R E P O R T

실습 및 과제 2

과목명	알고리즘및실습
분반	2 분 반
교수	한 연 희
학번	2020136129
이름	최 수 연
제출일	2022년 4월 5일 화요일

목차

서론

병합 정렬 알고리즘은 여러 개의 정렬된 자료의 집합을 병합하여 한 개의 정렬된 집합으로 만드는 방법이다. 대표적인 분할 정복 알고리즘으로 부분집합으로 분할하고, 각 부분집합에 대해서 정렬 작업을 완성한 후 정렬된 부분집합들을 다시 결합하는 알고리즘이다.

퀵 정렬 알고리즘 역시 분할 정복 알고리즘 기법을 사용하는데, 병합 정렬 알고리즘과는 달리 기준값 즉, 피봇(Pivot)을 사용한다. 피봇이란 일반적으로 전체 원소 중에서 첫 번째, 가운데나 마지막에 위치한 원소를 선택하는 것인데, 퀵 정렬 알고리즘은 정렬할 전체 원소에 대해 정렬을 수행하지 않고 기준값이 되는 피봇을 중심으로, 왼쪽 부분집합과 오른쪽 부분집합으로 분할하여 정렬하는 방법이다.

마지막은 삼진 검색 알고리즘에 관한 문제로, 본 알고리즘은 찾는 아이템이 있을 만한 부분리스트에서 아이템을 검색하는데 해당 부분리스트를 다시 거의 같은 크기의 3개의 부분리스트로 재귀적 분할한다. 따라서 아이템을 찾거나 아이템이 리스트에 없다고 결정될 때까지 해당 과정을 되풀이하는 방식이다.

본 과제는 병합 정렬과 퀵 정렬 알고리즘을 python 프로그램으로 작성하고, 병합 정렬 및 퀵 정렬 알고리즘을 주어진 리스트를 사용하여 단계별로 정렬하는 전체 과정을 직접 수기로 작성한다. 또한 python 프로그램으로 작성한 병합 정렬과 퀵 정렬 알고리즘 코드에 주어진 리스트를 넣은 결과를 수기로 작성한 결과와 일치하는지 비교한다. 마지막 문제는 n개의 아이템을 가진 정렬된 리스트를 거의 n/3개 아이템을 가진 3개의 부분 리스트로 분할하여 검색하는 알고리즘을 의사코드 및 python 프로그램으로 작성한다.

본론

문제 1. 병합정렬 및 퀵정렬

1. 프로그램 코드

병합정렬 및 퀵정렬

```
import random
import time

def main():

    ####################
    #    Create List    #
    ####################

    n = 1000
    s = []

    for i in range(n):
        s.append(random.randint(0, n))

    s1 = s.copy()
    s2 = s.copy()
    s.sort()

    # print("s1:", s1)  # merge sort
    # print("s2:", s2)  # quick sort
    # print()

    ###################
    #    Merge Sort    #
    ###################

    start = time.perf_counter()
    merge_sort(s=s1, low=0, high=len(s1) - 1)
    end = time.perf_counter()
    print("[Merge Sort Result]")
    print("Elapsed Time: {0:0.4f}ms".format((end - start) * 1_000))
```

```
print("s1:", s1)
print("Correct:", s == s1)
print()

##################
#    Quick Sort    #
##################

start = time.perf_counter()
quick_sort(s=s2, low=0, high=len(s2) - 1)
end = time.perf_counter()
print("[Quick Sort Result]")
print("Elapsed Time: {0:0.4f}ms".format((end - start) * 1_000))
print("s2:", s2)
print("Correct:", s == s2)
print()

##############
#    TRIAL    #
##############

TRIAL = 100
total_elapsed_time_merge_sort = 0
total_elapsed_time_quick_sort = 0

print("[progressing] - TRIAL: {}".format(TRIAL))
print(">" * (TRIAL // (TRIAL // 20)))

for trial in range(TRIAL):
    # Create list
    n = 1000
    s = []
    for i in range(n):
        s.append(random.randint(0, n))

    s1 = s.copy()
    s2 = s.copy()

    # Merge Sort
    start = time.perf_counter()
```

```
        merge_sort(s=s1, low=0, high=len(s1) - 1)
        end = time.perf_counter()
        total_elapsed_time_merge_sort += end - start

        # Quick Sort
        start = time.perf_counter()
        quick_sort(s=s2, low=0, high=len(s2) - 1)
        end = time.perf_counter()
        total_elapsed_time_quick_sort += end - start

        if TRIAL >= 20 and (trial + 1) % (TRIAL // 20) == 0:
            print(">", end="", flush=True)

    print()
    print("Merge Sort - Elapsed Time: {:.5}s".format(total_elapsed_time_merge_sor
t))
    print("Quick Sort - Elapsed Time: {:.5}s".format(total_elapsed_time_quick_sor
t))

# Merge Sort 함수
def merge_sort(s, low, high):
    if low < high:
        mid = (low + high) // 2
        merge_sort(s, low, mid)
        merge_sort(s, mid + 1, high)
        merge(s, low, mid, high)

# Merge 함수
def merge(s, low, mid, high):
    tmp = [0] * (high - low + 1)
    i = low
    j = mid + 1
    t = 0

    while i <= mid and j <= high:
      if s[i] <= s[j]:
        tmp[t] = s[i]
        t+=1
        i+=1
      else:
```

```
            tmp[t] = s[j]
            t+=1
            j+=1
        while i <= mid:
          tmp[t] = s[i]
          t+=1
          i+=1
        while j <= high:
          tmp[t] = s[j]
          t+=1
          j+=1
        i = low
        t = 0
        while i <= high:
          s[i] = tmp[t]
          i+=1
          t+=1

# Quick Sort 함수
def quick_sort(s, low, high):
    if low < high:
        pivot = partition(s, low, high)
        quick_sort(s, low, pivot - 1)
        quick_sort(s, pivot + 1, high)

# Partition 함수
def partition(s, low, high):
    x = s[high]
    i = low - 1

    for j in range(low, high):
      if s[j] <= x:
        i+=1
        s[i], s[j] = s[j], s[i]

    s[i+1], s[high] = s[high], s[i+1]
    return i+1

if __name__ == "__main__":
    main()
```

2. 프로그램 코드 설명

병합정렬

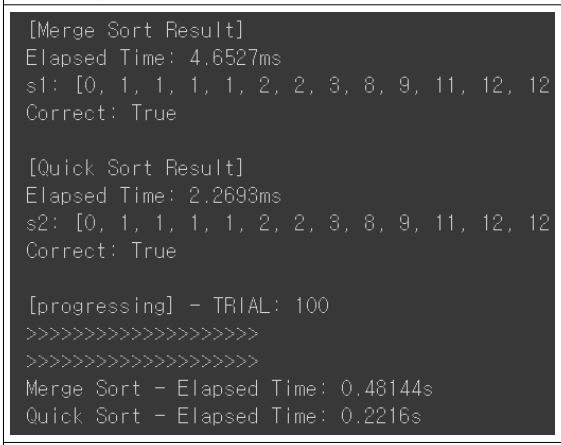
- 먼저 merge_sort 함수의 경우 리스트 s를 정렬하는 함수인데, 만약 리스트의 첫 번째 위치의 값인 low가 맨 마지막 위치의 값인 high보다 작을 때 해당 if문을 실행하도록 한다.
- 본 if 문에는 low와 high의 중간값 mid을 도출하고, merge_sort 함수에 mid를 사용하여 리스트의 크기가 반이 되도록 재귀 호출한다.
- 이렇게 전반부와 후반부를 나누어 계속 merge_sort 함수를 재귀 호출을 한 후, 각 분할된 리스트들의 원소가 하나씩 남아있을 때 merge 함수를 통해 병합한다.
- merge 함수의 경우. 인자로 받은 high 값과 low 값을 사용해 low에서 high까지의 크기를 갖는 빈 리스트를 만들고, 변수 i에 low값, 즉 전반부 정렬된 리스트의 첫 번째 원소를 가리키는 값으로 설정하고, j에 low와 high의 중간값인 mid에 1을 더하여 중간값보다 하나 더 큰 값, 즉 후반부 정렬된 리스트의 첫 번째 원소를 가리키는 값으로 설정한 후, 새로운 리스트에서 원소값이 비어있는 위치를 알려줄 변수 t를 0으로 선언한다.
- 다음으로 i가 mid보다 작거나 같고, j가 high보다 작거나 같을 때 해당 while 문이 실행되도록 한다.
- 해당 while 문에서는 위치 i에 있는 원소가 j에 있는 원소보다 작을 경우, 위치 i에 있는 원소를 빈 리스트 tmp에 넣고 t값과 i값을 1씩 증가시킨다. 만약 위치 j에 있는 원소가 i에 있는 원소보다 작을 경우, 위치 j에 있는 원소를 빈 리스트 tmp에 넣고 t값과 j값을 1씩 증가시킨다.
- 해당 while 문이 끝나면 전반부 혹은 후반부에 남아있는 원소들을 tmp에 넣기 위해 두 while 문을 실행한다. 만약 위치 i에 있는 원소들이 아직 남아있으면 tmp에 넣고 t값과 i값을 1씩 증가시킨다. 그렇지 않고 만약 위치 j에 있는 원소들이 아직 남아있으면 tmp에 넣고 t값과 j값을 1씩 증가시킨다.
- 위 while 문까지 끝나면, 빈 리스트 tmp에 정렬된 원소들을 기존 리스트 s에 순서대로 옮겨 담기 위한 while 문을 실행한다. i를 low, t를 0으로 두고, 만약 i가 high보다 작거나 같을 경우, 리스트 tmp에서 위치 t에 있는 값을 리스트 s의 i 위치로 옮긴다. 그리고 i값과 t값을 1씩 증가시킨다.

