11 - Merge Sort

21:48

Mark Lesson Complete View Code Prev Next

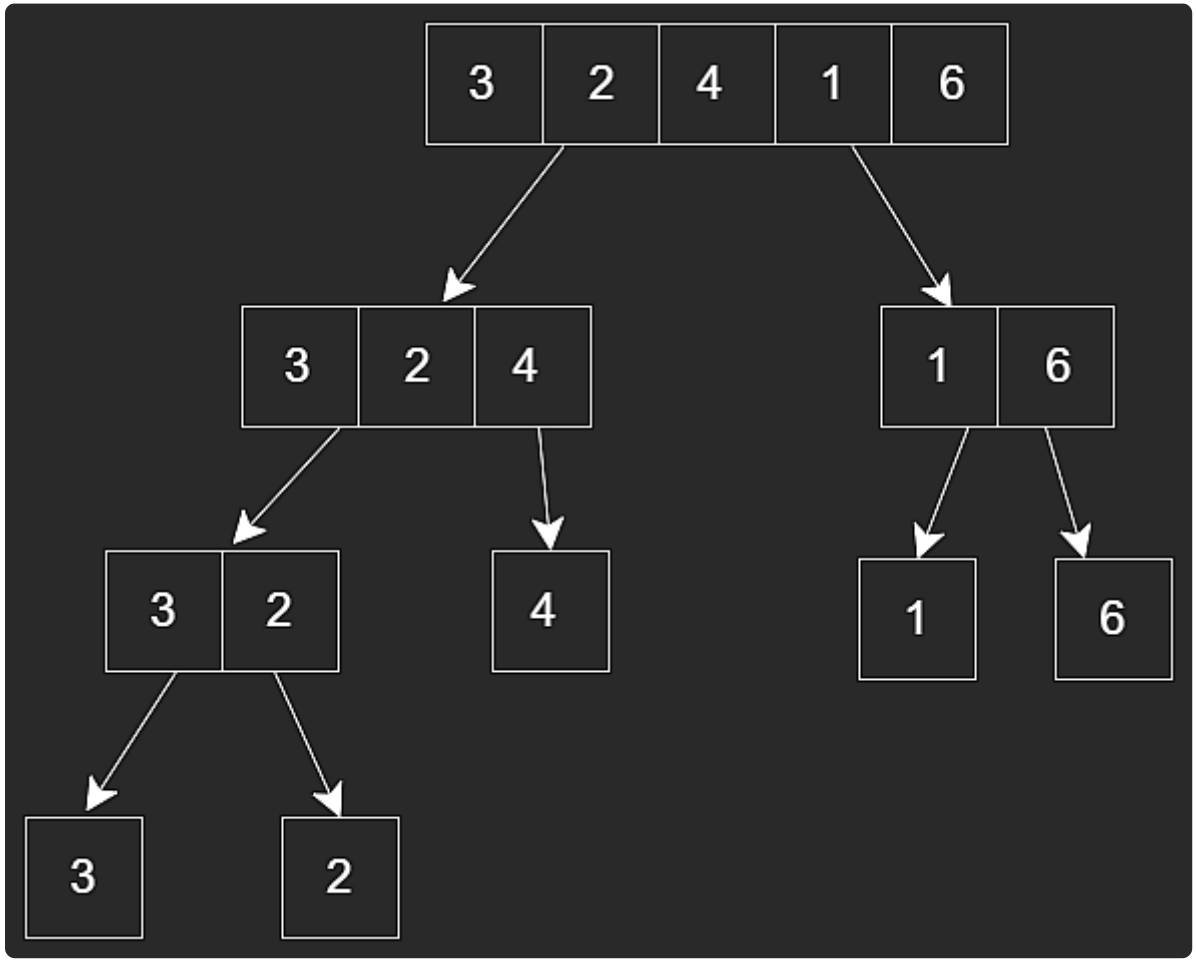
Suggested Problems

Status	Star	Problem	Difficulty	Video Solution	Code
☐	☆	Sort An Array	Medium		
☐	☆	Merge K Sorted Lists	Hard	▶	C++

Merge Sort

The concept behind merge sort is very simple. Keep splitting the array into halves until the subarrays hit a size of one, sort them, and merge the sorted arrays, which will result in an ultimate sorted array. You might have figured out that this sounds exactly like the fibonacci sequence using recursion, and you would be right! We can, and will be using recursion to perform this. More specifically, two branch recursion.

