Data structure

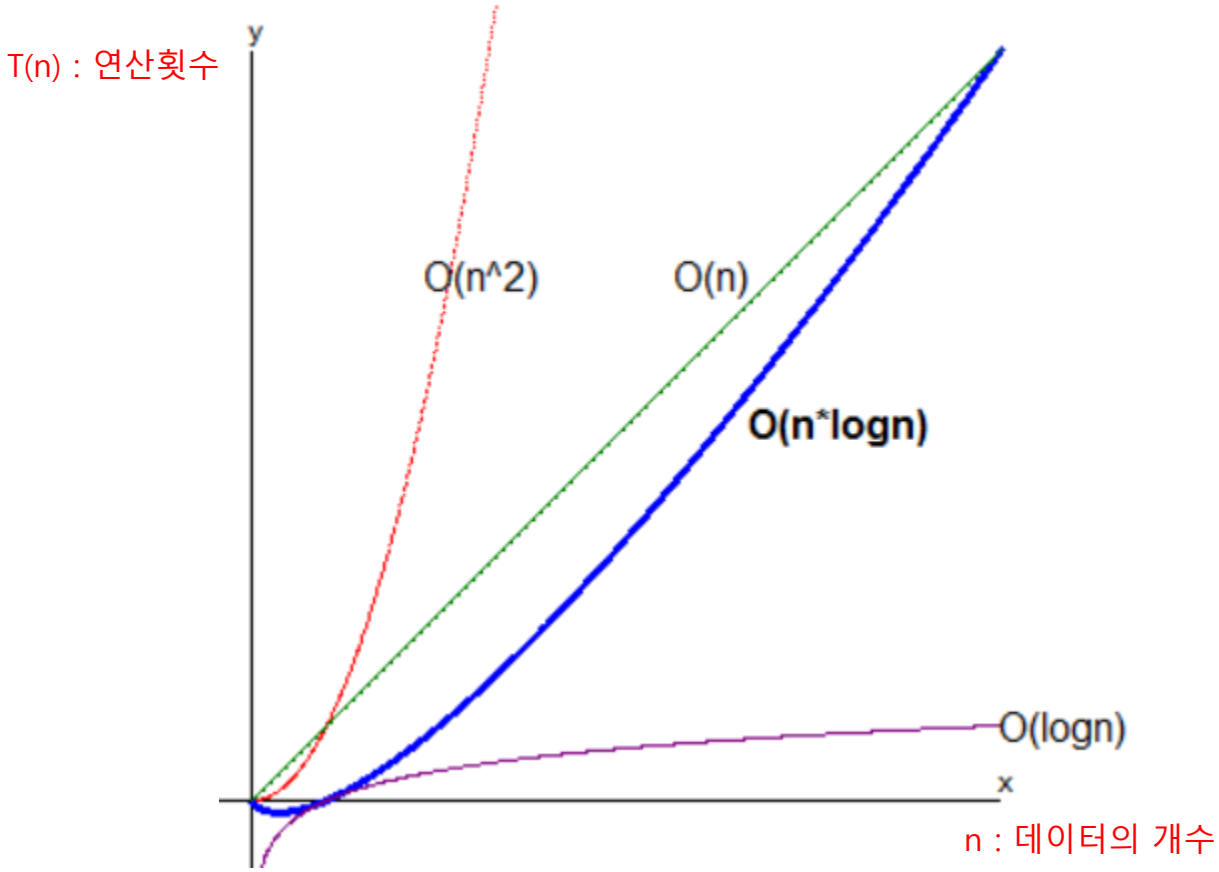
sorting

재귀함수
: 하노이 타워

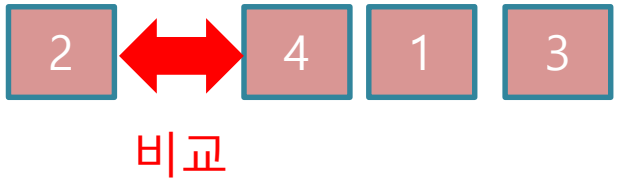
```
void HanoiTower(char from, char by, char to, int num)
{
    if (num == 1)  탈출조건
    {
        cout << "1 번째 쟁반이 " << from << "에서 " << to <<
            "로 이동" << endl;
        return;
    }

    HanoiTower(from, to, by, num - 1);
    cout << num << " 번째 쟁반이 " << from << "에서 " << to <<
        "로 이동" << endl;
    HanoiTower(by, from, to, num - 1);
}
```

같은 이름의 함수가 함수 내에 나올 때 이를 재귀함수라 부릅니다.

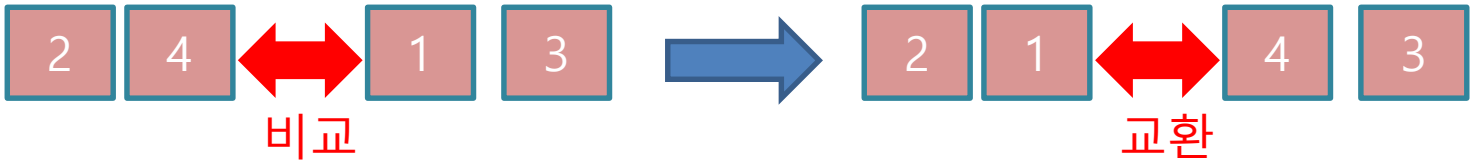
재귀함수
: 빅오 함수

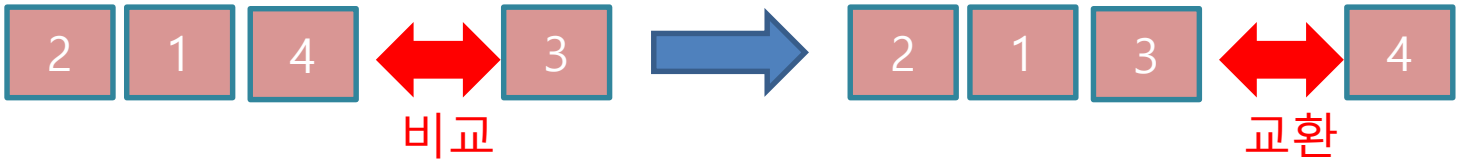


단순 정렬 알고리즘
: 버블 정렬



4가 더 크므로 바뀌지 않는다



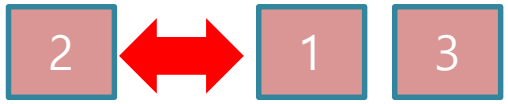


비교 3회

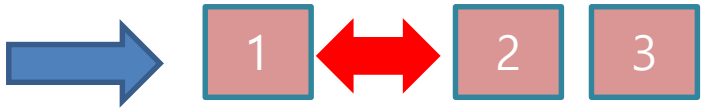
가장 큰 수가 맨 뒤로 가 있다!!

단순 정렬 알고리즘
: 버블 정렬

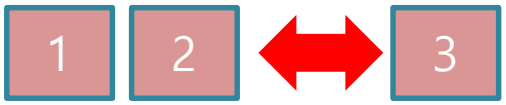
이미 정렬된 4를 빼고 나머지에서
같은 알고리즘을 적용



비교



교환



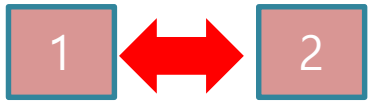
비교

비교 2회

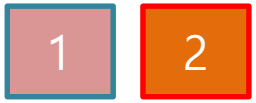
가장 큰 수가 맨 뒤로 가 있다!!

