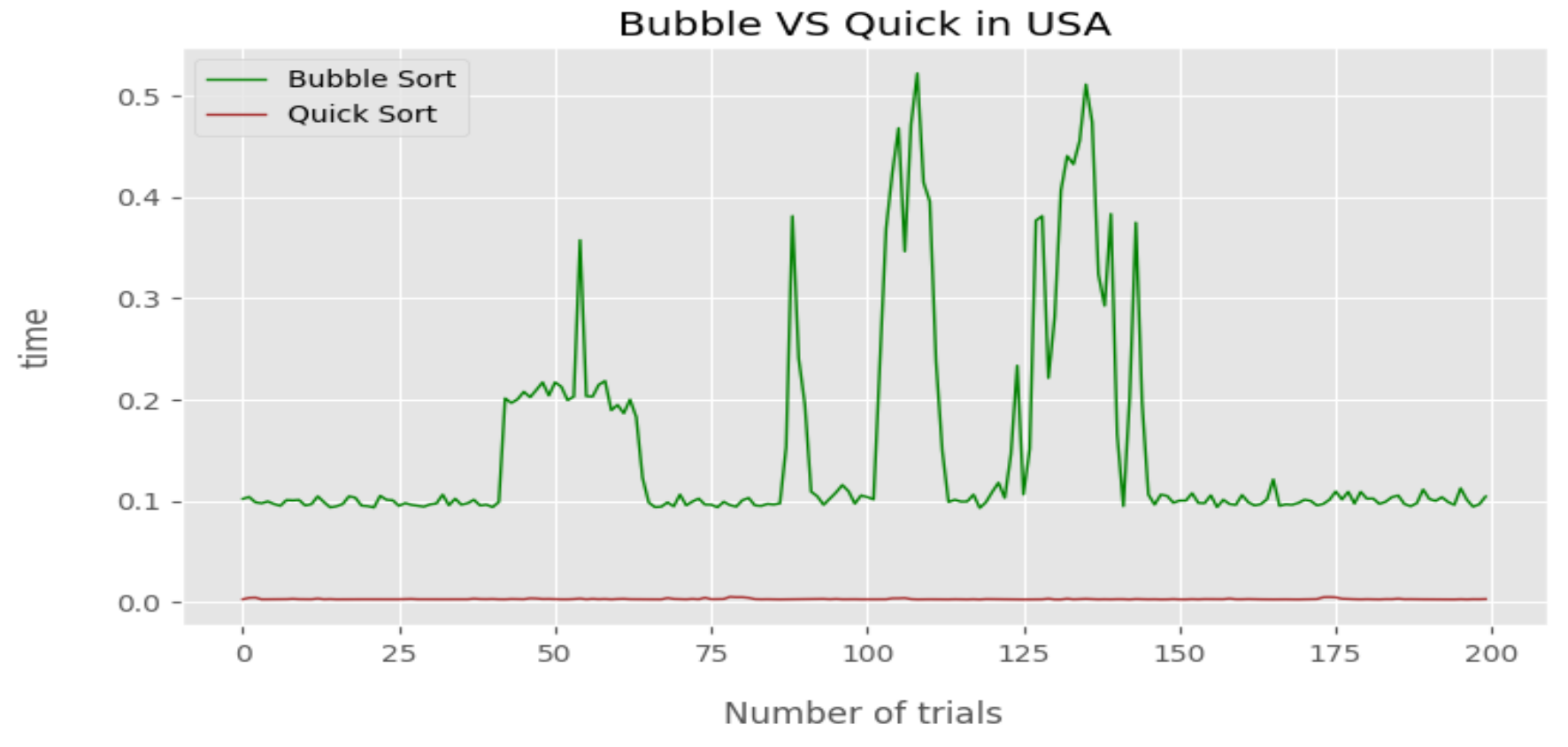
n=2010

Quick (약 0.07초)
Bubble (약 3.5초)

