Evolutionary Search - Part 2

November 16, 2024

1 Evolutionary Algorithms (Part 2)

In this chapter we take a closer look at some of the search operators and algorithmic choices that we took for granted so far. In particular, we reconsider selection, crossover, mutation, and the topology of the population itself.

[1]:
```
import matplotlib.pyplot as plt
import random
import math
import sys
from IPython.utils import io
from statistics import mean
```

We will start by considering the one max problem again, where a solution is a vector of length n, consisting of binary numbers.

[2]:
```
n = 20
```

Our representation is a simple list of bits.

[3]:
```
def get_random_solution():
    individual = [random.choice([0,1]) for _ in range(n)]
    return individual
```

The fitness function as well as variation operators are still the same:

[4]:
```
def get_fitness(solution):
    return sum(solution)
```

We start with the operators that we have used previously.

[5]:
```
def mutate(individual):
    P_mutate = 1/len(individual)
    copy = individual[:]
    for position in range(len(individual)):
        if random.random() < P_mutate:
            copy[position] = 1 - copy[position]
    return copy
```

[6]:
```
def singlepoint_crossover(parent1, parent2):
    pos = random.randint(0, len(parent1))
    offspring1 = parent1[:pos] + parent2[pos:]
    offspring2 = parent2[:pos] + parent1[pos:]
    return offspring1, offspring2
```

The first selection operator we considered was tournament selection:

[7]:
```
tournament_size = 3
def tournament_selection(population, replacement=False):
    if replacement:
        candidates = random.choices(population, k = tournament_size)
    else:
        candidates = random.sample(population, tournament_size)

    winner = max(candidates, key = lambda x: get_fitness(x))
    return winner
```

A general problem in evolutionary search is finding the right balance between exploration and exploitation. Using the wrong operators may lead to premature convergence, or the search may never converge at all at an optimum. As a first step towards understanding what is happening inside the population of a genetic algorithm, we will consider the *average fitness* of the population as well as the *diversity* within the population in addition to the best fitness value we already tracked in the past. For this, we first define the difference between two individuals as the hamming distance between the vector representations:

[8]:
```
def hamming_distance(individual1, individual2):
    return sum(c1 != c2 for c1, c2 in zip(individual1, individual2))
```

We can now calculate an overall population diversity as the sum of pairwise hamming distances for all pairs of individuals in the population.

[9]:
```
def pairwise_distance(population):
    distances = 0
    for i in range(len(population)-1):
        for j in range(i, len(population)):
           distances += hamming_distance(population[i], population[j])
    return distances
```

We need to set the parameters of our genetic algorithm first:

[10]:
```
population_size = 20
P_xover   = 0.7
max_steps = 1000
selection = tournament_selection
crossover = singlepoint_crossover

# These lists track values throughout the evolution
# so we can compare the behaviour of different operators
```

```
fitness_values = []
diversity_values = []
mean_fitness_values = []
```

In the lecture we discussed that there are many different variations of all the search operators involved in a genetic algorithm. We will now look at a couple of relevant search operators.

1.1 Survivor Selection

Most evolutionary algorithms use a fixed population size, so we need a way of going from (parents + offspring) to the next generation. - In age-based selection one usually produces as many offspring as there are parents, and then replaces the parents with the offspring. - In fitness-based selection, we rank the parents and the offspring, and take the best of all.

We saw both versions last time in the context of evolution strategies, but can also create versions of a genetic algorithm. In a generational genetic algorithm the offspring replaces the parents, while in a steady state genetic algorithm we apply fitness-based survivor selection.

Elitism is a special case, where the best individuals of the population always survive, while other means are used for the rest of the population. We implement a simple version of elitism by simply ranking the population by diversity and taking the top `elite_size` elements:

[11]:
```
elite_size = int(population_size * 0.05)

def elitism(population):
    population.sort(key=lambda k: get_fitness(k), reverse=True)
    return population[:elite_size]
```

Let's revisit the standard genetic algorithm with a generational selection model, integrated with elitism:

[12]:
```
def ga():
    population = [get_random_solution() for _ in range(population_size)]
    fitness_values.clear()

    # This could probably be written in a single line, but let's keep it␣
 ↪explicit
    best_fitness = -1
    for p in population:
        fitness = get_fitness(p)
        if fitness > best_fitness:
            best_fitness = fitness
        fitness_values.append(best_fitness)

    diversity_values.append(pairwise_distance(population))
    mean_fitness_values.append(mean([get_fitness(x) for x in population]))

    while len(fitness_values) < max_steps:
```

```
        new_population = elitism(population)
        while len(new_population) < len(population):
            parent1 = selection(population)
            parent2 = selection(population)

            if random.random() < P_xover:
                offspring1, offspring2 = crossover(parent1, parent2)
            else:
                offspring1, offspring2 = parent1[:], parent2[:]

            offspring1 = mutate(offspring1)
            offspring2 = mutate(offspring2)

            fitness1, fitness2 = get_fitness(offspring1),␣
↪get_fitness(offspring2)

            if fitness1 > best_fitness:
                best_fitness = fitness1
            fitness_values.append(best_fitness)
            if fitness2 > best_fitness:
                best_fitness = fitness2
            fitness_values.append(best_fitness)

            new_population += [offspring1, offspring2]

        population = new_population
        diversity_values.append(pairwise_distance(population))
        mean_fitness_values.append(mean([get_fitness(x) for x in population]))

    return max(population, key=lambda k: get_fitness(k))
```

The alternative was the steady state genetic algorithm, where we select two parents, derive their offspring, and then do fitness-based survivor selection:

[13]:
```
def steadystatega():
    population = [get_random_solution() for _ in range(population_size)]
    best_fitness = -1
    fitness_values.clear()

    for p in population:
        fitness = get_fitness(p)
        if fitness > best_fitness:
            best_fitness = fitness
            best_solution = p
        fitness_values.append(best_fitness)
    diversity_values.append(pairwise_distance(population))
    mean_fitness_values.append(mean([get_fitness(x) for x in population]))
```

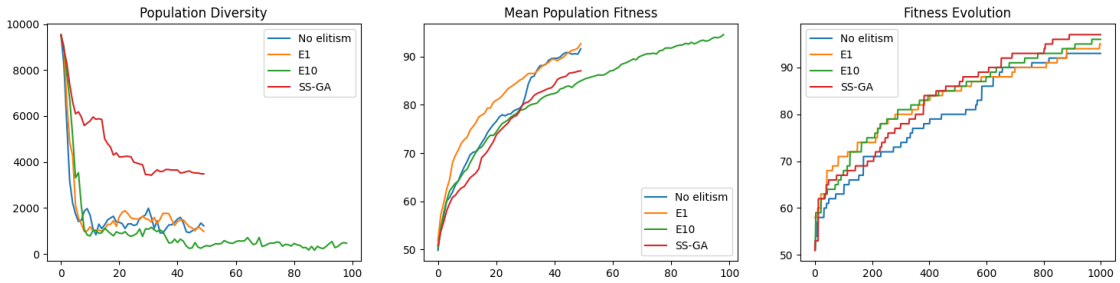
```
    while len(fitness_values) < max_steps:
        parent1 = selection(population)
        parent2 = selection(population)

        p1 = population.index(parent1)
        p2 = population.index(parent2)

        if random.random() < P_xover:
            offspring1, offspring2 = crossover(parent1, parent2)
        else:
            offspring1, offspring2 = parent1[:], parent2[:]

        offspring1 = mutate(offspring1)
        offspring2 = mutate(offspring2)

        best1, best2 = sorted([parent1, parent2, offspring1, offspring2],␣
↪key=lambda x: get_fitness(x), reverse=True)[:2]
        population[p1] = best1
        population[p2] = best2

        fitness1, fitness2 = get_fitness(best1), get_fitness(best2)

        if fitness1 > best_fitness:
            best_fitness = fitness1
        fitness_values.append(best_fitness)
        if fitness2 > best_fitness:
            best_fitness = fitness2
        fitness_values.append(best_fitness)

        # To make plots comparable with the generational GA
        if len(fitness_values) % population_size == 0:
            diversity_values.append(pairwise_distance(population))
            mean_fitness_values.append(mean([get_fitness(x) for x in␣
↪population]))

    return best_solution
```

Note that there are further possibilities for variation here: Instead of replacing the selected parents in the population, we could select other individuals (e.g., the worst individuals).

To consider the effects on diversity, let's compare diversity and fitness throughout one run each. We'll increase `n` to make the problem slightly more challenging.

[14]:
```
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 4))
n = 100
max_steps = 1000
population_size = 20
```

```
with io.capture_output() as captured:
    elite_size = 0
    fitness_values = []
    diversity_values = []
    mean_fitness_values = []
    ga()
    axes[0].plot(diversity_values, label=f"No elitism")
    axes[1].plot(mean_fitness_values, label=f"No elitism")
    axes[2].plot(fitness_values, label=f"No elitism")

    elite_size = 1
    fitness_values = []
    diversity_values = []
    mean_fitness_values = []
    ga()
    axes[0].plot(diversity_values, label=f"E1")
    axes[1].plot(mean_fitness_values, label=f"E1")
    axes[2].plot(fitness_values, label=f"E1")

    elite_size = 10
    fitness_values = []
    diversity_values = []
    mean_fitness_values = []
    ga()
    axes[0].plot(diversity_values, label=f"E10")
    axes[1].plot(mean_fitness_values, label=f"E10")
    axes[2].plot(fitness_values, label=f"E10")

    fitness_values = []
    diversity_values = []
    mean_fitness_values = []
    steadystatega()
    axes[0].plot(diversity_values, label=f"SS-GA")
    axes[1].plot(mean_fitness_values, label=f"SS-GA")
    axes[2].plot(fitness_values, label=f"SS-GA")

axes[0].set_title('Population Diversity')
axes[0].legend()
axes[1].set_title('Mean Population Fitness')
axes[1].legend()
axes[2].set_title('Fitness Evolution')
axes[2].legend()
plt.show()
```



As usual results may vary between runs, since these are randomised algorithms. However, a general trend we should see in the above plots is that the population diversity of the steady state GA reduces much slower than in a generational GA, and also the average fitness value in the population remains lower. The large elitism size of `10` means that the algorithm can run more generations with the same number of fitness evaluations, which is why it continues longer than the others in the first two plots (which shows generations rather than fitness evaluations). A small elitism size tends to generally lead to a better average fitness in this configuration. The best performing version will differ between runs.

1.2 Parent Selection

A major difference between the evolution strategies we considered initially and the canonical genetic algorithm we looked at afterwards is the parent selection strategy. In classical evolution strategies all μ (mu) parents are involved in recombination, and the survivor selection is what drives the selective pressure. In genetic algorithms, instead, the parent selection applies selective pressure.

We started off with tournament selection because it is the quickest to implement. A traditionally more common variant is fitness proportionate selection, where the probability of an individual to be selected is proportional to its fitness value. The selection thus first calculates the total fitness sum, and then probabilistically chooses an individual by sampling a number in the range between 0 and the total fitness sum. An important requirement is that the population is sorted by fitness values, starting with the best individual (largest fitness). This selection operator is also known as *roulette wheel selection*.

In our simple implementation, we create a list of tuples `fitness_list` that stores individuals with their fitness. Obviously, there's some redundant fitness calculations here; in practice one would cache fitness values.

