Grover's Algorithm

Suppose you've got 10 boxes, each with a paper with a random number on it, and you're searching for a number which may or may not be in one of the boxes. You'll have to search the first box, then the second, then the third... and so on, until you've found what you were looking for. Best case scenario - it just took one search (the number could be in the first box)! Worst case - it took 10 searches(it could've been in the last box, or in no box at all!).

Even if you get a bit clever, sort all the boxes according to the numbers in them, in ascending or descending order, and apply something like **binary search**, it'll almost take $\log_2(n)$ searches.

Grover's Algorithm is a search algorithm, which solves the problem in O($\sqrt{n}$) **time complexity**(it takes $\pi\sqrt{N}/4$ searches, for one possible match).

Sign flipping

Consider you've 3 qubits, with the state vector :-

$$\frac{1}{\sqrt{8}}|000\rangle + \frac{1}{\sqrt{8}}|001\rangle + \frac{1}{\sqrt{8}}|010\rangle + \frac{1}{\sqrt{8}}|011\rangle + \frac{1}{\sqrt{8}}|100\rangle + \frac{1}{\sqrt{8}}|101\rangle + \frac{1}{\sqrt{8}}|110\rangle + \frac{1}{\sqrt{8}}|111\rangle$$

```
using Yao, YaoPlots
```

```
8×1 Array{Complex{Float64},2}:
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
```

```
begin
    qubits = uniform_state(3)
    state(qubits)
end
```

Suppose there was a circuit U, and if you passed the three qubits to U, the resultant state vector would look somewhat like this :-

$$\frac{1}{\sqrt{8}}|000\rangle + \frac{1}{\sqrt{8}}|001\rangle - \frac{1}{\sqrt{8}}|010\rangle + \frac{1}{\sqrt{8}}|011\rangle + \frac{1}{\sqrt{8}}|100\rangle + \frac{1}{\sqrt{8}}|101\rangle + \frac{1}{\sqrt{8}}|110\rangle + \frac{1}{\sqrt{8}}|111\rangle$$

One thing we know about this *magical* circuit is that its matrix representation looks like this,

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

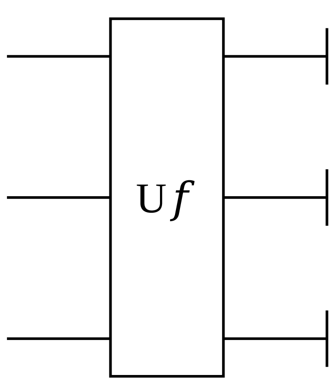
Try multiplying the above state vector to the new matrix.

```
8×1 Array{Complex{Float64},2}:
  0.35355339059327373 + 0.0im
  0.35355339059327373 + 0.0im
 -0.35355339059327373 + 0.0im
  0.35355339059327373 + 0.0im
  0.35355339059327373 + 0.0im
  0.35355339059327373 + 0.0im
  0.35355339059327373 + 0.0im
  0.35355339059327373 + 0.0im
```

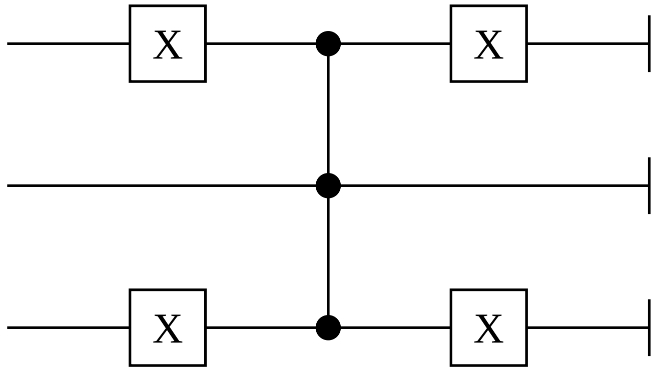
```
let
    U = rand(8,8) |> U->round.(round.(U * inv(U)))
    U[3,3] = -1
    U * state(qubits)
end
```

Creating the magic circuit

You'll have to think about each circuit individually, according to the the element you want to flip. We're flipping $|010\rangle$ in this case. The circuit will be denoted by U_f.

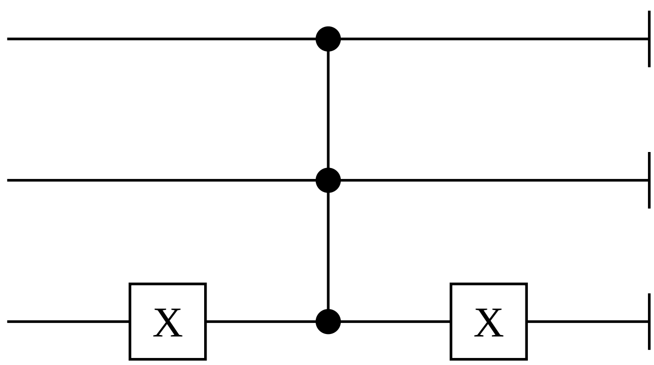


Below is my implementation of this magic circuit




```
begin
    Uf = chain(3, repeat(X, [1 3]), control(1:2, 3=>Z), repeat(X, [1 3]))
    plot(Uf)
end
```

The circuit for $|011\rangle$ will be




```
begin
    Uf_1 = chain(3, repeat(X, [3]), control(1:2, 3=>Z), repeat(X, [3]))
    plot(Uf_1)
end
```

```
8×1 Array{Complex{Float64},2}:
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
 0.35355339059327373 + 0.0im
```

```
state(uniform_state(3))
```

After passing the uniform qubits through Uf.

```
8×1 Array{Complex{Float64},2}:
  0.35355339059327373 + 0.0im
  0.35355339059327373 + 0.0im
 -0.35355339059327373 - 0.0im
  0.35355339059327373 + 0.0im
```

```
0.35355339059327373 + 0.0im
0.35355339059327373 + 0.0im
0.35355339059327373 + 0.0im
0.35355339059327373 + 0.0im
```

```
state(uniform_state(3) |> Uf)
```

Amplitude Amplification

Lets consider you want to increase one particular probability amplitude and decrease the rest.

If your qubits were initially in the state :-

$$\frac{1}{\sqrt{8}}|000\rangle + \frac{1}{\sqrt{8}}|001\rangle + \frac{1}{\sqrt{8}}|010\rangle + \frac{1}{\sqrt{8}}|011\rangle + \frac{1}{\sqrt{8}}|100\rangle + \frac{1}{\sqrt{8}}|101\rangle + \frac{1}{\sqrt{8}}|110\rangle + \frac{1}{\sqrt{8}}|111\rangle$$

Then you want them in the state :-

$$1 \times |010\rangle$$

That means, ideally, the probability amplitude of $|010\rangle$ being close to 1, while of others being close to 0.

Inversion about the mean is a neat trick which helps you achieve that.

