Chapter 10.
정렬(Sort)

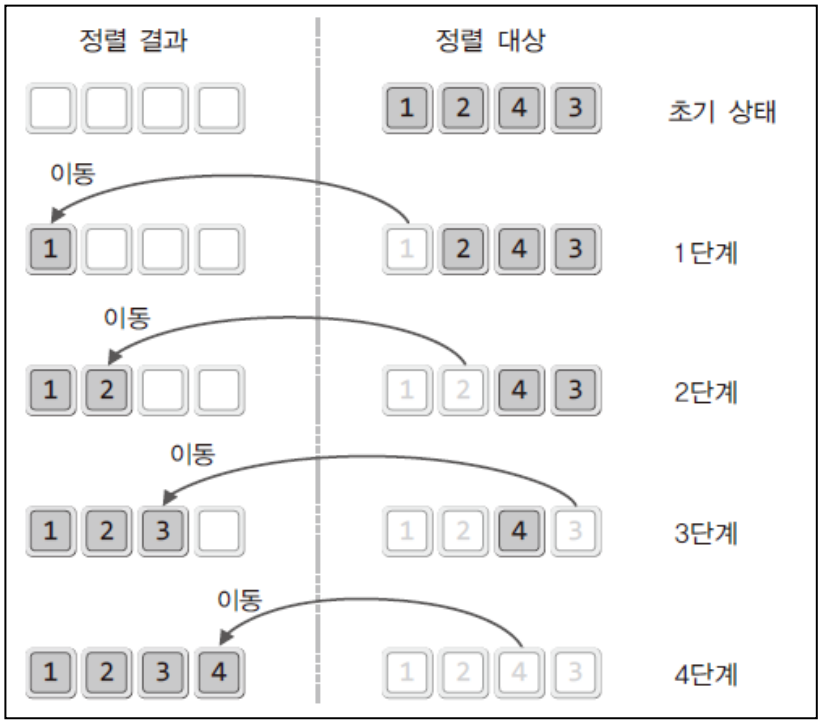
3주차

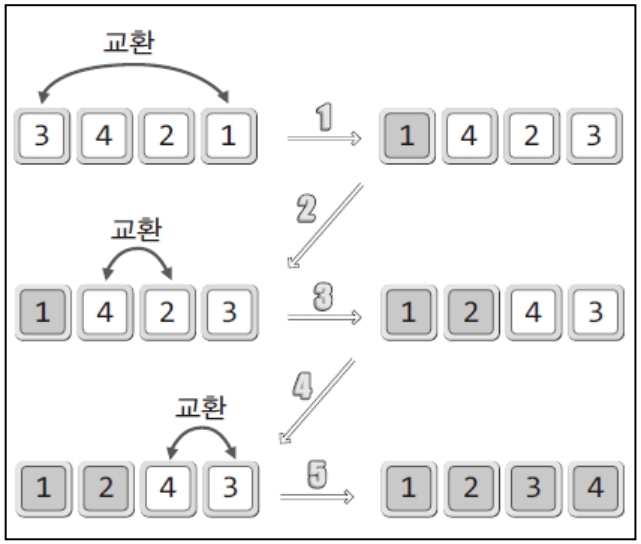
377-378p

버블 정렬
(성능평가)

- 비교의 횟수 : 두 데이터간의 **비교연산**의 횟수
  - $O(n^2)$
- 이동의 횟수 : 위치의 변경을 위한 데이터의 이동 횟수
  - 최악의 경우 비교의 횟수와 이동의 횟수는 일치한다.

선택 정렬
(이해)





별도의 메모리를 사용한 구현

별도의 메모리를 사용하지 않은 구현

379-380p

선택 정렬
(구현)

```
#include <stdio.h>

void SelSort(int arr[], int n) {
        int i, j;
        int maxIdx;
        int temp;

        for (i = 0; i < n - 1; i++) {
                maxIdx = i;
                for (j = i + 1; j < n; j++) {
                        if (arr[j] < arr[maxIdx])
                                maxIdx = j;
                }
                temp = arr[i];
                arr[i] = arr[maxIdx];
                arr[maxIdx] = temp;
        }
}
```

정렬 순서상 가장 앞서는 데이터의 index

최소값 탐색

교환

선택 정렬
(구현)

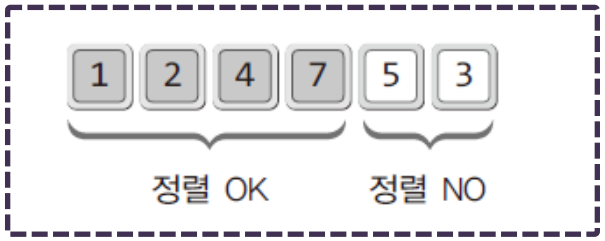
```
int main(void)
{
	int arr[4] = { 3, 4, 2, 1 };
	int i;

	SelSort(arr, sizeof(arr) / sizeof(int));

	for (i = 0; i < 4; i++)
		printf("%d ", arr[i]);

	printf("₩n");
	return 0;
}
```

선택 정렬
(성능평가)

- 비교의 횟수 : 두 데이터간의 비교연산의 횟수
  - $O(n^2)$

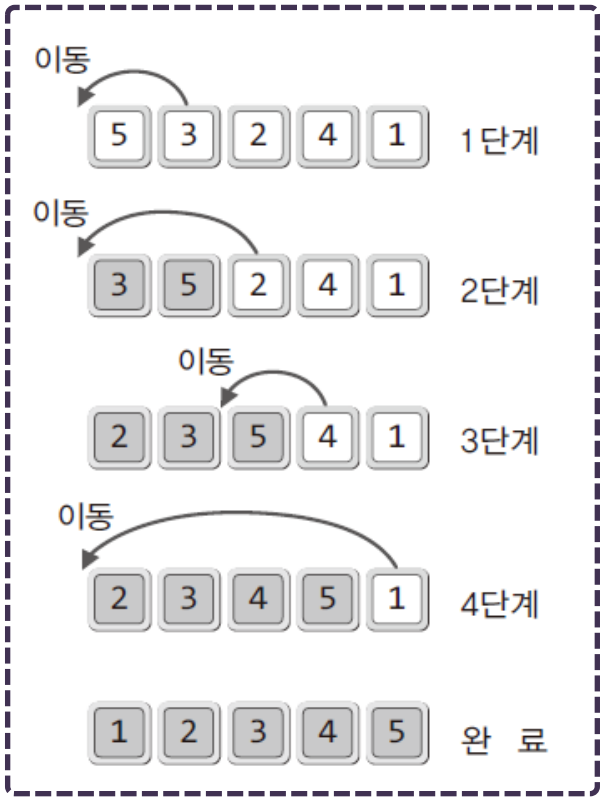
- 이동의 횟수 : 위치의 변경을 위한 데이터의 이동 횟수
  - 버블 정렬은 안쪽 for문에 속해 있지만 선택 정렬은 바깥쪽 for문에 속해 있다.
  - 최악의 경우와 최상의 경우 구분 없이 데이터 이동의 횟수는 동일하다!

삽입 정렬
(이해)

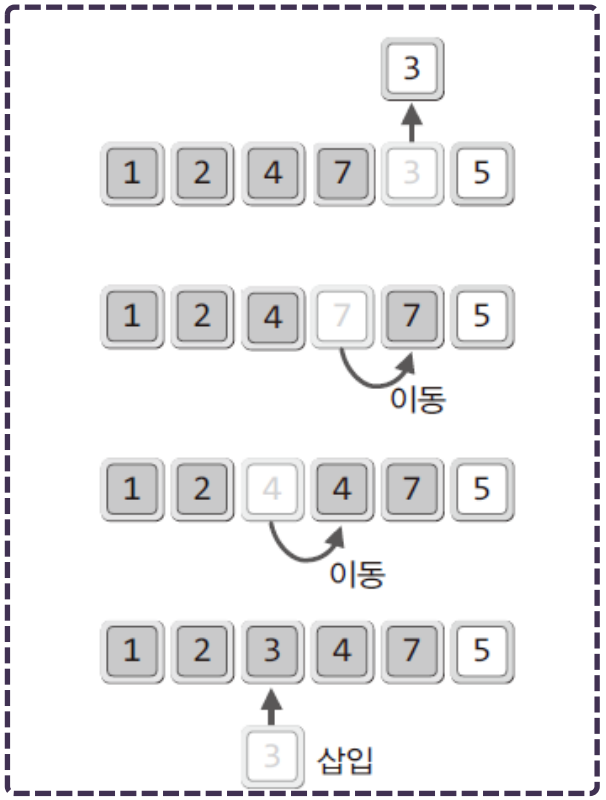


앞쪽에 정렬이 완료된 영역

뒤쪽에 그렇지 않은 영역이 있을 때 사용.



구현을 고려하면!



뒤로 밀어 내는 방법을 포함한 그림

383-384p

삽입 정렬
(구현)

```
#include <stdio.h>
void InserSort(int arr[], int n) {
    int i, j;
    int insData;

    for (i = 1; i < n; i++) {
        insData = arr[i];
        for (j = i - 1; j >= 0; j--) {
            if (arr[j] > insData)
                arr[j + 1] = arr[j];
            else
                break;
        }
        arr[j + 1] = insData;
    }
}
```

정렬 대상을 insData에 저장

비교 대상 한 칸 뒤로 밀기

삽입 위치 찾았으니 탈출!

찾은 위치에 정렬 대상 삽입!

삽입 정렬
(구현)

```
int main(void)
{
	int arr[5] = { 5, 3, 2, 4, 1 };
	int i;

	InserSort(arr, sizeof(arr) / sizeof(int));

	for (i = 0; i < 5; i++)
		printf("%d ", arr[i]);

	printf("₩n");
	return 0;
}
```

384-385p

삽입 정렬
(성능평가)

- 비교의 횟수 : 두 데이터간의 비교연산의 횟수
  - $O(n^2)$
- 이동의 횟수 : 위치의 변경을 위한 데이터의 이동횟수
  - $O(n^2)$

힙 정렬
(이해)

- 힙의 특성을 활용하여,
  힙에 정렬할 대상을 모두 넣었다가
  다시 꺼내어 정렬을 진행.

386-387p

힙 정렬
(구현 : 기존의 Heap 소스에 아래를 추가)

```
void HeapSort(int arr[], int n, PriorityComp pc)
{
        Heap heap;
        int i;

        HeapInit(&heap, pc);

        for (i = 0; i < n; i++)
                HInsert(&heap, arr[i]);

        for (i = 0; i < n; i++)
                arr[i] = HDelete(&heap);
}
```

정렬 대상을 가지고 힙을 구성한다.

순서대로 하나씩 꺼내서 정렬을 완성한다.

386-387p

힙 정렬
(구현 : 기존의 Heap 소스에 아래를 추가)

```
int PriComp(int n1, int n2) {
	return n2 - n1;
//	return n1 - n2;
}

int main(void) {
	int arr[4] = { 3, 4, 2, 1 };
	int i;

	HeapSort(arr, sizeof(arr) / sizeof(int), PriComp);

	for (i = 0; i < 4; i++)
		printf("%d ", arr[i]);

	printf("₩n");
	return 0;
}
```

힙 정렬
(성능평가)

- 하나의 데이터를 힙에 넣고 빼는 경우에 대한 시간 복잡도
  - 힙에 데이터 저장 시간 복잡도 : $O(log_2 n)$
  - 힙의 데이터 삭제 시간 복잡도 : $O(log_2 n)$
  - 위 둘을 합치면 : $O(2log_2 n) \rightarrow O(log_2 n)$
  - N 개의 데이터를 처리하면 : $O(nlog_2 n)$
  - n^2 과 $nlog_2 n$의 차이는 크다, (사용 가능과 사용 불가의 차이)

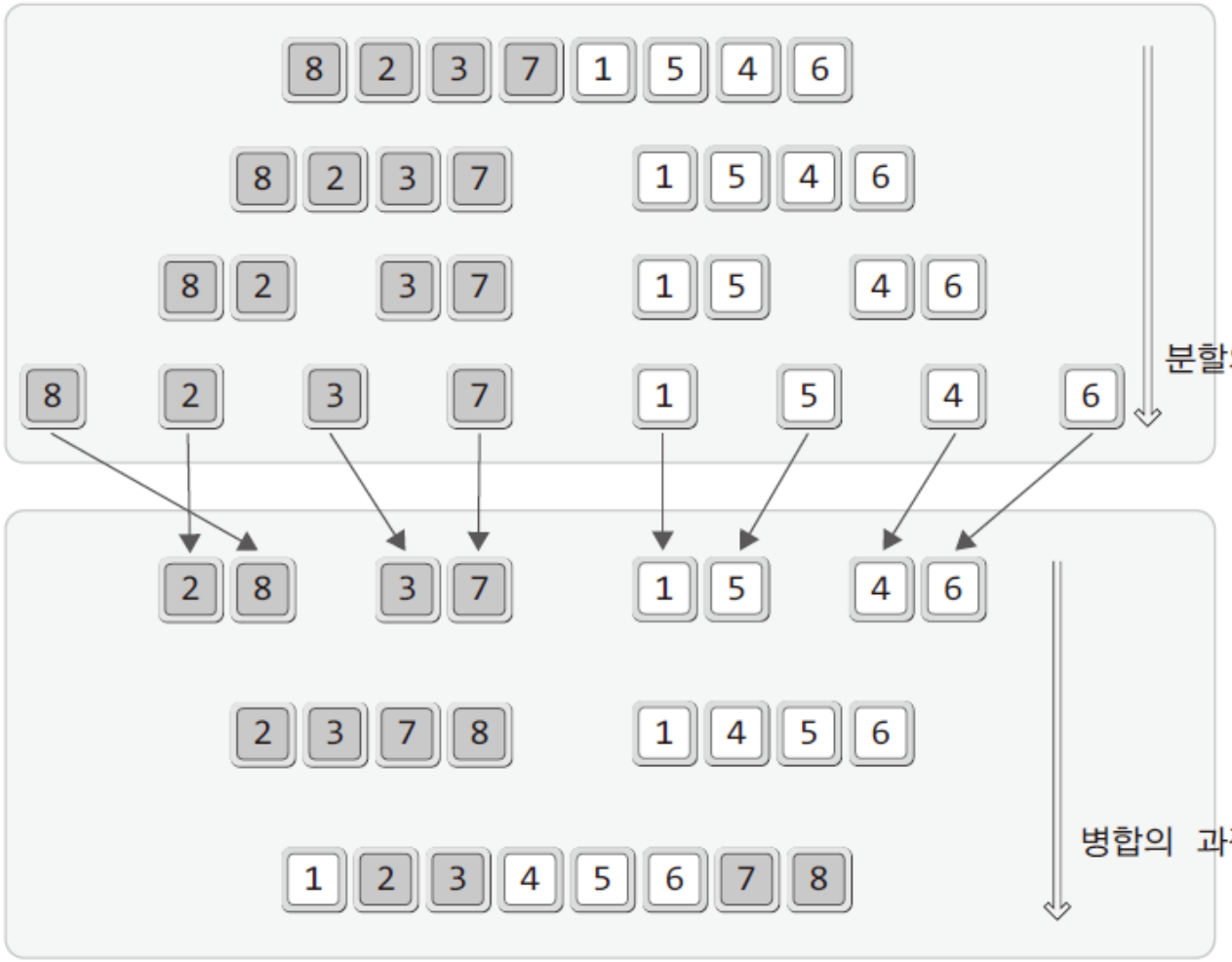
n	10	100	1,000	3,000	5,000
n^2	100	10,000	1,000,000	9,000,000	25,000,000
$n\log_2 n$	66	664	19,931	34,652	61,438

병합 정렬
(이해)

- 1단계 분할(Divide)
  - 해결이 용이한 단계까지 문제를 분할해 나간다.
- 2단계 정복(Conquer)
  - 해결이 용이한 수준까지 분할된 문제를 해결한다.
- 3단계 결합(Combine)
  - 분할해서 해결한 결과를 결합하여 마무리한다.

390-391p

병합 정렬
(이해)



재귀적으로
별도의 정렬이 필요 없을
때까지 분할!

분할보다 신경 써야
하는 것이 병합!

▶ [그림 10-10: 병합 정렬의 예]

394p

병합 정렬
(구현)

```
#include <stdio.h>
#include <stdlib.h>

void MergeSort(int arr[], int left, int right) {
        int mid;

        if (left < right)     {
                mid = (left + right) / 2;

                MergeSort(arr, left, mid);
                MergeSort(arr, mid + 1, right);

                MergeTwoArea(arr, left, mid, right);
        }
}
```

중간 지점 계산

둘로 나눠서 각각을 정렬

정렬된 두 배열을 병합

393p

병합 정렬 (구현)

```
void MergeTwoArea(int arr[], int left, int mid, int right) {
    int fIdx = left;
    int rIdx = mid + 1;
    int i;
    int * sortArr = (int*)malloc(sizeof(int)*(right + 1));
    int sIdx = left;
    while (fIdx <= mid && rIdx <= right) {
        if (arr[fIdx] <= arr[rIdx])
            sortArr[sIdx] = arr[fIdx++];
        else
            sortArr[sIdx] = arr[rIdx++];
        sIdx++;
    }
    if (fIdx > mid) {
        for (i = rIdx;i <= right;i++,sIdx++) sortArr[sIdx] = arr[i];
    }
    else {
        for (i = fIdx;i <= mid;i++,sIdx++) sortArr[sIdx] = arr[i];
    }
    for (i = left; i <= right; i++) arr[i] = sortArr[i];
    free(sortArr);
}
```

왼쪽배열 index와 오른쪽배열 index

sortArr : 결과를 담을 배열.
sIdx : sortArr의 index

왼쪽 또는 오른쪽 배열의 남은 원소들을 모두 결과 배열로 옮기기

병합 정렬
(구현)

```
int main(void)
{
	int arr[7] = { 3, 2, 4, 1, 7, 6, 5 };
	int i;

	MergeSort(arr, 0, sizeof(arr) / sizeof(int) - 1);

	for (i = 0; i < 7; i++)
		printf("%d ", arr[i]);

	printf("₩n");
	return 0;
}
```

배열 arr의 전체 영역 정렬

397-400p

병합 정렬
(성능평가)

- 비교 연산 횟수
  - MergeTwoArea 함수에서의 비교 연산 : $O(n)$
  - 단계의 수 : $O(\log_2 n)$
  - MergeTwoArea 함수를 각 단계마다 호출하므로 전체는… : $O(n\log_2 n)$

- 이동 연산 횟수
  - 각 단계마다…
    - 임시 배열에 데이터를 병합할 때 n회
    - 임시 배열에서 원위치로 옮길 때 n회
  - 단계의 수 : $O(\log_2 n)$
  - 전체는 $2n\log_2 n$ 이므로 $O(n\log_2 n)$

- 병합 정렬의 단점
  - 전체 데이터를 담을 임시 메모리 필요