분할 정복

최백준 choi@startlink.io

분할 정복

분할 정복

Divide & Conquer

- 분할 정복은 문제를 2개 또는 그 이상의 작은 부분 문제로 나눈 다음 푸는 것(분할)
- 푼 다음에는 다시 합쳐서 정답을 구할 때도 있음 (정복)
- 대표적인 분할 정복 알고리즘
    - 퀵 소트
    - 머지 소트
    - 큰 수 곱셈 (카라추바 알고리즘)
    - FFT

분할 정복

Divide & Conquer

- 분할 정복과 다이나믹은
- 문제를 작은 부분 문제로 나눈 다는 점은 동일하다
- 분할 정복: 문제가 겹치지 않음
- 다이나믹: 문제가 겹쳐서 겹치는 것을 Memoization으로 해결

이분 탐색

이분 탐색

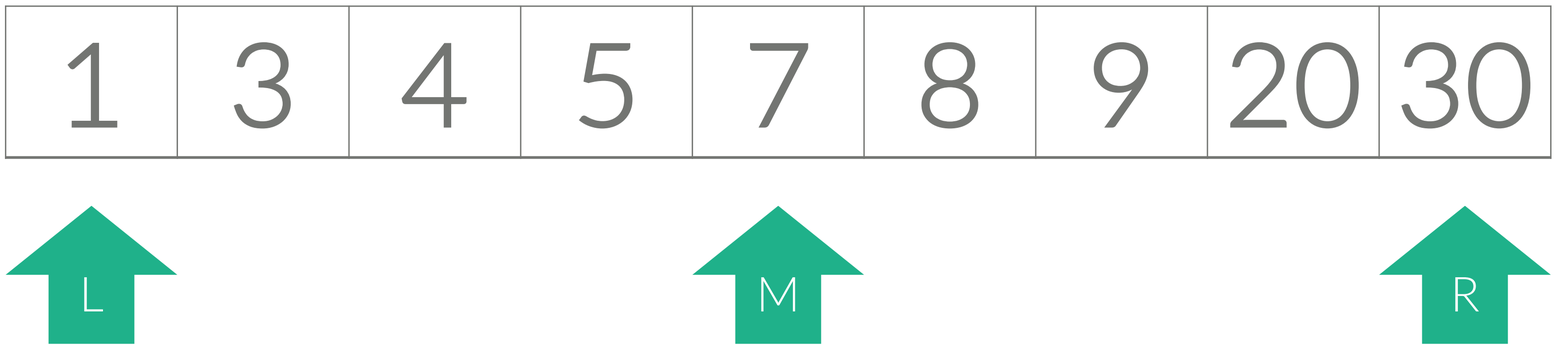
Binary Search

- 정렬되어 있는 리스트에서 어떤 값을 빠르게 찾는 알고리즘
- 리스트의 크기를 N 이라고 했을 때
- O(lgN)의 시간이 걸린다.

이분 탐색

Binary Search

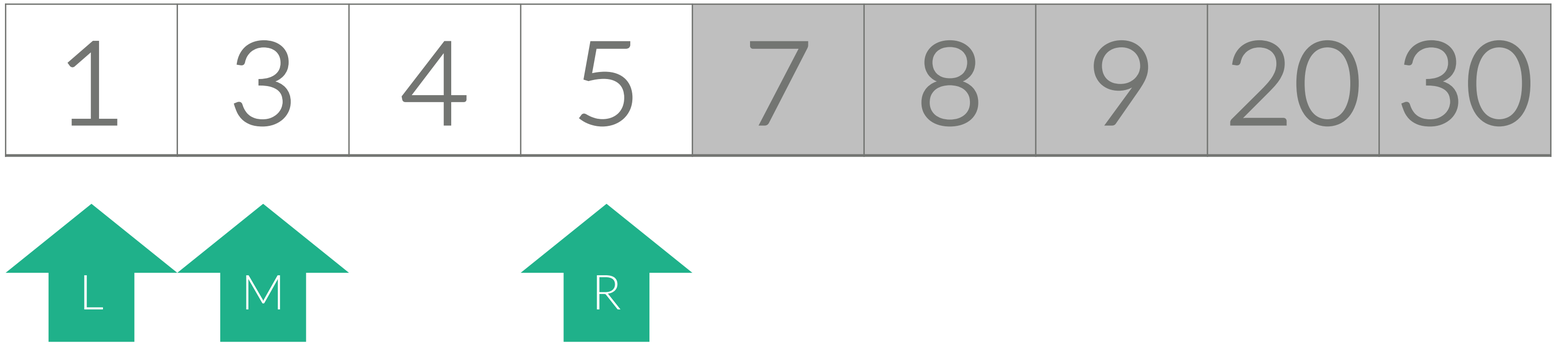
- 4를 찾아보자
- L = 0, R = 8, M = 4
- 4 < 7 이기 때문에 왼쪽 (L ~ M-1)에 4가 있을 수 있다.



이분 탐색

Binary Search

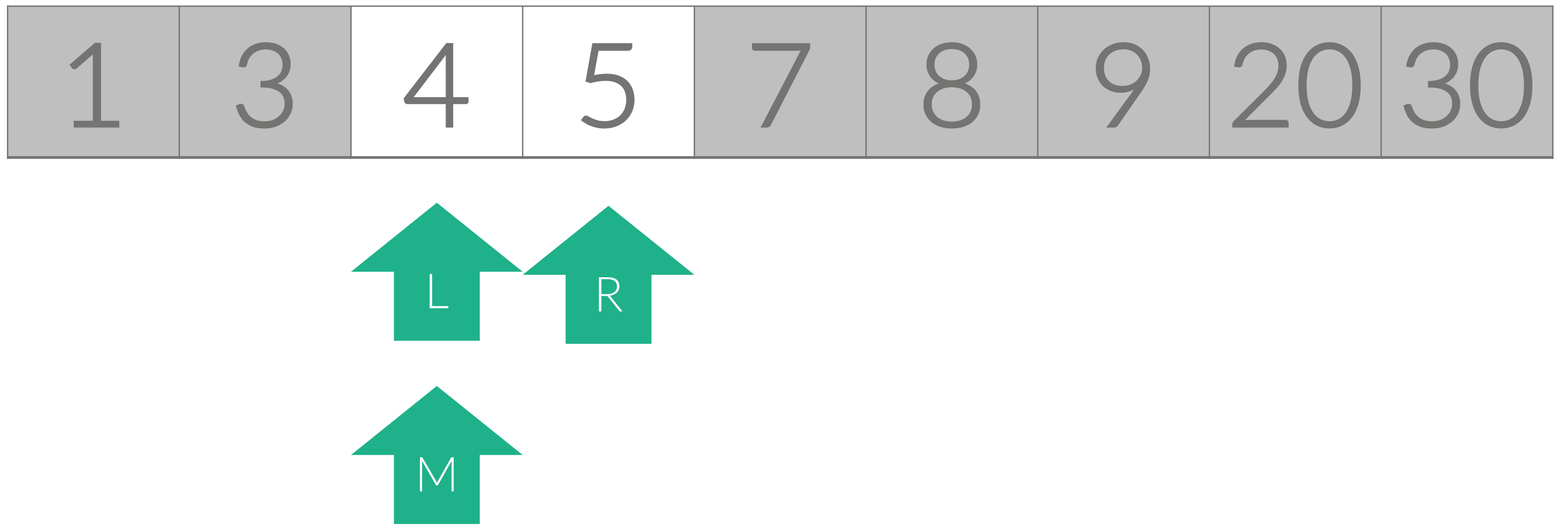
- 4를 찾아보자
- L = 0, R = 3, M = 1
- 4 > 3 이기 때문에 오른쪽 (M+1 ~ R)에 4가 있을 수 있다.



이분 탐색

Binary Search

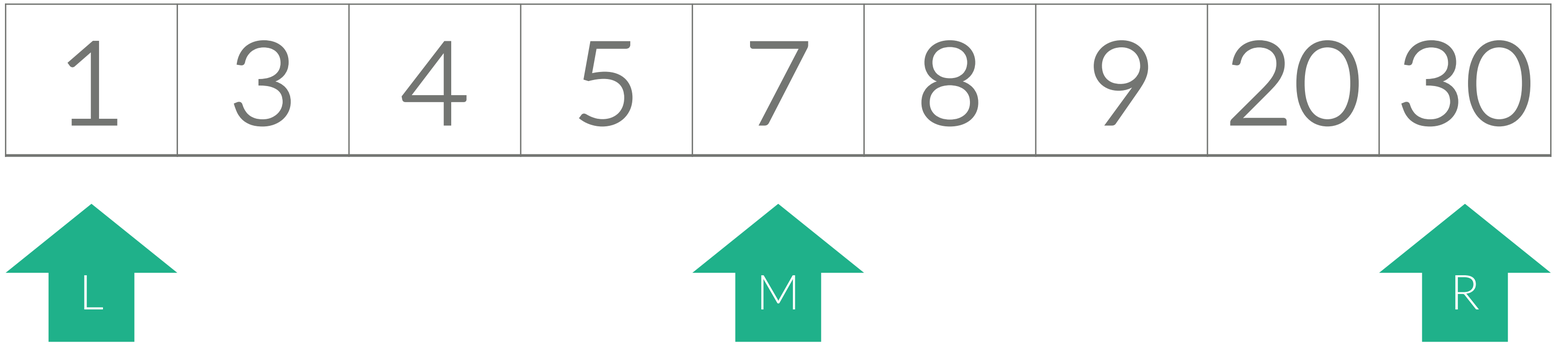
- 4를 찾아보자
- L = 2, R = 3, M = 2
- 4 == 4 이다. 4를 찾았다.



이분 탐색

Binary Search

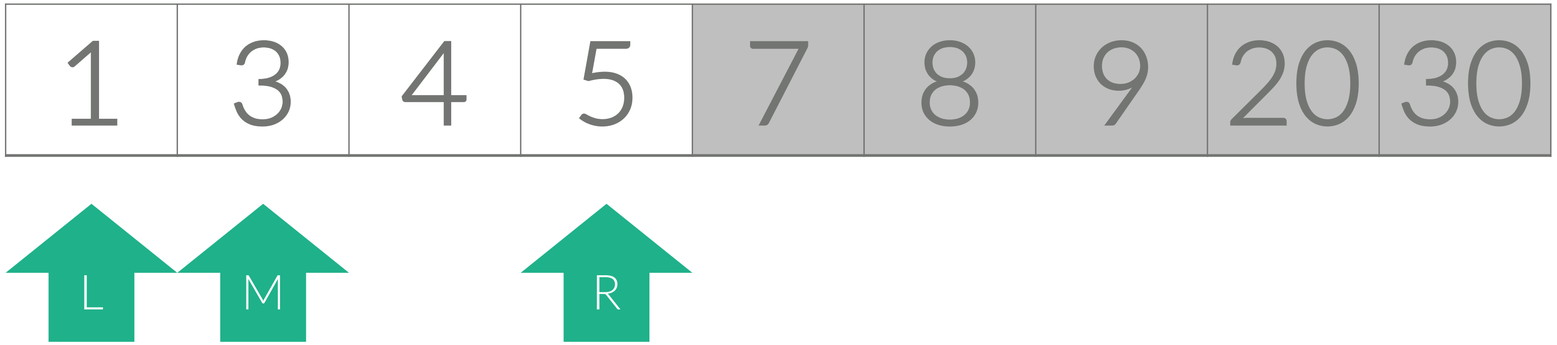
- 2를 찾아보자
- L = 0, R = 8, M = 4
- 2 < 7 이기 때문에 왼쪽 (L ~ M-1)에 4가 있을 수 있다.



이분 탐색

Binary Search

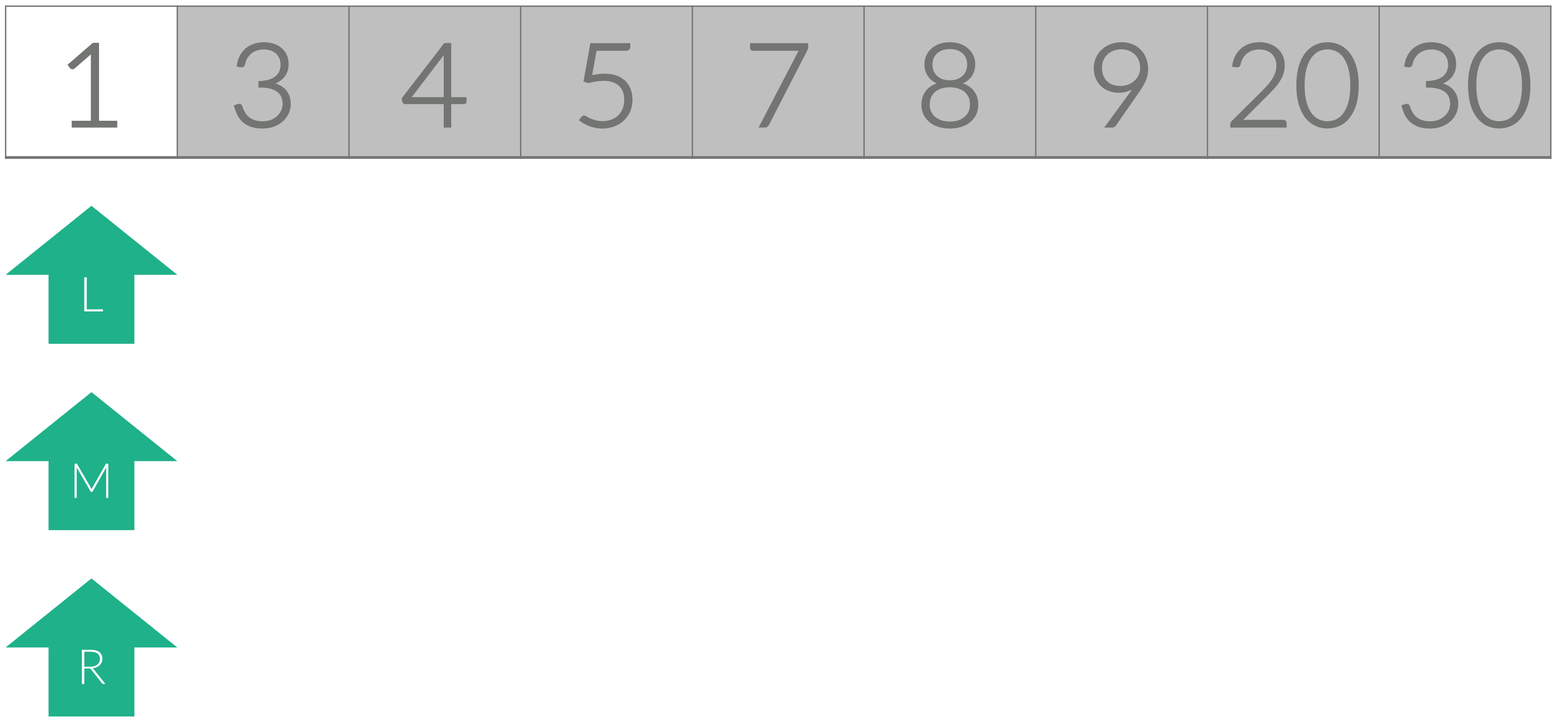
- 2를 찾아보자
- L = 0, R = 3, M = 1
- 2 < 3 이기 때문에 왼쪽 (L~ M-1)에 2가 있을 수 있다.



이분 탐색

Binary Search

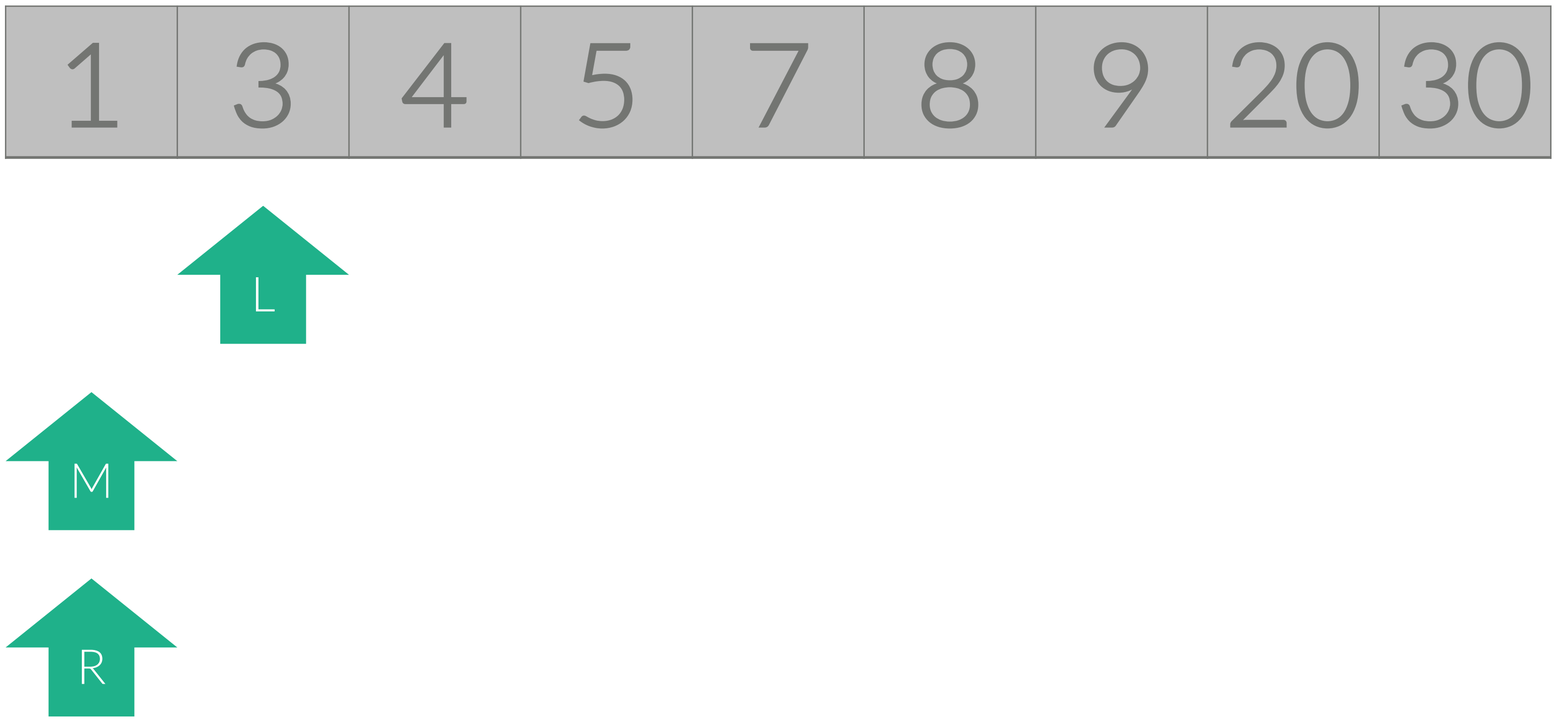
- 2를 찾아보자
- L = 0, R = 0, M = 0
- 2 > 1 이기 때문에 오른쪽 (M+1~ R)에 2가 있을 수 있다.



이분 탐색

Binary Search

- 2를 찾아보자
- L = 1, R = 0, M = 0
- L < R 이기 때문에, 이분 탐색을 종료한다. 2는 리스트에 없다.



이분 탐색

Binary Search

- 정렬되어 있는 리스트에서 어떤 값을 빠르게 찾는 알고리즘
- 리스트의 크기를 N 이라고 했을 때
- lgN의 시간이 걸린다.
- 시간 복잡도가 lgN인 이유는
- 크기가 N인 리스트를 계속해서 절반으로 나누기 때문이다.
- $2^k = N$ 일 때, $k = lgN$

이분 탐색

Binary Search


```
while (left <= right) {
    int mid = (left + right) / 2;
    if (a[mid] == x) {
        position = mid;
        break;
    } else if (a[mid] > x) {
        right = mid-1;
    } else {
        left = mid+1;
    }
}
```

숫자 카드

https://www.acmicpc.net/problem/10815

- 이분 탐색을 이용해 풀 수 있다.

