Data Structures and Algorithms

Lesson 11. Selection Algorithms

Alexandra Ciobotaru

5th of May 2022

Syllabus

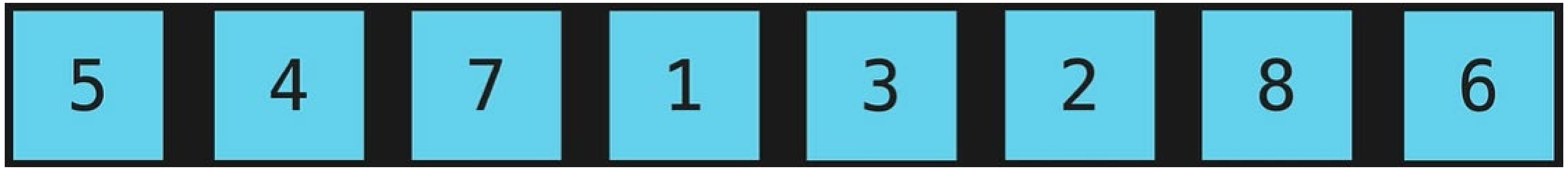
- **Lesson 1. Introduction: Algorithms, Data Structures and Cognitive Science.**
- **Lesson 2. Python Data Types and Structures.**
- **Lesson 3. Principles of Algorithm Design.**
- **Lesson 4. Lists and Pointer Structures.**
- **Lesson 5. Stacks and Queues.**
- **Lesson 6. Trees.**
- **Lesson 7. Hashing and Symbol Tables.**
- **Lesson 8. Graphs.**
- **Lesson 9. Searching.**
- **Lesson 10. Sorting.** Mid-term evaluation quiz (30%)
- **Lesson 11. Selection Algorithms.**
- Lesson 12. String Algorithms and Techniques.
- Lesson 13. Design Techniques and Strategies.
- Lesson 14. Implementations, Applications and Tools.
- Lab (20%) + Project (50%).

We’ll talk about

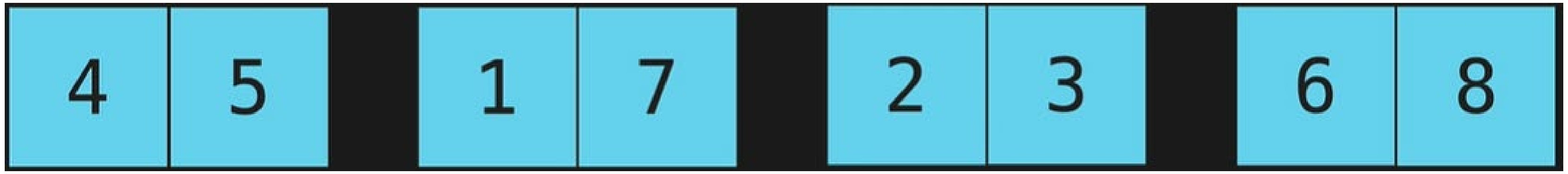
- 1. Merge Sort
- 2. Quick Sort
- 3. Selection Algorithms - Quick Select

1. Merge Sort

- The idea behind merge sort is:
  - If you have two sorted sub-lists, it is very easy to combine these into one sorted list
- So if I have a list of elements I break it down in two, break each half in two, and so on, until the only thing we have left are lists that only have one item in them



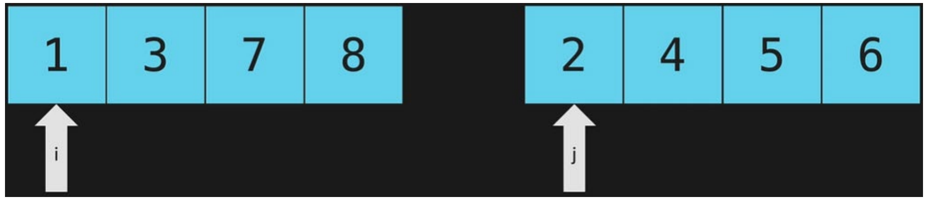
- Further, we take elements two by two from these lists, and create new lists that are sorted



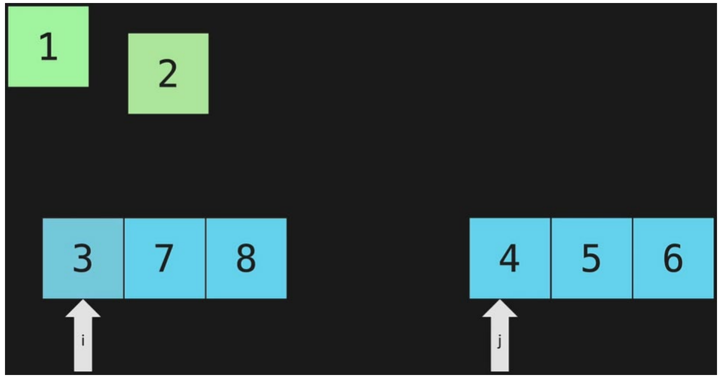
- Next we take lists two by two and combine them, until we have the whole list sorted

Merging

- Merge is what is called a helper function.
- The portion that merge does is to take two sorted lists - it takes the values from these and puts them into a new, combined, list.



