Merge sort

In today's data-driven world, the ability to organize and process vast amounts of information has become increasingly crucial.

Whether it's managing a large database, analyzing customer data, or simply organizing a music playlist, efficient sorting methods play a vital role in our lives. One of the most challenging aspects of handling enormous datasets is sorting them in an organized and timely manner, which calls for an efficient algorithm.

The problem we face is sorting a large collection of data elements in a way that minimizes the time and computational resources required.

Traditional sorting methods, such as bubble sort or insertion sort, may not be practical for massive datasets due to their time complexity. This is where a more advanced and efficient sorting algorithm is needed.

Description of the algorithm

Merge sort is a divide-and-conquer algorithm that recursively breaks down the input data into smaller subsets, sorts them, and then merges the sorted subsets to produce the final sorted output.
Imagine you have a deck of cards that you want to sort in ascending order by their face value. To do this as efficiently as possible, let's take advantage of the main features of the **merge sort** algorithm.

1. First, what the algorithm would do is to divide the entire deck into smaller parts until each part contains only one card. We'll omit this part to show the process of the merge sort itself.
2. Next, the algorithm would start merging these smaller parts. To do this, we combine every two sorted parts of the deck into one, also sorted. This process is called **merging**. This is done in what seems to be the most obvious way: we choose a minimum of two hands, and now this is the first element of the new, merged set of cards. Repeat the procedure for all other cards remaining in your hand.

- Next, we continue to sort and merge pairs of adjacent parts until we run out of them.
- At the end of this process, we have a fully sorted deck of cards in ascending order.

Implementations

The algorithm can be implemented in many ways. Here's a short description of the most popular ones:

- **top-down** is a recursive implementation that recursively divides the given array into two subarrays until there is only a single-element array remaining; it then merges the results together to produce a sorted subarray of a larger size.

- **bottom-up** is an iterative implementation that first merges pairs of adjacent single-element arrays and produces sorted subarrays of 2 elements, then merges pairs of adjacent arrays of 2 elements producing 4-elements sorted subarrays, then merges pairs of 4 elements, and so on until the whole array is merged (sorted).
- **in-place**: The standard merge sort algorithm requires extra memory to store the sorted subarrays during the merge phase. In-place merge sort eliminates the need for extra memory by using an alternative.
- **hybrid**: This algorithm uses a combination of insertion sort and merge sort. When the sub-array size becomes small enough, the algorithm switches to insertion sort which is faster for small inputs. This can improve the performance of merge sort for small arrays.

