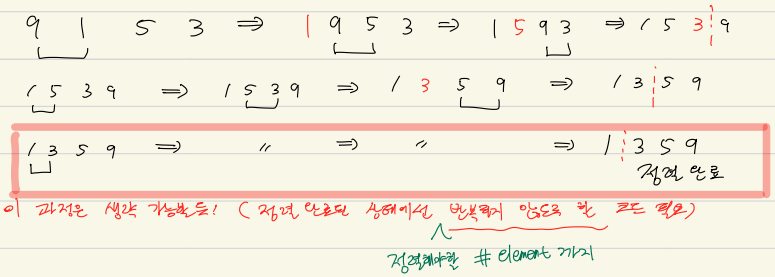
1. 버블 정렬

1) 문제 분석

인접한 숫자는 비교해 오름/내림 차순에 맞게, 인접 숫자는 swap 하는 알고리즘

ex)



이 과정은 생략 가능하도록! (정렬 완료된 상태에선 반복하지 않도록 한 코드 필요)

정렬해야할 # element 까지

2) 로직 정리

정렬 완료여부 확인 용

① 숫자 데이터는 list type 에 저장

② 오름 차순 인 경우, 가장 왼쪽의 두개 인접 숫자를 비교해 큰 숫자는 뒤로!

③ ②를 모든 element 에 반복시행 O(n) ⇒ 1개 element 정렬 완료 후 맨 뒤 확인

④ element 갯수 만큼 반복시행 O(n)

3) 수도 코드

```
data : list type
for i in range ( 1, # element) :
    if unsorted :
        for j in range (i, # elements) :
            if data[j] > data[j+1] :
                swap (i ⟷ i+1)
                "sorted"
    else :
        return data
```

4) 코드 레벨

5) 복잡도

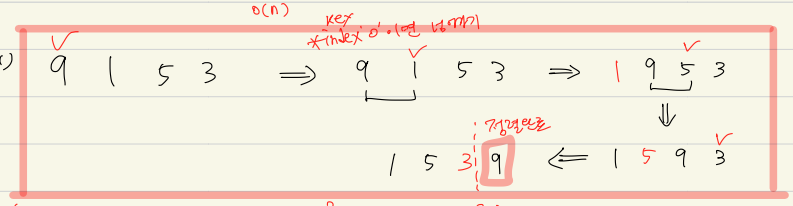
시간 복잡도 : $O(n^2)$ ⇐ for # element 이중!

"n"

2. 삽입정렬

1) 문제분석

Key value index 를 갖고, key index 이전 value 와 O(n) 반복 비교해 오름(내림) 차순에 맞게 작은 값이 왼쪽으로 오도록 swap ⇒ # element 만큼 반복.



(# element -1 만큼 반복!) ^{2} = $O(n^2)$

2) 로직 정리

정렬완료여부 확인 후

① 숫자 데이터는 list type 에 저장

② 오름 차순인 경우, key index는 두번째 부터 시작 ⇒ "index-1" "index" 비교해 작은 것이 왼쪽에 오도록 swap 반복 (# element, O(n))

③ ②를 모든 element 에 반복시행 O(n) ⇒ 1개 element 정렬 완료 후 맨 뒤 확치

④ element 갯수 만큼 반복시행 O(n)

3) 수도코드

```
Data : list type
for _ in range(0, #element):
    if unsorted:
        for i in range(1, #element):   * key index
            Data[i-1] 이 더 작으면, swap(Data[i-1] ⟷ Data[i])
            "Sorted"
    else:
        return Data
```

4) 코드레벨

5) 복잡도

시간 복잡도 : $O(n^2)$ ⇐ for # element 이중!

"n"

3. 선택 정렬

1) 문제 분석

key index를 지정 후, 나머지 index의 value 중 최소값을 찾아 (O(n))

key index의 value 와 최소값을 swap ⇒ # element 만큼 반복! (O(n))

(오름차순의 경우 해당)

(1개 정렬완료 한 것)

ex) 9 1 5 3 (minimum?) ⇒ <최소값 탐색> range(key , len(data)): 작으면 min_ 에 저장 ⇓ for loop로 갱신 ⇓ min_ : 최소값 ⇒ swap ⇒ 1 9 5 3 (key ⟺ min)

정렬 완료 1 | 9 5 3 (minimum?) ⇒ // ⇒ // ⇒ 1 3 5 9 (key ⟺ min)

1 3 | 5 9 (minimum?)

2) 로직 정리

① 숫자 데이터를 list type 에 저장

② 오름 차순인 경우, key index = 0 에서 시작해, [key index , last element] 중 최소 값을 찾아 (O(n)) swap (key index ⟷ 최소값)

③ ②를 모든 element 에 반복시행 (O(n)) ⇒ 1개 element 정렬 완료 후 맨 왼쪽 위치

3) 수도코드

data : list type

```
for i (0 to len(data)-1) :        # i : key
    min_idx = i                   * min_idx 초기화
    for idx (i+1 to len(data)):   # idx : 탐색 대상 index
        data[idx] 가 data[min_idx] 보다 작다면,
        min_idx = idx
    swap ( data[min_idx] , data[i] )
```

ex) 정렬 완료 1 | 9 5 3 (i, idx →, min_idx(1), min_idx(2), min_idx(3))

for loop로 idx 돌며 min_idx를 갱신하는 과정!