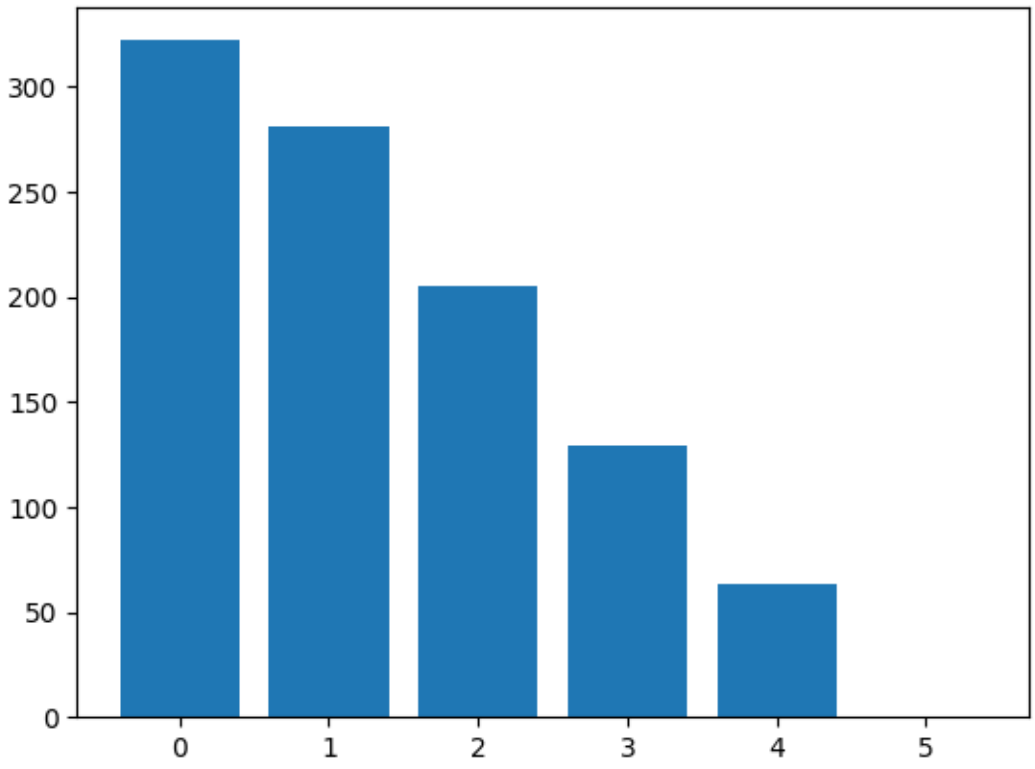
[15]:
```
def roulette_selection(population):
    fitness_sum = sum([get_fitness(x) for x in population])
    population.sort(key=lambda x: get_fitness(x), reverse=True)
    pick = random.uniform(0, fitness_sum)
    current = 0
    for x in population:
        current += get_fitness(x)
        if current > pick:
            return x
```

To evaluate this, let's create a simple example population for `n=5` with individuals with fitness 5, 4, 3, 2, 1, and 0:

[16]:
```
example_population = [ [1,1,1,1,1], [0,1,1,1,1], [0,0,1,1,1], [0,0,0,1,1],␣
↪[0,0,0,0,1], [0,0,0,0,0]  ]
```

Applying this selection operator will more likely select the best individual(s) (but may select worse individuals as well). We can do a simple experiment by sampling repeatedly from our `example_population` and looking at the resulting histogram:

[17]:
```
counts = {x: 0 for x in range(len(example_population))}

for i in range(1000):
    selected = roulette_selection(example_population)
    index = example_population.index(selected)
    counts[index] = counts[index] + 1

plt.bar(counts.keys(), counts.values())
plt.xticks(list(counts.keys()));
```

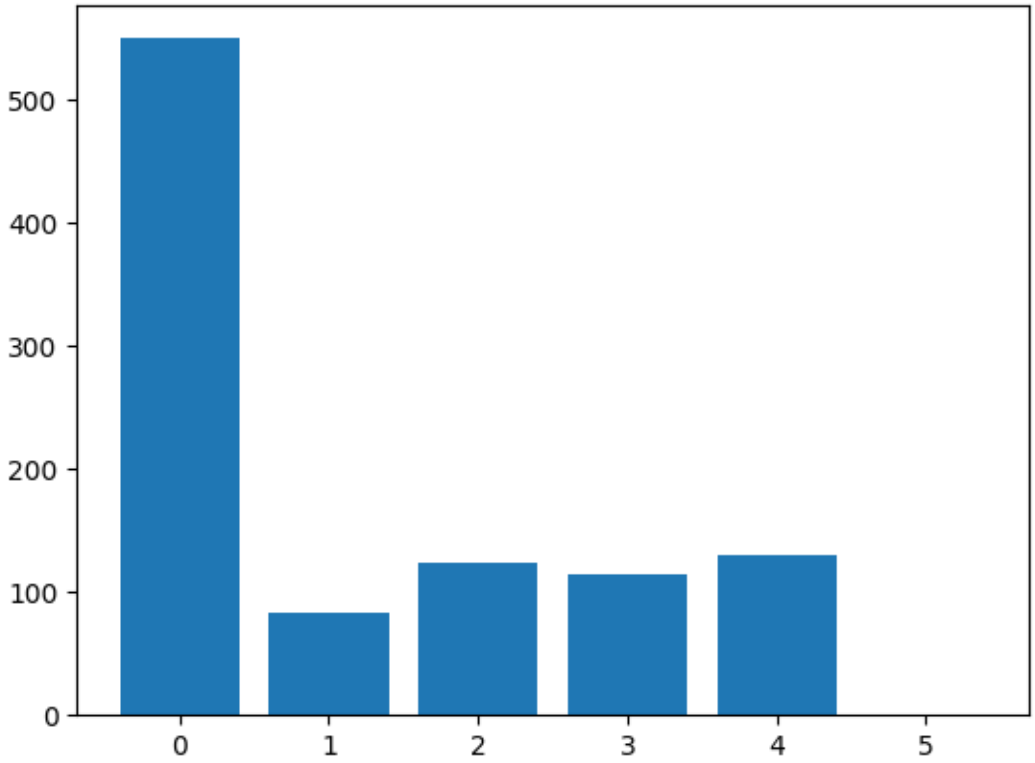


A problem with fitness proportionate selection is that individuals that have a much better fitness value will dominate the selection. For example, let's skew our example population:

[18]:
```
example_population = [ [1,1,1,1,1], [0,1,0,0,0], [1,0,0,0,0], [0,0,1,0,0],␣
↪[0,0,0,1,0],[0,0,0,0,0] ]
```

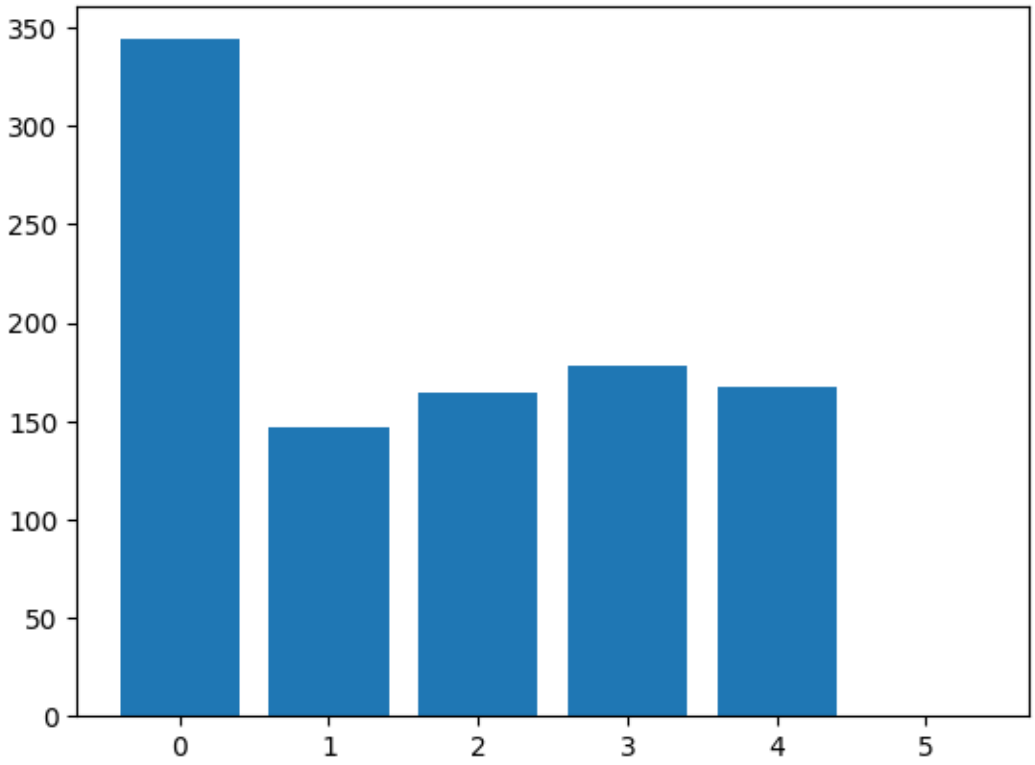
In this population, the first individual has fitness value 5, while the other individuals have fitness 1 and 0. The sum of fitness values is 9, and so the first individual has a probability of 56% of being selected:

[19]:
```
counts = {x: 0 for x in range(len(example_population))}

for i in range(1000):
    selected = roulette_selection(example_population)
    index = example_population.index(selected)
    counts[index] = counts[index] + 1

plt.bar(counts.keys(), counts.values())
plt.xticks(list(counts.keys()));
```



Tournament selection, which we implemented earlier, suffers less from this problem. In tournament selection, we can adjust the *selective pressure* by adjusting the tournament size. With our example population of size 6, a tournament size of 2 without replacement would imply a 33% chance of the best of the six individuals being selected:

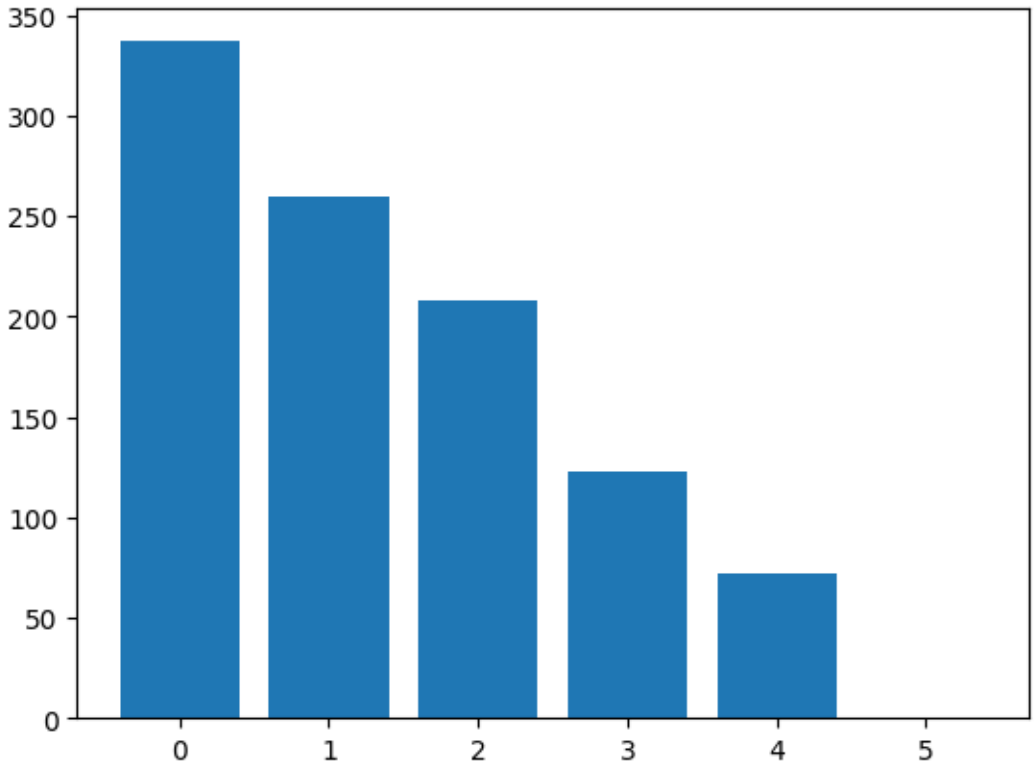
[20]:
```
tournament_size = 2

counts = {x: 0 for x in range(len(example_population))}

for i in range(1000):
    selected = tournament_selection(example_population)
    index = example_population.index(selected)
    counts[index] = counts[index] + 1

plt.bar(counts.keys(), counts.values())
plt.xticks(list(counts.keys()));
```



Let's also compare this to the population with a more equal spread:

