Data Structures Advanced Sorts Part 1: Mergesort

Phil Tayco

Slide version 1.4

Nov. 11, 2021

Advanced Sorts

Practical recursion

- Now that we have done some exercises to learn recursion, we can look at some practical ways to apply it
- Two things to keep in mind when reviewing these sort algorithms are the key parts of the recursive case:
  - The recursive case contains code where other operations or actions take place along with the recursive function call
  - The recursive case makes a function call on a smaller set of information

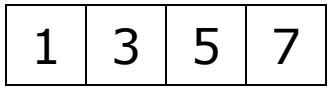
Advanced Sorts

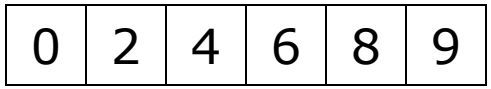
Merge

- Consider 2 arrays that are already sorted
- The algorithm to merge them into one sorted array requires comparing the first elements between the arrays and putting the smaller value into the new array
- Whichever array element gets selected, the index of it is increased and the process is repeated

Advanced Sorts

Example start







Advanced Sorts

First merge compares elements at first index of both arrays





Advanced Sorts

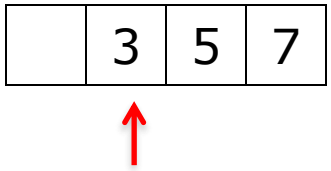
2nd array has the smaller value so that number is moved into the new array at the first spot. The 2nd array pointer is also increased by 1

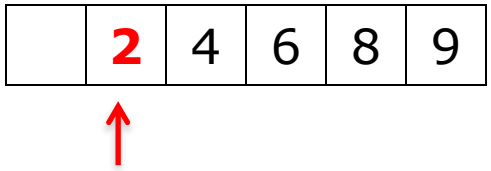




Advanced Sorts

On the next iteration, the 1st array element is the smaller value between the two and is subsequently moved

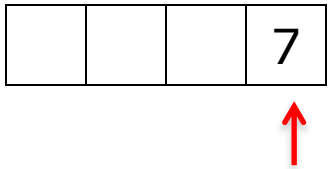


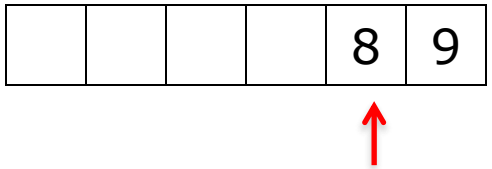




Advanced Sorts

After a few more iterations, the new array contains more of the elements from the 2 arrays while maintaining the sort







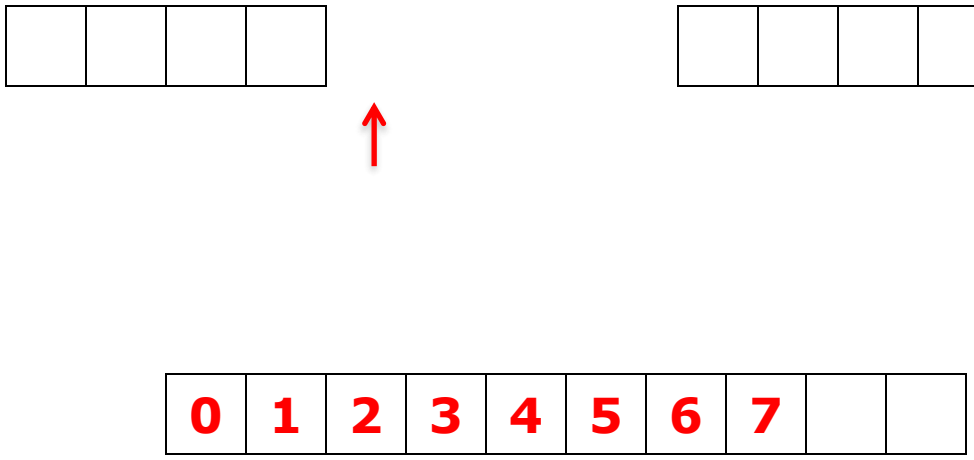
Advanced Sorts

Merge

- Eventually, one of the arrays will be empty
- At this point, because of the logic of the algorithm, the other array will contain the remaining elements to place into the new array
- These remaining elements will already be sorted and will be greater than the last element placed into the new array
- This means we simply add those elements in linearly to complete the merge

Advanced Sorts

The 7 has just been added into the array, completing the merge of elements from the 1st array



Advanced Sorts

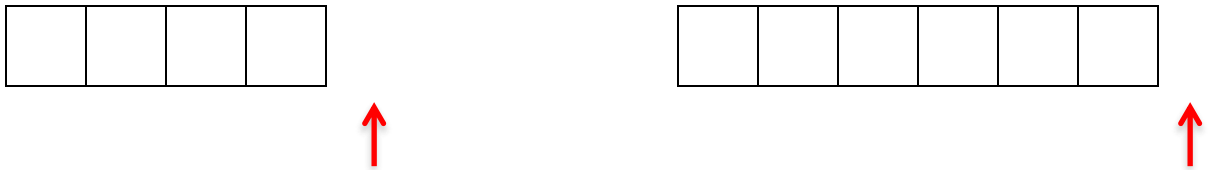
All that remains is to linearly add the remaining elements in 2nd array in the remaining spots of the new array





Advanced Sorts

When the remaining array is empty, the merge is complete





Advanced Sorts

Observations

- Using comparisons, the amount of time to fill in the new array is simply O(n) (where n is the number of total elements to merge)
- For memory space, the 2 arrays sorted that are merged require another array allocated that is equal to the size of the arrays combined (2 * n the total amount of space required)
- It’s a systematic algorithm that can be coded many ways

Advanced Sorts

```
int[] arrayOne = {1, 3, 5, 7};
int[] arrayTwo = {0, 2, 4, 6, 8, 9};
int[] mergedArray = new int[10];

int oneIndex = 0;
int twoIndex = 0;
int mergedIndex = 0;

int oneSize = arrayOne.length;
int twoSize = arrayTwo.length;
```

Advanced Sorts

```
while (oneIndex < oneSize && twoIndex < twoSize)
   if (arrayOne[oneIndex] < arrayTwo[twoIndex])
       mergedArray[mergedIndex++] = arrayOne[oneIndex++];
   else
       mergedArray[mergedIndex++] = arrayTwo[twoIndex++];

while (oneIndex < oneSize)
   mergedArray[mergedIndex++] = arrayOne[oneIndex++];

while (twoIndex < twoSize)
   mergedArray[mergedIndex++] = arrayTwo[twoIndex++];
```

Advanced Sorts

Code Analysis

- This example uses fixed values and variables without a function call to demonstrate the merge algorithm
- The 3 while loops make up the merge algorithm using these accessible variables. The comparisons in total will logically work out to O(n)
- In a general setting as part of a function, we'll need to manage these variables (pass them in and out effectively) to get the algorithm to work
- The key is to recognize that the 2 arrays are already sorted

Advanced Sorts

Mergesort

- Given the merge algorithm discussion, if you have 2 arrays that are part of an overall array of numbers to sort, you can merge them if the 2 arrays are separated and already sorted
- Another way of looking at this is to take the overall array in question and split it in half
- We then need to sort the left and right sides and then merge them together
- Thinking in recursive terms, if the algorithm is to perform Mergesort, then the recursive case is Mergesort the left side, Mergesort the right side and then merge the sides together

Advanced Sorts

Mergesort

- What's the base case? Remember that this represents when the recursion process stops
- For this algorithm, there's no need to sort an array that's already sorted
- We only know the array is sorted for sure if we're only looking at one element
- The overall recursive algorithm is then:
  - If the array has only one element, exit (no need to sort)
  - Otherwise, sort the array by:
    - Mergesort (left side)
    - Mergesort (right side)
    - Merge(left and right side of array together)
- This is best viewed with an example array

Advanced Sorts

Start with Mergesort(full array)

7	1	4	5	8	2	3	6

Advanced Sorts

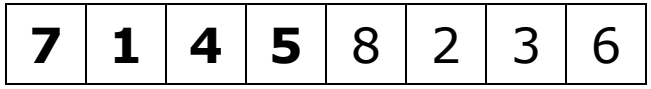
Base case not reached. Mergesort (array[0..3])

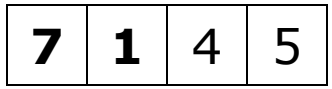
7	**1**	**4**	**5**	8	2	3	6

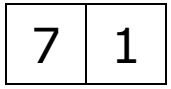
7	1	4	5

Advanced Sorts

Base case not reached again. Mergesort (array[0..1])

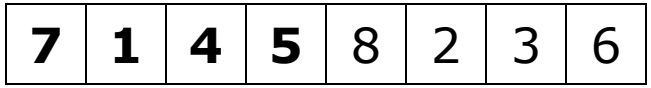


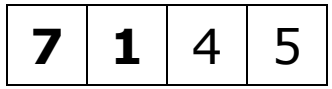


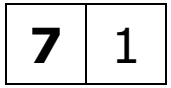


Advanced Sorts

Base case not reached again. Mergesort (array[0])



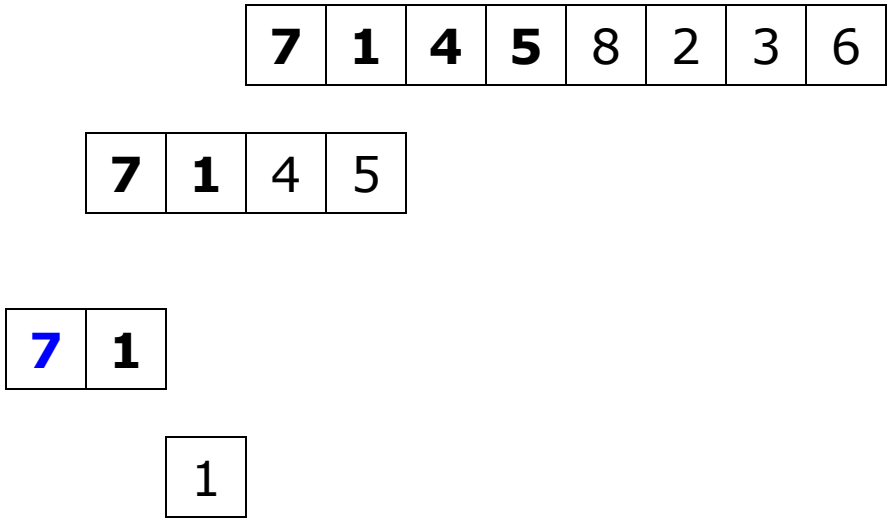






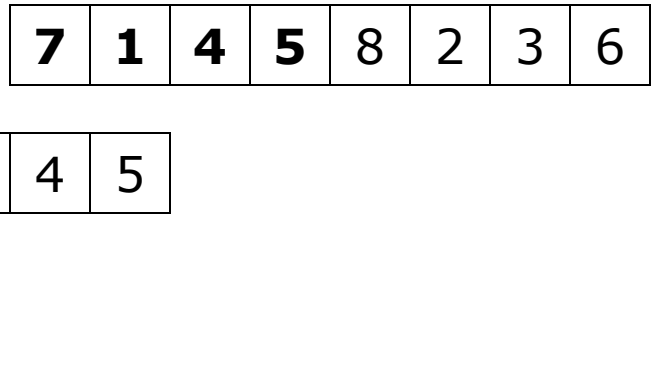
Advanced Sorts

Base case reached, return to Mergesort(array[0..1]) and now Mergsort its right side (array[1])



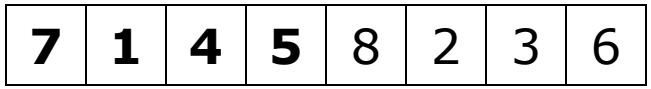
Advanced Sorts

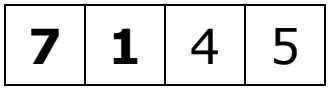
Base case reached again, return to Mergesort(array[0..1])

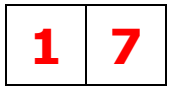


Advanced Sorts

For Mergesort([0..1]), the Mergesort calls for its left and right side are complete. Now Merge [0] and [1]

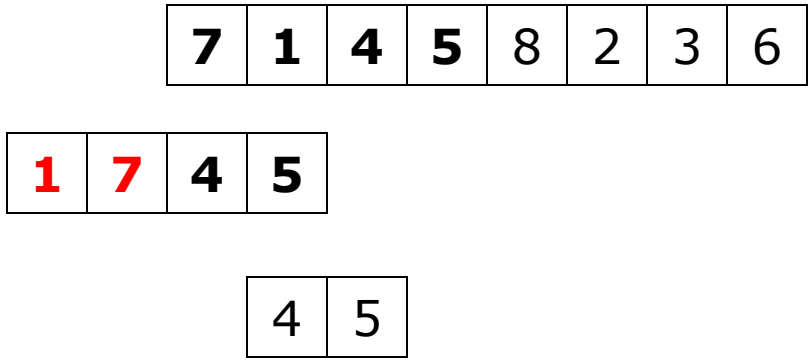






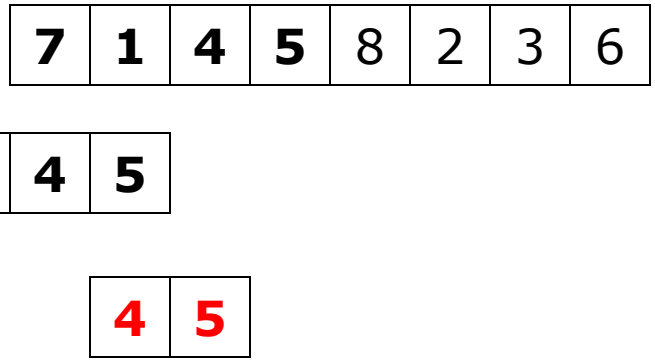
Advanced Sorts

For Mergesort([0..3]), the Mergesort([0..1]) is done.
Now call Mergesort for its right side ([2..3])



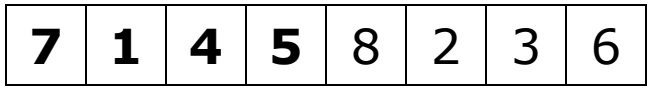
Advanced Sorts

Similar to Mergesort([0..1]), Mergesort([2..3]) will hit 2 base cases and merge (notice there is no net change)



Advanced Sorts

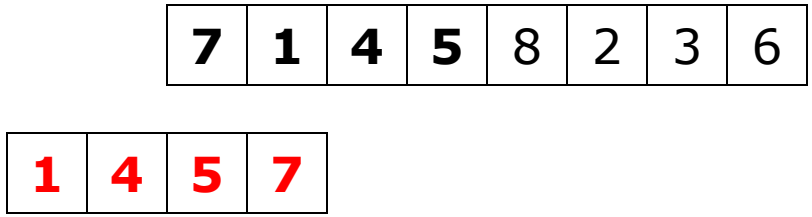
We return to Mergesort([0..3]). Its left and right are now mergesorted so we perform a merge





Advanced Sorts

This results in a merge that completes the sorting of [0..3]



Advanced Sorts

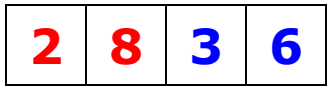
We return to the original Mergesort([0..7]). The left side is done and the process repeats on the right side!



