YouTube Summarizer

Transcript:

Hey guys, Greg here at what's solve add binary, leak code number 67. So we're given two binary strings A and B and we need to return their sum as a binary string. So let's say that we had A is 1, 1 and B is 1. If you were to add those together, you would get the result of 1, 0, 0. So in decimal, this number would be 3. This number would be 1. So you'd get 3 plus 1, which is equal to 4. And this is how you'd represent 4 in binary. Okay, with these numbers here, we have 1, 0, 1, 0, which would be 10 in binary. We have 1, 0, 1, 1, which is 11 in binary. So we should be getting 21. Here we get 16 plus 4 plus 1. And so this is the binary string for 21. So let's say that this is A and B. What would these numbers actually be? This would be 8. And then we'd get 4, 2 and 1. If you add those together, you're going to get a total of 15. And B is just going to be 2. So ultimately, of course, you would want the normal result of just 17. Which in binary would be represented as 1, 0, 0, 0, 1, because you're going to get a 16 and a 1 to make 17. So one technique would be to implement extremely normal binary addition, which would just be to do this. You would take the two numbers and add them. And you would just work out how addition works normally. You would take the 1 and the 0, you'd get 1. You'd take 1 and 1. That is going to make 0 with a carryover of 1. This would make 1 and 1, which makes a 0 with a carry of 1. Again, 1 and 1 makes 0 with a carry of 1. So that's your answer right there. And you could absolutely implement that. However, a more interesting version of the problem actually bans you from using addition at all. Basically, meaning you would have to do this with just bitwise operations. And this next solution, there is probably 0 way you're going to come up with this yourself. So feel free just to watch this and I'll explain it as we go. So the idea behind this technique is basically using two different variables without the carry, as well as just the carry itself. Basically, the point of addition is you keep in mind the carry. So while you're summing these things, while you take 1 and 0 to get 1, and then you do 1 and 1, which makes 0 with a carry of 1, this whole process is basically just combining these two things together, working without the carry, as well as keeping the carry in mind. However, it uses addition. To not use addition, we can actually just think about these two things individually via some very clever bitwise operations. So without the carry part, that actually turns out to just be an XOR, as we'll see. So this is a bitwise XOR. We would get 1 here, 0, 1, and 1. So this would be the sum without the carry. So we put that here, 1, 1, 0, 1. Why is this the sum without the carry? Well, think about it. If we take 1 and 0, we get 1. If we take 1 and 1, we would get 0 with a carry of 1. But we ignore the carry, and we just get 1 and 0 makes 1, 1 and 0 makes 1. And so this is that sum. But you do still need to keep in mind the carry. So to do that, to get the carry, it turns out it is actually an ampersand, so a bitwise and. And then after that, you left shift 1. You probably said, how would I come up with that? I have no idea how you would come up with that. That's just what it is. So if you were to do that here, you do an ampersand between these two. You're going to get 0, 1, 0, and 0. The left shift 1 is then going to shift this over. So it's going to move these over like this. And then we plug in a 0 at the beginning. This is exactly what that carry is. You take 1 and 1. You're going to get your carry of 1 here. And so it is simply just that carry spot there. So this is our carry 0, 1, 0, 0. After you figured out these two things, we then update both of these. So a is going to get updated to the without carry portion. And b is going to get updated just to be the carry portion. And then you just rinse and repeat this. So what we're going to do is take those two numbers here. We take 1, 1, 0, 1. We're going to x or that with 0, 1, 0, 0. That is going to give us 1, 0, 0, and 1. So here's our sum without the carry. Again, 1 and 0 makes 1, 0 and 0 makes 0. 1 and 1 makes 0. But then ignore that carry. So 1 and 0 makes 1. So that is going to get placed down here, 1, 0, 0, 1. We're then going to ampersand these two things together. And then left shift 1. So this would be our carry. That would cause this to be 0, 0, 1, and 0. We would left shift that. And so our carry gets moved over to here. Again, let's see why that is the case. 1 and 0 makes 0. There's no carry, 0 and 0, no carry. 1 and 1 makes 0. And then you carry that over here to 1.

We are simply just looking at the carry. And so that's a carry of 1. OK, so we update our carry to be 1, 0, 0, and 0. We update these two numbers over here. So this is going to be 1, 0, 0, 1. This is going to be 1, 0, 0, 0. So we get 1, 0, 0, 1. We are going to explore that with 1, 0, 0, 0. This will give us our sum without the carry, which is going to be 1, 0, 0, 0. So that updates this to be 0, 0, 0, 1. To get the carry, we are going to do our bitwise and left shift by 1. That gives us 0, 0, 0, and 1. And then we left shift that 1. Now what happens here is very tricky. Imagine here if we did have more space. And you most likely would. We're probably using something like 8-bit integers. So naturally, they would probably look something like this instead. So this isn't actually going to just get pushed off. This is going to get pushed over like this. And basically creates space for a new digit. So this actually makes this new carry to be 1, 0, 0, 0, 0. And so we'll update these things here. So this is going to be 0. I'll give it another padded 0. So it can match here. 0, 0, 0, 0, 1. And then 1, 0, 0, 0, 0. So let's do this again here. 0, 0, 0, 0, 1. And 1, 0, 0, 0, 0. If we x-order those two things together, this time we are going to get 1, 0, 0, 0, and 1. OK, so our without carry is going to be 1, 0, 0, 0, 1. And then when you get the carry here, something wonderful happens. Amperstand that. And then bitwise, move left. 1, here we are going to get 0, 0, 0, 0, and 0. And whether you left shift that or not, we are still going to be left with a result of 0. So here, our carry is then going to be 0. We're going to update both of these. So this is going to be a. And then this is going to be b. And we actually stop this when our carry is 0. Because we're trying to add the without carry with the carry. So this means that we're done. Ultimately, we end up with the result of 1, 0, 0, 0, 1, which is actually our desired number of 17 and decimal. Now it's kind of hard to think about the time complexity here. But it turns out to be big O of A plus B, where A is the number of digits in A, and B is the number of digits in B. That's because it's kind of simulating the same sort of addition of just sliding over to the left and bringing your carries here. And the space complexity of this is pretty much constant. We don't really need to store anything. Now, A and B are given as binary strings. But we're going to convert them to decimal just for the sake of doing our bitwise operations. So we'll do A and B is equal to the int of A and 2. And int of B and 2. So all this does is sets A equal to the decimal version of A and B to be the decimal version of B. Okay, then what we do is while we have B, so basically while there's no carry, we need to get our without carry portion is equal to the x or of A and B. So that's A, carrot B. And then the carry portion is A and B. So bitwise A and B. And then you need to left shift that one. And then we can update A and B at the same time. A is going to be without the carry. And B is going to be the carry itself. If you keep doing this operation, eventually that will do our sum. That's going to be stored as a decimal number. So we need to return the binary of A. And then you would need to get rid of this garbage that kind of shows up in Python. You need to write 2 colon so that you skip the 0B part. Now, as we saw, the time complexity of this is going to be big O of A plus B. Because as with normal addition, you're basically just going to go across those strings. And I would argue that the space complexity is constant. Because as you can see, we're not really storing anything. Now again, this question is absolutely brutal. So don't kick yourself if you didn't get that solution. I hope this was helpful guys. Check out AlgoMap.io in the description if you haven't already. And have a great day guys. Bye bye.

Summary

Problem: Add two binary strings A and B and we need to return their sum as a binary string . One technique would be to implement extremely normal binary addition, which would just be to do this . However, a more interesting version of the problem actually bans you from using addition at all . To not use addition, we can actually just think about these two things individually via some very clever bitwise operations . To do that, to get the carry part, that actually turns out to just be an XOR, as we'll see .