Lesson: DIVIDE AND CONQUER
(Slide 1) Divide and Conquer

Divide and Conquer Algorithms or Recursive Algorithms is one of the first algorithmic tools that many of you may have learn. For example, binary search or merge sort.

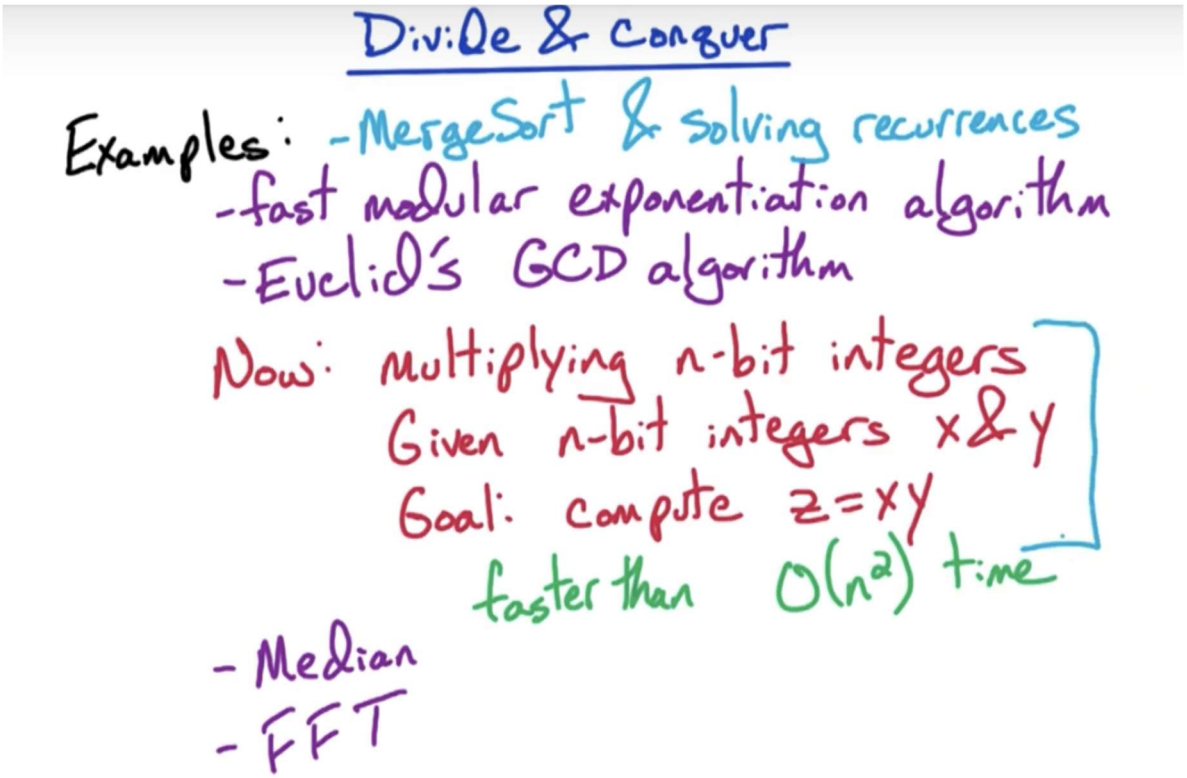
To see the power of divide and conquer, we'll look at a fundamental problem multiplying two n bit numbers. Here, we'll assume the numbers are huge. For example, they may be thousands of bits long. This will be the case in an application such as RSA. Since they are so large, we can no longer utilize the hardware implementation. We'll see a clever divide and conquer algorithm which is faster than the standard multiplication approach.

Another fundamental problem we'll look at is given n numbers, we'd like to find the median element. The numbers are unsorted, so they are in arbitrary order. Can we find the median without first sorting the list? We'll look at a really ingenious divide and conquer algorithm to do just that.

Finally, we'll dive into the beautiful FFT algorithm, Fast Fourier Transform. It's impossible to overstate the importance of this algorithm, it's used in many fields. For example, signal processing. In fact, it was called the most important numerical algorithm of our lifetime. At least that was the case at time of the quote in 1994. To understand FFT, we're going to have to recall some basics about complex numbers. Once you understand the basic mathematics behind it, you'll appreciate the beauty and simplicity of the algorithm. It will take some time and mental effort to get up to steam for the FFT algorithm, but it'll be worth it. It's a masterpiece.

(Slide 2) D&C: Overview

***



The next topic we'll dive into is the divide and conquer technique. We've seen a few examples of divide and conquer already. We saw two examples in the RSA algorithm. The first was the fast modular exponentiation algorithm which used the repeated squaring idea. The second was Euclid's GCD algorithm. These are two fairly simple applications of divide and conquer approach. We're going to see some more sophisticated examples in the following lecture.

What we're going to look at now is multiplying n-bit integers. One of the applications of this problem is in the RSA algorithm setting, there we have huge integers. The number of bits is like a thousand or 2000, so the basic arithmetic operations like multiplication are no longer built into hardware. The problem formulation is we're given two n-bit integers, x and y, and we want to compute their product, z, which is x times y. And we want to look at the running time as a function of the number of bits, that's the input size. What we discussed in the RSA algorithm lecture, was that the running time using a naive algorithm for computing the product of x times y would take n-squared time, O(n^2) time. What we're going to do now is an algorithm which is faster than this O(n^2) time. We're going to do a more sophisticated scheme.

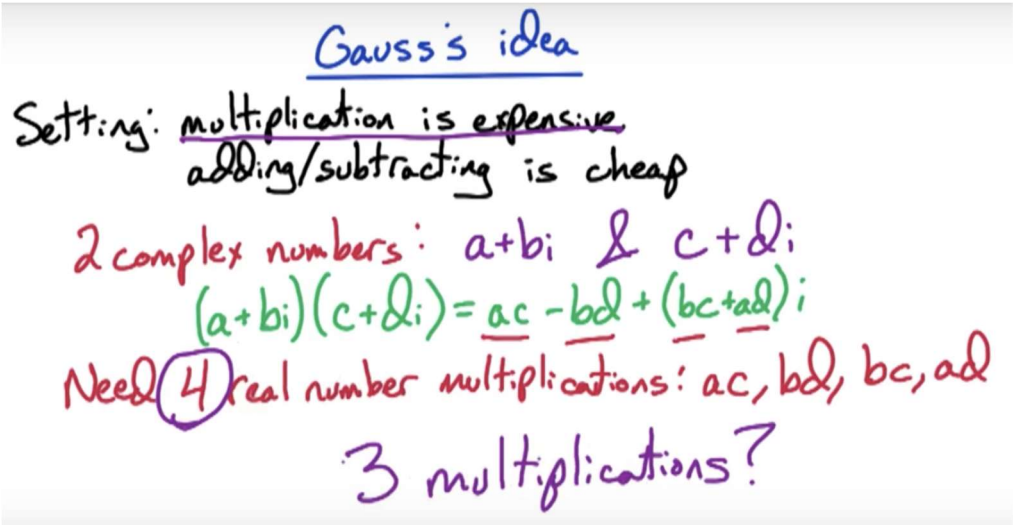
After this multiplication example, we're going to look at some other sophisticated examples, namely median. We're going to see how to compute the median in linear time and finally we're

also going to look at Fast Fourier Transform, FFT.

