

Sorting II

CS143: lecture 12

Konstantinos Ameranis, July 21

Sorting II

So far...

Sorting II

So far...

- We have seen:

Sorting II

So far...

- We have seen:
 - Selection sort

Sorting II

So far...

- We have seen:
 - Selection sort
 - Insertion sort

Sorting II

So far...

- We have seen:
 - Selection sort
 - Insertion sort
 - Bubble sort

Sorting II

So far...

- We have seen:
 - Selection sort
 - Insertion sort
 - Bubble sort
 - Tree sort

Sorting II

So far...

- We have seen:
 - Selection sort
 - Insertion sort
 - Bubble sort
 - Tree sort
- Two really fast sorting algorithms:

Sorting II

So far...

- We have seen:
 - Selection sort
 - Insertion sort
 - Bubble sort
 - Tree sort
- Two really fast sorting algorithms:
 - Merge sort

Sorting II

So far...

- We have seen:
 - Selection sort
 - Insertion sort
 - Bubble sort
 - Tree sort
- Two really fast sorting algorithms:
 - Merge sort
 - Quick sort

Merge and Quick Sorts

Merge and Quick Sorts

- Divide-and-conquer Algorithm:

Merge and Quick Sorts

- Divide-and-conquer Algorithm:
 1. Divide an array in ~half

Merge and Quick Sorts

- Divide-and-conquer Algorithm:
 1. Divide an array in ~half
 2. Sort both arrays individually

Merge and Quick Sorts

- Divide-and-conquer Algorithm:
 1. Divide an array in ~half
 2. Sort both arrays individually
 3. Combine the two arrays

Merge and Quick Sorts

- Divide-and-conquer Algorithm:

1. Divide an array in ~half
2. Sort both arrays individually
3. Combine the two arrays

Divide

Conquer

Glue

Merge and Quick Sorts

- Divide-and-conquer Algorithm:

1. Divide an array in ~half
2. Sort both arrays individually
3. Combine the two arrays

Divide

Conquer

Glue

- Merge Sort: Easy to divide and hard to glue
- Quick Sort: Hard to divide and easy to glue

Merge Sort

Merge Sort

Merge-Sort(list) :

```
l1 <- sort(first half of list)
```

```
l2 <- sort(second half of list)
```

```
list' <- merge(l1, l2)
```

```
return list'
```

Merge Sort

C code is just as short

```
void merge_sort(int *arr, int start, int end)
{
    int mid = start + (end - start) / 2;

    sort(arr, start, mid);
    sort(arr, mid + 1, end);
    merge(arr, start, mid, end);
}
```

Merge Sort

C code is just as short

```
void merge_sort(int *arr, int start, int end)
{
    int mid = start + (end - start) / 2;

    sort(arr, start, mid);
    sort(arr, mid + 1, end);
    merge(arr, start, mid, end);
}
```

Most of the work is in this merge operation

Merge Operation

- Given two sorted lists, combine them into one sorted list:

[6, 20, 45, 80]

[0, 32, 34, 90]

Merge Operation

- Given two sorted lists, combine them into one sorted list:

[6, 20, 45, 80]

[0, 32, 34, 90]

Compare the two front elements, take the smaller one and move on to the next.

Merge Operation

- Given two sorted lists, combine them into one sorted list:

[6, 20, 45, 80]



[0, 32, 34, 40]



[

Merge Operation

- Given two sorted lists, combine them into one sorted list:

[6, 20, 45, 80]



[0, 32, 34, 40]

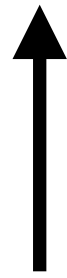


[0]

Merge Operation

- Given two sorted lists, combine them into one sorted list:

[6, 20, 45, 80]



[0, 32, 34, 40]



[0, 6

Merge Operation

- Given two sorted lists, combine them into one sorted list:

[6, 20, 45, 80]



[0, 32, 34, 40]



[0, 6, 20]

Merge Operation

- Given two sorted lists, combine them into one sorted list:

[6, 20, 45, 80]



[0, 32, 34, 40]



[0, 6, 20, 32]

Merge Operation

- Given two sorted lists, combine them into one sorted list:

[6, 20, 45, 80]



[0, 32, 34, 40]



[0, 6, 20, 32, 34]

Merge Operation

- Given two sorted lists, combine them into one sorted list:

[6, 20, 45, 80]



[0, 32, 34, 40]



[0, 6, 20, 32, 34, 40]

Merge Operation

- Given two sorted lists, combine them into one sorted list:

[6, 20, 45, 80]



[0, 32, 34, 40]



[0, 6, 20, 32, 34, 40]

Merge Operation

- Given two sorted lists, combine them into one sorted list:

[6, 20, 45, 80]

[0, 32, 34, 40]



When one list is empty, we can just move the remaining elements of the other list down.

[0, 6, 20, 32, 34, 40]

Merge Operation

- Given two sorted lists, combine them into one sorted list:

[6, 20, 45, 80]



[0, 32, 34, 40]



[0, 6, 20, 32, 34, 40, 45, 80]

Merge Operation

Pseudo*-code

```
int *merge(int *arr1, int len1, int *arr2, int len2)
{
    int *arr = /* allocate */;

    int front1 = 0, front2 = 0, curr_idx = start;

    while (front1 < len1 && front2 < len2) {
        if (arr1[front1] <= arr2[front2]) {
            arr[curr_idx++] = arr1[front1++];
        } else {
            arr[curr_idx++] = arr2[front2++];
        }
    }

    while (front1 < len1) {
        arr[curr_idx++] = arr1[front1++];
    }

    while (front2 < len2) {
        arr[curr_idx++] = arr2[front2++];
    }

    return arr;
}
```

Merge Operation

Pseudo*-code

While both lists are non-empty, move the smaller one to arr.

```
int *merge(int *arr1, int len1, int *arr2, int len2)
{
    int *arr = /* allocate */;

    int front1 = 0, front2 = 0, curr_idx = start;

    while (front1 < len1 && front2 < len2) {
        if (arr1[front1] <= arr2[front2]) {
            arr[curr_idx++] = arr1[front1++];
        } else {
            arr[curr_idx++] = arr2[front2++];
        }
    }

    while (front1 < len1) {
        arr[curr_idx++] = arr1[front1++];
    }

    while (front2 < len2) {
        arr[curr_idx++] = arr2[front2++];
    }

    return arr;
}
```

Merge Operation

Pseudo*-code

While both lists are non-empty, move the smaller one to arr.

Merge all the remaining elements. (Only one of the two loops will happen in one run)

```
int *merge(int *arr1, int len1, int *arr2, int len2)
{
    int *arr = /* allocate */;

    int front1 = 0, front2 = 0, curr_idx = start;

    while (front1 < len1 && front2 < len2) {
        if (arr1[front1] <= arr2[front2]) {
            arr[curr_idx++] = arr1[front1++];
        } else {
            arr[curr_idx++] = arr2[front2++];
        }
    }

    while (front1 < len1) {
        arr[curr_idx++] = arr1[front1++];
    }

    while (front2 < len2) {
        arr[curr_idx++] = arr2[front2++];
    }

    return arr;
}
```

Merge Operation

Actual code

```
void merge(int *arr, int start, int mid, int end)
{
    int len1 = mid - start;
    int len2 = end - mid;

    if (arr[mid - 1] <= arr[mid]) {
        return;
    }

    int *arr1 = malloc(len1 * sizeof(int));
    int *arr2 = malloc(len2 * sizeof(int));

    memcpy(arr1, &arr[start], len1 * sizeof(int));
    memcpy(arr2, &arr[mid], len2 * sizeof(int));

    <+= code from previous slide >

    free(arr1);
    free(arr2);
}
```

Merge Sort

```
void merge(int *arr, int start, int mid, int end);
```

```
void merge_sort(int *arr, int start, int end)
{
    int mid = start + (end - start) / 2;

    sort(arr, start, mid);
    sort(arr, mid + 1, end);
    merge(arr, start, mid, end);
}
```

Merge Sort

```
void merge(int *arr, int start, int mid, int end);
```

```
void merge_sort(int *arr, int start, int end)
```

```
{  
    int mid = start + (end - start) / 2;
```

```
    sort(arr, start, mid);
```

```
    sort(arr, mid + 1, end);
```

```
    merge(arr, start, mid, end);
```

```
}
```

What sorting algorithms should we use here?

Merge Sort

```
void merge(int *arr, int start, int mid, int end);
```

```
void merge_sort(int *arr, int start, int end)
{
    int mid = start + (end - start) / 2;

    merge_sort(arr, start, mid);
    merge_sort(arr, mid + 1, end);
    merge(arr, start, mid, end);
}
```

Merge Sort

```
void merge(int *arr, int start, int mid, int end);
```

```
void merge_sort(int *arr, int start, int end)
```

```
{  
    int mid = start + (end - start) / 2;
```

```
    merge_sort(arr, start, mid);  
    merge_sort(arr, mid + 1, end);  
    merge(arr, start, mid, end);
```

```
}
```

Just use itself!

Merge Sort

```
void merge_sort(int *arr, int start, int end)
{
    if (end - start <= 1) {
        return;
    }

    int mid = start + (end - start) / 2;

    merge_sort(arr, start, mid);
    merge_sort(arr, mid, end);
    merge(arr, start, mid, end);
}
```

Merge Sort

```
void merge_sort(int *arr, int start, int end)
{
    if (end - start <= 1) {
        return;
    }

    int mid = start + (end - start) / 2;

    merge_sort(arr, start, mid);
    merge_sort(arr, mid, end);
    merge(arr, start, mid, end);
}
```

Base case: one-element and zero-element lists are always sorted.

Merge Sort

```
void merge_sort(int *arr, int start, int end)
{
    if (end - start <= 1) {
        return;
    }

    int mid = start + (end - start) / 2;

    merge_sort(arr, start, mid);
    merge_sort(arr, mid, end);
    merge(arr, start, mid, end);
}
```

Base case: one-element and zero-element lists are always sorted.

Let's see this in action

Divide

[99, 5, 80, 29, 48, 20, 4, 25, 0]

Divide

[99, 5, 80, 29, 48, 20, 4, 25, 0]

[99, 5, 80, 29]

[48, 20, 4, 25, 0]

Divide

[99, 5, 80, 29, 48, 20, 4, 25, 0]

[99, 5, 80, 29]

[48, 20, 4, 25, 0]

[99, 5]

[80, 29]

[48, 20]

[4, 25, 0]

Divide

[99, 5, 80, 29, 48, 20, 4, 25, 0]

[99, 5, 80, 29]

[48, 20, 4, 25, 0]

[99, 5]

[80, 29]

[48, 20]

[4, 25, 0]

[99]

[5]

[80]

[29]

[48]

[20]

[4]

[25, 0]

Divide

[99, 5, 80, 29, 48, 20, 4, 25, 0]

[99, 5, 80, 29]

[48, 20, 4, 25, 0]

[99, 5]

[80, 29]

[48, 20]

[4, 25, 0]

[99] [5] [80] [29] [48] [20] [4] [25, 0]

[25] [0]

Divide

[99, 5, 80, 29, 48, 20, 4, 25, 0]

[99, 5, 80, 29]

[48, 20, 4, 25, 0]

[99, 5]

[80, 29]

[48, 20]

[4, 25, 0]

[99] [5] [80] [29] [48] [20] [4] [25, 0]

[25] [0]

Look! The lists are all ""sorted.""

Merge

[99]

[5]

[80]

[29]

[48]

[20]

[4]

[25]

[0]

Merge

[5, 99]

[29, 80]

[99]

[5]

[80]

[29]

[48]

[20]

[4]

[25]

[0]

Merge

[5, 29, 80, 99]

[5, 99]

[29, 80]

[99]

[5]

[80]

[29]

[48]

[20]

[4]

[25]

[0]

Merge

[5, 29, 80, 99]

[5, 99]

[29, 80]

[20, 48]

[99]

[5]

[80]

[29]

[48]

[20]

[4]

[25]

[0]

Merge

[5, 29, 80, 99]

[5, 99]

[29, 80]

[20, 48]

[99]

[5]

[80]

[29]

[48]

[20]

[4]

[0, 25]

[25]

[0]

Merge

[5, 29, 80, 99]

[5, 99]

[29, 80]

[20, 48]

[0, 4, 25]

[99]

[5]

[80]

[29]

[48]

[20]

[4]

[0, 25]

[25]

[0]

Merge

[5, 29, 80, 99]

[0, 4, 20, 25, 48]

[5, 99]

[29, 80]

[20, 48]

[0, 4, 25]

[99]

[5]

[80]

[29]

[48]

[20]

[4]

[0, 25]

[25]

[0]

Merge

[0, 4, 5, 20, 25, 29, 48, 80, 99]

[5, 29, 80, 99]

[0, 4, 20, 25, 48]

[5, 99]

[29, 80]

[20, 48]

[0, 4, 25]

[99]

[5]

[80]

[29]

[48]

[20]

[4]

[0, 25]

[25]

[0]

Merge

[0, 4, 5, 20, 25, 29, 48, 80, 99] Done!

[5, 29, 80, 99]

[0, 4, 20, 25, 48]

[5, 99]

[29, 80]

[20, 48]

[0, 4, 25]

[99]

[5]

[80]

[29]

[48]

[20]

[4]

[0, 25]

[25]

[0]

Merge Sort

Time Complexity

Merge Sort

Time Complexity

- What is the time complexity for **merge** (A_1 , A_2) ?

