

# <12강 정렬 (SORT), 김성현, 2019/07/23>

12.1 정렬이란?	12.2 선택정렬	12.3 삽입정렬	12.4 버블정렬	12.5 퀵정렬 <swap>	12.6 합병정렬	12.7 힙정렬	12.8 힙정렬 (알에서 이미 정렬)
				12.11 정렬 알고리즘	12.10 정렬 알고리즘	12.9 기수정렬 (11)	

12.11 정렬 알고리즘  
: 많이 사용되는 정렬

## part2 12.1 정렬이란?

• 정렬 (sort)이란?

정렬 (sort)은 크기 비교가 가능한 데이터를 오름차순 (ascending order) - (나 내림차순 (descending order) = 큰 나열하는 방법

정렬문제: 과학 기술 분야에서 가장 기본적이고, 중요한 문제 중 하나임 (정렬이 커지면 문제도 커진다)

정렬은 (문자 (string))을 크기 비교가 가능한 (예: 영어 사전에서 단어를 따라 알파벳 순으로 정렬함)

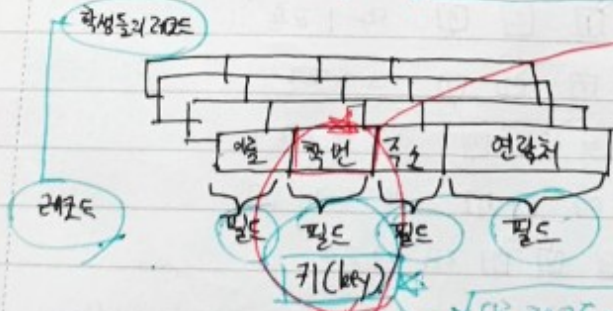
알고리즘 → sorting과 searching이 굉장히 많은 비중 차지함 !!

• 정렬의 대상

일반적으로 정렬해야 할 대상은 레코드 (record) 들임.

각 레코드는 하나의 필드들 (fields)로 구성됨.

각 레코드의 키 (key) 필드는 다른 레코드들과 구별하기 위하여 사용되는 필드임 (예: 학번)



이 key 필드가 "정렬"할 때 사용됨!! (같은 학번에 ... 많이 사용되다...)

특정 대상 (학생)에 대해

모든 정보를 모아놓은

하나의 레코드이다.

다른 레코드에서 정렬할 때 비교할 때 나타나는 것은 key로 해준다!!



정렬 알고리즘 개요

정렬 알고리즘

선택(Selection) 정렬, 삽입(Insertion) 정렬, 버블(Bubble) 정렬 등.

분할정복 방식: 합병(Merge) 정렬, 퀵(Quick) 정렬, 힙(Heap) 정렬 등

정렬 알고리즘의 특징 기준

정렬을 할 때 비교할 4 있는 데이터

비교에서는 시간이 걸림.

비교 횟수 및 이동 횟수로 따질. 이러한 데이터 처리를 반복적으로 반복.

안정적(stable) 정렬 알고리즘. / 각 요소의 원래 위치를 유지하며, 같은 값의 요소는 안정적으로 정렬.

정렬 후에 모든 키 값을 가지는 레코드들이 상대적인 위치가 정렬 전과 같은 알고리즘.

→ 키는 4 개씩 생각해서, 5 개씩 생각하면, 앞부분은 5 개, 뒤를 생각하면 정렬하기 전에도 5 개, 하리후서.

PA5~450 12.2 선택 정렬(selection sort)

array가 있다고 가정

정렬 후에 5 개, 5 개 순서까지

안정적인 정렬 알고리즘이겠지!!

정렬을 할 때

정렬을 할 때

(오른쪽 요소들!!)

정렬을

오른쪽의 정렬

배열에

가져오면

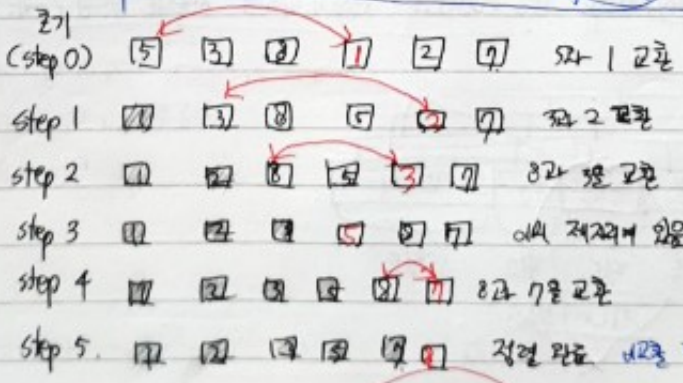
선택된 원소 리스트와 정렬 안된 오른쪽 리스트 가정 (초기에는 왼쪽 리스트가 비어 있고,

정렬할 리스트는 오른쪽 리스트일)

오른쪽 리스트에서 (최소값) 선택하여(selection) 왼쪽 리스트의 현재 항목과 교환(Swap)함.