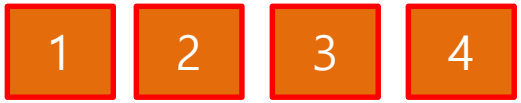
단순 정렬 알고리즘
: 버블 정렬

이미 정렬된 3, 4를 빼고 나머지에서
같은 알고리즘을 적용



비교





비교 1회

정렬이 완료됨

단순 정렬 알고리즘
: 버블 정렬

배열 길이가 4라면

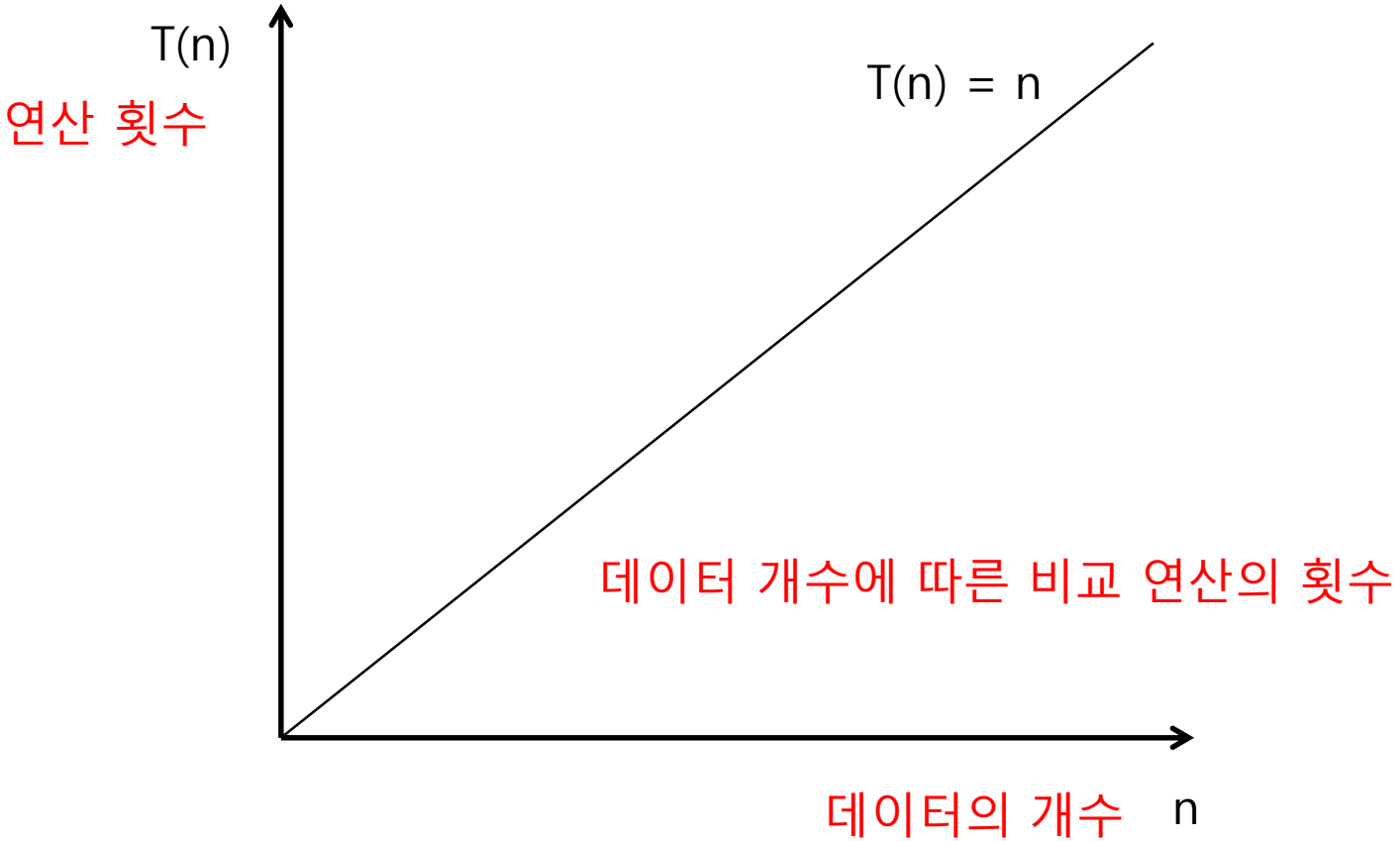
```
void SortingArray<T, n>::BubbleSort()
{
    for (int i = 0; i < arrlen() - 1; i++)
    {
        for (int j = 0; j < arrlen() - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                T temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

i는 0부터 2

i == 0 → j== 0, 1, 2 : 3회 비교
i == 1 → j== 0, 1 : 2회 비교
i == 2 → j== 0 : 1회 비교

단순 정렬 알고리즘
: 버블 정렬

시간복잡도(Time Complexity)



단순 정렬 알고리즘
: 버블 정렬

정렬이 끝날 때까지 몇 번 비교하는가?

```
void SortingArray<T, n>::BubbleSort()
{
    for (int i = 0; i < arrlen() - 1; i++)
    {
        for (int j = 0; j < arrlen() - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                T temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

비교연산

단순 정렬 알고리즘
: 버블 정렬

데이터 개수가 4개 일 때

3번 비교 → 2번 비교 → 1번 비교

1 + 2 + …….+ n-1

중학교 때 배운 등차수열을 이용합시다!

n : 항의 개수
a : 첫째 항
l : 마지막 항

$$\frac{n(a+l)}{2} = \frac{(n-1)*(1+n-1)}{2} = \frac{n-1+1+2n+n^2}{2}$$

빅오 계산할 때는 계수 상수 다 떼고 가장 큰 수만 남긴다.

$O(n^2)$ 데이터가 많아질수록 기하급수적!!

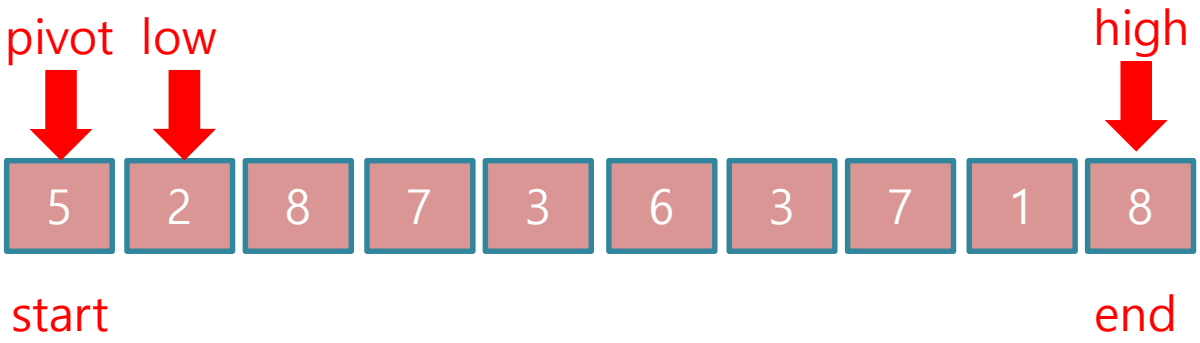
정렬 알고리즘
: Divide and Conquer

분할정복기법
: 재귀함수를 이용한 정렬 방식

```
void SortingArray<T, n>::quicksort(int start, int end)
{

    if (start >= end)//탈출 조건
        return;

    int middle = QsortExchange(start, end);
    quicksort(start, middle - 1);
    quicksort(middle + 1, end);
}
```

