COMP20230: Data Structures & Algorithms **2018-19 Semester 2**

Lecture 14: Sorting (2)

Lecturer: Dr. Andrew Hines *Scribes: David O'Dwyer, Kieran Daly*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

Contents

1 Introduction

In §2 *Context* we cover the most general attributes that characterise sorting algorithms. These attributes are important both with respect to understanding their operation and with respect to the implications they have for CPU and memory usage. In §3 and §4 we cover *Quick* and *Merge Sort* in depth. Finally, in §5 we provide visualisations of the the algorithms under discussion.

2 Context

2.1 What is Sorting?

Sedgewick and Wayne define sorting as: "the process of rearranging a sequence of objects so as to put them in some logical order." [1] Sort-able digital objects are likely to be either numerical representations, characters, or custom objects; and they are likely to be present in a one-dimensional array.

The sorting algorithms covered in this module ought not to be thought of in terms of "good" or "bad", but rather in terms of the ordering challenges that they are well-suited to address. For instance, *Quick Sort* or *Merge Sort* are much more efficient at ordering large arrays than are *Selection Sort* or *Insertion Sort*; but, for sufficiently small array sizes (10-20 items), the latter outperform the former. They outperform to such an extent that it is common to incorporate them as sub-routines in *Quick* and *Merge Sort*, to handle small-input cases.[4] There are many factors that go into choosing which sorting algorithm to implement. The most common consideration is perhaps whether to optimise for CPU or memory efficiency.

It is not difficult to think of use-cases for sorting algorithms. Search operations are generally more efficient when applied to ordered inputs. Thus, sorting is an important features of data storage and management; just as it is of data processing, which requires that data be classified prior to operating on sub-sets of that data. A non-theoretical or "real-world" appreciation of the importance of sorting algorithm *efficiency* can be a little more opaque, however. A historical comparison is illustrative: in the early days of computing, sorting represented a substantial computational overhead as up to 30% of all CPU cycles were spend executing sorting procedures.[1] This is a large proportion of time spend not directly addressing the problem at hand.

2.2 Attributes of Sorting Algorithms

2.2.1 Comparative Vs. Non-Comparative

There are both comparative and non-comparative sorting algorithms. Comparative sorting algorithms compare the items that are to be ordered, while non-comparative don't. This insight is important for a number of reasons. Firstly, and simply, is the understanding that sorting is possible without comparison. *Radix Sort*, for instance, sorts items according to their integer keys, where it groups digits sharing the same position and value (i.e. it is based on numerical positional notation).[5] Secondly, is the difference in time complexity between non-/comparison algorithms. Taking *Quick Sort* and *Merge Sort* as our example, the best that these algorithms can perform (i.e. their lower bound time complexity) is Ω(nLogn): to sort N items, one must make minimally logN comparisons.[7] A non-comparison sorting algorithm such as *Counting Sort* can, however, perform at O(N) complexity.

This is something to be aware of. Note that *Quick Sort* and *Merge Sort* are both comparative sorting algorithms.

2.2.2 Adaptive Vs. Non-Adaptive

There are both adaptive and non-adaptive sorting algorithms. An adaptive algorithm can take advantage of existing order in the input array, consequently perform less work, and thus perform more efficiently. An adaptive algorithm is particularly welcome in the context of sorting, as somewhat-sorted inputs tend to be quite common in real world contexts.

Of the algorithms we have studied thus far, including those of this lecture, the adaptive and non-adaptive algorithms are as follows:

Adaptive

- Bubble
- Insertion
- Quick

Non-Adaptive

- Select
- Merge

2.2.3 Stable Vs. Unstable

A sorting algorithm can be either stable or unstable. A stable sorting algorithm maintains the relative order of items with equal values. In other words, if two items have the same sorting key – perhaps names are ordered by their first letter, so that ”Knuth” and ”Kernighan” are sorted into the Ks – the order they were found in the unsorted sequence is perserved. In our example, if ”Knuth” appeared anywhere before ”Kernighan” in the original sequence, it will appear before ”Kernighan” in the sorted sequence.[3]

This can be an import factor since some applications may want to preserve the original ordering in this respect. Of the algorithms we have studied thus far, including those of this lecture, the stable and unstable algorithms are as follows:

Stable

- Bubble
- Insertion
- Merge

Unstable

- Quick
- Selection

Note that any unstable sorting algorithm can be altered to become stable.[8]

2.2.4 In-place Vs Not-in-place

Sorting algorithms can perform their operation either in-place or not-in-place. An in-place sorting algorithm usually over-writes the input sequence with the algorithmic output, and consequently does not require additional memory in which to initialise a second data structure and write output into that structure. A small amount of additional memory is admissible for an in-place algorithm, to hold additional variables. This clearly results in more efficient memory usage.[6]

You are already likely familiar with this distinction based on Python's two built-in sorting methods. *sorted()* is not in-place: it returns a new sorted object in a distinct memory location. sequence*.sort()* is in-place, and mutates the original sequence object.

Of the algorithms we have studied thus far, including those of this lecture, the in-place and not-in-place algorithms are as follows:

In-place

- Bubble Sort
- Insertion Sort
- Selection Sort

Not-in-place

- Merge Sort
- Quick Sort

2.3 Merge and Quick Sort: Recursive Sorting Algorithms

Earlier in the course, we learned about recursion in the abstract. To recap, recursion can be defined as follows:

- Recursion: a programming technique whereby a function calls itself as a subroutine (i.e., during execution) one or more times – or a data structure relies on smaller instances of that same structure – until a specified condition is met. When the condition is met, the remaining repeated calls are processed from the last one called to the first.

We also learned that a recursive program has three characteristics. Firstly, it contains a "recursive case," or, the case of the program calling itself. Secondly, it contains a "base case," or, a part of the program that is defined non-recursively in terms of fixed values, which causes the recursive case to terminate. Finally, it contains the attribute that the the series of recursive calls makes progress towards the base case. In other words, it is finite: self-calling terminates, allowing the original problem to be solved.

Merge and *Quick Sort* are algorithms that are written according to this technique. They are recursive algorithms, and as such they implement the above characteristics. Broadly speaking, since the specifics of each are covered below, these algorithms contain recursive functions that repeatedly divide the input, unsorted array of elements. They contain a base case, which is the point at which the array cannot be sub-divided any further. And they progress, through division, to this point. In this overview, we ignore important aspects such as whether the sub-arrays are stored in-place or not-in-place.

In a nutshell, recursive sorting algorithms are very effective because it is less costly to sort a few small arrays than it is to sort one large array!

3 Quick Sort

3.1 High Level Overview of Algorithm

There are two parts to the Quick Sort algorithm:

1. Quick Sort takes advantage of the fact that arrays of one element are always sorted. It works by selecting a one element array (called the ”pivot”) and finding the index where the pivot should end up in a sorted array — it places all the elements smaller than it to the left and the elements greater than it to the right (although it should be noted that the elements on left side and on the right side are not sorted at this point).
2. Once the pivot is positioned appropriately, quick sort can be recursively applied to the both sides of the pivot.

   This leads to a new pivots being selected on either side of the array. The elements are again moved to correct side of the pivot. This continues recursively until the array is sorted.

3.2 Illustrative Overview

Consider the array: [8,3,5,4]

If we pick an arbitrary element (let’s say that in this case we pick the first element, 8); we move all the elements less than 8 to the left of it and all the elements greater than it (none in this case) to the right, we will have:

[3,5,4,8] — and we know than 8 is the correct location.

Note: We have selected the first element here, which will result in the worst case time complexity. This was common in the earliest implementations of Quick Sort. The reasoning for this is explained below, §3.5

Now we recursively repeat this process on the left side and the right side:

- Now 3 is moved to correct location (No movement in this case).
- Then 5 is moved to the correct location.

This gives us: [3,4,5,8]

- Then 4 is is moved to correct location (No movement in this case).

