SCC.121: Fundamentals of Computer Science
Sorting, Trees and Graphs

Fabien Dufoulon

Plan for Weeks 16-20

Sorting, Trees and Graphs

- Week 16: Sorting Algorithms
- Week 17: Algorithms on Trees
- Week 18: Algorithms on Graphs
- Weeks 19-20: Quick Tour of Combinatorial Optimization

Quizz on Week 20

Today’s Lecture

Aim:

- Introduce the more efficient sorting algorithms, based on the divide-and-conquer paradigm.
  - Merge sort
  - Quick sort
- Prove their correctness & time complexity

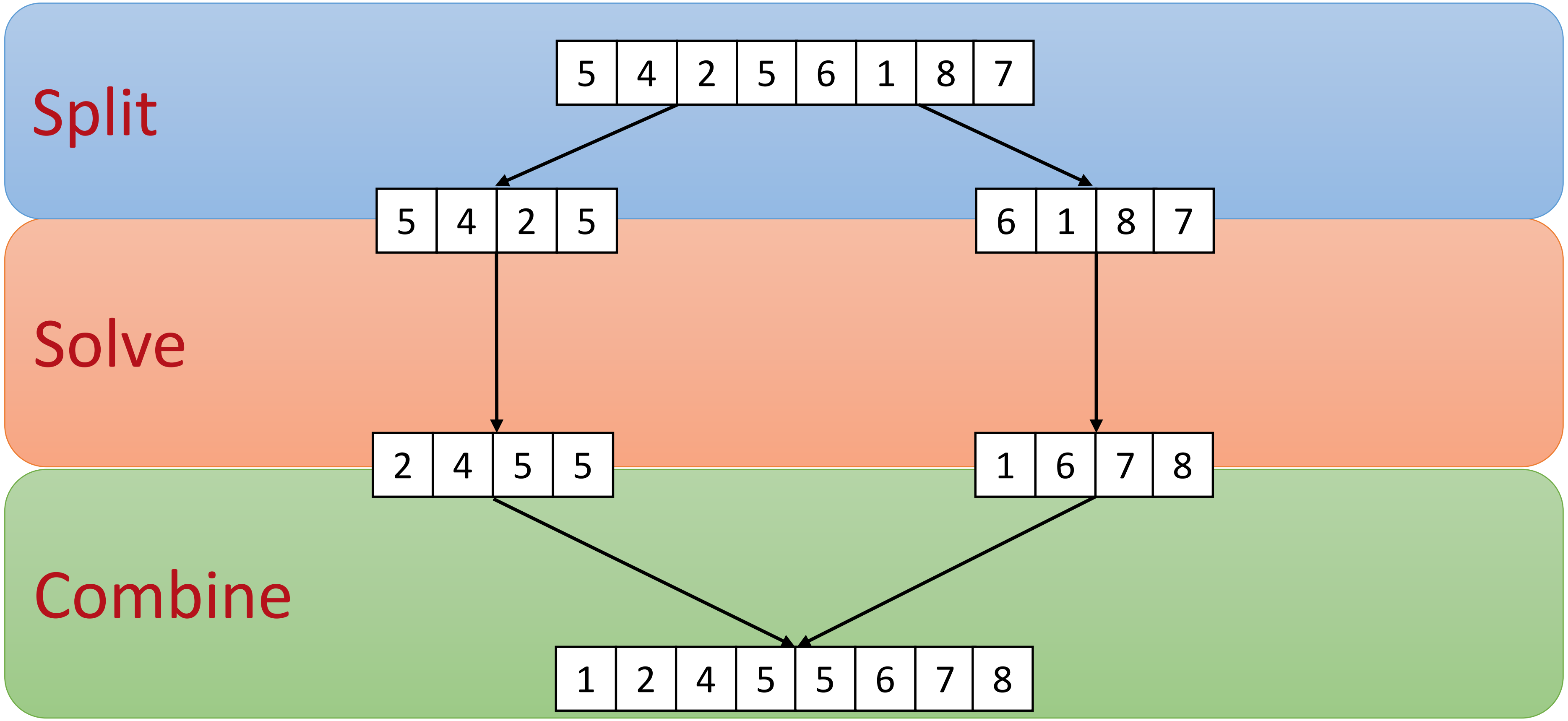
Merge Sort

- **Divide and Conquer Paradigm:**
  - The core idea is to decompose a problem into simpler sub-problems (of the same type).
  - Then, solve recursively on the sub-problems.
  - Finally, combine the solutions for the sub-problems to get a solution of the original problem.

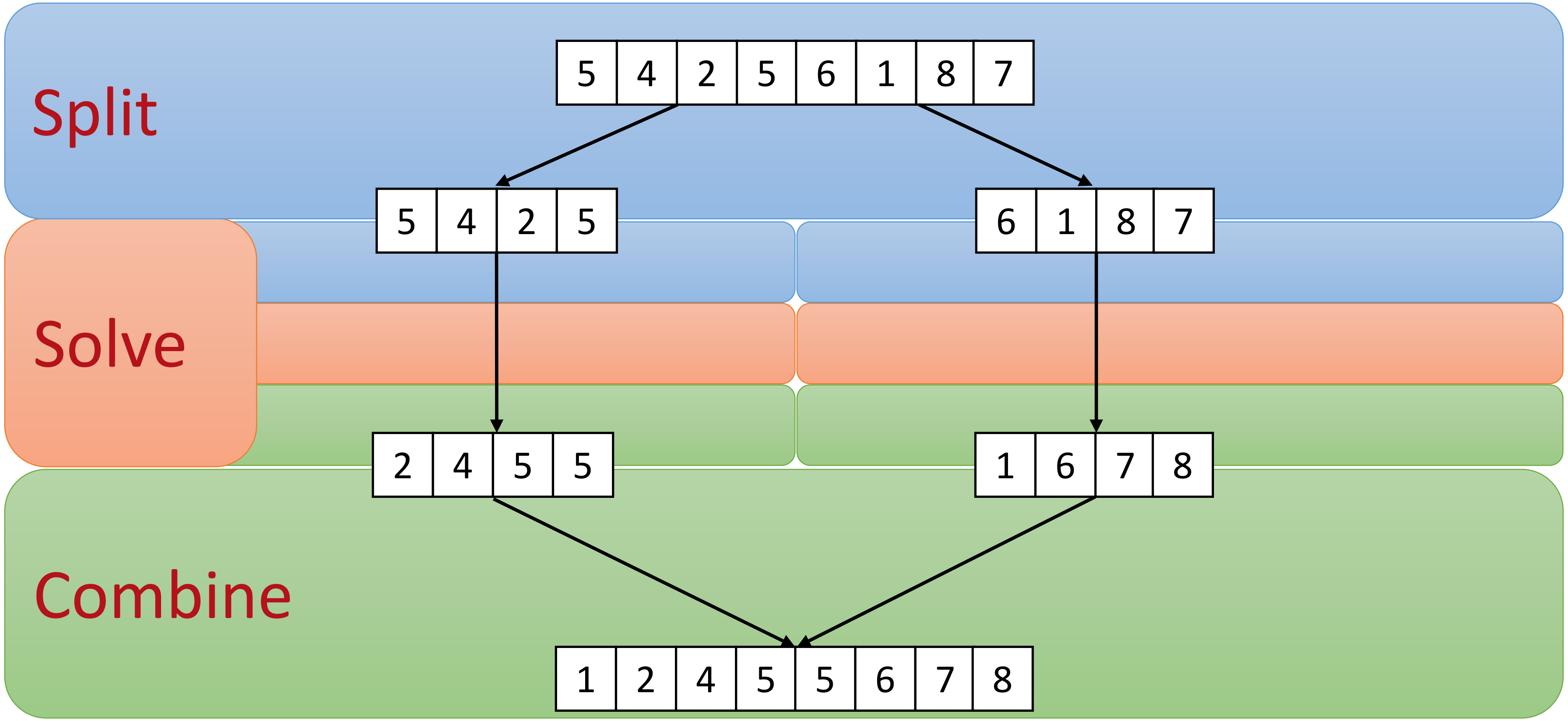
Merge Sort

- **Divide and Conquer in Merge Sort:**
  - <u>Split:</u> **Split** the input array into **two equal halves**,
  - <u>Solve:</u> **Recursively solve** (i.e., sort) the two subarrays independently,
  - <u>Combine:</u> **Combine** the two sorted subarrays **by merging** to get an overall sorted array.

