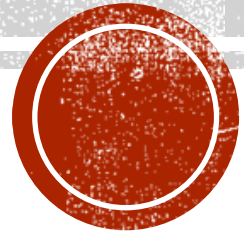
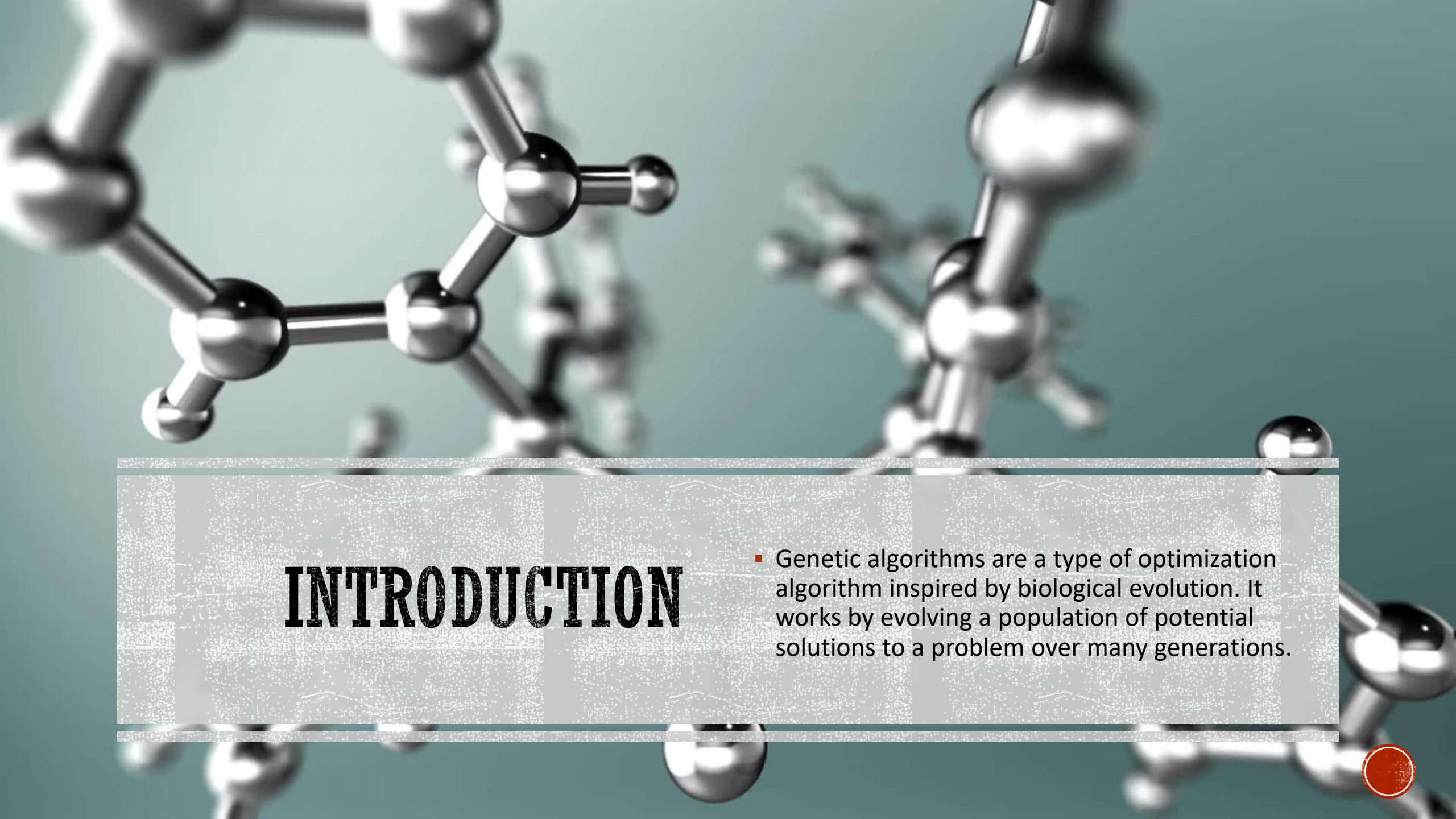
LECTURE # 05
CENETIC ALGORITHM





INTRODUCTION

- Genetic algorithms are a type of optimization algorithm inspired by biological evolution. It works by evolving a population of potential solutions to a problem over many generations.

WHAT IS MEANT BY OPTIMIZATION:

Optimization

Optimization is the process of **making something better**

Finding the values of inputs in such a way that we get the “best” output values

TERMINOLOGIES:

