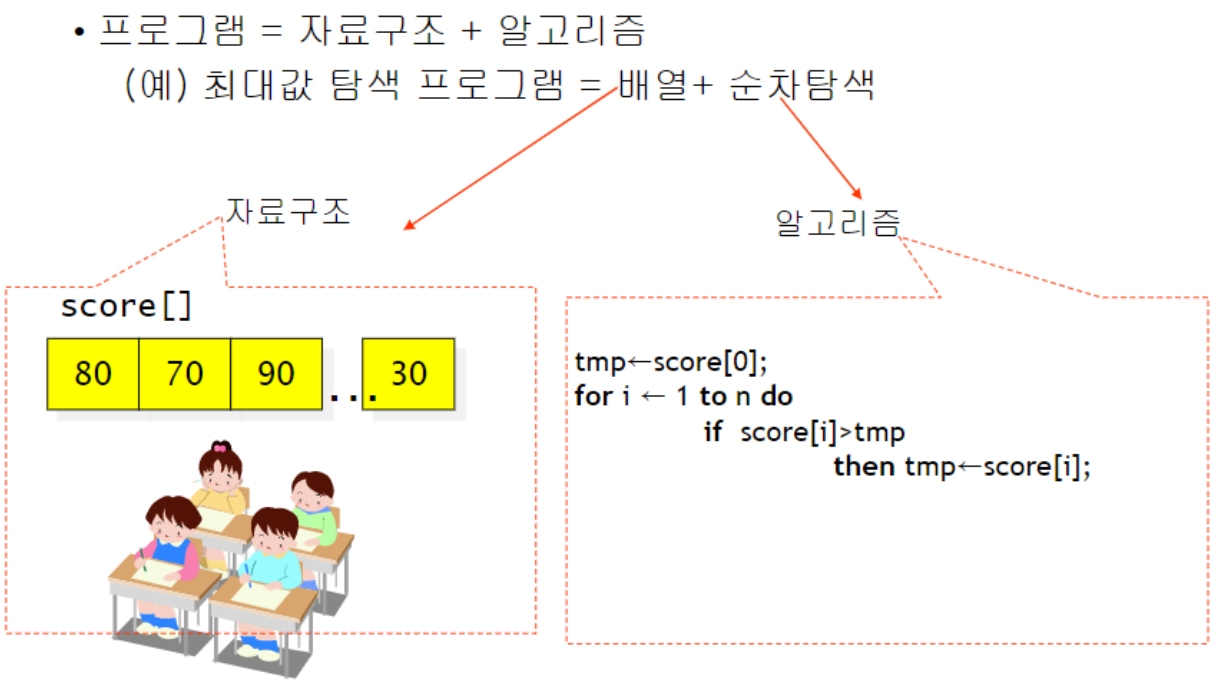
[자료구조] 자료구조 vs 알고리즘 / 정렬 알고리즘

자료구조 VS 알고리즘



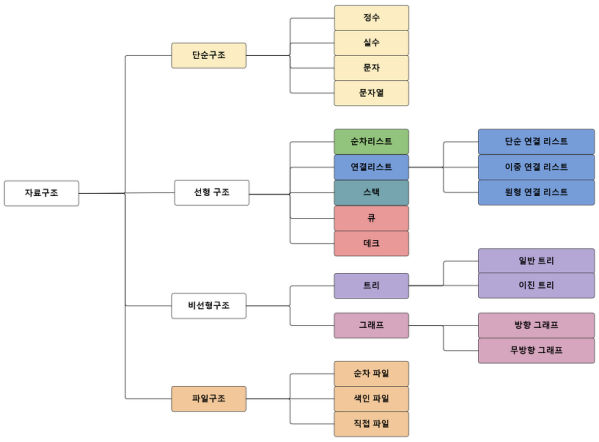
https://roi-data.com/entry/자료구조-개론-1-자료구조-알고리즘-①

자료구조

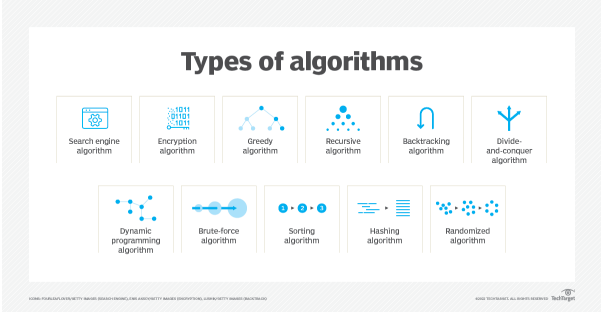
- 데이터를 저장하고 조작하는 방법

알고리즘

- 자료구조를 활용해 주어진 문제를 해결하기 위한 방법



https://velog.io/@jisoung/자료구조란



https://www.techtarget.com/whatis/definition/algorithm

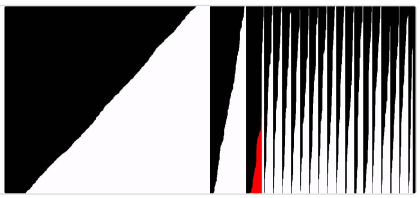
정렬 알고리즘

개요

15 Sorting Algorithms in 6 Minutes

Visualization and "audibilization" of 15 Sorting Algorithms in 6 Minutes.
Sorts random shuffles of integers, with both speed and the number of items adapted to each algorithm's complexity.

https://www.youtube.com/watch?v=kPRA0W1kECg



- 데이터를 특정 순서대로 나열
- 탐색을 빠르게 하기 위하여 사용
- 보통 입력데이터는 **배열 구조**

대표적인 비교 기반 정렬 알고리즘 6가지

정렬 알고리즘	평균 시간 복잡도	최선 시간 복잡도 (Best)	최악 시간 복잡도 (Worst)	메모리 사용량	안정성	방식
버블 정렬 (Bubble Sort)	O(n^2)	O(n)	O(n^2)	O(1)	안정	교환
선택 정렬 (Selection Sort)	O(n^2)	O(n^2)	O(n^2)	O(1)	불안정	선택
삽입 정렬 (Insertion Sort)	O(n^2)	O(n)	O(n^2)	O(1)	안정	삽입
퀵 정렬 (Quick Sort)	O(n log n)	O(n log n)	O(n^2)	O(log n)	불안정	파티셔닝 (분할 정복)

정렬 알고리즘	평균 시간 복잡도	최선 시간 복잡도 (Best)	최악 시간 복잡도 (Worst)	메모리 사용량	안정성	방식
병합 정렬 (Merge Sort)	O(n log n)	O(n log n)	O(n log n)	O(n)	안정	병합 (분할 정복)
힙 정렬 (Heap Sort)	O(n log n)	O(n log n)	O(n log n)	O(1)	불안정	선택