Now our array is sorted.

3.3 Pseudocode

The lecture notes present the pseudocode at high level (given below):

```
Algorithm quick sort:
Input: A an array
Output: A is sorted
if |A| > 1 then
            pivot  some element from A
            remove pivot element from L
            E.add(pivot)
            while A is not empty do
            elt  get first element of A and remove it
            if elt < pivot then
                S.add(elt)
            else
                if elt = pivot then
                    E.add(elt)
                else
                    G.add(elt)
                endif
            endif
    endwhile
    quick sort(S)
    quick sort(G)
    Reconstruct A by copying contents of S, E, G (in that order) back into A
endif
```

However these steps can be broken down into further detail. This detail will be shown below.

3.4 Implementation in Python

In order to implement quick sort, firstly it is necessary to implement a function responsible for arranging the elements in an array on either side of a pivot. Here we will call this helper function: Pivot. This corresponds to part 1 of the High Level Overview.

The Pivot function will accept three arguments: an array, a start index, and an end index (these can default to 0 and the array length minus 1, respectively).

The Pivot function will:

- Grab the pivot from the start of the array (the mid-point, or a random index are common)

- Store the current pivot index in a variable (this will keep track of where the pivot should end up)
- Loop through the array from the start until the end
- If the pivot is greater than the current element, increment the pivot index variable and then swap the current element with the element at the pivot index
- Swap the starting element (i.e. the pivot) with the pivot index
- Return the pivot index

Here is an implementation of the pivot function:

```
def pivot(list, start, end):

    pivot=list[start]
    swapIdx=start

    for i in range(start+1, len(list)):
#     check if pivot is greater than each element in list
        if pivot > list[i]:
            swapIdx+=1
# swap the element with index i with element with index swapIdx
            list[i], list[swapIdx]=list[swapIdx], list[i]
# swap the element at index start ie pivot with the element index swapIdx
    list[start], list[swapIdx]=list[swapIdx], list[start]
    return swapIdx
```

After implementing the Pivot function, implementing the rest of the Quick Sort algorithm is quite short. This corresponds to part 2 of the High Level Overview.

The Quick Sort Function will:

- Call the pivot function on the array
- When the pivot function returns the updated pivot index, recursively call the pivot function on the subarray to the left of that index, and the subarray to the right of that index
- The base case occurs when you consider a subarray with less than 2 elements

Here is an implementation of the Quick Sort function:

```
def quickSort(list,left,right):

    # when left becomes equal to right, the subarray is less than two elements.

    if (left<right):
```

```
#When the pivot function returns the updated pivot index, recursively call the pivot function on the

        pivotIndex=pivot(list,left,right)
        quickSort(list,left,pivotIndex-1)
        quickSort(list,pivotIndex+1,right)
    return list
```

3.5 Time Complexity

Quick Sort's efficiency depends on how well partitioning divides the array, because this determines how many compare operations are required. On average, the partitioning approaches the best case, and the array is subdivided 50 / 50 with each decision. This results in O(NlogN) time complexity. [1]

In the implementation, above, the pivot is chosen to be the first item. This would result in the worse case time complexity of $O(N\hat{2})$. This is because the unbalanced partition results repeatedly in a large array that is one item shorter than the previous long array. In fact, this is the implementation that was used commonly in the 'very early versions of quicksort.' [2]

4 Merge Sort

4.1 High Level Overview of Algorithm

Recursive algorithms are also known as "Divide and Conquer" algorithms, and this metaphor is particularly helpful in understanding Merge Sort. The algorithm can be though of as having two distinct phases. 1) Divide: the original array of elements is subdivided into two arrays, and these in turn are independently subdivided in similar manner, until all that is left of the original array are single items. 2) Conquer: the sorting problem is "conquered" by this phase, which is the merging phase from which the algorithm gets its name. The multiple single-items sub-arrays (the result of the Divide phase) are merged or re-combined into new arrays (pairwise and repeatedly) according to a rule that results in the smaller value occupying the lower index of a given new array. This Conquer / Merge phase takes several steps, and the result is a new finished array, containing all the original array items, sorted from low-to-high.

4.2 Illustrative Overview

Consider the array:

[8,3,5,4]

This can be divided into 2 smaller arrays:

[8,3] [5,4]

and again into 4 single arrays each with one item:

[8] [3] [5] [4]

We can then merge the first array with the second and the third array with the fourth. Ensuring that we put the smaller number first in the merged array:

[3,8] [4,5]

We can then merge these 2 arrays together, taking advantage of the fact that they are already sorted. When merging, we make comparisons to ensure the items are in the correct order in the merged array.

[3,4,5,8]

4.3 Pseudocode

The lecture notes present the pseudocode at high level (given below):

```
Input: A an array
Output: A is sorted
if |A| > 1 then
    for j  0 to |A|/2 do
        add A[j] to A1
    endfor

    for j |A|/2 +1 to |A| do
        add A[j] to A2
    endfor

    merge sort(A1)
    merge sort(A2)
    A = merge(A1, A2)
endif
return A
```

However these steps can be broken down into further detail. This detail will be shown below.

4.4 Implementation in Python

In order to implement merge sort, firstly it is necessary to implement a helper function which merges two arrays into a new array. This corresponds to ”conquer” phase of the algorithm as outlined above. Here we will call this helper function: Merge.

The ”divide” phase will be carried out by the main function: Merge Sort. This main function will also recursively call the helper function in order to implement the ”conquer” phase.

This helper function Merge will:

- Create an empty array (this will be our output array)

- Take a look the smallest values in each input array
- While there are still values we haven't looked at:
  - If the value in the first input array is smaller than the value in the second input array, push the value in the first array into our output array and move on to the next value in the first array
  - If the value in the first array is larger than the value in the second array, push the value in the second array into our results and move on to the next value in the second array
  - Once we exhaust one array, push all remaining values from the other array into our output array.

Here is an implementation of the Merge function:

```
def merge(list1, list2):
    results=[]
    i=0;
    j=0;

    while i<len(list1) and j<len(list2):
# Compare the jth item in list 2 and the ith item in list 1
# Append the smaller one to the results list
        if list2[j]>list1[i]:
            results.append(list1[i])
            i+=1
        else:
            results.append(list2[j])
            j+=1

# At the end of the loop, we still have the left over elements from the longer list
# We now append these to the results list

    while i<len(list1):
        results.append(list1[i])
        i+=1

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

    return results
```

After implementing the helper Merge function, implementing the rest of the Merge Sort algorithm is quite short.

We will now implement this main function: Merge Sort. The Merge Sort function will contain the logic for the "divide" phase. It will also recursively call the Merge Function (which implements the "conquer" phase).

The Merge Sort Function will:

- Break up the array into halves until you have arrays that are empty or have one element ("divide")

- Once you have smaller sorted arrays, merge those arrays with other sorted arrays until you are back at the full length of the array ("conquer"- by recursively calling the helper function)
- Once the array has been merged back together, return the merged (and sorted) array

Here is an implementation of the Merge Sort function:

```
def mergeSort(list):
#      base case
    if len(list)<=1:
        return list

#     find mid point
    mid = math.floor(len(list)/2)
#     recursively take a take a slice of all values left of mid point
    left=mergeSort(list[0:mid])
#     recursively take a take a slice of all values right of mid point
    right=mergeSort(list[mid:])
# Use the merge function to combine the left and right slices
    return merge(left,right)
```

4.5 Time Complexity

The worst (and average) case time complexity for Merge Sort is O(nlog n), which is a fast, *guaranteed* running time (recall that Quick Sort is O(n^2)).

Consider, first, splitting the array. Let's say we start with 4 items in an array; how many times do we need to split the the array to get single item arrays? In this case the answer is 2 times. If we had an array of 8 items, we would need to split 3 times. That is, log2(8 items)=3 splits. This means that Big O of splitting is O(log n) Consider now merging the array.