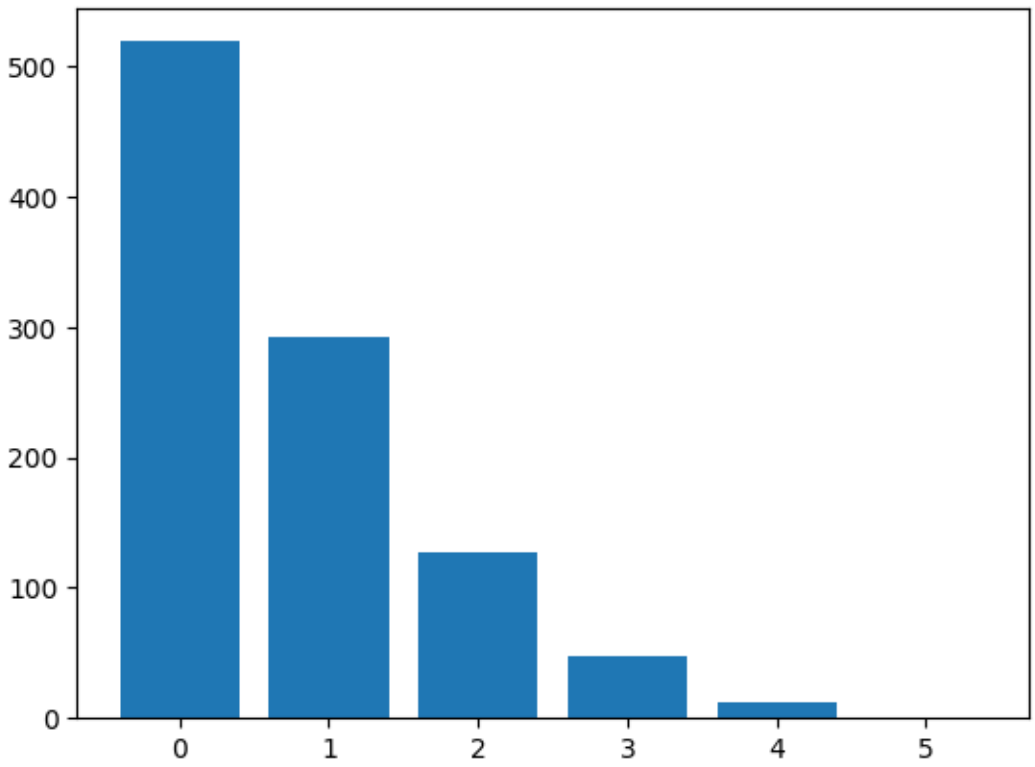
[21]:
```
example_population = [ [1,1,1,1,1], [0,1,1,1,1], [0,0,1,1,1], [0,0,0,1,1],␣
↪[0,0,0,0,1], [0,0,0,0,0]  ]

counts = {x: 0 for x in range(len(example_population))}

for i in range(1000):
    selected = tournament_selection(example_population)
    index = example_population.index(selected)
```

```
    counts[index] = counts[index] + 1

plt.bar(counts.keys(), counts.values())
plt.xticks(list(counts.keys()));
```



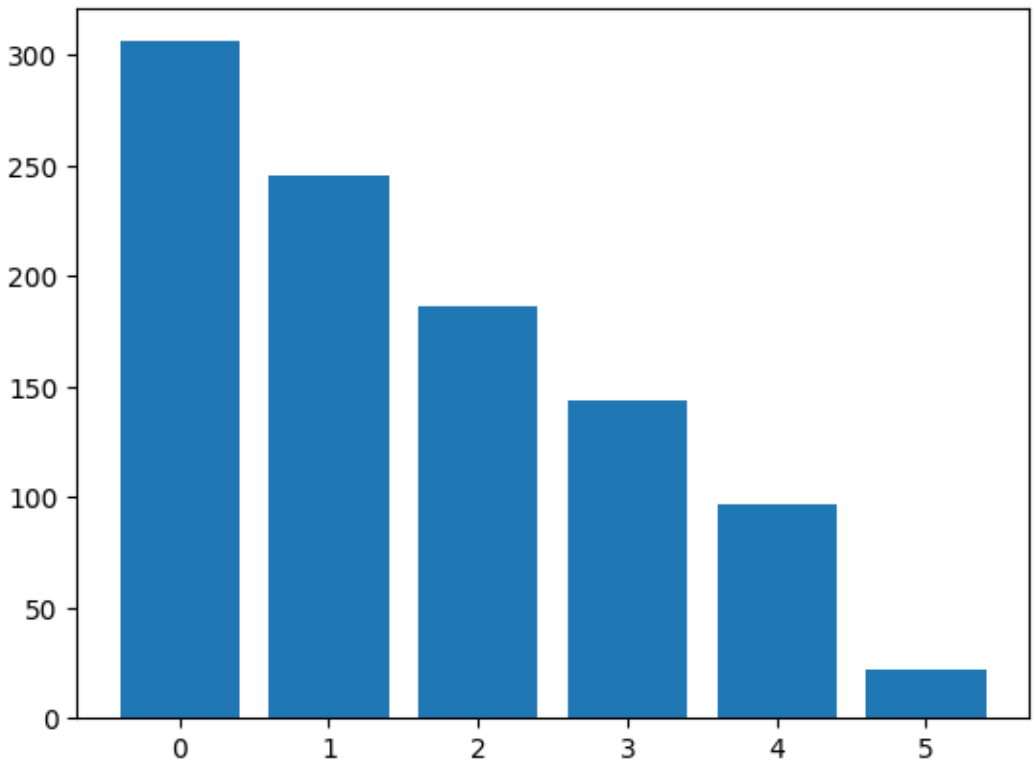
To see the effects of the tournament size on the selection, let's repeat this with a larger tournament size, which means higher selective pressure:

[22]:
```
tournament_size = 4

counts = {x: 0 for x in range(len(example_population))}

for i in range(1000):
    selected = tournament_selection(example_population, replacement=True)
    index = example_population.index(selected)
    counts[index] = counts[index] + 1

plt.bar(counts.keys(), counts.values())
plt.xticks(list(counts.keys()));
```



Even with small tournament size the worst individual was not chosen in any of these cases. The reason is that we are using tournament selection *without* replacement. If we pick any two individuals out of `example_population`, the individual with fitness 0 will *always* be worse. If we use replacement, then there is a chance that the worst individual gets selected:

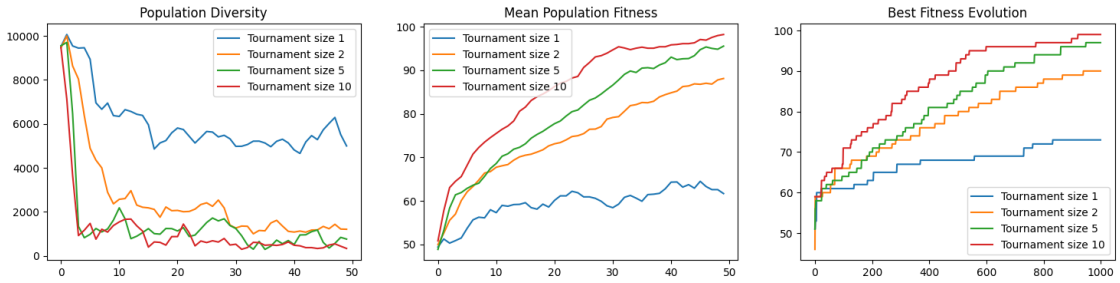
[23]:
```
tournament_size = 2

counts = {x: 0 for x in range(len(example_population))}

for i in range(1000):
    selected = tournament_selection(example_population, replacement=True)
    index = example_population.index(selected)
    counts[index] = counts[index] + 1

plt.bar(counts.keys(), counts.values())
plt.xticks(list(counts.keys()));
```



We can also observe the effects of the selective pressure throughout evolution.

[24]:
```
n = 100
from IPython.utils import io

elite_size = 1
selection = tournament_selection
tournament_sizes = [1, 2, 5, 10]
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 4))
for tournament_size in tournament_sizes:
    fitness_values, diversity_values, mean_fitness_values = [], [], []
    with io.capture_output() as captured:
        ga()
    axes[0].plot(diversity_values, label=f"Tournament size {tournament_size}")
    axes[1].plot(mean_fitness_values, label=f"Tournament size␣
 ↪{tournament_size}")
    axes[2].plot(fitness_values, label=f"Tournament size {tournament_size}")

axes[0].set_title('Population Diversity')
axes[0].legend()
axes[1].set_title('Mean Population Fitness')
axes[1].legend()
```

```
axes[2].set_title('Best Fitness Evolution')
axes[2].legend()
plt.show()
```

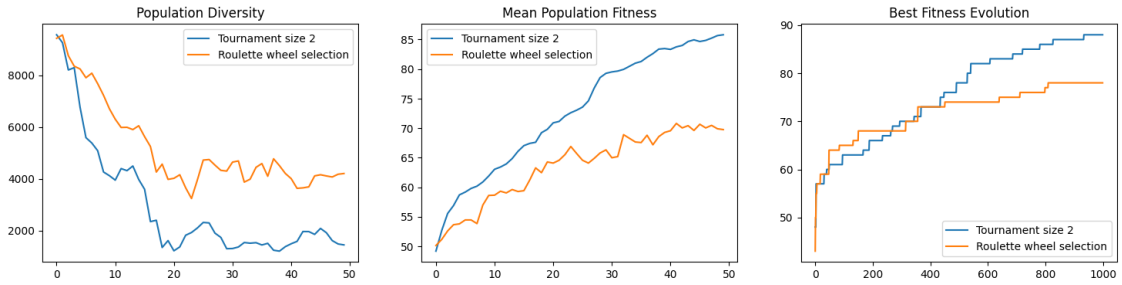


Too little selective pressure (e.g., tournament size of 1) tends to be bad. Very large tournaments might be too eager (which is not so much of a problem in one max though).

Should we use fitness proportionate selection or tournament selection?

[25]:
```
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 4))
n = 100
with io.capture_output() as captured:
    fitness_values = []
    diversity_values = []
    mean_fitness_values = []
    selection = tournament_selection
    tournament_size = 2
    ga()
    axes[0].plot(diversity_values, label=f"Tournament size {tournament_size}")
    axes[1].plot(mean_fitness_values, label=f"Tournament size␣
 ↪{tournament_size}")
    axes[2].plot(fitness_values, label=f"Tournament size {tournament_size}")

    selection = roulette_selection
    fitness_values = []
    mean_fitness_values = []
    diversity_values = []
    ga()
    axes[0].plot(diversity_values, label=f"Roulette wheel selection")
    axes[1].plot(mean_fitness_values, label=f"Roulette wheel selection")
    axes[2].plot(fitness_values, label=f"Roulette wheel selection")

axes[0].set_title('Population Diversity')
axes[0].legend()
axes[1].set_title('Mean Population Fitness')
axes[1].legend()
axes[2].set_title('Best Fitness Evolution')
```

```
axes[2].legend()
plt.show()
```



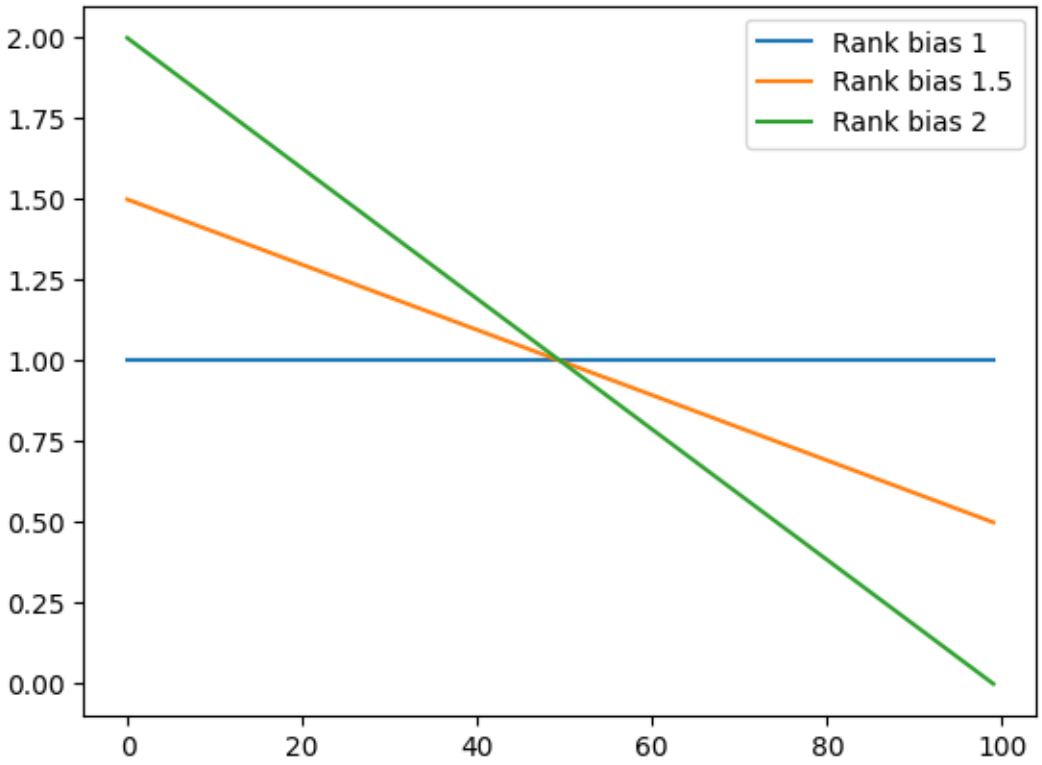
An alternative selection operator is *rank selection*, which is similar to fitness proportionate selection, except that the probability is calculated based on the *rank* in the population sorted by fitness, rather than the actual fitness value.

