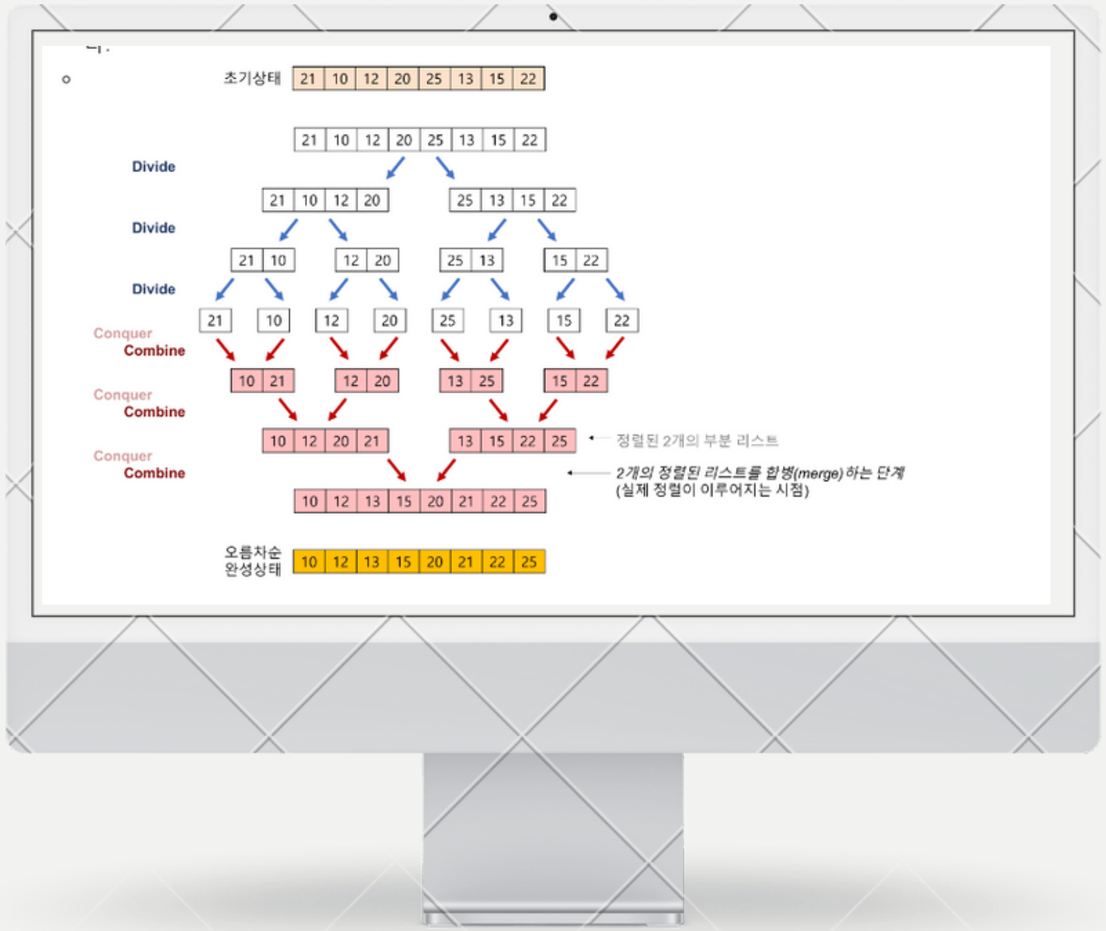
2023-11-01 김석희

정렬(2)



굳이 정렬을 배워야 하나?

그런데 알고리즘을 배우다 보면,
'정렬(Sorting)'을 아주 중요하게 다룬다.

교과서에서도 제일 먼저 나오는 게 정렬 알고리즘.
알고리즘 강의를 봐도 내용의 30-40% 정도가 정렬.

처음 알고리즘을 접했을 땐,
왜 이렇게 정렬을 중요하게 다루는지 의아했다.

> 흠, 정렬이 그렇게 중요한 건가...?

굳이 정렬을 배워야 하나?

어디에나 있는 정렬

우리가 쓰는 소프트웨어를 생각해보자. 곳곳에 정렬이 들어가지 않은 곳이 없다.

- 내 메일 보관함은 도착 시간 기준으로 정렬돼있다.
- 배달앱을 켜면 음식점이 인기순, 판매량순, 배달빠른 순으로 정렬된다.
- 구글에 검색어를 입력하면 가장 관련있는 페이지 순으로 정렬된다.
- 지도 앱에서 음식점을 검색하면 가까운 거리 순으로 정렬된 결과를 보여준다.
- 페이스북은 내가 좋아할만한 게시물 순으로 뉴스피드를 정렬한다.
- 트위터 피드. 인스타 스토리... 손 아프니까 이하 생략.

굳이 정렬을 배워야 하나?

그 외에 우리 눈에 보이지 않는 수많은 정렬까지.

정렬은 어디에나 있다.
스크롤 한번, 클릭 한번이 다 정렬이다.

다만 사용할 땐 우리가 의식하지 못할 뿐이다.
물이 뭐냐고 묻는 물고기처럼.

굳이 정렬을 배워야 하나?

사실 소프트웨어의 일을 아주 단순하게 표현하면,
'데이터를 저장하고 가공해서 사용자에게 보여주는 것'이다.

사람이 받아들일 수 있는 데이터의 양은 정해져있다. 저장된 수많은 데이터 중에서, **가장 중요하고 유용한 것 몇 개를 골라내야 한다. 골라내려면 당연히 줄을 세워야 한다.**

검색도 정렬과 관련이 있다. 수많은 데이터 중에서 사용자가 원하는 무언가를 효율적으로 찾아낼 때도 정렬이 활용된다.

정보를 다루는 소프트웨어에서 정렬은 거의 숨쉬기다.
자주 실행되는 가장 기본적인 작업이다.

그러다보니 정렬 알고리즘을 잘 만드는 건,
컴퓨터 공학에서 아주아주 중요한 주제다.

굳이 정렬을 배워야 하나?

사실 소프트웨어의 일을 아주 단순하게 표현하면,
'데이터를 저장하고 가공해서 사용자에게 보여주는 것'이다.

사람이 받아들일 수 있는 데이터의 양은 정해져있다. 저장된 수많은 데이터 중에서, **가장 중요하고 유용한 것 몇 개를 골라내야 한다. 골라내려면 당연히 줄을 세워야 한다.**

검색도 정렬과 관련이 있다. 수많은 데이터 중에서 사용자가 원하는 무언가를 효율적으로 찾아낼 때도 정렬이 활용된다.

정보를 다루는 소프트웨어에서 정렬은 거의 숨쉬기다.
자주 실행되는 가장 기본적인 작업이다.

그러다보니 정렬 알고리즘을 잘 만드는 건,
컴퓨터 공학에서 아주아주 중요한 주제다.

굳이 정렬을 배워야 하나?

잘 만든다는 건?

굳이 정렬을 배워야 하나?

더 빠르게 할 수 없을까?

굳이 정렬을 배워야 하나?

자, 여러분이 이 책을 제목순으로 정렬해야 한다면 어떻게 할까?

일단 눈에 띄는 걸 아무거나 하나 집는다.

앞쪽부터 다른 책들을 하나씩 훑으면서
이 책보다 뒤에 와야 하는 책이 있으면 그 앞에 꽂는다.

다시 정렬되지 않은 책을 뽑아서,
같은 방법으로 다른 책들을 훑으면서 적절한 자리에 꽂는다.

굳이 정렬을 배워야 하나?

문제는 효율성이다.

이렇게 하면 총 몇 번을 해야 정렬될까?

n개의 책에 대해서 최대 n-1 번의 비교를 해야 하니까,
이걸 Big O로 표기하면 O(n^2)이다.

책의 수가 늘어날 수록,

해야하는 작업의 수가 매우 빠르게 증가한다.

굳이 정렬을 배워야 하나?

하지만 바쁘다 바빠 현대사회인이 여기서 만족할리 없다.

> '이거보다 더 빠르게 할 수 없을까?'

굳이 정렬을 배워야 하나?

우리보다 먼저 살았던 컴퓨터 공학자, 소프트웨어 엔지니어들이 '이거보다 더 빠르게 할 수 없을까?'라며 온갖 잔머리와 테크닉을 짜냈다.

계속 새로운 정렬 방법을 시도했고, 결국 더 나은 방법을 발견해냈다.

굳이 정렬을 배워야 하나?

그 결과, 오늘날 쓰이는 알고리즘 대부분은 O(n log n)!

선형로그 시간 복잡도를 가진다.

O(n^2)보다 훨씬 더 빨라진 알고리즘이다.

굳이 정렬을 배워야 하나?

이것보다 더 빠른 건 없냐고?

아쉽게도 일대일 비교를 통한 정렬에서 시간 복잡도는 O(n log n)이 한계다.

수학적으로 확실히 증명된 사실.

굳이 정렬을 배워야 하나?

그래도 왜?

굳이 정렬을 배워야 하나?

이미 누군가 다 만들어놓았고,
sort함수가 이미 내장되어 있고,

굳이 정렬을 배워야 하나?

그냥 #include <algorithm>해서 sort만 가져오면 되지 왜 배우나?

굳이 정렬을 배워야 하나?

솔직히 말해서 정렬을 직접 구현해서 쓸 일은 거의 없을 겁니다.

그럼에도 우리가 자료구조와 알고리즘을 배우는 이유는.

굳이 정렬을 배워야 하나?

정렬은 알고리즘을 배우는 가장 좋은 교재다

c++ python c java 다 떠나서
근본적으로 코딩을 잘해지기 위함

괴물같은 선배들이 컴퓨터처럼 사고한 매뉴얼

굳이 정렬을 배워야 하나?

컴퓨터 공학에서 정렬을 잘 하는 건 너무 중요한 일이었다.

이걸 사용하면 더 빨라지지 않을까?
이 아이디어를 적용하면 더 안정적이지 않을까?

계속 이런 고민을 하다보니 정렬 알고리즘은
수많은 알고리즘 아이디어(잔머리)의 자연 집합소였다는 것이다.

굳이 정렬을 배워야 하나?

- 특수한 자료구조를 쓰면 알고리즘을 더 빠르게 만들 수 있다든지,
- 무작위 난수를 사용해서 알고리즘을 좋게 만드는 방법이라든지,
- 문제를 바로 풀 수 있을 정도로 작게 쪼개는 방법이라든지,
- 인풋에 대한 가정이 있을 때 그걸 활용하는 방법이라든지.

굳이 정렬을 배워야 하나?

그리고 이 알고리즘 중 무엇이 더 좋은가? 를 고민하다보면
각각의 알고리즘들을 평가하는 도구들도 자연스럽게 알게 된다.