These three examples, multiplying n-bit integers, the linear time median and FFT are all very beautiful algorithms and I hope you'll learn to appreciate the essence of the algorithm. Now one important note is I'm assuming that you've seen divide and conquer before. For example, I'm assuming you've seen things like mergesort, the O(n log n) time algorithm for sorting n integers. And I'm also assuming that you know how to solve recurrences. If you need a refresher on some of these topics, I suggest you look through the textbook. Now let's go ahead and dive into this topic, multiplying n-bit integers..

(Slide 3) Multiplying Complex #'s

***



Before we dive into our example of multiplying n-bit integers. Let's take a brief digression.

We're going to look at how clever idea from Gauss. It's not at all going to be apparent how this is useful for our multiplication for example but it will turn out to be a very useful idea which will give us an improved algorithm.

Here's a setting that we're working in, multiplication is expensive, so we want to minimize the number of multiplications. Adding and subtracting is relatively cheap, so we're not going to worry about the number of additions and subtractions. So we're willing to add in additional additions and subtractions in order to reduce the number of multiplications.

Now, this fact that multiplications are expensive, and additions and subtractions are much cheaper is true in most cases. For example, when we're looking at multiplying n-bit integers, multiplying them takes O(n^2) time whereas if we're adding or subtracting it takes O(n) time.

Now, here's the problem of Gauss. We're given two complex numbers. How do I give you the complex number? I have to tell you the real and the imaginary part. So the first number is a + bi, a is the real part, b is the imaginary part. And the second number is c + di, c is the real part, d is the imaginary part. And our goal is to compute their product. We want to compute (a + bi) x (c + di).

Well, in order to get a handle on it, let's start by just expanding this out. So I get ac, like bi x di, i^2 is negative one, so that gives me -bd. And then, I get two terms which are multiplied by i, the first is bc and the second is ad. In order to compute this product it looks like we need to

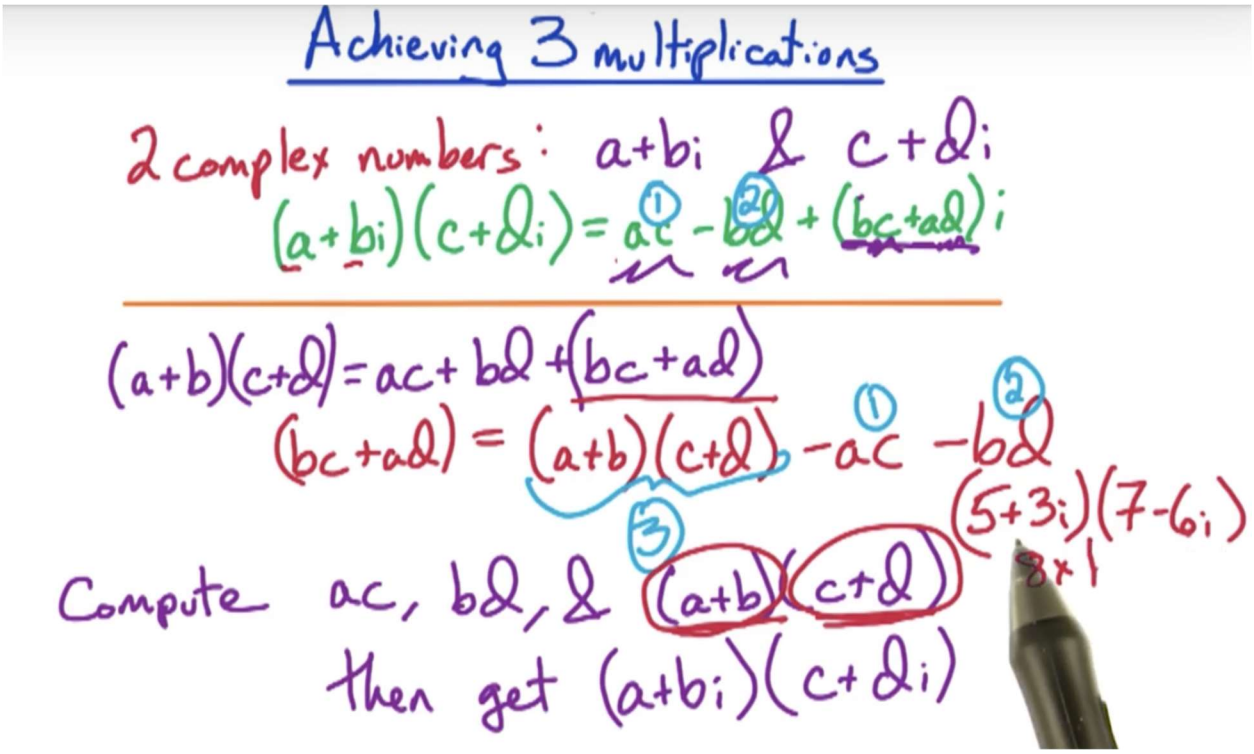
compute a x c, b x d, b x c, and a x d. Now each of these a, b, c, and d are just real numbers. So it looks like we need four real number multiplications. We need to compute a x c, b x d, b x c, a x d. If you give me these real number multiplications, then with one, two, three additions or subtractions, we can combine these products in order to get the product of these two complex numbers.

Now our setting is that multiplications are expensive. So we're trying to minimize the number of multiplications and we're willing to have additional additions or subtractions in order to reduce the number of multiplications. So, can we reduce the number of multiplications? Can we get it down from four? Is it possible to achieve this product with only three multiplications?

It turns out we can do it. The key is that we're going to compute this sum, bc + ad, we'll compute the sum without computing the individual terms. So we're not going to compute b x c and we're not going to compute a x d but we're going to compute their sum bc + ad.

(Slide 4) Improved Approach

***



So to recap, we have these two complex numbers: a+bi and c+di, and we want to compute their product. Now our goal is to minimize the number of real number multiplications in order to compute the product of these two complex numbers. We just saw that the naive approach has four real number multiplications: ac, bd and bc and ad. Now what we want to do is we want to try to achieve just three real number of multiplications in order to compute this product.

What we're going to do now is we're going to try to compute this sum, bc + ad without computing the individual terms bc and ad. That's going to leave us with a method for computing the product of these two complex numbers with just three real number multiplications needed.

How are we going to get this as a term, bc + ad? Well, notice it kind of looks like a cross terms. So let's write out the correct expression so that these are the cross terms. In order to achieve this, we need to have b on one side and c on the other side. And we need to have a on one side and d on the other side. So one side has b and a and the other side has c and d. Expand this out.

Notice these two terms bc + ad is exactly what we're trying to compute. What about these other two terms ac and bd? Notice, these are exactly the products that we're computing in order to compute the product of these two complex numbers.

Let's look back at this expression. We're trying to compute this sum, bc + ad. So let's solve it for this term. If we solve it for bc + ad, then we get that that's equal to (a + b)(c+d) - ac - bd. Here's the key now, in order to compute this product of these two complex numbers, what are the 3 now real number multiplications that we need? Well one of them is ac, just like before. The second one that we need is just like before is bd. The third term that we want to obtain is bc + ad, that's right here. How can we obtain it? Well let's use this expression. Look, ac we already know, bd we already know. What's the third thing we need? We need (a + b)(c + d). This is the third real number multiplication that we need. If we compute these three terms ac, bd and (a + b)(c + d) then from those we can compute these three terms, ac, bd, and bc + ad.

So the punchline is, if we compute these three real number multiplications, ac, bd, and (a + b)(c + d) then by just doing some additions and some subtractions, we can obtain the product of these two complex numbers.

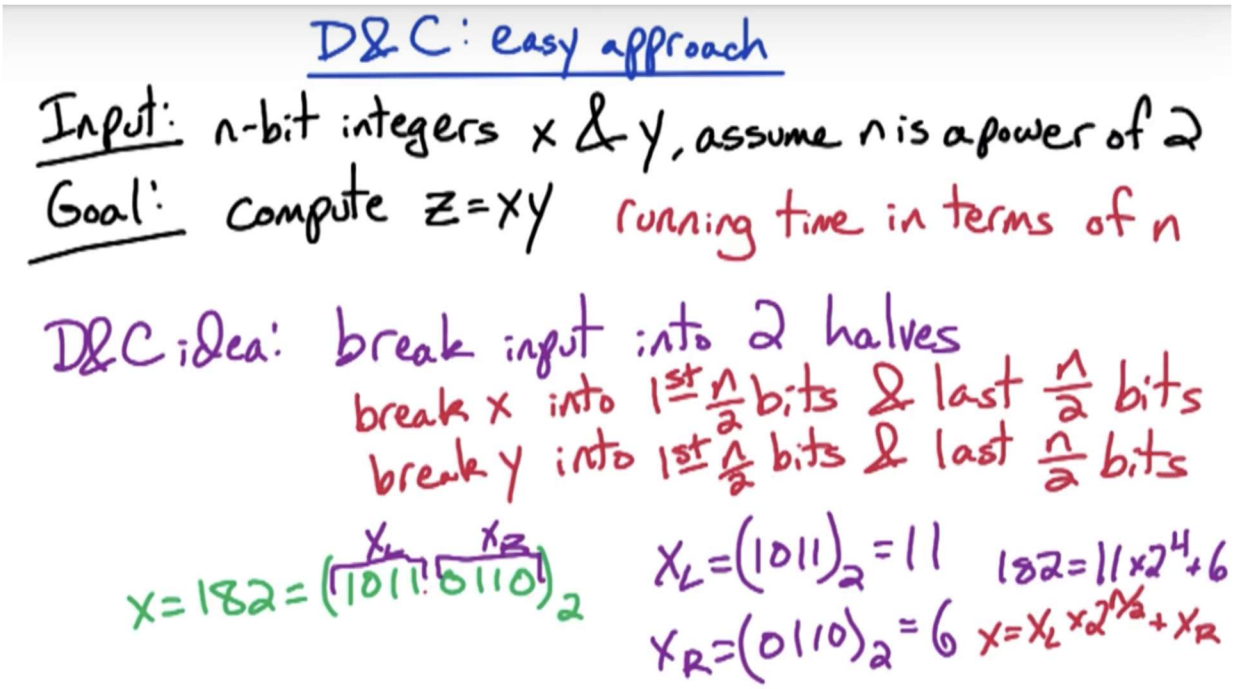
It turns out we're going to utilize this idea in order to multiply the n-bit integers faster than O(n^2) time. But before we dive back into that problem, I want to make sure that you appreciate the cleverness of this algorithm.

It's really quite amazing. I mean look at what you're computing here. You're taking (a + b)(c + d). Now, one of the complex numbers that you're inputting is a + bi. So what are you doing? You're taking the real part and the imaginary part and you're adding those up and that gives you a new real number, a + b. Why would you consider looking at this number, a + b, or this number c + d? Why would you consider taking the real and imaginary part of the same number and adding those up? Somehow by looking at the real plus the imaginary part for both of these complex numbers and multiplying those together that gives us a way for computing the product of these two complex numbers faster or at least using fewer multiplications.

Just think about it for a specific example. Let's say you're trying to compute (5 + 3i)(7 - 6i). What are you using here? You're using 8 x 1. So why would you think to use 8 and 1 from these numbers in order to compute the product of these two complex numbers? It's really quite clever.

(Slide 5) D&C: Naïve Approach

***



Let's get back to our original problem multiplying n-bit integers. And let's look at the straightforward divide and conquer approach for this problem.

The input to the problem are two integers x and y which are both n-bits long. And for simplicity, we're going to assume that n is a power of 2. This is a common assumption in divide and conquer algorithms. It allows us to get rid of floors and ceilings in our algorithm description and in the analysis of the running time. And our goal is to compute the product, x times y. We want to look at our running times in terms of n. n is the number of bits for these two integers x and y. It's the space required to represent these two numbers.

Now what's the standard divide and conquer idea? Think about mergesort. What do you do? You break the input, the n numbers that you're trying to sort, you break them into two halves. The left half and the right half. Then you recursively solve the problem on the two halves. So for mergesort, that means sort the left half and sort the right half and then you combine the answers together. You merge them together.

Now, how do we apply that same idea here for multiplying these two n-bit integers? How do we break the input into two halves? Well, we can't break it into x separately from y. So what do we do instead? We break x into two halves. The left half of x and the right half of x and similarly for y. x is n-bit number, how do we break it into two halves?

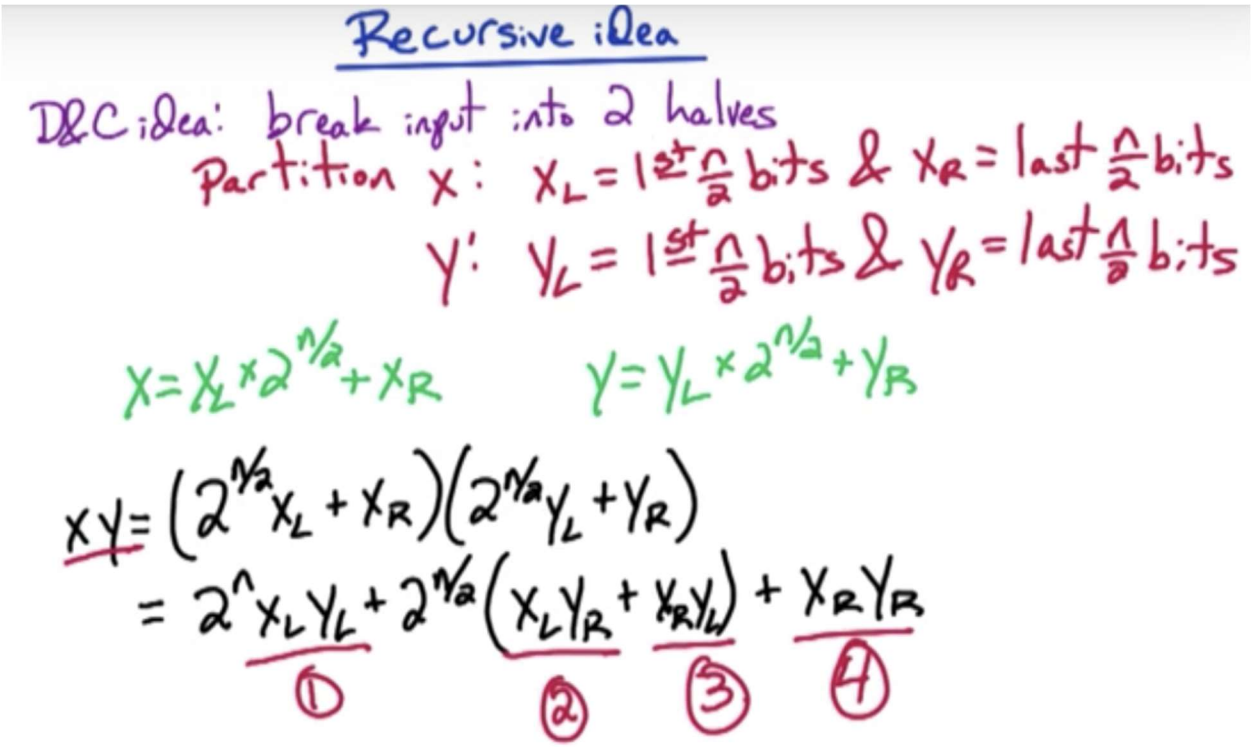
- Well, we look at the first n/2 bits and the last n/2 bits. And we take these first n/2 bits and we call that a new number, x_l. That's the left side of x. And we take the last n/2 bits. And that's another number, x_r, corresponding to the right side of x.
- Similarly for y, we do the same thing. Take the first n/2 bits and the last n/2 bits.

