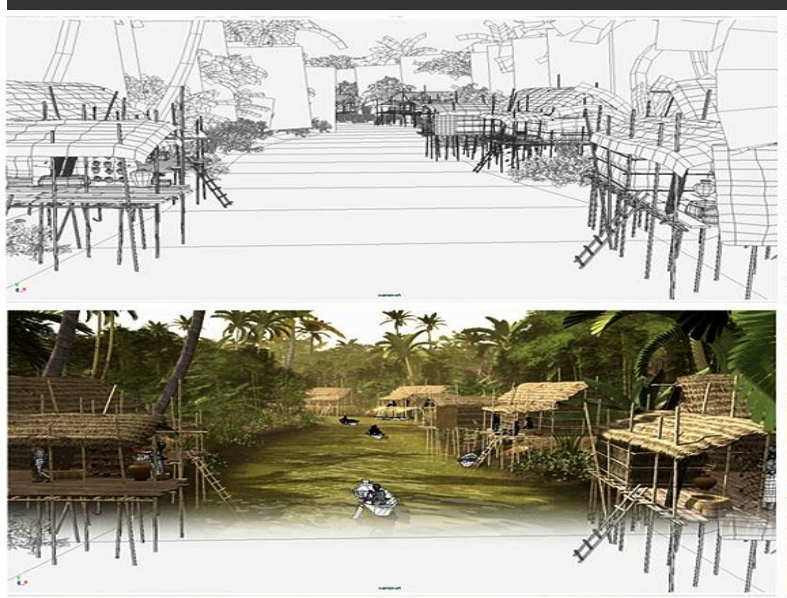
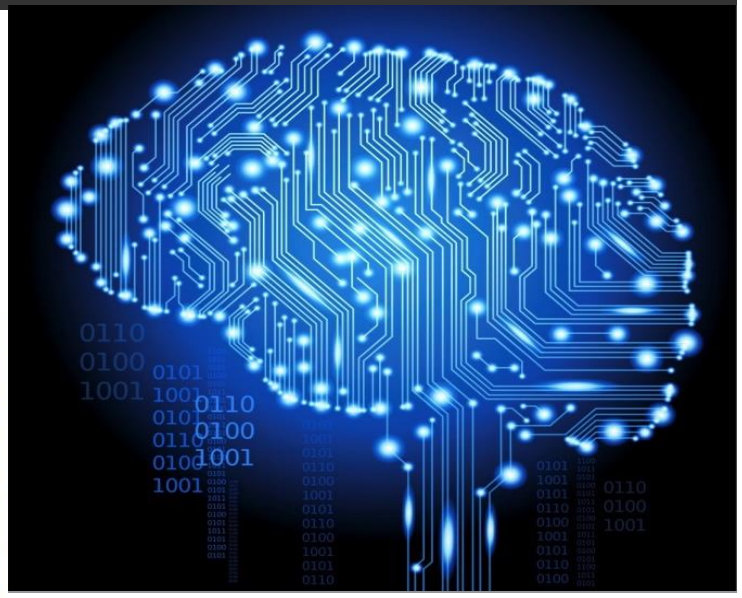
MONASH University

Information Technology

Recursive Sorts

Prepared by Maria Garcia de la Banda
Updated by Brendon Taylor





Overview

- **To review what a "divide and conquer" algorithm is**
- **To review in more depth two different "divide and conquer" sorting algorithms:**
  - Merge Sort
  - Quick Sort
- **To be able to implement them and compare their efficiency for different classes of inputs**

MONASH University

Divide and Conquer

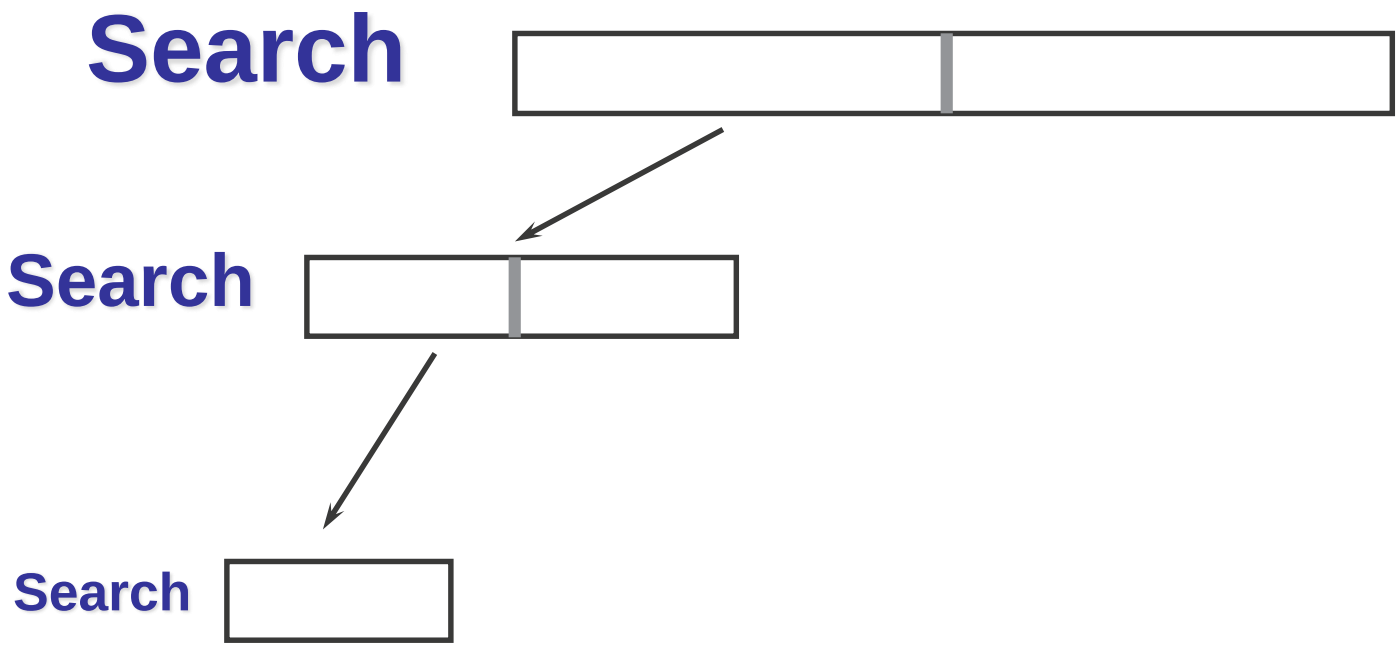
Divide-and-conquer algorithms

- **One common recursive strategy: divide-and-conquer**
  - Split the original problem into subproblems
  - Solve each subproblem independently
  - Combine their solutions to yield the final solution

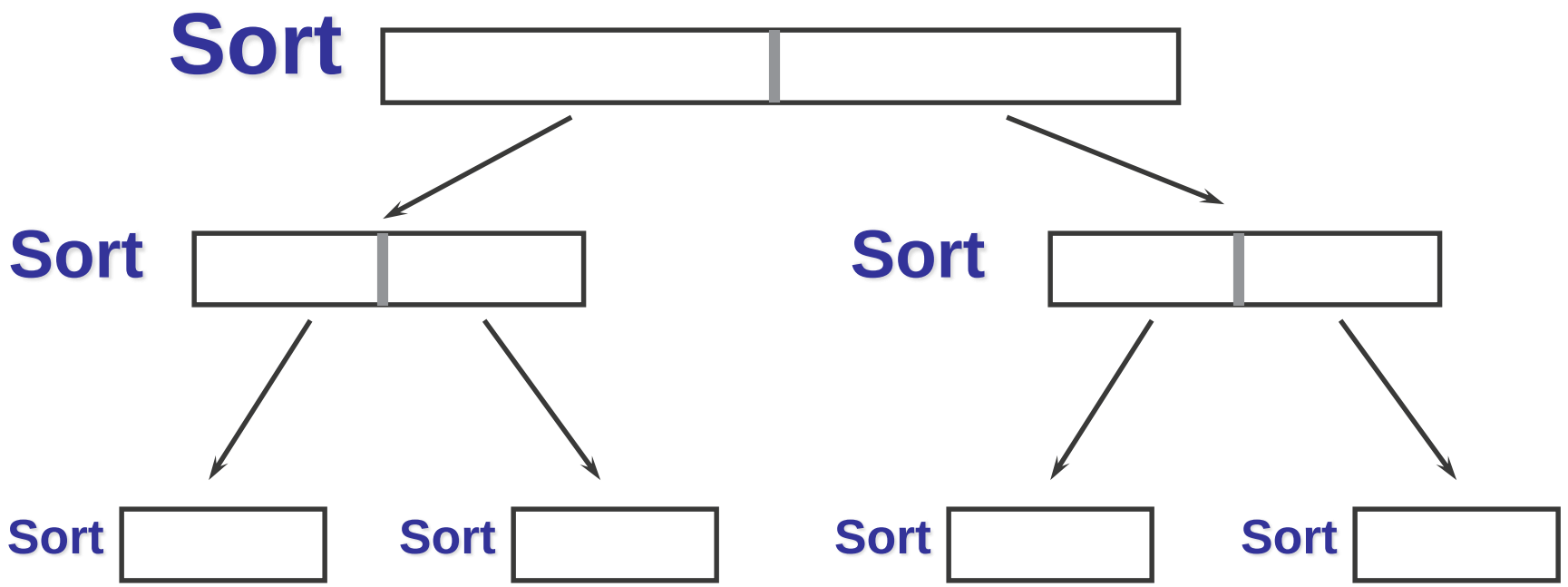
- **Such algorithms can be very efficient, especially when the subproblems have roughly the same size**

Searching by Divide and Conquer

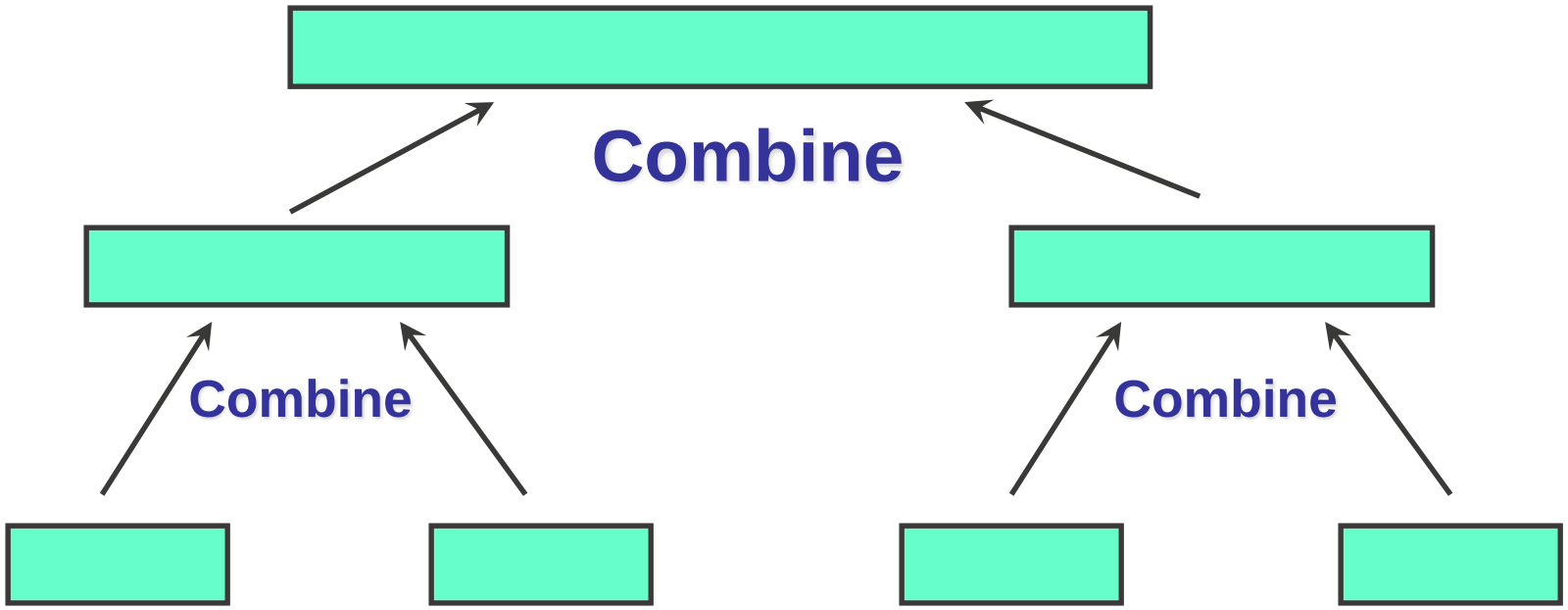
Recall: Binary Search



Sorting arrays by Divide and Conquer



Sorting arrays by Divide and Conquer



Sorting arrays by Divide and Conquer

- **The general strategy is (usually – not always -- for an array or a list implemented with arrays):**

```
def sort(array: ArrayR) -> None:
    if len(array) > 1:
        split(array, first_part, second_part)
        sort(first_part)
        sort(second_part)
        combine(first_part, second_part)
```

Why len > 1?

If len <=1, it is already sorted

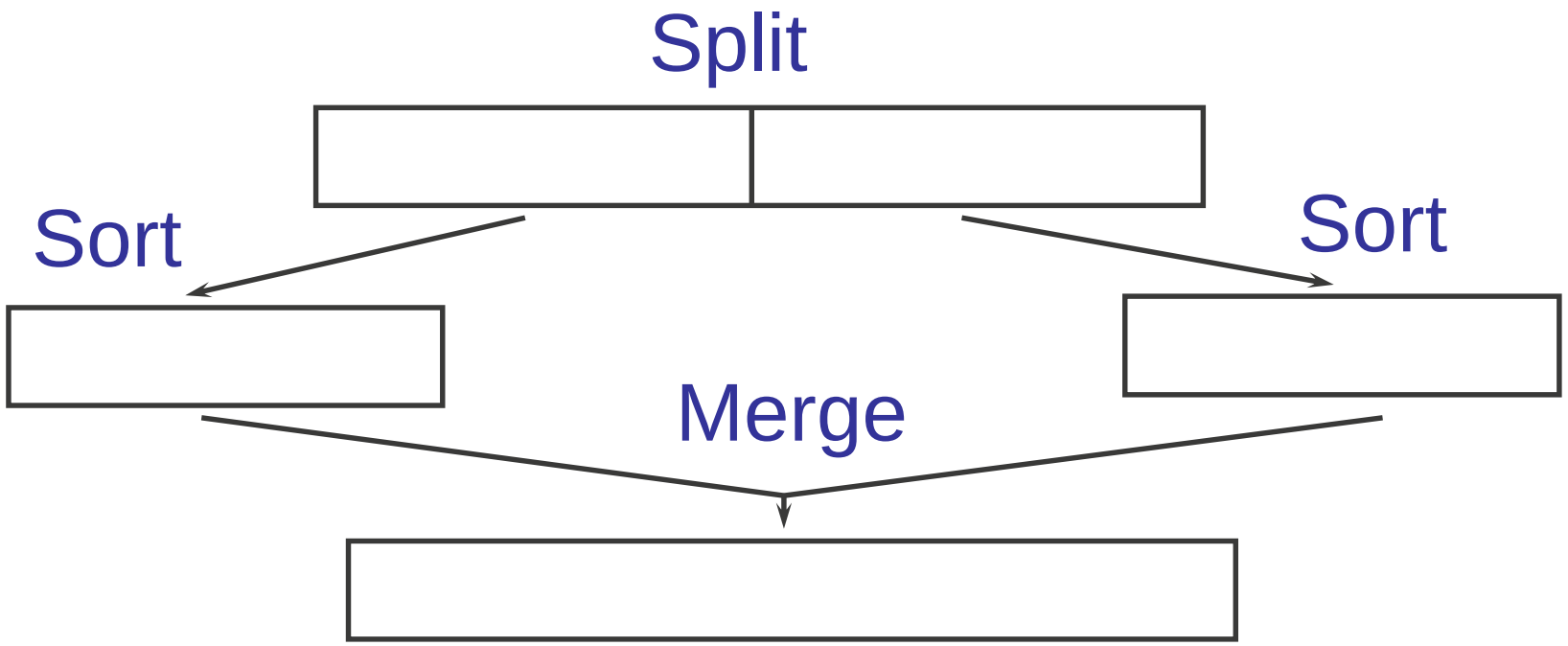
- **Merge Sort has a simple split and a complex combine**
- **Quick Sort has a complex split and a simple combine**

MONASH University

Merge Sort

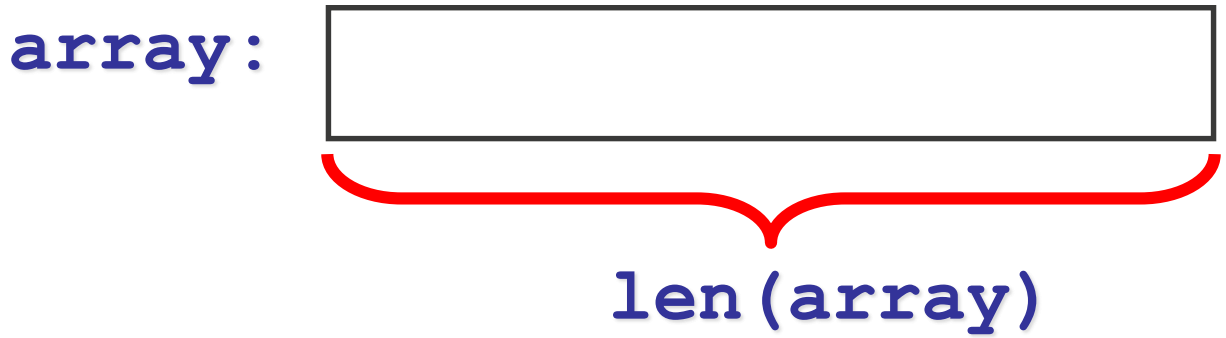
Merge Sort

- **The split is trivial: splits the array into two halves**
- **The combination is non-trivial: merges two sorted halves into a sorted array**

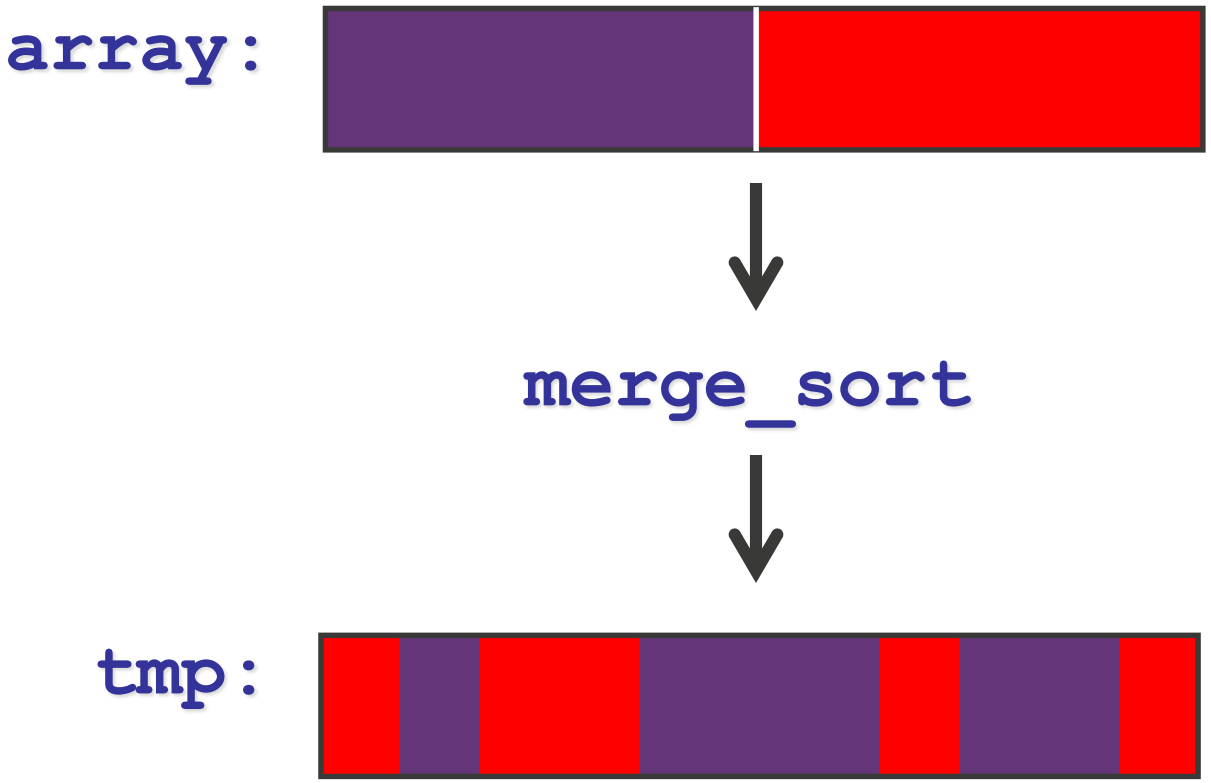


Merge Sort

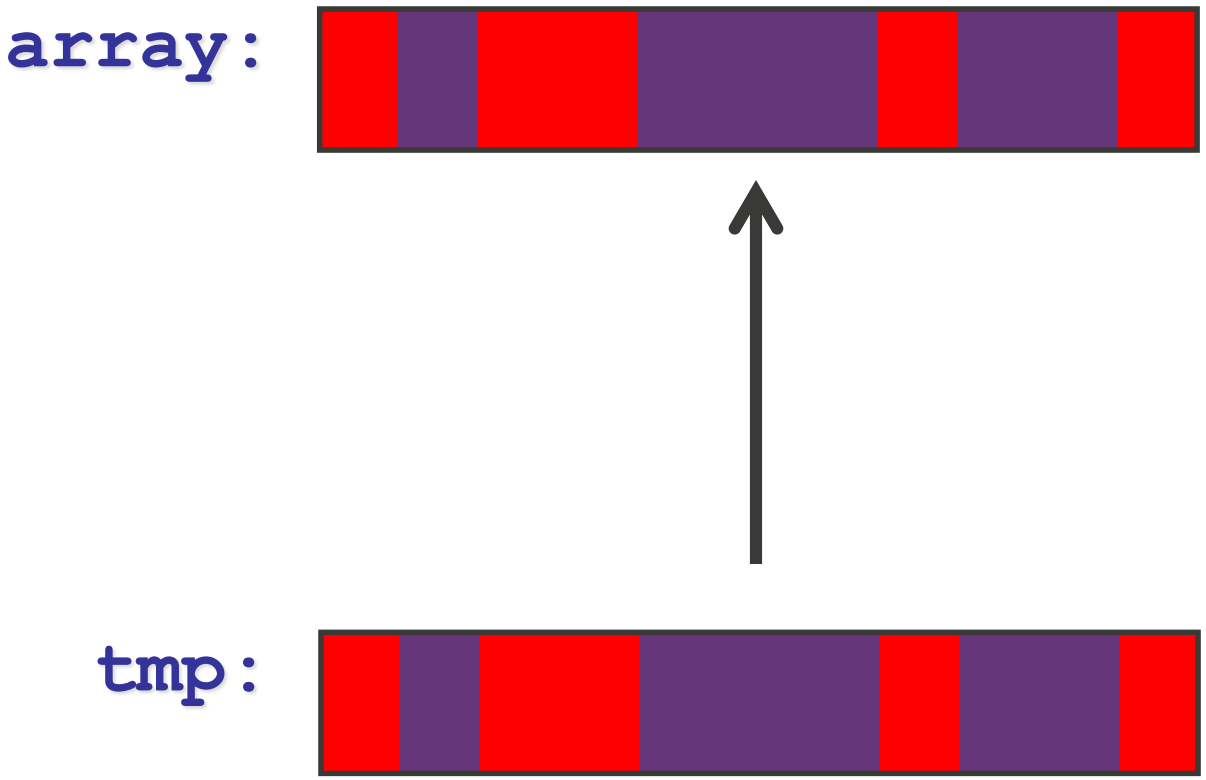
- For this it requires a temporary array of the same size as the original array



Merge Sort



Merge Sort



Merge Sort

`array:` 

Merge Sort method

- **So we need:**
  - A temporary array to store the merged (partial) solution
  - Markers for the part of the array I am looking at (no need for new array)

```
def merge_sort(array: ArrayR) -> None:

    tmp = ArrayR(len(array))

    start = 0

    end = len(array)-1

    merge_sort_aux(array, start, end, tmp)
```

The temporary array I need

Frontiers to the part of the array I am currently looking at

Auxiliary method with all arguments I need

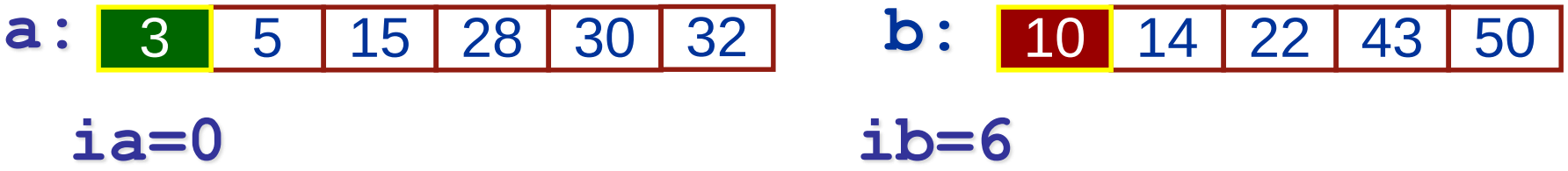
merge_sort_aux method

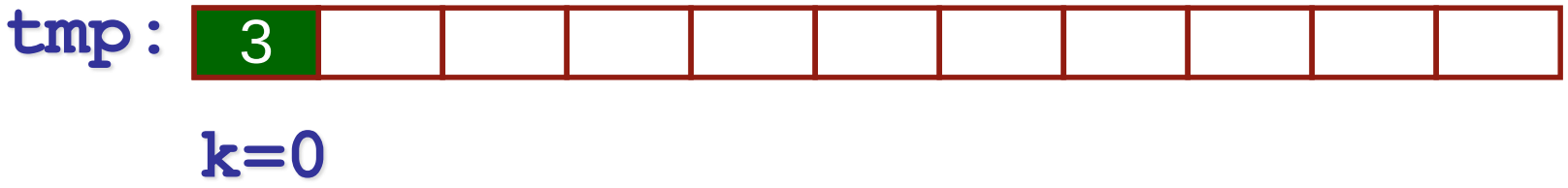
```
def merge_sort_aux(array: ArrayR, start: int, end: int, tmp: T) -> None:
    if not start == end: # 2 or more still to sort
        mid = (start + end)//2
        # split into two halves
        merge_sort_aux(array, start, mid, tmp)
        merge_sort_aux(array, mid+1, end, tmp)
        # merge
        merge_arrays(array, start, mid, end, tmp)
        # copy tmp back into the original
        for i in range(start, end+1):
            array[i] = tmp[i]
```

