Sergio E. Garcia Tapia
Algorithms by Sedgewick and Wayne (4th edition) [SW11]
October 13th, 2024

2.2: Mergesort

Exercise 1. Give a trace, in the style of the trace given at the beginning of this section, showing how the keys `A E Q S U Y E I N O S T` are merged with the abstract in-place `merge()` method.

Solution.

						a[]														aux[]						
k	0	1	2	3	4	5	6	7	8	9	10	11	i	j	0	1	2	3	4	5	6	7	8	9	10	11
	A	E	Q	S	U	Y	E	I	N	O	S	T														
	A	E	Q	S	U	Y	E	I	N	O	S	T			A	E	Q	S	U	Y	E	I	N	O	S	T
													0	6												
0	A												1	6	A	E	Q	S	U	Y	E	I	N	O	S	T
1	A	E											2	6		E	Q	S	U	Y	E	I	N	O	S	T
2	A	E	E										2	7			Q	S	U	Y	E	I	N	O	S	T
3	A	E	E	I									2	8			Q	S	U	Y		I	N	O	S	T
4	A	E	E	I	N								2	9			Q	S	U	Y			N	O	S	T
5	A	E	E	I	N	O							2	10			Q	S	U	Y				O	S	T
6	A	E	E	I	N	O	Q						3	10			Q	S	U	Y					S	T
7	A	E	E	I	N	O	Q	S					4	10				S	U	Y					S	T
8	A	E	E	I	N	O	Q	S	S				4	11					U	Y					S	T
9	A	E	E	I	N	O	Q	S	S	T			4	12					U	Y						T
10	A	E	E	I	N	O	Q	S	S	T	U		5	12					U	Y						
11	A	E	E	I	N	O	Q	S	S	T	U	Y	6	12						Y						
	A	E	E	I	N	O	Q	S	S	T	U	Y														

Exercise 2. Give traces, in the style of the trace given with Algorithm 2.4, showing how the keys `E A S Y Q U E S T I O N` are sorted with top-down mergesort.

Solution. The following shows the sequence of calls:

```
sort(a, 0, 11)
   sort(a, 0, 5) // left half
      sort(a, 0, 2)
         sort(a, 0, 1)
            merge(a, 0, 0, 1)
         sort(a, 2, 2)
            // no merge
         merge(a, 0, 1, 2)
      sort(a, 3, 5)
         sort(a, 3, 4)
            merge(a, 3, 3, 4)
         sort(a, 5, 5)
            // no merge
         merge(a, 3, 4, 5)
      merge(0, 2, 5) // done sorting left half
   sort(a, 6, 11)
      sort(a, 6, 8)
         sort(a, 6, 7)
            merge(a, 6, 6, 7)
         sort(a, 8, 8)
            // no merge
         merge(a, 6, 7, 8)
      sort(a, 9, 11)
         sort(a, 9, 10)
            merge(a, 9, 9, 10)
         sort(a, 11, 11)
            // no merge
         merge(a, 9, 10, 11)
      merge(a, 6, 8, 11) // done sorting right black
   merge(a, 0, 5, 11)
```

---

	a[]											
	0	1	2	3	4	5	6	7	8	9	10	11
	E	A	S	Y	Q	U	E	S	T	I	O	N
`merge(a, 0, 0, 1)`	A	E	S	Y	Q	U	E	S	T	I	O	N
`merge(a, 0, 1, 2)`	A	E	S	Y	Q	U	E	S	T	I	O	N
`merge(a, 3, 3, 4)`	A	E	S	Q	Y	U	E	S	T	I	O	N
`merge(a, 3, 4, 5)`	A	E	S	Q	U	Y	E	S	T	I	O	N
`merge(a, 0, 2, 5)`	A	E	Q	S	U	Y	E	S	T	I	O	N
`merge(a, 6, 6, 7)`	A	E	Q	S	U	Y	E	S	T	I	O	N
`merge(a, 6, 7, 8)`	A	E	Q	S	U	Y	E	S	T	I	O	N
`merge(a, 9, 9, 10)`	A	E	Q	S	U	Y	E	S	T	I	O	N
`merge(a, 9, 10, 11)`	A	E	Q	S	U	Y	E	S	T	I	N	O
`merge(a, 6, 8, 11)`	A	E	Q	S	U	Y	E	I	N	O	S	T
`merge(a, 0, 5, 11)`	A	E	E	I	N	O	Q	S	S	T	U	Y

