NOTES FOR OCTOBER 1, 2002

Divide and Conquer

Divide and Conquer a commonly used technique for designing algorithms. We will study three different algorithms that exploit this: Strassen's matrix multiply algorithm, Closest Pair of Points, and the FFT.

Divide and conquer goes a little like this.

- Define subproblems [Divide] ($D(n)$)
- Solve subproblems recursively [Conquer] ($C(n)$)
- Reconstruct from the subsolutions the answer [Recombine] ($R(n)$)

It is usually the case that $T(n) = aT(n/b) + C(n) + R(n)$ and $R(n) = O(n^k)$ and $C(n) \in O(f(n)) \in o(R(n))$. Here, $T(n)$ refers to the amount of time it takes to solve a problem of size n.

The *Master Theorem* in the book describes how to solve relations in this form, and divides the solution space into three parts. Please refer to the book for more information.

Mergesort is the canonical divide-and-conquer algorithm. Mergesort splits a list in half, and then calls itself recursively on each half. When the input list is only one element long, mergesort returns it as a trivailly sorted list. Mergesort then takes the two sorted sublists and merges them as you would a deck of cards. The full algorithm, its proof of correctness, and the analysis of its running time is in CLRS somewhere in the first few chapters.

The time complexity of mergesort is $T(n) \leq 2(T(n/2) + O(n)$. This can be proved by induction on n to be $O(n \log n)$.

As an aside, we're assuming a lot of things when we design these algorithms. We're assuming constant-time access to memory, that arithmetic operations and comparisons take constant time, and that we have some method of implementing arrays such that accessing any element in that array takes constant time. These are the assumptions that underlie the RAM model of computation, and are reasonable for the moment. Other types of computers don't behave this way, and even sequential processors have some difficulty with really large input sizes. One could point out that the asymptotic analyses that we do are sometimes meaningless because the asymptotic behavior is exhibited only in regions that fall outside of these assumptions. But, whatever, that's not a useful argument: the asymptotic properties are still fun to study and you'd be missing the larger point—algorithms are mathematical objects, and can be studied as such.

Returning to the problem of recurrence relations, we can identify three "styles" of recurrence:

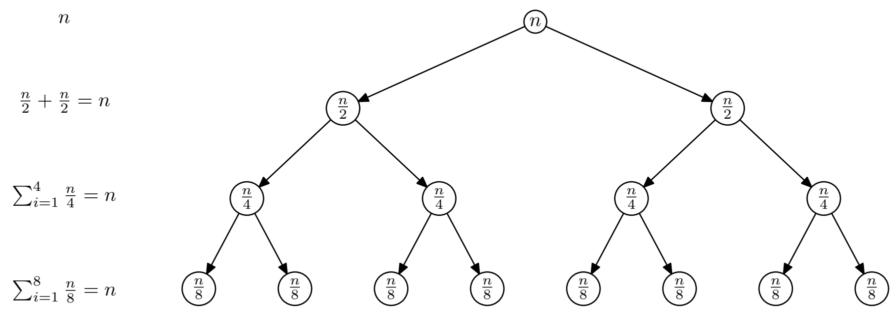
top-heavy: The bulk of the work is done almost entirely at the top level, which dominates the time for the recursive calls

bottom-heavy: The bulk of the work is done almost entirely in the base cases

steady-state: Each level of recursion does about the same amount of work.

Each of these cases is represented in the Master Theorem, but this gives you an intuitive way to quickly guess at the asymptotic behavior.

Figure 1. The recursion tree for mergesort.


n
$\frac{n}{2}+\frac{n}{2}=n$
$\sum_{i=1}^{4}\frac{n}{4}=n$
$\sum_{i=1}^{8}\frac{n}{8}=n$
n
$\frac{n}{2}$
$\frac{n}{4}$
$\frac{n}{8}$

The basic idea in solving these recurrences is to create a "recursion tree," and watch as the constant associated with the asymptotic order changes on each level. For example, for mergesort, the recursion tree looks like figure 1. Mergesort is a perfect example of the "steady-state" case, because the total amount of work on each level (i.e., the sum of the work done in the nodes of that level) remains constant for all levels. Here, the root level represents the amount of work performed on the input list, *exclusive* of the recursive calls to the function. The first level represents the amount of work performed in the first batch of recursive calls, and so on. The figure shows four levels, or the recursion chain for a list of size 8. As you can see, the total amount of work done on each level is the same, so we have $T(n) = 2T(n/2)+n$, to which we can guess the solution to be of the form $T(n) \leq cn \log n$ where c is some constant, and we prove it by induction, being quite careful of course to make sure that when we make our inductive conclusion that our new c is not different than the value of c we made in the inductive hypothesis. Right?

