재귀함수&정렬

목차

재귀함수

- 자기 자신을 다시 호출하여 문제를 해결해 나가는 함수

단점:재귀함수 종료 조건을 반드시 설정해야함
- 안 그러면 스택오버플로우가 발생할 수 있다.

재귀함수 대표 예제 팩토리얼 $n! = n \times (n-1) \times \cdots \times 1$

5!

5*4!

5*4*3!

5*4*3*2!

5*4*3*2*1

```java
public class Factorial {

    public static void main(String[] args) {
        int input = 5; // 5!
        System.out.println(fact(input));
    }

    public static int fact(int n) {
        if (n <= 1)
            return n;

        else
            return fact(n-1) * n;
    }

}
```

선택정렬

1. 주어진 리스트에서 최솟값을 찾는다.

2. 최솟값을 맨 앞 자리의 값과 교환한다.

3. 맨 앞 자리를 제외한 나머지 값들 중 최솟값을 찾아 위와 같은 방법으로 반복한다.

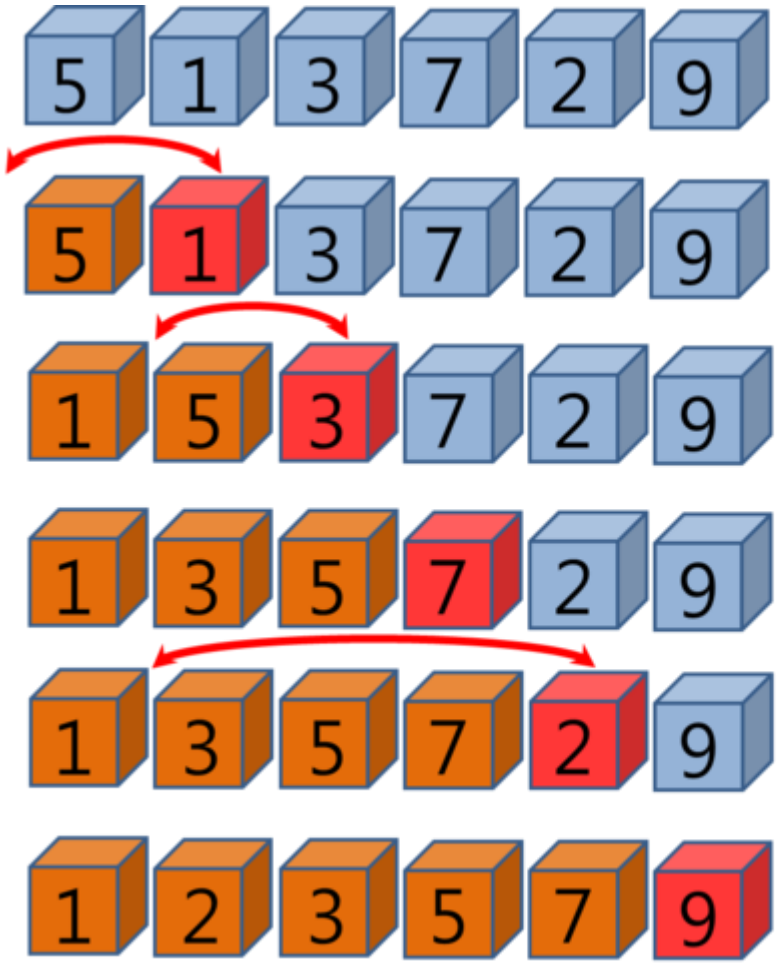
```c
int main(void){
    int i, j, min, index, temp;
    int array[10] = {1,10,5,8,7,6,4,3,2,9};
    for(i=0; i< 10; i++){
        min =9999;
        for(j=i; j<10; j++){ //for문 돌리면서 최소값 찾으면 min 값 바꾸기
            if(min>array[j]){
                min = array[j];
                index =j;
            }
        }
        //temp 이용해서 서로 교환
        temp = array[i];
        array[i] = array[index];
        array[index] = temp;
    }
}

```

단점

- 불안정 정렬이다
- **[B1, B2, C, A] (A < B < C)**
- B1이 B2보다 크거나 작은 것이 아니**다**

- 그럼 순서대로 순회하면서 교환한다면 이렇다.
- round 1 : [**A**, B2, C, **B1**]
- round 2 : [A, **B2**, C, B1]
- round 3 : [A, B2, **B1**, **C**]
-
- 이렇게 초기의 B1 B2의 순서가 뒤 바뀐 것을 볼 수 있다.