Merge Sort

Time Complexity

- What is the time complexity for **merge** ($A1$, $A2$) ?
- We merge one element from either arrays one at a time.

Merge Sort

Time Complexity

- What is the time complexity for **merge** ($A1$, $A2$) ?
- We merge one element from either arrays one at a time.
 - $O(\text{len}(A1) + \text{len}(A2))$

Merge Sort

[99, 5, 80, 29, 48, 20, 4, 25, 0]

[99, 5, 80, 29]

[48, 20, 4, 25, 0]

[99, 5]

[80, 29]

[48, 20]

[4, 25, 0]

[99]

[5]

[80]

[29]

[48]

[20]

[4]

[25, 0]

[25]

[0]

At each level, we perform **merge** on all elements from the original list.

We thus spend $O(n)$ on **merge** at each level.

Merge Sort

[99, 5, 80, 29, 48, 20, 4, 25, 0]

[99, 5, 80, 29]

[48, 20, 4, 25, 0]

[99, 5]

[80, 29]

[48, 20]

[4, 25, 0]

[99] [5] [80] [29] [48] [20] [4] [25, 0]

[25] [0]

How many levels are there in total?

Merge Sort

[99, 5, 80, 29, 48, 20, 4, 25, 0]

[99, 5, 80, 29]

[48, 20, 4, 25, 0]

[99, 5]

[80, 29]

[48, 20]

[4, 25, 0]

[99] [5] [80] [29] [48] [20] [4] [25, 0]

[25] [0]

How many levels are there in total?
 $O(\log_2 n)$

Merge Sort

Complexity

Merge Sort

Complexity

- $O(n)$ work (on **merge**) at each level

Merge Sort

Complexity

- $O(n)$ work (on **merge**) at each level
- $O(\log n)$ levels

Merge Sort

Complexity

- $O(n)$ work (on **merge**) at each level
- $O(\log n)$ levels
- Overall time complexity: $O(n \log n)$

Merge Sort

Complexity

- $O(n)$ work (on **merge**) at each level
- $O(\log n)$ levels
- Overall time complexity: $O(n \log n)$
- Space complexity:

Merge Sort

Complexity

- $O(n)$ work (on **merge**) at each level
- $O(\log n)$ levels
- Overall time complexity: $O(n \log n)$
- Space complexity:
 - Standard implementation uses temporary arrays during merging:

Merge Sort

Complexity

- $O(n)$ work (on **merge**) at each level
- $O(\log n)$ levels
- Overall time complexity: $O(n \log n)$
- Space complexity:
 - Standard implementation uses temporary arrays during merging:
 - $O(n)$ space

Merge Sort

Complexity

- $O(n)$ work (on **merge**) at each level
- $O(\log n)$ levels
- Overall time complexity: $O(n \log n)$
- Space complexity:
 - Standard implementation uses temporary arrays during merging:
 - $O(n)$ space
 - Can be done in-place, but bookkeeping erodes the overall time.

Merge and Quick Sorts

- Divide-and-conquer Algorithm:

1. Divide an array in ~half
2. Sort both arrays individually
3. Combine the two arrays

Divide

Conquer

Glue

- Merge Sort:

- **Divide:** Easy: just halve the list
- **Glue:** Hard: merge sorted lists

Quick Sort

Quick Sort

1. Select a **pivot** element:
 - First, last, random...
2. **Partition** the array into two sub-arrays:
 - Elements less than the **pivot**, and elements greater than the **pivot**
3. Sort each sub-array
4. Concatenate: first sub-array, the **pivot**, second sub-array
 - The step can be skipped if we do everything in-place in the original array.

Quick Sort

[6, 20, 3, 2, 45, 80]
↑
pivot

Quick Sort

[6, 20, 3, 2, 45, 80]



pivot

[
Less than pivot

[
Greater than pivot

Quick Sort

[6, 20, 3, 2, 45, 80]

↑ ↑
pivot

[
Less than pivot

[20,
Greater than pivot

Quick Sort

[6, 20, 3, 2, 45, 80]

↑
pivot

[3,
Less than pivot

[20,
Greater than pivot

Quick Sort

[6, 20, 3, 2, 45, 80]

↑
pivot



[3, 2,
Less than pivot

[20,
Greater than pivot

Quick Sort

[6, 20, 3, 2, 45, 80]

↑ ↑

pivot

[3, 2,
Less than pivot

[20, 45,
Greater than pivot

Quick Sort

[6, 20, 3, 2, 45, 80]

↑
pivot



[3, 2]

Less than pivot

[20, 45, 80]

Greater than pivot

Quick Sort

[6, 20, 3, 2, 45, 80]

[3, 2] [6] [20, 45, 80]

Quick Sort

[6, 20, 3, 2, 45, 80]

[3, 2]

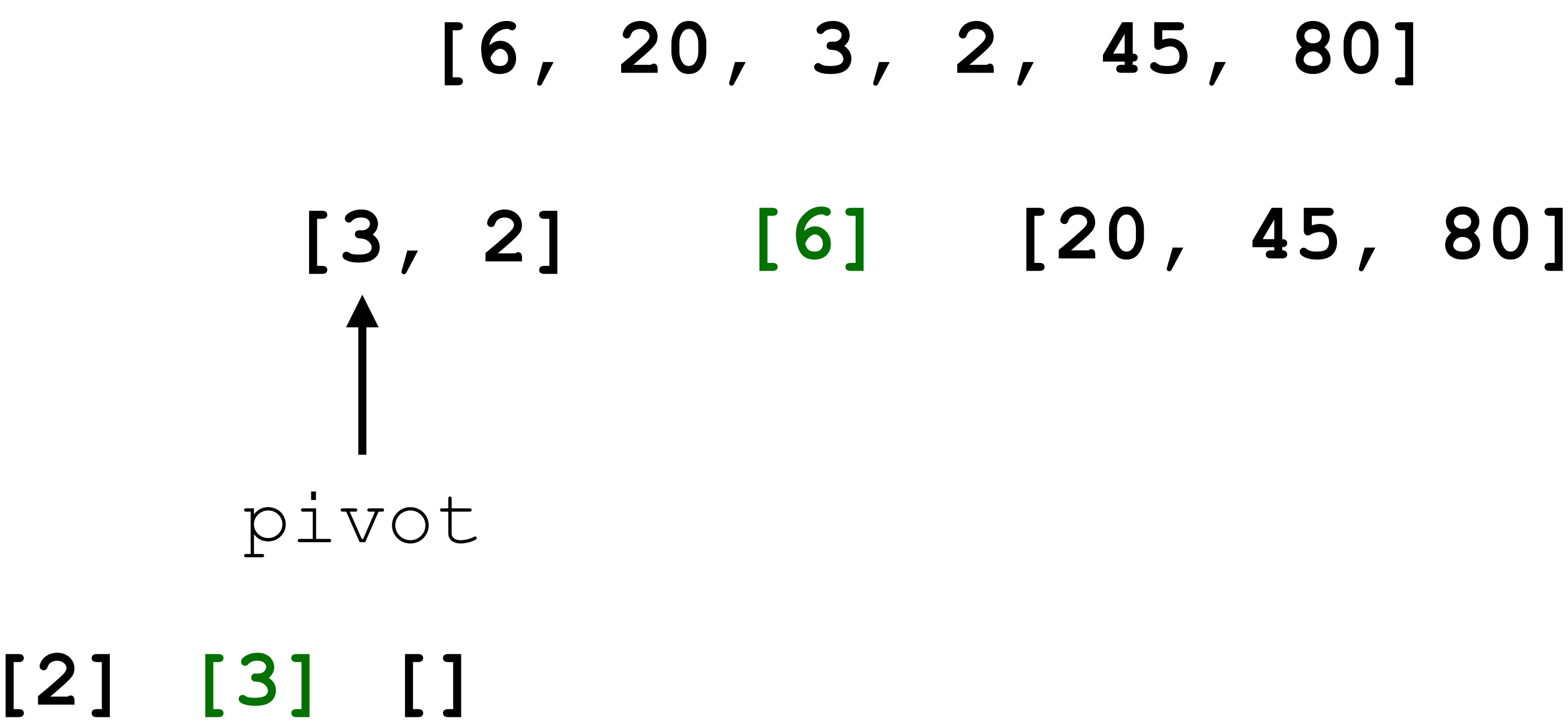
[6]

[20, 45, 80]

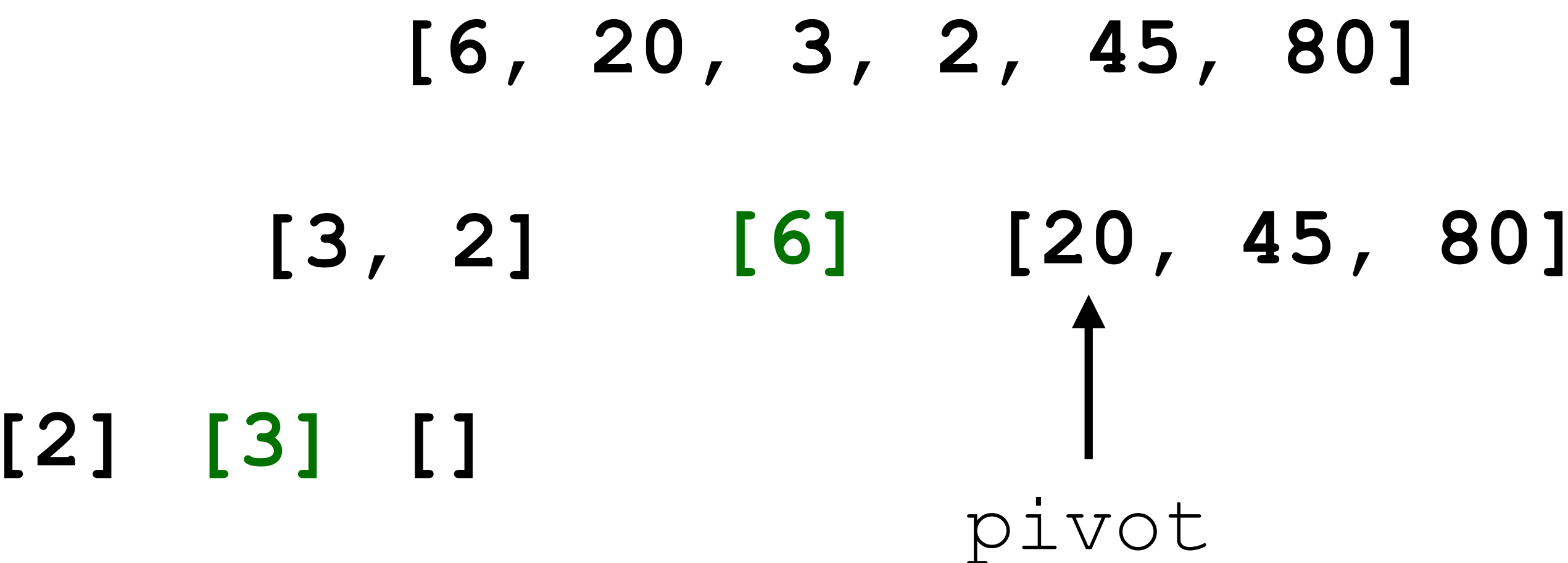
pivot



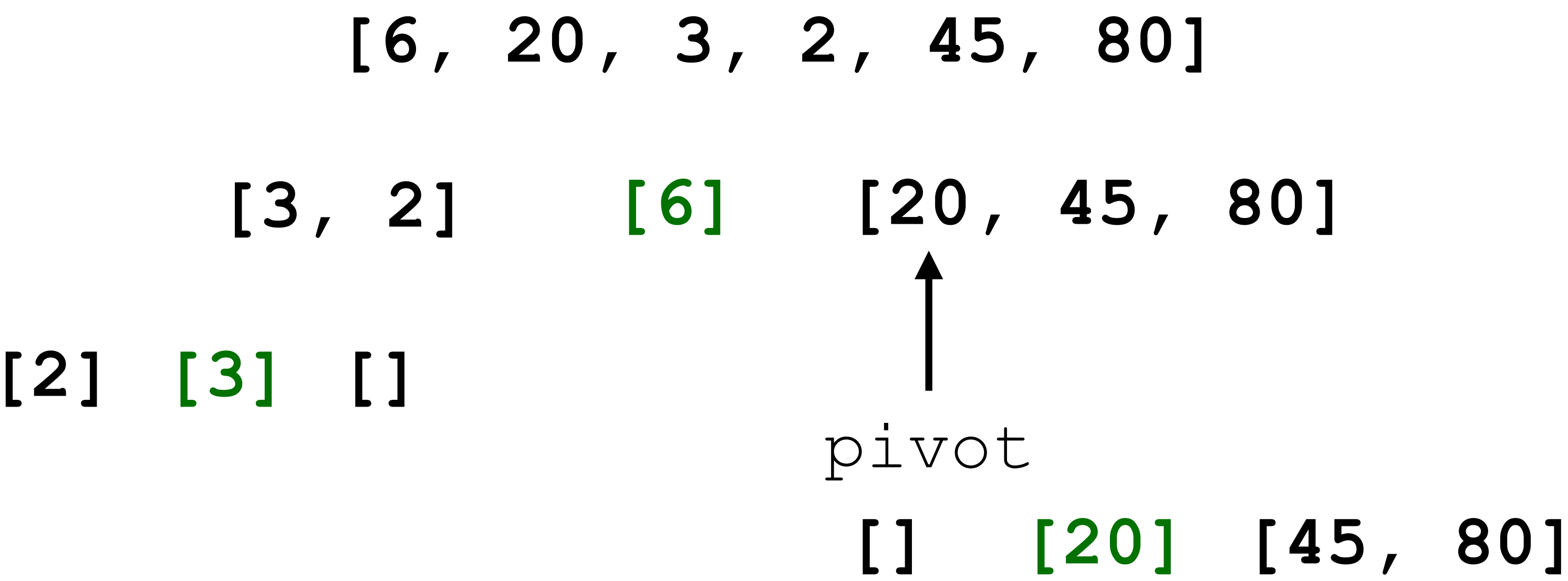
Quick Sort



Quick Sort



Quick Sort



Quick Sort

[6, 20, 3, 2, 45, 80]

