Introduction to Cybersecurity and Recursion

Video Transcript

Video 1 – Agile Methodology

So, let's take a look at what is called Agile. It's a development methodology. Now until now, you've been working on your own learning how to debug code and how to complete example exercises. But when you work in industry, you'll be part of a team and you'll be working with others and that has its advantages and also its challenges. And originally, most of industry had been using what was called the waterfall method of developing software, which had intensive planning and would layout milestones for years at a time. But gradually, people recognized that a lot of things were changing in the software world. We had cloud computing coming along, DevOps, other things that had a much faster clock tick, and that in those environments, we didn't have the experience that would be necessary to do waterfall.

And so, in 2001, over a dozen people that we're experts in software development got together in Snowbird, Utah and they came out with these principles, there were 12 of them. And these were much more realistic about how software was being developed at the time and as they thought how it should be developed. So, for example, the highest priority is to satisfy the customer. If you don't satisfy the customer, you're not going to stay in business. But that we should have early and continuous delivery of valuable software. So, we shouldn't wait three years for a software project to be completed. Also, we should welcome changing requirements even if it's late in the game. Agile processes harness change for the customer's competitive advantage, that's the principle. That the customer may have new needs and you need to take those into account.

So, go through all 12 of these. They're addressed though, at how do we operate in an environment where we may not have developed the software before? So, we're doing something new. And also the customer's requirements may be changing as we go along. Take a look at these 12 principles and along with those, they specified four values, so that they would value individuals and interactions over processes and tools. Would value working software over comprehensive documentation. Would value customer collaboration over contract negotiations and would value responding to change over following a plan blindly. So, this is all about how to operate in a changing environment. Now, you'll find that there's quite a bit of language associated with Agile, and if you go to the Agile Alliance, they have a glossary of terms.

But you'll hear about user stories and you'll hear about the three C's; card, Conversations, and Confirmation. These user stories, for example, imagine that you want to book an airline ticket. So, here's a real-world problem. Now, you have the user story of your going online to book your ticket, and it seems at first fairly straightforward but then when you look at the real problem, there are things like upselling and cross-selling. You're trying to book, for example, an economy class seat and they may be able to get you to use your air miles to upgrade to business class. Or they could be cross-selling. You might be sold a hotel and car rental to go along with your trip and what happens if you want to change the time of your flight or your seats, or you want to cancel and get a refund or a credit.

So, all of these things enter this user story. Different frameworks that have been developed in Agile. One of them is called Scrum. Scrum has things called sprints, and the sprint usually lasts less than a month. And the idea is to break down the problem into bite-sized pieces so that we can go through something in one month where we plan, design, build, test, and review, and launch, that we actually get working code, and one of the principles is to value working code over documentation. So, we're always trying to trade off scope of the project against the quality of what we deliver and the cost of the project against time. These are the tradeoffs and the usual joke is, you can usually have any two of three of these, but not all four. T

his has led to the environment that you'll be going into in a company. So, you should read more about these and as you see here, they've basically incorporated various, what they call tribes. So, these are different methodologies that you come across, such as Extreme programming or DevOps and Scrum here is this red line. But product management have other concerns and here you will see the green line which is DevOps, the purple, that is the Design considerations, and then Testing here on the far right. So, all of these things are part of what is called Agile. Typically in your company, not all these things will be implemented. They'll have their own flavor of what they think works best for them. But you should certainly be aware of Agile.

Video 2 – Introduction to Cybersecurity

So, in this module, we're going to take a look at cybersecurity. We're going to take a look at the basic building blocks and one of the most important is what we call the Public Key Infrastructure, PKI. And PKI produces two keys, a public key, and a private key. The public key can be shared, but the private key must be kept secret. And we're going to use what they call signing of documents as an example of how we can combine encryption and hashing. So, we'll go through exactly what a hash is. And we'll show you a demo of using what we call block keys to represent a hash of visualization, if you like, of what a hash is and we'll be combining those in signing a document. So, when you sign a document, you basically are assuring the person that you sent this document and also that the document hasn't been changed. So, these are the building blocks of cybersecurity, and you should understand that they exist and how to use them.

Video 3 – Cybersecurity - Encryption, Kerberos, And PKI

So, let's take a look at authentication and authorization. Authentication answers the question, who are you? So, I need to know who I'm dealing with, for example. And authorization answers the question, what are you allowed to do? So, we're going to be using symmetric encryption. And that basically, just says we're going to use the same key for encrypting as we are for decrypting. So, we're going to share the key, for example, between two users. I could encrypt Hello World and get c935af. I'd send that to you. An attacker would not be able to decrypt it, but you can using the same key. Now, the typical situation that we are going to be dealing with in a corporation is where we have multiple users that need to use a number of services.

Now, if we store the usernames and passwords on each service separately, it means that it becomes an N-squared problem. So, we have, you know, a total of 16 combinations there. Now, we can get around this N-squared problem or the concept of a third-party service. And in this case, we're going to use Kerberos as an example. And we'll just have an email server, and we'll have Alice as our main user and the attacker that's monitoring the network. So, we're going to assume that we don't trust the network to be secure. So, anything moving over the network can be copied and replayed. We'd like to have single sign-on in the sense that we don't want to have to resign in to say, the email server every time we want to get our email during the day.

