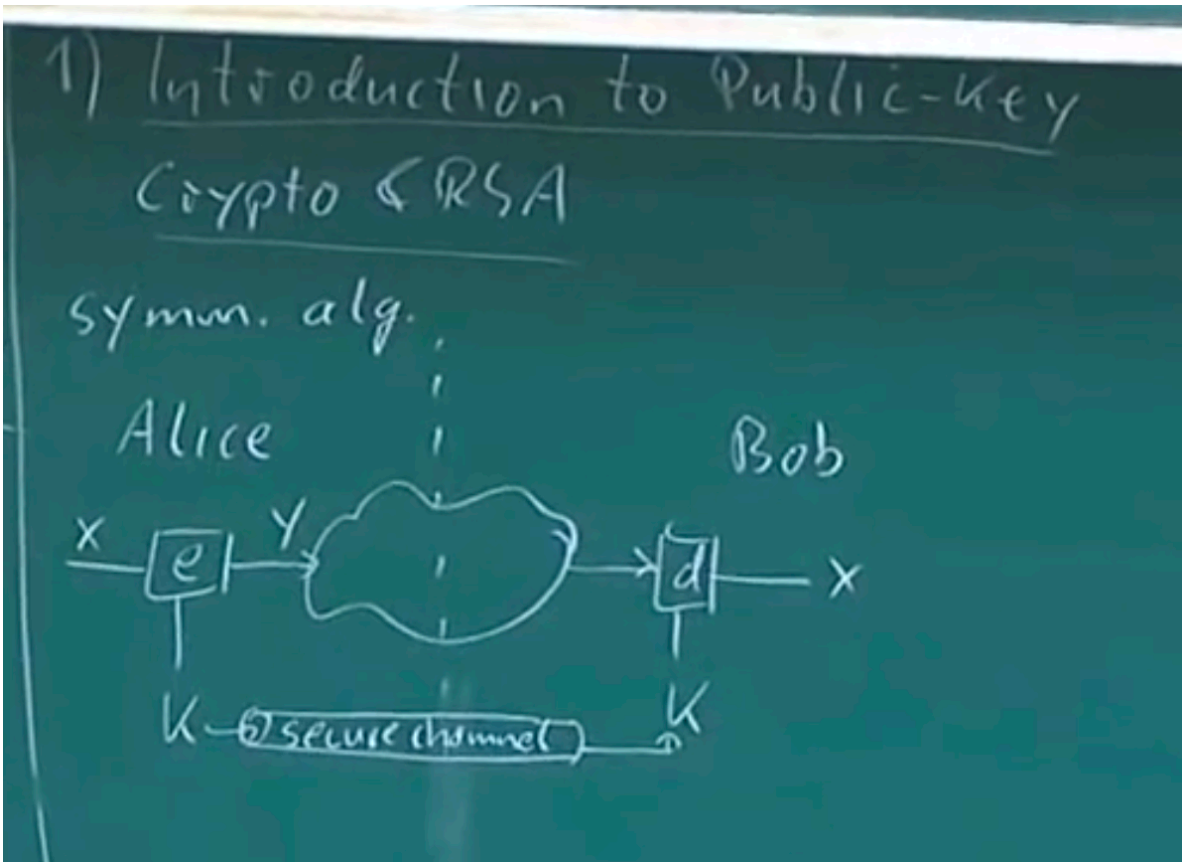
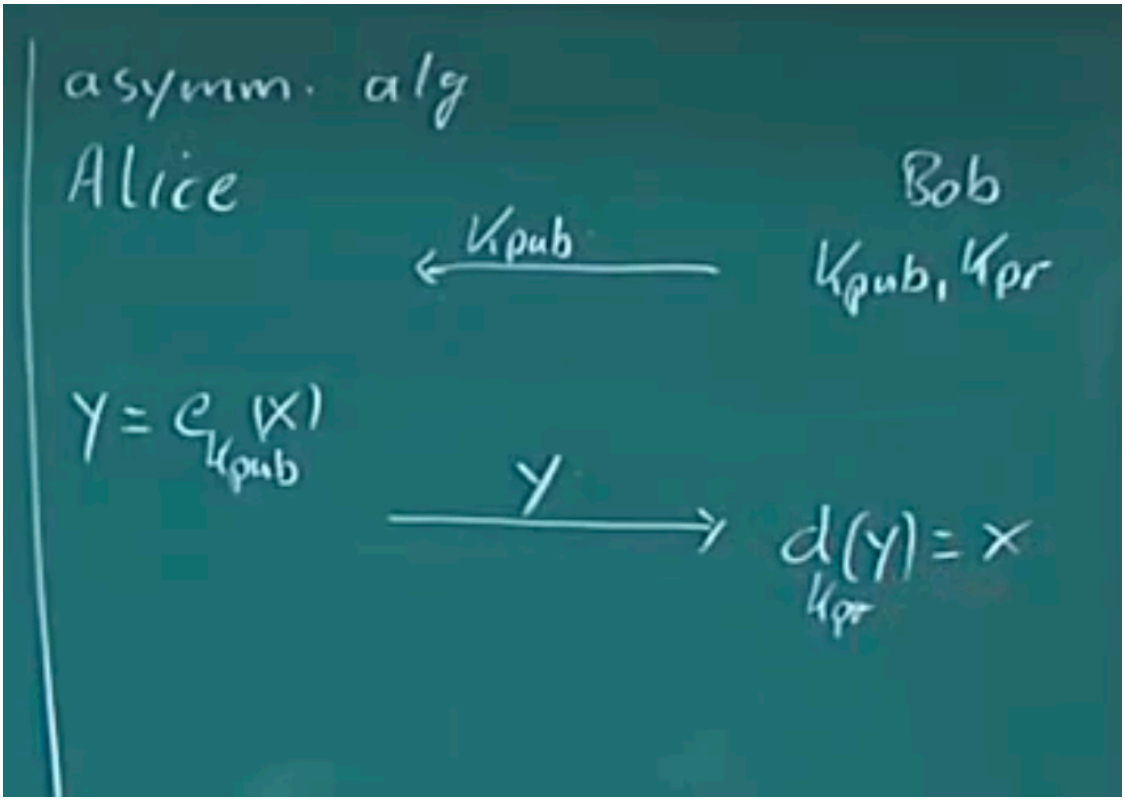
Lecture 12 RSA Cryptosystem and Efficient Exponentiation

More famous, heavily used in web browsers, it used all over the world in particular in internet.

