퀵 정렬

작성일시	@2023년 6월 15일 오전 3:08
강의 번호	Algorithm
유형	
자료	
복습	☐
Spring Framwork	

퀵 정렬은 구현방법이 정말 다양하다.
퀵 정렬은 '비교'하면서 찾기 때문에 '비교 정렬'이다.
또한 '제자리 정렬'이다.
피벗을 두고 두개의 부분리스트를 만들 때 서로 떨어진 원소끼리 교환이 일어나기 때문에 불안정정렬이다.

퀵 정렬의 메커니즘

1. 하나의 리스트를 '피벗'을 기준으로 두 개의 '부분 리스트'로 나눈다.
2. 하나는 피벗보다 작은 값들의 부분리스트
3. 다른 하나는 피벗보다 큰 값들의 부분리스트로 정렬
4. 각 부분리스트에 대해 다시 위처럼 재귀적으로 수행

위의 과정은 '분할 정복'이다.
병합 정렬 또한 '분할 정복'의 방식이지만,
퀵 정렬은 피벗을 기준으로 수행하며
병합정렬은 하나의 리스트를 절반으로 나누어 분할 정복을 수행한다.

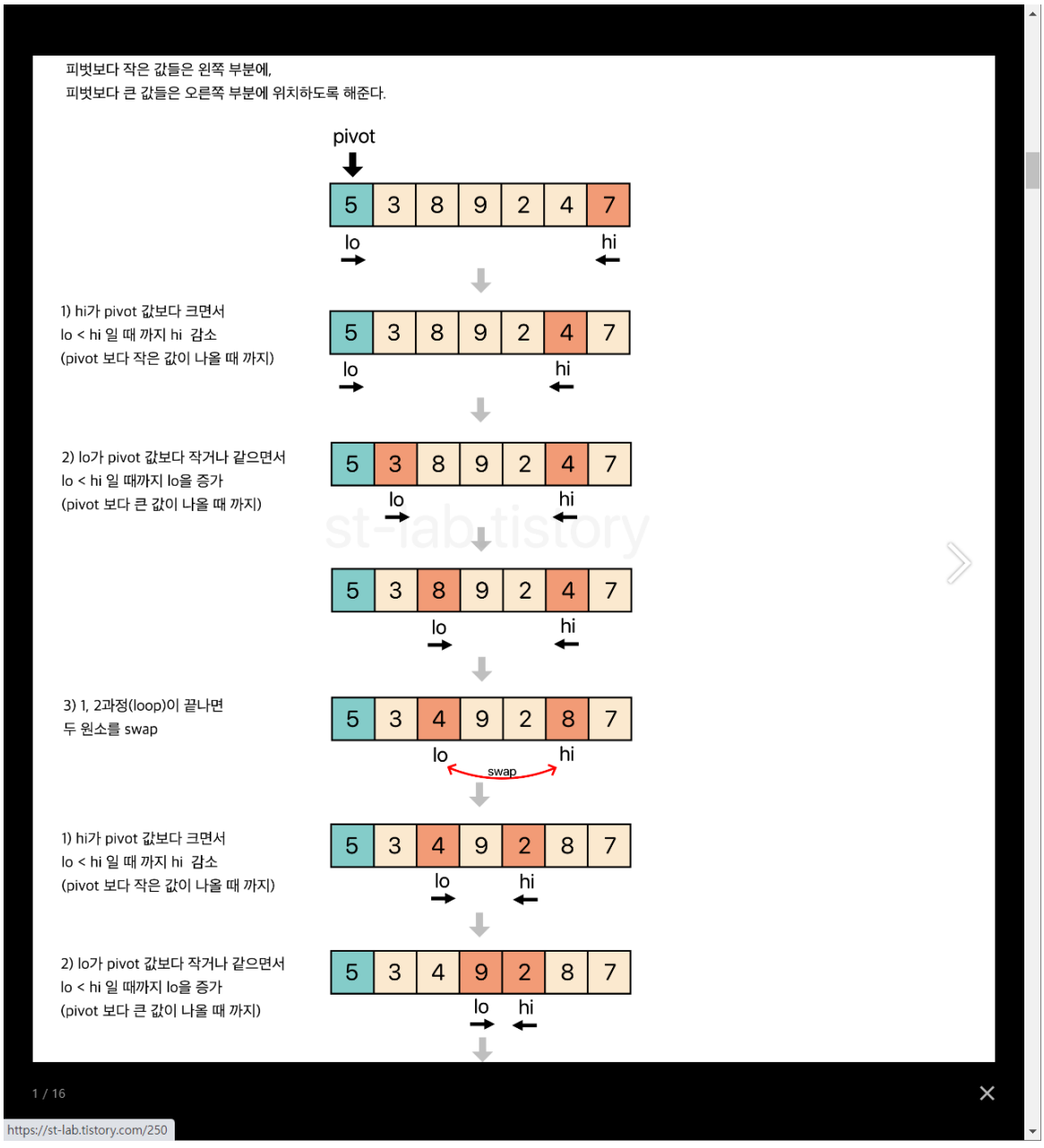
전체적인 과정
1. 피벗을 하나 선택
2. 피벗을 기준으로 양쪽에서 피벗보다 큰값, 혹은 작은 값을 찾는다.
3. 양 방향에서 찾은 두 원소를 교환한다.
4. 왼쪽에서 탐색하는 위치와 오른쪽에서 탐색하는 위치가 엇갈리지 않을 때까지 2번으로 돌아가 위 과정을 반복한다.
5. 엇갈린 기점을 기준으로 두 개의 부분 리스트로 나누어 1번으로 돌아가 해당 부분리스트의 길이가 1이 아닐때까지 1번 과정을 반복한다.
6. 인접한 부분리스트끼리 합친다.

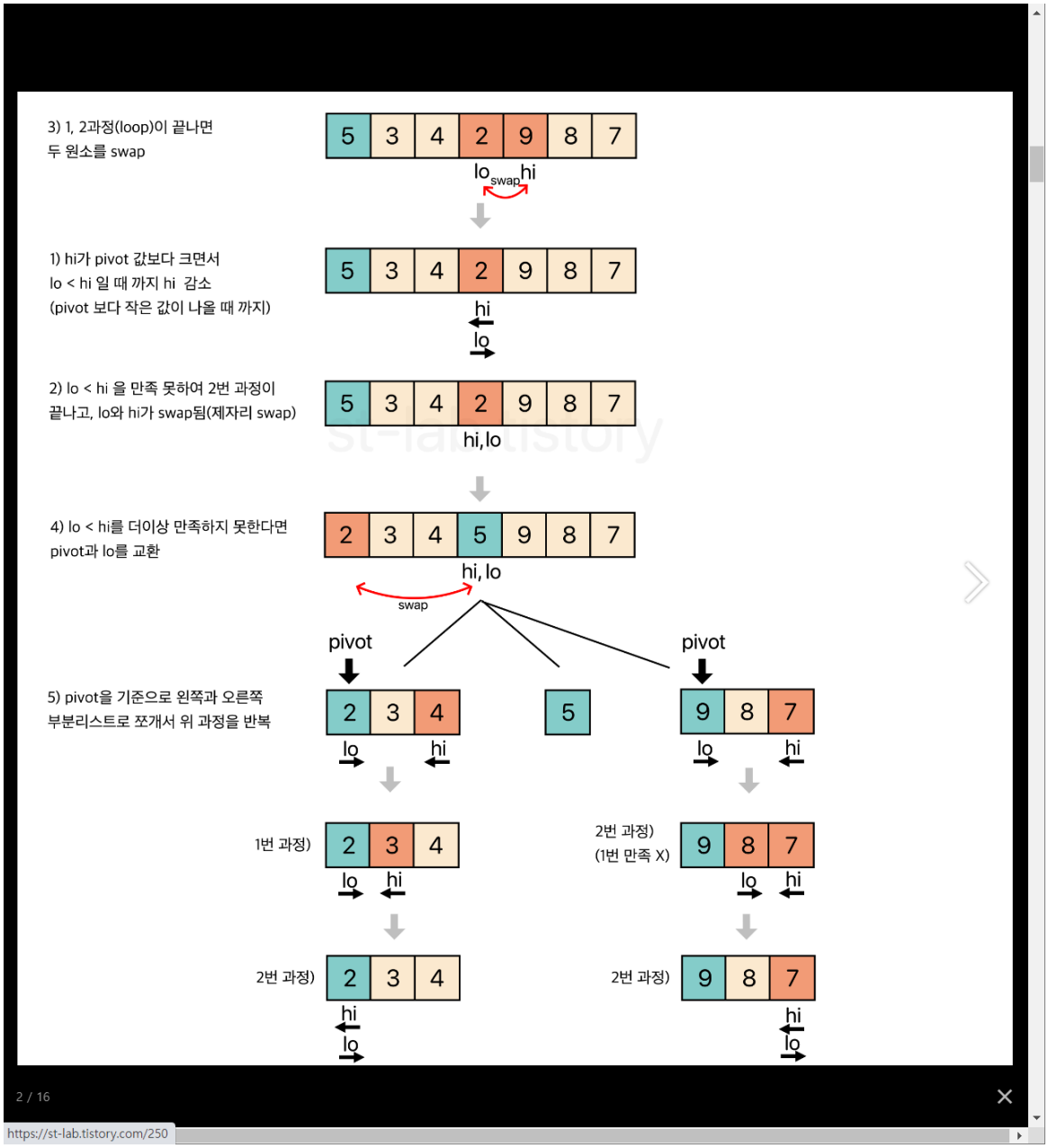
장점
1.특정 상태가 아닌 이상 평균 시간 복잡도는 NlogN이며, 다른 NlogN알고리즘에 비해 속도가 매우 빠르다. 유사한 분할정복 방식인 병합정렬에 비해 2~3배 정도 빠르다.
2. 추가적인 별도의 메모리를 필요로 하지않고 재귀 호출 스택프레임에 의한 공간복잡도는 logN으로 메모리를 적게 소비한다.