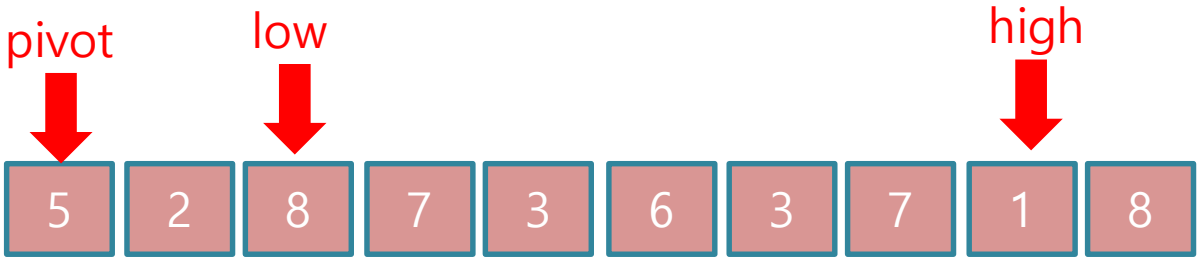
정렬 알고리즘
: Quicksort

Pivot을 맨 왼쪽에!
Low는 그 다음에!



정렬 알고리즘
: Quicksort

Low의 값이 pivot 값보다 클 때까지 오른쪽으로 이동

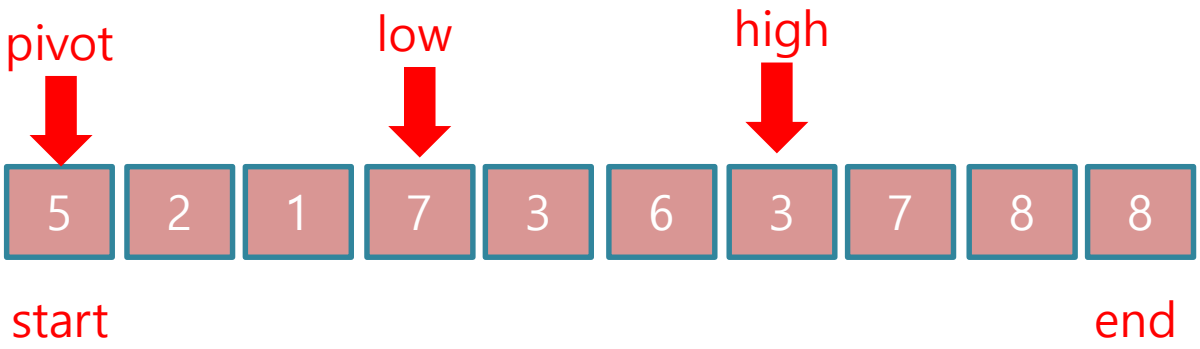


high의 값이 pivot 값보다 작을 때까지 왼쪽으로 이동

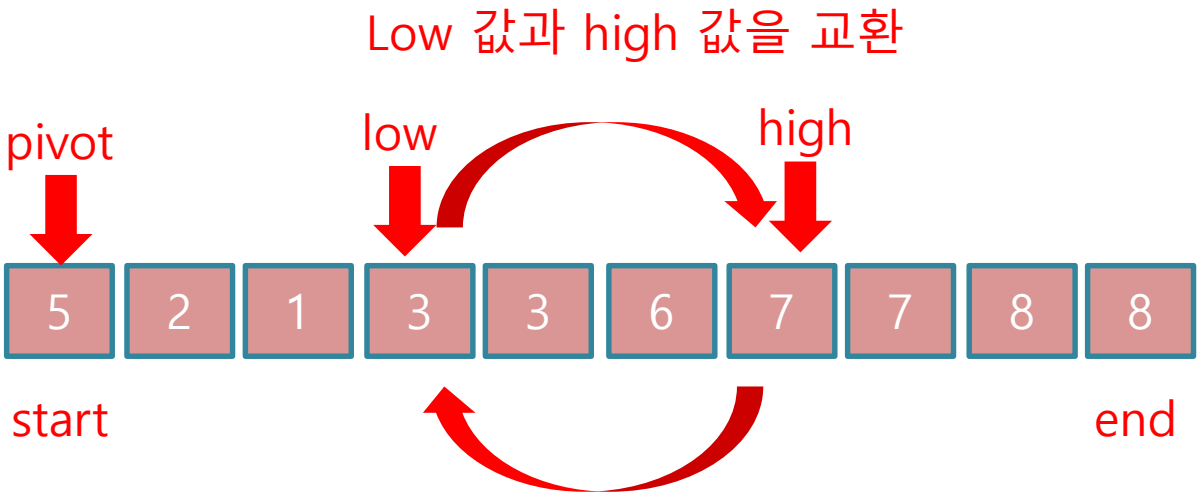
정렬 알고리즘
: Quicksort

정렬 알고리즘
: Quicksort

Low와 high가 교차할 때 까지 계속!

