Sorting --- Merge Sort

Steven J. Zeil

Last modified: Jun 11, 2014

Contents:

Our next algorithm actually achieves the optimal big-O behavior for a sorting algorithm. The merge sort has $O(n \log n)$ time for both its worst and average case.

This doesn't necessarily make it the ideal choice, however, in all sorting applications. The constant multiplier on the timing is somewhat high, and merge sort may require an unusually high amount of memory.

Variants of the basic merge sort algorithm are, however, often used with linked lists (which can't be sorted by most other $O(n \log n)$ algorithms and are used to sort data residing on disk or magnetic tape.

1. Merging Sorted Data

Before tackling the merge sort itself, we start with a simpler function that is used by merge sort.

Suppose that our sequence of data can be divided into two parts, such that `a[first..mid-1]` is already sorted and `a[mid..last-1]` is already sorted. Then we could merge the two parts into a combined sorted sequence using the code shown here.

```
template <typename T>
void merge(vector<T>& v, int first, int mid, int last)
{
  // temporary vector to merge the sorted sublists
  vector<T> tempVector;
  int indexA, indexB, indexV;

  // set indexA to scan sublistA (index range [first,mid)
  // and indexB to scan sublistB (index range [mid, last)
  indexA = first;
  indexB = mid;

  // while both sublists are not exhausted, compare v[indexA] and
  // v[indexB]copy the smaller to vector temp using push_back()
  while (indexA < mid && indexB < last)  <:\co1:>
    if (v[indexA] < v[indexB])
      {
        tempVector.push_back(v[indexA]);    // copy element to temp
        indexA++;                           // increment indexA
```

```
      }
    else
      {
        tempVector.push_back(v[indexB]);    // copy element to temp
        indexB++;                           // increment indexB
      }

  // copy the tail of the sublist that is not exhausted
  while (indexA < mid)                  <:\co2:>
    {
      tempVector.push_back(v[indexA]);
      indexA++;
    }

  while (indexB < last)
    {
      tempVector.push_back(v[indexB]);
      indexB++;
    }

  // copy vector tempVector using indexV to vector v using indexA
  // which is initially set to first
  indexA = first;

  // copy elements from temporary vector to original list
  for (indexV = 0; indexV < tempVector.size(); indexV++)  <:\co3:>
    {
      v[indexA] = tempVector [indexV];
      indexA++;
    }
}
```

1.1 Understanding the Merge Algorithm

The heart of the merge algorithm is the first loop (\co1).

The variables *first*, *mid*, and *last* mark off two subsequences that we want to merge. We can think of `a[first ... mid-1]` and `a[mid ... last-1]` as two separate, sorted sequences. We want to combine them into a single sorted sequence, *tempVector*.

The way to do this is quite simple. Just compare the first element in each of the two input (sub)sequences and copy the smaller one.

For example, if we were merging subsequences [2 \; 4 \; 5 \; 6] and [1 \; 3] we would compare the first element in each one (2 and 1) and decide to copy 1.

Then we continue with the remainder, merging [2 \; 4 \; 5 \; 6] and [3]

On the next step we would copy 2, and be left with the merge of [4 \; 5 \; 6] and [3]

We would then copy 3.

At this point, our temporary vector contains [1 \; 2 \; 3] We would now exit from this main loop, because one of the arrays has been completely emptied out.

The rest of the algorithm is “cleanup”. We exit the main loop when we have emptied one of the two subsequences, so there is a possibility that the other subsequence still has data. The next two loops (\co2) copy that data from the remainder of the two subsequences. (Because one of those subsequences has been emptied, one of these loops will execute zero times.)

Finally (\co3), we copy the entire merged data set back out of the temporary vector into the original vector.

Run this algorithm until you are comfortable with your understanding of how it works.

1.2 Merge Analysis

merge1.cpp +

```
template <typename T>
void merge(vector<T>& v, int first, int mid, int last)
{
  // temporary vector to merge the sorted sublists
  vector<T> tempVector;
  int indexA, indexB, indexV;

  // set indexA to scan sublistA (index range [first,mid)
  // and indexB to scan sublistB (index range [mid, last)
  indexA = first;
  indexB = mid;

  // while both sublists are not exhausted, compare v[indexA] and
  // v[indexB]copy the smaller to vector temp using push_back()
  while (indexA < mid && indexB < last)
    if (v[indexA] < v[indexB])
      {
        tempVector.push_back(v[indexA]);    // O(tempVector.size())
        indexA++;                           // O(1)
      }
    else
      {
        tempVector.push_back(v[indexB]);    // O(tempVector.size())
        indexB++;                           // O(1)
      }

  // copy the tail of the sublist that is not exhausted
  while (indexA < mid)
    {
      tempVector.push_back(v[indexA]);      // O(tempVector.size())
      indexA++;                             // O(1)
    }

  while (indexB < last)
    {
      tempVector.push_back(v[indexB]);      // O(tempVector.size())
      indexB++;                             // O(1)
    }

  // copy vector tempVector using indexV to vector v using indexA
  // which is initially set to first
  indexA = first;

  // copy elements from temporary vector to original list
  for (indexV = 0; indexV < tempVector.size(); indexV++)
    {
      v[indexA] = tempVector [indexV];     // O(1)
      indexA++;                            // O(1)
    }
}
```

There are several vector push_back calls, which have a worst-case behavior proportional to the size of the vector. However, a little time looking at them shows that they are all on *tempVector*, which is initially empty.

So this falls into the special case pattern of filling an initially empty vector with repeated push_backs, in which case those pushes will amortize to $O(1)$.

This means that all the loop bodies amortize to $O(1)$.

Looking at the code for the first 3 loops, note that

- each one adds one element into *tempVector*.
- no element is copied multiple times. If we copy the element at *indexA*, we also increment *indexA*, so we will not copy that element again. Similarly, if we copy the element at *indexB*, we also increment *indexB*, so we will not copy that element again.

Since there are a total of `last-first` elements, each loop can repeat no more than last-first times.

In fact, the *sum* of the number of iterations of *all three loops* is `last-first`.

So all three loops are $O(\text{last} - \text{first})$.

merge2.cpp +

```
template <typename T>
void merge(vector<T>& v, int first, int mid, int last)
{
  // temporary vector to merge the sorted sublists
  vector<T> tempVector;
  int indexA, indexB, indexV;

  // set indexA to scan sublistA (index range [first,mid)
  // and indexB to scan sublistB (index range [mid, last)
  indexA = first;
  indexB = mid;

  // while both sublists are not exhausted, compare v[indexA] and
  // v[indexB]copy the smaller to vector temp using push_back()
  while (indexA < mid && indexB < last)  // =O(last-first)
    // O(1)

  // copy the tail of the sublist that is not exhausted
  while (indexA < mid)                   // =O(last-first)
    {
      // O(1)
    }

  while (indexB < last)                  // =O(last-first)
    {
      // O(1)
    }

  // copy vector tempVector using indexV to vector v using indexA
  // which is initially set to first
  indexA = first;

  // copy elements from temporary vector to original list
  for (indexV = 0; indexV < tempVector.size(); indexV++) // O(1) (last-first)*
    {
      // O(1)
    }
}
```

The last loop clearly repeats once for each element in *tempVector*. But we have just determined that the total number of elements in *tempVector* will be last-first.

merge3.cpp +

```
template <typename T>
void merge(vector<T>& v, int first, int mid, int last)
{
  // temporary vector to merge the sorted sublists
  vector<T> tempVector;
  int indexA, indexB, indexV;

  // set indexA to scan sublistA (index range [first,mid)
  // and indexB to scan sublistB (index range [mid, last)
  indexA = first;
  indexB = mid;

  // O(last-first)
  // O(last-first)
  // O(last-first)

  // copy vector tempVector using indexV to vector v using indexA
  // which is initially set to first
  indexA = first;

  // copy elements from temporary vector to original list
  // O(last-first)
}
```

That leaves only a handful of O(1) statements that will all be dominated by the complexity of the loops, so

- merge is $O(\text{last} - \text{first})$.

2. Merge Sort

The merge function lets us combine two sorted sequences of data into a single sorted sequence. But how do we get the two sorted sequences in the first place? By merge'ing two even smaller sorted sequences!

2.1 The Algorithm

```
template <typename T>
void mergeSort(vector<T>& v, int first, int last)
{
  // if the sublist has more than 1 element continue
  if (first + 1 < last)
    {
      // for sublists of size 2 or more, call mergeSort()
      // for the left and right sublists and then
      // merge the sorted sublists using merge()
      int midpt = (last + first) / 2;

      mergeSort(v, first, midpt);
      mergeSort(v, midpt, last);
      merge(v, first, midpt, last);
    }
}
```

The algorithm shown here is the actual sorting algorithm. It is almost amazingly simple, consisting simply of two recursive calls to itself, each attempting to sort half the vector, followed by a call to merge to combine the two sorted halves into a single sorted sequence.

For many people, the very simplicity of this algorithm makes it hard to believe that it can work. I therefore recommend strongly that you run this algorithm until you are comfortable with your understanding of it.

