Bubble sort

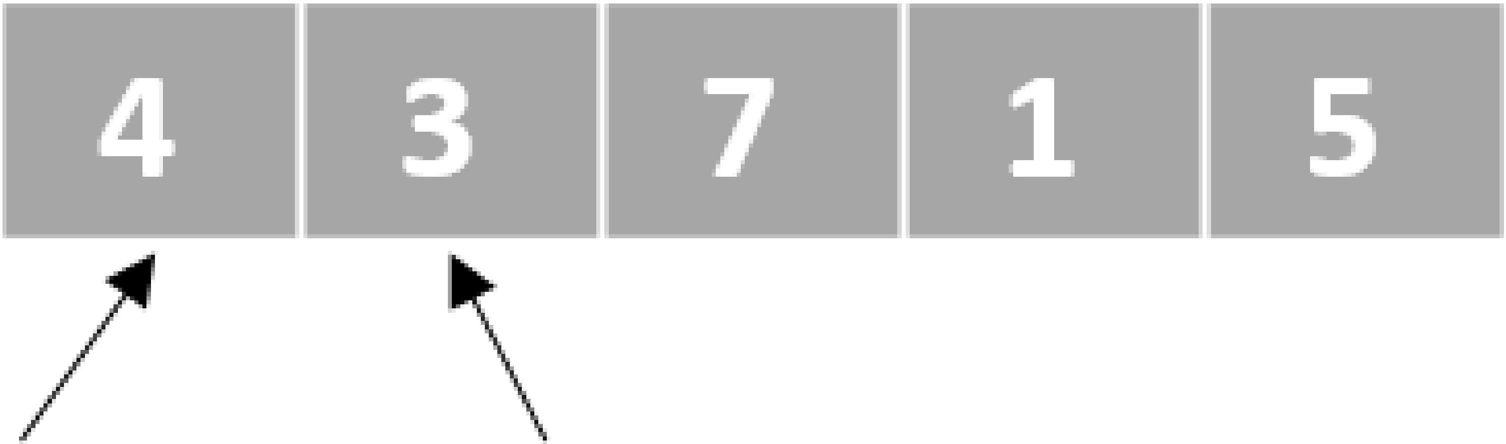
DATA STRUCTURES AND ALGORITHMS IN PYTHON

Miriam Antona
Software engineer

Sorting algorithms

- Deeply studied
- Solve how to **sort** an **unsorted collection** in **ascending/descending** order
- Can **reduce complexity** of problems
- Some sorting algorithms:
  - bubble sort
  - selection sort
  - insertion sort
  - merge sort
  - quicksort

Bubble sort

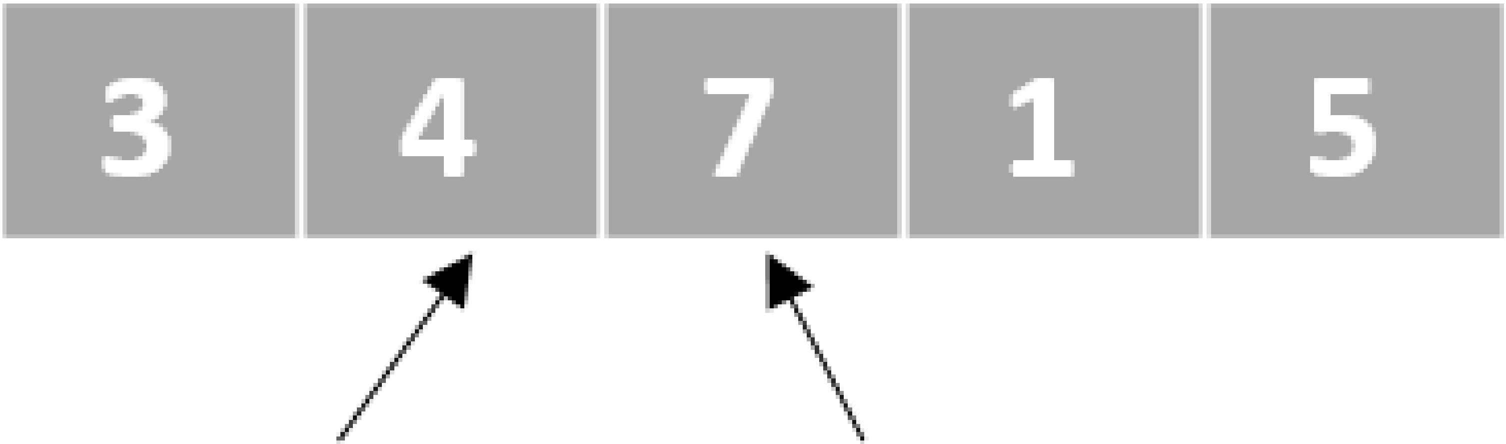


- First value greater than the second value

Bubble sort

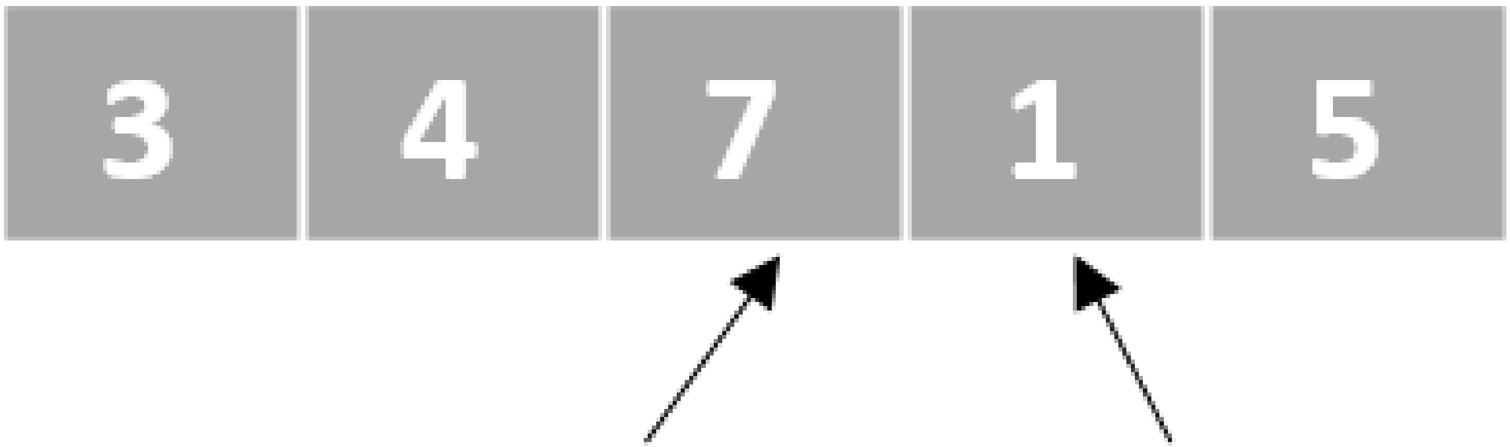
- First value greater than the second value
  - Swap them
- Second value greater than the first value
  - Nothing

Bubble sort



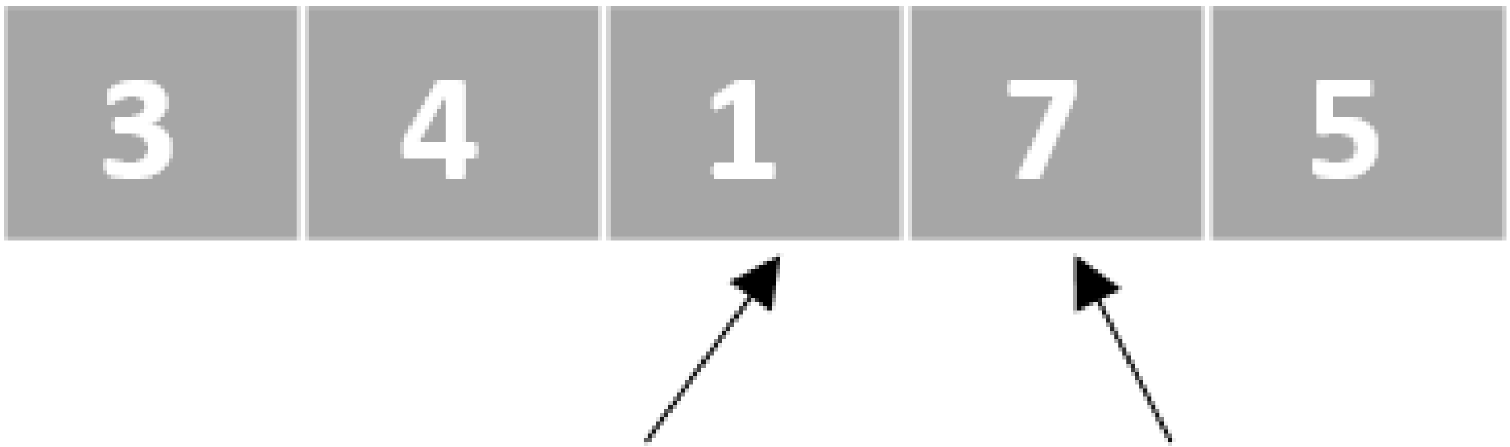
- First value greater than the second value
  - Swap them
- Second value greater than the first value
  - Nothing

Bubble sort



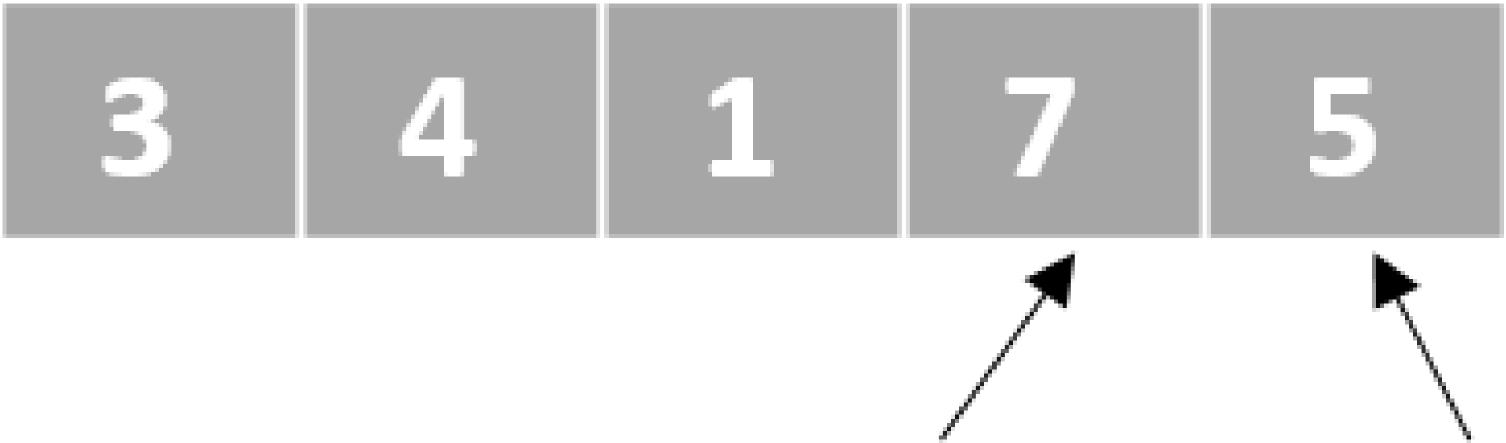
- First value greater than the second value
  - Swap them
- Second value greater than the first value
  - Nothing

Bubble sort



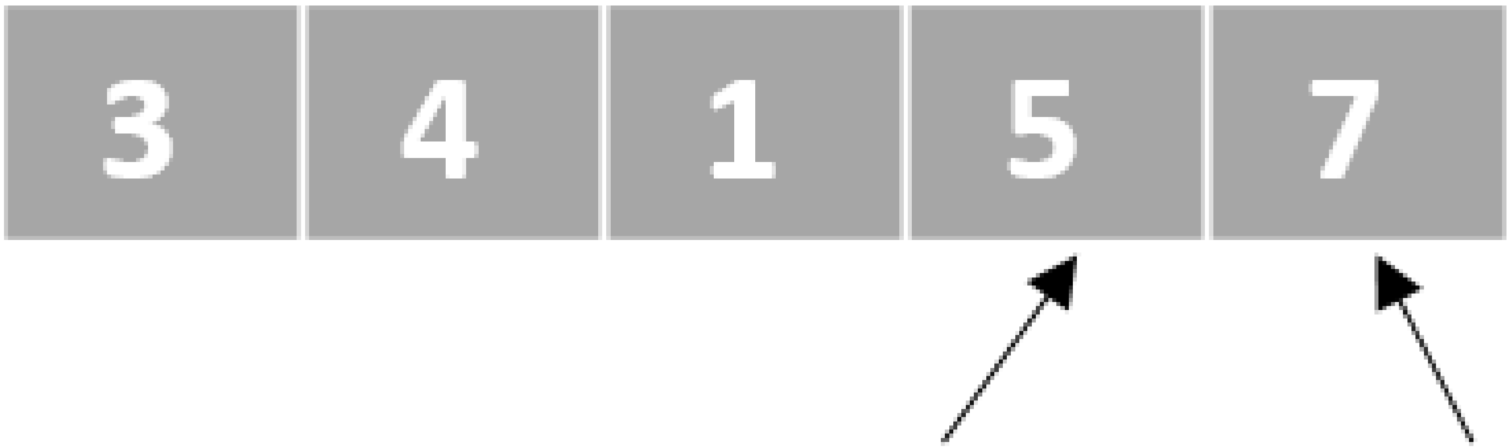
- First value greater than the second value
  - Swap them
- Second value greater than the first value
  - Nothing

Bubble sort



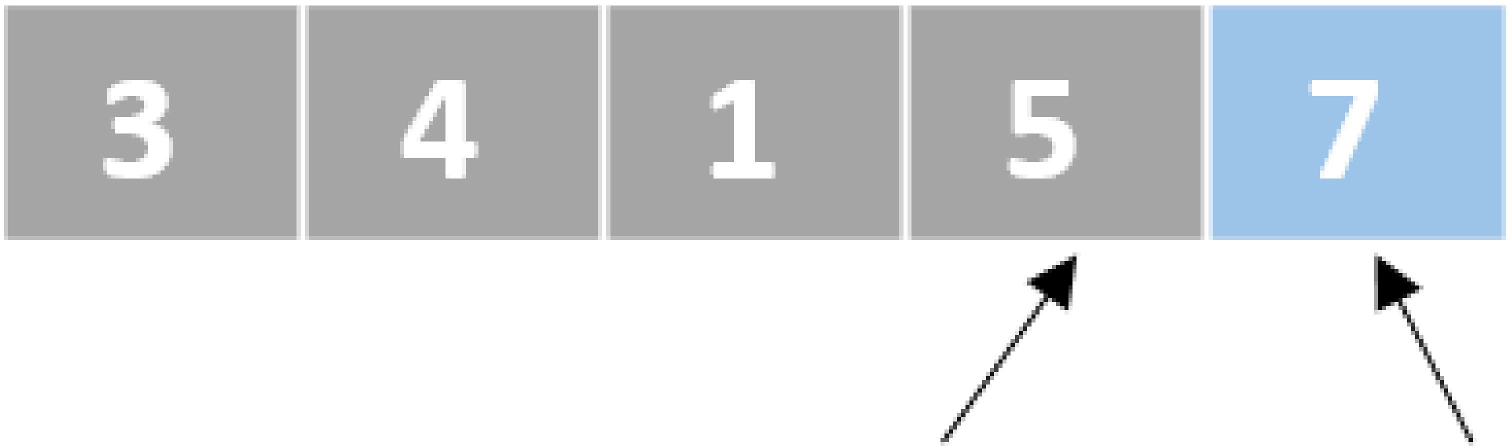
- First value greater than the second value
  - Swap them
- Second value greater than the first value
  - Nothing

Bubble sort



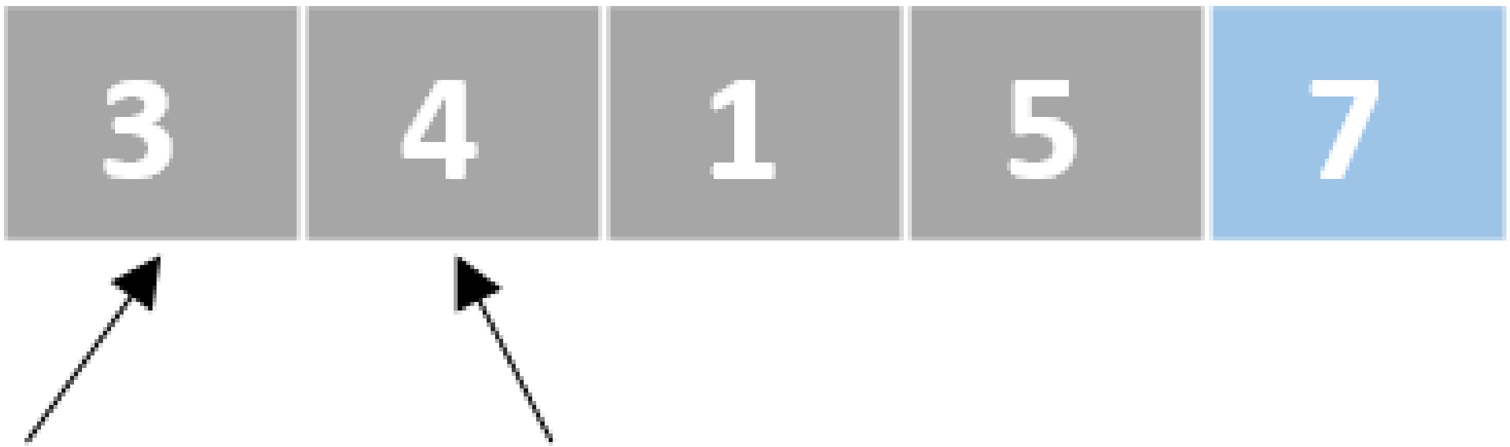
- First value greater than the second value
  - Swap them
- Second value greater than the first value
  - Nothing

Bubble sort



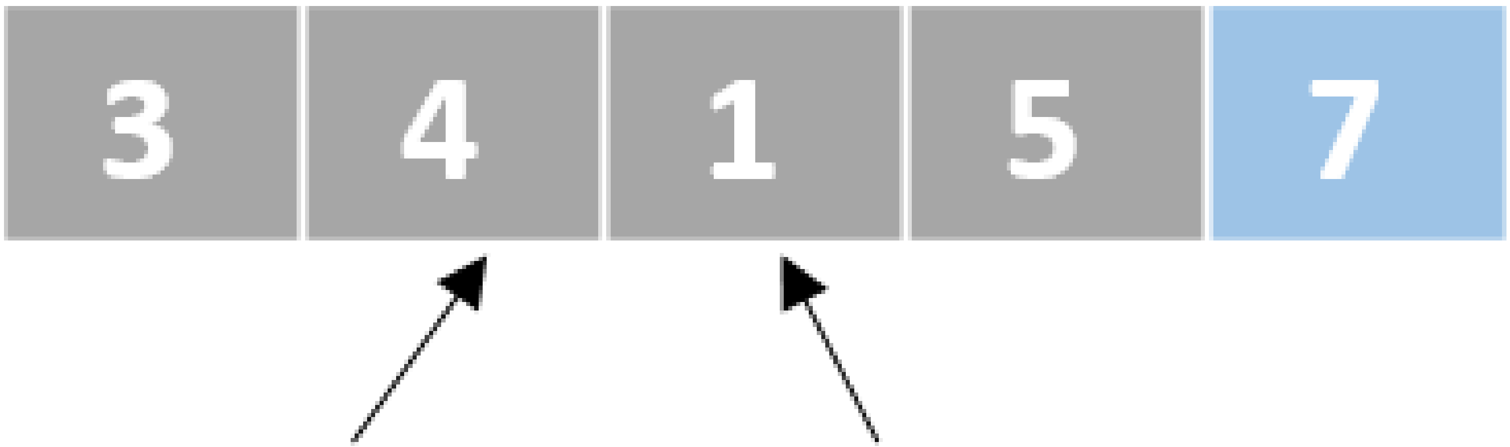
- First value greater than the second value
  - Swap them
- Second value greater than the first value
  - Nothing