- Big O 표기법은 물론이고,
- 평균적인 경우와 최악의 경우에 어떻게 성능이 달라지는지,
- 재귀 호출이 있을 때 복잡도는 어떻게 계산하는지,
- 왜 상황에 따라서 적절한 트레이드 오프가 필요한지

굳이 정렬을 배워야 하나?

이런 아이디어들과 평가법은,
우리가 정렬 알고리즘을 직접 만들지 않아도,
개발을 하면서 맞닥뜨릴 **수많은 문제들에서 유용하게 쓰일 도구들**이다.

굳이 정렬을 배워야 하나?

다시 말해서,

정렬 알고리즘은 알고리즘을 배우는 가장 좋은 교재다.

굳이 정렬을 배워야 하나?

eddy_song님의 “정렬 알고리즘은 왜 배워야 할까?” 글 참조

자, 이제 배워봅시다.

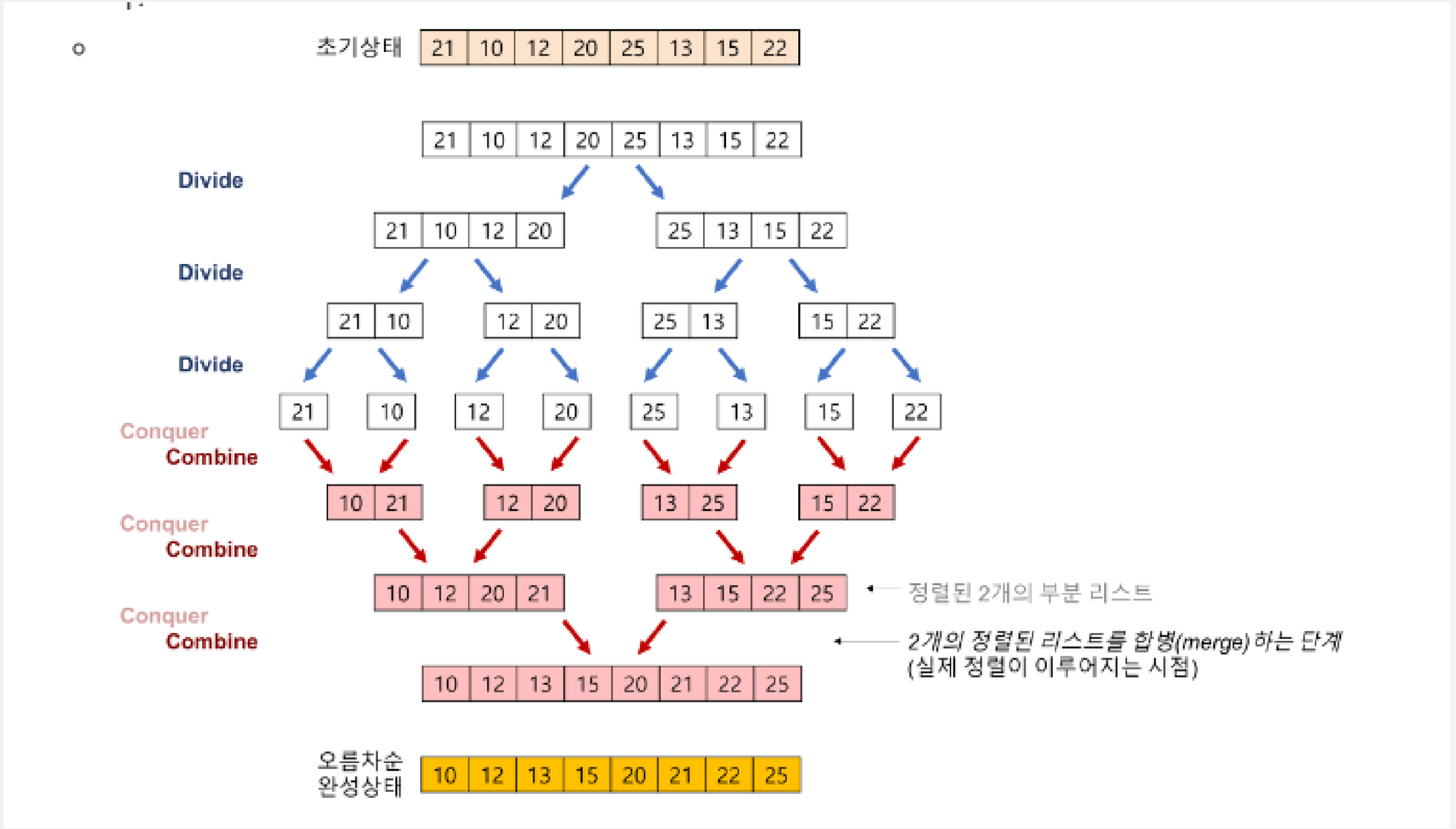
오늘 배울 정렬들

병합 정렬

퀵 정렬

힙 정렬

병합 정렬(merge sort)



병합 정렬

카이사르, 나폴레옹, 손자, 제갈량, 김석희....

병합 정렬

카이사르, 나폴레옹, 손자, 제갈량, 김석희....

전쟁의 고수들은 모두 **이 전략**을 사용해서 엄청난 전과를 거뒀다.

누구나 한번쯤 들어봤을 그 전략은 바로 '각개격파'다.

적을 여러 개로 각각 나누어서 분산시킨 다음, (각개各個)
하나씩 하나씩 때려눕힌다는 뜻이다. (격파擊破)

영어로는 'Divide and Conquer' 라고 한다.

'분할 정복'

병합 정렬

- 전체 문제를 더 작은 문제로 반복해서 쪼갠다. **(분할)**
- 충분히 문제가 작아지면 문제를 푼다. **(정복)**
- 작은 문제의 답을 합쳐서 전체 문제의 답을 도출한다. **(조합)**

병합 정렬

출처: Khan Academy

병합 정렬

분할

1. 주어진 배열의 중간 인덱스를 구한다.
2. 중간 인덱스를 기준으로 배열을 반토막 낸다.

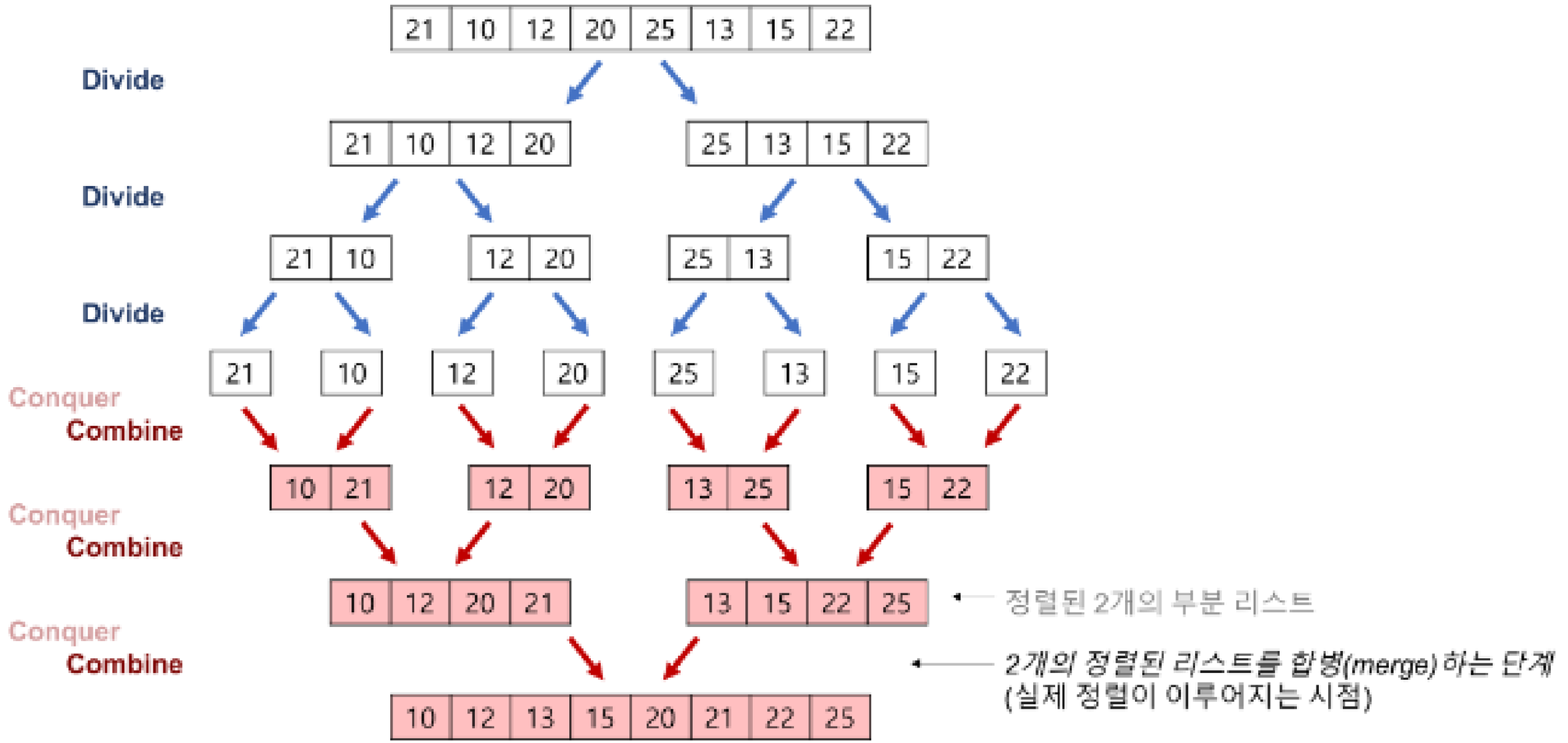
정복

3. 반복해서 반토막을 내다가 마침내 배열의 크기가 0이나 1이 된다.
4. 그러면 주어진 배열을 바로 '답'으로 반환한다.

조합

5. 왼쪽 배열, 오른쪽 배열을 인자로 넣어서 각각 정렬된 결과를 얻는다.
6. 양쪽 배열의 맨 앞을 비교해 더 작은 수를 찾는다.
7. 더 작은 수를 꺼내서 새로운 배열에 집어넣는다.
8. 새로운 배열에 모든 숫자가 들어갈 때까지 반복한다.

