정렬

13 주차-강의

남춘성

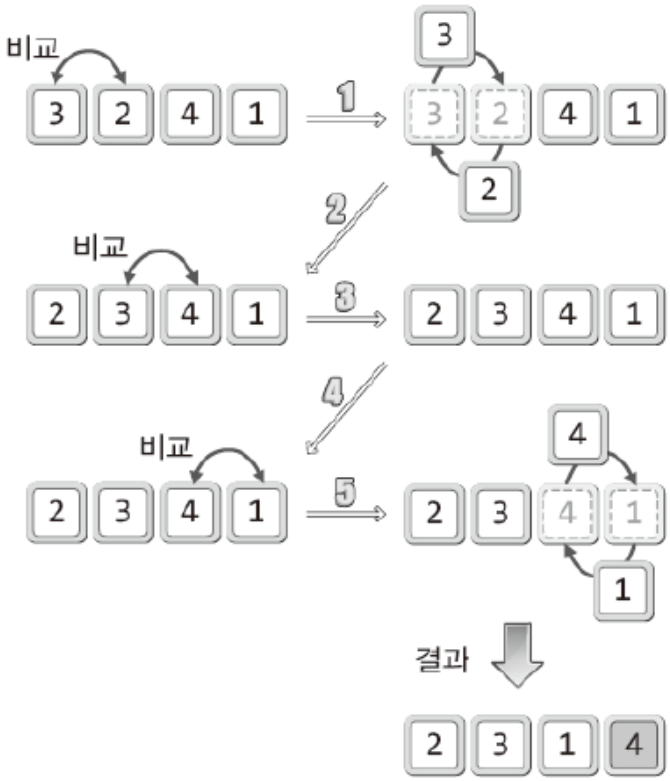
SUNG KYUN KWAN UNIVERSITY

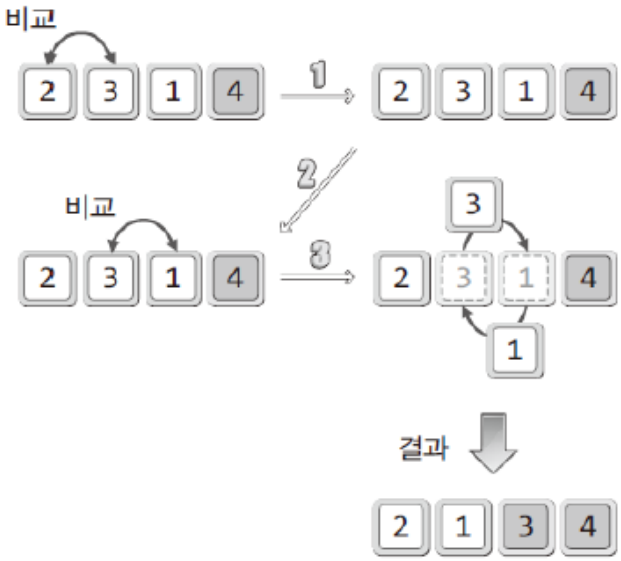
버블 정렬의 개념적 이해

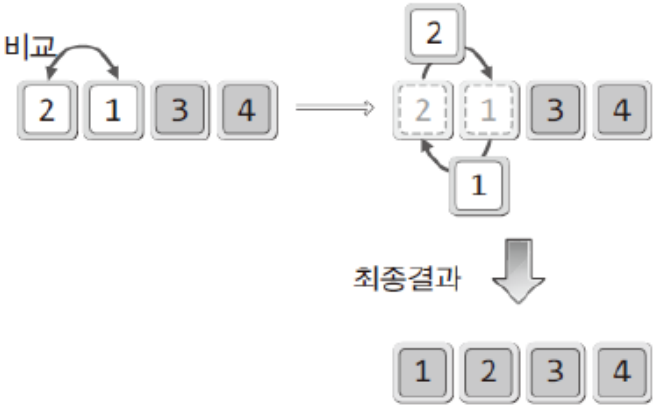
SUNG KYUN KWAN UNIVERSITY

- **버블 정렬(bubble sort)**
  - 인접한 두 개의 원소를 비교하여 자리를 교환하는 방식
    - 첫 번째 원소부터 마지막 원소까지 반복하여 한 단계가 끝나면 가장 큰 원소가 마지막 자리로 정렬
    - 첫 번째 원소부터 인접한 원소끼리 계속 자리를 교환하면서 맨 마지막 자리로 이동하는 모습이 방울 모양과 같다고 하여 버블(bubble) 정렬

버블정렬: 예를 통한 이해







인접한 두 개의 데이터를 비교해가면서 정렬을 진행하는 방식

버블정렬 : 구현

```
void BubbleSorT(int arr[], int n) {
        int i, j;
        int temp;

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

버블정렬의 실질적 구현코드

버블 정렬 알고리즘 분석

SUNG KYUN KWAN UNIVERSITY

- 메모리 사용공간
  - n개의 원소에 대하여 n개의 메모리 사용

- 연산 시간
  - 최선의 경우 : 자료가 이미 정렬되어있는 경우
    - 비교횟수 : i번째 원소를 (n-i)번 비교하므로, n(n-1)/2 번
    - 자리교환횟수(이동) : 자리교환이 발생하지 않음
  - 최악의 경우 : 자료가 역순으로 정렬되어있는 경우
    - 비교횟수 : i번째 원소를 (n-i)번 비교하므로, n(n-1)/2 번
    - 자리교환횟수(이동) : i번째 원소를 (n-i)번 교환하므로, n(n-1)/2 번
  - 평균 시간 복잡도는 $O(n^2)$이 된다.

비교횟수

$$(n-1) + (n-2) + \cdots + 2 + 1$$

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2-n}{2}$$

$$O(n^2)$$

선택 정렬의 개념적 이해

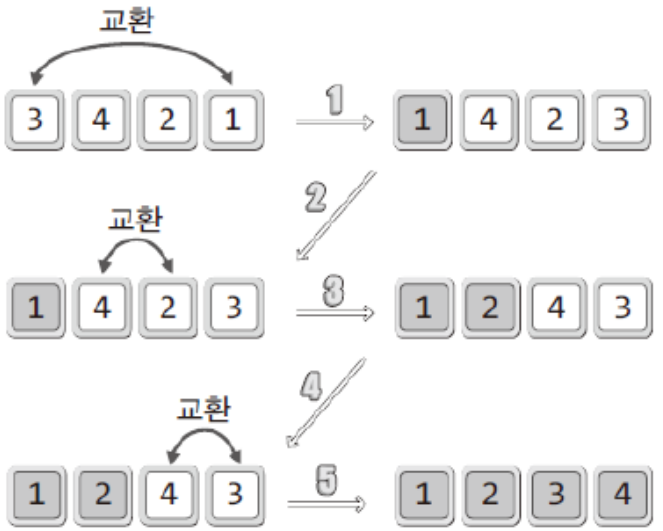
SUNG KYUN KWAN UNIVERSITY

- 전체 원소들 중에서 기준 위치에 맞는 원소를 선택하여 자리를 교환하는 방식으로 정렬

- 수행 방법
  - 전체 원소 중에서 가장 작은 원소를 찾아 선택하여 첫 번째 원소와 자리를 교환
  - 그 다음 두 번째로 작은 원소를 찾아 선택하여 두 번째 원소와 자리를 교환
  - 그 다음에는 세 번째로 작은 원소를 찾아 선택하여 세 번째 원소와 자리를 교환
  - 이 과정을 반복하면서 정렬을 완성

선택 정렬 : 예를 통한 이해

하나씩 선택해서 정렬 결과를 완성
별도의 메모리 공간이 요구됨



정렬순서상 가장 앞선 것을 왼쪽으로 이동,
원래 그 자리에 있던 데이터는 빈자리에 놓음
: 이방법도 교환!!

선택 정렬 : 구현

SUNG KYUN KWAN UNIVERSITY

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

        for (i = 0; i < n - 1; i++) {
                maxIdx = i;                              //교환할 idx

                for (j = i + 1; j < n; j++) {
                        if (arr[j] < arr[maxIdx])        //비교연산
                                maxIdx = j;              // 교환할 idx 값 갱신
                }

                // 교환 – 이동 연산
                temp = arr[i];
                arr[i] = arr[maxIdx];
                arr[maxIdx] = temp;
        }
}
```