- Population
- Chromosomes
- Gene

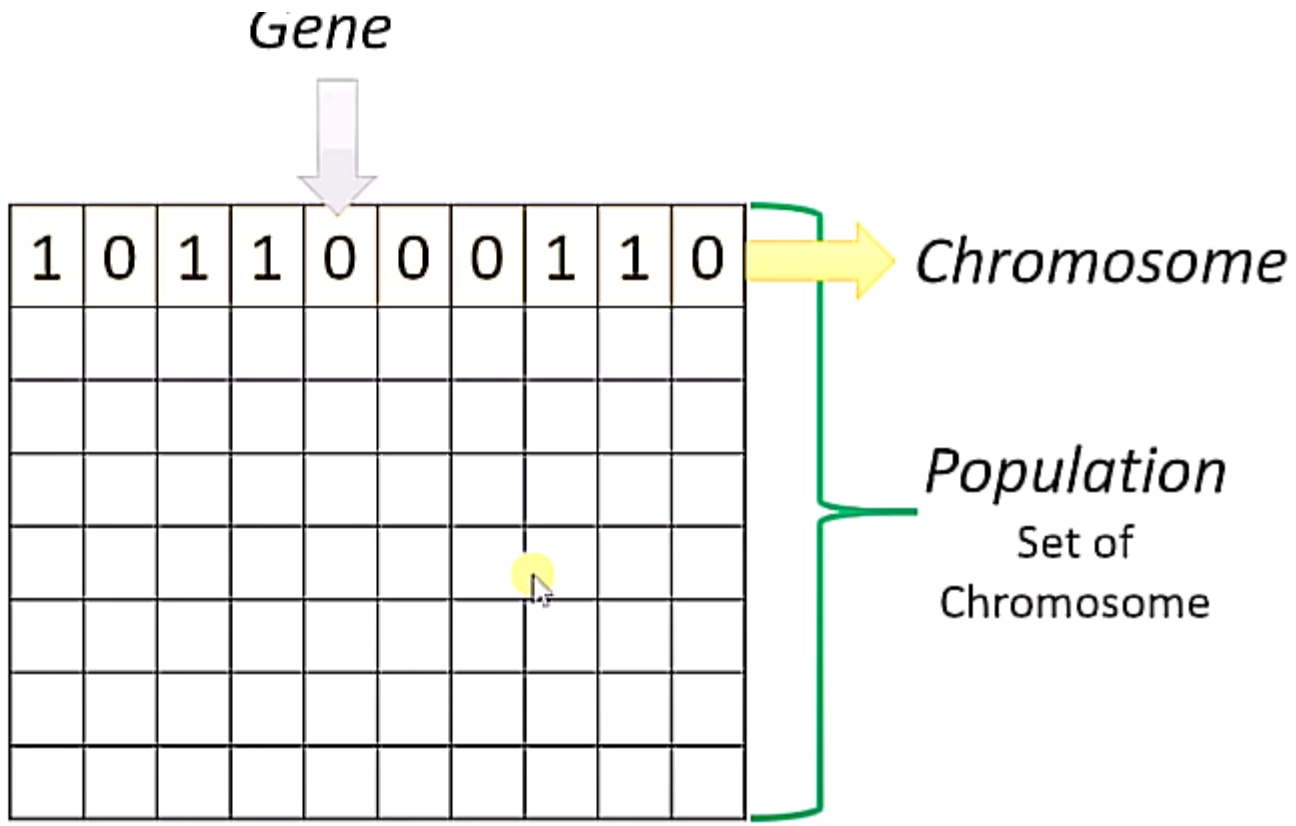
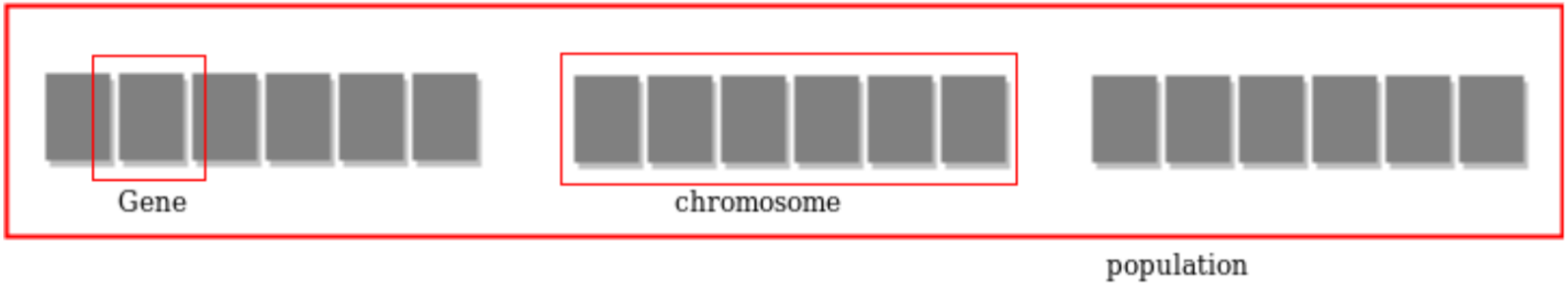


ILLUSTRATION:

