Sorting II

CS143: lecture 14

Byron Zhong, July 15

Function Pointers

Function Pointers

- Your code lives in memory too!
- ...so they have addresses
- ...so just like we have pointers to data, we have pointers to functions as well
- What's the point?
  - We can pass functions around!

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *))
```

- The second argument to this function is
  - A function pointer called `cmp`
  - The function that `cmp` points to takes two `void *` and returns `int`
  - It tells the sorting function how to compare two arbitrary elements
  - (negative if 1 < 2, 0 if 1 == 2, positive if 1 > 2)

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *)) {
        for (;;) {
                int swapped = 0;
                for (int i = 0; i < l->length; i++) {
                        if (cmp(l->elems[i], l->elems[i + 1]) < 0) {
                                void *tmp = l->elems[i];
                                l->elems[i] = l->elems[i + 1];
                                l->elems[i + 1] = tmp;
                                swapped = 1;
                        }
                }

                if (!swapped) {
                        break;
                }
        }
}
```

Function Pointers

Example

```
void alist_sort(struct alist *l, int (*cmp)(void *, void *));

int strcmp_wrapper(void *s1, void *s2)
{
        return strcmp(s1, s2);
}

int main(void)
{
        struct alist l;
        alist_sort(&l, &strcmp_wrapper);
        return 0;
}
```

^ optional

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

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

Merge and Quick Sorts

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

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

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);
}
```

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]
```

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:

Merge Operation

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

Merge Operation

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

Merge Operation

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

Merge Operation

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

Merge Operation

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

[0, 6, 20, 32, 34

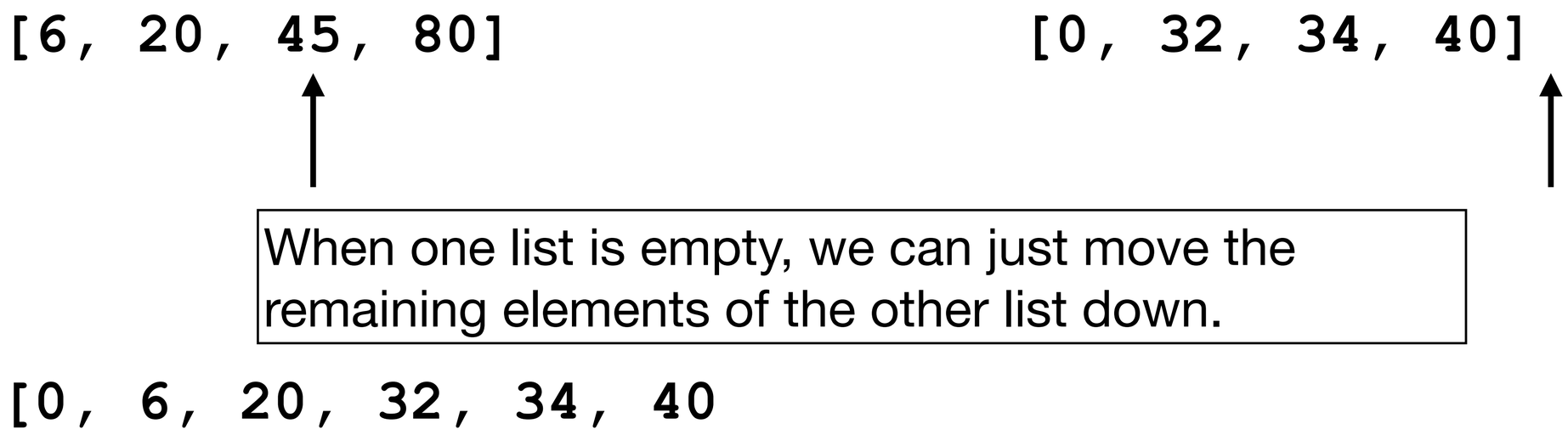
Merge Operation

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

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

Merge Operation

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



Merge Operation

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


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

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);
}
```

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);
}
```

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);
}
```

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]
```

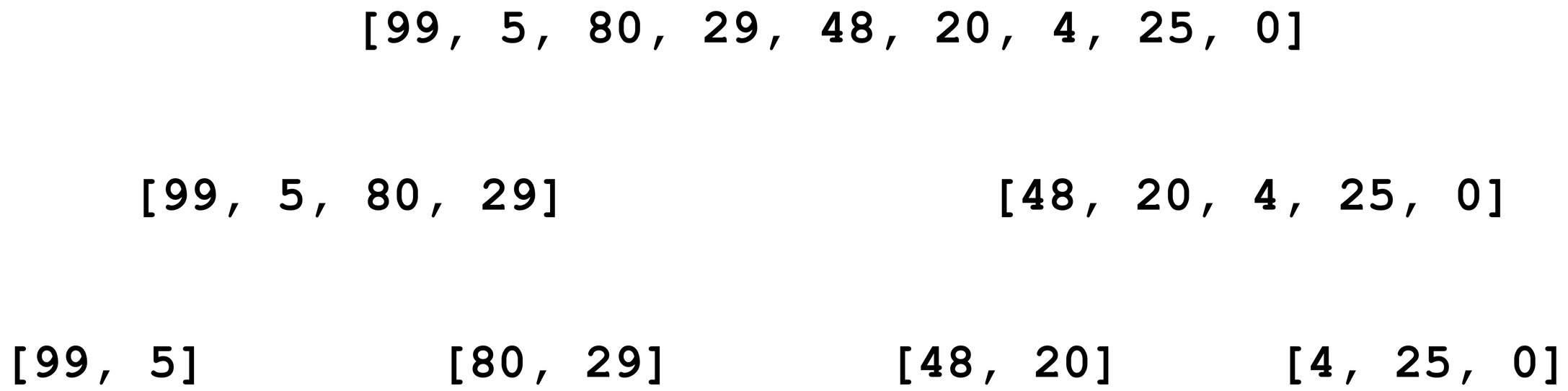
Divide

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

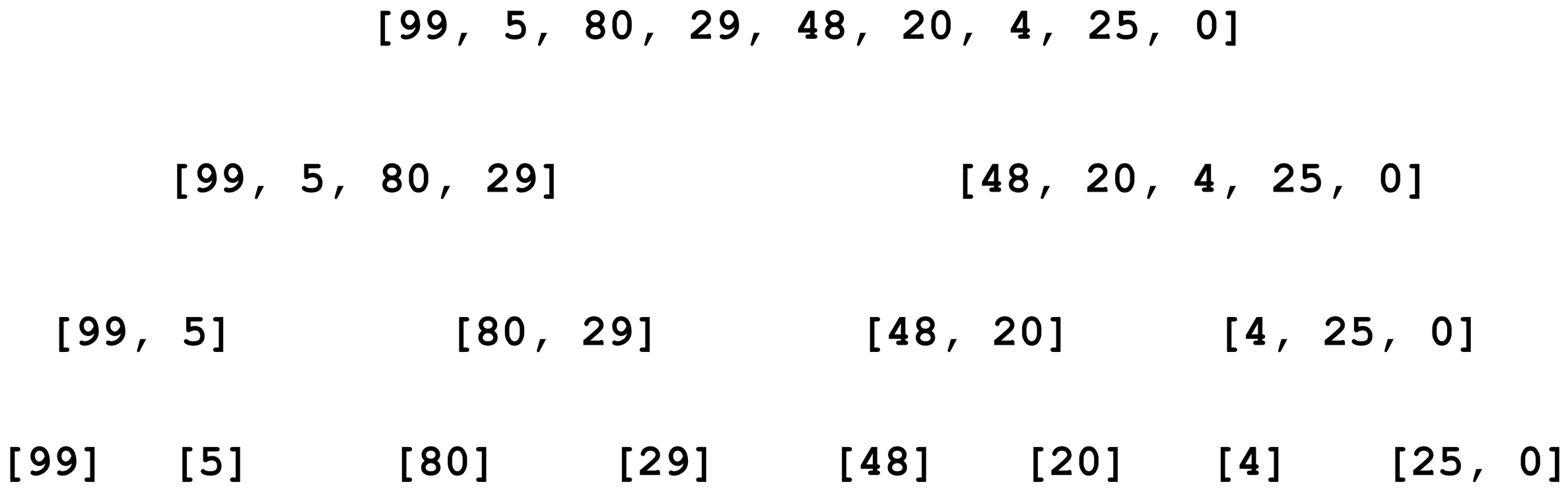
```
[99, 5, 80, 29]
```

```
[48, 20, 4, 25, 0]
```

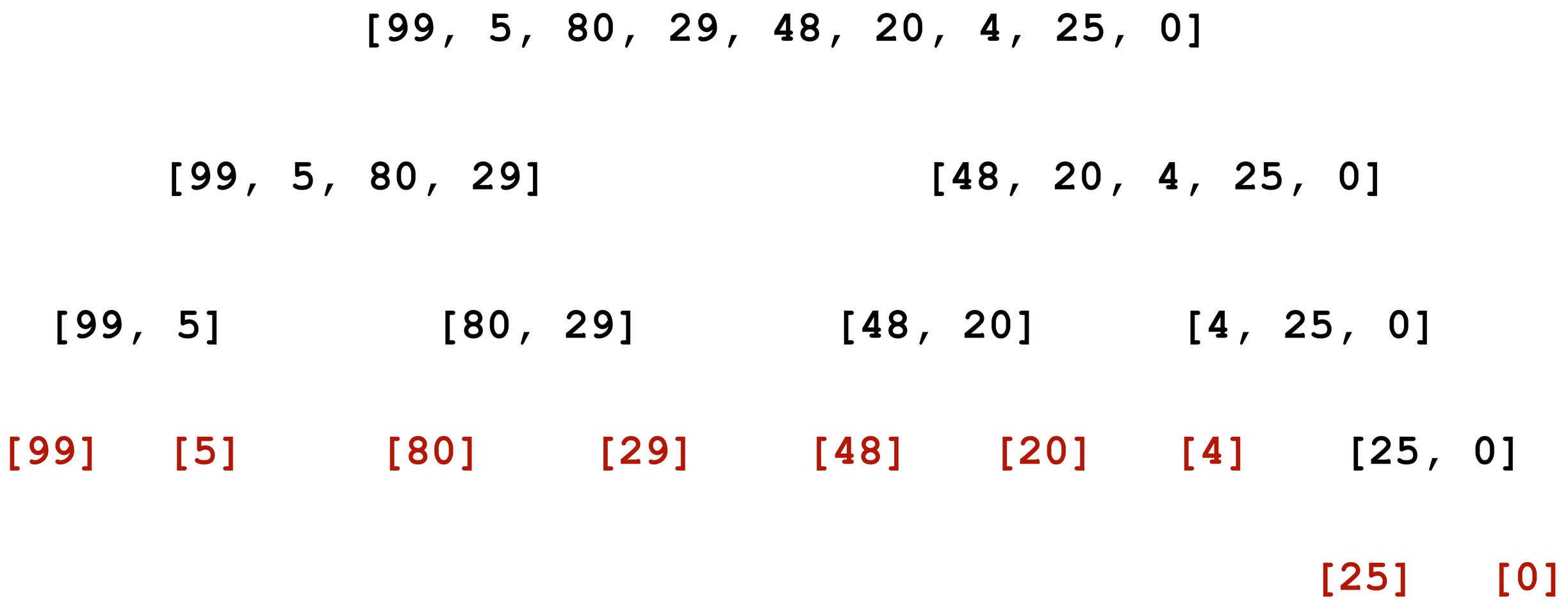
Divide



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



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

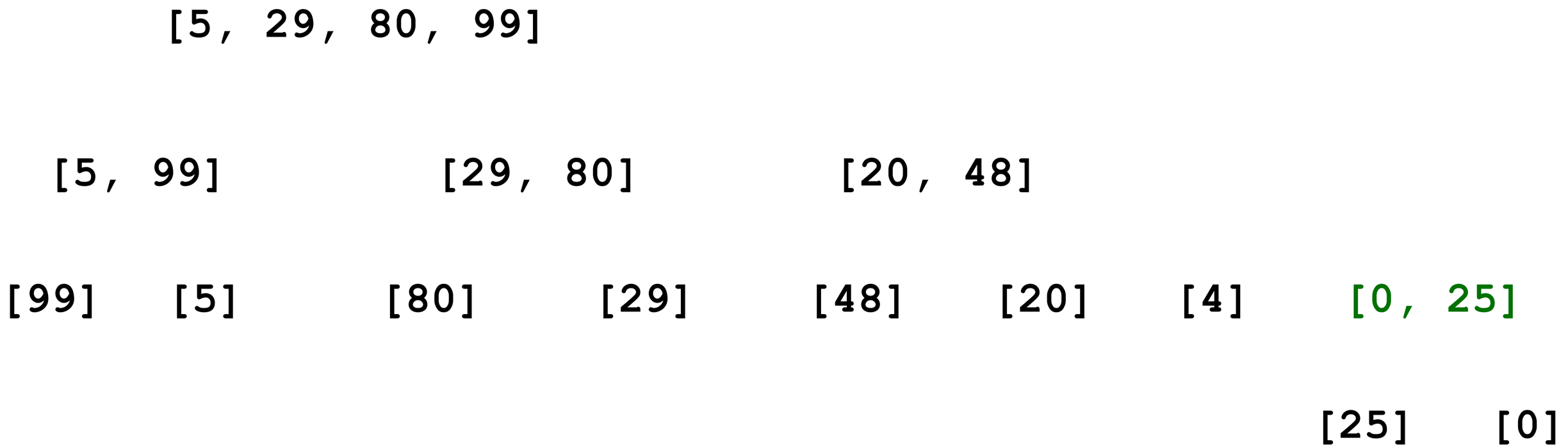
[5, 29, 80, 99]

[5, 99] [29, 80] [20, 48]

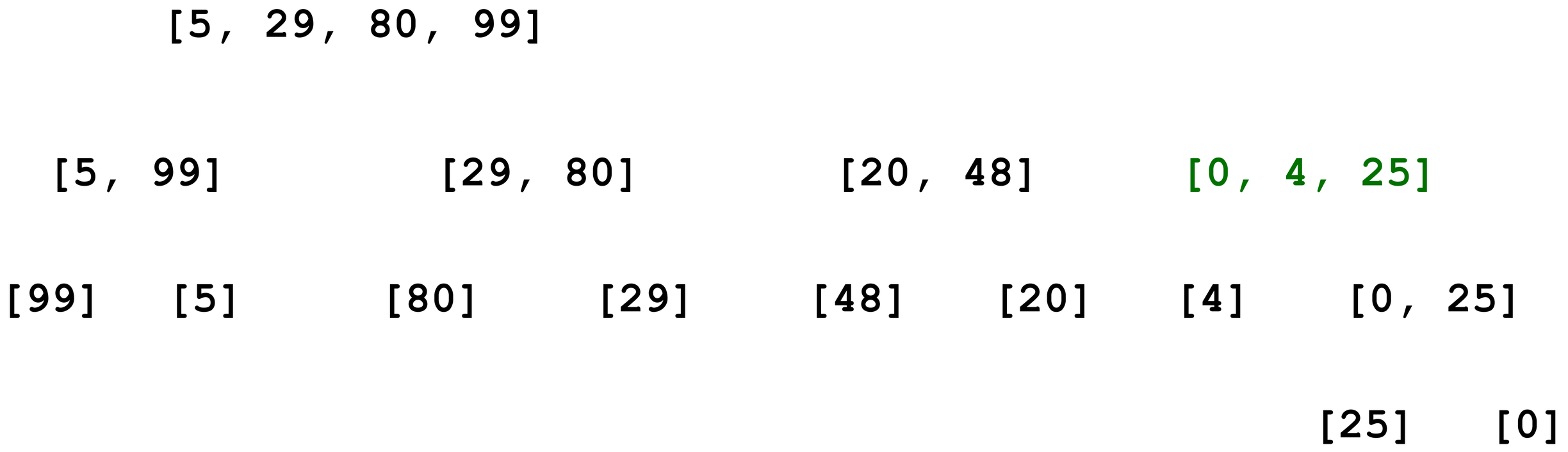
[99] [5] [80] [29] [48] [20] [4]

[25] [0]

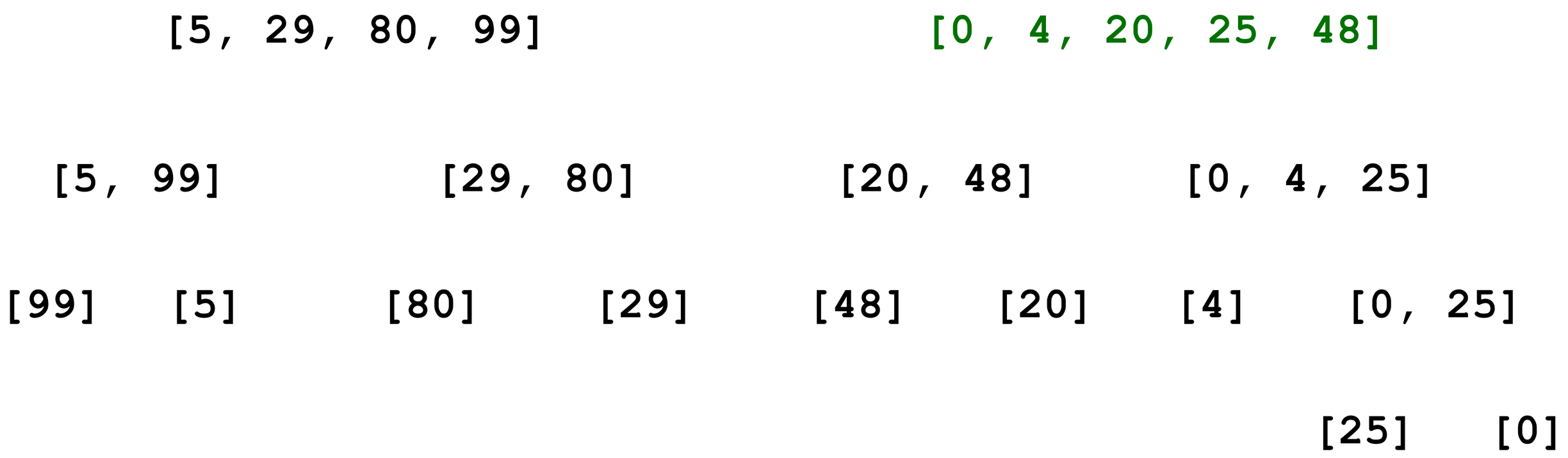
Merge



Merge



Merge



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]

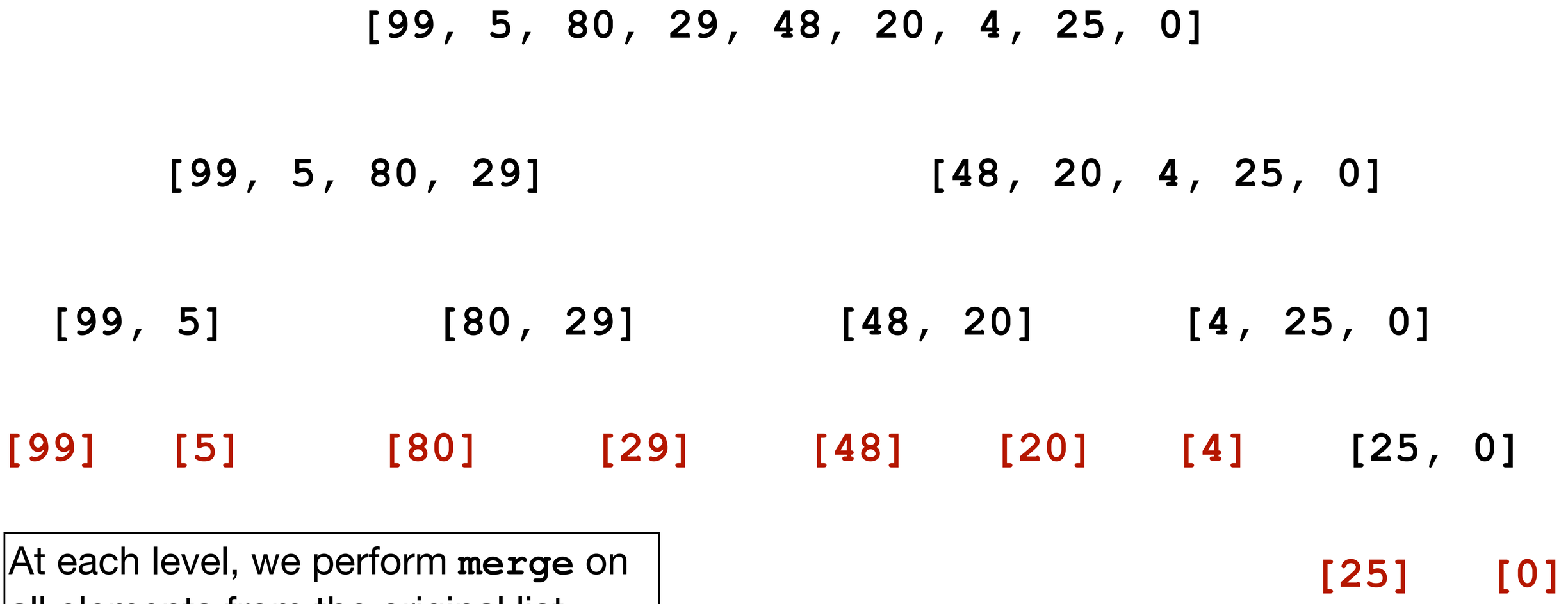
[25] [0]

Merge Sort

Time Complexity

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

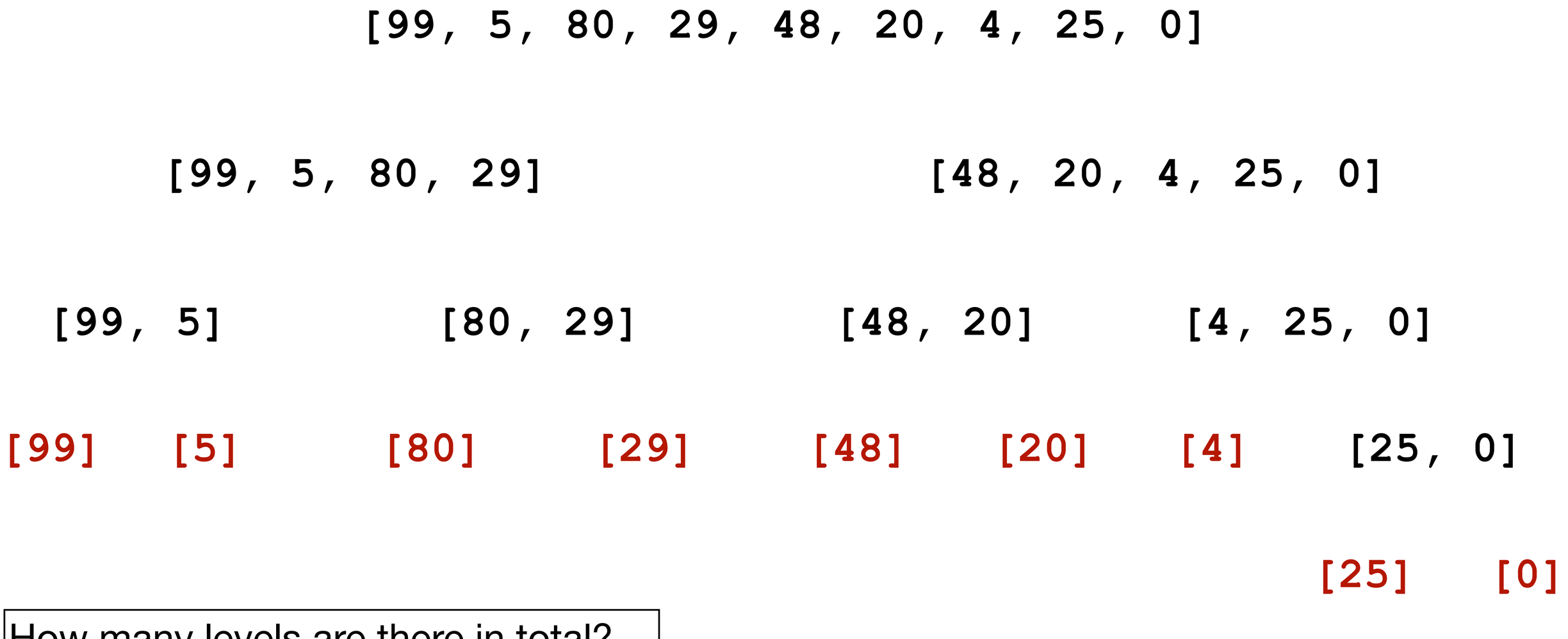
Merge Sort



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

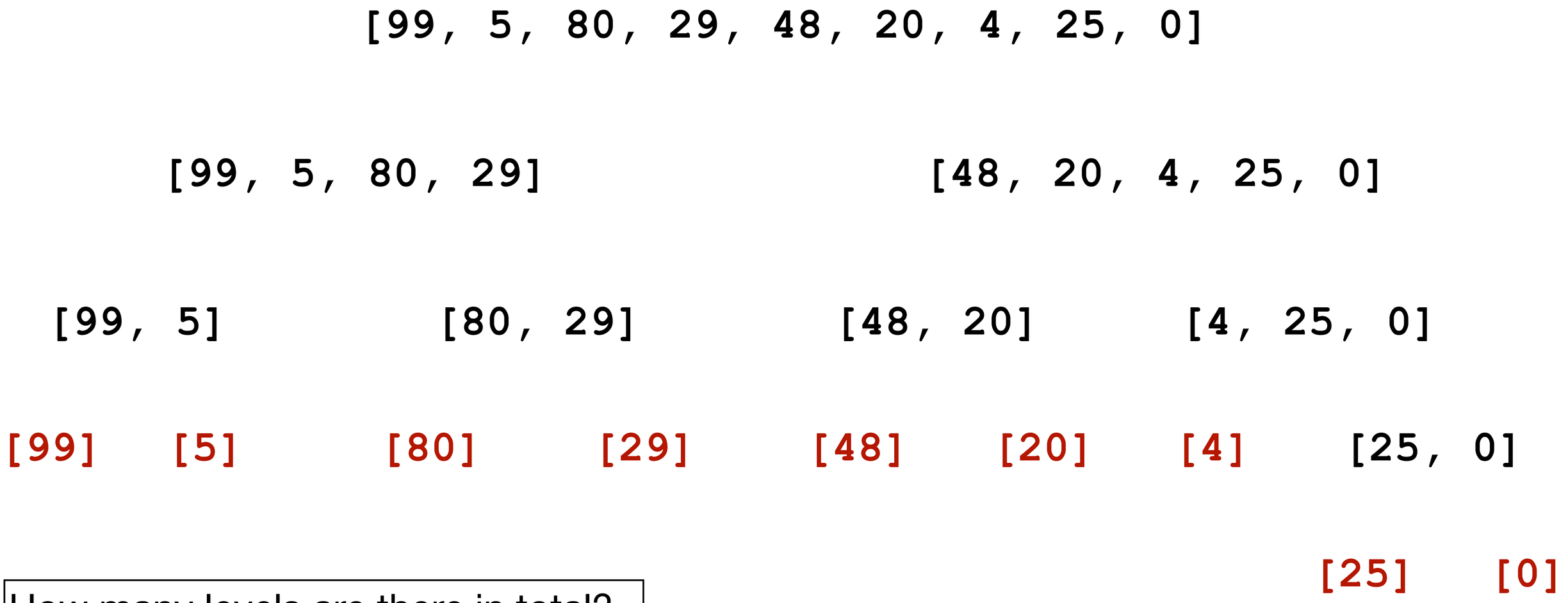
We thus spend $O(n)$ on **`merge`** *at each level*.

Merge Sort



How many levels are there in total?

Merge Sort



How many levels are there in total?

$O(\log_2 n)$

Merge Sort

Complexity

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

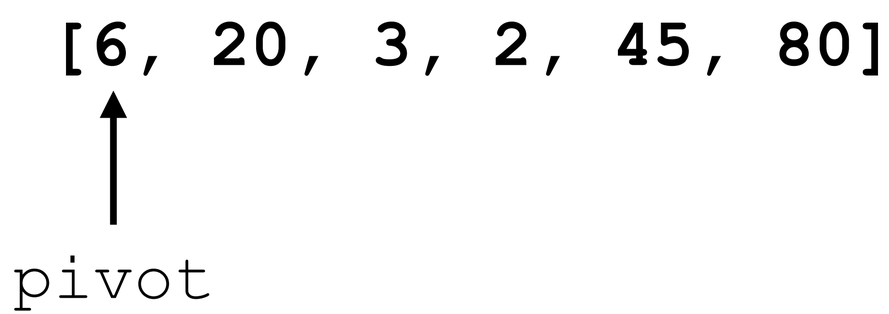
- Divide-and-conquer Algorithm:
  1. Divide an array in ~half — **Divide**
  2. Sort both arrays individually — **Conquer**
  3. Combine the two arrays — **Glue**

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

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



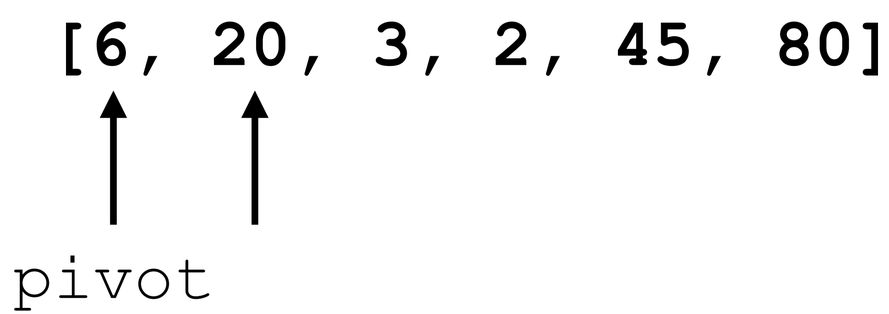
[

Less than pivot

[

Greater than pivot

Quick Sort



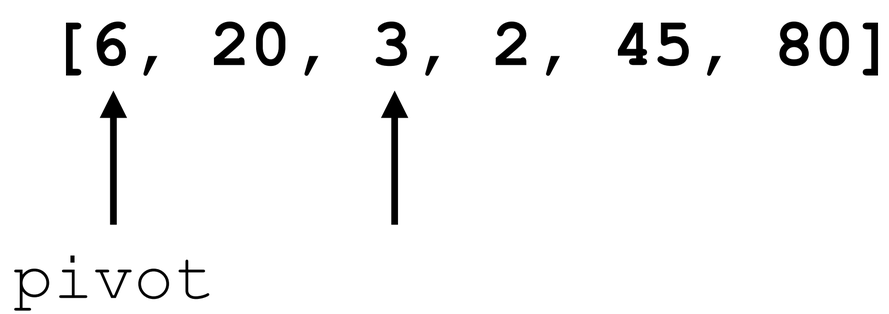
`[`

Less than pivot

`[20,`

Greater than pivot

Quick Sort



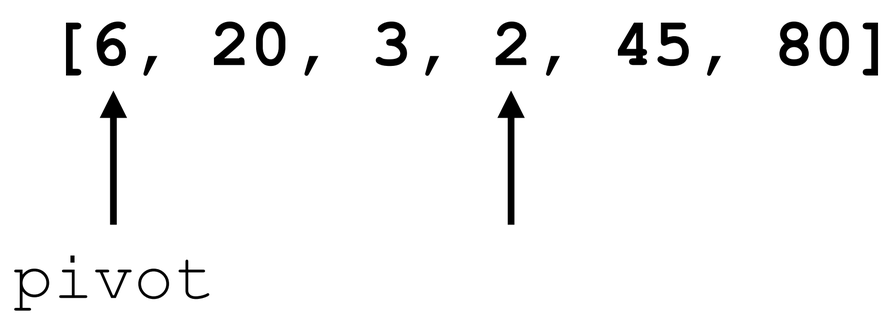
`[3,`

Less than pivot

`[20,`

Greater than pivot

Quick Sort



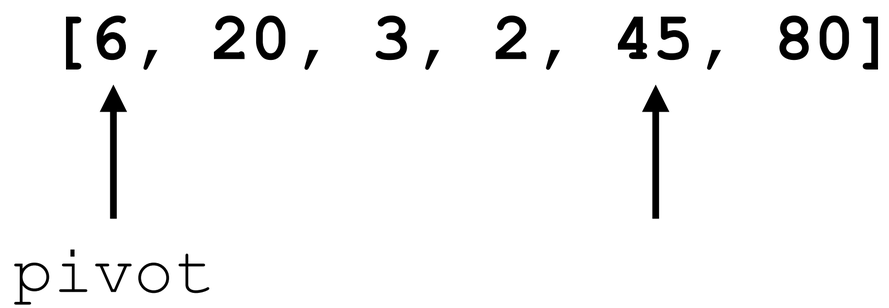
`[3, 2,`

Less than pivot

`[20,`

Greater than pivot

Quick Sort



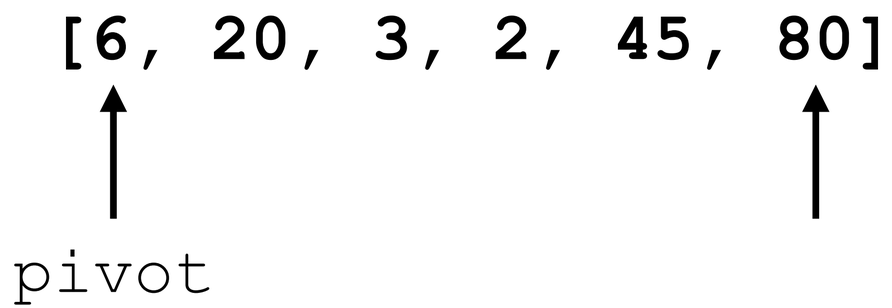
`[3, 2,`

Less than pivot

`[20, 45,`

Greater than pivot

Quick Sort



`[3, 2]`

Less than pivot

`[20, 45, 80]`

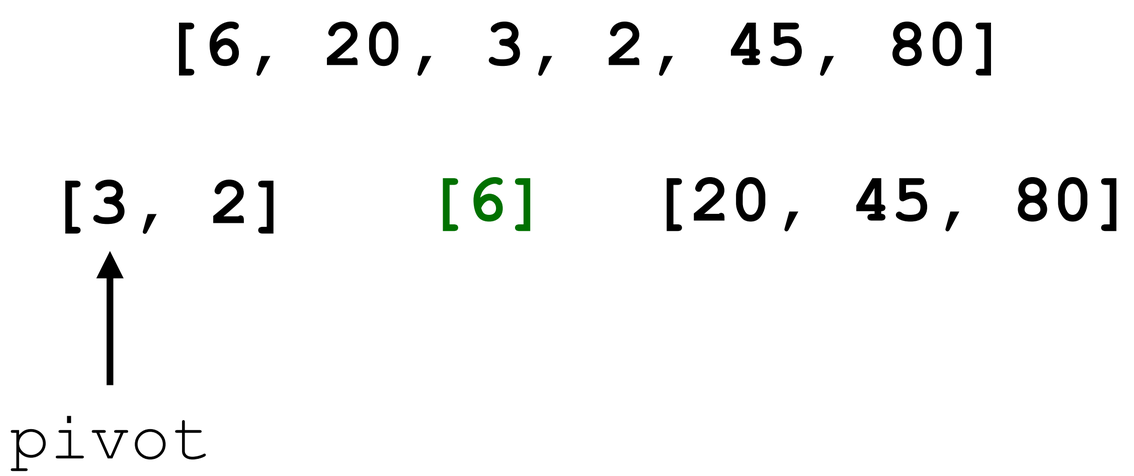
Greater than pivot

Quick Sort

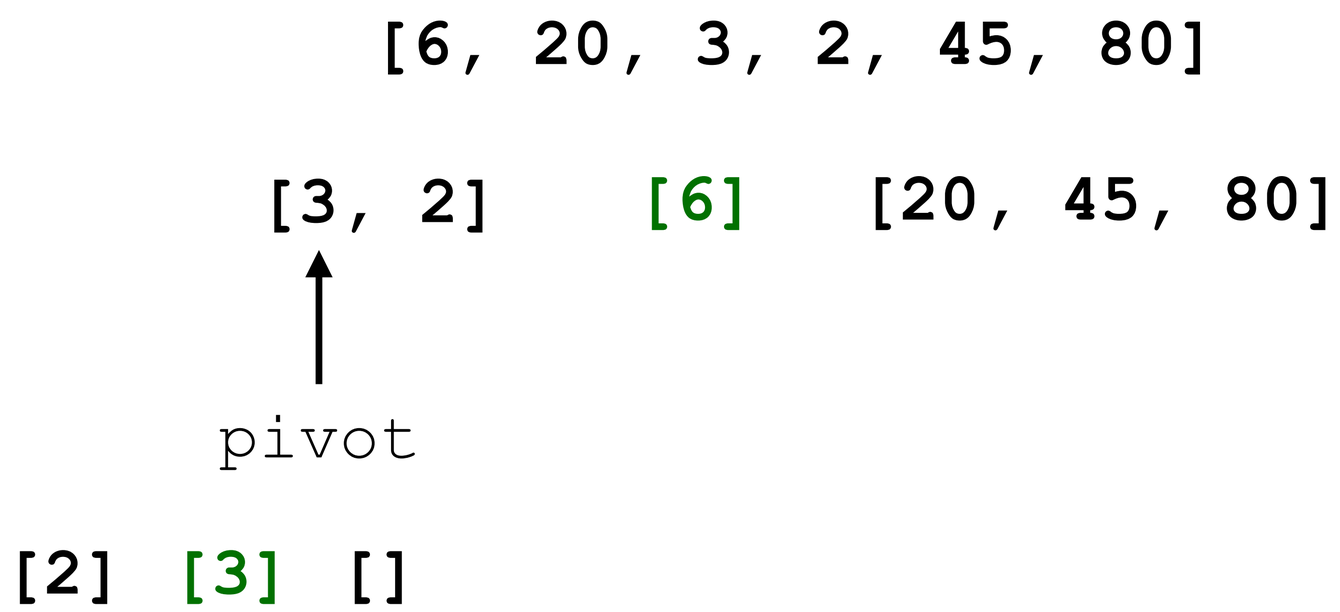
```
[6, 20, 3, 2, 45, 80]

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

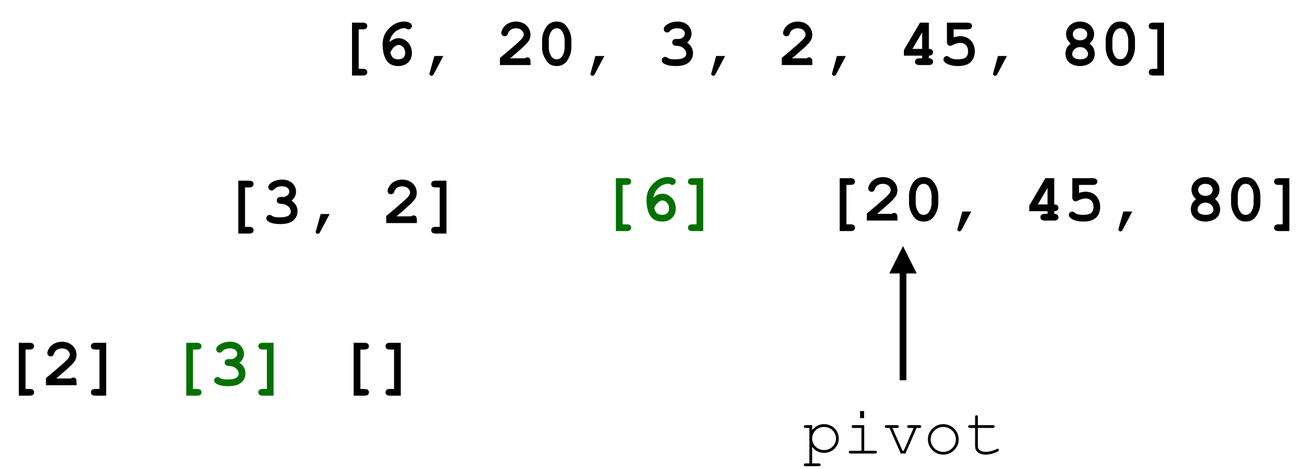
Quick Sort



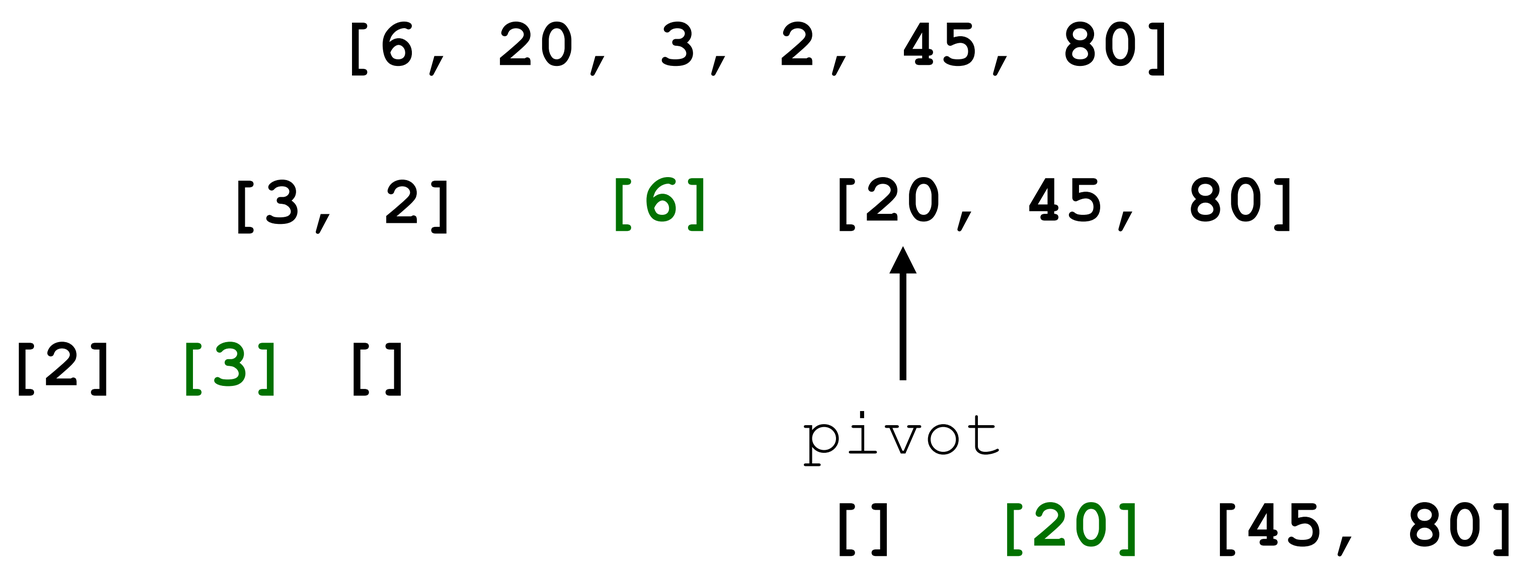
Quick Sort



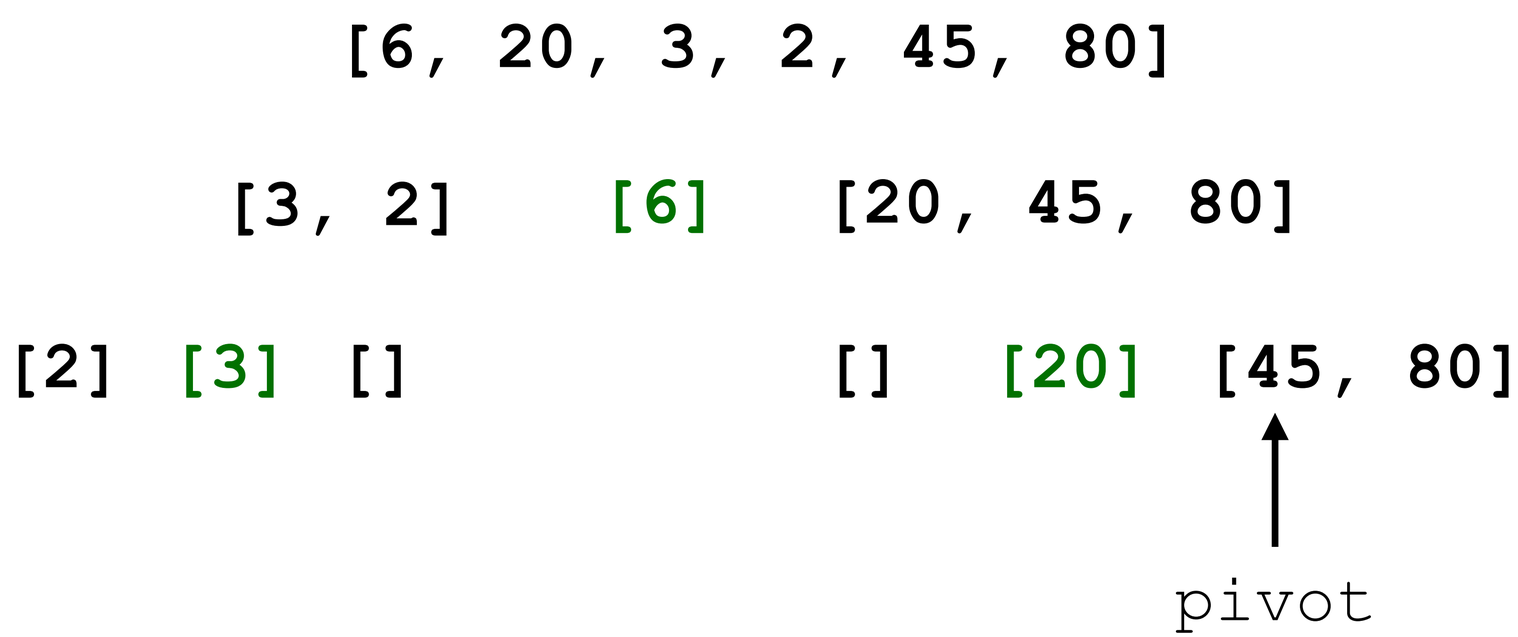
Quick Sort



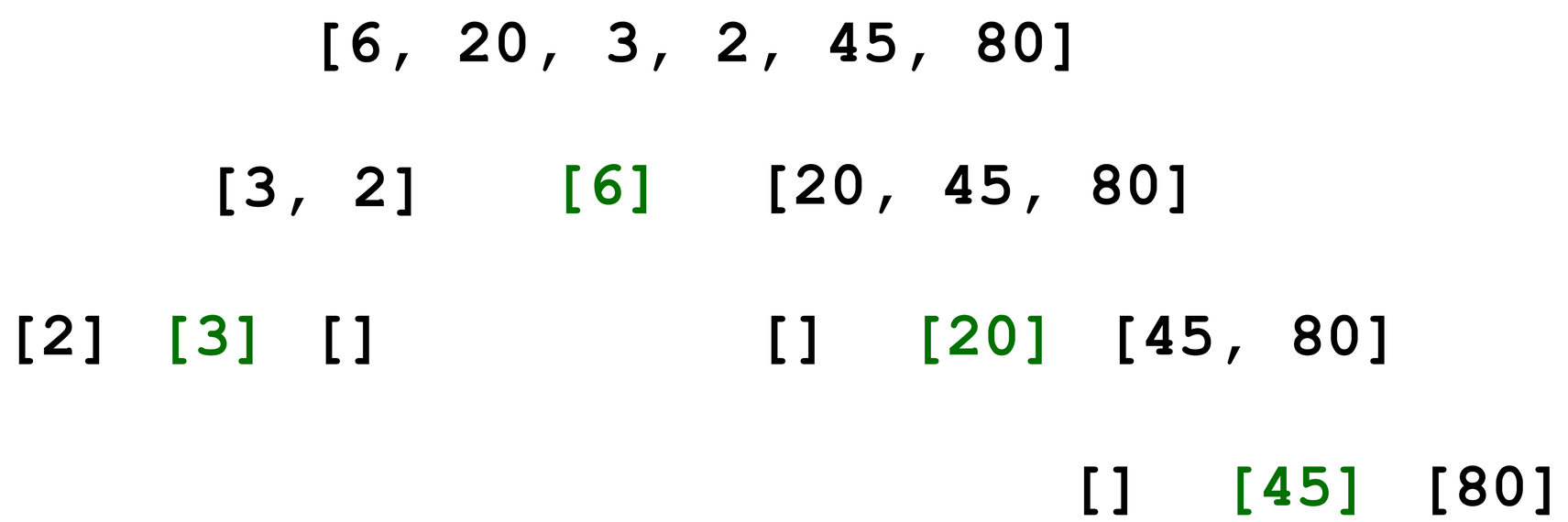
Quick Sort



Quick Sort



Quick Sort



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`**

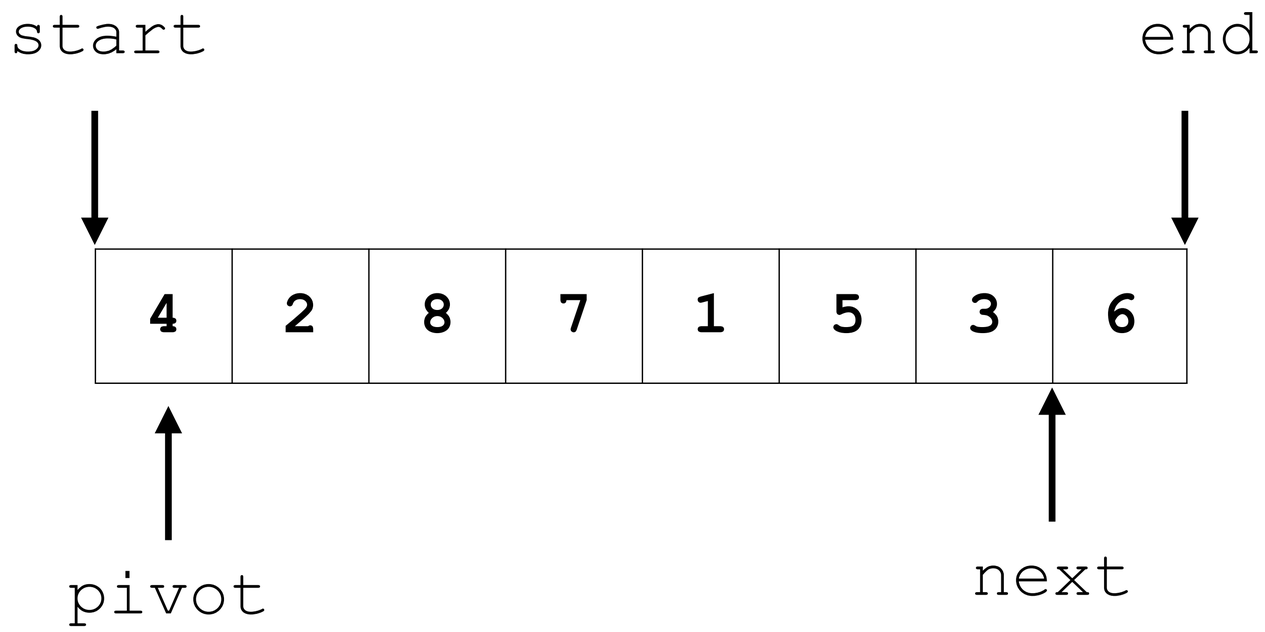
Quick Sort

Partition

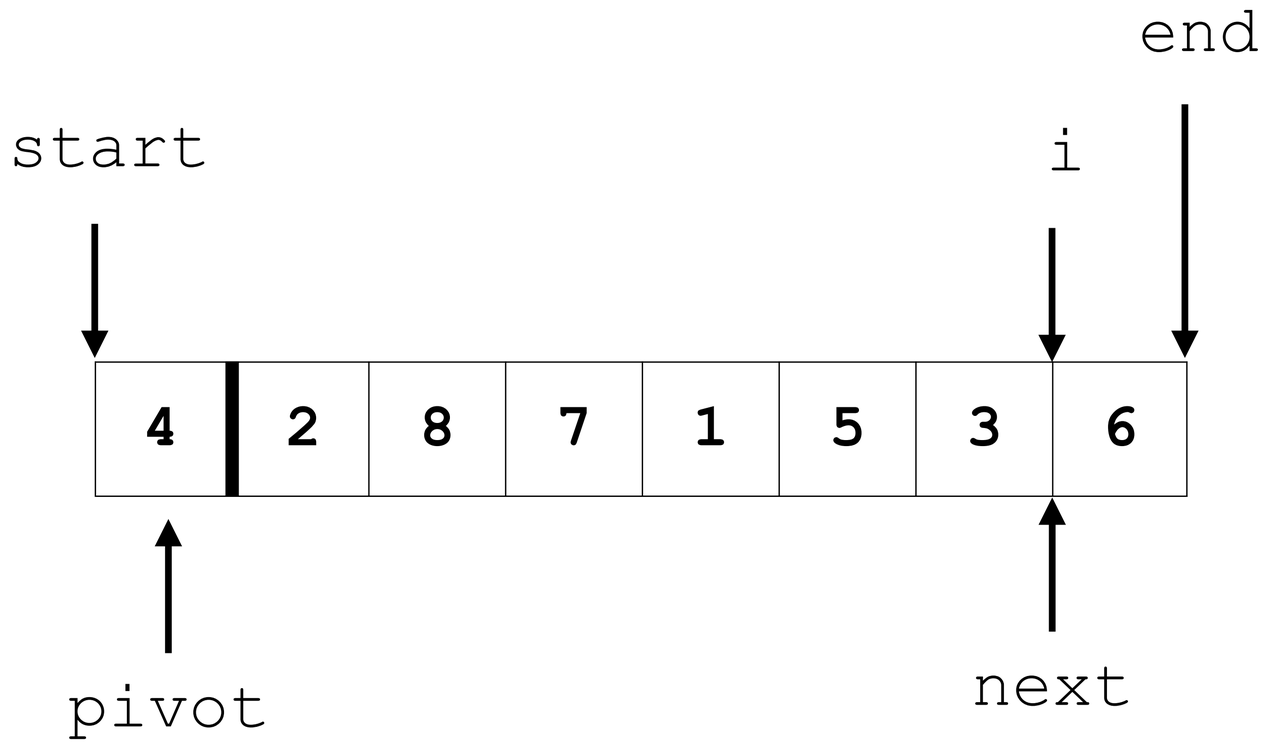
```
int partition(int *arr, int start, int end)
{
    int pivot = arr[start];
    int next = end - 1;

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

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

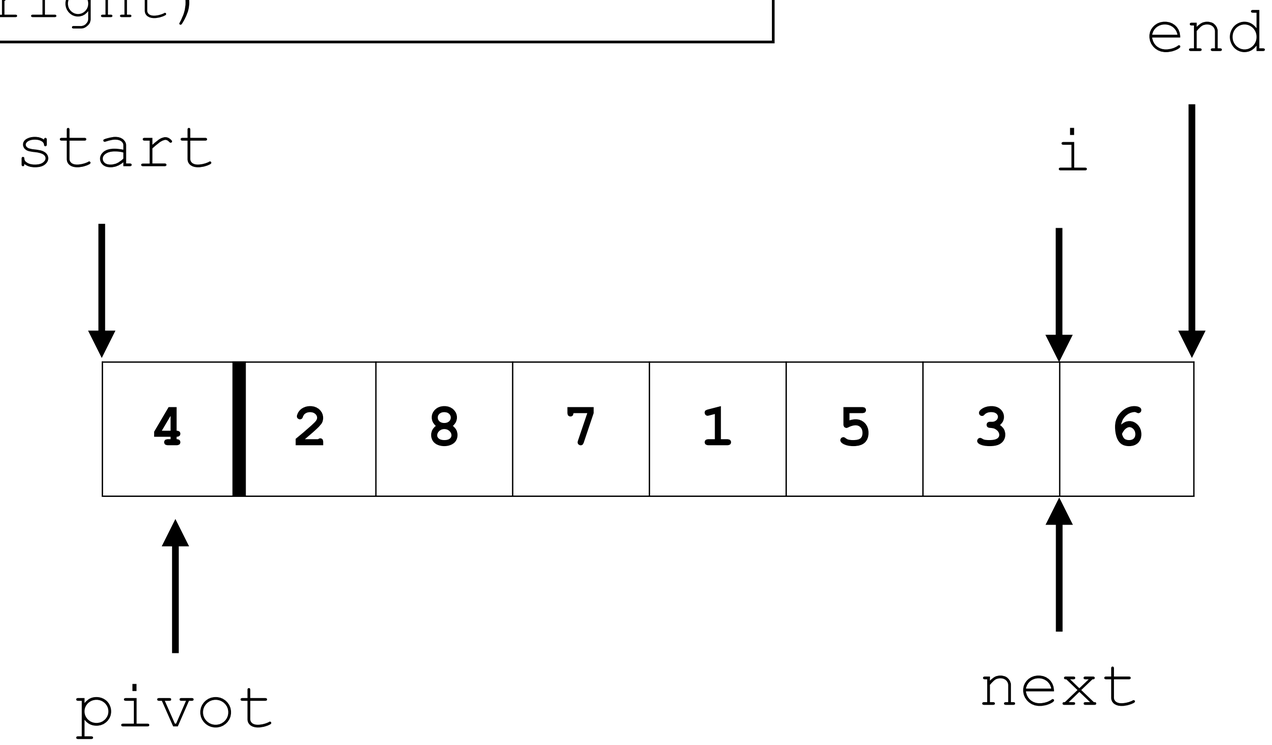



```
int pivot = arr[start];
int next = end - 1;
```



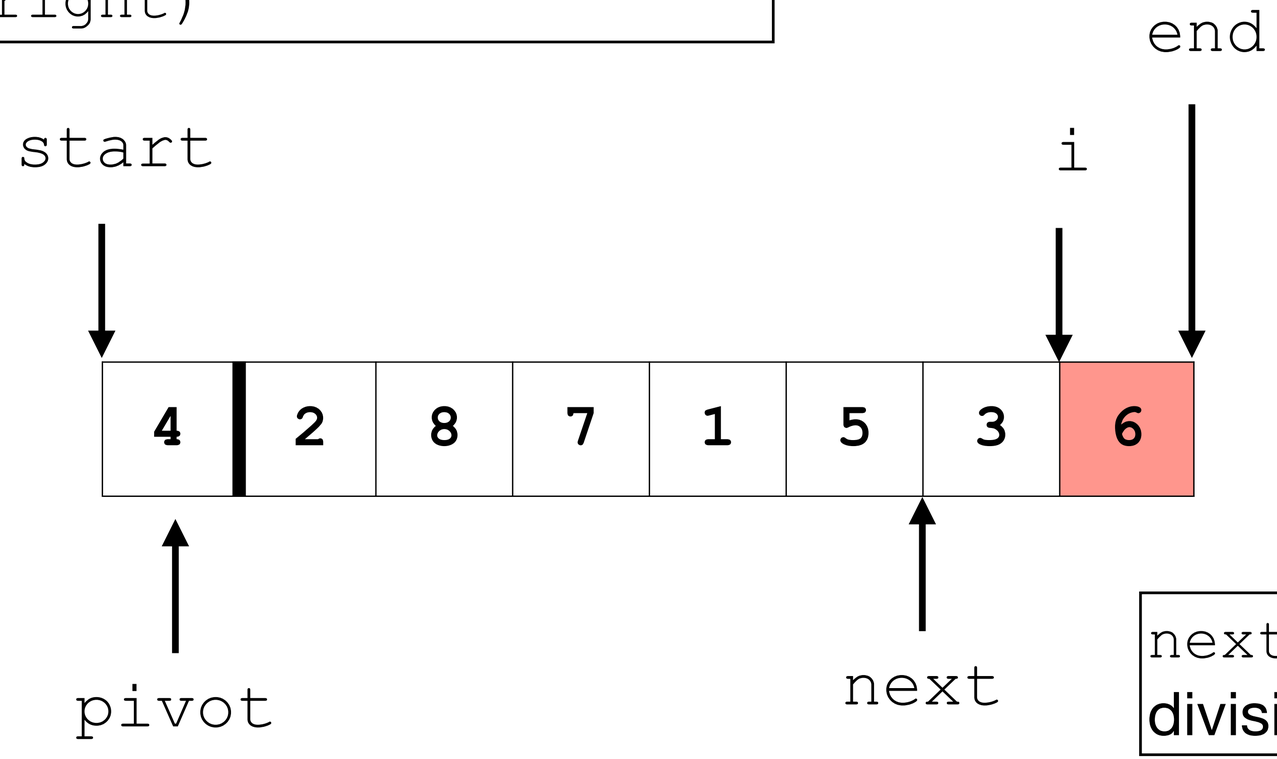

```
for (int i = end - 1; i > start; i--)
    if (arr[i] > pivot)
        swap(arr, next--, i);
```

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




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

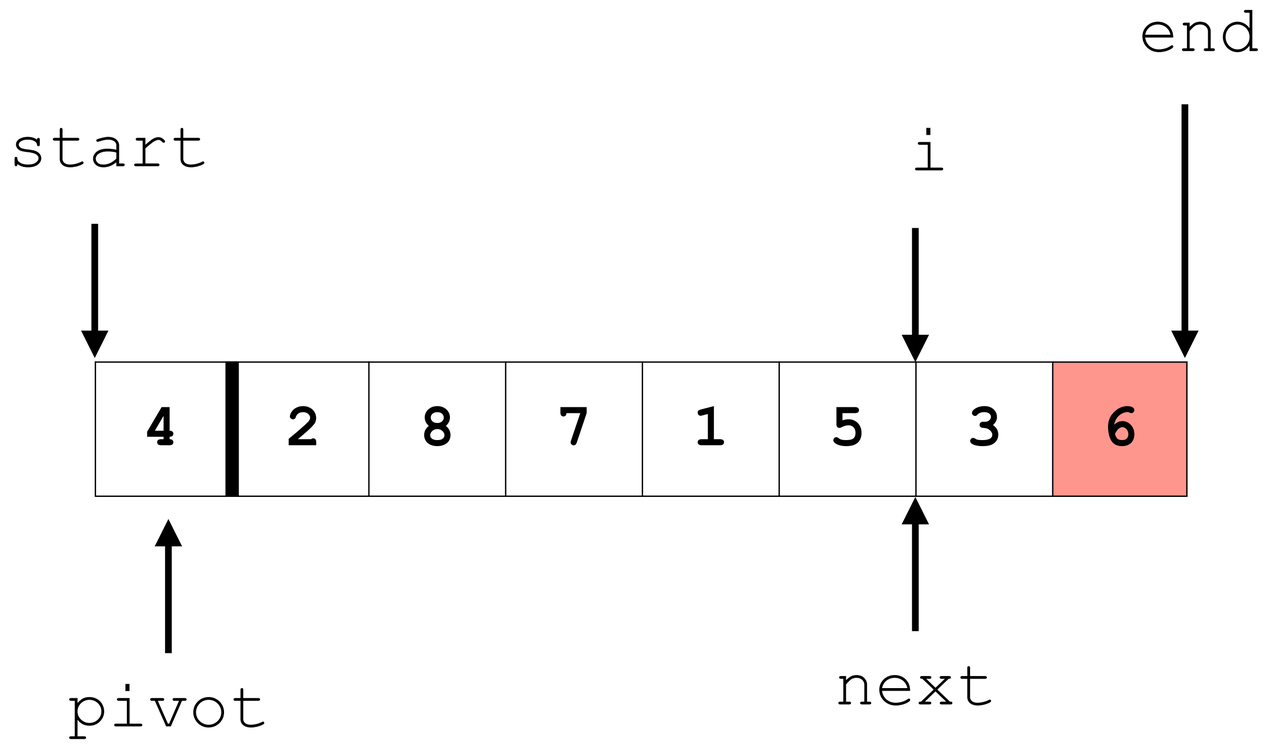
```
arr[i] > pivot?     -- yes
swap(arr, next, right)
```



`next + 1` is the division line.

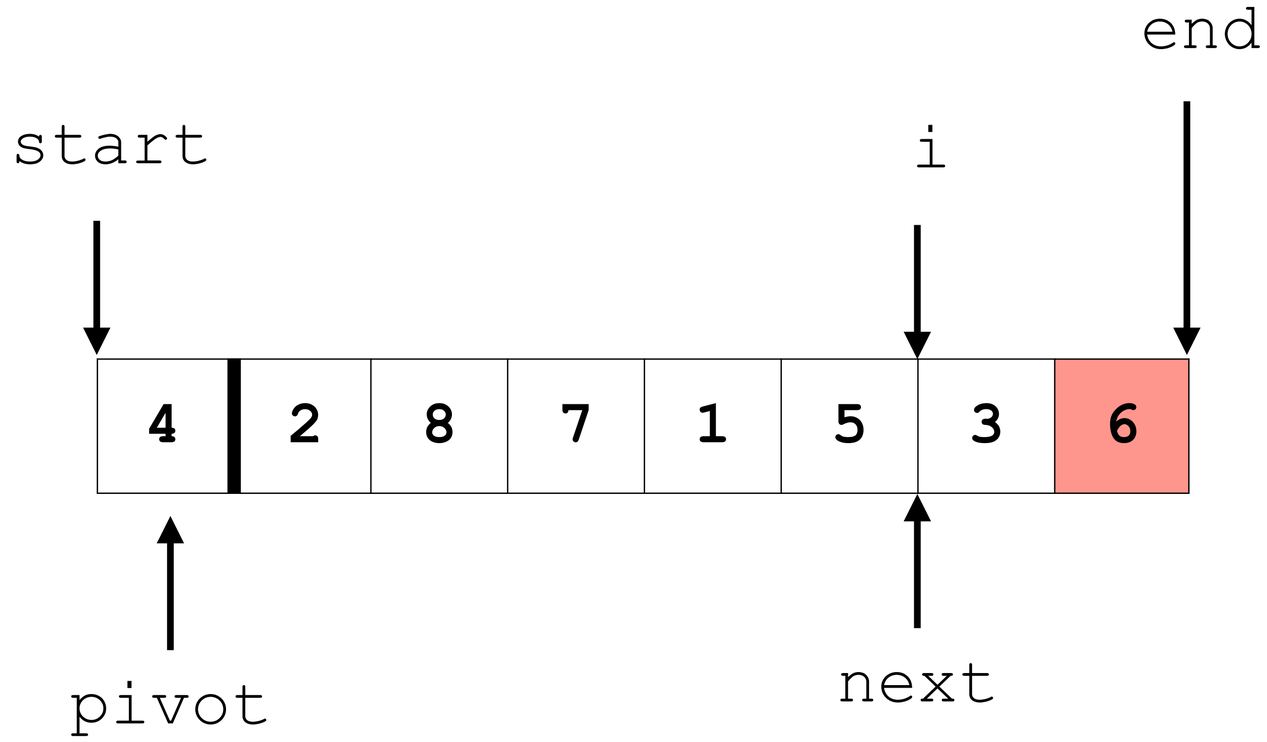
`i + 1` is the start of partitioned sub-array.

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

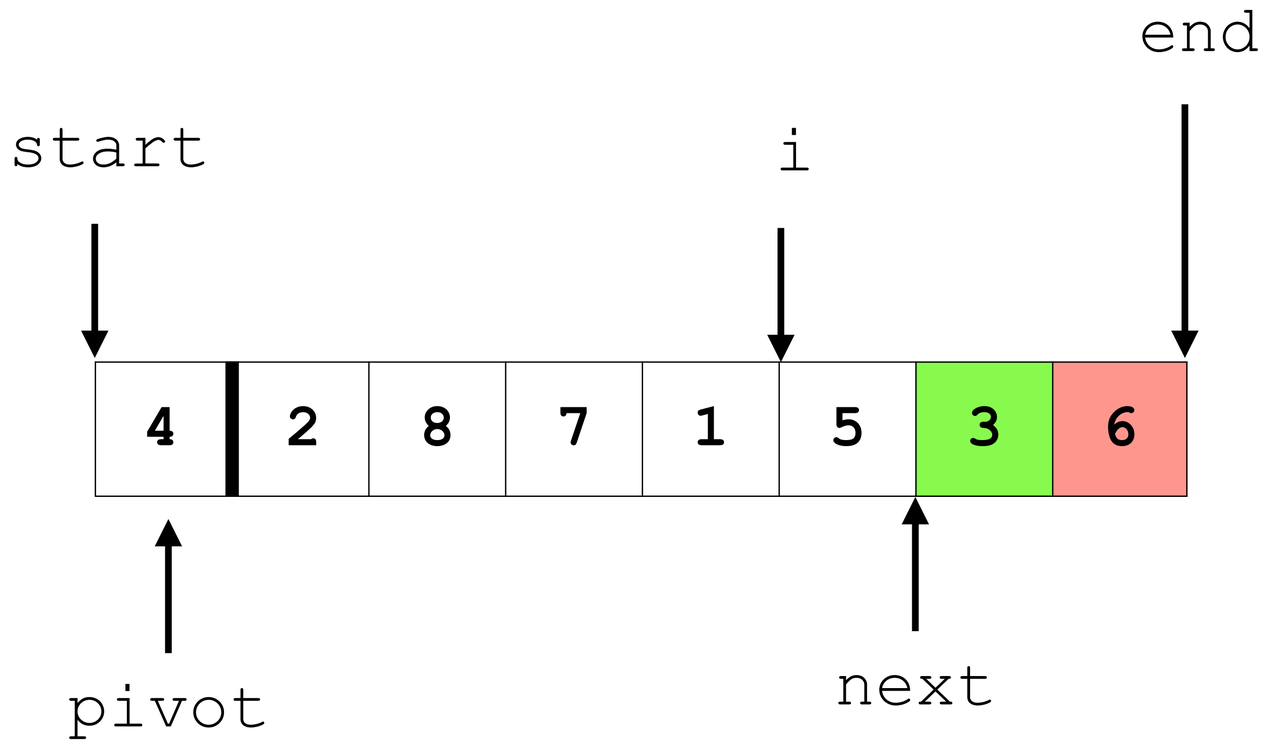



```
for (int i = end - 1; i > start; i--)
    if (arr[i] > pivot)
        swap(arr, next--, i);
```

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

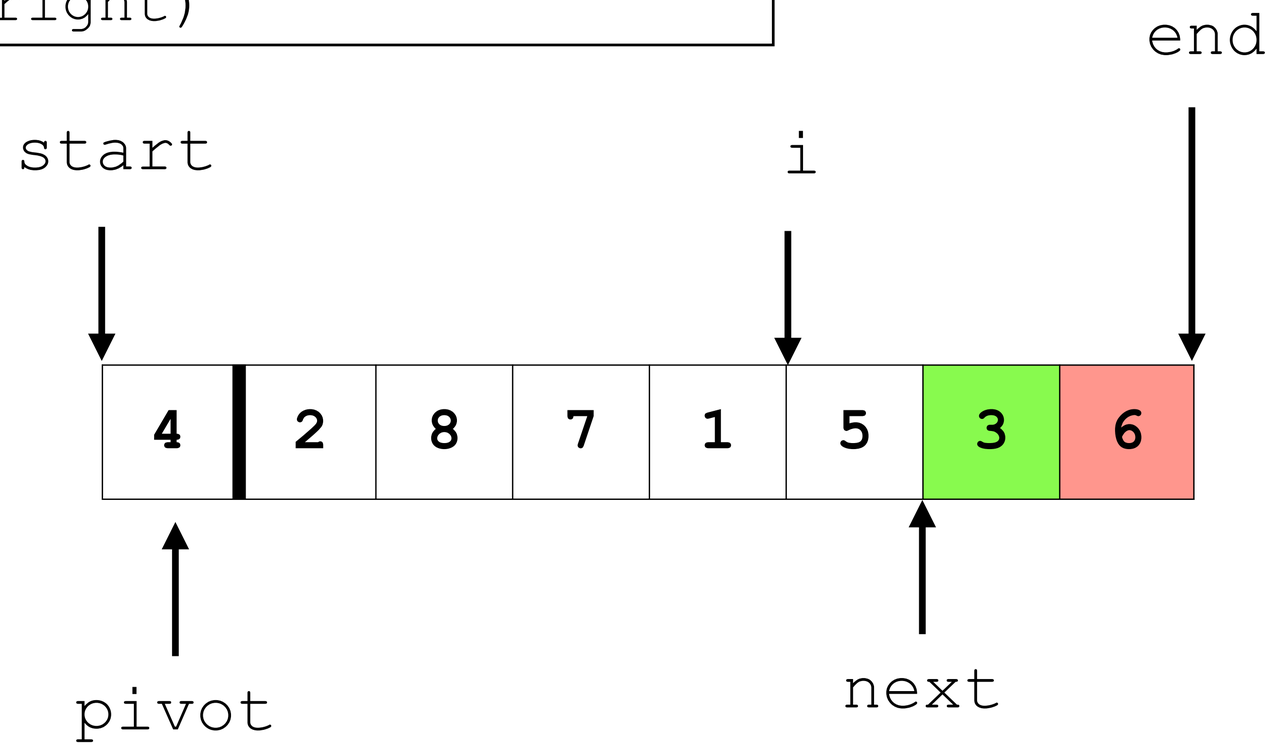



```
for (int i = end - 1; i > start; i--)
    if (arr[i] > pivot)
        swap(arr, next--, i);
```



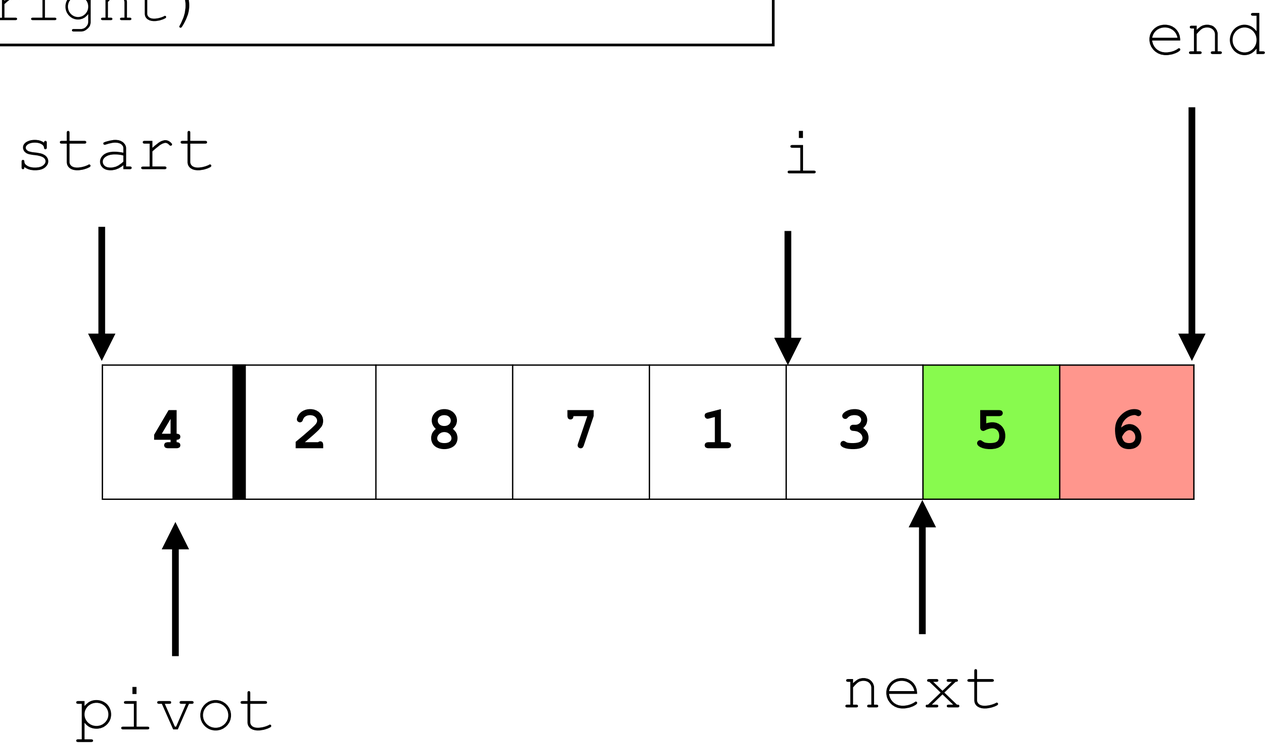

```
for (int i = end - 1; i > start; i--)
    if (arr[i] > pivot)
        swap(arr, next--, i);
```

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



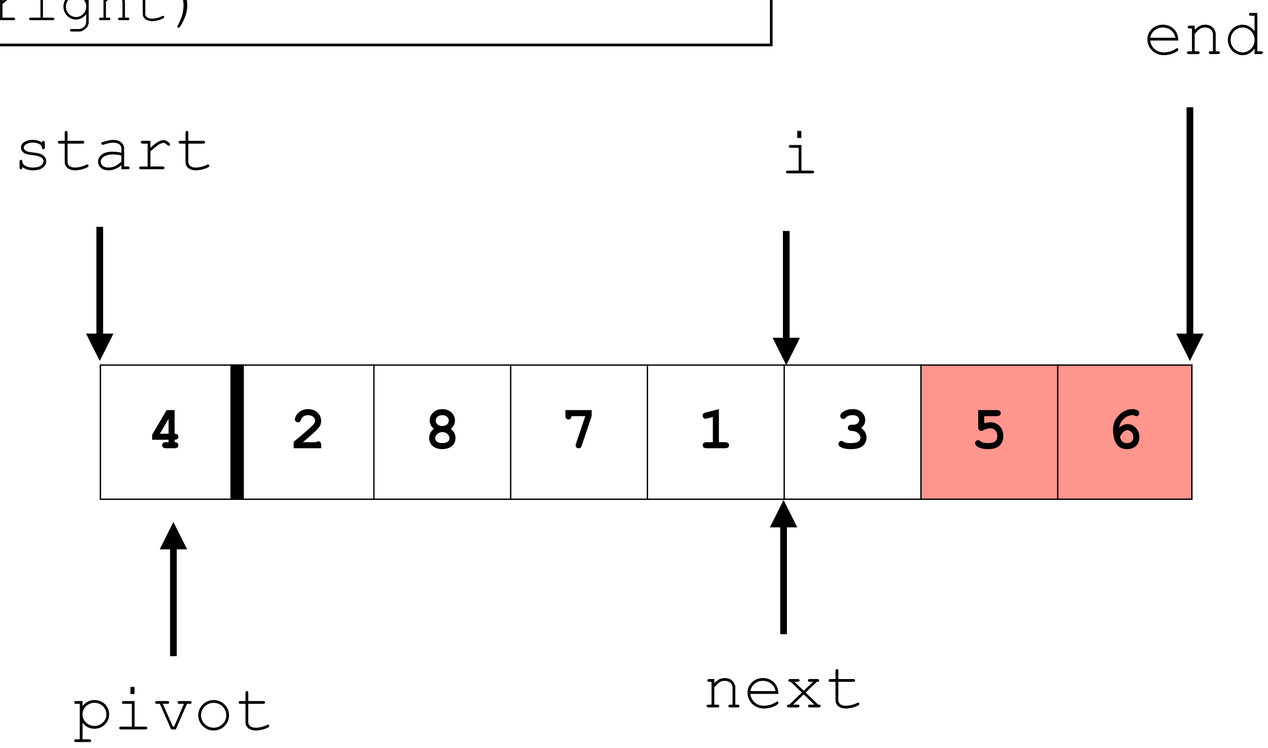

```
for (int i = end - 1; i > start; i--)
    if (arr[i] > pivot)
        swap(arr, next--, i);
```

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



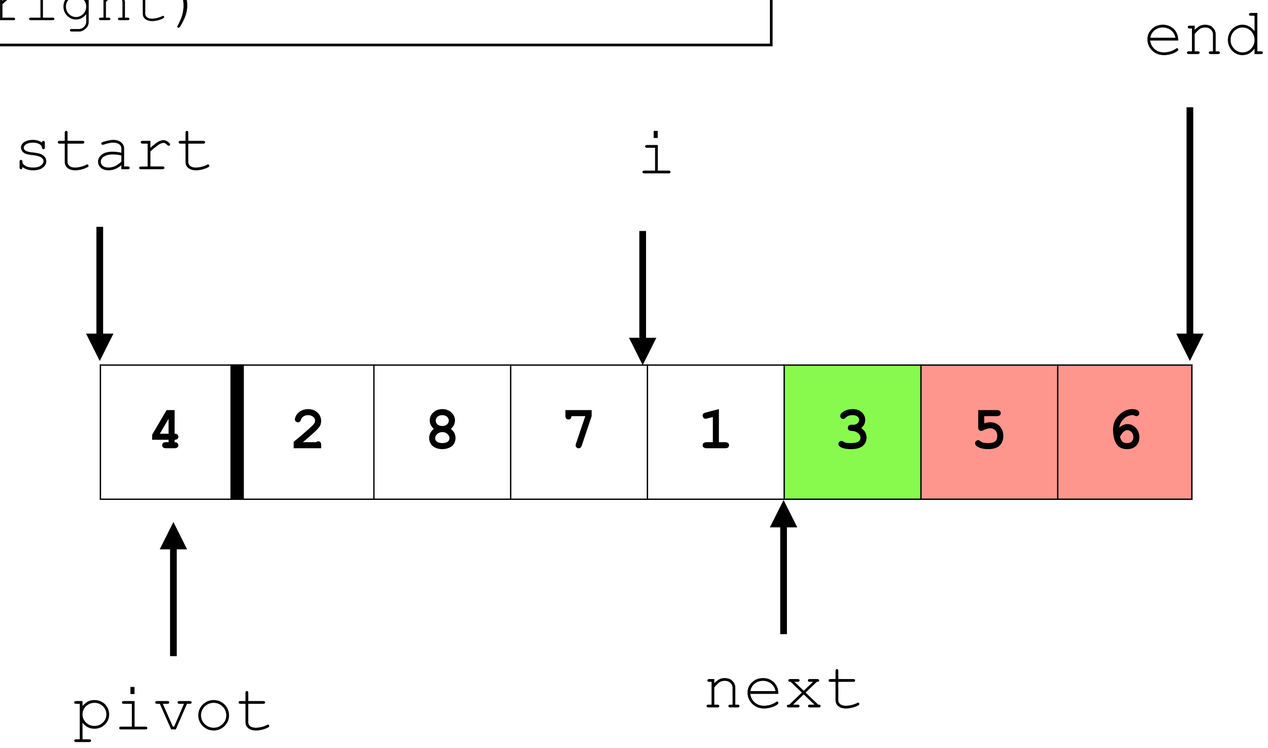

```
for (int i = end - 1; i > start; i--)
    if (arr[i] > pivot)
        swap(arr, next--, i);
```

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



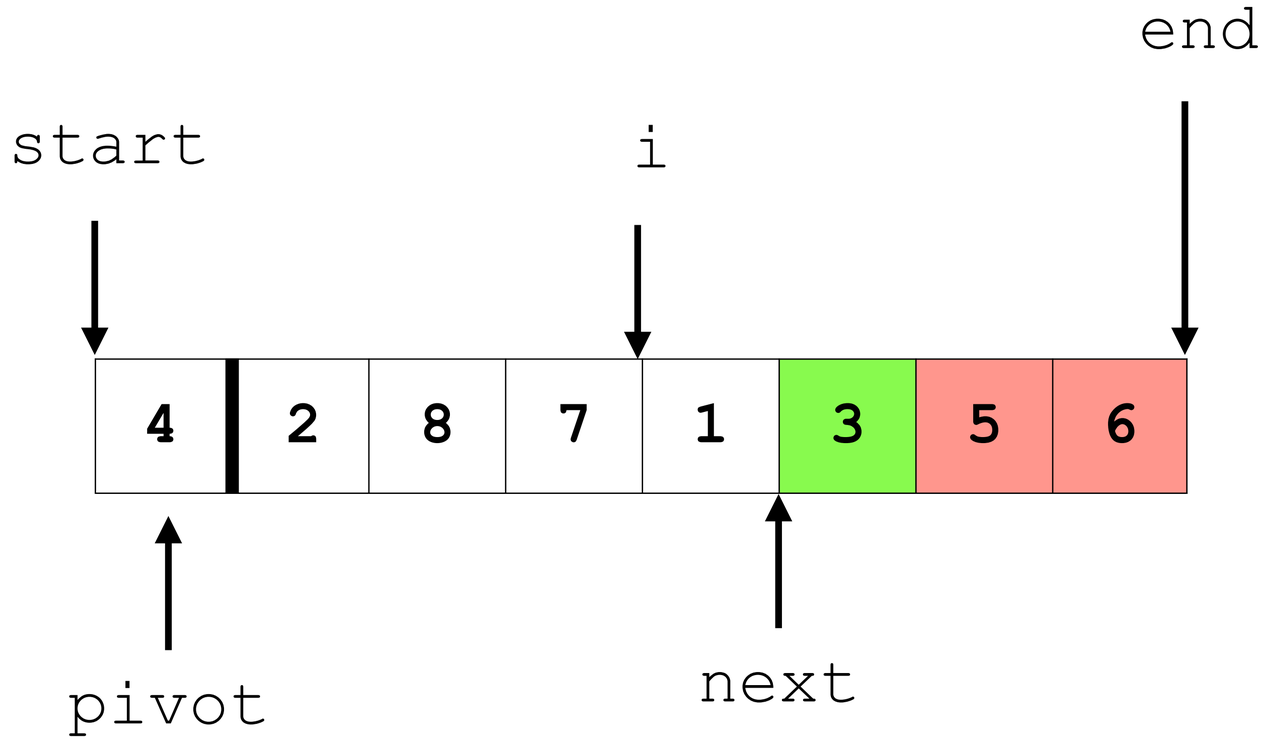

```
for (int i = end - 1; i > start; i--)
    if (arr[i] > pivot)
        swap(arr, next--, i);
```

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

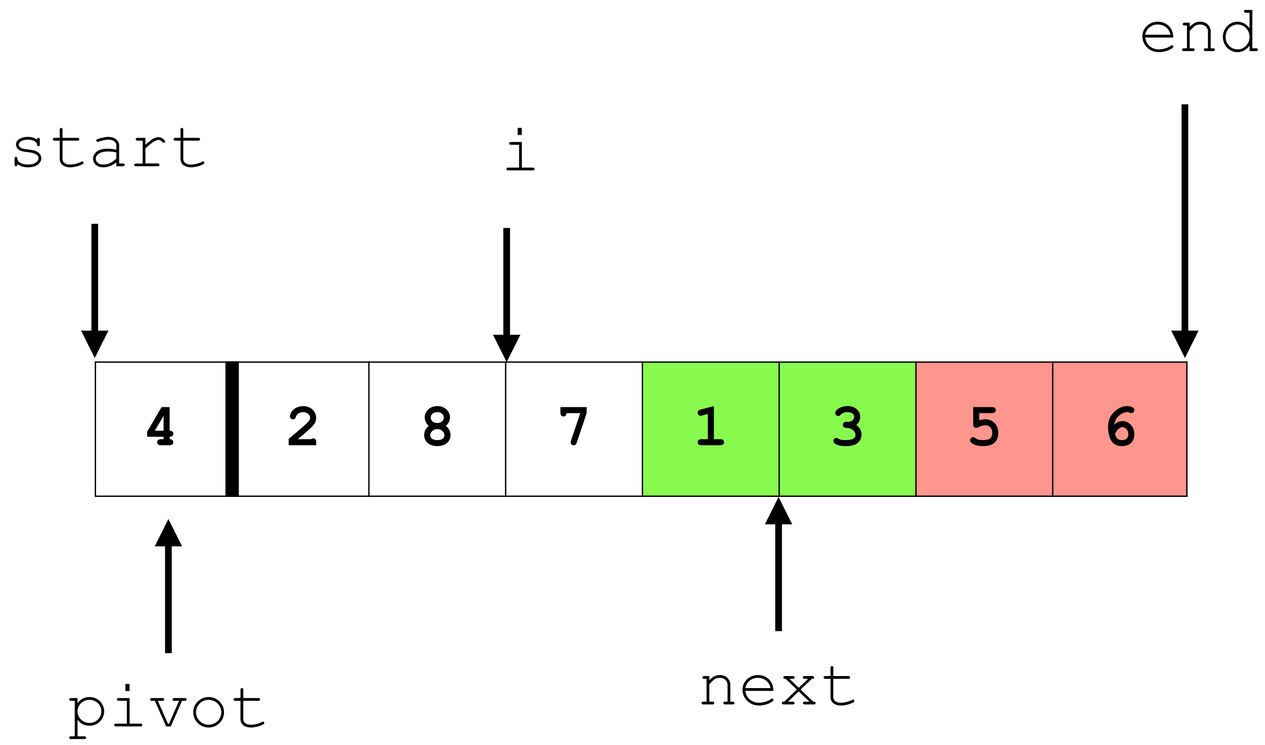



```
for (int i = end - 1; i > start; i--)
    if (arr[i] > pivot)
        swap(arr, next--, i);
```

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

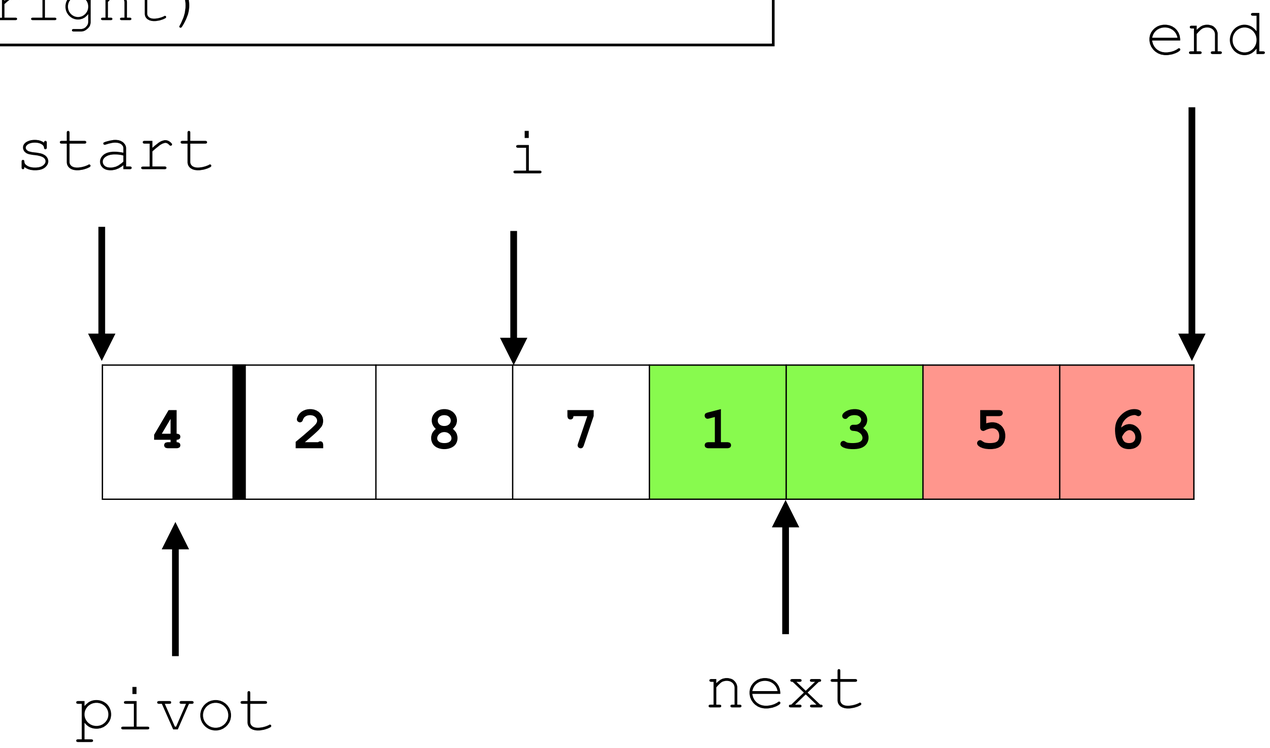



```
for (int i = end - 1; i > start; i--)
    if (arr[i] > pivot)
        swap(arr, next--, i);
```



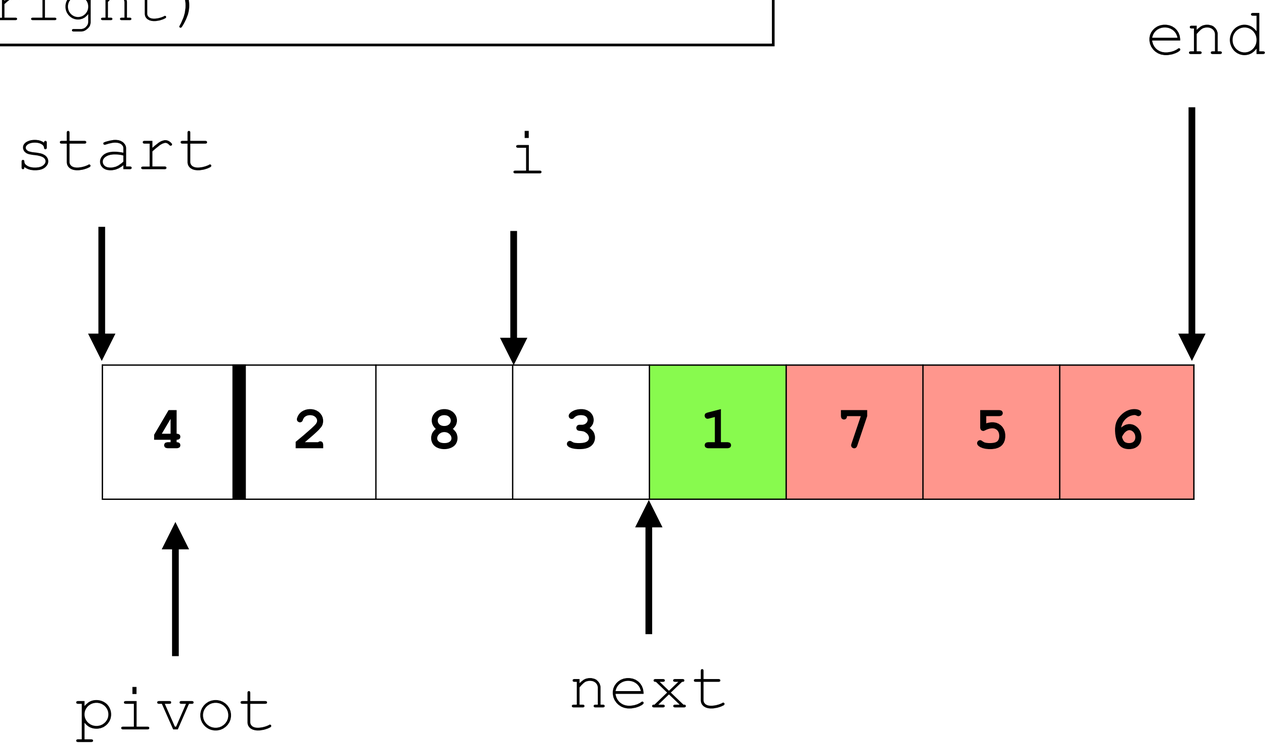

```
for (int i = end - 1; i > start; i--)
    if (arr[i] > pivot)
        swap(arr, next--, i);
```

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

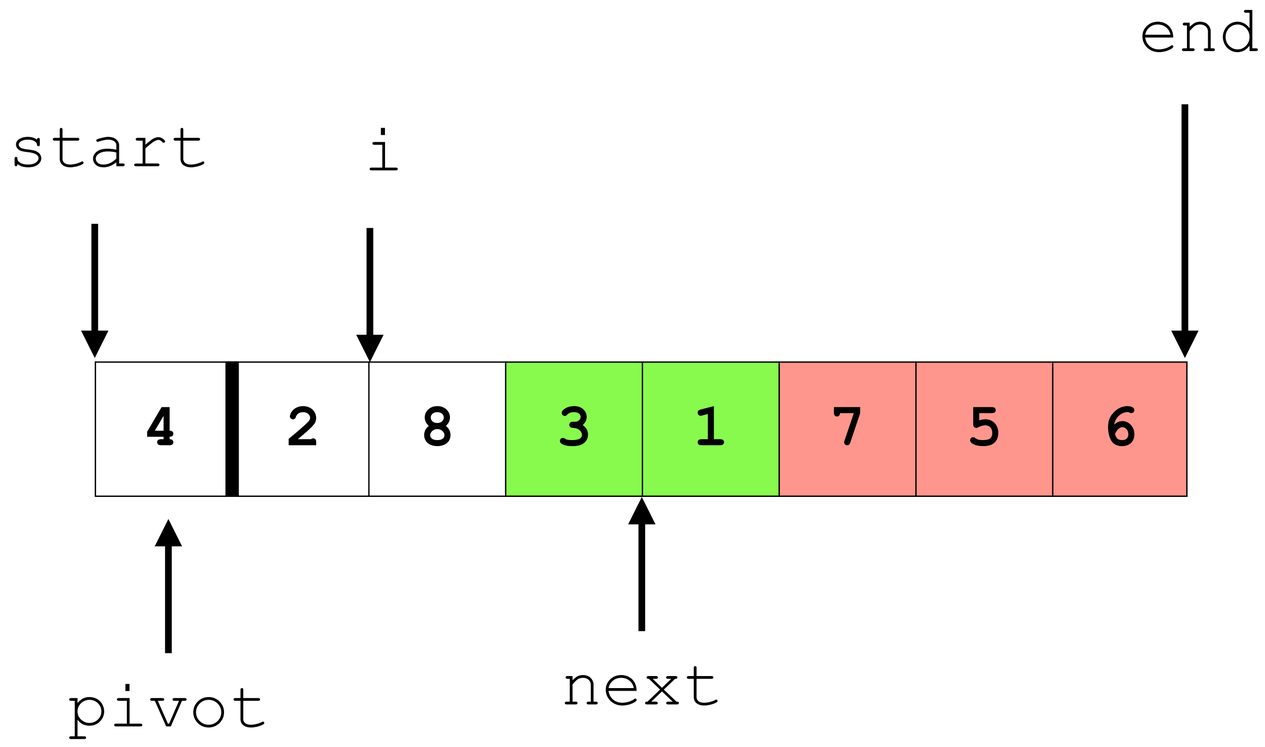



```
for (int i = end - 1; i > start; i--)
    if (arr[i] > pivot)
        swap(arr, next--, i);
```

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

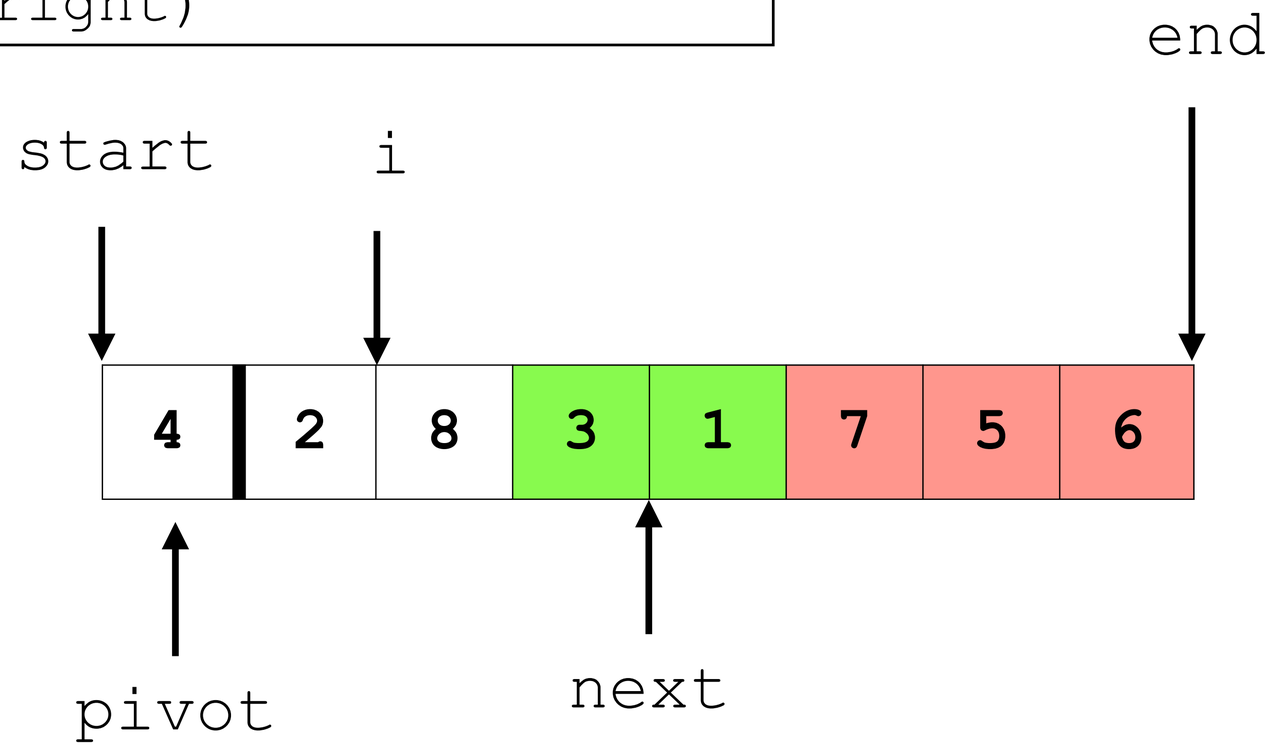



```
for (int i = end - 1; i > start; i--)
    if (arr[i] > pivot)
        swap(arr, next--, i);
```



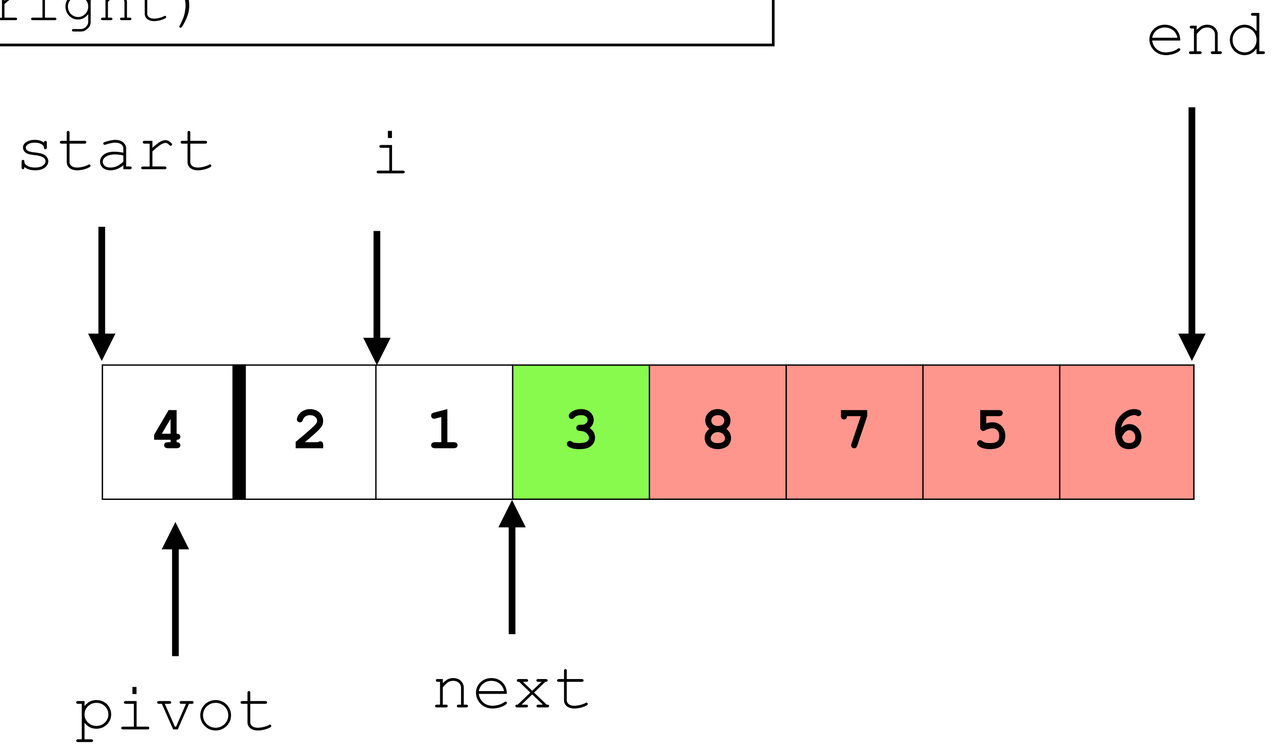

```
for (int i = end - 1; i > start; i--)
    if (arr[i] > pivot)
        swap(arr, next--, i);
```

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

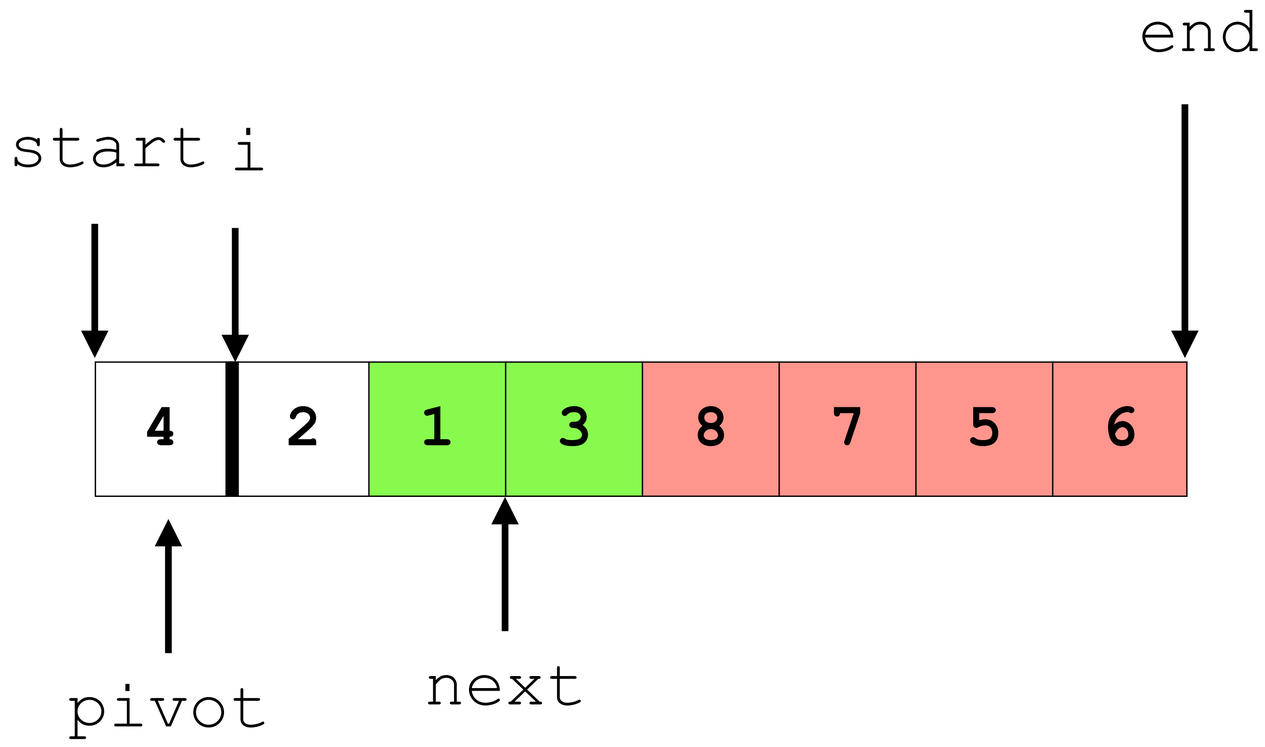



```
for (int i = end - 1; i > start; i--)
    if (arr[i] > pivot)
        swap(arr, next--, i);
```

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

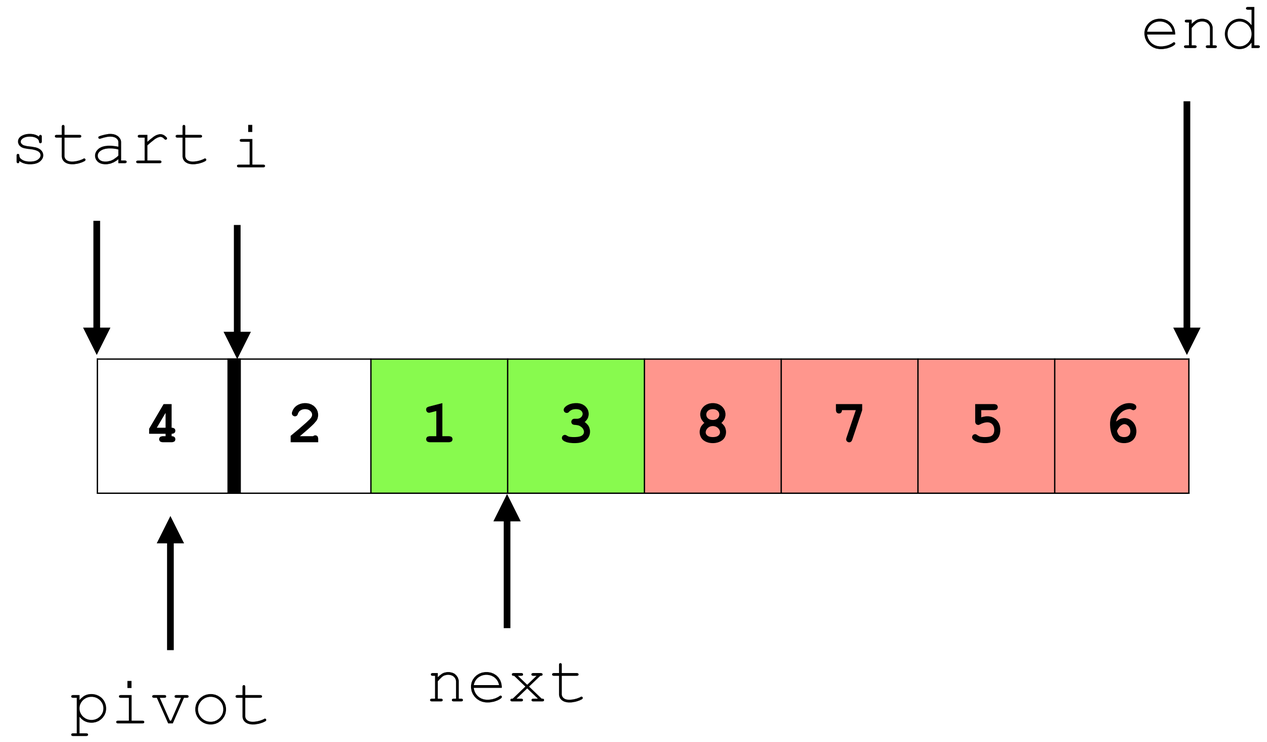



```
for (int i = end - 1; i > start; i--)
    if (arr[i] > pivot)
        swap(arr, next--, i);
```

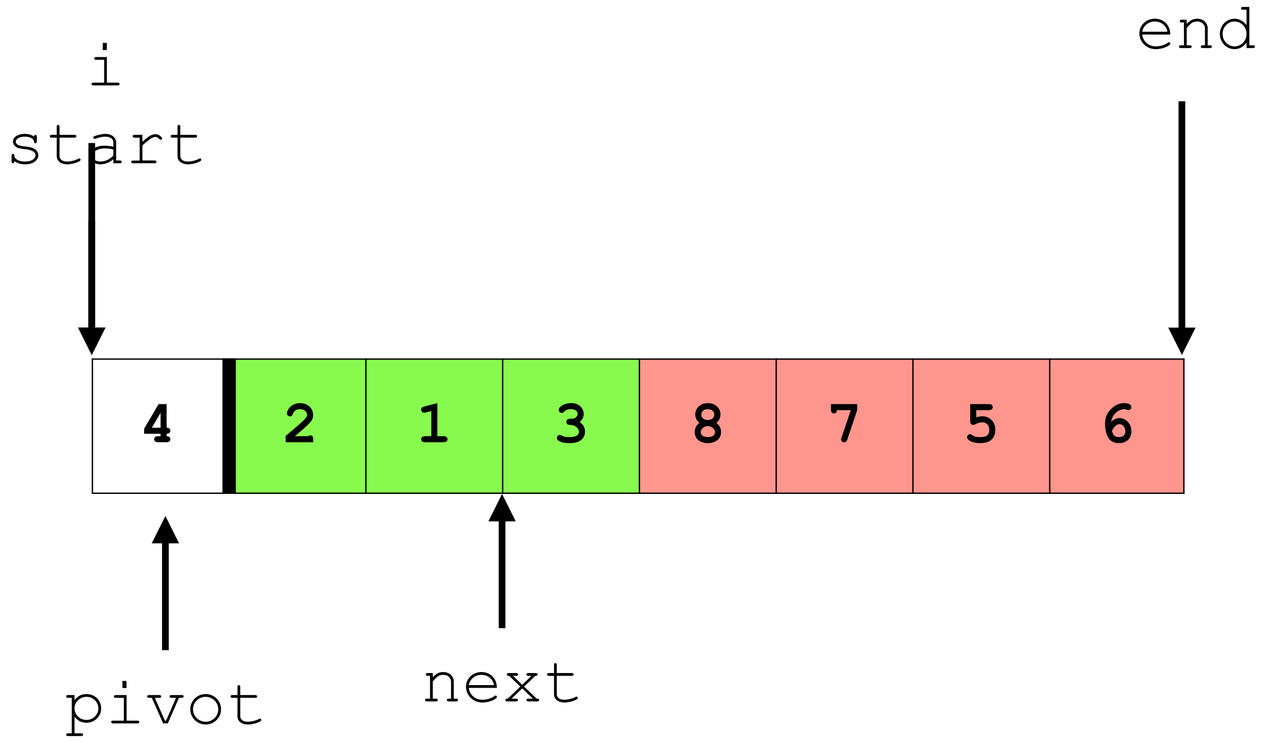



```
for (int i = end - 1; i > start; i--)
    if (arr[i] > pivot)
        swap(arr, next--, i);
```

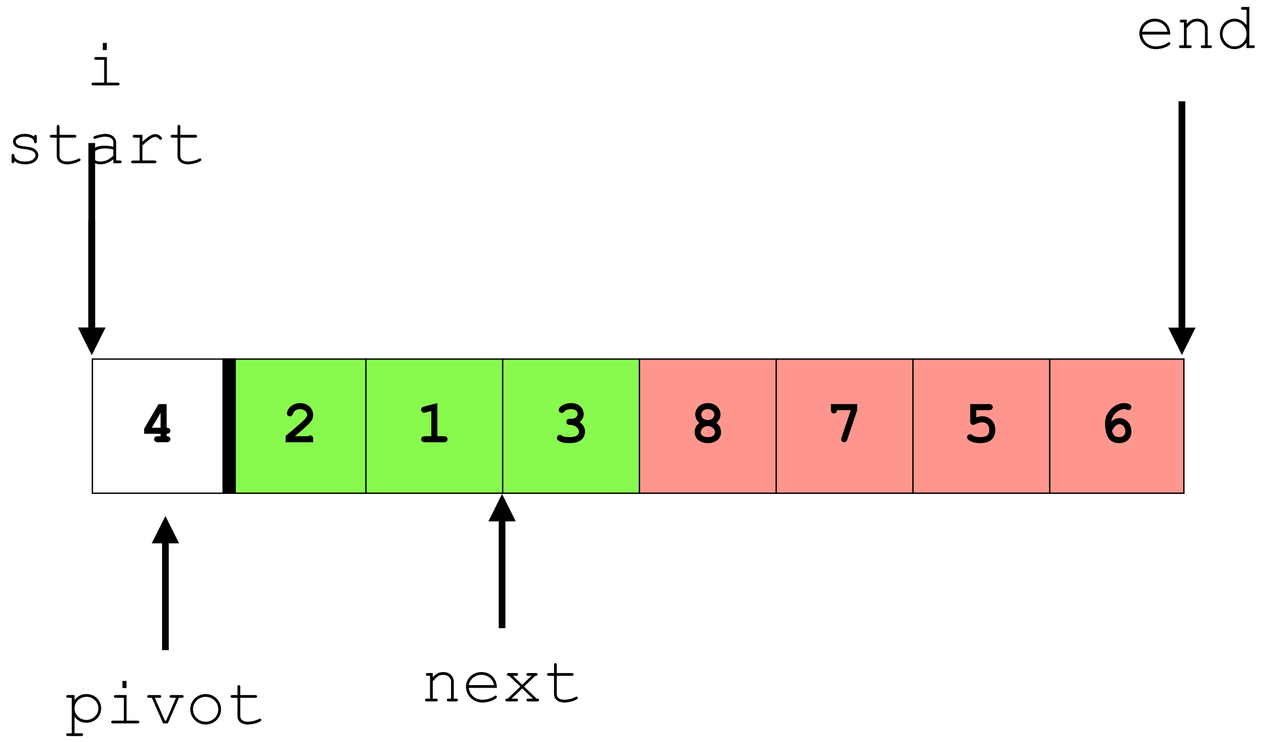
arr[i] > pivot? -- no



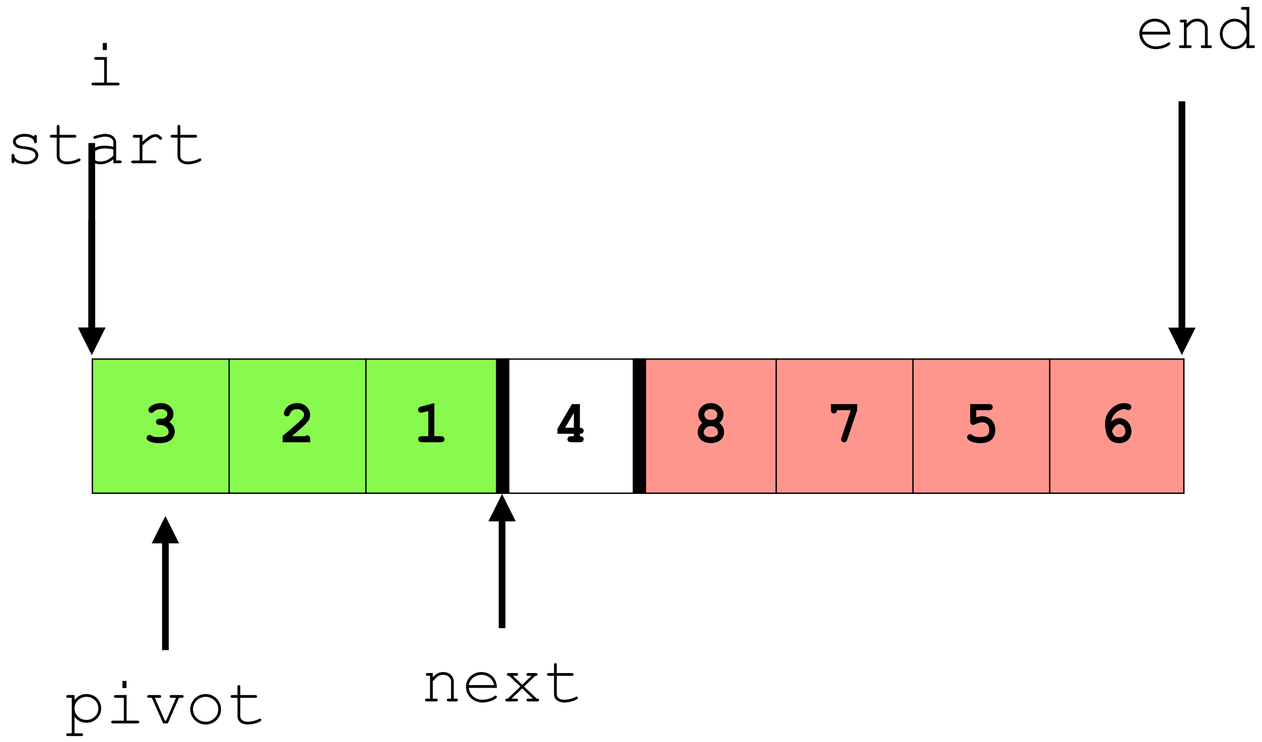

```
for (int i = end - 1; i > start; i--)
    if (arr[i] > pivot)
        swap(arr, next--, i);
```




```
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);
    }
}
```

Arrays with size <= 1 is already sorted.

Divide

Conquer

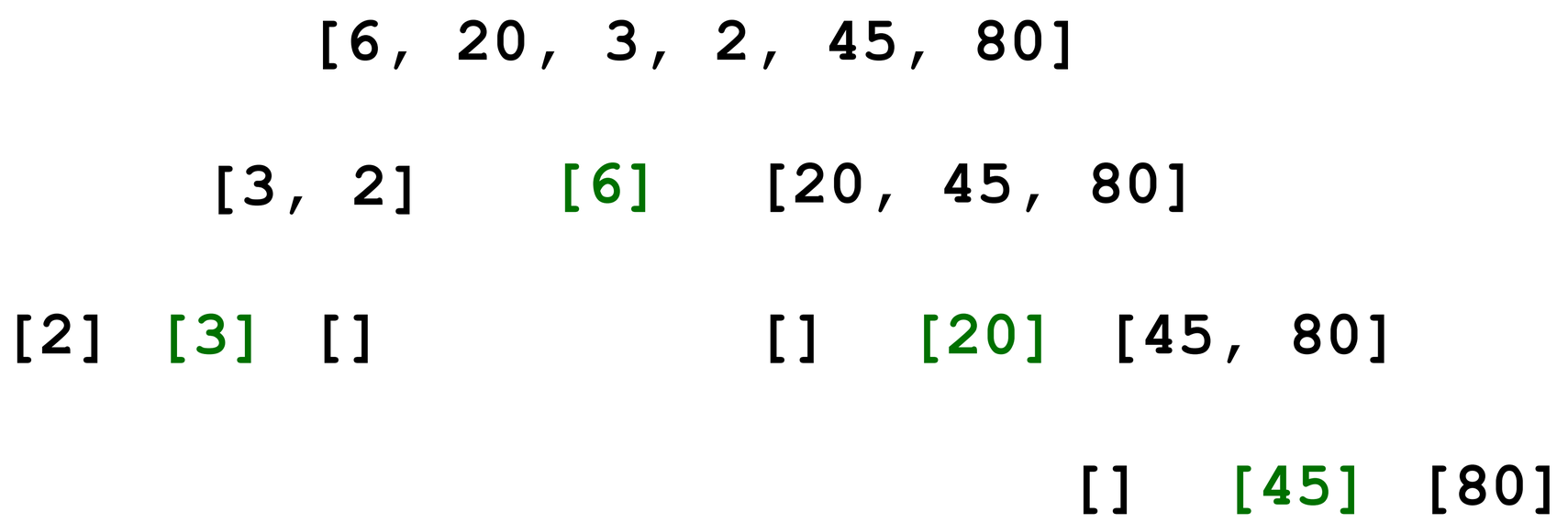
Quick Sort

Time Complexity

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

Quick Sort

Time Complexity



Depends on how good the partition is!

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

- 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

- 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