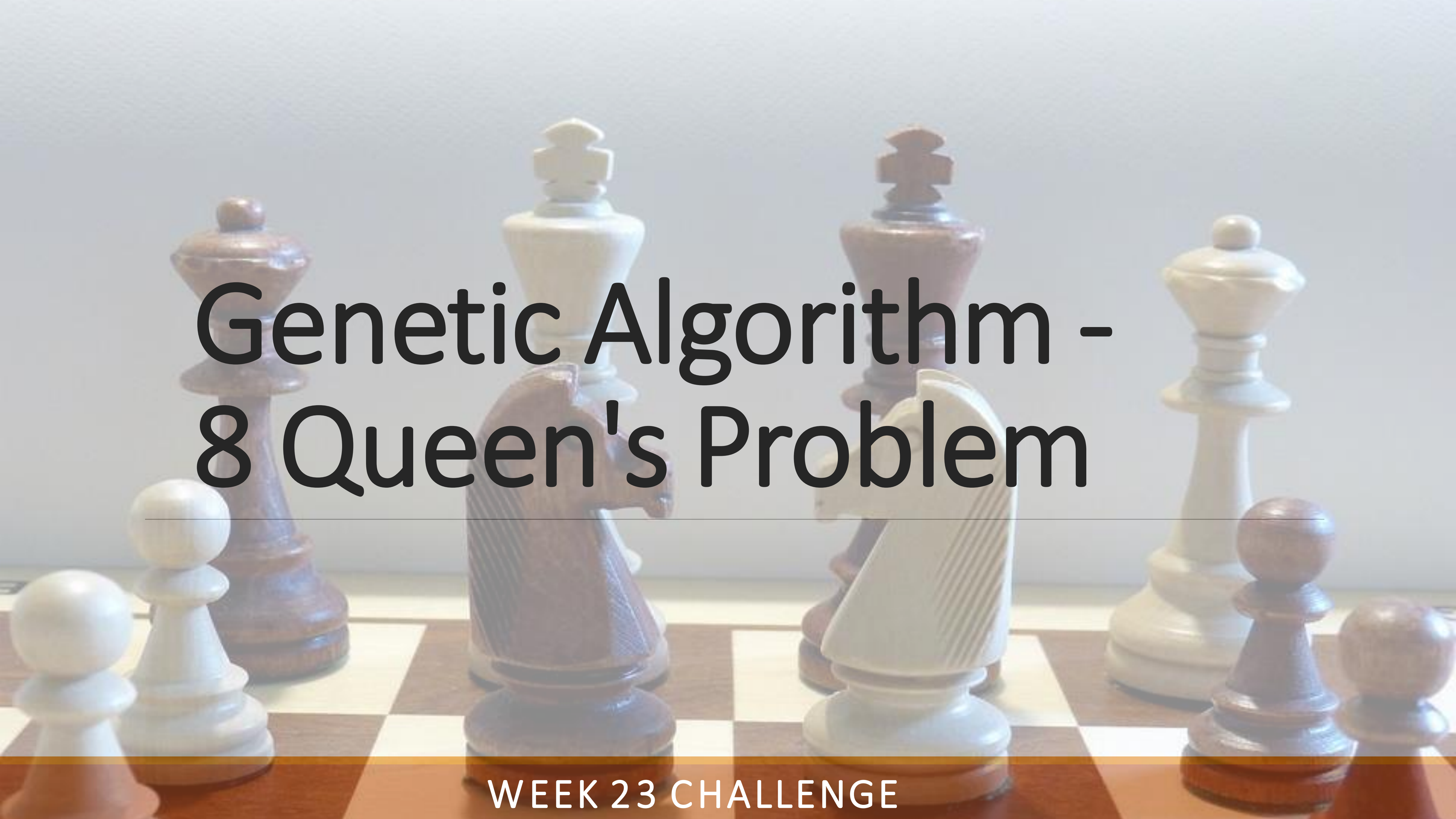
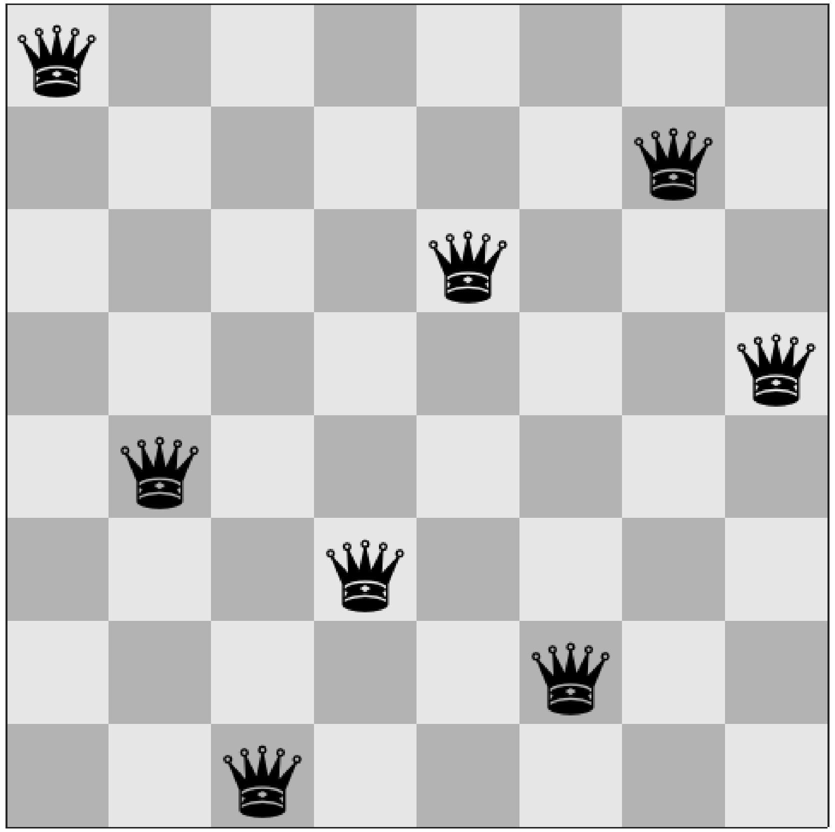


