1 선택 정렬

1 선택 정렬

1 정렬(Sorting)

- 데이터의 순서를 결정하는 것

- 데이터를 저장하는 위치에 따라
  - 내부 정렬(Internal Sort)과
  - 외부 정렬(External Sort)로 구분

1 선택 정렬

2 내부 정렬

- 데이터 양이 적을 때 주기억장치 내에 저장한 자료를 대상으로 정렬하는 방법
- 정렬할 자료의 양이 적어서 자료 전체가 주기억장치에 저장될 수 있는 경우에는 내부 정렬을 사용하여 자료를 정렬
- 선택 정렬, 버블 정렬, 삽입 정렬, 쉘 정렬, 퀵 정렬 등

1 선택 정렬

3 외부 정렬

- 입력의 크기가 주기억 장치 공간보다 큰 경우 보조 기억 장치에 있는 입력을 여러 번에 나누어 주기억 장치에 읽어 들인 후 정렬하여 보조 기억 장치에 다시 저장하는 과정을 반복

1 선택 정렬

4 정렬 알고리즘의 복잡도

- 대부분 $O(n^2)$과 $O(nlogn)$ 사이
- 입력이 특수한 성질을 만족하는 경우에는 $O(n)$ 정렬도 가능

1 선택 정렬

5 기본적인 정렬 알고리즘

- 평균적으로 $O(n^2)$의 시간이 소요되는 정렬 알고리즘들
  - 선택 정렬
  - 버블 정렬
  - 삽입 정렬

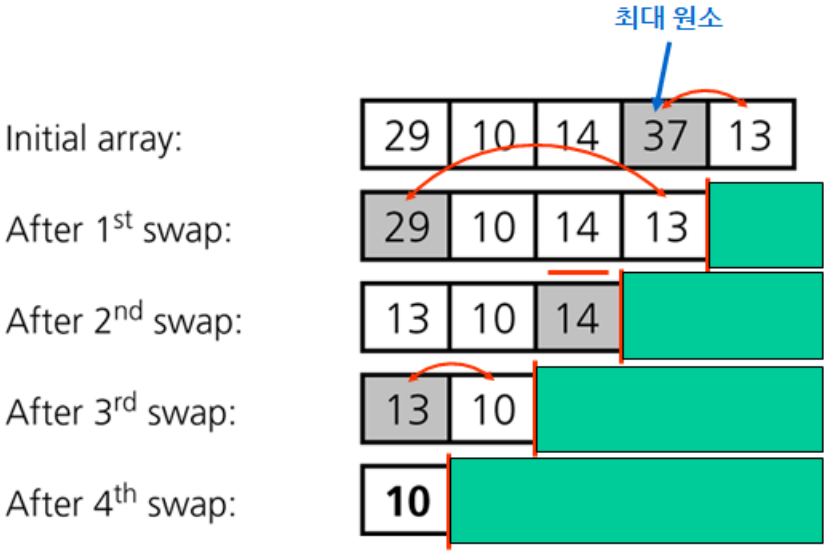
6 선택 정렬(Selection Sort)

- 실제로 많이 사용하는 내부 정렬에 속함
- 주어진 데이터에서 가장 큰 값이나 가장 작은 값을 찾은 후 그 값만을 교환하는 방식으로 단계적으로 정렬함

6 선택 정렬(Selection Sort)

- 각 루프마다
  - 최대 원소를 찾음
  - 최대 원소와 맨 오른쪽 원소를 교환함
  - 맨 오른쪽 원소를 제외함
- 하나의 원소만 남을 때까지 위의 루프를 반복

6 선택 정렬(Selection Sort)



- 수행 시간 : (n-1)+(n-2)+…+2+1 = $O(n^2)$ ← Worst case, Average case

※ 참고: $1+2+3+\cdots+n=\frac{n(n+1)}{2}$

6 선택 정렬(Selection Sort)

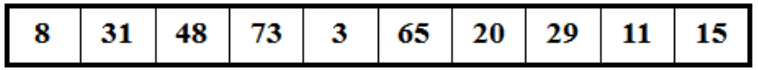
```
selectionSort(A[], n)      ▷ 배열 A[1 ... n]을 정렬한다
{
      for last ← n downto 2 {                          ---------- ①
            A[1 ... last] 중 가장 큰 수 A[k]를 찾는다;          ---------- ②
            A[k] ↔ A[last];  ▷ A[k]와 A[last]의 값을 교환   ---------- ③
      }
}
```

- 수행 시간 :
  ①의 for 루프는 n-1번 반복
  ②에서 가장 큰 수를 찾기 위한 비교횟수: n-1, n-2, …, 2, 1
  ③의 교환은 상수 시간 작업

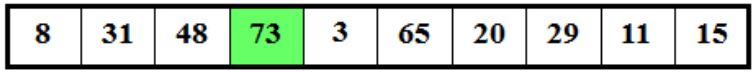
- 선택 정렬 복잡도
  $(n-1)+(n-2)+\cdots+2+1 = O(n^2)$

7 선택 정렬의 작동 예

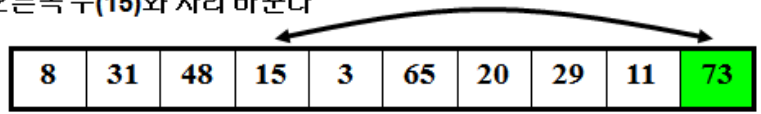
정렬할 배열이 주어짐



가장 큰 수를 찾는다(73)



73을 맨 오른쪽 수(15)와 자리 바꾼다



①의 첫번째 loop

1 선택 정렬

7 선택 정렬의 작동 예

맨 오른쪽 수를 제외한 나머지에서 가장 큰 수를 찾는다 **(65)**

8	31	48	15	3	65	20	29	11	73

65를 맨 오른쪽 수**(11)**와 자리 바꾼다

8	31	48	15	3	11	20	29	65	73

①의 두번째 **loop**

맨 오른쪽 두 수를 제외한 나머지에서 가장 큰 수를 찾는다 **(48)**

8	31	48	15	3	11	20	29	65	73

. . .

7 선택 정렬의 작동 예

. . .