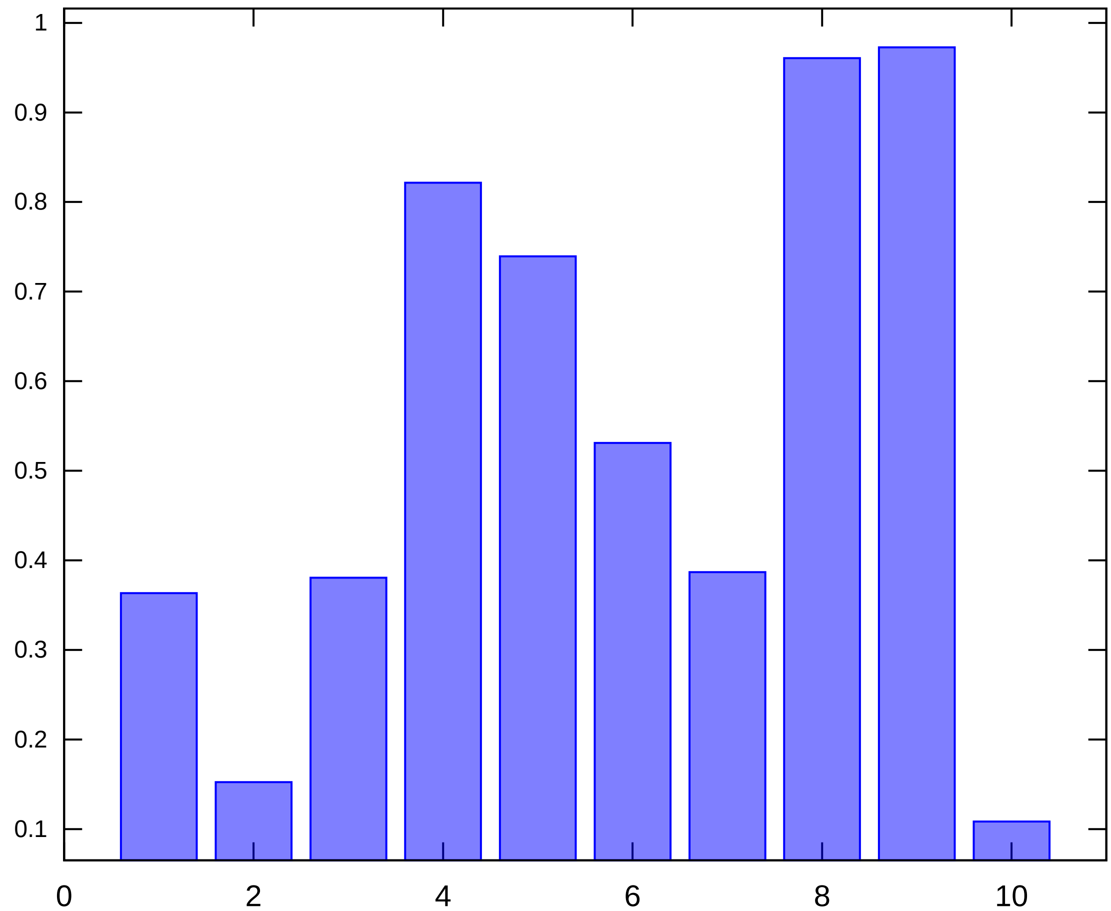
Inversion about the mean

Its simple. I'll give an example.

```
testmat =
▸Float64[0.363365, 0.152511, 0.38071, 0.821608, 0.739469, 0.531123, 0.386858, 0.96065
```

```
testmat = rand(10)
```