삽입정렬



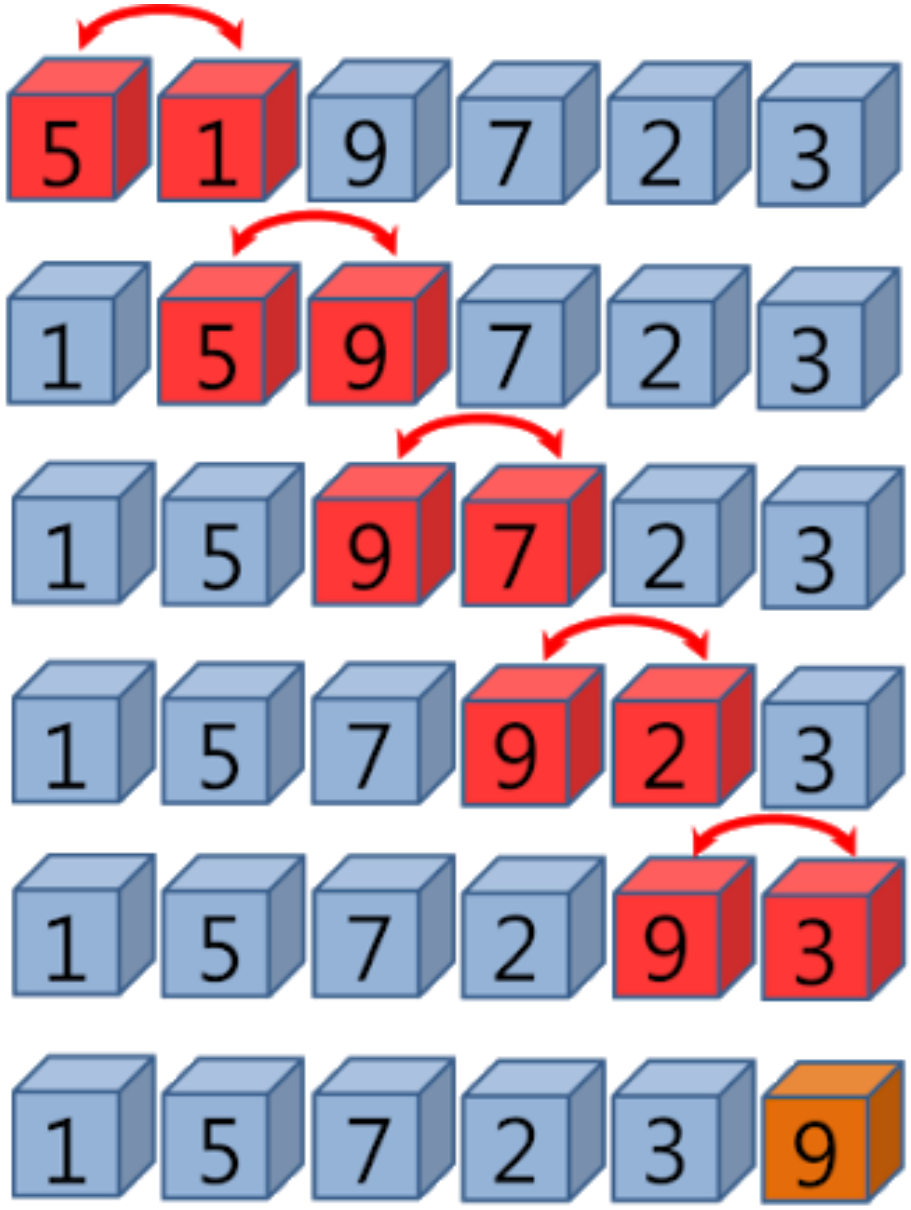
1. 현재 타겟이 되는 숫자와 이전 위치에 있는 원소들을 비교한다.
(첫 번째 타겟은 두 번째 원소부터 시작한다.)

2. 타겟이 되는 숫자가 이전 위치에 있던 원소보다 작다면 위치를 서로 교환한다.

3. 그 다음 타겟을 찾아 위와 같은 방법으로 반복한다.

```c
int main(void){
    int i, j, temp;
    int array[10] = {1,10,5,8,7,6,4,3,2,9};
    for(i=0; i< 9; i++){ //
        j=i;
        while(array[j]>array[j+1]){ //왼쪽에 있는 값이 오른쪽에 있는 값보다 크다면
            //위치 바꿈
            temp=array[j];
            array[j] = array[j+1];
            array[j+1]=temp;
            j--;
        }


}

```

버블 정렬

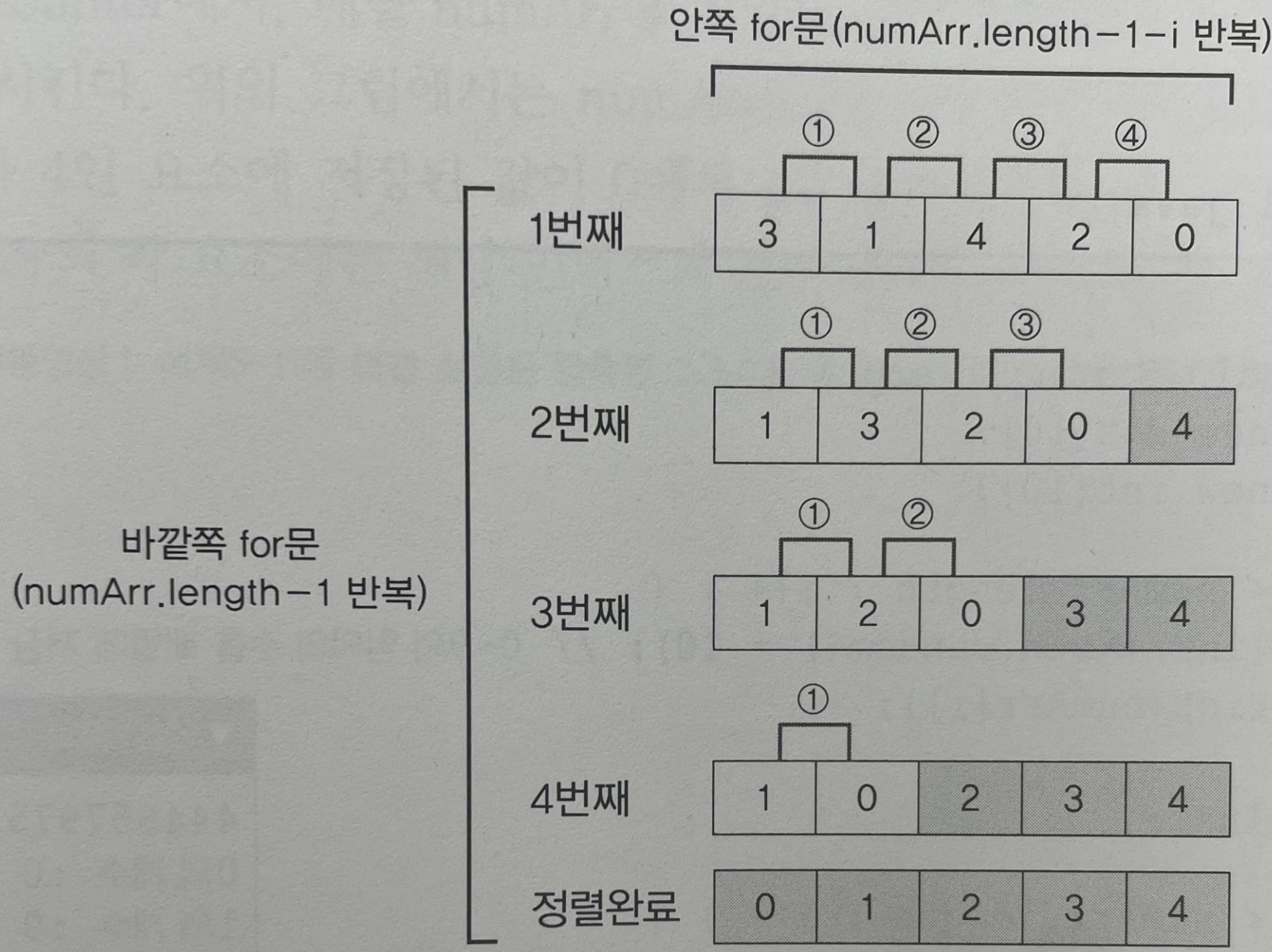


1. 앞에서부터 현재 원소와 바로 다음의 원소를 비교한다.

