Mergesort

Weiss, from the beginning of chapter 8 through section 8.3 and section 8.5 (pages 351–357 and 361–364)
Many useful algorithms are recursive. To solve a problem, they call themselves recursively to deal with subproblems.

The divide-and-conquer involves three steps:
- Divide the problem into a number of subproblems.
- Conquer the subproblems by solving them recursively. If they are small enough, just solve them.
- Combine the solutions to the subproblems to be used as the solution to the original problem.
Merge Sort

- Merge sort consists of the following steps:
  - Divide an n-element array into two sub-arrays, where each sub-array has n/2 elements.
  - Sort the two sub-arrays recursively.
  - Combine the two sorted sub-arrays to one large sorted array.
Divide and Conquer

left=0
right=6
public static <AnyType extends Comparable<? Super AnyType>>
(void mergeSort( AnyType[ ] a ) {
    AnyType[ ] tmpArray = (AnyType[ ]) new Comparable[a.length];
    mergeSort(a, tmpArray, 0, a.length - 1);
}

private static <AnyType extends Comparable<? Super AnyType>>
(void mergeSort( AnyType[ ] a, AnyType[ ] tmpArray, int left, int right ) {
    if (left < right ) {
        int center = (left + right) / 2;
        mergesort(a, tmpArray, left, center);
        mergesort(a, tmpArray, center + 1, right);
        merge(a, tmpArray, left, center + 1, right);
    }
}

Conquer step takes O(n) time.
Merge Sort

- Consider sorting [9 4 2 8 3 5 1]
- Assume that an array is a sequence of one-element sub-arrays

- Merge sub-array 1 with sub-array 2, 3 with 4, 5 with 6. This generates 2-element sorted sub-arrays (last sub-array is not full).

- Merge sub-array 1 with sub-array 2, 3 with 4. This generates 4-element sorted sub-arrays.

- Merge sub-array 1 with sub-array 2. This generates the full size sorted array.
Merge Sort

- Consider merging two sub-arrays.
- Create an *auxiliary* array. The size of this array is the sum of the sizes of the sub-arrays.
- Hold two pointers. Each pointer points to the *first* and the *smallest* element of its sub-array.
- Copy back auxiliary array into the original array.

<table>
<thead>
<tr>
<th>leftPos</th>
<th>rightPos</th>
<th>rightEnd</th>
<th>tmpPos</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 4 8 9</td>
<td>1 3 5</td>
<td>0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>
Merge Sort

2 4 8 9
2 4 8 9
2 4 8 9
2 4 8 9
2 4 8 9
2 4 8 9
2 4 8 9
2 4 8 9

1 3 5
1 3 5
1 3 5
1 3 5
1 3 5
1 3 5
1 3 5
1 3 5

1 0 0 0 0 0 0 0
1 2 0 0 0 0 0 0
1 2 3 0 0 0 0 0
1 2 3 4 0 0 0 0
1 2 3 4 5 0 0 0
1 2 3 4 5 8 0 0
1 2 3 4 5 8 9 0

1 0 0 0 0 0 0 0
1 2 0 0 0 0 0 0
1 2 3 0 0 0 0 0
1 2 3 4 0 0 0 0
1 2 3 4 5 0 0 0
1 2 3 4 5 8 0 0
1 2 3 4 5 8 9 0
private static <AnyType extends Comparable<? Super AnyType>>
void merge( AnyType[ ] a, AnyType[ ] tmpArray, int leftPos, int rightPos, int rightEnd) {

    int leftEnd = rightPos – 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;
    while( leftPos <= leftEnd && rightPos <= rightEnd )
        if( a[leftPos].compareTo(a[rightPos]) <= 0 )
            tmpArray[tmpPos++] = a[leftPos++];
        else
            tmpArray[tmpPos++] = a[rightPos++];

    while(leftPos <= leftEnd)
        tmpArray[tmpPos++] = a[leftPos++];
    while(rightPos <= rightEnd)
        tmpArray[tmpPos++] = a[rightPos++];

    for(int i = 0; i < numElements; i++; rightEnd--)
        a[rightEnd] = tmpArray[rightEnd];
}
Time Complexity

- *Merge* operation copies each element *twice*
  - From the original array to the auxiliary array
  - From the auxiliary array to the original array
- For a sub-array of length $n$, merge performs $2n$ operations.

- Compute the runtime of merge sort using a recurrence relation $T(n)$.  
  $T(1) = 0$ base case
  $T(n) = 2n + 2T(n/2)$

- Expand recurrence and rewrite the relation in terms of $k$
  $T(n) = 2n + 2T(n/2)$
  $= 2n + 2(2n/2 + 2T(n/4))$
  $= 2n + 2n + 4T(n/4)$
  $= 2n + 2n + 4(2n/4 + 2T(n/8))$
  $= 2n + 2n + 2n + 8T(n/8)$
  $= 2n + 2n + 2n + 8T(n/8)$
  $= k2n + 2^kT(n/2^k)$
Time Complexity

• Recursion stop when $n/2^k = 1 \rightarrow k = \log_2 n$

$T(n) = 2n\log_2 n + 2^{\log_2 n} T(n/2^{\log_2 n})$

$= 2n\log_2 n$

$T \in O(n \log n)$