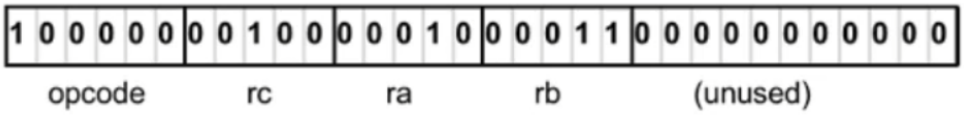
The complete quantum computing course

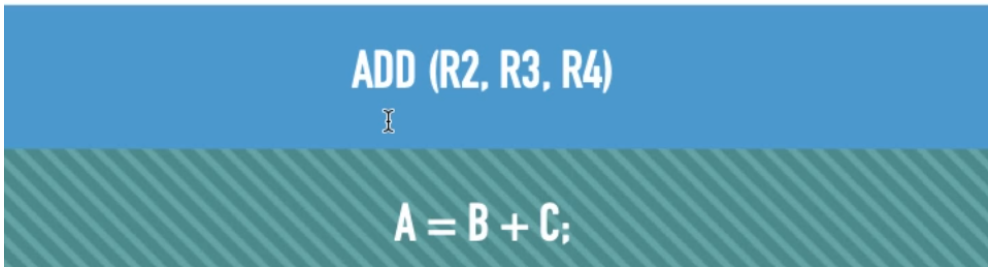
https://www.udemy.com/course/quantum-computers/

Mathematical Foundations

Bits vs qubits

- **Bits:** 1's and 0's. Transistors can eventually generate a 1 or 0 (is there electricity in this transistor - yes 1 or no 0?). This is a **machine language** - even with (any) programming languages it does not matter. It only understands 1's and 0's.



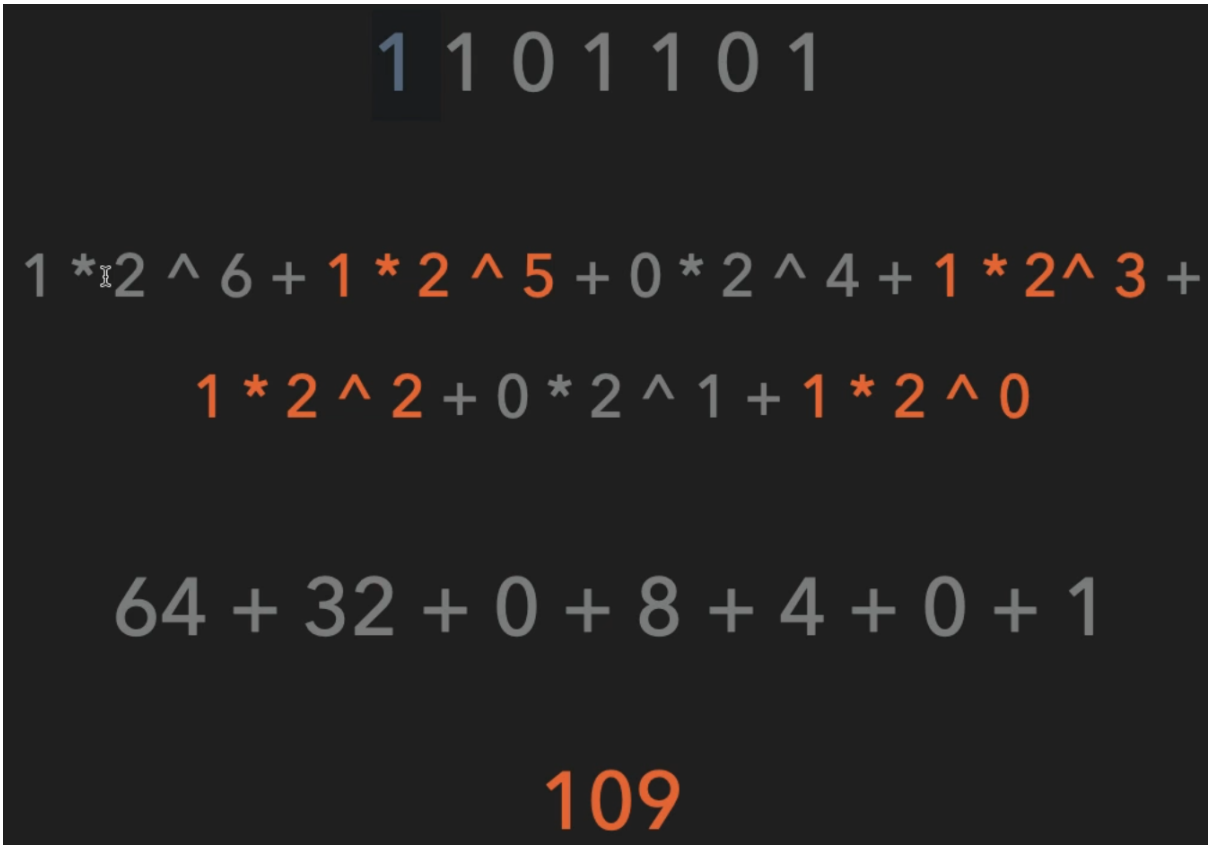


Programming in 1's and 0's is very difficult to do. The machine will do this "for us" -> thanks to processors and transistors. I only understand if there is electricity in a transistor or not. Program logic (like a = b + c) makes a lot more sense for humans than programming in bits.

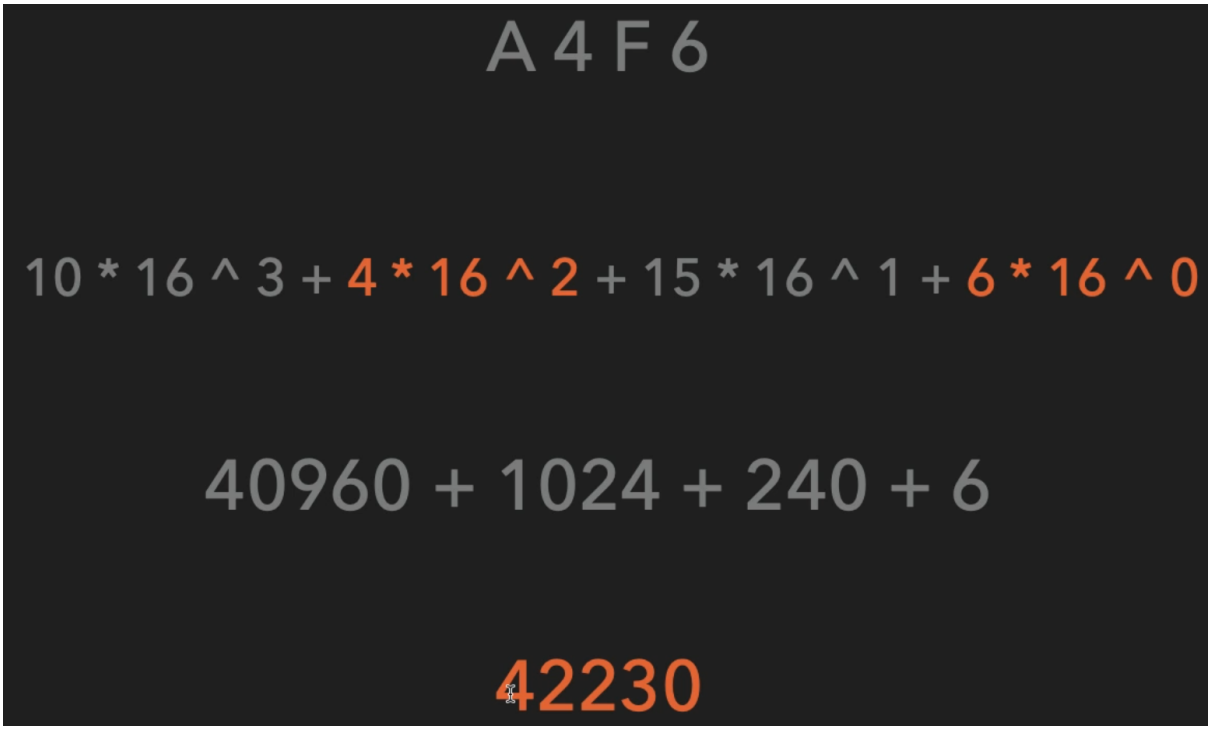
0's and 1's can be displayed in a **binary form**. This is related to the decimal system.

You can now do this for **binary numbers as well (binary to decimal).** Binary only has two numbers: 0's and 1's. So you need to come up with a calculation that can convert binary into decimal (and the other way around).

Binary 1101101 = decimal 109.



The same goes from **hexadecimal** to **binary** (and the other way around):

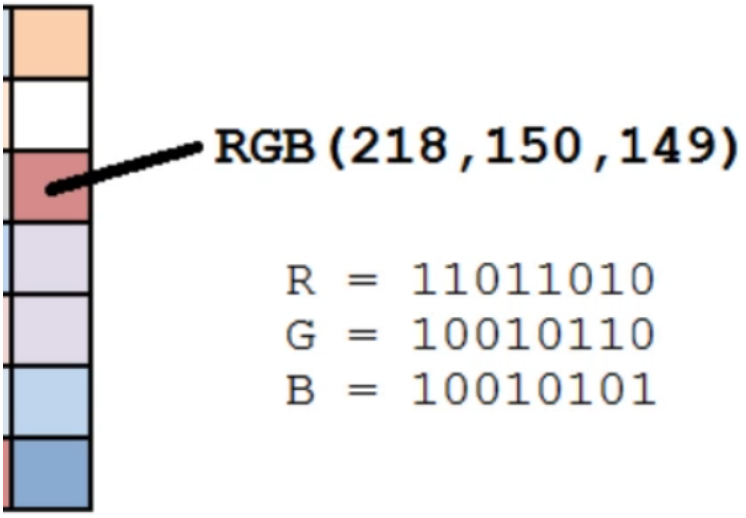


Hexadecimal A4F6 = 42230 decimal.

Decimal - Binary - Octal - Hex – ASCII Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

Colors can also be converted from binary to color (and the other way around):



A computer can only understand one single state at a time - a number from 0 to 256 due to our 0's and 1's system. The calculation of a regular computer can only represent **only a single number AT THE SAME TIME**.

Qubits: A Quantum Computer can use all these states at the same time instead of only one number at a time. The quantum computer can represent every single number between 0 and 256 **AT THE SAME TIME**.

Probability

Mathematical galore! Probability and binary logic.

- P(A) = Probability of A happening.
- P(A AND B): A and B are happening at the same time.
- P((A AND B) OR C): A and B happen at the same time, or C happens, or all at the same time? Who knows?

Binary logic:
- AND = both happen.
- OR = either option happens (the one OR the other).
- P(A AND B) = 0: **mutually exclusive**. Both A and B cannot happen at the same time.
    - P(A or B) = P(A) + P(B): what is the probability of one of those happening? They are dependent on each other.
- P(A AND B) = P(A) x P(B): **independent**. Either A or B can happen - they are not dependent on each other (by the AND). If it is mutually exclusive it is never possible - but by multiplying both options, both options are independent of each other.
    - P(A OR B) = P(A) + P(B) - P(A AND B): now they are **independent** of each other.

Example:
P(A) = 30%
P(B) = 40%
P(A OR B) = P(A) + P(B) - P(A AND B)
= 0.3 + 0.4 - (0.3 * 0.4)
= 0.58. 58 % of chance either A or B is happening. This is the **probability**.

Statistics

15, 22, 33, 40, 55, 61, 79

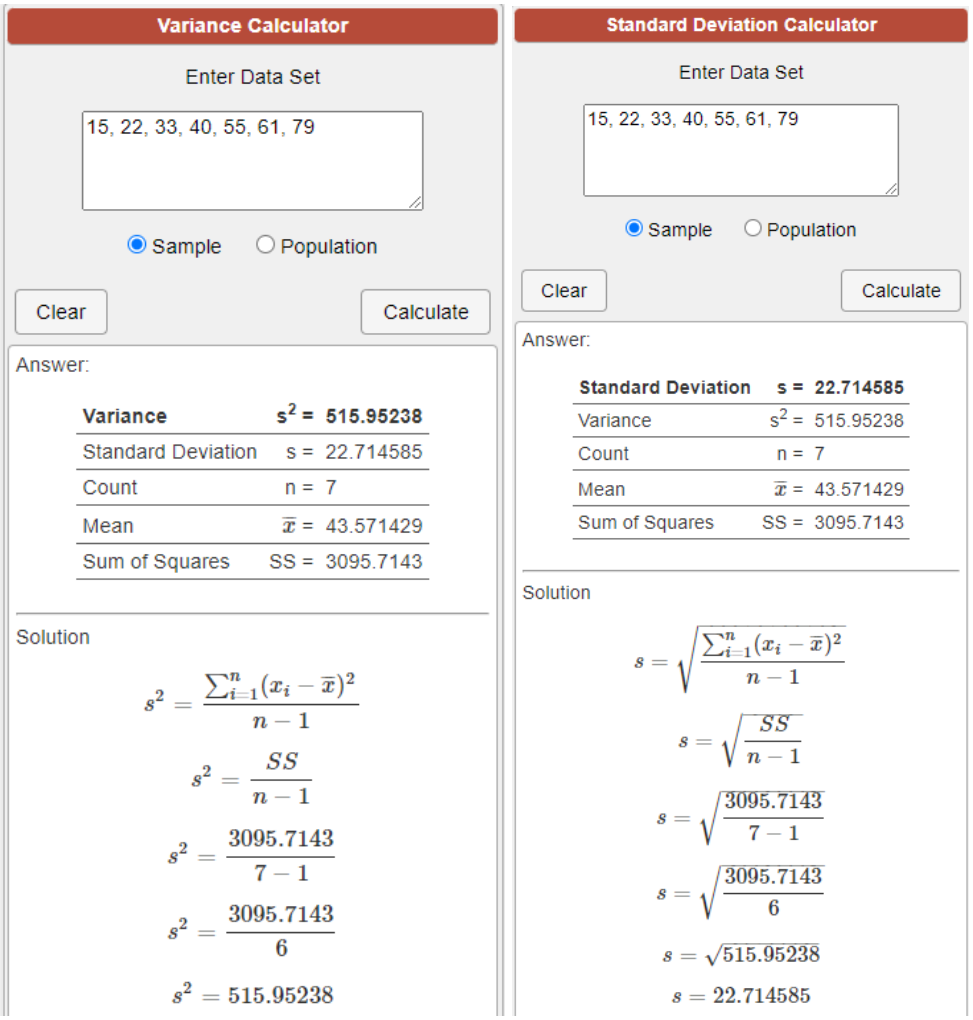
- **Min:** 15.
- **Max:** 79.
- **Mean:** 43,57 ~ (add all numbers up & divide by length of array).
- **Median:** 40.

Consider we have the mean - we want to know how many different points we have in an array. We have 7 numbers now - easy - but what happens if you have millions of numbers?
- **Variance:** in a given list you have an amount of variance. This number on its own doesn't mean a lot. It can be used to compare other variances. Low variance means the data points are generally **similar** and **do not vary widely from the mean**.
- **Standard Deviation:** measure of the amount of variation or dispersion of a set of values. Thus - you can also use this to determine variation. A **low standard deviation** means data points are **close to the mean or average value**.

For our above list the square root of all the numbers would be 515.95 square root.
The standard deviation would be 22.71.
$(15 - 43.57)^2 = 816.2449$

$(22 - 43.57)^2 = 465.2649$
$(33 - 43.57)^2 = 111.7249$
$(40 - 43.57)^2 = 12.7449$
$(55 - 43.57)^2 = 130.6449$
$(61 - 43.57)^2 = 303.8049$
$(79 - 43.57)^2 = 1,255.2849$

1,405.9796 + 1,689.7347 = 3,095.7143. Now take the amount of numbers in your list, minus one, and you have a variance! The standard deviation formula is fairly similar - except for now also including the square root of the variance result.



Complex numbers

$i \rightarrow \sqrt{-1}$ (Imaginary number)

$a + bi$ (Complex number)

$\sqrt{-9} = \sqrt{9} * \sqrt{-1}$

$= 3i$

Examples:

$(3+2i) + (5+4i) = (8+6i)$

$(3-2i) + (-5+4i) = (-2+2i)$

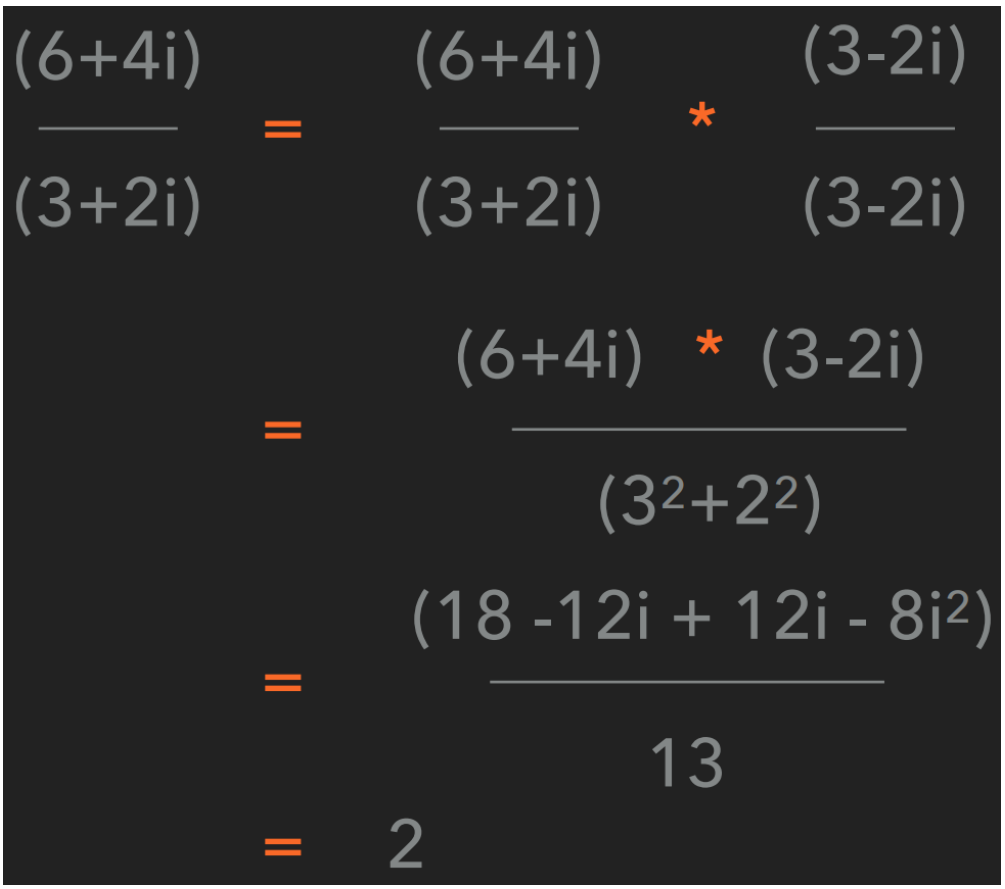
$(3+2i) - (5+4i) = (-2-2i)$

$5 * (3+2i) = (15+10i)$

$(3+2i) / 3 = (1 + \frac{2i}{3})$

- **Complex Conjugate:** change the sign of the imaginary part. + becomes -, - becomes +. This will be used **a lot** in quantum computations.
    - (3+2i) = (3-2i). Only the **imaginary part** - not the **real** part.
- **Squared magnitude:** complex number * complex conjugate!

$|3+2i|^2 \rightarrow 3^2 + 2^2$

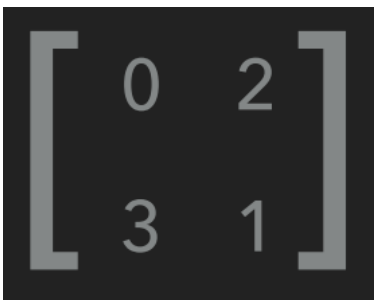


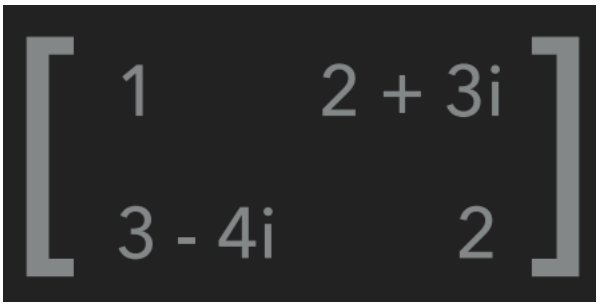
Use the complex conjugate of the nominator (noemer) and use this to calculate!

Matrix

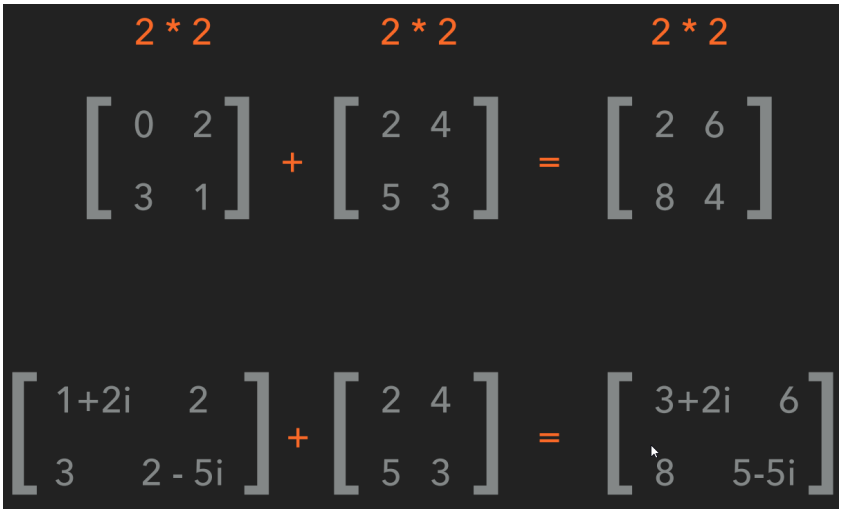
Basic understandings:

- **Rows vs Columns**. 2 rows, 2 columns = 2 x 2 matrix. *It doesn't matter if you have real numbers, or complex numbers.*

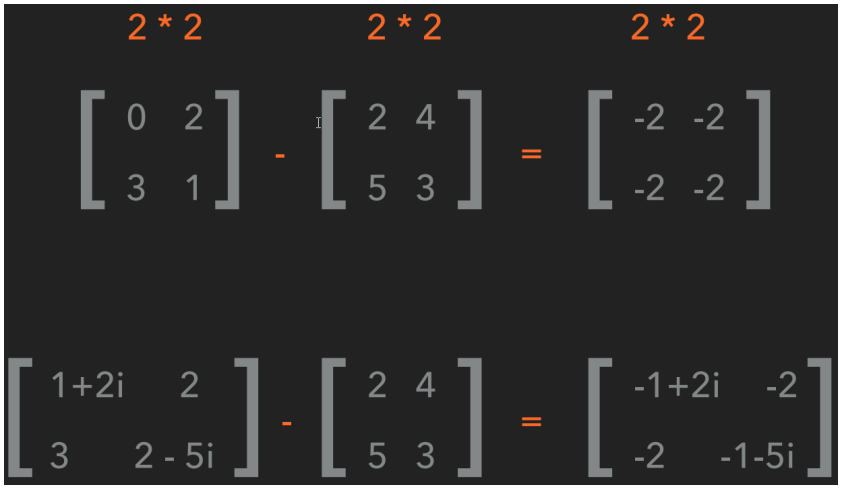




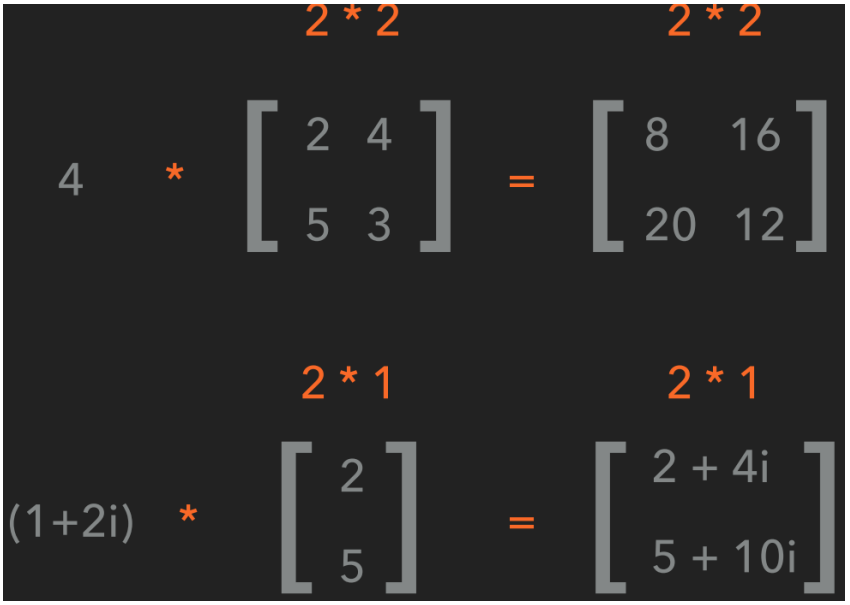
- **Calculations:** calculating matrices
    - **Sommation**:



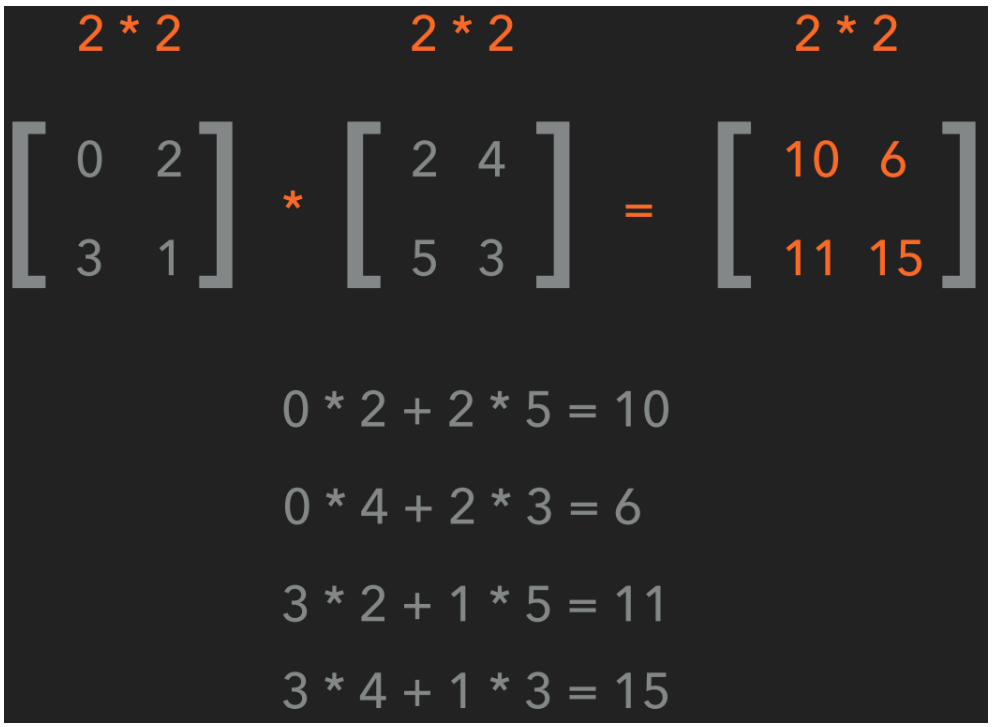
- **Subtract:**



- **Multiply:** a **scaler** is used in the front (real or complex number):

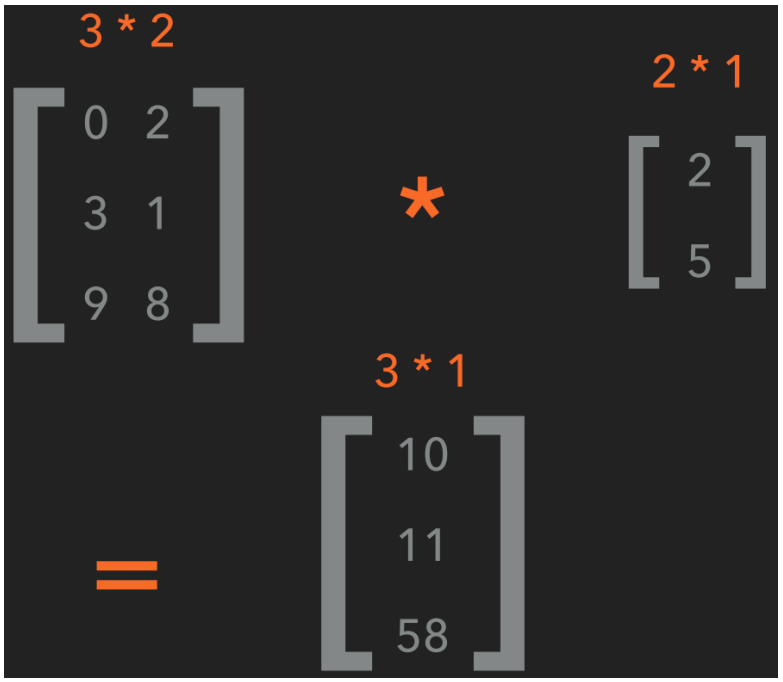


- **Matrix multiplying:** it is more complex - but **this is what a Quantum Computer does** in order to calculate things.