- How it is done: we compare i and j, and whichever is lowest is added to the a, combined, list. We do this recursively until one of the two lists becomes empty.



Further, we'll have another loop that goes through the elements left in the list and puts them in the combined, ordered, list.

Merging - Code

```
def merge(list1, list2):
    combined = []
    i = 0 # we initialize i and j
    j = 0
    while i < len(list1) and j < len(list2): # we don’t use for loops, because we don’t know how 
many elements to count. So while both lists still have elements in them:
        if list1[i] < list2[j]: # if the first element from first list is less than first element 
from second list, we put it in combined and increment i or j
            combined.append(list1[i])
            i += 1
        else:
            combined.append(list2[j])
            j += 1
    # the above while loop ends when one of the sublists becomes empty, and then:
    while i < len(list1):
        combined.append(list1[i])
        i += 1

    while j < len(list2):
        combined.append(list2[j])
        j += 1

    return combined

print(merge([1,2,7,8], [3,4,5,6]))
```

Merge Sort

- It takes a list and breaks it in two halves, then breaks that list in halves again, and then again and again…
- We then use merge function to compare the sub-lists.
- We need to write the code that:
  - 1. breaks the list in half over and over – we will use *recursion* for this
  - 2. Stops at the *base case*: when len(the_list) = 1 (here we stop breaking lists in half)
  - 3. Next, we use *merge()* to put lists together

Merge Sort - code

```
def merge_sort(my_list):
    # the base case
    if len(my_list) == 1:
        return my_list
    # first we need to find the mid point
    mid = int(len(my_list)/2)

    # left sublist
    left = my_list[:mid]
    # right sublist
    right = my_list[mid:]

    # recursion here
    return merge(merge_sort(left), merge_sort(right))

print(merge_sort([3,1,4,2]))
```

