Lecture 6: Divide and Conquer and MergeSort

(Thursday, Feb 12, 1998)

Read: Chapt. 1 (on MergeSort) and Chapt. 4 (on recurrences).

Divide and Conquer: The ancient Roman politicians understood an important principle of good algorithm design (although they were probably not thinking about algorithms at the time). You divide your enemies (by getting them to distrust each other) and then conquer them piece by piece. This is called *divide-and-conquer*. In algorithm design, the idea is to take a problem on a large input, break the input into smaller pieces, solve the problem on each of the small pieces, and then combine the piecewise solutions into a global solution. But once you have broken the problem into pieces, how do you solve these pieces? The answer is to apply divide-and-conquer to them, thus further breaking them down. The process ends when you are left with such tiny pieces remaining (e.g. one or two items) that it is trivial to solve them.

Summarizing, the main elements to a divide-and-conquer solution are

- Divide (the problem into a small number of pieces),
- Conquer (solve each piece, by applying divide-and-conquer recursively to it), and
- Combine (the pieces together into a global solution).

There are a huge number computational problems that can be solved efficiently using divide-and-conquer. In fact the technique is so powerful, that when someone first suggests a problem to me, the first question I usually ask (after what is the brute-force solution) is "does there exist a divide-and-conquer solution for this problem?"

Divide-and-conquer algorithms are typically recursive, since the conquer part involves invoking the same technique on a smaller subproblem. Analyzing the running times of recursive programs is rather tricky, but we will show that there is an elegant mathematical concept, called a *recurrence*, which is useful for analyzing the sort of recursive programs that naturally arise in divide-and-conquer solutions. For the next couple of lectures we will discuss some examples of divide-and-conquer algorithms, and how to analyze them using recurrences.

MergeSort: The first example of a divide-and-conquer algorithm which we will consider is perhaps the best known. This is a simple and very efficient algorithm for sorting a list of numbers, called *MergeSort*. We are given an sequence of n numbers A, which we will assume is stored in an array $A[1 \ldots n]$. The objective is to output a permutation of this sequence, sorted in increasing order. This is normally done by permuting the elements within the array A.

How can we apply divide-and-conquer to sorting? Here are the major elements of the MergeSort algorithm.

Divide: Split A down the middle into two subsequences, each of size roughly $n/2$.

Conquer: Sort each subsequence (by calling MergeSort recursively on each).

Combine: Merge the two sorted subsequences into a single sorted list.

The dividing process ends when we have split the subsequences down to a single item. An sequence of length one is trivially sorted. The key operation where all the work is done is in the combine stage, which merges together two sorted lists into a single sorted list. It turns out that the merging process is quite easy to implement.

The following figure gives a high-level view of the algorithm. The "divide" phase is shown on the left. It works top-down splitting up the list into smaller sublists. The "conquer and combine" phases are shown on the right. They work bottom-up, merging sorted lists together into larger sorted lists.

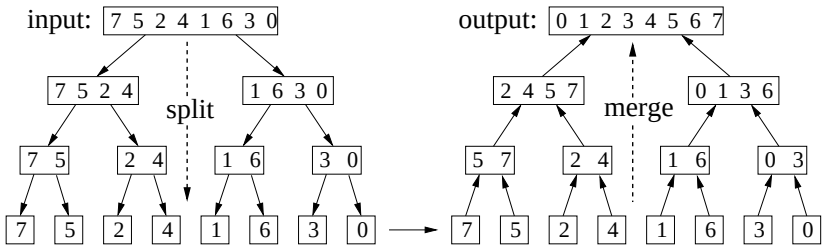

Figure 4: MergeSort.

MergeSort: Let's design the algorithm top-down. We'll assume that the procedure that merges two sorted list is available to us. We'll implement it later. Because the algorithm is called recursively on sublists, in addition to passing in the array itself, we will pass in two indices, which indicate the first and last indices of the subarray that we are to sort. The call `MergeSort(A, p, r)` will sort the subarray $A[p..r]$ and return the sorted result in the same subarray.

Here is the overview. If $r = p$, then this means that there is only one element to sort, and we may return immediately. Otherwise (if $p < r$) there are at least two elements, and we will invoke the divide-and-conquer. We find the index q, midway between p and r, namely $q = (p + r)/2$ (rounded down to the nearest integer). Then we split the array into subarrays $A[p..q]$ and $A[q + 1..r]$. (We need to be careful here. Why would it be wrong to do $A[p..q - 1]$ and $A[q..r]$? Suppose $r = p + 1$.) Call MergeSort recursively to sort each subarray. Finally, we invoke a procedure (which we have yet to write) which merges these two subarrays into a single sorted array.

MergeSort

```
MergeSort(array A, int p, int r) {
    if (p < r) {                          // we have at least 2 items
        q = (p + r)/2
        MergeSort(A, p, q)                // sort A[p..q]
        MergeSort(A, q+1, r)              // sort A[q+1..r]
        Merge(A, p, q, r)                 // merge everything together
    }
}
```

Merging: All that is left is to describe the procedure that merges two sorted lists. `Merge(A, p, q, r)` assumes that the left subarray, $A[p..q]$, and the right subarray, $A[q + 1..r]$, have already been sorted. We merge these two subarrays by copying the elements to a temporary working array called B. For convenience, we will assume that the array B has the same index range A, that is, $B[p..r]$. (One nice thing about pseudocode, is that we can make these assumptions, and leave them up to the programmer to figure out how to implement it.) We have to indices i and j, that point to the current elements of each subarray. We move the smaller element into the next position of B (indicated by index k) and then increment the corresponding index (either i or j). When we run out of elements in one array, then we just copy the rest of the other array into B. Finally, we copy the entire contents of B back into A. (The use of the temporary array is a bit unpleasant, but this is impossible to overcome entirely. It is one of the shortcomings of MergeSort, compared to some of the other efficient sorting algorithms.)

In case you are not aware of C notation, the operator `i++` returns the current value of i, and then increments this variable by one.

Merge

```
Merge(array A, int p, int q, int r) {       // merges A[p..q] with A[q+1..r]
```

```
    array B[p..r]
    i = k = p                                  // initialize pointers
    j = q+1
    while (i <= q and j <= r) {                // while both subarrays are nonempty
        if (A[i] <= A[j]) B[k++] = A[i++]      // copy from left subarray
        else              B[k++] = A[j++]      // copy from right subarray
    }
    while (i <= q) B[k++] = A[i++]             // copy any leftover to B
    while (j <= r) B[k++] = A[j++]
    for i = p to r do A[i] = B[i]              // copy B back to A
}
```

This completes the description of the algorithm. Observe that of the last two while-loops in the Merge procedure, only one will be executed. (Do you see why?)

If you find the recursion to be a bit confusing. Go back and look at the earlier figure. Convince yourself that as you unravel the recursion you are essentially walking through the tree (the *recursion tree*) shown in the figure. As calls are made you walk down towards the leaves, and as you return you are walking up towards the root. (We have drawn two trees in the figure, but this is just to make the distinction between the inputs and outputs clearer.)

Discussion: One of the little tricks in improving the running time of this algorithm is to avoid the constant copying from A to B and back to A. This is often handled in the implementation by using two arrays, both of equal size. At odd levels of the recursion we merge from subarrays of A to a subarray of B. At even levels we merge from from B to A. If the recursion has an odd number of levels, we may have to do one final copy from B back to A, but this is faster than having to do it at every level. Of course, this only improves the constant factors; it does not change the asymptotic running time.

Another implementation trick to speed things by a constant factor is that rather than driving the divide-and-conquer all the way down to subsequences of size 1, instead stop the dividing process when the sequence sizes fall below constant, e.g. 20. Then invoke a simple $\Theta(n^2)$ algorithm, like insertion sort on these small lists. Often brute force algorithms run faster on small subsequences, because they do not have the added overhead of recursion. Note that since they are running on subsequences of size at most 20, the running times is $\Theta(20^2) = \Theta(1)$. Thus, this will not affect the overall asymptotic running time.

It might seem at first glance that it should be possible to merge the lists "in-place", without the need for additional temporary storage. The answer is that it is, but it no one knows how to do it without destroying the algorithm's efficiency. It turns out that there are faster ways to sort numbers in-place, e.g. using either HeapSort or QuickSort.

Here is a subtle but interesting point to make regarding this sorting algorithm. Suppose that in the if-statement above, we have $A[i] = A[j]$. Observe that in this case we copy from the left sublist. Would it have mattered if instead we had copied from the right sublist? The simple answer is no—since the elements are equal, they can appear in either order in the final sublist. However there is a subtler reason to prefer this particular choice. Many times we are sorting data that does not have a single attribute, but has many attributes (name, SSN, grade, etc.) Often the list may already have been sorted on one attribute (say, name). If we sort on a second attribute (say, grade), then it would be nice if people with same grade are still sorted by name. A sorting algorithm that has the property that equal items will appear in the final sorted list in the same relative order that they appeared in the initial input is called a *stable sorting algorithm*. This is a nice property for a sorting algorithm to have. By favoring elements from the left sublist over the right, we will be preserving the relative order of elements. It can be shown that as a result, MergeSort is a stable sorting algorithm. (This is not immediate, but it can be proved by induction.)

Analysis: What remains is to analyze the running time of MergeSort. First let us consider the running time of the procedure `Merge(A, p, q, r)`. Let $n = r - p + 1$ denote the total length of both the left and right subarrays. What is the running time of Merge as a function of n? The algorithm contains four loops (none nested in the other). It is easy to see that each loop can be executed at most n times. (If you are a bit more careful you can actually see that all the while-loops together can only be executed n times in total, because each execution copies one new element to the array B, and B only has space for n elements.) Thus the running time to Merge n items is $\Theta(n)$. Let us write this without the asymptotic notation, simply as n. (We'll see later why we do this.)

Now, how do we describe the running time of the entire MergeSort algorithm? We will do this through the use of a *recurrence*, that is, a function that is defined recursively in terms of itself. To avoid circularity, the recurrence for a given value of n is defined in terms of values that are strictly smaller than n. Finally, a recurrence has some basis values (e.g. for $n = 1$), which are defined explicitly.

Let's see how to apply this to MergeSort. Let $T(n)$ denote the worst case running time of MergeSort on an array of length n. For concreteness we could count whatever we like: number of lines of pseudocode, number of comparisons, number of array accesses, since these will only differ by a constant factor. Since all of the real work is done in the Merge procedure, we will count the total time spent in the Merge procedure.

First observe that if we call MergeSort with a list containing a single element, then the running time is a constant. Since we are ignoring constant factors, we can just write $T(n) = 1$. When we call MergeSort with a list of length $n > 1$, e.g. `Merge(A, p, r)`, where $r-p+1 = n$, the algorithm first computes $q = \lfloor (p + r)/2 \rfloor$. The subarray $A[p..q]$, which contains $q - p + 1$ elements. You can verify (by some tedious floor-ceiling arithmetic, or simpler by just trying an odd example and an even example) that is of size $\lceil n/2 \rceil$. Thus the remaining subarray $A[q+1..r]$ has $\lfloor n/2 \rfloor$ elements in it. How long does it take to sort the left subarray? We do not know this, but because $\lceil n/2 \rceil < n$ for $n > 1$, we can express this as $T(\lceil n/2 \rceil)$. Similarly, we can express the time that it takes to sort the right subarray as $T(\lfloor n/2 \rfloor)$. Finally, to merge both sorted lists takes n time, by the comments made above. In conclusion we have

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

Lecture 7: Recurrences

(Tuesday, Feb 17, 1998)

Read: Chapt. 4 on recurrences. Skip Section 4.4.

Divide and Conquer and Recurrences: Last time we introduced divide-and-conquer as a basic technique for designing efficient algorithms. Recall that the basic steps in divide-and-conquer solution are (1) divide the problem into a small number of subproblems, (2) solve each subproblem recursively, and (3) combine the solutions to the subproblems to a global solution. We also described MergeSort, a sorting algorithm based on divide-and-conquer.

Because divide-and-conquer is an important design technique, and because it naturally gives rise to recursive algorithms, it is important to develop mathematical techniques for solving recurrences, either exactly or asymptotically. To do this, we introduced the notion of a *recurrence*, that is, a recursively defined function. Today we discuss a number of techniques for solving recurrences.

MergeSort Recurrence: Here is the recurrence we derived last time for MergeSort. Recall that $T(n)$ is the time to run MergeSort on a list of size n. We argued that if the list is of length 1, then the total sorting time is a constant $\Theta(1)$. If $n > 1$, then we must recursively sort two sublists, one of size $\lceil n/2 \rceil$ and the other of size $\lfloor n/2 \rfloor$, and the nonrecursive part took $\Theta(n)$ time for splitting the list (constant time)