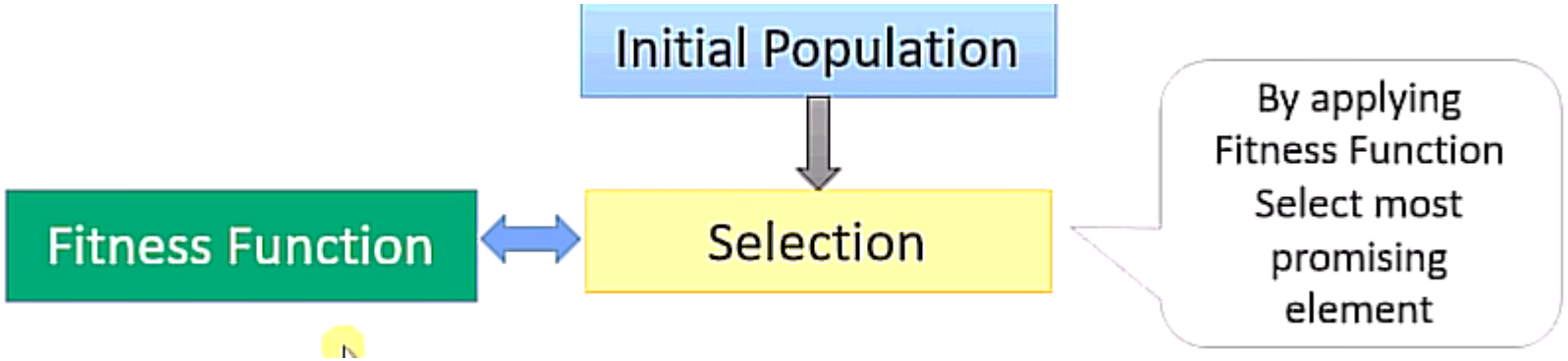


OPERATORS:

- The main operators used in genetic algorithms are:

1. **Selection**: This involves selecting chromosomes from the current population for reproduction. Fitness proportionate selection is commonly used, where higher fit chromosomes have a higher probability of selection.
2. **Crossover:** This operator mates two selected chromosomes and produces offspring with traits from both parents. A common one-point crossover randomly selects a point and swaps genes between parents after that point to form two offspring.
3. **Mutation:** This randomly alters some genes in a chromosome with a low probability. It introduces randomness to maintain diversity. Common types are bit flipping for binary strings or uniform mutation that randomly alters the value of a gene.
4. **Elitism:** The fittest chromosome(s) from the current generation are directly copied over to the next generation without undergoing crossover and mutation. This ensures the best solution is not lost.

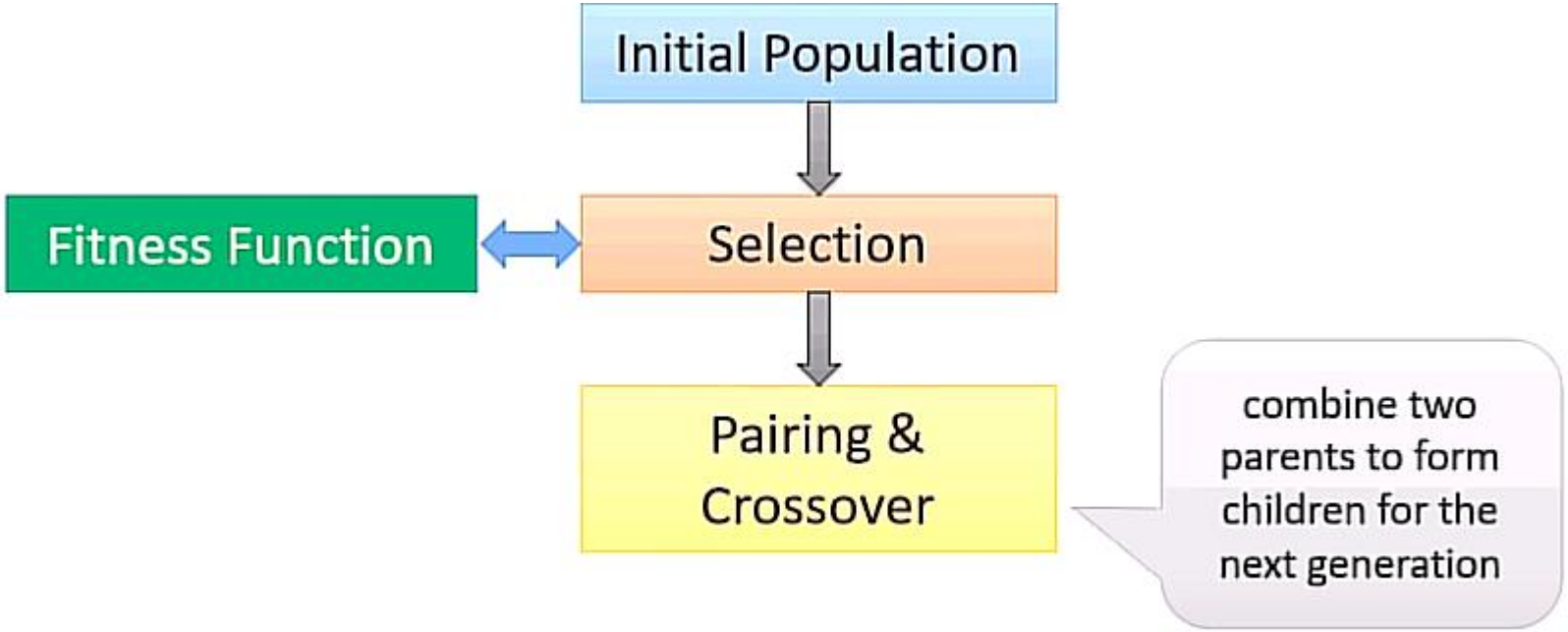
SELECTION:


Initial Population
Fitness Function
Selection
By applying Fitness Function Select most promising element

