CS50 Merge Sort

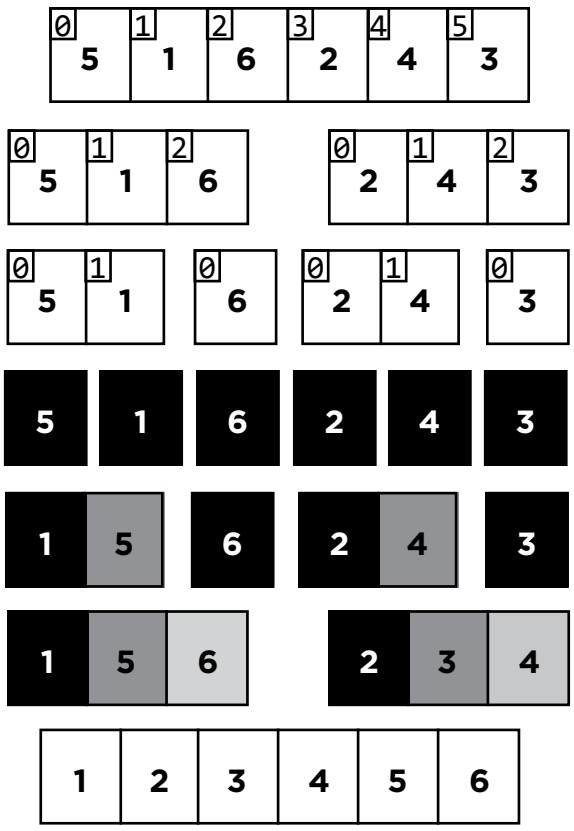
Overview

Sorting algorithms like selection sort, insertion sort, and bubble sort all suffer from the same general limitations and thus have the same worst-case runtime of $O(n^2)$. **Merge sort**, on the other hand, is fundamentally different, leveraging recursion to "pass the buck" of sorting, accomplishing a drastically superior runtime: *O(n log n)*!

Key Terms

- merge sort
- array
- recursive
- pseudocode

Step-by-step process for merge sort



Implementation

Merge sort works by breaking an **array** into sub arrays and merging the subarrays back in a **recursive** way. To understand how this works, let's take a look at the following **pseudocode**:

```
merge sort:
     if number of elements < 2
          return
     else
          sort the right half
          sort the left half
          merge sorted halves
```

Using the lines above and the array on the left (containing these numbers: 5 1 6 2 4 3), we are going to sort the left and right halves of the elements and merge them together. Note that when running merge sort, we only need enough space to store two copies of the array, despite the fact that the diagram on the left appears to require more space. At this point, we have no way of sorting the right or left halves, so we are going to recursively call the merge sort function. Similarly, we are going to continue to do this until we are left with all arrays of size 1. We'll need to handle running into an odd number of elements in a consistent way. Here, we implemented our program such that the left side of the split will have one more element than the right if the array has an odd number of elements.

After the elements are broken down into arrays of size 1, we are able to merge the sorted halves, since any array of size 1 is considered sorted. When we merge the two halves, we are removing the smallest numbers from the subarrays and appending them to the merged array, repeating until all elements of both subarrays are used up. (Note: The smallest elements will always be at the beginning of the subarrays, so we only need to check the first elements in the respective subarrays.) Since 6 was a single element array in the previous iteration, it does not need to be merged. We continue to do this until all the right and left halves are sorted from the previous iteration. Upon the next iteration, when we merge the arrays back into arrays of size 3, we need only look at the 0th index of each subarray to find the smallest element of the newly-merged array. In this case, this would be 1 and 6 for the left half and 2 and 3 for the right. Since 1 and 2 are the lowest numbers of their respective sides, they go into the 0th indices of the newly-merged array. And we'll continue to merge arrays in this way until the array is fully sorted.

Sorted Arrays

Like selection sort, merge sort has the same runtime in the best and worst case scenarios. Consider running merge sort on an already sorted array: since our program would have no way of knowing that it had already been sorted, it would have carry out the entire process the same way that it would with an unsorted array.