Lecture 3a

Divide-and-Conquer--Introduction

Many important problems in computer science lend themselves to elegant solutions via the Divide-and-Conquer approach. This general approach is not only natural for many problems, but it also leads to solutions of "better than expected" complexity. We have already considered a problem that has a nice Divide-and-Conquer approach, namely the "laundry room tiling" problem. In this introductory section, we'll look at three standard examples. Each gives an indication of how one might analyze recursive algorithms. The third example does give a "better than expected" solution.

Here is the basic general plan:

1. Divide the problem into smaller instances (usually two or more), preferably with all the subproblems about the same size.

2. Solve the smaller instances. If small enough, solve directly; otherwise, subdivide again.

3. Piece together, if necessary, solutions to the smaller problems to form a solution to the original problem.

Example 1: Binary Search

Why is it a bit different than typical D&Q problems? There is only one instance of a subproblem (i.e., we ignore the half of the array that can't contain the target). Also, the solution to the subproblem is the solution to the original problem (i.e., no step 3, piecing together, is required).

```
int recBinSrch(int A[], int target, int low, int hi)
{
 if (low>hi)
   return -1
 else
    //  check midpoint
    //   if equal to target return position of midpoint
    //    else if A[mid] > target
             recursively binary search lower half
          else
             recursively binary search upper half

}
```

What is its complexity of the above algorithm?

For simplicity, let's assume that n, the size of the array, is a perfect power of 2. That is, n =2^k. Let's count comparisons. And let's give a name to the quantity we're trying to count. So let's let C(n) represent the number of comparisons needed to binary search an array of size n.

Note that C(1) = 1—we need one comparison to check an array of size 1 for the target

Thinking recursively, we get the following basic fact:

$C(n) = C(n/2) + 1$ --we do one comparison with the middle element and then we must binary search an array of size n/2.

So what is C(n/2)? We can use the same approach, and in fact continue in this fashion to obtain:

$C(n) = C(n/2) + 1$

$C(n/2) = C(n/4) + 1$

and so $C(n) = C(n/2) + 1 = C(n/4) + 1 + 1$

$C(n/4) = C(n/8) + 1$

and so $C(n) = C(n/4) + 1 + 1 = C(n/8) + 1 + 1 + 1 = C(n/2^3) + 3$

…

$C(n) = C(n/2^k) + k \; = \; C(1) + k = 1 + k = 1 + \log n = \Theta(\log n)$

So our intuition is confirmed mathematically—binary search is a logarithmic algorithm. This makes sense—we can only divide the array in half log n times. Although this example is a simple one, it demonstrates a typical technique in analyzing recursive functions. This particular method, which we'll encounter often, is called "back substitution".

Example 2: **Mergesort**

The basic elementary sorts (selection sort, insertion sort, and bubble sort) are all quadratic sorts. We will now see a sort that is faster than quadratic and is probably the most "natural" of the faster sorts.

It's a nice example because:

1. Natural recursive solution
2. Easy to analyze
3. Merge operation is an important one
4. Basis of many external sorting algorithms

The Merge operation:

2	1	→	1
3	7		2
6	9		3
8	13		6
10	14		8
12	15		9
			…

How long does it take to merge (comparisons)? In the best case (everything in list1 less than everything in list 2), n/2 comparisons. In the worst case (lists are "perfectly alternating), n-1 comparison.

Basic idea of merge sort:

Split array in half
Mergesort left half
Mergesort right half
Merge the two halves together

(A picture and a tree diagram are provided on the Moodle page.)

Complexity Analysis: Let's follow our lead from the binary search analysis. Assume n = 2k and let M(n) represent the number of comparisons to Mergesort an array of size n and let C(n) represent the number of comparisons to merge two sorted subarrays of size n/2 into a single sorted array of size n.

$M(n) = 2M(n/2) + C(n) \le 2M(n/2) + n$ (since in the worst case $C(n) \le n$)

$M(n/2) \le 2M(n/2^2) + n/2$ and so

$$M(n) \le 2^2 M(n/2^2) + 2n \le \ldots \le 2^k M(n/2^k) + kn = n \cdot 1 + kn = n + n\log n = \Theta(n\log n)$$

Notes:
It can be shown that any sort based on comparisons is $\Omega(n\log n)$. That is, in terms of complexity classes, Mergesort is as good as we can do. However, there are other nlogn sorts that tend to outperform Mergesort.

Mergesort uses O(n) extra storage (and copying). There are techniques that do "in-place" merge sorting (using "back half of the array and swapping instead of copying) but such algorithms tend to execute more slowly.