In [1]:
```
import random
import datetime
import string
from itertools import dropwhile, accumulate, count

GENES = string.ascii_uppercase + string.ascii_lowercase

class Organism:
    mutation_rate = 0.001

    def __init__(self, genes):
        self.genes = genes

    def fitness(self, ideal):
        return sum(self_gene == ideal_gene
                   for self_gene, ideal_gene in zip(self.genes, ideal.genes))

    def __str__(self):
        return self.genes

    @classmethod
    def from_random_genes(cls, target_length):
        return cls(''.join(random.choices(GENES, k=target_length)))

    @classmethod
    def from_parents(cls, parent1, parent2):
        cross_over_point = random.randrange(0, len(parent1.genes))
        new_genes = list(parent1.genes[:cross_over_point] + parent2.genes[cros
s_over_point:])
        for i, _ in enumerate(new_genes):
            if random.randint(0, int(1 / cls.mutation_rate)) == 0:
                new_genes[i] = random.choice(GENES)
        return cls(''.join(new_genes))


def create_initial_generation(population_size, target_length):
    return [Organism.from_random_genes(target_length) for _ in range(populatio
n_size)]


def evaluate_organisms(organisms, ideal_organism):
    return [organism.fitness(ideal_organism) for organism in organisms]


def select_parent(fitness):
    """A utility function that decide which parent to select for crossover.

    Based on Roulette Wheel Sampling
    """
    pick = random.randint(0, sum(fitness))
    return next(dropwhile(lambda x: x[1] < pick, enumerate(accumulate(fitness
))))[0]


def produce_next_generation(current_generation, fitness):
    """A utility function to perform crossover."""
```

```
    next_generation = []
    for _ in range(len(current_generation)):
        # select two parents using Roulette Wheel Sampling
        parent1 = current_generation[select_parent(fitness)]
        parent2 = current_generation[select_parent(fitness)]
        next_generation.append(Organism.from_parents(parent1, parent2))
    return next_generation

def break_pw_genetic(target):
    population_size = 10 * len(target)
    ideal_organism = Organism(target)
    current_generation = create_initial_generation(population_size, len(target
))
    for generation in count():
        fitness = evaluate_organisms(current_generation, ideal_organism)
        # print(max(fitness), max(current_generation, key=lambda organism: org
anism.fitness(ideal_organism)))
        if max(fitness) == len(target):
            break
        current_generation = produce_next_generation(current_generation, fitne
ss)
    return generation

if __name__ == '__main__':
    start_time = datetime.datetime.now()
    generation = break_pw_genetic('Colorado')
    duration = datetime.datetime.now() - start_time

    print(f'Program terminated after {generation} generations.')
    print(f'Time Taken: {duration} seconds')
```

```
Program terminated after 2719 generations.
Time Taken: 0:00:05.177675 seconds
```

In [ ]: