Main topics are **divide-and-conquer** and **recurrances** with **example/merge_sort** and **example/Karatsuba_multiplication**.

**Prelude**
- Please fill out the prerequisite form if you haven't already. Thank you!
- Homework 1 has been released and is due Tuesday, February 5th.
- I’m going to start uploading my notes after each lecture in case you find them useful. Remember, they’re very rough script, so they represent what I find important, but they may have typos or even bugs.

**Divide-and-Conquer**
- Last week we discussed a couple recursive algorithms including MergeSort, an example of a **divide-and-conquer** algorithm.
- Divide-and-conquer algorithms all share this form:
  1. **Divide** the given instance into several *independent smaller* instances.
  2. **Delegate** each smaller instance to the Recursion Fairy.
  3. **Combine** the solutions for the smaller instances into the final solution for the given instance.
- When the instance size falls below some threshold, the recursion **bottoms out** and you switch to some different (usually trivial) algorithm instead.
- Proving divide-and-conquer algorithms correct always requires induction.
- But how do we figure out their running time? We’ll have to look at Merge Sort again.

**Merge Sort**
- Here’s the pseudocode for merge sort again.
MergeSort is a recursive algorithm, so it stands to reason that its running time should depend recursively on the running time of its recursive calls.

To model that, we'll write the runtime using a recurrence, a function that is defined recursively.

Let's define $T(n)$ as the worst-case running time of MergeSort on an array of size $n$.

For sufficiently small $n$, $n < n_0$ for some constant $n_0$, we can say $T(n) \leq C$ for some constant $C$ (so $T(n) = O(1)$ when $n < n_0$). Sorting an array of constant length takes constant time.

Merge takes $O(n)$ time, because it's a simple for loop going from 1 to $n$. For $n \geq n_0$, we'll say merging the dividing the array takes at most $cn$ time for some constant $c$.

MergeSort calls itself twice on inputs of size ceiling$(n/2)$ and floor$(n/2)$ so $T(n) \leq T(\text{ceiling}(n/2)) + T(\text{floor}(n/2)) + cn$.

Our goal is to find as tight an asymptotic (big-Oh) bound as possible to express $T(n)$.

Like most divide-and-conquer recurrences, we can simplify this somewhat by removing floors and ceilings and assuming that constant $c$ is a 1. We'll get the same result up to constant factors. Erickson's book has the formal details on why that's OK, and if we have time today, I'll show you the argument.

But for now, let's just say $T(n) \leq 2T(n/2) + n$. Now what?

### Recursion Trees

- We'll use something called recursion trees to solve this recurrence.
- 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...

```
MergeSort(A[1..n]):
  if n > 1
    m ← ⌈n/2⌉
    MergeSort(A[1..m])  \(\text{ }\text{\(\text{Recursively}\)\text{ }\)}
    MergeSort(A[m+1..n]) \(\text{ }\text{\(\text{Recursively}\)\text{ }\)}
    Merge(A[1..n], m)
```

```
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]
```
gets two children.

- In each node, we write the running time of the subproblem outside the recursive calls.
- So, we spend n time in the root call. Each child call works on a problem of size n/2 so the merge takes n/2 time.
- In general, a node at depth i gets a value of n / 2^i, and there are 2^i nodes of depth i.
- Finally, the sum over all the nodes is the solution to the recurrence. It’s what we get when we add up all the +n’s for the different values of n.
- The easiest way to evaluate this sum is to do so level-by-level. So what is the sum within each level?
- Each level has a sum of exactly n.
- We divide the problem size by 2 in each recursive call, so the depth or number of levels is lg n.
- So T(n) ≤ n lg n. MergeSort runs in O(n log n) time.

Let’s look at a more general case.

- Often, but not always, divide-and-conquer 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).
- The algorithm for this recurrence makes r recursive calls, each on a subproblem of size n / c. Each subproblem of size k spends f(k) time outside of recursive calls.
- 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 it’s 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).

![Recursion Tree Diagram]

- We can write the whole sum as sum_{i=0}^{L} r^i f(n / c^i) where L = log_c n is the depth of the recursion tree (i.e. n / c^L = Theta(1)).
- There are three common cases where this sum is easy to evaluate:
• **Decreasing**: If \( r \cdot f(n / c) = k \cdot f(n) \) where \( k < 1 \), then the sum is a *decreasing geometric series*. Geometric series are asymptotically bounded by their largest term. Specifically, the sum is at most \( f(n) / (1 - k) = O(f(n)) \). It’s at least \( f(n) \) also, so \( T(n) = \Theta(f(n)) \).

• **Equal**: If \( r \cdot f(n / c) = f(n) \), then each level sum is equal and we get \( T(n) = \Theta(f(n) \cdot L) = \Theta(f(n) \log n) \).

• **Increasing**: If \( r \cdot f(n / c) = K \cdot f(n) \) where \( K > 1 \), then the sum is an *increasing geometric series*. Again, it’s bounded by its largest term which is the number of leaves in the recursion tree. \( T(n) = \Theta(r^{\log_c n}) = \Theta(n^{\log_c r}) \).

• Looking for these three specific cases (\( r \cdot f(n / c) = k \cdot f(n), r \cdot f(n / c) = f(n), \) or \( r \cdot f(n / c) = K \cdot f(n) \)) is sometimes called the *Master Method* of solving divide-and-conquer recurrences.

