201201871 서현택

1.목표

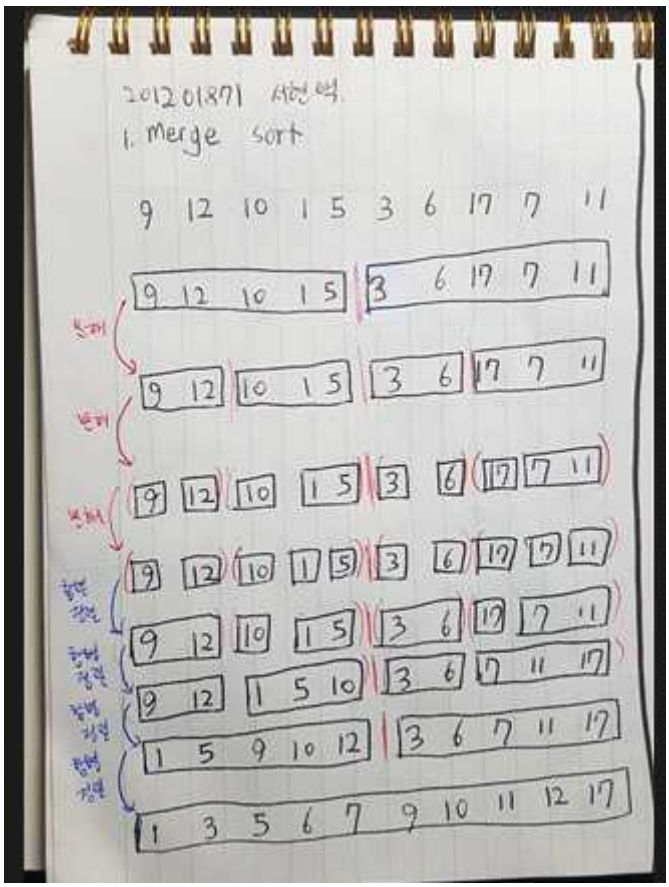
패키징을 이용하여 정렬 함수들을 묶고, sort_test를 정렬한다.
각 정렬별로 성능 비교를 해본다.

2.과제를 해결하는 방법

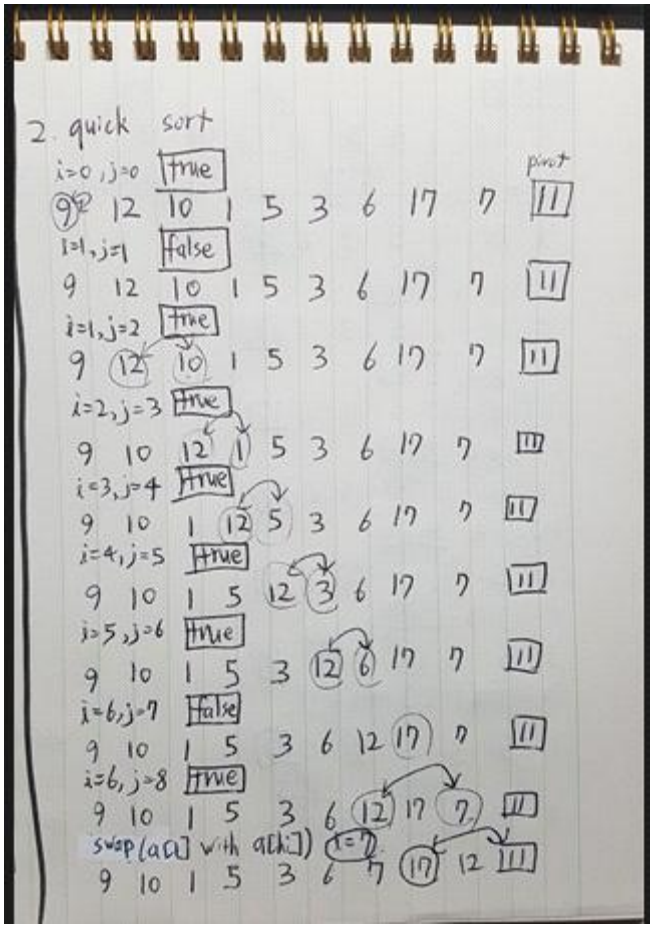
1. 각 정렬 알고리즘의 이해
2. swap 함수 이해
3. matplotlib를 통한 그래프 표시

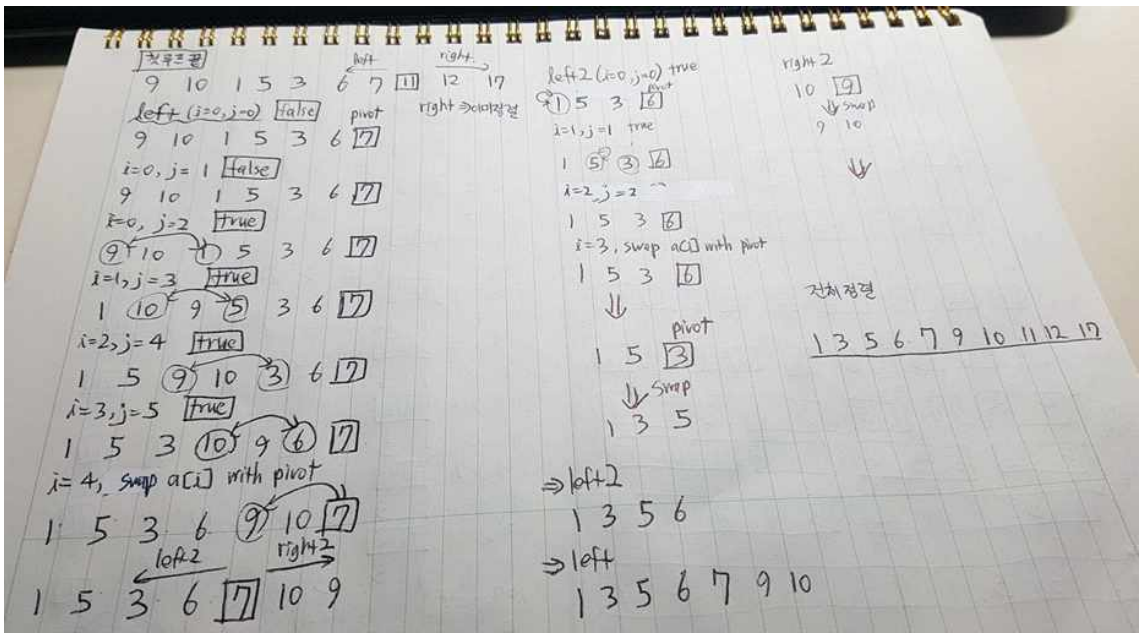
3.과제를 해결한 방법

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 (i=0, j=0) false pivot
9 10 1 5 3 6 17
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
right 정렬
left2 (i=0, j=0) true
7 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
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)
```

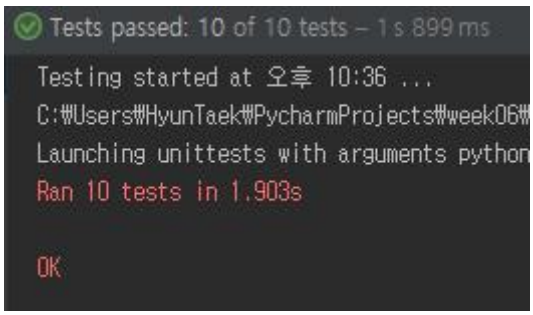
partion에 list인 target, 리스트 인덱스 최소값 lo, 최대값 hi, reverse=False를 저장하고, pivot에는 hi에서의 값을 저장한다. I에는 lo 값을 저장하고, reverse가 false일 때 는 lo부터 hi까지 반복하며 target[j]가 pivot보다 작다면 I와 j에 위치한 값을 바꾸고 I=i+1를 하면서 반복을 계속한다.
reverse가 true라면 lo부터 hi까지 반복하며 target[j]가 pivot보다 크다면 I와 j에 위치한 값을 바꾸고 I=i+1를 하면서 반복을 계속한다.