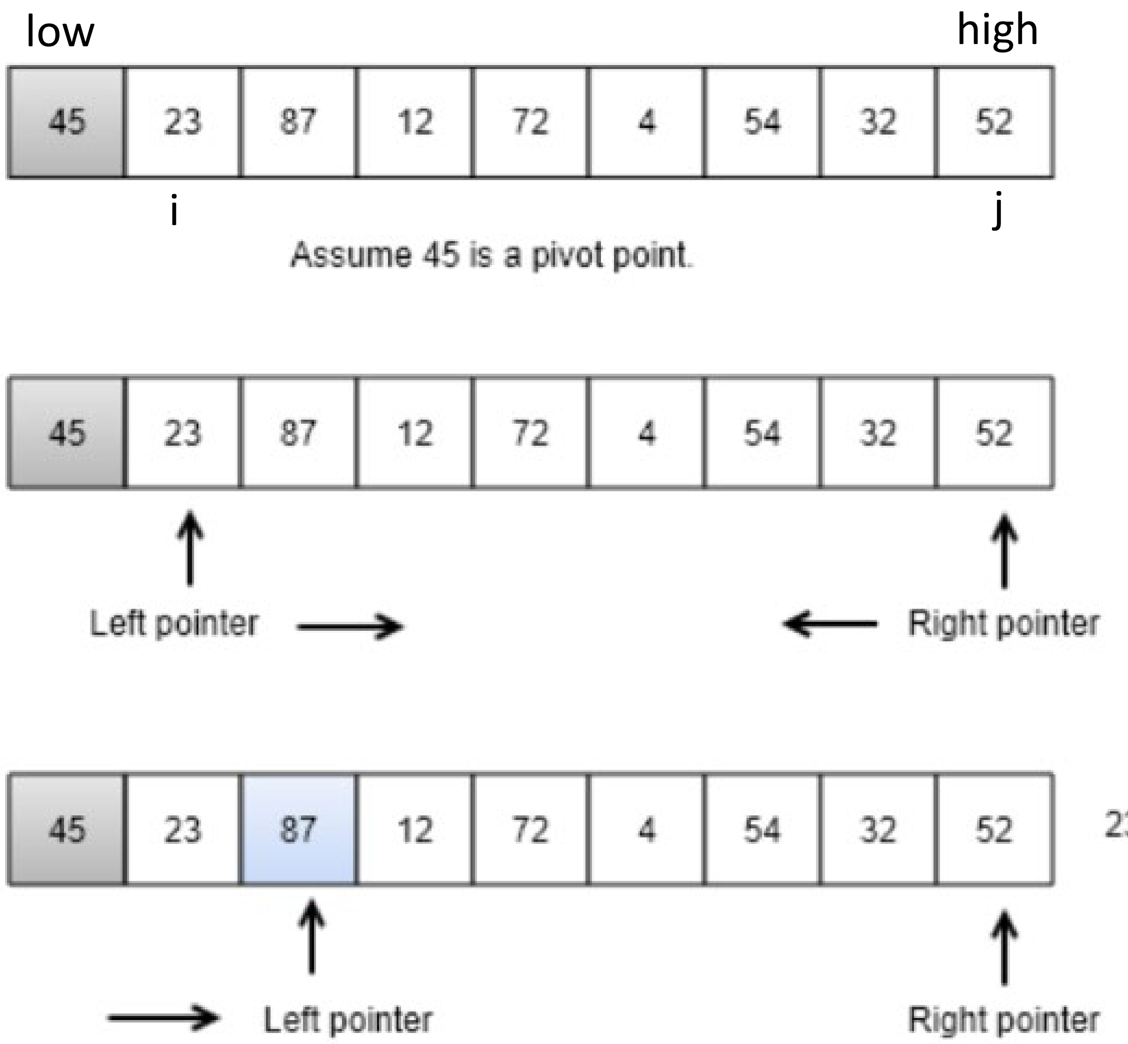
2. Quick Sort

- We start with a pivot point, that’s going to be the first element in the list.
- All the elements in the list will be compared with this pivot.
- <u>All elements that are less than the pivot will be put to the left</u> of the pivot, while <u>all elements greater than the pivot will be put to the right</u> of the pivot.
- This is a *divide and conquer algorithm.* We partition an unsorted list into two sub-lists, in such a way that all the elements on the left side of that partition point (also called a pivot) should be smaller than the pivot, and all the elements on the right side of the pivot should be greater than the pivot.

Quick Sort

- After the first iteration of the quick sort algorithm, the chosen pivot point is placed in the list at its correct position. After the first iteration, we obtain two unordered sub-lists, and follow the same process again on these two sub-lists.
- Thus, the quick sort algorithm partitions the list into two parts and recursively applies the quick sort algorithm on these two sub-lists, in order to sort the whole list.
- **The main idea is that an element is in its sorted position if all elements before it are smaller, and all the elements after it are greater than that element.**
- Check: https://www.youtube.com/watch?v=7h1s2SojIRw

