Computer Science 431
Algorithms
The College of Saint Rose
Spring 2013

Topic Notes: Divide and Conquer

Divide–and-Conquer is a very common and very powerful algorithm design technique. The general idea:

1. Divide the complete instance of problem into two (sometimes more) subproblems that are smaller instances of the original.
2. Solve the subproblems (recursively).
3. Combine the subproblem solutions into a solution to the complete (original) instance.

While the most common case is that the problem of size n is divided into 2 subproblems of size $\frac{n}{2}$. But in general, we can divide the problem into b subproblems of size $\frac{n}{b}$, where a of those subproblems need to be solved.

This leads to a general recurrence for divide-and-conquer problems:

$$T(n) = aT(n/b) + f(n), \text{ where } f(n) \in \Theta(n^d), d \geq 0.$$

When we encounter a recurrence of this form, we can use the *master theorem* to determine the efficiency class of T:

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Application of this theorem will often allow us to do a quick analysis of many divide-and-conquer algorithms without having to solve the recurrence in detail.

Mergesort

Each sorting procedure we have considered so far is an "in-place" sort. They require only $\Theta(1)$ extra space for temporary storage.

Next, we consider a divide-and-conquer procedure that uses $\Theta(n)$ extra space in the form of a second array.

It's based on the idea that if you're given two sorted arrays, you can merge them into a third in $\Theta(n)$ time. Each comparison will lead to one more item being placed into its final location, limiting the number of comparisons to $n - 1$.

In the general case, however, this doesn't do anything for our efforts to sort the original array. We have completely unsorted data, not two sorted arrays to merge.

But we can create two arrays to merge if we split the array in half, sort each half independently, and then merge them together (hence the need for the extra $\Theta(n)$ space).

If we keep doing this recursively, we can reduce the "sort half of the array" problem to the trivial cases.

This approach, the *merge sort*, was invented by John von Neumann in 1945.

How many splits will it take? $\Theta(\log n)$

Then we will have $\Theta(\log n)$ merge steps, each of which involves sub-arrays totaling in size to n, so each merge (which will be k independent merges into $\frac{n}{k}$-element arrays) step has $\Theta(n)$ operations.

This suggests an overall complexity of $\Theta(n \log n)$.

Let's look at pseudocode for this:

```
mergesort(A[0..n-1])
  if n>1
    copy first half of array A into a temp array B
    copy second half of array A into a temp array C
    mergesort(B)
    mergesort(C)
    merge(B and C into A)
```

where the merge operation is:

```
merge(B[0..p-1], C[0..q.1], A[0..(p+q-1)])
  while B and C have more elements
    choose smaller of items at the start of B or C
    remove the item from B or C
    add it to the end of A
  copy remaining items of B or C into A
```

Let's do a bit more formal analysis of mergesort. To keep things simple, we will assume that $n = 2^k$.

Our basic operation will be the number of comparisons that need to be made.

The recurrence for the number of comparisons for a mergesort of a problem of size $n = 2^k$ is

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, C(1) = 0.$$

The best, worst, and average cases depend on how long we are in the main while loop in the merge operation before one of the arrays is empty (as the remaining elements are then taken from the other array with no comparions needed). Let's consider the worst case, where the merge will take $n-1$ comparisons (one array becomes empty only when the other has a single element remaining). This leads to the recurrence:

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1, C_{worst}(1) = 0.$$

The master theorem, gives is that $C_{worst}(n) \in \Theta(n \log n)$.

Quicksort

Another very popular divide and conquer sorting algorithm is the *quicksort*. This was developed by C. A. R. Hoare in 1962.

Unlike merge sort, quicksort is an in-place sort.

While merge sort divided the array in half at each step, sorted each half, and then merged (where all work is in the merge), quicksort works in the opposite order.

That is, quicksort splits the array (which takes lots of work) into parts consisting of the "smaller" elements and of the "larger" elements, sorts each part, and then puts them back together (trivially).

It proceeds by picking a *pivot* element, moving all elements to the correct side of the pivot, resulting in the pivot being in its final location, and two subproblems remaining that can be solved recursively.

