Quick Sort

Jim Gao, Tianqi Huang, Steven Ye, Sina Sabouri
October 5th, 2016
ICS4U
Mr. Anandarajan

Algorithm Classification

- **Divide and conquer** algorithm (recursive)
  - Divide and conquer: Breaking down a problem into 2 or more subproblems of similar types. Until the problems are simple enough to be solved directly.
- Splits a large array into 2 parts and sorts them recursively.

	Average Case	Worse Case
Time Complexity	$O(N \log N)$	$O(N^2)$
Space Complexity	$O(N)$	$O(N)$

Basic Procedure

- Given an unsorted array $\{a_1, a_2, a_3, \dots, a_N\}$
- **Partition** the array such that for some k
  - no larger element to the left of a_k
  - no smaller element to the right of a_k
- Recursively sort the subarrays by repeating the procedure
  - $\{a_1, a_2, \dots, a_{k-1}\}$, and
  - $\{a_{k+1}, a_{k+2}, \dots, a_N\}$

Basic Procedure

$$\{\ \underbrace{a_1, a_2, \ldots}_{\text{Less than pivot}}\ , \underset{\substack{\uparrow \\ \text{pivot}}}{a_k},\ \underbrace{\ldots\ , a_{N-1}, a_N}_{\text{Greater than pivot}}\ \}$$

$$\{a_1, a_2, \ldots\ a_{k-1}\} \qquad \{a_{k+1}, a_{k+2}, \ldots\ a_N\}$$

```java
public static void quickSort(int left, int right){
        // Check if the range is valid
        if (left >= right) return;

        // Pick the pivot by calling the partition method
        int pivotIndex = partition(left, right);

        // Recursively sort the subarray to the left and right
        // of the pivot
        quickSort(left, pivotIndex - 1);
        quickSort(pivotIndex + 1, right);
}
```

Partitioning

- Used to guarantee the requirement for recursive quicksort (relative monotonicity of the pivot and the left and right subarrays)
- Chooses a pivot and moves everything smaller than it to the left and everything larger than it to the right
- The pivot can be chosen at any arbitrary location

Algorithm

Part 1: method QuickSort (Recursive)

If the subarray being sorted has less than 2 elements

- Return because this subarray is already sorted

Otherwise,

- Do the partition for the subarray
- Quicksort the left hand side of the array
- Quicksort the right hand side of the array

Algorithm (cont'd)

Part 2: the partition method

Set pivot index as the leftmost index

Set (assume) pivot value as the last value

For every element before the pivot value

- If the element is less than pivot value
  - Swap the current value and the value on the pivot index
  - Increase the pivot index by 1