In the steady state case, you can see that each level produces a reasonable amount of work. In the top heavy case, the work is primarily done in the top few levels of the tree and lower levels do less and less, so the work is often bounded by a polynomial. In the bottom heavy case, the work is done mostly in the leaves of the tree, and the work is bounded by another polynomial (generally formed by summing the work in the leaves).

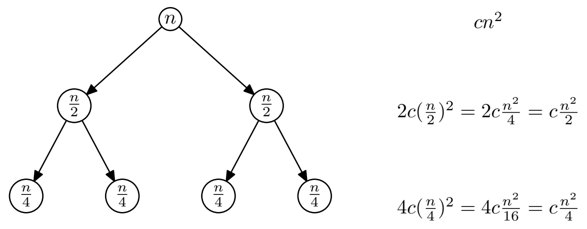
In general, you should now have some intuition that if $T(n) = aT(n/b) + O(n^k)$, you can see sort of by inspection what the recursion tree should look like (bottom heavy, top heavy, steady state) based on the values for a, b and k. From this you can generate a good guess for the order of $T(n)$. This is not actually a proof, by any stretch of the imagination, but a lot of this sort of work requires a bit of intuition at first that eventually gets filled in with formality. If nothing else, you can use this sort of intuitive approach as a basis of a real proof. Still, don't confuse the two.

Bottom Heavy Recurrence

```
AllSums($A[1..N]$): $S[N,N]$, array of integers.
1: $S[1..\frac{n}{2}, 1..\frac{n}{2}] \leftarrow$ AllSums($A[1..\frac{n}{2}]$) {Upper left corner}
2: $S[\frac{n}{2}+1..n, \frac{n}{2}+1..n] \leftarrow$ AllSums($A[\frac{n}{2}+1..n]$ {Lower right corner}
3: for $i = 1$ upto $\frac{n}{2}$ do
4:    for $j = \frac{n}{2}+1$ upto $n$ do
5:       $S[i,j] = S[i,\frac{n}{2}] + S[\frac{n}{2}+1, j]$
6:    end for
7: end for
```

Algorithm 1: Intra-array summation

FIGURE 2. A (partial) recursion tree for the `AllSums` algorithm


n
$\frac{n}{2}$
$\frac{n}{2}$
$\frac{n}{4}$
$\frac{n}{4}$
$\frac{n}{4}$
$\frac{n}{4}$
cn^2
$2c(\frac{n}{2})^2 = 2c\frac{n^2}{4} = c\frac{n^2}{2}$
$4c(\frac{n}{4})^2 = 4c\frac{n^2}{16} = c\frac{n^2}{4}$

We will now look at a new algorithm: summation of all array entries. The input is an array $A[1..n]$ of integers, and the output is $\mathrm{Sum}[N,N]$ where $\mathrm{Sum}[i,j] = \sum_{k=i}^{j} A[k]$. Not surprisingly, this algorithm will be $O(n^2)$, but we'll look at it as an example of how we should analyze algorithms. The algorithm is shown in algorithm 1.

The recursion tree here looks something like figure 2. What we see is that $T_i(n) = \frac{1}{2^i}cn^2$ where i is the level of recursion, so to calculate $T(n)$ we do

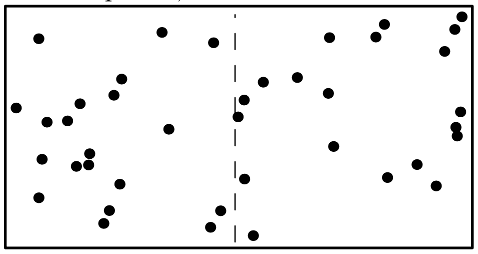
$$\begin{aligned} T(n) &= \sum_{i=0}^{\log n} T_i(n) \\ &= \sum_{i=0}^{\log n} \frac{1}{2^i}cn^2 \\ &\leq 2cn^2. \end{aligned}$$

where this last inequality is due to the summation of a harmonic series; see appendix A in the text for more information.

The math worked out here as our intuition expected: because we divided the array into halves and then performed quadratic work on inputs of size $n/2$, and then (in some sense) interpolated the information for the other parts of the matrix in *less than n^2 time*, we're dominated by the work done in the first iteration so a rough time analysis for the first iteration would be a good guess for the order.

□

Figure 3. A set of points, divided into a left set and a right set



Recall that the generic recursion relation that we're somewhat familiar with is $T(n) = aT(n/b) + f(n)$. One constraint on $f(n)$ is that it be a polynomial, that is, of the form cn^k. One algorithm we will look at today will violate this constraint, and we'll see how to deal with it.

At the ith level of a recursion tree with this type of recurrence relation, we have $a^i \frac{n^k}{b^{ik}}$ amount of work to do. This term forms a geometric series in i (taking k and n to be constants for this particular instance of the problem). This is where the idea of "phase-changes" comes into play: if $\frac{a}{b^k} > 1$ we have the bottom heavy case, since work increases as i increases. If we have $\frac{a}{b^k} < 1$ then work decreases and we have the top-heavy case. When $\frac{a}{b^k} = 1$ we're in steady-state.

As an aside, this seems like a motivation for the Master Theorem. Eventually we will need to deal with non-constant a's and b's, and cases where $f(n)$ is not a simple polynomial function, but neither this method nor the Master Theorem cover this. A good place to learn about these sorts of things is the ACM Computing Surveys article done a while back on this, or the back of Neapolitan and Naimapour.

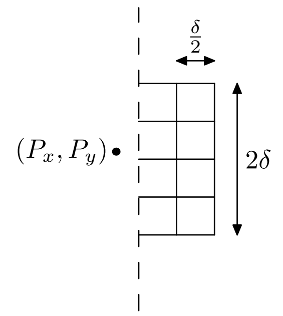
Closest Pair of Points

Let's examine a new problem of finding the two points that are closest together in a set of points. Clearly the input is a set of points $p_i = (x_i, y_i)$, and we'll use the Euclidean distance between two points p_i and p_j as the metric that we're minimizing: $\|p_i, p_j\| = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

The naive way of handling this problem would be to take all combinations of two points in the set and to calculate the distance between each pair, saving the smallest pair. Since there are $\frac{n(n-1)}{2}$ different ways of combining two items from a set of n things, this algorithm runs in $O(n^2)$ time. This is not good enough.

If we look at the divide-and-conquer approach, we might consider dividing the points into two sections, as indicated in figure 3. Suppose for an instant that we could find the minimum distance (and the corresponding points) between the points in a set. Then, if we had such a function we could run it on each half of the original set of points, and then do some weird stitching together at the boundary; just because we found the closest two points in each set doesn't mean that we found the closest points *between* the sets. We are helped to some extent by the fact that we already have an upper bound for the smallest distance (it's the smallest of the two minimal distances that were calculated through this mystery function) so we know that there are only certain points that we'll need to check, and each of those points will have only finitely (and in fact, small) many neighbors that we need to look at.

FIGURE 4. Points at the boundary



In particular, let's just say δ is the smallest distance in either of the sets. Then we should look at points in each set that are only δ from the boundary. Furthermore, any point on the other side of the boundary would have to be at least as close as δ to the boundary, and within 2δ of the altitude (because each point could be within δ above or below the current altitude. Note that this bounding box does not represent points that necessarily are less than δ away from the candidate point, but points that *might* be closer. The situation is presented in figure 4: we have 8 boxes, each of height $\delta/2$ and each of width $\delta/2$ that we need to check.

Intuitively, you can see that there may be at most two points in this $\delta \times 2\delta$ bounding box and that they must be at least δ apart, so we don't need to consider distances between points on the same side of the boundary (another way to motivate this is that you have already computed the smallest possible distance, through this magical function).

A first pass at an algorithm is shown in algorithm 1.

An interesting insight is that because you can determine a constant upper bound on the number of points to check on each iteration, and because you have an initial upper bound for the minimum distance δ, you can limit the search space *a priori.*

So how does the timing for this algorithm work out? If we take a naive approach to calculating the median, and if we do the sorting of the y-coordinates, then we have the recurrence relation $T(n) = 2T(n/2) + \Theta(n \log n)$. This, of course, does not work in the set of equations we have above since $f(n)$ is not a polynomial. What we need to do is form a common-sense bound to show how this function behaves. Fortunately, as is generally the case, we're not trying to solve the recurrence relation, but simply to put some sort of reasonable bound on it.