Genetic Algorithm - 8 Queen's Problem

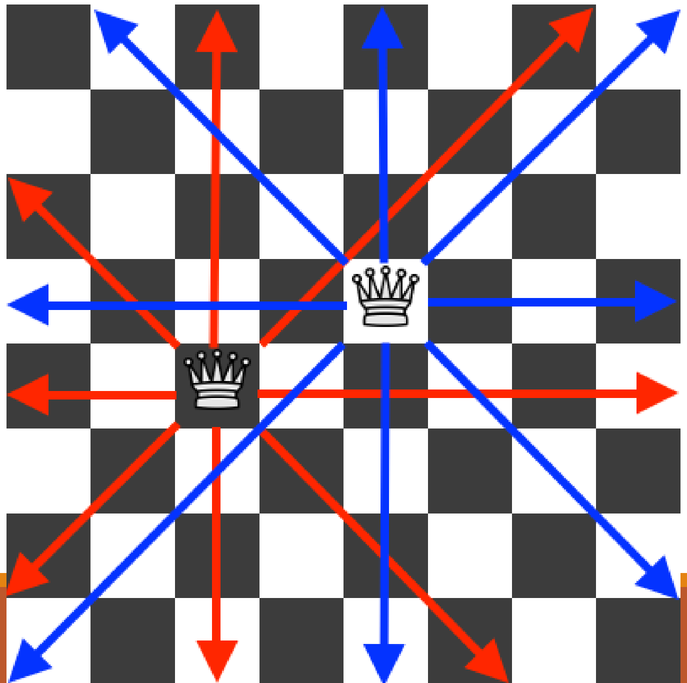
WEEK 23 CHALLENGE

Evolutionary processing defines an algorithm as follows: “**There is a population of individuals (states) in which more appropriate individuals (the highest value) produce offspring (substitute states) that fill the next generation, a process called recombination."**



Problem definition of 8 ministers:

To find the possible arrangements of 8 ministers on a standard 8x8 chessboard so that no minister is in an offensive position. If we know the basics of chess, we can say that a minister can move horizontally, vertically, or diagonally. Therefore, for two or more ministers to be on the offensive, they must be horizontally, vertically, or transversely aligned with the other minister. The figure below shows the minister's attack situations.

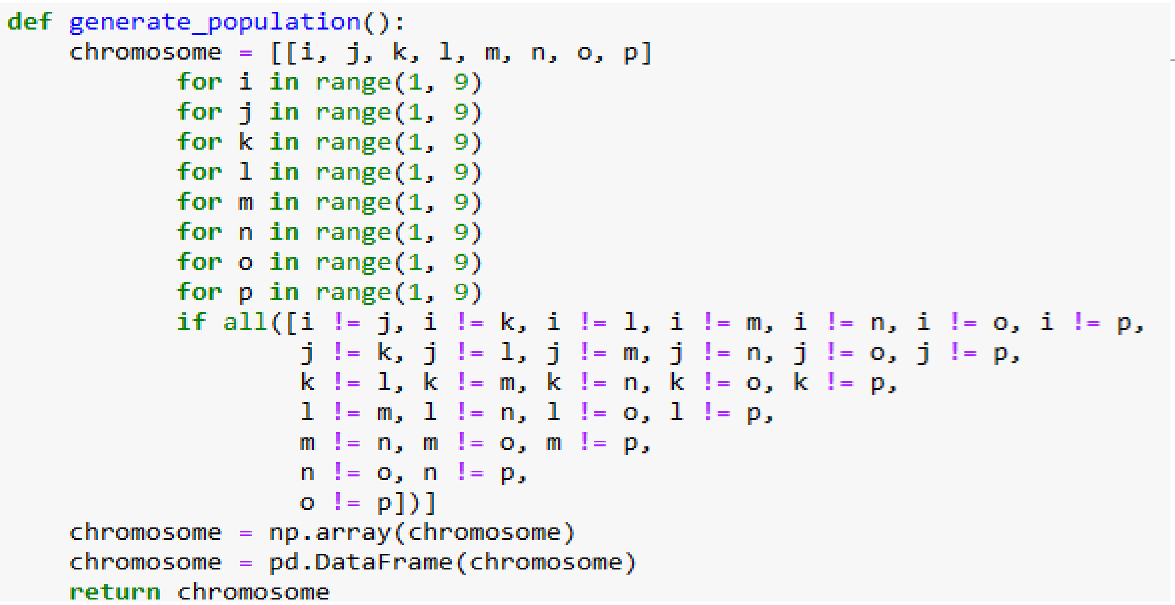


The issue of 8 ministers:

In the case of 8 ministers, our space is an 8 * 8 range, and for this space, which is our search space, we have 2^64 cases, which is a huge search space and makes our work difficult. So we add problem information to make the search space smaller, and we use an 8-element array to display the answers to the problem, which gives us an 8-digit permutation, and finally, we have 8! We have a mode, and the search space becomes smaller, and the search in this space is more accessible.

Generate_population function:

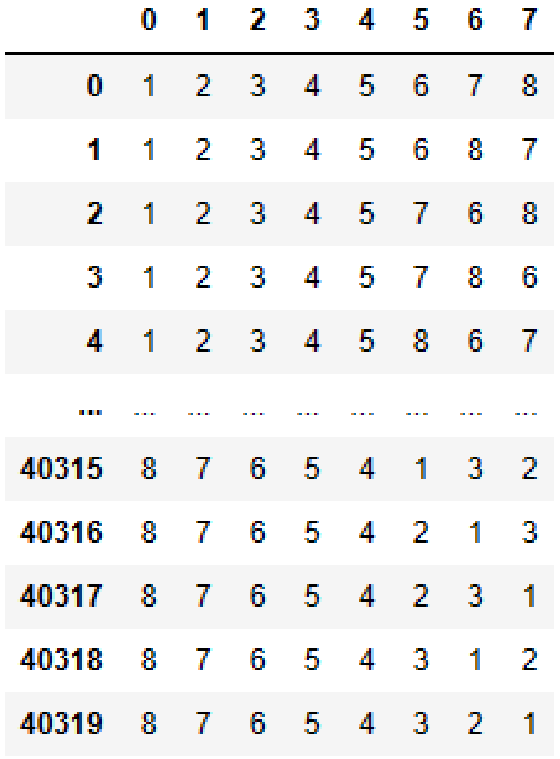
This function produces the initial population, this function produces all the positions of the ministers, ie 8! = 40320:



```python
def generate_population():
    chromosome = [[i, j, k, l, m, n, o, p]
           for i in range(1, 9)
           for j in range(1, 9)
           for k in range(1, 9)
           for l in range(1, 9)
           for m in range(1, 9)
           for n in range(1, 9)
           for o in range(1, 9)
           for p in range(1, 9)
           if all([i != j, i != k, i != l, i != m, i != n, i != o, i != p,
                   j != k, j != l, j != m, j != n, j != o, j != p,
                   k != l, k != m, k != n, k != o, k != p,
                   l != m, l != n, l != o, l != p,
                   m != n, m != o, m != p,
                   n != o, n != p,
                   o != p])]
    chromosome = np.array(chromosome)
    chromosome = pd.DataFrame(chromosome)
    return chromosome
```

Check for Initial Population:

The below output represent the initial population:



```
         0   1   2   3   4   5   6   7
     0   1   2   3   4   5   6   7   8
     1   1   2   3   4   5   6   8   7
     2   1   2   3   4   5   7   6   8
     3   1   2   3   4   5   7   8   6
     4   1   2   3   4   5   8   6   7
   ...  ... ... ... ... ... ... ... ...
 40315   8   7   6   5   4   1   3   2
 40316   8   7   6   5   4   2   1   3
 40317   8   7   6   5   4   2   3   1
 40318   8   7   6   5   4   3   1   2
 40319   8   7   6   5   4   3   2   1
```