• CLRS offers a somewhat more technical version of the master method which they call the master theorem. It’s a bit more general as well, but recursion trees can be used in even more cases than the master method. I recommend just getting used to recursion trees so you don’t need to memorize either version.

• In particular, recursion trees can be used in the case that not every recursive call is the same size. For example, if you have an algorithm with the running time \( T(n) = T(n - 1) + T(1) + n \), the recursion tree looks like [left figure is MergeSort, right figure the current example].

- We can determine \( T(n) = O(n^2) \) by observing there are \( n \) levels, each of which sum to at most \( n \).

### Multiplication

• Let’s finish with another example of divide-and-conquer algorithms.

• A couple times already, we’ve discussed algorithms for multiplying large numbers.

• Both the peasant multiplication algorithm and the lattice multiplication algorithm you probably learned as children multiply two \( n \)-digit numbers in \( O(n^2) \) time.

• Maybe we can do better using divide-and-conquer?

• Let \( m = \text{ceil}(n/2) \). 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 \( (10^m a + b)(10^m c + d) = 10^{(2m)} ac + 10^m (bc + ad) + bd \).
These four products use numbers with fewer digits, so we can compute them using recursion.

In pseudocode, we get the following algorithm.

```
MULTIPLY(x, y, n):
  if n = 1
    return x · y
  else
    m ← ⌊n/2⌋
    a ← ⌊x/10^m⌋;  b ← x mod 10^m
    c ← ⌊y/10^m⌋;  d ← y mod 10^m
    e ← MULTIPLY(a, c, m)
    f ← MULTIPLY(b, d, m)
    g ← MULTIPLY(b, c, m)
    h ← MULTIPLY(a, d, m)
    return 10^2m e + 10^m (g + h) + f
```

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? Multiplying by 10^whatever takes linear time since it's just digit shifts. Between that and the additions, the combine step 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) + n.

The recursion tree method shows us the level-sums make an increasing geometric series, bounded by the number of leaves. In other words, we have the third case of the master method. 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 - (a - b)(c - d).
- So we get this alternative algorithm with only three recursive calls instead of four!

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

- Now we have three recursive calls of size roughly \(n/2\), so the running time is \(T(n) = 3T(n/2) + n\).

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

- Andrei Toom and Steven Cook found an infinite family of algorithms that split the integers into \(k\) parts, each with \(n/k\) digits. The product is computed using only \(2k - 1\) recursive multiplications. The resulting algorithm runs in \(O(n^{1 + 1/(\log k)})\) time where the \(O\) hides more constants that depend on your choice of \(k\).

- You can do even more clever things to get running times really really close to but not quite \(O(n \log n)\). Right now, the best running time comes from an algorithm of Harvey and van der Hoeven from last year. Their algorithm runs in \(O(n \log n \, 4^{\log^* n})\) time, where \(\log^* n\) is the number of times you need to take the log of \(n\) before it becomes less than 1. For all practical purposes, \(\log^* n \leq 6\), so the algorithm is essentially \(O(n \log n)\) time. Many people believe \(O(n \log n)\) should be the best running time possible.

**Domain Transformation**

- We somehow have time, so I’m going to show you why it was OK to ignore the ceilings and floors in MergeSort.

- We can do this using a technique called a *domain transformation*.

- First, the actual constants don’t matter when doing asymptotic bounds, pretty much by design. So solving \(R(n) \leq R(\text{ceiling}(n/2)) + R(\text{floor}(n/2)) + n\) would also give us the asymptotic running time bound for MergeSort.

- The worst-case running time of MergeSort is increasing in \(n\), so we can only make things
worse by assuming the recurrence uses two ceilings instead of a ceiling and a floor. Let's solve \( R(n) \leq 2R(\text{ceiling}(n/2)) + n \leq 2R(n/2 + 1) + n. \)

- Still weird looking, but now we can play one more trick to get rid of that +1.
- Let \( S(n) = R(n + \alpha) \). Hopefully we can get a clean \( S(n/2) \) in the recurrence if we just look at slightly higher values of \( R \).
- \( S(n) \)
  - \( = R(n + \alpha) \)
  - \( \leq 2R(n/2 + \alpha/2 + 1) + n + \alpha \)
  - \( = 2S(n/2 - \alpha/2 + 1) + n + \alpha \)
- If we set \( \alpha = 2 \), we get \( S(n) = 2S(n/2) + n + 2 \). One important fact for big-Oh notation is that when adding functions together, only the biggest part of the function matters. So we may as well solve \( 2S(n/2) + n \) like we wanted to.
- Finally, \( R(n) = S(n - 2) \). Plugging that \( n - 2 \) into any reasonable running time function \( g(n) \) won’t change the big-Oh bound, so it suffices to solve \( S(n) = 2S(n/2) + n \).
- Yeah, that was a lot of junk, and it’s totally not worth doing for every divide-and-conquer algorithm.
- From now on, I’m going to ask you to trust me that we can just ignore floors, ceilings, lower order terms, and the specific constant on the non-recursive part.
- So in the case of MergeSort, we’ll just say the running time is bounded by \( T(n) \leq 2T(n/2) + n \).