COMP 200: Elements of Computer Science
Fall 2004
Lecture 16: October 1, 2004

Divide and Conquer

On the Board

Homework 4: available Monday

Back to Algorithms

Chapter 4 of the book focuses on different strategies for solving algorithmic problems. Along the way, it introduces new algorithmic problems and their solutions (as examples that are amenable to solution with a variety of methods). Some of them will appear contrived; others will be of obvious utility.

Consider the problem of finding the maximal value in a list.

Draw an 8 element list on the board.

The obvious algorithm runs down the list and keeps track of the smallest value.

```
max ← list[1]
for i ← 2 to n
   if (max < list[i])
      then max ← list[i]
; at this point, max holds the biggest element in the list
```

The algorithm assumes that the list has at least one element. (To develop this program using the methodology, we need to invent a new definition for list, a "non-empty-list". With a non-empty-list, we can derive the Scheme version of this program from the template-based, data-driven methodology. However, the simple pidgin-Algol above is equally clear and much less trouble to explain.) We can find the smallest element in an analogous way.

```
min ← list[1]
for i ← 2 to n
   if (min > list[i])
      then min ← list[i]
; at this point, min holds the smallest element in the list
```

These two algorithms take the expected amount of time — each must look at every element in the list to determine if it is the extreme value (max or min). Stepping back from the algorithms, it seems reasonable to expect that we must look at each value to find the max (or min); so these two algorithms are reasonably efficient.

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
2. Combine the solutions for the smaller parts into a solution for the whole.

A divide-and-conquer approach to finding the maximum element in a list is easy to derive and to understand:

1. *Divide step:* split the list into smaller lists (recursively) until each list has one element. For a one-element list, the element's value is the maximum.
2. *Combine step:* to find the maximum of two smaller lists, take the larger of their maximum values.

Making it work: We need to introduce the notion of a pair (a structure of two counting numbers) to represent ranges inside a list. Assume that a pair has two elements, lower and upper. In Scheme, we would write

```
(define-struct pair (lower upper)).)
```

Given pairs, we might write the divide and conquer Max as

```
; assume the list has an even length
max(fee) ; where fee is a pair
    if (fee-lower = fee-upper)
       then return list[fee-lower]
       else
       {
          x ← max((make-pair
                          fee-lower
                          floor((fee-upper — fee-lower) /2)))
         y ← max((make-pair
                          ceiling((fee-upper — fee-lower) /2))
                          fee-upper)
          if x < y
             then return x
             else  return y
       }
; to invoke max, we simply call it with the full range
max(make-pair 1 k)
```

What happens if the list is not a power of two in length? Some piece of the recursion simply finds the degenerate (size one) case more quickly than others. If the list has odd length, we can write a wrapper function that tests for it and checks that last element against max(rest of the list) — at no extra cost.

```
wrapper(fie) ; where fie is a pair
   if (fie-lower = fie-upper)
      then return list[fie-lower]  ; one element
   else if (fie-upper – fie-lower + 1) is even
      then return max(fie)
      else
      {
         x ← max( (make-pair fie-lower (fie-upper – 1))
         if list[fie-upper] < x
            then return x
            else return list[fie-upper]
      }
```

Did divide-and-conquer improve the algorithm? Not in any obvious way. The algorithm still looks at each element in the list and makes a comparison for each element in the list. The code is actually much more complex than

the simple list-traversing loop that we wrote out for the obvious algorithm. So, what's the big deal?

A Win for Divide-and-Conquer

If, instead of max, we wanted to compute both max and min, we could use the same approach. Conceptually, we would divide the list down to the point where each list has two elements and return a pair (min, max). To combine the pairs for two lists, it suffices to compare the minimum values against each other and the maximum values against each other. The overall code returns a pair, (min, max) for the whole list.

This approach does fewer comparisons than running separate max and min calculations, since it only performs one comparison on the degenerate lists of length two. Divide and conquer produced a better answer than writing the obvious list traversal code.