We'd like to sign-on the morning and be able to retrieve email during the day without having to sign-on again. We shouldn't store our passwords on any machine, and we're assuming that passwords should never be passed in clear text on the network. Let's take a look at the Kerberos server. It's going to store the passwords of all the users and each services while it's going to have the equivalent of a password. We'll combine the password just in case, there's a collision with the username to make it unique. And we'll assume that every user has the secret key. Here, Alice has a secret key, Akey, the email server has a secret key, Vkey. And so, those keys are going to be used to encrypt messages. Let's take a look now at how we're going to do this.

There's a concept of a shared secret. So, here we have the user in Kerberos have this same shared secret, and they can encrypt messages. So, Kerberos knows if it gets a message from Alice, and if Alice's secret key decrypts it, then the service knows that it came from Alice. Similarly, Alice, if she gets a message and she can decrypt it with her secret key then she knows it's come from Kerberos cause that should be the only other entity that has her password. So, encrypted messages couldn't be copied by an attacker, but not decrypted. So, a message could be replayed, for example. Now, there's a great algorithm called Diffie-Hellman for key exchange. So, how do we set up these keys, so Kerberos has my key, my password, and the secret key that it derives from the password?

And obviously, I got it as well. So here, Alice and Bob have a common paint. And it doesn't matter whether an attacker knows what that color is or not because they both add secret colors now. So, Alice adds red and Bob adds blue. And that results in Alice having a brown color paint and Bob, blue. Now, they swap those. And Alice now adds her secret color to Bob's mixture. And Bob adds his secret color to Alice's mixture. And they both get a common secret because they both added red and blue to yellow. But they've done it in different orders. And we're assuming here that when they pass the brownish color and the blue over the network, that those can't be separated back to their original colors. That would be a difficult problem, for example.

But that allows us to get the, to share the passwords if you like, and get them on to Kerberos in a secure way. So, that's usually called a secure channel. Now, let's suppose Alice needs to sign-on, and her goal is to, ultimately, get to the email server, but to allow her not to have to enter a password all the time. We're going to give Alice what we call a ticket to granting ticket, a TGT. So, Alice requests a TGT and she only passes her username, and she does that in cleartext. And so, the attacker can certainly see that, and a request for a TGT ticket. And Kerberos now looks up, Alice retrieves her secret key and issues a TGT, and encrypts it using Alice's key.

So, now that message is sent back to Alice. She's got this TGT, and she can decrypt the message and get TGT, and stores it in her tray. Now, every other time she goes back to the Kerberos server, she can pass this ticket and Kerberos will recognize it. How does it do that? Well, it encrypts something that it knows with its master key. And so, it doesn't really matter what's in the TGT, but the Kerberos server can recognize it, that it's issued it because it can decrypt it. Okay. So, now Alice has a TGT, she needs to go on and get the email. So, those are the interchanges so far. Now, that the TGT can last for up to eight hours, so a normal working day, for example.

So, now Alice needs to get access to the email server. So, what she does, now she passes a request for a ticket to the email server or a token, security token for the email server plus this TGT back to Kerberos. She basically passes the service that she wants, the email service plus this TGT. Now, Kerberos recognizes the TGT so it knows that it's legitimate, a message coming in. And it issues it, an email token encrypted with the email services secret key. I've called it Vkey here. The email token is passed back and is stored on Alice's machine. Now, Alice can send that token to the email server. And the email server can decrypt it because it's in possession of its own secret key, the Vkey.

And so, it knows the message is approved by Kerberos. But there's a problem. What if the attackers stole this message? So, here I'm showing the attacker, and it's got a copy of that Vkey encrypted message. And suppose the attacker spins up a machine and takes the username Alice, and passes this token now to the service. So, the email server now needs more information than we've given it so far. So, suppose we add the IP address. Will that stop the attacker? Well, no, the attacker can fake the IP address as well. And so, we need something more again. So, let's go back and design our system slightly differently. So, now we're going back to Alice making a request for the TGT. Now, in this case, Kerberos now invents a sessionkey, and we'll talk about this sessionkey, but encrypts it with Alice's and sends it back.

So, sessionkey1 is send back to Alice. She can decrypt it and put it in her tray. But also sends back the TGT. And I'm explaining what TGT is, in this case. It's encrypted with the master key, and it's the username, address, lifespan, and sessionkey. What does Alice do? Well, now when she makes a request for the email server access, she sends two messages, I call them C and D. She passes back the TGT and also the name of the service that she wants to access, so the email service. Now, she encrypts with the sessionkey that was passed to her username and a timestamp. So, Kerberos now can use the fact that the sessionkey has been encrypted and can decrypt it. It's got the TGT from Alice, and so it knows exactly who she is.

And now, it sends back to messages to Alice. It creates using the email server's secret key, the Vkey. And it sends back a session 2 key, and also username, address, servicename, lifespan, and timestamp. And sends back to Alice with the sessionkey1, encrypting it, sessionKey2 So, sessionkey1 is known to Alice already, so she could now get sessionKey2. Now, what Alice does is she passes the first message as it was, encrypted with the services Vkey. But she encrypts using sessionKey2, her username, and timestamp. And service can decrypt message (E) because it's encrypted using a secret key and can retrieve sessionKey2. And also the username address, servicename, lifespan, and timestamp.