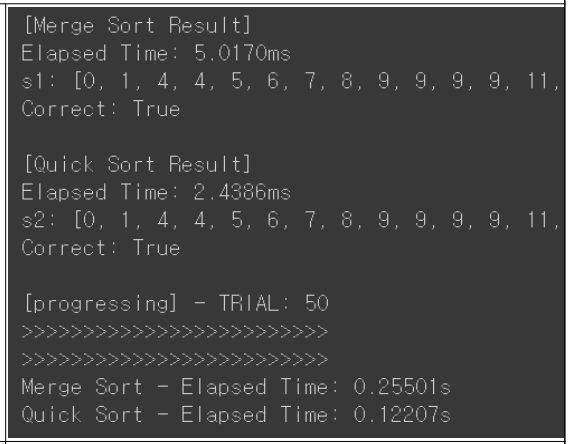
퀵정렬

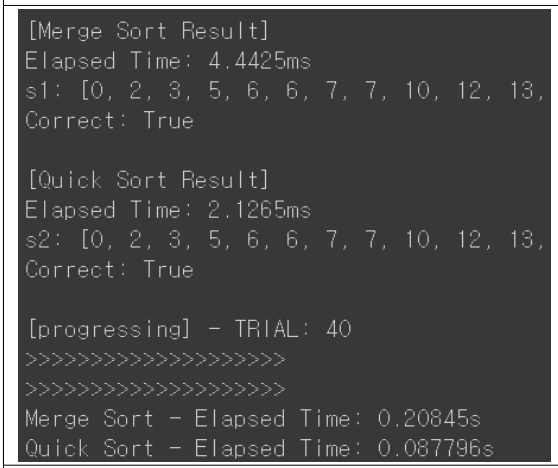
- 먼저 quick_sort 함수의 경우, 리스트 s를 정렬하는 함수인데, 만약 리스트의 첫 번째 위치의 값인 low가 맨 마지막 위치의 값인 high보다 작을 때 해당 if 문을 실행하도록 한다.
- if 문 안에서 먼저 pivot을 설정하기 위해 partition 함수를 호출하여 리스트 s와 low, high 값을 인자로 전달한다. partition 함수에서 high를 기준으로 양쪽으로 원소를 재배치한 뒤 s[high] 자리의 위치를 반환한다. 그리고 pivot을 기준으로 왼쪽 부분 리스트, 오른쪽 부분 리스트로 나누어 정렬하도록 quick_sort 함수를 low가 high보다 작지 않아질 때까지 재귀 호출한다.
- partition 함수는 high 위치에 있는 리스트 s의 값을 pivot x로 두고, i를 low-1, 즉 첫 번째 구간의 끝지점으로 둔다.
- for 문을 통해 j를 low부터 high – 1까지 반복하면서, 즉 정렬되지 않은 구역의 시작점부터 끝점까지 if 문을 통해 pivot인 x보다 작거나 같을 때 i값 1 증가 후 s[i]와 s[j]를 교환하도록 한다.
- 위 for 문이 종료되면, pivot이 되는 원소와 i+1의 위치에 있는 원소, 즉 해당 pivot보다 큰 원소들이 모여있는 구역의 첫 번째 위치의 원소를 서로 교환하고, 해당 위치를 반환한다. 그렇게 되면 pivot은 오름차순 기준, 제 위치를 찾아가고, pivot의 왼쪽에는 pivot보다 작은 값, pivot의 오른쪽에는 pivot보다 큰 값의 원소들이 모이게 된다.

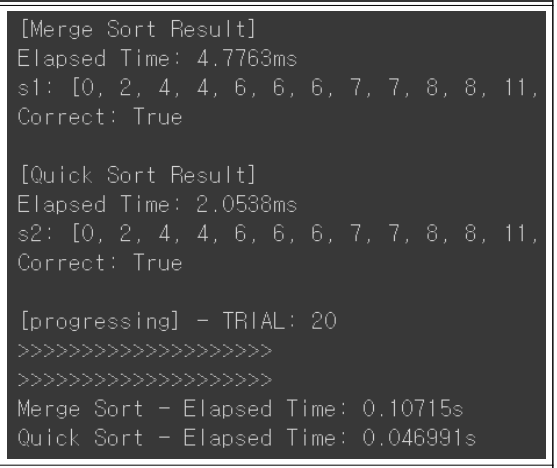
3. 프로그램 수행화면

n = 1,000



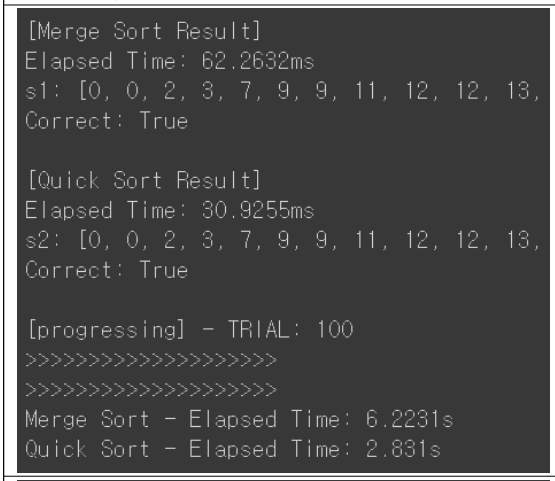


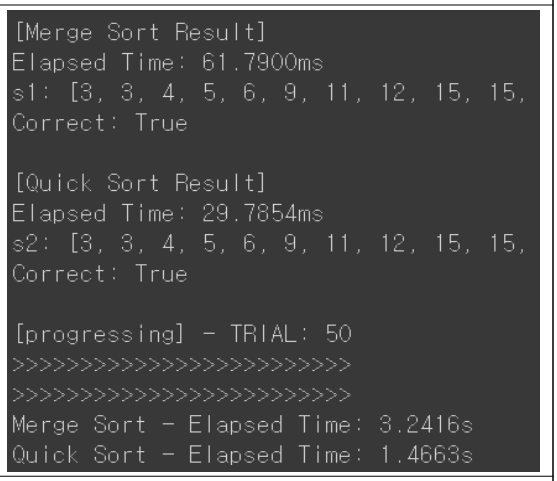


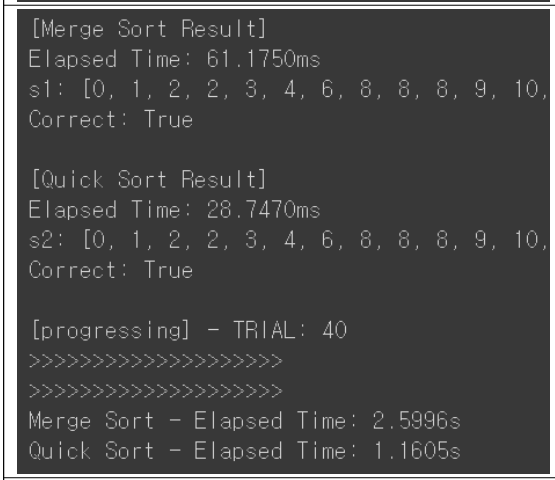


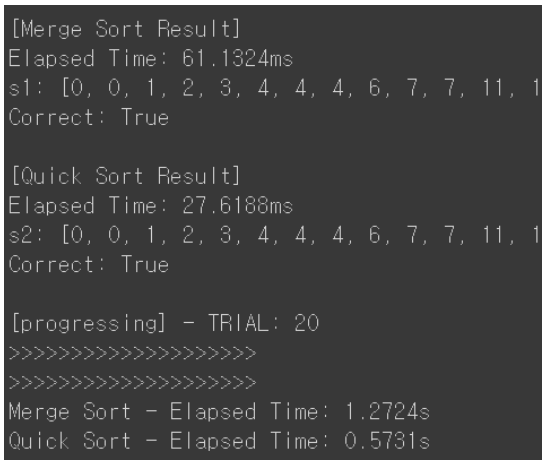
1. 첫 번째 사진은 각 함수를 한 번만 실행 시, Merge Sort는 **4.6527ms,** Quick Sort는 **2.2693ms**가 나온다. 그리고 TRIAL을 100번 수행했을 때 각 함수의 결과는 0.48144s, 0.2216s인데, TRIAL이 100번이므로 결과에 100을 나누어주고, 시간 단위 ‘s’를 ‘ms’로 바꾸기 위해 1000을 곱해주면 각 함수 한 번의 실행 시간 결과는 각각 **4.8144ms, 2.216ms**이다. 즉 한 번만 실행했을 때와 TRIAL로 여러 번 실행했을 때의 각 함수 실행 시간 결과가 거의 일치한다는 것을 알 수 있다.
2. 두 번째 사진은 각 함수를 한 번만 실행 시, Merge Sort는 **5.0170ms**, Quick Sort는 **2.4386ms**가 나온다. 그리고 TRIAL을 50번 수행했을 때 각 함수의 결과는 0.25501s, 0.12207s인데, TRIAL이 50번이므로 결과에 50을 나누어주고, 시간 단위 ‘s’를 ‘ms’로 바꾸기 위해 1000을 곱해주면 각 함수 한 번의 실행 시간 결과는 각각 **5.1002ms, 2.4414ms**이다. 즉 한 번만 실행했을 때와 TRIAL로 여러 번 실행했을 때의 각 함수 실행 시간 결과가 거의 일치한다는 것을 알 수 있다.
3. 세 번째 사진도 위 1, 2번과 같은 방식으로 계산해보면, 각 함수를 한 번만 실행 시, Merge Sort는 **4.4425ms,** Quick Sort는 **2.1265ms**가 나온다. 또한 TRIAL을 40번으로 하여 수행하면, 각 함수 한 번 실행 시간 결과는 각각 **5.21125ms, 2.1949ms**이다.
4. 네 번째 사진도 마찬가지로 각 함수를 한 번만 실행 시 Merge Sort는 **4.7763ms,** Quick Sort는 **2.0538ms**이고, TRIAL을 20번으로 하여 수행하면, 각 함수 한 번 실행 시간 결과는 각각 **5.3575ms, 2.34955ms**이다.

