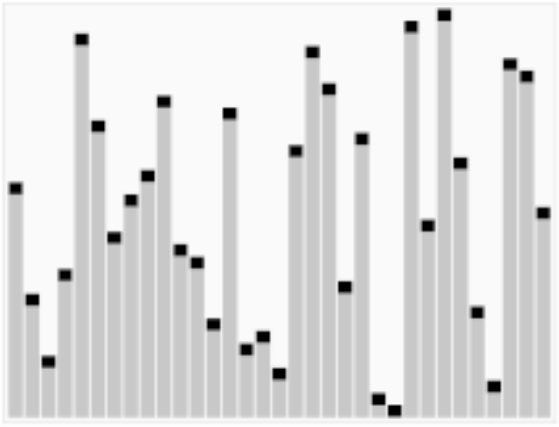
15주차 : Quick Sort

> 퀵 정렬(Quick Sort)은 **분할 정복 방법**을 통해 구현된 정렬알고리즘이다.



난수열에 대해 퀵 정렬을 실행한 그림. 수평선은 피벗 값을 가리킨다.

분류	정렬 알고리즘
자료구조	배열
최악 시간복잡도	**O(n^2)**
최선 시간복잡도	**O(nlogn)**
평균 시간복잡도	**O(nlogn)**

알고리즘

퀵 정렬은 분할 정복(divide and conquer) 방법을 통해 리스트를 정렬한다.

1. 리스트 가운데서 하나의 원소를 고른다. 이렇게 고른 원소를 **피벗**이라고 한다.

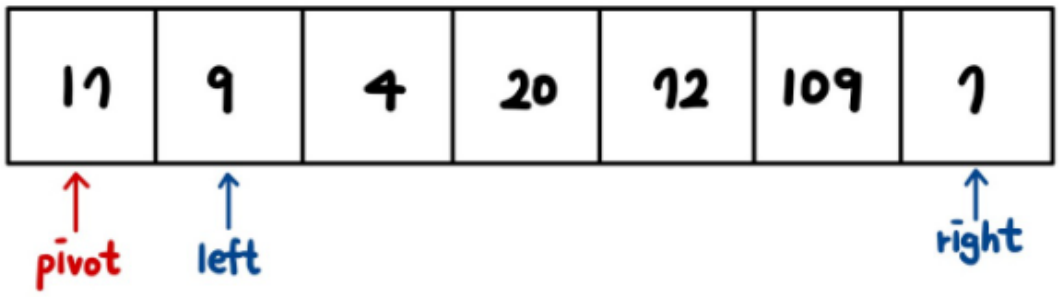
2. 피벗 앞에는 피벗보다 값이 작은 모든 원소들이 오고, 피벗 뒤에는 피벗보다 값이 큰 모든 원소들이 오도록 피벗을 기준으로 리스트를 둘로 나눈다. 이렇게 리스트를 둘로 나누는 것을 **분할**이라고 한다. 분할을 마친 뒤에 피벗은 더 이상 움직이지 않는다.
3. 분할된 두 개의 작은 리스트에 대해 재귀(Recursion)적으로 이 과정을 반복한다. 재귀는 리스트의 크기가 0이나 1이 될 때까지 반복된다.

재귀 호출이 한번 진행될 때마다 최소한 하나의 원소는 최종적으로 위치가 정해지므로, 이 알고리즘은 반드시 끝난다는 것을 보장할 수 있다.

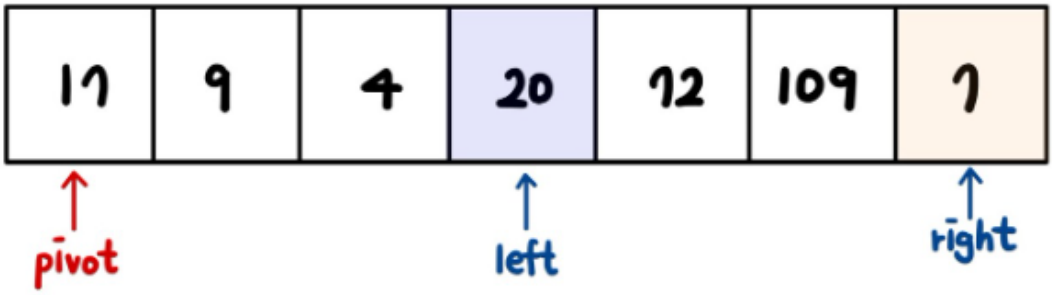
3줄 정리

1. 피벗 선택
2. 배열을 3가지로 분할! (피벗 왼쪽 | 피벗 | 피벗 오른쪽)
3. 재귀 호출!