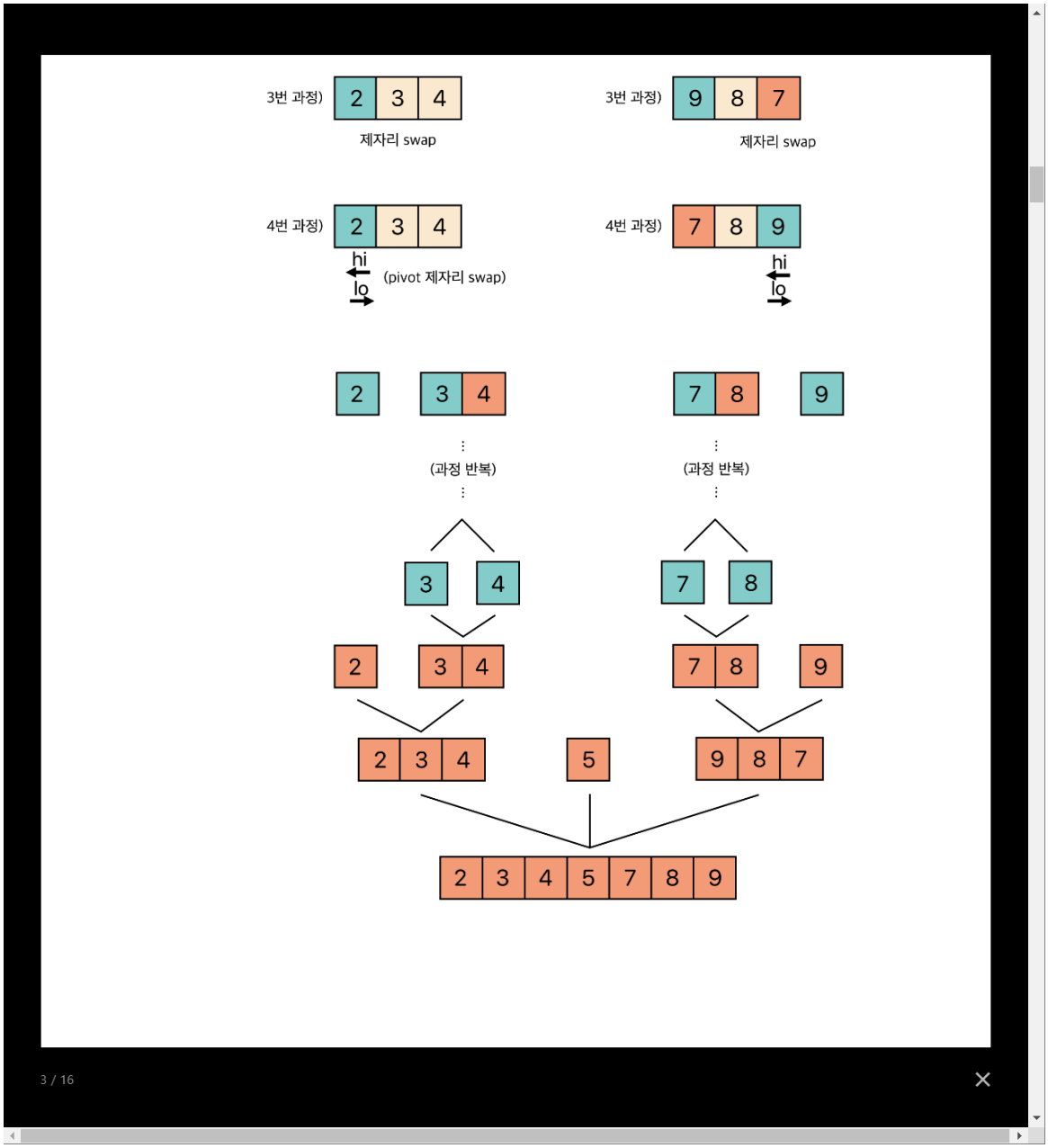
단점

1. 특정 조건하에 성능이 급격하게 떨어진다.
2. 재귀를 사요하기 때문에 사용하지 못하는 환경일 경우 그 구현이 매우 복잡하다.

