In [1]:
```
import numpy as np
import pandas as pd
```

1. Genetic Algorithms

In case you need:

In [2]:
```
Fitness_key = {'1011': 15, '0011': 22, '1001': 27, '1000': 30,
               '0010': 31, '0001': 30, '0000': 27, '1010': 22,
               '0100': 15, '1100': 6, '0101': -5, '0110': -18,
               '0111': -33, '1101': -50,'1110': -69, '1111': -90}

def fitness(individual):
    """
    Looks up fitness of an individual in a population

    Parameters
    ----------
    individual: string
    Individual's encoding

    Returns
    --------
    fitness : int
    The fitness of the individual
    """
    fitness_value = Fitness_key.get(individual)
    return fitness_value
```

1a

ENCODING A Good solutions are: 3, 4, 5 Schema here is *0**. This schema has an order of 1 and length of 0.

ENCODING B Good solutions are: 3, 4, 5. Schema here is 1**1. This schema has an order of 2 and length of 3.

I will be choosing ENCODING A, because shorter, lower order schema with high fitness will increase exponentially in descendant generations.

1b

We draw candidates 10, 1, 15, 6, 0 and 9 from ENCODING A and pair the most fit with the least fit members.

In [3]:
```
encodingA_data = {'Solution': [6, 15, 1, 10,  0, 9], 'Fitness': [27, -90, 22
F0_df = pd.DataFrame(data=encodingA_data, index=['Pairing 1a', 'Pairing 1b',
```

```
F0_df
```

Out[3]:

	Solution	Fitness	Encoding
Pairing 1a	6	27	0000
Pairing 1b	15	-90	1111
Pairing 2a	1	22	0011
Pairing 2b	10	-5	0101
Pairing 3a	0	15	1011
Pairing 3b	9	6	1100

1c

In [4]:
```
F0_population = F0_df['Encoding'].tolist()

def crossover(a, b):
    """
    This function performs a crossover. It exchanges the last three elements

    Parameters
    ----------
    a,b : string
    Parent chromosomes

    Returns
    -------
    offspring : list
    Offspring
    """
    #offspring1 = a.replace(a[1:4], b[1:4])
    #offspring2= b.replace(b[1:4], a[1:4])

    offspring1 = a[0] + b[1:4]
    offspring2 = b[0] + a[1:4]

    return [offspring1, offspring2]
```

In [5]:
```
# PERFORM CROSSOVER BETWEEN PAIRS
Pair1_offspring = crossover(F0_df.iloc[0, 2], F0_df.iloc[1, 2])
Pair2_offspring = crossover(F0_df.iloc[2, 2], F0_df.iloc[3, 2])
Pair3_offspring = crossover(F0_df.iloc[4, 2], F0_df.iloc[5, 2])

F1_population = Pair1_offspring + Pair2_offspring + Pair3_offspring
F1_population
```

Out[5]:
```
['0111', '1000', '0101', '0011', '1100', '1011']
```

Pairing 1 has given us two new solutions: 0111 and 1000. The fitness of the new solutions and of the F1 population are as follows:

In [6]:
```
def population_fitness(population):
    """
    This function pulls fitness values for a population (using a previously

    Parameters
    ----------
    population : list (or array) of binary strings
    Encodings of current population

    Returns
    --------
    population_fitness : int
    Total fitness of population
    """
    fitnesses = []
    for individual in population:
        new_fitness_values = Fitness_key.get(individual)
        fitnesses.append(int(new_fitness_values))

    sum = 0
    for i in fitnesses:
        sum += i

    return sum

print(f'Fitness of 0111 is {fitness("0111")} and fitness of 1000 is {fitness
print(f'Fitness of F0 generation is: {population_fitness(F0_population)}.')
print(f'Fitness of F1 generation is: {population_fitness(F1_population)}.')
```

```
Fitness of 0111 is -33 and fitness of 1000 is 30
Fitness of F0 generation is: -25.
Fitness of F1 generation is: 35.
```

Yes, fitness has increased from F0 to F1.

1d

In [7]:
```
F1_population = Pair1_offspring + Pair2_offspring + Pair3_offspring

# mutate 3rd element
mutated_F1_population = []
for individual in F1_population:
    third_element = individual[2]

    if third_element == "0":
        mutated_element = "1"
    if third_element == "1":
        mutated_element = "0"

    mutated_individual = individual[0:2] + mutated_element + individual[3:]
    mutated_F1_population.append(mutated_individual)
```

```
print(f'Fitness of F1 generation is: {population_fitness(F1_population)}.')
print(f'Fitness of mutated F1 population is: {population_fitness(mutated_F1_
```

```
Fitness of F1 generation is: 35.
Fitness of mutated F1 population is: -28
```

Fitness has not increased in the mutated F1 population.

Compared to the F1 population, there are new solutions in the mutated F1 population: 1010, 0001, 1110, 1001. Their fitnesses are as follows.

In [8]:
```
print(fitness('1010'))
print(fitness('0001'))
print(fitness('1110'))
print(fitness('1001'))
```

```
22
30
-69
27
```

No, this mutation has not increased fitness in the population.

1e

In [9]:
```
#print(f'Mutated F1 {mutated_F1_population}.')

def natural_selection(population):
    """
    This function eliminates the least fit member of the population and clor

    Parameters
    ----------
    population : list
    list of solutions

    Returns
    -------
    mutated population : list
    list of mutated solutions
    """
    fittest_individual = max(population, key=fitness)
    least_fit_individual = min(population, key=fitness)

    # replace index of least fit with fittest
    population[population.index(least_fit_individual)] = fittest_individual

    return population

#print(natural_selection(mutated_F1_population))
```

In [10]:
```
def two_point_crossover(a,b):
    """
    This function performs a 2-point crossover. It exchanges the middle two
```

```
    """
    offspring1 = a[0] + b[1:3] + a[3]
    offspring2 = b[0] + a[1:3] + b[3]

    return [offspring1, offspring2]

print('F2 solutions are:', two_point_crossover(mutated_F1_population[0], mut
                                               + two_point_crossover(mutate
                                               +two_point_crossover(mutatec
```

```
F2 solutions are: ['0011', '1100', '0001', '0111', '1000', '1111']
```

New solutions from the two-point crossover are 0011, 1100, 1000, 1111

In [11]:
```
F2_population = ['0011', '1100', '0001', '0111', '1000', '1111']

print('Popuation fitness of F2 solutions is:', population_fitness(F2_populat
print('Individual fitnesses:')
print(fitness('0011'))
print(fitness('1100'))
print(fitness('1000'))
print(fitness('1111'))
```

```
Popuation fitness of F2 solutions is: -35
Individual fitnesses:
22
6
30
-90
```

No, the two point crossover has not increased fitness in the population.

1f

In [12]:
```
natural_selection(F2_population)

def crossover_3_4_switch_3(a,b):
    """
    This function does a cross over between 3rd and 4th elements then exchar
    """
    middle1 = a[0:2] + a[3] + a[2]
    middle2 = b[0:2] + b[3] + b[2]

    offspring1 = middle2[0:3] + middle1[3]
    offspring2 = middle1[0:3] + middle2[3]

    return [offspring1, offspring2]

mutated_F2_population = (crossover_3_4_switch_3(F2_population[0], F2_populat
                         + crossover_3_4_switch_3(F2_population[2], F2_popul
                         + crossover_3_4_switch_3(F2_population[4], F2_popul

print(f'Mutated F2 population is {mutated_F2_population}.')
```

```
Mutated F2 population is ['1101', '0010', '0110', '0011', '0010', '1000'].
```

New solutions are 1101, 0010, and 0110. Below are their fitness values:

In [13]:
```
print(fitness('1101'))
print(fitness('0010'))
print(fitness('0110'))

population_fitness(mutated_F2_population)
```

```
-50
31
-18
```

Out[13]: 46

Yes, this cross has increased fitness in the population.

1g

no, I don't think the solution space was adequate for maximizing this function. There are only 16 solutions so crossovers ended up being repetitive and not yielding many new reesults. We could increase this number by making the encodings (for example) 5 digits instead of 4. With more possible solutions, we are more likely to find a better maximum.

2. Artificial Neural Networks

In [14]:
```
class NeuralNetwork:
    def __init__(self, input_neurons, hidden_layer, output_neurons):
        # initialize weights
        np.random.seed(0)
        self.weights_ih = np.random.rand(input_neurons, hidden_layer)
        self.bias_h = np.random.rand()

        self.weights_ho = np.random.rand(hidden_layer, output_neurons)
        self.bias_o = np.random.rand()


    def feed_forward(self, x):
        hidden = np.tanh(np.dot(x, self.weights_ih) + self.bias_h)
        output = np.tanh(np.dot(hidden, self.weights_ho) + self.bias_o)

        return output

    def backprop(self, x, output, alpha):
        hidden = np.tanh(np.dot(x, self.weights_ih) + self.bias_h)
        z = hidden * self.weights_ho + self.bias_h
        observed_output = np.array([-1, -1])
        theta21_output = output[0]
        theta22_output = output[1]

        delta_n21 = (theta21_output - observed_output[0]) * (1- np.tanh(thet
        delta_n22 = (theta22_output - observed_output[1]) * (1- np.tanh(thet
```

```
        self.weights_ho = self.weights_ho - alpha * self.weights_ho * 1 - np
        self.bias_h = self.bias_h - alpha * 1 - np.tanh(z)**2 * (np.tanh(z)

        return np.array([delta_n21, delta_n22])

network = NeuralNetwork(6,2,2)
input = np.array([-1, 1, -1, -1, 1, -1])


model_output = network.feed_forward(input)
print(model_output)
```

```
[0.6016075 0.669866 ]
```

In [15]:
```
network.backprop(input, model_output, 0.01)
```

Out[15]:
```
array([1.13770012, 1.09860762])
```