11.13 Merge Sort

- An efficient sorting algorithm
    - Conceptually more complex than selection sort and insertion sort
- Sorts an array by splitting it into two equal-sized subarrays, sorting each subarray, then merging them into one larger array
    - With an odd number of elements, the algorithm creates the two subarrays such that one has one more element than the other

11.13 Merge Sort (cont.)

- Implementation of merge sort in this example is recursive
- Base case is an array with one element, which is, of course, sorted
- Recursion step splits the array into two approximately equal pieces, recursively sorts them, then merges the two sorted arrays into one larger, sorted array

11.13 Merge Sort (cont.)

- Suppose the algorithm has already merged smaller arrays to create sorted arrays

```
array1 : 14 20 34 56 77
array2 : 15 30 51 52 93
```

- Merge sort combines these two arrays into one larger, sorted array
- Smallest element in `array1` is `14` (located in index 0 of `array1`)
- Smallest element in `array2` is 15 (located in index 0 of `array2`)
- To determine the smallest element in the larger array, the algorithm compares `14` and `15`
- The value from `array1` is smaller, so `14` becomes the first element in the merged array
- The algorithm continues by comparing `20` (the second element in `array1`) to `15` (the first element in `array2`)
- The value from `array2` is smaller, so `15` becomes the second element in the larger array
- The algorithm continues by comparing `20` to `30` , with `20` becoming the third element in the array, and so on.

11.13.1 Merge Sort Implementation

- `mergesort.py` defines:
  - Function `merge_sort` to initiate the sorting
  - Function `sort_array` to implement the recursive merge sort algorithm—this is called by function `merge_sort`
  - Function `merge` to merge two sorted subarrays into a single sorted subarray
  - Function `subarray_string` to get a subarray's string representation for output purposes to help visualize the sort
  - Function `main` tests function `merge_sort`
- It's well worth your time to step through this program's outputs to fully understand this elegant and fast sorting algorithm

Function `merge_sort`

- Calls function `sort_array` to initiate the recursive algorithm, passing `0` and `len(data) - 1` as the low and high indices of the array to be sorted
  - These values tell function `sort_array` to operate on the entire array

In [1]:

```python
# mergesort.py
"""Sorting an array with merge sort."""
import numpy as np

# calls recursive sort_array method to begin merge sorting
def merge_sort(data):
    sort_array(data, 0, len(data) - 1)
```

Recursive Function `sort_array`

- Performs the recursive merge sort algorithm
- If the size of the array is 1, the array is already sorted, so the function returns immediately
- If the size of the array is greater than 1, the function splits the array in two, recursively calls function `sort_array` to sort the two subarrays, then merges them.
  - The first recursive call to function sort_array operates on the first half of the array, and the second recursive call operates on the second half of the array
  - When these two function calls return, each half of the array has been sorted
- The call to function `merge` with the indices for the two halves of the array, combines the two sorted arrays into one larger sorted array

In [2]:

```
def sort_array(data, low, high):
    """Split data, sort subarrays and merge them into sorted array."""
    # test base case size of array equals 1
    if (high - low) >= 1:  # if not base case
        middle1 = (low + high) // 2  # calculate middle of array
        middle2 = middle1 + 1  # calculate next element over

        # output split step
        print(f'split:   {subarray_string(data, low, high)}')
        print(f'         {subarray_string(data, low, middle1)}')
        print(f'         {subarray_string(data, middle2, high)}\n')

        # split array in half then sort each half (recursive calls)
        sort_array(data, low, middle1)  # first half of array
        sort_array(data, middle2, high)  # second half of array

        # merge two sorted arrays after split calls return
        merge(data, low, middle1, middle2, high)
```

Function `merge`

- The `while` loop iterates until the end of either subarray is reached, merging the sorted subarrays
- When the `while` loop completes, one entire subarray has been placed in the combined array, but the other subarray still contains data
    - The `if/else` statement after the loop uses slices to fill the appropriate elements of the combined array with the remaining elements
- Finally, the function copies the combined array into the original array that `data` references