[26]:
```
rank_bias = 2
def rank_selection(population):
    population.sort(key=lambda c: get_fitness(c), reverse=True)

    individuals = []
    N = len(population)
    for i in range(N):
        f2 = rank_bias - (2 * i * (rank_bias - 1))/(N - 1)
        individuals.append((population[i], f2))

    # Now implement fitness proportionate selection using the f2 values
    fitness_sum = sum([f for (c, f) in individuals])
    pick = random.uniform(0, fitness_sum)
    current = 0
    for (chromosome, fitness) in individuals:
        current += fitness
        if current > pick:
            return chromosome
```

The rank bias, which is in the range `[1,2]` allows us to adjust the selective pressure.

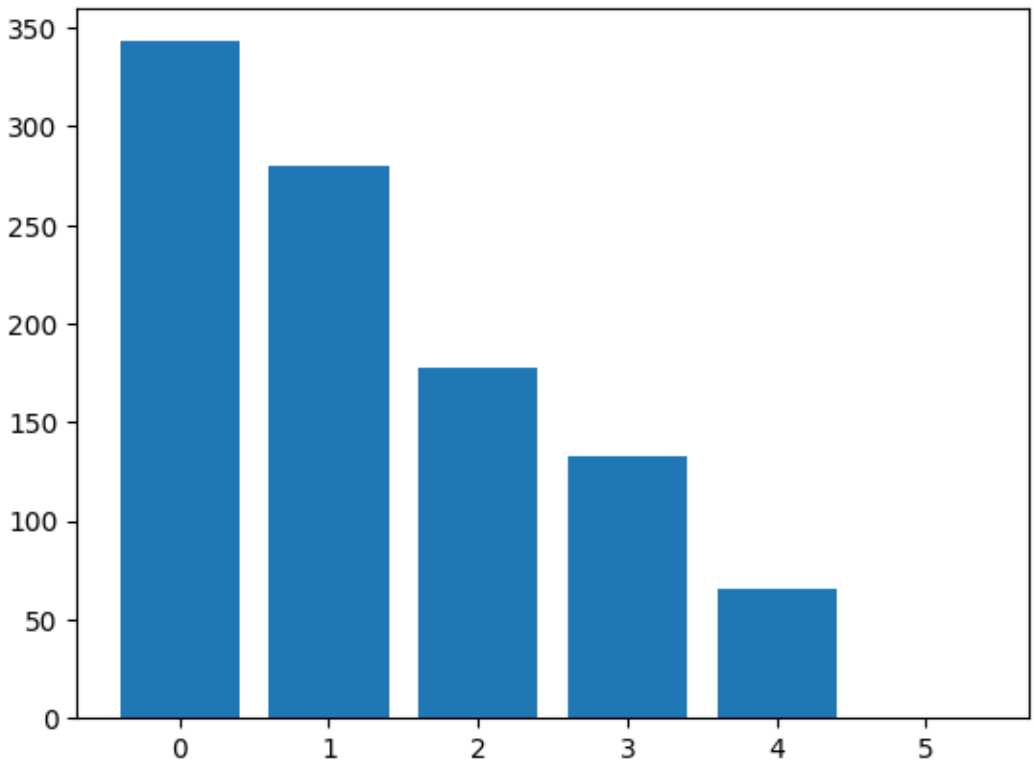
[27]:
```
N = 100
for rank_bias in [1, 1.5, 2]:
    plt.plot([rank_bias - (2 * i * (rank_bias - 1)) / (N - 1) for i in␣
 ↪range(100)], label = f"Rank bias {rank_bias}")
plt.legend()
```

[27]:
```
<matplotlib.legend.Legend at 0x1284b8580>
```



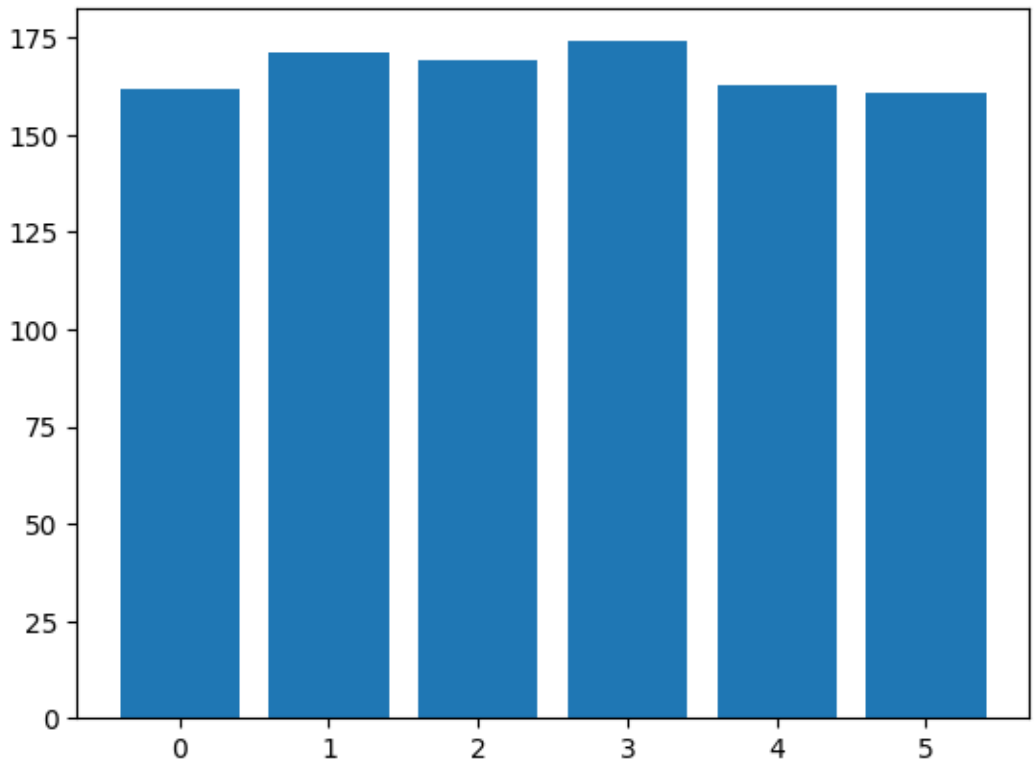
With a bias of 2, the worst individual has a 0% chance of being selected:

[28]:
```
rank_bias = 2

example_population = [ [1,1,1,1,1], [0,1,0,0,0], [1,0,0,0,0], [0,0,1,0,0],␣
 ↪[0,0,0,1,0],[0,0,0,0,0] ]
counts = {x: 0 for x in range(len(example_population))}

for i in range(1000):
    selected = rank_selection(example_population)
    index = example_population.index(selected)
    counts[index] = counts[index] + 1

plt.bar(counts.keys(), counts.values())
plt.xticks(list(counts.keys()));
```



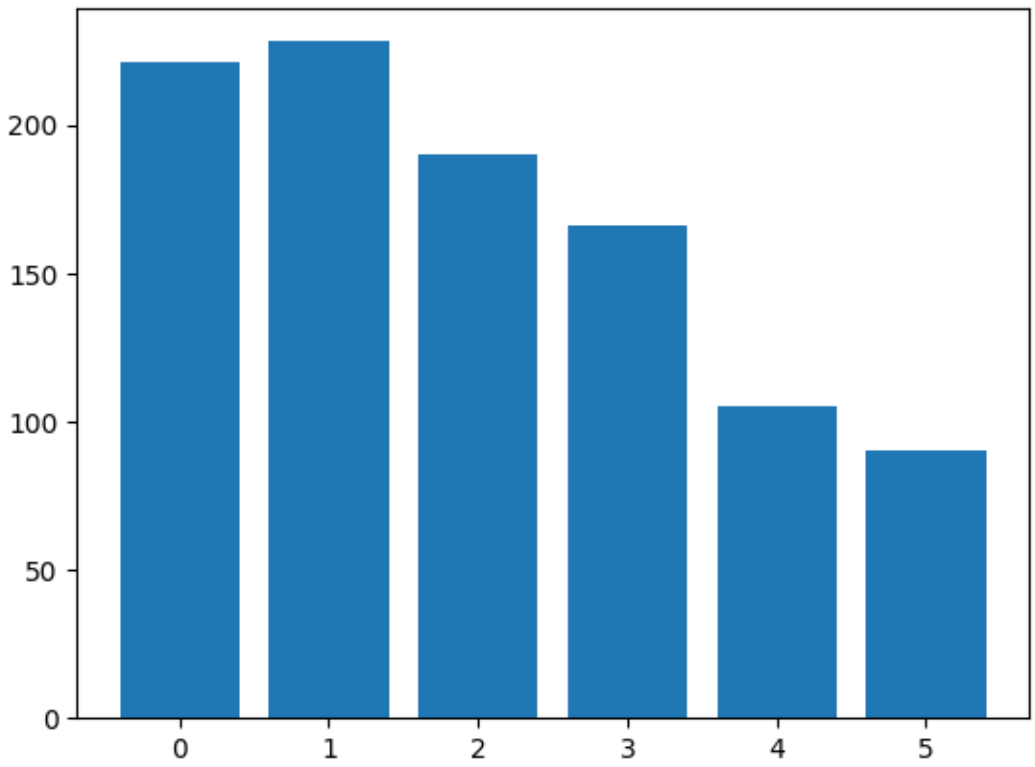
With a bias of 1, all individuals have the same probability of being selected:

[29]:
```
rank_bias = 1

example_population = [ [1,1,1,1,1], [0,1,0,0,0], [1,0,0,0,0], [0,0,1,0,0],␣
 ↪[0,0,0,1,0],[0,0,0,0,0] ]
counts = {x: 0 for x in range(len(example_population))}

for i in range(1000):
    selected = rank_selection(example_population)
    index = example_population.index(selected)
    counts[index] = counts[index] + 1

plt.bar(counts.keys(), counts.values())
plt.xticks(list(counts.keys()));
```



We will select a reasonable default for the selective pressure:

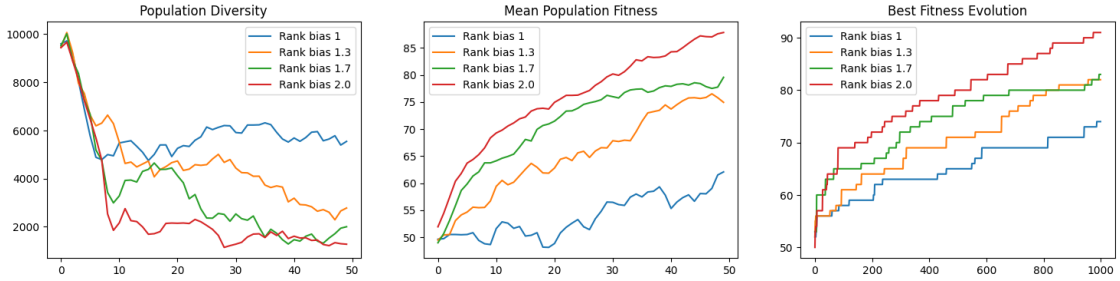
[30]:
```
rank_bias = 1.4

example_population = [ [1,1,1,1,1], [0,1,0,0,0], [1,0,0,0,0], [0,0,1,0,0],␣
 ↪[0,0,0,1,0],[0,0,0,0,0] ]
counts = {x: 0 for x in range(len(example_population))}

for i in range(1000):
    selected = rank_selection(example_population)
    index = example_population.index(selected)
    counts[index] = counts[index] + 1

plt.bar(counts.keys(), counts.values())
plt.xticks(list(counts.keys()));
```



We can again observe the effects of the rank bias on the evolution:

[31]:
```
from IPython.utils import io
selection = rank_selection
rank_biases = [1, 1.3, 1.7, 2.0]
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 4))
for rank_bias in rank_biases:
    fitness_values = []
    mean_fitness_values = []
    diversity_values = []
    with io.capture_output() as captured:
        ga()
    axes[0].plot(diversity_values, label=f"Rank bias {rank_bias}")
    axes[1].plot(mean_fitness_values, label=f"Rank bias {rank_bias}")
    axes[2].plot(fitness_values, label=f"Rank bias {rank_bias}")

axes[0].set_title('Population Diversity')
axes[0].legend()
axes[1].set_title('Mean Population Fitness')
axes[1].legend()
axes[2].set_title('Best Fitness Evolution')
axes[2].legend()
```

```
plt.show()
```



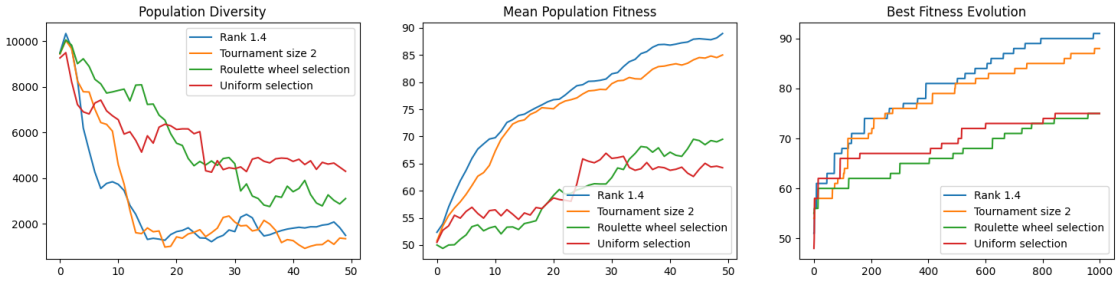
Some variants of genetic algorithms use selection where each individual has the same chance of being selected. Although this removes selection pressure, this is usually compensated with a strong fitness-based survivor selection mechanism.