선택 정렬 알고리즘 분석

SUNG KYUN KWAN UNIVERSITY

- 메모리 사용공간
  - n개의 원소에 대하여 n개의 메모리 사용

- 비교횟수
  - 1단계 : 첫 번째 원소를 기준으로 n개의 원소 비교
    2단계 : 두 번째 원소를 기준으로 마지막 원소까지 n-1개의 원소 비교
    3단계 : 세 번째 원소를 기준으로 마지막 원소까지 n-2개의 원소 비교
  - i 단계 : i 번째 원소를 기준으로 n-i개의 원소 비교

$$\text{전체 비교횟수} = \sum_{i=1}^{n-1} n-i = \frac{n(n-1)}{2}$$

  - 어떤 경우에서나 비교횟수가 같으므로 시간 복잡도는 **$O(n^2)$**이 된다.

최악의 경우와 최상의 경우 구분 없이 데이터 이동 횟수가 동일함

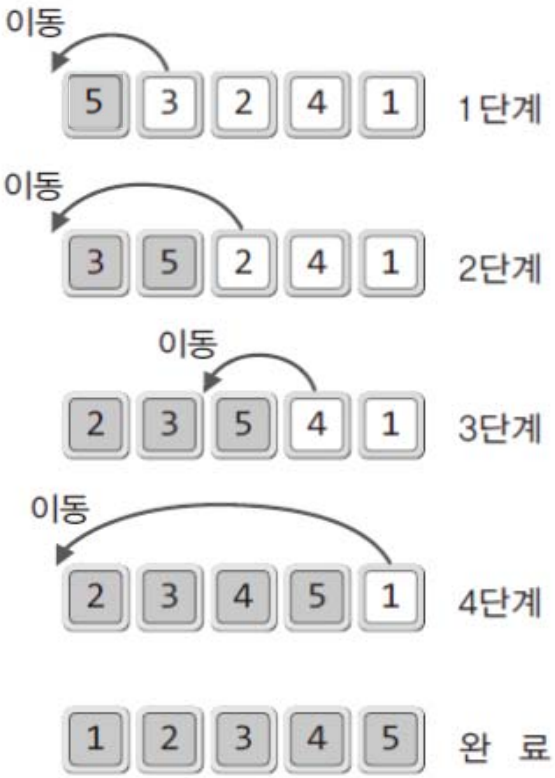
삽입 정렬의 개념적 이해

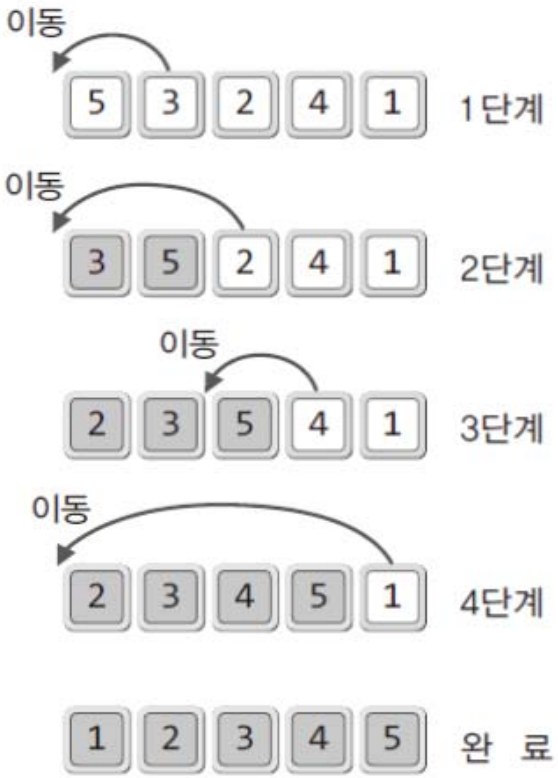
SUNG KYUN KWAN UNIVERSITY

- 정렬되어있는 부분집합에 정렬할 새로운 원소의 위치를 찾아 삽입하는 방법
- 정렬할 자료를 두 개의 부분집합 S와 U로 가정
  - 부분집합 S: 정렬된 앞부분의 원소들
  - 부분집합 U: 아직 정렬되지 않은 나머지 원소들
    - 정렬되지 않은 부분집합 U의 원소를 하나씩 꺼내서 이미 정렬되어있는 부분집합 S의 마지막 원소부터 비교하면서 위치를 찾아 삽입
    - 삽입 정렬을 반복하면서 부분집합 S의 원소는 하나씩 늘리고 부분집합 U의 원소는 하나씩 감소
    - 부분집합 U가 공집합이 되면 삽입 정렬이 완성
- 선택 정렬보다 두 배 정도 빨라서 평균적인 성능이 $O(n^2)$ 알고리즘들 중에서 뛰어난 축으로, 다른 정렬 알고리즘의 일부로도 자주 사용
- 대입이 많고, 데이터의 상태, 데이터 한 개의 크기에 따라 성능 편차가 심함

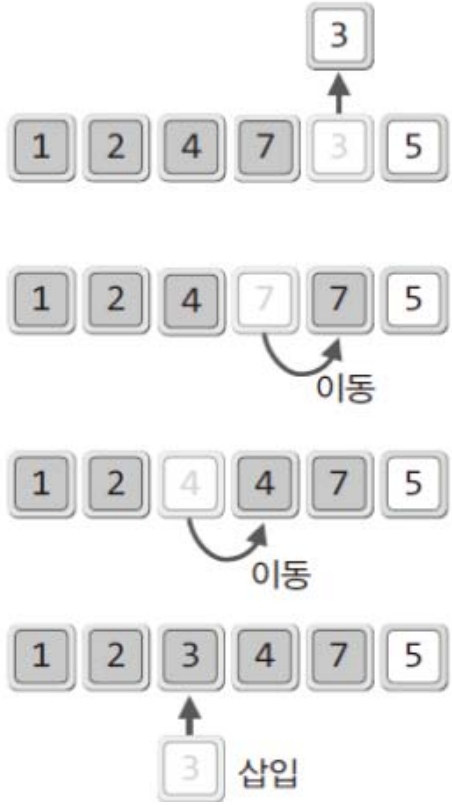
삽입 정렬 : 예를 통한 이해

정렬이 완료된 곳과 그렇지 않은 곳
구분 방법





뒤로 밀어 내는 방법을 포함한 그림



삽입 정렬 : 구현

SUNG KYUN KWAN UNIVERSITY

```
void InsertSort(int arr[], int n) {
	int i, j;
	int insData;

	for (i = 1; i < n; i++) {
		insData = arr[i];			//정렬 대상 저장

		for (j = i – 1; j >= 0; j--) {
			if (arr[j]>insData)		//데이터간 비교연산
				arr[j + 1] = arr[j];	//비교대상 뒤로 한칸 밀기, 데이터간 이동연산
			else
				break;
		}

		arr[j + 1] = insData;

	}
}
```

