

버블/선택/합병/퀵 정  
렬

# 주요 내용

- 버블 정렬
- 선택 정렬
- 합병 정렬
- 퀵 정렬

## 정렬

- 모음 객체에 포함된 항목들을 특정 크기 기준에 따라 오름차순 또는 내림차순으로 순서대로 위치시키는 과정
- 문자열들로 이루어진 리스트의 항목을 알파벳 순서대로 정렬하기
- 도시들의 리스트를 인구, 면적, 또는 우편번호 등으로 정렬하기
- 정렬 활용 예제: 어구전철 관계 확인, 이진탐색 등

## 정렬 알고리즘

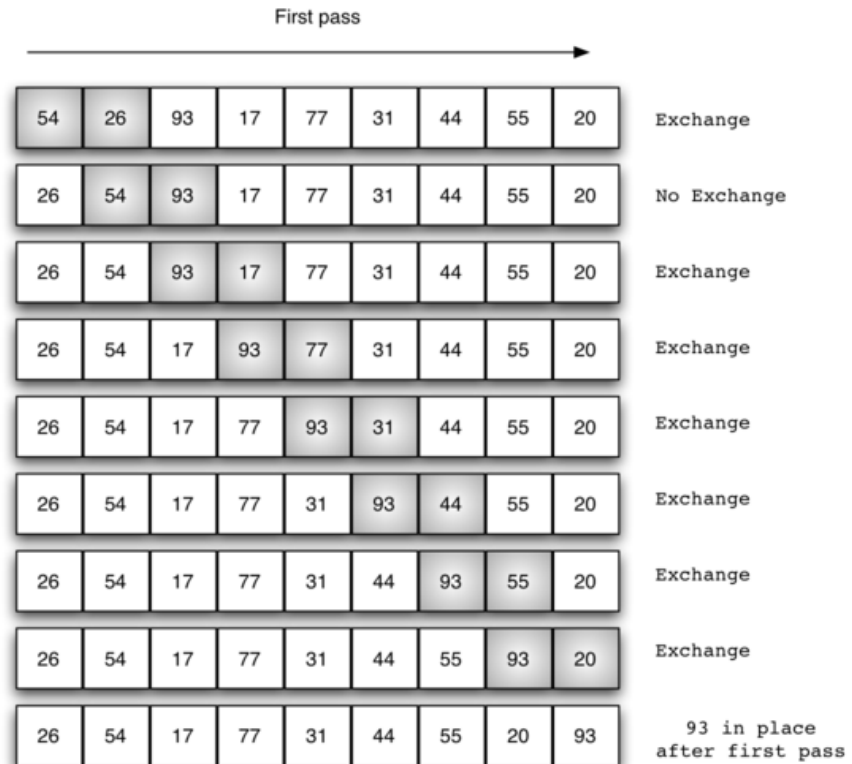
- 다양한 종류의 정렬 알고리즘 존재
- 모든 경우에 항상 효율적으로 작동하는 정렬 알고리즘은 없음.
- 정렬 알고리즘의 시간복잡도 분석에 사용되는 기본 단위연산
  - 두 값의 크기 비교: 두 항목의 크기를 서로 비교할 수 있어야 함.
  - 두 항목의 자리 교환: 예를 들어, 보다 작은 값을 보다 큰 값보다 왼쪽에 위치하도록 해야 함.

버블 정렬

## 패스

- 버블 정렬 bubble sort 알고리즘은 패스 pass로 구성됨.
- 패스: 연속된 위치의 두 수를 비교하여 작은 항목은 왼쪽으로, 큰 항목은 오른쪽으로 자리를 바꾸는 일을 리스트의 맨 왼쪽에서 시작하여 오른쪽 끝까지 진행
- 하나의 패스가 완료되면 패스 과정에서 확인된 가장 큰 항목이 리스트의 맨 오른쪽에 위치
- 가장 큰 수가 위치한 곳을 제외한 왼쪽 구간을 대상으로 패스 반복
- 가장 큰 항목은 리스트의 맨 오른쪽에 위치하고 모든 항목이 크기 순서대로 정렬된 리스트 생성

# 첫번째 패스



## 버블 정렬 알고리즘 구현

```
def bubble_sort(a_list):  
    # 패스. 자리교환 인덱스 끝자리 지정  
    for a_pass in range(len(a_list) - 1, 0, -1):  
        # 크기비교 및 자리교환  
        for j in range(a_pass):  
            if a_list[j] > a_list[j + 1]:  
                a_list[j], a_list[j+1] = a_list[j + 1], a_list[j]
```



