OpenAI

ChatGPT

```
import random

# Fitness function: calculate the number of conflicts
def f(s):
    count = 0
    for i in range(8):
        for j in range(i+1, 8):
            if s[i] == s[j]:  # same row
                count += 1
            elif abs(i-j) == abs(s[i]-s[j]):  # same diagonal
                count += 1
    return -count  # minimize the conflicts

# Selection function: Tournament selection
def selection(pop):
    can1 = random.choice(pop)
    can2 = random.choice(pop)
    if can1['value'] > can2['value']:  # choose the better individual
        return can1
    return can2

# Crossover function: combining two parents into two offspring
def crossover(p1, p2):
    s1 = p1['state']
    s2 = p2['state']
    sc1 = s1[0:4] + s2[4:8]  # combine first half of p1 with second half of p2
    sc2 = s2[0:4] + s1[4:8]  # combine first half of p2 with second half of p1
    c1 = {'state': sc1, 'value': f(sc1)}  # create new individual
    c2 = {'state': sc2, 'value': f(sc2)}  # create new individual
    return c1, c2

# Mutation function: randomly change a position of one queen
def mutation(c):
    idx = random.randint(0, 7)
    newPos = random.randint(1, 8)
    c['state'][idx] = newPos

# Initialize population size and random population
N = 100
pop = [random.sample(range(1, 9), 8) for i in range(N)]  # generate random population
pop = [{'state': x, 'value': f(x)} for x in pop]  # calculate fitness

# Initialize the best solution
best = pop[0]
```

```
# GA loop for 200 generations
for i in range(200):
    newgen = []

    for j in range(int(N / 2)):  # pair up individuals to create new population
        p1 = selection(pop)
        p2 = selection(pop)
        c1, c2 = crossover(p1, p2)

        if random.random() < 0.1:  # 10% chance for mutation
            mutation(c1)
            mutation(c2)

        newgen += [c1, c2]  # add the offspring to the new generation

        # Keep track of the best solution found
        if c1['value'] > best['value']: best = c1
        if c2['value'] > best['value']: best = c2


    pop = newgen[:]  # update population
    print(i+1, 'total', sum (x['value'] for x in pop)/len(pop))
# Output the best solution found
print('genetic algorithm', best)
wan to print of all 200
```

If you want to print the best solution found for all 200 generations, you can add a print statement within the loop to show the current best solution at each generation. Here's how you can modify your code to achieve this:

```python
import random

# Fitness function: calculate the number of conflicts
def f(s):
    count = 0
    for i in range(8):
        for j in range(i+1, 8):
            if s[i] == s[j]:  # same row
                count += 1
            elif abs(i-j) == abs(s[i]-s[j]):  # same diagonal
                count += 1
    return -count  # minimize
```

want to print state of all 200

To print the state of the population at each generation (i.e., after every iteration of the loop), you can add a print statement inside your GA loop. This will print the state of the population or the best individual for each generation. Here's the updated version of your code:

```python
import random

# Fitness function: calculate the number of conflicts
def f(s):
    count = 0
    for i in range(8):
        for j in range(i+1, 8):
            if s[i] == s[j]:  # same row
                count += 1
            elif abs(i-j) == abs(s[i]-s[j]):  # same diagonal
                count += 1
    return -count  # minimize the conflicts

# Selection function: Tournament selection
def selection(pop):
    can1 = random.choice(pop)
    can2 = random.choice(pop)
    if can1['value'] > can2['value']:  # choose the better individual
        return can1
    return can2

# Crossover function: combining two parents into two offspring
def crossover(p1, p2):
    s1 = p1['state']
    s2 = p2['state']
    sc1 = s1[0:4] + s2[4:8]  # combine first half of p1 with second half of p2
    sc2 = s2[0:4] + s1[4:8]  # combine first half of p2 with second half of p1
    c1 = {'state': sc1, 'value': f(sc1)}  # create new individual
    c2 = {'state': sc2, 'value': f(sc2)}  # create new individual
    return c1, c2

# Mutation function: randomly change a position of one queen
def mutation(c):
    idx = random.randint(0, 7)
    newPos = random.randint(1, 8)
    c['state'][idx] = newPos

# Initialize population size and random population
N = 100
pop = [random.sample(range(1, 9), 8) for i in range(N)]  # generate random population
pop = [{'state': x, 'value': f(x)} for x in pop]  # calculate fitness

# Initialize the best solution
best = pop[0]

# GA loop for 200 generations
for i in range(200):
    newgen = []

    for j in range(int(N / 2)):  # pair up individuals to create new population
        p1 = selection(pop)
        p2 = selection(pop)
        c1, c2 = crossover(p1, p2)

        if random.random() < 0.1:  # 10% chance for mutation
            mutation(c1)
            mutation(c2)

        newgen += [c1, c2]  # add the offspring to the new generation

        # Keep track of the best solution found
        if c1['value'] > best['value']: best = c1
        if c2['value'] > best['value']: best = c2

    pop = newgen[:]  # update population

    # Print the state of the best individual for this generation
    print(f"Generation {i + 1} - Best State: {best['state']} with fitness: {best['value']}")
```

```
    # Optionally, print the average fitness of the population at each generation
    avg_fitness = sum(x['value'] for x in pop) / len(pop)
    print(f"Generation {i + 1} - Average fitness: {avg_fitness}")
```