- 왼쪽 리스트 크기 1 증가
- 오른쪽 리스트 크기 1 감소
- 이 step을 반복하여 오른쪽 리스트가 비면 정렬 완료

6 개



이제 주어진  
→ 9 개  
6 개  
가져오면

⇒ 이제 주어진  
9 개 (11)만 가져오면

Step 5

가져오면

오른쪽 element의

가져오면!!



• 선택 정렬 알고리즘 및 프로그램

selection-sort (A, n) <sup>(1) n개의 index</sup>

for i ← 0 to n-2 do // i는 오른쪽 리스트의 첫 번째 인덱스 (1번일 때 step 1일)

least ← A[i], A[i+1], ..., A[n-1] 중에서 최소값의 인덱스 j

A[i]와 A[least]를 교환

→ 최소값 찾기 위한 범위가 하나씩 줄어듦

#define SWAP(x, y, t) ((t)=(x), (x)=(y), (y)=(t))

void selection-sort (int list[], int n) {

int i, j, least, temp;

for (i = 0; i < n-1; i++) {

// i는 오른쪽 리스트의 첫 번째 인덱스

least = i;

// least를 오른쪽 리스트의 첫 번째로 초기화. → 초기 step과 끝 step 차이

for (j = i+1; j < n; j++)

// j는 오른쪽 리스트의 둘째 인덱스로부터 n-1까지 증가

if (list[j] < list[least])

→ j보다 더 작거나 같아야 함!

least = j;

// least는 오른쪽 리스트에서 최소값의 인덱스

SWAP(list[i], list[least], temp);

// list[i]와 list[least]를 교환

순서가 맞지 않을 때는 오른쪽 리스트에서 최소값을

오른쪽의 첫 번째로 순서가 맞아서 가장 왼쪽으로

least = i = j (가 변함) X!!

• 선택 정렬 복잡도

number of element → 2차로 증가함

선택 정렬 알고리즘 (n-1)번의 step 4행

비교 횟수

각 step에서 (n-1), (n-2), ... 1번의 비교 4행

비교 횟수: (n-1) + (n-2) + ... + 1 = n(n-1)/2 =  $O(n^2)$

이동 횟수

각 step에서 (3)번 이동 수행 → swap은 2번 수행!!

이동 횟수:  $3 \times (n-1) = O(n^2)$  ⇒ 3번 이동 = 1회 swap

4행 복잡도:  $O(n^2)$

안정적인 정렬 방법 아님

swap하면서 같 key도 순서 바뀌어 버림



p.450 ~ 455. 12.3 삽입 정렬 (insertion sort)

• 삽입 정렬 (insertion sort)

가장 내적인 요소를 찾기 위해 선별과 다른 점.

□ **정렬된 왼쪽 리스트**와 **정렬 안된 오른쪽 리스트** 가정 (포기에는 왼쪽 리스트에 현재 항목만 있음)

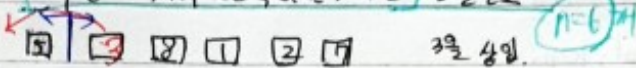
□ **오른쪽 리스트의 현재 항목을 왼쪽 리스트의 바른 위치에 삽입 (insertion)** → 삽입이 될 때 sorting이 되었다

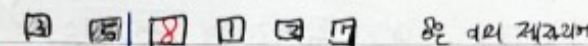
□ 이때 삽입할 위치 오른쪽에 있는 항목을 오른쪽으로 한 위치씩 이동시켜야 함.