삽입 정렬 알고리즘 분석

SUNG KYUN KWAN UNIVERSITY

- 메모리 사용공간
  - n개의 원소에 대하여 n개의 메모리 사용

- 연산 시간
  - 최선의 경우 : 원소들이 이미 정렬되어있어서 비교횟수가 최소인 경우
    - 이미 정렬되어있는 경우에는 바로 앞자리 원소와 한번만 비교한다.
    - 전체 비교횟수 = n-1
    - 시간 복잡도 : O(n)
  - 최악의 경우 : 모든 원소가 역순으로 되어있어서 비교횟수가 최대인 경우
    - 전체 비교횟수 = $1+2+3+\cdots+(n-1) = n(n-1)/2$
    - 시간 복잡도 : $O(n^2)$
  - 삽입 정렬의 평균 비교횟수 = $n(n-1)/4$
  - 평균 시간 복잡도 : $O(n^2)$

힙 정렬의 개념적 이해

SUNG KYUN KWAN UNIVERSITY

- 힙의 특성
  - 히프 자료구조를 이용한 정렬 방법
  - 히프에서는 항상 가장 큰 원소가 루트 노드가 되고 삭제 연산을 수행하면 항상 루트 노드의 원소를 삭제하여 반환
    - 최대 히프에 대해서 원소의 개수만큼 삭제 연산을 수행하여 내림차순으로 정렬 수행
    - 최소 히프에 대해서 원소의 개수만큼 삭제 연산을 수행하여 오름차순으로 정렬 수행
  - 히프 정렬 수행 방법

(1) 정렬할 원소들을 입력하여 최대 히프 구성

(2) 히프에 대해서 삭제 연산을 수행하여 얻은 원소를 마지막 자리에 배치

(3) 나머지 원소에 대해서 다시 최대 히프로 재구성

원소의 개수만큼 (2)~(3) 을 반복 수행

힙 정렬 : 구현(이전코드로 형식만)

SUNG KYUN KWAN UNIVERSITY

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

	//정렬대상을 가지고 힙을 구성함
	for (i = 0; i < n; i++)			//연산시간
		HInsert(&heap, arr[i]);

