# 정렬

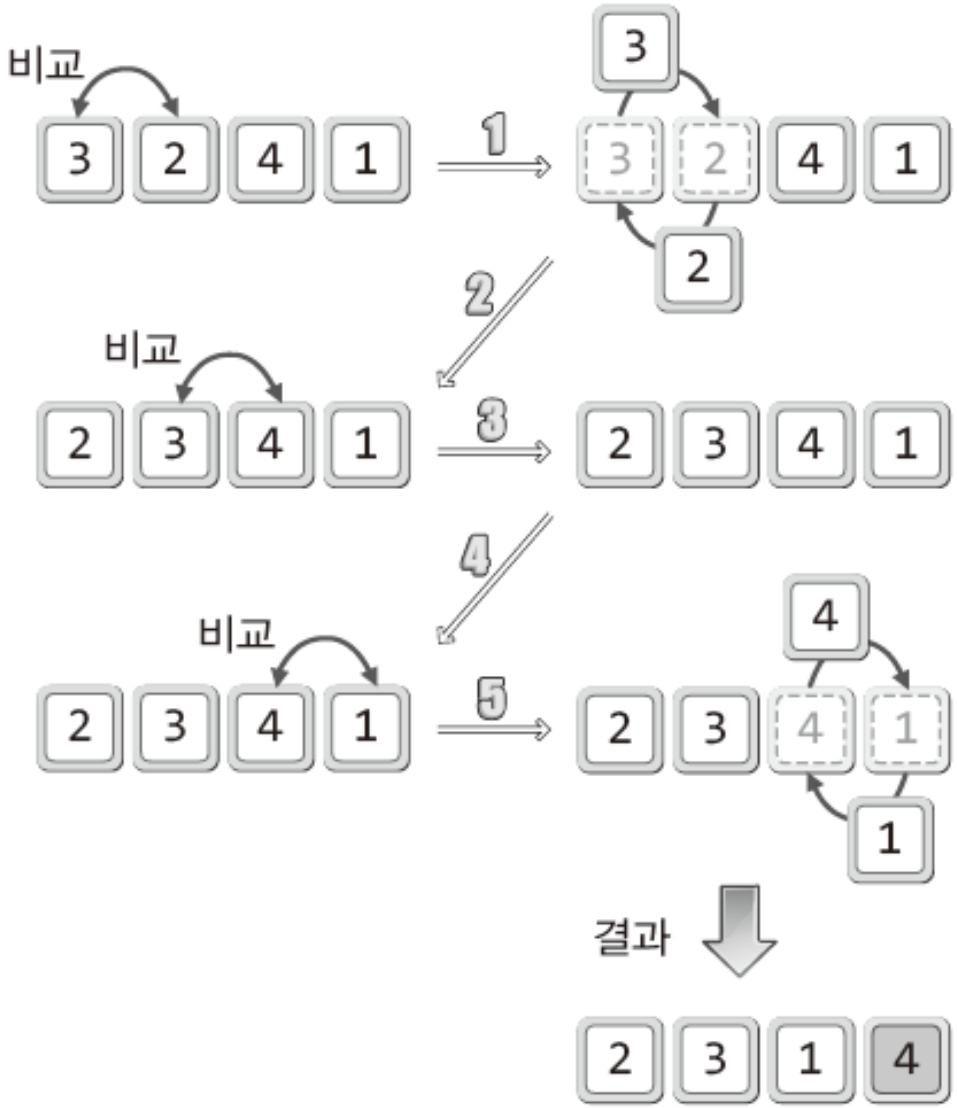
Data Structures and Algorithms

# 목차

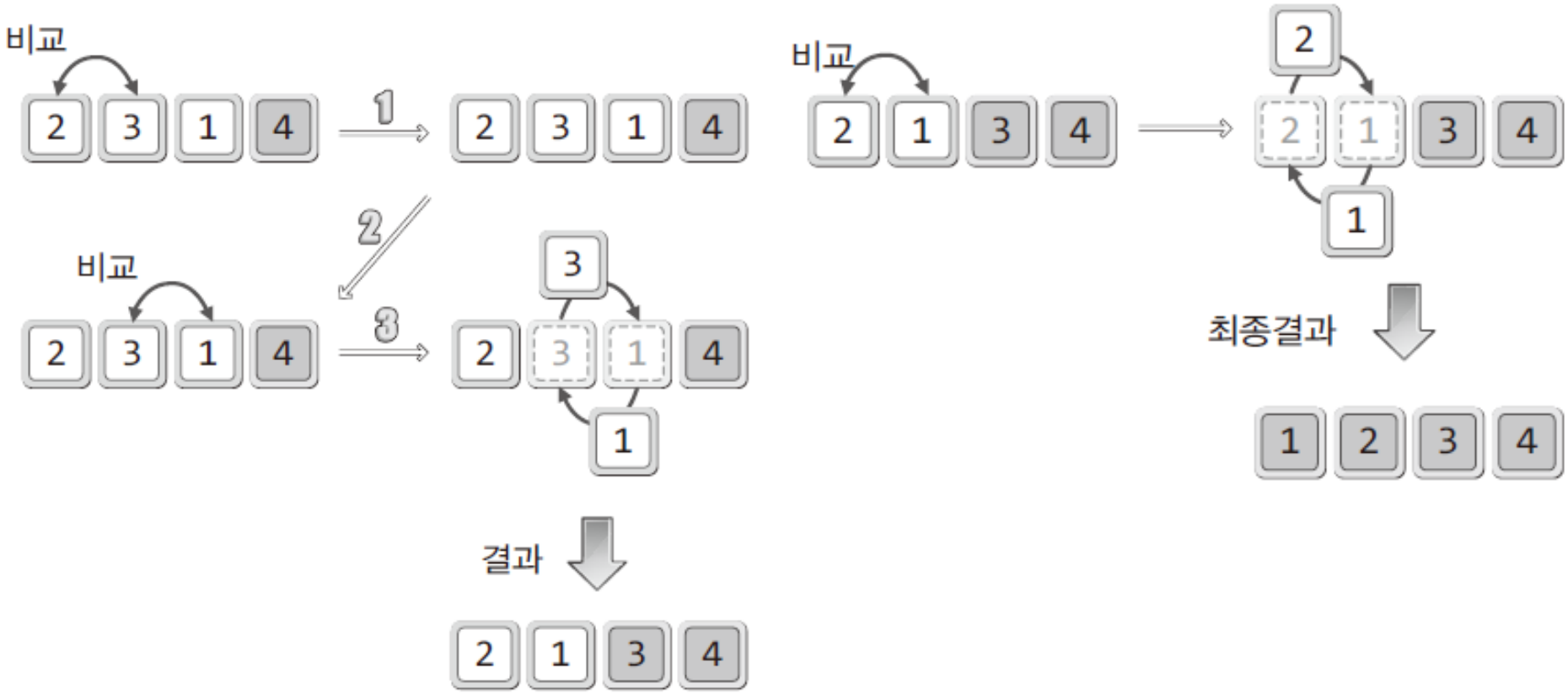
- 버블 정렬
- 선택 정렬
- 삽입 정렬
- 힙 정렬
- 병합 정렬
- 퀵 정렬
- 기수 정렬

# 버블 정렬

# 버블 정렬



# 버블 정렬


비교
2 3 1 4
1
2 3 1 4
비교
2 3 1 4
2
3
3
2 3 1 4
1
결과
2 1 3 4
비교
2 1 3 4
2
2 1 3 4
1
최종결과
1 2 3 4

# 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++)
        {
            if(arr[j] > arr[j+1])
            {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
     }
}
```

# 구현

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

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

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

    printf("\n");
    return 0;
}
```

# 성능평가

- 성능 평가의 두 가지 기준
  - 비교의 횟수: 두 데이터간의 비교 연산의 횟수
  - 이동의 횟수: 위치의 변경을 위한 데이터의 이동 횟수
- 비교의 횟수

```
for(i=0; i<n-1; i++)
    for(j=0; j<(n-i)-1; j++)
        if(arr[j] > arr[j+1])
        …
```

