```
void merge(int arr[], int l, int m, int r)
{

       cout << "l: " << l << " m: " << m << " r: " << r << endl;
       printArray(arr, 11);
       cout << endl;



       if (arr[m] < arr[m + 1]) {         return;
       }

       int  a = l; // create temp variable pointing at first element in first half
        int b = m + 1; // create temp variable pointing at first element in second half
       while (a < b && b <= r) { // while everyting still w/in bounds
              //     printArray(arr, sizeof(arr) / sizeof(arr[0]));
              if (arr[a] > arr[b]) { // if l > r, then push everything down, and insert
                     int temp = arr[b];
                     for (int i = b; i >= a; i--) {//operation that causes the
bottleneck.
                            arr[i] = arr[i - 1]; //reason code is stable
                     }
                     arr[a] = temp; //reason code is stable
                     a++; // increment a and b
                     b++; // increment a and b
              }
              else { // if l <= r, then leave it be and increment a
                     a++;
              }
       }
       // ----------------------------------------

}
```

The code above only showcases the merge function. This is because the recursive calls for this code is identical to a traditional merge sort. This means that the recursive tree still has log(n) +1 levels. The best-case scenario occurs when the first number on the right subarray array is greater than the last number on the left sub array. This means they don't need to merge because they are already in order and the code will then leave the function. If this was the case every time the function was called (i.e. the array is already sorted). The run time would be O(log(n)). The worse case occurs when the entire right subarray needs to be moved to the left. This is because the for loop that must shift the elements (for loop inside the while loop) will be called every time. If the merge is doing the first layer (n elements) then the run time of the merge function will be (n/2) *(n/2) or (n^2/4). This means that the run time during the worst condition would be equal to O(n^2log(n)).

Like traditional merge sort this implementation is also stable. This is because the data movement portion of the code (where while loop starts) always puts like elements in the same order as the original array. For example, if the data movement portion was processing the numbers (11, 12, 3, 3), after the first pass through the while loop the array would now be (3, 12, 11, 3). The while loop would run through again and the array would become (3, 3, 11, 12). This example clearly shows that the order of 3s in the original array is the same as the sorted array.