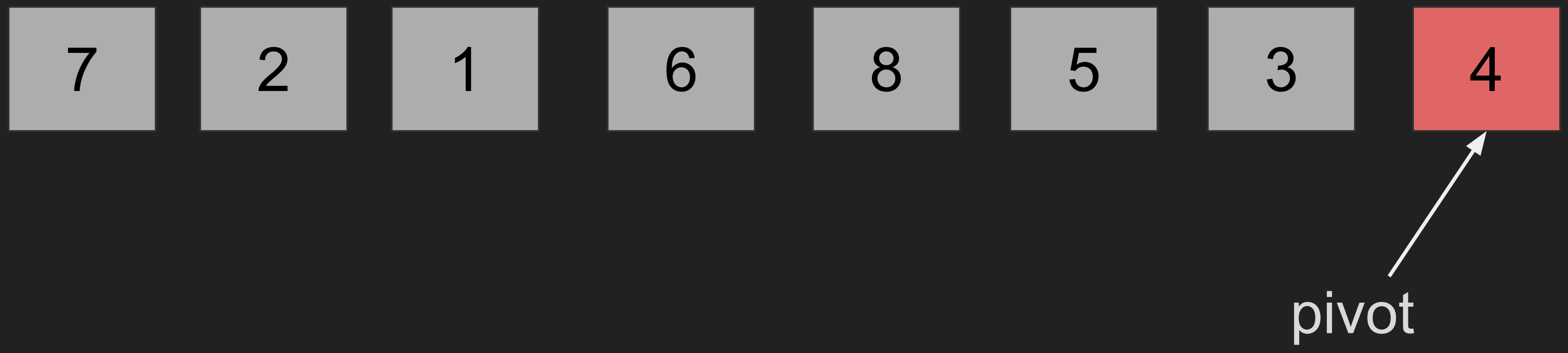
Return the value of pivot index

Implementation Tips

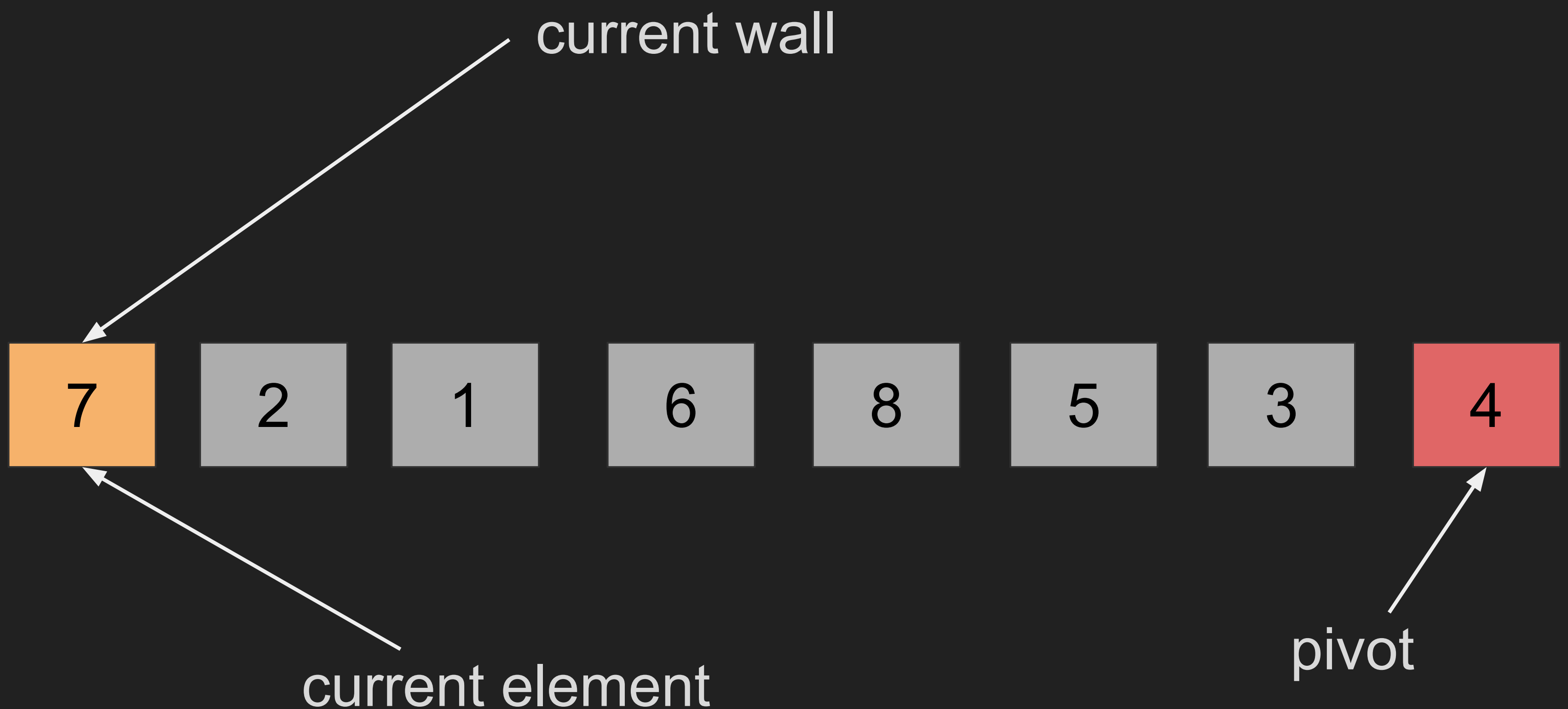
- Pass starting and ending indexes of sub-arrays as parameters between the methods
- Make the whole array static so every method can access it
- There are more ways to do the partitioning, but the general idea is still the same!

```java
public static int partition(int left, int right){
        // Make the right-most element the pivot
        int pIndex = left, pValue = numbers[right];

        // Go through every element and move the element if needed
        for (int i = left; i <= right - 1; i++){
            if (numbers[i] < pValue){
                swap(i, pIndex);
                pIndex ++;
            }
        }

        // Placing the pivot at the correct index
        swap(pIndex, right);
        return pIndex;
}
```