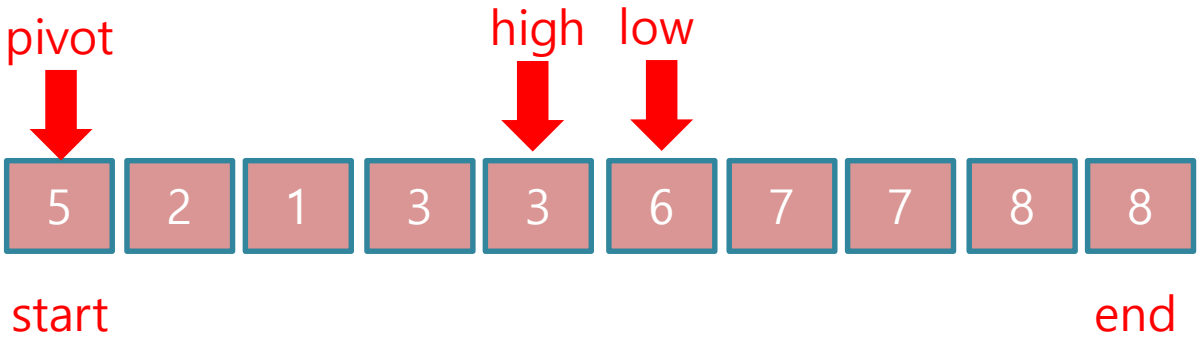


정렬 알고리즘
: Quicksort



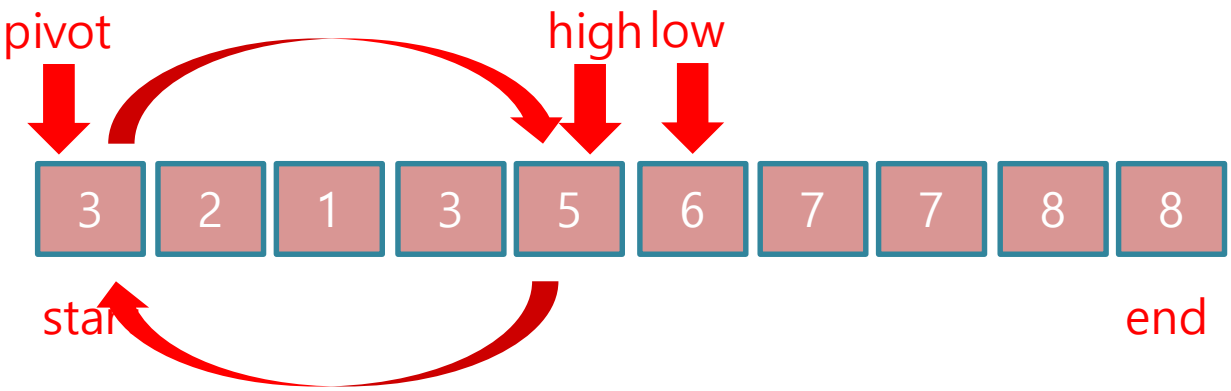
정렬 알고리즘
: Quicksort

Low와 high가 교차할 때!!!!!



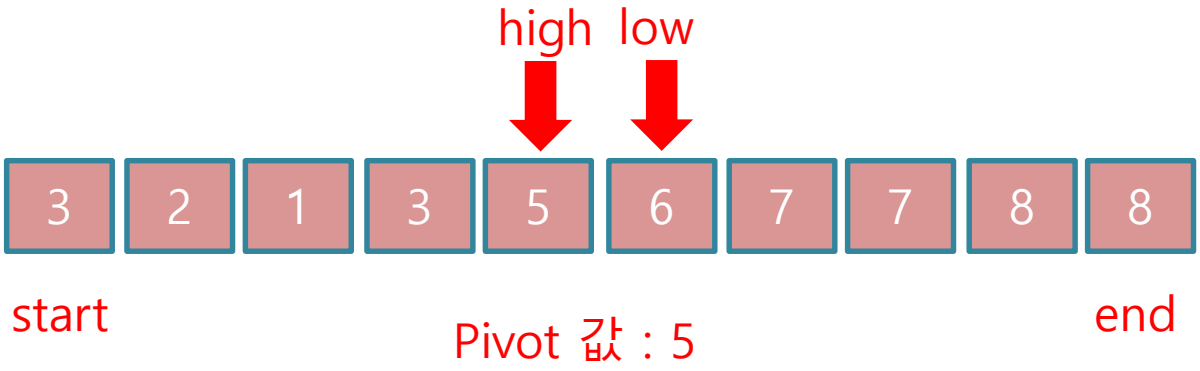
정렬 알고리즘
: Quicksort

Pivot과 high를 교환



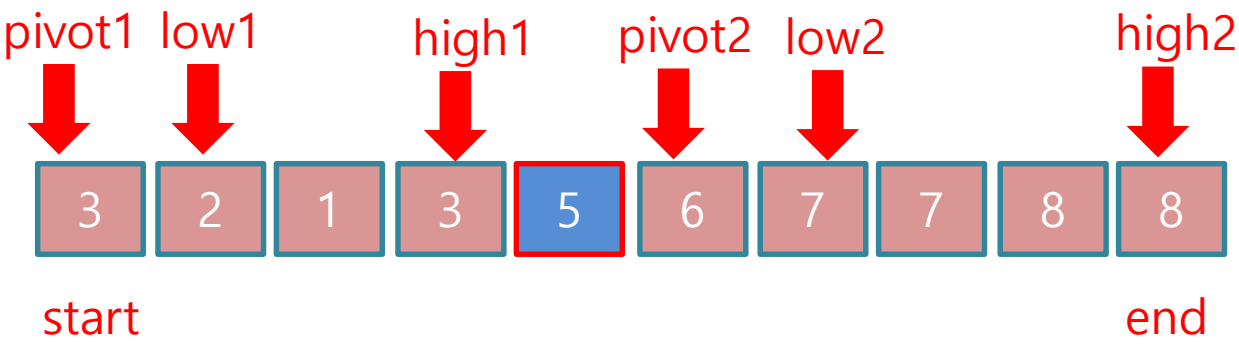
정렬 알고리즘
: Quicksort

Pivot 값을 기준으로 왼쪽에는 작은 값만
오른쪽에는 큰 값만 모였다!!



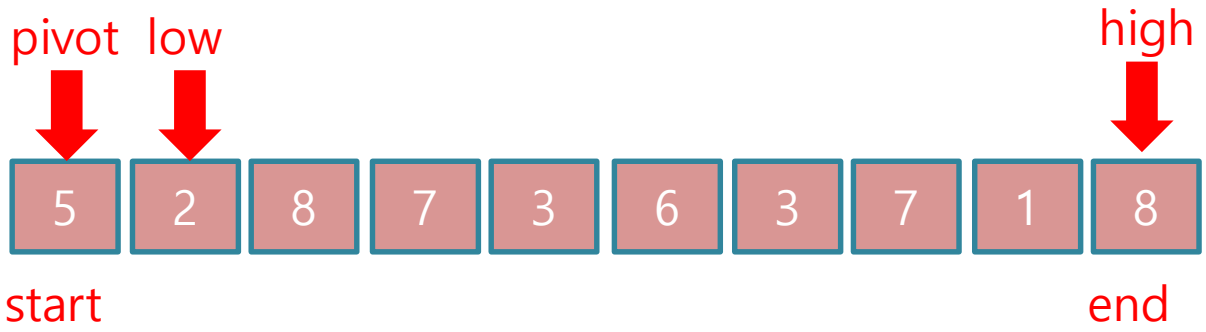
정렬 알고리즘
: Quicksort

Pivot 은 제외 하고 pivot의 왼쪽과 오른쪽에서 같은 알고리즘을 수행



정렬 알고리즘
: Quicksort

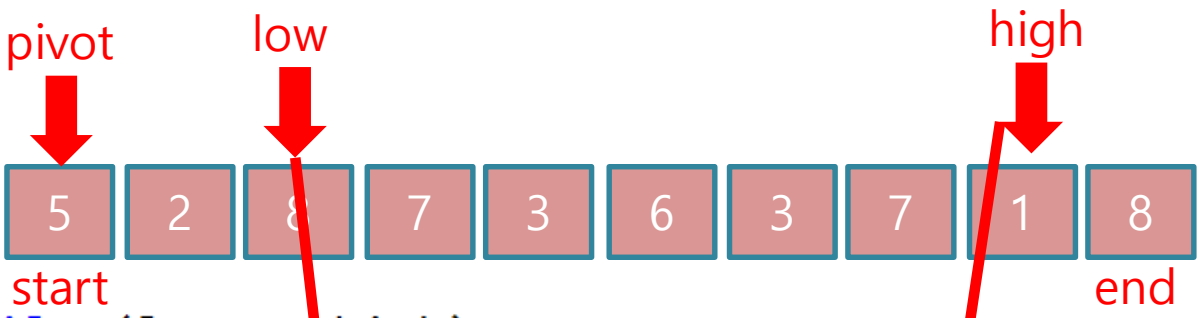
```
:QsortExchange(int start, int end)
```




```
T pivot = arr[start];
int low = start+1;
int high = end;
```