[3, 2] [6] [20, 45, 80]

[2] [3] [] [] [20] [45, 80]

↑
pivot

Quick Sort

[6, 20, 3, 2, 45, 80]

[3, 2] [6] [20, 45, 80]

[2] [3] [] [] [20] [45, 80]

 [] [45] [80]

Quick Sort

[6 , 20 , 3 , 2 , 45 , 80]

[2] [3] [] [6] [] [20] [] [45] [80]

Quick Sort

[6, 20, 3, 2, 45, 80]

[2] [3] [] [6] [] [20] [] [45] [80]

Concatenate:

[2, 3, 6, 20, 45, 80]

Quick Sort

Partition

- Input: `arr`, `start` (start index), `end` (end index)
- Action: choose a pivot, rearrange array as: [`<= pivot`, `pivot`, `> pivot`]
- Output: index of pivot in the rearranged array

Quick Sort

Partition

```
int partition(int *arr, int start, int end);
```

- Plan:
 - Keep track of a partition point (**next**)
 - Sweep across the array from right to left:
`for i = end - 1 downto start + 1`
 - If `arr[i] > pivot`: put `arr[i]` right of the partition point (**next**) and advance the partition point
 - Otherwise: put `arr[i]` left of the partition point
 - Swap pivot with the partition point **next**

Quick Sort

Partition

```
int partition(int *arr, int start, int end)
{
    int pivot = arr[start];
    int next = end - 1;

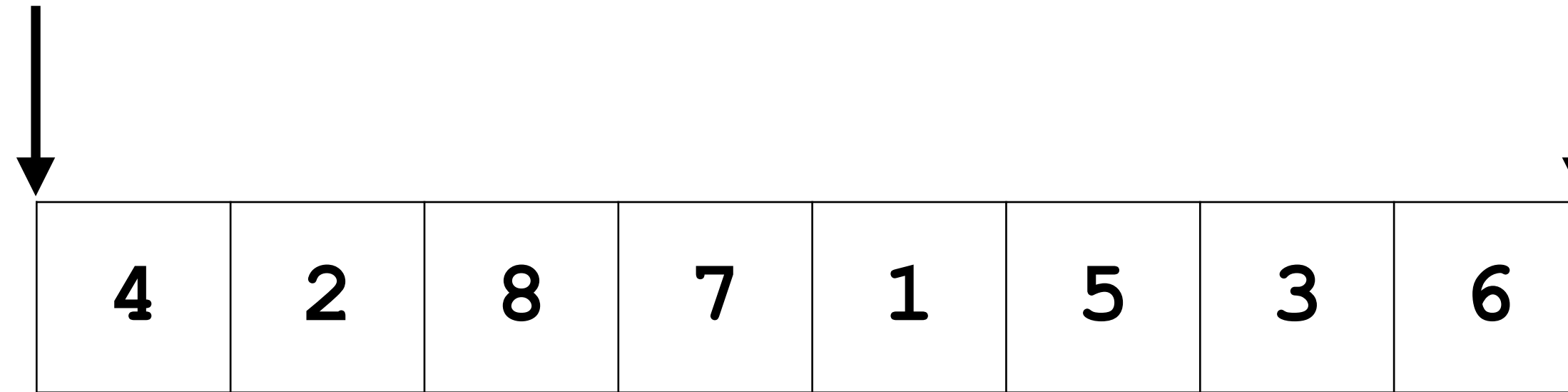
    for (int i = end - 1; i > start; i--) {
        if (arr[i] > pivot) {
            swap(arr, next--, i);
        }
    }

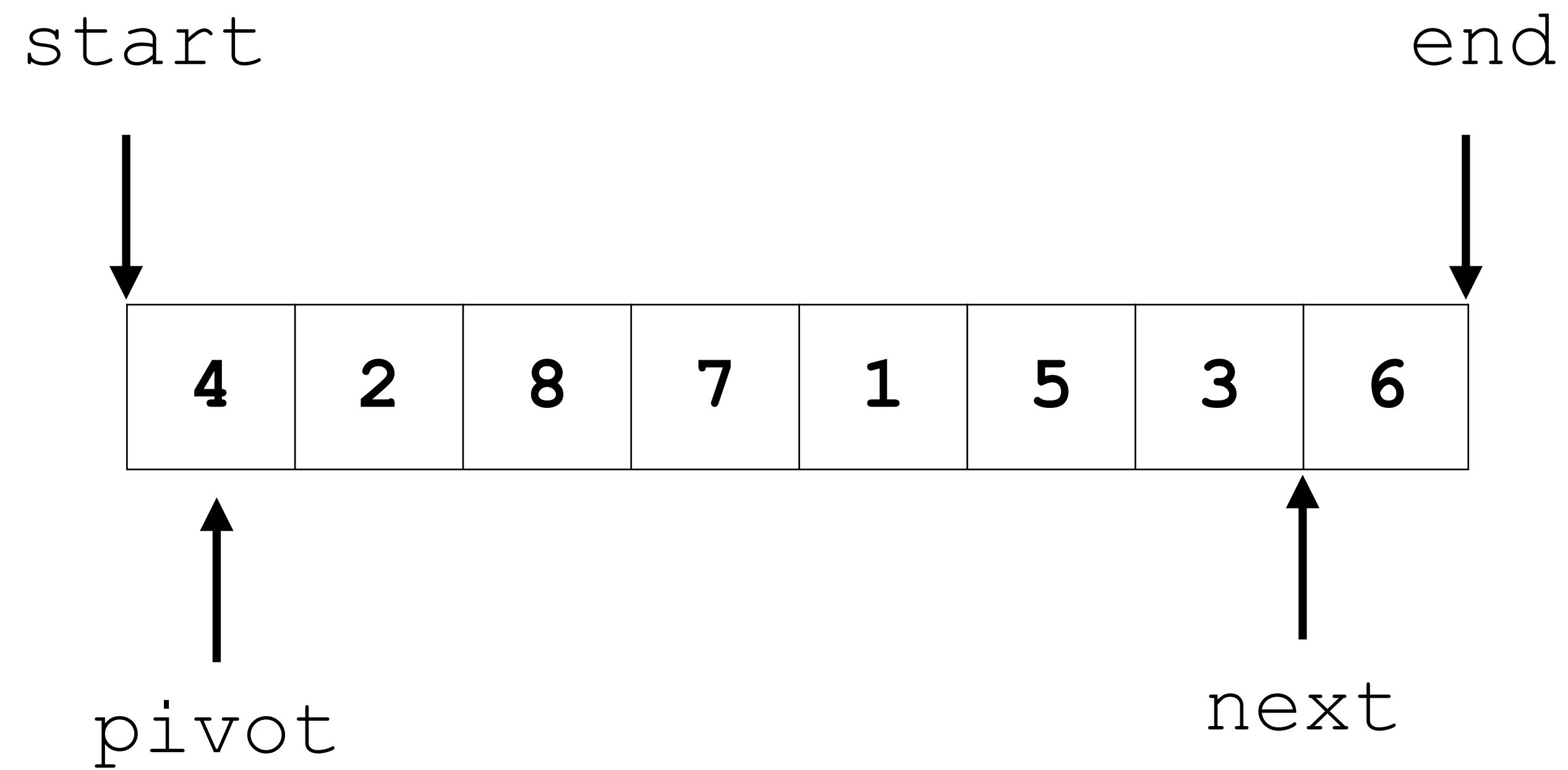
    swap(arr, start, next);
    return next;
}
```

4	2	8	7	1	5	3	6
---	---	---	---	---	---	---	---

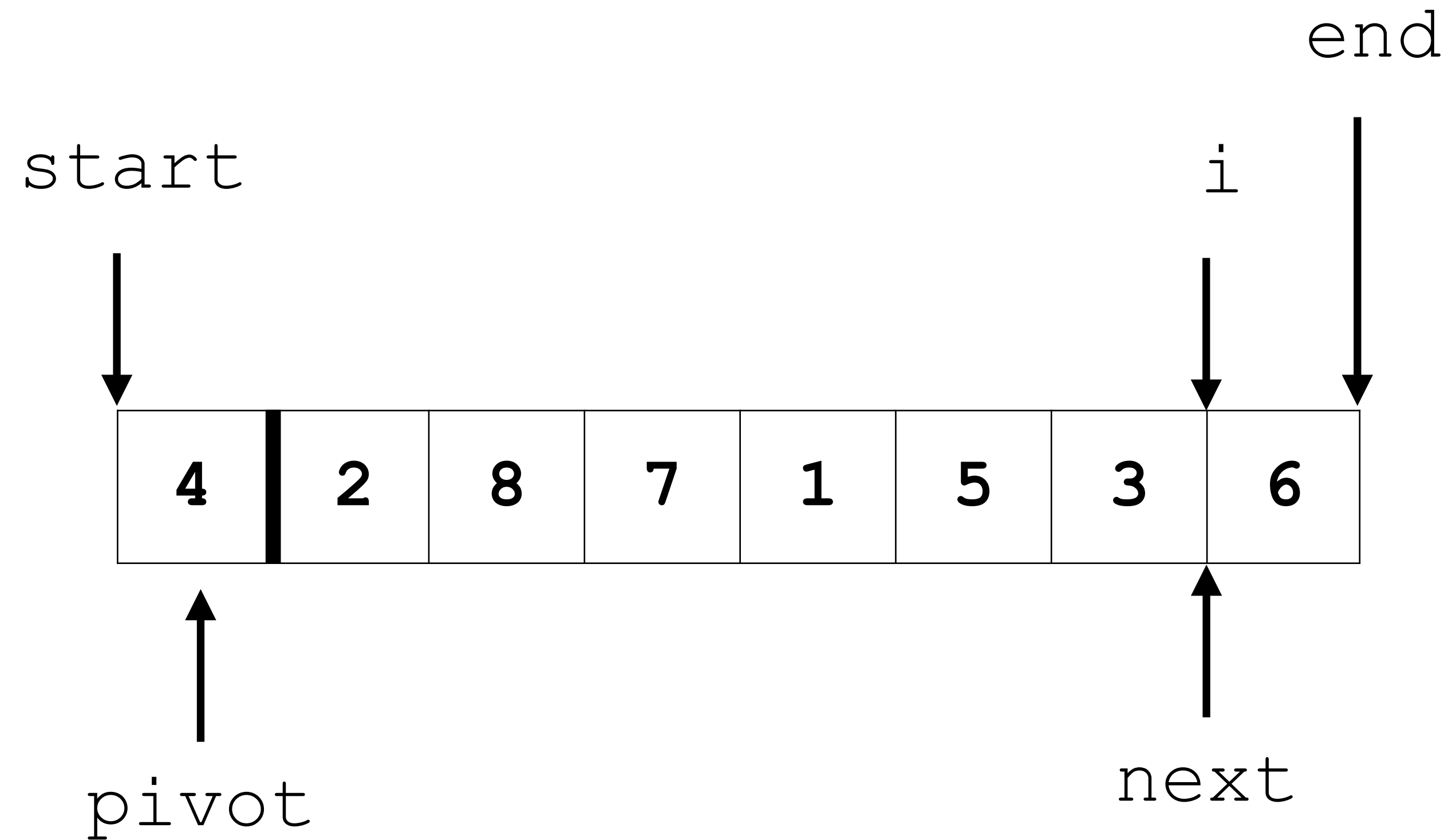
start

end





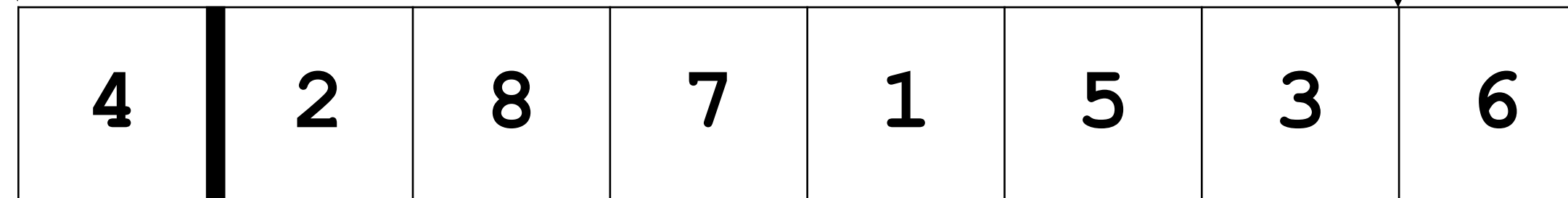
```
int pivot = arr[start];  
int next = end - 1;
```



```
for (int i = end - 1; i > start; i--)  
    if (arr[i] > pivot)  
        swap(arr, next--, i);
```

```
arr[i] > pivot?    -- yes  
swap(arr, next, right)
```