Let's take an array of size 5 as an example, `[3,2,4,1,6]` . Our job is to make sure that we sort this in an increasing, or non-decreasing order if we had duplicates. We will be splitting the array like the following.



As observed, we have two branches. Let's work on sorting and merging the left branch first. The work required here is that we will have to hit the base case first, after which we can begin sorting and merging the arrays together, achieving `[2,3,4]` as a result. Once our recursion reaches the base case, we have two subarrays, `[3]` and `[2]`. We need a way to compare these two elements to know where to put them in their original subarray, which is `[3,2]`. For this, copies of both

the subarrays is created and using two-pointers, values are compared and put in the original subarray in the sorted order. This can be seen in the pseudocode below.

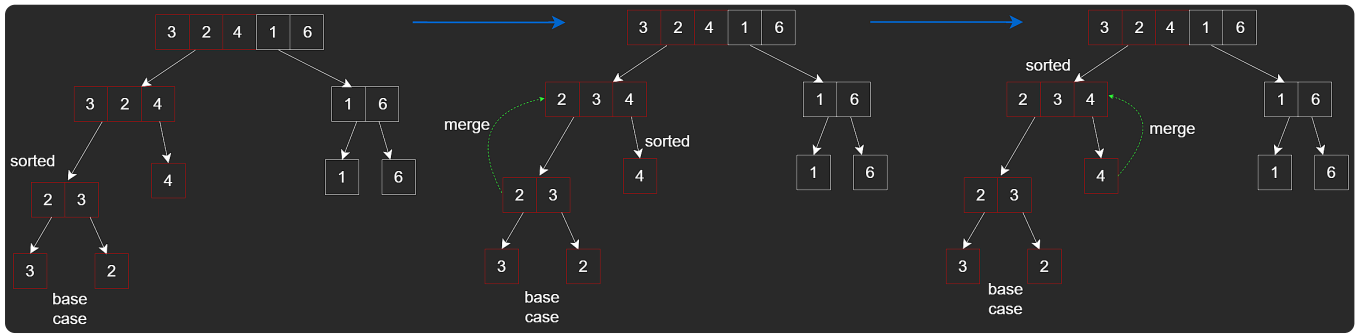
```
fn mergeSort(arr, s, e):
    // If the current subarray is of size 1 or 0 - base case
    if e - s + 1 <= 1:
        return arr
    // Split the array into two equal halves
    m = (s + e)/2
    // Keep splitting until base case is hit
    mergeSort(arr, s, m)
    mergeSort(arr, m + 1, e)
    // Merge arrays once sorted
    merge(arr, s, m, e)
    // Return the resulting array
    return arr
```