	//순서대로 하나씩 꺼내서 정렬을 완성
	for (i = 0; i < n; i++)			//연산시간
		arr[i] = HDelete(&heap);
}
```

힙 알고리즘 분석

SUNG KYUN KWAN UNIVERSITY

- 메모리 사용공간
  - 원소 n개에 대해서 n개의 메모리 공간 사용
  - 크기 n의 히프 저장 공간
- 연산 시간
  - 히프 재구성 연산 시간
    - n개의 노드에 대해서 완전 이진 트리는 $\log_2(n+1)$의 레벨을 가지므로 완전 이진 트리를 히프로 구성하는 평균시간은 $O(\log_2 n)$
    - n개의 노드에 대해서 n번의 히프 재구성 작업 수행
  - 평균 시간 복잡도 : $O(n \log_2 n)$

시간복잡도
- 힙의 데이터 저장 시간 복잡도 : $O(\log_2 n)$
- 힙의 데이터 삭제 시간 복잡도 : $O(\log_2 n)$

힙 구성 : $O(2\log_2 n)$ → $O(\log_2 n)$ → n개의 데이터 $O(n\log_2 n)$

병합 정렬의 개념적 이해 - I

SUNG KYUN KWAN UNIVERSITY

- 여러 개의 정렬된 자료의 집합을 병합하여 한 개의 정렬된 집합을 만드는 방법
- 부분집합으로 분할(divide)하고, 각 부분집합에 대해서 정렬 작업을 완성(conquer)한 후에 정렬된 부분집합들을 다시 결합(combine)하는 분할 정복(divide and conquer) 기법 사용
- 병합 정렬 방법의 종류
  - 2-way 병합 : 2개의 정렬된 자료의 집합을 결합하여 하나의 집합으로 만드는 병합 방법
  - n-way 병합 : n개의 정렬된 자료의 집합을 결합하여 하나의 집합으로 만드는 병합 방법
- 병합 정렬의 큰 결점은 데이터 전체 크기만한 메모리가 더 필요
- 시간이 데이터 상태에 별 영향을 받지 않고, 시간 복잡도가 O(n log n)인 알고리즘 중에 유일하게 안정적

병합 정렬의 개념적 이해 - II

SUNG KYUN KWAN UNIVERSITY

- 2-way 병합 정렬 : 세 가지 기본 작업을 반복 수행하면서 완성.

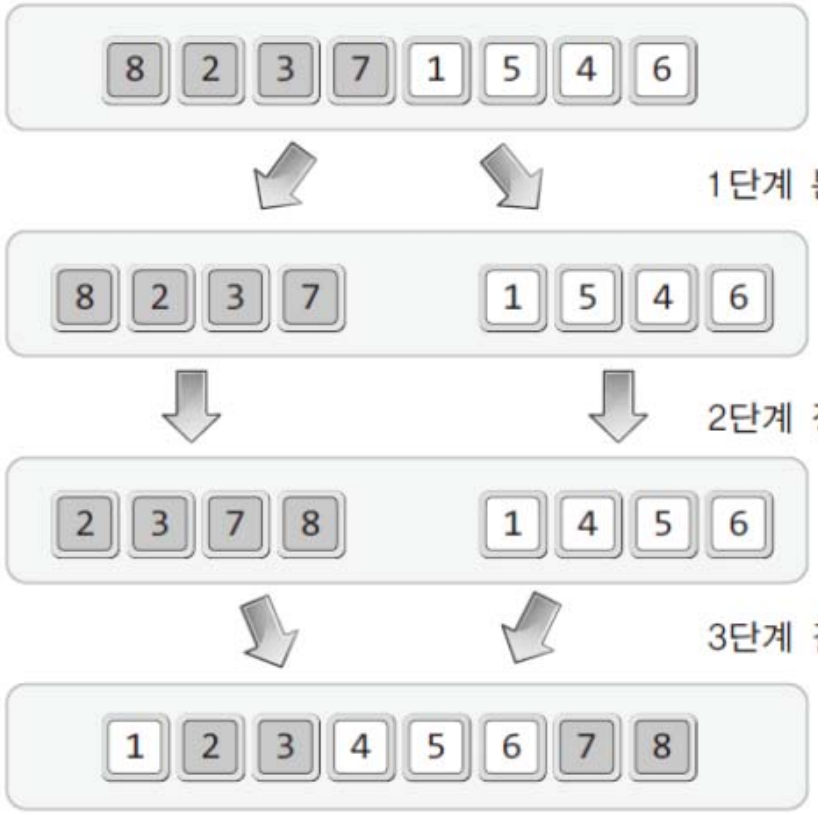
  (1) 분할(divide) : 입력 자료를 같은 크기의 부분집합 2개로 분할한다.

  (2) 정복(conquer) : 부분집합의 원소들을 정렬한다.

  부분집합의 크기가 충분히 작지 않으면 순환호출을 이용하여 다시 분할 정복 기법을 적용한다.

  (3) 결합(combine) : 정렬된 부분집합들을 하나의 집합으로 결합한다.

병합 정렬 : 예를 통한 이해 - I

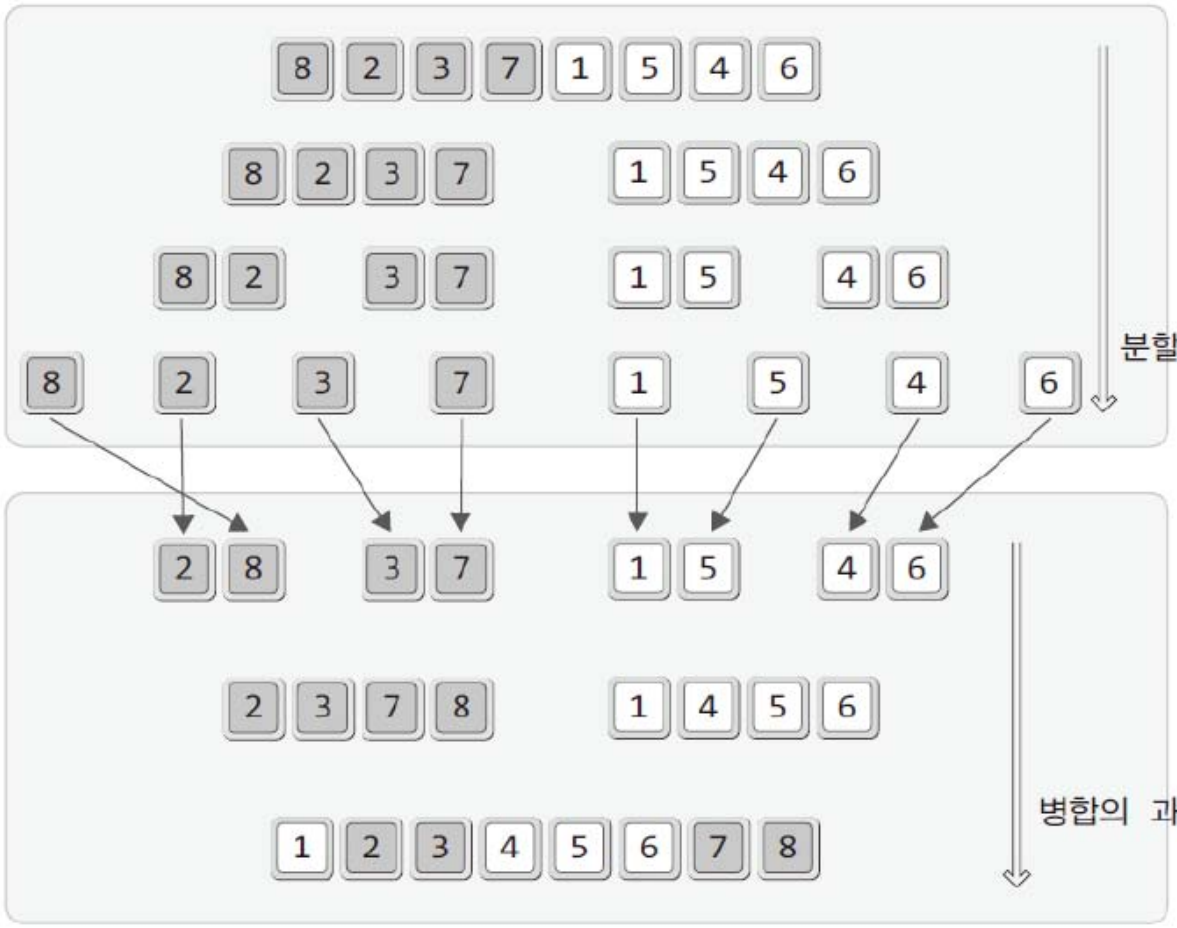


정렬하기 좋은 상태로 분할을 진행해 감

정렬하기 좋은 상태에서 정렬을 진행함

정렬이 완료된 조각들을 결합하여 정렬을 끝냄

병합 정렬 : 예를 통한 이해 - II



분할의 과정 : 재귀적

별도의 정렬을 진행하지 않아도 될 수준까지 분할을 진행함

병합 정렬 : 구현 - I

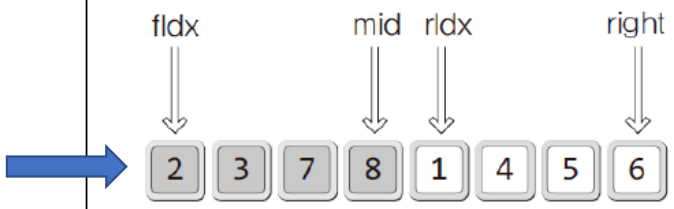
SUNG KYUN KWAN UNIVERSITY

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

        if (left < right) {                          // left가 작다는 것은 더 더 나뉠 수 있다는 의미
                mid = (left + right) / 2;            //중간지점을 계산

                MergeSort(arr, left, mid);           // left~mid에 위치한 데이터 정렬!
                MergeSort(arr, mid + 1, right);      // mid+1 ~ right에 위치한 데이터 정렬!

                MergeTwoArea(arr, left, mid, right); // 정렬된 두 배열 병합
        }
}
```

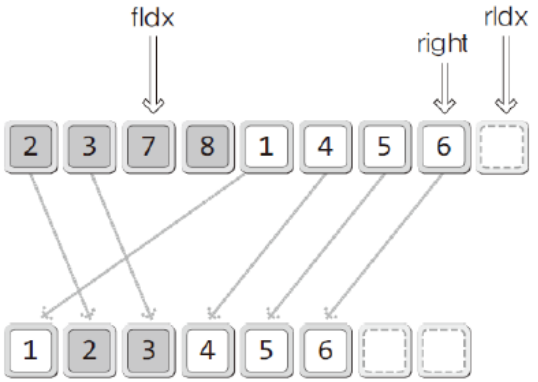
병합 정렬 : 구현 - II

SUNG KYUN KWAN UNIVERSITY

```
void MergeTwoArea(int arr[], int left, int mid, int right) {
        int fIdx = left; Int rIdx = mid + 1; int sIdx = left; int i;
        int * sortArr = (int*)malloc(sizeof(int)*(right + 1));  //병합 결과를 담을 공간 할당
        //비교연산
        while (fIdx <= mid && rIdx <= right) {                  // 두 영역의 데이터를 비교하여 sortArr에 담음
                if (arr[fIdx] <= arr[rIdx])
                        sortArr[sIdx] = arr[fIdx++];
                else
                        sortArr[sIdx] = arr[rIdx++];
                sIdx++;
        }

        if (fIdx > mid) {                                       //배열 앞부분 모두 이동된 후 뒷부분만 sortArr에 이동
                for (i = rIdx; i <= right; i++, sIdx++)
                        sortArr[sIdx] = arr[i];
        }
        else {                                                  //배열 뒷부분 모두 이동된 후 앞부부만 sortArr에 이동
                for (i = fIdx; i <= mid;  i++, sIdx++)
                        sortArr[sIdx] = arr[i];
        }
        //비교연산(원래위치로)
        for (i = left; i <= right; i++)                         //결과를 원래 배열에 옮겨 담는다
                arr[i] = sortArr[i];
        free(sortArr);
}
```