Lets plot it as a histogram




```
bar(testmat)
```

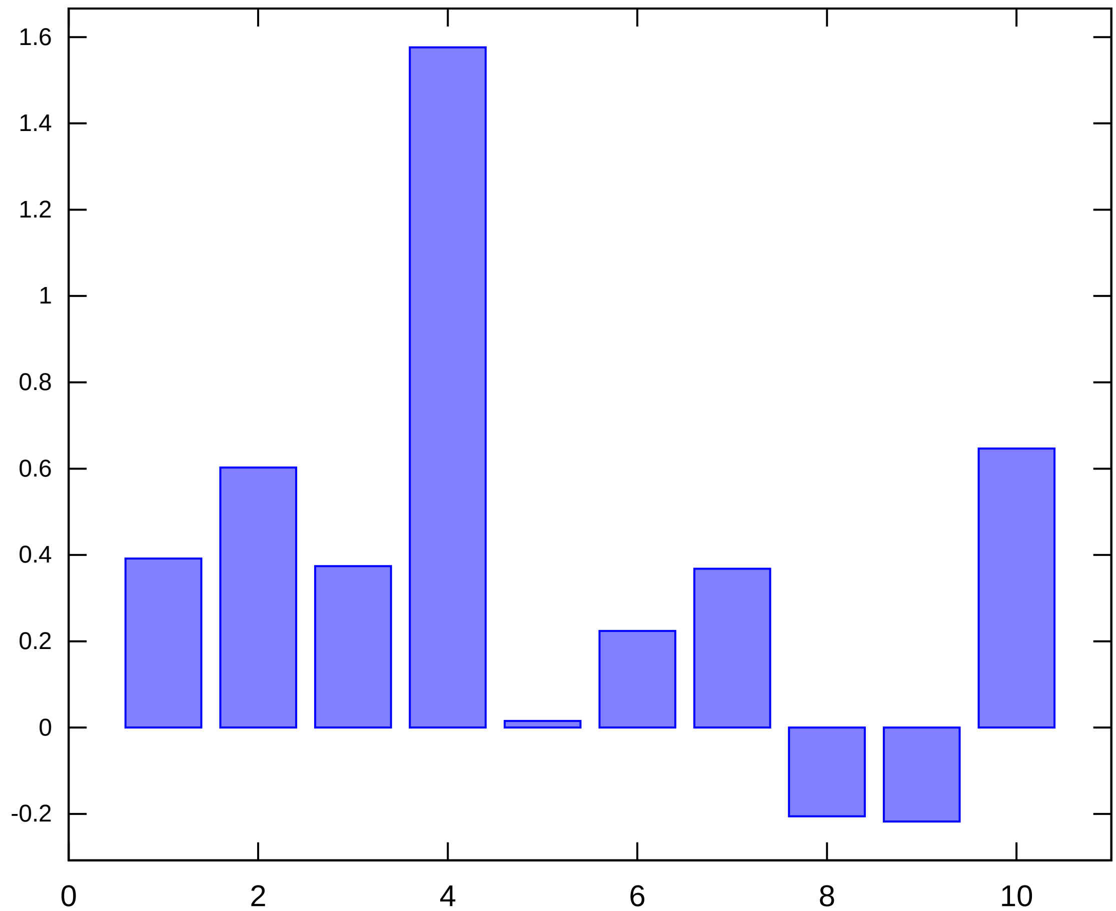
Say, I want to amplify the 4th element. Here's the procedure

```
mean (generic function with 1 method)
```

```
mean(x) = sum(x)/length(x)
```

```
Float64[0.391478, 0.602332, 0.374132, 1.57645, 0.0153731, 0.22372, 0.367985, -0.2058
```

```
begin
    matamplified = copy(testmat)
    matamplified[4] = -matamplified[4]
    matamplified = (2 .* mean(matamplified)) .- matamplified
end
```




```
bar(matamplified)
```

What just happened:-

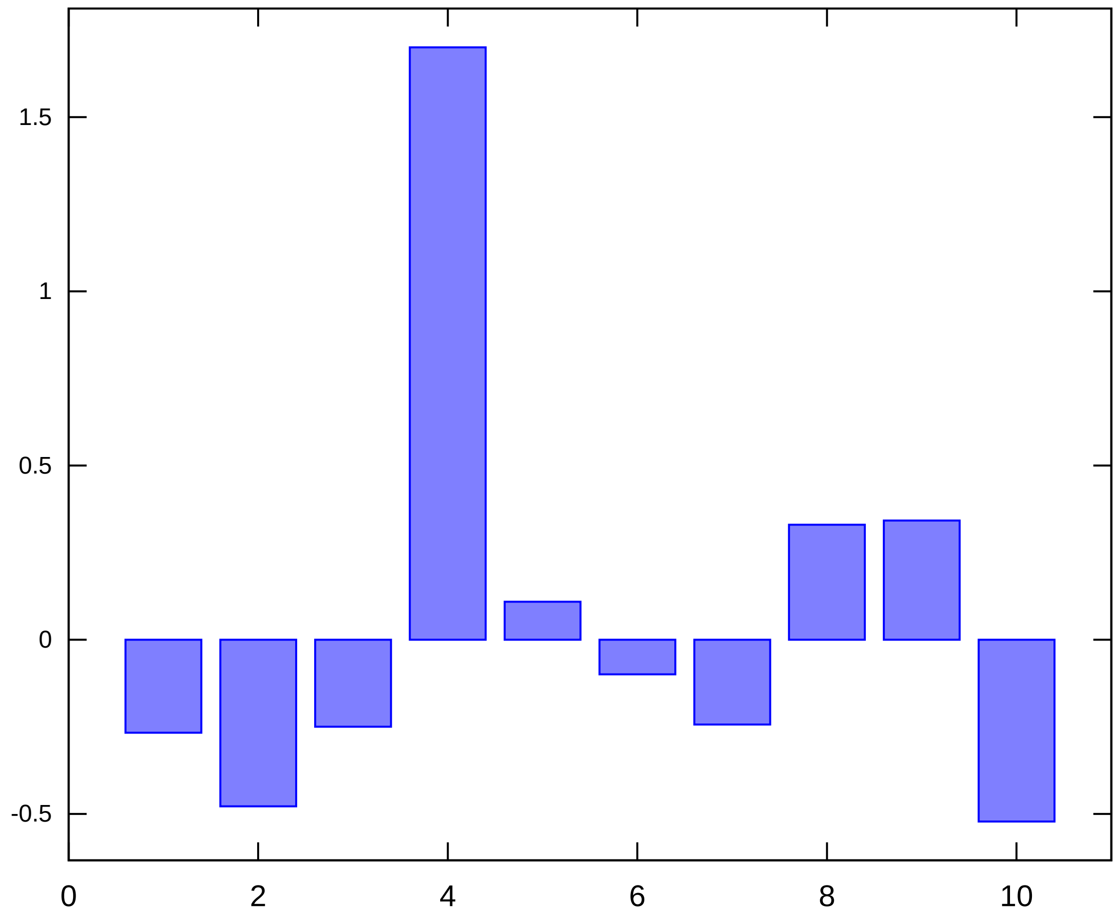
- In an array, choose the element you want to amplify.
- Flip the sign of that element.
- Then the new array with the element amplified, will have the elements :-

