

Lab 1 – Genetic Algorithms:

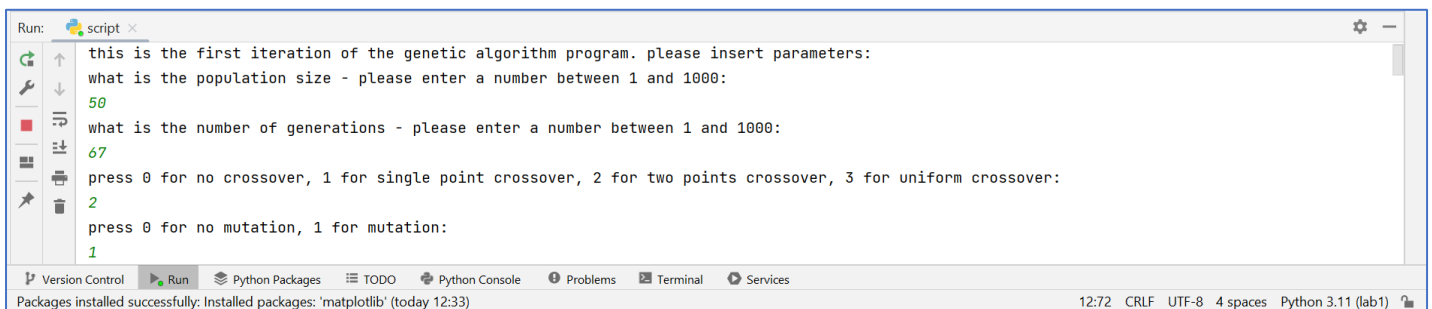
Submitters: Inbar Lev Tov 316327246, Gal Gibly 315093435

Note that the assignment was done with python version 3.11.0

1A:

For the first iteration of the algorithm, we would represent the target as a string. We would generate our population of genes that might one day reach the same level of perfection as our target. We would collect parameters for the algorithm, such as:

- Population size for each generation.
- Number of generations to be made.
- Type of crossover when we reproduce a child from 2 parents.
- Whether or not we want a mutation.



The screenshot shows a terminal window titled 'Run: script'. It contains the following text: 'this is the first iteration of the genetic algorithm program. please insert parameters:'. Then it asks 'what is the population size - please enter a number between 1 and 1000:' with the input '50'. Next, it asks 'what is the number of generations - please enter a number between 1 and 1000:' with the input '67'. Then it asks 'press 0 for no crossover, 1 for single point crossover, 2 for two points crossover, 3 for uniform crossover:' with the input '2'. Finally, it asks 'press 0 for no mutation, 1 for mutation:' with the input '1'. The bottom status bar shows 'Python 3.11 (lab1)'.

In case of any input mistakes, we have default values: 100 generations, 100 genes per generation, single point crossover and yes to mutation. Once the algorithm receives everything it needs, we would generate all 100 generations at once, and include calculations such as:

- Fitness heuristic.
- Mean, variance and standard deviation.
- Clock ticks time and elapsed time.
- Histograms.
- Crossover operators.
- Mutation.
- Bull's eye heuristic.

Fitness Heuristic:

- We compare every gene in every generation to our target.
- For every allele found in the **right place** – we get a point.
- The genes with the highest fitness score are later considered for the "elite team" – the next generation would accept them as they are, as they have the highest chance to get us closer to our target.

```
# returns a fitness score for a single person
def fitness(individual):
    score = 0
    for i in range(len(individual)):
        if individual[i] == target[i]:
            score += 1
    return score
```

Mean, Variance and Standard Deviation:

- We calculate mean, variance and standard deviation from looking at all the fitness scores of each generation.
- Those calculations helps us see how parameters such as crossover type, number of generations, and mutation affect each generation.
- For example, we would later compare and see that generations tend to converge to a local maximum rather than show more diversity when you have **no mutation**. We can see it immediately by the mean of each group – after a certain point, they all become the same number.

```
# return mean, variance and std of fitness
def meanVarianceStd(fitness):
    length = len(fitness)
    mean = sum(fitness) / length
    variance = sum(pow(i - mean, 2) for i in fitness) / length
    std = pow(variance, 0.5)
    return mean, variance, std
```

Clock Ticks & Elapsed Time:

- Elapsed = the time it took for the entire genetic algorithm to run from start to finish.
- When we started the process of generating the population, we placed a counter that would count that.
- Clock ticks = number of atomic operations performed in the genetic algorithm code.
- Since every generation is assembled differently (different parents, fitness scores and elite groups), we measured that time for each generation separately.
- Local minimum in the example below is 0.0009582042694091797 seconds.
- Global minimum is in nanoseconds – we show 0.0 seconds.
- Those counters are part of the "genetic algorithm" function – and have no function of their own.