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

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

- 최악의 경우
  - 비교의 횟수와 이동의 횟수 일치

$$O(n^2)$$

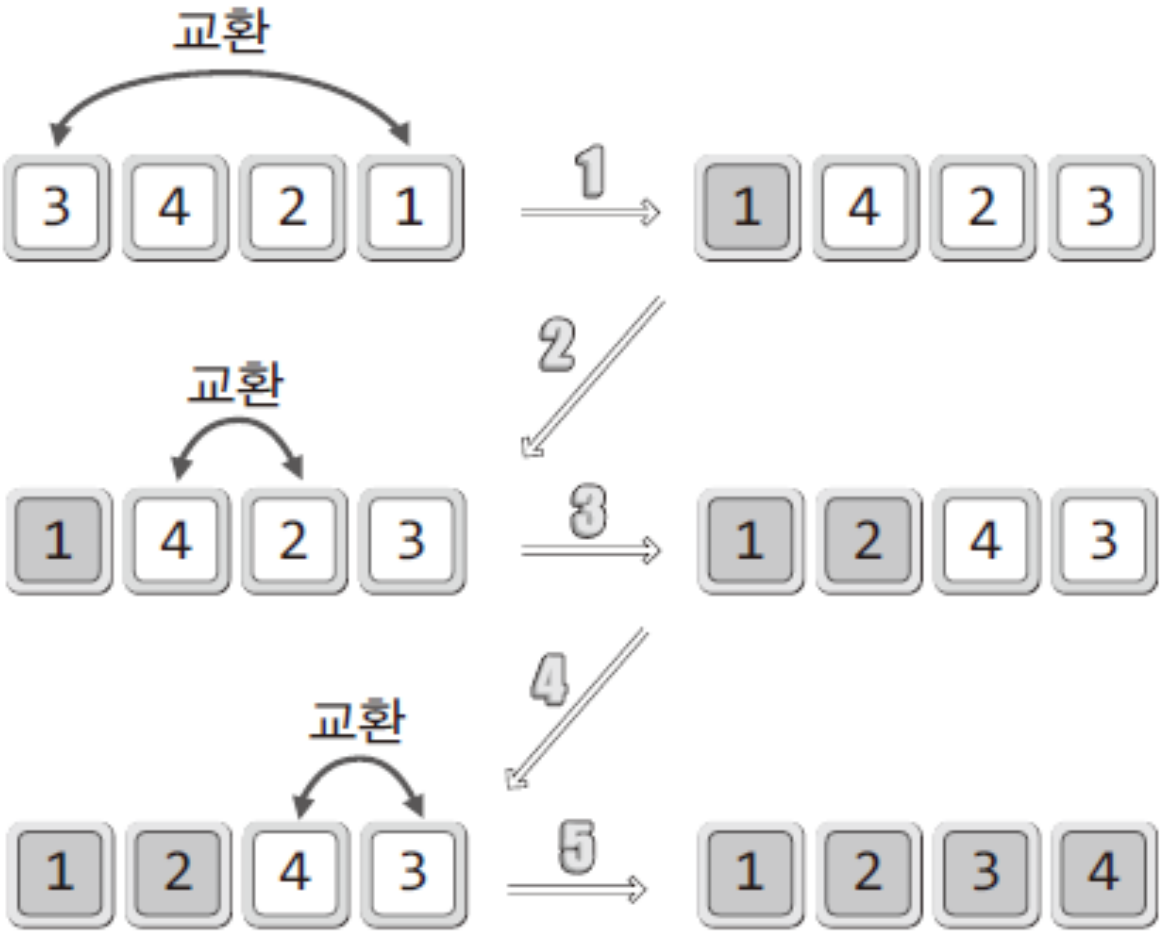
# 선택 정렬

# 선택 정렬


하나씩 선택해서 정렬

추가 메모리 공간 필요

# 개선된 선택 정렬




가장 작은 값 선택

정렬된 위치에 삽입

# 구현

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

    for(i=0; i<n-1; i++)
    {
        maxIdx = i;    // 정렬 순서상 가장 앞서는 데이터의 index
        for(j=i+1; j<n; j++)   // 최소값 탐색
        {
            if(arr[j] < arr[maxIdx])
                maxIdx = j;
        }

        /* 교환 */
        temp = arr[i];
        arr[i] = arr[maxIdx];
        arr[maxIdx] = temp;
    }
}
```

# 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;
}
```

# 성능평가

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

    temp = arr[i];
    arr[i] = arr[maxIdx];
    arr[maxIdx] = temp;
```

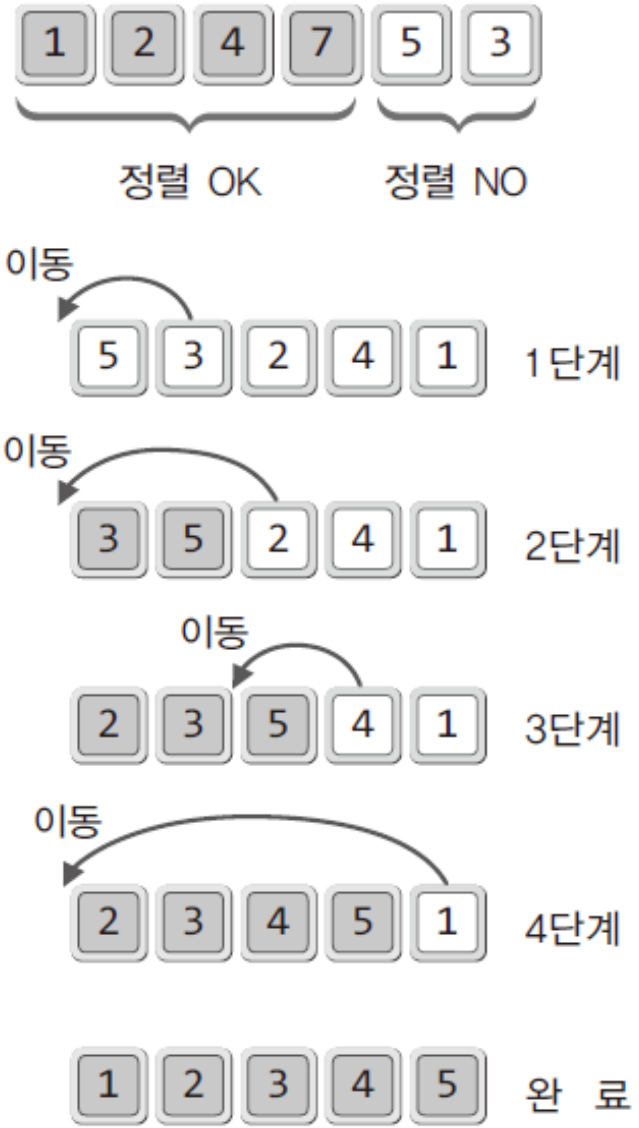
최악의 경우와 최상의 경우 구분 없이

데이터 이동의 횟수 동일

$O(n^2)$

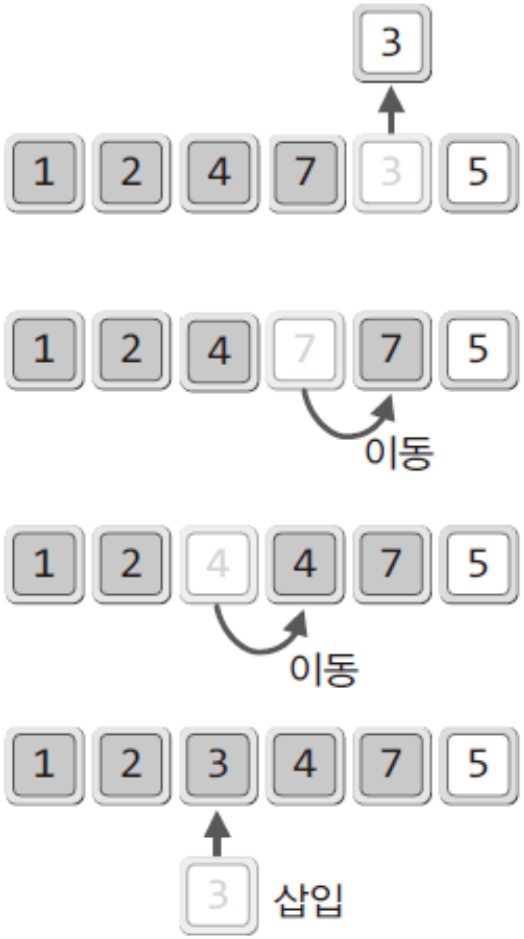
# 삽입 정렬

# 삽입 정렬




구현

## 뒤로 밀어 내기



# 구현

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

    for(i=1; i<n; i++)
    {
        insData = arr[i];   // 정렬 대상을 insData에 저장.
        for(j=i-1; j>=0 ; j--)
        {
            if(arr[j] > insData)
                arr[j+1] = arr[j];    // 비교 대상 한 칸 뒤로 밀기
            else
                break;   // 삽입 위치 찾았으니 탈출!
        }
        arr[j+1] = insData;  // 찾은 위치에 정렬 대상 삽입!
    }
}
```

# 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;
}
```

# 성능 평가

- 데이터의 비교 및 이동 연산이 안쪽 for문 안에 존재
- 최악의 경우 빅-오는 버블 및 삽입 정렬과 마찬가지로 $O(n^2)$

```
for(i=1; i<n; i++)
    for(j=i-1; j>=0 ; j--)
        if(arr[j] > insData) // 데이터간 비교연산
            arr[j+1] = arr[j]; // 데이터간 이동연산
        else
            break;   // 최악의 경우 break문 실행 안됨
```

# 힙 정렬

# 힙 정렬

- 힙의 특성
  - 힙에 데이터를 삽입 후 추출을 하는 과정에서 값이 정렬됨

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

void HeapSort(int arr[], int n, PriorityComp pc)
{
    Heap heap;
    int i;
    HeapInit(&heap, pc);
    for(i=0; i<n; i++) // 정렬 대상으로 힙 구성
        HInsert(&heap, arr[i]);

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

# main 함수