숫자 카드

https://www.acmicpc.net/problem/10815

- 소스: http://codeplus.codes/4e188439e8044b0db595d3f58bb7a383

상한과 하한

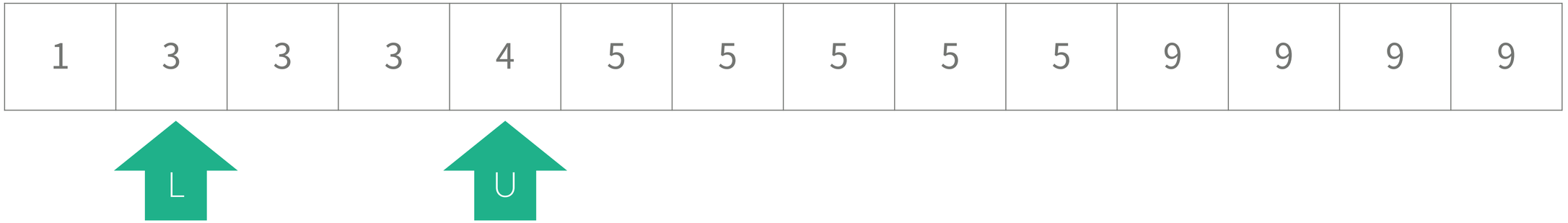
Upper & Lower Bound

- 어떤 수열 A가 있을 때, K의 상한은 크거나 같은 수, 하한은 작거나 같은 수
- 보통 구현을 할 때는 아래와 같은 의미로 사용한다.
- 상한: 큰 수 중 첫 번째 수
- 하한: 크거나 같은 수 중 첫 번째 수

상한과 하한

Upper & Lower Bound

- 어떤 수열 A가 있을 때, K의 상한은 크거나 같은 수, 하한은 작거나 같은 수
- 보통 구현을 할 때는 아래와 같은 의미로 사용한다.
- 상한: 큰 수 중 첫 번째 수
- 하한: 크거나 같은 수 중 첫 번째 수



상한과 하한

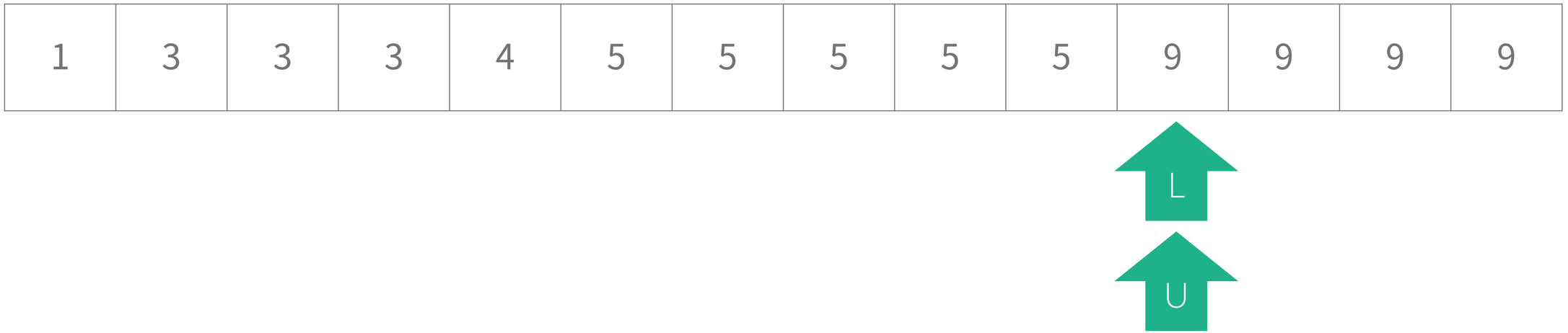
Upper & Lower Bound

- 어떤 수열 A가 있을 때, K의 상한은 크거나 같은 수, 하한은 작거나 같은 수
- 보통 구현을 할 때는 아래와 같은 의미로 사용한다.
- 상한: 큰 수 중 첫 번째 수
- 하한: 크거나 같은 수 중 첫 번째 수

상한과 하한

Upper & Lower Bound

- 어떤 수열 A가 있을 때, K의 상한은 크거나 같은 수, 하한은 작거나 같은 수
- 보통 구현을 할 때는 아래와 같은 의미로 사용한다.
- 상한: 큰 수 중 첫 번째 수
- 하한: 크거나 같은 수 중 첫 번째 수



상한과 하한

Upper & Lower Bound

- 이분 탐색을 이용해서 구현할 수 있다.

상한과 하한

Upper & Lower Bound


```
// lower bound
while (l <= r) {
    int mid = (l+r)/2;
    if (a[mid] == num) {
        ans = mid;
        r = mid-1;
    } else if (a[mid] > num) {
        r = mid-1;
    } else {
        l = mid+1;
    }
}
```

상한과 하한

Upper & Lower Bound


```
// upper bound
while (l <= r) {
    int mid = (l+r)/2;
    if (a[mid] == num) {
        ans = mid+1;
        l = mid+1;
    } else if (a[mid] > num) {
        r = mid-1;
    } else {
        l = mid+1;
    }
}
```

숫자 카드 2

https://www.acmicpc.net/problem/10816

- 이분 탐색을 이용해 풀 수 있다.

숫자 카드 2

https://www.acmicpc.net/problem/10816

- 소스: http://codeplus.codes/4b0580d84145483dbab368477d257505

머지 소트

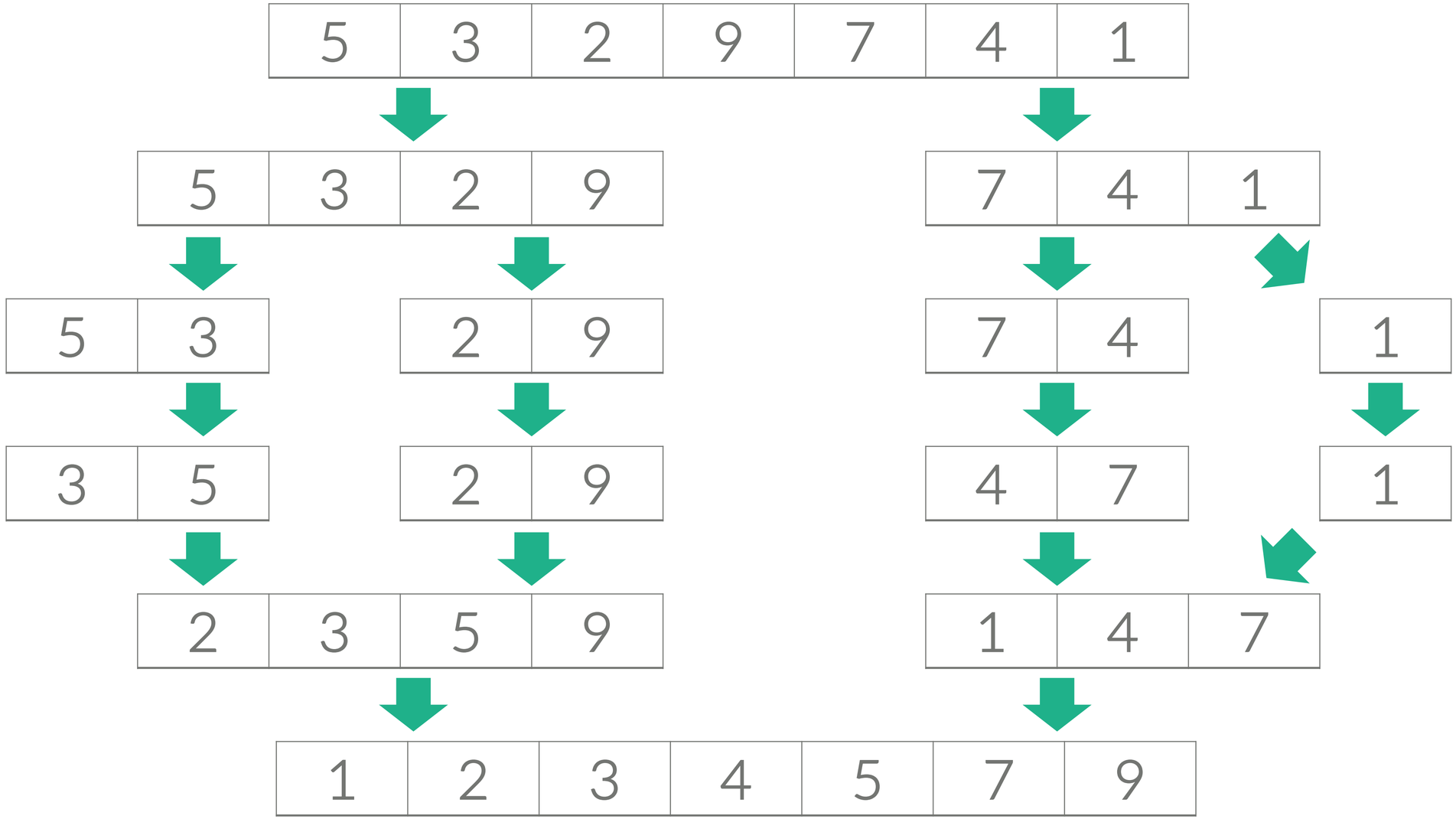
머지 소트

Merge Sort

- N개를 정렬하는 알고리즘
- N개를 N/2, N/2개로 나눈다.
- 왼쪽 N/2와 오른쪽 N/2를 정렬한다.
- 두 정렬한 결과를 하나로 합친다.

머지 소트

Merge Sort



머지 소트

Merge Sort


```
void sort(int start, int end) {
    if (start == end) {
        return;
    }
    int mid = (start+end)/2;
    sort(start, mid);
    sort(mid+1, end);
    merge(start, end);
}
```

머지 소트

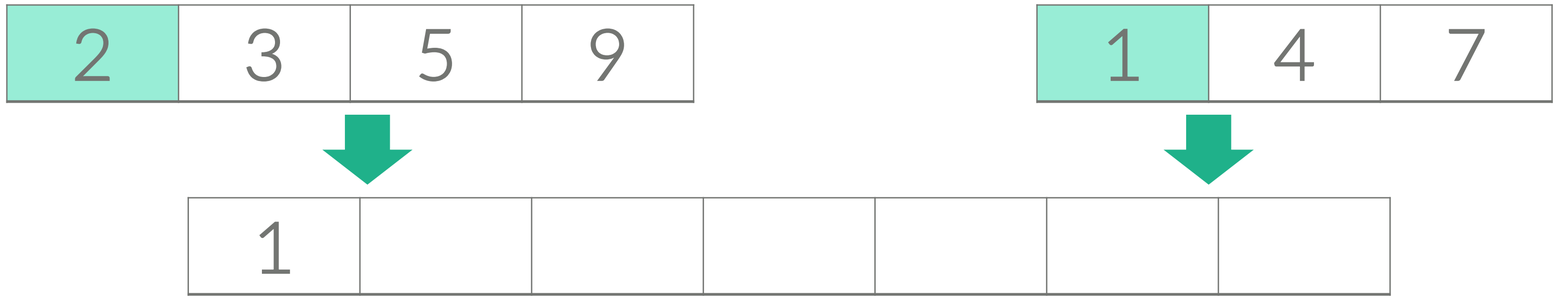
Merge Sort


```
void merge(int start, int end) {
    int mid = (start+end)/2;
    int i = start, j = mid+1, k = 0;
    while (i <= mid && j <= end) {
        if (a[i] <= a[j]) b[k++] = a[i++];
        else b[k++] = a[j++];
    }
    while (i <= mid) b[k++] = a[i++];
    while (j <= end) b[k++] = a[j++];
    for (int i=start; i<=end; i++) {
        a[i] = b[i-start];
    }
}
```

배열 합치기

https://www.acmicpc.net/problem/11728

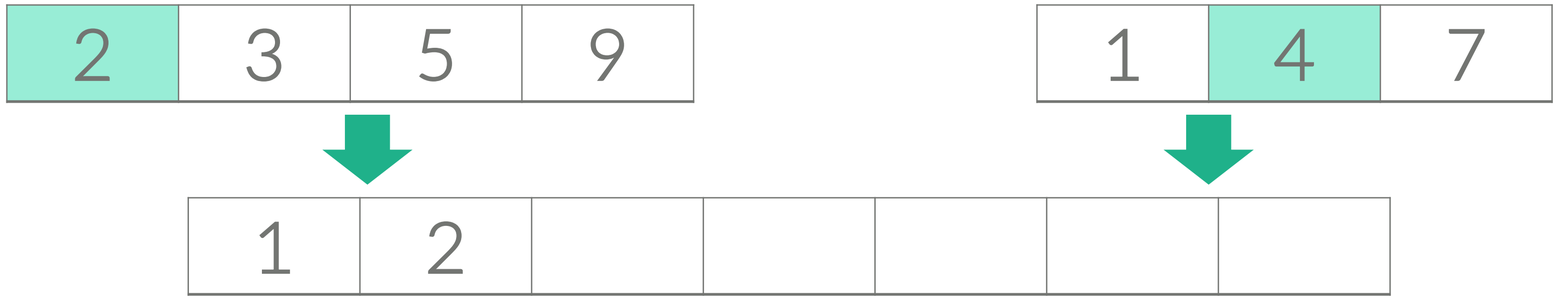
- 머지 소트에서 배열을 합치는 알고리즘 구현



배열 합치기

https://www.acmicpc.net/problem/11728

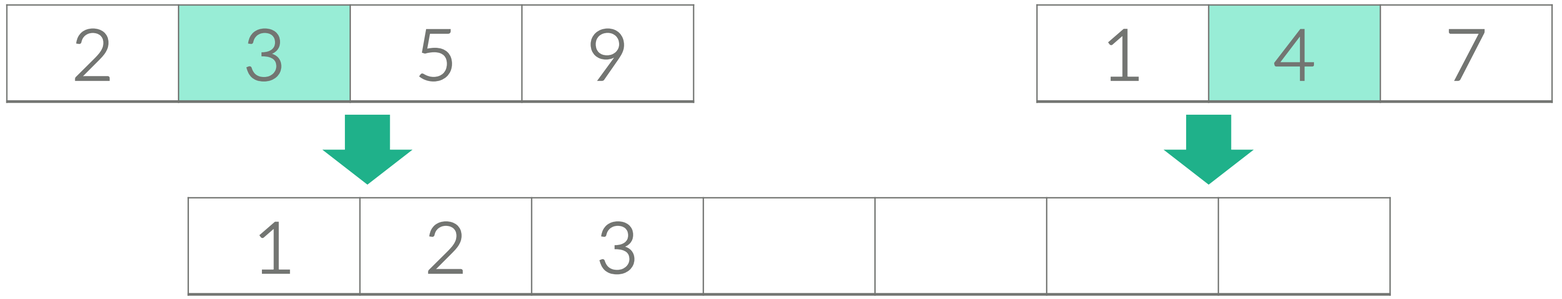
- 머지 소트에서 배열을 합치는 알고리즘 구현



배열 합치기

https://www.acmicpc.net/problem/11728

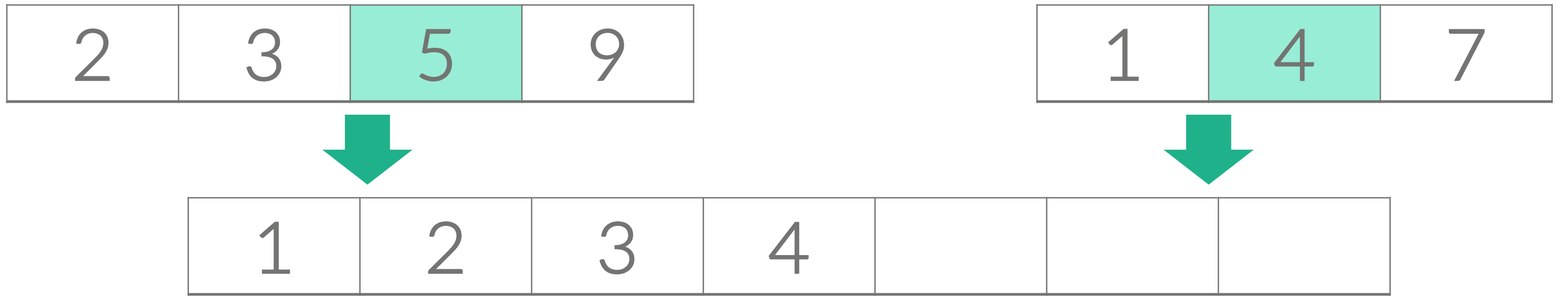
- 머지 소트에서 배열을 합치는 알고리즘 구현



배열 합치기

https://www.acmicpc.net/problem/11728

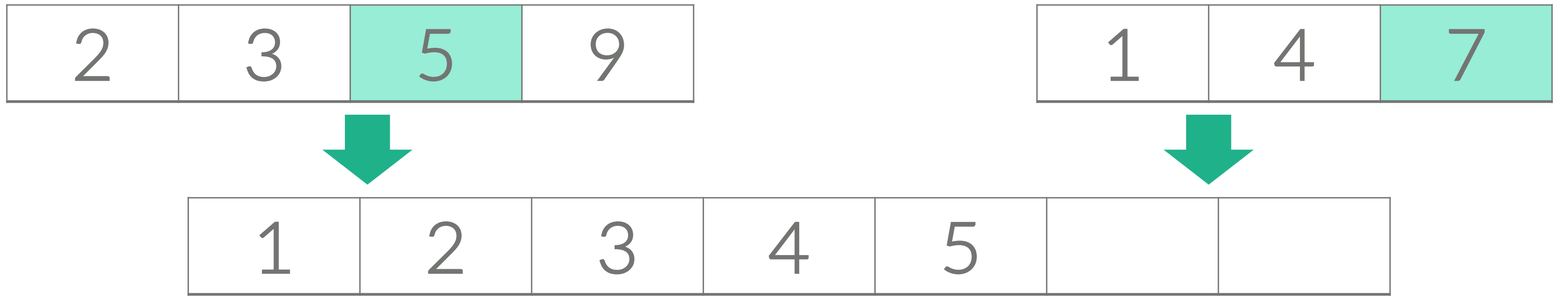
- 머지 소트에서 배열을 합치는 알고리즘 구현



배열 합치기

https://www.acmicpc.net/problem/11728

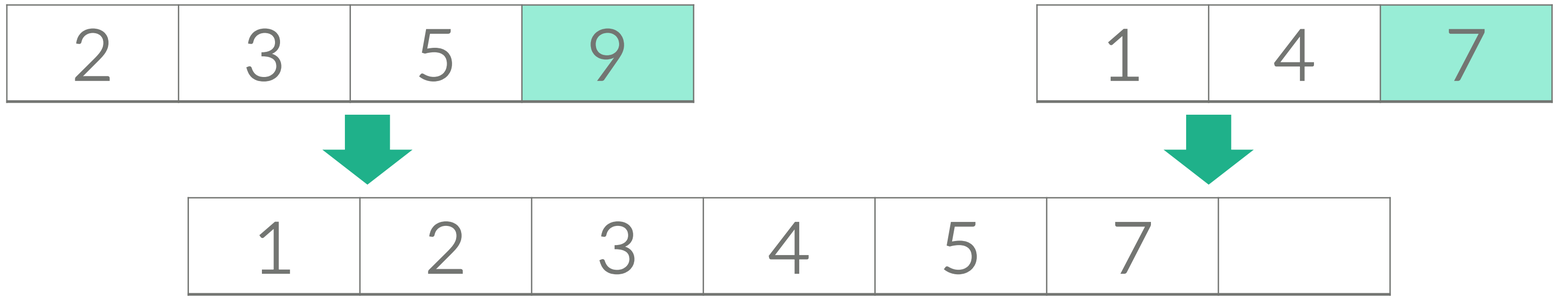
- 머지 소트에서 배열을 합치는 알고리즘 구현



배열 합치기

https://www.acmicpc.net/problem/11728

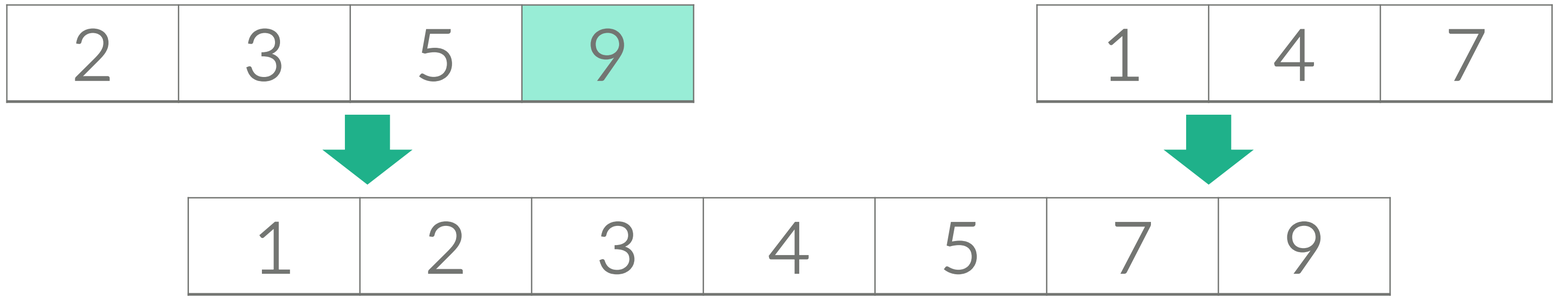
- 머지 소트에서 배열을 합치는 알고리즘 구현



배열 합치기

https://www.acmicpc.net/problem/11728

- 머지 소트에서 배열을 합치는 알고리즘 구현



배열 합치기

https://www.acmicpc.net/problem/11728

- 시간 복잡도: O(|A|+|B|)

배열 합치기

https://www.acmicpc.net/problem/11728

- 소스: http://codeplus.codes/e6498ddc3e5546f2a1609ed175ccb360

퀵 소트

퀵 소트

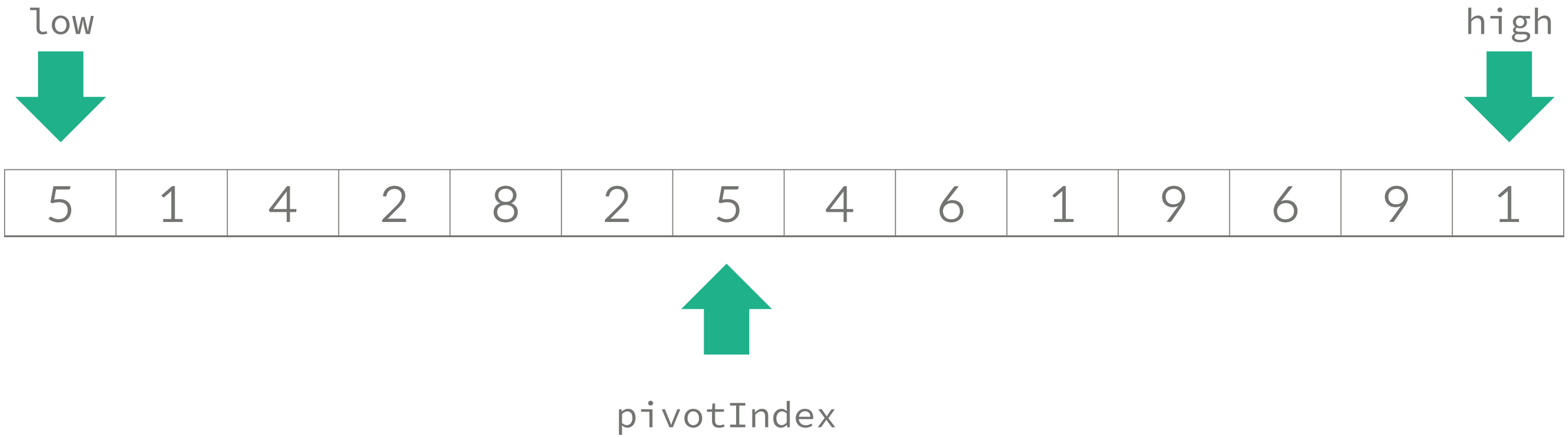
Quick Sort

- 평균적인 상황에서 최고의 성능을 자랑하는 알고리즘
- 피벗(pivot)을 하나 고른 다음, 그것보다 작은 것을 앞으로 큰 것을 뒤로 보낸다.
- 그 다음, 피벗의 앞과 뒤에서 퀵 정렬을 수행한다.
- 최악의 경우에는 $O(N^2)$이 걸린다.

퀵 소트

Quick Sort

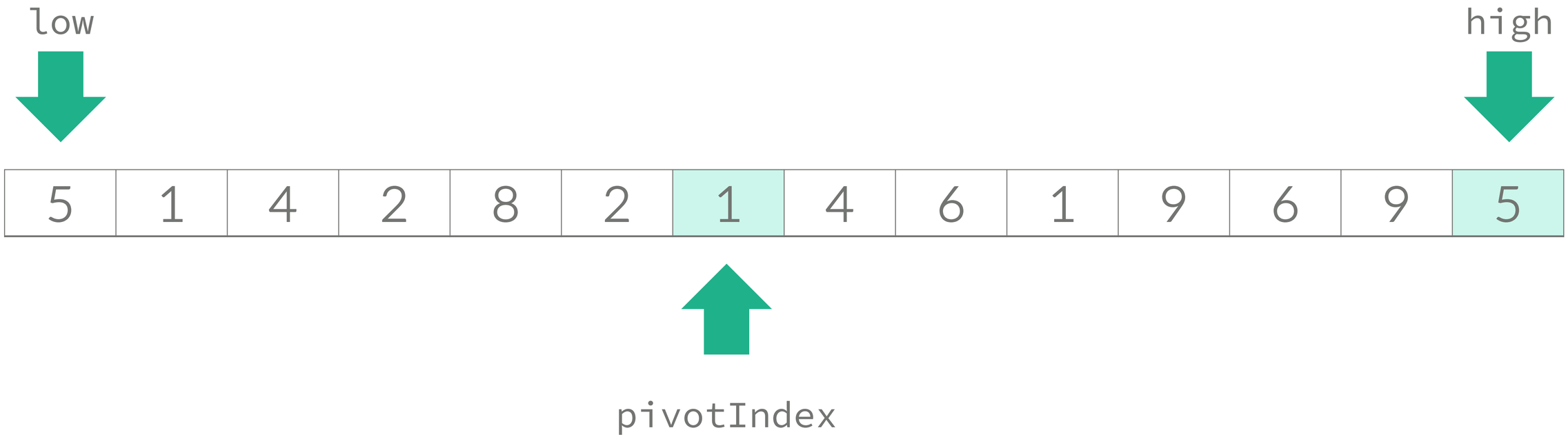
- `pivotValue = 5`



퀵 소트

Quick Sort

- `pivotValue = 5`



퀵 소트

Quick Sort

- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`

`i`

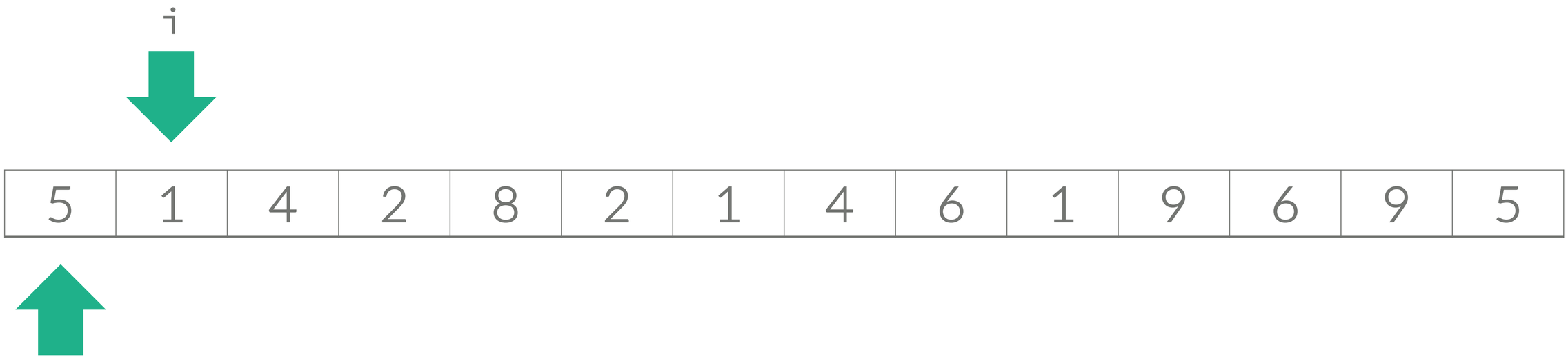
5	1	4	2	8	2	1	4	6	1	9	6	9	5

`storeIndex`

퀵 소트

Quick Sort

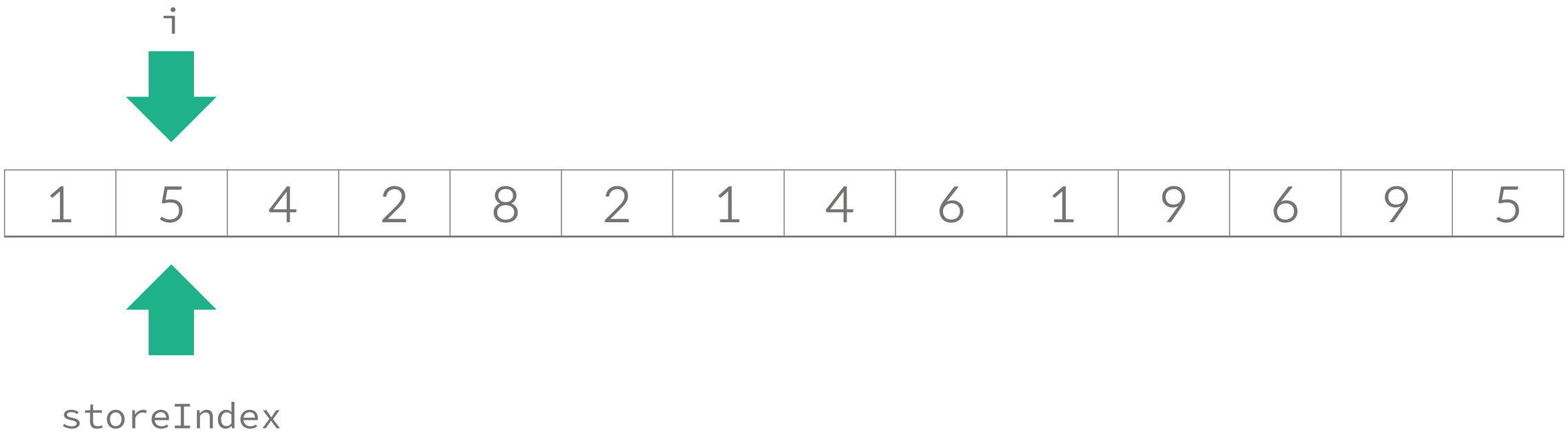
- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`



퀵 소트

Quick Sort

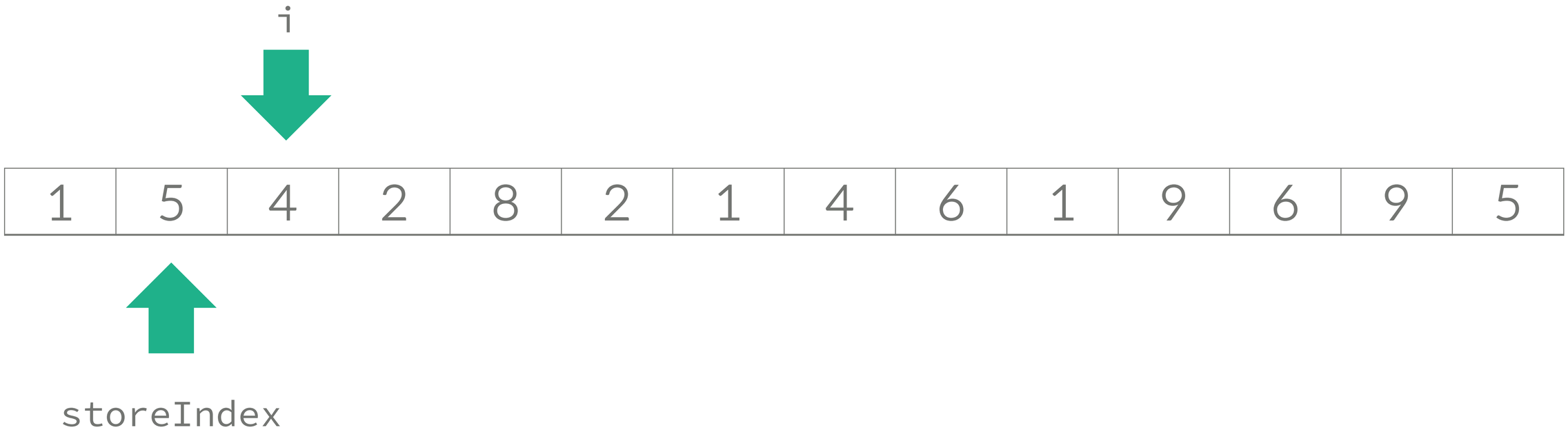
- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`



퀵 소트

Quick Sort

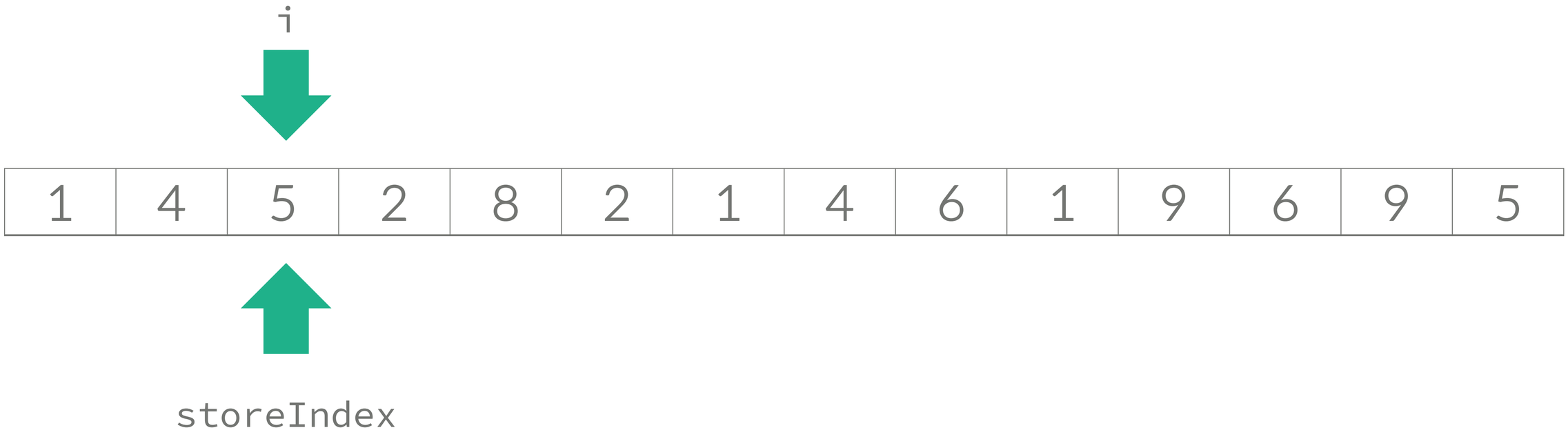
- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`



퀵 소트

Quick Sort

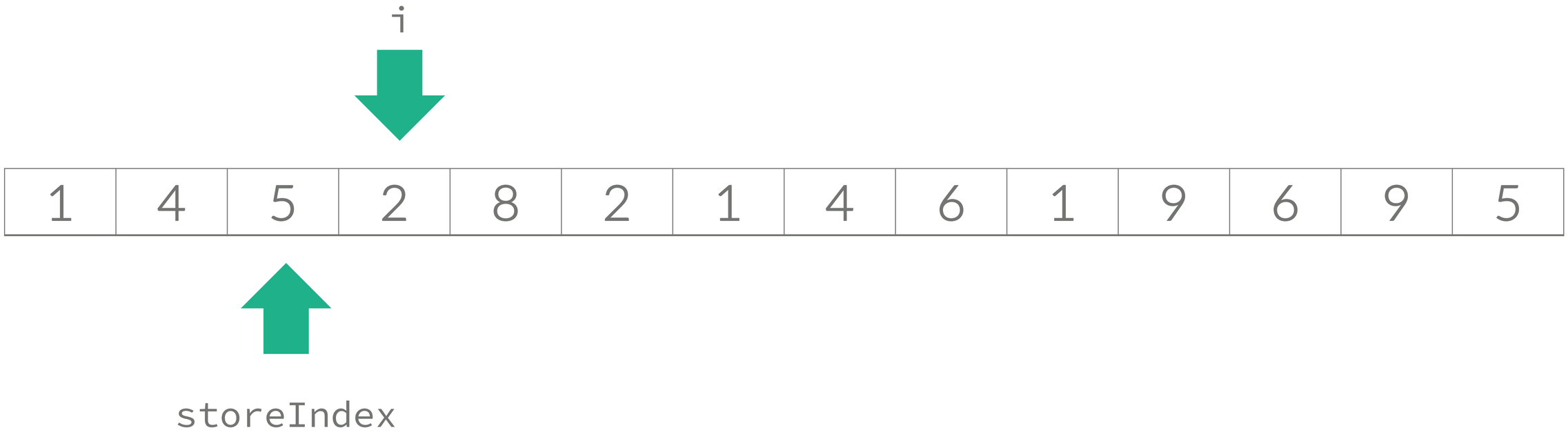
- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`



퀵 소트

Quick Sort

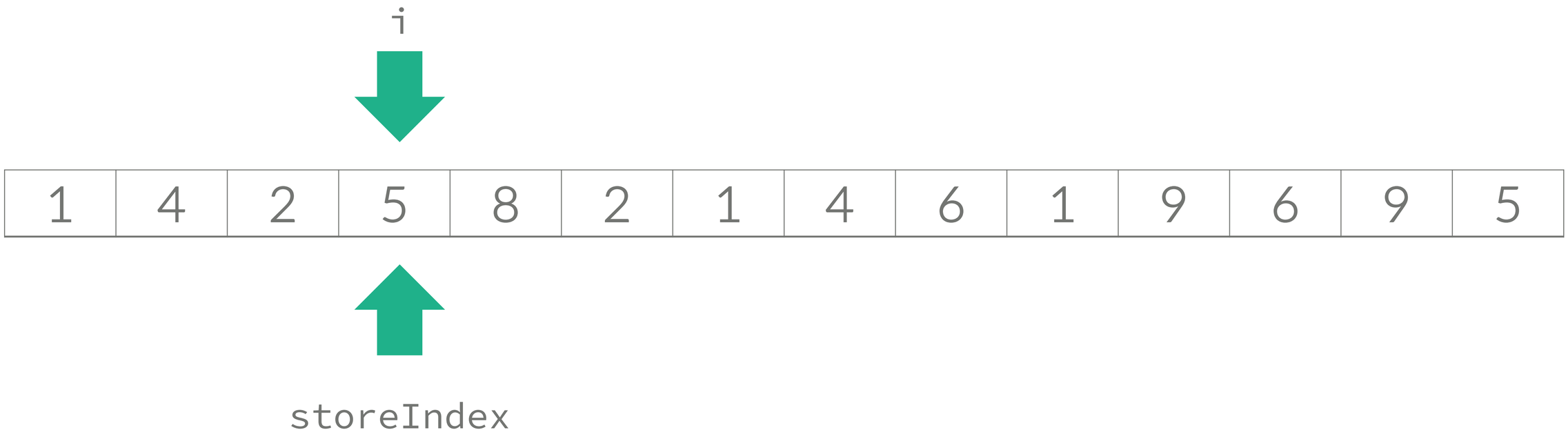
- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`



퀵 소트

Quick Sort

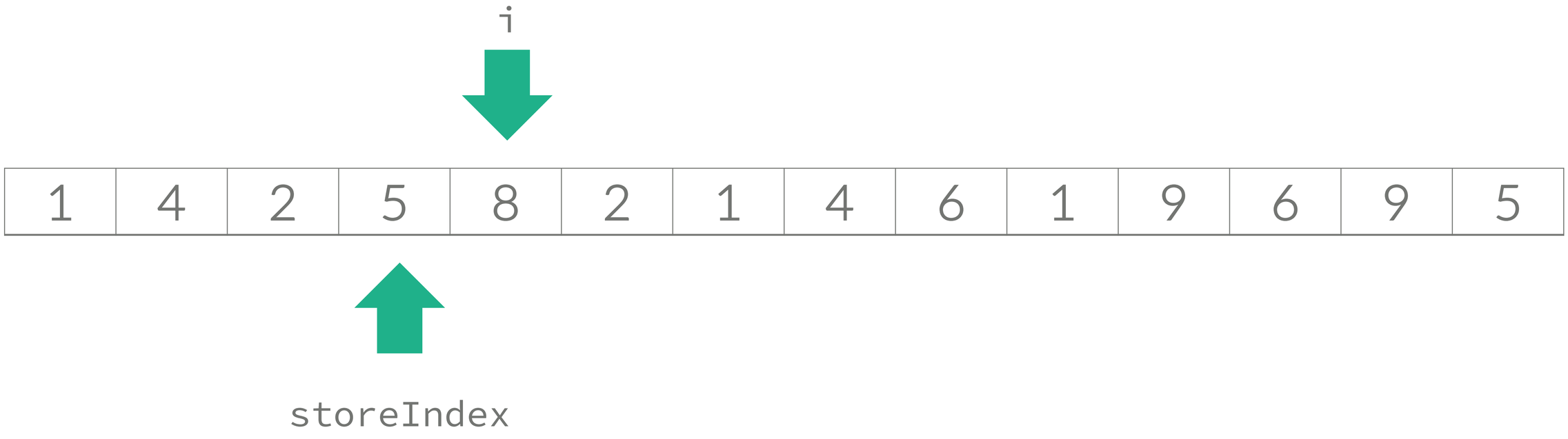
- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`



퀵 소트

Quick Sort

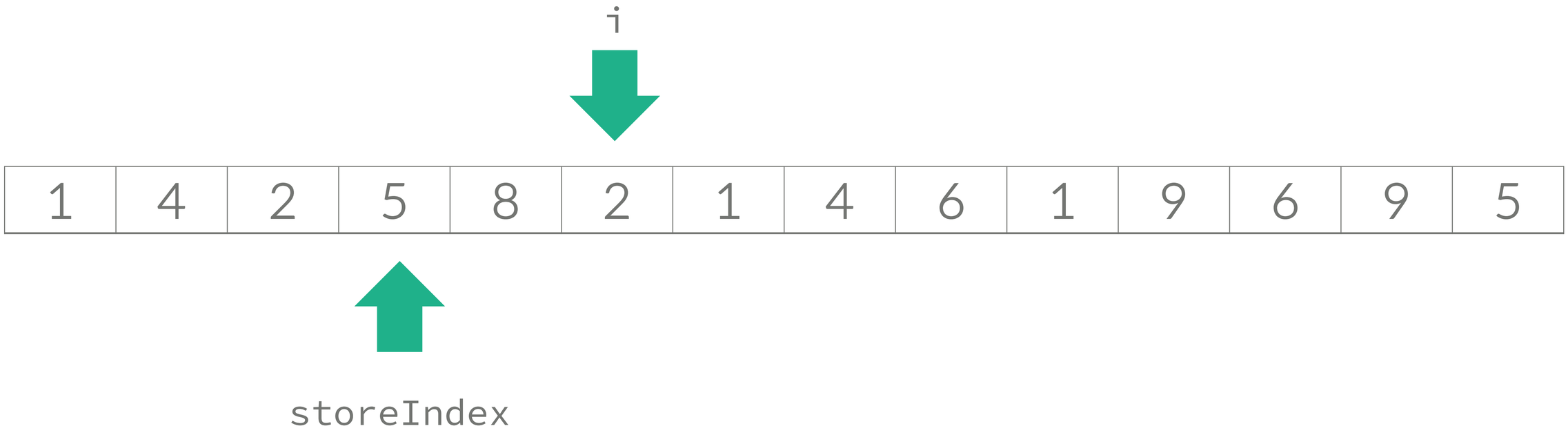
- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`



퀵 소트

Quick Sort

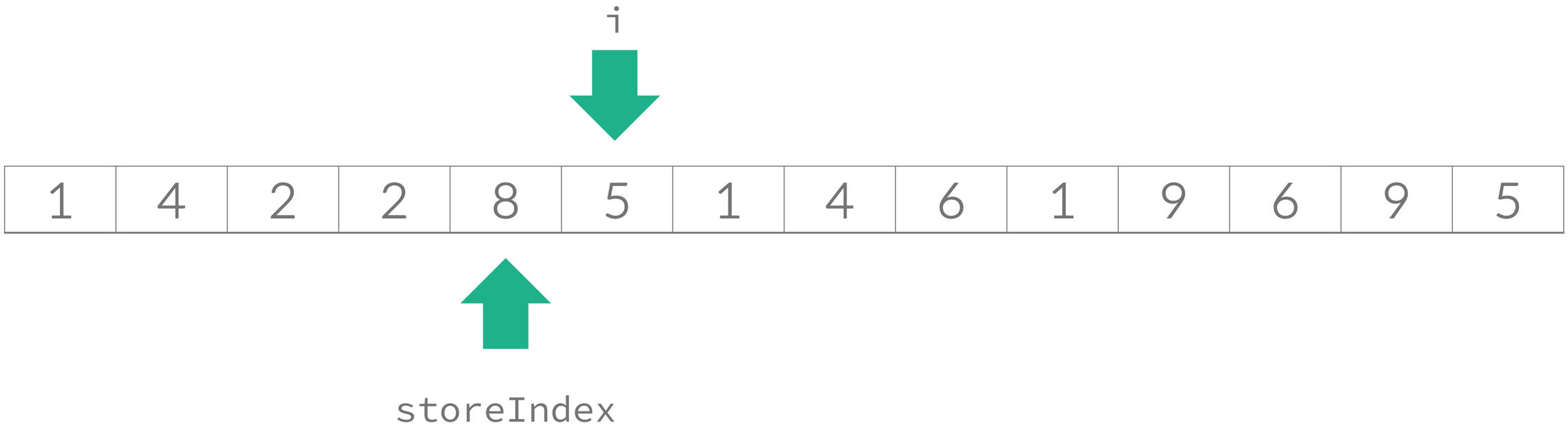
- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`



퀵 소트

Quick Sort

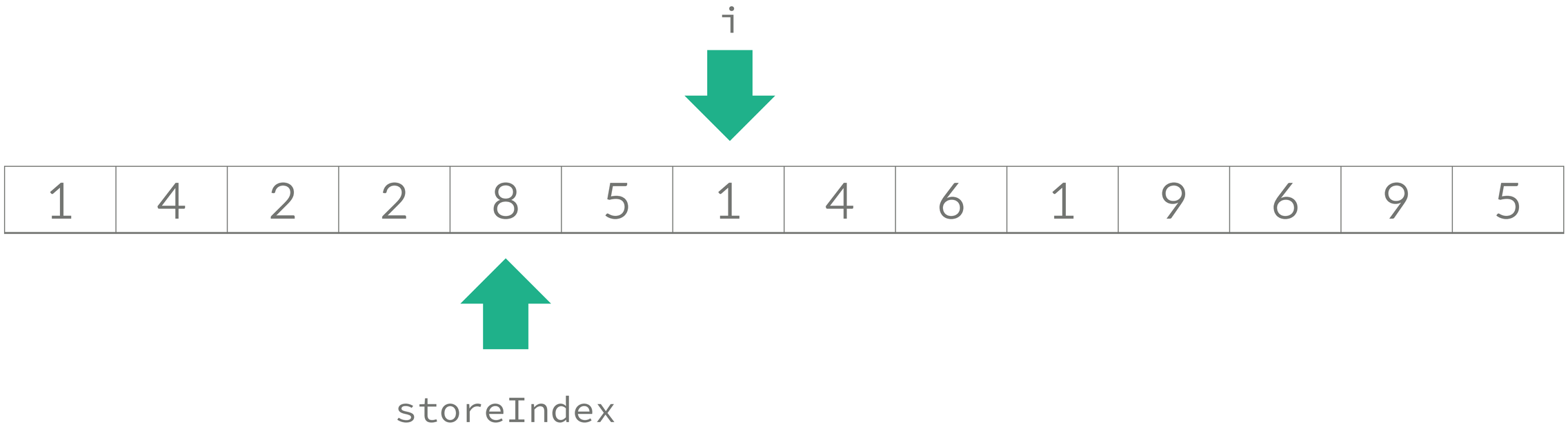
- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`



퀵 소트

Quick Sort

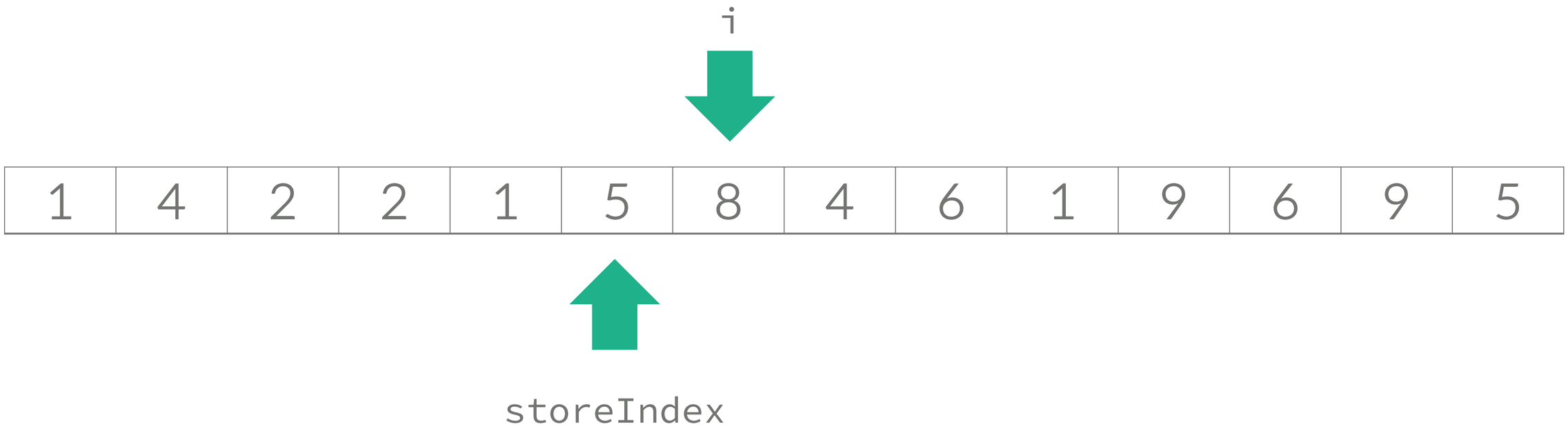
- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`



퀵 소트

Quick Sort

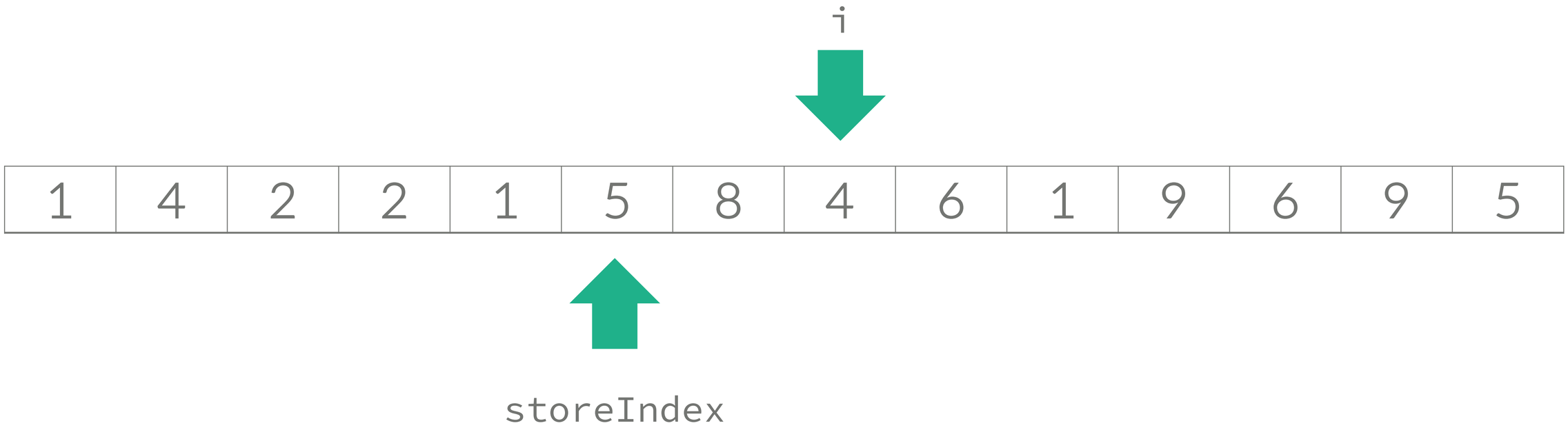
- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`