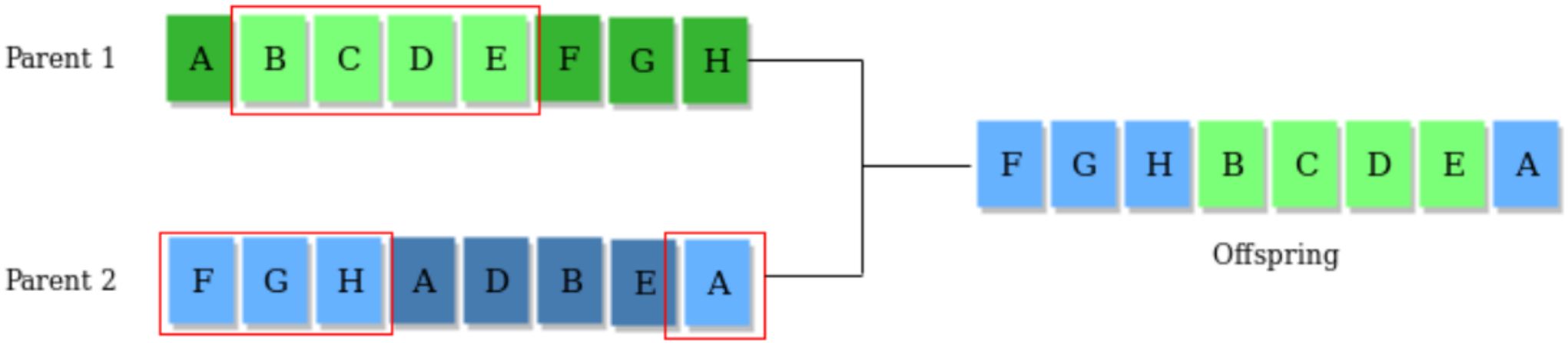
FITNESS FUNCTION IN GA

- A Fitness Score is given to each individual which **shows the ability of an individual to "compete"**. The individual having optimal fitness score (or near optimal) are sought.
- The GAs maintains the population of n individuals (chromosome/solutions) along with their fitness scores. The individuals having better fitness scores are given more chance to reproduce than others. The individuals with better fitness scores are selected who mate and produce **better offspring** by combining chromosomes of parents. The population size is static so the room must be created for new arrivals. So, some individuals die and get replaced by new arrivals eventually creating new generation when all the mating opportunity of the old population is exhausted. It is hoped that over successive generations better solutions will arrive while least fit die.

CROSS-OVER OPERATOR:

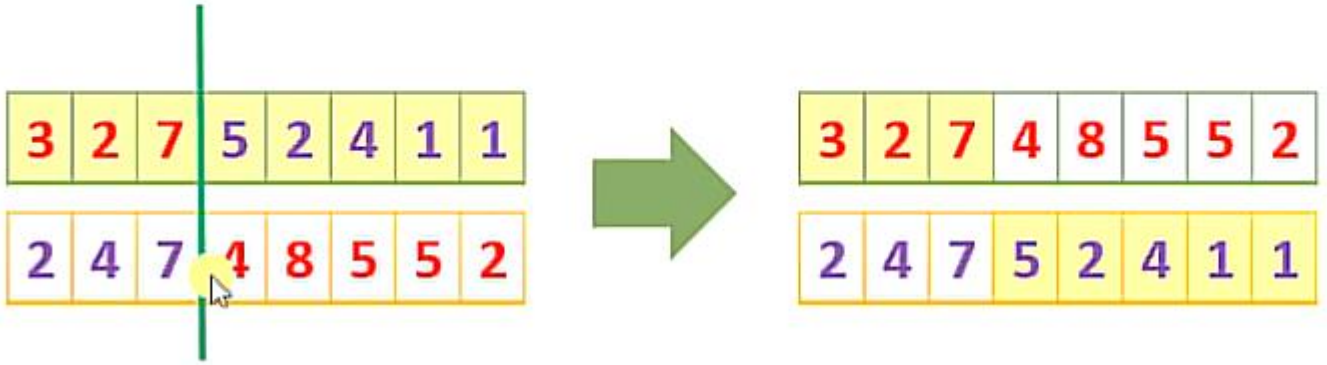


CROSS-OVER OPERATOR

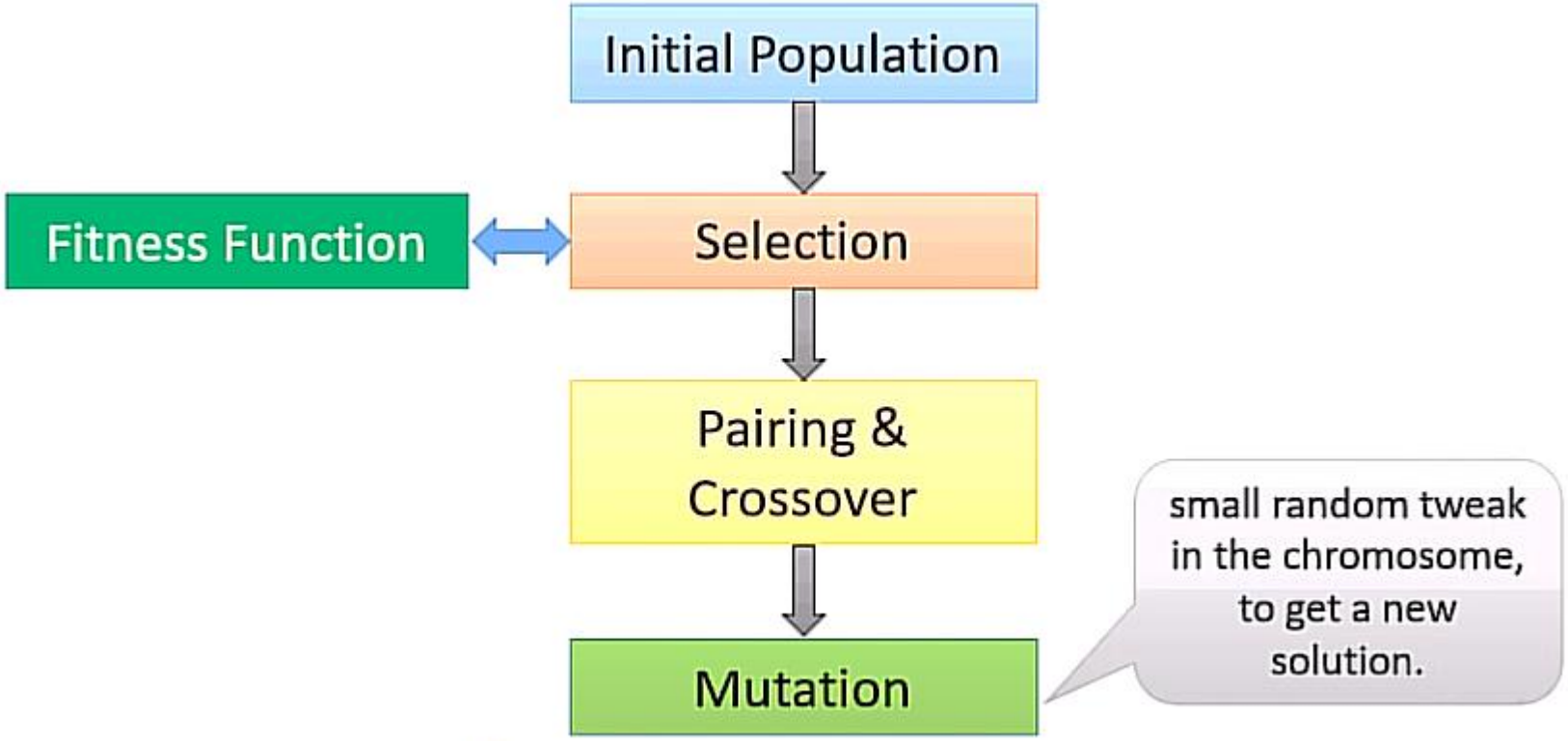


TECHNIQUES FOR CROSSOVER:

- *One Point Crossover* : A random crossover point is selected and the tails of its two parents are swapped to get new off-springs.



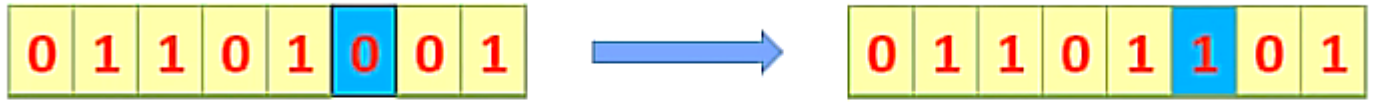
MUTATION:


Initial Population
Fitness Function
Selection
Pairing &
Crossover
Mutation
small random tweak
in the chromosome,
to get a new
solution.

MUTATION OPERATOR:

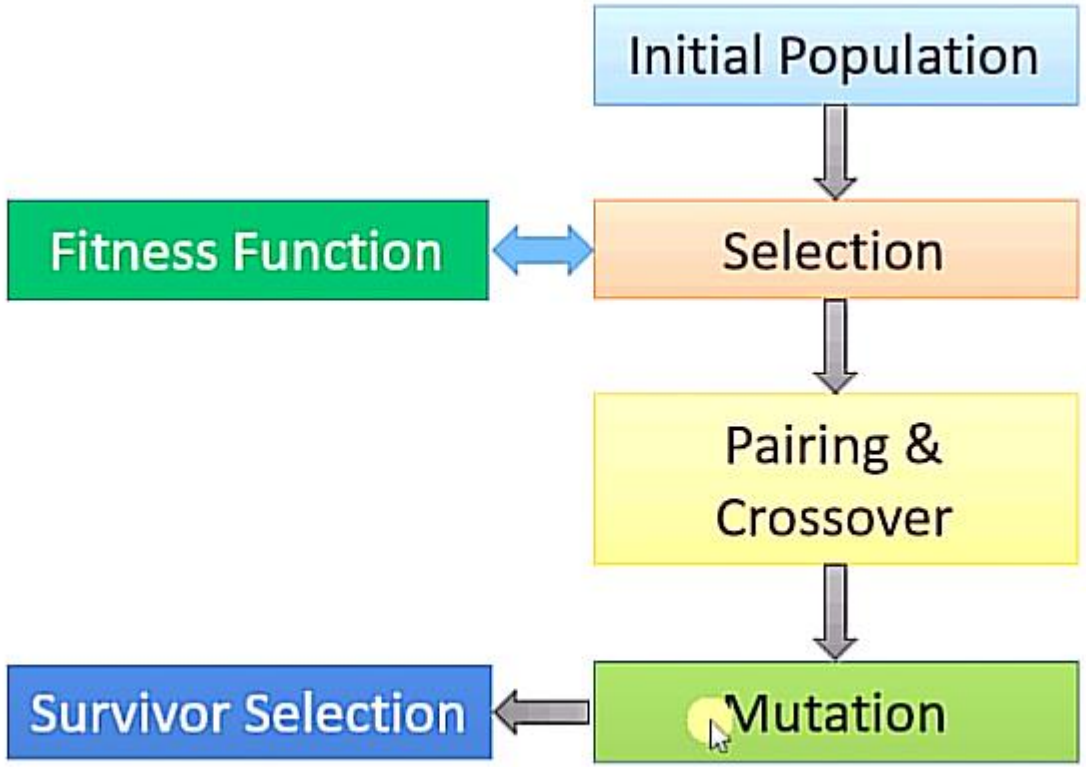
TECHNIQUES FOR MUTATION:

- *Bit Flip Mutation* : Select one or more random bits and flip them.

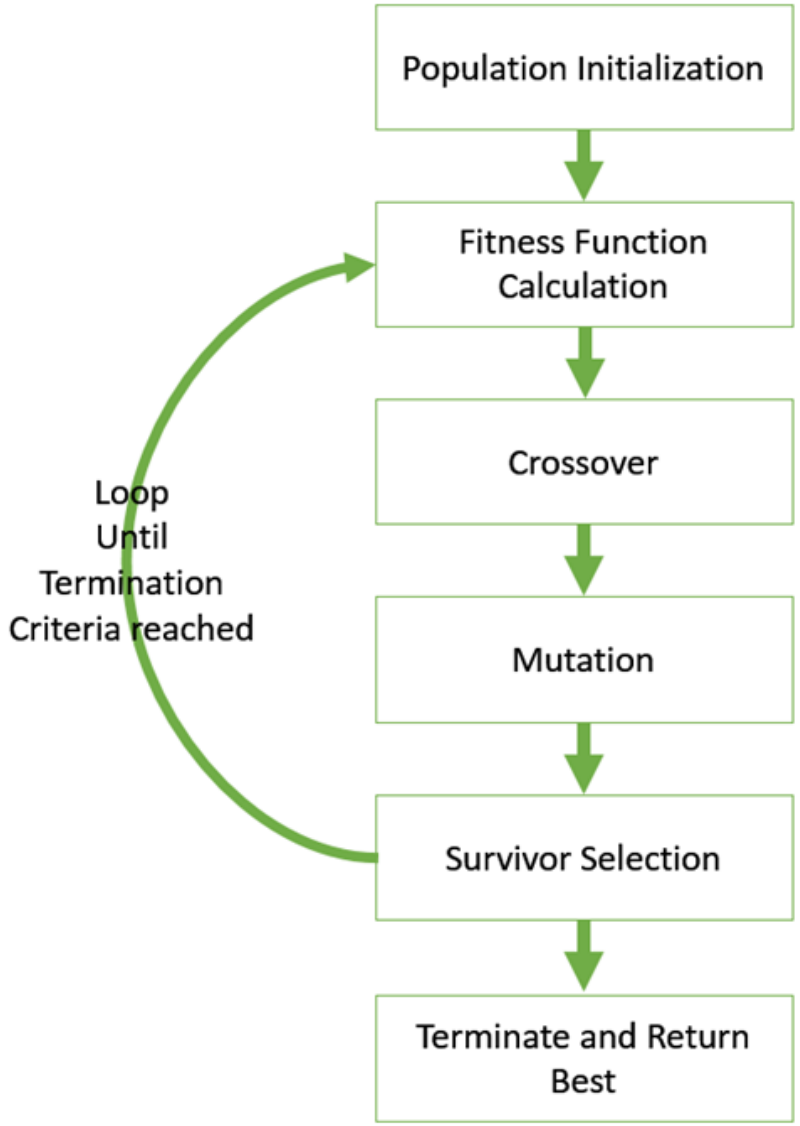


- *Swap Mutation*

SURVIVAL SELECTOR AFTER MUTATION:



COMPLETE FLOW OF GA



ADVANTAGES:

- Is faster and more efficient as compared to the traditional methods.
- Provides a list of “good” solutions and not just a single solution.
- Always gets an answer to the problem, which gets better over the time.
- Useful when the search space is very large and there are a large number of parameters involved.

LIMITATIONS:

- Computationally expensive as Fitness value is calculated repeatedly.
- Not suited for all problems, especially problems which are simple and for which derivative information is available.
- GA may not converge to the optimal solution, if not implemented properly.