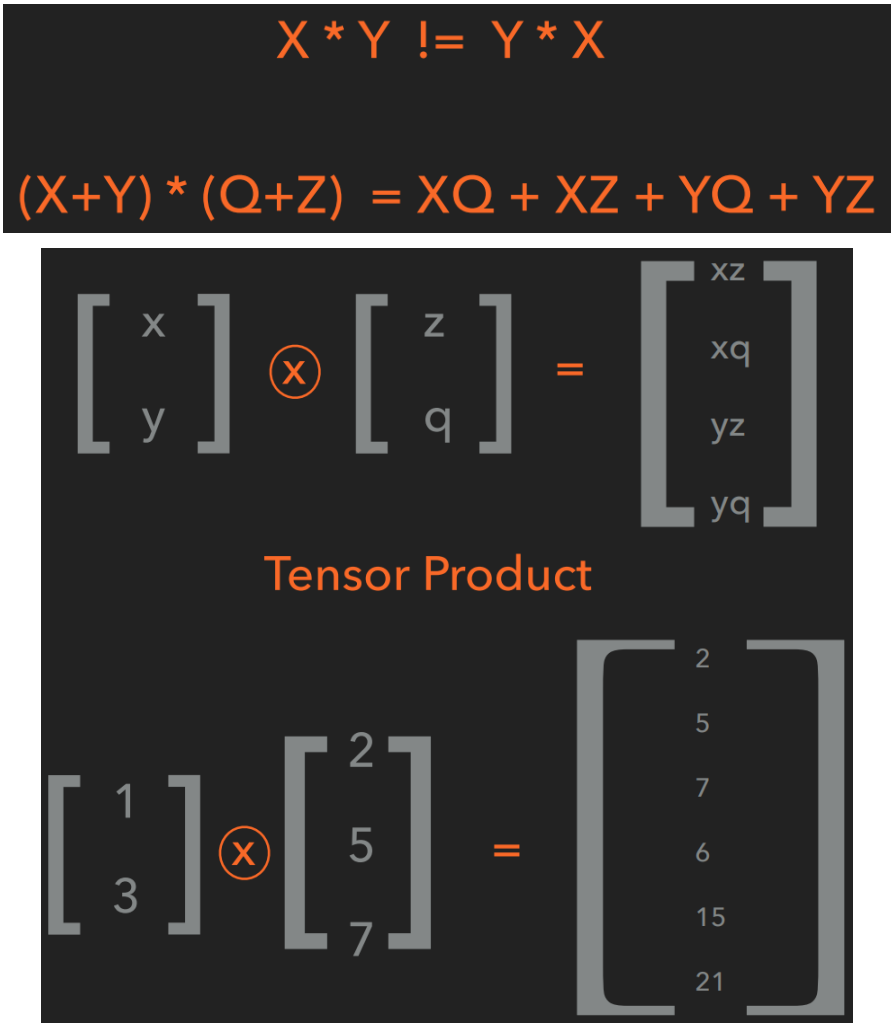
How do you do this: take the **first row** and the **first column** of the two matrices. First result: 0 * 2 + 2 * 5 = 10. Second result is calculated by taking the **first row** and the **second column**. Next is the **second row** with the **first column**, and finally the **second row with the second column.**

The difficulty starts when the **matrices do not align** with the same columns or rows. To calculate such matrices you always take the **rows** of the first matrix and the **columns** of the second matrix. In the below example you first take the **three rows** and the **one column**:

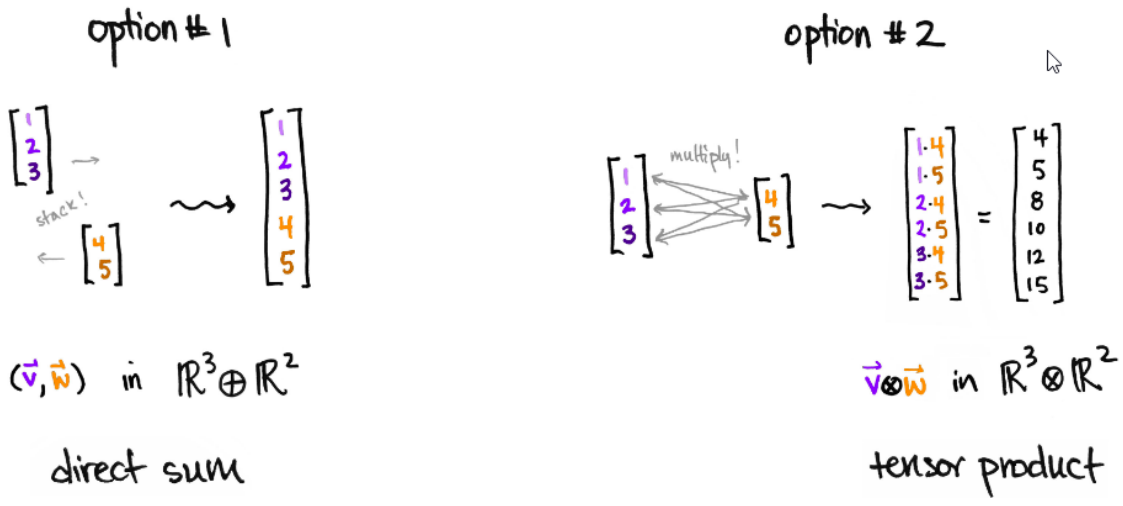


Example calculation: (0 * 2) + (2 * 5) = 10. (3 * 2) + (1 * 5) = 11. (9 * 2) + (8 * 5) = 58.

- **Tensor Product:** you can make a new matrix from two matrices!

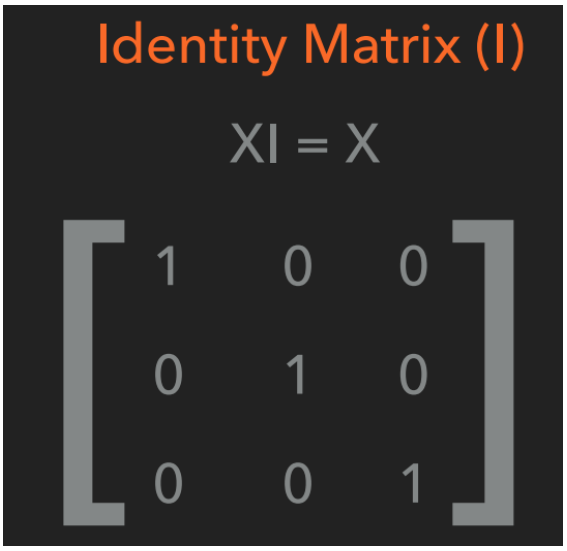


The difference between a direct sum vs tensor product:



Special matrix

- **Identity matrix:** (I), XI = X. Don't forget about the 1's. This matrix always is identical: 1's in the middle row, 0's everywhere else.



- **Inverse (real number) vs inverse (matrix):** Why do we need an inverse matrix? Because we **cannot divide** - but you can use a multiplication by an inverse for this.

Example if we would not be able to divide with regular numbers either. How can I share 10 apples with 2 people? I cannot divide, so i take a **reciprocal** of 2 (which is 0.5) and multiply this with the amount of apples. 10 x 0.5 = 5. Each person gets 5 apples!

When you calculate the **reciprocal** of a number - you always **get 1**. When you calculate the inverse of a matrix you will always **end up with an Identity matrix**! How do we calculate this reciprocal?

2's reciprocal is ½. This will directly calculate to 0.5. A reciprocal is always the same for each regular number: put a '1' in the numerator, while the number itself is the denominator. Since we cannot divide with a matrix - we can only use a -1 variant as such. This variant will **always be the identity matrix** with 0's and 1's.

Just like a **number** has a reciprocal ...

Reciprocal of a Number (note: $\frac{1}{8}$ can also be written 8^{-1})

... a **matrix** has an **inverse :**

Inverse of a Matrix

We write A^{-1} instead of $\frac{1}{A}$ because we don't divide by a matrix!

OK, how do we calculate the inverse?

Well, for a 2x2 matrix the inverse is:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad-bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

In other words: **swap** the positions of a and d, put **negatives** in front of b and c, and **divide** everything by **ad−bc** .

Note: **ad−bc** is called the determinant.

Let us try an example:

$$\begin{bmatrix} 4 & 7 \\ 2 & 6 \end{bmatrix}^{-1} = \frac{1}{4\times6-7\times2}\begin{bmatrix} 6 & -7 \\ -2 & 4 \end{bmatrix}$$

$$= \frac{1}{10}\begin{bmatrix} 6 & -7 \\ -2 & 4 \end{bmatrix}$$

$$= \begin{bmatrix} 0{,}6 & -0{,}7 \\ -0{,}2 & 0{,}4 \end{bmatrix}$$

How do we know this is the right answer?

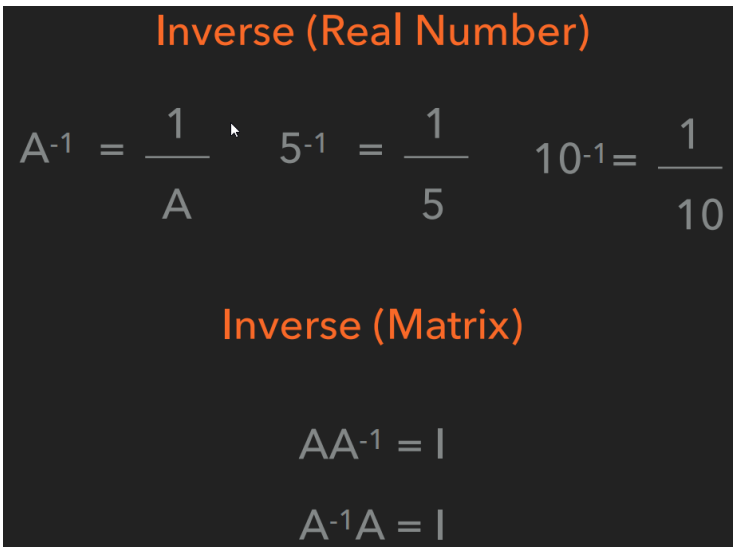
Remember it must be true that: $AA^{-1} = \mathbf{I}$

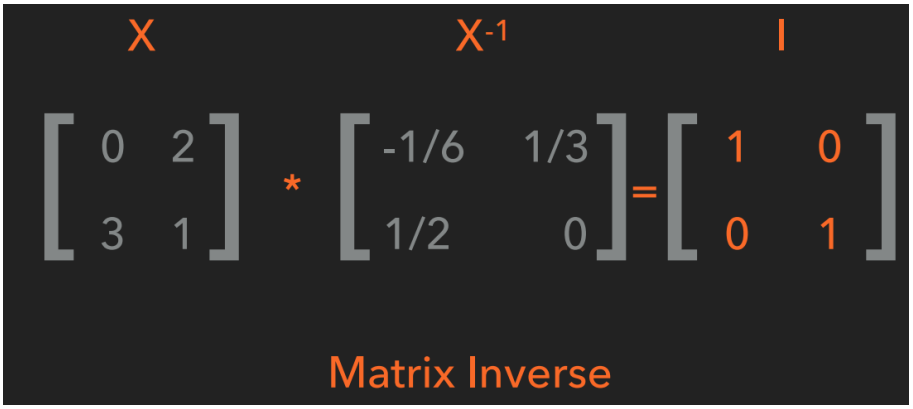
So, let us check to see what happens when we multiply the matrix by its inverse:

$$\begin{bmatrix} 4 & 7 \\ 2 & 6 \end{bmatrix}\begin{bmatrix} 0{,}6 & -0{,}7 \\ -0{,}2 & 0{,}4 \end{bmatrix} = \begin{bmatrix} 4\times0{,}6+7\times-0{,}2 & 4\times-0{,}7+7\times0{,}4 \\ 2\times0{,}6+6\times-0{,}2 & 2\times-0{,}7+6\times0{,}4 \end{bmatrix}$$

$$= \begin{bmatrix} 2{,}4-1{,}4 & -2{,}8+2{,}8 \\ 1{,}2-1{,}2 & -1{,}4+2{,}4 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$





A real-life example if we cannot simply divide and need to use matrices:

A group took a trip on a bus, at $3 per child and $3,20 per adult for a total of $118,40. They took the train back at $3,50 per child and $3,60 per adult for a total of $135,20.

How many children, and how many adults?

$$\begin{bmatrix} 3 & 3{,}5 \\ 3{,}2 & 3{,}6 \end{bmatrix}^{-1} = \frac{1}{3\times3{,}6-3{,}5\times3{,}2}\begin{bmatrix} 3{,}6 & -3{,}5 \\ -3{,}2 & 3 \end{bmatrix}$$

$$= \begin{bmatrix} -9 & 8{,}75 \\ 8 & -7{,}5 \end{bmatrix}$$

$$X = BA^{-1}$$

$$\begin{bmatrix} x_1 & x_2 \end{bmatrix} = \begin{bmatrix} 118{,}4 & 135{,}2 \end{bmatrix}\begin{bmatrix} -9 & 8{,}75 \\ 8 & -7{,}5 \end{bmatrix}$$

$$= \begin{bmatrix} 118{,}4\times-9 + 135{,}2\times8 & 118{,}4\times8{,}75 + 135{,}2\times-7{,}5 \end{bmatrix}$$

$$= \begin{bmatrix} 16 & 22 \end{bmatrix}$$

There were 16 children and 22 adults!

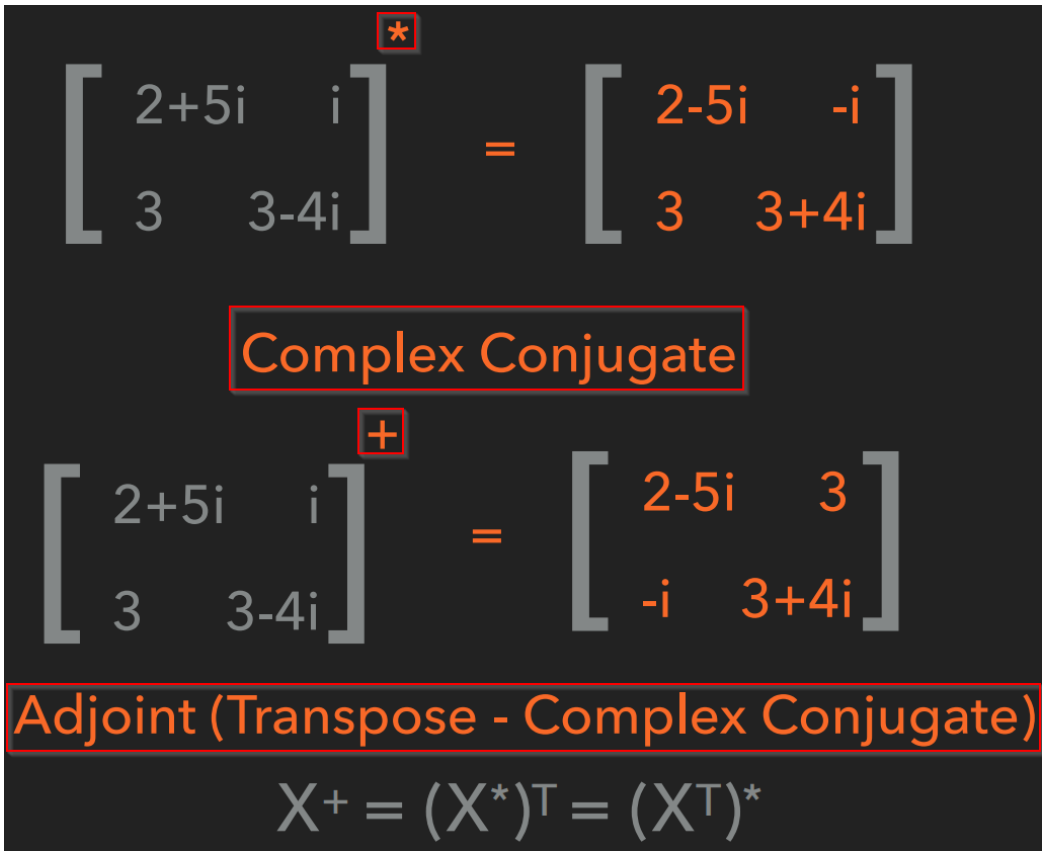
- **Transpose:** switch a result from a row to a column. The same can be done with matrices.

$$\begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix}^{T} = \begin{bmatrix} 0 & 3 \\ 2 & 1 \end{bmatrix}$$

Transpose

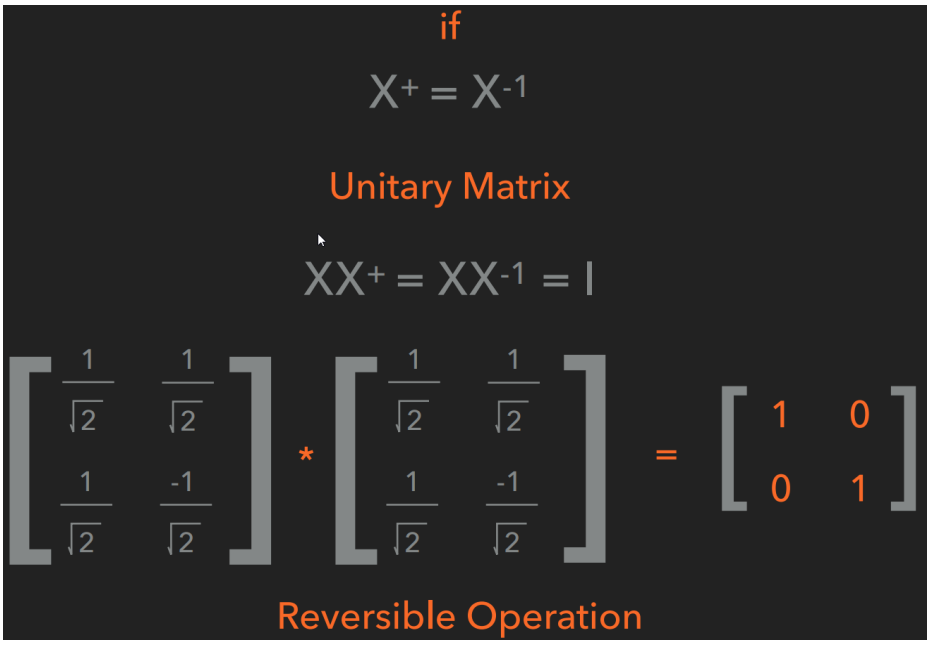
$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}^{T} = \begin{bmatrix} 0 & 3 & 6 \\ 1 & 4 & 7 \\ 2 & 5 & 8 \end{bmatrix}$$

- **Complex conjugate vs Adjoints:** transpose - complex conjugate!

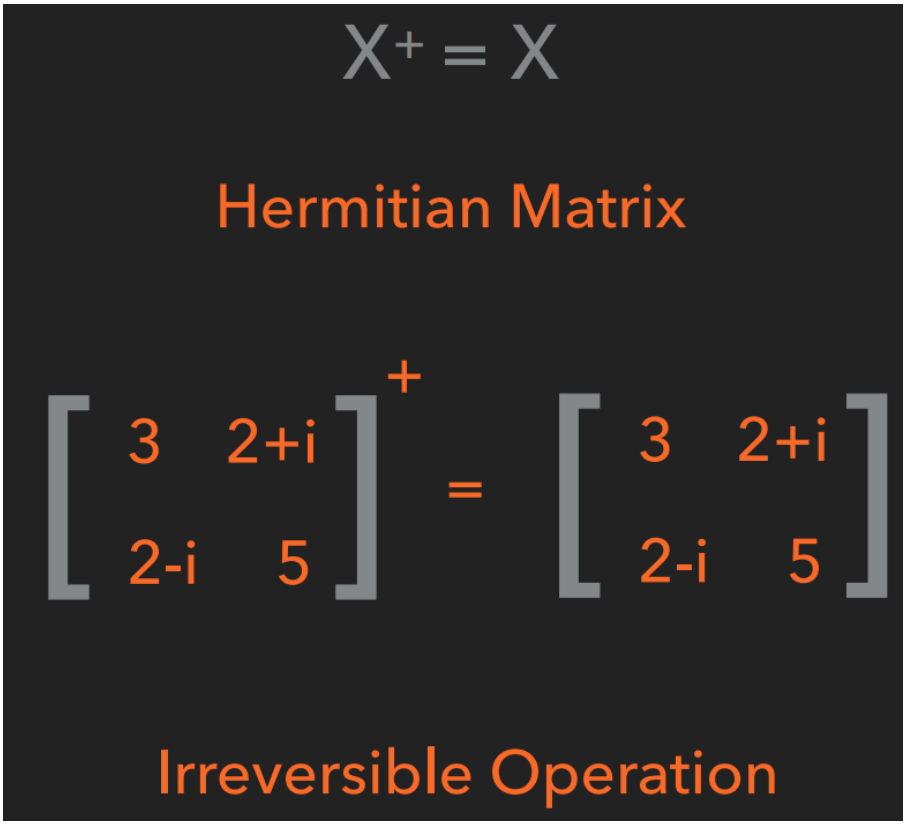


Linear transformation

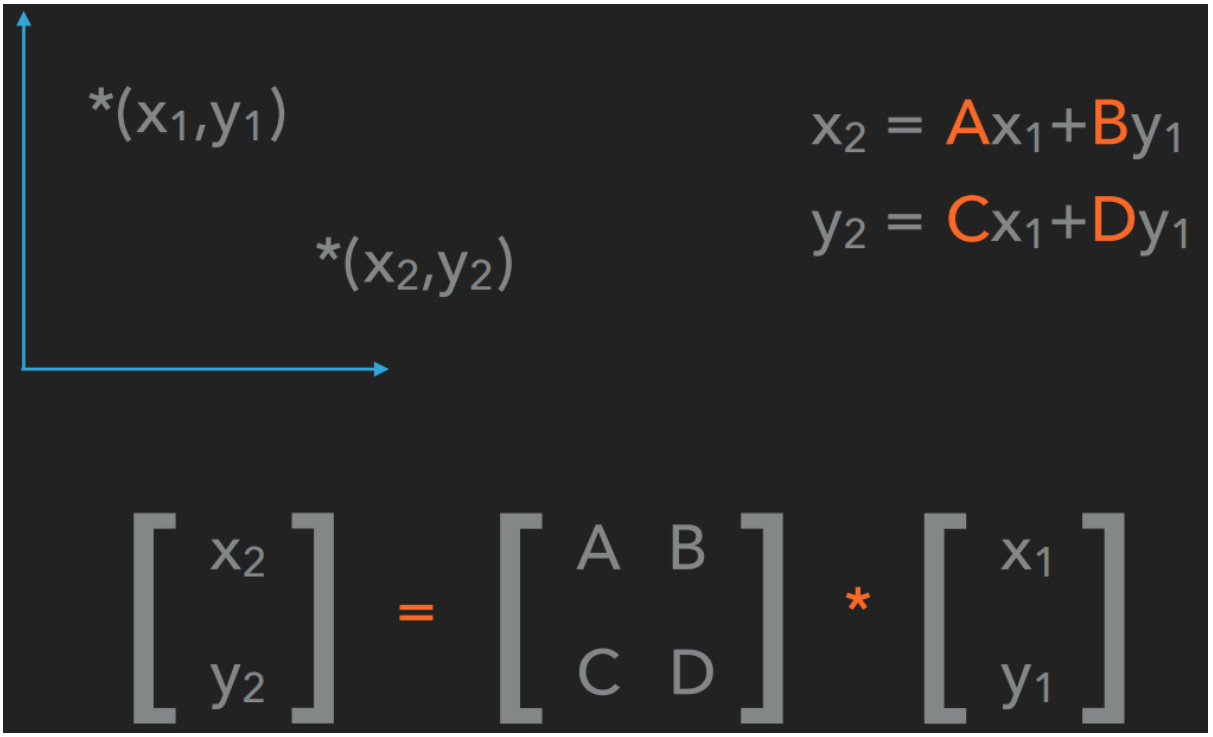
- **Unitary matrix:** if an adjoint / conjugate transpose is equal to an inverse you end up with an identity matrix (thus - the matrix result is the same).



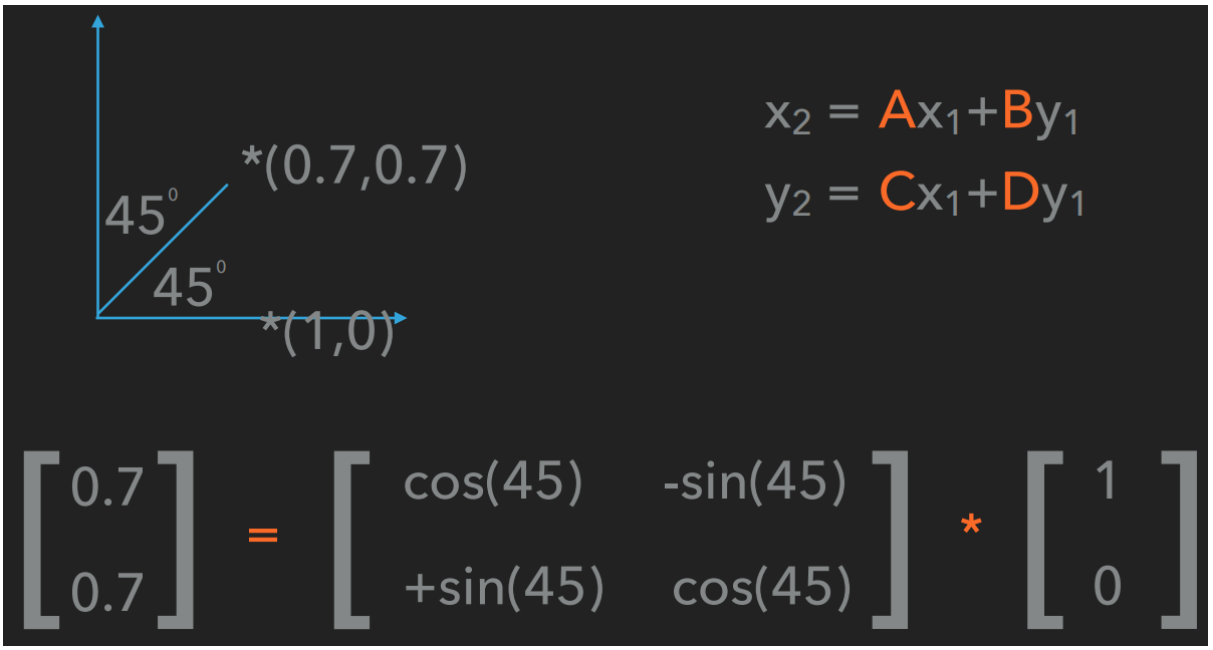
- **Hermitian Matrix:** Adjoint equals X itself (you cannot reverse this operation).



