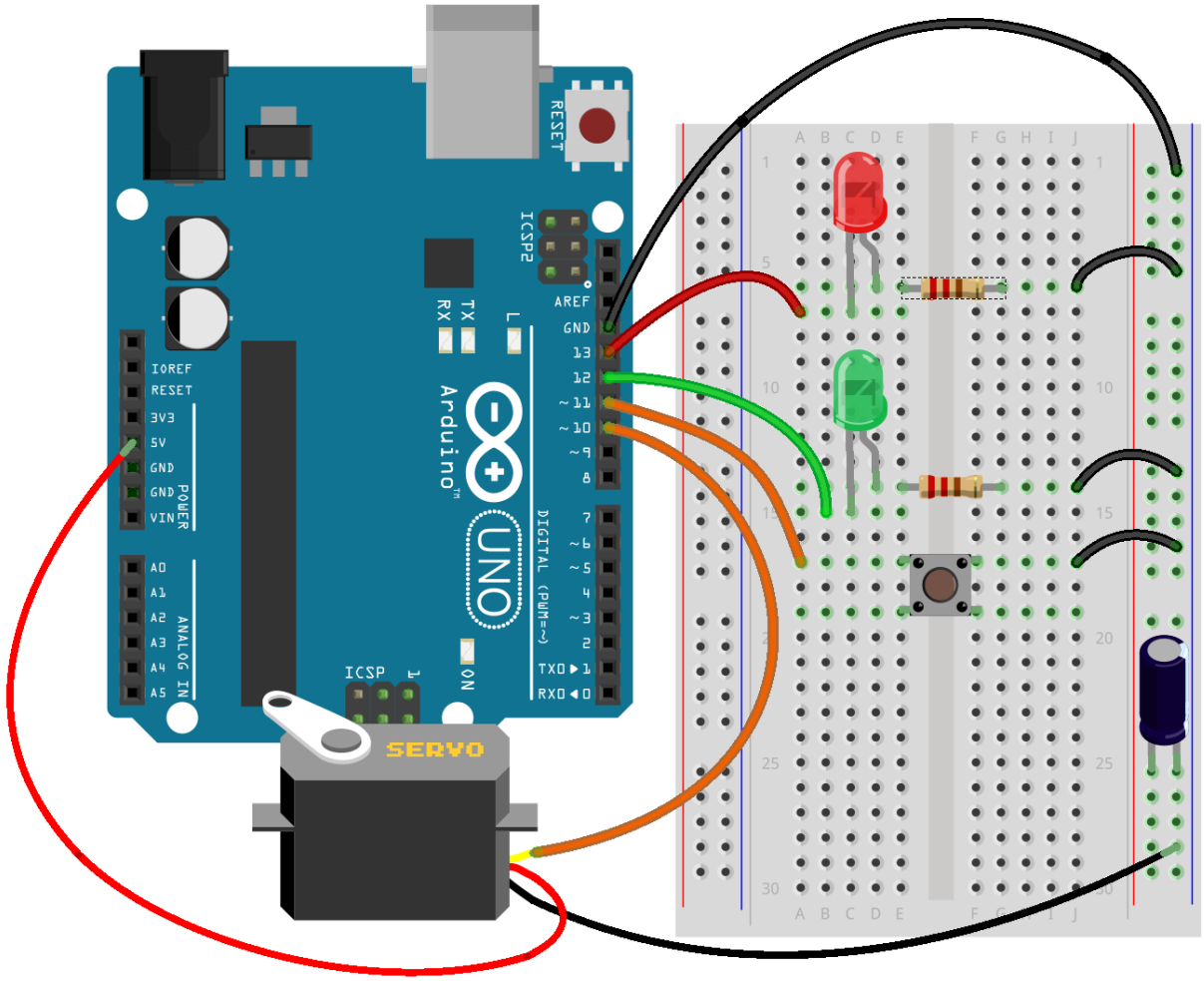
Truth or Dare Game



What this game will demonstrate:

- How to generate a (pseudo) random number
- Moving a servo between 0-180 degrees
- Lighting LEDs (digital outputs)
- Accepting digital inputs (button presses)
- Implementing a state machine
- Why we need capacitors

In this project we're creating a simple "truth or dare" type game. A bit like spin-the-bottle, the servo will choose a player at random and point at them. It will also light up one of two LEDs - a red one (truth) or a green one (dare).