4) 코드 레벨

5) 복잡도

시간 복잡도 : O(n²) ⟸ for # element 이중! ("n")

* Dynamic Programming / Divide and conquer
"DP"

: 일반화된 문제 (큰 문제) 를 잘게 나눠서 이를 바탕으로 해결한다는 공통점이 있다.
하지만,
DP는 Memoization을 토대로 f(100) 계산에 f(1) 연산 결과를 기억하고 (memoization) 활용한다는 점. "피보나치"
DC는 recursive call 구조를 통해 memoization 없이 문제를 해결 (merge 과정에서 특정 연산을 적용하여 큰 문제를 해결)
"merge sort, quick sort"

1) 피보나치 문제 (recursive call)

```
In [7]:
def fibo(num):
    if num <= 1:
        return num
    return fibo(num - 1) + fibo(num - 2)

In [8]:
fibo(4)
Out[8]:
3
```

f(4) = f(3) + f(2) ⟹ 연산이 반복됨... 비효율적
f(3) = f(2) + f(1)
f(2) = f(1) + f(0)
∴ DP - Memoization 기법 활용.

2) 피보나치 문제 (Memoization - DP)

```
In [10]:
def fibo_dp(num):
    cache = [ 0 for index in range(num + 1)]
    cache[0] = 0
    cache[1] = 1

    for index in range(2, num + 1):
        cache[index] = cache[index - 1] + cache[index - 2]
    return cache[num]

In [12]:
fibo(10)
Out[12]:
55
```

↳ 0~num까지의 공간생성 ⟹ Memoization 위한 저장공간 생성.
⟹ 재귀함수로 매번 새로 연산하는 것 탈피!
Memoization!!!

4. 병합 정렬 Merge Sort

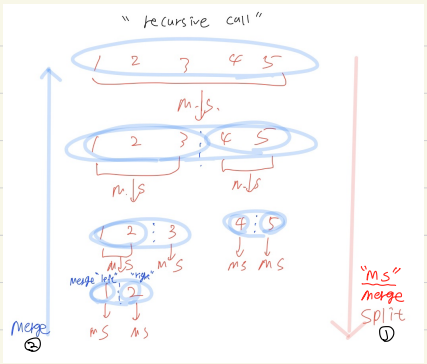
1) 문제 분석

divide and conquer 전략, recursive call로 최소 단위까지 divide 후, 데이터 비교는 토대로 merge를 실시해 정렬 문제는 해결.

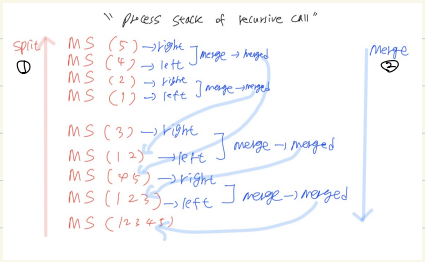
* recursive 구조로 divide process를 stack 후, LIFO로 merge 하여 전체정렬 문제 해결.

ex) 1 2 3 4 5 의 merge sort !

⇒ merge split , merge 함수로 구현해 해결.
(divide) (conquer)



시스템 상 처리 순서는 아래와 같이 진행



2) 로직 정리

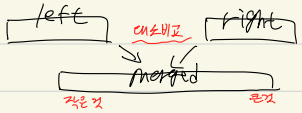
① data을 list type 으로 저장

② len(data) == 1 인 때까지 divide 하는 merge split("ms") 함수 작성

divide
- 2-1) len(data) == 1 이면, return data
- 2-2) recursive call로 활용해 왼/오 로 divide 하도 과정 포함. (merge split 함수로 활용해)

conquer
- 2-3) 나눠진 data (왼/오) 는 merge 하도록 연결!

③ merge 함수 작성 (merge 규칙은 작은게 먼저 !)