Bubble sort



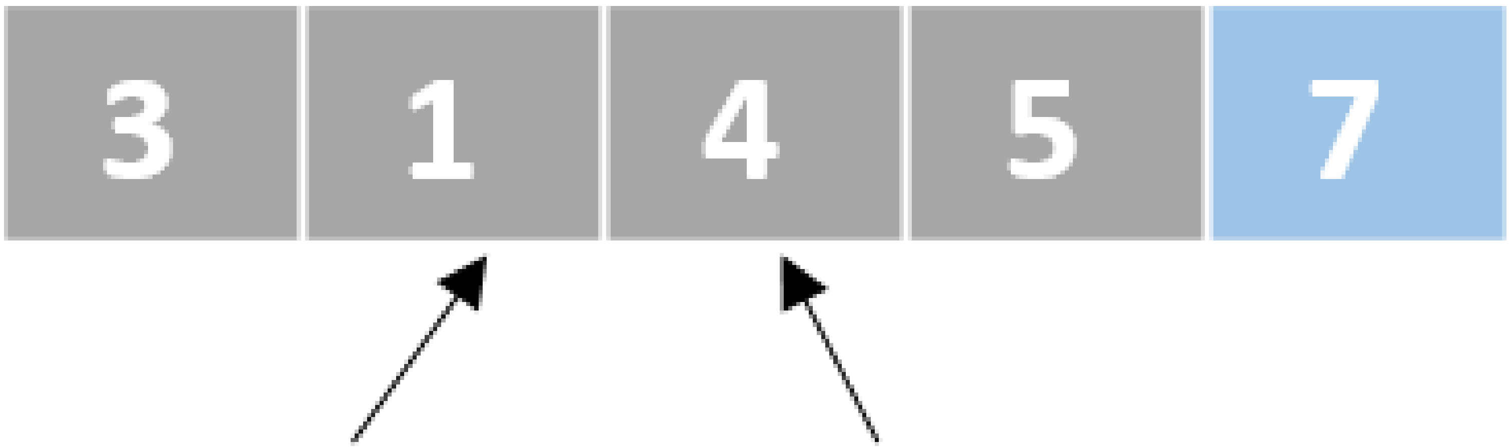
- First value greater than the second value
  - Swap them
- Second value greater than the first value
  - Nothing

Bubble sort



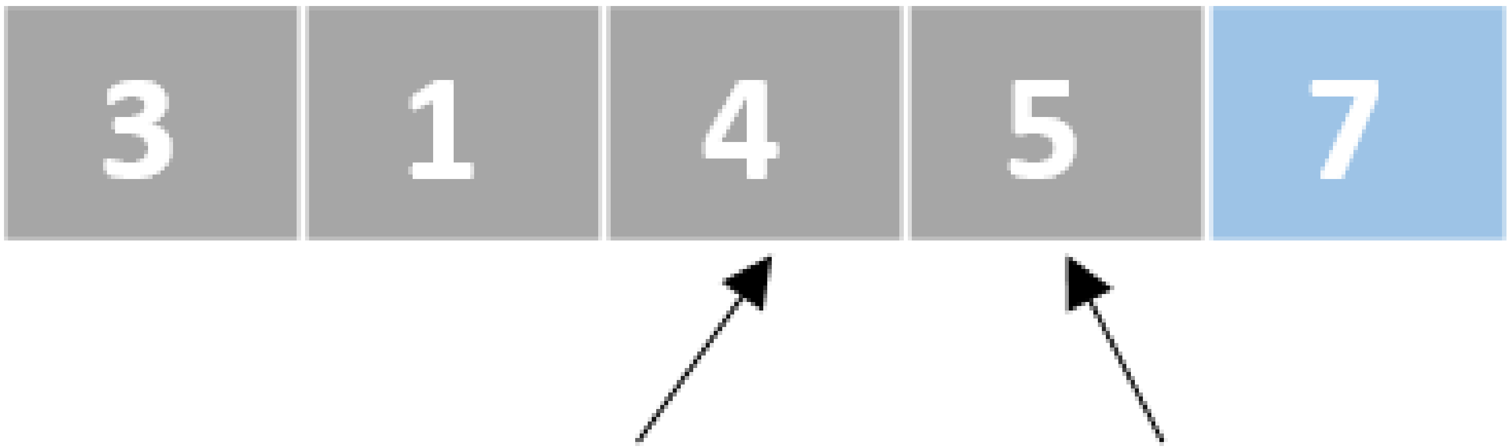
- First value greater than the second value
  - Swap them
- Second value greater than the first value
  - Nothing

Bubble sort



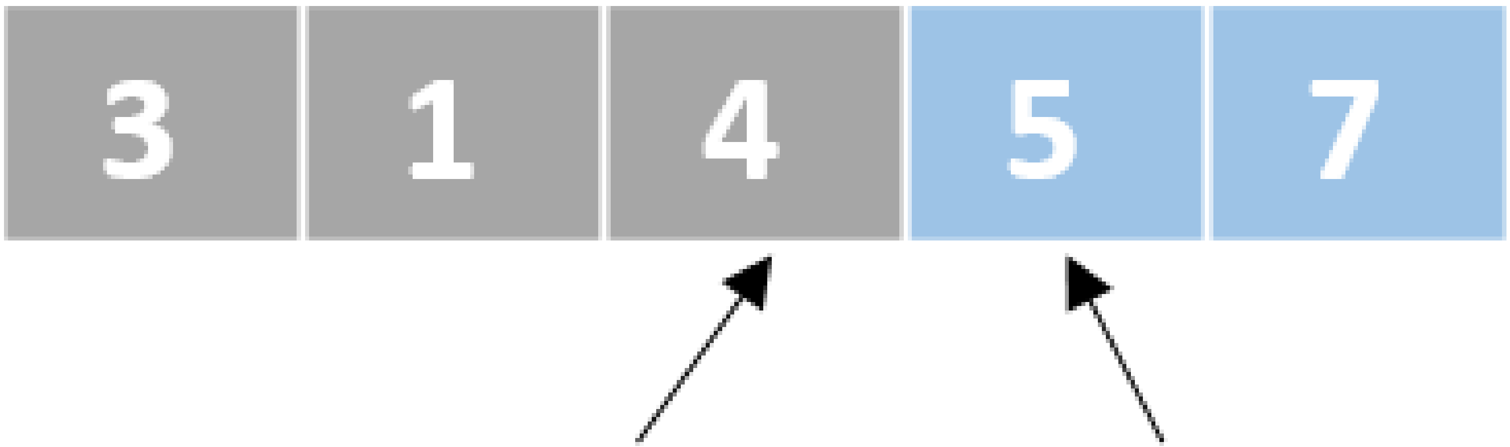
- First value greater than the second value
  - Swap them
- Second value greater than the first value
  - Nothing

Bubble sort



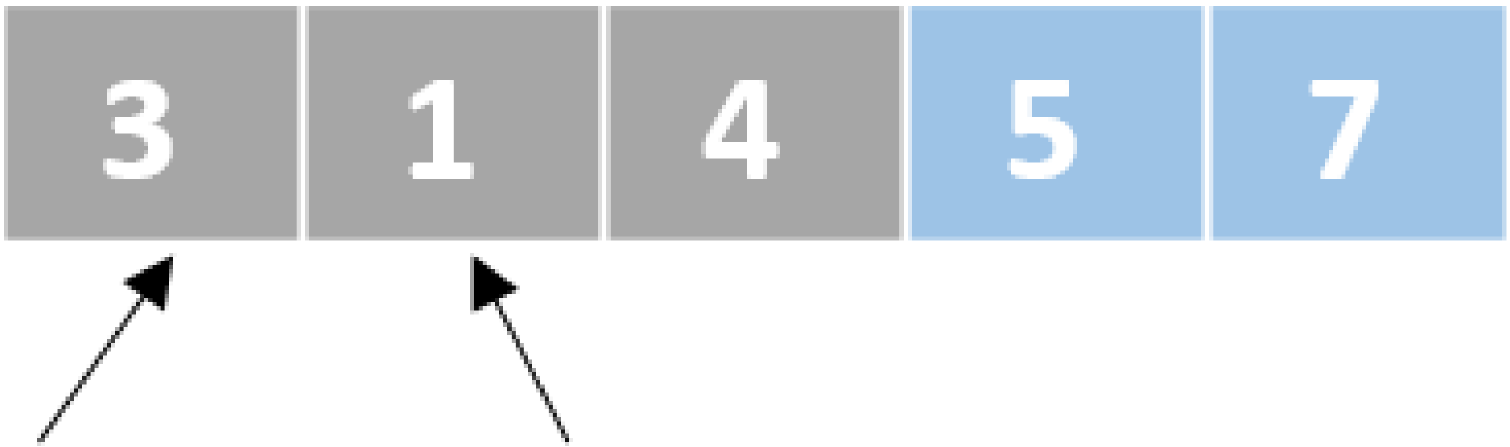
- First value greater than the second value
  - Swap them
- Second value greater than the first value
  - Nothing

Bubble sort



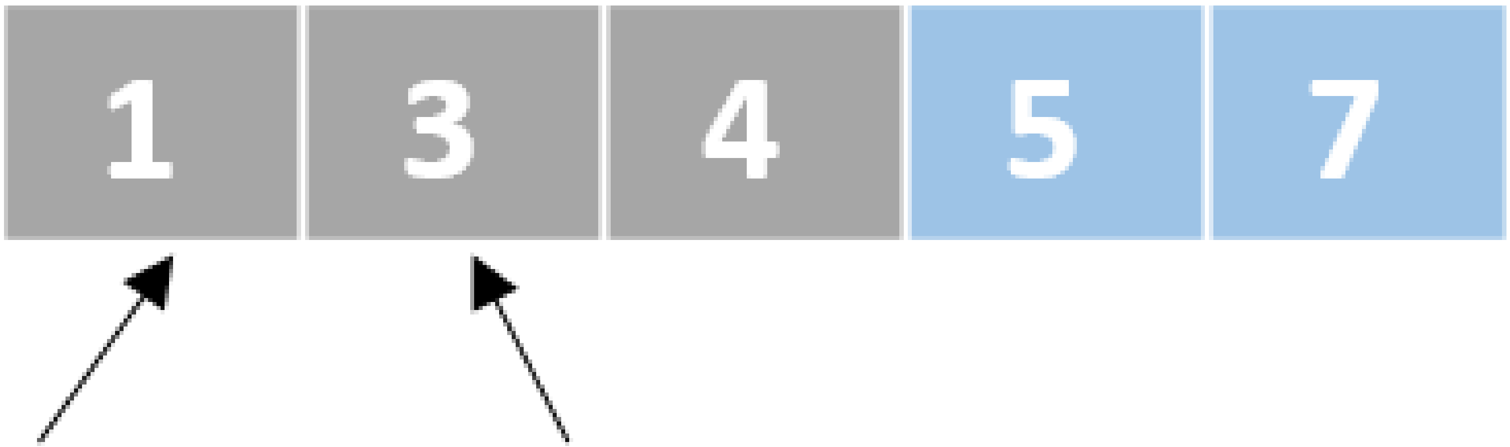
- First value greater than the second value
  - Swap them
- Second value greater than the first value
  - Nothing

Bubble sort



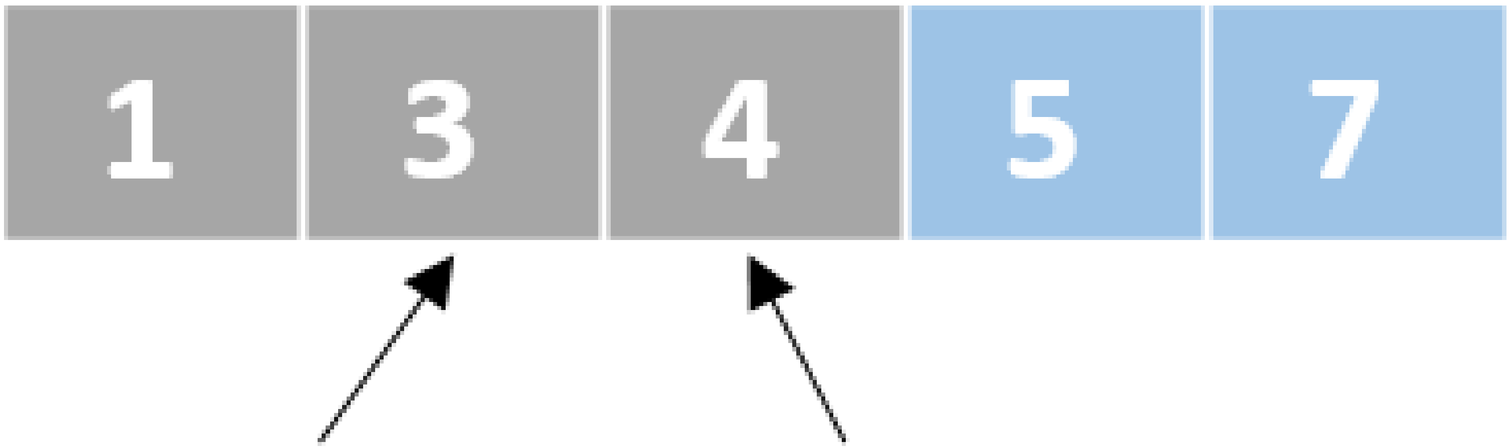
- First value greater than the second value
  - Swap them
- Second value greater than the first value
  - Nothing

Bubble sort



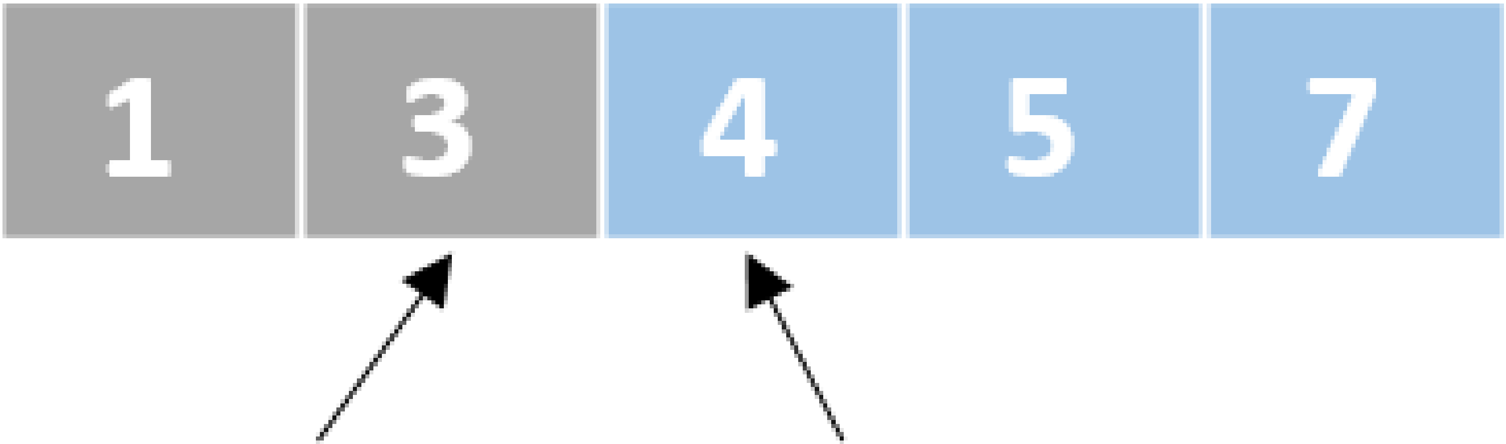
- First value greater than the second value
  - Swap them
- Second value greater than the first value
  - Nothing

Bubble sort



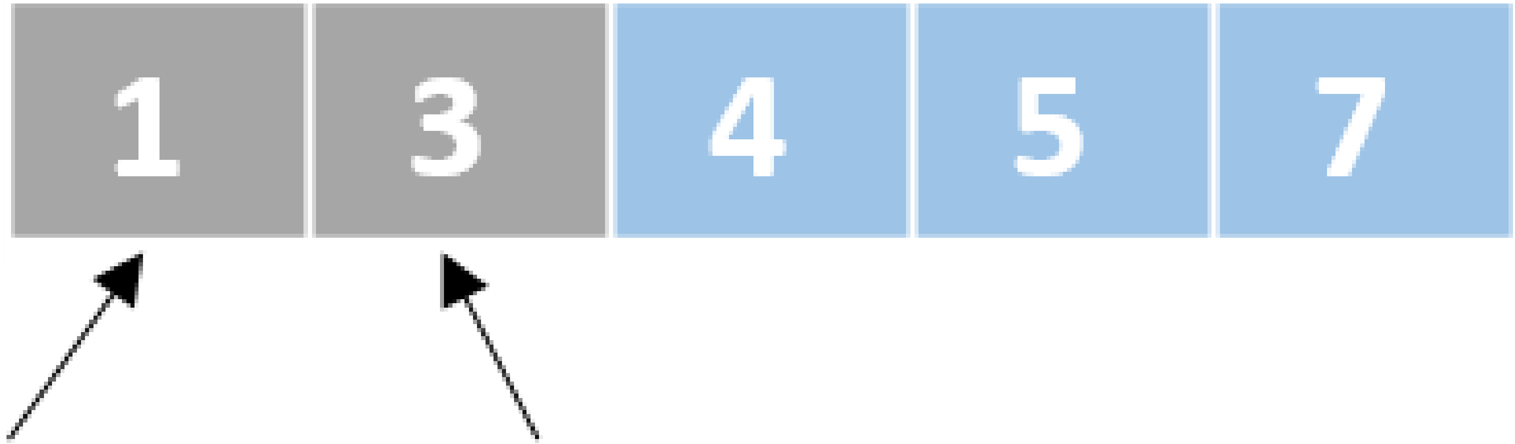
- First value greater than the second value
  - Swap them
- Second value greater than the first value
  - Nothing

Bubble sort



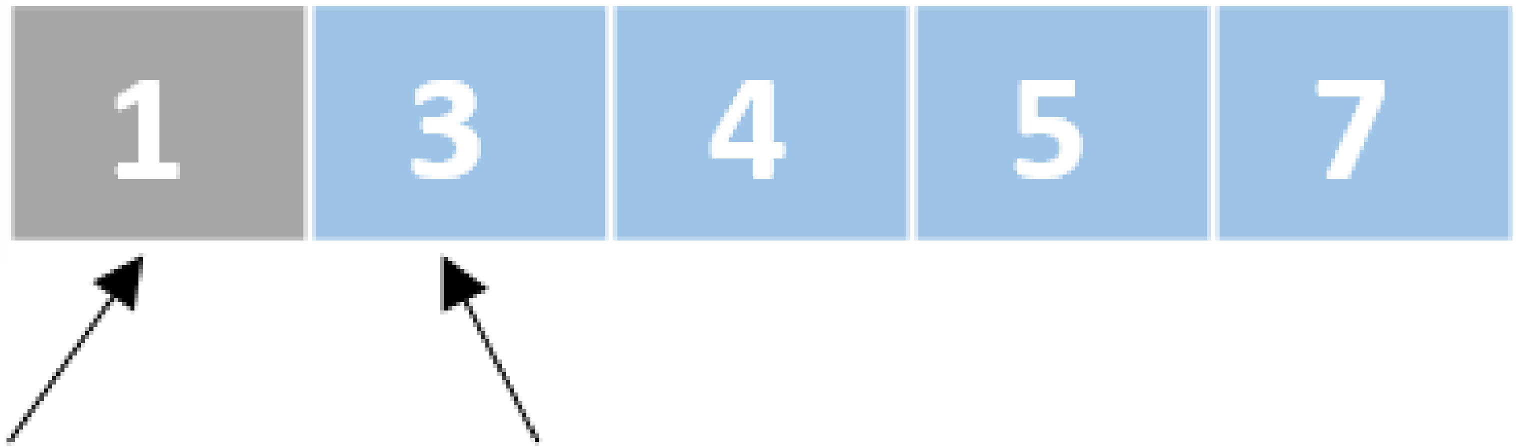
- First value greater than the second value
  - Swap them
- Second value greater than the first value
  - Nothing