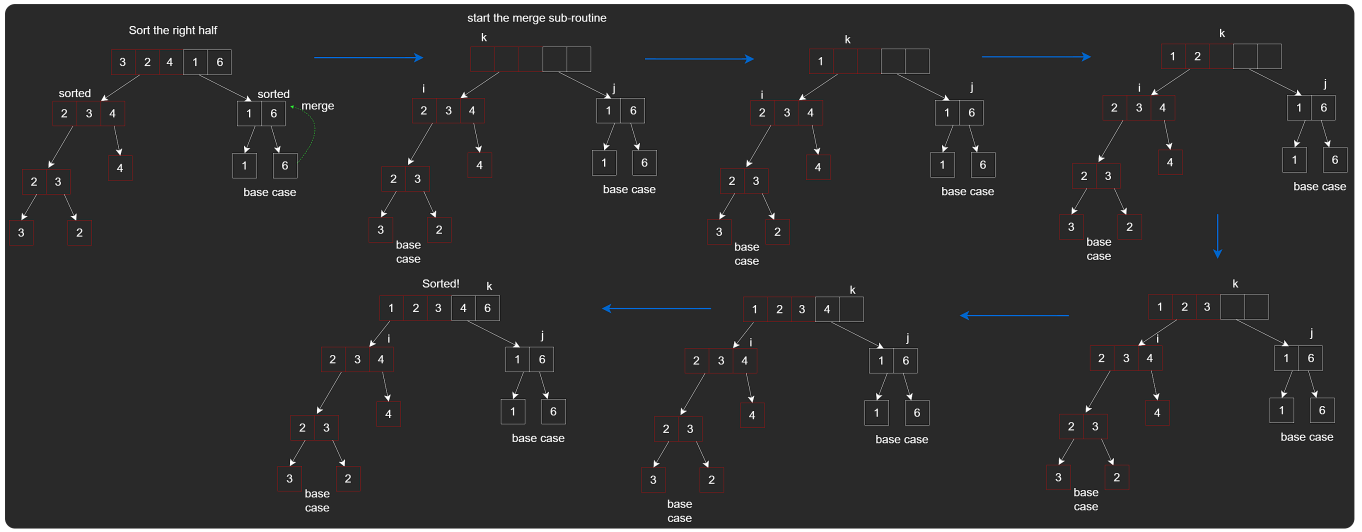
Visualization and Pseudocode

The `mergeSort()` recursive call

As we learned with two branch recursion, we solve both the branches and 'piece' back together the solutions to the subproblems to arrive at the ultimate solution. Once we have the subarray `[3,2]` sorted to `[2,3]` - this is the `mergeSort(arr, s,`

`m)` part. Now, we can move on to sorting the `[4]`, which corresponds to the `mergeSort(arr, m + 1, e)`. It is important to note the sequence in which the calls are exectued. The `merge()` call will not be exectued until both the recursive `mergeSort()` calls have returned for the current subarray. The first visual below shows sorting and merging the left half. The second visual shows sorting and merging the second half to get the ultimate sorted array.





> *For the sake of the example and making sure that the above visual is not confusing, the visual above shows how the final array, which is the resulting array, comes about. It visualizes what is going on inside the `merge()` function.*

The `merge()` function and three pointers

As observed in the visual above, we have three pointers, `k`, `j` and `i`.

- `k` keeps track of where the next element in `arr` needs to be placed.
- `i` points to the element in the `leftSubarray` that is currently being compared to the `j` element in the `rightSubarray`.
- One of `i` or `j` will increment depending on which element in smaller.
- `k` will increment regardless because `arr` will have an element placed inside of it regardless of which one of `i` or `j` increments.

This is clear in the visual above and demonstrated in the `merge()` function pseudocode below.

```
fn merge(arr, l, m, r):
    leftSubarray = new array of length m - l + 1
    rightSubarray = new array of length r - m

    i = 0 // starting index for leftSubarray
    j = 0 // starting index for rightSubarray
    k = l // starting index of arr, the merged array
```

```
// Comparing both arrays to find the smaller value
while i < leftSubarray.length and j < rightSubarray.length:
    if leftSubarray[i] <= rightSubarray[j]:
        arr[k] = leftSubarray[i]
        i += 1
    else:
        arr[k] = rightSubarray[j]
        j += 1
    k += 1

// Subarrays may not be split in equal length
// This will exhaust any items that are left in either
while i < leftSubarray.length:
    arr[k] = leftSubarray[i]
    i += 1
    k += 1
while j < rightSubarray.length:
    arr[k] = rightSubarray[j]
    j += 1
    k += 1
```

> *Recall that even though the visual only demonstrates the merging of the ultimate subarray, recursion tells us that the merge happens on every level after the arrays are sorted because we would never have gotten to the ultimate array if the subarrays had not been sorted and merged.*

> *The piece of code used for `i` pointer and `j` pointer is actually referred to as the two pointer technique and is extremely useful when we have two arrays and need to go through them simultaneously to perform some logic. This could actually be used to combine two arrays and do so in $O(n)$ time.*

Time Complexity

Merge Sort runs in $O(n\ log\ n)$. How so? Let's do some analysis. Recall from the fibonacci example when we kept splitting each sub-problem into two other sub-problems. We have a similar case here because our recursive tree is a tree with branching-factor of 2, except we are going the opposite direction. If n is the length of our array at any given level, our subarrays in the next level have a length $n/2$.

From our example above, we go from $n = 4$ to $n = 2$ to $n = 1$ which is the base case. The question here is how many times can we divide n by 2 until we hit the base case i.e. This would look like $n/2^x$ where x is the number of times we need to divide n by two until we get to one. Indeed,

$$n/2^x = 1$$

> *Isolate n by multiplying both sides by 2^x*

$$n = 2^x$$

> *To solve for x, take log of both sides*

$$log\ n = log 2^x$$

> *According to log rules, we can simplify this to:*

$$log\ n = x\ log\ 2$$

> *$log\ 2$ is basically asking 2 to the power of what is equal to 2 i.e . $2^? = 2$, which is just 1*

$$log\ n = x\ 1\ log\ n = x$$

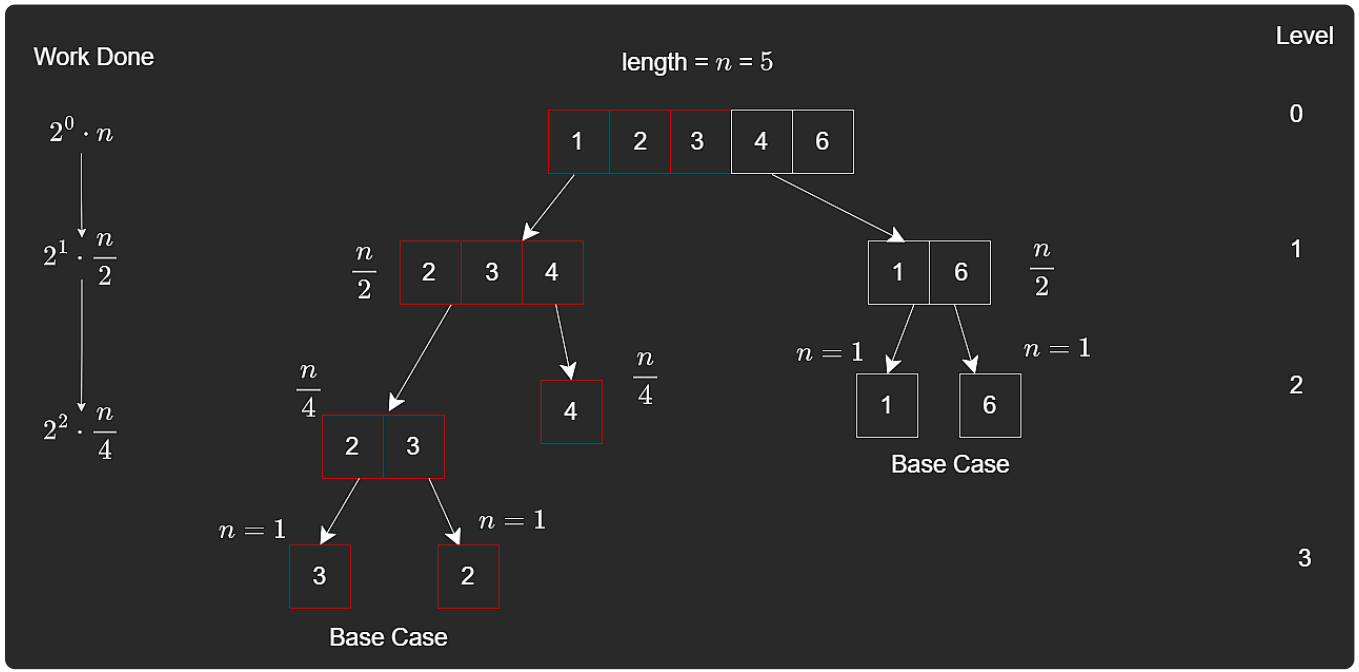
> *The ultimate answer*

$$x = log\ n$$

Having solved for x, our resulting answer is $log\ n$ which is the complexity for the `mergeSort` recursive call. We are not done yet, however.

Let's analyze the `merge` subroutine. The merge subroutine will take n steps because at any level of the tree, we have n elements to compare and sort, where n is the length of the original array.

This results in a $O(n\ log\ n)$ time complexity. The visual below gives more detail on how we arrived to our result. If you are interested, you can solve for the summation of the "Work Done" column on the left and you will arrive to the same answer.



Stability

Merge Sort proves to be a **stable** algorithm because if we have a pair of duplicates, say, 7, the 7 in the left subarray will always end up in the merged array first followed by the 7 in the right subarray. This is because when we compare the `ith` element in the left subarray to the `jth` element in the right subarray for equality, we pick the

one in the left subarray, maintaining the relative order. Recall the following pseudocode from the `merge()` subroutine.

```
if leftSubarray[i] <= rightSubarray[j]:
    arr[k] = leftSubarray[i]
    i += 1
```

Closing Notes

So how does merge sort stack up with insertion sort? In the worst case scenario, insertion sort runs in $O(n^2)$ with merge sort running in $O(n\ log\ n)$ in the worst, average and best case scenarios, making merge sort superior. The only time where insertion sort might be preferred if it is known that the array has fewer elements and is already, or nearly sorted as it would skip the swapping. But, merge sort is more efficient in terms of time because like we said before, unless we know the contents of the input given, insertion sort will perform worse than merge sort.

Github Privacy Terms