□ 왼쪽 리스트에 증가


□ 오른쪽 리스트 크기 1 감소

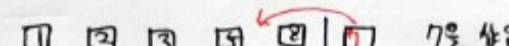
□ 이 step을 반복하여 오른쪽 리스트가 비어 정렬 완료

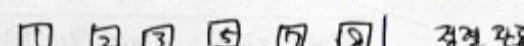
→ 초기 (step 0)  3을 삽입.

step 1  5은 4의 자리로 이동

step 2  1을 삽입

step 3  2를 삽입.

step 4  4를 삽입.

steps 5  정렬 완료

왼쪽 리스트는 정렬됨, 오른쪽 리스트는 분할됨 (4로 나뉘게).

오른쪽 리스트에서 현재 항목을 왼쪽으로 이동

• 삽입 정렬 알고리즘 및 프로그램

오른쪽 리스트의  
첫 번째 항목을  
key에 저장.

insertion-sort(A, n)

for  $i \leftarrow 1$  to  $n-1$  do

// i는 오른쪽 리스트 현재 인덱스 (i=1일 때 step 1번).

key  $\leftarrow A[i]$ ;

$j \leftarrow i-1$ ;

while  $j > 0$  and  $A[j] > \text{key}$  do

// key에 오른쪽 리스트의 현재 항목 저장 (이전 모든 항목을 위해)  
// 현재 왼쪽 리스트의 마지막 항목이 현재 key보다 (비교) 큰 (이전)을 위해  
// A[j] > key 동안 (j를 감소시키면서)

$A[j+1] \leftarrow A[j]$ ;

// A[j]를 오른쪽으로 이동

$j \leftarrow j-1$ ;

← 마지막으로 현재 넣을 공간 확보

$A[j+1] \leftarrow \text{key}$ ;

// key를 A[j+1]에 삽입

감소시키면서 j+1이 넣는 위치 (j < j-1이 될 때까진)

void insertion\_sort(int list[], int n) {

int i, j, key;

for ( $i=1$ ;  $i < n$ ;  $i++$ ) {

// i는 오른쪽 리스트 현재 인덱스 (i=1일 때 step 1번)

key = list[i];

// key에 오른쪽 리스트의 현재 항목 저장

for ( $j=i-1$ ;  $j >= 0$  and  $list[j] > \text{key}$ ;  $j--$ )

// list[j] > key 만족할 동안 감소시키면서 끝난

list[j+1] = list[j];

// A[j]를 오른쪽으로 이동

list[j+1] = key;

// key를 A[j+1]에 삽입.

key값은 다른 모든 위치가 key보다 작거나 같을 때까진

만족할 때까지 반복

이렇게 하는 것이다.

(그래서 A[j]를 오른쪽으로 이동)

정답!!

이렇게 반복해서

key를 넣는 것이다!!



## 삽입 정렬 복잡도

삽입 정렬 알고리즘  $(n-1)$  회 step 수행.

최선 (이미 정렬된 경우)

이 정렬되어 있는 모든 원소를 비교해서 (step 4번)

비교 회수: 각 step에서 1회 비교 수행하여  $n-1 = O(n)$

이동 회수: 각 step에서 2회 이동 수행하여  $2 * (n-1) = O(n)$

최악 (역순으로 정렬된 경우)

비교 회수: 각 step에서 1, 2, ...,  $(n-1)$  회 비교 수행하여  $n(n-1)/2 = O(n^2)$

이동 회수: 비교 후 1회 이동 및 각 step에서 2회 이동하여  $n(n-1)/2 + 2 * (n-1) = O(n^2)$

최악인 경우 이동 회수가 큰 편에서 레코드가 큰 경우 발생함.

일반적으로 1회 이동은 1회 비교보다 더 시간적 절감.

안정된 정렬 방법.

## 12.4 버블 정렬

버블 정렬 (bubble sort) 인접한 2개끼리 순서가 맞지 않으면 교환!!

인접한 2개의 항목을 비교하여 순서대로 되새기지 않으면 서로 교환.

이 비교 및 교환 (bubble) 과정을 리스트의 왼쪽 시작부터 오른쪽으로 가면서 반복 (한 step)

step 1 (0 ~ n-1까지 범위)가 끝되면 최대값이 가장 오른쪽에 위치하게 됨.

(다음 2절에 step 1의 수행 과정이며 어떤 최대값이 올 것).

step 2 (0 ~ n-2까지 범위)에는 가장 오른쪽 1개 항목은 이미 최대값이니 제외하고 반복.

...

