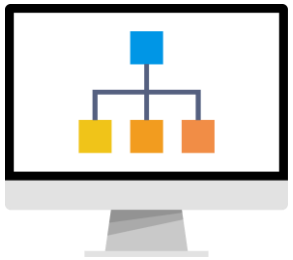


Data Structure & Algorithm
자료구조 및 알고리즘

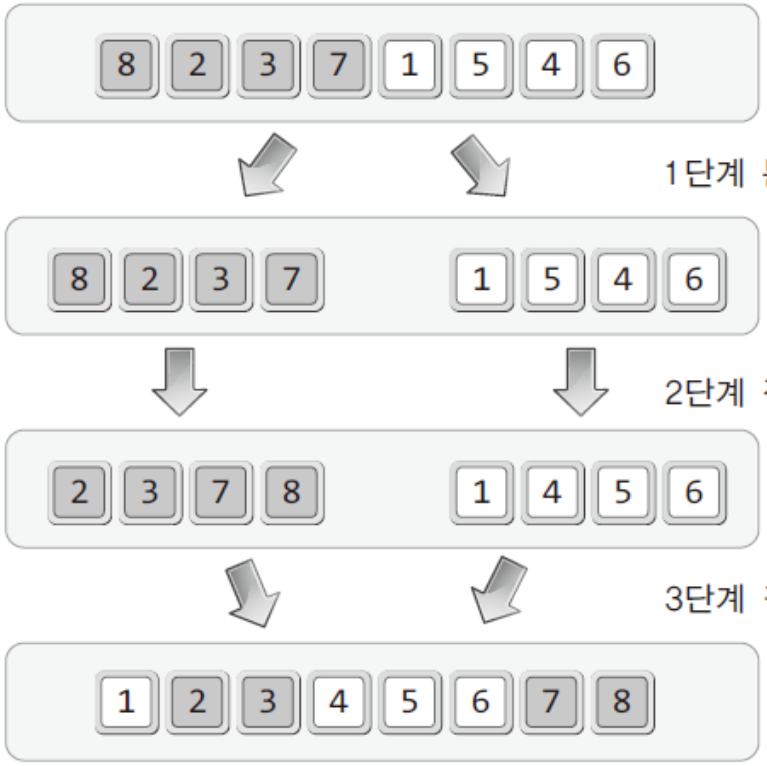
18. 정렬 (Sorting, Part 2)

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 DAC를 기반으로 설계된 알고리즘이다.

병합 정렬: DAC 관점에서의 이해



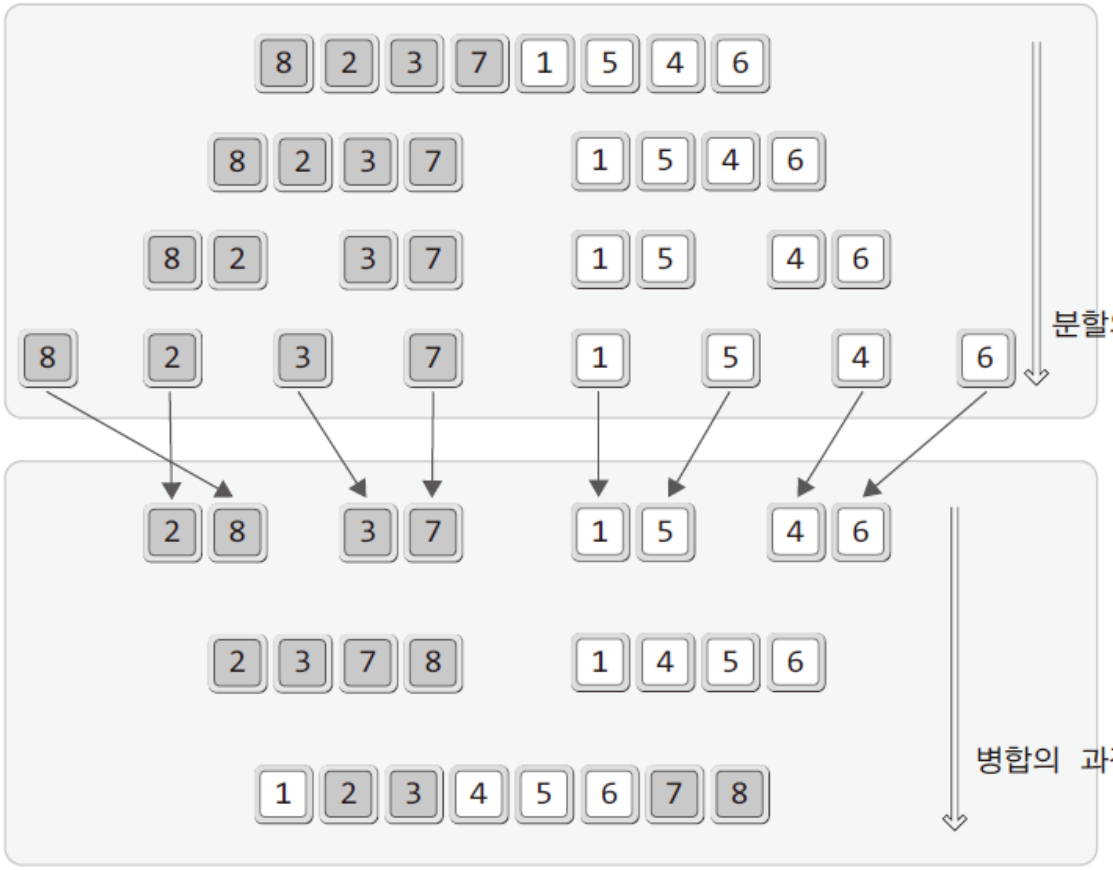
정렬하기 좋은 상태로 분할을 진행해 나간다!

정렬하기 좋은 상태에서 정렬을 하고!

정렬이 완료된 조각들을 결합하여 정렬을 끝낸다!

▶ [그림 10-9: 병합 정렬의 기본 원리]

병합 정렬: 분할의 방법은?



▶ [그림 10-10: 병합 정렬의 예]

분할의 과정이 재귀적이다!

별도의 정렬을 진행하지 않아도 될 수준까지 분할을 진행한다!

분할보다 신경 써야 하는 것이 병합과정이다.
그래서 병합정렬이라 한다.