In order to merge the elements together in the correct order, we need to make n comparisons for each decomposition. We therefore have an overall complexity of O(n) for comparisons.

Taken together, Merge Sort thus has a worst case time complexity of O(nlogn).

References

[1] Sedgewick, R.; Wayne, K.; ”Algorithms” (Pearson Education, 2011)
`(Pearson Education, 2011)`

[2] https://en.wikipedia.org/wiki/Quicksort
`(Wikipedia)`

[3] Goodrich, M.T.; Tamassia, R.; Goldwasser, M.H; ”Data Structures Algorithms in Python”
`(Pearson Education, 2011)`

[4] Lecture Notes: Sorting Algorithm
`https://www.cpp.edu/ ftang/courses/CS241/notes/sorting.htm`

[5] Radix Sort
`https://en.wikipedia.org/wiki/Radix`$_{s}ort$

[6] In-Place Algorithm
`https://en.wikipedia.org/wiki/In-place`$_{a}lgorithm$

[7] Radix Sort
`https://www.geeksforgeeks.org/radix-sort/`

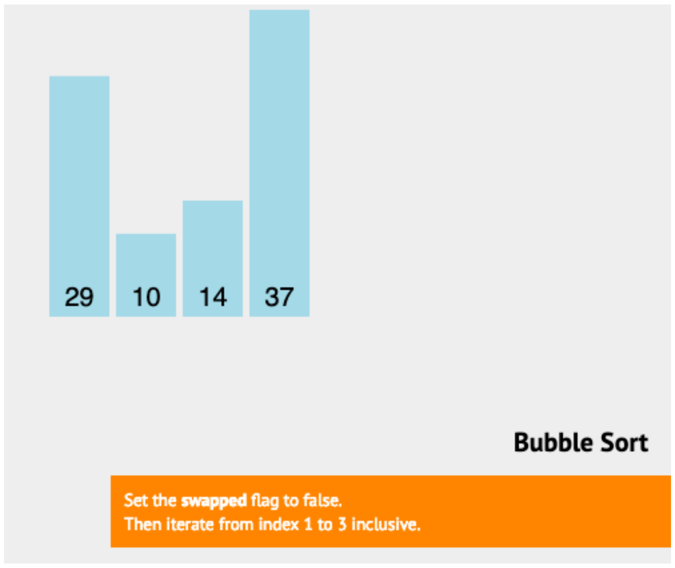
[8] Stability in Sorting Algorithms
`https://www.geeksforgeeks.org/stability-in-sorting-algorithms/`

[9] Colt Steele (2018), JavaScript Algorithms and Data Structures Masterclass (Online Course: Udemy.com)

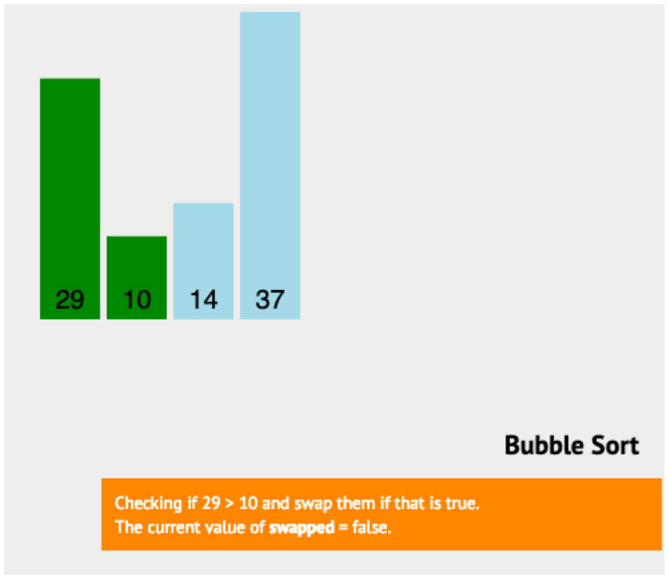
[10] Visualgo.net- Sorting Tutorial (2019) Retrieved from https://visualgo.net/en/sorting?slide=1

5 Visual Walkthrough of Sorting Algorithms

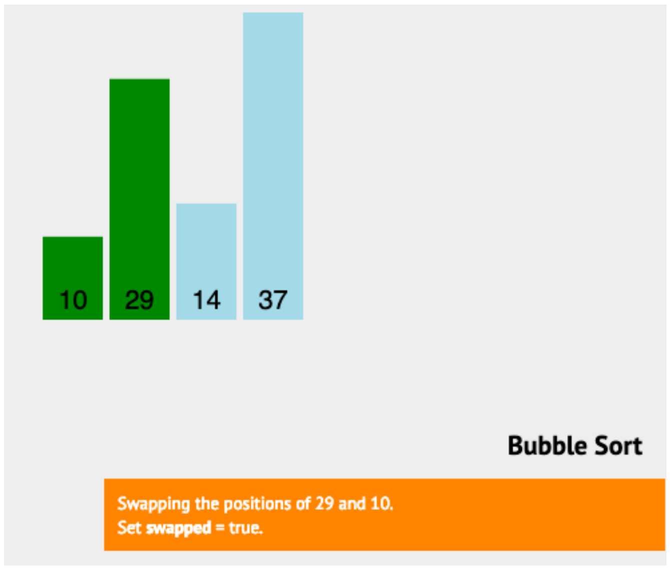
The website Visualgo provides detailed visualisations of different algorithms and data structures. Visualisations for quick sort and merge sort have been included at the end of this document.



Set the **swapped** flag to false.
Then iterate from index 1 to 3 inclusive.

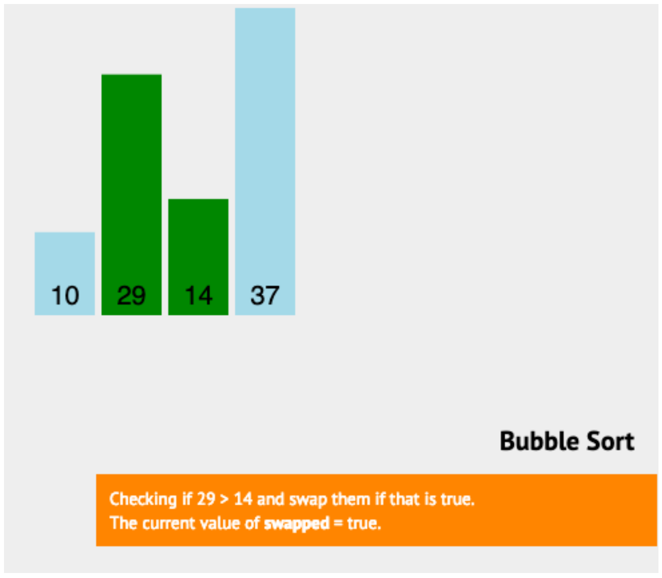


Checking if 29 > 10 and swap them if that is true.
The current value of **swapped** = false.



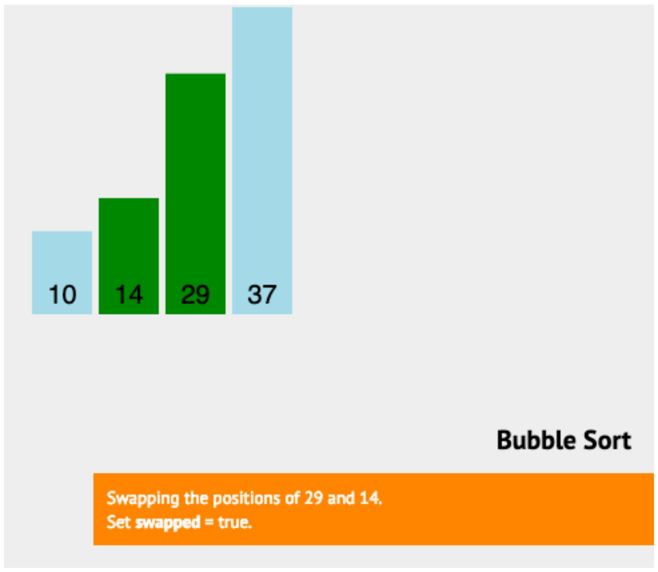
Bubble Sort

Swapping the positions of 29 and 10.
Set **swapped** = true.