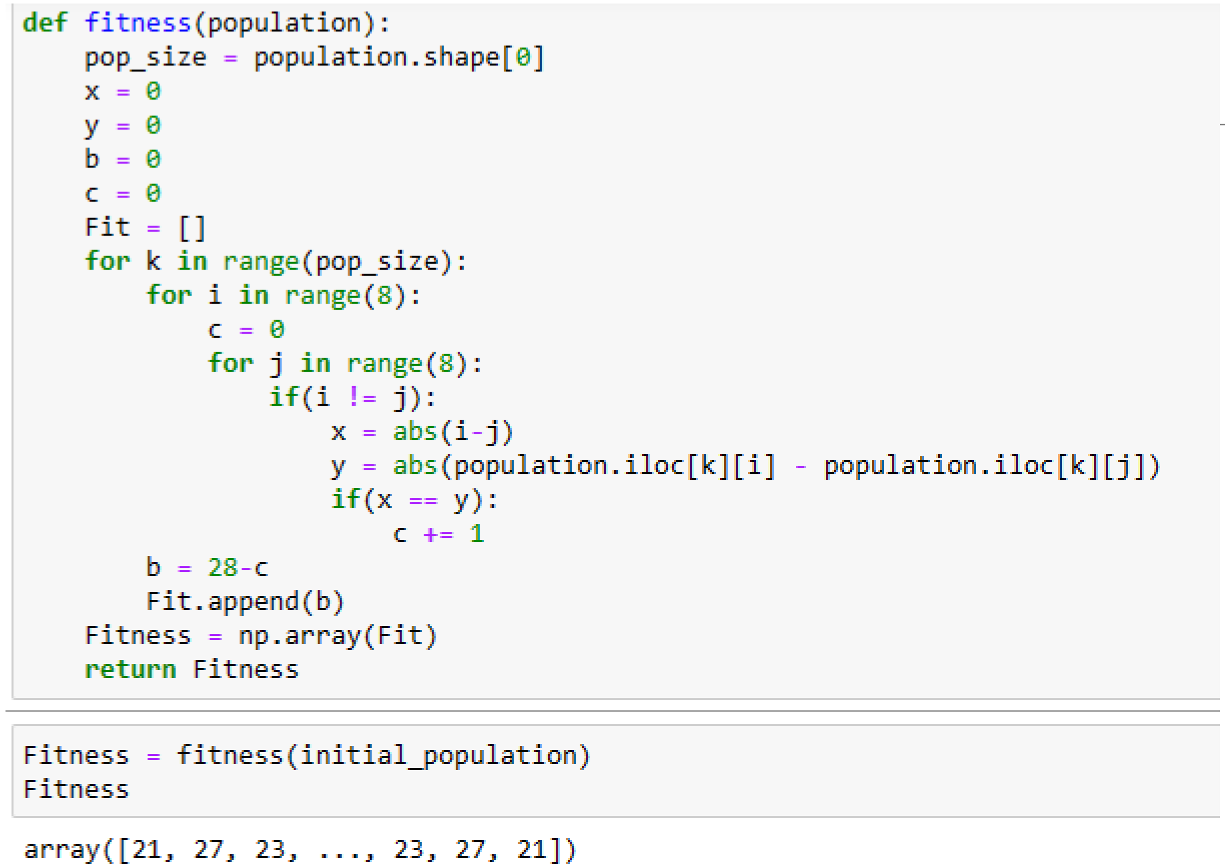
Fitness:

Fitness is a quantity that helps us know how good the answer to the problem is.
Here is the best Fitness as follows:

$$\text{Fitness}(n) = \frac{n(n-1)}{2} = \frac{8(8-1)}{2} = 28$$

Fitness function:

This function takes population and achieves Fitness for each chromosome:



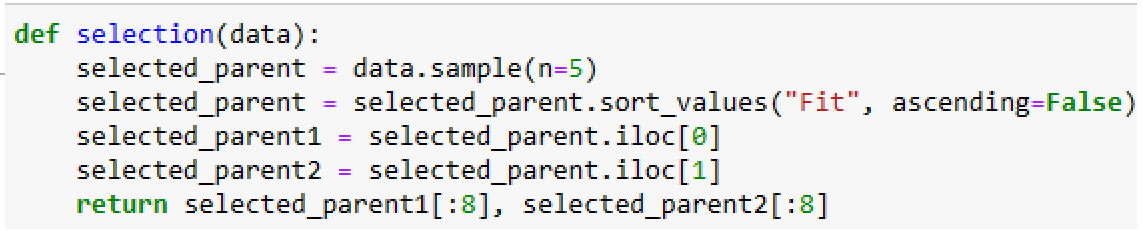
```
def fitness(population):
    pop_size = population.shape[0]
    x = 0
    y = 0
    b = 0
    c = 0
    Fit = []
    for k in range(pop_size):
        for i in range(8):
            c = 0
            for j in range(8):
                if(i != j):
                    x = abs(i-j)
                    y = abs(population.iloc[k][i] - population.iloc[k][j])
                    if(x == y):
                        c += 1
        b = 28-c
        Fit.append(b)
    Fitness = np.array(Fit)
    return Fitness
```

```
Fitness = fitness(initial_population)
Fitness
```

```
array([21, 27, 23, ..., 23, 27, 21])
```