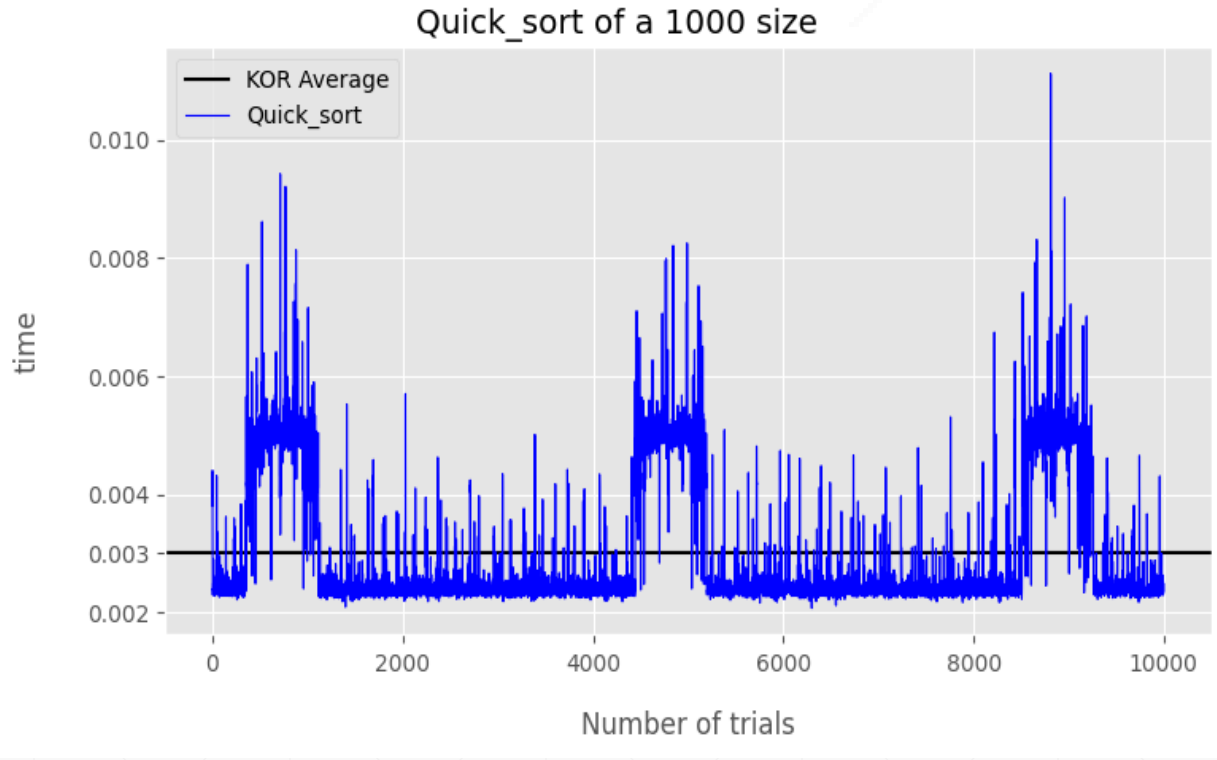
Bubble sort VS Quick sort KRX

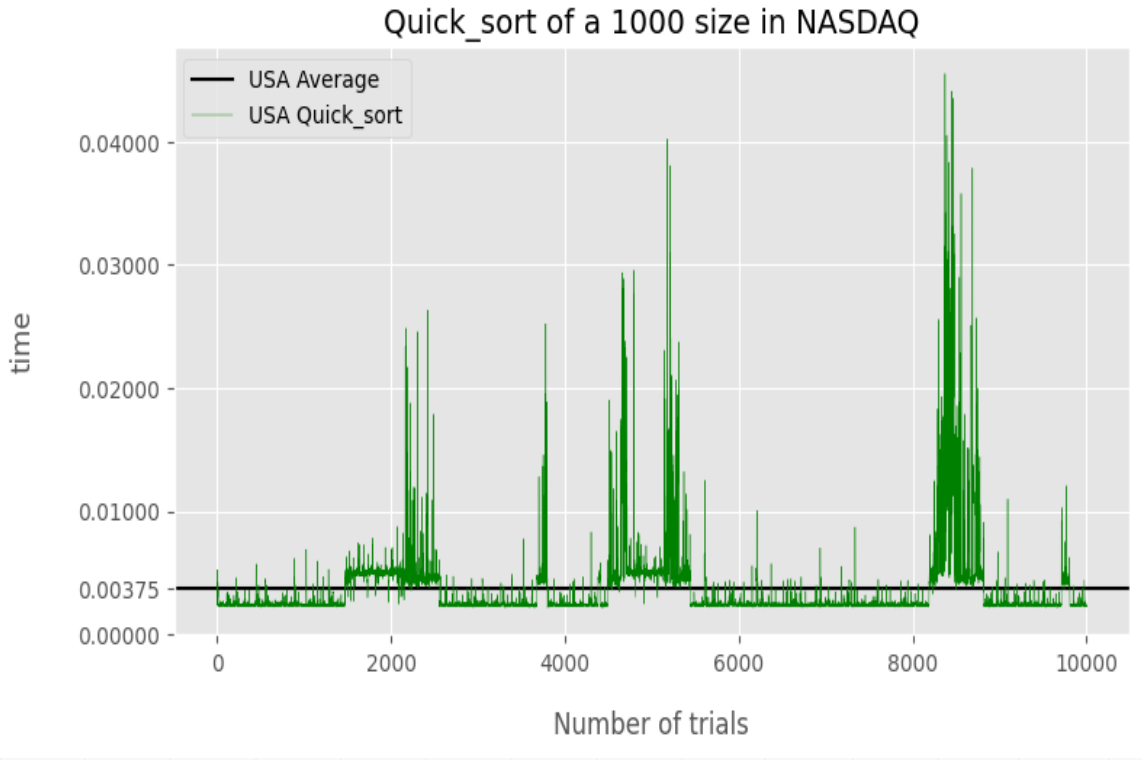


Bubble sort VS Quick sort NASDAQ



Quick sort 분포도

