

윤성우의 열혈 자료구조

: C언어를 이용한 자료구조 학습서



Chapter 10. 정렬

Introduction To Data Structures Using C

Chapter 10. 정렬

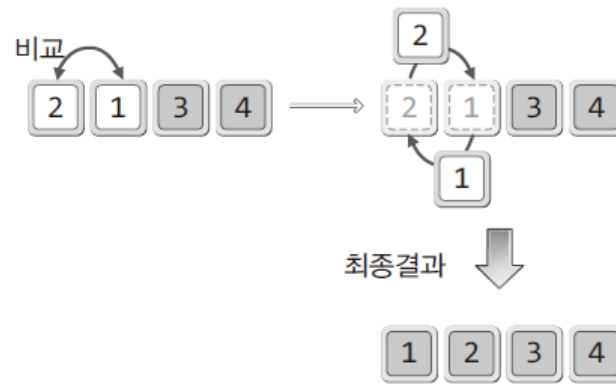
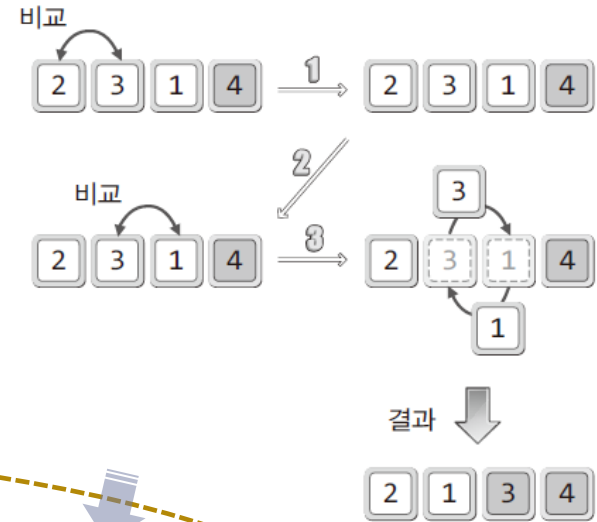
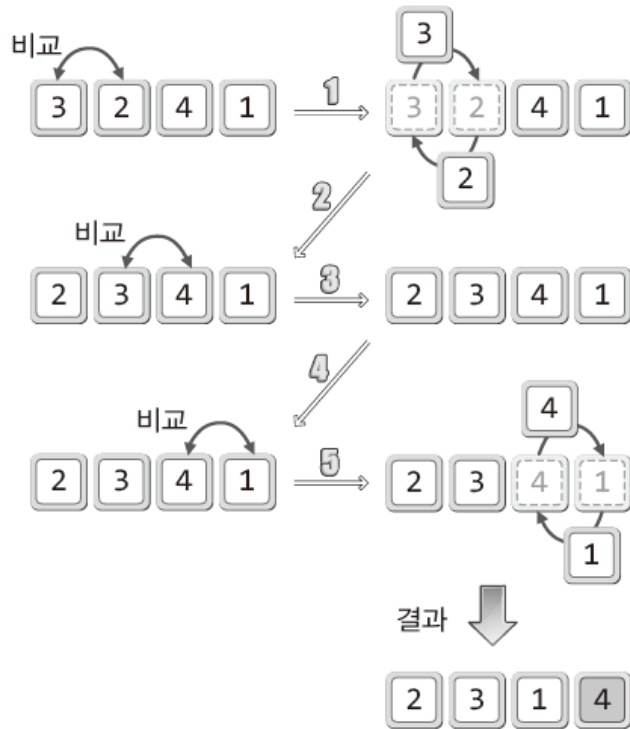


Chapter 10-1:

단순한 정렬 알고리즘



버블 정렬: 이해



버블 정렬: 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])
            {
                // 데이터의 교환 /////
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

버블 정렬의 실질적 구현 코드

```
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;
}
```

실행결과

1	2	3	4	
---	---	---	---	--

버블 정렬: 성능평가

성능 평가의 두 가지 기준

- 비교의 횟수 두 데이터간의 비교연산의 횟수
- 이동의 횟수 위치의 변경을 위한 데이터의 이동횟수

비교의 횟수

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

비교 연산

최악의 경우

비교의 횟수와 이동의 횟수는 일치한다.

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



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



$$O(n^2)$$

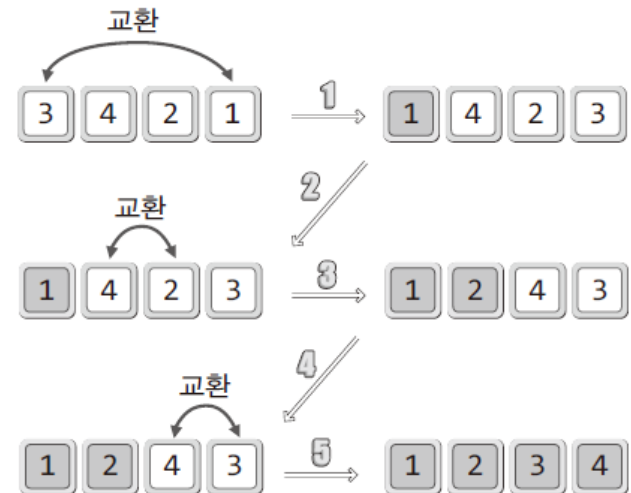
선택 정렬: 이해



하나씩 **선택**해서 정렬 결과를 완성해 나간다!

별도의 메모리 공간이 요구된다는 단점이 있다!

개선!



하나씩 비워 가면서 이동을 시킨다.

선택 정렬: 구현

```
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++)    // 최솟값 탐색
        {
            if(arr[j] < arr[maxIdx])
                maxIdx = j;
        }

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

```
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;
}
```

실행결과

1 2 3 4

선택 정렬: 성능평가

비교의 횟수

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

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



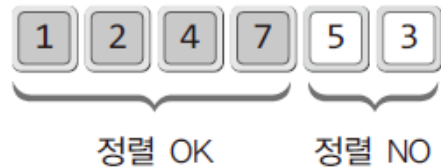
$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2-n}{2}$$



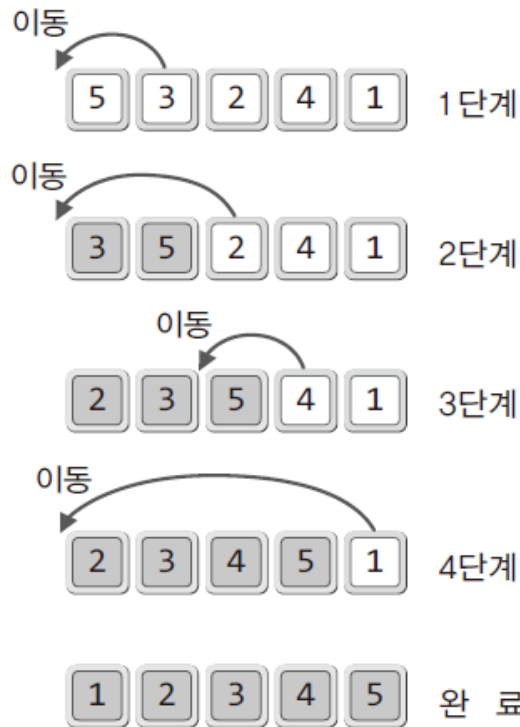
$$O(n^2)$$

최악의 경우와 최상의 경우 구분 없이 데이터 이동의 횟수는 동일하다!

삽입 정렬: 이해

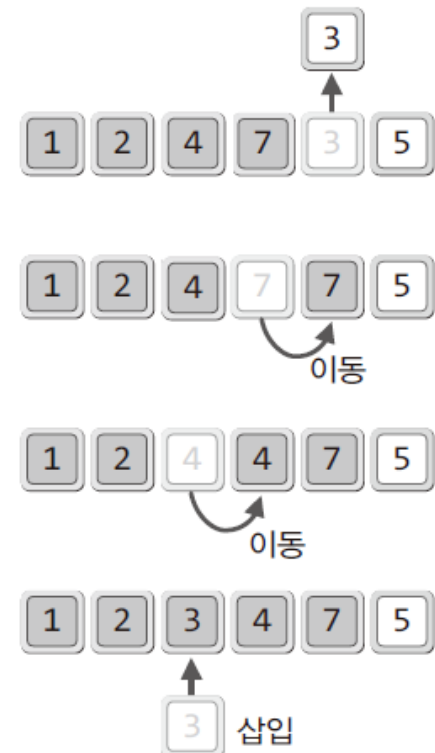


정렬이 완료된 영역과 그렇지 않은 영역을 구분하는 방법



구현을 고려하면!

뒤로 밀어 내는 방법을 포함한 그림



삽입 정렬: 구현

```
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 대상 저장

비교 대상 뒤로 한 칸 밀기

찾은 위치에 정렬대상 삽입

```
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;
}
```

실행결과

1	2	3	4	5
---	---	---	---	---

삽입 정렬: 성능평가

```
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 이 break문은 한번도 실행이 되지 않는다.

```
....
```

```
}
```

데이터의 비교 및 이동 연산이 안쪽 for문 안에 존재한다.

따라서 최악의 경우 빅-오는 버블 및 삽입 정렬과 마찬가지로 $O(n^2)$

Chapter 10. 정렬



Chapter 10-2:

복잡하지만 효율적인 정렬 알고리즘



힙 정렬: 이해와 구현

힙의 특성을 활용하여, 힙에 정렬할 대상을 모두 넣었다가 다시 꺼내어 정렬을 진행한다!

```
int PriComp(int n1, int n2)
{
    return n2-n1;      // 오름차순 정렬을 위한 문장
// return n1-n2;
}

// 힙에서 요구하는 정렬 기준 함수와 동일한 기준을 적용한다!
void HeapSort(int arr[], int n, PriorityComp pc)
{
    Heap heap;
    int i;
    HeapInit(&heap, pc);

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

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

```
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;
}
```

UsefulHeap.h

UsefulHeap.c

HeapSort.c

1 2 3 4

실행결과

힙 정렬: 성능평가

하나의 데이터를 힙에 넣고 빼는 경우에 대한 시간 복잡도

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

↓ 하나로 묶으면

$O(2\log_2 n)$, 그런데

앞의 2는 빅-오에서 무시 할 수 있으므로! $O(\log_2 n)$

N개의 데이터



$O(n\log_2 n)$

두 빅-오의 비교를 위한 테이블

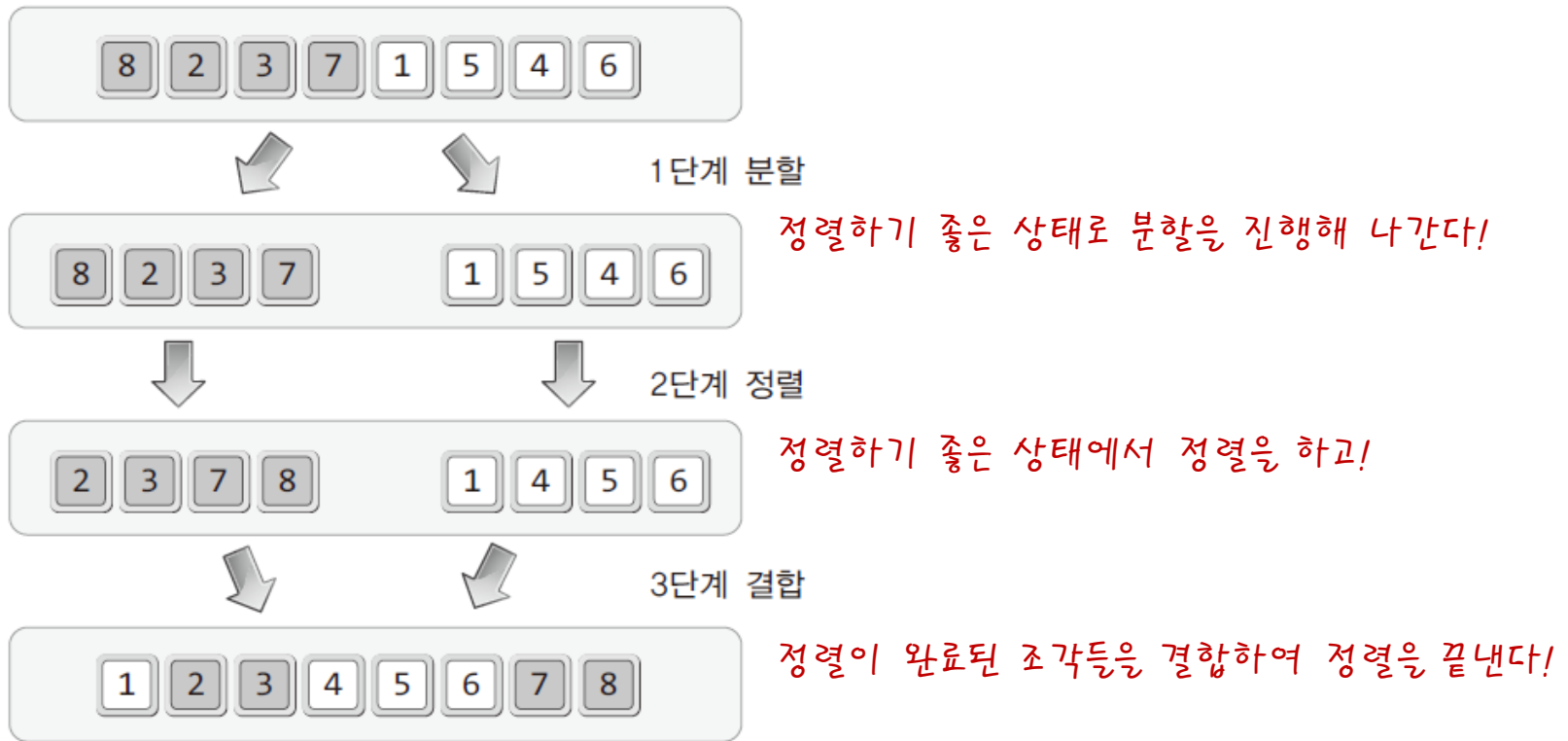
n	10	100	1,000	3,000	5,000
n^2	100	10,000	1,000,000	9,000,000	25,000,000
$n \log_2 n$	66	664	19,931	34,652	61,438