Example

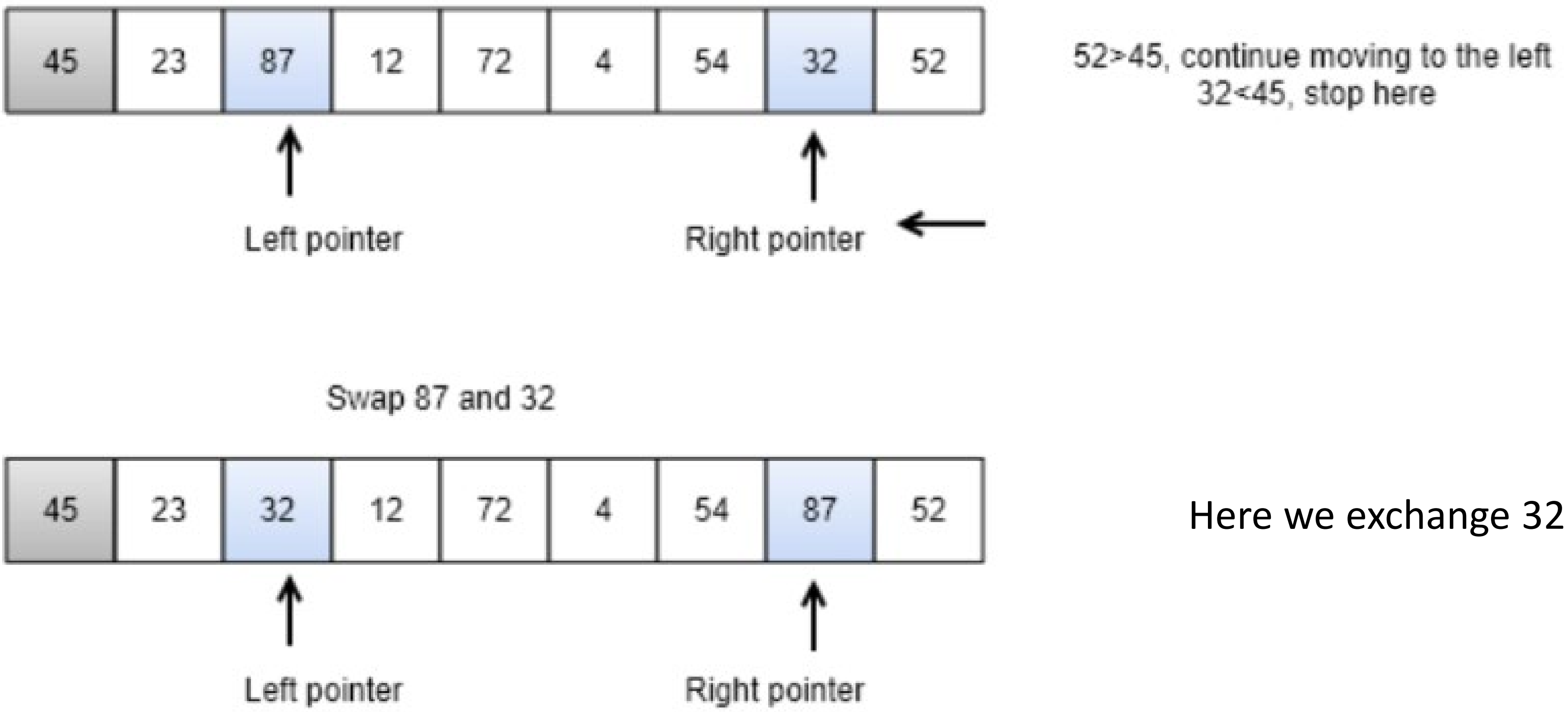


I want to find the sorted position of 45

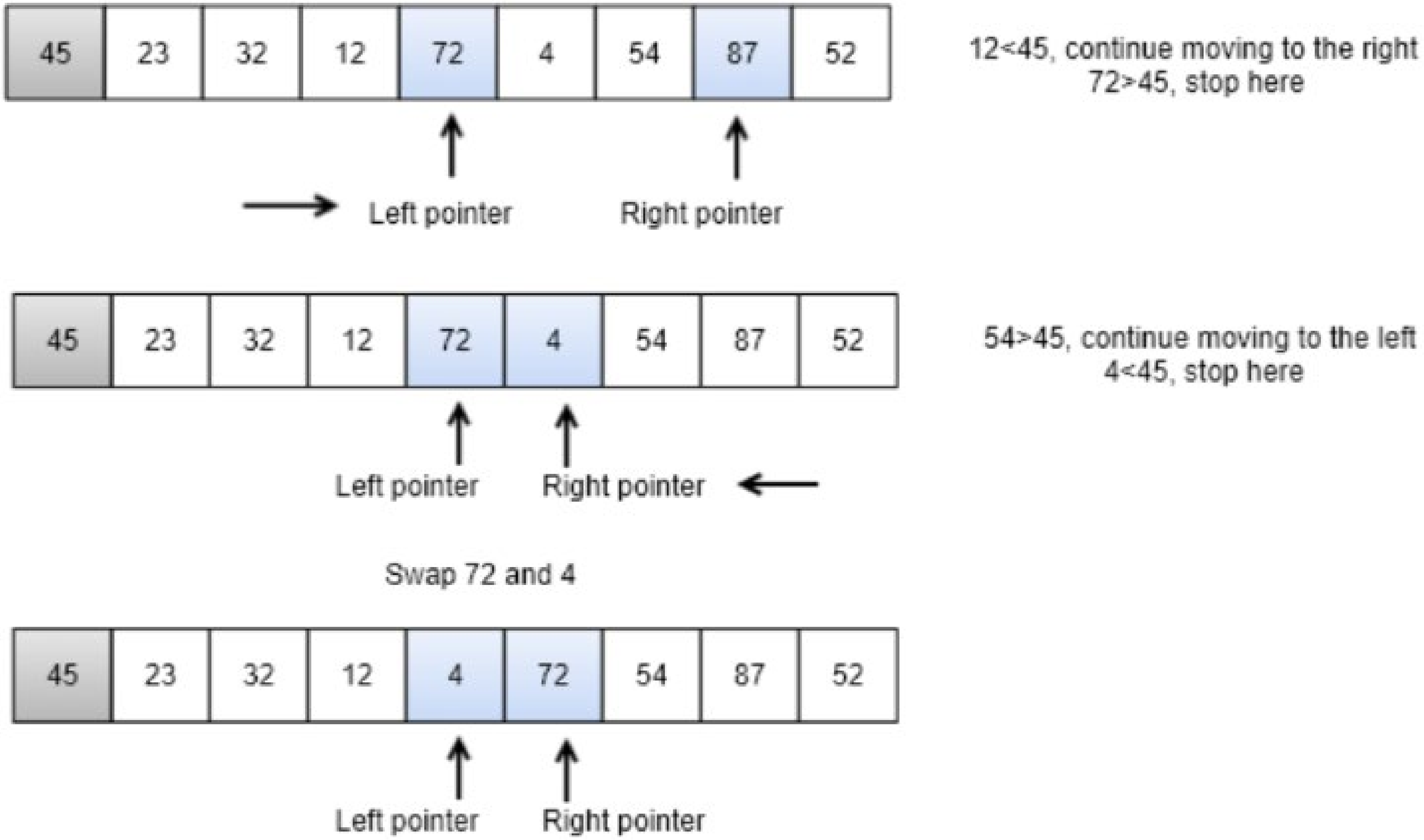
- i will search for elements greater than pivot
- j will search for elements smaller than pivot
- After finding the elements, exchange them