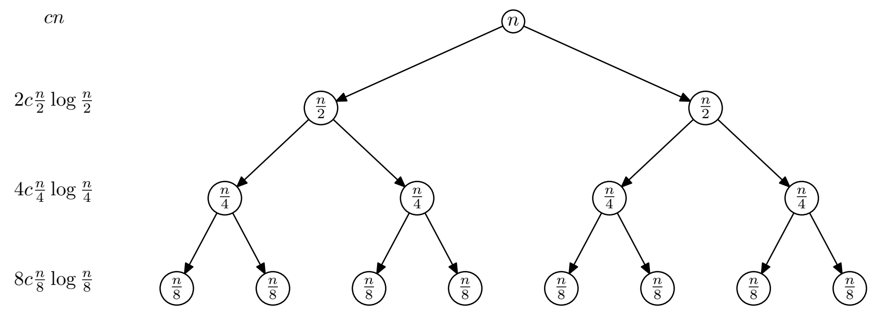
Basically, what's happening in this recurrence is that at each level of the recursion tree, the amount of work is changing predictably, but not as a harmonic series. At some point, this stops being a top-heavy recursion and starts being steady-state. We need to figure out an upper and lower bound for the way that this work is changing, and then bound the time as the sum of two pieces. A recursion tree for the first few levels is shown in figure 5, and you can see how the work is changing as a function of the level i: $T(n) = \Sigma_{i=1}^{\log n} cn \log \frac{n}{2^i}$.

We still need to determine a decent bound for this. We see that there are 2^i problems of $\frac{cn}{2^i} \log \frac{n}{2^i}$ size each, and we see that the work starts off in a steady-state case and then switches to top-heavy. What we'll do is pick the top row of the tree, multiply that by the depth of the tree, and call that an upper bound. That is, $T(n) \leq cn \log n \times \log n = cn \log^2 n$, or $T(n) \in O(n \log^2 n)$. We can get

```
    CPoP({p_1, p_2, ... p_n}): two points
1:  $x_m \leftarrow$ median of$x_i$
2:  $L \leftarrow p_i : x_i \leq x_m$, $R \leftarrow p_i : x_i > x_m$
3:  $(p_i, p_j) = \text{CPoP}(L)$, $(p_k, p_l) = \text{CPoP}(R)$
4:  $\delta \leftarrow \min(\|p_i, p_j\|, \|p_k, p_l\|)$
5:  $L \leftarrow \{p_i \in L : x_i \geq x_m - \delta\}$
6:  $R \leftarrow \{p_i \in R : x_i \leq x_m + \delta\}$
7:  Sort $L$ and $R$ by $y$-coordinate
8:  $i \leftarrow 1$, $j \leftarrow 1$.
9:  for all points in $L$ do
10:     $(x', y') \leftarrow L_i$
11:     $(x'', y'') \leftarrow R_j$
12:     repeat
13:         increment $j$
14:     until $y'' > y' - \delta$
15:     $k \leftarrow j$, $(x''', y''') \leftarrow R_k$
16:     repeat
17:         save $i, k$ as champion
18:     until $y''' > y + \delta$
19: end for
```

Algorithm 2: First take at Closest Pair of Points algorithm

FIGURE 5. The Recursion Tree for $T(n) = 2T(n/2) + \Theta(n \log n)$


cn
$2c\frac{n}{2}\log\frac{n}{2}$
$4c\frac{n}{4}\log\frac{n}{4}$
$8c\frac{n}{8}\log\frac{n}{8}$
n
$\frac{n}{2}$
$\frac{n}{4}$
$\frac{n}{8}$

a lower bound on $T(n)$ by noticing that at the level k halfway up the tree that $2^k = 2^{1/2 \log n} = \sqrt{n}$, so that $cn \log n/2^k = 1/2cn \log n$. We know that the work in the upper part of the tree must be greater than the depth of this row times the work it performs, which is $1/2cn \log n \times 1/2 \log n$, and $T(n) \in \Omega(n \log^2 n)$. Thus, $T(n) \in \Theta(n \log^2 n)$, but we had to go about it the long way.

A particular problem that we have is that the algorithm is still kind of slow, mostly because we are redoing some repetitious work repeatedly over and over again the same way. Specifically, we are resorting a list of points at each recursive step; we sort it in the x direction to find the median, and we sort it in the y direction to do the stitching together of the two sides of the recursion. If we pre-sorted the lists and then maintained a mapping of each point into the sorted x list and the sorted y list, then this algorithm falls completely into the steady-state case with $f(n) = n$ and we get $T(n) \in O(n \log n)$. We will see, fairly shortly, a method (actually, two methods) for calculating the median in linear time, but this won't help us since for this algorithm we have to process the L and R lists in sorted order of y.$\square$