ALGORITHM:

1) Randomly initialize populations p
2) Determine fitness of population
3) Until convergence repeat:
a) Select parents from population
b) Crossover and generate new population
c) Perform mutation on new population
d) Calculate fitness for new population

Python code

```
import random

# Number of individuals in each generation
POPULATION_SIZE = 100

# Valid genes
GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP
QRSTUVWXYZ 1234567890, .-;:_!"#%&/()=?@${[]}'''

# Target string to be generated
TARGET = "Pakistan Zindabad"

class Individual(object):
  '''
  Class representing individual in population
  '''
  def __init__(self, chromosome):
    self.chromosome = chromosome
    self.fitness = self.cal_fitness()

  @classmethod
  def mutated_genes(self):
    '''
    create random genes for mutation
    '''
    global GENES
    gene = random.choice(GENES)
    return gene


  @classmethod
  def create_gnome(self):
    '''
    create chromosome or string of genes
    '''
    global TARGET
    gnome_len = len(TARGET)
    gnome= [self.mutated_genes() for _ in range(gnome_len)]
    return gnome


  def mate(self, par2):
    '''
    Perform mating and produce new offspring
    '''

    # chromosome for offspring
    child_chromosome = []
    for gp1, gp2 in zip(self.chromosome, par2.chromosome):

      # random probability
      prob = random.random()
```

```
      # if prob is less than 0.45, insert gene
      # from parent 1
      if prob < 0.45:
        child_chromosome.append(gp1)

      # if prob is between 0.45 and 0.90, insert
      # gene from parent 2
      elif prob < 0.90:
        child_chromosome.append(gp2)

      # otherwise insert random gene(mutate),
      # for maintaining diversity
      else:
        child_chromosome.append(self.mutated_genes())

    # create new Individual(offspring) using
    # generated chromosome for offspring
    return Individual(child_chromosome)

  def cal_fitness(self):
    '''
    Calculate fitness score, it is the number of
    characters in string which differ from target
    string.
    '''
    global TARGET
    fitness = 0
    for gs, gt in zip(self.chromosome, TARGET):
      if gs != gt: fitness+= 1
    return fitness

# Driver code
def main():
  global POPULATION_SIZE

  #current generation
  generation = 1

  found = False
  population = []

  # create initial population
  for _ in range(POPULATION_SIZE):
        gnome = Individual.create_gnome()
        population.append(Individual(gnome))

  while not found:

    # sort the population in increasing order of fitness score
    population = sorted(population, key = lambda x:x.fitness)
```

```
  while not found:

    # sort the population in increasing order of fitness score
    population = sorted(population, key = lambda x:x.fitness)

    # if the individual having lowest fitness score ie.
    # 0 then we know that we have reached to the target
    # and break the loop
    if population[0].fitness <= 0:
      found = True
      break

    # Otherwise generate new offsprings for new generation
    new_generation = []

    # Perform Elitism, that mean 50% of fittest population
    # goes to the next generation
    s = int((50*POPULATION_SIZE)/100)

    new_generation.extend(population[:s])



    for _ in range(s):
      parent1 = random.choice(population[:50])
      parent2 = random.choice(population[:50])
      child = parent1.mate(parent2)
      new_generation.append(child)

    population = new_generation

    print("Generation: {}\tString: {}\tFitness: {}".\
      format(generation,
      "".join(population[0].chromosome),
      population[0].fitness))

    generation += 1


  print("Generation: {}\tString: {}\tFitness: {}".\
    format(generation,
    "".join(population[0].chromosome),
    population[0].fitness))

if __name__ == '__main__':
  main()
```

POPULATION LIST:

```
# create initial population
for _ in range(POPULATION_SIZE):
      gnome = Individual.create_gnome()
      population.append(Individual(gnome))

while not found:

  # sort the population in increasing order of fitness score
  population = sorted(population, key = lambda x:x.fitness)
```

```
population = [
Individual(['r','a','n','d','o','m','s','t','r','i','n','g']),
Individual(['a','b','c','d','e','f','g','h','i','j']),
Individual(['z','y','x','w','v','u','t','s','r','q']),
...
Individual(['p','o','n','m','l','k','j','i','h','g'])
]
```

SORTING POPULATION BASED ON FITNESS:

```
    # sort the population in increasing order of fitness score
    population = sorted(population, key = lambda x:x.fitness)
```

```
def cal_fitness(self):
    '''
    Calculate fitness score, it is the number of
    characters in string which differ from target
    string.
    '''
    global TARGET
    fitness = 0
    for gs, gt in zip(self.chromosome, TARGET):
        if gs != gt: fitness+= 1
    return fitness
```

population = [
Individual(fitness=5),
Individual(fitness=3),
Individual(fitness=1),
Individual(fitness=4),
Individual(fitness=2)
]

After sorting:

population = [
Individual(fitness=1),
Individual(fitness=2),
Individual(fitness=3),
Individual(fitness=4),
Individual(fitness=5)
]

CONVERGENCE CONDITION(TERMINATION CONDITION)

```
#current generation
generation = 1

found = False
population = []
```

Initially this found flag is false

```
# if the individual having lowest fitness score ie.
# 0 then we know that we have reached to the target
# and break the loop
if population[0].fitness <= 0:
  found = True
  break
```

After we found potential solution, it becomes true and loop breaks

HOW THIS LOOPS WORK

```
# Perform Elitism, that mean 50% of fittest population
# goes to the next generation
s = int((50*POPULATION_SIZE)/100)

new_generation.extend(population[:s])



for _ in range(s):
  parent1 = random.choice(population[:50])
  parent2 = random.choice(population[:50])
  child = parent1.mate(parent2)
  new_generation.append(child)

population = new_generation
```