## 버블 정렬 시간복잡도 분석

패 스	비교 횟수
1	$n - 1$
2	$n - 2$
3	$n - 3$
$\vdots$	$\vdots$
$n - 1$	1

- 항목 비교

$$1 + 2 + \cdots + (n - 1) = \frac{n(n - 1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n \in O(n^2)$$

- 자리교환: 하나의 패스 동안 최소 0, 최대 비교횟수와 동일하게 발생. 평균적으로 비교횟수의 절반정도 발생.
- 최선, 최악, 평균 시간복잡도 모두:  $O(n^2)$

## 버블 정렬 조기종료

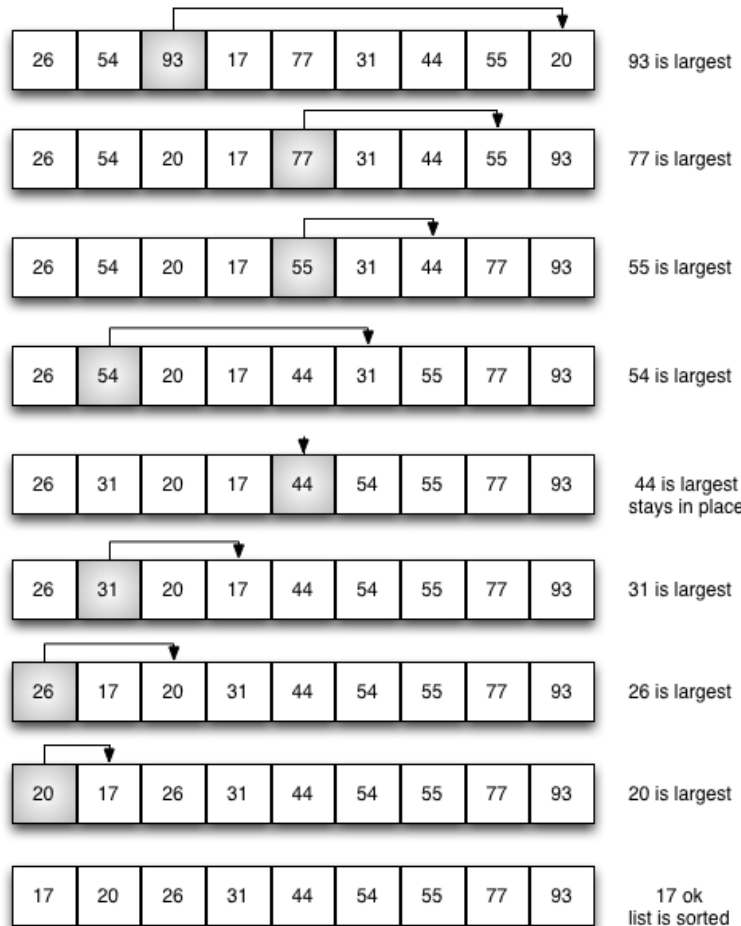
```
def bubble_sort_early_stop(a_list):  
    for a_pass in range(len(a_list) - 1, 0, -1):  
        exchanges = False # 패스 기간 내 자리교환 발생 여부 저장  
        for j in range(a_pass):  
            if a_list[j] > a_list[j + 1]:  
                a_list[j], a_list[j+1] = a_list[j + 1], a_list[j]  
                exchanges = True  
  
        # 자리교환이 발생하지 않은 경우 조기종료  
        if not exchanges:  
            print("조기종료!")  
            break
```

## 퀴즈

리스트 [19, 1, 9, 7, 3, 10, 13, 15, 8, 12] 에 대해 버블 정렬을 진행할 때 세 번의 패스가 완성된 후의 리스트는 어떤 모습인가?