병합 정렬: Divide And Conquer

- 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.

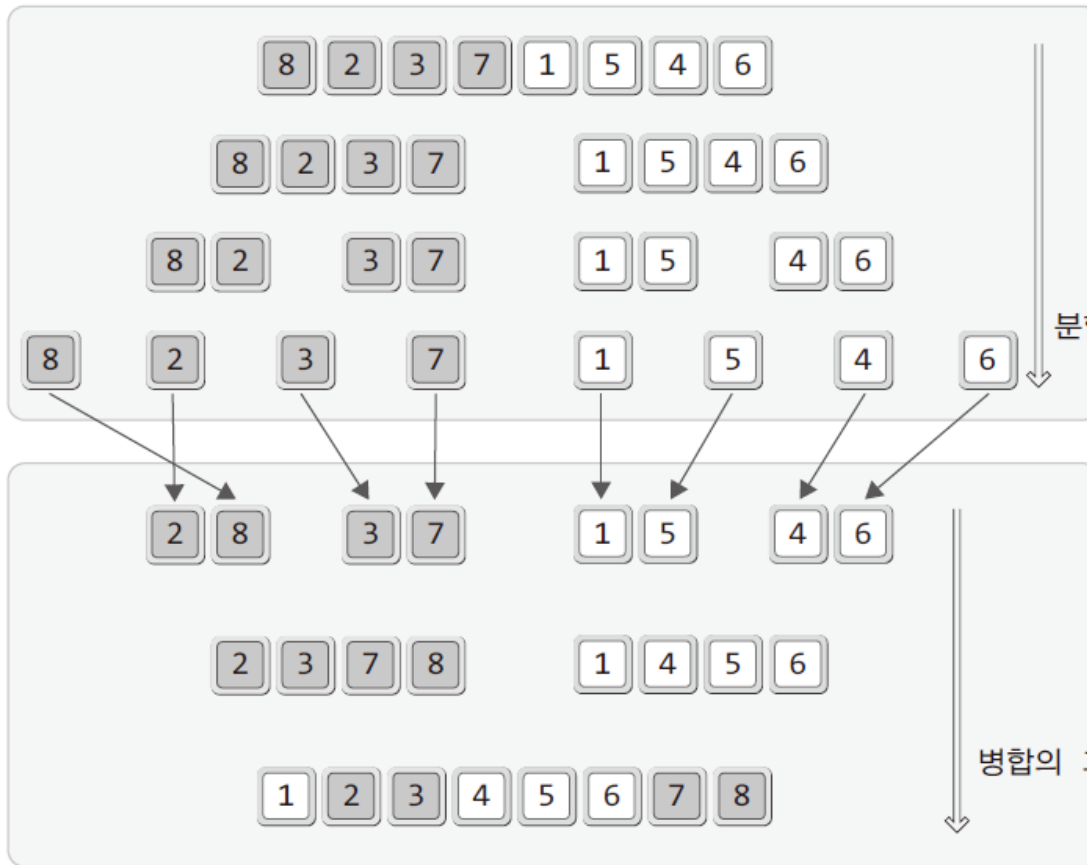
병합 정렬 알고리즘 역시 *DAC*를 기반으로 설계된 알고리즘이다.

병합 정렬: DAC 관점에서의 이해



▶ [그림 10-9: 병합 정렬의 기본 원리]

병합 정렬: 분할의 방법은?



분할의 과정이 재귀적이다!

분할의 과정

별도의 정렬을 진행하지 않아도
될 수준까지 분할을 진행한다!

병합의 과정

분할보다 신경 써야 하는 것이 병합과정이다.
그래서 병합정렬이라 한다.

▶ [그림 10-10: 병합 정렬의 예]

병합 정렬: 재귀적 구현

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

    if(left < right)    // left가 작다는 것은 더 나눌 수 있다는 뜻!
    {
        // 중간지점을 계산한다.
        mid = (left+right) / 2;

        // 둘로 나뉘서 각각을 정렬한다.
        MergeSort(arr, left, mid);    // left~mid에 위치한 데이터 정렬!
        MergeSort(arr, mid+1, right); // mid+1~right에 위치한 데이터 정렬!

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

MergeSort 함수는 둘로 나눌 수 없을 때까지 재귀적으로 호출된다.