### Step 1 수행 과정

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

2 1 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

3 1 2 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

3 4 1 2 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

3 4 5 1 2 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

3 4 5 6 1 2 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

비교한 인접한 2개 항목은 순서대로 표시함.

step 1 완료 후, 최대값이 가장 오른쪽에 위치하게 됨.

이제 step 2

(이제 step 2 수행)



## • 버블 정렬 알고리즘의 프로그램

BubbleSort(A, n)

for  $i \leftarrow n-1$  to 1 do    //  $i$ 는 이 step의 외곽 끝 (step 1은  $i=n-1$ 일 때일)  
 for  $j \leftarrow 0$  to  $i-1$  do    //  $j$ 와  $j+1$ 은 비교할 항목들의 인덱스  
      $j$ 와  $j+1$ 번째의 요소가 크기 순이 아니면 교환

SWAP

#define SWAP(x, y, t) { (t)=(x), (x)=(y), (y)=(t); }

void bubble-sort(int list[], int n) {

    int i, j, temp;

    for ( $i=n-1$ ;  $i>0$ ;  $i--$ ) {    //  $i$ 는 이 step의 외곽 끝 (step 1은  $i=n-1$ 일 때일). → step 1은  $i=n-1$

        for ( $j=0$ ;  $j<i$ ;  $j++$ )    //  $j$ 와  $j+1$ 은 비교할 항목들의 인덱스

            if ( $list[j] > list[j+1]$ )    //  $list[j] > list[j+1]$  이면 교환.

                SWAP(list[j], list[j+1], temp);

    }

}

→ step 1은  $i=n-1$   
 →  $i$ 는 이 step의 외곽 끝  
 →  $j$ 와  $j+1$ 은 비교할 항목들의 인덱스  
 →  $list[j] > list[j+1]$  이면 교환.  
 → step 1은  $i=n-1$   
 →  $i$ 는 이 step의 외곽 끝  
 →  $j$ 와  $j+1$ 은 비교할 항목들의 인덱스  
 →  $list[j] > list[j+1]$  이면 교환.

## • 버블 정렬 복잡도

• 버블 정렬 알고리즘은  $(n-1)$ 회 step 4행

□ 최선 (이미 정렬된 경우)

□ 비교 회수: 각 step에서 1, 2, ...,  $(n-1)$ 회 비교 수행하여  $n(n-1)/2 = O(n^2)$

□ 이동 회수:  $0 = O(1)$     정렬이 되어 있으면 비교는 해주지 않음

□ 최악 (역으로 정렬된 경우) → 정렬되어 있지 않다면 이동은 사실상 필요

□ 비교 회수: 각 step에서 1, 2, ...,  $(n-1)$ 회 비교 수행하여  $n(n-1)/2 = O(n^2)$

□ 이동 회수:  $3 \times$  비교 회수 =  $3 \times n(n-1)/2 = O(n^2)$     [비교 값들]

□ 최악의 경우 이동 회수가 커서 알고리즘을 풀 경우 불리함.

□ 일반적으로 1회 이동은 1회 비교보다 더 오래 걸림.

□ 안정된 정렬 방법임.

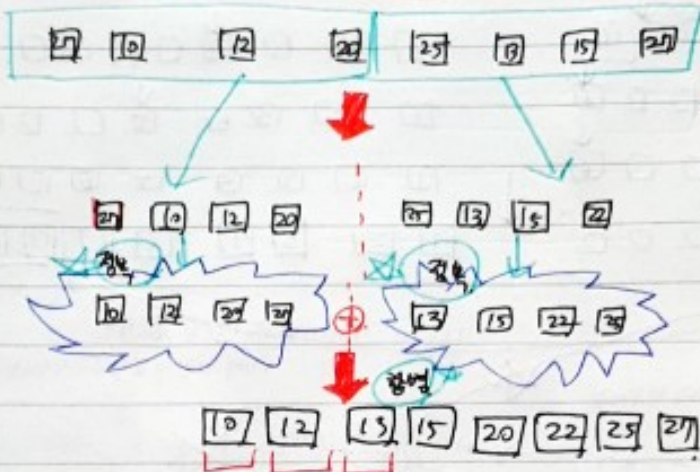
→ 순서가 같음과 같은 값을 가진 여러 원소를 정렬한 후의 순서가 바뀌지 않음



• 합병 정렬(merge sort) 여러 작은 문제를 작은 단위로 쪼개고, 이를 차례대로 풀어나가는 방법

□ 분할 정복(divide and conquer)이란 방법 사용. → 큰 문제를 작은 2개 문제로 나눠서 풀어보고, 다시 이 두개를 합쳐서 풀 해답으로 만든다!!

