

---

# 알고리즘및실습

[실습 및 과제 2]

---



과목명/분반	알고리즘및실습/02	담당교수	한연희
학 부 명	컴퓨터공학부	제 출 일	2022/ 04 / 05
학번	2018136121	이름	조원석

# INDEX

표지 및 차례-----	1
--------------	---

## 서론

Homework의 내용 및 목적 -----	3
-------------------------	---

## 본론

[문제1] 합병 정렬과 퀵 정렬에 대한 Python 프로그램 작성하기-----	4
---	---

- > merge\_and\_quick\_sort.py 분석하기
- > 완성되지 않은 다음 두 함수와 각각의 수행시간 그래프 작성하기
  - def merge(s, low, mid, high):
  - def partition(s, low, high):

[문제2] 병합 정렬 과정을 그림으로 나타내기 -----	6
---------------------------------	---

- > [123, 34, 189, 56, 150, 12, 9, 240]을 정렬하는 과정을
  - 병합 정렬을 손으로 그려서 구현하기
  - 병합 정렬 리스트의 정렬된 결과와 비교하기

[문제3] 퀵 정렬 과정을 그림으로 나타내기 -----	7
--------------------------------	---

- > [123, 34, 189, 56, 150, 12, 9, 240]을 정렬하는 과정을
  - 퀵 정렬을 손으로 그려서 구현하기
  - 퀵 정렬 리스트의 정렬된 결과와 비교하기

[문제4] 삼진 검색 알고리즘 제시하기-----	8
----------------------------	---

- > 의사코드로 작성하기
- > python 코드로 작성하기 : def ternary(s, key, low, high):  
condition.     1) 기준연산  
                  2) 알고리즘 수행시간  
                  3) 최악의 경우 점근적 복잡도

## 결론

고찰-----	15
---------	----

## 서론 : Homework의 내용 및 목적

### ▶ 내용

: merge\_and\_quick\_sort.py 코드상에 정의된 merge와 partition 함수를 구현하고 각각의 알고리즘에서 분할정복이 일어나는 과정을 관찰하고 각 과정을 직접 손으로 그려보면서 병합 정렬과 퀵 정렬의 동작 원리 이해하기

: 삼진 검색 알고리즘을 각각 의사코드와 파이썬 코드로 구현함으로써 기준연산, 알고리즘 수행시간, 최악의 경우 점근적 복잡도를 확인해보는 시간 갖기

### ▶ 목적

: 정렬 알고리즘을 통한 분할정복 적용  
: 분할정복을 통한 삼진 검색 구현  
: 분할정복 알고리즘의 이해  
: 알고리즘의 기준연산, 수행시간, 최악의 경우 점근적 복잡도 표현 방법 탐구

## 본론 : 문제 풀이

[문제1] 합병 정렬과 퀵 정렬에 대한 Python 프로그램 작성하기

### - merge\_and\_quick\_sort.py 분석하기

#### ▶ 내용

: merge\_and\_quick\_sort.py는 수행횟수가 num이고 그 범위가 각각 [0, num]인 랜덤 리스트 s와 그를 s1, s2에 복제한 상태에서 시작합니다.

s는 정렬되어 대조군처럼 사용되고 s1은 합병정렬의 대상, s2는 퀵정렬의 대상이 되어 연산을 진행합니다.

이중에서 정렬을 실질적으로 구현하는 부분인 merge 함수와 partition 함수를 구현해보려고 합니다.

### - 완성되지 않은 두 함수 완성하기

▶ *def merge(s, low, mid, high):*

*# 병합 정렬*

▶ *def partition(s, low, high):*

*# 퀵 정렬*

▶ *def merge\_sort(s, low, high):*

```
def merge_sort(s, low, high):          # 분할과 정복으로 구현된 병합 정렬 함수
    if low < high:
        mid = (low + high) // 2
        # low와 high의 중간 지점 계산 ==> 내림 연산 : ⌊ (low + high) / 2 ⌋
        merge_sort(s, low, mid)        # 분할① : 전반부(low ~ mid까지) 정렬
        merge_sort(s, mid + 1, high)    # 분할② : 후반부(mid + 1 ~ high까지) 정렬
        merge(s, low, mid, high)        # 정복(병합_Merge) : 분할①과 분할②를 합친다.
```

## ▶ def merge(s, low, mid, high):

```
tmp = [0] * (high - low + 1)    #high ~ low의 크기를 갖는 임시 리스트 tmp
# i는 0 ~ mid, j는 mid ~ high를 이동하고
# s[i]와 s[j]가 릴레이 가위바위보를 하듯이 비교를 진행하며
#① 둘 중 작은 값이 tmp[t]에 저장되고 인덱스가 이동된다
i = low
j = mid + 1
t = 0
while (i <= mid) & (j <= high):
    if s[i] <= s[j]:
        tmp[t] = s[i]
        t += 1
        i += 1
    else:
        tmp[t] = s[j]
        t += 1
        j += 1
#② 비교 후에 남은 것은 tmp에 남은 부분에 순차적으로 저장된다
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
#③ 정렬이 끝난 tmp를 s에 순차적으로 넣으며 마무리한다
while i <= high:
    s[i] = tmp[t]
    i += 1
    t += 1
```

def merge(s, low, mid, high):