정렬 알고리즘
: Quicksort

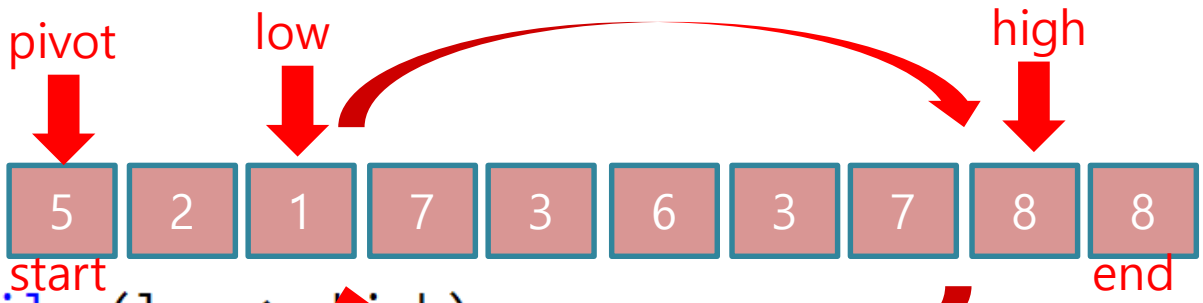
```
:QsortExchange(int start, int end)
```




```
while (low <= high)
{
    while (arr[low] <= pivot && low <= end)
        low++;

    while (arr[high] >= pivot && high >= (start+1))
        high--;

    if (low <= high)
        Swap(arr[low], arr[high]);
}
```

정렬 알고리즘
: Quicksort

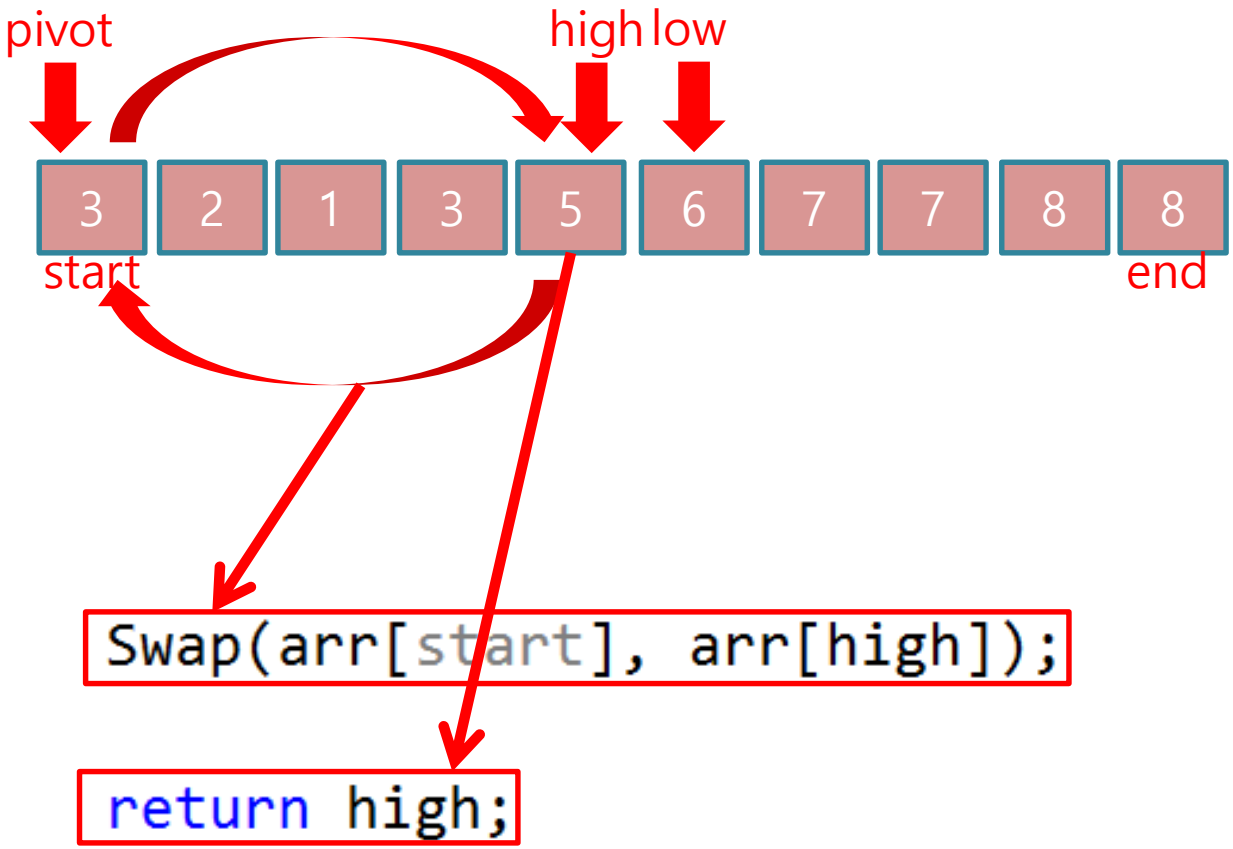
```
:QsortExchange(int start, int end)
```




```
while (low <= high)
{
    while (arr[low] <= pivot && low <= end)
        low++;

    while (arr[high] >= pivot && high >= (start+1))
        high--;

    if (low <= high)
        Swap(arr[low], arr[high]);
}
```

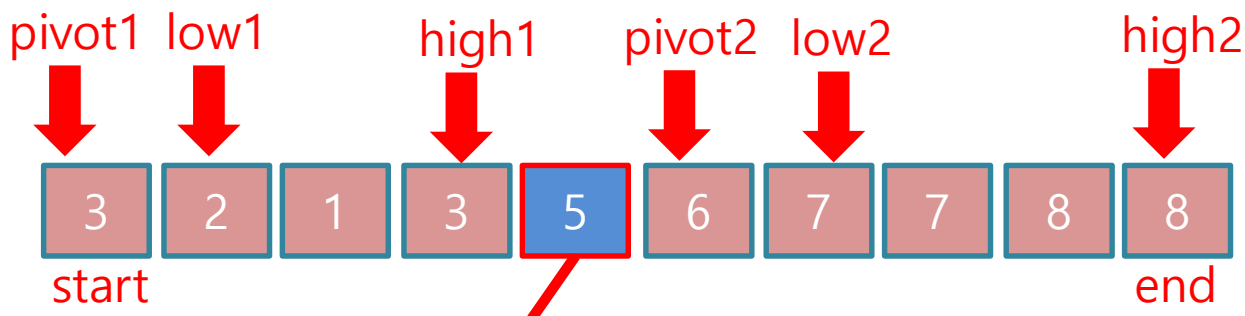
정렬 알고리즘
: Quicksort

```
:QsortExchange(int start, int end)
```



정렬 알고리즘
: Quicksort

quicksort(int start, int end)




```
void SortingArray<T, n>::quicksort(int start, int end)
{

    if (start >= end)//탈출 조건
        return;

    int middle = QsortExchange(start, end);
    quicksort(start, middle - 1);
    quicksort(middle + 1, end);
}
```