And can now decrypt message (F) and get the username, and timestamp that this message has come from, and knows that it's Alice. So, it knows that the first message (E) must have come from Kerberos. So, it knows that that's approved by Kerberos, and knows that also that the username and timestamp have come from this person that sent the message. So, I'm just laying out here the full interchanges. And we'll step through in an exercise and look at exactly why these steps are needed.

But this is how typically these kinds of interchanges work. Every message needs to prove who it's come from. So, here we're using these sessionkeys and the second sessionkey, session, sessionKey2, for example, can expire quite quickly. That we can arrange that it expires in minutes rather than hours. So, certain sessionkeys will be short-lived, and certain sessionkeys can be longer lived but this will achieve what we want. We'll go through in more detail how authentication and authorization are used today with APIs and more complex environments.

Video 4 – PKI Keys

So, let's take a look at how we use keys. So, we're going to generate some OpenSSL keys and we're going to show first how to sign a document. And then we'll look at how to encrypt a document and also sign it. Okay, so first we need to generate a key. To do this, we're going to use OpenSSL. So, here's my OpenSSL. And what I need to do is to first generate a key. So, let's generate a key. So, we use 'openssl genrsa' we're going to use an RSA key of '512' bits, and we're going to output it to a file called 'myprivate.pem'. So, this is going to use a standard called pem. We'll see later that we may have keys using a different standard, but this one generates a key called 'myprivate.pem'.

Let me just 'cat' that. So, that's what it looks like. 'BEGIN RSA PRIVATE KEY' and then 'END RSA PRIVATE KEY', okay? So, that's our private key. Now what we need to do, let me clear. We need to generate from that our public key. So, let's do that. We're going to use our private key to output our public key. So, let's take a look now we should have our public key. So, here's our public key. And you can see it's typically shorter than the private key. So, now we have our two keys. Let's sign a document. So, the document we're going to sign, I've already got is a file called 'hello.text'. So, it just says 'Hello World'. So, we're just going to sign that file. Notice this one is not encrypted. And basically what we're going to do if we go back to our diagram.

So, what we're going to do now is take this file and we're going to create a hash code that needs to be padded. But then we're going to use our private key to encrypt that hash code. And so, that's called our signature. Now the only way to decrypt is with our public key, which is widely shared. So, what we're going to do then is we're going to share the file with another person and we're going to share our signature. And they've already got our public key, which is widely distributed. So, they apply the public key signature, decrypt it, they get the padded hash. They then get the unpadded hash.

Now because I sent them the file, they can take the hash of that and they can compare it with the hash they've just decrypted. And if those two match, it means that I must have signed with my private key, because my public key decrypts and gets the same hash. And so, you know that the file came from me. Okay, so that's what we're going to do now. So, let's go back to our terminal. Okay, so now we're going to use 'openssl'. And notice we're taking the 'dgst' and that's going to be a '-sha1' digest. And we're going to '-sign ' with 'myprivate.pem' key And we're going to output to 'sha1.sign'. And we're going to sign the file 'hello.text'. Okay, so let's do that. If I 'cat sha1.sign' that's our signature.

So, now we send our signature and the hello.txt file to, for example, a friend. They can access my public signature. So, they've got all three of those. Now, what do they do? Well, they want to verify that it came from me. So, they now want to verify using my public key and the signature that it was applied to this document, 'hello.text'. So, they have those three pieces of information. Now, they're going to do exactly what I said and see if it came from me and they verify that it does. So, just to remind you, this is what they did. So, we signed without private key, which basically sign the hash of the document, encrypted the hash, and that's our signature. So, we sent the document separately. And we sent this hash and our signature basically.

And they used our public key to decrypt it. They took the hash of the document and compared it with the hash that they got by decryption. And if those hashes matches, they know the file came from me. So, now let's see how this was a file sent in the open Hello World. But let's suppose now we want to encrypt that. So, what we're going to do now is encrypt it. And so, our file now will be this file that's input it will be an encrypted file, but we're just going to sign it in the same way. Okay, so now that allows us to send hidden text to our friend. Okay, so we'll go back and we'll encrypt document. Okay, so let's take a look now at encrypting the hello.text. And we need to encrypt it with the public key of my friend. So, now we need to encrypt with the public key of my friend. So, I've generated my friend's keys, and here's their public key.

Okay, and now what I need to do is I'm going to use their public key here to encrypt the 'hello.text'. And I want the output put into 'encrypthello.txt'. So, let's do that. Okay, and let's look at 'encrypthello.txt'. Okay, so to decrypt that encrypted message, we need to use the private key. So, this my friend's private key. So, this is my friend doing it. And they've got the encrypted text and now they can decrypt. So, let's do that. They're going to put it in 'myDecryptedMessage.txt' And we've retrieved 'Hello World'. So, that's how we encrypt a message. Now what we need to do, we actually would sign, we'd go through that signature process on this encrypted message.

Instead of feeding the plain text Hello World in, we would now sign this encrypted hello. That's how you encrypt and decrypt documents and how you sign them.

Video 5 – Hash Demo Blockies

Let's take a look at hashes and exactly how they function. So, here's a simple example of a hash, but we take the text in a document. So, here we've got hello, and we assign values, say, to those characters. So here, h is 8, e is 5, l is 12, etcetera. And in this case, the hash function is just we sum these values so we can hash 52. Now, the hash is like a "fingerprint" in the sense that if anything in the document changes, let's suppose the e here change to an o. So, now instead of e being five, it'll be an o be 15. So, there'd be now a total of 62. So, the hash would change quite radically. So, let's take a look now at another different kind of hash function.

This one is that we just take the modulus of the number. So, for example, here 25, we divide by 15, goes once and the remainder is 10. So, the hash is modulus, it's the remainder that we get when we divide by 15. So, in this case, we could store these numbers, for example, in an array that's 15 long, 0 to 14. Because we will always get a remainder of either 0, 1, 2, 3, after 14. So, no matter what the number k, it'll be put into one of these bins. In some sense, we can view the hash as a key, that will tell us exactly where the value is stored. So, if I know the key is, for example, 5, then, in this case, I know that 35 is stored there. This is another example of using hashes to bin things.

And that's the way it's used in Bitcoin and blockchain. So, here's another property of a hash that, so this is real hash function SHA-1. And if the input changes just by one bit, you see that we get a radically different output. So, flipping one bit of input over half the bits of the output changed. We can use Merkle trees to lock down documents. In this case, we're taking a series of documents. We're calculating the hashes. And then we combine pairs of hashes and hash those to get a single hash. And eventually, we get a Merkle, what's called a Merkle root at the top hash here, the root of the Merkle tree. Now, if any document changes, this top hash changes. And that's the way it's used in blockchain.

And we'll come back and examine that in more detail later. Okay, so now let's take a look at a real hash function. So, here we've chosen SHA-1, we've got a choice of various hashes. Now, let's put in, for example, "h, e, l, l, o." And you see that we've got a radically different hash every time. "this is a typical". Ah! notice that when I, this hash isn't, it starts with the leading 0. In blockchain, when we need to solve a puzzle, we need to alter a document so that we get a number of leading zeros. So, here we got one leading 0, and that occurs quite often. But to have, say, six leading zeros. Well, we have to experiment quite a lot. You see here "document the quick", there was another zero. But we haven't got two zeros.

So finally, let's take a look at something called blockkeys. And these are just ways of representing the Hash. Hashes are very difficult for humans to look at, and to recognize, say, if two are the same or not. And so, the idea was to translate that into a set of pixels, and they're symmetric about the y-axis here, the middle of this y-axis. And that makes it easier for humans to recognize apparently because we're looking at faces all the time. So, in this case, when we type something in like "hello", we get a different hash all the time. And allows us to recognize if two hashes is the same, very quickly. Humans are very good at visual recognition. Okay, so we'll give you this that you can play with.

Video 6 – Create Your Own Blockie

So, now that you've seen how to create a blockie, why don't you create your own personal blockie? So, type in your name and save your blockie and why don't you put it into your GitHub portfolio and show other people have blockies work. So, here's mine, John Williams. So, let's have a look at my, it's the last one. Okay. I'm going to save that Let's have a look here So, I'll cut and paste it and that'll be my blockie. Okay. So, why don't you show me your blockie on your GitHub site?

Video 7 – Introduction to Recursion

In this section, we're going to deal with recursion. Recursion is an important technique that we use and is very useful and we want to show you the power of recursion. So, recursion is when, in a function, we have statements that call the function again. So, it calls itself. Now, we'll use this on three different topics. One, we're going to permute a string. What we mean by that is we have a string, say abc, and we want to know what are all the different permutations. So, abc, acb, bca, bac, etc. Now, if the string gets longer, there's going to be a lot more permutations. And we'd like to show you the power of recursion in listing out all of these permutations. So, that's one exercise.

Another one is we want you to rotate an image. So, for example, we may have an image and it may be 500 pixels long by 500 deep. Now, we need to take those top 500 pixels and instead of having them in a row, put them into a column. So, we'd have rotated that row of the matrix, but we need to do it for every pixel. And the question is, can we do it efficiently? And what we mean by efficiently is, if it's a large image, we don't want to make a copy of that image. We should be able to do this operation pixel by pixel. So, we'll take a look to see if you can figure out how to manipulate this two-dimensional array of pixels. And this is great exercise for two-dimensional arrays.

And finally, we're going to look at a fun problem that was posed to the monks, supposedly in Hanoi. There were three towers and in one of the towers, there was a stack of disks. And they were stacked so the largest was at the bottom and then the smallest at the top. And their task was to move those 64 disks from tower A to tower C. And they could use tower B to temporarily store things. But the rules were that they can only move one disk at a time and that we could never put a larger disk on top of a smaller one. We could only put smaller disks on top of larger ones.

So, the question was, how do you do this? How many moves does it take? It turns out with 64 disks, it will take you longer than the lifetime of the universe. But we want you to write an algorithm to move these disks. We're going to do it for four or five disks. And we'll see that it gets quite complicated. But with the use of the computer, we can write a very compact algorithm. It turns out this is a great exercise as well for manipulating the DOM. So, we're going to ask you to make changes to what we give you and to manipulate the DOM so we can draw the disks in the right places.