[32]:
```
def uniform_selection(population):
    return random.choice(population)
```

We would not usually use uniform selection without some other survival selection, but to see what the effects on the search are, let's put all the options together in one experiment.

[33]:
```
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 4))
n = 100
with io.capture_output() as captured:
    rank_bias = 1.4
    fitness_values = []
    mean_fitness_values = []
    diversity_values = []
    selection = tournament_selection
    tournament_size = 2
    ga()
    axes[0].plot(diversity_values, label=f"Rank {rank_bias}")
    axes[1].plot(mean_fitness_values, label=f"Rank {rank_bias}")
    axes[2].plot(fitness_values, label=f"Rank {rank_bias}")

    fitness_values = []
    mean_fitness_values = []
    diversity_values = []
    selection = tournament_selection
    tournament_size = 2
    ga()
    axes[0].plot(diversity_values, label=f"Tournament size {tournament_size}")
    axes[1].plot(mean_fitness_values, label=f"Tournament size␣
 ↪{tournament_size}")
    axes[2].plot(fitness_values, label=f"Tournament size {tournament_size}")
```

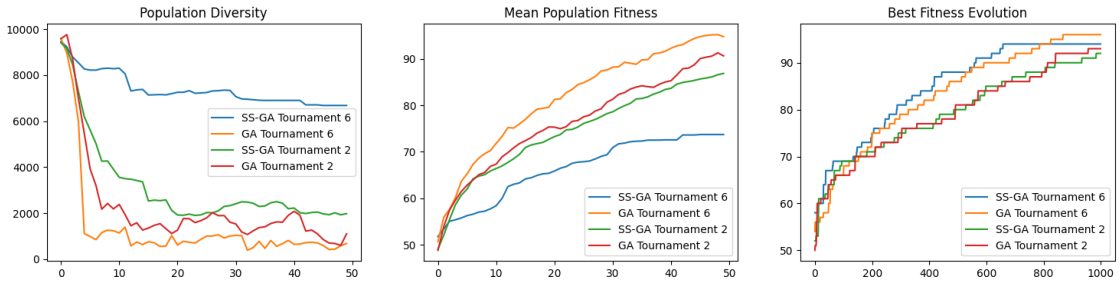
```
    selection = roulette_selection
    fitness_values = []
    mean_fitness_values = []
    diversity_values = []
    ga()
    axes[0].plot(diversity_values, label=f"Roulette wheel selection")
    axes[1].plot(mean_fitness_values, label=f"Roulette wheel selection")
    axes[2].plot(fitness_values, label=f"Roulette wheel selection")

    selection = uniform_selection
    fitness_values = []
    mean_fitness_values = []
    diversity_values = []
    ga()
    axes[0].plot(diversity_values, label=f"Uniform selection")
    axes[1].plot(mean_fitness_values, label=f"Uniform selection")
    axes[2].plot(fitness_values, label=f"Uniform selection")

axes[0].set_title('Population Diversity')
axes[0].legend()
axes[1].set_title('Mean Population Fitness')
axes[1].legend()
axes[2].set_title('Best Fitness Evolution')
axes[2].legend()
plt.show()
```



Survivor selection and parent selection are not independent: As an example, let's consider the effects of selective pressure on the generational GA and the steady state GA. We'll run tournament selection with two different tournament sizes to represent reasonable and high selective pressure.

[34]:
```
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 4))
n = 100
with io.capture_output() as captured:
    tournament_size = 6
    fitness_values = []
    mean_fitness_values = []
```

```
    diversity_values = []
    selection = tournament_selection
    steadystatega()
    axes[0].plot(diversity_values, label=f"SS-GA Tournament {tournament_size}")
    axes[1].plot(mean_fitness_values, label=f"SS-GA Tournament␣
↪{tournament_size}")
    axes[2].plot(fitness_values, label=f"SS-GA Tournament {tournament_size}")

    fitness_values = []
    mean_fitness_values = []
    diversity_values = []
    selection = tournament_selection
    ga()
    axes[0].plot(diversity_values, label=f"GA Tournament {tournament_size}")
    axes[1].plot(mean_fitness_values, label=f"GA Tournament {tournament_size}")
    axes[2].plot(fitness_values, label=f"GA Tournament {tournament_size}")

    tournament_size = 2
    fitness_values = []
    mean_fitness_values = []
    diversity_values = []
    selection = tournament_selection
    steadystatega()
    axes[0].plot(diversity_values, label=f"SS-GA Tournament {tournament_size}")
    axes[1].plot(mean_fitness_values, label=f"SS-GA Tournament␣
↪{tournament_size}")
    axes[2].plot(fitness_values, label=f"SS-GA Tournament {tournament_size}")

    fitness_values = []
    mean_fitness_values = []
    diversity_values = []
    selection = tournament_selection
    ga()
    axes[0].plot(diversity_values, label=f"GA Tournament {tournament_size}")
    axes[1].plot(mean_fitness_values, label=f"GA Tournament {tournament_size}")
    axes[2].plot(fitness_values, label=f"GA Tournament {tournament_size}")



axes[0].set_title('Population Diversity')
axes[0].legend()
axes[1].set_title('Mean Population Fitness')
axes[1].legend()
axes[2].set_title('Best Fitness Evolution')
axes[2].legend()
plt.show()
```



With high selective pressure, the steady state GA maintains very high diversity, even though the best fitness improves reasonably over time. The reason is that with a large tournament size of 6 (with a population size of 20) the parent selection will repeatedly select the same few individuals with a high probability. If the result leads to an improvement, these are replaced but will be picked again with high probability afterwards, while large parts of the population (worse individuals that keep losing tournaments) will simply remain unchanged. This is also reflected by the lower average fitness in the population. For the generational GA the high selective pressure has the opposite effect: The population loses diversity very quickly, because most offspring will be produced from the same few parents. For one max, this doesn't appear to be a bad thing though.

1.3 Alternative Crossover Operators

Our crossover operator so far only considers a single point for crossing two individuals. While this is the most common variant in practice, there is one potential downside: Only locally neighbouring genetic material is preserved; if a parent has relevant genes at the beginning and the end of the chromosome, these will not be inherited to the offspring directly. One way to circumvent this is by defining more than one crossover point. For example, we can define a two-point crossover operator:

[35]:
```
def twopoint_crossover(parent1, parent2):
    pos1 = random.randint(1, len(parent1))
    pos2 = random.randint(pos1, len(parent1))
    offspring1 = parent1[:pos1] + parent2[pos1:pos2] + parent1[pos2:]
    offspring2 = parent2[:pos1] + parent1[pos1:pos2] + parent2[pos2:]
    return offspring1, offspring2
```

[36]:
```
parent1 = [0,0,0,0,0,0,0,0,0,0]
parent2 = [1,1,1,1,1,1,1,1,1,1]
```

In the single point crossover, the offspring of `parent1` and `parent2` will *always* be either a sequence of `0` followed by a sequence of `1`, or vice versa:

[37]:
```
singlepoint_crossover(parent1, parent2)
```

[37]:
```
([0, 0, 0, 0, 0, 0, 0, 0, 0, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1, 0])
```

In the two point crossover, there will be some variation:

[38]:
```
twopoint_crossover(parent1, parent2)
```

[38]:
```
([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

It is not common to increase the number of crossover points beyond two, but instead if more variation is required, it is simply possible to *uniformly* select genes from either of the parents, resulting in *uniform crossover*:

[39]:
```
def uniform_crossover(parent1, parent2):
    offspring1 = []
    offspring2 = []
    for pos in range(len(parent1)):
        if random.choice([True, False]):
            offspring1.append(parent1[pos])
            offspring2.append(parent2[pos])
        else:
            offspring1.append(parent2[pos])
            offspring2.append(parent1[pos])
    return offspring1, offspring2
```

Applying this to `parent1` and `parent2` from above, we will see offspring consisting of more variation in `1`s and `0`s, but these will always be chosen from parents and not random:

[40]:
```
uniform_crossover(parent1, parent2)
```

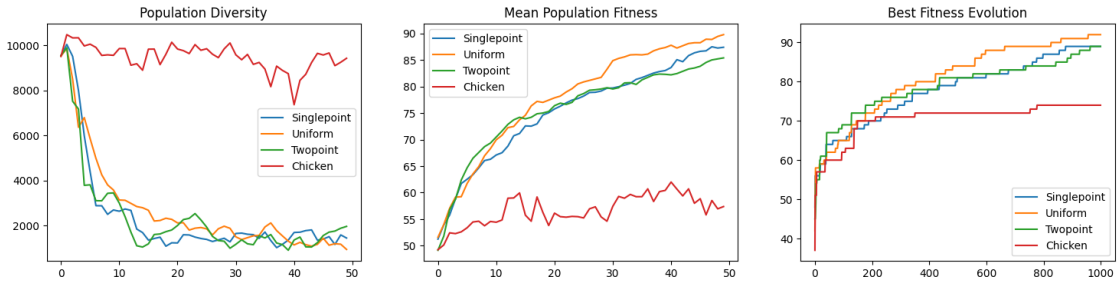
[40]:
```
([0, 1, 1, 1, 0, 1, 1, 1, 0, 1], [1, 0, 0, 0, 1, 0, 0, 0, 1, 0])
```

The importance and influence of the crossover operator on the search is part of active research, and also depends on the problem we are trying to solve. To see whether there are any benefits to using crossover on our one max example, we can conduct a *headless chicken* test: During crossover, we use a randomly generated individual as one of the parents. If the search still performs as well, then the crossover operator actually just serves as a kind of macro-mutation. If the search no longer performs as well, then it is the actual combination of parent genetic material that leads to an improvement.

[41]:
```
def chicken_crossover(parent1, parent2):
    offspring1 = []
    offspring2 = []
    parent2 = get_random_solution()
    for pos in range(len(parent1)):
        if random.choice([True, False]):
            offspring1.append(parent1[pos])
            offspring2.append(parent2[pos])
        else:
            offspring1.append(parent2[pos])
            offspring2.append(parent1[pos])
    return offspring1, offspring2
```

We will run the usual combination of experiments and analyses:

[42]:
```
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 4))
n = 100
with io.capture_output() as captured:
    crossover = singlepoint_crossover
    fitness_values = []
    mean_fitness_values = []
    diversity_values = []
    ga()
    axes[0].plot(diversity_values, label=f"Singlepoint")
    axes[1].plot(mean_fitness_values, label=f"Singlepoint")
    axes[2].plot(fitness_values, label=f"Singlepoint")

    crossover = uniform_crossover
    fitness_values = []
    mean_fitness_values = []
    diversity_values = []
    ga()
    axes[0].plot(diversity_values, label=f"Uniform")
    axes[1].plot(mean_fitness_values, label=f"Uniform")
    axes[2].plot(fitness_values, label=f"Uniform")

    crossover = twopoint_crossover
    fitness_values = []
    mean_fitness_values = []
    diversity_values = []
    ga()
    axes[0].plot(diversity_values, label=f"Twopoint")
    axes[1].plot(mean_fitness_values, label=f"Twopoint")
    axes[2].plot(fitness_values, label=f"Twopoint")

    crossover = chicken_crossover
    fitness_values = []
    mean_fitness_values = []
    diversity_values = []
    ga()
    axes[0].plot(diversity_values, label=f"Chicken")
    axes[1].plot(mean_fitness_values, label=f"Chicken")
    axes[2].plot(fitness_values, label=f"Chicken")

