201201871 서현택

1.목표

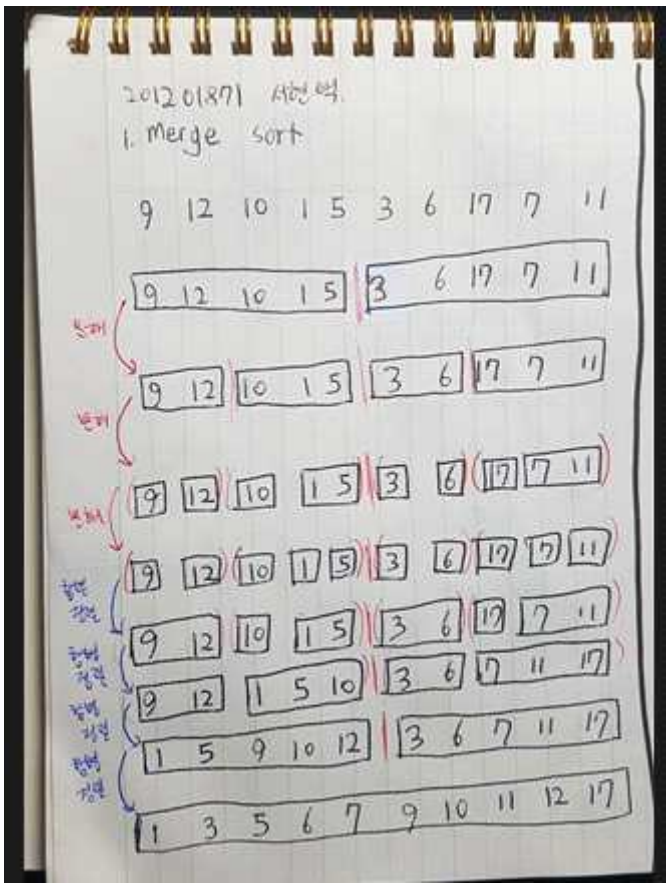
패키징을 이용하여 정렬 함수들을 묶고, sort_test를 정렬한다.
각 정렬별로 성능 비교를 해본다.

2.과제를 해결하는 방법

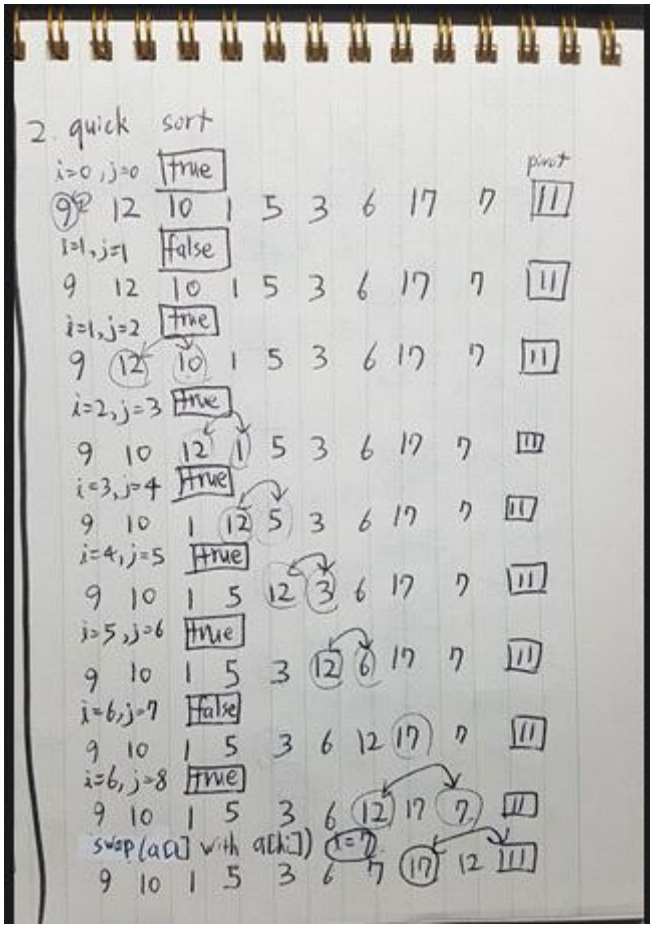
1. 각 정렬 알고리즘의 이해
2. swap 함수 이해
3. matplotlib를 통한 그래프 표시

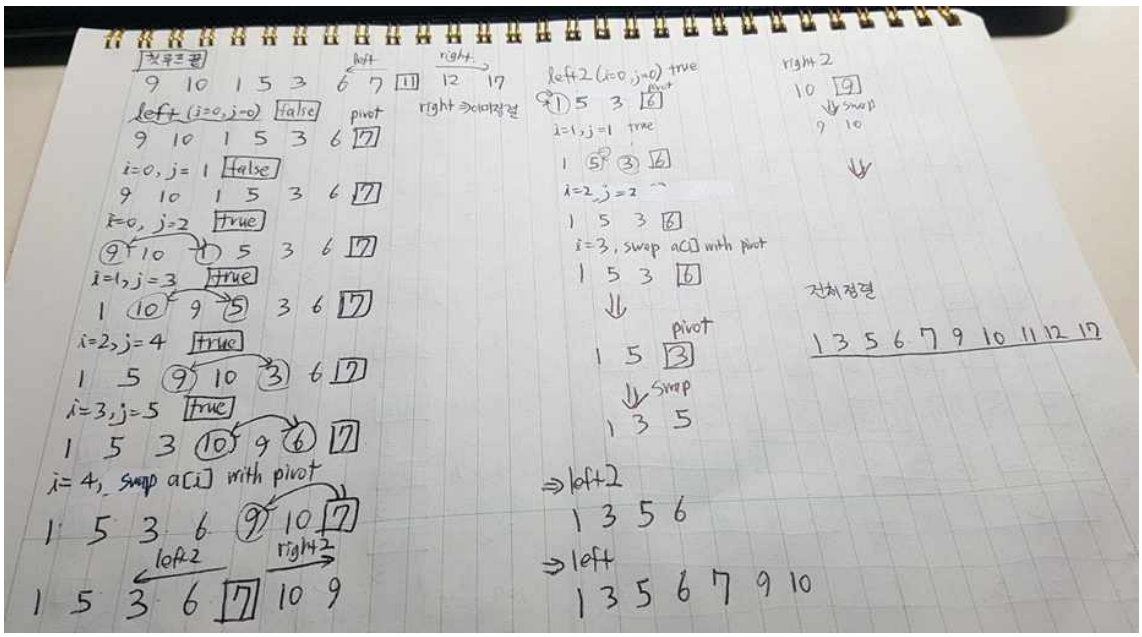
3.과제를 해결한 방법

1) 그림을 통한 merge sort 이해



2) 그림을 통한 quick sort 이해


2. quick sort
i=0, j=0 True
9 12 10 1 5 3 6 17 7 11
pivot
i=1, j=1 False
9 12 10 1 5 3 6 17 7 11
i=1, j=2 True
9 12 10 1 5 3 6 17 7 11
i=2, j=3 True
9 10 12 1 5 3 6 17 7 11
i=3, j=4 True
9 10 1 12 5 3 6 17 7 11
i=4, j=5 True
9 10 1 5 12 3 6 17 7 11
i=5, j=6 True
9 10 1 5 3 12 6 17 7 11
i=6, j=7 False
9 10 1 5 3 6 12 17 7 11
i=6, j=8 True
9 10 1 5 3 6 12 17 7 11
swap(a[i] with a[h])
9 10 1 5 3 6 7 17 12 11


첫 부분 끝
9 10 1 5 3 6 7 11 12 17
left
right
left (i=0, j=0) false
pivot
9 10 1 5 3 6 17
right 오른쪽정렬
i=0, j=1 false
9 10 1 5 3 6 17
i=0, j=2 True
9 10 1 5 3 6 17
i=1, j=3 True
1 10 9 5 3 6 17
i=2, j=4 True
1 5 9 10 3 6 17
i=3, j=5 True
1 5 3 10 9 6 17
i=4, swap a[i] with pivot
1 5 3 6 9 10 17
left2
right2
1 5 3 6 17 10 9
left2 (i=0, j=0) true
pivot
1 5 3 6
i=1, j=1 true
1 5 3 6
i=2, j=2
1 5 3 6
i=3, swap a[i] with pivot
1 5 3 6
pivot
1 5 3
swap
1 3 5
⇒left2
1 3 5 6
⇒left
1 3 5 6 7 9 10
right 2
10 9
swap
9 10
전체정렬
1 3 5 6 7 9 10 11 12 17

3) swap

```
def _swap(input_list, index_a, index_b):
    temp = input_list[index_a]
    input_list[index_a] = input_list[index_b]
    input_list[index_b] = temp
```

swap을 실행할 리스트와 바꿀 두 변수의 index인 index_a, index_b를 넣고, temp에 리스트의 index_a번째 데이터를 저장한다. input_list[index_a]를 input_list[index_b]를 저장하고, input_list[index_b]에는 temp를 저장한다. 이렇게 함으로써 리스트 내 두 변수의 위치가 바뀌게 된다.

4) bubble

```
def bubble_sort(target, reverse=False):
    swapped = True
    while swapped:
        swapped = False
        for index in range(0, len(target) - 1, 1):
            if reverse:
                if target[index] < target[index+1]:
                    _swap(target, index, index + 1)
                    swapped = True
            else:
                if target[index] > target[index+1]:
                    _swap(target, index, index + 1)
                    swapped = True
```

리스트인 target과 False로 초기화된 reverse를 받고, swapped에 True를 저장한다. swapped가 True일 때까지 반복을 하는데, 반복문에서 처음엔 swapped에 False를 저장한 뒤 0부터 target의 길이-1까지 반복하면서 reverse 일 때 target[index]와 target[index+1]을 비교하여 target[index+1]이 더 크다면 두 변수를 바꾼다. reverse가 아닐 때는 target[index]가 더 크다면 두 변수를 바꾼다.

5) selection

```
def selection_sort(target, reverse=False):
    if reverse is False:
        for j in range(0, len(target) - 1):
            i_min = j
            for i in range(j+1, len(target)):
                if target[i] < target[i_min]:
                    i_min = i
            if i_min != j:
                _swap(target, i_min, j)
    else:
        for j in range(0, len(target) - 1):
            i_min = j
            for i in range(j+1, len(target)):
                if target[i] > target[i_min]:
                    i_min = i
            if i_min != j:
                _swap(target, i_min, j)
```

리스트인 target과 False로 초기화된 reverse를 받고, reverse가 false라면 0부터 target의 길이-1까지 반복하면서 I_min에 j를 저장하여 index의 최소값을 저장한 뒤, 다시 j+1부터 target까지의 반복을 통해 target[i_min]보다 작은 target[i]가 있다면 I_min에 해당 인덱스 I를 저장한다. 반복을 마친 뒤 I_min이 j와 같지 않다면 target에서 I_min과 j에 위치한 변수를 바꾼다.

reverse 일 때는, 0부터 target의 길이-1까지 반복하면서 I_min에 j를 저장하여 index의 최소값을 저장한 뒤, 다시 j+1부터 target까지의 반복을 통해 target[i_min]보다 큰 target[i]가 있다면 I_min에 해당 인덱스 I를 저장한다. 반복을 마친 뒤 I_min이 j와 같지 않다면 target에서 I_min과 j에 위치한 변수를 바꾼다.

6) insertion

```
def insertion_sort(target, reverse=False):
    if reverse is False:
        i = 1
        while i < len(target):
            j = i
            while j > 0 and target[j-1] > target[j]:
                _swap(target, j, j-1)
                j = j-1
            i = i+1
    else:
        i = 1
        while i < len(target):
            j = i
            while j > 0 and target[j - 1] < target[j]:
                _swap(target, j, j-1)
                j = j - 1
            i = i + 1
```

리스트인 target과 False로 초기화된 reverse를 받고, reverse가 false라면 I가 target의 길이보다 작은 동안 반복하면서 j에 I를 저장하고 다시 j>0 이고 target[j-1]>target[j]때까지 반복하면서, j와 j-1을 바꾼다. 그리고 j에는 j-1를 저장하고, 해당 반복이 끝나면 바깥 반복문에서는 I에 I+1를 저장한다.
reverse가 true라면 I가 target의 길이보다 작은 동안 반복하면서 j에 I를 저장하고 다시 j>0 이고 target[j-1]<target[j]때까지 반복하면서, j와 j-1을 바꾼다. 그리고 j에는 j-1를 저장하고, 해당 반복이 끝나면 바깥 반복문에서는 I에 I+1를 저장한다.

7) merge

```python
def _merge_sort(target, reverse=False):
    if len(target) <= 1:
        return target

    left = []
    right = []
    mid = len(target)//2

    for x in target[:mid]:
        left.append(x)
    for x in target[mid:]:
        right.append(x)

    left = _merge_sort(left, reverse)
    right = _merge_sort(right, reverse)

    return merge(left, right, reverse)


def merge_sort(target, reverse=False):
    result = _merge_sort(target, reverse)
    target.clear()
    target.extend(result)
```

```python
def merge(left, right, reverse):
    result = []
    while len(left) > 0 and len(right) > 0:
        if reverse is False:
            if left[0] <= right[0]:
                result.append(left.pop(0))
            else:
                result.append(right.pop(0))
        else:
            if left[0] <= right[0]:
                result.append(right.pop(0))
            else:
                result.append(left.pop(0))

    while len(left) > 0:
        result.append(left.pop(0))

    while len(right) > 0:
        result.append(right.pop(0))

    return result
```

_merge_sort에서는 target의 길이가 1보다 작거나 같다면 target을 그대로 반환한다. list인 left와 right 만들고, mid에는 target길이를 2로 나눈 것을 저장한다. for문을 통해서 left에 target의 미드 전까지를 저장하고, right에 mid후를 저장한다. 다시 left, right에는 left, right에서의 _merge_sort 값을 저장한다.

merge_sort에서는 result에 정렬된 _merge_sort 값을 저장하여 target에 최종적으로 정렬된 상태를 저장한다.

merge에서는 left와 right의 길이가 0보다 큰 동안, reverse일 때와 아닐 때를 구별한 뒤 pop을 이용하여 list의 원소를 삭제하면서 result에 값을 저장하고, 최종적으로 완성된 result를 반환한다.

8) quick

```
def partition(target, lo, hi, reverse):
    pivot = target[hi]
    i = lo
    if reverse is False:
        for j in range(lo, hi):
            if target[j] < pivot:
                _swap(target, i, j)
                i = i+1
    else:
        for j in range(lo, hi):
            if target[j] > pivot:
                _swap(target, i, j)
                i = i + 1
    _swap(target, i, hi)
    return i
```

```
def quick(target, lo, hi, reverse=False):
    if lo < hi:
        p = partition(target, lo, hi, reverse)
        quick(target, lo, p-1, reverse)
        quick(target, p+1, hi, reverse)


def quick_sort(target, reverse = False):
    quick(target, 0, len(target)-1, reverse)
```

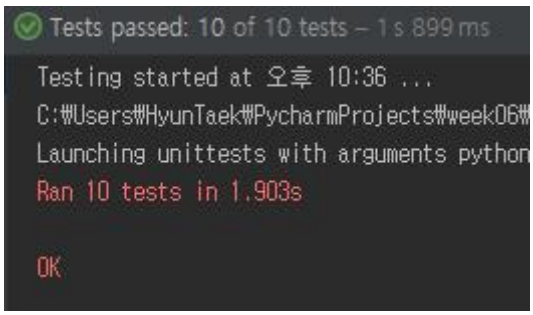
partion에 list인 target, 리스트 인덱스 최소값 lo, 최대값 hi, reverse=False를 저장하고, pivot에는 hi에서의 값을 저장한다. I에는 lo 값을 저장하고, reverse가 false일 때 는 lo부터 hi까지 반복하며 target[j]가 pivot보다 작다면 I와 j에 위치한 값을 바꾸고 I=i+1를 하면서 반복을 계속한다.
reverse가 true라면 lo부터 hi까지 반복하며 target[j]가 pivot보다 크다면 I와 j에 위치한 값을 바꾸고 I=i+1를 하면서 반복을 계속한다.

quick에 list인 target, 리스트 인덱스 최소값 lo, 최대값 hi, reverse=False를 저장하고, lo<hi라면 partition(target, lo, hi, reverse)를 p에 저장한다. 또한 quick에 p-1까지의 target과 p+1부터의 target을 넣어 재귀를 하게 한다.

이런 과정을 통해 만들어진 quick에 target, 인덱스 최소값 0, len(target)-1, reverse를 저장하여 test코드 형식이 들어갈 수 있게한다.

4.결과화면

1) sort_test



2) draw_sorting_comparison

```
C:₩Users₩HyunTaek₩PycharmProjects₩week06₩venv₩Scripts₩python.exe C:/Users/HyunTaek/PycharmProje
[0.0, 0.015625, 0.046875, 0.078125, 0.125, 0.171875, 0.265625, 0.296875, 0.4375, 0.515625]
[0.0, 0.015625, 0.015625, 0.015625, 0.03125, 0.03125, 0.046875, 0.078125, 0.078125, 0.109375]
[0.015625, 0.0, 0.03125, 0.046875, 0.0625, 0.109375, 0.109375, 0.203125, 0.1875, 0.28125]
[0.0, 0.015625, 0.0, 0.015625, 0.0, 0.015625, 0.0, 0.015625, 0.015625, 0.015625]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.015625, 0.015625]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

Process finished with exit code 0
```