Bubble sort



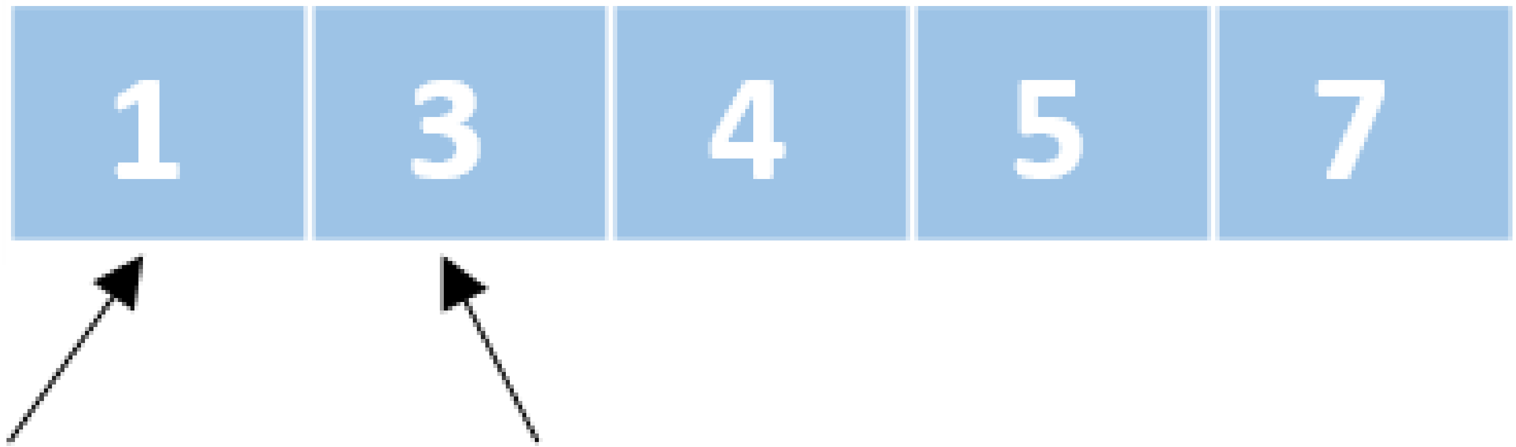
- First value greater than the second value
  - Swap them
- Second value greater than the first value
  - Nothing

Bubble sort



- First value greater than the second value
  - Swap them
- Second value greater than the first value
  - Nothing

Bubble sort



- First value greater than the second value
  - Swap them
- Second value greater than the first value
  - Nothing

Bubble sort - implementation

```python
def bubble_sort(my_list):
  list_length = len(my_list)
  for i in range(list_length-1):
    for j in range(list_length-1-i):
      if my_list[j] > my_list[j+1]:
        my_list[j] , my_list[j+1] = my_list[j+1] , my_list[j]
  return my_list
```

```python
print(bubble_sort([4,3,7,1,5]))
```

```
[1, 3, 4, 5, 7]
```

Bubble sort - implementation

```
def bubble_sort(my_list):
  list_length = len(my_list)
  is_sorted = False
  while not is_sorted:
    is_sorted = True
    for i in range(list_length-1):
      if my_list[i] > my_list[i+1]:
        my_list[i] , my_list[i+1] = my_list[i+1] , my_list[i]
        is_sorted = False
    list_length -= 1
  return my_list
```

Bubble sort - complexity

- Worst case: $O(n^2)$
- Best case - not improved version: $\Omega(n^2)$
- Best case - improved version: $\Omega(n)$
- Average case: $\Theta(n^2)$
- Doesn't perform well with highly unsorted large lists
- Performs well:
  - large sorted/almost sorted lists
  - small lists

Let's practice!

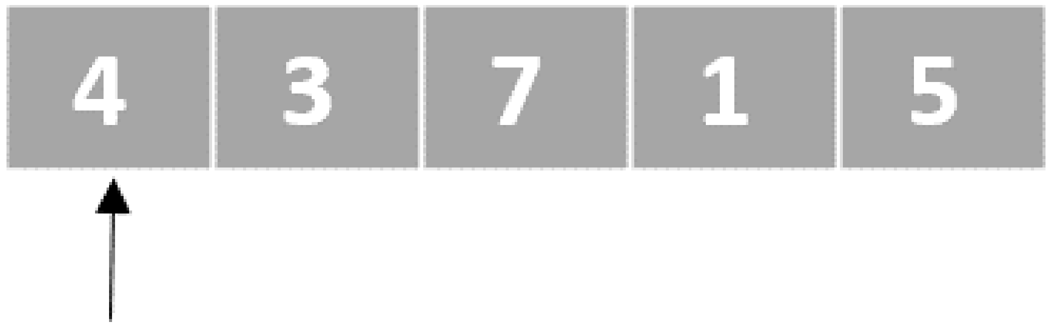
DATA STRUCTURES AND ALGORITHMS IN PYTHON

Selection Sort and Insertion Sort

DATA STRUCTURES AND ALGORITHMS IN PYTHON

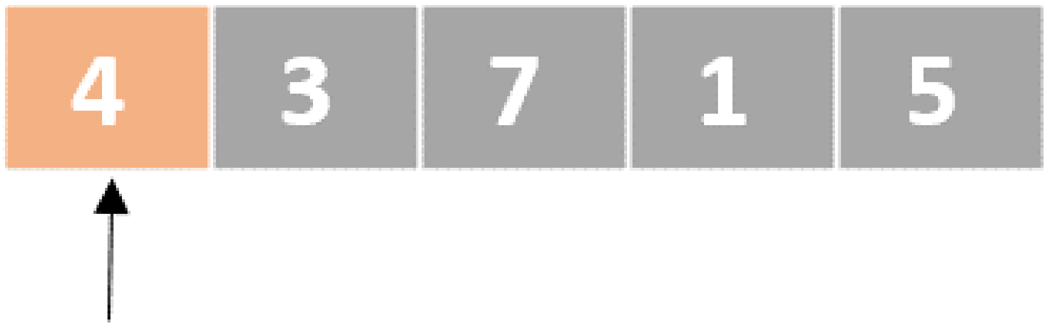
Miriam Antona
Software engineer

Selection sort



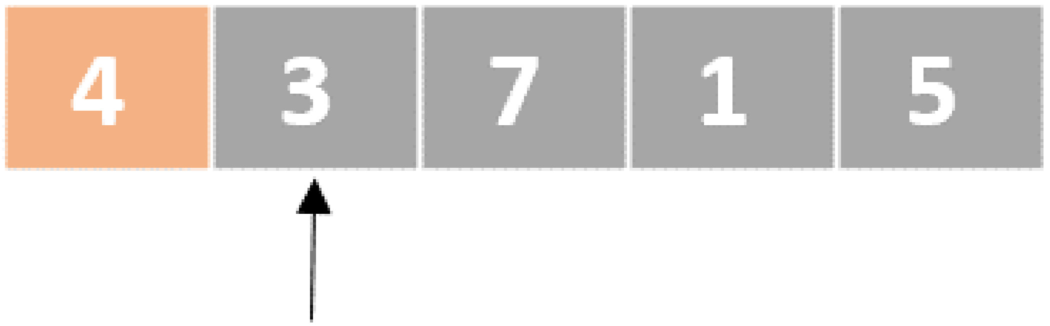
- Determine the lowest value

Selection sort



- Determine the lowest value

Selection sort

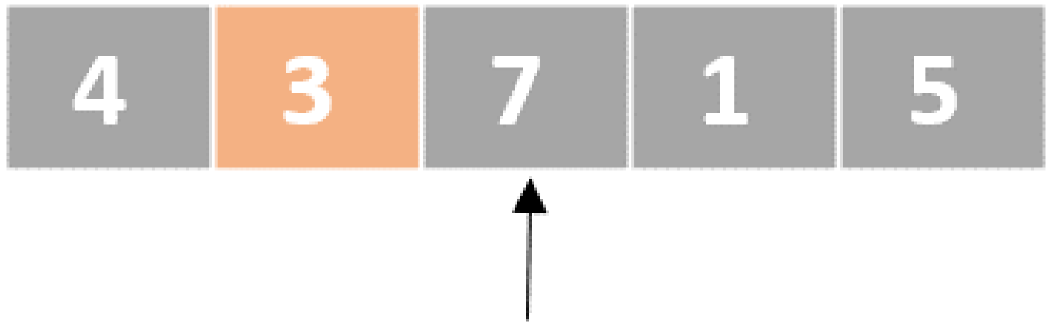


- Determine the lowest value

Selection sort

- Determine the lowest value

Selection sort



- Determine the lowest value

Selection sort

- Determine the lowest value

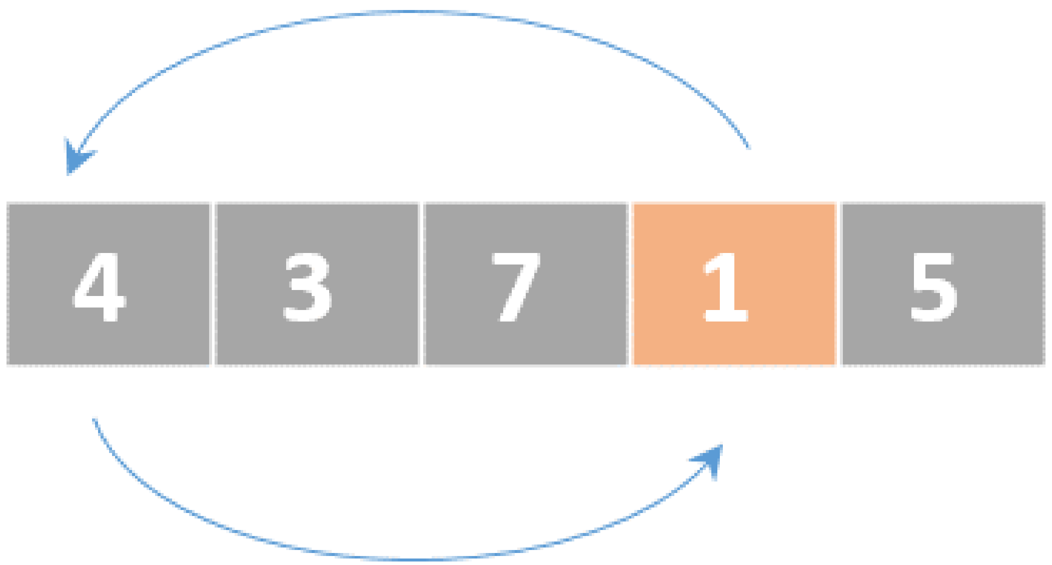
Selection sort

- Determine the lowest value

Selection sort

- Determine the lowest value

Selection sort



- Determine the lowest value
- Swap the lowest value with the first unordered element

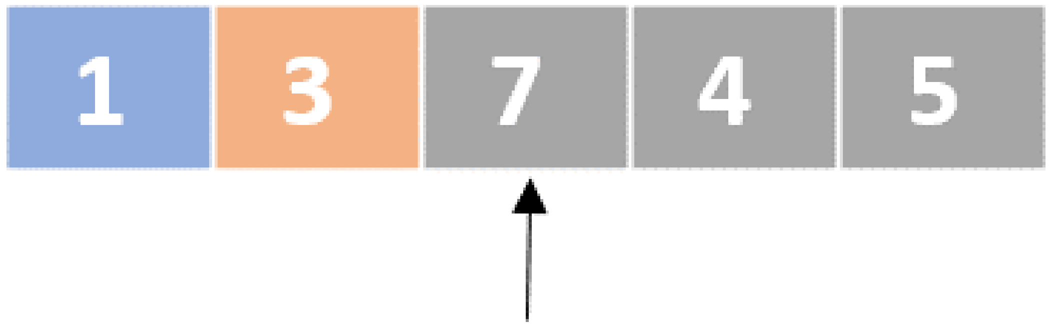
Selection sort

- Determine the lowest value
- Swap the lowest value with the first unordered element

Selection sort

- Determine the lowest value
- Swap the lowest value with the first unordered element

Selection sort



- Determine the lowest value
- Swap the lowest value with the first unordered element

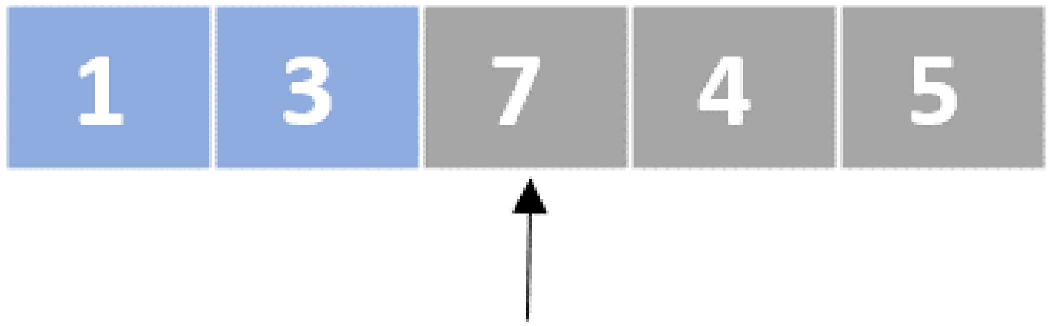
Selection sort

- Determine the lowest value
- Swap the lowest value with the first unordered element

Selection sort

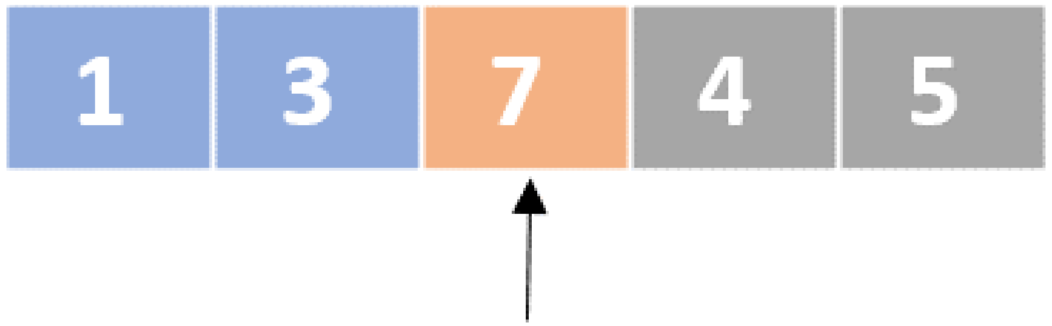
- Determine the lowest value
- Swap the lowest value with the first unordered element

Selection sort



- Determine the lowest value
- Swap the lowest value with the first unordered element

Selection sort



- Determine the lowest value
- Swap the lowest value with the first unordered element

Selection sort

- Determine the lowest value
- Swap the lowest value with the first unordered element

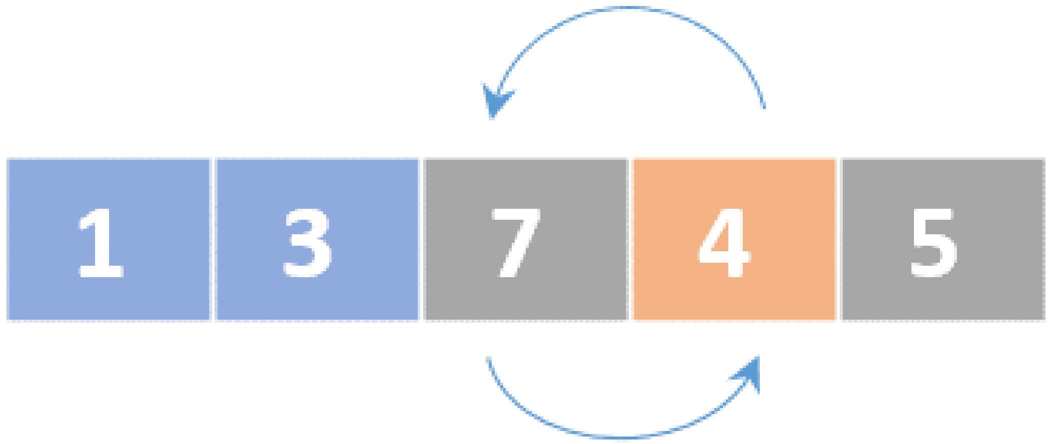
Selection sort

- Determine the lowest value
- Swap the lowest value with the first unordered element

Selection sort

- Determine the lowest value
- Swap the lowest value with the first unordered element

Selection sort

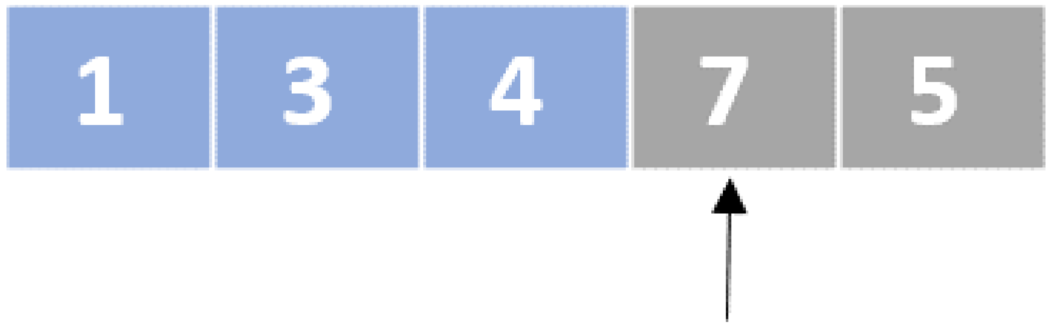


- Determine the lowest value
- Swap the lowest value with the first unordered element

Selection sort

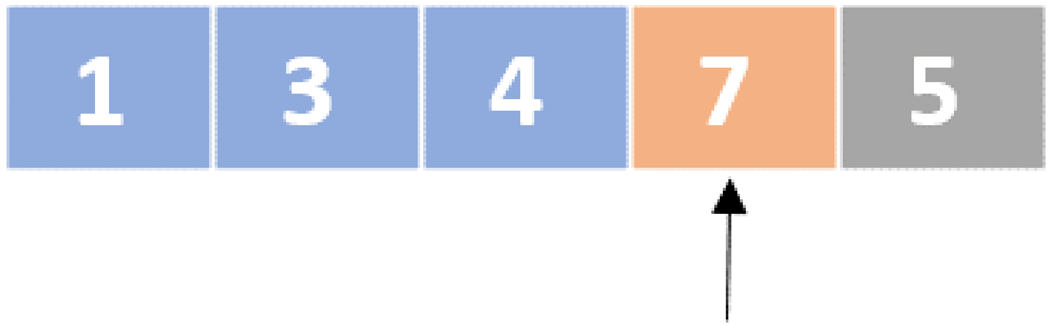
- Determine the lowest value
- Swap the lowest value with the first unordered element

