Tight Genes:

Intro to Genetic Algorithms

by Dave Aronson
T.Rex-2023@Codosaur.us
twitter.com/DaveAronson
linkedin.com/in/DaveAronson
github.com/CodosaurusLLC/tight-genes

Tight Genes:

Intro to Genetic Algorithms

by Dave Aronson
T.Rex-2023@Codosaur.us
twitter.com/DaveAronson
linkedin.com/in/DaveAronson
github.com/CodosaurusLLC/tight-genes

Speaker notes

Γεια σας, Αθήνα!

(Hello, Athens!)

Speaker notes

Ya Sas, AhTHEEna!

Είμαι ο Dave Aronson,

(I'm Dave Aronson,)

Speaker notes

eeMAY o Dave Aronson,

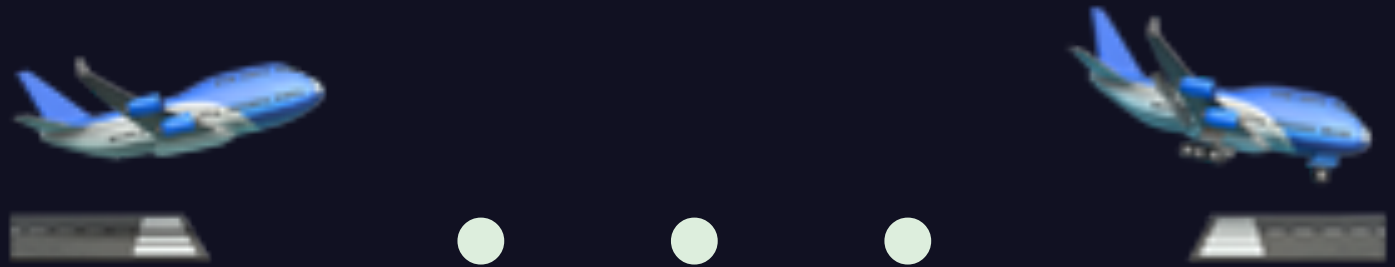
ο T. Rex του Codosaurus,

(the T. Rex of Codosaurus,)

Speaker notes

o T. Rex tu Codosaurus,

και πέταξα εδώ



(and I flew here)

Speaker notes

kay paytaxAH ehDOH

με το κατοικίδιο μου πτεροδάκτυλο

(on my pet pterodactyl)

Images: https://pixabay.com/vectors/dinosaur-tyrannosaurus-t-rex-6273164/
and https://pixabay.com/vectors/bird-flying-wings-dinosaur-ancient-44859/

Speaker notes

meh to katoiKIdio moh pteroDAKtilo

για να σας μάθω τους

(to teach you about)

Speaker notes

yeh na Sas MAtho tohss

Γενετικούς Αλγόριθμους.

(Genetic Algorithms.)

Image: https://pixabay.com/vectors/genetic-testing-gene-panel-genetics-2316642

Speaker notes

yenetiKOHSS alGOrithmohss.

Αλλά . . .

(But . . .)

Speaker notes

AHlah . . .

θα το κάνω στα αγγλικά.

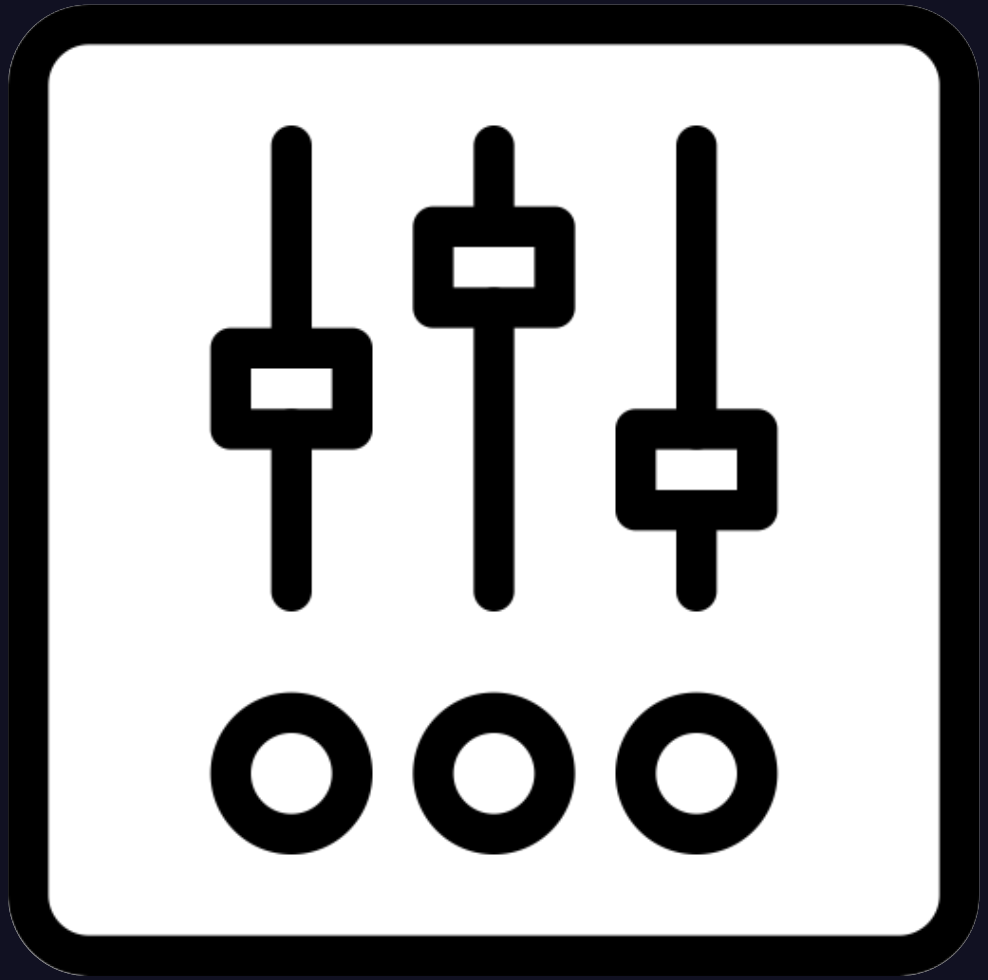
(I will do it in English.)

Speaker notes

Tha toh KAno sta AngliKAH.

Mainly because you've just heard almost all the Greek I speak!

So what are genetic algorithms? They are . . .



Speaker notes

optimization heuristics,

WAT?!

Speaker notes

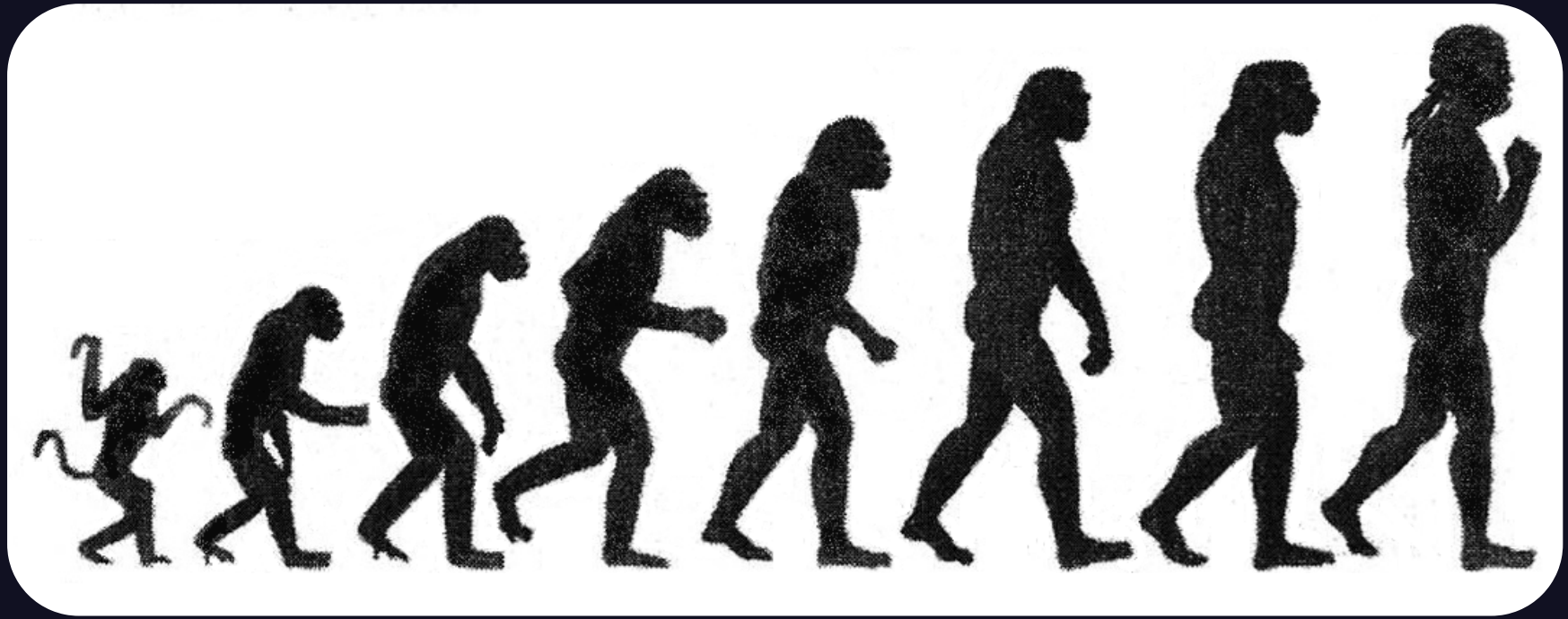
which is fancy-talk for . . .

Optimization Heuristic:
shortcut to find
"good enough" solutions
(ideally the best,
but OK if not).

Speaker notes

. . . shortcuts to find solutions to a problem, ideally the best, but we usually have to settle for something "good enough", due to constraints like time or money.

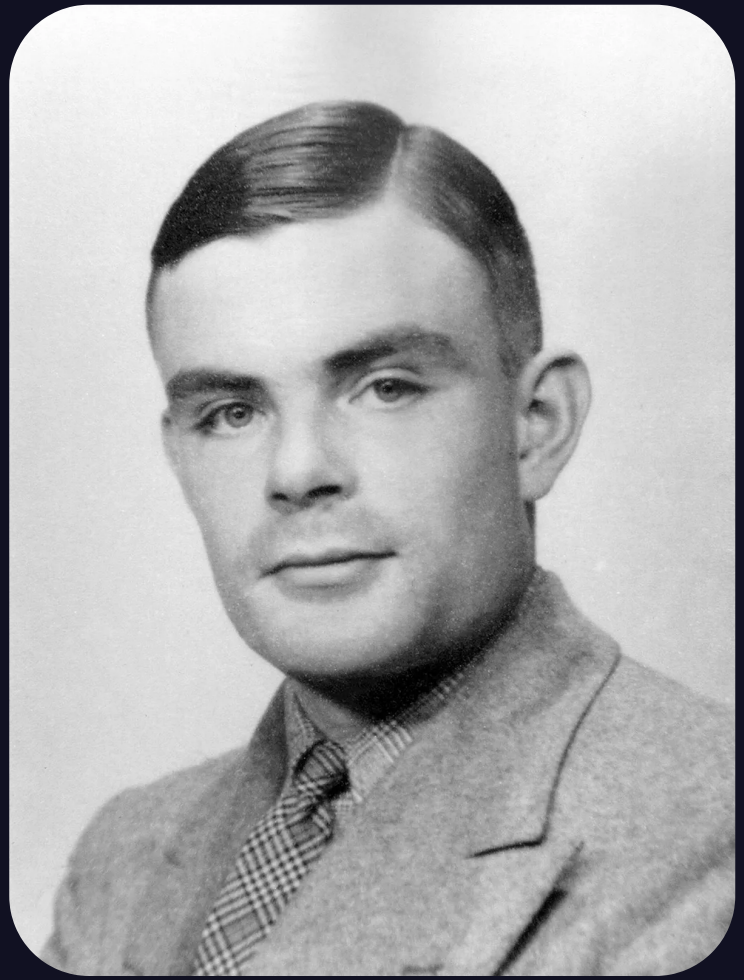
There are many kinds of optimization heuristics, but genetic algorithms are uniquely inspired by . . .



Speaker notes

. . . real-world biological evolution, mainly the principles of survival of the fittest, random combination of old sets of genes (no, not like I'm wearing) into one new set, and random mutation.

The history goes back to 1950, when . . .



Alan Turing

Image: https://cdn.britannica.com/81/191581-050-8C0A8CD3/Alan-Turing.jpg

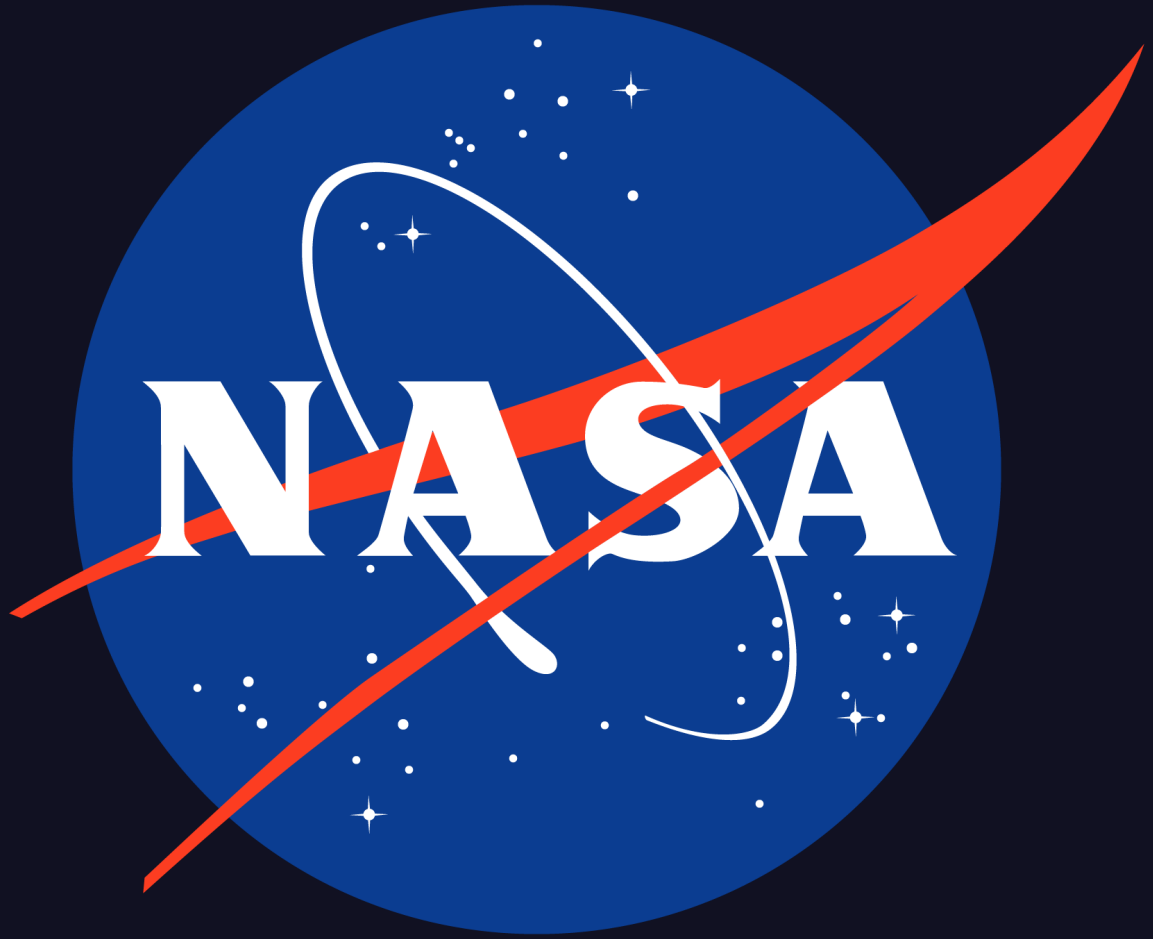
Speaker notes

. . . Alan Turing, as in Turing Test, Turing Machine, and so on, proposed a "learning machine" in which the mechanism of learning would be similar to evolution. Nothing much came of that, and it took a few decades for genetic algorithms in general to get some traction. The first commercial product based on genetic algorithms, a mainframe toolkit for . . .

Speaker notes

. . . industrial processes, came out in the late 1980's, from General Electric. These days, MATLAB and such tools have some genetic algorithm facilities built-in, and many programming languages have genetic algorithm libraries available. However, the actual uses of genetic algorithms remain mostly boring and obscure, used by companies in their internal industrial processes, logistics, scheduling, and so on.

But once in a while, they get used for something more interesting, and more publicly known. Most famously, in 2005 . . .


NASA

Speaker notes

. . . NASA used a genetic algorithm to design an . . .

ST5 antenna, and US quarter for scale

Image: https://www.jpl.nasa.gov/nmp/st5/IMAGES/st5-antenna.jpg

Speaker notes

. . . antenna for the ST5 series of satellites. (No, that's not just a paperclip bent up by someone fidgeting in a boring meeting.) The NASA Jet Propulsion Laboratory website says: "Its unusual shape is expected because most human antenna designers would never think of such a design." And that is one of the great advantages of this approach.

So how do genetic algorithms work? They consist of a simple series of steps:

Initialize

Speaker notes

First, we create an initial population of candidates. In Genetic Algorithm terms, these are called "chromosomes", but since most living beings contain many chromosomes in each and every cell, I don't like that term, I think it leads to confusion, so I'm just going to say "candidates".

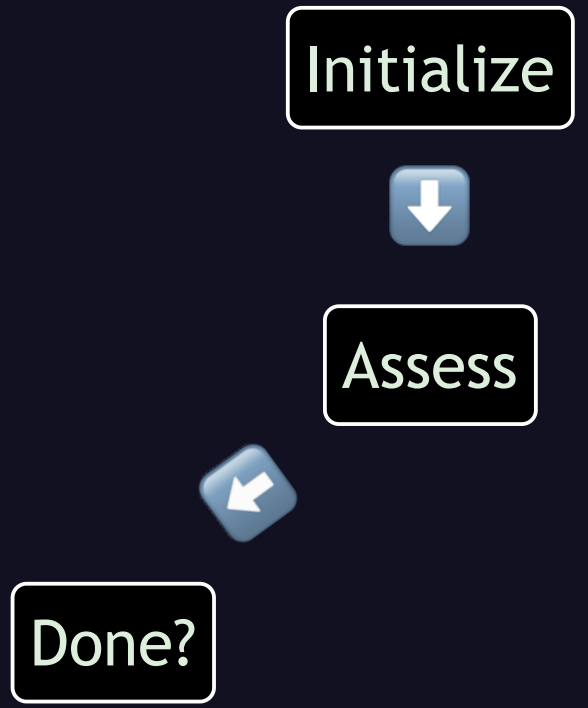
The next step is to . . .

Initialize

Assess

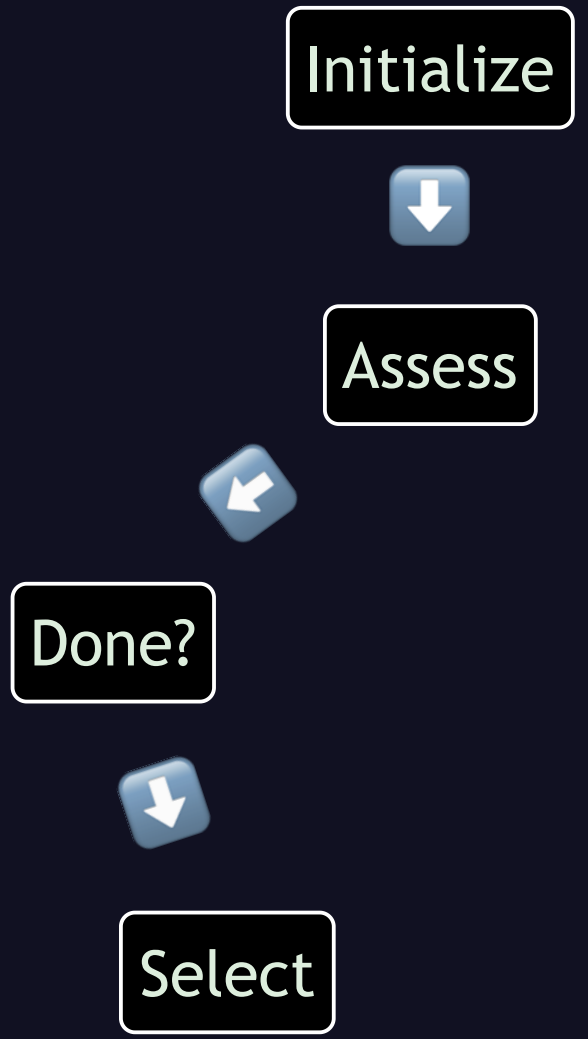
Speaker notes

. . . assess the "fitness" of each candidate, according to whatever criteria we want to apply. We do it here mainly because it supplies the data usually used in the next step, which is to ask, are we . . .


Initialize
Assess
Done?

Speaker notes

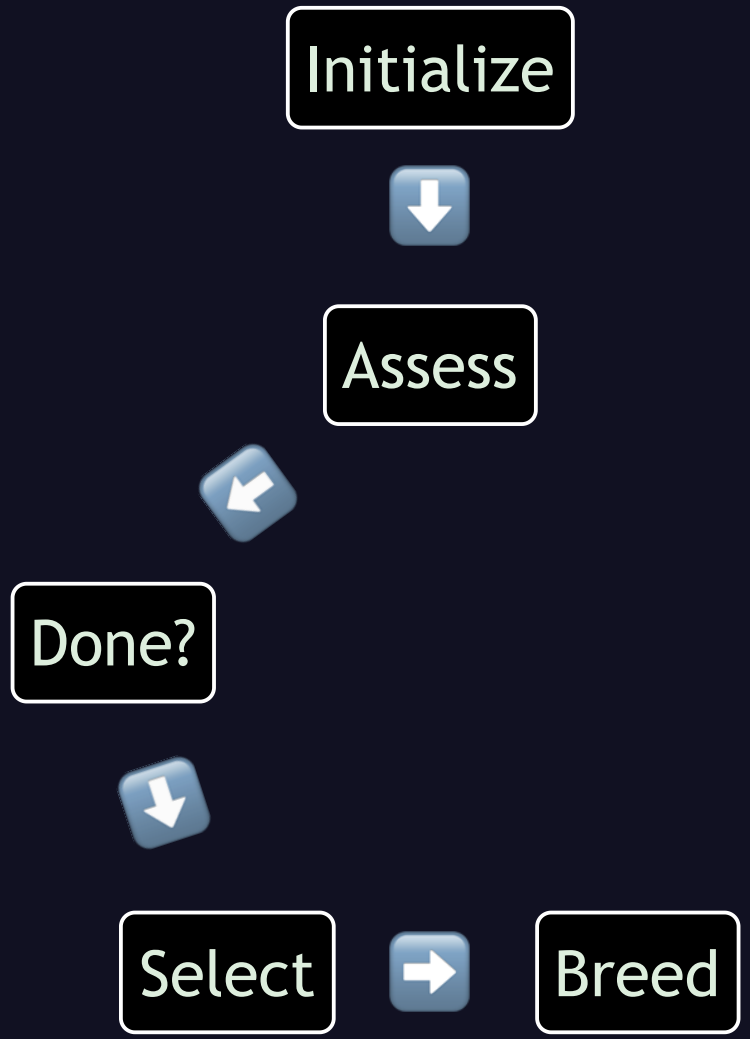
. . . done yet? This is usually based on the fitness, but could be based on other criteria, or a combination. If we're not done, then next we . . .


Initialize
Assess
Done?
Select

Speaker notes

. . . select some candidates to breed the next generation. This is also usually based on the fitness, to simulate survival of the fittest.

