자료구조 실습

09/24

김민서 (imkmsh@khu.ac.kr)

목차

- 실습 코드 충돌 방지 설정
- selection_sort
- bubble_sort
- insertion_sort
- quick_sort
- ~~merge_sort~~
- 생각해 볼 만한 QUIZ
- 샘플 크기에 따른 시간 차이 확인

김민서 (imkmsh@khu.ac.kr)

실습 코드 충돌 방지 설정

README.md 파일 참고

- 아직 clone 받기 전 or 그냥 초기화하고 싶다면

저장소에 `.gitattributes` 파일을 추가해 `answer.*` 파일은 항상 학생 본인 실습 코드가 우선 적용되도록 설정했습니다.

따라서 각자 한 번씩 아래 명령어를 실행해야 합니다.

```
git config merge.ours.driver true
```

- 이 설정을 해두면 `pull` 시 충돌 없이 본인 답안이 유지됩니다.

김민서 (imkmsh@khu.ac.kr)

실습 코드 충돌 방지 설정

README.md 파일 참고

- ⚠️ 이미 pull에서 충돌이 발생한 경우

`.gitattributes` 파일이 반영되지 않아 `pull` 도중 에러가 날 수 있습니다.

이 경우 내 답안을 `stash`로 임시 저장한 뒤 다시 `pull` 하세요.

```
git stash push -m "my work backup" --include-untracked
git pull --rebase origin main
git stash pop
```

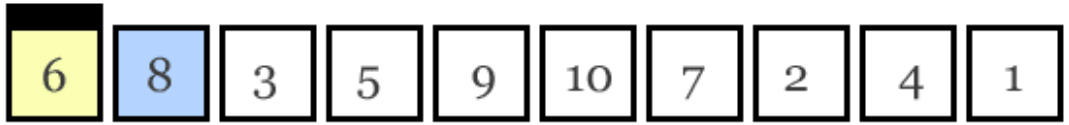
그 후에 `config`를 실행합니다.

```
git config merge.ours.driver true
```

김민서 (imkmsh@khu.ac.kr)

Selection Sort

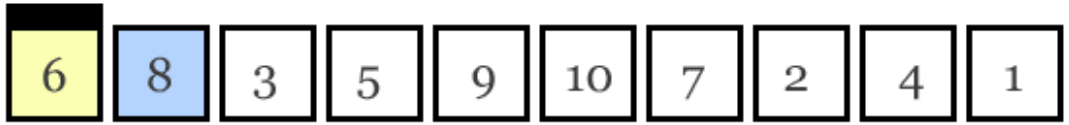
selection_sort




```python
def selection_sort(arr):  # 선택 정렬 함수 정의
    """
    선택 정렬 (Selection Sort)
    - 아이디어: 매 단계 i에서 i..n-1 범위에서 '최소값'의 위치(min_idx)를 찾아 i와 교환
    - 불변식(invariant): 반복문의 단계가 끝날 때마다 arr[0..i]은 오름차순으로 정렬되어 있고,
                         이 구간에는 입력 배열의 가장 작은 i+1개의 원소가 위치한다.
    - 시간복잡도: O(n^2) (최선/평균/최악 모두)
    """
```

김민서 (imkmsh@khu.ac.kr)

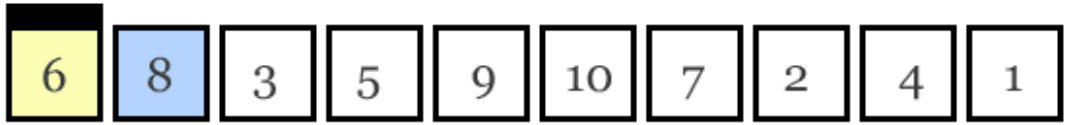
selection_sort




```python
n = len(arr)  # 배열의 길이 n을 저장

for i in range(NEED_TO_SOLVE):  # 0 ~ n-2까지 반복 (마지막은 자동 정렬됨)
    min_idx = NEED_TO_SOLVE  # i번째 원소를 현재 최소값으로 가정

    for j in range(NEED_TO_SOLVE, NEED_TO_SOLVE):  # i 이후 구간에서 최소값 탐색
        if arr[NEED_TO_SOLVE] < arr[NEED_TO_SOLVE]:  # 더 작은 값 발견 시
            NEED_TO_SOLVE = NEED_TO_SOLVE  # 최소값 인덱스 갱신

    arr[NEED_TO_SOLVE], arr[NEED_TO_SOLVE] = arr[NEED_TO_SOLVE], arr
    [NEED_TO_SOLVE]  # 최소값을 i번째 위치로 교환

return arr  # 정렬된 배열 반환
```

김민서 (imkmsh@khu.ac.kr)

