

**Thanarit Kanjanameyawat ID: 6410322**

**Lynn Thit Nyi Nyi ID: 6411271**

## 1. Introduction

The provided code is a Python implementation of a genetic algorithm to solve the problem of balancing the weights of stones between two piles, Pile A and Pile B. The goal is to distribute the stones between the two piles in a way that minimizes the weight difference between them. The genetic algorithm is an optimization technique inspired by natural selection and evolutionary processes.

## 2. Problem Statement

The problem can be described as follows:

- There are 'n' stones, each with a given weight. The weights of the stones are provided as input to the algorithm.
- The genetic algorithm aims to find a binary chromosome of length 'n' that represents how the stones are distributed between Pile A and Pile B.
- Each element in the chromosome corresponds to a stone, where '1' indicates the stone is in Pile A, and '0' indicates the stone is in Pile B.
- The fitness of a chromosome is calculated based on the absolute difference between the weights of Pile A and Pile B. Lower weight difference results in a higher fitness value.

## 3. Fitness function

**Before anything, the most critical part of the genetic algorithm is the fitness function. In this implementation, the fitness function is defined as follows:**

```
8 # Define the fitness function for a chromosome
9 def fitness(ch):
10     # Calculate the weight of pileA based on the given chromosome
11     pileA_weight = sum(weights[i] for i, val in enumerate(ch) if val == 1)
12     # Return the inverse of the absolute difference between the weights of the two piles (+1 to avoid division by zero)
13     return 1 / (abs(pileA_weight - (sum(weights) - pileA_weight)) + 1)
14
```

The 'fitness' function takes a binary chromosome 'ch' as input and calculates the fitness value for that chromosome.

It first calculates the total weight of stones in Pile A ('pileA\_weight') by summing the weights of stones with '1' in the chromosome.

Then, it calculates the total weight of stones in Pile B by subtracting 'pileA\_weight' from the total weight of all stones ('sum(weights) - pileA\_weight').

The fitness value is calculated as the inverse of the absolute difference between the weights of Pile A and Pile B, plus one to avoid division by zero.

The fitness function rewards chromosomes that achieve a more balanced distribution of stone weights between the two piles, resulting in a higher fitness value.

#### 4. Genetic Algorithm Code Explanation

- Import random for the genetic algorithm and read inputs

```
1  import random
2
3  # Read the number of stones and their weights
4  n = int(input().strip()) # number of stones
5  weights = list(map(int, input().split()))
6
7
```

- Fitness function (as previously explained)

```
8  # Define the fitness function for a chromosome
9  def fitness(ch):
10     # Calculate the weight of pileA based on the given chromosome
11     pileA_weight = sum(weights[i] for i, val in enumerate(ch) if val == 1)
12     # Return the inverse of the absolute difference between the weights of the two piles
13     return 1 / (abs(pileA_weight - (sum(weights) - pileA_weight)) + 1)
14
```

- Defining the chromosome class

```
16  # Define the Chromosome class
17  class Chromosome:
18     def __init__(self, chrom=None):
19         # Initialize a chromosome, either from given list or randomly
20         self.chrom = chrom if chrom else [random.randint(0, 1) for _ in range(n)]
21         # Calculate the fitness for this chromosome
22         self.fit = fitness(self.chrom)
23
24
```

This block defines the Chromosome class, representing a binary chromosome that represents the distribution of stones between Pile A and Pile B. Each chromosome has 'chrom' (the binary list) and 'fit' (its fitness value) attributes. The constructor `__init__` initializes the chromosome either from a given list ('chrom') or randomly if 'chrom' is not provided. It also calculates the fitness value using the 'fitness' function defined earlier.

#### - Defining the crossover function

```
25  # Define the crossover function to generate offspring from parents
26  def crossover(parents):
27      # Choose a random crossover point
28      x = random.randint(0, n - 1)
29      # Create two offspring using the crossover point
30      offspring1 = parents[0].chrom[:x + 1] + parents[1].chrom[x + 1:]
31      offspring2 = parents[1].chrom[:x + 1] + parents[0].chrom[x + 1:]
32      return offspring1, offspring2
33
34
```

This block defines the crossover function 'crossover(parents)', which generates two offspring chromosomes from two parent chromosomes. The crossover point 'x' is randomly chosen between 0 and 'n-1' (inclusive). The offspring chromosomes are created by swapping the chromosome segments before and after the crossover point between the two parents.

#### - Defining the mutation function

```
35  # Define the mutation function
36  def mutate(ch):
37      # Randomly select a gene and flip it (0 to 1 or 1 to 0)
38      i = random.randint(0, n - 1)
39      ch[i] = random.randint(0, 1)
40
41
```


This block defines the mutation function 'mutate(ch)', which randomly flips a gene (bit) in the given chromosome 'ch'. It selects a random index 'i' between 0 and 'n-1' and flips the bit at that position.

#### - Defining the selection function

```
42  # Define the selection function using a stochastic acceptance method
43  def select(population, total_fitness):
44      while True:
45          # Randomly select a chromosome
46          chosen = random.choice(population)
47          # Select the chromosome based on its fitness proportionally
48          if random.uniform(0, 1) < chosen.fit / total_fitness:
49              return chosen
50
51
```

This block defines the selection function 'select(population, total\_fitness)', which uses a stochastic acceptance method to select a chromosome from the given population based on its fitness value. The higher the fitness value of a chromosome, the more likely it is to be chosen.

## - Genetic Algorithm Parameters and Initialization

```
52  # Parameters for the genetic algorithm
53  n_pop = 200
54  mut_prob = 0.2
55  max_gen = 1000
56  plateau_count = 0
57
58   Initialize a random population of chromosomes
59  population = [Chromosome() for _ in range(n_pop)]
60  # Sort the population based on their fitness in descending order
61  population.sort(key=lambda x: x.fit, reverse=True)
62
63  # Track the previous and current maximum fitness in the population
64  old_max = 0
65  new_max = population[0].fit
66  gen = 1
67
```

## - Genetic Algorithm Main Loop

```
68  # Genetic algorithm main loop
69  while plateau_count < 5 and gen <= max_gen:
70      new_gen = []
71      # Calculate the total fitness of the current population
72      total_fitness = sum(ch.fit for ch in population)
73      for _ in range(n_pop // 2):
74          # Select two parents and perform crossover and mutation
75          parents = [select(population, total_fitness) for _ in range(2)]
76          offspring_chroms = crossover(parents)
77          offspring = []
78          for chrom in offspring_chroms:
79              if random.uniform(0, 1) < mut_prob:
80                  mutate(chrom)
81              offspring.append(Chromosome(chrom))
82          new_gen += offspring
83      # Merge the old and new generations, and select the top n_pop chromosomes
84      population = sorted(population + new_gen, key=lambda x: x.fit, reverse=True)[:n_pop]
85
86      # Track and compare fitness to detect convergence
87      old_max = new_max
88      new_max = population[0].fit
89      if new_max <= old_max:
90          plateau_count += 1
91      else:
92          plateau_count = 0
93      gen += 1
```

This block represents the main loop of the genetic algorithm. It continues until either the maximum number of generations ('max\_gen') is reached or the fitness value remains constant for 'plateau\_count' consecutive generations.

Within each generation, a new generation ('new\_gen') is created through selection, crossover, and

mutation. The 'total\_fitness' variable stores the sum of fitness values of all chromosomes in the population. For each pair of parent chromosomes, two offspring are created using the 'crossover' function. Each offspring has a chance to undergo mutation with a probability of 'mut\_prob' using the 'mutate' function. The offspring are then added to the 'new\_gen' list.

After creating the new generation, the 'population' is updated by combining the current population and the new generation, then sorting them based on fitness to select the top 'n\_pop' chromosomes for the next generation.

The algorithm tracks the maximum fitness value to detect convergence. If the maximum fitness remains the same for 'plateau\_count' generations, it indicates convergence, and the 'plateau\_count' is incremented. Otherwise, 'plateau\_count' is reset to zero.

- Extract best solution

```
95 # Extract the best solution and compute the two piles
96 best_chromosome = population[0].chrom
97 pileA = [weights[i] for i, val in enumerate(best_chromosome) if val == 1]
98 pileB = [weights[i] for i, val in enumerate(best_chromosome) if val == 0]
99 difference = abs(sum(pileA) - sum(pileB))
100
101 # Print the results
102 print(f"Weight Difference: {difference}")
103 print(f"Pile A: {pileA}")
104 print(f"Pile B: {pileB}")
```

Note: In the online judge submission, we modified the print statements to match the expected output.