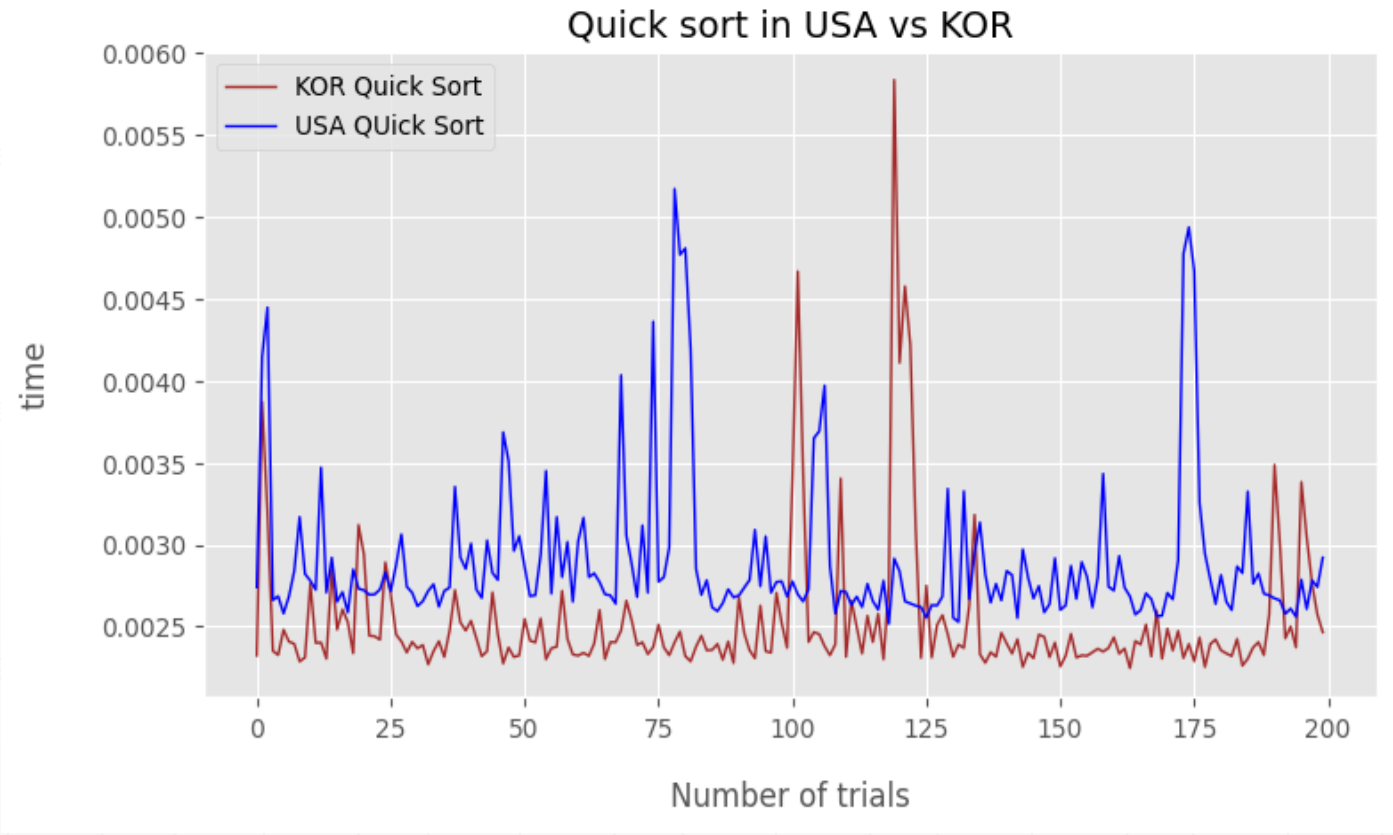




KRX 사이즈 1000개 고정
평균시간- 0.0030060745716094972

NASDAQ 사이즈 1000개 고정
평균시간-0.0037472602128982542

Quick sort NASDAQ VS KRX



KRX 경우
숫자 6개를 정렬하는 시간

NASDAQ
알파벳 4~5개를 정렬하는 시간

저희 조의 예상
NASDAQ의 정렬시간이 더 빠를 것

실제 실험 결과
KRX가 더 빠른 결과가 나옴

Quick sort 특징

장점

1. 속도가 빠르다.
2. Bubble sort 에 비해 사용하는 메모리 공간이 적다.

단점

정렬된 리스트의 경우 수행시간이 오래 걸린다.