퀵 소트

Quick Sort

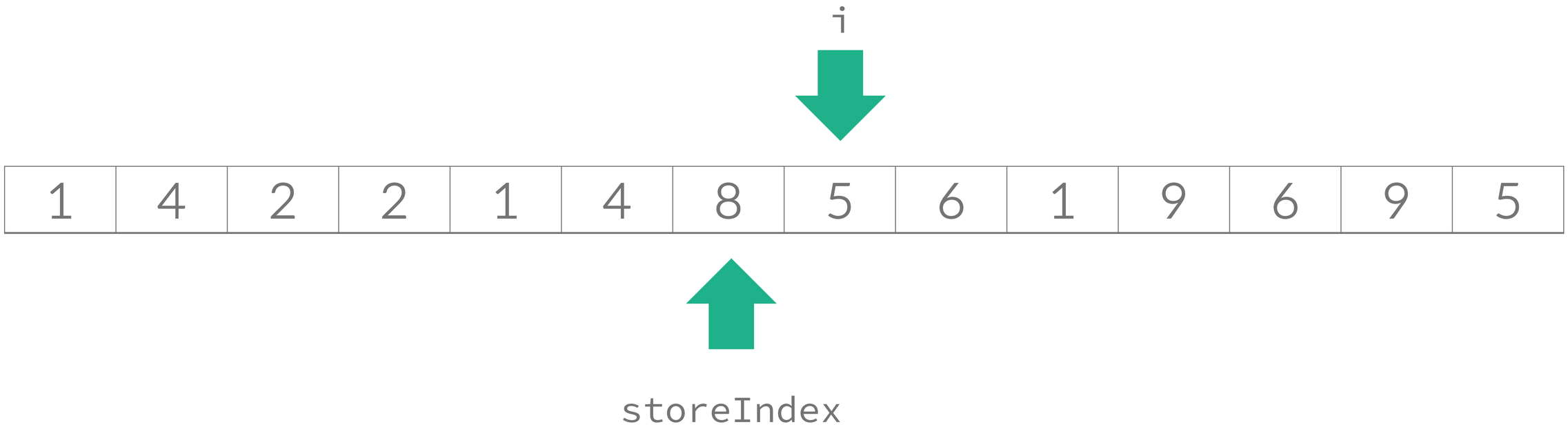
- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`



퀵 소트

Quick Sort

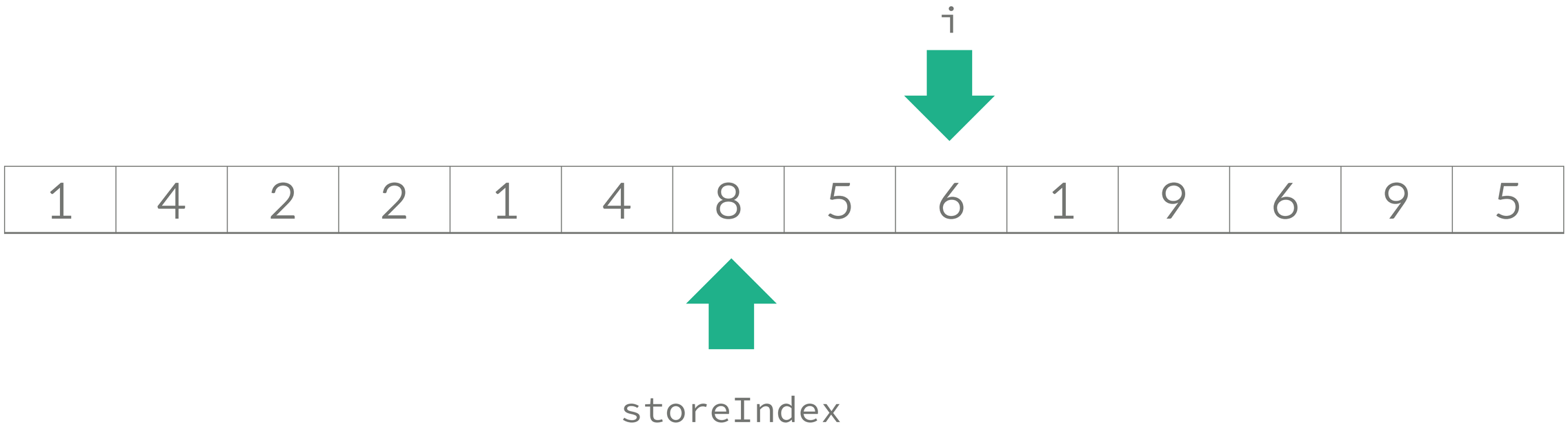
- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`



퀵 소트

Quick Sort

- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`



퀵 소트

Quick Sort

- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`

퀵 소트

Quick Sort

- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`

퀵 소트

Quick Sort

- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`

퀵 소트

Quick Sort

- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`

퀵 소트

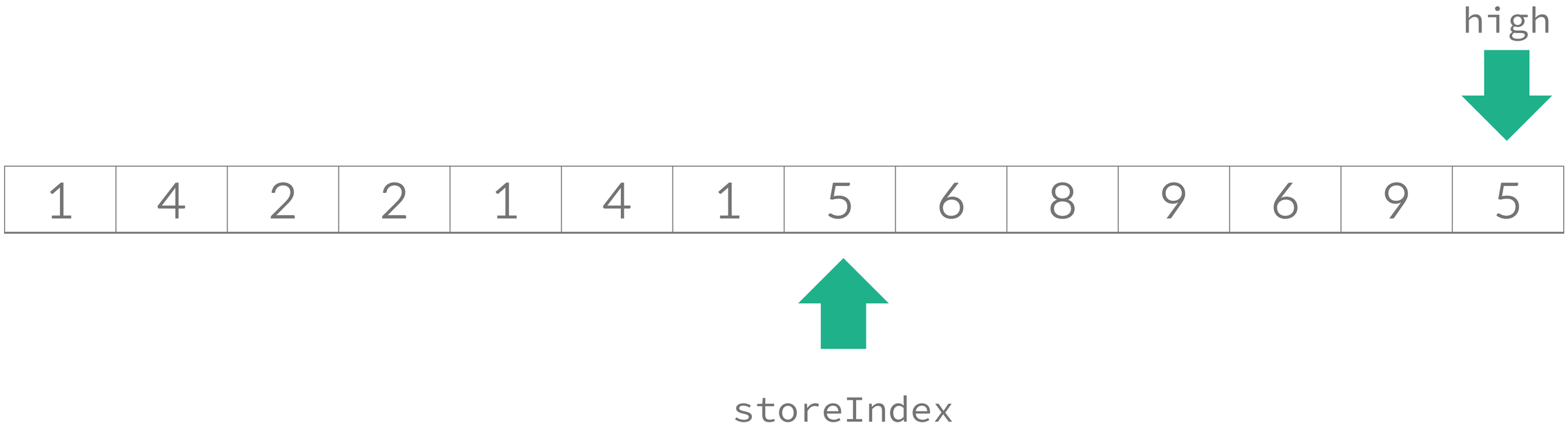
Quick Sort

- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`

퀵 소트

Quick Sort

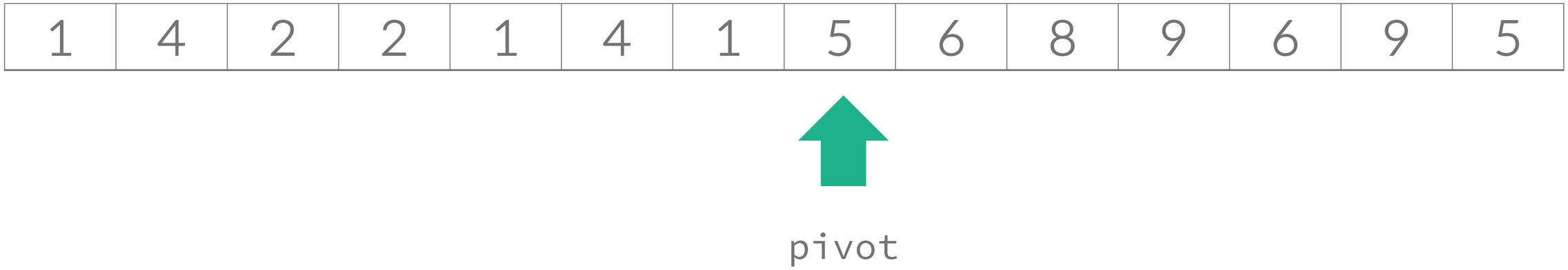
- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`



퀵 소트

Quick Sort

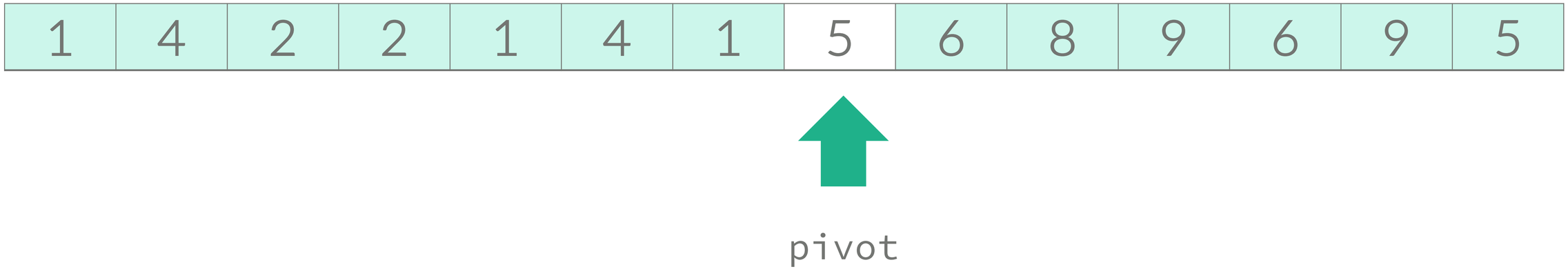
- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`



퀵 소트

Quick Sort

- `pivotValue = 5`
- `if (a[i] < pivotValue) swap(a[i], a[storeIndex]), storeIndex += 1`



퀵 소트

Quick Sort


```
int choosePivot(int low, int high) {
    return low + (high-low)/2;
}
```

퀵 소트

Quick Sort


```
int partition(int low, int high) {
    int pivotIndex = choosePivot(low, high);
    int pivotValue = a[pivotIndex];
    swap(a[pivotIndex], a[high]);
    int storeIndex = low;
    for (int i=low; i<high; i++) {
        if (a[i] < pivotValue) {
            swap(a[i], a[storeIndex]);
            storeIndex += 1;
        }
    }
    swap(a[storeIndex], a[high]);
    return storeIndex;
}
```

퀵 소트

Quick Sort


```
void quicksort(int low, int high) {
    if (low < high) {
        int pivot = partition(low, high);
        quicksort(low, pivot-1);
        quicksort( pivot+1, high);
    }
}
```

퀵 셀렉트

Quick Sort

- 정렬되지 않은 리스트에서 k번째 작은 수를 찾는 알고리즘
- 퀵 소트와 같지만, 한 쪽만 호출한다.
- 따라서, 시간 복잡도가 O(N)으로 줄어들지만, 최악의 경우에는 $O(N^2)$ 이다.

퀵 셀렉트

Quick Sort


```
int quickselect(int low, int high, int k) {
    int pivot = partition(low, high);
    if (pivot == k) {
        return a[k];
    } else if (k < pivot) {
        return quickselect(low, pivot-1, k);
    } else {
        return quickselect(pivot+1, high, k);
    }
}
```