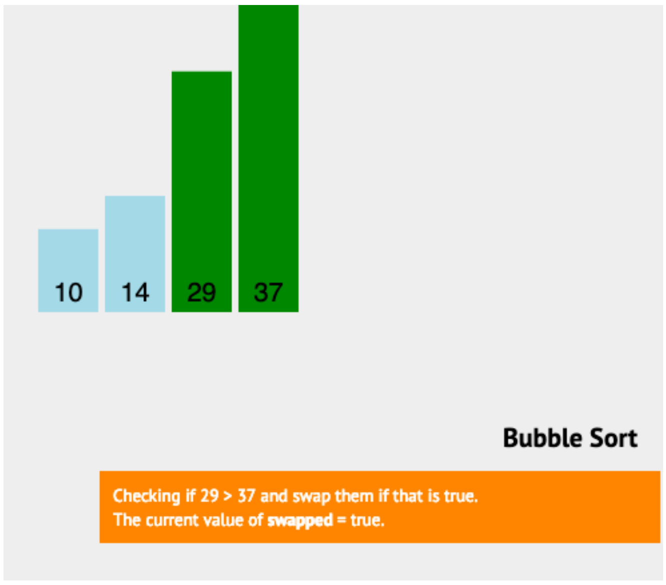


Bubble Sort

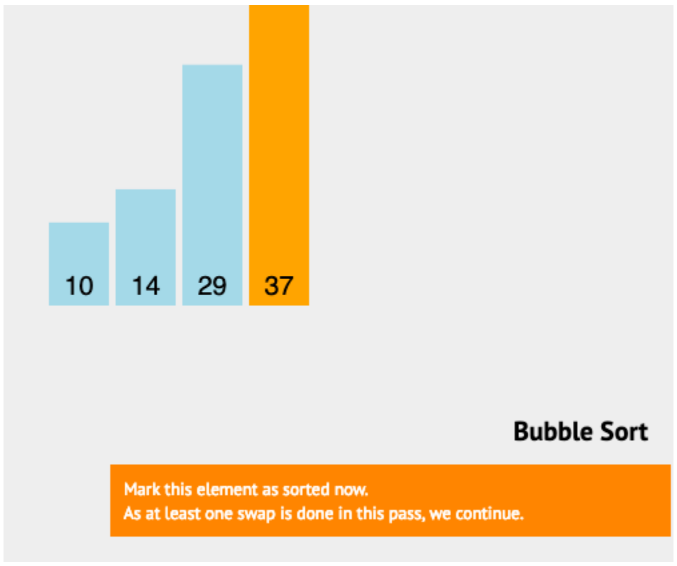
Checking if 29 > 14 and swap them if that is true.
The current value of **swapped** = true.



Swapping the positions of 29 and 14.
Set **swapped** = true.

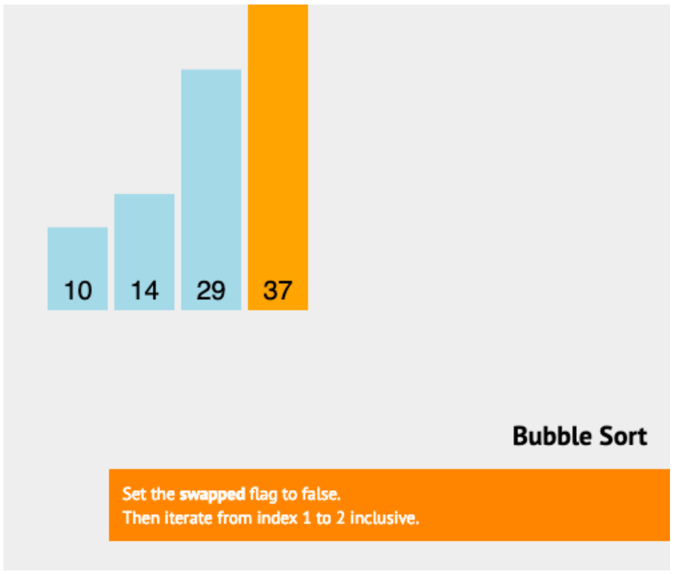


Checking if 29 > 37 and swap them if that is true.
The current value of **swapped** = true.



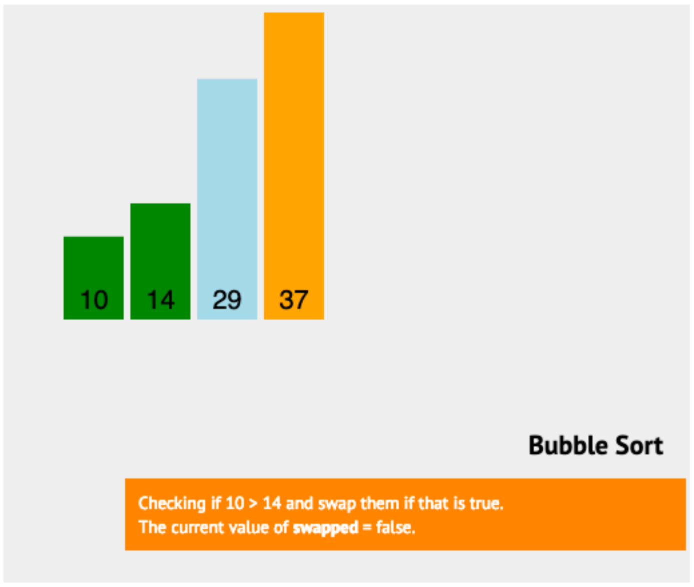
Bubble Sort

Mark this element as sorted now.
As at least one swap is done in this pass, we continue.



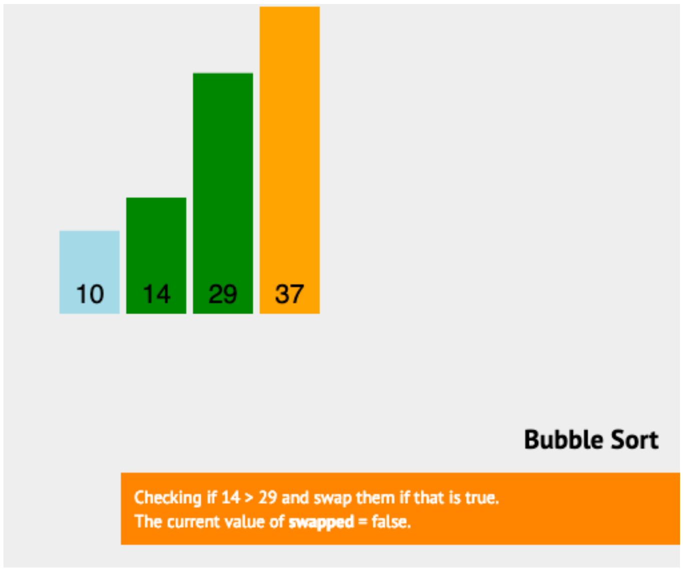
Bubble Sort

Set the **swapped** flag to false.
Then iterate from index 1 to 2 inclusive.