병합 정렬


21 10 12 20 25 13 15 22
Divide
21 10 12 20
25 13 15 22
Divide
21 10
12 20
25 13
15 22
Divide
21
10
12
20
25
13
15
22
Conquer
Combine
10 21
12 20
13 25
15 22
Conquer
Combine
10 12 20 21
13 15 22 25
Conquer
Combine
10 12 13 15 20 21 22 25
정렬된 2개의 부분 리스트
2개의 정렬된 리스트를 합병(merge)하는 단계
(실제 정렬이 이루어지는 시점)

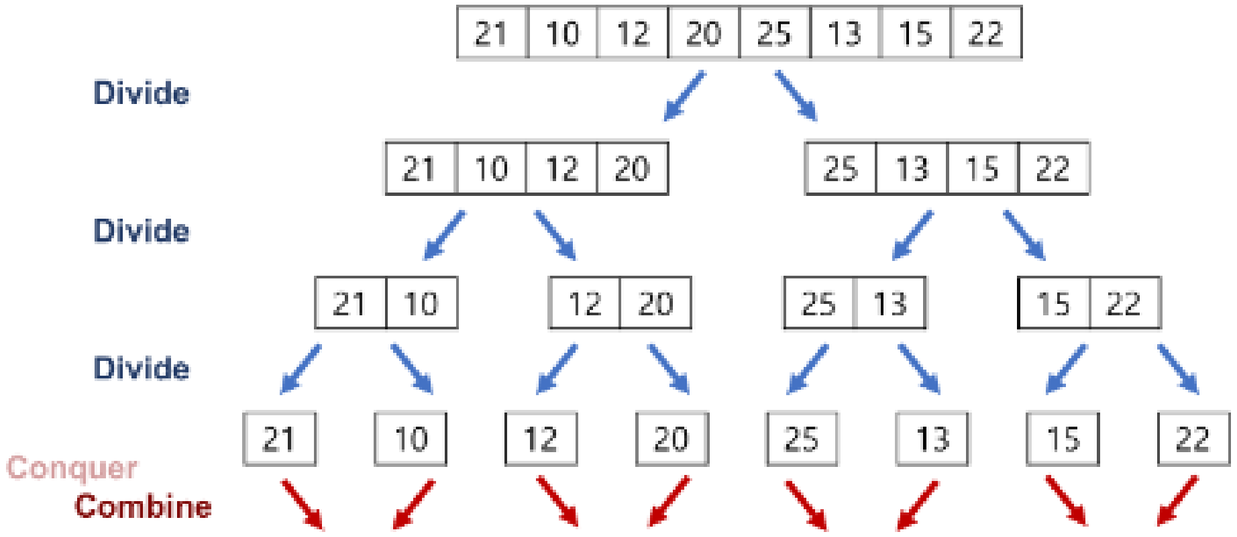
병합 정렬

분할

1. 주어진 배열의 중간 인덱스를 구한다.
2. 중간 인덱스를 기준으로 배열을 반토막 낸다.

정복

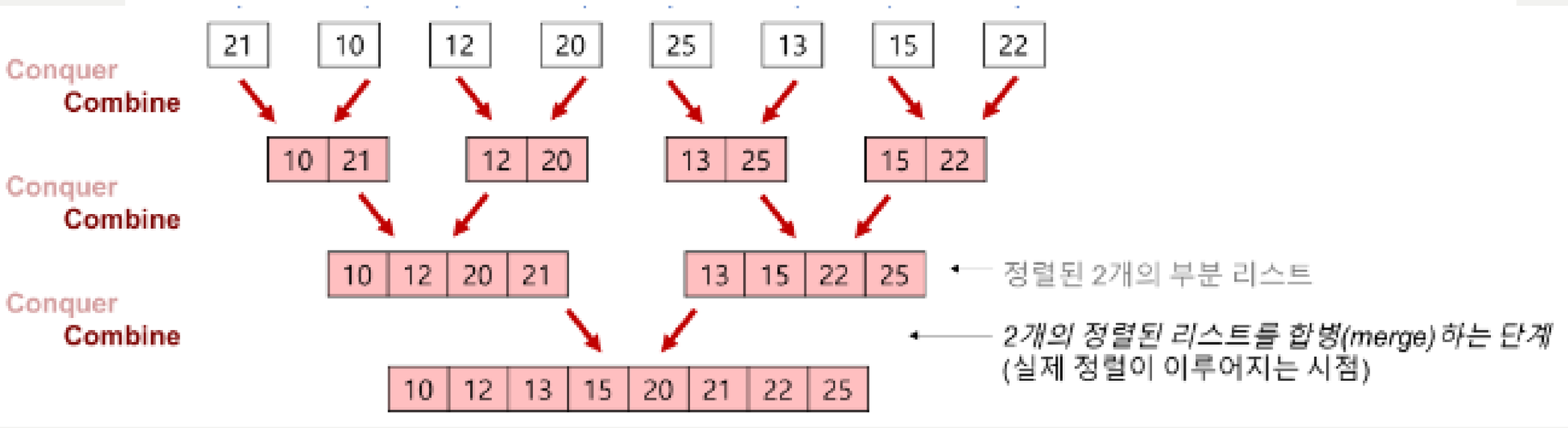
3. 반복해서 반토막을 내다가 마침내 배열의 크기가 0이나 1이 된다.
4. 그러면 주어진 배열을 바로 '답'으로 반환한다.



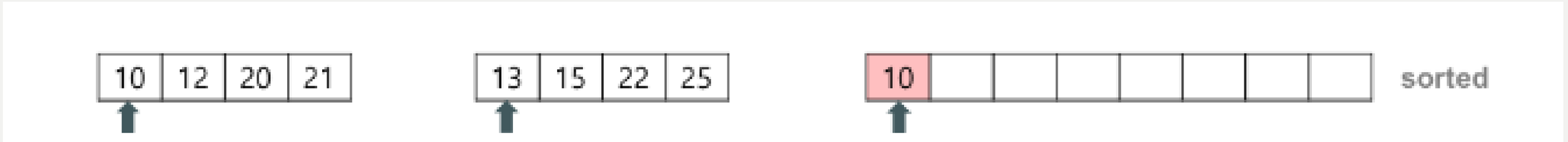
병합 정렬

조합

5. 왼쪽 배열, 오른쪽 배열을 인자로 넣어서 각각 정렬된 결과를 얻는다.
6. 양쪽 배열의 맨 앞을 비교해 더 작은 수를 찾는다.
7. 더 작은 수를 꺼내서 새로운 배열에 집어넣는다.
8. 새로운 배열에 모든 숫자가 들어갈 때까지 반복한다.