Example: merge_arrays

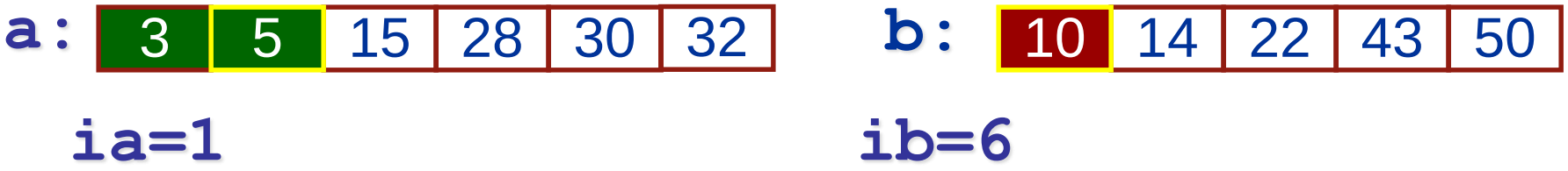

```
start: 0        mid: 5        end: 10
```

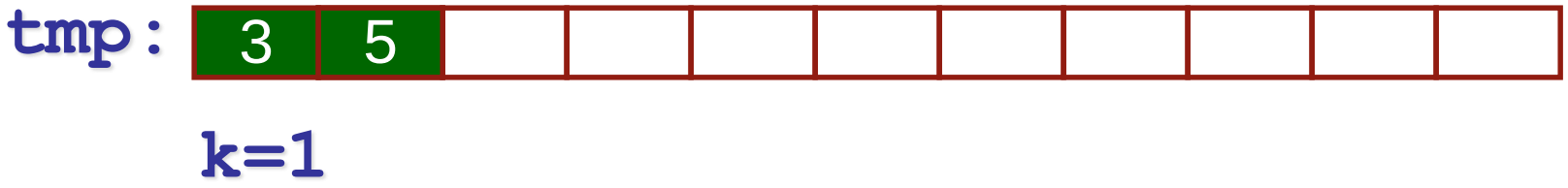
Example: merge_arrays



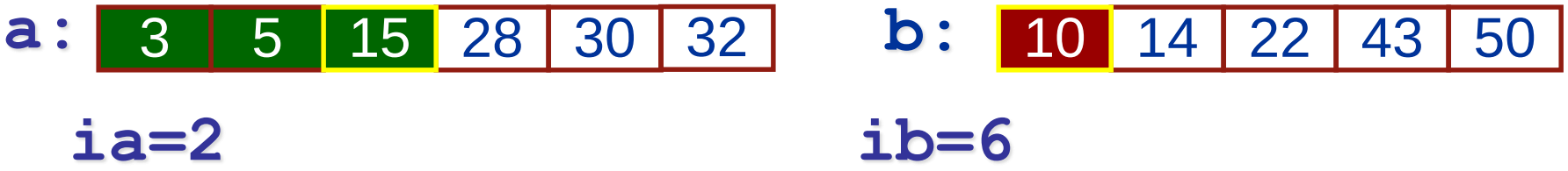


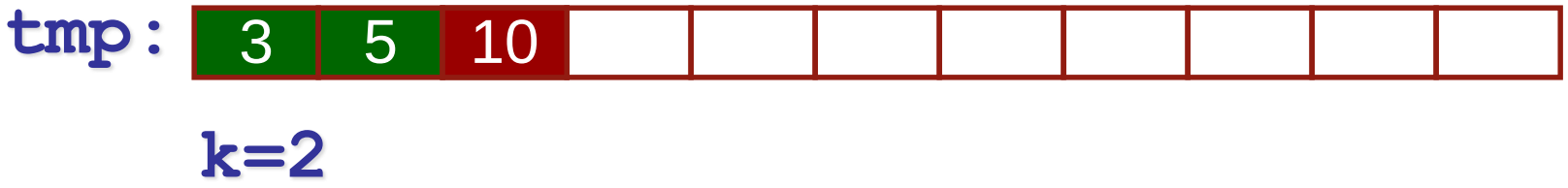
Example: merge_arrays



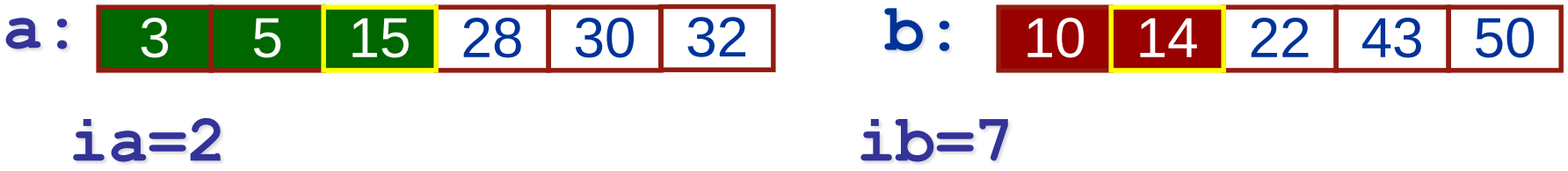


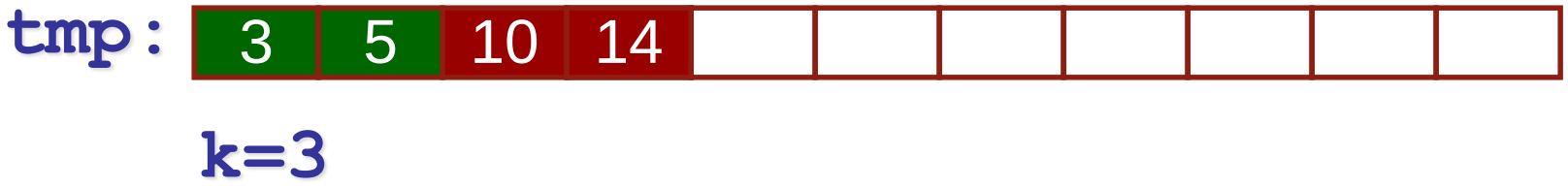
Example: merge_arrays



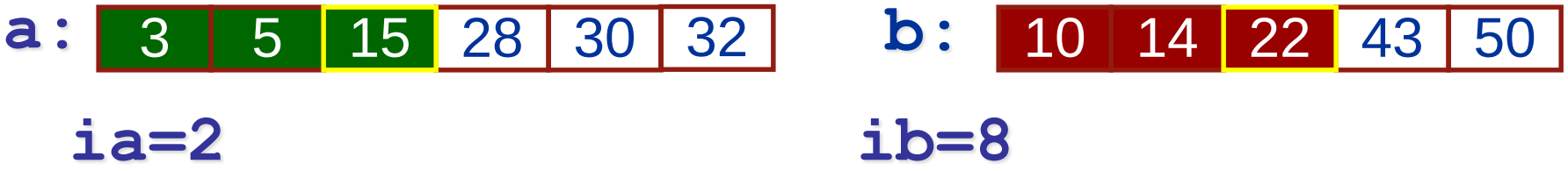


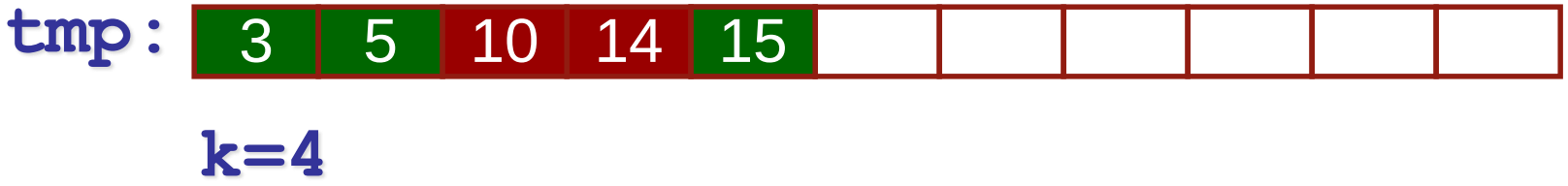
Example: merge_arrays





Example: merge_arrays





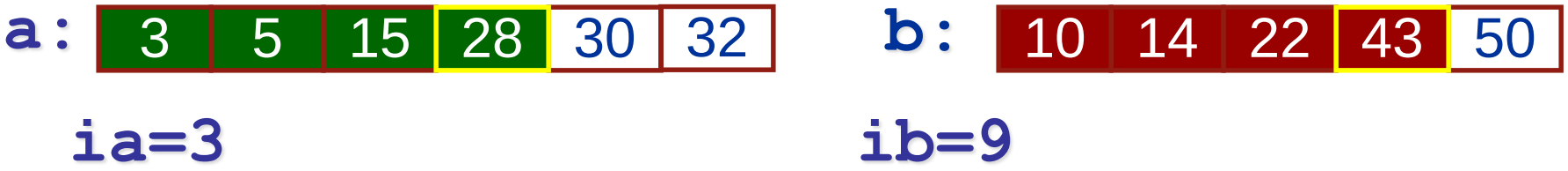
Example: merge_arrays

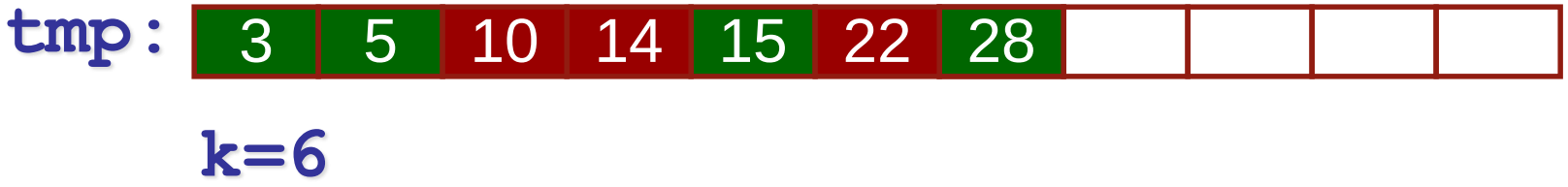
`ia=3`

`ib=8`

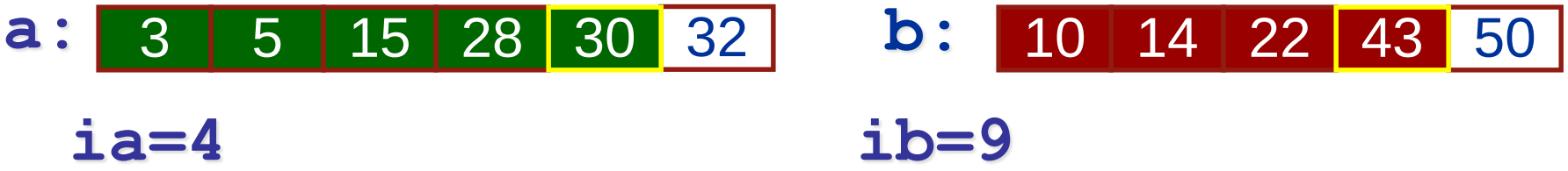
`k=5`

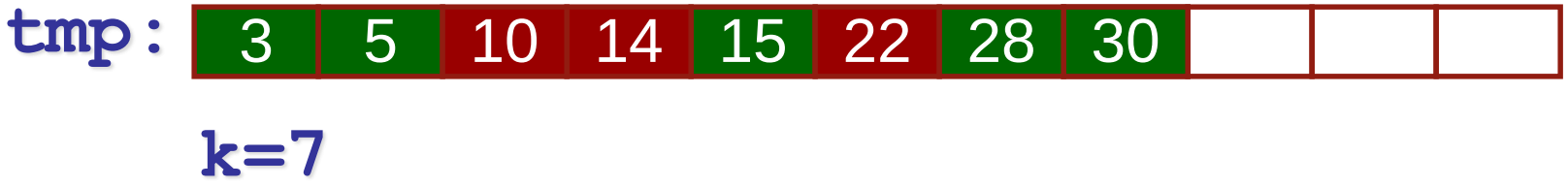
Example: merge_arrays



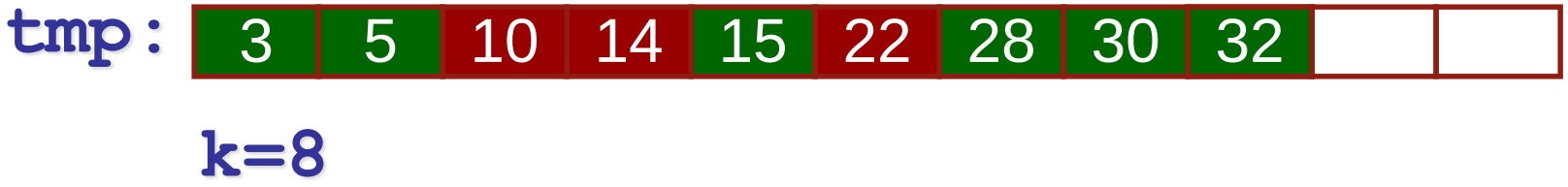


Example: merge_arrays

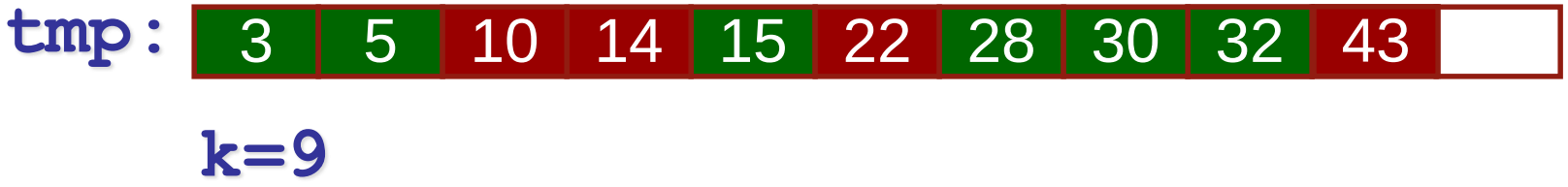




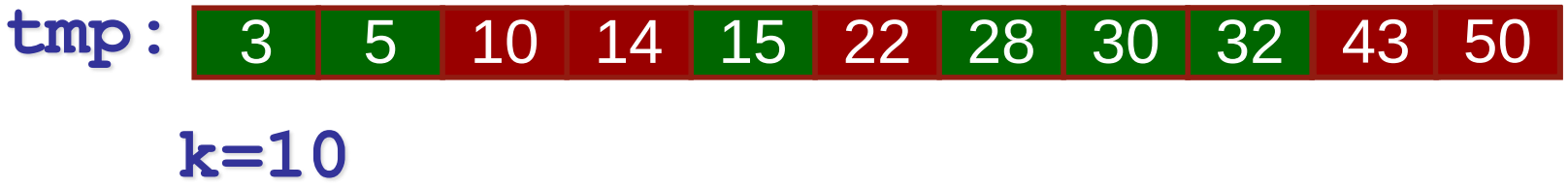
Example: merge_arrays



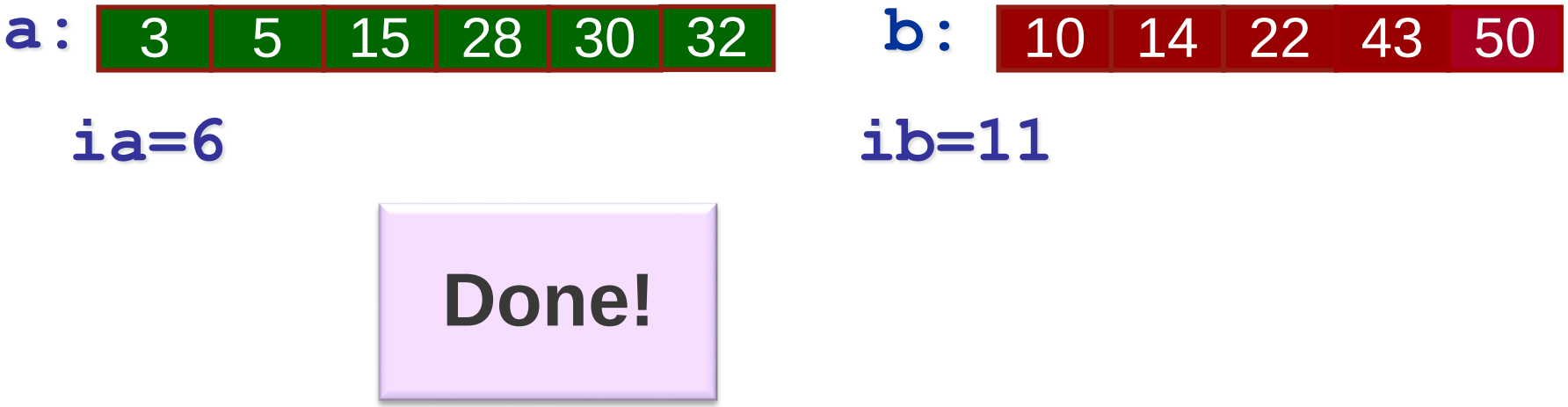
Example: merge_arrays



Example: merge_arrays



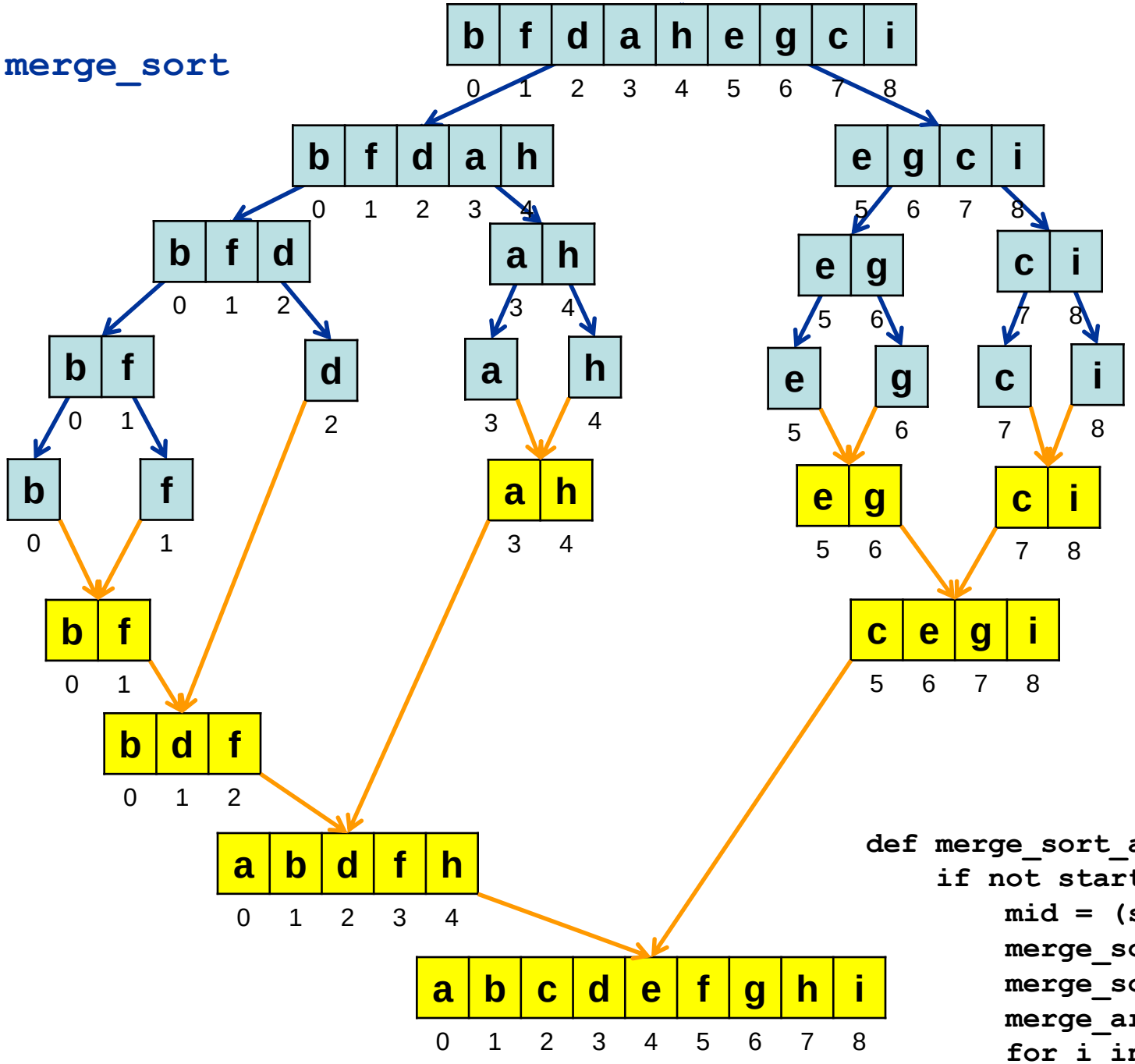
Example: merge_arrays



merge_arrays method

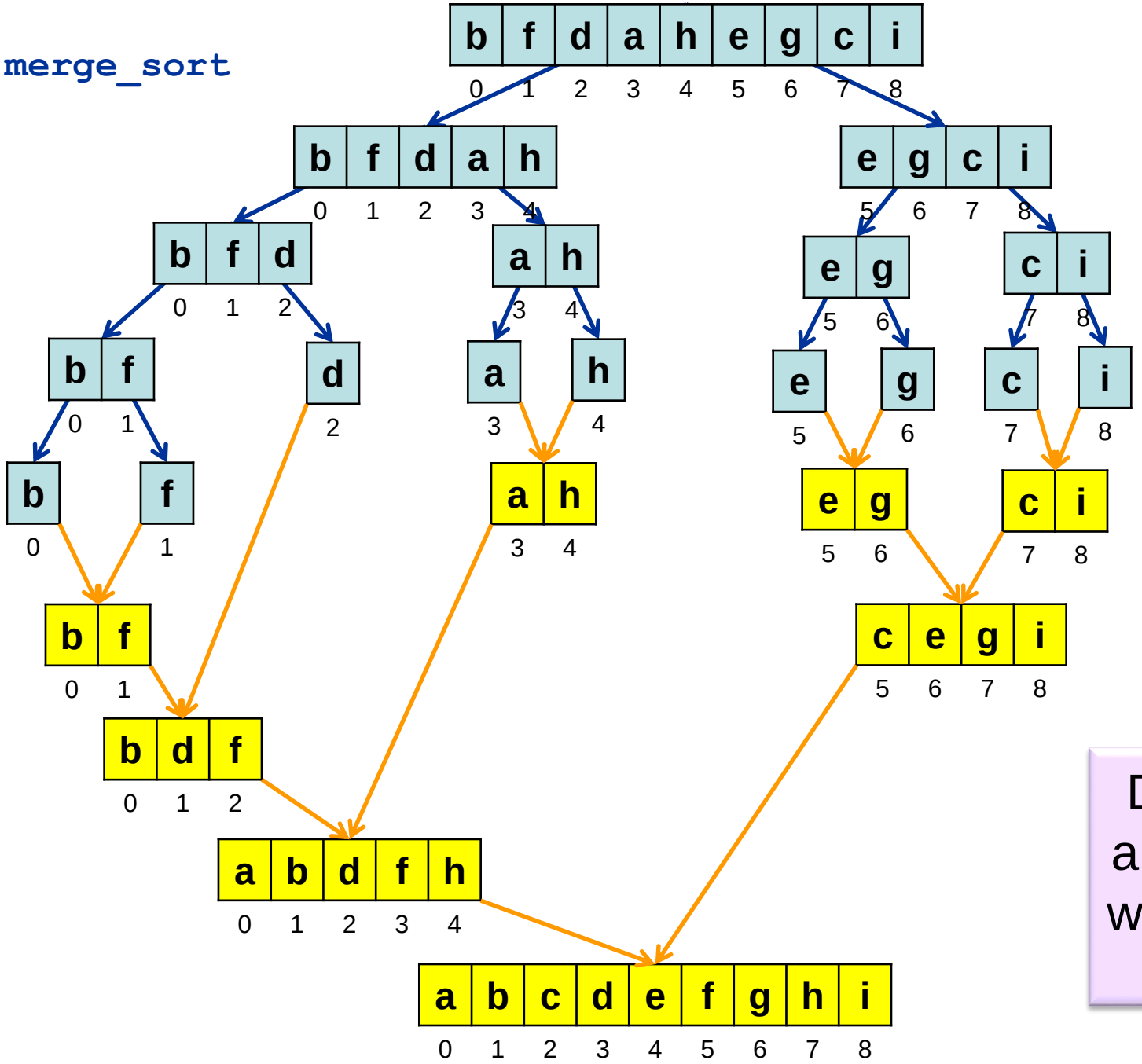
```
def merge_arrays(a: ArrayR, start: int, mid: int, end: int, tmp: T) -> None:
    ia = start
    ib = mid+1
    for k in range(start, end+1):