- **Vector:** Column matrix (Quantum Computing). It is simply a single-column matrix.
- **Linear transformation / linear map:** we can represent the same thing with a column matrix once you do the calculation.



To do this you need to do **a rotation** - with cos and sin). Not a huge deep-dive for quantum computing. This formulation exists and how linear transformation is done. This kind of rotation will happen in Qubits all the time in order to **transform / rotate state**:

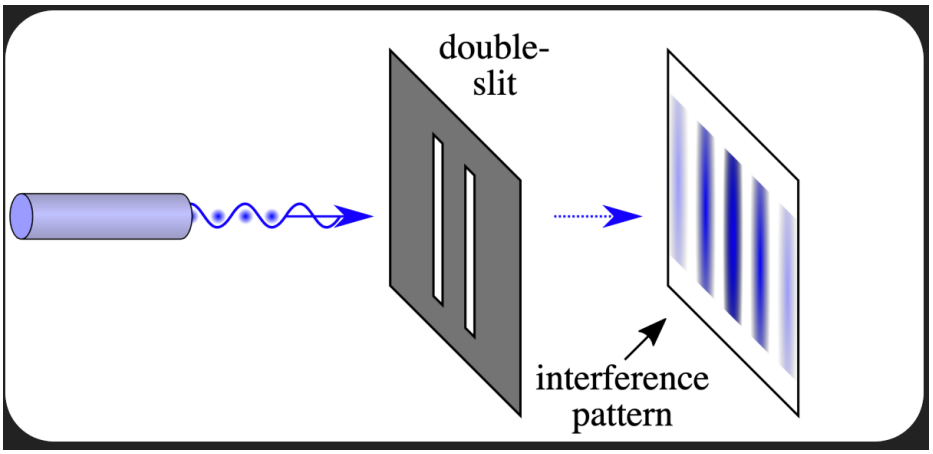


Qubit and physics

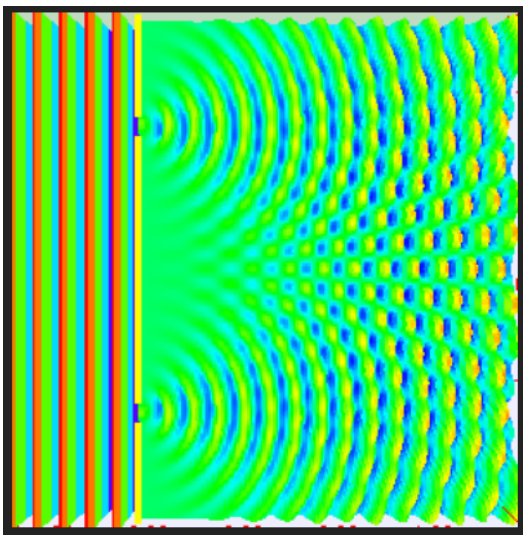
Superposition and interference

Young generated a theory about 100 years ago about light. A candle light / laser light has photons. When dealing with quantum physics we are going to deal with subatomic particles (electrons, for example).

A double-slit is a card without any slit. If you give a light to this card it doesn't go beyond this card. No light on the end of the line. If you have one slit, light will get through. One ray of light is reflected. If you have a double-slit you will get a distinct pattern which doesn't make sense. One slit = one ray of light. Two slits = five rays of light.



Light is like a ray instead of a particle - this theory solves this problem as multiple rays are reflected, thus light is a wave-like structure. The light waves are interfering with each other as such:



This interference pattern is noticeable once the light passes through the slits. The waves are colliding with each other and creating an interference pattern. The measuring for this is an observer for this experiment. In this experiment a similar setup is created - and the wave of light is suddenly only behaving as a particle. As such you will only create two rays of light, opposed to the previous statement. If no observation is done a multitude of rays of light is noticeable. This is called a **collapse of the wave function**. Even when sending single photons - this interference pattern is still occurring. The logical explanation: the photon is interfering with itself. It is in a superposition - in both the left slit and the right slit. When we observe this, this superposition is broken. When it reaches the position (the slits), it lands somewhere and creates this interference pattern again.

- **Superposition:** qubit can have one and zero, and anything in between. A qubit can have any position at the same time - but this is only a probability. A single photon can be on the left or right side, it can land anywhere on the reflected side.
- **Interference:** light / photons can interfere with each other (see drawing above).
- **Collapse:** without the observation, the light interferes again and collapses in multiple positions.

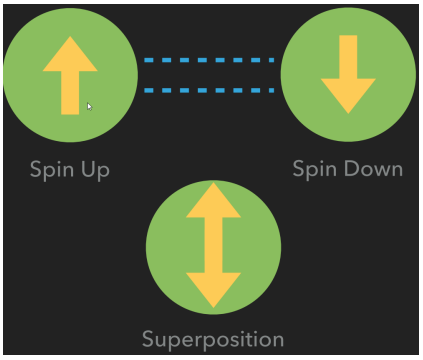
Entanglement

We'll be talking about electrons and atoms. The below picture displays an atom with electrons (green dots).

How do we know this position of each electron / object? Latitude, longitude? We can try to determine their position according to coordinates - and you have different values to determine this position.

- **Principal.**
- **Orbital Angular Momentum.**
- **Magnetic.**
- **Electron Spin.**

As this is not a quantum physics class - we will not go further in depth with these terms. We will look at the Electron spin: ***spin up or spin down***.

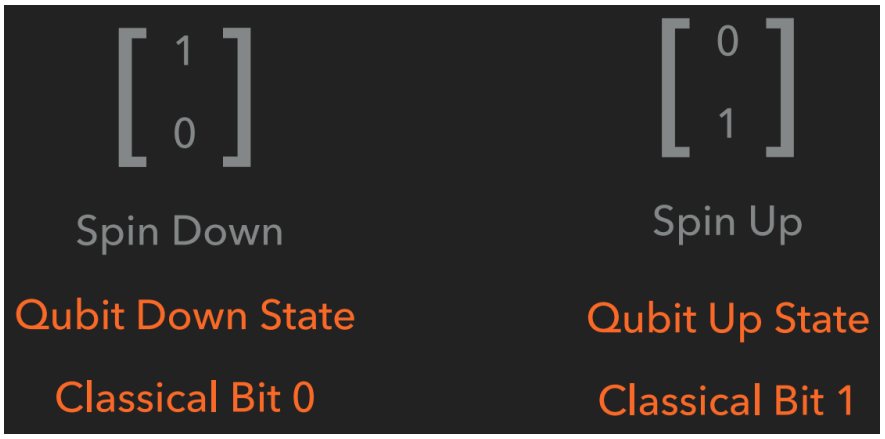


An electron is always in the superposition. We cannot determine if it is either in a spin up or spin down state - as it can be in both states. Before measurement **we don't know this position**. We can only say it is either spin up or spin down, but actually is in a superposition. After we measure this - we get spin up for the first time, afterwards it will be in a spin up position again. Once this state is determined, it will always be in the same position afterwards. *Example: a cat in a box. Radio-active substances near the cat box are closed. 50% chance this radio-active substance will decay, but we don't know it (poison the cat). Before opening the box: the cat is in a superposition of being alive or dead. Once we open the box we know what the actual position of the cat is (dead or alive? Spin up or spin down state?).* At the end of the measurement it is either spin up or spin down.

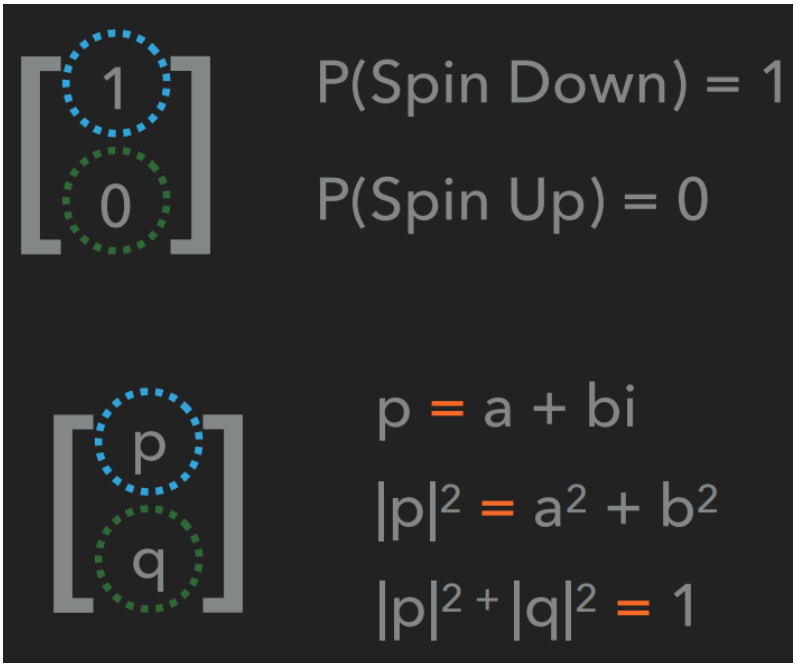
Entanglement refers to the fact that each spin is correlated to one another. One will spin up, while the other qubit will spin down. It doesn't matter where they are located as they entangle with each other. It is faster than the speed of light as it happens instantaneously. Einstein referred to this as a "spooky action at a distance". We can correlate both qubits as both up and down since they can be entangled with each other. **It is just a correlation between two qubits, photons, electrons,...** Once we measure one qubit - we know the state of the other qubit. We will know the result of the other state immediately!

Qubit state

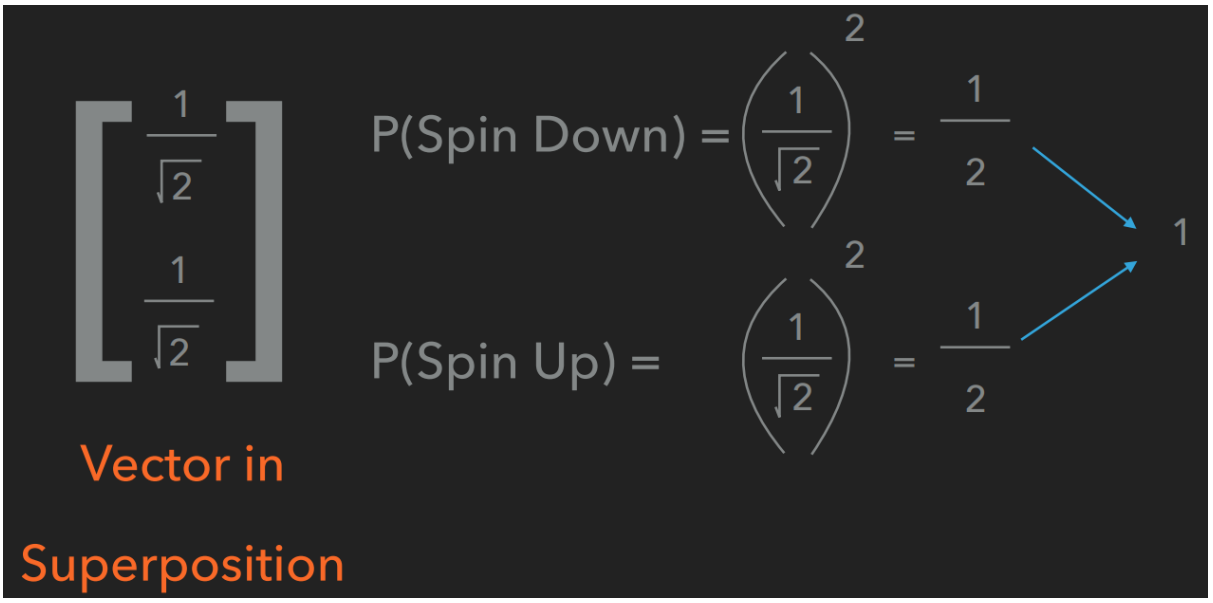
Qubits will be displayed in a matrix - as seen in previous lessons. Either it is spin up, or spin down, and is represented in a column matrix to perform operations (such as superposition, or entanglement).



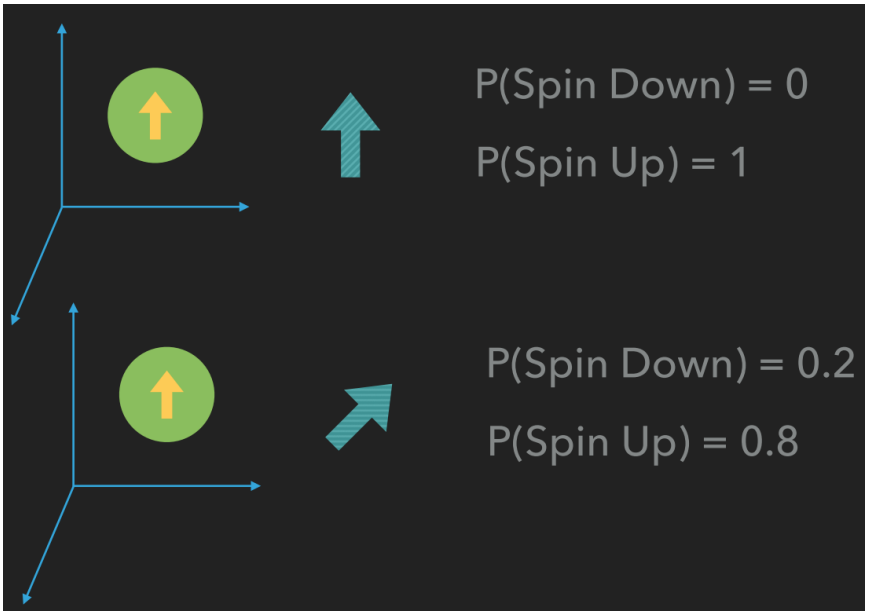
The top number represents the spin down state, while the lower number will represent the spin up state. Once we perform a calculation we will know where the "1" is represented. This will create a 100% probability of being in this state.



At this point we refer again to the superposition - as it doesn't have to be in the spin up or spin down position. 50% chance of getting zero or one - or anything in between as well.

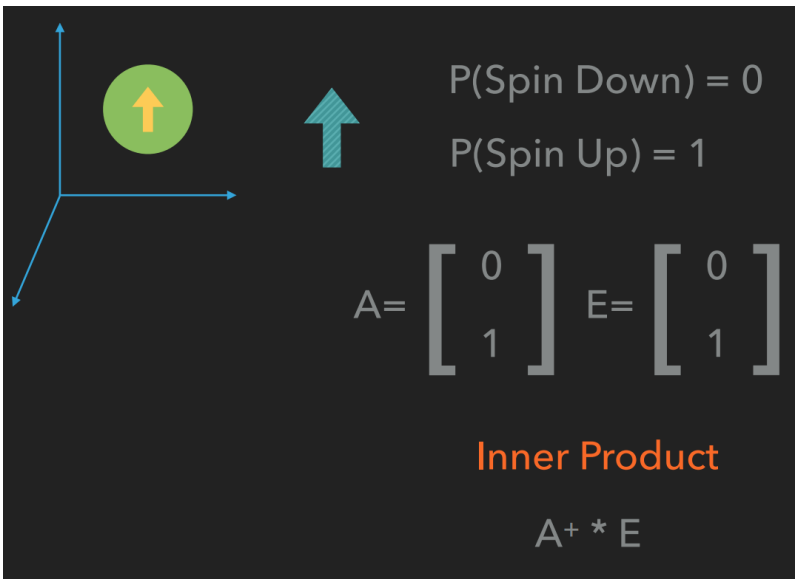


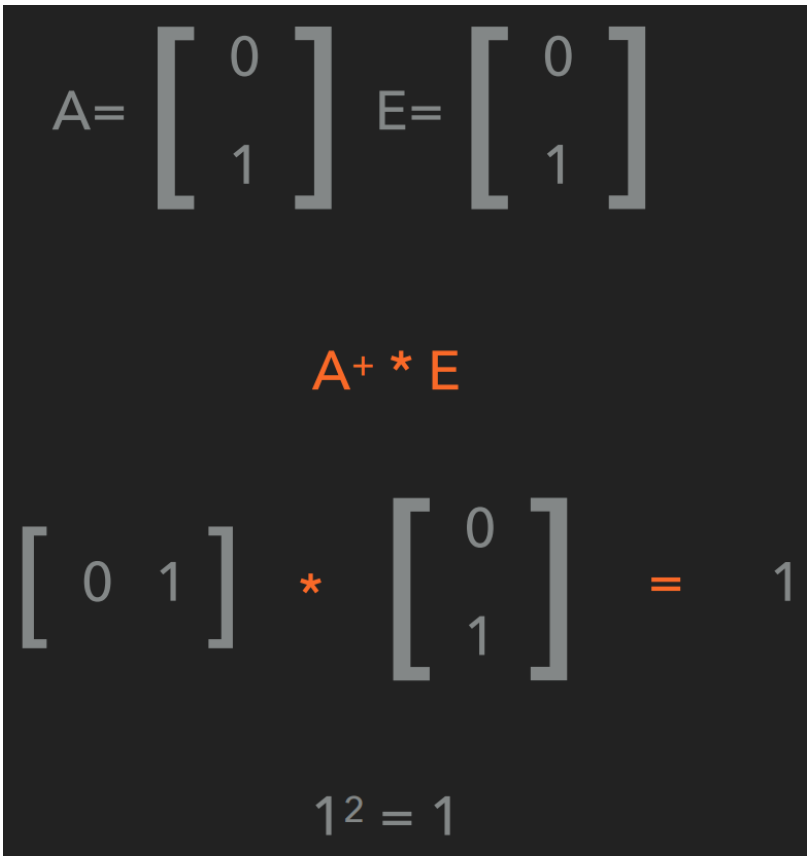
If we are to determine the spin - we need to calculate this. The state of the electron is determined:



If both align perfectly - they will be 100% in one state. If we reduce the probability in one state - it will shift the probability. Chances are now higher the spin position will be in the other state - a probability of it being in that state at least.

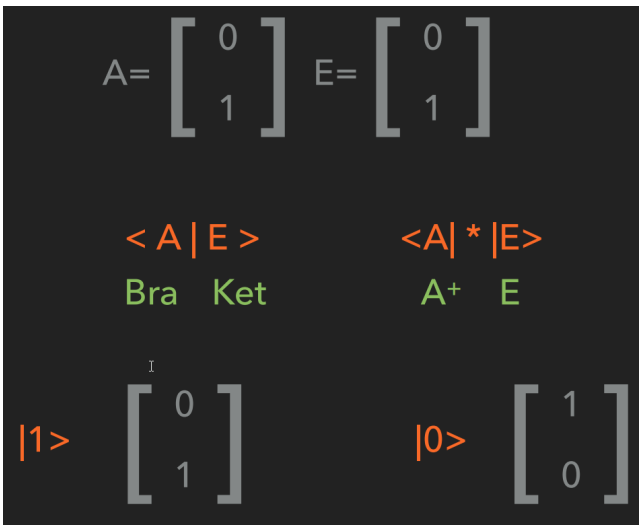
If we want to calculate this position - we need to take the adjoint of A (remember: + sign), and multiply it to the electron. This is called the **inner product**.



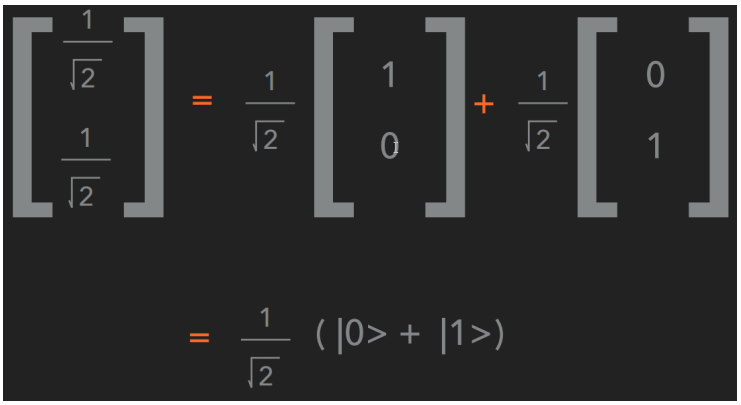


Bra-ket

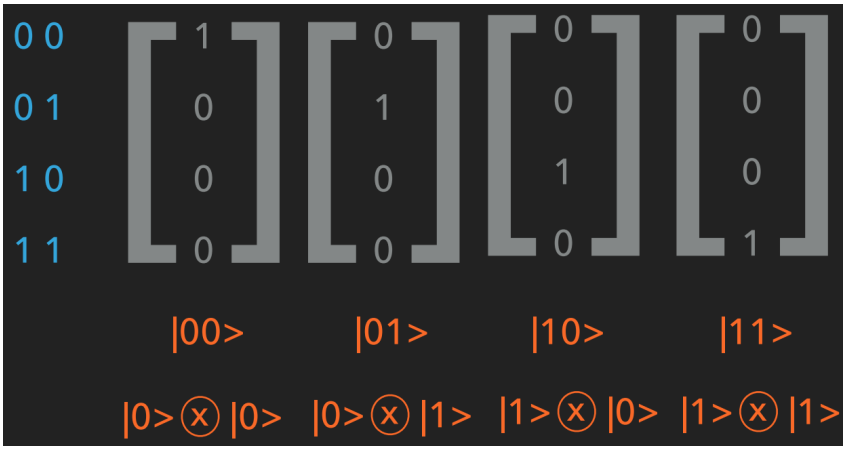
This annotation will simplify the mathematics behind this calculation. We are going to use the **inner product** a lot - Bra Ket notation simplifies this operation. The **angular signs < …. >** refers to the first bracket being the Bra, the second one being the Ket. If you see either only one of both this will represent a clear vector. Ket 0 (|1>) means 1 0, Ket 1 (|0>) means 0 1. We don’t perform mathematics - we just simplify the notation.



Returning back to the superposition matrix - we can simplify the matrices as following:



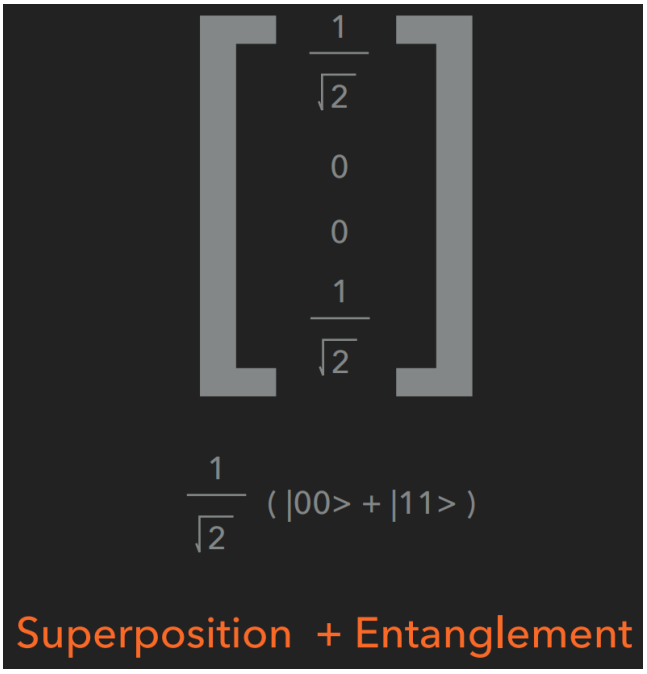
You can clearly see the simple Ket annotation at the bottom, instead of having to annotate the probability with matrices. It simplifies the annotation! Now we can also start looking at **multiple qubits** - and determine their states in Kat notation opposed to creating matrices all the time. We will use the **tensor product** here to create matrices. Remember that each Ket state is one single matrix - with tensor product you can combine both into one matrix!



Once we take more advanced formulas - we can see how superposition and entanglement works in practice for two qubits. In the below graph you have four results - as we have two qubits:

- P(00) = ½.
- P(01) = 0.
- P(10) = 0.
- P(11) = ½.

Now we truly see what this superposition and entanglement mean. As the below qubits have a 50% probability of either being in P(00) state, or P(11) state - they both have a 50% probability.

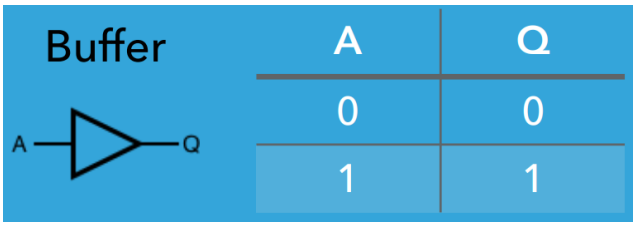


Qiskit

Classical gates

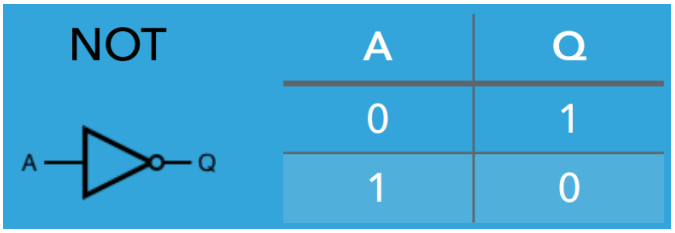
When writing code it will eventually be converted to 0's and 1's. This needs to be translated for transistors in order to obtain X amount of voltage. Classical gates can manipulate these 0's and 1's. We refer back to the **truth table.** This is for regular computing - not quantum computing, as this will be a lot more difficult. A is input, Q is output.

- **Buffer:** basic gate - it doesn't do anything. 0 for input, 0 for output. 1 for input, 1 for output.



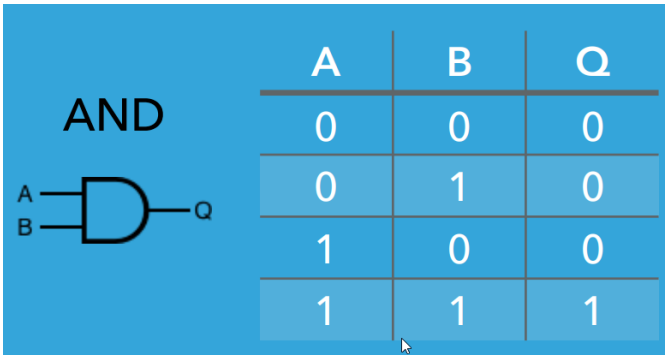
A	Q
0	0
1	1

- **NOT:** reverses the value - 1 input, 0 output!



