**The divide-and-conquer approach**

Many useful algorithms are ***recursive*** in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems.

These algorithms typically follow a ***divide-and-conquer*** approach: they break the problem into several sub problems that are similar to the original problem but smaller in size, solve the sub problems recursively, and then combine these solutions to create a solution to the original problem.

The divide-and-conquer paradigm involves three steps at each level of the recursion:

**Divide** the problem into a number of sub problems.

**Conquer** the sub problems by solving them recursively. If the sub problem sizes are small enough, however, just solve the sub problems in a straightforward manner.

**Combine** the solutions to the sub problems into the solution for the original problem.

The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it

operates as follows.

**Divide:** Divide the *n*-element sequence to be sorted into two subsequences of *n*/2 elements each.

**Conquer:** Sort the two subsequences recursively using merge sort.

**Combine:** Merge the two sorted subsequences to produce the sorted answer.

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there

is no work to be done, since every sequence of length 1 is already in sorted order.

 **Divide:**  
Recursively split the array into halves until each subarray has just **one element** — this is straightforward.

 **Conquer:**  
This step refers to **recursively solving the smaller subproblems** — which, in merge sort, means sorting the smaller subarrays.

* If the subarrays have only **one element**, they are already sorted.
* When the recursion starts returning, you have to **merge the sorted subarrays**.

 **Combine:**  
This step is where the real work happens — **merging the sorted subarrays** into larger sorted arrays until the full array is merged and sorted.

* The merge process involves comparing elements from two sorted subarrays and placing them in the correct order in a new array.
* This step combines the sorted halves into a larger sorted sequence.

Merge Sort is a [Divide and Conquer](https://www.geeksforgeeks.org/divide-and-conquer-introduction/) algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

**MergeSort(arr[], l, r)**

If r > l

**1.** Find the middle point to divide the array into two halves:

middle m = (l+r)/2

**2.** Call mergeSort for first half:

Call mergeSort(arr, l, m)

**3.** Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

**4.** Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)

The following diagram shows the complete merge sort process for an example array {2, 4, 5, 7, 1, 2, 3,6}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged. The numbers shown with arrows indicate the order in which the steps are performed.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 5 | 4 | 2 | 7 | 6 | 3 | 2 | 1 |

1

|  |  |  |  |
| --- | --- | --- | --- |
| 6 | 3 | 2 | 1 |

|  |  |  |  |
| --- | --- | --- | --- |
| 5 | 4 | 2 | 7 |

Divide

8

2

|  |  |
| --- | --- |
| 2 | 7 |

|  |  |
| --- | --- |
| 6 | 3 |
|  |  |

|  |  |
| --- | --- |
| 2 | 1 |

|  |  |
| --- | --- |
| 5 | 4 |

11

9

5

3

|  |
| --- |
| 4 |

|  |
| --- |
| 1 |

|  |
| --- |
| 5 |

|  |
| --- |
| 2 |

|  |
| --- |
| 7 |

|  |
| --- |
| 6 |

|  |
| --- |
| 3 |

|  |
| --- |
| 2 |

12

Conquer

10

6

4

Merging Starts here

|  |  |
| --- | --- |
| 2 | 7 |

|  |  |
| --- | --- |
| 4 | 5 |

|  |  |
| --- | --- |
| 3 | 6 |

|  |  |
| --- | --- |
| 1 | 2 |

13

7

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 4 | 5 | 7 |

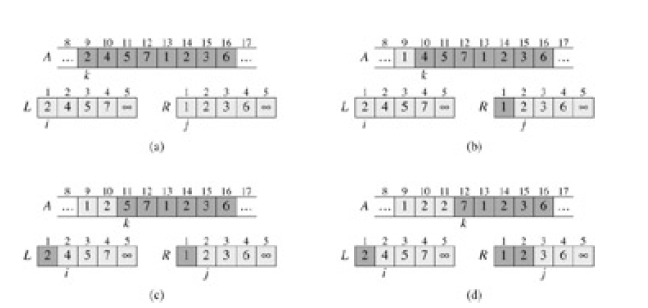
|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 6 |