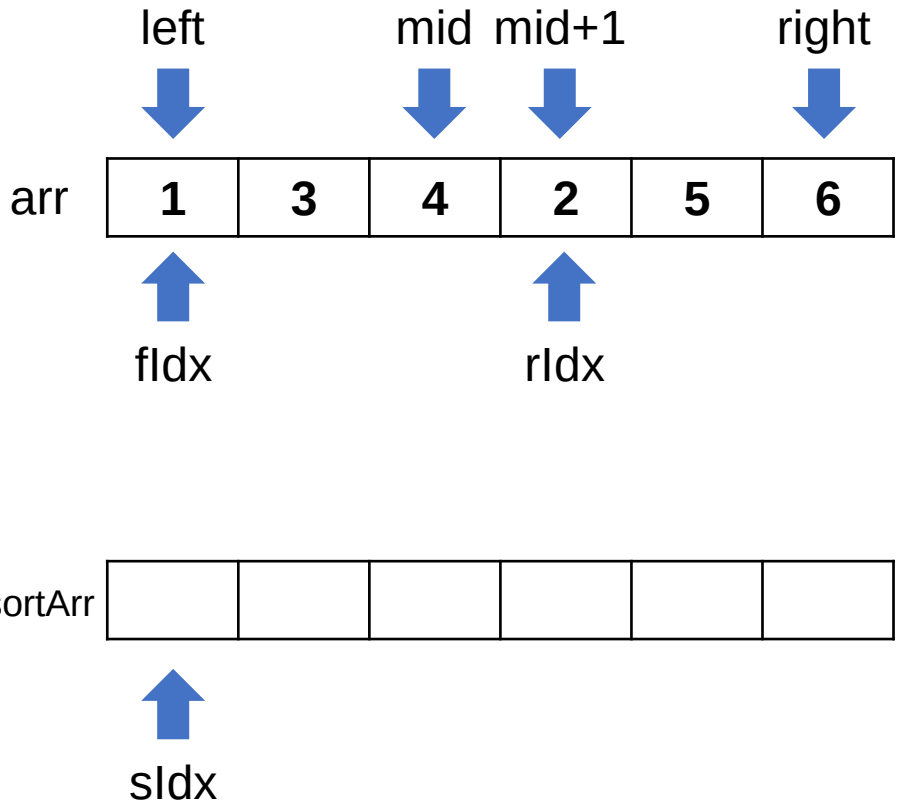
병합 정렬: 재귀적 구현

```
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 나눠서 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.

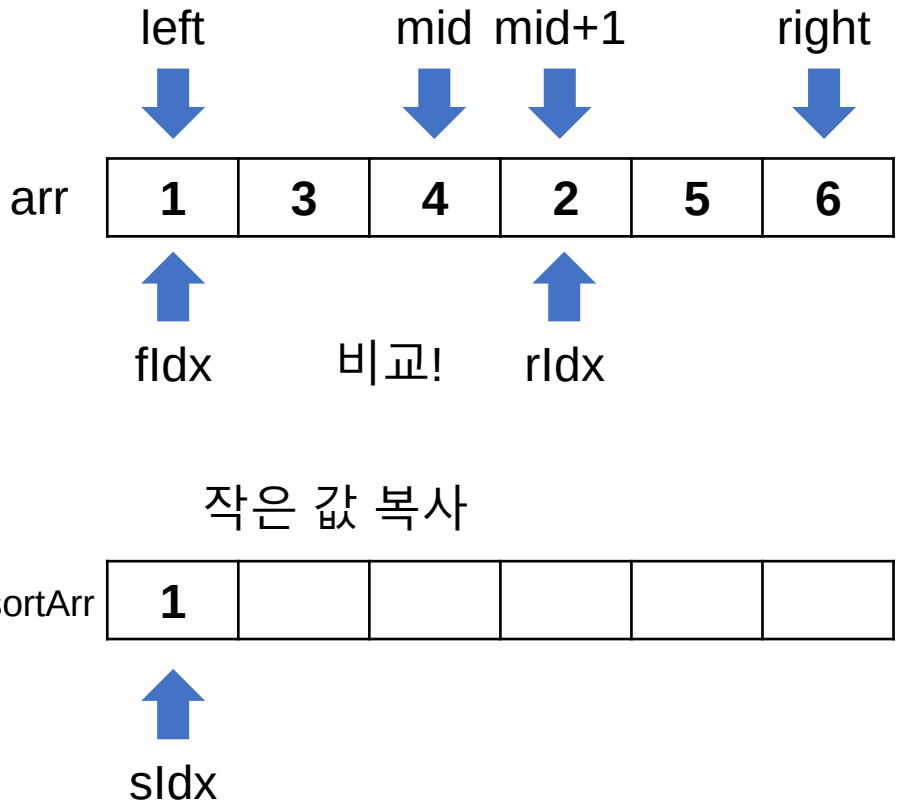
병합 정렬: 병합을 위한 함수의 정의

```
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++;
    }

    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);
}
```



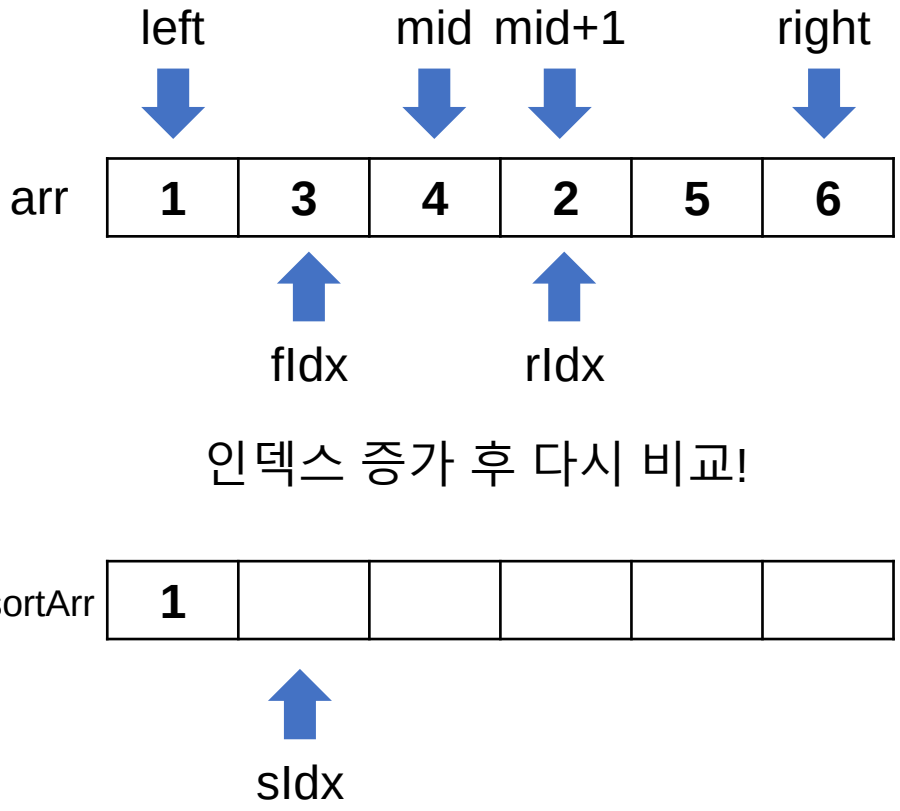
병합 정렬: 병합을 위한 함수의 정의