$$Amplified\ array = (2 \times mean) - (the\ original\ array\ with\ the\ flipped\ element).$$

What if we do it again? Will it get amplified again?

```
Float64[-0.267216, -0.478069, -0.24987, 1.70071, 0.108889, -0.0994576, -0.243723, 0.
```

```
begin
    newmatamplified = copy(matamplified)
    newmatamplified[4] = -newmatamplified[4]
    newmatamplified = (2 .* mean(newmatamplified)) .- newmatamplified
end
```

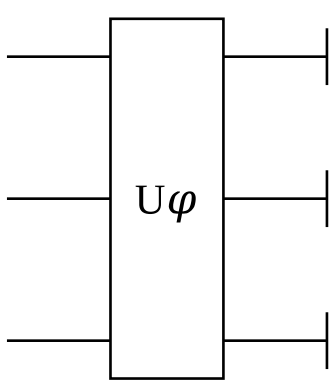



```
bar(newmatamplified)
```

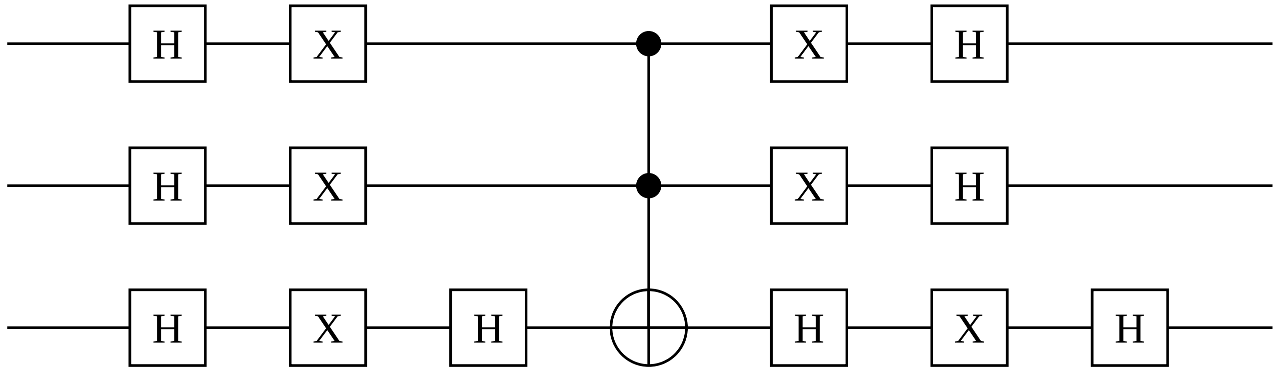
So yes, it does amplified again

The Circuit Implementation

The circuit for amplification looks like this, if the sign of the desired state is flipped