- ① 분할(divide): 배열을 같은 크기의 2개의 부분 배열로 나눈다.
- ② 정렬(sort): 2개의 부분 배열을 각각 정렬한다 (재귀 호출 필요)
- ③ 합병(merge): 정렬된 2개의 부분 배열을 합병한다

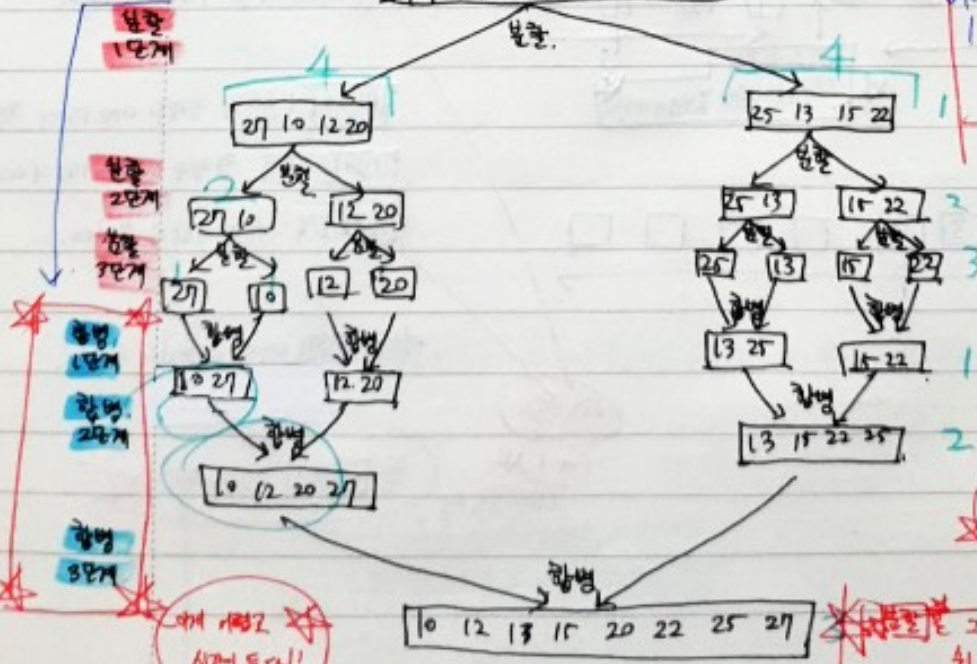
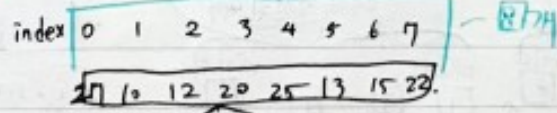


★ 2분할하는 건 아니고, 정렬된 2개를 다시 합병시켜서, 하나로 정렬이 되게 만드는 거야, "정렬되게 합친다"가 바로 "합병"인 것이다!!

반으로 똑같이 작은 2개의 array를 만든다!! 그걸 정렬(정복)!!

2개(부분)에서 작은게부터 차례대로 내려서 이어지면서 합병된다!!

합병 정렬 전체 과정



어떻게 분할 하려든 것... 분할, 합병하면서 정렬이 된다!! 배열 크기 => n개 1/2, 1/4, 1/8... n개 만큼의 분할, 1/8, 1/16... n개의 분할이 필요해

참고: 배열 크기가 n이면  $\log_2 n$  단계의 분할 안 합병 필요 (그림에서  $n=8$ 이면  $\log_2 8=3$  단계의 분할 및 합병 필요)

★ 분할도 단계별로 새로 이뤄지고, 합병도 새로 이뤄진다!!

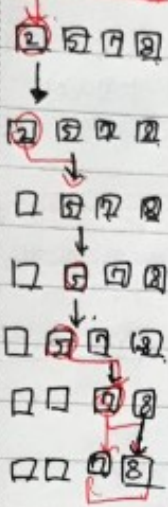
★ 그냥 반씩 나눠주는 것만으로도 최고 문제라기엔, (문제가 다) 시간복잡도도 좋고, 분할이든 해서 메모리도 아낄 수 있다!!