```
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++;
    }

    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);
}
```



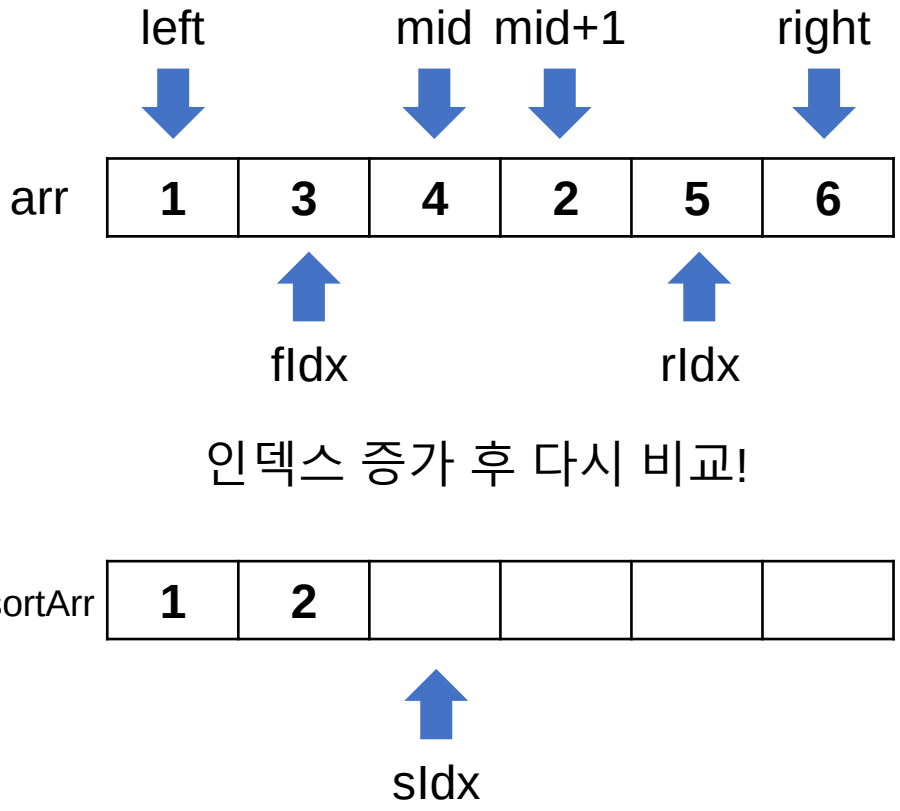
병합 정렬: 병합을 위한 함수의 정의

```
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++;
    }

    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);
}
```



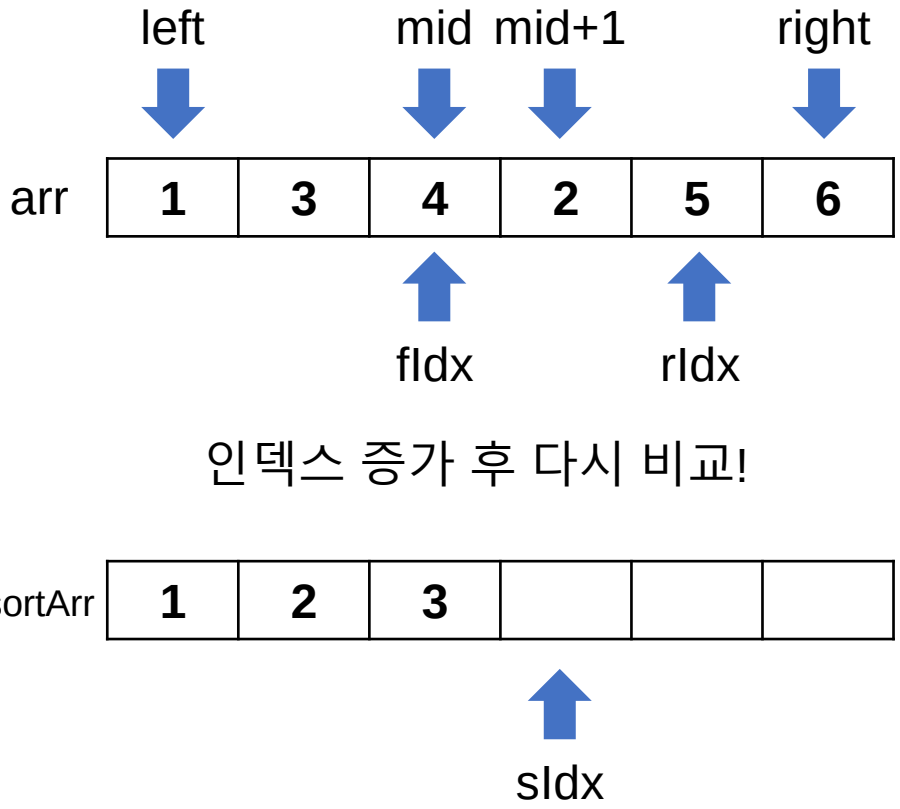
병합 정렬: 병합을 위한 함수의 정의