23<45, continue moving to the right
87<45, stop here

Example

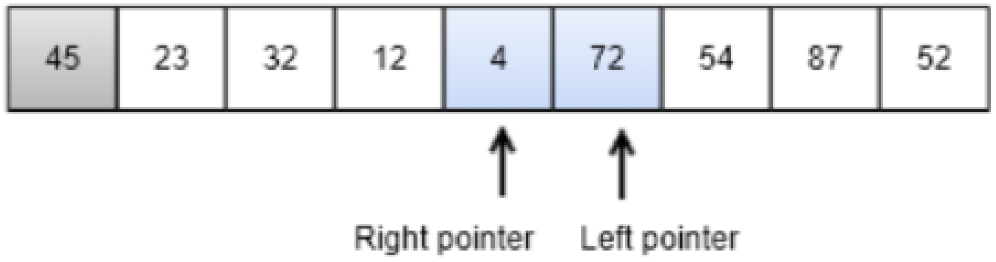


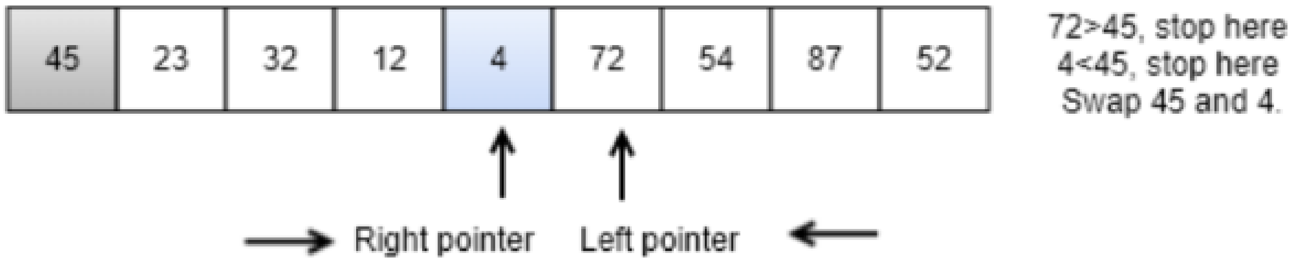
Here we exchange 32 with 87.

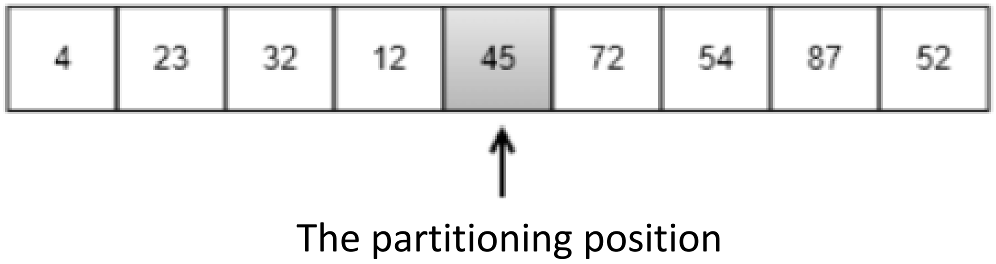


72 is greater than pivot
4 is smaller than pivot

We interchange them.