        if ia > mid: # a finished, copy b
            tmp[k] = a[ib]
            ib += 1
        elif ib > end: # b finished, copy a
            tmp[k] = a[ia]
            ia += 1
        elif a[ia] <= a[ib]: # a[ia] is the item to copy
            tmp[k] = a[ia]
            ia += 1
        else:
            tmp[k] = a[ib] # b[ib] is the item to copy
            ib += 1
```

merge_sort




```
def merge_sort_aux(array, start, end, tmp):
    if not start == end: # 2+ still to sort
        mid = (start + end)//2
        merge_sort_aux(array, start, mid, tmp)
        merge_sort_aux(array, mid+1, end, tmp)
        merge_arrays(array,start,mid,end,tmp)
        for i in range(start,end+1):
            array[i] = tmp[i]
```

merge_sort



Depth is log(N), and at each level we do N constant operations

Merge Sort Analysis

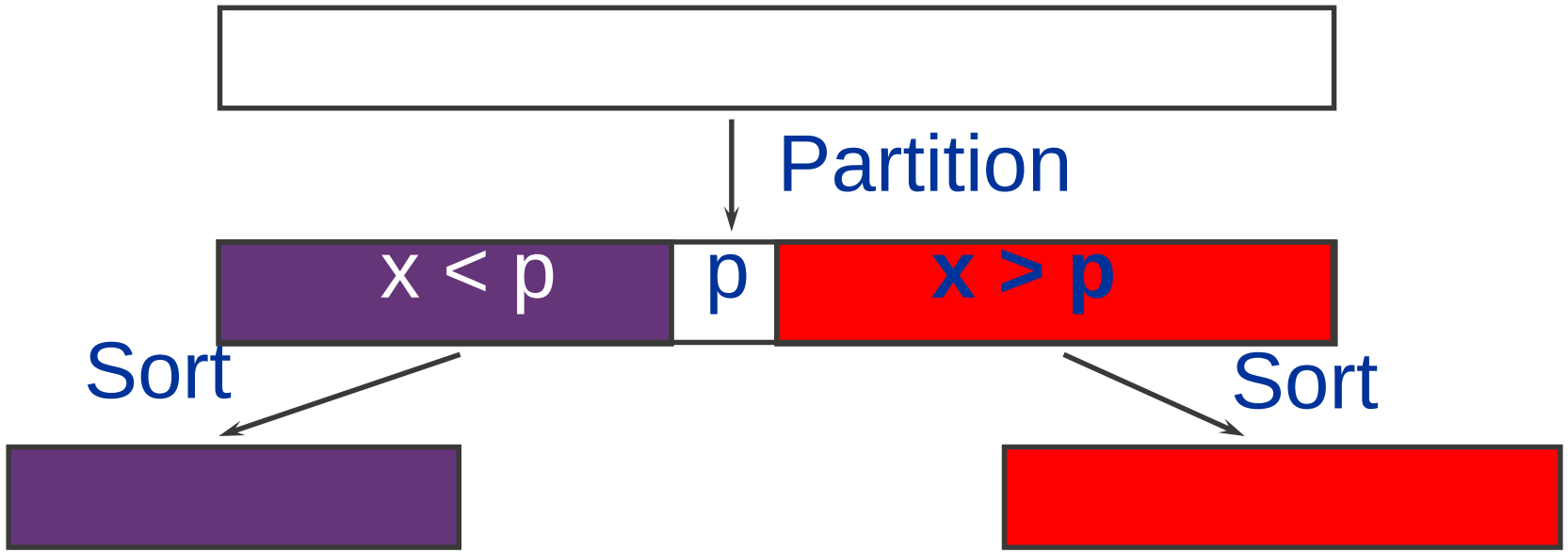
- **This is typically the method that you would naturally use when sorting a pile of books, CDs cards, etc.**
- **Most of the work is in the merging**
- **Uses more space than other sorts**
- **Takes O(n log(n))*CompEq, where n is the number of elements in the array and CompEq is the complexity of the comparison**
- **Close to optimal in number of comparisons**
  - Good for languages where comparison is expensive