```
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;
}
```

Heap의 구현은 UsefulHeap.c 참고

# 성능 평가

- 하나의 데이터를 힙에 넣고 빼는 경우에 대한 시간 복잡도
  - 데이터 저장 시간 복잡도: $O(\log_2 N)$
  - 데이터 삭제 시간 복잡도: $O(\log_2 N)$

➡ $O(2\log_2 N)$ ➡ $O(\log_2 N)$

- N개의 데이터 $O(N\log_2 N)$

| $n$ | 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 |

빅오 비교 테이블

# 병합 정렬

Divide and Conquer

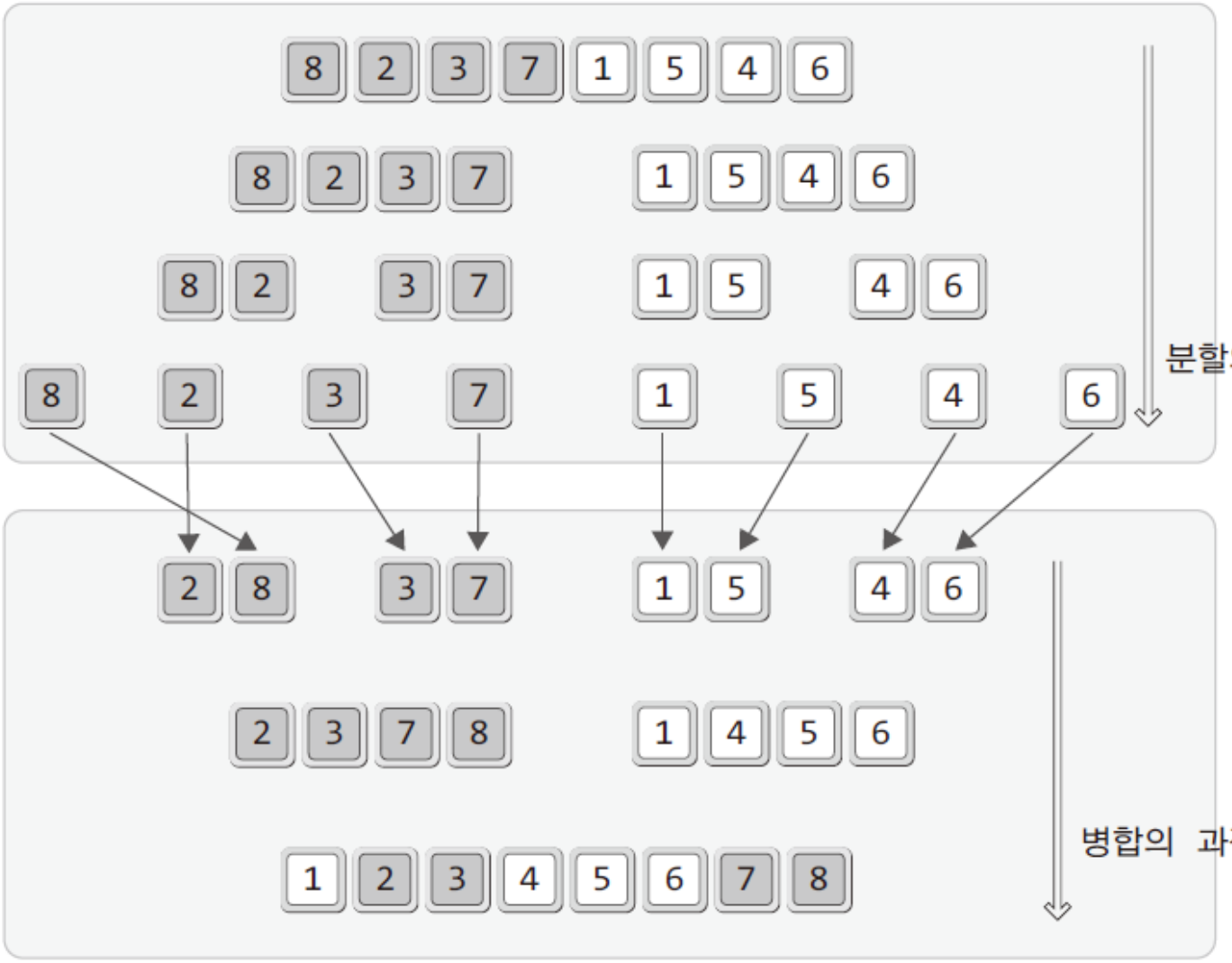
# 병합 정렬: Divide And Conquer

- 1단계 분할(Divide)
  - 해결이 용이한 단계까지 문제 분할
- 2단계 정복(Conquer)
  - 해결이 용이한 수준까지 분할된 문제 해결
- 3단계 결합(Combine)
  - 분할해서 해결한 결과 결합

# 병합 정렬


▶ [그림 10-9: 병합 정렬의 기본 원리]

# 분할과 정렬




재귀적 분할 과정

더 이상 분할 할 수 없을 때까지 분할

병합 과정이 중요

▶ [그림 10-10: 병합 정렬의 예]

# 1단계: 분할

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

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

        MergeSort(arr, left, mid); // 둘로 나눠서 각각을 정렬
        MergeSort(arr, mid+1, right);

        // 정렬된 두 배열을 병합한다.
        MergeTwoArea(arr, left, mid, right);
    }
}
```

# 3 단계: 병합

```
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);
}
```

병합 결과를 담을 메모리 공간 할당

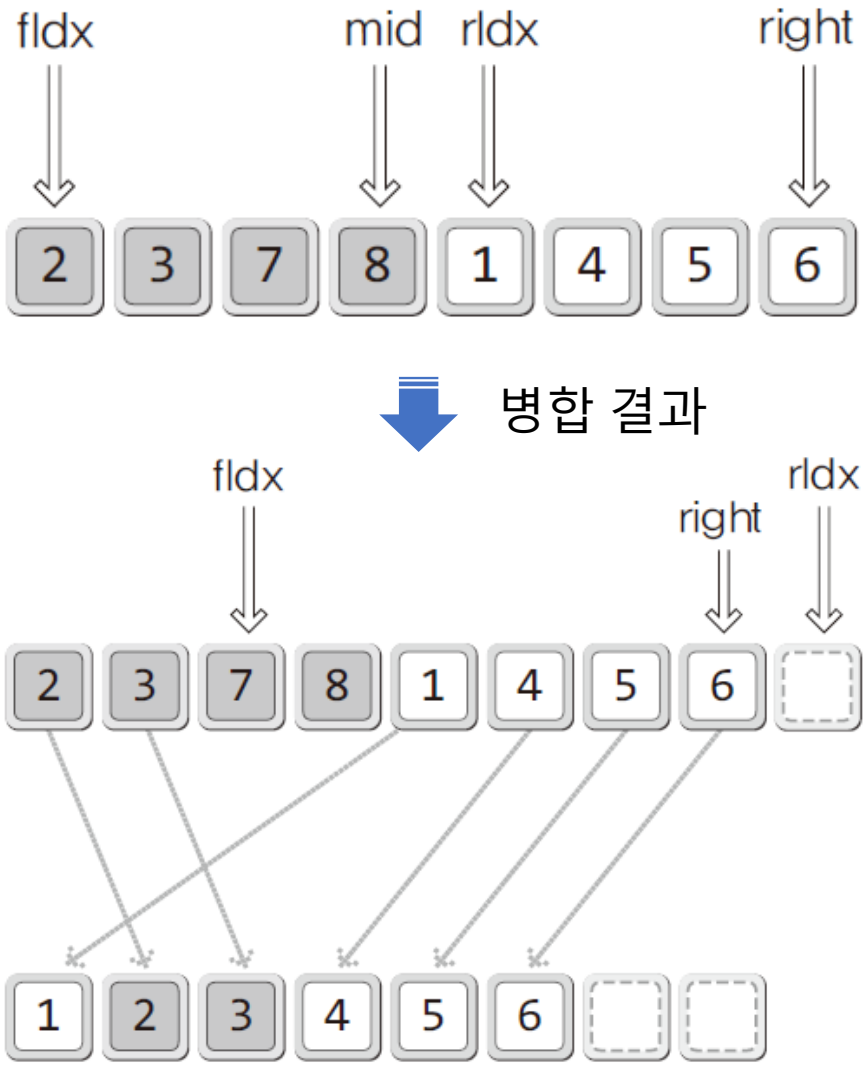
병합 할 두 영역의 데이터를 비교하여
배열 sortArr에 저장!

배열의 앞 부분이 sortArr로 모두 이동되어서 배열
뒷부분에 남은 데이터를 모두 sortArr로 이동!

배열의 뒷 부분이 sortArr로 모두 이동되어서 배열
앞부분에 남은 데이터를 모두 sortArr로 이동!

병합 결과를 옮겨 담는다!

