Lecture 20

CSE 110

6 August 1992

1 Divide-and-Conquer Sorting

For a more serious application of recursion, consider the sorting problem again.

First observe that simple sorting algorithms, like insertion sort, run much faster on short lists than on long lists.

1.1 How Long Does insertion Sort Take?

Suppose we have a list of n things, and we want to find the smallest member. We have to scan over all the elements in the list, looking for the smallest one. The three fundamental operations we have to perform are looking at an element, comparing it to the current smallest, and copying the element into our 'current smallest' variable if it is smaller.

The number of times we have to perform the first two operations is clearly n. The number of times we have to perform the third operation is clearly no more than n. If all three operations take 1 unit of time,[1] then the running time of our algorithm to find the smallest element is no more than $3n$. It can be demonstrated that the number of times we can expect to have to perform the third step is about $(n+1)/2$, so the running time of the find-smallest-element operation averages about $(5n+1)/2$.

Thus if we double the length of the list, the time it takes to find the smallest element also doubles.

Now consider the insertion sort: We repeatedly find the smallest element,

[1] This is usually true on conventional computers; however, the main conclusion of this section, that insertion sort takes time proportional to the square of the number of elements in the list, is still valid even if these three operations do not all take the same amount of time.

and copy it to an auxiliary array. To do this once on a list of n things takes $(5n + 3)/2$ units of time ($(5n + 1)/2$ for the search and 1 for the copy). We have to do it once for each element in the original list, so the total time to do an insertion sort on n things is about $n \cdot (5n + 3)/2$ or $(5n^2 + 3n)/2$.

n	$(5n^2 + 3n)/2$
1	4
2	13
3	27
4	46
5	70
10	265
20	1030
40	4060
80	16120

It seems that if we double to length of the list, the time it takes to sort it approximately quadruples.

1.2 An Improvement to Insertion Sorting

Suppose we have a list of 20 things we want to sort. We can see from the table above that sorting them with insertion sort will take about 1,030 units of time.

Now suppose we broke the list of 20 things into two lists of 10 things. It would take 265 units of time to sort each of the two lists of 10 things, for a total of 530 units of time. If we would merge the two sorted lists together in less than 500 units of time, we'd have done the sort faster by splitting it into two pieces.

In fact, it's quick and easy to merge two sorted lists. You look at the first element in each list, find the smaller of the two, remove it from the list it's in, and copy it to an auxiliary list. This takes 4 units of time. Then you repeat the process until both lists are empty. If the two lists have a total of n things in them, you have to repeat this operation n times, so it takes $4n$ units of time. In the case above, n was 20, so it takes only 80 units of time to merge the two sorted lists of length 10.

That means the split-sort-and-merge technique has reduced the running time of our sort from 1030 units to 610 units.

If we were sorting 80 things instead, we would reduce the running time from 16120 units to 8440 ($4060 + 4060 + 320$), a savings of almost 50%. Clearly this is a substantial improvement.

1.3 Recursion

Now suppose we're sorting the 80 things. We break the list into two lists of 40 things each. We need to sort each list.

We've already seen that it's a substantial improvement to break a list in half, sort the parts separately, and merge them, rather than sorting the entire list. So let's take this sub-list of 40 elements and apply our improvement, by breaking it into two sub-sub-lists of 20 things each. To sort each list of 40 things by this method takes $1030 + 1030 + 160 = 2220$ units of time, for a total of $2220 + 2220 + 320 = 4760$ units of time, down from 16120.

But we already saw that by using the improved method to sort a list of 20 things we could get a speed up, so let's apply the improvement yet again, breaking the lists of 20 things into two lists of 10 things each. This reduces the running time of our sort to 3080 units. The improvement is less this time, because the savings we got by sorting lists of 10 things instead of 20 things is not so big, and because the cost of merging the lists back together an extra time is larger in proportion.

Nevertheless if we sort the lists of 10 things by breaking them each into two lists of 5 things and doing insertion sort on those lists and merging the results, we can sort our original 80 objects in 2400 time units, instead of the 16120 that our original straight insertion sort yielded.

So here's our improved sorting algorithm:

1. If the list to sort is very short, use insertion sort.
2. Otherwise, break the list into two lists of approximately equal size, sort each list with *this* algorithm, and merge the two sorted lists back together.

This algorithm is called *mergesort*. We won't see a C implementation, because to do it properly you get bogged down in a lot of little details about how to store the objects, and allocating auxiliary space, and handling odd-length lists which can't be broken evenly, and soforth. But it's clear that the algorithm will be recursive. Somewhere in our program we will have a function something like this:

```
void mergesort(struct list *data, int length)
{
  ...
  if (length < SMALL_SIZE )
    insertion_sort(data, length);
  else {
    split_list(data, top, bottom);
    mergesort(top,    length/2);
    mergesort(bottom, length/2);
    merge_lists(top, bottom, data);
    return;
  }
}
```

`mergesort` does a little preparatory work, calls itself do to the bulk of the work on simpler cases, and does a little cleanup work.

2 Optimization and Performance

By doing a careful analysis and by using a better algorithm, we were able to improve the speed of a sorting program enormously. The gain for a short list, of 80 things, was to cut the running time of the sort by 85%. The improvement would be even greater for longer lists; for a 500-element list our mergesort would take about $\frac{1}{30}$ the time that the insertion sort would, instead of $\frac{1}{7}$.

This demonstration suggests a number of things: First, that program performance is largely determined by the overall efficiency of the algorithm the program uses, and much less so by micro details of the code.

Therefore, concern about whether you are doing one extra test or not each time through your add-a-node-to-a-list function is misplaced. Such considerations are usually dominated by more important matters.

Mergesort uses a clever algorithm and sorts n object is time that is approximately proportional to $n \cdot logn$; straight insertion sort sorts n objects in time that is approximately proportional to n^2. Changing a sort program to use a fast sort like mergesort rather than a slow sort like straight insertion sort will improve performance more than fussing around with extra tests.

There's a saying in the business that there are two rules for when to optimize:[2]

[2]The Berkeley UNIX fortune file attributes this to 'Michael Jackson'.

1. Don't do it.
2. (For experts only) Don't do it yet.

This is good advice. Write the code itself to be a straightforward implementation of a good algorithm, and write it to be clear to humans and easy to maintain, rather than to avoid a couple of unnecessary tests.

If, *after* the program is written, it actually turns out to be 'too slow' for some practical use, then optimize first by considering obvious waste in the program and by researching better algorithms.

If, after due consideration and research, you decide that no faster suitable algorithm is available, then micro-optimization may be appropriate. (Then again it may not.) There are tools available on most platforms that will analyze a run of your program and tell you where the program is spending most of its time. Then you can micro-optimize these parts. There's a rule of thumb called the '90-10' rule, which is that the program spends about 90% of its running time executing about 10% of the program, and the other 90% of the program is initializations and special cases that get executed rarely or only once.

Computer time is cheap these days, and if your program runs a little slow, that is probably all right. On the other hand programmer time is expensive, and if someone has to spend a lot of time figuring out your code because you made it obscure in an effort to save a few microseconds, then you've made a bad trade.

The summary is: Write for style, not for efficiency, because style is probably more important than you think it is. and efficiency is probably less important than you think it is. When you direct your attentions toward efficiency considerations, you should do so in a way that is likely to yield the most results: Improve the algorithm first, then, if you must, do real research to find out what parts of your code are actually slowing down the program, and fix those and nothing else.