So we're going to break x into the first n/2 bits. That's going to be this new number x_l and we're going to break it into the last n/2 bits. That's going to be this number x_r. Similarly for y, we're going to break it into the first n/2 bits, call that y_l. And the last n/2 bits, call that y_r.

Let's look at this specific example to see what this partition of x into two halves is going to signify. Let's look at x=182. In binary, this is 1011 0110. This is 8-bits long. So we're going to break it into the first 4 bits and the last 4 bits. This is the first 4 bits, that's going to be x_l. The last 4 bits are going to be x_r. So x_l equals 1011 in binary. In decimal, that corresponds to 11, x_r is 0110 in binary which is 6. How does 182 relate to 11 and 6? Well, notice 182 is the same as 11 times 2^4 (=16) plus 6. And in general, x satisfies $x = x_l \times 2^{n/2} + x_r$. We take this number x_l and we multiply it by 2 to the n over 2. That corresponds to shifting it, n over 2 times and then we add in x_r. Which means add in these n over 2 bits.

(Slide 6) Naïve: Recursive idea

***



Let's follow through on our recursive approach. So our basic idea was to break the input into two halves. So we partition this n-bit number x into two n/2-bit numbers. So x_l is the first n/2 bits of x and x_r is the last n/2 bits of x. Similarly for y, we partitioned it to y_l, the first n/2-bit and y_r the last n/2-bit plus x_r. Notice we just saw how x_l and x_r are related to x. In particular, x = x_l x 2^n/2 + x_r. Note that 2^n/2 corresponds to shifting it n/2 times. Similarly for y, y = y_l x 2^n/2 + y_r.

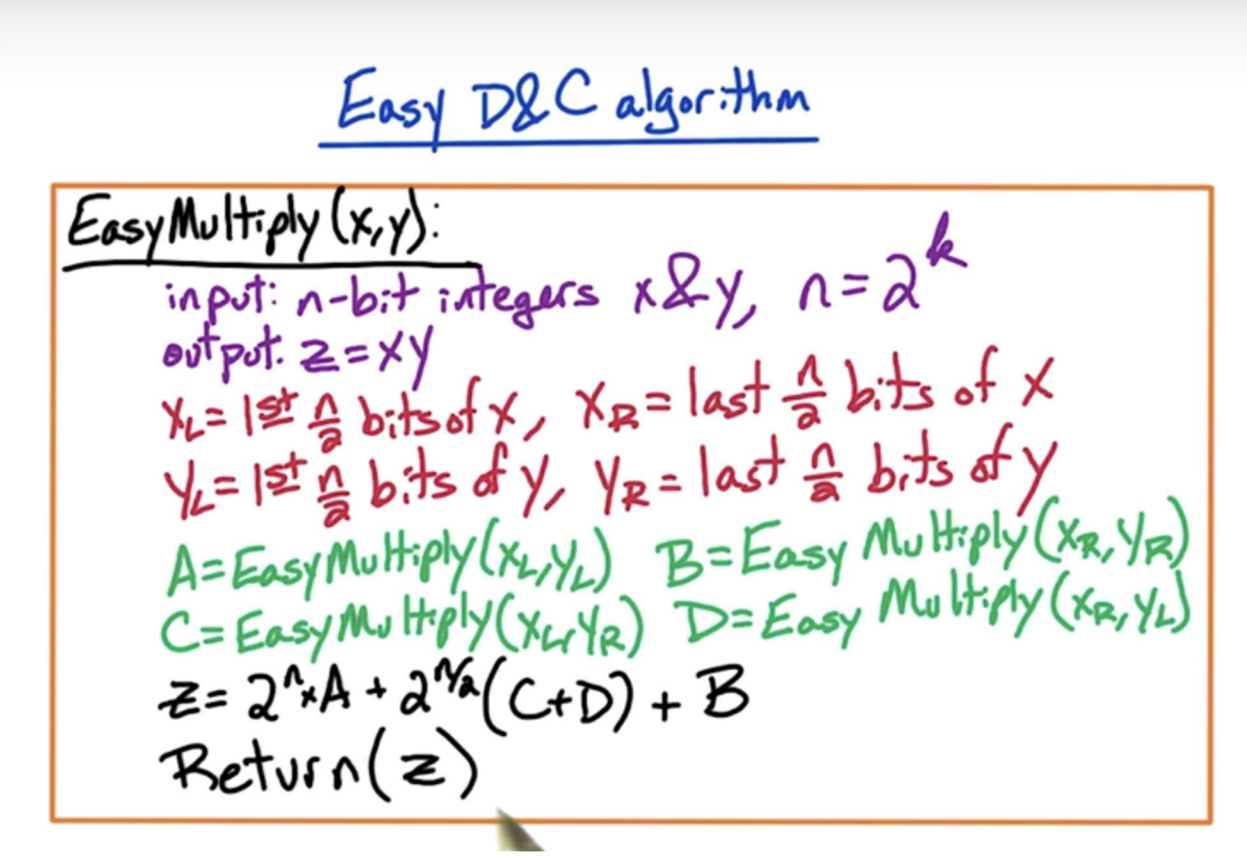
Well, the goal is to compute xy. Well, we can replace x by this quantity. X is the same as 2 to the n over 2 times x_l plus xr. And, we can replace y by this quantity (2^n/2 x y_l + y_r).

Now let's expand out this expression and see what we get. Multiplying the first terms together we get 2^n x_l y_l, we get two terms which are scaled by 2^n/2. Namely we get 2^n/2 times x_l times y_r and we also get 2^n/2 times y_l times x_r. Finally, the last term is x_r times y_r.

If you notice now we have a recursive algorithm for computing x times y. Now recall x and y are both n-bit numbers, so we're trying to compute the product of these two n-bit numbers. What we can recursively do, is compute the product of two n/2 bit numbers – well, we have four n/2 bit numbers over here. So we can compute the product of any pair of them. For example, we can recursively compute the product of x_l and y_l, and x_l and y_r, x_r and y_l

and x_r and y_r. These are four products of n/2 bit numbers which we can recursively compute. This gives us a natural recursive algorithm for computing the product of x times y. Let's go ahead and detail that algorithm to make sure everybody follows what we've done so far and then we are going to try and improve on that.

(Slide 7) Naïve: Pseudocode
***



Let's go ahead and detail this straightforward divide and conquer algorithm for multiplying x times y.

The input to the problem are these two integers x and y, which are both at most n-bits long. And we assume for simplicity that n is a power of two. So n=2^k for some non-negative integer k. And the output of our algorithm is this number z which is equal to the product of x times y.

First thing we do, is we break x and y into two n/2 bit numbers. So x_l is the number corresponding to the first n/2 bits of x. x_r is the number corresponding to last n/2 bits of x. Since n is a power of two, these always divide evenly. So we don't have to worry about putting any floors or ceilings on either of these two terms. Similarly, we partition y into the first n/2 bits, call that y_l, and the last n/2 bits of y, that's y_r.