MONASH University

Quick Sort

Quick Sort

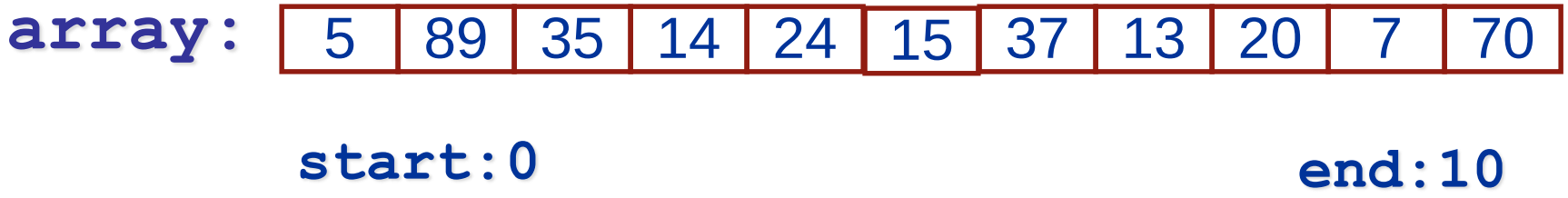
- **The split is non-trivial: partitions the array into two:**
  - One with all elements smaller than the pivot p
  - Another with all elements greater than the pivot
- **Combination is trivial: append with pivot in the middle**



Quick Sort: choosing the pivot

- **It must be an element in the array**
- **Ideally, its value is the median of all values**
  - This is does not mean is in the middle position! (unless the list is already sorted)
- **Why: best when the subarrays are about the same size**
- **Hitch: how to pick the median element? (or similar)**
- **One strategy: pick a small sample of elements (e.g. first, last, mid) and choose the median of these**
- **Or you could just pick a random element**
- **First element is usually a bad idea: presorted input**

Example: Quick Sort



Example: Quick Sort

Say we randomly choose 15 as the pivot.

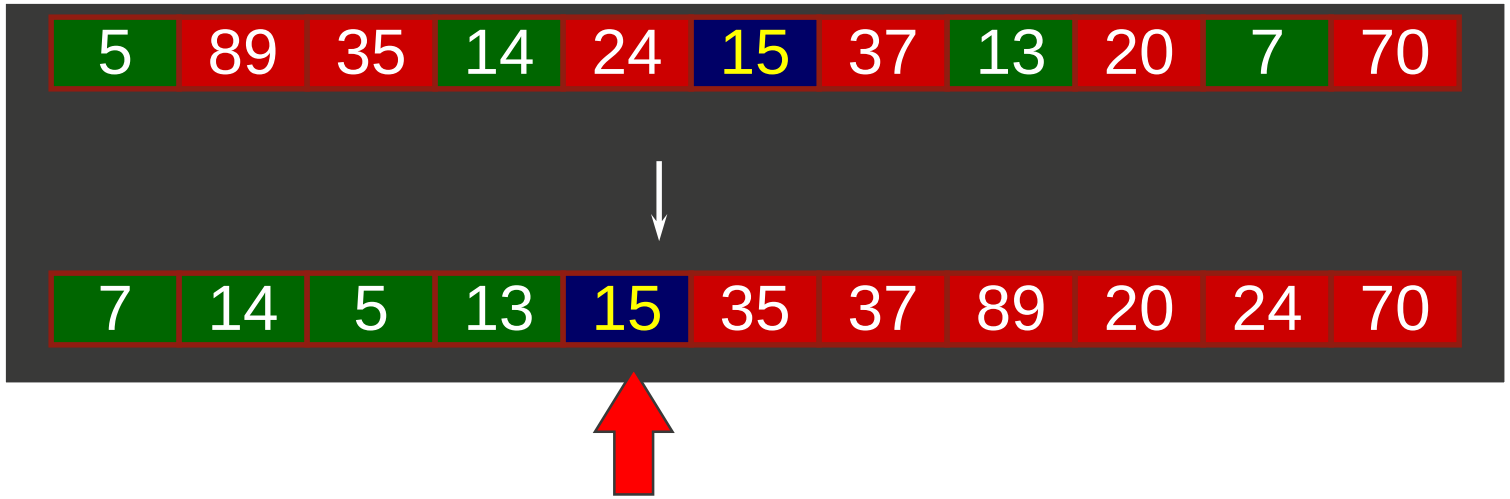
It happens to be in the middle of the list, but that's just a coincidence!

Example: Quick Sort

array:

5	89	35	14	24	15	37	13	20	7	70

partition:



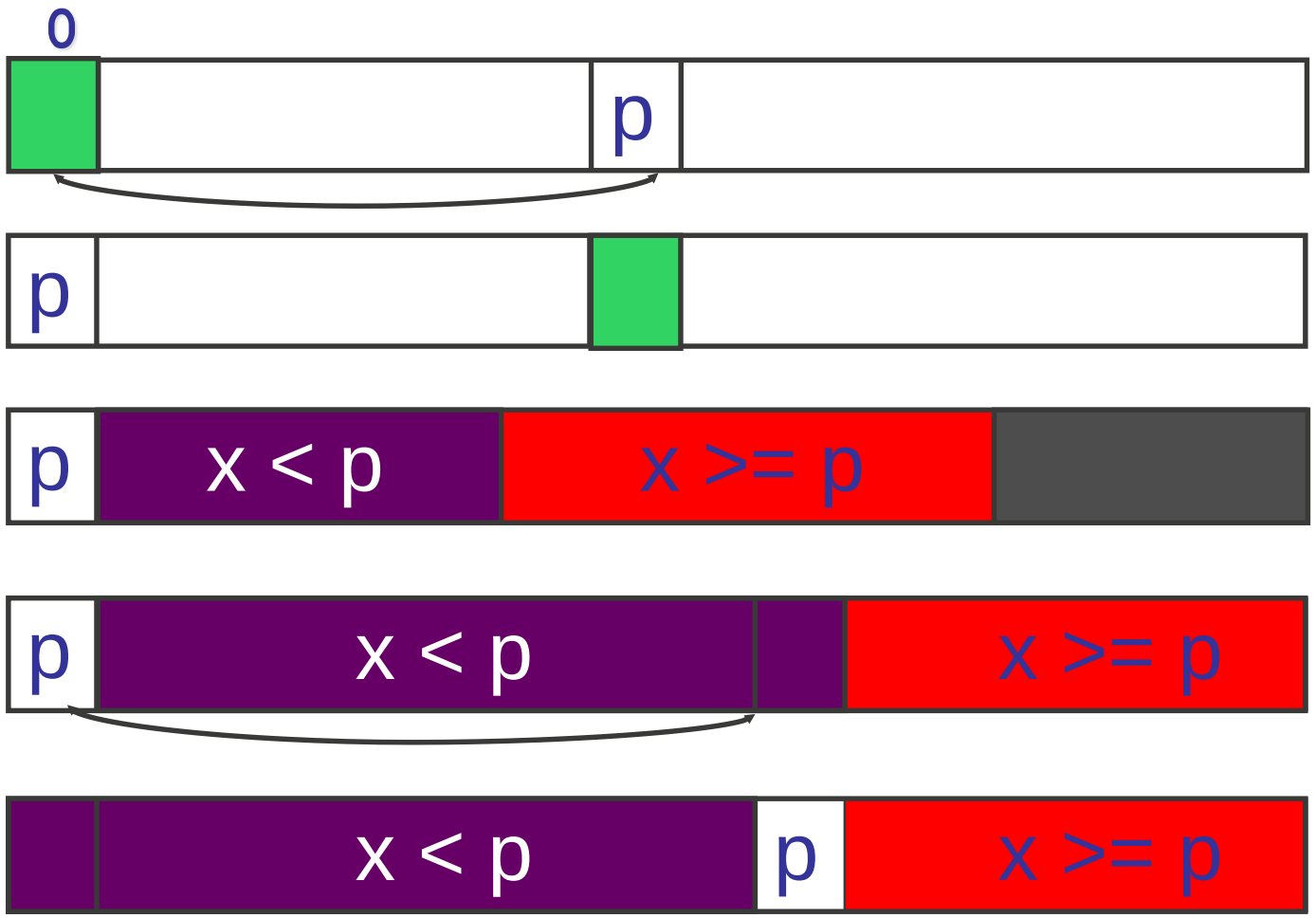
Quick Sort method

```
def quick_sort(array: ArrayR) -> None:
    start = 0
    end = len(array)-1
    quick_sort_aux(array, start, end)

def quick_sort_aux(array: ArrayR, start: int, end: int) -> None:
    if start < end:
        boundary = partition(array, start, end)
        quick_sort_aux(array, start, boundary-1)
        quick_sort_aux(array, boundary+1, end)
```

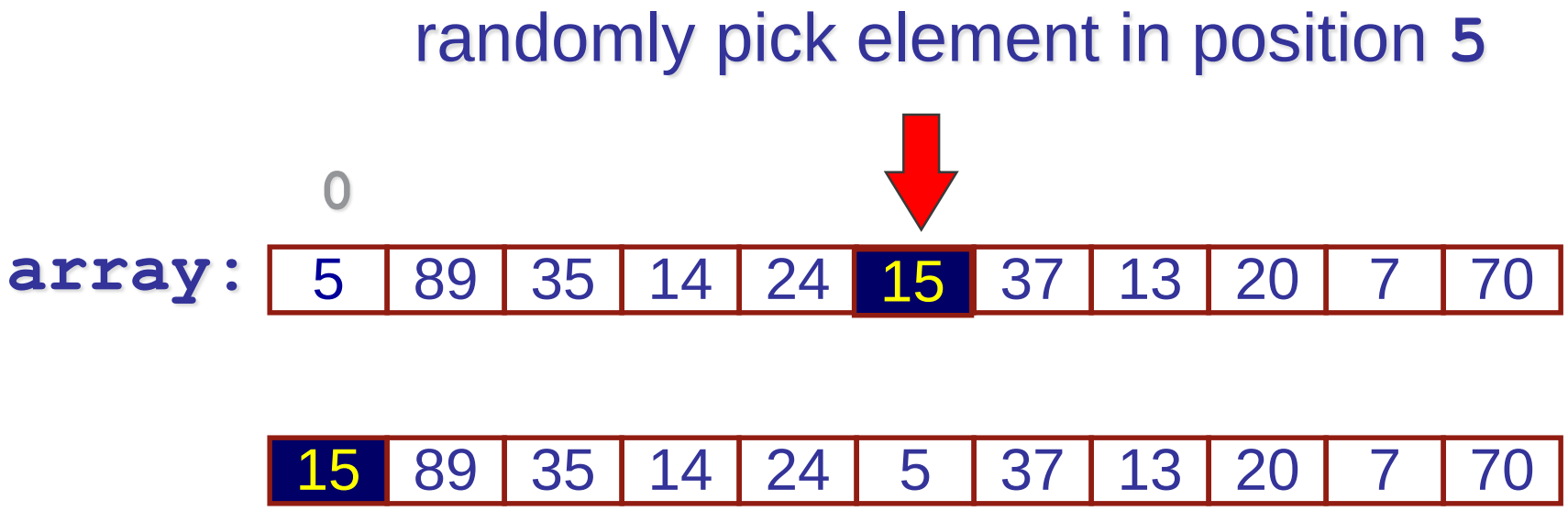
But no need for a temporary array this time