```
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++;
    }

    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);
}
```



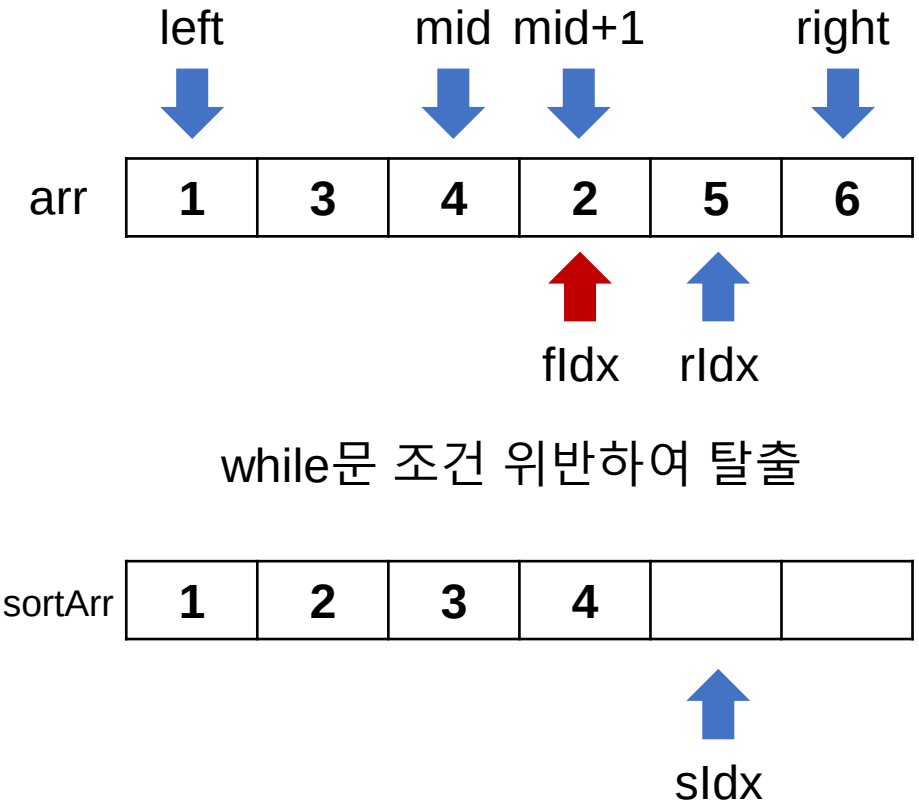
병합 정렬: 병합을 위한 함수의 정의

```
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++;
    }

    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 정렬: 병합을 위한 함수의 정의

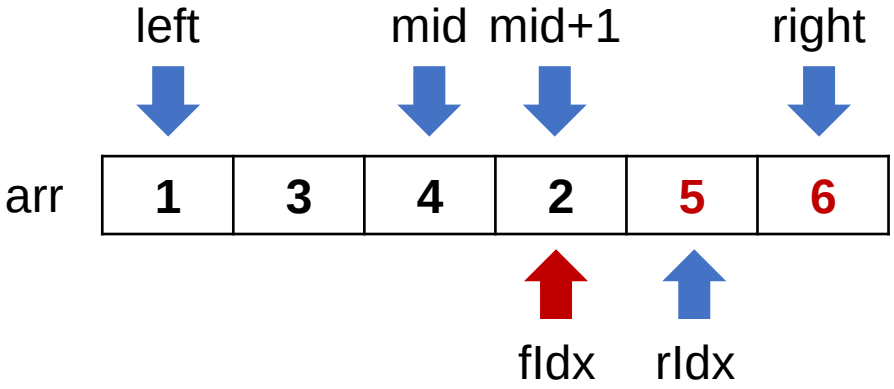
```
void MergeTwoArea(int arr[], int left, int mid, int right)
{
    int fIdx = left;     int rIdx = mid+1;     int i;
    int * sortArr = (int*)malloc(sizeof(int)*(right+1));
    int sIdx = left;

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

    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);
}
```



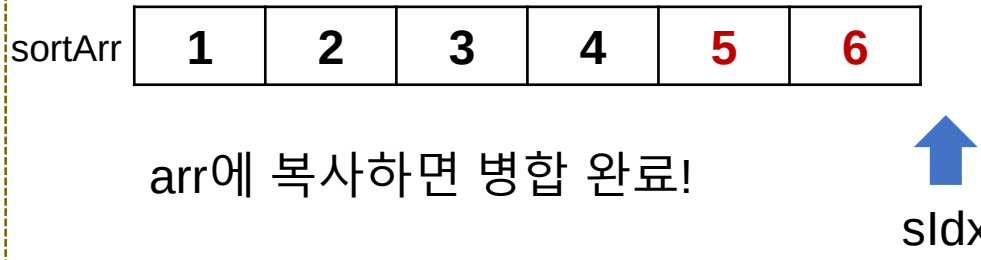
병합 정렬: 병합을 위한 함수의 정의

```
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++;
    }

    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);
}
```

현재 배열의 왼쪽 반은 복사되었으나
배열의 오른쪽 반의 일부가 남은 상태

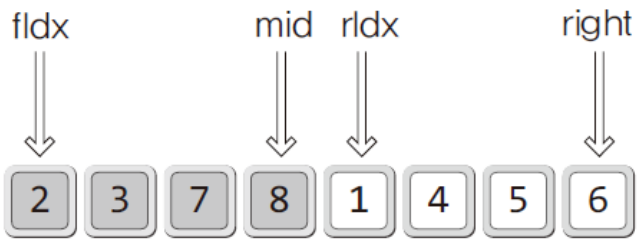


남은 데이터를 다 옮겨준다.

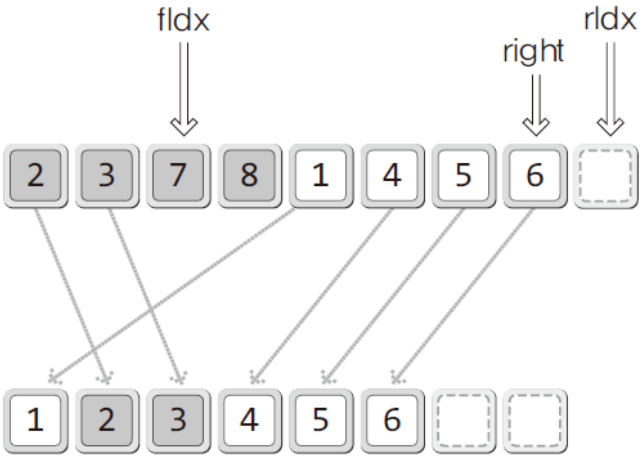


arr에 복사하면 병합 완료!