Exercise 3. Answer Exercise 2.2.2 for bottom-up mergesort.

Solution.

	a[i]											
	0	1	2	3	4	5	6	7	8	9	10	11
`len = 1`	E	A	S	Y	Q	U	E	S	T	I	O	N
`merge(a, 0, 0, 1)`	A	E	S	Y	Q	U	E	S	T	I	O	N
`merge(a, 2, 2, 3)`	A	E	S	Y	Q	U	E	S	T	I	O	N
`merge(a, 4, 4, 5)`	A	E	S	Y	Q	U	E	S	T	I	O	N
`merge(a, 6, 6, 7)`	A	E	S	Y	Q	U	E	S	T	I	O	N
`merge(a, 8, 8, 9)`	A	E	S	Y	Q	U	E	S	I	T	O	N
`merge(a, 10, 10, 11)`	A	E	S	Y	Q	U	E	S	I	T	N	O
`len = 2`												
`merge(a, 0, 1, 3)`	A	E	S	Y	Q	U	E	S	I	T	N	O
`merge(a, 4, 5, 7)`	A	E	S	Y	E	Q	S	U	I	T	N	O
`merge(a, 8, 9, 11)`	A	E	S	Y	E	Q	S	U	I	N	O	T
`len = 4`												
`merge(a, 0, 3, 7)`	A	E	E	Q	S	S	U	Y	I	N	O	T
`len = 8`												
`merge(a, 0, 8, 11)`	A	E	E	I	N	O	Q	S	S	T	U	Y

Exercise 4. Does the abstract in-place merge produce proper output if and only if the two input subarrays are in sorted order? Prove your answer, or provide a counterexample.

Solution.

Proof. If the arrays are sorted, the algorithm certainly places the result in proper order, as we have seen throughout this chapter.

Suppose that one of the input arrays `a[]` is not in sorted order. Then there is an index `i` such that `a[i] > a[i + 1]`. The algorithm will not increase `i` and add `a[i + 1]` to the result array until `a[i]` is in the result array. That is, `a[i]` will still appear before `a[i + 1]` in the result array, and the result array will still not be properly sorted. □

Exercise 5. Give the sequence of subarray lengths in the merges performed by both the top-down and bottom-up mergesort, for $n = 39$.

Solution. For top-down mergesort, we can build the sequence top-down and then reverse it:

```
a[0..38] // 39
   a[20..38] // 19
      a[30..38] // 9
         a[35..38] // 4
            a[37..38] // 2
            a[35..36] // 2
         a[30..34] // 5
            a[33..34] // 2
            a[30..32] // 3
               a[32..32] // no merge
               a[30..31] // 2
      a[20..29] // 10
         a[25..29] // 5
            a[28..29] // 2
            a[25..27] // 3
               a[27..27] // no merge
               a[25..26] // 2
         a[20..24] // 5
            a[23..24] // 2
            a[20..22] // 3
               a[22..22] // no merge
               a[20..21] // 2
   a[0..19] // 20
      a[10..19] // 10
         a[15..19] // 5
            a[18..19] // 2
            a[15..17] // 3
               a[17..17] // no merge
               a[15..16] // 2
         a[10..14] // 5
            a[13..14] // 2
            a[10..12] // 3
            a[12..12] // no merge
            a[10..11] // 2
      a[0..9] // 10
         a[5..9] // 5
            a[8..9] // 2
            a[5..7] // 3
               a[7..7] // no merge
               a[5..6] // 2
         a[0..4] // 5
            a[3..4] // 2
            a[0..2] // 3
               a[2..2] // no merge
               a[0..1] // 2
```

