



# 버블정렬, 선택정렬, 삽입정렬

## 0. 내장함수로 정렬할 수 있잖아?

- 리스트: `sort()`, `sorted(list)`
- 딕셔너리: `sorted(dict.keys())`, `sorted(dict.values())` 등등..
- 내장 메소드만으로 정렬이 가능한데 왜 여러 종류의 정렬 알고리즘을 알아야 하는것일 까?
  - 내장 메소드의 정렬 알고리즘이 늘 최선의 퍼포먼스를 보장하지는 않기 때문이다. 데이터의 양이나 상황에 따라 적합한 정렬 알고리즘을 직접 구현하여 사용하는 것이 더 높은 효율을 낼 수 있다.
- 각 정렬의 시간복잡도

Name	Best	Avg	Worst	Run-time(정수 60,000개) 단위: sec
삽입정렬	$n$	$n^2$	$n^2$	7.438
선택정렬	$n^2$	$n^2$	$n^2$	10.842
버블정렬	$n^2$	$n^2$	$n^2$	22.894
퀵 정렬	$n \log_2 n$	$n \log_2 n$	$n^2$	0.014
힙 정렬	$n \log_2 n$	$n \log_2 n$	$n \log_2 n$	0.034
병합정렬	$n \log_2 n$	$n \log_2 n$	$n \log_2 n$	0.026

출처: <https://gmlwjd9405.github.io/2018/05/06/algorithm-insertion-sort.html>

## 1. 버블 정렬

- 정렬 되는 과정

6 5 3 1 8 7 2 4

- 서로 인접한 두 원소를 검사하여 정렬하는 알고리즘. 인접한 2개의 레코드를 비교하여 크기가 순서대로 되어 있지 않으면 서로 교환한다.
- 앞에서부터 옆의 숫자와 비교해가면서 스위칭하는 것
- 장단점
  - 장점: 구현하기 쉽다
  - 단점: 시간복잡도는 무조건  $O(n^2)$  → 비효율적
- 선택 정렬과 기본 개념이 유사하다.

#### ▼ 코드

```
def bubble_sort(arr):
    for i in range(len(arr) - 1, 0, -1):
        for j in range(i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

#### ▼ 한 번 읽어보기 - 최적화

출처: <https://www.daleseo.com/sort-bubble/>

1. 이전 패스에서 앞뒤 자리 비교(swap)이 한 번도 일어나지 않았다면 정렬되지 않는 값이 하나도 없었다고 간주할 수 있습니다. 따라서 이럴 경우, 이후 패스를 수행하지 않아도 됩니다.

Initial: [1, 2, 3, 5, 4]

Pass 1: [1, 2, 3, 4, 5] => Swap 있었음

\*

Pass 2: [1, 2, 3, 4, 5] => Swap 없었음

\* \*

=> 이전 패스에서 swap이 한 번도 없었으니 종료

```
def bubble_sort(arr):
    for i in range(len(arr) - 1, 0, -1):
        swapped = False
        for j in range(i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break
```

2. 이전 패스에서 앞뒤 자리 비교(swap)가 있었는지 여부를 체크하는 대신 마지막으로 앞뒤 자리 비교가 있었던 index를 기억해두면 다음 패스에서는 그 자리 전까지만 정렬해도 됩니다. 따라서 한 칸씩 정렬 범위를 줄여나가는 대신 한번에 여러 칸씩 정렬 범위를 줄여나갈 수 있습니다.

Initial: [3, 2, 1, 4, 5]


Pass 1: [2, 1, 3, 4, 5] => 마지막 Swap 위치가 index 1  
           ^                  \*


Pass 2: [1, 2, 3, 4, 5] => 중간 패스 skip하고 바로 index 1로 보낼 값 찾기  
           ^      \*   \*   \*


```
def bubble_sort(arr):
    end = len(arr) - 1
    while end > 0:
        last_swap = 0
        for i in range(end):
            if arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                last_swap = i
        end = last_swap
```

## 2. 선택 정렬

- 정렬 되는 과정

 비교하는 위치

 현재최소값

 정렬된데이터



- 정렬되지 않은 데이터를 살피고, 가장 작은 데이터를 앞으로 가지고 나온다.
- 입력 데이터의 크기가 n개일 경우, 이 알고리즘은 n-1회, n-2회 .....의 비교를 반복한다.  
→ 시간복잡도 = 무조건  $O(n^2)$
- 장단점
  - 장점: 자료 이동 횟수가 미리 결정된다.
  - 단점: 안정성을 만족하지 않는다. 즉, 값이 같은 레코드가 있는 경우에 상대적인 위치가 변경될 수 있다.

#### ▼ 코드

```
def selection_sort(arr):
    for i in range(len(arr) - 1):
        min_idx = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

### 3. 삽입 정렬

- 정렬 되는 과정

6 5 3 1 8 7 2 4

- 자료 배열의 모든 요소를 앞에서부터 차례대로 이미 정렬된 배열 부분과 비교 하여, 자신의 위치를 찾아 삽입함으로써 정렬을 완성하는 알고리즘. 매 순서마다 해당 원소를 삽입할 수 있는 위치를 찾아 해당 위치에 넣는다.

#### ▼ 시간복잡도 최선, 최악 계산

- 최선의 경우
  - 비교 횟수  
이동 없이 1번의 비교만 이루어진다.  
외부 루프: (n-1)번  
 $\text{Best } T(n) = O(n)$
- 최악의 경우(입력 자료가 역순일 경우)
  - 비교 횟수  
외부 루프 안의 각 반복마다 i번의 비교 수행  
외부 루프:  $(n-1) + (n-2) + \dots + 2 + 1 = n(n-1)/2 = O(n^2)$
  - 교환 횟수  
외부 루프의 각 단계마다 (i+2)번의 이동 발생  
 $n(n-1)/2 + 2(n-1) = (n^2+3n-4)/2 = O(n^2)$   
 $\text{Worst } T(n) = O(n^2)$
- 장단점
  - 장점: 안정한 정렬 방법, 데이터의 수가 적을 경우 알고리즘 자체가 매우 간단하므로 다른 복잡한 정렬 방법보다 유리할 수 있다, 대부분의 데이터가 이미 정렬되어 있는 경우에 매우 효율적일 수 있다.
  - 단점: 비교적 많은 데이터들의 이동을 포함한다, 데이터 수가 많고 데이터 크기가 클 경우에 적합하지 않다.

#### ▼ 코드

```
def insertion_sort(arr):
    for end in range(1, len(arr)):
        for i in range(end, 0, -1):
            if arr[i - 1] > arr[i]:
                arr[i - 1], arr[i] = arr[i], arr[i - 1]
```

## ▼ 한 번 읽어보기 - 최적화

출처: <https://www.daleseo.com/sort-bubble/>

1. 기존에 있던 값들은 이전 패스에서 모두 정렬되었다는 점을 활용하면 불필요한 비교 작업을 제거할 수 있습니다. 새롭게 추가된 값보다 작은 숫자를 만나는 최초의 순간까지만 내부 반복문을 수행해도 됩니다.

```
[1, 2, 3, 5, 4, ...]: 5 > 4 => swap
* * * * ^
[1, 2, 3, 4, 5, ...]: 3 < 4 => OK => all sorted!
* * * * *
```

```
def insertion_sort(arr):
    for end in range(1, len(arr)):
        i = end
        while i > 0 and arr[i - 1] > arr[i]:
            arr[i - 1], arr[i] = arr[i], arr[i - 1]
            i -= 1
```

2. swap 작업없이 단순히 값들을 shift 시키는 것만으로도 삽입 정렬의 구현이 가능합니다. 앞의 값이 정렬 범위에 추가시킨 값보다 클 경우 앞의 값을 뒤로 밀다가 최초로 작은 값을 만나는 순간 그 뒤에 추가된 값을 꼽으면 됩니다.

```
def insertion_sort(arr):
    for end in range(1, len(arr)):
        to_insert = arr[end]
        i = end
        while i > 0 and arr[i - 1] > to_insert:
            arr[i] = arr[i - 1]
            i -= 1
        arr[i] = to_insert
```

## 문제들

BOJ. 2750 수 정렬하기 (브론즈 2)

BOJ. 1517 버블소트 (플레5ㄷㄷ)