SAN FRANCISCO STATE UNIVERSITY
COMPUTER SCIENCE DEPARTMENT

SF STATE

SORTING CSC340
BUBBLE SORT, MERGE SORT, QUICK SORT

DUC TA

EXPERIENTIA DOCET

Bubble Sort


Bubble Sort

Bubble Sort

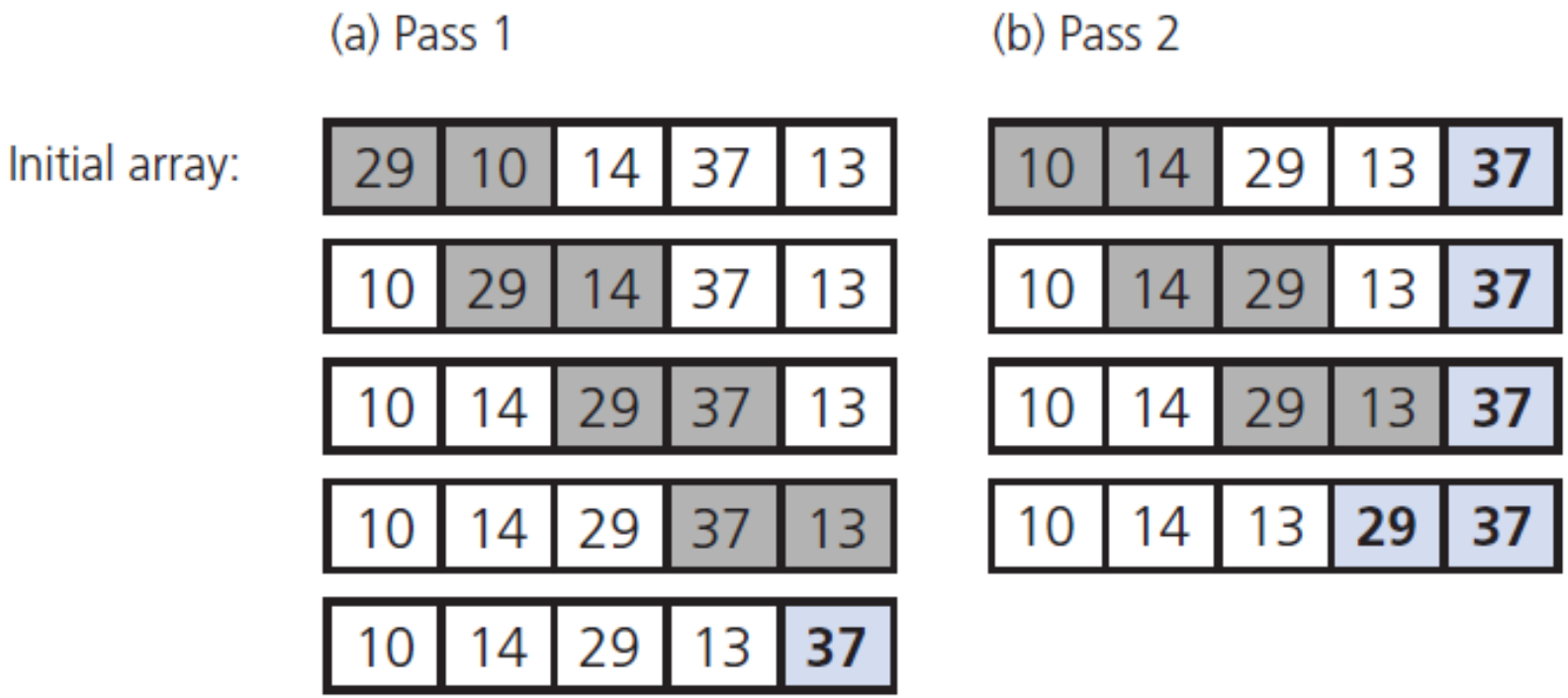
- Bubble Sort orders a list of values by repetitively **comparing neighboring elements** and **swapping their positions** if necessary.

- More specifically:

  1. Scan the list, **exchanging adjacent elements** if they are not in relative order; this bubbles the highest value to the top/right.
  2. Scan the list **again**, bubbling up the **second highest value.**
  3. **Repeat** until all elements have been placed in their proper order.

Highest neighbor → Final Position

Bubble Sort

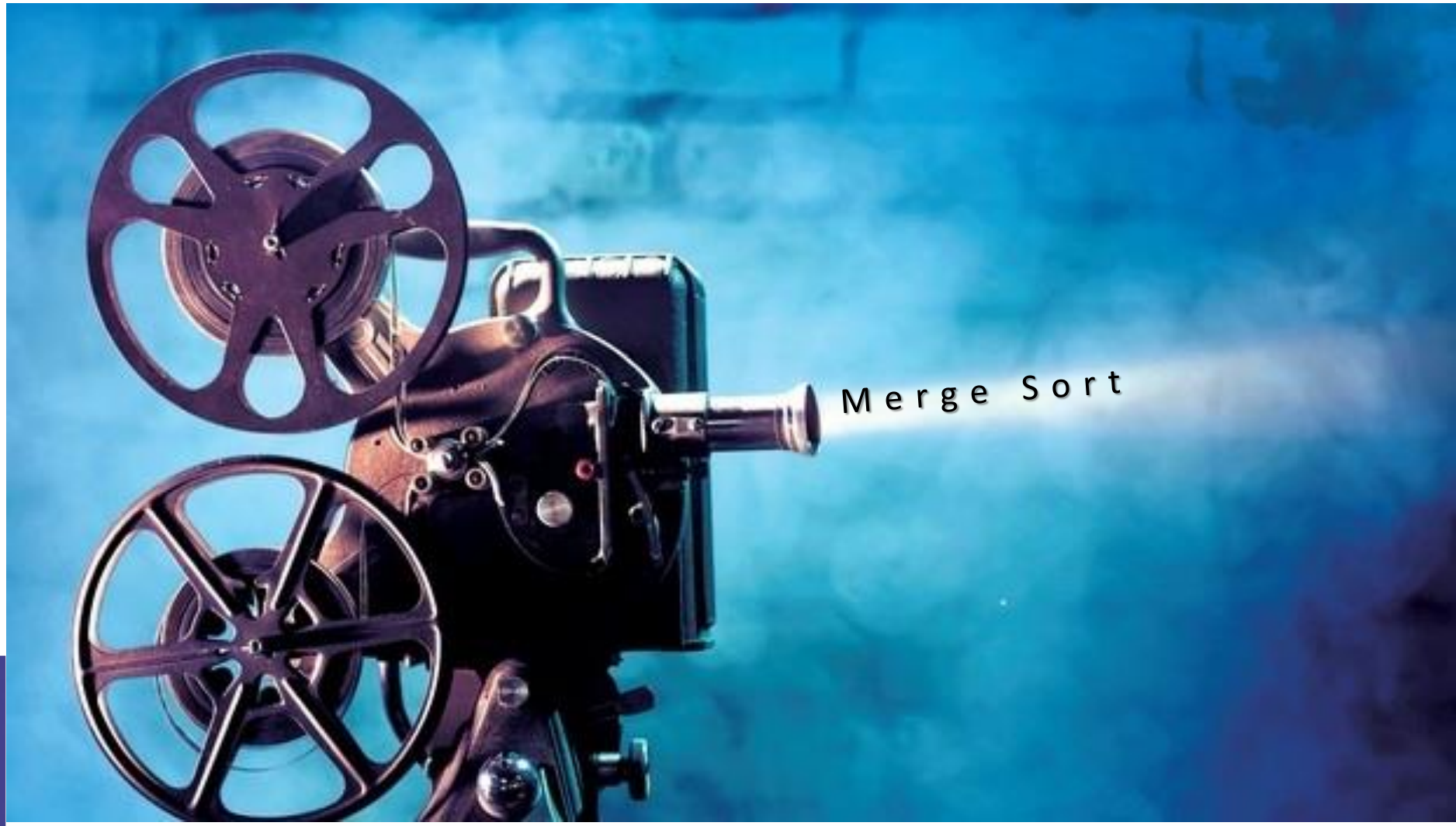


Bubble Sort

```
 7  template<class ItemType>
 8  void bubbleSort(ItemType theArray[], int n) {
 9      bool sorted = false;
10      int pass = 1;
11      while (!sorted && (pass < n)) {
12          sorted = true;
13          for (int index = 0; index < n - pass; index++) {
14              int nextIndex = index + 1;
15              if (theArray[index] > theArray[nextIndex]) {
16                  std::swap(theArray[index], theArray[nextIndex]);
17                  sorted = false;
18              }
19          }
20          pass++;
21      }
22  }
```

FASTER SORTING ALGORITHMS

MERGE SORT


Merge Sort

Merge Sort

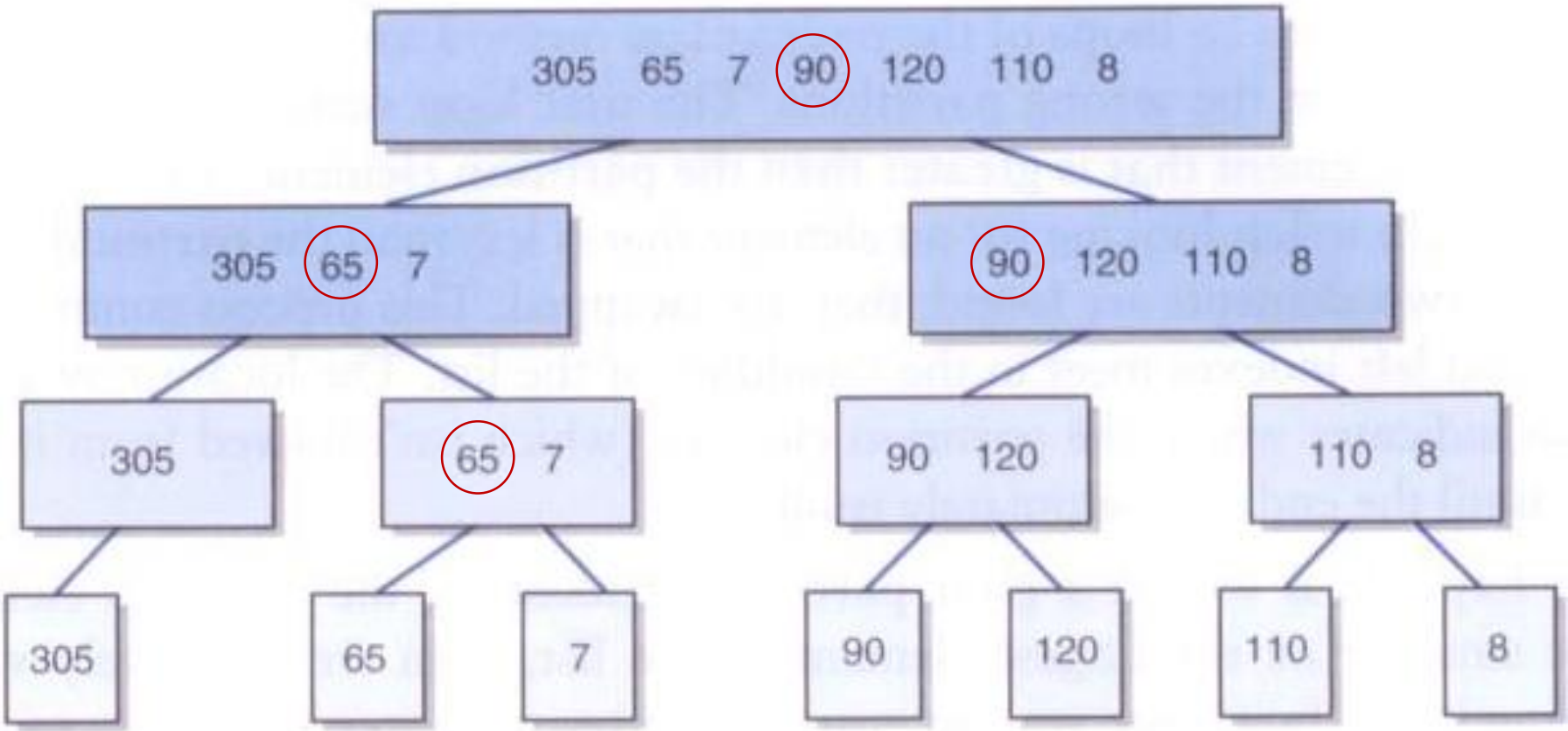
- Merge Sort orders values by **recursively dividing the list in half** until each sub-list has one element, then recombining

- More specifically:
  1. Divide the list into **two roughly equal parts**.
  2. Recursively divide **each part in half**, continuing until a part contains only **one element.**
  3. **Merge** the two parts into one sorted list.
  4. **Continue to merge** parts as the recursion unfolds

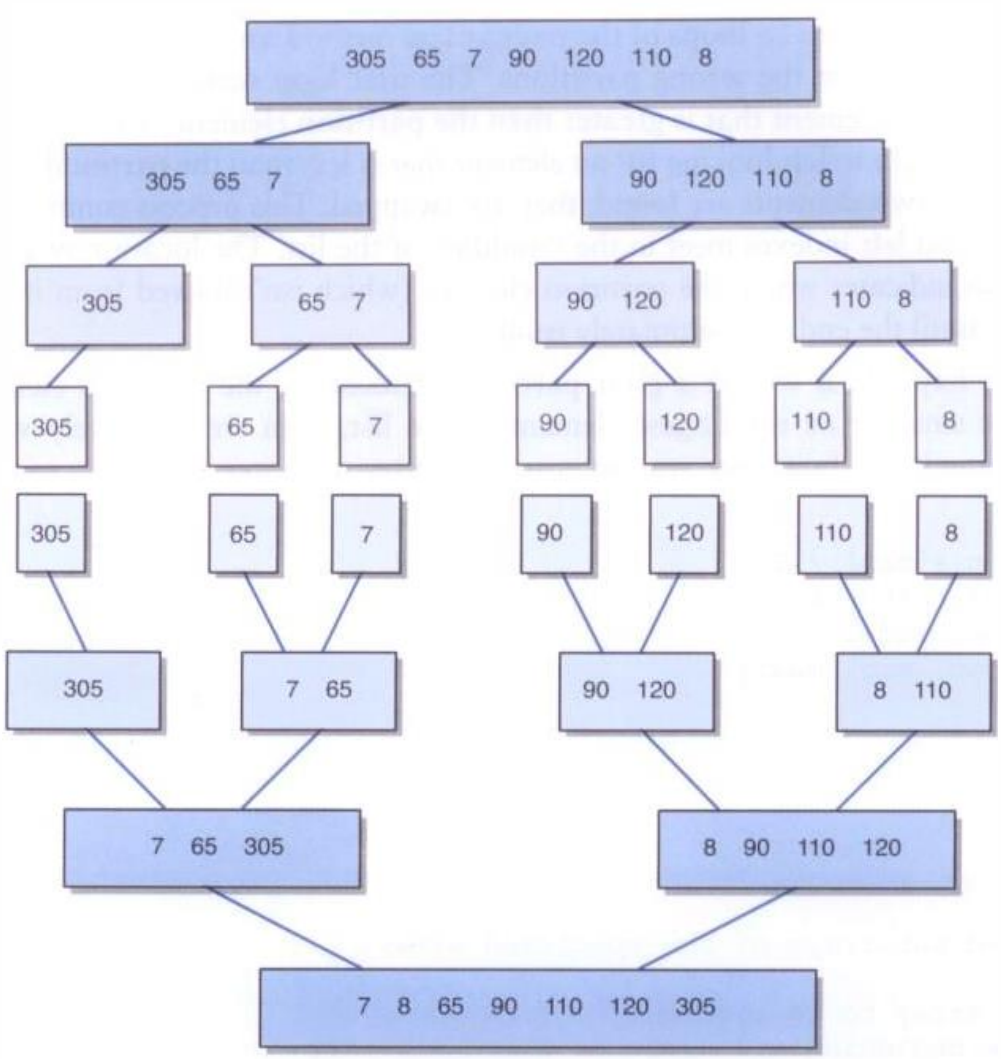
Recursive Dividing → Merge

(Divide and Conquer / Recursive)

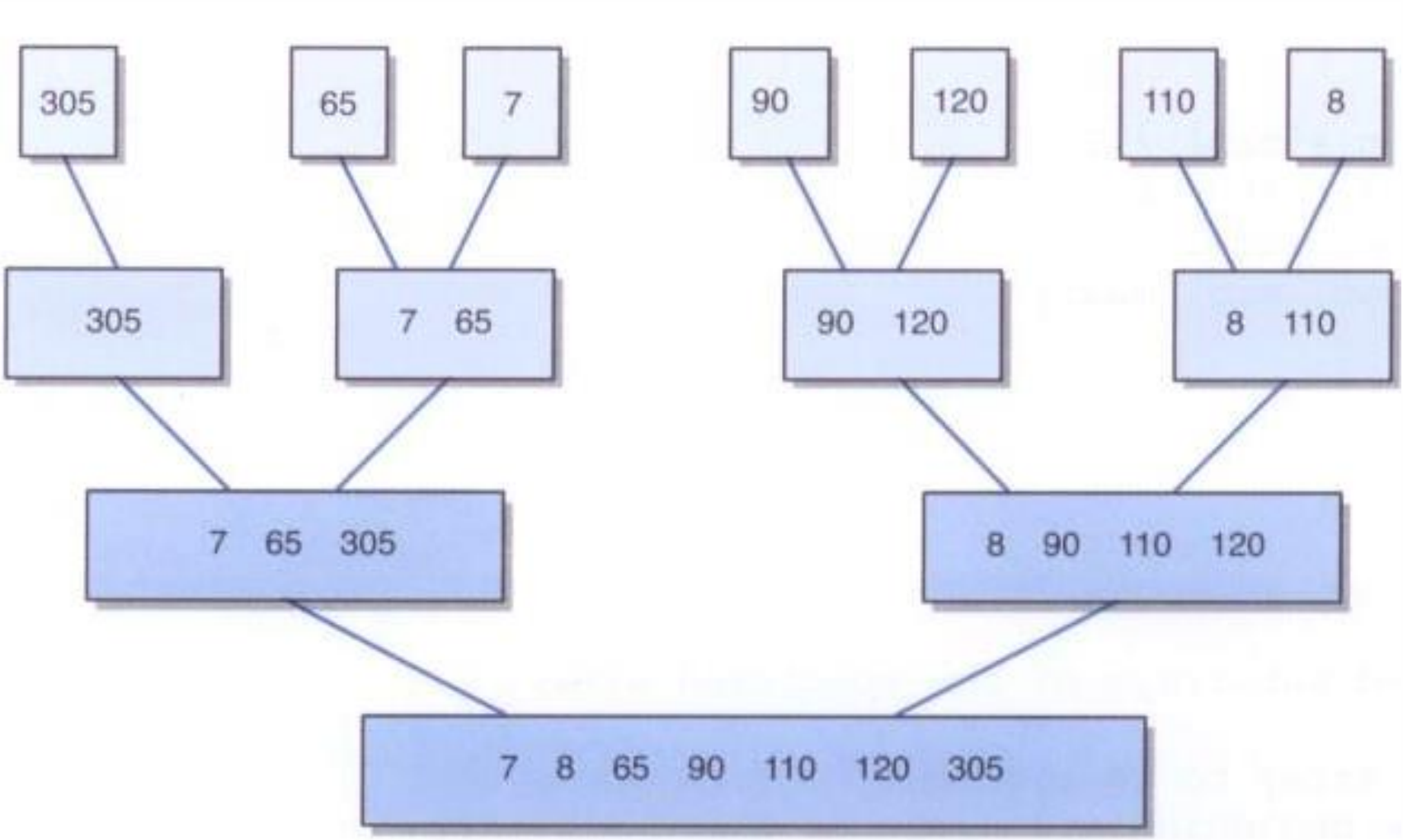
- Dividing lists in half repeatedly:



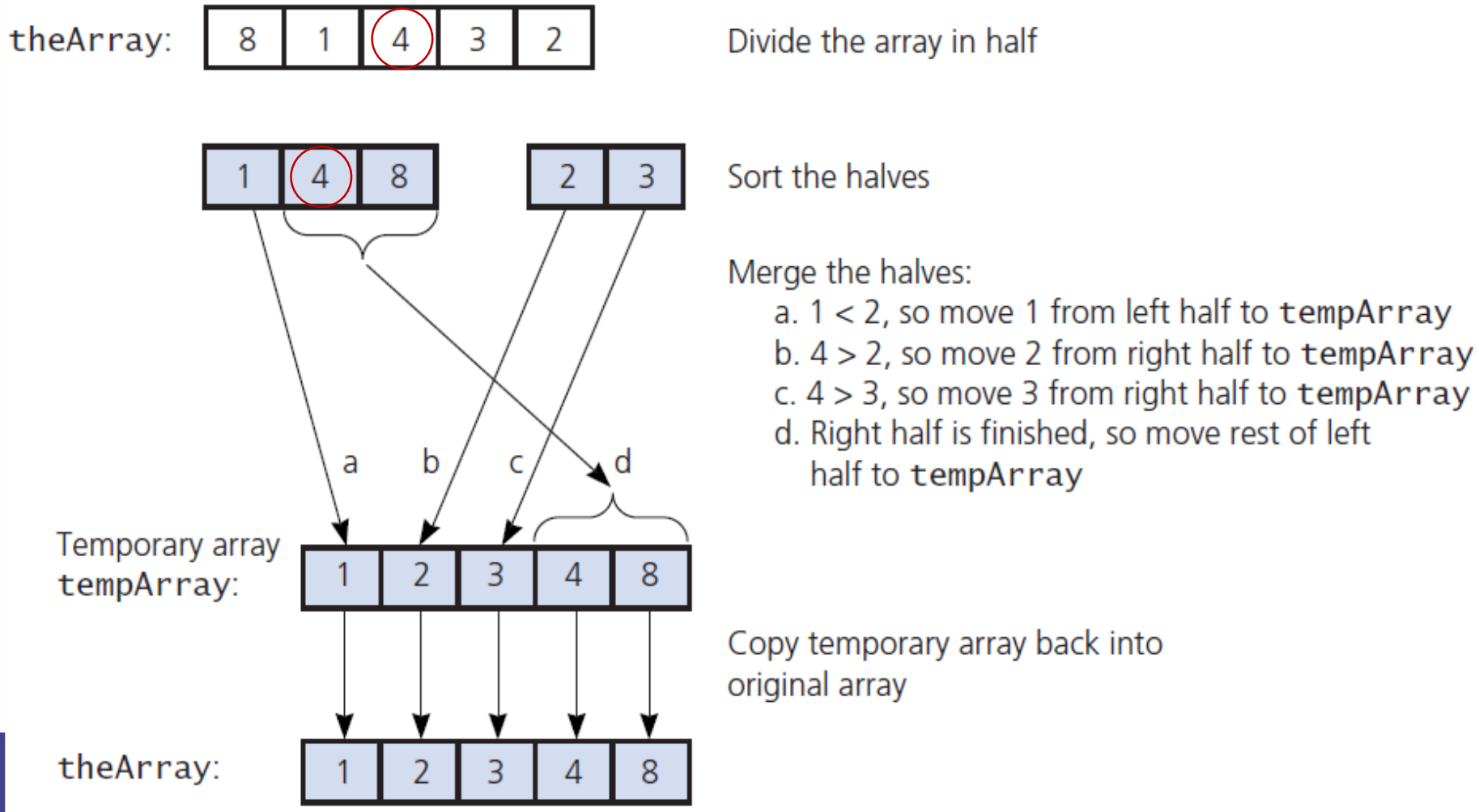
- → Merging sorted elements



- Merging sorted elements

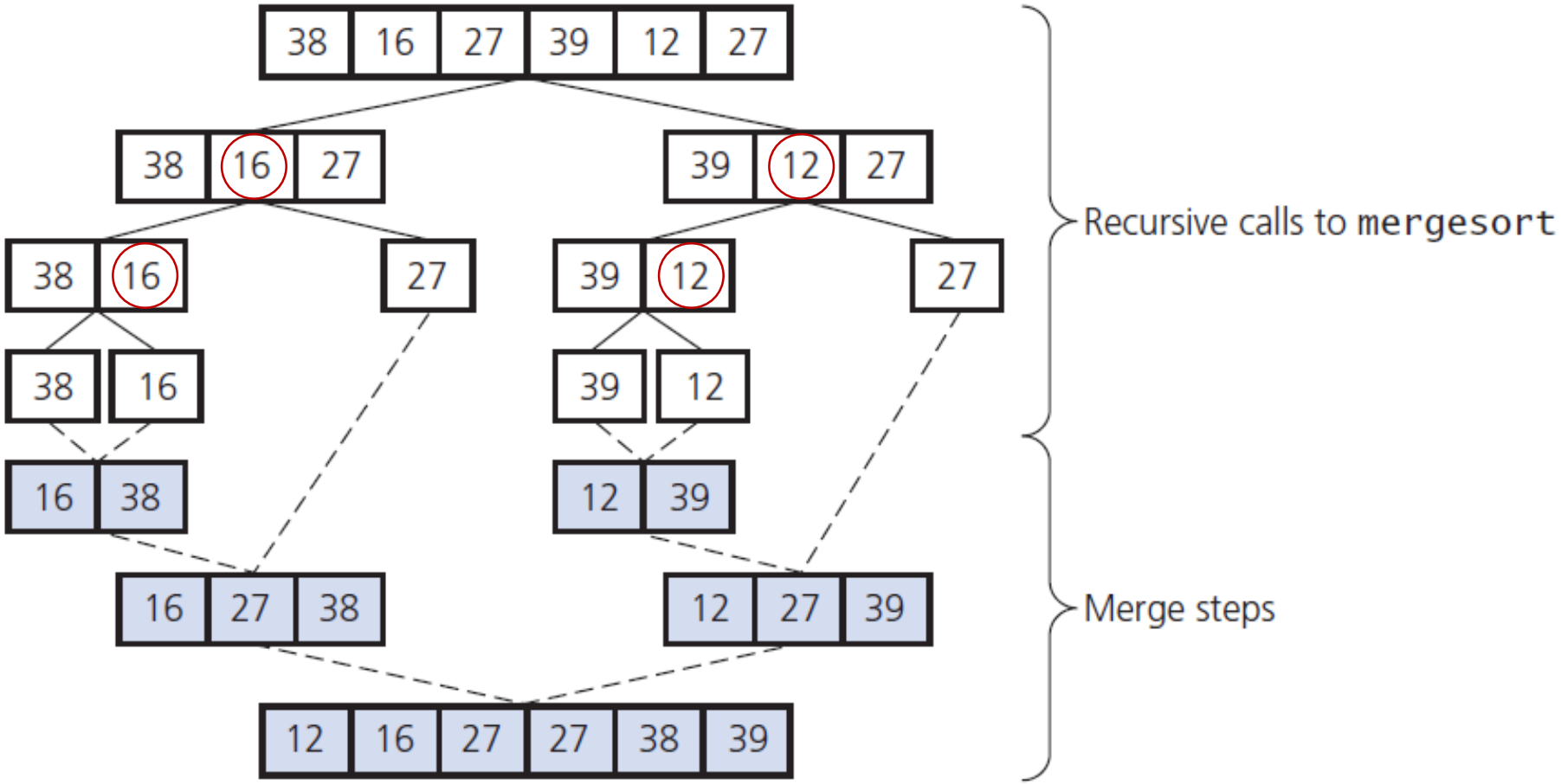


MERGE SORT



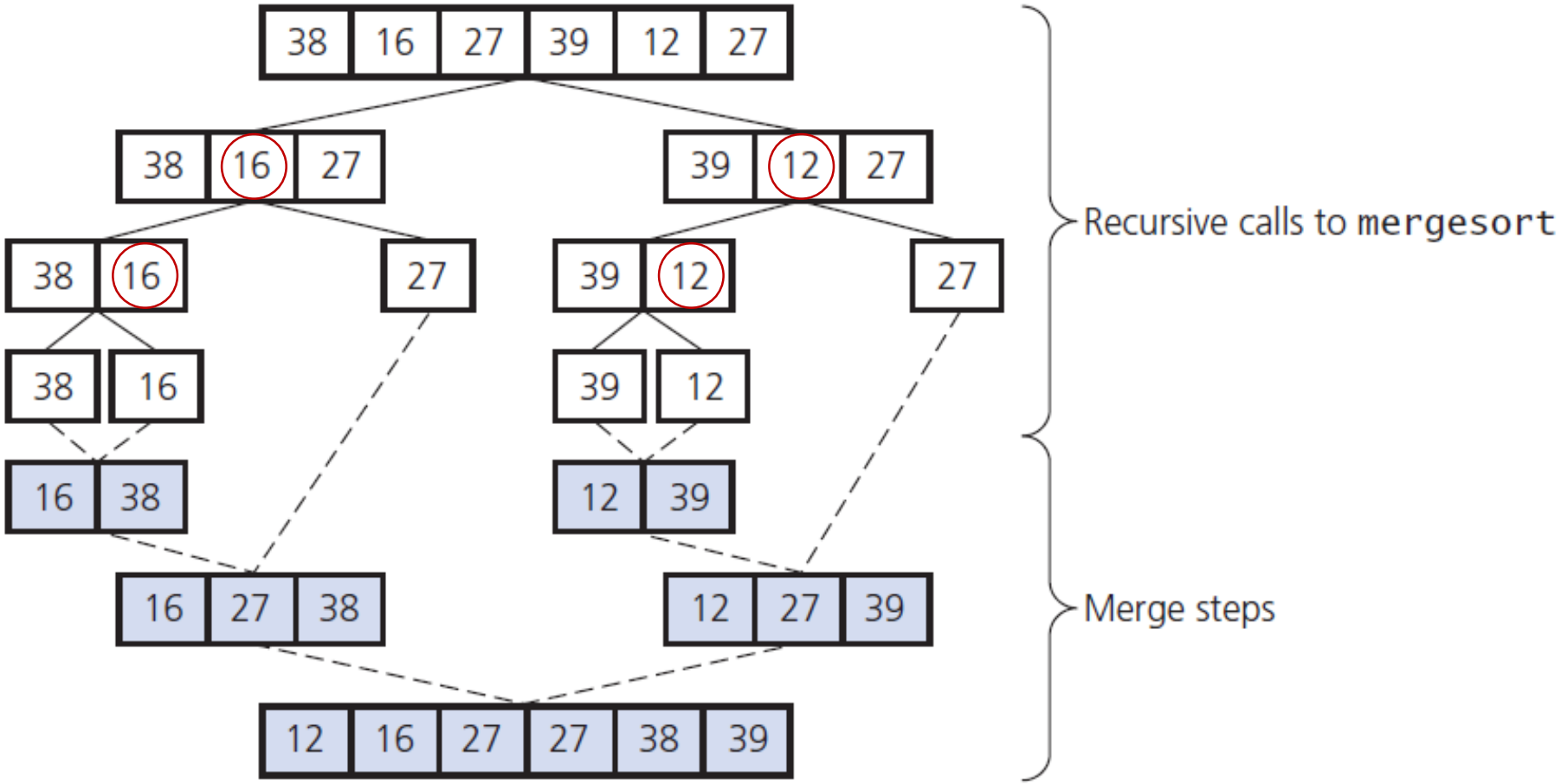
A merge sort with an auxiliary temporary array