Crossover Operators:

- If we choose not to use crossover operators, the next generation is in fact a copy of the original generation – simply passed on as it was.
- Single point: we choose a single point where we switch between parent A to parent B.
- Two point: we start with parent A until point1. We then switch to parent B until point2. We then come back to parent A until the end of the child's creation process.
- Uniform: at each point we randomly choose whether to take parent A or B.

```
# crossover according to type: single point, two point, uniform
def crossover(type, p1, p2, numberOf6):
    if type == 1:
        # 1 point for separating parent 1 and 2
        point = random.randint(1, numberOf6 - 1)
        child = p1[0: point] + p2[point:]
    elif type == 2:
        # 2 points separating: parent 1 until point1, then parent 2, then parent 1 again
        point1 = random.randint(1, numberOf6 - 2)
        point2 = random.randint(point1 + 1, numberOf6 - 1)
        child = p1[0: point1] + p2[point1: point2] + p1[point2:]
    elif type == 3:
        # at each point, we choose randomly between parent1 and parent2
        child = [p1[i] if random.random() < 0.5 else p2[i] for i in range(numberOf6)]
    return child
```

Mutation:

- We randomly choose a point within the gene.
- We randomly choose a char that would function as the new allele.
- We replace the original allele with the new one at the chosen point.

```
# mutation
def mutation(individual, length):
    point = random.randint(0, length - 1)
    newC = chr(random.randint(32, 126))
    individual[point] = newC
    return individual
```

Bull's Eye Heuristic:

- We compare every gene in every generation to our target.
- For every allele found in our target – we get a point.
- For every allele found in the **right place** – we get an extra point.

```
# gives a point if the word has the letter, and an extra point if the letter is in the right place
def bullsEye(individual):
    score = 0
    for i in range(len(individual)):
        if target[i] in individual:
            score += 1
        if individual[i] == target[i]:
            score += 1
    return score
```

Printing Results & Fitness Histograms:

- After all of this is done, we print the details for each generation.
- For example: 100 generation, 100 genes per generation, two points crossover and yes to mutation.
- Here are the early generations:

```
generations 0:
mean 0.14
std 0.3469870314579495
clock ticks 0.001960277557373047 seconds
```

```
generations 1:
mean 0.14
std 0.3469870314579495
clock ticks 0.0020210742950439453 seconds
```

```
generations 2:
mean 0.2
std 0.3999999999999993
clock ticks 0.0011661052703857422 seconds
```

```
generations 3:
mean 0.35
std 0.536190264738179
clock ticks 0.0008406639099121094 seconds
```

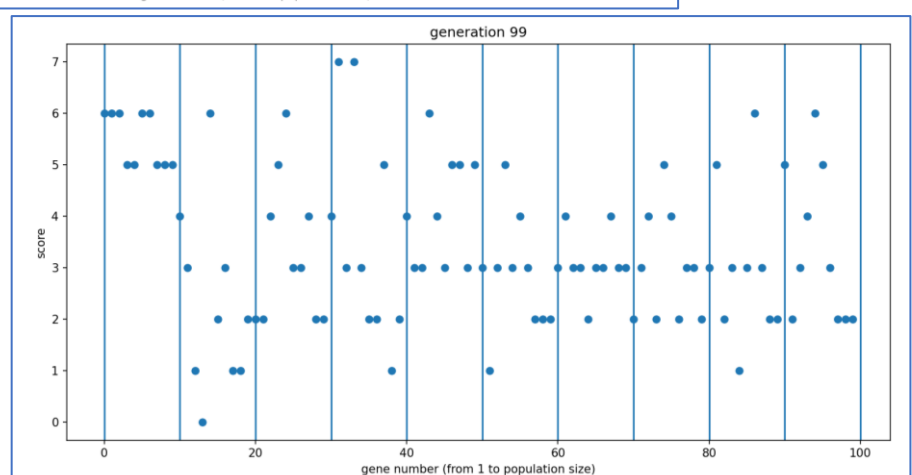
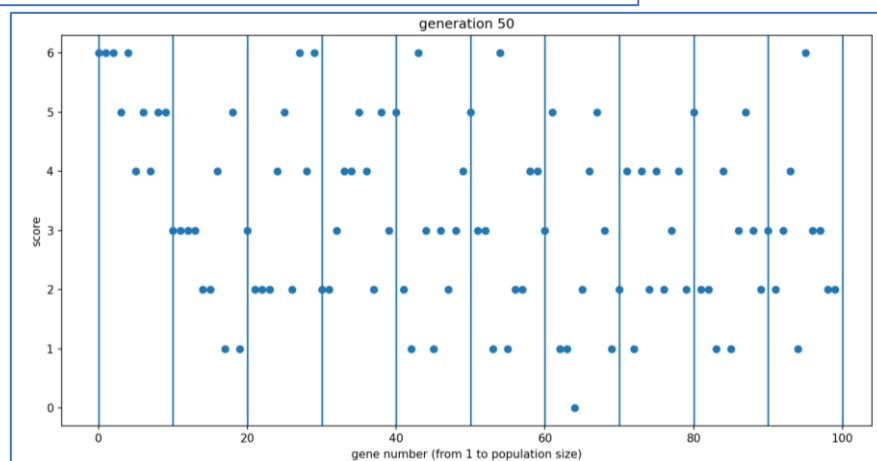
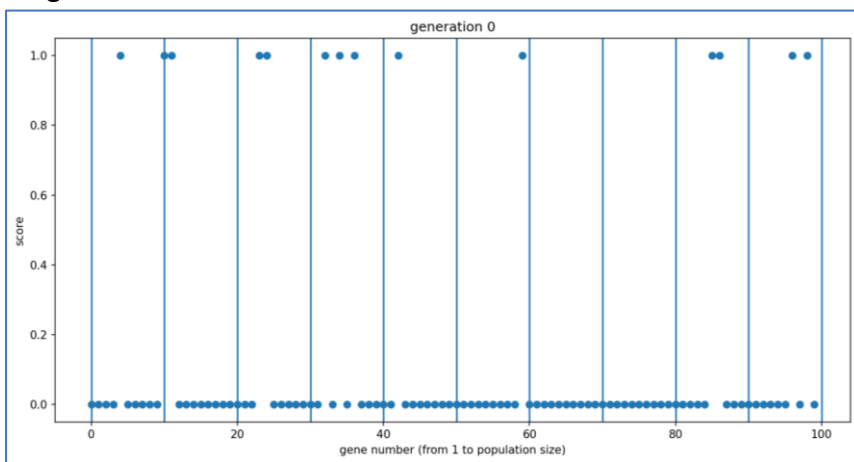
- And the last generations. Elapsed time is only printed once, at the end.

```
generations 97:
mean 3.74
std 1.6710475756243448
clock ticks 0.0 seconds
```

```
generations 98:
mean 3.57
std 1.5182555779578093
clock ticks 0.0010006427764892578 seconds
```