★ 이게 어렵고 시간 오래 걸린다!!

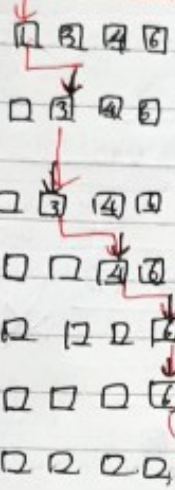


• 합병 과정

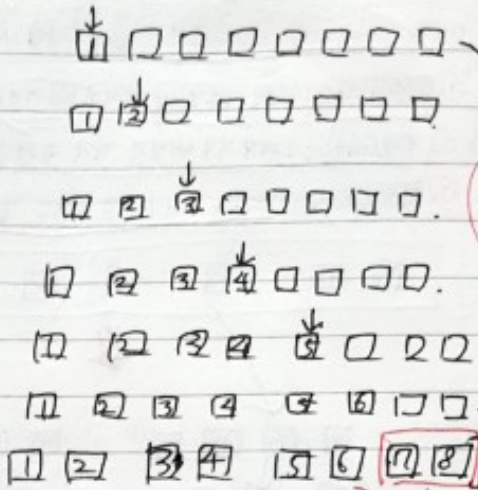
정렬된 배열 A



정렬된 배열 B



합병된 배열 C



배열 A 및 B의 순서를 비교하여 작은 것을 배열 C에 복사하는 작업은 반복함

위 작업 끝난 후 배열 A 혹은 B에서 남은 원소들은 일괄적으로 배열 C에 복사

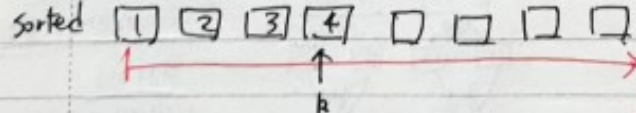
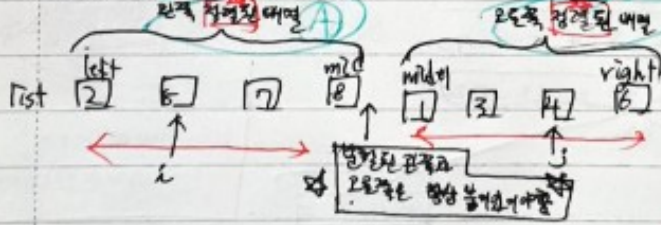
A, B의 각 요소들을 하나씩 비교해서 배열 C에 복사(정렬)하는데 작은 값이 발견되면 배열 C에 가장 앞에 정렬되어진다!!

남은 것 2개는 (원래의) 순서대로 옮긴다.

<이미 정렬된 것만 (원래의) 순서대로 복사>

각 요소를 <작은 값>을 구하는 것!!

• 합병 과정에 사용되는 index들



• left: 왼쪽 정렬된 배열 list의 첫 index.

• mid: 왼쪽 정렬된 배열 list의 마지막 index.

• i: 왼쪽 배열 list의 index.

• mid+1: 오른쪽 정렬된 배열 list의 첫 index.

• right: 오른쪽 정렬된 배열 list의 마지막 index.

• j: 오른쪽 배열 list의 index.

• k: 합병된 배열 sorted의 index.

mid + 1 (현재 비교 A)

중간 index 비교



• 합병 정렬 프로그램.

```
void merge_sort (int list[], int left, int right)
{
    int mid;
    if (left < right) {
        mid = (left + right) / 2; // 분할
        merge_sort (list, left, mid); // 정렬 왼쪽
        merge_sort (list, mid + 1, right); // 정렬 오른쪽
        merge (list, left, mid, right); // 합병
    }
}
```

int sorted[1000]; // 추가 공간이 필요.

void merge (int list[], int left, int mid, int right)

```
{
    // i는 정렬된 왼쪽 배열 list[]의 인덱스 (left ~ mid)
    // j는 정렬된 오른쪽 배열 list[]의 인덱스 (mid + 1 ~ right)
    // k는 정렬할 배열 sorted[]의 인덱스 (left ~ right)
    int i = left, j = mid + 1, k = left, l;

    // 분할 정렬된 list의 합병.
    while (i <= mid & j <= right) {
        if (list[i] <= list[j])
            sorted[k++] = list[i++]; // 왼쪽 배열 복사.
        else
            sorted[k++] = list[j++]; // 오른쪽 배열 복사.
    }

    if (i > mid)
        for (l = j; l <= right; l++)
            sorted[k++] = list[l];
    else
        for (l = i; l <= mid; l++)
            sorted[k++] = list[l];

    // 정렬된 배열 sorted[]를 원래 배열 list[]로 복사.
    for (l = left; l <= right; l++)
        list[l] = sorted[l];
}
```

여기서  
정렬된 배열!!



## • 합병 정렬 복잡도

- 합병 정렬은 총  $(\log n)$  단계의 합병을 할 것임. (n개 = 2진법)
- **비교 복잡도**: 각 합병 단계에서  $n$ 번의 비교 연산을 수행하므로  $n \log n = O(n \log n)$ .
- **이동 복잡도**: 각 합병 단계에서  $2n$ 번의 이동 발생하므로  $2n \log n = O(n \log n)$ .
- 합병 정렬은 최적, 최악, 평균 모두  $O(n \log n)$ 의 시간 복잡도를 가진다.
- 안정된 정렬 방법임. (두 개가 같은 경우는 절대 순서가 바뀌지 않음!!)

## p.470 ~ 477 12.7 퀵 정렬

### • 퀵 정렬 (quick sort)

#### □ 분할 정복 방법 4종

□ 리스트를, 가장 왼쪽 항목인 피벗 (pivot)을 기준으로 2개의 부분 리스트로 **비균등 분할** 하고 (별 2일),

각각의 부분 리스트를 다시 퀵 정렬함 (재귀 호출). (각 부분을 다룰 때 또 피벗 = 2로 나누는 것) (매번 나눌 때는 피벗을 고르면 되는데)

□ 평균적으로 가장 빠른 정렬 방법.



(별 2일)  
비균등 분할 된 것임.  
원래 4에 1을 9보다 작고, 8은 9보다 작고, 6은 9보다 작고, 2는 9보다 작고, 7은 9보다 작고.

(매번 나눌 때는 피벗을 고르면 되는데)  
다르게 거둬내면 이쁘게 정렬이 안됨!!

### • 퀵 정렬 프로그램

```
void quick_sort (int list[], int left, int right)
```

```
{
```

```
    int q;
```

```
    if (left < right) {
```

(퀵 정렬은 partition, 분할이 가장 중요!!!)

// 리스트에 2개 이상의 항목이 있으면

(q = partition(list, left, right)) // q = 분할 (partition)을 피벗 가장 왼쪽

```
        quick_sort(list, left, q-1); // 왼쪽 리스트 정렬 (left ~ q-1)
```

```
        quick_sort(list, q+1, right); // 오른쪽 리스트 정렬 (q+1 ~ right)
```

```
    }
```

```
}
```



• 분할 (partition) 알고리즘

1. low 가장 작은 원소 선택

2. high 가장 오른쪽 원소 + 1

3. pivot (pivot)이 가장 작은 원소. pivot의 index가 아닌 value이다!!