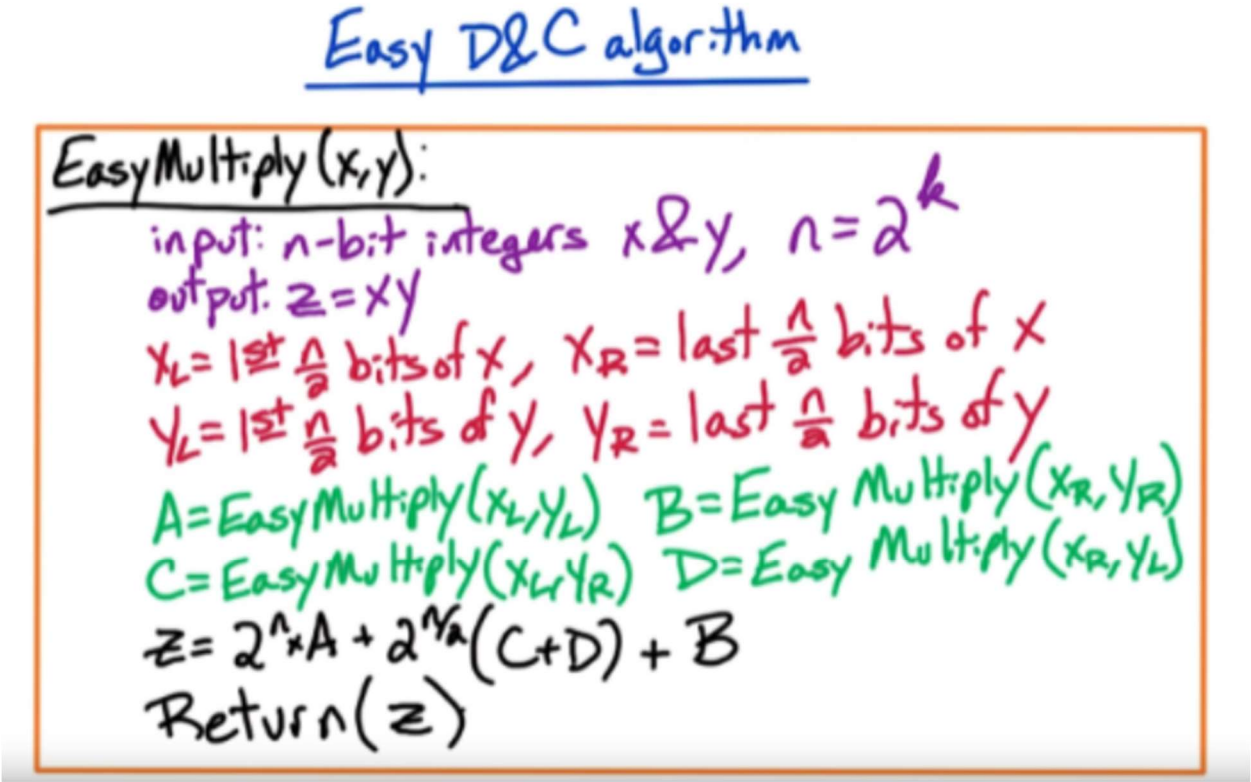
Now what we can do is we can recursively compute the product of pairs of these n/2 bit numbers. We have four n/2 bit numbers. We're going to take particular pairs, recursively compute their product, and combine those together to get this answer z. First pair of n/2 bit numbers that we're going to use is x_l and y_l. We're going to recursively compute their

product and we're going to store that answer in A. The second one we're going to use is x_r times y_r and we're going to store that answer in B. The last two recursive subproblems that we're going to do are x_l times y_r, store that in C; x_r times y_l, store that in D.

Now we can get the product x times y using these for quantities A, B, C, and D. In particular, Z = 2^n x A + 2^n/2 (C+D) + B … Z which is x times y, is equal to 2^n x A … that's a times that's not an X… + 2^n/2 x (C + D) (that's these cross terms, x_l times y_r and x_r times y_l) + B. We got this expression on the last slide.

Finally, Z is the quantity that we want to return. Z is equal to the product x times y. That completes the pseudocode for this easy divide and conquer algorithm. Let's take a look at the running time of this algorithm now.

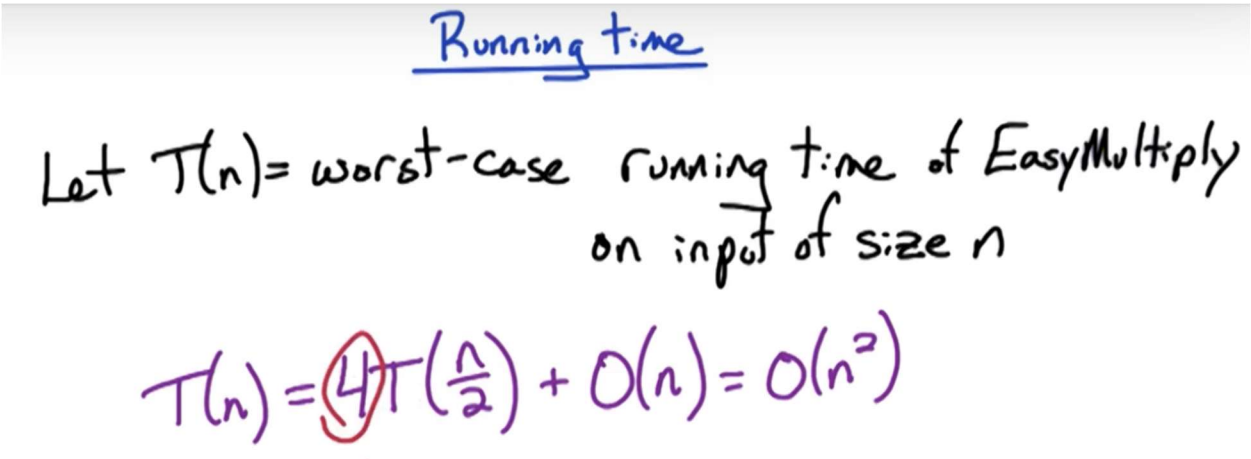
DC1: Fast Integer Multiplication: Naïve: Pseudocode



(Question) What is the running time of the above multiplication algorithm?

(Slide 8) Naïve Running Time

***



Let's go ahead and analyze the running time of this algorithm. How long does it take us to partition x into the two halves and y into the two halves? These steps take us O(n) time, in

order to break up x into the first n/2 bits and the last n/2 bits and similarly for y. How long does it take us to recursively compute these four products A, B, C and D? These are each pairs of n/2 bit numbers.

So if we use T(n) to denote the running time and the worst case for inputs of size n, then each of these takes T(n/2), and there's four of them. So the total time for these four recursive subproblems is 4 T(n/2).

Finally, given the solutions to these four recursive subproblems, how long does it take us to compute z? Well we have three additions of O(n) bit numbers. How long does that take? That takes O(n) time. We also have to multiply this O(n) bit number times 2^n. How do we do that? Well it's much faster than a multiplication which takes O(n^2) time. This is just a shift n times. In order to multiply by 2^n, we just have to shift this number A n times.

Similarly, to multiply this number C + D times 2^n/2 we have to shift it n/2 times. So it just takes O(n) time to do this multiplication by a power of 2. So to compute z, it takes us O(n) time. So the total time is O(n) + 4 times T(n/2).