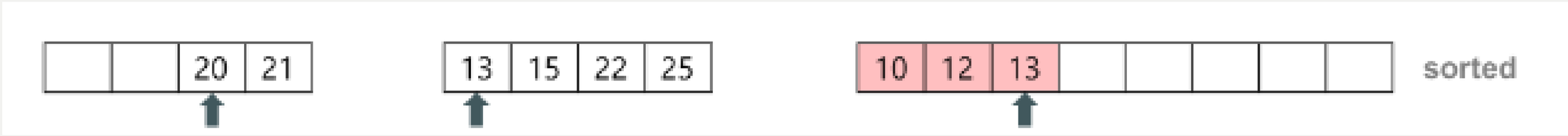


병합 정렬

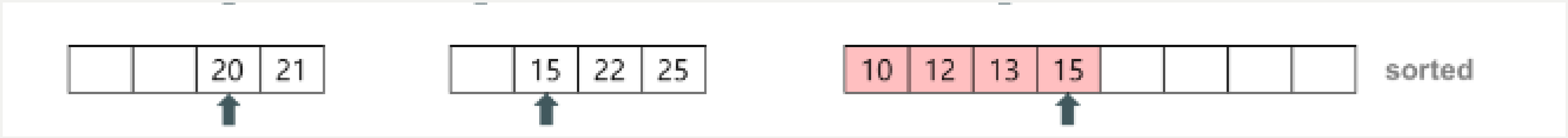


병합 정렬

병합 정렬

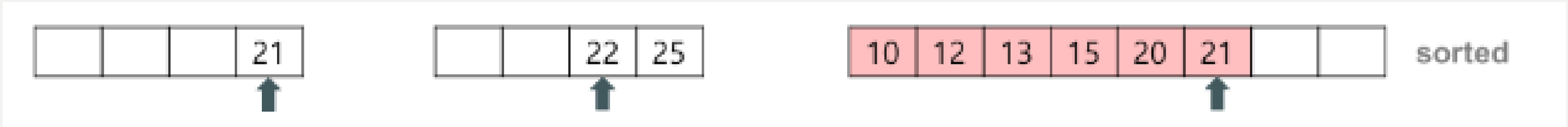


병합 정렬

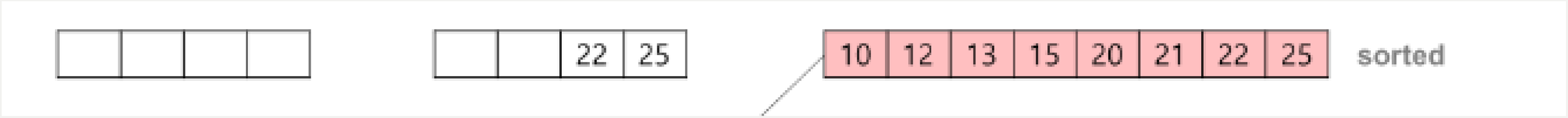


병합 정렬

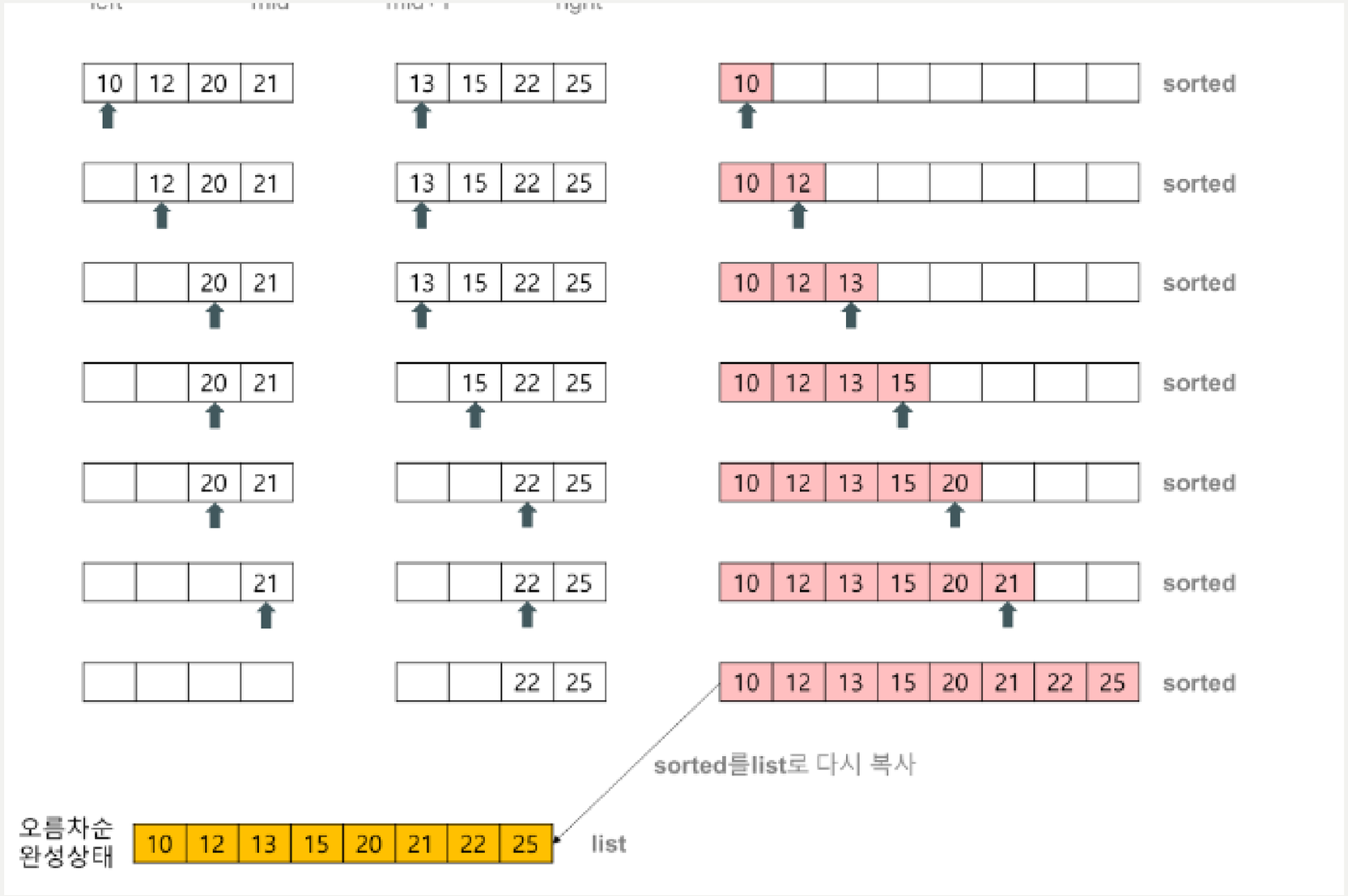
병합 정렬



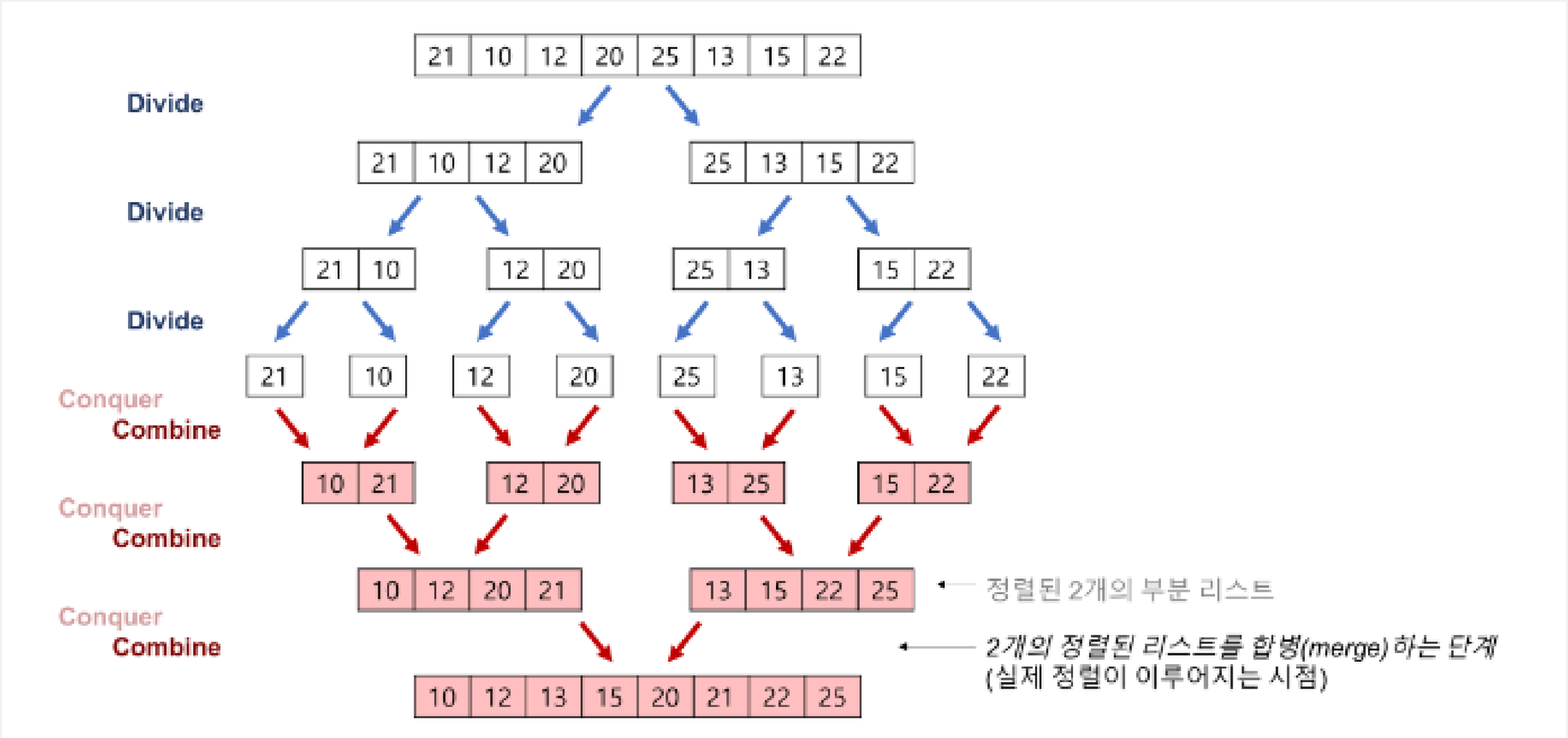
병합 정렬



병합 정렬


10 12 20 21
13 15 22 25
10
sorted
12 20 21
13 15 22 25
10 12
sorted
20 21
13 15 22 25
10 12 13
sorted
20 21
15 22 25
10 12 13 15
sorted
20 21
22 25
10 12 13 15 20
sorted
21
22 25
10 12 13 15 20 21
sorted
22 25
10 12 13 15 20 21 22 25
sorted
sorted를list로 다시 복사
오름차순
완성상태
10 12 13 15 20 21 22 25
list

병합 정렬 최종


21 10 12 20 25 13 15 22
Divide
21 10 12 20
25 13 15 22
Divide
21 10
12 20
25 13
15 22
Divide
21
10
12
20
25
13
15
22
Conquer
Combine
10 21
12 20
13 25
15 22
Conquer
Combine
10 12 20 21
13 15 22 25
Conquer
Combine
10 12 13 15 20 21 22 25
정렬된 2개의 부분 리스트
2개의 정렬된 리스트를 합병(merge)하는 단계
(실제 정렬이 이루어지는 시점)

병합 정렬 코드

```
    int n = MAX_SIZE;
    int list[n] = {21, 10, 12, 20, 25, 13, 15, 22};

    // 합병 정렬 수행(start: 배열의 시작 = 0, end: 배열의 끝 = 7)
    merge_sort(list, 0, n - 1);
```

병합 정렬 코드 (main)

```cpp
int main() {
    int i;
    int n = MAX_SIZE;
    int list[n] = {21, 10, 12, 20, 25, 13, 15, 22};

    // 합병 정렬 수행(start: 배열의 시작 = 0, end: 배열의 끝 = 7)
    merge_sort(list, 0, n - 1);

    for (auto a : list) {
        cout << a << " ";
    }
    return 0;
}
```

병합 정렬 코드 도입부

```c
void merge_sort(int *list, int start, int end) {
    if (start < end) { // start==end라는 뜻 즉 원소가 한개
        // 중간 위치를 계산하여 리스트를 균등 분할 -분할(Divide)
        int middle = (start + end) / 2;
        // 앞쪽 부분 리스트 정렬 -정복(Conquer)
        merge_sort(list, start, middle);
        // 뒤쪽 부분 리스트 정렬 -정복(Conquer)
        merge_sort(list, middle + 1, end);
        // 정렬된 2개의 부분 배열을 합병하는 과정 -결합(Combine)
        merge(list, start, middle, end);
    }
}
```

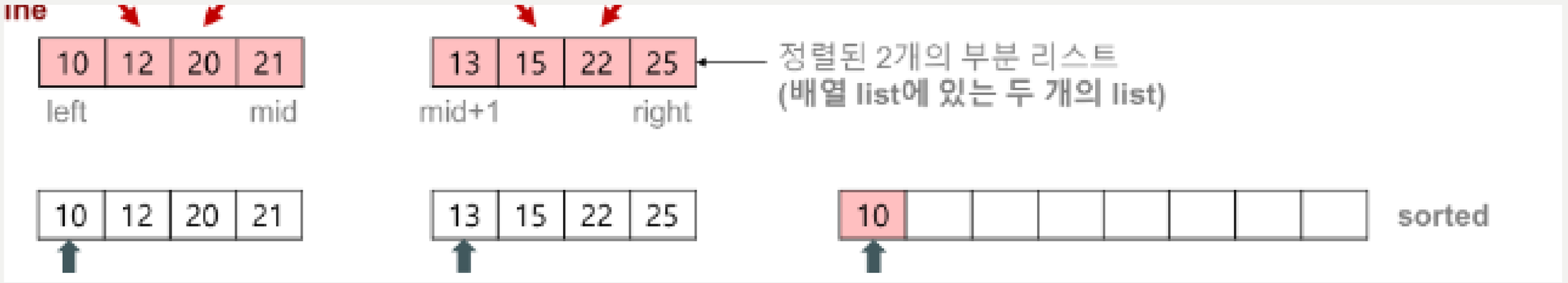
병합 정렬 코드

```
void merge(int *answer, int start, int middle, int end) {
    int left = start, temp = start;
    int right = middle + 1;

    while (left <= middle && right <= end) {
        if (answer[left] <= answer[right])
            temp_answer[temp] = answer[left++];
        else
            temp_answer[temp] = answer[right++];
        temp++;
    }

    if (left > middle) {
        for (int i = right; i <= end; ++i)
            temp_answer[temp++] = answer[i];
    }

    else {
        for (int i = left; i <= middle; i++)
            temp_answer[temp++] = answer[i];
    }

    for (int i = start; i <= end; i++) {
        answer[i] = temp_answer[i];
    }
}
```