3-1) left와 right 의 element를 1개씩 비교후 작은 것을 merged에 "append" + "index+1"

3-2) 또, 3-1)은 index가 각각 left와 right의 길이, 즉 안에서 증가해야 함
만약, index >= len(left) 라면, right의 value는 차례로 merged에 "append" + "index+1"

3) 슈도코드

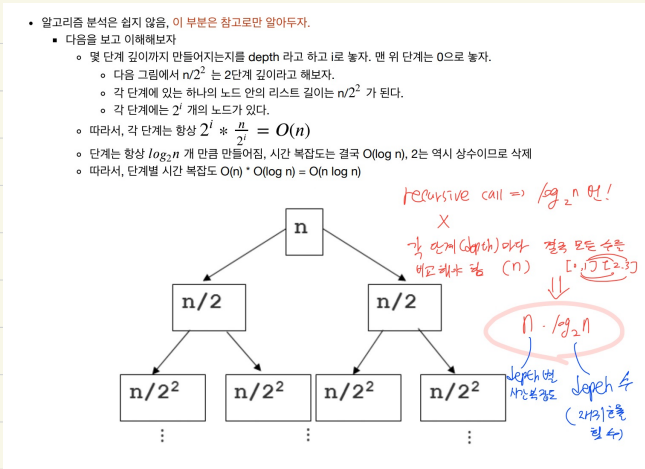
```
data : list type

def merge_split (data):
    if len(data) == 1 :
        return data
    else :
        중간 index 값 ("median") 지정
        left = merge_split(data[ : median])
        right = merge_split(data[median : ])
        return merge (left, right)

def merge (left, right):
    merged = [ ]
    left_index, right_index = 0, 0
    if left, right 모두 index가 범위내 있을 때:
        merged.append( )
        index+1
    elif left에서 index가 벗어났을 때.:
        merged.append ( right[index] )
        right_index+1
    else :
        merged.append ( left[index] )
        left_index+1
    return merged
```

4) 코드레벨

5) 복잡도

" $n \log_2 n$ "

- 알고리즘 분석은 쉽지 않음, 이 부분은 참고로만 알아두자.
  - 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 $n/2^2$ 가 된다.
      - 각 단계에는 2^i 개의 노드가 있다.
    - 따라서, 각 단계는 항상 $2^i * \frac{n}{2^i} = O(n)$
    - 단계는 항상 $log_2 n$ 개 만큼 만들어짐, 시간 복잡도는 결국 O(log n), 2는 역시 상수이므로 삭제
    - 따라서, 단계별 시간 복잡도 O(n) * O(log n) = O(n log n)



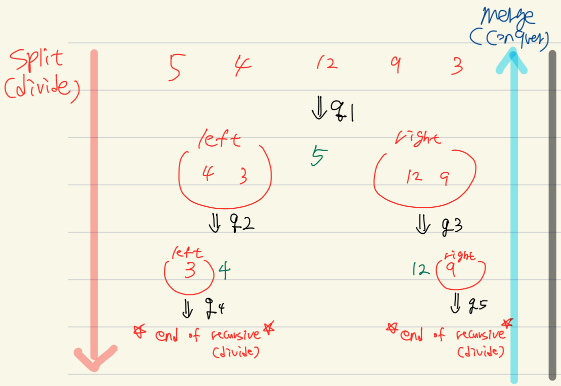
5. 퀵 정렬 Quick sort

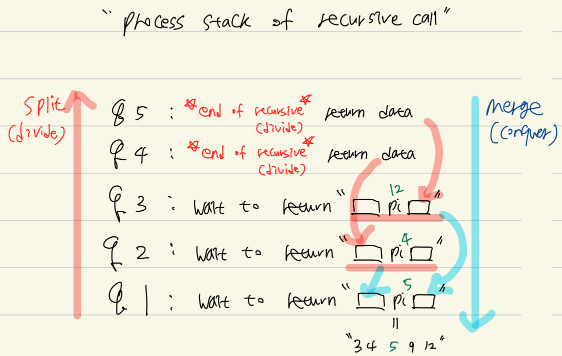
1) 문제 분석

divide and conquer 전략, recursive call로 최소 단위까지 divide 후, "pivot"을 기준으로 대소 비교를 토대로 merge를 실시해 정렬 문제를 해결.

* recursive 구조로 divide process를 stack 후, LIFO로 merge하여 전체정렬 문제 해결.

ex) 5 4 12 9 3 의 quick sort!





2) 로직 정리

① data를 list type으로 받기

② quick_sort 함수 정의

(2-1) pivot 설정, left, right 공간 초기화 (list type)

(2-2) pivot 보다 작으면 left에, 반대면 right에 넣기

(2-3) return으로 divide 와 conquer를 함께 한번에 설정!

⇑ recursive call ⇑ (merge) ⬭ + pivot + ⬭ 을 return!

* merge sort와 달리, merge 시 대소비교 없음 X

3) 수도코드

```
def q_sort (data):

    pivot 설정
    for loop ⇐ data[1:] vs pivot 대소비교
        pivot 보다 작으면 left 저장
        반대면 right 저장

    return q_sort(left) + pivot + q_sort(right)
```

4) 코드레벨

5) 복잡도

- 병합정렬과 유사, 시간복잡도는 O(n log n)
  - 단, 최악의 경우 (정말 극단의 경우) O(n²)
    - 맨 처음 pivot이 가장 크거나, 가장 작으면
    - 모든 데이터를 비교하는 상황이 나옴
    - $O(n^2)$

Worst Case

아래의 배열을 가장 왼쪽 값을 피벗으로 하여 오름차순 정렬해보자