n = 10,000



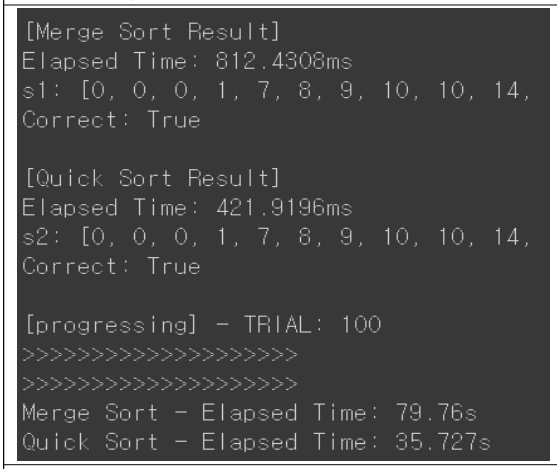


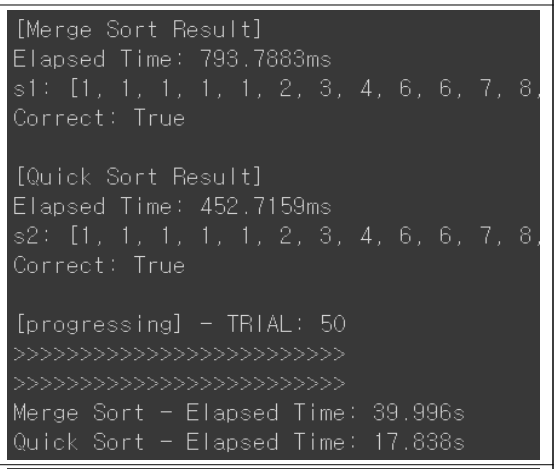


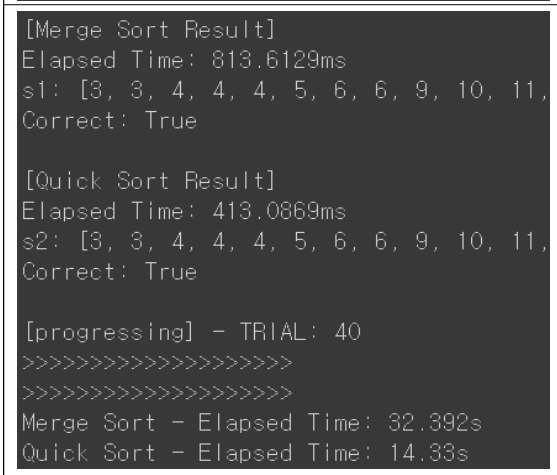


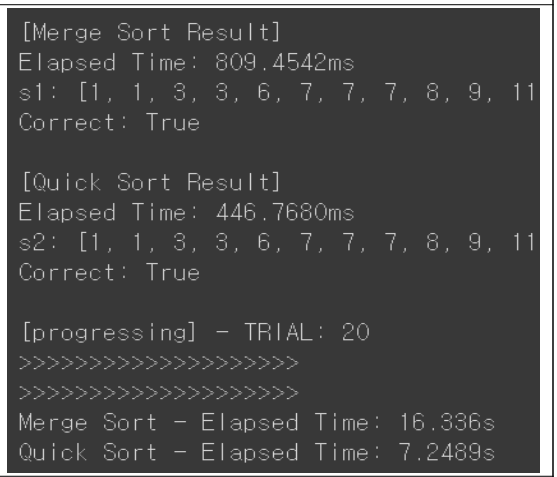
1. 첫 번째 사진은 각 함수를 한 번만 실행 시, Merge Sort는 **62.2632ms,** Quick Sort는 **30.9255ms**가 나온다. 그리고 TRIAL을 100번 수행했을 때 각 함수의 결과는 6.2231s, 2.831s인데, TRIAL이 100번이므로 결과에 100을 나누어주고, 시간 단위 ‘s’를 ‘ms’로 바꾸기 위해 1000을 곱해주면 각 함수 한 번의 실행 시간 결과는 각각 **62.231ms, 28.31ms**이다. 즉 한 번만 실행했을 때와 TRIAL로 여러 번 실행했을 때의 각 함수 실행 시간 결과가 거의 일치한다는 것을 알 수 있다.
2. 두 번째 사진은 각 함수를 한 번만 실행 시, Merge Sort는 **61.7900ms**, Quick Sort는 **29.7854ms**가 나온다. 그리고 TRIAL을 50번 수행했을 때 각 함수의 결과는 3.2416s, 1.4663s인데, TRIAL이 50번이므로 결과에 50을 나누어주고, 시간 단위 ‘s’를 ‘ms’로 바꾸기 위해 1000을 곱해주면 각 함수 한 번의 실행 시간 결과는 각각 **64.832ms, 29.326ms**이다. 즉 한 번만 실행했을 때와 TRIAL로 여러 번 실행했을 때의 각 함수 실행 시간 결과가 거의 일치한다는 것을 알 수 있다.
3. 세 번째 사진도 위 1, 2번과 같은 방식으로 계산해보면, 각 함수를 한 번만 실행 시, Merge Sort는 **61.1750ms,** Quick Sort는 **28.7470ms**가 나온다. 또한 TRIAL을 40번으로 하여 수행하면, 각 함수 한 번 실행 시간 결과는 각각 **64.99ms, 29.0125ms**이다.
4. 네 번째 사진도 마찬가지로 각 함수를 한 번만 실행 시 Merge Sort는 **61.1324ms,** Quick Sort는 **27.6188ms**이고, TRIAL을 20번으로 하여 수행하면, 각 함수 한 번 실행 시간 결과는 각각 **63.62ms, 28.655ms**이다.

n = 100,000









1. 첫 번째 사진은 각 함수를 한 번만 실행 시, Merge Sort는 **812.4308ms,** Quick Sort는 **421.9196ms**가 나온다. 그리고 TRIAL을 100번 수행했을 때 각 함수의 결과는 79.76s, 35.727s인데, TRIAL이 100번이므로 결과에 100을 나누어주고, 시간 단위 ‘s’를 ‘ms’로 바꾸기 위해 1000을 곱해주면 각 함수 한 번의 실행 시간 결과는 각각 **797.6ms, 357.27ms**이다. 즉 한 번만 실행했을 때와 TRIAL로 여러 번 실행했을 때의 각 함수 실행 시간 결과가 거의 일치한다는 것을 알 수 있다.
2. 두 번째 사진은 각 함수를 한 번만 실행 시, Merge Sort는 **793.7883ms**, Quick Sort는 **452.7159ms**가 나온다. 그리고 TRIAL을 50번 수행했을 때 각 함수의 결과는 39.996s, 17.838s인데, TRIAL이 50번이므로 결과에 50을 나누어주고, 시간 단위 ‘s’를 ‘ms’로 바꾸기 위해 1000을 곱해주면 각 함수 한 번의 실행 시간 결과는 각각 **799.92ms, 356.76ms**이다. 즉 한 번만 실행했을 때와 TRIAL로 여러 번 실행했을 때의 각 함수 실행 시간 결과가 거의 일치한다는 것을 알 수 있다.
3. 세 번째 사진도 위 1, 2번과 같은 방식으로 계산해보면, 각 함수를 한 번만 실행 시, Merge Sort는 **813.6129ms,** Quick Sort는 **413.0869ms**가 나온다. 또한 TRIAL을 40번으로 수행하면, 각 함수 한 번 실행 시간 결과는 각각 **809.8ms, 358.25ms**이다.
4. 네 번째 사진도 마찬가지로 각 함수를 한 번만 실행 시 Merge Sort는 **809.4542ms,** Quick Sort는 **446.7680ms**이고, TRIAL을 20번으로 하여 수행하면, 각 함수 한 번 실행 시간 결과는 각각 **816.8ms, 362.445ms**이다.

n = 1,000,000

```
[Merge Sort Result]
Elapsed Time: 11119.0846ms
Correct: True

[Quick Sort Result]
Elapsed Time: 5137.2133ms
Correct: True

[progressing] - TRIAL: 100
>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>
Merge Sort - Elapsed Time: 1100.5s
Quick Sort - Elapsed Time: 521.61s
```

TRIAL=100일 때 코드 실행 시간: 약 27분

```
[Merge Sort Result]
Elapsed Time: 11234.1906ms
Correct: True

[Quick Sort Result]
Elapsed Time: 5481.6774ms
Correct: True

[progressing] - TRIAL: 50
>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>
Merge Sort - Elapsed Time: 571.3s
Quick Sort - Elapsed Time: 273.08s
```

TRIAL=50일 때 코드 실행 시간: 약 14분

```
[Merge Sort Result]
Elapsed Time: 11166.3013ms
Correct: True

[Quick Sort Result]
Elapsed Time: 5222.0970ms
Correct: True

[progressing] - TRIAL: 40
>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>
Merge Sort - Elapsed Time: 450.2s
Quick Sort - Elapsed Time: 213.88s
```

TRIAL=40일 때 코드 실행 시간: 약 11분

```
[Merge Sort Result]
Elapsed Time: 11634.6569ms
Correct: True

[Quick Sort Result]
Elapsed Time: 5469.8830ms
Correct: True

[progressing] - TRIAL: 20
>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>
Merge Sort - Elapsed Time: 227.98s
Quick Sort - Elapsed Time: 108.62s
```

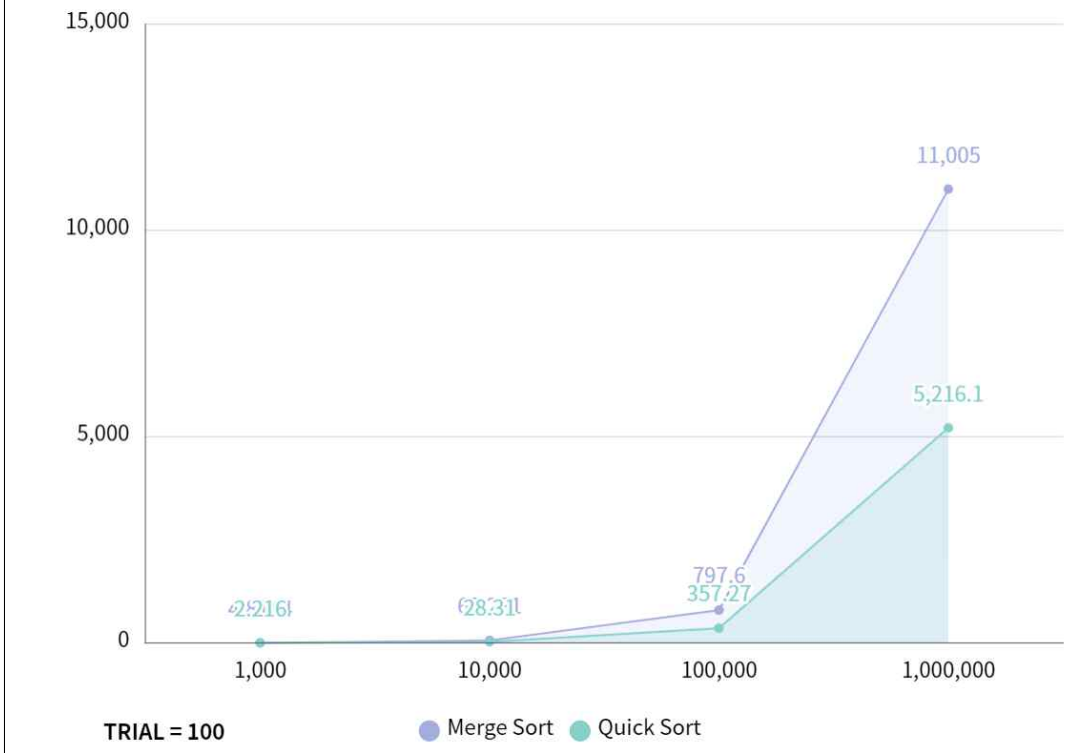
TRIAL=20일 때 코드 실행 시간: 약 6분

※ num을 1,000,000으로 설정했을 경우, IOPub data rate exceeded 오류가 계속 발생하여 리스트 s1과 s2을 전체 출력하는 부분은 주석 처리하고 수행시간과 Correct만 결과를 출력하였다.