병합 정렬: 병합 함수의 코드 설명



1차 병합의 결과

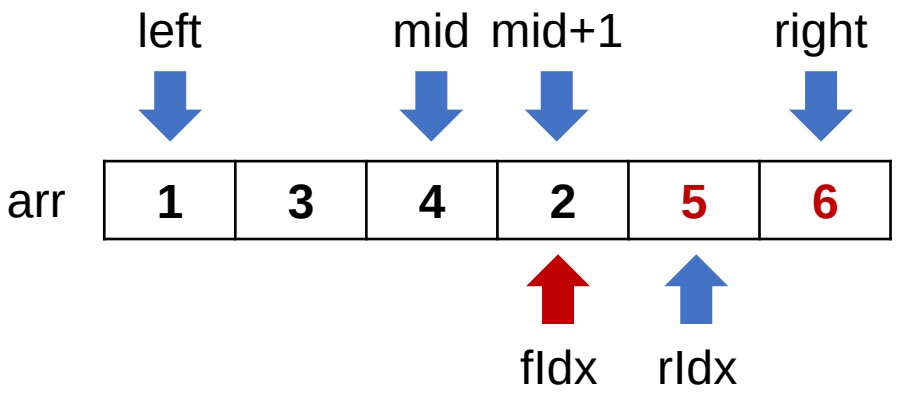



```
void MergeTwoArea(int arr[], int left, int mid, int right)
{
    int fIdx = left;     int rIdx = mid+1;     int i;
    int * sortArr = (int*)malloc(sizeof(int)*(right+1));
    int sIdx = left;

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

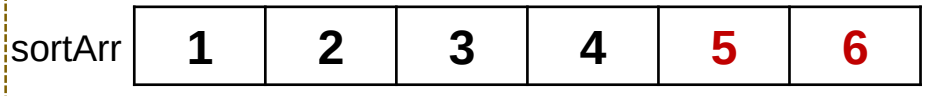
1차 병합을 진행하는 부분

병합 정렬: 병합 함수의 코드 설명