1차 병합의 결과



병합 정렬 알고리즘 분석

SUNG KYUN KWAN UNIVERSITY

- 메모리 사용공간
  - 각 단계에서 새로 병합하여 만든 부분집합을 저장할 공간이 추가로 필요
  - 원소 n개에 대해서 (2 x n)개의 메모리 공간 사용

- 연산 시간
  - 분할 단계 : n개의 원소를 분할하기 위해서 $\log_2 n$번의 단계 수행
  - 병합 단계 : 부분집합의 원소를 비교하면서 병합하는 단계에서 최대 n번의 비교 연산 수행
  - 전체 병합 정렬의 시간 복잡도 : $O(n \log_2 n)$

퀵 정렬의 개념적 이해 - I

SUNG KYUN KWAN UNIVERSITY

- 정렬할 전체 원소에 대해서 정렬을 수행하지 않고, 기준 값을 중심으로 왼쪽 부분 집합과 오른쪽 부분 집합으로 분할하여 정렬하는 방법
  - 왼쪽 부분 집합에는 기준 값보다 작은 원소들을 이동시키고, 오른쪽 부분 집합에는 기준 값보다 큰 원소들을 이동시킨다.
  - 기준 값 : 피봇(pivot)
    - 일반적으로 전체 원소 중에서 가운데에 위치한 원소를 선택

- 퀵 정렬은 다음의 두 가지 기본 작업을 반복 수행하여 완성
  - 분할(divide)
    - 정렬할 자료들을 기준 값을 중심으로 2개의 부분 집합으로 분할
  - 정복(conquer)
    - 부분 집합의 원소들 중에서 기준 값보다 작은 원소들은 왼쪽 부분 집합으로, 기준 값보다 큰 원소들은 오른쪽 부분집합으로 정렬
    - 부분 집합의 크기가 1 이하로 충분히 작지 않으면 순환호출을 이용하여 다시 분할

퀵 정렬의 개념적 이해 - II

SUNG KYUN KWAN UNIVERSITY

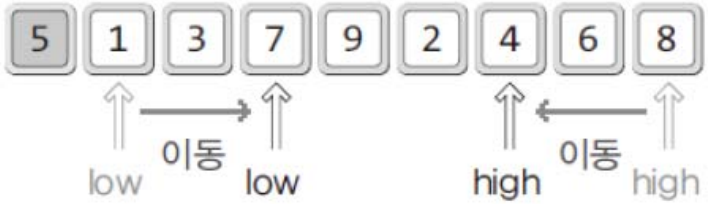
- 부분 집합으로 분할하기 위해서 L과 R을 사용
  - ① 왼쪽 끝에서 오른쪽으로 움직이면서 크기를 비교하여 피봇보다 크거나 같은 원소를 찾아 L(low)로 표시
  - ② 오른쪽 끝에서 왼쪽으로 움직이면서 피봇보다 작은 원소를 찾아 R(high)로 표시
  - ③ L이 가리키는 원소와 R이 가리키는 원소를 서로 교환한다.

- L와 R이 만나게 되면 피봇과 R의 원소를 서로 교환하고, 교환한 위치를 피봇의 위치로 확정
- 피봇의 확정된 위치를 기준으로 만들어진 새로운 왼쪽 부분 집합과 오른쪽 부분 집합에 대해서 퀵 정렬을 순환적으로 반복 수행
- 모든 부분 집합의 크기가 1 이하가 되면 퀵 정렬을 종료

퀵 정렬 : 예를 통한 이해 - I

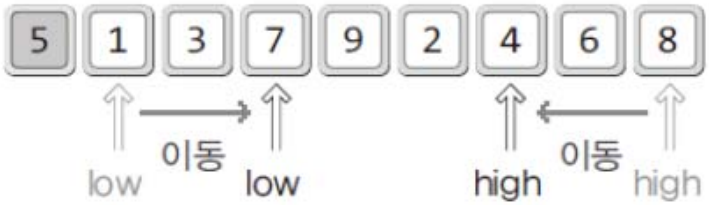
5개의 변수 정의

Left : 정렬대상의 가장 왼쪽
Right : 정렬대상의 가장 오른쪽
Pivot : 중심축
Low : 피벗을 제외한 가장 왼쪽에 위치한 지점
High : 피벗을 제외한 가장 오른쪽에 위치한 지점

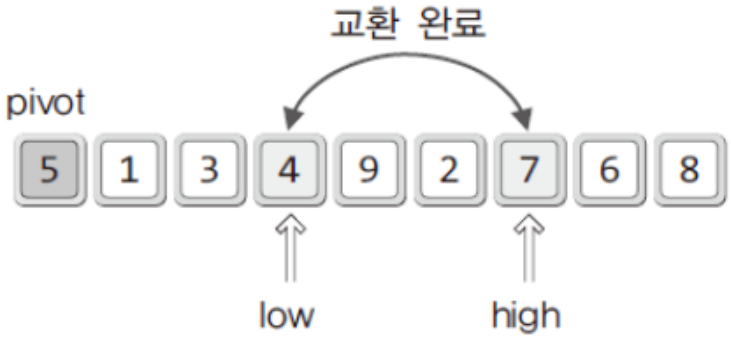


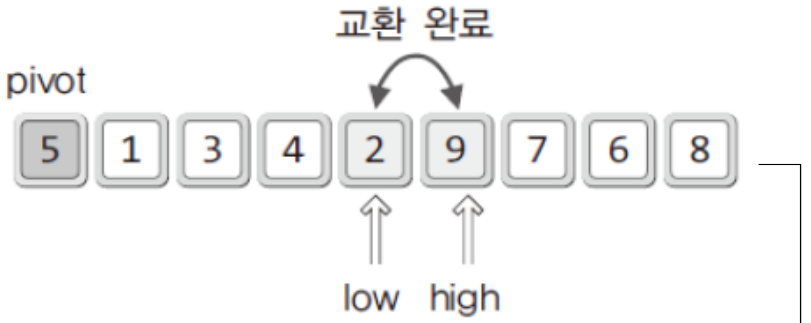
Low의 오른쪽 방향 이동 : 피벗보다 큰 값
High의 왼쪽 방향 이동 : 피벗보다 작은 값

퀵 정렬 : 예를 통한 이해 - II

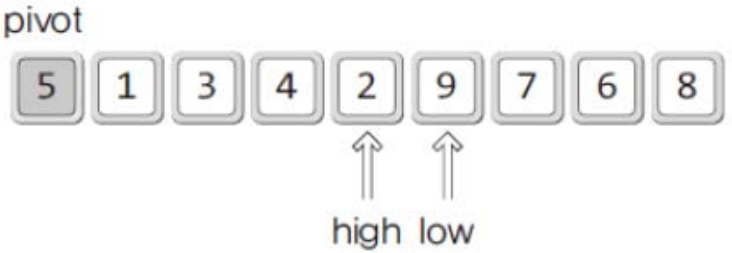


Low의 오른쪽 방향 이동 : 피벗보다 큰 값
High의 왼쪽 방향 이동 : 피벗보다 작은 값
Low와 high 데이터 교환



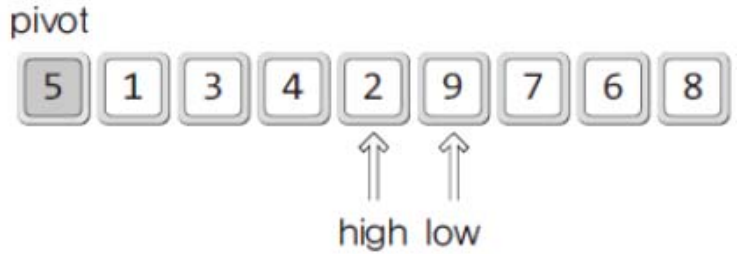


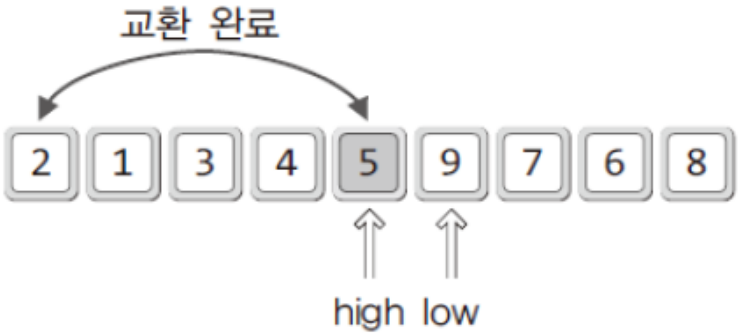
교환 후 이동을 계속하다 같은 조건으로 만나면 다시 교환



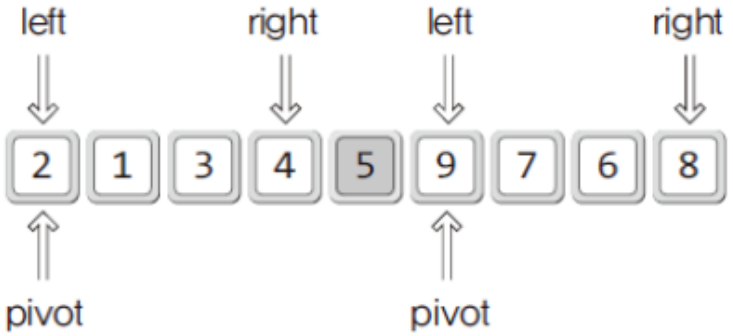
High와 low가 역전될 때까지 이동을 계속 함

퀵 정렬 : 예를 통한 이해 - III





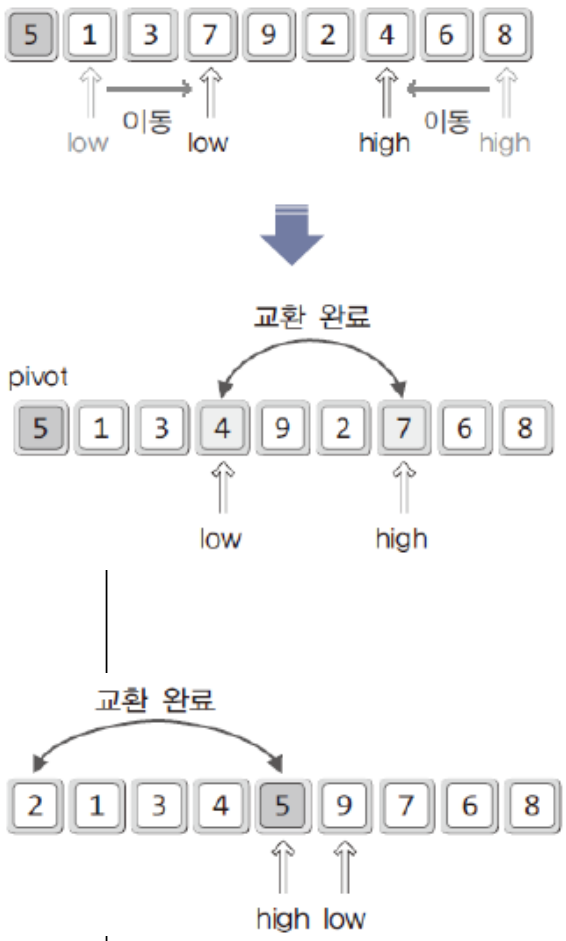
High와 low가 역전되면, 피벗과 high의 데이터 교환



두 개의 영역(피벗을 중심으로) 나누어 반복 실행함

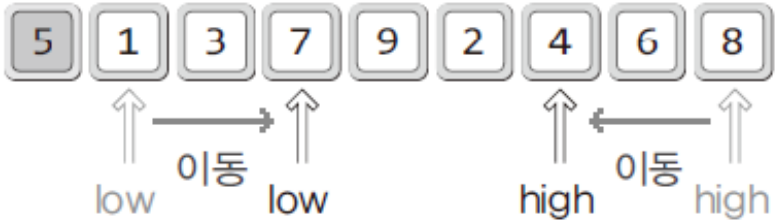
퀵 정렬 : 구현

```
int Partition(int arr[], int left, int right) {
        int pivot = arr[left];//피벗의 위치는 가장 왼쪽으로 설정
        int low = left + 1;
        int high = right;

        while (low <= high) {//교차되지 않을 때까지 반복
                while (pivot >= arr[low] && low <= right) //위치가 같은 경우 증감을 위해
                                                          //정렬 범위 넘지 않기 위해
                //while (pivot > arr[low])//피벗보다 큰 값을 찾는 과정
                        low++;
                while (pivot <= arr[high] && high >= (left+1))
                //while (pivot < arr[high])//피벗보다 작은 값을 찾는 과정
                        high--;

                if (low <= high) //교차되지 않은 상태라면 Swap 실행
                        Swap(arr, low, high);
        }

        Swap(arr, left, high);//피벗과 high가 가리키는 대상 교환

        return high;//옮겨진 피벗의 위치정보 반환
}
```



퀵 정렬 : 구현 (피벗 선택)

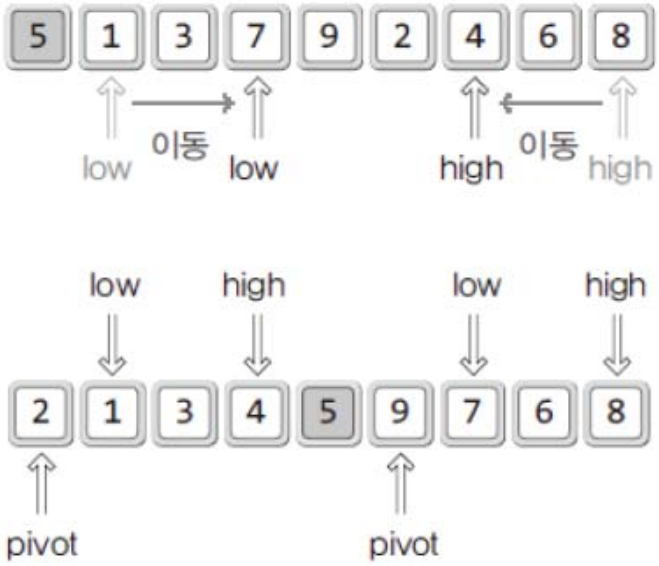
피벗이 정렬 대상의 한쪽 끝에 치우치는 경우 최악의 경우



정렬 과정에서 선택되는 피벗의 수가 적을수록 최상의 경우

데이터 불규칙하면 피벗이 중간에 해당하는 값에 가깝게 선택될 수 록 최상의 경우

퀵 정렬 : 성능평가(최선의 경우)