Advanced Sorts

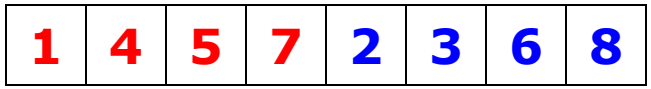
The recursive process is the same resulting sorting [4..7] followed by [4..5] and [6..7]





Advanced Sorts

[4..5] and [6..7] merge and return to [0..7] leaving only one step left to perform at the [0..7] level…



Advanced Sorts

The last step in Mergesort[0..7] is a merge! The array is now sorted!



Now, let's take a look at the code…

Advanced Sorts

```
public class AdvancedSortingSupplement
{
   public static int[] numbers;
   public final static int MAX_NUMBERS = 15;
   public static Random gen;

   public static void mergeSort()
   {
       int[] temp = new int[numbers.length];
       recursiveMergeSort(temp, 0, numbers.length - 1);
   }
```

Advanced Sorts

```
private static void recursiveMergeSort(int[] temp,
   int low, int hi)
{
    if (low == hi)
        return;

    int mid = (low + hi) / 2;
    recursiveMergeSort(temp, low, mid);
    recursiveMergeSort(temp, mid+1, hi);
    merge(temp, low, mid+1, hi);
}
```

Advanced Sorts

```
private static void merge(int[] temp, int low,
   int mid, int high)
{
    int j = 0;
    int loBound = low;
    int midBound = mid - 1;
    int n = high - loBound + 1;

    while (low <= midBound && mid <= high)
        if (numbers[low] < numbers[mid])
            temp[j++] = numbers[low++];
        else
            temp[j++] = numbers[mid++];
```

Advanced Sorts

```
    while (low <= midBound)
        temp[j++] = numbers[low++];

    while (mid <= high)
        temp[j++] = numbers[mid++];

    for (int c = 0; c < n; c++)
        numbers[loBound+c] = temp[c];
}
```

Advanced Sorts

Mergesort analysis

- This code contains a global array called “numbers” representing the actual array storing the necessary data
- When mergeSort is called, it sorts numbers using the merge sort algorithm
- The mergeSort method itself creates a temporary array the size of the number of elements numbers has
- The recursiveMergeSort is then called passing in the temp array and numbers’ lower and upper bounds

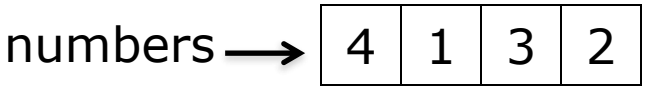
Advanced Sorts

Mergesort analysis

- The recursiveMergeSort function uses the temporary array as a parameter while applying the merge sort recursive algorithm
- When the merge occurs, the same merge algorithm previously discussed merging the appropriate sub arrays within the temp array
- When the appropriate merge of temp array is applied, those contents are copied back into numbers – this is where the extra memory space for the merge sort is required and utilized
- When all merges are done, the temp array will no longer be used and the memory is freed

Advanced Sorts

Example of code walkthrough with 4 elements

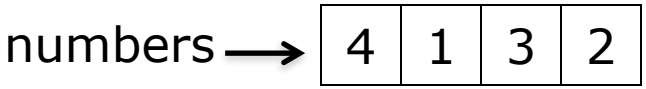


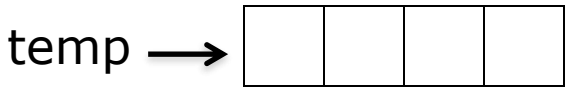

```
main()

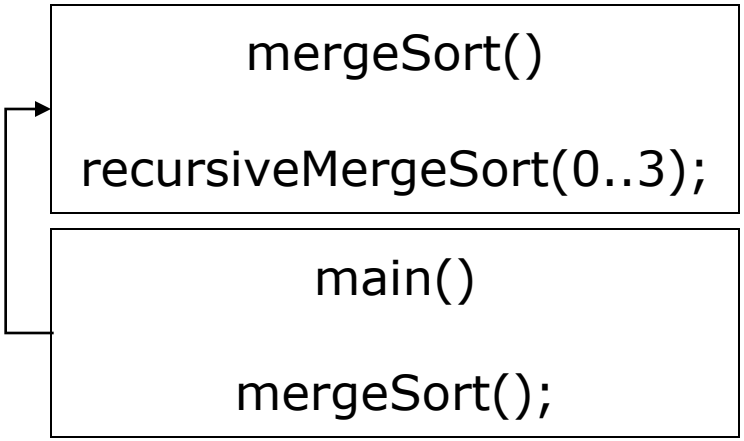
mergeSort();
```