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

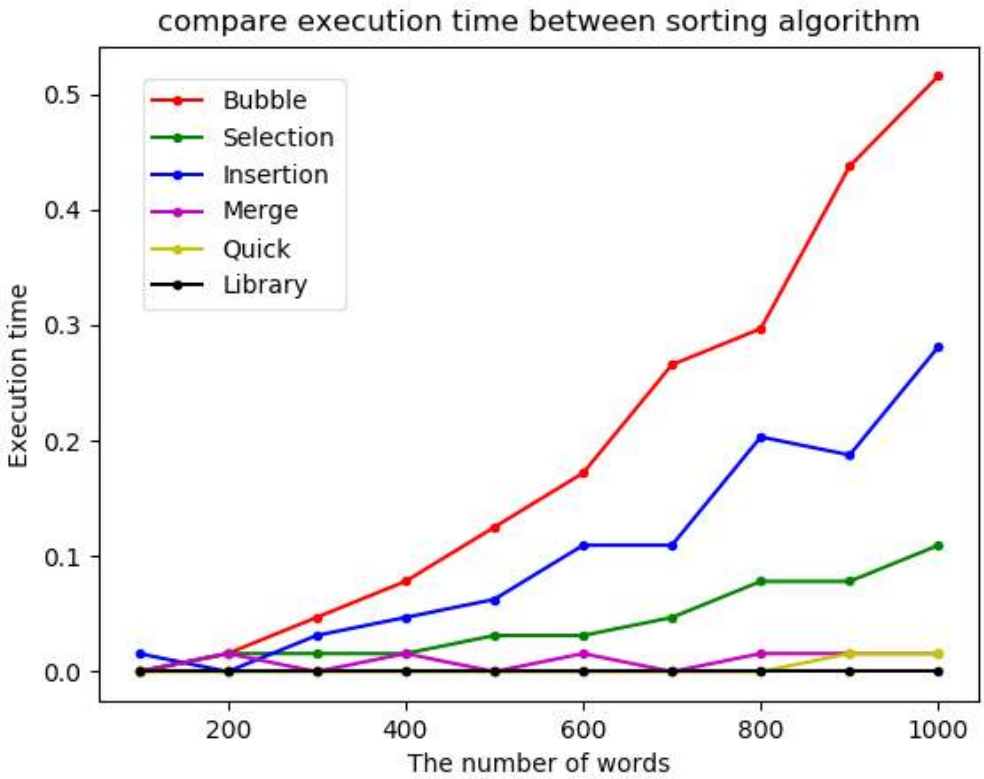
1) sort_test



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
```



1 개요

정렬은 순서가 있는 '모음'에서 기준에 맞게 순서를 변경해주는 것을 말한다. 예컨대 가나다 순으로 '정렬'되어있는 사전이나 숫자로 순서가 표시된 설명서 등이 있다. 만일 이러한 정보가 기준없이 나열되어 있다면 원하는 정보를 찾기 굉장히 어려울 것이다. 이처럼 정렬이 되어있지 않으면 많은 불편을 초래하기 때문에 정렬은 실생활에서 자연스레 사용되고 있다.

일상 생활에 자연스레 녹아있기 때문에 '정렬'은 프로그래밍에서도 전통적인 알고리즘 문제로 꼽힌다. 굉장히 많은 정렬 '알고리즘'이 있으며, 각각의 정렬들은 그 특징이 있기 때문에 적재적소에 활용된다. 즉, 시간복잡도와 공간복잡도가 낮은 정렬 뿐 아니라 성능은 조금 나쁘더라도 특정한 상황에서 더 유용하게 활용되는 정렬 방법이 있다. 기준에 맞게 순서를 변경한다는 결과물은 같지만 연산하는 방법에 따라서 성능/활용 가능 영역이 달라지기 때문에 알고리즘을 공부하는 우리는 꼭 배워야한다.

Python 3에는 이미 '정렬'이 구현되어 있다. 따라서 '순서'와 '중복'이 있는 자료형인 `List`에 `sort()` 혹은 `sorted()`를 호출하여 기준대로 나열할 수 있다. 하지만 우리는 '정렬 알고리즘'을 공부하는데에 목적이 있으므로 내부에 구현되어 있는 도구를 사용하지 않고, `List`에 대한 정렬을 학습한다. 우리는 **Bubble sort**, **Selection sort**, **Insertion sort**, **Merge sort**, **Quick sort** 총 5개의 정렬 알고리즘을 살펴본다. 그리고 마지막으로 각 정렬의 성능을 측정하여 그래프로 출력하고 비교한다.

2.1 Bubble Sort

거품 정렬은 마치 거품 안에서 연산이 이루어지는 것 같다는 의미로 명명되었다. 거품 안에 앞뒤 총 2개의 원소를 넣어두고 기준에 맞게 정렬이 되었는지 확인한다.

만약 기준에 맞게 정렬되어있지 않다면 그 순서를 바꾼다. 그 후 그 다음 원소 2개를 거품안에 넣고 같은 연산을 반복한다.

Listing 1: Buuble sort 정렬 순서(최적화 버전)

```
[4, 19, 1, 9, 10]: 초기 상태 ( 입력 ), 오름차순 정렬 , () 로 거품
    표시
------ 첫번째 반복 ------
[(4, 19), 1, 9, 10]: 위치 유지
[4, (19, 1), 9, 10]: 위치 변경
-> [4, (1, 19), 9, 10]
[4, 1, (19, 9), 10]: 위치 변경
-> [4, 1, (9, 19), 10]
[4, 1, 9, (19, 10)]: 위치 변경
-> [4, 1, 9, (10, 19)]
------ 두번째 반복 ------
[(4, 1), 9, 10, 19]: 위치 변경
-> [(1, 4), 9, 10, 19]
[1, (4, 9), 10, 19]: 위치 유지
[1, 4, (9, 10), 19]: 위치 유지
[1, 4, 9, (10, 19)]: 위치 유지
------ 세번째 반복 ------
[(1, 4), 9, 10, 19]: 위치 유지
[1, (4, 9), 10, 19]: 위치 유지
[1, 4, (9, 10), 19]: 위치 유지
[1, 4, 9, (10, 19)]: 위치 유지
------ 반복동안 위치 변경이 없었으므로 종료 ------
[1, 4, 9, 10, 19]: 정렬 상태 ( 출력 )
```

위의 예제처럼 2개의 원소를 계속 비교하며 N-1번 반복하여 정렬을 완료할 수 있다. 하지만 약간의 트릭(True / False)을 사용하여 매 반복마다 위치가 변경되었는지 확인하는 것으로 N-1번 반복하기 전에 정렬을 끝낼 수 있다. Bubble sort의 수도 코드는 아래와 같다.

Listing 2: Bubble sort 수도 코드

```
1  procedure bubbleSort( A : list of sortable items )
2      n = length(A)
3      repeat
4          swapped = false
5          for i = 1 to n-1 inclusive do
6              if A[i-1] > A[i] then
```

```
7                   swap(A[i-1], A[i])
8                   swapped = true
9               end if
10          end for
11          n = n - 1
12      until not swapped
13  end procedure
```

처음 배우는 정렬을 수도 코드로만 이해하면 어려우므로 Python 3 코드도 함께 살펴보자. 우리는 아래의 코드에서 `reverse`라는 인자를 활용하여 오름차순과 내림차순을 선택할 수 있음을 배운다. 이번 실습 및 과제에서 배우는 모든 정렬에는 `reverse`라는 인자를 활용하여 구현하는 것이 목표이므로 코드를 잘 읽어보도록 하자. 또한 Python 3에서 _(언더바)로 시작하는 함수는 '비공개' 함수로 간주하므로 `_swap()`과 같이 내부에서만 활용하는 함수에 활용하도록 하자.

2.2 Selection Sort

선택 정렬은 원소를 하나씩 선택하여 '정렬된' 순서로 놓아 문제를 해결한다. 가장 작은 원소를 선택하여 정렬할 수도 있고 가장 큰 원소를 선택하여 정렬할 수도 있으니 원하는 방법으로 구현 가능하다. '정렬된 리스트'에 원소를 하나씩 추가한다는 개념으로 접근하면 이해하기 쉽다. 예제를 통해 살펴보도록 하자.

Listing 4: Selection sort 정렬 순서

```
[4, 19, 1, 9, 10]: 초기 상태 ( 입력 ), 오름차순 , 앞에서부터
    정렬
**: 채울 위치 , (): 선택된 원소
[**4, 19, (1), 9, 10]: 위치 변경
-> [**(1), 19, 4, 9, 10]
[1, **19, (4), 9, 10]: 위치 변경
-> [1, **(4), 19, 9, 10]
[1, 4, **19, (9), 10]: 위치 변경
-> [1, 4, **(9), 19, 10]
[1, 4, 9, **19, (10)]: 위치 변경
-> [1, 4, 9, **(10), 19]
[1, 4, 9, 10, 19]: 정렬 상태 ( 출력 )
```

위의 예제처럼 한칸씩 정렬시켜가며 순서를 맞추는 방법이 **선택 정렬**이다. Python 3로 구현해야하므로 수도 코드도 살펴보자.

Listing 5: Selection sort 수도 코드

```
1  int i,j;
2  int n;
```

```
3  for (j = 0; j < n-1; j++)
4  {
5      int iMin = j;
6      for (i = j+1; i < n; i++)
7      {
8          if (a[i] < a[iMin])
9          {
10             iMin = i;
11         }
12     }
13
14     if (iMin != j)
15     {
16         swap(a[j], a[iMin]);
17     }
18 }
```

2.3 Insertion Sort

삽입 정렬은 선택 정렬과 비슷해보이지만 '위치 변경'을 정렬 될 때까지 계속하며 맞는 위치를 찾아가는 부분이 다르다. 예제를 통하여 살펴보도록 하자. 선택 정렬과 '정렬된 범위'를 늘려간다는 점에서 비슷하지만 위치를 찾는 방법이 다르다. 사소한 차이지만 복잡도 및 코드에서는 큰 차이가 나므로 기억해두자.

Listing 6: Insertion sort 정렬 순서

```
[4, 19, 1, 9, 10]: 초기 상태 ( 입력 ), 오름차순
(): 선택된 원소 , **: 정렬된 리스트의 끝
[(4)**, 19, 1, 9, 10]: 첫 원소이자 정렬된 리스트의 끝 ,
    정렬됨
[4, (19)**, 1, 9, 10]: 위치 변경 시작
-> [4, (19)**, 1, 9, 10]
[4, 19, (1)**, 9, 10]: 위치 변경 시작
-> [4, (1), 19**, 9, 10]
-> [(1), 4, 19**, 9, 10]
[1, 4, 19, (9)**, 10]: 위치 변경 시작
-> [1, 4, (9), 19**, 10]
[1, 4, 9, 19, (10)**]: 위치 변경 시작
-> [1, 4, 9, (10), 19**]
[1, 4, 9, 10, 19]: 정렬 상태 ( 출력 )
```

삽입 정렬의 수도코드는 다음과 같다.

Listing 7: Insertion sort 수도 코드

```
1  i ← 1
2  while i < length(A)
3      j ← i
4      while j > 0 and A[j−1] > A[j]
5          swap A[j] and A[j−1]
6          j ← j − 1
7      end while
8      i ← i + 1
9  end while
```

2.4 Merge Sort

지금까지의 정렬은 시간 복잡도가 높아서 실제로 활용하기 어려웠다. 하지만 **병합 정렬**은 시간 복잡도가 낮아 활용하기 좋으며, 특히 빅데이터 처리에서 큰 강점을 보인다. 예제로 살펴보도록 하자.

