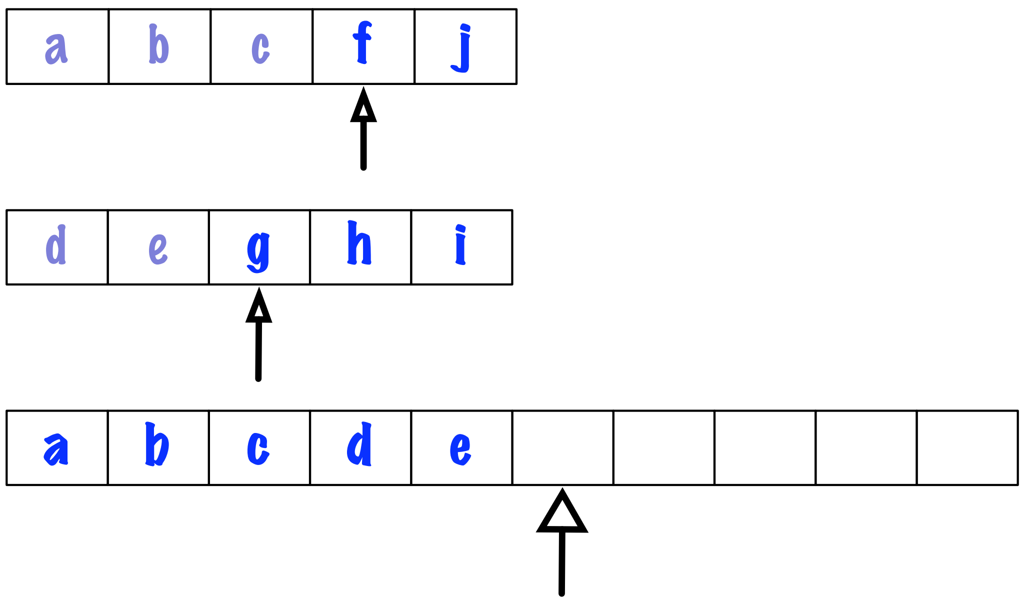


Figure 1: The merging algorithm

Algorith Analysis and Development
Fall 2007

These notes are intended for 2007-10-25. They introduce various concepts related to sorting and explain merge sorting using `vectors`.

1 The Merge Sort Algorithm

The heart of the merge-sort algorithm is *merging*. If we have two sorted collections of sizes m and n we combine them into one sorted collection in time $O(m+n)$. With lists we can do this merging by doing list surgery without extra storage. In general we need extra storage of size $O(m+n)$.

To merge two sorted collections into one we need essentially three pointers. The first two point at the beginning of the data that remains to be merged, and the third at the remaining space into which the merge is being done. This is illustrated in Figure 1.

The merge sort algorithm now works by divide-and-conquer. First we check to see if we have an array of size less than two. If we do, it is certainly sorted and we can stop.

If it is not,

- split the array into two.
- Mege sort the first half (recursively).
- Mege sort the second half (recursively).
- Merge the two halves.

Managing the memory involved is a little bit tricky. In general, we have a target where we want the merged and sorted result to show up, and we have a region of temporary memory that we can use for intermediate steps. We can refine our algorithm above to be:

If we are merge-sorting a chunk of storage with two or more elements and we want the data to end up in **`target`** and we have spare storage in **`temp`** we need to:

- Mege sort the first half (recursively) to end up in **`temp`** using **`target`** as temporary stoage.
- Mege sort the second half (recursively) to end up in the second half of **`temp`** using the second half of **`target`** as temporary storage.
- Merge the two halves of **`temp`** back into **`target`**.

2 Templated Merge Sort Code

We know convert the ideas above into templated code in the STL style using iterators wherever we can.

2.1 The merge algorithm

Generically we can merge two different kinds of input streams into one output stream. This is expressed in the algorithm shown in Figure 2.

Notes

```
template <typename Iterator1,
          typename Iterator2,
          typename Iterator3>
Iterator3
mergeRanges(Iterator1 begin1,
            Iterator1 end1,
            Iterator2 begin2,
            Iterator2 end2,
            Iterator3 destination)
{
    while (begin1!=end1 && begin2!=end2)
        {
        if (*begin2 < *begin1)
            *destination++ = *begin2++ ;
        else
            *destination++ = *begin1++ ;
        }
    while (begin1!=end1)
            *destination++ = *begin1++ ;
    while (begin2!=end2)
        *destination++ = *begin2++ ;
    return destination ;
}
```

Figure 2: The Merge algorithm

lines 15-20 Note that we prefer to take items from the first sequence. That is, if items in the two sequences are equal take from the first sequence.

lines 15-20 Also note that we can write the loop body with flatter looking logic as:

```
bool useFirst = ! (*begin2 < *begin1) ;
*destination++ = * (useFirst ? begin1 : begin2)++ ;
```

It is arguable whether this is any better than the code given. Note that we are trying to do all comparison in terms of "`<`", as this is what the STL does.

line 21 Note that when we reach line 21 we know that either `begin1==end1` or `begin2==end2`, so at least one of the following `while` loops does nothing, and the other might copy the tail of one sequence.

2.2 The `mergeSortTo` algorithm

The `mergeSortTo` algorithm is shown in Figure 3. Before beginning to understand the internal details of the algorithm, it is important to understand what it claims to do. The claim is that the data from **`begin`** to **`end`** is sorted to a (possibly) new space that begins at **`to`**. The algorithm returns an iterator that points one past the end of the range where the sorted data has been put. The iterator **`temp`** points a chunk of storage of size at least `end-begin` that can be used as temporary storage.

This has the usual structure of a recursive algorithm. First we test for the base cases. Here we can't hope to split the problem into smaller problem if original problem has size two, so the size zero and size one cases are our base cases.

Lines 42–46 handle the base case of size zero. Note that there is nothing to do here.

Lines 47–51 handle the base case of size one. We haven't yet described the `middleOf` algorithm, but it attempts to set **`middle`** to be half way between **`begin`** and **`end`**. It only equals **`begin`** when `end-begin<2`, which justifies the claim that we are in the size one base case here. Here we need to copy the data item to the output range.

```
template <typename Iterator>
Iterator
mergeSortTo(Iterator begin, Iterator end, Iterator to, Iterator temp)
{
    Iterator middle=middleOf(begin,end) ;
    if (begin==end)
        {
        // base case: range is size 0
        return to ;
        }
    else if (begin==middle)
        {
        // base case: range is size 1
        *to++ = *begin ;
        return to ;
        }
    else
        {
        // we have range of at least size 2.  recursion kicks in
        Iterator temp2 = mergeSortTo(begin, middle, temp, to) ;
        Iterator temp3 = mergeSortTo(middle, end, temp2, to) ;
        return mergeRanges(temp, temp2, temp2, temp3, to) ;
        }
}
```

Figure 3: the **`mergeSortTo`** algorithm

```
template <typename Iterator1>
Iterator1 middleOf(Iterator1 b, Iterator1 e)
{
    Iterator1 middle(b) ;
    std::advance(middle, std::distance(b,e)/2) ;
    return middle ;
}
```

Figure 4: the **`middleOf`** function

```
template <typename Iterator>
void mergeSort(Iterator begin, Iterator end)
{
    typedef typename std::iterator_traits<Iterator>::value_type elt_t ;
    std::vector<elt_t> temp(begin, end) ;
    mergeSortTo(begin, end, begin, temp.begin()) ;
    return ;
}
```

Figure 5: the **`mergeSort`** code

The recursive case in lines 53–61 is the most complicated. The main trick is to reverse the roles of `temp` and `to` in the recursive calls so that the data is sorted to `temp`. We use the iterators `temp2` and `temp3` generated by the recursive calls to capture the sorted ranges that we wish to merge. We then merge those ranges to `to` and return a pointer to one beyond the region of `to` that we filled.

2.3 The `middleOf` function

Figure 4 shows the coding of `middleOf`. The `advance` and `distance` functions are from the STL. For iterators that are random access the code becomes `middle += (e-b)/2`. Note that we *cannot* write `middle = (e+b)/2;`, even with pointers.

2.4 A driver algorithm

It may not be clear how to get the recursive `mergeSortTo` algorithm started. We show how to do this in Figure 5. The essence of this algorithm is to create a vector for temporary storage, which is done on line 23. All of the STL containers have this sort of two iterator argument constructor, which creates a copy of the range described by the iterators.

Line 21 appears quite tricky. Its purpose is to define **`elt_t`** to be the type of the things that are stored in the range defined by [`begin`,`end`). Because iterators might just be pointers, there is no direct way to ask **`Iterator`** what it points to. However, the STL `<iterator>` header provides a templated class `std::iterator<...>` that does the right thing. It, in turn, contains a number of `typedef`s that the programmer can use, including `value_type`, which is the type of things that its template argument points at.

The other tricky feature of line 21 is the presence of the `typename`. This is required, because without it we cannot tell at the time that we are reading the source code whether `std::iterator_traits<Iterator>::value_type` is

1. a public member typename, or
2. a publici member variable.

The compiler always follows the rule that in circumstances like this, if it cannot tell, it assumes that the thing in question is a value, not a type. To tell the compiler otherwise, we insert the word `typename` just before the expression.

3 Testing

In this section we present some simple code for testing our algorithm. The code shown in Figure 6 gives a simple true-false `test` function that tests whether or not the algorithm correctly sorts a randomly shuffled vector of integers from `0` to `i`.

Finally, Figure 7 shows a simple way to call the `test`-function.

```
bool test(int i)
{
    assert(i>=0) ;
    std::vector<int> v ;
    for(int j=0;j<i;++j)
        {
        v.push_back(j) ;
        }
    std::random_shuffle(v.begin(), v.end()) ;
    mergeSort(v.begin(), v.end()) ;
    return isSorted(v.begin(), v.end()) ;
}
```

Figure 6: the **`test`** function

```
int main()
{
    unsigned long seed(time(0)) ;
    srand(seed) ;
    srand48(seed) ;
    std::cout << "seed is " << seed
              << "; first rand() is " << rand()
              << "; first lrand48() is " << lrand48()
              << "." << std::endl ;


    bool everythingsOK = test(0) && test(1) && test(2) && test(10000) ;
    std::cout << (everythingsOK ? "tests pass." : "tests fail.")
              << std::endl ;
    return (everythingsOK ? 0 : 100) ;
}
```

Figure 7: the **`main`** function

Worst-Case Time:	$\Theta(n \log n)$
Average-Case Time:	$\Theta(n \log n)$
Extra Storage:	$\Theta(n)$
Stable:	yes
Other characteristics:	Works very well with lists, where extra storage is not needed.

Figure 8: the Merge Sort Report Card

4 Stable Sorting

One subtle but important question that one can ask about a sorting algorithm is whether or not it keeps equal elements in their original order. If it does we say that the algorithm is *stable*. This is illustrated in Figures 9–11. Figure 11 is not a stable sorting of the data in Figure 9 because the two "**1**"s have swapped order. Stable sorting is important because it gives a way of combining multiple sorts.

Merge-sort is the only fast sorting algorithm that we shall study that is inheritently stable. It is stable because merging can be made stable, and the base cases of merge sort are clearly stable.

5 The Merge Sort Report Card

We shall be studying various different sorting algorithms. In order to compare them we shall give them each a report card. The report card for merge sort is shown in Figure 8.

Note the list of questions answered by the report card. We shall ask these questions about each of the other sorting algorithms too.

Here are some justifications of the data given. The stability of merge sort is discussed above. The extra storage is obvious from the algorithm.

The timings are perhaps the most complex. However, when you look at both the **`mergeSortTo`** and the **`mergeRanges`** algorithms, it should be clear that the overall running time does not depend on the data, only on the amount of it. (**`mergeRanges`** runs a little bit faster when the two

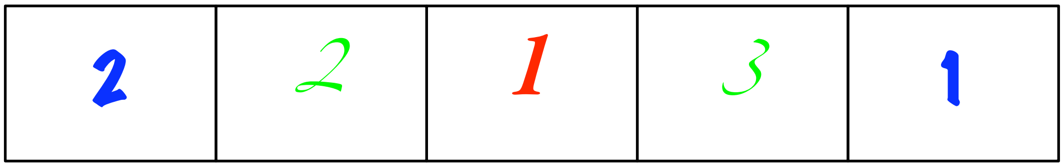


Figure 9: Initial data.

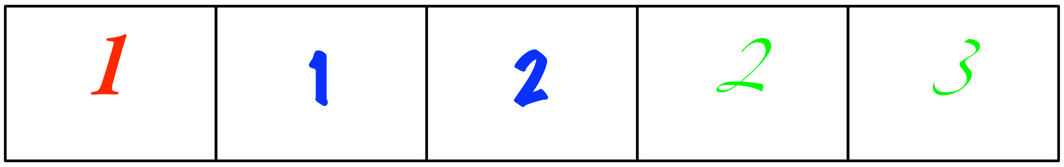
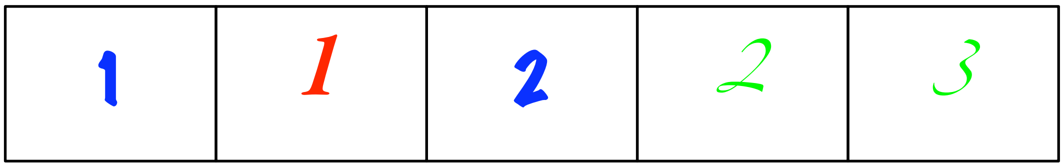


Figure 10: Data from Figure 9 stably-sorted.



Note that the order of the two "1"s has been reversed.

Figure 11: Data from Figure 9 unstably-sorted.

ranges do not overlap, but the overall running time is still proportional to the sum of the sizes.)

The recursive **`mergeSortTo`** function has a running time that looks very much like that of the third subsequence sum algorithm. Reading the code from top to bottom, we get

$$T(n) = \begin{cases} c_0 & \text{if } n = 0, \\ c_1 & \text{if } n = 1, \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c_2 n & \text{otherwise.} \end{cases} \tag{1}$$

By the same kinds of telescoping sum arguments as before we $T(n) = \Theta(n \log n)$. Alternately, we can ask each element of the array what it sees; and the answer is that it sees a merge of total size 2, …, $n/2$, n, that is $\approx \log_2 n$ different merges. As each element takes $\Theta(1)$ work per merge, this gives $\Theta(1 \times n \times \log_2 n)$ running time.