모든 데이터가 피벗과의 데이터 비교, 즉, 약 N번의 비교연산진행

중간에 있더라도 마찬가지로 약 N번의 비교연산이 진행

데이터 수 N에 대해 나뉘는 횟수를 K라고 하면 $K=\log_2 n$ 이고 비교연산 횟수 n을 곱하여 $O(n\log_2 n)$

퀵 정렬 : 성능평가(최악의 경우)

둘로 나뉘는 횟수가 약 n
매 단계별 비교 연산의 횟수 약 n

중간에 가까운 값으로 O()를 선택하는 노력을 조금만 하더라도 퀵 정령은 최악의 경우를 만들지 않는다. 따라서 퀵 정렬의 성능은 최상의 경우를 근거로 이야기 하는 경우가 맞음

퀵 정렬은 $O(n\log_2 n)$ 의 성능을 보이는 정렬 알고리즘 중에서 평균적으로 가장 좋은 성능을 보이는 알고리즘임. 또한 다른 알고리즘에 비해 데이터의 이동이 적고 별도의 메모리 공간을 요구하지 않음.

기수 정렬의 개념적 이해 - I

SUNG KYUN KWAN UNIVERSITY

- 원소의 키값을 나타내는 기수를 이용한 정렬 방법
  - 정렬할 원소의 키 값에 해당하는 버킷(bucket)에 원소를 분배하였다가 버킷의 순서대로 원소를 꺼내는 방법을 반복하면서 정렬
    - **원소의 키를 표현하는 기수만큼의 버킷 사용**
    - **예) 10진수로 표현된 키 값을 가진 원소들을 정렬할 때에는 0부터 9까지 10개의 버킷 사용**
  - 키 값의 자리수 만큼 기수 정렬을 반복
    - 키 값의 일의 자리에 대해서 기수 정렬을 수행하고,
    - 다음 단계에서는 키 값의 십의 자리에 대해서,
    - 그리고 그 다음 단계에서는 백의 자리에 대해서 기수 정렬 수행
  - 한 단계가 끝날 때마다 버킷에 분배된 원소들을 버킷의 순서대로 꺼내서 다음 단계의 기수 정렬을 수행해야 하므로 큐를 사용하여 버킷을 만든다.

기수 정렬의 개념적 이해 - II

SUNG KYUN KWAN UNIVERSITY

- 기수 정렬의 특징
  - 정렬순서의 앞서고 뒤섬을 비교하지 않음
  - 정렬 알고리즘의 한계로 알려진 $O(nlog_2n)$ 을 뛰어넘을 수 있음
  - 적용할 수 있는 대상이 매우 제한적. (길이가 동일한 데이터들만 정렬이 용이)

"배열에 저장된 1, 7, 9, 5, 2, 6을 오름차순으로 정렬하라!"

기수 정렬 OK!

"영단어 red, why, zoo, box를 사전편찬 순서대로 정렬하여라!"

기수 정렬 OK!

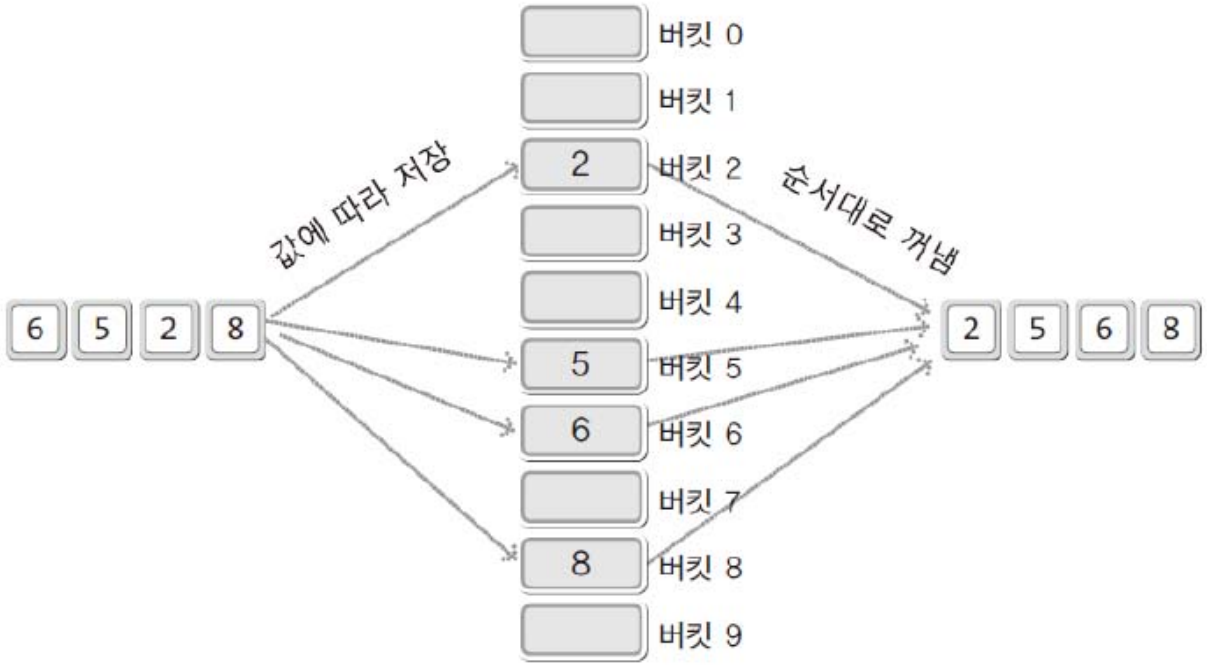
"배열에 저장된 21, -9, 125, 8, -136, 45를 오름차순으로 정렬하라!"

기수 정렬 NO!

"영단어 professionalism, few, hydroxyproline, simple을 사전편찬 순서대로 정렬하여라!"

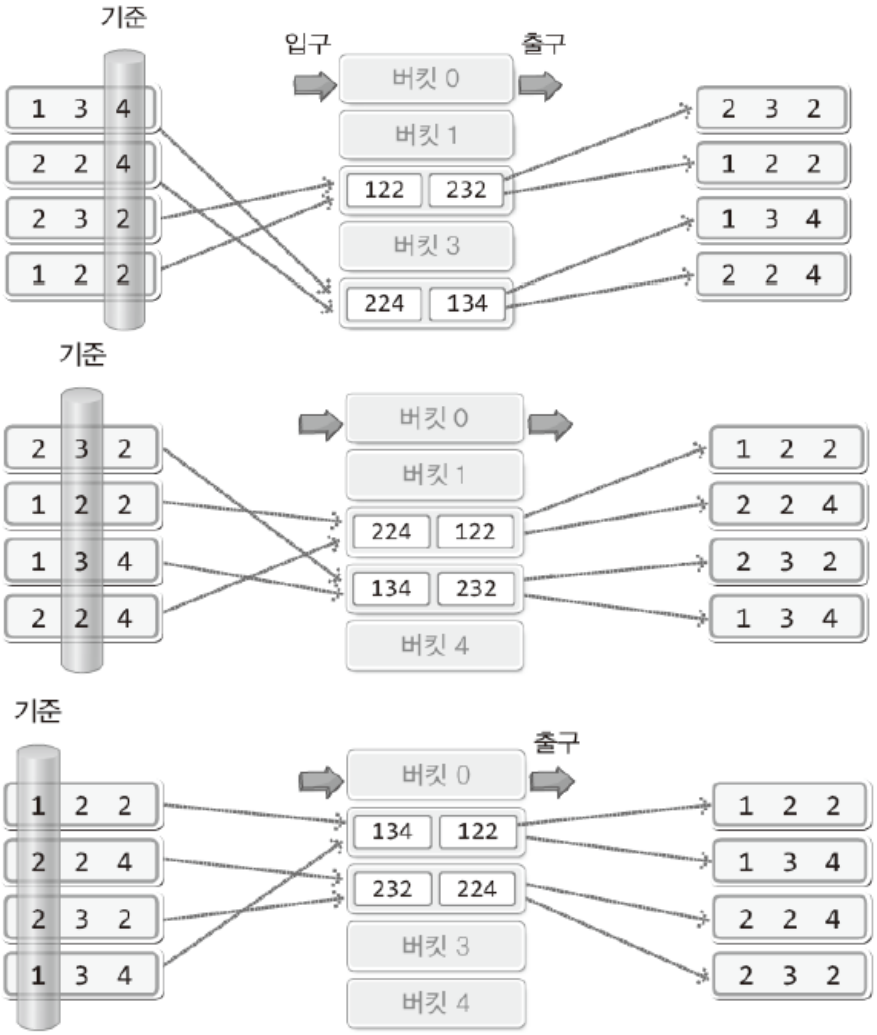
기수 정렬 NO!

기수 정렬 : 예를 통한 이해 - I



기수(radix) : 주어진 데이터를 구성하는 기본 요소(기호)
버킷(bucket) : 기수의 수에 해당하는 만큼의 버킷을 활용

기수 정렬 : 예를 통한 이해 – II (Least Significant Digit)



1의 자리 정렬

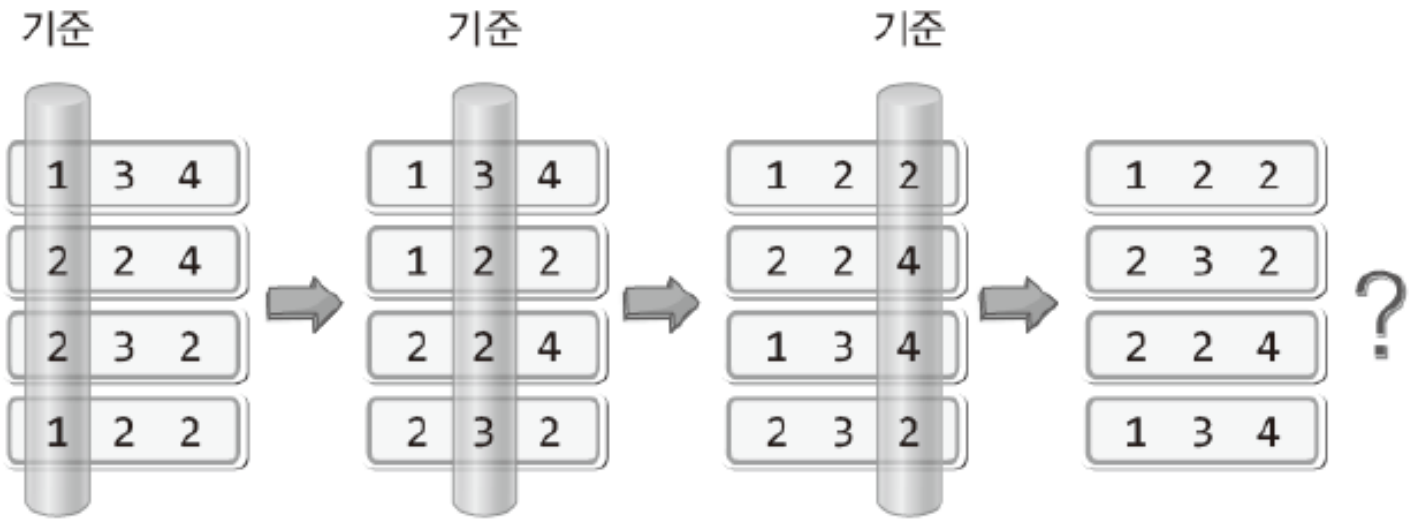
10의 자리 정렬

100의 자리 정렬

기수 정렬 : 예를 통한 이해 – III (Most Significant Digit)

SUNG KYUN KWAN UNIVERSITY

MSD : 정렬 기준 선정 방향이 LSD와 반대임. LSD와 같이 마지막에 정렬이 완성되는 것이 아닌 중간에 정렬이 완료된 데이터는 더 이상 정렬 과정을 진행하지 않아야 함.



기수 정렬 : 구현 (LSD) - I

SUNG KYUN KWAN UNIVERSITY

- LSD의 구현이 더 용이
  - MSD와 LSD의 빅-오는 같음

  - Num로부터 추출 방법(3자리인 경우)
    - Num으로 부터 첫 번째 자리 추출 : Num / 1%10
    - Num으로 부터 두 번째 자리 추출 : Num / 10%10
    - Num으로 부터 세 번째 자리 추출 : Num / 100%10

기수 정렬 : 구현 (LSD) - II

SUNG KYUN KWAN UNIVERSITY

```
void RadixSort(int arr[], int num, int maxLen) {
        Queue buckets[BUCKET_NUM];              // 매개변수 maxLen에는 정렬대상 중 가장 긴 데이터의 길이 정보 전달
        int bi; int pos; int di; int divfac = 1; int radix;

        for (bi = 0; bi < BUCKET_NUM; bi++)     //10개의 버킷 초기화
                QueueInit(&buckets[bi]);

        for (pos = 0; pos < maxLen; pos++){     //가장 긴 데이터의 길이만큼 반복 (데이터의 길이만큼 연산과정(I)만큼)
                for (di = 0; di < num; di++){           //삽입 : 연산과정(n만큼)
                        radix = (arr[di] / divfac) % 10;        //n번째 자리의 숫자 추출
                        Enqueue(&buckets[radix], arr[di]);      //추출한 숫자를 근거로 버킷에 데이터 저장
                }

                for (bi = 0, di = 0; bi < BUCKET_NUM; bi++) {   //버킷의 수 만큼 반복
                        while (!QIsEmpty(&buckets[bi]))         //버킷에 저장된 것 순서대로 다 꺼내서 다시 arr에 저장
                                arr[di++] = Dequeue(&buckets[bi]);
                }

                divfac *= 10;//n번째 자리의 숫자 추출을 위한 피제수 증가
        }
}
```

O(ln)