# 선택 정렬

- 작동과정은 기본적으로 버블 정렬과 동일
- 자리교환을 바로 실행하는 게 아니라 패스 별로 최댓값을 확인한 다음에 최종적으로 필요에 따라 딱 한 번 자리교환 실행



## 선택 정렬 알고리즘 구현

```
def selection_sort(a_list):  
    for a_pass in range(len(a_list)-1, 0, -1):  
        max_idx = 0  
        for j in range(a_pass):  
            if a_list[j] > a_list[max_idx]:  
                max_idx = j  
        # a_pass 인덱스 이전까지 찾은 최대값과  
        # a_pass 위치의 값 자리교환 여부 판단  
        if a_list[max_idx] > a_list[a_pass]:  
            a_list[max_idx], a_list[a_pass] = a_list[a_pass], a_list[max_idx]
```

## 선택 정렬 시간복잡도 분석

- 크기비교: 버블 정렬의 경우와 동일
- 자리교환 횟수: 최대  $(n - 1)$  번 발생
- 버블 정렬보다 조금 빠름.

## 퀴즈

리스트 [11, 7, 12, 14, 19, 1, 6, 18, 8, 20] 에 대해 선택 정렬을 진행할 때 세 번의 패스가 완성된 후의 리스트는 어떤 모습인가?

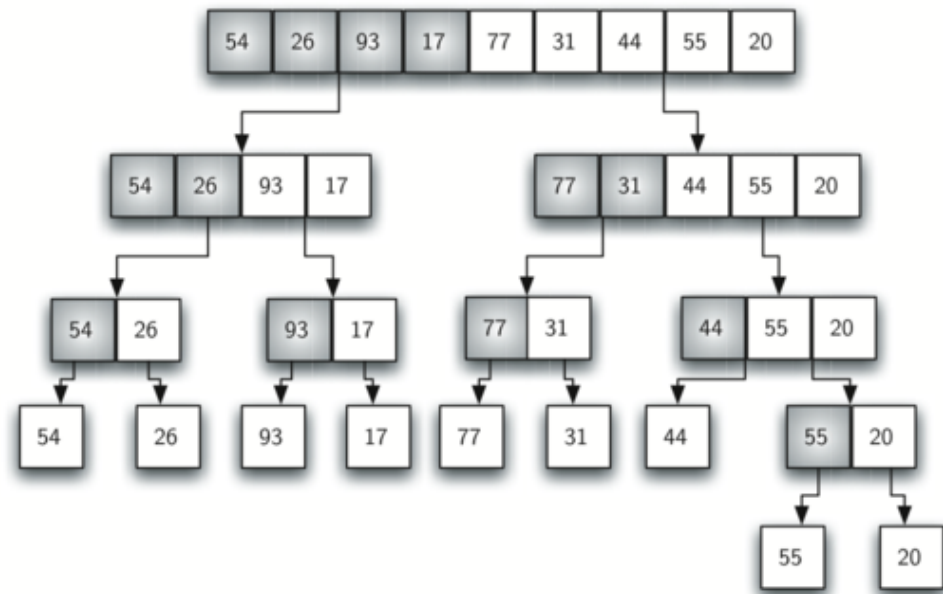


# 합병 정렬

- 분할 정복 기법을 이용
- **합병 정렬** merge sort에 사용되는 분할과 정복
  - 분할: 리스트의 길이가 1이 될 때까지 반복적으로 이등분한다.
  - 정복: 길이가 작은 두 개의 리스트를 합병해서 보다 큰 길이의 리스트를 생성한다. 합병 과정에서 항목들을 크기 순으로 정렬한다.

## 분할 과정

```
def merge_sort(a_list):  
    # 분할 과정: 이등분 반복  
    if len(a_list) > 1:  
        print("분할:", a_list)  
  
        mid = len(a_list) // 2  
        left_half = a_list[:mid]  
        right_half = a_list[mid:]  
  
        # 재귀 호출  
        merge_sort(left_half)  
        merge_sort(right_half)  
  
        # 합병 과정: 작은 리스트 두 개 합병하면서 정렬  
        merge(a_list, left_half, right_half)  
        print("합병:", a_list)
```

