CS125 : Introduction to Computer Science

Lecture Notes #36 and #37
Mergesort

Lectures 36 and 37 : Mergesort

Mergesort is related to `InsertionSort` in the same way that `Quicksort` was related to `SelectionSort`. `SelectionSort` has one recursive call as the last work that needs to be done, and `Quicksort` has two recursive calls as the last work that needs to be done. `InsertionSort` began with one recursive call and then did other work; `MergeSort` will begin with two recursive calls and then do other work.

That is to say, the primary work on `MergeSort` occurs *after* we've already made two recursive calls to sort two parts of the array recursively. `Quicksort` did a bunch of partitioning work, and then, once the array was fully partitioned, made two recursive calls to sort the two halves of the partition. `Mergesort` will make recursive calls to sort both the first half, and then the second half, of the array – and so the work that remains after that is to *merge* the two sorted halves of the array – hence the name of the algorithm.

```
Mergesort:
   1) sort the first half of the array (recursively)
   2) sort the second half of the array (recursively)
   3) now that you have two sorted subarrays, merge them
        into one sorted array
```

So, just as `FindMinimum` was where most of the work was done in `SelectionSort`, just as `InsertInOrder` was where most of the work was done in `InsertionSort`, and just as `Partition` and `PartitionWrapper` were where most of the work was done in `Quicksort`, we will have a `Merge` algorithm in `Mergesort` that will handle most of the work for this algorithm. The recursive calls are all going to make `Merge` calls themselves, and over the lifetime of the algorithm, most of the time will be spent in `Merge`.

What does `Merge` do? Well, it takes two sorted collections and builds a single sorted collection out of them. This can actually be done in one pass down the data, because for any two sorted collections, it's easy to find the minimum value:

```
 a1   a2    a3   a4

 b1   b2    b3   b4

(a1 < a2 < a3 < a4, and b1 < b2 < b3 < b4)
```

Above are two sorted collections of values. And if we want to find the overall minimum, well, we know it can't be any of `a2`, `a3`, or `a4`, because the first collection is sorted so `a1` is smaller than all of them, and so if `a2`, `a3`, and `a4` aren't even the minimum of their own collection, they certainly can't be the overall minimum. (If we had duplicates, `a1` still has to be the smallest value of its collection, even of other values are tied with it.) Likewise, in the second collection, since it is sorted, `b1` is the minimum of that collection by definition, and so there is no way `b2`, `b3`, or `b4` can be the overall minimum since they aren't even the minimum of their own collection. So, the overall minimum has to be either `a1` or `b1`, and to find out which of the two is the overall minimum, we just compare them and select the smaller of the two values.

In the event that it is `a1`, that's the first element of your new sorted collection, and you get the rest of the sorted collection by merging the remaining two sorted collections recursively:

```
a1     then merge these:       a2    a3   a4

                               b1   b2    b3   b4

                              (a2 < a3 < a4, and b1 < b2 < b3 < b4)
```

And if instead `b1`, then that's the first element of your new sorted collection, and you get the rest of the sorted collection by merging the remaining two sorted collections recursively:

```
b1     then merge these:       a1   a2    a3   a4

                               b2    b3   b4

                              (a1 < a2 < a3 < a4, and b2 < b3 < b4)
```

If you reach the point where one collection is completely empty, you would then just automatically pick the minimum value from the other collection for your first element. If both collections are completely empty, then you have no elements and just return. For example:

```
Start of merge call #1:

   2   6   18   26  50

   1   10  20   23  25

1 < 2, so...


Start of merge call #2:

                      2   6   18   26  50

   1                  10  20   23  25

2 < 10, so...


Start of merge call #3:

                      6   18   26  50

   1  2               10  20   23  25

6 < 10, so...
```

```
Start of merge call #4:

                      18  26  50

   1  2  6             10  20   23  25

10 < 18, so...

Start of merge call #5:

                            18  26  50

   1  2  6 10              20  23  25

18 < 20, so...

Start of merge call #6:

                            26  50

   1  2  6 10 18           20  23  25

20 < 26, so...

Start of merge call #7:

                            26  50

   1  2  6 10 18 20         23  25

23 < 26, so...

Start of merge call #8:

                                26  50

   1  2  6 10 18 20 23          25

25 < 26, so...
```

```
Start of merge call #9:

                                    26  50

   1  2  6 10 18 20 23 25


first collection is all that's left, and min of that is 26, so...



Start of merge call #10:

                                        50

   1  2  6 10 18 20 23 25 26


first collection is all that's left, and min of that is 50, so...


Start of merge call #11:

   1  2  6 10 18 20 23 25 26 50


no collections are left, we are done, start returning from
recursive method calls
```

The one sticking point here is that we can't do this efficiently in place:

```
   2   6   18   26  50  1  10  20  23  25
 --------------------  -----------------
   first sorted          second sorted
       half                  half
```

Above, we see the two sorted collections we just merged in our example, stored side-by-side in an array as they would be after `MergeSort`'s two recursive calls. In this case, again, `1` is the minimum of the entire collection, since it is less than `2`. However, if we try to move `1` to the front of the array, the entire first collection – half the cells – needs to be shifted to the right one cell to make room for `1` to go at the front. If we did that shifting of the first collection repeatedly, that cost will add up.

So instead, we use a temporary array. We keep track of the lo and hi of each of the two collections, but have a temporary array that we write the next minimum into, cell by cell. This means we'll need to keep track of the next spot to write into in the temporary array as well. When we've finished the merge, we copy everything from the temporary array back into the original array. For example:

```
Start of merge call #1:

  2     6     18     26     50     1     10     20     23     25
  loA                       hiA    loB                       hiB


          // temp values written here
   cur    // index to temp array


1 < 2, so...


Start of merge call #2:

  2     6     18     26     50     1     10     20     23     25
  loA                       hiA          loB                 hiB


   1
         cur


2 < 10, so...


Start of merge call #3:

  2     6     18     26     50     1     10     20     23     25
        loA                 hiA          loB                 hiB


   1    2
               cur


6 < 10, so...
```

```
Start of merge call #4:

  2     6     18     26     50     1     10     20     23     25
              loA           hiA          loB                 hiB



   1    2     6
                   cur


10 < 18, so...


Start of merge call #5:

  2     6     18     26     50     1     10     20     23     25
              loA           hiA                 loB          hiB



   1    2     6      10
                          cur
```

... and so on. You know the first collection is empty when `loA > hiA`, and you know the second collection is empty when `loB > hiB`.

This gives us the following code:

```
public static void MergeSort(int[] A, int lo, int hi)
{
   if (lo < hi)
   {
      int mid = (lo + hi)/2;
      MergeSort(A, lo, mid);
      MergeSort(A, mid+1, hi);
      int[] temp = new int[hi-lo+1];
      Merge(A, temp, lo, mid, mid+1, hi, 0);
      Copy(temp, A, lo, hi, 0); // copy temp array back to A
   }
}
```

```
public static void Merge(int[] A, int[] temp, int cur1, int end1,
        int cur2, int end2, int cur3)
{
   if ((cur1 > end1) && (cur2 > end2))
      return;
   else if (cur1 > end1)
   {
      temp[cur3] = A[cur2];
      Merge(A, temp, cur1, end1, cur2+1, end2, cur3+1);
   }
   else if (cur2 > end2)
   {
      temp[cur3] = A[cur1];
      Merge(A, temp, cur1+1, end1, cur2, end2, cur3+1);
   }
   else if (A[cur1] < A[cur2])
   {
      temp[cur3] = A[cur1];
      Merge(A, temp, cur1+1, end1, cur2, end2, cur3+1);
   }
   else // (A[cur2] <= A[cur1]
   {
      temp[cur3] = A[cur2];
      Merge(A, temp, cur1, end1, cur2+1, end2, cur3+1);
   }
}



/*
   you could also combine those conditions, i.e.

   if (( cur1 > end1) && (cur2 > end2))
      return;
   else if ((cur1 > end1) || ((cur2 <= end2) && (A[cur2] <= A[cur1])))
   {
      temp[cur3] = A[cur2];
      Merge(A, temp, cur1, end1, cur2+1, end2, cur3+1);
   }
   else // ((cur2 > end2) || ((cur1 <= end1) && (A[cur1] < A[cur2])))
   {
      temp[cur3] = A[cur1];
      Merge(A, temp, cur1+!, end1, cur2, end2, cur3+1);
   }
 */
```

```
public static void Copy(int[] temp, int[] A, int lo, int hi, int index)
{
   if (index <= hi-lo)
   {
      A[lo+index] = temp[index];
      Copy(temp, A, lo, hi, index+1);
   }
}
```