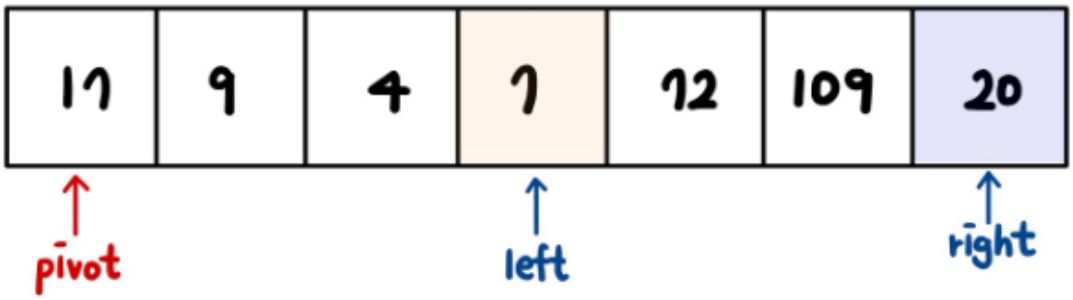
과정



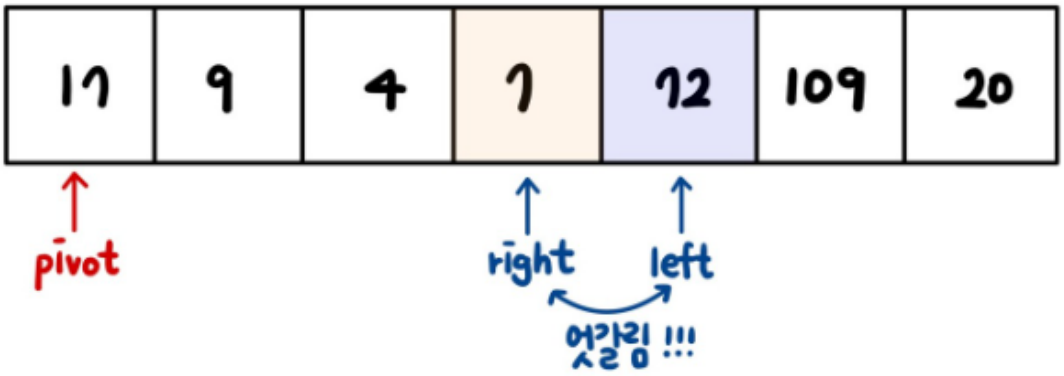
1. 피벗을 선택한다. left와 right를 양 끝 값으로 선택한다



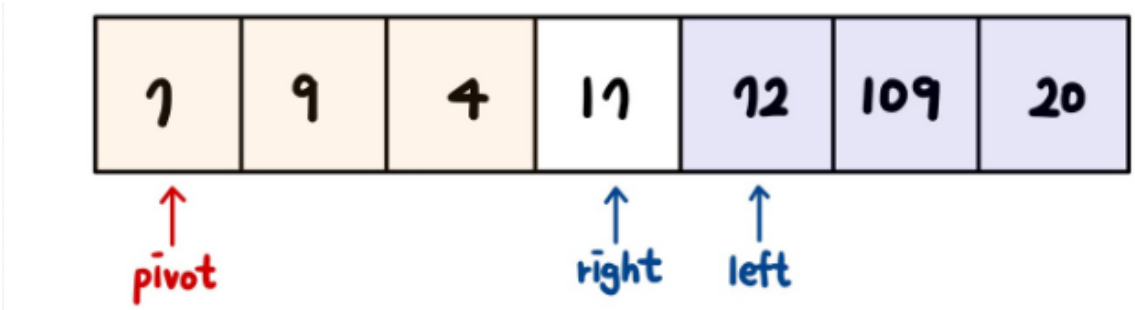
2-1. left와 right를 각 방향으로 진행하며 피벗을 기준으로 값이 크거나 작을경우 잠시 STOP!



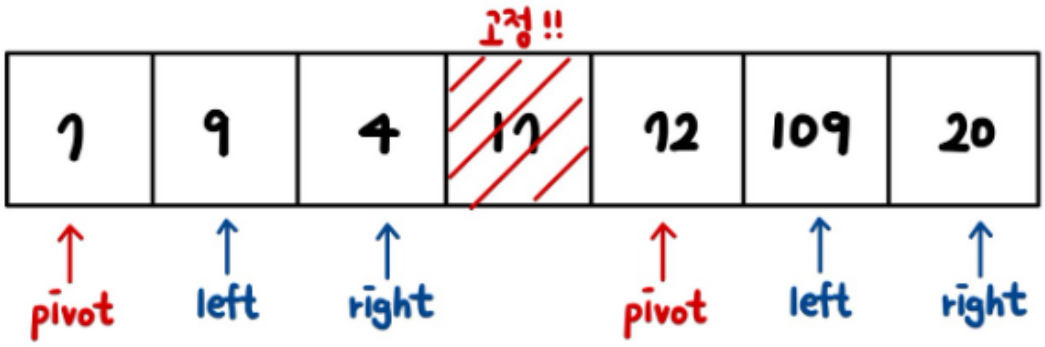
2-2. left, right가 모두 멈춘 경우, 두 값을 swap한다.



left와 right가 엇갈릴때까지 위 과정을 진행한다.