Again, the frontiers to the part of the array I am currently looking at

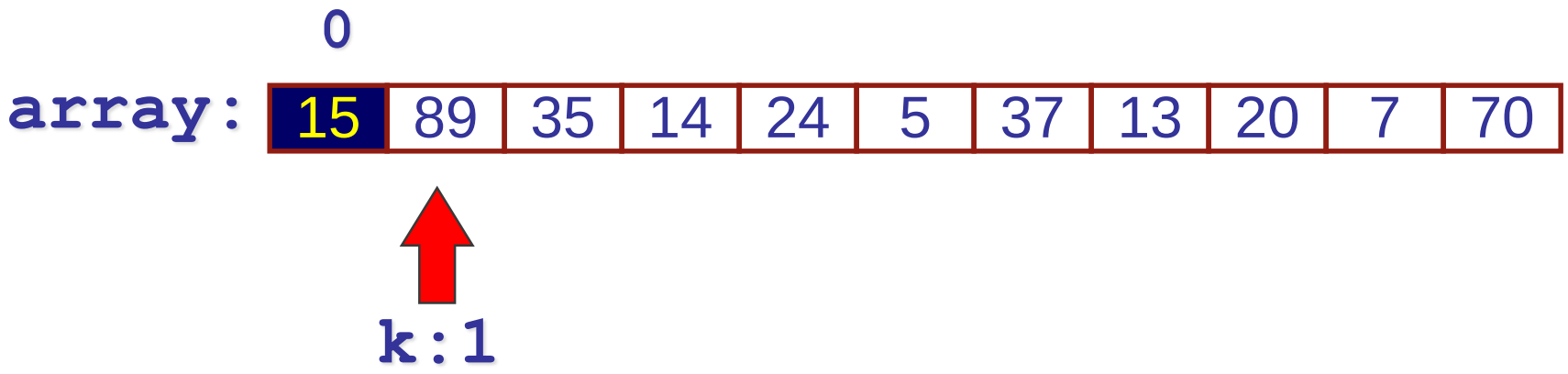
Partition: Checklist



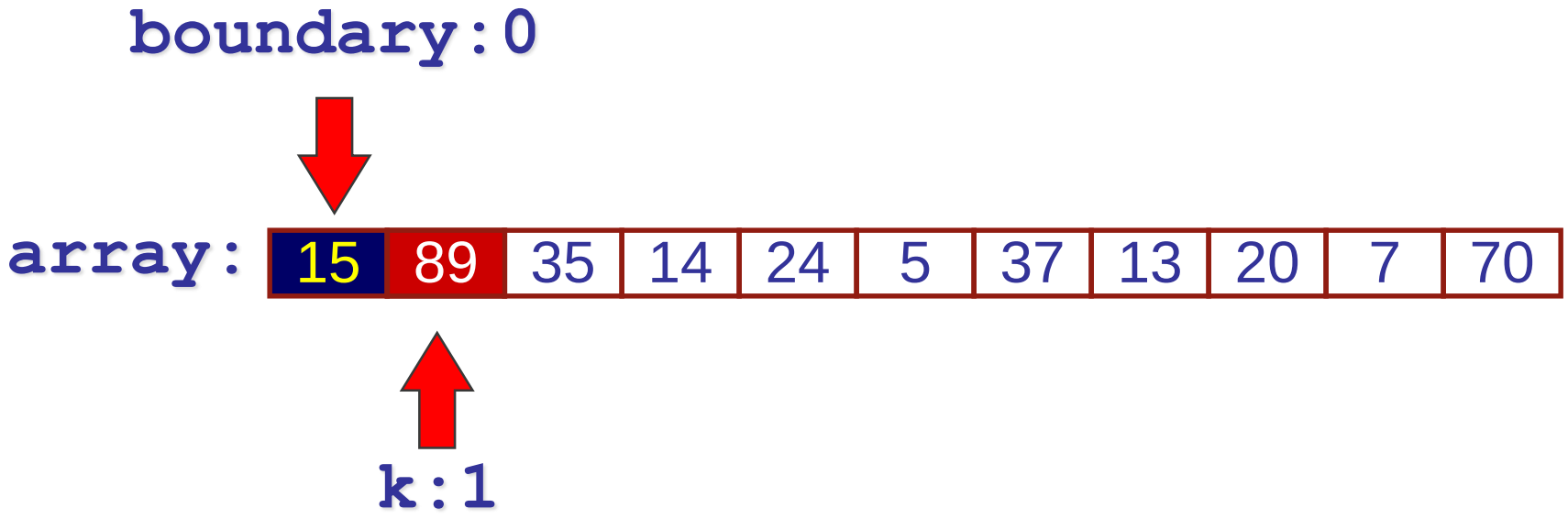
Example: Partition



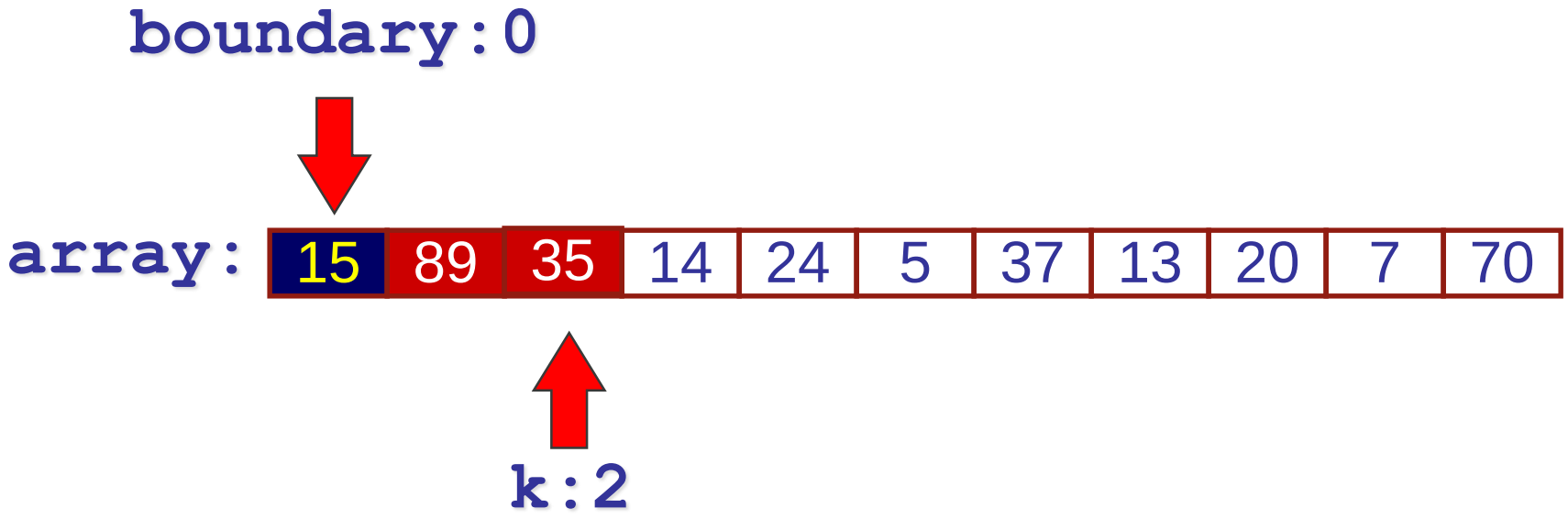
Example: Partition



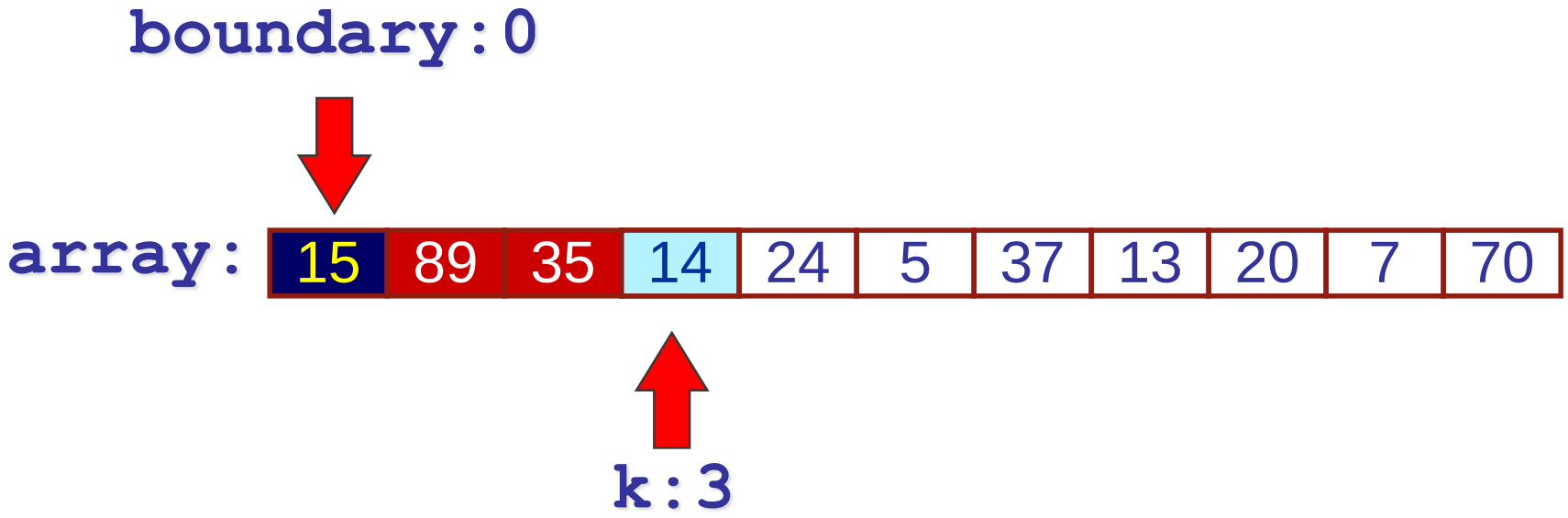
Example: Partition



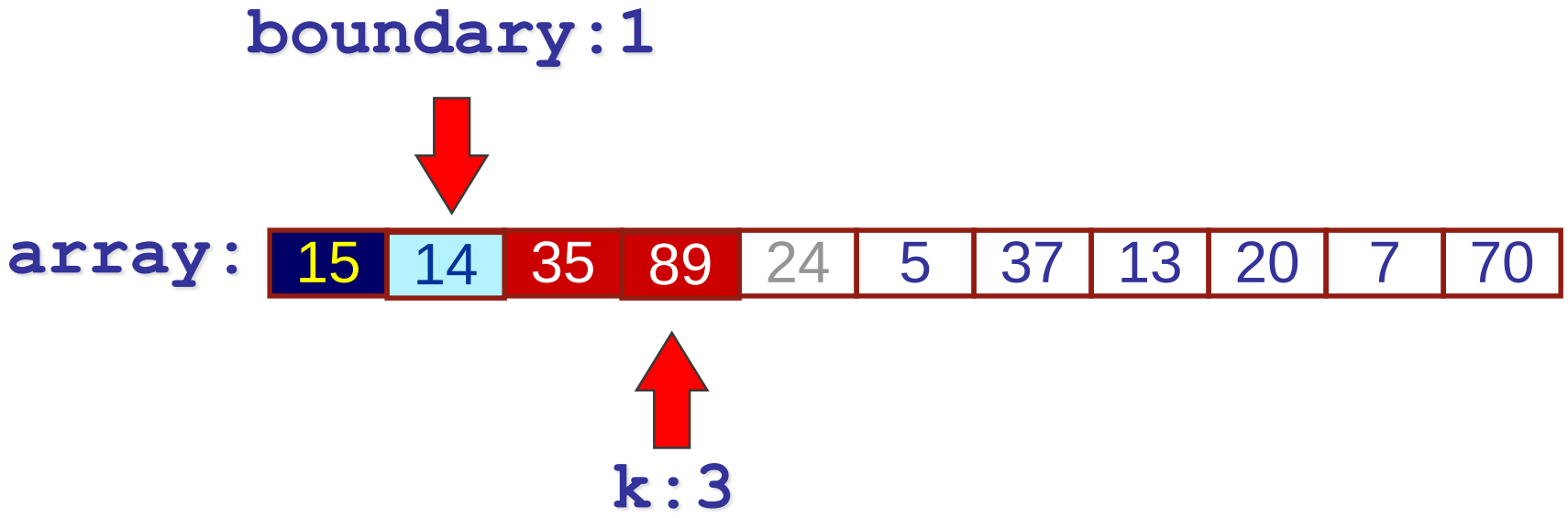
Example: Partition



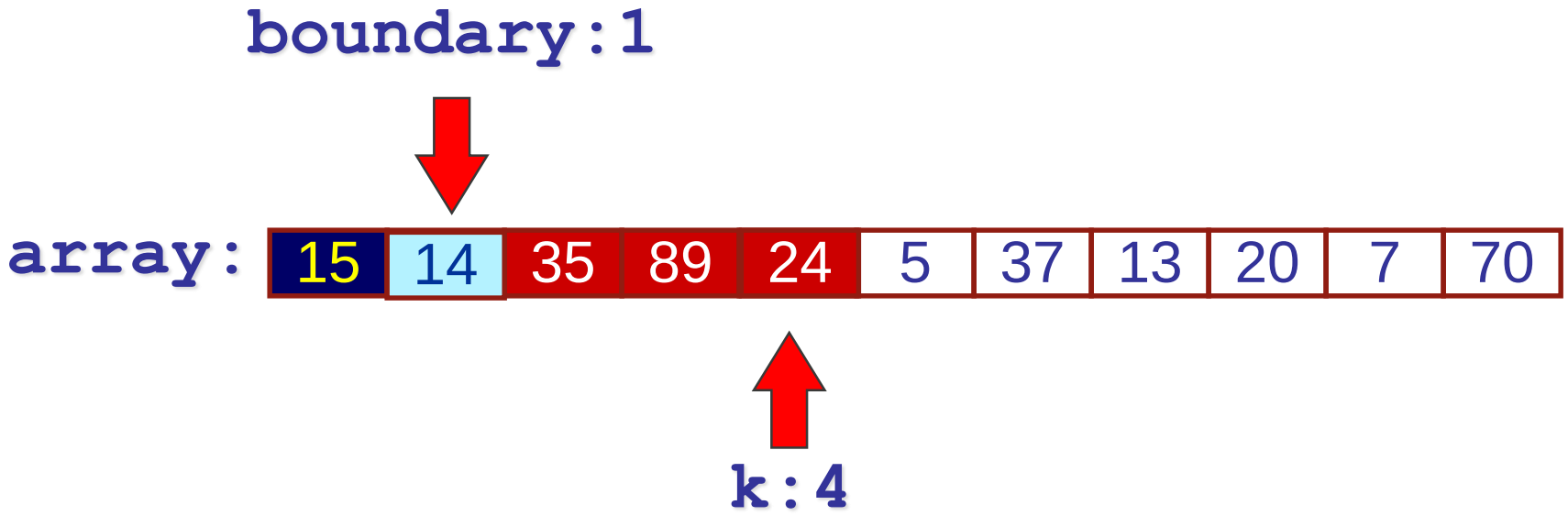
Example: Partition



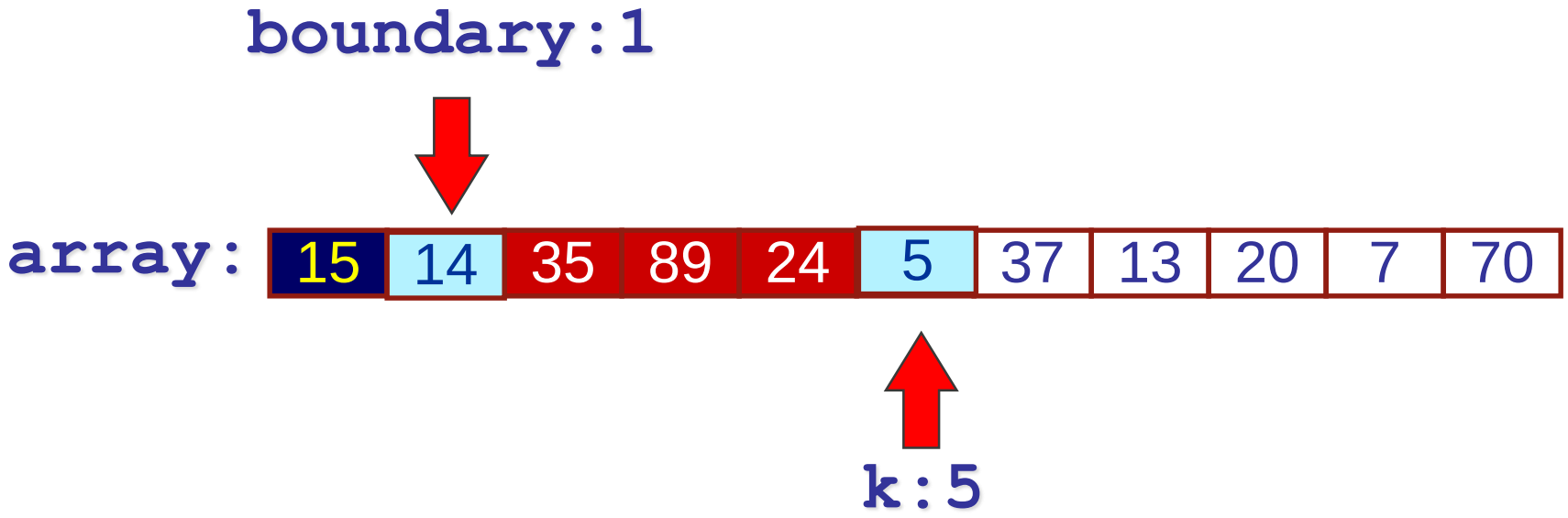
Example: Partition



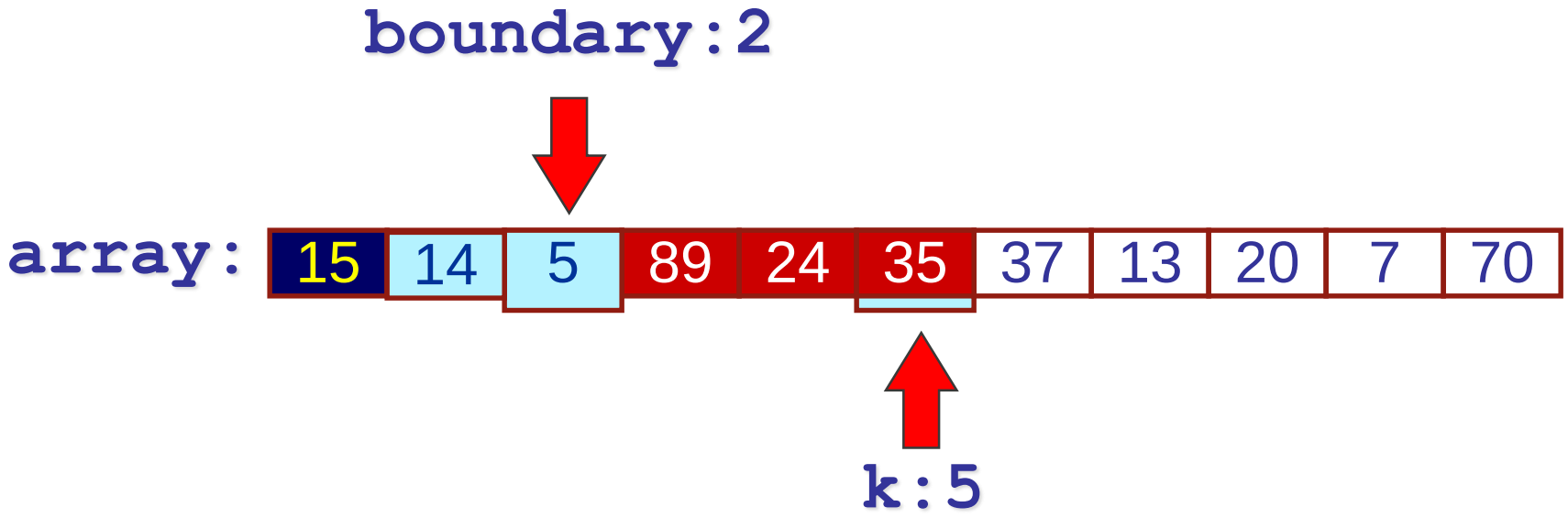
Example: Partition



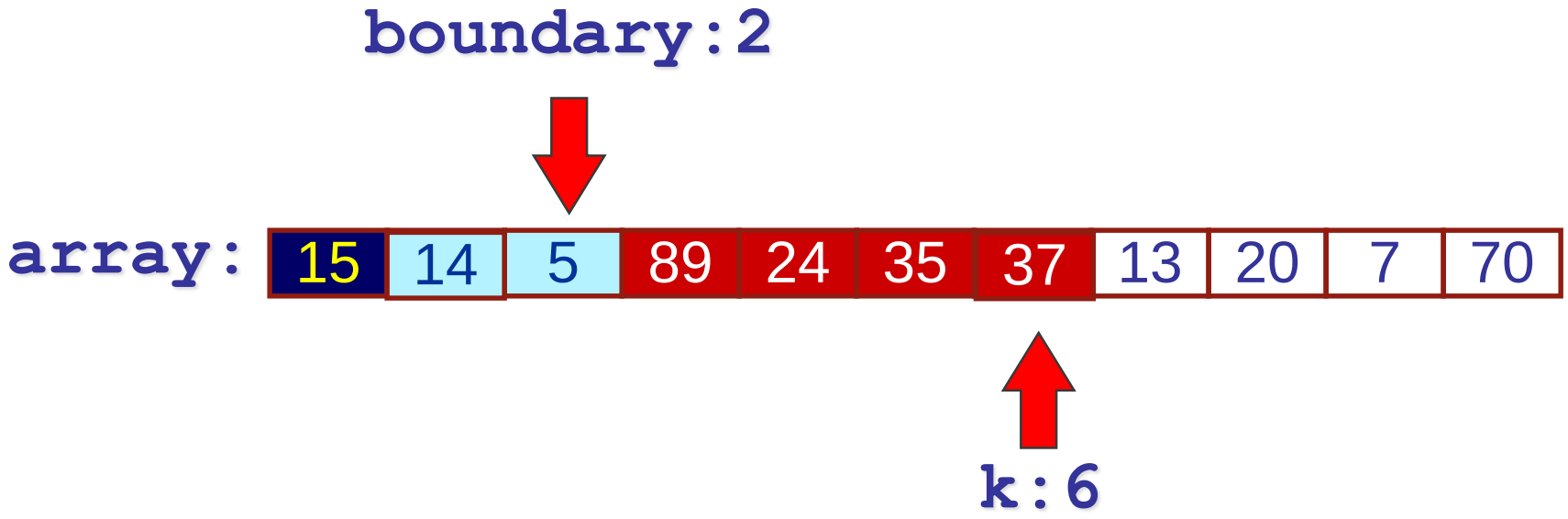
Example: Partition



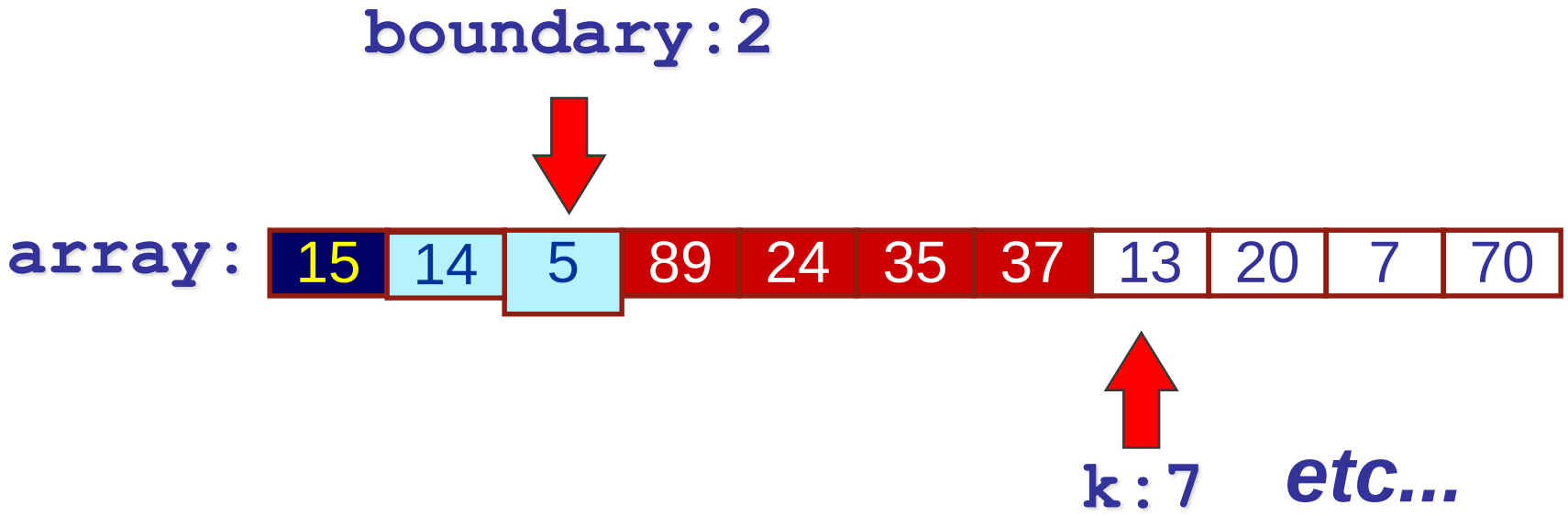
Example: Partition



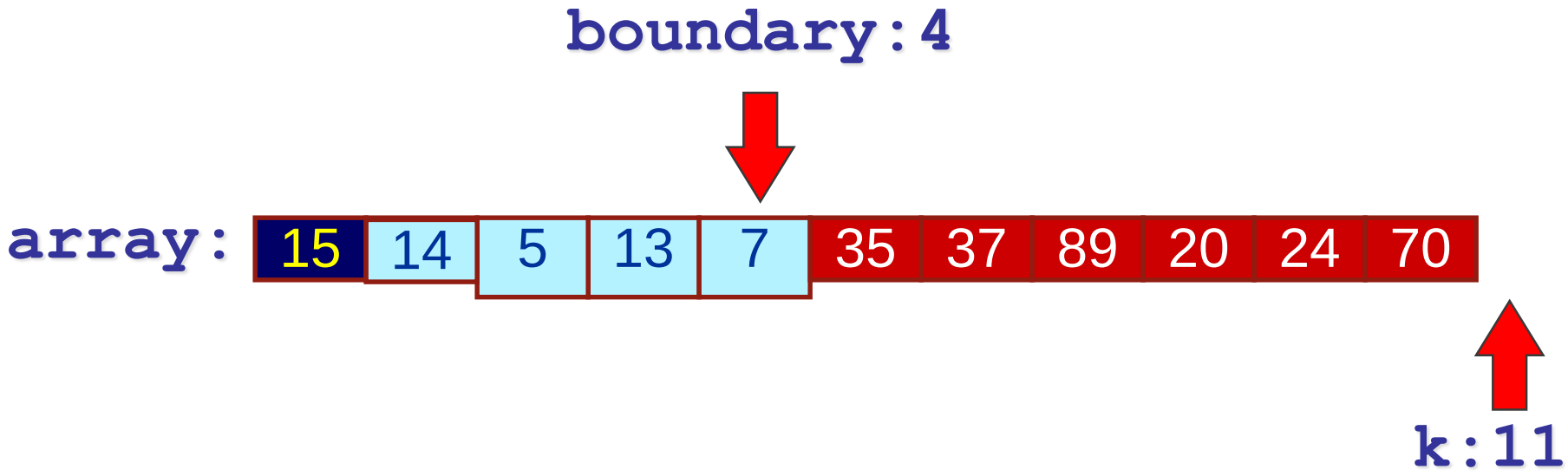
Example: Partition



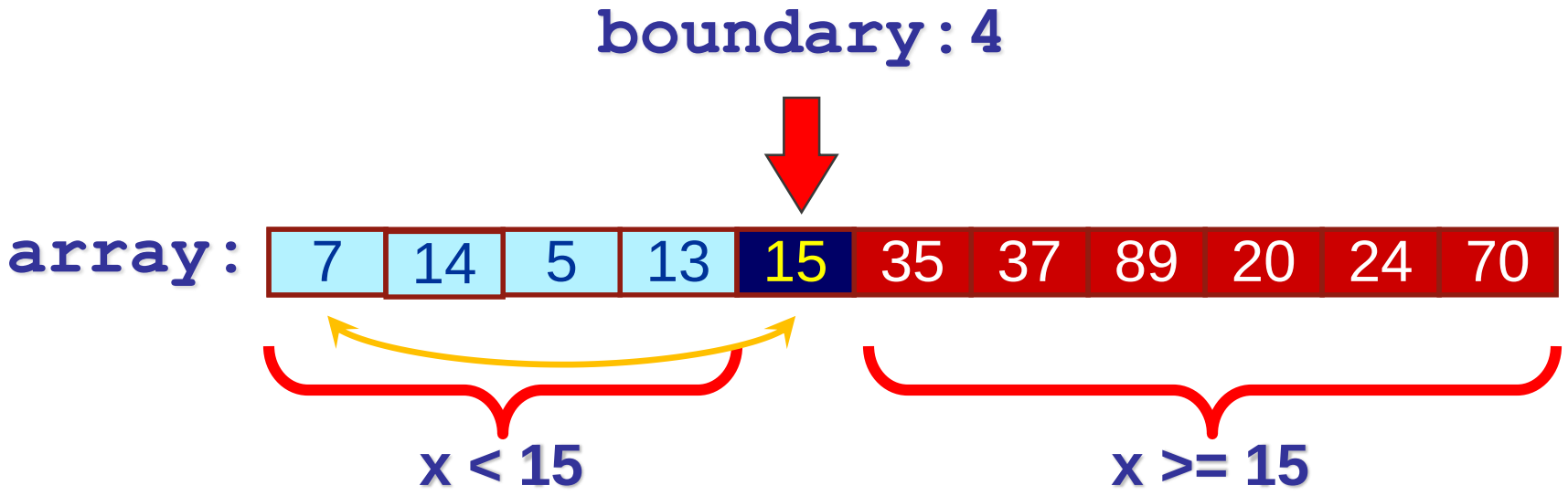