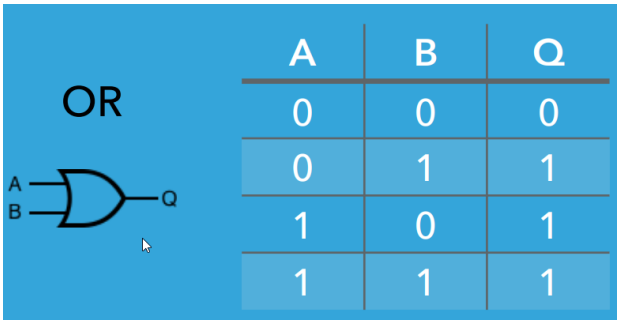
A	Q
0	1
1	0

- **AND:** we now have two values: A and B. In order to have a 1 as output, you will need to have both A and B as 1 state.



A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

- **OR:** if either value is 1 - the output will be 1. If both are 1 - you will get 1.



A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

- **NAND: NOT** and - being the opposite of the AND gate.

NAND

A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

- **NOR: NOT** OR - reverse of OR!

NOR

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

- **XOR: EXCLUSIVE** OR - different inputs will result in a 1.

XOR

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

- **XNOR: EXCLUSIVE NOT** OR - same inputs will result to 1, or 0. Both inputs need to be the same!

XNOR

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

In quantum computing these concepts are similar - but on a whole different level. The classical gates will be different when dealing with quantum gates.

Quantum gates

- **X(NOT):** reverses 1 and 0 to 1 and 0. It is now a matrix!
- **X | 1 >:** reverses the Ket 1 state.
- **X | 0 >:** reverses the Ket 0 state.

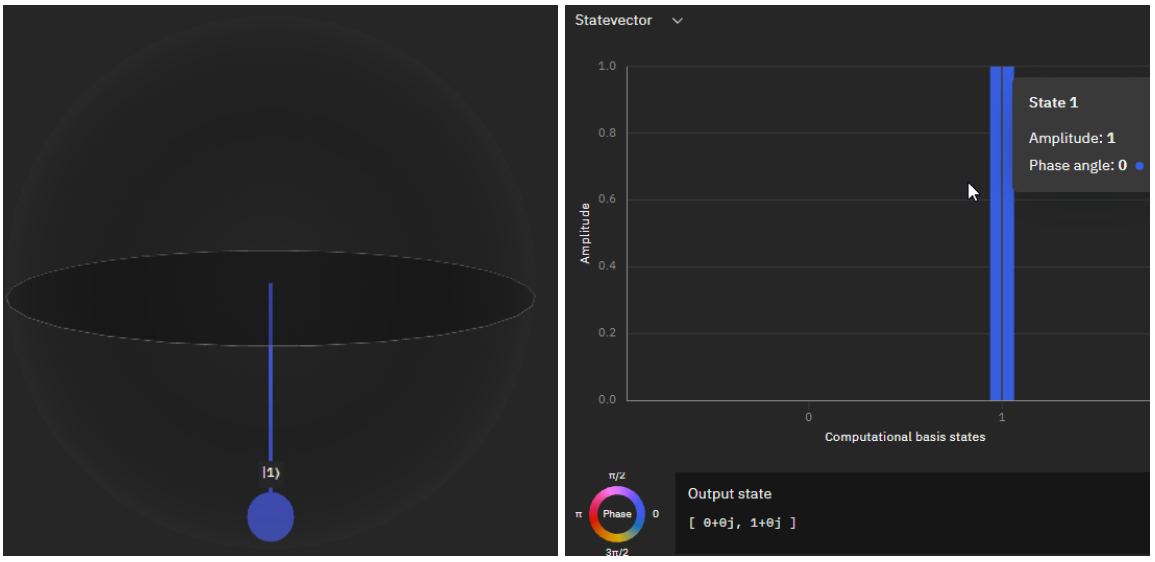
$$X\ (NOT) \quad \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$X\,|\,1\,\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle$$

$$X\,|\,0\,\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

Now to go IBMs Quantum Compose to practise these gates.

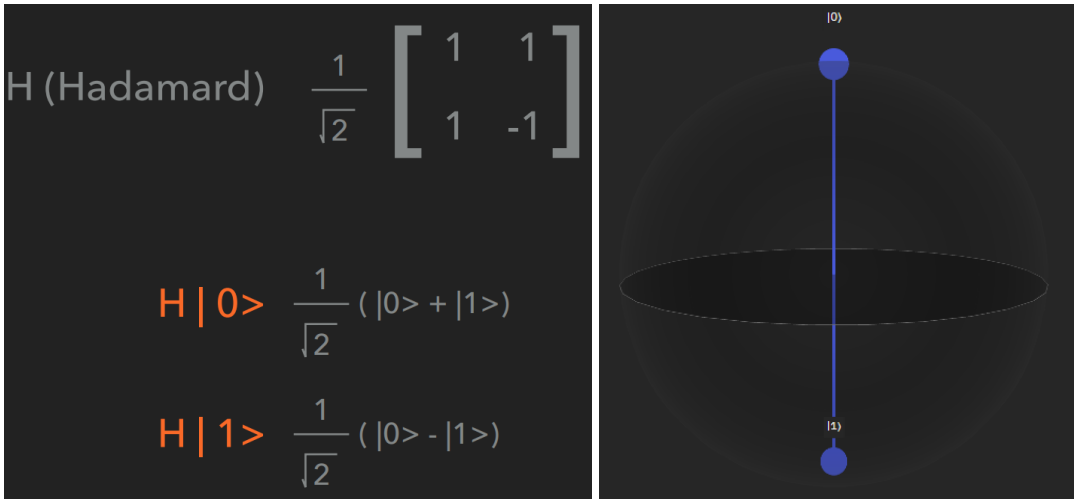
- **Q-Sphere:** provides a sphere with the Ket state.
- **Statevector:** provides the matrices used.



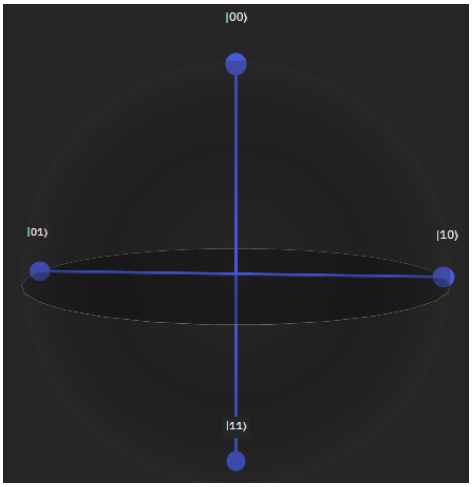
The idea you need to obtain is that the end result is provided in matrices.

- **Pauli gates:** names after the physicist who came up with these gates.
    - Y (Pauli Y)
    - Z (Pauli Z)

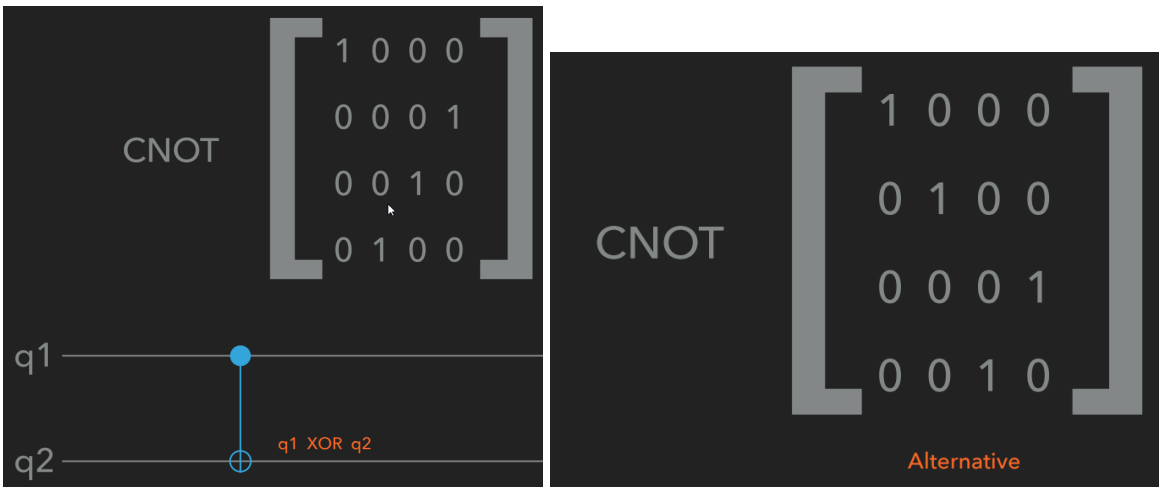
- **H (Hadamard): most important gate of all!** Hadamard gates will make sure we will enter superposition. Now both Ket 1 and Ket 0 have a 50% probability - superposition! With multiple qubits this will get a little crazy.



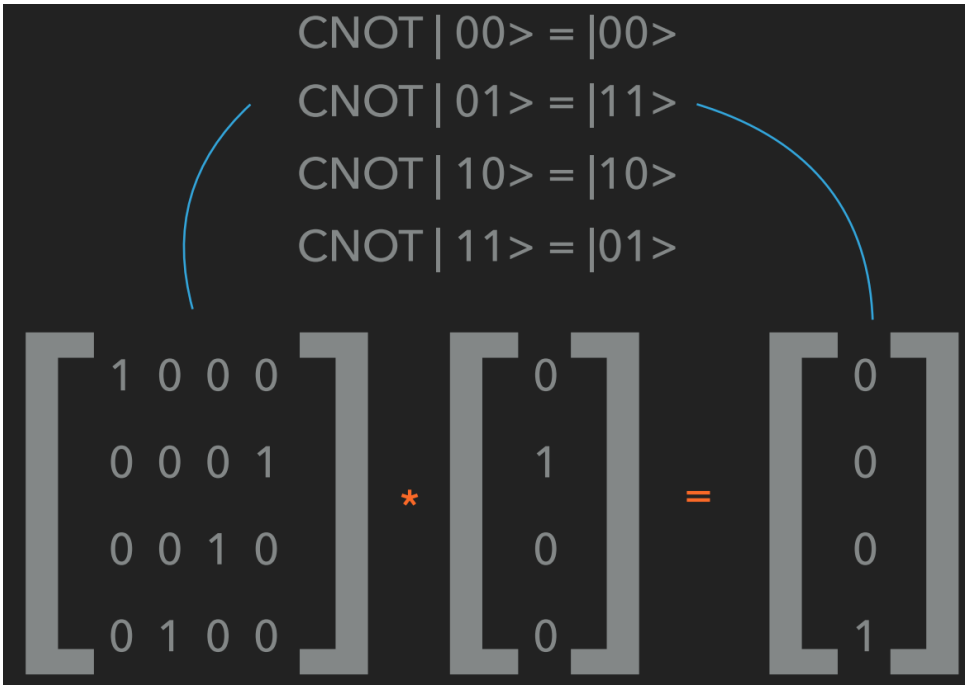
When applying two qubits, both with a Hadamard gate, probabilities will be 25% for each state! Now this is what the superposition means - it can be either any state (with probability).

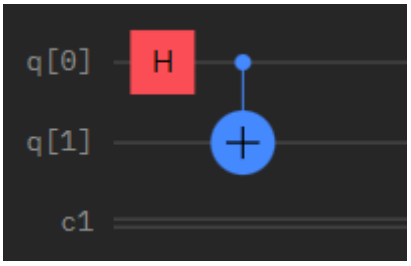


- **CNOT:** similar to NOT gate - but it is a **controlled** NOT gate. *In other books this can be a different matrix. In Qiskit the below matrix will always be the same (alternative matrix = different calculation).* It applies the XOR logic.

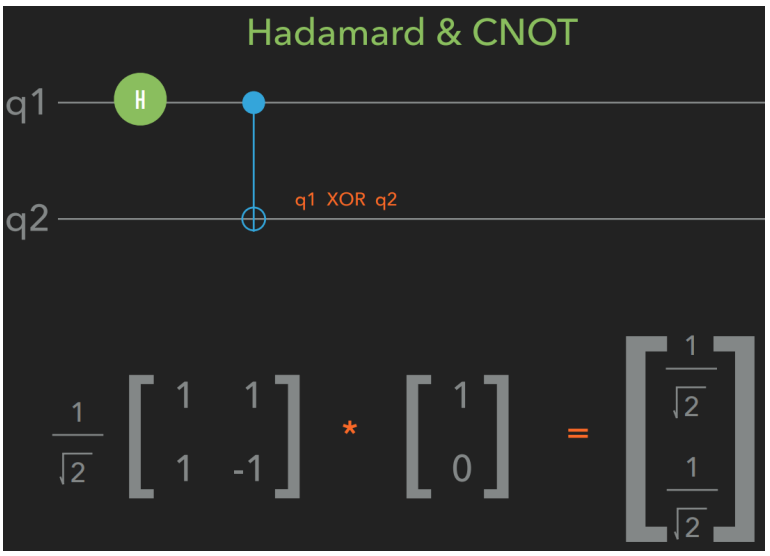


You need to apply the CNOT gate to **at least 2 qubits**. Below are the maths for this gate.

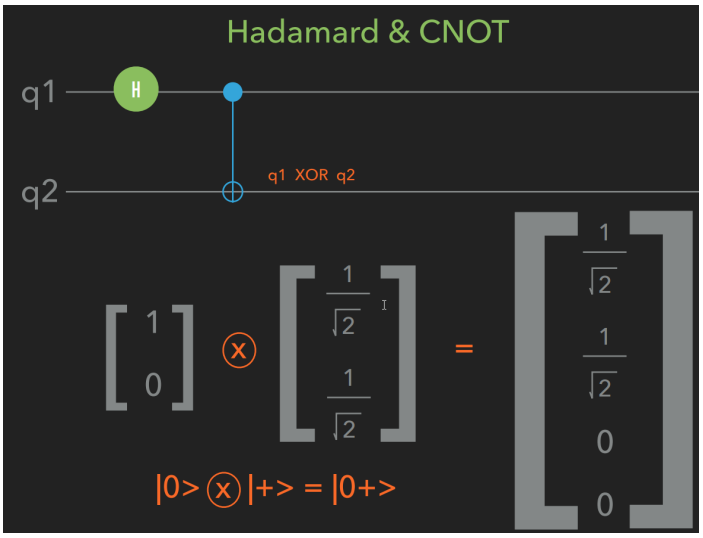




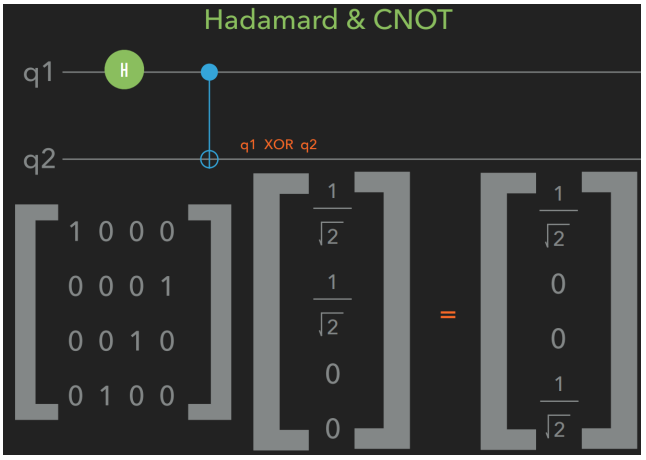
This will create a **superposition** gain + entanglement!



|+>: Ket + means **the qubit is in the superposition**.



Now apply the CNOT gate to the Hadamard gate - and calculate for superposition + entanglement! Hadamard will put a qubit in a superposition - CNOT will entangle them.



Qiskit

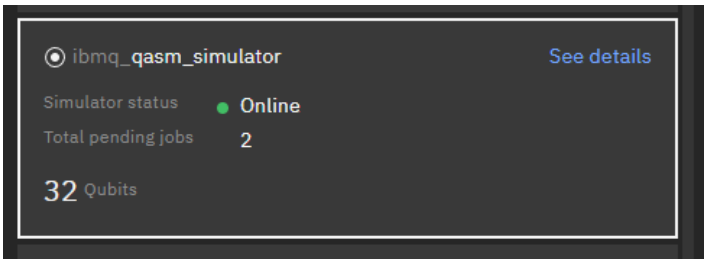
Framework for python enables us to interact with IBM quantum computers to create circuits, program them and run code on these machines. A variety of modules are available.

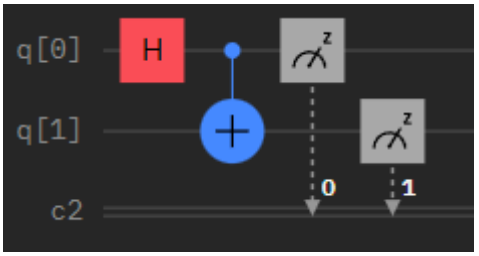
GitHub: https://github.com/Qiskit/qiskit

- Qiskit Terra: open-source SDK for working with open source quantum computers. This is the main module we'll be working on.
- Qiskit Simulator: Aer.
- Qiskit Experiments: Ignis.
- Qiskit Application Modules: Aqua.
- Qiskit IBM Quantum: Provider.

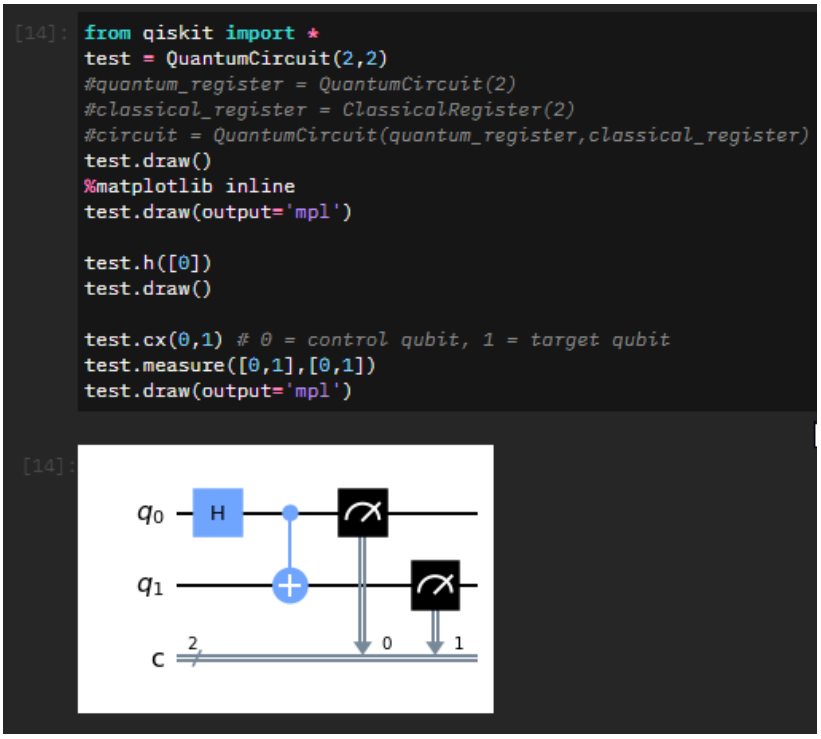
They serve the purpose of creating quantum software, or quantum technologies. Use the IBM Jupyer notebooks and composer to work with the exercises.

Choose the **ibmq_qasm_simulator:** select the amount of shots that need to be calculated. And add **measurements** to your quantum composer.





Above you see the visual representation - below you see the coded part! H will add hadamard gates, cx will add the XNOT gate, measure will add measurements!

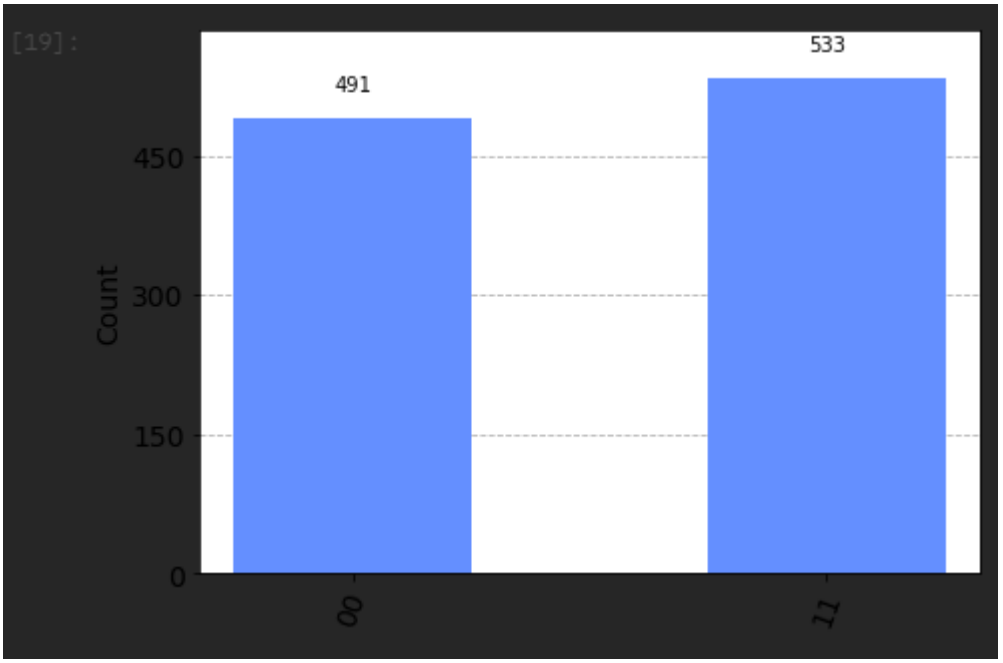


Now we can also use the IBM quantum machines to get the results - as you can see at the bottom, the result will now return our results… but we want to make it more clear (not an entire list).

```
[17]: from qiskit import *
      test = QuantumCircuit(2,2)
      #quantum_register = QuantumCircuit(2)
      #classical_register = ClassicalRegister(2)
      #circuit = QuantumCircuit(quantum_register,classical_register)
      #test.draw()
      %matplotlib inline
      #test.draw(output='mpl')

      test.h([0])
      #test.draw()

      test.cx(0,1) # 0 = control qubit, 1 = target qubit
      test.measure([0,1],[0,1])
      test.draw(output='mpl')

      simulator = Aer.get_backend('qasm_simulator')

      result = execute(test,backend=simulator).result()

      print(result)

      Result(backend_name='qasm_simulator', backend_version='0.11.1', qobj_id='0b09c838-c345-4b98-8b07-b919923e2330', job_id='ce2345a5-8e7d-4fde-8a1e-c82d48e98d3d',
      success=True, results=[ExperimentResult(shots=1024, success=True, meas_level=2, data=ExperimentResultData(counts={'0x3': 514, '0x0': 510}), header=QobjExperim
      entHeader(clbit_labels=[['c', 0], ['c', 1]], creg_sizes=[['c', 2]], global_phase=0.0, memory_slots=2, metadata={}, n_qubits=2, name='circuit-107', qreg_sizes=
      [['q', 2]], qubit_labels=[['q', 0], ['q', 1]]), status=DONE, seed_simulator=3769862488, metadata={'noise': 'ideal', 'batched_shots_optimization': False, 'meas
      ure_sampling': True, 'parallel_shots': 1, 'remapped_qubits': False, 'active_input_qubits': [0, 1], 'num_clbits': 2, 'parallel_state_update': 8, 'sample_measur
      e_time': 0.001199623, 'num_qubits': 2, 'device': 'CPU', 'input_qubit_map': [[1, 1], [0, 0]], 'method': 'stabilizer', 'fusion': {'enabled': False}}, time_taken
      =0.005112522)], date=2023-02-05T14:26:48.942136, status=COMPLETED, header=QobjHeader(backend_name='qasm_simulator', backend_version='0.11.1'), metadata={'time
      _taken': 0.005851848, 'time_taken_execute': 0.005272831, 'mpi_rank': 0, 'num_mpi_processes': 1, 'max_gpu_memory_mb': 0, 'max_memory_mb': 31960, 'parallel_expe
      riments': 1, 'time_taken_load_qobj': 0.000552418, 'num_processes_per_experiments': 1, 'omp_enabled': True}, time_taken=0.006201505661010742)
```

By adding the visualisation code you can obtain a very clear histogram:

```
from qiskit.visualization import plot_histogram

plot_histogram(result.get_counts(test))
```



Getting real quantum computer properties

For this you will need the IBM API key.
You can create a file and put the API in there - now you need to select the proper machine.

```
from qiskit import *
IBMQ.save_account(open("ibmapi.txt","r").read())
IBMQ.load_account()

Aer.backends()

provider = IBMQ.get_provider("ibm-q")

provider.backends()
```

```
configrc.store_credentials:WARNING:2023-02-05 14:35:39,806: Credentials already present. Set overwrite=True to overwrite.
ibmqfactory.load_account:WARNING:2023-02-05 14:35:44,899: Credentials are already in use. The existing account in the session will be replaced.

[<IBMQSimulator('ibmq_qasm_simulator') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_lima') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_belem') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_quito') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQSimulator('simulator_statevector') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQSimulator('simulator_mps') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQSimulator('simulator_extended_stabilizer') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQSimulator('simulator_stabilizer') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_manila') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibm_nairobi') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibm_oslo') from IBMQ(hub='ibm-q', group='open', project='main')>]
```

We can further code everything here to include everything we can see in the visual representation provided by IBM.

```
from qiskit import *
IBMQ.save_account(open("ibmapi.txt","r").read())
IBMQ.load_account()

Aer.backends()

provider = IBMQ.get_provider("ibm-q")

providers = provider.backends()

for backend in providers:
    try:
        qubit_count = len(backend.properties().qubits)
    except:
        qubit_count = "simulated"
    print(f"{backend.name()} : {backend.status().pending_jobs} & {qubit_count} qubits")
```

```
ibmq_qasm_simulator : 3 & simulated qubits
ibmq_lima : 4 & 5 qubits
ibmq_belem : 98 & 5 qubits
ibmq_quito : 14 & 5 qubits
simulator_statevector : 3 & simulated qubits
simulator_mps : 3 & simulated qubits
simulator_extended_stabilizer : 3 & simulated qubits
simulator_stabilizer : 3 & simulated qubits
ibmq_manila : 28 & 5 qubits
ibm_nairobi : 15 & 7 qubits
ibm_oslo : 23 & 7 qubits
```

Now you can see every single detail: the name, the pending jobs and the qubit count!

Running on a real quantum computer

Since we know which quantum computer we will be using - we will execute code on the selected quantum computer! Choose one with lesser jobs.

We need to add new code in order to monitor and watch the job we are running - as we cannot see what our position is in the queue!

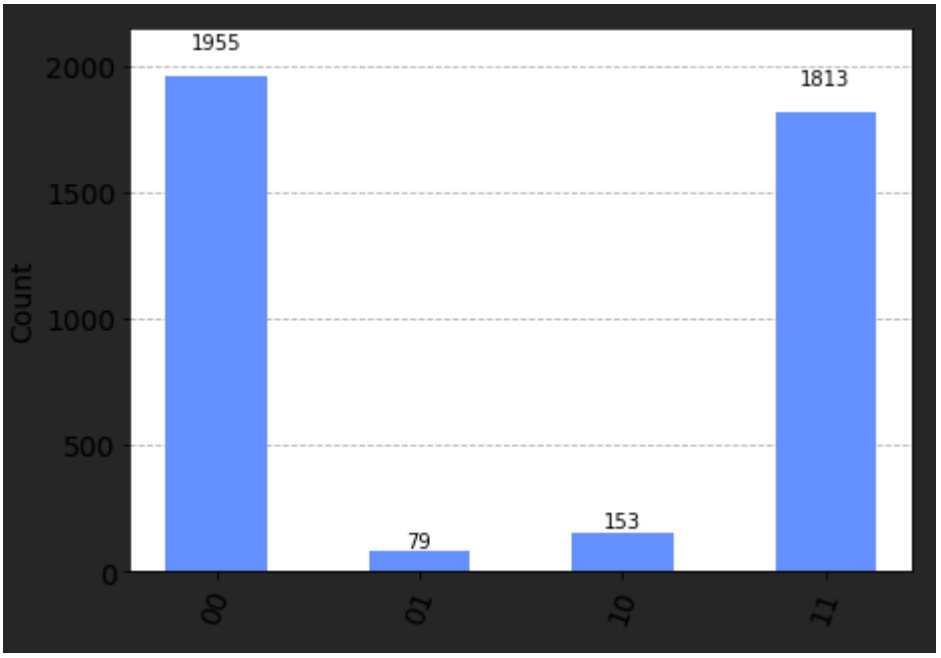
```
import qiskit.tools.jupyter
%qiskit_job_watcher
from qiskit.tools.monitor import job_monitor

IBMQ.load_account()

provider = IBMQ.get_provider(hub="ibm-q")
qcomputer = provider.get_backend('ibmq_lima')

job = execute(test,backend=qcomputer)

job_monitor(job)
```

```
Job Status: job has successfully run
```

Now you will get a handy dynamic parameter job_monitor that will display your position in the queue and yet you know once the result is completed.

Now we still want the end results by adding the following code:

```
quantum_result = job.result()
plot_histogram(quantum_result.get_counts(test))
```



Why do they end up in 01 and 10? This is called **Quantum Noise** (heat, air,...) and a quantum computer is subject to these factors. It will give back some “faulty” results, but are not perfect. Once using real computers, these “errors” will persist (for now).

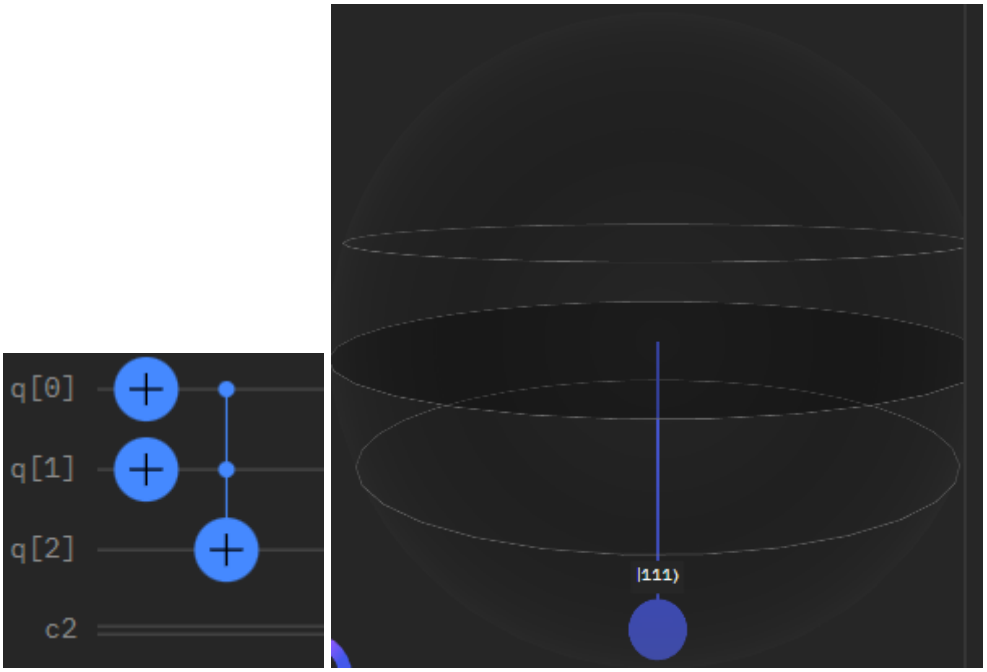
Toffoli

This is a **three-qubit gate** with two controls and one target. It performs an X on the target only if both controls are in the state |1> (Ket 1).

There are two types of qubits: a **control** qubit, and a **target** qubit.

- **Control:** if these qubits are in a certain state, the target qubit will perform an action.
- **Target:** if the control qubit(s) are in a certain state, the target qubit will perform an action. This action is the equivalent of putting the qubit in a certain state (up or down).

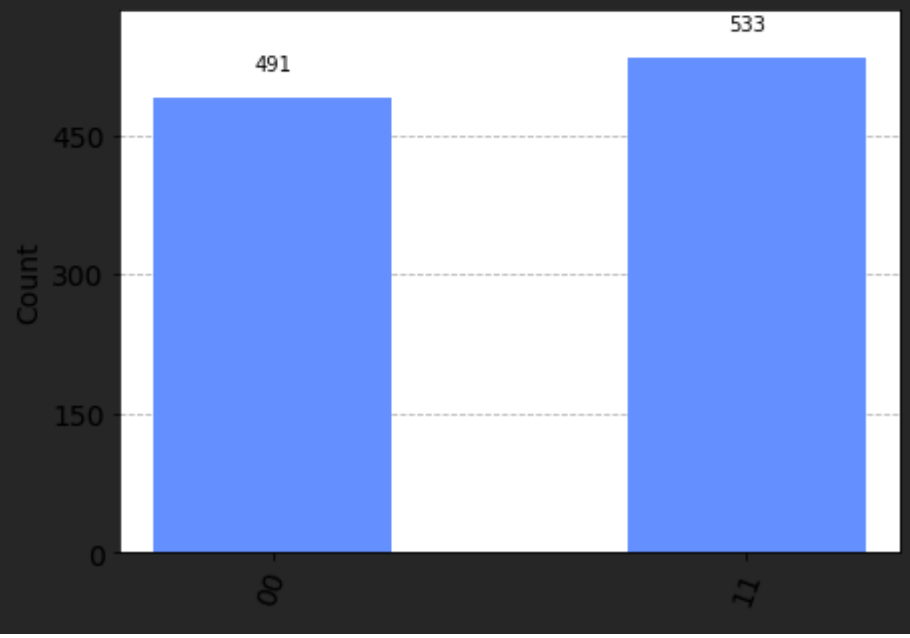
This is called a controlled-controlled-not gate (or CCX gate). Now we have 2 control qubits instead of only 1. If both qubits are in **the same state**, it will execute the **target qubit**. If not - it will not perform a clear state in the target gate.



[19]:
```
from qiskit import *
test = QuantumCircuit(2,2)
#quantum_register = QuantumCircuit(2)
#classical_register = ClassicalRegister(2)
#circuit = QuantumCircuit(quantum_register,classical_register)
#test.draw()
%matplotlib inline
#test.draw(output='mpl')

test.h([0])
#test.draw()

test.cx(0,1) # 0 = control qubit, 1 = target qubit
test.measure([0,1],[0,1])
test.draw(output='mpl')

simulator = Aer.get_backend('qasm_simulator')

result = execute(test,backend=simulator).result()

print(result)

from qiskit.visualization import plot_histogram

plot_histogram(result.get_counts(test))
```

```
Result(backend_name='qasm_simulator', backend_version='0.11.1',
1', success=True, results=[ExperimentResult(shots=1024, success
perimentHeader(clbit_labels=[['c', 0], ['c', 1]], creg_sizes=[[
sizes=[['q', 2]], qubit_labels=[['q', 0], ['q', 1]]), status=DO
e, 'measure_sampling': True, 'parallel_shots': 1, 'remapped_qub
le_measure_time': 0.001902352, 'num_qubits': 2, 'device': 'CPU'
time_taken=0.005827458)], date=2023-02-05T14:28:34.723519, stat
data={'time_taken': 0.00643045, 'time_taken_execute': 0.0058922
rallel_experiments': 1, 'time_taken_load_qobj': 0.000518102, 'n
```

[19]:



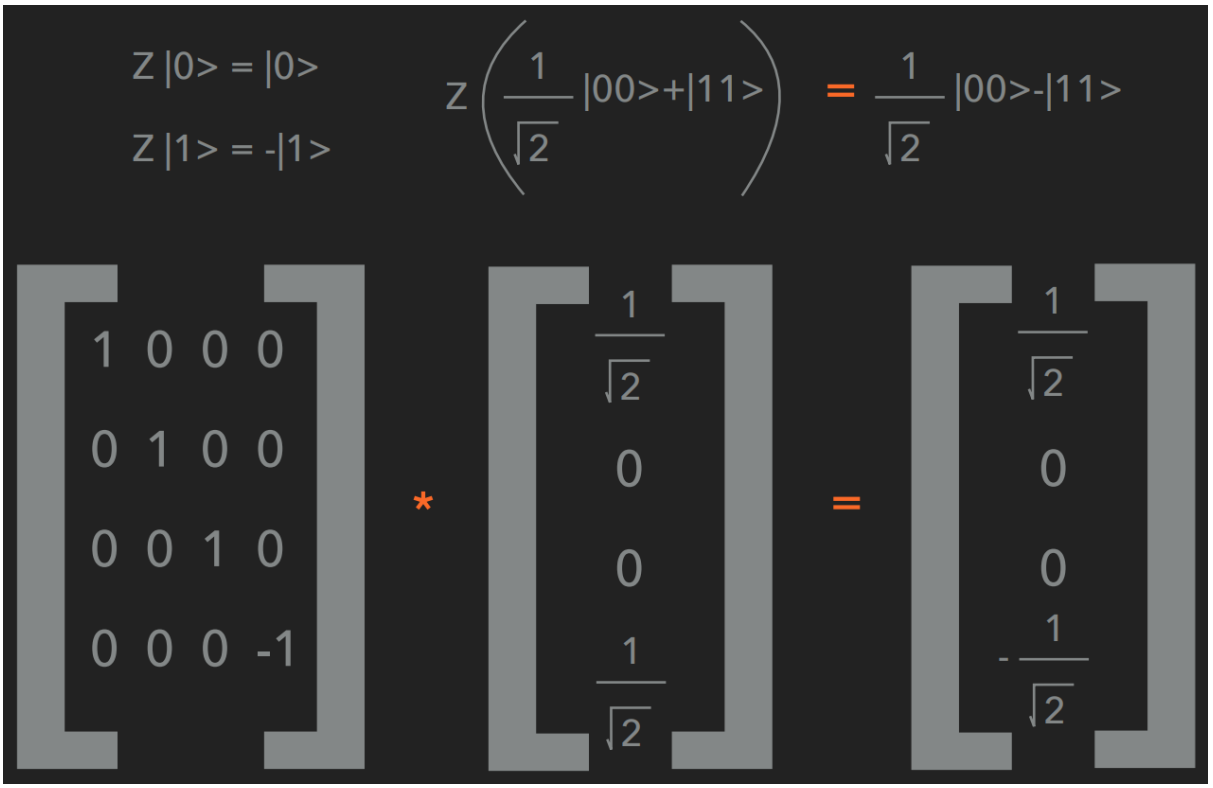
Teleportation (quantum)

We'll look into quantum algorithms and reimburse what we have learned so far.

Phase

Zgate is a new kind of gate - kind of an identity matrix with a minus. This gate is the equivalent of a phase flip. It turns Ket 0 into Ket 0, but it turns Ket 1 into a minus Ket 1 state.

The possibilities will not change - it just turns one Ket 1 into a minus Ket 1.



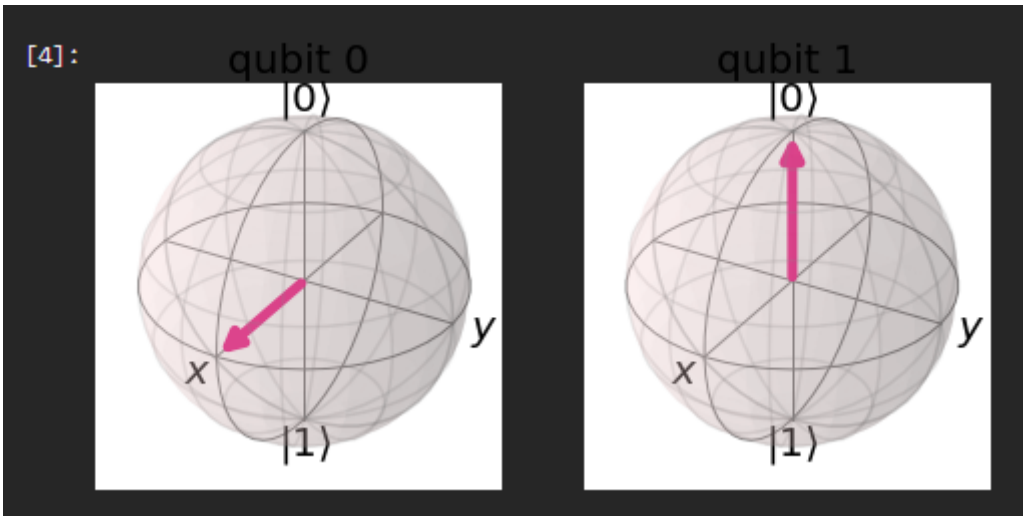
The phase circle is allocated in the IBM quantum composer - which we haven't used yet. Once we apply the Hadamard and Z gates - we have a phase, Minus Ket 1.

It doesn't affect the outcome - but in a complicated circuit it will affect the outcome (red Ket). In the Bloch sphere it will rotate the Ket sphere into a different direction.

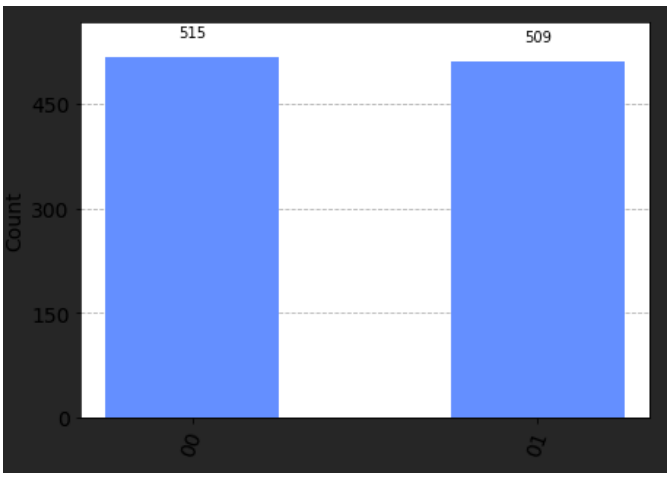
Phase and bloch sphere

The bloch sphere will return a 3-D vector, opposed to the regular sphere.

```
from qiskit import *
from qiskit.tools.visualization import plot_bloch_multivector
from qiskit.visualization import plot_histogram
%matplotlib inline
import math

Aer.backends()
#We now have tow differnt simulators
qasm_simulator = Aer.get_backend('qasm_simulator')
statevector_simulator = Aer.get_backend('statevector_simulator')

def run_on_simulators(circuit):
    statevec_job = execute(circuit, backend=statevector_simulator)
    result = statevec_job.result()
    statevec = result.get_statevector()

    num_qubits = circuit.num_qubits
    circuit.measure([i for i in range(num_qubits)], [i for i in range(num_qubits)])

    qasm_job = execute(circuit, backend=qasm_simulator, shots=1024).result()
    counts = qasm_job.get_counts()

    return statevec, counts


circuit = QuantumCircuit(2,2)
statevec, counts = run_on_simulators(circuit)
plot_bloch_multivector(statevec)

circuit.h(0)
statevec, counts = run_on_simulators(circuit)
plot_bloch_multivector(statevec)
```

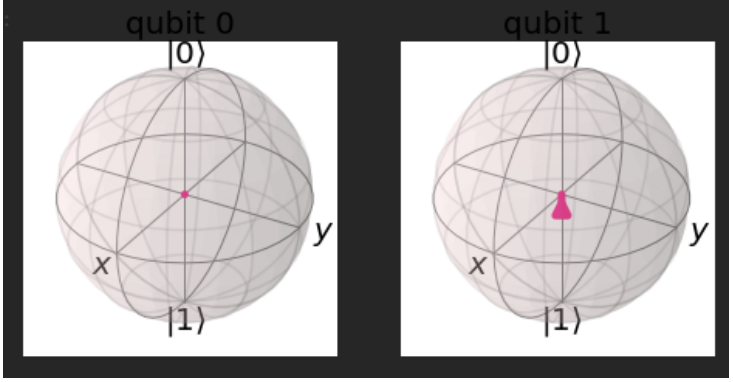


Since we added the Hadamar gate - it will be in a superposition. The qubit is either in the 0 state or in the 1 state (probability is ~50%).



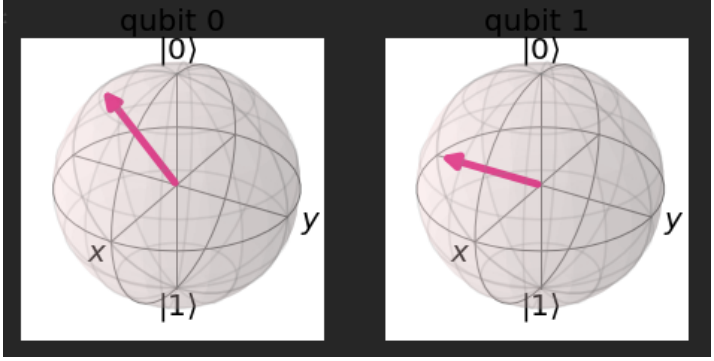
Adding another CNOT / XNOT gate - we get a very “weird” result:

```
circuit = QuantumCircuit(2,2)
circuit.h(0)
circuit.cx(0,1)
statevec, counts = run_on_simulators(circuit)
plot_bloch_multivector(statevec)
```



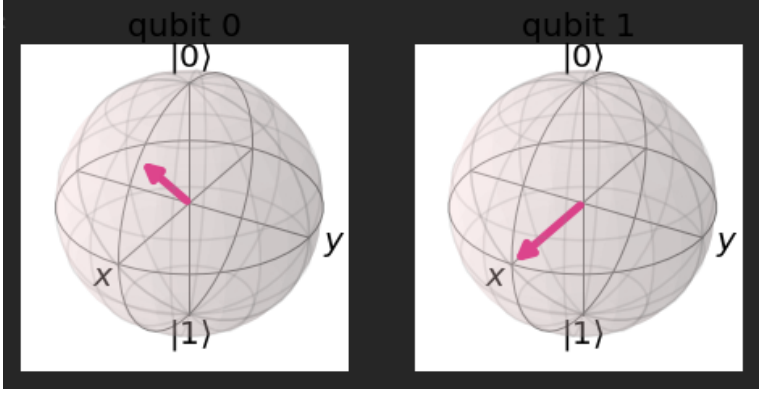
Now we will look into the ZGate for custom rotations. For each circuit, a custom gate:

```
circuit = QuantumCircuit(2,2)
circuit.rx(math.pi/4, 0)
circuit.rx(math.pi/2, 1)
statevec, counts = run_on_simulators(circuit)
plot_bloch_multivector(statevec)
```



Now we can apply the same for the y parameter:

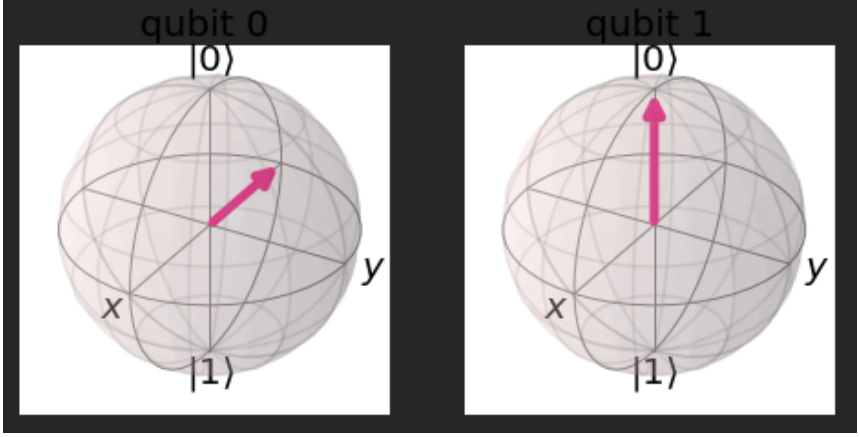
```
circuit = QuantumCircuit(2,2)
circuit.ry(math.pi/4, 0)
circuit.ry(math.pi/2, 1)
statevec, counts = run_on_simulators(circuit)
plot_bloch_multivector(statevec)
```



What does this mean: Hadamard will rotate this vector to a specific direction in a mathematical equation. If we apply Hadamard again - we will see that the second qubit (qubit 1 - picture on the right) is exactly the same.

Adding a ZGate to this equation would make a lot more sense - so we add a Hadamard gate AND a ZGate to the equation. Compared to a single Hadamard gate in the previous picture - we see the ZGate simply reversing the Hadamard gate for Qubit 0.

```
circuit = QuantumCircuit(2,2)
circuit.h(0)
circuit.z(0)
statevec, counts = run_on_simulators(circuit)
plot_bloch_multivector(statevec)
```



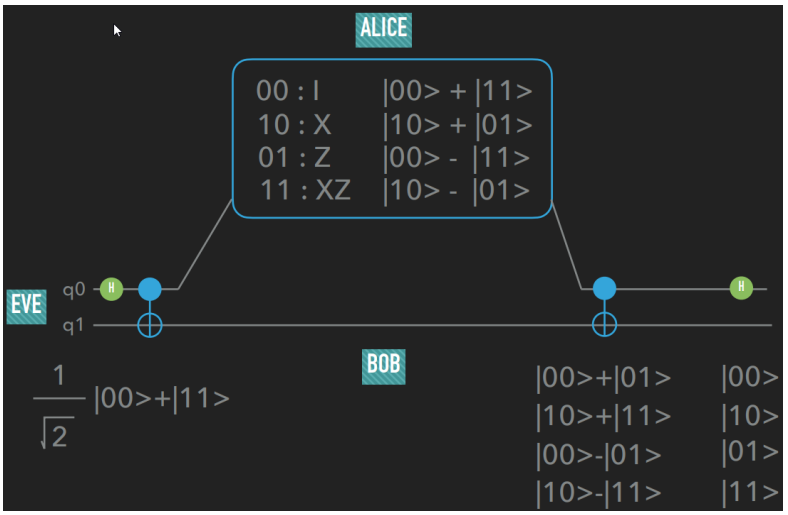
By using Rx, Ry rotations, we can provide a variety of rotations to the sphere. A qubit is very flexible in which we can provide further advanced calculations.
https://github.com/atilsamancioglu/QX03-PhaseAndBlochSphere

Superdense coding

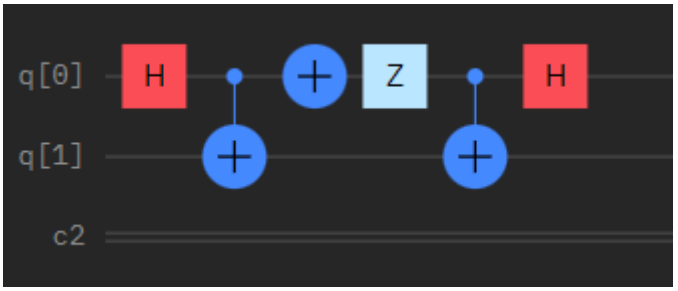
We're going into quantum algorithms and see quantum teleportation. Superdense coding means we can store two bits of numbers in one, single qubit. **But this statement is not correct…** We can put a qubit in a superposition and entangle them, but it will eventually collapse to one classical bit (0 or 1). **How is it possible to use only one qubit, and return 00, 01, 10 or 11?** We'll be using the method of entanglement. We can manipulate the outcome by utilising only one single qubit.

EVE = an "eavesdropper" trying to listen to the messages of Alice and Bob. Alice and Bob are simply fictional examples. One qubit is for Alice, the other one for Bob. Before Alice gives the qubit to Bob, Alice provides a Hadamard gate and a CNOT gate. It is in a superposition, and both qubits are entangled. Alice will eventually take one of the qubits - and apply a gate on this qubit. When she returns this qubit to Bob, she will manipulate the qubit. You can now apply any kind of date (NOT gate, ZGate,...).

How does it affect the outcome? If you apply an X Gate it will apply an effect. The eventual value will change eventually. The **phase** is represented by the "+" or "-", the result will depend on the **Ket** value. Both of them will change accordingly.



We can **now manipulate one qubit, and change the value of the other qubit** if they are entangled (due to the CNOT gate and Hadamard superposition). We can now effectively use different gates to affect the outcome - and use one qubit to store two bits of data if we want to.

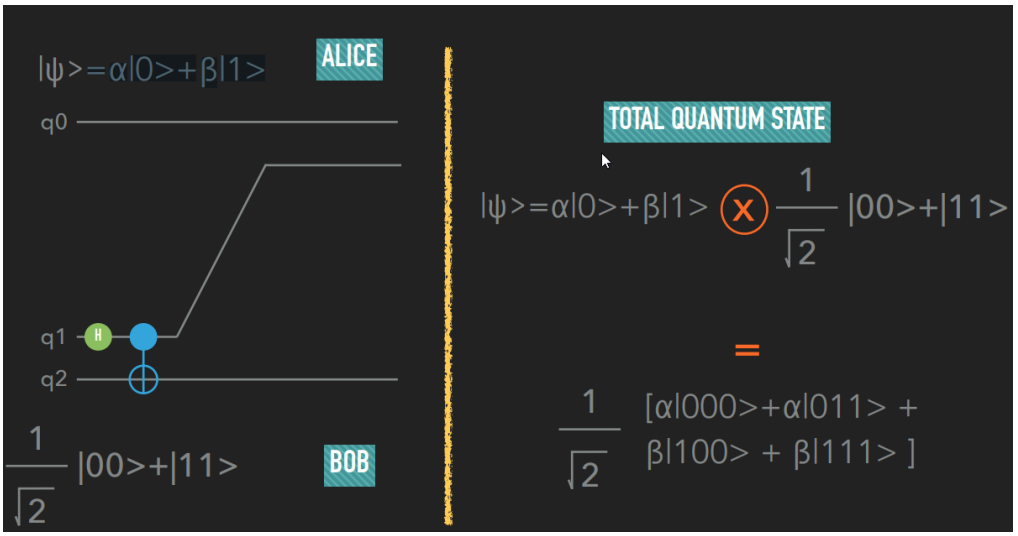


Quantum teleportation

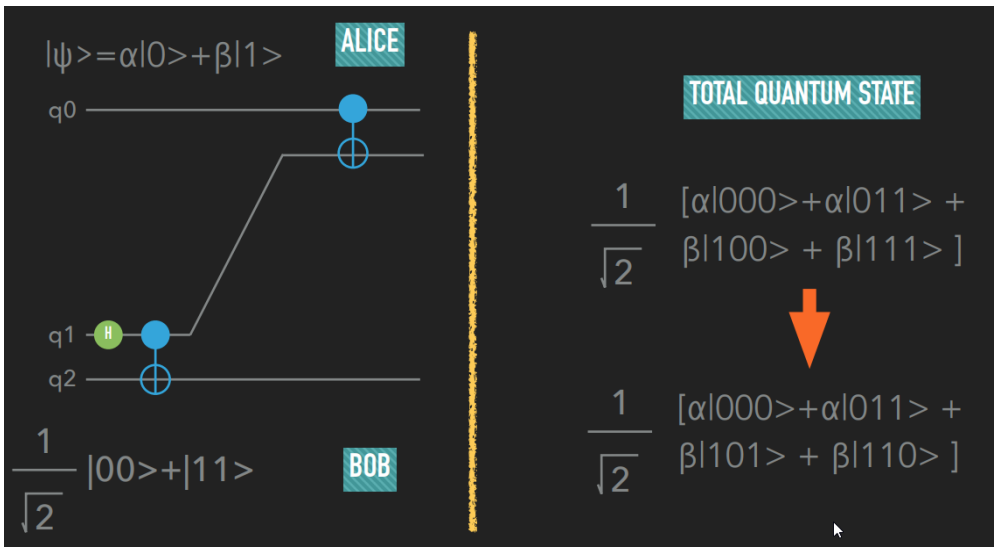
We are going to transform the state of one qubit to another qubit. This is called quantum teleportation. **Why does this matter?** You can clone classical bits in a simple way (since they can only be 1 or 0) but in **qubits it is not easy to do that**. You cannot clone quantum bit states and this is called **no clone theorem:** *the no-cloning theorem states that it is impossible to create an independent and identical copy of an arbitrary unknown quantum state.*

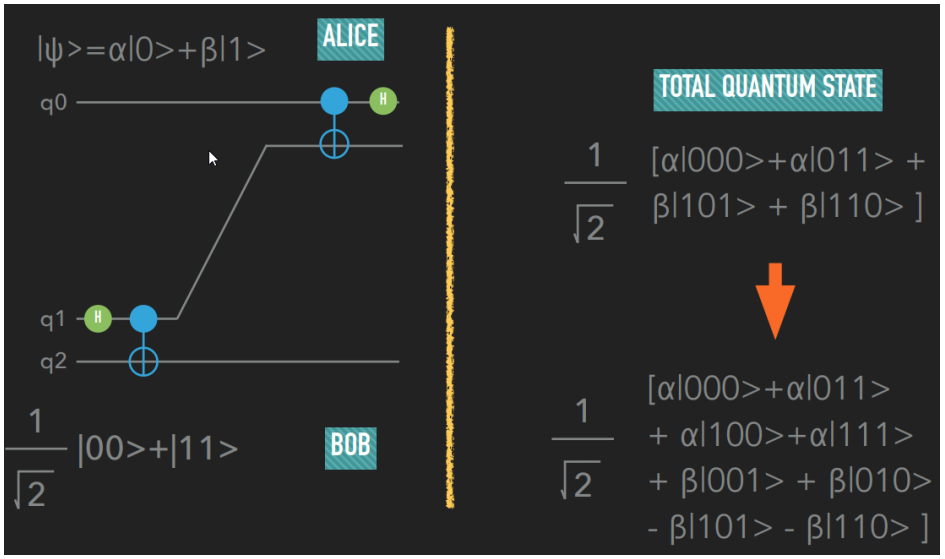
A greek letter is introduced to determine if we have an **unknown state:** $|\psi\rangle=\alpha|0\rangle+\beta|1\rangle$ This is called Ket “Sy”. We will try to transform this unknown state into Bob’s qubit state.