2. 현재 원소가 다음 원소보다 크면 원소를 교환한다.

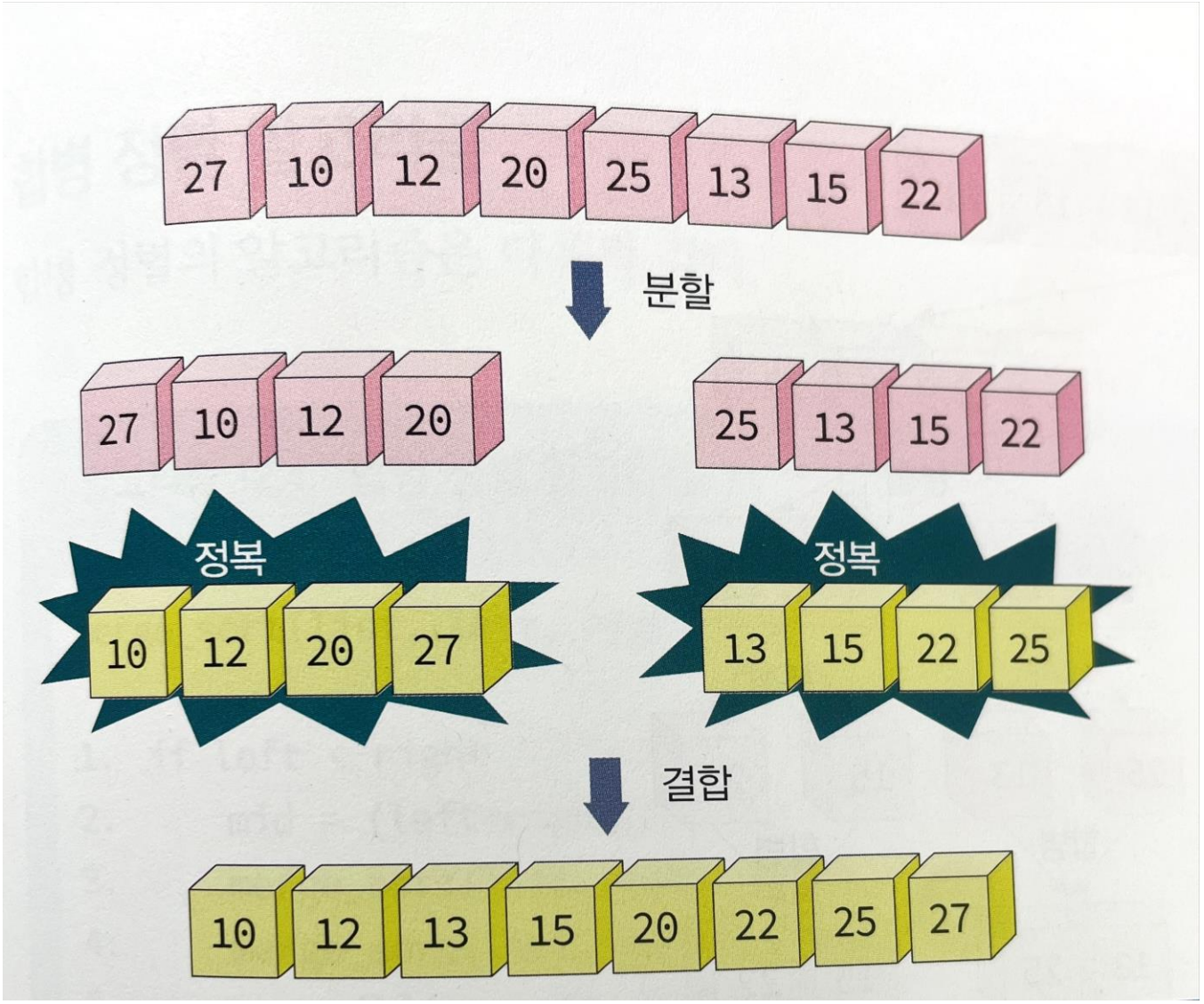
3. 다음 원소로 이동하여 해당 원소와 그 다음원소를 비교한다.

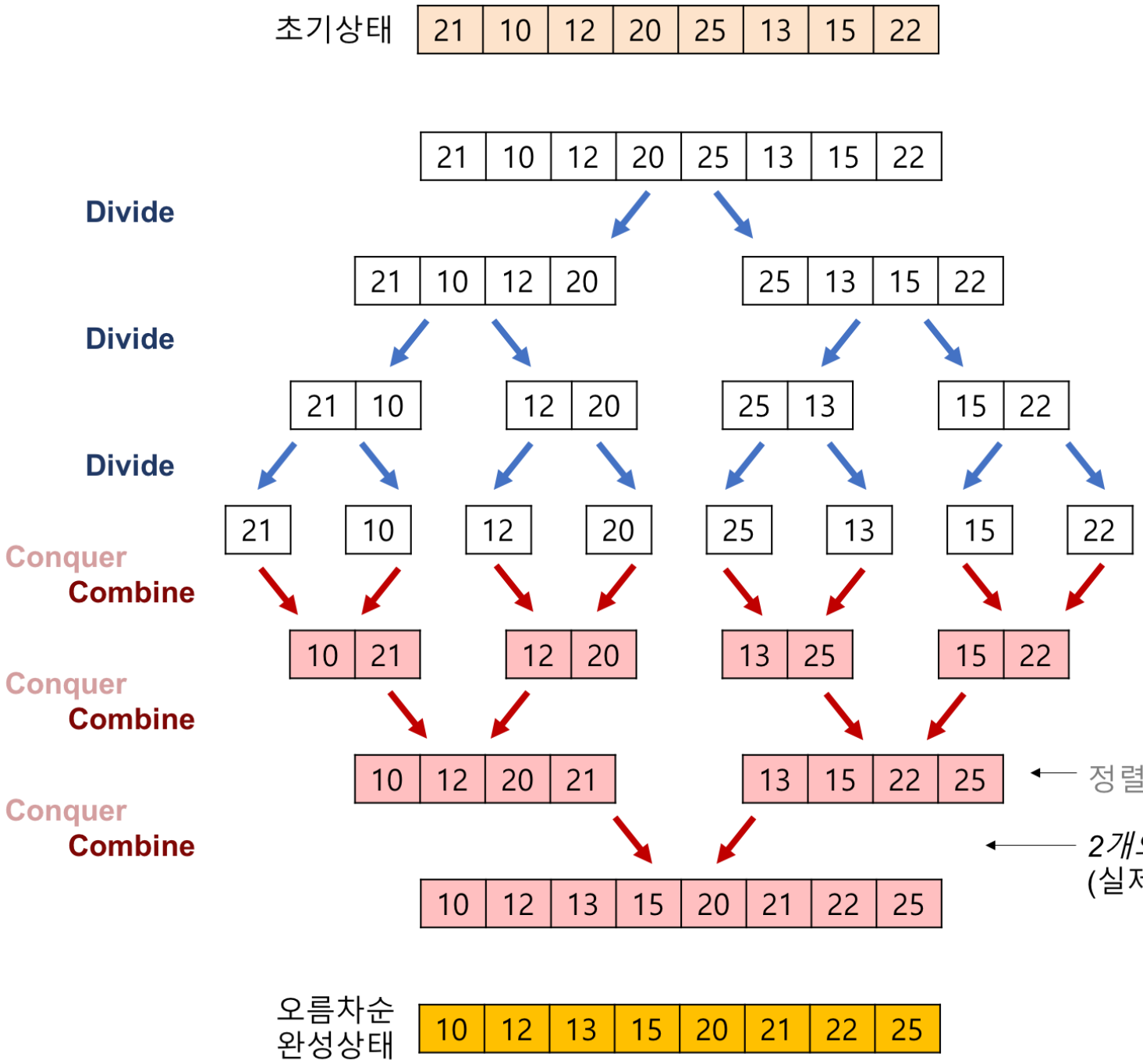
교해야 한다.


안쪽 for문(numArr.length−1−i 반복)
바깥쪽 for문
(numArr.length−1 반복)
1번째
2번째
3번째
4번째
정렬완료
① ② ③ ④
3 1 4 2 0
① ② ③
1 3 2 0 4
① ②
1 2 0 3 4
①
1 0 2 3 4
0 1 2 3 4

```
int main(void) {
    int i, j, temp;
    int array[10] = {1, 10, 5, 8, 7, 6, 4, 3, 2, 9};
    for(i = 0; i < array.length-1; i++) {
        for(j = 0; j < array.length -1 -i; j++) {
            if(array[j] > array[j + 1]) {
                temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
    return 0;
}
```

병합 정렬


27 10 12 20 25 13 15 22
분할
27 10 12 20
25 13 15 22
정복
10 12 20 27
정복
13 15 22 25
결합
10 12 13 15 20 22 25 27



1. **주어진 리스트를 절반으로 분할하여 부분리스트로 나눈다. (Divide : 분할)**

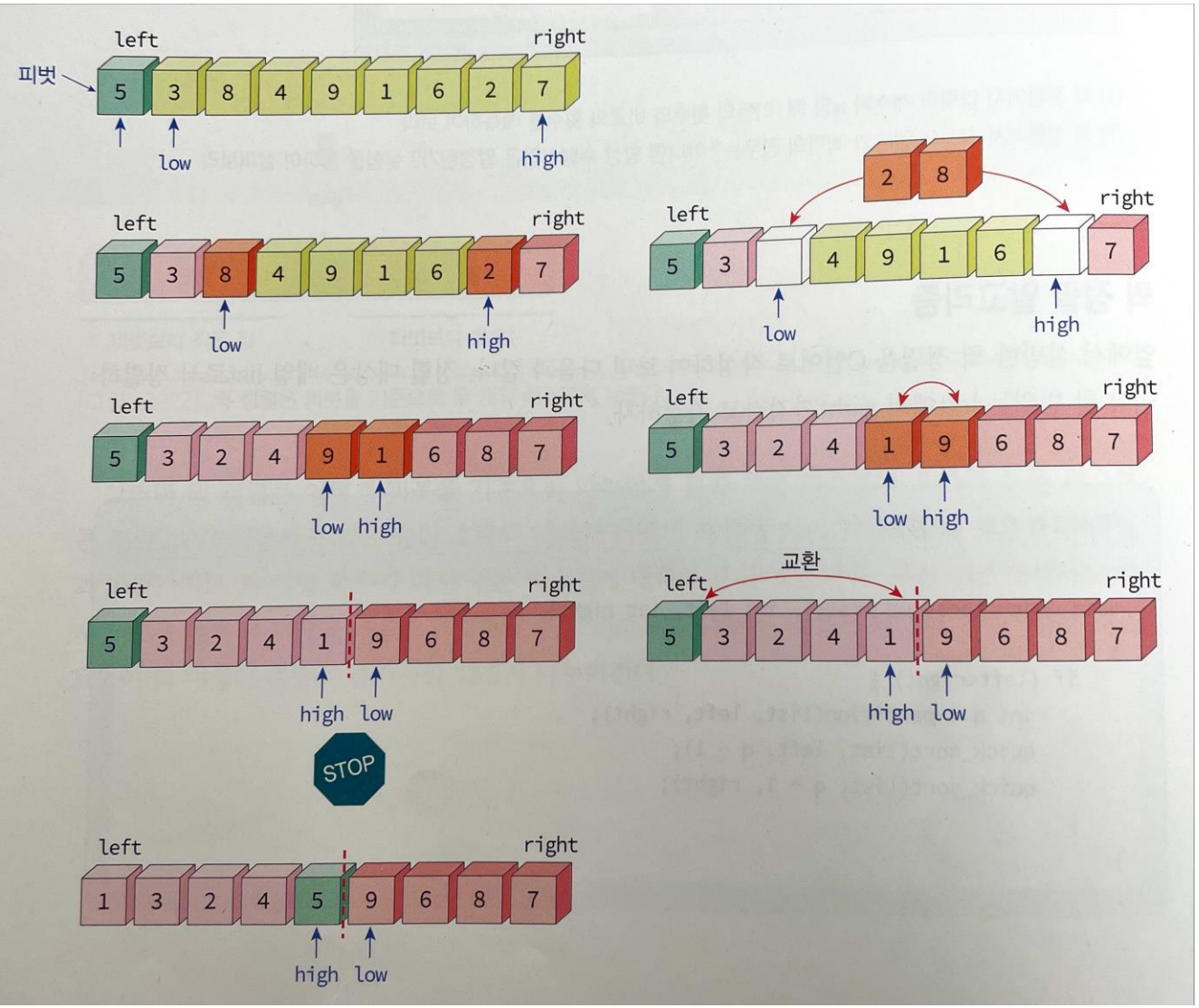
2. **해당 부분리스트의 길이가 1이 아니라면 1번 과정을 되풀이한다.**

3. **인접한 부분리스트끼리 정렬하여 합친다. (Conqure : 정복)**

```java
package Sort.Merge;

public class MergeBasic {
    public static void sort(int[] arr, int left, int right) {
        mergeSort(arr, left, right);
    }

    private static void mergeSort(int[] arr, int left, int right) {
        int mid = 0;
        if (left < right) {
            mid = (left + right) / 2; // 데이터 리스트의 중앙 인덱스를 구함
            mergeSort(arr, left, mid); // 중앙을 기준으로 왼쪽 데이터들을 분할한다.
            mergeSort(arr, mid + 1, right); // 중앙을 기준으로 오른쪽 데이터들을 분할한다.
            merge(arr, left, mid, right); // 정복 및 결합 과정을 진행한다.
        }
    }

    private static void merge(int[] arr, int left, int mid, int right) {
        // 분할된 왼쪽 리스트들의 시작점 변수
        int leftIndex = left;
        // 분할된 오른쪽 리스트들의 시작점 변수
        int rightIndex = mid + 1;
        // 정렬된 데이터가 저장될 인덱스
        int sortedIndex = left;
        // 정렬된 데이터를 임시로 저장할 곳
        int[] tmpSortedArray = new int[right + 1];

```

```
27
28          // 분할된 왼쪽 리스트의 인덱스가 mid까지 온 경우 왼쪽 정렬 완료
29          // 분할된 오른쪽 리스트의 인덱스가 right까지 온 경우 오른쪽 정렬 완료
30          // 즉, 왼쪽 또는 오른쪽 둘 중 하나라도 정렬이 완료된 경우 반복문을 빠져나감
31          while(leftIndex <= mid && rightIndex <= right) {
32              // 오름차순 조건문
33              if (arr[leftIndex] <= arr[rightIndex]) {
34                  tmpSortedArray[sortedIndex++] = arr[leftIndex++];
35              }
36              else {
37                  tmpSortedArray[sortedIndex++] = arr[rightIndex++];
38              }
39          }
40
41          // 왼쪽이 다 정렬된 경우 오른쪽 데이터들의 남은 부분들을 다 옮겨야 함
42          if (leftIndex > mid) {
43              for(int i=rightIndex; i<=right; i++) {
44                  tmpSortedArray[sortedIndex++] = arr[i];
45              }
46          }
47          else {
48              for(int i=leftIndex; i<=mid; i++) {
49                  tmpSortedArray[sortedIndex++] = arr[i];
50              }
51          }
52
53          // 원래 배열에 정렬된 데이터로 덮어씌움
54          for(int i=left; i<=right; i++) {
55              arr[i] = tmpSortedArray[i];
56          }
57      }
58  }
```

퀵 정렬



1. 피벗을 하나 선택한다.
2. 피벗을 기준으로 양쪽에서 피벗보다 큰 값, 혹은 작은 값을 찾는다. 왼쪽에서부터는 피벗보다 큰 값을 찾고, 오른쪽에서부터는 피벗보다 작은 값을 찾는다.

3. 양 방향에서 찾은 두 원소를 교환한다.