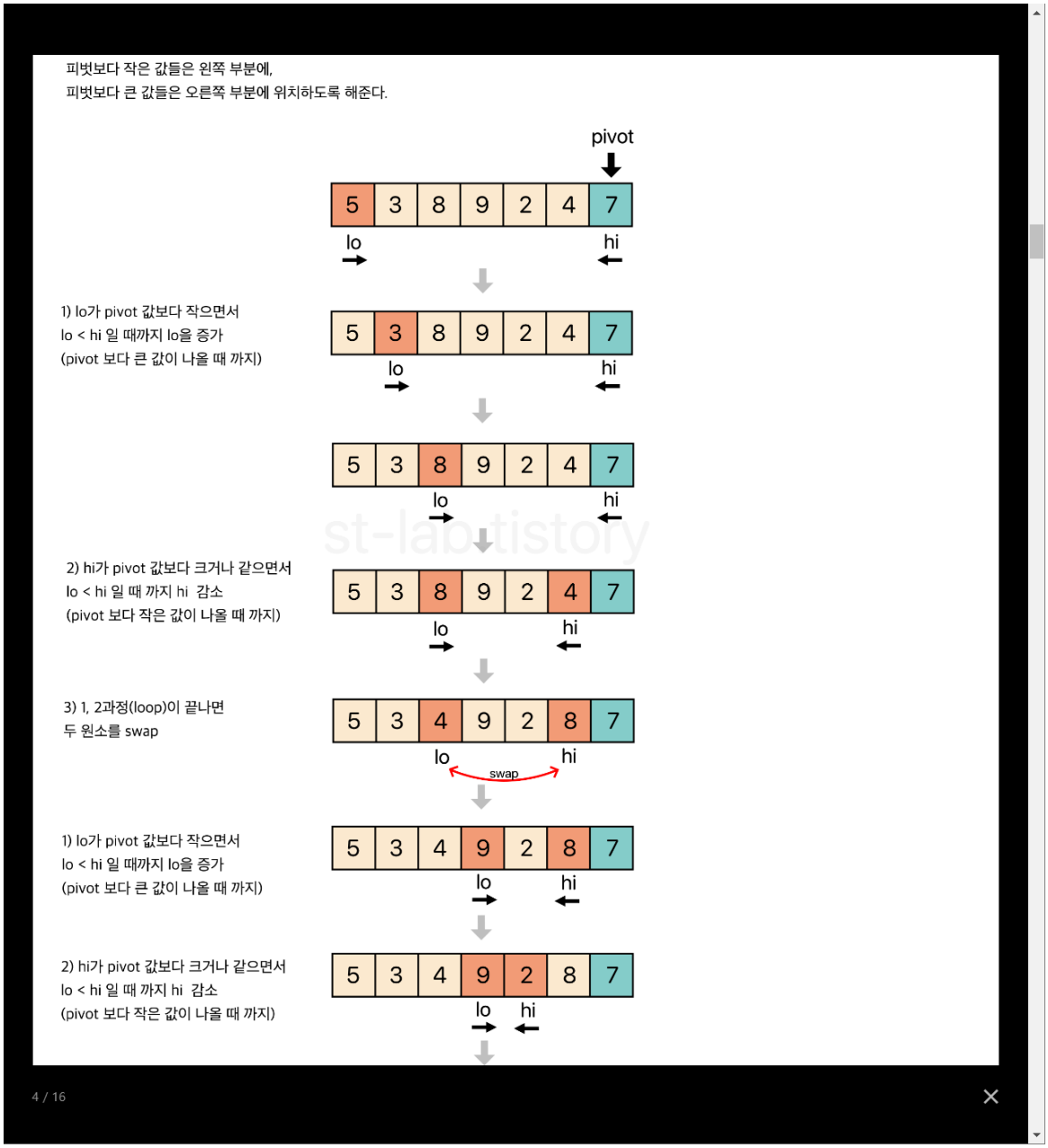
..왼쪽 피벗 선택 방식

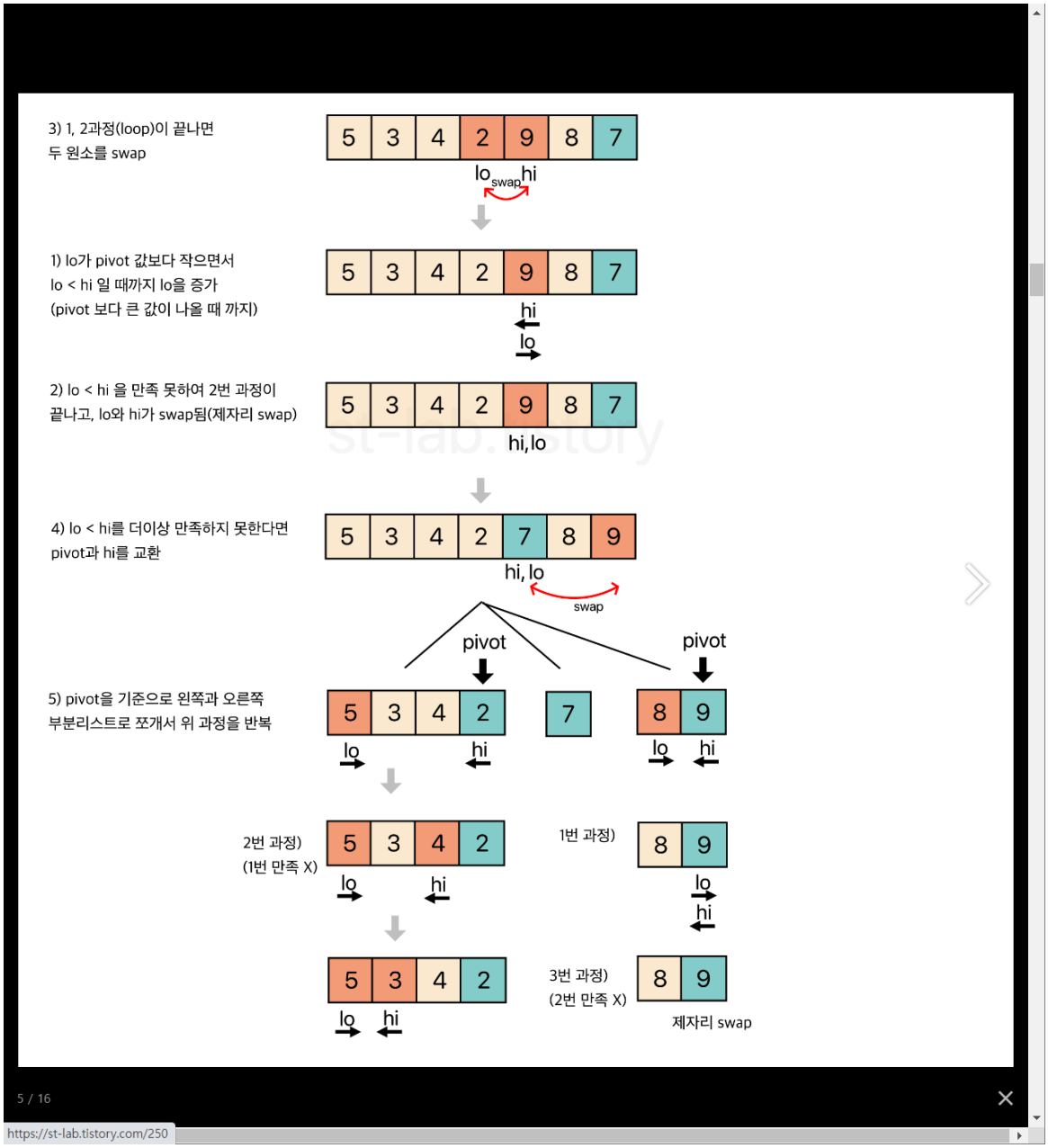


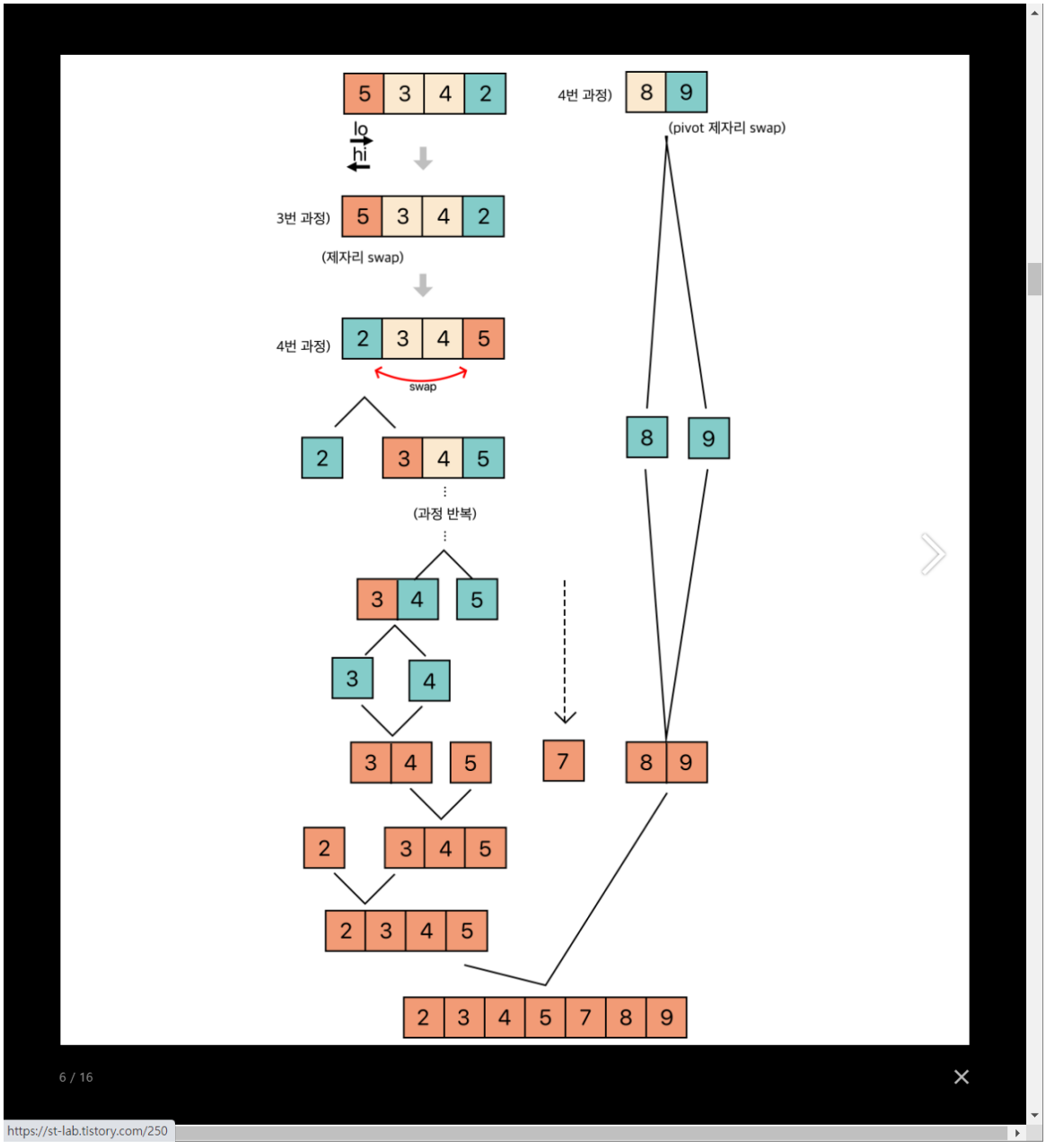

3) 1, 2과정(loop)이 끝나면
두 원소를 swap
5 3 4 2 9 8 7
lo swap hi
1) hi가 pivot 값보다 크면서
lo < hi 일 때 까지 hi 감소
(pivot 보다 작은 값이 나올 때 까지)
5 3 4 2 9 8 7
hi
lo
2) lo < hi 을 만족 못하여 2번 과정이
끝나고, lo와 hi가 swap됨(제자리 swap)
5 3 4 2 9 8 7
hi, lo
4) lo < hi를 더이상 만족하지 못한다면
pivot과 lo를 교환
2 3 4 5 9 8 7
hi, lo
swap
5) pivot을 기준으로 왼쪽과 오른쪽
부분리스트로 쪼개서 위 과정을 반복
pivot
2 3 4
lo hi
5
pivot
9 8 7
lo hi
1번 과정)
2 3 4
lo hi
2번 과정)
2 3 4
hi
lo
2번 과정)
(1번 만족 X)
9 8 7
lo hi
2번 과정)
9 8 7
hi
lo
2 / 16
https://st-lab.tistory.com/250


3번 과정)
2 3 4
제자리 swap
3번 과정)
9 8 7
제자리 swap
4번 과정)
2 3 4
hi
lo
(pivot 제자리 swap)
4번 과정)
7 8 9
hi
lo
2
3 4
(과정 반복)
7 8
9
(과정 반복)
3 4
7 8
2 3 4
7 8 9
2 3 4
5
9 8 7
2 3 4 5 7 8 9
3 / 16