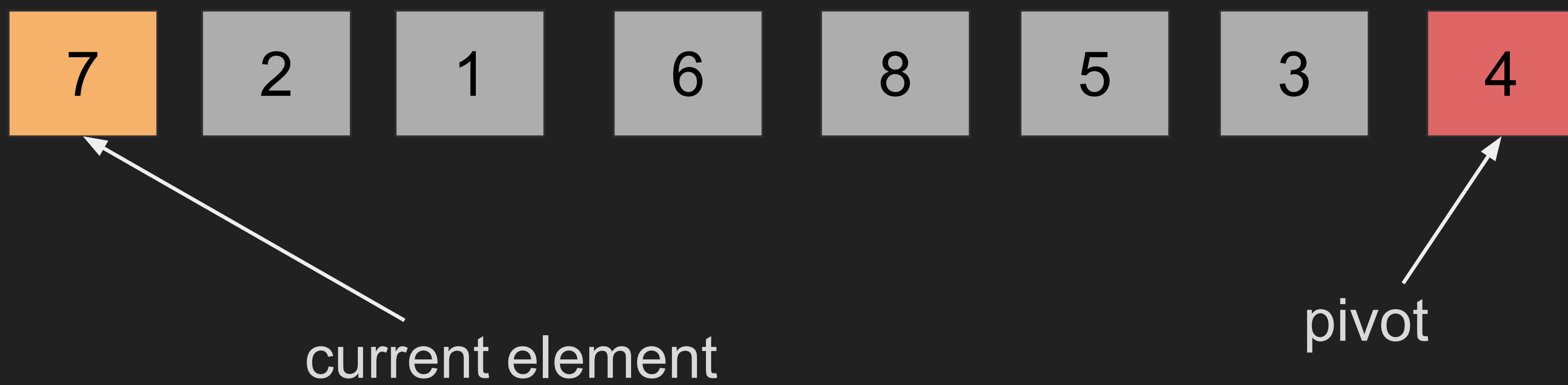
7	2	1	6	8	5	3	4

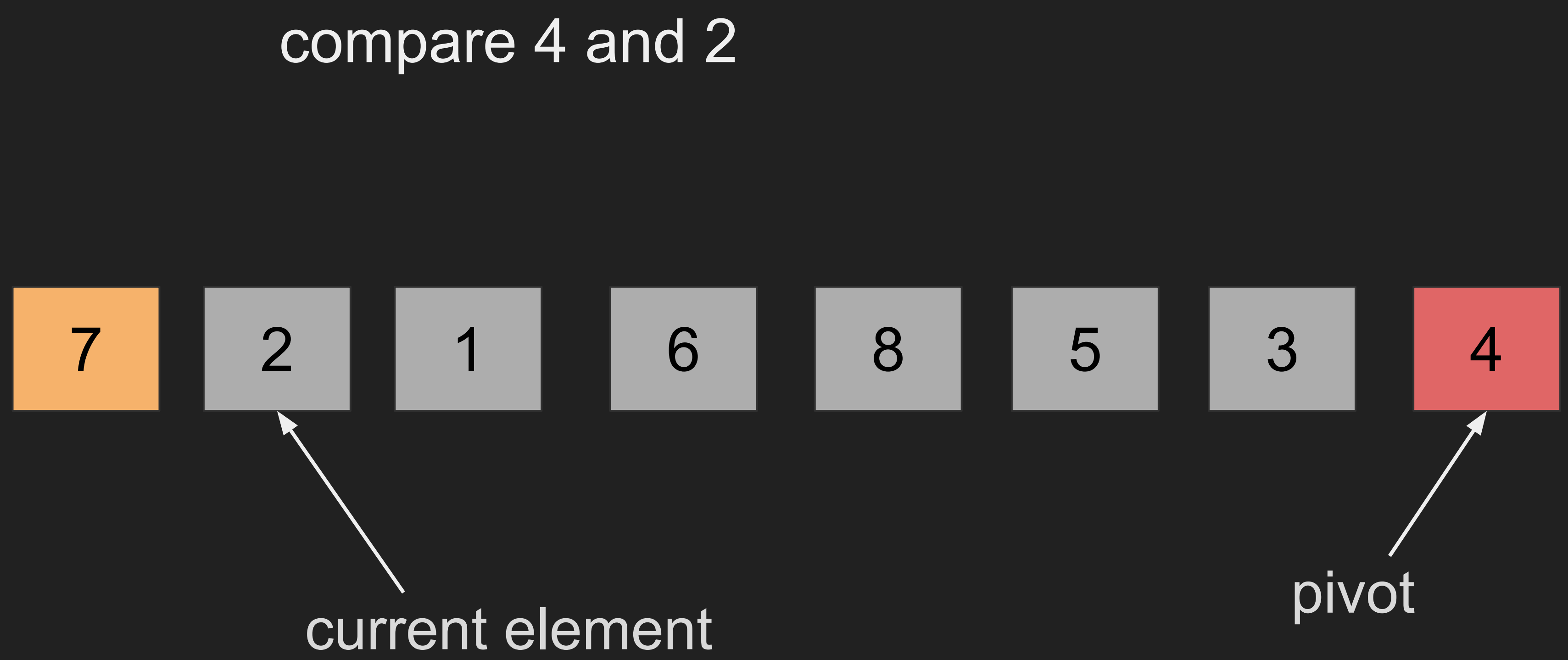


7
2
1
6
8
5
3
4
pivot

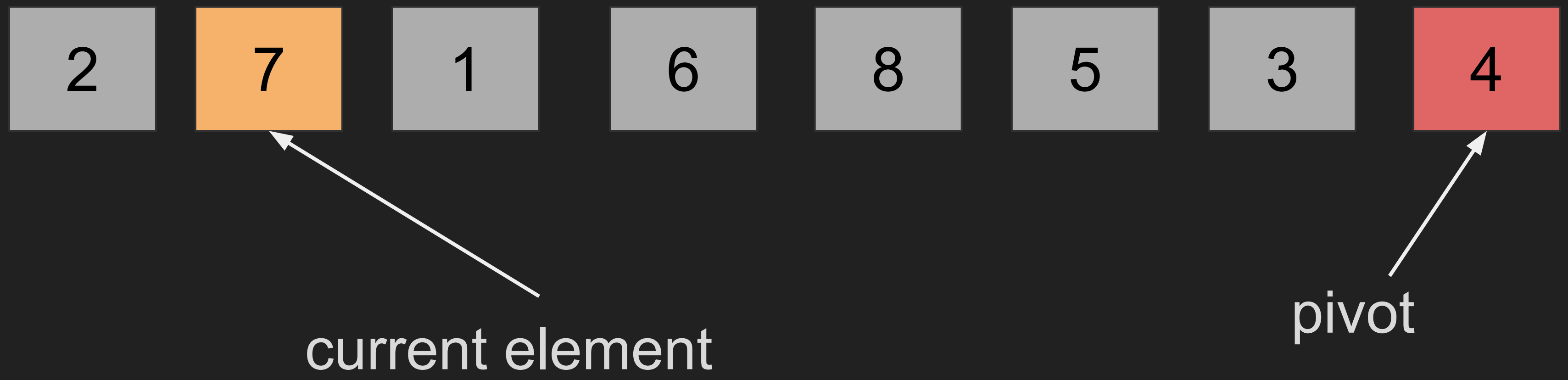

current wall
7
2
1
6
8
5
3
4
current element
pivot

Compare 4 and 7
(no swap)

