1. Debugging Outputs

Question 1

(a) length of schedule: 5940 for 30K, 5980 for 10K. The function evaluation of your solution usually falls in the range of 3000-5000

Question 2

- "pairs" refers to the pairing instruction in 1b, which you need to do every time the population changes
- You can use the following to debug your cross-over and mutate operators

In [85]:
```
import numpy as np

def plot_surface(func,x_min=-2,x_max=2,y_min=-2,y_max=2):
    x = np.linspace(x_min,x_max,100)
    y = np.linspace(y_min,y_max,100)
    X,Y = np.meshgrid(x,y)
    Z = func([X,Y])

    fig = plt.figure(figsize=(6,3))
    ax = fig.add_subplot(111, projection='3d')
    ax.plot_surface(X,Y,Z, alpha=0.9)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    # ax.view_init(60,60) # set angles for viewing
    plt.tight_layout()
    plt.show()
```

Simulated annealing code Q1

In [39]:
```
def SA(solution,evaluation,delta,boundary,cooling_schedule,verbose=False):
    """ Simulated Annealing for minimization
    solution: np.array. Initial guess of solution
    evaluation: func. Function to evaluate solution
    delta: float. Magnitude of random displacement
    boundary: array of int/float. [lowerbound,upperbound]
    cooling_schedule: np.array. An array of temperatures for simulated annea
    """
    best_solution = solution.copy()
    lowest_eval = evaluation(best_solution)
    for idx,temp in enumerate(cooling_schedule):
        if idx%500 == 0 and verbose:
            print(f'{str(idx).zfill(len(str(len(cooling_schedule))))}/{len(c
        for n in range(len(solution)):
```

```
                trial = solution.copy()
                trial[n] += delta*(2*np.random.random()-1)
                if trial[n] >= boundary[0] and trial[n] <= boundary[1]:
                    #fill in acceptance criterion
                    accepted = np.exp( - (evaluation(trial)-evaluation(solution)
                    if accepted > np.random.random():
                        solution = trial
                        if evaluation(solution) < lowest_eval:
                            #update solution here
                            lowest_eval = evaluation(solution)
    return {"solution":best_solution, "evaluation":lowest_eval}
```

1 a)

In [50]:
```
def schwefel(x_vector):
    return 418.9892 * len(x_vector) - np.sum(x_vector * np.sin(np.sqrt(np.ab

solution = np.random.uniform(-500, 500, 10)
# evaluation = schwefel function
delta = 0.5
boundary = [-500, 500]
TSA = 3000
alpha = 0.5
low_temp_30 = 30
low_temp_10 = 10
len_cool_30 = (TSA - low_temp_30) / alpha
len_cool_10 = (TSA - low_temp_10) / alpha
cooling_schedule_30 = np.linspace(start=TSA, stop=low_temp_30, num=int(len_c

print(f"The length of the cooling schedule for 30K is : {len_cool_30}")
print(f"The length of the cooling schedule for 10K is : {len_cool_10}")
```

```
The length of the cooling schedule for 30K is : 5940.0
The length of the cooling schedule for 10K is : 5980.0
```

In [47]:
```
result = SA(solution, schwefel, delta, boundary, cooling_schedule_30)
print(result['evaluation'])
```

```
4590.757346257131
```

In [49]:
```
result = SA(solution, schwefel, delta, boundary, cooling_schedule_30)
print(result['evaluation'])
```

```
4431.18570896581
```

In [51]:
```
result = SA(solution, schwefel, delta, boundary, cooling_schedule_30)
print(result['evaluation'])
```

```
4967.847171138068
```

The three results for the cooling schedule to 30K are shown above. The three results for the cooling schedule to 10K are shown below

In [64]:
```
solution = np.random.uniform(-500, 500, 10)
```

```
cooling_schedule_10 = np.linspace(start=TSA, stop=low_temp_10, num=int(len_c
```

In [61]:
```
result = SA(solution, schwefel, delta, boundary, cooling_schedule_10)
print(result['evaluation'])
```

```
4577.335779571215
```

In [63]:
```
result = SA(solution, schwefel, delta, boundary, cooling_schedule_30)
print(result['evaluation'])
```

```
4173.545578788456
```

In [65]:
```
result = SA(solution, schwefel, delta, boundary, cooling_schedule_30)
print(result['evaluation'])
```

```
3783.1700213887843
```

The results do look better when cooling to the lower temperature, as the global minima evaluation decreases (closer to the global minimum)

1b)

In [79]:
```
TSA_3k = 3000
TSA_6k = 6000
sigma = 1000
k = 6000

def create_logspace_cooling(TSA):
    # [TSA / (1 + TSA * np.log(1 + i)/(3*sigma) for i in range(1, k+1)]
    return np.array([TSA / (1 + TSA * np.log(1 + i)/(3*sigma) ) for i in rar

cooling_3k = create_logspace_cooling(TSA_3k)
cooling_6k = create_logspace_cooling(TSA_6k)

print(f"The length of the log cooling schedule for 3000K is : {len(cooling_3
print(f"The length of the log cooling schedule for 6000K is : {len(cooling_6

solution = np.random.uniform(-500, 500, 10)
result = SA(solution, schwefel, delta, boundary, cooling_3k)
print(result['evaluation'])
```

```
The length of the log cooling schedule for 3000K is : 6000
The length of the log cooling schedule for 6000K is : 6000
3530.1512465460246
```

In [74]:
```
solution = np.random.uniform(-500, 500, 10)
result = SA(solution, schwefel, delta, boundary, cooling_3k)
print(result['evaluation'])
```

```
4199.608272639098
```

In [75]:
```
solution = np.random.uniform(-500, 500, 10)
result = SA(solution, schwefel, delta, boundary, cooling_3k)
print(result['evaluation'])
```

```
3450.8069313035544
```

In [76]:
```
solution = np.random.uniform(-500, 500, 10)
result = SA(solution, schwefel, delta, boundary, cooling_6k)
print(result['evaluation'])
```

```
4349.086270974157
```

In [77]:
```
solution = np.random.uniform(-500, 500, 10)
result = SA(solution, schwefel, delta, boundary, cooling_6k)
print(result['evaluation'])
```

```
3589.046116217409
```

In [78]:
```
solution = np.random.uniform(-500, 500, 10)
result = SA(solution, schwefel, delta, boundary, cooling_6k)
print(result['evaluation'])
```

```
2968.863431869962
```

For both 3k and 6k, the cooling schedules converge better than linear cooling as they find a lower global minimum on average

1 c)

In [95]:
```
geometric_cooling_3k = np.array([(alpha ** n) * 3000 for n in range(0, 10)])
geometric_cooling_3k
```

Out[95]:
```
array([3000.      , 1500.      ,  750.      ,  375.      ,  187.5     ,
         93.75    ,   46.875   ,   23.4375  ,   11.71875 ,    5.859375])
```

In [97]:
```
solution = np.random.uniform(-500, 500, 10)
result = SA(solution, schwefel, delta, boundary, geometric_cooling_3k)
print(result['evaluation'])
```

```
4437.716582098562
```

In [98]:
```
solution = np.random.uniform(-500, 500, 10)
result = SA(solution, schwefel, delta, boundary, geometric_cooling_3k)
print(result['evaluation'])
```

```
5181.237499138457
```

In [99]:
```
solution = np.random.uniform(-500, 500, 10)
result = SA(solution, schwefel, delta, boundary, geometric_cooling_3k)
print(result['evaluation'])
```

```
4698.450759403982
```

I implemented a Geometric annealing schedule, and it appears that this annealing schedule did not lead to increased convergence (the evaluations of the function on average were higher than when I used logarithmic or linear cooling schedules).

In [88]:
```
from scipy.optimize import minimize
```

```
def save_step(*args):
    for arg in args:
        if type(arg) is np.ndarray:
            steps.append(arg)

def minimize_function(x0, func, method):
    """
    Minimize a function

    Parameters
    ----------
    x0: np.ndarray
        Starting point
    func: function
        Scalar function to minimize
    method: str
        Method for minimization

    Returns
    -------
    res: OptimzizeResult
        Result object of scipy optimization
    """
    res = minimize(
        func,
        x0,
        method=method,
        options={"gtol": 1e-5, "disp":True},
        callback=save_step,
    )
    return res
```

Performing a Conjugate Gradient local optimization technique on my solution using a geometric cooling schedule

In [104…

```
steps = []
res = minimize_function(result['solution'], schwefel, "CG")
print(res.x)
```

```
         Current function value: 2468.209950
         Iterations: 19
         Function evaluations: 638
         Gradient evaluations: 58
[ 203.81423992  203.81425178 -124.82936228 -124.82937266  -25.87741651
 -124.82933152   65.54786231    5.23919631 -559.14867017 -302.52492513]
```

```
/Users/chu/miniconda3/envs/chem277b/lib/python3.10/site-packages/scipy/optim
ize/_minimize.py:706: OptimizeWarning: Desired error not necessarily achieve
d due to precision loss.
  res = _minimize_cg(fun, x0, args, jac, callback, **options)
```

After performing local optimization (conjugate gradient), an even better solution was found with a function evaluation closer to 0. The function evaluation converged more closely, dropping from 4698.450759403982 to 2468.209950, so it improved!

2 a)

Encoding A

Solution 3 - Vector [1000]

Solution 4 - Vector [0010]

Solution 5 - Vector [0001]

Schema: `[*0**]` Order = 1, Length: 2-2 = 0

Encoding B

Solution 3 - Vector [1101]

Solution 4 - Vector [1011]

Solution 5 - Vector [1111]

Schema: `[1**1]` Order = 2, Length: 4 -1 = 3

We will choose Encoding A because for genetic algorithms, we want schema with low order and low length. This is because operations like crossing-over will often disrupt schema with high orders or length, preventing those positive traits from being retained. Instead we want low order and low length schema to decrease the chance of those positive encodings to be disrupted by crossing over

In [276…
```
def one_point_crossover(vec1, vec2, point):
    new_vec1 = vec1[:point] + vec2[point:]
    new_vec2 = vec2[:point] + vec1[point:]
    return new_vec1, new_vec2

def mutate(vec, point):
    binaryAtPoint = vec[point]
    if binaryAtPoint == 0:
        return vec[:point] + '1' + vec[point + 1:]
    else:
        return vec[:point] + '0' + vec[point + 1:]

def two_point_crossover(vec1, vec2, point1, point2):
    new_vec1 = vec1[:point1] + vec2[point1:point2] + vec1[point2:]
    new_vec2 = vec2[:point1] + vec1[point1:point2] + vec2[point2:]
    return new_vec1, new_vec2


def test_one_point_crossover():
    c1, c2 = one_point_crossover("0000", "1111", 1)
    if {c1, c2} == {"1000", "0111"}:
        print("Well done!")
```

```
    else:
        raise Exception("Wrong implementation")


def test_mutate():
    if "0000" == mutate("0100", 1):
        print("Well done")
    else:
        raise Exception("Wrong implementation")


def test_two_point_crossover():
    c1, c2 = two_point_crossover("0000", "1111",1,3)
    if {c1, c2} == {"0110", "1001"}:
        print("Well done")
    else:
        raise Exception("Wrong implementation")

test_one_point_crossover()
test_mutate()
test_two_point_crossover()
```

```
Well done!
Well done
Well done
```

Population evaluation code Q2

In [270…

```
# Complete the following code to get the below output

def fitness(x):
    return -(x**2) + 8 * x + 15

def evaluate_population(pop):
    # sort polpulation by their fitness
    pop_copy = pop.copy()
    pop_copy.sort(key = fitness,reverse=True)

    print("Solutions    Vector      Fitness")
    for sol in pop_copy:
        print("%-13d%-13s%-6d"%(sol,vector_dict[sol],fitness(sol)))
    print("="*20)
    print("Total fitness:",sum([fitness(sol) for sol in pop_copy]))
    print(f'Best solution: {pop_copy[0]} with fitness {fitness(pop_copy[0])}

# decoding based on encoding A: vector -> solution
solution_dict = {
    "1011": 0, "0011": 1, "1001": 2, "1000": 3,
    "0010": 4, "0001": 5, "0000": 6, "1010": 7,
    "0100": 8, "1100": 9, "0101":10, "0110":11,
    "0111":12, "1101":13, "1110":14, "1111":15,
}

# encoding A: solution -> vector
vector_dict = dict(zip(solution_dict.values(),solution_dict.keys()))
```

```
# initial population
pop = [4,7,13,10]
evaluate_population(pop)
```

```
Solutions     Vector      Fitness
4             0010         31
7             1010         22
10            0101         -5
13            1101         -50
====================
Total fitness: -2
Best solution: 4 with fitness 31
```

In [271…

```
# you should get this
```

2 b)

In [272…

```
pop = [10, 1, 15, 6, 0 , 9]
pop.sort(key=fitness)
pop
```

Out[272…

```
[15, 10, 9, 0, 1, 6]
```

In [273…

```
evaluate_population(pop)
```

```
Solutions     Vector      Fitness
6             0000         27
1             0011         22
0             1011         15
9             1100         6
10            0101         -5
15            1111         -90
====================
Total fitness: -25
Best solution: 6 with fitness 27
```

1. x = 10, Vector: [0101], Fitness: -5
2. x = 1, Vector: [0011], Fitness: 22
3. x = 15, Vector: [1111], Fitness: -90
4. x = 6, Vector: [0000], Fitness: 27
5. x = 0, Vector: [1011], Fitness: 15
6. x = 9, Vector: [1100], Fitness: 6

Pair 1: x = 15 and x = 6

Pair 2: x = 10 and x = 1

Pair 3: x = 0 and x = 9

In [274…

```
def generate_pairs(population):
    #Takes a sorted list and returns pairings of least-best fit solutions
    pairings = []
```

```
    for i in range(int(len(population)/2)):
        low = population[i]
        high = population[len(population) - i - 1]
        pair = (low, high)
        pairings.append(pair)
    return pairings

b_pairs = generate_pairs(pop)
print(b_pairs)
```

```
[(15, 6), (10, 1), (9, 0)]
```

2 c)

In [277…

```
pair1_1_vector, pair1_2_vector = one_point_crossover("0000", "1111", 1)

new_sol_1 = solution_dict[pair1_1]
new_sol_2 = solution_dict[pair1_2]

pair1 = [new_sol_1, new_sol_2]
pair1_vectors = [pair1_1_vector, pair1_2_vector]
print(pair1)

print(f"Solution x=6 is now {new_sol_1} with fitness {fitness(new_sol_1)}")
print(f"Solution x=15 is now {new_sol_2} with fitness {fitness(new_sol_2)}")
```

```
[12, 3]
Solution x=6 is now 12 with fitness -33
Solution x=15 is now 3 with fitness 30
```

In [278…

```
pair2_1, pair2_2 = one_point_crossover("0011", "0101", 1)

new_sol_1 = solution_dict[pair2_1]
new_sol_2 = solution_dict[pair2_2]

pair2 = [new_sol_1, new_sol_2]
pair2_vectors = [pair2_1, pair2_2]
print(pair2)

print(f"Solution x=1 is now {new_sol_1} with fitness {fitness(new_sol_1)}")
print(f"Solution x=10 is now {new_sol_2} with fitness {fitness(new_sol_2)}")
```

```
[10, 1]
Solution x=1 is now 10 with fitness -5
Solution x=10 is now 1 with fitness 22
```

In [250…

```
pair3_1, pair3_2 = one_point_crossover("1011", "1100", 1)

new_sol_1 = solution_dict[pair3_1]
new_sol_2 = solution_dict[pair3_2]

pair3 = [new_sol_1, new_sol_2]
pair3_vectors = [pair3_1, pair3_2]
print(pair3)
```

```
print(f"Solution x=0 is now {new_sol_1} with fitness {fitness(new_sol_1)}")
print(f"Solution x=9 is now {new_sol_2} with fitness {fitness(new_sol_2)}")
```

```
[9, 0]
Solution x=0 is now 9 with fitness 6
Solution x=9 is now 0 with fitness 15
```

In [279…

```
c_pop_vectors = [pair1_1, pair1_2, pair2_1, pair2_2, pair3_1, pair3_2]
print("Population vectors from c) is: \n", c_pop_vectors)
```

```
Population vectors from c) is: 
 ['0111', '1000', '0101', '0011', '1100', '1011']
```

In [280…

```
c_pop_solutions = [solution_dict[vector] for vector in c_pop_vectors]
print("Population solutions from c) is: \n", c_pop_solutions)
```

```
Population solutions from c) is: 
 [12, 3, 10, 1, 9, 0]
```

In [281…

```
evaluate_population(c_pop_solutions)
```

```
Solutions     Vector      Fitness
3             1000         30
1             0011         22
0             1011         15
9             1100         6
10            0101         -5
12            0111         -33
====================
Total fitness: 35
Best solution: 3 with fitness 30
```

For pair 1, (x=15, x=6), one point crossover at position 1 created two new vectors, "0111" and "1000". These corresponded to the solutions 12 with fitness -33, and solution 3 with fitness 30.

For pair 2, (x=1, x=10), the one point crossover just resulted in them swapping, so (x=10, x=1). No new strings/solutions were created.

Finally, for pair 3, (x=0, x=9), the one point crossover changed x=0 into solution 9 with fitness 6, and changed x=9 into solution 0 with fitness 15; once again, no new vectors or solutions were created.

The fitness of the population did increase as a whole. The total fitness after crossing over was 35, while the previous population had a total fitness of -25.

The best solution also changed to be x=3 with a fitness of 30

2 d)

New pairs are generated from population c), pairing the best fit and least fit solutions:

In [282…

```
# Sort c_pop_solutions
c_pop_solutions.sort(key=fitness)
d_pairs = generate_pairs(c_pop_solutions)
print(d_pairs)
```

```
[(12, 3), (10, 1), (9, 0)]
```

In [283…

```
d_pop_vectors = []

for vector in c_pop_vectors:
    d_pop_vectors.append(mutate(vector, 3))

print("Population d) vectors after mutating position 3 from population c)  i
```

```
Population d) vectors after mutating position 3 from population c)  is :
 ['0110', '1000', '0100', '0010', '1100', '1010']
```

In [284…

```
# d_pair1_vectors = []
# d_pair2_vectors = []
# d_pair3_vectors = []

# d_pair1_solutions = []
# d_pair2_solutions = []
# d_pair3_solutions = []

# for vector in pair1_vectors:
#     mutated_vector = mutate(vector, 3)
#     d_pair1_vectors.append(mutated_vector)
#     d_pair1_solutions.append(solution_dict[mutated_vector])

# for vector in pair2_vectors:
#     mutated_vector = mutate(vector, 3)
#     d_pair2_vectors.append(mutated_vector)
#     d_pair2_solutions.append(solution_dict[mutated_vector])

# for vector in pair3_vectors:
#     mutated_vector = mutate(vector, 3)
#     d_pair3_vectors.append(mutated_vector)
#     d_pair3_solutions.append(solution_dict[mutated_vector])

# print("New pair 1 for d): ", d_pair1_solutions, d_pair1_vectors)
# print("New pair 2 for d): ", d_pair2_solutions, d_pair2_vectors)
# print("New pair 3 for d): ", d_pair3_solutions, d_pair3_vectors)
```

In [285…

```
d_pop_solutions = [solution_dict[vector] for vector in d_pop_vectors]
print("Population solutions from d) is: \n", d_pop_solutions)
print("Population solutions from b) is: \n", pop)
print("Population solutions from c) is: \n", c_pop_solutions)
```

```
Population solutions from d) is:
 [11, 3, 8, 4, 9, 7]
Population solutions from b) is:
 [15, 10, 9, 0, 1, 6]
Population solutions from c) is:
 [12, 10, 9, 0, 1, 3]
```

In [286…
```
evaluate_population(d_pop_solutions)
```

```
Solutions     Vector      Fitness
4             0010         31
3             1000         30
7             1010         22
8             0100         15
9             1100         6
11            0110         -18
====================
Total fitness: 86
Best solution: 4 with fitness 31
```

We have new solutions after mutating population C. The new solutions are 11, 8, 4, and 7. These are solutions that have not appeared before in either Population C) or the original Population from B).

The fitness of solution 11 is -18. The fitness of solution 8 is 15. The fitness of solution 4 is 31, and the fitness of solution 7 is 22.

Mutation increased the total fitness of the population, as the total fitness is now 86. Mutation also found a better solution, as the best solution is now 4 with a fitness of 31, as opposed to the previous best fitness of 30.

Pairs were not really important in this step because only mutations occurred (pairs only relevant for crossing over)

2 e)

In [287…
```
# Sort the population from d) in ascending order by fitness

d_pop_solutions.sort(key=fitness)
d_pop_solutions
```

Out[287…
```
[11, 9, 8, 7, 3, 4]
```

In [288…
```
# Replace the least fit member with a clone of the best fit member
e_pop_solutions = d_pop_solutions.copy()
e_pop_solutions[0] = e_pop_solutions[len(e_pop_solutions)-1]
e_pop_solutions
```

Out[288…
```
[4, 9, 8, 7, 3, 4]
```

In [289…
```
# Generate pairs of best-fit and least-fit solutions
e_pop_solutions.sort(key=fitness)
print(e_pop_solutions)
e_pairs = generate_pairs(e_pop_solutions)
e_pairs
```

```
[9, 8, 7, 3, 4, 4]
```

Out[289… [(9, 4), (8, 4), (7, 3)]

In [300…
```
# Perform 2 point cross over for each pair, exchanging the inner two element
# point1 = 1, point2 = 3

e_pop_vectors = []
e_pop_solutions = []

for pair in e_pairs:
    print("Old Pair is: ", pair)
    vec1 = vector_dict[pair[0]]
    vec2 = vector_dict[pair[1]]
    new_vec1, new_vec2 = two_point_crossover(vec1, vec2, 1, 3)
    e_pop_vectors.append(new_vec1)
    e_pop_vectors.append(new_vec2)

    solution1 = solution_dict[new_vec1]
    solution2 = solution_dict[new_vec2]
    e_pop_solutions.append(solution1)
    e_pop_solutions.append(solution2)
    print(f"New Pair is now: ({solution1}, {solution2})")

print("Population e) vectors after two point crossing over of population d):
```

```
Old Pair is:  (9, 4)
New Pair is now: (7, 8)
Old Pair is:  (8, 4)
New Pair is now: (4, 8)
Old Pair is:  (7, 3)
New Pair is now: (3, 7)
Population e) vectors after two point crossing over of population d):
 ['1010', '0100', '0010', '0100', '1000', '1010']
```

In [302…
```
print("Population solutions from b) is: \n", pop)
print("Population solutions from c) is: \n", c_pop_solutions)
print("Population solutions from d) is: \n", d_pop_solutions, "\n")
print("Population solutions from e) is: \n", e_pop_solutions)
```

```
Population solutions from b) is:
 [15, 10, 9, 0, 1, 6]
Population solutions from c) is:
 [12, 10, 9, 0, 1, 3]
Population solutions from d) is:
 [11, 9, 8, 7, 3, 4]

Population solutions from e) is:
 [7, 8, 4, 8, 3, 7]
```

In [303…
```
evaluate_population(e_pop_solutions)
```

```
Solutions    Vector      Fitness
4            0010         31
3            1000         30
7            1010         22
7            1010         22
8            0100         15
8            0100         15
====================
Total fitness: 135
Best solution: 4 with fitness 31
```

We don't have any brand new solutions, but after two point crossing over, pair (9, 4) has been replaced by pair (7, 8). The other stayed the same, but swapped places. We have increased the total population fitness, as it is now 135 as opposed to the previous 86. We have already found the best solution of 4 with fitness 31

2 f)

In [293…

```python
# Sort the population from e) in ascending order by fitness

e_pop_solutions.sort(key=fitness)
e_pop_solutions
```

Out[293…

```
[8, 8, 7, 7, 3, 4]
```

In [294…

```python
# Replace the least fit member with a clone of the best fit member
f_pop_solutions = e_pop_solutions.copy()
f_pop_solutions[0] = f_pop_solutions[len(f_pop_solutions)-1]
f_pop_solutions
```

Out[294…

```
[4, 8, 7, 7, 3, 4]
```

In [295…

```python
# Generate pairs of best-fit and least-fit solutions
f_pop_solutions.sort(key=fitness)
print(f_pop_solutions)
f_pairs = generate_pairs(f_pop_solutions)
f_pairs
```

```
[8, 7, 7, 3, 4, 4]
```

Out[295…

```
[(8, 4), (7, 4), (7, 3)]
```

In [298…

```python
# Perform one point cross over for each pair, between the 3rd and 4th elemer
# and exchange the first 3 elements of each pair

f_pop_vectors = []
f_pop_solutions = []

for pair in f_pairs:
    print("Old Pair is: ", pair)
    vec1 = vector_dict[pair[0]]
    vec2 = vector_dict[pair[1]]
    new_vec1, new_vec2 = one_point_crossover(vec1, vec2, 3)
```

```
    f_pop_vectors.append(new_vec1)
    f_pop_vectors.append(new_vec2)

    solution1 = solution_dict[new_vec1]
    solution2 = solution_dict[new_vec2]
    f_pop_solutions.append(solution1)
    f_pop_solutions.append(solution2)
    print(f"New Pair is now: ({solution1}, {solution2})")

print("Population f) vectors after one point crossing over between 3rd and 4

# Population d) vectors ['1010', '0100', '0010', '0100', '1000', '1010']
```

```
Old Pair is:  (8, 4)
New Pair is now: (8, 4)
Old Pair is:  (7, 4)
New Pair is now: (7, 4)
Old Pair is:  (7, 3)
New Pair is now: (7, 3)
Population f) vectors after one point crossing over between 3rd and 4th elem
ent of population e):
 ['1010', '0100', '0010', '0100', '1000', '1010']
```

In [299…

```
print("Population solutions from b) is: \n", pop)
print("Population solutions from c) is: \n", c_pop_solutions)
print("Population solutions from d) is: \n", d_pop_solutions)
print("Population solutions from e) is: \n", e_pop_solutions, "\n")
print("Population solutions from f) is: \n", f_pop_solutions)
```

```
Population solutions from b) is:
 [15, 10, 9, 0, 1, 6]
Population solutions from c) is:
 [12, 10, 9, 0, 1, 3]
Population solutions from d) is:
 [11, 9, 8, 7, 3, 4]
Population solutions from e) is:
 [8, 8, 7, 7, 3, 4]

Population solutions from f) is:
 [8, 4, 7, 4, 7, 3]
```

In [305…

```
evaluate_population(f_pop_solutions)
```

```
Solutions     Vector      Fitness
4             0010         31
4             0010         31
3             1000         30
7             1010         22
7             1010         22
8             0100         15
====================
Total fitness: 151
Best solution: 4 with fitness 31
```

No brand new solutions were generated, and no new solutions were generated from crossing-over. However, deleting the least fit solution 8 and replacing it with

another 4 increased the total population fitness from 135 to 141. We already found the best solution of 4 with fitness 31, so we did not improve upon that.

2 g)

The encoding of the solution space was adequate. This is because crossing-over did not tend to lose the positive characteristics of the most fit solutions, while also introducing diversity into lesser fit solutions to make them more fit. A great example of this is in *f)*, where crossing over between the 3rd and 4th elements did not cause any of the already very fit solutions to be lost, but instead simply swapped their positions in the pair. Thus, this encoding is adequate for Genetic Algorithms, as it allows the introduction of diversity for lesser fit solutions while retaining the best fitness solutions. Eventually the lesser fit solutions are competed out by the best fitness solutions, ala Darwin's theory of natural selection.