피벗과 right의 값을 바꿔주고 과정을 마무리한다.



3. 피벗 값은 정렬을 마쳤다!(더이상 swap 하지 않음). 피벗의 왼쪽과 오른쪽에 대해 또 다시 퀵소트를 진행한다.

코드

```
array = [5, 7, 9, 0, 3, 1, 6, 2, 4, 8]


def quick_sort(array, start, end):
    if start >= end:  # 원소가 1개인 경우 종료
        return
    pivot = start  # 피벗은 첫 번째 원소
    left = start + 1
    right = end
    while left <= right:
        # 피벗보다 큰 데이터를 찾을 때까지 반복
        while left <= end and array[left] <= array[pivot]:
            left += 1
        # 피벗보다 작은 데이터를 찾을 때까지 반복
        while right > start and array[right] >= array[pivot]:
            right -= 1
        if left > right:  # 엇갈렸다면 right와 피벗을 교체
            array[right], array[pivot] = array[pivot], array[right]
        else:  # 엇갈리지 않았다면 left와 right를 교체
            array[left], array[right] = array[right], array[left]
    # 분할 이후 왼쪽 부분과 오른쪽 부분에서 각각 정렬 수행
    quick_sort(array, start, right - 1)
    quick_sort(array, right + 1, end)


quick_sort(array, 0, len(array) - 1)
print(array)
```

```
# 파이썬 장점 list comprehension

array = [5, 7, 9, 0, 3, 1, 6, 2, 4, 8]

def quick_sort(array):
    # 리스트가 하나 이하의 원소만을 담고 있다면 종료
    if len(array) <= 1:
        return array

    pivot = array[0] # 피벗은 첫 번째 원소
    tail = array[1:] # 피벗을 제외한 리스트

    left_side = [x for x in tail if x <= pivot] # 분할된 왼쪽 부분
    right_side = [x for x in tail if x > pivot] # 분할된 오른쪽 부분

    # 분할 이후 왼쪽 부분과 오른쪽 부분에서 각각 정렬을 수행하고, 전체 리스트를 반환
    return quick_sort(left_side) + [pivot] + quick_sort(right_side)

print(quick_sort(array))
```