# 병합 정렬: 병합 함수의 코드 설명




병합 코드

```
while(fIdx<=mid && rIdx<=right){
        if(arr[fIdx] <= arr[rIdx])
            sortArr[sIdx] = arr[fIdx++];
        else
            sortArr[sIdx] = arr[rIdx++];
        sIdx++;
}
```

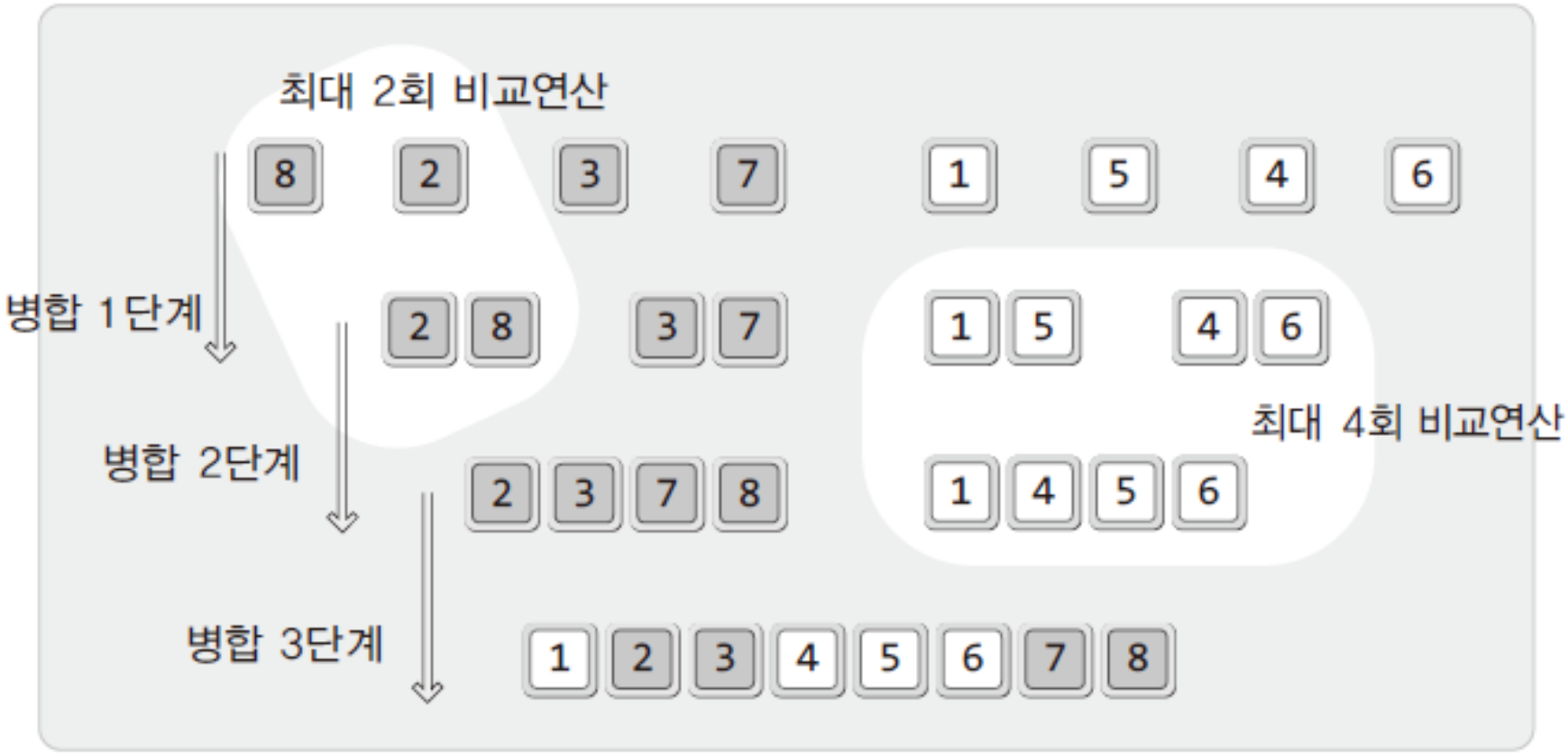
# 성능 평가: 비교 연산

- 데이터의 비교 및 데이터의 이동은 MergeTwoArea 함수를 중심으로 진행!
  - 병합 정렬의 성능은 MergeTwoArea 함수를 기준으로 계산!

```
while(fIdx<=mid && rIdx<=right){
        if(arr[fIdx] <= arr[rIdx])
            sortArr[sIdx] = arr[fIdx++];
        else
            sortArr[sIdx] = arr[rIdx++];
        sIdx++;
}
```

MergeTwoArea의 핵심

# 성능 평가 : 비교 연산




- 1과 4 비교 후 1 이동
- 5와 4 비교 후 4 이동
- 5와 6 비교 후 5 이동
- 6을 이동하기 위한 비교

# 성능 평가 : 비교 연산

- 정렬 대상 데이터가 n개 일 때, 병합 단계마다 최대 n 번 비교
  - 비교 연산 횟수 : $n \log_2 n$

- 빅오: $O(n \log_2 n)$

# 성능 평가: 이동

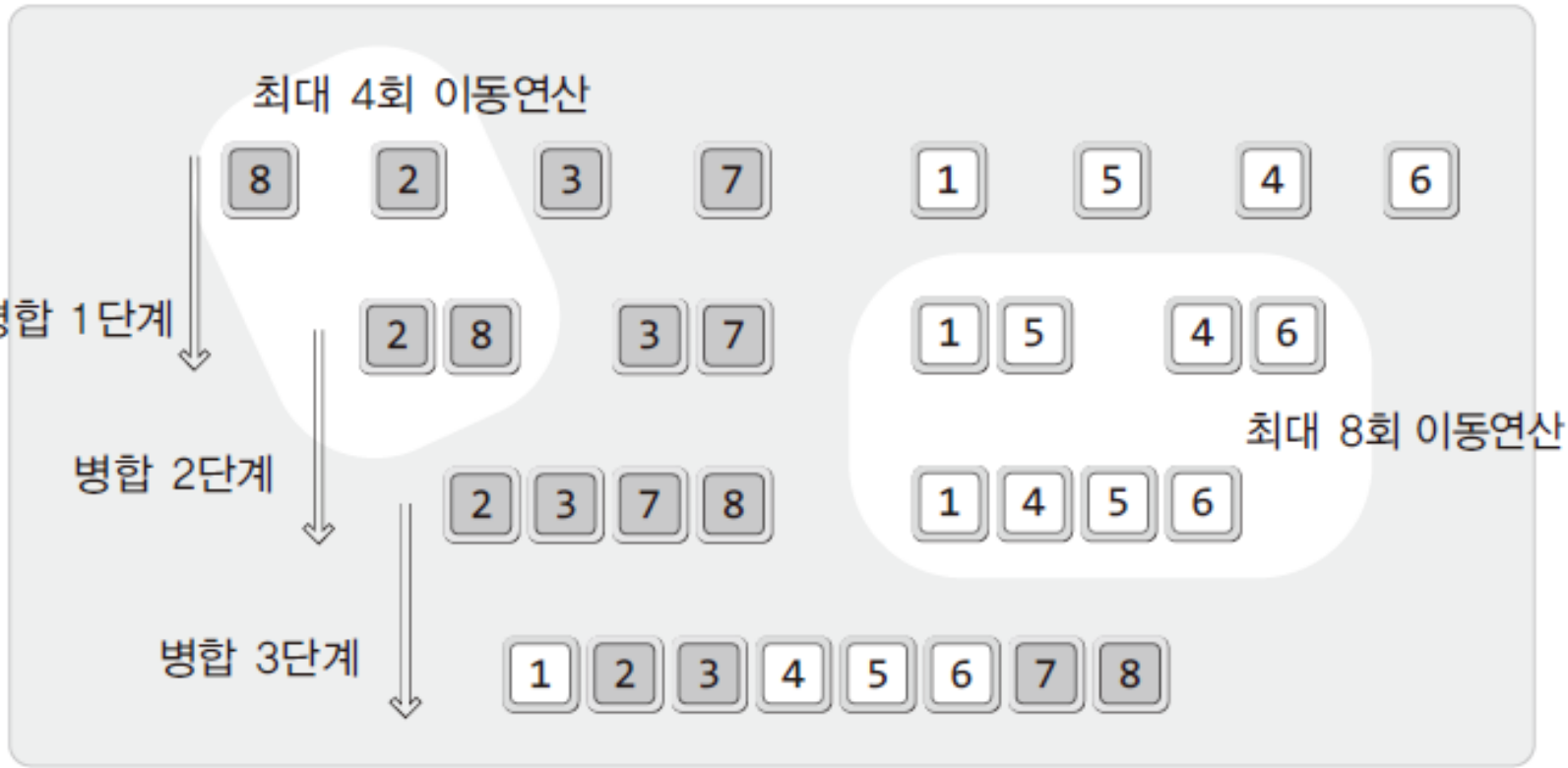
- 임시 배열에 데이터를 병합하는 과정에서 한 번!
- 배열에 저장된 모든 데이터를 원위치로 옮기는 과정에서 한 번!

