# Question 1

The D-dimensional Schwefel function:

$$f(x_1, x_2, \cdots, x_D) = 418.9829D - \sum_i^D x_i \sin(\sqrt{|x_i|})$$

where $x_i \in [-500, 500]$ for $i = 1, 2, \cdots, D$.

*For debugging*: The global minimum is 0, which is reached at $x_i = 420.9687$

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

In [ ]:
```
def Schwefel(X):
    D = len(X)
    sum = np.sum(X * np.sin(np.sqrt(np.abs(X))))
    return 418.9829 * D - sum

# for visualization if you want
def plot_surface(func, x_min=-2, x_max=2, y_min=-2, y_max=2):
    a = np.linspace(x_min, x_max, 100)
    b = np.linspace(y_min, y_max, 100)
    x,y = np.meshgrid(a, b)
    z = func((x, y))
    fig = plt.figure()
    ax = fig.add_subplot(projection='3d')
    ax.plot_surface(x, y, z)

print(Schwefel([420.9687]))
```

```
1.272783748618167e-05
```

In [ ]:
```
def SA(solution, func, schedule, delta, boundary, n_iter=10, report_interval
    """
    Simulated Annealing for minimization

    Parameters
    ----------
    solution: np.ndarray
        Initial guess
    func: Callable
        Function to minimize
    schedule: np.ndarray
        An array of temperatures for simulated annealing
    delta: float
        Magnitude of random displacement
    boundary: tuple
        Boundary of the variables to minimize. (lowerbound,upperbound)
    n_iter: int
        Number of random displacement move in each temperature
```

```
    report_interavl: int
        Number of temperature steps to report result

    Returns
    --------
    res: dict
        Minimized point and its evaulation value
    """

    best_solution = solution.copy()
    lowest_eval = func(best_solution)

    for idx, temp in enumerate(schedule):
        if report_interval is not None and ((idx + 1) % report_interval == 0
            msg = (
                f"{idx + 1}/{len(schedule)}, Temp: {temp:.2f}, "
                f"Best solution: {best_solution}, Value: {lowest_eval:.7f}"
            )
            print(msg)

        for n in range(n_iter):
            trial = solution.copy()
            # do a random displacement
            urn = np.random.uniform(size=len(solution))
            trial += (2 * urn - 1) * delta
            if np.all(trial >= boundary[0]) and np.all(trial <= boundary[1])
                # fill in acceptance criterion
                energy = func(trial) - func(solution)
                if energy < 0 or np.random.uniform() < np.exp(-1 / temp * er
                    solution = trial
                    if func(solution) < lowest_eval:
                        # update solution here
                        best_solution = solution.copy()
                        lowest_eval = func(best_solution)
                        assert np.array_equal(solution, best_solution)

    return {"solution":best_solution, "evaluation":lowest_eval}
```

In [ ]:
```
starting = 500 * (np.random.random(10) * 2 - 1)
#print(SA(starting, func=Schwefel, schedule=np.arange(3000, 30-0.5, -0.5), c
```

# (a)

*For debugging*:

Length of schedule 5941 for 30K, 5981 for 10K (both initial temperature and final temperature are included in the schedule). The function evaluation of your solution usually falls in the range of 2000~4000 with `delta=0.5` and `n_iter=10`.

In [ ]:
```
def linear_cooling(init_temp, final_temp, alpha):
    #starting = 500 * (np.random.random(10) * 2 - 1)
    return SA(starting, func=Schwefel, schedule=np.arange(init_temp, final_t
              boundary=[-500, 500], report_interval=None)
```

In [ ]:
```
# Print average & standard deviation of minimized values of 3 runs
res1 = []
res2 = []
for i in range(3):
    res1.append(linear_cooling(3000, 30, 0.5))
    res2.append(linear_cooling(3000, 10, 0.5))
print(f"Minimized values at 30K: {np.mean([j['evaluation'] for j in res1])}
print(f"Minimized values at 10K: {np.mean([j['evaluation'] for j in res2])}

print("length of schedule (30k):", len(np.arange(3000, 30-0.5, -0.5)))
print("length of schedule (10k):", len(np.arange(3000, 10-0.5, -0.5)))
```

```
Minimized values at 30K: 3469.47271173174 +/- 488.46507913959505
Minimized values at 10K: 2734.1306068545723 +/- 384.48488578277437
length of schedule (30k): 5941
length of schedule (10k): 5981
```

## (a) The lower temperature has a better solution than the higher temperature. It seems like it converged closer to the minimum.

## (b)

*For debugging*:

The final temperature should be 326.10415680714726 (starting from 6000K) or 309.29382323518576 (starting from 3000K).

In [ ]:
```
def log_cooling(init_temp, sigma, k):
    sched = [init_temp]
    for i in range(1, k+1):
        sched.append(init_temp / (1 + init_temp*np.log(1 + i) / (3 * sigma )
    print(f"The final temperature of schedule, with initial temperature of {
    return SA(solution=starting, func=Schwefel, schedule=sched, delta=0.5, b
```

In [ ]:
```
# Print average & standard deviation of minimized values of 3 runs
hot = []
cold = []
for i in range(3):
    hot.append(log_cooling(6000, sigma=1000, k=6000))
    cold.append(log_cooling(3000, 1000, 6000))
print(f"Minimized values starting from 6000K: {np.mean([i['evaluation'] for
print(f"Minimized values starting from 3000K: {np.mean([i['evaluation'] for
```

```
The final temperature of schedule, with initial temperature of 6000 is 326.0
982494108243
The final temperature of schedule, with initial temperature of 3000 is 309.2
885091765957
The final temperature of schedule, with initial temperature of 6000 is 326.0
982494108243
The final temperature of schedule, with initial temperature of 3000 is 309.2
885091765957
The final temperature of schedule, with initial temperature of 6000 is 326.0
982494108243
The final temperature of schedule, with initial temperature of 3000 is 309.2
885091765957
Minimized values starting from 6000K: 2897.8650074372345 +/- 470.19887126989
14
Minimized values starting from 3000K: 2675.431549454165 +/- 211.434628812265
15
```

## Do these cooling schedules converge better than linear cooling?

In general, the logarithmic cooling minimized better than the linear cooling. The linear cooling converged on values around ~3000, but the logarithmic cooling was around 2500-2800. However, from a mathematical standpoint, we know that logarithmic cooling is more likely to converge than linear cooling, but logarithmic cooling will take much longer because it is exponential.

## (c)

In [ ]:
```python
# construct your cooling schedule
def custom_cooling(init_temp, final_temp):
    schedule = []
    cooling = True
    segment_size = 1000
    curr_temp = init_temp
    while curr_temp > 0:
        if cooling:
            decreasing = np.arange(curr_temp, curr_temp - segment_size, -1)
            schedule = np.concatenate((schedule, decreasing))
            curr_temp = curr_temp - segment_size
            cooling = False
        else:
            increasing = np.arange(curr_temp, curr_temp + segment_size / 2)
            schedule = np.concatenate((schedule, increasing))

            curr_temp = curr_temp + segment_size / 2
            cooling = True
    return SA(solution=starting, func=Schwefel, schedule=schedule, delta=0.5
```

In [ ]:
```python
# Print average & standard deviation of minimized values of 3 runs
hot = []
cold = []
for _ in range(3):
```

```
    hot.append(custom_cooling(6000, 1))
    cold.append(custom_cooling(3000, 1))
print(f"Minimized values starting from 6000K: {np.mean([i['evaluation'] for
print(f"Minimized values starting from 3000K: {np.mean([i['evaluation'] for
```

```
Minimized values starting from 6000K: 2368.1970597160102 +/- 91.082182575625
21
Minimized values starting from 3000K: 2646.2211656741933 +/- 703.36756075849
73
```

### Can you find an even better solution?

What I did was essentially linear, but with extra steps, in hopes that I would find a better minimum by preventing cooling too fast. My algorithm seemed to minimize better than the straightforward linear cooling and the log cooling.

# Question 2

## (a) In encoding a, the best solutions with f(x) > 27 are 3, 4, 5.

```
3 30 1000
4 31 0010
5 30 0001

The best shared schema for Encoding A is *0**.  The length is
0.  The order is 1.
```

## In encoding b, the best solutions are 3,4,5.

```
3 30 1101
4 31 1011
5 30 1111

The best shared schema for Encoding B 1**1.  The length is 3.
The order is 2.
```

## Based on these results, I will be using Encoding A because it has shorter length and small order.

## (b)

In [ ]:
```
import pandas as pd
```

```
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
}


def func(vec):
    x = solution_dict[vec]
    return -x ** 2 + 8 * x + 15


def one_point_crossover(parent1, parent2, point):
    point -= 1
    return (parent1[:point] + parent2[point] + parent1[point + 1:], parent2[

def evaluate_population(pop):
    df = pd.DataFrame({
        "Solutions": [solution_dict[vec] for vec in pop],
        "Vectors": pop,
        "Fitness": [func(vec) for vec in pop]
    })
    df.sort_values(by=["Fitness"], ascending=False, inplace=True)
    df.reset_index(inplace=True, drop=True)
    print(f"Total Fitness: {np.sum(df['Fitness'])}")
    print(f"Best Solution: {df.loc[0, 'Solutions']} (with fitness {df.loc[0,
    return df
```

*For debugging*: Use the following function to test `one_point_crossover()`

In [ ]:
```
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")

test_one_point_crossover()
```

```
Well done!
```

In [ ]:
```
pop_eval = evaluate_population(["0101", "0011", "1111", "0000", "1011", "110
pop_eval = [list(row) for row in [pop_eval[i] for i in range(len(pop_eval)//
print(pop_eval)
```

```
Total Fitness: -25
Best Solution: 6 (with fitness 27)
[[6, '0000', 27], [15, '1111', -90], [1, '0011', 22], [10, '0101', -5], [0,
'1011', 15], [9, '1100', 6]]
```

*Hint*: Use `list.sort(key=...)` to sort a list of population according to its evaluation value. Maybe you can find this useful.

## (c)

In [ ]:
```
new_population = []
for i in range(0, len(pop_eval), 2):
    p1 = pop_eval[i][1]
    p2 = pop_eval[i+1][1]
    new_solution = one_point_crossover(p1, p2, point=1)
    new_population.append(new_solution[0])
    new_population.append(new_solution[1])
print(evaluate_population(new_population))
```

```
Total Fitness: 35
Best Solution: 3 (with fitness 30)
   Solutions Vectors  Fitness
0          3    1000       30
1          1    0011       22
2          0    1011       15
3          9    1100        6
4         10    0101       -5
5         12    0111      -33
```

(c) Refer to print statement above to see new solutions. Their fitness is 35, which is better than before. The best solution is vector 0001 with fitness of 30, and vector 1000 with fitness 30. The best solution here is better than the best solution from before.

# (d)

In [ ]:
```
def mutate(vec, point):
    vector = ""
    if vec[point] == "1":
        vector = vec[:point] + "0" + vec[point+1:]
    else:
        vector = vec[:point] + "1" + vec[point+1:]
    return vector
```

In [ ]:
```
mutated_population = new_population.copy()
for i in range(len(mutated_population)):
    mutated_population[i] = mutate(mutated_population[i], 2)
mutation_eval = evaluate_population(mutated_population)
print(mutation_eval)
```

```
Total Fitness: -28
Best Solution: 5 (with fitness 30)
   Solutions Vectors  Fitness
0          5    0001       30
1          2    1001       27
2          7    1010       22
3         10    0101       -5
4         12    0111      -33
5         14    1110      -69
```

*For debugging*: Use the following function to test `mutate()`

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

test_mutate()
```

```
Well done
```

## (d) We have new solutions (see print output above). The fitness is -28. Mutations do not guarantee an increase in the fitness of a population, but there is certainly a chance of it happening.

# (e)

In [ ]:
```
def two_point_crossover(parent1, parent2):
    p1 = parent1[:1] + parent2[1:3] + parent1[3:]
    p2 = parent2[:1] + parent1[1:3] + parent2[3:]
    return (p1, p2)
```

*For debugging*: Use the following function to test `two_point_crossover()`

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

test_two_point_crossover()
```

```
Well done
```

In [ ]:
```
mutation_eval = mutation_eval.to_numpy()
mutation_eval = np.delete(mutation_eval, -1, 0) # remove last one, worst fit
mutation_eval = np.append(mutation_eval, [mutation_eval[0]], axis=0) # add b
sort_i = np.argsort(mutation_eval[:, -1])[::-1] # sort by last row, in desce
mutation_eval = mutation_eval[sort_i] # apply sort
mutation_eval = [list(row) for row in [mutation_eval[i] for i in range(len(m

two_point_population = []
for i in range(0, len(mutation_eval), 2):
    p1 = mutation_eval[i][1]
    p2 = mutation_eval[i+1][1]
    new_solution = two_point_crossover(p1, p2)
    two_point_population.append(new_solution[0])
    two_point_population.append(new_solution[1])
```

```
two_point_eval = evaluate_population(two_point_population)
print(two_point_eval)
```

```
Total Fitness: 67
Best Solution: 5 (with fitness 30)
   Solutions Vectors  Fitness
0          5    0001       30
1          5    0001       30
2          3    1000       30
3          0    1011       15
4         10    0101       -5
5         12    0111      -33
```

### (e) We have new solutions with a total fitness of 67, which is an improvement. There are more solutions with a higher fitness of 30, although the max fitness did not go up (still 30).

# (f)

In [ ]:
```
two_point_eval = two_point_eval.to_numpy()
two_point_eval = np.delete(two_point_eval, -1, 0) # remove last one, worst f
two_point_eval = np.append(two_point_eval, [two_point_eval[0]], axis=0) # ac
sort_i = np.argsort(two_point_eval[:, -1])[::-1] # sort by last row, in desc
two_point_eval = two_point_eval[sort_i] # apply sort

final_population = []

for i in range(0, len(two_point_eval), 2):
    p1 = two_point_eval[i][1]
    p2 = two_point_eval[i+1][1]
    new_solution = one_point_crossover(p1, p2, 4)
    final_population.append(new_solution[0])
    final_population.append(new_solution[1])

print(evaluate_population(final_population))
```

```
Total Fitness: 124
Best Solution: 5 (with fitness 30)
   Solutions Vectors  Fitness
0          5    0001       30
1          5    0001       30
2          6    0000       27
3          2    1001       27
4          0    1011       15
5         10    0101       -5
```

### (f) The fitness is the best it has been, with 124. However, the best solution only had a fitness of 30, which was the same as before. We did not find a fitness above 30.

(g) The encoding of the solution space was adequate. We accounted for enough solutions that we were able to find the optimal population through Darwinian elimination. Populations with short length and low order can generate a multitude of matching strings, which is what we found in Encoding A. The number of matching strings that we can derive from a low-order, low-length encoding greatly increases our chance of finding the most fit solutions.