```
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);
}
```



남은 데이터를 다 옮겨준다.



arr에 복사하면 병합 완료!

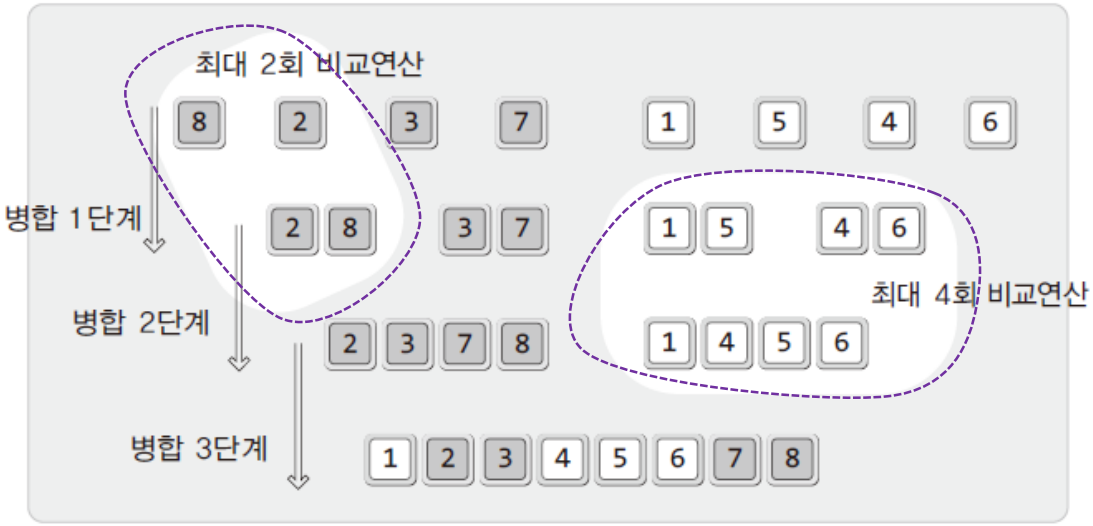
병합 정렬: 성능평가(내용 비교)

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 n개 일 때, 각 병합의 단계마다 최대 n번의 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)

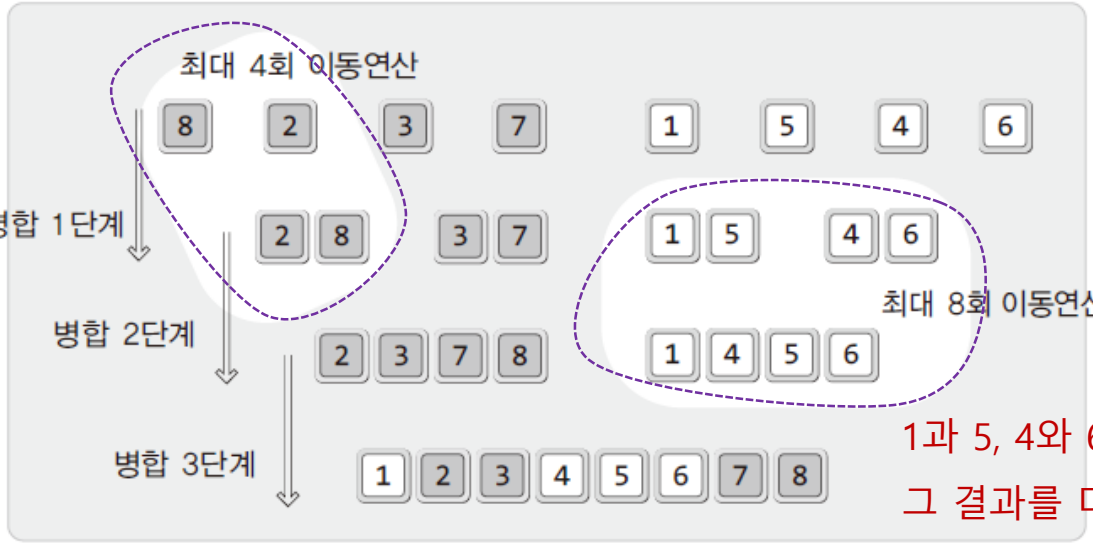
- 임시 배열에 데이터를 병합하는 과정에서 한 번!
- 임시 배열에 저장된 데이터 전부를 원위치로 옮기는 과정에서 한 번!

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

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

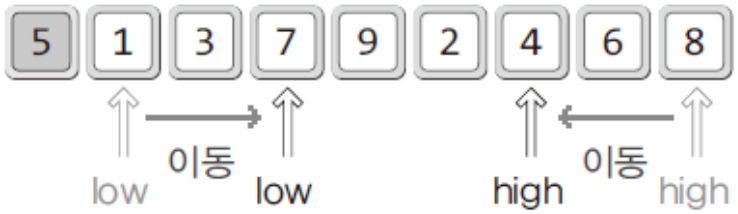
퀵 정렬: 이해(1단계: 초기화)

퀵 정렬을 위해서는 총 5개의 변수
left, right, pivot, low, high를 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!

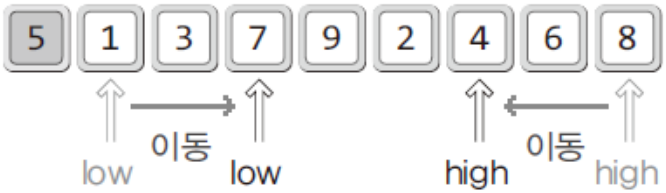
low와 high의 첫번째 정거장

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

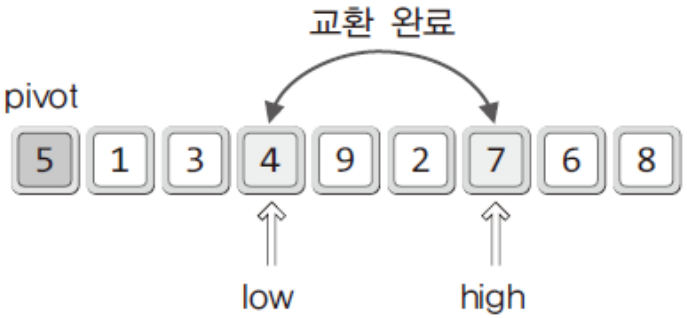
일반화 해서 표현하면

- 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)

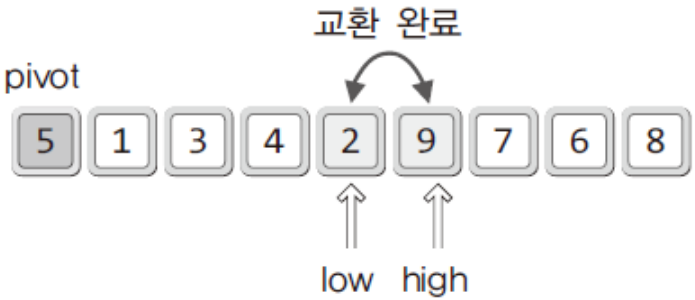


low와 high의 데이터 교환



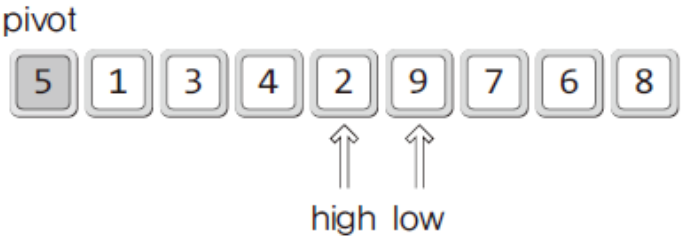
교환 후 이동을 계속,

그리고 또 교환!

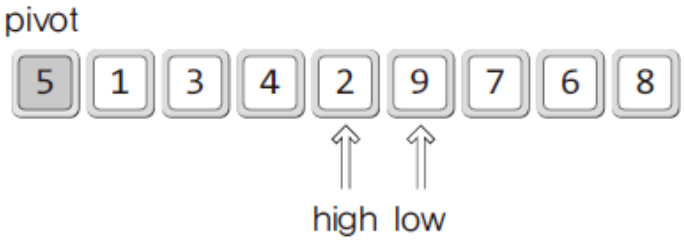


이동을 계속,

high와 low가 역전 될때까지

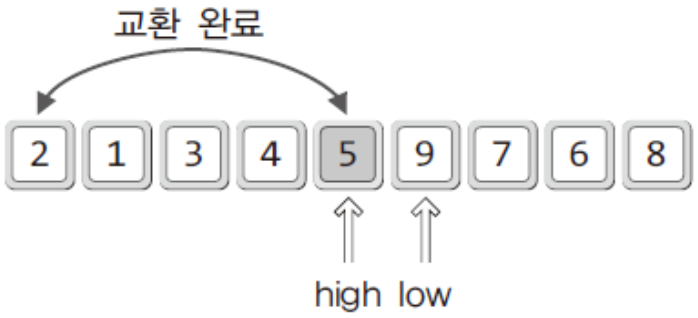


퀵 정렬: 이해(4단계: 피벗의 이동)



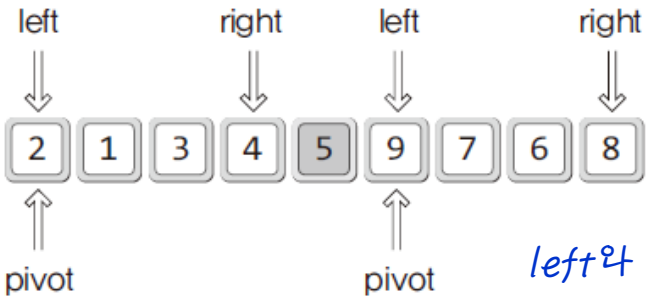
high와 low가 역전되면,

피벗과 high의 데이터 교환



여기까지가 1회전 완료!

두 개의 영역으로 나누어 반복 실행



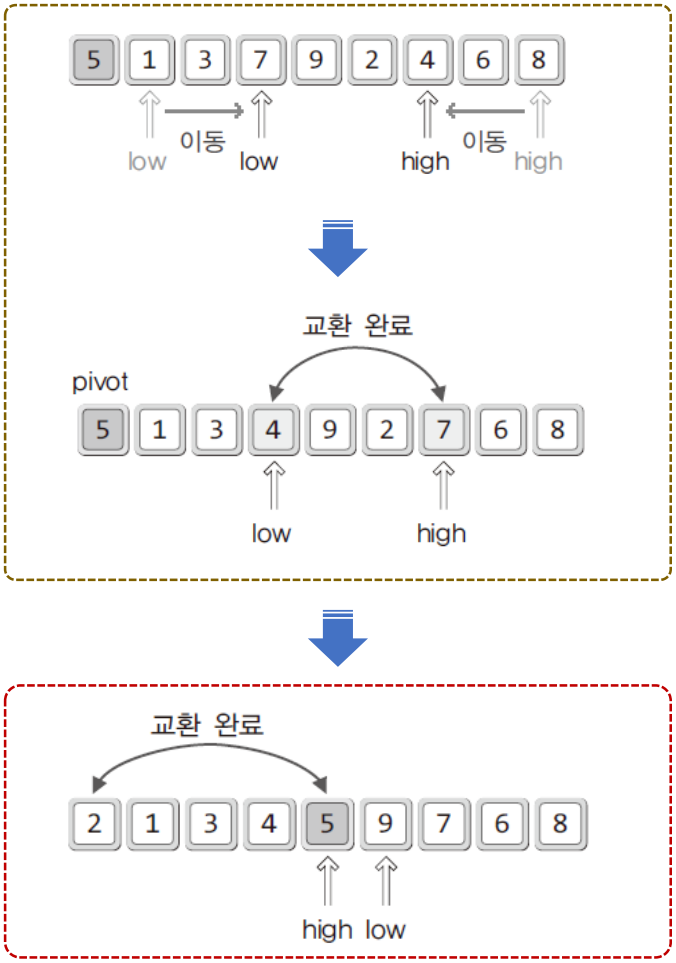
left와 right가 다음 관계에 놓일 때까지 반복!

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 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 '참'일 수밖에 없는 상황!
    {
        while(pivot > arr[low])       // 문제가 되는 지점!
            low++;
        while(pivot < arr[high])      // 문제가 되는 지점!
            high--;
        ....
    }
    ....
}
```