Merge Sort

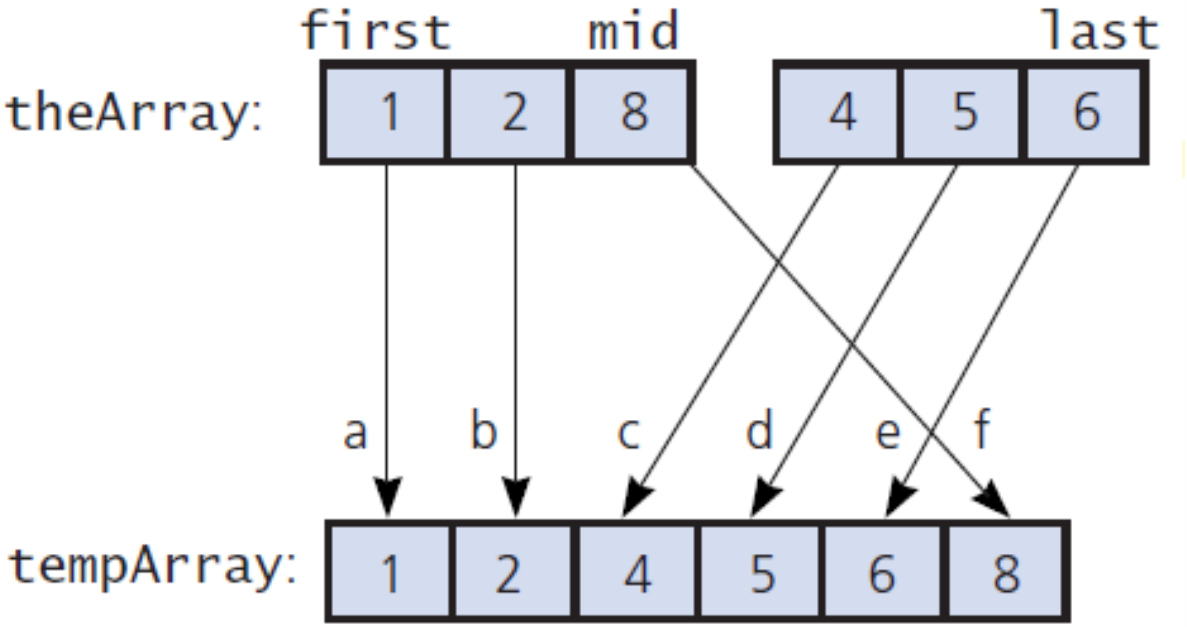


A merge sort with of an array of six integers

Merge Sort



A merge sort with of an array of six integers

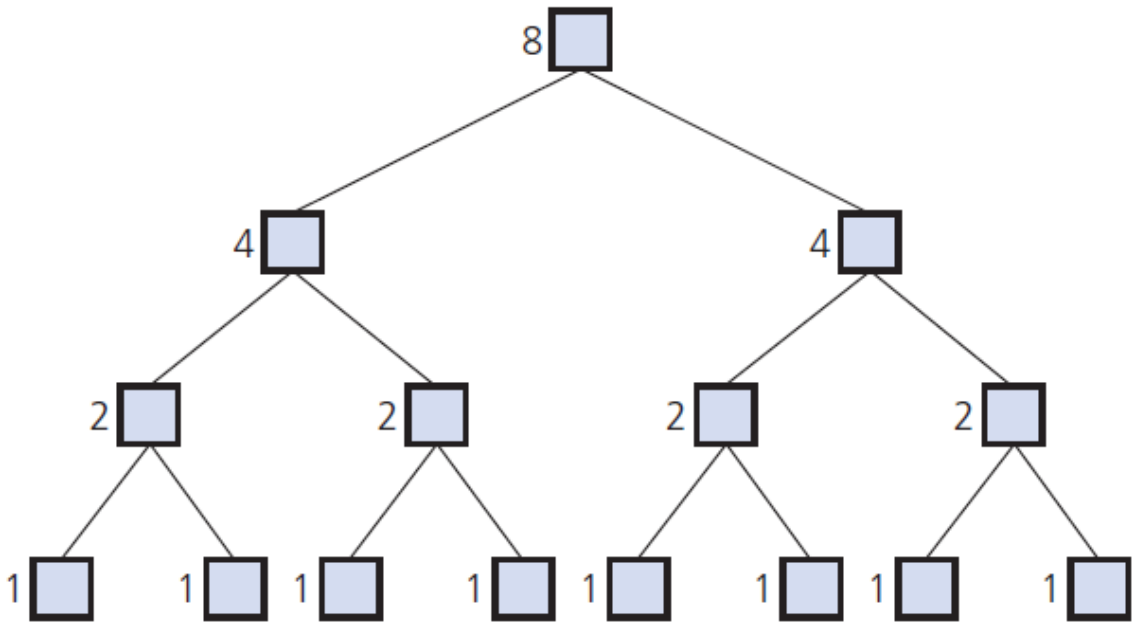


Merge the halves:

a. 1 < 4, so move 1 from `theArray[first..mid]` to `tempArray`
b. 2 < 4, so move 2 from `theArray[first..mid]` to `tempArray`
c. 8 > 4, so move 4 from `theArray[mid+1..last]` to `tempArray`
d. 8 > 5, so move 5 from `theArray[mid+1..last]` to `tempArray`
e. 8 > 6, so move 6 from `theArray[mid+1..last]` to `tempArray`
f. `theArray[mid+1..last]` is finished, so move 8 to `tempArray`

A worst-case instance of the merge step in a merge sort

MERGE SORT



Level 0: `mergesort` 8 items

Level 1: 2 calls to `mergesort` with 4 items each

Level 2: 4 calls to `mergesort` with 2 items each

Level 3: 8 calls to `mergesort` with 1 item each

Levels of recursive calls to Merge Sort, given an array of eight items

Merge Sort

```
46  template<class ItemType>
47  void mergeSort(ItemType theArray[], int first, int last) {
48      if (first < last) {
49          int mid = first + (last - first) / 2;
50          mergeSort(theArray, first, mid);
51          mergeSort(theArray, mid + 1, last);
52          merge(theArray, first, mid, last);
53      }
54  }
```

Merge Sort

```
7   const int MAX_SIZE = 50;
8
9   template<class ItemType>
10  void merge(ItemType theArray[], int first, int mid, int last) {
11      ItemType tempArray[MAX_SIZE];
12
13      int first1 = first;
14      int last1 = mid;
15      int first2 = mid + 1;
16      int last2 = last;
17
18      int index = first1;
19      while ((first1 <= last1) && (first2 <= last2)) {
20          if (theArray[first1] <= theArray[first2]) {
21              tempArray[index] = theArray[first1];
22              first1++;
23          } else {
24              tempArray[index] = theArray[first2];
25              first2++;
26          }
27          index++;
28      }
```

```
30      while (first1 <= last1) {
31          tempArray[index] = theArray[first1];
32          first1++;
33          index++;
34      }
35
36      while (first2 <= last2) {
37          tempArray[index] = theArray[first2];
38          first2++;
39          index++;
40      }
41
42      for (index = first; index <= last; index++)
43          theArray[index] = tempArray[index];
44  }
```

Worst Case of Merge Sort

The worst case of merge sort will be the one where merge sort will have to do maximum number of comparisons.

Suppose the array in final step after sorting is {0,1,2,3,4,5,6,7}

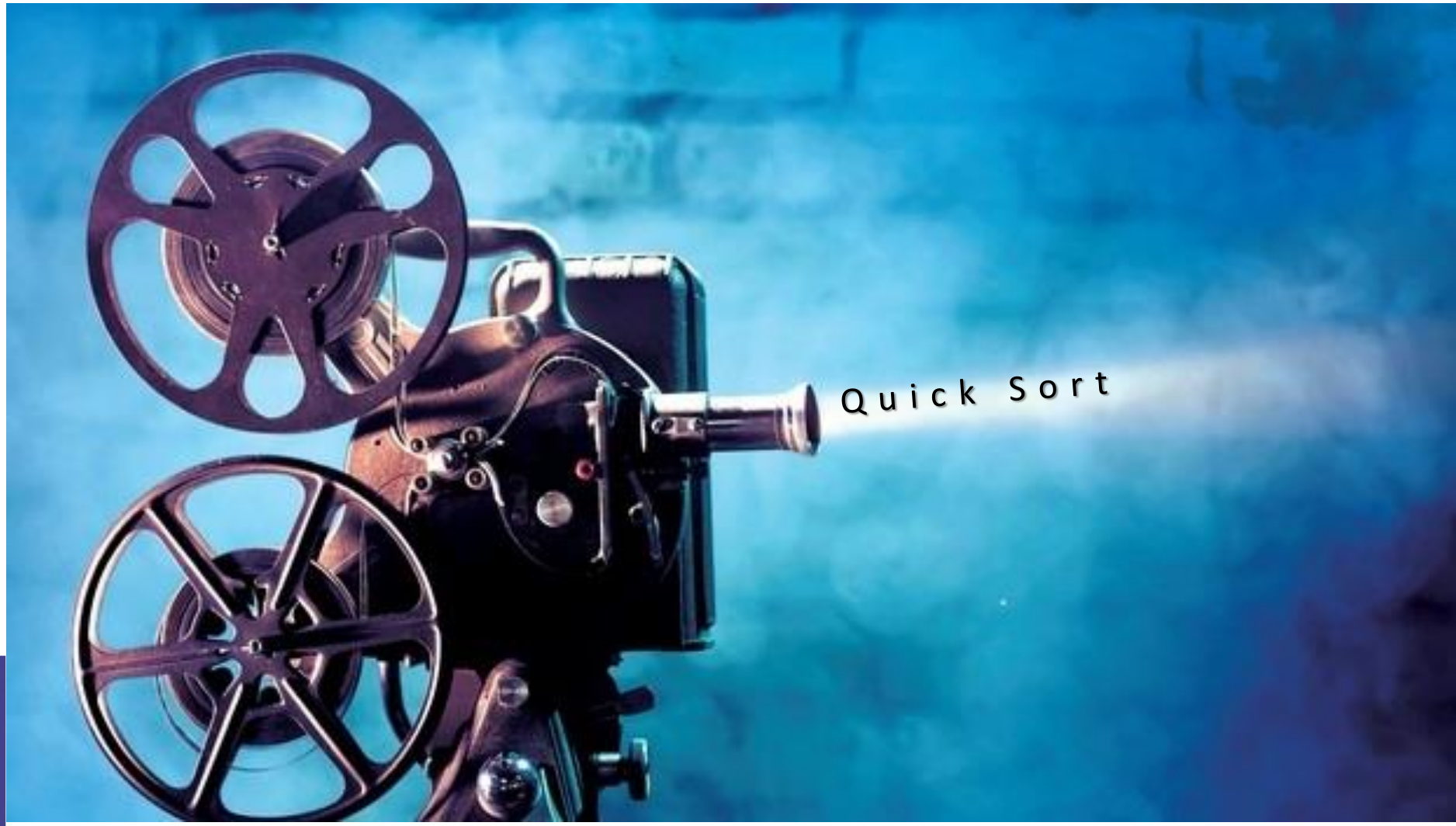
For worst case the array before this step must be {0,2,4,6,1,3,5,7} because here left subarray = {0,2,4,6} and right subarray = {1,3,5,7} will result in maximum comparisons. (Storing alternate elements in left and right subarray)

Reason: Every element of array will be compared at least once.

Applying the same above logic for left and right subarray for previous steps : For array {0,2,4,6} the worst case will be if the previous array is {0,4} and {2,6} and for array {1,3,5,7} the worst case will be for {1,5} and {3,7}.

Now applying the same for previous step arrays: For worst cases: {0,4} must be {4,0} , {2,6} must be {6,2} ,{1,5} must be {5,1} {3,7} must be {7,3} . This step is not necessary because if the size of set/array is 2 then every element will be compared at least once even if array of size 2 is sorted.

QUICK SORT


Quick Sort

- Quick sort orders values by **partitioning the list around one element**, then sorting each partition.

- More specifically:
  1. **Choose** one element in the list to be **the partition element/the pivot**
  2. **Organize** the elements so that all elements **less than** the pivot are **to the left** and all **greater** are **to the right**
  3. Apply the **quick sort** algorithm (recursively) to **both partitions**

The Pivot

(Divide and Conquer / Recursive)

- Quick sort orders values by **partitioning the list around one element**, then sorting each partition.

- More specifically: OR
    1. Find the **pivot**
    2. Quick Sort the **left**
    3. Quick Sort the **right**

The Pivot

(Divide and Conquer / Recursive)

Quick Sort

- The pivot: **LEFT** < the **Pivot** < **RIGHT**

1. Find the Pivot

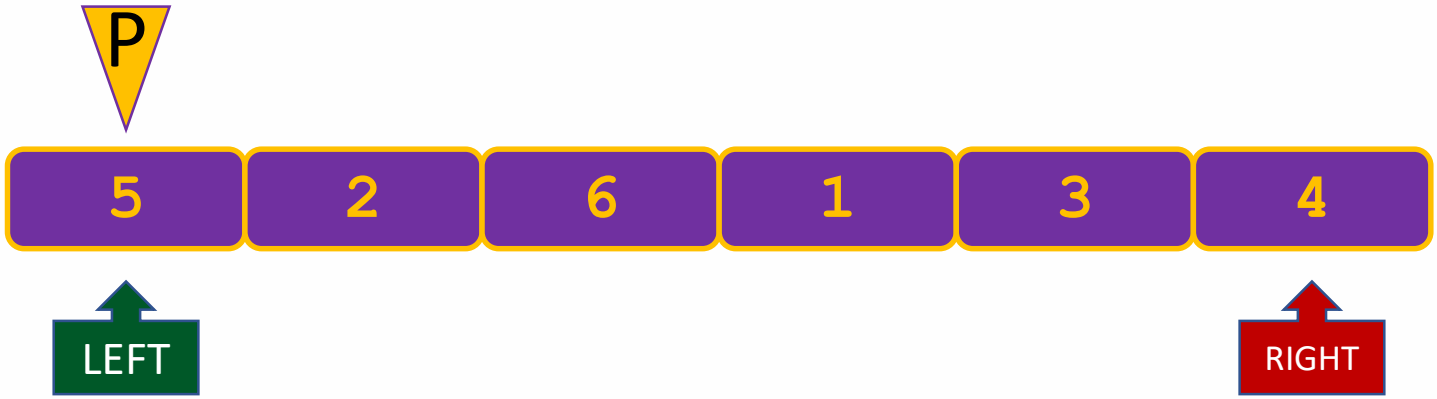
- The pivot: **LEFT** < the **Pivot** < **RIGHT**

1. Find the Pivot

- The pivot: **LEFT** < the **Pivot** < **RIGHT**

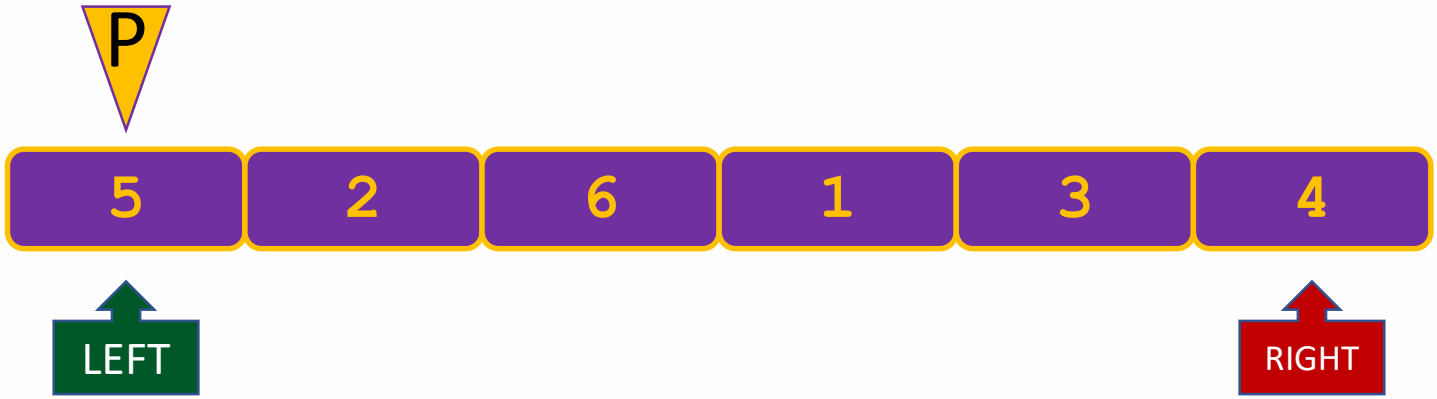
1. Find the Pivot

- The pivot: **LEFT** < the **Pivot** < **RIGHT**



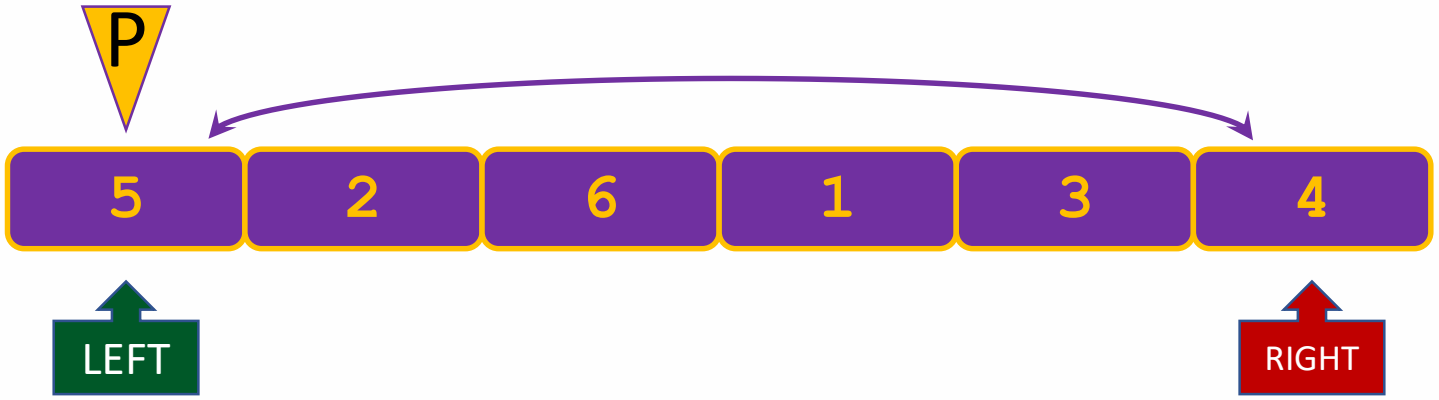
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



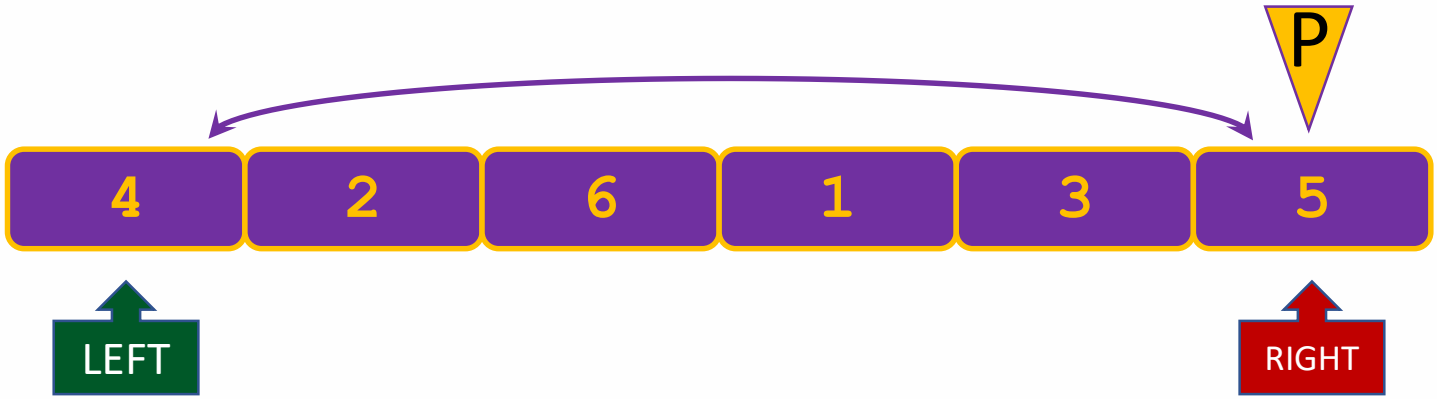
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



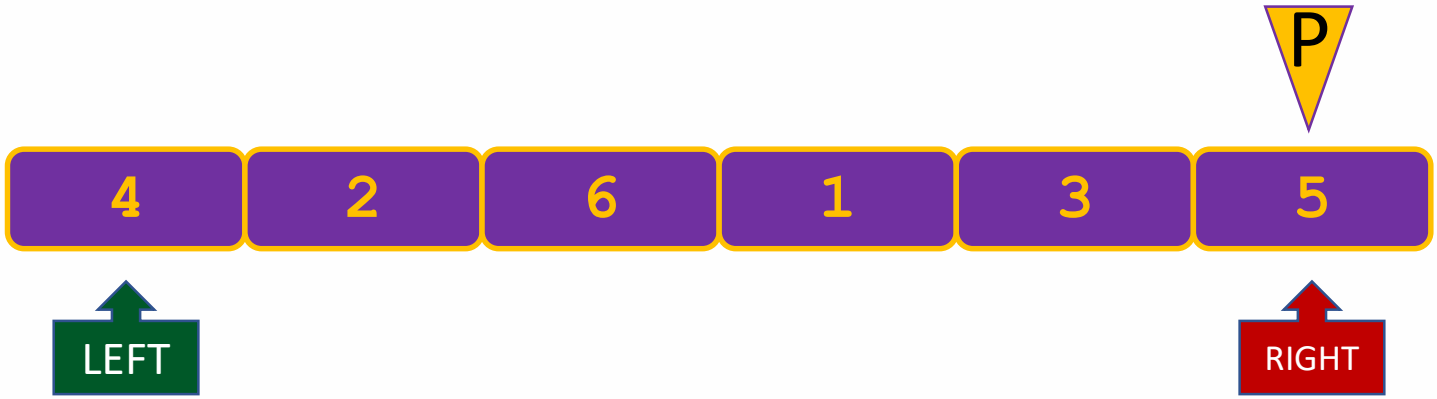
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



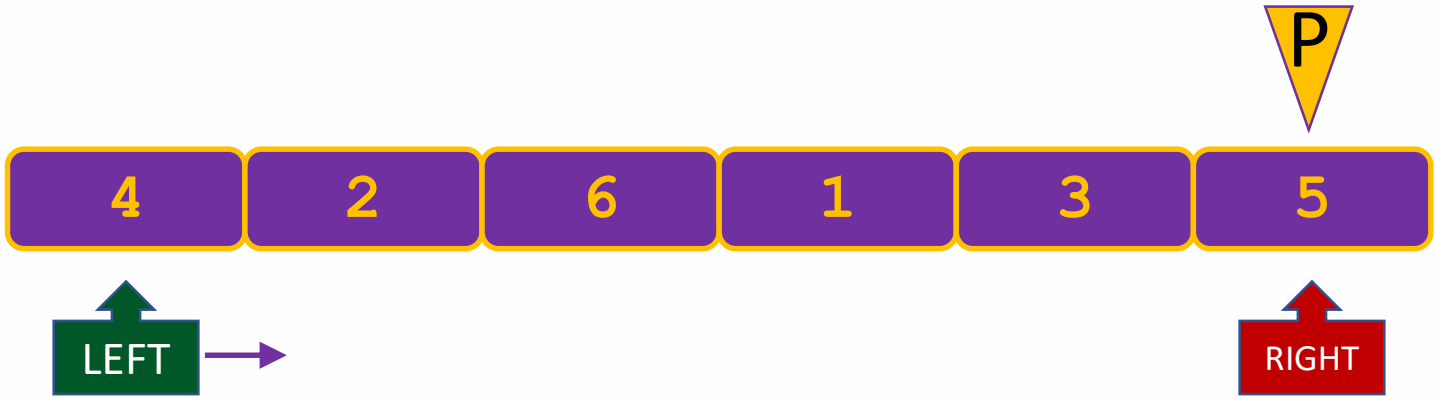
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



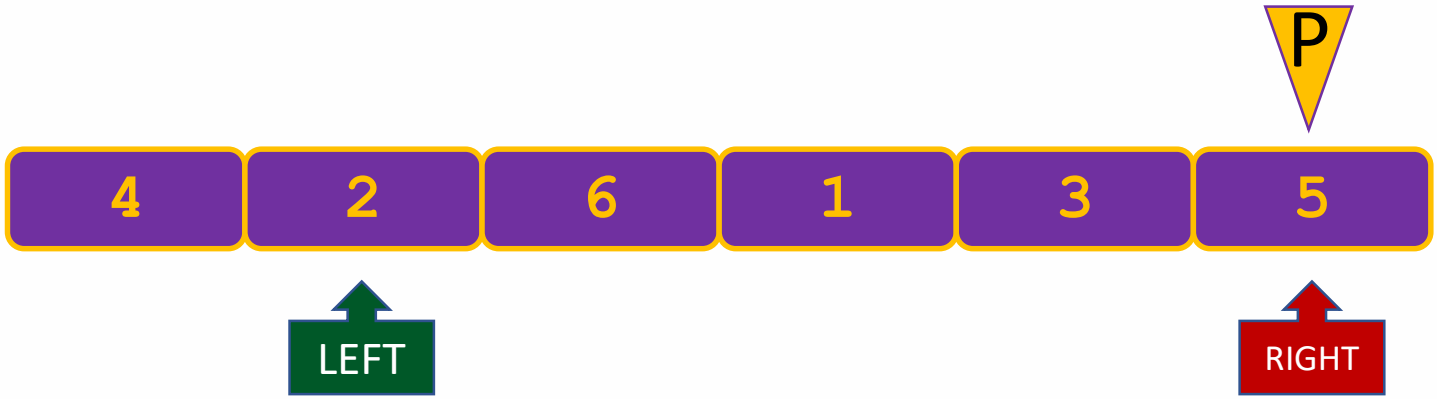
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



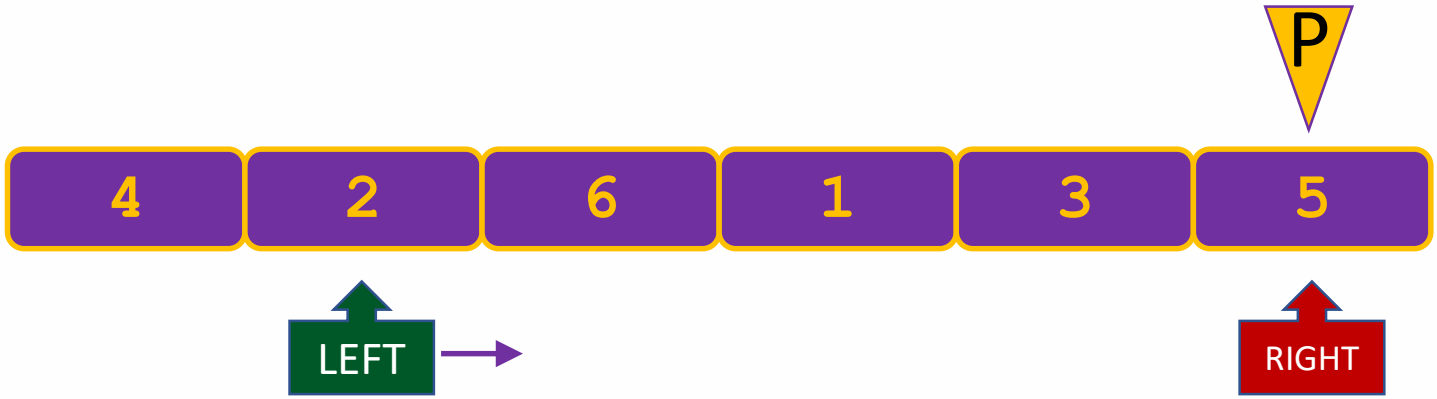
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



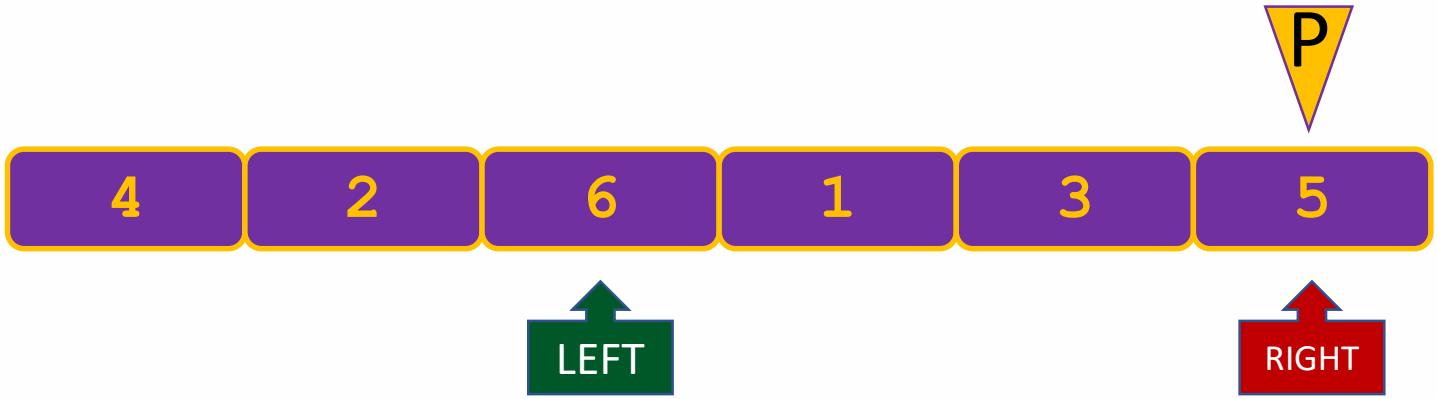
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



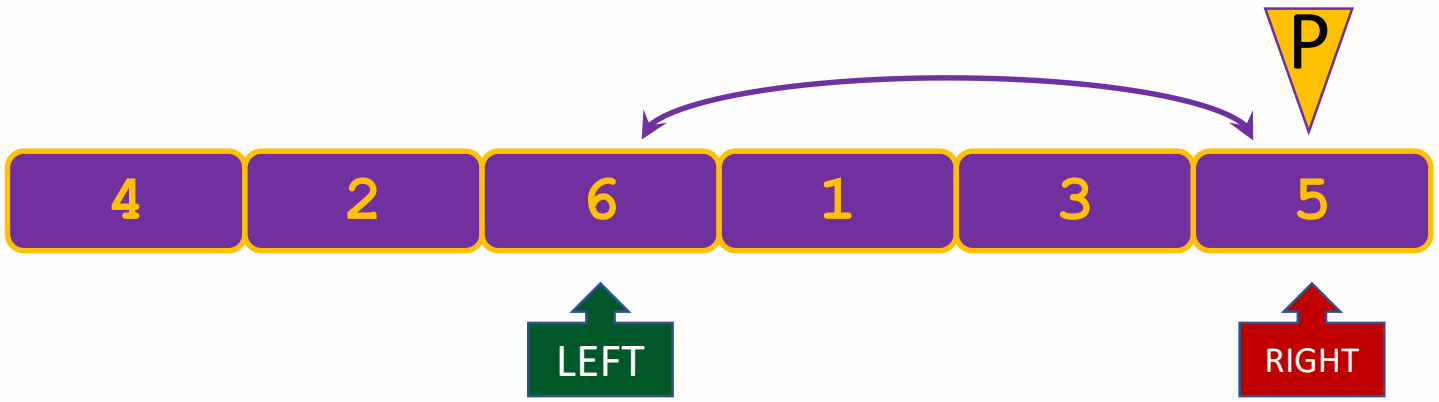
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



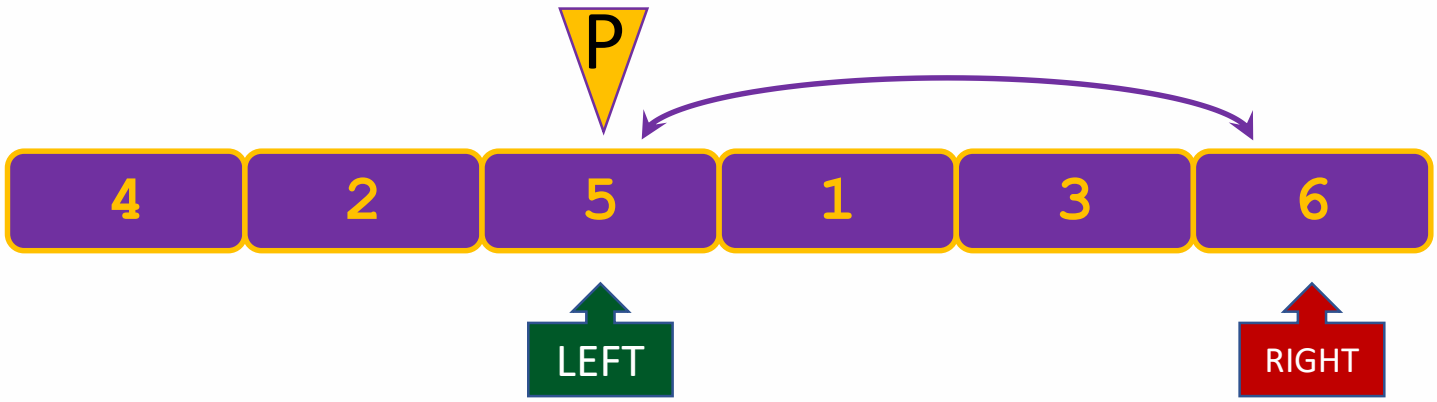
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