Selection sort



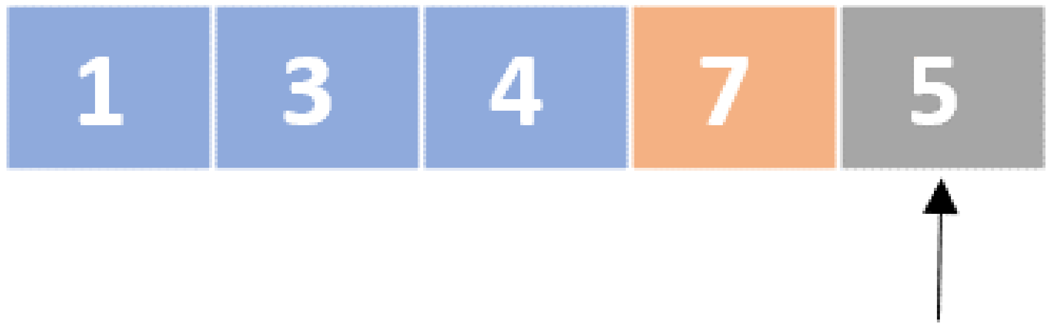
- Determine the lowest value
- Swap the lowest value with the first unordered element

Selection sort



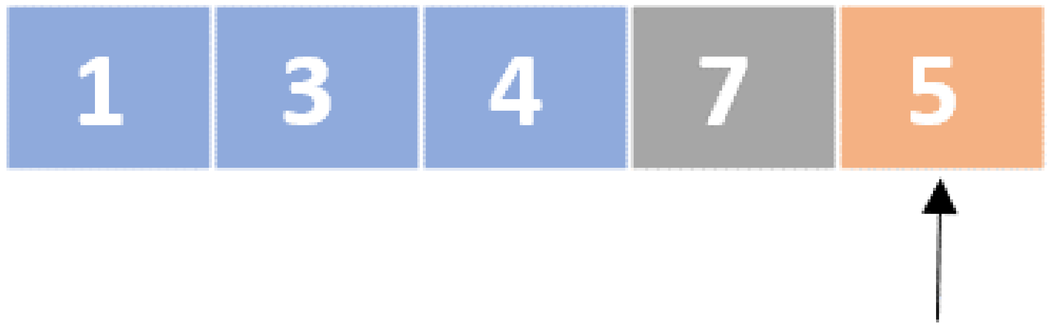
- Determine the lowest value
- Swap the lowest value with the first unordered element

Selection sort



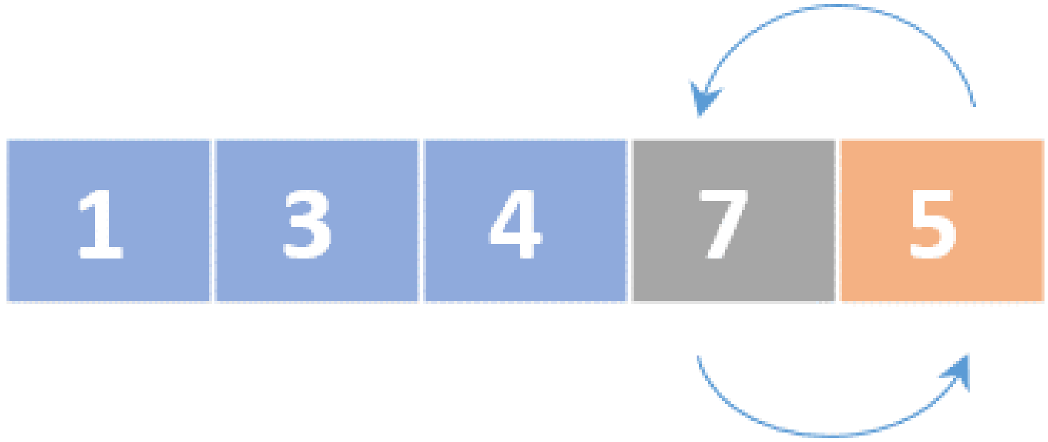
- Determine the lowest value
- Swap the lowest value with the first unordered element

Selection sort



- Determine the lowest value
- Swap the lowest value with the first unordered element

Selection sort



- Determine the lowest value
- Swap the lowest value with the first unordered element

Selection sort

- Determine the lowest value
- Swap the lowest value with the first unordered element

Selection sort

- Determine the lowest value
- Swap the lowest value with the first unordered element

Selection sort

- Determine the lowest value
- Swap the lowest value with the first unordered element

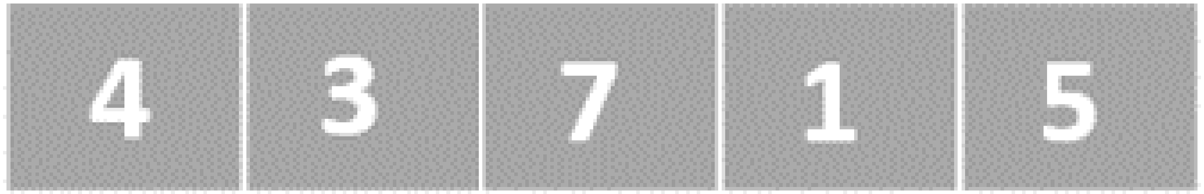
Selection sort - implementation

```python
def selection_sort(my_list):
  list_length = len(my_list)
  for i in range(list_length - 1):
    lowest = my_list[i]
    index = i
    for j in range(i + 1, list_length):
      if my_list[j] < lowest:
        index = j
        lowest = my_list[j]
    my_list[i] , my_list[index] = my_list[index] , my_list[i]
  return my_list
```

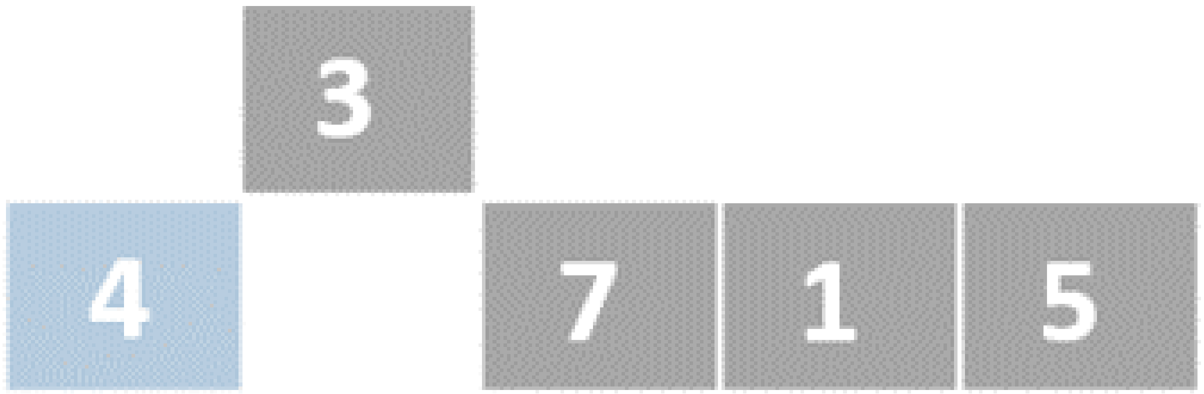
Selection sort - complexity

- Worst case: $O(n^2)$
- Average case: $\Theta(n^2)$
- Best case: $\Omega(n^2)$

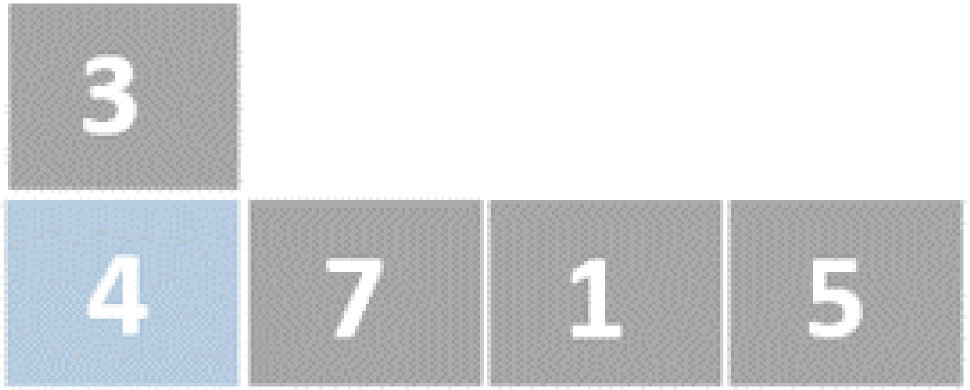
Insertion sort



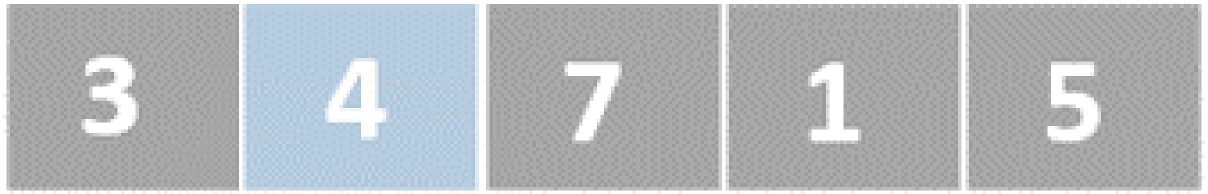
Insertion sort



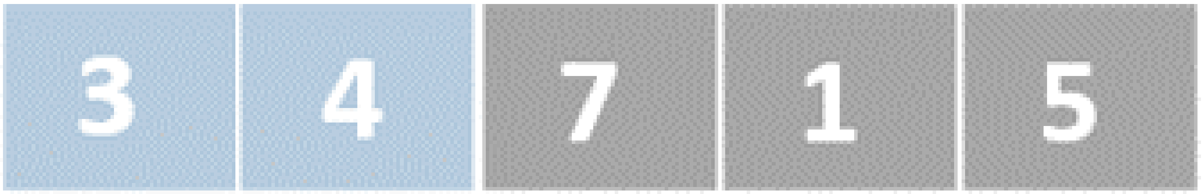
Insertion sort



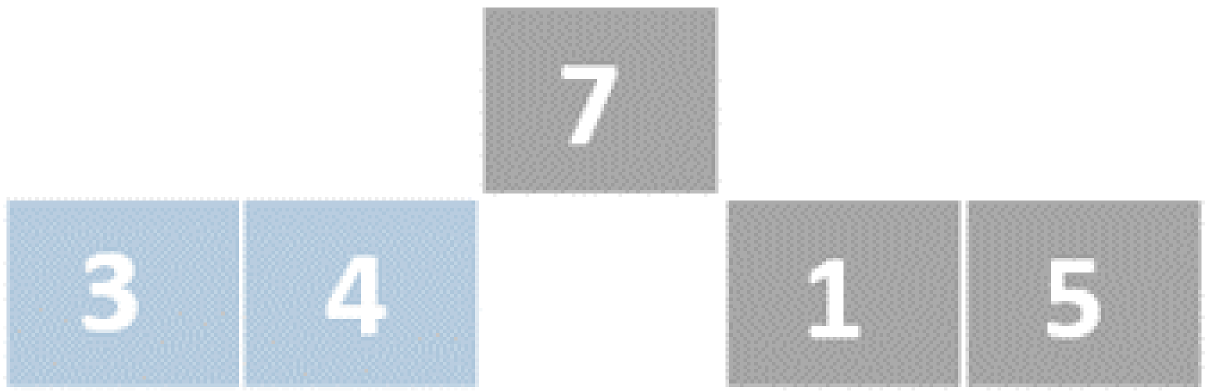
Insertion sort



Insertion sort

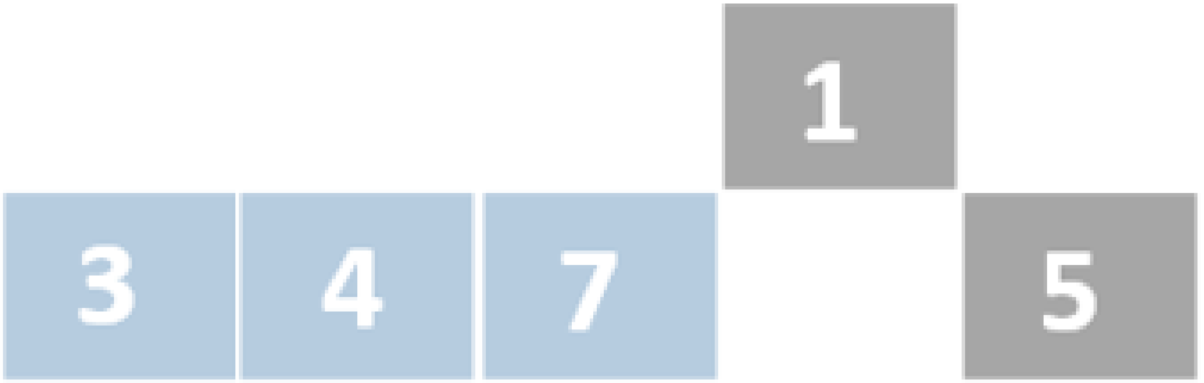


Insertion sort

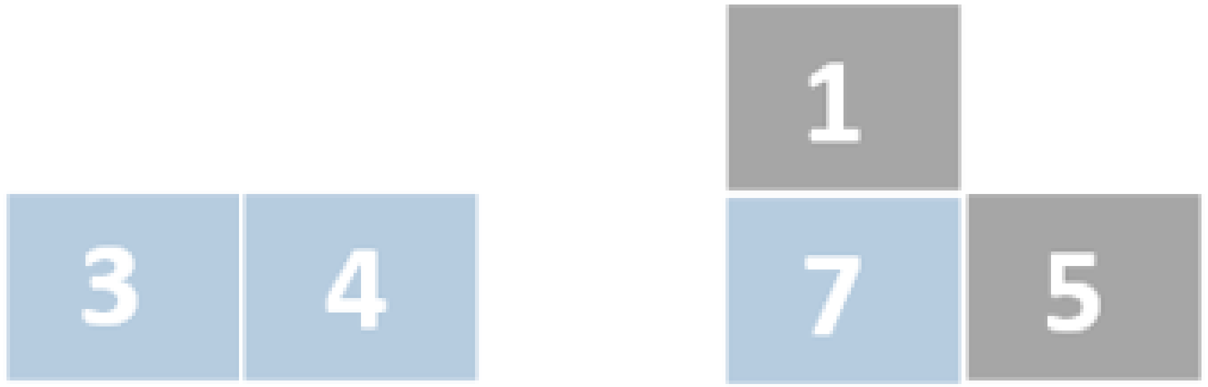


Insertion sort

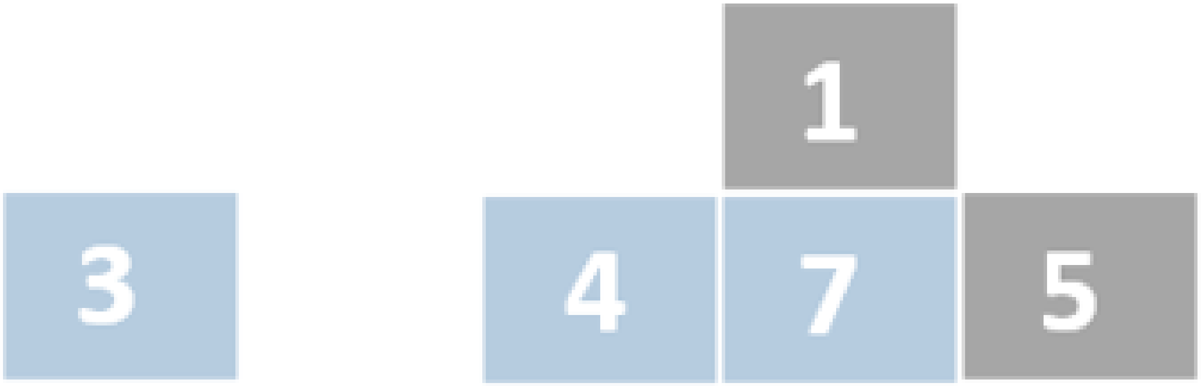
Insertion sort



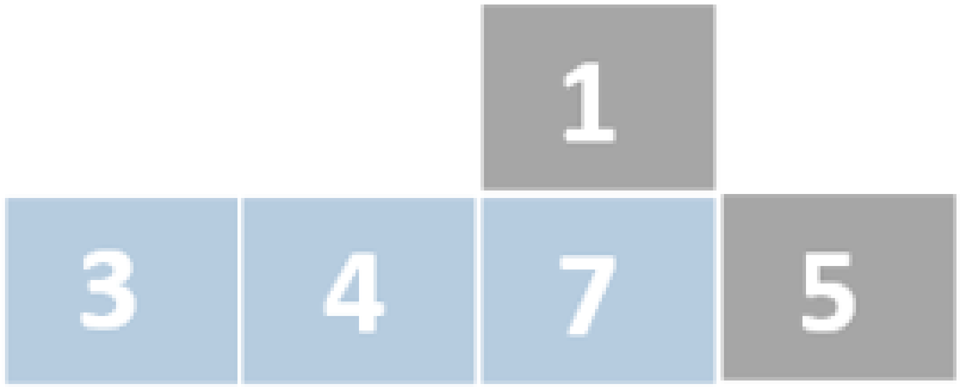
Insertion sort



Insertion sort



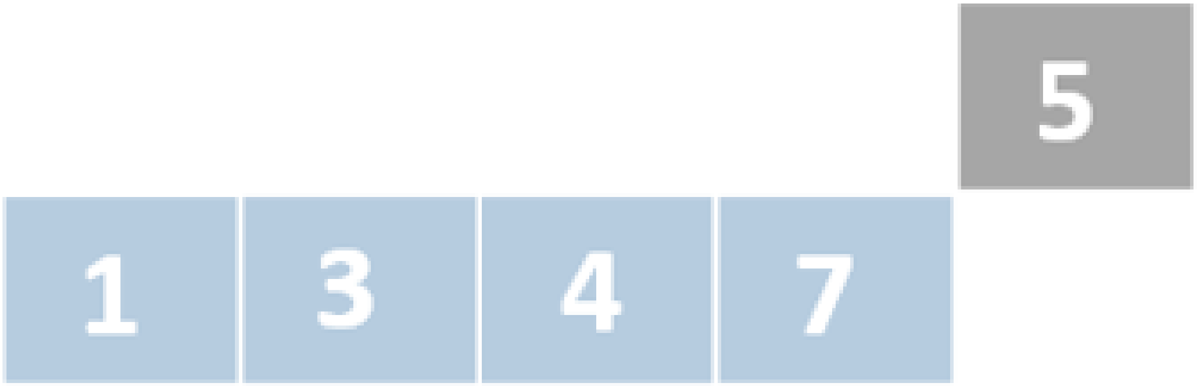
Insertion sort



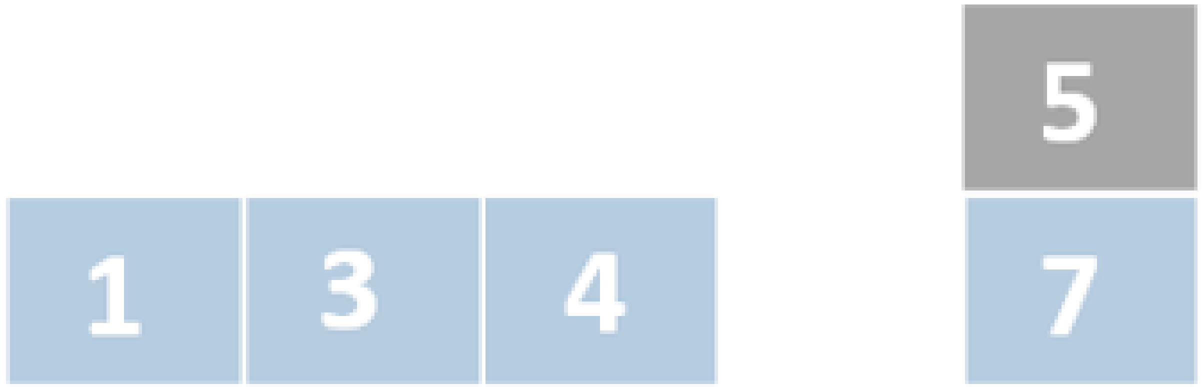
Insertion sort

Insertion sort

Insertion sort



Insertion sort



Insertion sort

Insertion sort - implementation

```
def insertion_sort(my_list):
  for i in range(1, len(my_list)):
      number_to_order = my_list[i]
      j = i - 1
      while j >= 0 and number_to_order < my_list[j]:
          my_list[j + 1] = my_list[j]
          j -= 1
      my_list[j + 1] = number_to_order
  return my_list
```

Insertion sort - complexity

- Worst case: $O(n^2)$
- Average case: $\Theta(n^2)$
- Best case: $\Omega(n)$

Let's practice!