최악의 경우 개선방안

1. 랜덤 기준점(Pivot) 선택

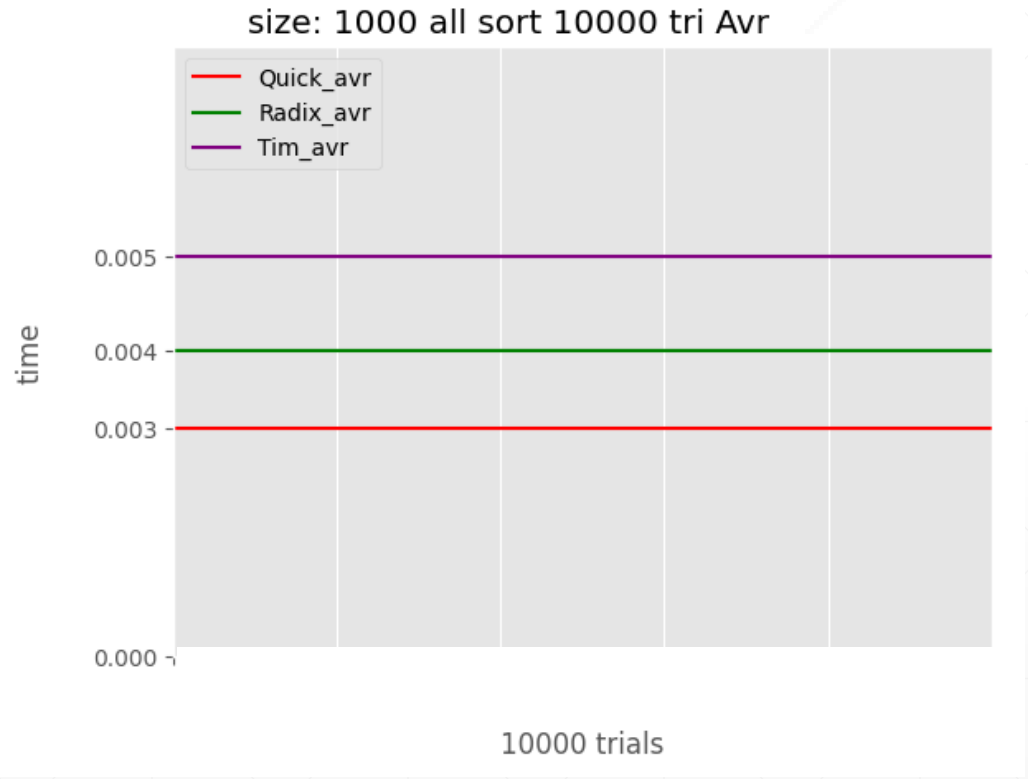
=> 최악의 경우가 나올 확률이 매우 낮아짐

2. Median Of Three Pivot

Pivot 선택할 때 3개의 원소를 후보로 두고 그 중 중간 값을

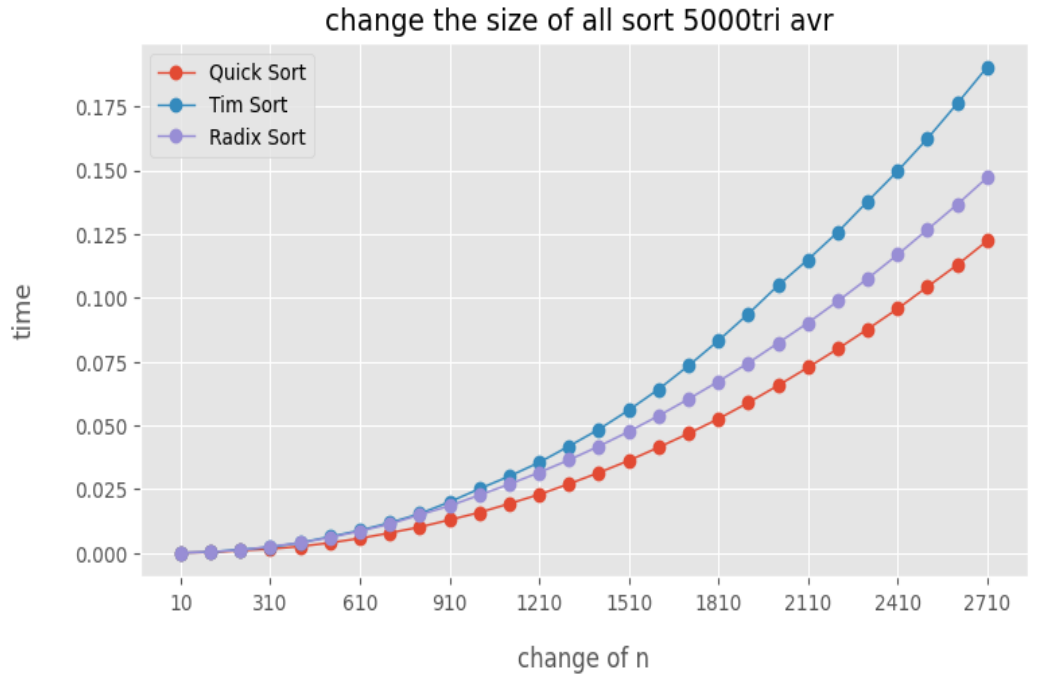
Pivot으로 설정 => 최악의 경우는 반드시 피함

소개한 3개의 정렬 시간 복잡도 비교



- X축은 시행횟수, y축은 시간복잡도인 그래프
- 시행횟수를 고정했을 때, 각 정렬들의 시간복잡도를 나타낸 그래프
- 평균적으로 퀵, 기수, 팀 소트 순으로 빠른 것을 알 수 있다.

소개한 3개의 정렬 시간 복잡도 비교



- 사이즈를 x축, 시간 복잡도를 y로 한 그래프
- 사이즈가 커질 때 각 정렬들의 시간 복잡도를 비교한 그래프
- 사이즈가 올라가도 퀵, 기수, 팀 정렬 순으로 빠른 것을 관찰할 수 있다.

감사합니다.