병합 정렬: 병합을 위한 함수의 정의

```
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++;  
    }
```

병합 할 두 영역의 데이터를 비교하여
배열 *sortArr*에 담는다!

```
    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];  
    }
```

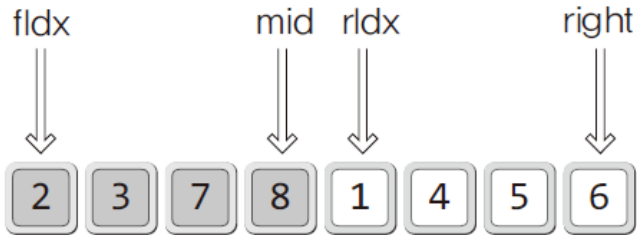
배열의 앞 부분이 *sortArr*로 모두 이동되어서 배열
뒷부분에 남은 데이터를 모두 *sortArr*로 이동!

배열의 뒷 부분이 *sortArr*로 모두 이동되어서 배열
앞부분에 남은 데이터를 모두 *sortArr*로 이동!

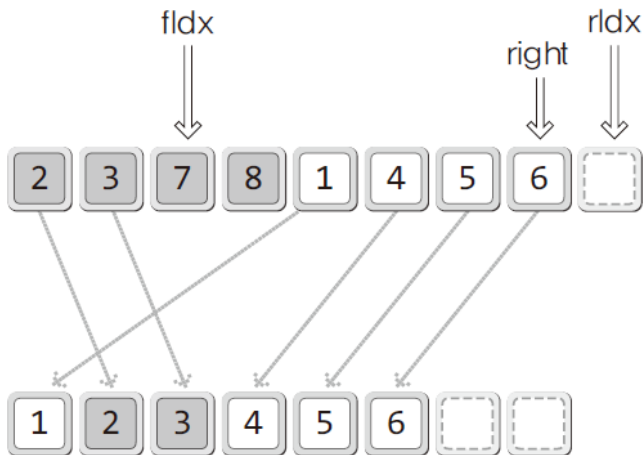
```
    for(i=left; i <= right; i++)  
        arr[i] = sortArr[i];  
    free(sortArr);  
}
```

병합 결과를 옮겨 담는다!

병합 정렬: 병합 함수의 코드 설명



↓ 1차 병합의 결과



```
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++;
    }
```

... 1차 병합을 진행하는 부분

병합 정렬: 성능평가(내용 비교)

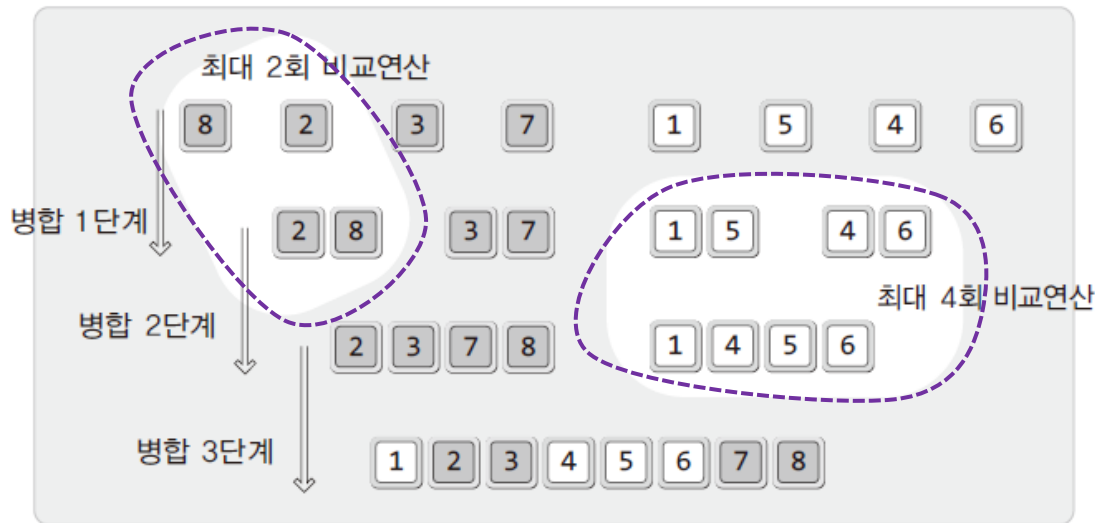
데이터의 비교 및 데이터의 이동은
MergeTwoArea 함수를 중심으로 진행!

따라서 병합 정렬의 성능은 MergeTwoArea
함수를 기준으로 계산!



MergeTwoArea의
핵심 루틴

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



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

병합 정렬: 성능평가(내용 비교)

“정렬의 대상인 데이터의 수가 n 개 일 때, 각 병합의 단계마다 최대 n 번의 비교연산이 진행된다.”



따라서 데이터 수에 대한 비교 연산의 횟수는

$$n \log_2 n$$



따라서 비교 연산에 대한 빅-오는

$$O(n \log_2 n)$$



병합 정렬: 성능평가(이동)

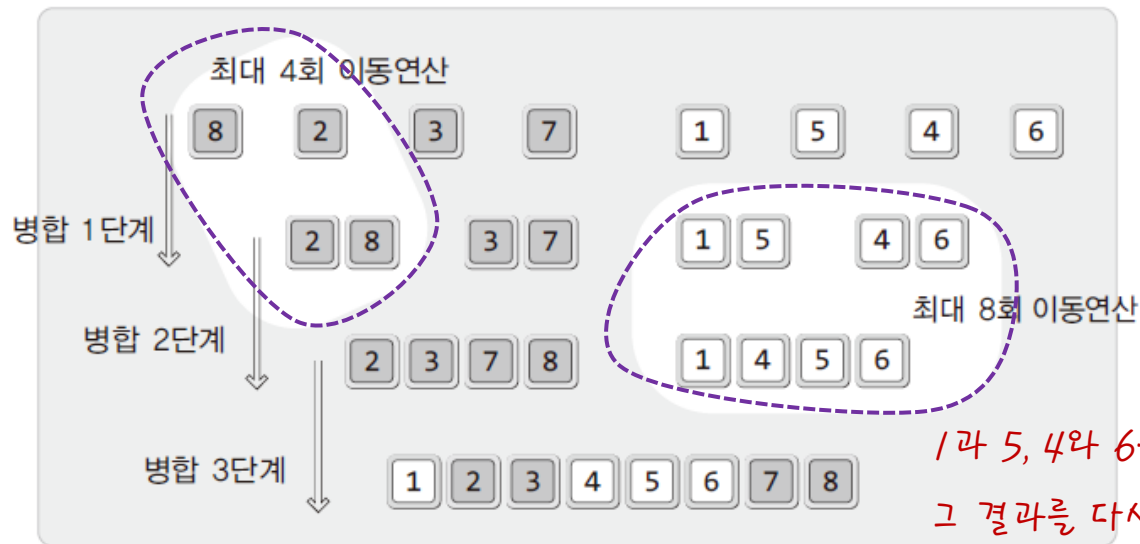
- 임시 배열에 데이터를 병합하는 과정에서 한 번!
- 임시 배열에 저장된 데이터 전부를 원위치로 옮기는 과정에서 한 번!

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

최악, 최선 상관 없이!

8과 2를 정렬해서 옮기고! 2회

그 결과를 다시 옮기고! 2회



1과 5, 4와 6을 정렬해서 옮기고! 4회
그 결과를 다시 옮기고! 4회

퀵 정렬: 이해(1단계: 초기화)

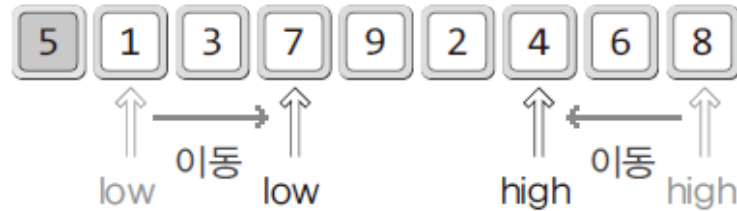


퀵 정렬을 위해서는 총 5개의 변수
*left, right, pivot, low, high*를 선언해야 한다.

- **left** 정렬대상의 가장 왼쪽 지점을 가리키는 이름
- **right** 정렬대상의 가장 오른쪽 지점을 가리키는 이름
- **pivot** 피벗이라 발음하고 중심점, 중심축의 의미를 담고 있다.
- **low** 피벗을 제외한 가장 왼쪽에 위치한 지점을 가리키는 이름
- **high** 피벗을 제외한 가장 오른쪽에 위치한 지점을 가리키는 이름

가장 왼쪽에 위치한 데이터를 피벗으로 결정하기로 하자! 물론 피벗은 달리 결정할 수 있다!

퀵 정렬: 이해(2단계: low와 high의 이동)



Low와 high의 이동은 완전 별개이다!

low와 high의 첫번째 정거장

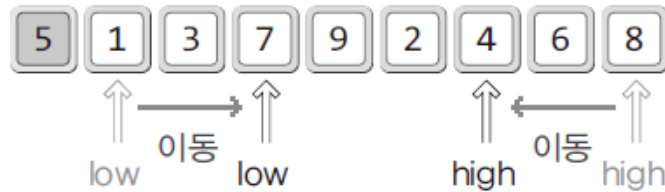
- low의 오른쪽 방향 이동 피벗보다 큰 값을 만날 때까지
- high의 왼쪽 방향 이동 피벗보다 작은 값을 만날 때까지



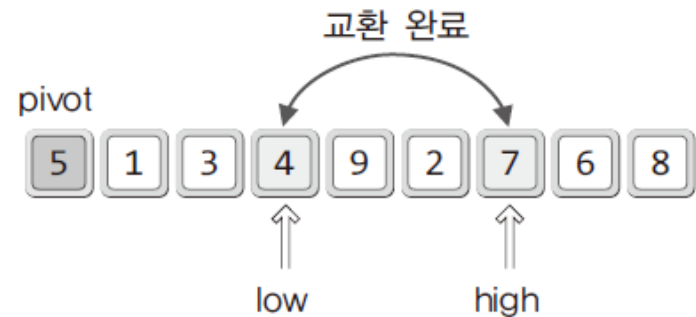
일반화 해서 표현하면

- low의 오른쪽 방향 이동
 피벗보다 정렬의 우선순위가 낮은 데이터를 만날 때까지
- high의 왼쪽 방향 이동
 피벗보다 정렬의 우선순위가 높은 데이터를 만날 때까지

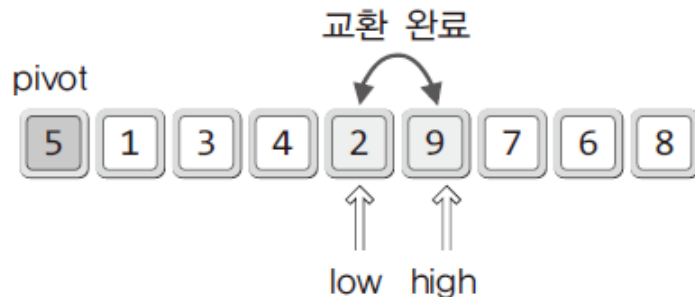
퀵 정렬: 이해(3단계: low와 high의 교환)



low와 high의 데이터 교환



교환 후 이동을 계속,
그리고 또 교환!



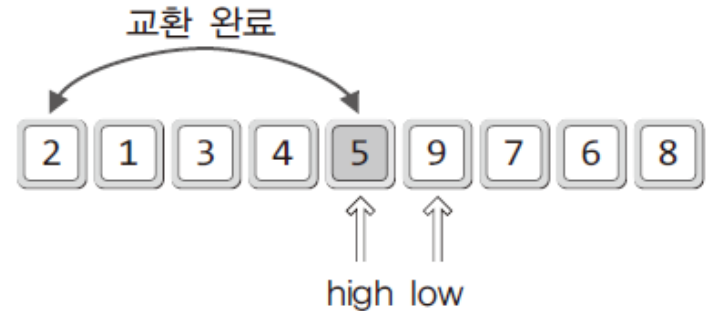
이동을 계속,
high와 low가 역전 될때까지



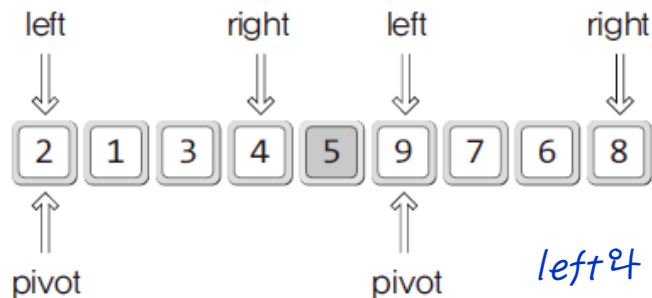
퀵 정렬: 이해(4단계: 피벗의 이동)



high와 low가 역전되면,
피벗과 high의 데이터 교환



두 개의 영역으로 나누어 반복 실행



여기까지가 /회전 완료!