DATA STRUCTURES AND ALGORITHMS IN PYTHON

Merge sort

DATA STRUCTURES AND ALGORITHMS IN PYTHON

Miriam Antona
Software engineer

Merge sort

- Follows **divide and conquer**
  - **Divide**
    - divides the problem into smaller sub-problems
  - **Conquer**
    - sub-problems are solved recursively
  - **Combine**
    - solutions of sub-problems are combined to achieve the final solution

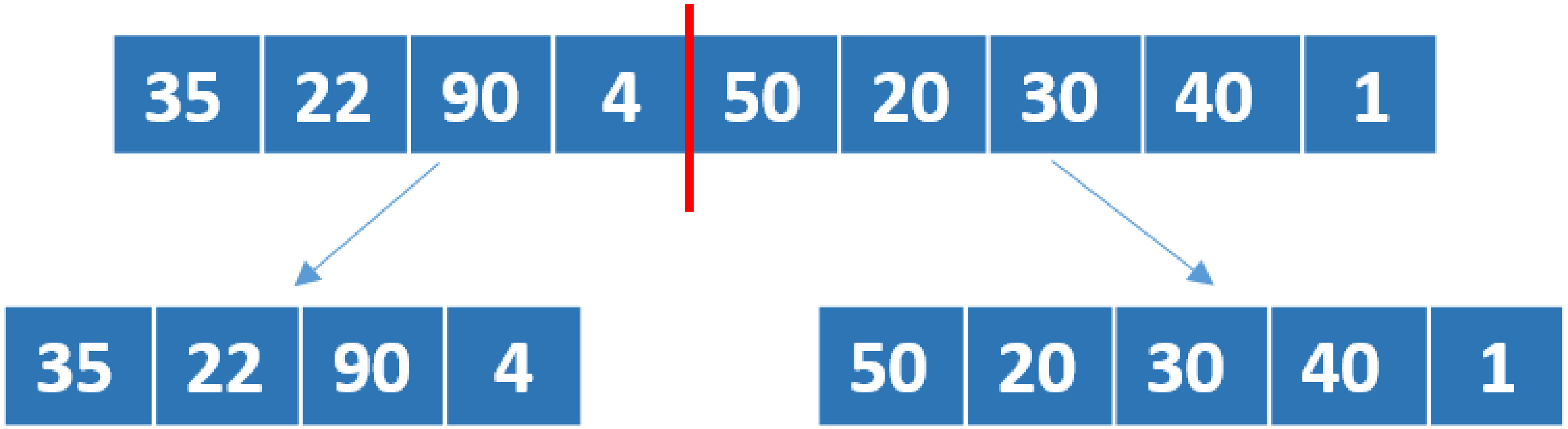
Merge sort - in action

35	22	90	4	50	20	30	40	1

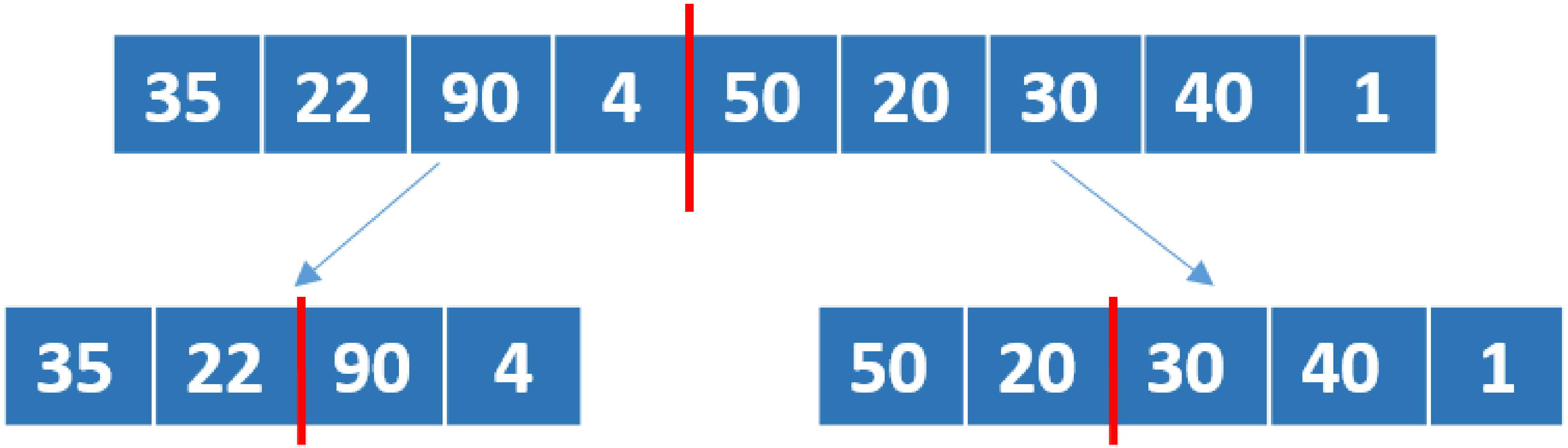
Merge sort - in action

35	22	90	4	50	20	30	40	1

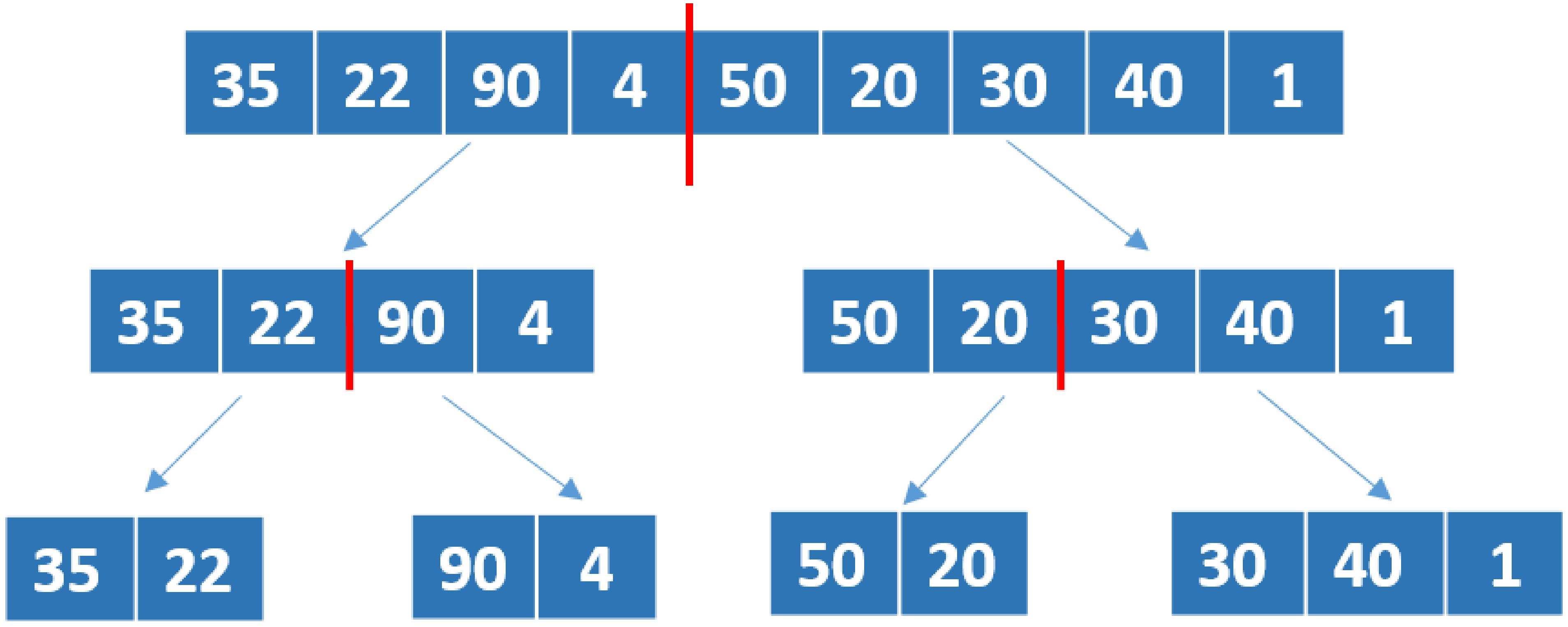
Merge sort - in action



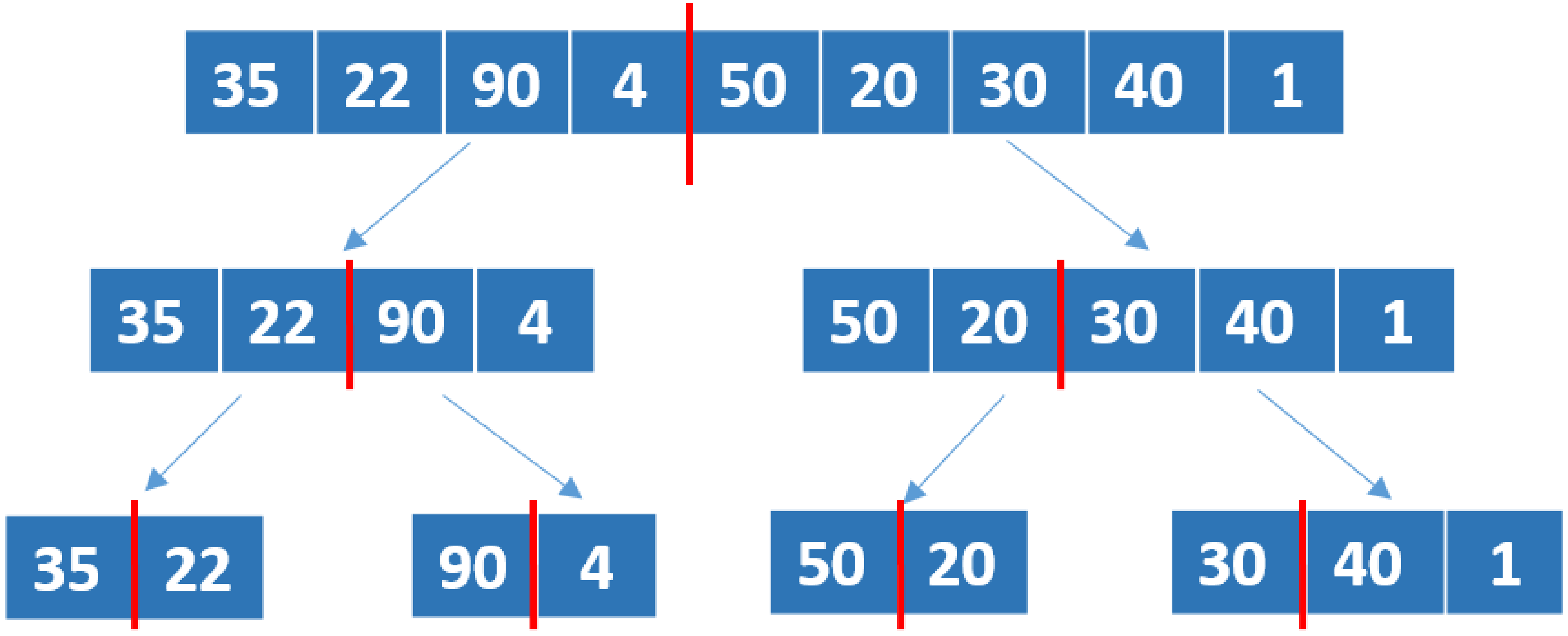
Merge sort - in action



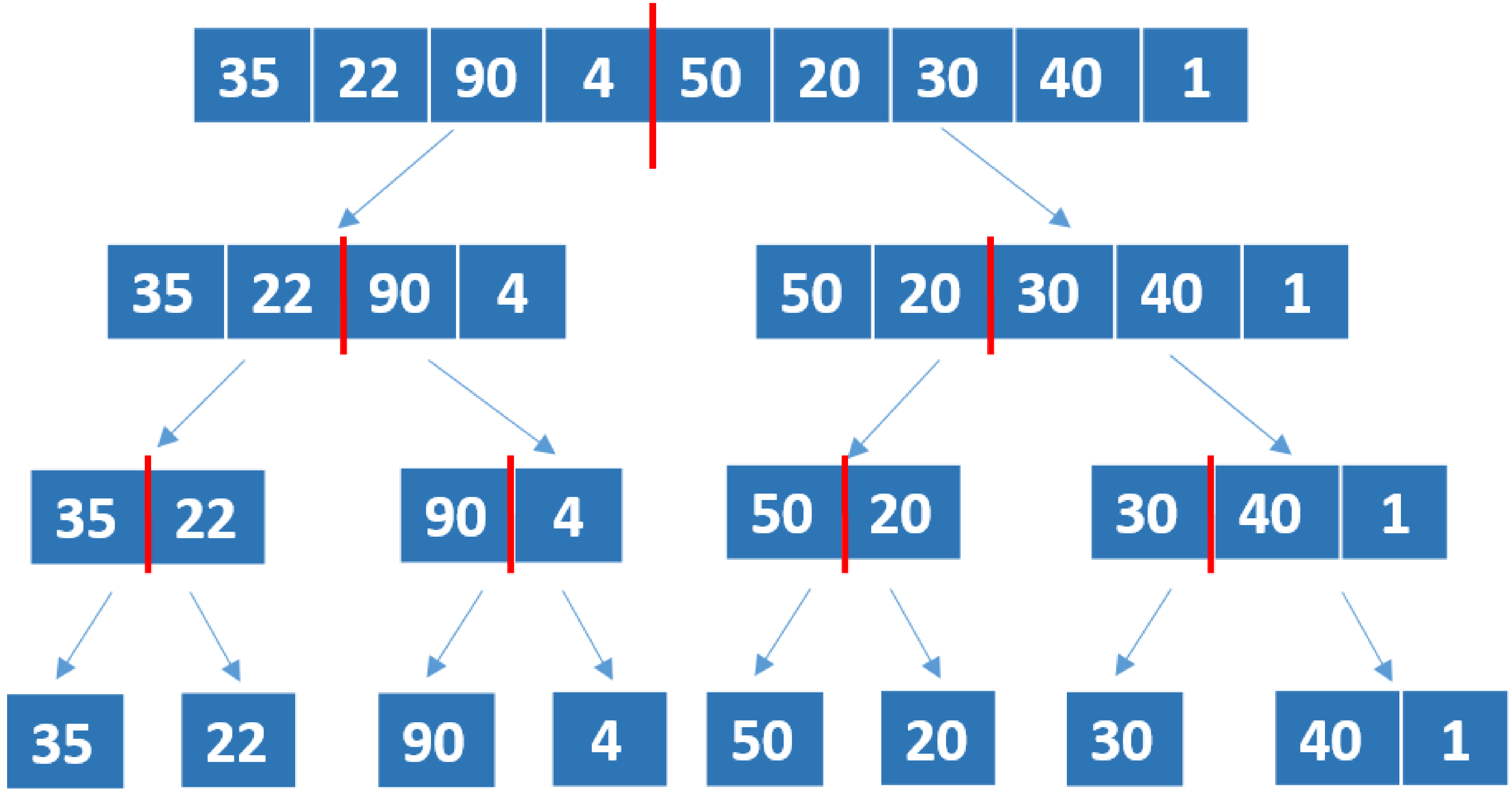
Merge sort - in action



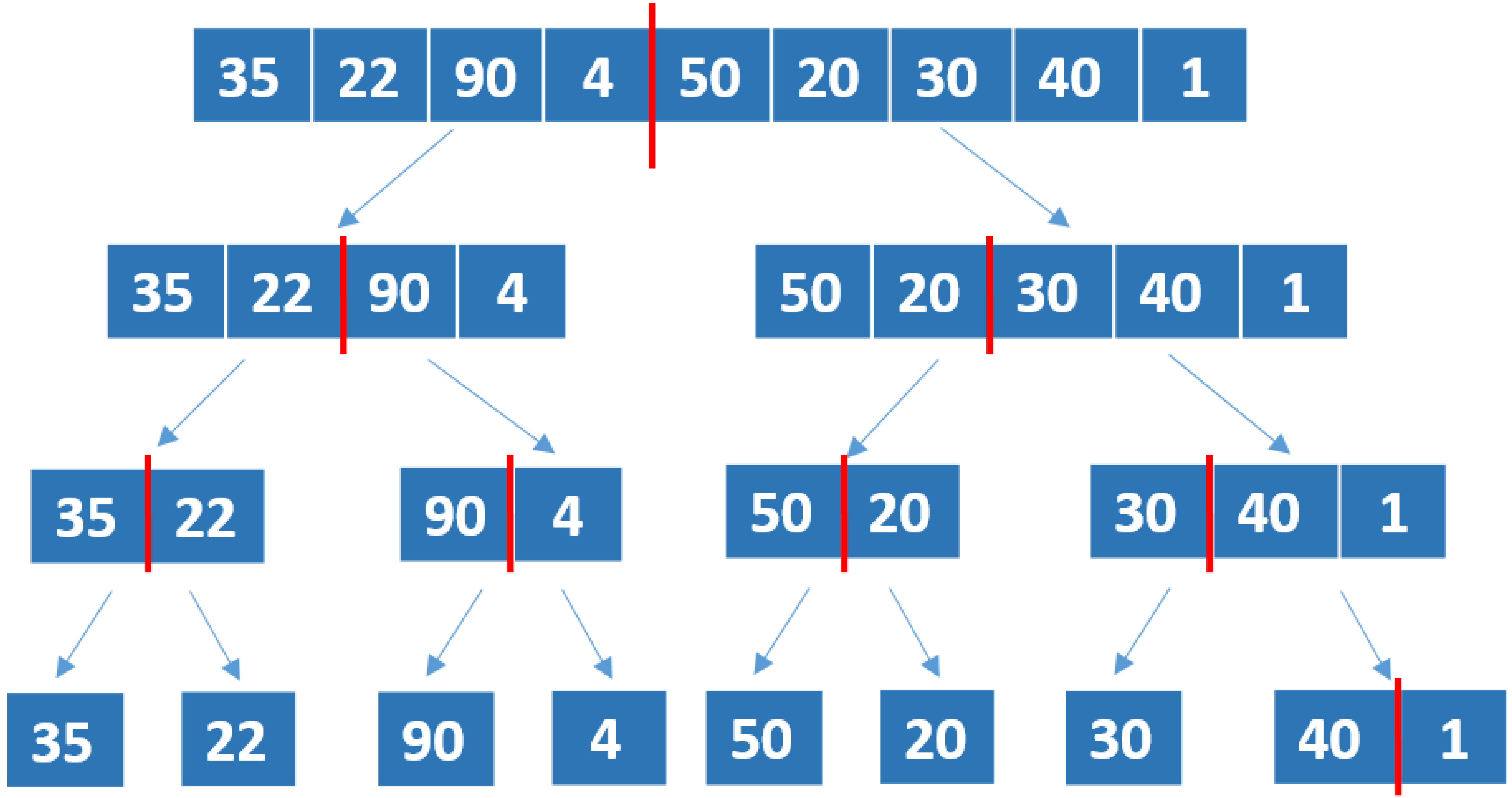
Merge sort - in action



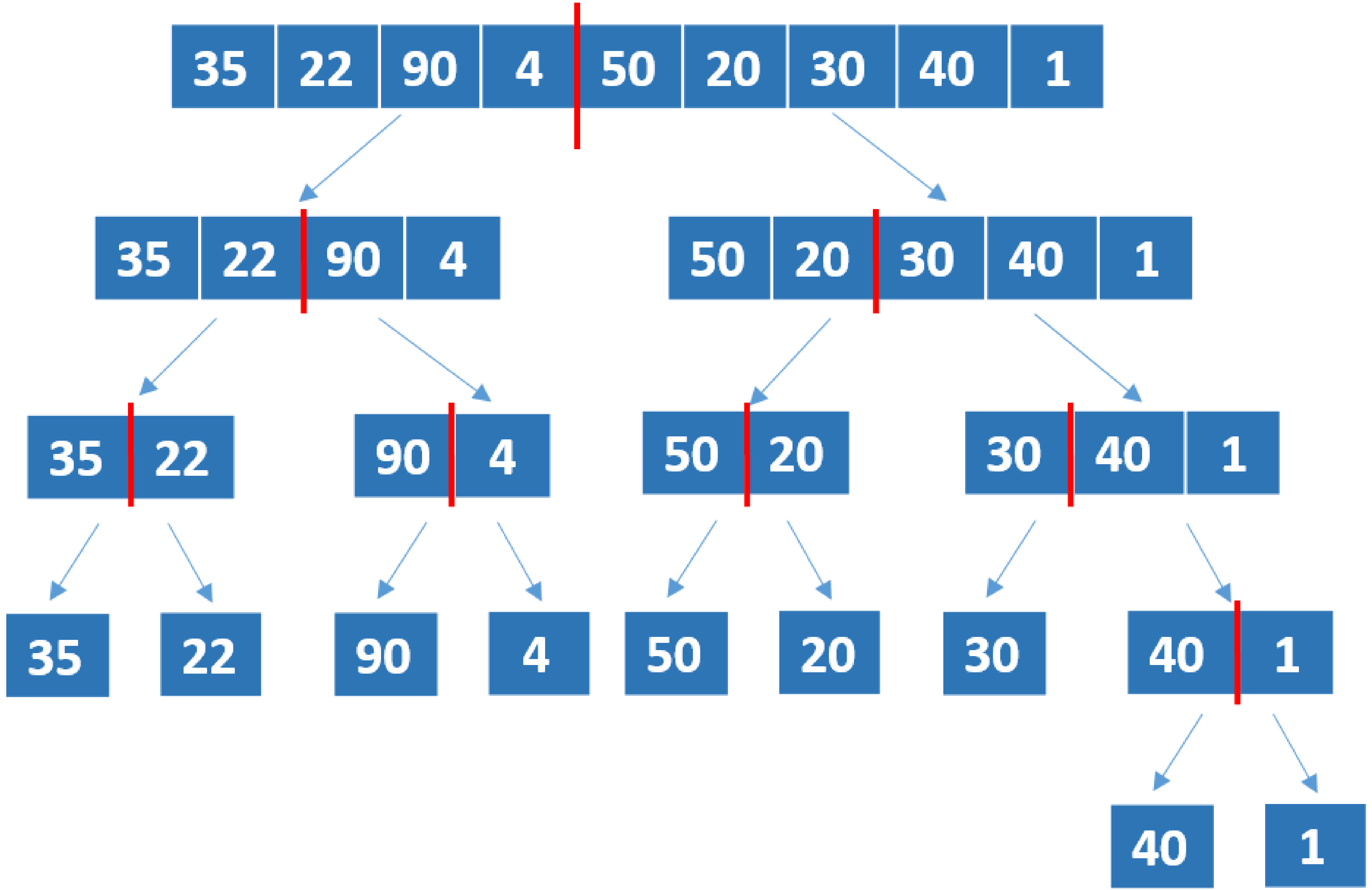
Merge sort - in action



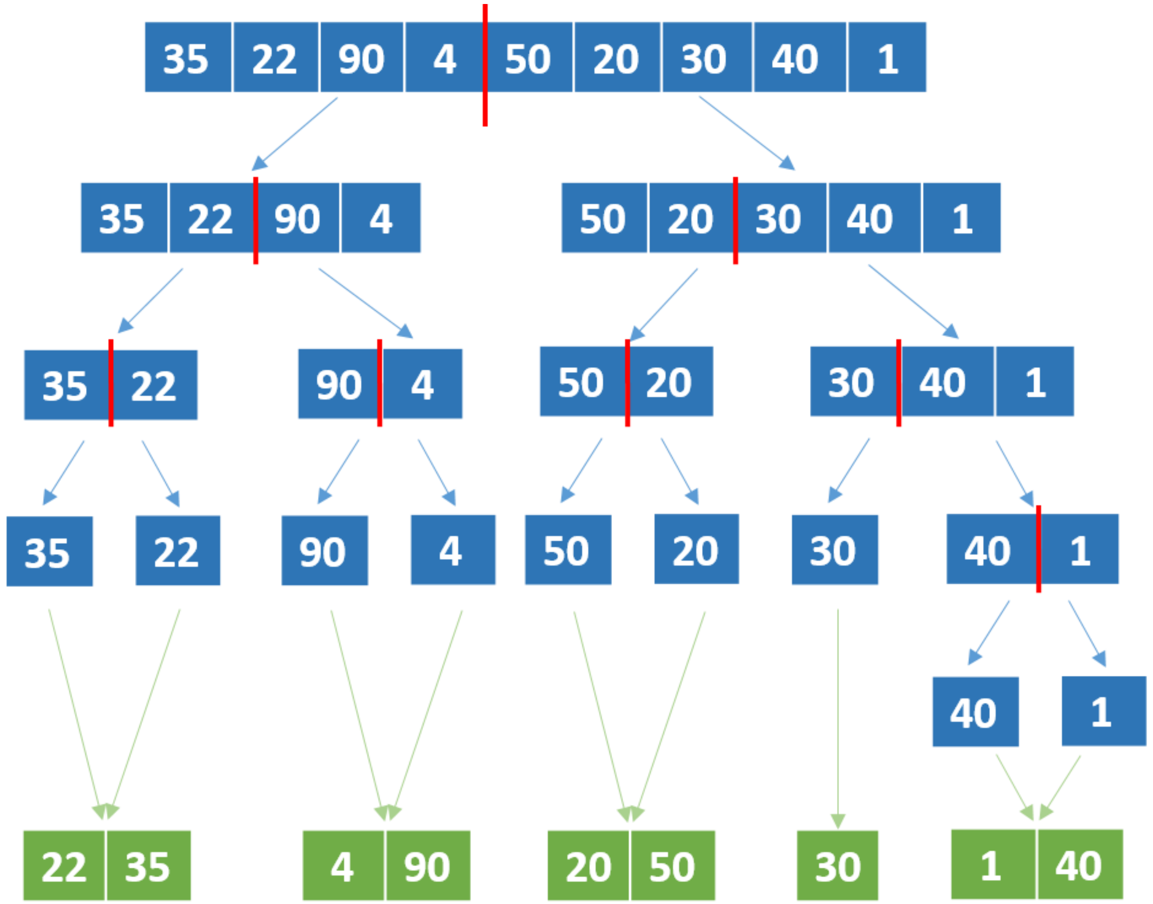
Merge sort - in action



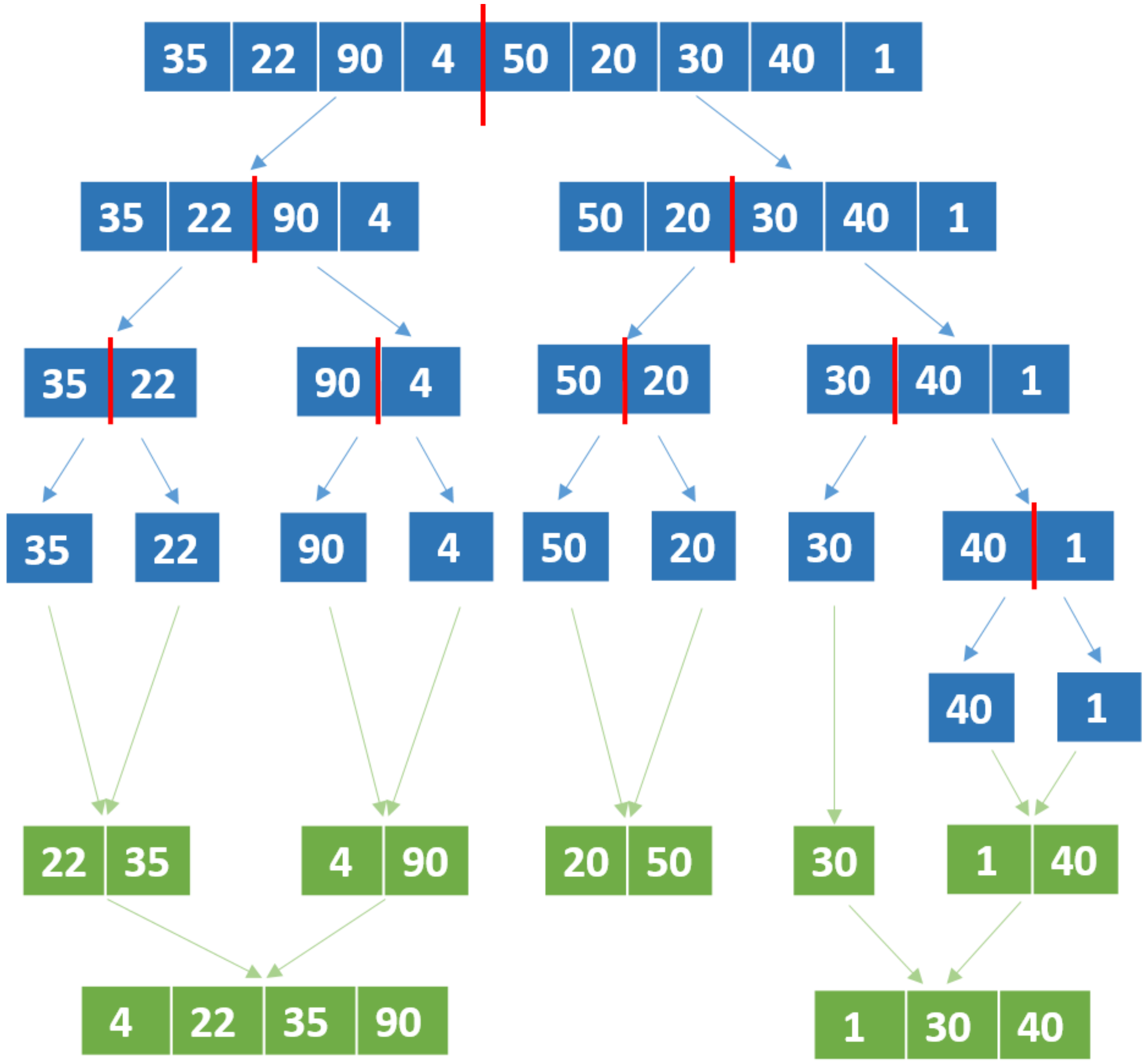
Merge sort - in action



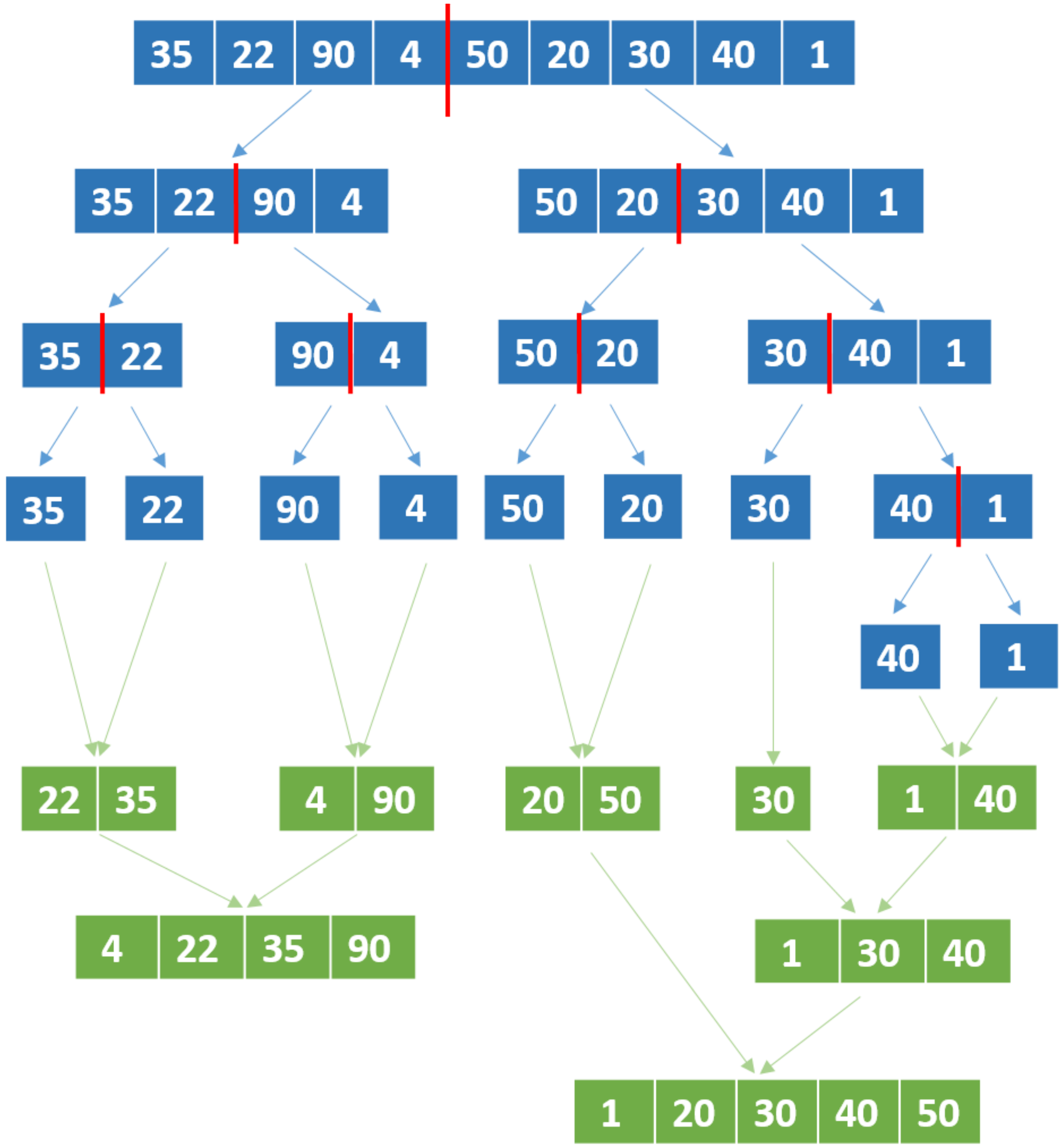
Merge sort - in action



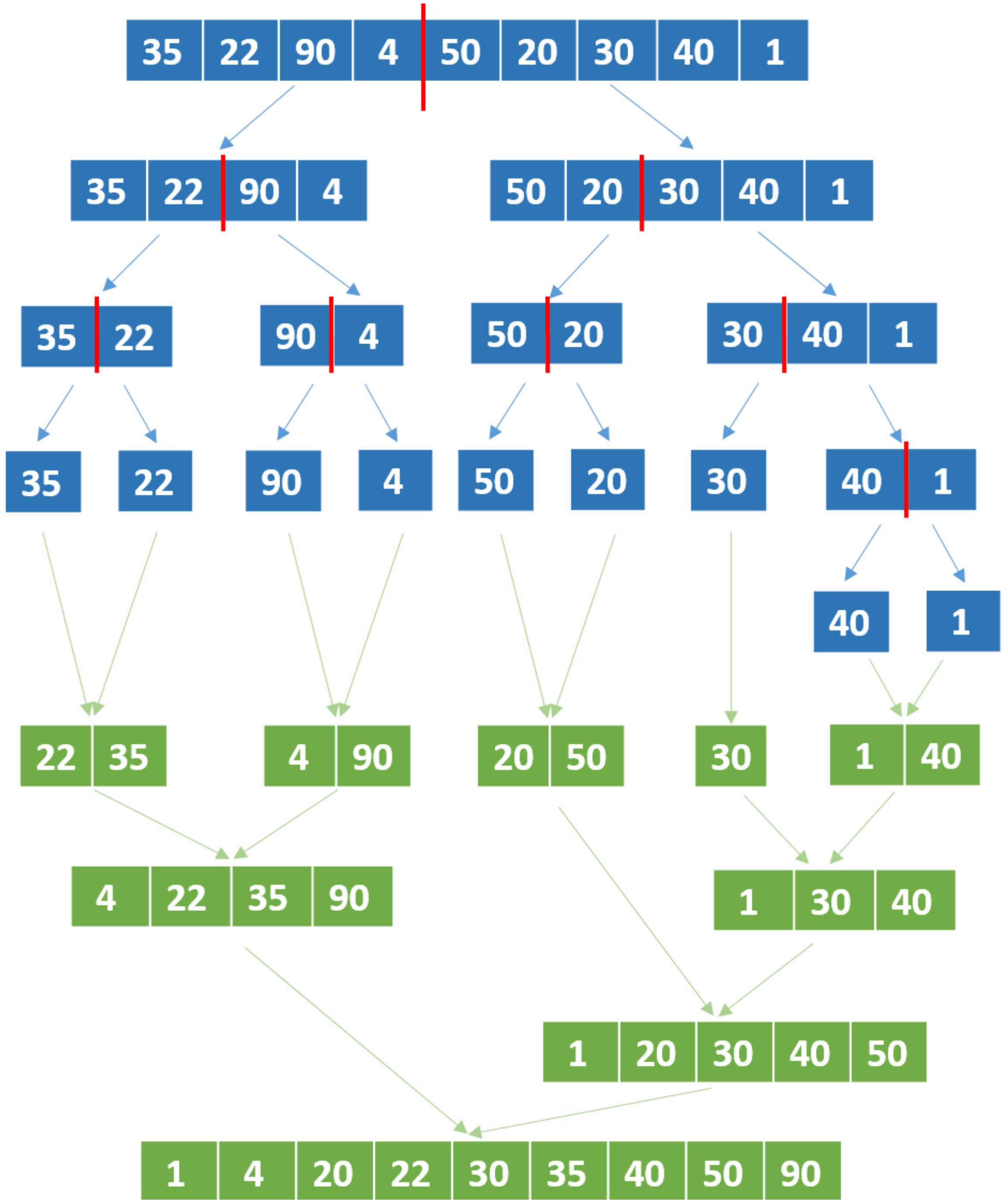
Merge sort - in action



Merge sort - in action



Merge sort - in action



Merge sort - implementation