Explanation of the changes:

1. **Print Best State at Each Generation**:
   - Inside the loop, after updating the population, I added a print statement to show the state of the best individual for that generation and its fitness (`` `best['state']` `` and `` `best['value']` ``).
2. **Print Average Fitness**:
   - I also included a print statement that calculates and shows the average fitness of the population at each generation. This can help you track how the population's overall fitness is improving over time.

Now, when you run this code, it will print the best solution and the average fitness for each of the 200 generations.

1 total -6.3
2 total -5.99
3 total -5.74
4 total -5.2
5 total -4.91
6 total -4.59
7 total -4.23
8 total -3.97
9 total -3.73
10 total -3.4
11 total -3.35
12 total -2.95
13 total -2.62
14 total -2.78
15 total -2.89
16 total -2.45
17 total -2.22
18 total -2.17
in this i need value of state too

To print the value of the state along with the fitness for each generation, you can modify the print statement to display both the best state and its corresponding fitness value. Based on the format you've provided, I assume you're looking for the best state at each generation along with the average fitness value.

Here's the updated version of the code to print the state and fitness at each generation:

```python
import random

# Fitness function: calculate the number of conflicts
```

```python
def f(s):
    count = 0
    for i in range(8):
        for j in range(i+1, 8):
            if s[i] == s[j]:  # same row
                count += 1
            elif abs(i-j) == abs(s[i]-s[j]):  # same diagonal
                count += 1
    return -count  # minimize the conflicts

# Selection function: Tournament selection
def selection(pop):
    can1 = random.choice(pop)
    can2 = random.choice(pop)
    if can1['value'] > can2['value']:  # choose the better individual
        return can1
    return can2

# Crossover function: combining two parents into two offspring
def crossover(p1, p2):
    s1 = p1['state']
    s2 = p2['state']
    sc1 = s1[0:4] + s2[4:8]  # combine first half of p1 with second half of p2
    sc2 = s2[0:4] + s1[4:8]  # combine first half of p2 with second half of p1
    c1 = {'state': sc1, 'value': f(sc1)}  # create new individual
    c2 = {'state': sc2, 'value': f(sc2)}  # create new individual
    return c1, c2

# Mutation function: randomly change a position of one queen
def mutation(c):
    idx = random.randint(0, 7)
    newPos = random.randint(1, 8)
    c['state'][idx] = newPos

# Initialize population size and random population
N = 100
pop = [random.sample(range(1, 9), 8) for i in range(N)]  # generate random population
pop = [{'state': x, 'value': f(x)} for x in pop]  # calculate fitness

# Initialize the best solution
best = pop[0]

# GA loop for 200 generations
for i in range(200):
    newgen = []

    for j in range(int(N / 2)):  # pair up individuals to create new population
        p1 = selection(pop)
        p2 = selection(pop)
        c1, c2 = crossover(p1, p2)

        if random.random() < 0.1:  # 10% chance for mutation
            mutation(c1)
            mutation(c2)

        newgen += [c1, c2]  # add the offspring to the new generation

        # Keep track of the best solution found
        if c1['value'] > best['value']: best = c1
        if c2['value'] > best['value']: best = c2

    pop = newgen[:]  # update population

    # Print the best individual state and its fitness for this generation
    print(f"Generation {i + 1} - Best State: {best['state']} with fitness: {best['value']}")

    # Optionally, print the average fitness of the population at each generation
    avg_fitness = sum(x['value'] for x in pop) / len(pop)
    print(f"Generation {i + 1} - Average fitness: {avg_fitness}")
```

What this update does:

- The **print statement** now prints the best state found in each generation along with its corresponding fitness.
- The format is `` `Generation {i + 1} - Best State: {best['state']} with fitness: {best['value']}` ``, so you can clearly see both the state (the board configuration) and the fitness value of the best solution at each generation.