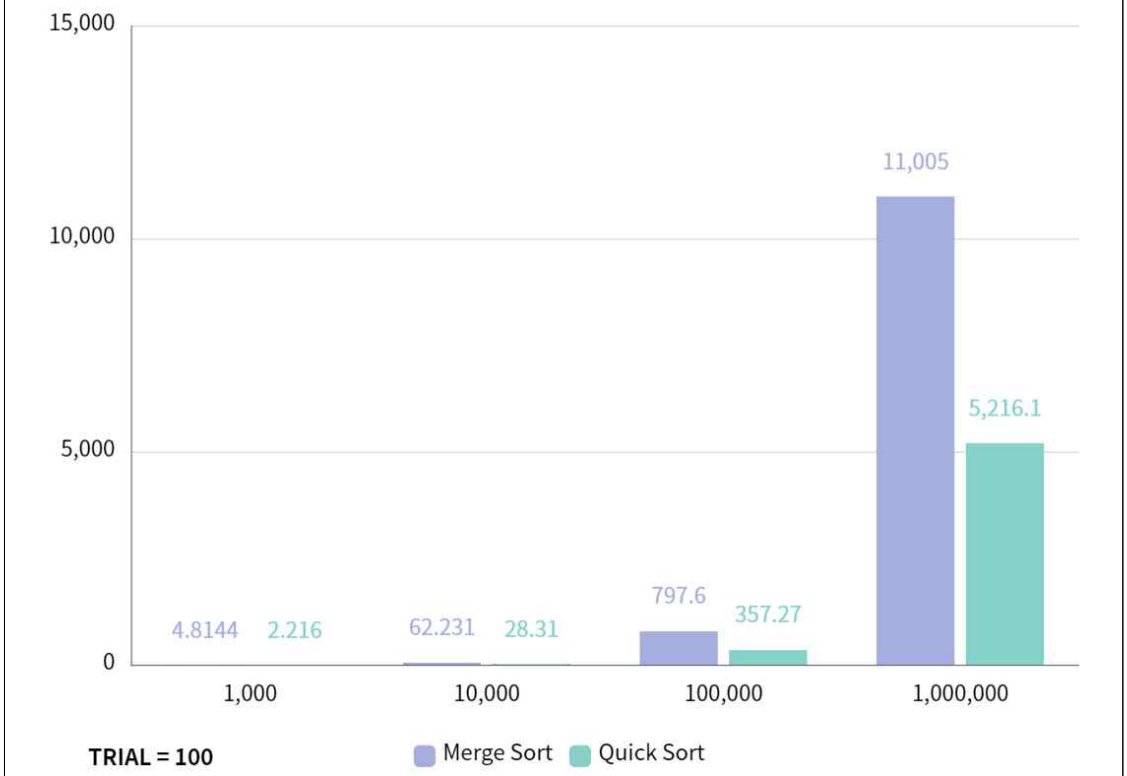
1. 첫 번째 사진은 각 함수를 한 번만 실행 시, Merge Sort는 **11119.0846ms,** Quick Sort는 **5137.2133ms**가 나온다. 그리고 TRIAL을 100번 수행했을 때 각 함수의 결과는 1100.5s, 521.61s인데, TRIAL이 100번이므로 결과에 100을 나누어주고, 시간 단위 ‘s’를 ‘ms’로 바꾸기 위해 1000을 곱해주면 각 함수 한 번의 실행 시간 결과는 각각 **11005ms, 5216.1ms**이다. 즉 한 번만 실행했을 때와 TRIAL로 여러 번 실행했을 때의 각 함수 실행 시간 결과가 거의 일치한다는 것을 알 수 있다.
2. 두 번째 사진은 각 함수를 한 번만 실행 시, Merge Sort는 **11234.1906ms**, Quick Sort는 **5481.6774ms**가 나온다. 그리고 TRIAL을 50번 수행했을 때 각 함수의 결과는 571.3s, 273.08s인데, TRIAL이 50번이므로 결과에 50을 나누어주고, 시간 단위 ‘s’를 ‘ms’로 바꾸기 위해 1000을 곱해주면 각 함수 한 번의 실행 시간 결과는 각각 **11426ms**, **5461.6ms**이다. 즉 한 번만 실행했을 때와 TRIAL로 여러 번 실행했을 때의 각 함수 실행 시간 결과가 거의 일치한다는 것을 알 수 있다.
3. 세 번째 사진도 위 1, 2번과 같은 방식으로 계산해보면, 각 함수를 한 번만 실행 시, Merge Sort는 **11166.3013ms,** Quick Sort는 **5222.0970ms**가 나온다. 또한 TRIAL을 40번으로 하여 수행하면, 함수 한 번 실행 시간 결과는 각각 **11255ms, 5347ms**이다.
4. 네 번째 사진도 마찬가지로 각 함수를 한 번만 실행 시 Merge Sort는 **11634.6569ms,** Quick Sort는 **5469.8830ms**이고, TRIAL을 20번으로 하여 수행하면, 각 함수 한 번 실행 시간 결과는 각각 **11399ms, 5431ms**이다.

4. 수행시간 그래프

TRIAL = 100

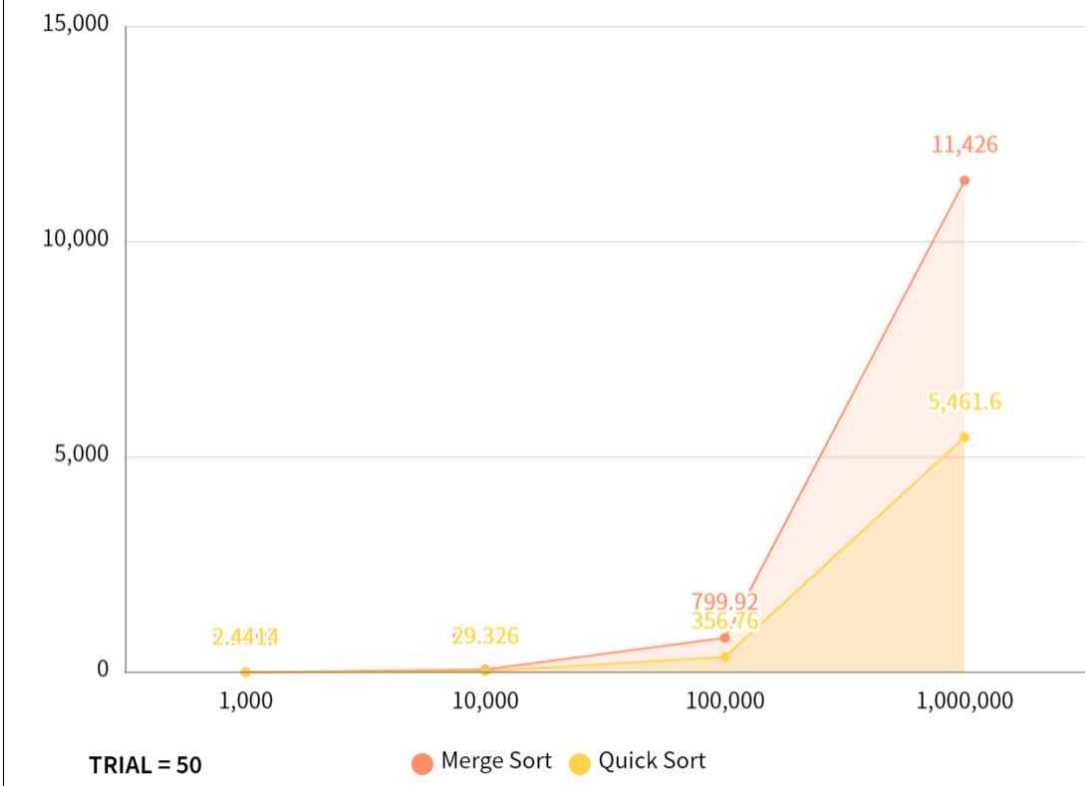


표[1-1]

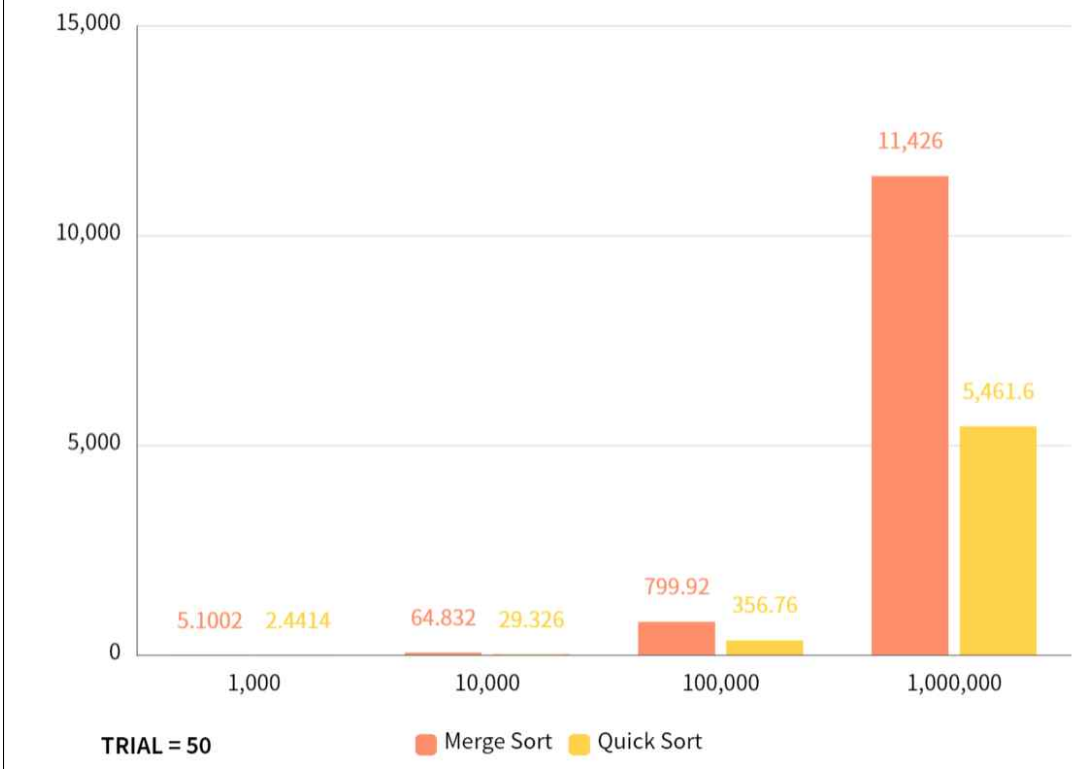


표[1-2]

TRIAL = 50

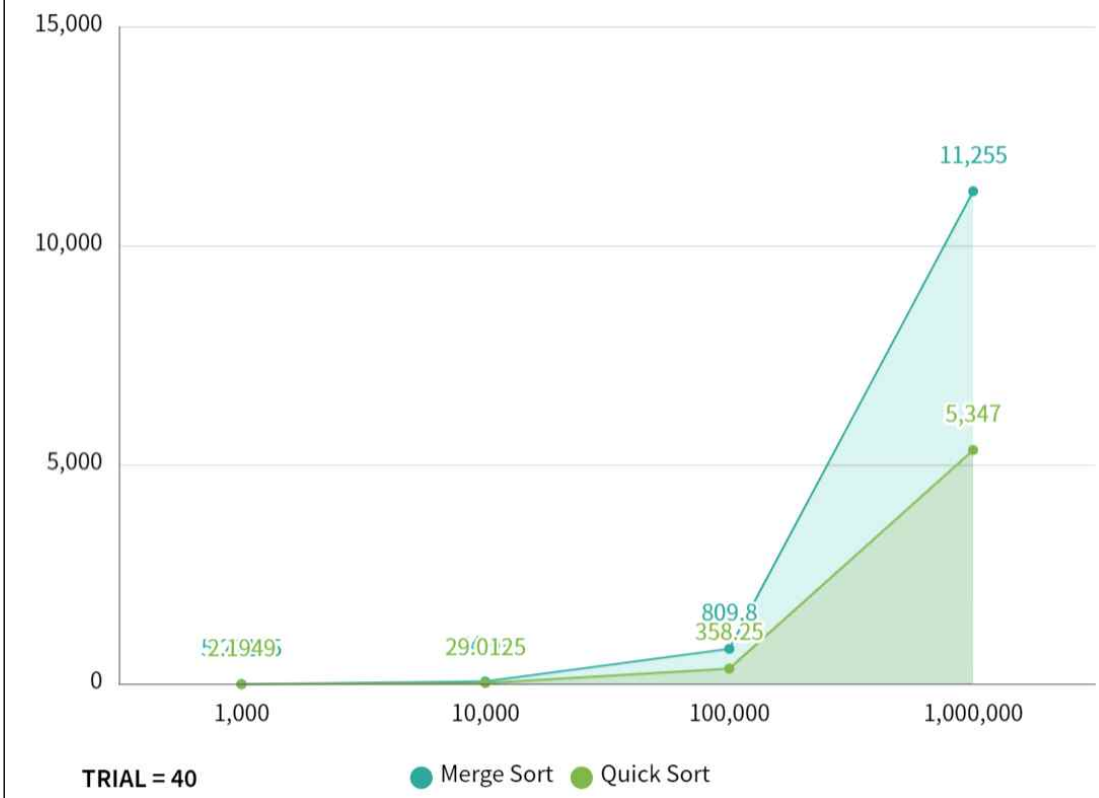


표[1-3]