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

정렬 범위 넘지 않기 위해!

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

정렬 범위 넘지 않기 위해!

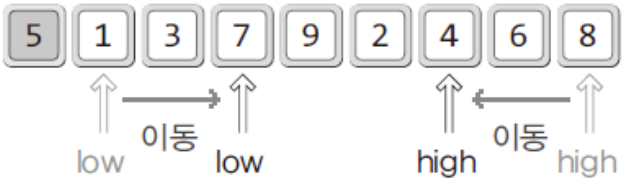
피벗 low high

3, 3, 3 을 정렬하는 상황을 가정하자!

퀵 정렬: 구현
(피벗 선택에 대한 논의)

pivot

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



pivot

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

정렬과정에서 선택되는 피벗의 수가 적을수록 최상의 경우에 해당이 된다!

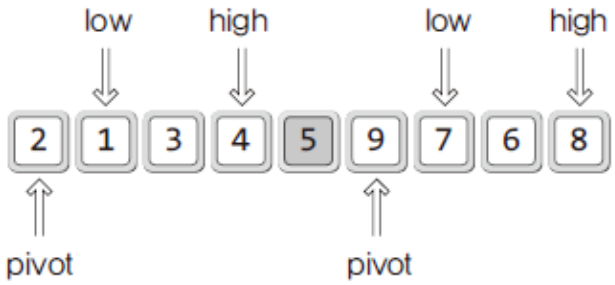
결론은 피벗이 가급적 중간에 해당하는 값이 선택되어야 좋은 성능을 보인다는 것!

퀵 정렬: 성능평가(최선의 경우)

모든 데이터가 피벗과의 데이터 비교를 진행한다.

따라서 이 경우 약 n번의 비교 연산이 진행된다.

▶ [그림 10-24: 비교연산 횟수의 힌트1]



마찬가지로 약 n번의 비교 연산이 진행된다.

▶ [그림 10-25: 비교연산 횟수의 힌트2]

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

$$k = log_2 n$$

$$O(n \log_2 n)$$

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 빅-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

· 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!

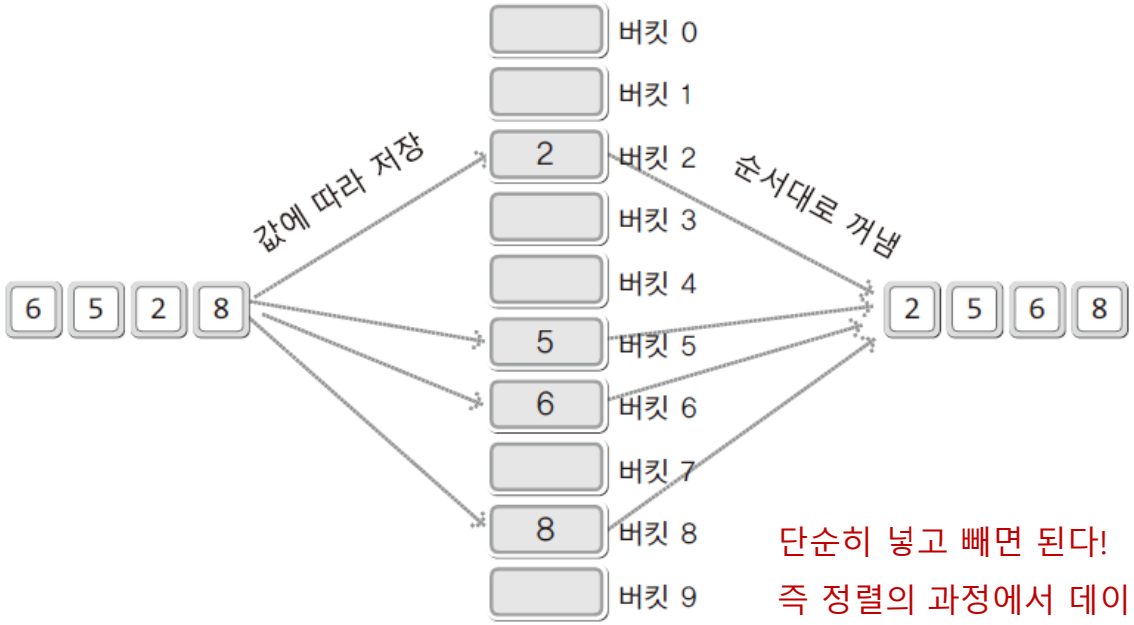
"배열에 저장된 21, -9, 125, 8, -136, 45를 오름차순으로 정렬하라!"

기수 정렬 NO!

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

기수 정렬 NO!