4. low를 1 증가시키고 (low < right 및 low < pivot)이면 이 과정 반복

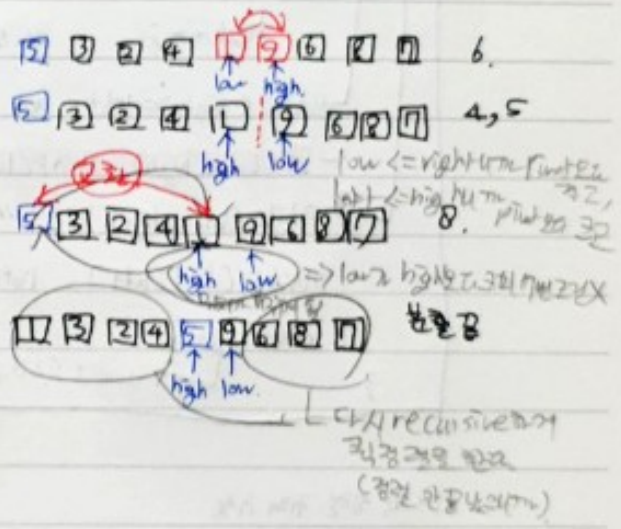
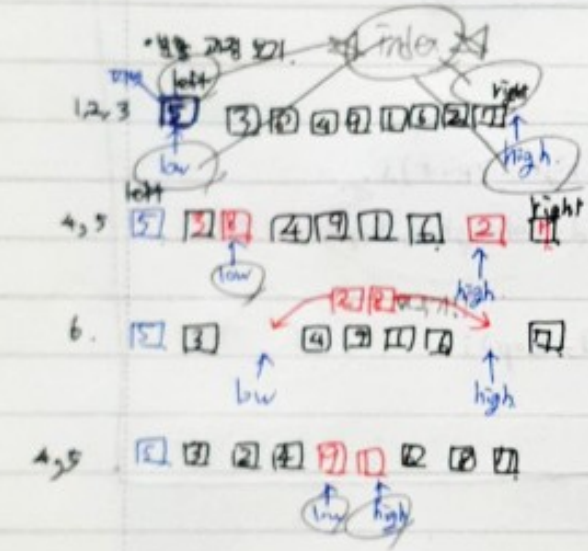
5. high를 1 감소시키고 만약 (left < high 및 high < pivot)이면 이 과정 반복

low < high이면 low와 high의 값을 교환

low < high이면 job 4

8. left (최소 원소)와 high가 등호로 같아지면

high가 작은 값이므로 left와 pivot과 교환해주고  
정확하게는 pivot이 중간에 위치!!



다시 recursive로  
작은 경우만  
(정렬 완료 상태)



• partition 함수

```
int partition (int list[], int left, int right) {
```

```
    int low, high, pivot, temp;
```

```
    low = left; // 1
```

```
    high = right + 1; // 2.
```

```
    pivot = list[left]; // 3
```

```
    do {
```

```
        do
```

```
            // 4
```

```
            low++;
```

```
        while (low <= right && list[low] < pivot);
```

```
        do
```

```
            high--;
```

```
            // 5
```

```
        while (high > left && list[high] > pivot);
```

```
        if (low < high) swap(list[low], list[high], temp); // 6.
```

```
    } while (low < high); // 7
```

```
    swap(list[left], list[high], temp);
```

```
    // 8.
```

```
    return high;
```

```
}
```

→ 9 번째 라인 (pivot이 정확히 맞춰져 들어갈 위치를 반환)

• 퀵 정렬 전체 과정



→ 9 번째 라인 (pivot이 정확히 맞춰져 들어갈 위치를 반환)

전체 원소를 다 이동시켜, 정렬된 배열이 반환된 것을 볼 수 있음.

pivot 위치에 놓은 원소 4, 정렬되지 않은 배열은 9개씩.



• 퀵 정렬 복잡도

□ 최선의 경우 (거의 균등한 리스트로 분할되는 경우)

□ 패스 4:  $\log(n)$

□ 각 패스 안에서 비교 횟수:  $n$

□ 총 비교 횟수:  $n \cdot \log n = O(n \log n)$

□ 총 이동 횟수: 비교 횟수에 비례하여 적으므로 무시 가능

□ 최악의 경우 (매우 불균등한 리스트로 분할되는 경우)

□ 패스 4:  $n$

□ 각 패스 안에서 비교 횟수:  $n$

□ 총 비교 횟수:  $n^2 = O(n^2)$

□ 총 이동 횟수: 비교 횟수에 비례하여 적으므로 무시 가능.

p. 486 ~ 487 12.10 정렬 알고리즘 비교.

• 정렬 알고리즘 비교

	알고리즘	최선	평균	최악
	선택 정렬	$O(n^2)$	$O(n^2)$	$O(n^2)$
	삽입 정렬	$O(n)$	$O(n^2)$	$O(n^2)$
	버블 정렬	$O(n^2)$	$O(n^2)$	$O(n^2)$
skip	수열 정렬	$O(n)$	$O(n^{1.5})$	$O(n^{1.5})$
	합병 정렬	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$
	퀵 정렬	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$
9장	히프 정렬	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$
skip	기타 정렬	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(\sqrt{n})$

• 정렬 알고리즘의 실행 시간 (정수 60,000개)

알고리즘	실행 시간 (단위: sec)
선택 정렬	10.842
삽입 정렬	17.438 data 전도시
버블 정렬	22.894
합병 정렬	0.026 data 많을시
퀵 정렬	0.014