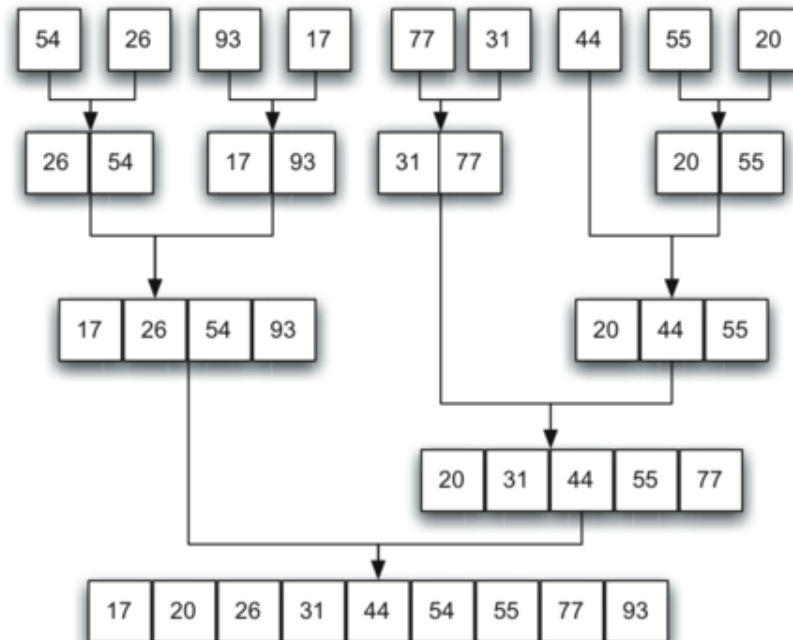


## 합병 과정

```
def merge(a_list, left_half, right_half):
    i, j, k = 0, 0, 0
    while i < len(left_half) and j < len(right_half):
        if left_half[i] <= right_half[j]:
            a_list[k] = left_half[i]
            i = i + 1
        else:
            a_list[k] = right_half[j]
            j = j + 1
        k = k + 1

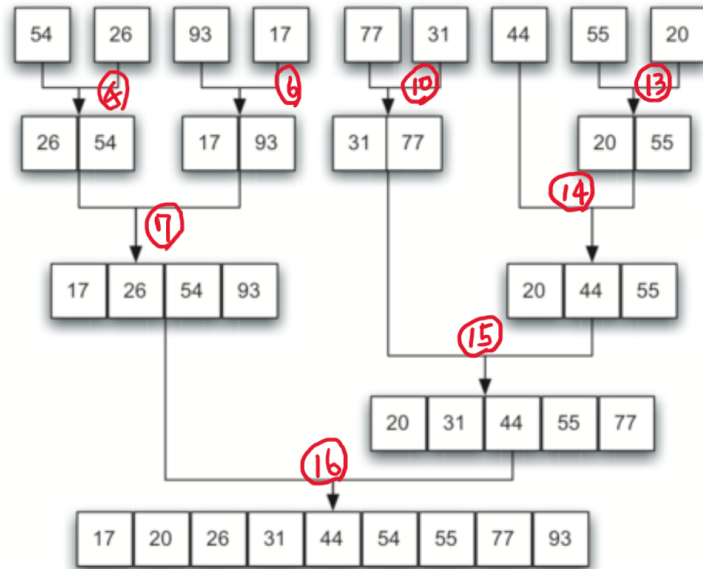
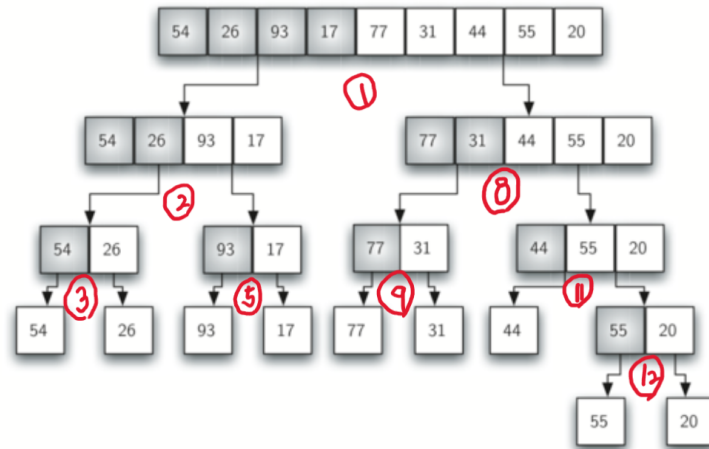
    while i < len(left_half):
        a_list[k] = left_half[i]
        i = i + 1
        k = k + 1

    while j < len(right_half):
        a_list[k] = right_half[j]
        j = j + 1
        k = k + 1
```