Selection function:

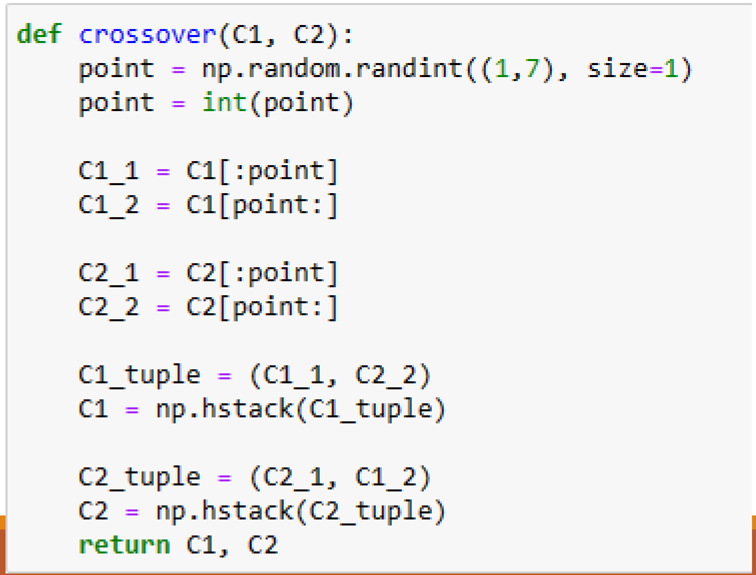
This function takes our data from the input and gives us two chromosomes as a parent:



```
def selection(data):
    selected_parent = data.sample(n=5)
    selected_parent = selected_parent.sort_values("Fit", ascending=False)
    selected_parent1 = selected_parent.iloc[0]
    selected_parent2 = selected_parent.iloc[1]
    return selected_parent1[:8], selected_parent2[:8]
```

Crossover function:

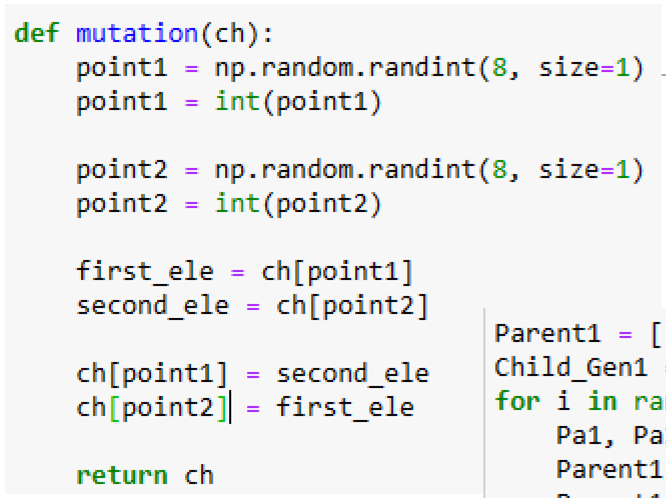
This function takes two chromosomes from the input and returns the two chromosomes after Crossover:



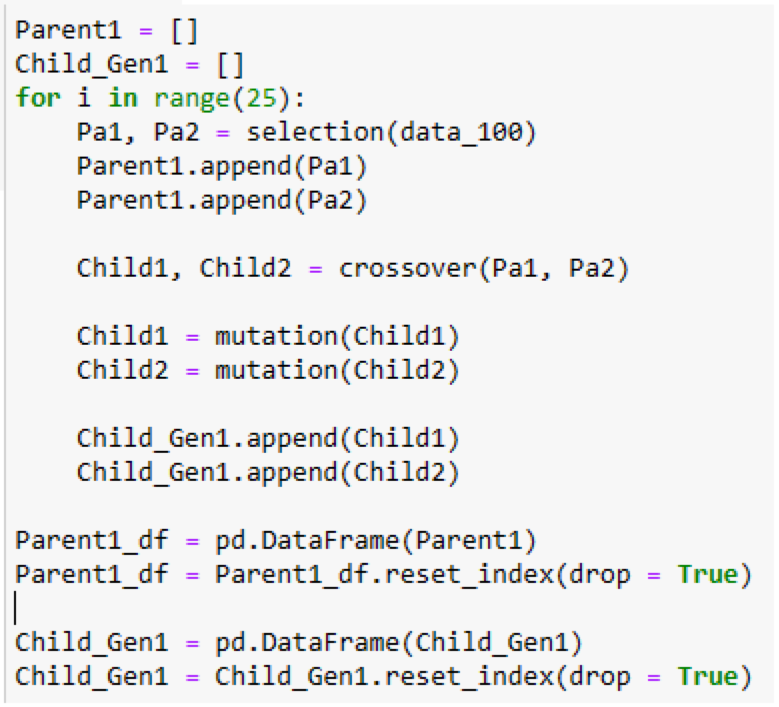
```
def crossover(C1, C2):
    point = np.random.randint((1,7), size=1)
    point = int(point)

    C1_1 = C1[:point]
    C1_2 = C1[point:]

    C2_1 = C2[:point]
    C2_2 = C2[point:]

    C1_tuple = (C1_1, C2_2)
    C1 = np.hstack(C1_tuple)

    C2_tuple = (C2_1, C1_2)
    C2 = np.hstack(C2_tuple)
    return C1, C2
```