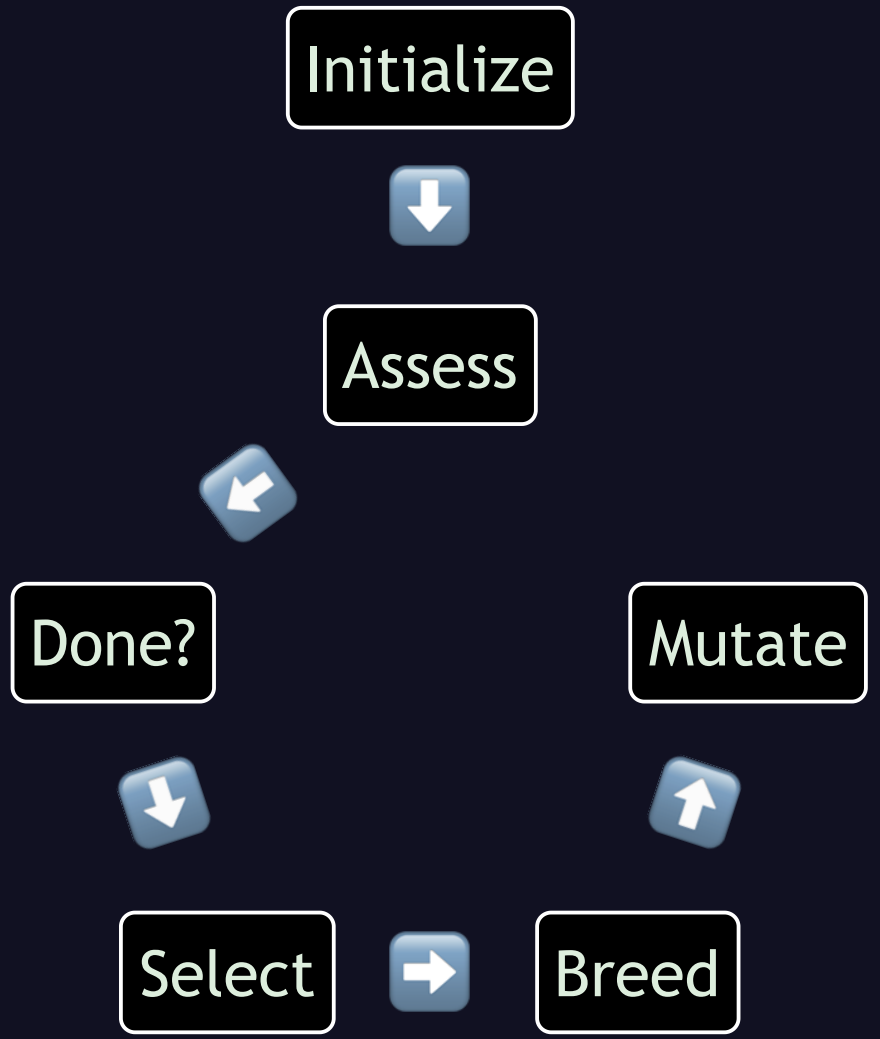
After that, we use those candidates we just selected . . .


Initialize
Assess
Done?
Select
Breed

Speaker notes

. . . to breed a new population.

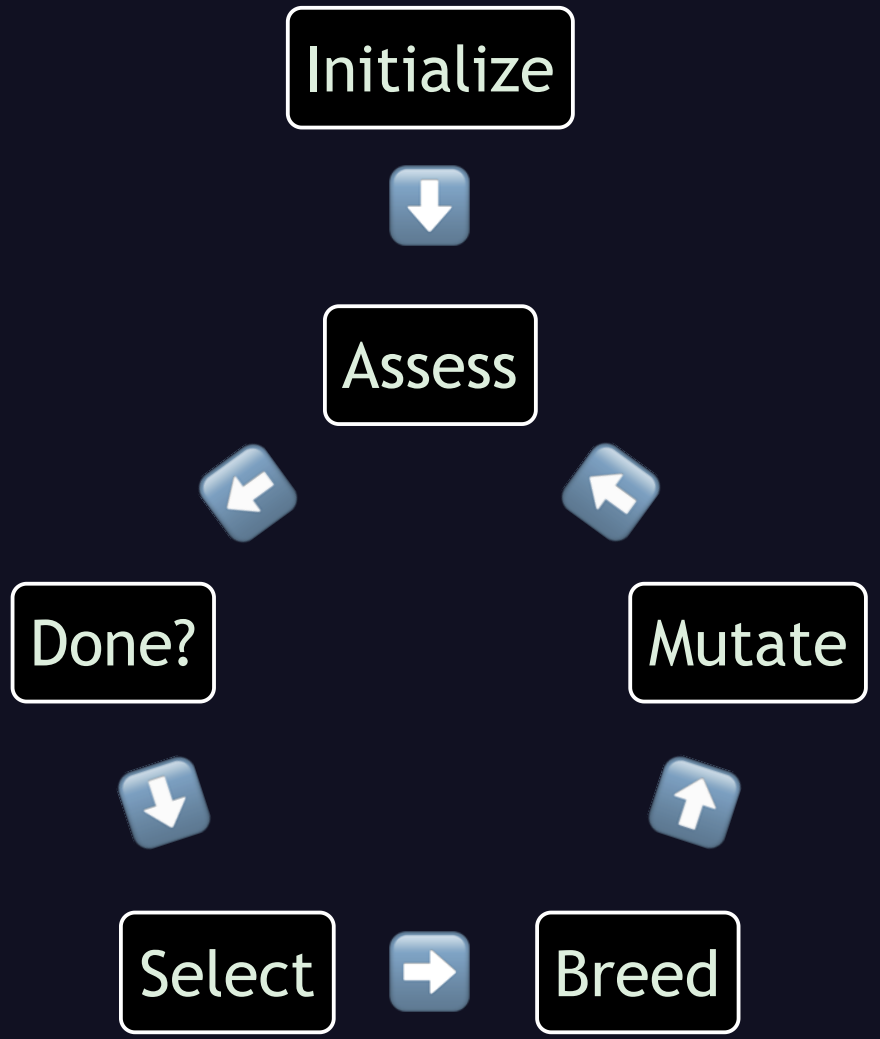
Next we . . .


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

. . . mutate those new candidates, for more diversity in the gene pool.

Finally, we . . .


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

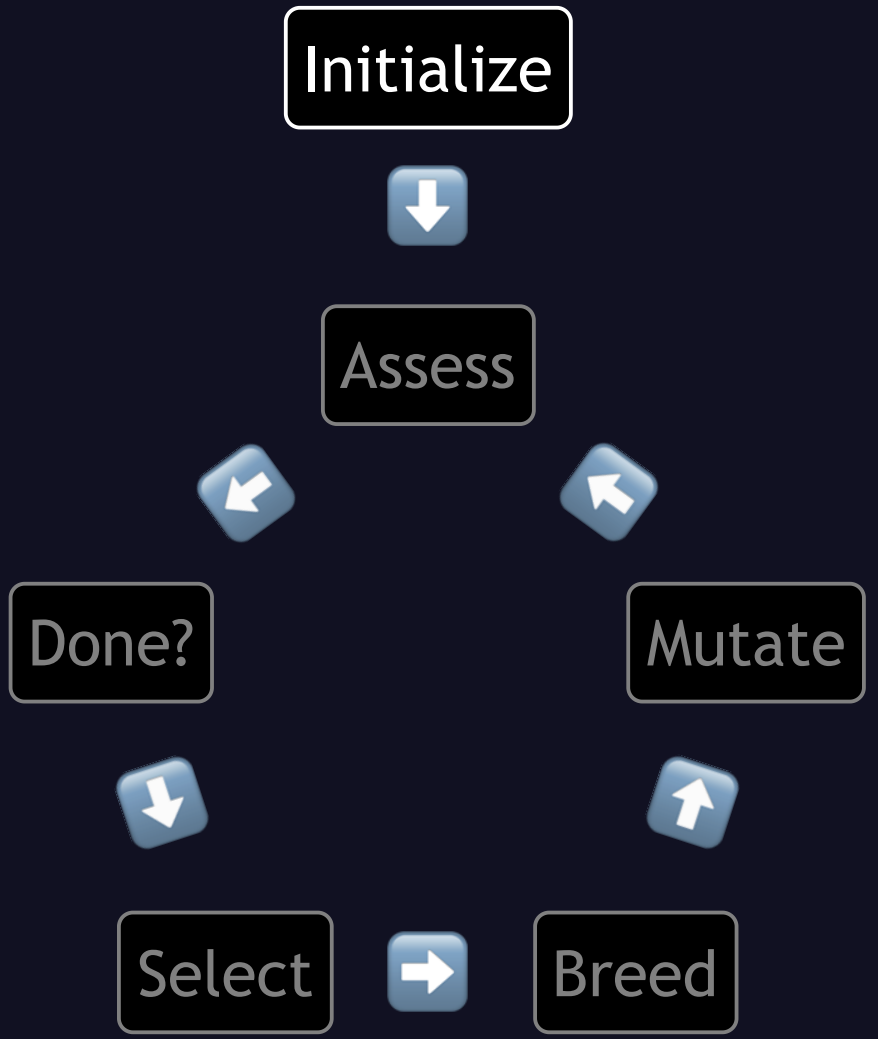
. . . go back to step 2, assessing their fitness. This sequence could be represented at a high level with some rather simple code, like so:

```
how_many = 10  # or however big we want
pop = initial_pop(how_many)
evaluate(pop)
while not done?(pop)
  breeders = select_breeders(pop)
  pop = breed(breeders, how_many)
  mutate(pop)
  evaluate(pop)
end
```

Speaker notes

This is in Ruby, which I chose because it reads so close to plain English, so even if you don't know Ruby, I'm confident you'll understand the ideas.

Now let's take a closer look at each step, by working through an example.


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

First we create an initial population of candidates. But what is a candidate, and how do we create one? These are different solutions to some problem, usually represented as different instances of the same data structure. The simplest common type of candidate is . . .

01001000
01100101
01101100
01101100
01101111
00100000
01110111
01101111
01110010
01101100
01100100
00100001

Speaker notes

. . . a simple string of bits. This will do fine for candidates that consist of a simple series of yes/no decisions. This may sound simplistic, but there is a huge class of problems that boil down to this, called . . .

Knapsack / Rucksack / Backpack / Whatever!

Image: https://publicdomainvectors.org/en/free-clipart/Rucksack-vector-image/9910.html

Speaker notes

. . . knapsack problems. The canonical example is that you have a knapsack, and many things you want to carry in it, but they won't all fit, so you want to find the combination of items, that will fit, and has the maximum value. To look at a concrete example, suppose we know . . .

Speaker notes

. . . a farmer, with a smallish truck, and he needs to decide what to take to market each week. And on this farm he has . . .

Speaker notes

. . . some cows. (E-I-E-I-O!) So among the things he can take to market are:







Speaker notes

. . . cows, milk, cheese, butter, ice cream, meat, and leather. His truck has room to take all the items, but it can only carry so much weight, so that's our constraint. His choices are as follows:

What	Unit	Qty	Pounds	Value
Cow	cow	1	1,500	$2,000
Milk	1-gal jug	200	1,720	$800
Cheese	5-lb wheel	200	1,000	$12,000
Butter	1-lb block	1,000	1,000	$3,000
Ice Cream	1-gal tubs	200	1,000	$2,000
Meat	side	4	1,280	$8,000
Leather	hide	20	1,100	$6,000
TOTAL WEIGHT:			8,600	

Speaker notes

You don't need to remember all that, just notice that it totals 8,600 pounds. But, his truck's suspension can only handle two tons, or 4,000 pounds. Let's see what happens if we use a genetic algorithm to determine a "good enough" truckload. First we need a way to represent each candidate. In code, we could represent them as a class, and create one randomly, like so:

```
class Truckload
  def initialize()
    @contents = rand(128)
  end
end
```

Speaker notes

we're making a random number with seven bits, so we have a random 1 or 0 for each of our seven possible items. We could get as complex as we want in this function, like dictating a minimum or maximum number of items, but let's keep it simple.

To create an initial population, we can just create a bunch of candidates and stuff them into an array, . . .

```ruby
def self.initial_population(how_many)
  population = []
  for i in 1..how_many
    population.append(self.new)
  end
  return population
end
```

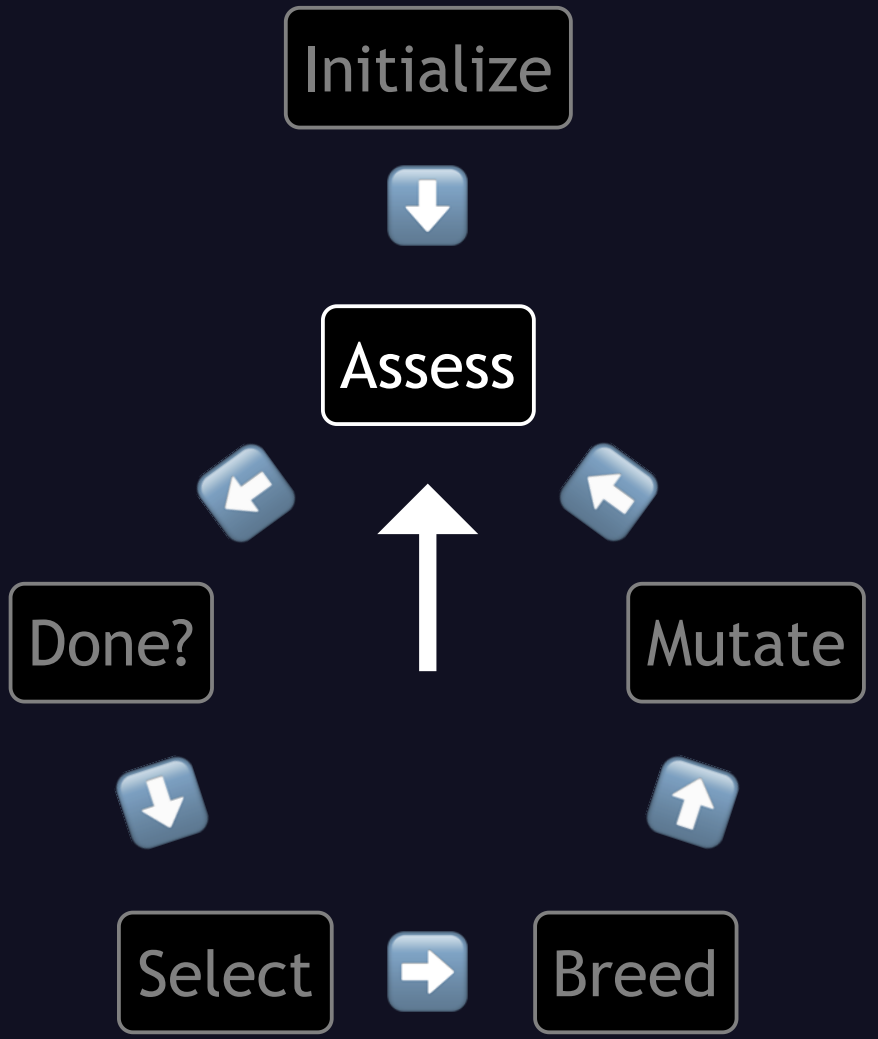
Speaker notes

. . . like so. (This could actually be done in much more idiomatic Ruby, so don't scold me for that, I'm just trying to keep it easily understandable by people who don't know Ruby.) So if we create a population of ten Truckloads, we might wind up with a list like this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather
Y	N	N	Y	N	Y	Y
N	N	N	Y	Y	N	N
N	Y	N	N	N	Y	N
N	Y	Y	N	Y	N	N
Y	Y	Y	N	Y	Y	N
Y	Y	N	Y	N	N	N
Y	N	N	Y	N	Y	N
Y	Y	N	N	N	N	N
N	N	Y	Y	Y	Y	Y
N	N	Y	N	N	Y	N

Speaker notes

We can get from random numbers to those combinations, by iterating over the bits and seeing which are turned on, but in the interests of time, I'll handwave over those details. Next, we . . .


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

. . . assess how "fit" each of these truckloads is. We do this with what's called a "fitness function". (Surprise!) In this case, we already know we want the total value, BUT, any load that's too heavy for the truck, is worthless. In Ruby, that would look like this:

```ruby
def fitness()
  items = ITEMS.
    with_index.
    select { |_itm, idx| bit_on?(idx, @contents) }.
    map { |item, _idx| itm }
  weight = items.map { |item| itm.weight }.sum
  return 0 if weight > 4000
  return items.map { |item| itm.value }.sum
end
```

Speaker notes

We decode which items we want to take, then sum up their weights. If that exceeds the truck's capacity, we return zero, else we sum up their values.

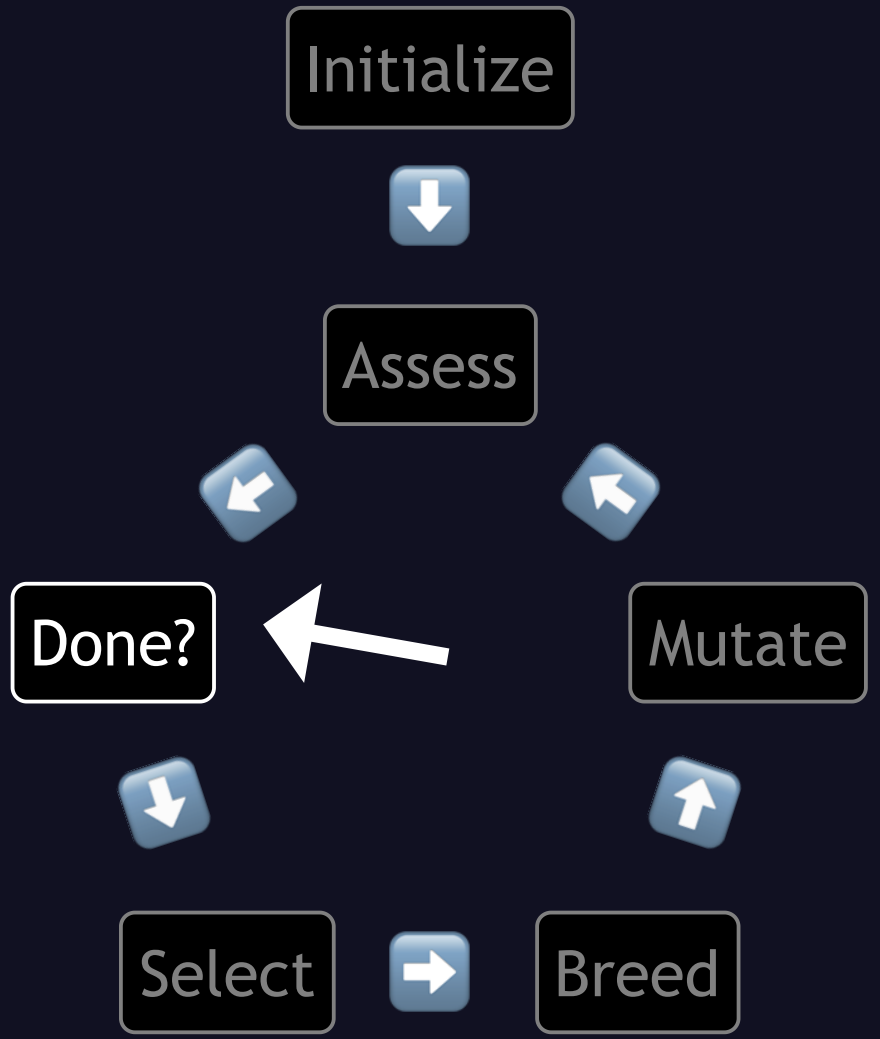
Again, we could get as complex as we want in this function, and NASA's antenna fitness function certainly must have been.

If we run this fitness function on our population, and sort on fitness descending, we get this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather	Fitness
N	N	Y	N	N	Y	N	20,000
N	Y	Y	N	Y	N	N	14,800
Y	N	N	Y	N	Y	N	13,000
N	Y	N	N	N	Y	N	8,800
N	N	N	Y	Y	N	N	5,000
Y	Y	N	N	N	N	N	2,800
Y	N	N	Y	N	Y	Y	0
Y	Y	Y	N	Y	Y	N	0
Y	Y	N	Y	N	N	N	0
N	N	Y	Y	Y	Y	Y	0

Speaker notes

So now that we've assessed their fitness, we . . .


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

. . . check if we're done. So what are our criteria? The function can be simple, but it can take some thinking to figure out what the function should do. With a knapsack problem, a good solution, especially the best, can be made totally worthless by adding just one more . . .

Speaker notes

. . . waffer-theen item, and thereby exceeding the constraints. So, we're going to record the best we've seen, and stop if we don't see anything better within 100 generations.

In Ruby, that would look like this:

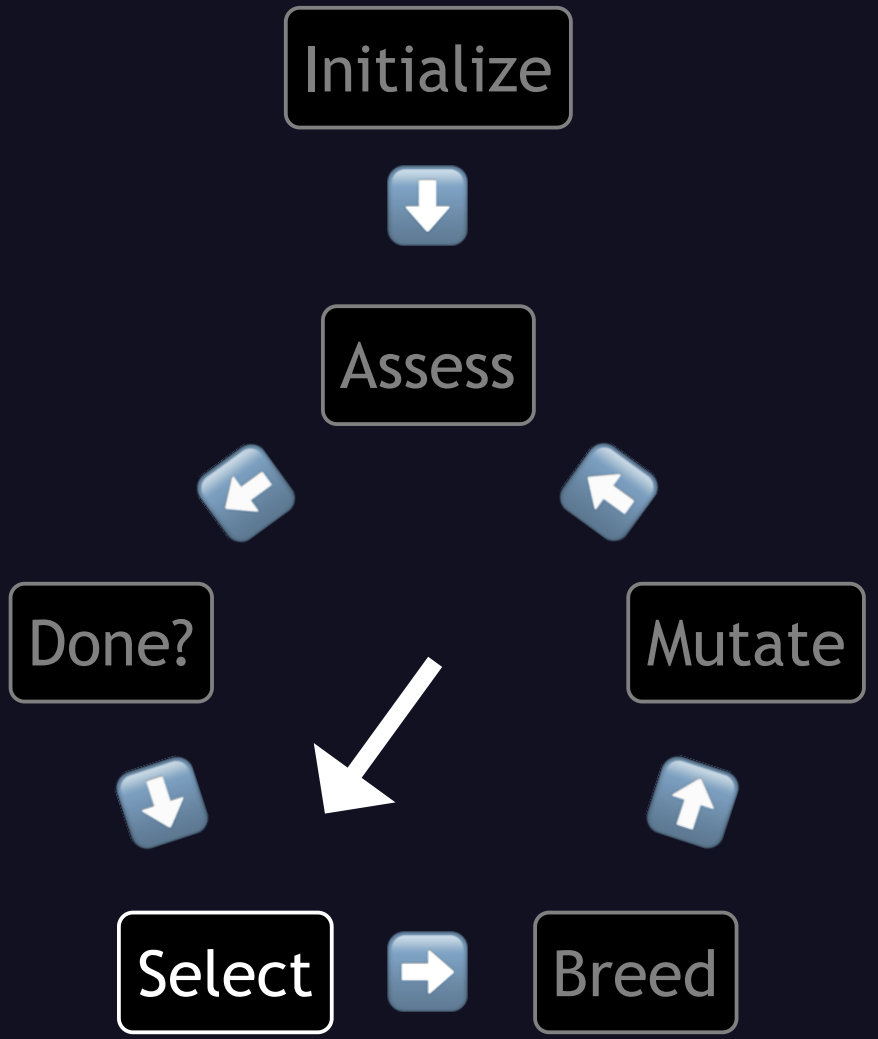
```ruby
@@best_combo = self.new(0)
@@generations = 0

def self.done?(population)
  @@generations += 1
  better = population.
    select { |c| c.fitness > @@best_combo.fitness }
  if better.any?
    @@best_combo = better.sort_by(&:fitness).last
    @@generations = 0
    return false
  else
    return @@generations >= 100
  end
end
```

Speaker notes

When this code is initially parsed, we set the initial best combo as empty, and we set how many generations it's been since we saw that, as zero. When the function is called, we increment the number of generations, look at the fitness of the current candidates, and select the ones with a better fitness than our benchmark, the best one so far. If there are any better candidates, we make the fittest one our new benchmark, reset the generation counter, and return false. Else if it's been 100 generations since the best one, we return true, else we return false.

Again, we can get as complex as we want, not only in checking the maximum fitness, but we could look at other stopping criteria, like achieving some specific level of maximum fitness, some maximum number of generations, or amount of time, or let the user click a STOP button, or many other ways, or a combination of ways.

Since we're not done, the next step is to . . .


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

. . . select some candidates to breed the next generation. The obvious way is to take the top two most fit, like so:

```ruby
def self.select_breeders(population)
  return population.
    sort_by(&:fitness).
    reverse.
    take(2)
end
```

Speaker notes

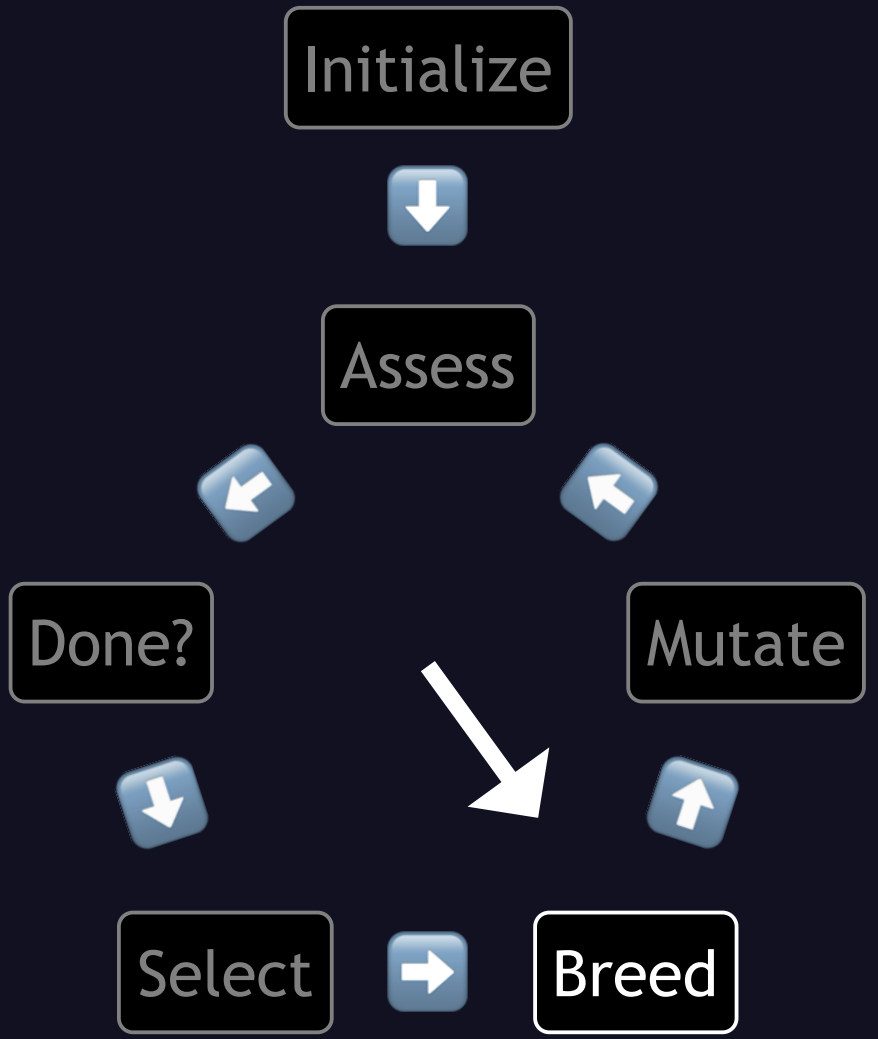
We take the population, sort them by fitness in descending order, and take the first two. Out of our current population, we would choose:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather	Fitness
N	N	Y	N	N	Y	N	20,000
N	Y	Y	N	Y	N	N	14,800

Speaker notes

these two. As usual, we also could get more complicated, and there are some common more complex alternatives. For instance, with "Roulette Wheel" selection, we would select the candidates randomly, but each one has a chance to be selected, proportional to their fitness.

Now that we've chosen our breeders, next we . . .


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

. . . breed them. The usual way is called crossover. This consists of taking the data points, or in Genetic Algorithm terms, the "genes", from one parent, up to some randomly chosen crossover point, then switching to the other parent, like so:

```
def self.combine(p1, p2)
  cross_point = rand(ITEMS.count + 1)
  list = (0..ITEMS.count).
    map { |index|
      parent = index < cross_point ? p1 : p2
      parent.contents & (1 << index)
    }.
    sum
  return self.new(list)
end
```

Speaker notes

We establish the crossover point for each new candidate, as a random number between zero and how many items there are, inclusive. Then we iterate through the list of items. If we haven't yet hit the crossover point, we get the decision for that item from the first parent, else we get it from the other parent. If we do this once, with a crossover point of 3, that would get us a result like this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather
N	N	Y	N	N	Y	N
+						
N	Y	Y	N	Y	N	N
=						
N	N	Y	N	Y	N	N

Speaker notes

But this is just one of ten results, because we're making a whole new population, like so:

```
def self.new_population(p1, p2, how_many)
  population = []
  for i in 1..how_many
    population.append(self.breed(p1, p2))
  end
  return population
end
```

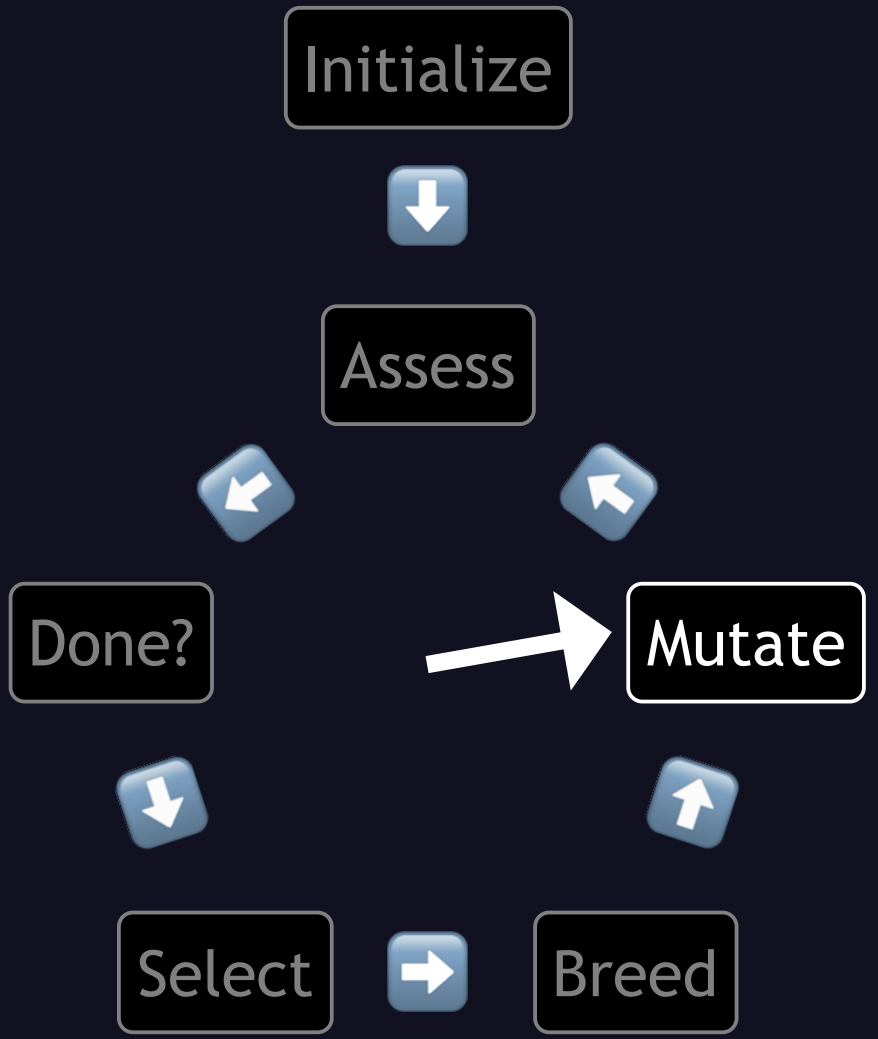
Speaker notes

This is just like how we created the initial population, except that instead of each candidate being made from scratch, they're the product of breeding our chosen breeders. The whole list might look like this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather
N	N	Y	N	Y	N	N
N	Y	Y	N	Y	N	N
N	N	Y	N	Y	N	N
N	N	Y	N	N	Y	N
N	N	Y	N	Y	N	N
N	N	Y	N	Y	N	N
N	N	Y	N	N	N	N
N	N	Y	N	Y	N	N
N	N	Y	N	Y	N	N
N	N	Y	N	N	Y	N

Speaker notes

Lots of family resemblance there, eh? None of these loads include a cow, butter, or leather, and they all include cheese. That's because both of our two breeders were like that. If we were to just continue breeding the fittest of each generation, we wouldn't ever see any loads including a cow, butter, leather, or no cheese, but we fix that in the next step, which is to . . .


Initialize
Assess
Mutate
Done?
Select
Breed

Speaker notes

. . . mutate them. Again, I'm going to keep it very simple, and give each gene a 1 in 4 chance of flipping. In code, that looks like this:

```ruby
def maybe_mutate()
  (0..ITEMS.count).each do |index|
    if rand(4) == 0
      @contents ^= (1 << index)
    end
  end
end
```

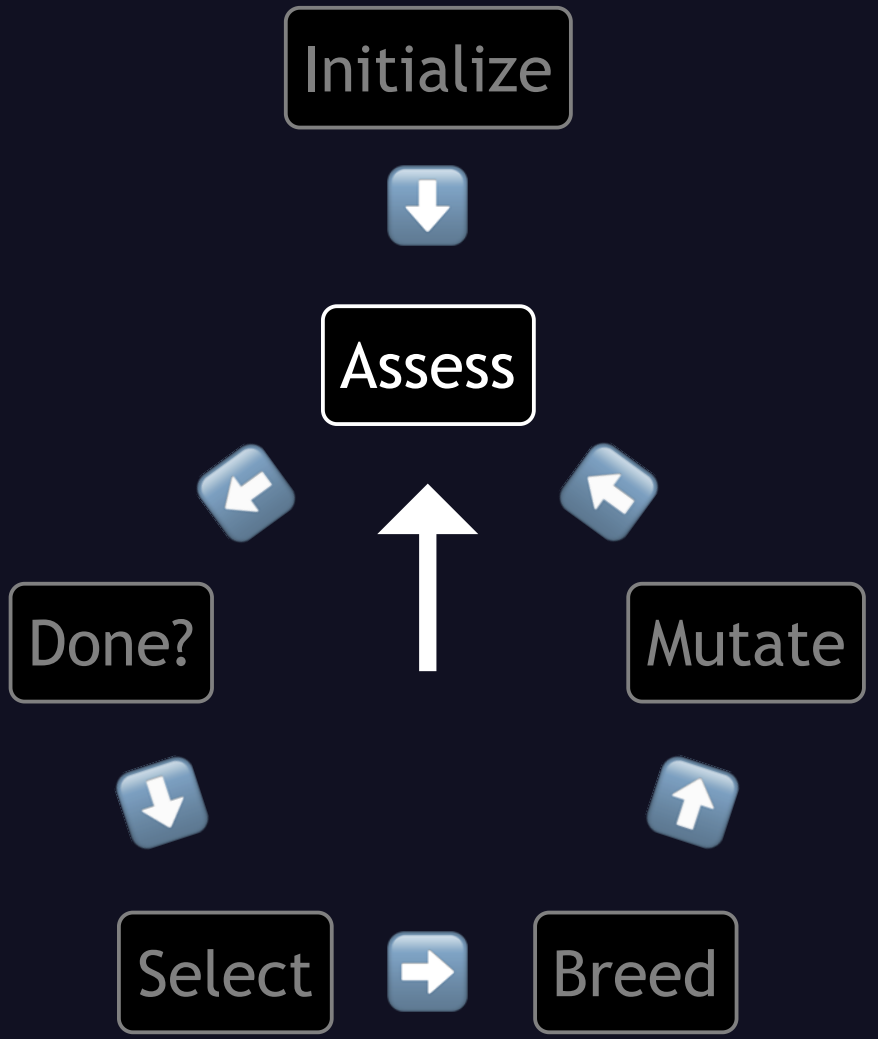
Speaker notes

We iterate through the item numbers, and for each one, if a random number from zero to three is a zero, we flip that bit. Again, we could get as complex as we want, like having some genes more likely to mutate than others, or having some minimum or maximum number of mutations per candidate, or all kinds of other options. If we run this mutation function on these new candidates, we might wind up with something like this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather
N	Y	N	Y	N	N	Y
N	N	Y	Y	N	N	N
Y	N	Y	Y	Y	Y	N
Y	Y	Y	Y	Y	Y	N
Y	Y	Y	Y	N	N	Y
N	Y	N	Y	N	N	Y
Y	Y	N	Y	N	N	Y
N	N	Y	N	Y	N	N
Y	N	N	Y	Y	N	N
N	N	Y	N	Y	N	N

Speaker notes

. . . where green means that it changed. You can see that we now DO have some truckloads that include a cow, butter, or leather, or no cheese. Now we go back to . . .


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

. . . assessing the fitness of these new candidates, and we get this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather	Fitness
N	N	Y	Y	N	N	N	15,000
N	N	Y	N	Y	N	N	14,000
N	N	Y	N	Y	N	N	14,000
N	Y	N	Y	N	N	Y	9,800
N	Y	N	Y	N	N	Y	9,800
Y	N	N	Y	Y	N	N	7,000
Y	Y	Y	Y	N	N	Y	0
Y	Y	Y	Y	Y	Y	N	0
Y	N	Y	Y	Y	Y	N	0
Y	Y	N	Y	N	N	Y	0

Speaker notes

Oh noes! Our maximum fitness actually went down! As you may recall, our previous best one scored 20,000. But don't worry, as you may recall from our "are we done yet" function, we hang onto the best one, and just try to outdo it, so we haven't lost it. The next generation might look like this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather	Fitness
N	Y	Y	N	N	Y	N	20,800
N	N	Y	N	N	Y	N	20,000
N	N	Y	N	N	Y	N	20,000
N	N	N	N	N	Y	Y	14,000
N	N	Y	N	N	N	N	12,000
N	Y	N	N	Y	N	N	2,800
Y	N	Y	Y	N	Y	N	0
Y	Y	Y	N	Y	Y	N	0
N	Y	Y	N	Y	Y	N	0
N	Y	Y	Y	N	Y	N	0

Speaker notes

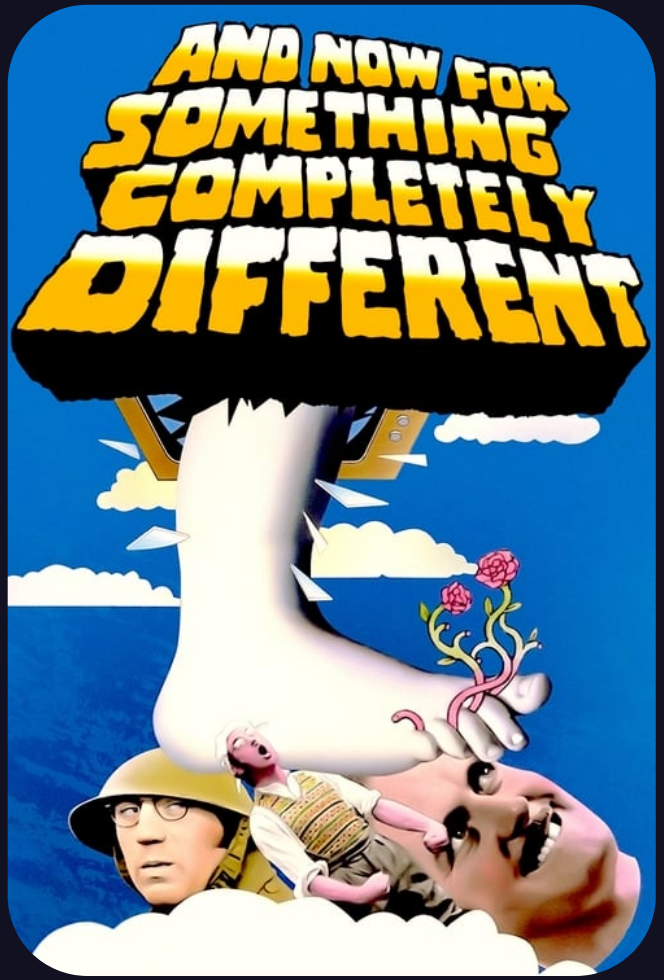
. . . a small improvement over our prior best! So, we set that top one as our benchmark, and reset the counter of generations since we saw it. If we let this run to completion, we might wind up with something like this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather	Fitness
N	N	Y	N	N	Y	Y	26,000
N	N	Y	Y	N	N	Y	21,000
N	Y	N	N	Y	Y	N	10,800
Y	N	N	N	N	N	Y	8,000
N	N	N	N	N	Y	N	8,000
N	N	N	Y	Y	N	N	5,000
N	Y	N	N	N	N	N	8,00
N	Y	N	N	N	N	N	8,00
Y	Y	N	Y	N	N	Y	0
Y	Y	N	Y	Y	Y	Y	0

Speaker notes

. . . with our best truckload scoring 26,000, made up of cheese, meat, and leather. So that's one complete run of a genetic algorithm. If we wanted to check whether that was the best that this algorithm could produce, we could just run it again, as many times as we like, within reason, since it's so much faster than brute force. Okay, maybe writing all this code is not so much faster when we've only got seven items, and such simple criteria, but if we had to choose among many more items, with more complex criteria, for many truckloads a day, creating a genetic algorithm might well be worthwhile.

Now, suppose we want to evolve . . .


AND NOW FOR
SOMETHING
COMPLETELY
DIFFERENT

Speaker notes

. . . something completely different. Suppose we want to "evolve" a good set of stats for a Dungeons and Dragons fighter character, so our candidates are tuples of numbers, rather than strings of bits. D&D character stats are . . .

STRength
INTelligence
DEXterity
CONstitution
WISdom
CHArisma

3d6 each
ignoring STR 18/xx

Speaker notes

. . . Strength, Intelligence, Dexterity, Constitution, Wisdom, and Charisma, each determined by rolling three six-sided dice, or 3d6 for short. (I'm going to gloss over how you can sometimes have extra strength.) In Ruby that might look like this:

```
class Character
  def initialize()
    @str = roll(3, 6)
    @int = roll(3, 6)
    @dex = roll(3, 6)
    @con = roll(3, 6)
    @wis = roll(3, 6)
    @cha = roll(3, 6)
  end
```

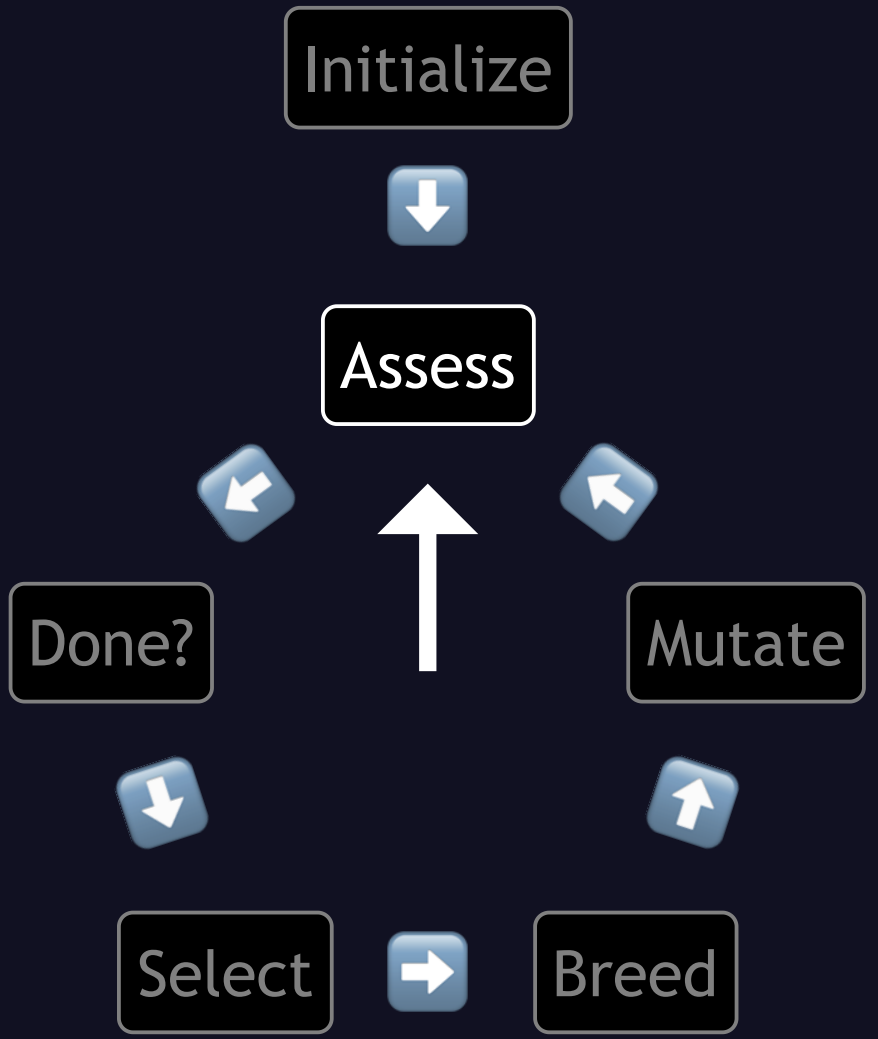
Speaker notes

So if we create an initial population of ten Characters, it might look like this:

Str	Int	Dex	Con	Wis	Cha
11	9	9	10	7	15
4	14	8	12	13	10
9	14	15	11	9	16
14	15	10	7	6	14
13	12	7	13	11	10
12	12	10	9	5	16
11	12	9	13	6	12
10	14	12	8	8	16
14	7	8	9	8	8
14	12	13	5	13	13

Speaker notes

The next step is . . .


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

. . . to assess how "fit" each one of these characters is. We're trying to evolve a good set of Fighter stats, so it should be based mainly on strength, constitution, and dexterity. Intelligence, wisdom, and charisma, not so much, but we don't want them too low, for the sake of occasional saving throws. I tried several different things, such as . . .

```
def fitness()
  str * 2 + con + dex / 2
end
```

Speaker notes

totaling up double the strength, the constitution, and half the dexterity. But, the other stats tended to get too low, and even the dexterity. So I tried . . .

```ruby
def fitness()
  stats = [str, con, dex, int, wis, cha]
  (0..5).
    map { |idx|
      stats[idx] * (6 - idx)
    }.
    sum
end
```

Speaker notes

prioritizing them linearly, adding up six times the strength, five times the constitution, and so on down to one times the charisma. But then the other stats got too high, and the characters seemed too generalized. So I finally settled on this:

```ruby
def fitness()
  stats = [str, con, dex, int, wis, cha]
  (0..5).
    map { |idx|
      stats[idx] * 2 ** (5 - idx)
    }.
    sum
end
```

Speaker notes

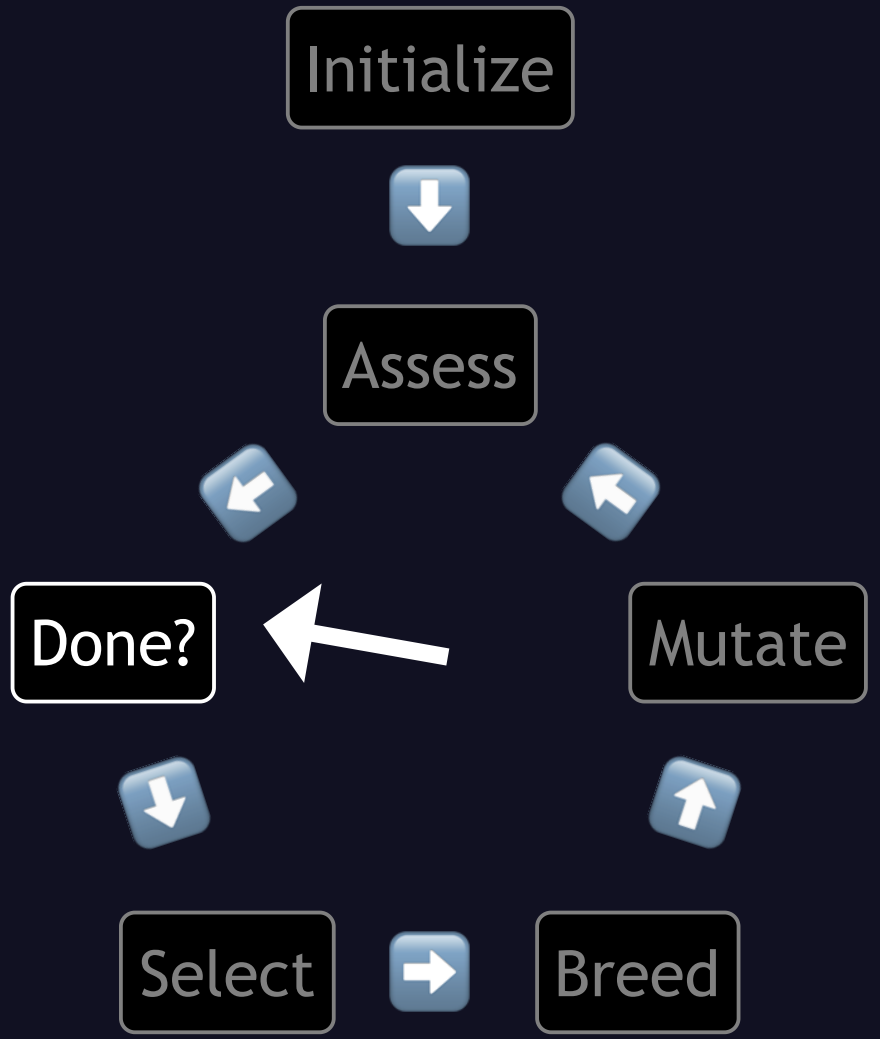
. . . prioritizing the stats again but much more strongly, totaling up 32 times the strength, 16 times the constitution, and so on down to one times the charisma. Here we see that even though the fitness function itself can be very simple, it can be difficult to figure out one that will yield good results, whatever that means in the situation at hand.

If we run this on our population, and sort them, we get this:

Str	Int	Dex	Con	Wis	Cha	Fit
13	12	7	13	11	10	760
14	15	10	7	6	14	726
14	12	13	5	13	13	719
14	7	8	9	8	8	708
11	12	9	13	6	12	703
12	12	10	9	5	16	682
9	14	15	11	9	16	674
11	9	9	10	7	15	649
10	14	12	8	8	16	632
4	14	8	12	13	10	476

Speaker notes

Now that we've assessed their fitness, we can ask, are we . . .


Initialize
Assess
Mutate
Breed
Select
Done?

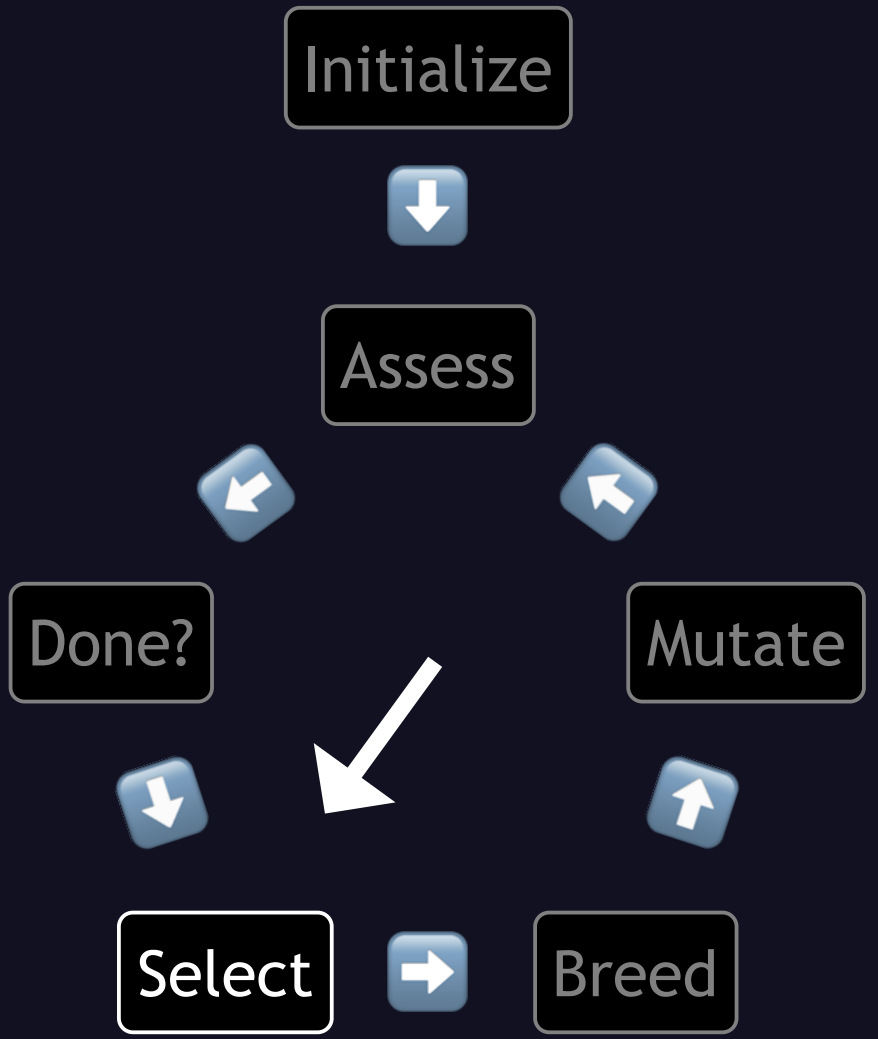
Speaker notes

. . . done? What are our criteria? Let's say we're done if any candidates get 90% of the way to the maximum score of our fitness function. I'll spare you the math, but that would be 1,021. In code, checking that would look like this:

```ruby
def Character.done?(population)
  population.any? { |cand|
    cand.fitness >= 2021
  }
end
```

Speaker notes

Very simple. None of our current candidates score anywhere near 1021, so we . . .


Initialize
Assess
Mutate
Breed
Select
Done?

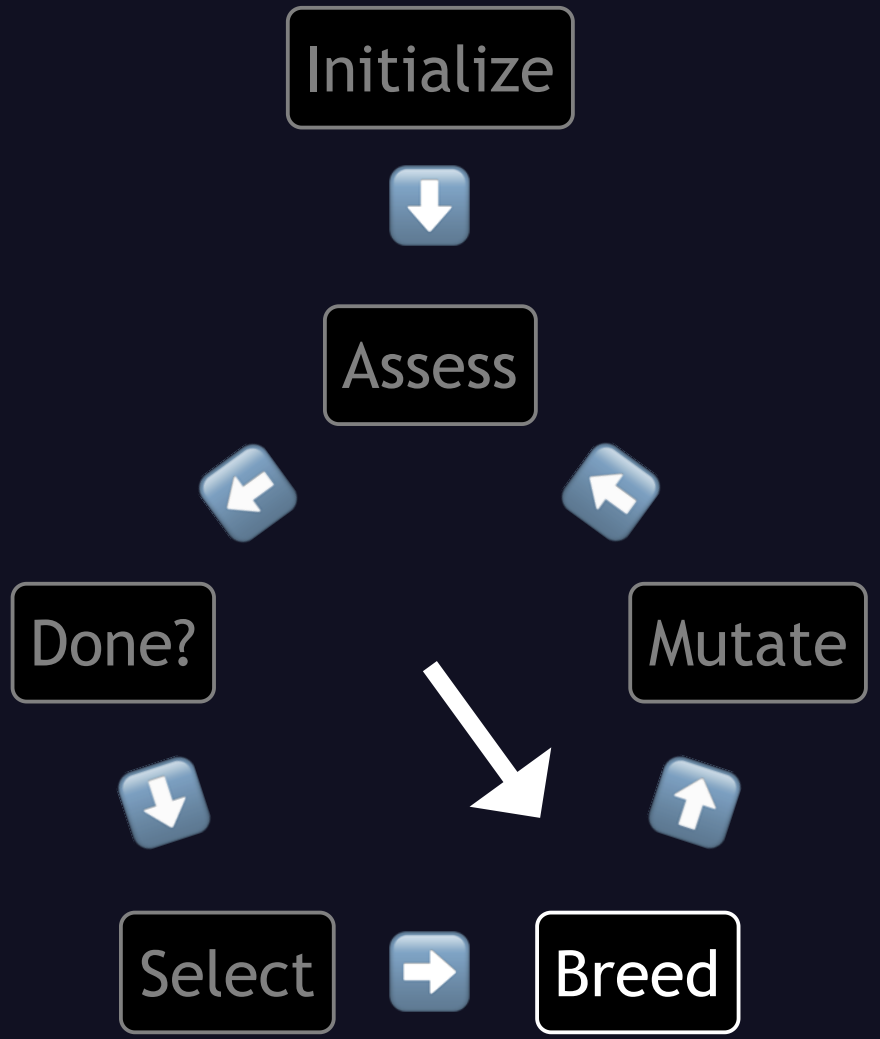
Speaker notes

. . . select some candidates to breed the next generation. Taking the top two scorers again we get:

Str	Int	Dex	Con	Wis	Cha	Fit
13	12	7	13	11	10	760
14	15	10	7	6	14	726

Speaker notes

these two. And then of course we actually . . .


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

. . . breed our chosen pair, this time using another common strategy, of essentially flipping a coin for each gene, like so:

```
def Character.breed(p1, p2)
  char = Character.new
  char.str = rand(2) == 1 ? p1.str : p2.str
  char.int = rand(2) == 1 ? p1.int : p2.str
  char.dex = rand(2) == 1 ? p1.dex : p2.str
  char.con = rand(2) == 1 ? p1.con : p2.str
  char.wis = rand(2) == 1 ? p1.wis : p2.str
  char.cha = rand(2) == 1 ? p1.cha : p2.str
end
```

Speaker notes

We go through the stats one by one, flip a coin (or "roll a d2"), and if it comes up 1, we get that stat from the first parent, else we get it from the other parent. That could get us a result like this:

Str	Int	Dex	Con	Wis	Cha
13	12	7	13	11	10
		+			
14	15	10	7	6	14
		=			
13	15	10	13	6	10

Speaker notes

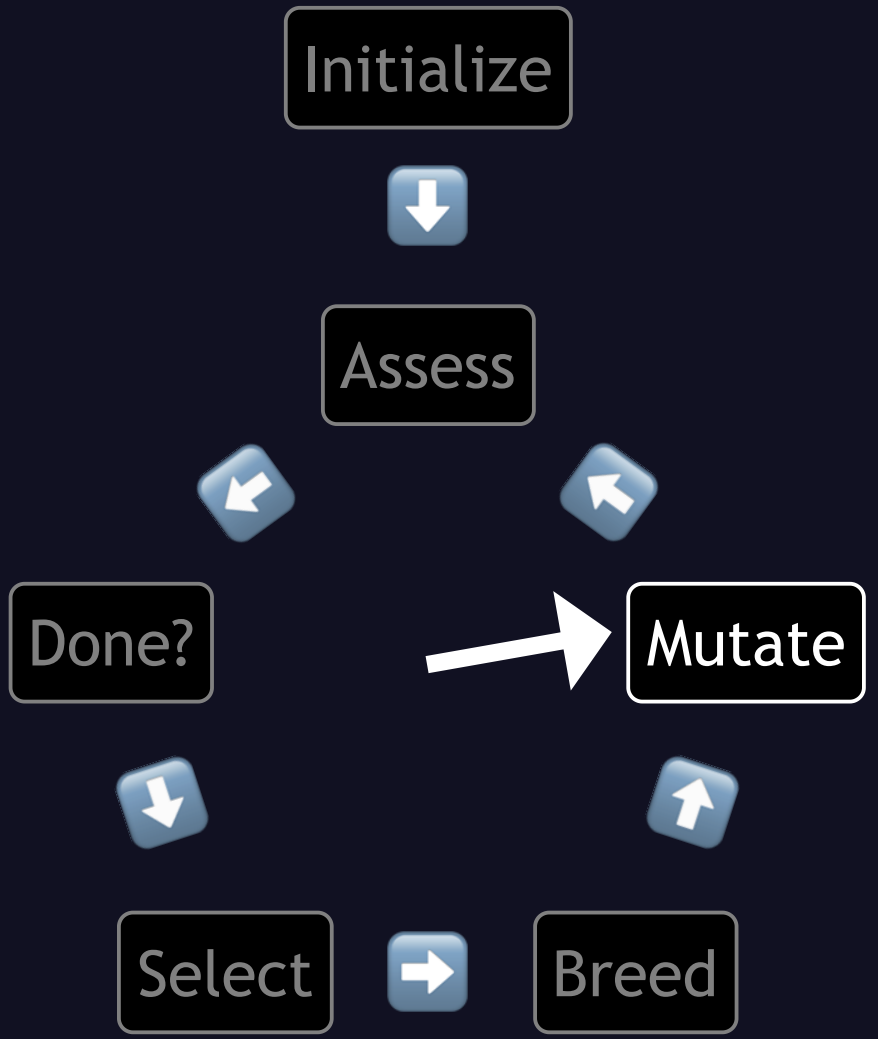
But again, this is just one of ten results, because we're making a whole new population, which might look something like this:

Str	Int	Dex	Con	Wis	Cha
13	12	10	7	6	14
13	12	7	13	6	14
14	12	10	13	11	14
14	15	7	13	6	14
13	12	10	13	6	14
14	15	10	7	11	10
14	12	10	13	6	10
13	15	10	13	6	14
13	15	10	13	6	10
14	12	7	7	6	14

Speaker notes

There are two things to notice here. First, the number of red and green is not always the same, neither in a single candidate nor the whole population. It's a series of random coin flips, so on average there will be three and three, but just like with the Truckloads, it could be anything up to six and zero either way. Second, notice the family resemblance again! Even though we're no longer using yes/no decisions, for each stat, there are only two possible values, or in Genetic Algorithm terms, "alleles", for a total of 64 possible combinations. There would be only one possible value, and therefore half as many possible combinations, if any stats were the same between the parents.

At a glance, these look on average much more suitable as fighters than the previous generation. (We'll figure their actual fitness scores later.) Just like before, if we were to simply continue breeding the fittest of each generation, we wouldn't see any change, let alone improvement, in the possible values of each stat. For instance, the Wisdom would never be anything other than 6 or 11. But again, we fix that in the next step, which is to . . .


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

. . . mutate them. Again, I'm going to keep it very simple, and give each stat a 1/3 chance of staying the same, going up a point, or going down a point, within the valid range. In code, that could look like this:

```ruby
def maybe_mutate()
  @str = maybe_mutate_stat(@str)
  @int = maybe_mutate_stat(@int)
  @dex = maybe_mutate_stat(@dex)
  @con = maybe_mutate_stat(@con)
  @wis = maybe_mutate_stat(@wis)
  @cha = maybe_mutate_stat(@cha)
end

def maybe_mutate_stat(stat)
  (stat + rand(3) - 1).clamp(3, 18)
end
```

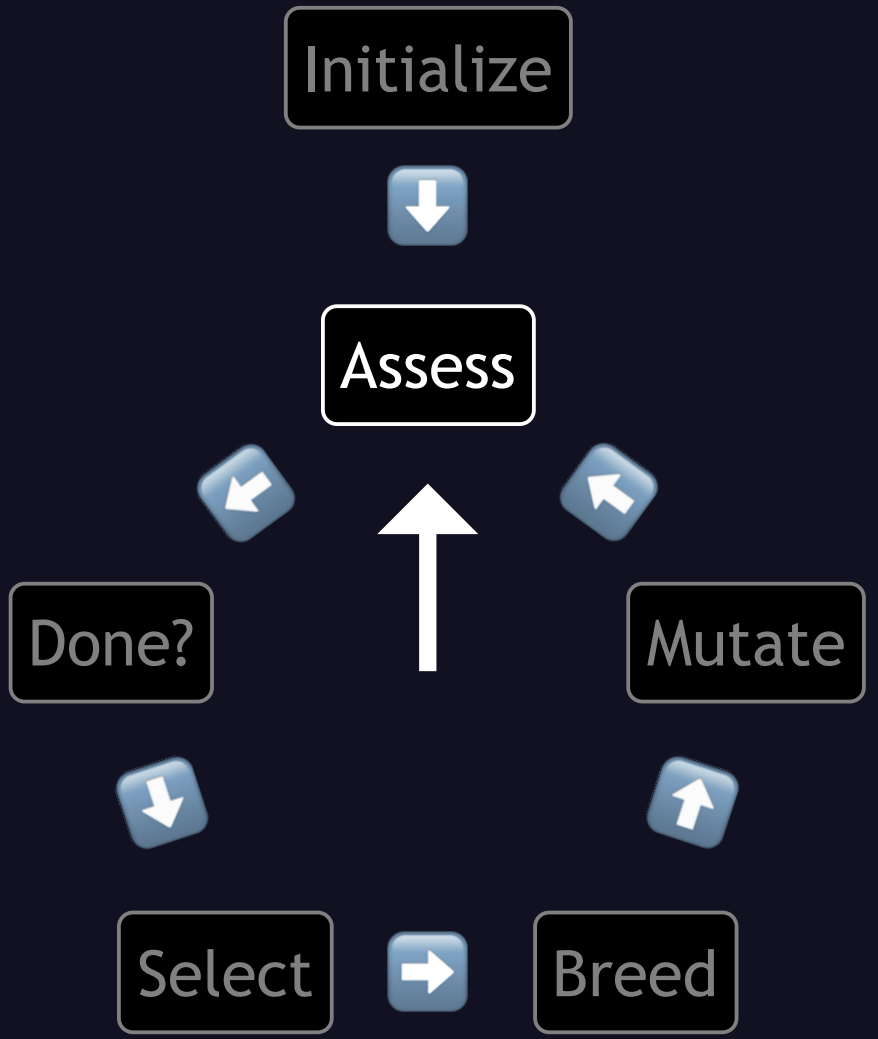
Speaker notes

For each stat, we add a random number from 0 to 2, and subtract one, which is like adding a random number from -1 to 1, but we clamp it to the range of 3 to 18. Again, we could get as complex as we want, like giving it a higher chance of going up or down, maybe by multiple points, if it's very low or very high, or many other options. If we run this on our new population, we wind up with something like this:

Str	Int	Dex	Con	Wis	Cha
14	12	10	7	7	13
12	13	7	14	6	14
13	12	10	14	11	15
15	16	7	14	6	15
13	11	10	12	5	15
13	15	11	7	11	10
14	13	9	12	6	9
14	15	9	13	6	15
13	15	11	12	5	9
13	12	6	6	7	13

Speaker notes

. . . where green means it went up, and red means down. Looking at the values in each column, you can see it's now much more diverse. Now we go back to . . .


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

. . . Step 2, and assess the fitness of these new candidates, which yields this result:

Str	Int	Dex	Con	Wis	Cha	Fit
15	16	7	14	6	15	851
14	15	9	13	6	15	815
13	12	10	14	11	15	805
14	13	9	12	6	9	785
13	15	11	12	5	9	775
13	11	10	12	5	15	757
12	13	7	14	6	14	742
14	12	10	7	7	13	715
13	15	11	7	11	10	708
13	12	6	6	7	13	635

Speaker notes

We can see that this generation is much improved from the prior one. The old one ranged from 476 to 760, and the new one from 635 to 851. It's still nowhere near our stopping criterion of 1021, so fast-forwarding through six more rounds, we finally get . . .

Str	Int	Dex	Con	Wis	Cha	Fit
18	**18**	**9**	**18**	**4**	**13**	**1029**
18	17	7	18	6	14	1014
18	16	8	18	3	13	1011
18	15	7	18	3	13	999
18	16	8	17	4	13	997
18	16	6	18	3	15	997
17	18	8	18	6	13	993
17	16	9	18	3	14	988
18	16	6	17	3	13	979
17	16	8	17	4	12	964

Speaker notes

. . . one suitable character, with 18 Strength and Constitution, acceptable though sub-par Dexterity, and surprisingly high Intelligence. That's not a problem, just a bit of a waste. If we really wanted it more specialized, we could complicate the fitness function further, and do things like explicitly demand well above average scores in the class's useful stats, and forbid it to be so high in the others, or at least apply a penalty.

So we've evolved a set of Fighter stats. Let's suppose we don't need any more Fighters in the party . . . but now we need a Wizard. All we need to do is tweak our fitness function, like so:

```ruby
def fitness()
  # below is the only line that changed!
  stats = [int, wis, dex, con, cha, str]
  (0..5).
    map { |idx| stats[idx] * 2 ** (5 - idx) }.
    sum
end
```

Speaker notes

. . . to prioritize intelligence first, then wisdom, dexterity, and so on, down to strength. An initial population would look roughly the same, since we haven't changed how that is generated, so I'll spare you those steps, but after 11 generations I got . . .

Str	Int	Dex	Con	Wis	Cha	Fit
12	**18**	**17**	**12**	**15**	**18**	**1048**
14	**18**	**16**	**9**	**15**	**16**	**1026**
15	**18**	**15**	**10**	**15**	**16**	**1023**
13	18	17	11	13	18	1013
14	18	17	10	13	18	1010
13	18	16	8	14	18	1009
12	17	16	11	15	17	1002
12	18	15	9	14	16	1000
14	17	16	10	15	16	998
14	17	17	11	13	18	982

Speaker notes

. . . three candidates 90% fit to be wizards. They're mostly pretty good in the other stats, but not so much as to be obviously better suited for some other class, except that that top one looks like a bard to me.

There are many other ways we can use genetic algorithms. They can create images, music, even code, and I'm now working on a system to schedule conference talks. So, think about it, and you might be able to use them for something.

To recap what you've learned here today:

Genetic Algorithms:

- are ~~optimization heuristics~~ shortcuts

Speaker notes

Genetic Algorithms are optimization heuristics, which is fancy-talk for shortcuts to finding good-enough solutions. They're . . .

Genetic Algorithms:

- are ~~optimization heuristics~~ shortcuts
- are simpler than you probably thought

Speaker notes

. . . simpler than you probably thought. They . . .

Genetic Algorithms:

- are ~~optimization heuristics~~ shortcuts
- are simpler than you probably thought
- can use very simple functions

Speaker notes

. . . can use very simple functions, but it . . .

Genetic Algorithms:

- are ~~optimization heuristics~~ shortcuts
- are simpler than you probably thought
- can use very simple functions
- can be tricky to figure out good functions

Speaker notes

. . . can be tricky to figure out exactly what the functions should do. This approach is also . . .

Genetic Algorithms:

- are ~~optimization heuristics~~ shortcuts
- are simpler than you probably thought
- can use very simple functions
- can be tricky to figure out good functions
- applicable to a wide variety of problems

Speaker notes

. . . applicable to a huge variety of problems, including ones so complex that . . .

Genetic Algorithms:

- are ~~optimization heuristics~~ shortcuts
- are simpler than you probably thought
- can use very simple functions
- can be tricky to figure out good functions
- applicable to a wide variety of problems
- can create solutions humans would not

Speaker notes

. . . a semi-random algorithm can come up with excellent solutions that we humans would never have thought of.

Now, if you have any . . .

? ? ? ? ?

T.Rex-2023@Codosaur.us
twitter.com/DaveAronson
linkedin.com/in/DaveAronson

Repo and Slides:
github.com/CodosaurusLLC/tight-genes
Codosaur.us/reds/gen-algs-open-23-slides

Speaker notes

. . . questions, I'll take them now, or at the contact info shown up there. As for the other URLs, the Github one is for the code, and slides in HTML, and the other one is for the slides as a PDF, complete with a full script. Any questions?