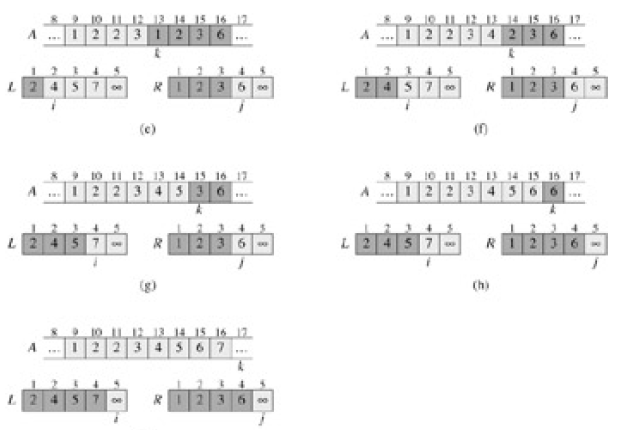
merge

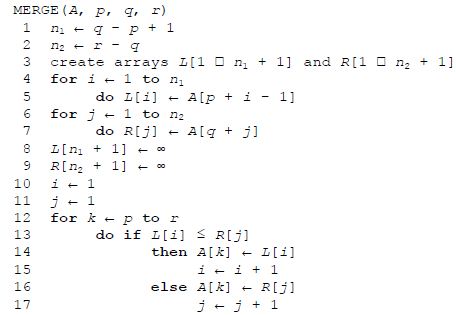
14

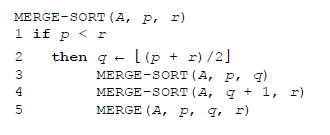
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 2 | 3 | 4 | 5 | 6 | 7 |

**Procedure of Merging two sub-arrays**









**Tracing merge sort:**

**Step 1: Initial call**

We start with:

**MERGE-SORT(A, 1, 8)**

* **Divide:** q = 4
* **Left half:** MERGE-SORT(A, 1, 4)
* **Right half:** MERGE-SORT(A, 5, 8)

The **left recursive call happens first**. Control moves **down the recursion tree** until reaching the base case.

**Step 2: Left subtree (sorting [5, 4, 2, 7])**

* **Left half:** MERGE-SORT(A, 1, 2)
* **Right half:** MERGE-SORT(A, 3, 4)

**Breaking it down:**

1. MERGE-SORT(A, 1, 2)
   * Further splits into **[5]** and **[4]**
   * Base case reached — control now returns **up** to merge:
   * **MERGE(A, 1, 1, 2)** → **[4, 5]**
2. Next, **right half** of the left subtree:
   * MERGE-SORT(A, 3, 4)
   * Further splits into **[2]** and **[7]**
   * Base case reached — control returns **up** to merge:
   * **MERGE(A, 3, 3, 4)** → **[2, 7]**

After both halves are sorted:

* **MERGE(A, 1, 2, 4)** → **[2, 4, 5, 7]**

**Control now returns to the first call** (MERGE-SORT(A, 1, 8)), ready to sort the **right half**.

**Step 3: Right subtree (sorting [6, 3, 2, 1])**

1. **Left half:** MERGE-SORT(A, 5, 6)
   * Splits into **[6]** and **[3]**
   * Base case reached — control returns to merge:
   * **MERGE(A, 5, 5, 6)** → **[3, 6]**
2. **Right half:** MERGE-SORT(A, 7, 8)
   * Splits into **[2]** and **[1]**
   * Base case reached — control returns to merge:
   * **MERGE(A, 7, 7, 8)** → **[1, 2]**

After both halves are sorted:

* **MERGE(A, 5, 6, 8)** → **[1, 2, 3, 6]**

**Control now returns again** to the top-level call (MERGE-SORT(A, 1, 8)).

**Step 4: Final merge (combining both halves)**

* **Left sorted array:** [2, 4, 5, 7]
* **Right sorted array:** [1, 2, 3, 6]
* Final merge:

**MERGE(A, 1, 4, 8)**

**Result: [1, 2, 2, 3, 4, 5, 6, 7]**

**Control flow summary:**

**The recursive calls push onto the call stack as they go deeper into dividing the array.  
Then, once the base case is hit (subarrays of size 1), control "pops back up":**

* **First merging small arrays (like [5] and [4])**
* **Then combining larger sorted subarrays step by step**
* **Finally returning control to the initial call**