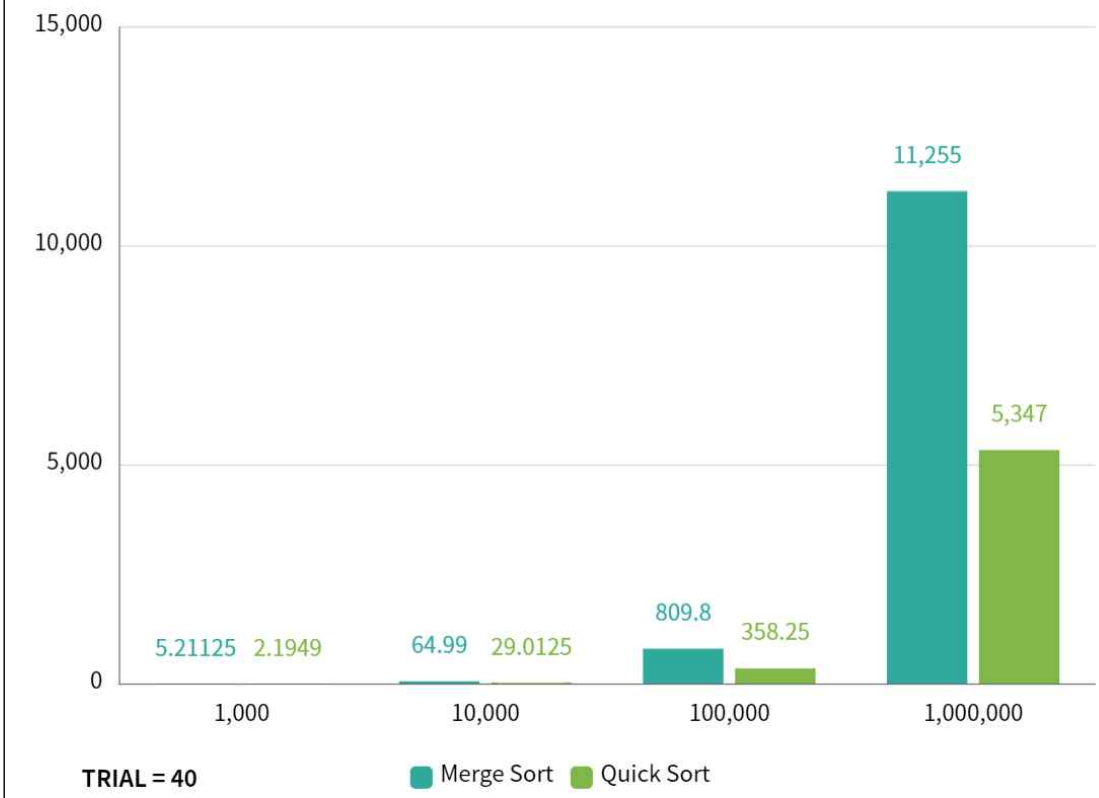


표[1-4]

TRIAL = 40

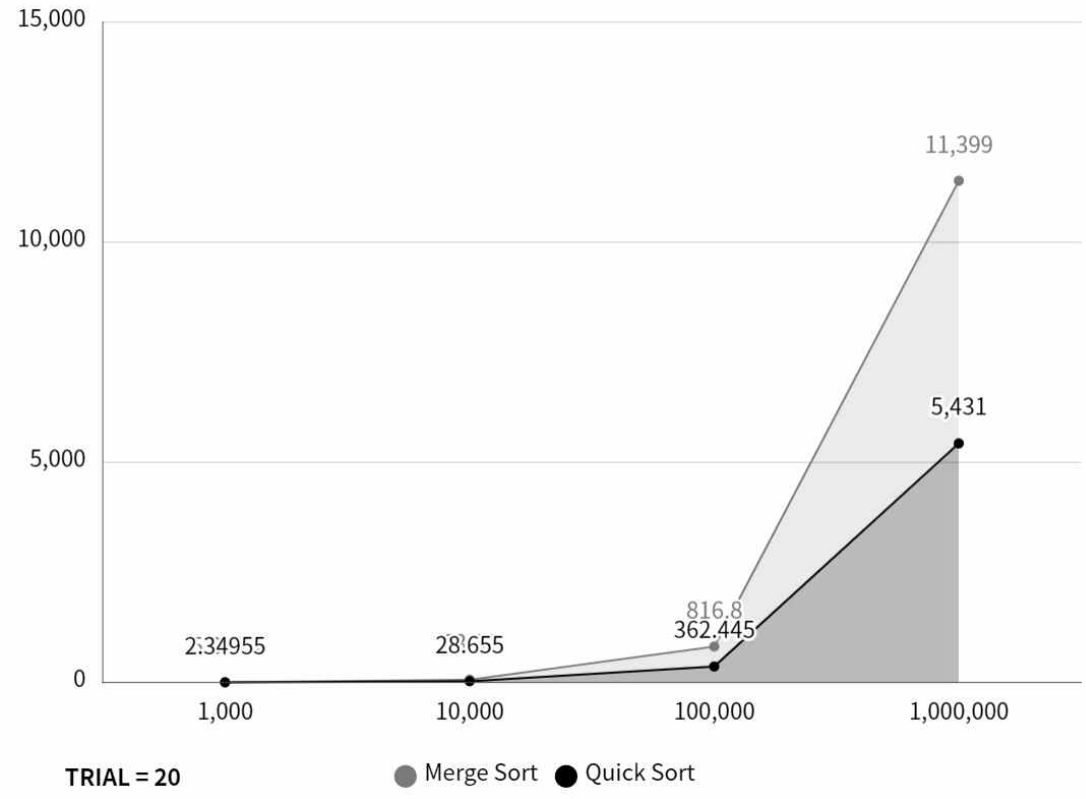


표[1-5]

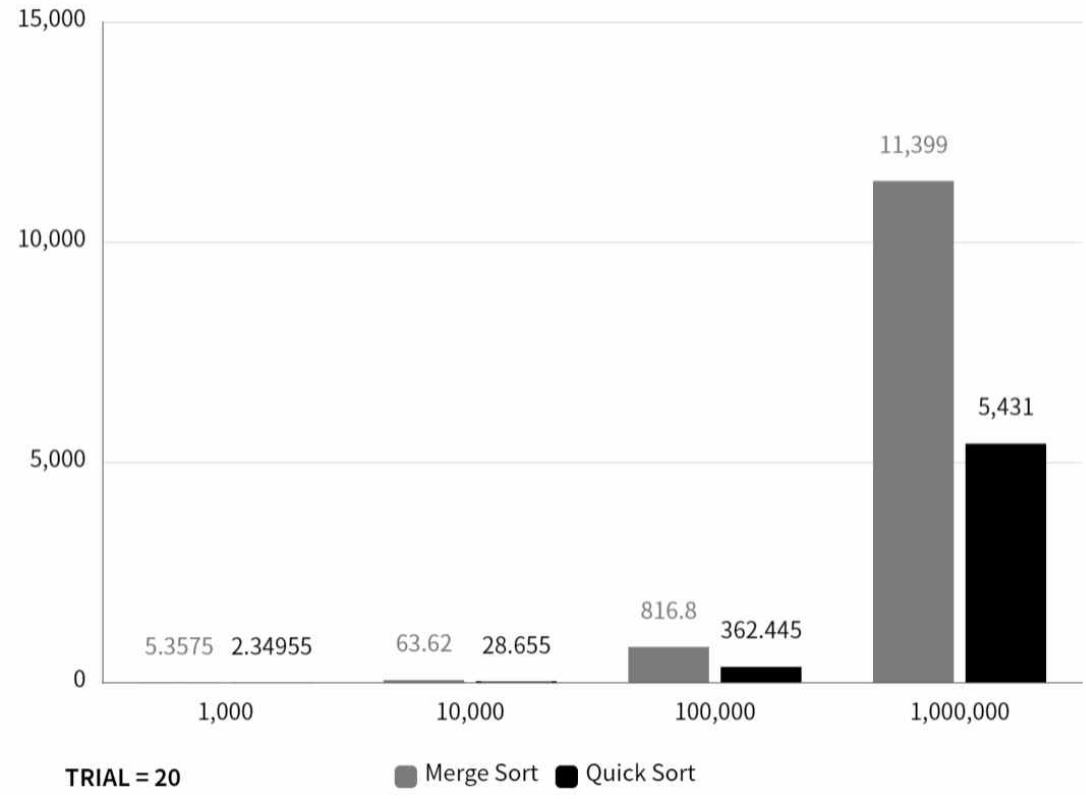


표[1-6]

TRIAL = 20



표[1-7]



표[1-8]

5. 알고리즘별 수행시간에 대한 비교 및 기술적 고찰

우선 위 8개의 수행시간에 대한 그래프를 토대로 보았을 때, TRIAL마다 Merge Sort와 Quick Sort의 차이는 크게 발생하지 않으므로 가장 수행 횟수가 높은 TRIAL인 100을 선택하여 수행시간에 대한 비교 및 기술적 고찰을 진행하였다.

먼저 **Merge Sort의 경우,** n이 1,000일 때 4.8144ms, n이 10,000일 때 62.231ms, n이 100,000일 때 797.6ms, n이 1,000,000일 때 11005ms 라는 결과가 나왔다. n이 10의 배수로 증가할 때마다 Merge Sort로 정렬한 리스트의 수행시간이 증가하는데, 실제 마스터 정리의 근사 버전에 의한 Merge Sort 복잡도 분석 결과에 따르면 Merge Sort의 시간 복잡도는 $O(nlogn)$ 이다. 이 복잡도를 고려하여 계산하면 다음과 같다.

n이 1,000일 때 $nlogn$에 대입해보면, $1000 * \log 1000 = 3000$이다.

n이 10,000일 때 $nlogn$에 대입해보면, $10000 * \log 10000 = 40000$이다.

n이 100,000일 때 $nlogn$에 대입해보면, $100000 * \log 100000 = 500000$이다.

n이 1,000.000일 때 $nlogn$에 대입해보면, $1000000 * \log 1000000 = 6000000$이다.

따라서, 40000/3000 = 13.3, 500000/40000 = 12.5, 6000000/500000 = 12이고, 해당 값들을 위에서 구한 수행시간에 하나씩 곱하여 값을 비교해보면 다음과 같은 결과가 나온다.

n이 1,000일 때의 수행시간 4.8144ms에 13.3을 곱하면 64.03152ms, 64.03152ms에 12.5를 곱하면 800.394ms, 800.394ms에 12를 곱하면 9604.728ms가 나온다.

즉, n이 1000일 때의 수행시간을 기준으로 $nlogn$에 대입한 값과 실제 함수에 의한 결괏값을 비교해보면 다음과 같다.

n의 개수	측정값	이론값
n = 1,000	4.8144ms	4.8144ms
n = 10,000	62.231ms	64.03152ms
n = 100,000	797.6ms	800.394ms
n = 1,000,000	11005ms	9604.728ms

위와 같이 이론값에 대하여 측정값이 크게 차이 나지 않는 것을 확인할 수 있다. 따라서 Merge Sort의 점근적 복잡도는 $O(nlogn)$ 라는 것을 알 수 있다.

