정렬

1. 선택정렬

처리되지 않은 데이터 중 가장 작은 데이터 선택해 맨 앞에 있는 데이터와 바꾸는 것 반복

## 선택 정렬 소스코드 (Python)

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

for i in range(len(array)):
    min_index = i # 가장 작은 원소의 인덱스
    for j in range(i + 1, len(array)):
        if array[min_index] > array[j]:
            min_index = j
    array[i], array[min_index] = array[min_index], array[i] # 스와프

print(array)
```

선택 정렬은 N번 만큼 가장 작은 수를 찾아서 맨 앞으로 보내야 합니다.

구현 방식에 따라서 사소한 오차는 있을 수 있지만, 전체 연산 횟수는 다음과 같습니다.

$$N + (N - 1) + (N - 2) + \dots + 2$$

이는 $(N^2 + N - 2) / 2$ 로 표현할 수 있는데, 빅오 표기법에 따라서 $O(N^2)$이라고 작성합니다.

2. 삽입정렬

- 처리되지 않은 데이터를 하나씩 골라 **적절한 위치에 삽입**합니다.
- 선택 정렬에 비해 구현 난이도가 높은 편이지만, 일반적으로 더 효율적으로 동작합니다.

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

for i in range(1, len(array)):
    for j in range(i, 0, -1): # 인덱스 i부터 1까지 1씩 감소하며 반복하는 문법
        if array[j] < array[j - 1]: # 한 칸씩 왼쪽으로 이동
            array[j], array[j - 1] = array[j - 1], array[j]
        else: # 자기보다 작은 데이터를 만나면 그 위치에서 멈춤
            break

print(array)
```

삽입 정렬의 시간 복잡도는 $O(N^2)$이며, 선택 정렬과 마찬가지로 반복문이 두 번 중첩되어 사용됩니다.

삽입 정렬은 현재 리스트의 데이터가 거의 정렬되어 있는 상태라면 매우 빠르게 동작합니다.

- 최선의 경우 $O(N)$의 시간 복잡도를 가집니다.
- 이미 정렬되어 있는 상태에서 다시 삽입 정렬을 수행하면 어떻게 될까요?

3. 퀵 정렬

재귀적으로 수행되며 점점 범위가 좁아짐/ 이미 정렬된 배열이면 시간복잡도 N제곱

pivot(기준값)

기준 데이터를 설정하고 그 **기준보다 큰 데이터와 작은 데이터의 위치를 바꾸는 방법**입니다.

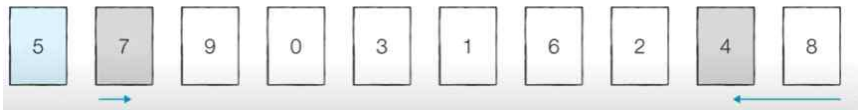
일반적인 상황에서 가장 많이 사용되는 정렬 알고리즘 중 하나입니다.

병합 정렬과 더불어 대부분의 프로그래밍 언어의 정렬 라이브러리의 근간이 되는 알고리즘입니다.

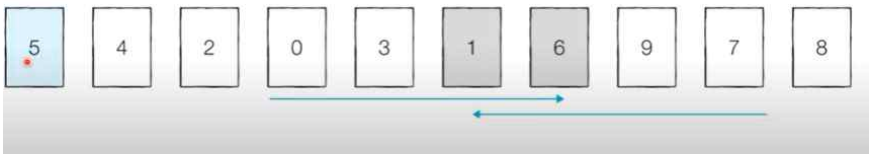
가장 기본적인 퀵 정렬은 **첫 번째 데이터를 기준 데이터(Pivot)로 설정**합니다.

### 퀵 정렬 동작 예시

- **[Step 0]** 현재 피벗의 값은 '5'입니다. 왼쪽에서부터 '5'보다 큰 데이터를 선택하므로 '7'이 선택되고, 오른쪽에서부터 '5'보다 작은 데이터를 선택하므로 '4'가 선택됩니다. 이제 이 두 데이터의 위치를 서로 변경합니다.