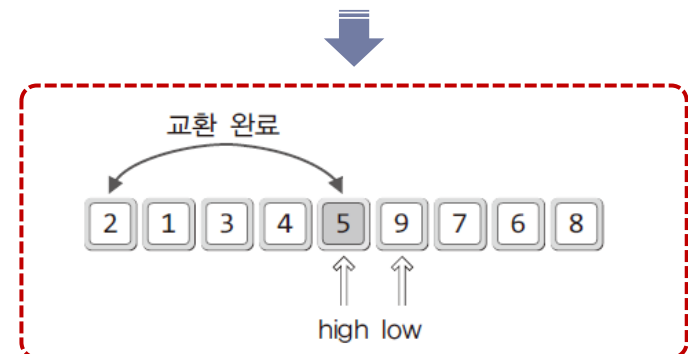
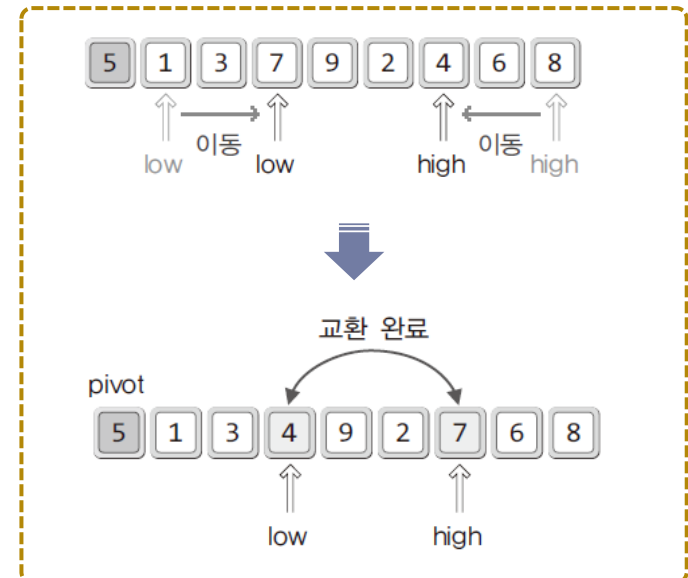
left와 right가 다음 관계에 놓일 때까지 반복!

$left > right$

퀵 정렬: 구현(핵심 알고리즘)

```
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
    {
        // 피벗보다 큰 값을 찾는 과정
        while(pivot > arr[low])
            low++;            // low를 오른쪽으로 이동
        // 피벗보다 작은 값을 찾는 과정
        while(pivot < arr[high])
            high--;           // high를 왼쪽으로 이동
        // 교차되지 않은 상태라면 Swap 실행
        if(low <= high)
            Swap(arr, low, high);
    }
    Swap(arr, left, high);    // 피벗과 high가 가리키는 대상 교환
    return high;              // 옮겨진 피벗의 위치정보 반환
}
```

while문 내에서 진행되는 일의 내용



퀵 정렬: 구현(재귀적 완성과 수정사항)

```
void QuickSort(int arr[], int left, int right)
{
    if(left <= right)
    {
        int pivot = Partition(arr, left, right);    // 둘로 나뉘서
        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)
    {
        while(pivot > arr[low])
            low++;
        while(pivot < arr[high])
            high--;
        ....
    }
    ....
}
```

while(pivot >= arr[low] && low <= right)

정렬 범위 넘지 않기 위해!

// 항상 '참'일 수밖에 없는 상황!

// 문제가 되는 지점!

// 문제가 되는 지점!

while(pivot <= arr[high] && high >= (left+1))

정렬 범위 넘지 않기 위해!

피벗 low high
↓ ↓ ↓
3, 3, 3 을 정렬하는 상황을 가정하자!

퀵 정렬: 구현(피벗 선택에 대한 논의)

pivot

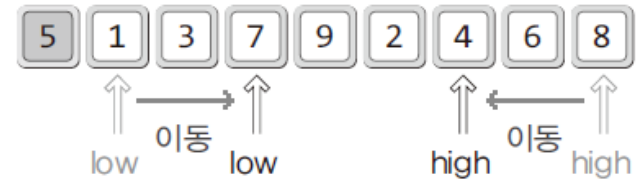


이렇듯 정렬이 되어 있고 피벗이 정렬 대상의 한 쪽 끝에 치우치는 경우 최악의 경우를 보인다.

pivot



이렇듯 데이터가 불규칙적으로 나열되어 있고 피벗이 중간에 해당 하는 값에 가깝게 선택이 될 수록 최상의 경우를 보인다.



정렬과정에서 선택되는 피벗의 수가 적을 수록 최상의 경우에 해당이 된다!

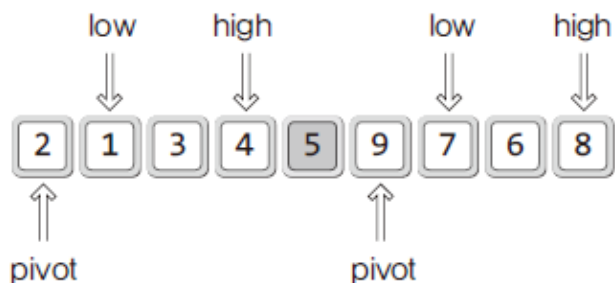
결론은 피벗이 가급적 중간에 해당하는 값이 선택되어야 좋은 성능을 보인다는 것!

퀵 정렬: 성능평가(최선의 경우)



모든 데이터가 피벗과의 데이터 비교를 진행한다.
따라서 이 경우 약 n 번의 비교 연산이 진행된다.

▶ [그림 10-24: 비교연산 횟수의 힌트1]



마찬가지로 약 n 번의 비교 연산이 진행된다.

▶ [그림 10-25: 비교연산 횟수의 힌트2]

- 31개의 데이터는 15개씩 둘로 나뉘어 총 2 조각이 된다.
- 이어서 각각 7개씩 둘로 나뉘어 총 4 조각이 된다.
- 이어서 각각 3개씩 둘로 나뉘어 총 8 조각이 된다.
- 이어서 각각 1개씩 둘로 나뉘어 총 16 조각이 된다.