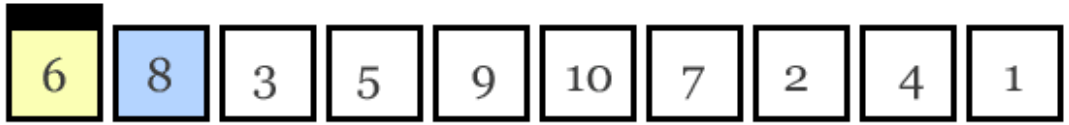
selection_sort




```python
n = len(arr)  # 배열의 길이 n을 저장

for i in range(n - 1):  # 0 ~ n-2까지 반복 (마지막은 자동 정렬됨)
    min_idx = i  # i번째 원소를 현재 최소값으로 가정

    for j in range(NEED_TO_SOLVE, NEED_TO_SOLVE):  # i 이후 구간에서 최소값 탐색
        if arr[NEED_TO_SOLVE] < arr[NEED_TO_SOLVE]:  # 더 작은 값 발견 시
            NEED_TO_SOLVE = NEED_TO_SOLVE  # 최소값 인덱스 갱신

    arr[NEED_TO_SOLVE], arr[NEED_TO_SOLVE] = arr[NEED_TO_SOLVE], arr
    [NEED_TO_SOLVE]  # 최소값을 i번째 위치로 교환

return arr  # 정렬된 배열 반환
```

김민서 (imkmsh@khu.ac.kr)

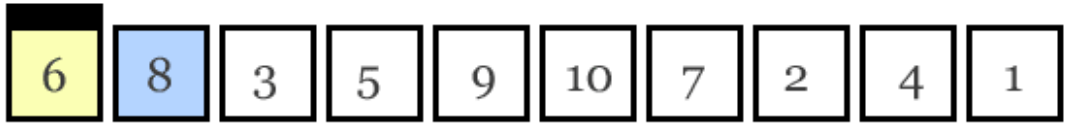
selection_sort




```python
n = len(arr)  # 배열의 길이 n을 저장

for i in range(n - 1):  # 0 ~ n-2까지 반복 (마지막은 자동 정렬됨)
    min_idx = i  # i번째 원소를 현재 최소값으로 가정

    for j in range(NEED_TO_SOLVE, NEED_TO_SOLVE):  # i 이후 구간에서 최소값 탐색
        if arr[NEED_TO_SOLVE] < arr[NEED_TO_SOLVE]:  # 더 작은 값 발견 시
            NEED_TO_SOLVE = NEED_TO_SOLVE  # 최소값 인덱스 갱신

    arr[NEED_TO_SOLVE], arr[NEED_TO_SOLVE] = arr[NEED_TO_SOLVE], arr
    [NEED_TO_SOLVE]  # 최소값을 i번째 위치로 교환

return arr  # 정렬된 배열 반환
```

김민서 (imkmsh@khu.ac.kr)

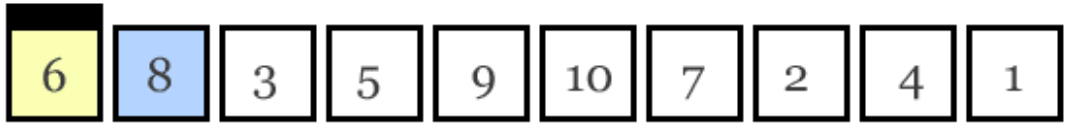
selection_sort




```python
n = len(arr)  # 배열의 길이 n을 저장

for i in range(n - 1):  # 0 ~ n-2까지 반복 (마지막은 자동 정렬됨)
    min_idx = i  # i번째 원소를 현재 최소값으로 가정

    for j in range(i + 1, n):  # i 이후 구간에서 최소값 탐색
        if arr[j] < arr[min_idx]:  # 더 작은 값 발견 시
            min_idx = j  # 최소값 인덱스 갱신

    arr[NEED_TO_SOLVE], arr[NEED_TO_SOLVE] = arr[NEED_TO_SOLVE], arr
    [NEED_TO_SOLVE]  # 최소값을 i번째 위치로 교환

return arr  # 정렬된 배열 반환
```

김민서 (imkmsh@khu.ac.kr)

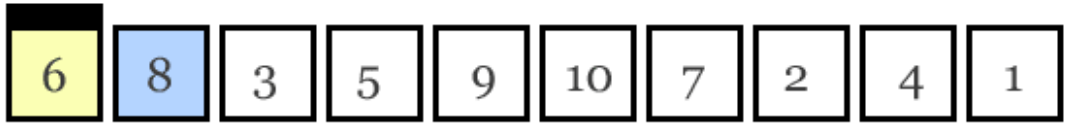
selection_sort




```python
n = len(arr)  # 배열의 길이 n을 저장

for i in range(n - 1):  # 0 ~ n-2까지 반복 (마지막은 자동 정렬됨)
    min_idx = i  # i번째 원소를 현재 최소값으로 가정

    for j in range(i + 1, n):  # i 이후 구간에서 최소값 탐색
        if arr[j] < arr[min_idx]:  # 더 작은 값 발견 시
            min_idx = j  # 최소값 인덱스 갱신

    arr[NEED_TO_SOLVE], arr[NEED_TO_SOLVE] = arr[NEED_TO_SOLVE], arr
    [NEED_TO_SOLVE]  # 최소값을 i번째 위치로 교환

return arr  # 정렬된 배열 반환
```

김민서 (imkmsh@khu.ac.kr)

selection_sort




```python
n = len(arr)  # 배열의 길이 n을 저장

for i in range(n - 1):  # 0 ~ n-2까지 반복 (마지막은 자동 정렬됨)
    min_idx = i  # i번째 원소를 현재 최소값으로 가정

    for j in range(i + 1, n):  # i 이후 구간에서 최소값 탐색
        if arr[j] < arr[min_idx]:  # 더 작은 값 발견 시
            min_idx = j  # 최소값 인덱스 갱신

    arr[i], arr[min_idx] = arr[min_idx], arr[i]  # 최소값을 i번째 위치로 교환

return arr  # 정렬된 배열 반환
```

김민서 (imkmsh@khu.ac.kr)

Bubble Sort

bubble_sort


```python
def bubble_sort(arr):  # 버블 정렬 함수 정의
    """
    버블 정렬 (Bubble Sort)
    - 아이디어: 이웃한 두 원소(arr[j], arr[j+1])를 비교하여 '큰 값'을 오른쪽으로 보내는 과정을
                맨 끝까지 반복하여, 매 단계마다 '가장 큰 값'이 배열의 끝에 '거품처럼' 올라가도록
                함.
    - 최적화: 한 바퀴에서 교환이 한 번도 일어나지 않으면 이미 정렬된 상태 → 조기 종료 가능.
    - 시간복잡도: O(n^2) (최악/평균), 최선은 O(n) (이미 정렬되어 있고 스왑 없음)
    """
```

bubble_sort


```
n = len(arr)  # 배열의 길이 n을 저장

for i in range(NEED_TO_SOLVE):  # 0 ~ n-2까지 반복 (마지막은 자동 정렬됨)
    min_idx = NEED_TO_SOLVE  # i번째 원소를 현재 최소값으로 가정

    for j in range(NEED_TO_SOLVE, NEED_TO_SOLVE):  # i 이후 구간에서 최소값 탐색
        if arr[NEED_TO_SOLVE] < arr[NEED_TO_SOLVE]:  # 더 작은 값 발견 시
            NEED_TO_SOLVE = NEED_TO_SOLVE  # 최소값 인덱스 갱신

    arr[NEED_TO_SOLVE], arr[NEED_TO_SOLVE] = arr[NEED_TO_SOLVE], arr
    [NEED_TO_SOLVE]  # 최소값을 i번째 위치로 교환

return arr  # 정렬된 배열 반환
```

김민서 (imkmsh@khu.ac.kr)

bubble_sort


```python
n = len(arr)  # 배열의 길이 n을 저장

for i in range(NEED_TO_SOLVE):  # 0 ~ n-2까지 반복
    for j in range(NEED_TO_SOLVE):  # 맨 끝 i개는 이미 정렬되었으므로 제외
        if arr[NEED_TO_SOLVE] > arr[NEED_TO_SOLVE]:  # 앞이 더 크면
            arr[NEED_TO_SOLVE], arr[NEED_TO_SOLVE] = arr[NEED_TO_SOLVE],
            arr[NEED_TO_SOLVE]  # 두 값을 교환

return arr  # 정렬된 배열 반환
```

김민서 (imkmsh@khu.ac.kr)

bubble_sort


```
n = len(arr)  # 배열의 길이 n을 저장

for i in range(n - 1):  # 0 ~ n-2까지 반복
    for j in range(NEED_TO_SOLVE):  # 맨 끝 i개는 이미 정렬되었으므로 제외
        if arr[NEED_TO_SOLVE] > arr[NEED_TO_SOLVE]:  # 앞이 더 크면
            arr[NEED_TO_SOLVE], arr[NEED_TO_SOLVE] = arr[NEED_TO_SOLVE],
            arr[NEED_TO_SOLVE]  # 두 값을 교환

return arr  # 정렬된 배열 반환
```

김민서 (imkmsh@khu.ac.kr)

bubble_sort


```python
n = len(arr)  # 배열의 길이 n을 저장

for i in range(n - 1):  # 0 ~ n-2까지 반복
    for j in range(NEED_TO_SOLVE):  # 맨 끝 i개는 이미 정렬되었으므로 제외
        if arr[NEED_TO_SOLVE] > arr[NEED_TO_SOLVE]:  # 앞이 더 크면
            arr[NEED_TO_SOLVE], arr[NEED_TO_SOLVE] = arr[NEED_TO_SOLVE],
            arr[NEED_TO_SOLVE]  # 두 값을 교환

return arr  # 정렬된 배열 반환
```

김민서 (imkmsh@khu.ac.kr)

bubble_sort


```python
n = len(arr)  # 배열의 길이 n을 저장

for i in range(n - 1):  # 0 ~ n-2까지 반복
    for j in range(n - i - 1):  # 맨 끝 i개는 이미 정렬되었으므로 제외
        if arr[NEED_TO_SOLVE] > arr[NEED_TO_SOLVE]:  # 앞이 더 크면
            arr[NEED_TO_SOLVE], arr[NEED_TO_SOLVE] = arr[NEED_TO_SOLVE],
            arr[NEED_TO_SOLVE]  # 두 값을 교환

return arr  # 정렬된 배열 반환
```

김민서 (imkmsh@khu.ac.kr)

bubble_sort


```python
n = len(arr)  # 배열의 길이 n을 저장

for i in range(n - 1):  # 0 ~ n-2까지 반복
    for j in range(n - i - 1):  # 맨 끝 i개는 이미 정렬되었으므로 제외
        if arr[NEED_TO_SOLVE] > arr[NEED_TO_SOLVE]:  # 앞이 더 크면
            arr[NEED_TO_SOLVE], arr[NEED_TO_SOLVE] = arr[NEED_TO_SOLVE],
            arr[NEED_TO_SOLVE]  # 두 값을 교환

return arr  # 정렬된 배열 반환
```

김민서 (imkmsh@khu.ac.kr)

bubble_sort


```python
n = len(arr)  # 배열의 길이 n을 저장

for i in range(n - 1):  # 0 ~ n-2까지 반복
    for j in range(n - i - 1):  # 맨 끝 i개는 이미 정렬되었으므로 제외
        if arr[j] > arr[j + 1]:  # 앞이 더 크면
            arr[j], arr[j + 1] = arr[j + 1], arr[j]  # 두 값을 교환

return arr  # 정렬된 배열 반환
```

김민서 (imkmsh@khu.ac.kr)

Insertion Sort

insertion_sort

6 5 3 1 8 7 2 4

```python
def insertion_sort(arr):  # 삽입 정렬 함수 정의
    """
    삽입 정렬 (Insertion Sort)
    - 아이디어: i 번째 원소를 '키(key)'로 두고, 앞쪽 정렬 구간(arr[0..i-1])에서
                key가 들어갈 적절한 위치를 찾을 때까지 원소들을 오른쪽으로 한 칸씩 밀어낸 뒤 삽입.
    - 불변식(invariant): 각 단계 i가 끝나면 arr[0..i] 구간은 항상 정렬되어 있다.
    - 시간복잡도: 최악/평균 O(n^2), 최선 O(n) (이미 거의 정렬된 경우 유리)
    """
```

김민서 (imkmsh@khu.ac.kr)

insertion_sort

6 5 3 1 8 7 2 4

```
n = len(arr)  # 배열의 길이 n을 저장

for i in range(NEED_TO_SOLVE, NEED_TO_SOLVE):  # index 1의 원소부터 차례대로 삽입
    key = arr[NEED_TO_SOLVE]  # 현재 삽입할 값
    j = NEED_TO_SOLVE  # 정렬된 구간의 끝 인덱스

    while j >= 0 and arr[NEED_TO_SOLVE] > NEED_TO_SOLVE:  # key보다 큰 값은 오른쪽으로 밀기
        arr[NEED_TO_SOLVE] = arr[NEED_TO_SOLVE]  # 값 이동
        j -= 1  # 왼쪽으로 한 칸 이동

    arr[NEED_TO_SOLVE] = NEED_TO_SOLVE  # 올바른 위치에 key 삽입

return arr  # 정렬된 배열 반환
```

김민서 (imkmsh@khu.ac.kr)

insertion_sort

6 5 3 1 8 7 2 4

```
n = len(arr)  # 배열의 길이 n을 저장

for i in range(1, n):  # index 1의 원소부터 차례대로 삽입
    key = arr[i]  # 현재 삽입할 값
    j = i - 1  # 정렬된 구간의 끝 인덱스

    while j >= 0 and arr[NEED_TO_SOLVE] > NEED_TO_SOLVE:  # key보다 큰 값은 오른쪽으로 밀기
        arr[NEED_TO_SOLVE] = arr[NEED_TO_SOLVE]  # 값 이동
        j -= 1  # 왼쪽으로 한 칸 이동

    arr[NEED_TO_SOLVE] = NEED_TO_SOLVE  # 올바른 위치에 key 삽입

return arr  # 정렬된 배열 반환
```

김민서 (imkmsh@khu.ac.kr)

insertion_sort

6 5 3 1 8 7 2 4

```
n = len(arr)  # 배열의 길이 n을 저장

for i in range(1, n):  # index 1의 원소부터 차례대로 삽입
    key = arr[i]  # 현재 삽입할 값
    j = i - 1  # 정렬된 구간의 끝 인덱스

    while j >= 0 and arr[NEED_TO_SOLVE] > NEED_TO_SOLVE:  # key보다 큰 값은 오른쪽으로 밀기
        arr[NEED_TO_SOLVE] = arr[NEED_TO_SOLVE]  # 값 이동
        j -= 1  # 왼쪽으로 한 칸 이동

    arr[NEED_TO_SOLVE] = NEED_TO_SOLVE  # 올바른 위치에 key 삽입

return arr  # 정렬된 배열 반환
```

김민서 (imkmsh@khu.ac.kr)

insertion_sort

6 5 3 1 8 7 2 4

```
n = len(arr)  # 배열의 길이 n을 저장

for i in range(1, n):  # index 1의 원소부터 차례대로 삽입
    key = arr[i]  # 현재 삽입할 값
    j = i - 1  # 정렬된 구간의 끝 인덱스

    while j >= 0 and arr[j] > key:  # key보다 큰 값은 오른쪽으로 밀기
        arr[j + 1] = arr[j]  # 값 이동
        j -= 1  # 왼쪽으로 한 칸 이동

    arr[NEED_TO_SOLVE] = NEED_TO_SOLVE  # 올바른 위치에 key 삽입

return arr  # 정렬된 배열 반환
```

김민서 (imkmsh@khu.ac.kr)

insertion_sort

6 5 3 1 8 7 2 4

```
n = len(arr)  # 배열의 길이 n을 저장

for i in range(1, n):  # index 1의 원소부터 차례대로 삽입
    key = arr[i]  # 현재 삽입할 값
    j = i - 1  # 정렬된 구간의 끝 인덱스

    while j >= 0 and arr[j] > key:  # key보다 큰 값은 오른쪽으로 밀기
        arr[j + 1] = arr[j]  # 값 이동
        j -= 1  # 왼쪽으로 한 칸 이동

    arr[NEED_TO_SOLVE] = NEED_TO_SOLVE  # 올바른 위치에 key 삽입

return arr  # 정렬된 배열 반환
```

김민서 (imkmsh@khu.ac.kr)

insertion_sort

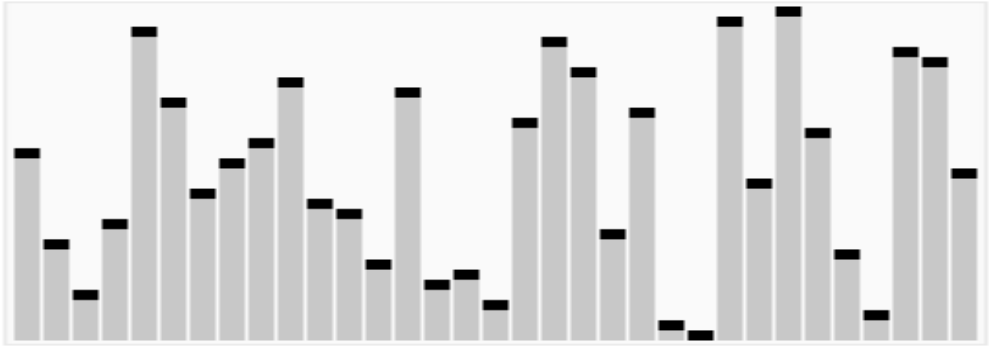
6 5 3 1 8 7 2 4

```
n = len(arr)  # 배열의 길이 n을 저장

for i in range(1, n):  # index 1의 원소부터 차례대로 삽입
    key = arr[i]  # 현재 삽입할 값
    j = i - 1  # 정렬된 구간의 끝 인덱스

    while j >= 0 and arr[j] > key:  # key보다 큰 값은 오른쪽으로 밀기
        arr[j + 1] = arr[j]  # 값 이동
        j -= 1  # 왼쪽으로 한 칸 이동

    arr[j + 1] = key  # 올바른 위치에 key 삽입

return arr  # 정렬된 배열 반환
```

김민서 (imkmsh@khu.ac.kr)

Quick Sort

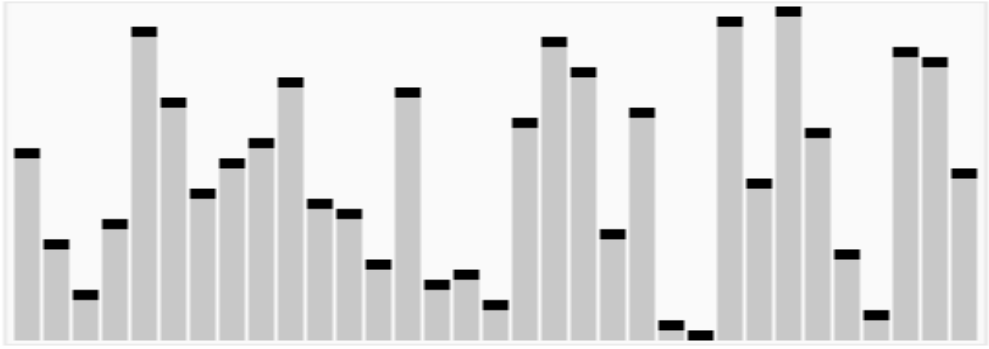
quick_sort



```python
def quick_sort(arr):  # 퀵 정렬 함수 정의
    """
    퀵 정렬 (Quick Sort)
    - 아이디어: 피벗을 기준으로 '작은 값 / 큰 값'으로 분할(partition) 후, 각 부분 배열을 재귀 정렬.
    - 시간복잡도: 평균 O(n log n), 최악 O(n^2) (이미 정렬된 입력에서 나쁜 피벗을 선택하면 발생 가능)
    """
```

김민서 (imkmsh@khu.ac.kr)

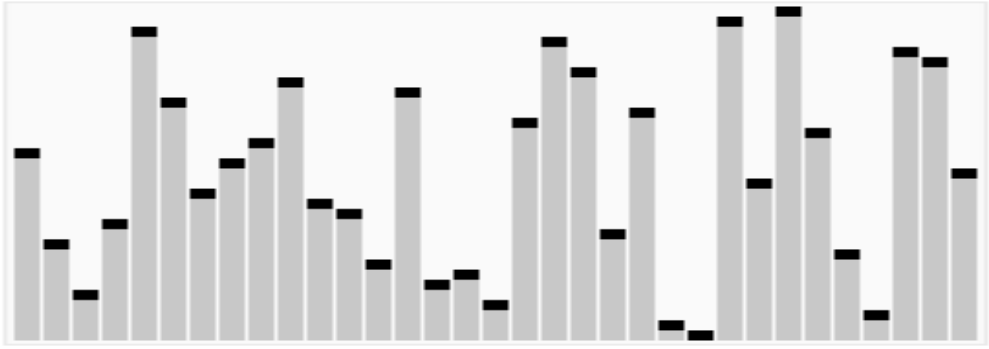
quick_sort



```python
if len(arr) <= 1:  # 원소가 1개 이하이면 그대로 반환
    return arr

pivot = arr[NEED_TO_SOLVE]  # 마지막 원소를 피벗으로 선택

left = []   # 피벗보다 작은 값들을 담을 리스트
middle = [] # 피벗과 같은 값들을 담을 리스트
right = []  # 피벗보다 큰 값들을 담을 리스트

for x in arr:
    if x < pivot:
        NEED_TO_SOLVE.append(x)     # 피벗보다 작은 값
    elif x == pivot:
        NEED_TO_SOLVE.append(x)   # 피벗과 같은 값
    else:
        NEED_TO_SOLVE.append(x)    # 피벗보다 큰 값

return quick_sort(NEED_TO_SOLVE) + NEED_TO_SOLVE + quick_sort(NEED_TO_SOLVE)  # 분할 정복 후 합쳐서 반환
```

김민서 (imkmsh@khu.ac.kr)

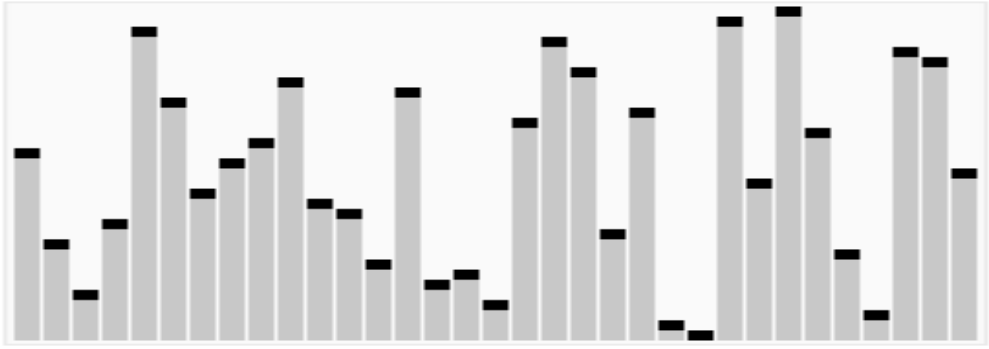
quick_sort



```python
if len(arr) <= 1:  # 원소가 1개 이하이면 그대로 반환
    return arr

pivot = arr[-1]  # 마지막 원소를 피벗으로 선택

left = []   # 피벗보다 작은 값들을 담을 리스트
middle = [] # 피벗과 같은 값들을 담을 리스트
right = []  # 피벗보다 큰 값들을 담을 리스트

for x in arr:
    if x < pivot:
        NEED_TO_SOLVE.append(x)     # 피벗보다 작은 값
    elif x == pivot:
        NEED_TO_SOLVE.append(x)   # 피벗과 같은 값
    else:
        NEED_TO_SOLVE.append(x)    # 피벗보다 큰 값

return quick_sort(NEED_TO_SOLVE) + NEED_TO_SOLVE + quick_sort(NEED_TO_SOLVE)  # 분할 정복 후 합쳐서 반환
```

김민서 (imkmsh@khu.ac.kr)

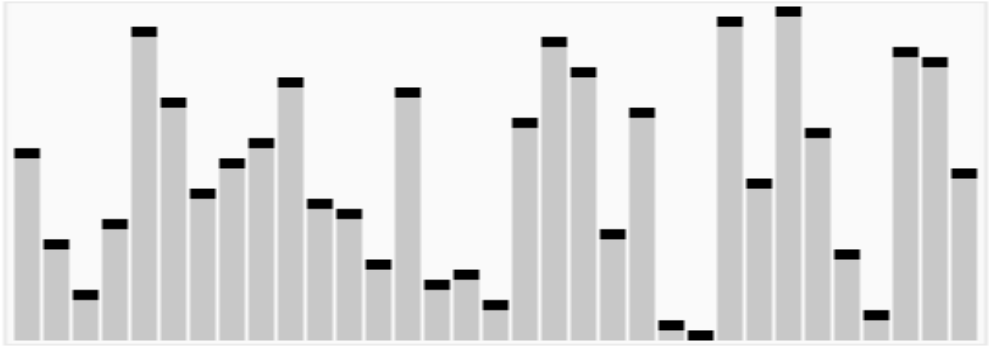
quick_sort



```python
if len(arr) <= 1:  # 원소가 1개 이하이면 그대로 반환
    return arr

pivot = arr[-1]  # 마지막 원소를 피벗으로 선택

left = []   # 피벗보다 작은 값들을 담을 리스트
middle = [] # 피벗과 같은 값들을 담을 리스트
right = []  # 피벗보다 큰 값들을 담을 리스트

for x in arr:
    if x < pivot:
        NEED_TO_SOLVE.append(x)     # 피벗보다 작은 값
    elif x == pivot:
        NEED_TO_SOLVE.append(x)   # 피벗과 같은 값
    else:
        NEED_TO_SOLVE.append(x)    # 피벗보다 큰 값

return quick_sort(NEED_TO_SOLVE) + NEED_TO_SOLVE + quick_sort(NEED_TO_SOLVE)  # 분할 정복 후 합쳐서 반환
```

김민서 (imkmsh@khu.ac.kr)

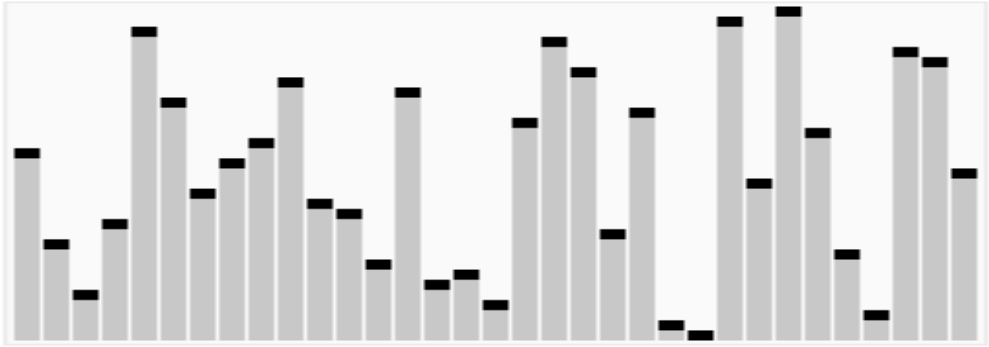
quick_sort



```
if len(arr) <= 1:  # 원소가 1개 이하이면 그대로 반환
    return arr

pivot = arr[-1]  # 마지막 원소를 피벗으로 선택

left = []   # 피벗보다 작은 값들을 담을 리스트
middle = [] # 피벗과 같은 값들을 담을 리스트
right = []  # 피벗보다 큰 값들을 담을 리스트

for x in arr:
    if x < pivot:
        left.append(x)      # 피벗보다 작은 값
    elif x == pivot:
        middle.append(x)    # 피벗과 같은 값
    else:
        right.append(x)     # 피벗보다 큰 값

return quick_sort(NEED_TO_SOLVE) + NEED_TO_SOLVE + quick_sort(NEED_TO_SOLVE)  # 분할 정복 후 합쳐서 반환
```

김민서 (imkmsh@khu.ac.kr)

quick_sort

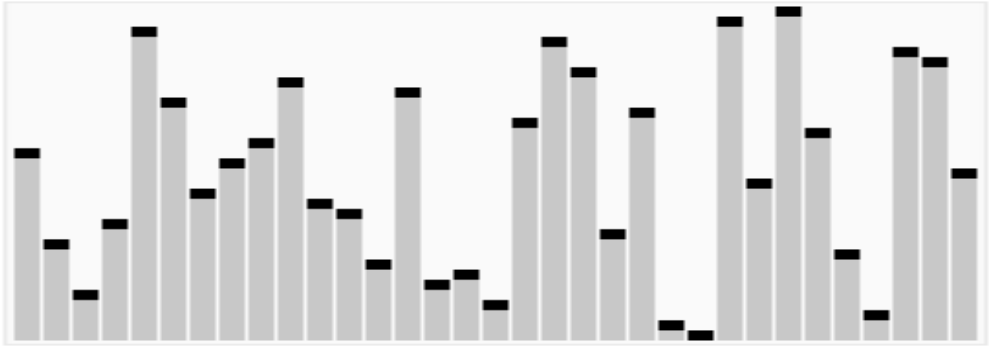


```
if len(arr) <= 1:  # 원소가 1개 이하이면 그대로 반환
    return arr

pivot = arr[-1]  # 마지막 원소를 피벗으로 선택

left = []   # 피벗보다 작은 값들을 담을 리스트
middle = [] # 피벗과 같은 값들을 담을 리스트
right = []  # 피벗보다 큰 값들을 담을 리스트

for x in arr:
    if x < pivot:
        left.append(x)      # 피벗보다 작은 값
    elif x == pivot:
        middle.append(x)    # 피벗과 같은 값
    else:
        right.append(x)     # 피벗보다 큰 값
```

```
return quick_sort(NEED_TO_SOLVE) + NEED_TO_SOLVE + quick_sort(NEED_TO_SOLVE)  # 분할 정복 후 합쳐서 반환
```

김민서 (imkmsh@khu.ac.kr)

quick_sort



```python
if len(arr) <= 1:  # 원소가 1개 이하이면 그대로 반환
    return arr

pivot = arr[-1]  # 마지막 원소를 피벗으로 선택

left = []   # 피벗보다 작은 값들을 담을 리스트
middle = [] # 피벗과 같은 값들을 담을 리스트
right = []  # 피벗보다 큰 값들을 담을 리스트

for x in arr:
    if x < pivot:
        left.append(x)      # 피벗보다 작은 값
    elif x == pivot:
        middle.append(x)    # 피벗과 같은 값
    else:
        right.append(x)     # 피벗보다 큰 값

return quick_sort(left) + middle + quick_sort(right)  # 분할 정복 후 합쳐서 반환
```

김민서 (imkmsh@khu.ac.kr)

생각해 볼 만한 QUIZ

WHY?

- quick_sort, merge_sort에는
  if len(arr) <= 1: return arr 같은 **예외 처리**가 있는데,
- selection_sort, bubble_sort, insertion_sort에는 없을까요

생각해 볼 만한 QUIZ

WHY?

- quick_sort나 merge_sort에는
  if len(arr) <= 1: return arr 같은 **예외 처리**가 있는데,
- selection_sort, bubble_sort, insertion_sort에는 없을까요

- **재귀 (Quick, Merge)**
  → 종료 조건이 반드시 필요 (len <= 1 없으면 무한 재귀)
- **반복 (Selection, Bubble, Insertion)**
  → 종료 조건이 없어도 반복문이 자동으로 0회 실행되므로 안전

생각해 볼 만한 QUIZ

1. quick_sort / merge_sort (재귀 기반 정렬)

- **분할 정복(divide & conquer)** 방식이라서 **재귀 호출**을 사용
- 매번 배열을 나눠서 다시 quick_sort 또는 merge_sort를 호출하기 때문에, 빈 리스트([])나 원소가 하나만 있는 리스트([x])가 인자로 넘어올 수 있음
- 이때는 더 이상 분할할 필요가 없으므로 **즉시 반환하는 예외 처리**가 필요

생각해 볼 만한 QUIZ

2. selection_sort / bubble_sort / insertion_sort (반복 기반 정렬)

- 이 세 가지는 모두 **반복문(loop)** 기반의 정렬
- 길이가 0이거나 1일 때는
  - selection_sort: for i in range(n - 1) → n이 0 또는 1이면 range(-1) 또는 range(0) → **반복문 자체가 실행되지 않음**
  - bubble_sort: for i in range(n - 1) → 마찬가지로 반복문 실행 안 됨
  - insertion_sort: for i in range(1, n) → n이 0이나 1이면 range가 비어 있음 → 반복문 실행 안 됨
- 따라서 if len(arr) <= 1 조건이 없어도 그냥 **원본 배열 그대로 반환**

샘플 크기에 따른 시간 차이

- small

```
(base) minseo@minseo-MacBookAir-2 data-structure-assist % python 0924-sort/main.py
샘플 크기 선택 (big/small): small
작은 샘플 (20개) 선택 ============================
샘플 크기: 20
앞부분 20개: [52085, 11210, 94293, 3206, 30442, 56560, 33124, 62590, 5353, 5528, 53205, 79093, 89933, 75656, 85745, 55601, 25184, 79717, 74662, 59198]


Selection Sort:
[3206, 5353, 5528, 11210, 25184, 30442, 33124, 52085, 53205, 55601, 56560, 59198, 62590, 74662, 75656, 79093, 79717, 85745, 89933, 94293]
Selection Sort Time: 0.030 ms


Bubble Sort:
[3206, 5353, 5528, 11210, 25184, 30442, 33124, 52085, 53205, 55601, 56560, 59198, 62590, 74662, 75656, 79093, 79717, 85745, 89933, 94293]
Bubble Sort Time: 0.014 ms


Insertion Sort:
[3206, 5353, 5528, 11210, 25184, 30442, 33124, 52085, 53205, 55601, 56560, 59198, 62590, 74662, 75656, 79093, 79717, 85745, 89933, 94293]
Insertion Sort Time: 0.006 ms


Quick Sort:
[3206, 5353, 5528, 11210, 25184, 30442, 33124, 52085, 53205, 55601, 56560, 59198, 62590, 74662, 75656, 79093, 79717, 85745, 89933, 94293]
Quick Sort Time: 0.014 ms


Merge Sort:
[3206, 5353, 5528, 11210, 25184, 30442, 33124, 52085, 53205, 55601, 56560, 59198, 62590, 74662, 75656, 79093, 79717, 85745, 89933, 94293]
Merge Sort Time: 0.017 ms
```

샘플 크기에 따른 시간 차이

- big

```
(base) minseo@minseo-MacBookAir-2 data-structure-assist % python 0924-sort/main.py
샘플 크기 선택 (big/small): big
큰 샘플 (10,000개) 선택 ==========================
샘플 크기: 10000
앞부분 20개: [57508, 13226, 9510, 85480, 98363, 98776, 66691, 7603, 96563, 28146, 55334, 30163, 93804, 96704, 91714, 60599, 38536, 14869, 66121, 84904]


Selection Sort:
[1, 1, 5, 6, 19, 77, 92, 97, 130, 157, 177, 177, 206, 212, 243, 261, 264, 271, 272, 281]
Selection Sort Time: 1488.783 ms


Bubble Sort:
[1, 1, 5, 6, 19, 77, 92, 97, 130, 157, 177, 177, 206, 212, 243, 261, 264, 271, 272, 281]
Bubble Sort Time: 3216.982 ms


Insertion Sort:
[1, 1, 5, 6, 19, 77, 92, 97, 130, 157, 177, 177, 206, 212, 243, 261, 264, 271, 272, 281]
Insertion Sort Time: 1270.869 ms


Quick Sort:
[1, 1, 5, 6, 19, 77, 92, 97, 130, 157, 177, 177, 206, 212, 243, 261, 264, 271, 272, 281]
Quick Sort Time: 6.582 ms


Merge Sort:
[1, 1, 5, 6, 19, 77, 92, 97, 130, 157, 177, 177, 206, 212, 243, 261, 264, 271, 272, 281]
Merge Sort Time: 13.076 ms
```

수고하셨습니다

자료구조 실습 09/24

EOF

김민서 (imkmsh@khu.ac.kr)