💡 **정렬 알고리즘 구분짓는 특징들**

>> 적절한 알고리즘을 선택하자! <<

1. **비교 기반 vs 비교되지 않는 기반 (Comparison-based vs Non-comparison-based)**:
   - **비교 기반**:

     데이터를 직접적으로 비교하여 정렬
   - **비교되지 않는 기반**:

     데이터를 직접적으로 비교 없이 정렬

     ex) 계수 정렬: 데이터의 등장횟수에 따라 정렬

2. **제자리 정렬 vs 제자리가 아닌 정렬 (In-place vs Not-in-place)**:
   - **제자리 정렬(In-place sorting):**

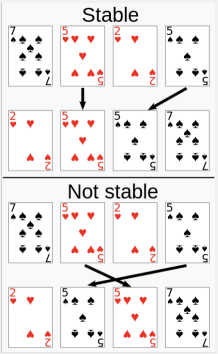
     정렬을 수행하는 동안에 추가적인 메모리 사용하지 않음
   - **제자리가 아닌 정렬 (Non-in-place sorting):**

     추가적인 메모리를 사용하여 정렬

3. **안정 정렬 vs 불안정 정렬 (Stable vs Unstable)**:
   - **안정 정렬 (Stable)**:

     중복 값에 대하여 정렬 후에도 입력 순서가 유지
   - **불안정 정렬 (Unstable):**

     중복 값이 정렬 후에도 순서가 유지된다는 보장이 없음



4. **반복적인 vs 재귀적인 (Iterative vs Recursive)**:
   - 반복문 / 재귀를 통한 정렬

5. **분할 정복 (Divide and Conquer)**:

주어진 배열을 작은 부분으로 분할하고, 각 부분을 정렬한 후, 병합하는 방식 사용

ex) 병합 정렬(Merge Sort),퀵 정렬(Quick Sort)

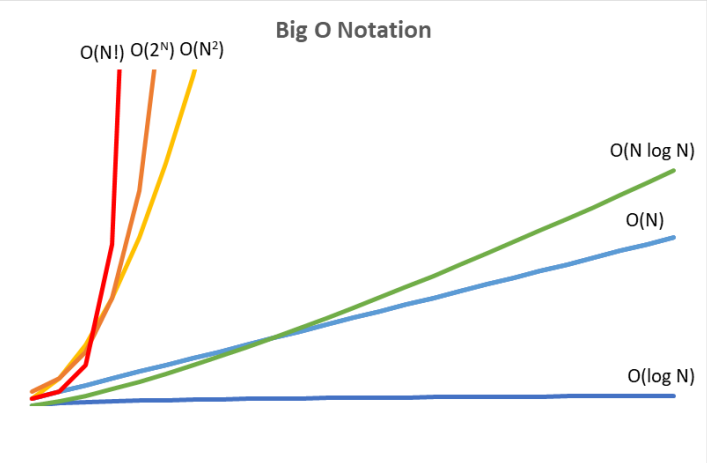
- 그 외: 쉘 정렬(Shell Sort), 기수 정렬(Radix Sort)...

정렬 알고리즘

컴퓨터 과학과 수학에서 정렬 알고리즘(sorting algorithm)이란 원소들을 번호순이나 사전 순서와 같이 일정한 순서대로 열거하는 알고리즘이다. 효율적인 정렬은 탐색이나 병합 알고리즘처럼 다른 알고리즘을 최적화하는 데 중요하다. 또 정렬 알고리즘은 데이터의 정규화나 의미있는 결과물을 생성하는 데 유용히

W https://ko.wikipedia.org/wiki/정렬_알고리즘

▼ 참고



버블 정렬 (Bubble Sort)

인접한 두 데이터의 크기를 비교하여 작은 값이 앞으로 이동시키는 정렬

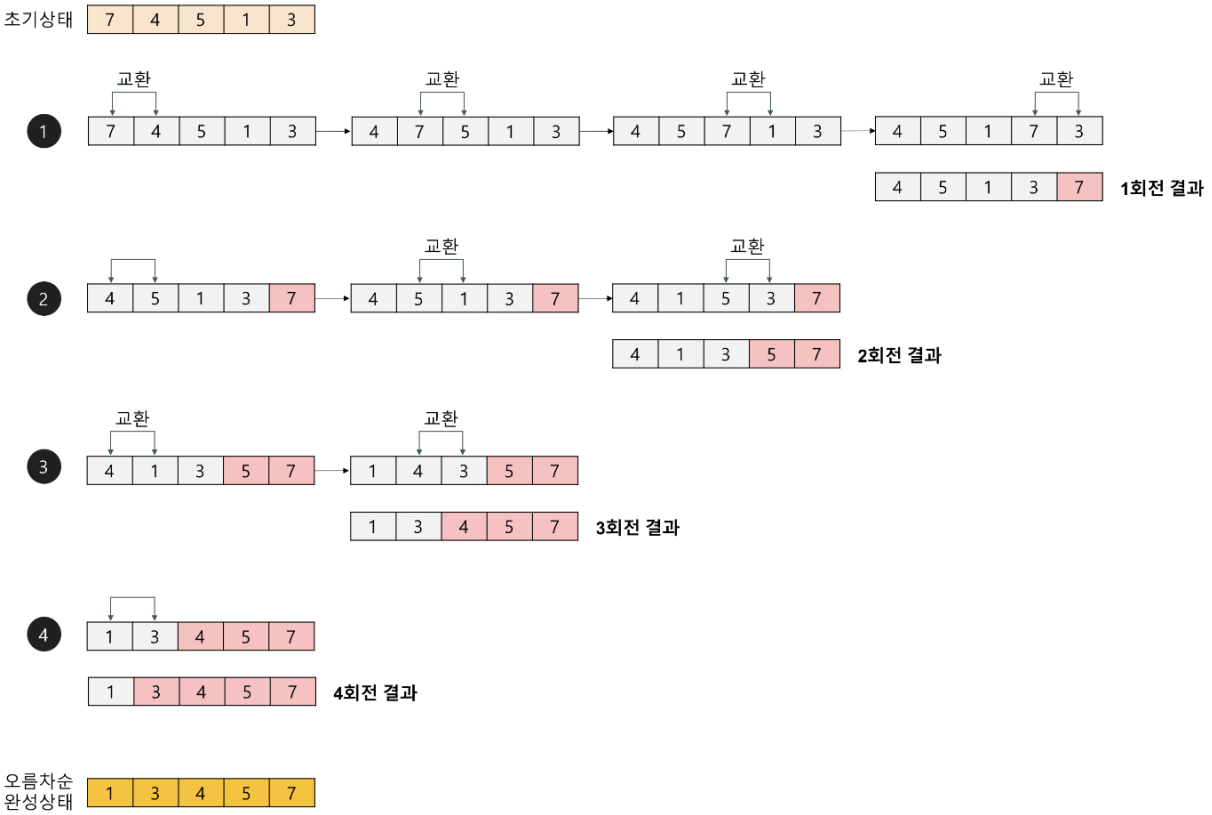
정렬 알고리즘	평균 시간 복잡도	최선 시간 복잡도 (Best)	최악 시간 복잡도 (Worst)	메모리 사용량	안정성	방식
버블 정렬 (Bubble Sort)	O(n^2)	O(n)	O(n^2)	O(1)	안정	교환