즉, merge 함수에서는 [low, mid]의 구간을 담은 i팀과 [mid+1,high]의 구간을 담은 j팀이 **릴레이 가위바위보**를 하듯이 ①s[i]와 s[j]가 비교되고, 둘 중 진 부분, 즉 더 작은 부분이 tmp[t]에 들어가고 진 팀이 속한 인덱스와 t의 인덱스가 1씩 증가하면서 차례가 넘어갑니다. 비교를 계속 진행하다가 결국 ②두 팀중 하나에 아무도 남지 않게 되었을 때 이긴 팀원이 더 많은, 즉 더 큰 값이 많이 남아있던 부분이 나머지 tmp 배열을 채우고, ③ 정렬이 끝나면 tmp값을 순차적으로 s에 대입하며 마무리됩니다.

병합정렬의 시간복잡도는  $O(N\log_2 N)$ 인데 merge의 반복문의 tmp,와 s의 대입연산과 merge\_sort의 분할정복 부분으로 결정됩니다. tmp와 s의 대입 연산이 ' $N = (\text{high} - \text{low} + 1)$ '번 일어나  $O(N)$ 을, 분할정복 부분은 s가 merge\_sort를 통해 low와 high로 나뉘어 s가 0.5배씩 줄어드는, 즉  $\log_2 N$ 씩 쪼개지는 과정으로  $O(\log N)$ 의 시간복잡도를 가지며 둘이 중첩된  $O(N\log N)$ 이 도출됩니다.

## ▶ def quick\_sort(s, low, high):

def quick_sort(s, low, high):	# 분할과 정복을 이용한 퀵 정렬 함수
if low < high:	
pivot = partition(s, low, high)	# 분할 : partition의 결과값을 pivot에 대입
quick_sort(s, low, pivot - 1)	# 좌측(low ~ pivot - 1) 리스트 정렬
quick_sort(s, pivot + 1, high)	# 우측(pivot + 1 ~ high) 리스트 정렬

## ▶ def partition(s, low, high):

# partition은 일정한 기록을 갖는 사람들의 **릴레이 뒀을** 과정이라 생각할 수 있다.

# ①뒀들의 기준을 줄에서 가장 뒤에 서 있는 사람의 기록(pivot)이라 가정하고

# ②피벗을 제외하고 뒀들을 넘은 사람과 뒀들을 넘지 못한 사람으로 나누어

# ③넘은 사람중 첫 사람과 넘지 못한 사람의 자리를 바꿔주는 방식으로

# ④줄을 세운 뒤 pivot과 넘은 사람중 첫 사람 자리를 바꿔주고 ①의 과정을 반복한다.

# 조건 : 사람들의 기록은 항상 일정하고 서로 다른 수치를 갖는다.

# 위의 과정을 계속하면 결국 분할되면서 함께 자리도 정렬되는 효과를 얻을 수 있다.

x = s[high]	# 기준 원소 x (pivot) : 맨 끝 원소
i = low - 1	# i는 가장 처음(왼쪽) 지점
for j in range(low, high):	# j는 오른쪽으로 이동하며 x와 s[j] 비교
if s[j] <= x:	# 뒀들을 못 넘었다면
i += 1	# j라는 사람은 i + 1에 자리한다.
s[i], s[j] = s[j], s[i]	
i += 1	# 마지막에 끝자리의 피벗을
j += 1	# 뒀들을 못 넘은 사람 뒤에 넣고
s[i], s[j] = s[j], s[i]	# 자리를 바꿔준다.
return i	# 그리고 바뀐 피벗의 인덱스를 반환한다.

def partition(s, low, high):

즉, partition 함수는 i와 j라는 두 포인터(점)를 이용해서 s의 구간을 오른쪽 맨 끝 원소인 피벗 기준으로 둘로 나눕니다. 그 결과 원소는 [low, i-1] 인덱스에 들어가는 피벗 이하의 원소와 [i+1, j(=high)]에 들어가는 피벗 이상의 원소로 나뉘어 피벗보다 큰 원소중 첫 원소와 피벗보다 작은 s[j]와 자리를 바꿔주는 방식으로 정렬됩니다. 그리고 피벗은 j(high)에서 i로 s[i]에 있던 값은 high로 이동하면서 배치가 끝나게 됩니다.