axes[0].set_title('Population Diversity')
axes[0].legend()
axes[1].set_title('Mean Population Fitness')
axes[1].legend()
axes[2].set_title('Best Fitness Evolution')
axes[2].legend()
plt.show()
```



This experiment just conducts a single run, and as usual we need to conduct an experiment with repetitions in order to draw conclusions. However, what most likely shows is that the headless chicken test leads to substantially worse results, while the uniform crossover tends to produce the best results. Consequently, it seems that crossover actually is useful for our problem.

1.4 Mutation Operators

The mutation operator we considered so far flips each bit in a sequence of length n with a probability of $1/n$. To be able to configure our genetic algorithm with alternative mutation operators, let's redefine this operator in a differently named function.

[43]:
```
def avg_mutate(individual):
    P_mutate = 1/len(individual)
    copy = individual[:]
    for position in range(len(individual)):
        if random.random() < P_mutate:
            copy[position] = 1 - copy[position]
    return copy
```

One of the reasons we used this operator so far was that it implicitly defines a mutation probability and we had one less parameter to worry about. However, when considering alternative operators, we will require a probability for applying mutation. This probability is usually fairly small.

[44]:
```
P_mutate = 0.05
```

A basic alternative operator would be to flip a single bit, with probability `P_mutate`.

[45]:
```
def mutate1(individual):
    copy = individual[:]
    if random.random() < P_mutate:
        position = random.randint(0, len(copy) - 1)
        copy[position] = 1 - copy[position]
    return copy
```

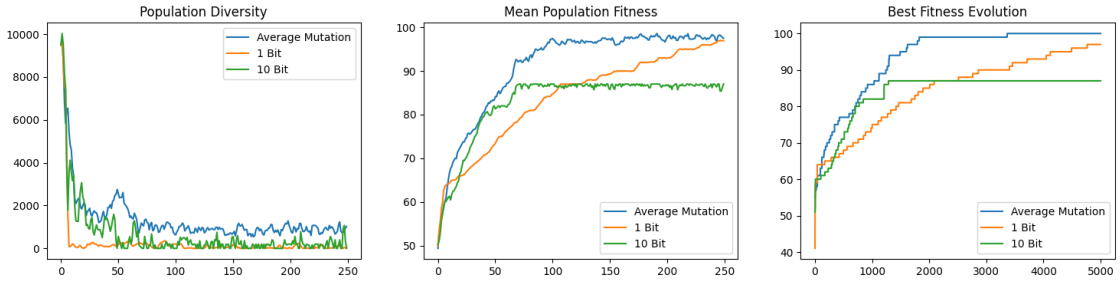
The scope for different operators on our bitvector representation is limited. Let's consider an alternative where we flip multiple bits at the same time.

[46]:
```
def mutate10(individual):
    copy = individual[:]
    if random.random() < P_mutate:
        for _ in range(10):
            position = random.randint(0, len(copy) - 1)
            copy[position] = 1 - copy[position]
    return copy
```

Given these three mutation operators, we can now run some comparative experiments again.

[47]:
```
crossover = singlepoint_crossover
selection = tournament_selection
tournament_size = 2

fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 4))
n = 100
max_steps = 5000
with io.capture_output() as captured:
    fitness_values = []
    mean_fitness_values = []
    diversity_values = []
    mutate = avg_mutate
    ga()
    axes[0].plot(diversity_values, label=f"Average Mutation")
    axes[1].plot(mean_fitness_values, label=f"Average Mutation")
    axes[2].plot(fitness_values, label=f"Average Mutation")

    mutate = mutate1
    fitness_values = []
    mean_fitness_values = []
    diversity_values = []
    ga()
    axes[0].plot(diversity_values, label=f"1 Bit")
    axes[1].plot(mean_fitness_values, label=f"1 Bit")
    axes[2].plot(fitness_values, label=f"1 Bit")

    mutate = mutate10
    fitness_values = []
    mean_fitness_values = []
    diversity_values = []
    ga()
    axes[0].plot(diversity_values, label=f"10 Bit")
    axes[1].plot(mean_fitness_values, label=f"10 Bit")
    axes[2].plot(fitness_values, label=f"10 Bit")


axes[0].set_title('Population Diversity')
axes[0].legend()
```

```
axes[1].set_title('Mean Population Fitness')
axes[1].legend()
axes[2].set_title('Best Fitness Evolution')
axes[2].legend()
plt.show()
```



Mutating each bit with a probability of $1/n$ is beneficial for the diversity. Comparing the operators that flip 1 and 10 bits, we typically can observe that the 10 bit flip makes larger jumps in fitness improvements, but towards the end of the search it will struggle to home in on a solution as it flips *too much*.

1.5 Cellular Genetic Algorithms

The last aspect of the genetic algorithm we will consider is the population: The population is usually a multiset, and we mostly implemented it as a list to impose an order on individuals. A cellular evolutionary algorithm imposes a topology on the population. During reproduction, an individual is only allowed to mate with its neighbours, as defined by the topology (commonly rings, grids, or two-dimensional torus graphs).

Let's put our population into a grid, i.e. a two-dimensional list. Let's assume we consider only the 4-neighbourhood (von Neumann neighbourhood) during selection, then we randomly select one of these 4 neighbours and the selected position itself:

[48]:
```
def grid_selection(population, row, col):
    neighbours = []
    for dx in [-1, 1]:
        if row + dx >= 0 and row + dx < len(population):
            neighbours.append(population[row + dx][col])
    for dy in [-1, 0, 1]:
        if col + dy >= 0 and col + dy < len(population):
            neighbours.append(population[row][col + dy])

    return random.choice(neighbours)
```

The genetic algorithm needs to be modified such that the population is a properly initialised grid, and then uses the `grid_selection` we just defined:

[49]:
```
grid_size = 5
```

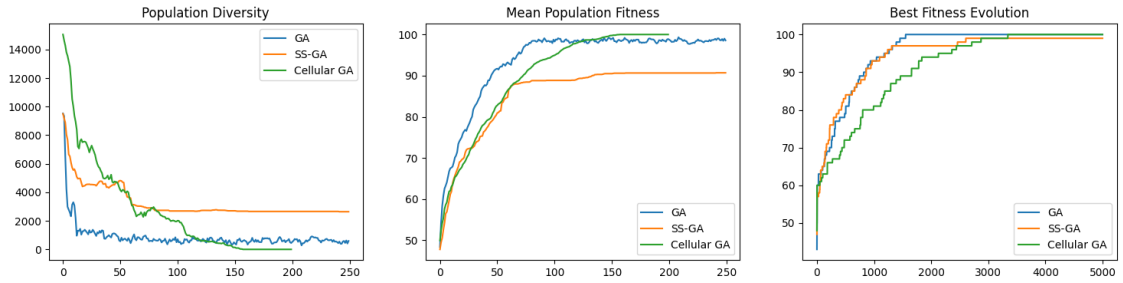
[50]:
```
def cellular_ga():
    fitness_values.clear()
    population = [[get_random_solution() for _ in range(grid_size)] for _ in␣
↪range(grid_size)]

    diversity_values.append(pairwise_distance([y for x in population for y in␣
↪x]))
    mean_fitness_values.append(mean([get_fitness(x) for x in [y for x in␣
↪population for y in x]]))

    best_fitness = -1
    for p in [y for x in population for y in x]:
        fitness = get_fitness(p)
        if fitness > best_fitness:
            best_fitness = fitness
        fitness_values.append(best_fitness)

    while len(fitness_values) < max_steps:
        new_population = []
        for row in range(grid_size):
            new_population.append([])
            for col in range(grid_size):
                parent1 = grid_selection(population, row, col)
                parent2 = grid_selection(population, row, col)

                if random.random() < P_xover:
                    offspring1, offspring2 = crossover(parent1, parent2)
                else:
                    offspring1, offspring2 = parent1[:], parent2[:]

                offspring = mutate(random.choice([offspring1, offspring2]))

                if get_fitness(offspring) >= get_fitness(population[row][col]):
                    new_population[row].append(offspring)
                else:
                    new_population[row].append(population[row][col])

                if get_fitness(offspring) > best_fitness:
                    best_fitness = get_fitness(offspring)
                fitness_values.append(best_fitness)

        population = new_population

        diversity_values.append(pairwise_distance([y for x in population for y␣
↪in x]))
        mean_fitness_values.append(mean([get_fitness(x) for x in [y for x in␣
↪population for y in x]]))
```

```
    return max([y for x in population for y in x], key=lambda k: get_fitness(k))
```

When producing the next generation the cellular GA iterates over the grid, and produces an offspring at each grid location using only the neighbourhood for reproduction. We have implemented an elitist approach where the grid location is only replaced with the offspring if the offspring has the same or better fitness.

[51]:
```
mutate = avg_mutate
crossover = singlepoint_crossover
selection = tournament_selection
tournament_size = 3

fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 4))
n = 100
max_steps = 5000
with io.capture_output() as captured:
    fitness_values = []
    mean_fitness_values = []
    diversity_values = []
    ga()
    axes[0].plot(diversity_values, label=f"GA")
    axes[1].plot(mean_fitness_values, label=f"GA")
    axes[2].plot(fitness_values, label=f"GA")

    fitness_values = []
    mean_fitness_values = []
    diversity_values = []
    steadystatega()
    axes[0].plot(diversity_values, label=f"SS-GA")
    axes[1].plot(mean_fitness_values, label=f"SS-GA")
    axes[2].plot(fitness_values, label=f"SS-GA")

    fitness_values = []
    mean_fitness_values = []
    diversity_values = []
    cellular_ga()
    axes[0].plot(diversity_values, label=f"Cellular GA")
    axes[1].plot(mean_fitness_values, label=f"Cellular GA")
    axes[2].plot(fitness_values, label=f"Cellular GA")


axes[0].set_title('Population Diversity')
axes[0].legend()
axes[1].set_title('Mean Population Fitness')
axes[1].legend()
axes[2].set_title('Best Fitness Evolution')
```

```
axes[2].legend()
plt.show()
```



For larger grids the cellular GA may be slower initially – it takes longer for genetic material to spread across the population of a large grid. This, however, is also its benefit: Premature convergence is less likely to occur.

1.6 Memetic Algorithms

Since an improvement to a solution does not give us any guarantees that we are in fact moving towards the optimum, evolutionary algorithms allow non-local moves. However, evolution may take longer to fine-tune solutions, while local search does exactly this very efficiently. The term *Memetic Algorithms* is commonly used to denote the combination of global and local search. The term sometimes also refers to the use of instance-specific knowledge in search operators, but we will focus on the combination of global and local search in this notebook.

As an example problem we consider a variation of the one-max problem, such that the search operators are simplistic, and we understand the search landscape well. We are still trying to optimise the bitstring to contain all 1s, but assume we have a somewhat more complex fitness landscape. Given a bitstring encoding, let the fitness function be the following *hurdle* function:

$$f(x) = -\left\lceil \frac{z(x)}{w} \right\rceil - \frac{rem(z(x),w)}{w}$$

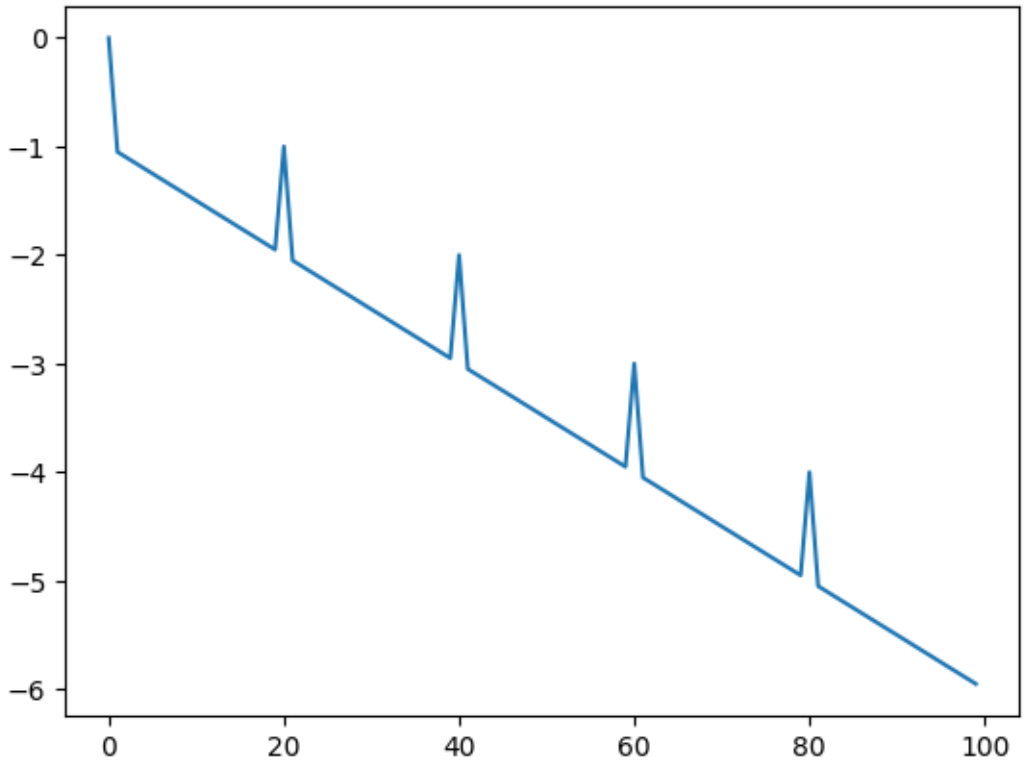
Here $z(x)$ is the number of zeros in the bitstring x; $w \in \{2, 3, \ldots, n\}$ is the hurdle width; $rem(z(x), w)$ is the remainder of $z(x)$ divided by w, and $\lceil \cdot \rceil$ is the ceiling function.

It will be easiest to understand what this does by looking at the resulting fitness landscape over a limited input range.

[52]:
```
w = 20
```

[53]:
```
values = []
for z in range(0, 100):
    values.append(-math.ceil(z/w) - (z % w)/w)
plt.plot(values)
```

[53]:
```
[<matplotlib.lines.Line2D at 0x1282913c0>]
```



The optimal value has the fitness value 0 (i.e. $z(0) = 0$ implies all bits are 1). The representation is our usual bitstring, and we will use a moderate length for our bitstring:

[54]:
```
n = 100
```

As usual, an individual is a random sequence of bits.

[55]:
```
def get_random_solution():
    domain = [0,1]
    return [random.choice(domain) for _ in range(n)]
```

The fitness function calculates the hurdle function. In order to avoid redundantly calculating fitness values, we first check if the individual already has a cached fitness value, and if so, we simply return that. If we do have to calculate a new fitness value then we append an item to our `fitness_values` list; we will use this list to store the best fitness value seen to date, so that we can plot the result afterwards. We will also use it as a stopping condition, since it reliably keeps track of how many times the fitness was actually *calculated*.

[56]:
```
fitness_values = []

