병합 정렬 (merge sort)

1. 병합 정렬(merge sort)

- 재귀용법을 활용한 정렬 알고리즘
  1. 리스트를 절반으로 잘라 비슷한 크기의 두 부분 리스트로 나눈다.
  2. 각 부분 리스트를 다시 재귀적으로 합병 정렬을 이용해 정렬한다.
  3. 두 부분 리스트를 다시 하나의 정렬된 리스트로 합병한다.

https://visualgo.net/en/sorting?slide=1

2. 알고리즘 이해

- 데이터가 네 개 일때 (데이터 개수에 따라 복잡도가 떨어지는 것은 아니므로, 네 개로 바로 로직을 이해해보자.)
- 예 : data_list = [1,9,3,2]
  - 먼저 [1.9], [3,2]로 나누고
  - 다시 앞 부분은 [1], [9] 로 나누고
  - 다시 정렬해서 합친다. [1,9]
  - 다음 [3,2] 는 [3], [2]로 나누고
  - 다시 정렬해서 합친다. [2,3]
  - 이제 [1, 9], [2, 3]을 합친다.
    - 1<2 이니 [1]

        - 9>2 이니 [1,2]
        - 9>3 이니 [1,2,3]
        - 9 밖에 없으니, [1, 2, 3, 9]

3. 알고리즘 구현

- mergesplit 함수 만들기
    - 만약 리스트 개수가 한 개이면 해당 값 리턴
    - 그렇지 않으면, 리스트를 앞뒤, 두 개로 나누기
    - left = mergesplit(앞)
    - right = mergesplit(뒤)
    - merge(left, right)
- merge 함수 만들기
    - 리스트 변수 하나 만들기(sorted)
    - left_index, right_index = 0
    - while left_index < len(left) or right_index < len(right):
        - 만약 left_index 나 right_index가 이미 left또는 right 리스트를 다 순회했다면, 그 반대쪽 데이터를 그대로 넣고, 해당 인덱스 1 증가
        - if left[left_index] < right[right_index]:
            - sorted.append(left[left_index])
            - left_index+=1
        - else:
            - sorted.append(right[right_index])
            - right_index+=1

```
def split_func(data):
  medium = int(len(data) / 2)
  print(medium)
  left = data[:medium]
```

```
  right = data[medium:]
  print(left, right)
```

```
split_func([1,5,3,2,4])
>>2
  [1, 5] [3, 2, 4]
```

재귀용법 활용하기

- mergesplit 함수 만들기
  - 만약 리스트 개수가 한 개이면 해당 값 리턴
  - 그렇지 않으면, 리스트를 앞뒤, 두 개로 나누기
  - left = mergesplit(앞)
  - right = mergesplit(뒤)
  - merge(left, right)

```
def mergesplit(data):
  if len(data) <=1:
    return data
  medium = int(len(data)/2)
  left = mergeplit(data[:medium])
  right = mergesplit(data[merge:])
  return merge(left, right)
```

merge 함수 만들기

- 목표 : left와 right의 리스트 데이터를 정렬해서 sorted_list라는 이름으로 return 하기

```
def merge(left, right):
  merged = list()
  left_point, right_point = 0, 0

  #case1 - left/right 둘 다 있을 때
  while len(left) > left_point and len(right) > right_point:
    if left[left_point] > right[right_point]:
      merged.append(right[right_point])
      right_point+=1
    else:
      merged.append(left[left_point])
```

```
      left_point+=1

  #case2 - right 데이터가 없을 때
  while len(left) > left_point:
    merged.append(left[left_point])
    left_point+=1

  #case3 - left 데이터가 없을 때
  while len(right) > right_point:
    merged.append(right[right_point])
    right_point+=1

  return merged
```

최종 코드

```
def merge(left, right):
  merged = list()
  left_point, right_point = 0, 0

  #case1 - left/right 둘 다 있을 때
  while len(left) > left_point and len(right) > right_point:
    if left[left_point] > right[right_point]:
      merged.append(right[right_point])
      right_point+=1
    else:
      merged.append(left[left_point])
      left_point+=1

  #case2 - right 데이터가 없을 때
  while len(left) > left_point:
    merged.append(left[left_point])
    left_point+=1

  #case3 - left 데이터가 없을 때
  while len(right) > right_point:
    merged.append(right[right_point])
    right_point+=1

  return merged


def mergesplit(data):
  if len(data) <=1:
    return data
  medium = int(len(data)/2)
  left = mergeplit(data[:medium])
  right = mergesplit(data[merge:])
  return merge(left, right)
```

```
import random

data_list = random.sample(range(100), 10)
mergesplit(data_list)
>>[8, 12, 24, 40, 47, 70, 81, 87, 92, 96]
```

4. 알고리즘 분석

- 알고리즘 분석은 쉽지 않음
  - 다음을 보고 이해해보자
    - 몇 단계 깊이까지 만들어지는지를 depth 라고 하고 i로 놓자. 맨 위 단계는 0으로 놓자.
      - 다음 그림에서 n/2^2 는 2단계 깊이라고 해보자.
      - 각 단계에 있는 하나의 노드 안의 리스트 길이는 n/2^2가 된다.
      - 각 단계에는 2^i개의 노드가 있다.
    - 따라서, 각 단계는 항상 2^i * n/2^i = O(n)
    - 단계는 항상 logn개 만큼 만들어짐, 시간 복잡도는 결국 o(log n),2는 역시 상수이므로 삭제
    - 따라서, 단계별 시간 복잡도 O(n) * O(logn) = O(nlogn)