## 5. Test Cases

1. 5 stones (default test case as shown in the problem, matches output and is consistent)

```
5.in
1 5
2 5 8 13 27 14
```

```
D:\School work\Year 3\Artificial Intelligence Concepts\Assignment>solution.py < 5.in
Weight Difference: 3
Pile A: [8, 27]
Pile B: [5, 13, 14]
```

## 2. 30 stones (consistent output with different partitioning)

```
30.in
1 30
2 25 40 31 27 36 20 22 35 45 33 30 41 38 29 24 39 43 26 34 32 37 23 28 42 46 21 44 48 47 49
3

D:\School work\Year 3\Artificial Intelligence Concepts\Assignment>solution.py < 30.in
Weight Difference: 1
Pile A: [25, 31, 27, 20, 22, 35, 45, 38, 43, 26, 34, 37, 42, 46, 47]
Pile B: [40, 36, 33, 30, 41, 29, 24, 39, 32, 23, 28, 21, 44, 48, 49]

D:\School work\Year 3\Artificial Intelligence Concepts\Assignment>solution.py < 30.in
Weight Difference: 1
Pile A: [25, 31, 27, 36, 22, 35, 45, 33, 38, 29, 26, 32, 23, 21, 48, 47]
Pile B: [40, 20, 30, 41, 24, 39, 43, 34, 37, 28, 42, 46, 44, 49]
```

## 3. 50 stones (output starts getting different in different runtimes)

```
50
57882 14378 95797 85888 13120 90177 89864 87827 310 9413 20809 94580 95868 60482 13347 83124 95514 70461 95839 68190
3

D:\School work\Year 3\Artificial Intelligence Concepts\Assignment>solution.py < 50.in
Weight Difference: 351
Pile A: [95797, 90177, 9413, 94580, 95868, 60482, 13347, 83124, 95839, 68190, 73156, 69647, 44993, 71175, 87218, 55337, 76745, 94524, 94349, 54138, 2243, 9340, 8158]
Pile B: [57882, 14378, 85888, 13120, 89864, 87827, 310, 20809, 95514, 70461, 39764, 71931, 47331, 30325, 2467, 71573, 27509, 53147, 80741, 36210, 93552, 94275, 89961, 47939, 71795, 25800, 27116]

D:\School work\Year 3\Artificial Intelligence Concepts\Assignment>solution.py < 50.in
Weight Difference: 447
Pile A: [57882, 95797, 85888, 13120, 89864, 13347, 83124, 68190, 39764, 73156, 69647, 44993, 71573, 27509, 71175, 53147, 80741, 93552, 94349, 54138, 94275, 2243, 9340, 25800, 27116, 8158]
Pile B: [14378, 90177, 87827, 310, 9413, 20809, 94580, 95868, 60482, 95514, 70461, 95839, 71931, 47331, 30325, 2467, 87218, 55337, 76745, 36210, 94524, 89961, 47939, 71795]
```

## 6. Evaluation

Problem	Language	Judgement result	Test #	Execution time	Memory used
<a href="#">1005. Stone Pile</a>	Python 3.8 x64	Time limit exceeded	3	1.046	572 KB

Even though the runtime doesn't take long, in some test cases the process gets computationally intensive, exceeding the expected time limit from the online judge, especially since we used a genetic algorithm approach.

1. Parameters: The parameters used in the solution code may not be suitable for all test cases leading to inconsistencies in the algorithm, increasing the computational intensity as a result.
2. Randomness and Seed: The genetic algorithm involves random processes like initialization, selection, and mutation. The results can vary between runs due to this randomness. In some cases, an unfortunate combination of random events might lead to slower convergence or getting stuck in suboptimal solutions, and therefore increasing the time limit in the online judge.

All in all, genetic algorithms are not guaranteed to find the optimal solution in every case, especially for complex problems or large solution spaces. It does not guarantee to find the

optimal solution even with small test cases. Using a dynamic programming approach could have been a more optimal approach for this problem, guaranteeing an optimal solution for smaller instances and avoiding redundant calculations through memorization.