A common and simple choice for the pivot is the leftmost element. We put it into its correct position and put all elements on their correct side of the pivot.

Psuedocode for a quicksort:

```
quicksort(A[l..r]) // we would start with l=0, r=n-1
  if l < r
    s = partition(A[l..r]) // s is pivot's location
    quicksort(A[l..s-1])
    quicksort(A[s+1..r])

partition(A[l..r])
  p = A[l]  // leftmost is pivot
  i = l; j = r+1
  do
    do i++ until i = r || A[i] >= p
    do j-- until j = l || A[j] <= p
    swap(A[i],A[j])
  until i>=j
  swap(A[i],A[j]) // undo last
```

```
swap(A[l],A[j]) // swap in pivot
return j
```

Note: we always make a recursive call on a smaller array (but it's easy to make a coding mistake where it doesn't, and then the sort never terminates).

The complexity of quicksort is harder to evaluate than merge sort because the pivot will not always wind up in the middle of the array (in the worst case, the pivot is the largest or smallest element).

Again, the basic operation will be the comparisons that take place in the partition.

The `partition` method is clearly $\Theta(n)$ because every comparison results in `left` or `right` moving toward the other and quit when they cross.

In the best case, the pivot element is always in the middle.

This would lead to a number of comparisons according to the recurrence:

$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0.$$

By solving the recurrence or applying the Master Theorem, we find that $C_{best}(n) \in \Theta(n \log n)$, exactly like merge sort.

In the worst case the pivot is at one of the ends and quicksort behaves like a selection sort. This occurs with already-sorted input or reverse-sorted input. To analyze this case, think of the first pass through the full n-element array. The first element, `A[0]`, is chosen as the pivot. The left-to-right inner loop will terminate after one comparison (since `A[0]` is the smallest element). The right-to-left inner loop will perform comparisons with `A[n-1]`, `A[n-2]`, ... all the way down to `A[0]` since we need to "move" the pivot item to its own position. That's $n+1$ comparisons for the partition step. In the process, the problem size is decreased by 1, so there will be n comparisons in the next step, $n-1$ in the third, and so on. We stop after processing the two-element case (requiring 3 comparisons), so the total comparisons is given by:

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2)$$

A careful analysis can show that quicksort is $\Theta(n \log n)$ in the average case (under reasonable assumptions on distribution of elements of array). We can proceed by assuming that the partition can occur at any position with the same probability ($\frac{1}{n}$). This leads to a more complex recurrence:

$$C_{avg}(n) = \frac{1}{n} \sum_{s=0}^{n-1} [(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1, C_{avg}(0) = 0, C_{avg}(1) = 0.$$

We will not solve this in detail, but it works out to:

$$C_{avg}(n) \approx 2n \ln n \approx 1.38n \log_2 n \in \Theta(n \log n).$$

Clearly, the efficiency of quicksort depends on the selection of a good pivot. Improving our chances to select a good pivot will ensure quicker progress. One way to do this is to consider three candidates (often the leftmost, rightmost, and middle elements) and take the median of these three as the pivot value.

Other improvements include switching to a simpler ($\Theta(n^2)$) sort once the subproblems get below a certain threshold size, and implementing quicksort iteratively rather than recursively.

Quicksort is often the method of choice for general purpose sorting with large data sizes.

Binary Trees

No discussion of divide and conquer can proceed without a discussion of *binary trees* and algorithms on them.

We discussed the idea of tree data structures and introduced lots of tree-related terminology at the start of the semester. For now, we will just consider a few examples.

We can make analysis convenient by defining a binary tree as either the empty set, or a tree node plus two other binary trees: a left subtree and a right subtree.

To find the height of a tree:

```
height(T)
  if (T is empty) return -1
  else return max(height(T.left),height(T.right)) + 1
```

To analyze this, we first note that the size of the problem is the number of nodes in our tree T, denoted $n(T)$.