특징

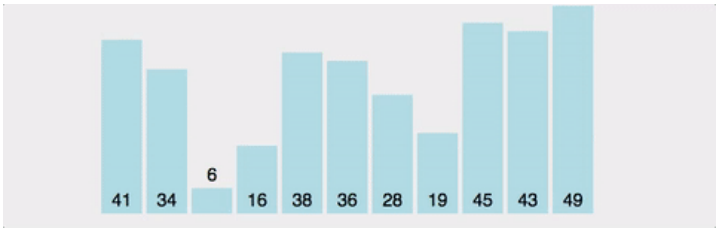
- 비교 기반
- 제자리 정렬
- 안정 정렬
- 반복적인

동작 방식

```
1. 앞에서부터 순차적으로 두 개의 데이터를 값을 비교하여, 작은값이 앞으로 오도록 swap한다
2. 1을 배열 끝까지 반복한다
3. 모든 데이터가 정렬될 때까지 1,2를 반복한다
```



https://gmlwjd9405.github.io/2018/05/06/algorithm-bubble-sort.html



https://jinhyy.tistory.com/9

장점

- 구현이 단순하다
- 적은 데이터에 유리
- 제자리 정렬
- 안정 정렬

단점

- 오래걸린다 (가장 느림)
- swap 연산이 많이 일어난다

선택 정렬 (Selection Sort)

해당 위치에 어떤 원소를 넣을지 선택하는 알고리즘
(순차적으로 데이터를 검사하여 최소값을 골라, 선택한 위치에 이동시켜서 정렬)

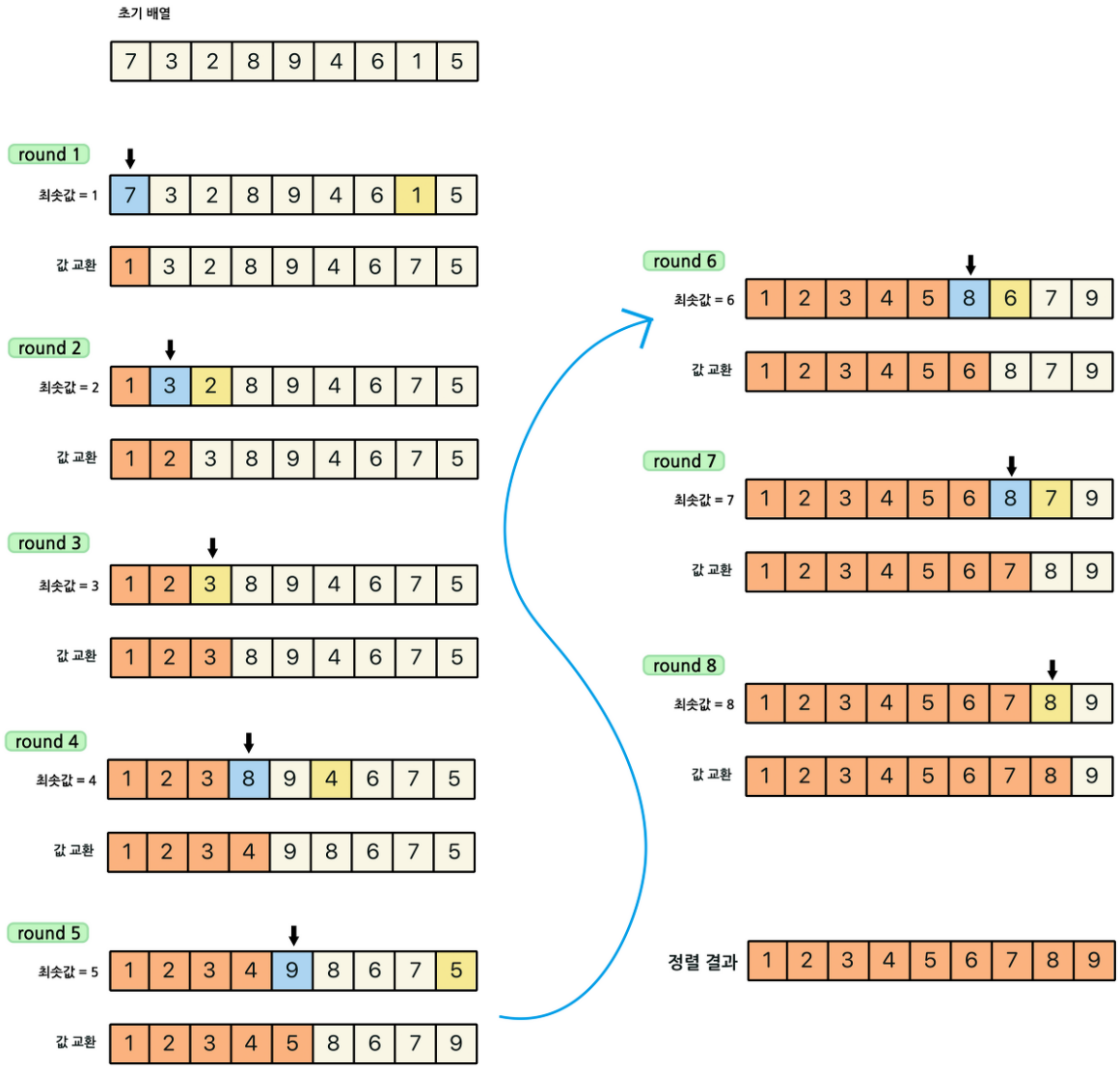
정렬 알고리즘	평균 시간 복잡도	최선 시간 복잡도 (Best)	최악 시간 복잡도 (Worst)	메모리 사용량	안정성	방식
선택 정렬 (Selection Sort)	O(n^2)	O(n^2)	O(n^2)	O(1)	불안정	선택

특징

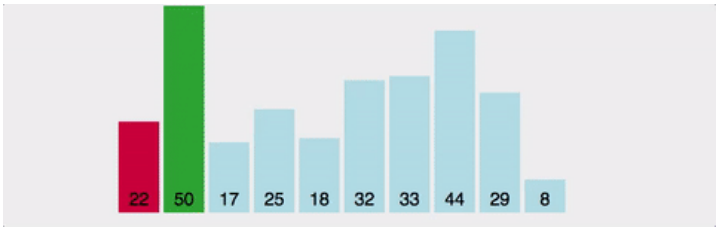
- 비교 기반
- 제자리 정렬
- 불안정 정렬
- 반복적인

동작 방식

```
1. 배열에서 최소값을 찾는다
2. 최소값을 앞자리(선택한 위치)로 이동시킨다
3. 1, 2를 반복한다
```



https://st-lab.tistory.com/168



https://velog.io/@velgmzz/Algorithm-선택-정렬-Selection-Sort

장점

- 구현이 단순

- 적은 데이터에 유리
- 제자리 정렬

단점

- 오래걸린다
- 불안정한 정렬

삽입 정렬 (Insertion Sort)

각 데이터를 적절한 위치에 삽입하는 방식의 정렬

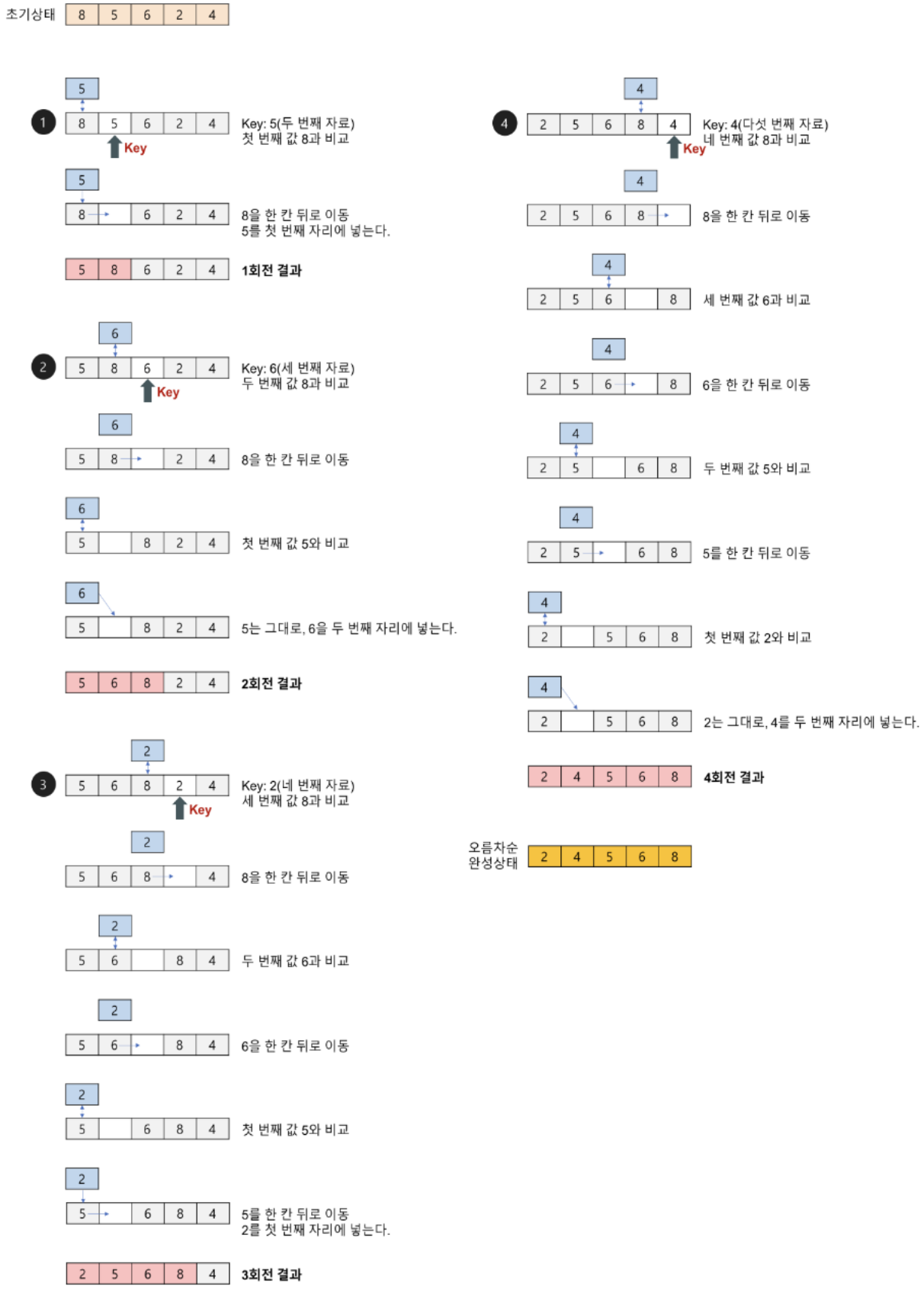
정렬 알고리즘	평균 시간 복잡도	최선 시간 복잡도 (Best)	최악 시간 복잡도 (Worst)	메모리 사용량	안정성	방식
삽입 정렬 (Insertion Sort)	O(n^2)	O(n)	O(n^2)	O(1)	안정	삽입

특징

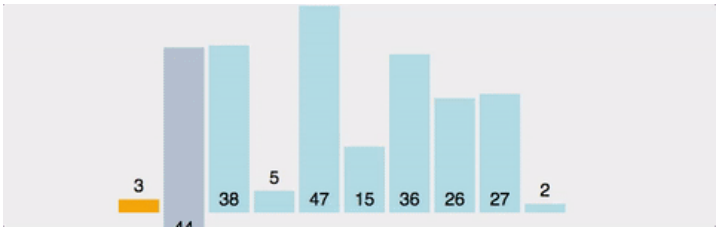
- 비교 기반
- 제자리 정렬
- 안정 정렬
- 반복적인

동작 방식

```
** 첫번째 값은 정렬되어있다고 가정하고 넘어간다 -> 2번쨰 위치부터 비교 시작
1. 2번째 위치의 값을 따로 저장한다
2. 현재 위치의 값과 그 이전 값들을 뒤에서 순차적으로 검사하여, 적절한 위치에 삽입한다
3. 순차적으로 다음 값을 타켓으로 잡아 배열이 끝날 때까지 1,2를 반복한다
```


초기상태
8 5 6 2 4
1
Key: 5(두 번째 자료)
첫 번째 값 8과 비교
Key
8을 한 칸 뒤로 이동
5를 첫 번째 자리에 넣는다.
1회전 결과
2
Key: 6(세 번째 자료)
두 번째 값 8과 비교
8을 한 칸 뒤로 이동
첫 번째 값 5와 비교
5는 그대로, 6을 두 번째 자리에 넣는다.
2회전 결과
3
Key: 2(네 번째 자료)
세 번째 값 8과 비교
8을 한 칸 뒤로 이동
두 번째 값 6과 비교
6을 한 칸 뒤로 이동
첫 번째 값 5와 비교
5를 한 칸 뒤로 이동
2를 첫 번째 자리에 넣는다.
3회전 결과
4
Key: 4(다섯 번째 자료)
네 번째 값 8과 비교
8을 한 칸 뒤로 이동
세 번째 값 6과 비교
6을 한 칸 뒤로 이동
두 번째 값 5와 비교
5를 한 칸 뒤로 이동
첫 번째 값 2와 비교
2는 그대로, 4를 두 번째 자리에 넣는다.
4회전 결과
오름차순
완성상태
2 4 5 6 8

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



https://jinhyy.tistory.com/9

장점

- 구현이 단순
- 대부분의 원소가 이미 정렬되어있는 경우 매우 효율적
  (정렬되어있을 때(최선의 경우) O(n))
- 적은 데이터에 유리
- 제자리 정렬
- 안정정렬

단점

- 오래걸린다

퀵 정렬 (Quick Sort)

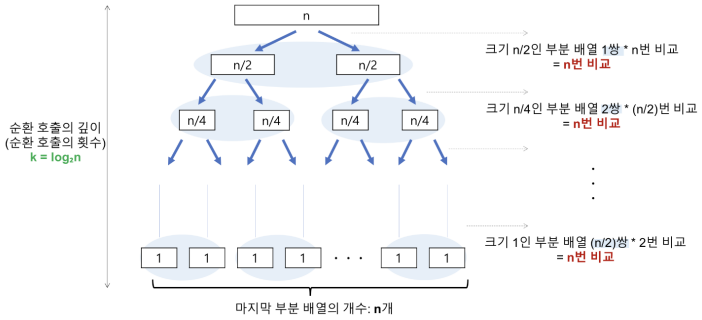
기준값(Pivot)을 기준으로 작은 데이터와 큰데이터를 나누어 정렬

정렬 알고리즘	평균 시간 복잡도	최선 시간 복잡도 (Best)	최악 시간 복잡도 (Worst)	메모리 사용량	안정성	방식
퀵 정렬 (Quick Sort)	O(n log n)	O(n log n)	O(n^2)	O(log n)	불안정	파티셔닝 (분할 정복)

특징

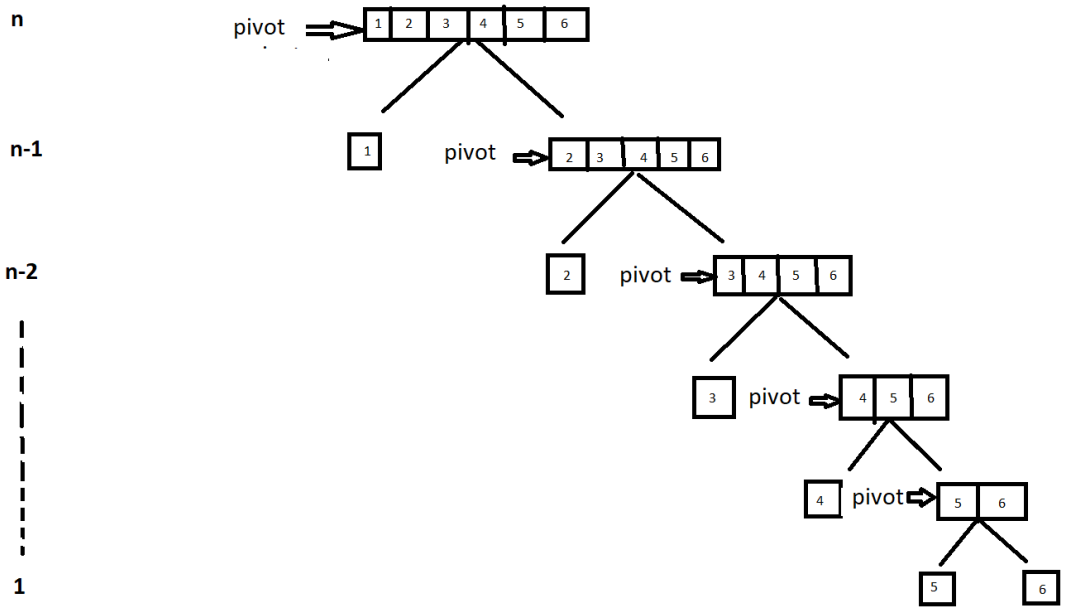
- 비교 기반
- 제자리 정렬
  (여기서 메모리 사용량이 O(log n)이 나오는 이유는 재귀함수 때문)
- 불안정 정렬
- 재귀적인
- 분할 정복 방법 사용
- 배열을 비균등하게 분할

- 정렬되지 않은 배열에는 빠르고,
  정렬된 배열에는 느림
  - 최선



https://gyoogle.dev/blog/algorithm/Quick Sort.html

  - 최악



https://iq.opengenus.org/worst-case-of-quick-sort/

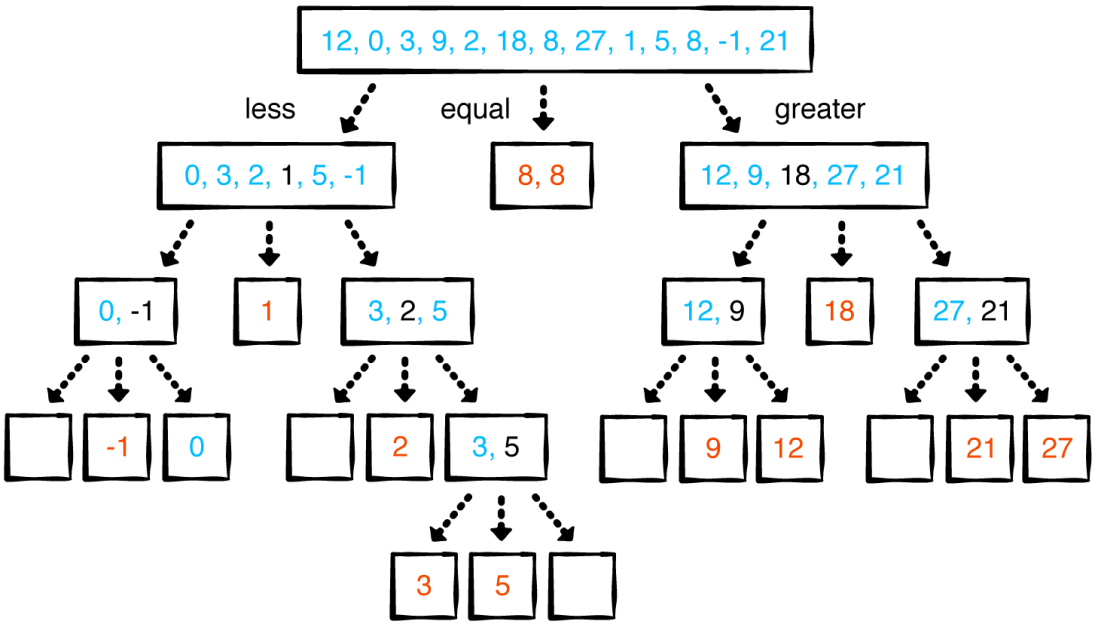
> 💡 **최악(O(n^2))이 발생하는 경우**
>
> → 재귀호출의 깊이가 깊어져서 발생
>
> - 최대, 최소값이 pivot으로 선정되었을 경우
> - 이미 정렬된 배열에서 pivot을 중간값을 선택하는 경우
>   (내림차순으로 정렬되어있는 배열에 대해서?)

동작 방식

> 💡 **퀵소트 종류**
>
> >> pivot의 기준을 어떤 것으로 잡느냐에 따라, 같은 퀵소트라도 큰 차이 발생!
>
> 1. pivot 위치를 맨 왼쪽으로 설정
> 2. pivot 위치를 맨 오른쪽으로 설정
> 3. pivot 위치를 중간으로 설정 (가장 보편적) → 최악의 경우 O(n^2) 발생
> 4. pivot의 위치를 랜덤으로 설정
>   (운에 기대어 평균값 기대)
>
> .... (아직 연구 중)

- pivot 위치를 중간으로 설정한 경우

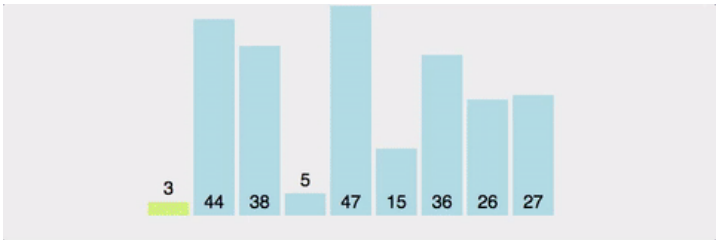
```
1. 주어진 배열 중간값(pivot)을 고른다
2. pivot 왼쪽에는 작은 값, pivot 오른쪽에는 큰 값이 올 수 있도록 배열 분할한다.(divide)
3. 여기서 pivot으로 값으로 선정된 값은 위치를 고정하고 검사하는 값에서 제외한다.
4. 분할한 배열들에 대하여 재귀적으로 1,2,3을 수행한다.
```



https://www.kodeco.com/books/data-structures-algorithms-in-swift/v3.0/chapters/34-quicksort

- pivot 위치를 맨 왼쪽으로 설정한 경우

https://ju-nam2.tistory.com/61



https://gyoogle.dev/blog/algorithm/Quick Sort.html

장점

- 빠르다
- 제자리 정렬
- 대규모 데이터에 유리

단점

- 정렬된 배열이라면 불균형 분할에 의해 더 오래걸림
- 불안정 정렬

병합 정렬 (Merge Sort)

배열을 반으로 나누고 각각을 정렬한 후 병합하는 방식의 정렬

정렬 알고리즘	평균 시간 복잡도	최선 시간 복잡도 (Best)	최악 시간 복잡도 (Worst)	메모리 사용량	안정성	방식
병합 정렬 (Merge Sort)	O(n log n)	O(n log n)	O(n log n)	O(n)	안정	병합 (분할 정복)

특징

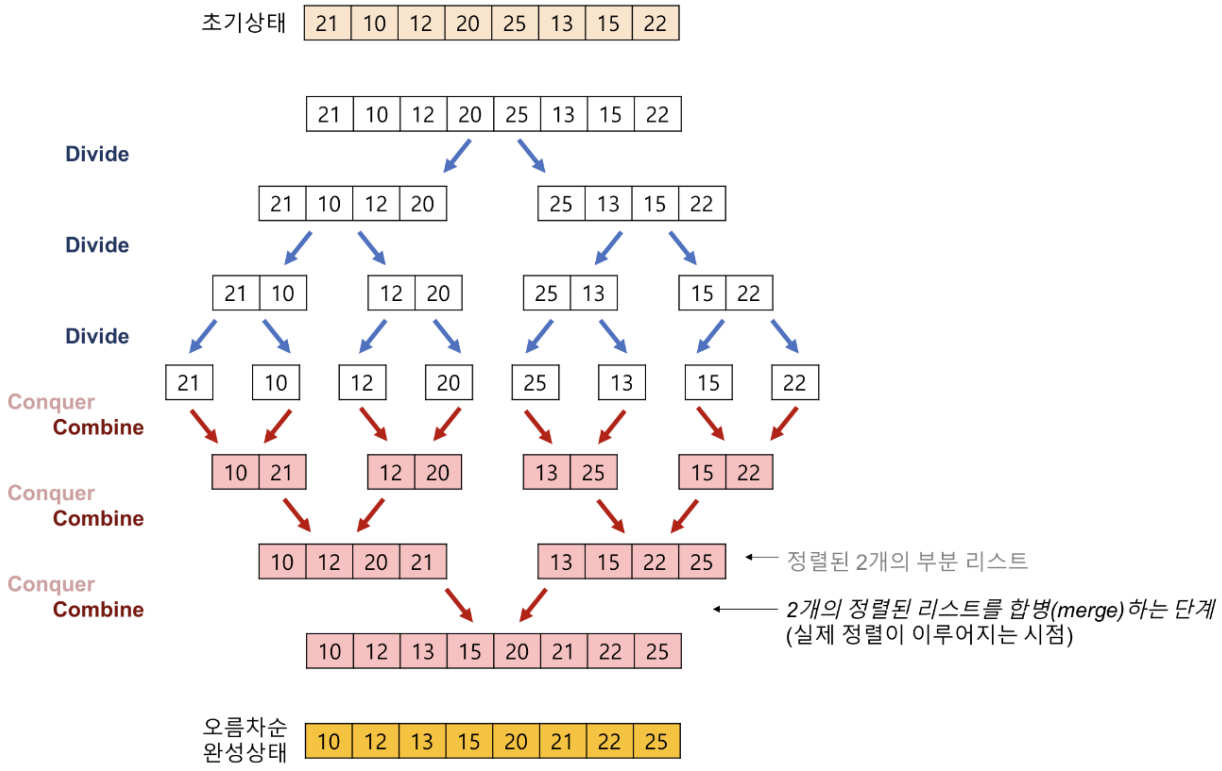
- 비교 기반
- 제자리가 아닌 정렬
  (병합 단계에서 추가적인 배열 사용)
- 안정 정렬
- 재귀적인
- 분할 정복 방법 사용
- 배열을 균등하게 분할
- linkedlist 정렬이 필요할 때 사용하면 효율적

💡 **퀵소트 vs 머지 소트**

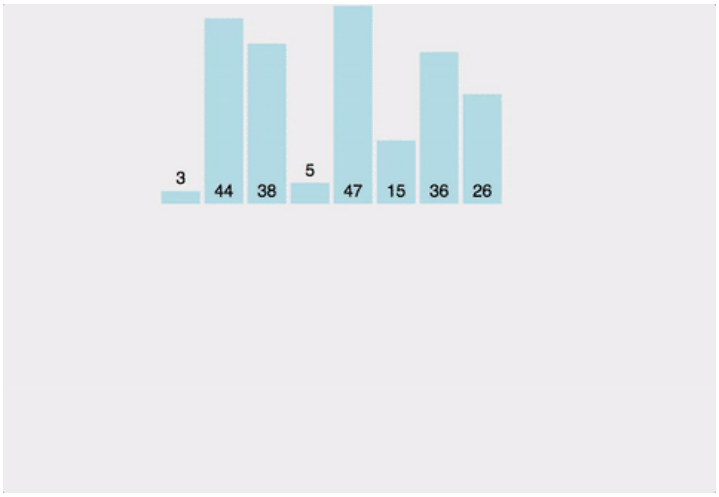
특징	퀵 소트 (Quick Sort)	머지 소트 (Merge Sort)
제자리 정렬	제자리 정렬	제자리 정렬이 아님
안정성	불안정 정렬	안정 정렬
활용	배열 (인덱스를 통해 비교하여 정렬)	링크드 리스트 (순차적으로 비교하여 정렬)
배열 분할	불균등하게 분할	균등하게 분할
정렬 동작 방식	pivot을 통해 정렬 후, 배열 분할	가장 작은 사이즈까지 배열 분할 후, 정렬

동작 방식

```
1. 가장 단위까지 배열을 반으로 계속 쪼갠다(divide)
2. 부분 배열의 데이터값을 순차적으로 비교하면서 정렬한다 (conquer)
3. 정렬된 부분 배열들을 하나의 배열에 병합한다 (combine)
4. 모든 부분 배열들을 2,3을 통해 하나로 합쳐질 때까지 반복한다
```



https://gmlwjd9405.github.io/2018/05/08/algorithm-merge-sort.html



https://jinhyy.tistory.com/9

장점

- 빠르다
- 링크드리스트 정렬에 효율적
- 안정 정렬
- 대규모 데이터에 유리

단점

- 제자리 정렬이 아니기 때문에 추가적인 메모리가 필요하다

힙 정렬 (Heap Sort)

힙(Heap) 자료구조를 기반으로한 정렬 방식

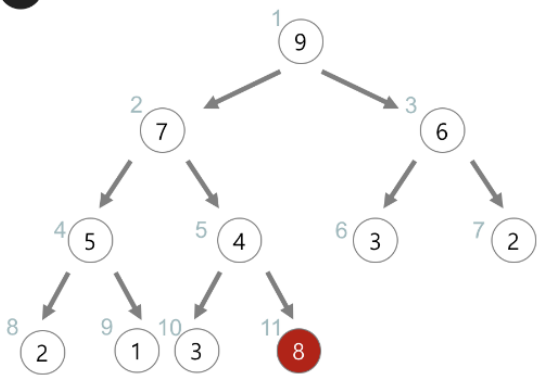
정렬 알고리즘	평균 시간 복잡도	최선 시간 복잡도 (Best)	최악 시간 복잡도 (Worst)	메모리 사용량	안정성	방식
힙 정렬 (Heap Sort)	O(n log n)	O(n log n)	O(n log n)	O(1)	불안정	선택

특징

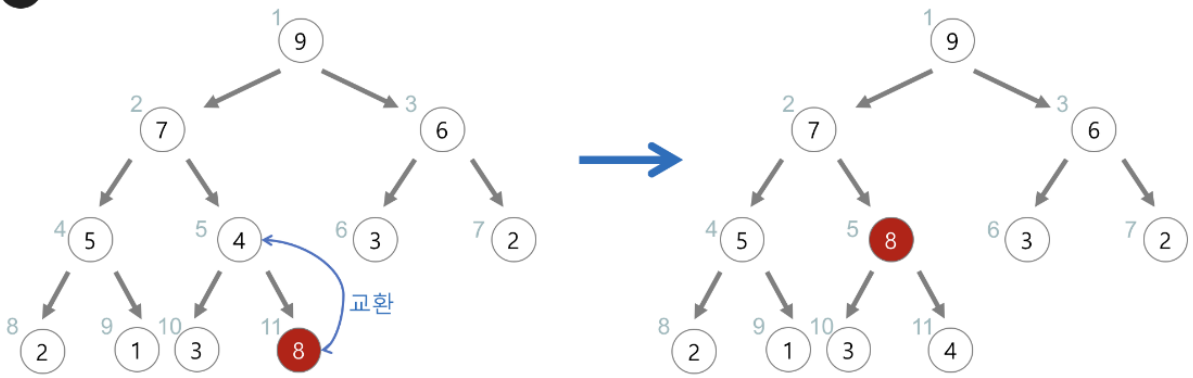
- 비교 기반
- 제자리 정렬
- 불안정 정렬
- 재귀적인
- 힙 자료구조 사용
- 최대값, 최소값 구할 때 용이

동작방식

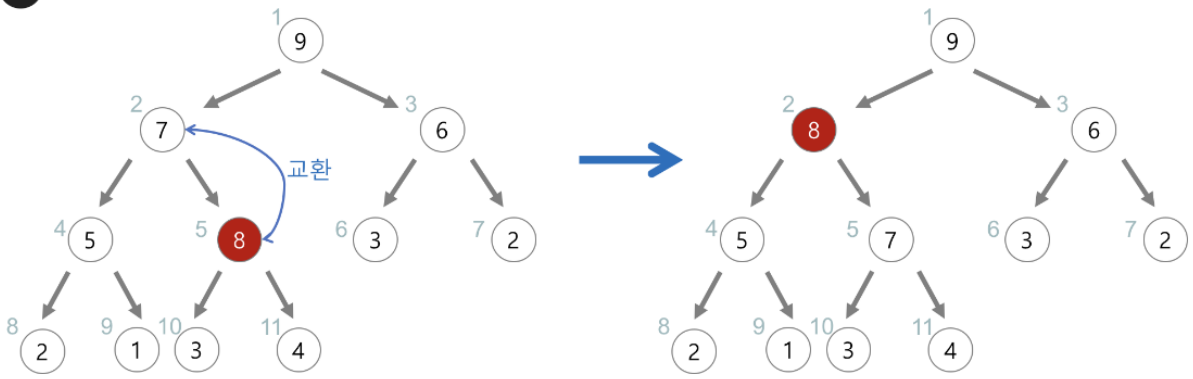
1 인덱스순으로 가장 마지막 위치에 이어서 새로운 요소 8을 삽입



2 부모 노드 4 < 삽입 노드 8 이므로 서로 교환

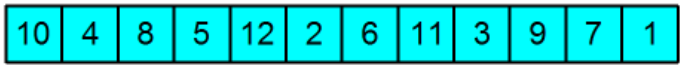


3 부모 노드 7 < 삽입 노드 8 이므로 서로 교환



4 부모 노드 9 > 삽입 노드 8 이므로 더 이상 교환하지 않는다.

https://gmlwjd9405.github.io/2018/05/10/algorithm-heap-sort.html



https://commons.wikimedia.org/wiki/File:Heap_sort_example.gif

장점

- 빠르다
- 최대값, 최소값 구할 때 용이
- 제자리 정렬
- 대규모 데이터에 유리

단점

- 불안정 정렬

JAVA에서의 정렬 구현?

Arrays.sort()

- DualPivotQuicksort 사용

```
Sorts the specified array into ascending numerical order.
Params:          a – the array to be sorted
Implementation Note:  The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on all data sets, and is typically faster than traditional (one-pivot) Quicksort implementations.
```

```java
@Contract(mutates = "param1")
public static void sort( @NotNull int[] a) {
    DualPivotQuicksort.sort(a,  parallelism: 0,  low: 0, a.length);
}
```

- 두 개의 pivot를 사용하는 퀵소트 (3분할) → 정렬 속도 향상

Collections.sort()

- TimSort 사용

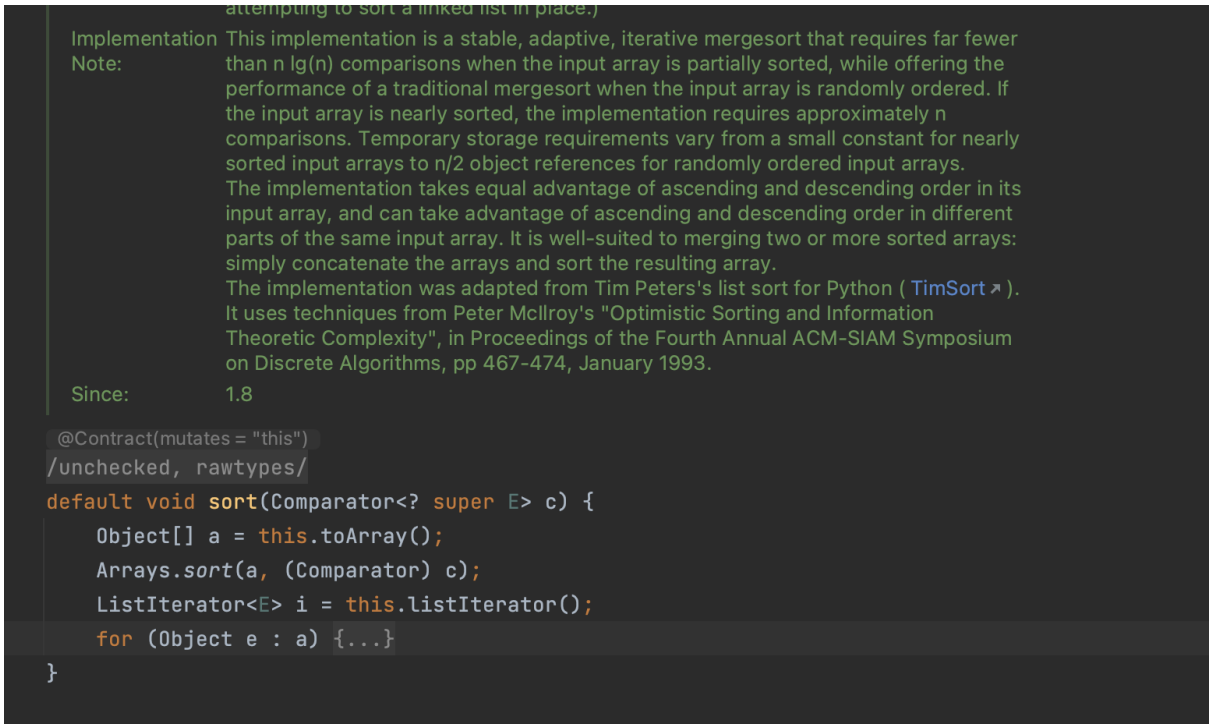
```
Sorts the specified list into ascending order, according to the natural ordering of its elements. All elements in the list must implement the Comparable interface. Furthermore, all elements in the list must be mutually comparable (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the list).

This sort is guaranteed to be stable: equal elements will not be reordered as a result of the sort.

The specified list must be modifiable, but need not be resizable.

Params:          list – the list to be sorted.
Throws:          ClassCastException – if the list contains elements that are not mutually comparable (for example, strings and integers).
                 UnsupportedOperationException – if the specified list's list-iterator does not support the set operation.
                 IllegalArgumentException – (optional) if the implementation detects that the natural ordering of the list elements is found to violate the Comparable contract
Implementation Note:  This implementation defers to the List.sort(Comparator) method using the specified list and a null comparator.
See Also:        List.sort(Comparator)
```

```java
@Contract(mutates = "param1")
/unchecked/
public static <T extends Comparable<? super T>> void sort( @NotNull List<T> list) {
    list.sort( c: null);
}
```


attempting to sort a linked list in place.)
Implementation Note:
This implementation is a stable, adaptive, iterative mergesort that requires far fewer than n lg(n) comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately n comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to n/2 object references for randomly ordered input arrays.
The implementation takes equal advantage of ascending and descending order in its input array, and can take advantage of ascending and descending order in different parts of the same input array. It is well-suited to merging two or more sorted arrays: simply concatenate the arrays and sort the resulting array.
The implementation was adapted from Tim Peters's list sort for Python (TimSort). It uses techniques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.
Since:
1.8
@Contract(mutates = "this")
/unchecked, rawtypes/
default void sort(Comparator<? super E> c) {
Object[] a = this.toArray();
Arrays.sort(a, (Comparator) c);
ListIterator<E> i = this.listIterator();
for (Object e : a) {...}
}

- 삽입 정렬 + 병합 정렬