퀵정렬의 시간복잡도는 **최악의 경우  $O(N^2)$ , 평균의 경우  $\Theta(N \log N)$** 을 따른다고 합니다.

partition 함수의 핵심연산인 s[j]와 x의 비교연산을 (high - low)번 수행하며 점근적으로  $\Theta(N)$ 에 가까워지고 quick\_sort함수에서 분할 이후 한쪽에 원소가 하나도 없고 다른 한 쪽에 물리는 경우 점화식이  $T(N) = T(N-1) + N - 1$ 이 되어  $T(N) = \Theta(N^2)$ 이 되고, 평균적으로는

점화식이  $T(N) = \frac{1}{N} \sum_{i=1}^N T(i-1) + T(N-i) + N - 1$ 이 되어

$T(N) = \Theta(N \log N)$ 을 따른다고 합니다.

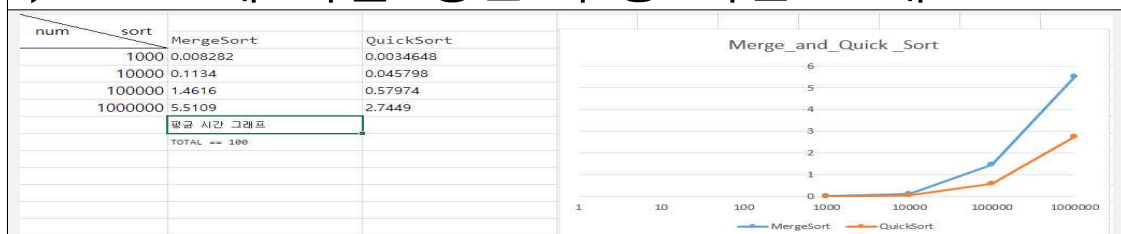
## ▶ 실행 화면

<b>num = 1,000일 때</b>	<b>num = 10,000일 때</b>
<b>num = 100,000일 때</b>	<b>num = 1,000,000일 때</b>

## ▶ 실행 결과

<b>100번 TOTAL 실행 시간</b> Merge Sort : 0.008282s Quick Sort : 0.0034648s <b>100번 평균 실행 시간</b> Merge Sort : 0.008282s Quick Sort : 0.0034648s	<b>100번 TOTAL 실행 시간</b> Merge Sort : 11.34s Quick Sort : 4.5798s <b>100번 평균 실행 시간</b> Merge Sort : 0.1134s Quick Sort : 0.045798s
<b>num = 1,000일 때</b>	<b>num = 10,000일 때</b>
<b>100번 TOTAL 실행 시간</b> Merge Sort : 146.16s Quick Sort : 57.974s <b>100번 평균 실행 시간</b> Merge Sort : 1.4616s Quick Sort : 0.57974s	<b>100번 TOTAL 실행 시간</b> Merge Sort : 551.09s Quick Sort : 274.49s <b>100번 평균 실행 시간</b> Merge Sort : 5.5109s Quick Sort : 2.7449s
<b>num = 100,000일 때</b>	<b>num = 1,000,000일 때</b>

## ▶ num에 따른 평균 수행 시간 그래프

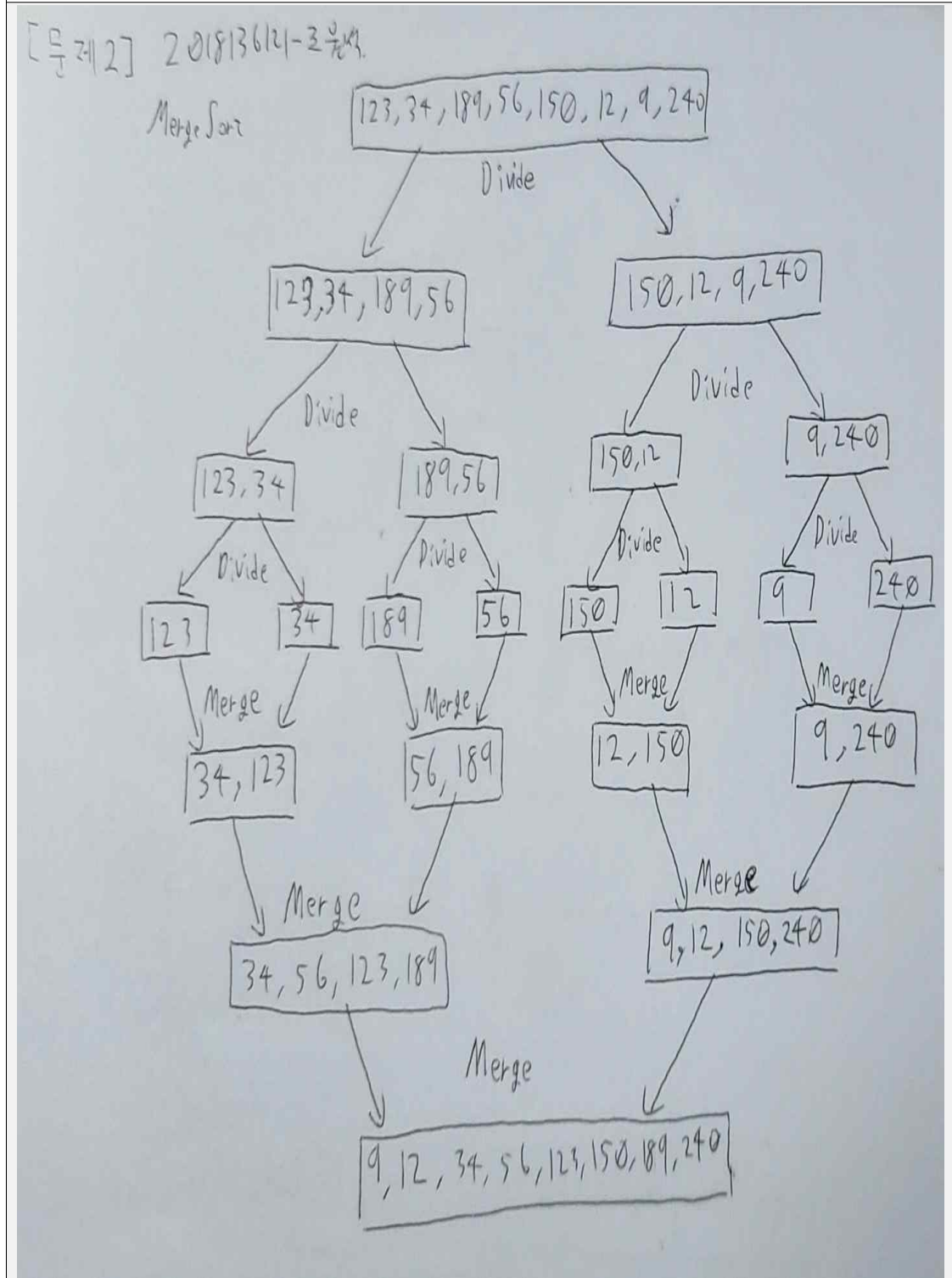


[문제1] 결론

빅-O 표기법에 따랐을 때 MergeSort가 QuickSort보다 더 효율적인 결과를 나타냈지만 일반적인 상황에서 MergeSort가 QuickSort보다 느리다는 것을 알 수 있었습니다.

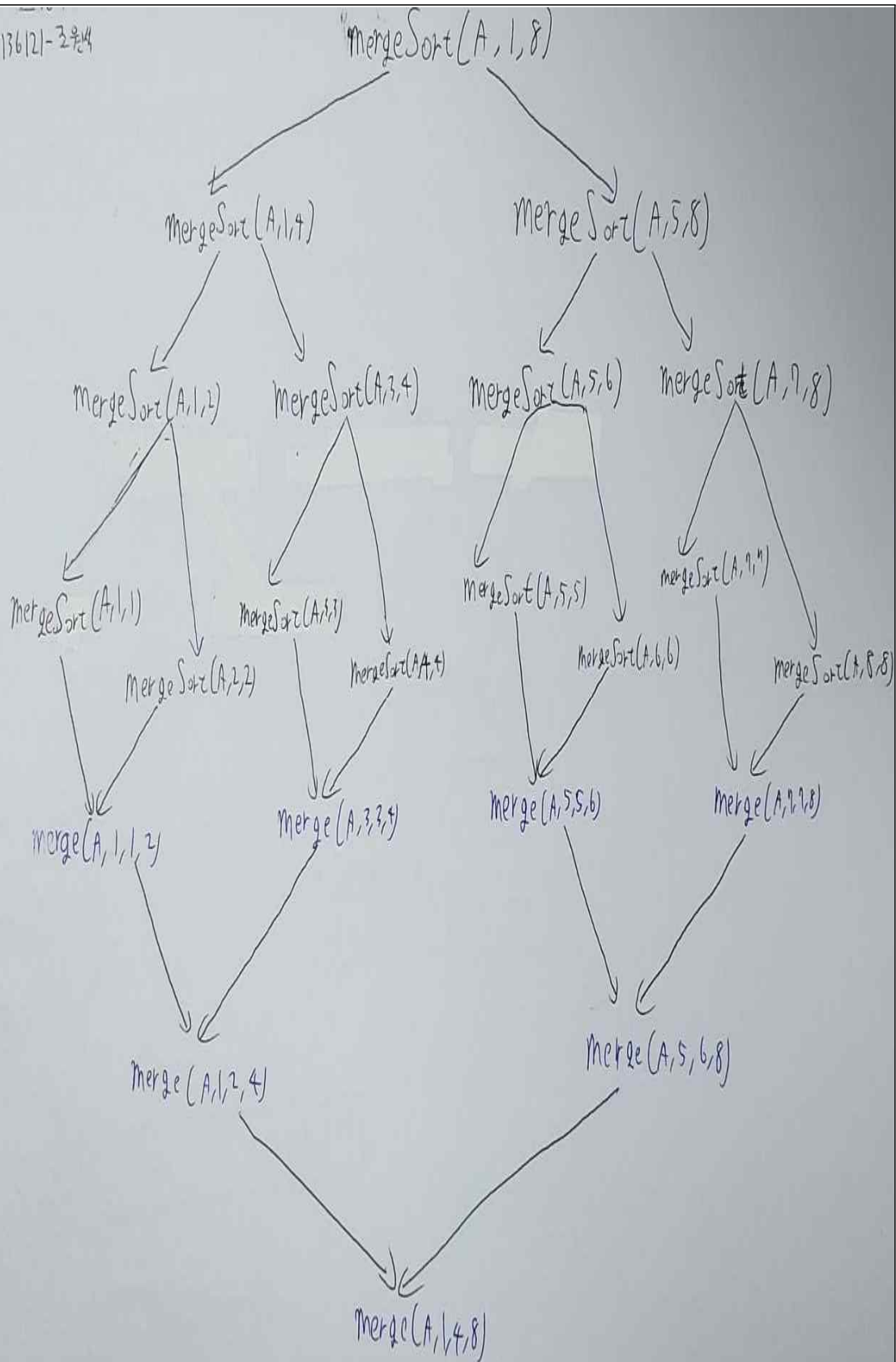
[문제2] 병합 정렬 과정을 그림으로 나타내기

[문제2-1] 병합 정렬을 손으로 그려서  
구현하기





136121-2914



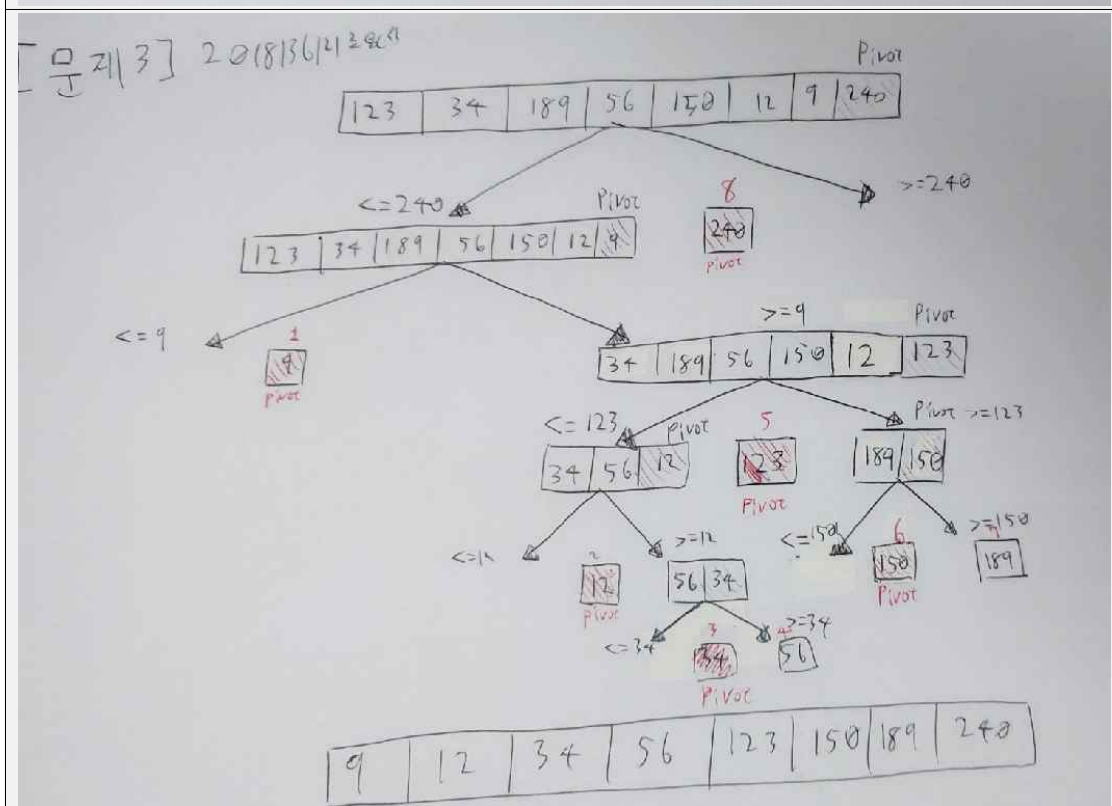
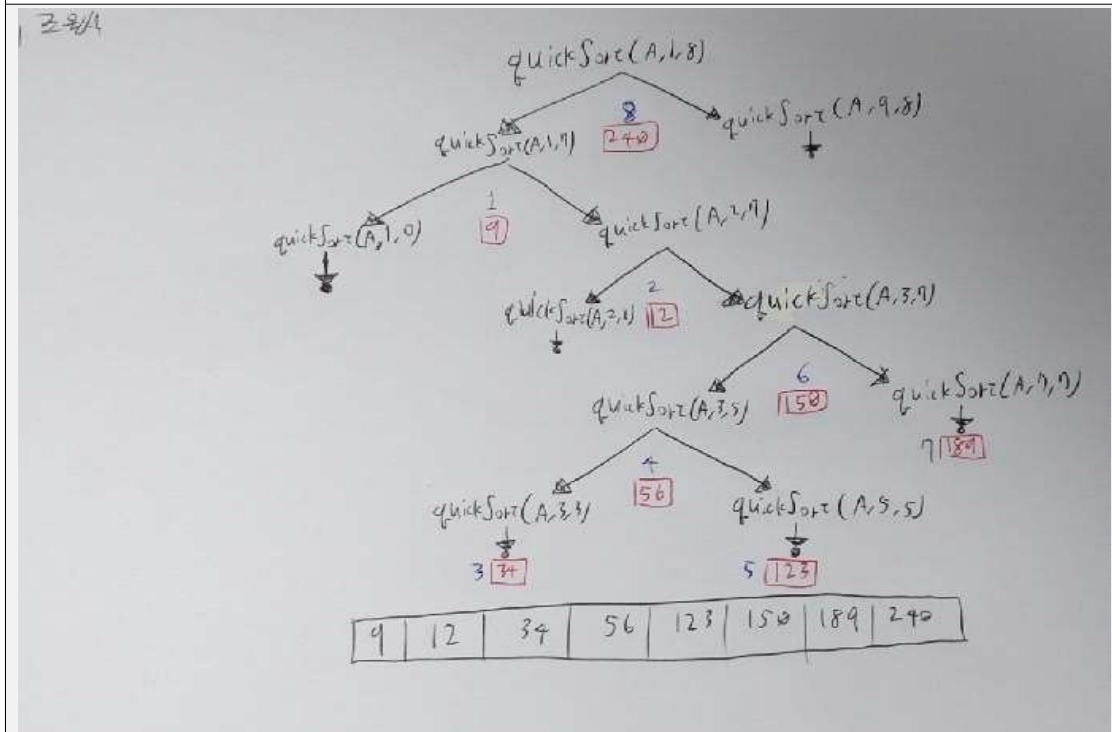
## [문제2-2] 병합 정렬 리스트의 정렬된 결과와 비교하기

```
[Merge Sort Result]
Elapsed Time: 0.0439ms
s1: [9, 12, 34, 56, 123, 150, 189, 240]
Correct: True
```