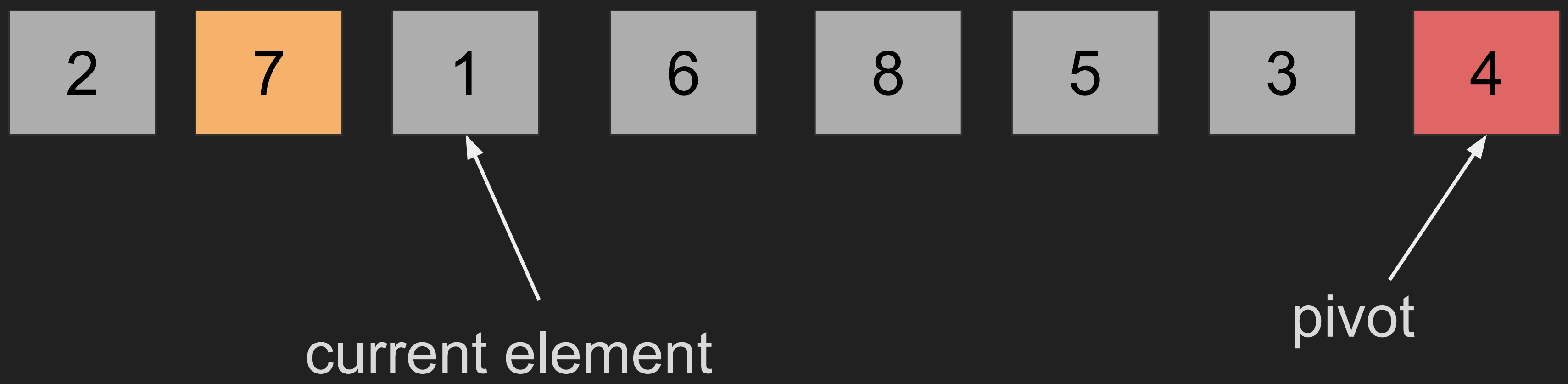



compare 4 and 2
7
2
1
6
8
5
3
4
current element
pivot

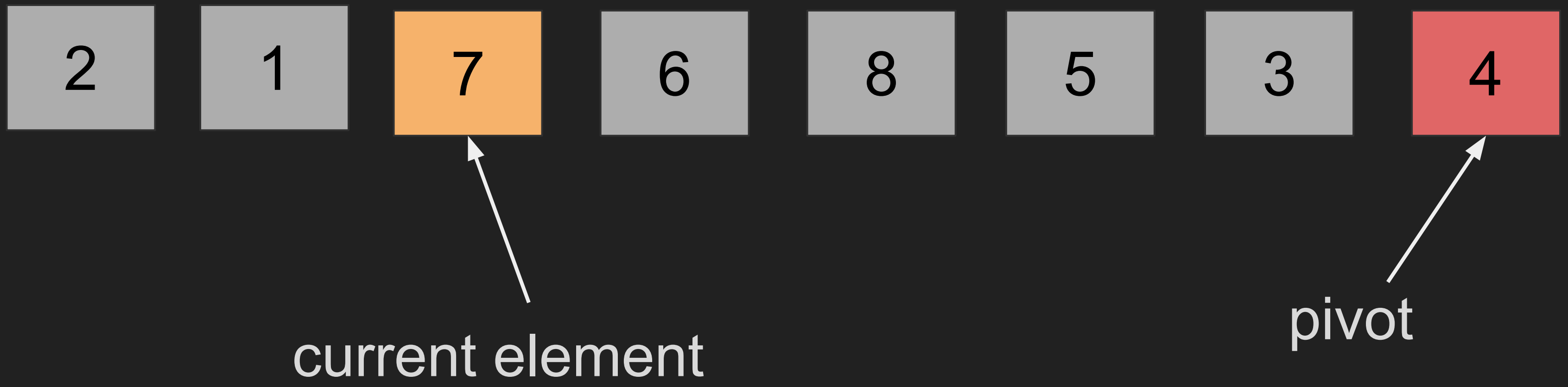
Swap 7 and 2



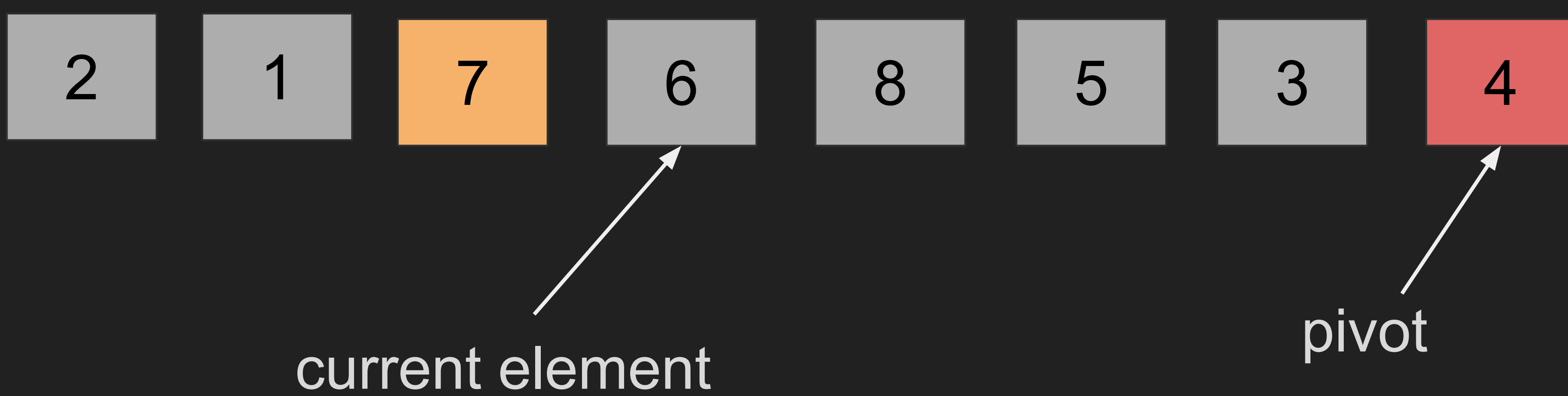
compare 4 and 1



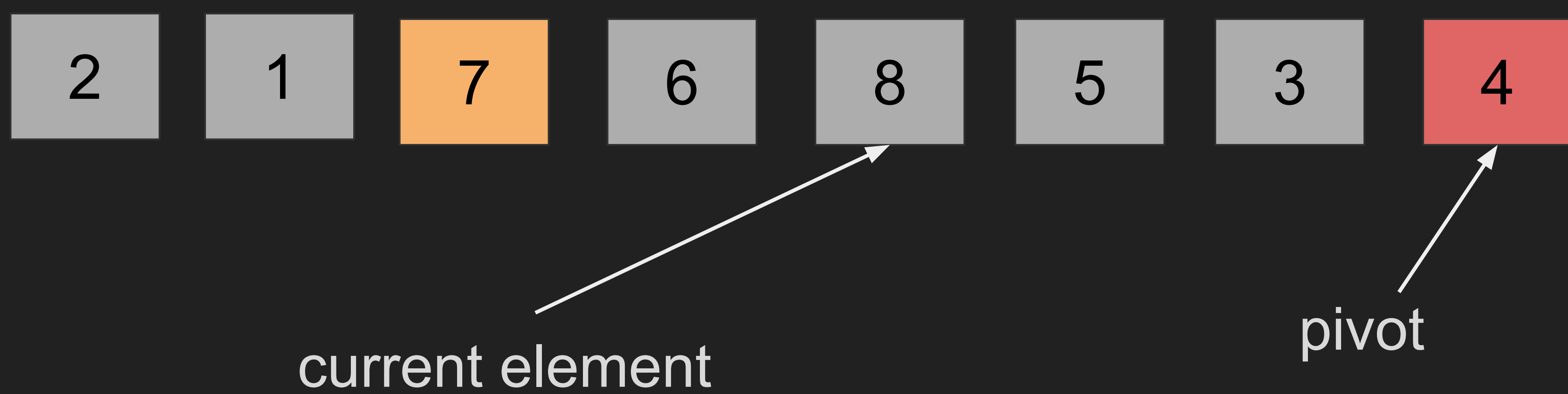
Swap 7 and 1

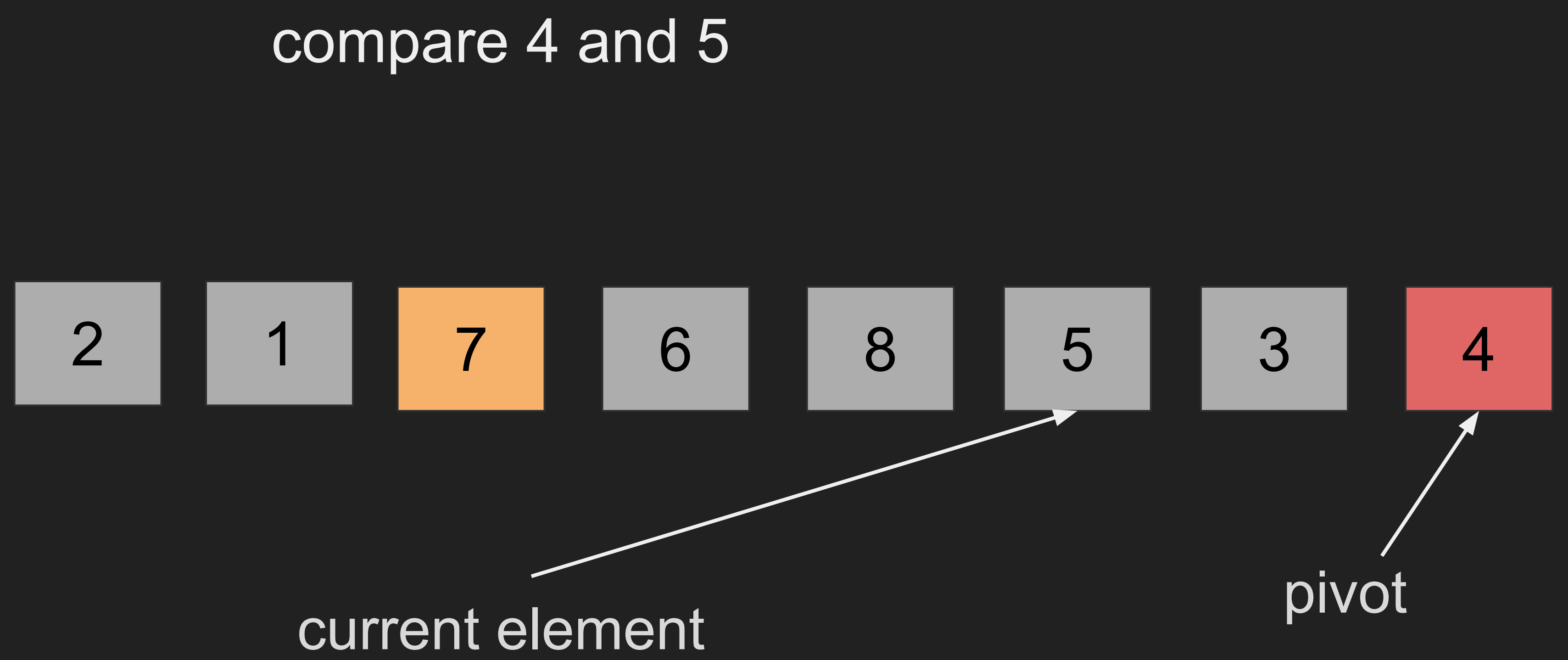


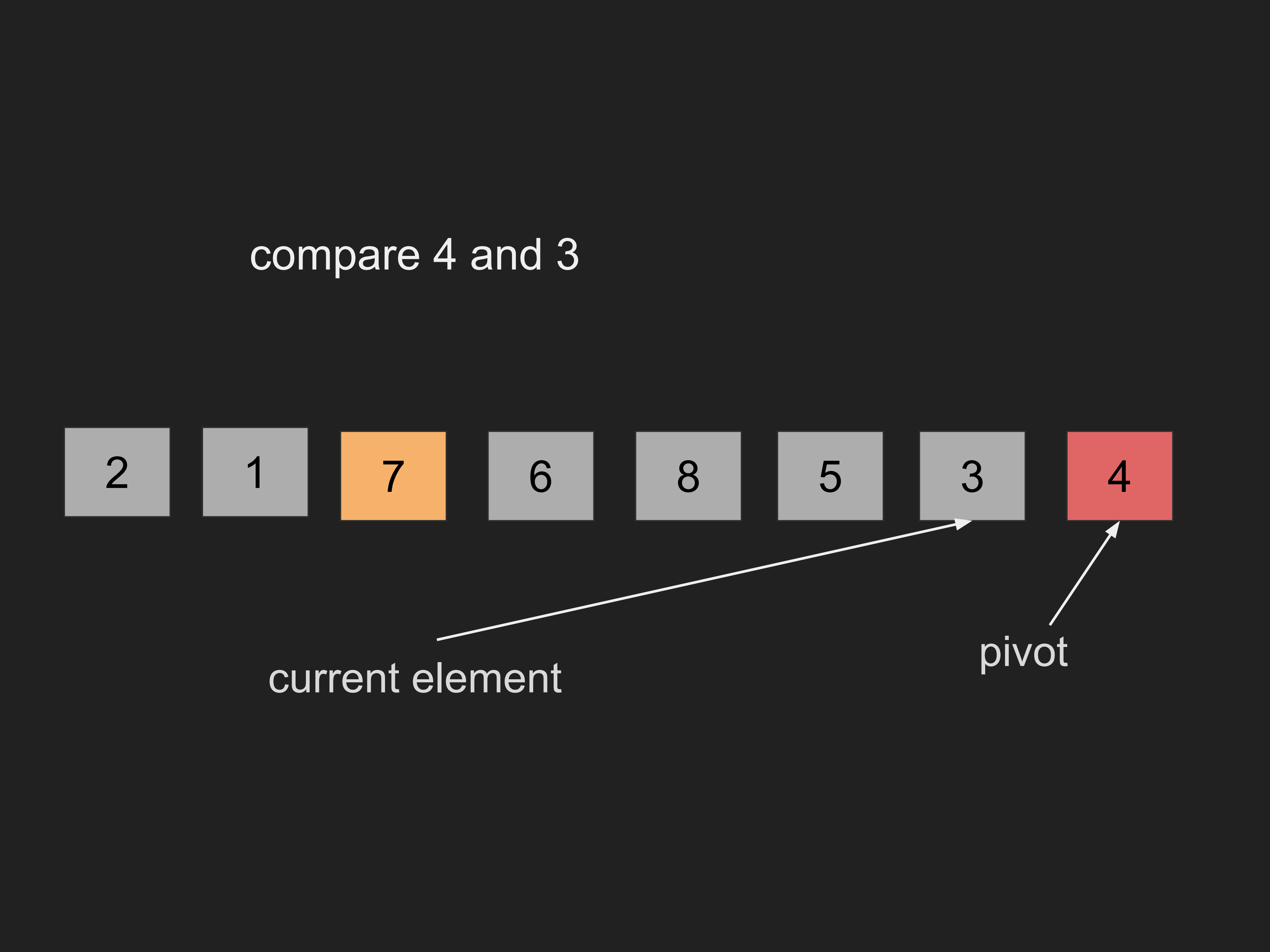
compare 4 and 6



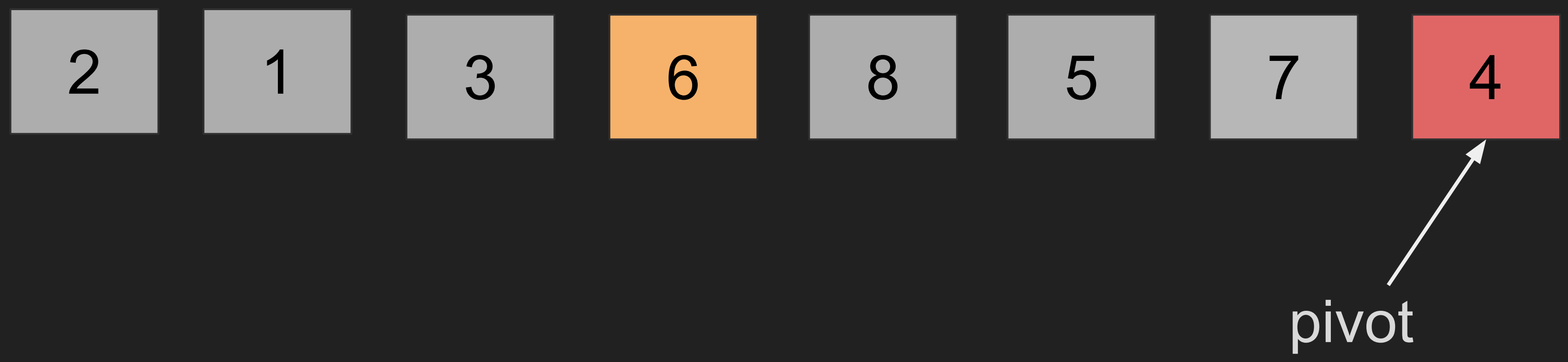
compare 4 and 8



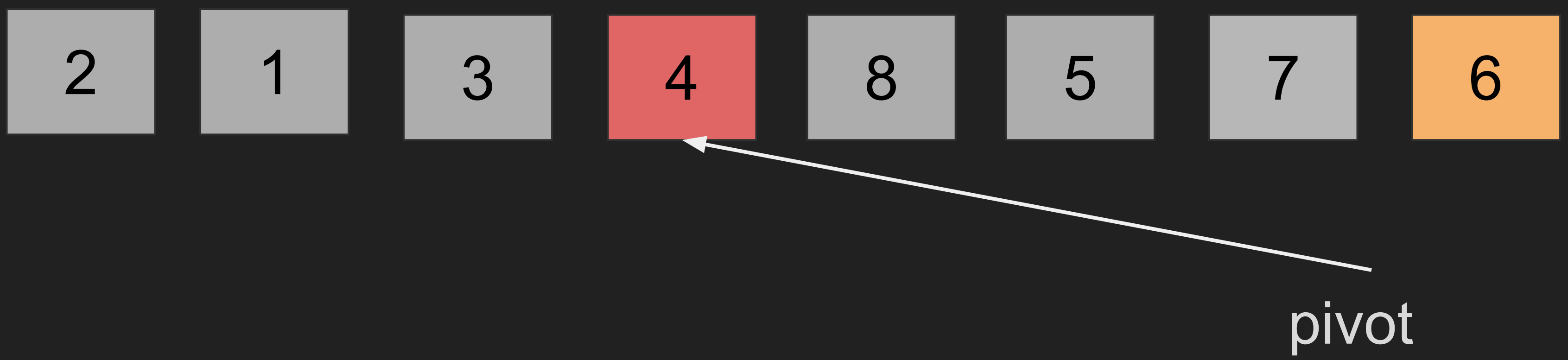

compare 4 and 5
2
1
7
6
8
5
3
4
current element
pivot

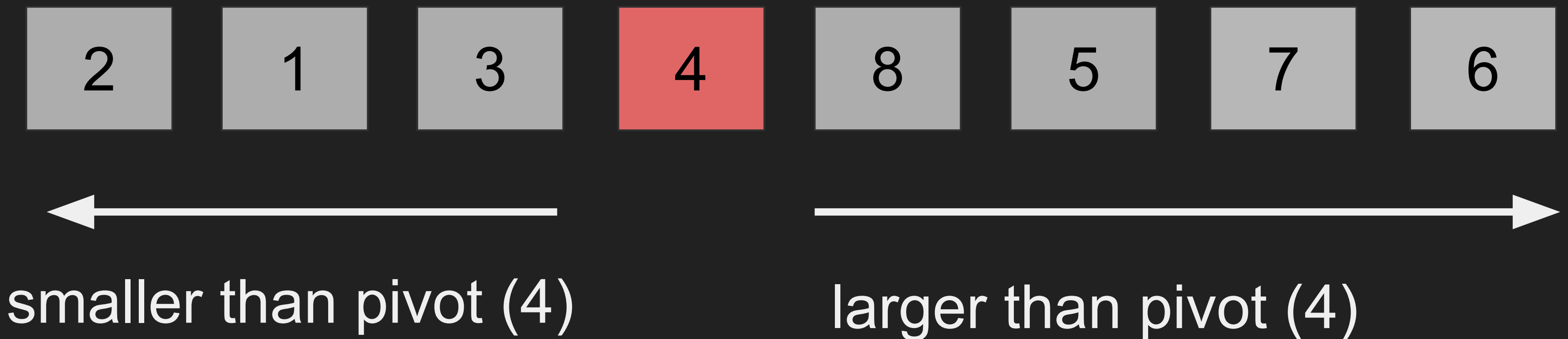

compare 4 and 3
2
1
7
6
8
5
3
4
current element
pivot

Swap 3 and 7



swap pivot with wall index




2
1
3
4
8
5
7
6