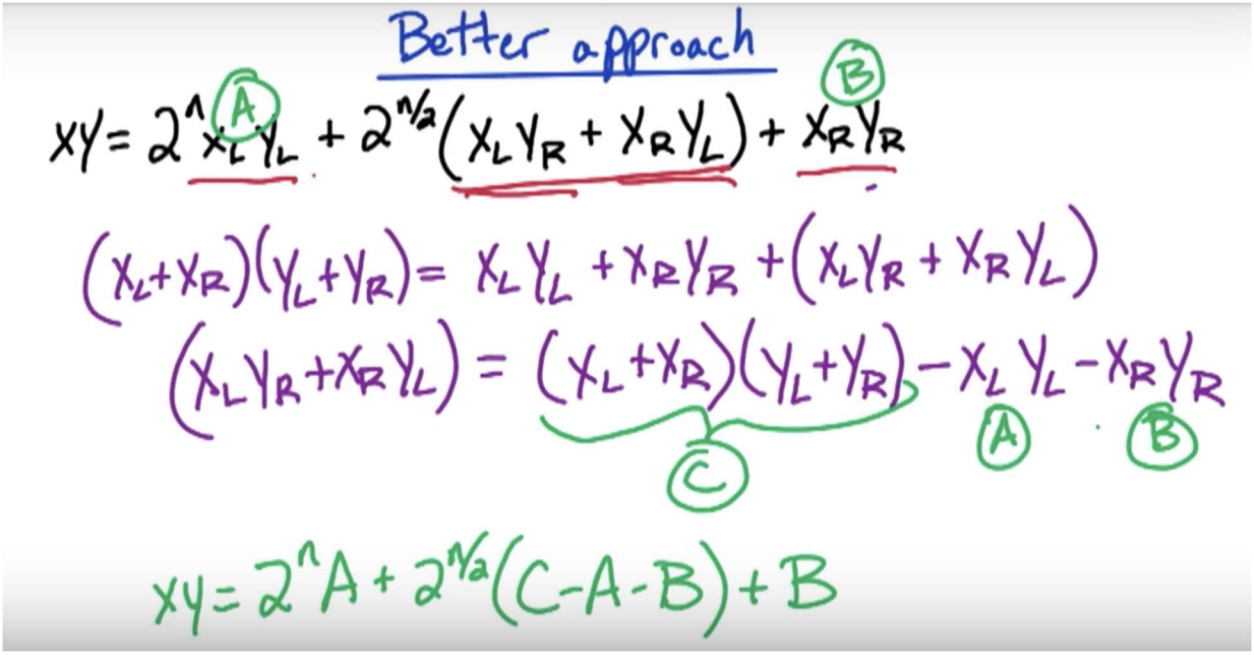
So let's let T(n) denote the running time of this algorithm EasyMultiply on input of size n and this is for the worst input. So we take the worst input of size n and T(n) is the running time on that worst case input.

We just looked at the running time of our pseudocode and we saw the T(n) satisfies the following relation. We have four subproblems of size n/2 each. And we take an additional O(n) time to combine these solutions together to get the product of x times y. So the running time for an input of size n, is at most 4 T(n/2) + O(n). Now if you remember how to solve recurrences or this is a good time to brush up on it, this recurrence might look familiar. And what you might recall is that this recurrence solves to O(n^2).

So, the easy divide and conquer algorithm that we just described takes O(n^2) time. So the running time of this divide and conquer algorithm is the same as of running time of the straightforward multiplication approach. Can we improve this? Can we do better? The key thing is, can we improve this four down to three? Can we get away with just doing three subproblems? This is where we're going to utilize the Gauss's idea that we talked about.

(Slide 9) D&C: Improved Approach

***



This is the key expression from before. x y = 2^n x_l y_l + 2^n/2 (x_l y_r + x_r y_l) + x_r y_r.

Now, a straightforward divide and conquer approach computes four subproblems recursively. It multiplies (see A in diagram) x_l and y_l, x_l and y_r, x_r and y_l and finally, (B) x_r and y_r. We have four subproblems and then it takes us O(n) time to combine those solutions together. This leads to an O(n^2) time algorithm.

What we're trying to achieve now is we're trying to reduce the number of subproblems from four to three. Thinking back on the idea from Gauss that we talked about earlier for multiplying complex numbers, what do we want to do here? We want to compute this sum, (x_l y_r + x_r y_l) without computing these individual terms. Well, how do we do that?

Well, we think of these as cross terms again. And on one side we're going to have x_l and x_r, that's one quantity. The other quantity is going to be y_l and y_r. When we expand this out, we get the following: (x_l + x_r) (y_l + y_r) = x_l y_l + x_r y_r + (x_l y_r + x_r y_l).

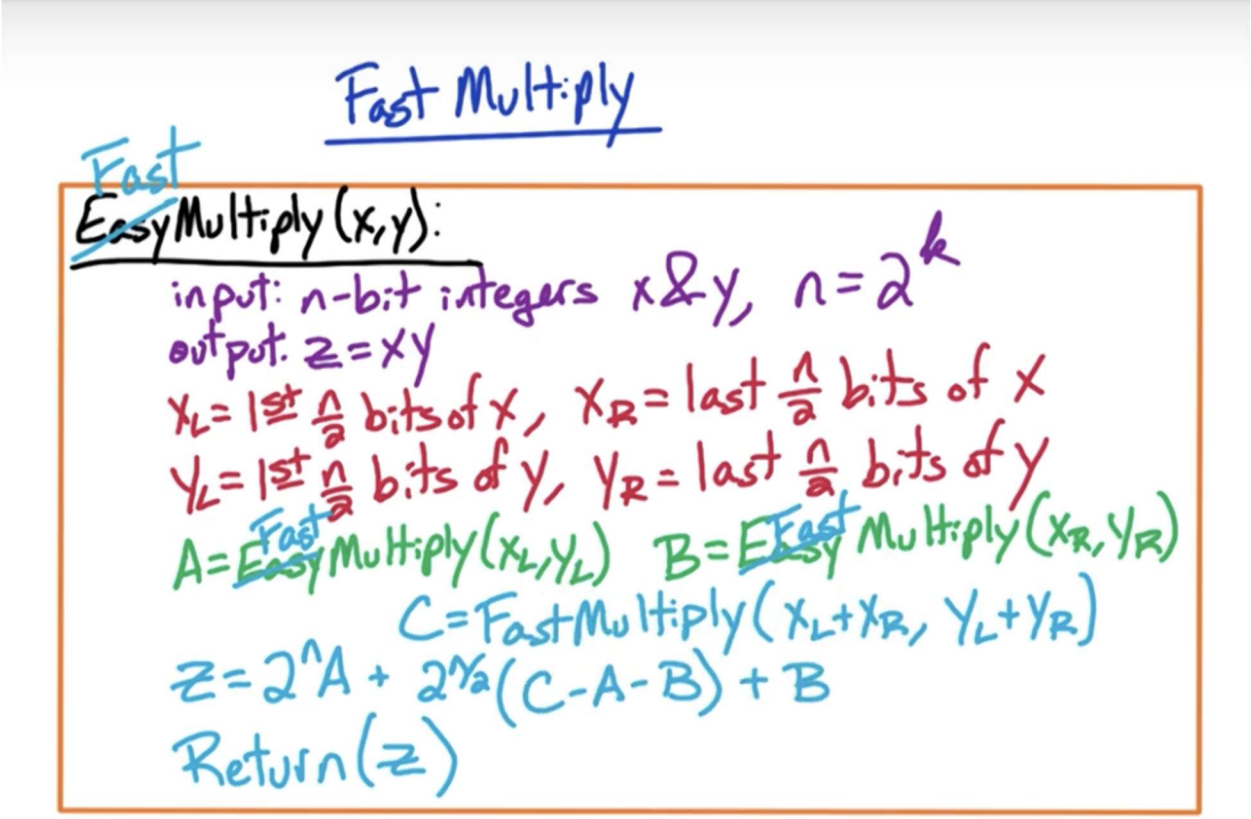
Now, our goal is to compute the sum, so let's solve for that. This quantity (x_l y_r + x_r y_l) … that's equal to this left hand side, (x_l + x_r) (y_l + y_r) and then we subtract off these two terms: x_l y_l and x_r y_r.

Now in the algorithm, we just had this term x_l y_l, this was the quantity A which we

recursively computed. x_r y_r was B which we recursively computed. So we have this term x_l y_l and we have this term x_r y_r. What's the final term that we want to compute? It's this quantity. So we're going to take (x_l + x_r) and we're going to multiply that by (y_l + y_r) And we're going to store that answer in C. We're going to recursively compute A, B, and C and then we can combine them together using this expression to get x times y.

In particular, x times y = 2^n A + 2^n/2 (C – A – B) + B. And this is going to give us a better divide and conquer algorithm because now we only have three subproblems that we have to recursively compute. And then we can combine them together using this expression in order to get the product of X times Y.

(Slide 10) Improved: Pseudocode
***



Let's go ahead and detail the pseudocode for this faster multiplication algorithm. Now this is the algorithm from before. This is the O(n^2) algorithm, which utilized four recursive subproblems. This new algorithm is fairly similar. It's just that it differs at the end. Just this last bit is different.

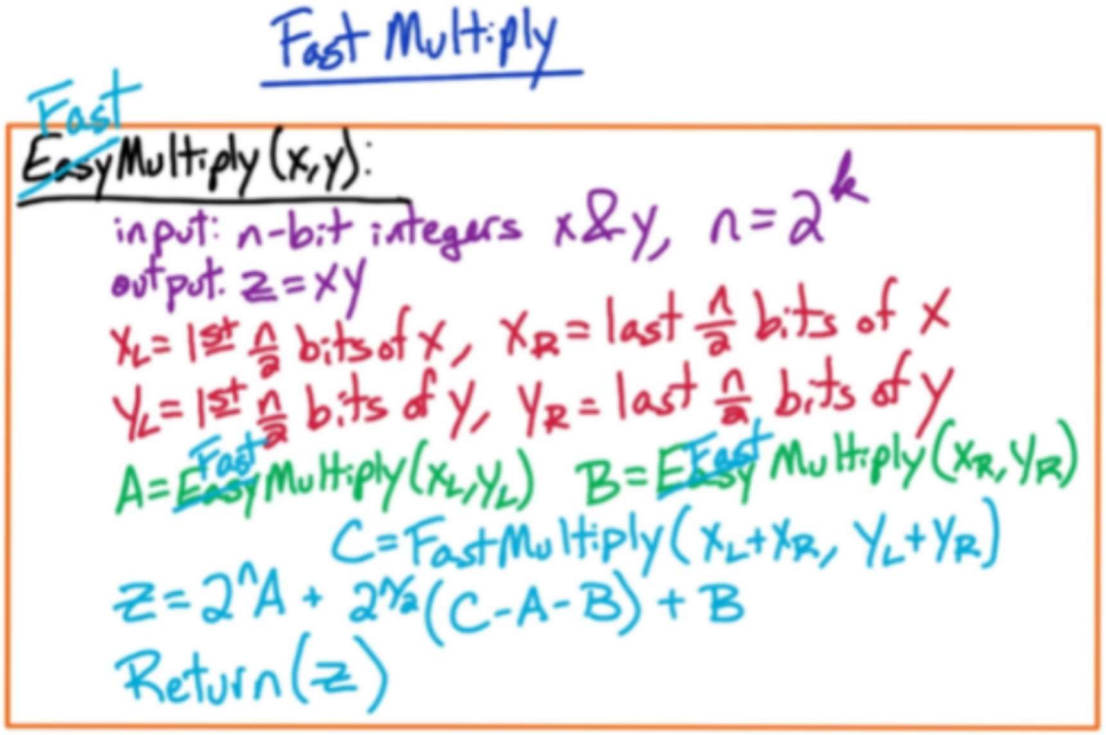
So let's get rid of the last bit of the algorithm. Let's change the name of this algorithm from EasyMultiply to FastMultiply. Basic set up of the algorithm stays the same. The input is two n-bit integers x and y, and the output is the product. We partition the input x into the first n/2 bits and the last n/2 bits that's x_l and x_r. And we also do similarly for y. We partition it into y_l and y_r.

Now we're going to recursively compute the product of three pairs of n/2 bit numbers. The first pair, as before, is x_l times y_l. The second pair, as before, is x_r times y_r. We are going to store those in A and B, as before. Now the new term is we're going to compute x_l + x_r and we're going to multiply that by y_l + y=r and we're going to store that in C. This is where we're utilizing Gauss's idea. Now we utilize the expression from the last slide. Z, the product of x times y is equal to 2^n times A which is x_l times y_l + 2^n/2 times the quantity C - A - B.

This is where we are utilizing Gauss's idea. Finally, we add in B. Then we return Z, which is equal to the product of x and y.

This completes our algorithm, and the key fact is that now we have three subproblems that we're recursively computing.

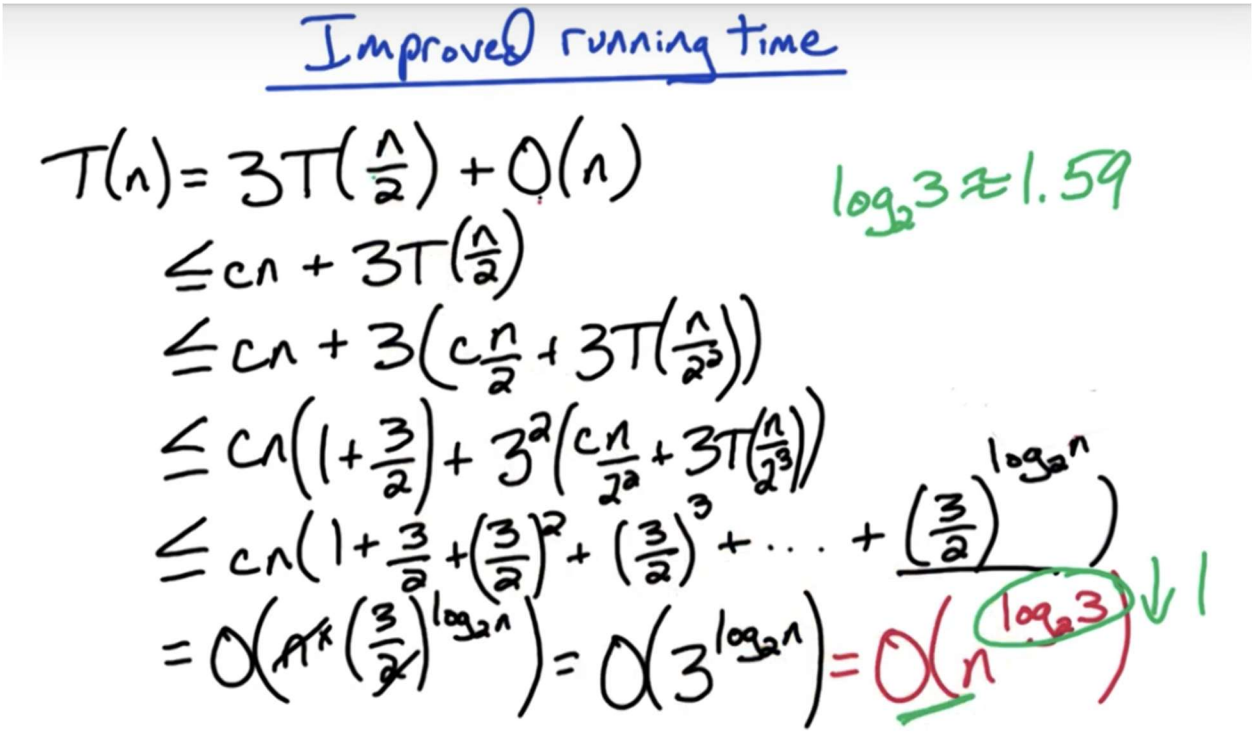
DC1: Fast Integer Multiplication: Naïve: Pseudocode