Now 45, the pivot, is sorted.

The next step is to perform quick sort recursively on both remaining sub-lists, left and right.
So following the partitioning method we will do the quick sort, recursively.

Quick Sort – code for partitioning

```
def partition(unsorted_array, first_index, last_index):
    pivot = unsorted_array[first_index] # here we store the value of the pivot
    pivot_index = first_index
    index_of_last_element = last_index
     # we create two variables for comparing i and j
     greater_than_pivot_index = first_index + 1  #i - my left pointer
    less_than_pivot_index = index_of_last_element  #j - my right pointer
  while True:
        while unsorted_array[greater_than_pivot_index] < pivot and greater_than_pivot_index
< last_index:
            greater_than_pivot_index += 1 # we increment i
        while unsorted_array[less_than_pivot_index] > pivot and less_than_pivot_index >=
first_index:
            less_than_pivot_index -= 1 # we decrement j
        if greater_than_pivot_index < less_than_pivot_index: # we swap i and j using temp
           # In such a condition, it means that greater_than_pivot_index and less_than_pivot_index have
crossed over each other.
            temp = unsorted_array[greater_than_pivot_index]
            unsorted_array[greater_than_pivot_index] = unsorted_array[less_than_pivot_index]
            unsorted_array[less_than_pivot_index] = temp
        else:
            break
# we interchange the element at unsorted_array[less_than_pivot_index] with that of the index of the pivot:
    unsorted_array[pivot_index] = unsorted_array[less_than_pivot_index]
    unsorted_array[less_than_pivot_index] = pivot
    return less_than_pivot_index
```

Quick Sort – code for sorting

- The quick_sort function is a very simple method, taking up no more than six lines of code. The heavy lifting is done by the partition function.
- When the partition method is called, it returns the partition point. This is the point in the unsorted_array array where all elements to the left are less than the pivot value, and all elements to its right are greater than it.

```
def quick_sort(unsorted_array, first, last):
    if last - first <= 0:
        return
    else:
        partition_point = partition(unsorted_array, first, last)
        quick_sort(unsorted_array, first, partition_point-1)
        quick_sort(unsorted_array, partition_point+1, last)

my_array = [43, 3, 77, 89, 4, 20]
print(my_array)
quick_sort(my_array, 0, 5)
print(my_array)
```

Big O – Sorting algorithms

Algorithm	Worst-case	Average-case	Best-case
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n)$
Insertion sort	$O(n^2)$	$O(n^2)$	$O(n)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Quicksort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

3. Selection Algorithms

- Given a list of elements, selection algorithms are used to find the k^{th} smallest element from the list – that number is called the **k^{th} order statistic.**
- In doing so, we shall be answering questions that have to do with selecting the median of a set of numbers, as well as selecting the k^{th} smallest or largest element in a list.
- Finding the mean of a list does not require the list to be ordered. However, finding the median in a list of numbers requires the list to be sorted (finding the median requires you to find the element in the middle position of the ordered list).