Q2 will be a helper qubit, also called an **Ancilla qubit**. We now have three qubits - entangles Q1 and Q2, and give Q2 to bob. Now we tensor product this unknown state to the known state.

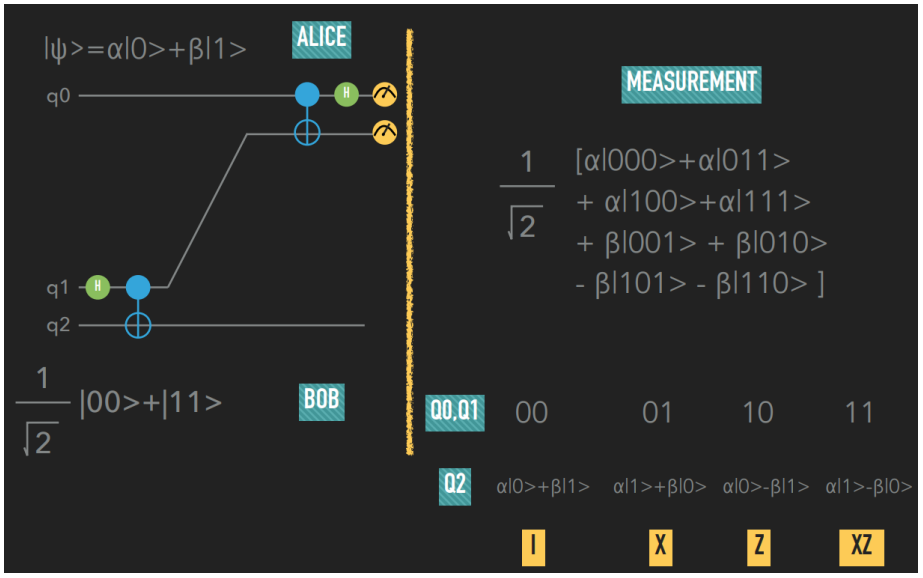


Later on Alice applies a CNOT gate to Q1 - and this will make sure the total state is transferred. Additionally - she will apply a Hadamard gate.

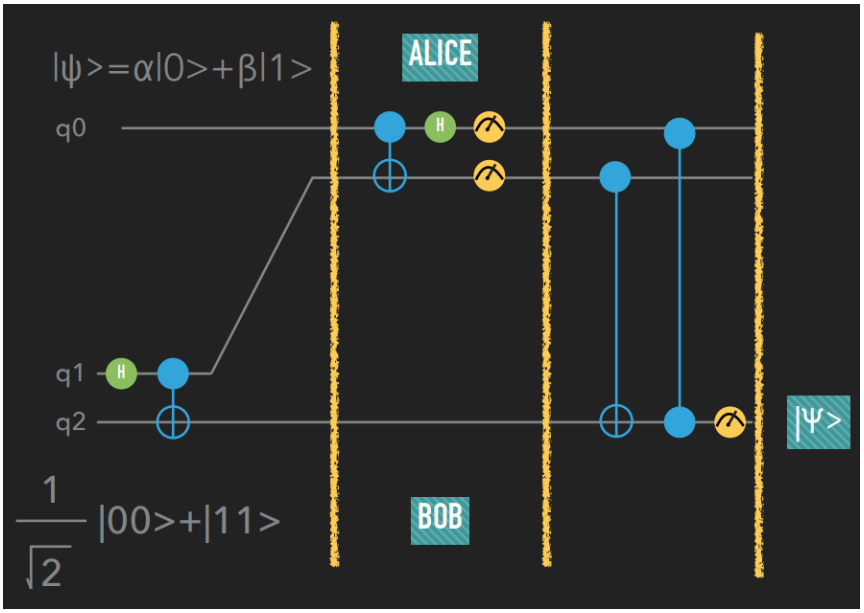




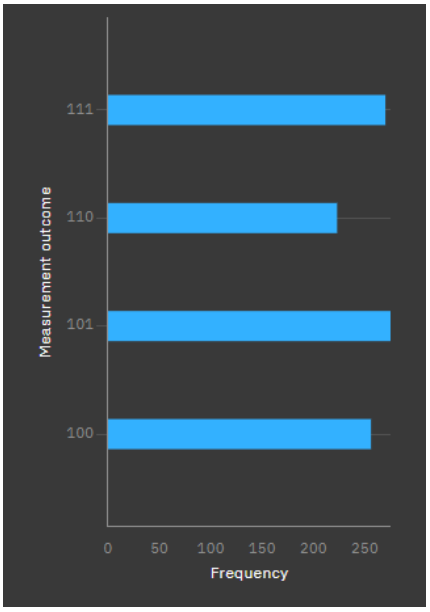
At the end she is going to **measure Q0 and Q1** to understand the values:

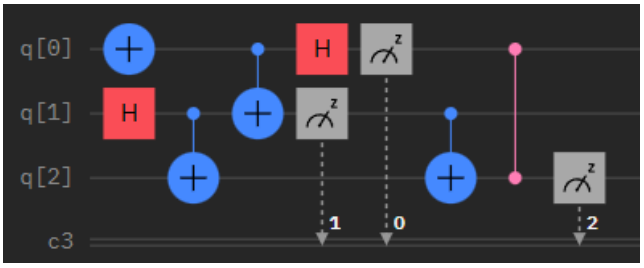


In order to transform the states to obtain a final result - Bob needs to transform the states with additional gates (I gate, X gate, Z gate, or XZ gate). This way Bob can transform the final state by looking at the values Alice has measured.



If we try this in the Quantum Composer we get the following results:





We can see that the second qubit (which is always ‘1’) is successfully transferred. We’ll look into Qiskit for an even clearer view of these states.

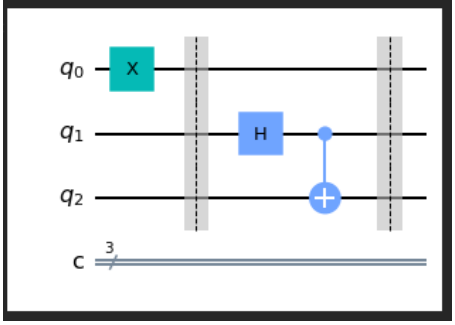
Teleportation in Qiskit

This is a gate to go towards the quantum algorithms - it does not solve a real-life problem (just yet). This part will have the same result as the last time - but coded into Qiskit.

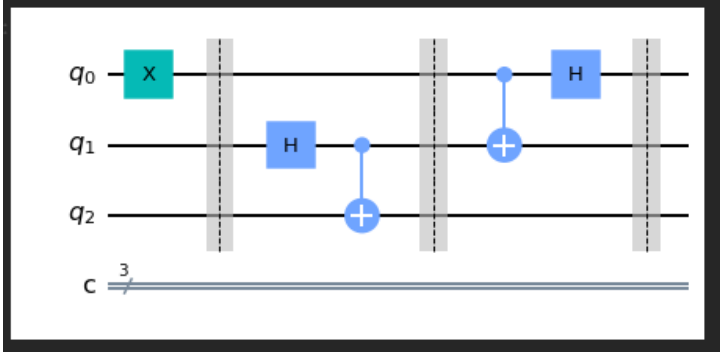
https://github.com/atilsamancioglu/QX04-QuantumTeleportation

```
from qiskit import *
from qiskit.visualization import plot_histogram
%matplotlib inline

circuit = QuantumCircuit(3,3)
```

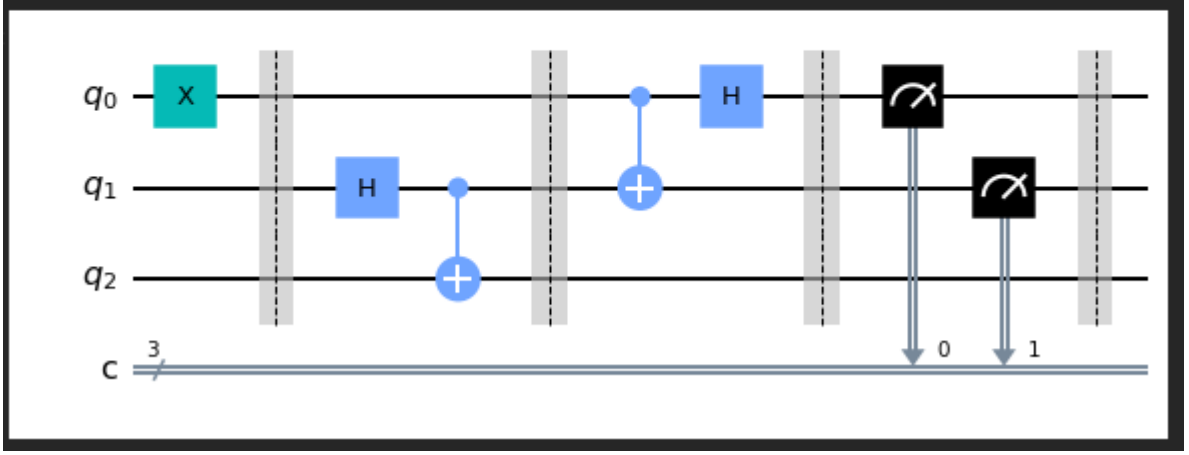
```
circuit.x(0)

circuit.barrier()

circuit.h(1)
circuit.cx(1,2)

circuit.barrier()

circuit.draw(output='mpl')
```



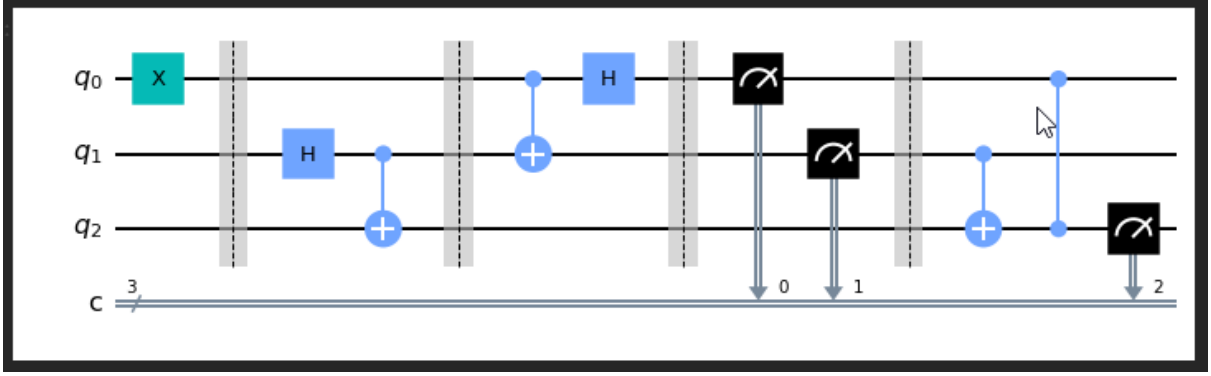

```
circuit.cx(0,1)
circuit.h(0)

circuit.barrier()
circuit.draw(output='mpl')
```




```
circuit.measure([0,1],[0,1])
circuit.barrier()

circuit.draw(output='mpl')
```

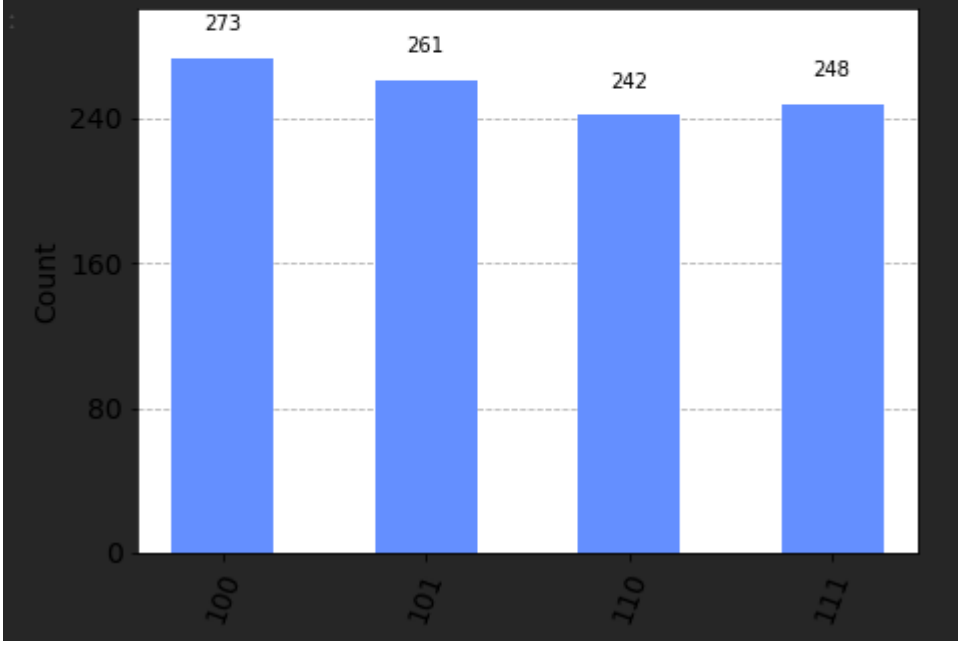


```
circuit.cx(1,2)
circuit.cz(0,2)

circuit.measure([2],[2])
circuit.draw(output='mpl')
```




```
simulator = Aer.get_backend('qasm_simulator')
result = execute(circuit, backend=simulator, shots=1024).result

plot_histogram(result.get_counts(circuit))
```

```
<frozen importlib._bootstrap>:219: RuntimeWarning: scipy._lib.m
C header, got 64 from PyObject
```



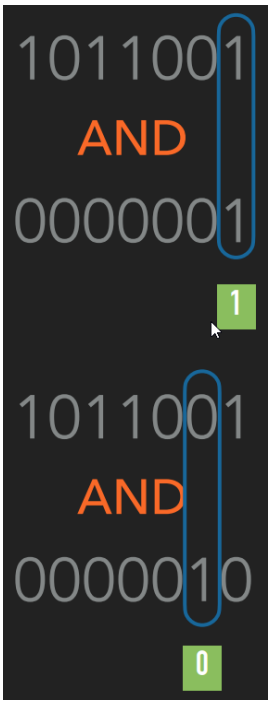
As you can see by the end result - it is in fact very similar. Since we are looking into transportation - we are mostly (only) interested in the state of Q2: which is always 1.

Bernstein vazirani

This is our first quantum algorithm! Once quantum computers get better (close to “perfect”, error free quantum qubits) we can see those improvements in our daily lives as well. From that point on these algorithms can be effectively used to solve real-life problems.

Quantum computers do not always solve everything faster than regular computers. If a “problem” can be solved by applying superposition and entanglement effectively - it can be calculated faster. Certain algorithms can outperform classical computers in some way. **But sometimes classical computers still perform better** - such as Brute Forcing.

Every algorithm is named after their founder.



We pick a number - and we try to guess it. In a string format (it doesn’t really matter - binary format) we can guess this number. A human will try any other combination to try and guess this number. We have A LOT of numbers - and with luck - someone can find this number instantaneously. Classical computers can do it more efficiently - and can take a certain bit - and use an AND operation. If it sees ‘1’ - ‘1’ it knows the binary number is a 1. For each combination it will do this - and it only needs 7 tries to guess this number. Is it 1 - then we know it is correct, is the result 0, then we know this is incorrect (and ignore the 1, put a 0).

Quantum computers can do this calculation **in one single try.**

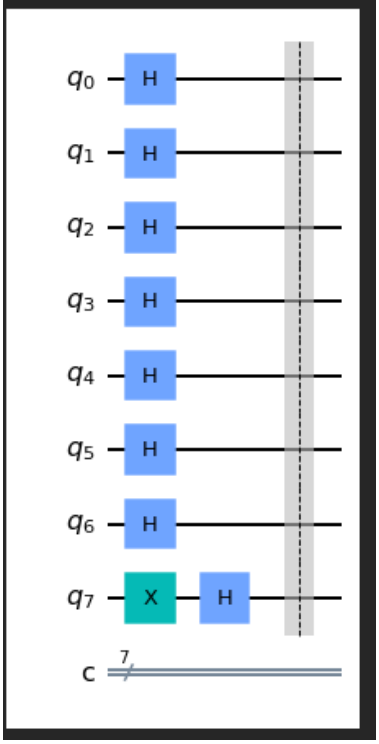
Qiskit

We will build the bernstein algorithm in a hard-coded fashion in the first example - and improve the code afterwards to automate this further.

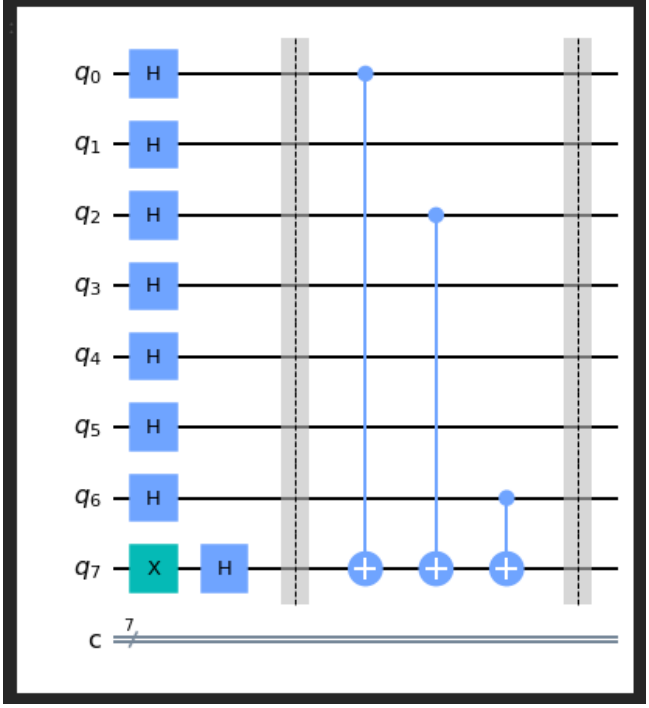
```
from qiskit import *
from qiskit.tools.visualization import plot_histogram
%matplotlib inline

secretNumber = '1000101'

circuit = QuantumCircuit(8,7)
```

Here we simply define our imports, our “secretNumber” to guess, and our circuit.

```
circuit.h([0,1,2,3,4,5,6])
circuit.x(7)
circuit.h(7)

circuit.barrier()
```

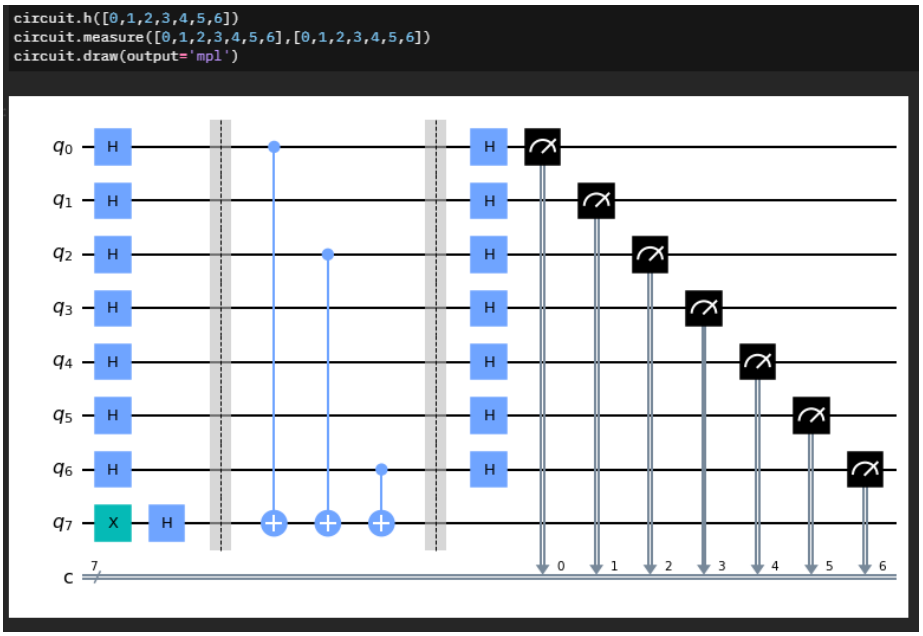



```
circuit.cx(0,7)
circuit.cx(2,7)
circuit.cx(6,7)

circuit.barrier()

circuit.draw(output='mpl')
```



Next up is applying Hadamard gates in order to shift every single qubit to a superposition. A X Gate is applied to our latest qubit (8th qubit) and a Hadamard gate. This qubit will obtain the eventual result. As we are hardcoding this - we put a CNOT Gate at **every '1' position**. This is to simply understand the role of these gates - and transform their state to our last qubit. In the end we simply measure every qubit's state and print the result.

```
circuit.h([0,1,2,3,4,5,6])
circuit.measure([0,1,2,3,4,5,6],[0,1,2,3,4,5,6])
circuit.draw(output='mpl')
```



```
simulator = Aer.get_backend('qasm_simulator')
result = execute(circuit, backend=simulator, shots=1).result()
counts = result.get_counts()

print(counts)
```

```
<frozen importlib._bootstrap>:219: RuntimeWarning: scipy._lib.m
C header, got 64 from PyObject

{'1000101': 1}
```

As you can clearly see - by applying this algorithm and method - the quantum computer is effectively able to guess our secretNumber **in one, single go**. This is currently hard coded.

Automation

To automate this entire flow - you simply use a lot of len() and range() formulas to obtain the initial value from the length of the secretNumber. The end result will be exactly the same - but the code is much more condensed - and fully automated. Additional Hadamard gates are being added according to the for loop - which automatically detects wether or not a “1” or “one” is detected.

```
circuit = QuantumCircuit(len(secretNumber) + 1,len(secretNumber))


circuit.h(range(len(secretNumber)))
circuit.x(len(secretNumber))
circuit.h(len(secretNumber))

circuit.barrier()
```

```
<qiskit.circuit.instructionset.InstructionSet at 0x7f0e8c5253d0>
```

```
for index, one in enumerate(reversed(secretNumber)):
    print(f"index {index} is {one}")
    if one == "1":
        circuit.cx(index, len(secretNumber))

circuit.barrier()
circuit.h(range(len(secretNumber)))

circuit.barrier()

circuit.measure(range(len(secretNumber)),range(len(secretNumber)))

circuit.draw(output='mpl')
```

This way - our quantum computer can now guess **any binary number in one single try**.
https://github.com/atilsamancioglu/QX06-BernsteinVaziraniAlgorithmComplete
https://github.com/atilsamancioglu/QX05-BernsteinVaziraniAlgorithmSimple

Deutsch

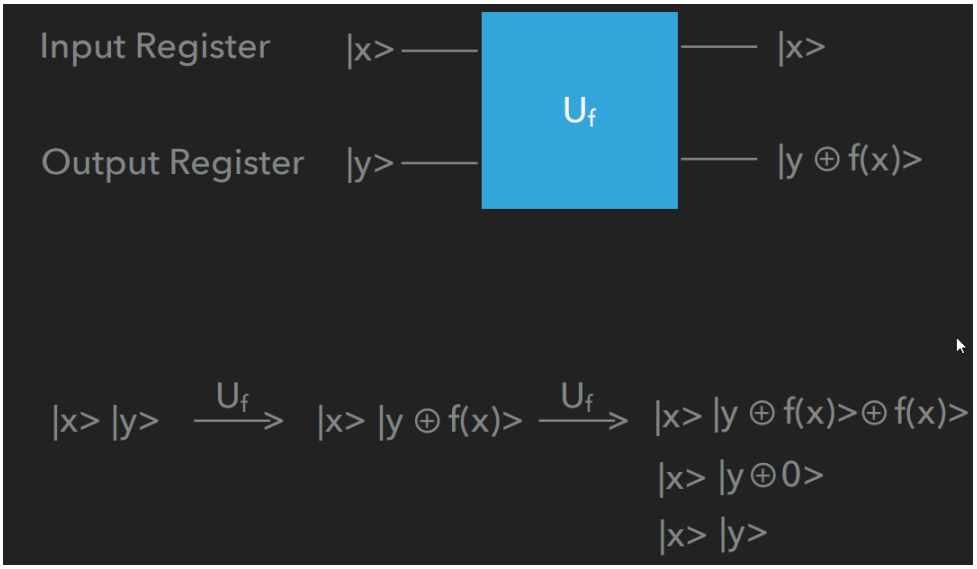
The problem we want to solve with this algorithm is - we have a specific function. It takes 0 or 1 as an input. As output we have 4 possibilities - 00, 01, 10 or 11. If the result is 11 or 00 - we can call this function a constant function; If the input is 00 or 10 we call this a balanced function.

If output does not depend on the input - this is a **constant function**. Same input = same output.
If output depends on the input - this is a **balanced function**. Same input = different output.

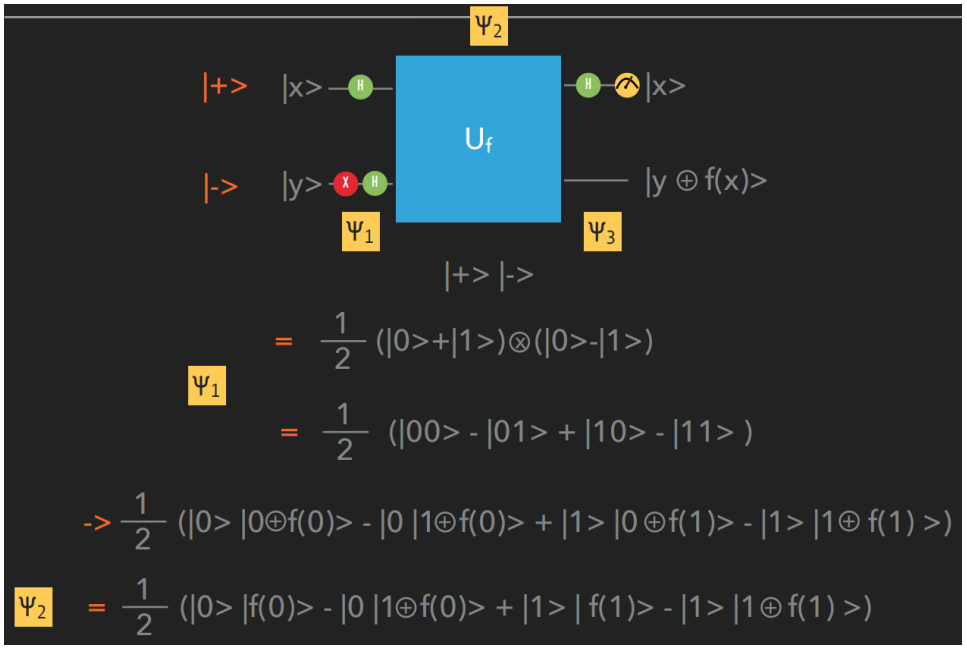
The solution is either: is this a constant function, or a balanced function? This algorithm requires at least two shots - but a quantum computer can do this in **one single go**. *This will not solve real-life problems, but will conclude quantum computers are better in some area's compared to classical computers.*

	f(0)	f(1)	
1	0	0	Constant
2	0	1	Balanced
3	1	0	Balanced
4	1	1	Constant

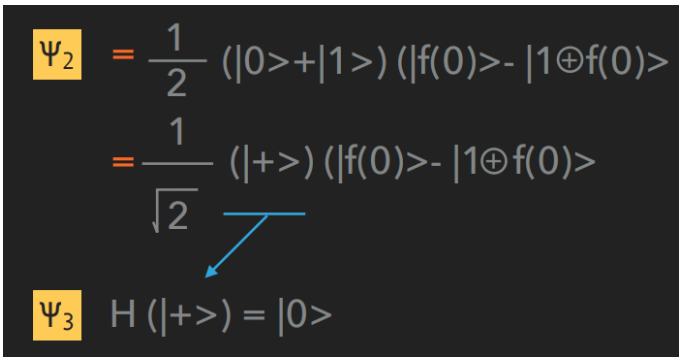
f : {0,1} -> {0,1}
f(0) = f(1) Constant



There are correct and wrong ways of utilising this algorithm. Only using one Hadamard gate for our first Qubit isn't going to work correctly - hence why we need multiple superpositions, and an X Gate.

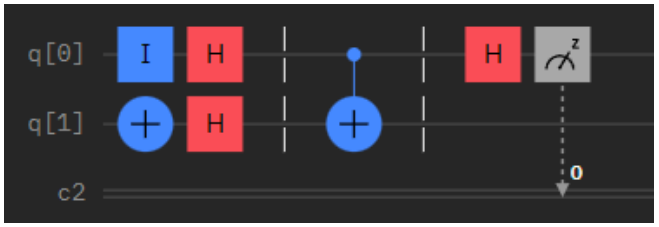


We can further simplify "Sy" 2 to the below formula - and now we still need our final result "Sy" 3.

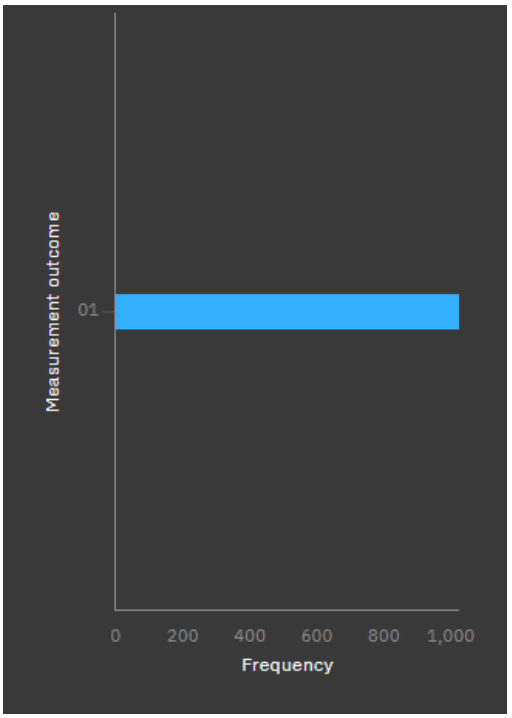


Qiskit & Composer

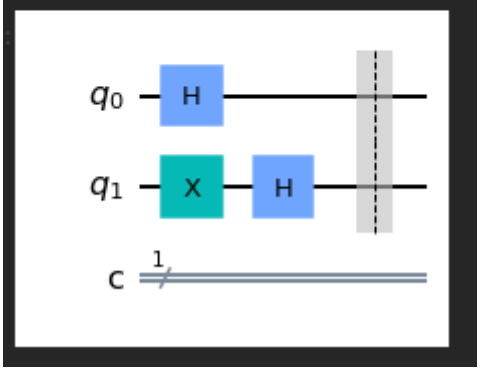
To visualise this better - we can create this algorithm in the composer and code it in Qiskit.



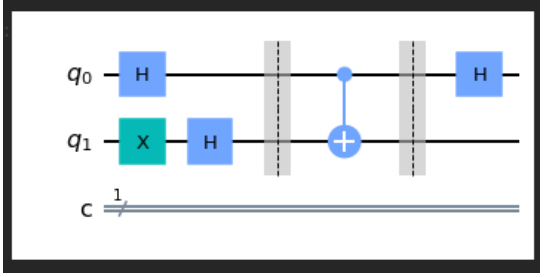
If we run the following composer - we will end up with **one single result** - which is what the Deutsch algorithm does**:**



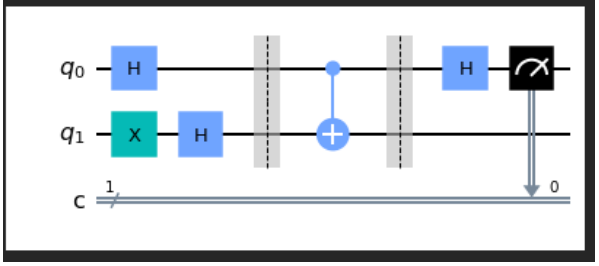

```
circuit.h(0)
circuit.x(1)
circuit.h(1)
circuit.barrier()
circuit.draw(output='mpl')
```




```
circuit.cx(0,1)
circuit.barrier()
circuit.h(0)
circuit.draw(output='mpl')
```



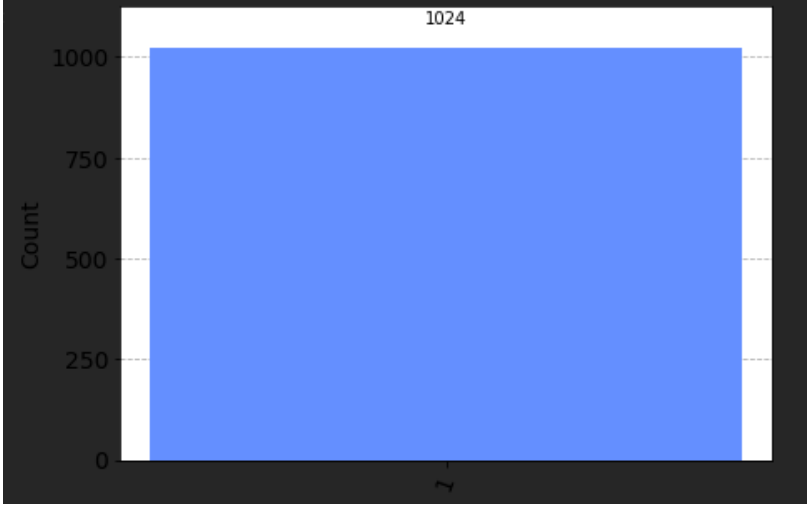

```
circuit.measure(0,0)
circuit.draw(output='mpl')
```



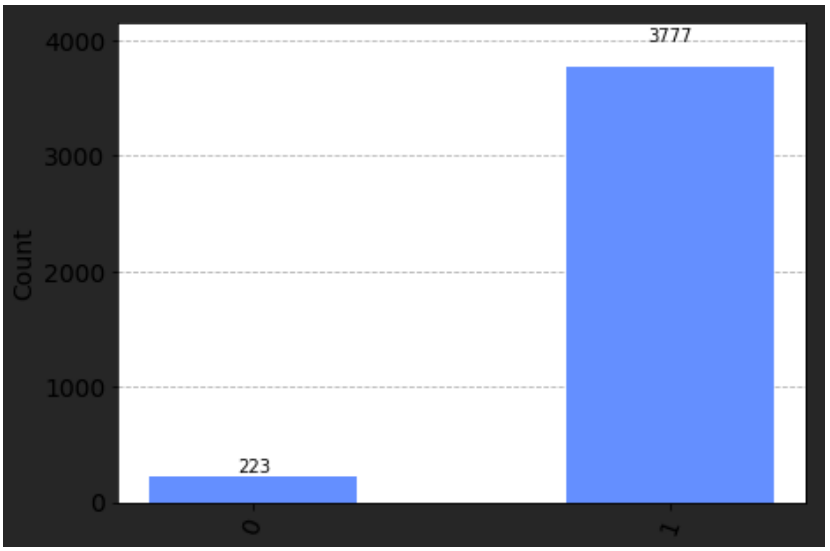
Now you can see that we only get one result for every shot we took. This is a **balanced function** and requires only one single shot:

```
backend = Aer.get_backend('qasm_simulator')
result = execute(circuit, backend=backend, shots=1024).result()
counts = result.get_counts(circuit)

plot_histogram([counts])
```

```
<frozen importlib._bootstrap>:219: RuntimeWarning: scipy._lib.me
C header, got 64 from PyObject
```



Now the real issue is the fact we still have **quantum noise.** We now can effectively see that the quantum computers from today still aren't perfect yet (his is the same code as used in a previous exercise - HeloQuantum).



Once this calculation is more accurate - we know the quantum computer is effectively being a **lot more accurate.**
https://github.com/atilsamancioglu/QX07-DeutschAlgorithm

Grover’s

This algorithm can be used in real-life situations (unlike the previous algorithms). This is a **search algorithm** that can be used to search something. You have a series of numbers - you need to find a number - but the list is mixed. In order to find the number - you can guess it one by one… but this algorithm can effectively search the list and give you the number.

Classic search algorithm

In this example we will look at how a **classic approach** is handled nowadays in Python. This is a very simple example - but the list could have been a thousand numbers. Since a classic solution will have to iterate over **each single number** to look for our winningNumber - it will **take time** until you have found the solution. Grover’s algorithm can do this **much faster** (in one, single shot) - which may not succeed every single time at doing this. It just **increases the possibility / probability of finding the number significantly faster**.

```
from qiskit import *
from qiskit.tools.visualization import plot_histogram
%matplotlib inline
```

```
#Classical problem
MyList = [5,4,6,9,1,2,3,7,8,0]
```

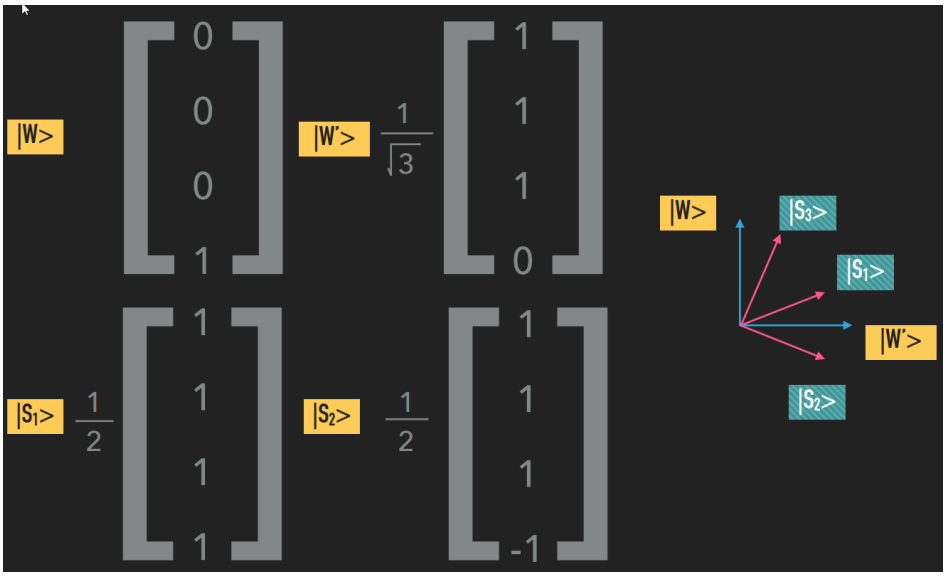
```
def oracle(number):
    winningNumber = 8
    if number == winningNumber:
        response = True
    else:
        response = False
    return response
```

```
for index, number in enumerate(MyList):
    if oracle(number) is True:
        print(f"Winning number index: {index}")
        print(f"execution count: {index + 1}")
        break
```

```
Winning number index: 8
execution count: 9
```

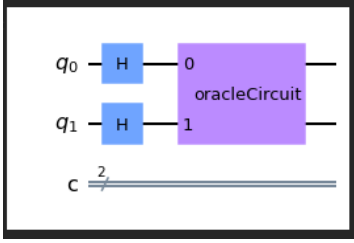
Applying Grover's

We're going to flip the phase of the winning state - instead of "11" we will succeed by applying "-11". This is a **reflection circuit** - which will increase the possibility of finding this winning state. We will not entangle - we are just dealing with a superposition and taking a tensor product of these possibilities. We will be looking into creating our own gate to be utilised with this algorithm.

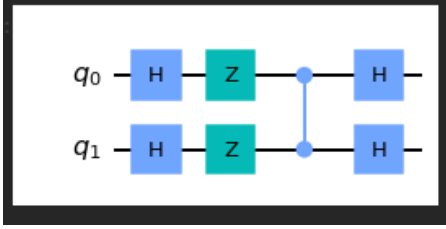


Reflection makes sure that we can look for the winning state (|w>). Every single time we do this reflection we get one step closer to the winning state. Perhaps not with 100% probability - but it tries to maximise the possibility of obtaining this Kat w state.

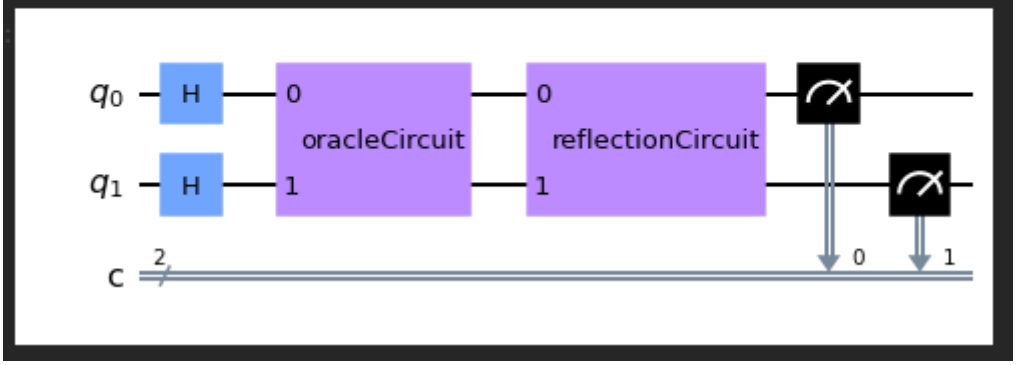
```
mainCircuit = QuantumCircuit(2,2)
mainCircuit.h([0,1])
mainCircuit.append(oracleCircuit,[0,1])
mainCircuit.draw(output='mpl')
```



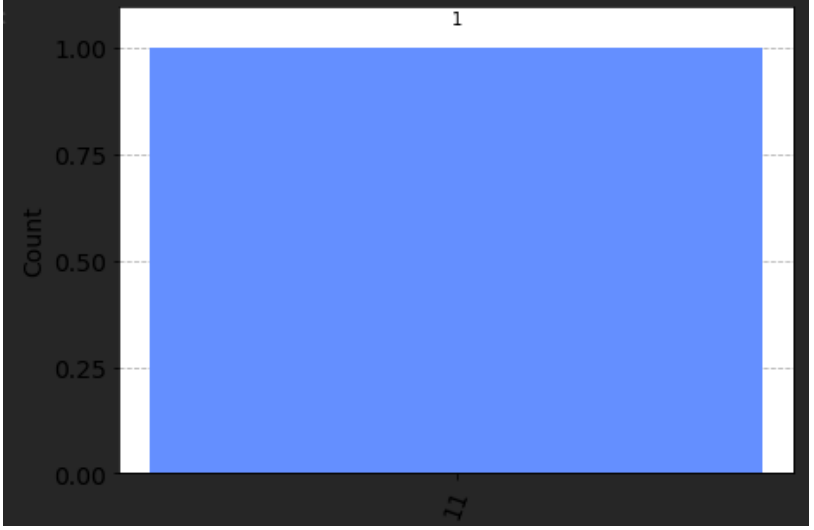

```
reflectionCircuit = QuantumCircuit(2, name="reflectionCircuit")
reflectionCircuit.h([0,1])
reflectionCircuit.z([0,1])
reflectionCircuit.cz(0,1)
reflectionCircuit.h([0,1])
reflectionCircuit.to_gate()
reflectionCircuit.draw(output='mpl')
```



```
mainCircuit.append(reflectionCircuit,[0,1])
mainCircuit.measure([0,1],[0,1])
mainCircuit.draw(output='mpl')
```




```
backend = Aer.get_backend('qasm_simulator')
result = execute(mainCircuit,backend=backend,shots=1).result()
counts = result.get_counts(mainCircuit)
plot_histogram([counts])
```

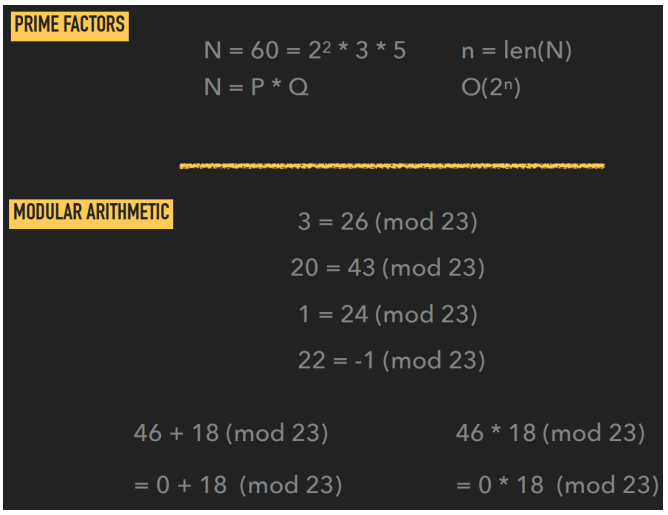


In the above coding example you can see that Grover's can get the winning state in one, single go. https://github.com/atilsamancioglu/QX08-GroversAlgorithm

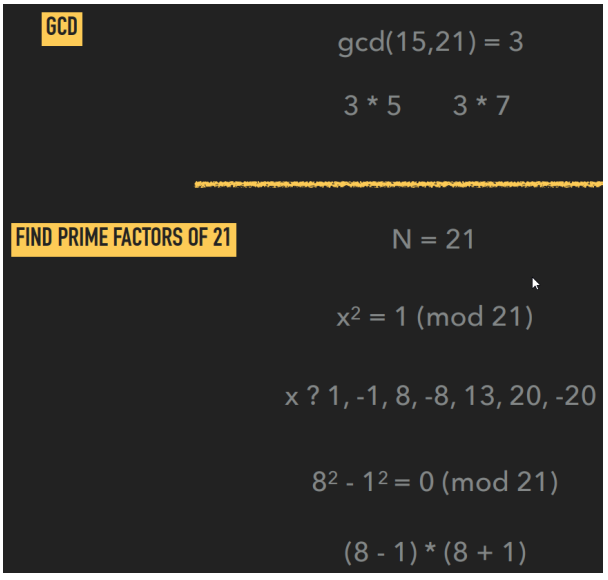
Shor's

One of the most important - but one of the hardest to understand. It will clearly determine how our lives will change - we will be able to find prime factors of large numbers. This can effectively make or break encryption we understand nowadays.

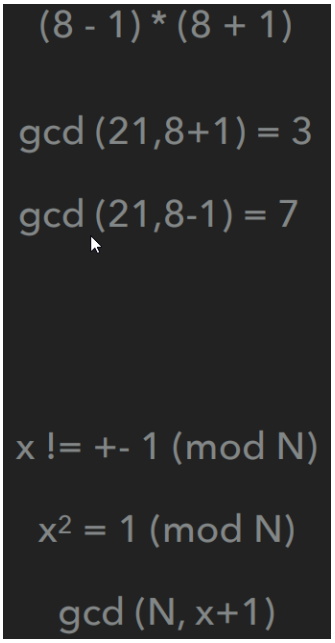
Finding a prime factor is easily done with a small number. But finding prime factors of large numbers is a lot harder for classical computers. Quantum computers will be able to do this a lot faster. In order to break an algorithm such as RSA - you need to spend a "thousand years" in order to calculate this prime factor. A Quantum computer with X qubits will be able to do this in just a matter of time (possibly less than a minute) in the near future once qubits are more accurate. It is concerning - but **this is a real probability now quantum computing is effectively being created and progressing.**



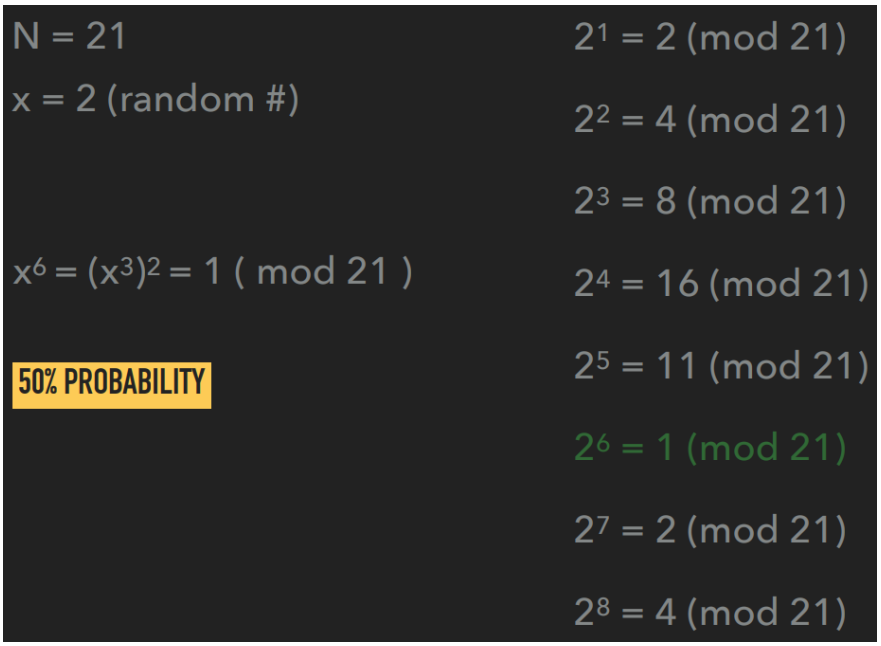
Modular arithmetic is 'simple': 3 is the rest number, and then we divide 26 to the mod of 23. Thanks to a GCD - we will be able to find a prime factor of, for example, 21. In the below example we want to find X.



The prime factors of 21 are effectively 3 and 7 - this is a classical computation. **General rule:** it shouldn't be -1 or +1 - this will just make it easier. But we might consider a more real-life calculation.



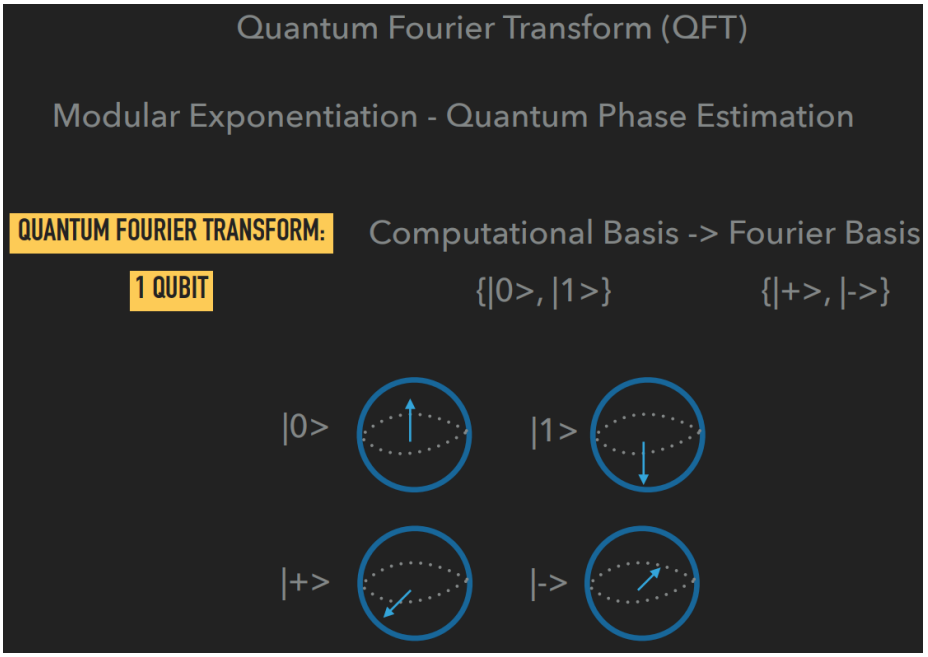
To find these prime numbers - we simply go into a (big) loop to find the end result. This is a periodic function - we take a random number - and we have a 50% probability of finding the solution (luck). The period number should be an even number. Effectively we want to find the periodic number.