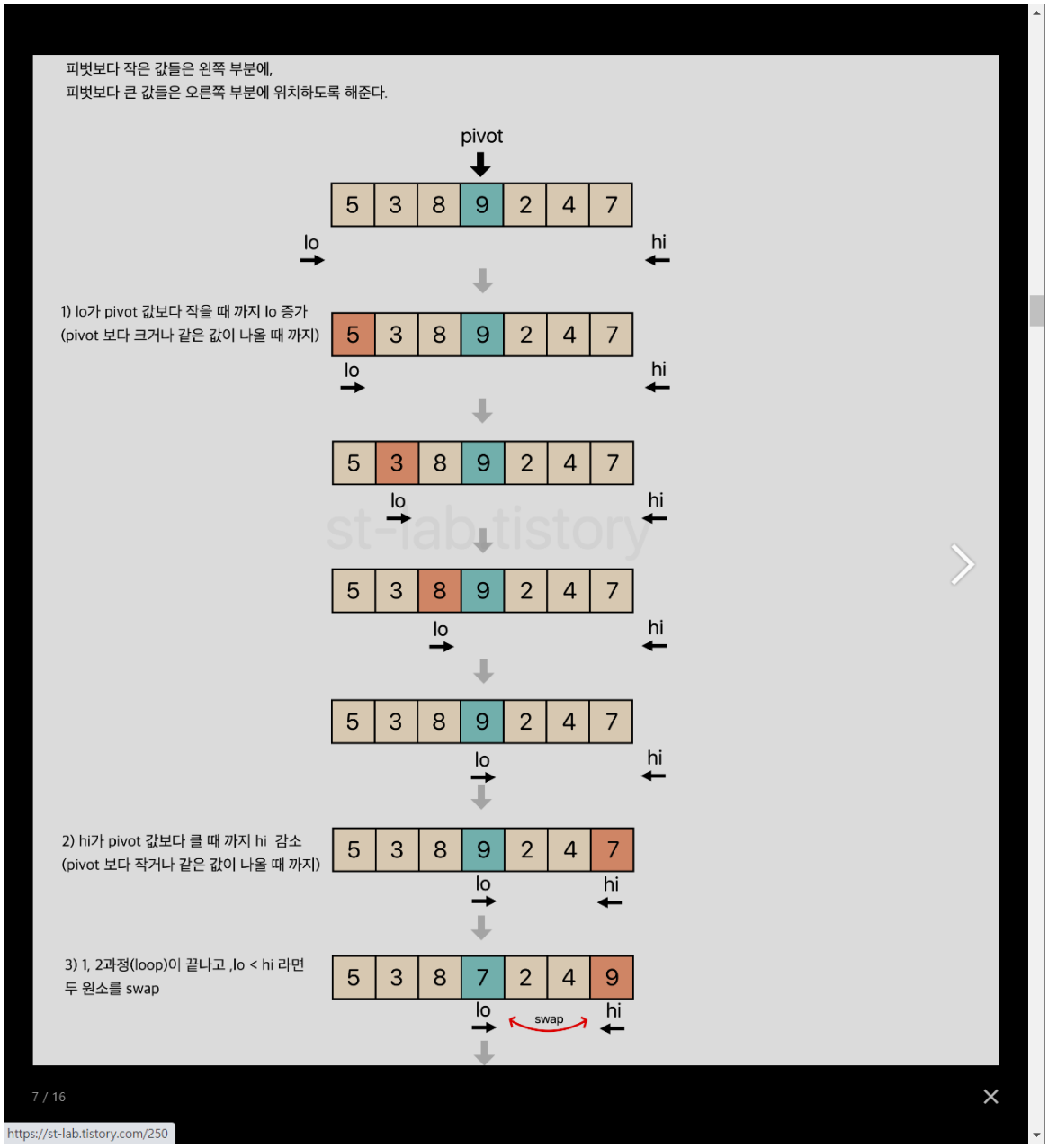
..오른쪽 피벗 선택 방식

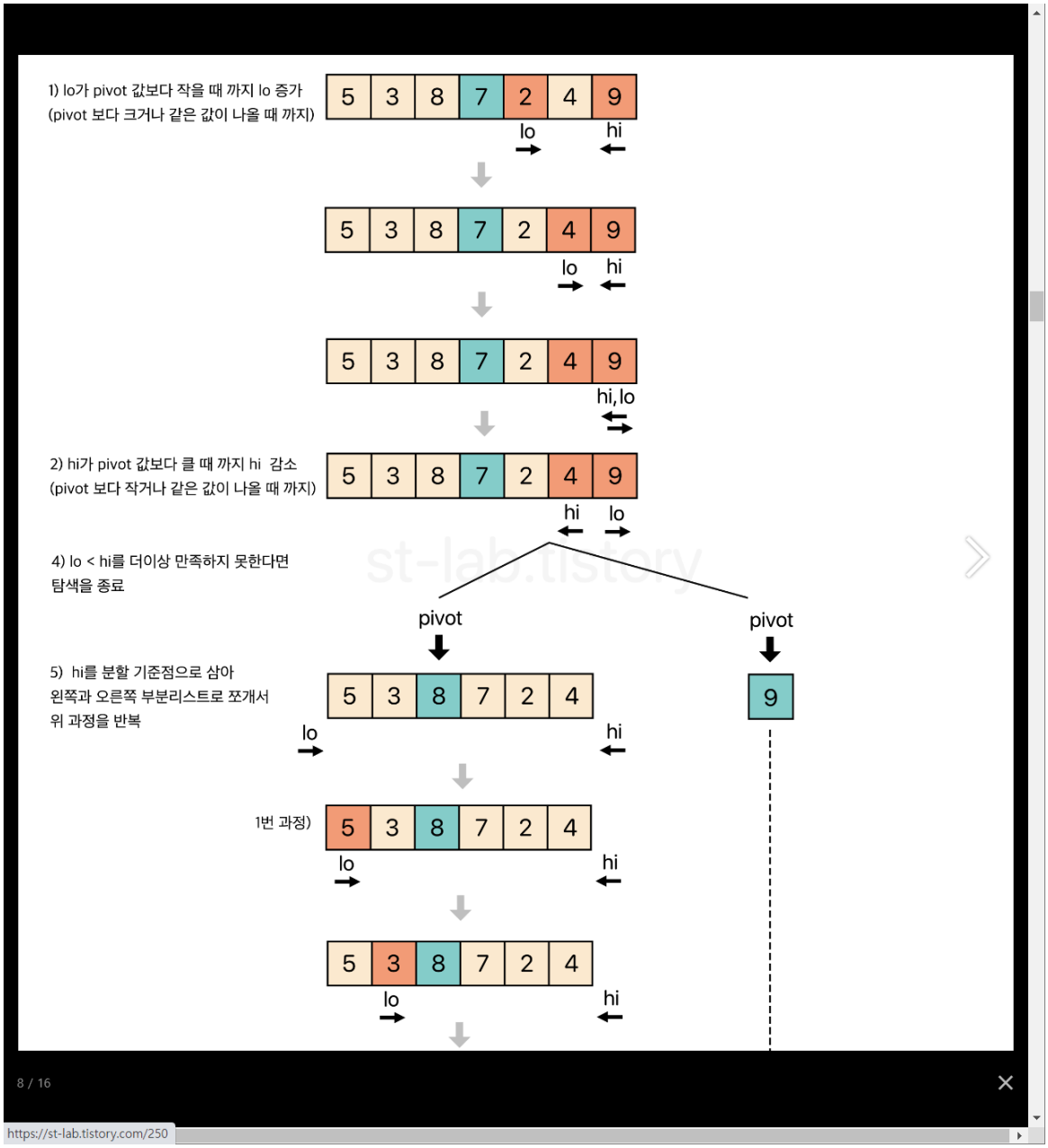

피벗보다 작은 값들은 왼쪽 부분에,
피벗보다 큰 값들은 오른쪽 부분에 위치하도록 해준다.
pivot
5 3 8 9 2 4 7
lo
hi
1) lo가 pivot 값보다 작으면서
lo < hi 일 때까지 lo을 증가
(pivot 보다 큰 값이 나올 때 까지)
5 3 8 9 2 4 7
lo
hi
5 3 8 9 2 4 7
lo
hi
st-lab.tistory
2) hi가 pivot 값보다 크거나 같으면서
lo < hi 일 때 까지 hi 감소
(pivot 보다 작은 값이 나올 때 까지)
5 3 8 9 2 4 7
lo
hi
3) 1, 2과정(loop)이 끝나면
두 원소를 swap
5 3 4 9 2 8 7
lo
hi
swap
1) lo가 pivot 값보다 작으면서
lo < hi 일 때까지 lo을 증가
(pivot 보다 큰 값이 나올 때 까지)
5 3 4 9 2 8 7
lo
hi
2) hi가 pivot 값보다 크거나 같으면서
lo < hi 일 때 까지 hi 감소
(pivot 보다 작은 값이 나올 때 까지)
5 3 4 9 2 8 7
lo
hi
4 / 16

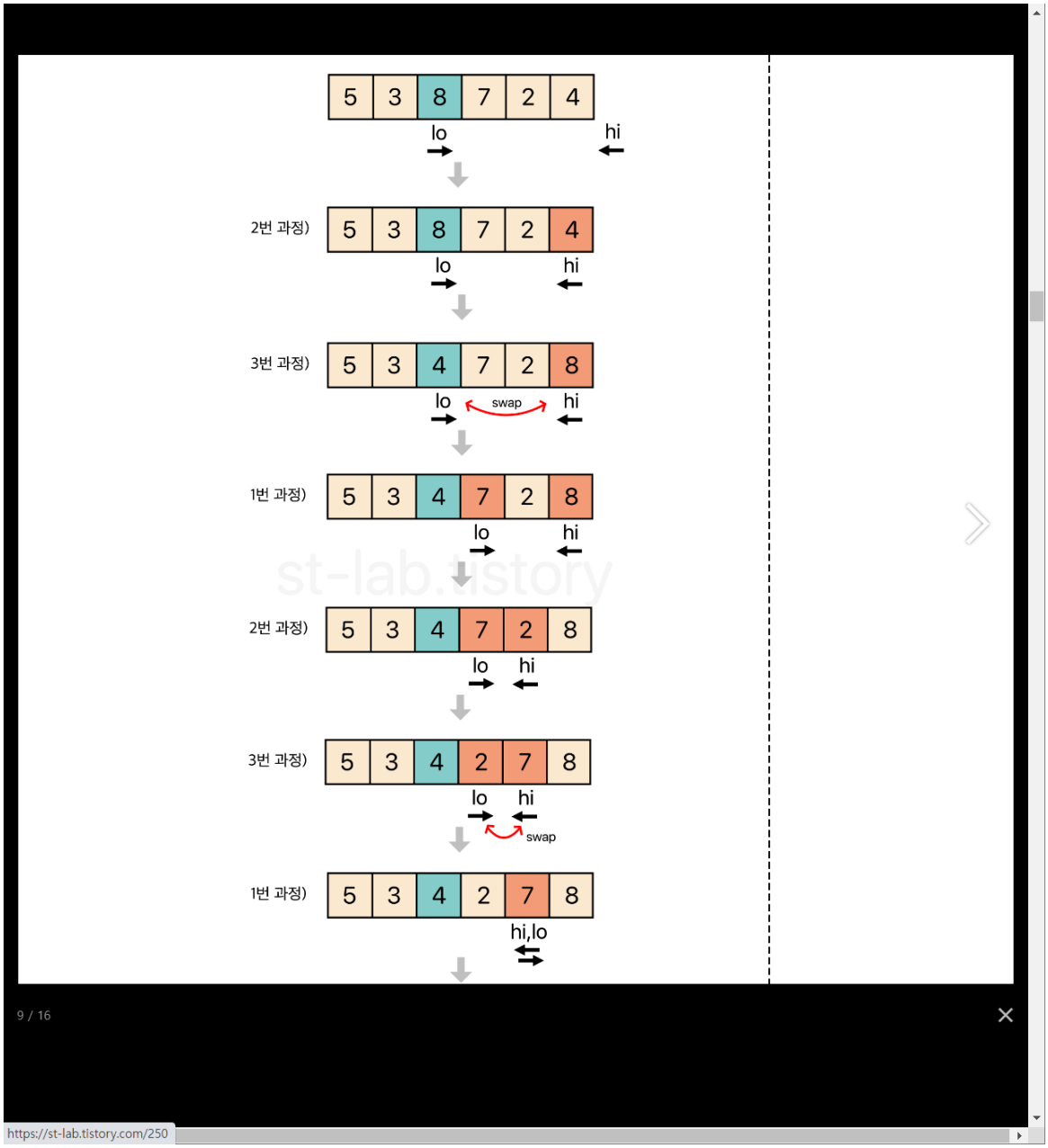

3) 1, 2과정(loop)이 끝나면
두 원소를 swap
5 3 4 2 9 8 7
lo swap hi
1) lo가 pivot 값보다 작으면서
lo < hi 일 때까지 lo을 증가
(pivot 보다 큰 값이 나올 때 까지)
5 3 4 2 9 8 7
hi
lo
2) lo < hi 을 만족 못하여 2번 과정이
끝나고, lo와 hi가 swap됨(제자리 swap)
5 3 4 2 9 8 7
hi, lo
4) lo < hi를 더이상 만족하지 못한다면
pivot과 hi를 교환
5 3 4 2 7 8 9
hi, lo
swap
5) pivot을 기준으로 왼쪽과 오른쪽
부분리스트로 쪼개서 위 과정을 반복
pivot
pivot
5 3 4 2
7
8 9
lo hi
lo hi
2번 과정)
(1번 만족 X)
5 3 4 2
lo hi
1번 과정)
8 9
lo
hi
5 3 4 2
lo hi
3번 과정)
(2번 만족 X)
8 9
제자리 swap
5 / 16
https://st-lab.tistory.com/250

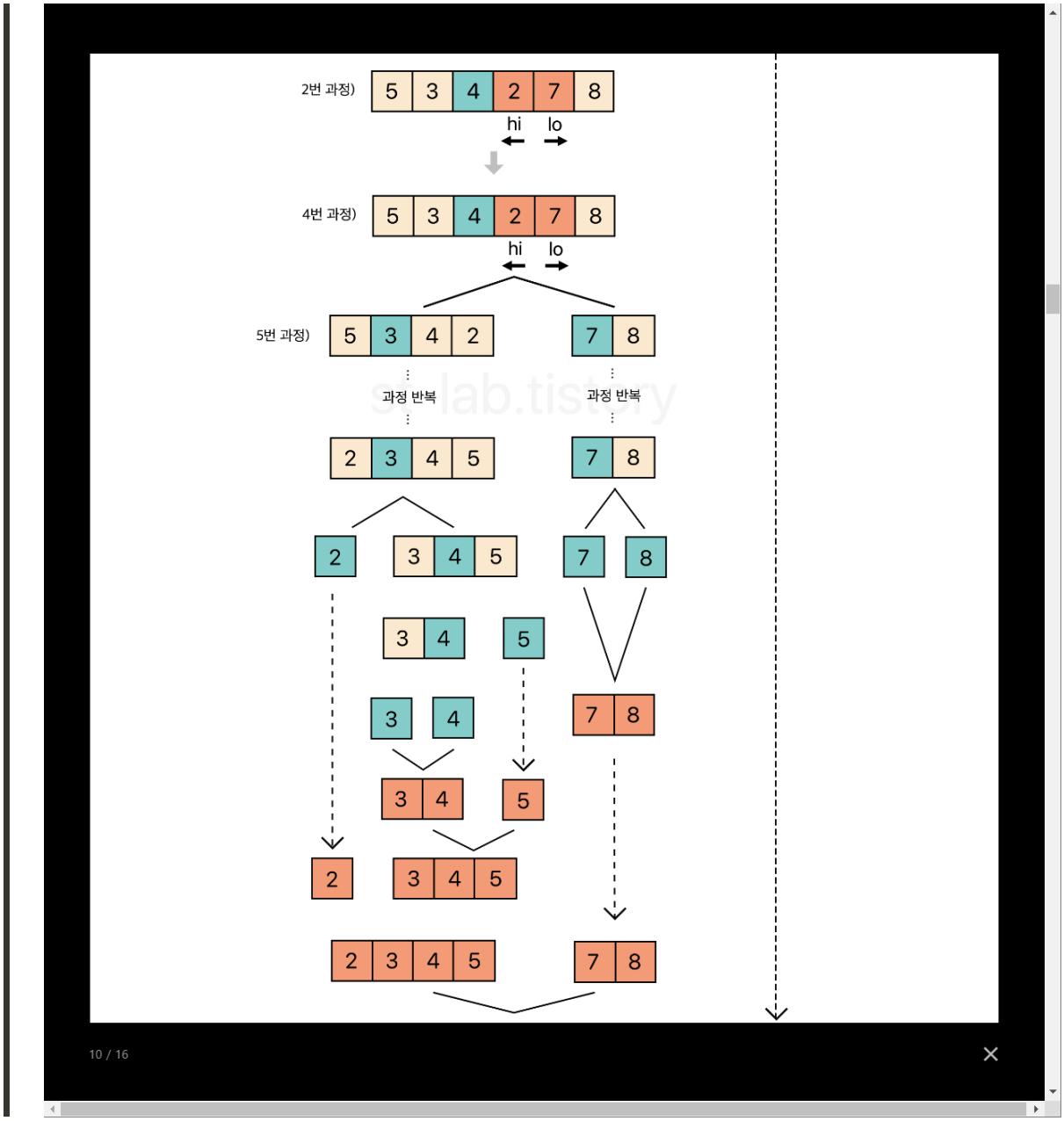

5 3 4 2
lo
hi
3번 과정)
5 3 4 2
(제자리 swap)
4번 과정)
2 3 4 5
swap
2
3 4 5
(과정 반복)
3 4
5
3
4
3 4
5
2
3 4 5
2 3 4 5
4번 과정)
8 9
(pivot 제자리 swap)
8
9
7
8 9
2 3 4 5 7 8 9
6 / 16
https://st-lab.tistory.com/250

..중앙피벗선택 방식


피벗보다 작은 값들은 왼쪽 부분에,
피벗보다 큰 값들은 오른쪽 부분에 위치하도록 해준다.
pivot
5 3 8 9 2 4 7
lo
hi
1) lo가 pivot 값보다 작을 때 까지 lo 증가
(pivot 보다 크거나 같은 값이 나올 때 까지)
5 3 8 9 2 4 7
lo
hi
5 3 8 9 2 4 7
lo
hi
st-lab.tistory
5 3 8 9 2 4 7
lo
hi
5 3 8 9 2 4 7
lo
hi
2) hi가 pivot 값보다 클 때 까지 hi 감소
(pivot 보다 작거나 같은 값이 나올 때 까지)
5 3 8 9 2 4 7
lo
hi
3) 1, 2과정(loop)이 끝나고 ,lo < hi 라면
두 원소를 swap
5 3 8 7 2 4 9
lo
swap
hi
7 / 16
https://st-lab.tistory.com/250


1) lo가 pivot 값보다 작을 때 까지 lo 증가
(pivot 보다 크거나 같은 값이 나올 때 까지)
5 3 8 7 2 4 9
lo
hi
5 3 8 7 2 4 9
lo hi
5 3 8 7 2 4 9
hi, lo
2) hi가 pivot 값보다 클 때 까지 hi 감소
(pivot 보다 작거나 같은 값이 나올 때 까지)
5 3 8 7 2 4 9
hi lo
4) lo < hi를 더이상 만족하지 못한다면
탐색을 종료
pivot
pivot
5) hi를 분할 기준점으로 삼아
왼쪽과 오른쪽 부분리스트로 쪼개서
위 과정을 반복
5 3 8 7 2 4
9
lo
hi
1번 과정)
5 3 8 7 2 4
lo
hi
5 3 8 7 2 4
lo
hi
8 / 16
https://st-lab.tistory.com/250


5 3 8 7 2 4
lo
hi
2번 과정)
5 3 8 7 2 4
lo
hi
3번 과정)
5 3 4 7 2 8
lo
swap
hi
1번 과정)
5 3 4 7 2 8
lo
hi
st-lab.tistory
2번 과정)
5 3 4 7 2 8
lo
hi
3번 과정)
5 3 4 2 7 8
lo
hi
swap
1번 과정)
5 3 4 2 7 8
hi,lo
9 / 16
https://st-lab.tistory.com/250



위와 같이 피벗보다 작은 값을 왼쪽, 큰 값을 오른쪽에두는 것을 파티션이이라고 한다.
파티셔닝을 통해 배치된 피벗의 위치를 기준으로 좌 우 부분리스트로 나누어 각각의 리스트에 대해 재귀호출을 해주면된다.

왼쪽피벗 선택 방식 구현

```
public class QuickSort {

  public static void sort(int[] a) {
    l_pivot_sort(a, 0, a.length - 1);
  }

  /**
   *  왼쪽 피벗 선택 방식
   * @param a   정렬할 배열
   * @param lo  현재 부분배열의 왼쪽
   * @param hi  현재 부분배열의 오른쪽
   */
  private static void l_pivot_sort(int[] a, int lo, int hi) {

    /*
     *  lo가 hi보다 크거나 같다면 정렬 할 원소가
     *  1개 이하이므로 정렬하지 않고 return한다.
     */
    if(lo >= hi) {
      return;
    }

    /*
     * 피벗을 기준으로 요소들이 왼쪽과 오른쪽으로 약하게 정렬 된 상태로
     * 만들어 준 뒤, 최종적으로 pivot의 위치를 얻는다.
     *
     * 그리고나서 해당 피벗을 기준으로 왼쪽 부분리스트와 오른쪽 부분리스트로 나누어
     * 분할 정복을 해준다.
     *
     * [과정]
     *
     * Partitioning:
     *
     *   a[left]          left part              right part
     * +-------------------------------------------------------+
     * |  pivot  |    element <= pivot    |    element > pivot   |
     * +-------------------------------------------------------+
     *
     *
     *  result After Partitioning:
     *
     *         left part          a[lo]          right part
     * +-------------------------------------------------------+
     * |   element <= pivot    |  pivot  |    element > pivot    |
     * +-------------------------------------------------------+
     *
     *
     *  result : pivot = lo
     *
     *
     *  Recursion:
     *
     * l_pivot_sort(a, lo, pivot - 1)     l_pivot_sort(a, pivot + 1, hi)
     *
```

```
     *         left part                          right part
     * +-----------------------+             +-----------------------+
     * |   element <= pivot    |    pivot    |    element > pivot    |
     * +-----------------------+             +-----------------------+
     * lo                pivot - 1        pivot + 1                hi
     *
     */
    int pivot = partition(a, lo, hi);

    l_pivot_sort(a, lo, pivot - 1);
    l_pivot_sort(a, pivot + 1, hi);
  }



  /**
   * pivot을 기준으로 파티션을 나누기 위한 약한 정렬 메소드
   *
   * @param a   정렬 할 배열
   * @param left  현재 배열의 가장 왼쪽 부분
   * @param right 현재 배열의 가장 오른쪽 부분
   * @return    최종적으로 위치한 피벗의 위치(lo)를 반환
   */
  private static int partition(int[] a, int left, int right) {

    int lo = left;
    int hi = right;
    int pivot = a[left];    // 부분리스트의 왼쪽 요소를 피벗으로 설정

    // lo가 hi보다 작을 때 까지만 반복한다.
    while(lo < hi) {

      /*
       * hi가 lo보다 크면서, hi의 요소가 pivot보다 작거나 같은 원소를
       * 찾을 때 까지 hi를 감소시킨다.
       */
      while(a[hi] > pivot && lo < hi) {
        hi--;
      }

      /*
       * hi가 lo보다 크면서, lo의 요소가 pivot보다 큰 원소를
       * 찾을 때 까지 lo를 증가시킨다.
       */
      while(a[lo] <= pivot && lo < hi) {
        lo++;
      }

      // 교환 될 두 요소를 찾았으면 두 요소를 바꾼다.
      swap(a, lo, hi);
    }



    /*
     *  마지막으로 맨 처음 pivot으로 설정했던 위치(a[left])의 원소와
     *  lo가 가리키는 원소를 바꾼다.
     */
    swap(a, left, lo);
```

```
    // 두 요소가 교환되었다면 피벗이었던 요소는 lo에 위치하므로 lo를 반환한다.
    return lo;
  }

  private static void swap(int[] a, int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
  }
}
```

오른쪽피벗 선택 방식 구현

```
public class QuickSort {

  public static void sort(int[] a) {
    r_pivot_sort(a, 0, a.length - 1);
  }

  /**
   *  오른쪽 피벗 선택 방식
   * @param a   정렬할 배열
   * @param lo  현재 부분배열의 왼쪽
   * @param hi  현재 부분배열의 오른쪽
   */
  private static void r_pivot_sort(int[] a, int lo, int hi) {

    /*
     *  lo가 hi보다 크거나 같다면 정렬 할 원소가
     *  1개 이하이므로 정렬하지 않고 return한다.
     */
    if(lo >= hi) {
      return;
    }

    /*
     * 피벗을 기준으로 요소들이 왼쪽과 오른쪽으로 약하게 정렬 된 상태로
     * 만들어 준 뒤, 최종적으로 pivot의 위치를 얻는다.
     *
     * 그리고나서 해당 피벗을 기준으로 왼쪽 부분리스트와 오른쪽 부분리스트로 나누어
     * 분할 정복을 해준다.
     *
     * [과정]
     *
     * Partitioning:
     *
     *         left part                right part       a[right]
     * +---------------------------------------------------------+
     * |    element < pivot    |    element >= pivot   |  pivot  |
     * +---------------------------------------------------------+
     *
```

```
     *
     *  result After Partitioning:
     *
     *         left part         a[hi]          right part
     * +-------------------------------------------------------+
     * |   element < pivot    |  pivot  |    element >= pivot    |
     * +-------------------------------------------------------+
     *
     *
     *  result : pivot = hi
     *
     *
     *  Recursion:
     *
     * r_pivot_sort(a, lo, pivot - 1)     r_pivot_sort(a, pivot + 1, hi)
     *
     *         left part                          right part
     * +-----------------------+            +-----------------------+
     * |   element <= pivot    |    pivot   |    element > pivot    |
     * +-----------------------+            +-----------------------+
     * lo                pivot - 1        pivot + 1                hi
     *
     */
    int pivot = partition(a, lo, hi);

    r_pivot_sort(a, lo, pivot - 1);
    r_pivot_sort(a, pivot + 1, hi);
  }



  /**
   * pivot을 기준으로 파티션을 나누기 위한 약한 정렬 메소드
   *
   * @param a   정렬 할 배열
   * @param left  현재 배열의 가장 왼쪽 부분
   * @param right 현재 배열의 가장 오른쪽 부분
   * @return    최종적으로 위치한 피벗의 위치(lo)를 반환
   */
  private static int partition(int[] a, int left, int right) {

    int lo = left;
    int hi = right;
    int pivot = a[right];   // 부분리스트의 오른쪽 요소를 피벗으로 설정

    // lo가 hi보다 작을 때 까지만 반복한다.
    while(lo < hi) {

      /*
       * hi가 lo보다 크면서, lo의 요소가 pivot보다 큰 원소를
       * 찾을 때 까지 lo를 증가시킨다.
       */
      while(a[lo] < pivot && lo < hi) {
        lo++;
      }

      /*
       * hi가 lo보다 크면서, hi의 요소가 pivot보다 작거나 같은 원소를
```

```
       * 찾을 때 까지 hi를 감소시킨다.
       */
      while(a[hi] >= pivot && lo < hi) {
        hi--;
      }


      // 교환 될 두 요소를 찾았으면 두 요소를 바꾼다.
      swap(a, lo, hi);
    }


    /*
     *  마지막으로 맨 처음 pivot으로 설정했던 위치(a[right])의 원소와
     *  hi가 가리키는 원소를 바꾼다.
     */
    swap(a, right, hi);

    // 두 요소가 교환되었다면 피벗이었던 요소는 hi에 위치하므로 hi를 반환한다.
    return hi;
  }

  private static void swap(int[] a, int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
  }
}
```

중간 피벗 선택방식 구현

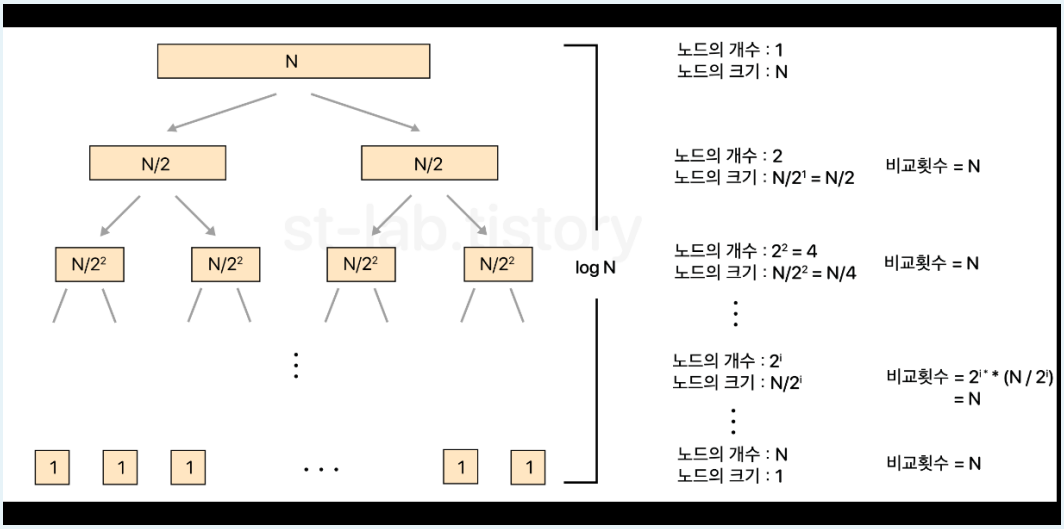
```
public class QuickSort {

  public static void sort(int[] a) {
    m_pivot_sort(a, 0, a.length - 1);
  }

  /**
   *  중간 피벗 선택 방식
   * @param a   정렬할 배열
   * @param lo  현재 부분배열의 왼쪽
   * @param hi  현재 부분배열의 오른쪽
   */
  private static void m_pivot_sort(int[] a, int lo, int hi) {

    /*
     *  lo가 hi보다 크거나 같다면 정렬 할 원소가
     *  1개 이하이므로 정렬하지 않고 return한다.
     */
    if(lo >= hi) {
      return;
    }
```

```
  /*
   * 피벗을 기준으로 요소들이 왼쪽과 오른쪽으로 약하게 정렬 된 상태로
   * 만들어 준 뒤, 최종적으로 pivot의 위치를 얻는다.
   *
   * 그리고나서 해당 피벗을 기준으로 왼쪽 부분리스트와 오른쪽 부분리스트로 나누어
   * 분할 정복을 해준다.
   *
   * [과정]
   *
   * Partitioning:
   *
   *      left part      a[(right + left)/2]      right part
   * +-------------------------------------------------------+
   * |    element < pivot    |  pivot  |    element >= pivot   |
   * +-------------------------------------------------------+
   *
   *
   *  result After Partitioning:
   *
   *         left part         a[hi]          right part
   * +-------------------------------------------------------+
   * |   element < pivot    |  pivot  |    element >= pivot    |
   * +-------------------------------------------------------+
   *
   *
   *  result : pivot = hi
   *
   *
   *  Recursion:
   *
   * m_pivot_sort(a, lo, pivot)         m_pivot_sort(a, pivot + 1, hi)
   *
   *         left part                           right part
   * +----------------------+            +----------------------+
   * |   element <= pivot    |            |    element > pivot    |
   * +----------------------+            +----------------------+
   * lo                pivot          pivot + 1                   hi
   *
   */
  int pivot = partition(a, lo, hi);

  m_pivot_sort(a, lo, pivot);
  m_pivot_sort(a, pivot + 1, hi);
}



/**
 * pivot을 기준으로 파티션을 나누기 위한 약한 정렬 메소드
 *
 * @param a   정렬 할 배열
 * @param left  현재 배열의 가장 왼쪽 부분
 * @param right 현재 배열의 가장 오른쪽 부분
 * @return    최종적으로 위치한 피벗의 위치(hi)를 반환
 */
private static int partition(int[] a, int left, int right) {
```

```
    // lo와 hi는 각각 배열의 끝에서 1 벗어난 위치부터 시작한다.
    int lo = left - 1;
    int hi = right + 1;
    int pivot = a[(left + right) / 2];    // 부분리스트의 중간 요소를 피벗으로 설정


    while(true) {

      /*
       * 1 증가시키고 난 뒤의 lo 위치의 요소가 pivot보다 큰 요소를
       * 찾을 때 까지 반복한다.
       */
      do {
        lo++;
      } while(a[lo] < pivot);


      /*
       * 1 감소시키고 난 뒤의 hi 위치가 lo보다 크거나 같은 위치이면서
       * hi위치의 요소가 pivot보다 작은 요소를 찾을 때 까지 반복한다.
       */
      do {
        hi--;
      } while(a[hi] > pivot && lo <= hi);


      /*
       * 만약 hi가 lo보다 크지 않다면(엇갈린다면) swap하지 않고 hi를 리턴한다.
       */
      if(lo >= hi) {
        return hi;
      }


      // 교환 될 두 요소를 찾았으면 두 요소를 바꾼다.
      swap(a, lo, hi);
    }

  }



  private static void swap(int[] a, int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
  }

}
```

시간 복잡도

일반적인 시간 복잡도



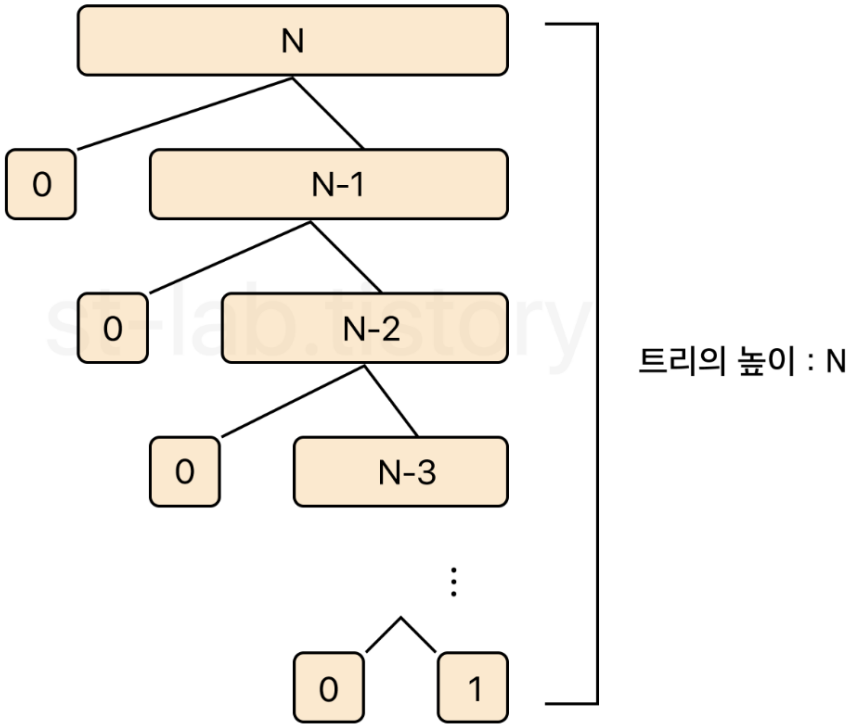
이상적으로 피벗이 중앙에 위치하게 되어 절반으로 나눈다고 할때, N개의 데이터가 있는 리스트를 1개까지 쪼개어 트리로 나타나면 이진트리형태로 나온다는 것은 우리가 확인할 수 있다.
N개 노드에 대한 이진트리의 높이는 logN이다.

- 비교 정렬 과정
피벗보다 큰, 작은 요소를 양 끝에서 시작하여 탐색하며 만족하지 못하는 경우 swap을 한다. 이는 현재 리스트의 요소들을 탐색하기 때문에 O(N)이다.

자세히 말하면 i번째 레벨에서 노드의 개수가 2^i개 이고, 노드의크기. 한 노드에 들어있는 원소의 개수는 N/2^i개이다.
그래서 2^i * N/2^i = O(N)이다.

그리고 트리의 높이인 logN번을 수행하므로
O(N) * O(logN)의 결과를 갖는다.

최악의 시간 복잡도



왼쪽을 기준으로 피벗을 잡을때를 예시로 들어보자.
N개가 있는 리스트에 대해 가장 왼쪽 요소를 pivot으로 잡고, pivot보다 작은 요소는 왼쪽, 큰 요소는 오른쪽으로 위치시킨다.
만약 pivot이 가장 작은 요소였다면 부분리스트는 위처럼 나눠진다.
왼쪽의 부분 리스트는 없고, N-1개의 요소를 갖는 오른쪽 부분 리스트만 생성될 것이다.
이를 반복적으로 재귀호출 하면 위처럼 생성된다.

마찬가지로 정렬할 리스트가 이미 내림차순으로 정렬되어 있다면 $O(N^2)$의 시간 복잡도를 갖게 된다.
어느 피벗을 선택하건 위와 같이 트리가 치중되며 공간 복잡도 또한 O(N)으로 되어버린다.

중간 피벗이 선호되는 이유가 바로 이러한 이유때문이다.
거의 정렬 된 배열이라도 거의 중간지점에 가까운 위치에서 왼쪽 리스트와 오른쪽 리스트가 균형에 가까운 트리를 얻어낼 수 있기 때문이다.

최선의 시나리오

💡 위와 같이 거의 정렬 된 상태에서 성능이 떨어진다는 점을 고려할 때, 임계값을 설정하여 일정 크기 이하로 떨어질 경우 정렬 된 배열에서 좋은 성능을 발휘하는 삽입정렬을 하도록 바꾸면 거의 정렬 된 배열에서도 어느 정도 성능 하락을 방지할 수 있다.

코드 구현 (퀵 + 삽입 정렬)

```
public class QuickSortWithInsertionSort {
    private static final int THRESHOLD = 10; // 임계치 설정

    public static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            if (high - low < THRESHOLD) {
                // 임계치 이하인 경우 삽입 정렬 호출
                insertionSort(arr, low, high);
            } else {
                int partitionIndex = partition(arr, low, high);
                quickSort(arr, low, partitionIndex - 1);
                quickSort(arr, partitionIndex + 1, high);
            }
        }
    }

    private static int partition(int[] arr, int low, int high) {
        int pivot = arr[high];
        int i = low - 1;

        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) {
                i++;
                swap(arr, i, j);
            }
        }

        swap(arr, i + 1, high);
        return i + 1;
    }

    private static void insertionSort(int[] arr, int low, int high) {
        for (int i = low + 1; i <= high; i++) {
            int key = arr[i];
            int j = i - 1;

            while (j >= low && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }

            arr[j + 1] = key;
        }
```

```
    }

    private static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    public static void main(String[] args) {
        int[] arr = {5, 2, 9, 1, 3, 7, 6, 4, 8};
        quickSort(arr, 0, arr.length - 1);
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```