Listing 8: Merge sort 정렬 순서

```
[4, 19, 1, 9, 10]: 초기 상태 ( 입력 ), 오름차순
(): 원소 블락 ,
[(4, 19), (1, 9, 10)]: 재귀 호출
−>[(4), (19)]: 최소 원소 , 합병 시작
−−>[(4, 19)]: 부분 정렬

−>[(1), (9, 10)]: 재귀 호출
−−>[(1)]: 최소 원소 , 합병 시작
−−>[(9), (10)]: 최소 원소 , 합병 시작
−−−>[(9, 10)]: 부분 정렬
−−−−>[(1, 9, 10)]: 부분 정렬

−−−−−>[(1, 4, 9, 10, 19)]: 부분 정렬
[1, 4, 9, 10, 19]: 정렬 상태 ( 출력 )
```

합병 정렬은 재귀적 기법을 사용하기 때문에 예제로 따라가기 굉장히 난해한 알고리즘이다. 수도 코드로 살펴보도록 하자. 2개의 함수로 구성되어 하나는 '분할', 다른 하나는 '합병'을 수행함에 집중하자.

Listing 9: Merge sort 수도 코드

```
function merge_sort(list m)
    if length of m ≤ 1 then
        return m

    var left := empty list
    var right := empty list
    for each x with index i in m do
        if i < (length of m)/2 then
            add x to left
        else
            add x to right

    left := merge_sort(left)
    right := merge_sort(right)

    return merge(left, right)

function merge(left, right)
    var result := empty list

    while left is not empty and right is not empty do
        if first(left) ≤ first(right) then
            append first(left) to result
            left := rest(left)
        else
            append first(right) to result
            right := rest(right)

    while left is not empty do
        append first(left) to result
        left := rest(left)
    while right is not empty do
        append first(right) to result
        right := rest(right)
    return result
```

2.5 Quick Sort

퀵 정렬은 다른 원소와 비교만으로 정렬을 하는 알고리즘으로 알고리즘 테스트에서 가장 많이 활용되는 알고리즘 중 하나다. 이론적 성능이 가장 좋은 것은 아니지만,

특수한 상황이 아닌이상 무난하게 사용할 수 있기에 실제로도 많이 구현되어 사용된다. 예제를 통해 살펴보도록 하자.

Listing 10: Quick sort 정렬 순서

```
[4, 19, 1, 9, 10]: 초기 상태 ( 입력 ), 오름차순 , 마지막 원소
    Pivot
**: Pivot, M: C: L: Left index, R: Right index, (): 재귀 호출시 그룹
-> [4MC, 19, 1, 9, 10**]: Marker, Checker 교환
-> [4, 19M, 1C, 9, 10**]: Marker, Checker 교환
-> [4, 1, 19M, 9C, 10**]: Marker, Checker 교환
-> [4, 1, 9, 19MC, 10**]:
-> [4, 1, 9, 19M, 10**]: Pivot, Marker 교환
[(4, 1, 9), 10, (19)]: index 4 반환

-> [4MC, 1, 9**]: Marker, Checker 교환
-> [4, 1MC, 9**]: Marker, Checker 교환
-> [4, 1, 9**M]: Pivot, Marker 교환
[(4, 1), 9]: index 2 반환

-> [4MC, 1**]
-> [4M, 1**]: Pivot, Marker 교환
[1, 4]
[1, 4, 9]
[1, 4, 9, 10, 19]
```

Quick sort는 이해하기 굉장히 어려우므로 수도 코드로 잘 살펴보아야 한다. 예제와 코드를 따라가며 살펴보는 것이 필수다.

Listing 11: Quick sort 수도 코드(Lomuto 버전)

```
1  algorithm quicksort(A, lo, hi) is
2      if lo < hi then
3          p := partition(A, lo, hi)
4          quicksort(A, lo, p - 1 )
5          quicksort(A, p + 1, hi)
6
7  algorithm partition(A, lo, hi) is
8      pivot := A[hi]
9      i := lo
10     for j := lo to hi - 1 do
11         if A[j] < pivot then

12             swap A[i] with A[j]
13             i := i + 1
14     swap A[i] with A[hi]
15     return i
```