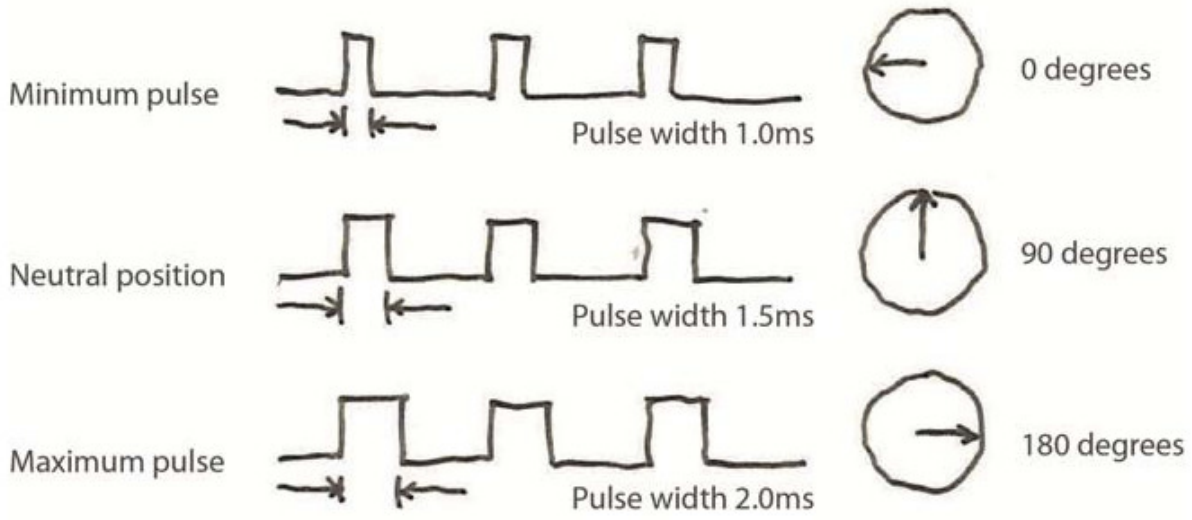
These kinds of games are usually associated with lots of alcohol - and a falling out between friends, usually around twenty minutes into the "game"!

Rather than go upsetting your friends by telling them that actually, yes, you do think they have a big nose really, we're going to use this game to look at some useful components and microcontroller techniques, which can be carried over into over - hopefully more useful - projects.

We've already looked at digital inputs and outputs (LEDs and pushbuttons) so let's jump straight in and look at the component that's at the heart of this game- the finger-pointing servo.

A servo is a special kind of motor. It has some clever gubbins inside it - and usually a few gears to - which stops it from spinning round and around at high speed (like most motors do) and allows us to make it turn only a tiny amount at a time. The servo has three wires - power and ground (to allow the motor to spin) and a third wire, where all the magic happens.

A servo can usually turn between 0 and 180 degrees (half a full revolution).
To get it to rest at it's centre-point, at 90 degrees, we hold the signal/data line high for 1.5ms (that's milliseconds - thousandths of a second). Any less than this, and the servo arm turns towards the 0 degrees position. Any more, and it turns towards the 180 degrees position. This signal on the data line is then repeated every 20ms (which worked out at 50 times a second - it's a good job these microcontrollers can work really really quickly!)



Luckily we don't actually have to worry about making all the tricky signals and generating such precise timing intervals - Arduino to the rescue!

There's a pre-built library for handling servos.
Just include it in the top of your code like this:

```
#include <Servo.h>
```

choose a digital i/o pin to connect to the data line of the servo:

```
int servo_pin=10;
```

we've gone with pin 10 but you could use any digital output pin.
Then create a servo object and give it a name.

```
Servo myservo;
```

You can use any name. Don't try to be too clever or witty here. It may be hilarious when asked to give something a name to call it "Bob" or "Dave". But once you've got a few things on the breadboard and you can't remember if "Geoff" was the name of your servo, or a stepper motor or the multi-line character LCD display, the novelty wears off. Try to choose sensible names for your objects. We called ours "myservo".

