17.04 Virtual Lecture Notes

Merge sort can be one of the most efficient sorting algorithms, but also the most complicated to understand. You will need to review your eIMACs readings and these notes several times to get a complete grasp of this sort method. But once you have it down pat, it will be an asset to your library of sorting algorithms.

Merge sort works by using a divide and conquer methodology. Rather than work on the entire array at once, merge sort divides the array in half (or close to half if the array has an odd number of elements). The process is repeated on each half. Then it is repeated on those halves.

When a division results in sets of one element each, the sets are sorted by just comparing the elements. After that, the halves are merged back together. The merge process involves taking the smallest from the front of each array and adding it to a larger array. The process is repeated until both smaller arrays are empty. This way, the large array will have the elements from the other two arrays, still in sorted order. This is repeated until all sorted halves are finally merged together into one whole array again.

This works as the merge process combines the elements in sorted order. In fact, the merge sort algorithm discussed above sorts items into ascending order due to the merge process selecting the smallest element each time it merges. To sort descending, you change the merge process to select the largest.

First, let us look at the process for integers. This is always the easiest way to begin. First, take a look at the merge sort in general:

```
public static void mergeSort( int[] a, int low, int high )
{
  if ( low == high )
    return;

  int mid = ( low + high ) / 2;

  mergeSort( a, low, mid );
  mergeSort( a, mid + 1, high );

  merge( a, low, mid, high );
}
```

Note that this works by using recursive calls. Each time the merge sort method is called, it checks to see if low is equal to high. If they are equal, then this is a one-element array, and we can return. If they are not equal, then we have more than one element; therefore, we determine a mid point for the array. Since the array may have odd number of elements, we use the formula shown in the method. This is done using integer division, as that will give us an integer no matter if the number of elements is odd or even.

Then we recursively call our merge method with the array and the values for the first half of the array, and then again for the second half. After the recursive calls end, we call merge, which will sort the array.
Crucial to this functioning is the merge method. Here it is:

```
public static void merge( int[] a, int low, int mid, int high )
{
  int[] temp = new int[ high - low + 1 ];

  int i = low, j = mid + 1, n = 0;
  // put elements (sorted) into temp
  while ( i <= mid || j <= high )
  {

    if ( i > mid )
    // we have processed everything in the range low-mid
    // so now we finish what is left in range mid-high
    // by manipulating j
    {
      temp[ n ] = a[ j ];
      j++;
    }
    else if ( j > high )
    // we have processed everything in the range mid-high
    // so now we finish what is left in range low-mid
    // by manipulating i
    {
      temp[ n ] = a[ i ];
      i++;
    }
    else if ( a[ i ] < a[ j ] )
     // both sections have values so we compare and see if
     // smallest is in i position – if so, pull from there
    {
      temp[ n ] = a[ i ];
      i++;
    }
    else
     // smallest is in j position – pull from there
    {
      temp[ n ] = a[ j ];
      j++;
    }
    n++;
  }
  // put elements back into a in a[low] to a[high]
  for ( int k = low ; k <= high ; k++ )
```

```
    a[ k ] = temp[ k - low ];
}
```

The first thing merge does is to create a temporary array into which we will copy elements from our array as we sort them, using a merge process. After the elements are merged, then we will put them back into the array, but only in the positions where they belong; this is indicated by low and high. Read the algorithm and make sure you understand what is happening.

To apply it to our list of houses, we just need to modify the methods to use a **`HouseListing`** array and the **`getCost()`** method. To be more interesting, we are going to change merge so that it will result in us sorting in descending order, rather than the usual ascending order.

```
   public static void mergeSort(HouseListing[] a, int low,
                                                   int high)
    {
        if ( low == high )
            return;

        int mid = ( low + high ) / 2;

        mergeSort( a, low, mid );
        mergeSort( a, mid + 1, high);

        merge( a, low, mid, high);
    }
    public static void merge( HouseListing[] a, int low,
                              int mid, int high )
    {
        HouseListing[] temp =
                   new HouseListing[ high - low + 1 ];

        int i = low, j = mid + 1, n = 0;

        while ( i <= mid || j <= high )
        {
            if ( i > mid )
            {
                temp[ n ] = a[ j ];
                j++;
            }
            else if ( j > high )
            {
                temp[ n ] = a[ i ];
                i++;
            }
            else if ( a[ i ].getCost() > a[ j ].getCost() )
            // > is used so that it sorts descending
            // use < if want to sort ascending
            {
                temp[ n ] = a[ i ];
                i++;
            }
            else
            {
                temp[ n ] = a[ j ];
                j++;
            }
            n++;
        }

        for ( int k = low ; k <= high ; k++ )
            a[ k ] = temp[ k - low ];

    } // end of merge
```

Now you should try it and see how it works.

- Download the TestListing5.java file to your module 17 demo programs directory and open it.
- Run the file and make sure you understand it before you continue.