## 분할과 합병이 재귀적으로 발생하는 순서

```
>>> a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
>>> merge_sort(a_list)
재귀 분할 : [54, 26, 93, 17, 77, 31, 44, 55, 20]
재귀 분할 : [54, 26, 93, 17]
재귀 분할 : [54, 26]
합병 정렬 : [26, 54]
재귀 분할 : [93, 17]
합병 정렬 : [17, 93]
합병 정렬 : [17, 26, 54, 93]
재귀 분할 : [77, 31, 44, 55, 20]
재귀 분할 : [77, 31]
합병 정렬 : [31, 77]
재귀 분할 : [44, 55, 20]
재귀 분할 : [55, 20]
합병 정렬 : [20, 55]
합병 정렬 : [20, 44, 55]
합병 정렬 : [20, 31, 44, 55, 77]
합병 정렬 : [17, 20, 26, 31, 44, 54, 55, 77, 93]
```



# 합병 정렬 시간복잡도 분석

- 입력 크기: 리스트의 길이  $n$
- 분할 과정
  - 비교연산이나 위치 이동이 전혀 없음.
- 합병 과정
  - 합병과정을 그래프로 간주하면 높이가  $\text{int}(\log(n)) + 1$  인 가지가 두 개로 갈라지는 이진 트리임.
  - 하나의 레벨(나무의 동일한 높이)에서 합병이 여러 번 발생하는 데 각각의 합병 과정에서 발생하는 항목 비교와 위치 이동을 다 합치면 최대  $2*n$  번임.
  - 따라서 합병 과정에서 발생하는 항목 비교와 위치 이동은 최악의 경우  $O(n \log n)$ 의 시간복잡도를 가짐.



## 슬라이싱과 추가 메모리 사용

- 슬라이싱을 이용: 실제 시간복잡도는  $O(n^2 \log n)$
- 길이가  $k$ 인 구간을 슬라이싱 하는 알고리즘의 시간복잡도:  $O(k)$
- 버블 정렬과 선택 정렬: 추가 메모리 사용하지 않음

합병 정렬 연습

54	26	93	17	77	31	44	55	20
----	----	----	----	----	----	----	----	----

5	3	8	4	9	1	6	2	7
---	---	---	---	---	---	---	---	---

퀵 정렬

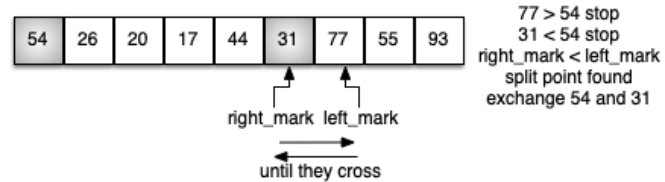
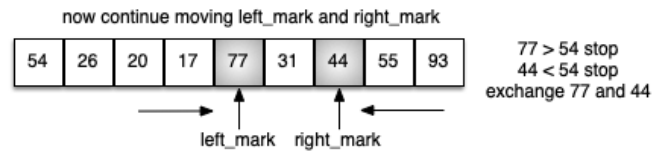
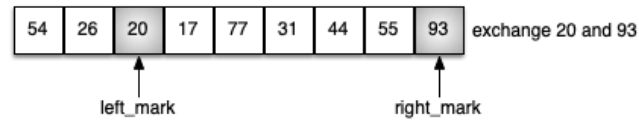
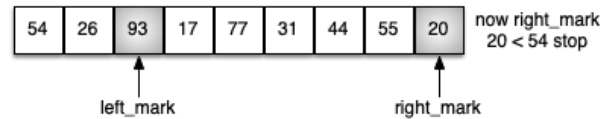
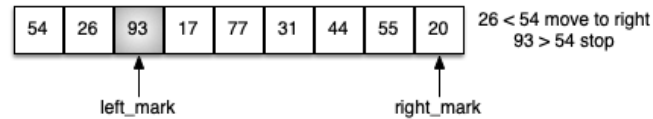
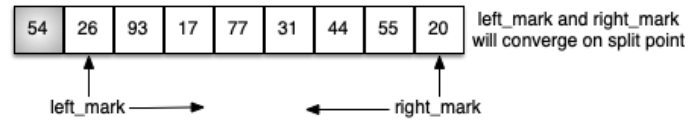
## 피벗 지정

- 리스트의 임의의 값을 사용 가능
- 여기서는 맨 왼쪽에 위치한 값을 사용
- 경우에 따라 양끝과 중앙에 위치한 세 값의 중앙값을 사용
- 오른쪽 맨 끝, 또는 중앙에 위치한 값 등도 사용
- 알고리즘의 성능 차이는 기본적으로 없으며, 입력 사례에 따라 달라짐.



## 분할과 정복

- 분할과 정복을 동시에 진행
- 한 번의 분할과정을 통해 두 개의 보다 작은 리스트로 분할
- 분할 과정 중에 정복을 동시 실행
- 쪼개진 작은 구간에 대해 동일 과정 반복 적용





## 분할과 정복의 한 단계

```
def partition(a_list, first, last):
    pivot_val = a_list[first]    # 피벗
    left_mark = first + 1        # 탐색 구간 시작
    right_mark = last            # 탐색 구간 끝
    done = False                 # 탐색 종료여부 확인

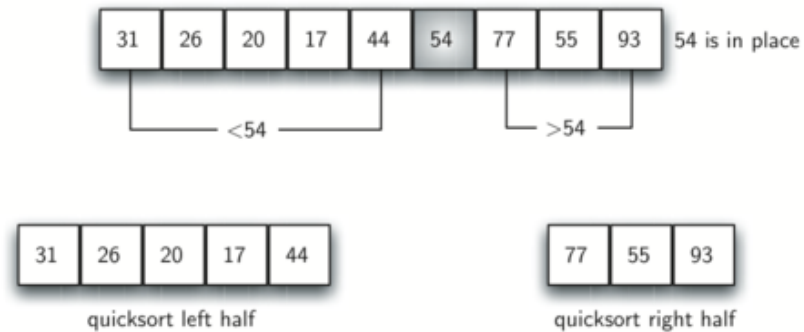
    while not done:
        while left_mark <= right_mark and a_list[left_mark] < pivot_val:
            left_mark = left_mark + 1
        while left_mark <= right_mark and a_list[right_mark] >= pivot_val:
            right_mark = right_mark - 1
        # 자리 교환
        if right_mark < left_mark:
            done = True
        else:
            a_list[left_mark], a_list[right_mark] = a_list[right_mark],
a_list[left_mark]

    # 피벗 자리 교환
    a_list[first], a_list[right_mark] = a_list[right_mark], a_list[first]

    # 피벗 위치 반환
    return right_mark
```

## 재귀 적용

```
def quick_sort_helper(a_list, first, last):  
    if first < last:  
        split = partition(a_list, first, last)  
  
        quick_sort_helper(a_list, first, split - 1)  
        quick_sort_helper(a_list, split + 1, last)
```



## 재귀 함수 구현

```
def quick_sort(a_list):  
    quick_sort_helper(a_list, 0, len(a_list) - 1)
```

## 퀵 정렬 시간복잡도 분석

- 이상적인 경우
  - 리스트를 거의 정확하게 이등분
  - 이 경우 분할 횟수:  $\log n$
  - 한 번 분할할 때마다 피벗과 나머지 값들이 비교되고 필요에 따라 자리교환이 발생:  $O(n)$
  - 최선 시간복잡도:  $O(n \log n)$
- 최악의 경우
  - 분할이 한쪽으로 쏠림.
  - 예를 들어, 거의 정렬이 되어있는 경우: 피벗을 기준으로 1대  $(n-1)$  개의 부분 리스트로 분할될 수 있음
  - 최대  $n$  번에 가까운 분할이 필요
  - 최악 시간복잡도:  $O(n^2)$

## 합병 정렬 대 퀵 정렬

- 합병 정렬
  - 퀵 정렬에 비해 시간복잡도 측면에서 이론적으로 좋음
  - 하지만 공간복잡도와 자리 교환 횟수 측면에서 비효율적
- 퀵 정렬이 대부분의 운영체제와 프로그래밍언어에서 최적화된 형식으로 활용됨

## 정렬 알고리즘 시간복잡도 비교

알고리즘	최선	평균	최악	모의실험
버블 정렬	$n^2$	$n^2$	$n^2$	9.129473
버블 정렬(조기 종료)	$n^2$	$n^2$	$n^2$	9.414862
선택 정렬	$n^2$	$n^2$	$n^2$	4.365451
합병 정렬	$n \log n$	$n \log n$	$n \log n$	0.023475
퀵 정렬	$n \log n$	$n \log n$	$n^2$	0.056168

퀴 정렬 연습

54	26	93	17	77	31	44	55	20
----	----	----	----	----	----	----	----	----



5	3	8	4	9	1	6	2	7
---	---	---	---	---	---	---	---	---