Example: Partition



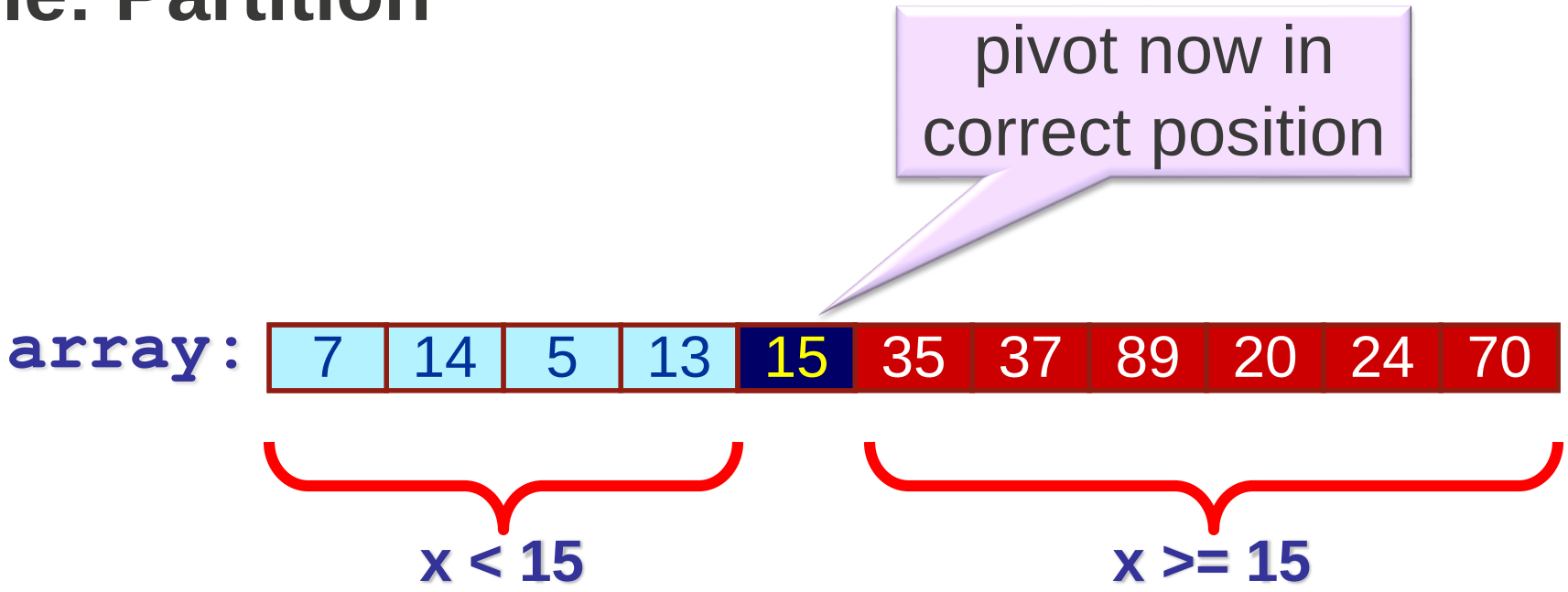
Example: Partition



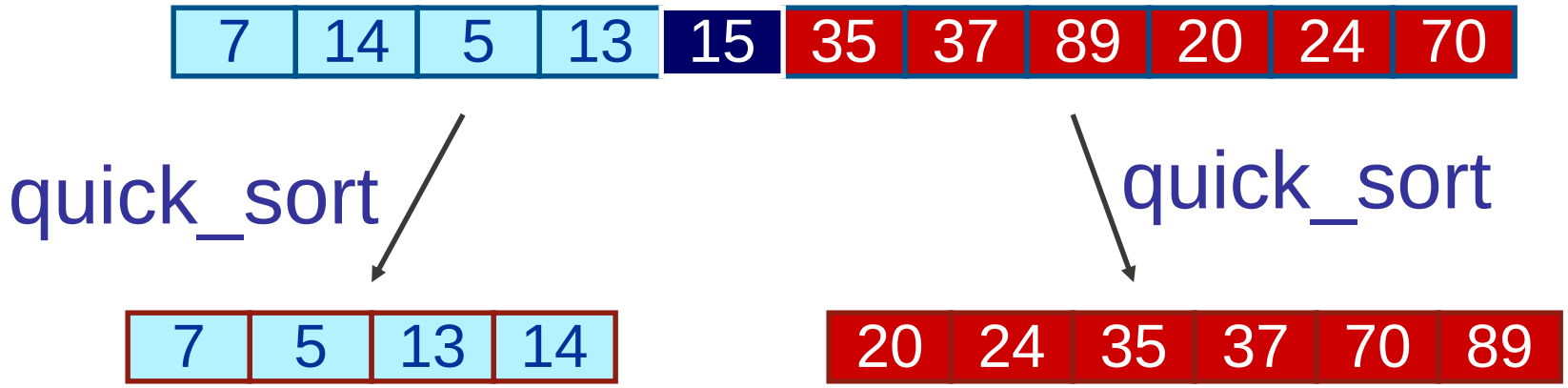
Example: Partition



Example: Partition



Example: Partition

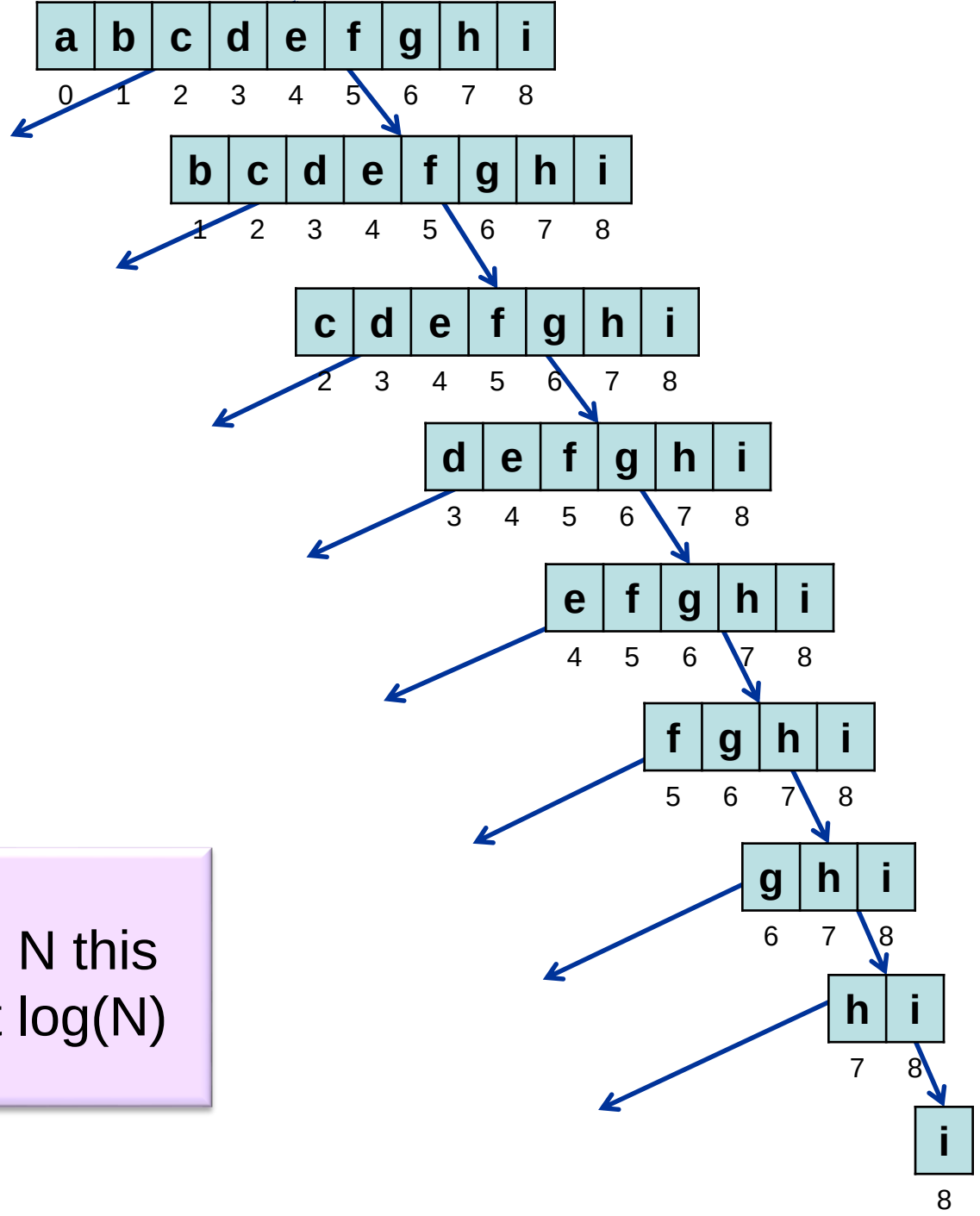


Partition Method

```
def partition(array: ArrayR, start: int, end: int) -> int:
    mid = (start+end)//2
    pivot = array[mid]
    swap(array, start, mid)

    boundary = start

    for k in range(start+1, end+1):
        if array[k] < pivot:
            boundary += 1
            swap(array, k, boundary)

    swap(array, start, boundary)
    return boundary
```

For simplicity, let me choose mid element here…

quick_sort with pivot being the first element



Depth is N this time, not log(N)

Quick Sort Analysis

- **Most of the work done in partitioning**
- **Remember: careful about choice of pivot**
  - Example: 2, 4, 6, 7, 3, 1, 5 what happens if we use the mid element as pivot?
  - Often used: median among 3 random
- **Complexity?**
  - Best case takes **O(n log(n))*CompEq** time
    - Always pick median as pivot
  - Worst case takes **$O(n^2)$*CompEq** time
    - Always pick min/max as pivot
- **Main advantage over merge ksort:**
  - no need to copy back (so quicksort has a smaller constant if the pivot is good)
- **Still, for small arrays (≤20) it has significant overhead (due to the recursion); for those better use insertion sort**

Summary

- **Divide and Conquer algorithms**
  - Relationship with recursive algorithms
- **Merge Sort**
  - Easy split
  - Complex part: merge method
- **Quick Sort**
  - Complex split: partition method
  - Easy combination