다음으로 **Quick Sort의 경우,** n이 1,000일 때 2.216ms, n이 10,000일 때 28.31ms, n이 100,000일 때 357.27ms, n이 1,000,000일 때 5216.1ms 라는 결과가 나왔다. Quick Sort의 경우 Merge Sort와 마찬가지로 n이 10의 배수로 증가할 때마다 Quick Sort로 정렬한 리스트의 수행시간이 증가하는데, Quick Sort의 평균 점근적 복잡도 역시 Merge Sort와 마찬가지로 $O(nlogn)$ 이다. 따라서 Merge Sort와 같은 방법으로 계산하면, n이 1,000일 때의 수행시간 2.216ms에 13.3을 곱하면 29.4728ms, 29.4728ms에 12.5를 곱하면 368.41ms, 368.41ms에 12를 곱하면 4,420.92ms가 나오며, 다음과 같이 비교할 수 있다.

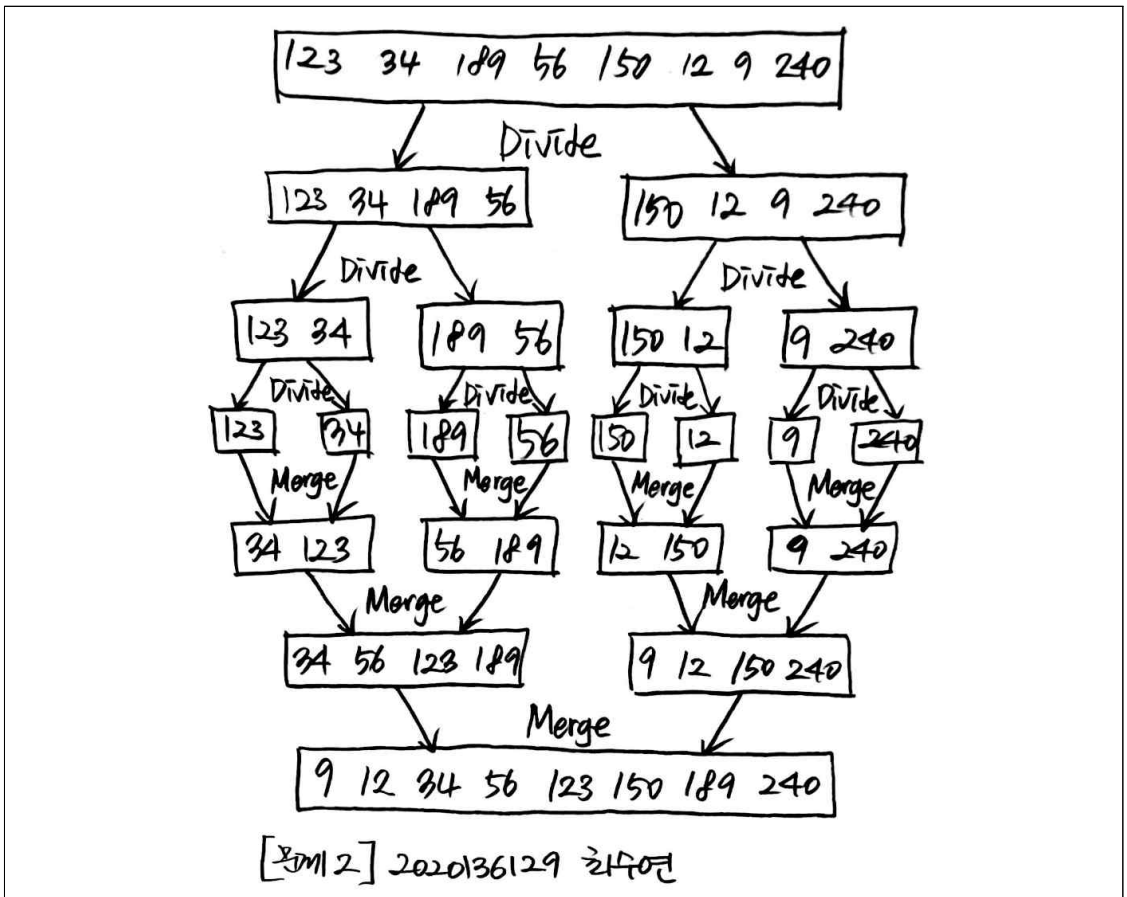
n의 개수	측정값	이론값
n = 1,000	2.216ms	2.216ms
n = 10,000	28.31ms	29.4728ms
n = 100,000	357.27ms	368.41ms
n = 1,000,000	5216.1ms	4,420.92ms

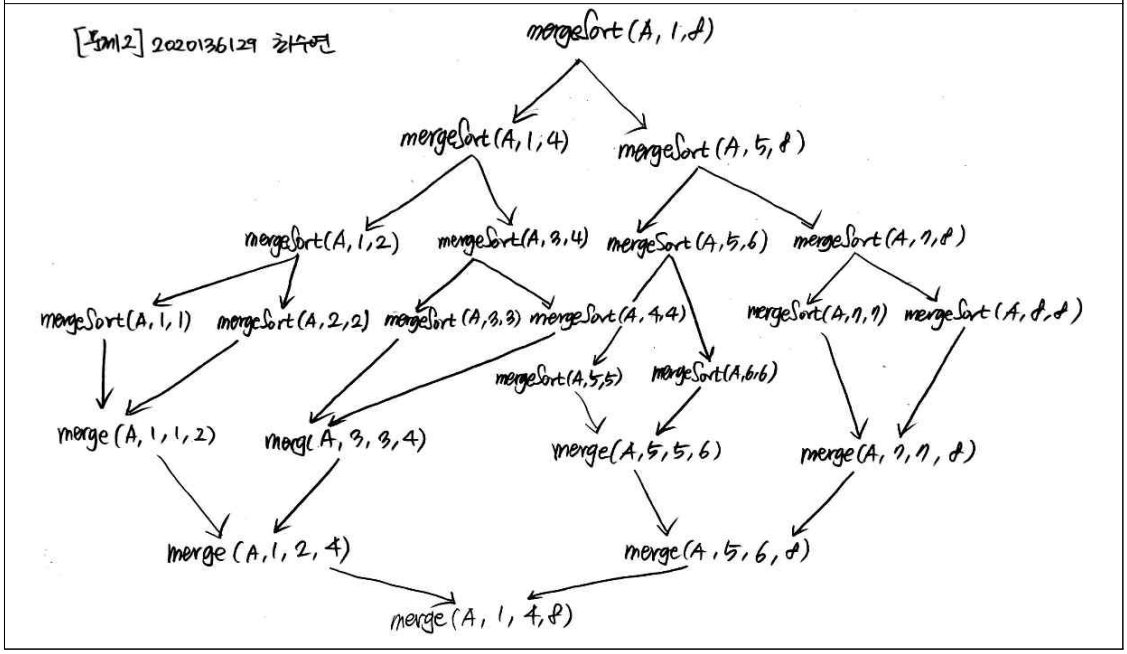
위와 같이 이론값에 대하여 측정값이 크게 차이 나지 않는 것을 확인할 수 있다. 따라서 Quick Sort의 점근적 복잡도는 $O(nlogn)$ 라는 것을 알 수 있다.

문제 2. 병합정렬

[주어진 리스트]
123, 34, 189, 56, 150, 12, 9, 240

2-1. 병합정렬 그림





2-2. 병합정렬 python 코드와 병합정렬 그림의 수행 결과 비교

python 코드 결과

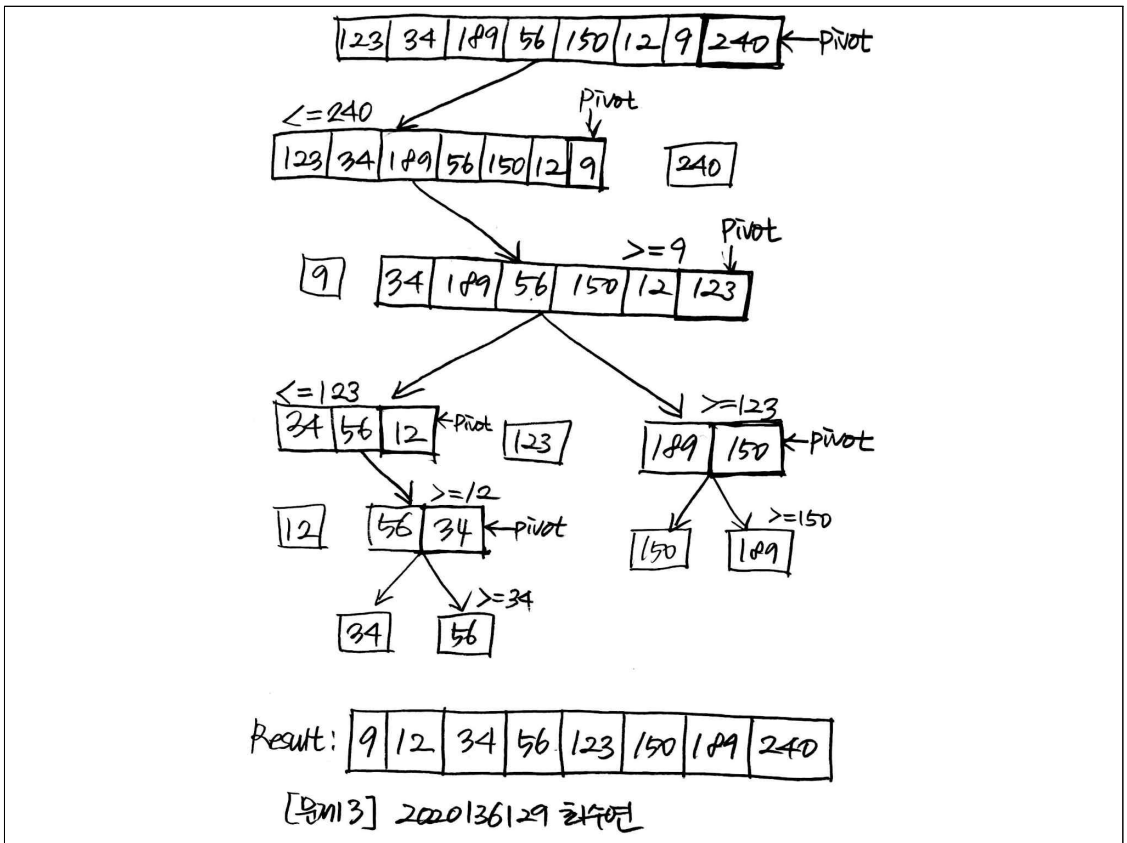
```
[Merge Sort Result]
Elapsed Time: 0.0234ms
s1: [9, 12, 34, 56, 123, 150, 189, 240]
Correct: True
```

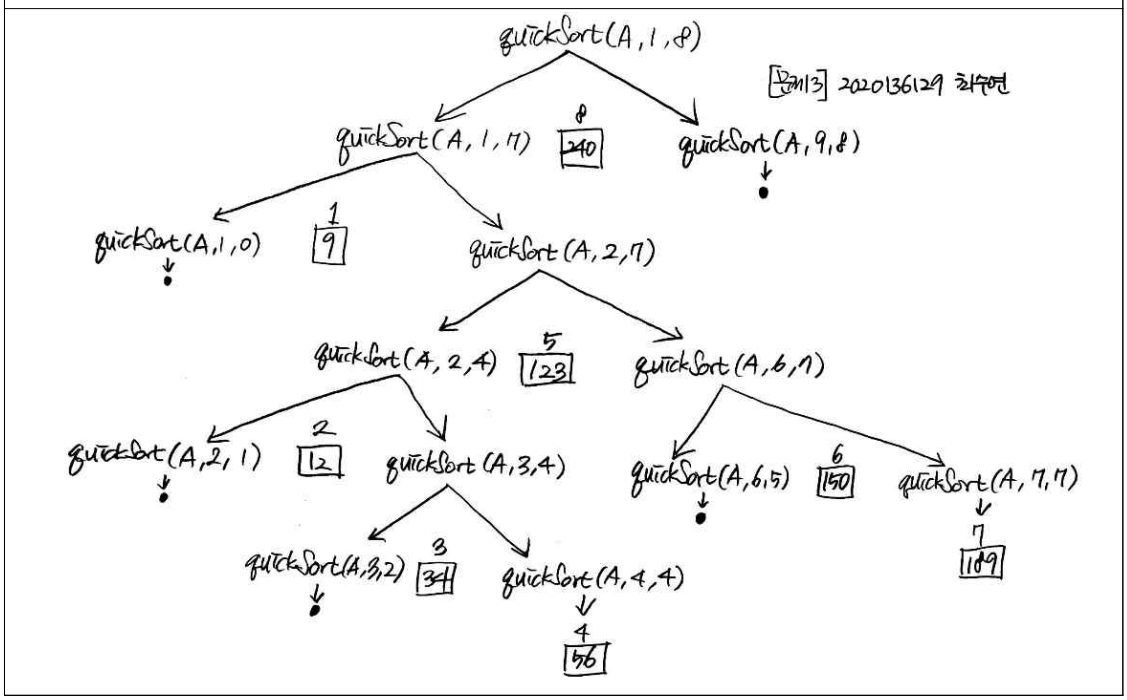
- 위 주어진 리스트에 대한 python 코드의 병합정렬 결과와 [2-1] 병합정렬 그림에서의 리스트 정렬 결과가 [9, 12, 34, 56, 123, 150, 189, 240] 으로 일치한다는 것을 알 수 있다.

문제 3. 퀵정렬

[주어진 리스트]
123, 34, 189, 56, 150, 12, 9, 240

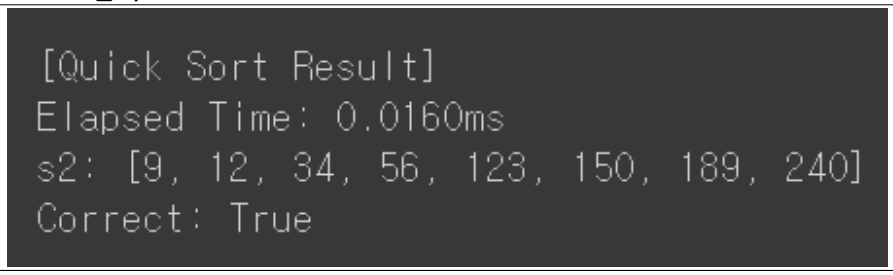
3-1. 퀵정렬 그림





3-2. 퀵정렬 python 코드와 퀵정렬 그림의 수행 결과 비교

python 코드 결과



- 위 주어진 리스트에 대한 python 코드의 퀵정렬 결과와 [3-1] 퀵정렬 그림에서의 리스트 정렬 결과가 [9, 12, 34, 56, 123, 150, 189, 240] 으로 일치한다는 것을 알 수 있다.

문제 4. 삼진 검색 알고리즘

1. 프로그램 코드

의사코드

```
index binsearch(int n, // 리스트 요소 개수
keytype[] S, // 정렬된 리스트
keytype x) // 찾고자 하는 key 값
{
  low = 0; high = n - 1; // 리스트의 양 끝 low와 high 값 지정
  location = -1; //현재 위치
  while (low <= high && location == -1) { //low가 high보다 작거나 같고 현재 위치가 -1일 때
    mid1 = ⌊low + ((high − low + 1)/3)⌋ ;
    // mid1은 전체를 세 구간으로 나누었을 때 첫 번째와 두 번째 구간 사이 & floor
    mid2 = ⌊high − ((high − low + 1)/3)⌋ ;
    // mid2는 전체를 세 구간으로 나누었을 때 두 번째와 세 번째 구간 사이 & floor
    if (x == S[mid1]) location = mid1; // key값이 mid1과 같을 때 mid1을 현위치로 설정
    else if (x == S[mid2]) location = mid2; // key값이 mid2와 같을 때 mid2를 현위치로 설정
    else if (x < S[mid1]) high = mid - 1; // key값이 mid1보다 작을 때 high를 mid1-1로 설정
    else if (x > S[mid2]) low = mid2 + 1; // key값이 mid2보다 클 때 low를 mid2+1로 설정
    else { // 위 네 가지 경우가 모두 아닐 경우
       low = mid1 + 1; // low 값을 mid1의 다음 위치 값으로 설정
       high = mid2 - 1; // high 값을 mid2의 이전 위치 값으로 설정
     }
  }
  return location; // 현재 위치 반환
}
```

python 코드

```
import random
import time
import math #floor 함수 사용을 위한 math 모듈 import

def main():
    num = 1000 #num 값을 변경하여 수행시간 비교 및 분석
    s = []

    for value in range(num):
        s.append(random.randint(0, num))
    #배열 s에 0부터 (num-1)까지의 수를 무작위로 (num-1)번 추가

    key = random.randint(0, num)
```

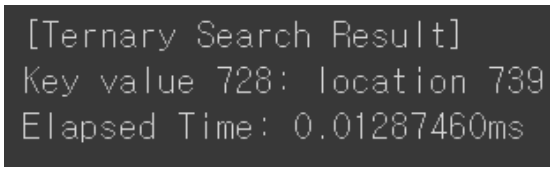
```
    #찾고자 하는 key 값을 0부터 (num-1)까지의 수 중 무작위로 선정

    s.sort() #삼진검색 알고리즘을 위한 정렬

    start = time.time()
    location = Ternary_search(s, key, 0, num - 1)
    end = time.time()
    #삼진검색 알고리즘 수행시간 측정

    #print(s)
    print("[Ternary Search Result]")
    print("Key value {0}: location {1}".format(key, location))
    print("Elapsed Time: {0:0.8f}ms".format((end - start) * 1_000))
    print()
    #삼진검색 알고리즘 key 값, 위치 및 수행시간 출력

def Ternary_search(s, key, low, high): #재귀적 삼진검색 알고리즘 함수
    mid1 = math.floor(low + ((high - low + 1) / 3))
    mid2 = math.floor(high - ((high - low + 1) / 3))

    if low <= high:
      if key == s[mid1]:
        return mid1
      elif key == s[mid2]:
        return mid2
      elif key < s[mid1]:
        return Ternary_search(s, key, low, mid1 - 1)
      elif key > s[mid2]:
        return Ternary_search(s, key, mid2 + 1, high)
      else:
        return Ternary_search(s, key, mid1 + 1, mid2 - 1)
    else: return -1

if __name__ == "__main__":
    main()
```

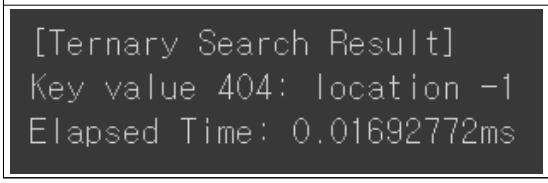
2. 프로그램 코드 설명

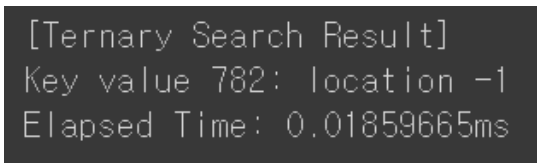
Ternary_search 함수
- 재귀적 삼진검색 알고리즘의 함수에서는 매개변수로 리스트 s, key, low, high 값을 모두 받는다.(재귀적 삼진검색 알고리즘의 수행시간 측정 부분 코드에서 매개변수가 4개로 구성되어 있음.) - 내림 함수 floor를 사용하여 mid1과 mid2를 정하는데, mid1은 정렬된 리스트 전체를 세 구간으로 나누었을 때 첫 번째와 두 번째 구간 사이에 있는 값이 mid1이 되도록 하였다. 그리고 mid2는 정렬된 리스트 전체를 세 구간으로 나누었을 때 두 번째와 세 번째 구간 사이에 있는 값이 mid2가 되도록 하였다. - 만약 low가 high보다 작거나 같을 때만 if 문이 실행되도록 코드를 작성한다. 만약 low가 high보다 클 경우, 정수 -1을 반환한다. - 위 if 문 안에 if 문을 작성하는데, 이때 내부에 있는 if 문은 key 값과 리스트 s에서 인덱스 mid1, mid2에 해당하는 값을 비교하는 방식으로 작성한다. - 내부 if 문에서 만약 key 값이 리스트 s에서 인덱스 mid1에 해당하는 값과 같을 경우, mid1을 반환한다. - 그렇지 않고 만약 key 값이 리스트 s에서 인덱스 mid2에 해당하는 값과 같을 경우, mid2을 반환한다. - 그렇지 않고 key 값이 리스트 s에서 인덱스 mid1에 해당하는 값보다 작을 경우, 해당 재귀적 이진검색 알고리즘 함수를 다시 호출하는데, 이때 high 값을 (mid1 – 1)로 바꾸어 해당 함수를 호출한다. - 그렇지 않고 key 값이 리스트 s에서 인덱스 mid2에 해당하는 값보다 클 경우, 해당 재귀적 이진검색 알고리즘 함수를 다시 호출하는데, 이때 low 값을 (mid2 + 1)로 바꾸어 해당 함수를 호출한다. - 만약 위 네 가지 조건 모두 만족하지 않을 때, 즉 key 값이 리스트 s의 인덱스 mid1에 해당하는 값보다 크고 mid2에 해당하는 값보다 작을 경우, low 값을 (mid1 + 1), high 값을 (mid2 - 1)로 바꾸어 해당 함수를 다시 호출한다.