정렬 결과 s1 : [9,12,34,56,123,150,189,240]으로 손으로 그려서 도출한 MergeSort의 결과값과 동일하다는 것을 알 수 있었습니다.

[문제3] 퀵 정렬 과정을 그림으로 나타내기

[문제3-1] 퀵 정렬을 손으로 그려서 구현하기



### [문제3-2] 퀵 정렬 리스트의 정렬된 결과와 비교하기

```
[Quick Sort Result]
Elapsed Time: 0.0251ms
s2: [9, 12, 34, 56, 123, 150, 189, 240]
Correct: True
```

정렬 결과 s2 : [9,12,34,56,123,150,189,240]으로 손으로 그려서 도출한 QuickSort의 결과값과 동일하다는 것을 알 수 있었습니다.

#### [문제4] 삼진 검색 알고리즘 제시하기

##### ▶ 의사코드로 나타내기

```
ternary_search(s[0 ~ num-1], key, low, high) {  
    border1 ← low + ⌊ (high - low) ÷ 3 ⌋  
    border2 ← high - ⌊ (high - low) ÷ 3 ⌋  
    if (s[border1] == key)                #①  
        return border1  
    elif (s[border2] == key)              #②  
        return border2  
    elif (key < s[border1])                #③  
        return ternary_search(s, key, low, border1 - 1)  
    elif (key < s[border2])                #④  
        return ternary_search(s, key, border1 + 1, border2 - 1)  
    elif                                  #⑤  
        return ternary_search(s, key, border2 + 1, high)  
    else                                  #⑥  
        return -1  
}
```

n개의 아이템을 가진 정렬된 리스트를 거의 n/3개 아이템을 가진 3개의 부분리스트로 분할하여 검색하는 알고리즘

즉 삼진 검색 알고리즘을 구현하기 위해서는 3개의 부분으로 나누기 위한 경계선(border)이 두 개가 필요합니다.

그리고 이 경계선을 통해 key의 위치가 두 경계선 안에 위치한 경우( ①, ②), 작은 경계선보다 작은 곳에 위치한 경우 ( ③ ), 작은 경계선과 큰 경계선 사이에 위치한 경우( ④ ), 큰 경계선보다 큰 곳에 위치한 경우 ( ⑤ ), 값이 존재하지 않는 경우 ( ⑥ )와 같이 총 6개의 케이스로 분류할 수 있습니다.

(위는 리스트에 중복된 값이 나오지 않는다고 가정했을 때의 의사코드입니다.)

