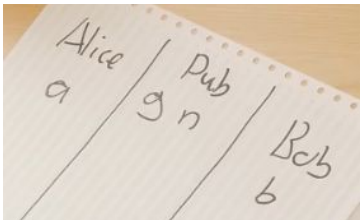
Diffie-Hellman – the mathematics bit [1]

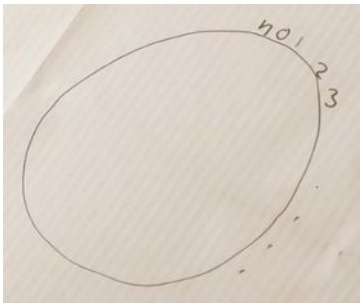
1. Every time I talk about Diffie-Hellman and use any kind of analogy, people were like: “Oh! Show us the maths! Show us the maths! I could have taken the maths!” So, this is for the maths people, right? if you want to know ***how mathematically Diffie-Hellman works***, watch this video. If you don't want to know that, there's a nice *close button* in the top corner, or a *back button* – you know, be my guest! So, that… this is for mathematically inclined people.

2. Let's go back to where we were before: so we have *Alice*, and we've got *Bob*, and we've got our public area here… so, public… public… and we’re gonna draw down here. Now, the mathematics behind Diffie-Hellman is usually ***modulo arithmetic***. Recall that we have our public numbers *g* and *n*:
   - ***g*** is often very small, is usually a small prime number;
   - ***n*** is often very big and needs to be big for the security of this to work; *n* is often 2,000 bits long – or 4,000 bits is more common now – erm… so *n* is very, very big (it can't be too big because you won't gain much in security, but you lose in efficiency, so you know, you have to think about that).

3. Alice and Bob need to pick their numbers ***a*** and ***b***. So Alice picks a number *a*, Bob picks a number *b*: these are the ones they're going to keep private. Now, *a* is somewhere between 1 and *n* but it's random, and *n* is so vast, that it's not going to be 1: it's gonna be a very big number, let’s not worry about what it is, she's not gonna tell anyone what that is. Same for Bob.

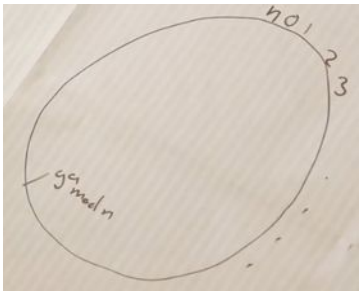


   The first thing Alice does is she calculates g^a mod *n*, right? Now, ***modulo***, if you do any programming, you'll be familiar with – it's a percent symbol – is the remainder after division. So, another way of looking at modulo is to have this kind of clock face, so if we have a clock face, which should be a circle, and we go from 1, 2, 3, all the way round to *n*. These are the numbers modulo *n.* So when we perform some arithmetic in this space, we just go around and around the clock face. We don't ever leave and go above *n*, or below 0. In fact, it… this should be 0 as well (this should be a 0 in here)[2].



4. So when you do g^a **mod** ***n***, what happens is you're raising *g* to the power of some massive number, which would be very normally very big. But in actual fact, it just goes round and round this clock face and ends up somewhere. So let's say g^a mod *n* arrived somewhere here on the clock face.

   Now, what's important about this is it's very difficult, given this, to work out what *a* was, right? We know *g*: it’s, let's say 3, right? If I say we are here on the clock face, and we started at 3, what number is *a*? It's… It is impossible to know, right? because its position on this clock has no bearing on how many times it’s gone round, or what *a* was at all, right?



1 https://www.youtube.com/watch?v=Yjrfm_oRO0w

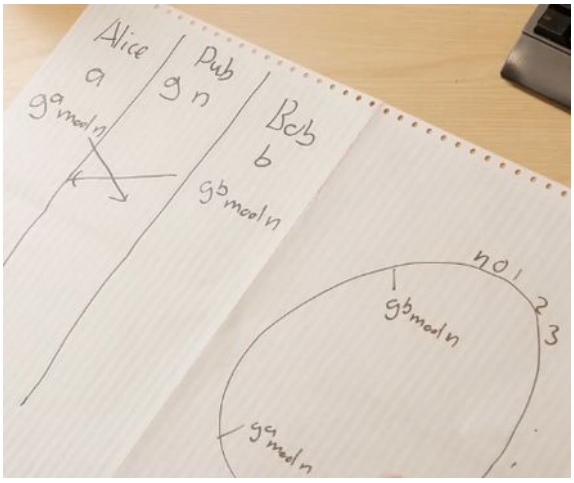
2 Transcriber note: on the clock face, 0 should replace *n* (because *x* mod *n* goes from 0 to *n* − 1).

5. The only way to do this is essentially to ***brute-force*** it to go:
   - Well, is it a^1? No, it's not.
   - Is it a^2? No…
   - … and then so on and so forth, for an infeasible amount of time.

   Brady — Well, we did a little bit of this with the hashing video, didn't we? There was a little bit of the modulo function there in… in calculating the hash.

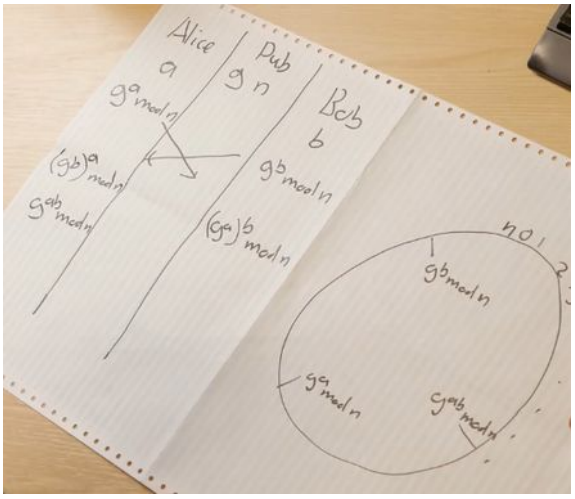
   Mike — Erm, yes, so that was used to shorten something at the end, but it's the same kind of principle: modulo is very useful for taking something that could be any length, and putting it into a sort of finite loop… a finite group of actual numbers.

6. Now, Bob: we're going to take g^b **mod** n, so let's say that turns off over here somewhere. So this is g^b mod n. So again, what we've done is we've taken g, we've raised it to the power of b, and we've done all of this modulo n, which means that it just… if it ever goes above n, it just loops back round to 0 and keeps going, so this is somewhere else.

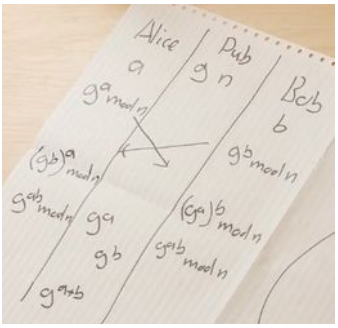
   These are the ***two public components***, so they share these… like this. So now these are public, but again calculating a and b from this is very, very difficult. It's called "solving with discrete log problem" and practically, very, very difficult… for even a supercomputer.

7. All right, now Alice is going to take this – I'm just going to simplify the notation slightly to make it fit on the page – but:
   - Alice going to take the g^b that Bob sent, and raised it again to the power of a mod n,
   - and Bob is going to take g^a that Alice sent and raise it to the power of b mod n.

   And anyone that's done any exponentiation knows that if you do something to the power of something, to the power of something else, it's actually just those two things multiplied, so g… it's g^{ab} mod n, that's the answer. So that will put you, that will come somewhere around – you know, let's say here – so this is g^{ab} mod n. Now, this will be some number between 0 and n. Bob's done the same thing, he's also got g^{ab} mod n and they're exactly the same.

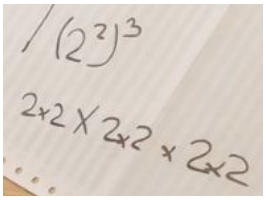
8. These are the two identical colors we were looking at in our colour example. So, they've both arrived at the exact same position in this group, despite the fact that neither they knew what each other's private key was – that's what's really cool about Diffie-Hellman. To try and ***reverse this process***, we have to know a or b. We know publicly g^a and we also know g^b. If we try and multiply them together for example, we'll get g^{a+b}, which is not the same, right? That's mi… mi… a sort of mixing my public colours together in the hope of getting the right answer – I haven't done it. That would be somewhere else – a completely different number.

9. Remember, this is ***cryptography***: if you're one position out, it… it not going… it's not going to decrypt, right? so you know you have to get it exactly right.

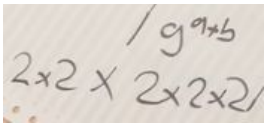
   Brady — The fact that she's got $(g^b)^a$ is no different to $(g^a)^b$.

   Mike — It's exactly the same, I mean you could… you could look at an example if you went $(2^2)^3$, that's $(2 \times 2) \times (2 \times 2) \times (2 \times 2)$, right? which is 2^6 because there's… there's 6 of them. You can do it in any order, it doesn't matter, you get the same number out at the end.



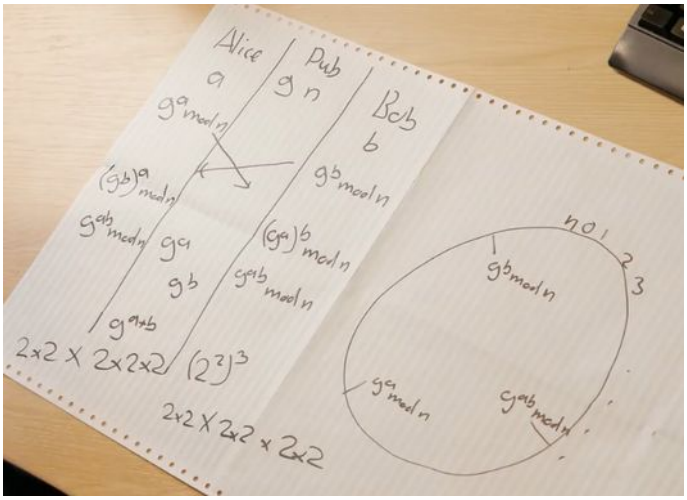
   Brady — Whereas that is a completely different thing.

   Mike — Yeah, so that is equivalent of $2^2 \times 2^3$, which is $(2 \times 2) \times (2 \times 2 \times 2)$, which is 2^5 – entirely different number. Now, those numbers are fairly similar because these examples are small. You're just gonna be somewhere else completely on this… on this modulo arithmetic clock face, you're not gonna… it's not gonna work at all.



10. Brady — The ***n*** number is kind of important though, is it, right?

    Mike — It's mostly important that *n* is big, because if *n* is small then, in essence, this clock face is going to have only a few numbers on it. You can brute-force that very quickly, right? You can find the value of *a* or the value of *b* and reverse this process. If *n* is, you know, astronomically large like 2,000 or 4,000 bits, the amount of time it's going to take you to find the correct… the correct values for *a* or *b* is… I mean intentionally is long enough that you won't bother – that's the argument. It's technically possible, but you would… they'd be long dead by the time you did it yet, and so you… your… you finding out what image they send each other is not very useful!



11. Actually, this is… this is ***quite simple***, right? I mean, let's not underplay: this is incredibly important for computer science and mathematics. But it's actually not that complicated in some sense, very elegant. If you want to see some worked examples of this, *Wikipedia* and other websites have lots of small examples with small numbers, so that you can work this through if you want to have a go at the math yourself, right? And you'll get the same answer out, and it's, you know, it's impressive!