**Merge Sort**

* Merge Sort is a **divide-and-conquer** sorting algorithm.
* Merge sort is a recursive sorting algorithm that always gives the same performance (same for best, average, and worst-case time complexity), regardless of the initial order of the array items.
* Suppose you need to sort an array. As seen in Figure 11-5, you would
  + divide the array into halves,
  + sort each half,
  + and then
  + merge the sorted halves into one sorted array

Diagram

Description automatically generated

* In the figure, the halves <1, 4, 8> and <2, 3> are merged to form the array <1, 2, 3, 4, 8>.
* This merge step compares an item in one half of the array with an item in the other half and moves the smaller item to a temporary array.
* This process continues until there are no more items to consider in one half.
* At that time, you simply move the remaining items to the temporary array.
* Finally, you copy the temporary array back into the original array.
* Although the merge step of the merge sort produces a sorted array, how do you sort the array halves prior to the merge step?
* The merge sort sorts the array halves by using a merge sort—that is, by calling itself recursively.

**Pseudocode**

mergeSort(arr[], first, last)

If first < last

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

middle mid = first + (last - first) / 2

2. Call mergeSort for first half:

Call mergeSort(arr, first, mid)

3. Call mergeSort for second half:

Call mergeSort(arr, mid+1, last)

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

Call merge(arr, first, mid, last)

* The merge(arr, l, m, r) is a key process that assumes that arr[l..m] (first half) and arr[m+1..r] (second half) are sorted and merges the two sorted sub-arrays into one.
* Clearly, most of the effort in the merge sort algorithm is in the merge step, but does this algorithm actually sort?
* The recursive calls continue dividing the array into pieces until each piece contains only one item; obviously an array of one item is sorted.
* The algorithm then merges these small pieces into larger sorted pieces until one sorted array results.
* Figure 11-6 illustrates both the recursive calls and the merge steps in a merge sort of an array of six integers.

Diagram

Description automatically generated

**Binary Tree**

* Merge sort is synonymous to a binary tree in that it splits two ways recursively.
* The following diagram from [wikipedia](http://en.wikipedia.org/wiki/File:Merge_sort_algorithm_diagram.svg) shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}.
* If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1.
* Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



**Algorithm Analysis**

**merge()**

* The merge() step of the algorithm requires the most effort, let’s begin the analysis there.
* Each merge() call merges
  + arr[first..mid]

and

* + arr[mid+1..last].
* If the total number of items in the two array segments to be merged is *n*, then merging the segments requires at most *n –* 1 comparisons.
  + Figure 11-7 provides an example of a merge step that has 6 total items to be merged, and therefore requires 5 comparisons

A picture containing text

Description automatically generated

* In addition, there are *n* moves from the original array to the temporary array, and *n* moves from the temporary array back to the original array.
* Thus, each merge step requires 3 × *n* – 1 major operations.

**mergeSort()**

* Each call to mergeSort recursively calls itself twice.
* Therefore, we can compare the algorithm’s calls on the array to a binary tree.

As Figure 11-8 illustrates:

* + The original call (from client or calling module) to mergeSort() is at level 0
  + Two calls to mergeSort() occur at level 1 of the recursion.
  + Each of these calls then calls mergeSort twice, so four calls to mergeSort occur at level 2 of the recursion, and so on.
  + How many levels of recursion are there? We can count them, as follows.
* Each call to mergeSort() halves the array.
* Halving the array the first time produces two pieces.
* The next recursive calls to mergeSort() halve each of these two pieces to produce four pieces of the original array; the next recursive calls halve each of these four pieces to produce eight pieces, and so on.
* The recursive calls continue until the array pieces each contain one item—that is, until there are *n* pieces, where *n* is the number of items in the original array.
* If *n* is a power of 2 (*n* = 2*k*), the recursion goes *k* = log2*n* levels deep.
* For example, in Figure 11-8, there are three levels of recursive calls to mergeSort() because the original array contains eight items and 8 = 23.
* If *n* is not a power of 2, there are 1 + log2*n* (rounded down) levels of recursive calls to mergeSort().
* The original call to mergeSort() (at level 0) calls merge once.
* Then merge merges all *n* items and requires 3 × *n* – 1 operations, as was shown earlier.
* At level 1 of the recursion (bottom level), two calls to mergeSort(), and hence to merge, occur.
* Each of these two calls to merge() merges *n* / 2 items and requires 3 × (*n* / 2) – 1 operations.
* Together these two calls to merge require 2 × (3 × (*n* / 2) – 1) or 3 × *n* – 2 operations.
* At level *m* of the recursion, 2*m* calls to merge occur; each of these calls merges *n* / 2*m* items and so requires 3 × (*n* / 2*m*) – 1 operations.
* Together the 2*m* calls to merge require 3 × *n* – 2*m* operations. Thus, each level of the recursion requires O(*n*) operations.
* Because there are either log2*n* or 1 + log2*n* levels, the merge sort is O(*n* × log *n*) in both the worst and average cases.
* Although the merge sort is an extremely efficient algorithm with respect to time, it does have one drawback:
* The merge step requires an auxiliary array, meaning O(N) space.
* This extra storage and the necessary copying of entries are disadvantages.

Note that calls and merges at each level vary. This is best illustrated in the following diagram:



Though a level may have call mergeSort() twice, the recursive will follow calls like a binary tree