8을 맨 오른쪽 수(3)와 자리 바꾼다

정렬이 완료된 최종 배열

3	8	11	15	20	29	31	48	65	73

8 선택 정렬의 특징

- 입력이 거의 정렬되어 있든지, 역순으로 정렬되어 있든지, 랜덤하게 되어 있든지를 구분하지 않고 항상 일정한 시간복잡도를 나타냄

- **입력에 민감하지 않은(Input Insensitive)** 알고리즘

2 버블 정렬

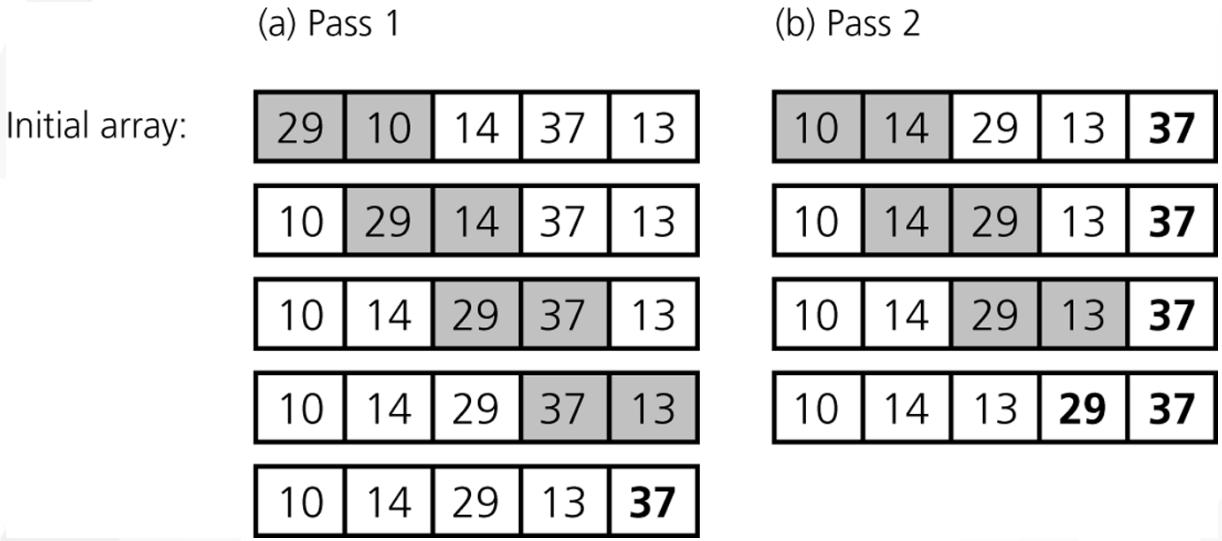
2 버블 정렬

1 버블 정렬(Bubble Sort)

- 이웃하는 숫자를 비교하여 작은 수를 앞쪽으로 이동시키는 과정을 반복하여 정렬하는 알고리즘
- 주어진 파일에서 인접한 2개의 레코드 키값을 비교하여 그 크기에 따라 레코드 위치를 서로 교환하는 정렬 방식
- 오름차순으로 정렬한다면 작은 수는 배열의 앞부분으로 이동

2 버블 정렬

1 버블 정렬(Bubble Sort)



- 수행 시간: (n-1)+(n-2)+…+2+1 = $O(n^2)$ ← Worst case, Average case

2 버블 정렬

1 버블 정렬(Bubble Sort)

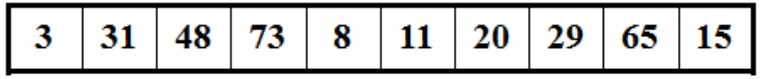
```
bubbleSort(A[], n)    ▷ A[1 ... n]을 정렬한다
{
      for last ← n downto 2                                  ---------- ①
            for i ← 1 to last-1                              ---------- ②
                  if (A[i] > A[i+1]) then A[i] ↔ A[i+1]; ▷ 원소 교환  ③
}
```

- 수행 시간:
  ①의 for 루프는 n-1번 반복
  ②의 for 루프는 각각 n-1, n-2, …, 2, 1번 반복
  ③은 상수 시간 작업

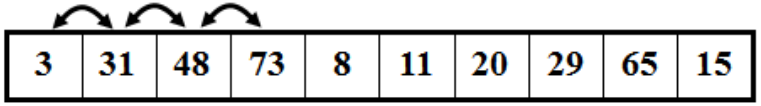
- 버블 정렬 복잡도
  $(n-1)+(n-2)+\cdots+2+1 = O(n^2)$

2 버블 정렬의 작동 예

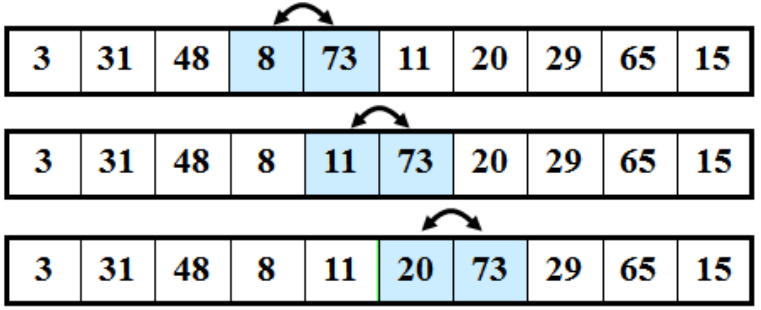
정렬할 배열이 주어진다



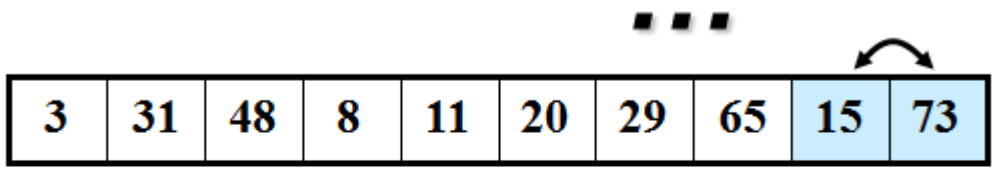
왼쪽부터 시작해 이웃한 쌍들을 비교해 나간다



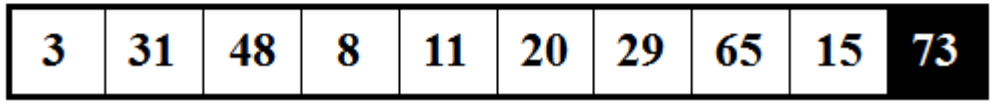
순서대로 되어 있지 않으면 자리 바꾼다



2 버블 정렬의 작동 예

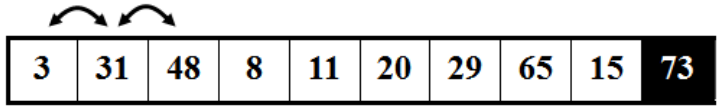


맨 오른쪽 수(73)를 대상에서 제외한다

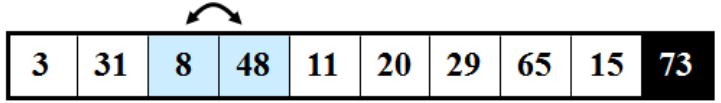


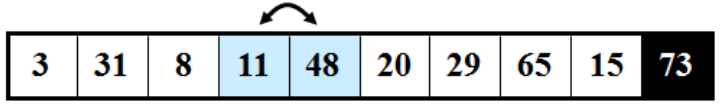
2 버블 정렬의 작동 예

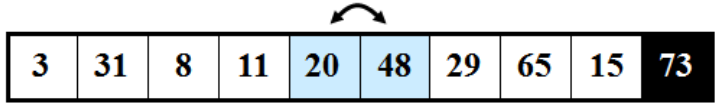
왼쪽부터 시작해 이웃한 쌍들을 비교해간다

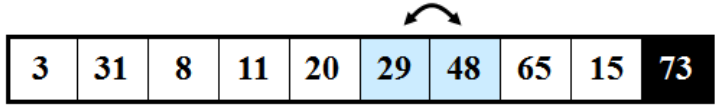


순서대로 되어 있지 않은 경우에는 자리 바꾼다



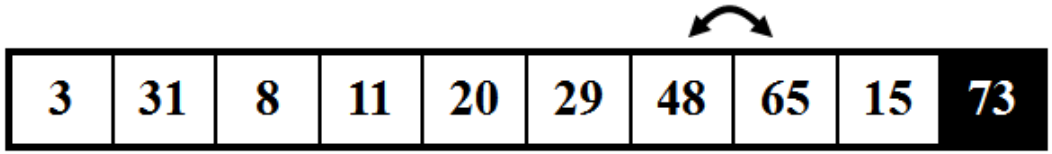


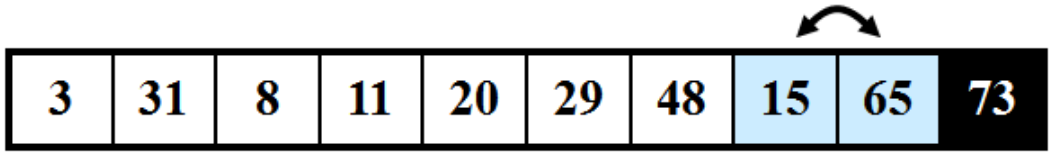




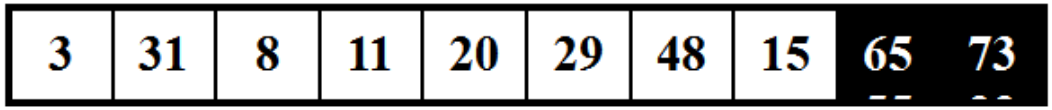
2 버블 정렬

2 버블 정렬의 작동 예





맨 오른쪽 수(**65**)를 대상에서 제외한다

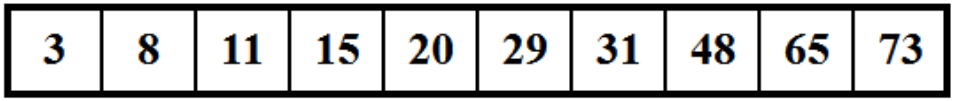


2 버블 정렬의 작동 예

앞의 작업을 반복하면서 계속 제외해 나간다

...

두개짜리 배열의 처리를 끝으로 정렬이 완료된다



3 삽입 정렬

3 삽입 정렬

1 삽입 정렬(Insertion Sort)

- 새로운 데이터를 정렬된 데이터에 삽입하는 과정을 반복하여 전체 데이터를 정렬하는 방식
- 가장 간단한 정렬 방식
- 이미 순서화된 파일에 새로운 하나의 레코드를 순서에 맞게 삽입시켜 정렬하는 방식
- 어느 정도 정렬이 되어 있을 경우 매우 효과적

3 삽입 정렬

1 삽입 정렬(Insertion Sort)

◆ 삽입 정렬 방식

① 처음 A[0]은 정렬된 데이터로 취급

② 다음 데이터 A[1]은 정렬된 데이터 A[0]과 비교하여 적절한 위치에 삽입

③ 다음 데이터 A[2]는 정렬된 데이터 A[0], A[1]과 비교하여 적절한 위치에 삽입

④ 같은 방식으로 나머지 데이터들을 삽입하여 정렬

3 삽입 정렬

2 삽입 정렬의 작동 예

3 삽입 정렬

3 삽입 정렬의 복잡도

```
insertionSort(A[], n)      ▷ A[1 ... n]을 정렬한다
{
      for i ← 2 to n                                    ---------- ①
            A[1 ... i]의 적당한 자리에 A[i]를 삽입한다;      ----------②
}
```

- 수행 시간:
  ①의 for 루프는 n-1번 반복
  ②의 삽입은 최악의 경우 i-1회 비교

- 삽입 정렬 복잡도
  Worst case : 1+2+…+(n-2)+(n-1) = $O(n^2)$
  Average case : ½ (1+2+…+(n-2)+(n-1)) = $O(n^2)$
  Best case : 1+1+…+1+1 = $O(n)$
  (1+1+…+1+1 : n-1)

3 삽입 정렬

4 삽입 정렬의 특징

- **입력의 상태에 따라 수행 시간이 달라질 수 있음**
- 입력이 이미 정렬되어 있으면 항상 각각 현재 원소가 자신의 왼쪽 원소와 비교 후 자리이동 없이 원래 자리에 있게 됨
- 따라서 (n-1)번의 비교만 하면 정렬을 마치게 됨
- 이때가 삽입 정렬의 최선 경우이고 시간복잡도는 O(n)

3 삽입 정렬

4 삽입 정렬의 특징

- 삽입 정렬은 거의 정렬된 입력에 대해서
  다른 정렬 알고리즘보다 빠름

- 반면에 **역으로(반대로) 정렬된 입력**에 대해서는 $O(n^2)$
  시간이 걸림

- 삽입 정렬의 평균 경우 시간복잡도는 최악 경우와 같음