1. **Introduction to Public Key crypto & RSA**



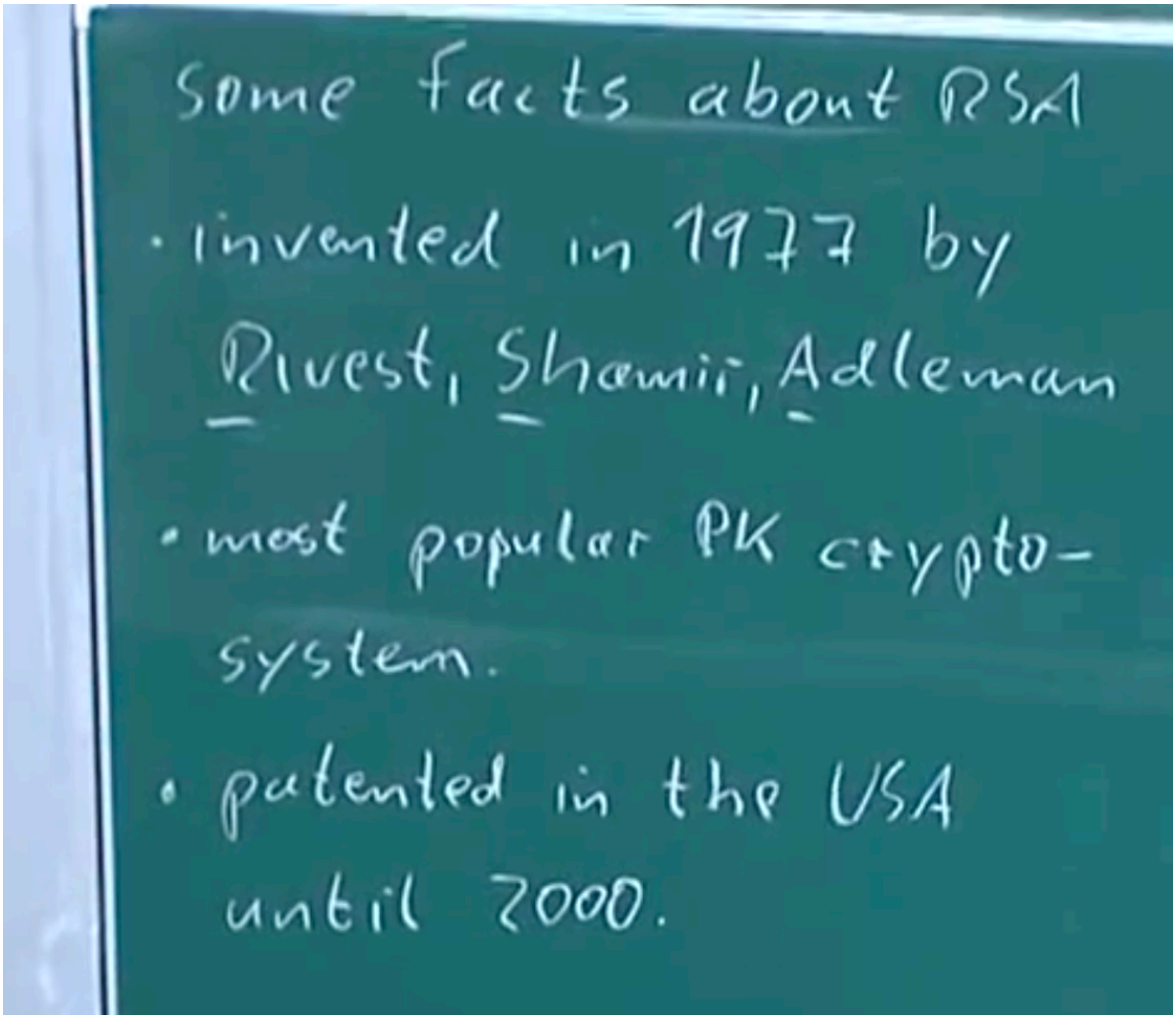
the general view of Symmetric cryptography.
Only issue is passing key over secure channel and as we know it is tough and in case of larger networks it is very difficult.