50 % of the fittest population appended to the next generation

Other 50 % of the population generated using crossover among parents and the resulted offsprings appended to the next generation

PRINTING GENERATIONS:

```
    print("Generation: {}\tString: {}\tFitness: {}".\
      format(generation,
      "".join(population[0].chromosome),
      population[0].fitness))

    generation += 1


  print("Generation: {}\tString: {}\tFitness: {}".\
    format(generation,
    "".join(population[0].chromosome),
    population[0].fitness))

if __name__ == '__main__':
  main()
```

The first printing statement prints the generation based on its fitness value & print all the generations as it is present with in the loop

The second printing statement prints the final potential solution as loop breaks just after the flag become true and break the loop that's why it is out of the while loop

OUTPUT:

```
Generation: 1   String: A@slKdw# 01Z!#,}M       Fitness: 16
Generation: 2   String: A@slKdw# 01Z!#,}M       Fitness: 16
Generation: 3   String: _n,y
{39 ZX&:}B_#    Fitness: 15
Generation: 4   String: dtQi9,=]?Z KdYR[F       Fitness: 14
Generation: 5   String: dtQi9,=]?Z KdYR[F       Fitness: 14
Generation: 6   String: ,n?y;ra$ Zk3]ThaM       Fitness: 13
Generation: 7   String: ,n?y;ra$ Zk3]ThaM       Fitness: 13
Generation: 8   String: ,n?y;ra$ Zk3]ThaM       Fitness: 13
Generation: 9   String: $UkXhea? Z0.dTQ[Q       Fitness: 12
Generation: 10  String: $UkXhea? Z0.dTQ[Q       Fitness: 12
Generation: 11  String: u7kv${av Z8ndn%aw       Fitness: 10
Generation: 12  String: u7kv${av Z8ndn%aw       Fitness: 10
Generation: 13  String: u7kv${av Z8ndn%aw       Fitness: 10
Generation: 14  String: Pnki9wa   i3dQhax       Fitness: 9
Generation: 15  String: Pnki9wa   i3dQhax       Fitness: 9
Generation: 16  String: Pnki9wa   i3dQhax       Fitness: 9
Generation: 17  String: Pnki9wa   i3dQhax       Fitness: 9
Generation: 18  String: P
ki9Da? ZindQPax Fitness: 7
Generation: 19  String: P
ki9Da? ZindQPax Fitness: 7
Generation: 20  String: P
ki9Da? ZindQPax Fitness: 7
Generation: 21  String: P
ki9Da? ZindQPax Fitness: 7
```

```
[ ]  Fitness: 2
Generation: 97  String: Pakista  Zindaba
        Fitness: 2
Generation: 98  String: Pakista  Zindaba
        Fitness: 2
Generation: 99  String: Pakista  Zindaba
        Fitness: 2
Generation: 100 String: Pakista  Zindaba
        Fitness: 2
Generation: 101 String: Pakista  Zindaba
        Fitness: 2
Generation: 102 String: Pakista  Zindaba
        Fitness: 2
Generation: 103 String: Pakista  Zindaba
        Fitness: 2
Generation: 104 String: Pakista  Zindaba
        Fitness: 2
Generation: 105 String: Pakista  Zindaba
        Fitness: 2
Generation: 106 String: Pakista  Zindaba
        Fitness: 2
Generation: 107 String: Pakista  Zindaba
        Fitness: 2
Generation: 108 String: Pakista  Zindabad       Fitness: 1
Generation: 109 String: Pakista  Zindabad       Fitness: 1
Generation: 110 String: Pakista  Zindabad       Fitness: 1
Generation: 111 String: Pakista  Zindabad       Fitness: 1
Generation: 112 String: Pakista  Zindabad       Fitness: 1
Generation: 113 String: Pakista  Zindabad       Fitness: 1
Generation: 114 String: Pakista  Zindabad       Fitness: 1
Generation: 115 String: Pakista  Zindabad       Fitness: 1
Generation: 116 String: Pakista  Zindabad       Fitness: 1
Generation: 117 String: Pakista  Zindabad       Fitness: 1
Generation: 118 String: Pakista  Zindabad       Fitness: 1
Generation: 119 String: Pakista  Zindabad       Fitness: 1
Generation: 120 String: Pakista  Zindabad       Fitness: 1
Generation: 121 String: Pakista  Zindabad       Fitness: 1
Generation: 122 String: Pakista  Zindabad       Fitness: 1
Generation: 123 String: Pakista  Zindabad       Fitness: 1
Generation: 124 String: Pakista  Zindabad       Fitness: 1
Generation: 125 String: Pakista  Zindabad       Fitness: 1
Generation: 126 String: Pakista  Zindabad       Fitness: 1
Generation: 127 String: Pakista  Zindabad       Fitness: 1
Generation: 128 String: Pakista  Zindabad       Fitness: 1
Generation: 129 String: Pakista  Zindabad       Fitness: 1
Generation: 130 String: Pakista  Zindabad       Fitness: 1
Generation: 131 String: Pakistan Zindabad       Fitness: 0
```

TASK1:

- Using a genetic Algorithm Create a GUI based Guess a password given the number of correct letters in the guess. Build a mutation Engine