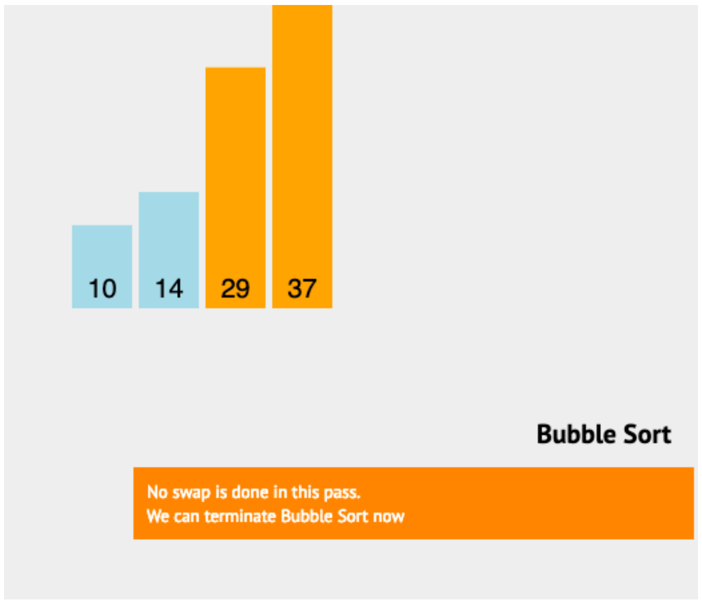
Bubble Sort

Checking if 10 > 14 and swap them if that is true.
The current value of **swapped** = false.



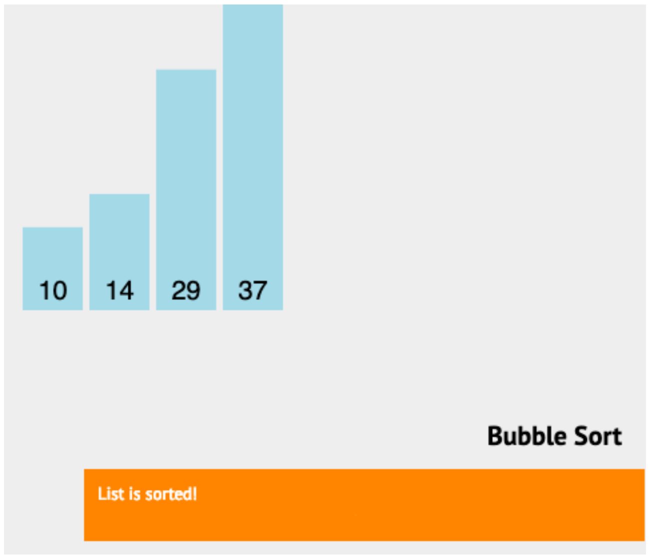
Bubble Sort

Checking if 14 > 29 and swap them if that is true.
The current value of **swapped** = false.



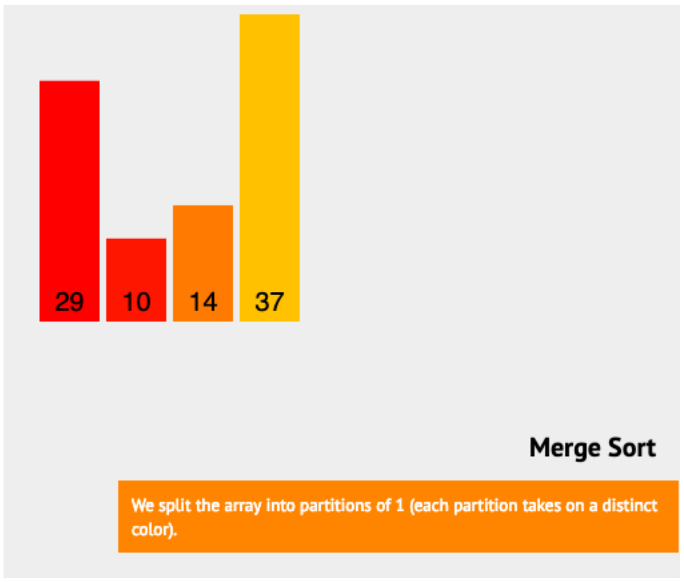
Bubble Sort

No swap is done in this pass.
We can terminate Bubble Sort now



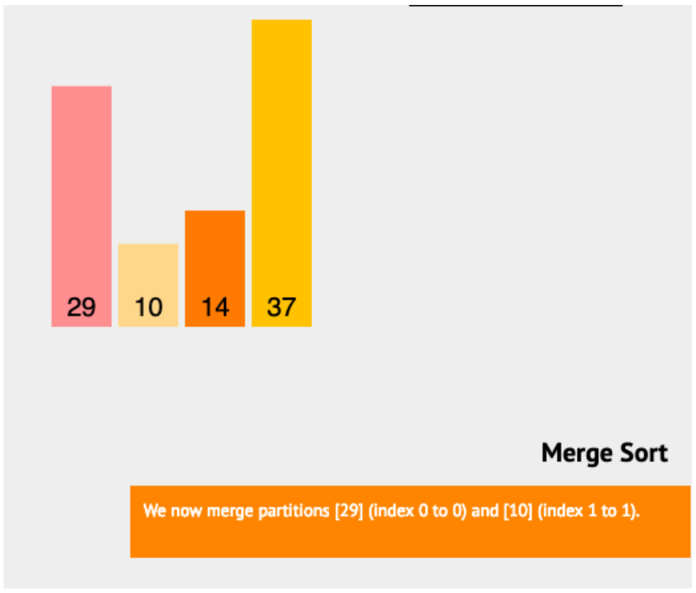
Bubble Sort

List is sorted!



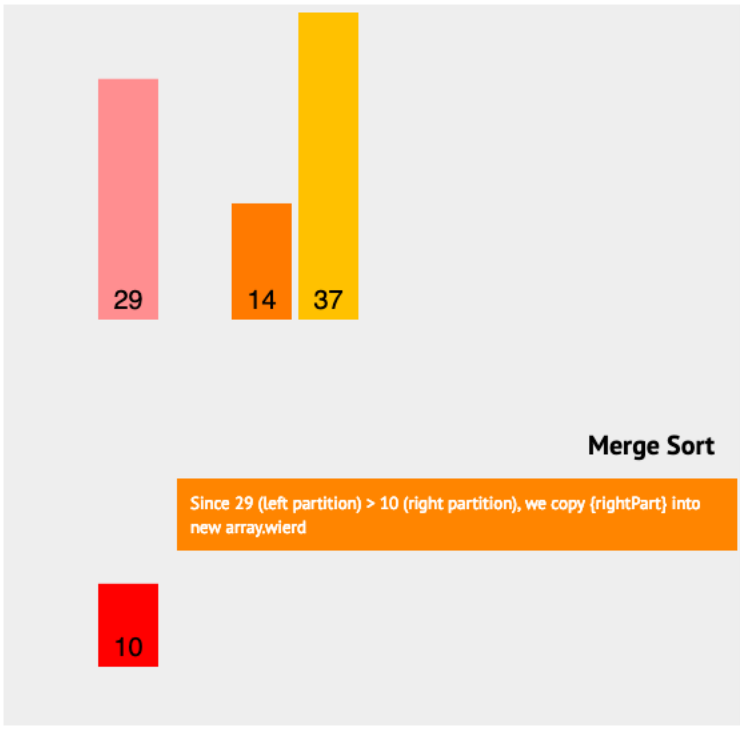
Merge Sort

We split the array into partitions of 1 (each partition takes on a distinct color).