1차 나뉘

2차 나뉘

3차 나뉘

4차 나뉘

$$k = \log_2 n$$

$$O(n \log_2 n)$$

최선의 경우 빅-오

퀵 정렬: 성능평가(최악의 경우)

$O(n^2)$

pivot



- 둘로 나뉘는 횟수가 약 $n!$
- 매 단계별 비교 연산의 횟수 약 $n!$

중간에 가까운 값으로 박-오를 선택하려는 노력을 조금만 하더라도 퀵 정렬은 최악의 경우를 만들지 않는다. 따라서 퀵 정렬의 성능은 최상의 경우를 근거로 이야기 한다.

퀵 정렬은 $O(n \log_2 n)$ 의 성능을 보이는 정렬 알고리즘 중에서 평균적으로 가장 좋은 성능을 보이는 알고리즘이다. 다른 알고리즘에 비해서 데이터의 이동이 적고 별도의 메모리 공간을 요구하지도 않는데 그 이유가 있다.

기수 정렬: 특징과 적용 범위

기수 정렬의 특징

- 정렬순서의 앞서고 뒤섬을 비교하지 않는다.
- 정렬 알고리즘의 한계로 알려진 $O(n \log_2 n)$ 을 뛰어 넘을 수 있다.
- 적용할 수 있는 대상이 매우 제한적이다. 길이가 동일한 데이터들의 정렬에 용이하다!

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

기수 정렬 OK!

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

기수 정렬 OK!

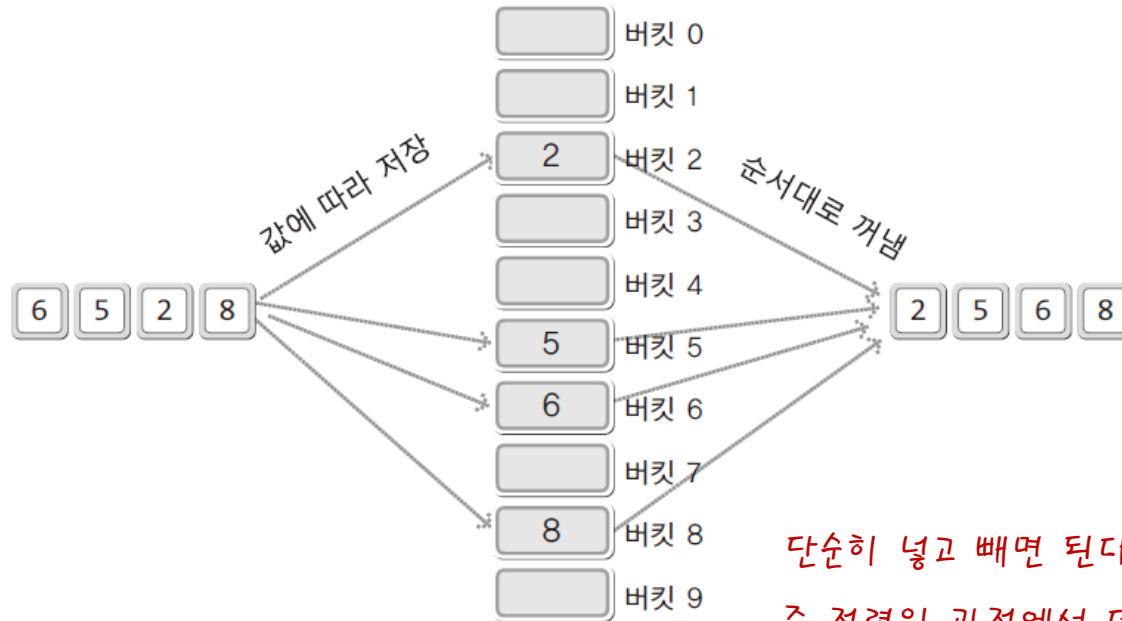
"배열에 저장된 21, -9, 125, 8, -136, 45를 오름차순으로 정렬하라!"

기수 정렬 NO!

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

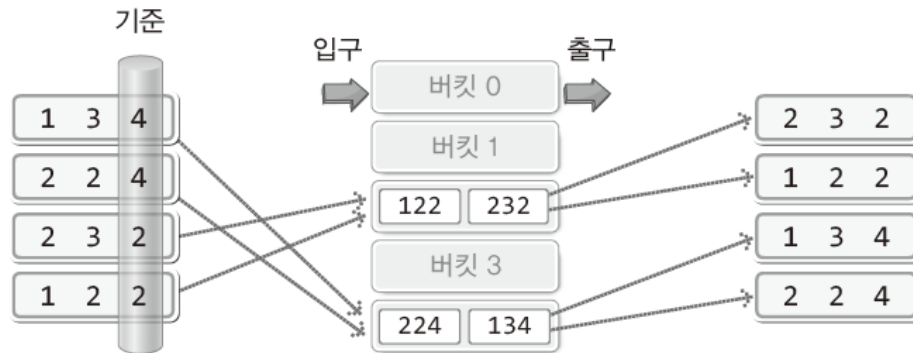
기수 정렬 NO!

기수 정렬: 정렬의 원리

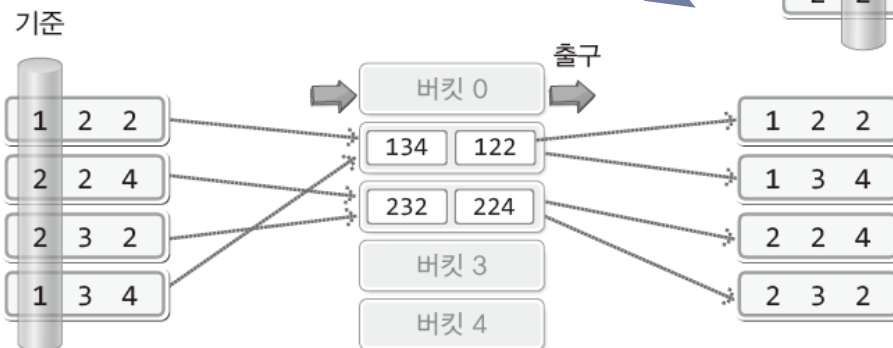
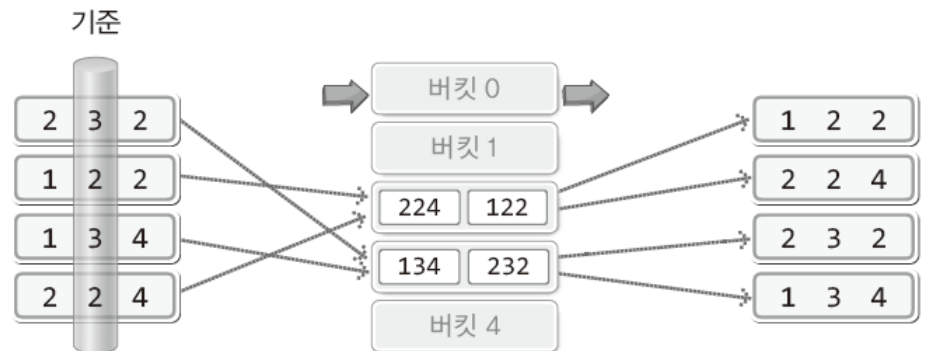


- 기수(radix): 주어진 데이터를 구성하는 기본 요소(기호)
- 버킷(bucket): 기수의 수에 해당하는 만큼의 버킷을 활용한다.