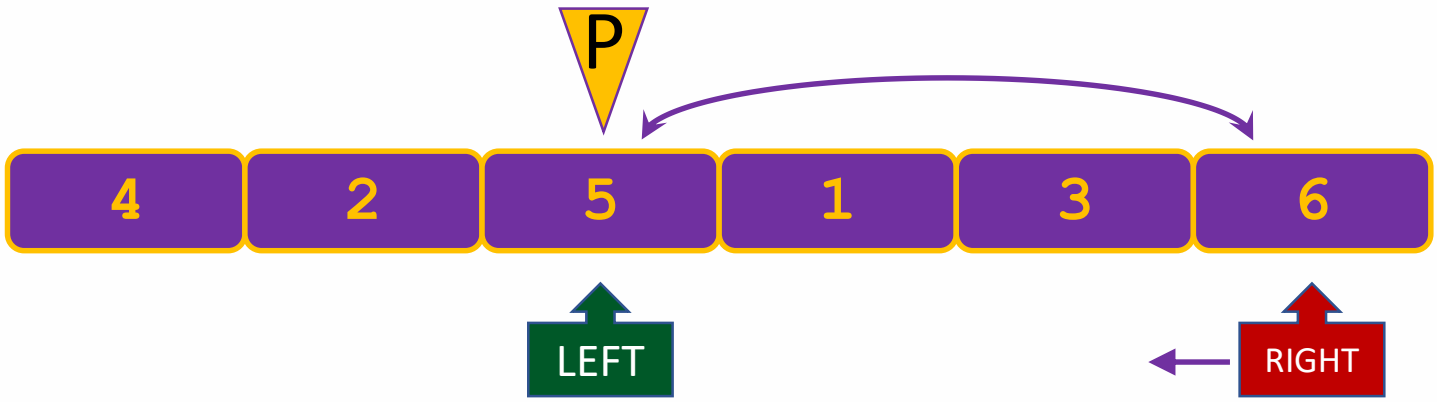
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



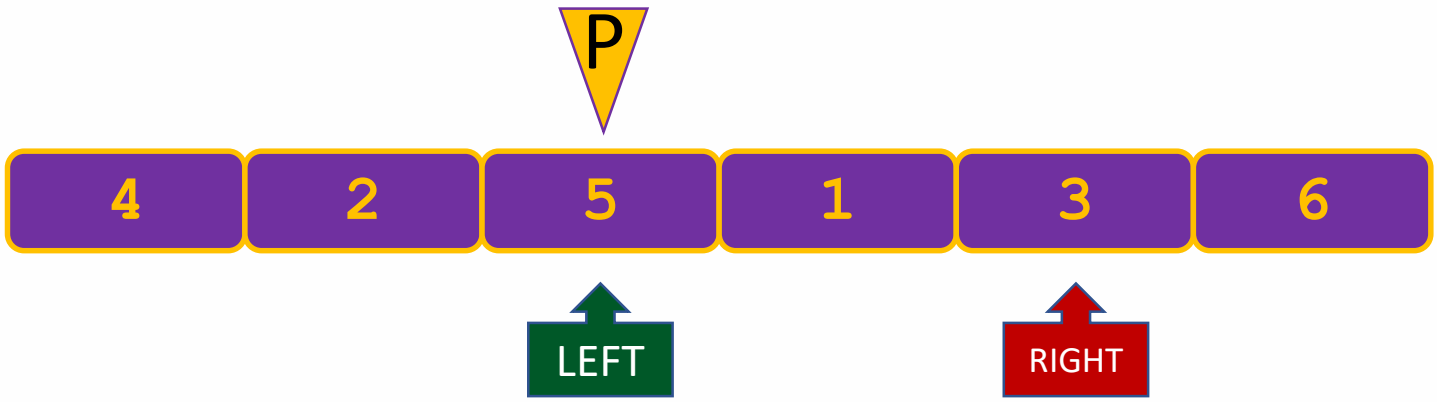
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



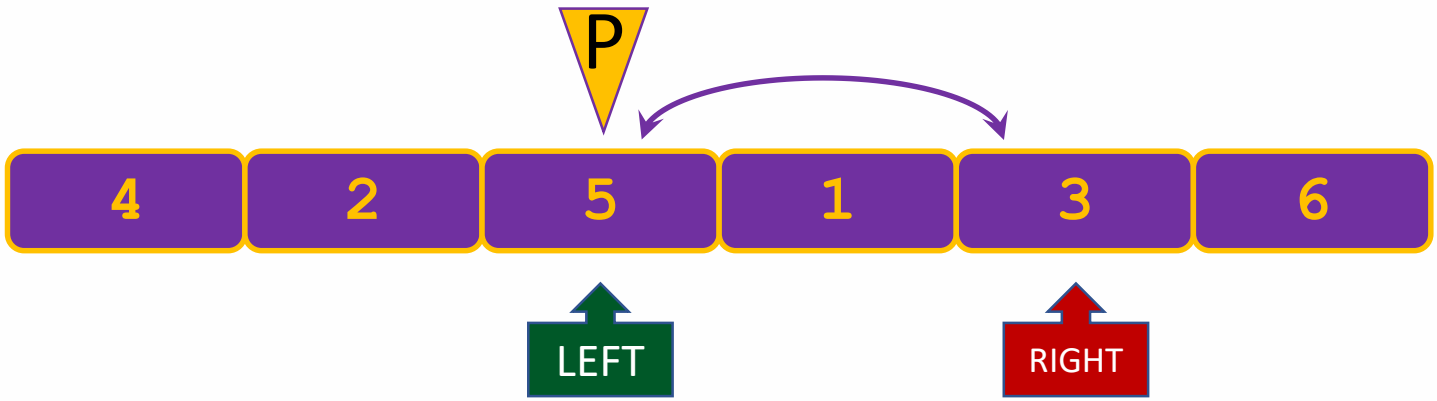
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



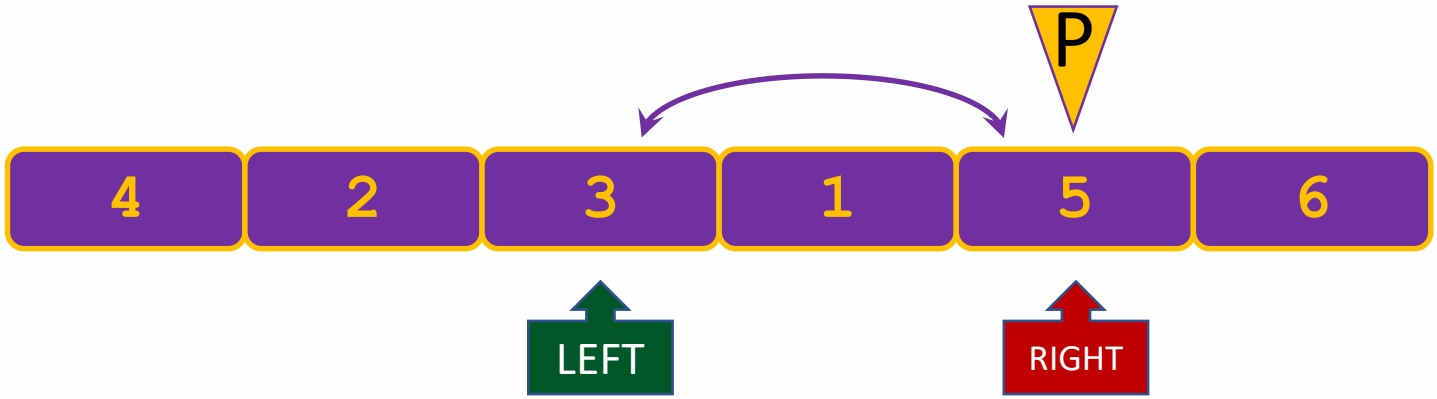
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



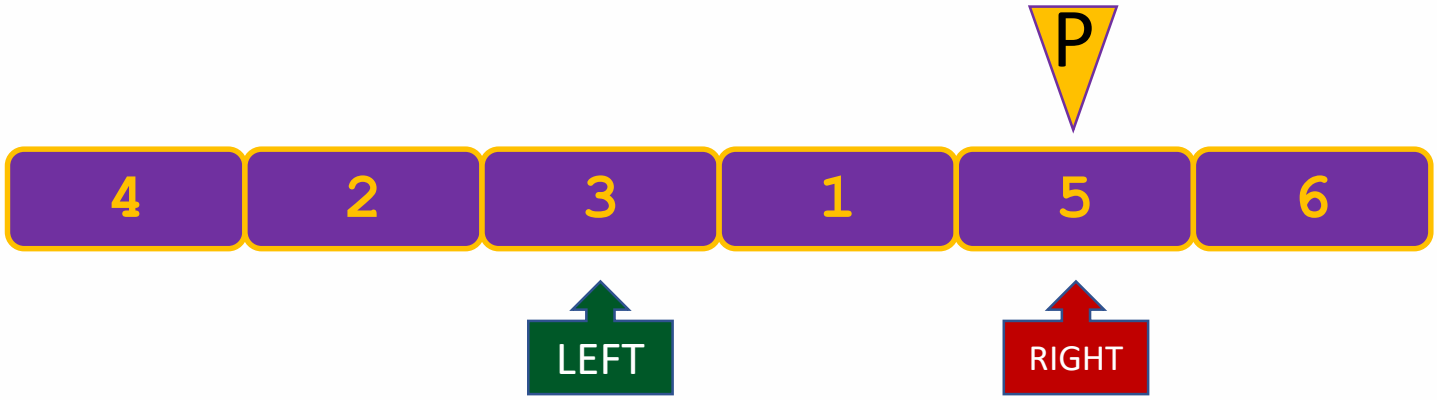
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



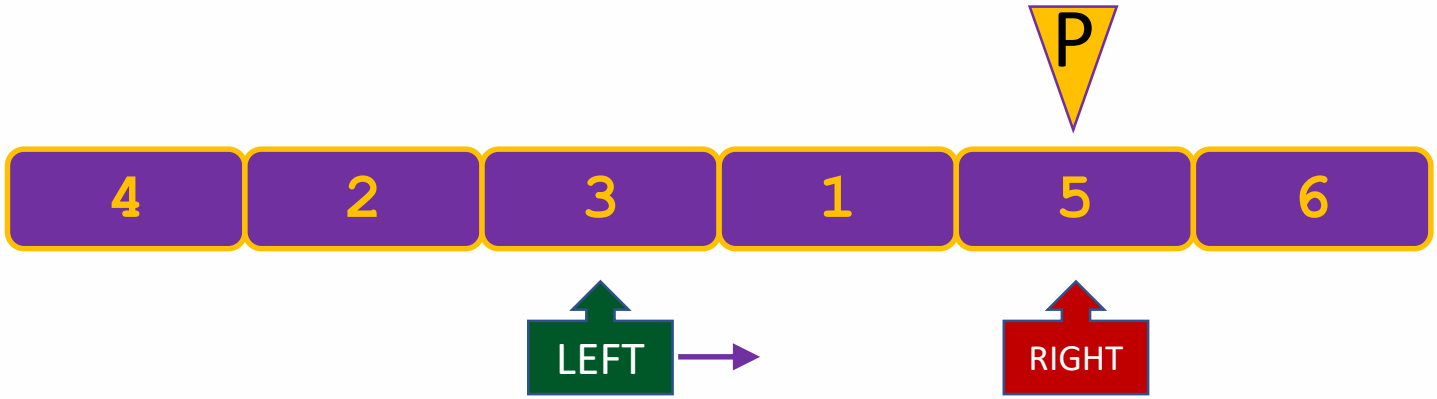
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



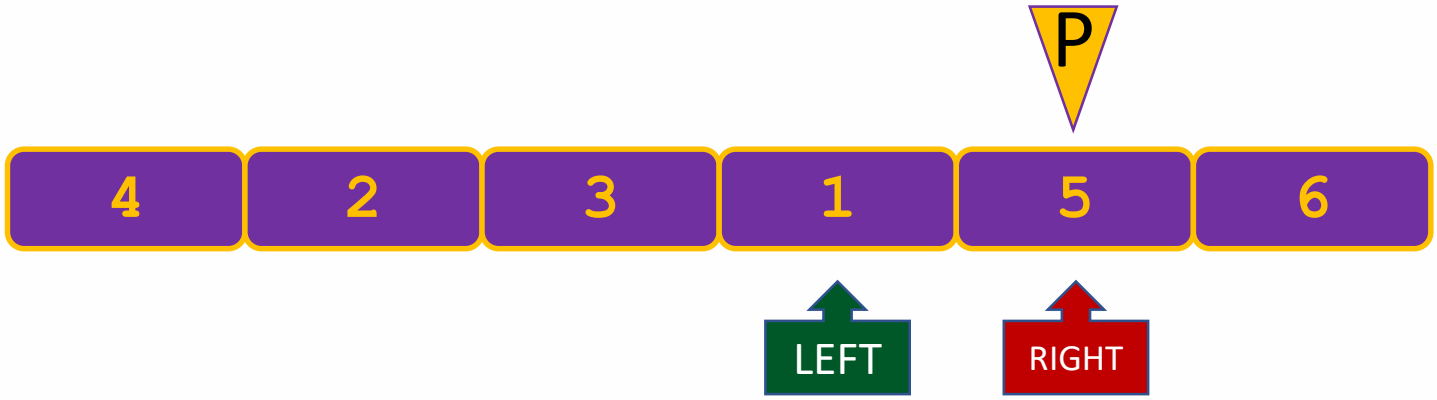
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



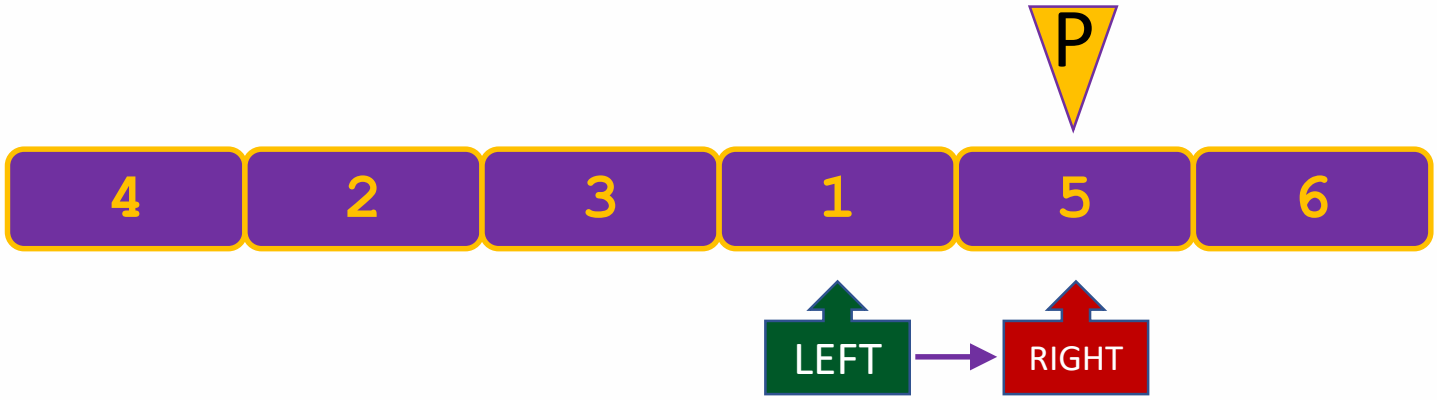
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



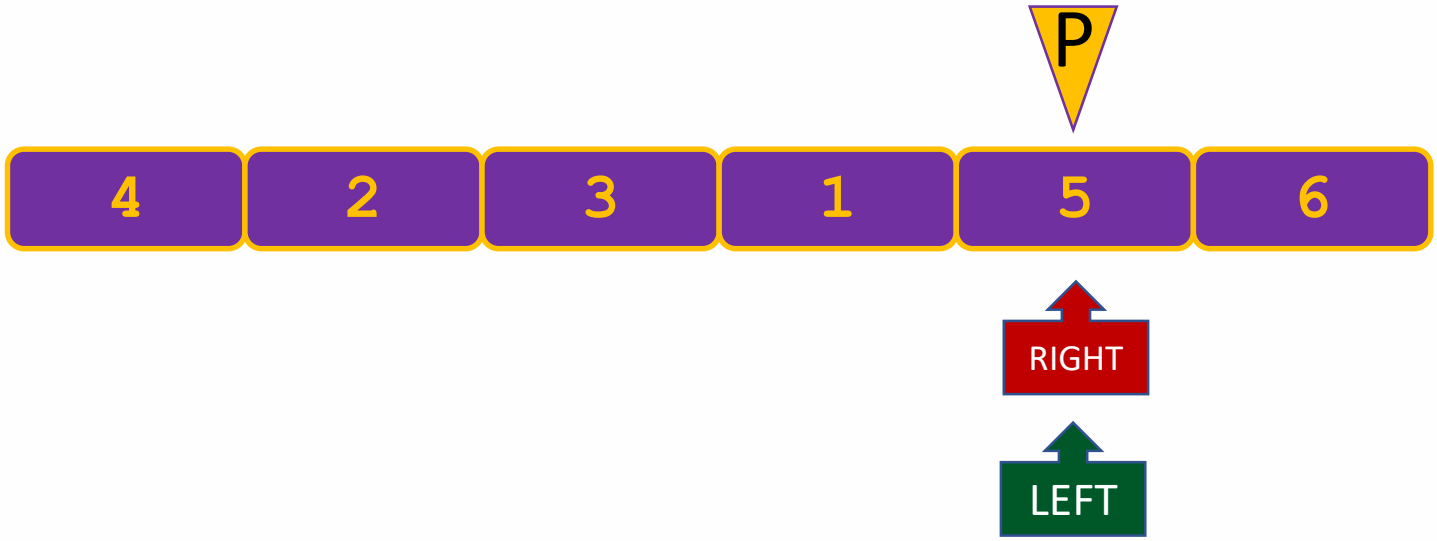
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



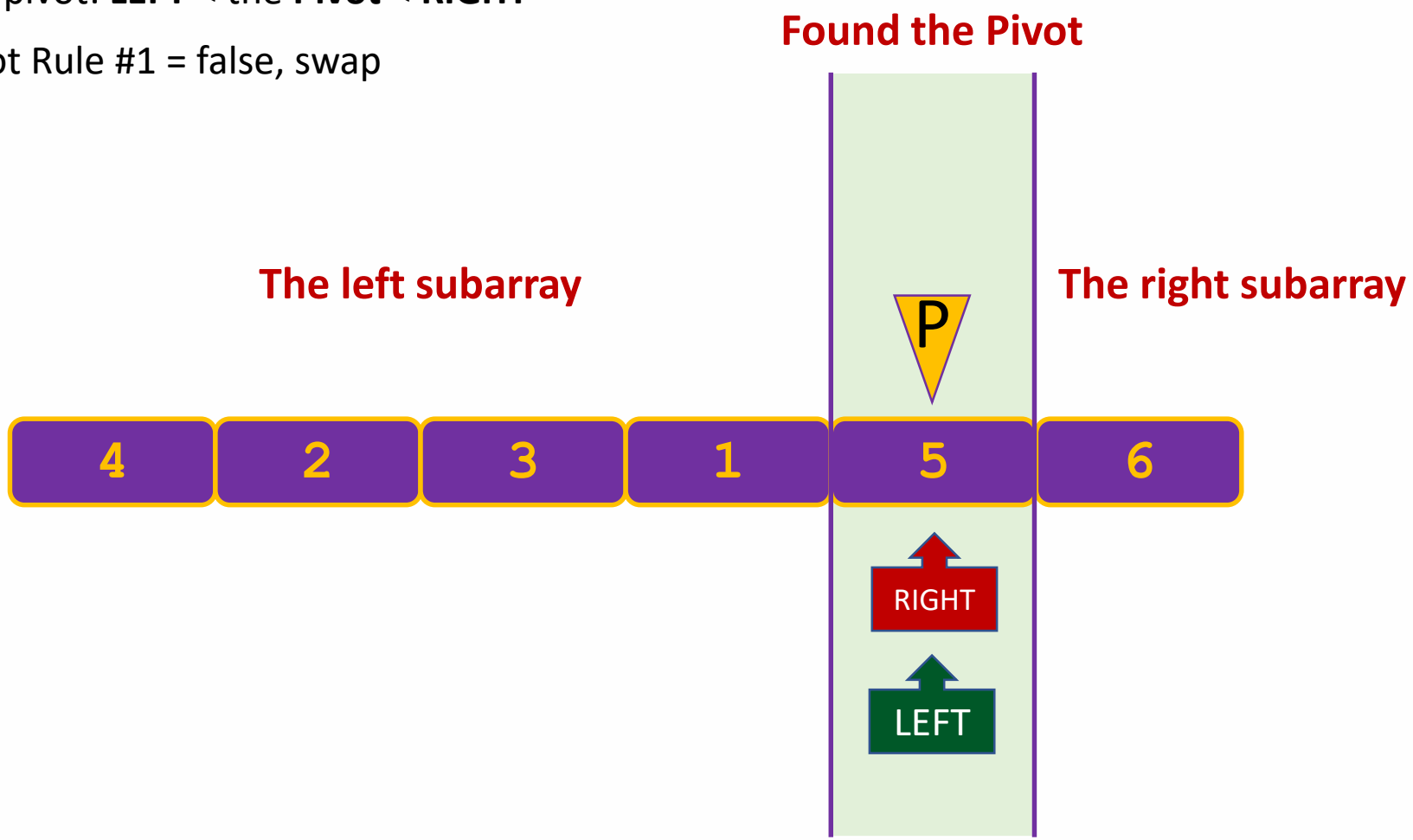
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



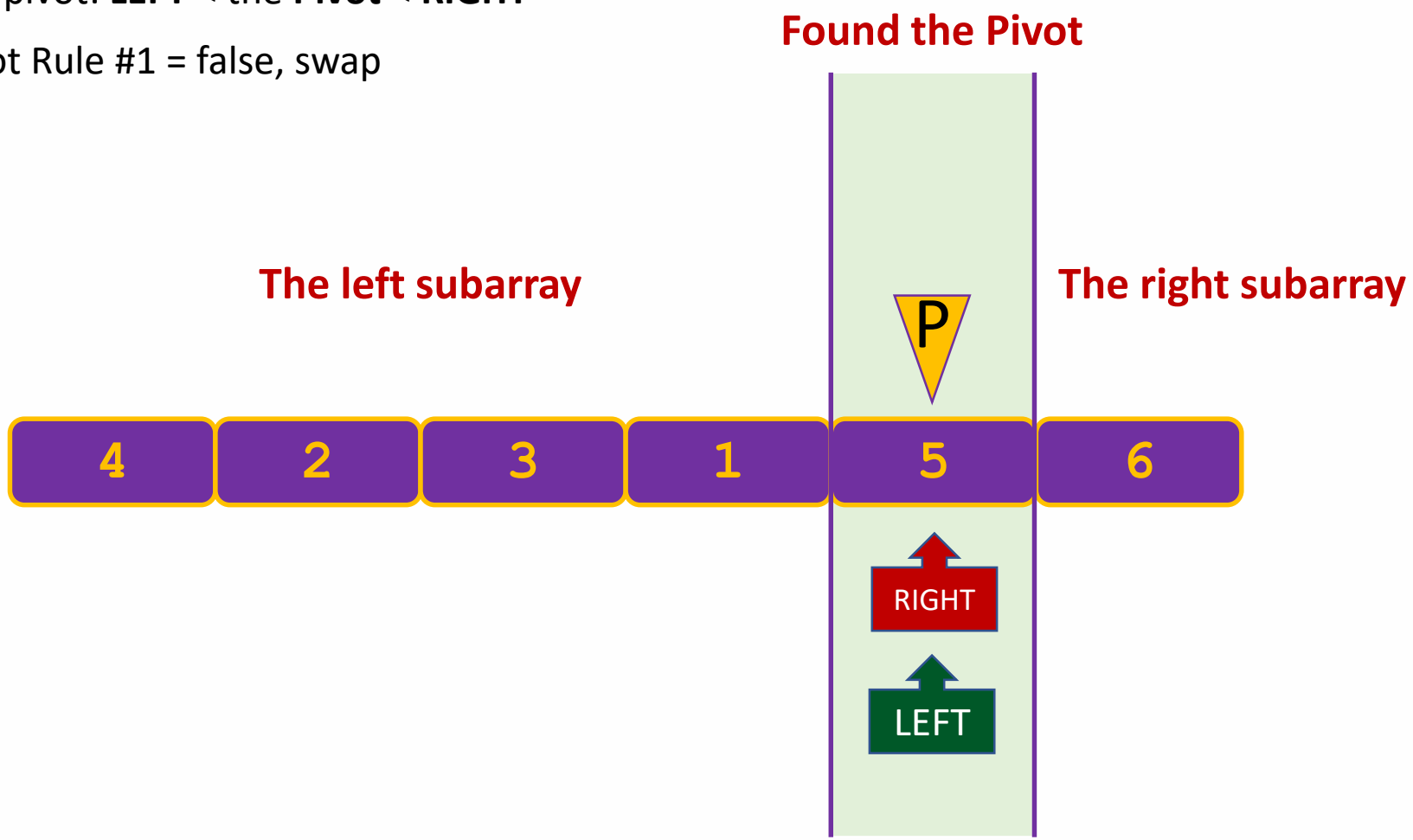
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



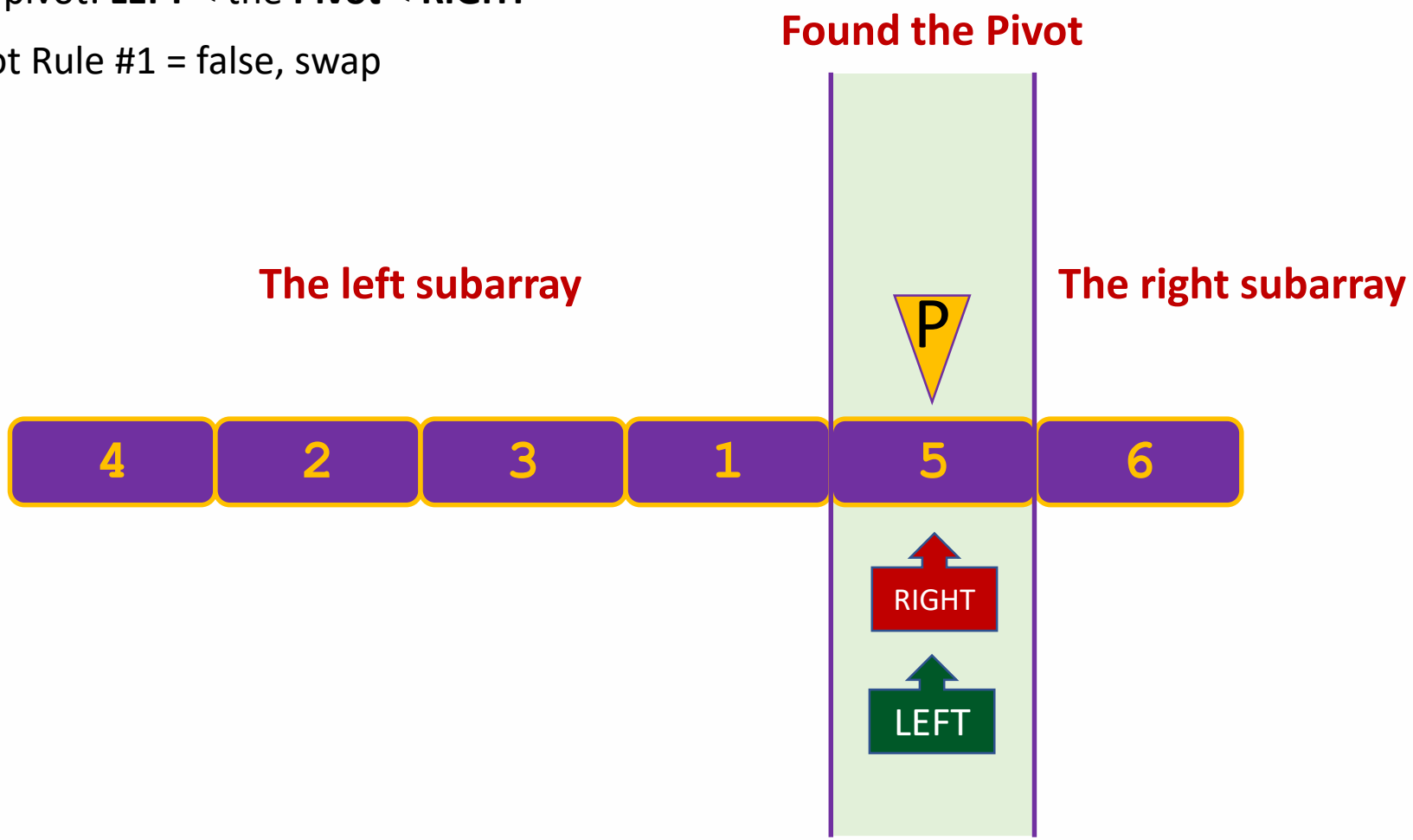
1. Find the Pivot

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



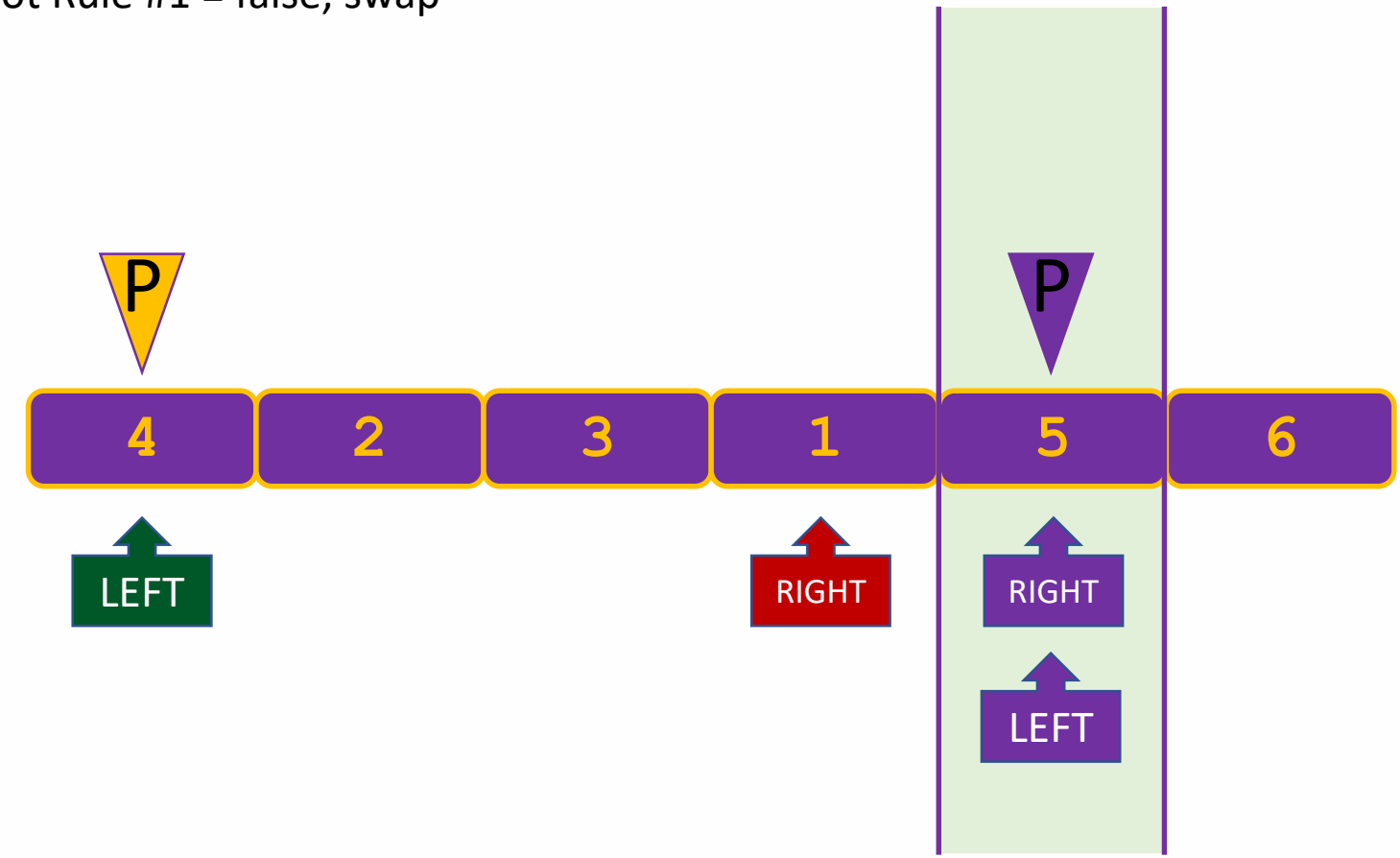
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



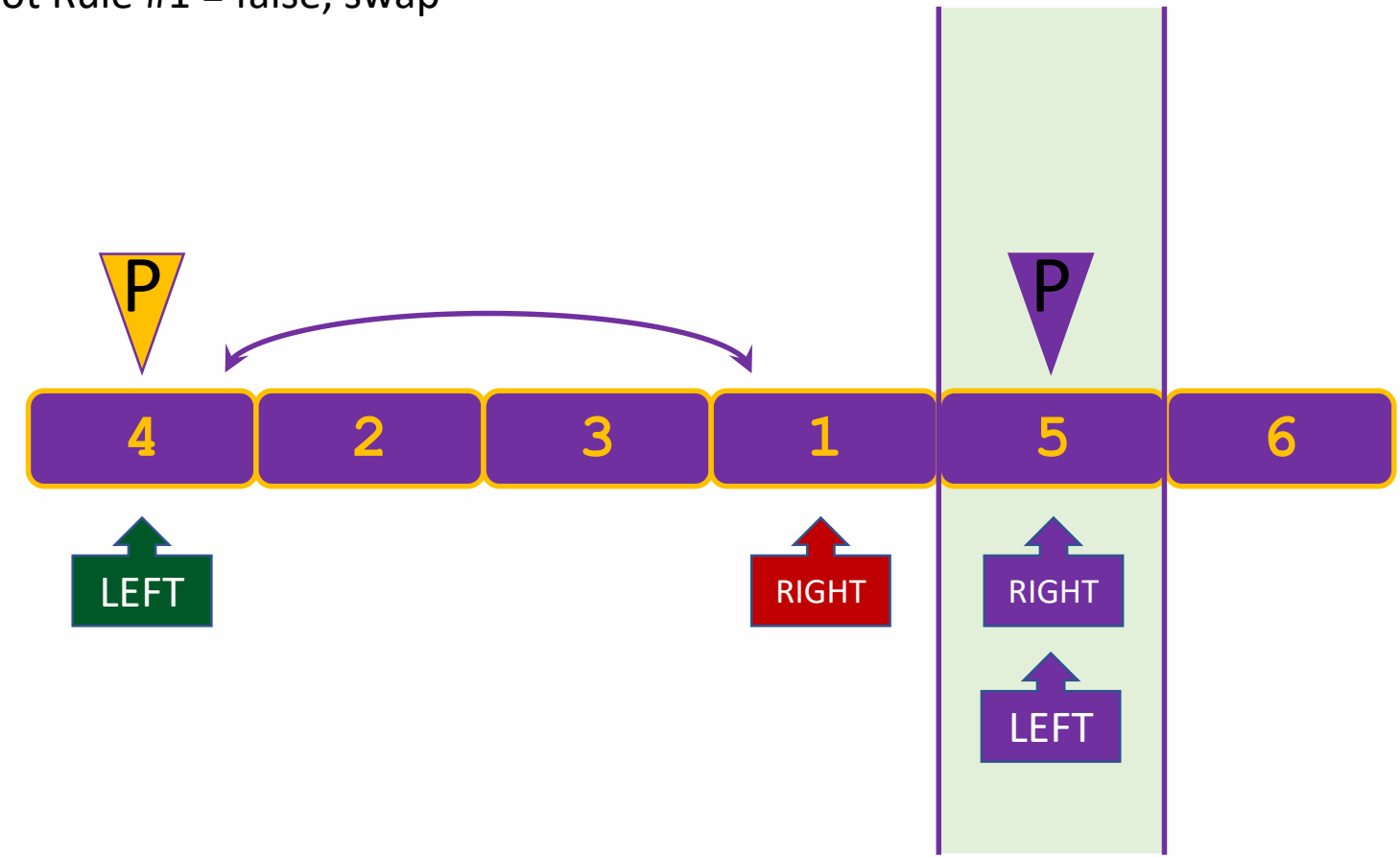
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



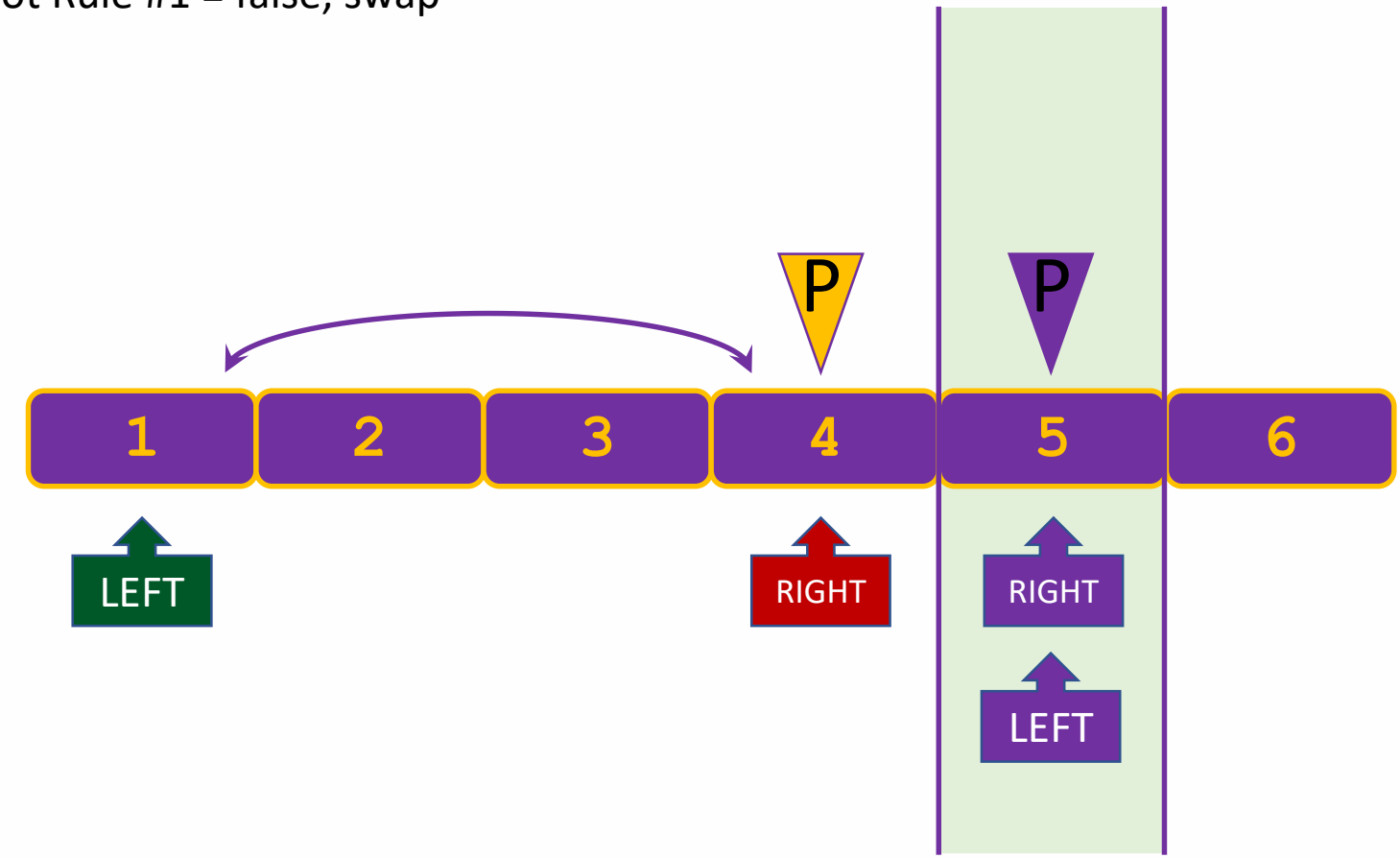
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



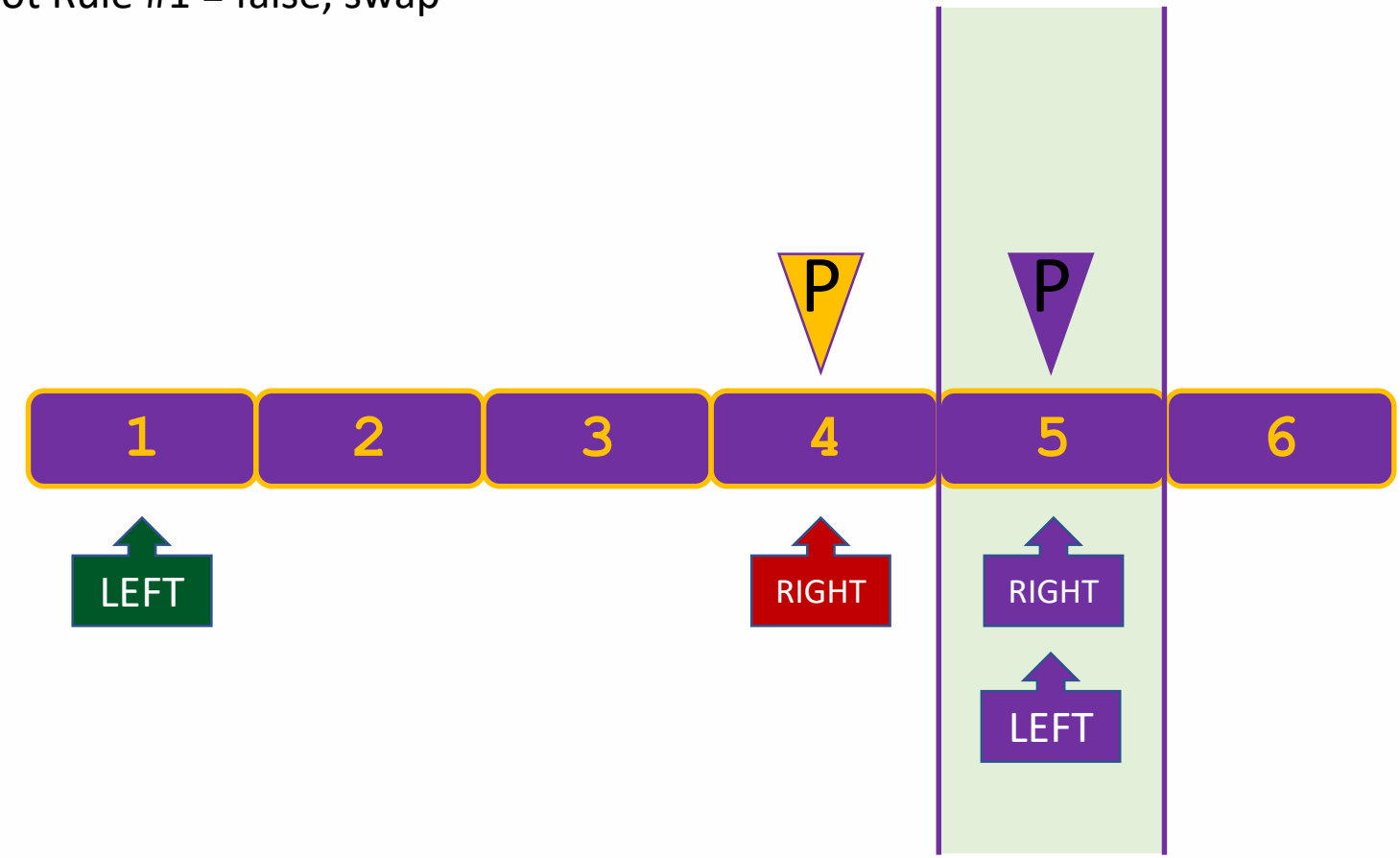
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



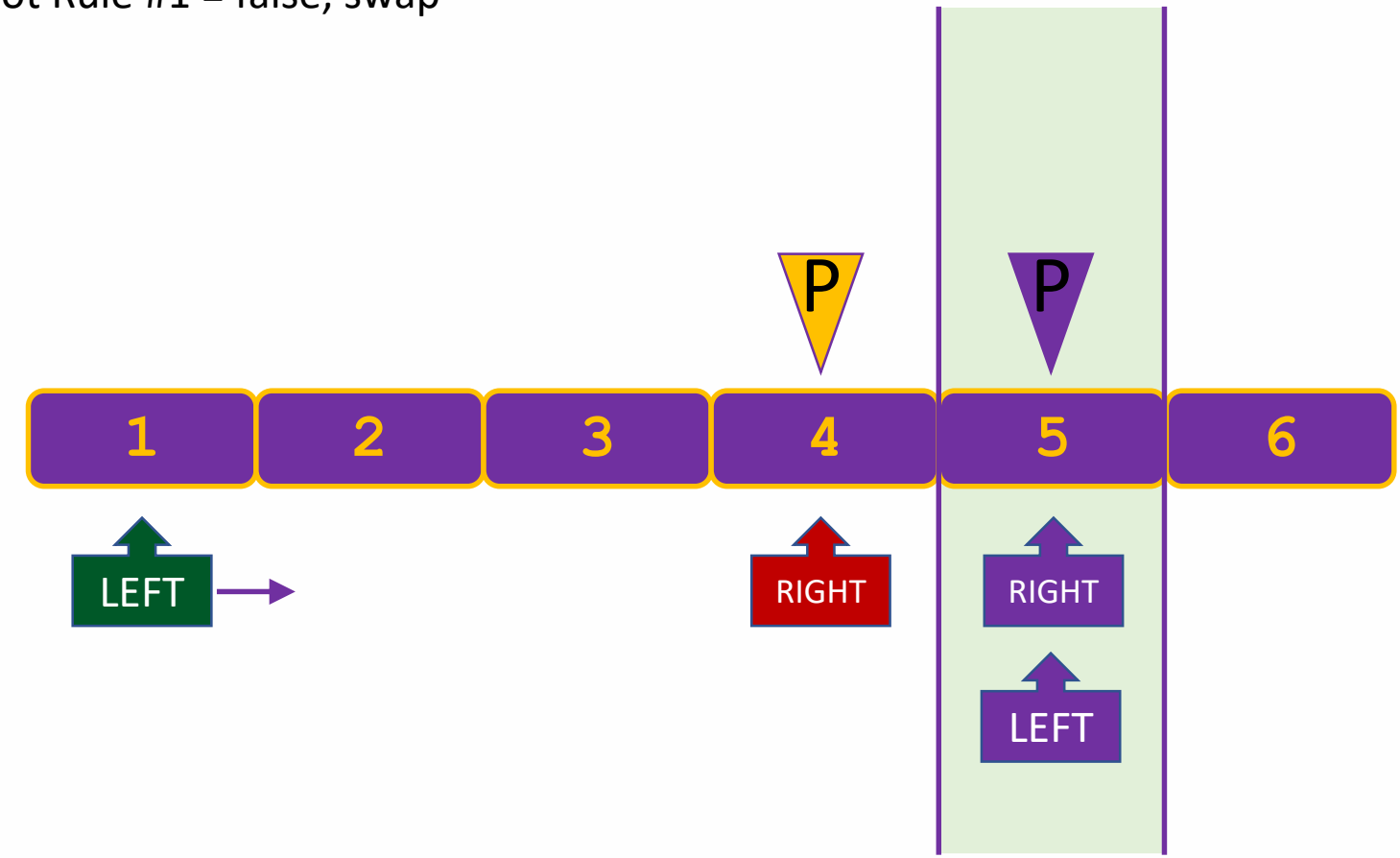
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



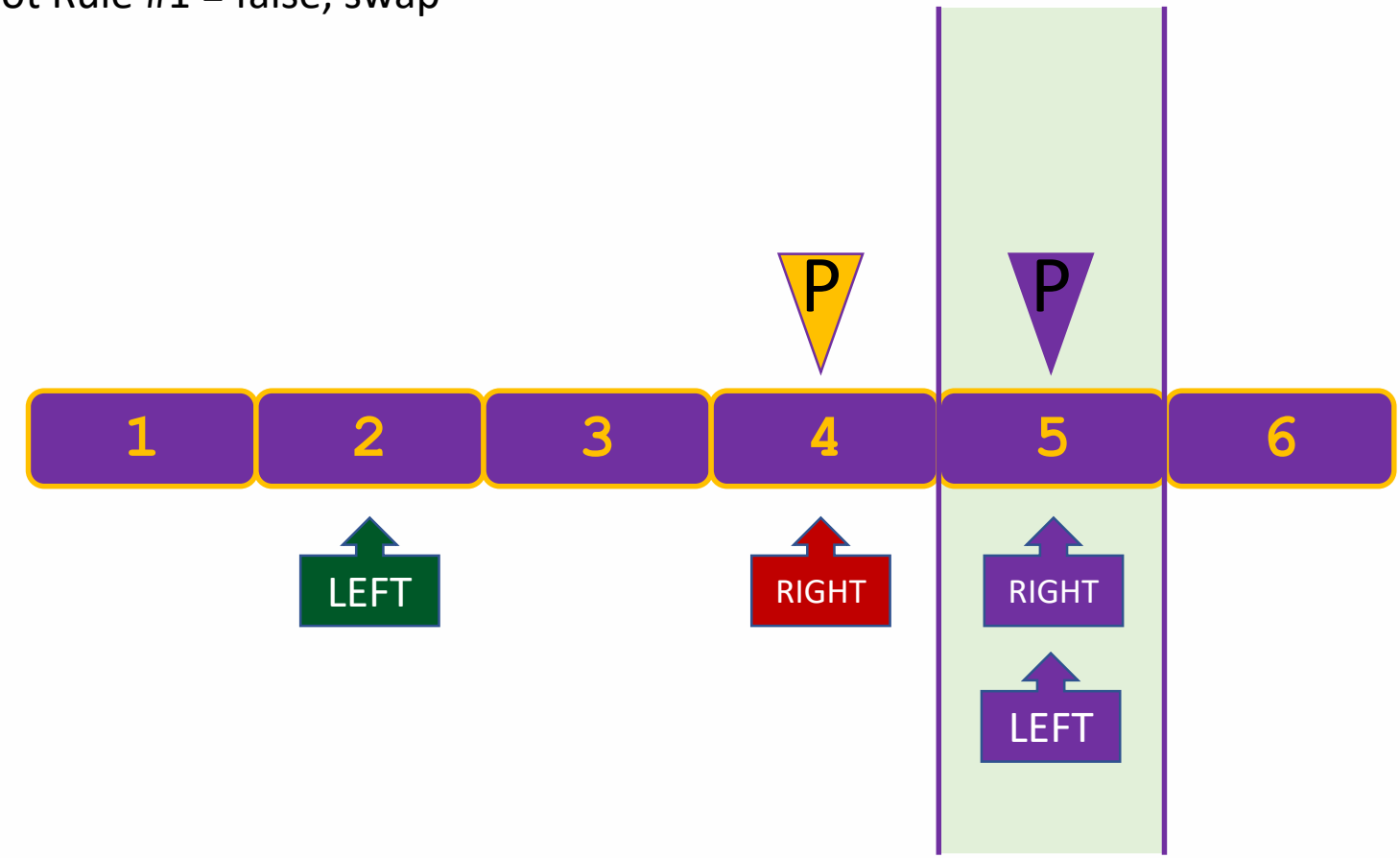
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



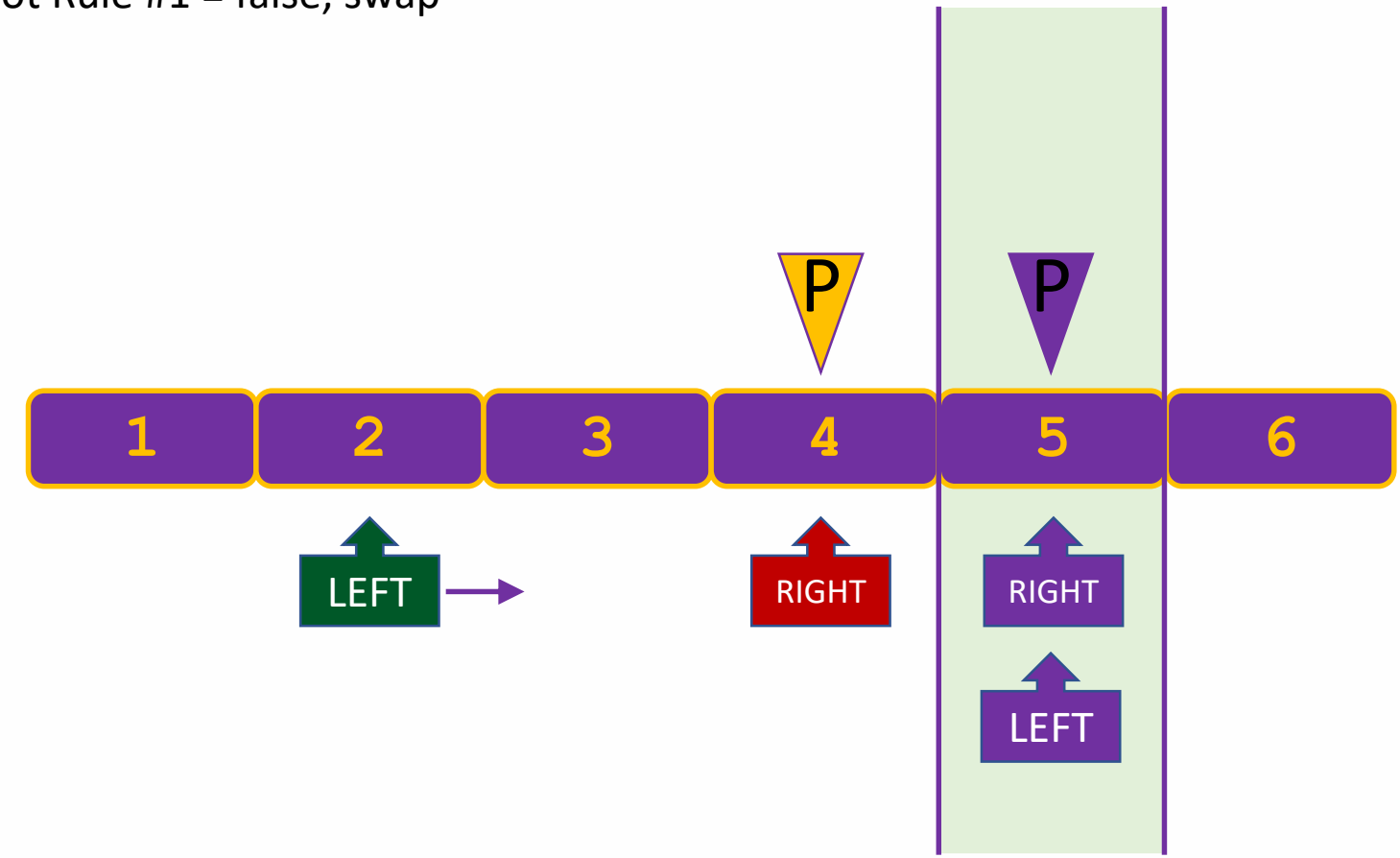
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



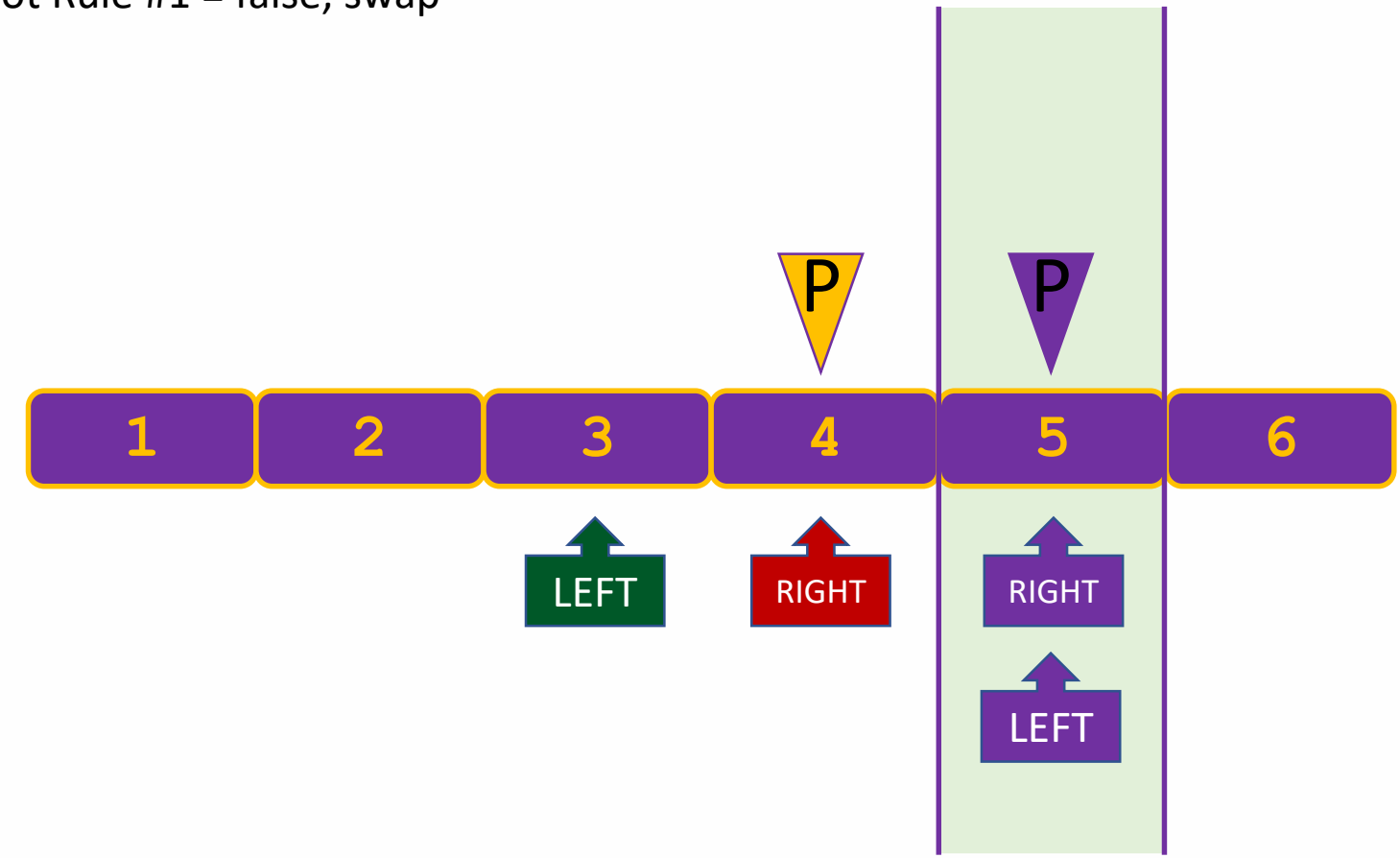
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



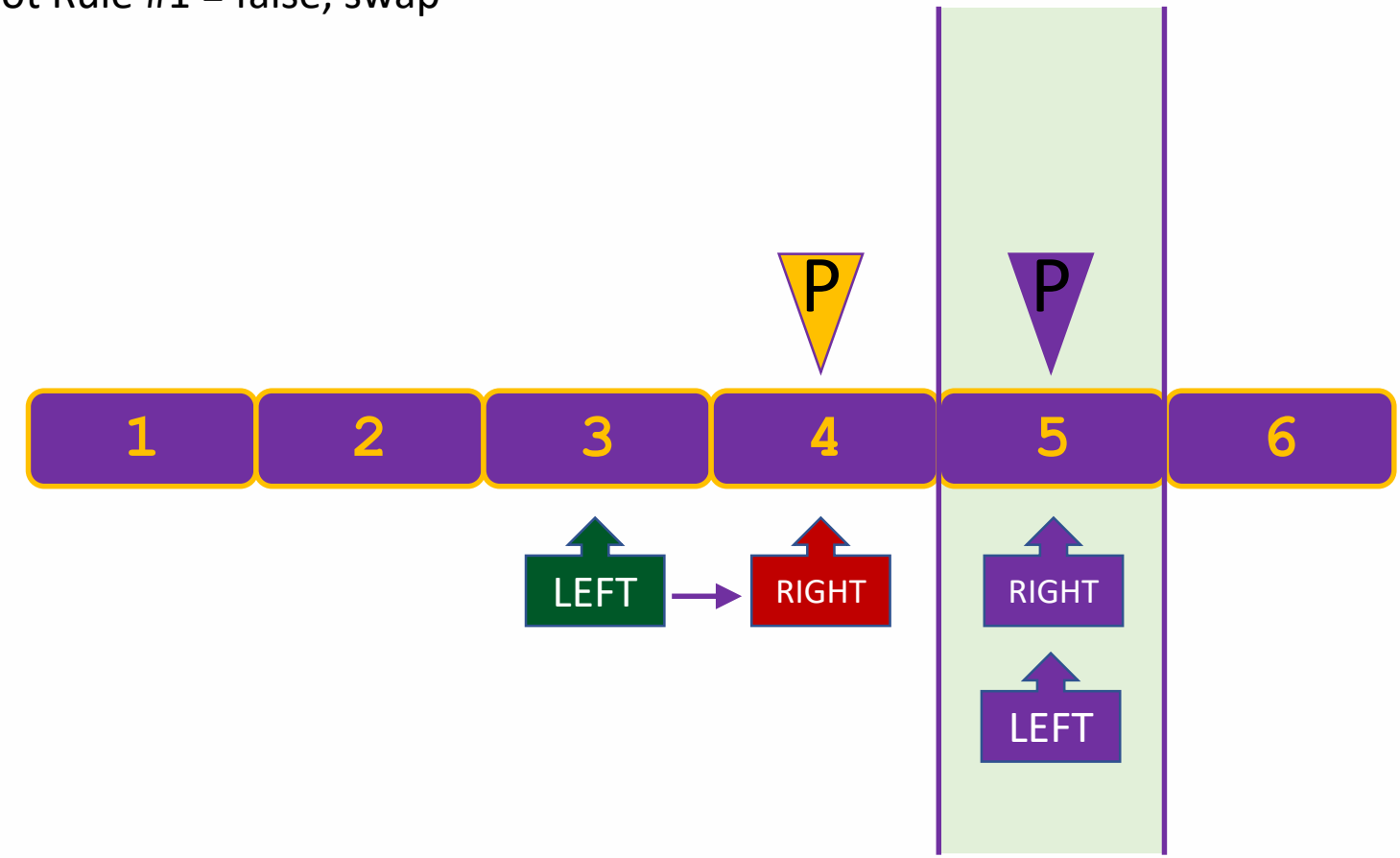
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



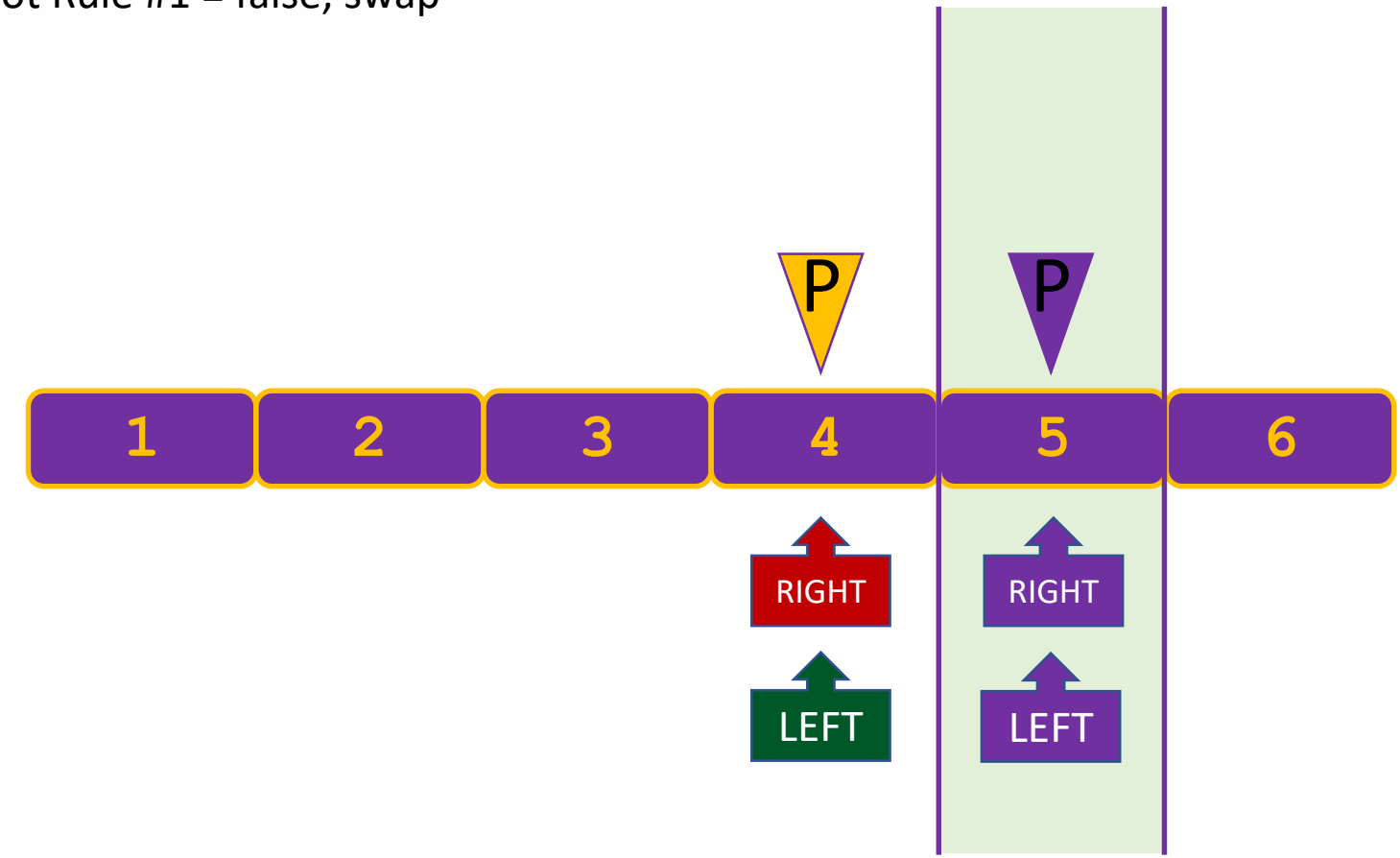
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