Advanced Sorts

main calls mergeSort to start the process. Temp space is created

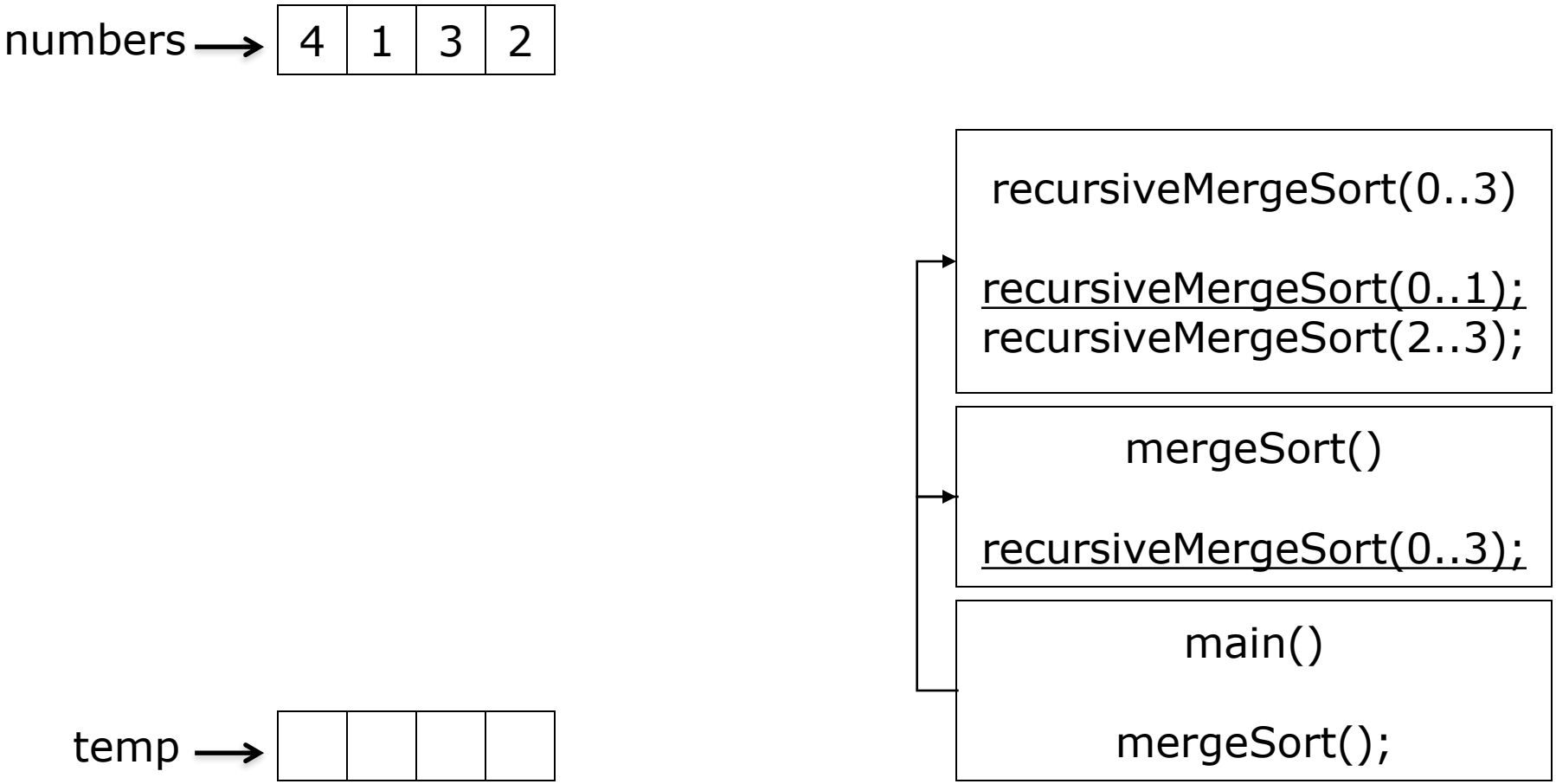






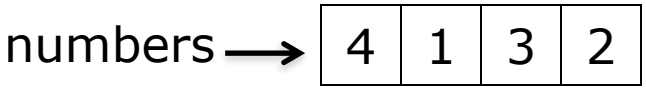
Advanced Sorts

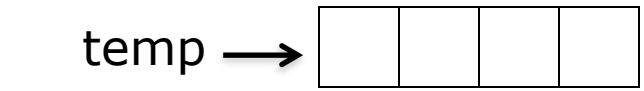
The first recursive call occurs with mergeSort(0..1)

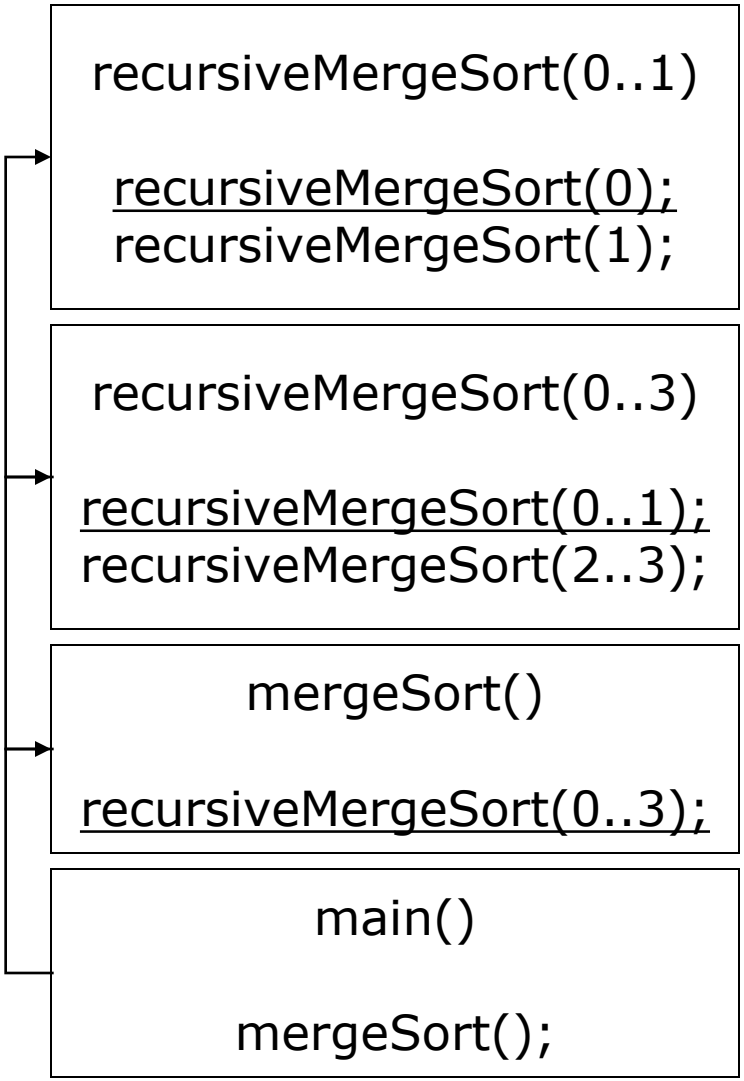


Advanced Sorts

(0..1) will make recursive calls as well

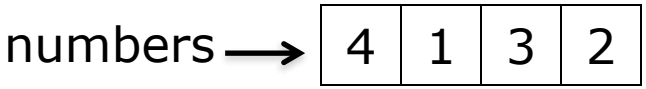


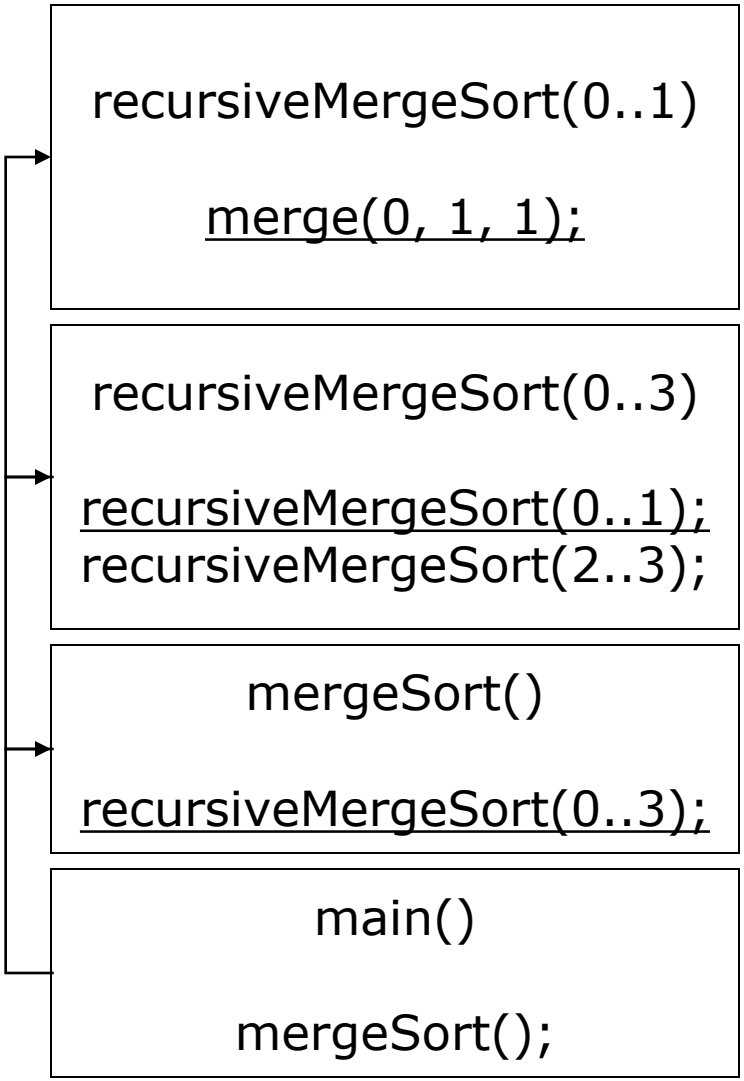


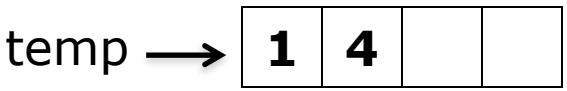


Advanced Sorts

The call to merge will merge [0] and [1] within temp





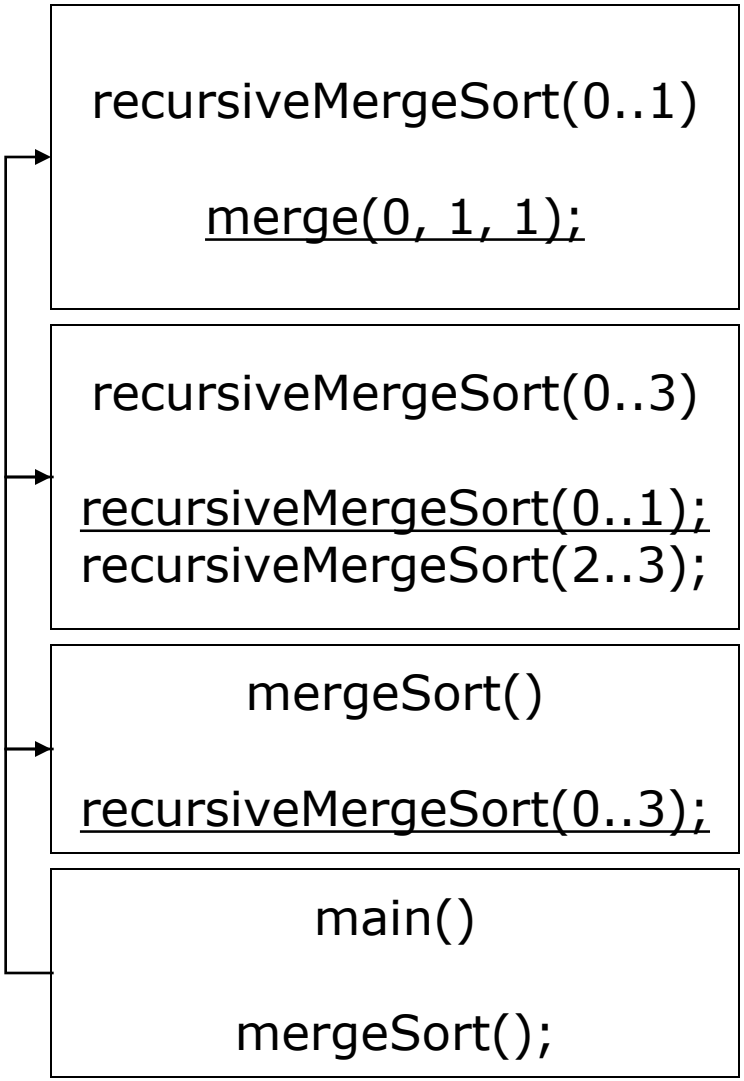


Advanced Sorts

Last step of merge copies temp data to theArray

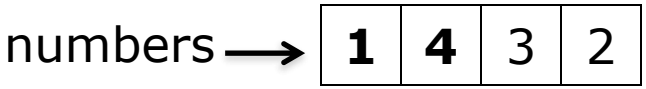
numbers → | **1** | **4** | 3 | 2 |

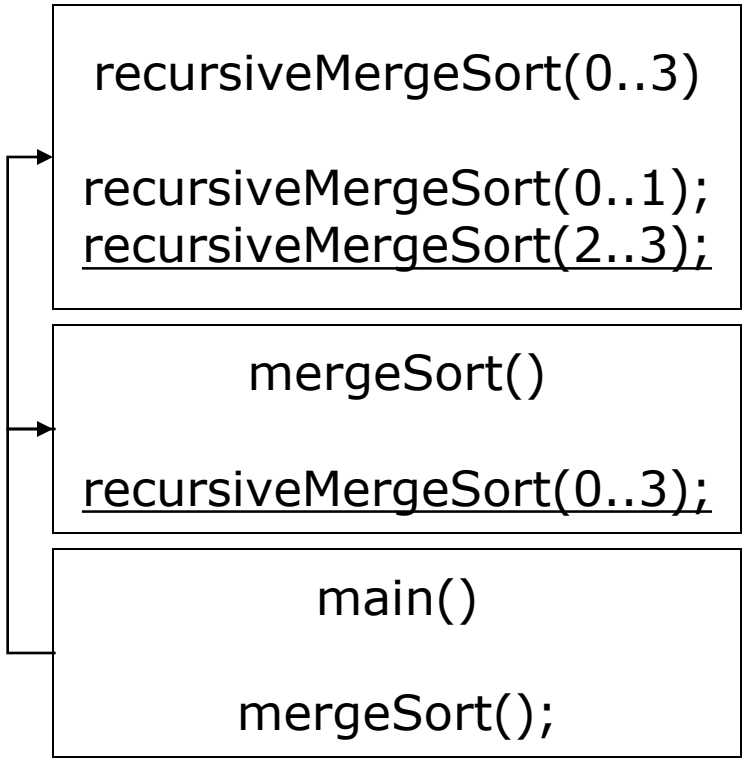
temp → | **1** | **4** | | |

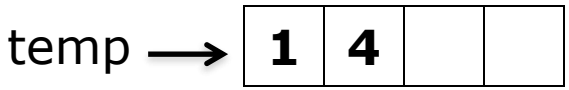


Advanced Sorts

After merge, (0..1) is done. We return to (0..3) where (2..3) is now called







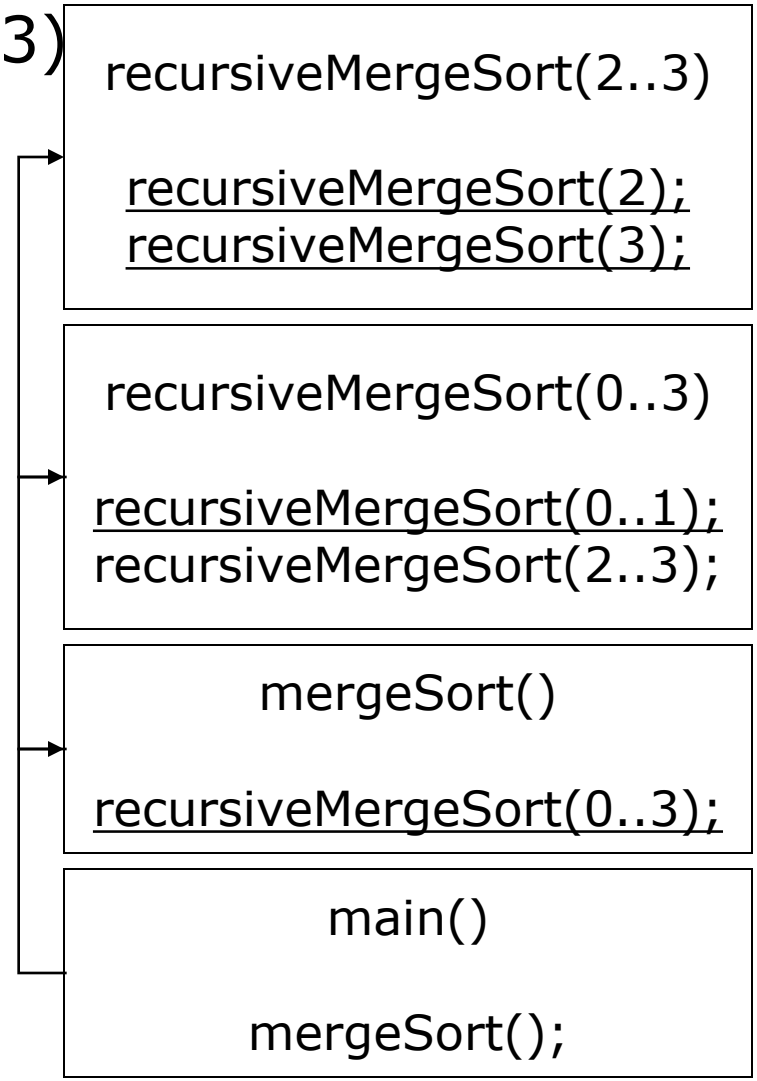
Advanced Sorts

(2..3) recursively calls (2) and (3) which are base cases

numbers → | **1** | **4** | 3 | 2 |

Notice temp data from (0..1) is still present…

temp → | 1 | 4 | | |



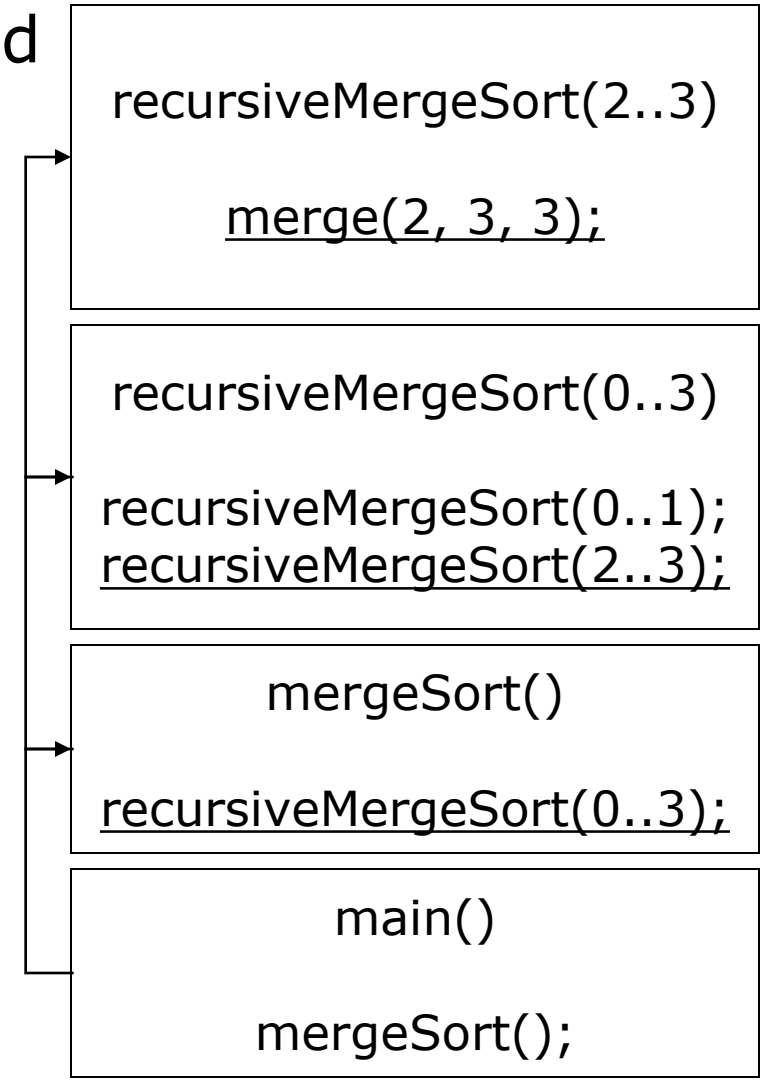
Advanced Sorts

3rd step of (2..3) calls merge and will merge 3 and 2 in temp

numbers → | **1** | **4** | 3 | 2 |

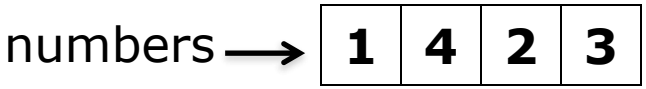
Notice the 2 and 3 replace the 1 and 4 from the previous merge

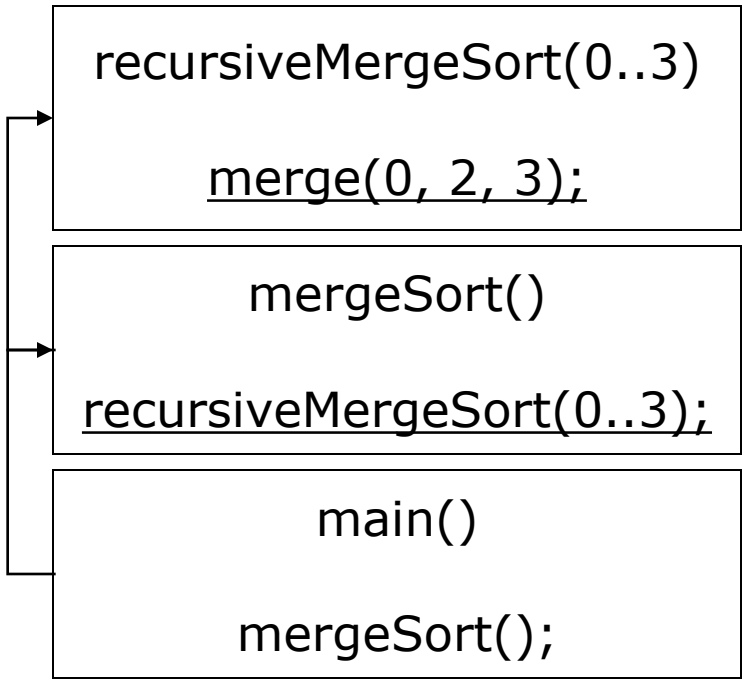
temp → | **2** | **3** | | |

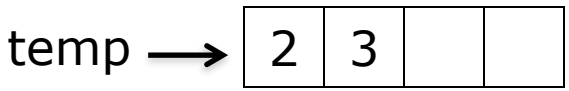


Advanced Sorts

Temp's [2..3] data is copied back to theArray.
(2..3) is done. Back in (0..3), now we merge

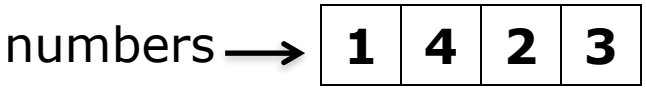


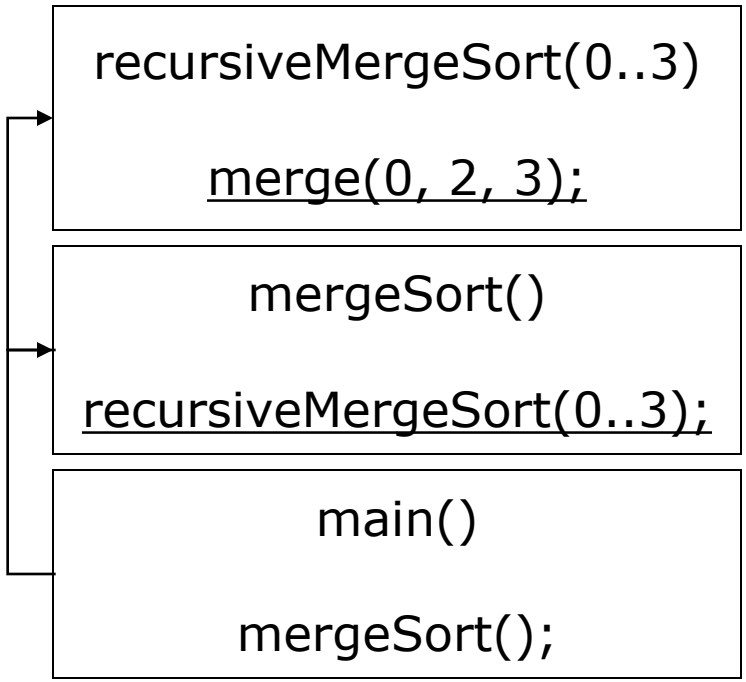


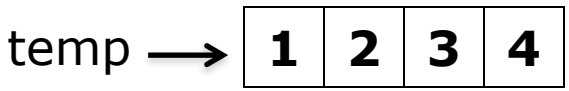


Advanced Sorts

Merge of 0..3 in temp occurs by merging theArray[0..1] and theArray[2..3]

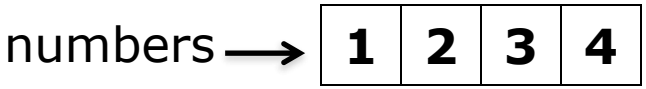


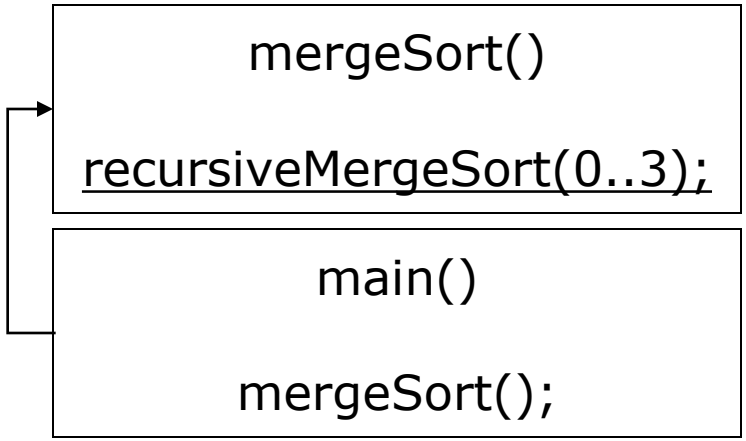


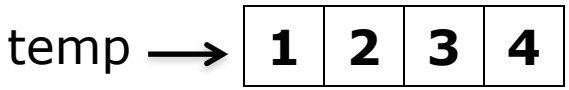


Advanced Sorts

Data from temp copied back to theArray and completes the merge of [0..3]

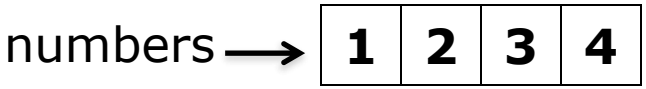






Advanced Sorts

mergeSort function is now complete removing the need for temp and theArray is now sorted




```
main()

mergeSort();
```

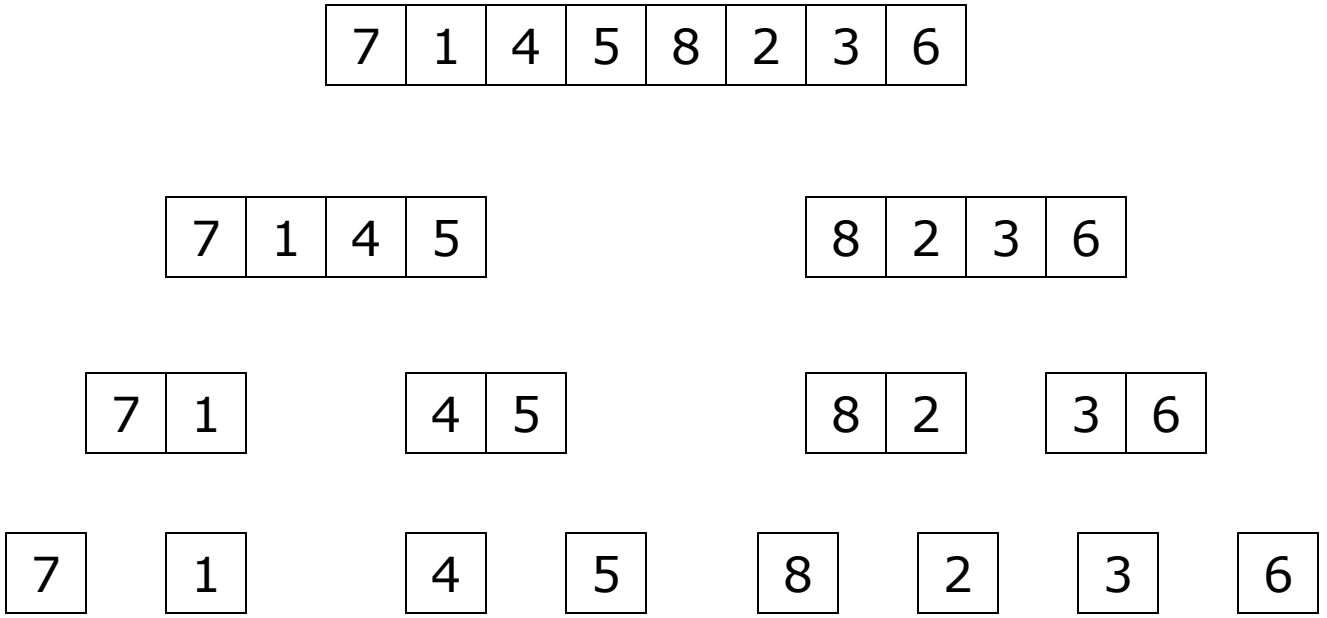
Advanced Sorts

Efficiency

- Looking again at comparison operations, there are 2 areas where comparisons occur:
  - Checking for base case
  - Performing the merge
- The base case is only one comparison operation and will execute with each recursive function execution
- The merge function contains 4 loops and will execute when the recursive instances occur
- A visual helps with seeing function instances and which cases are occurring

Advanced Sorts

Here's the number of recursive function executions for an array of 8 elements



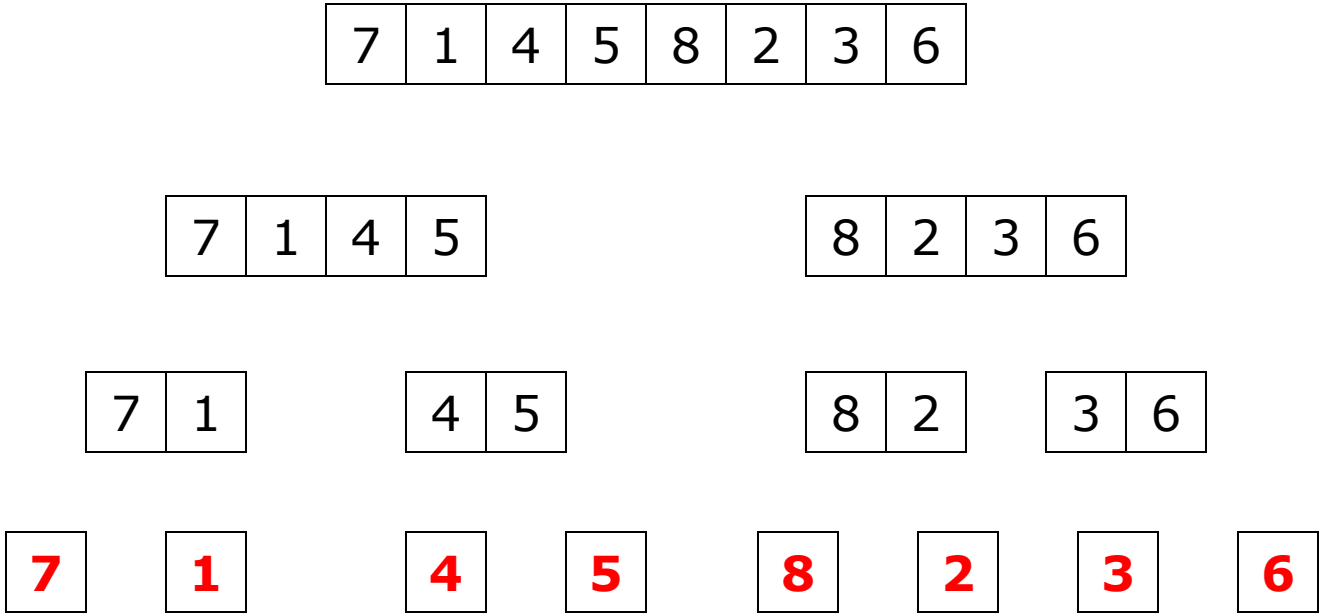
Each "block" represents an instance when a recursiveMergeSort function is executed

Advanced Sorts

- We could count the number of comparisons being performed in these cases and eventually see a pattern with the counts
- The category is what we're after though. Recall that with simple sorts, we had a similar look at the number of elements being reviewed with each iteration
  - Each row effectively was dealt with a categorical O(n) number of elements
  - The number of rows that appeared in the worst cases with each algorithm was n times
- This led to an $O(n^2)$ performance. If we use a similar analysis here, we notice there are less rows of iteration and different types of execution happening per row

Advanced Sorts

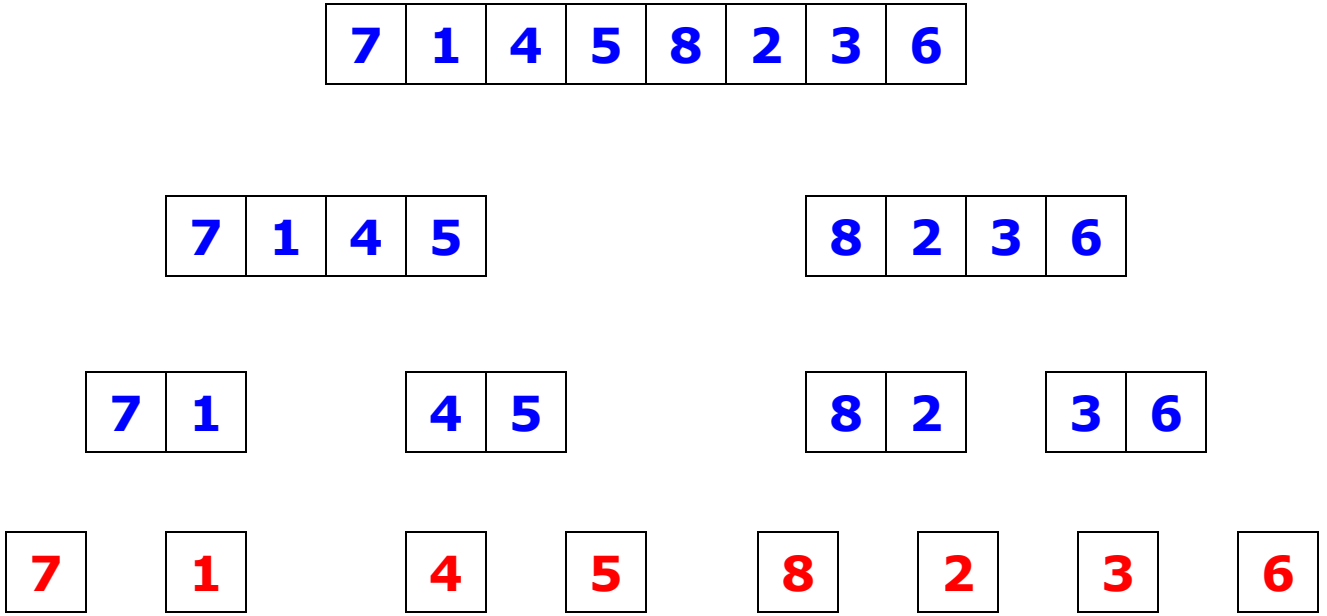
Base cases are reached in **red**



The base case level is the lowest one making the recursive cases the remaining top levels

Advanced Sorts

Recursive cases are in **blue**



3 levels of recursive cases for N = 8

Advanced Sorts

- The number of recursive function call instances is represented with each set of blocks
- Each time the recursiveMergeSort function is called, the base case is checked. If the base case is reached, that's it for that function (only 1 comparison)
- This is represented with the blocks in red on the illustration and they represent a minimal number of comparison operations being performed (base case check and then you're done)
- The blocks in blue, though, have a lot more action because in those instances, the merge function is called

Advanced Sorts

- The merge function contains 4 loops. They are not nested though so if each loop somehow represented execution of n times, it would still be categorically O(n)
- What really matters is how many times merge is called in relation to the original n
- The top level, let's call it level 1, merges a total of 8 elements. This puts the category of comparisons at this level at O(n)
  - If you count the number of comparisons that actually get executed at level 1 during the merge, it's around 4n, but categorically it's still O(n)
- Level 2 performs will have 2 function executions, each performing a merge
- That may seem like twice as much but here, each merge is on a smaller set of data

Advanced Sorts

- A similar effect is on level 3, with 4 merges performed on even smaller amounts of data
- Level 4 is nice because no merge takes place in any of those blocks (they are base cases) so we're effectively looking at levels 1-3
- The use of the word "level" helps with this analysis because now notice each level contains a total of n elements ultimately being merged
  - Level 1 does 1 merge of n elements
  - Level 2 does 2 merges, but each is (n/2) elements
  - Level 3 does 4 merges, but each is (n/4) elements
- If we get picky, the increased number of merge functions does increase the comparison operation count at each level, but is not as significant as what we will look at next

Advanced Sorts

- What it comes down to for analyzing the efficiency is how many times the merge function operates on a total of n elements, and not how many times the merge function is called
- This works out to how many levels of n elements where the recursive case occurs (i.e. the blocks in blue)
- This example works out nicely because we can see an equally divisible number of elements in each block with each level
- So now the question is how many levels do we get given a size of n elements?

Advanced Sorts

- Mathematically, it works out as follows:
  - N = 8: 3 levels of recursive cases merging a total of n elements
  - N = 16: 4 levels
  - N = 32: 5 levels
  - N = n: $\log_2$ n levels
- We've seen this pattern before with the binary search algorithm and is a recurring pattern with analysis where algorithms are applying this "divide and conquer" approach
- Here, the division is recursively analyzing half the array
- This results in the number of levels where merge occurs on n elements as O(log n)
- Each level represents comparisons performed at an O(n) category, therefore…

Advanced Sorts

O(n log n)

- This gives us another category of performance
- It lies in between $O(n^2)$ and O(n) and is significantly better than $O(n^2)$
- The actual counts, especially for smaller sized arrays may exceed the counts for the simple sort algorithms, but remember what Big-O measures: performance as the list grows in size
- If you work out growing numbers of n, clearly O(n log n) outperforms $O(n^2)$
- There are other factors to consider still

Advanced Sorts

- Optimized BubbleSort and regular InsertionSort actually perform close to O(n) when the array is nearly sorted
- MergeSort is a very consistent O(n log n) no matter what the condition of the array
- MergeSort also requires twice the memory space because of the temp array – this is unavoidable because of the nature of the merge
- Because MergeSort uses recursion, the size of the function call stack can be a factor where the simple sorts don't
- On random distribution of data, though, MergeSort clearly outperforms the simple sorts if you can take the memory hit

Advanced Sorts

Summary:

	Comparisons	Notes
Bubble	$O(n^2)$ (worst case)	Optimized version is like Insertion when near sorted
Selection	$O(n^2)$	Swaps are O(n)
Insertion	$O(n^2)$ (worst case)	Range is O(n) to (n^2) Excellent for partially sorted lists
Merge	O(n log n)	Requires 2x memory space because of temp array

Advanced Sorts

Final thoughts

- Mergesort compared to the simple sorts in the worst cases is a significant improvement
- The downside to mergesort is the use of temp space
- Simple sorts operate in the space of the given array
- It would be nice if we could keep the n log n efficiency without the use of extra space. Guess where we will head next…