정렬 알고리즘
: Quicksort

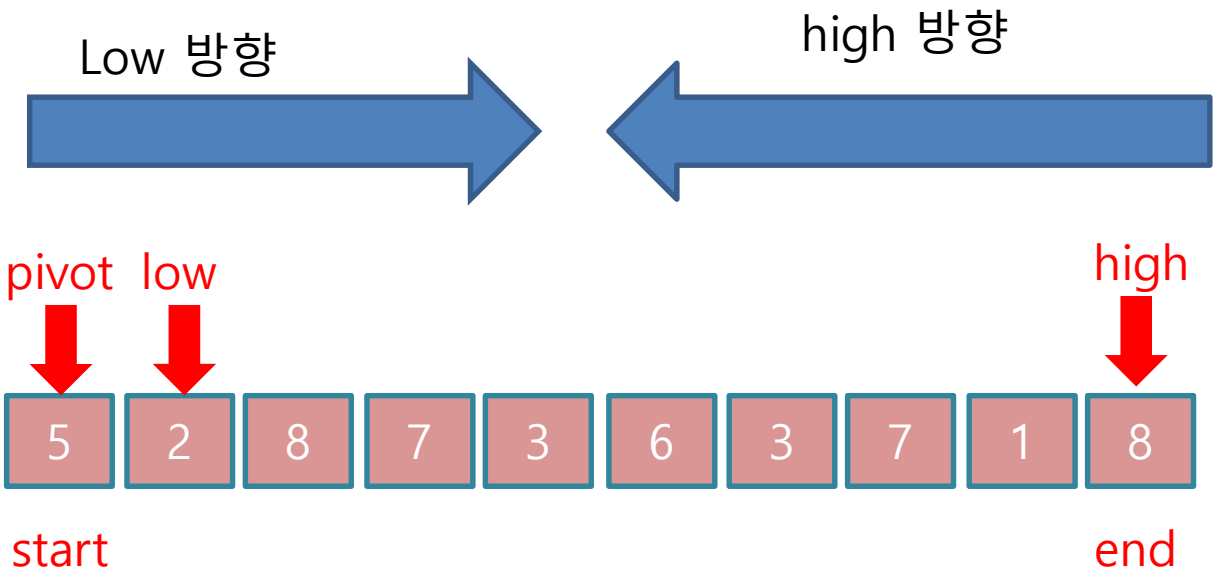
빅오를 계산해봅시다

연산 횟수를 계산하기 위해 주요 비교 연산 부분을 결정합니다.

Low와 pivot, high와 pivot과의 크기 비교 연산!!

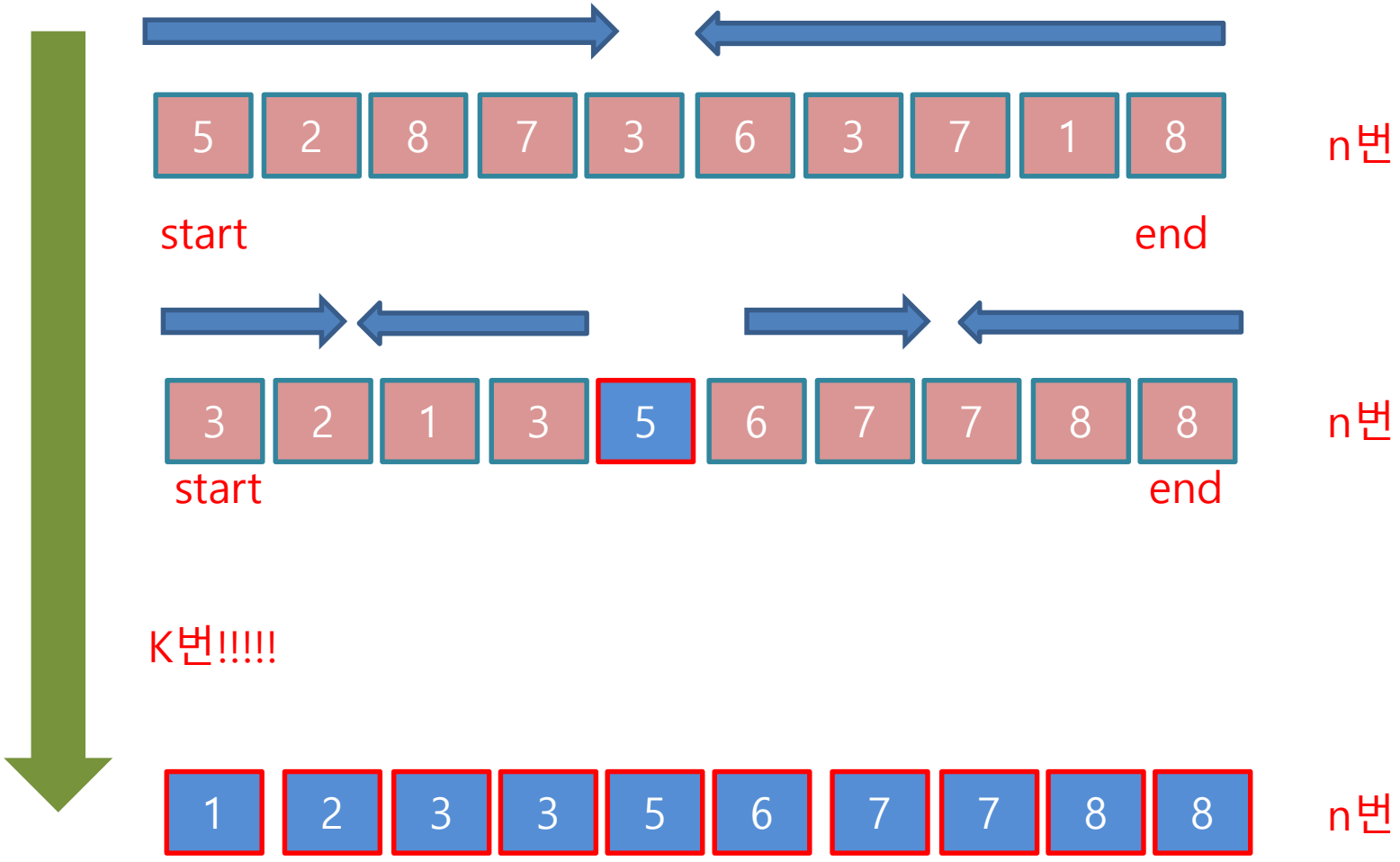
```
while (arr[low] <= pivot && low <= end)
    low++;

while (arr[high] >= pivot && high >= (start+1))
    high--;
```