def get_fitness(solution):
    z = len(solution) - sum(solution)
    fitness = -math.ceil(z/w) - (z % w)/w
```

```
    if not fitness_values:
        fitness_values.append(fitness)
    else:
        fitness_values.append(max(fitness, fitness_values[-1]))
    return fitness
```

[57]:
```
get_fitness(get_random_solution())
```

[57]:
```
-3.75
```

Let's first consider how local search fares with our hurdle problem. We define a neighbourhood and recall our basic hillclimber.

[58]:
```
def get_neighbours(candidate):
    neighbours = []
    for pos in range(len(candidate)):
        copy = candidate[:]
        copy[pos] = 1 - copy[pos]
        neighbours.append(copy)
    return neighbours
```

As a small change to prior implementations, we will use the `fitness_values` list as a stopping condition, and make sure that we don't exceed `max_steps` fitness evaluations.

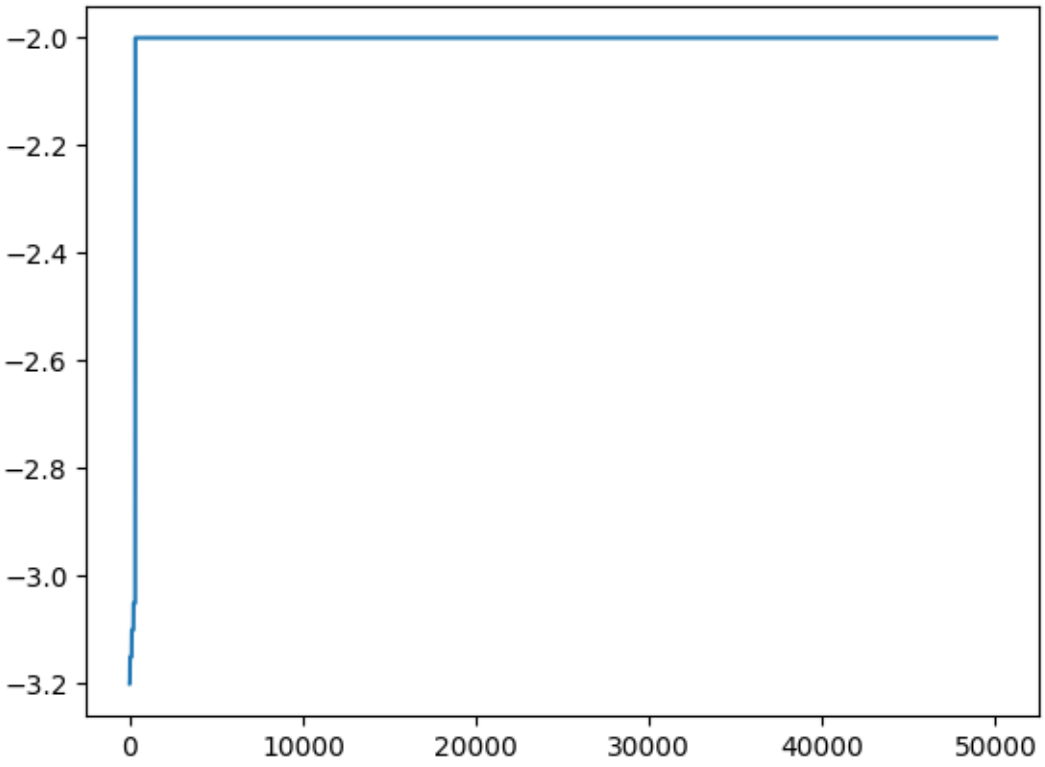
[59]:
```
max_steps = 50000
```

[60]:
```
def hillclimbing_restart():
    fitness_values.clear()
    current = get_random_solution()
    best = current[:]
    best_fitness = get_fitness(current)

    while len(fitness_values) < max_steps:

        best_neighbour = None
        neighbour_fitness = -sys.maxsize
        for neighbour in get_neighbours(current):
            fitness = get_fitness(neighbour)

            if fitness > neighbour_fitness:
                best_neighbour = neighbour
                neighbour_fitness = fitness

        # Random restart if no neighbour is better
        if neighbour_fitness <= get_fitness(current):
            current = get_random_solution()
            neighbour_fitness = get_fitness(current)
```

```
        else:
            current = best_neighbour

        if neighbour_fitness > best_fitness:
            best = current[:]
            best_fitness = neighbour_fitness

    return best
```

[61]:
```
hillclimbing_restart()
plt.plot(fitness_values)
```

[61]: [<matplotlib.lines.Line2D at 0x1283d6bc0>]



The hillclimber is quite unlikely to reach the optimal solution: The fitness landscape contains many local optima, and a random (re-)start needs to luckily jump beyond the last hurdle for the gradient to point to the global optimum.

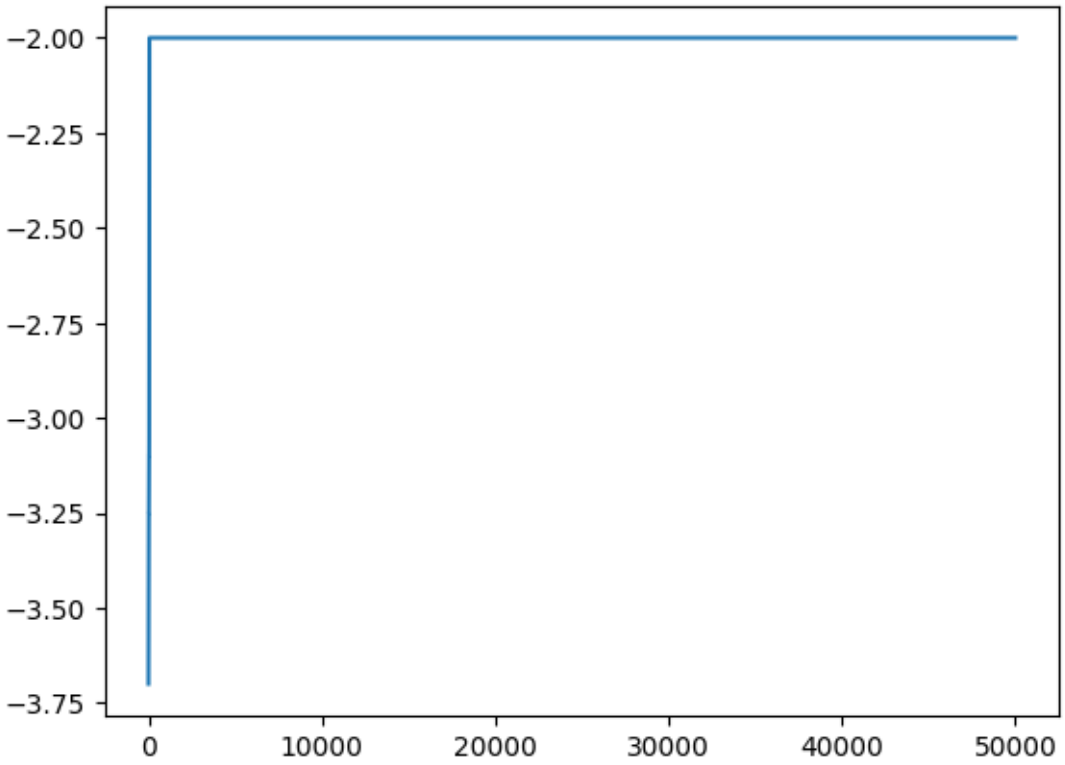
Let's compare this to a basic evolutionary search algorithm; to keep things simple initially we will just use a (1+1)EA, so we just need to add a mutation function that implements the usual probabilistic bitflips:

[62]:
```
def mutate(solution):
    P_mutate = 1/len(solution)
    mutated = solution[:]
    for position in range(len(solution)):
        if random.random() < P_mutate:
            mutated[position] = 1 - mutated[position]
    return mutated
```

[63]:
```
def oneplusoneea():
    fitness_values.clear()
    current = get_random_solution()
    fitness = get_fitness(current)

    while len(fitness_values) < max_steps:
        candidate = mutate(current)
        candidate_fitness = get_fitness(candidate)
        if candidate_fitness >= fitness:
            fitness = candidate_fitness
            current = candidate

    return candidate
```

[64]:
```
oneplusoneea()
plt.plot(fitness_values)
```

[64]:
```
[<matplotlib.lines.Line2D at 0x12836edd0>]
```



The hurdles are problematic also for this global search algorithm: The search not only needs to jump over all hurdles, but it also needs to jump higher than the preceding hurdles. Again the result tends to be that the algorithm does not find an optimal solution.

A memetic algorithm combines these two algorithms, and can overcome these problems: The global search can effectively jump over hurdles, and the local search can climb the next hurdle. As an initial memetic algorithm we create a simple (1+1)MA. For this we will adapt the hillclimber so that it takes a starting point of the search as a parameter. To avoid spending too much time exploring the neighbourhood, we will also make this a first-ascent hillclimber:

[65]:
```
def hillclimbing(starting_point):

    current =  starting_point
    fitness = get_fitness(current)

    improved = True
    while improved and len(fitness_values) < max_steps:
        improved = False

        for neighbour in get_neighbours(current):
            neighbour_fitness = get_fitness(neighbour)

            if neighbour_fitness > fitness:
```

```
                current = neighbour
                fitness = neighbour_fitness
                improved = True
                break