```python
def merge_sort(my_list):
  if len(my_list) > 1:
    mid = len(my_list)//2
    left_half = my_list[:mid]
    right_half = my_list[mid:]
    merge_sort(left_half)
    merge_sort(right_half)

    i = j = k = 0
    while i < len(left_half) and j < len(right_half):
      if left_half[i] < right_half[j]:
        my_list[k] = left_half[i]
        i += 1
      else:
        my_list[k] = right_half[j]
        j += 1
      k += 1
```

```python
    while i < len(left_half):
      my_list[k] = left_half[i]
      i += 1
      k += 1

    while j < len(right_half):
      my_list[k] = right_half[j]
      j += 1
      k += 1
```

```python
my_list = [35,22,90,4,50,20,30,40,1]
merge_sort(my_list)
print(my_list)
```

```
[1, 4, 20, 22, 30, 35, 40, 50, 90]
```

Merge sort - complexity

- Worst case: $O(n \log n)$
  - significant improvement over bubble sort, selection sort, and insertion sort
  - suitable for sorting large lists
- Average case: $\Theta(n \log n)$
- Best case: $\Omega(n \log n)$
  - other algorithms (e.g. bubble sort, insertion sort) have better best case complexity
- Space complexity: $O(n)$
  - worst space complexity than other algorithms with $O(1)$
- Other variants reduce this space complexity

Let's practice!

