Main topics are #divide-and-conquer and #recurrences including example/Karatsuba_multiplication.

Merge Sort

- Last week we discussed divide-and-conquer algorithms including the prototypical example of mergesort.

```mergesort
MERGE(A[1..n], m):
    i ← 1; j ← m + 1
    for k ← 1 to n
        if j > n
            B[k] ← A[i]; i ← i + 1
        else if i > m
            B[k] ← A[j]; j ← j + 1
        else if A[i] < A[j]
            B[k] ← A[i]; i ← i + 1
        else
            B[k] ← A[j]; j ← j + 1
    for k ← 1 to n
        A[k] ← B[k]
```

- To analyze any divide-and-conquer algorithm, including mergesort, we need to write and solve recurrence for its running time.
- Let T(n) denote the worst-case running time for MergeSort(A[1 .. n]), whatever it is.
- MergeSort does an O(n) time for loop plus it takes the time to do two recursive calls on arrays of size ceil(n / 2) and floor(n / 2). We can typically ignore the floors and ceilings in divide-and-conquer recurrences, so T(n) = T(n) = 2T(n / 2) + O(n)
- But what is T(n) in simpler terms? We need some asymptotic bound that is true for all T(n) following that recurrence. Of course, it’s O(n log n). But why?

Recursion Trees

- We can use a recursion tree to solve this and similar recurrences.
- A recursion tree is a rooted tree that describes the contributions to a recurrence or the time spent in a recursive algorithm.
- Each node is a recursive subproblem called at some point during the algorithm’s execution.
- A node’s children are the recursive subproblems called by that node, and the root is the top-level call to the algorithm.
- So in MergeSort, for example, we have the root representing the top call, and each node except for the leaves gets two children.
The value of each node is the time spent by the recursive subproblem excluding other recursive calls.

So recall the definition of big-Oh. For large enough $n$, $T(n) \leq 2T(n/2) + cn$ for some constant $c$.

So, we write $cn$ in the root node. Each child call works on a problem of size $n/2$ so these nodes have value $cn/2$.

In general, a node at depth $i$ gets a value of $cn/2^i$, and there are $2^i$ nodes of depth $i$.

The overall time spent by mergesort, the solution to $T(n)$, is the time spent in all those recursive calls. We need to sum the value of all the nodes.

The easiest way to evaluate this sum is to do so level-by-level. So what is the sum within each level?

Each level (except the base cases) has a sum of exactly $cn$.

We divide the problem size by 2 in each recursive call, so the depth or number of levels is $\lg n$.

So $T(n) \leq cn \lg n$. MergeSort runs in $O(n \log n)$ time.

Let's look at a more general case.

Often, but not always, you'll be dealing with an algorithm that does $f(n)$ non-recursive work and makes $r$ recursive calls, each on a subproblem of size $n/c$.

These algorithms have run time recurrences that look like $T(n) = r T(n/c) + f(n)$ with a base case of $f(1) = \Theta(1)$.

In this case, each internal node of the recursion tree has $r$ children.

The root gets a value of $f(n)$. The problem size at depth $i$ is $n/c^i$, so those nodes get value $f(n/c^i)$.

Again, to compute $T(n)$, we need to sum the values on each node, and its easiest do so level-by-level.

There are $r^i$ nodes at depth $i$, so the sum at that depth is $r^i f(n/c^i)$.

The easiest thing to do in practice is to draw the first few levels of the tree to see the big picture:
The leaves of the recursion tree correspond to base cases of the algorithm. Since we’re aiming for an asymptotic bound anyway, we’ll assume the worst that the leaves go all the way down to instances of size $n_0 = 1$.

- $T(n)$ is the sum of all node values, so
  $$T(n) = \sum_{i=0}^{L} r^i f(n/c^i)$$
  where $L$ is the depth of the recursion tree.

- Our choice of $n_0 = 1$ means $L = \log_c n$ and there are $r^L = r^{\log_c n} = n^{\log_c r}$ leaves. The values of the leaves sum to $n^{\log_c r} f(1) = \Theta(n^{\log_c r})$.

- There are three common cases where the level-by-level series (the sum over all node values) is easy to evaluate.
  - **Decreasing**: If the series decays exponentially, meaning each term is at most a constant $< 1$ times the previous one, then the sum is dominated by its first and largest term. $T(n) = \Theta(f(n))$.
  - **Equal**: If all the term in the series are equal, then $T(n) = \Theta(f(n) \cdot L) = \Theta(f(n) \log n)$. Remember, the base of the log doesn’t matter if you’re just multiplying by it.
  - **Increasing**: If the series grows exponentially, meaning each term is at least a constant $> 1$ times the previous, then the sum is dominated by its last and largest term. $T(n) = \Theta(n^{\log_c r})$.

Looking for these three specific cases, exponential decay, everything being equal, or exponential growth is a variant of the **Master Method** of solving recurrences taught in CLRS. However, I think working with the recursion trees is easier to remember. These three particular cases work almost every time you would use the master method, and recursion trees in general work in more situations than the master method.

- For example, recursion trees can be used in the case that not every recursive call is the same size.
- Suppose $T(n) = T(n/3) + T(2n/3) + n$. [use the figure on the left]
Each full level of the recursion tree sums to $n$, and non-full levels sum to less.

The tree has depth $\log_{3/2} n = O(\log n)$, so $T(n) = O(n \log n)$.

On the other hand, there at least $\log_3 n = \Omega(\log n)$ full levels, so $T(n) = \Omega(n \log n)$. Ah, so $T(n) = \Theta(n \log n)$.

We’ll see more examples of unbalanced recursion trees on Monday.

**Multiplication**

- But for now, let’s use our new tool to analyze another example of a divide-and-conquer algorithm.
- A couple times already, we’ve discussed an algorithm for multiplying large numbers.
- But as we saw, that algorithm takes $O(n^2)$ time to multiply two $n$-digit numbers.
- Maybe we can do better using divide-and-conquer?
- Let $m$ be some non-negative integer. We can split the digits of $x$ and $y$ roughly in half so that $x = (10^m a + b)$ and $y = (10^m c + d)$ for some numbers $a$, $b$, $c$, and $d$.
- And now multiplying $x$ and $y$ comes down to observing $(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$.
- The four products that don’t involve $10^\text{something}$ use numbers with fewer digits, so maybe we can use divide-and-conquer!
- We’ll do the easier multiplications recursively, and then combine them using that formula.
- In pseudocode, we get the following algorithm. SplitMultiply($x$, $y$, $n$) computes $x \times y$ assuming they both use at most $n$ digits.
Correctness follows easily from induction (if \( n = 1 \), we just return the product. Otherwise, we correctly multiply the smaller values by the induction hypothesis and combine them according to the identity.)

So what is the running time? The mods and multiplying by \( 10^\text{whatever} \) takes linear time since it’s just digit shifts. Between that and the additions, everything outside the recursive calls takes \( O(n) \) time. There are 4 recursive calls on problems of roughly half the size, so we’ll say the running time is \( T(n) = 4T(n/2) + O(n) \).

The recursion tree method shows us the level-sums are exponentially increasing, meaning \( T(n) \) is bounded by the number of leaves. \( T(n) = O(n^{\log_2 4}) = O(n^2) \).

Oh, that didn’t help at all.

In the 1950’s, renounced mathematician Andrei Kolmogorov publicly conjectured that there is no algorithm for multiplying two \( n \)-digit numbers in \( o(n^2) \) time. He organized a seminar in 1960 where he planned to discuss this conjecture and several related problems. Almost one week later, 23-year-old student Anatolii Karatsuba found a better algorithm after all. Kolmogorov told the seminar participants about the better algorithm, and immediately terminated the seminar.

So, how do we do better? Well, for our divide-and-conquer algorithm, we need to compute \( bc + ad \). It turns out, given \( ac \) and \( bd \), we can compute that sum using only one more multiplication instead of two.

\[
bc + ad = ac + bd - ac + bc + ad - bd = ac + bd - (a - b)(c - d).
\]

So we get this alternative algorithm with only three recursive calls instead of four!
Now we have three recursive calls of size roughly $n/2$, so the running time is $T(n) = 3T(n/2) + O(n)$.

The level-sums of the recursion tree still form an increasing geometric series bounded by the number of leaves,…

\[
\begin{align*}
&\text{FastMultiply}(x, y, n): \\
&\quad \text{if } n = 1 \\
&\quad \quad \text{return } x \cdot y \\
&\quad \text{else} \\
&\quad \quad m \leftarrow \lfloor n/2 \rfloor \\
&\quad \quad a \leftarrow \lfloor x/10^m \rfloor; \ b \leftarrow x \mod 10^m \quad \{(x = 10^m a + b)\} \\
&\quad \quad c \leftarrow \lfloor y/10^m \rfloor; \ d \leftarrow y \mod 10^m \quad \{(y = 10^m c + d)\} \\
&\quad \quad e \leftarrow \text{FastMultiply}(a, c, m) \\
&\quad \quad f \leftarrow \text{FastMultiply}(b, d, m) \\
&\quad \quad g \leftarrow \text{FastMultiply}(a - b, c - d, m) \\
&\quad \quad \text{return } 10^{2m}e + 10^n(e + f - g) + f
\end{align*}
\]

but now $T(n) = O(n^{\log_2 3}) \approx O(n^{1.58496})$. That’s a big improvement!

- So when you’re designing your own divide-and-conquer algorithms, see if you can limit the number of recursive calls you perform to help speed things up.
- As for multiplication, you can take this idea even further by splitting the numbers into more pieces and combining the products in more complicated ways.
- And after a long line of improvements David Harvey and Joris van der Hoeven ended up finding an $O(n \log n)$ time algorithm. They found this algorithm only this year!
- Unfortunately, the constants are so ridiculous, that you would need $n$ to be bigger than the number of particles in the universe before their algorithm runs faster than others known before.
- Next Monday, we’ll look at examples where the subproblems are not so nicely balanced.