The basic operation is either the comparison needed to find the max or the addition of 1 once we find the max. A recurrence for either:

$$A(n(T)) = A(n(T.left)) + A(n(T.right)) + 1 \quad \text{for } n(T) > 0, A(0) = 0$$

Note, however, that there are in fact more checks to see if the tree is empty than there are additions or comparisons. That check happens at the base case only.

The number of additions/max comparisons: $A(n) = n$, while the number of checks for an empty tree: $C(n) = 2n + 1$.

The other key divide and conquer operations are the recursively defined tree traversals, which we discussed earlier:

1. *preorder*: visit the root, then visit the left subtree, then visit the right subtree.
2. *in-order* visit the left subtree, then visit the root, then visit the right subtree.

3. *postorder*: visit the left subtree, then visit the right subtree, then visit the root.

Pseudocode is staightforward, for example:

```
inorder(T)
  if (T is not empty)
    inorder(T.left)
    visit(T.root)
    inorder(T.right)
```

Analysis is similar to that of height for this and for preorder and postorder. Each node is visited once.

Strassen's Matrix Multiplication

Our text describes a divide and conquer multiplication of large numbers, but since you likely saw that in discrete math, we will move along to consider Strassen's matrix-matrix multiplication.

This algorithm improves upon the standard matrix-matrix multiplication by observing that the product of two 2×2 matrices can be performed using 7 multiplications instead of the usual 8.

This in itself does not seem that significant, especially when we consider that the standard algorithm requires 4 additions, while Strassen's algorithm requires 18 additions/subtractions.

The real benefit comes when we apply this idea to get a divide-and-conquer algorithm for multiplying matrices. To multiply 2 $n \times n$ matrices, we break it into 4 $\frac{n}{2} \times \frac{n}{2}$ matrices and use Strassen's algorithm to multiply them.

Our recurrence for the number of multiplications with this approach:

$$M(n) = 7M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

From which the Master Theorem (or a backward substitution) will yield:

$$M(n) \in \Theta(n^{\log_2 7}) \approx n^{2.807}.$$

Which is definitely a slower rate of growth than $\Theta(n^3)$.

An analysis of the (larger) number of additions/subtractions results in the same efficiency class: $A(n) \in \Theta(n^{\log_2 7})$.

Many other approaches have been invented with even smaller rates of growth than Strassen's algorithm, but most have constant factors that make them impractical for real usage.

Computational Geometry

We return to two familiar problems from computational geometry to explore divide-and-conquer solutions that are more efficient than the brute force approaches considered previously.

Closest Pairs

Our problem is to find among a set of points in the plane the two points that are closest together.

We begin by assuming that the points in the set are ordered by increasing x coordinate values. If this is not the case, the points can certainly be sorted in $O(n \log n)$ time as a preprocessing step.

We then divide the points into two subsets, S_1 and S_2, each of which contains $\frac{n}{2}$ points (which is easy since the points are sorted by x coordinate).

We then recursively find the closest pair of points in each subset S_1 and S_2. If the distance between the closest pair in $S_1 = d_1$ and the distance between the closest pair in $S_2 = d_2$. We then know that $d = \min\{d_1, d_2\}$ is an upper bound on the minimum distance between any pair, but we still need to make sure we check for shorter distances where one point is in S_1 and the other is in S_2.

The only points which might be closer together than distance d are those within a strip of width d from the dividing line between the subsets. For each point within that strip and within one subset, we potentially need to consider all points from within the strip within the other subset. That still sounds like a lot of work. The key observation is that for each point on one side, we have to consider points whose y coordinates are within d. This will mean at most 6 points from the other side, since if there are more points than that within d in the y coordinate on the other side, at least one pair from among that point would be closer than distance d to each other.

So how do we find those points to check? They can be found quickly if we also keep the points sorted in order by y coordinate. Still, this seems difficult but it can be done efficiently (see the text's description for details).

We end up with a recurrence:

$$T(n) = 2T(n/2) + O(n)$$

which given an overall time of $T(n) \in O(n \log n)$.

Quickhull

The other computational geometry problem discussed in the text is called *quickhull* – an algorithm for finding the convex hull of a set of points. We will not discuss it in class, but it is worth reading.