start



pivot



i



end



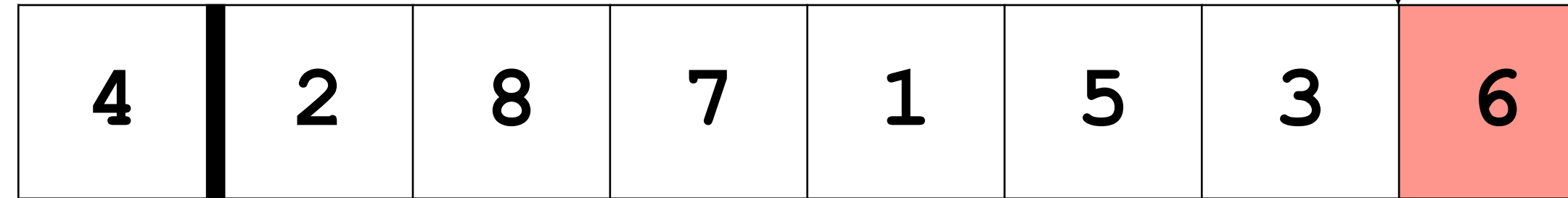
next



```
for (int i = end - 1; i > start; i--)  
    if (arr[i] > pivot)  
        swap(arr, next--, i);
```

```
arr[i] > pivot?    -- yes  
swap(arr, next, right)
```

start



pivot



next



i



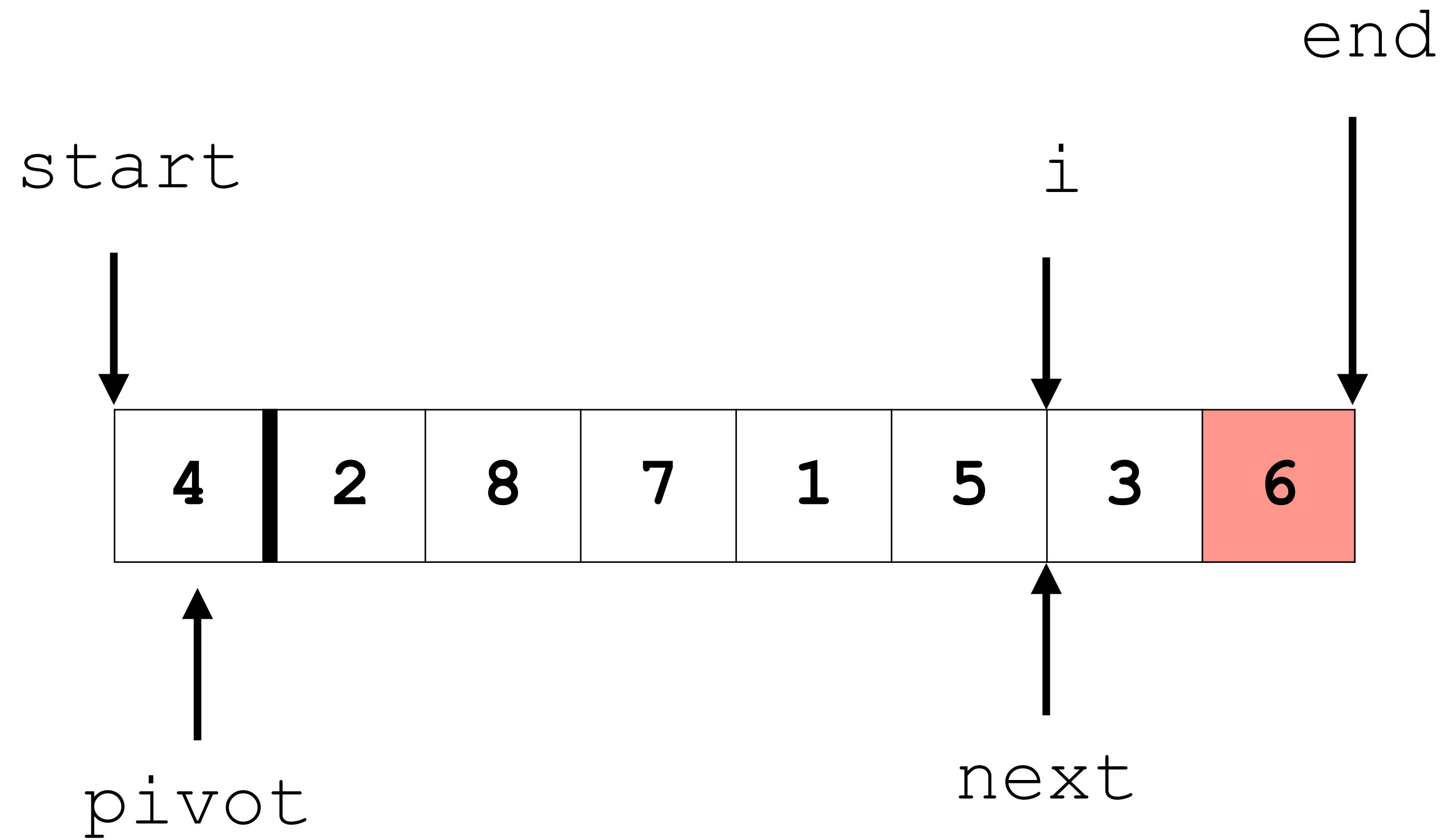
end



next + 1 is the
division line.

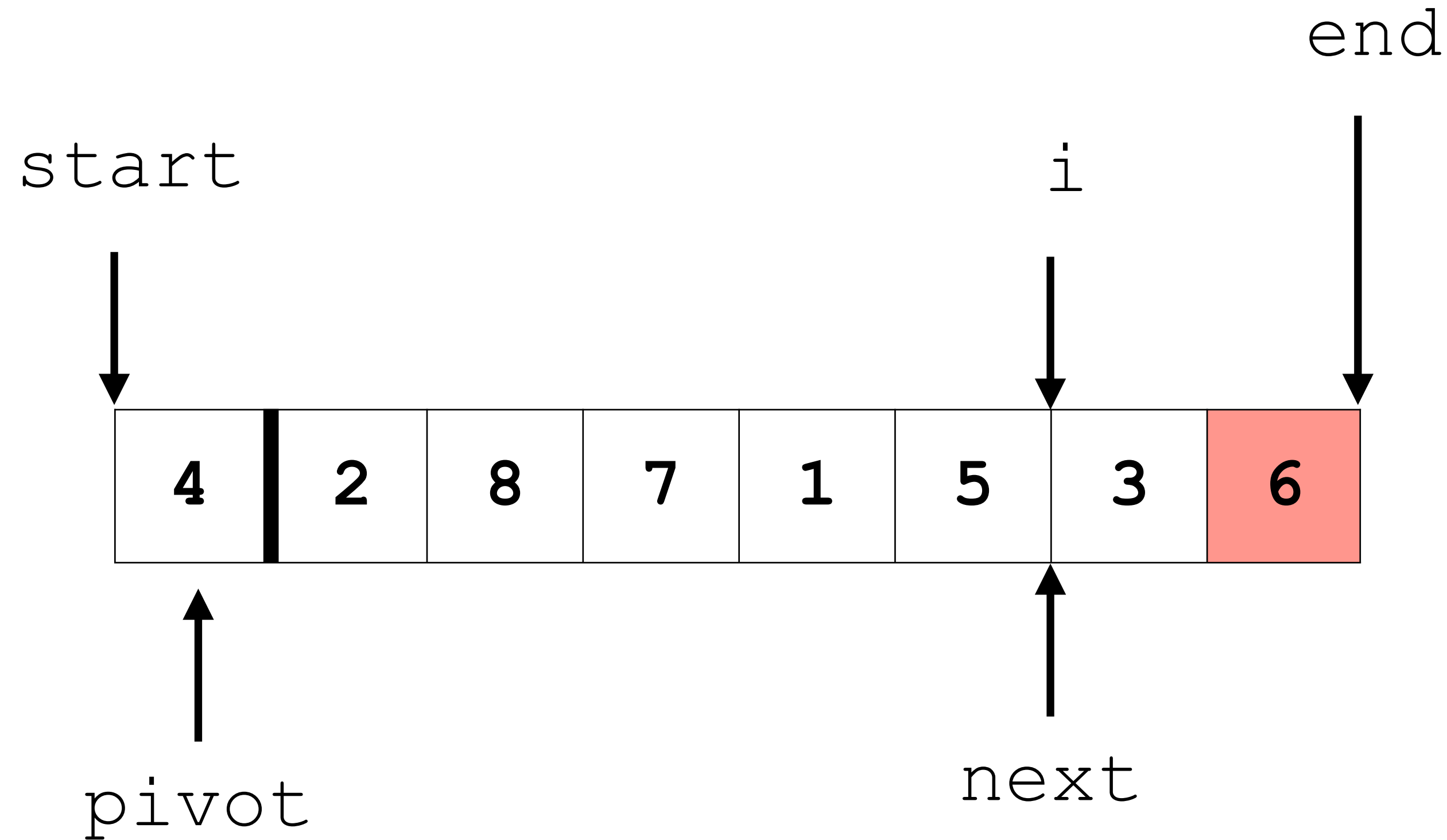
i + 1 is the start of
partitioned sub-array.

```
for (int i = end - 1; i > start; i--)  
    if (arr[i] > pivot)  
        swap(arr, next--, i);
```

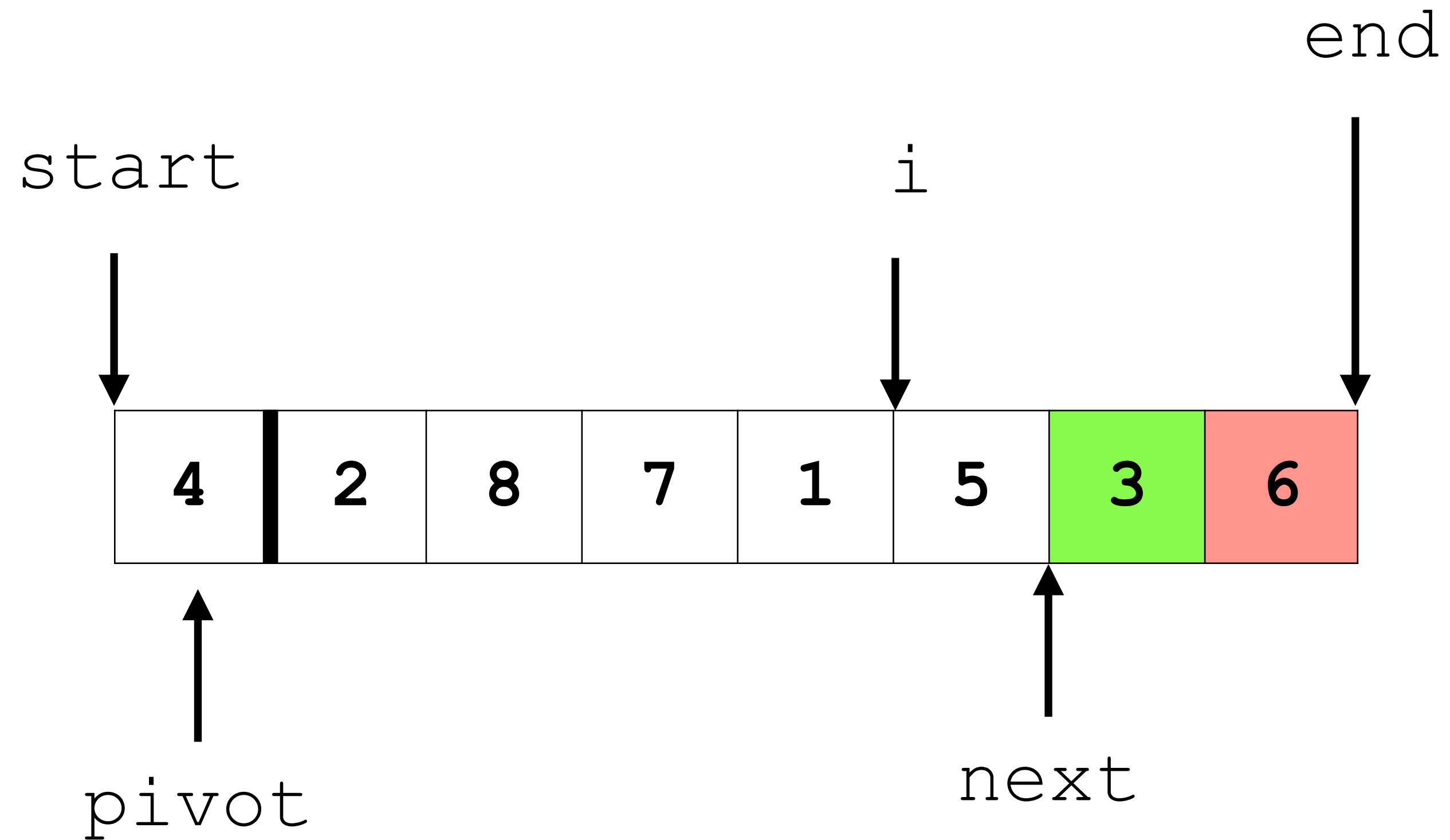


```
for (int i = end - 1; i > start; i--)  
    if (arr[i] > pivot)  
        swap(arr, next--, i);
```

```
arr[i] > pivot?    -- no
```



```
for (int i = end - 1; i > start; i--)  
    if (arr[i] > pivot)  
        swap(arr, next--, i);
```