병합 정렬 코드 합치기

```
int left = start, temp = start;
int right = middle + 1;
```

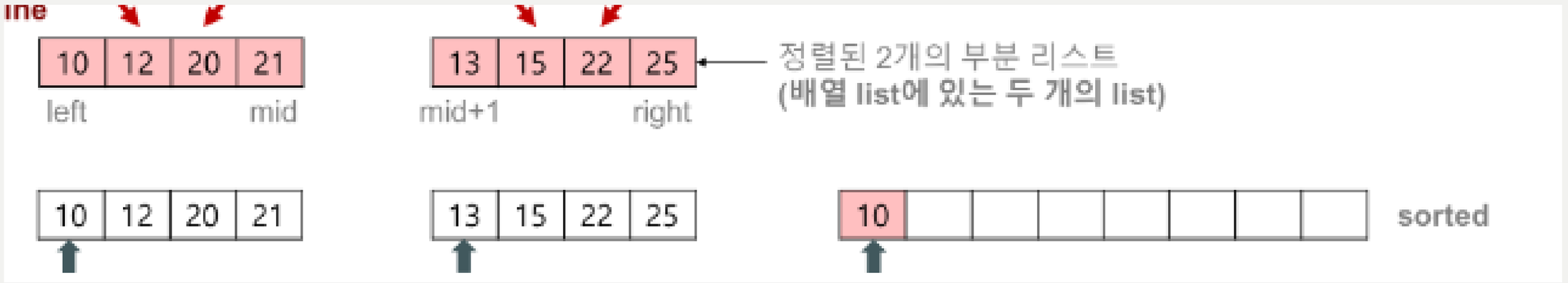
```
// left: 정렬된 왼쪽 리스트에 대한 인덱스
// right: 정렬된 오른쪽 리스트에 대한 인덱스
// temp: 정렬될 리스트에 대한 인덱스
/* 2개의 인접한 배열 list[left...mid]와 list[mid+1...right]의 합병 과정 */
```



병합 정렬 코드 합치기

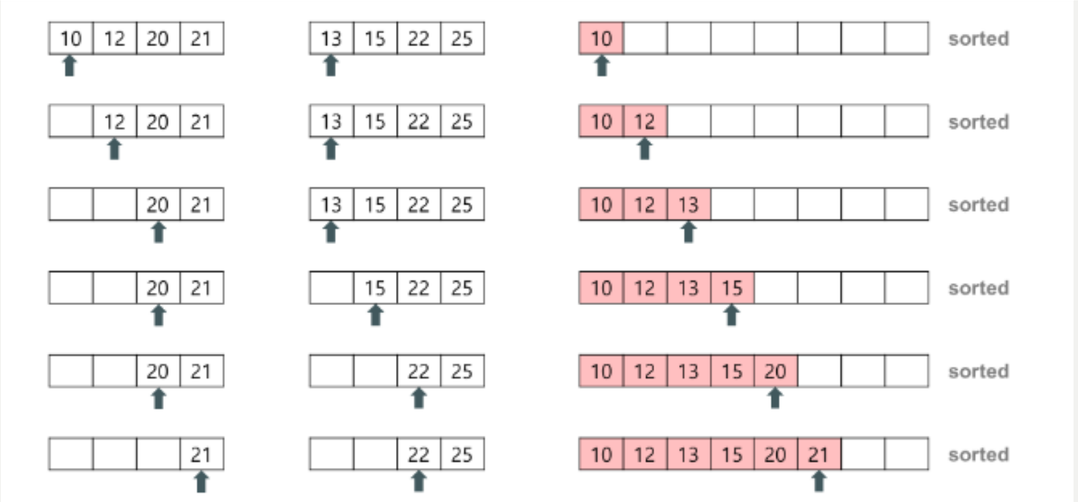
```
int left = start, temp = start;
int right = middle + 1;
```

```
// left: 정렬된 왼쪽 리스트에 대한 인덱스
// right: 정렬된 오른쪽 리스트에 대한 인덱스
// temp: 정렬될 리스트에 대한 인덱스
/* 2개의 인접한 배열 list[left...mid]와 list[mid+1...right]의 합병 과정 */
```



병합 정렬 코드

```
while (left <= middle && right <= end) {
    if (answer[left] <= answer[right])
        temp_answer[temp++] = answer[left++];
    else
        temp_answer[temp++] = answer[right++];
}
```



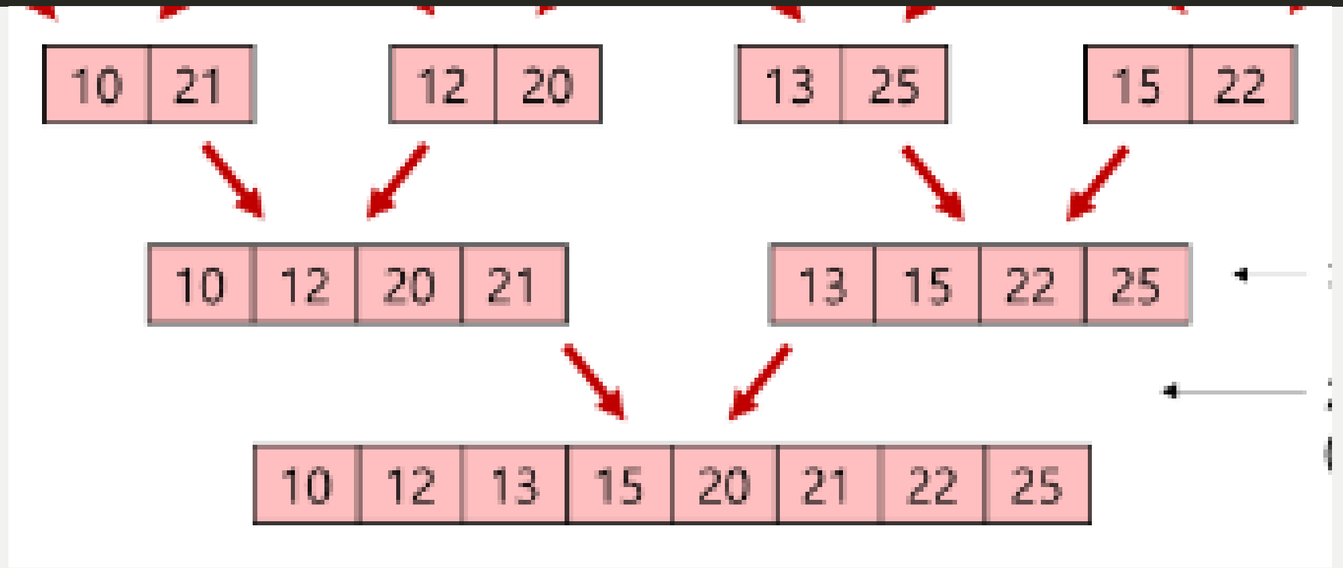
병합 정렬 코드

```
if (left > middle) {
    for (int i = right; i <= end; ++i)
        temp_answer[temp++] = answer[i];
}

else {
    for (int i = left; i <= middle; i++)
        temp_answer[temp++] = answer[i];
}
```

병합 정렬 코드

```
for (int i = start; i <= end; i++) {
    answer[i] = temp_answer[i];
}
```

```
// 배열 sorted[](임시 배열)의 리스트를 배열 list[]로 재복사
```



병합 정렬 코드

```
#include <iostream>
#define MAX_SIZE 8
using namespace std;

int temp_answer[MAX_SIZE];
```

병합 정렬 코드

```
int main() {
    int i;
    int n = MAX_SIZE;
    int list[n] = {21, 10, 12, 20, 25, 13, 15, 22};

    // 합병 정렬 수행(start: 배열의 시작 = 0, end: 배열의 끝 = 7)
    merge_sort(list, 0, n - 1);

    for (auto a : list) {
        cout << a << " ";
    }
    return 0;
}
```

병합 정렬 코드

```
int main() {
    int i;
    int n = MAX_SIZE;
    int list[n] = {21, 10, 12, 20, 25, 13, 15, 22};

    // 합병 정렬 수행(start: 배열의 시작 = 0, end: 배열의 끝 = 7)
    merge_sort(list, 0, n - 1);

    for (auto a : list) {
        cout << a << " ";
    }
    return 0;
}
```

병합 정렬 코드

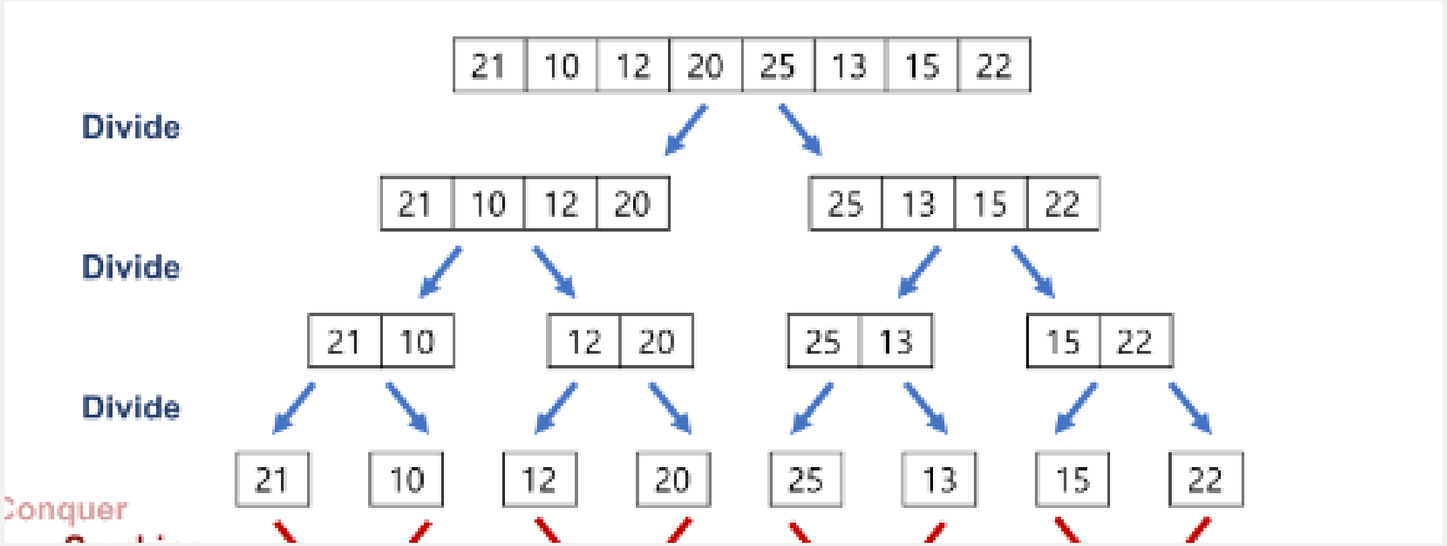
```c
void merge_sort(int *list, int start, int end) {
    if (start < end) { // start==end라는 뜻 즉 원소가 한개
        // 중간 위치를 계산하여 리스트를 균등 분할 -분할(Divide)
        int middle = (start + end) / 2;
        // 앞쪽 부분 리스트 정렬 -정복(Conquer)
        merge_sort(list, start, middle);
        // 뒤쪽 부분 리스트 정렬 -정복(Conquer)
        merge_sort(list, middle + 1, end);
        // 정렬된 2개의 부분 배열을 합병하는 과정 -결합(Combine)
        merge(list, start, middle, end);
    }
}
```