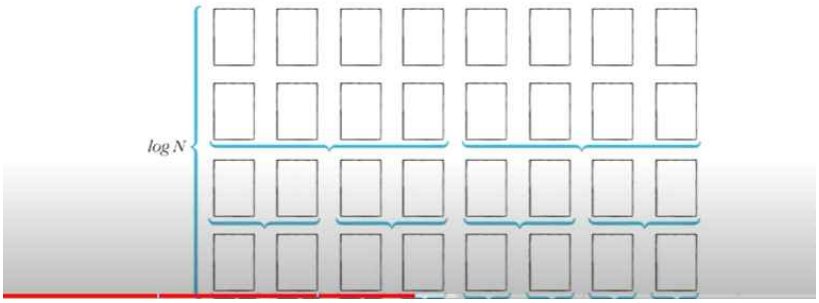

- **[Step 2]** 현재 피벗의 값은 '5'입니다. 왼쪽에서부터 '5'보다 큰 데이터를 선택하므로 '6'이 선택되고, 오른쪽에서부터 '5'보다 작은 데이터를 선택하므로 '1'이 선택됩니다. 단, 이처럼 **위치가 엇갈리는 경우 '피벗'과 '작은 데이터'의 위치를 서로 변경**합니다.




- **[오른쪽 데이터 묶음 정렬]** 오른쪽에 있는 데이터에 대해서 마찬가지로 정렬을 수행합니다.
  - 이러한 과정을 반복하면 전체 데이터에 대해서 정렬이 수행됩니다.

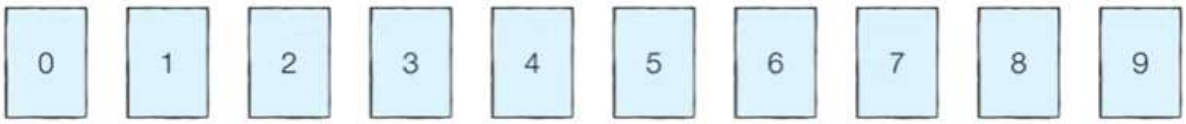

### 퀵 정렬이 빠른 이유: 직관적인 이해

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



## 퀵 정렬의 시간 복잡도

- 퀵 정렬은 평균의 경우 $O(NlogN)$의 시간 복잡도를 가집니다.
- 하지만 최악의 경우 $O(N^2)$의 시간 복잡도를 가집니다.
  - 첫 번째 원소를 피벗으로 삼을 때, 이미 정렬된 배열에 대해서 퀵 정렬을 수행하면 어떻게 될까요?




## 퀵 정렬 소스코드: 일반적인 방식 (Python)

```python
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 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)

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

실행 결과

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

## 퀵 정렬 소스코드: 파이썬의 장점을 살린 방식

```python
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 정렬 수행하고, 전체 리스트 반환
    return quick_sort(left_side) + [pivot] + quick_sort(right_side)

print(quick_sort(array))
```

실행 결과

4.계수 정렬
데이터의 범위가 크면 비효율적 0, 99999 단지 두 개 원소지만 배열은 1000000개 만들어야함

특정한 조건이 부합할 때만 사용할 수 있지만 **매우 빠르게 동작하는** 정렬 알고리즘입니다.

- 계수 정렬은 **데이터의 크기 범위가 제한되어 정수 형태로 표현할 수 있을 때** 사용 가능합니다.

데이터의 개수가 $N$, 데이터(양수) 중 최댓값이 $K$일 때 최악의 경우에도 수행 시간 $\boldsymbol{O(N+K)}$를 보장합니다.

- **[Step 0]** 가장 작은 데이터부터 가장 큰 데이터까지의 범위가 모두 담길 수 있도록 리스트를 생성합니다.
- **정렬할 데이터**: 7 5 9 0 3 1 6 2 9 1 4 8 0 5 2

| 인덱스 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 개수(Count) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## 계수 정렬 소스코드 (Python)

