CSE 101 Class Notes
Divide and Conquer Algorithms

October 25, 2006

D&C has the following basic steps:

- *Divide* the problem into smaller, similar sub-problems.
- *Conquer* each sub-problem recursively.
- *Combine* the sub-problem results to yield a result for the full problem.

A D&C algorithm, usually stated recursively, may be appropriate if the following conditions obtain:

- The problem can be divided into a *fixed* number of sub-problems.
- Each sub-problem is smaller than the original, and all are of similar size.
- Sub-problems have the same structure as the original problem.
- Sub-problems can be solved independently.
- Sub-problems' results can be combined cheaply.

Example: Sorting

We want to sort an array of n elements by sorting some smaller sub-arrays, and combining the results. One way to do this would be to sort $n-1$ elements, then insert the remaining element (i.e. insertion sort). This not the best application of D&C because it divides the problem into uneven-sized subparts – D&C is effective by reducing the problem size by a multiplicative (e.g. $n/2$) rather than additive (e.g. $n-1$) factor. Indeed, insertion sort ends up being $O(n^2)$.

A better approach is to divide the array into two equal-sized sub-arrays, recursively sort them, and combine the results. In a bit more detail:

```
Mergesort(A[1..n])
    if n > 1
        A1 <- Mergesort(A[1 .. n/2])
        A2 <- Mergesort(A[n/2+1 .. n])
        return Merge(A1, A2)
```

```
6        else
7            return A
8
9    Merge(A[1..n], B[1..m])
10       i <- 1
11       j <- 1
12       C <- { Array [1..n+m] }
13       for k = 1 .. n + m
14           if i = n
15               C[k] <- B[j]
16               j <- j + 1
17           else if j = m
18               C[k] <- A[i]
19               i <- i + 1
20           else if A[i] < B[j]
21               C[k] <- A[i]
22               i <- i + 1
23           else
24               C[k] <- B[j]
25               j <- j + 1
26       return C
```

Here the *divide* step happens on lines 3-4, while the *combine* step happens on lines 5 and 9-26.

Running Time

Line 5 takes $O(n)$. To find the total time required, we need to state it as a recurrence

$$\begin{aligned} T(n) &= 2T(n/2) + O(n) \\ T(1) &= O(1) \end{aligned}$$

We can think of all the calls as forming a tree where each node has two children, each of which is half the size of its parents, and has a cost proportional to its size. Since it bottoms out at $n = 1$, we know this tree has height $\log n$. This information can also be represented in a table:

depth	sub-problems	size	cost per node	total cost
0	1	n	cn	cn
1	2	$n/2$	$cn/2$	cn
2	4	$n/4$	$cn/4$	cn
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
i	2^i	$n/2^i$	$cn/2^i$	cn
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$\log n$	n	1	c	cn
TOTAL				$cn \log n$

Implementation

While the recursive version is easy to analyze, an iterative implementation can be more efficient. Mergesort lends itself naturally to the following bottom-up iterative implementation:

```
Mergesort(A[1..n])
    j <- 1
    while j < n
        for i = 1 to n-2*j by 2*j
            Merge_inplace(A[i .. i+j], A[i+j+1 .. i+2*j])
        j <- j * 2
```

Since this iterative version does all the same work as the recursive version, its asymptotic running time is the same, while its constant factors will be smaller (*Exercise:* show the correspondence between merges in the iterative and recursive versions of Mergesort).

Example: All Pairs Binary Tree Distances

Input: A complete binary tree T with n nodes and distances on the edges.

Output: The total distance between every pair of nodes (i, j).

The naive approach of considering every pair of nodes separately takes time $O(n^2 \log n)$, because there are n^2 pairs of nodes, and a path between them has length $\log n$. Since the the size of our output is n^2, we know the algorithm must be at least $O(n^2)$. However, since the naive approach repeats a lot of work, we suspect that we can do better, and since we are dealing with a recursive problem on trees, D&C may be appropriate.

The D&C algorithm first (recursively) computes distances in the subtrees rooted at each of a node's children, then updates distances to reflect the paths to the parent node, and between nodes in opposite child subtrees.

```
M <- { Array [1..n, 1..n] of distances }
M <- Infiniti
Alldist(t)
    if !has_children(t)
        return
    Alldist(left(t))
    Alldist(right(t))
    M[left(t),t] <- dist(t,left(t))
    M[right(t),t] <- dist(t,right(t))
    for c = { descendants(left(t)) }
        M[c,t] <- M[c,left(t)] + dist(t,left(t))
    for c = { descendants(right(t)) }
        M[c,t] <- M[c,right(t)] + dist(t,right(t))
    for c = { descendants(left(t)) }
```

```
15            for d = { descendants(right(t)) }
16                M[c,d] = M[c,t] + M[t,d]
```

Running Time

Lines 1-2 each take $O(n^2)$. Each subtree has size $n/2$, so the loops on lines 10 and 12 repeat $n/2$ times, while the nested loops on lines 14-16 run $(n/2)^2$ times. The operations in these loops take $O(1)$, so their total cost is $O(n^2)$. The two recursive calls on lines 6-7 take $T(n/2)$, so the recurrence relation is

$$\begin{aligned} T(n) &= 2T(n/2) + O(n^2) \\ T(1) &= O(1) \end{aligned}$$

This looks like $O(n^2 \log n)$, which is disappointing given how much more complicated our algorithm has become. But let's write down the table:

depth	sub-problems	size	cost per node	total cost
0	1	n	cn^2	cn^2
1	2	$n/2$	$c(n/2)^2$	$cn^2/2$
2	4	$n/4$	$c(n/4)^2$	$cn^2/4$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
2^i	2^i	$c(n/2^i)^2$	$c(n/2^i)^2$	$cn^2/2^i$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$\log n$	2^n	1	c	$cn^2/2^{\log n}$
TOTAL				$cn^2(1/2 + 1/4 + \cdots + 1/n)$ $= 2cn^2 = O(n^2)$

Happily, the total time is actually $O(n^2)$. Note that the table illustrates that the tree is top-heavy, i.e. that almost all the work is done at the top level.