DATA STRUCTURES AND ALGORITHMS IN PYTHON

Quicksort

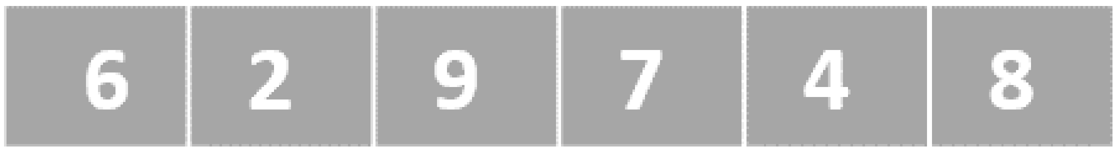
DATA STRUCTURES AND ALGORITHMS IN PYTHON

Miriam Antona
Software engineer

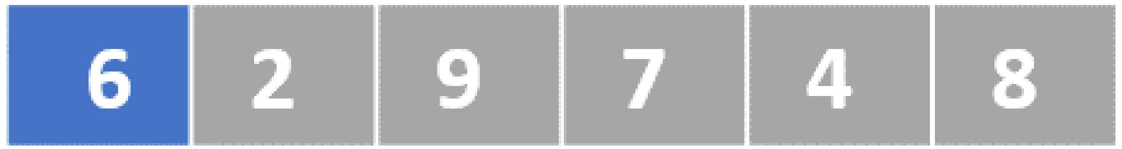
Quicksort

- Follows **divide and conquer** principle
- Implemented by many **programming languages**
- **Partition** technique
  - **Pivot**
  - items **smaller** than the pivot -> **left**
  - items **greater** than the pivot -> **right**
- Elements to the **left** will be sorted **recursively**
- Elements to the **right** will be sorted **recursively**

Quicksort - in action

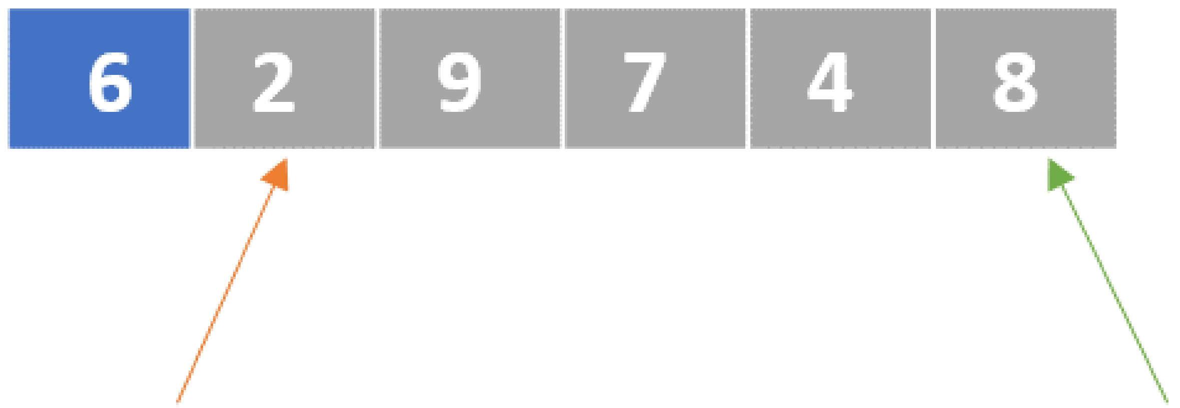


Quicksort - in action



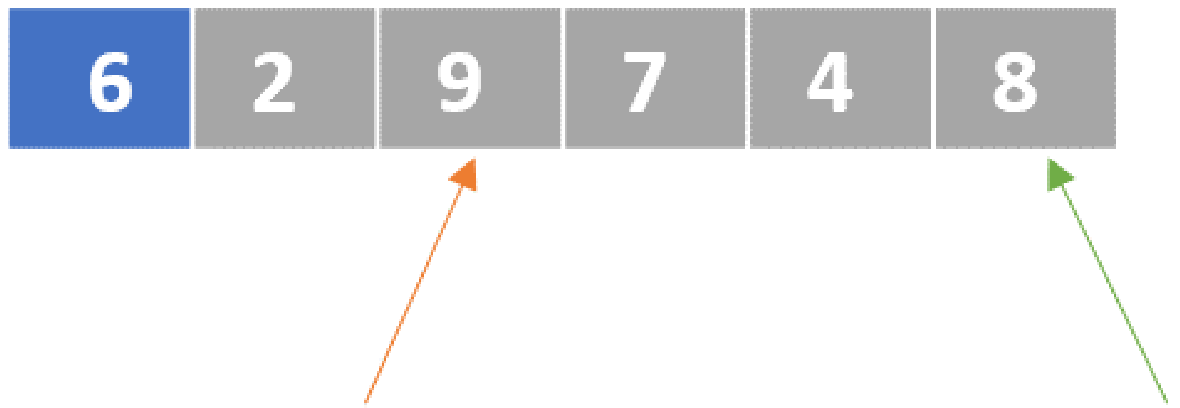
- Hoare's partition

Quicksort - in action



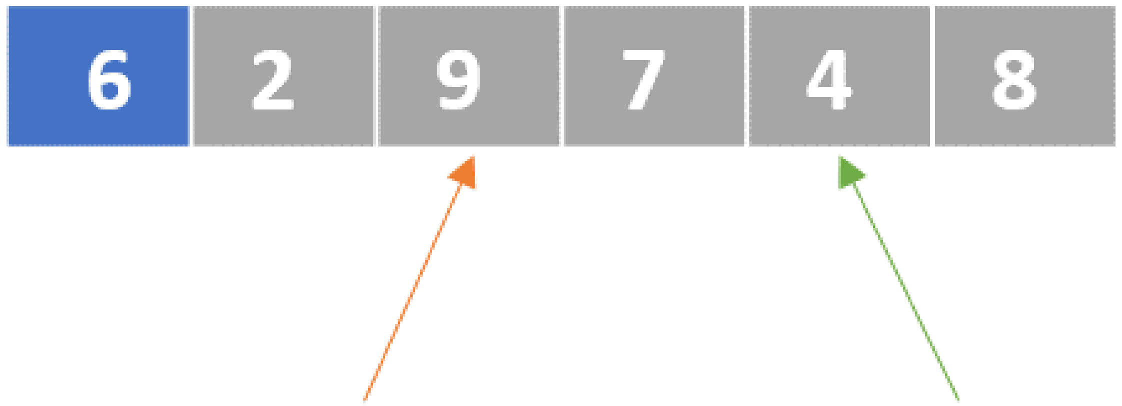
- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found

Quicksort - in action



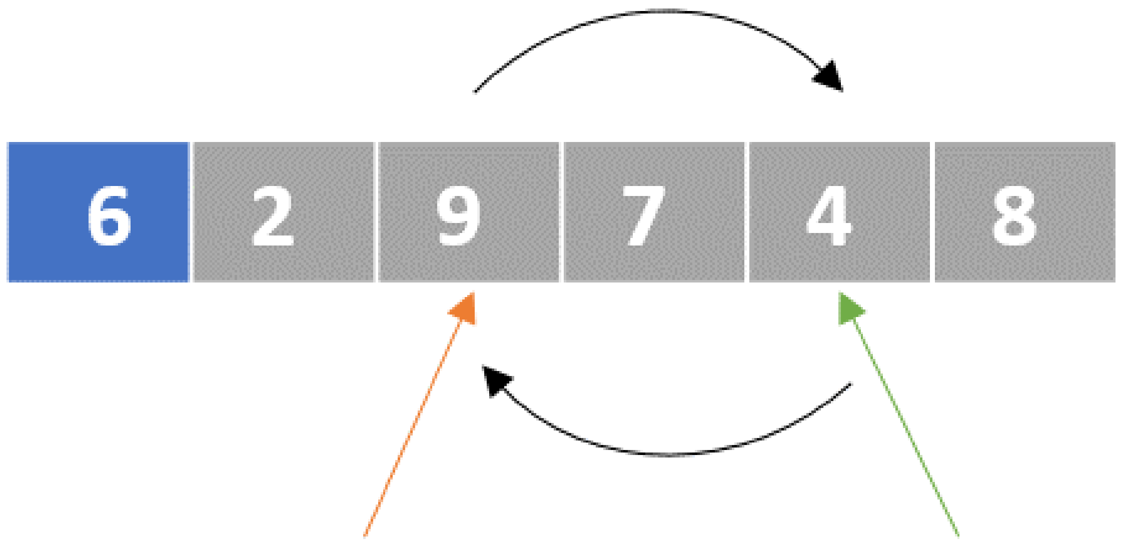
- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found

Quicksort - in action



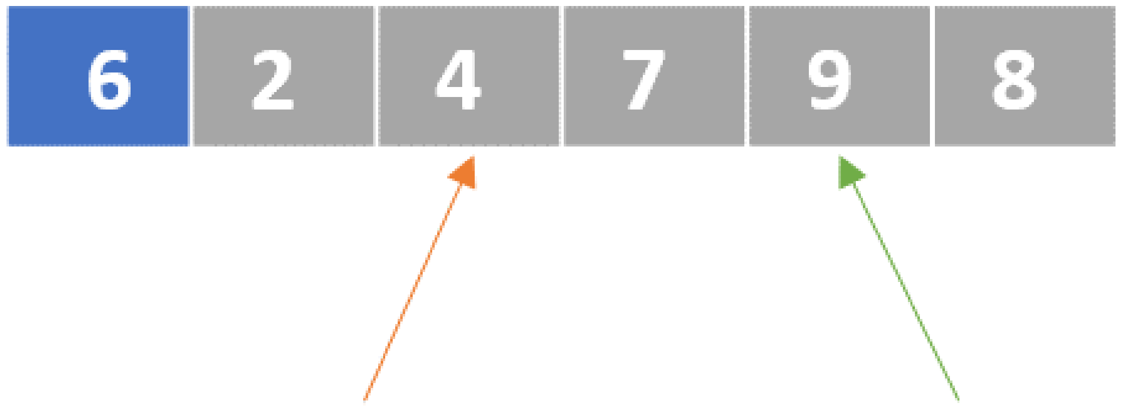
- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action



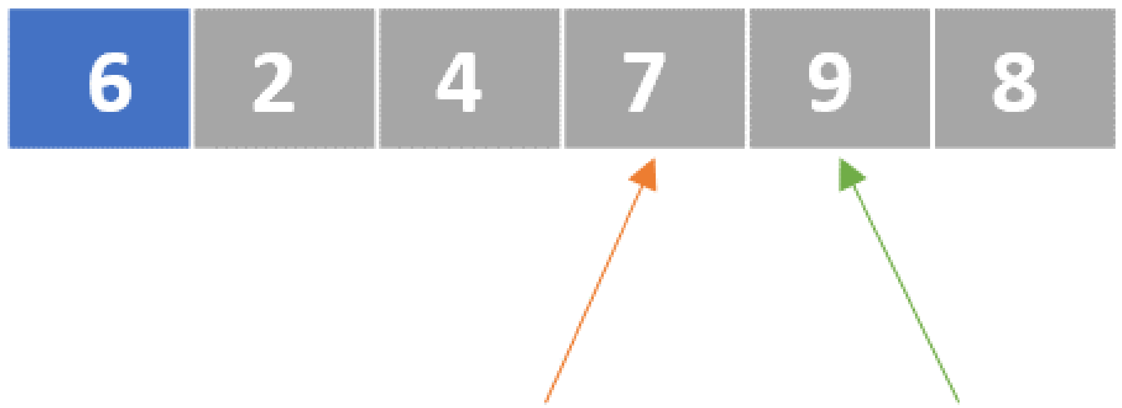
- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action



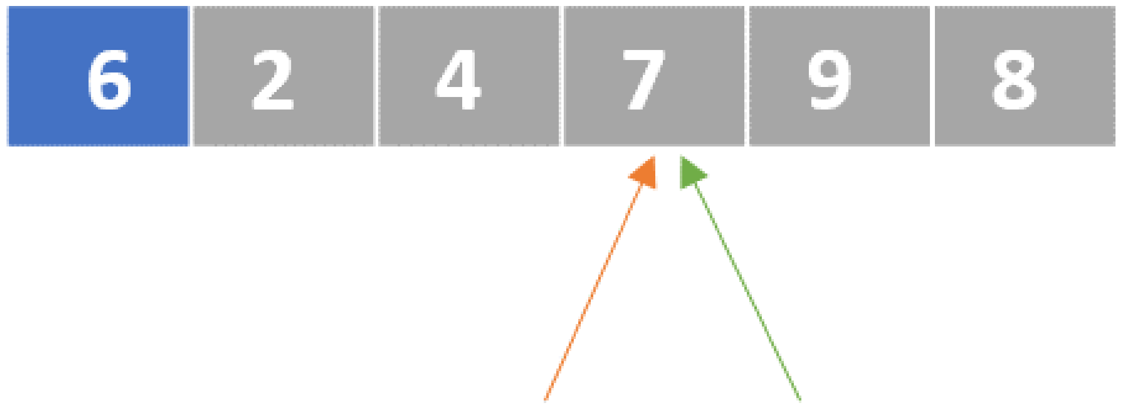
- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action



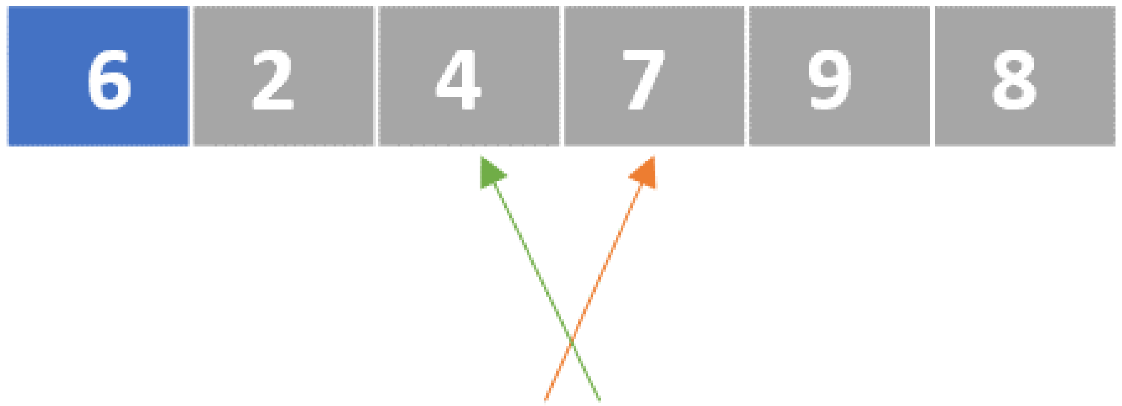
- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action



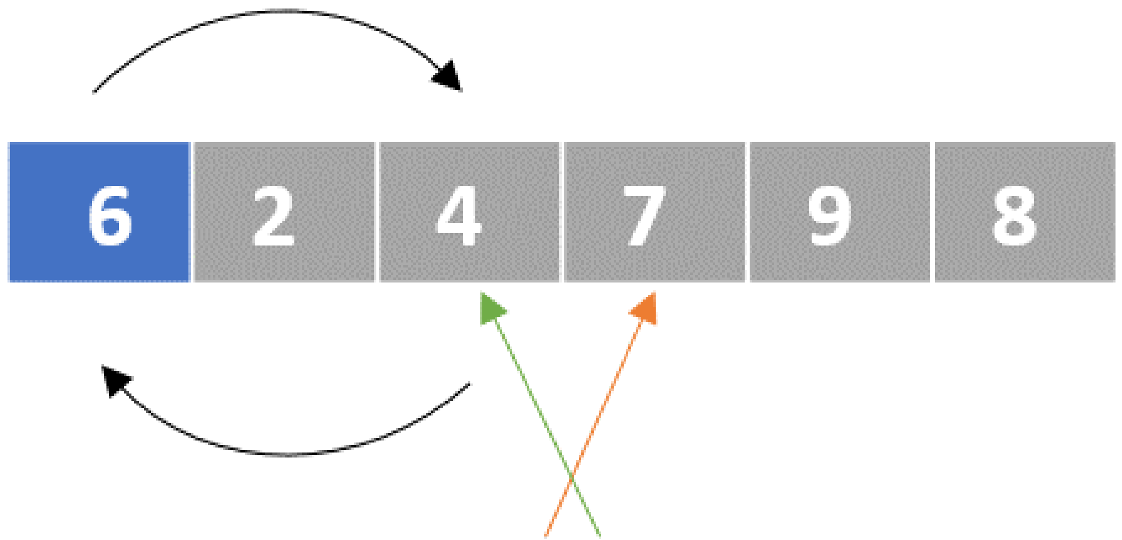
- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action



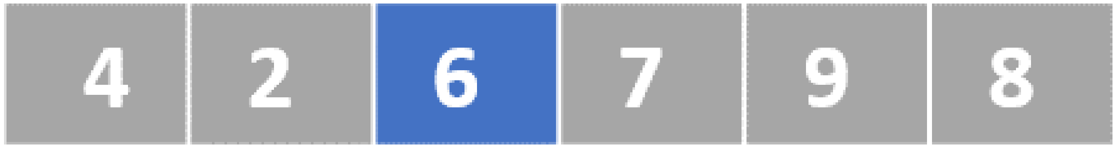
- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action



- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action



- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

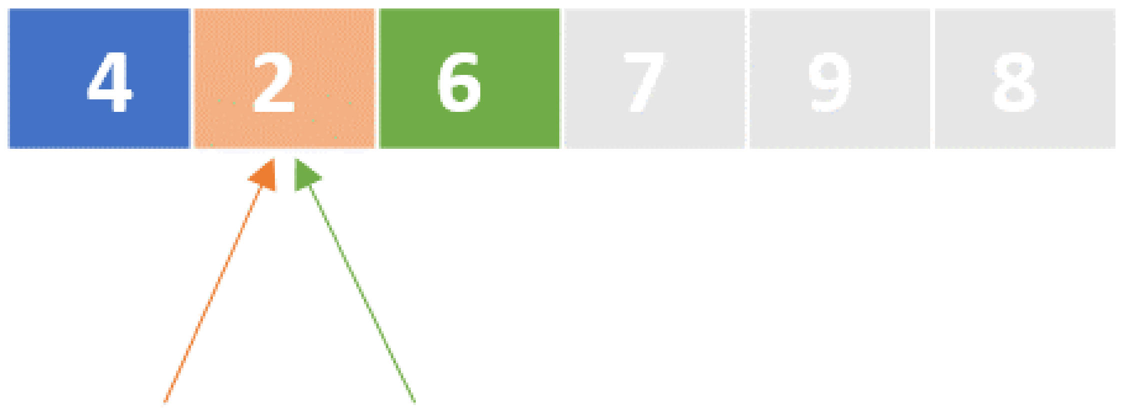
Quicksort - in action

- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action

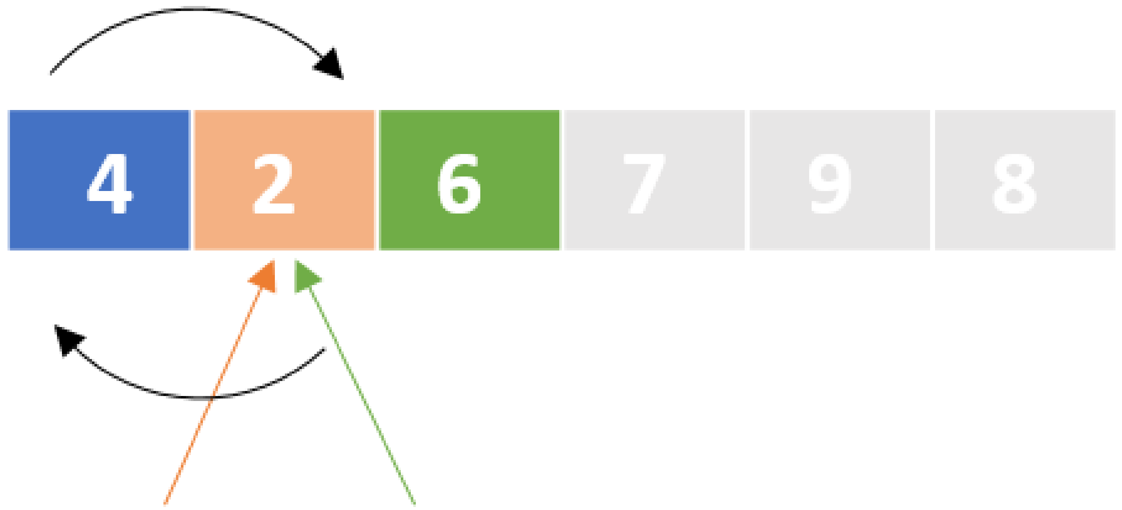
- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action



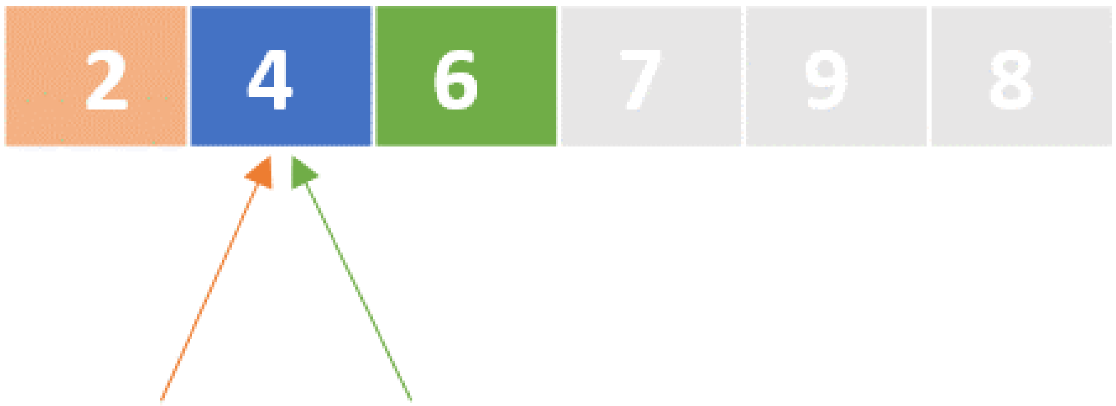
- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action



- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action



- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action

- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action

- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action

- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

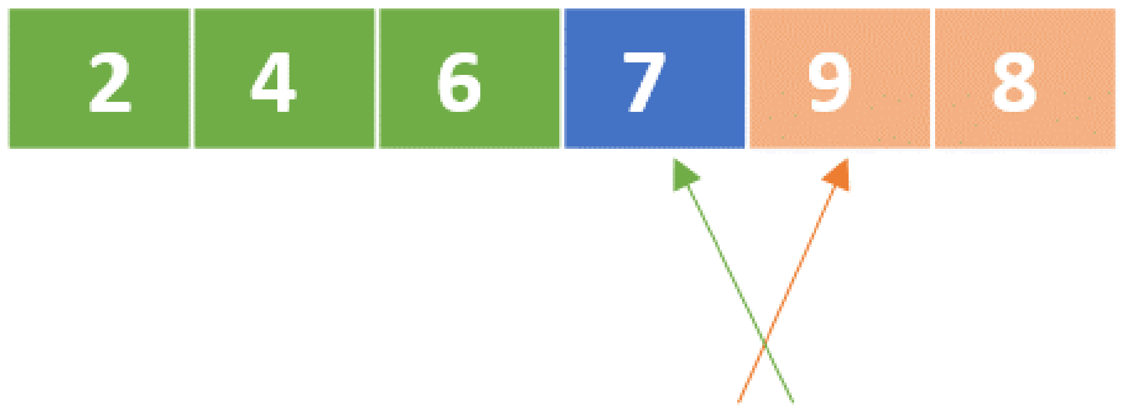
Quicksort - in action

- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action

- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action

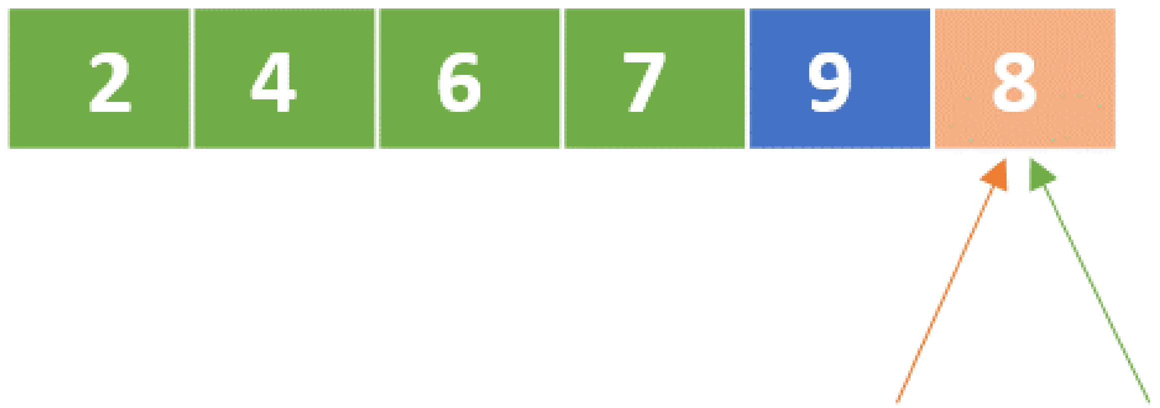


- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action

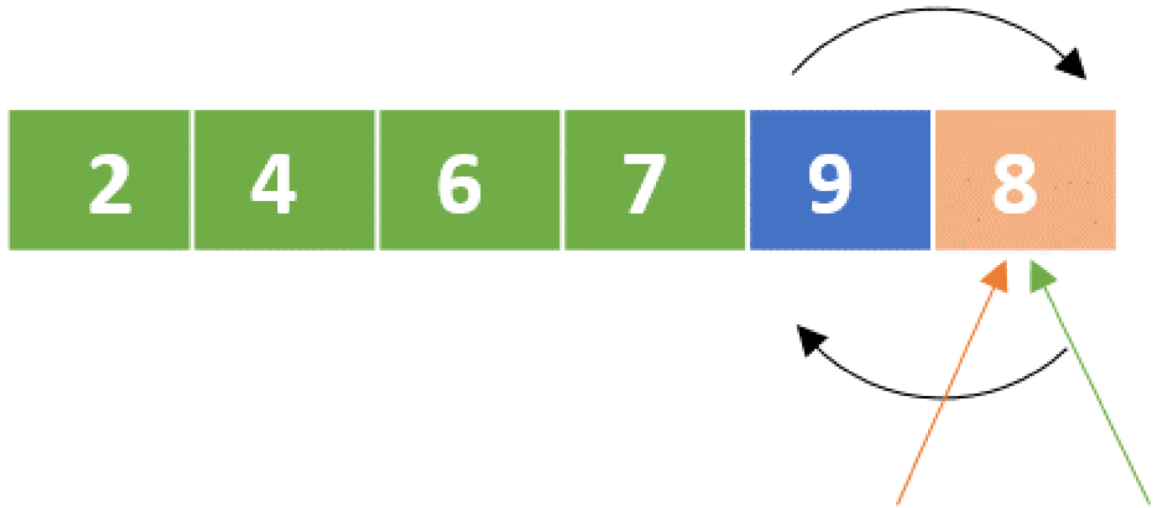
- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action



- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action



- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action

- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - in action

- Hoare's partition
  - Move **left** pointer until a value **greater** than pivot is found
  - Move **right** pointer until a value **lower** than pivot is found

Quicksort - implementation

```
def quicksort(my_list, first_index, last_index):
    if first_index < last_index:
        partition_index = partition(my_list, first_index, last_index)
        quicksort(my_list, first_index, partition_index)
        quicksort(my_list, partition_index + 1, last_index)
```

Quicksort - implementation

```
def partition(my_list, first_index, last_index):
    pivot = my_list[first_index]
    left_pointer = first_index + 1
    right_pointer = last_index
    while True:
        while my_list[left_pointer] < pivot and left_pointer < last_index:
            left_pointer += 1
        while my_list[right_pointer] > pivot and right_pointer >= first_index:
            right_pointer -= 1
        if left_pointer >= right_pointer:
            break
        my_list[left_pointer], my_list[right_pointer] = my_list[right_pointer], my_list[left_pointer]
    my_list[first_index], my_list[right_pointer] = my_list[right_pointer], my_list[first_index]
    return right_pointer
```

Quicksort - implementation

```
my_list = [6, 2, 9, 7, 4, 8]
quicksort(my_list, 0, len(my_list) - 1)
print(my_list)
```

```
[2, 4, 6, 7, 8, 9]
```

Quicksort - complexity

- Worst case: $O(n^2)$
- Very efficient!
  - Average case: $\Theta(n \log n)$
  - Best case: $\Omega(n \log n)$
- Space complexity: $O(n \log n)$

Let's practice!

DATA STRUCTURES AND ALGORITHMS IN PYTHON

Congratulations!

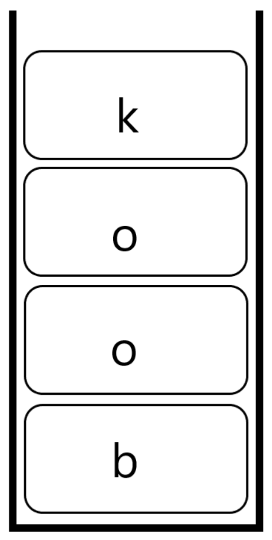
DATA STRUCTURES AND ALGORITHMS IN PYTHON

Miriam Antona
Software engineer

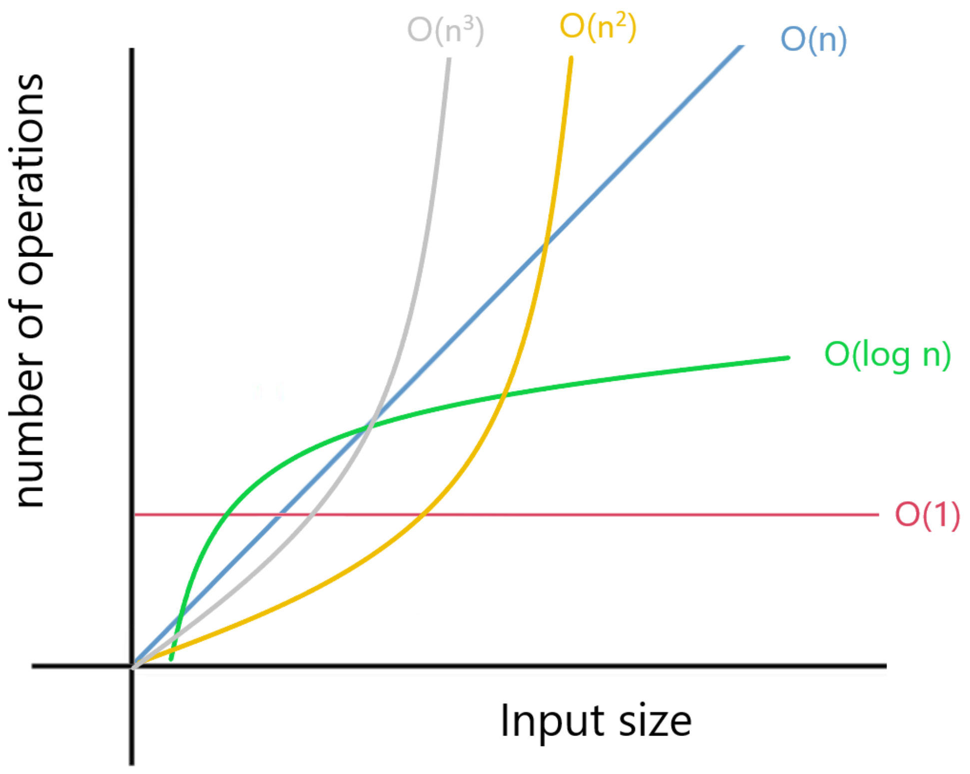
Chapter 1

- What algorithms and data structures are
- Linked lists

- Stacks

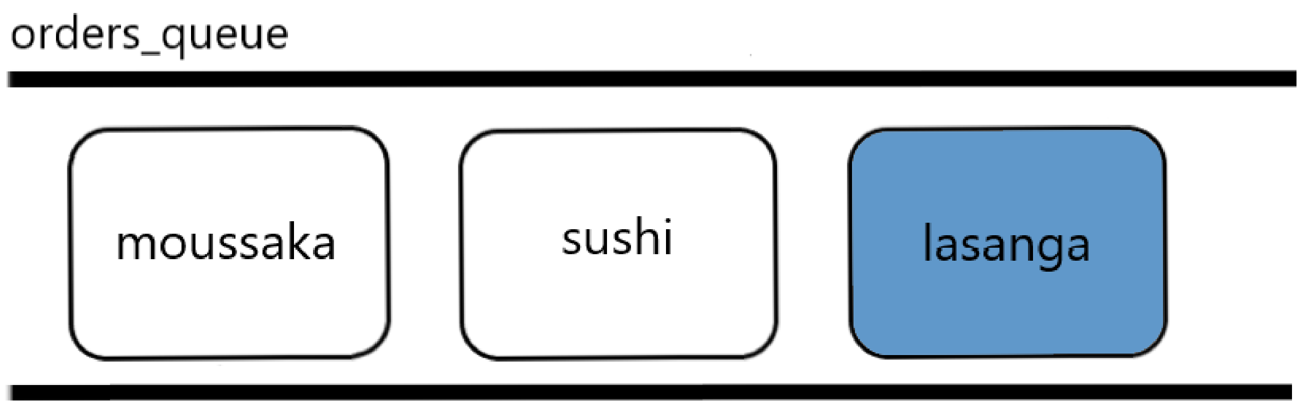


- Calculate time complexity using Big O Notation



Chapter 2

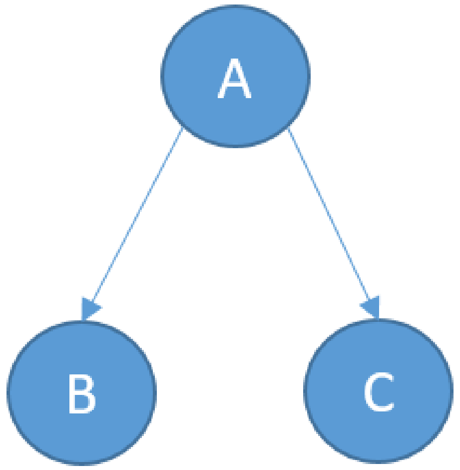
- Queues



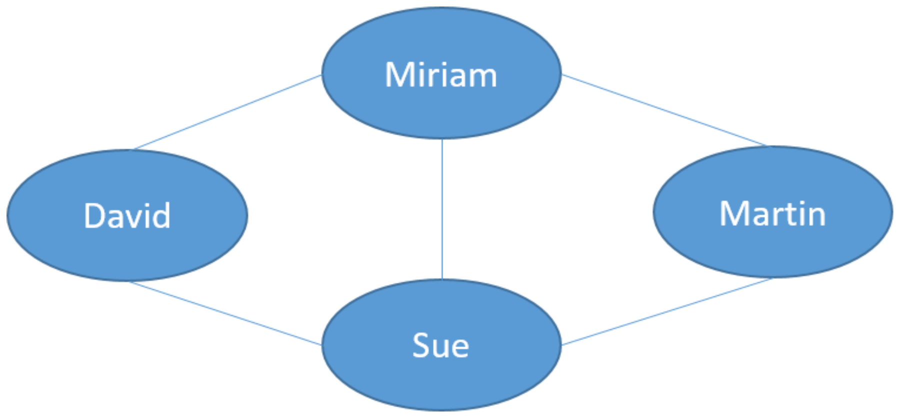
- Hash tables

```
my_menu = {
    'lasagna': 14.75,
    'moussaka': 21.15,
    'sushi': 16.05
}
```

- Trees



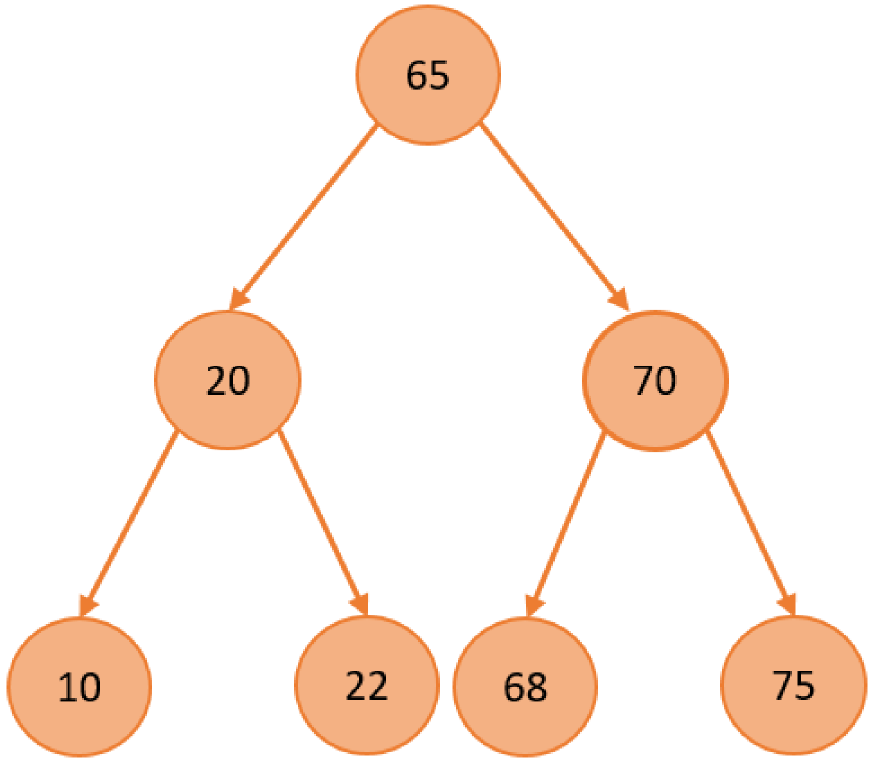
- Graphs



- Recursion

Chapter 3

- Searching algorithms:
  - Linear search
  - Binary search
  - Depth first search
  - Breadth first search
- Binary search trees



Chapter 4

- Sorting algorithms
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Merge sort
  - Quicksort

Thank you!

DATA STRUCTURES AND ALGORITHMS IN PYTHON