Video 8 – Permute String Exercise

So, here's the kind of question that you might get in an interview. Often they'll ask you to do some coding on a whiteboard, for example. And so, they might ask you, how would you go about printing out all the permutations of these letters in a string? And you should be able to do it for any string. So, this could be quite long. So, how do you go about that? How do you print out all the permutations? Well, the first thing to do is to make it a small problem. So, for example, we might just have three of them. So, let's permute all these letters a b c, say. And the question is, well first, how would we do it? Well, we'd maybe pick the first letter and then write out the permutations of these two. So, b and c, to permute those, there's just its c and b.

So, we might do the same thing that we did here. We pick the first letter b, say here, and then permute everything else. But there's only one letter. So, that would be easy. And then we take the second one and this would be the remaining. And so, we'd picked c and then b. So, what we probably need to do is to be able to pick out of this string one of the letters, and then deal with the remaining letters. And you can see that the remaining letters, if we're trying to print out the permutation of all of these, maybe we can break it down to a task of picking, we'll have a loop for picking, and then will permute the rest of the letters. So, one last thing we started with will be, instead of permuting all three now, we'll just be permuting two.

But we'll have to do it several times. So, what I'm suggesting here is we might need a loop that picks out the ith letter from the string, and then we've got to get the other letters. Yes, so if I pick out b, then I need to get a and c. How would I go about that? Have you got any ideas? So, dealing with strings is like an array. And remember we did slice and splice? So, maybe we can use those. So, stop now and have a go at this problem, okay? Because I'm going to give you some more clues now. But I'd rather that you struggle a little bit with this as the clues that I've given you already to see if you can print out, say this small array a b c. Okay, so try it now. Okay, now I'm going to give you another clue.

So, maybe here we pick out from the string the ith letter. Now, we calculate the remainder. And so, we take a slice up to the ith letter. Then we miss the ith letter out and go from the i +1, out to the very end. And we combine those two strings. So, for example, if I was picking b, then the first part of the slice from 0 to, b is 1, from 0 to 1 would be a. And then we'd be going from two to the whole length, which is only, it would just be the two, so we'd pick out c. So, if we picked b, then the remainder would be a and c. Now, we can just operate on the remainder. We can just find how those permute. So, we're going to call ourselves recursively. And this is going to return permutations.

So, for each of those permutations, we want to combine what we've picked with whatever permutation it is and push it onto an array called, we'll call it permutations, okay? So, somewhere we've got to define this array. And the permutations we can eventually print them out. They'll be a long array of all the permutations. So, I've written 'permute.js', and I'd just run it. And you see here that it does it for that many letters. Let me bring it up here. Letters ' "abcde"; '. Okay, and I can run it again. So, I just changed the letters to abcde, and we can run it again and you'll see that it gives us a much longer array, okay? So, this is the kind of problem that they'll ask you, maybe in an interview. And practice, those people going to interviews before, practice programming, okay? So, we'll give you a couple more of these to practice.

Video 9 – Permute Exercise

So, we saw how to permute a string of letters. And what I want you to do now is to provide a user interface for that. And I'll give you starter code. So, the way it should work is that, here's our letters a, b, c, d. And then we step through them, a, b, d, c, etcetera. And let me give you the starter code here. So, here's the starter code. We're setting up a grid with a certain number of elements. And then 'onclick', I want you to call 'Step'. And here, we're fixing this array, and you need to change this to enter code there. 'setTags' is called.

And we're calculating all the permutations in one go. So, this gets executed. And so, permutations now is an array that contains all the permutations. So, here it is. And we're just stepping through it one by one. Each time you click step, we get a different combination. Now, if you could generalize this as well to any number of letters, say, at least a few more. That would be great, and you might even do it on the user interface. Okay, so that's our permutation exercise.

Video 10 – Rotate Image

So, here's another exercise that you might want to go through to prepare yourself for an interview. So, suppose they asked you; how would you rotate an image like this 90 degrees clockwise? So, here we've got an image of MIT and it's about, it's about 500 pixels by 500 pixels. So, you can imagine 500 times 500. That's an awful lot of pixels. That's about 25,000 pixels. How are we going to do this? Well, think about it. We need somehow to take this row of pixels and make it into that column of pixels. So, if they ask you a problem like this, try to come up with a very simplified version. So, let's take a look at this next slide.

So, here we've got the same problem, but we've just got 3 pixels across the top, a total of 9 pixels. And the value of this pixel is 1, this pixel is 2, etc. But essentially, to rotate it, this is what we need to do. We need to make the 1 2 3 down that column. We need to make the 4 5 6 that column. And the 7 8 9 needs to be this column. So, how do we do that? That's a much simpler problem. But it's still not obvious. And it's especially not obvious to do it without, say, storing the whole matrix.

Because as we saw, here this is quite a big matrix. This is 25,000 entries, 500 by 500. We mustn't copy the whole matrix, okay? We should be able to do this just by moving one at a time, somehow.

Even this is a tricky problem. Try to do this. Just rotate a small matrix like this. So, let's break it down. Let's suppose I can take a transpose of a matrix. Now, this is actually, mathematically, we do this all the time. So, let's take a look. This first row becomes the first column, second row becomes the second column. Third row becomes the third column. Now, can you see a correspondence between the transpose and the rotated? So, the difference is just that the 7 8 9 column here appears here. And the 1 2 3 column appears here. 4 5 6 appears in the same place. So, if we can get the transpose, all we have to do then is swap the columns.

And we need to work from the outside columns to the next column in, to the next column in, etc. Until we hit either the middle column which doesn't get swapped or if the number of entries is an even number. Here, we're showing a three by three. But suppose it was a four by four, then we'd have the middle two would get swap it. Okay? So, that's a much simpler problem. So, let's focus on this one. How to get the transpose? Well, let's look at how a matrix is represented. In the computer, we represent a matrix like this, 1 2 3 4 5 6 7 8 9, like this, that we have 1 2 3, that top row in this A[0][0] is 1, A[0][1], is 2, A[0][2] is 3.

Here, this is the A[1] row. So, A[1][0] is 4, etc. A[1][1] is 5. So, this is how this matrix is represented and how we access things. So, let's suppose we want to do the transpose. Now, what you'll notice with the transpose is we don't need to move every entry. We only need to swap the 2 and the 4, that diagonal, that diagonal, and this diagonal. Notice the 9s stay in place. The 5s don't move at all and the 1 doesn't move, but the others do. Okay? So, what we need to do is to be able to swap A[i][j] with A[j][i]. So, we need to write a swap routine. That's the first thing we need to think about. And we might want to write it more generally that we might want to swap any entry with any other one.

But you can see how to do that I think. What we need to do is, let's suppose I was swapping 2 and 4. So, this is A[0][1], that's 2. And then this is A[1][0]. So, we need to take this element and store it in a temporary, move the 4 to where the 2 is, and now take that temporary one and write it back into there. Let's take a look at what swap looks like. So, here's a 'function swap', and it'll swap i, j, and k, l. So, we take now i, j. We usually think of x, i being x, and y being j. So, that's why I swapped these here, that a this j is the row and i is the x-direction, right? So, we put that into temporary 'A[j][i];'. Now, we overwrite 'A[j][i];' with 'A[l][k];' And now we put into 'A[l][k];' the temporary. So, this will work for any values i, j, and k, l.

So, that swap, we know how to swap 2. So, now we need to figure out loops. Here's transpose. So, we need to find out the length of the matrix A. Yes, we're going to have to move the 0 of i remember? But we only go to if the length is 3. We just go to i less than 2. So, I1, we don't need to do the corner ones. Remember they stay in the same place. So, you might want to check this out. Let me just, I've written it all. So, you need to write transpose. Then you need to exchange columns and then you can do rotate matrix as transpose and exchange columns. So, here's a matrix, it'll take any matrix A and write the transpose. And then by exchanging the columns, we get the rotated one.

So, I'll put this up for you so you can copy it. Get used to breaking the problem down. There are other ways actually of doing this. If you're into mathematics. We know how to rotate coordinate systems and we could do it that way. But without mathematics, you can see it how to do it here that just by taking this transpose of it, so swapping the rows to be columns and then swapping the columns. So, here's exchanging columns. You'll see there's in here, two loops, two for loops and then we use swap. So, we're not using up the whole matrix space with just two entries. Yes, swap. and here's transpose. Again, it's another two loops. So, you need to figure out those loops and at the top here, this is how we input matrix.

Video 11 – Rotate Matrix Exercise

Okay, so we've seen how to rotate a matrix and transpose a matrix, and swap columns. So now, I want you to produce a user interface for it that looks something like this. So, here's our starting matrix, 1, 2, 3, 4, 5, 6, 7, 8, 9. Now, if we transpose that matrix, I click transpose. Now, update the graphics. And you see now 1, 2, 3, in the columns, 4, 5, 6, 7, 8, 9. So, that's the transpose of the matrix and if I click transpose again, we get back to the starting point, okay? The transpose of the transpose is the original matrix. Now, we want to rotate it. So, I transpose. I'll update the graphics, and now I'll swap the columns. There we go. And now, when I click swap columns, we've got 7, 8, 9, 1, 2, 3 there. And so, that's our rotated matrix. So, I'll give you some starter code for this. You'll need to make some minor changes perhaps to your solution just in the way that functions are called. But that's what your task is. Okay, bye for now.

Video 12 – Tower of Hanoi Exercise

So, let's see if we can solve what's called the Tower of Hanoi problem. This is a famous problem where monks were asked to move all the disks from one tower to another tower, using a third tower as a staging point. But they could only ever move the top disk of any stack. And they can only move one disk at a time. And any disk that was moved must be moved to a tower, to sit on top of a larger disk. So, you can never have a larger disk sitting on top of a smaller one. So, the goal is to get from this configuration to this configuration. But to do it moving disks one at a time. So, let's see if we can program this. So, let's have some notation for suppose I have got a function that can move n disks from A to C using B because the intermediate, peg, the spare peg.

So, that's what this means. Now, let's suppose if we know how to do it for n disks, then we certainly know how to do it for n-1 disks. So, we can do move n-1 from A to B. Move with C, for example, as the intermediate. So, you can take n-1 of these and move them to peg B. Now, what use is that to us? Well, if we can do that with n-1, then we're going to get to this state. We'll move n-1 to B. Now, can you see how we can solve it? Now, we can move this one to C. And if we know how to move n-1, we can move the rest here onto C. So, let's see that. And so, now I move disk, the larger disk n from A to C, I've moved n-1 of them from A to B. Now, what do I need to do to finish this? Well, I need to move n-1 from B to C.

Okay, so this is going to be our algorithm. Let's just take a look at what happens for, say, three disks. So, let's suppose we're applying our algorithm to three disks. So, we want to move three of them from A to C. And we know we can break that down into moving two of them from A to B. Then we move disk 3 from A to C, and move two of the disks back then from B to C. Now, notice we're calling move with the arguments in different permutations. So, we have to be a little bit careful in our logic as we think through it here. But actually, when we come to programming, it's going to automatically take care of it, which we'll see. So, now let's have a look at these move two's. So, here we have a move 2, and let's break it down.

We want to move two of the disks from A to B. Then we move the big disk, 3rd disk from A to C directly. And then we move the two from here back. So, we want to go from A to B with two of them. So, that means we move one of them from A to C. So, the top one, we're taking two of them from here now. And we want to move 1 from A to C. Then we move disk 2 from A to B. There it is at the bottom. And now, we'll move 1 from C to B. And now, we move disk 3 over, and then repeat this with moving 1 from B to A. So, we move the top one back to A, move disk 2 B to C, and then move from A to C, okay?

So, that's going to be what we need to do. Let's take a little look what happens when we hit 1 here. We're just doing this one, moving 2 from A to B using C. Now, we call move with 1 A to C. Now, we don't want to call move 0. So, we put an if statement in, if(n = 1) just move that disk A to C. Now, we move disk 2 from A to B. There's 2 from A to B. And then again, we want to call move this disk back from C to B. But we want to do this, check, if. So, we need an if statement in there to stop it when it gets to 1. So, that's the trick. So, let's go and take a look at our code. So, here's the code. Here's our function 'moveDisks', and we move 'n' disks 'from', whatever the from is. So,

we're going to set that up as down here. Here, we're actually firing 'moveDisks'. So, the 'from' is ' "A" ', and the 'to' is ' "C" ', and the 'spare' is ' "B" '. We're going to call so from A to C, with B as the spare. So, you put the 'if' statement, and if 'n' happens to be '==1', then we directly move the disk. Otherwise, we call 'moveDisks(n - 1,' and we move 'from, ', whatever from is to the 'spare', and using the other one which is 'to' as a spare, okay? So, this is how the letters get changed around. And now, we know we're moving from the 'spare', 'to' the final destination, and using 'from'. So, and in-between, we just move that last biggest disk across whatever it is.

So, that's the algorithm. Notice that we're calling ourselves, this is a function and this is called recursion. So, we'll see if we step through this, that we're calling the same function again. And so, we have to hold a pointer to where we got to. If we got to this point here, then when we return from this function, we need to step onto here. However, if we call the function from this point, we need to make sure we step on it. And to, that'll be to the end. Okay, so that's the code. And the unbelievable thing is that that will work for any number of disks. And now, the monks found out that if n is large like 64, and that was the task for the monks, they needed to move 64 disks. They found that it would be longer than the life of the whole universe to date.

That if they could move one disk a second, even that it would just take forever. And so, this is the trick, if you like, of the Tower of Hanoi, but we can program it just like this. Okay, let's see it run. So, I've set it up to move two disks. Let's see if it runs it. To run it, I'm using something called 'node', which you've loaded on your machine, but we haven't dealt with yet. So, don't worry about that. And hanoi.js, that was the code I wrote. So, for two disks, we just move 1, the top disk from A to B, the bottom disk from A to C, and then the little disk from B to C. And that will solve it. Let's run it now with three disks. Okay, so now, I've set it up with three disks. Let's run it again and see what the result is.

So, you see this time we have to do a lot more moves. So, take a look, check this out. It does give the right solution. And let's try something with four disks. So, here we are with four disks now. I'm going to clear that and run with four disks. So, how many steps? Counting, there're actually, if I count these, there're 15 moves. And before we found there were seven moves actually. And before there were three. So, three, seven, 15. See if you can spot the pattern and predict how many moves five disks will take. There is a pattern there. Okay, so that's recursion and we'll have an exercise on this.

Video 13 – Tower of Hanoi Solution

This is a famous problem where monks were asked to move all the disks from one tower to another tower, using a third tower as a staging point. But they could only ever move the top disk of any stack. And they can only move one disk at a time. And any disk that was moved must be moved to a tower, to sit on top of a larger disk. So, you can never have a larger disk sitting on top of a smaller one. So, the goal is to get from this configuration to this configuration but to do it by moving disks one at a time. So, let's see if we can program this. So, let's have some notation for suppose I have got a function that can move n disks from A to C using B because the intermediate, peg, the spare peg. So, that's what this means.

Now, let's suppose if we know how to do it for n disks, then we certainly know how to do it for n-1 disks. So, we can do move n-1 from A to B. Move with C, for example, as the intermediate. So, you could take n-1 of these and move them to peg B. Now, what use is that to us? Well, if we can do that with n-1, then we're going to get to this state. We'll move n-1 to be. Now, can you see how we can solve it? Now, we can move this one to C. And if we know how to move n-1, we can move the rest here on to C. So, let's see that. And so, now I move disk, the largest disk, n from A to C, I've moved n-1 of them from A to B. Now, what do I need to do to finish this?

Well, I need to move n-1 from B to C. Okay, so this is going to be our algorithm. Let's just take a look at what happens for, say, three disks. So, let's suppose we're applying our algorithm to three disks. So, we want to move three of them from A to C. And we know we can break that down into moving two of them from A to B. Then we move disk 3 from A to C and move two of the disks back then from B to C. Now, notice we're calling move with The arguments in different permutations. So, we have to be a little bit careful in our logic as we think through it here. But actually, when we come to programming, it's going to automatically take care of it, which we'll see.

So, now let's have a look at these move 2's. So, here we have a move 2, and let's break it down. We want to move two of the disks from A to B. Then we move the big disk 3rd disk from A to C directly. And then we move the two from here back. So, we want to go from A to B with two of them. So, that means we move one of them from A to C. So the top one, we're taking two of them from here now. And we want to move one from A to C. Then we move disk 2 from A to B, there it is at the bottom. And now, move 1 from C to B. And now, we move disk 3 over, and then repeat this with moving 1 from B to A. So, we move the top one back to A, move disk 2 B to C, and then move from A to C, okay?

So, that's going to be what we need to do. Let's take a little look what happens when we hit 1 here. We're just doing this one, moving 2 from A to B using C. Now, we call move with 1 A to C. Now, we don't want to call move 0. So, we put an if statement, if(n=1), just move that disk A to C. Now, we move disk 2 from A to B. There's 2 from A to B. And then again, we want to call move this disk back from C to B. But we want to do this, check, if. So, we need an if statement in there to stop it when it gets to 1. So, that's the trick. So, let's go and take a look at our code. So, here's the code. Here's our function 'moveDisks', and we move 'n' disks 'from', whatever the from is. So, we're going to set that up as down here.

Here, we're actually firing loop disks. So, the 'from' is ' "A" ', and the 'to' is ' "C" ', and the 'spare' is ' "B" '. We're going to call. So, from A to C with B as the spare. So, we put the 'if' statement in, 'if' 'n' happens to be '==1', then we directly move the disk. Otherwise, we call 'moveDisks(n-1,' and we move 'from, ', whatever from is to the 'spare', and using the other one which is 'to' as a spare, okay? So, this is how the letters get changed around. And now, we know we're moving from the 'spare' 'to' our final destination, and using 'from'. So, and in-between, we just move that last biggest disk across whatever it is.

So, that's the algorithm. Notice that we're calling ourselves, this is a function and this is called recursion. So, we'll see if we step through this, that we're calling the same function again. And so, we have to hold a pointer to where we got to. If we got to this point here, then when we return from this function, we need to step onto here. However, if we call the function from this point, we need to make sure we step on. And to, that'll be to the end. Okay, so that's the code. And the unbelievable thing is that that will work for any number of disks n. Now, the monks found out that if n is large like 64, and that was the task for the monks, they needed to move 64 disks.

They found that it would be longer than the life of the whole universe to date. That if they could move one disk a second, even that it would just take forever. And so, this is the trick, if you like, of the Tower of Hanoi, but we can program it just like this. Okay, let's see it run. So, I've set it up to move two disks. Let's see if it runs it. To run it, I'm using something called 'node', which you've loaded on your machine, but we haven't dealt with yet. So, don't worry about that. And 'hanoi.js', that was the code I wrote. So, for two disks, we just move one, the top disk from A to B, the bottom disk from A to C. And then the little disk from B to C, and that will solve it. Let's run it now with three disks.

Okay, so now I've set it up with three disks. Let's run it again and see what the result is. So, you see this time we have to do a lot more loose. So, take a look, check this out. It does give the right solution. And let's try something with four disks. So, here we are with four disks now. I'm going to clear that and run with four disks. So, how many steps? Counting, there are actually, if I count these, there're 15 moves. And before we found there were seven moves actually. And before there were three. So, three, seven, 15. See if you can spot the pattern and predict how many moves five disks will take, there is a pattern there. Okay, so that's recursion and we'll have an exercise on this.

So, here I've integrated some HTML code with the main algorithm for Towers of Hanoi. And I've set it up here for five disks. And we can make moves. So, for example, let's move 1. And as I click through, it makes a move every time. And you will see that we end up with n-1 of them on this middle peg, peg B. And now, you can see that we can transfer the last disk to peg C, and then we just have to move the n-1 onto C. Of course, that takes quite a number of steps but, okay, there you are, that's the solution. So, what I want you to do is to take this code and I want you to set up an input here that will specify how many disks that we're going to be using. So, take the code and put an input here so that it specifies how many disks you want made. Okay, so that's your exercise. Okay, good luck. Bye for now.

Reflecting on Your Skills in Web Development

The president of MIT, Rafael Reif said, "Today everyone needs to be bilingual." And what he meant was everyone needs to have a programming language. In this course, we've given you not one programming language, but three, JavaScript, HTML, and CSS. You can have a career in any one of those. But combining the three together makes your web programmer. It's the basis for developing websites and web applications. It's the first building block of becoming a full-stack programmer. You've done the most difficult part now. You've learned JavaScript.

You've learned how to apply it and build web pages and websites. You've got a portfolio that you can be proud of. You can show your portfolio to any employer and they'll be impressed with what you've done. So, you've taken the most difficult step. It's up to you now. You can get a great career in this area. Abel and I program almost every other day, we're passionate about it. We hope you've given, we hope we've given you some of that passion. It's up to you now. We hope you take our other courses and become a full-stack programmer. But you know enough to teach yourself now. So, good luck with your career, and thanks for taking our course.