정렬 알고리즘
: Quicksort



즉 데이터 개수와 같이 n번 연산합니다

정렬 알고리즘
: Quicksort



$$T(n) = n * k$$

정렬 알고리즘
: Quicksort

$$n * \left(\frac{1}{2}\right)^k = 1 \quad \Longrightarrow \quad k = \log_2 n$$

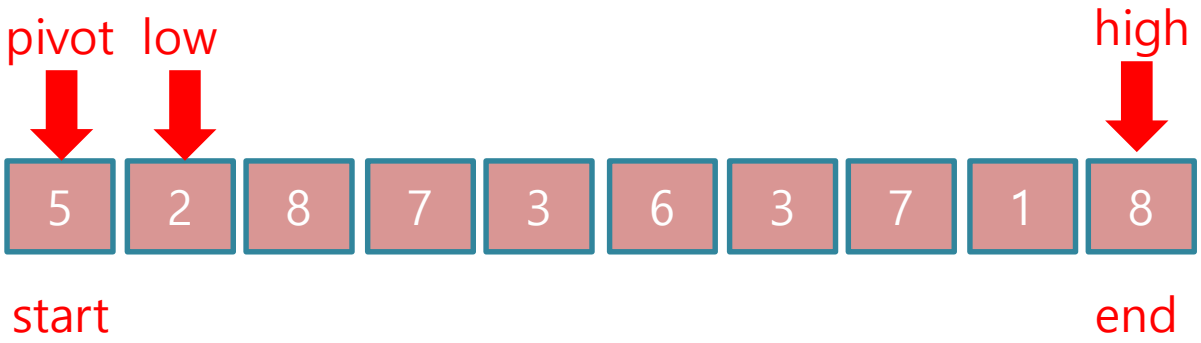
$$T(n) = n * \log_2 n$$

$$O(n * log\, n)$$

정렬 알고리즘
: Quicksort

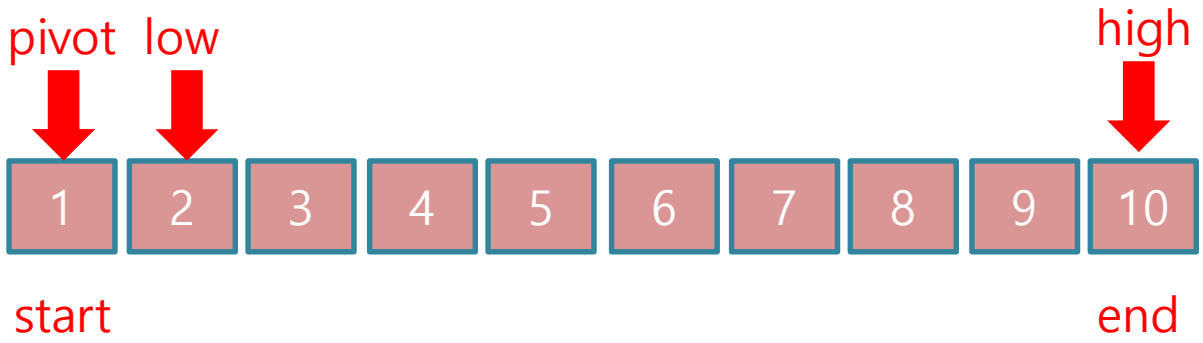
항상 pivot을 맨 왼쪽에 두면.....

Pivot을 맨 왼쪽에!
Low는 그 다음에!



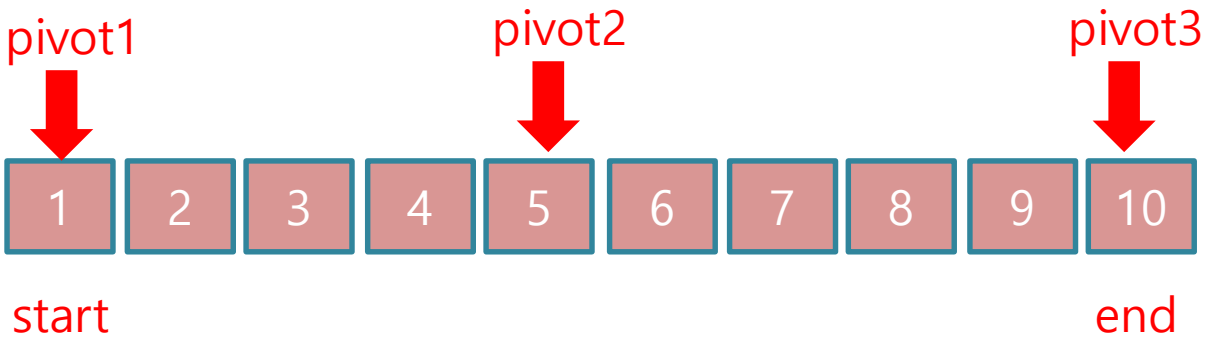
정렬 알고리즘
: Quicksort

이렇게 완벽하게 정렬된 경우 배열이 두 개로 쪼개지지 않으므로
성능이 좋지 않습니다



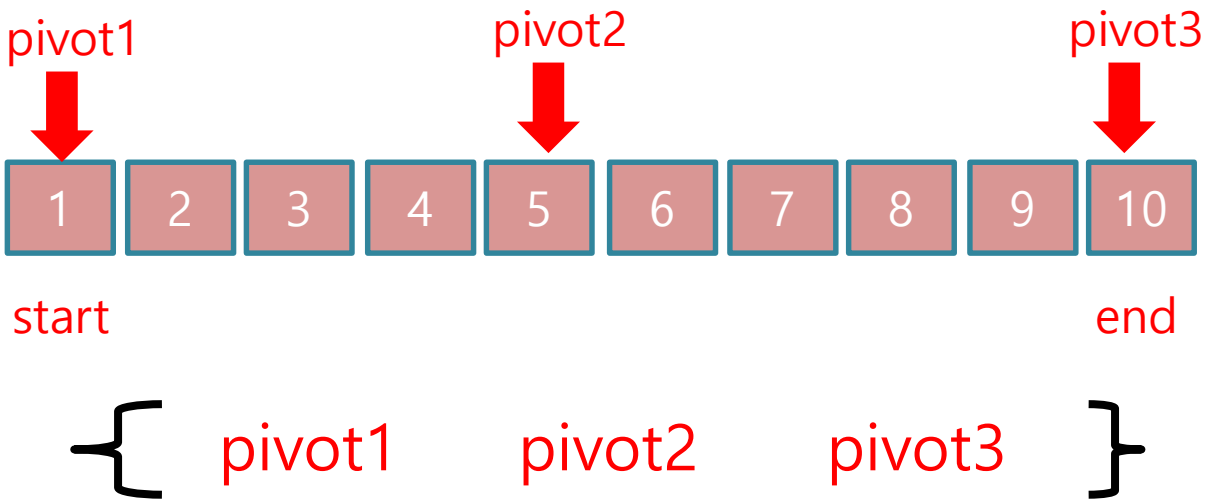
정렬 알고리즘
: Quicksort

이럴 경우에는 pivot을 결정할 때
맨 왼쪽, 가운데, 맨 오른쪽 값을 비교해 중간 값을
Pivot으로 만듭니다



정렬 알고리즘
: Quicksort

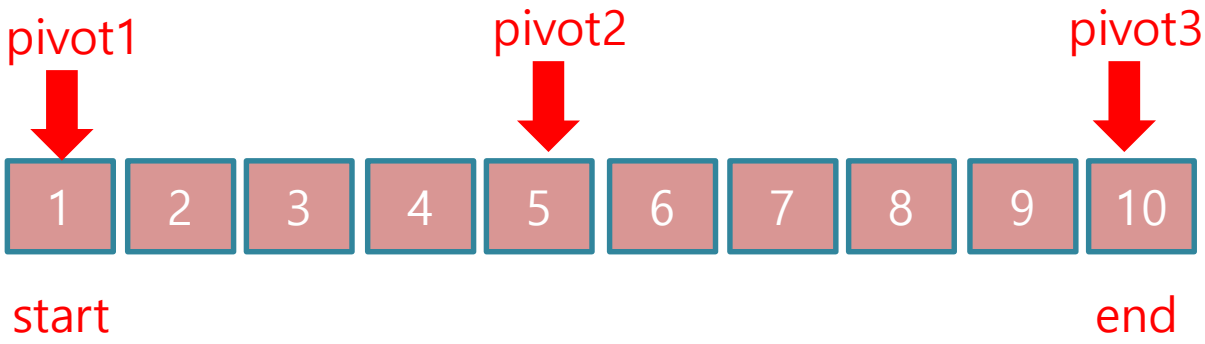
```
GetPivot(int start, int end)
```



Pivot 인덱스에서의 값을 정렬하기 위해 버블 정렬을 사용

정렬 알고리즘
: Quicksort

GetPivot(int start, int end)