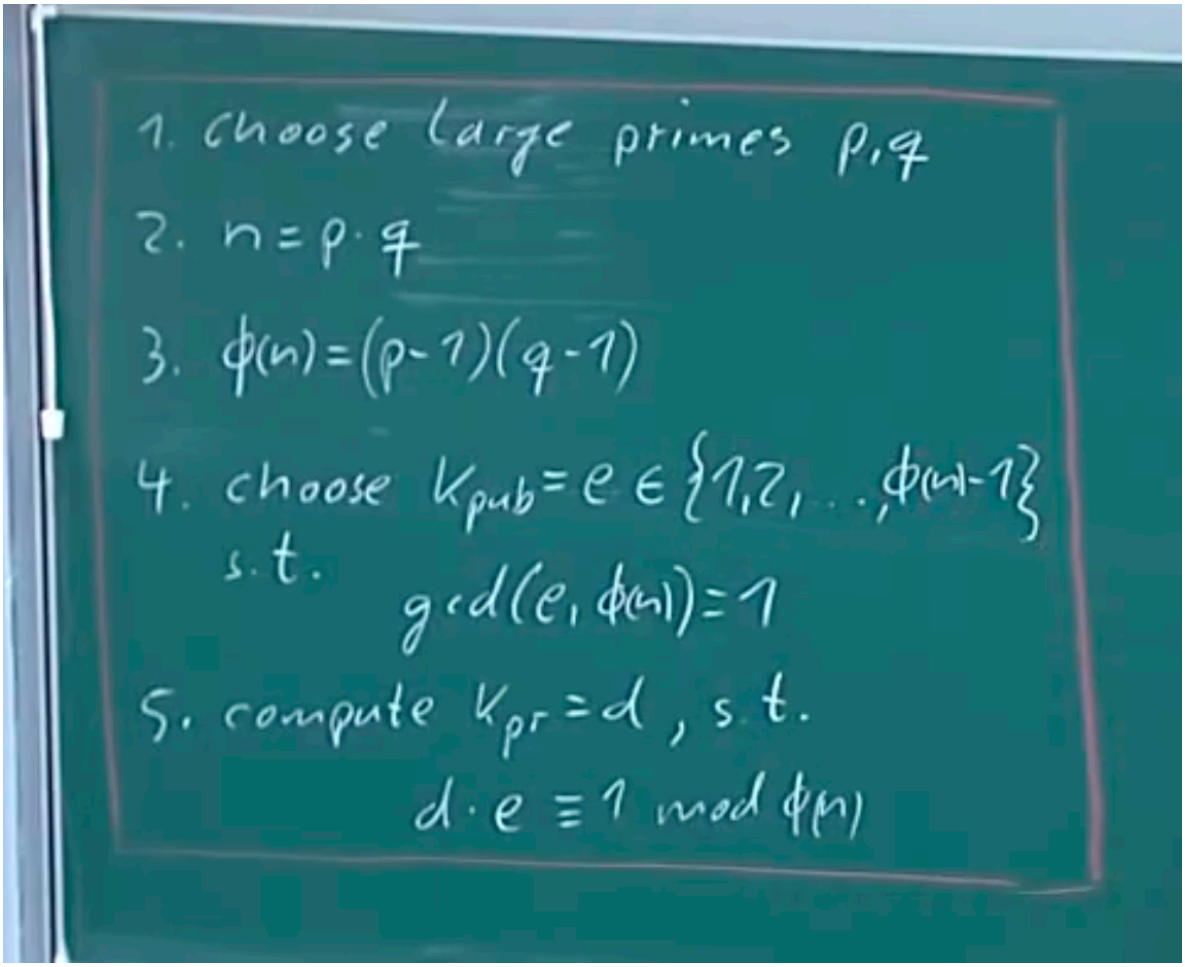
RSA is e and d function in above picture.

Invention of Public key was introduced by Diffie Helman.
there are many types of Symmetric Crypto Systems and therefore it is tough to learn or taught in class. however there are only 3 types of Asymmetric Crypto systems mainly.

1. RSA
2. Discrete algorithm
3. Elliptical Curve cryptograophy



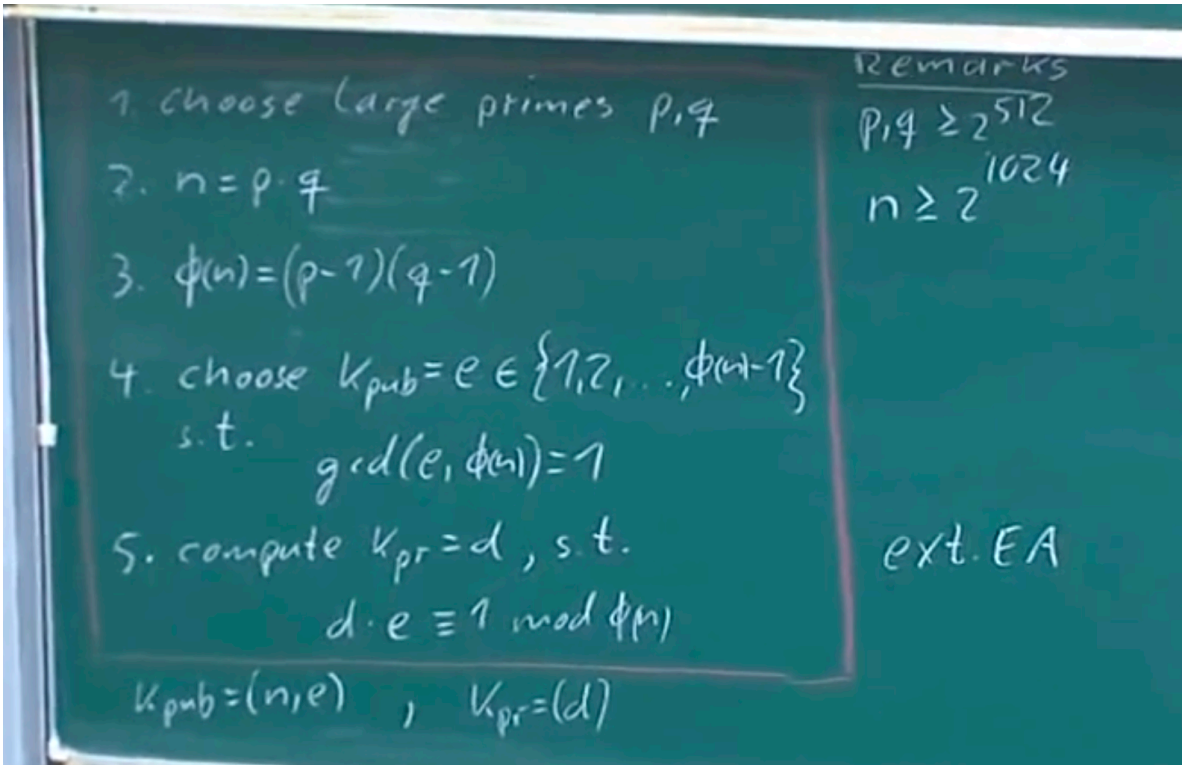
2. **RSA Algorithm**
    1. **Key Generation:** unlike Symmetric Key Cryptography, Public key cryptography requires computation of Kpub and Kpri keys. in AES or DES we don't require to generate the Key and it is generated from TrueRandom Number Generator but in public key cryptography, it requires computations which internally requires hefty maths and in case of Elliptical Curve, it sometimes requires hours of computations to generate the Key. Key Generation in RSA is not bad in key computations. Below are the 5 steps for Key Generation.



3. Choose large p and q.
4. n = p*q, note n ends up being the modulus
5. we need to find phi(n) question what was the condition to find the phi functions, **if we know the prime factorization**
6. phi(n) => (p-1)(q-1)
7. choose Kpub = e = {1,2,..,..,phi(n)-1} such that GCD(Kpub and phi(n)) = 1
8. Note that in above case we are saying Kpub is used for encryption, why as we already know encryption is done in RSA with public key and decryption is done using private key.
9. as we are choosing Kpub such that GCD is one and now question is if GCD is one then what value exists ? **Modulo Inverse**
10. Kpub calling it as e and Kpri calling it as d
11. e*d = 1 mod phi(n)

What is the output of the above steps ?
we got Kpub and Kpri keys, Kpub=(n,e) and kpr=(d)

Remarks:
P and Q should be atleast 2^512 and n should be atleast 2^1024
however in practice now we need twice that length.



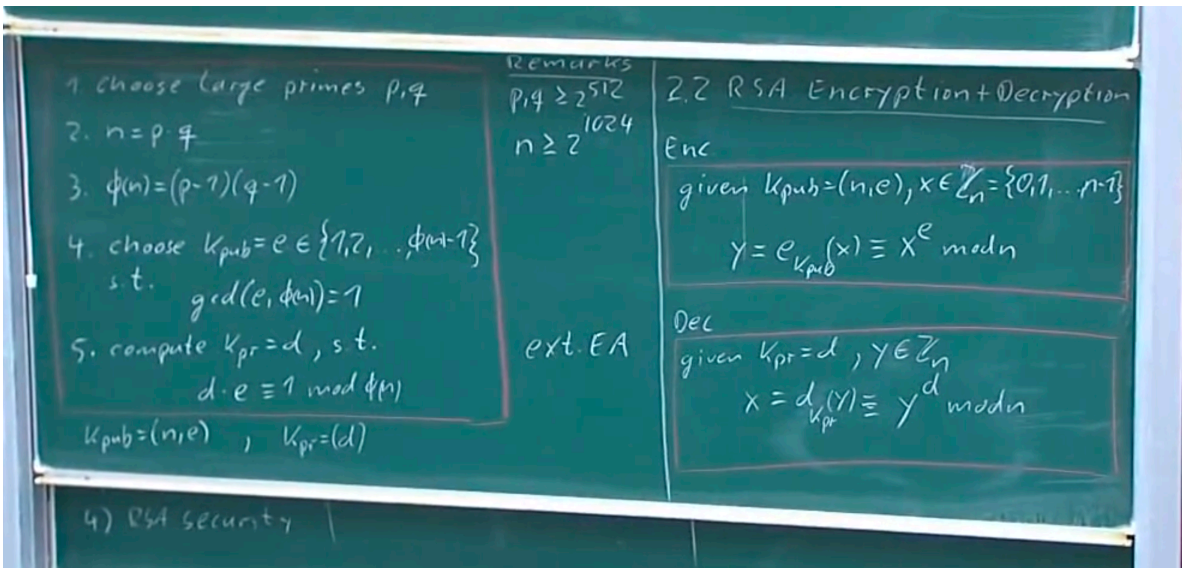
Important point: when encrypting with PGP or any other tool, when we say encrypt with key of 1024 bit length, it means we are taking about "n".

so Key length in RSA is "n" parameter strength.

So we have done the key generation but we have not done encryption and decryption.

RSA is one algorithm where encryption and decryption are very simple conceptually and they are one operation.

Entire RSA:



In Encryption:
Given is Kpub which consists of n and e where e is the key and n is the prime p and q multiplier.
X the block to be encrypted belongs to Zn which is {0,1,2....,n-1}
now what is the encryption method ? simple **X^e mod n** where e is the Key and n is the prime p and q multiplier.

In Decryption:
Given is Kpri = d and Y (encryption message)
Y^d mod n

A little remark on Security of RSA:
Question: What does Oscar require to decrypt Y message ?
Answer: I think Oscar requires "d"
Question: He doesn't have "d" then what he needs to do ?
Answer: he can try to compute the inverse of "e" using extended Euclidean algorithm ?
Question: then what is the problem, he can break RSA right ?
Answer: As he is missing one more information that is Phi(n) to find the "d"
Question: then he can find using phi formula, which is very fast ? what is the problem in that ?
Answer: we know finding Phi is easy only if we know the prime factorization of a number. as without prime factorization it is computationally infeasible to find the
phi.
This is the reason modern cryptography and factoring are closely related.

but the formula is only fast if we know P and Q, it turns out that if we have 1024 bits number which is a 300 digit number, you cannot factor that and factoring record is 650 bits. we cannot do 1024 and definitely we cannot factor 2000 bit.
37-38

Example:

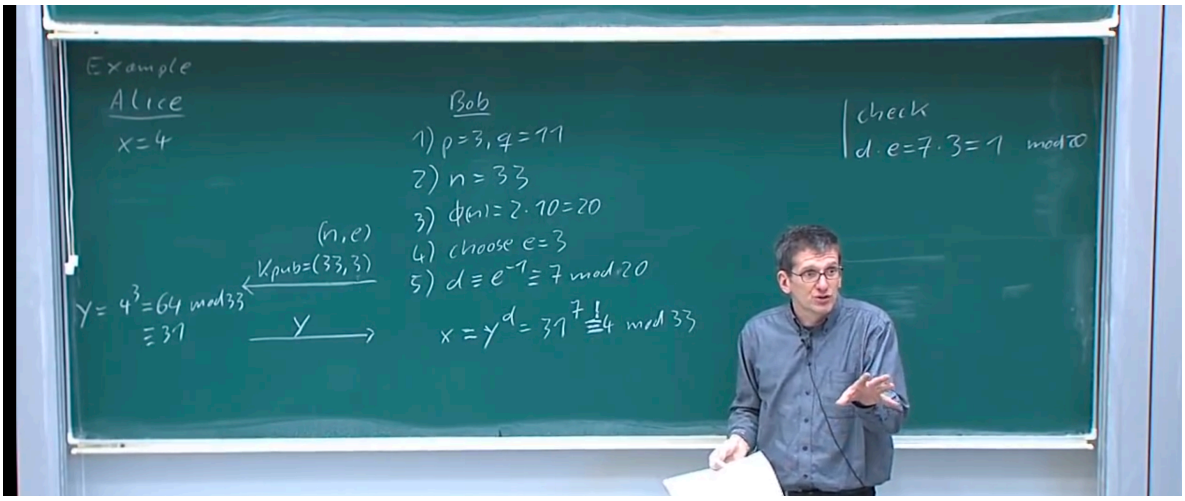
Say Alice and Bob want to converse.
now Bob is generating Key
for demonstration purpose we are choosing

1. P = 3 and Q = 11
2. n = 33
3. phi(n) = 20
4. say bob chooses e ie public key as 3
5. so private key will be 7 as 3*7 mod 20 == 1

so we have 2 keys, public Key as (3,33) and private key as (7)

so say we are trying to encrypt 4 ie 4 is our plaintext so

cipher text will be 4^3 mod 33 = 31
decryption on cipher text, 31^7 mod 33 => (-2)^7 mod 33 => by equivalence classes => -4*33 + 4 mod 33 => 4 mod 33 and as we know 4 is the cipher text.