    return current, fitness
```

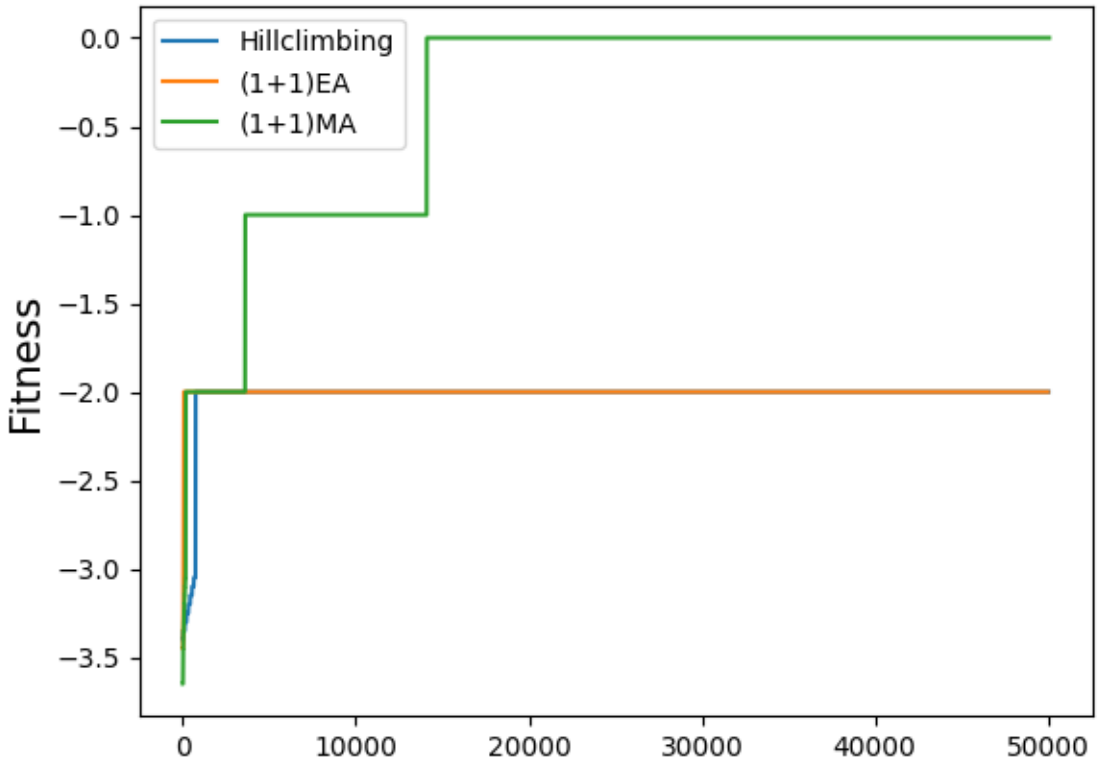
The (1+1)MA now applies local search after each mutation step:

[66]:
```
def oneplusonema():
    fitness_values.clear()
    current = get_random_solution()
    fitness = get_fitness(current)

    while len(fitness_values) < max_steps:
        candidate = mutate(current)
        candidate, candidate_fitness = hillclimbing(candidate)
        if candidate_fitness >= fitness:
            fitness = candidate_fitness
            current = candidate

    return candidate
```

Now let's see how the three algorithms compare on our hurdle problem:

[67]:
```
hillclimbing_restart()
hc_values = fitness_values[:]

oneplusoneea()
opo_values = fitness_values[:]

oneplusonema()
opoma_values = fitness_values[:]

plt.ylabel('Fitness', fontsize=15)
plt.plot(hc_values, label = "Hillclimbing")
plt.plot(opo_values, label = "(1+1)EA")
plt.plot(opoma_values, label = "(1+1)MA")
plt.legend()
```

[67]:
```
<matplotlib.legend.Legend at 0x1286dfc10>
```



The local improvement actually reflects Lamarckian evolution theory, as individuals can change their own genotype. An alternative way to implement a memetic algorithm would be to exploit the *Baldwin effect*: The genotype is not improved by the local search, but the fitness is evaluated on an improved genotype. The Baldwin effect describes how the preference of the locally improvable individuals leads to this establishing in the genotype eventually through evolution. However, when implementing algorithms we are not bound by biological realisties, so we can just implement Lamarckism.

When generalising our (1+1)MA to population-based memetic algorithms we are faced with a new parameter: Applying local improvement on *all* offspring is intuitively very computationally expensive, as we need to explore the neighbourhoods of (potentially) many individuals. The local improvement is therefor usually only done probabilistically.

[68]:
```
P_localsearch = 0.1
```

[69]:
```
def ma():
    fitness_values.clear()
    population = [get_random_solution() for _ in range(population_size)]
    best_solution = max(population, key=lambda k: get_fitness(k))
    best_fitness = get_fitness(best_solution)

    while len(fitness_values) < max_steps:
        new_population = []
```

```
        while len(new_population) < len(population):
            parent1 = tournament_selection(population)
            parent2 = tournament_selection(population)

            if random.random() < P_xover:
                offspring1, offspring2 = crossover(parent1, parent2)
            else:
                offspring1, offspring2 = parent1, parent2

            offspring1 = mutate(offspring1)
            offspring2 = mutate(offspring2)

            if random.random() < P_localsearch:
                offspring1, offspring1_fitness = hillclimbing(offspring1)
                offspring2, offspring2_fitness = hillclimbing(offspring2)

            new_population.append(offspring1)
            new_population.append(offspring2)

        population = new_population

        best_solution = max(population, key=lambda k: get_fitness(k))
        best_fitness = get_fitness(best_solution)

    return best_solution
```

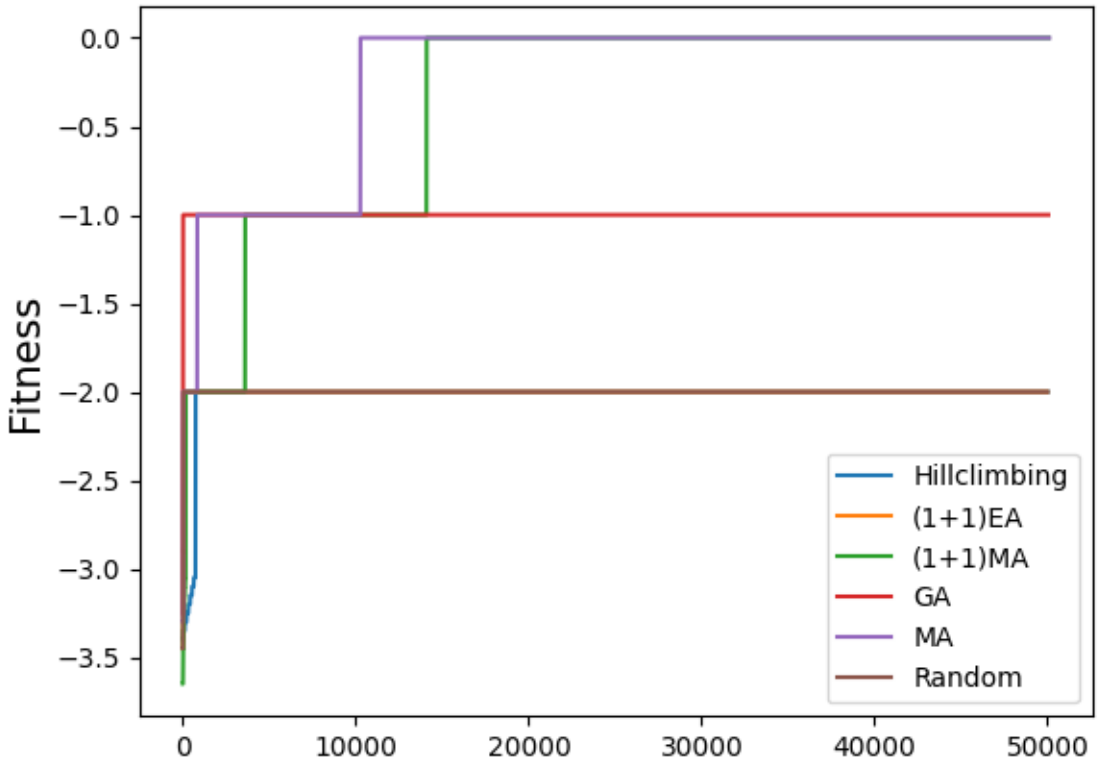
As a baseline for our experiments, we will also use a random search as usual:

[70]:
```
def randomsearch():
    fitness_values.clear()
    best = get_random_solution()
    best_fitness = get_fitness(best)

    while len(fitness_values) < max_steps:
        candidate = get_random_solution()
        fitness = get_fitness(candidate)
        if fitness > best_fitness:
            best = candidate
            best_fitness = fitness

    return best
```

[71]:
```
ga()
ga_values = fitness_values[:]

ma()
ma_values = fitness_values[:]
```

```
randomsearch()
random_values = fitness_values[:]

plt.ylabel('Fitness', fontsize=15)
plt.plot(hc_values, label = "Hillclimbing")
plt.plot(opo_values, label = "(1+1)EA")
plt.plot(opoma_values, label = "(1+1)MA")
plt.plot(ga_values, label = "GA")
plt.plot(ma_values, label = "MA")
plt.plot(random_values, label = "Random")
plt.legend()
```

[71]: <matplotlib.legend.Legend at 0x128621300>



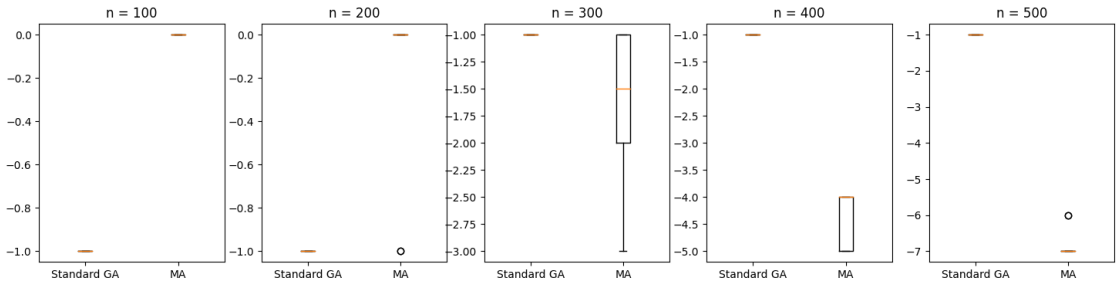
Individual runs may vary (and we will do a more systematic comparison later on), but generally the MA-variants tend to find optimal solutions, while the others do not.

While the memetic versions seem to have an edge over the non-memetic versions, the performance is dependent on the size of the neighbourhood: For example, if we use a large `n` in a bitstring representation, then the neighbourhood may become very large, and the exploration used in the local search may be very expensive. An upper bound on the number of fitness evaluations could prevent the local search from wasting the search budget in those cases where the local search does not

actually contribute to improving the solution. We can investigate the effect of the neighbourhood size by re-running the search for different values of `n`:

[72]:
```
from IPython.utils import io

def run_times(algorithm, repetitions):
    global fitness_values
    result = []
    for i in range(repetitions):
        with io.capture_output() as captured:
            algorithm()
        result.append(fitness_values[-1])
    return result
```

[73]:
```
fig, axes = plt.subplots(nrows=1, ncols=5, figsize=(18, 4))

num_plot = 0
for n in [100, 200, 300, 400, 500]:
    results = {
        "Standard GA"  : run_times(ga, 10),
        "MA"           : run_times(ma, 10)
    }
    axes[num_plot].boxplot(results.values())
    axes[num_plot].set_title(f"n = {n}")
    axes[num_plot].set_xticklabels(results.keys())
    num_plot += 1
plt.show()
```



As a point for comparison, let's also consider the standard one-max problem for the same sizes, where we should see a bit more spread in terms of fitness values:

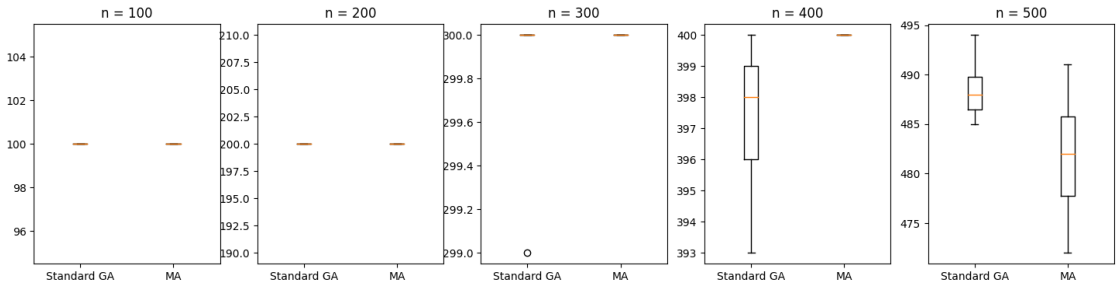
[74]:
```
hurdle_fitness = get_fitness
```

[75]:
```
def get_fitness(solution):
    fitness = sum(solution)

    if not fitness_values:
```

```
        fitness_values.append(fitness)
    else:
        fitness_values.append(max(fitness, fitness_values[-1]))
    return fitness
```

[76]:
```
onemax_fitness = get_fitness
```

[77]:
```
fig, axes = plt.subplots(nrows=1, ncols=5, figsize=(18, 4))

num_plot = 0
for n in [100, 200, 300, 400, 500]:
    results = {
        "Standard GA"  : run_times(ga, 10),
        "MA"           : run_times(ma, 10)
    }
    axes[num_plot].boxplot(results.values())
    axes[num_plot].set_title(f"n = {n}")
    axes[num_plot].set_xticklabels(results.keys())
    num_plot += 1
plt.show()
```



[78]:
```
# Restore
get_fitness = hurdle_fitness
n = 100
```