Selection by Sorting

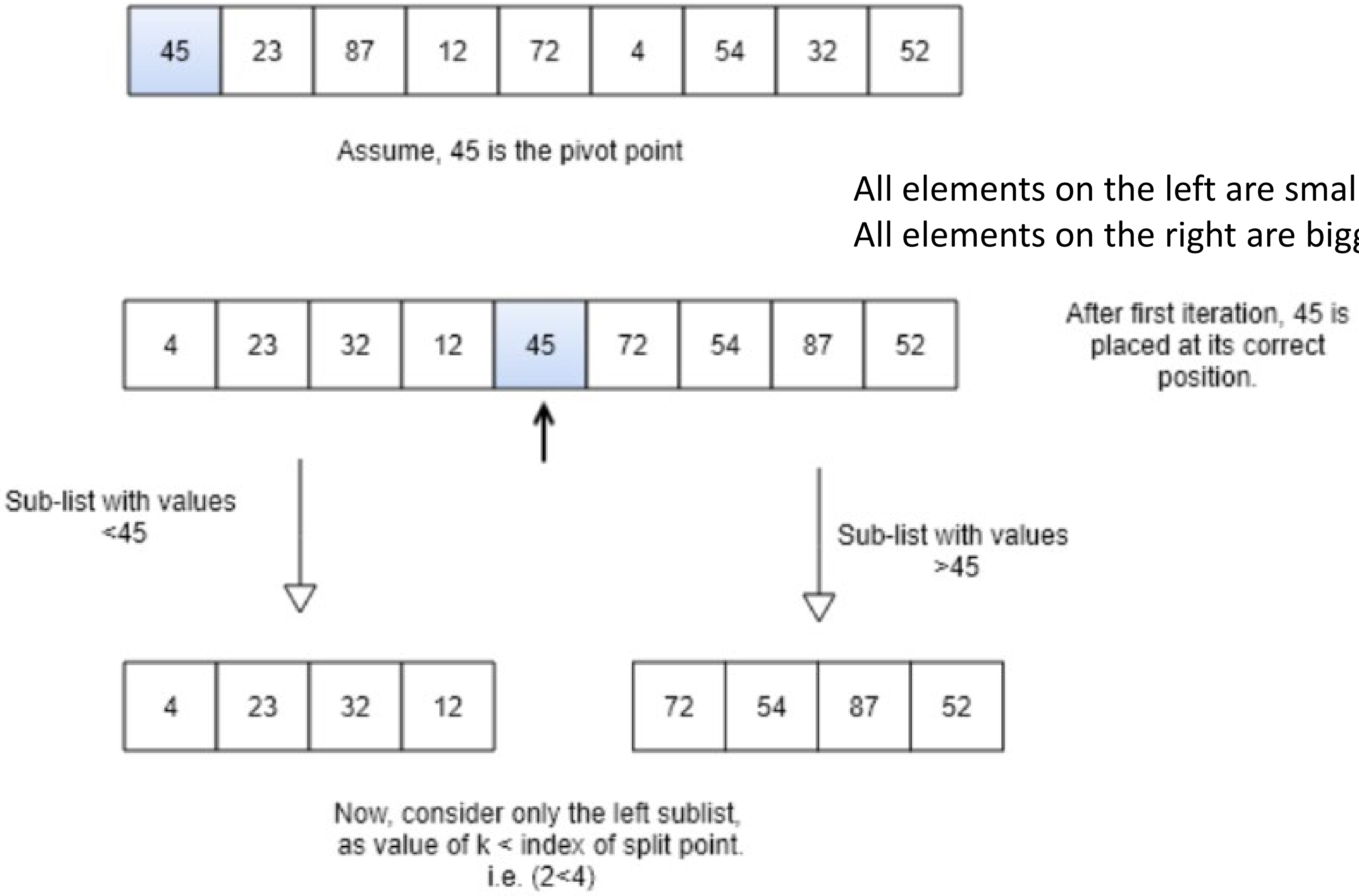
- A pragmatic and obvious thing to do when dealing with unordered lists is to first sort the list.
- After the list is sorted, you can rest assured that the element at the 0 index will hold the first-smallest element in the list. Likewise, the last element in the list will hold the last-smallest element in the list. And then you can find the median.
- However, it is not a good solution to apply a sorting algorithm on a long list of elements to obtain the minimum or maximum value from the list as sorting is quite an expensive operation.
- Instead of sorting the entire list, we can use partial sorting to select the k^{th} smallest (or largest) element in a list. Then the k^{th} smallest (or largest) is the largest (or smallest) element of the partially sorted list.

Randomized Selection (or Quick Select)