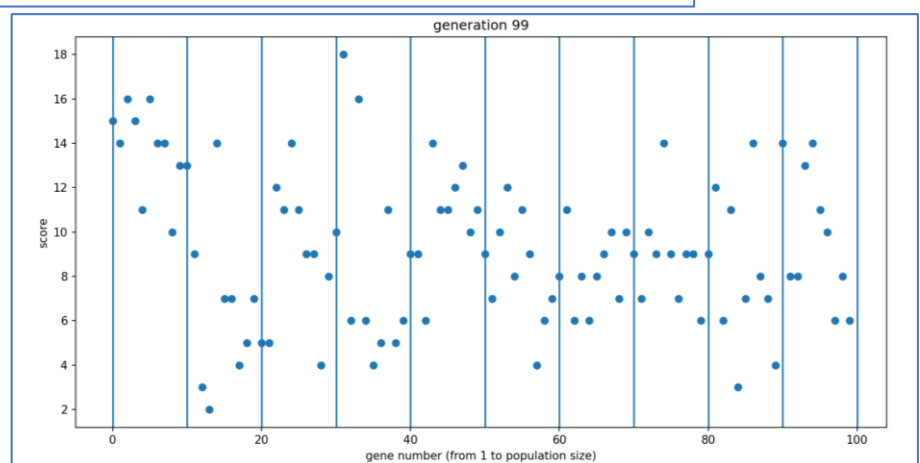
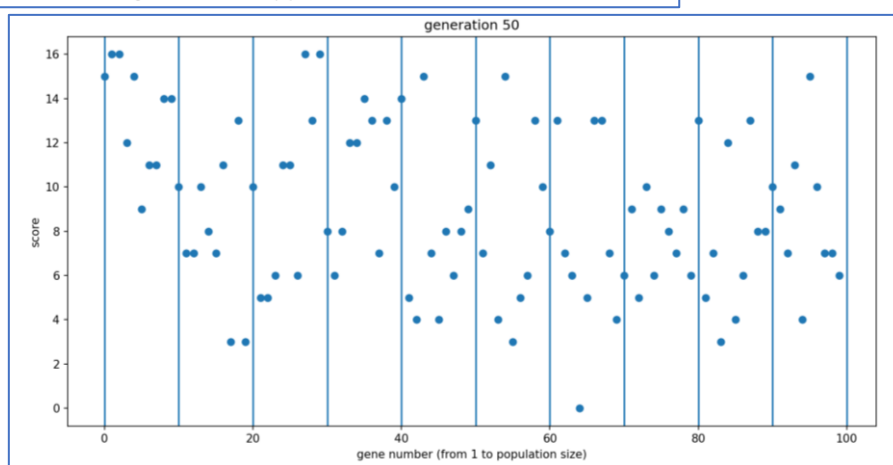
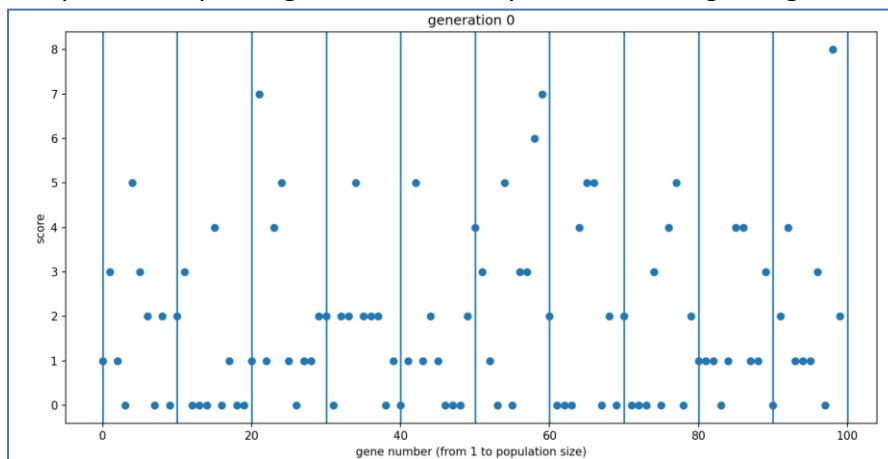
```
generations 99:
mean 3.41
std 1.5303267624922468
clock ticks 0.0009958744049072266 seconds
elapsed time 0.07165949999762233 seconds
```

- As you can see, while the earliest generations had relatively low fitness score (which means the mean and std are low), the last generations show massive improvement in that regard.
- It means that the algorithm is working – our "elite team" truly manages to find the best parents to pass on as children for the next generation, and a combination of crossover operators and mutation manages to produce improved and "stronger" children. They have a better chance to survive, and a better chance to reach the same level of perfection as our target.
- The clock ticking time is **not** rising monotonously. That's because we measured it separately for each generation, which means that the only thing affecting that time is the "complexity" of each generation.
- After the results are printed, you can also enter the letter 'A' (**caps lock**) and see 3 fitness histograms: beginning, middle, last.
- Here we had 100 generations – therefore we received histograms of generations 0, 50 and 99.
- While generation 0 was the worst (86 genes with no match), by the time we reach generation 50, our histogram is way more diverse.
- Generation 99 is the last product of elite teams, crossover operators and mutation – and has the widest range of results.

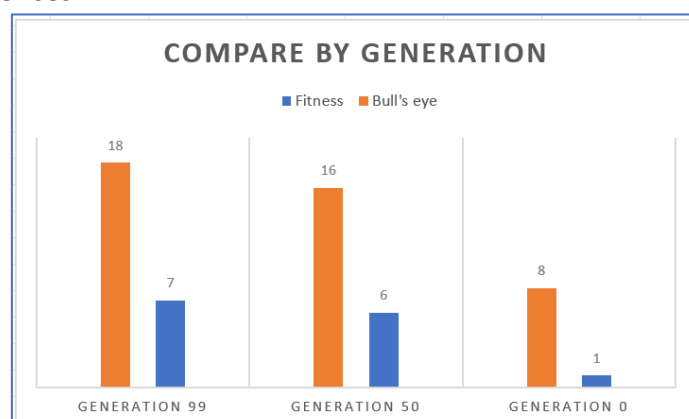


Fitness Heuristic VS. Bull's Eye Heuristic:

- After presenting the fitness histograms, we enter the letter '**B**' (**caps lock**) and see the 3 complementary histograms for bull's eye heuristic: beginning, middle, last.



- If we compare between the fitness and bull's eye heuristics in terms of maximum range of results, we can see significant differences:



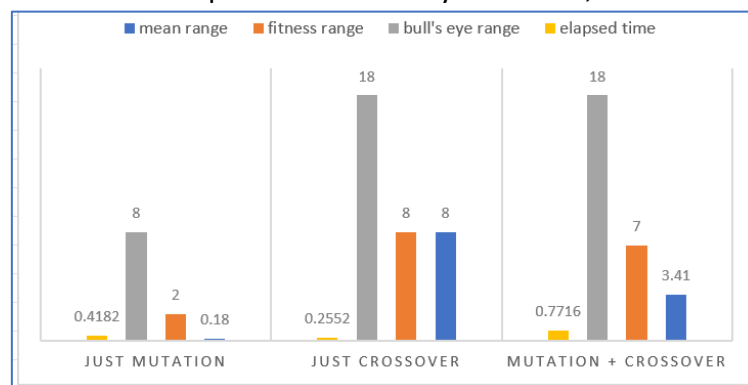
- The reason for those differences is the fact that, while fitness only measures perfection (right allele at the right place), bull's eye measures the ability to survive if you have the allele in the first place.
- You can also see it in the real world: lots of people aren't perfect, but most of us have a combination of strong alleles that can help us survive and live a pretty good life.
- Another thing to consider is the fact that we added **mutation** to the mix. Even after the crossover process is done, we can still change random alleles in the string and increase the chance of the child to be more fitting – regardless of its parents. Those changes makes the bull's eye heuristic grow.
- That is why the bull's eye heuristic is an **improved** version of the fitness heuristic, and has a much bigger range – we allow more genes each time the chance to compete for a spot at the next generation even though they're not perfect.