Remarks:

1. **Proof of correctness of RSA.**
    1. Point here is that we have raise a number to a power mod n ie encryption and then we raised the output to a different number and we got the previous number How ?
    2. More formally:

> What is interesting is that the message x is first raised to the eth power during encryption and the result y is raised to the dth power in the decryption, and the result of this is again equal to the message x. Expressed as an equation, this process is:
>
> $$d_{k_{pr}}(y) = d_{k_{pr}}(e_{k_{pub}}(x)) \equiv (x^e)^d \equiv x^{de} \equiv x \bmod n. \qquad (7.3)$$
>
> This is the essence of RSA. We will now prove why the RSA scheme works.

Answer:

1. As we know d and e are inverse of each other so
    1. d*e = 1 mod phi(n)
    2. d*e = 1 + phi(n)*t
    3. x^d*e = x^(1+phi(n)*t) mod n
    4. x*x^phi(n)*t mod n
2. As we know phi(n) => (p-1)*(q-1)
3. x*x^(p-1)*(q-1)*t mod n
4. if x and n are prime then Euler algorithm holds ie x^phi(m) = 1 mod m so if x and n are prime then x*1 = x
5. if x and n are not prime then there must be some multiple of p or q is x so we can write x = p*r or x = q*r where r can be anything. so say x = p*r so gcd(x,q) = 1 so we can write equation 2 as x*x^(phi(q)*(p-1)*t) mod n
6. x*1^(p-1)*(t) mod n which is x only.

Proof. We need to show that decryption is the inverse function of encryption, $d_{k_{pr}}(e_{k_{pub}}(x)) = x$. We start with the construction rule for the public and private key: $d \cdot e \equiv 1 \bmod \Phi(n)$. By definition of the modulo operator, this is equivalent to:

$$d \cdot e = 1 + t \cdot \Phi(n),$$

where t is some integer. Inserting this expression in Eq. (7.3):

$$d_{k_{pr}}(y) \equiv x^{de} \equiv x^{1+t\cdot\Phi(n)} \equiv x^{t\cdot\Phi(n)} \cdot x^1 \equiv (x^{\Phi(n)})^t \cdot x \mod n. \tag{7.4}$$

This means we have to prove that $x \equiv (x^{\Phi(n)})^t \cdot x \mod n$. We use now Euler's Theorem from Sect. 6.3.3, which states that if $\gcd(x,n) = 1$ then $1 \equiv x^{\Phi(n)} \bmod n$. A minor generalization immediately follows:

$$1 \equiv 1^t \equiv (x^{\Phi(n)})^t \bmod n, \tag{7.5}$$

where t is any integer. For the proof we distinguish two cases:

First case: $\gcd(x,n) = 1$

Euler's Theorem holds here and we can insert Eq. (7.5) into (7.4):

$$d_{k_{pr}}(y) \equiv (x^{\Phi(n)})^t \cdot x \equiv 1 \cdot x \equiv x \bmod n. \qquad q.e.d.$$

This part of the proof establishes that decryption is actually the inverse function of encryption for plaintext values x which are relatively prime to the RSA modulus n. We provide now the proof for the other case.

Second case: $\gcd(x,n) = \gcd(x, p\cdot q) \neq 1$

Since p and q are primes, x must have one of them as a factor:

$$x = r \cdot p \quad \text{or} \quad x = s \cdot q,$$

where r, s are integers such that $r < q$ and $s < p$. Without loss of generality we assume $x = r \cdot p$, from which follows that $\gcd(x,q) = 1$. Euler's Theorem holds in the following form:

$$1 \equiv 1^t \equiv (x^{\Phi(q)})^t \bmod q,$$

where t is any positive integer. We now look at the term $(x^{\Phi(n)})^t$ again:

$$(x^{\Phi(n)})^t \equiv (x^{(q-1)(p-1)})^t \equiv ((x^{\Phi(q)})^t)^{p-1} \equiv 1^{(p-1)} = 1 \mod q.$$

Using the definition of the modulo operator, this is equivalent to:

$$(x^{\Phi(n)})^t = 1 + u \cdot q,$$

where u is some integer. We multiply this equation by x:

$$\begin{aligned} x \cdot (x^{\Phi(n)})^t &= x + x \cdot u \cdot q \\ &= x + (r \cdot p) \cdot u \cdot q \\ &= x + r \cdot u \cdot (p \cdot q) \\ &= x + r \cdot u \cdot n \\ x \cdot (x^{\Phi(n)})^t &\equiv x \mod n. \end{aligned} \tag{7.6}$$

Inserting Eq. (7.6) into Eq. (7.4) yields the desired result:

$$d_{k_{pr}} = (x^{\Phi(n)})^t \cdot x \equiv x \bmod n.$$

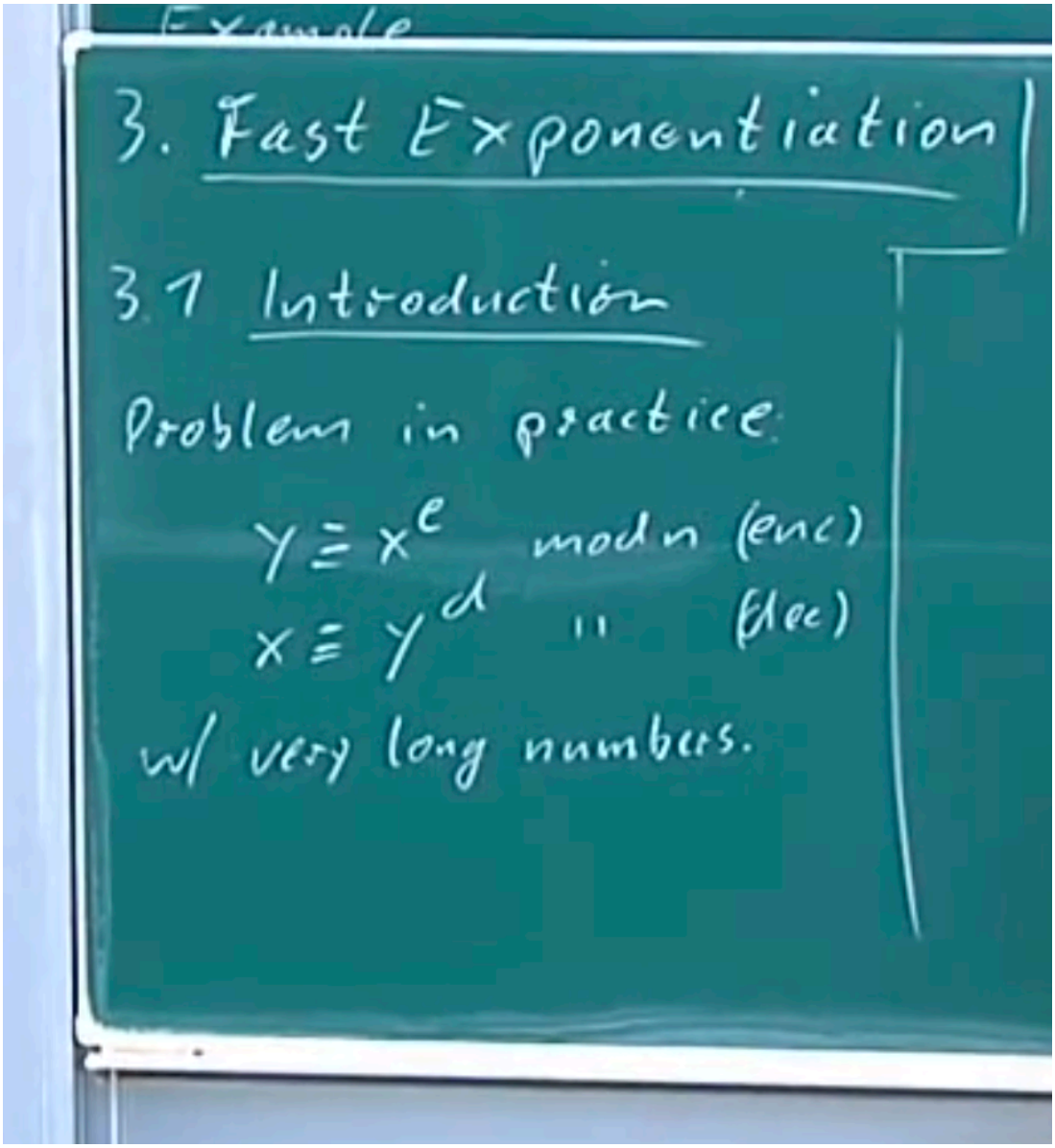
□

If this proof seems somewhat lengthy, please remember that the correctness of RSA is simply assured by Step 5 of the RSA key generation phase. The proof becomes simpler by using the Chinese Remainder Theorem which we have not introduced.

2. **Real World RSA parameters are very large ie:**

Practical RSA parameters are much, much larger. As can be seen from Table 6.1, the RSA modulus n should be at least 1024 bit long, which results in a bit length for p and q of 512. Here is an example of RSA parameters for this bit length:

Chapter3:
Fast Exponent:



say we want to compute X^4 then what are the ways ?

1. Naive way :
    1. X*X = X^2
    2. X^2*X = X^3
    3. X^3*X = X^4
2. Cost in above case is 3 Multiply

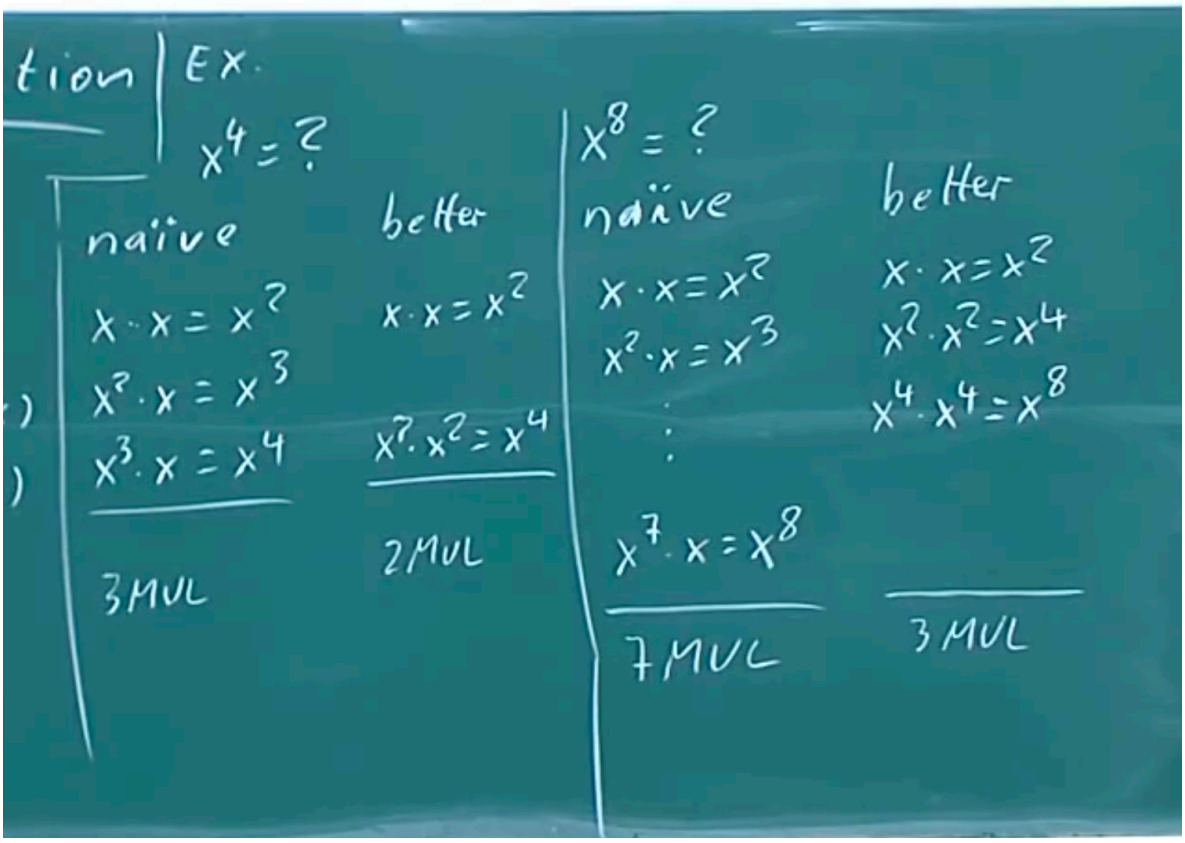
Are there any other ways to do this in less than 3 Multiplications ?