▼ 코드

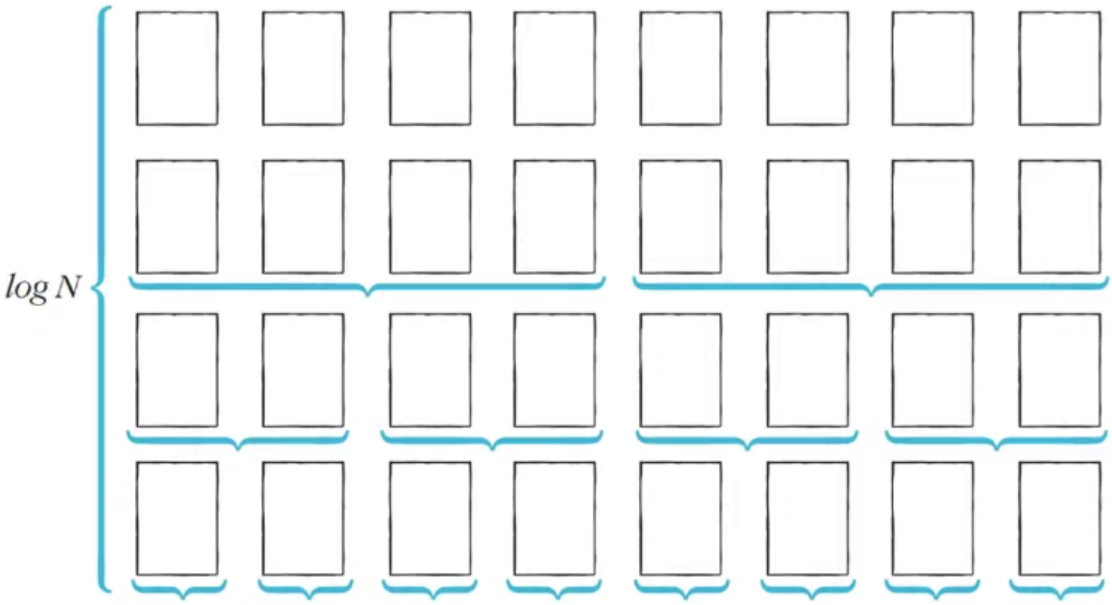
```
import sys
import time, random, copy
sys.setrecursionlimit(1000000)

def quicksort(x):
    if len(x) <= 1:
        return x

    pivot = x[len(x) // 2]
    less = []
    more = []
    equal = []
    for a in x:
        if a < pivot:
            less.append(a)
        elif a > pivot:
            more.append(a)
        else:
            equal.append(a)

    return quicksort(less) + equal + quicksort(more)


def quick_sort(array, start, end):
    if start >= end:  # 원소가 1개인 경우 종료
        return
    pivot = start  # 피벗은 첫 번째 원소
    left = start + 1
    right = end
    while left <= right:
        # 피벗보다 큰 데이터를 찾을 때까지 반복
        while left <= end and array[left] <= array[pivot]:
            left += 1
        # 피벗보다 작은 데이터를 찾을 때까지 반복
        while right > start and array[right] >= array[pivot]:
            right -= 1
        if left > right:  # 엇갈렸다면 작은 데이터와 피벗을 교체
            array[right], array[pivot] = array[pivot], array[right]
        else:  # 엇갈리지 않았다면 작은 데이터와 큰 데이터를 교체
            array[left], array[right] = array[right], array[left]
    # 분할 이후 왼쪽 부분과 오른쪽 부분에서 각각 정렬 수행
    quick_sort(array, start, right - 1)
    quick_sort(array, right + 1, end)


def quick_sort2(array):
    # 리스트가 하나 이하의 원소만을 담고 있다면 종료
    if len(array) <= 1:
        return array

    pivot = array[0]  # 피벗은 첫 번째 원소
    tail = array[1:]  # 피벗을 제외한 리스트
```

```
    left_side = [x for x in tail if x <= pivot]  # 분할된 왼쪽 부분
    right_side = [x for x in tail if x > pivot]  # 분할된 오른쪽 부분

    # 분할 이후 왼쪽 부분과 오른쪽 부분에서 각각 정렬을 수행하고, 전체 리스트를 반환
    return quick_sort2(left_side) + [pivot] + quick_sort2(right_side)


rnd_array = [random.randint(1, 1000000) for i in range(500000)]
array = copy.deepcopy(rnd_array)
st = time.time()
print(quicksort(array)[:10])
print(time.time() - st)

array = copy.deepcopy(rnd_array)
st = time.time()
quick_sort(array, 0, len(array) - 1)
print(array[:10])
print(time.time() - st)

array = copy.deepcopy(rnd_array)
st = time.time()
print(quick_sort2(array)[:10])
print(time.time() - st)

array = copy.deepcopy(rnd_array)
st = time.time()
array.sort()
print(array[:10])
print(time.time() - st)
```

시간복잡도

이상적인 경우

- 분할이 절반씩 일어난다면 전체 연산 횟수로 ***O(NlogN)*** 를 기대할 수 있다
  - **너비 X 높이** = $N \times logN = NlogN$



최악의 경우

- 모든 분할이 1 / n-1로 일어나는 경우 최대 ***O(N^2)*** 의 시간복잡도를 가진다.
- 해결 방법
  - 피벗을 난수로 잡는다
    - 이때 최악의 pivot을 선택될 확률은 1/N
    - 매 재귀마다 그렇게 선택될 확률은 (1/N^2)
  - Median-of-Three Partition 사용
    - 처음, 가운데, 끝 값을 비교해 가운데 값을 피봇으로 잡아준다.

왜 퀵소트인가?

- 최악의 경우 ***O(N^2)*** 인 정렬 알고리즘인 퀵소트를 왜 사용할까?
- 다른 ***O(nlogn)*** 의 시간복잡도를 가지는 정렬을 사용하지 않는 이유는?
- 퀵소트가 가진 장점 (위키 원문 참조)

> 퀵 정렬의 내부 루프는 대부분의 컴퓨터 아키텍처에서 효율적으로 작동하도록 설계되어 있고(그 이유는 메모리 참조가 지역화되어 있기 때문에 CPU 캐시의 히트율이 높아지기 때문이다.), 대부분의 실질적인 데이터를 정렬할 때 제곱 시간이 걸릴 확률이 거의 없도록 알고리즘을 설계하는 것이 가능하다. 또한 매 단계에서 적어도 1개의 원소가 자기 자리를 찾게 되므로 이후 정렬할 개수가 줄어든다. 때문에 일반적인 경우 퀵 정렬은 다른 O(n log n) 알고리즘에 비해 훨씬 빠르게 동작한다.

문제

수 정렬하기 2 : https://www.acmicpc.net/problem/2751

참고자료

위키 : https://ko.wikipedia.org/wiki/퀵_정렬

과정 이미지 : https://guiyum.tistory.com/66

퀵소트 강의 영상 : https://www.youtube.com/watch?v=EuJSDghD4z8&list=PLVsNizTWUw7H9_of5YCB0FmsSc-K44y81&index=23

강의 정리 : https://freedeveloper.tistory.com/377