8과 2를 정렬해서 옮기면서 2회

그 결과를 다시 옮기면서. 2회

$2n\log_2 n \Rightarrow O(n\log_2 n)$

최악, 최선 상관 없이!



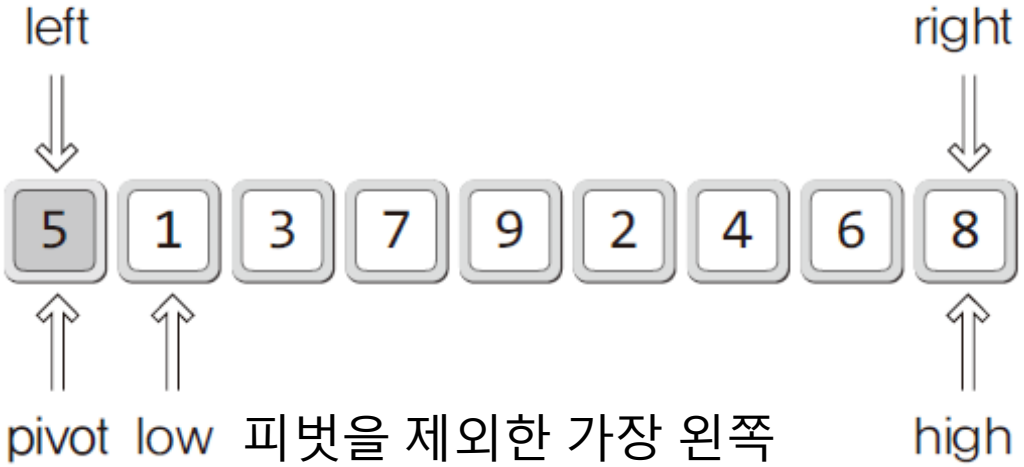
# 퀵 정렬

# 퀵 정렬: 1단계, 초기화

- 총 5개(left, right, pivot, low, high) 핵심 변수 선언

정렬 대상의 가장 왼쪽 지점

정렬 대상의 가장 오른쪽 지점



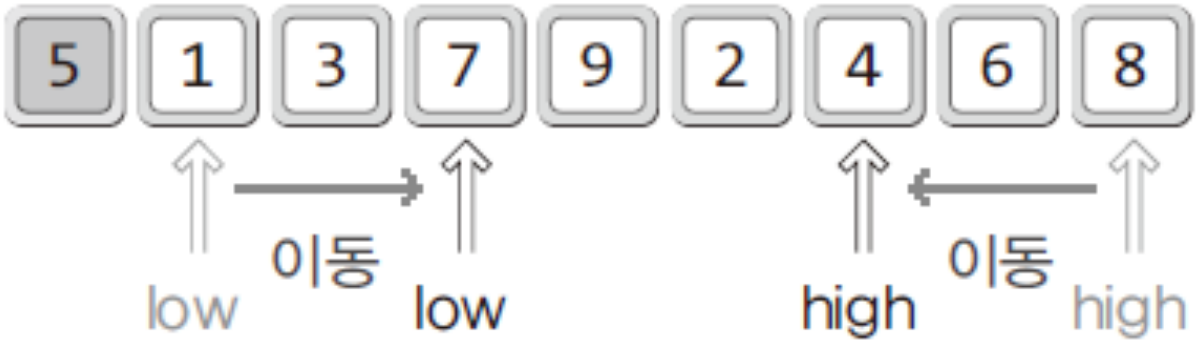

피벗
기준 중심축

피벗을 제외한 가장 왼쪽

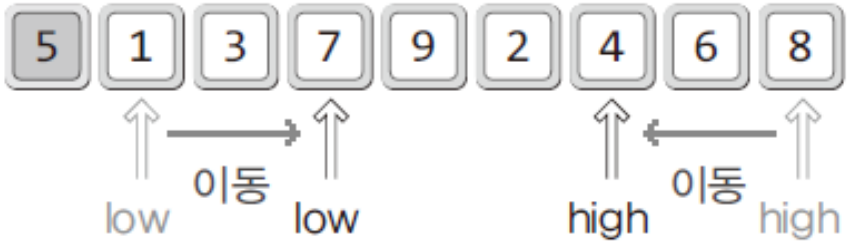
피벗을 제외한 가장 오른쪽

# 퀵 정렬: 2단계, low and high 이동

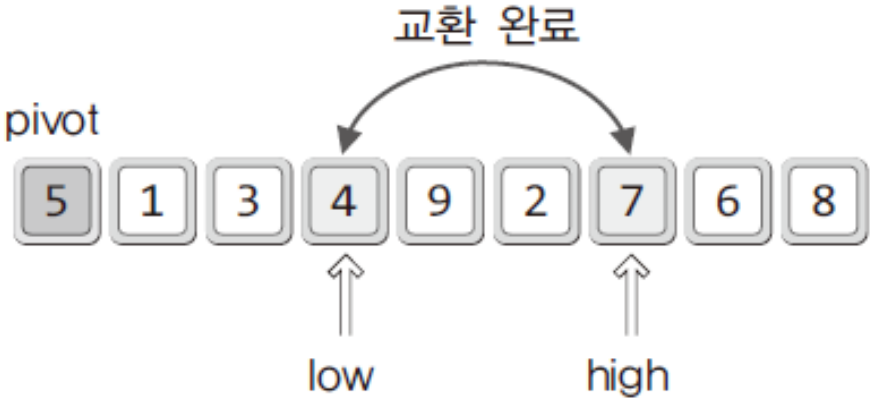
- low의 이동: 피벗보다 클 때까지 우측으로
- high의 이동: 피벗보다 작을 때까지