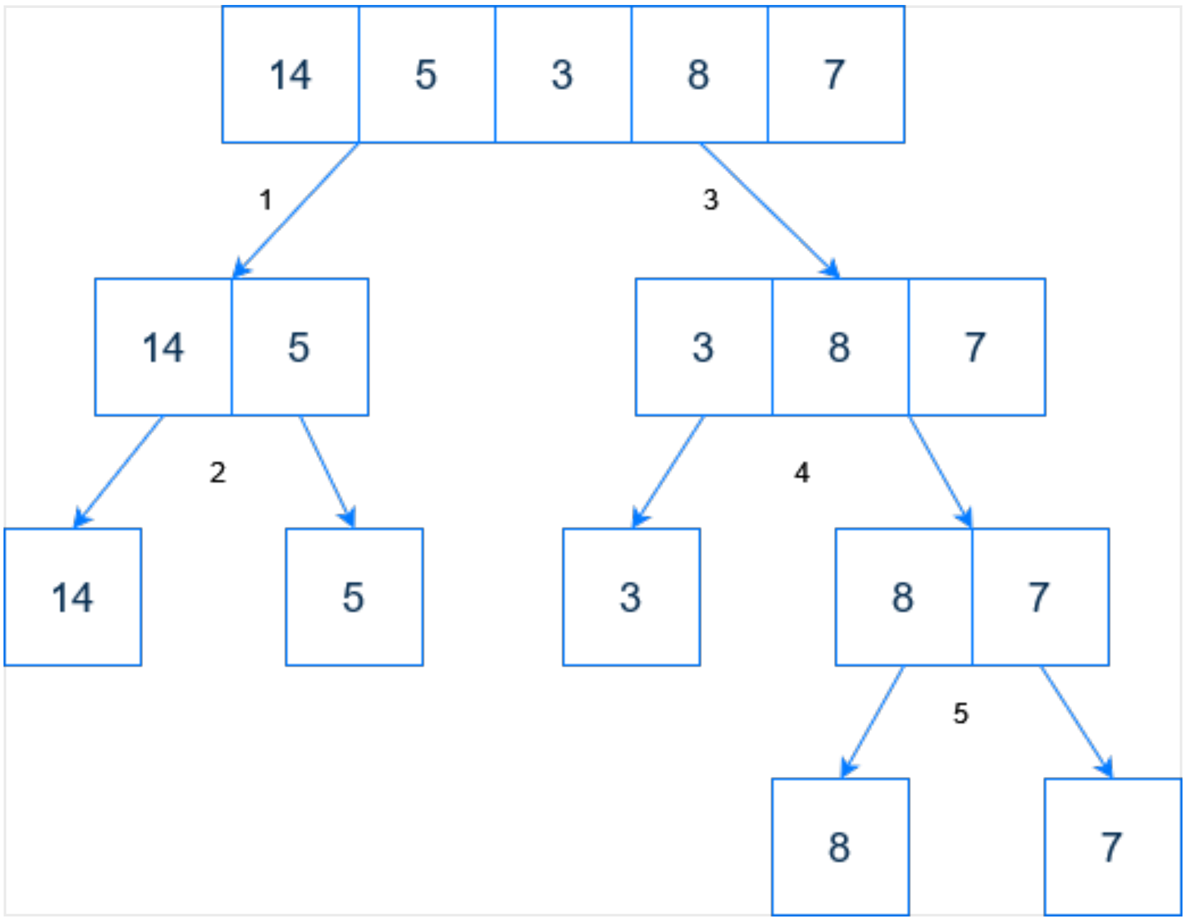
Top-down **merge sort** is generally considered more intuitive and easier to grasp, so let's take a look at this closer in terms of pseudocode:

```
MergeSort(arr[], left, right):          // Send the left and right boundaries of the subarray to the function
    if left < right                      // Check if the left index is less than the right index
        middle = (left + right) / 2      // Calculate the middle index of the subarray
        MergeSort(arr, left, middle - 1)    // Recursively sort the left half of the subarray
        MergeSort(arr, middle, right) // Recursively sort the right half of the subarray
        Merge(arr, left, middle, right) // Merge the two sorted halves of the subarray

Merge(arr[], left, middle, right):      // Define Merge function, taking an array and its left, middle, and right boundaries
    len1 = (middle - 1) – left + 1           // Calculate the lengths of the left and right subarrays
    len2 = right – middle + 1
    leftArr[len1]                        // Create temporary arrays for the left and right subarrays
    rightArr[len2]

    for i in [0, len1 – 1]:              // Copy the elements of the left subarray into the temporary left array
        leftArr[i] = arr[left + i]
    for j in [0, len2 – 1]:              // Copy the elements of the right subarray into the temporary right array
        rightArr[j] = arr[middle + j]

    i = 0                                // Initialize indices for the temporary arrays and the
```

