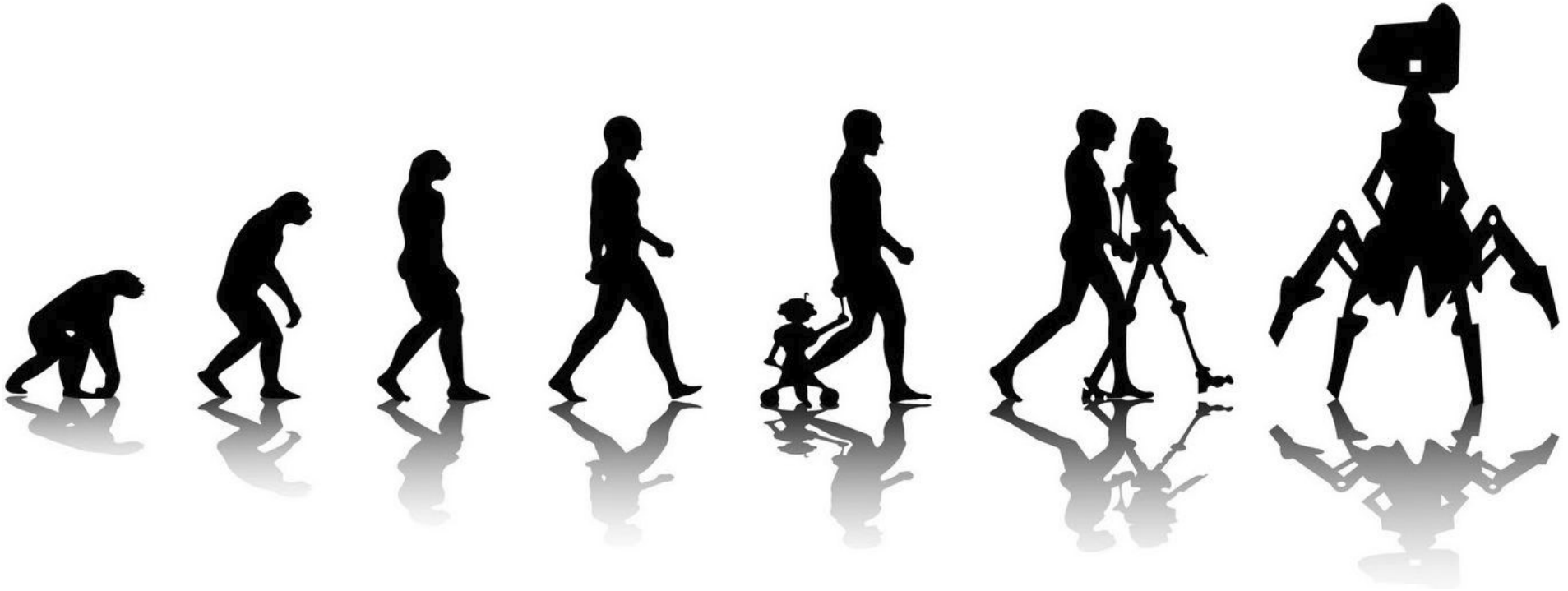


Modern Robotics: Evolutionary Robotics

COSC 4560 / COSC 5560

Professor Cheney

1/26/18

Intro to Artificial Evolution

Major pieces:

1) Genetic Variation

2) Evaluation

3) Selection

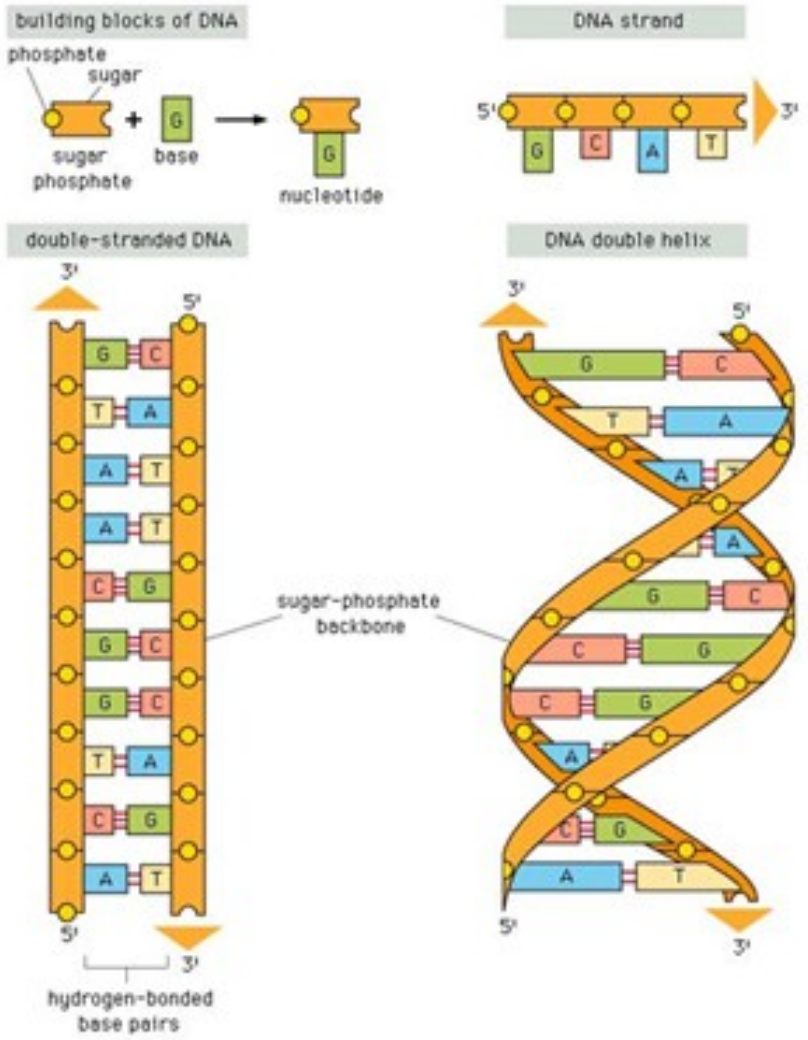
Genetic Variation

Random variations will be used
to propose new designs/solutions
that the algorithm will try out

How will we go about this?

Genetic Encoding

What is the DNA of our designs?
What are the basic building blocks?



What's the best encoding for artificial evolution?

This is a very hard problem with lots of active research!

We don't even understand all the properties of biological DNA!

What's the simplest genetic encoding we could use?

[0 0 1 0 1 1 0 1 1 0]

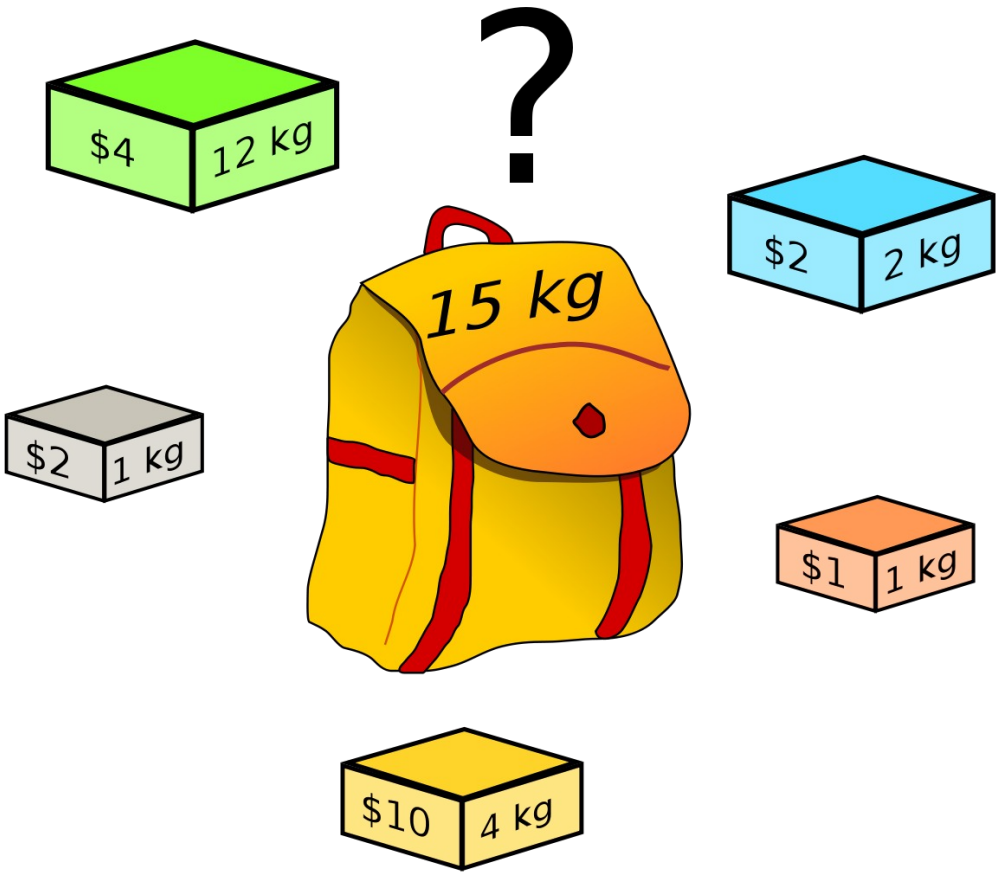
A vector encoding (e.g. a bit string)?

What could this represent?

Anything…

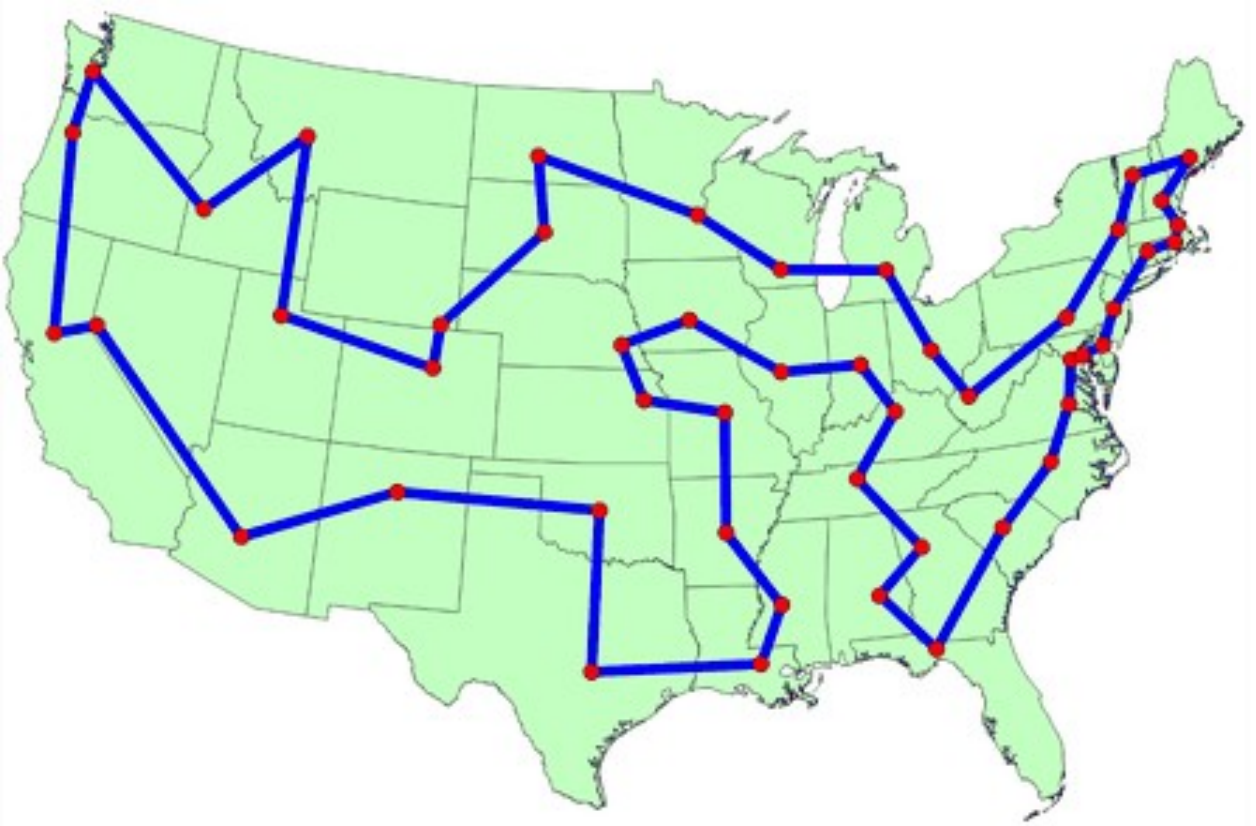
What could this represent?

Items included in a knapsack problem (binary)



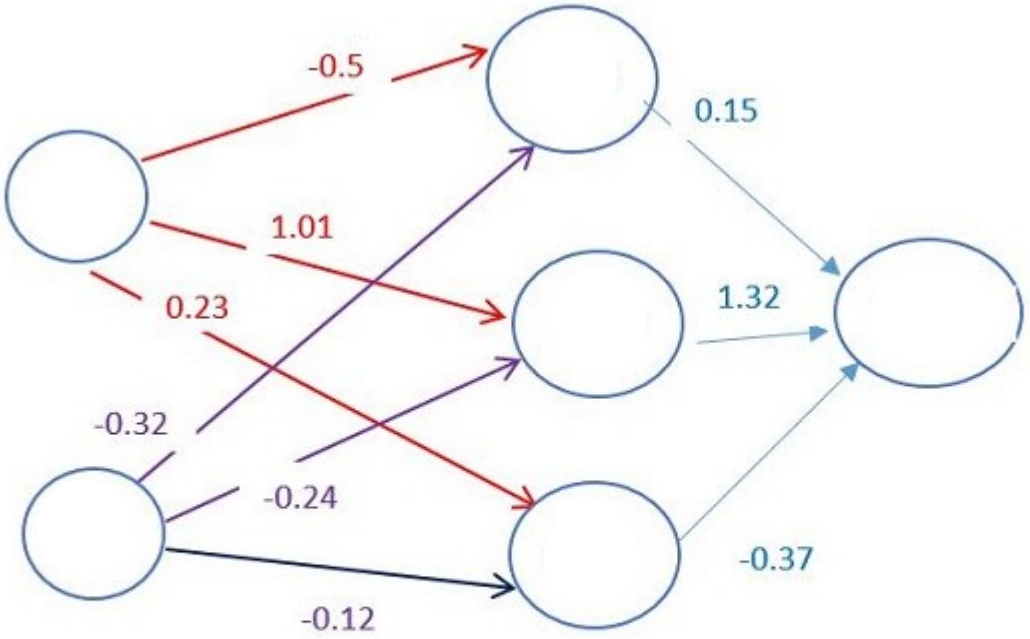
What could this represent?

Visitation order in a traveling salesman problem (int)

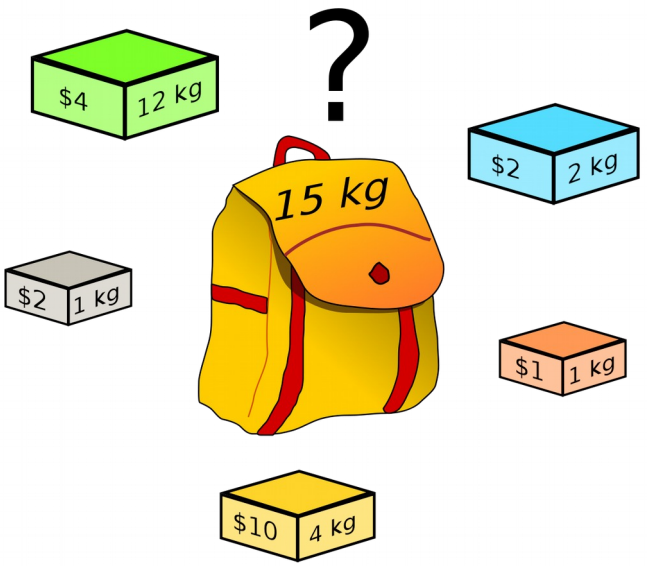


What could this represent?

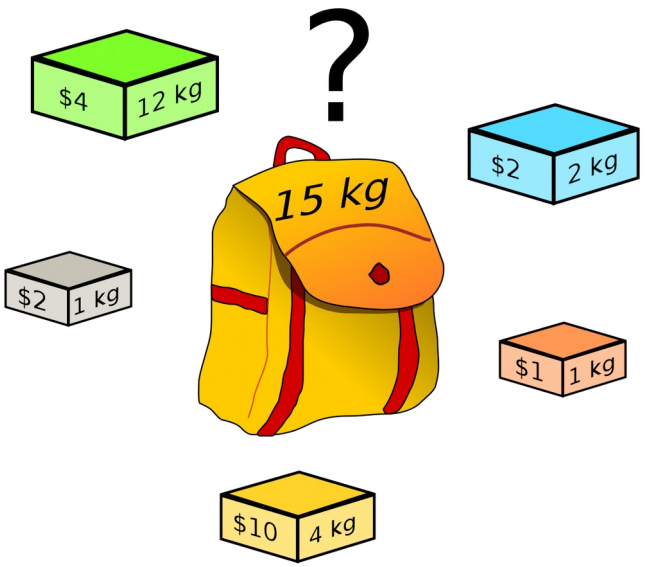
Weights in an artificial neural network (float)



Most problems can be represented in many different ways...

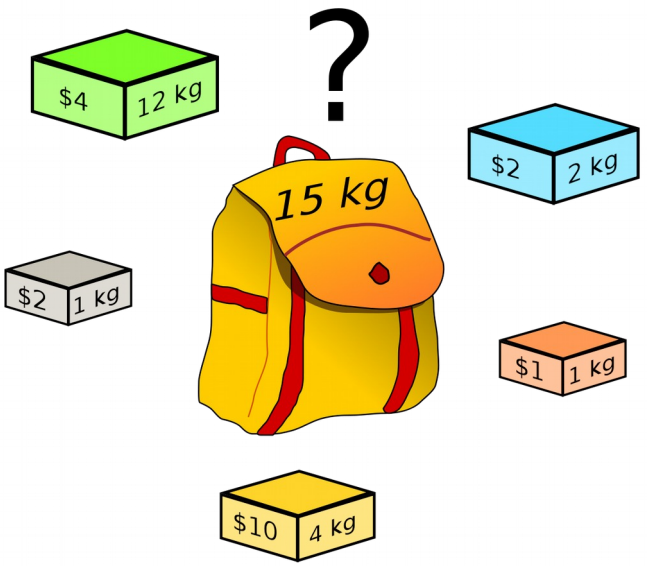


Integer list of which items to bring?

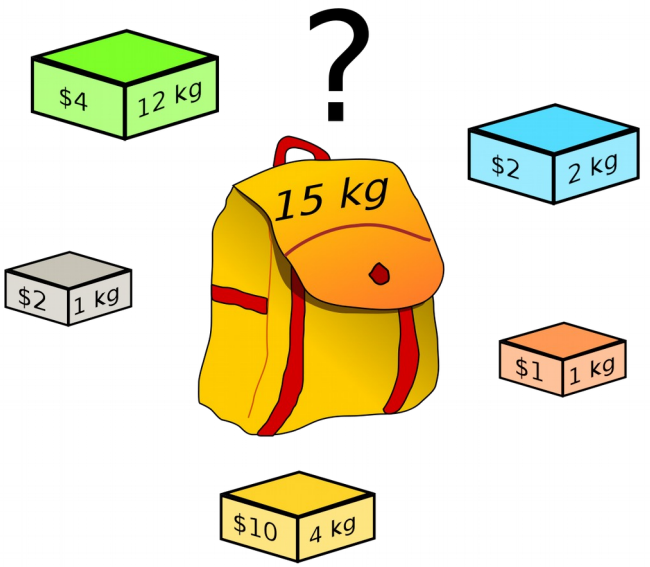


Binary list of inclusion or not for each item?

Which is better?



Integer list of which items to bring?



Binary list of inclusion or not for each item?

Which is better?

2 parameters:
volume of hot water,
volume of cold water

2 parameters:
volume of total water,
temperature of water

Depends on what you want from it!

Better for using a fixed amount of hot water

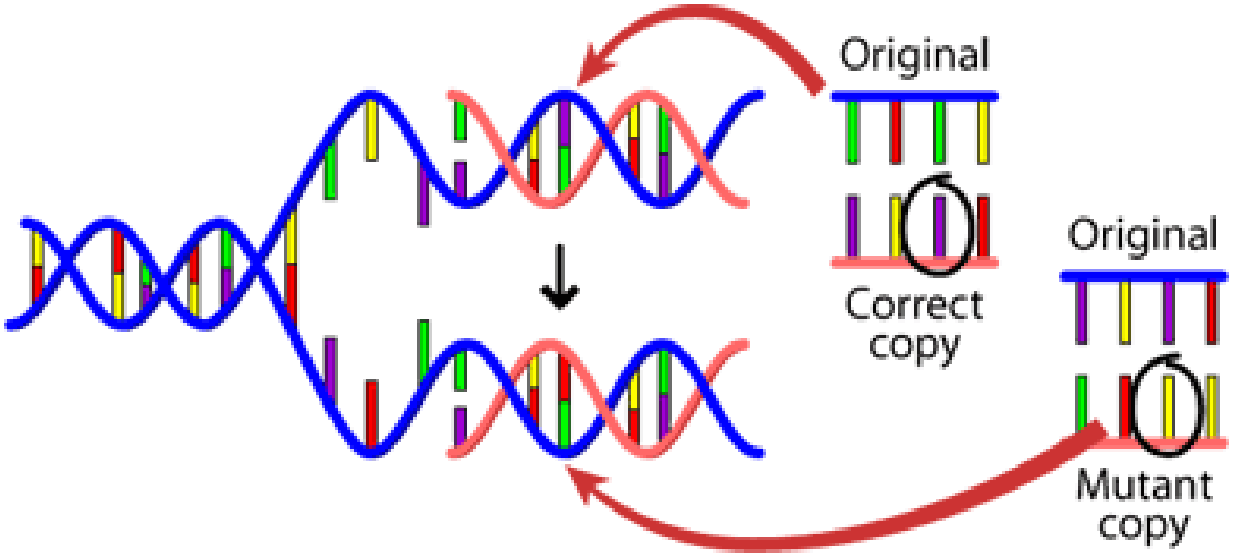
Better for keeping a constant temperature

But for now (and for you 1st HW assignment), let's just keep it as an abstract vector

[0 0 1 0 1 1 0 1 1 0]

Genetic Variations

Genetic Mutation:
(transcription error)



Genetic Mutation:

[0 0 1 0 1 1 0 1 1 0]

With some (small) probability, change each entry

[0 0 1 0 1 0 0 1 1 0]

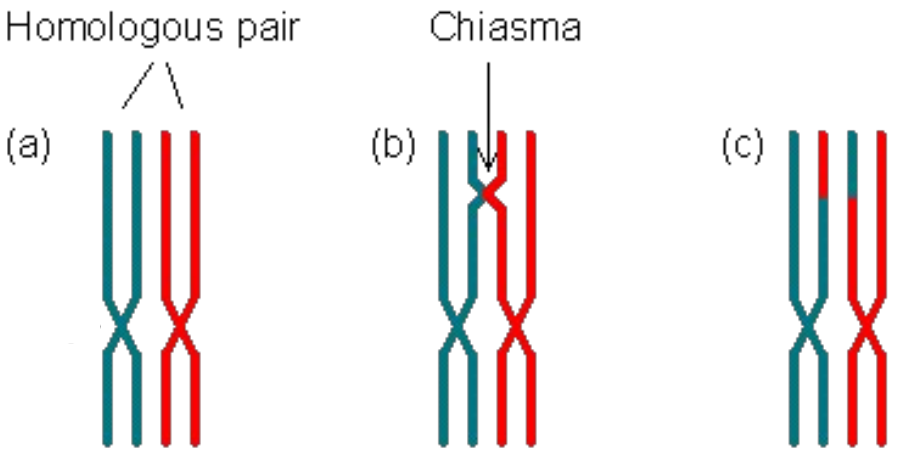
Genetic Mutation:

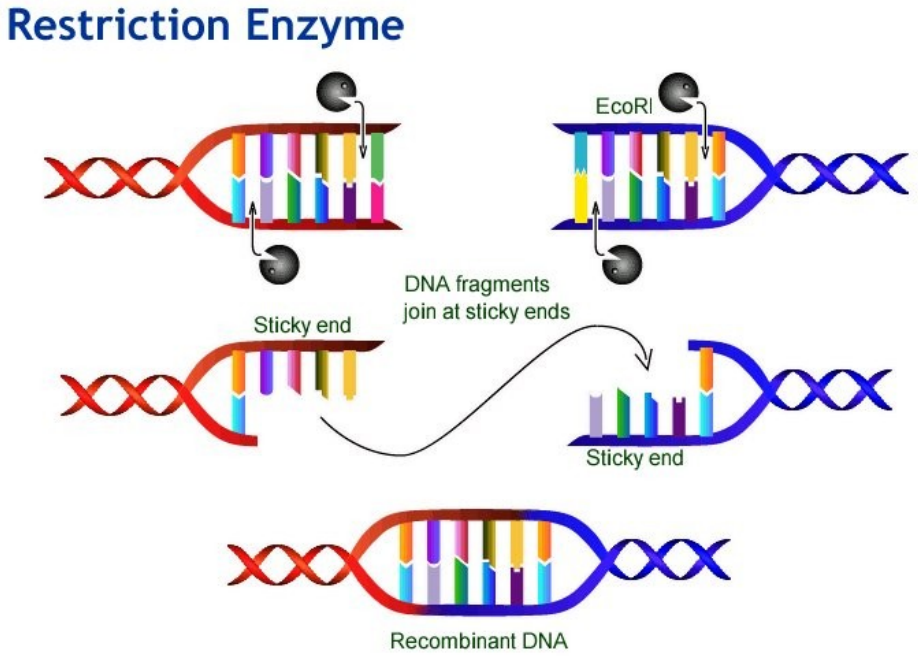
[0 0 1 0 1 1 0 1 1 0] “parent”

With some (small) probability,
change each entry

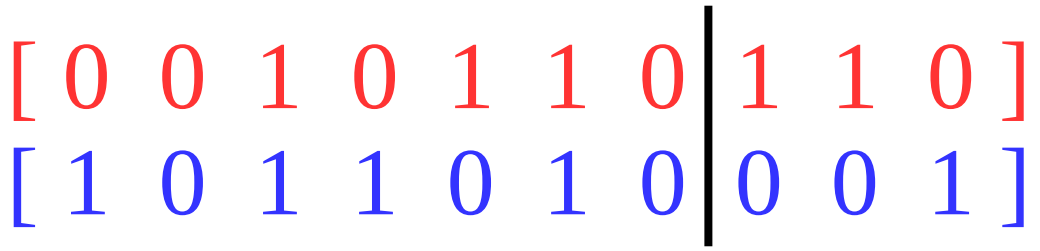
[0 0 1 0 1 0 0 1 1 0] “child”

Genetic Crossover: (homologous recombination)





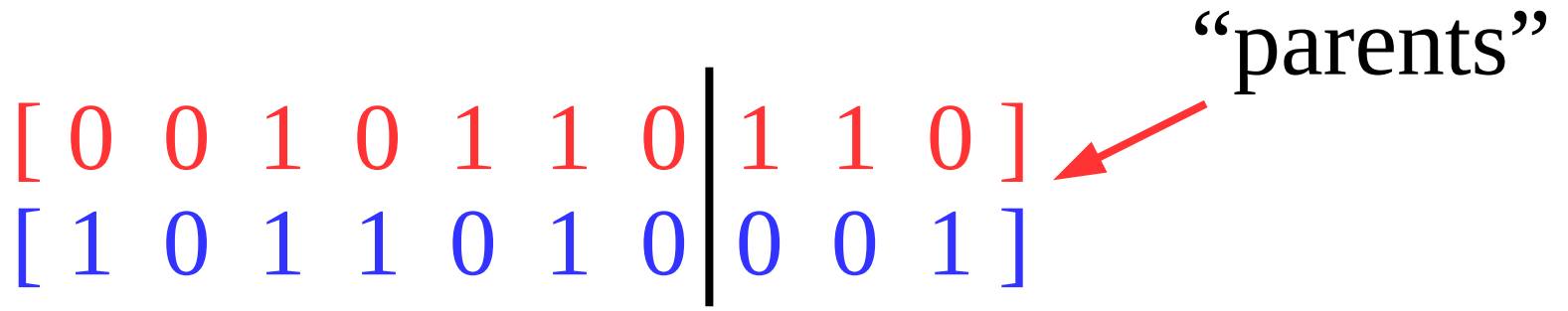
Genetic Crossover:



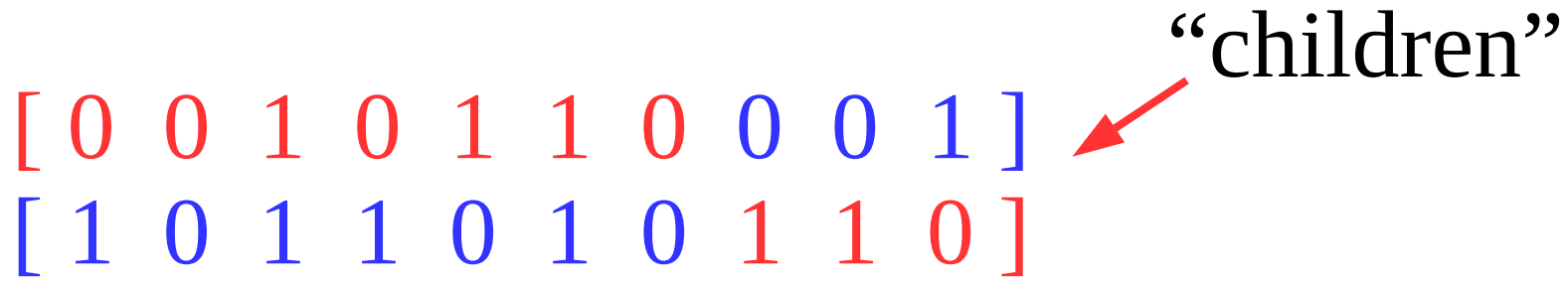
Pick a crossover point, and exchange genes

[0 0 1 0 1 1 0 0 0 1]
[1 0 1 1 0 1 0 1 1 0]

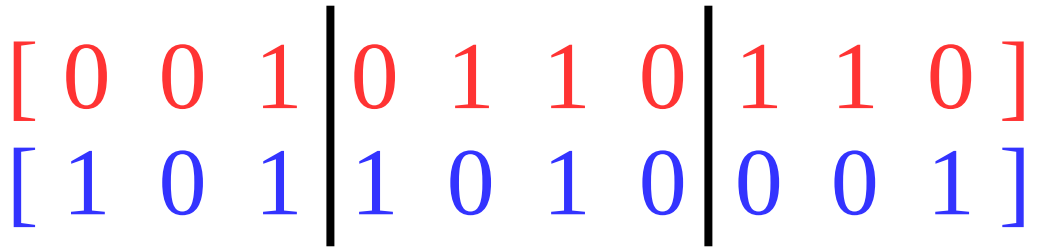
Genetic Crossover:



Pick a crossover point, and exchange genes



Genetic Crossover:



Two-point crossover exchanges a section of genes

[0 0 1 1 0 1 0 1 1 0]
[1 0 1 0 1 1 0 0 0 1]

Two-point crossover is more biased to modify genes near the middle of the “genome”

[0 0 1 1 0 1 0 1 1 0]
[1 0 1 0 1 1 0 0 0 1]

While single-point crossover is biased towards making changes near the edges

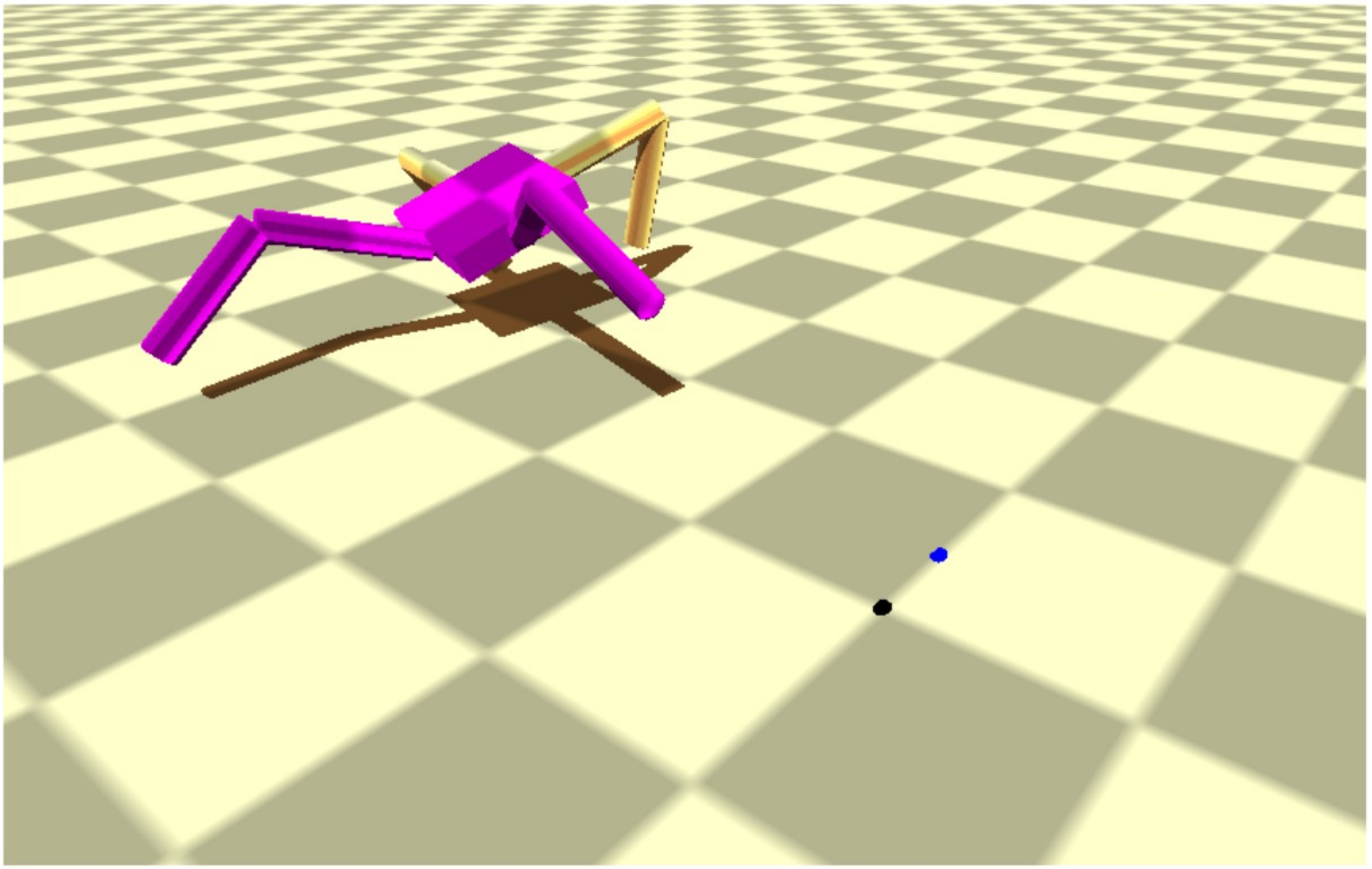
[0 0 1 0 1 1 0 0 0 1]
[1 0 1 1 0 1 0 1 1 0]

Both are more likely to change nearby genes together than those far apart on the genome

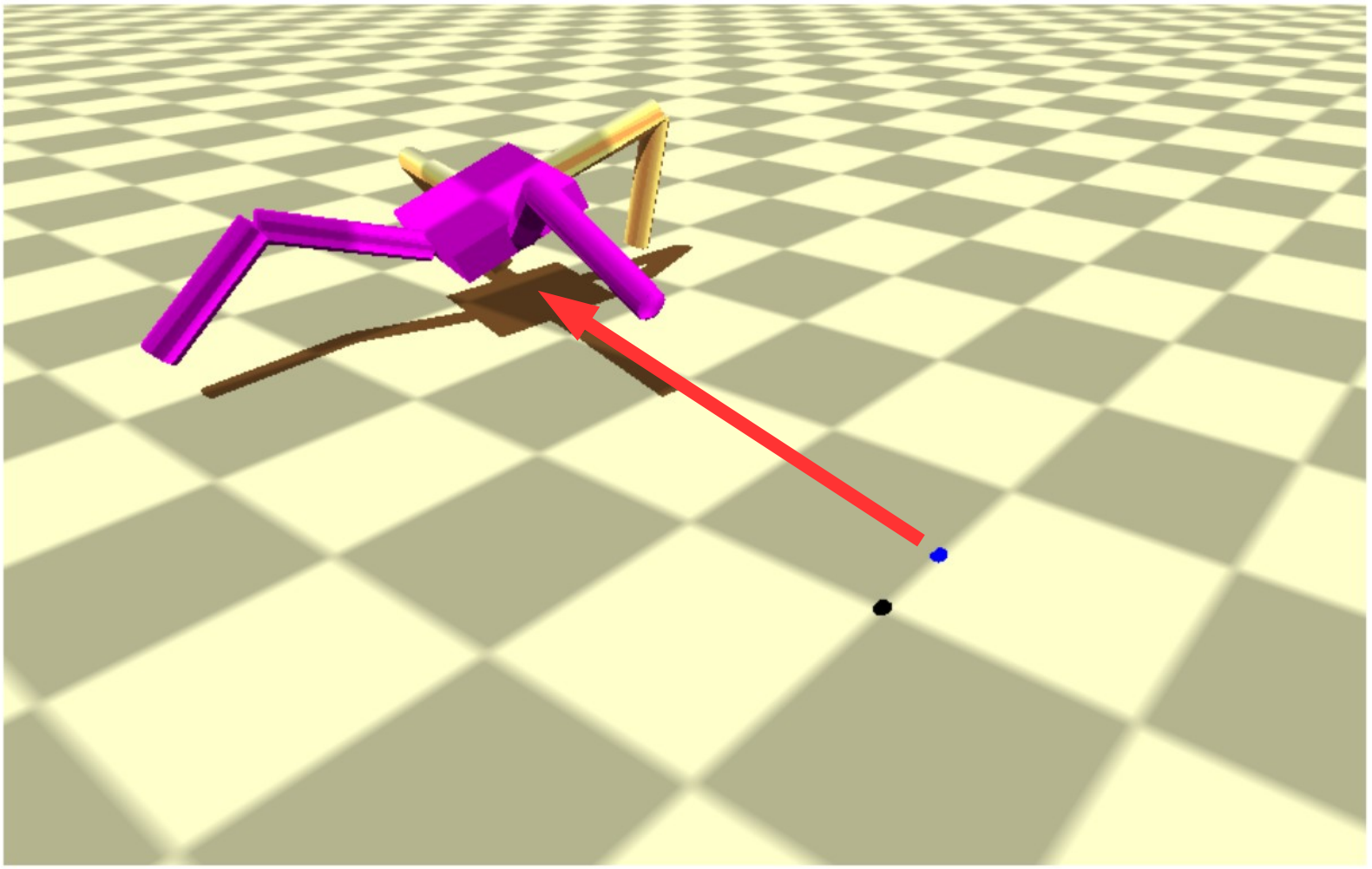
Evaluation

Now that we have some potential solutions (the new children from genetic variations), let's see how good they are...

For our robotics examples, this will mean building a physical robot (“phenotype”) that is described by the blueprints in the DNA (“genotype”) inside of a physics engine (simulator)

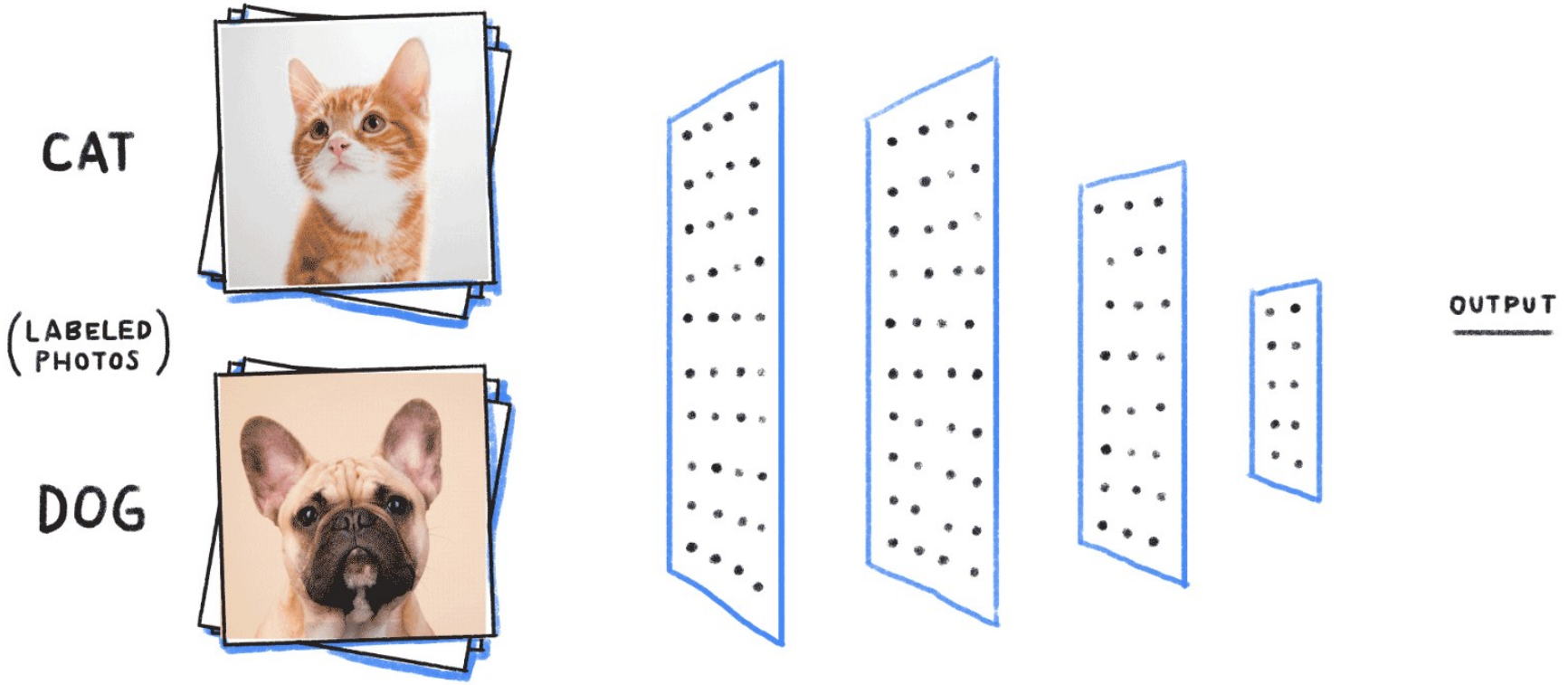


Then asking how well it did at our desired task (e.g. measuring how far it walked)



But as long as you're able to
rank solutions, it doesn't really matter
how your evaluator does it...

For example, your “fitness” could be proportion of images correctly classified by a neural network with the weights described in your genome



For our bit string example (and for your HW), let's just maximize the sum of the vector

$$\text{Fitness}(x) = \text{Sum}(x)$$

$$\text{Fitness}([\ 0\ \ 0\ \ 1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 1\ \ 1\ \ 0\ ]) = 5$$

Selection

Given a “population” of genomes which ones will go on to be the parents of the next “generation”?

Population (pre-selection)

[1 1 1 0 0 1 0 1 1 0]
[1 0 1 0 1 1 1 0 1 1]
[0 0 0 0 1 0 1 1 1 0]
[0 1 1 0 1 1 0 0 1 0]
[1 1 1 0 1 0 0 1 0 1]
[0 0 1 1 0 1 0 1 0 0]

Population fitnesses (pre-selection)

Fit([1 1 1 0 0 1 0 1 1 0]) = 4
Fit([1 0 1 0 1 1 1 0 1 1]) = 7
Fit([0 0 0 0 1 0 1 1 1 0]) = 4
Fit([0 1 1 0 1 1 0 0 1 0]) = 5
Fit([1 1 1 0 1 0 0 1 0 1]) = 6
Fit([0 0 1 1 0 1 0 1 0 0]) = 4

Population sorted by fitness (pre-selection)

Fit([1 0 1 0 1 1 1 0 1 1]) = 7
Fit([1 1 1 0 1 0 0 1 0 1]) = 6
Fit([0 1 1 0 1 1 0 0 1 0]) = 5

Fit([0 0 1 1 0 1 0 1 0 0]) = 4
Fit([0 0 0 0 1 0 1 1 1 0]) = 4
Fit([1 1 1 0 0 1 0 1 1 0]) = 4

Population (post “truncation” selection)

Fit([1 0 1 0 1 1 1 0 1 1]) = 7
Fit([1 1 1 0 1 0 0 1 0 1]) = 6
Fit([0 1 1 0 1 1 0 0 1 0]) = 5

~~Fit([0 0 1 1 0 1 0 1 0 0]) = 4~~
~~Fit([0 0 0 0 1 0 1 1 1 0]) = 4~~
~~Fit([1 1 1 0 0 1 0 1 1 0]) = 4~~

Population (post “truncation” selection)

Fit([1 0 1 0 1 1 1 0 1 1]) = 7
Fit([1 1 1 0 1 0 0 1 0 1]) = 6
Fit([0 1 1 0 1 1 0 0 1 0]) = 5

We now have ways to:

1) Grow the population (mutation)

2) Sort the population (evaluation)

3) Shrink the population (selection)

That's enough to build our first optimization algorithm!

Hillclimber

This is the simplest type of optimization algorithm (and the one you'll implement in HW 1)

It consists of an initial population of size 1,

We then use mutation to create a child from that parent (ballooning the population to size 2),

Evaluate both of those individuals using our fitness function (let's stick with “maximize the sum”)

Then use truncation selection to choose the best of those two (reducing the population back to 1),

later, rinse, and repeat… until desired outcome

Generation 0:

[0 0 1 0 1 1 0 1 1 0]

Generation 0:

[0 0 1 0 1 1 0 1 1 0]
[1 0 1 0 1 1 0 1 1 0]

Generation 0:

Fitness([0 0 1 0 1 1 0 1 1 0]) = 5
Fitness([1 0 1 0 1 1 0 1 1 0]) = 6

Generation 0:

~~Fitness([0 0 1 0 1 1 0 1 1 0]) = 5~~

Fitness([1 0 1 0 1 1 0 1 1 0]) = 6

Generation 1:

Fitness([1 0 1 0 1 1 0 1 1 0]) = 6

Generation 1:

Fitness([1 0 1 0 1 1 0 1 1 0]) = 6
Fitness([1 0 1 0 1 0 0 1 1 0]) = 5

Generation 1:

Fitness([1 0 1 0 1 1 0 1 1 0]) = 6

~~Fitness([1 0 1 0 1 0 0 1 1 0]) = 5~~

Generation 2:

Fitness([1 0 1 0 1 1 0 1 1 0]) = 6

Generation 2:

Fitness([1 0 1 0 1 1 0 1 1 0]) = 6
Fitness([1 0 1 0 1 1 0 1 1 1]) = 7

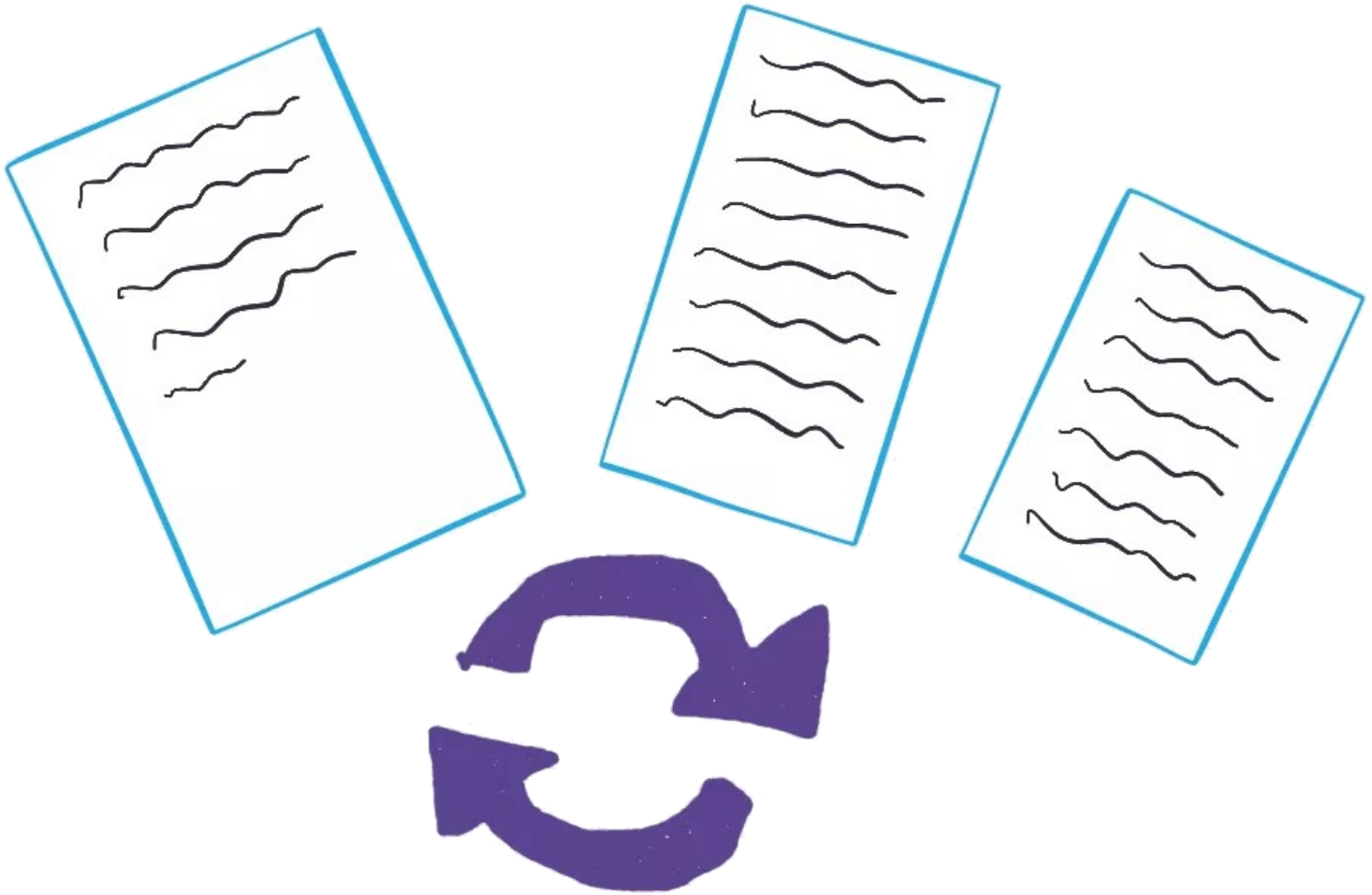
Generation 2:

~~Fitness([1 0 1 0 1 1 0 1 1 0]) = 6~~

Fitness([1 0 1 0 1 1 0 1 1 1]) = 7

Generation 3:

Fitness([1 0 1 0 1 1 0 1 1 1]) = 7



Lather, rinse, repeat

Does it seem simple and easy…

That's because it is!

How many generations will this algorithm take until it converges to the optimal solution?

(for this problem of maximizing sum for a binary vector of length 10)

What's the expected number of 1's in the vector in generation 0?

5/10

How likely is it to change a 0 to a 1 in generation 0?

5/10 = 0.5

What is the expectation for how many generations this will take to accomplish?

1/0.5 = 2

How many 1's are in the vector now?

6/10

How likely is it to change a 0 to a 1?

4/10 = 0.4

What is the expectation for how many generations this will take to accomplish?

1/0.4 = 2.5

How many 1's are in the vector now?

7/10

How likely is it to change a 0 to a 1?

3/10 = 0.3

What is the expectation for how many generations this will take to accomplish?

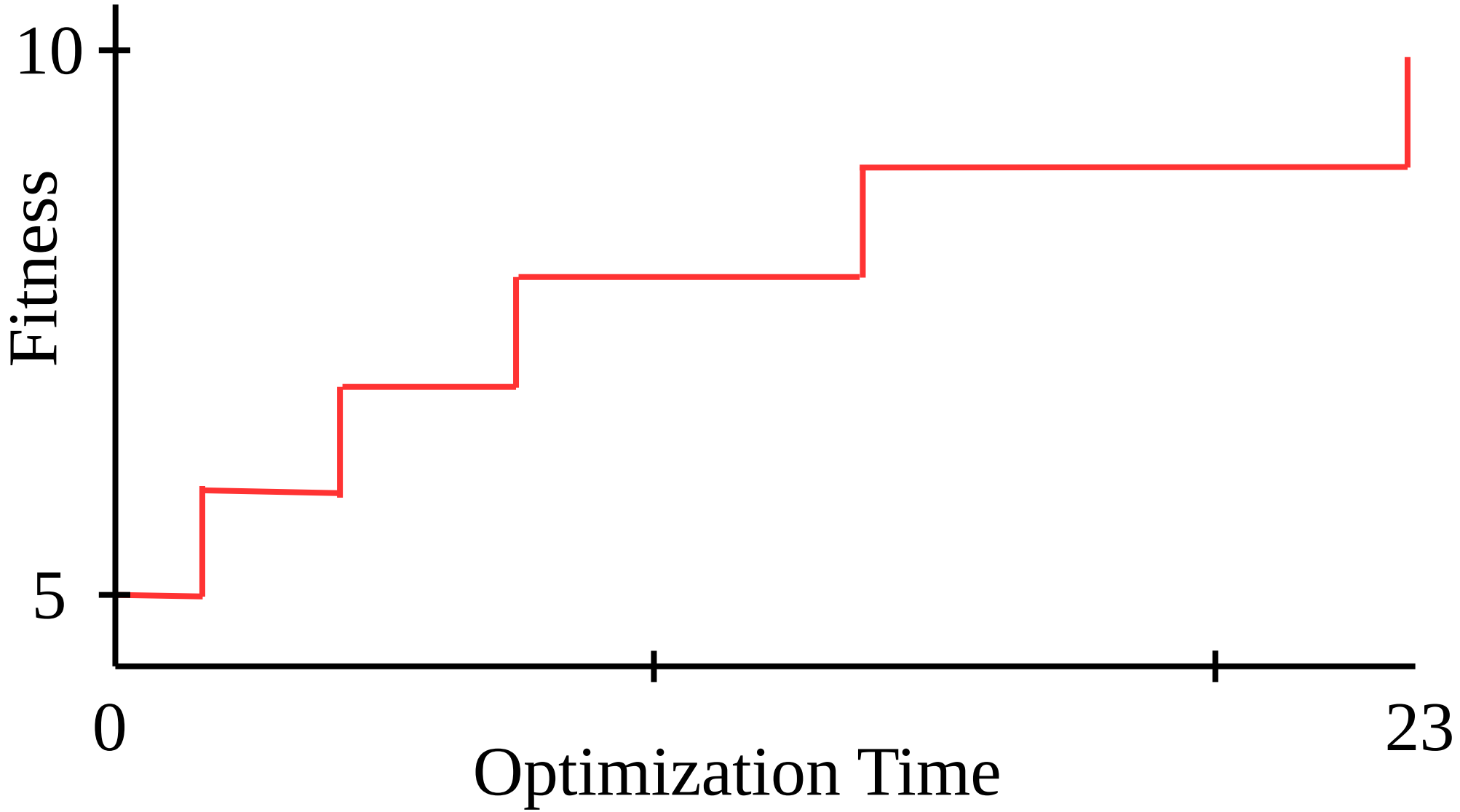
1/0.3 = 3.33

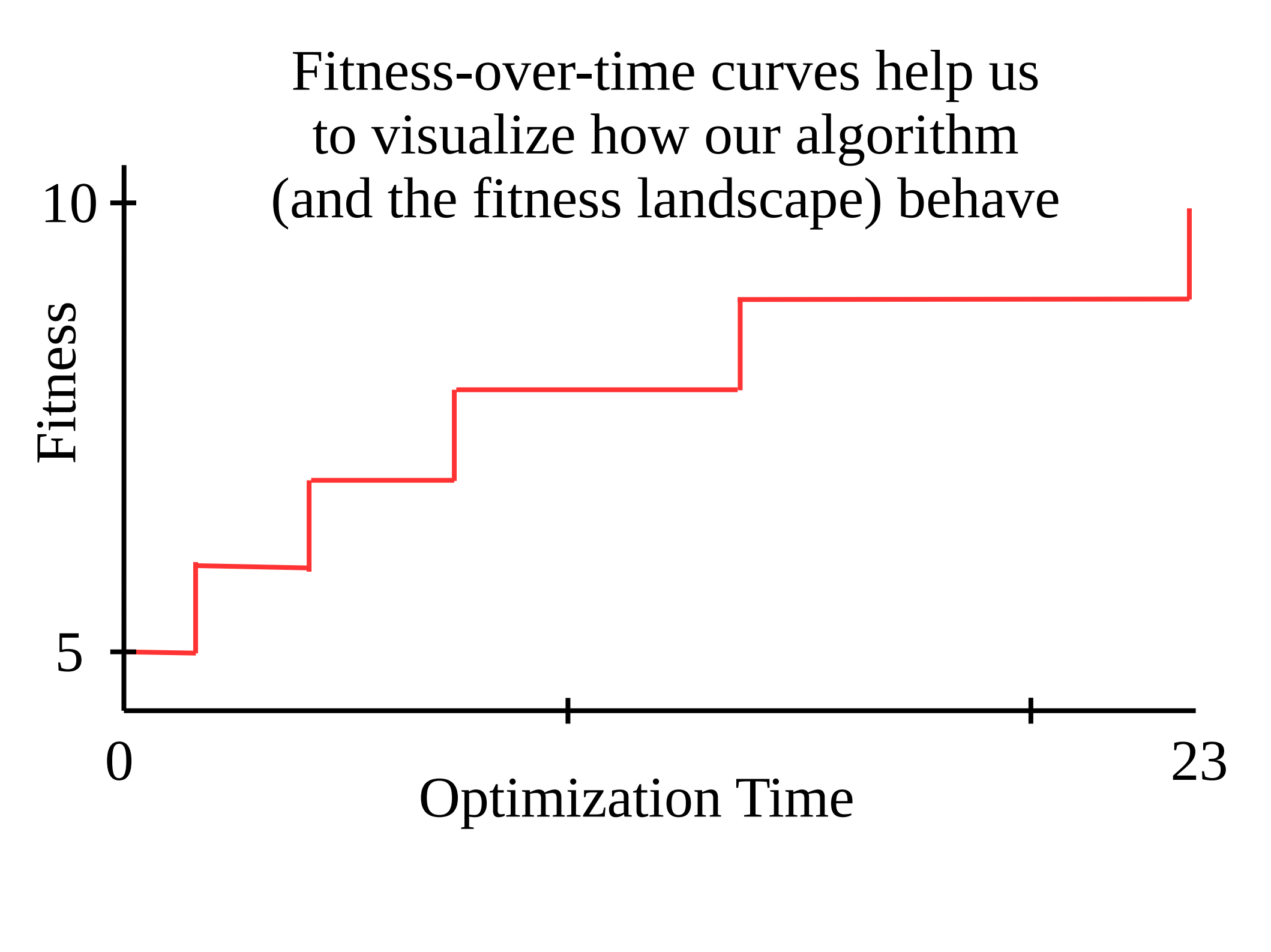
So how long will it take total?

10/5 + 10/4 + 10/3 + 10/2 + 10/1 = 22.83 generations

Note how the better the fitness of the vector
(the closer it is to being optimal),
the longer it takes to improve!

(i.e. asymptotic optimization curve)


10
5
Fitness
0
23
Optimization Time


Fitness-over-time curves help us
to visualize how our algorithm
(and the fitness landscape) behave
10
5
Fitness
0
23
Optimization Time

In your homework this weekend, you'll be asked to implement exactly this problem at a larger scale, (look for a similar asymptote and logarithmic curve)