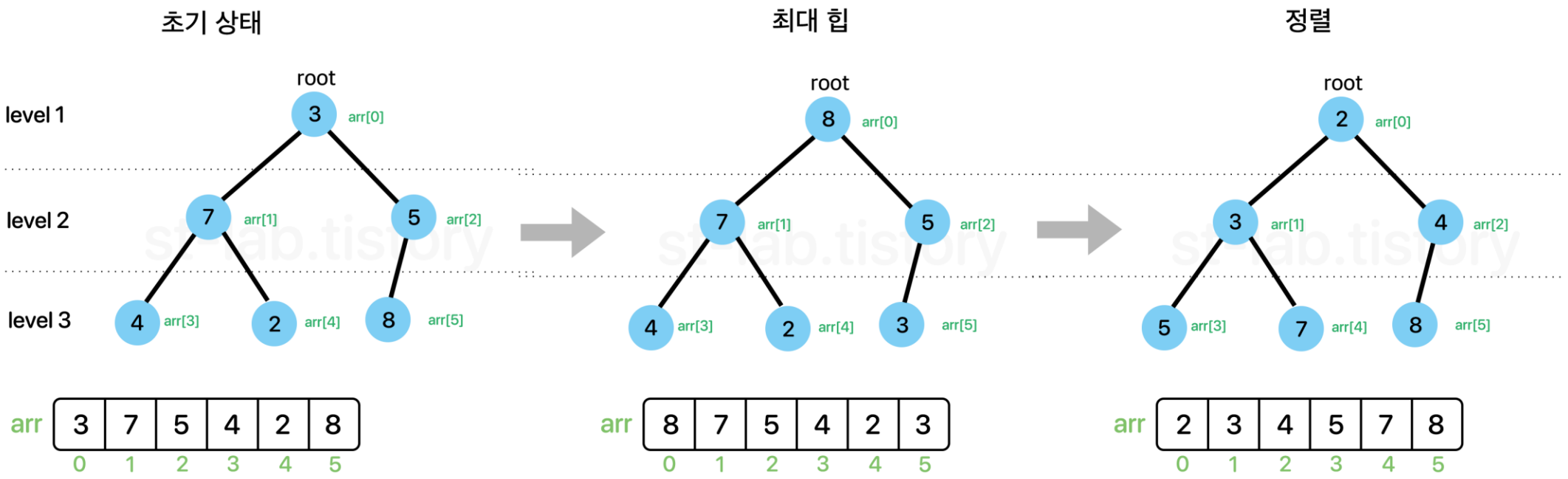
4. 왼쪽에서 탐색하는 위치와 오른쪽에서 탐색하는 위치가 엇갈리지 않을 때 까지 2번으로 돌아가 위 과정을 반복한다.

5. 엇갈린 기점을 기준으로 두 개의 부분리스트로 나누어 1번으로 돌아가 해당 부분리스트의 길이가 1이 아닐 때 까지 1번 과정을 반복한다. (Divide : 분할)

6. 인접한 부분리스트끼리 합친다. (Conqure : 정복)

```java
public class QuickSort {

  public static void main(String[] args) {
    int[] arr = { 3, 1, 5, 6, 20, 10, 7, 11, 15, 9 };
    quickSort(arr);
  }

  public static void quickSort(int[] arr) {
    quickSort(arr, 0, arr.length - 1);
  }

  private static void quickSort(int[] arr, int start, int end) {
    // start가 end보다 크거나 같다면 정렬할 원소가 1개 이하이므로 정렬하지 않고
    if (start >= end)
      return;

    // 가장 왼쪽의 값을 pivot으로 지정, 실제 비교 검사는 start+1 부터 시작
    int pivot = start;
    int lo = start + 1;
    int hi = end;

    // lo는 현재 부분배열의 왼쪽, hi는 오른쪽을 의미
    // 서로 엇갈리게 될 경우 while문 종료
    while (lo <= hi) {
      while (lo <= end && arr[lo] <= arr[pivot]) // 피벗보다 큰 값을 만날
        lo++;
      while (hi > start && arr[hi] >= arr[pivot]) // 피벗보다 작은 값을 만
        hi--;
      if (lo > hi)                // 엇갈리면 피벗과 교체
        swap(arr, hi, pivot);
      else
        swap(arr, lo, hi);            // 엇갈리지 않으면 lo, hi 값 교체
      }

    // 엇갈렸을 경우,
    // 피벗값과 hi값을 교체한 후 해당 피벗을 기준으로 앞 뒤로 배열을 분할하여 정
    quickSort(arr, start, hi - 1);
    quickSort(arr, hi + 1, end);

  }

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

힙 정렬



정렬해야 할 n개의 요소들로 최대 힙(완전 이진 트리 형태)을 만든다.

그 다음으로 한 번에 하나씩 요소를 힙에서 꺼내서 배열의 뒤부터 저장하면 된다.

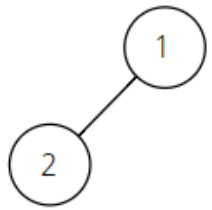
삭제되는 요소들(최댓값부터 삭제)은 값이 감소되는 순서로 정렬되게 된다.

완전 이진 트리

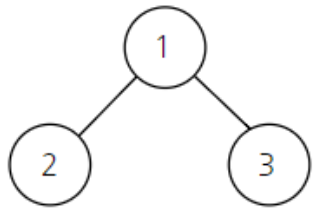
완전 이진 트리는 데이터가 루트(Root) 노드부터 시작해서 자식 노드가 왼쪽 자식 노드, 오른쪽 자식 노드로 차근차근 들어가는 구조의 이진 트리

1

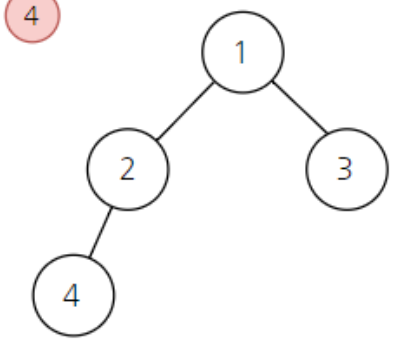
2



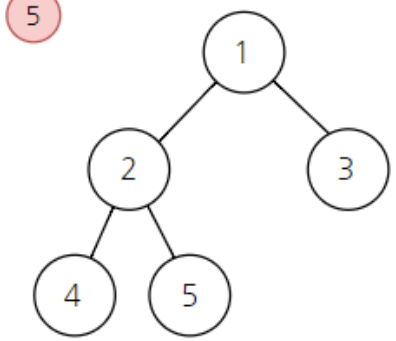