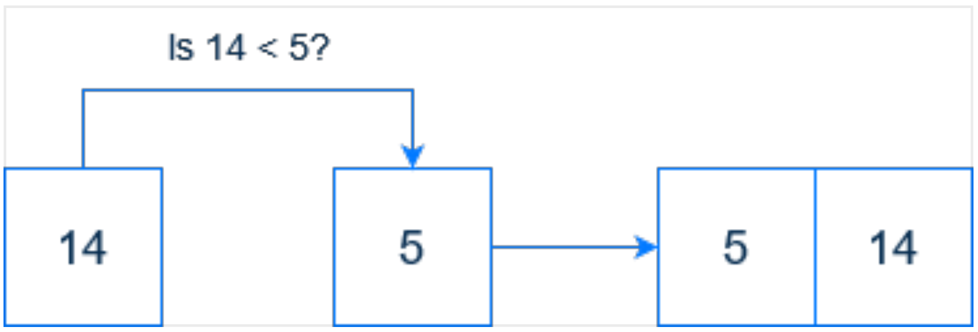
```
main array
    j = 0
    k = left

    while i < len1 and j < len2:        // Merge the temporary arrays back into the main array
        if leftArr[i] <= rightArr[j]:   // Compare elements from the left and right arrays and add the smaller one to the main array
            arr[k] = leftArr[i]
            i = i + 1
        else:
            arr[k] = rightArr[j]
            j = j + 1
        k = k + 1                       // Increment the main array index

    while i < len1:                     // Copy any remaining elements from the left temporary array to the main array
        arr[k] = leftArr[i]
        i = i + 1
        k = k + 1

    while j < len2:                     // Copy any remaining elements from the right temporary array to the main array
        arr[k] = rightArr[j]
        j = j + 1
        k = k + 1
```

Example

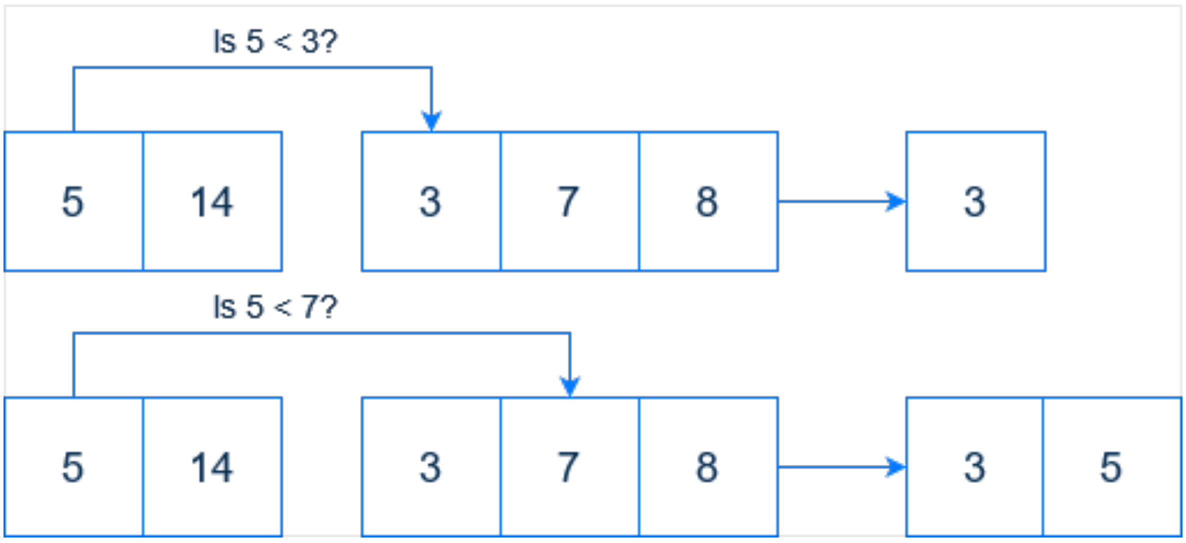
Suppose we have an unsorted five-element array of integers. It takes 5 operations to split the array into single-size subarrays.



