AI is ubiquitous in the world, present in all but the most mundane tasks. Some AI is boring and easily described, such as a thermostat that turns the AC on when it's "hot" and the heat on when it's "cold". It "learns" the meaning of "hot" and "cold" via a single input that has a desired temperature specified.

There are obviously more complex AI. A chess-playing AI could, for instance, search a state space and α-β prune it to find the best move. This is an improvement, but it's not really learning anything. It's hard to justify the I in AI.

This brings us to genetic algorithms, one of many ways AI learns to do things. The basic idea is to emulate evolution. Good solutions survive to make better solutions while bad solutions are culled for being weak. The outline of the algorithm is simple.

```
Genetic Algorithm
    Initialize Population
    Repeat until done or good enough
    |   Select Parents
    |   Breed
    |   Mutate
    |   Evaluate Offspring
    |   Merge Offspring
    |   Cull Population
    Return Best solution
```

Let's take a closer look at each of these steps. Suppose we have some goal function $g : \mathbb{Z}_2^8 \to \mathbb{R}$ that we want to maximize. We have no idea what it does, but we have a black box implementation of it. We could just try all 2^8 inputs and look for the best one, but in general, the state space may be enormous or even continuous. So for this example, we'll ignore the easy brute force solution.

Initialization

We need an initial population to work with. We can either simply pick a boring initialization and leave it to mutation to get the real initialization running, or we can pick some random inputs to start with. Both are reasonable choices depending on context. Here we'll go with the latter. We introduce into the population

11010000
00101110
10001100
01010101,

Select Parents

There are a few ways to do this. We could select parents randomly, but that probably won't get us anywhere.

We could select the "best" parents, but that can get us hard stuck at local maxima.

We could hold a tournament arc where individuals compete against each other in a single elimination tournament with a probability of winning proportional to their relative "fitness", which here would mean how large g is on them.

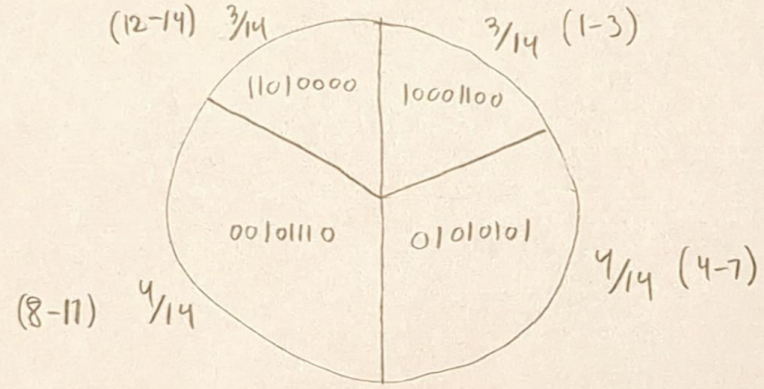
Alternatively, we can spin to win, with a roulette selection. Here you have a population with a probability of being selected proportional to its "fitness". Let's go with this.

Suppose g is secretly just counting set bits. Then

$g(11010000) = 3$ $\quad$ $g(10001100) = 3$

$g(00101110) = 4$ $\quad$ $g(01010101) = 4$,

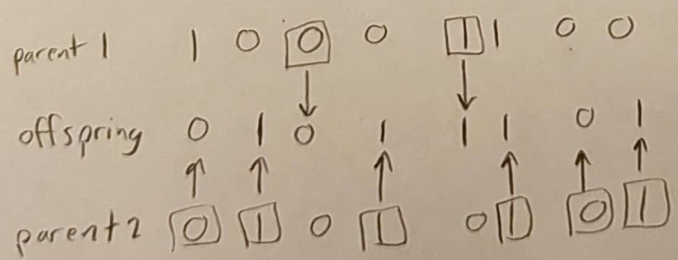
we have them stored in an array and calculate their fitness. We have total fitness $n = 3+3+4+4 = 14$, so we roll a d14 to select the parents (cloning is allowed).



Maybe we select 10001100 and 01010101 as the parents.

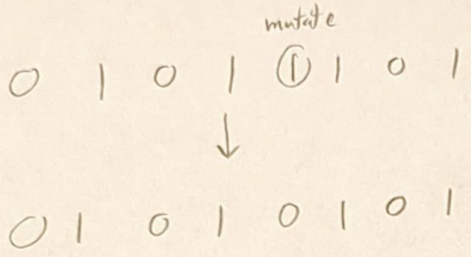
Breeding

How breeding occurs depends on context, but generally speaking, you randomly select a parent for each "chromosome" and make children from them or try to combine chromosomes. Here, we'll just treat each bit as a chromosome and take from parents randomly to make a single child (but we could produce more).



Mutation

To avoid getting stuck or being unable to explore the entire state space we mutate offspring with some (usually small) mutation rate. This causes chromosomes to randomly change. In our example, if no member of the population has a 1 (or 0) at some bit position i, this would allow us to mutate the i^{th} bit by chance. We'll not do so this time but will provide an example.



Evaluate Offspring

We want to keep track of the best solution found so far, so we do that here. Easy peasy lemon squeezy.

Merge Offspring

We simply add offspring into the general population here.

Cull Population

We want to keep the population at a small (relatively speaking) and managable level, so when we add new offspring into the population, we have to cull the underperformers. In this example, we'll drop 11010000.

Now let's put this all together.

1101 0000
0010 1110

1000 1100
0101 0101

10001100 → 01011101
01010101

call 11010000

01011101
00101110

1000 1100
0101 0101

01011101
10001100 → 00001100

call 00001100 (it was unworthy)

01011101
00101110

1000 1100
0101 0101

01011101
00101110 → 01111111

call 10001100

01011101
00101110

01111111
01010101

01111111
01010101 → 11111111
lucky mutation

call 00101110

01011101
11111111

01111111
01010101

⋮

We can continue until all strings are 18 with high probability or cut off early here and go with 18 as "good enough".