The last thing to do is to "attach" the servo to the pin, in code:

```
myservo.attach(servo_pin);
```

That's it. Your servo is ready to use, in your Arduino code-world.
Thanks to the magic of libraries, which take care of all the complex stuff for us, we can get the servo to move by using a single line of code:

```
myservo.write(servo_angle);
```

where servo_angle is a number between 0 and 180, which tells the servo how many degrees to turn through. And you thought this was going to be tricky huh?

So how to tell the Arduino where to point the servo?
We're using a random number generator. It's part of the Arduino language and every time we call the

```
random (1,11)
```

function, it chooses a random number between 1 and 10 (because of the way the internal stuff works, the random function will never actually pick the number 11. It might get to 10.99999 but never 11 - and because all the numbers get *rounded down* to the previous whole number, it never actually returns 11).

Now strictly speaking, it's not a genuinely random number.
It's *pseudo-random*.
The Arduino uses a clever algorithm to choose the next number in a great big long sequence of numbers - and the algorithm simply generates numbers which look like they've been randomly generated. But, because they're calculated, it's quite possible that the same sequence of numbers can come up again. And again. And again.

So while the list of numbers *appears* random, it's actually quite possible to predict the next number, if you're familiar with the sequence that the Arduino generated last time (or the time before that).

So we use what's called a *random seed.*
This basically tells the Arduino not to start from the first number in the sequence, but from the 50th number. Or the 83rd number. Or something like that. And to start the random "list" off at a different point, we use

```
randomSeed(analogRead(0));
```

What this does is take a reading from a floating (empty/not-connected) analogue input pin, and use it to generate a number, to tell the Arduino where in the list to start reading the "random" numbers from.
Now this really is random. Because the analogue pin is reading data from its immediate environment (rather than from a list of pre-calculated ones) it could have a different value every time it's read. So it we use a genuinely random event to choose a different starting point in our "random" list of numbers, every time we run the program.

And this helps give a true feeling of randomness to the `random` function (when really we know it's actually a bit of a kludge!).

Before we get bogged down with state machines, let's take a quick look at this capacitor thingy on the board.

What is a capacitor? What does it do? And what the blazes is it doing on my breadboard for goodness sake?

In this instance, the capacitor is acting like a tiny battery.
It just sits across the power and ground rails, charging up.
It's a bit like holding the special-move combo buttons in your favourite fighting game

Nothing actually happens while it's charging. It's just storing up energy for when it's needed. And when you've something big and lumpy on your circuit, like a motor - or a servo - sometimes a capacitor is needed.

Imagine you've just turned your servo on from a standing start. That can take a bit of juice to get going. And in some instances, it takes enough power to create a tiny little ripple on the power supply voltage rail. Not much. And not for very long. But sometimes - just sometimes - that ripple can be enough to cause the Arduino to reset. So we use the capacitor to store charge, just as we store up our special-move-combo. And then when the voltage dips, it's like releasing all the buttons on your gamepad

Ker-pow!
When power is removed from a capacitor, all the stored up charge is released.
So when our servo demands a lot of power from the voltage supply rail, and causes it to dip slightly, the capacitor *discharges* and releases it's stored up voltage.

It may not be as violent as a super-power-up-combo-attack but the principle is the same - while it's charging, nothing much happens. When it's released, it lets the stored up energy out. And in our case, it releases it's charge just when the voltage supply dips, so the Arduino doesn't get starved of power, and doesn't reset.

It's like the capacitor helps smooth out any ripples in the supply voltage.
And, when used in this way, the capacitors are often referred to - not surprisingly - as *smoothing capacitors* (and sometimes, *decoupling capacitors*, but that doesn't really work with this analogy, so we'll pretend we didn't hear that name for now).

So that's all the hardware taken care of.
We've already discussed inputs and outputs earlier on.
We've touched on servos, how they work, and how we write code to control them.
We looked at how they can affect the rest of the circuit when they're activated - and what we can do to reduce this effect, thanks to the smoothing influence of capacitors.

All that's left to do is bang some code into the Arduino, and see it all in action!