Sample Output Format:

```yaml
Generation 1 - Best State: [3, 6, 8, 2, 4, 1, 7, 5] with fitness: -7
Generation 2 - Best State: [3, 6, 8, 2, 4, 1, 7, 5] with fitness: -6
Generation 3 - Best State: [3, 6, 8, 2, 4, 1, 7, 5] with fitness: -5
...
```

This code will print the state of the best individual found in each generation and the corresponding fitness score, as well as the average fitness of the population at that generation.

```
import random

# Fitness function: calculate the number of conflicts
def f(s):
    count = 0
    for i in range(8):
        for j in range(i+1, 8):
            if s[i] == s[j]:  # same row
                count += 1
            elif abs(i-j) == abs(s[i]-s[j]):  # same diagonal
                count += 1
    return -count  # minimize the conflicts

# Selection function: Tournament selection
def selection(pop):
    can1 = random.choice(pop)
    can2 = random.choice(pop)
    if can1['value'] > can2['value']:  # choose the better individual
        return can1
    return can2

# Crossover function: combining two parents into two offspring
def crossover(p1, p2):
    s1 = p1['state']
    s2 = p2['state']
    sc1 = s1[0:4] + s2[4:8]  # combine first half of p1 with second half of p2
    sc2 = s2[0:4] + s1[4:8]  # combine first half of p2 with second half of p1
    c1 = {'state': sc1, 'value': f(sc1)}  # create new individual
    c2 = {'state': sc2, 'value': f(sc2)}  # create new individual
    return c1, c2
```

```
# Mutation function: randomly change a position of one queen
def mutation(c):
    idx = random.randint(0, 7)
    newPos = random.randint(1, 8)
    c['state'][idx] = newPos

# Initialize population size and random population
N = 100
pop = [random.sample(range(1, 9), 8) for i in range(N)]  # generate random population
pop = [{'state': x, 'value': f(x)} for x in pop]  # calculate fitness

# Initialize the best solution
best = pop[0]

# GA loop for 200 generations
for i in range(200):
    newgen = []

    for j in range(int(N / 2)):  # pair up individuals to create new population
        p1 = selection(pop)
        p2 = selection(pop)
        c1, c2 = crossover(p1, p2)

        if random.random() < 0.1:  # 10% chance for mutation
            mutation(c1)
            mutation(c2)

        newgen += [c1, c2]  # add the offspring to the new generation

        # Keep track of the best solution found
        if c1['value'] > best['value']: best = c1
        if c2['value'] > best['value']: best = c2


    pop = newgen[:]  # update population
    print(i+1, 'total', sum (x['value'] for x in pop)/len(pop))
# Output the best solution found
print('genetic algorithm', best)
in the looping print statement also print corresponding state
```

To include the corresponding state in the print statement for each generation, you can simply modify the print statement inside the loop to display both the average fitness and the state of the best solution found during that generation.