3



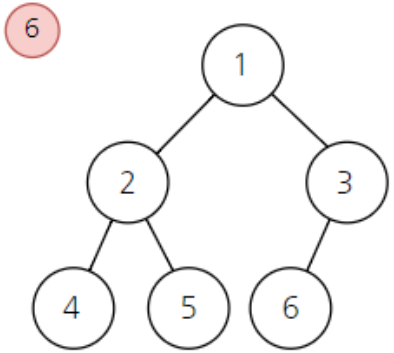
4



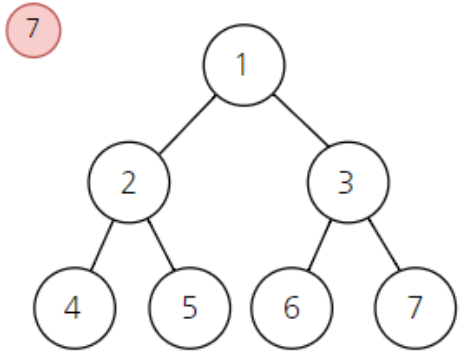
5



6



7

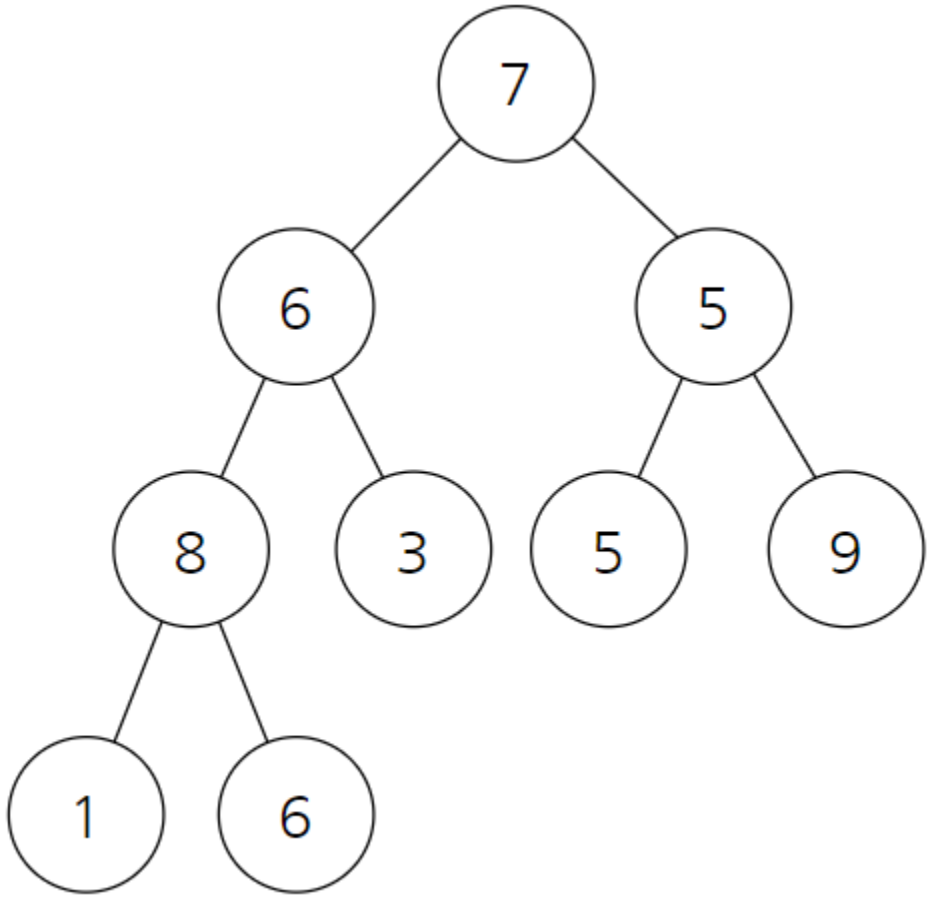


힙정렬하기

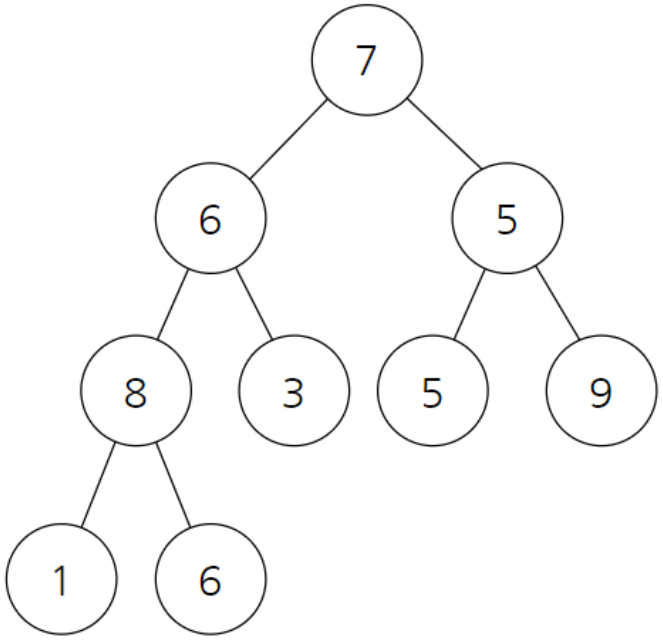
- ex) 7 6 5 8 3 5 9 1 6 를 오름차순으로 배열

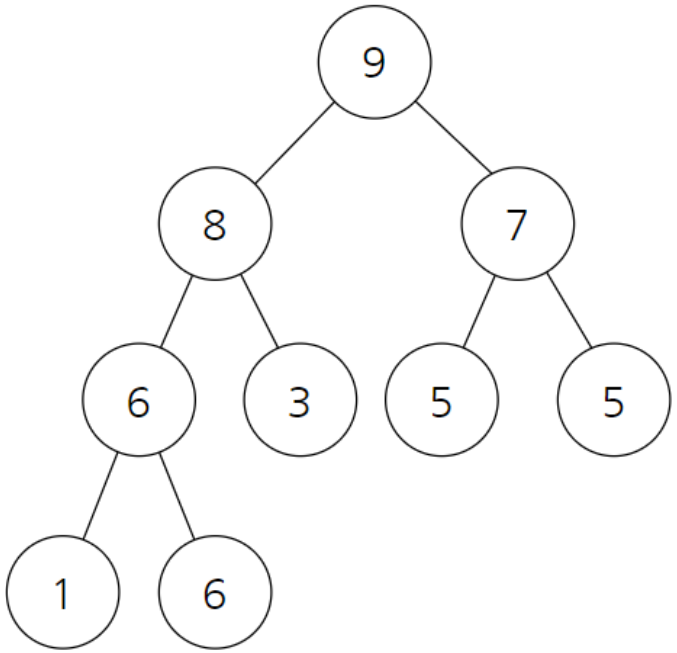
- 완전 이진 트리를 표현하는 가장 쉬운 방법은 배열에 그대로 삽입하는 겁니다.

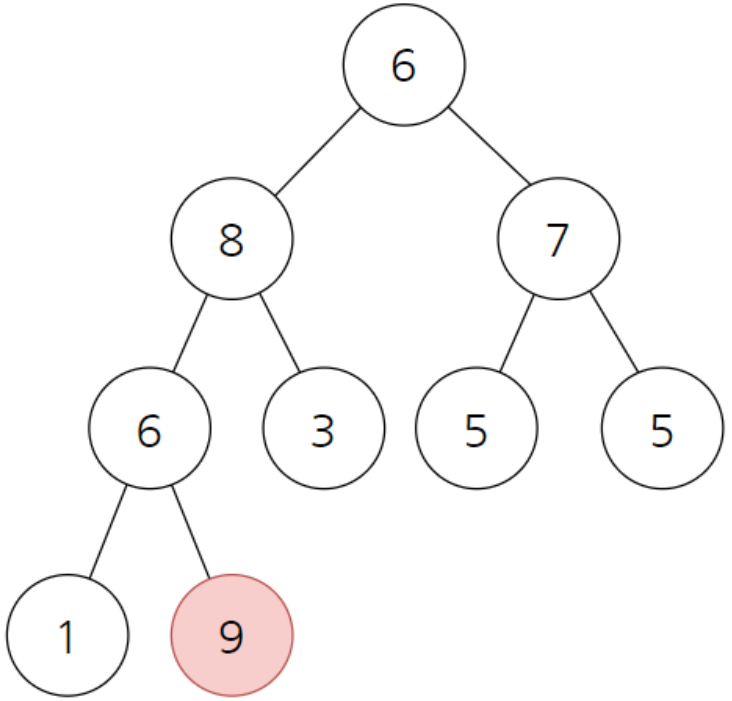
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
7	6	5	8	3	5	9	1	6

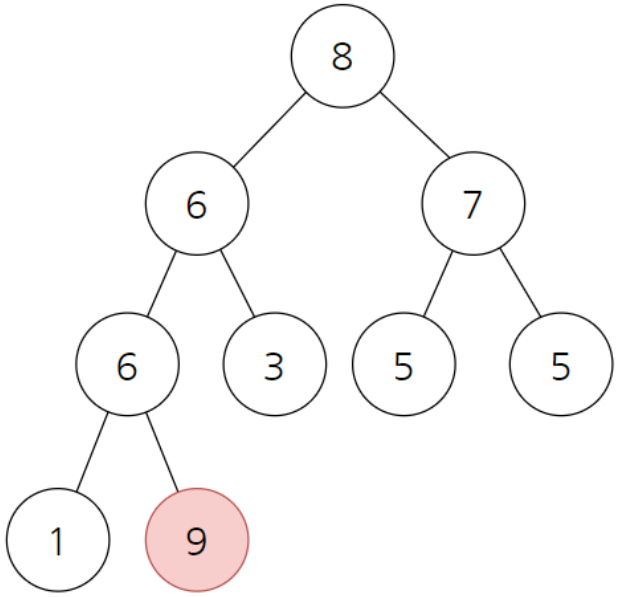


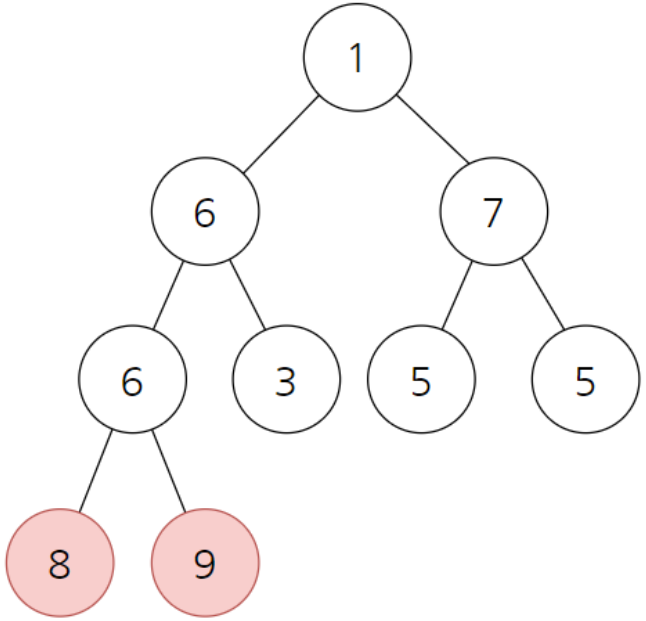
힙 생성 알고리즘을 적용해서 전체 트리를 힙 구조로 만들기











```java
import java.util.PriorityQueue;

public class test {
    public static void main(String[] args) {

        int[] arr = {3, 7, 5, 4, 2, 8};
        System.out.print(" 정렬 전 original 배열 : ");
        for(int val : arr) {
            System.out.print(val + " ");
        }

        PriorityQueue<Integer> heap = new PriorityQueue<Integer>();

        // 배열을 힙으로 만든다.
        for(int i = 0; i < arr.length; i++) {
            heap.add(arr[i]);
        }

        // 힙에서 우선순위가 가장 높은 원소(root노드)를 하나씩 뽑는다.
        for(int i = 0; i < arr.length; i++) {
            arr[i] = heap.poll();
        }


        System.out.print("\n 정렬 후 배열 : ");
        for(int val : arr) {
            System.out.print(val + " ");
        }

    }
}
```

시간복잡도 비교

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
셸 정렬	n	$n^{1.5}$	n^2	0.056
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

단순(구현 간단)하지만 비효율적인 방법: 삽입 정렬, 선택 정렬, 버블 정렬
복잡하지만 효율적인 방법: 퀵 정렬, 힙 정렬, 합병 정렬, 기수 정렬