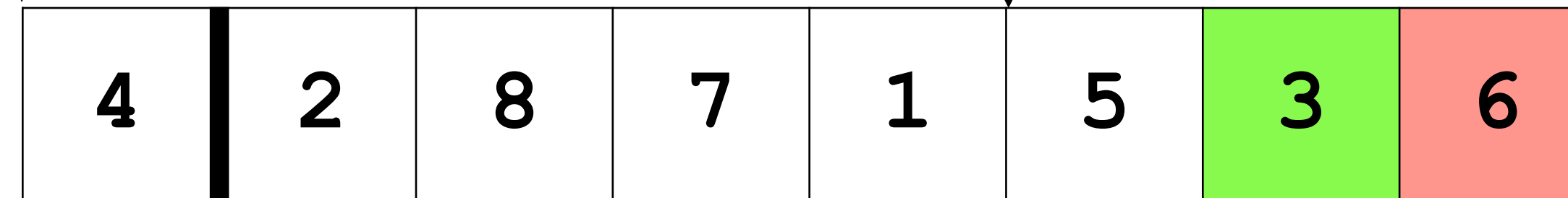
```
for (int i = end - 1; i > start; i--)  
    if (arr[i] > pivot)  
        swap(arr, next--, i);
```

```
arr[i] > pivot?    -- yes  
swap(arr, next, right)
```

start

i

end



pivot

next

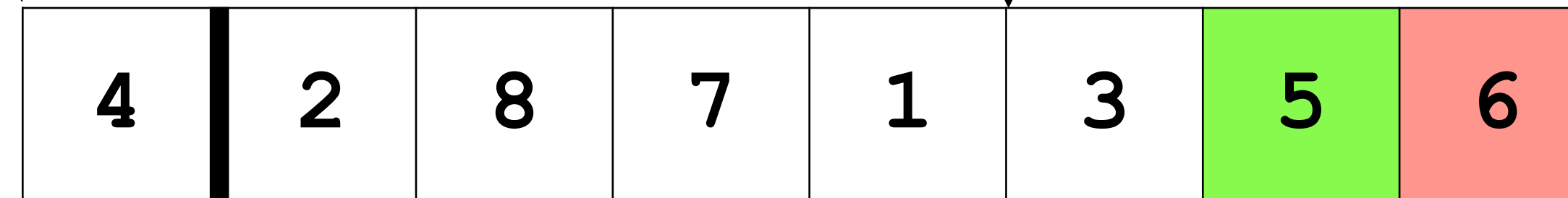
```
for (int i = end - 1; i > start; i--)  
    if (arr[i] > pivot)  
        swap(arr, next--, i);
```

```
arr[i] > pivot?    -- yes  
swap(arr, next, right)
```

start

i

end



pivot

next

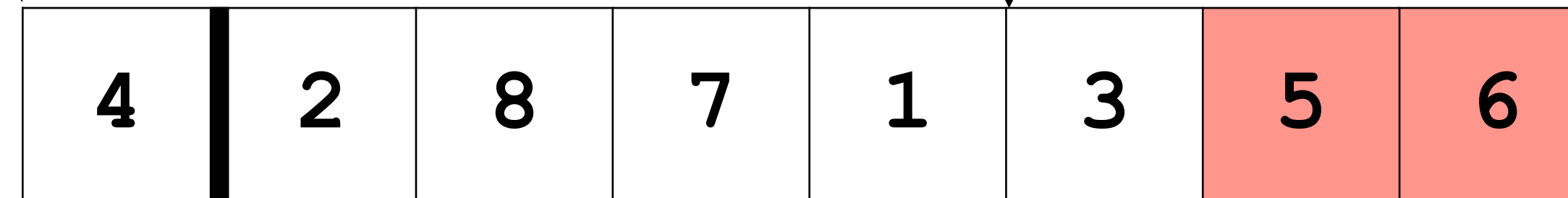
```
for (int i = end - 1; i > start; i--)  
    if (arr[i] > pivot)  
        swap(arr, next--, i);
```

```
arr[i] > pivot?    -- yes  
swap(arr, next, right)
```

start

i

end



pivot

next

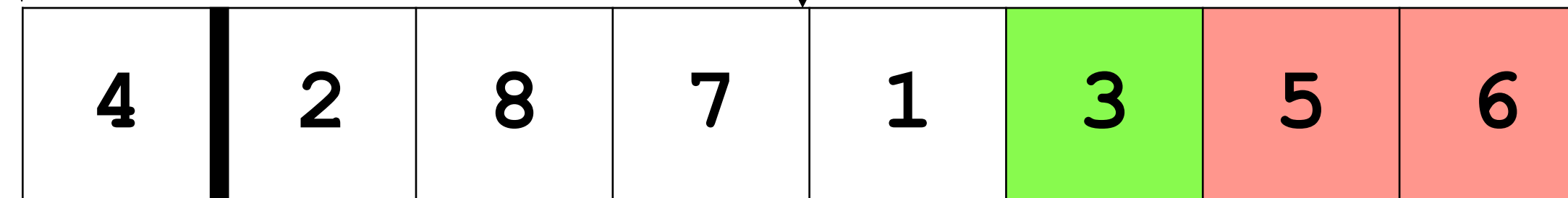
```
for (int i = end - 1; i > start; i--)  
    if (arr[i] > pivot)  
        swap(arr, next--, i);
```

```
arr[i] > pivot?    -- yes  
swap(arr, next, right)
```

start

i

end

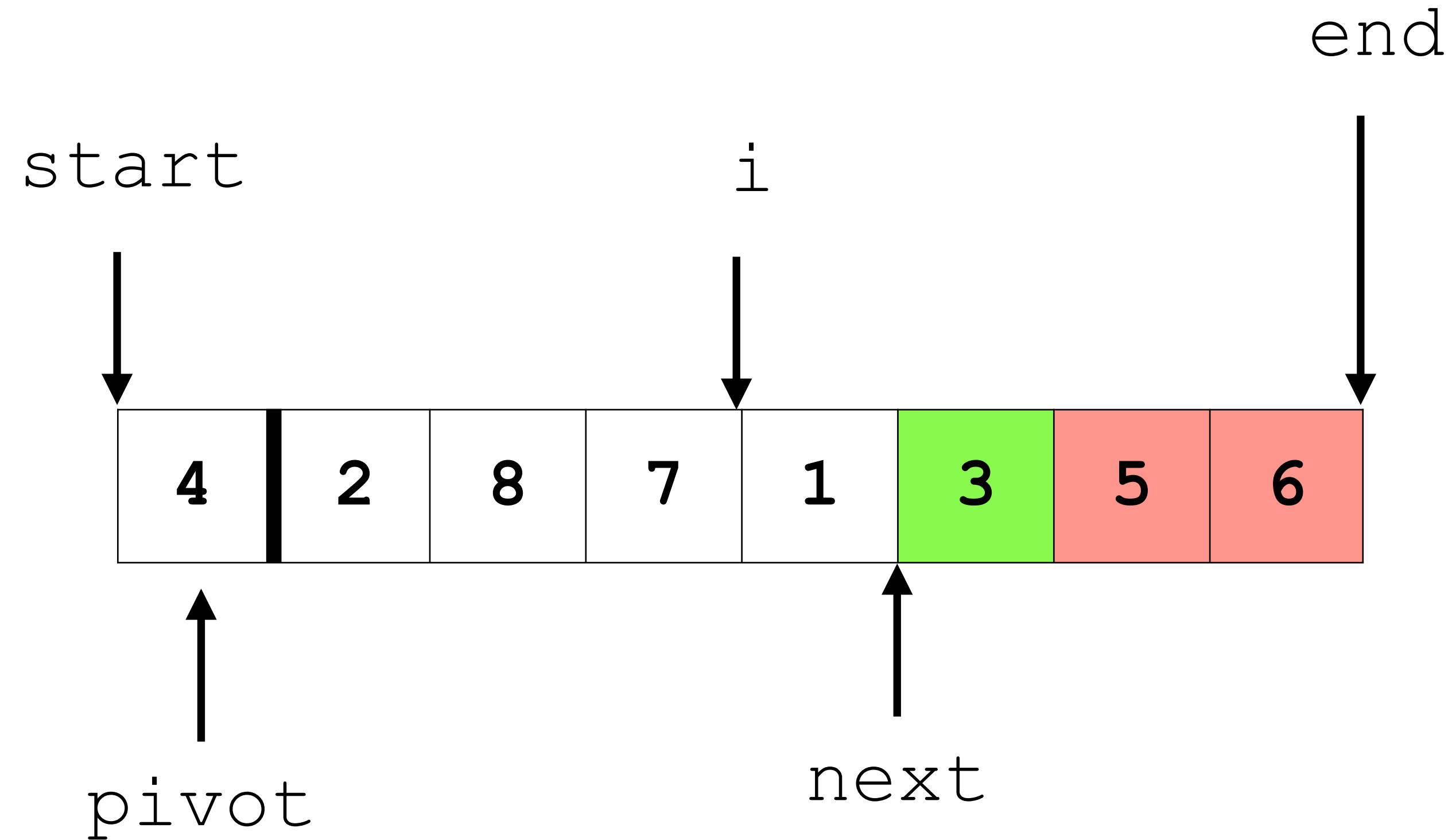


pivot

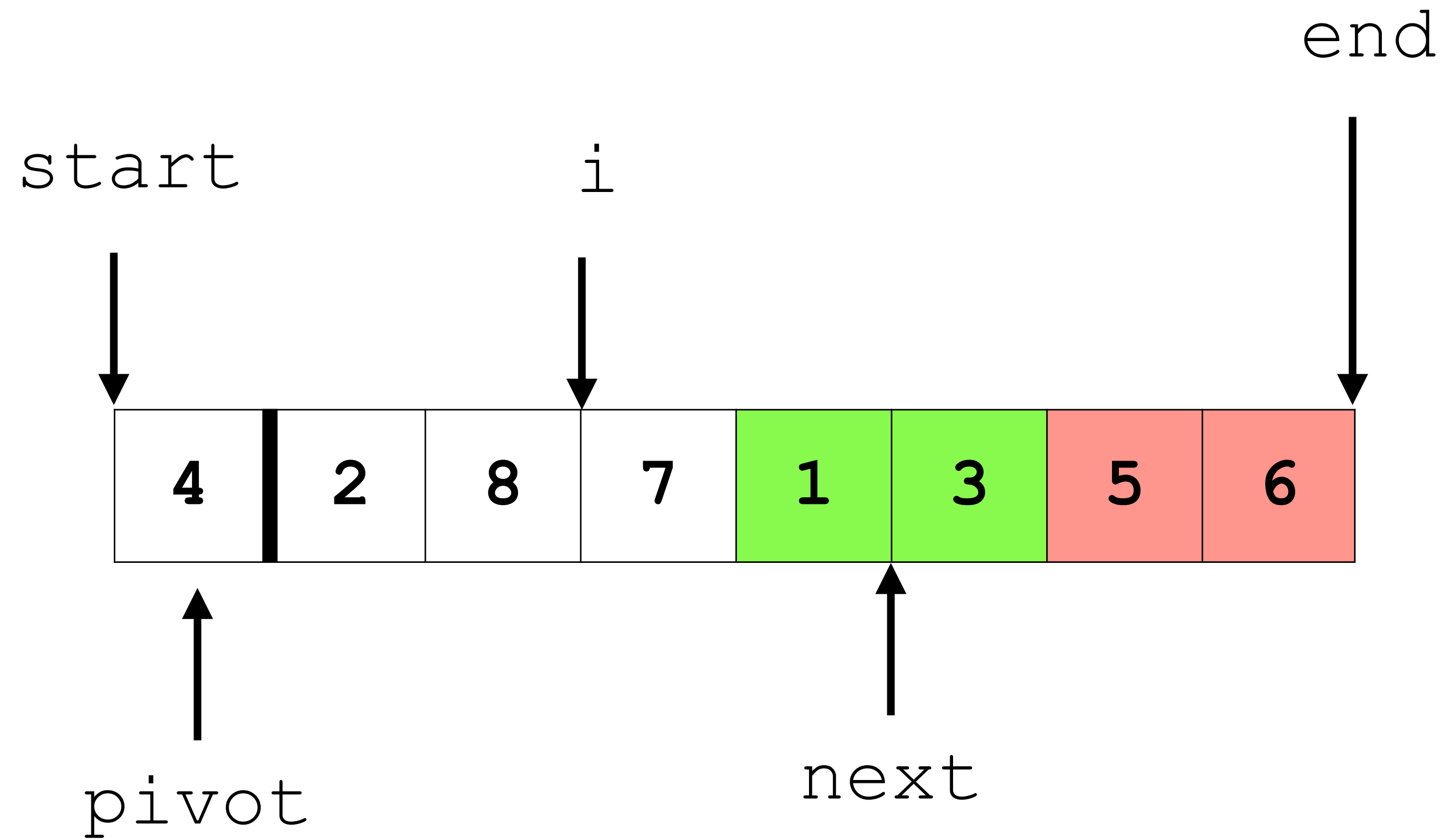
next

```
for (int i = end - 1; i > start; i--)  
    if (arr[i] > pivot)  
        swap(arr, next--, i);
```

```
arr[i] > pivot?    -- no
```



```
for (int i = end - 1; i > start; i--)  
    if (arr[i] > pivot)  
        swap(arr, next--, i);
```



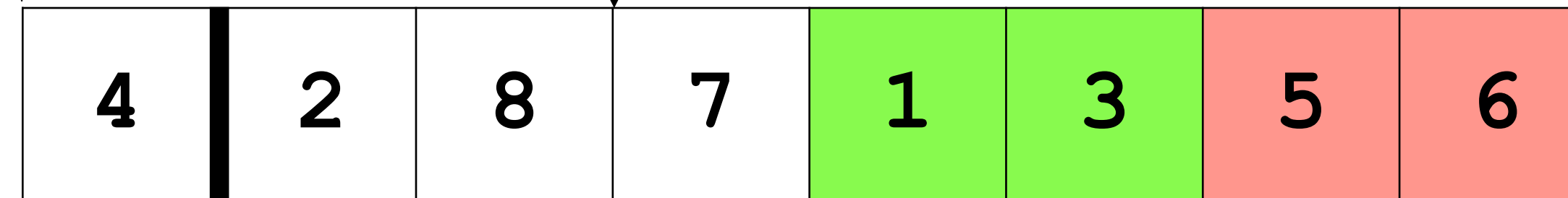
```
for (int i = end - 1; i > start; i--)  
    if (arr[i] > pivot)  
        swap(arr, next--, i);
```

```
arr[i] > pivot?    -- yes  
swap(arr, next, right)
```

start

i

end



pivot

next

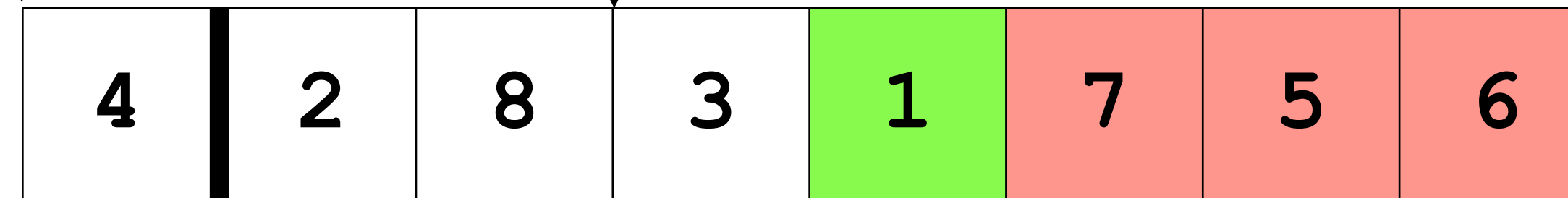
```
for (int i = end - 1; i > start; i--)  
    if (arr[i] > pivot)  
        swap(arr, next--, i);
```

```
arr[i] > pivot?    -- yes  
swap(arr, next, right)
```

start

i

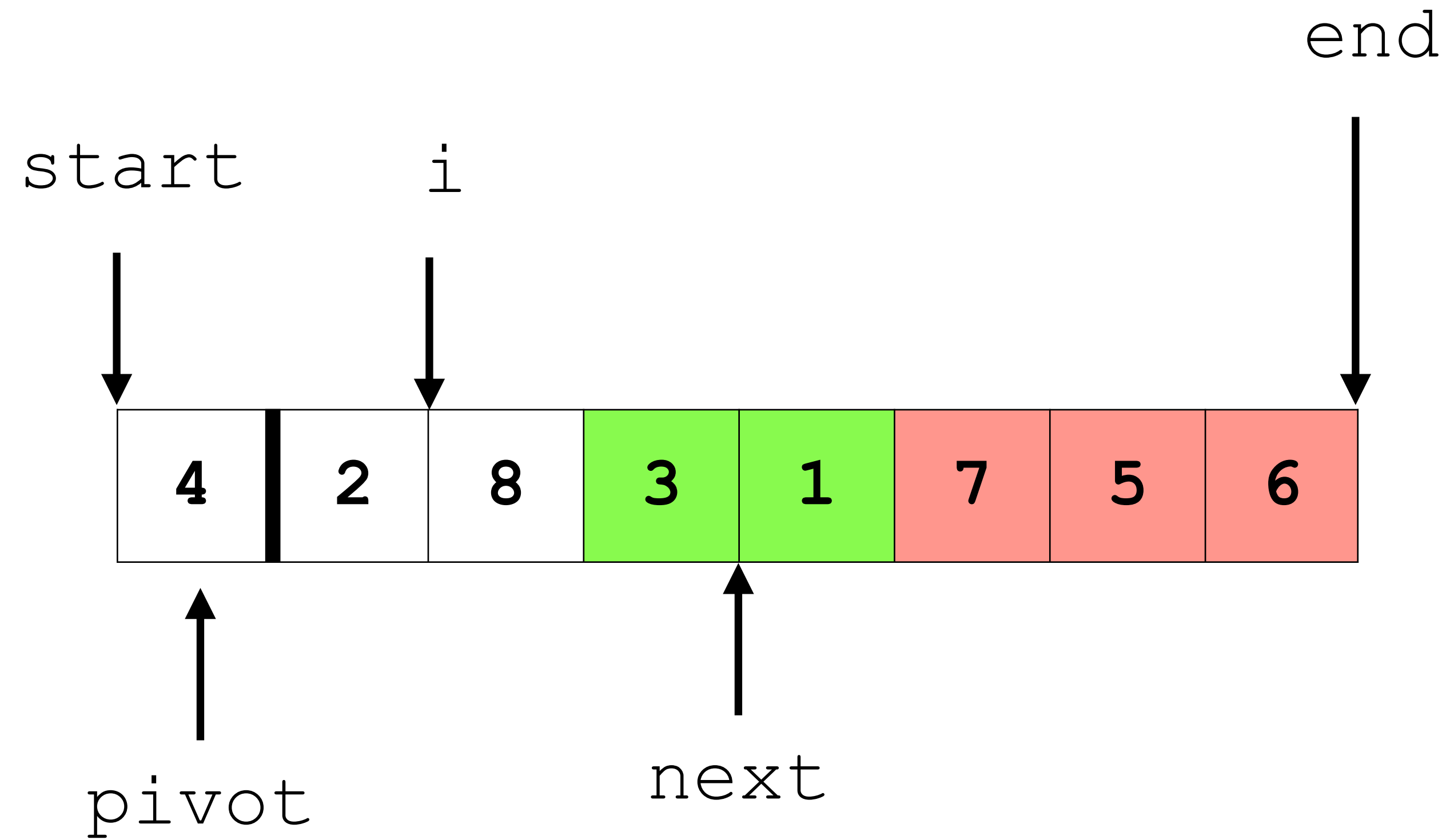
end



pivot

next

```
for (int i = end - 1; i > start; i--)  
    if (arr[i] > pivot)  
        swap(arr, next--, i);
```



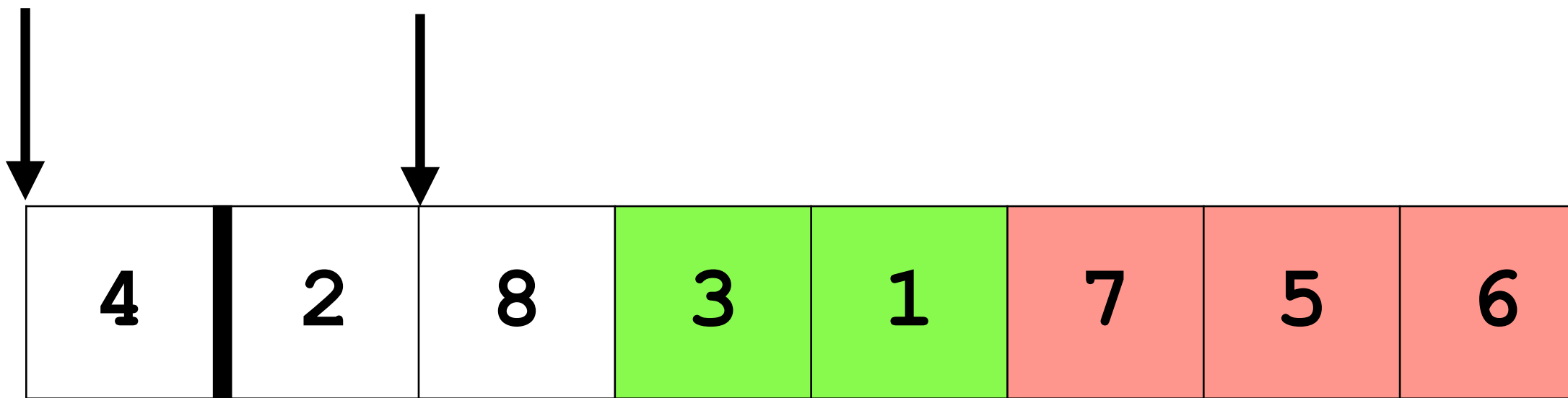
```
for (int i = end - 1; i > start; i--)  
    if (arr[i] > pivot)  
        swap(arr, next--, i);
```

```
arr[i] > pivot?    -- yes  
swap(arr, next, right)
```

start

i

end



pivot

next

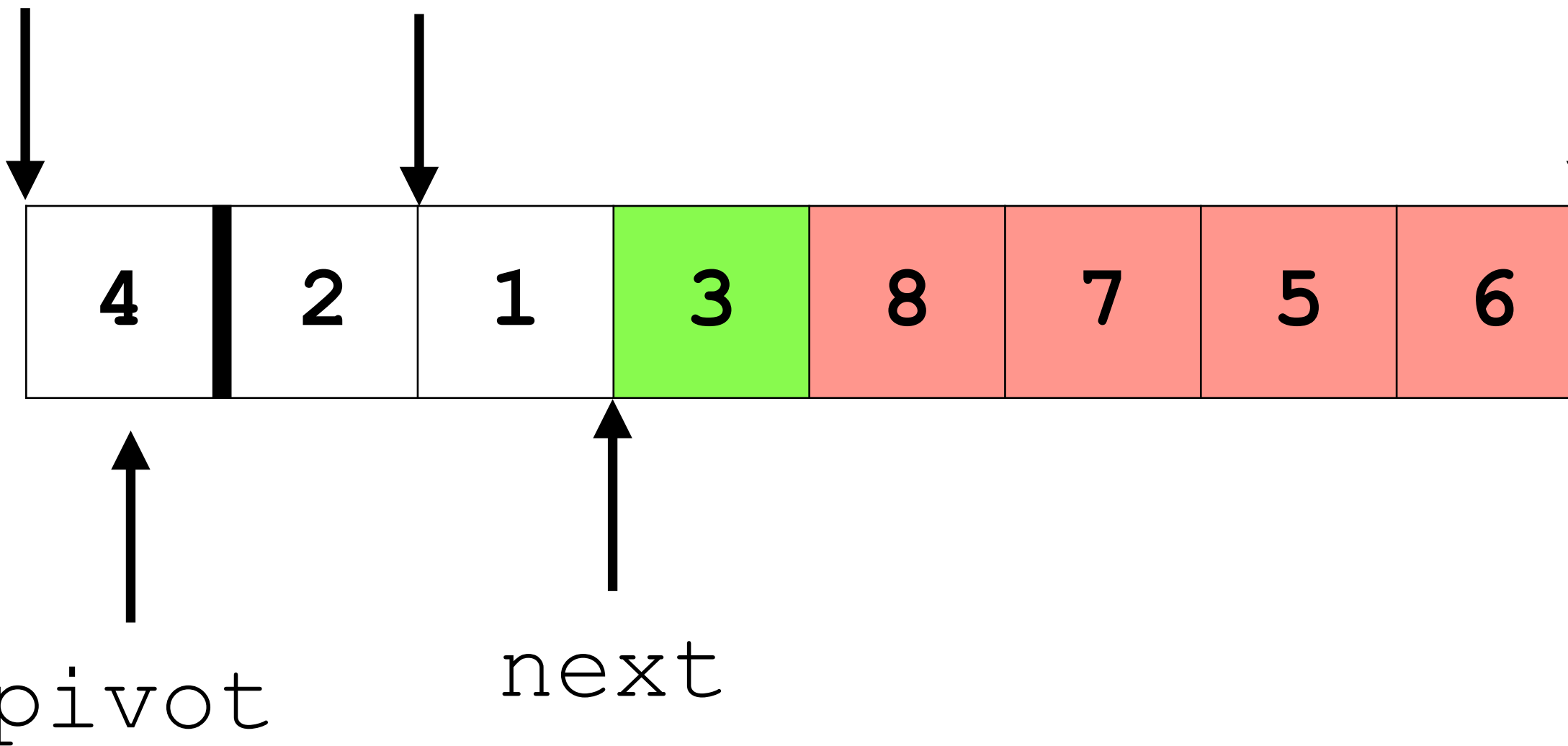
```
for (int i = end - 1; i > start; i--)  
    if (arr[i] > pivot)  
        swap(arr, next--, i);
```

```
arr[i] > pivot?    -- yes  
swap(arr, next, right)
```

