Quick Merge Sort
Jeff Edmonds

It is hard to believe that Merge Sort can be improved after the 70 years since Von Neumann invented it in 1945, but who knows. The input and the output of the sorting are to be in the same array A, but an auxiliary array B is needed. After recursively sorting the first and second halves of its input, Merge Sort merges these taking the values from A into B and then has to copy the result back into A. Much attempt has been made avoid this auxiliary memory – at the expense of extra time and complications. Here we still use B, but avoid the extra copying time. This cuts in half the number of times an element moves, from $2n \log n$ to $n \log n \ + n$ moves. As the program recurses, the input will always appear in array $A[]$, but the required location of the output alternates between $A[]$ and $B[]$.

algorithm $QuickMergeSortToA(A[], iBegin, iEnd, B[])$

$\langle$***pre−cond***$\rangle$**:** The values $A[iBegin, iEnd]$ are to be sorted.

$\langle$***post−cond***$\rangle$**:** Then the result must appear in $A[iBegin, iEnd]$.
$B[iBegin, iEnd]$ is used as an auxiliary.

```
begin
    if(iEnd − iBegin > 0) {
        iMiddle = (iEnd + iBegin)/2;                  // iMiddle = midpoint
        QuickMergeSortToB(A, iBegin, iMiddle, B)      // Sort first half into B.
        QuickMergeSortToB(A, iMiddle+1, iEnd, B)      // Sort second half into B.
        Merge(B, iBegin, iMiddle, iEnd, A);           // merge from B to A.
    }
end algorithm
```

algorithm $QuickMergeSortToB(A[], iBegin, iEnd, B[])$

$\langle$***pre−cond***$\rangle$**:** The values $A[iBegin, iEnd]$ are to be sorted.

$\langle$***post−cond***$\rangle$**:** Then the result must appear in $B[iBegin, iEnd]$.

```
begin
    if(iEnd − iBegin = 0)
        B[iBegin] = A[iBegin];                        // Each element moves like this at most once.
    else {
        iMiddle = (iEnd + iBegin)/2;                  // iMiddle = midpoint
        QuickMergeSortToA(A, iBegin, iMiddle, B)      // Sort first half into A.
        QuickMergeSortToA(A, iMiddle+1, iEnd, B)      // Sort second half into A.
        Merge(A, iBegin, iMiddle, iEnd, B);           // merge from A to B.
    }
end algorithm
```

Note that in normal merge sort, for each of the $\log_2 n$ levels of recursion each of the n elements is moved once for the *merge* and once to be copied back for a total of $2n \log n$ moves. Here the $1n \log n$ moves still happen for the *merge* but not for the copy. For a random value of n, half the basecases will have to move the element, but each element moves at most once, for an at most n additional moves.