Mutation function:

This function takes a chromosome from the input and returns the chromosome after mutation is applied to it:



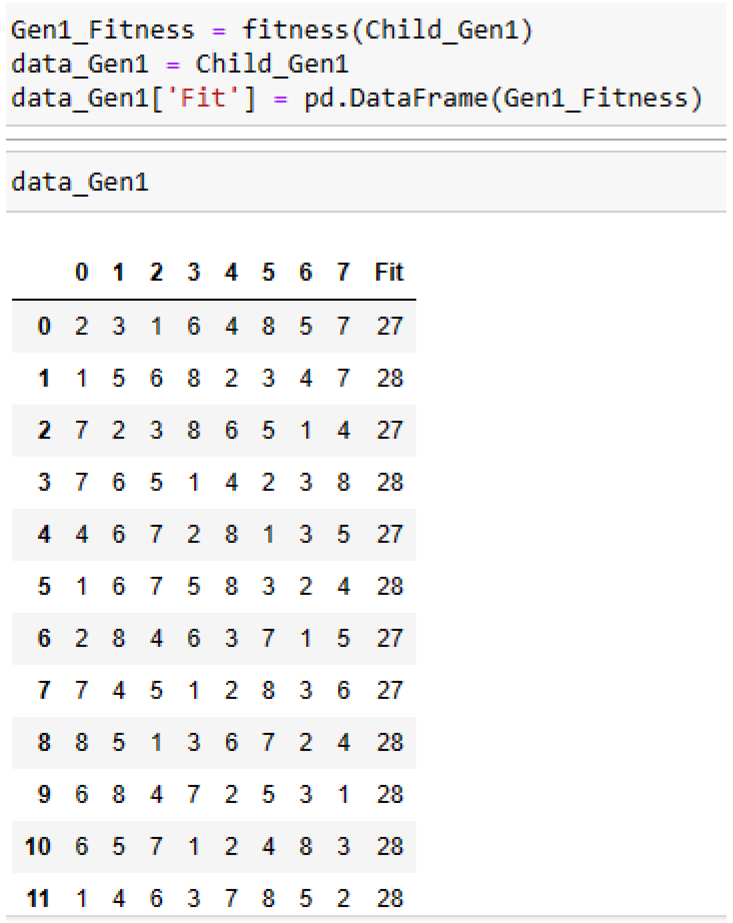
```python
def mutation(ch):
    point1 = np.random.randint(8, size=1)
    point1 = int(point1)

    point2 = np.random.randint(8, size=1)
    point2 = int(point2)

    first_ele = ch[point1]
    second_ele = ch[point2]

    ch[point1] = second_ele
    ch[point2] = first_ele

    return ch
```



```python
Parent1 = []
Child_Gen1 = []
for i in range(25):
    Pa1, Pa2 = selection(data_100)
    Parent1.append(Pa1)
    Parent1.append(Pa2)

    Child1, Child2 = crossover(Pa1, Pa2)

    Child1 = mutation(Child1)
    Child2 = mutation(Child2)

    Child_Gen1.append(Child1)
    Child_Gen1.append(Child2)

Parent1_df = pd.DataFrame(Parent1)
Parent1_df = Parent1_df.reset_index(drop = True)

Child_Gen1 = pd.DataFrame(Child_Gen1)
Child_Gen1 = Child_Gen1.reset_index(drop = True)
```

Generating First Generation:

First generation of children (**`Child_Gen1`**), calculating their fitness values using a function called **`fitness`**. Then, adding the fitness values as a new column named 'Fit' to the DataFrame **`data_Gen1`**.



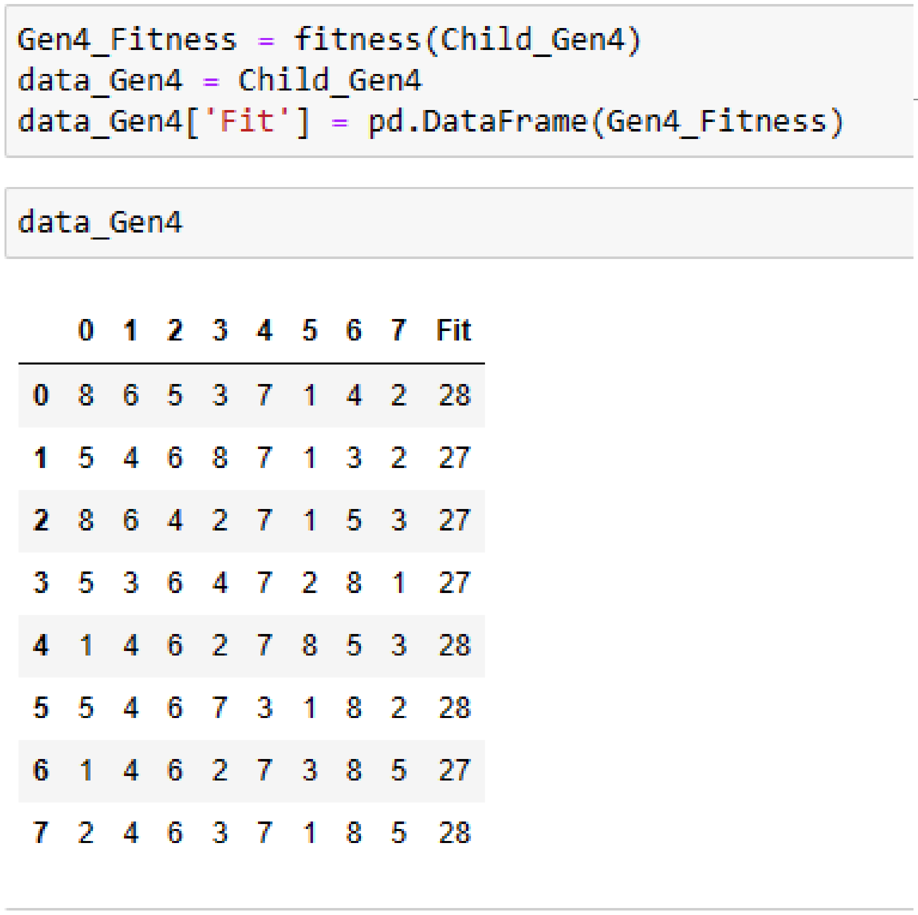
```python
Gen1_Fitness = fitness(Child_Gen1)
data_Gen1 = Child_Gen1
data_Gen1['Fit'] = pd.DataFrame(Gen1_Fitness)
```

```python
data_Gen1
```

	0	1	2	3	4	5	6	7	Fit
0	2	3	1	6	4	8	5	7	27
1	1	5	6	8	2	3	4	7	28
2	7	2	3	8	6	5	1	4	27
3	7	6	5	1	4	2	3	8	28
4	4	6	7	2	8	1	3	5	27
5	1	6	7	5	8	3	2	4	28
6	2	8	4	6	3	7	1	5	27
7	7	4	5	1	2	8	3	6	27
8	8	5	1	3	6	7	2	4	28
9	6	8	4	7	2	5	3	1	28
10	6	5	7	1	2	4	8	3	28
11	1	4	6	3	7	8	5	2	28

Repetition of the Mutation:

Performing of mutation & crossover until we find best fit of generation.



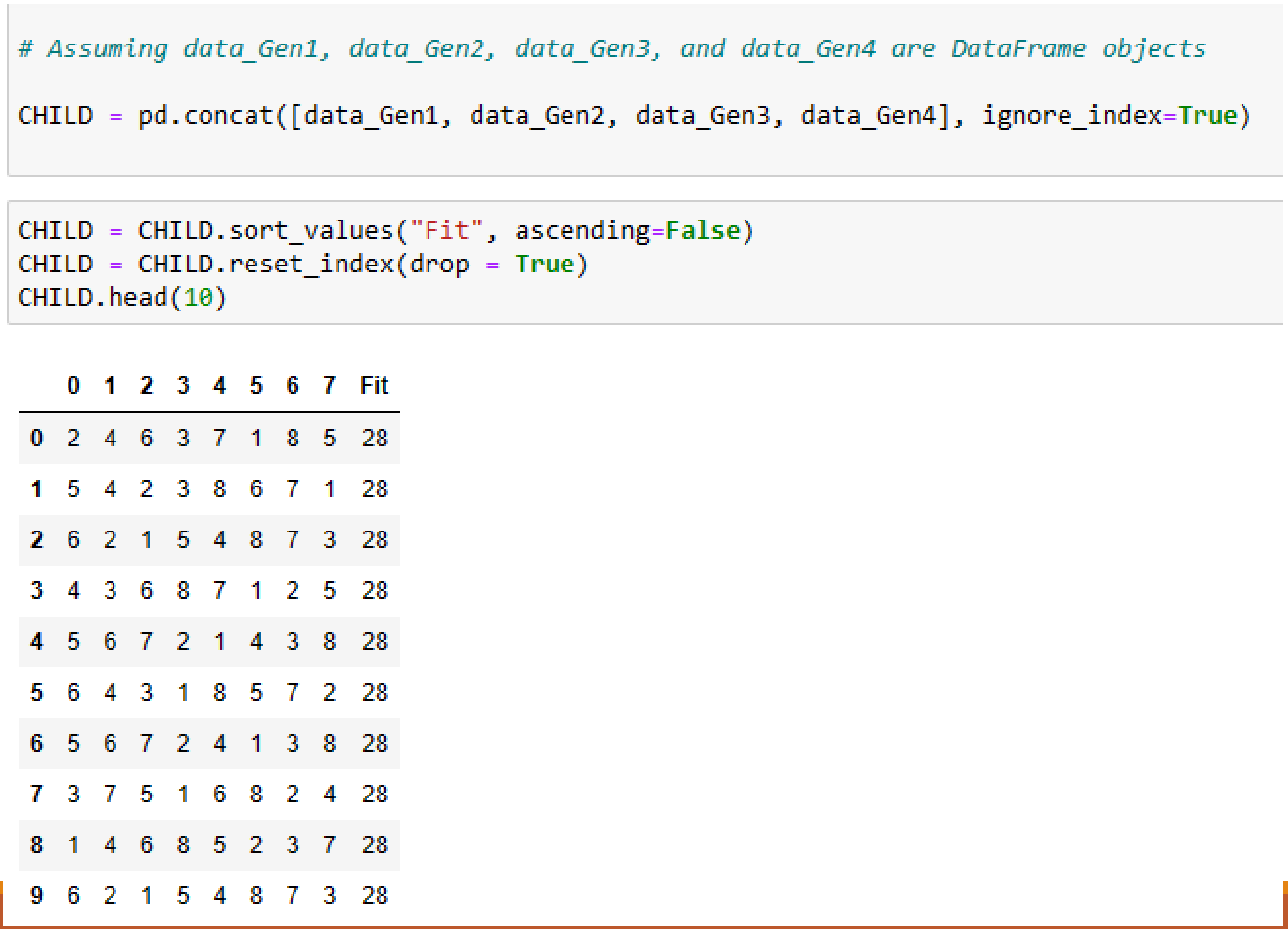
```
Gen4_Fitness = fitness(Child_Gen4)
data_Gen4 = Child_Gen4
data_Gen4['Fit'] = pd.DataFrame(Gen4_Fitness)
```

```
data_Gen4
```

	0	1	2	3	4	5	6	7	Fit
0	8	6	5	3	7	1	4	2	28
1	5	4	6	8	7	1	3	2	27
2	8	6	4	2	7	1	5	3	27
3	5	3	6	4	7	2	8	1	27
4	1	4	6	2	7	8	5	3	28
5	5	4	6	7	3	1	8	2	28
6	1	4	6	2	7	3	8	5	27
7	2	4	6	3	7	1	8	5	28

Aggregating of the data:

Effectively aggregating data from multiple generations, sorting them based on fitness, and displaying the top chromosomes in a single DataFrame. This process is common in evolutionary algorithms to track the progress of the population across generations and identify the best solutions found so far



```
# Assuming data_Gen1, data_Gen2, data_Gen3, and data_Gen4 are DataFrame objects

CHILD = pd.concat([data_Gen1, data_Gen2, data_Gen3, data_Gen4], ignore_index=True)
```

```
CHILD = CHILD.sort_values("Fit", ascending=False)
CHILD = CHILD.reset_index(drop = True)
CHILD.head(10)
```

	0	1	2	3	4	5	6	7	Fit
0	2	4	6	3	7	1	8	5	28
1	5	4	2	3	8	6	7	1	28
2	6	2	1	5	4	8	7	3	28
3	4	3	6	8	7	1	2	5	28
4	5	6	7	2	1	4	3	8	28
5	6	4	3	1	8	5	7	2	28
6	5	6	7	2	4	1	3	8	28
7	3	7	5	1	6	8	2	4	28
8	1	4	6	8	5	2	3	7	28
9	6	2	1	5	4	8	7	3	28

Thank You