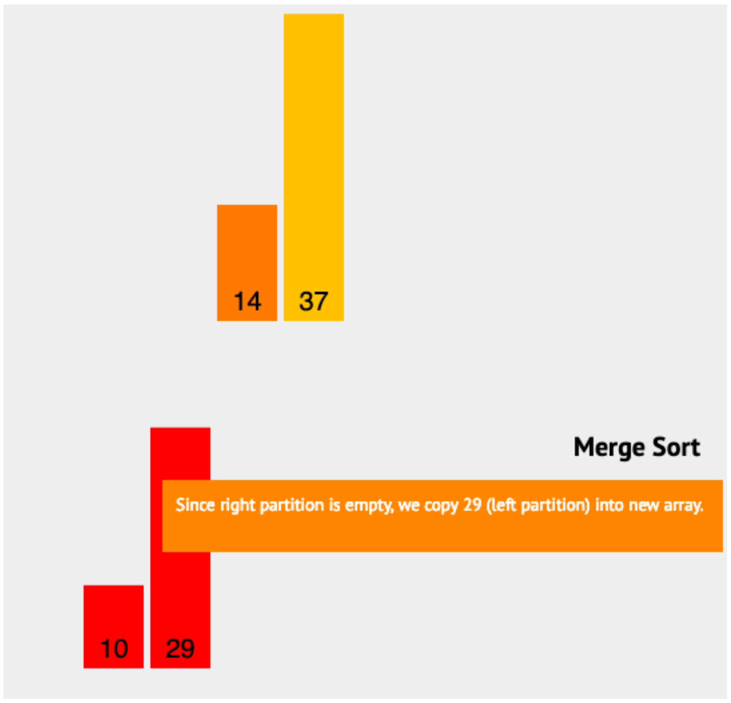
Merge Sort

We now merge partitions [29] (index 0 to 0) and [10] (index 1 to 1).



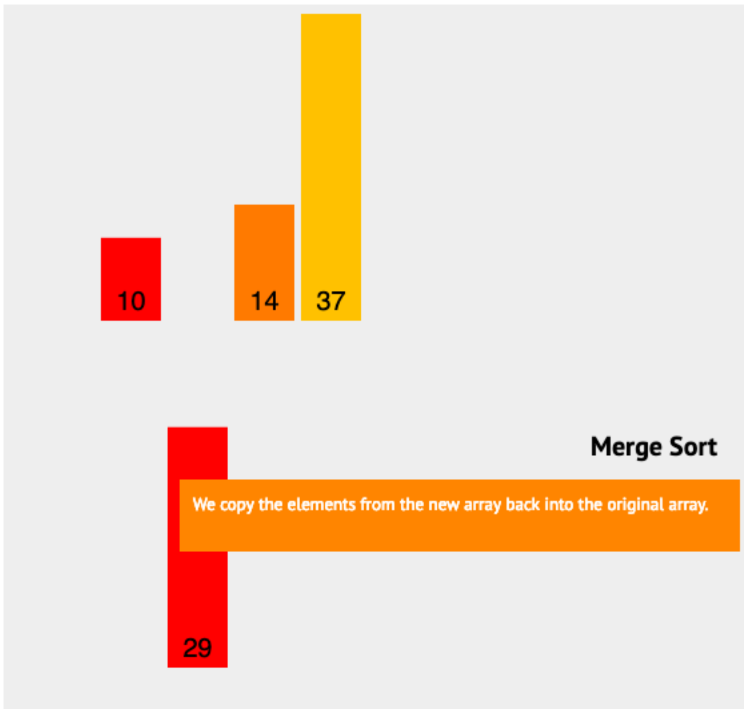
Merge Sort

Since 29 (left partition) > 10 (right partition), we copy {rightPart} into new array.wierd



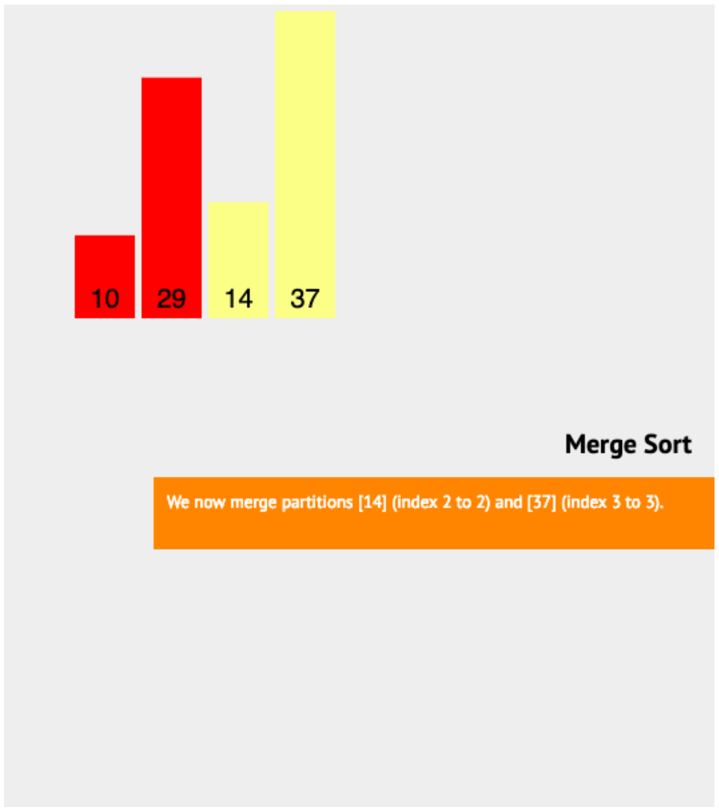
Merge Sort

Since right partition is empty, we copy 29 (left partition) into new array.



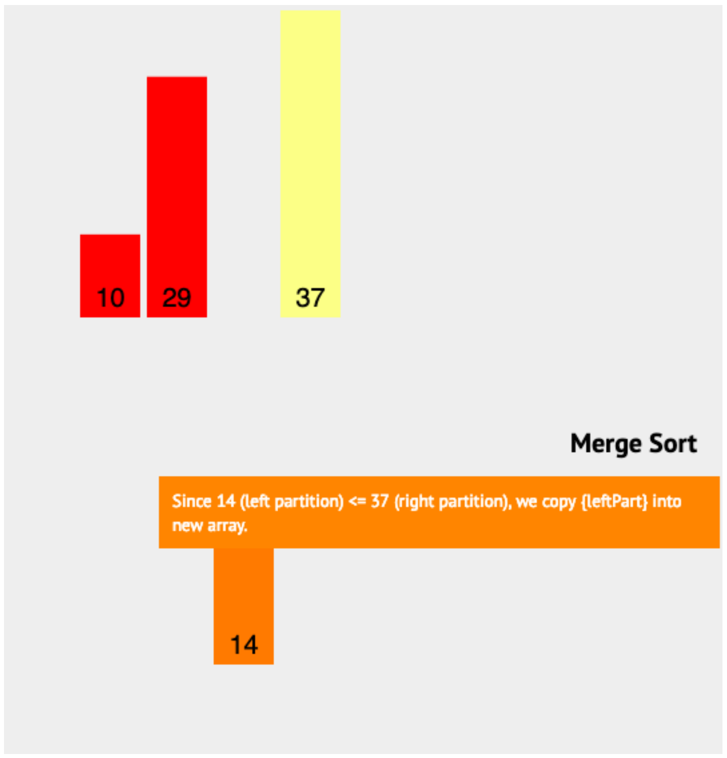
Merge Sort

We copy the elements from the new array back into the original array.



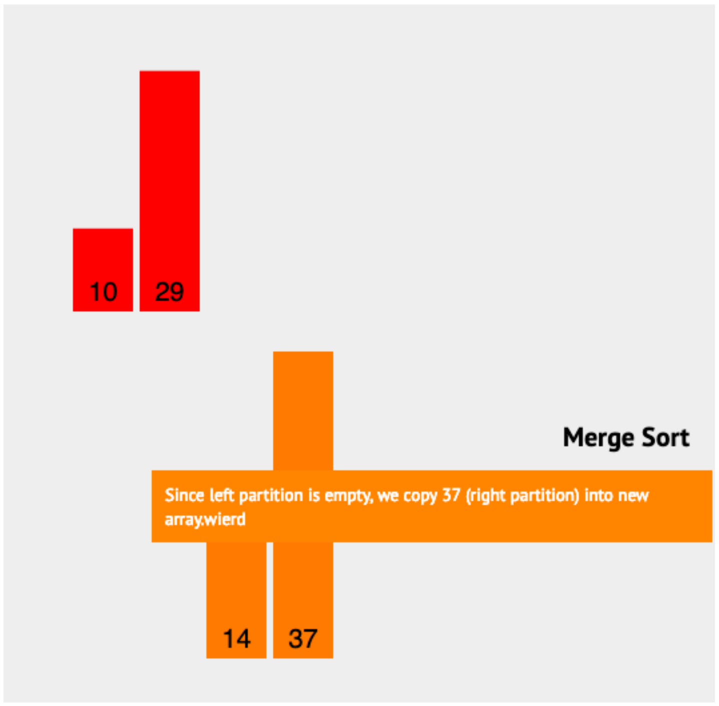
Merge Sort

We now merge partitions [14] (index 2 to 2) and [37] (index 3 to 3).



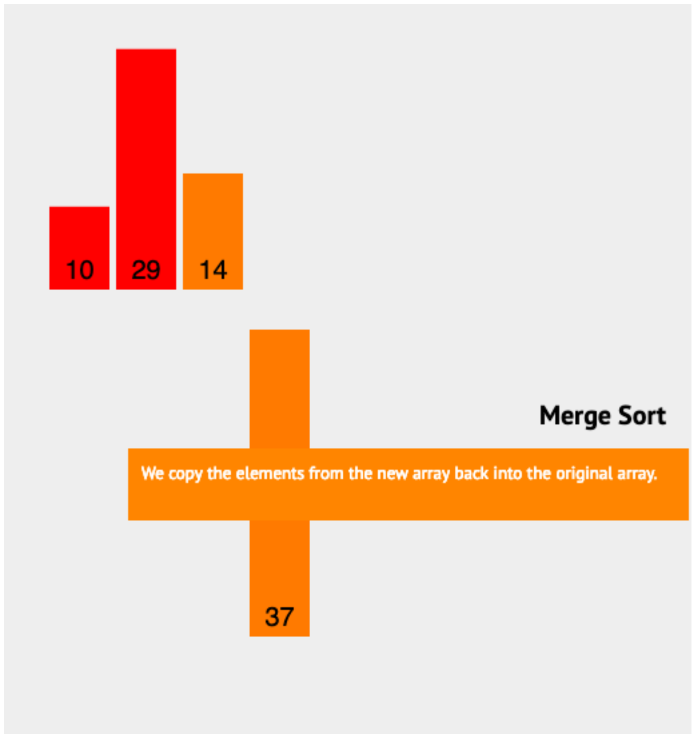
Merge Sort

Since 14 (left partition) <= 37 (right partition), we copy {leftPart} into new array.



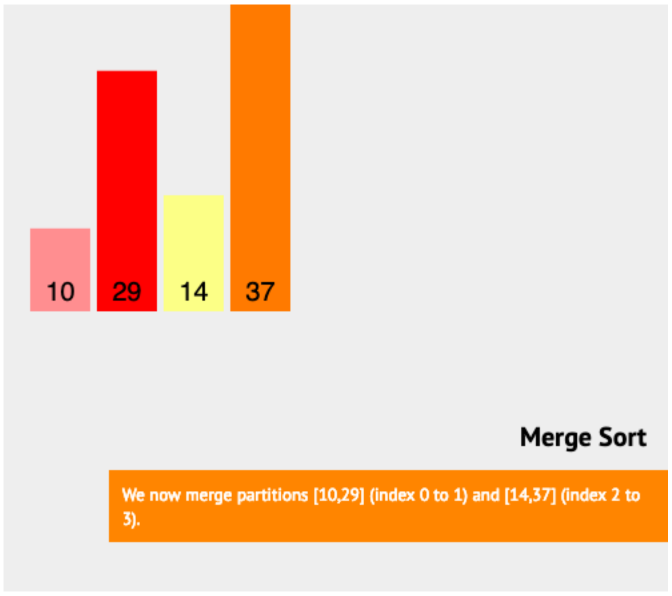
Merge Sort

Since left partition is empty, we copy 37 (right partition) into new array.wierd



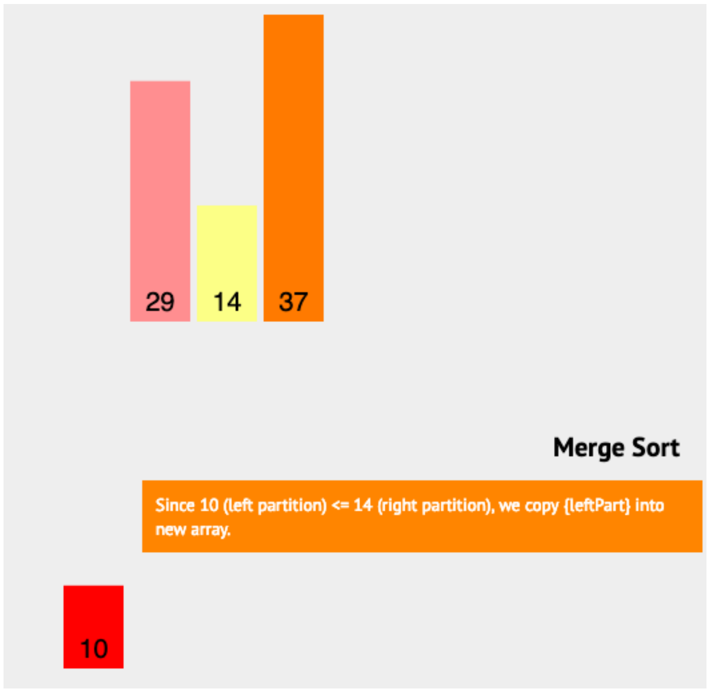
Merge Sort

We copy the elements from the new array back into the original array.



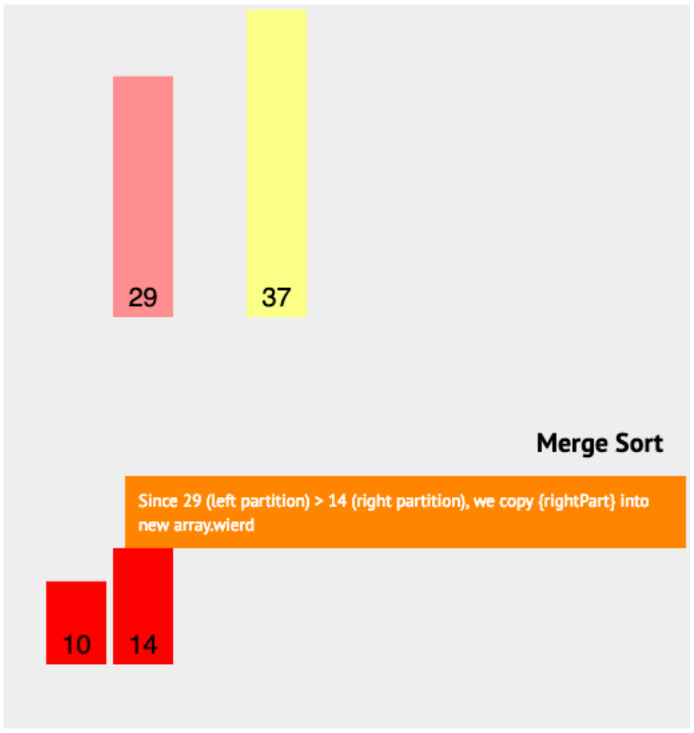
Merge Sort

We now merge partitions [10,29] (index 0 to 1) and [14,37] (index 2 to 3).