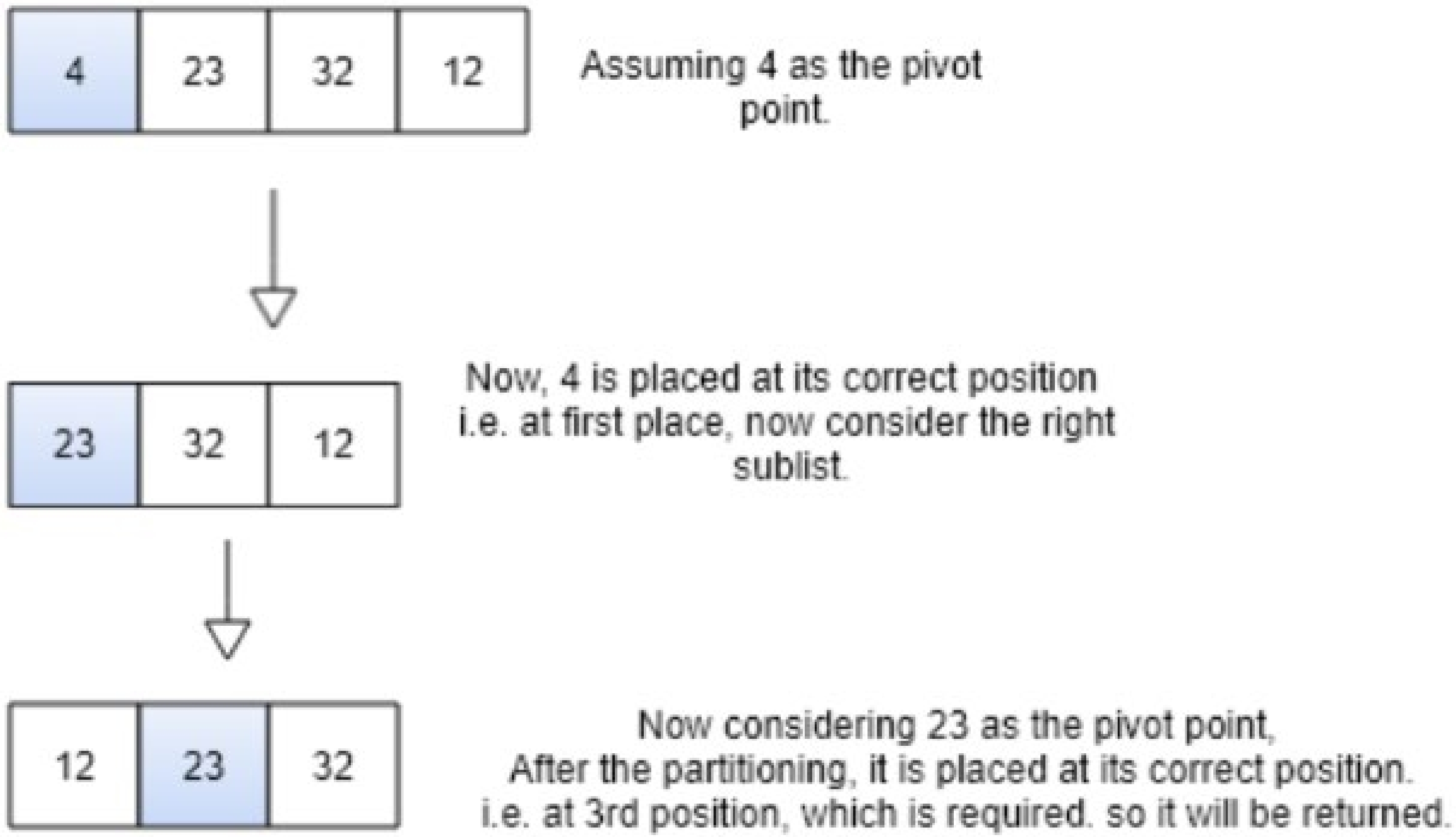
- The quickselect algorithm is used to obtain the k^{th} smallest element in an unordered list of items, and is based on the quicksort algorithm.
- The quicksort algorithm:
  1. Selects a pivot
  2. Partitions the unsorted list around the pivot
  3. Recursively sorts the two halves of the partitioned list using steps 1 and 2
- The quickselect algorithm for finding the k^{th} smallest element:
  - If the index of the pivot < k, then the k^{th} smallest value will be in the right sublist, so we apply the recursion only there;
  - If the index of the pivot > k, then the k^{th} smallest value will be in the left side from the pivot point, so we apply the recursion only there;
  - If the index of the pivot = k, then it means that we have found out the k^{th} smallest value , and we return it.

Finding the 3rd smallest element in the list (at index 2):

All elements on the left are smaller than pivot
All elements on the right are bigger than pivot



The index of the pivot point is greater than the 2nd index,
so we only recursively look for the 3rd smaller element in the left sub-list.



Randomized Selection - code

```
def quick_select(array_list, left, right, k):
    split = partition(array_list, left, right)

    if split == k:
        return array_list[split]
    elif split < k:
        return quick_select(array_list, split + 1, right, k)
    else:
        return quick_select(array_list, left, split-1, k)



stored = [5, 3]
print(stored)
print(quick_select(stored, 0, 1, 0))

stored = [3, 5]
print(stored)
print(quick_select(stored, 0, 1, 0))
```

```
stored = [3,1,10,4,6,5]
print(stored)
print(quick_select(stored, 0, 5, 0))
stored = [3,1,10,4,6, 5]
print(quick_select(stored, 0, 5, 1))
stored = [3,1,10,4,6, 5]
print(quick_select(stored, 0, 5, 2))
stored = [3,1,10,4,6, 5]
print(quick_select(stored, 0, 5, 3))
stored = [3,1,10,4,6, 5]
print(quick_select(stored, 0, 5, 4))
stored = [3,1,10,4,6, 5]
print(quick_select(stored, 0, 5, 5))
```

Thank you!