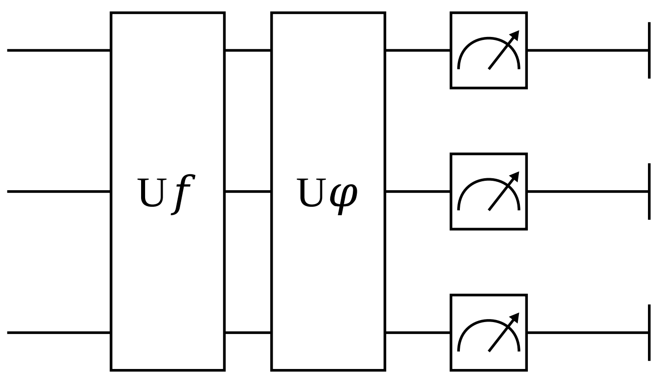


Again, below is my implementation of it.

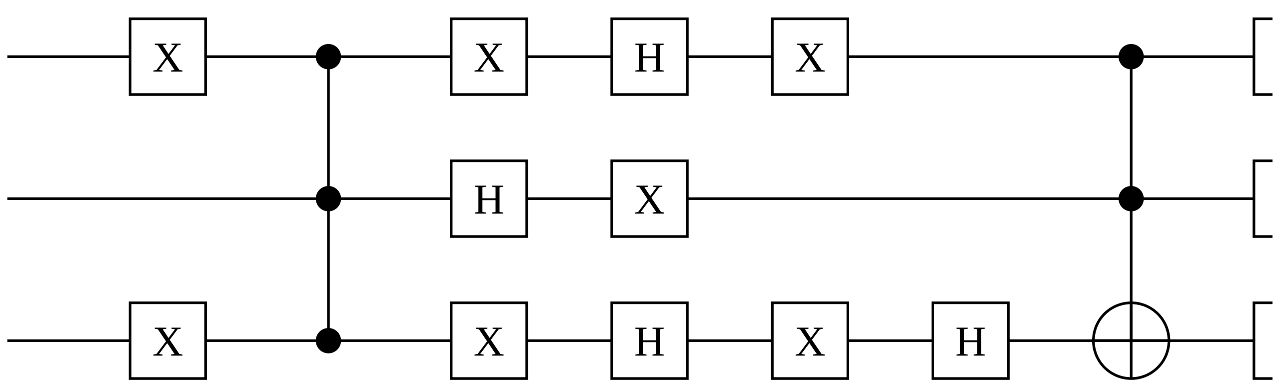



```
begin
    Uφ = chain(3, repeat(H, 1:3), repeat(X, 1:3), put(3=>H), control(1:2, 3=>X),
put(3=>H), repeat(X, 1:3), repeat(H, 1:3))
    plot(Uφ)
end
```

Combining this with the circuit for flipping, we get the Grovers Search Circuit, to which we feed qubits with ***uniform state***.



Below is the complete circuit implementation. Remember, the input to the circuit is quits with uniform state.




```
begin
    GroversSearchCircuit = chain(3, put(1:3=>Uf), put(1:3=>Uφ), Measure(3, locs=1:3))
    plot(GroversSearchCircuit)
```