Exploration VS. Exploitation:

- We defined the crossover and mutation functions to be considered as **exploration**. The reason for that is that we search for a closer match to our target by changing the data to be more diverse. Each generation produces completely different results with those functions – and the search becomes more wide range.
- We defined the calculations functions to be considered as **exploitation**. The reason for that is that we calculate the best result each time on the existing data without changing it, and update accordingly. Functions such as fitness / bull's eye heuristic, mean, variance and std fits that description.

Different Configurations:

- We have already seen an example of crossover and mutation together.
- We would compare it to 2 other examples: one with only crossover, one with only mutation.



Conclusions:

- Genetic drift effect: a side effect that happens only in "**just crossover**" mode.
 - After a certain amount of time (here it was generation 45 out of 100), the system reaches an absorption state – all genes becomes identical.
 - That means the variance (diversity) of population members decreases over every generation, until it reaches global minimum of 0 (no diversity at all).
 - This effect is pronounced in the fitness heuristic – it converges to 8 and stays there forever – so much so that even the mean range converges to the same number (the effect is that strong).
- Deflated values: a side effect that happens only in "**just mutation**" mode.
 - Once you use crossover, the range of numbers of fitness and bull's eye heuristics stays pretty much the same (7 / 8 for fitness, 18 for bull's eye).
 - But in the just mutation mode those numbers deflate drastically.
 - That shows that the effect of mutation (changing one allele) is so much weaker than the effect of crossover ("changing" half or the entire string in one parent by combining it with the other parent).

- Conclusion: Only mutation doesn't allow much evolvement of the population – the original generation experience only minor changes in structure, and stays pretty much the same. You must have crossover operators!
- Elapsed time: notice the changes in elapsed time for the entire genetic algorithm.
 - While the combination of crossover and mutation receives maximum time, just crossover is twice as faster than just mutation.
 - That happens because mutation uses "random", and "random" is an expansive operator.
 - Since we measured it in terms of atomic operations, that difference is noticeable.
 - Crossover, however, is just combining two existing strings at a certain point, and therefore cheaper.

1B:

For the second iteration of the algorithm, we would look at the **N Queens problem**.

Representation:

- After receiving the population size and number of generations, we receive the number of queens.
- Every gene would hold a permutation of the queens – detailing their location on the board.
- Our "target" is to reach a perfect board – 0 collision between any 2 queens through rows, columns and diagonals on the board.

Fitness Heuristic:

- Since every gene is an "n queens" board, we would compare every pair of queens inside that board.
- For every collision (rows, columns and diagonals), you lose a point.
- That means that genes with a perfect board (has no collision) holds the **highest** score of 0.

```
# for every collision between queens, you lose a point
def fitness(self):
    score = 0

    for i in range(len(self.rows)):
        if i != len(self.rows) - 1:
            for j in range(i + 1, len(self.rows)):
                if self.rows[i] == self.rows[j]:
                    score += 1
                if self.cols[i] == self.cols[j]:
                    score += 1
                if abs(self.rows[i] - self.rows[j]) == abs(self.cols[i] - self.cols[j]):
                    score += 1

    score *= (-1)
    return score
```

Elitism & Aging:

- When we search for the elite team, we calculate a new fitness score that takes age into consideration.
- Because our old fitness ranges from 0 downward, every new candidate would receive a better score than the old one.
- After that it is the same elitism process as before: take the best 10% of candidates.

```
# elitism (choosing elite team) while fitting the aging criteria
def elitismAndAging(populationSize, fitnessesI, generation, numberOfGenerations):
    newFitness = [0.5 * fitnessesI[i] + 0.5 * (1 - (generation[i].age / numberOfGenerations)) for i in range(len(fitnessesI))]
    eliteSize = int(populationSize * 0.1)
    eliteIndices = sorted(range(populationSize), key=lambda i: newFitness[i], reverse=True)[:eliteSize]
    elites = [generation[i] for i in eliteIndices]
    return elites
```

Scaling:

- In the case of RWS and SUS – we must build a roulette.
- In order for us to do that we must also perform scaling on the existing fitness values – get all values closer together to reduce the influence of the highest value (the alpha male).
- We calculate the 5th percentile and 95th percentile and adjust the old fitness values accordingly.
- After that we subtract the fitness mean from every old fitness value – big values would go down, small values would go up – the gap between maximum and minimum decreases.

```
# scaling existing fitness
def scalingFitness(fitnessI, mean):
    lower = numpy.percentile(fitnessI, 5)
    upper = numpy.percentile(fitnessI, 95)
    for i in range(len(fitnessI)):
        if fitnessI[i] < lower:
            fitnessI[i] = lower
        if fitnessI[i] > upper:
            fitnessI[i] = upper

    newFitness = [fitnessI[i] - mean for i in range(len(fitnessI))]
    return newFitness
```

Roulette Building:

- We sort the new fitness array after scaling.
- We count the number of appearances for every value and divide by population size – that is the probability to get this value in a roulette spin.
- We are now ready to begin parent selection.

```
# create roulette for RWS, SUS parent selecting
def createRoulette(fitnessI):
    fitnessI.sort()
    roulette = []
    cell = fitnessI[0]
    counter = 1
    length = len(fitnessI)
    values = []

    for i in range(1, length):
        if int(fitnessI[i]) == int(cell):
            counter += 1
        else:
            values.append(cell)
            roulette.append(counter / length)
            cell = fitnessI[i]
            counter = 1
    values.append(cell)
    roulette.append(counter / length)

    return roulette, values
```

Parent Selection:

- RWS: spin the roulette and land on a random probability. Return the gene attached to it.
- SUS:
 - Spin the roulette the first time and land on a random probability. That would be the first parent.
 - Walk a measured step forward and land on another probability (we make sure they are different by removing the first probability from the roulette). That would be the second parent.
- Tournament with K competitors: we have already ranked them by fitness, and 'k' would always be 10% of the population. All that's left to do is to draw 2 random parents from that group.


```

if parents == 0:
    parent1 = RWS(roulette, newFitness, generation)
    parent2 = RWS(roulette, newFitness, generation)
elif parents == 1:
    parent1, parent2 = SUS(roulette, newFitness, generation)

elif parents == 2:
    parent1 = random.choice(kPool)
    parent2 = random.choice(kPool)

```

```

# RWS parent selection
def RWS(roulette, newFitness, generation):
    probability = random.choice(roulette)
    item = newFitness.index(probability)
    return generation[item]

# SUS parent selection
def SUS(roulette, newFitness, generation):
    r1 = roulette
    probability1 = random.choice(r1)
    if len(r1) != 1:
        r1.remove(probability1)
    probability2 = random.choice(r1)

    item1 = newFitness.index(probability1)
    item2 = newFitness.index(probability2)

    return generation[item1], generation[item2]

```

Crossover Operators:

- Unlike everything we've done so far, the crossover operators (as does the mutation) are now being activated on the permutation of queens rather than the gene itself.
- Which means that, in the event of only 1 queen, we simply combine the parents without doing any crossover.
- 2 queens or more:
 - PMX: choose two values and switch their spot in each of the parents.
 - CX: we created a cycle where: if it's an odd cycle number, take rows from parent 1 and columns from parent 2. If it's an even cycle number, take the opposite.

```

# PMX crossover operator
def PMX(parent1, parent2, permutation):
    child = Nqueens(permutation, 0, 0)
    child.rows = parent1.rows
    child.cols = parent2.cols

    if len(parent1.rows) == 1:
        return child
    else:
        i1 = random.choice(child.rows)
        i2 = random.choice(child.rows)
        index1 = child.rows.index(i1)
        index2 = child.rows.index(i2)
        a = child.rows[index1]
        child.rows[index1] = child.rows[index2]
        child.rows[index2] = a
        index1 = child.cols.index(i1)
        index2 = child.cols.index(i2)
        a = child.cols[index1]
        child.cols[index1] = child.cols[index2]
        child.cols[index2] = a

    return child

```

```

# CX crossover operator
def CX(parent1, parent2, permutation):
    child = Nqueens(permutation, 0, 0)
    if len(parent1.rows) == 1:
        return child
    else:
        temp = random.choice(permutation[0])
        if temp % 2 == 0:
            child.rows = parent1.rows
            child.cols = parent2.cols
        else:
            child.rows = parent2.rows
            child.cols = parent1.cols

    return child

```

Mutations:

- In the event of only 1 queen, there's no mutation to be made.
- 2 queens or more:
- Inversion:
 - We decided that the inner sequence to be reinserted is the last half of the original permutation.
 - We reverse it and insert it at the beginning of the permutation, effectively pushing the first half of the original permutation forward.
- Scramble:
 - In order to make the mutation effect more powerful, we decided to scramble the entire permutation – not just a single allele.
 - It creates more diversity, and sometimes even adds **new** permutation to the mix that were not there when generation 0 was born.

```
# inversion mutation
def inversion(self):
    if len(self.rows) == 1:
        return
    else:
        length = len(self.rows)
        point = int(length / 2)

        temp = [self.rows[i] for i in range(point, length)]
        temp.reverse()
        self.rows = temp + self.rows[0: point]
        temp = [self.cols[i] for i in range(point, length)]
        temp.reverse()
        self.cols = temp + self.cols[0: point]
```

```
# scramble mutation
def scramble(self):
    if len(self.rows) == 1:
        return
    else:
        length = len(self.rows)
        temp = [self.rows[i] for i in range(length)]
        for i in range(length):
            val = random.choice(temp)
            self.rows[i] = val
            self.cols[length - i - 1] = val
            temp.remove(val)
```

Selection Pressure:

- We call it the exploitation factor because it helps us push forward not just some average candidate but the best candidate.
- if we tend to propagate the genes that have the best fitness score forward to the next generation, we make sure that each generation has some "strong" players – helps to ensure survivability ratio.
- We have two ways to measure it:
 - Fitness variance, which we already measured before in our "mean, std" function.
 - Top average selection probability ratio: what's the ratio of above average genes in our generation (ratio of best fitting genes).
 - The higher the ratio, the better the generation becomes.
 - We measured it by counting above average candidates and dividing it by population size.

```
# top average ratio
def topAverageF(fitnessI, mean):
    counter = 0
    for i in fitnessI:
        if i > mean:
            counter += 1

    return counter / len(fitnessI)
```

Continue on the next page

Genetic Diversity:

- We call it the exploration factor because it helps us push forward not just some average candidates, but the most diverse candidate.
- Distance between genes:
 - If we tend to propagate forward the genes that have the most distance from everyone else, we basically pass forward the most unique gene (most unique sequence of queens).
 - Due to the combination of parent selection, crossover operator and mutation, that gene could take a pivotal part in shaping many different sequences of queens – some of them would even have a better chance of surviving and reaching optimal solution rather than their parents.
- Number of different permutations:
 - If our representation was a string (like in the first iteration of the algorithm) we would have counted the number of different alleles in every generation.
 - As it happens, our iteration is about queens – therefore we count the number of different permutations in each generation.
 - The generation that has the maximum number of permutations – is a generation we need to pay attention to. We can use that generation to create many more children – ensuring a much more diverse generation than the one before us.

```
# gene distance - measure the number of changes required to get from one permuto
def geneDistance(generation):
    length = len(generation)
    counter = 0

    for i in range(length):
        for j in range(i + 1, length):
            counter += differences(generation[i].rows, generation[j].rows)
            counter += differences(generation[i].cols, generation[j].cols)

    return counter

def differences(a, b):
    return sum(i != j for i, j in zip(a, b))
```

```
# number of different permutations in each generation
def countD(generation):
    list = []
    counter = 0

    for i in range(len(generation)):
        if generation[i].rows in list:
            continue
        else:
            list.append(generation[i].rows)
            counter += 1
        if generation[i].cols in list:
            continue
        else:
            list.append(generation[i].cols)
            counter += 1

    return counter
```

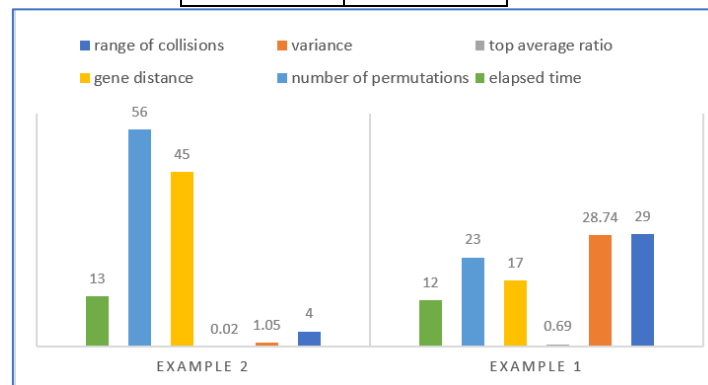
Printing Results & Comparisons:

- After all of this is done, we print the details for each generation.
- For example: 200 generations, 200 genes per generation, 10 queens for each board, RWS for parent selection, PMX for crossover and inversion for mutation.
- Here is a snapshot of one of the generations:

```
generations 124:
mean -6.025
std 2.6008412100703113
selection pressure - variance 6.764375
selection pressure - top average ratio 0.59
genetic diversity - distance between genes 348180
genetic diversity - number of permutations 339
clock ticks 0.05396294593811035 seconds
```

- We would compare it to a different example with the same amount of genes, generations and queen – but completely opposite parameters for everything else:

Example 2	Example 1
SUS	RWS
CX	PMX
scramble	inversion



Conclusions:

- As you can see in the graph, while example 1 has more collisions between queens than example 2, it has a much larger variance – which means that a lot of those genes **can** be considered as good candidates for future generations, because they have way less collisions than the average.
- You can also see a key difference between the results:
 - Example 1 puts much more emphasis on selection pressure. 69% of candidates in each generation are above average – which means that those candidates has the best fitness score, and can be considered as the alpha male. We send the best forward every time.
 - Example 2, on the other hand, puts much more emphasis on genetic diversity. The distance between genes (sequences of queens) is much bigger, as does the number of unique permutations in each generation.
 - It means that every generation is way more diverse than the last one, and that diversity ups our chances on landing on an optimal solution.
- As for elapsed time, although the difference seems to be minor (12 second vs. 13 seconds), we know it's not. We saw in clock ticks time that processing a generation takes mere nanoseconds – which means that the genetic diversity effect makes generation processing slower. You have more options, more possible optimal solutions to choose from – it takes more time.

Overall we can conclude that genetic diversity is more influential than selection pressure. We reach optimal solution much faster (in the sense that it appears in a much earlier generation in example 2 than it is in example 1) and manage to hold that solution for many generations hence.

Bin Packing Problem:

For the third and last iteration of the algorithm, we would look at the **Bin Packing problem**.

Most of the implementations here are the same as with n queens, the only difference is:

- Representation.
- Fitness heuristic.
- The addition of first fit heuristic.

Representation:

- Unlike the queens, here we have a permutation that dictates the order of insertion of items into bins.
- Since we have the same 120 items in each generation – that order is important.
- Different order of insertion brings very different results.

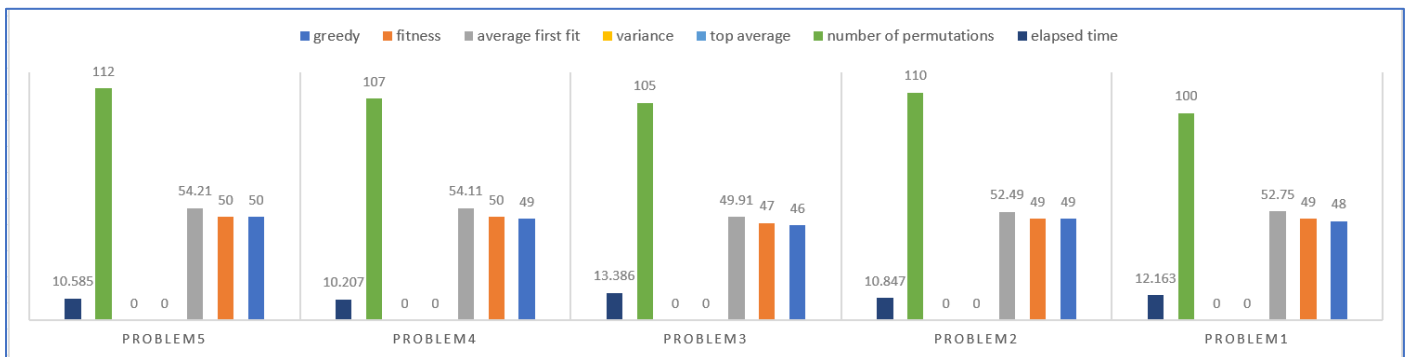
Heuristics:

- First Fit heuristic:
 - We have bins with capacity of 150.
 - For every item in the list, we scour the existing bins and ask – can this item fit here?
 - If the answer is yes, we insert the item in the bin. If the answer is no, we assign a new bin with the item as it's first fit.
- Fitness heuristic:
 - We used the "first fit decreasing" heuristic" in order to calculate fitness. It simply means that we sort the items in decreasing order (maximum is first) and then send it to first fit
 - We would later compare and see, that the effect of sorting is quite powerful – reduces bins and gets us closer to the greedy solution.

```
# first fit
def firstFit(self):
    remain = [self.capacity]
    sol = [[]]
    s = [self.orderOfInsertion[i] for i in range(self.numberOfItems)]

    for item in s:
        for j, free in enumerate(remain):
            if free >= item:
                remain[j] -= item
                sol[j].append(item)
                break
        else:
            sol.append([item])
            remain.append(self.capacity - item)
    return len(sol)
```

Comparisons Between All 5 Problems:



Conclusions:

- Our fitness heuristic is much better than first fit heuristic – always bring better results. In addition, notice that in problems 2 and 5 we even manage to get the greedy solution.
- Since our fitness is measured through **sorting** of the same 120 items, there is no selection pressure at all: All the genes in a single generation have the same fitness, which means there is no top average (all of them are equal to the mean) and they do not vary from one another (no variance).
- Elapsed time & genetic diversity: notice that in problems 2, 4 and 5 the elapsed time converges to 10 seconds. Also notice that those problems have the largest number of unique permutations. Here, unlike in queens, the genetic diversity allows us to reduce processing time rather than increase it.