병합 정렬 코드

```
void merge(int *answer, int start, int middle, int end) {
    // left: 정렬된 왼쪽 리스트에 대한 인덱스
    // right: 정렬된 오른쪽 리스트에 대한 인덱스
    // temp: 정렬될 리스트에 대한 인덱스
    /* 2개의 인접한 배열 list[left...mid]와 list[mid+1...right]의 합병 과정 */
    int left = start, temp = start;
    int right = middle + 1;

    while (left <= middle && right <= end) {
        if (answer[left] <= answer[right])
            temp_answer[temp++] = answer[left++];
        else
            temp_answer[temp++] = answer[right++];
    }

    if (left > middle) {
        for (int i = right; i <= end; ++i)
            temp_answer[temp++] = answer[i];
    }

    else {
        for (int i = left; i <= middle; i++)
            temp_answer[temp++] = answer[i];
    }

    for (int i = start; i <= end; i++) {
        answer[i] = temp_answer[i];
    }
}
```

병합 정렬 시간 복잡도

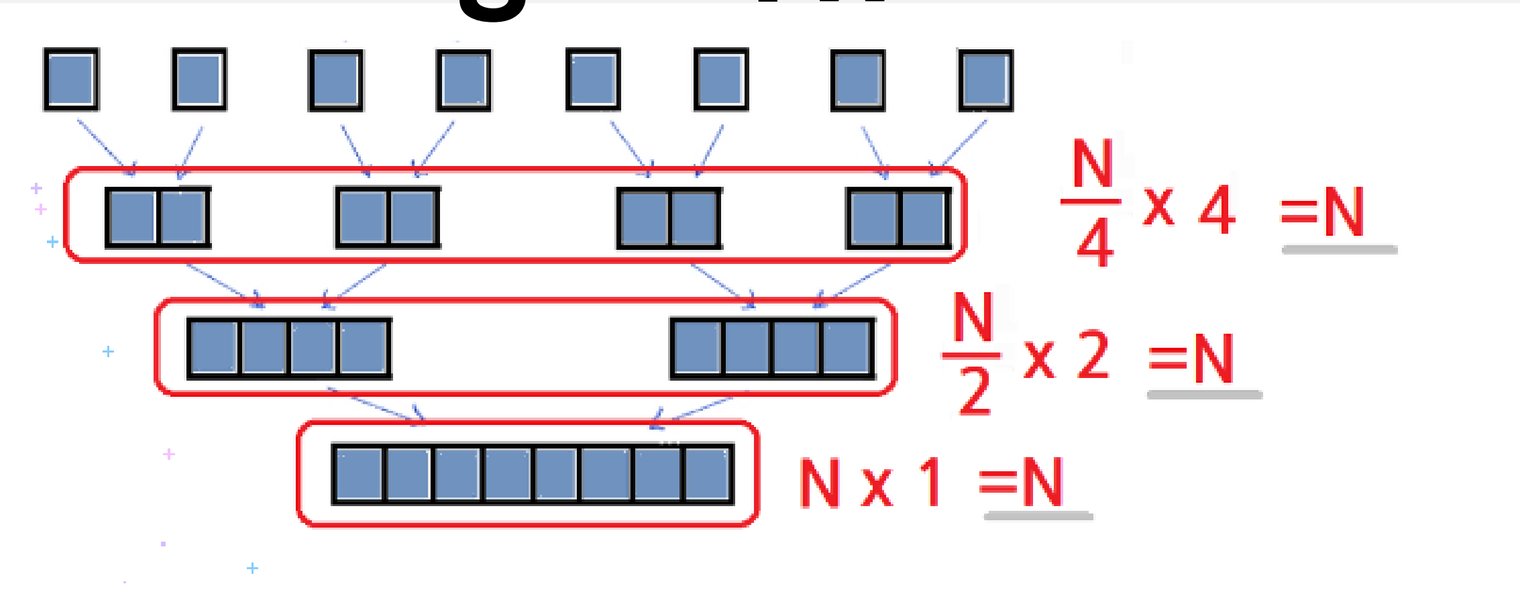
n log n 이유는?



리스트를 1개의 원소로 쪼개는데 필요한 시간 log_2 n

병합 정렬 시간 복잡도

n log n 이유는?

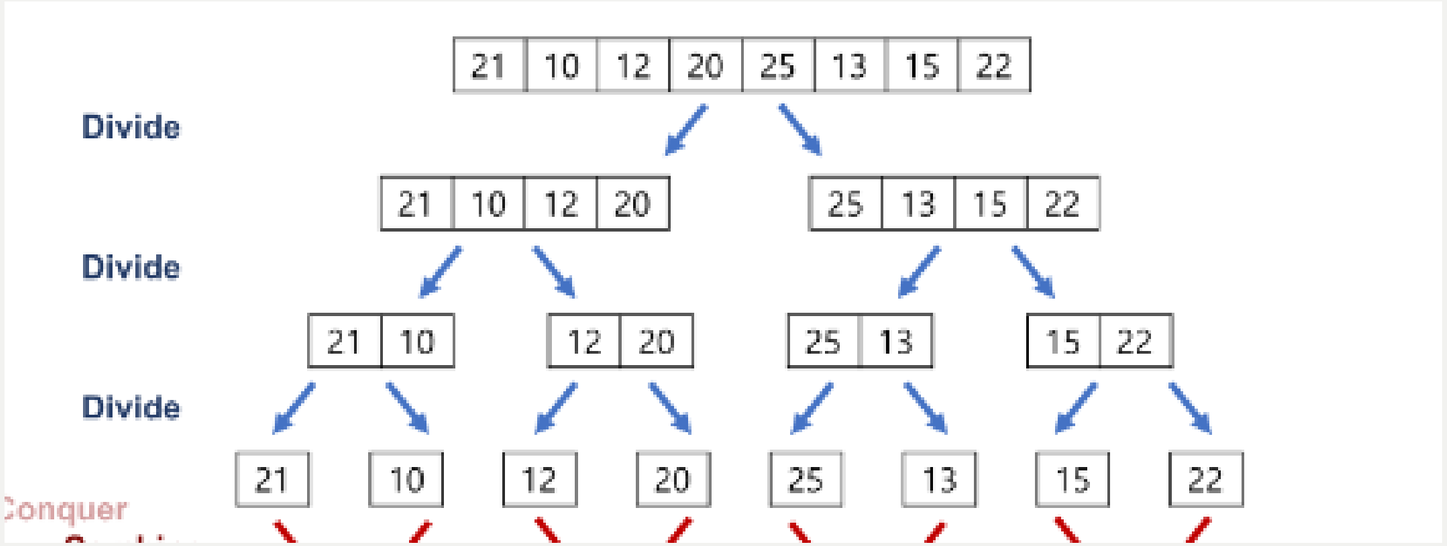


합칠 때 각 줄에서 필요한 비교 횟수

n

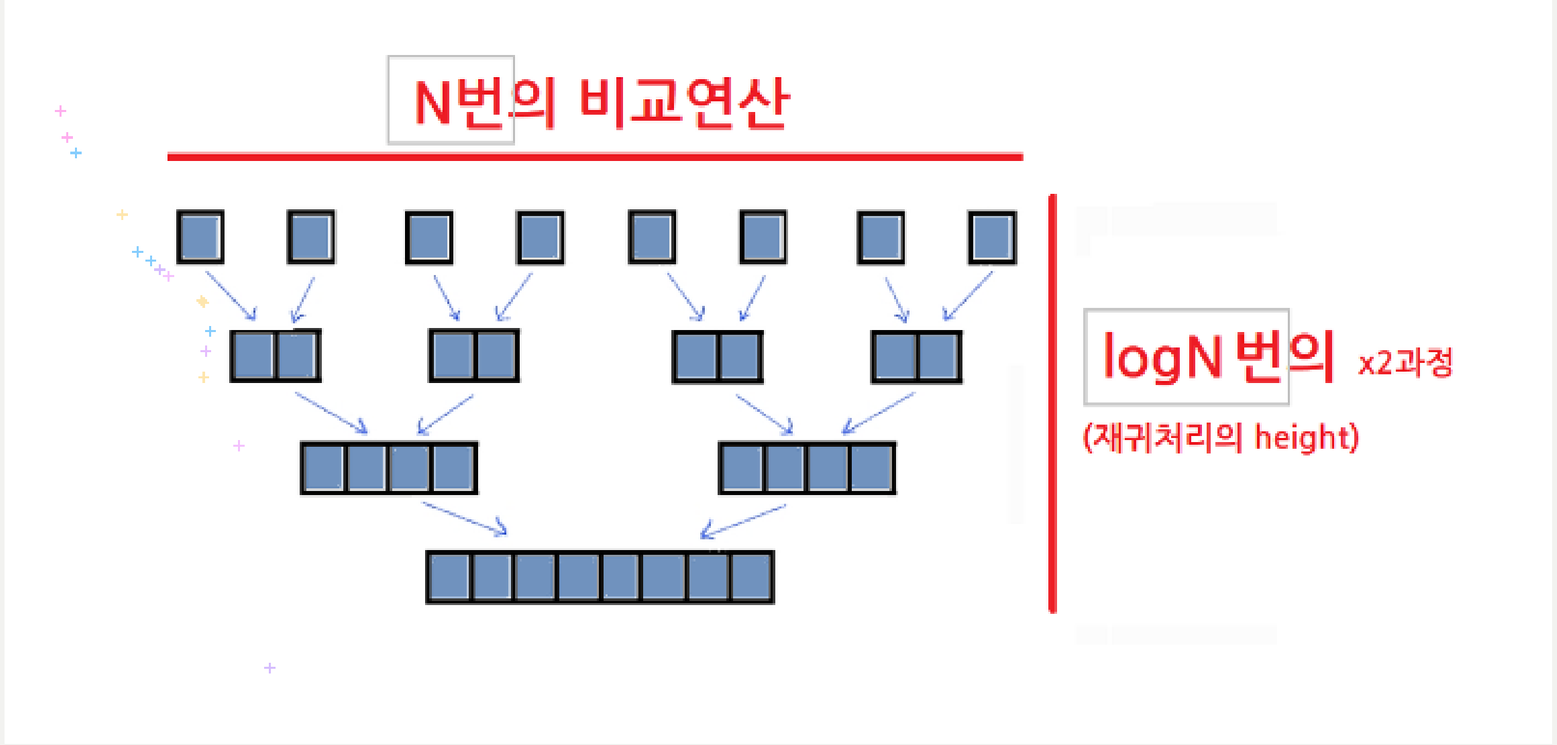
병합 정렬 시간 복잡도

n log n 이유는?