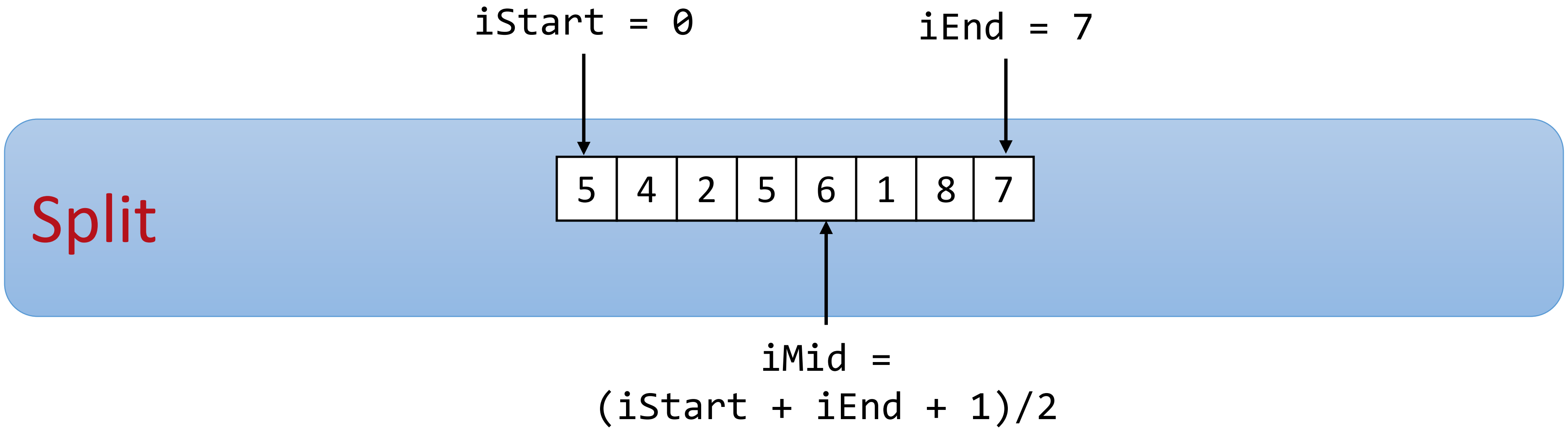
Merge Sort: Split, Solve, Combine



Merge Sort: Split, Solve, Combine



The Split Step



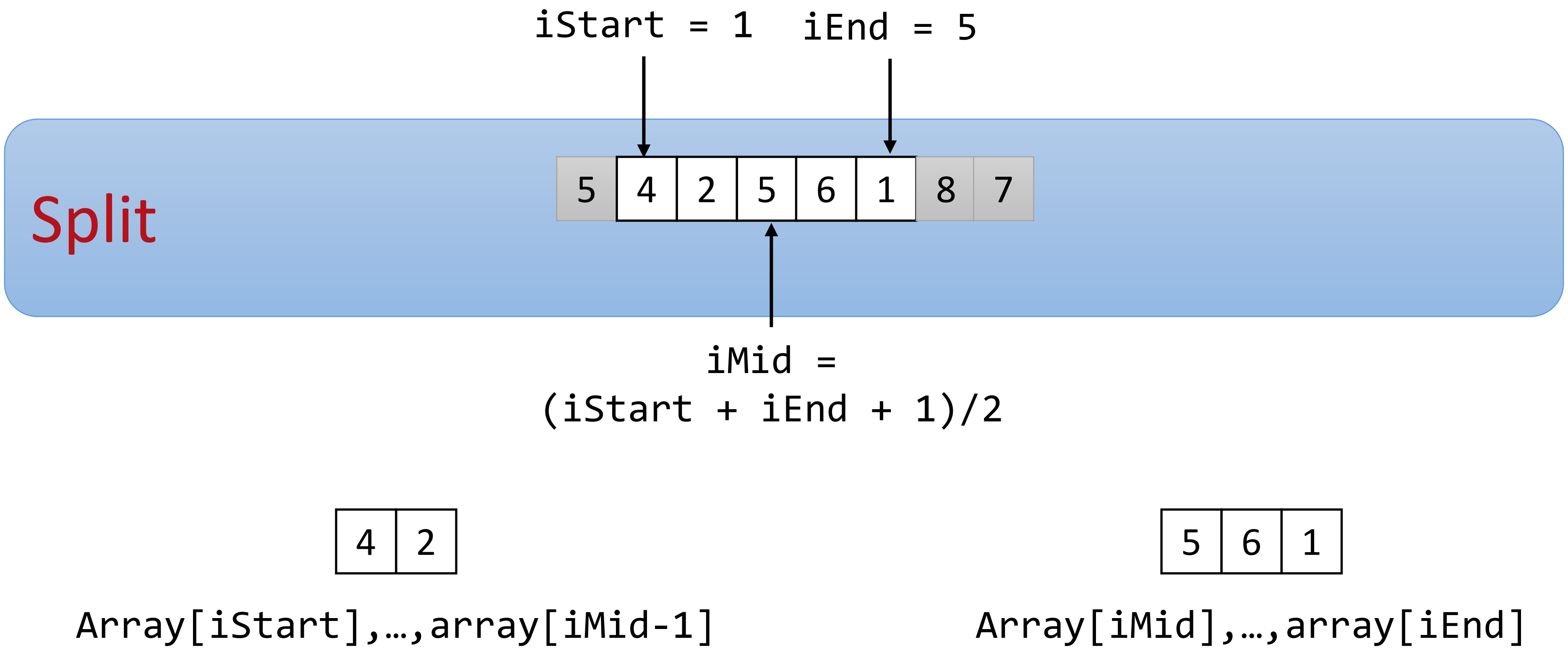
5	4	2	5

`Array[iStart],…,array[iMid-1]`

6	1	8	7

`Array[iMid],…,array[iEnd]`

The Split Step



The Split Step

iEnd = 3

iStart = 0

Split

5	4	2	5	6	1	8	7

iMid =
(iStart + iEnd + 1)/2

5	4

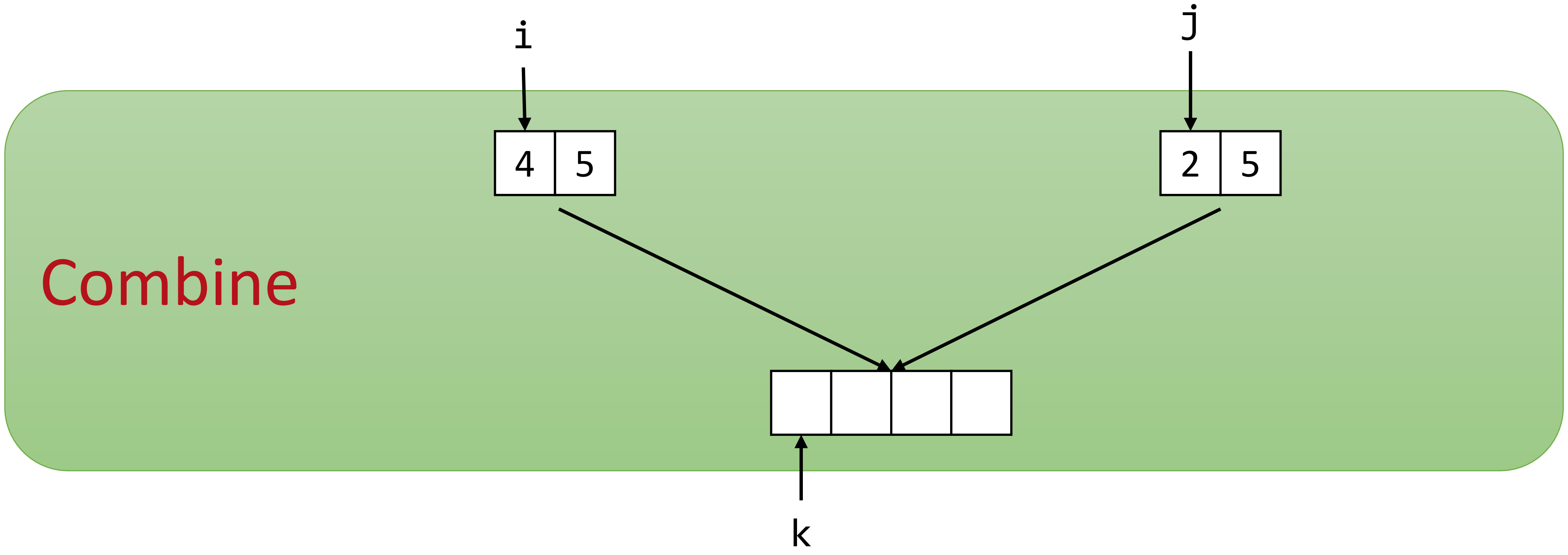
Array[iStart],…,array[iMid-1]

2	5

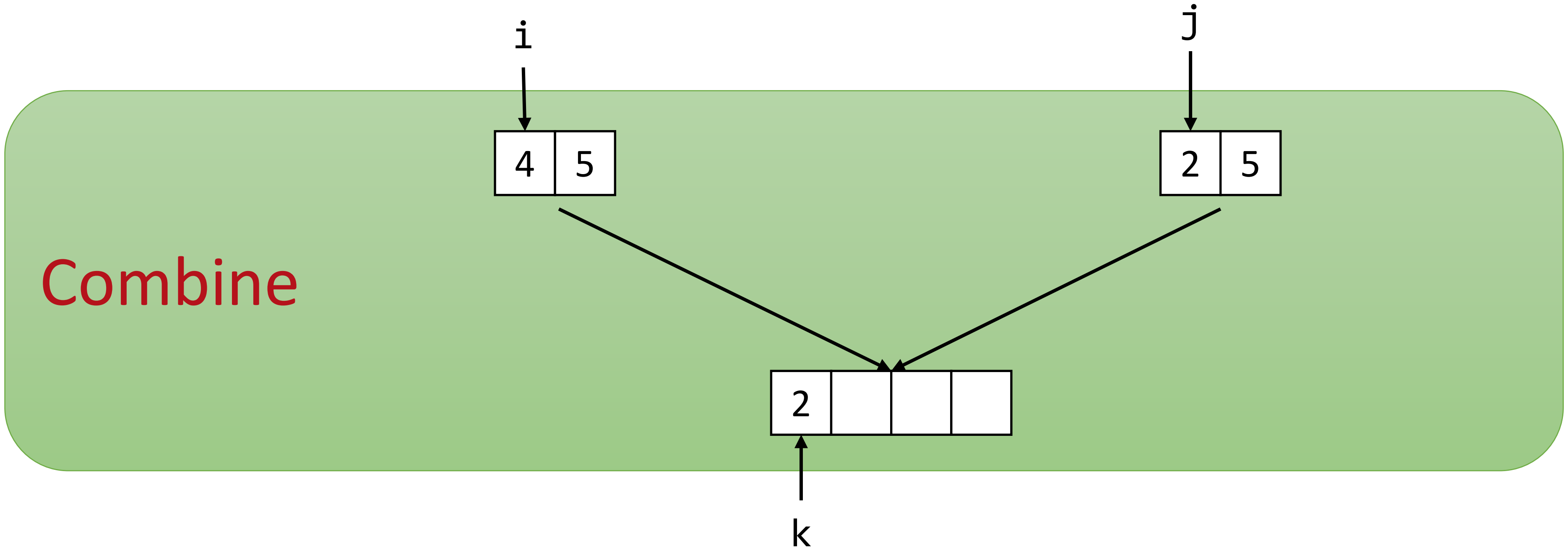
Array[iMid],…,array[iEnd]

The Solve Step

The Combination Step

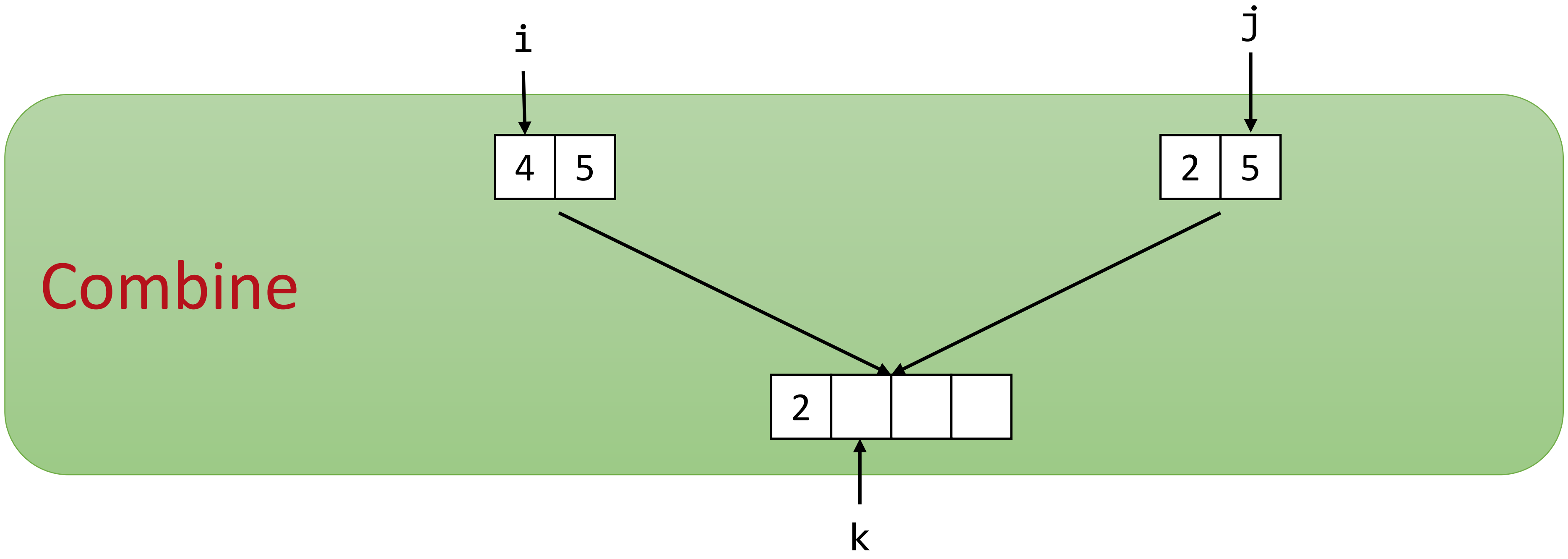


The Combination Step



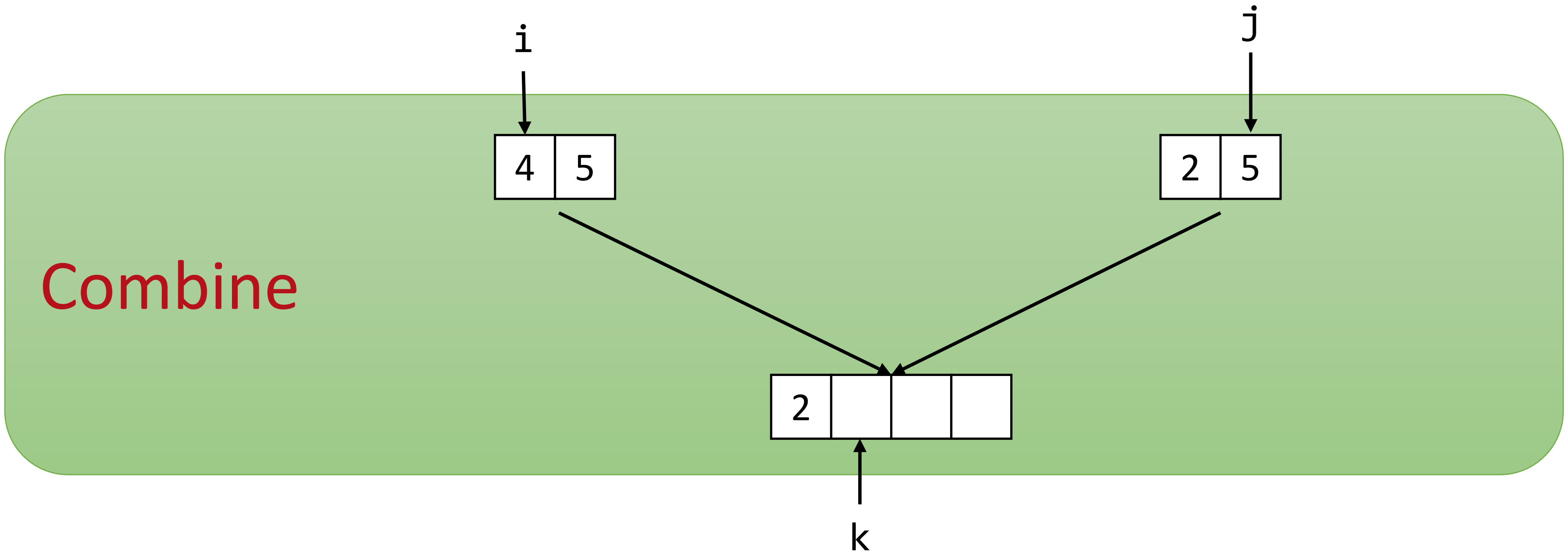

```
array[i] > array[j]
thus array[k] = array[j]
```

The Combination Step



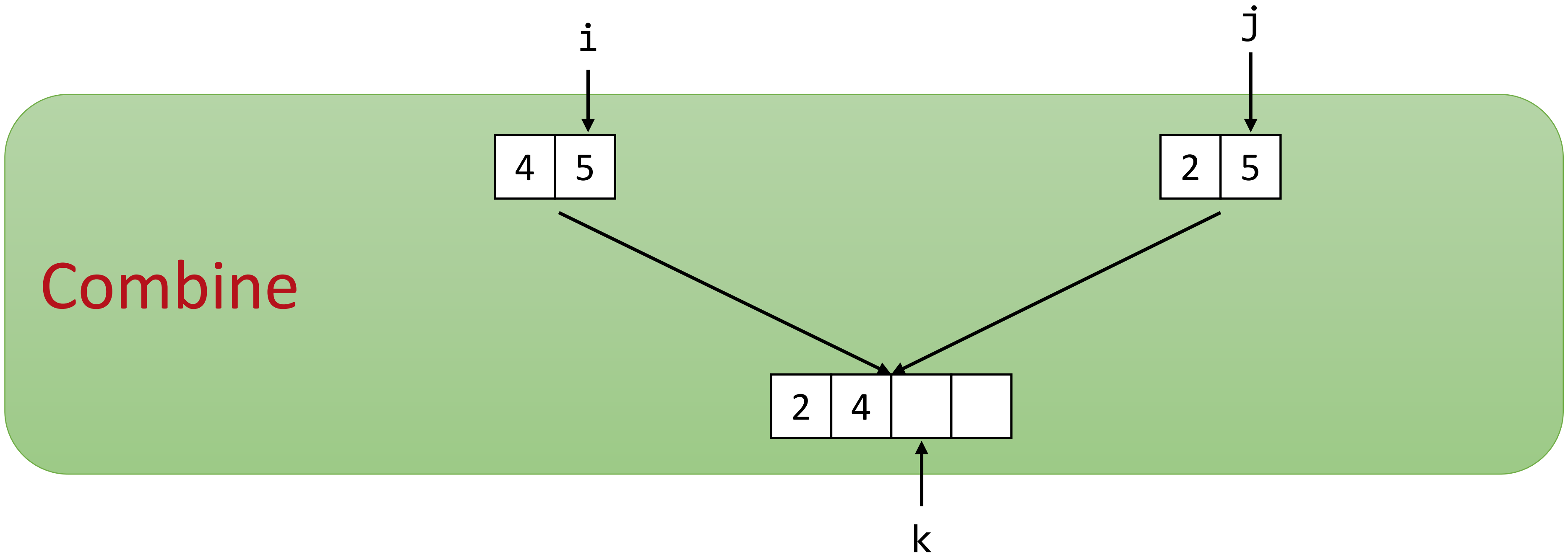

```
Increment j and k
```

The Combination Step



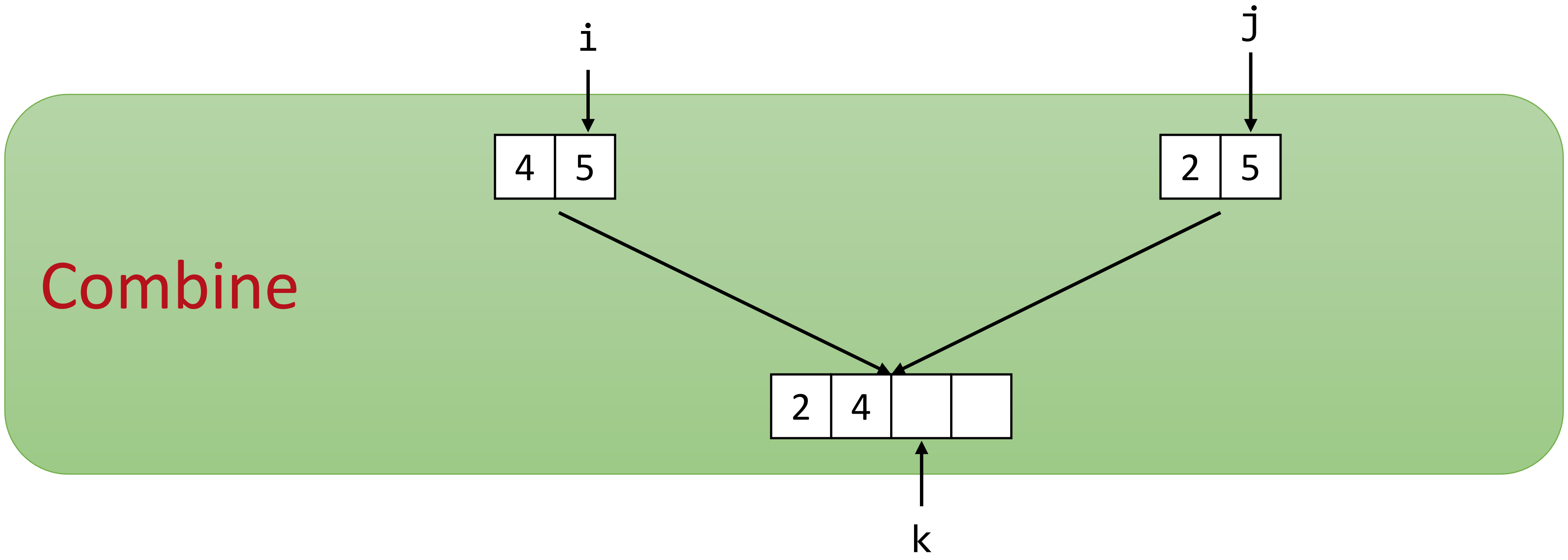

```
array[i] < array[j]
thus array[k] = array[i]
```

The Combination Step



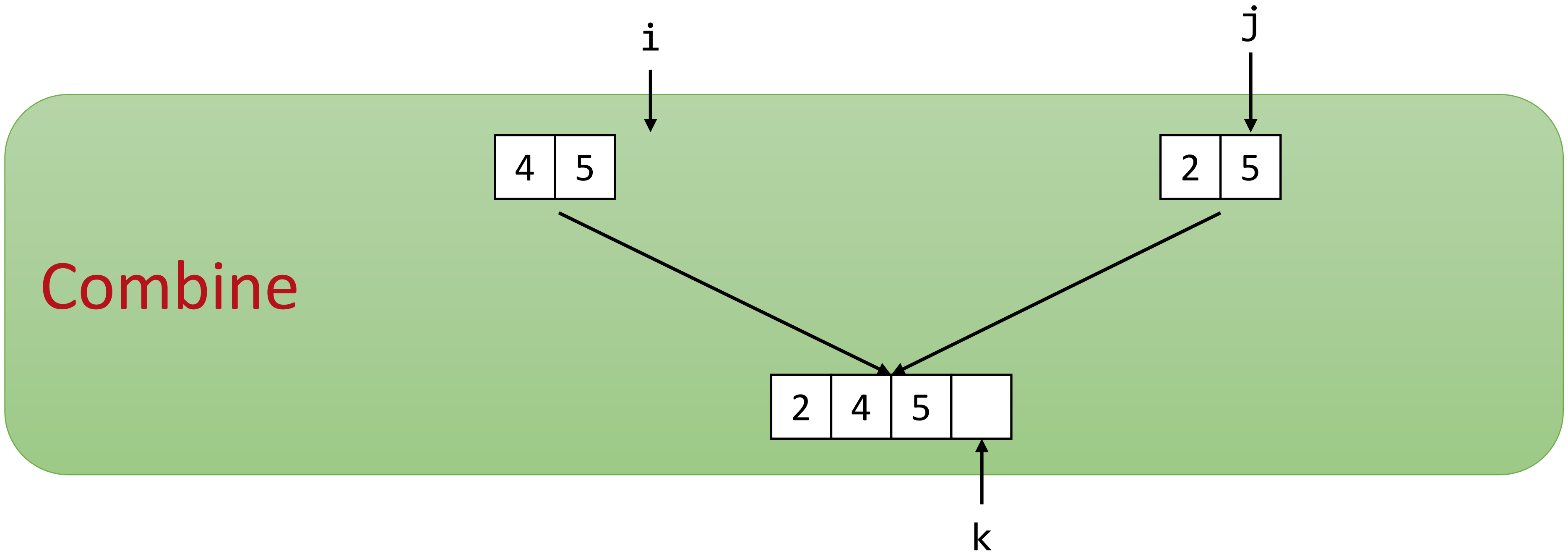
Increment i and k

The Combination Step



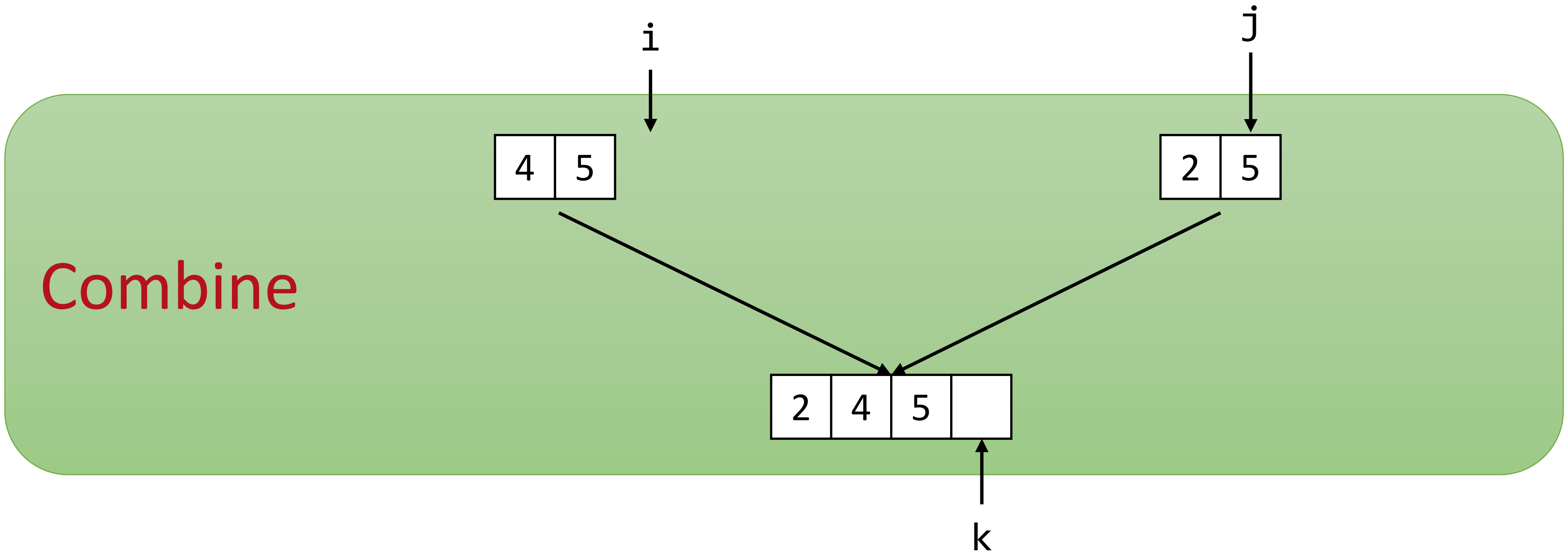

```
array[i] <= array[j]
thus array[k] = array[i]
```

The Combination Step



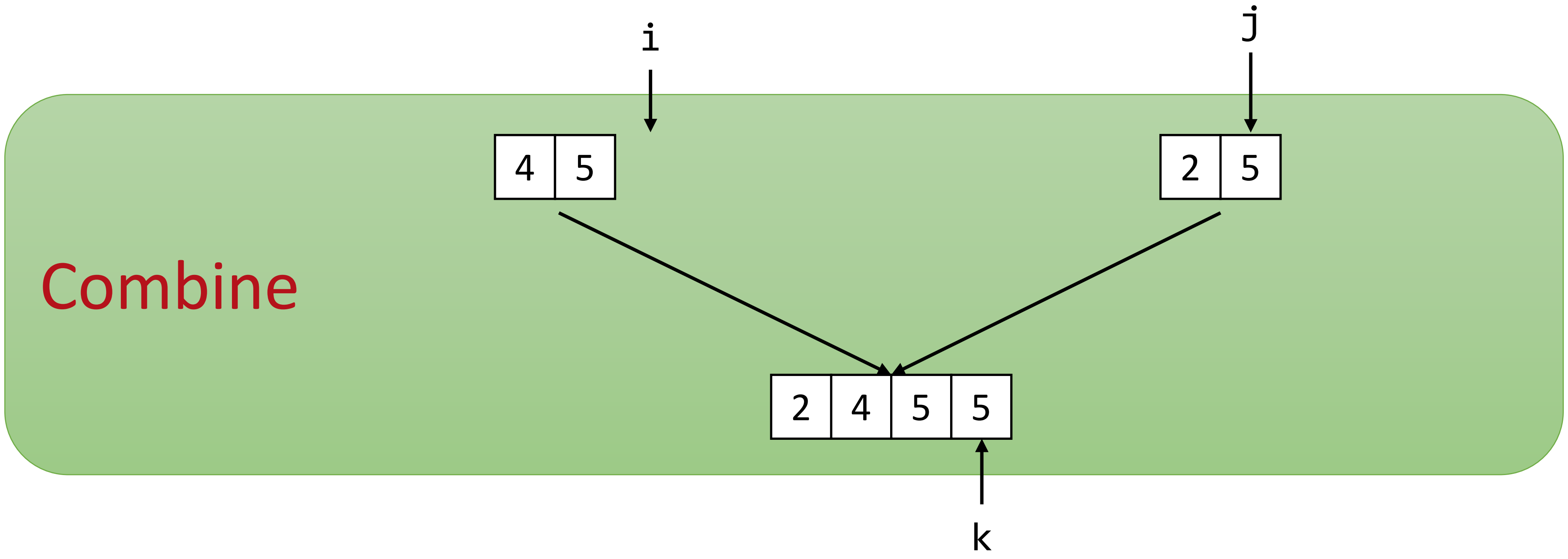

```
Increment i and k
```

The Combination Step




```
First half is done
thus array[k] = array[j]
```

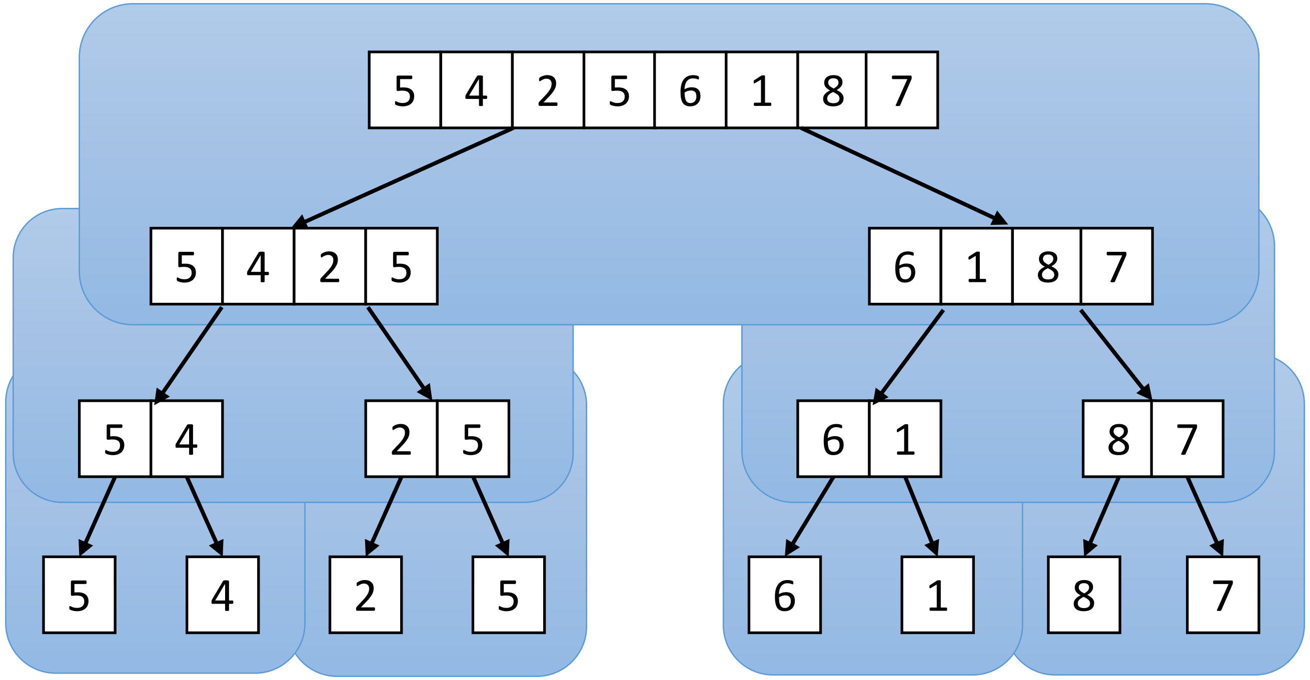
The Combination Step



Last iteration since k is now the last index of merged array

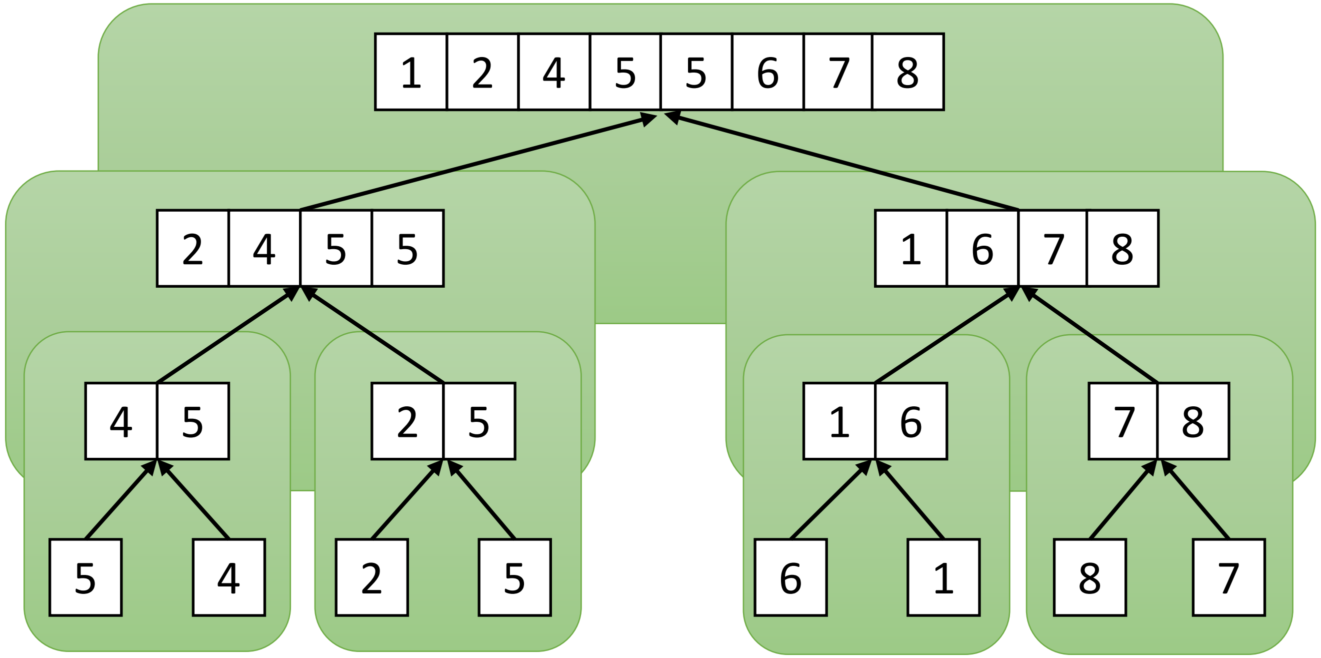
Merge Sort: Top-down Splitting

- Let's put these steps together. First we split the input array.



Merge Sort: Bottom-up Merging

- Then we merge the sorted arrays, starting with the (sorted) sub-arrays of size 1



Merge Sort: Code

Sort function is not itself recursive

```
class MergeSort {
    static void mergeSort(int array[]){
        int[] tmp = array.clone();
        recursiveMergeSort(tmp, 0, array.length-1,
array);
    }

    static void recursiveMergeSort(int B[], int
iStart, int iEnd, int A[]){
        if (iEnd == iStart) {
            return;
        }
        int iMid = (iStart + iEnd + 1)/2;
        recursiveMergeSort(A, iStart, iMid-1, B);
        recursiveMergeSort(A, iMid, iEnd, B);
        merge(B, iStart, iMid, iEnd, A);
    }
```

```
    static void merge(int A[], int iStart, int iMid, int
iEnd, int B[]){
        int i = iStart;
        int j = iMid;
        for (int k = iStart; k <= iEnd; k++){
            if(i < iMid && (j > iEnd || A[i] <= A[j])) {
                B[k] = A[i];
                i++;
            }
            else{
                B[k] = A[j];
                j++;
            }
        }
    }
}
```

Recursive sort sub-function

Merge Sort: Code

Sort function is not in-place, uses $O(n)$ additional memory

```
class MergeSort {
    static void mergeSort(int array[]){
        int[] tmp = array.clone();
        recursiveMergeSort(tmp, 0, array.length-1, 
array);
    }

    static void recursiveMergeSort(int B[], int 
iStart, int iEnd, int A[]){
        if (iEnd == iStart) {
            return;
        }
        int iMid = (iStart + iEnd + 1)/2;
        recursiveMergeSort(A, iStart, iMid-1, B);
        recursiveMergeSort(A, iMid, iEnd, B);
        merge(B, iStart, iMid, iEnd, A);
    }
```

```
    static void merge(int A[], int iStart, int iMid, int 
iEnd, int B[]){
        int i = iStart;
        int j = iMid;
        for (int k = iStart; k <= iEnd; k++){
            if(i < iMid && (j > iEnd || A[i] <= A[j])) {
                B[k] = A[i];
                i++;
            }
            else{
                B[k] = A[j];
                j++;
            }
        } } }
```

Recursive calls switch between the A and B input arrays, or in our case, between tmp and array.

Recursive sort sub-function is in-place

Merge Sort: Correctness

Correctness(n): Merge Sort is correct for inputs of size $\leq n$

Disclaimer:
Correctness proof is for the easier merge sort, that creates temporary arrays in each split step!

How to prove correctness of merge sort?

- Prove by (strong) induction that Correctness(n) is true for all $n \geq 1$.
- Base step (for $n = 1$):
  Correctness(1) is clearly true, as an array with one entry is already sorted.
- Induction step (for $n > 1$):
  - First, assume that the induction hypothesis holds for all integers $n' \geq 1$, $n' < n$.
  - Recall that Merge Sort splits an input array of size n into two smaller arrays of size $\frac{n}{2} < n$, and then recursively runs Merge Sort on each smaller array.
  - The induction hypothesis holds for $n/2$ so the recursive calls return sorted arrays.
  - Finally, the merging function successfully combines the two smaller, sorted arrays into the sorted version of the input array (of size n).

Merge Sort: Worst-case Time Complexity

Disclaimer:
Worst-case time complexity proof is (also) for the easier merge sort, that creates temporary arrays in each split step!

<u>Cost of Operations:</u>

- Merging two arrays, of size k each, takes $c_1 k$ elementary operations,
- Splitting an array of size k takes $c_2 k$ elementary operations,

<u>Time Complexity:</u>

- $T_M(n)$ is time complexity of Merge Sort when run on an input of size n.

$$T_M(n) \leq 2T_M\left(\frac{n}{2}\right) + (c_1 + c_2) \cdot n$$

Merge Sort: Worst-case Time Complexity

Disclaimer:
Worst-case time complexity proof is (also) for the easier merge sort, that creates temporary arrays in each split step!

$$T_M(n) \leq c_2 n + 2T_M\left(\frac{n}{2}\right) + c_1 n$$

- <u>Justification:</u>
  - First, we split the array of size n,
  - Then, we run Merge Sort on two different arrays, each of size $n/2$,
  - Finally, we merge the two small ($n/2$-sized) arrays.

Merge Sort: Worst-case Time Complexity

Disclaimer:
Worst-case time complexity proof is (also) for the easier merge sort, that creates temporary arrays in each split step!

$$T_M(n) \leq 2T_M\left(\frac{n}{2}\right) + (c_1 + c_2) \cdot n$$

- By using the **Master Theorem,**
- Or **solving equation** $T_M(n) = 2T_M\left(\frac{n}{2}\right) + (c_1 + c_2) \cdot n$:

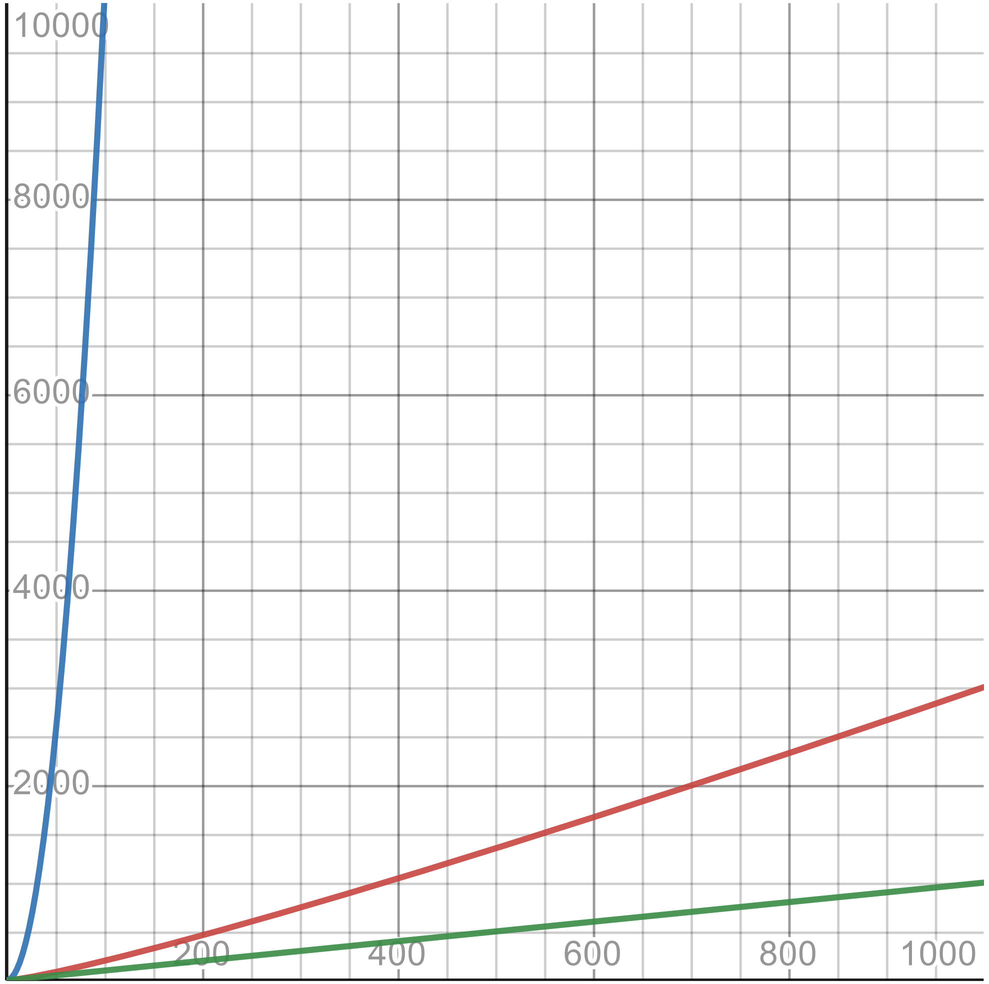
$$T_M(n) = O(n \log n)$$

Merge Sort: Summary

	Selection Sort	Insertion Sort	Merge Sort
Best case	$\mathrm{O}(\mathrm{n}^2)$	$\mathrm{O}(n)$	$\mathrm{O}(\mathrm{n} \log \mathrm{n})$
Average case	$\mathrm{O}(\mathrm{n}^2)$	$\mathrm{O}(\mathrm{n}^2)$	$\mathrm{O}(\mathrm{n} \log \mathrm{n})$
Worst case	$\mathrm{O}(\mathrm{n}^2)$	$\mathrm{O}(\mathrm{n}^2)$	$\mathrm{O}(\mathrm{n} \log \mathrm{n})$
In-place	Yes	Yes	No

Implied by the worst-case time complexity upper bound

Selection Sort

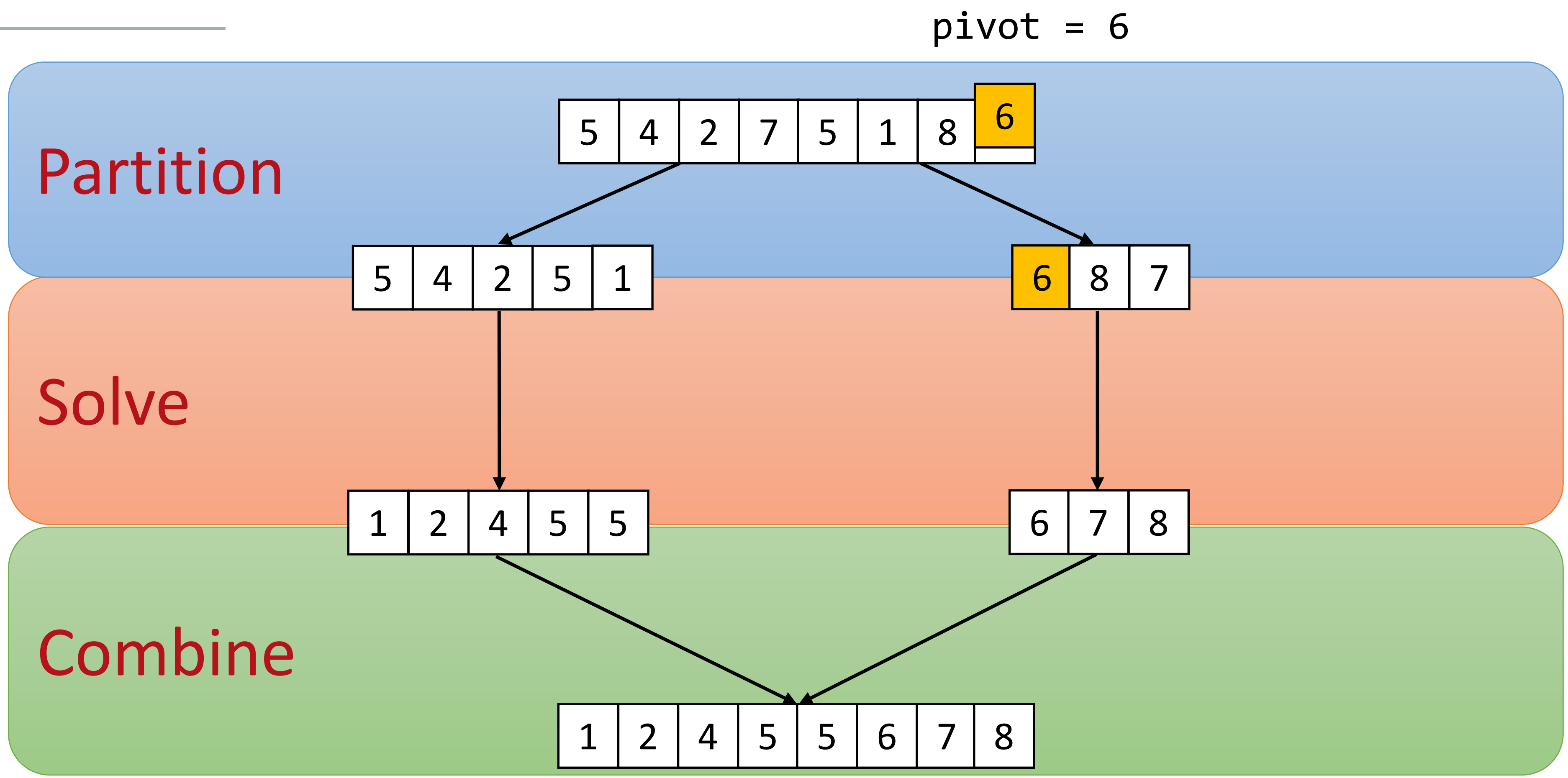


Insertion Sort	Merge Sort	
$O(n)$	$O(n \log n)$	Best case
$O(n^2)$	$O(n \log n)$	Average case
$O(n^2)$	$O(n \log n)$	Worst case
Yes	No	In-place

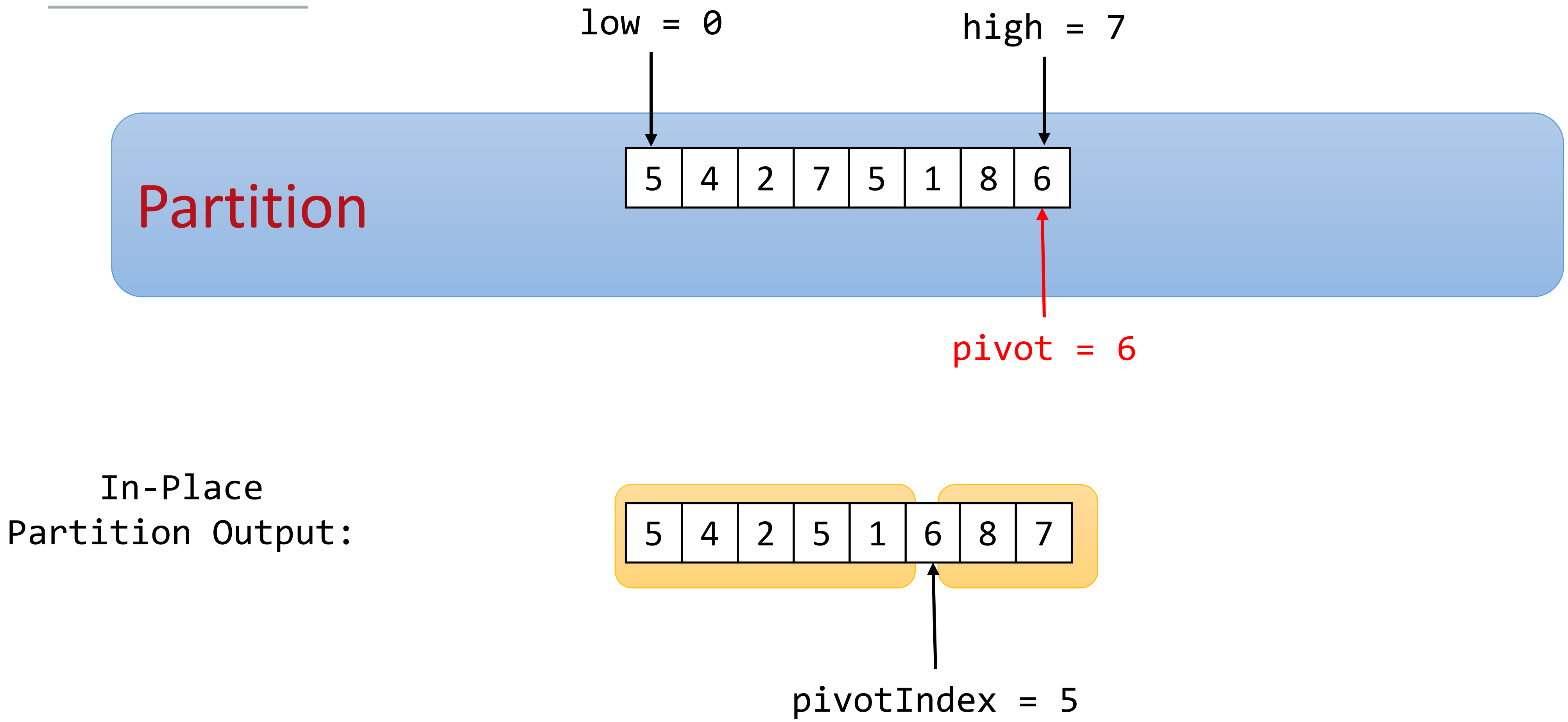
Quick Sort

- **Divide and Conquer in Quick Sort:**
  - Partition: **Partition** the input array **according to a pivot.**
    - One sub-array contains all elements **smaller** than the pivot,
    - The other contains all elements **greater** than the pivot.
  - Solve: **Recursively solve** (i.e., sort) the two subarrays independently,
  - Combine: **Combine** the two sorted subarrays **by (simple) merging** to get an overall sorted array.

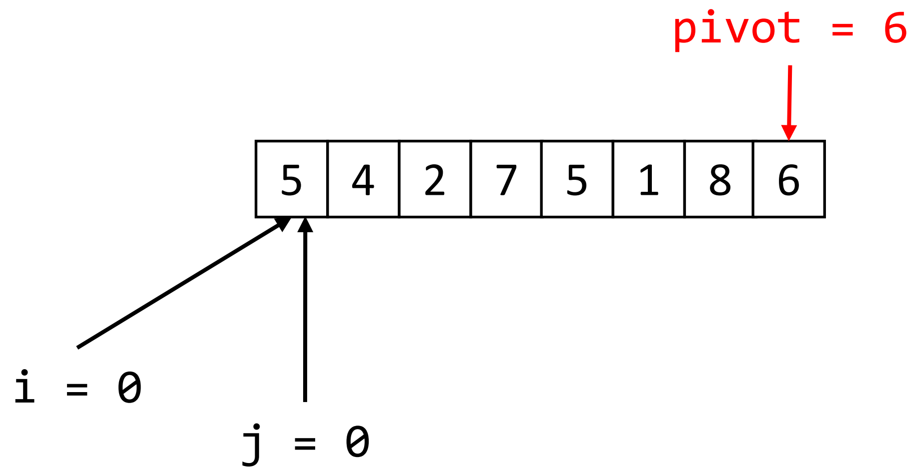
Quick Sort: Split, Solve, Combine



The (In-Place) Partition Step

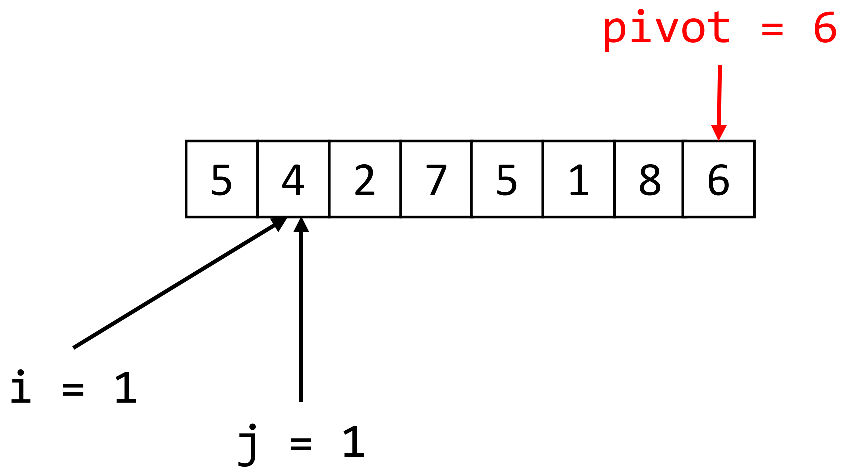


The (In-Place) Partition Step



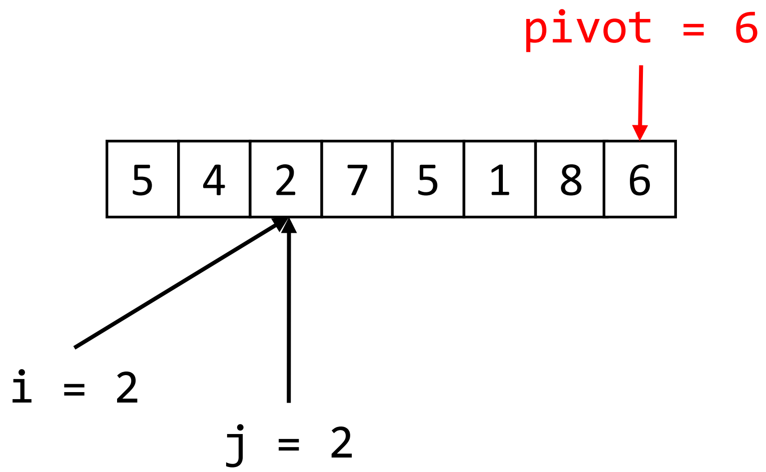

```
     Array[j] < pivot:
Swap array[i] and array[j]
     Increment i and j
```

The (In-Place) Partition Step



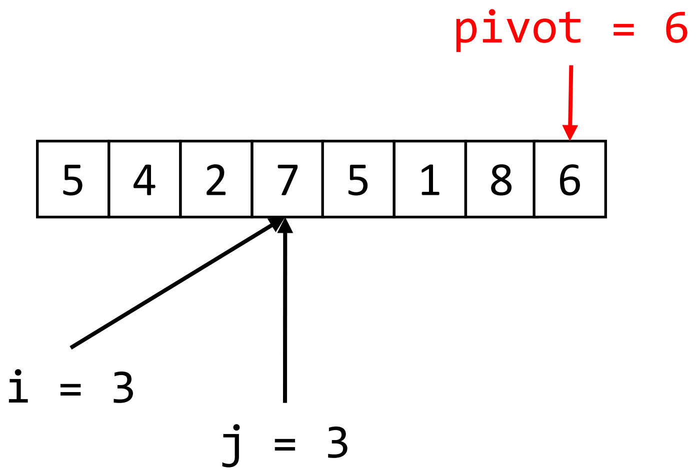

```
     Array[j] < pivot:
Swap array[i] and array[j]
     Increment i and j
```

The (In-Place) Partition Step



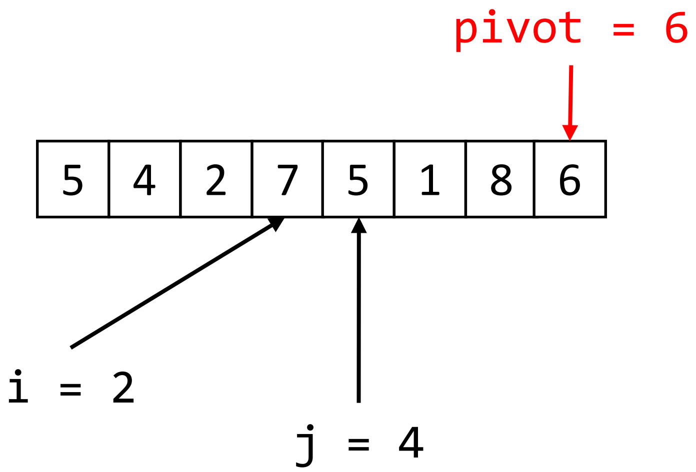

```
     Array[j] < pivot:
Swap array[i] and array[j]
     Increment i and j
```

The (In-Place) Partition Step



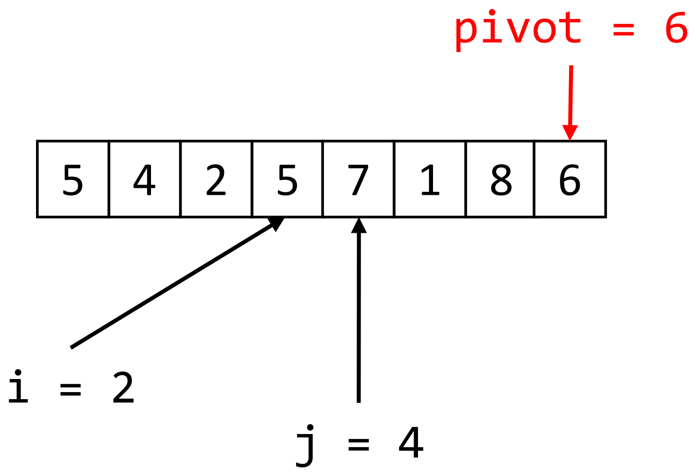

```
Array[j] >= pivot:
    No swapping
    Increment j
```

The (In-Place) Partition Step



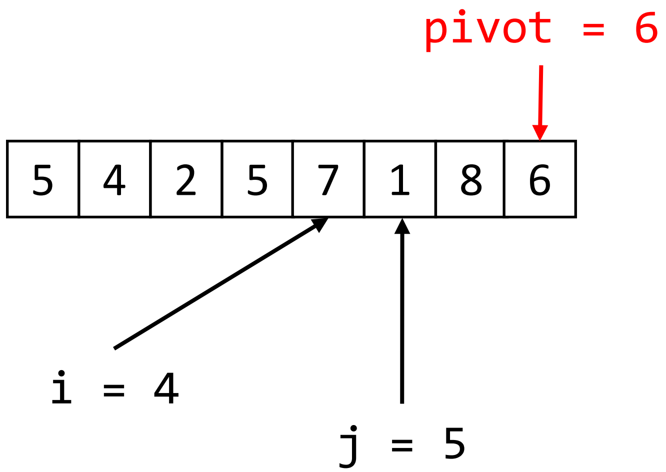

```
     Array[j] < pivot:
Swap array[i] and array[j]
     Increment i and j
```

The (In-Place) Partition Step



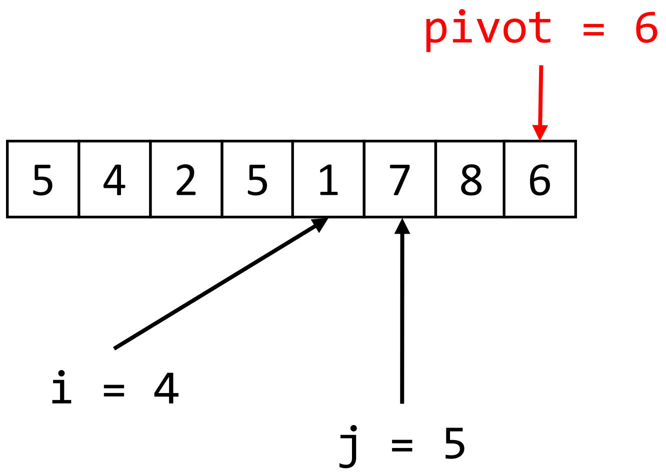

```
    Array[j] < pivot:
Swap array[i] and array[j]
    Increment i and j
```

The (In-Place) Partition Step



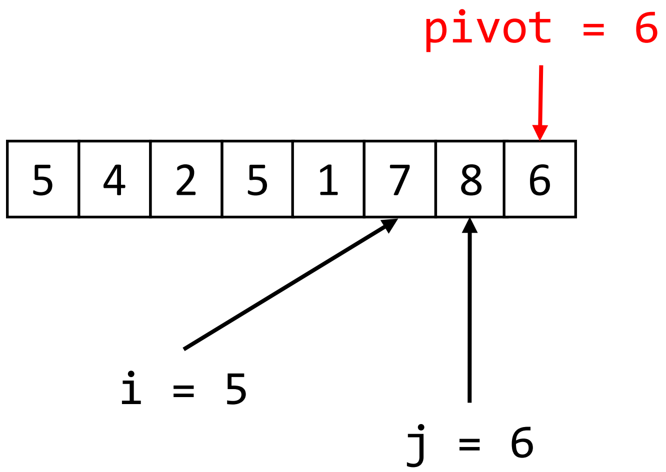

```
    Array[j] < pivot:
Swap array[i] and array[j]
    Increment i and j
```

The (In-Place) Partition Step



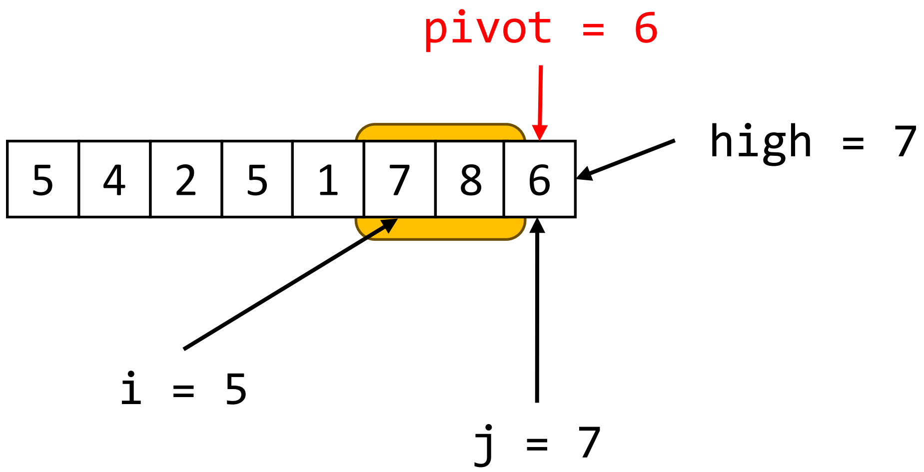

```
     Array[j] < pivot:
Swap array[i] and array[j]
     Increment i and j
```

The (In-Place) Partition Step




```
Array[j] >= pivot:
    No swapping
    Increment j
```

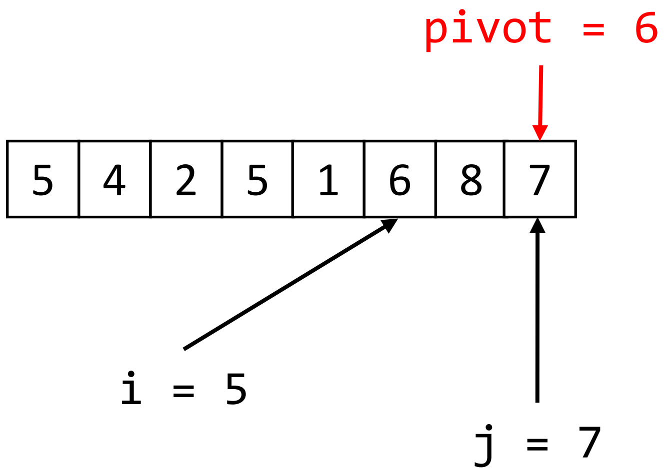
The (In-Place) Partition Step



Here, high = 7
is index of last
entry

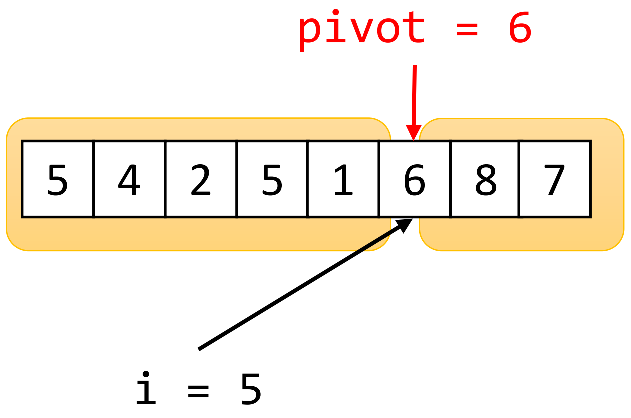
j = high:
Swap array[high] and array[i]

The (In-Place) Partition Step



j = high:
Swap array[high] and array[i]

The (In-Place) Partition Step



Return i as the pivot index in the output (in-place) array

Quick Sort: Code

```java
class QuickSort {
    static void quickSort(int array[]){
        recursiveQuickSort(array, 0, array.length-1);
    }

    static void recursiveQuickSort(int array[], int
low, int high){
        if (low < high) {
            int p = partition(array,low,high);
            recursiveQuickSort(array, low, p-1);
            recursiveQuickSort(array, p+1, high);
        }
    }
```

```java
    static int partition(int array[], int low,
int high){
        int pivot = array[high];
        int i = low;
        for (int j = low; j < high ; j++){
            if (array[j] < pivot){
                int tmp = array[i];
                array[i] = array[j];
                array[j] = tmp;
                i++;
            }
        }
        int tmp = array[i];
        array[i] = array[high];
        array[high] = tmp;
        return i;
    }
}
```

Quick Sort: Correctness

```
class QuickSort {
    static void quickSort(int array[]){
        recursiveQuickSort(array, 0, array.length-1);
    }

    static void recursiveQuickSort(int array[], int 
low, int high){
        if (low < high) {
            int p = partition(array,low,high);
            recursiveQuickSort(array, low, p-1);
            recursiveQuickSort(array, p+1, high);
        }
    }
```

```
    static int partition(int array[], int low, 
int high){
        int pivot = array[high];
        int i = low;
        for (int j = low; j < high ; j++){
            if (array[j] < pivot){
                int tmp = array[i];
                array[i] = array[j];
                array[j] = tmp;
                i++;
            }
        }
        int tmp = array[i];
        array[i] = array[high];
        array[high] = tmp;
        return i;
    }
}
```

Correctness(n): Quicksort is correct for inputs of size $\leq n$

Quick Sort: Correctness

Correctness(n): Quick Sort is correct for inputs of size $\leq n$

How to prove correctness of quick sort?

- Prove by (strong) induction that Correctness(n) is true for all $n \geq 1$.
- Base step (for $n = 1$):
  Correctness(1) is clearly true, as an array with one entry is already sorted.
- Induction step (for $n > 1$):
  - First, assume that the induction hypothesis holds for all integers $n' \geq 1$, $n' < n$.
  - Recall that Quick Sort splits an input array of size n into two smaller arrays of size $\mathrm{n}_1, \mathrm{n}_2 < n$, and then recursively runs Quick Sort on each smaller array.
  - The induction hypothesis holds for n_1 and n_2, and thus these recursive calls return sorted arrays.
  - Finally, the two smaller, sorted arrays combine into the sorted version of the input array (of size n). This completes the induction step.

Quick Sort: Worst-case Time Complexity

```
class QuickSort {
    static void quickSort(int array[]){
        recursiveQuickSort(array, 0, array.length-1);
    }

    static void recursiveQuickSort(int array[], int
low, int high){
        if (low < high) {
            int p = partition(array,low,high);
            recursiveQuickSort(array, low, p-1);
            recursiveQuickSort(array, p+1, high);
        }
    }
```

Combining the two sorted arrays from the recursive calls takes c_2 elementary operations.

```
    static int partition(int array[], int low,
int high){
        int pivot = array[high];
        int i = low;
        for (int j = low; j < high ; j++){
            if (array[j] < pivot){
                int tmp = array[i];
                array[i] = array[j];
                array[j] = tmp;
                i++;
            }
        }
        int tmp = array[i];
        array[i] = array[high];
        array[high] = tmp;
        return i;
    }}
```

Partitioning an array of size k takes $c_1 k$ elementary operations.

Quick Sort: Worst-case Time Complexity

Cost of Operations:

- Partitioning an array of size k takes $c_1 k$ elementary operations.
- Combining two sorted arrays from the recursive calls takes c_2 elementary operations.

Time Complexity:

- $T_Q(n)$ is the time complexity of Quick Sort when run on an input of size n.
- Quicksort takes the **most time when the recursive calls are unbalanced:** one of the recursive calls is done on an array of size $n - 1$

$$T_Q(n) \leq T_Q(n-1) + T_Q(0) + c_1 \cdot n + c_2.$$

Quick Sort: Worst-case Time Complexity

$$T_Q(n) \leq T_Q(n-1) + T_Q(0) + c_1 \cdot n + c_2$$

$$T_Q(n) \leq T_Q(n-2) + 2T_Q(0) + c_1 \cdot (n + n - 1) + 2c_2$$

- Repeating this, we get: $T_Q(n) \leq n\, T_Q(0) + c_1 \cdot n^2 + c_2 \cdot n$
- From which you can show that: $T_Q(n) = O(n^2)$

Selection Sort

	Selection Sort	Insertion Sort	Merge Sort	Quick Sort
Best case	$O(n^2)$	$O(n)$	$O(n \log n)$	$O(n \log n)$
Average case	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Worst case	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
In-place	Yes	Yes	No	Yes

Summary

Today's lecture:

- Introduced two widely-used sorting algorithms, based on the popular divide-and-conquer paradigm.
  - Merge sort
  - Quick sort
- Proved their (correctness and) worst-case time complexity

- **Next Lecture:** Algorithms on trees.
- **Any questions?**