2. Quick Sort 알고리즘

속성	@September 5, 2021 6:25 PM
작성자	현빈 김현빈

이론적으로 수행 시간을 따져보면 **Quick Sort 알고리즘의 수행시간은 O(n²)으로 별로 좋지 않다.** **하지만 실제로 구현하여 돌려보면 어떤 알고리즘보다 빠르다.** 이 알고리즘은 k번째로 작은 값을 찾는 quick selection 알고리즘과 동일한 작동 방식이다. 이는 순서대로 모든 숫자들을 나열한다. 상수 시간만큼 투자하여 피벗 숫자를 먼저 정한 다음에, 피벗보다 작은 값들은 S그룹에 모으고, 같은 값들은 M그룹에 모으고, 크면 L에 집어 넣는다. 이 알고리즘은 S와 L 그룹을 각각 다시 quick sort를 다시 불러서 정렬해준다. 하나를 버릴 수 없고 S와 L 모두 수행해주어야 한다.

```
def quick_sort(A):
if A 리스트 원소의 개수 <= 1 : return A
else:
  pivot = A[0]
  S,M,L = [], [], []
  for x in A:
    if pivot > x:
      S.append(x)
    elif pivot < x:
      L.append(x)
    else:
      M.append(x)
  return quick_sort(s) + M[] + quick_sort(L)
```

⇒ not in-place 알고리즘이다. A의 n개의 숫자가 S,M,L로 다 카피 되어 입력 크기에 비례하는 메모리를 추가적으로 사용했기 때문이다.

하지만 Stable 한 알고리즘이다. A에 들어있는 x를 왼쪽부터 차례대로 피벗과 비교하기 때문에, 같은 값들이 M에 들어가게 되면, append로 인해 차례대로 리스트에 정렬된다. 결국에는 그 순서를 그대로 유지하는 것이다.

수행 시간: n-1번 비교하여 S,M,L로 나눈 후 재귀적으로 이것을 반복한다.

T(n) = T(S[]) + T (L[]) + cn(=비교 횟수)

최악의 경우: 한쪽 그룹에 모든 원소들이 들어가는 것.

⇒ T(n) = T(n-1) + cn ⇒ O(n²) = 공식적으로 이론적인 수행 시간.

최고의 경우: S와 L에 원소들이 균등하게 나눠진 경우.

⇒ T(n) = T(n/2) + T(n/2) + cn = 2T(n/2) + cn = O(nlog₂n)

worst case는 n²이지만 평균적인 케이스는 최고의 경우의 수행 시간과 같다. 따라서 최악의 경우를 거의 생각하지 않아도 된다.

In-place Quick sort algorithm

```
def quick_sort(A, first, last):
  # A[first] ... A[last]를 quick sort 해라
  # 호출 시 quick_sort(A, 0, len(A)-1)

pivot = A[first] # 첫 번째 수를 피벗으로 지정한다.
# 왼쪽에서 오른쪽으로 가면서 우리는 피벗과 숫자를 비교하며 그룹화를 한다.
left = first + 1 # 왼쪽 - 시작 원소 위치
right = last # 오른쪽 - 마지막 원소 위치

  while left <= right: # 왼쪽과 오른쪽이 엇갈릴 때 종료
    while left <= last & A[left] < pivot:
       left += 1 # A[left]는 왼쪽에서 오른쪽으로 가다가 피벗 숫자보다 크면 멈춘다.
    while A[right] > p:
       right -= 1 # A[right]은 오른쪽에서 왼쪽으로 가다가 피벗보다 작으면 멈춘다.

    if left <=right # 아직 왼쪽과 오른쪽이 엇갈리지 않았을 때
  # if A[left] >= A[right]: 이런식으로 사용하지 않는 이유는 왼쪽의 원소와 오른쪽의 원소의 크기는 왼쪽은 피벗보다 크고, 오른쪽은 피벗보다 작은 것이 위에서 정해졌기
      A[left] -> A[right] #swap
      A[right] -> A[left] #swap
      left += 1
      right -= 1
```

```
#피벗을 S와 L사이로 옮기기
A[first] -> A[right] #swap
A[right] -> A[first] #swap

# 이전까지는 작은 그룹, 같은 그룹, 큰 그룹을 묶어준 것이였다.
# 이를 다시 재귀적으로 작은 그룹과 큰 그룹으로 실행해야되니
auick_sort(A, first, right-1)
puick_sort(A, left, last)
#를 해주어야 된다. 이 알고리즘은 In-place이다.
```