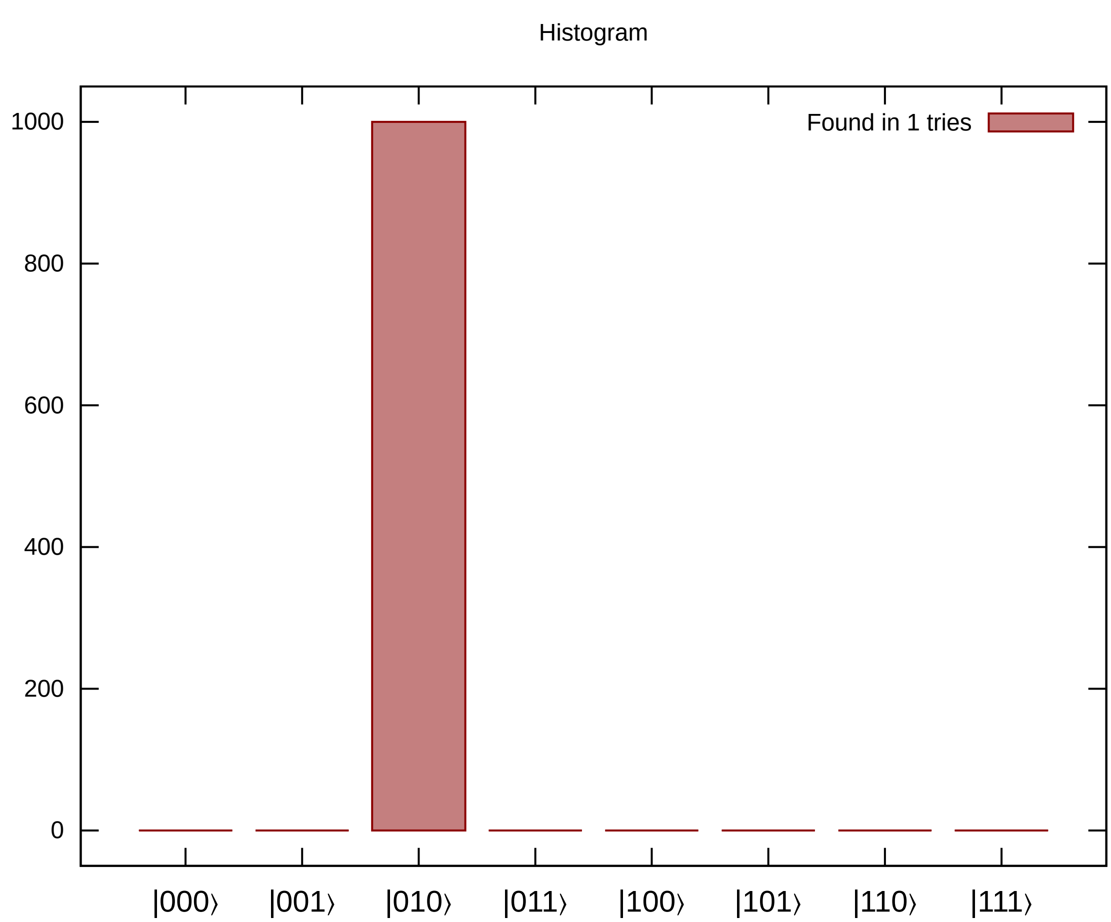
```
end
```

The below function plots the measurement function, just pass the measured values to it. You don't need to know its inner mechanisms to use it.

```
plotmeasure (generic function with 2 methods)
```

```
begin
    using BitBasis: BitStr
    using StatsBase: fit, Histogram
    using Gaston: bar, set, Axes
    set(showable="svg")
    function plotmeasure(x::Array{BitStr{n,Int},1}, s = "#") where n
        set(preamble="set xtics font ',$(n<=3 ? 15 : 15/(2^(n-3)))'")
        hist = fit(Histogram, Int.(x), 0:2^n)
        bar(hist.edges[1][1:end-1], hist.weights, fc="'dark-red'", legend="'Found in
$s tries'", Axes(title = :Histogram, xtics = (0:(2^n-1), "|" .* string.(0:(2^n-1),
base=2, pad=n) .* "⟩")))
    end
end
```

Below is the implementation of the ***Grover's Search Algorithm*** to find $|010\rangle$.




```
begin
    n = 3
    output = 0
    j = 0
    for i in 1:(2^3)
```

```julia
        input = uniform_state(n)
        global output = input |> GroversSearchCircuit |> r->measure(r, nshots=1000)
        if(output[1] == bit"010") #Checking for |010⟩
            break
        end
        global j = i
    end
    plotmeasure(output, j+1)
end
```

You can keep running the above block of code, and you'll find that it takes a maximum of 4 tries to find the state $|010\rangle$, which would be the worst case. Most of the times, you can find it in the first try!

Which sounds about right.