1. X*X = X^2
2. X^2*X^2 = X^4

above one is quite faster.

if we want to do X^8 ?

1. Naive way is simple so we can do that

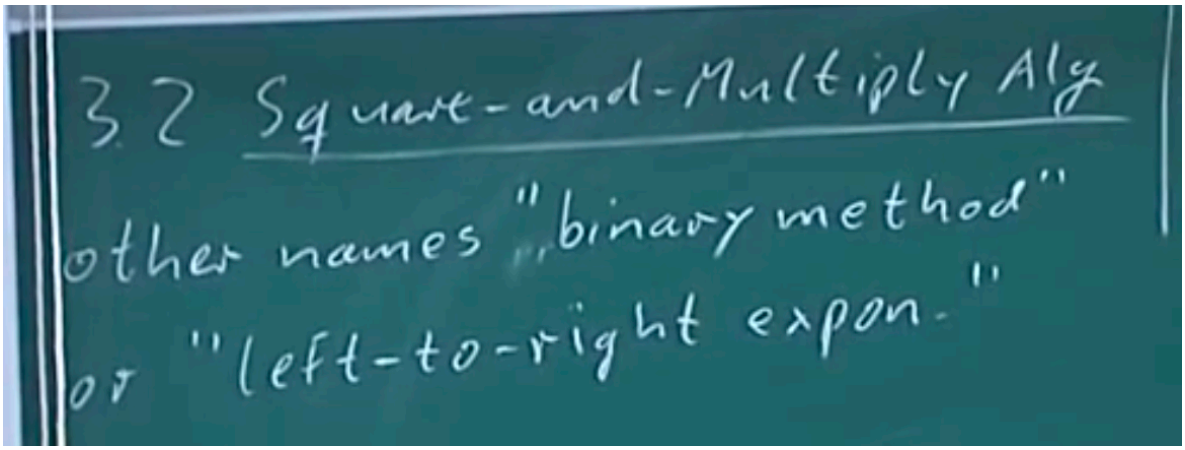


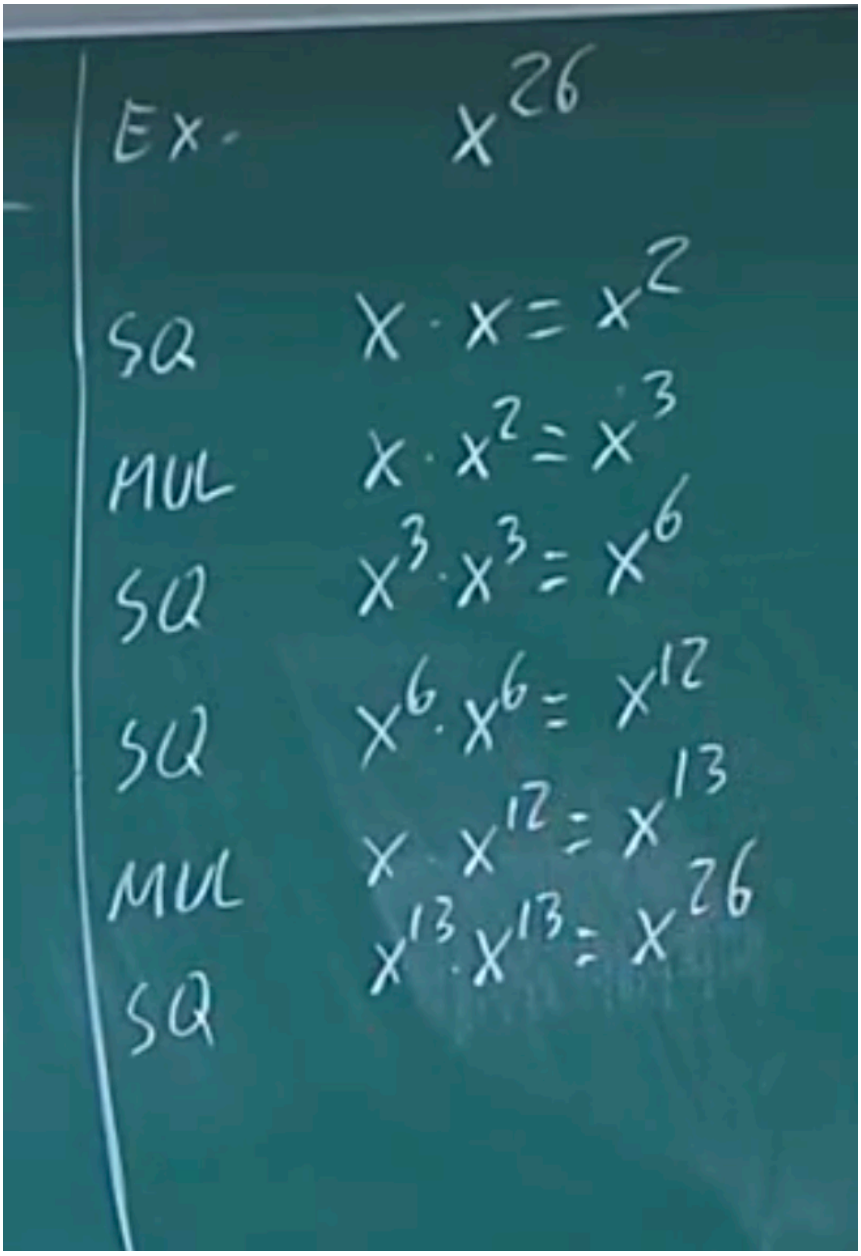
for X^8 we have done in quite less than the naive way.