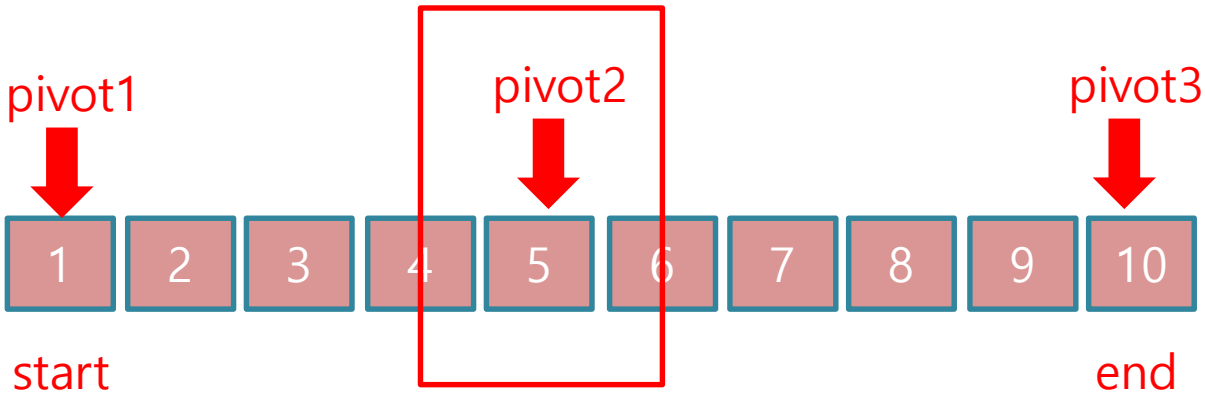



```
int SortingArray<T, n>::GetPivot(int start, int end)
{
    int mid = (start + end) / 2;

    int idxArr[3] = { start, mid, end };

    if (arr[idxArr[0]] > arr[idxArr[1]])
        SwapIdx(idxArr[0], idxArr[1]);
    if (arr[idxArr[1]] > arr[idxArr[2]])
        SwapIdx(idxArr[1], idxArr[2]);
    if (arr[idxArr[0]] > arr[idxArr[1]])
        SwapIdx(idxArr[0], idxArr[1]);

    return idxArr[1];
}
```

버블정렬

가운데 값인 5의 인덱스 mid를 반환할 것

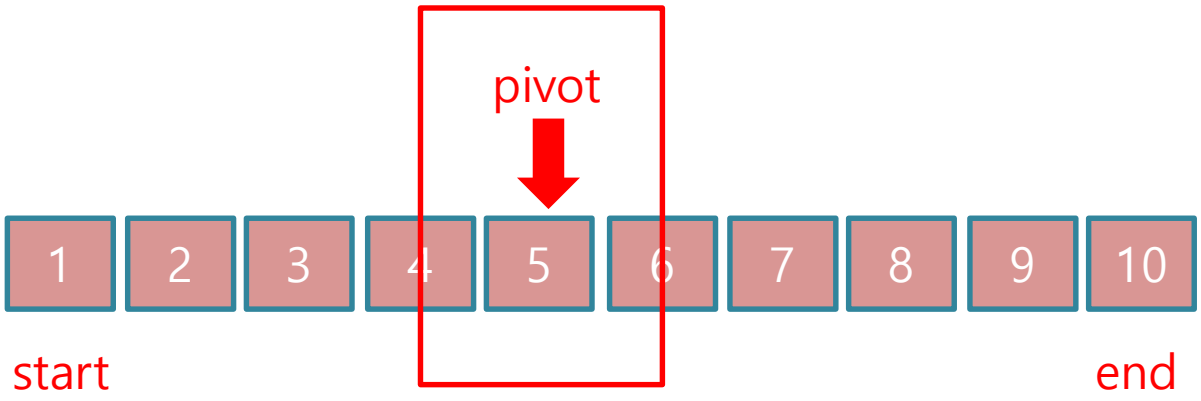
정렬 알고리즘
: Quicksort

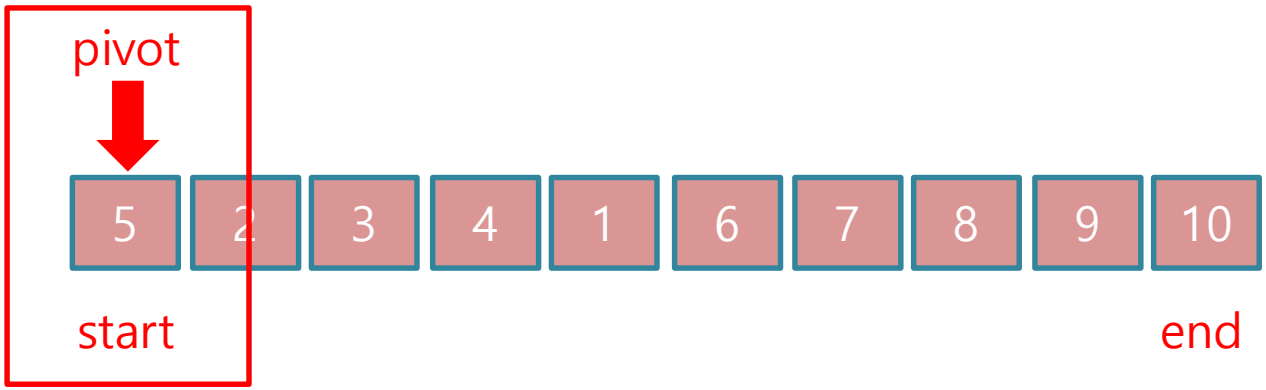


그럼 중간 값 pivot을 구했으니 코드를 다시 짜야 하나요?

정렬 알고리즘
: Quicksort

Pivot을 맨 왼쪽으로 옮기면 됩니다.





정렬 알고리즘
: Quicksort

```
int SortingArray<T, n>::QsortExchange(int start, int end)
{
    callOfExchange++;

    //pivot을 start로 고정할 때와 세 가지 중 하나를 고를 때의
    int idxPivot = GetPivot(start, end);
    Swap(arr[start], arr[idxPivot]);


    //pivot은 맨 앞쪽 것으로 한다.
    T pivot = arr[start];
    int low = start+1;
    int high = end;
```

Getpivot()함수로 가운데값 pivot을 얻고

Pivot을 맨 앞쪽으로 옮긴다(바꾼다)

위처럼 두 줄만 넣어주면
평균적으로 성능이 좋아진다고
합니다!!

Quicksort는 정렬 알고리즘 중 성능이 꽤 좋다고 알려져 있고
Worst case로 빅오를 따지는 다른 경우와 다르게 average case로 계산합니다