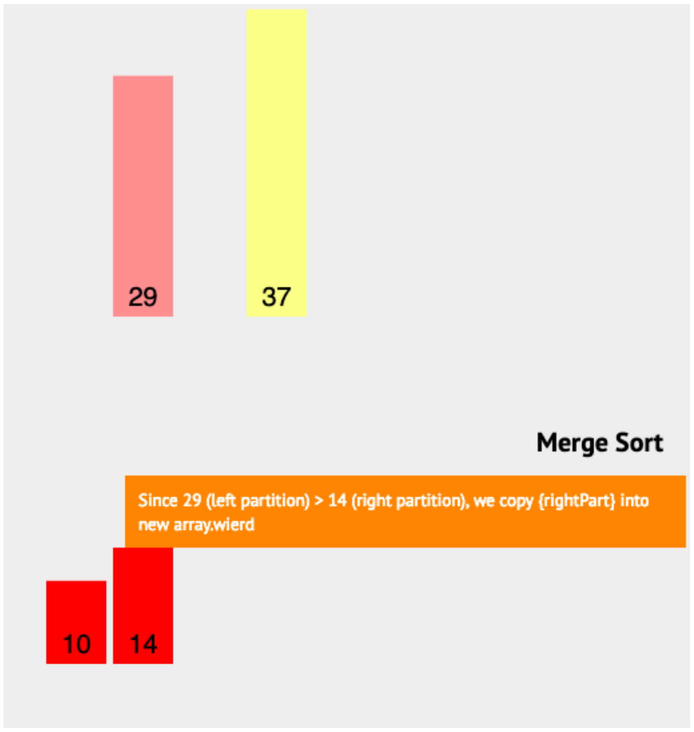
Merge Sort

Since 10 (left partition) <= 14 (right partition), we copy {leftPart} into new array.



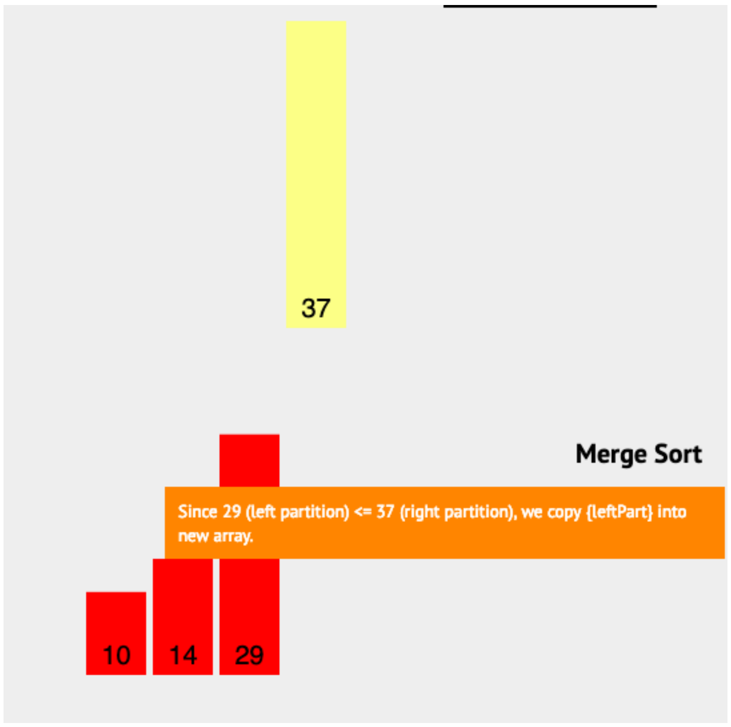
Merge Sort

Since 29 (left partition) > 14 (right partition), we copy {rightPart} into new array.wierd



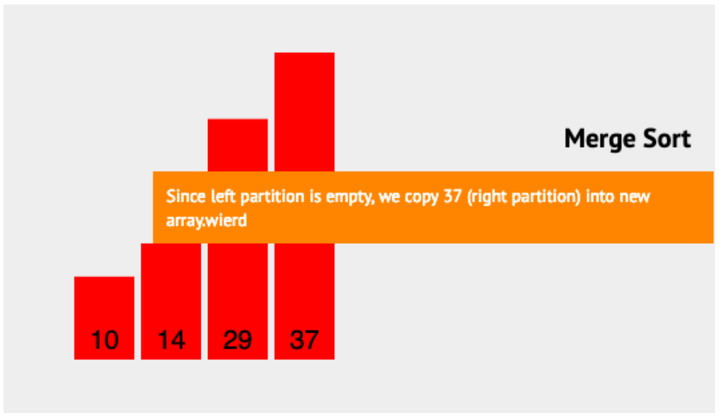
Merge Sort

Since 29 (left partition) > 14 (right partition), we copy {rightPart} into new array.wierd



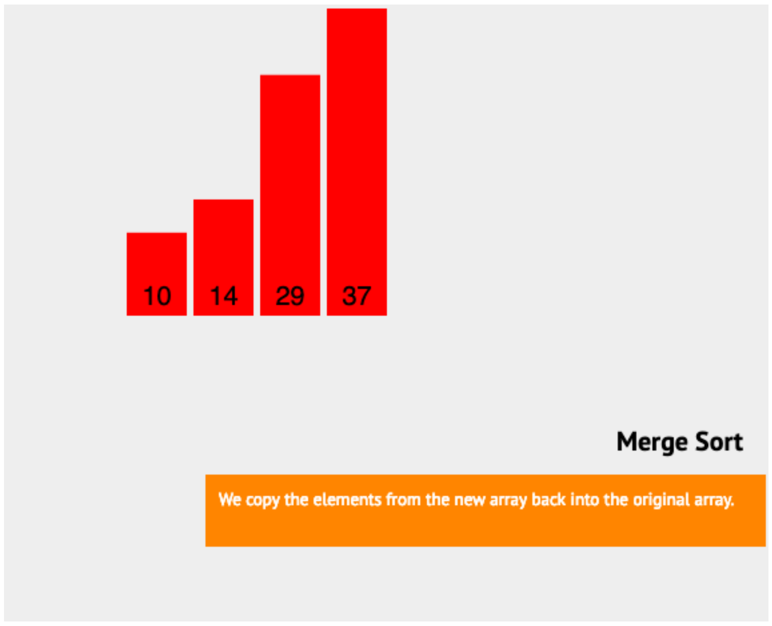
Merge Sort

Since 29 (left partition) <= 37 (right partition), we copy {leftPart} into new array.



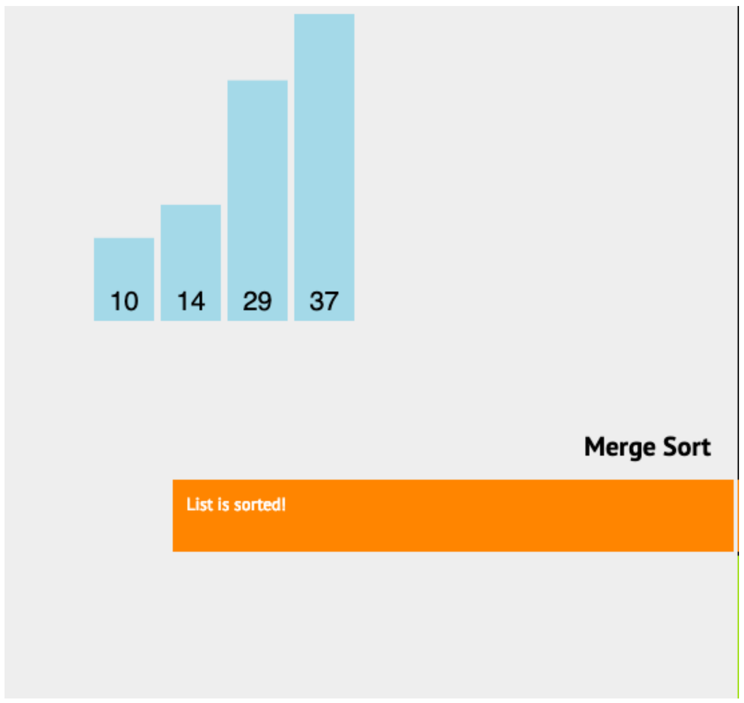
Merge Sort

Since left partition is empty, we copy 37 (right partition) into new array.wierd



Merge Sort

We copy the elements from the new array back into the original array.



List is sorted!