In [3]:

```
# merge two sorted subarrays into one sorted subarray
def merge(data, left, middle1, middle2, right):
    left_index = left  # index into left subarray
    right_index = middle2  # index into right subarray
    combined_index = left  # index into temporary working array
    merged = [0] * len(data)  # working array

    # output two subarrays before merging
    print(f'merge:   {subarray_string(data, left, middle1)}')
    print(f'         {subarray_string(data, middle2, right)}')

    # merge arrays until reaching end of either
    while left_index <= middle1 and right_index <= right:
        # place smaller of two current elements into result
        # and move to next space in arrays
        if data[left_index] <= data[right_index]:
            merged[combined_index] = data[left_index]
            combined_index += 1
            left_index += 1
        else:
            merged[combined_index] = data[right_index]
            combined_index += 1
            right_index += 1

    # if left array is empty
    if left_index == middle2:  # if True, copy in rest of right array
        merged[combined_index:right + 1] = data[right_index:right + 1]
    else:  # right array is empty, copy in rest of left array
        merged[combined_index:right + 1] = data[left_index:middle1 + 1]

    data[left:right + 1] = merged[left:right + 1]  # copy back to data

    # output merged array
    print(f'         {subarray_string(data, left, right)}\n')
```

Function `subarray_string`

- Throughout the algorithm, we display portions of the array to show the split and merge operations

In [4]:

```
# method to output certain values in array
def subarray_string(data, low, high):
    temp = '   ' * low  # spaces for alignment
    temp += ' '.join(str(item) for item in data[low:high + 1])
    return temp
```

Function `main`

In [5]:

```
def main():
    data = np.array([34, 56, 14, 20, 77, 51, 93, 30, 15, 52])
    print(f'Unsorted array: {data}\n')
    merge_sort(data)
    print(f'\nSorted array: {data}\n')
```

In [6]:

```
# call main to execute the sort (we removed the if statement from the script in the book)
main()
```

```
Unsorted array: [34 56 14 20 77 51 93 30 15 52]

split:   34 56 14 20 77 51 93 30 15 52
         34 56 14 20 77
                        51 93 30 15 52

split:   34 56 14 20 77
         34 56 14
                  20 77

split:   34 56 14
         34 56
               14

split:   34 56
         34
            56

merge:   34
            56
         34 56

merge:   34 56
               14
         14 34 56

split:            20 77
                  20
                     77

merge:            20
                     77
                  20 77

merge:   14 34 56
                  20 77
         14 20 34 56 77

split:                  51 93 30 15 52
                        51 93 30
                                 15 52

split:                  51 93 30
                        51 93
                              30

split:                  51 93
                        51
                           93

merge:                  51
                           93
                        51 93

merge:                  51 93
                              30
                        30 51 93
```

```
split:                             15 52
                                   15
                                      52

merge:                             15
                                      52
                                   15 52

merge:                    30 51 93
                                   15 52
                          15 30 51 52 93

merge:    14 20 34 56 77
                          15 30 51 52 93
          14 15 20 30 34 51 52 56 77 93


Sorted array: [14 15 20 30 34 51 52 56 77 93]
```

11.13.2 Big O of the Merge Sort

- Merge sort is far more efficient than insertion or selection sort
- Consider the first (nonrecursive) call to `sort_array`
- This results in two recursive calls to `sort_array` with subarrays each approximately half the original array's size, and a single call to `merge` , which requires, at worst, n – 1 comparisons to fill the original array, which is O(n)
- The two calls to `sort_array` result in four more recursive `sort_array calls` , each with a subarray approximately a quarter of the original array's size, along with two calls to `merge` that each require, at worst, n/2 – 1 comparisons, for a total number of comparisons of O(n)
- This process continues, each `sort_array` call generating two additional `sort_array` calls and a `merge` call until the algorithm has split the array into one-element subarrays

11.13.2 Big O of the Merge Sort (cont.)

- At each level, O(n) comparisons are required to merge the subarrays
- Each level splits the arrays in half, so doubling the array size requires one more level
- Quadrupling the array size requires two more levels
- This pattern is logarithmic and results in $\log_2 n$ levels
- Total efficiency of O(n log n) which, of course, is much faster than the O(n^2) sorts we studied