2.2 MergeSort Analysis

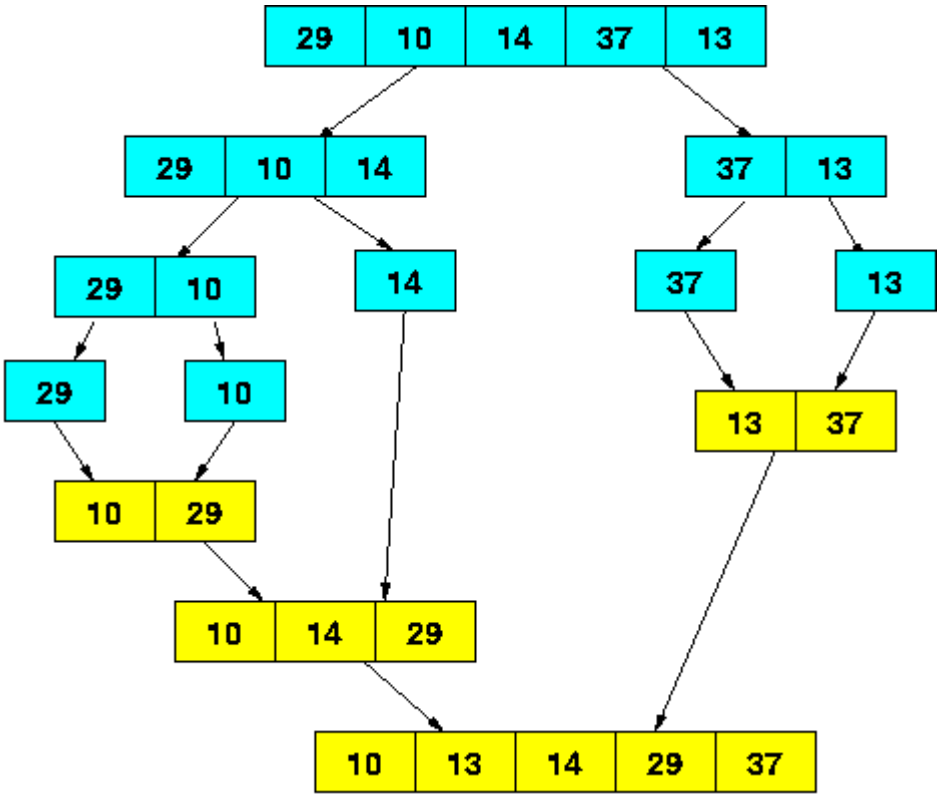
```
template <typename T>
void mergeSort(vector<T>& v, int first, int last)
{
  // if the sublist has more than 1 element continue
  if (first + 1 < last)
    {
      // for sublists of size 2 or more, call mergeSort()
      // for the left and right sublists and then
      // merge the sorted sublists using merge()
      int midpt = (last + first) / 2;

      mergeSort(v, first, midpt);
      mergeSort(v, midpt, last);
      merge(v, first, midpt, last);
    }
}
```

- Each call to mergeSort is either done in O(1) time (if `first+1 \ensuremath{\geq` last}) or splits the array into two equal (\ensuremath{\pm}1) pieces. We can do this split up to log N times.

We can envision the recursive mergeSort calls (in blue) and the subsequent calls to merge (in yellow) as a tree-like structure.

Let N denote the total number of elements being sorted (the value of `last-first` on the very first call to mergeSort). \ePicOnRight



- Each level in the tree involves no more than N objects, split in various ways and needing to be merged.
- merge is $O(k)$, where k is the number of elements to be merged. The sum of all the k values at any level of the yellow tree is N. Consequently the combined set of merges at each level of the tree is $O(N)$.
- The blue tree represents all the non-merge work in mergeSort. But there's only $O(1)$ work in each of those blue nodes. Since the most blue nodes we could have at one level is N, each blue level is, at most, $O(N)$ total work.

- Because we have $\log N$ levels, each level taking $O(N)$ work, the overall merge sort code is (worst & average case) $O(N \log N)$.

So merge sort is as fast as any pairwise-comparison sort can be. Still, merge sort is not considered to be the "ideal" sorting algorithm. Its primary drawbacks are

- It requires $O(N)$ extra storage (for the *tempVector*)
- It does the full set of comparisons and copies even when applied to arrays that are already sorted.

On the other hand, merge sort has an advantage that may, at first glance, not have seemed very important. The merge routine itself moves sequentially through its working arrays, not jumping from place to place. This behavior would be absolutely wonderful if we were storing our arrays in some strange kind of memory where moving forward one place is cheap, but jumping to an arbitrary position is expensive.

In fact, that "strange kind of memory" does exist: disk drives and magnetic tape both meet that description. Hence variations of merge sort have long been the algorithm of choice in *external sorting*, sorting sets of material stored in disk/tape files that are too large to load into memory.