Here's the updated code with the corresponding state printed alongside the fitness value:

```python
import random

# Fitness function: calculate the number of conflicts
def f(s):
```

```
    count = 0
    for i in range(8):
        for j in range(i+1, 8):
            if s[i] == s[j]:  # same row
                count += 1
            elif abs(i-j) == abs(s[i]-s[j]):  # same diagonal
                count += 1
    return -count  # minimize the conflicts

# Selection function: Tournament selection
def selection(pop):
    can1 = random.choice(pop)
    can2 = random.choice(pop)
    if can1['value'] > can2['value']:  # choose the better individual
        return can1
    return can2

# Crossover function: combining two parents into two offspring
def crossover(p1, p2):
    s1 = p1['state']
    s2 = p2['state']
    sc1 = s1[0:4] + s2[4:8]  # combine first half of p1 with second half of p2
    sc2 = s2[0:4] + s1[4:8]  # combine first half of p2 with second half of p1
    c1 = {'state': sc1, 'value': f(sc1)}  # create new individual
    c2 = {'state': sc2, 'value': f(sc2)}  # create new individual
    return c1, c2

# Mutation function: randomly change a position of one queen
def mutation(c):
    idx = random.randint(0, 7)
    newPos = random.randint(1, 8)
    c['state'][idx] = newPos

# Initialize population size and random population
N = 100
pop = [random.sample(range(1, 9), 8) for i in range(N)]  # generate random population
pop = [{'state': x, 'value': f(x)} for x in pop]  # calculate fitness

# Initialize the best solution
best = pop[0]

# GA loop for 200 generations
for i in range(200):
    newgen = []

    for j in range(int(N / 2)):  # pair up individuals to create new population
        p1 = selection(pop)
        p2 = selection(pop)
        c1, c2 = crossover(p1, p2)

        if random.random() < 0.1:  # 10% chance for mutation
            mutation(c1)
            mutation(c2)

        newgen += [c1, c2]  # add the offspring to the new generation

        # Keep track of the best solution found
        if c1['value'] > best['value']: best = c1
        if c2['value'] > best['value']: best = c2

    pop = newgen[:]  # update population

    # Print the average fitness and the best individual state for this generation
    avg_fitness = sum(x['value'] for x in pop) / len(pop)
    print(f"Generation {i + 1} - Average fitness: {avg_fitness:.2f} | Best State: {best['state']} |

# Output the best solution found
print('Final Best Solution:', best)
```

Key Changes:

1. **Print Statement Update**:
   In the GA loop, I updated the print statement to:

   ```python
   print(f"Generation {i + 1} - Average fitness: {avg_fitness:.2f} | Best State: {best['state']}
   ```

   This will print:

   - The generation number
   - The average fitness of the population
   - The best state found at that generation
   - The fitness of that best state
2. **Formatting**:
   The fitness values are printed with two decimal places (`avg_fitness:.2f`), but you can adjust the precision as needed.

Sample Output Format:

```yaml
Generation 1 - Average fitness: -6.33 | Best State: [3, 6, 8, 2, 4, 1, 7, 5] | Best Fitness: -7
Generation 2 - Average fitness: -5.99 | Best State: [2, 5, 8, 3, 1, 6, 4, 7] | Best Fitness: -6
Generation 3 - Average fitness: -5.50 | Best State: [4, 6, 7, 1, 3, 5, 8, 2] | Best Fitness: -5
...
```

This modification ensures that both the state and its corresponding fitness value are printed after each generation.

i dont get it what is teh avg fitness and hwat is the best fitness

Let me explain the difference between **average fitness** and **best fitness** in the context of your genetic algorithm (GA) and how they relate to your solution.

1. Best Fitness:

- **Best Fitness** refers to the fitness of the best individual (solution) found in the population at a particular generation.
- The **fitness** of an individual is determined by how many conflicts there are in the board arrangement (in the case of the N-Queens problem, this would be how many queens are attacking each other).
- In your code, `best` is a dictionary that contains the state of the best individual and its fitness score. The state refers to the positions of the queens on the board, and the fitness value is the number of conflicts.

The **best fitness** will be the fitness of the individual in the population that currently has the least number of conflicts (most optimal solution).

2. Average Fitness:

- **Average Fitness** refers to the average fitness of the entire population for that generation.
- The **average fitness** is calculated by summing the fitness values of all individuals in the population and dividing that sum by the number of individuals. This gives you an idea of how well the population is performing on average.

What Is Happening in the Code?

1. **Best Fitness**:
   - The variable `best` stores the best individual found so far. The fitness of this best individual is printed with each generation.
   - This individual could be the one that has the least number of conflicts or attacks (i.e., the most optimal state of queens on the board).
2. **Average Fitness**:
   - For every generation, the code calculates the average fitness of the entire population. The average fitness is the sum of the fitness values of all individuals divided by the population size (100 in this case).
   - This helps you see how the overall population is improving or changing over time. If the population is improving, the average fitness should be getting better (less negative in your case, since you are minimizing conflicts).