기수 정렬: LSD



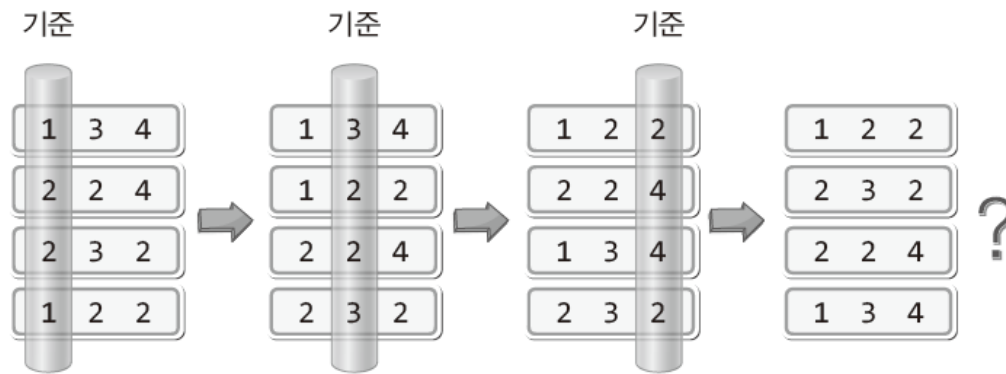
List Significant Digit을 시작으로 정렬을
진행하는 방식!



이렇듯 마지막까지 진행을 해야
값의 우선순위대로 정렬이 완성된다.

기수 정렬: MSD(Most Significant Digit)

MSD는 정렬의 기준 선정 방향이 LSD와 반대이다! 그렇다면 방향성에서만 차이를 보일까?



정렬의 기준 선정 방향만을 바꾸어서 기수 정렬을 진행한 결과!

MSD 방식은 점진적으로 정렬이 완성되어 가는 방식이다!

따라서 중간 중간에 정렬이 완료된 데이터는 더 이상의 정렬과정을 진행하지 않아야 한다.

기수 정렬: LSD 기준 구현

MSD와 LSD의 빅-오는 같다. 하지만 LSD의 구현이 더 용이하다! 따라서 LSD를 기준으로 기수 정렬을 구현하는 것이 일반적이다.

- | | |
|-------------------------|--------------------------|
| · NUM으로부터 첫 번째 자리 숫자 추출 | $\text{NUM} / 1 \% 10$ |
| · NUM으로부터 두 번째 자리 숫자 추출 | $\text{NUM} / 10 \% 10$ |
| · NUM으로부터 세 번째 자리 숫자 추출 | $\text{NUM} / 100 \% 10$ |

양의 정수라면 길이에 상관 없이 정렬 대상에 포함시키기 위한 간단한 알고리즘

LSD 방식에서는 모든 데이터가 정렬의 과정을 동일하게 거치도록 구현한다. 반면 MSD 방식에서는 선별해서 거치도록 구현해야 한다. 따라서 LSD 방식의 구현이 더 용이할 뿐만 아니라 생각과 달리 성능도 MSD에 비교해서 떨어지지 않는다.

기수 정렬: code 구현

```
#include "ListBaseQueue.h"

#define BUCKET_NUM    10

void RadixSort(int arr[], int num, int maxLen)
{
    Queue buckets[BUCKET_NUM];
    int bi;
    int pos;
    int di;
    int divfac = 1;
    int radix;

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

    // 가장 긴 데이터의 길이만큼 반복
    for(pos=0; pos<maxLen; pos++)
    {
        . . . .
    }
}
```

```
// 정렬대상의 수만큼 반복
for(di=0; di<num; di++)
{
    // N번째 자리의 숫자 추출
    radix = (arr[di] / divfac) % 10;

    // 추출한 숫자를 근거로 버킷에 데이터 저장
    Enqueue(&buckets[radix], arr[di]);
}

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

// N번째 자리의 숫자 추출을 위한 피제수의 증가
divfac *= 10;
```

RadixSort 함수의 호출의 예

```
int arr[7] = {13, 212, 14, 7141, 10987, 6, 15};
int len = sizeof(arr) / sizeof(int);
RadixSort(arr, len, 5);
```

기수 정렬: 성능평가

```
void RadixSort(int arr[], int num, int maxLen)
```

```
{
```

```
    . . . . .
```

```
    // 가장 긴 데이터의 길이만큼 반복
```

```
    for(pos=0; pos<maxLen; pos++)
```

```
    {
```

```
        // 정렬대상의 수만큼 버킷에 데이터 삽입
```

```
        for(di=0; di<num; di++)
```

```
        {
```

```
            // 버킷으로의 데이터 삽입 진행
```

```
        }
```

```
        // 정렬대상의 수만큼 버킷으로부터 데이터 추출
```

```
        for(bi=0, di=0; bi<BUCKET_NUM; bi++)
```

```
        {
```

```
            // 버킷으로부터의 데이터 추출 진행
```

```
        }
```

```
    . . . . .
```

```
}
```

```
}
```

버킷으로의 데이터 삽입과 추출을 근거로 빅-오를 결정!

삽입

삽입과 추출 연산을 한 쌍으로 묶으면,
이 한 쌍의 연산 수행 횟수는 다음과 같다.

$\text{maxLen} \times \text{num}$

추출

따라서 정렬대상의 수가 n 이고, 모든 정렬대상의 길이를 l 이라 할때 시간 복잡도에 대한 기수 정렬의 빅-오는 다음과 같다.

$O(ln)$

수고하셨습니다~



Chapter 10에 대한 강의를 마칩니다!