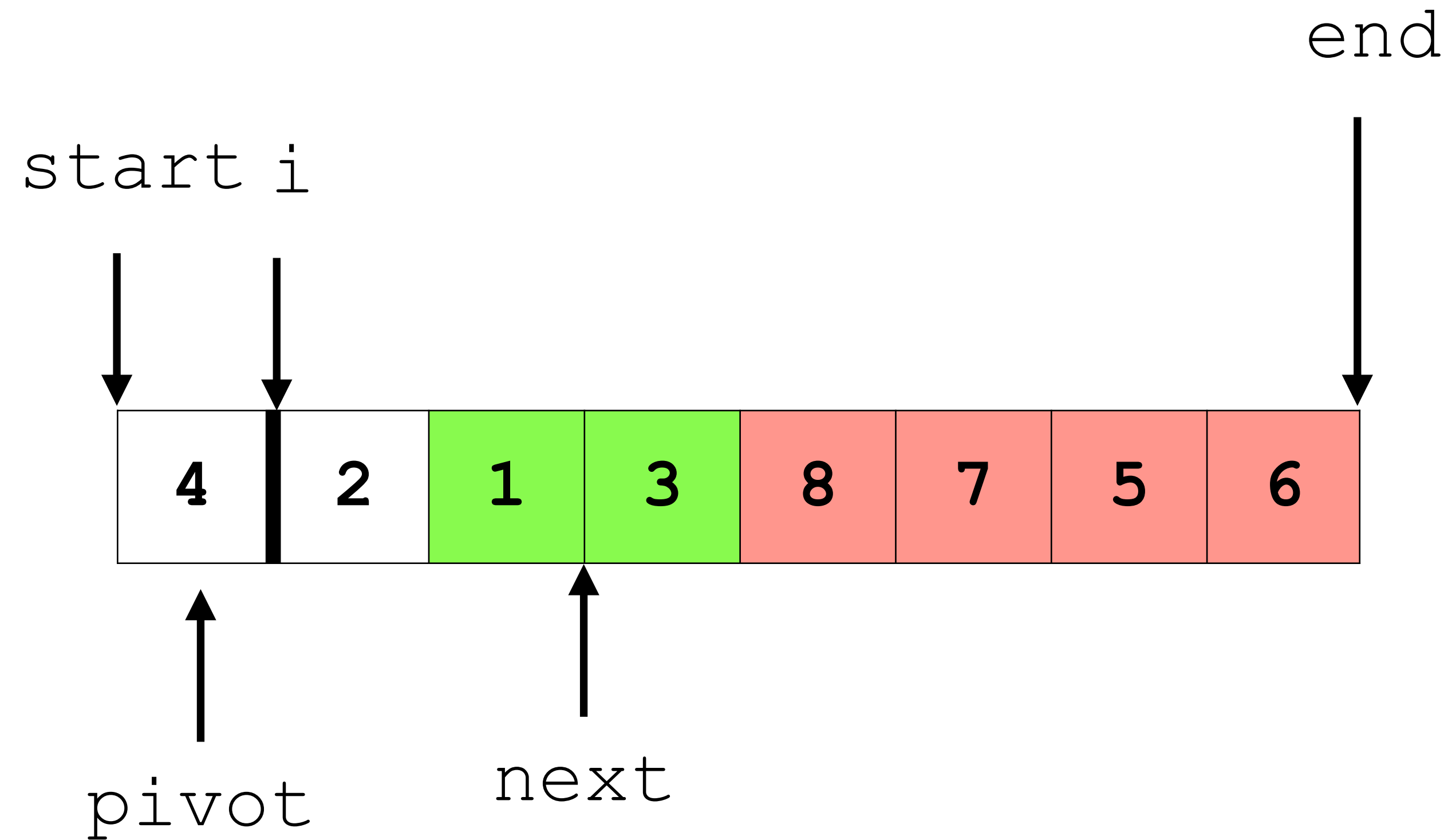
start

i

end

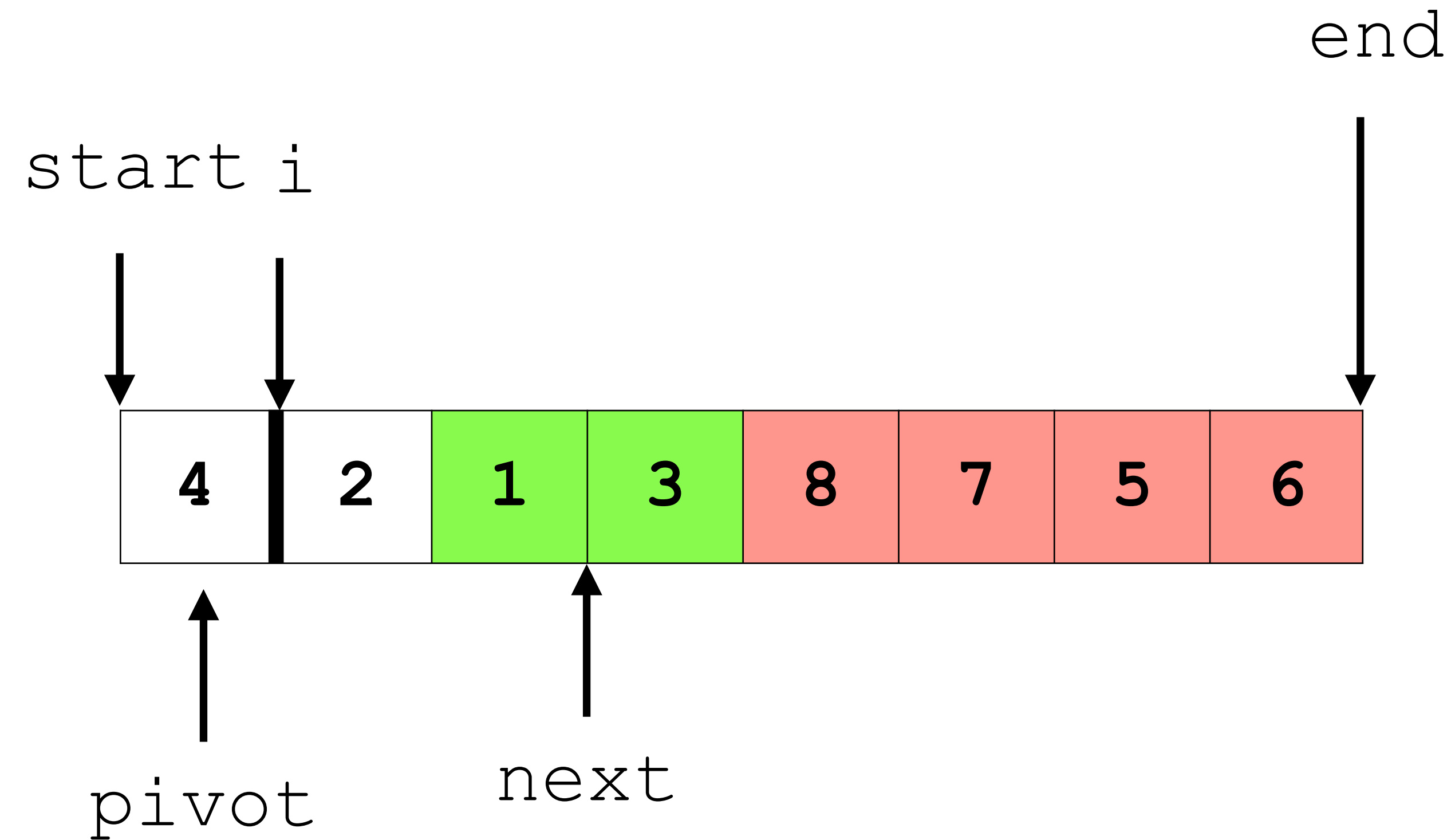


```
for (int i = end - 1; i > start; i--)  
    if (arr[i] > pivot)  
        swap(arr, next--, i);
```

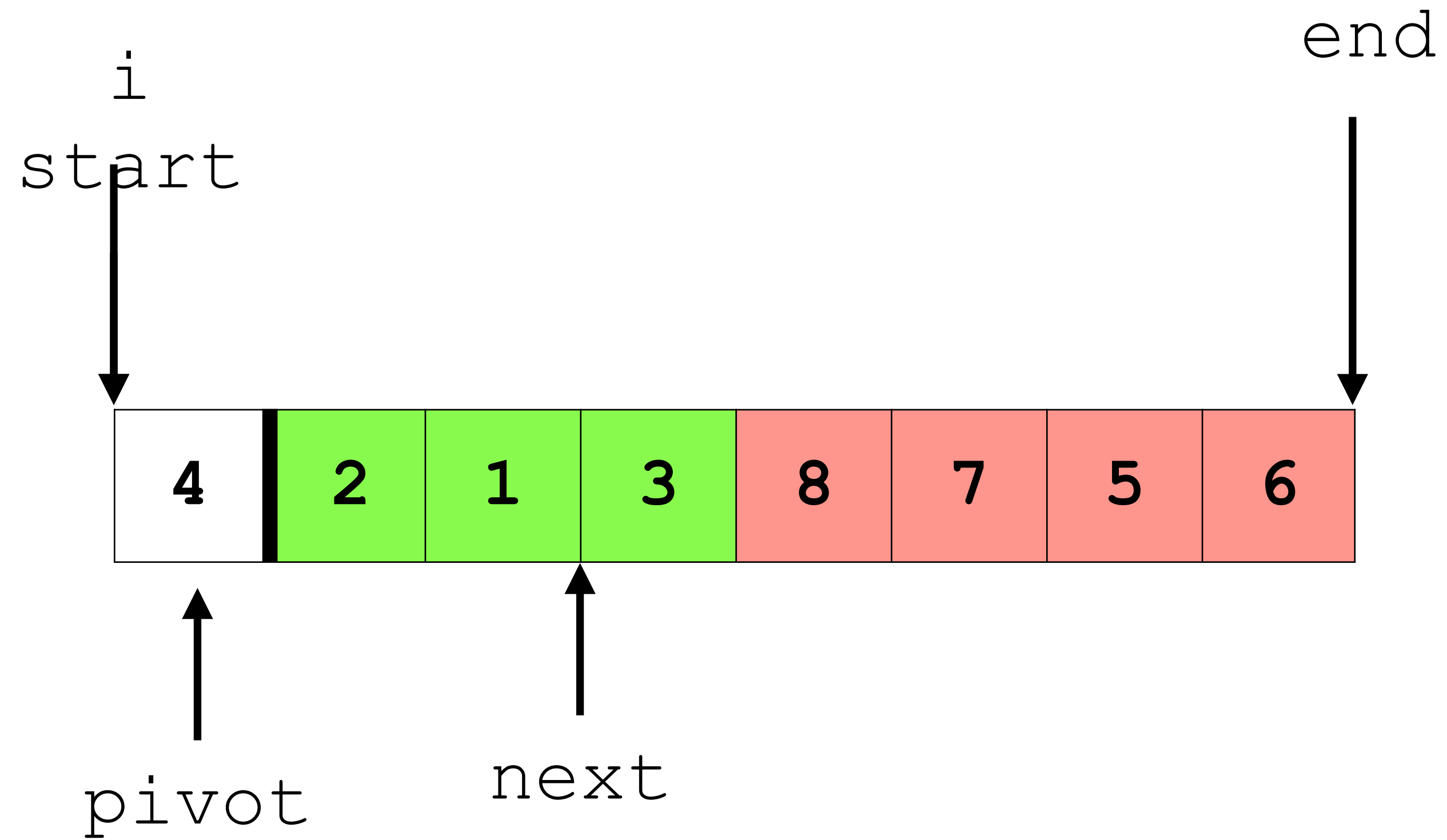


```
for (int i = end - 1; i > start; i--)  
    if (arr[i] > pivot)  
        swap(arr, next--, i);
```

