COMP 200: Elements of Computer Science
Fall 2004
Lecture 17: October 4, 2004

Finish Divide and Conquer; Start Greedy

On the Board

Homework 3: handed back
Homework 4: available by Wednesday
Exam 1: not yet graded

Back to Algorithms

Last class, we looked at the computation of max, min, and talked about the computation of MaxMin. We saw simple algorithms for max and min, then built up a divide and conquer algorithm for max or min. The added complexity of the divide and conquer approach did not help us much with max or min — in fact, the complex algorithm was no faster than the simplest algorithm. I pointed out, however, that finding both the max and the min at the same time can be faster in a divide and conquer algorithm than the sum of the costs of the straightforward algorithms.

Review: Divide and Conquer

Divide and conquer is an algorithmic strategy that we can apply to many algorithmic problems. The name is intuitive and hints at the fact that the strategy has much broader application than algorithmic problem solving. Divide and conquer algorithms follow a simple pattern:

1. Split the problem instance into smaller parts until solution is easy — the *divide step*. Almost always, balance in the size of the smaller parts is important to the efficiency of the resulting algorithm.
2. Combine the solutions for the smaller parts into a solution for the whole — the *combine step*. The algorithm must combine solutions to subproblems in a quick and efficient way.

Sorting with Divide and Conquer

The same insights can lead us to an efficient sorting technique, often called a *merge sort*. Assume, again, an even-length list.

1. *Divide step:* recursively divide the lists down to lists of length one. At each step.

2. *Combine step:* Given two sorted lists, merge them together to produce a single sorted list. The code to handle this step involves dealing with arbitrary relationships among the first remaining element in each list.

   Four real cases:

   - First < second: remove head of first list and add it to the end of the result list
   - First ≥ second: remove head of second list and add it to the end of the result list
   - First is exhausted, second is not: copy the rest of second onto the end of the result list
   - Second is exhausted, first is not: copy the rest of the first list onto the end of the result list

   Given these four cases in *Merge*, we can write the merge sort as a recursion, much like the max algorithm, where the recursion calls

   Merge(first half of list, second half of list)

Of course, we've skipped many details along the way.

In particular, we need to manage the data and data movement. In an environment where we can see the data movement, we don't want to create new lists at each step. We can get away with two vectors and always work from one to the other. An easier way to envision this process is to merge into a temporary location and copy back into the final location. (This approach doubles the data movement but simplifies understanding.)

Code for Mergesort

Sort is easy:

```
sort ( list )
   if (list has one element)
      then return a list containing that element
      else
     split the list into two lists, left and right
        return merge( sort( left), sort( right ) )
```

Of course, list manipulation is easy in Scheme — we can use cons, first, and rest. To split the lists, we simply toss one element onto left and the next onto right, and keep repeating this pattern until we encounter empty.

In pidgin-Algol, with the list stored in a vector, the list-management details are harder. Assume that we can represent a range in the vector with a pair (lower, upper). Then, we can write the code as

```
sort ( lower, upper  ) ; where lower & upper describe a range in keys
  if (lower = upper)
     then return fee  ; a one-element list is always sorted
     else
        midpt ← floor( (lower+upper-1) / 2)
        merge( sort( lower, midpt ) , sort( midpt+1, upper )
```

Merge is harder. It must take two pairs that represent sorted lists and return a single sorted list. Conceptually:

```
merge ( leftlist, rightlist )
   while (neither leftlist nor rightlist is empty)
      if (first (leftlist) < first (rightlist))
         then
               move first(leftlist) to outputlist
      else if (first (leftlist) ≥ first (rightlist))
         then
               move first(righttlist) to outputlist

   ;  one list is empty
   if (leftlist is empty)
      move rest of leftlist to outputlist
   else if (rightlist is empty)
      move rest of rightlist to outputlist

   ;  and …
   return outputlist
```

Working out the details with (lower,upper) pairs to represent the lists involved a fair amount of detail. Working out the meaning of "move" in that context requires either a temporary array — into which we can move the values — and an operation that copies them back into the Keys array, or a more complex scheme for managing the lists — such as swapping between two arrays at each level in the recursion. [***Draw this on the board.***]

How well does this work? At each level in the recursion tree, *Merge* performs *n* comparisons, where *n* is the length of the original list. The recursion can only go *n/2* deep. (Dividing a list of length *n* in half will recur at most *n*/2 times.) Thus, the total number of comparisons must be n * n/2

or $n^2/2$. This is half the number of comparisons that we found in the worst case for bubblesort, so the algorithm has promise.

Greedy Algorithms

Many algorithmic problems involve trying to minimize or maximize some measure of utility. In making change for an amount less than one dollar, the classic algorithm tries to minimize the number of coins that must be used. The algorithm is simple:

1. *Use as many quarters as possible*
2. *Use as many dimes as possible*
3. *Use as many nickels as possible*
4. *Finish the amount with pennies*

Of course, we can add fifty-cent pieces in the obvious way. This simple algorithm is an example of a dynamic greedy solution. We don't plan out the number of coins. We take the largest benefit first — covering as many cents of the amount with the largest coins at each step. Because we only use four (or five) kinds of coin, it halts quickly — four decisions. It always produces the optimal answer.

Greedy algorithms arise in many contexts. Driving from Rice to the Galleria is one of my favorites. At each intersection, a hurried driver decides to go straight, right, or left. That choice is made on the basis of local traffic conditions plus the history of all past decisions made since leaving Rice — in essence, where the car currently sits. The decision-maker is ignorant, except in a general way, of local traffic conditions at the next intersection on each of the possible paths.

Graphs

The driving problem, like many problems that Computer Scientists formulate, can be represented with a discrete structure called a ***graph***. To a Computer Scientist, a graph is a set of nodes (or vertices) and a set of edges. Edges connect nodes in the graph. [***Draw an example graph on the board.***]

The trees that we considered last week are specialized forms of a graph. A binary tree (recall the ancestor tree) has the property that each node has two descendants and a single parent. These properties ensure that no sharing occurs — each node has a clear and unique ancestry.

We can formulate the driving problem as a graph, where the vertices represent intersections in the road system and the edges represent blocks of road between intersections. If we associate a number (or ***weight***) with each

edge that represents the number of seconds required to drive that stretch of road, including average stop-light delays, then the problem becomes one of finding a sequence of edges from the vertex for Rice to the vertex for the Galleria that has minimal total edge-weights.

A greedy approach does moderately well on this problem.