(Question) What is the running time of the above multiplication algorithm?

(Slide 11) Improved: Running Time

***



Now if we look at the running time of our new algorithm, we have three sub-problems that were recursively computing. Each is a product of pair of n/2 bit numbers. And then to combine these solutions from these three sub-problems it takes us O(n) time to get the product of X*Y.

What does this solve to? Well, let's go ahead and dive into it to give you a bit of a refresher on solving recurrences.

The first step is upper bounding this O(n), by cn for some constant c. Now we can substitute in this expression back and forth, T(n/2). So we get cn + 3 times… Now we are substituting for T(n/2), which is c n/2 + 3 (T(n/2^2.). Collecting terms, we have, cn , plus another cn …3/2. Then we have a 3^2 x $T(n/2)^2$. Plugging that back in, we get $cn/2^2$ from that term, + 3*T(n/2^3). Collecting terms we have cn x (1 +3/2 +$(3/2)^2$ + $(3/2)^3$ + ⋯ + $(3/2)^{\log 2n}$). The next term is going to be $3/2^3$, and so on. We're going to get this geometric series.

How many terms in this geometric series, what's the last exponent? We're going to keep going until this term is a constant. There are going to be log2n terms. Now we have this geometric series. We got to look at the series see whether the terms are equal, they're not. Is it decreasing geometric series, in which case the first term dominates? It's not. It's an increasing geometric series, because three halves are bigger than one so the last term dominates. This whole thing is

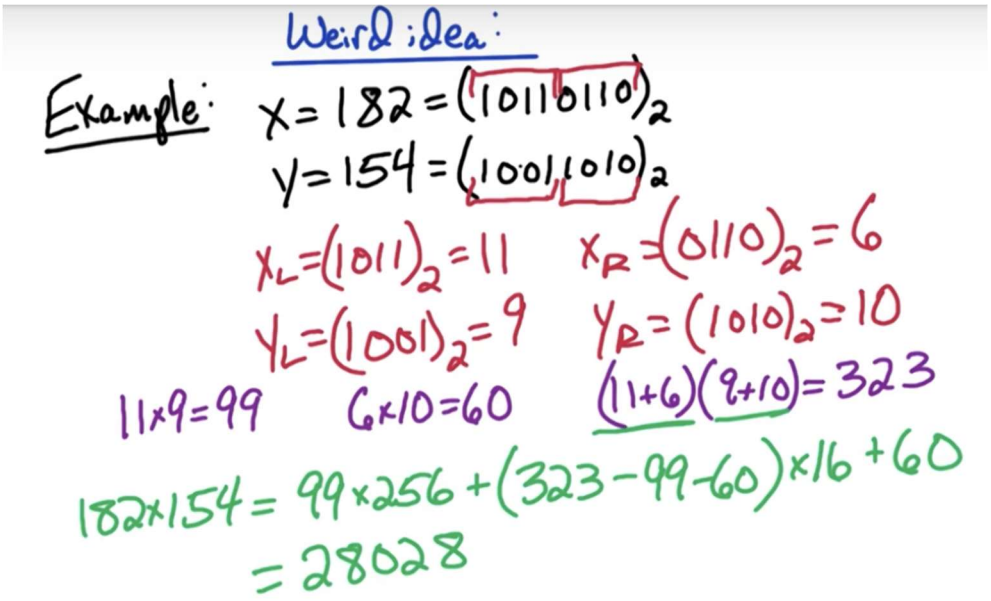
on the order of the last term. We get $O(n \times (3/2)^{\log_2 n})$. Now this $2^{\log n}$ is the same as n, so those cancel. We're left with $O(3^{\log_2 n})$. Let's convert that into a polynomial.

You should remind yourself how to convert this into a polynomial. Let me give you a quick reminder. That's the definition of log. So I have this expression, $3^{\log_2 n}$. And I can replace 3 by this expression. And then I raise that whole quantity ^log2n. Now these two exponents multiply together, so I can swap them. This quantity is the same as, $(2^{\log_2(n)})^{\log_2(3)}$. Now these two things are the same. Now, this is much simpler. Because what is 2^log2n? This is simply n. Now, I have n raised to a power, which is a constant, log2(3). this solves to $O(n^{\log_2(3)})$. What is this number $\log_2 3$? Well, if you plug it into a calculator, you see that $\log_2 3$ is roughly 1.59. So we went from an $O(n^2)$ algorithm, to an $O(n^{1.59})$ algorithm.

And in fact, we can improve this exponent. We can get arbitrarily close to 1, but there's an expense for that. This constant hidden in the Big O notation is going to grow as this exponent decreases. And instead of breaking the input up into two halves, we're going to break it up into more parts, and then we're going to have to work harder in order to combine the solutions together.

(Slide 12) Improved Summary

***



That completes the description and the analysis of our algorithm. But before we move on, I want to look at a particular example to illustrate the cleverness of the approach.

So, let's take this example. x equals 182 and y equals 154. If I write 182 in binary, it's 10110110. And 154 in binary is 10011010. Our approach breaks up x into the two halves, x_l and x_r. And similarly, we have y_l and y_r. x_l is 1011, which is 11 in decimal, and x_r corresponds to 6, y_l corresponds to 9 and y_r corresponds to 10. Now what our algorithm does is it first computes x_l times y_l, which is 11 times nine which is 99. Then we compute x_r times y_r which is 6 times 10, which is 60. Finally, we get the non-trivial weird idea. We take x_l + x_r, which is 11 plus 6, and we multiply that by 9 + 10. That gives us 17 times 19, which equals 323. How from this number 182, which can be broken up into 11 and 6, somehow we're working with 17? We combine these two, then we get 182 times 154. That equals 99 times 2^n. In this case, n equals eight. So, 2^8 is 256. Then we take 323 minus 99 minus 60, and we multiply that by 2^4, which is 16. Finally, we add in the last term, 60. And if you plug that into your calculator, that exactly equals 28,028, which is exactly 182 times 154. And the amazing part is this 17 and this 19. How do we get those? That's a very non-intuitive part.

That completes the description of this faster multiplication of n-bit integers. There are similar ideas for multiplying matrices, that's referred to as Strassen's algorithm. You can look in the textbook to learn about Strassen's algorithm. I'm going to skip it because algebra gets a bit messy for that. Next we're going to look at linear-time median algorithm. It's a very clever divide and conquer approach.