smaller than pivot (4)
larger than pivot (4)

```java
public static int partition(int left, int right){
        // Make the right-most element the pivot
        int pIndex = left, pValue = numbers[right];

        // Go through every element and move the element if needed
        for (int i = left; i <= right - 1; i++){
            if (numbers[i] < pValue){
                swap(i, pIndex);
                pIndex ++;
            }
        }

        // Finally placing the pivot at the correct index
        swap(pIndex, right);
        return pIndex;
}
```

Complexity Analysis

- Let C_K be the amount of operations required for K items.
- Since sorting an empty set or a set with 1 element does not require any operations, $C_0 = C_1 = 0$
- We can establish that:

partitioning time

$$C_N = (N + 1) + \left(\frac{C_0 + C_{N-1}}{N}\right) + \left(\frac{C_1 + C_{N-2}}{N}\right) + ... + \left(\frac{C_{N-1} + C_0}{N}\right)$$

partitioning

partitioning probability

$$C_N = (N+1) + \left(\frac{C_0 + C_{N-1}}{N}\right) + \left(\frac{C_1 + C_{N-2}}{N}\right) + \ldots + \left(\frac{C_{N-1} + C_0}{N}\right)$$

some (hardcore) math later…

$$C_N = 2(N+1)\left(\frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{N+1}\right)$$

Approximating the sum with an integral:

$$C_N \approx 2(N+1) \int_3^{N+1} \frac{1}{x} dx$$

and finally...

$$\boxed{C_N \approx 2(N+1) \ln N \approx 1.39 N \log N}$$

Best & Worst Case Scenarios

- **Best Case:** When pivot is close to the arithmetic mean of given range. Time Complexity: $O(N \log N)$
- **Worst Case:** When pivot is close to the minimum / maximum of given range. Time Complexity: $O(N^2)$
- **Average Case** Complexity: $O(N \log N)$

  (achieved by choosing random pivot)

Usage Scenario

Effective When:

- When sorting efficiency is required
- When additional memory is limited
- When stability is not required
- When average time complexity is more important than worst case

Not Effective When:

- Coding time is limited (more difficult to code)
    - Most languages implement quick sort (e.g. `Arrays.sort()`)

Sorting Visualization

Sources & References

Sedgewick, Robert. "Quick Sort." *Quick Sort* (2016): n. pag. Print.

Wild, Sebastian (2012). "Java 7's Dual Pivot Quicksort". Technische Universität Kaiserslautern.

Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990]. *Introduction to Algorithms* (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4.

Hoare, C. a. R. (1962-01-01). "Quicksort". *The Computer Journal*. **5** (1): 10–16. doi:10.1093/comjnl/5.1.10. ISSN 0010-4620.

"Sir Antony Hoare". Computer History Museum. Retrieved 22 April 2015.

Umut A. Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan, Quicksort and Sorting Lower Bounds, *Parallel and Sequential Data Structures and Algorithms*. 2013.