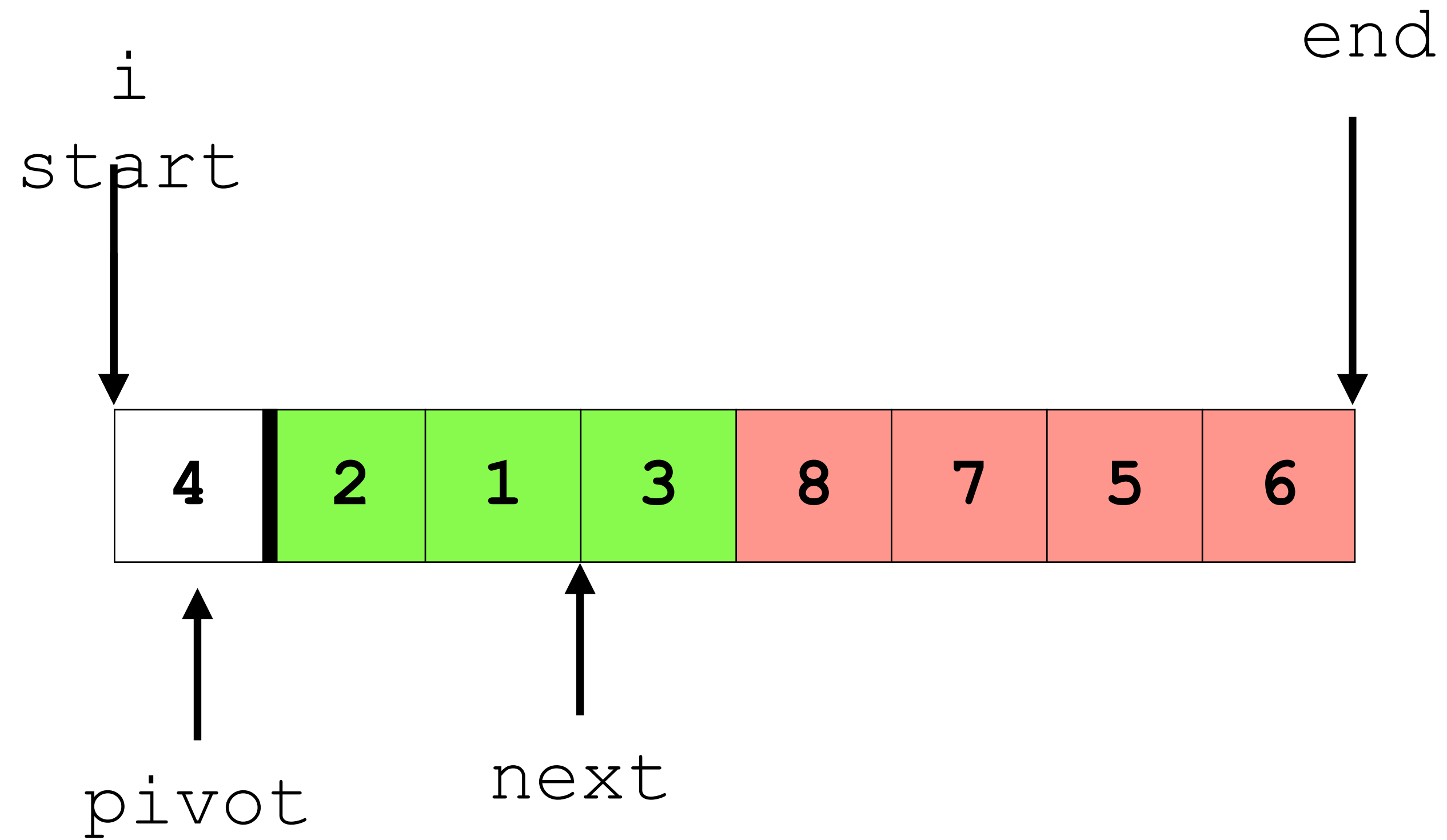
```
arr[i] > pivot?    -- no
```



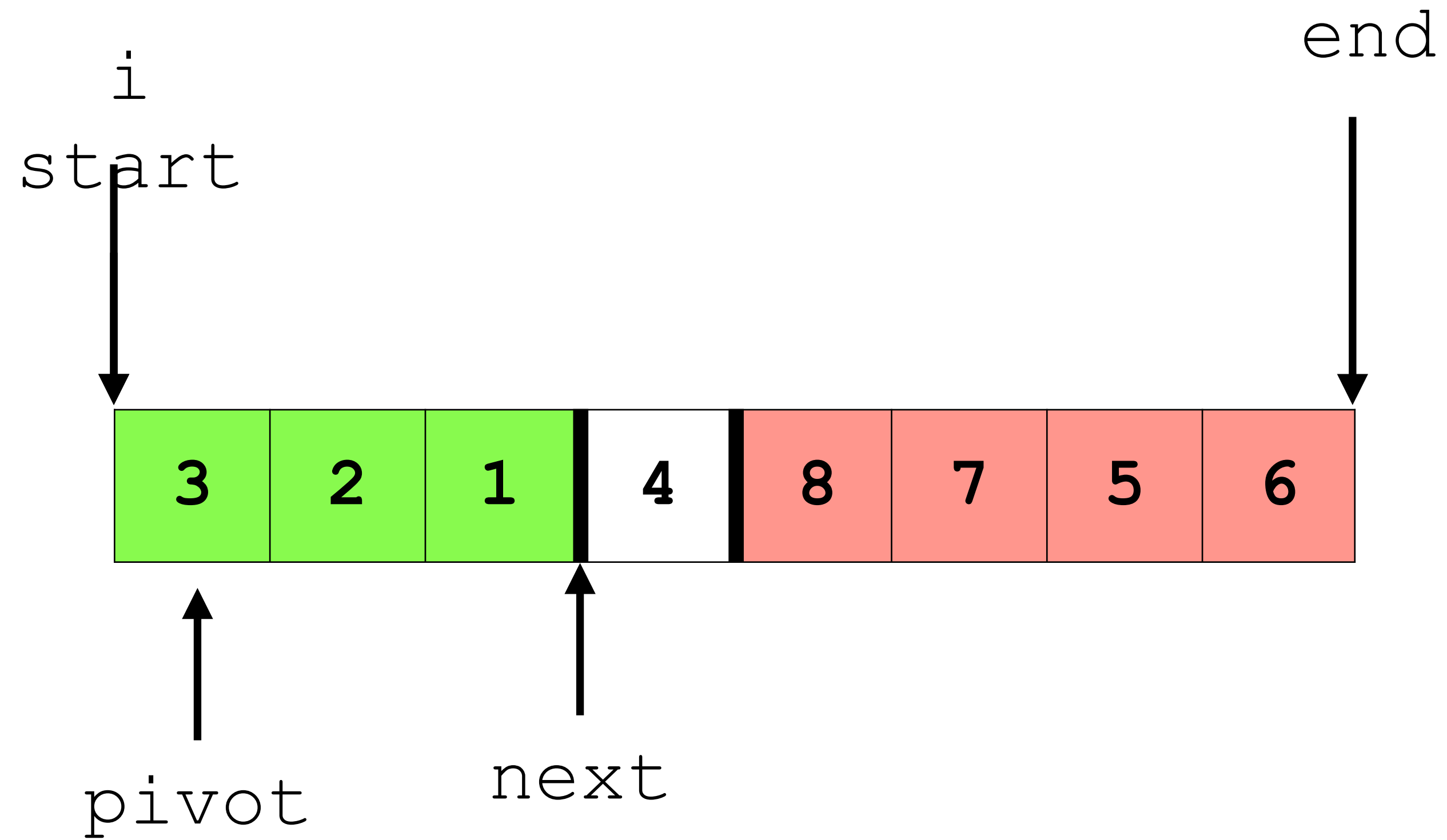
```
for (int i = end - 1; i > start; i--)  
    if (arr[i] > pivot)  
        swap(arr, next--, i);
```



```
for (int i = end - 1; i > start; i--)  
    if (arr[i] > pivot)  
        swap(arr, next--, i);
```



```
swap(arr, start, next);  
return next;
```



```
swap(arr, start, next);  
return next;
```

Quick Sort

Partition

```
int partition(int *arr, int start, int end)
{
    int pivot = arr[start];
    int next = end - 1;

    for (int i = end - 1; i > start; i--) {
        if (arr[i] > pivot) {
            swap(arr, next--, i);
        }
    }

    swap(arr, start, next);
    return next;
}
```

Quick Sort

```
void quick_sort(int *arr, int start, int end)
{
    if (start < end - 1) {
        int p = partition(arr, start, end);

        quick_sort(arr, start, p);
        quick_sort(arr, p + 1, end);
    }
}
```

Quick Sort

```
void quick_sort(int *arr, int start, int end)
{
    if (start < end - 1) {
        int p = partition(arr, start, end);

        quick_sort(arr, start, p);
        quick_sort(arr, p + 1, end);
    }
}
```

Arrays with size ≤ 1 is already sorted.

Quick Sort

```
void quick_sort(int *arr, int start, int end)
{
    if (start < end - 1) {
        int p = partition(arr, start, end);

        quick_sort(arr, start, p);
        quick_sort(arr, p + 1, end);
    }
}
```

Arrays with size ≤ 1 is already sorted.

Divide

Quick Sort

```
void quick_sort(int *arr, int start, int end)
{
    if (start < end - 1) {
        int p = partition(arr, start, end);

        quick_sort(arr, start, p);
        quick_sort(arr, p + 1, end);
    }
}
```

Arrays with size ≤ 1 is already sorted.

Divide

Conquer

Quick Sort

Time Complexity

Quick Sort

Time Complexity

- Partitioning an array of size n takes $O(n)$ time

Quick Sort

Time Complexity

- Partitioning an array of size n takes $O(n)$ time
- How many levels of partitions are there?

Quick Sort

Time Complexity

[6, 20, 3, 2, 45, 80]

[3, 2] [6] [20, 45, 80]

[2] [3] [] [] [20] [45, 80]

 [] [45] [80]

Depends on how good the partition is!

Quick Sort

Time Complexity

Quick Sort

Time Complexity

- Partitioning an array of size n takes $O(n)$ time

Quick Sort

Time Complexity

- Partitioning an array of size n takes $O(n)$ time
- How many levels of partitions are there?

Quick Sort

Time Complexity

- Partitioning an array of size n takes $O(n)$ time
- How many levels of partitions are there?
 - Best case: We partition each level *evenly* -- there are $O(\log n)$ levels.

Quick Sort

Time Complexity

- Partitioning an array of size n takes $O(n)$ time
- How many levels of partitions are there?
 - Best case: We partition each level *evenly* -- there are $O(\log n)$ levels.
 - Happens when the pivot is approximately the median

Quick Sort

Time Complexity

- Partitioning an array of size n takes $O(n)$ time
- How many levels of partitions are there?
 - Best case: We partition each level *evenly* -- there are $O(\log n)$ levels.
 - Happens when the pivot is approximately the median
 - But we don't know the median until the list is sorted.

Quick Sort

Time Complexity

- Partitioning an array of size n takes $O(n)$ time
- How many levels of partitions are there?
 - Best case: We partition each level *evenly* -- there are $O(\log n)$ levels.
 - Happens when the pivot is approximately the median
 - But we don't know the median until the list is sorted.
 - Worst case: Lopsided partition (similar to BST) -- there are $O(n)$ levels

Quick Sort

Time Complexity

- Partitioning an array of size n takes $O(n)$ time
- How many levels of partitions are there?
 - Best case: We partition each level *evenly* -- there are $O(\log n)$ levels.
 - Happens when the pivot is approximately the median
 - But we don't know the median until the list is sorted.
 - Worst case: Lopsided partition (similar to BST) -- there are $O(n)$ levels
 - Happens when the pivot is the smallest or the greatest

Quick Sort

Pivot

Quick Sort

Pivot

- In our choice of pivot (leftmost element), what input array will result in the worst case behavior?

Quick Sort

Pivot

- In our choice of pivot (leftmost element), what input array will result in the worst case behavior?
 - Answer: An array that is already sorted.

Quick Sort

Pivot

- In our choice of pivot (leftmost element), what input array will result in the worst case behavior?
 - Answer: An array that is already sorted.
- Choosing a pivot:

Quick Sort

Pivot

- In our choice of pivot (leftmost element), what input array will result in the worst case behavior?
 - Answer: An array that is already sorted.
- Choosing a pivot:
 - Random element

Quick Sort

Pivot

- In our choice of pivot (leftmost element), what input array will result in the worst case behavior?
 - Answer: An array that is already sorted.
- Choosing a pivot:
 - Random element
 - Middle element: works well for almost sorted arrays

Quick Sort

Pivot

- In our choice of pivot (leftmost element), what input array will result in the worst case behavior?
 - Answer: An array that is already sorted.
- Choosing a pivot:
 - Random element
 - Middle element: works well for almost sorted arrays
 - Median of [first, last, middle elements]

Quick Sort

Pivot

- In our choice of pivot (leftmost element), what input array will result in the worst case behavior?
 - Answer: An array that is already sorted.
- Choosing a pivot:
 - Random element
 - Middle element: works well for almost sorted arrays
 - Median of [first, last, middle elements]
 - Median of random sample of elements

Quick Sort

Pivot

- In our choice of pivot (leftmost element), what input array will result in the worst case behavior?
 - Answer: An array that is already sorted.
- Choosing a pivot:
 - Random element
 - Middle element: works well for almost sorted arrays
 - Median of [first, last, middle elements]
 - Median of random sample of elements
 - ...

Quick Sort

Complexity

Quick Sort

Complexity

- Time Complexity:

Quick Sort

Complexity

- Time Complexity:
 - Best case: $O(n \log n)$

Quick Sort

Complexity

- Time Complexity:
 - Best case: $O(n \log n)$
 - Worst case: $O(n^2)$

Quick Sort

Complexity

- Time Complexity:
 - Best case: $O(n \log n)$
 - Worst case: $O(n^2)$
- Space Complexity:

Quick Sort

Complexity

- Time Complexity:
 - Best case: $O(n \log n)$
 - Worst case: $O(n^2)$
- Space Complexity:
 - No temporary array, call stack uses $O(\log n)$

Merge and Quick Sorts

- Divide-and-conquer Algorithm:

1. Divide an array in ~half
2. Sort both arrays individually
3. Combine the two arrays

Divide

Conquer

Glue

- Merge Sort:

- **Divide:** Easy: just halve the list
- **Glue:** Hard: merge sorted lists

- Quick Sort:

- **Divide:** Hard: partition
- **Glue:** Easy: no-op