## ▶ python 코드로 나타내기

# 1. 삼진검색 함수 본문

```
def ternary_search(s, key, low, high):
```

```
    # 조건문 이전의 코드 => 수행시간 : 3C
```

```
    # 초기화문 : C
```

```
    num = len(s)
```

```
    # num == s의 길이
```

```
    mid1 = low + (high - low) // 3
```

```
    # mid1 == 부분리스트 1, 2의 경계
```

```
    mid2 = high - (high - low) // 3
```

```
    # mid2 == 부분리스트 2, 3의 경계
```

```
    #조건문 이후의 코드
```

```
    # ①, ②, ⑥인 경우 수행시간 : C
```

```
    # 조건문 수행시간 : C
```

```
    # ③ ~ ⑤이 최대로 나오는 경우 수행시간 :  $T(n/3) + C$  # 재귀호출 수행 :  $T(N/3)$ 
```

```
    if s[mid1] == key:
```

```
        #①mid1이 키일 때
```

```
        return mid1
```

```
    elif s[mid2] == key:
```

```
        #②mid2이 키일 때
```

```
        return mid2
```

```
    elif key < s[mid1]:
```

```
        #③(low, mid1)일 때
```

```
        return ternary_search(s, key, low, mid1) #  $T(N/3)$ 
```

```
    elif key < s[mid2]:
```

```
        #④(mid1, mid2)일 때
```

```
        return ternary_search(s, key, mid1+1, mid2) #  $T(N/3)$ 
```

```
    elif key < high:
```

```
        #⑤(mid2, high)일 때
```

```
        return ternary_search(s, key, mid2+1, high) #  $T(N/3)$ 
```

```
    else:
```

```
        #⑥값이 존재하지 않을 때
```

```
        return -1
```

## ▶ 기준연산 제시하기

위 코드에서의 기준연산은 굵은 글씨로 표시된 재귀함수 호출문들입니다.

①, ②, ⑥에서 바로 끝나는 경우 상수 시간( $T(N) = 4C$ )에 해결되는데 **num**이 0이 될 때까지 ③, ④, ⑤을 거치는 경우 굵은 글씨로 표기된 호출문이 기준연산으로 실행되고 이 연산들을 기준으로 수행시간이 정해지게 됩니다.

## ▶ 알고리즘 수행 시간 제시하기

즉, `ternary_search`의 최악의 수행시간을 점화식으로 나타내면  $T(N) = T(N/3) + 4C$ 으로 정리됩니다.

## ▶ 최악의 경우 점근적 복잡도 제시하기

이는  $T(N) = aT(N/b) + f(N)$  ( $a \geq 1, b \geq 1$ )의 형태로 표현가능하고,

$a = 1, b = 3, f(N) = 4C, h(N) = N^{\log_b a}$  이므로 [마스터 정리의 근사 버전]에 의해

$h(N) = 1, \frac{f(N)}{h(N)} = \Theta(1)$ 이므로  $T(N) = T(N/3) + 4C = \Theta(4C \log_3 N)$ 으로 정리됩니다.

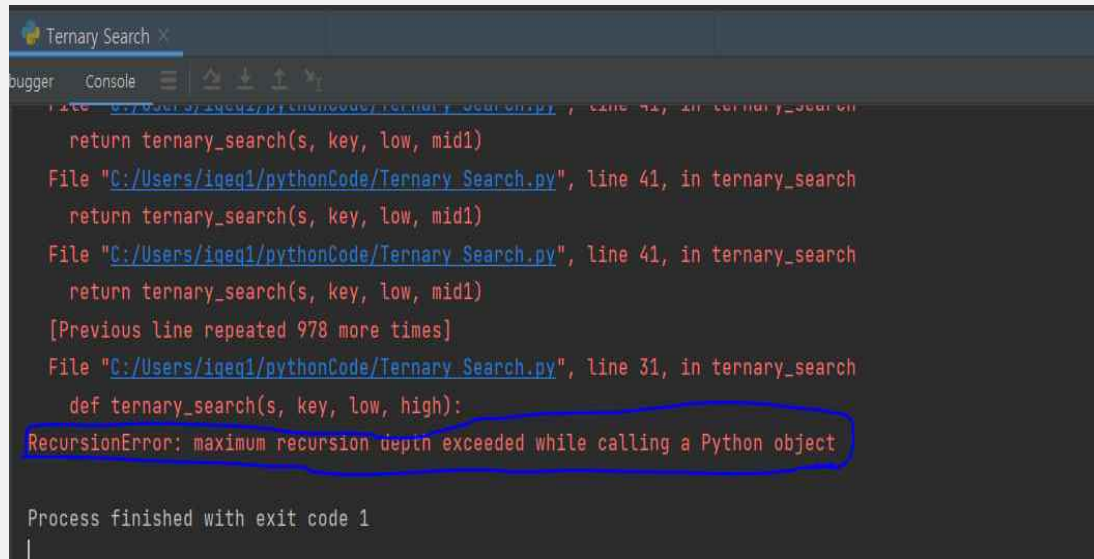
따라서 최악의 경우 점근적 복잡도는  $T(N) = O(\log_3 N)$ 으로 정리 가능합니다.