기수 정렬: 정렬의 원리

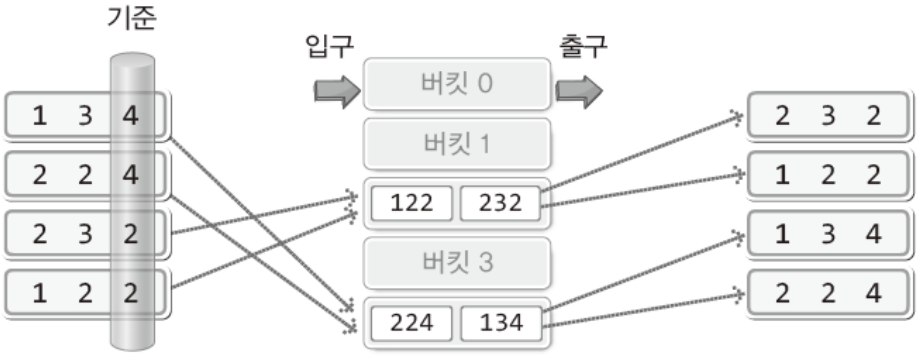


단순히 넣고 빼면 된다!

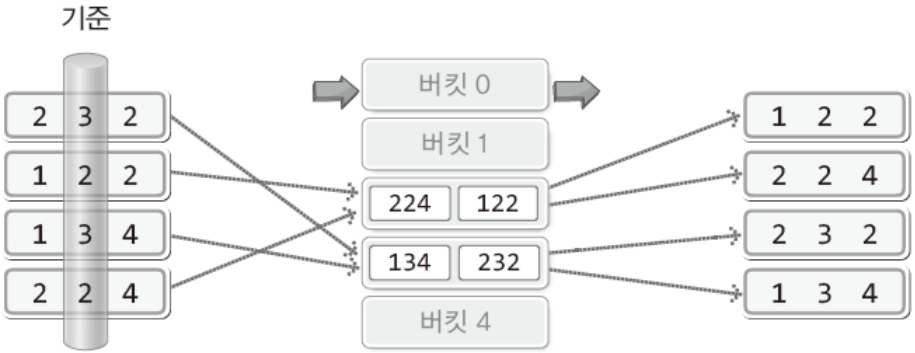
즉 정렬의 과정에서 데이터간 비교가 발생하지 않는다!

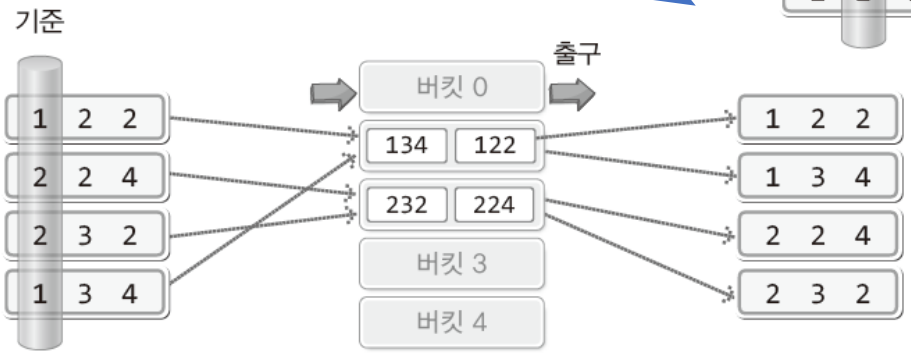
- 기수(radix): 주어진 데이터를 구성하는 기본 요소(기호)
- 버킷(bucket): 기수의 수에 해당하는 만큼의 버킷을 활용한다.

기수 정렬: LSD



Least Significant Digit을 시작으로 정렬을 진행하는 방식!

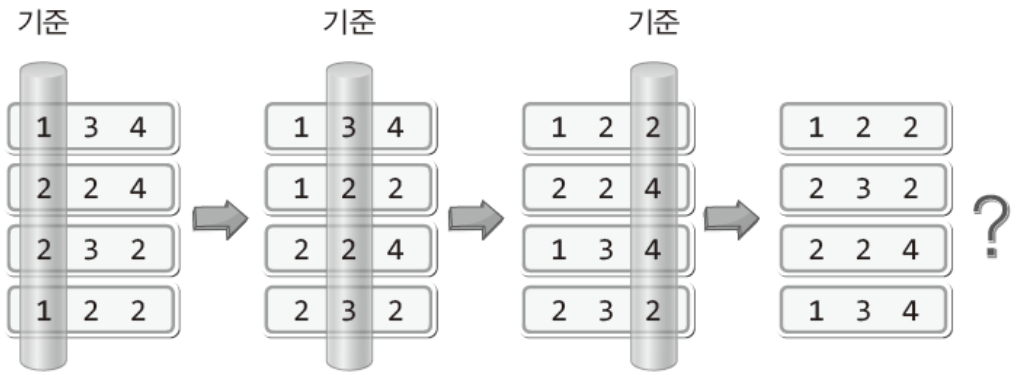




이렇듯 마지막까지 진행을 해야 값의 우선순위대로 정렬이 완성된다.

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!

모든 데이터에 대해서 비교를 하면 정렬이 되지 않는다.

높은 자릿수의 숫자가 같은 수들만 따로 이후 자릿수를 비교해야 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 NUM / 1 % 10
- NUM으로부터 두 번째 자리 숫자 추출 NUM / 10 % 10
- NUM으로부터 세 번째 자리 숫자 추출 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.

maxLen x num

추출

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

$O(ln)$

요약

- 병합 정렬: 배열을 분할할 수 있을 때까지 분할하고, 정렬 순서를 유지하면서 병합
- 퀵 정렬: 선택된 피벗을 정렬 순서상 제자리에 넣고 왼쪽 배열과 오른쪽 배열을 재귀적으로 정렬
- 기수 정렬: 낮은 자리수부터 해당하는 버킷에 넣고 정렬 순서상 앞서는 버킷부터 데이터를 빼어서 나열
- 버블 정렬, 선택 정렬, 삽입 정렬, 힙 정렬, 병합 정렬, 퀵 정렬, 기수 정렬

출석 인정을 위한 보고서 작성

- 아래 질문에 대한 답을 포털에 제출

- 정렬의 성능 비교에는 최악의 경우 시간복잡도 외에도 여러 기준이 있다. 각 기준이 무엇을 뜻하는지 조사해보세요.
  - In-place / Out-place 정렬의 차이는 무엇일까?
  - Stable / Unstable 정렬의 차이는 무엇일까?