3. 프로그램 수행화면

num = 1,000일 때

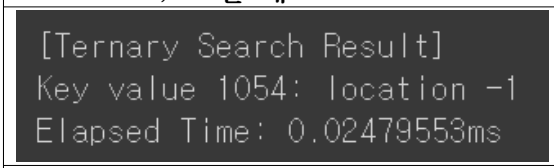


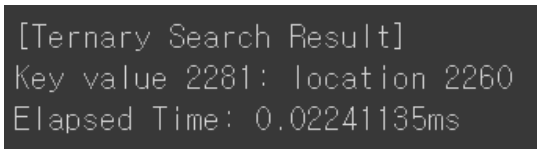




- num = 1,000일 때 수행시간 평균: 0.0163316725(ms)

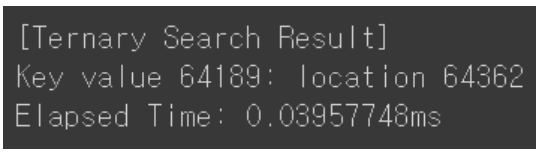
num = 10,000일 때

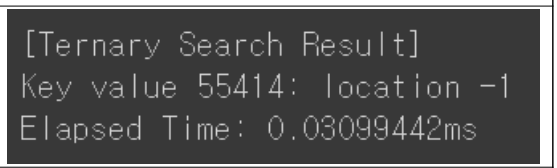


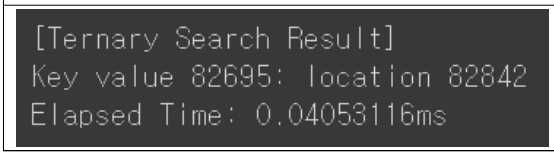


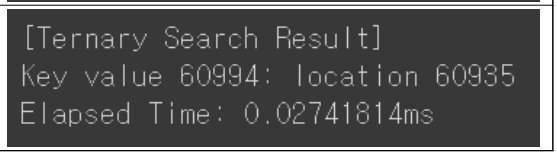
- num = 10,000일 때 수행시간 평균: 0.025868415(ms)

num = 100,000일 때



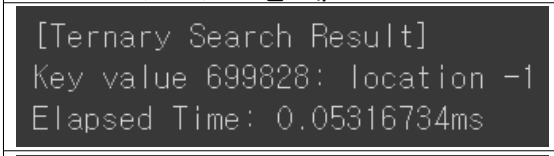


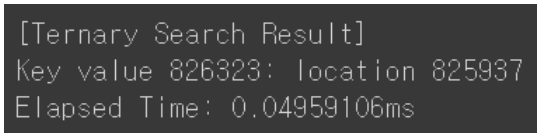


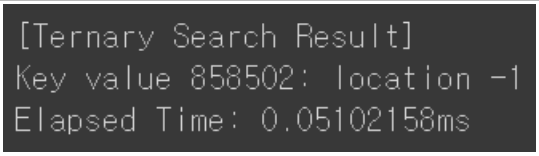


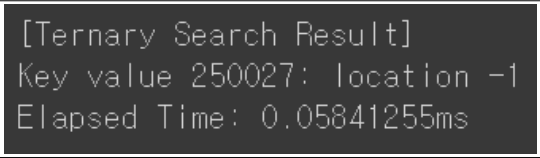
- num = 100,000일 때 수행시간 평균: 0.034571115(ms)

num = 1,000,000일 때





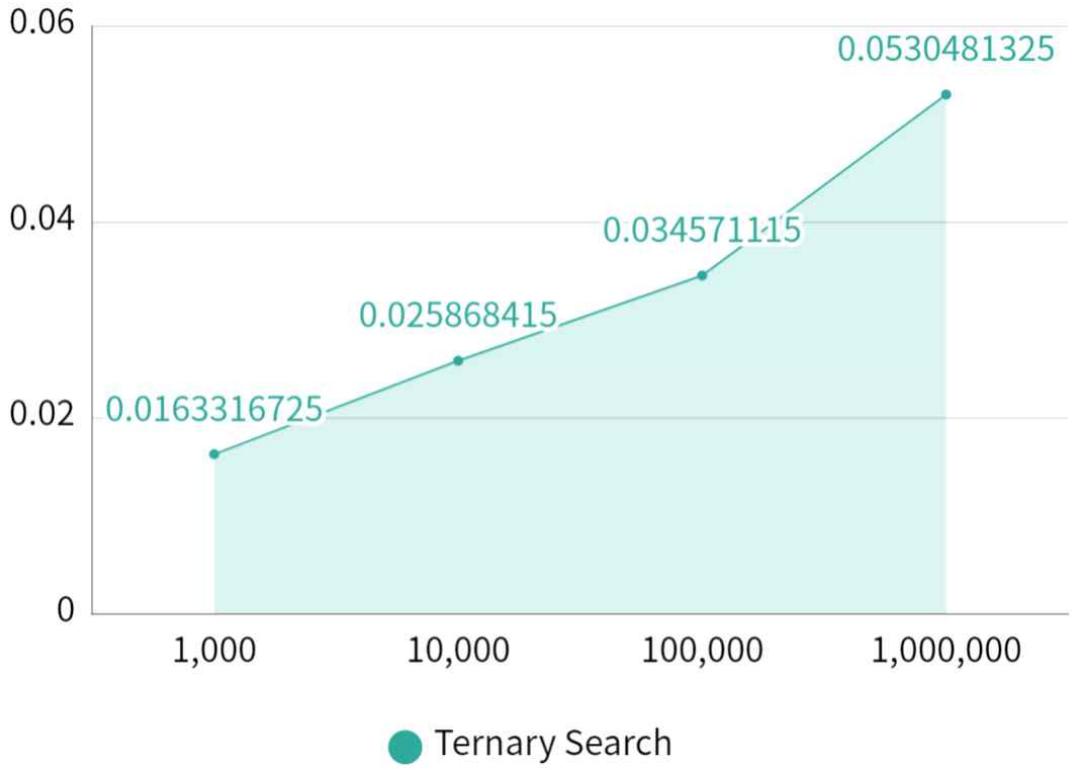




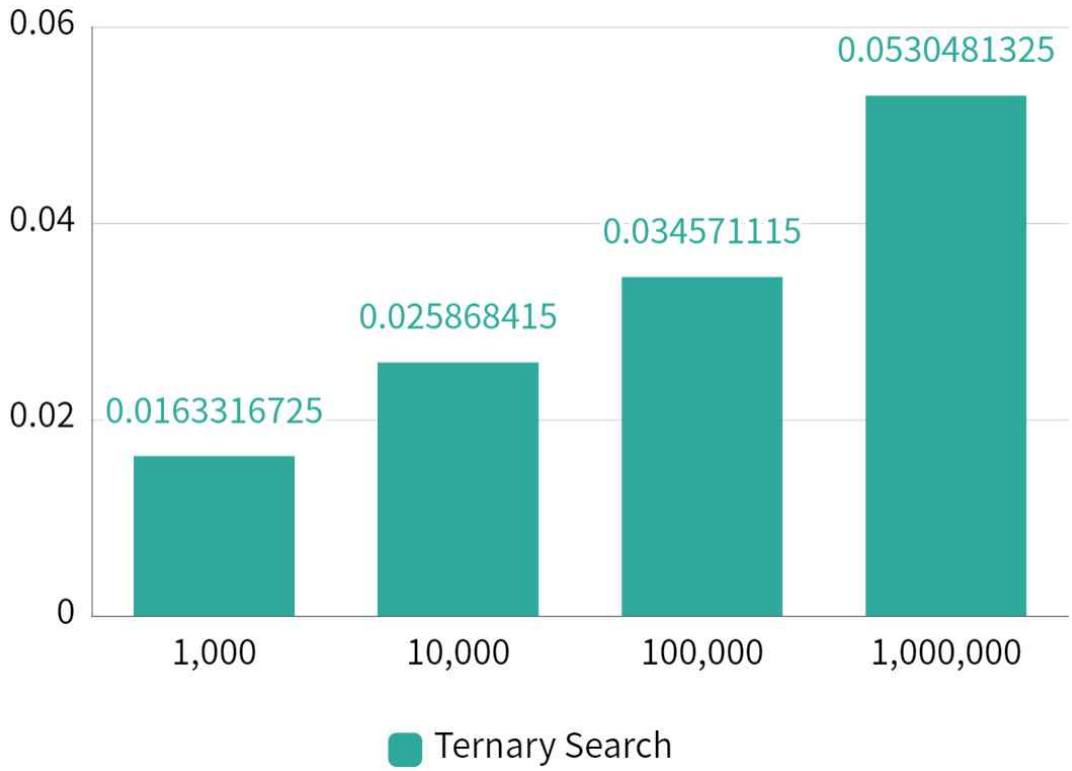
- num = 1,000,000일 때 수행시간 평균: 0.0530481325(ms)

4. 수행시간 그래프

Ternary_search 그래프



표[4-1]



표[4-2]

5. Ternary search 함수의 알고리즘 수행시간에 대한 비교 및 기술적 고찰

[기준 연산]

본 Ternary search 함수는 재귀 함수로 구현되었으므로 점근적 복잡도를 구하기 위해 기준 연산이 되는 부분은 재귀 호출할 때마다 n/3씩 분할되는 부분인 조건문 안의 다음 세 부분이라고 볼 수 있다.

Ternary_search(s, key, low, mid1 - 1)

Ternary_search(s, key, mid2 + 1, high)

Ternary_search(s, key, mid1 + 1, mid2 - 1)

[알고리즘 수행시간]

먼저 Ternary_search에 대한 수행시간 그래프 표[4-1], 표[4-2]를 보았을 때, num이 10배씩 증가할수록 수행시간이 그래프 기준, 평균 0.01223882ms, 약 0.01ms씩 증가하는 것을 알 수 있다. 그러나 num의 증가율에 비해 수행시간의 증가율은 크지 않으므로, num이 매우 커져도 num에 대한 수행시간은 한 번에 크게 증가하지 않을 것이라 예측할 수 있다.

크기가 n인 리스트가 1/3으로 나뉘어 계속 부분 리스트가 생성되므로, 이를 알고리즘 수행시간을 점화식으로 나타내면, $T(n) \leq T\left(\frac{1}{3}\right) + c$이다.

(단, c는 재귀 호출을 제외한 나머지 수행시간)

[최악의 경우 점근적 복잡도]

Ternary search 함수를 통해 반복적으로 재귀 호출될 때 해당 리스트의 크기가 계속 n/3씩 분할된다. 위 기준 연산에 따라 재귀 호출될 때마다 1/3씩 나뉘어 실행되므로 만약 k번 실행하면 $\left(\frac{1}{3}\right)^k \times n$이 되고, Ternary search 함수를 k번 수행 시 마지막에 리스트에 남는 원소는 하나이므로 $\left(\frac{1}{3}\right)^k \times n = 1$ 이라고 가정하면, $k = \log_3 n$ 이 된다. 따라서, 계속 재귀 호출되어 1/3로 분할되었을 때, 최악의 경우 본 함수의 점근적 복잡도는 $O(\log_3 n)$이다.

결론

오프라인 및 온라인 강의에서 들은 병합 정렬과 퀵 정렬에 대해 과제를 수행하고 나니 좀 더 깊이 있는 이해가 가능해졌다. 사실 퀵 정렬 같은 경우, 온라인 강의 먼저 시청했을 때 이해가 되는 듯하면서도 막상 온라인 강의를 다 듣고 직접 생각해보는 과정에서 쉽게 이해되지 않았다. 그리고 오프라인 강의 시간에도 병합 정렬과 퀵 정렬에 대해 복습하는 차원에서 다시 열심히 수업을 들었지만 제대로 이해하지 못했다. 그러나 이번 과제를 하면서 직접 손으로 풀어보고 그려보는 과정에서 좀 더 쉽게 이해할 수 있었던 것 같아 공부가 많이 되었다.

또한 지난 시간 배운 이진검색 알고리즘을 응용한 삼진검색 알고리즘을 재귀로 구현하는 과정에서 생각보다 재밌다고 느꼈고, 이런 방식으로 코드를 구현하면 수행시간을 크게 늘리지 않고도 삼진이 아닌 n진 검색 알고리즘도 충분히 구현이 가능할 것이라 생각했다.

저번과 비슷하게 과제 보고서 작성하는 시간은 많이 걸렸지만 좀 더 익숙해지면 효율적으로 빠르게 과제를 수행할 수 있을 것이라고 생각한다.