Example Output Explanation:

Suppose you have the following output:

```yaml
Generation 1 - Average fitness: -6.33 | Best State: [3, 6, 8, 2, 4, 1, 7, 5] | Best Fitness: -7
Generation 2 - Average fitness: -5.99 | Best State: [2, 5, 8, 3, 1, 6, 4, 7] | Best Fitness: -6
Generation 3 - Average fitness: -5.50 | Best State: [4, 6, 7, 1, 3, 5, 8, 2] | Best Fitness: -5
...
```

- **Generation 1**:
  - **Average Fitness**: -6.33: This means that, on average, all individuals in the population have a fitness of -6.33 (i.e., the average number of conflicts in the entire population).
  - **Best Fitness**: -7: This means the best individual in the population has a fitness of -7, indicating it has 7 conflicts.
  - **Best State**: [3, 6, 8, 2, 4, 1, 7, 5]: This is the specific arrangement of queens on the board for the best individual.
- **Generation 2**:
  - **Average Fitness**: -5.99: The average fitness of the population has slightly improved from -6.33 to -5.99.
  - **Best Fitness**: -6: The best individual now has 6 conflicts, which is slightly better than the previous best fitness of -7.
- **Generation 3**:
  - **Average Fitness**: -5.50: The average fitness has improved again, showing that the population is making progress toward better solutions.
  - **Best Fitness**: -5: The best individual now has only 5 conflicts, which is an improvement from the previous best fitness.

Why Track Both Best Fitness and Average Fitness?

- **Best Fitness** gives you the *best solution* found in the population. This is the one you're interested in because it represents the individual with the least conflicts (most optimal).
- **Average Fitness** shows the *overall progress* of the population. If the average fitness is improving over time, it suggests that the algorithm is making progress and that the population is moving toward better solutions.

Key Point:

- The **best fitness** is the fitness of the best individual in the population.
- The **average fitness** is the average of the fitness values of all individuals in the population.

Does this help clarify the difference between the two?

give aniother example to relatge pls

Sure! Let's use a different analogy to help clarify the concept of **best fitness** and **average fitness**. We'll use an analogy of **students' test scores** for comparison.

Analogy: Students in a Class Taking a Test

Imagine you have a class of 10 students, and they all take a test. Their scores (out of 100) represent their "fitness" in this analogy. In the case of your genetic algorithm, instead of test scores, it's the number of conflicts (or lack of conflicts) in a solution.

Here's a hypothetical situation for 3 students' scores:

- Student 1: 75 (good score, few mistakes)
- Student 2: 85 (even better score, fewer mistakes)
- Student 3: 60 (not so good, more mistakes)

Best Fitness (Best Test Score):

- **Best Fitness** in this case is the highest score among all students.
    - In this case, the best score is **85** from Student 2, so the **best fitness** for this group of students is **85**. The student with the best score is the "best individual" in the population.
    - **Best Fitness = 85**

Average Fitness (Average Test Score):

- **Average Fitness** is calculated by adding up all the students' scores and dividing by the number of students:

$$\text{Average Fitness} = \frac{75 + 85 + 60}{3} = \frac{220}{3} = 73.33$$

So the **average fitness** of the students is **73.33**. This represents the overall performance of the class on the test.

Now, How Does This Relate to Your Genetic Algorithm?

- **Best Fitness**: In the context of your genetic algorithm, this corresponds to the individual (or solution) in the population that has the highest fitness value (i.e., the least number of conflicts). It tells you how well the best solution in the population is performing.
  - Example: The **best fitness** might be -5, meaning that the best individual has 5 conflicts (fewer is better). Over time, the **best fitness** will improve as the algorithm evolves better solutions.
- **Average Fitness**: This corresponds to the average performance of all individuals in the population. Just like the average test score, it tells you how the entire population is doing on average.
  - Example: The **average fitness** could be -7, meaning that, on average, the population of solutions has 7 conflicts. If this number decreases over time, it suggests the population is improving as a whole.

Breakdown of an Example:

Let's say you are running the genetic algorithm for **3 generations**. Here's how things might look:

Generation 1:

- Best Fitness = -6 (Best solution has 6 conflicts)
- Average Fitness = -6.33 (The entire population of solutions has an average of 6.33 conflicts)

Generation 2:

- Best Fitness = -5 (Best solution now has 5 conflicts)
- Average Fitness = -6.00 (The population's average fitness improves slightly)

Generation 3:

- Best Fitness = -4 (Best solution now has 4 conflicts)
- Average Fitness = -5.75 (The population’s average fitness improves even more)

Visual Example with Population (Hypothetical Fitness Values):

Let’s say the population contains 5 individuals in each generation, and we track both **best fitness** and **average fitness**:

Generation 1 (Initial Population):

Individual	Fitness (Conflicts)
Individual 1	-5
Individual 2	-6
Individual 3	-7
Individual 4	-5
Individual 5	-8

- **Best Fitness**: The best individual in Generation 1 has **-5** fitness (least number of conflicts).
- **Average Fitness**: The average fitness of all individuals is:

$$\text{Average Fitness} = \frac{-5 + (-6) + (-7) + (-5) + (-8)}{5} = \frac{-31}{5} = -6.2$$

Generation 2 (After Crossover/Mutation):

Individual	Fitness (Conflicts)
Individual 1	-4
Individual 2	-6
Individual 3	-6
Individual 4	-5
Individual 5	-7

- **Best Fitness**: The best individual now has **-4** fitness (fewer conflicts).
- **Average Fitness**: The average fitness of the new population is:

$$\text{Average Fitness} = \frac{-4 + (-6) + (-6) + (-5) + (-7)}{5} = \frac{-28}{5} = -5.6$$

Generation 3 (Further Improvement):

Individual	Fitness (Conflicts)
Individual 1	-3
Individual 2	-5
Individual 3	-4
Individual 4	-4
Individual 5	-6

- **Best Fitness**: The best individual now has **-3** fitness (only 3 conflicts).
- **Average Fitness**: The average fitness of the population is:

$$\text{Average Fitness} = \frac{-3 + (-5) + (-4) + (-4) + (-6)}{5} = \frac{-22}{5} = -4.4$$

Conclusion:

- **Best Fitness** tells you about the **best individual** solution's fitness in the population. In this example, it improves over generations, as the genetic algorithm finds better solutions.
- **Average Fitness** gives you an idea of how the entire population is performing on average. If the average fitness is improving (getting less negative), it suggests that the genetic algorithm is making progress.

In summary:

- **Best Fitness**: Best solution's fitness (lowest number of conflicts).
- **Average Fitness**: The average of all individuals' fitness values in the population.

Does this analogy make the concept clearer?

make me understand in my code what is 100 and what is 200

Sure! Let me explain the meaning of the numbers **100** and **200** in the context of your code:

1. 100 (Population Size):

In your code, the number **100** represents the **population size**. It refers to the total number of individuals (solutions) in your population at the start of each generation. The population evolves through selection, crossover, and mutation, and the genetic algorithm uses this group of individuals to find better solutions over time.

In your code:

```python
N = 100  # Population size
pop = [random.sample(range(1, 9), 8) for i in range(N)]  # generate random population
```

- **100** indicates that you will have 100 randomly generated individuals (solutions) at the start.
- Each individual is represented by a list of 8 integers (positions of queens on a chessboard).
- **Population size = 100** means you have 100 candidate solutions (i.e., 100 different ways of arranging the queens).

2. 200 (Number of Generations):

The number **200** represents the **number of generations** (iterations) the genetic algorithm will run through to evolve the population of solutions. The algorithm will repeat the process of selection, crossover, and mutation for **200 generations** to improve the population's fitness.

In your code:

```python
for i in range(200):  # GA loop for 200 generations
    ...
```

- **200** indicates that the genetic algorithm will run for **200 generations**.
- Each generation consists of pairing up individuals for crossover, applying mutation, and replacing the old population with the new population.
- After **200 generations**, the algorithm will output the best solution it found during the entire process.

What Happens in the Loop?

In the loop where the algorithm evolves the population over generations:

- **100**: Refers to the number of solutions (individuals) in the population in each generation.
- **200**: Refers to the number of times the algorithm will perform the evolution process (i.e., for **200 generations**).

Each time the loop runs (for each generation):

1. The **selection** function chooses two individuals from the population.
2. The **crossover** function combines their states (solutions) to produce two offspring.
3. The **mutation** function sometimes applies small random changes to the offspring.
4. These offspring replace the previous population, and the algorithm proceeds to the next generation.

At the end of **200 generations**, the algorithm will have a "best" solution (the one with the least number of conflicts) and the final **best fitness** for that solution.

Summary:

- **100**: This is the population size, meaning there are 100 individuals in the population at each generation.
- **200**: This is the number of generations the algorithm will run, meaning it will evolve the population for 200 iterations.

In short:

- **100** is how many possible solutions (individuals) you have.
- **200** is how many times the algorithm tries to improve those solutions (through generations).