

Merge algorithm



Manal alsallal Alqadri

What is the merge algorithm?

The merge algorithm, also known as merge sort, is a popular and efficient sorting algorithm that follows the divide-and-conquer paradigm. It works by recursively dividing the input list into smaller sub lists, sorting them individually, and then merging them back together to create a sorted list.

How the merge algorithm works:

1. \*\*Divide\*\*: The input list is divided into two roughly equal-sized sub lists repeatedly until each sub list contains at most one element. This recursive division is performed until the base case is reached.

2. \*\*Conquer\*\*: Once the sub lists reach the base case (i.e., they contain at most one element), they are considered sorted by definition. If a sub list has more than one element, the merge algorithm is applied recursively to sort it.

3. \*\*Merge\*\*: The sorted sub lists are then merged back together by comparing the elements from the two sub lists and placing them in the correct order in a new merged list. This merging process continues until all the elements from both sub lists have been merged into a single sorted list.

The key step in the merge algorithm is the merging process. When merging two sub lists, the algorithm compares the first elements of each sub list and selects the smaller (or larger, depending on the sorting order) element to be placed in the merged list. The process continues recursively until all elements from both sub lists are merged.

The time complexity of the merge algorithm is O(n log n), where n is the number of elements in the input list. This makes merge sort efficient for large datasets. Additionally, merge sort is a stable sorting algorithm, meaning it preserves the relative order of elements with equal values.

Overall, merge sort's divide-and-conquer approach and efficient time complexity make it a popular choice for sorting tasks. However, it does require additional memory space to store the sub lists during the merging process, which can be a consideration for large lists with limited memory resources.

An example of how the merge algorithm (merge sort) works step by step. Consider the following unsorted list:

```

[5, 2, 8, 3, 1, 6, 4]

```

Step 1: Divide the list

The algorithm starts by recursively dividing the list into smaller sub lists until each sub list contains at most one element.

This process is repeated until the base case is reached.

```

[5, 2, 8, 3, 1, 6, 4]

```

Dividing the list in half gives us:

```

[5, 2, 8] [3, 1, 6, 4]

```

Further division gives us:

```

[5] [2, 8] [3, 1] [6, 4]

```

```

[5] [2] [8] [3] [1] [6] [4]

```

Step 2:

Conquer (Sort the sub lists)

Now, the algorithm sorts the individual sub lists. Since each sub list contains only one element, they are considered sorted.

```

[5] [2] [8] [3] [1] [6] [4]

```

Step 3: Merge the sub lists

The sorted sub lists are then merged back together by comparing the elements and placing them in the correct order. The merging process continues until all elements from both sub lists are merged into a single sorted list.

Starting with the smallest sub lists, we merge and sort them:

```

[2, 5] [3, 8] [1, 4, 6]

```

Continuing the merging process:

```

[2, 3, 5, 8] [1, 4, 6]

```

Finally, merging the last two sub lists:

```

[1, 2, 3, 4, 5, 6, 8]

```

The resulting sorted list is

```

[1, 2, 3, 4, 5, 6, 8]

```

The merge algorithm (merge sort) divides the list, sorts the sub lists, and merges them back together in a sorted manner. This process guarantees that the final merged list is sorted correctly. This example demonstrates the basic steps of the merge algorithm, showcasing its divide-and-conquer approach to sorting.

A few additional points to expand on the merge algorithm (merge sort):

1. \*\*Stability\*\*: Merge sort is a stable sorting algorithm, meaning that it preserves the relative order of elements with equal values. During the merging process, if two elements are considered equal, the algorithm will place the element from the first sub list before the element from the second sub list. This property is useful when sorting objects with multiple keys or when preserving the original order of equal elements is important.

2. \*\*Time Complexity\*\*: The time complexity of merge sort is O(n log n), where n is the number of elements in the input list. This makes merge sort efficient for large datasets. The algorithm consistently divides the input list into halves until reaching sub lists with at most one element, and then merges them back together. The logarithmic factor arises from the number of times the list can be divided in half during the recursive process.

3. \*\*Space Complexity\*\*: Merge sort requires additional memory space to store the sub lists during the merging process.

Specifically, it requires an auxiliary array of the same size as the input list. This can be a consideration for large lists with limited memory resources. However, the space complexity of merge sort is generally considered to be O(n), which is still efficient in most cases.

4. \*\*Adaptability\*\*: Merge sort performs consistently well regardless of the initial order of the elements. Unlike some other sorting algorithms whose performance degrades significantly when presented with partially sorted or nearly sorted lists, merge sort maintains its time complexity of O(n log n) under all scenarios.

5. \*\*Parallelism\*\*: Merge sort can be easily parallelized, as the merging process is amenable to concurrent execution. By dividing the input list into smaller sub lists

and merging them independently, merge sort can take advantage of parallel processing capabilities, potentially leading to improved performance on systems with multiple processors or cores.

Merge sort's efficient time complexity, stability, adaptability, and potential for parallelism make it a popular choice for sorting large datasets or situations where stability and performance are important considerations. However, it's worth noting that merge sort does have a relatively higher constant factor compared to some other sorting algorithms, which can impact its practical performance for small lists or in situations where memory usage is a critical concern.

# Advantages:

* It is quicker for larger lists because unlike insertion and bubble sort it doesn't go through the whole list several times.
* It has a consistent running time, carries out different bits with similar times in a stage.

It can be applied to files of any size.

Reading of the input during the run-creation step is sequential ==> Not much seeking.

Merge Sort requires more space than other sort.

If heap sort is used for the in-memory part of the merge, its operation can be overlapped with I/O

# Disadvantages

* Slower comparative to the other sort algorithms for smaller tasks.
* goes through the whole process even i he list is sorted (just like insertion and bubble sort?)
* uses more memory space to store the sub elements of the initial split list.

Requires extra space »N

Merge Sort requires more space than other sort.

Merge sort is less efficient than other sort.