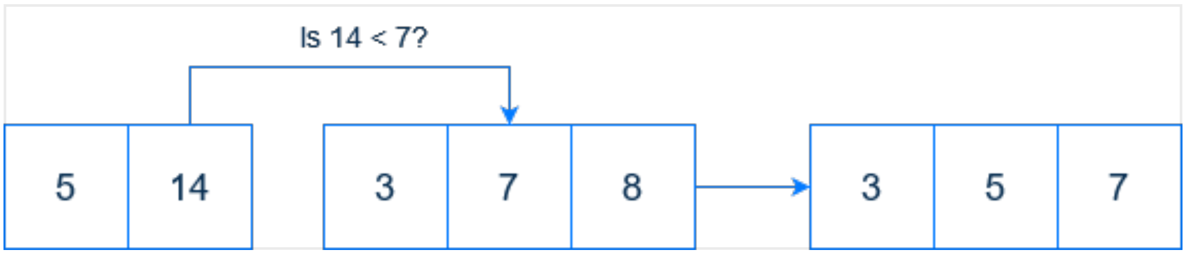
Now let's look at the most interesting part. How does merging and sorting work?



In this example, we'll sort the array in ascending order. Here we take the first element of the first array and compare it with the first element of the second array. Which is smaller goes first to the new array.
The same happens with the second half of the array. Omit the merging of single-element arrays and go straight to the final stage.



The algorithm uses pointers and constantly shifts them in both arrays depending on the result of the comparison. So at the current step, you can see that 5 is fewer than 7. Therefore, the current element adds to the end of the result array, then the pointer of the first array shifts to the next element.



As you may have guessed, the same principle adds 8. Now we have only 14 left in the first array. The final part of the algorithm checks if anything is left in both arrays and voilà, at the output we have a sorted array.

Algorithm properties

Consider the following properties of Merge sort:

- The time complexity is $O(n \cdot \log n)$
  , which makes it highly efficient for large arrays of data. The complexity arises from the combination of
  $\log n$
  levels of divisions and the linear
  $O(n)$
  time required to merge subarrays at each level. This makes merge

sort an efficient and widely used sorting algorithm.
- It is a stable sorting algorithm, which means that it maintains the relative order of equal elements in the sorted array. This property is essential in some applications where preserving the order of equal elements is critical.
- Merge sort is not an in-place sorting algorithm in the original top-down implementation. It means that it requires additional memory to store the sorted array. This property makes it less suitable for sorting arrays with limited memory or requiring constant updates. We'll also consider in-place implementation by example.

In-place implementation example

We are already familiar with the top-down implementation, which requires additional memory. What if you need to save up some memory? This is when the in-place sort comes in handy. Let's take a look at how it works.

The in-place merge sort algorithm works by maintaining two pointers that traverse the two sorted sub-arrays, and a third pointer that indicates the current position in the output array. At each step, the algorithm compares the elements pointed to by the two sub-array pointers and swaps them if needed to maintain the correct order. This process continues until one of the pointers reaches the end of its sub-array. Once that happens, the remaining elements in the other sub-array are already in their correct positions, and no further action is required. This approach eliminates the need for additional memory to store sorted sub-arrays, making it more memory-efficient.

Note that in each step, the merge sort algorithm works recursively, dividing and merging the parts until the entire deck is sorted. This is different from the insertion sort algorithm, which sorts the elements by repeatedly inserting them into the correct position in the sorted part of the deck.

```
Merge(arr[], left, middle, right):
    i = left                        // Initialize the index for the left sub-array
    j = middle + 1                  // Initialize the index for the right sub-array

    while i <= middle and j <= right: // Iterate until we reach the end of either sub-array
        if arr[i] <= arr[j]:         // Compare elements at the current indices of both sub-arrays
            i = i + 1                // If the left element is smaller or equal, it is already in the correct position
        else:
            temp = arr[j]            // If the right element is smaller, store it in a temporary variable
            k = j                // Initialize a new index for shifting elements

            while k > i:         // Shift elements in the left sub-array to the right by one position
                arr[k] = arr[k – 1]
                k = k – 1
            arr[i] = temp        // Place the right element in its correct position in the merged array
            i = i + 1            // Update the indices to continue the comparison and merging process
            middle = middle + 1
            j = j + 1
```