리스트를 1개의 원소로 합치는데 필요한 시간 log_2 n

병합 정렬 시간 복잡도



시간 복잡도 : N * log N

병합 정렬 장단점

장점 1. 인풋에 상관없이 안정적인 성능

단점 1. O(n)의 추가적인 공간이 필요하다.

퀵 정렬

> *특정한 값(Pivot)을 기준으로 큰 숫자와 작은 숫자를 구분하자*

퀵 정렬

합병 정렬은 주어진 리스트를 절반으로 나눠 분할하고,
퀵 정렬은 주어진 리스트를 **'파티셔닝'으로 분할**한다.

파티셔닝은 단순한 작업이다.

1. 정렬을 해야할 리스트에서 임의의 수를 고른다.
2. 나머지 수를 고른 수와 하나씩 비교한다.
3. 고른 수보다 작은 숫자는 왼쪽으로 옮기고, 고른 수보다 큰 숫자는 오른쪽으로 옮긴다.

퀵 정렬

> [1, 3, 9, 8, 2, 7, 5]

우리는 맨 끝에 있는 '5'를 고르기로 하자.

'5'보다 작은 수는 왼쪽에 두고,
'5'보다 큰 수는 오른쪽에 둔다.

결과적으로 리스트는 이런 순서가 된다.

> [1, 3, 2, **5**, 9, 8, 7]

퀵 정렬

이게 파티셔닝이다.

파티셔닝을 해도 여전히 모든 리스트가 정렬된 상태는 아니다.
'5'보다 앞에 있는 수들, '5'보다 뒤에 있는 수들은 여전히 정렬이 되지 않았다.

하지만 파티셔닝의 결과로 **'5'만큼은 확실하게 정렬**된다.

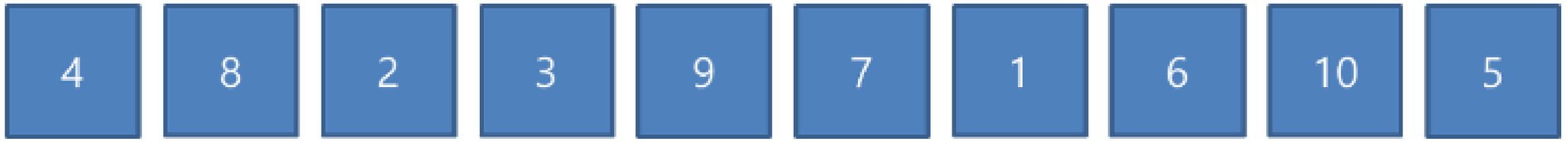
'5'의 양쪽을 정렬해도 어차피 '5'의 위치는 바뀌지 않기 때문이다.
따라서 우리가 고른 '5'라는 숫자는 정확히 제자리에 와있는 상태다.

우리가 고른 '5'라는 숫자를 **'피벗(pivot)'**이라고 부른다.

퀵 정렬

> *특정한 값(Pivot)을 기준으로 큰 숫자와 작은 숫자를 구분하자*

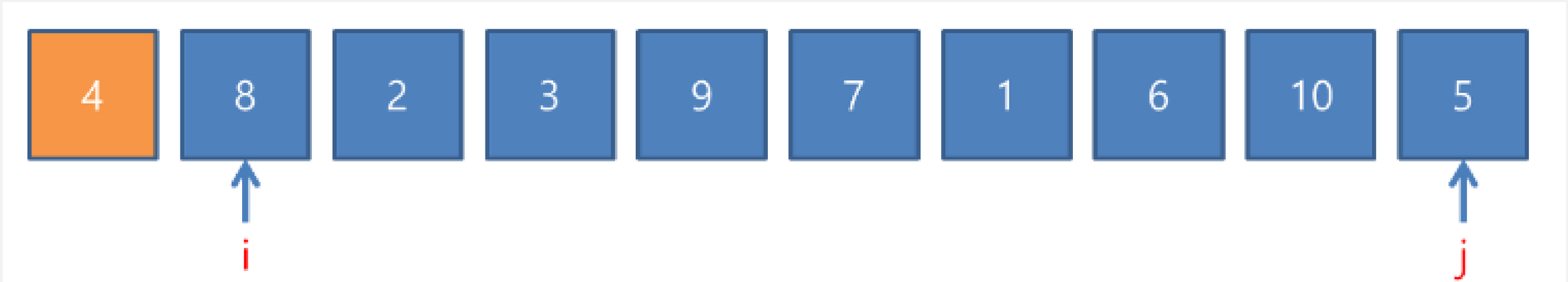
퀵 정렬



맨 앞의 숫자 4를 pivot으로 설정한다.

i, j 포인터를 설정한다.

퀵 정렬



i는 피봇을 제외한 처음 원소부터 피봇보다 큰 값을 찾는다.

피봇보다 큰 값은 8이다.

j는 끝에서부터 피봇보다 작은 값을 찾는다.

피봇보다 작은 값은 1이다.

퀵 정렬

i는 피봇을 제외한 처음 원소부터 피봇보다 큰 값을 찾는다.

피봇보다 큰 값은 8이다.

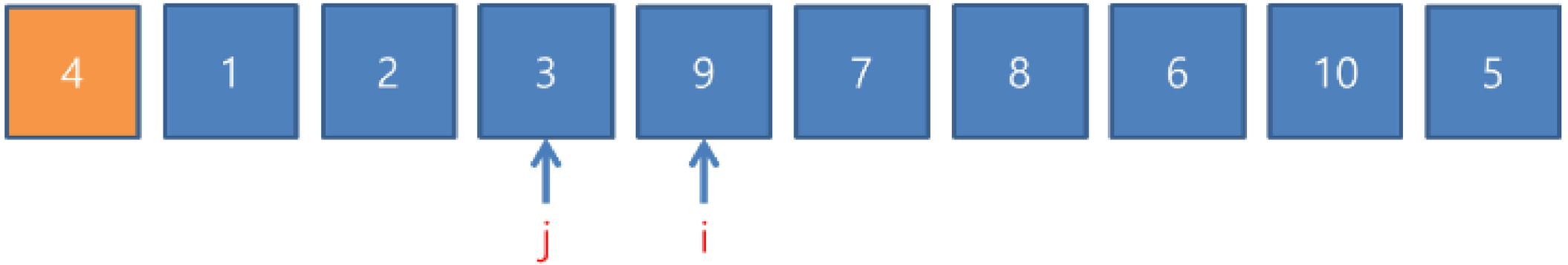
j는 끝에서부터 피봇보다 작은 값을 찾는다.

피봇보다 작은 값은 1이다.

퀵 정렬

1과 8을 바꾼다. (i번째와 j번째를 바꾼다.)

퀵 정렬



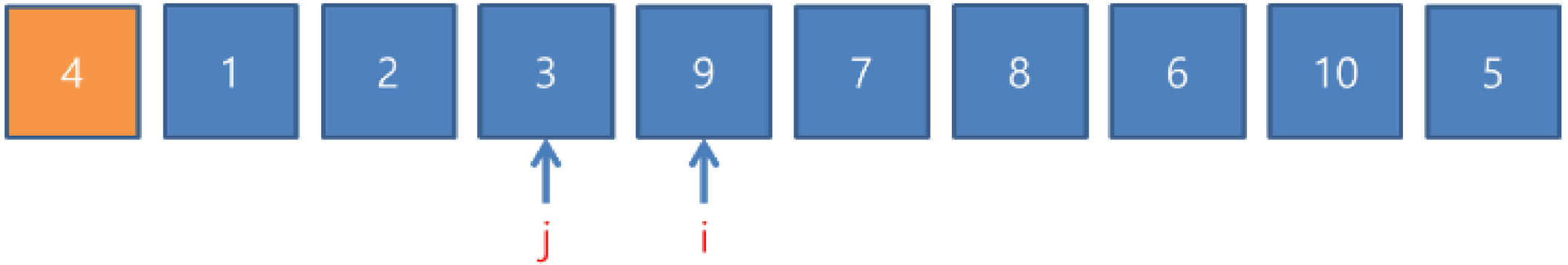
다시 i는 피봇보다 큰값을, j는 피봇보다 작은 값을 찾는다.

찾는도중에 i > j 가 되었다. (엇갈렸다.)

이럴 경우는 i와 j를 바꾸는 것이 아니라

피봇과 j를 바꾼다.

퀵 정렬



다시 i는 피봇보다 큰값을, j는 피봇보다 작은 값을 찾는다.

찾는도중에 i > j 가 되었다. (엇갈렸다.)