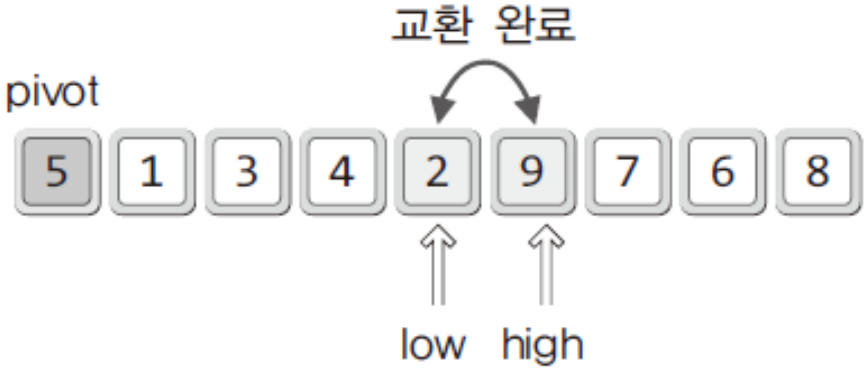
# 퀵 정렬: 3단계, low와 high 교환



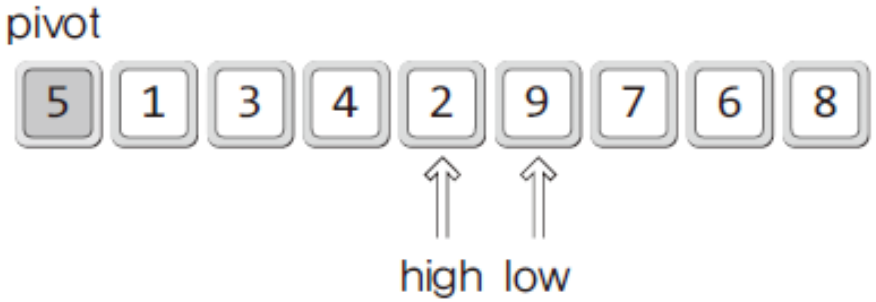

1 low 와 high 이동




2 low와 high의 데이터 교환

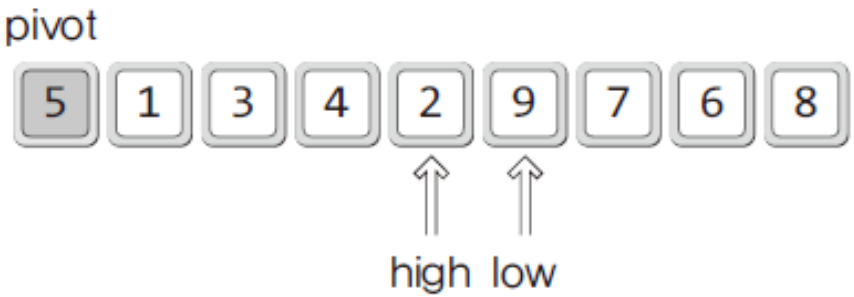



3 1, 2번 반복

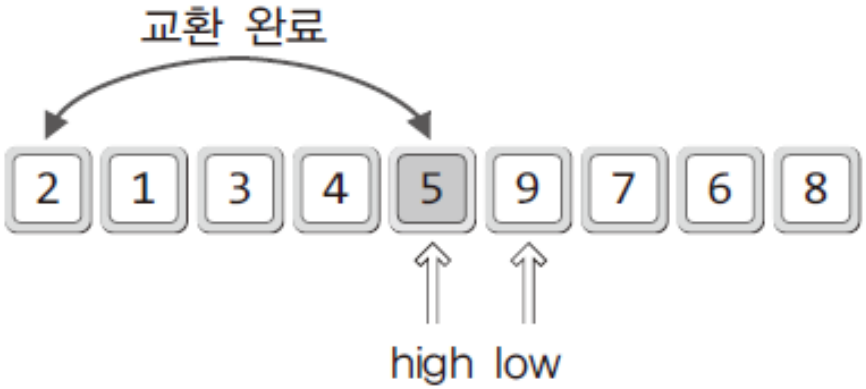



4 high와 low가 역전할 때까지 반복

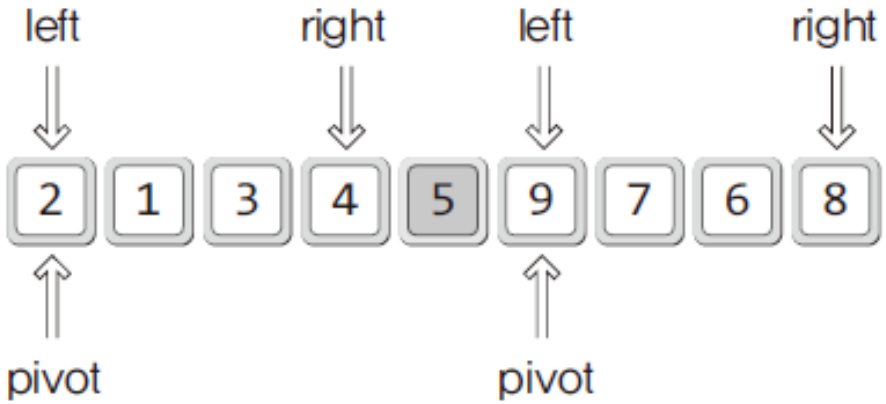
# 퀵 정렬: 4단계, 피벗 이동




1 high와 low가 역전




2 피벗과 high의 데이터 교환




3 두 개의 영역으로 나누어 반복

# 구현

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

    while(low <= high)    // 교차되지 않을 때까지 반복
    {
        while(pivot > arr[low])
            low++;

        while(pivot < arr[high])
            high--;

        if(low <= high)    // 교차되지 않은 상태라면 Swap 실행
            Swap(arr, low, high);    // low와 high가 가리키는 대상 교환
    }
    Swap(arr, left, high);    // 피벗과 high가 가리키는 대상 교환
    return high;    // 옮겨진 피벗의 위치 정보 반환
}
```

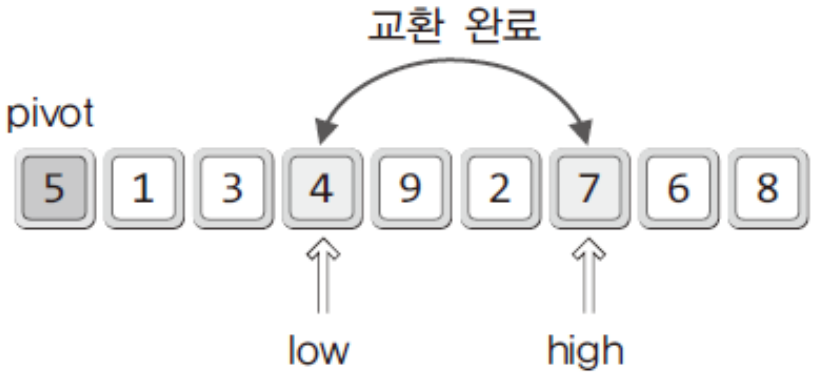
# 구현: 파티션

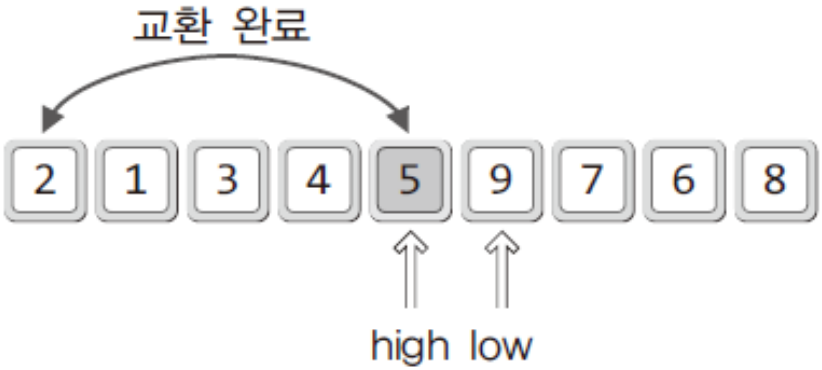
```
int Partition(int arr[], int left, int right)
{
    int pivot = arr[left];
    int low = left+1;
    int high = right;

    while(low <= high)
    {
        while(pivot > arr[low])
            low++;

        while(pivot < arr[high])
            high--;

        if(low <= high)
            Swap(arr, low, high);
    }
    Swap(arr, left, high);
    return high;
}
```

# 구현: 재귀적 파티션

```
void QuickSort(int arr[], int left, int right)
{
    if(left <= right)
    {
        int pivot = Partition(arr, left, right);    // 둘로 나눠서
        QuickSort(arr, left, pivot-1);    // 왼쪽 영역을 정렬
        QuickSort(arr, pivot+1, right);    // 오른쪽 영역을 정렬
    }
}


int Partition(int arr[], int left, int right)
{
    …
    while(low <= high)  // 항상 참일 수 밖에 없는 상황 존재
    { …
        while(pivot >= arr[low] && low <= right) // 문제 상황 해소
            low++;
        while(pivot <= arr[high] && high >= (left+1)) // 문제 상황 해소
            high--;
```

# 퀵 정렬: 피벗

- 피벗은 중앙 값에 가까울 수록 성능이 좋음
  - 확률적으로 중앙에 가까운 값 선정 방법:
    - 첫 번째, 마지막, 가운데의 값을 선택하여 중앙 값으로 사용
    - 첫 세개의 값 중 중앙 값 사용
- 정렬과정에서 피벗 변경 횟수가 적을 수록 성능이 좋음


불규칙한 패턴인 경우 성능이 좋음


어느 정도 정렬이 되어 있고
피벗이 한 쪽 끝에 있으면 성능 최악

# 최선의 경우 성능평가

- 모든 데이터가 피벗과 데이터 비교: n번
- 피벗 이동 후 약 n번 비교
  - 1차: 31개의 데이터가 있을  때 15개씩 둘로 나뉘어 2 덩어리
  - 2차: 7개씩 2덩어리, 총 4덩어리
  - 3차: 3개씩 2덩어리, 총 8덩어리
  - 4차:1개씩 2덩어리, 총 16덩어리
  - 계속 반씩 나뉘는 구조

$$k = log_2 n$$

- 최선의 경우 빅오 $O(n \log_2 n)$

# 최악의 경우 성능 평가

- 둘로 나뉘는 횟수: n
- 피벗 선정 후 비교 횟수: n
- 최악의 경우 빅 오: $O(n^2)$

- 퀵 정렬은 평균적으로 가장 좋은 성능을 보이는 정렬 알고리즘
  - 데이터 이동이 적음
  - 별도의 메모리 공간 유지 필요 없음

# 기수 정렬

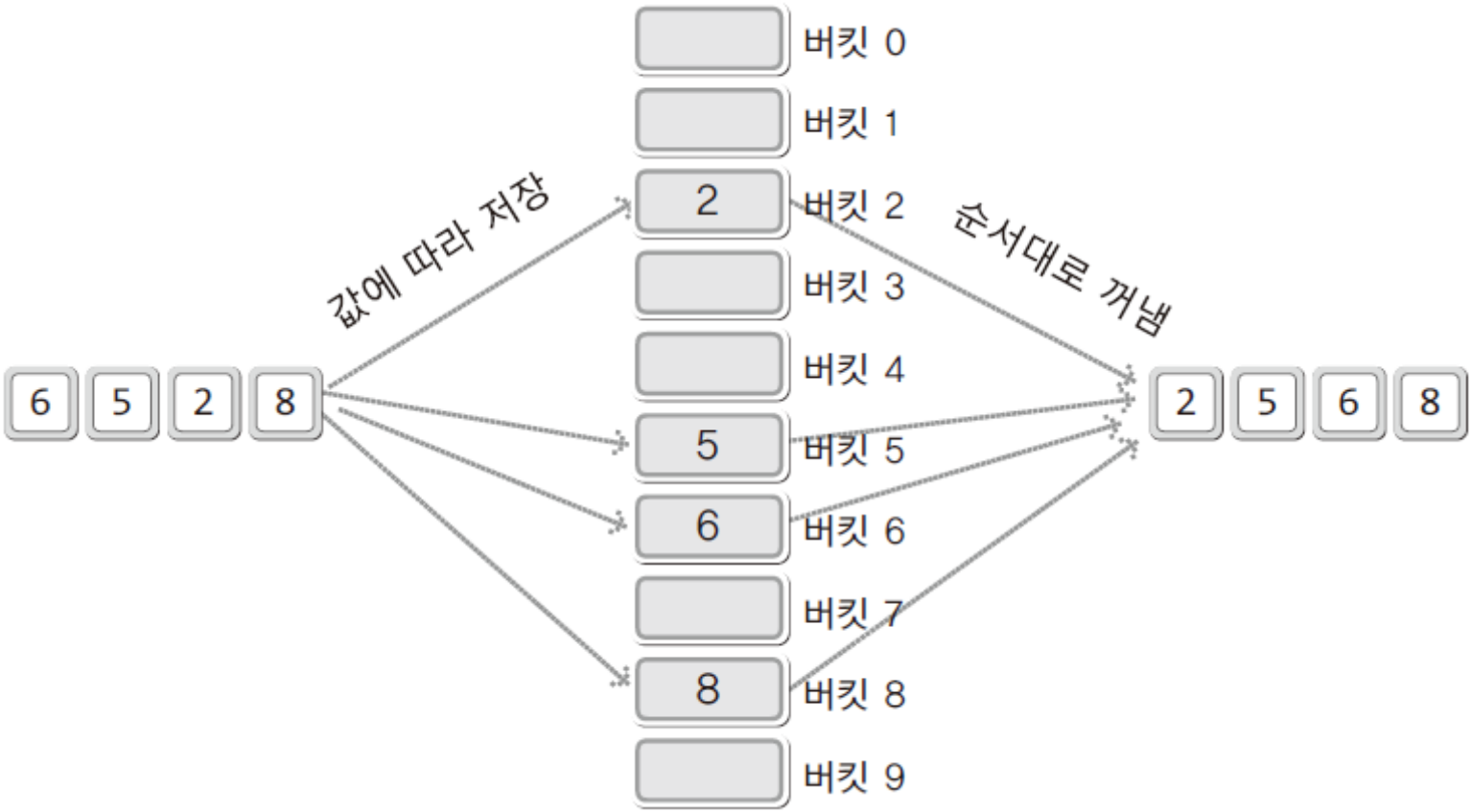
# 기수 정렬

- 특징
  - 정렬순서가 앞섬이나 뒤섬을 비교하지 않음
  - 정렬 알고리즘의 한계로 알려진 O(nlog2n)을  뛰어넘을 수도 있음
  - 적용할 수 있는 대상이 매우 제한적임
  - 길이가 동일한 데이터들의 정렬에 용이함

- 기수 정렬 사용 가능 예
  - 배열에 저장된 1, 7, 9, 5, 2, 6을 오름차순으로 정렬하라!
  - 영단어 red, why, zoo, box를 사전편찬 순서대로 정렬하여라!
  - 배열에 저장된 21, -9, 125, 8, -136, 45를 오름차순으로 정렬하라!

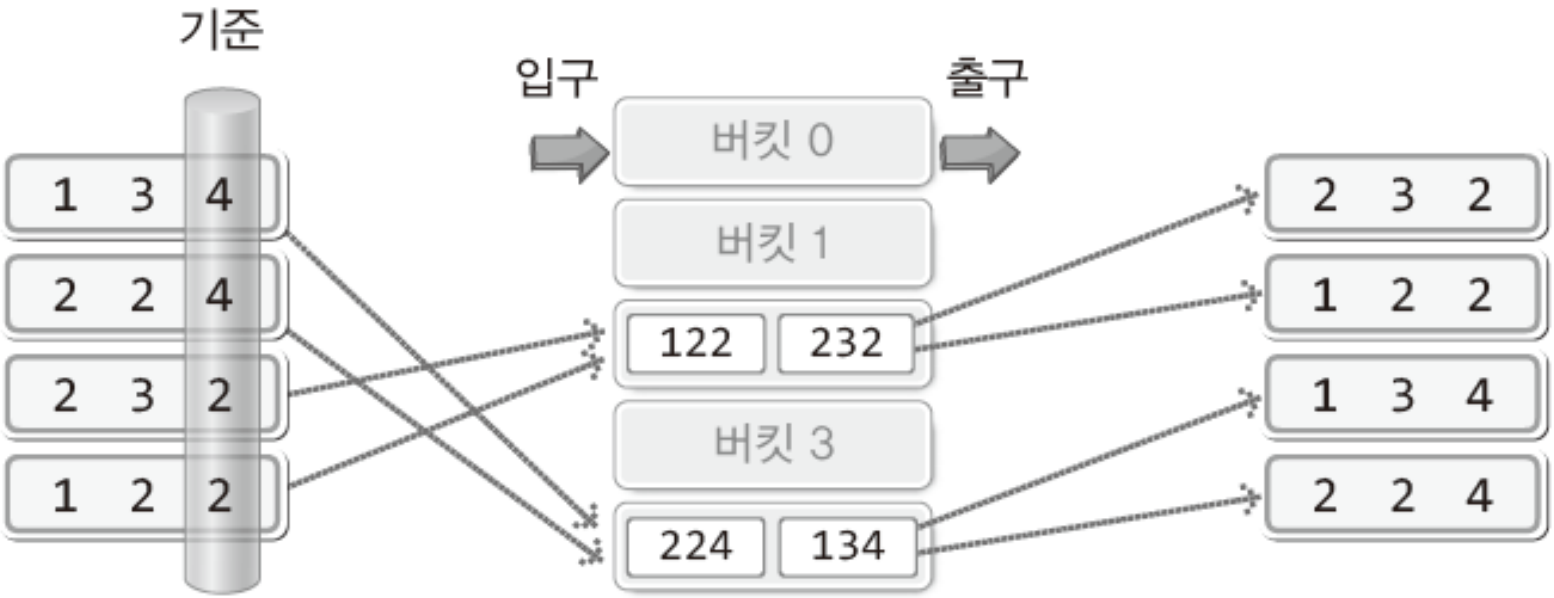
# 원리

- 기수(radix): 주어진 데이터를 구성하는 기본 요소(기호)
- 버킷(bucket): 기수의 수에 해당하는 만큼의 버킷을 활용
  - 버킷에 값을 넣고 순서대로 추출, 비교가 없음

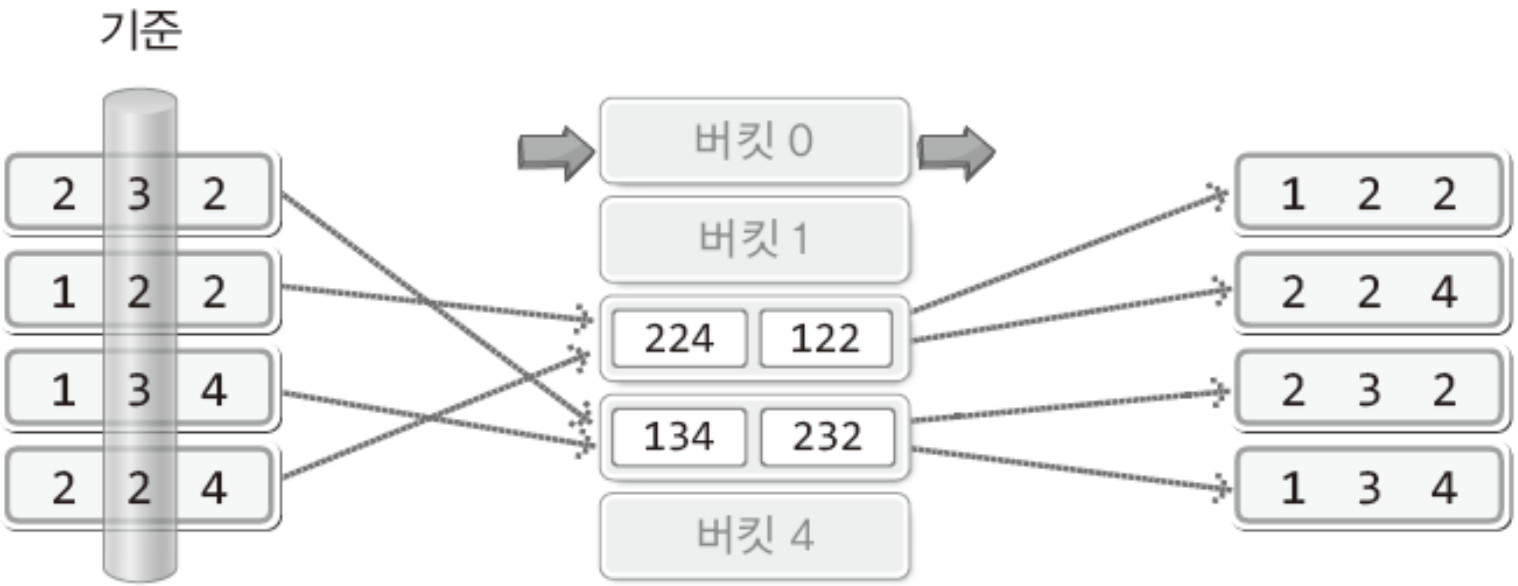


# 기준: Least Significant Digit (LSD) 1/3

- List Significant Digit을 시작으로 정렬

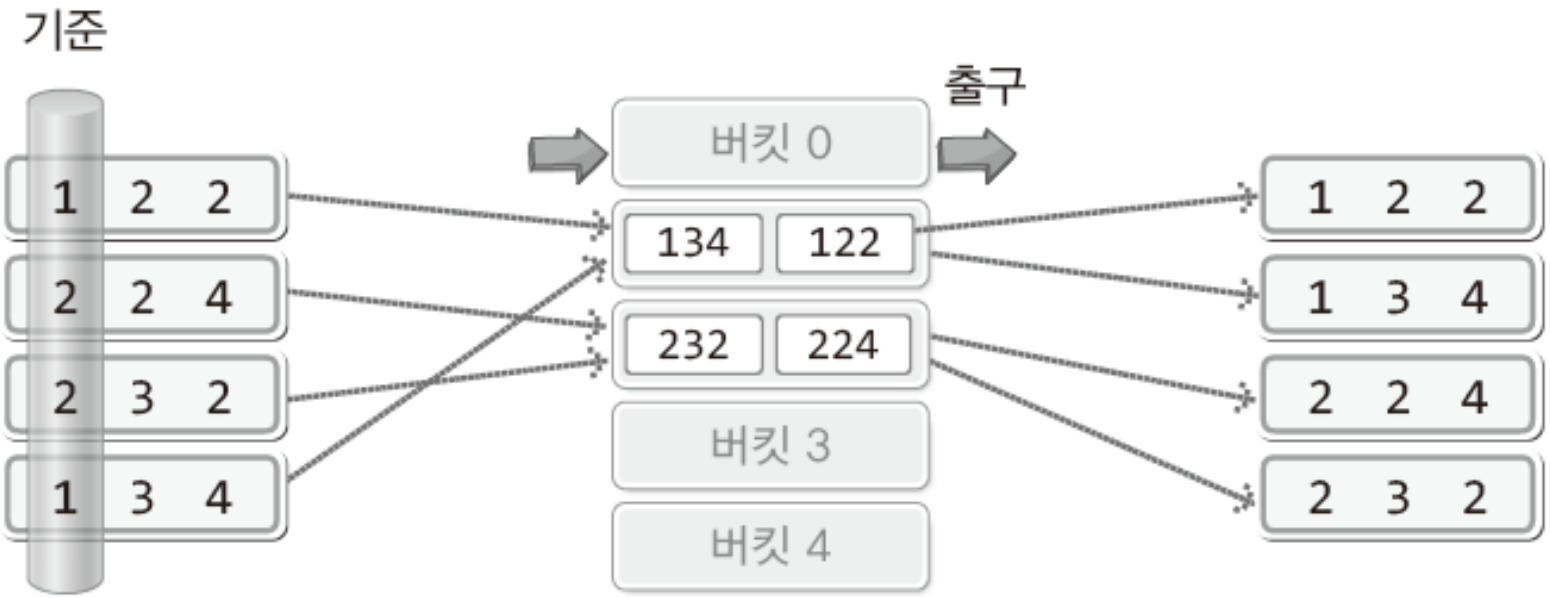


# 기준: Least Significant Digit (LSD) 2/3



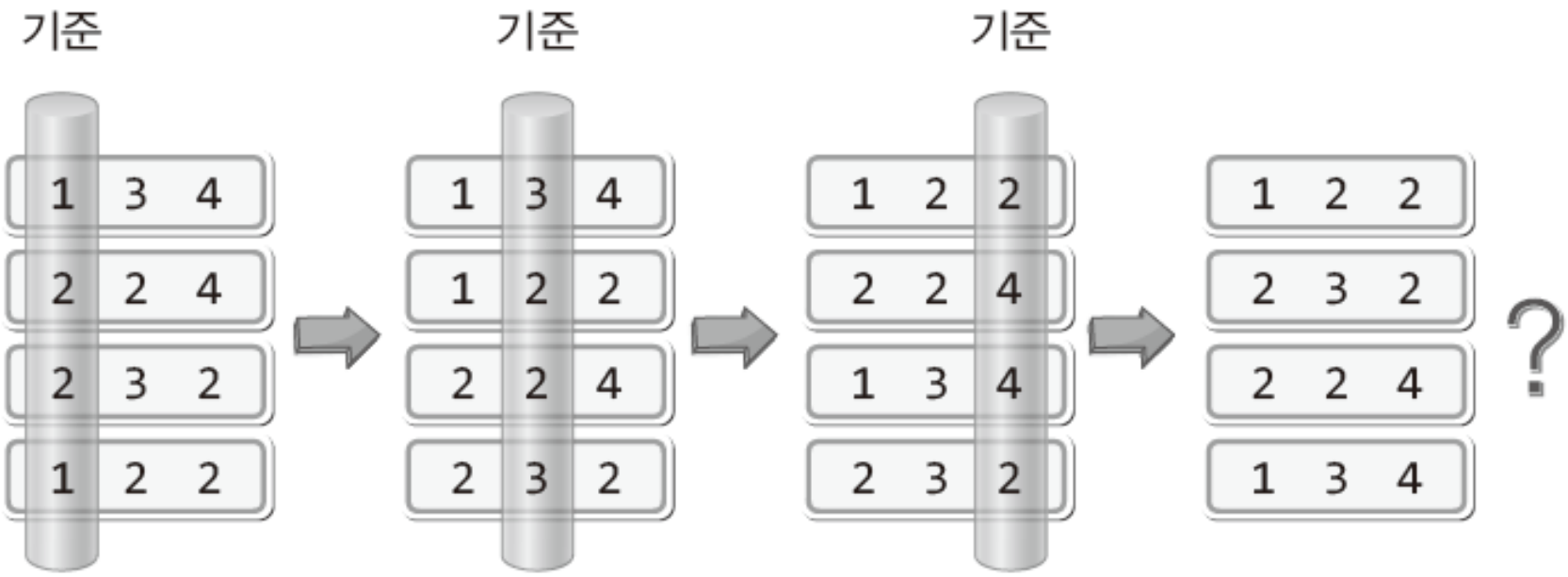
# 기준: Least Significant Digit (LSD) 3/3

- 마지막까지 진행을 해야 값의 우선순위대로 정렬 됨