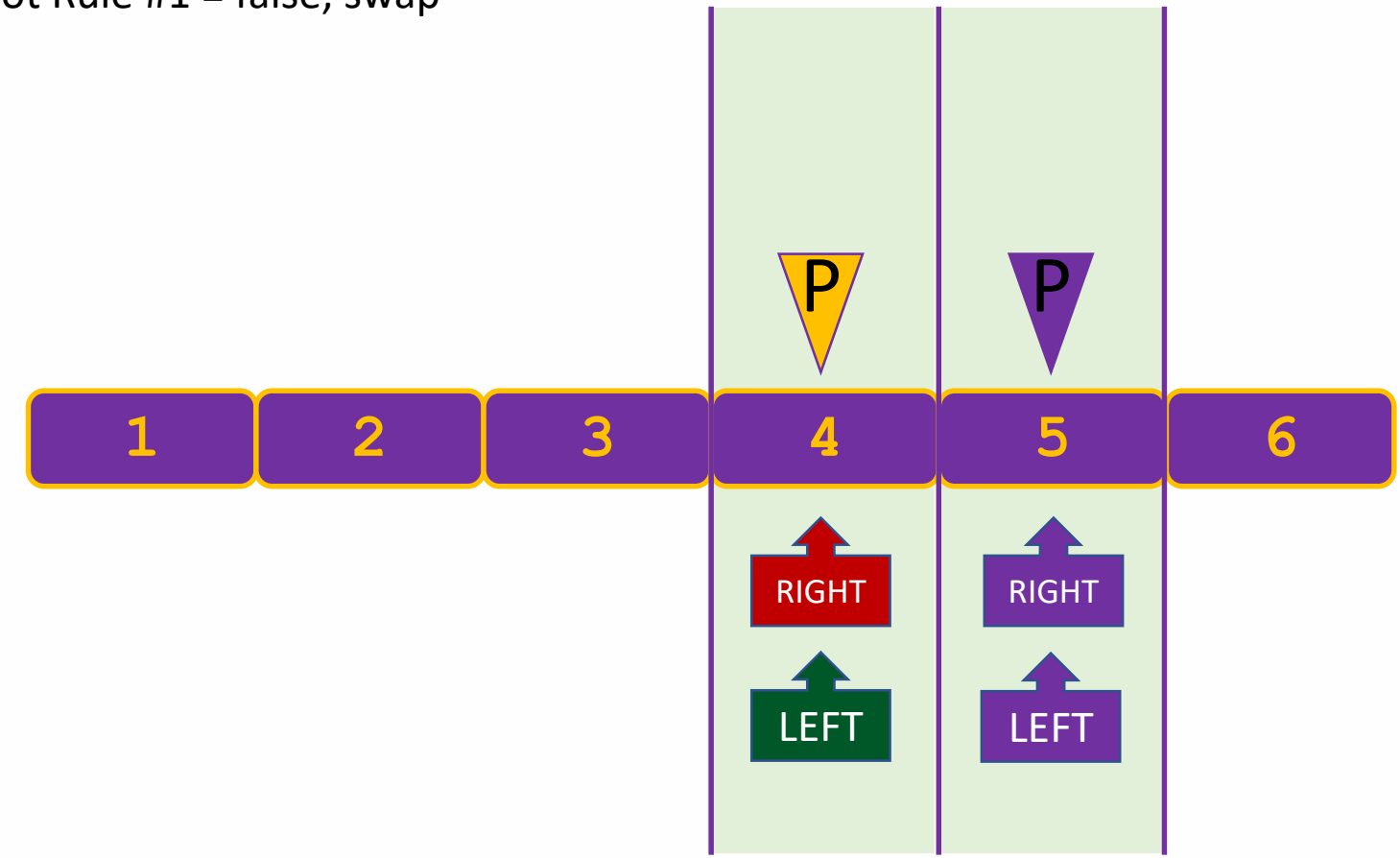
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



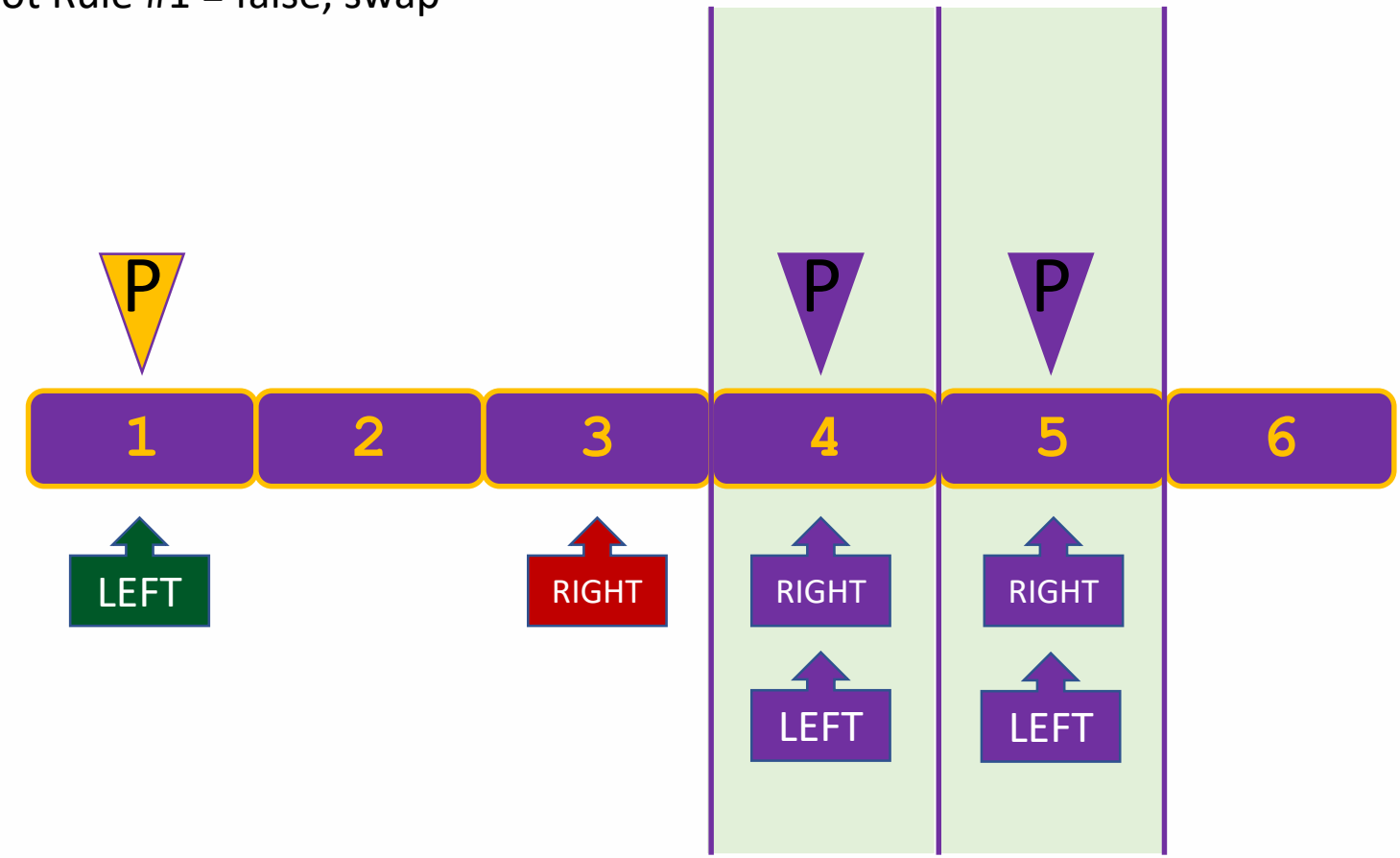
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



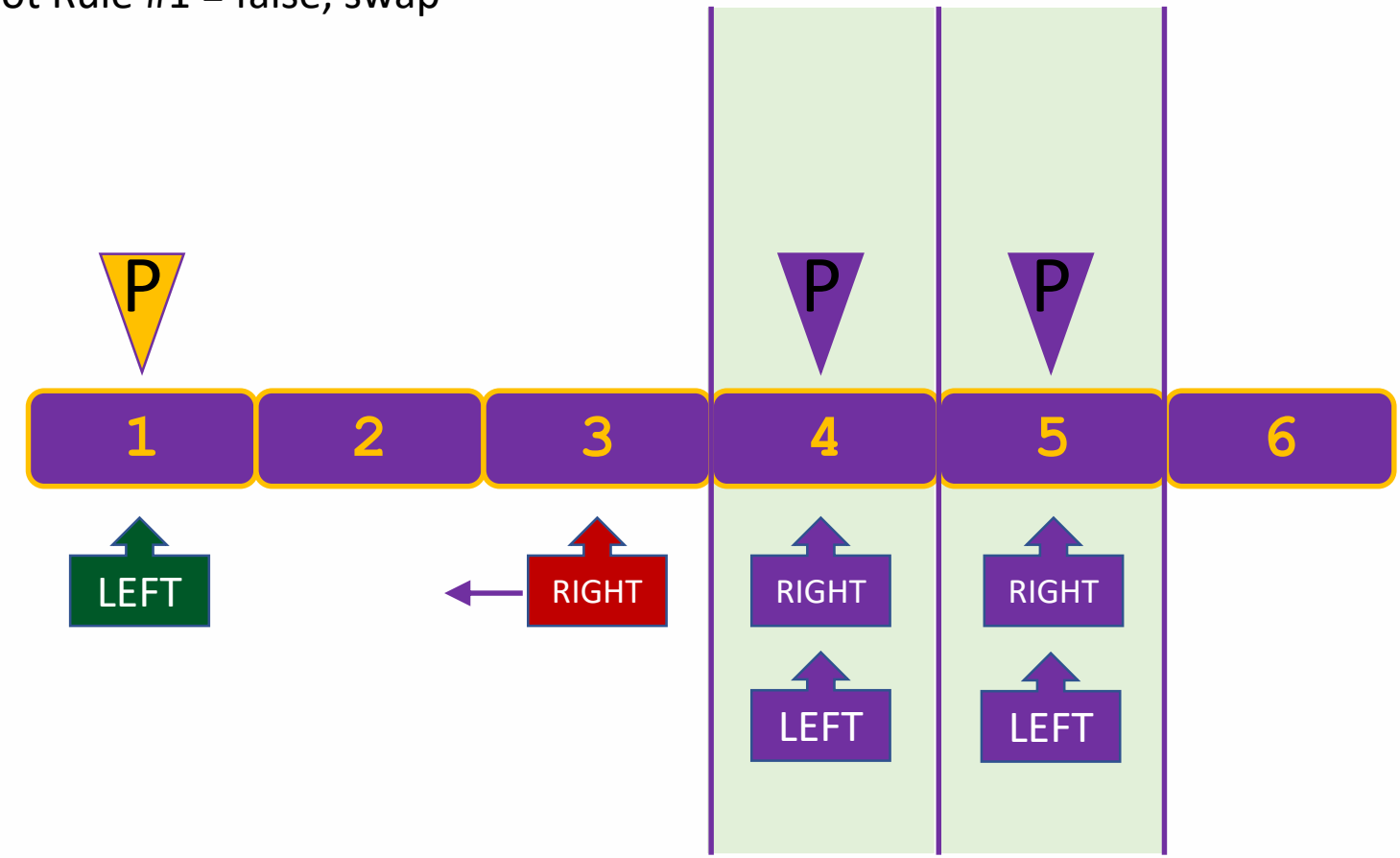
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



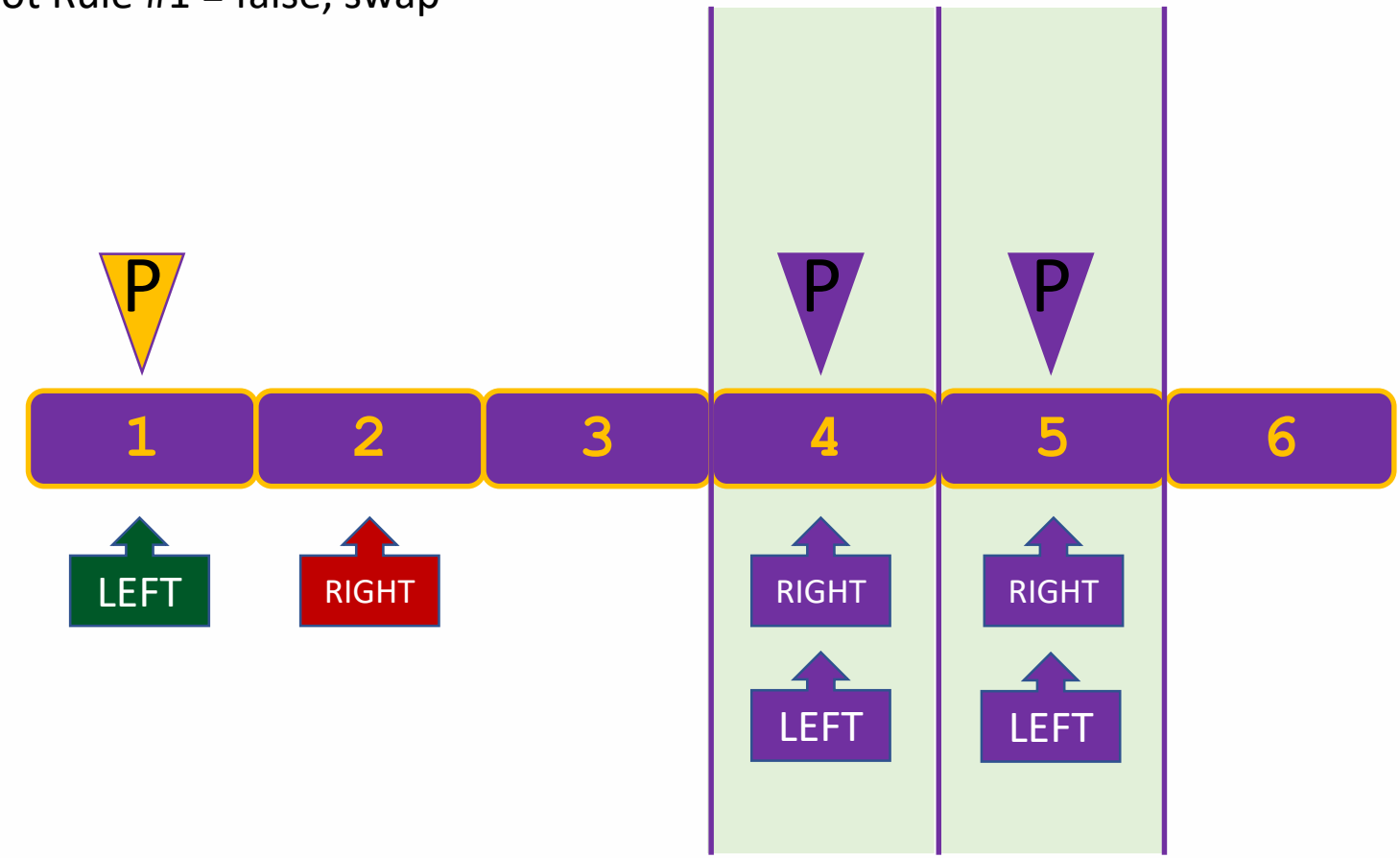
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



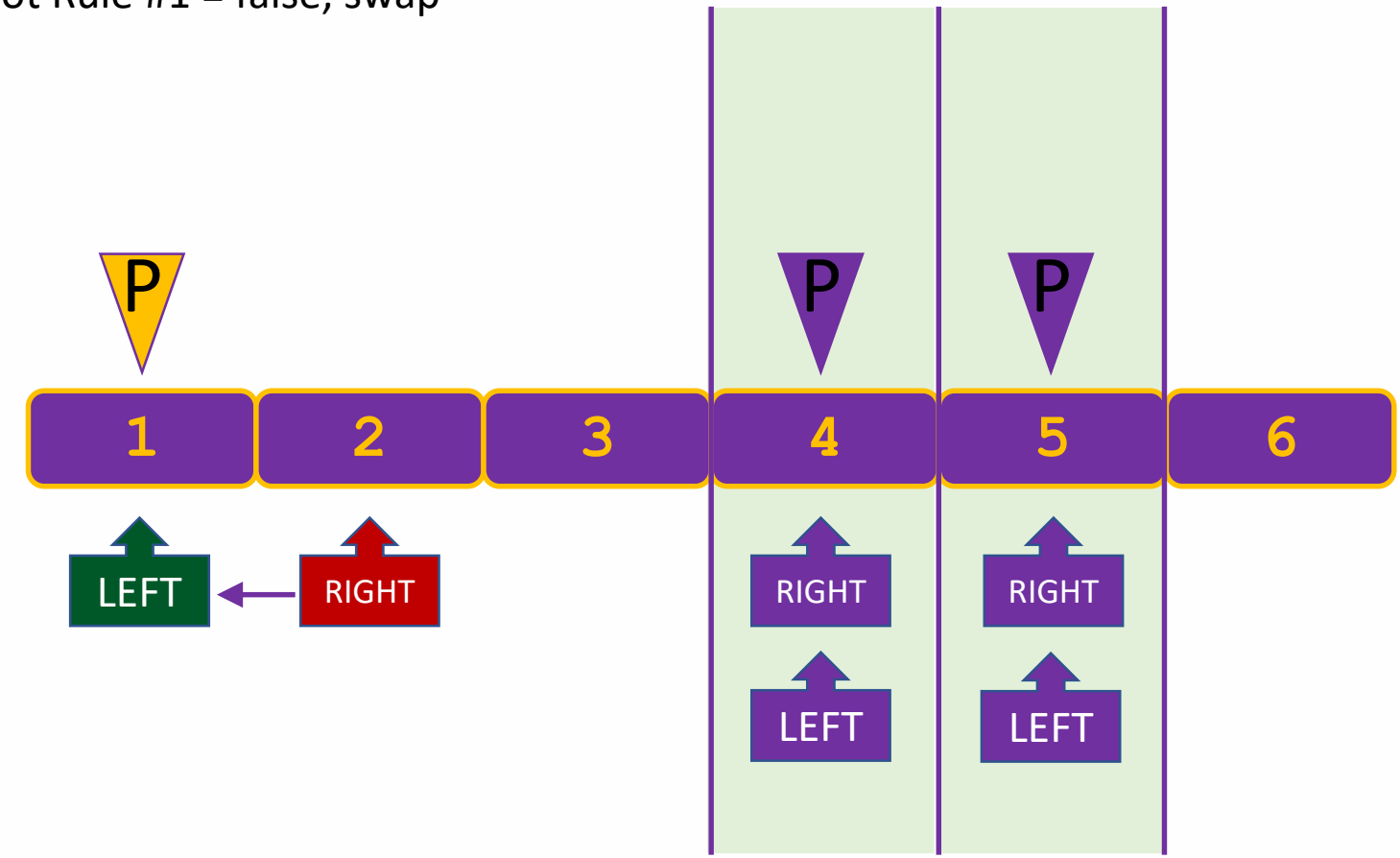
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



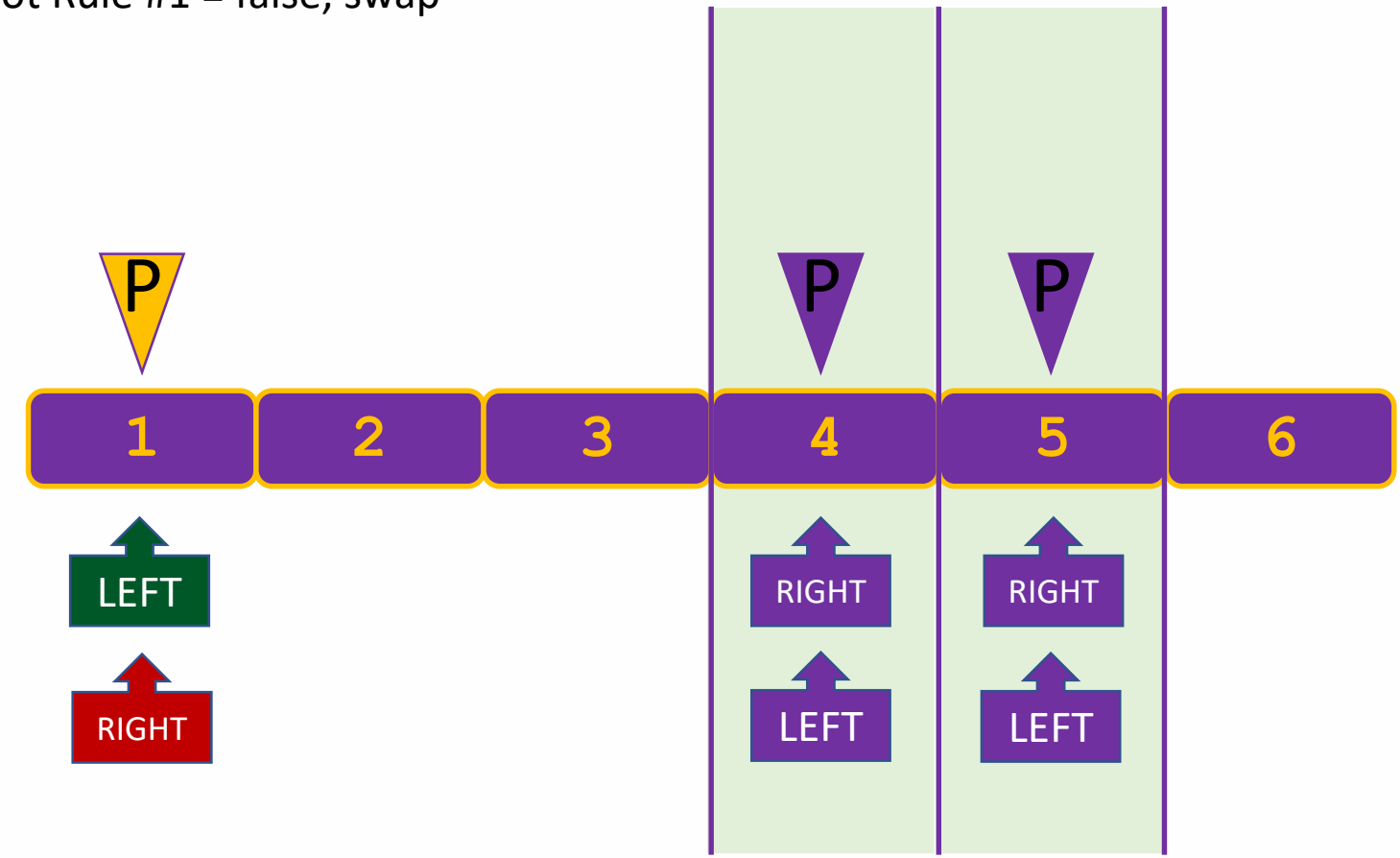
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



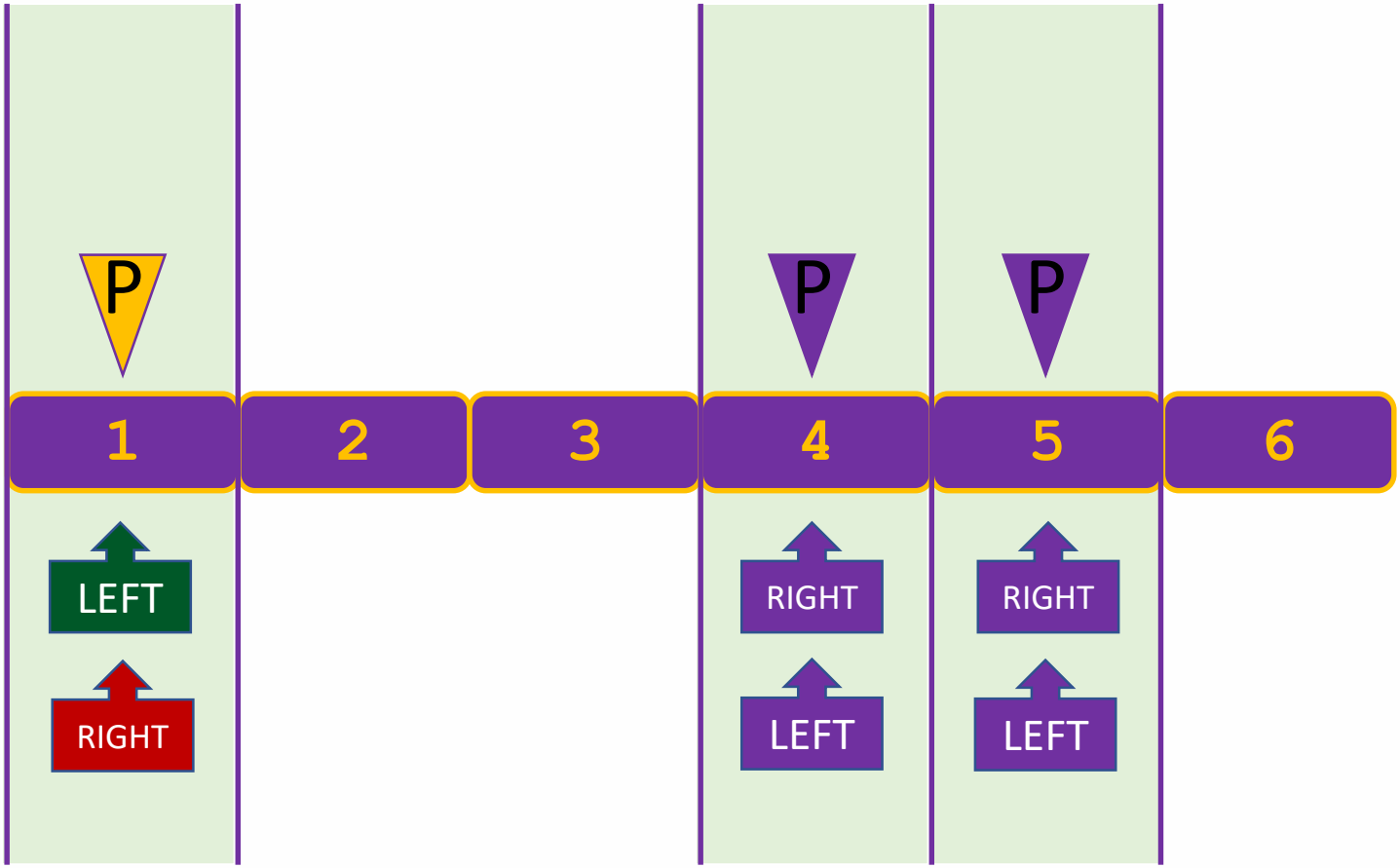
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



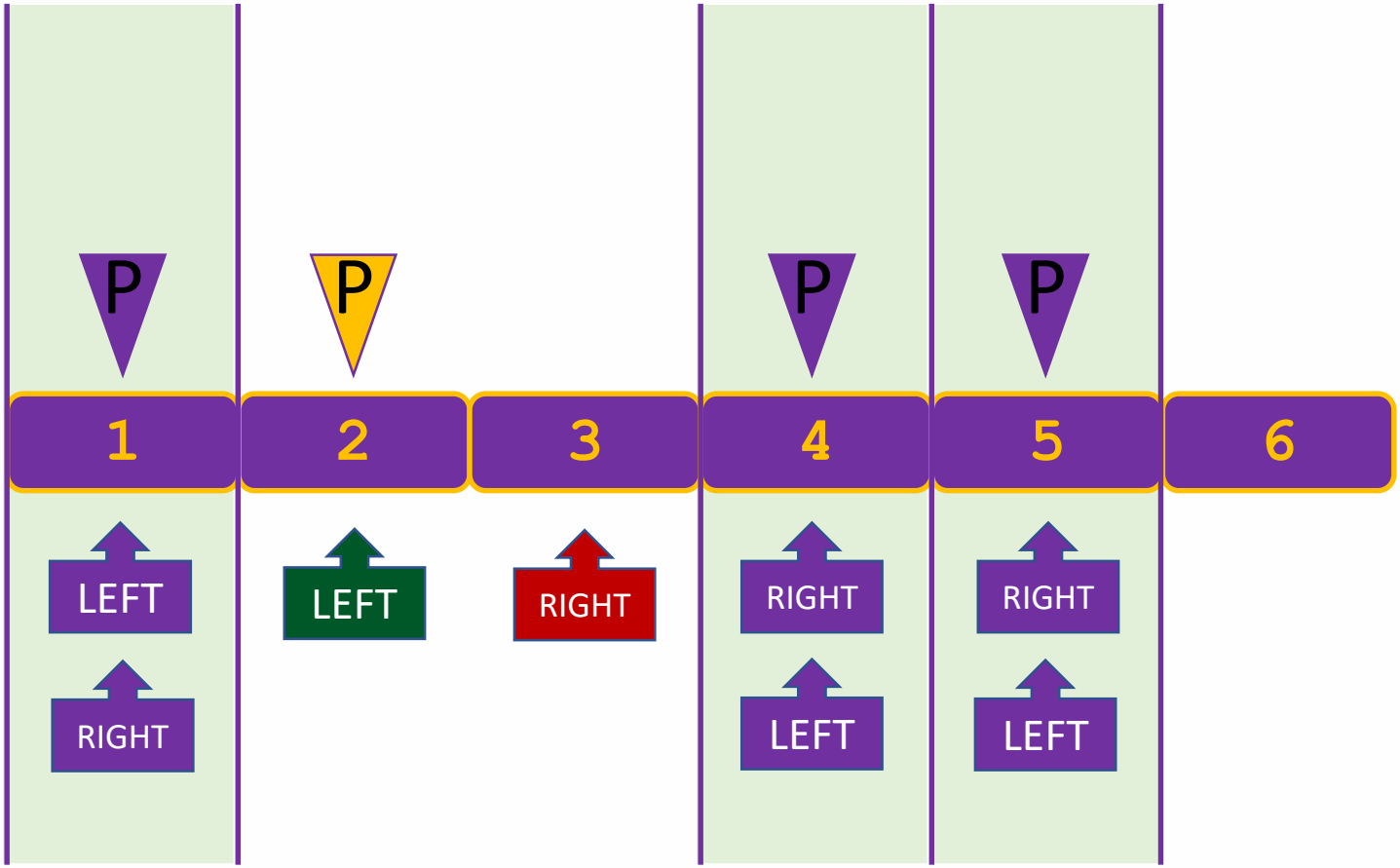
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



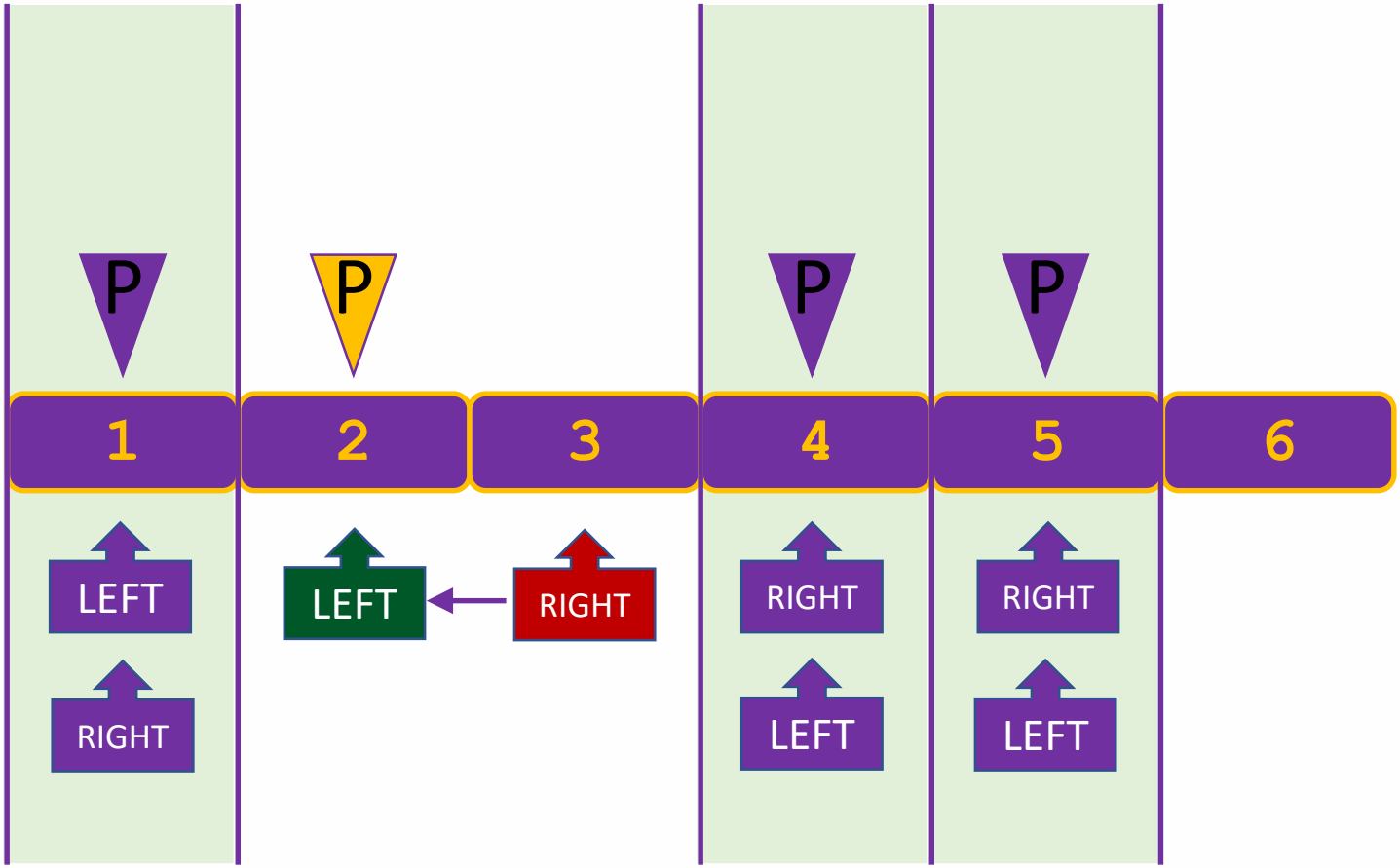
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



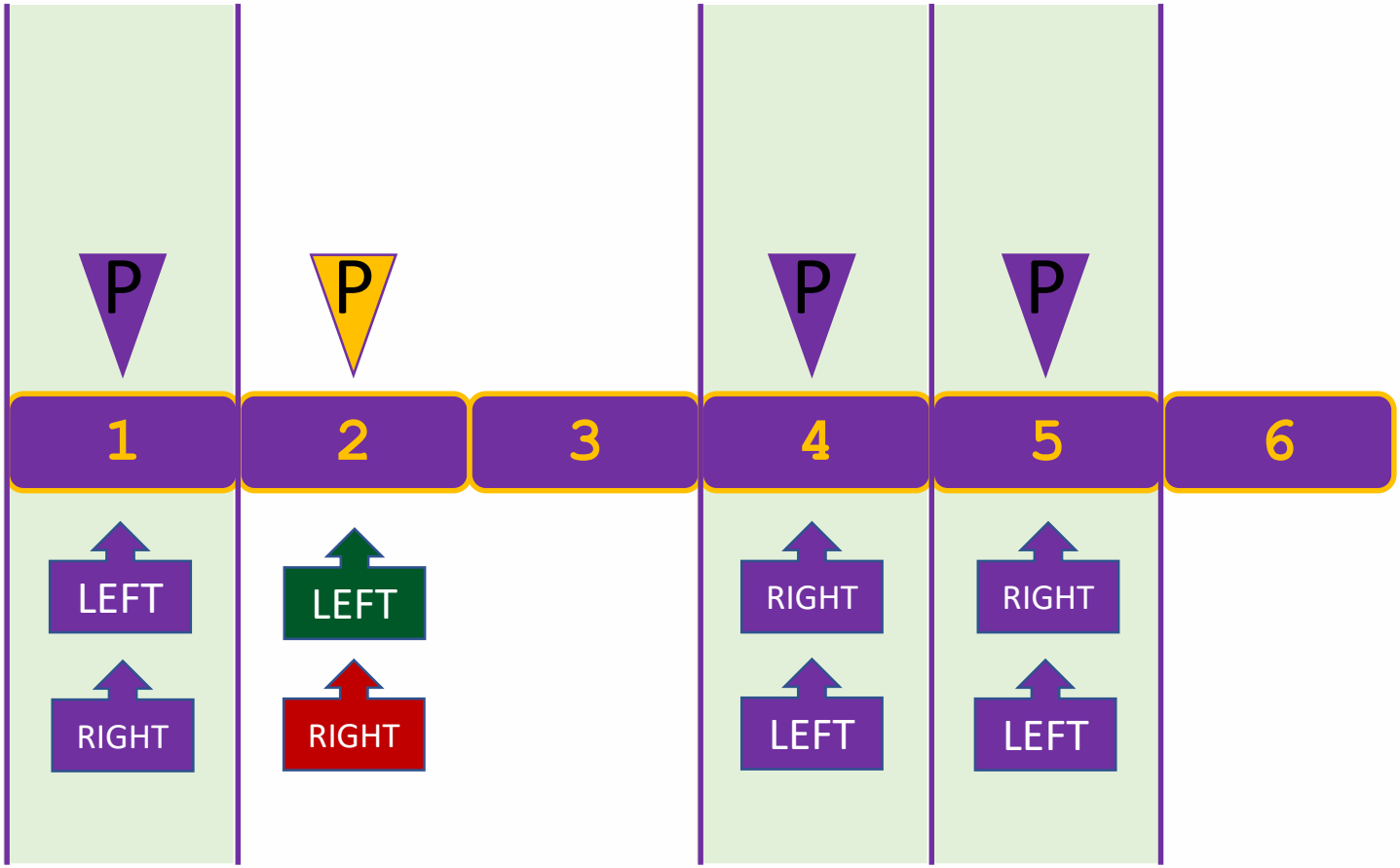
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



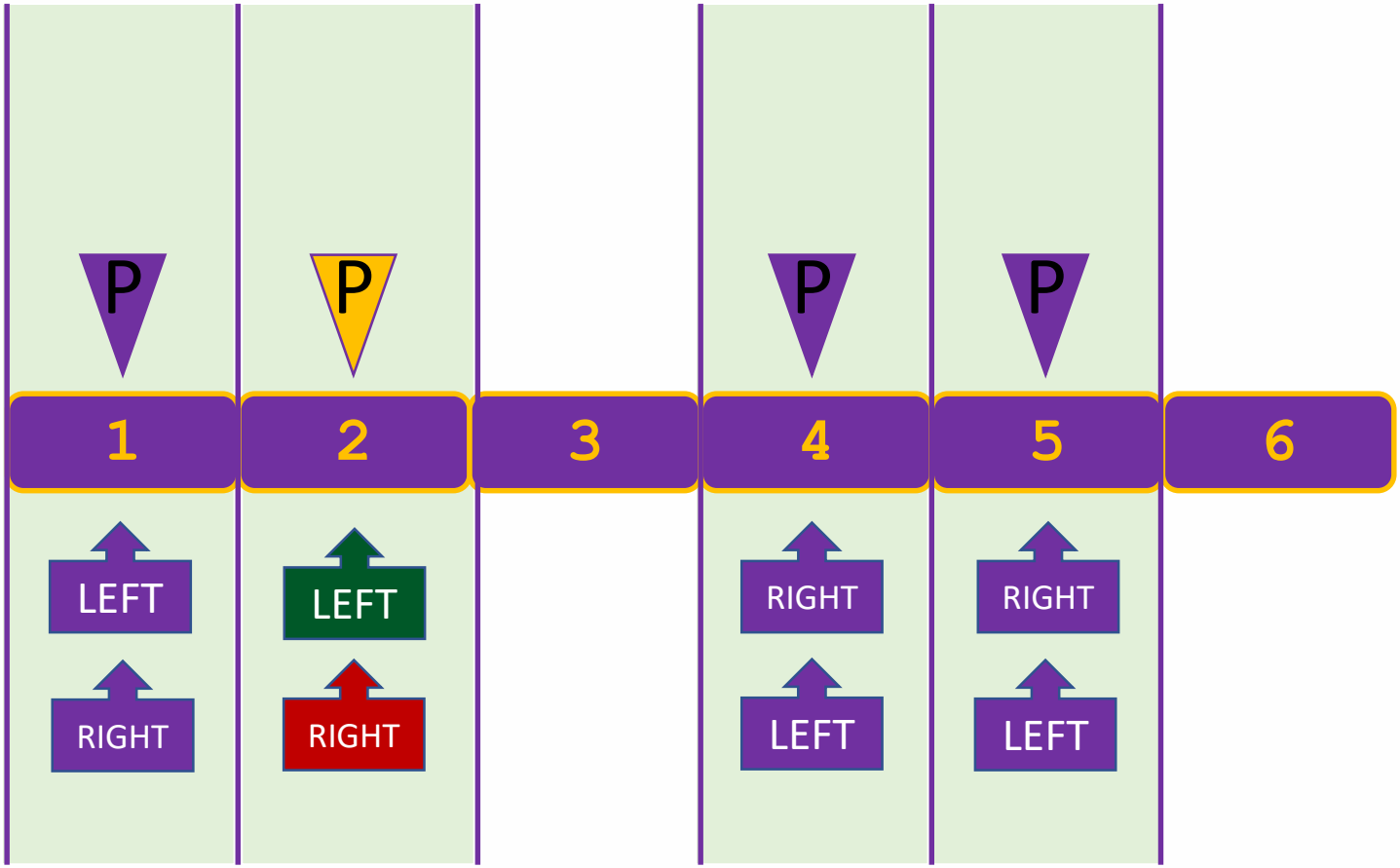
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



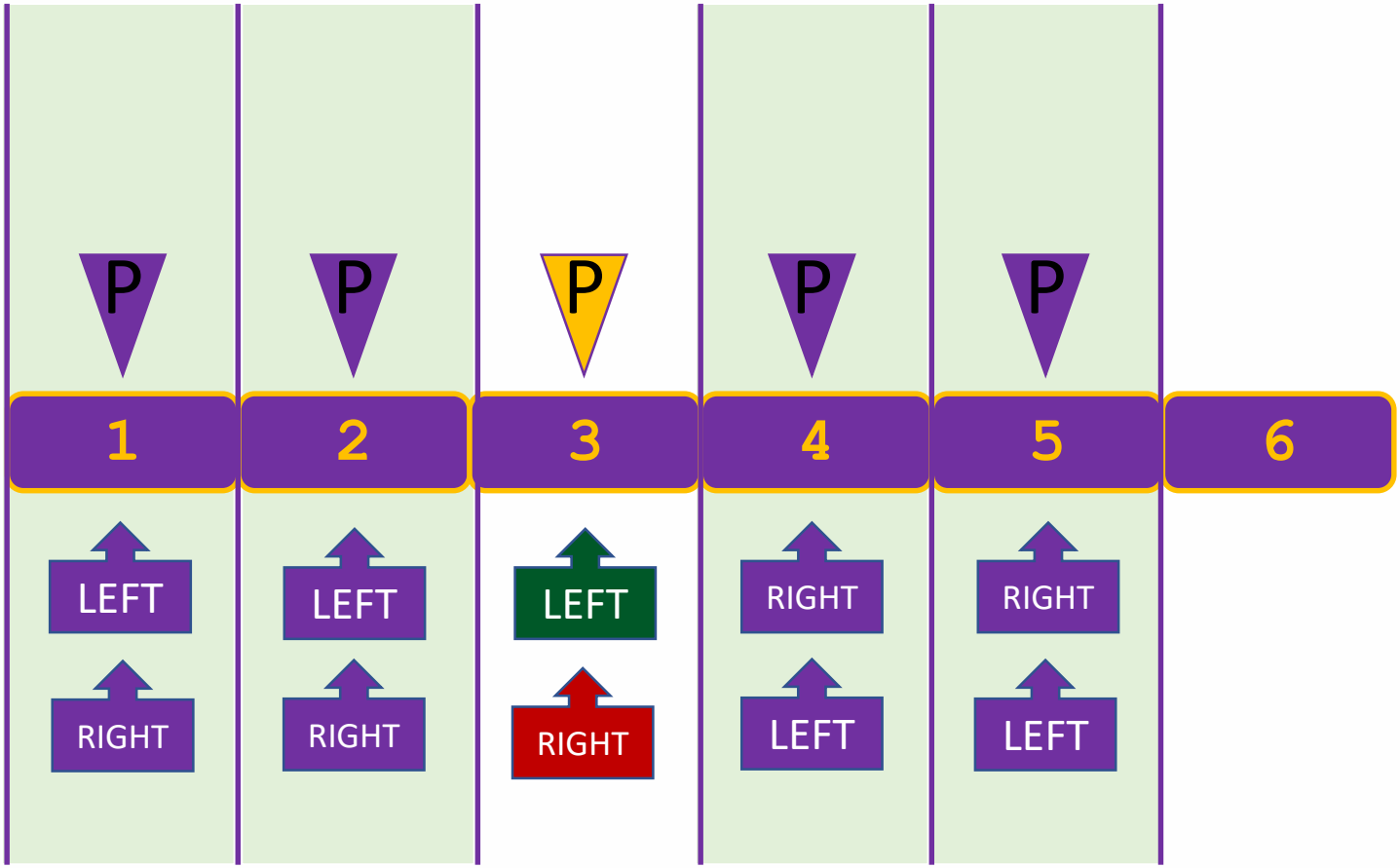
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



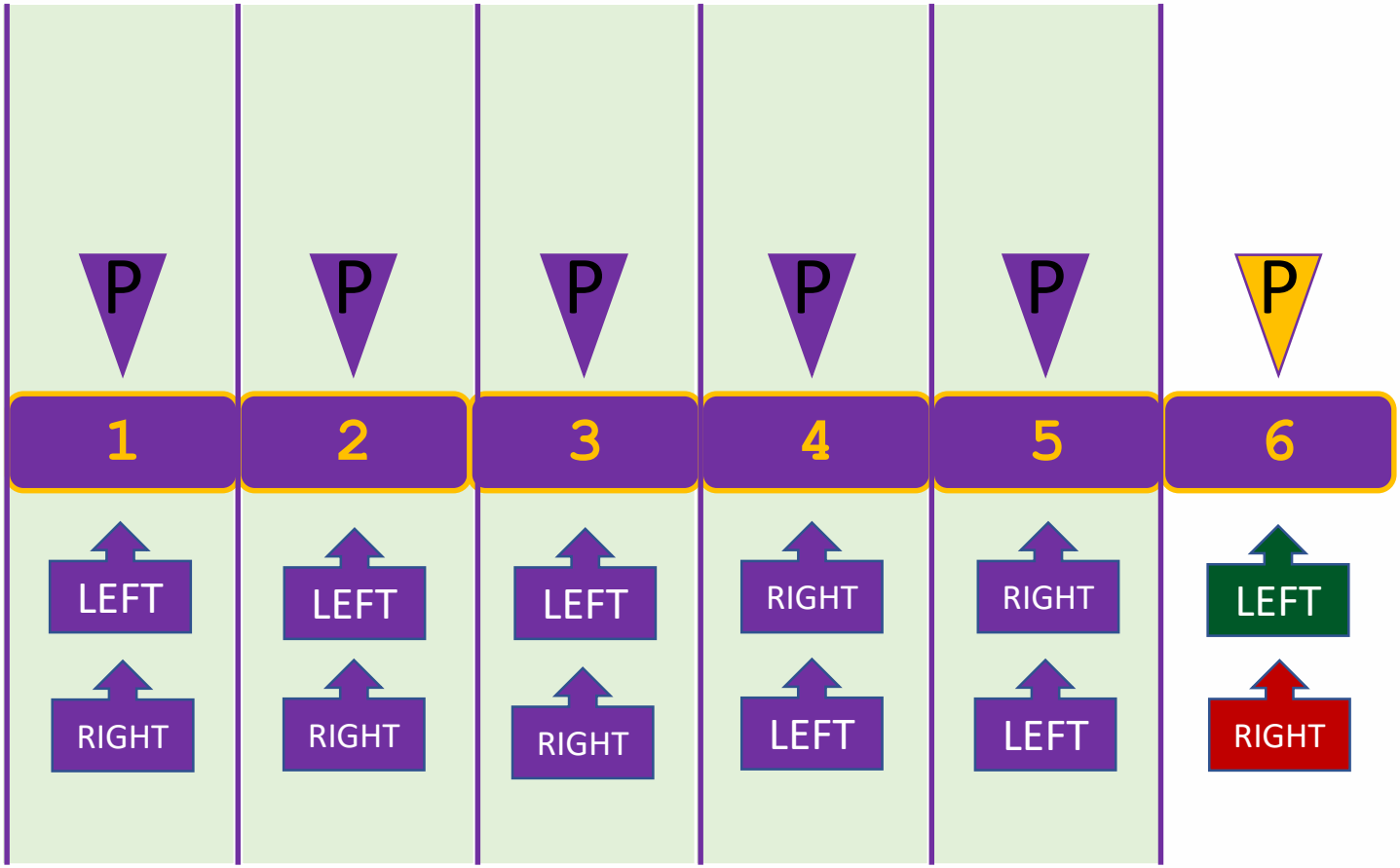
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



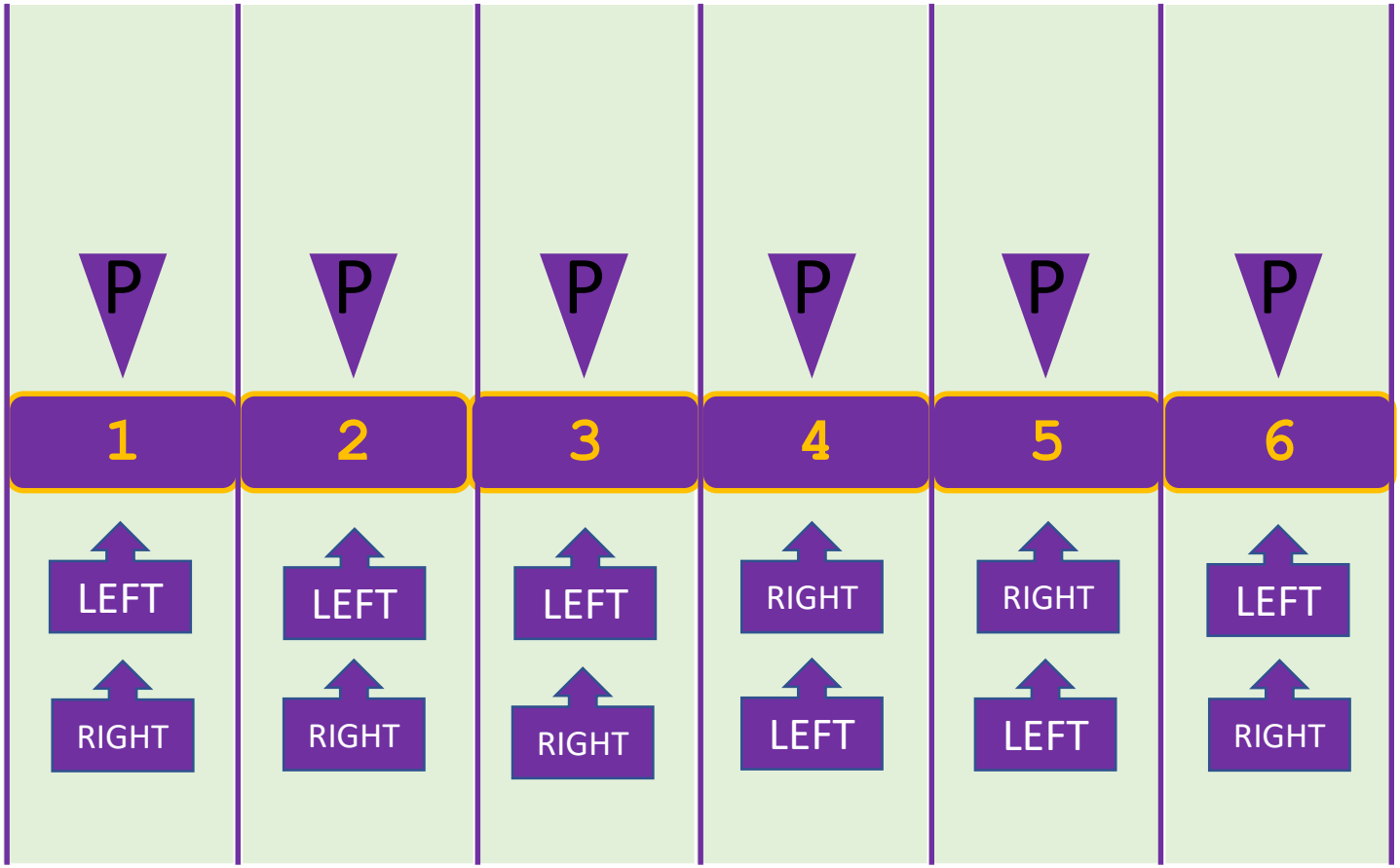
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap



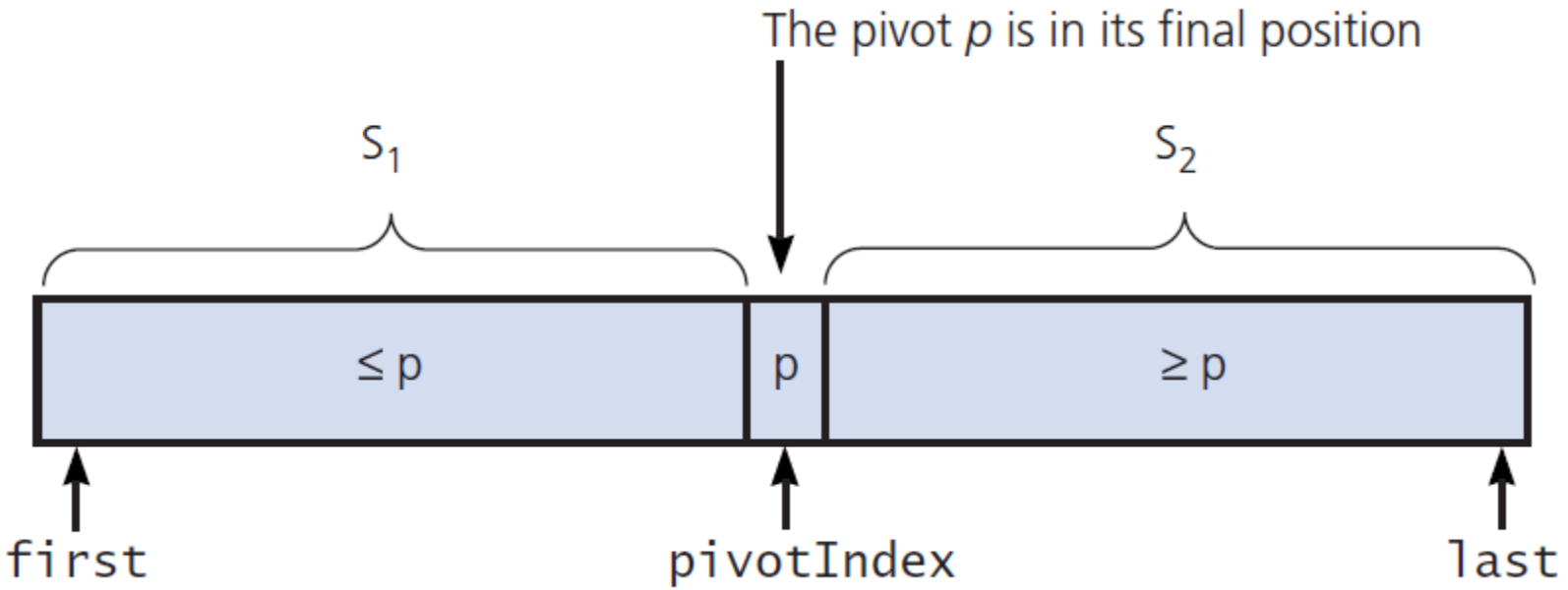
2. Quick Sort the left. THEN 3. Quick Sort the right

1. The pivot: **LEFT** < the **Pivot** < **RIGHT**
2. If not Rule #1 = false, swap

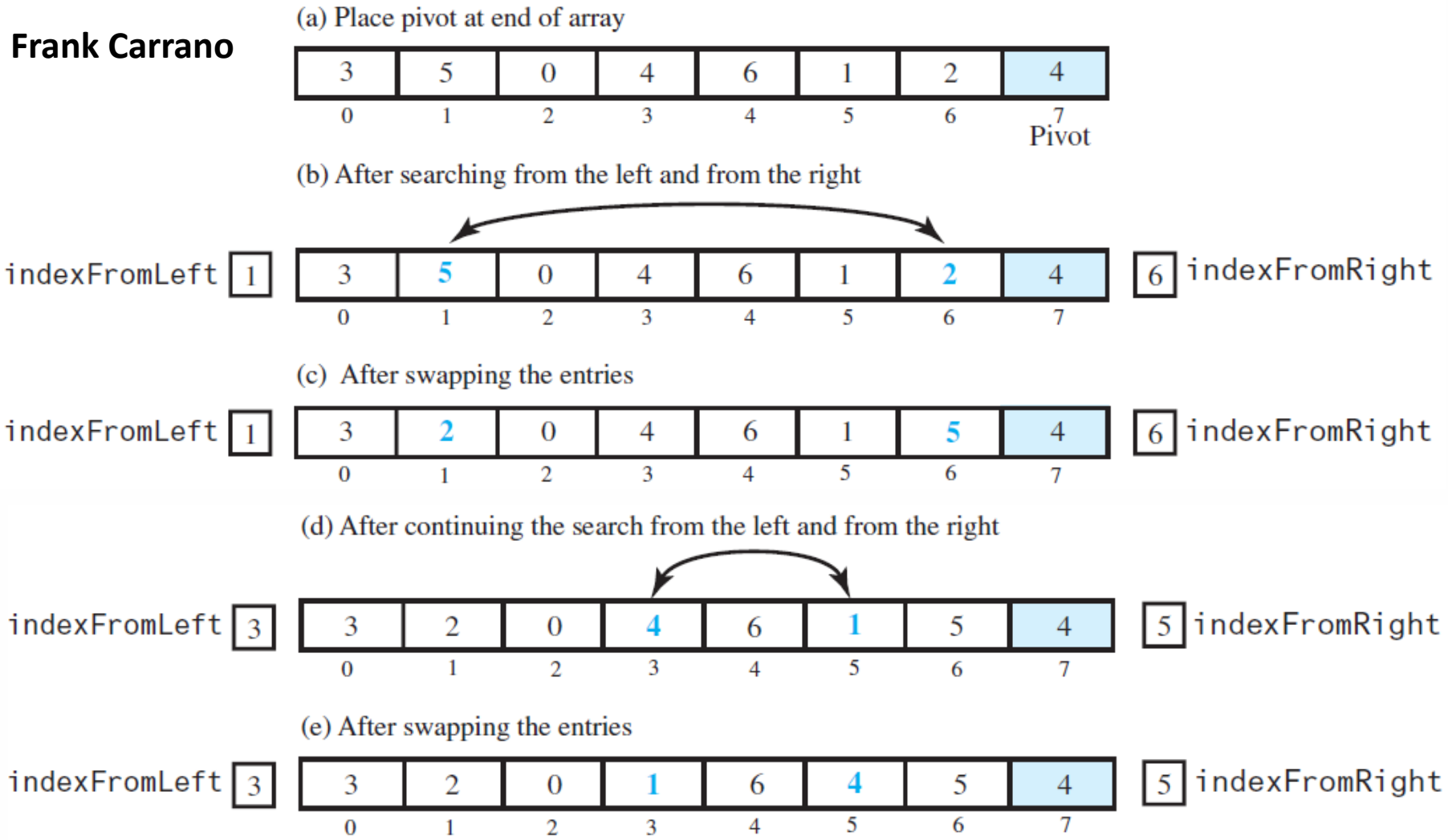


2. Quick Sort the left. THEN 3. Quick Sort the right

Frank Carrano

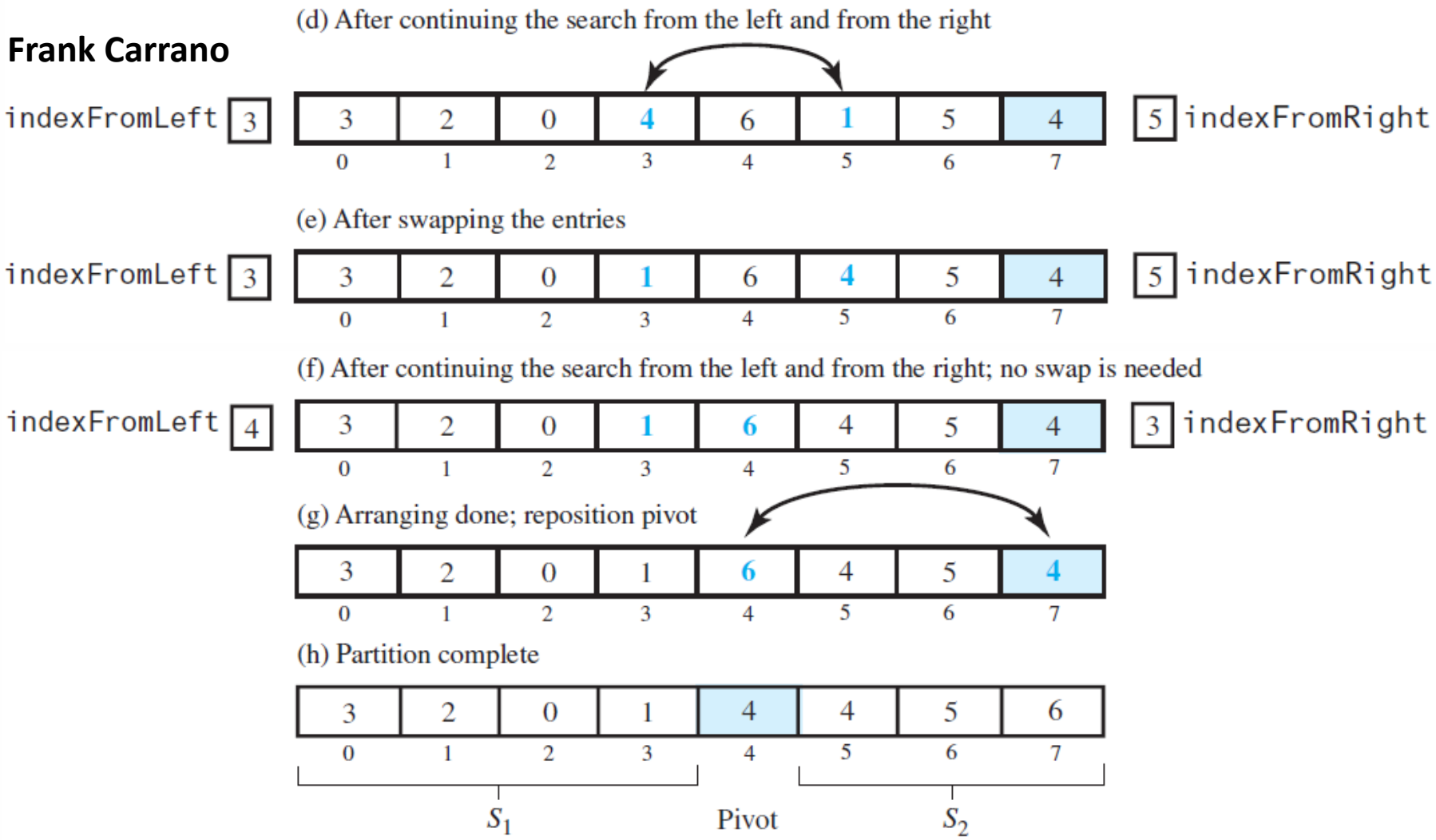


Frank Carrano



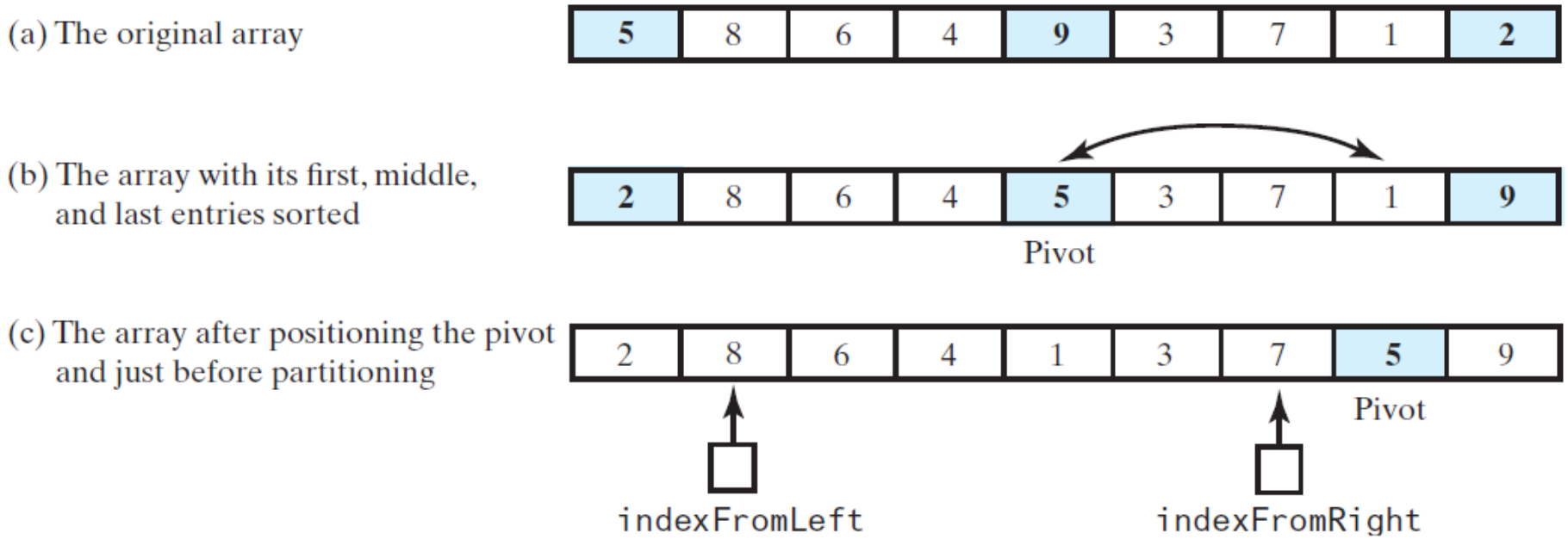
A partitioning of an array during a quick sort

Frank Carrano



A partitioning of an array during a quick sort

Frank Carrano



Median-of-three pivot selection

Quick Sort

```
68  template<class ItemType>
69  void quickSort(ItemType theArray[], int first, int last) {
70      if (last - first + 1 < MIN_SIZE) {
71          insertionSort(theArray, first, last);
72      } else {
73          int pivotIndex = partition(theArray, first, last);
74
75          quickSort(theArray, first, pivotIndex - 1);
76          quickSort(theArray, pivotIndex + 1, last);
77      }
78  }
```

QUICK SORT

```
4     static const int MIN_SIZE = 10;
5
6     template<class ItemType>
7     void insertionSort(ItemType theArray[], int first, int last)
8     {
9         for (int unsorted = first + 1; unsorted <= last; unsorted++) {
10            ItemType nextItem = theArray[unsorted];
11            int loc = unsorted;
12            while ((loc > first) && (theArray[loc - 1] > nextItem)) {
13                theArray[loc] = theArray[loc - 1];
14                loc--;
15            }
16            theArray[loc] = nextItem;
17        }
18    }
19    template<class ItemType>
20    void order(ItemType theArray[], int i, int j) {
21        if (theArray[i] > theArray[j])
22            std::swap(theArray[i], theArray[j]);
23    }
24
```

Quick Sort

```
25    template<class ItemType>
26    int sortFirstMiddleLast(ItemType theArray[], int first, int last) {
27        int mid = first + (last - first) / 2;
28        order(theArray, first, mid);
29        order(theArray, mid, last);
30        order(theArray, first, mid);
31
32        return mid;
33    }
34
```

Quick Sort

```
35  template<class ItemType>
36  int partition(ItemType theArray[], int first, int last) {
37      int pivotIndex = sortFirstMiddleLast(theArray, first, last);
38      std::swap(theArray[pivotIndex], theArray[last - 1]);
39      pivotIndex = last - 1;
40      ItemType pivot = theArray[pivotIndex];
41
42      int indexFromLeft = first + 1;
43      int indexFromRight = last - 2;
44
45      bool done = false;
46      while (!done) {
47          while (theArray[indexFromLeft] < pivot)
48              indexFromLeft = indexFromLeft + 1;
49
50          while (theArray[indexFromRight] > pivot)
51              indexFromRight = indexFromRight - 1;
52
53          if (indexFromLeft < indexFromRight) {
54              std::swap(theArray[indexFromLeft], theArray[indexFromRight]);
55              indexFromLeft = indexFromLeft + 1;
56              indexFromRight = indexFromRight - 1;
57          }
58          else
59              done = true;
60      }
61
62      std::swap(theArray[pivotIndex], theArray[indexFromLeft]);
63      pivotIndex = indexFromLeft;
64
65      return pivotIndex;
66  }
```

Worst Case of Quick Sort

It depends on strategy for choosing pivot. **In early versions** of Quick Sort where leftmost (or rightmost) element is chosen as pivot, the worst occurs in following cases.

1) Array is already sorted in same order.
2) Array is already sorted in reverse order.
3) All elements are same (special case of case 1 and 2)

Since these cases are very common use cases, the problem was easily solved by choosing either **a random index for the pivot**, choosing **the middle index of the partition** or (especially for longer partitions) choosing **the median of the first, middle and last element** of the partition for the pivot. With these modifications, the worst case of Quick sort has less chances to occur, but worst case can still occur if the input array is such that the maximum (or minimum) element is always chosen as pivot.

Compare Sorts

Compare Sorts

- Selection sort, insertion sort, and bubble sort use different techniques, but are all $O(n^2)$
- They are all based in a nested loop approach
- In quick sort, if the partition element divides the elements in half, each recursive call operates on about half the data
- The act of partitioning the elements at each level is O(n)
- The effort to sort the entire list is O(n log n)
- It could deteriorate to $O(n^2)$ if the partition element is poorly chosen

Compare Sorts

- Merge sort divides the list repeatedly in half, which results in the O(log n) portion
- The act of merging is O(n)
- So the efficiency of merge sort is O(n log n)

- Selection, insertion, and bubble sorts are called *quadratic sorts*
- Quick sort and merge sort are called *logarithmic sorts*

(Frank Carrano)

- The selection sort, bubble sort, and insertion sort are all O(n2) algorithms. Although in a particular case one might be faster than another, for large problems they all are slow. For small arrays, however, the insertion sort is a good choice.
- The quick sort and merge sort are two very efficient recursive sorting algorithms. In the average case, the quick sort is among the fastest known sorting algorithms. However, the quick sort's worst-case behavior is significantly slower than the merge sort's. Fortunately, the quick sort's worst case rarely occurs in practice. The merge sort is not quite as fast as the quick sort in the average case, but its performance is consistently good in all cases. The merge sort has the disadvantage of requiring extra storage equal to the size of the array to be sorted.
- The radix sort is unusual in that it does not sort the array entries by comparing them. Thus, it is not always applicable, making it inappropriate as a general-purpose sorting algorithm. However, when the radix sort is applicable, it is on O(n) algorithm.

(Frank Carrano)

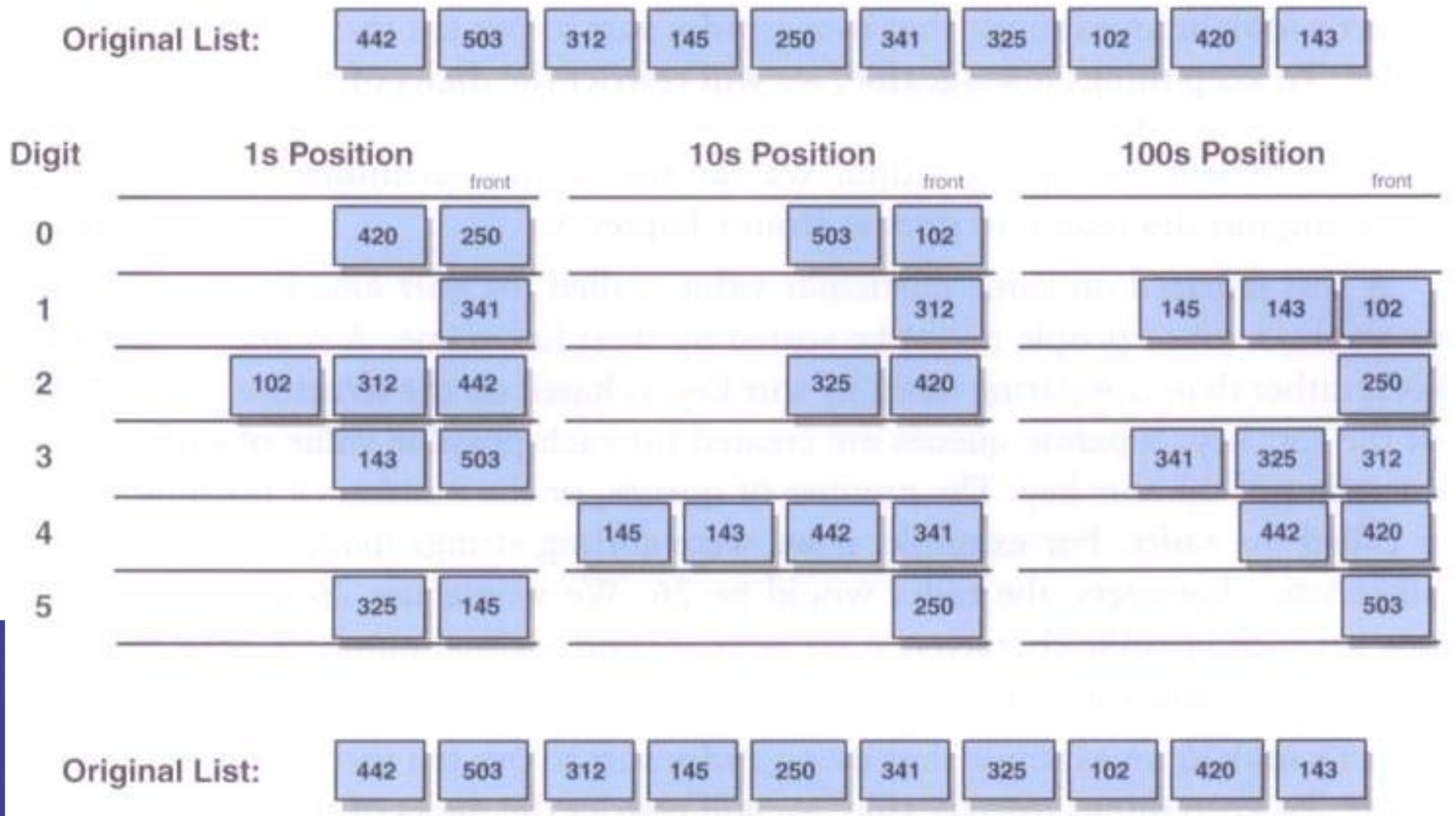
- The quick sort rearranges the entries in an array during the partitioning process. Each partition places one entry – the pivot – in its correct sorted position. The entries in each of the two subarrays that are before and after the pivot will remain in their respective subarrays.
- The choice of pivots affects the quick sort's efficiency. Some pivot-selection schemes can lead to worst-case behavior if the array is already sorted or nearly sorted. In practice, nearly sorted arrays can occur more frequently than you might imagine. Fortunately, median-of-three pivot selection avoids worst-case behavior for sorted arrays.
- The quick sort is often used to sort large arrays, as it is usually extremely fast in practice, despite its unimpressive theoretical worst-case behavior. Although a worst-case situation is not typical, even if the worst case occurs, the quick sort's performance is acceptable for moderately large arrays.
- The quick sort is appropriate when you are confident that the data in the array to be sorted is arranged randomly. Although the quick sort's worst-case behavior is $O(n^2)$, the worst case rarely occurs in practice.

RADIX SORT

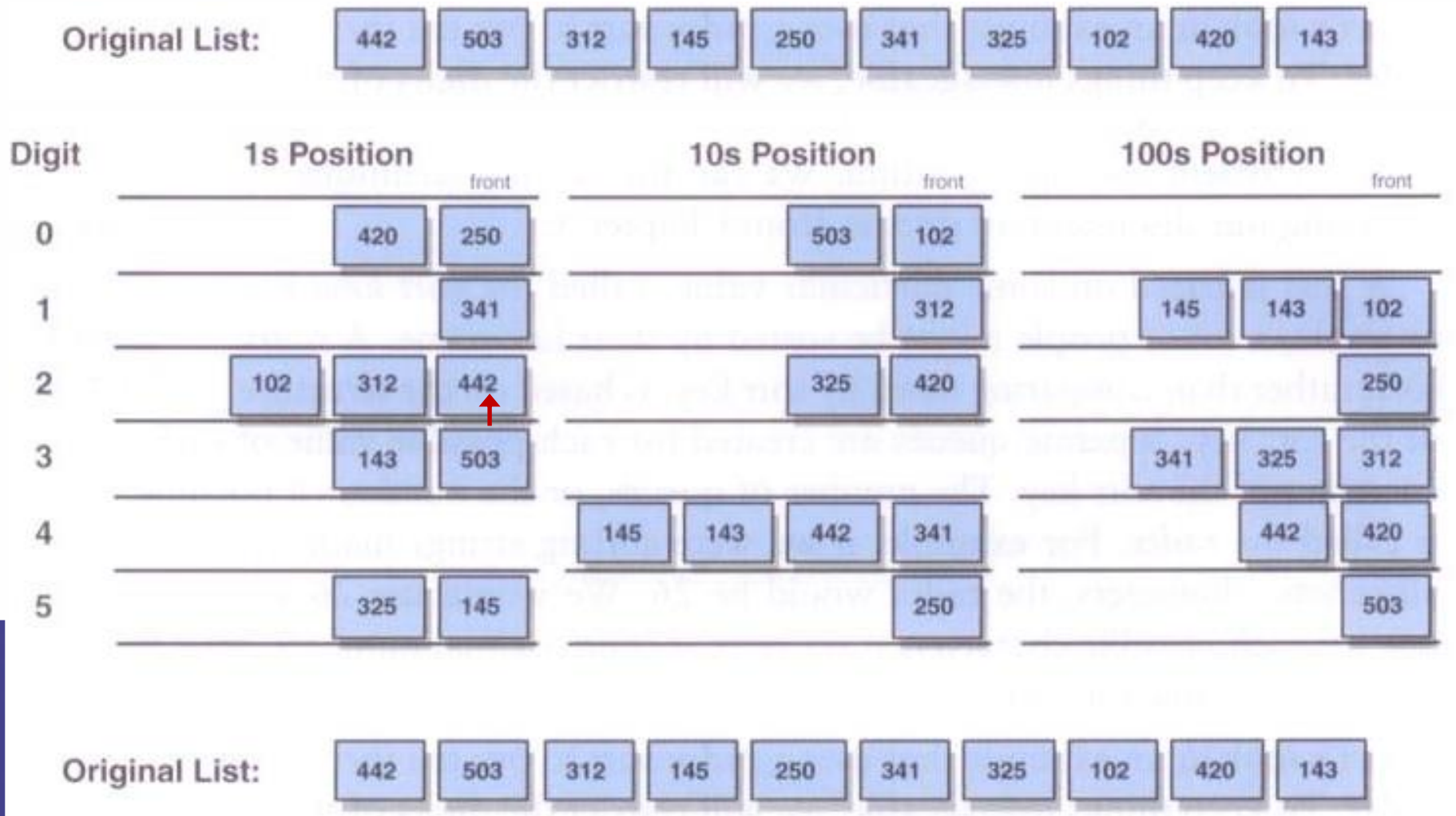

Radix Sort

- Radix Sort only works when a **sort key** can be defined.
- **Separate queues** are used to store elements based on the **structure of the sort key.**
    - For example, to sort decimal numbers, we would use ten queues, one for each possible digit (0-9)
    - To keep our example simpler, we will restrict our values to the digits 0-5
- The radix sort makes three **passes through the data**, for each position of our 3-digit numbers
- **A value** is put on the queue corresponding to that position's digit
- Once all three **passes are finished**, the data is sorted in each queue

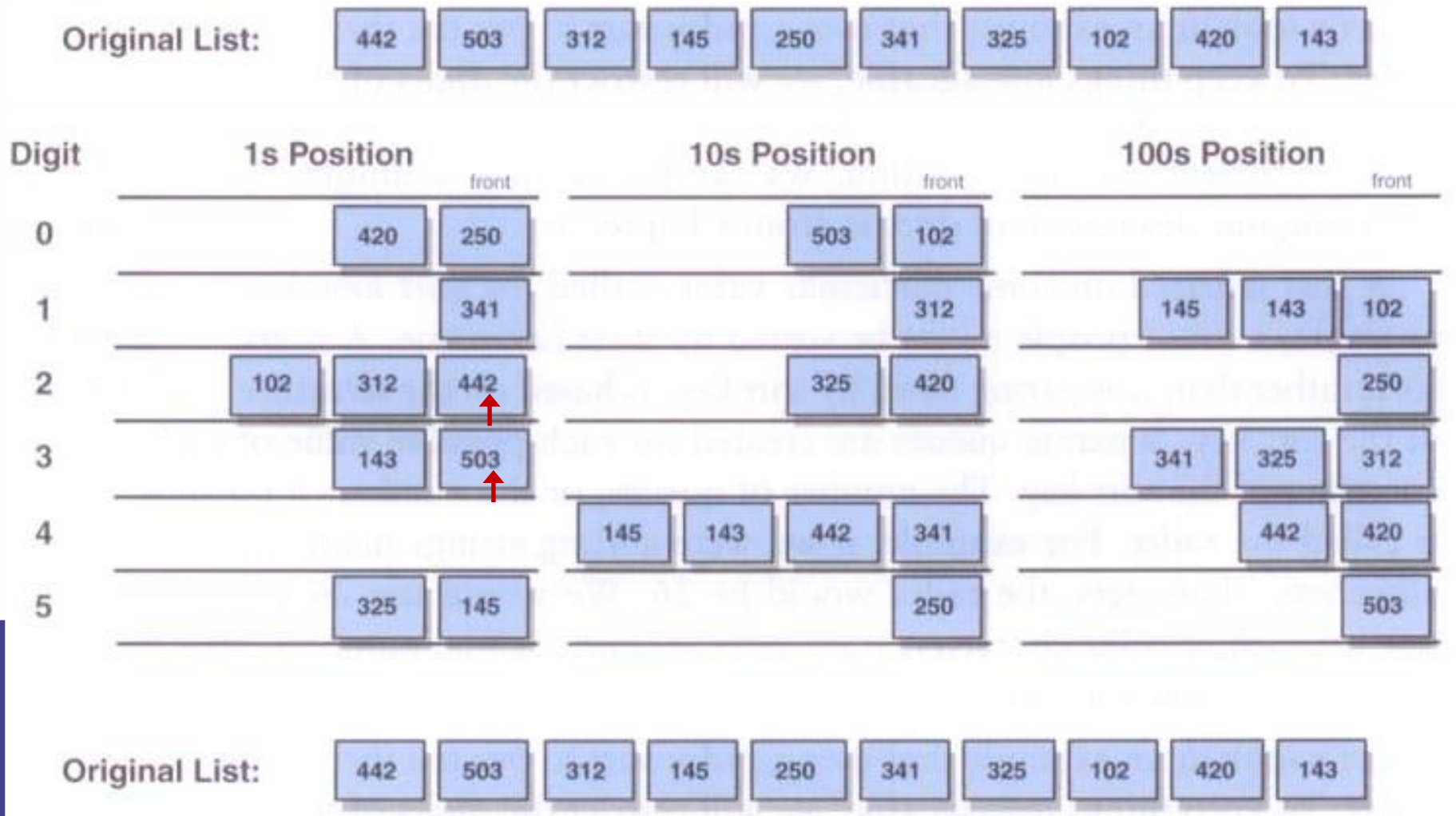
- An example using six queues to sort 10 three-digit numbers:



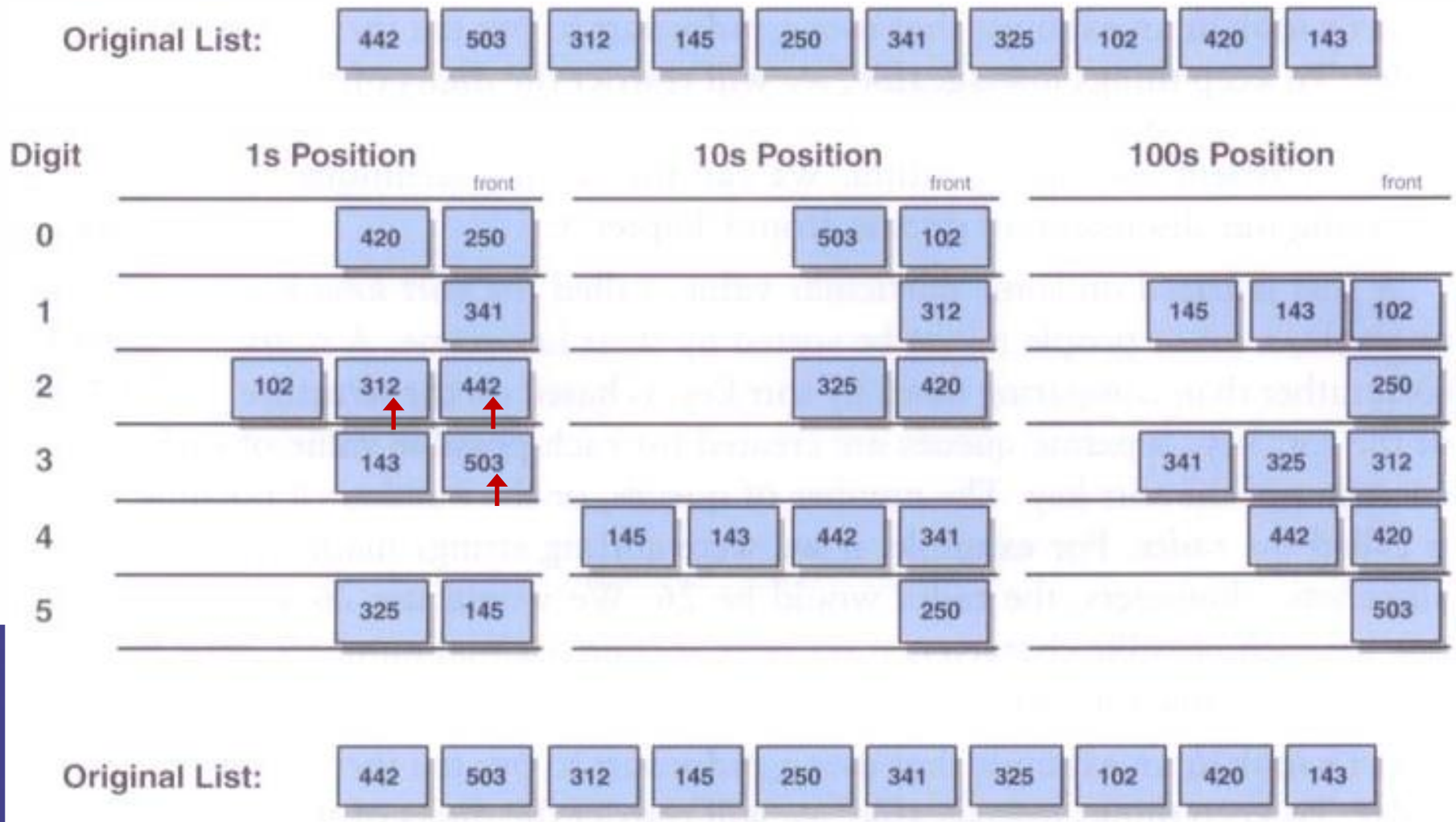
- An example using six queues to sort 10 three-digit numbers:



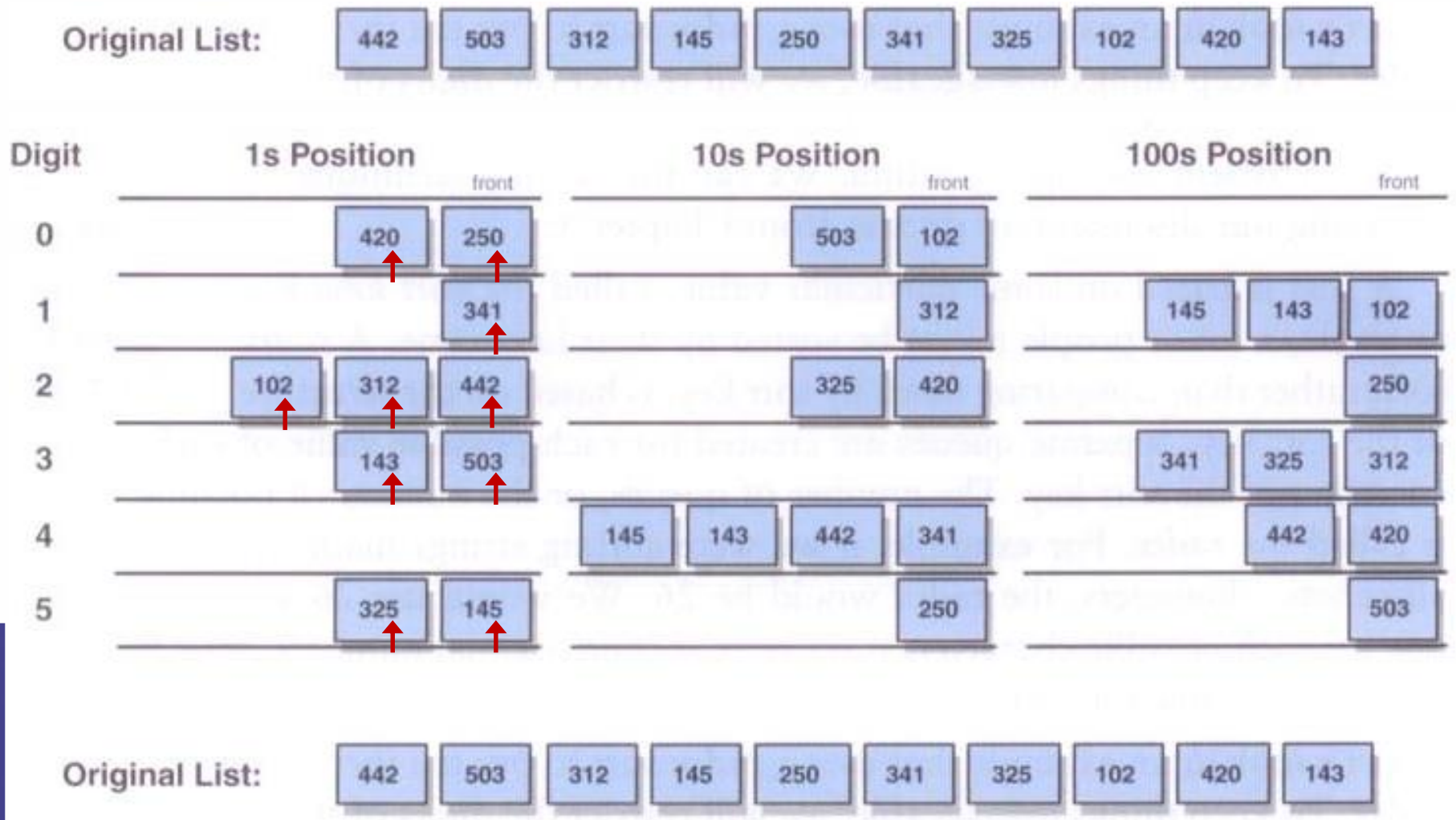
- An example using six queues to sort 10 three-digit numbers:



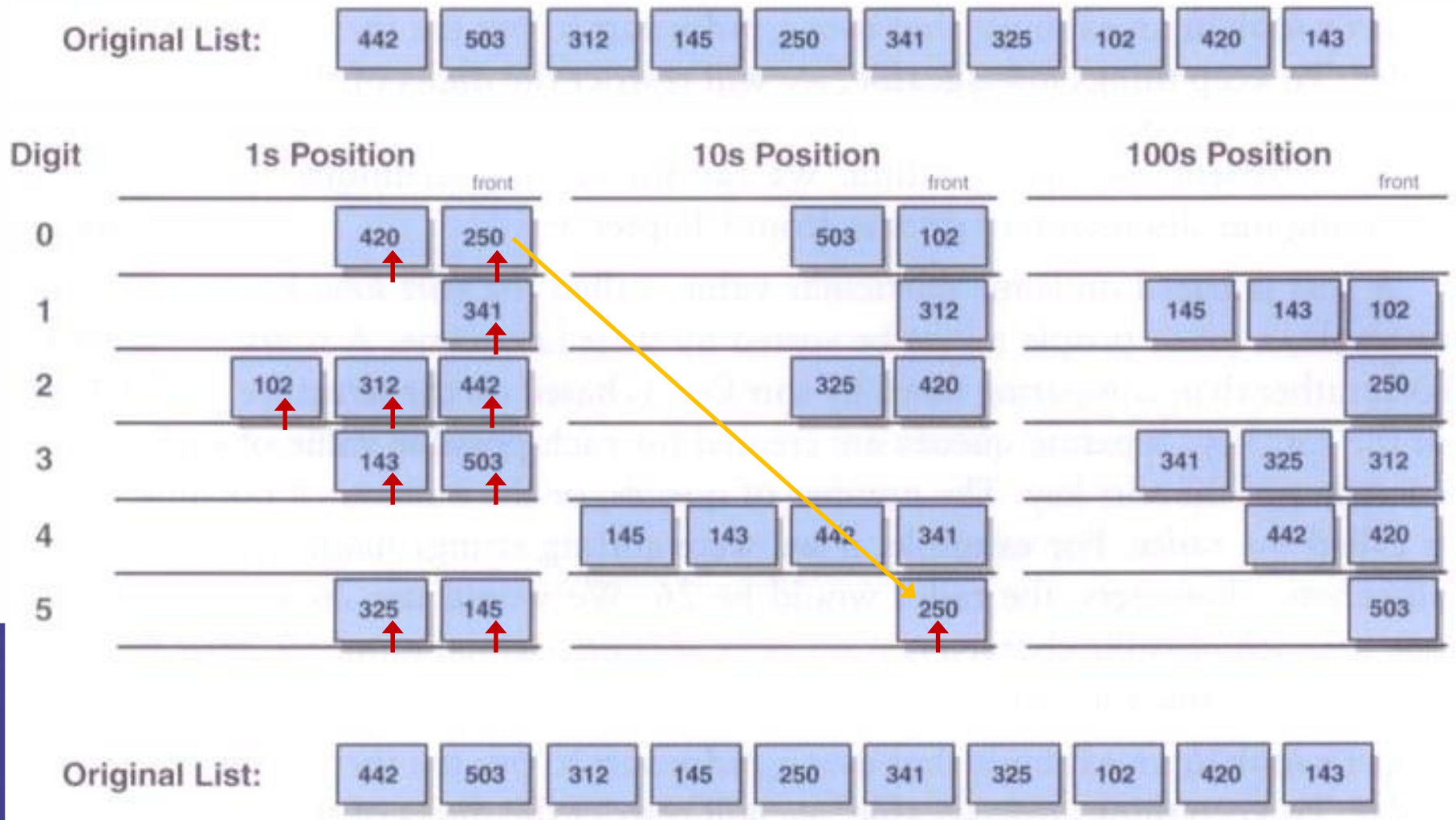
- An example using six queues to sort 10 three-digit numbers:



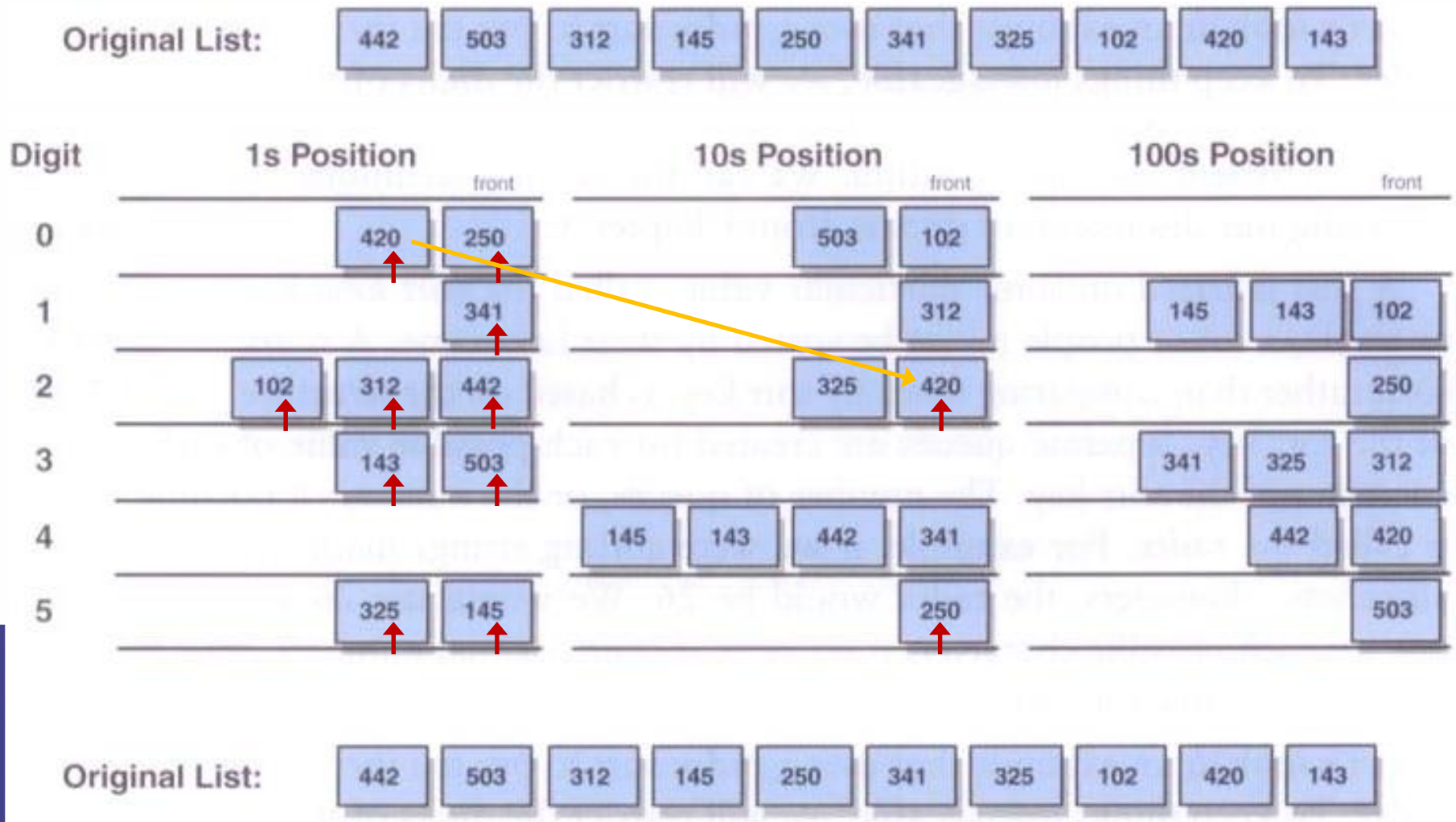
- An example using six queues to sort 10 three-digit numbers:



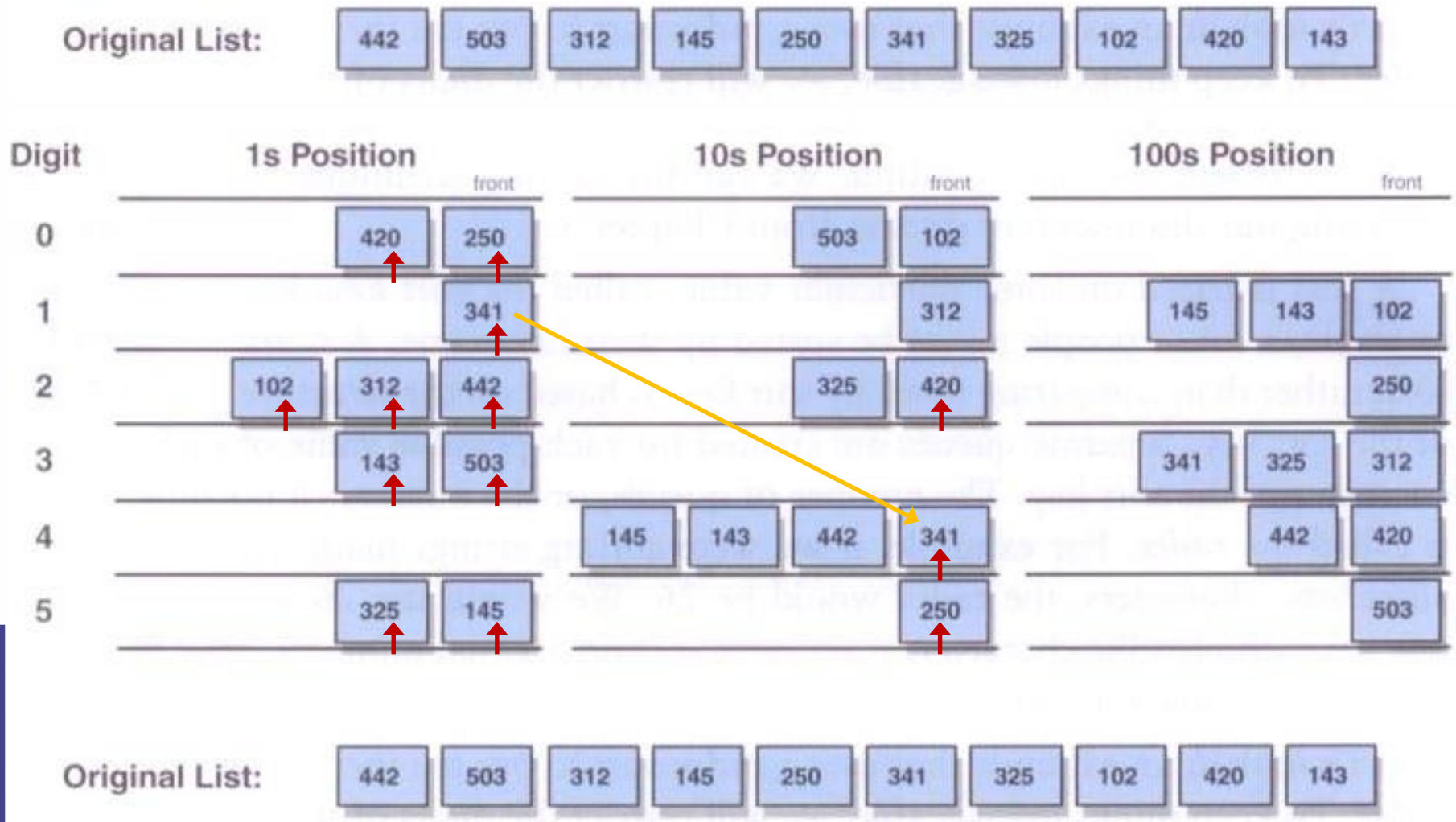
- An example using six queues to sort 10 three-digit numbers:



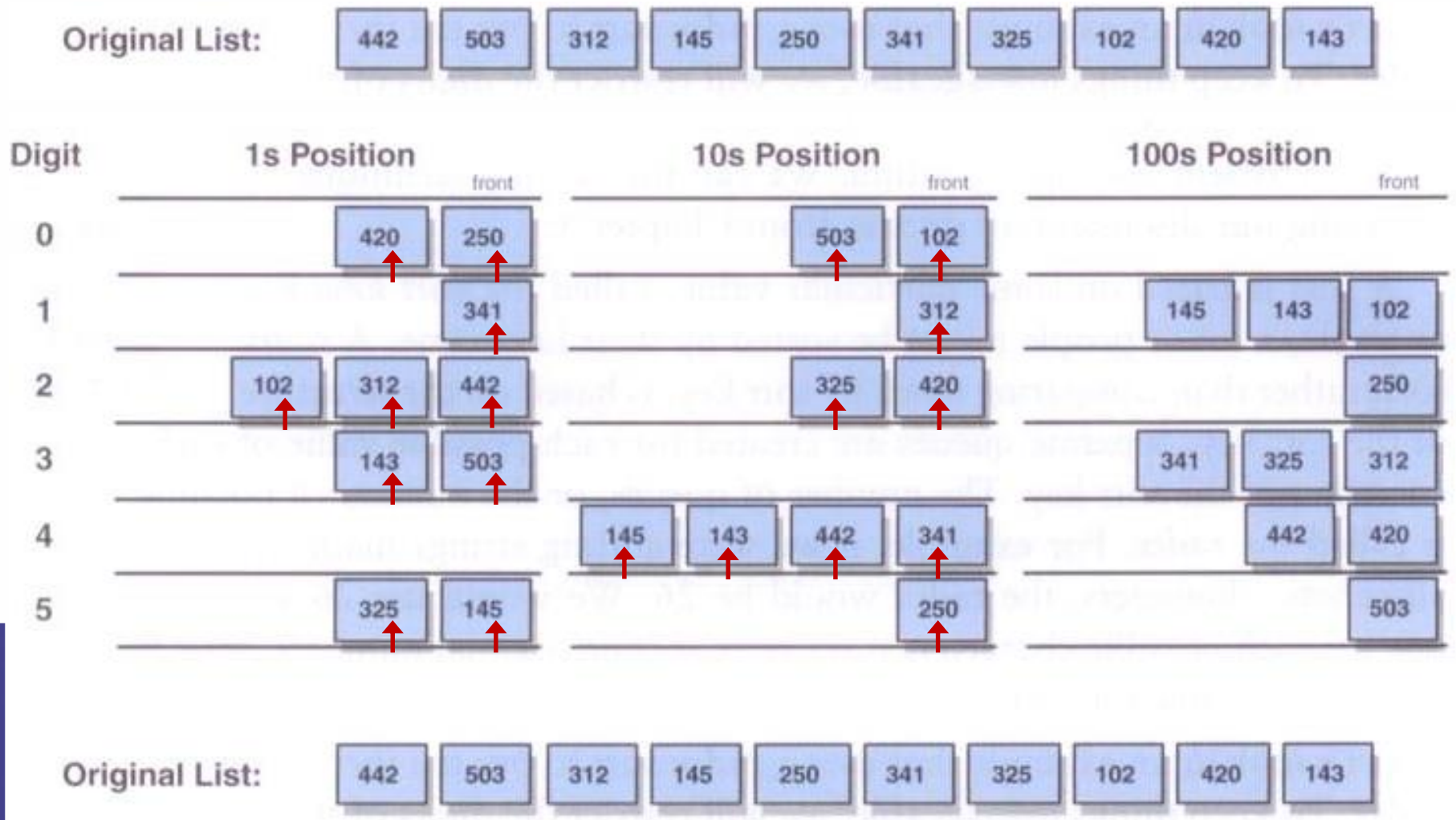
- An example using six queues to sort 10 three-digit numbers:



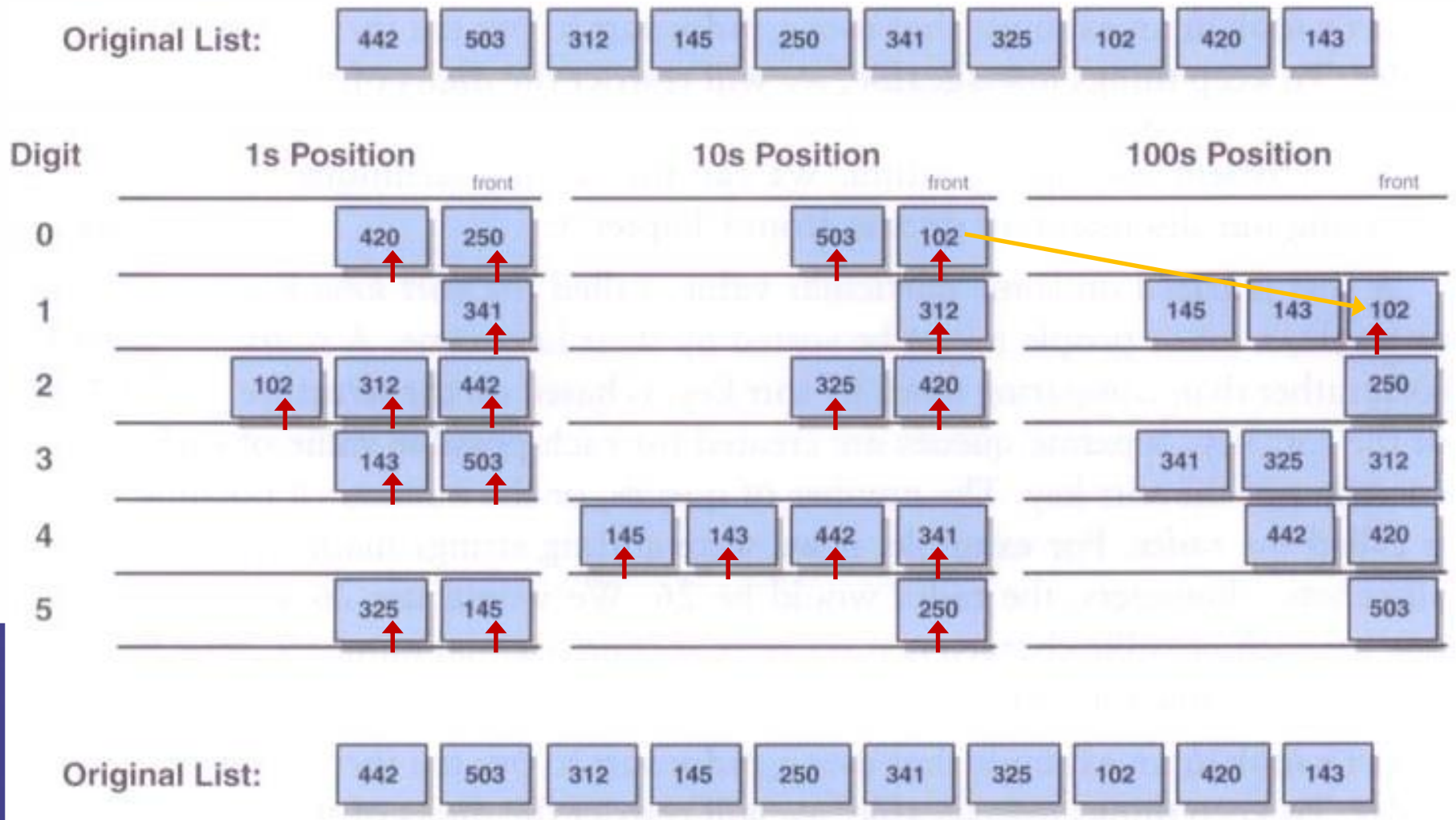
- An example using six queues to sort 10 three-digit numbers:



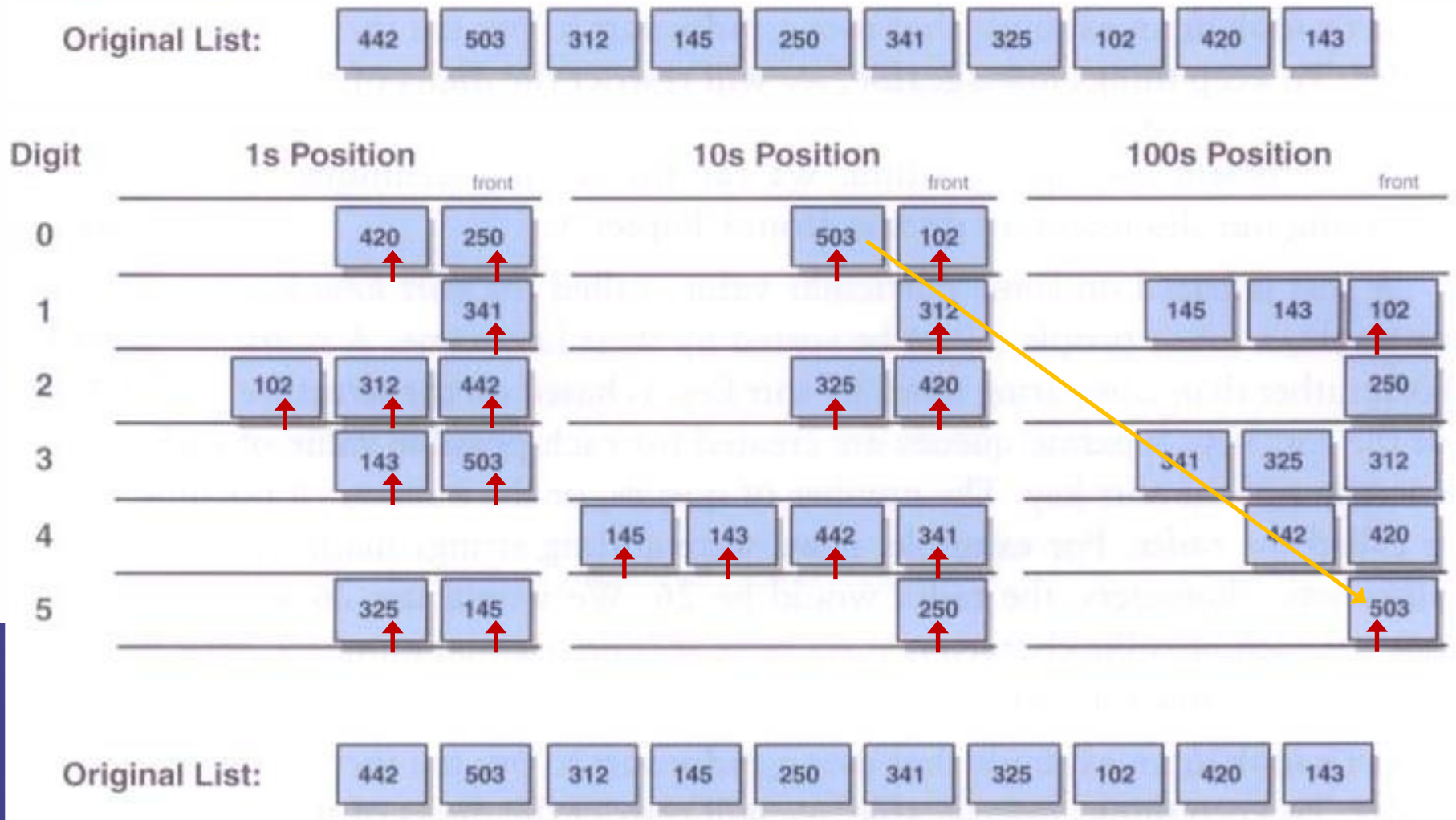
- An example using six queues to sort 10 three-digit numbers:



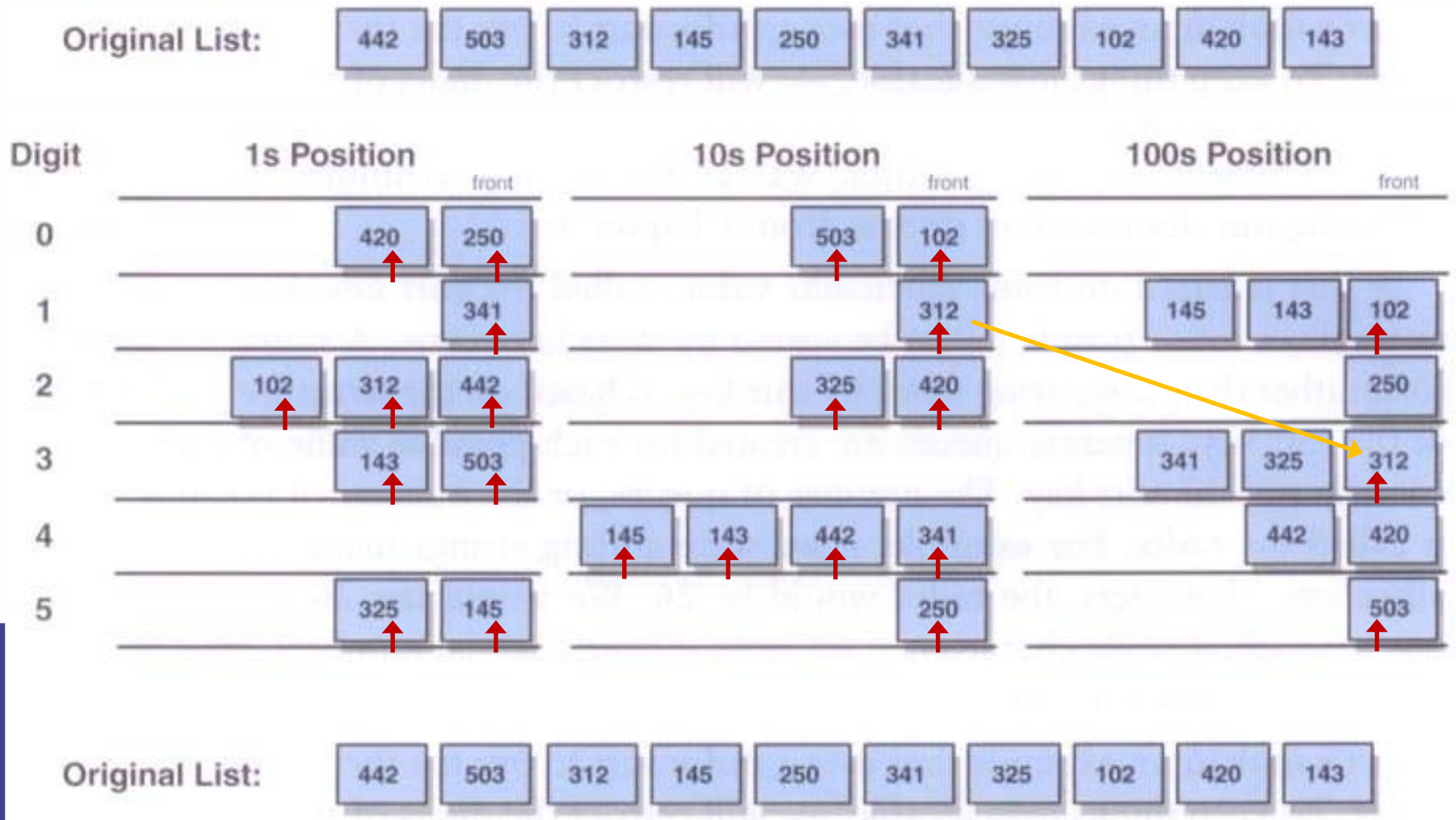
- An example using six queues to sort 10 three-digit numbers:



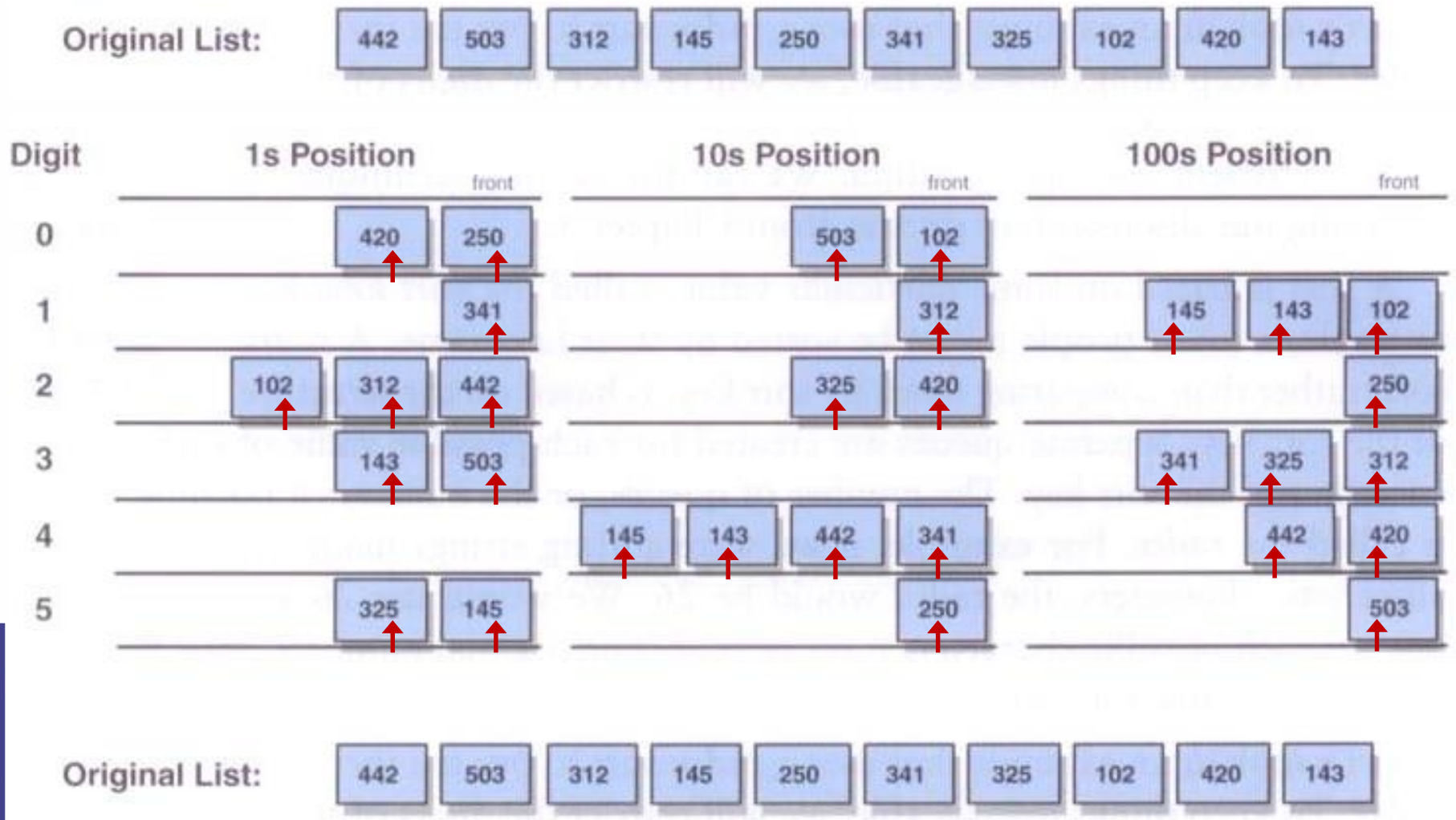
- An example using six queues to sort 10 three-digit numbers:



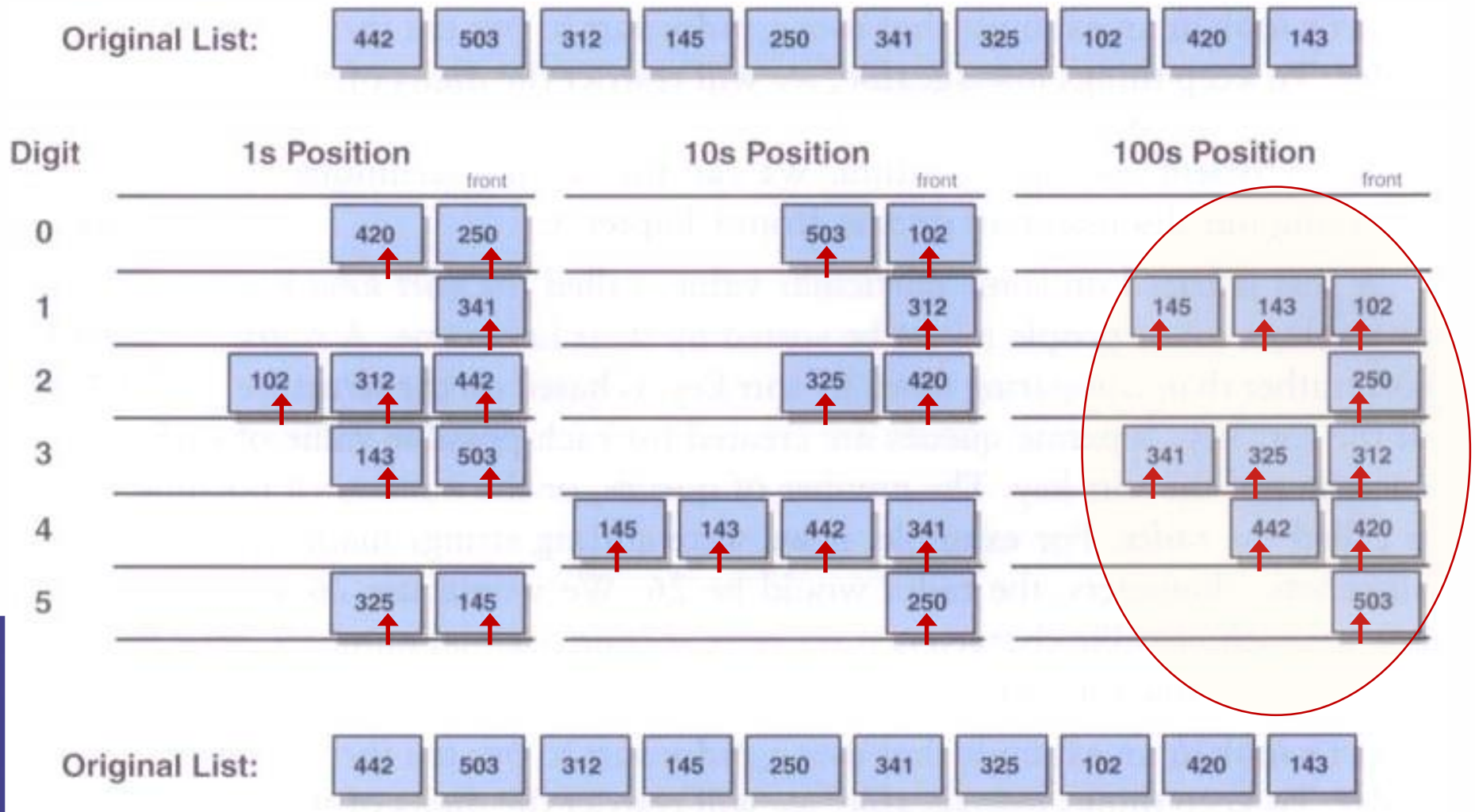
- An example using six queues to sort 10 three-digit numbers:



- An example using six queues to sort 10 three-digit numbers:



- An example using six queues to sort 10 three-digit numbers:



0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150	Original integers
(1560, 2150) (1061) (0222) (0123, 0283) (2154, 0004)	Grouped by fourth digit
1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004	Combined
(0004) (0222, 0123) (2150, 2154) (1560, 1061) (0283)	Grouped by third digit
0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283	Combined
(0004, 1061) (0123, 2150, 2154) (0222, 0283) (1560)	Grouped by second digit
0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560	Combined
(0004, 0123, 0222, 0283) (1061, 1560) (2150, 2154)	Grouped by first digit
0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154	Combined (sorted)

A radix sort of eight integers

Compare the Sorting Algorithms

Comparing the Algorithms

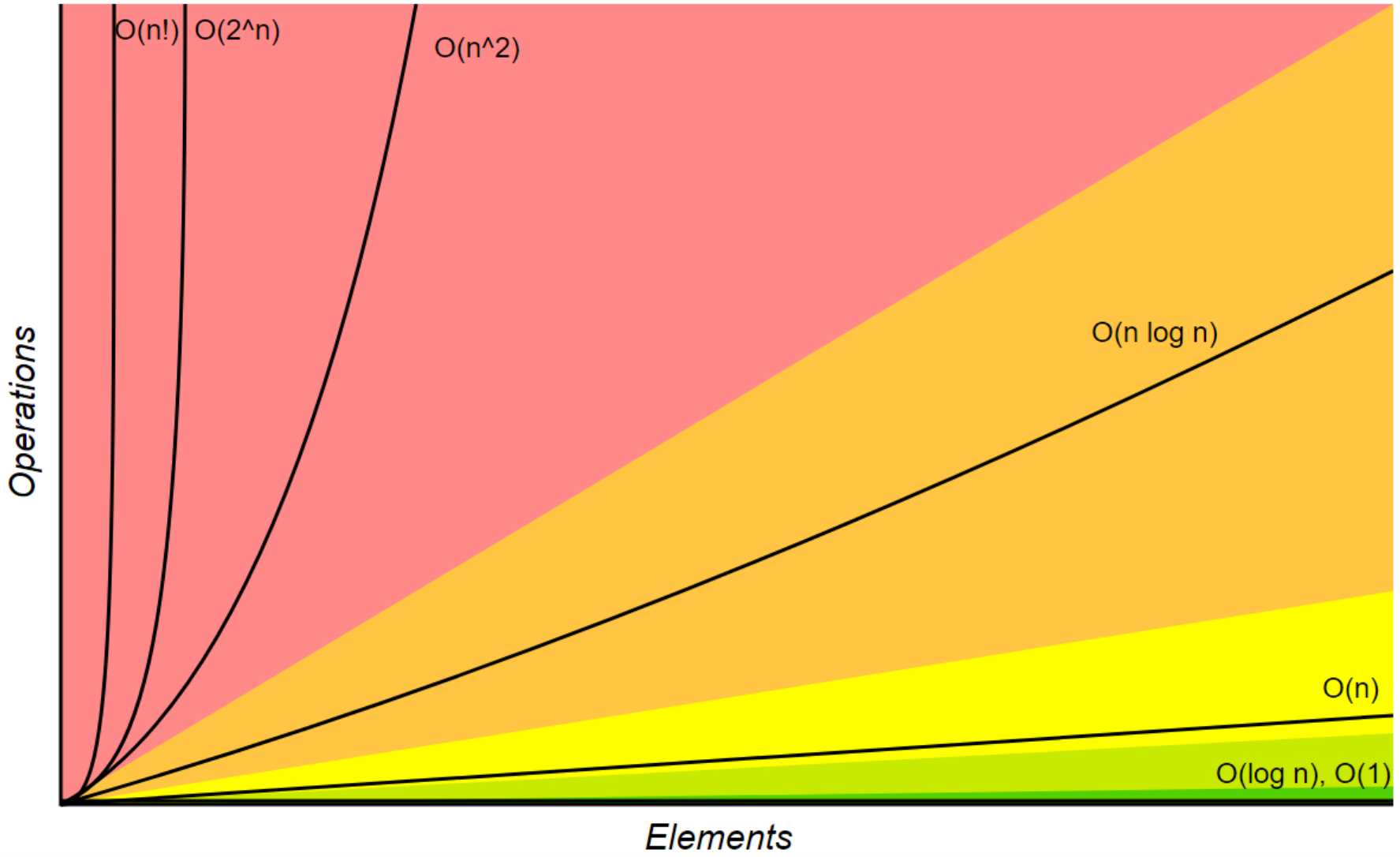
Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Horrible | Bad | Fair | Good | Excellent

http://bigocheatsheet.com/
https://en.wikipedia.org/wiki/Sorting_algorithm

Comparing the Algorithms



Horrible | Bad | Fair | Good | Excellent

http://bigocheatsheet.com/
https://en.wikipedia.org/wiki/Sorting_algorithm

See you next class!