## 결론 : 고찰

### ▶ 고찰

#### ▶ 어려웠던 점

: 이번 과제를 진행하면서 처음 어려웠던 점은 **재귀로 인한 런타임 에러**였습니다.



```
Ternary Search x
bugger Console
File "C:/Users/igee1/pythonCode/Ternary_Search.py", line 41, in ternary_search
    return ternary_search(s, key, low, mid1)
File "C:/Users/igee1/pythonCode/Ternary_Search.py", line 41, in ternary_search
    return ternary_search(s, key, low, mid1)
File "C:/Users/igee1/pythonCode/Ternary_Search.py", line 41, in ternary_search
    return ternary_search(s, key, low, mid1)
[Previous line repeated 978 more times]
File "C:/Users/igee1/pythonCode/Ternary_Search.py", line 31, in ternary_search
    def ternary_search(s, key, low, high):
RecursionError: maximum recursion depth exceeded while calling a Python object

Process finished with exit code 1
```

재귀 함수의 블록 내부는 메모리 영역 중 스택 영역에 할당되는데 여기서 재귀 함수의 깊이가 깊어질수록 스택에 쌓이는 재귀 함수의 호출이 증가하여 버퍼 오버플로우의 일종인 스택 오버플로우를 야기합니다.

처음에 num을 1,000정도로 설정하였을 때 이는 큰 문제를 주지 않았지만 1,000,000 정도로 설정하다보니 재귀의 깊이가 파이참에서 기본으로 제공하는 재귀의 깊이보다 더 깊어져서 코드상의 에러를 발생시켰습니다.

```
import sys # 재귀의 최대 깊이를 변경하기 위한 라이브러리
sys.setrecursionlimit(10**8) # 최대 깊이를 998에서 100000000으로 변경
```

그래서 오류를 해결하기 위해 위의 문장을 추가해주며 재귀의 최대 깊이를 늘려주었고 결론적으로 다음과 같은 정상적인 출력을 내보내며 문제를 해결할 수 있었습니다.

```
[Ternary Search Result]
Key value 7262641: location -1
Elapsed Time: 0.06210000ms
```

: 이번 과제를 진행하면서 다른 어려웠던 점은 바로 **퀵정렬을 수기로 표현하는 방식**이었습니다. 퀵정렬을 그림으로 그리기 전에는 의사코드를 이해없이 눈으로 베끼면서 “그냥 인덱스 순서대로 큰 건 피벗 뒤에, 작은 건 피벗 앞에 끼워져서 자리들이 밀리는구나” 하고 대충 생각했었는데

막상 그림으로 그려보니 강의 자료(교재)에 있던 이러한 주석문들이 눈에 들어왔습니다.

```
partition(A[], p, r)
{
    x ← A[r];           ▷ 기준원소
    i ← p-1;            ▷ i는 1구역의 끝지점
    for j ← p to r-1    ▷ j는 3구역의 시작 지점
        if (A[j] ≤ x) then A[++i] ↔ A[j];
                                ▷ 의미는 i값 증가 후 A[i] ↔ A[j] 교환
    A[i+1] ↔ A[r];      ▷ 기준원소와 2구역 첫 원소 교환
    return i+1;
}
```

1구역(□): 15보다 작거나 같은 원소들  
 2구역(■): 15보다 큰 원소들  
 3구역(□): 아직 정해지지 않은 원소들  
 4구역(●): 15 자신

이를 보고 느낀 것은 “배열로 구현하게 되었을 때 두 원소의 위치를 바꾸어주지 않고 인덱스 순서대로 끼워서 밀게 되면 복잡도가 피벗의 교환횟수만큼 n번씩 늘어나게 되겠구나” 였습니다.

즉, 그러한 이유에서 복잡도를 줄일 수 있는 방식으로 ‘원소 간의 교환’을 선택했고 알고리즘을 학습할 때 ‘생각하는 것’이 필요하다는 것을 느끼게 되었습니다.



▶ 배운 점

: 이번 과제를 통해서 ①고급 정렬 중 병합정렬과 퀵정렬을 구현하는 방법과 ②각각의 정렬을 수기로 구현하는 방법, ③삼진 탐색을 하는 방법, ④점근적 복잡도를 구하는 마스터 정리를 삼진 탐색에 적용하는 방법, ⑤재귀로 인한 런타임 에러가 발생하였을 때 해결하는 방법 등에 대해 알아볼 수 있었습니다.

▶ 느낀 점

: 이번 과제를 통해 느낀 점은 머리로만 알고 있다고 생각했던 것은 참된 앎이 아니라는 것과 구현이든 복잡도 계산이든 그림 그리기든 깊이있는 '생각'을 통해 '이해'하는 것이 중요하다는 것을 느끼게 되었습니다.

과제는 이상으로 마치겠습니다. 감사합니다.