# 기준: Most Significant Digit (MSD)

- MSD는 정렬의 기준 선정 방향이 LSD와 반대이다! 방향성 외의 차이는 무엇인가?
  - 점진적으로 정렬이 완성됨
  - 중간 과정에서 정렬이 완료된 데이터는 더 이상의 정렬하지 않아야 함



# LSD 정렬 기준 구현

- MSD와 LSD의 빅오는 같음
- LSD를 기준으로 하는 구현이 일반적임
  - LSD 기준은 모든 데이터에 동일한 정렬 방법을 적용할 수 있음
  - MSD 기준은 경우를 나눠 정렬해야 함

- (양의 정수 only) LSD 기반 정렬 용 자릿수 추출 알고리즘
  - NUM으로부터 첫 번째 자리 숫자 추출: NUM / 1 % 10
  - NUM으로부터 두 번째 자리 숫자 추출: NUM / 10 % 10
  - NUM으로부터 세 번째 자리 숫자 추출: NUM / 100 % 10

# 구현

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

    for(bi=0; bi<BUCKET_NUM; bi++)
        QueueInit(&buckets[bi]);

    for(pos=0; pos<maxLen; pos++) // 가장 긴 데이터의 길이만큼 반복
    {
        for(di=0; di<num; di++) // 정렬 대상의 수만큼 반복
        {
            radix = (arr[di] / divfac) % 10; // N번째 자리의 숫자 추출
            Enqueue(&buckets[radix], arr[di]); // 추출한 숫자를 데이터 버킷에 저장
        }
        for(bi=0, di=0; bi<BUCKET_NUM; bi++) // 버킷 수만큼 반복
        {
            // 버킷에 저장된 것 순서대로 다 꺼내서 다시 arr에 저장
            while(!QIsEmpty(&buckets[bi]))
                arr[di++] = Dequeue(&buckets[bi]);
        }
        divfac *= 10; // N번째 자리의 숫자 추출을 위한 피제수의 증가
    }
}
```

# 성능 평가

- 버킷에 데이터 삽입/추출을 근거로 빅-오 결정!

```
void RadixSort(int arr[], int num, int maxLen)
{
    …
    for(pos=0; pos<maxLen; pos++)
    {
        for(di=0; di<num; di++)
        {
            radix = (arr[di] / divfac) % 10;
            Enqueue(&buckets[radix], arr[di]);
        }
        for(bi=0, di=0; bi<BUCKET_NUM; bi++)
        {
            while(!QIsEmpty(&buckets[bi]))
                arr[di++] = Dequeue(&buckets[bi]);
        }
        divfac *= 10;
    }
}
```

삽입

추출

삽입+추출=한 쌍
maxLen x num

O( L n )

L: 정렬 대상 길이
N: 정렬 대상 수