Question: How are we dealing with X^2^1024 ?
if we are doing with Naive way then it requires **2^1024-1**
Better way is very fast and it requires 1024 steps or Multiplications.

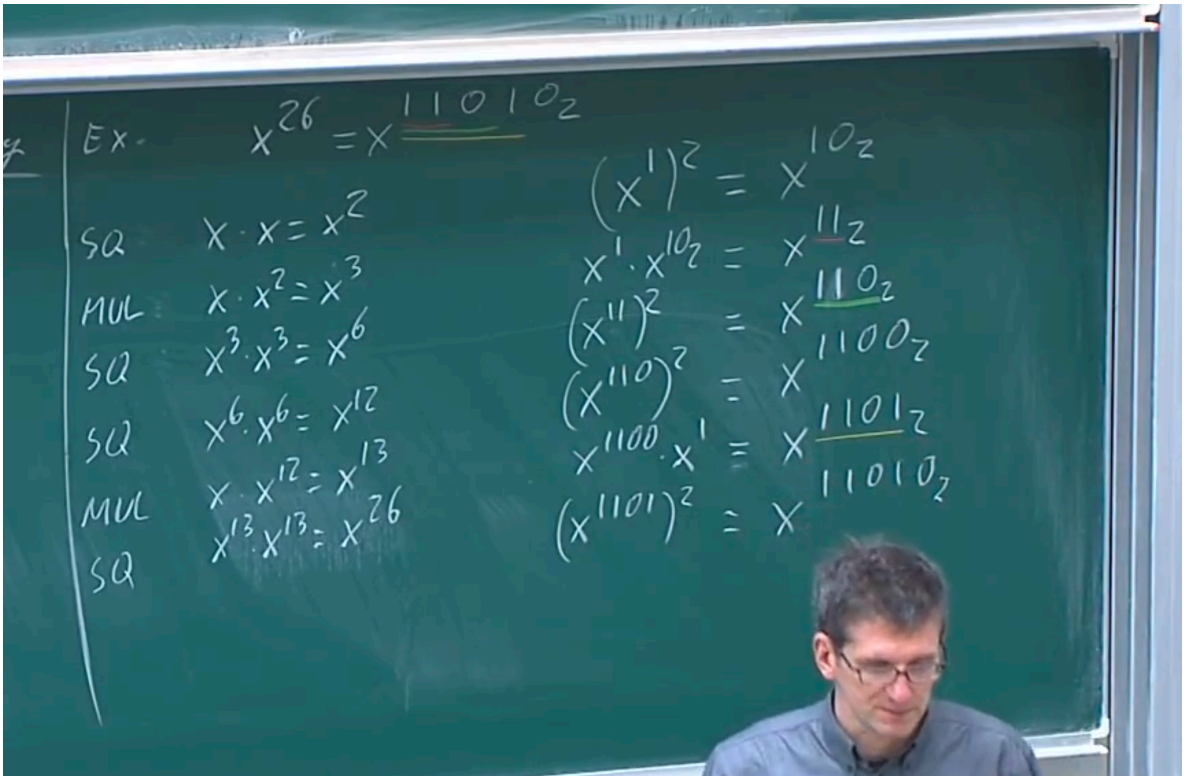
Actually with better way it is Logarithmic complexity
Question: what is the problem with this really cool trick ?
problem is exponent is always "power of 2"

So what is the solution ?
Chapter 3.2: Square and Multiply Algo:





So if we want to compute x26 we will do SQ, Mul, SQ, SQ, MUL, SQ
So we need to find when to Square and when to Multiply
Actually when to Mul and when to square depends on the Binary Representation of the 26.
26 = 1 1 0 1 0 in binary



Funda is simple:
11010 is binary representation of 26 so to find how to compute is simple ie leave 1st bit (MSB) so we are left with "**1010**"
when we get one, **square and multiply** and when we get zero **only square**
so
1010 -> SQ and MUL
1**0**10 -> SQ
10**1**0. -> SQ and MUL
101**0** -> SQ
so sequence is SQ MUL SQ SQ MUL SQ which is same as depicted in above image.

bits.

Example 7.4. We again consider the exponentiation x^{26}. For the square-and-multiply algorithm, the binary representation of the exponent is crucial:

$$x^{26} = x^{11010_2} = x^{(h_4 h_3 h_2 h_1 h_0)_2}.$$

The algorithm scans the exponent bits, starting on the left with h_4 and ending with the rightmost bit h_0.

Step		
#0	$x = x^{\mathbf{1}_2}$	inital setting, bit processed: $h_4 = 1$
#1*a*	$(x^1)^2 = x^2 = x^{\mathbf{10}_2}$	SQ, bit processed: h_3
#1*b*	$x^2 \cdot x = x^3 = x^{10_2} x^{1_2} = x^{\mathbf{11}_2}$	MUL, since $h_3 = 1$
#2*a*	$(x^3)^2 = x^6 = (x^{11_2})^2 = x^{\mathbf{110}_2}$	SQ, bit processed: h_2
#2*b*		no MUL, since $h_2 = 0$
#3*a*	$(x^6)^2 = x^{12} = (x^{110_2})^2 = x^{\mathbf{1100}_2}$	SQ, bit processed: h_1
#3*b*	$x^{12} \cdot x = x^{13} = x^{1100_2} x^{1_2} = x^{\mathbf{1101}_2}$	MUL, since $h_1 = 1$
#4*a*	$(x^{13})^2 = x^{26} = (x^{1101_2})^2 = x^{\mathbf{11010}_2}$	SQ, bit processed: h_0
#4*b*		no MUL, since $h_0 = 0$

To understand the algorithm it is helpful to closely observe how the binary representation of the exponent evolves. We see that the first basic operation, squaring, results in a left shift of the exponent, with a 0 put in the rightmost position. The other basic operation, multiplication by x, results in filling a 1 into the rightmost position of the exponent. Compare how the highlighted exponents change from iteration to iteration.

◇

Here is the pseudo code for the square-and-multiply algorithm: