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

NOTE TO SELF: click on timer to reset it at the start

Hallo, Utrecht!

(Hello, Utrecht!)

Image: standard emoji

Speaker notes

haLLO uTREKHT!

Ik ben

(I'm Dave Aronson,)

Image: me speaking at JSConf Hawai'i 2020!

Speaker notes

Ik ben Dave Aronson,

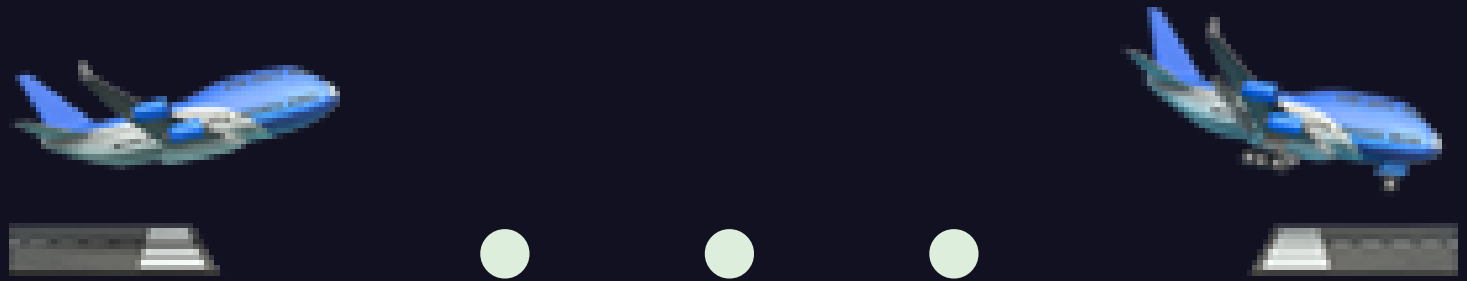
de T. Rex van Codosaurus,

(the T. Rex of Codosaurus,)

Speaker notes

deh Tay Rrrex fon Codosowrus

en ik ben hierheen gevlogen



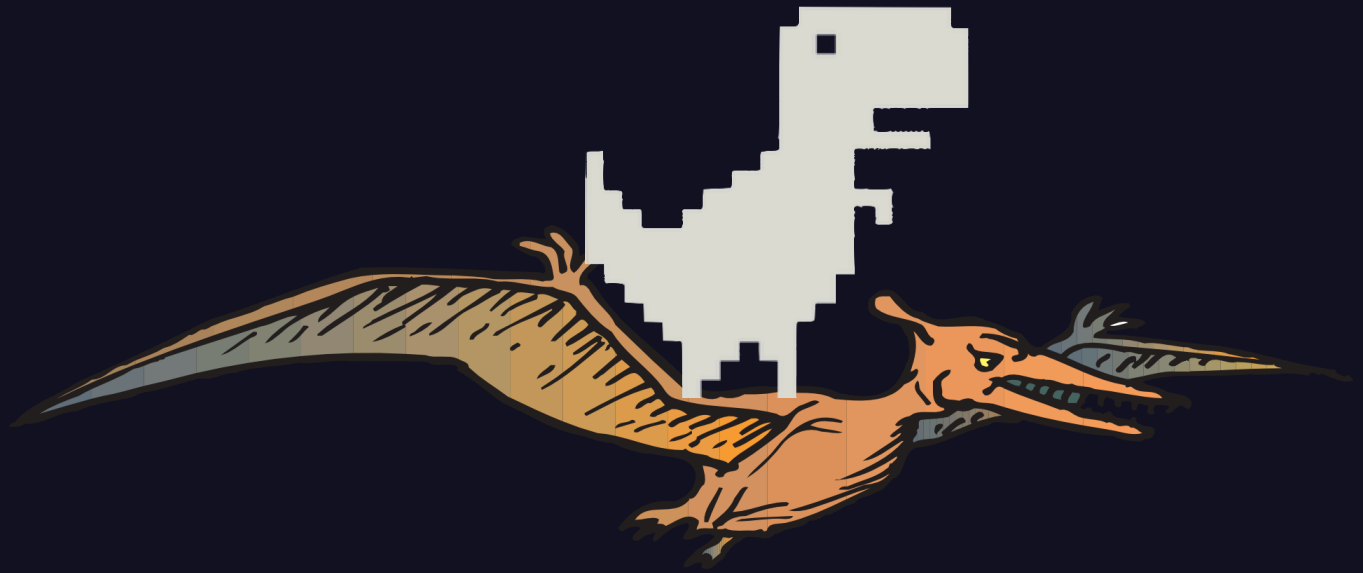
(and I flew here)

Image: standard emoji

Speaker notes

en ik ben heerhay ghefloghe h

op mijn pterodactylus



(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

oap main pteroDACteeluss

om jullie iets over

(to teach you about)

Speaker notes

ohm Yully eets ofer

Genetische Algoritmen te leren.

(Genetic Algorithms.)

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

Speaker notes

GheNEHtische AlghoRITmeh te lereh.

Maar . . .

(But . . .)

Image: standard emoji

Speaker notes

Maar . . .

ik zal het in het Engels doen.

(I will do it in English.)

Speaker notes

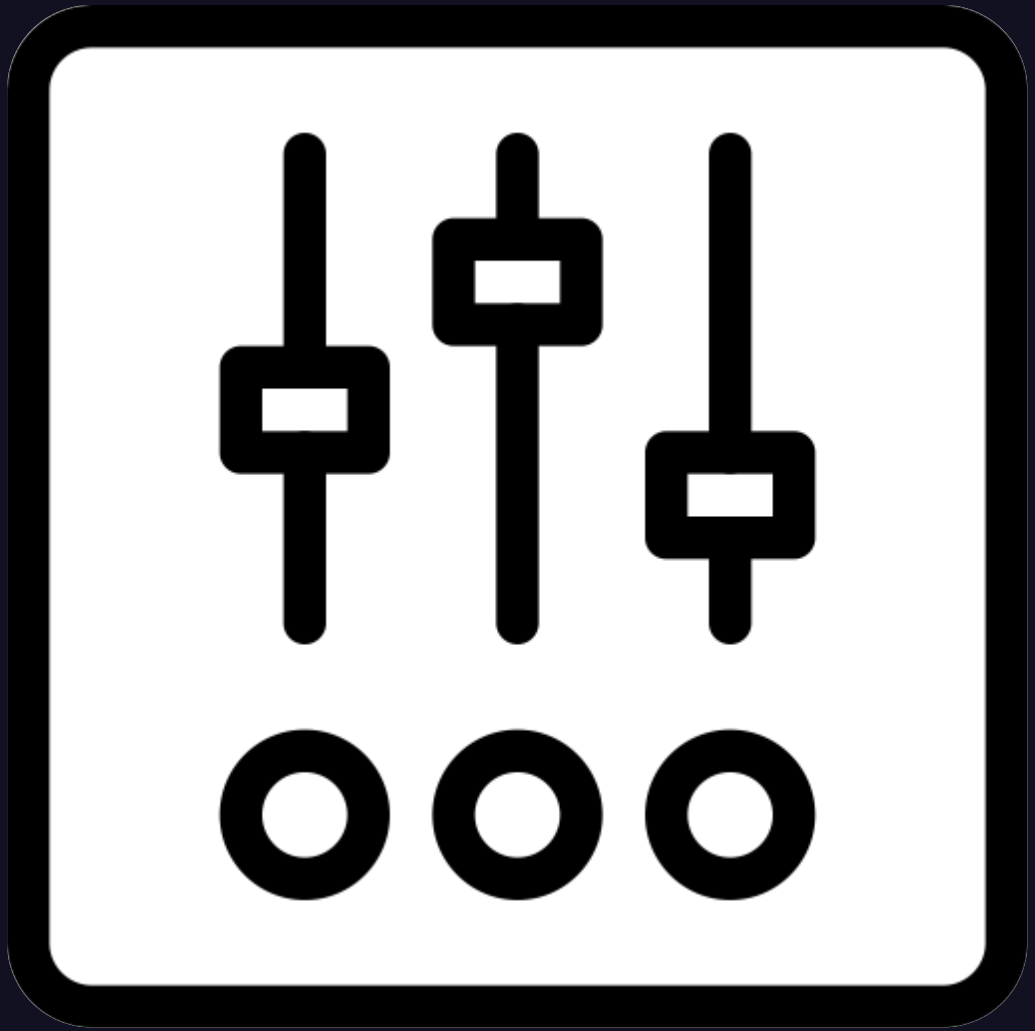
ik ZAL-et in-et ENgels doon.

Mainly because you've just heard almost all the Dutch I speak!
So . . .


Más
Educación
Adventista

Speaker notes

. . . what are genetic algorithms in the first place? The entire concept is an . . .



Speaker notes

... optimization meta-heuristic, so that individual genetic algorithms are optimization heuristics, . . .

WAT?!

Speaker notes

. . . and all that babble is just fancy-talk for . . .

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

Speaker notes

. . . the idea that genetic algorithms are a category of shortcuts to find solutions to hard problems. Generally these apply to problems that would be computationally intractable to attack by normal means. Ideally they'll find the best solution, but in reality we usually have to settle for something "good enough", due to constraints like time or money. After all, that's generally why we're using a shortcut in the first place.

There are many kinds of optimization heuristics, but genetic algorithms are (as you may have guessed from the name) uniquely inspired by . . .

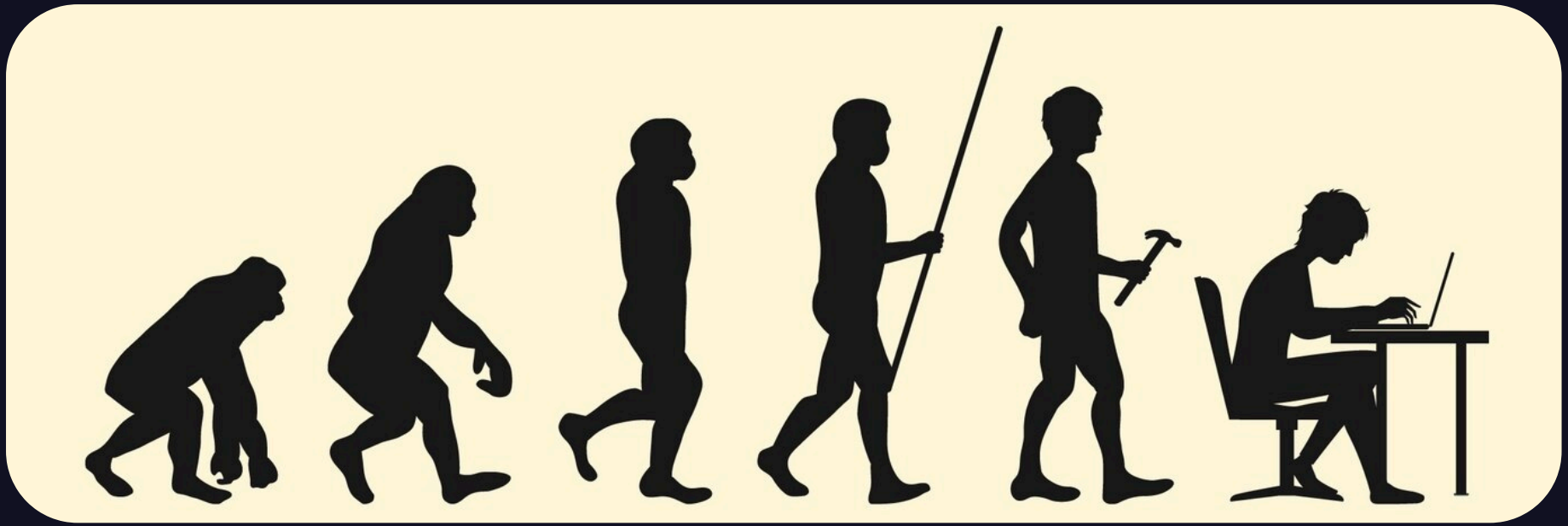
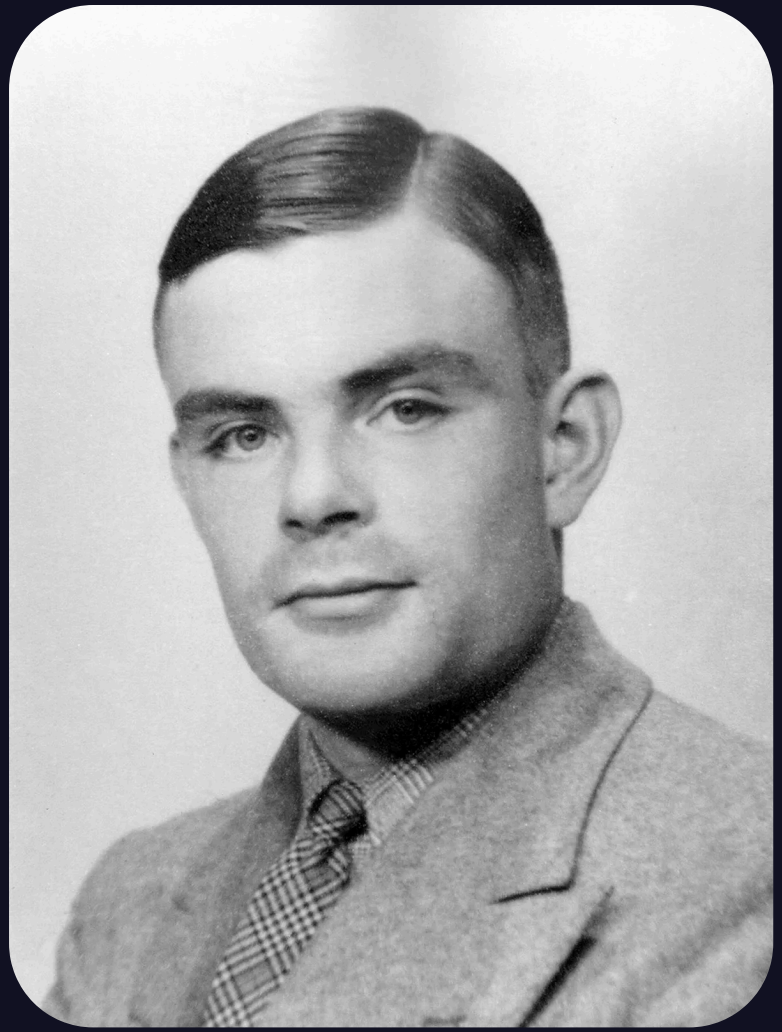


Image: https://www.freepik.com/free-vector/human-workers-evolution-silhouettes_725773.htm

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

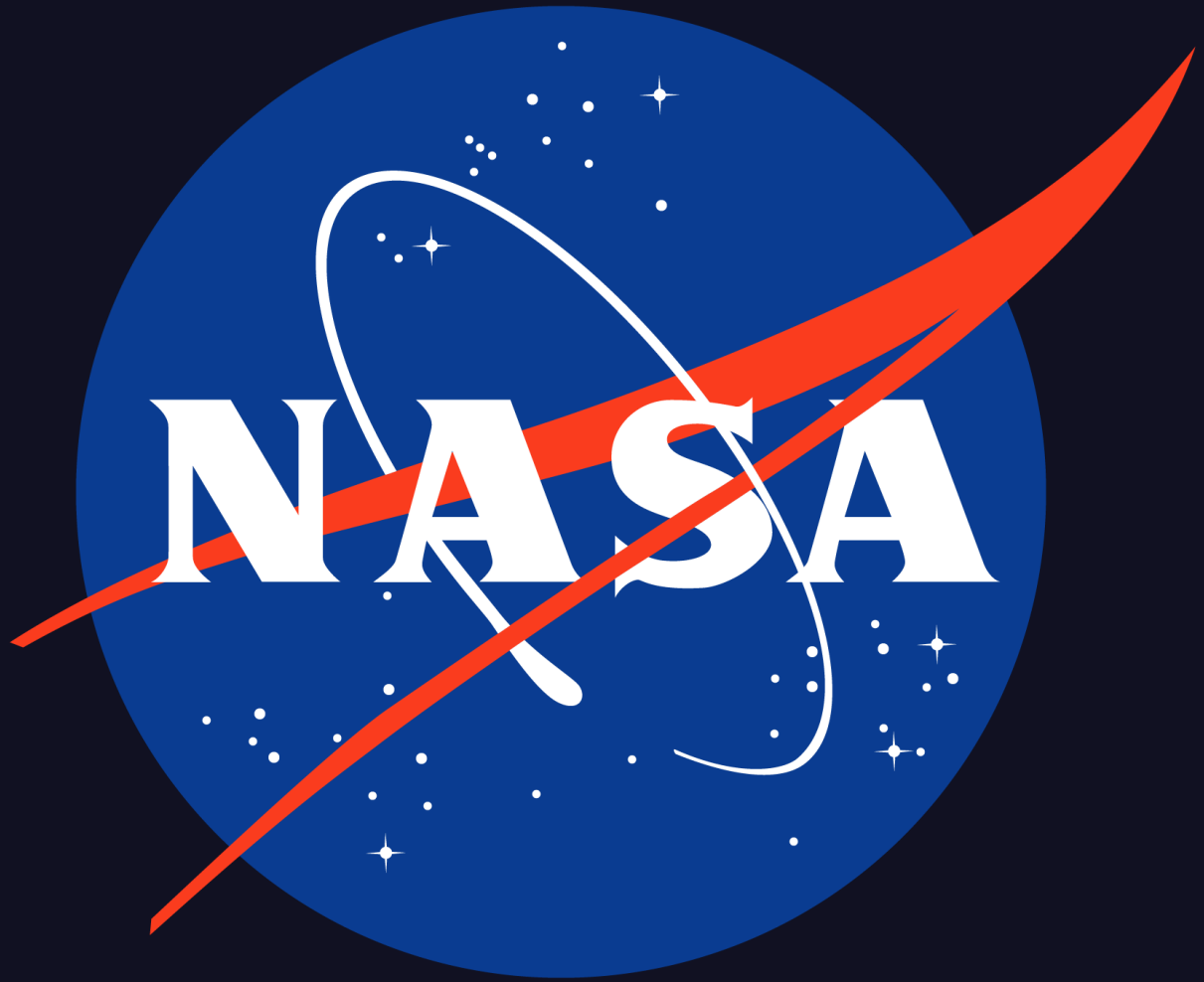
Speaker notes

. . . Alan Turing, as in Turing Test, Turing Machine, Turing Completeness, Turing Award and so on, proposed a "learning machine" in which he thought that the mechanism of learning would be similar to evolution. Nothing much ever 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. In 1989 a more general genetic algorithm toolkit, called Evolver, came out for PCs. These days, MATLAB and such tools have some genetic algorithm facilities built-in, and most major programming languages have genetic algorithm libraries available, including over 30 Ruby gems. However, the actual uses of genetic algorithms remain mostly hidden, and in my opinion frankly rather boring, used by companies in their internal industrial processes, logistics, scheduling, and so on.

But once in a while, they do 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, launched in 2006. (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, simple enough that I personally don't consider this to be a form of AI, but some conferences do, so here I am! Anyway:

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". I've also heard them called individuals, solutions, or phenotypes, to use another actual genetic term, but most people don't know that word, so I'll skip that one too.

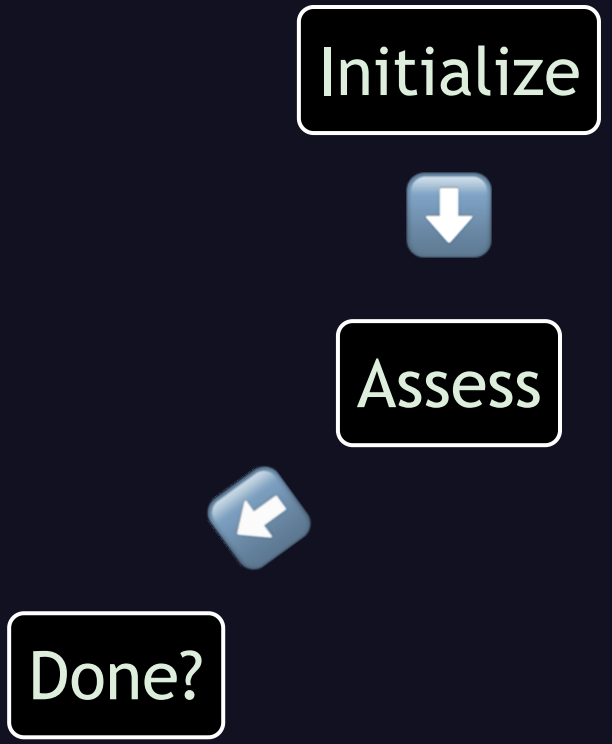
The next step is to . . .

Initialize

Assess

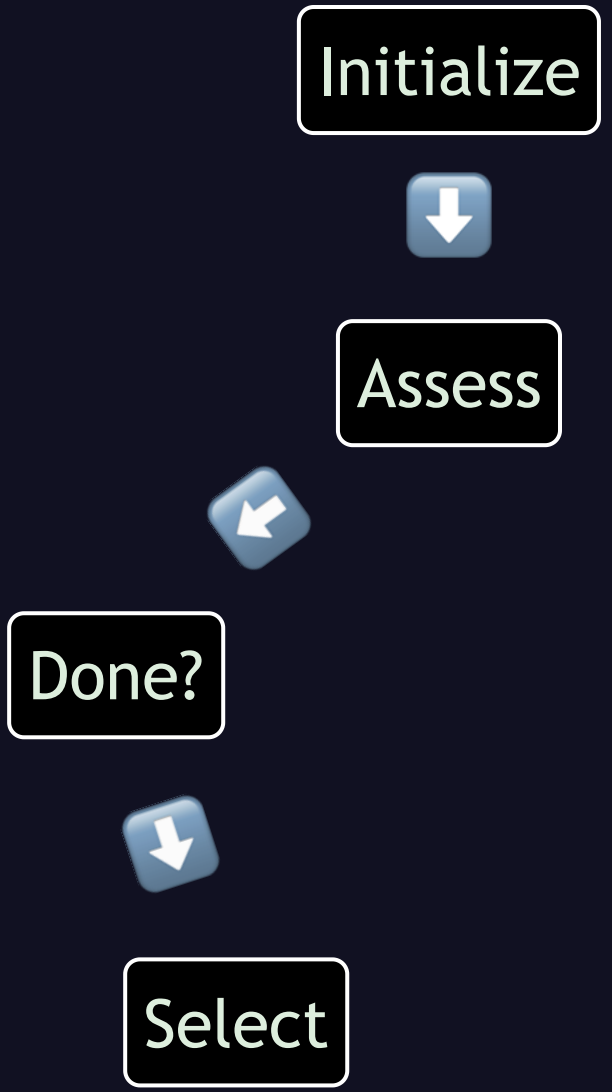
Speaker notes

. . . 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

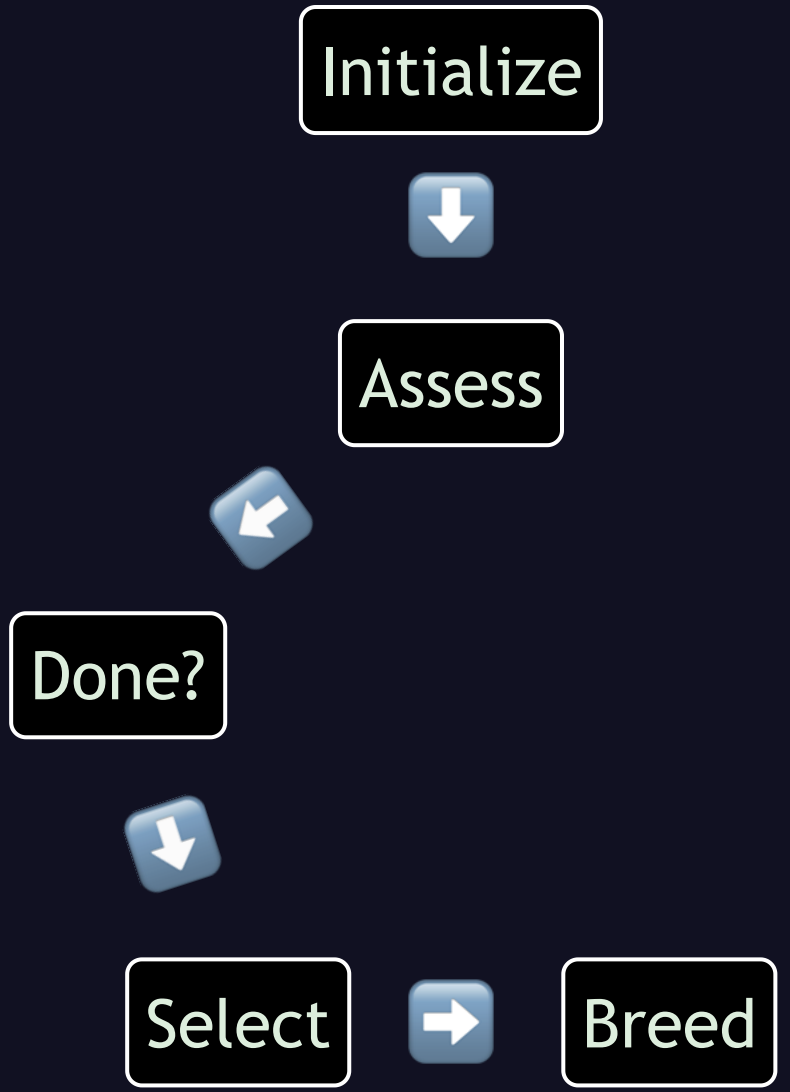
. . . done yet? This is usually based on the fitness, but could be based on other criteria, and we'll discuss some of those later, or a combination. If we're not done, then next we . . .


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.

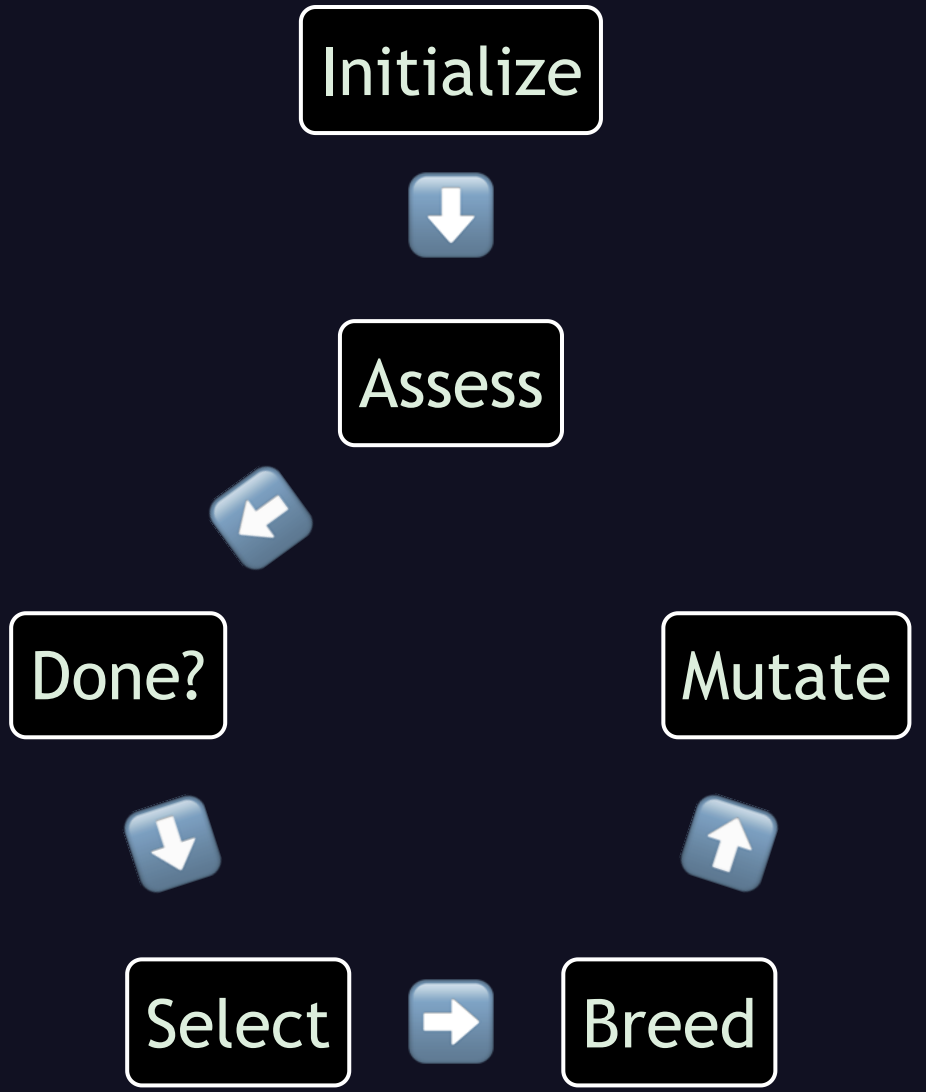
After that, as you may have guessed, we use those candidates we just selected . . .


Initialize
Assess
Done?
Select
Breed

Speaker notes

. . . to breed a new population. Some of the previous population, especially the fittest ones, may be carried over into this new population, but usually not.

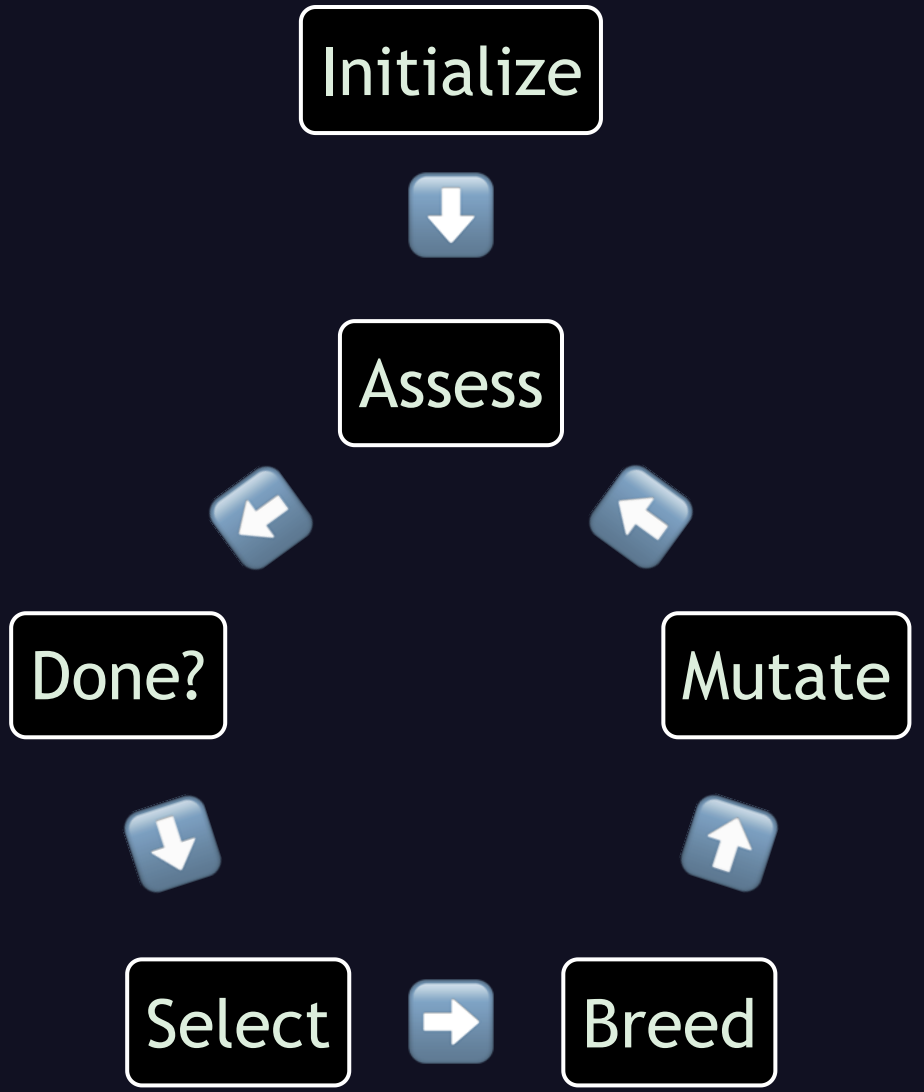
Next is a very important but easily forgotten step, which is to . . .


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

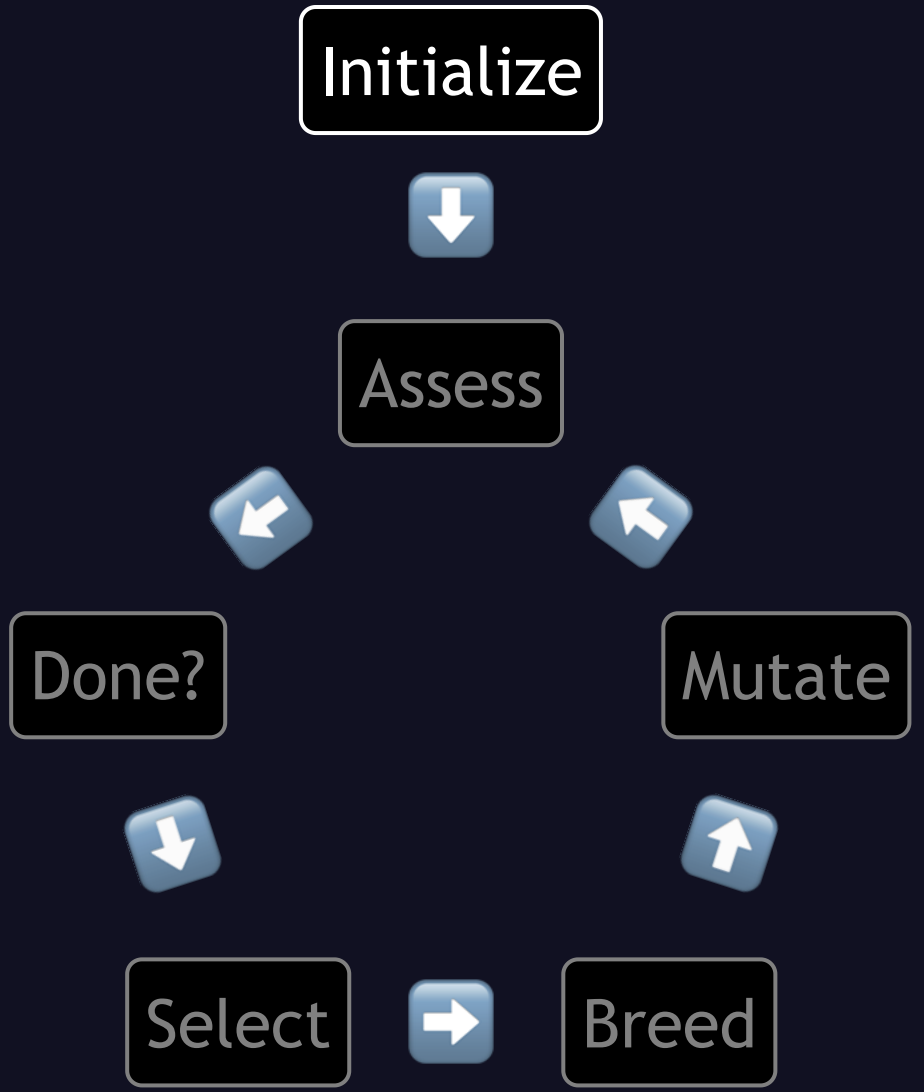
. . . go back to step 2, assessing their fitness. (You might want to take a picture of this slide and the next one, because they're the big takeaways about how genetic algorithms work at a high level. I'll give you a moment to get out your phones.) This sequence could be represented at a high level with 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

I know this conference is mainly focused on $WHATEVER, but this This code is in Ruby. I chose that to reduce boilerplate overhead, and because it reads so close to plain English, so even if you don't know Ruby, I'm confident you'll understand the ideas, and maybe even the actual code. I'll try to remember to explain any weird bits of Ruby syntax.

Now let's take a closer look at what goes on in 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. They could be any data structure we want, so long as we can evaluate their fitness, combine old ones to make a new one, and mutate them. 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, which is a category of constrained resource allocation problems. The canonical example is that you have a knapsack (or rucksack, backpack, or whatever you call it), and many things you want to carry in it, but they won't all fit, or the total weight is more than you can carry, or some such similar constraint, or combination of constraints. So you want to find the combination of items, that will fit the constraints, and has the maximum value. That could be the literal cash value, or something more metaphorical. In the canonical example, you might be taking the knapsack on a camping trip, so you want to pack whatever supplies you might need, for your survival, or at least your comfort. Different items will have different levels of importance for those purposes. For instance, common . . .


Mountain House
Mountain House
Mountain House
Mountain House
Mountain House Wraps
The MILEPOST
Dry
20 Litres
Dry
20 Litres
CLASSIC SERIES
Extra Value!
Long 24" hose and regulator adapts to bulk cylinder hookup for extended cooking time
Fast push-button start – no matches needed!
Two over-sized burners adjustable up to 12,000 BTUs each
Durable nylon carry bag holds stove and extra propane tanks
24,000 BTU high output
2-BURNER STOVE
Post De Distributeur
De 76 cm À 3
Conexión QCC1 Para Tanques De Tipo 1
Systéme De Distribution
De 76 cm À 3
Connexion QCC1
Pour Bouteille
De Type 1
LOWA
GORE-TEX

Speaker notes

. . . camping supplies would be much more valuable in this context than, say, an accordion, especially per kilogram or cubic centimeter.

There is a standard heuristic for knapsack problems, of simply adding the most valuable item that will fit, until we can't fit anything more. However, it's easy to envision times when that would not get us the best solution.

To look at a concrete example, suppose we know . . .

Image: standard emoji

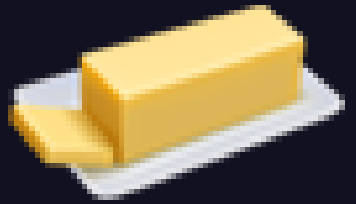
Speaker notes

. . . 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. (My apologies to any vegetarians out there.) For the sake of simplicity, we won't differentiate between price and profit, nor dairy versus meat cows, and I know it's very unrealistic but we'll assume that he can only take a set amount of each item, and always has that amount on hand. 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 tub	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, in other words, 4,000 pounds. (Don't get me started on stupid American units of measure!)

We could brute-force the problem by running through all the combinations, and see which of the light-enough ones totals the highest value. Or we could use the standard heuristic. But 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
  attr_reader :contents
  def initialize()
    @contents = rand(128)
  end
end
```

Speaker notes

Whoa, that looks like we're just making a random number! That's right! Making a random number from 0 to 2^7-1, gives us 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, . . .

```
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 those of you who do know Ruby, please 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

Why ten? Because that's what fits on the screen in a decently readable size, even all the way in the back. If I were doing this for real, with a much more complex domain, I might use a hundred, a thousand, a million, or even more.

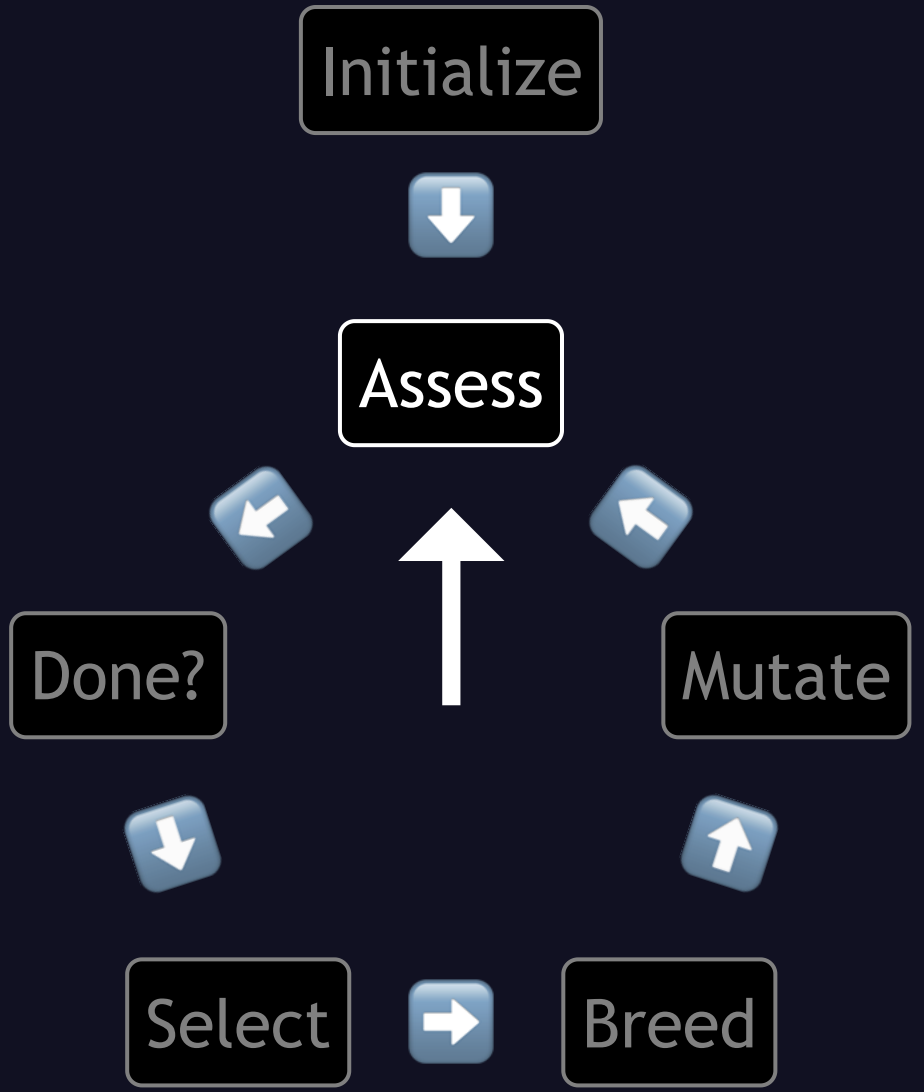
But how did we get from random numbers to those combinations? Behind the scenes, that translation might look like this:

```ruby
class Truckload
  class Item
    attr_reader :name, :weight, :value
    def initialize(name, weight, value)
      @name   = name
      @weight = weight
      @value  = value
    end
  end
  ITEMS = [
    Item.new("Cow",         1500,  2000),
    Item.new("Milk",        1720,   800),
    Item.new("Cheese",      1000, 12000),
    Item.new("Butter",      1000,  3000),
    Item.new("Ice Cream",   1000,  2000),
    Item.new("Meat",        1280,  8000),
    Item.new("Leather",     1100,  6000)
  ]
```

Speaker notes

We have a list of items we can take, as instances of another class describing them, with name, weight, and value attributes. (In this example it's an inner class, but it doesn't have to be.) To check what's in our cargo manifest, represented by our Truckload's contents value, we can iterate through the list of possible items, checking whether the corresponding bit is on. In the interests of time, I'll handwave over those details. Alternately, it could be implemented as an array of booleans, or even a hash, rather than a bitfield, but I wanted to illustrate the "string of bits" type of chromosome.

Now that we're done with Initialization, we . . .


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

. . . assess how "fit" each of these truckloads is. We do this with something called a "fitness function". (Surprise!) Just like how biological creatures might be perfectly fit for one environment but a lousy fit for another, this should reflect how fit a candidate is for some particular purpose. In this case, we already know we want the total cash value, BUT, any load that's too heavy for the truck, is worthless. In Ruby, that would look like this:

```ruby
def fitness()
  items = (0...ITEMS.count).
    select { |idx| bit_on?(idx) }.
    map { |idx| ITEMS[idx] }
  weight = items.map(&:weight).sum
  if weight > 4000
    return 0
  else
    return items.map(&:value).sum
  end
end
```

Speaker notes

We decode which items we want to take (abstracting away the actual bit-checking again for simplicity), 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. For instance, we could take into account the costs of refrigerating or freezing any items that need it.

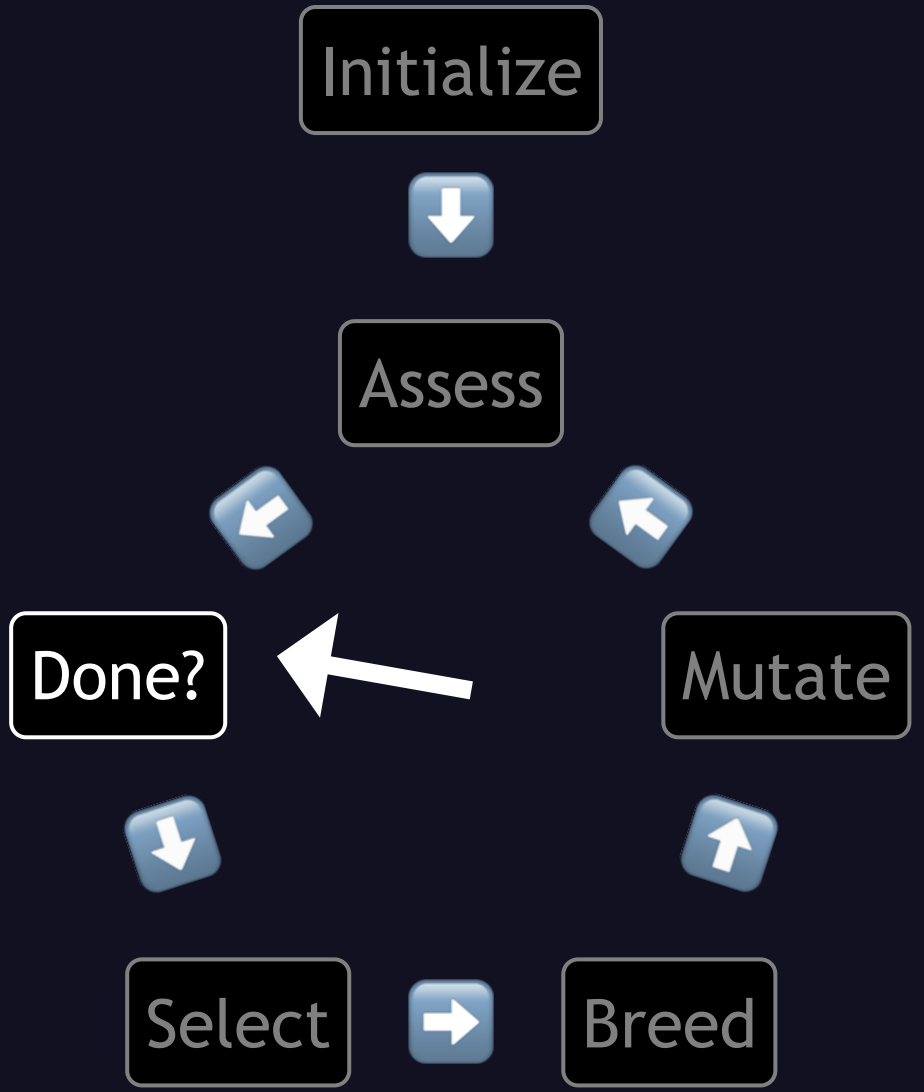
If we run this fitness function on our population, and sort on fitness descending, to make it easy to find the best, we get this:

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

Remember that 20,000 figure.

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.

Why 100? Pretty much random. It seems like enough for a good chance for improvement, and since what we're doing is so simple, and our population is so small, using lots of generations isn't very slow. In Ruby, that would look like this:

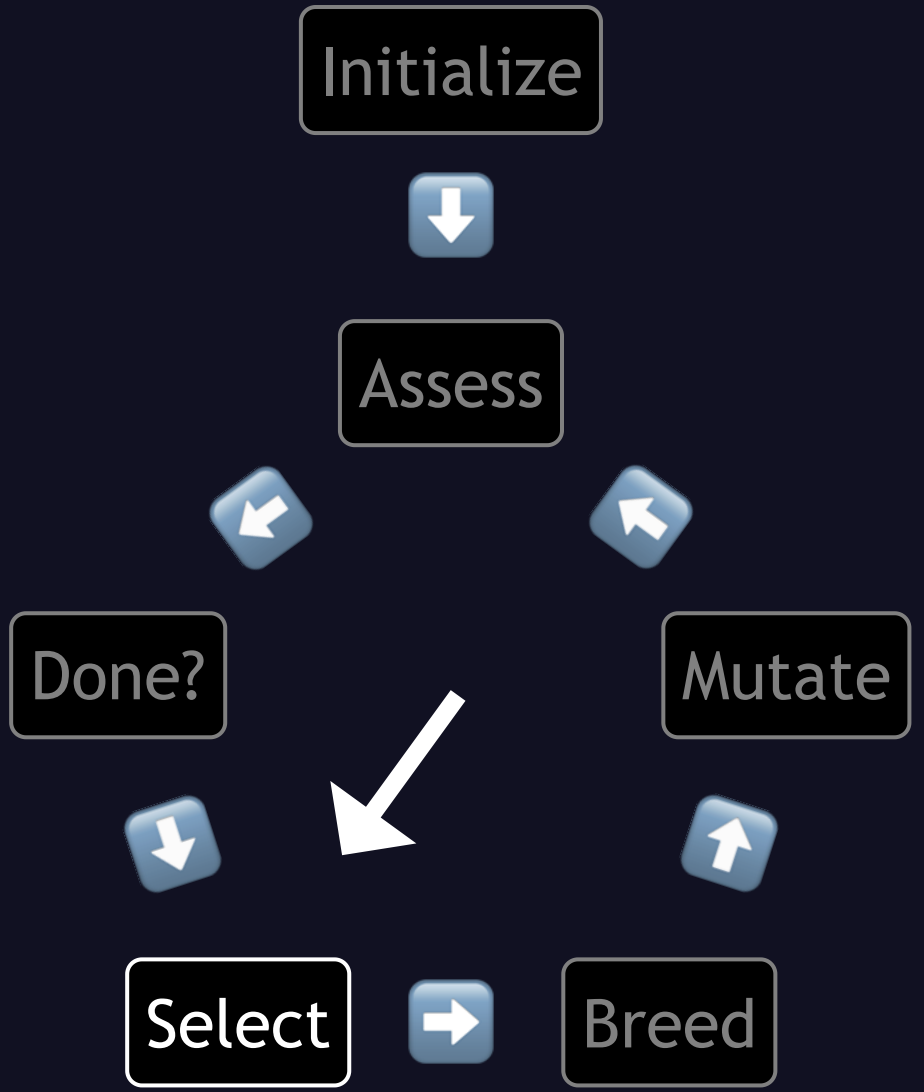
```ruby
@@best_combo = self.new(0)
@@generations = 0

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 run, to define the function, we set the initial best combo as empty, and we set how many generations it's been since we saw that, as zero, both as class variables; that's what the double at-signs means. 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. 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 the average or minimum fitness, or achieving some specific level of maximum fitness, some maximum number of generations, or amount of time, (whether clock time or CPU or whatever), 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, that would choose:

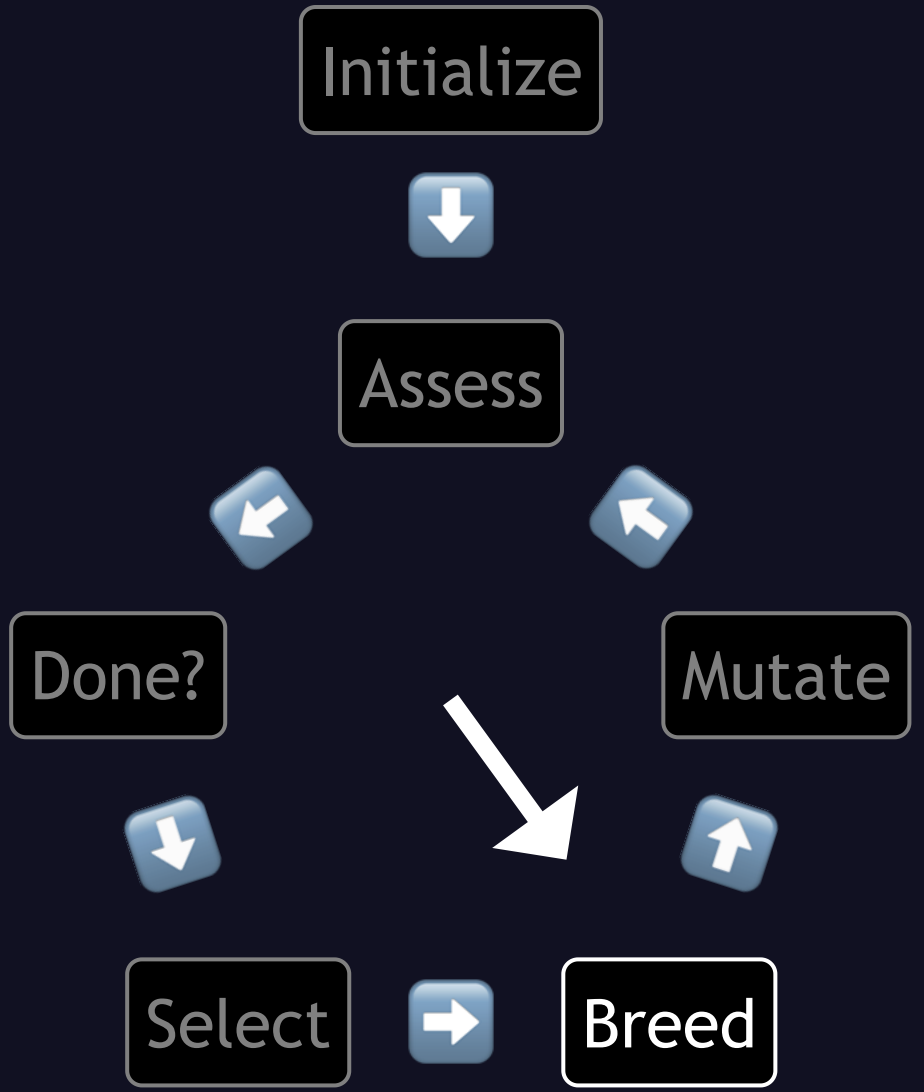
Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather	Fitness
N	N	Y	N	N	Y	N	20,000
N	Y	Y	N	Y	N	N	14,800

Speaker notes

these two. As usual, we also could get more complicated, and there are some more complex common alternatives, that we will look into later.

We could also take more than two, whether to combine more than two at once or to breed all pairs in that set, or larger subsets, such as all trios out of a set of five breeders, or even randomly choose those subsets.

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 from one parent, up to some randomly chosen crossover point, then switching to the other parent. In Genetic Algorithm terms, each place in the list is called a "gene" (that's where the name comes from), and the actual value there (in this case a yes/no decision) is called an allele. This can be extended with multiple crossover points, especially in the case of multiple parents being combined, but we're just going to use one, like so:

```
def self.breed(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, by index number. 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. This means that it could be all copied from one parent or the other, or it could switch at some point. We could just copy the sets of bits, but the code for that would be more complex and non-portable than I want to explain here.

As usual, we could get as complex as we want, like making some crossover points more or less likely, or even mandatory or forbidden. But we're going to keep it simple.

If we use this function once, with a crossover point of 3, so we take 3 values from the first parent, and the rest from the other, 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
```

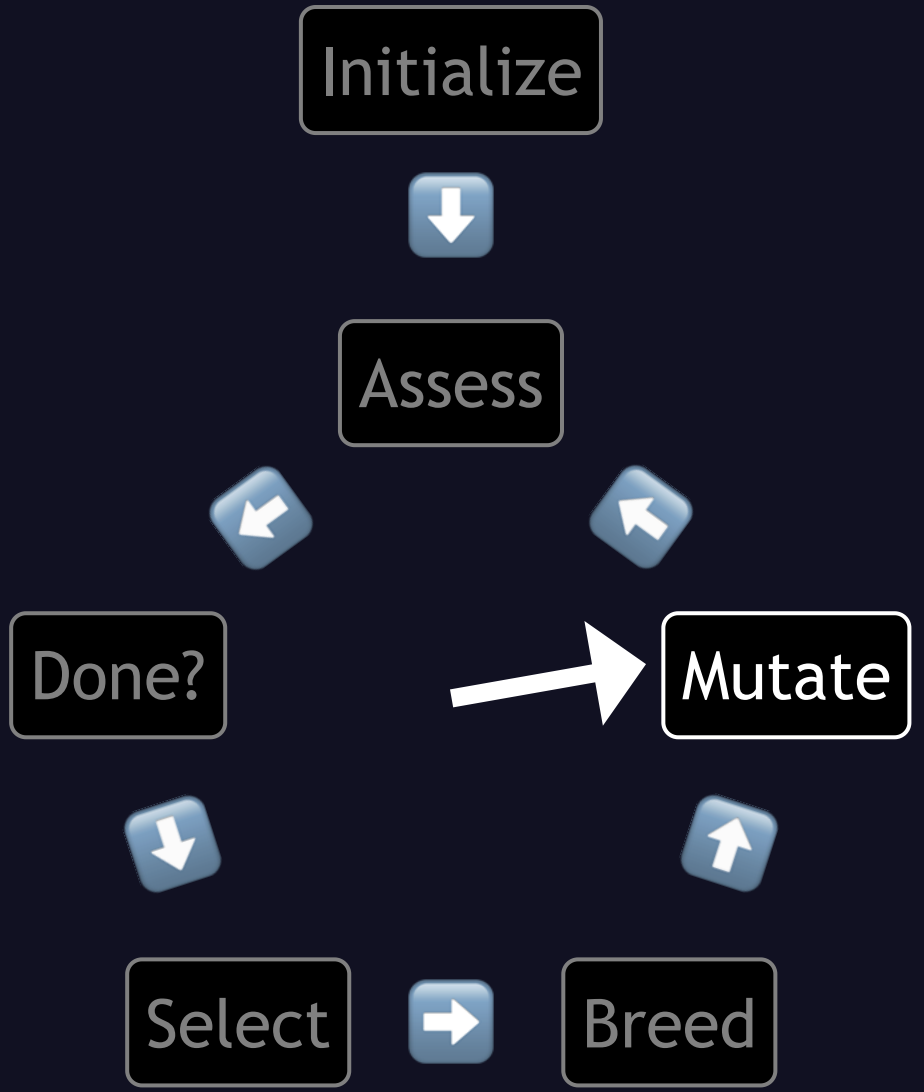
Speaker notes

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. Or in Genetic Algorithms terms, both chromosomes had the same alleles for each of those genes. 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:

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

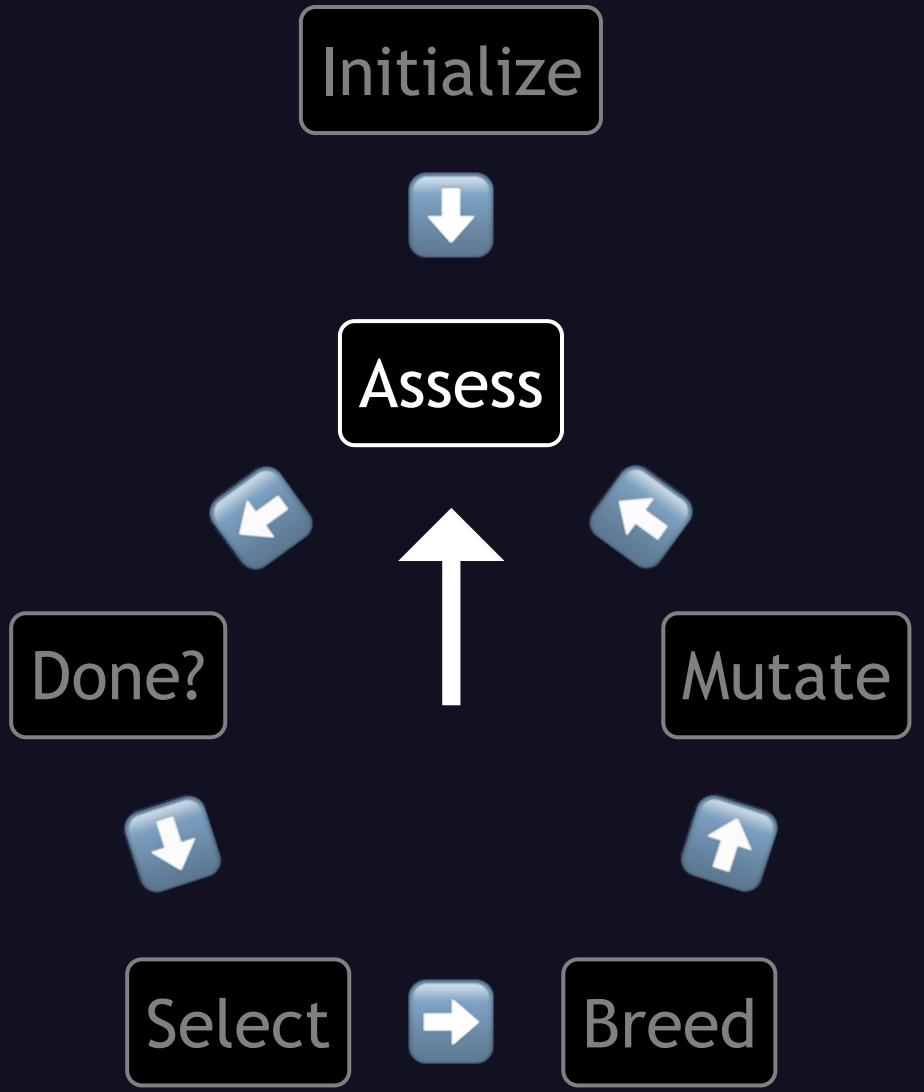
Speaker notes

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 or less 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.

But that's not really the best approach. Remember how I said that sometimes the fittest members of a population might be carried over into the next one? If I had carried over the fittest one, the maximum fitness would never go down, so the "are we done" function could have been a bit simpler, and we'd be done a bit faster. But, I didn't think of it until I had already made all the slides for this talk, and I didn't want to redo them. Also, I think this version is still worth exploring, to make the point that the fitness can go down and then come back up. So let's keep going.

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, with a few more occasions of a new best combo being found, we might wind up with something like this:

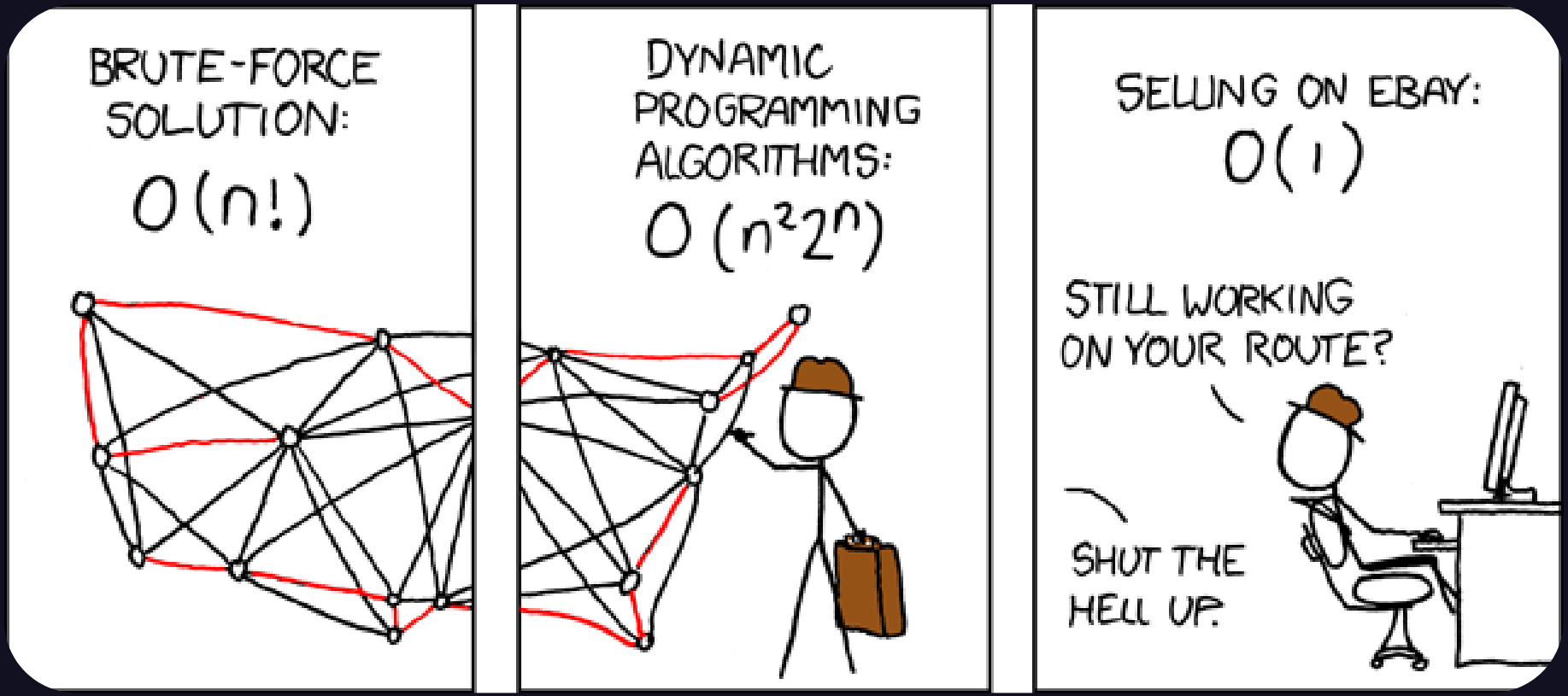
Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather	Fitness
N	N	Y	N	N	Y	Y	26,000
N	N	Y	Y	N	N	Y	21,000
N	Y	N	N	Y	Y	N	10,800
Y	N	N	N	N	N	Y	8,000
N	N	N	N	N	Y	N	8,000
N	N	N	Y	Y	N	N	5,000
N	Y	N	N	N	N	N	8,00
N	Y	N	N	N	N	N	8,00
Y	Y	N	Y	N	N	Y	0
Y	Y	N	Y	Y	Y	Y	0

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.

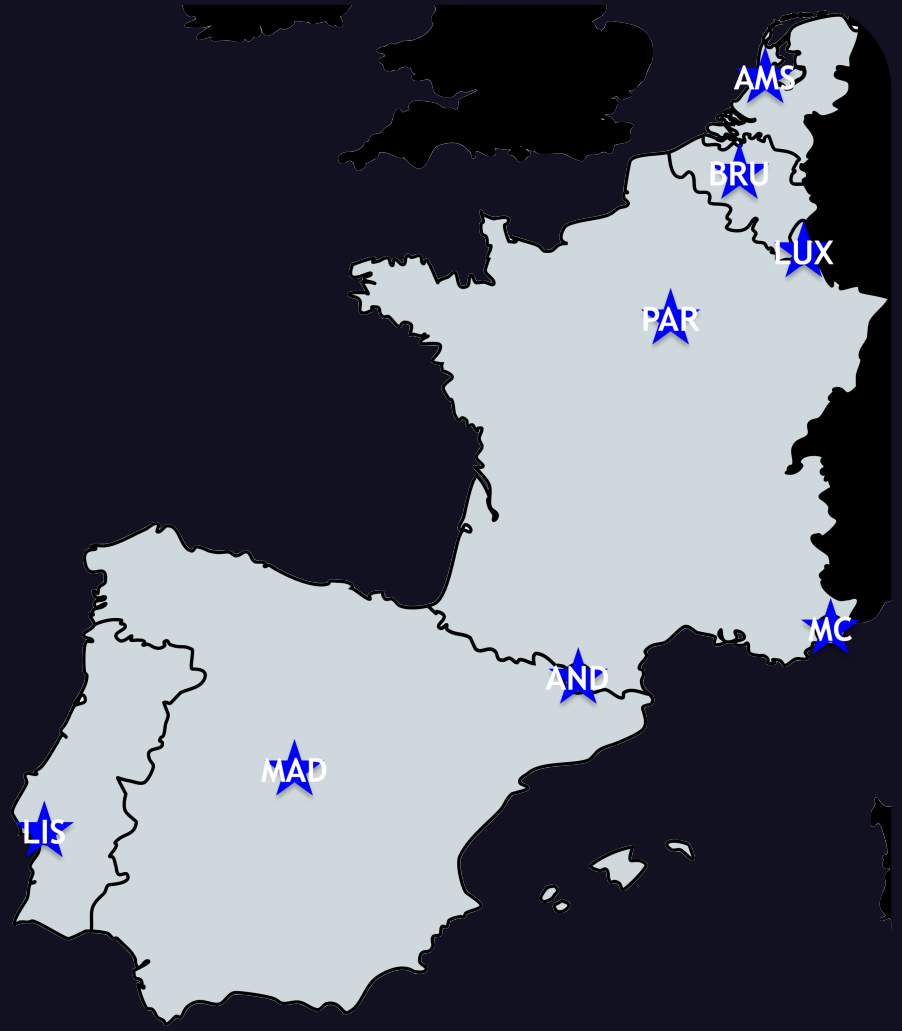
Now, suppose we want to evolve solutions to a different huge class of problems: . . .


BRUTE-FORCE SOLUTION:
O(n!)
DYNAMIC PROGRAMMING ALGORITHMS:
O(n²2ⁿ)
SELLING ON EBAY:
O(1)
STILL WORKING ON YOUR ROUTE?
SHUT THE HELL UP.

Speaker notes

. . . Traveling Salesman problems. The canonical example is that you're literally a traveling salesman, and you want to find the shortest route to visit a list of cities. Realistically, we may want to include other factors, such as the time or money it takes to get there, which may not be proportional to the distance, and the expected time to spend in each city. But to keep this example simple, we'll just look at the distance.

Our list of cities will be the capitals of mainland Europe, west of Germany. In alphabetical order, that's:


AMS
BRU
LUX
PAR
MC
AND
MAD
LIS

Speaker notes

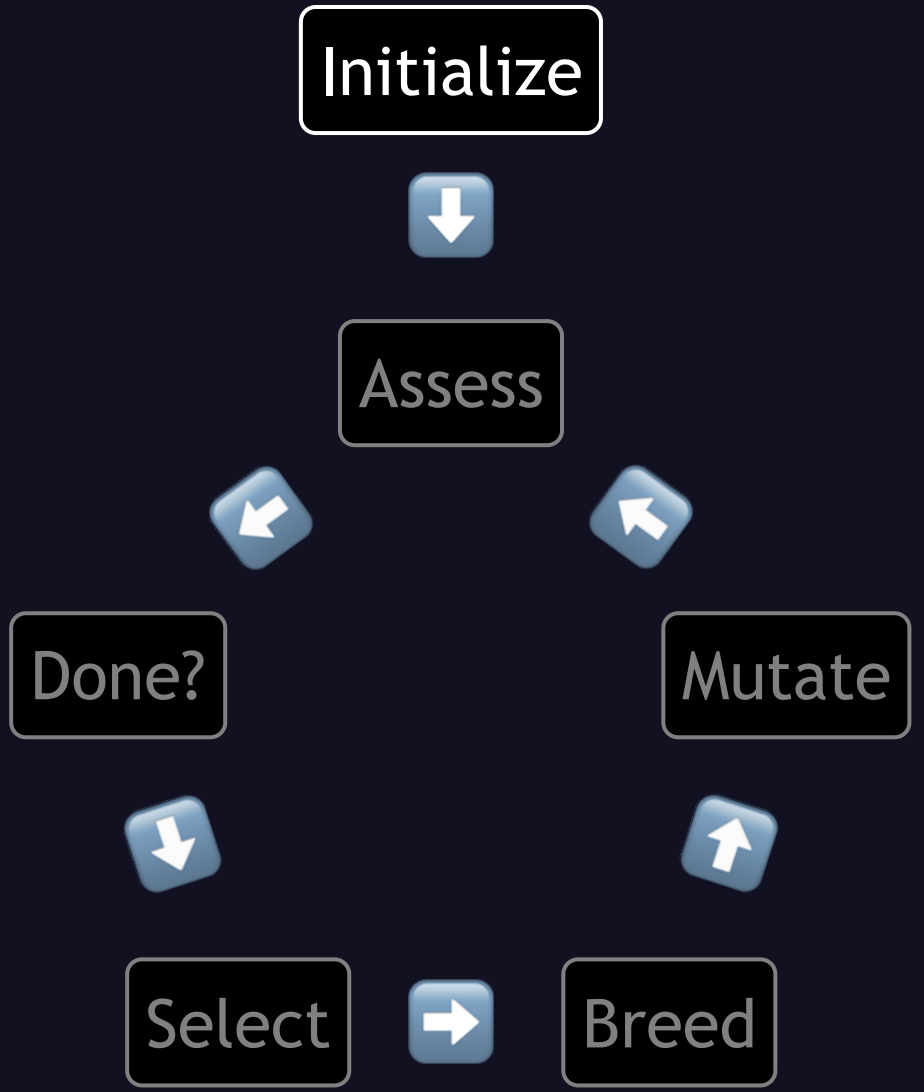
Amsterdam, Andorra la Vella, Brussels, Lisbon, Luxembourg, Madrid, Monte Carlo, and Paris. We have the . . .

To From	AND	BRU	LIS	LUX	MAD	MTC	PAR
AMS	1357	210	2233	417	1773	1421	502
AND	-	1162	1232	1178	613	653	862
BRU	-	-	2038	213	1577	1200	307
LIS	-	-	-	2153	625	1838	1739
LUX	-	-	-	-	1691	1041	386
MAD	-	-	-	-	-	1288	1278
MTC	-	-	-	-	-	-	956

Speaker notes

. . . distances between them, as shown here in kilometers. This is the minimum driving distance according to Google Maps, and to keep things simple we'll assume it's the same in either direction.

So let's dive back into our process. That starts with . . .


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

. . . creating an initial population, which would be done the same way as before, except that we're creating routes instead of truckloads. So how do we create a route? It could be done quite easily, like this:

```ruby
class Route
  CITIES = %w(AMS AND BRU LIS LUX MAD MTC PAR)

  attr_reader :stops
  def initialize(stops=CITIES.shuffle)
    @stops = stops
  end
end
```

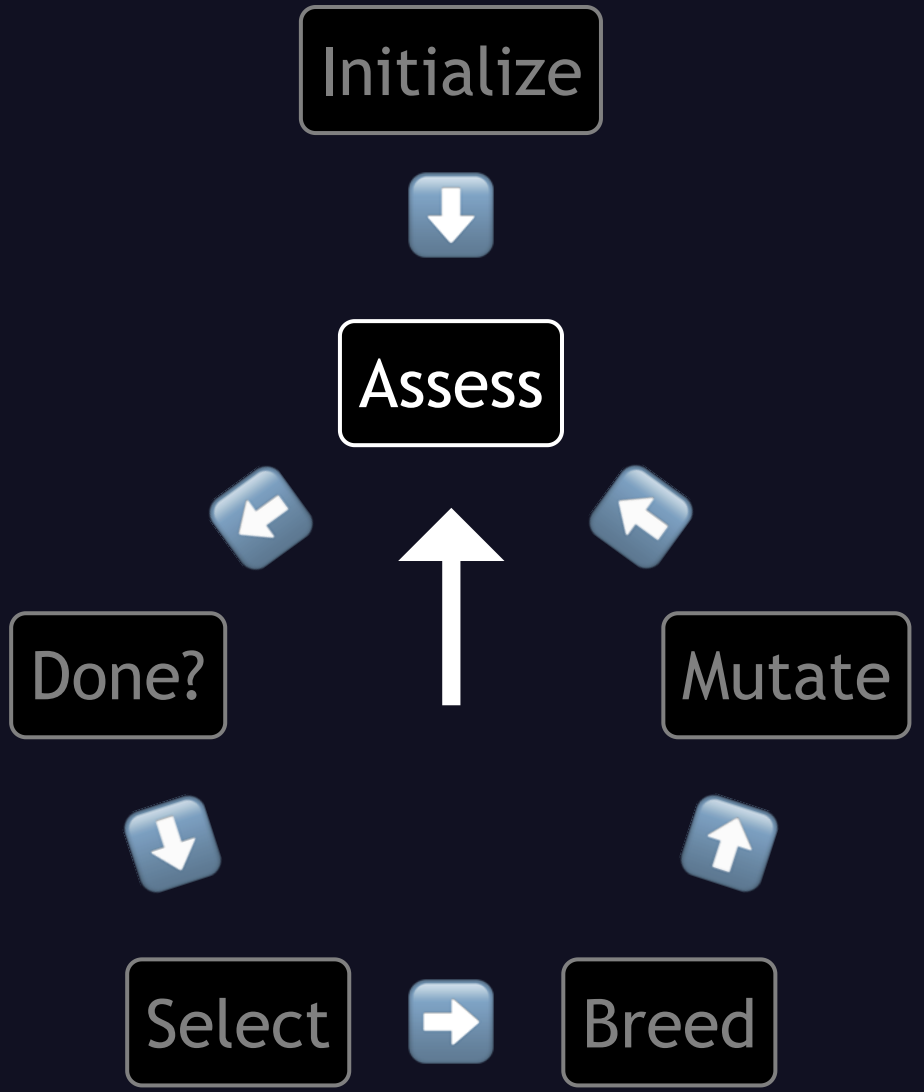
Speaker notes

Again we're going to make a class, this time called Route, with the list of city abbreviations as a constant. If we create ten of these and put them in an array, it might look like this:

1st	2nd	3rd	4th	5th	6th	7th	8th
AND	MAD	AMS	LUX	PAR	BRU	MTC	LIS
MTC	PAR	LUX	AND	MAD	LIS	AMS	BRU
MAD	LUX	BRU	AMS	LIS	AND	MTC	PAR
AND	PAR	LUX	BRU	AMS	LIS	MAD	MTC
LIS	LUX	MTC	AMS	AND	BRU	PAR	MAD
BRU	LUX	LIS	PAR	AND	MAD	MTC	AMS
AND	LUX	PAR	MTC	BRU	MAD	AMS	LIS
LUX	BRU	AND	MAD	PAR	LIS	AMS	MTC
AND	AMS	MTC	PAR	LIS	BRU	LUX	MAD
PAR	MAD	AMS	BRU	AND	MTC	LIS	LUX

Speaker notes

So that's the Initialize step done. Now we have to . . .


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

. . . determine how fit each route is. The measure we have is better when smaller and worse when larger, but that's easy to take care of. We can subtract it from a constant, divide a constant by it, or all kinds of other solutions. I decided to subtract the total distance of each one from . . .

```
WORST_ROUTE = Route.new(%w(AMS
                           LIS
                           LUX
                           MAD
                           BRU
                           MTC
                           PAR
                           AND))

# this works out to 12_029
WORST_DISTANCE = WORST_ROUTE.fitness()
```

Speaker notes

. . . the fitness of the worst route I could easily construct manually. I started in Amsterdam, and then did the opposite of the usual heuristic, repeatedly going to the furthest unvisited city, until I had included them all, then back to Amsterdam. The fitness function is fairly straightforward:

```ruby
def fitness = WORST_DISTANCE - total_distance

def total_distance
  stops.
    each_cons(2).
    to_a.
    map { |src, dst| distance(src, dst) }.
    sum +
    distance(stops.first, stops.last)
end
```

Speaker notes

. . . if you know Ruby. If you don't, what's going on here is, we calculate the total distance by taking the stops, extracting each consecutive pair, but that gets us an Enumerator so we have to convert that to an actual array, then we map each pair to its distance by calling a function which I'll save for the next slide, add them all up, and finally add the distance back to the starting point. Then to get the fitness we finally subtract that total distance from the worst distance. I'm making the simplifying assumption that the route could start anywhere, but we could say that we have to start and end at some other city. There are many variations on the Traveling Salesman Problem, and another one is that we don't have to get back to the starting point.

Now that you all grok that, let's take a brief look at how that distance function might work:

```ruby
DISTANCES = {
  "AMS" => {
    "AND" => 1357, "BRU" =>  210, "LIS" => 2233, "LUX" =>  417,
    "MAD" => 1773, "MTC" => 1421, "PAR" =>  502
  },
  "AND" => {
    "BRU" => 1162, "LIS" => 1232, "LUX" => 1178,
    "MAD" =>  613, "MTC" =>  653, "PAR" =>  862
  },
  "BRU" => {
    "LIS" => 2038, "LUX" =>  213, "MAD" => 1577,
    "MTC" => 1200, "PAR" =>  307
  },
  "LIS" => { "LUX" => 2153, "MAD" =>  625, "MTC" => 1838, "PAR" => 1739 },
  "LUX" => { "MAD" => 1691, "MTC" => 1041, "PAR" =>  386 },
  "MAD" => { "MTC" => 1288, "PAR" => 1278 },
  "MTC" => { "PAR" =>  956 },
}

def distance(src, dst)
  src, dst = dst, src if src > dst
  DISTANCES[src][dst]
end
```

Speaker notes

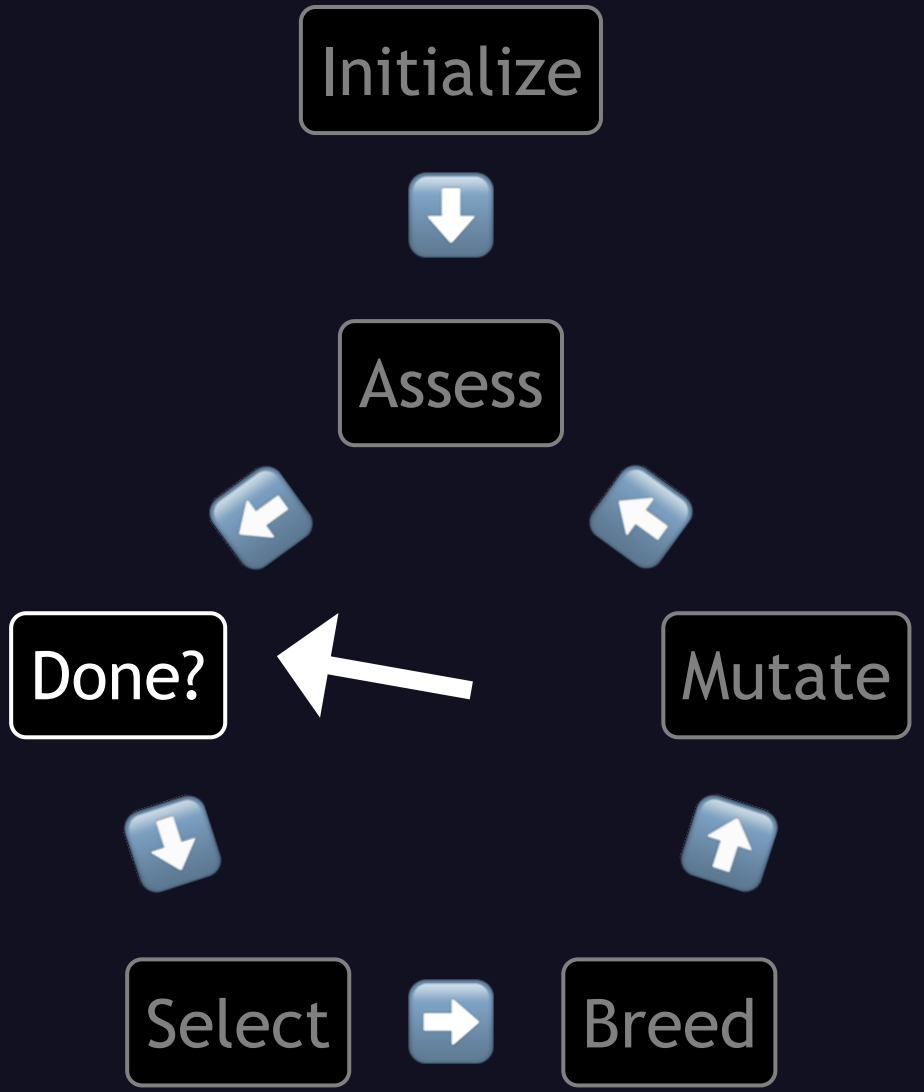
Conceptually, it's pretty simple. First we declare a constant hash of hashes, (or whatever your preferred language calls an associative array), from each city other than the last, to each city alphabetically later than it. Then we declare a function, that takes two cities, makes sure we've got them in the right order, then looks up the distance in the hash of hashes. This could probably be done in a more Ruby-ish way with some mechanism like hash defaults, and we could have saved some memory by using symbols instead of strings, but once again, I'm trying to keep this understandable for people who don't know Ruby.

Now, if we finally run that fitness function on our current population, and sort on fitness descending, we get:

1st	2nd	3rd	4th	5th	6th	7th	8th	Fit
AND	PAR	LUX	BRU	AMS	LIS	MAD	MTC	5559
MTC	PAR	LUX	AND	MAD	LIS	AMS	BRU	4628
AND	MAD	AMS	LUX	PAR	BRU	MTC	LIS	4263
MAD	LUX	BRU	AMS	LIS	AND	MTC	PAR	3563
BRU	LUX	LIS	PAR	AND	MAD	MTC	AMS	3530
LIS	LUX	MTC	AMS	AND	BRU	PAR	MAD	2685
PAR	MAD	AMS	BRU	AND	MTC	LIS	LUX	2576
LUX	BRU	AND	MAD	PAR	LIS	AMS	MTC	2329
AND	AMS	MTC	PAR	LIS	BRU	LUX	MAD	2001
AND	LUX	PAR	MTC	BRU	MAD	AMS	LIS	1494

Speaker notes

. . . this. The best route of this generation is 5_559 km shorter than the worst route I could easily construct manually, and the worst of this generation is still 1_494 km shorter than that same worst route. So now we can use this information to decide . . .


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

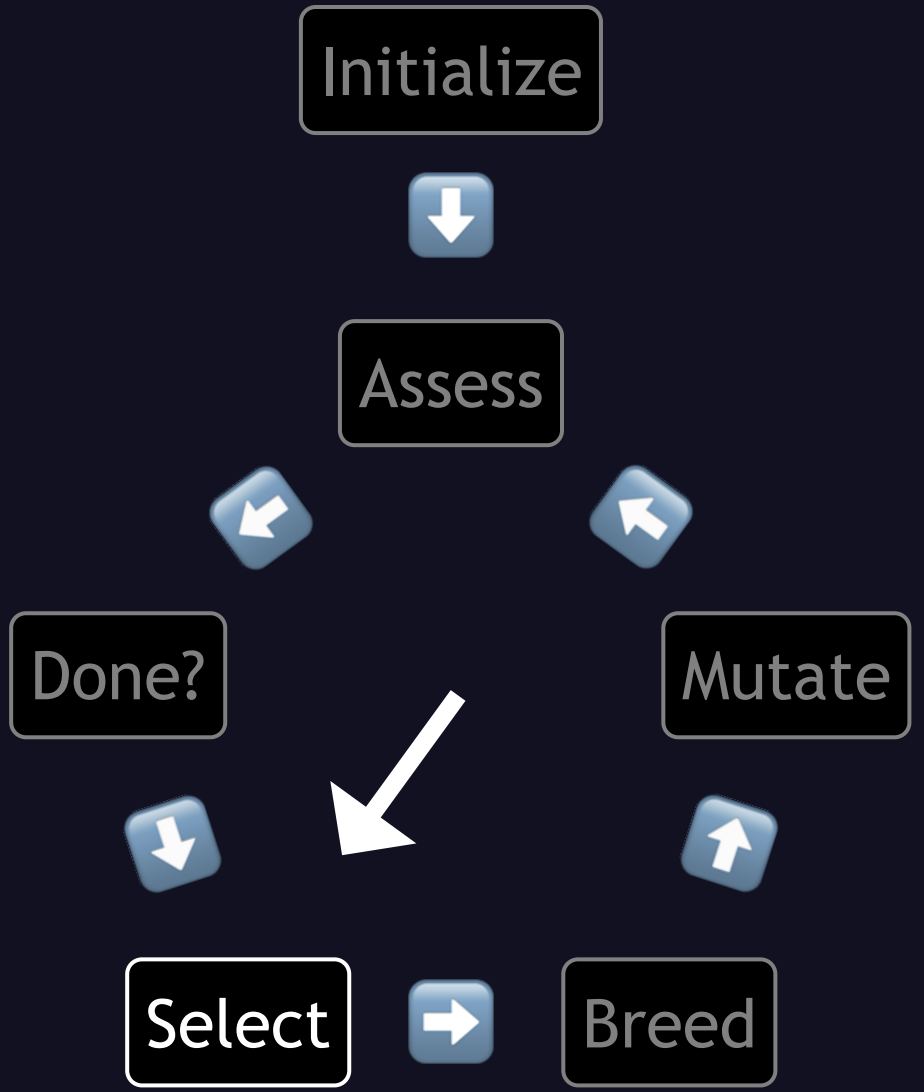
. . . are we done? What are our criteria? For now I'm going to stick with the idea of declaring a winner if it hasn't been outdone in 100 generations, but with the twist that we'll also stop after a maximum of 1000 generations, so of course we're not done on the first pass. The code . . .

```ruby
@@best_route = WORST_ROUTE
@@generations = 0

def self.done?(population)
  @@generations += 1
  better = population.
    select { |r| r.fitness > @@best_route.fitness }
  if better.any?
    @@best_route = better.sort_by(&:fitness).last
    @@generations = 0
    result = false  # just stash it
  else
    result = @@generations >= 100  # just stash it
  end
  return result || @@generations >= 1000  # apply limit
end
```

Speaker notes

. . . is exactly the same as last time, except for stashing the result in a variable and OR-ing that with the total generation check, and of course that now we're using Routes instead of Truckloads.

Since we're not done, we now . . .


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

. . . pick some breeders, and this time we're going to delve into Roulette Wheel selection. The concept there is that every candidate has a chance to be selected, but the size of the chance is based on the candidate's fitness. (So I think this is actually a misnomer; one would hope that an actual roulette wheel has an equal chance for all the numbers! But, they didn't ask my opinion when naming this.) The simplest version is that it's just equal, or at least directly proportional, to the fitness. The . . .

```ruby
def self.select_breeders(pop)
  p1 = pick_winner(pop)
  p2 = pick_winner(pop - [p1])
  [p1, p2]
end

def self.pick_winner(pop)
  total = pop.map(&:fitness).sum
  target = rand(total)
  so_far = 0
  pop.each do |p|
    so_far += p.fitness
    return p if so_far > target
  end
end
```

Speaker notes

. . . code for that is of course more complex than last time, when we simply chose the top two. Basically what we're doing is summing up all the fitnesses, generating a random number from 0 to the sum minus 1, and figuring out which route's range that falls in. To do that, we iterate over the routes, summing the fitnesses again, until we exceed the random number. When that happens, it means that the one whose fitness we just added is the lucky winner. Then we do that again, without the one we already picked.

As usual we could make it even more complex, such as by applying some function to the actual fitness. We might amplify the fitter routes' chances by squaring the fitness, or diminish them by taking the square root, or a logarithm or whatever, or we could make those things part of the definition of the fitness in the first place. We could also bias it by generating that random number differently, with an uneven distribution, whether favoring the best or the middle or whatever.

If we run this on our current population, we just might randomly wind up with . . .

1st	2nd	3rd	4th	5th	6th	7th	8th	Fit
MAD	LUX	BRU	AMS	LIS	AND	MTC	PAR	3563
LUX	BRU	AND	MAD	PAR	LIS	AMS	MTC	2329

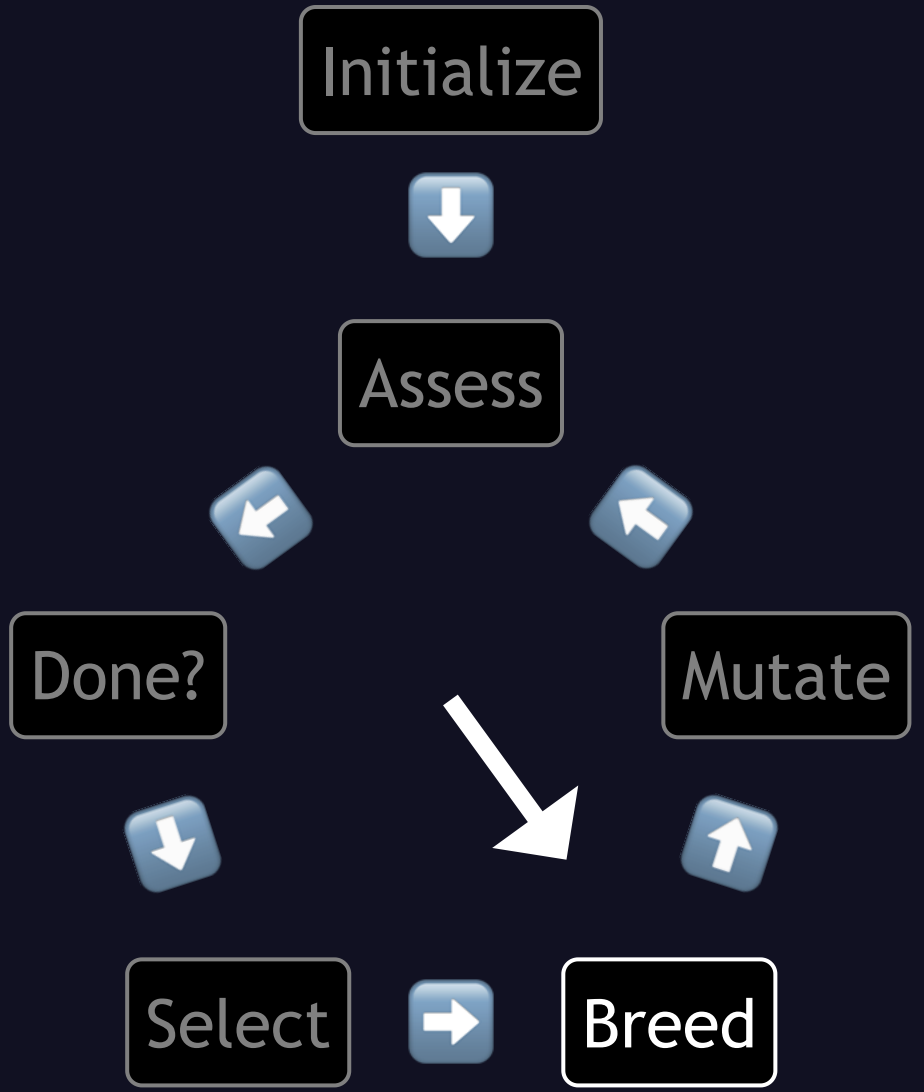
Speaker notes

. . . these two. These weren't the top two, in fact they were #4 and #8, which is fairly poor, averaging out to #6 out of 10. But just like in real life, to quote the late great . . .

Image: https://commons.wikimedia.org/wiki/File:Tom_Petty_(8191710373).jpg

Speaker notes

. . . Tom Petty, even the losers get lucky sometimes. Even I managed to land a wife! So now it's time to . . .


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

. . . breed them together. Breeding Traveling Salesman routes is much more complex than breeding Knapsack contents. It's actually still an area of ongoing research! But the simplest way to breed Traveling Salesman routes is fairly similar to ordinary crossover: we . . .

```ruby
def self.breed(p1, p2)
  xover = rand(CITIES.length + 1)
  cities = []
  cities[0 .. (xover - 1)] =
    p1.stops.slice(0, xover)
  cities[xover .. (CITIES.length - 1)] =
    p2.stops.reject { |city| cities.member?(city) }
  return Route.new(cities)
end
```

Speaker notes

. . . take a random number, between 0 and the number of cities, inclusive, copy that many from the first parent, and fill the rest (if any) from the second parent. However, we don't do it by copying them straight down in place, like with regular crossover, but by using the missing cities, in the order they appear in the second parent, no matter exactly where they appeared. So, supposing we have a crossover point of 3, the result would look like . . .

1st	2nd	3rd	4th	5th	6th	7th	8th
MAD	LUX	BRU	AMS	LIS	AND	MTC	PAR
			+				
LUX	BRU	AND	MAD	PAR	LIS	AMS	MTC
			=				
MAD	LUX	BRU	AND	PAR	LIS	AMS	MTC

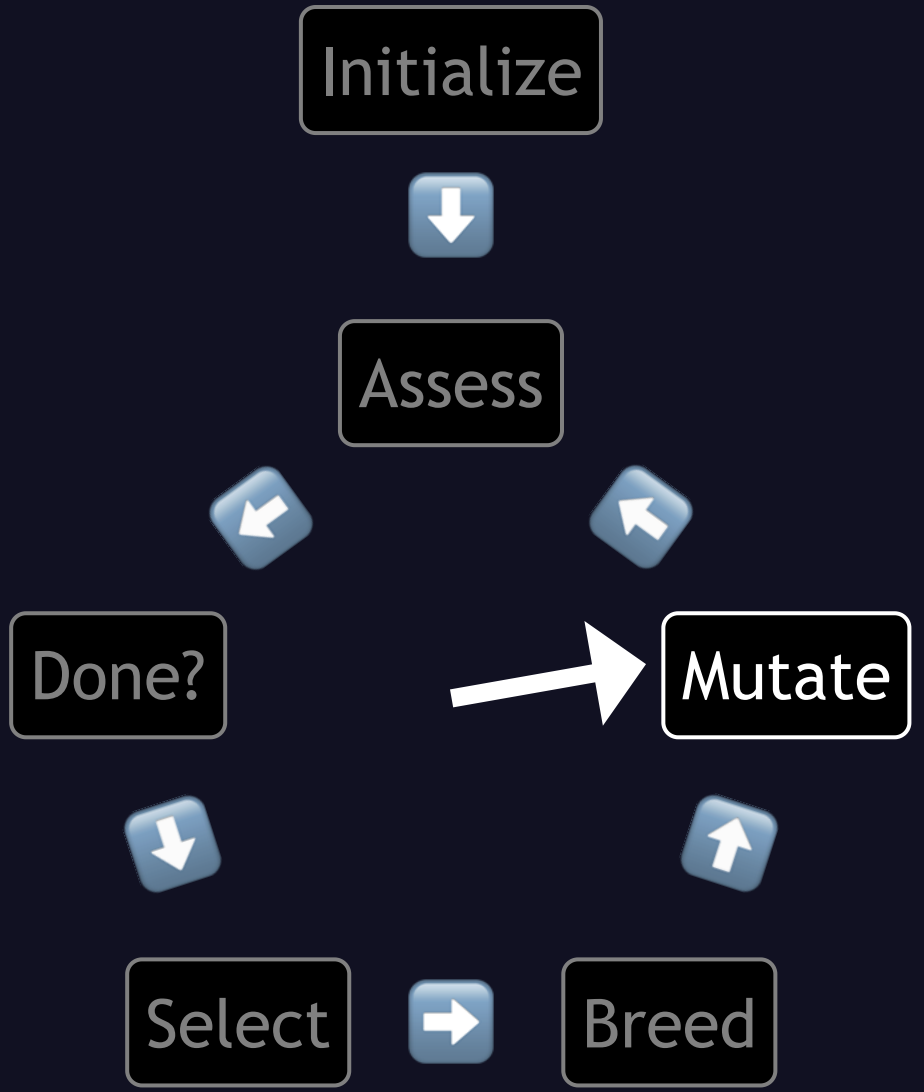
Speaker notes

. . . this, with Madrid, Luxembourg, and Brussels copied straight down, and then Andorra la Vella, Paris, Lisbon, Amsterdam, and Monte Carlo filled in in the order they appear in the second parent. It happens to match their order in the first parent, but that's actually unusual, and they're not consecutive in the second one, interrupted by Madrid. Do you see how that works? (PAUSE FOR CONFIRMATION, EXPLAIN IF NEEDED.) Next, as you may recall, this is just one of ten results, as we're making a whole new population. The way we would do that would be exactly the same as before, so I won't bore you with the code, but One possible result might be . . .

1st	2nd	3rd	4th	5th	6th	7th	8th
MAD	LUX	BRU	AND	PAR	LIS	AMS	MTC
MAD	LUX	BRU	AND	PAR	LIS	AMS	MTC
MAD	LUX	BRU	AND	PAR	LIS	AMS	MTC
LUX	BRU	AND	MAD	PAR	LIS	AMS	MTC
MAD	LUX	BRU	AMS	AND	PAR	LIS	MTC
MAD	LUX	BRU	AND	PAR	LIS	AMS	MTC
MAD	LUX	BRU	AND	PAR	LIS	AMS	MTC
MAD	LUX	BRU	AND	PAR	LIS	AMS	MTC
LUX	BRU	AND	MAD	PAR	LIS	AMS	MTC
LUX	BRU	AND	MAD	PAR	LIS	AMS	MTC

Speaker notes

. . . this. This time we have not only strong family resemblance, but a fair bit of full duplication too. If you want to continue the biological analogy, you could call them twins, triplets, and so on. But we get some more variety in the next step:


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

mutation. So how do we mutate a Traveling Salesman route? It's actually pretty easy:

```
def mutate()
  i1 = rand(CITIES.length)
  i2 = rand(CITIES.length)
  stops[i1], stops[i2] = [stops[i2], stops[i1]]
end
```

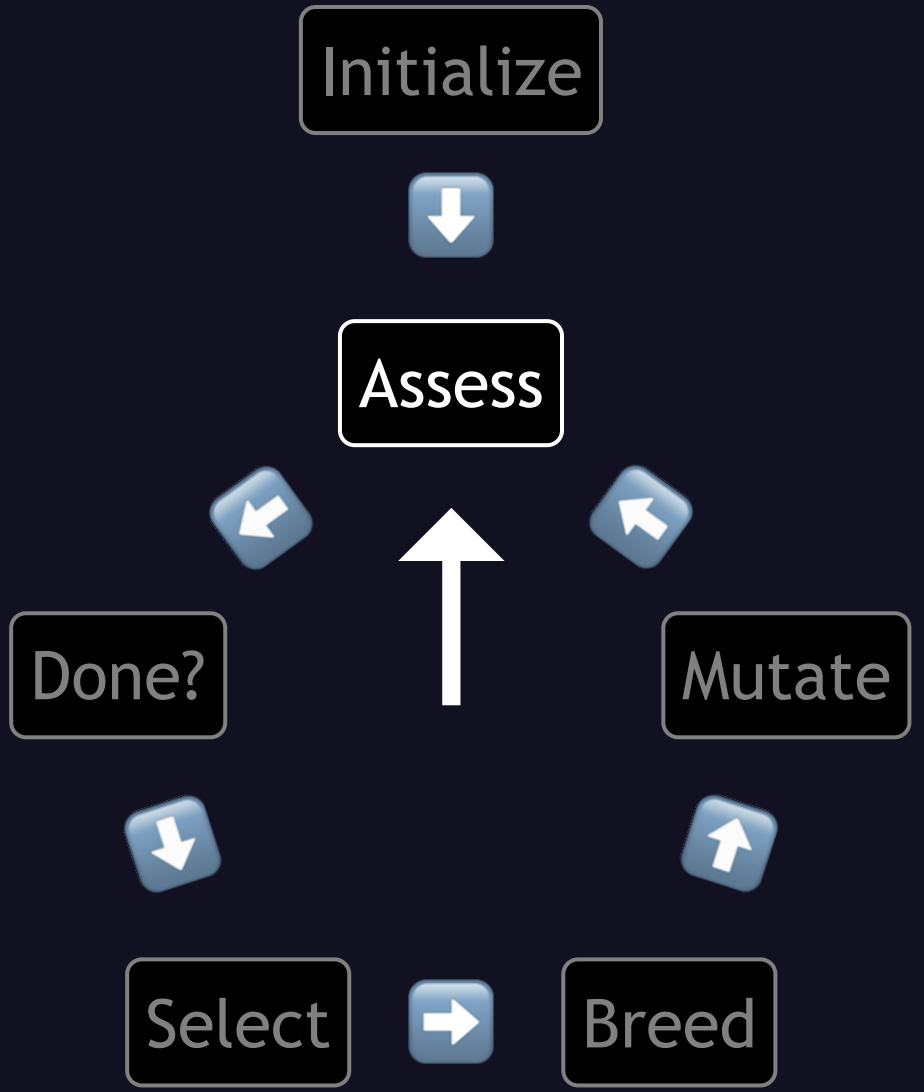
Speaker notes

we just pick two indices in the array, and swap the cities there. If we wanted to make the code more complex, we could make some cities or indices more or less likely to swap, or require that they be consecutive, or not consecutive, or we could have a probability of multiple mutations, but we're going to keep it simple and just do one swap each. The chance of the two numbers being the same will give us some chance of not really mutating. If we apply this to our current routes, we might wind up with . . .

1st	2nd	3rd	4th	5th	6th	7th	8th
BRU	LUX	MAD	AND	PAR	LIS	AMS	MTC
MAD	LUX	BRU	MTC	PAR	LIS	AMS	AND
MAD	LUX	BRU	AND	PAR	LIS	AMS	MTC
LUX	BRU	AND	MAD	PAR	LIS	AMS	MTC
MAD	LUX	BRU	AMS	AND	PAR	LIS	MTC
LIS	LUX	BRU	AND	PAR	MAD	AMS	MTC
AMS	LUX	BRU	AND	PAR	LIS	MAD	MTC
PAR	LUX	BRU	AND	MAD	LIS	AMS	MTC
LUX	BRU	MTC	MAD	PAR	LIS	AMS	AND
LUX	BRU	AND	PAR	MAD	LIS	AMS	MTC

Speaker notes

. . . this, where the green color means it changed. You can see that there are still many that somewhat resemble others, but there's no more exact duplication. That's not guaranteed though; some twins could undergo identical mutation, or some may mutate into twinship, though we could guarantee it with a much more complicated mutation function. (So long, of course, as we're using a small enough population.) Now that they're in their final forms, we can . . .


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

. . . start our cycle over again by asking how fit they are, and we get . . .

1st	2nd	3rd	4th	5th	6th	7th	8th	Fit
PAR	LUX	BRU	AND	MAD	LIS	AMS	MTC	4420
AMS	LUX	BRU	AND	PAR	LIS	MAD	MTC	4302
LUX	BRU	AND	PAR	MAD	LIS	AMS	MTC	3194
MAD	LUX	BRU	AMS	AND	PAR	LIS	MTC	2831
LUX	BRU	AND	MAD	PAR	LIS	AMS	MTC	2329
BRU	LUX	MAD	AND	PAR	LIS	AMS	MTC	2057
MAD	LUX	BRU	MTC	PAR	LIS	AMS	AND	2027
LUX	BRU	MTC	MAD	PAR	LIS	AMS	AND	1543
MAD	LUX	BRU	AND	PAR	LIS	AMS	MTC	1420
LIS	LUX	BRU	AND	PAR	MAD	AMS	MTC	1329

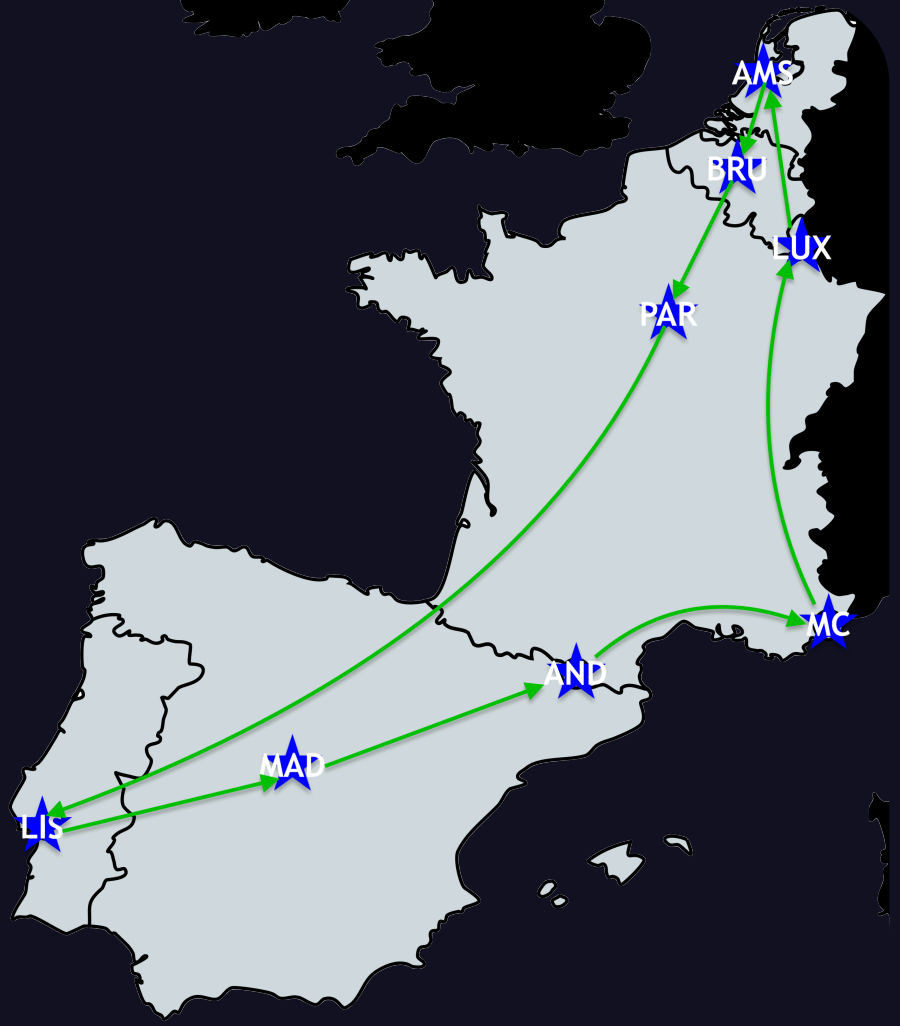
Speaker notes

. . . this, after sorting by fitness. Just as before, our best fitness went down! But, also just like before, we haven't lost that best one, it's recorded in that best_route class variable. Let's let it run to completion, producing . . .

1st	2nd	3rd	4th	5th	6th	7th	8th	Fit	Gens
LIS	AND	MTC	LUX	PAR	BRU	AMS	MAD	5802	22
LUX	BRU	PAR	MTC	AND	MAD	LIS	AMS	6012	18
LIS	MAD	PAR	AMS	BRU	LUX	MTC	AND	6275	26
AMS	BRU	PAR	LIS	MAD	AND	MTC	LUX	6424	100

Speaker notes

. . . these additional best routes, with those fitnesses, that stay as the best for that many generations. The total count of generations is well under a thousand, so it turns out we didn't really need that, especially since only the final best one lasted anywhere near 100 generations. On a map, that final best one, that reigned supreme for a hundred generations, would look like . . .


AMS
BRU
LUX
PAR
MC
AND
MAD
LIS

Speaker notes

. . . this, and we can see it really makes sense, it's roughly what we would have thought of just by eyeballing it.

This sequence of cities would have the same fitness if we started in any of them, and went in either direction first, so there are in fact sixteen equivalent routes. As I mentioned before, we could have dictated a starting city, whether one on this list or not, and the cost might not really be the same in either direction, plus there may be further complications such as how long we need to stay in each city, which could affect the best route to the next one and the time of arrival at the next one, which could how affect many hotel nights we need, which of course contributes to the total cost. So, realistically, it could almost certainly be narrowed down to one, or at least far fewer than sixteen, but at the cost of a far more complicated fitness function. There are . . .

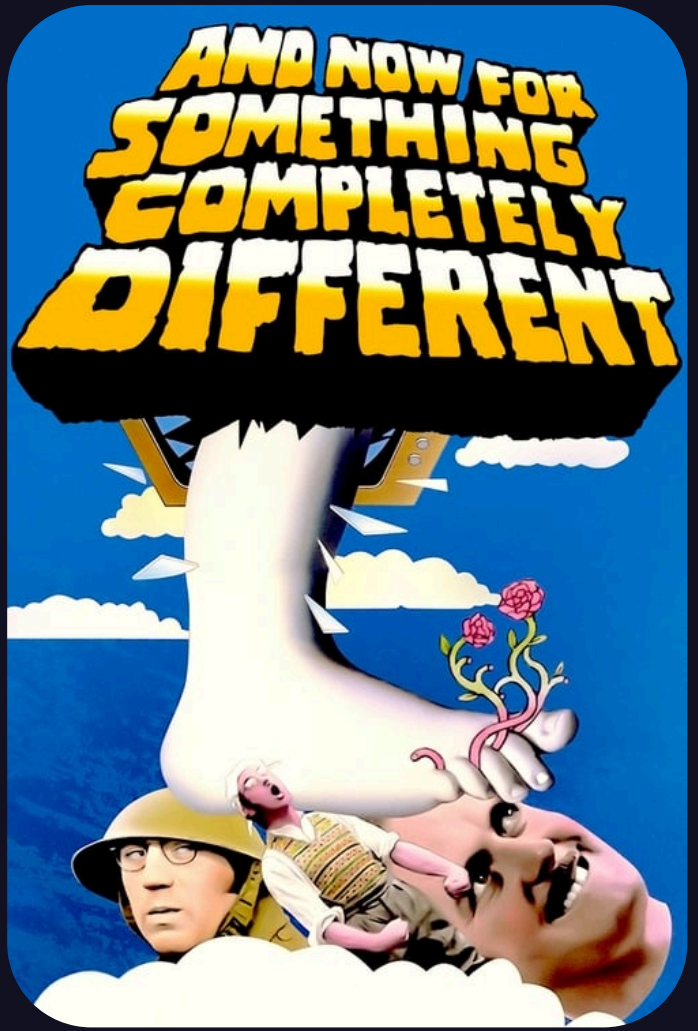


Image: https://image.tmdb.org/t/p/w500/ajbdFQLvJTlNu4LnVWGnNMb4mZ8.jpg (used for edu fair use)

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:

```ruby
class Character
  attr_reader %i(str int dex con wis cha)
  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
end
```

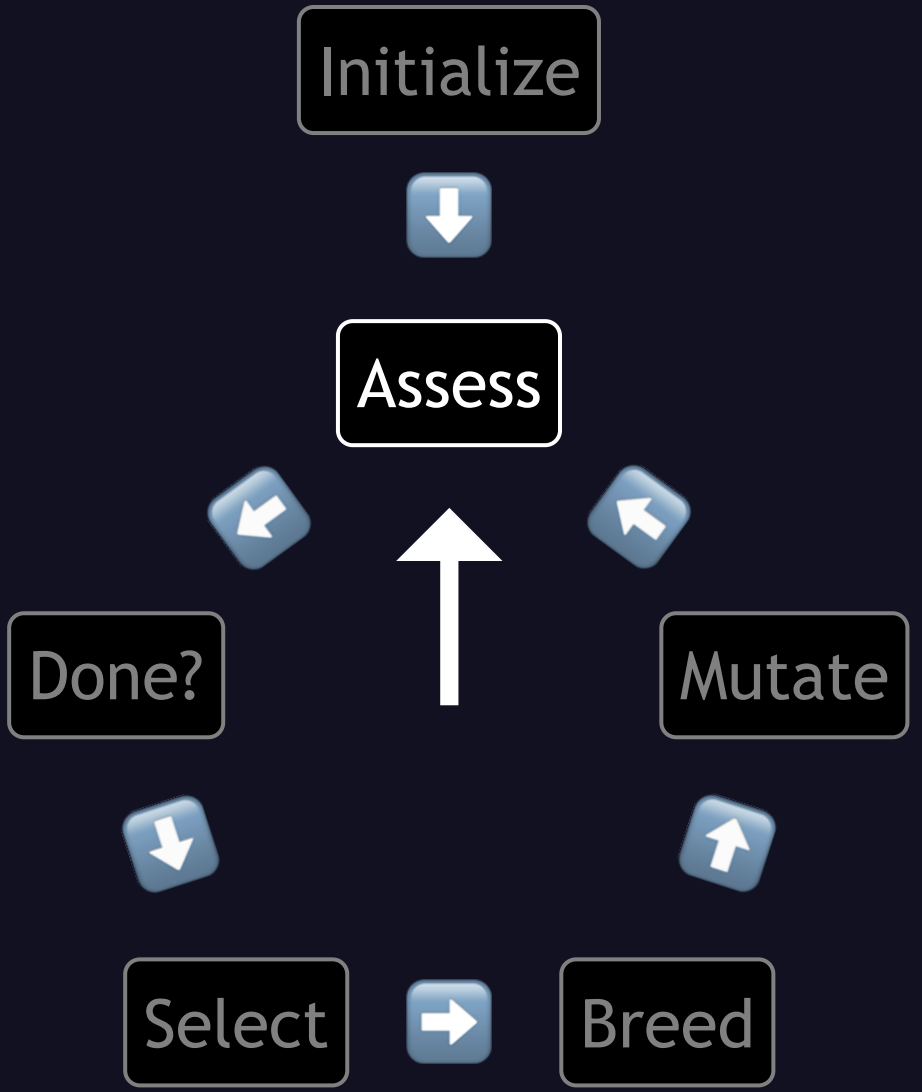
Speaker notes

(Defining the roll function is left as an exercise, and yes there's a great heuristic for D&D character creation, just assign 18s across the board and 00 for the extra strength, but . . . that's cheating! Let's be realistic.) And we could complicate this by having a minimum, maximum, or specific total number of points, or minimum and maybe maximum score in each stat, but as usual we'll just keep it simple. 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

So that's the Initialize step. 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 all of 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

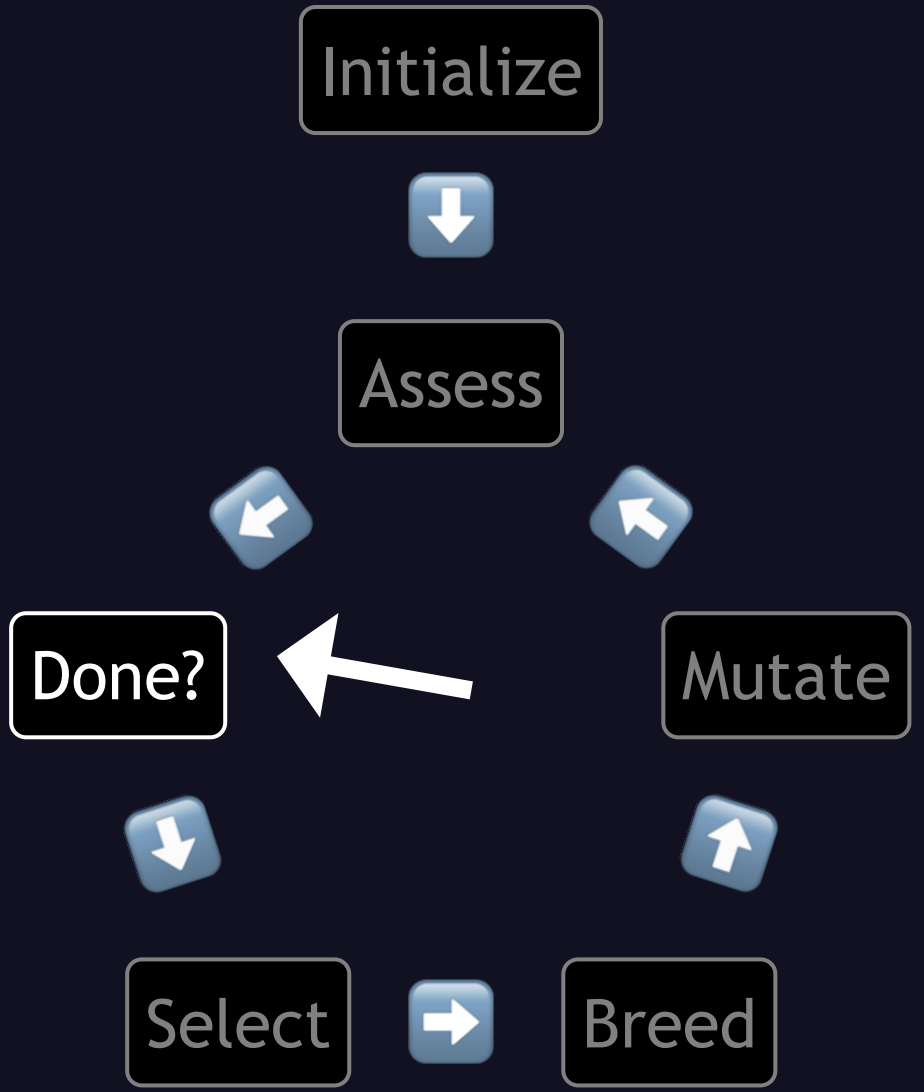
. . . prioritizing the stats again but much more strongly, totaling up 32 times the strength, 16 times the constitution, and so on down to one times the charisma. Here we see that even though the fitness function itself can be very simple, it can be difficult to figure out one that will yield good results,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

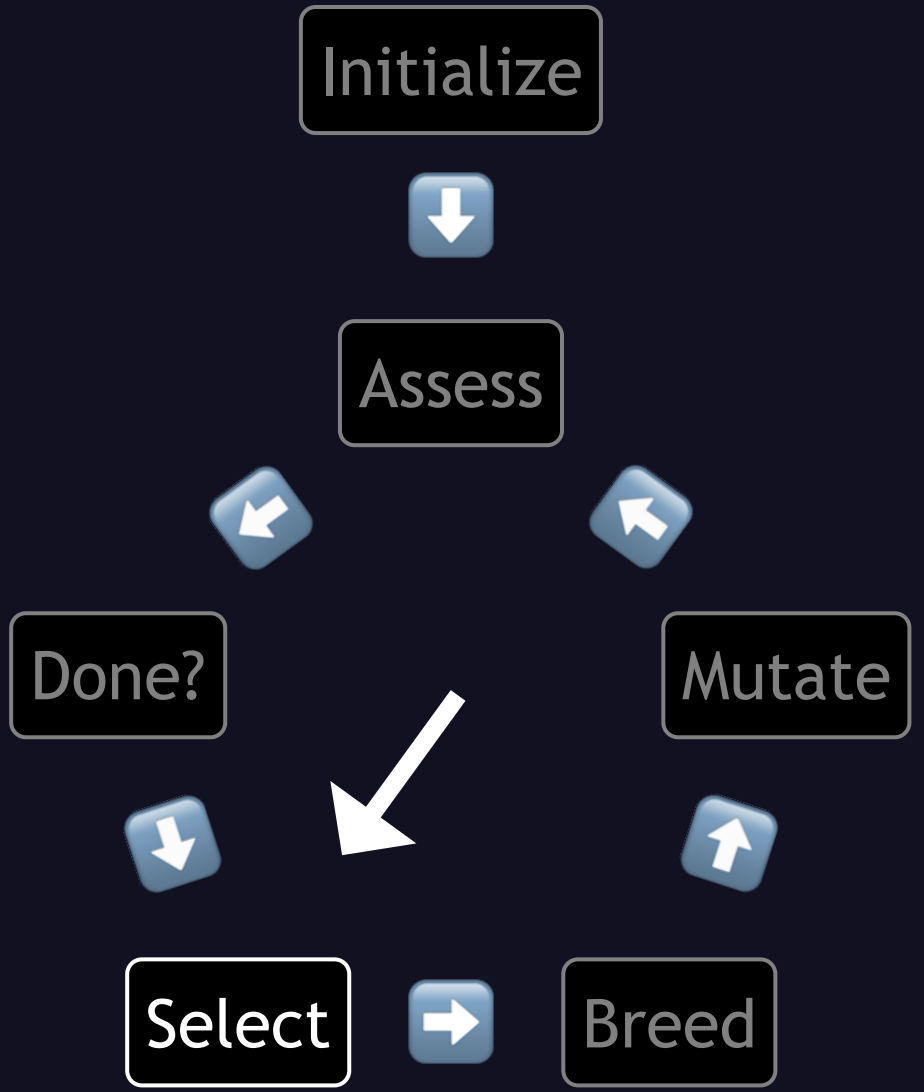
Speaker notes

. . . 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 most of the math, but the maximum is 1,134, so 90% would round to 1,021. In code, checking that would look like this:

```
def Character.done?(population)
  population.any? { |cand|
    cand.fitness >= 1021
  }
end
```

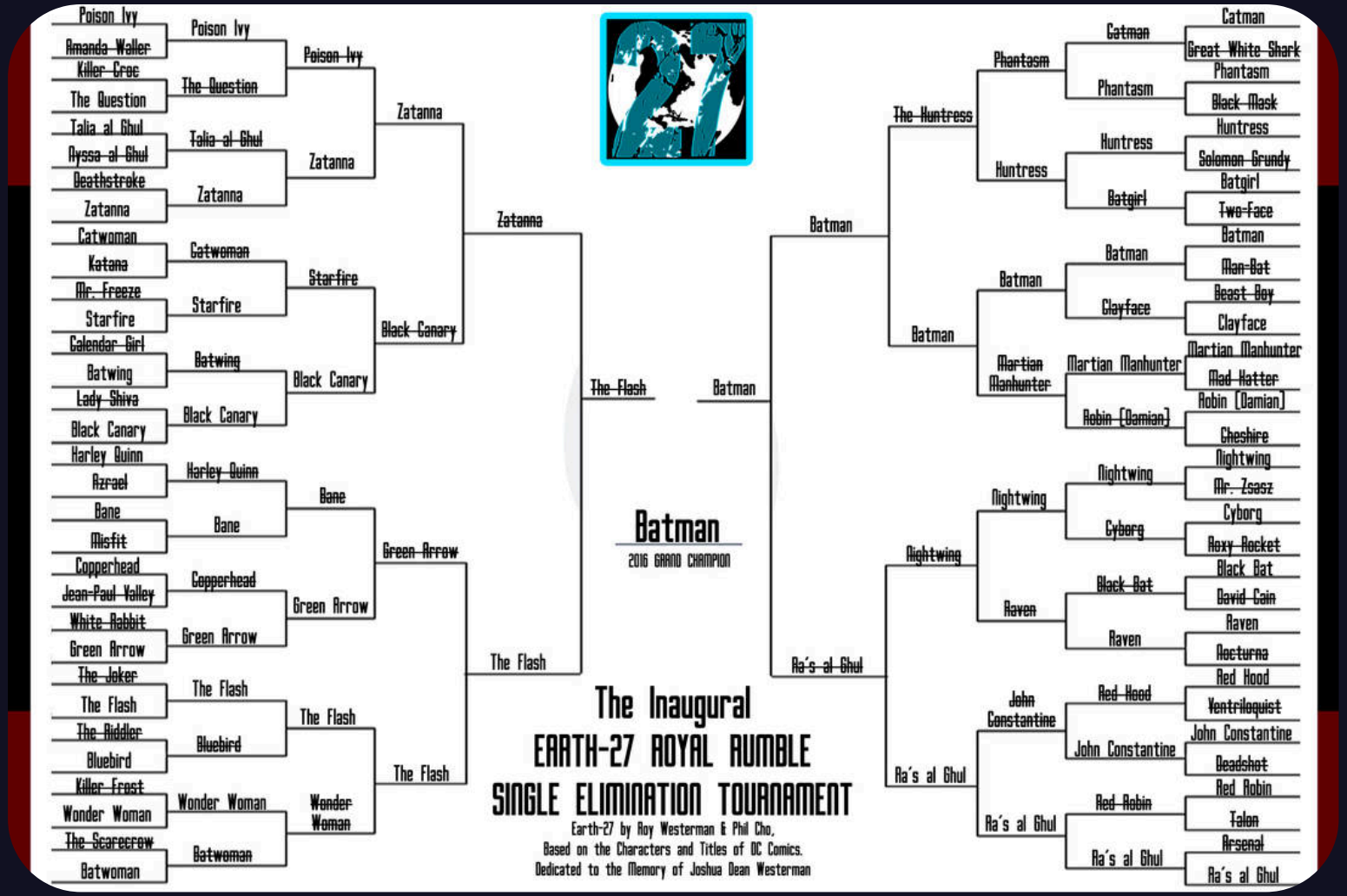
Speaker notes

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. This time we're going to do . . .


Poison Ivy
Amanda Waller
Killer Croc
The Question
Talia al Ghul
Ayssa al Ghul
Deathstroke
Zatanna
Catwoman
Katana
Mr. Freeze
Starfire
Calendar Girl
Batwing
Lady Shiva
Black Canary
Harley Quinn
Azrael
Bane
Misfit
Copperhead
Jean-Paul Valley
White Rabbit
Green Arrow
The Joker
The Flash
The Riddler
Bluebird
Killer Frost
Wonder Woman
The Scarecrow
Batwoman
Poison Ivy
The Question
Talia al Ghul
Zatanna
Catwoman
Starfire
Batwing
Black Canary
Harley Quinn
Bane
Copperhead
Green Arrow
The Flash
Bluebird
Wonder Woman
Batwoman
Poison Ivy
Zatanna
Starfire
Black Canary
Bane
Green Arrow
The Flash
Wonder Woman
Zatanna
Black Canary
Green Arrow
The Flash
Zatanna
The Flash
Batman
2016 GRAND CHAMPION
Batman
Ra's al Ghul
Batman
The Huntress
Batman
Nightwing
Ra's al Ghul
Phantasm
Huntress
Batgirl
Batman
Clayface
Martian Manhunter
Robin (Damian)
Nightwing
Cyborg
Raven
John Constantine
Ra's al Ghul
Catman
Phantasm
Huntress
Batgirl
Batman
Clayface
Martian Manhunter
Robin (Damian)
Nightwing
Cyborg
Raven
John Constantine
Ra's al Ghul
Catman
Great White Shark
Phantasm
Black Mask
Huntress
Solomon Grundy
Batgirl
Two-Face
Batman
Man-Bat
Beast Boy
Clayface
Martian Manhunter
Mad Hatter
Robin (Damian)
Cheshire
Nightwing
Mr. Zsasz
Cyborg
Roxy Rocket
Black Bat
David Cain
Raven
Nocturna
Red Hood
Ventriloquist
John Constantine
Deadshot
Red Robin
Talon
Arsenal
Ra's al Ghul
The Inaugural
EARTH-27 ROYAL RUMBLE
SINGLE ELIMINATION TOURNAMENT
Earth-27 by Roy Westerman & Phil Cho.
Based on the Characters and Titles of DC Comics.
Dedicated to the Memory of Joshua Dean Westerman

Speaker notes

. . . tournament selection. This is somewhat like Roulette Wheel selection, in that it involves an element of chance, but it gives much more of an edge to the fitter candidates. It works by first . . .

[X] Alice
[ ] Bob
[X] Charlie
[ ] Darlene
[X] Edwina
[ ] Felicity
[ ] Georgette
[X] Harry
[ ] Ian
[ ] Jackie

Speaker notes

. . . selecting some number of candidates at random, choosing the . . .


35
IRON GRIP

Speaker notes

. . . fittest of those as a breeder (this is why they call it "Tournament" selection), and then . . .

Image: https://www.streamlinehq.com/icons/download/refresh-circle--29591 w/ colors inverted

Speaker notes

. . . repeating that as needed to get as many breeders as we want. We're going to stick with just taking two breeders, though. Using tournaments of four, to get two breeders, might look like this:

```
# in our main loop
breeders = self.select_breeders(pop, 4, 2)
```

Speaker notes

In our main loop, we call select_breeders. In this class, it's implemented tournament-style, so we need to tell it what size of tournament to use, and how many breeders we want. (Or of course we could make those things class constants, or even hardcode them in these functions, whatever.) In the actual breeder selection function, we . . .

```
def self.select_breeders(pop,
                         tourney_size,
                         num_breeders)
  breeders = []
  while breeders.count < num_breeders
    breeders.append(run_tourney(pop - breeders,
                                tourney_size))
  end
  return breeders
end
```

Speaker notes

. . . start with an empty array of breeders, and so long as we don't have enough, we append one more, chosen from among the ones that haven't yet been chosen. And how they're chosen is . . .

```
def self.run_tourney(left, tourney_size)
  return left.
    sample(tourney_size).
    sort(&:fitness).
    last
end
```

Speaker notes

. . . simply to get a sample of the appropriate size, sort them by fitness, and take the last.

To step through it working, suppose our first call to run_tourney happens to pick . . .

Str	Int	Dex	Con	Wis	Cha	Fit
11	9	9	10	7	15	649
14	15	10	7	6	14	726
4	14	8	12	13	10	476
14	12	13	5	13	13	719

Speaker notes

these four. They're no longer in order because the sample function doesn't necessarily keep order. But then run_tourney will . . .

Str	Int	Dex	Con	Wis	Cha	Fit
4	14	8	12	13	10	476
11	9	9	10	7	15	649
14	12	13	5	13	13	719
14	15	10	7	6	14	726

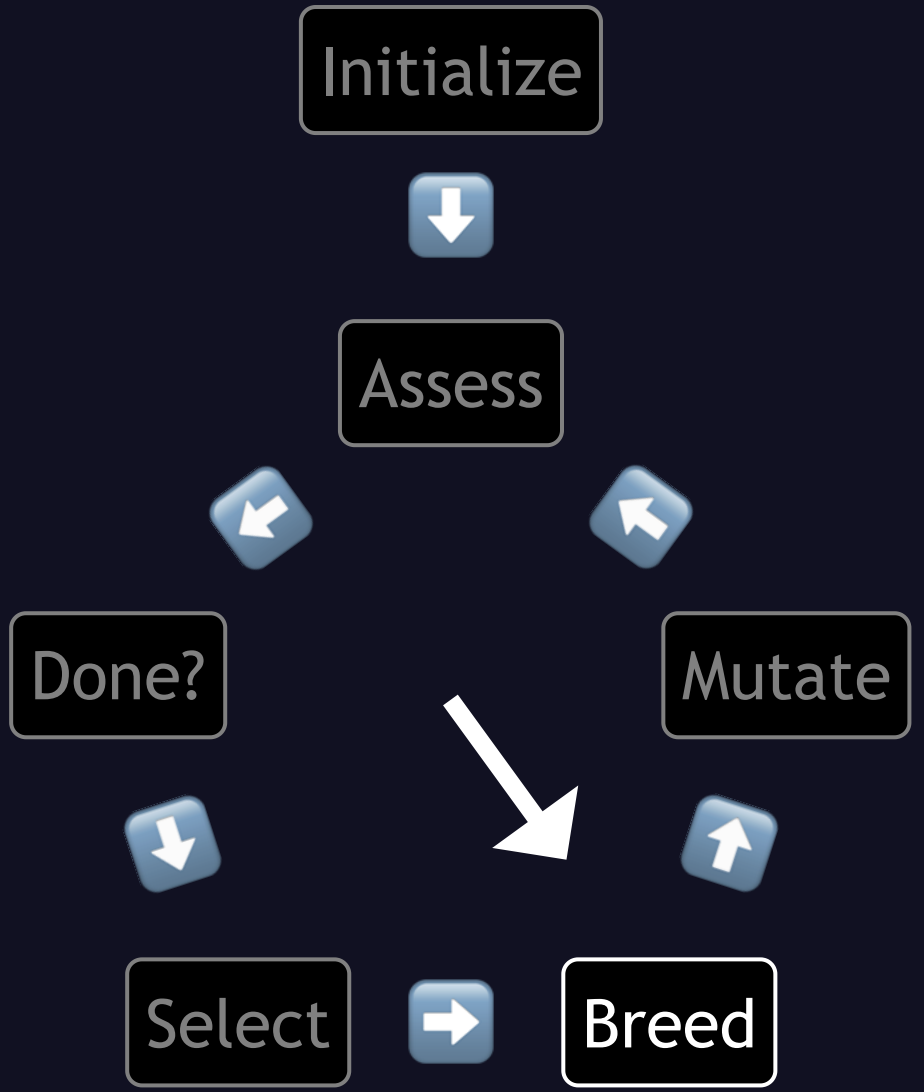
Speaker notes

sort them by fitness, and return the last, in other words, the most fit. select_breeders puts that in our breeders array, so now we have one, but we wanted two, so it goes through that process again. However, this time, the array of candidates that it passes to run_tourney isn't the whole population, but only the ones that haven't yet been chosen, so this time that's everybody except that last one there. Let's assume that this time, run_tourney picks a set of four that includes the most fit one, so it returns that, select_breeders puts it in the array, which now has our desired size of two, so it gets returned to our main loop, and we wind up with . . .

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

Speaker notes

. . . these two for our breeders. These just happen to be our two most fit, but it's not at all guaranteed to turn out like that. Calculating the probability, given then population size, tourney size, and number of breeders, is left as an exercise for the reader. Also, The abstraction gets a bit more obviously leaky now, because we're ignoring sexes; we have no guarantee that these characters will be a male and a female, as we're not even making that part of the data. We had the same leak before, with the Truckloads and Routes, but then it was not so relevant, so I let it slide. Now that they're living beings (whether humans or elves or whatever), though, we could add such complications, and many other such factors. That would complicate our breeder selection enormously, maybe even make it impossible to find a viable pair in such a small population. Anyway, since we're not doing that, we don't have that concern, so we actually . . .


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

. . . breed our chosen pair. This time we'll be using another common strategy, of essentially flipping a coin for each gene, like so:

```ruby
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.int
  char.dex = rand(2) == 1 ? p1.dex : p2.dex
  char.con = rand(2) == 1 ? p1.con : p2.con
  char.wis = rand(2) == 1 ? p1.wis : p2.wis
  char.cha = rand(2) == 1 ? p1.cha : p2.cha
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

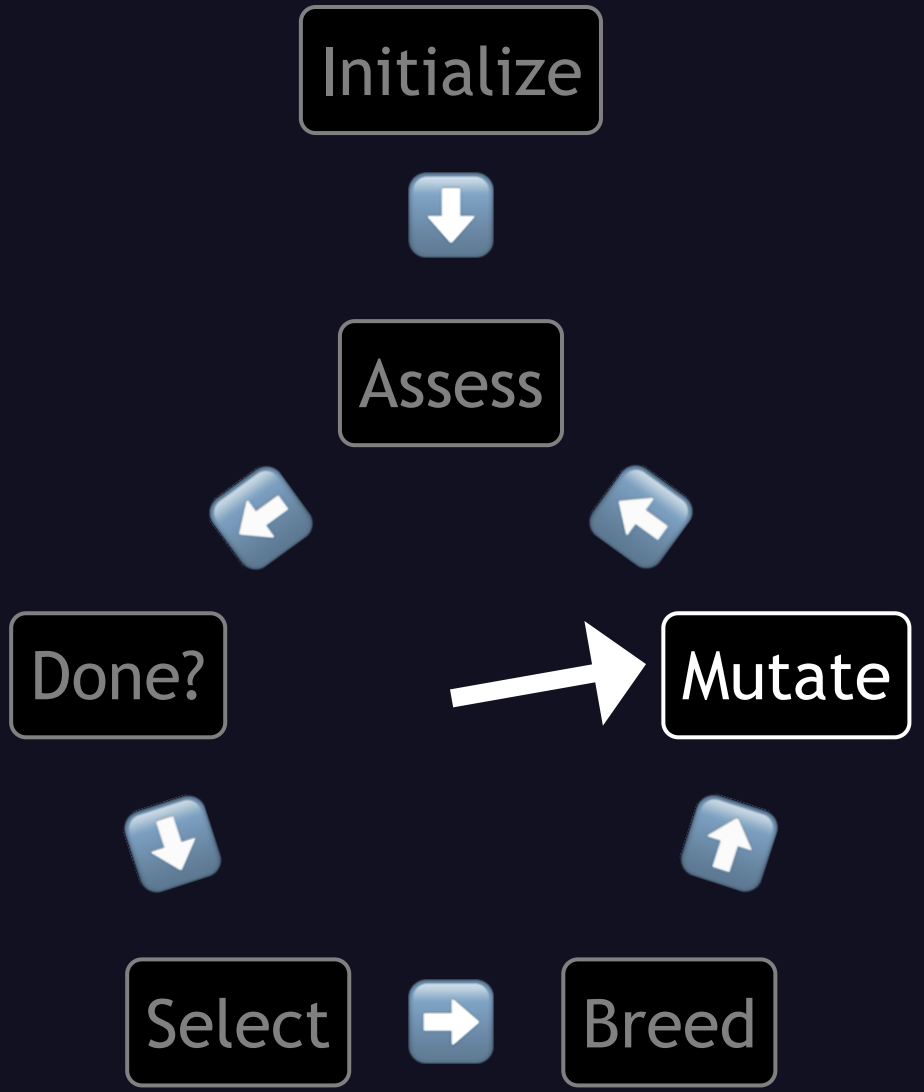
But again, this is just one of ten results, because we're making a whole new population, which might look something like this:

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

Notice the family resemblance again! Even though we're no longer using yes/no decisions, there are only two possible alleles for each gene, in other words two possible values for each stat, the ones in each of the breeders. That means a total of 2^6, or 64, possible combinations. There would be only one possible value, and therefore half as many possible combinations, if any alleles were the same between the two parents, for any given gene, as we saw with the Truckloads.

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, in other words, the alleles for each gene. 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
```

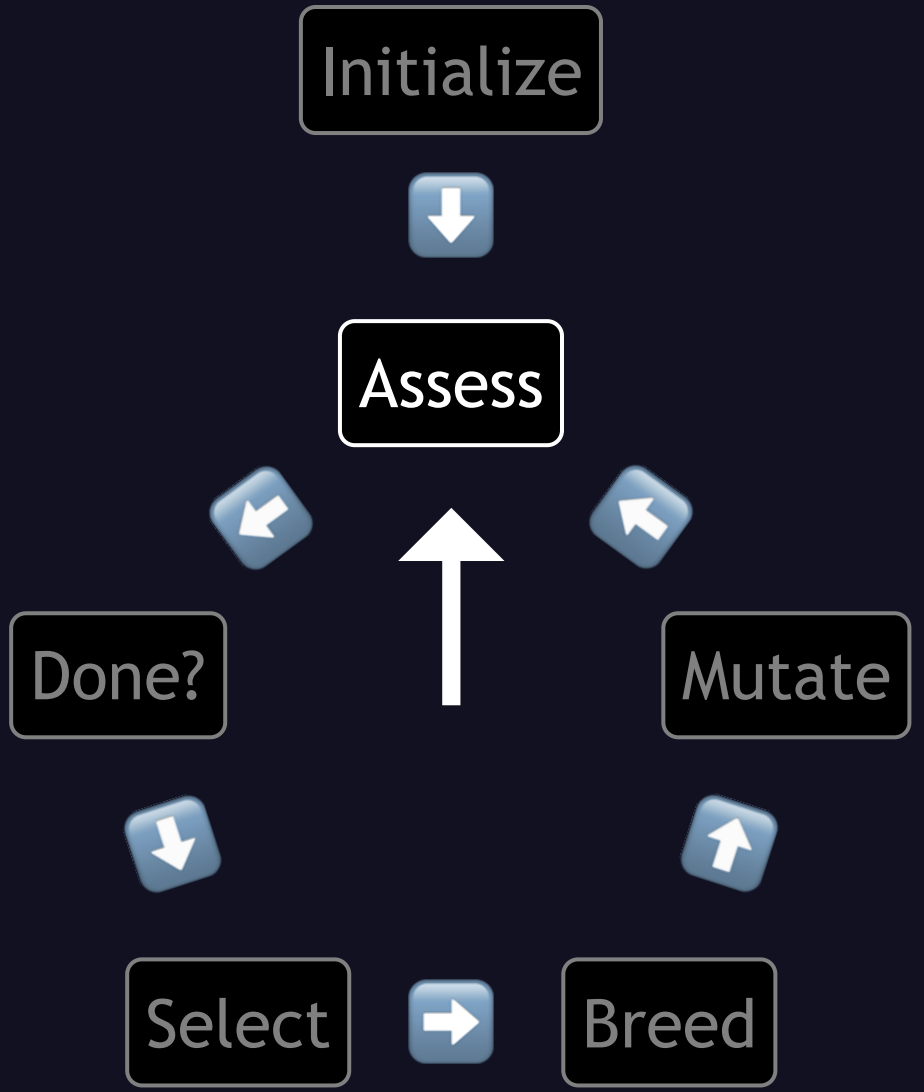
Speaker notes

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, to simulate the real-world phenomenon of regression to the mean, 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 it went down. Looking at the values in each column, you can see it's now much more diverse. Now we . . .


Initialize
Assess
Mutate
Breed
Select
Done?

Speaker notes

. . . start the cyle over, assessing 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. (We could also make it take parameters, and pass in different ones this time, but I'm just going to hardcode it here for simplicity.) 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.

Remember though that this is all very random. It may converge on a good solution faster, or more slowly, and the fitness function may be good or poor at getting just the right mix of alleles.

There are . . .

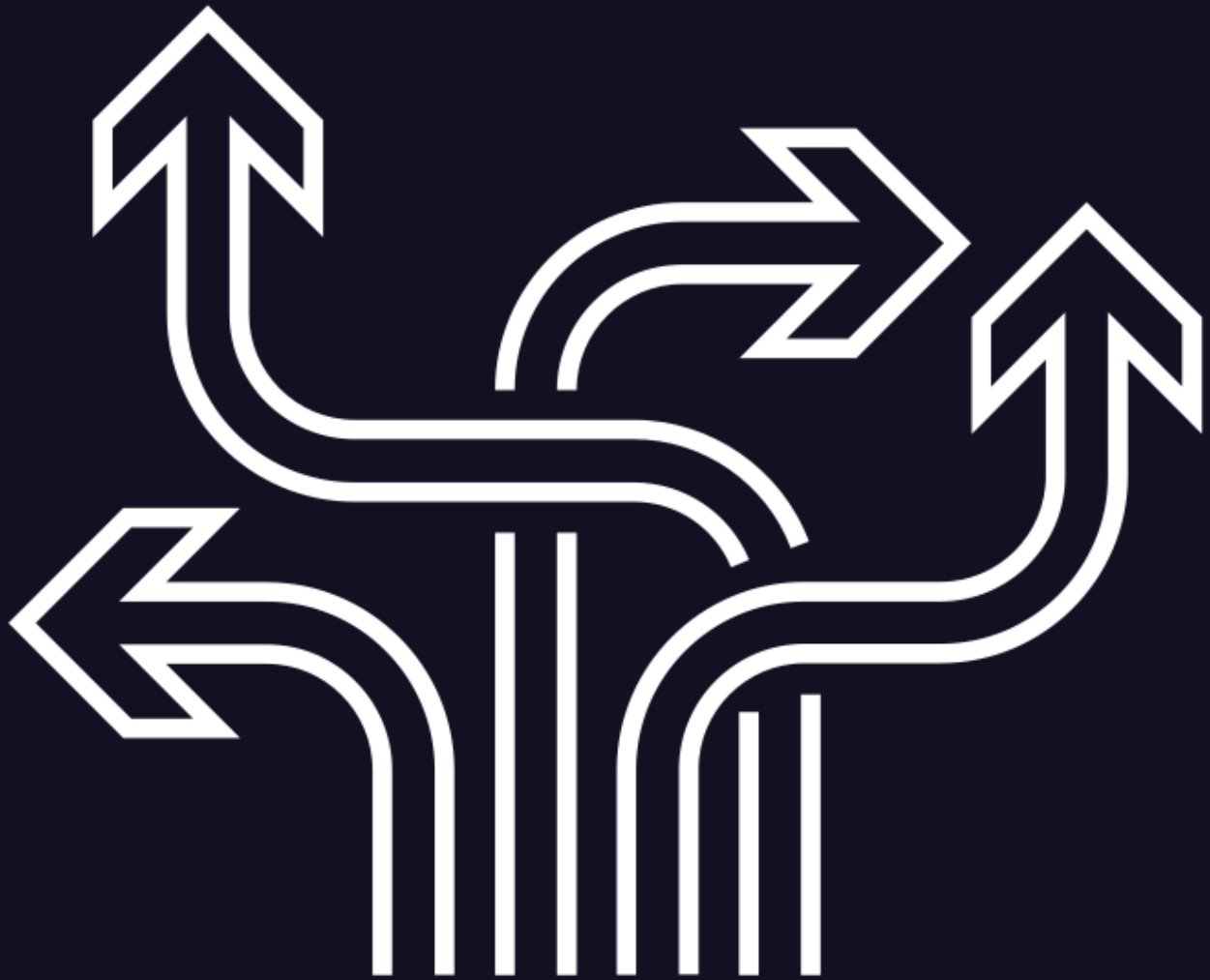


Image: https://commons.wikimedia.org/wiki/File:Noun_76041_-_Different_Ways.svg

Speaker notes

. . . many other ways we can use genetic algorithms. Longer versions of this talk show how to use them to generate Dungeons & Dragons character stats, and recipes for brewing mead. I'm now working on a system to use them to schedule the talks for a conference. Mey Beisaron has a talk on using one to schedule her college classes. They can create images and music (though the fitness functions would be rather difficult), even code (maybe with a fitness function consisting of passing tests). So, think about it, and you might be able to use them for something. If not, keep the idea in your toolchest, and maybe eventually you'll be called upon to code up something to solve problems that might be computationally intractable by normal means.

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 -- all you have to do is create an initial population, and cycle through five simple steps, of assessing their fitness, checking if you're done, picking breeders, breeding them, and mutating the new candidates, over and over until you're done. 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, for best results, especially for evaluating fitness and checking if you're done, so as to make the solutions converge quickly enough, and not ignore too many other very good solutions. 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-teqnation-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 (as you may have guessed) is for the slides as a PDF, complete with a full script... which I've mostly stuck to. Anyway, any questions?