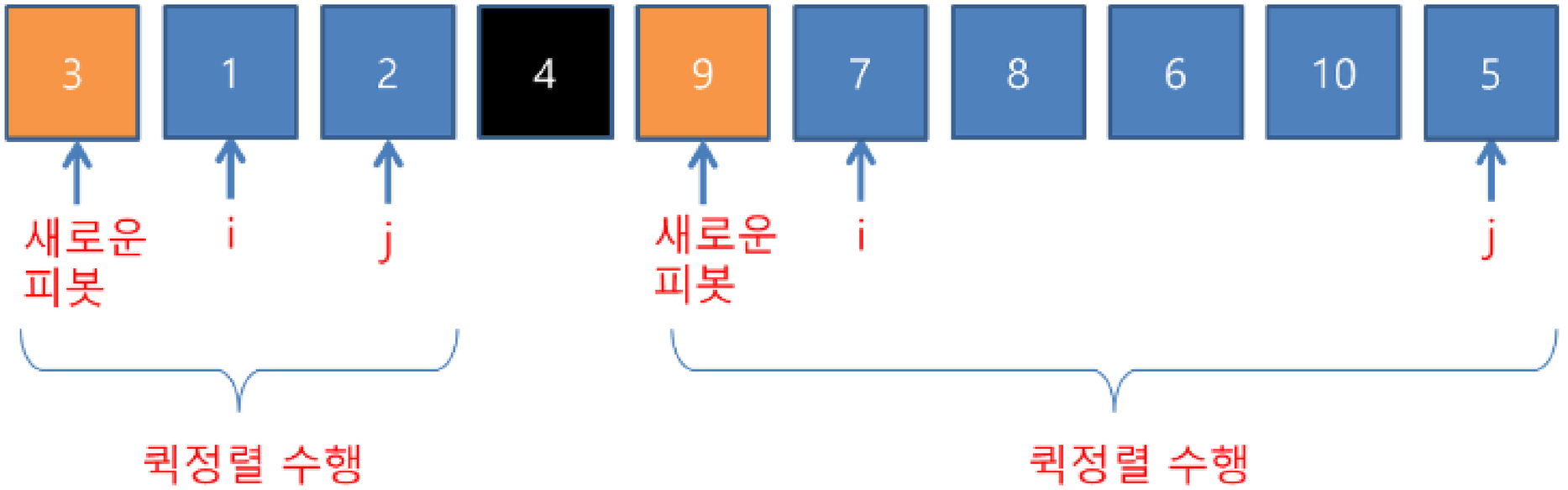
이럴 경우는 i와 j를 바꾸는 것이 아니라

피봇과 j를 바꾼다.

퀵 정렬

바꾸게 되면, 피봇(4)의 좌측으로는 피봇보다 작은값들만 존재하게 된다.

반대로 피봇(4)의 우측으로는 피봇보다 큰 값들만 존재하게 된다.



퀵 정렬

> *특정한 값(Pivot)을 기준으로 큰 숫자와 작은 숫자를 구분하자*

퀵 정렬

시작

```
quick_sort(data, 0, 9);
```

퀵 정렬

시작

```
void quick_sort(int *data, int start, int end) {
    if (start >= end) {
        // 원소가 1개인 경우
        return;
    }
```

퀵 정렬

변수 설정

```
int pivot = start;
int i = pivot + 1; // 왼쪽 출발 지점
int j = end;       // 오른쪽 출발 지점
```

퀵 정렬

```
while (i <= j) {
    // 포인터가 엇갈릴때까지 반복
    while (i <= end && data[i] <= data[pivot]) {
        i++;
    }
    while (j > start && data[j] >= data[pivot]) {
        j--;
    }

    if (i > j) {
        // 엇갈림
        int temp = data[j];
        data[j] = data[pivot];
        data[pivot] = temp;
    } else {
        // i번째와 j번째를 스왑
        int temp = data[i];
        data[i] = data[j];
        data[j] = temp;
    }
}
```

pivot 보다 작은수는 왼쪽
큰 수는 오른쪽으로 보내는 행동

퀵 정렬

```
while (i <= j) {
    // 포인터가 엇갈릴때까지 반복
    while (i <= end && data[i] <= data[pivot]) {
        i++;
    }
    while (j > start && data[j] >= data[pivot]) {
        j--;
    }

    if (i > j) {
        // 엇갈림
        int temp = data[j];
        data[j] = data[pivot];
        data[pivot] = temp;
    } else {
        // i번째와 j번째를 스왑
        int temp = data[i];
        data[i] = data[j];
        data[j] = temp;
    }
}
```

만약 엇갈리면 무조건 while문이 끝남

```
// 분할 계산
quick_sort(data, start, j - 1);
quick_sort(data, j + 1, end);
```

퀵 정렬

```c
void quick_sort(int *data, int start, int end) {
    if (start >= end) {
        // 원소가 1개인 경우
        return;
    }
    int pivot = start;
    int i = pivot + 1; // 왼쪽 출발 지점
    int j = end;       // 오른쪽 출발 지점
    while (i <= j) {
        // 포인터가 엇갈릴때까지 반복
        while (i <= end && data[i] <= data[pivot]) {
            i++;
        }
        while (j > start && data[j] >= data[pivot]) {
            j--;
        }
        if (i > j) {
            // 엇갈림
            int temp = data[j];
            data[j] = data[pivot];
            data[pivot] = temp;
        } else {
            // i번째와 j번째를 스왑
            int temp = data[i];
            data[i] = data[j];
            data[j] = temp;
        }
    }

    // 분할 계산
    quick_sort(data, start, j - 1);
    quick_sort(data, j + 1, end);
}
```

퀵 정렬

```c
void quick_sort(int *data, int start, int end) {
    if (start >= end) {
        // 원소가 1개인 경우
        return;
    }
    int pivot = start;
    int i = pivot + 1; // 왼쪽 출발 지점
    int j = end;       // 오른쪽 출발 지점
    while (i <= j) {
        // 포인터가 엇갈릴때까지 반복
        while (i <= end && data[i] <= data[pivot]) {
            i++;
        }
        while (j > start && data[j] >= data[pivot]) {
            j--;
        }
        if (i > j) {
            // 엇갈림
            int temp = data[j];
            data[j] = data[pivot];
            data[pivot] = temp;
        } else {
            // i번째와 j번째를 스왑
            int temp = data[i];
            data[i] = data[j];
            data[j] = temp;
        }
    }

    // 분할 계산
    quick_sort(data, start, j - 1);
    quick_sort(data, j + 1, end);
}
```

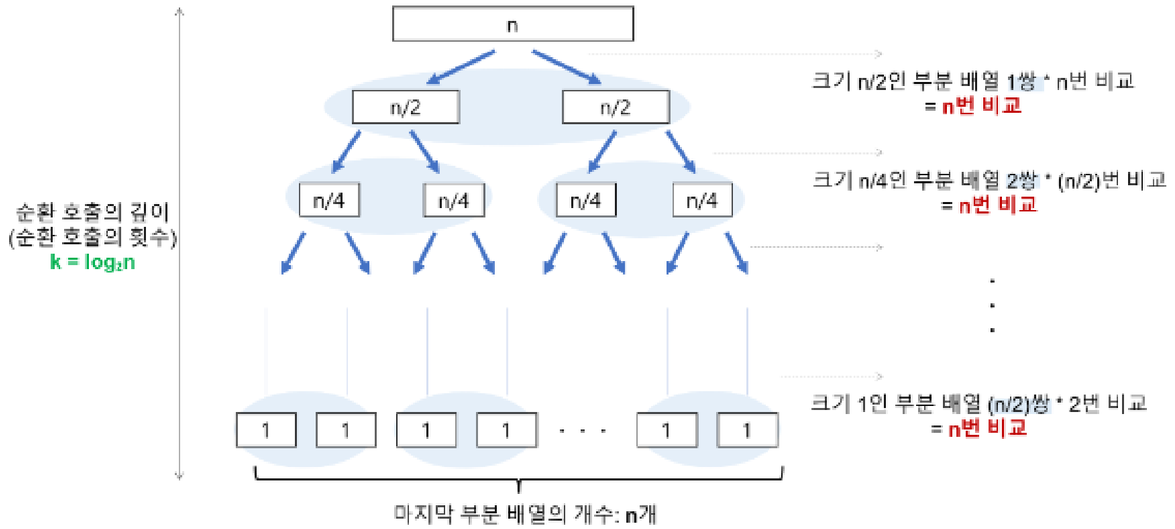
퀵 정렬

'분할 정복' 알고리즘으로 평균속도가 O(N * logN) 이다.

최악의 경우 $T(n) = O(n^2)$

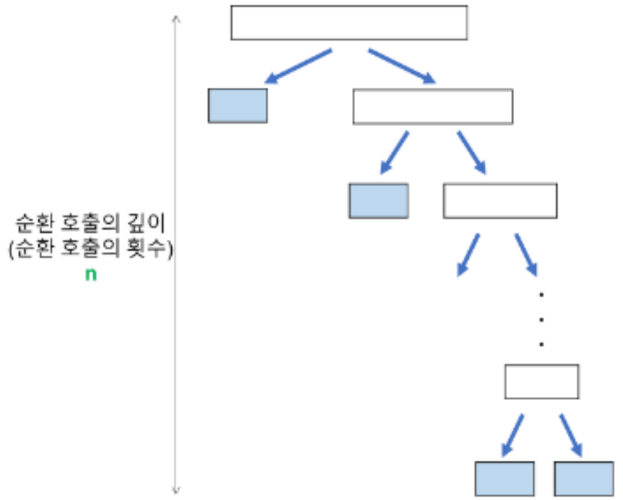
- 평균 $T(n) = O(n\log_2 n)$
- 시간 복잡도가 $O(n\log_2 n)$를 가지는 다른 정렬 알고리즘과 비교했을 때도 가장 빠
- 퀵 정렬이 불필요한 데이터의 이동을 줄이고 먼 거리의 데이터를 교환할 뿐만 아니

퀵 정렬

- 최선의 경우
  - 비교 횟수
    - 

    - *순환 호출의 깊이*
      - 레코드의 개수 n이 2의 거듭제곱이라고 가정(n=2^k)했을 때, n=2^3의 경우, 2^3 -> 2^2 -> 2^1 -> 2^0 순으로 줄어들어 순환 호출의 깊이가 3임을 알 수 있다. 이것을 **일반화하면** n=2^k의 경우, k(k=log₂n)임을 알 수 있다.
        - k=log₂n
    - *각 순환 호출 단계의 비교 연산*
      - 각 순환 호출에서는 전체 리스트의 대부분의 레코드를 비교해야 하므로 평균 n번 정도의 비교가 이루어진다.
        - 평균 n번
    - 순환 호출의 깊이 * 각 순환 호출 단계의 비교 연산 = **nlog₂n**

퀵 정렬

- 최악의 경우
  - 리스트가 계속 불균형하게 나누어지는 경우 (특히, 이미 정렬된 리스트에 대하여 퀵 정렬을 실행하는 경우)
  - 

  - 비교 횟수
    - *순환 호출의 깊이*
      - 레코드의 개수 n이 2의 거듭제곱이라고 가정(n=2^k)했을 때, 순환 호출의 깊이는 n임을 알 수 있다.
        - n
    - *각 순환 호출 단계의 비교 연산*
      - 각 순환 호출에서는 전체 리스트의 대부분의 레코드를 비교해야 하므로 평균 n번 정도의 비교가 이루어진다.
        - 평균 n번
    - 순환 호출의 깊이 * 각 순환 호출 단계의 비교 연산 = **n^2**
  - 이동 횟수
    - 비교 횟수보다 적으므로 무시할 수 있다.
  - 최악의 경우 T(n) = **O(n^2)**

문제

merge sort 구현