Above is mainly the way we calculate these prime factors from a classical computer standpoint - or basic mathematics. The quantum way does this in a completely different way.
https://github.com/atilsamancioglu/QX11-ShorsClassical

Shor’s easy way

We will be using layers of abstraction in order to make things easier for us. One algorithm is



As we write out simple program - Shor’s can calculate the prime numbers in a matter of time (and probably a matter of minutes as well). The calculation still takes a bit of time - but is going to be scaled up in the future. The below code can hereby calculate the prime numbers for 15. When we increase this prime number - it will take more time and effort already.

```
import math
import numpy as np
from qiskit import Aer
from qiskit.utils import QuantumInstance
from qiskit.algorithms import Shor
```

```
<frozen importlib._bootstrap>:219: RuntimeWarning: scipy._lib.messagestream.MessageStream size chan
C header, got 64 from PyObject
```

```
N = 15
backend = Aer.get_backend('aer_simulator')
quantum_instance = QuantumInstance(backend, shots=1024)
shor = Shor(quantum_instance=quantum_instance)
result = shor.factor(N)
print(f"The list of factors of {N} as computed by the Shor's algorithm is {result.factors[0]}.")
```

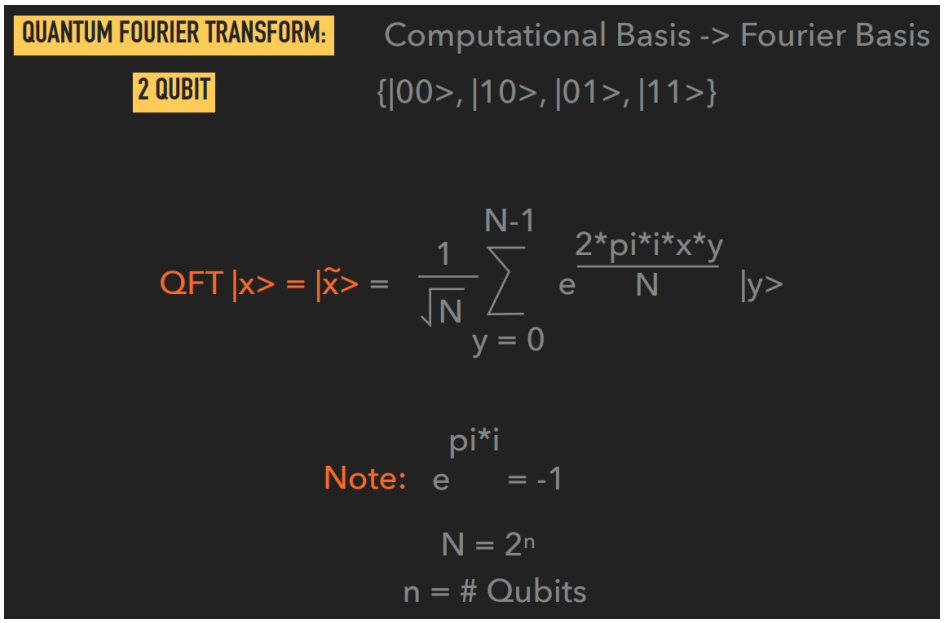
```
/tmp/ipykernel_691/2546798916.py:4: DeprecationWarning: The Shor class is deprecated as of Qiskit T
        no sooner than 3 months after the release date.
        It is replaced by the tutorial at https://qiskit.org/textbook/ch-algorithms/shor.html

  shor = Shor(quantum_instance=quantum_instance)

The list of factors of 15 as computed by the Shor's algorithm is [3, 5].
```

After about a minute or 2 - the algorithm was able to calculate the prime numbers 3 and 7 for 12. This still requires quite some time - even for a simulation!

```
The list of factors of 21 as computed by the Shor's algorithm is [3, 7].
```

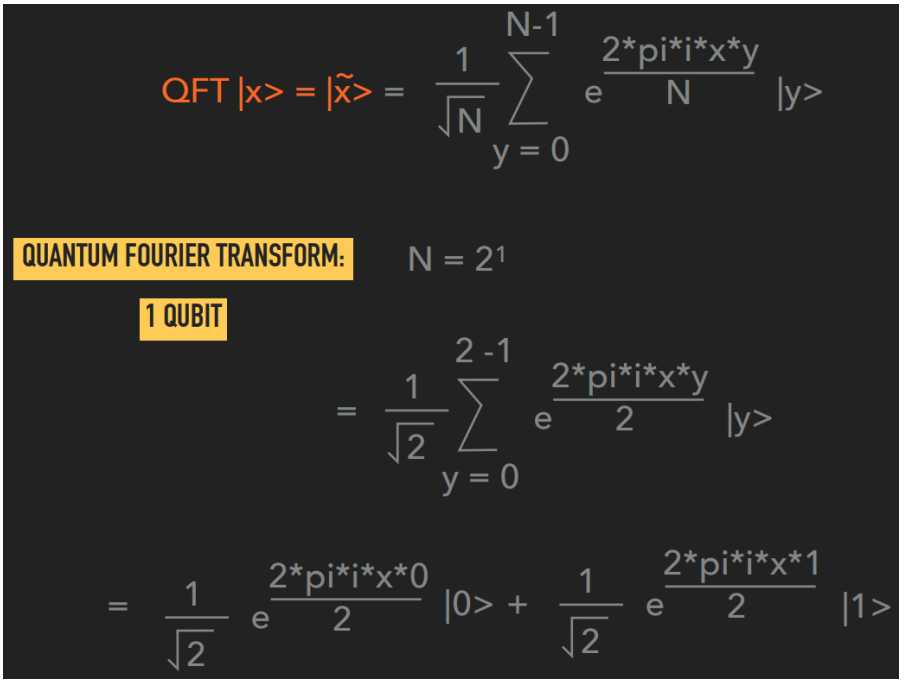
Quantum Fourier Transform (QFT)

In order to understand the algorithm better - we need to understand QFT. We will be going to transform the basis. The computational basis is either in 0 state or 1 state.

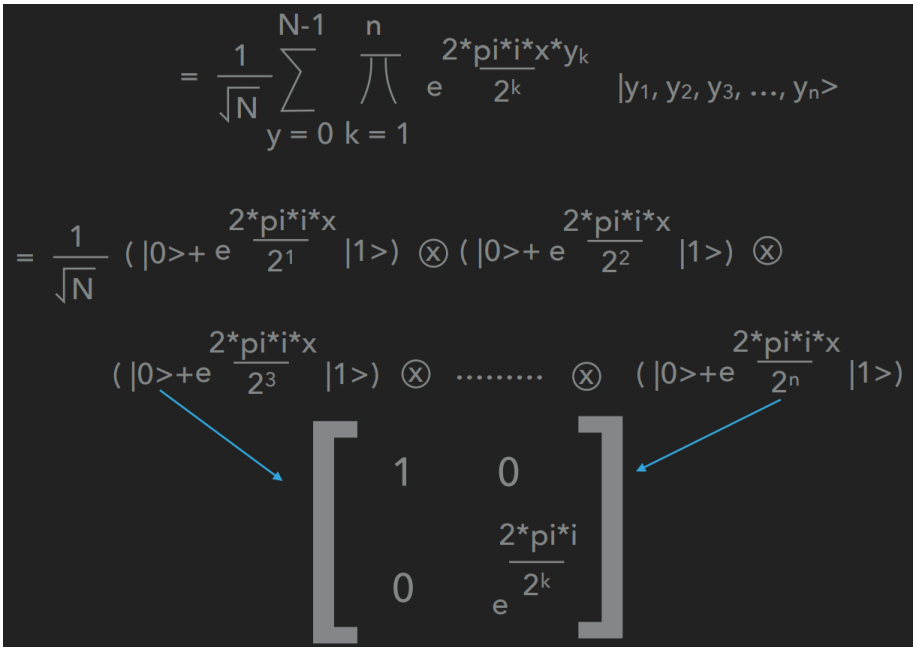


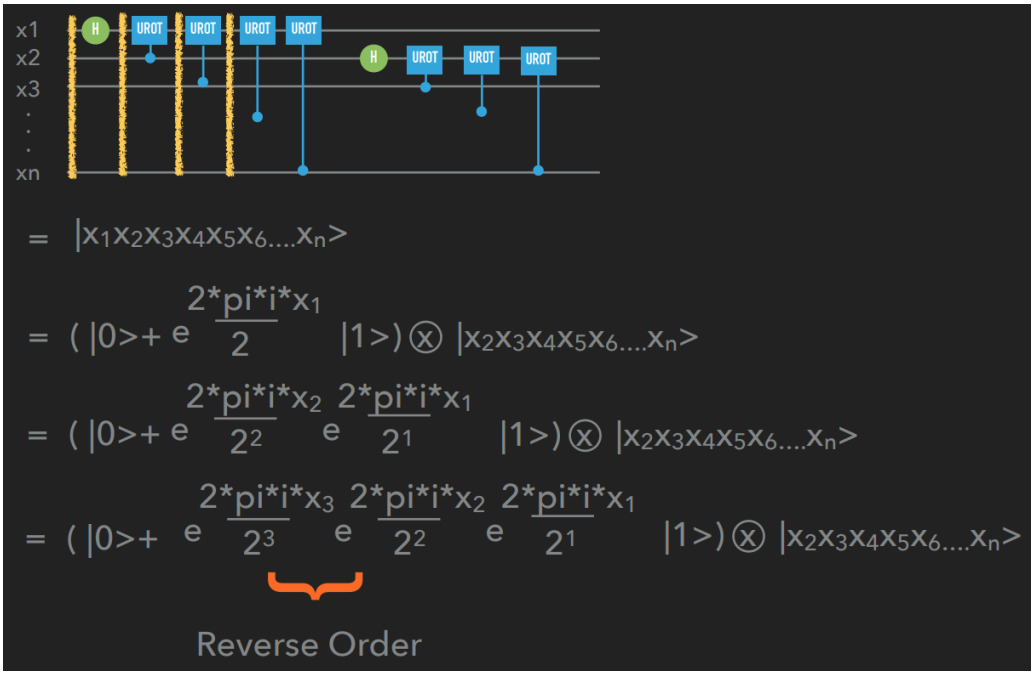
We effectively sum up the formula and specifically look at e to the pi*i = -1.

https://www.youtube.com/watch?v=v0YEaeICIKY going more in depth about the mathematical explanation.



The QFT formula is essentially the basic formula used. Essentially we end up with a matrix (or a U gate) - we know how to build our circuit (and the matrix / formula).

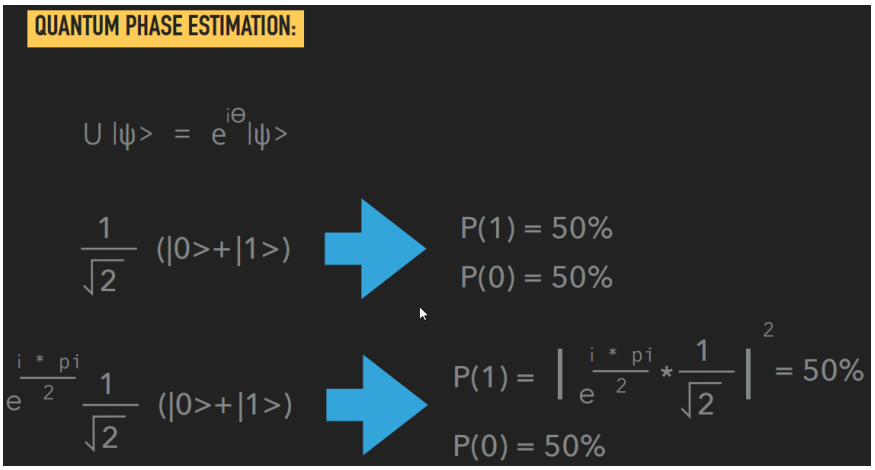




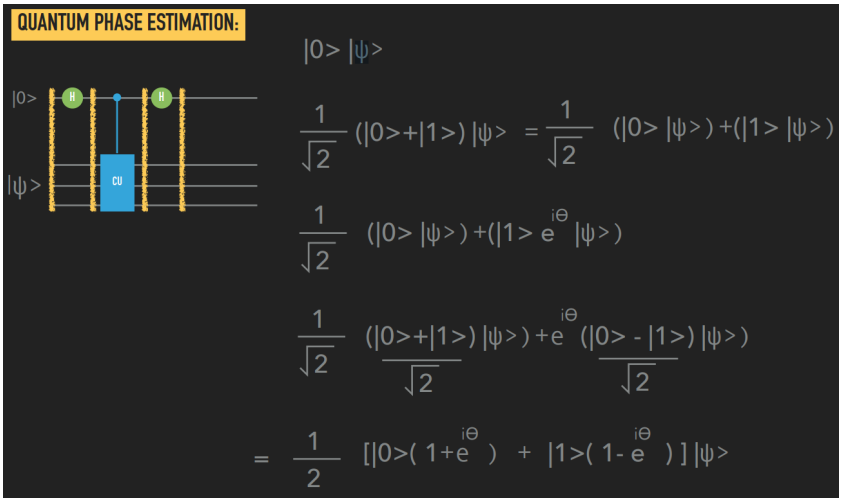
Mathematically it is essentially the reverse order. If you find a QFT circuit you can see swap gates. Now we want to use this in a Quantum Phase Estimation.

Quantum Phase Estimation (QPE)

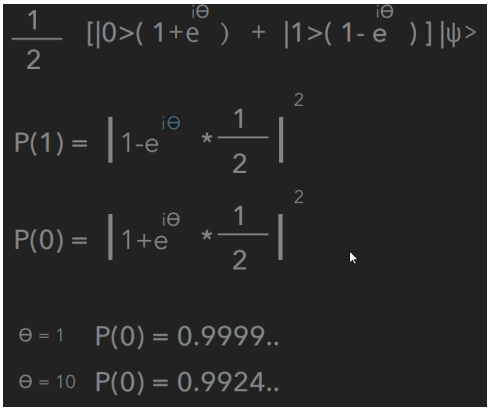
Essentially we want to find a phase - an angle. QPE will help us find the Theta (iΘ).



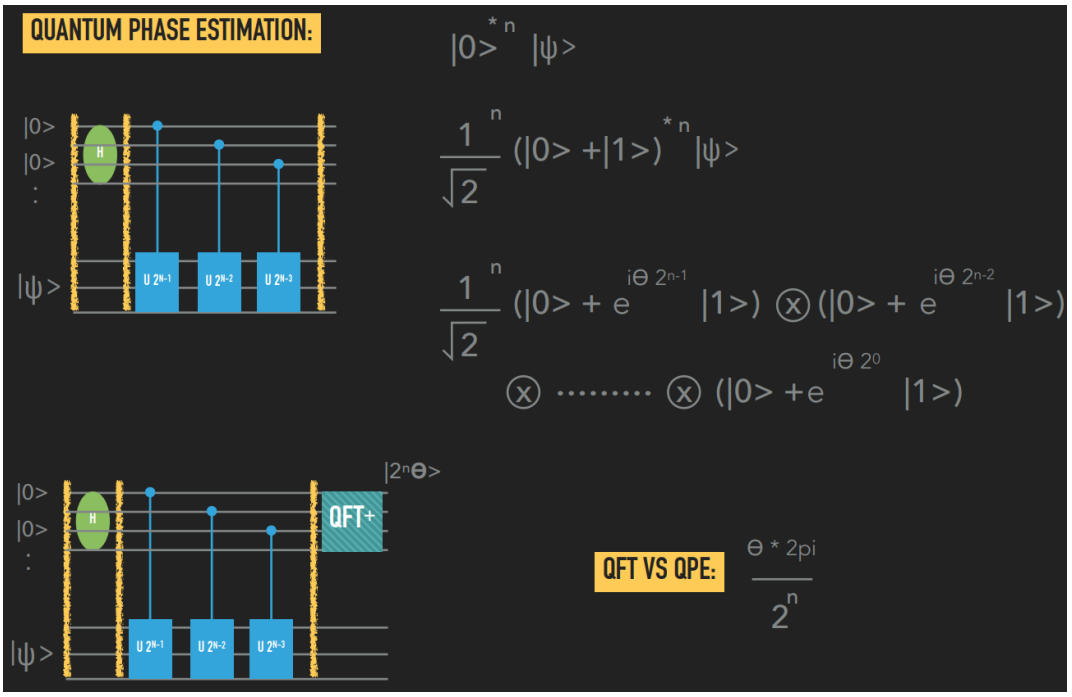
With one qubit (represented by the |0> Ket) - we can calculate further formula.



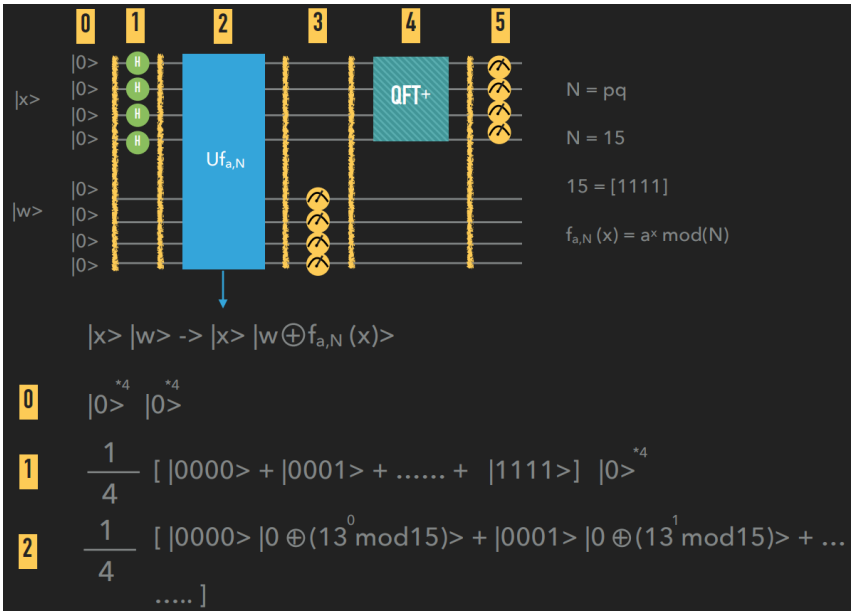
If we want to calculate the probability of one - we need to decrease the theta / phase. Once we find this theta, we can effectively increase the probability. It may take a very long time. Theta could be 1, 10 or even 99.



The above example is provided with **one qubit** - we obviously want to do this with **more qubits**.

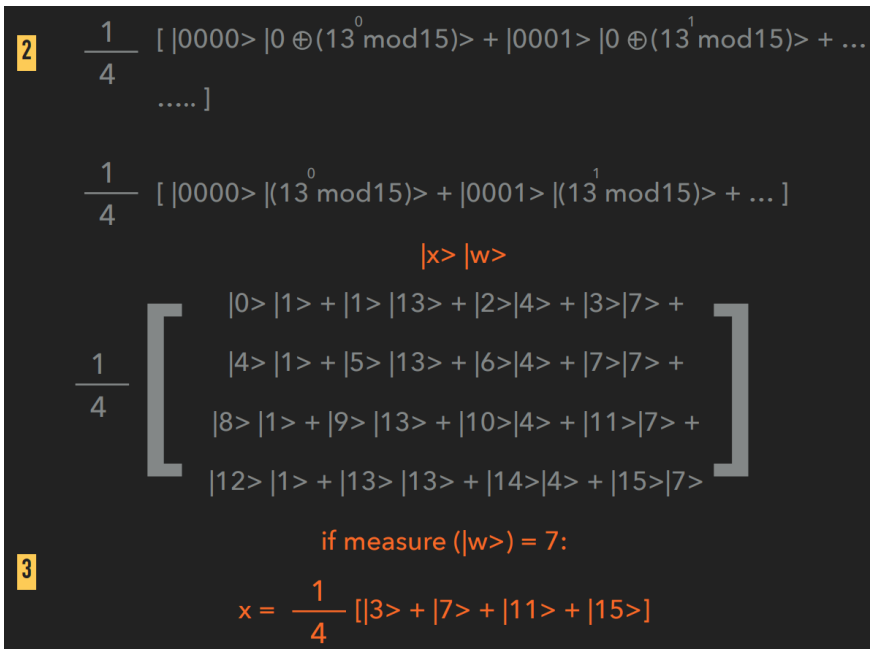


The major difference between both: the angle is different. If we provide the adjoint of the QFT - we end up with the theta: |2nΘ>.

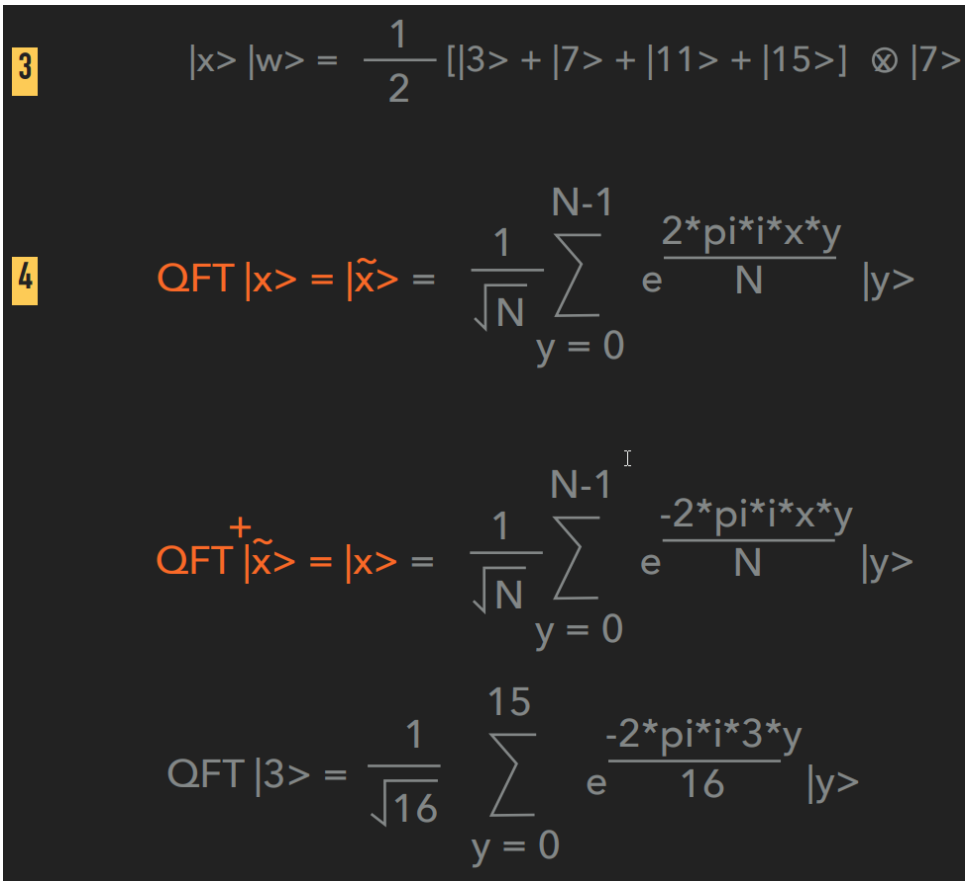


- 0: qubits!
- 1: apply Hadamard gates for superpositions.
- 2: apply for the QPE.
- 3: measure the results to find the P and Q. (N = pq).
- 4: apply the adjoint of QFT(+).
- 5: measuring the final results.

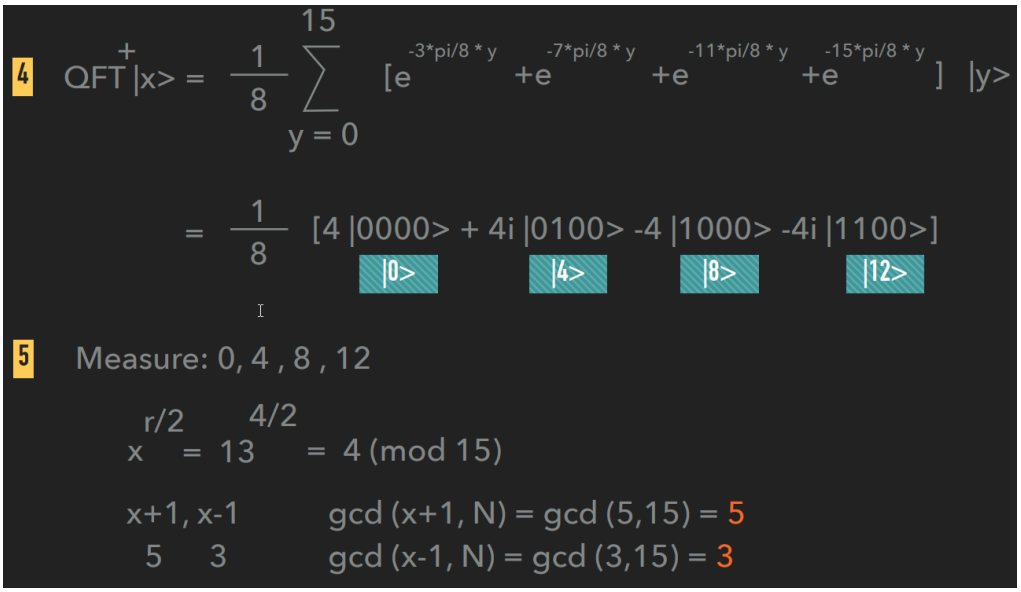
We are trying to write Ket X and Ket W in one go. Calculating these values will result in a “final” matrix. Ket W is either 1, 13, 4 or 7, for example. Now we take one measurement (assumption) - for example 7. It would change the X values in the end.



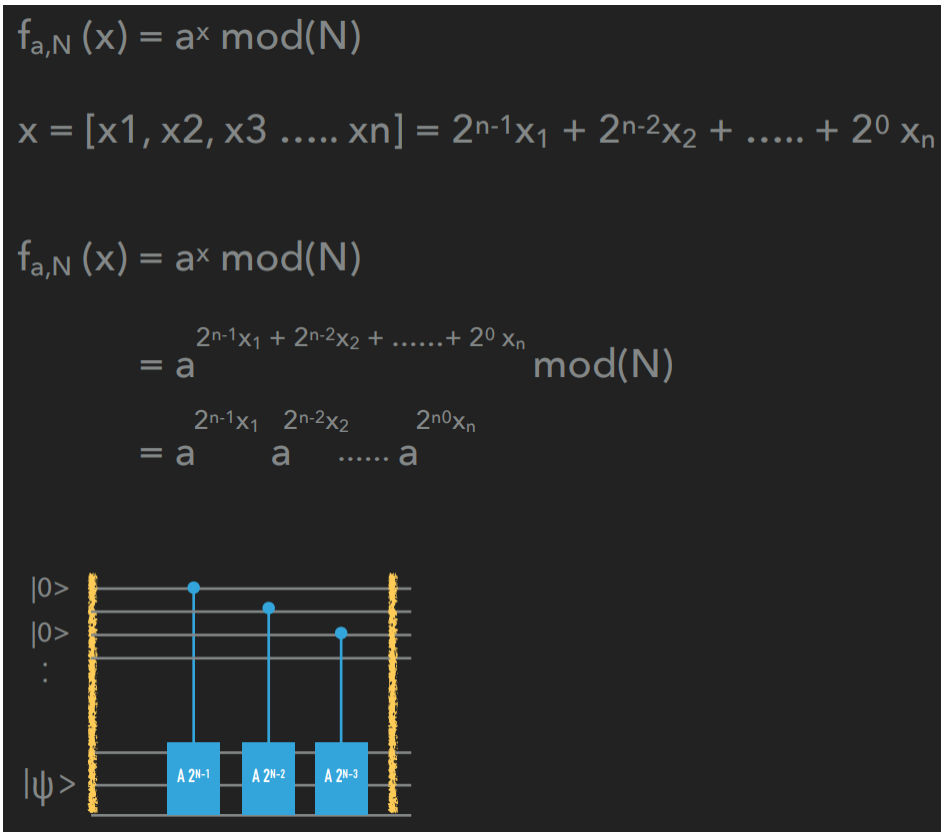
In step 4 we implement the QFT - take the adjoint and transpose the imaginary parts. (2 -> -2). Number 4 is only one single example - we should do this for every number.



If we calculate all 16 values - we eventually end up with 4 values (0, 4, 8 and 12 - binary or decimal). Now we can calculate the X and effectively get the prime numbers.



The function we eventually have defined is the following in order to find the modular powers:



Exponential calculator: https://github.com/atilsamancioglu/QX10-ExpCalculator
Shor's quantum circuit manual:
https://github.com/atilsamancioglu/QX12-ShorsQuantumManual