Therefore, we read the sequence from the bottom to get: 2, 3, 2, 5, 2, 3, 2, 5, 10, 2, 3, 2, 5, 2, 3, 2, 5, 10, 20, 2, 3, 2, 5, 2, 3, 2, 5, 10, 2, 3, 2, 5, 2, 2, 4, 9, 19, 39.

For the bottom-up mergesort, it is much simpler because most sizes are powers of 2 except possibly the last one for a given `len` value:

```
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
4, 4, 4, 4, 4, 4, 4, 4, 4, 3,
8, 8, 8, 8, 7,
16, 16,
32,
39
```

Exercise 6. Write a program to compute the exact value of the number of array accesses used by top-down mergesort and by bottom-up mergesort. Use your program to plot the values of n from 1 to 512, and to compare the exact values with the upper bound $6n \lg n$.

Solution. See the class `com.segarciat.algs4.ch2.sec2.ex06.MergesortPlot`. See Figure 1.

Exercise 7. Show that the number of compares used by mergesort is monotonically increasing, meaning $C(n+1) > C(n)$ for all $n > 0$.

Solution. The result actually depends on the contents of the array. For example, if an array is not sorted with n elements and another array *is* sorted and has $n+1$ elements, then the number of compares is smaller for the second array. Therefore, I do not think this is true in general, and I need more information to understand what conditions makes this true.

Exercise 8. Suppose that Algorithm 2.4 is modified to skip the call on `merge()` whenever `a[mid] <= a[mid+1]`. Prove that the number of compares used to mergesort a sorted array is linear.

Solution.

Proof. With this modification, the algorithm does one compare for each recursive call. Let $k = \lfloor \lg(n) \rfloor$. If i is an integer between 0 and k (inclusive), then i, then i represents

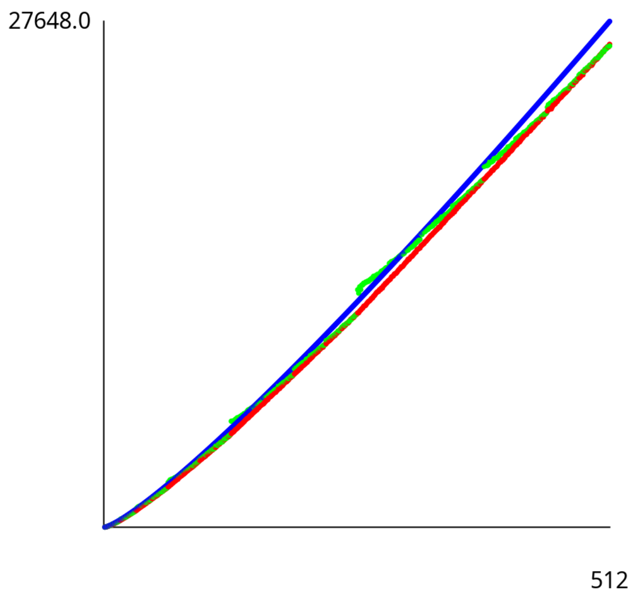

Figure 1: Plot for Exercise 2.2.6; top-down mergesort in red, bottom-up mergesort in green, and the $6n \lg n$ bound in blue

the recursion depth of merge sort. In particular, $i = 0$ is the initial call, and the ith level has 2^i recursive calls. The total number of recursive calls, and hence the total number of compares, is bounded by

$$\begin{aligned}\sum_{i=0}^{k} 2^i &= 2^{k+1} - 1 \\ &= 2 \cdot 2^k - 1 \\ &= 2 \cdot 2^{\lfloor \lg n \rfloor} - 1 \\ &\leq 2 \cdot 2^{\lg n + 1} - 1 \\ &= 4n - 1\end{aligned}$$

Similarly it is bounded below by $\sum_{i=0}^{k-1} 2^i$. We conclude that it is linear. □

Exercise 9. Use of a static array like `aux[]` is inadvisable in library software because multiple clients might use the class concurrently. Give an implementation of `Merge` that does not use a static array. Do *not* make `a[]` local to `merge()` (see the Q & A for this section). *Hint*: Pass the auxiliary array as an argument to the recursive `sort()`.

Solution. See the `com.segarciat.algs4.ch2.sec2.ex09.Merge` class.

Exercise 10. *Faster merge.* Implement a version of `merge()` that copies the second half of `a[]` to `aux[]` in *decreasing order* and then does the merge back to `a[]`. This change allows you to remove the code to test that each of the halves has been exhausted from the inner loop. *Note*: The resulting sort is not stable (see page 341).

Solution. See the `com.segarciat.algs4.ch2.sec2.ex10.FasterMerge` class.

Exercise 11. *Improvements.* Implement the three improvements to mergesort that are described in the text on page 275: Add a cutoff for small subarrays, test whether the array is already in order, and avoid the copy by switching arguments in the recursive code.

Solution. See the `com.segarciat.algs4.ch2.sec3.ex11.MergeImproved` class.

Exercise 12. *Sublinear extra space.* Develop a merge implementation that reduces the extra space requirement to $\max(n/m)$, based on the following idea: Divide the array into n/m blocks of size m (for simplicity in this description, assume that n is a multiple of m). Then,

(i) Considering the blocks as items with their first key as the sort key, sort them using selection sort; and

(ii) Run through the array merging the first block with the second, then the second with the third, and so forth.

Solution. TODO.

Exercise 13. *Lower bound for average case.* Prove that the expected number of compares used by any compared-based sorting algorithm must be at least $n \lg n$ (assuming that all possible orderings of the input are equally likely). *Hint*: The expected number of compares is at least the external path length of the compare tree (the sum of the lengths of the paths from the root to all leaves), which is minimized when it is balanced.

Solution. TODO.

Exercise 14. *Merging sorted queues.* Develop a static method that takes two queues of sorted items as arguments and returns a queue that results from merging the the queues into sorted order.

Solution. See the `com.segarciat.algs4.ch2.sec2.ex14.MergeQueues` class.

Exercise 15. *Bottom-up queue mergesort.* Develop a bottom-up mergesort implementation based on the following approach: Given n items, create n queues, each containing one of the items. Create a queue of the n queues. Then, repeatedly apply the merging operation of Exercise 2.2.14 to the first two queues and reinsert the merged queue at the end. Repeat until the queue of queues contains only one queue.

Solution. See the`com.segarciat.algs4.ch2.sec2.ex15.MergeBUQueue` class.

Exercise 16. *Natural mergesort.* Write a version of bottom-up mergesort that takes advantage of order in the array by proceeding as follows each time it needs to find two arrays to merge: find a sorted subarray (by incrementing a pointer until finding an entry that is smaller than its predecessor in the array), then find the next, then merge them. Analyze the running time of this algorithm in terms of the array length and the number of maximal increasing sequences in the array.

Solution. See the `com.segarciat.algs4.ch2.sec2.ex16.NaturalMergeBU` class. Let n be the length of the array we are sorting. In the worst case, there's n increasing sequences, because the array may be sorted in reverse with distinct elements. In that case, there are $n-1$ merges, in each with an array of size 1 for the second array. Because the `merge()` operation is linear in the size of the input arrays (given that it must copy all elements), the performance degrades to quadratic in this case. For example, it would require copying 1 element, then 2, then 3, and so on until the last merge where it has to copy $n-1$ elements. In general, if there are k maximal increasing sequences in the array, then each pass through the array decreases the number of maximal increasing sequences by 1, and the algorithm stops when only one remains; at that point, the array is sorted. The cost is then k merges.

Exercise 17. *Linked-list sort.* Implement a natural mergesort for linked lists. (This is the method of choice for sorting linked lists because it uses no extra space and is guaranteed to be linearithmic).

Solution. See the `com.segarciat.algs4.ch2.sec2.ex17.LinkedListNaturalMergesort` class.

Exercise 19. *Inversions.* Develop and implement a linearithmic algorithm for computing the number of inversions in a given array (the number of exchanges that would be performed by insertion sort for that array — see Section 2.1). This quantity is related to the *Kendall tau distance*; see Section 2.5.

Solution. See the `com.segarciat.algs4.ch2.sec2.ex19.Inversions` class.

Exercise 20. *Index sort.* Develop and implement a version of mergesort that does not rearrange the array, but returns an `int[]` array 'perm' such that `perm[i]` is the index of the ith smallest entry in the array.

Solution. See the `com.segarciat.algs4.ch2.sec2.ex20.IndexSort` class.

Exercise 21. *Triplicates.* Given three lists of n names each, devise a linearithmic algorithm to determine if there is a name common to all three lists,and if so,return the lexigraphically first such name.

Solution. See the `com.segarciat.algs4.ch2.sec2.ex21.Triplicates` class.

Exercise 22. *3-way mergesort.* Suppose instead of dividing in half at each step, you divide into thirds,s ort each third, and combine using a 3-way merge. What is the order of growth of the overall running time of this algorithm?

Solution. If we make a tree of the recursive calls implied by this algorithm, rooted at the initial call, then we get a trinary three, and the height of the tree is about $\log_3(n)$, where n is the number of elements in the array. At each recursive call when having a size of $N \le n$, the merge takes up at most N comparisons. We could write a recursive formula similar to the one for (binary) mergesort:

$$C(n) \le C(\lceil n/3 \rceil) + C(\lceil n/3 \rceil) + C(\lfloor n/3 \rfloor) + n$$

On the right-hand side, the first two are upper bounds on the cost for sorting the first two thirds, the third term is the cost of compares for sorting the last third, and an upper bound for the n is the cost of the merge. I conjecture (but will not prove) that the resulting algorithm is thus linearithmic, because it will likely use $\sim n \log_3 n$ compares.

Exercise 23. *Improvements.* Run empirical studies to evaluate the effectiveness of each of the three improvements to mergesort that are described in the text (see Exercise 2.2.11). Also, compare the performance of the merge implementation given in the text with the merge described in Exercise 2.2.10. In particular, empirically determine the best value of the parameter that decides when to switch to insertion sort for small subarrays.

Solution. See `com.segarciat.algs4.ch2.sec2.ex23`. In summary, all of these improvements are observed, with the least prominent being the improvement that skips the merge for arrays that are already sorted, though in that case, it still performs around linearithmically. Of course, the added bonus is that it is linear for sorted arrays. For the insertion cutoff value, I found that on my system I get an improvement of about 200% for a `CUTOFF` of about 63. My implementation`MergeImproved` from Exercise 2.2.11 that combines all three improvements leads to a performance improvement of about 300%, or so I observed..

Exercise 24. *Sort-test improvement.* Run empirical studies for large randomly ordered arrays to study the effectiveness of the modification described in Exercise 2.2.8 for random data. In particular, develop a hypothesis about the average number of times the test (whether an array is sorted) succeeds, as function of n (the original array length for the sort).

Solution. See `com.segarciat.algs4.ch2.sec2.ex23.MergeOrderCheck`. It uses the same class from Exercise 23 but adds a static variable that keeps track of the number of times the test passes. I observed that the ratio of test successes to the array size n was about $0.29n$ for large n by using a doubling test.

References

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. 4th ed. Addison-Wesley, 2011. ISBN: 9780321573513.