```python
# 모든 원소의 값이 0보다 크거나 같다고 가정
array = [7, 5, 9, 0, 3, 1, 6, 2, 9, 1, 4, 8, 0, 5, 2]
# 모든 범위를 포함하는 리스트 선언(모든 값은 0으로 초기화)
count = [0] * (max(array) + 1)

for i in range(len(array)):
    count[array[i]] += 1 # 각 데이터에 해당하는 인덱스의 값 증가

for i in range(len(count)): # 리스트에 기록된 정렬 정보 확인
    for j in range(count[i]):
        print(i, end=' ') # 띄어쓰기를 구분으로 등장한 횟수만큼 인덱스 출력
```

계수 정렬의 시간 복잡도와 공간 복잡도는 모두 $\boldsymbol{O(N+K)}$입니다.

계수 정렬은 때에 따라서 심각한 비효율성을 초래할 수 있습니다.

- 데이터가 0과 999,999로 단 2개만 존재하는 경우를 생각해 봅시다.

계수 정렬은 **동일한 값을 가지는 데이터가 여러 개 등장할 때** 효과적으로 사용할 수 있습니다.

- 성적의 경우 100점을 맞은 학생이 여러 명일 수 있기 때문에 계수 정렬이 효과적입니다.

## 정렬 알고리즘 비교하기

- 앞서 다룬 네 가지 정렬 알고리즘을 비교하면 다음과 같습니다.
- 추가적으로 대부분의 프로그래밍 언어에서 지원하는 표준 정렬 라이브러리는 최악의 경우에도 $O(NlogN)$을 보장하도록 설계되어 있습니다.

| 정렬 알고리즘 | 평균 시간 복잡도 | 공간 복잡도 | 특징 |
|---|---|---|---|
| 선택 정렬 | $O(N^2)$ | $O(N)$ | 아이디어가 매우 간단합니다. |
| 삽입 정렬 | $O(N^2)$ | $O(N)$ | 데이터가 거의 정렬되어 있을 때는 가장 빠릅니다. |
| 퀵 정렬 | $O(NlogN)$ | $O(N)$ | 대부분의 경우에 가장 적합하며, 충분히 빠릅니다. |
| 계수 정렬 | $O(N+K)$ | $O(N+K)$ | 데이터의 크기가 한정되어 있는 경우에만 사용이 가능하지만 매우 빠르게 동작합니다. |

## 선택 정렬과 기본 정렬 라이브러리 수행 시간 비교

```python
from random import randint
import time

# 배열에 10,000개의 정수를 삽입
array = []
for _ in range(10000):
    # 1부터 100 사이의 랜덤한 정수
    array.append(randint(1, 100))

# 선택 정렬 프로그램 성능 측정
start_time = time.time()

# 선택 정렬 프로그램 소스코드
for i in range(len(array)):
    min_index = i # 가장 작은 원소의 인덱스
    for j in range(i + 1, len(array)):
        if array[min_index] > array[j]:
            min_index = j
    array[i], array[min_index] = array[min_index], array[i]

# 측정 종료
end_time = time.time()
# 수행 시간 출력
print("선택 정렬 성능 측정:", end_time - start_time)
```

```python
# 배열을 다시 무작위 데이터로 초기화
array = []
for _ in range(10000):
    # 1부터 100 사이의 랜덤한 정수
    array.append(randint(1, 100))

# 기본 정렬 라이브러리 성능 측정
start_time = time.time()

# 기본 정렬 라이브러리 사용
array.sort()

# 측정 종료
end_time = time.time()
# 수행 시간 출력
print("기본 정렬 라이브러리 성능 측정:", end_time - start_time)
```

실행 결과

```
선택 정렬 성능 측정: 35.841460943222046
기본 정렬 라이브러리 성능 측정: 0.0013387203216552734
```