Assignment 4

Question 1

In [1]:
```
import numpy as np
import random

def hits(N):
    black_hits = 0
    purple_hits = 0
    green_hits = 0
    blue_hits = 0
    red_hits = 0
    out_hits = 0

    for i in range(N):
        x = random.uniform(-1, 1)
        y = random.uniform(-1, 1)

        # Radius
        r = np.sqrt(x**2 + y**2)

        if r <= 0.1:
            black_hits += 1
        elif r <= 0.3:
            purple_hits += 1
        elif r <= 0.5:
            green_hits += 1
        elif r <= 0.7:
            blue_hits += 1
        elif r <= 1.0:
            red_hits += 1
        else:
            out_hits += 1

    # Probabilities
    total_hits = black_hits + purple_hits + green_hits + blue_hits + red_hits + out_hits
    black_prob = black_hits / total_hits
    purple_prob = purple_hits / total_hits
    green_prob = green_hits / total_hits
    blue_prob = blue_hits / total_hits
    red_prob = red_hits / total_hits

    return black_prob, purple_prob, green_prob, blue_prob, red_prob

# Simulation
N_sims = 20000

black_prob, purple_prob, green_prob, blue_prob, red_prob = hits(N_sims)

# Results
print("Probability of hitting each section:")
print("Black:", black_prob)
print("Purple:", purple_prob)
print("Green:", green_prob)
print("Blue:", blue_prob)
print("Red:", red_prob)
```

```
Probability of hitting each section:
Black: 0.00775
Purple: 0.06305
Green: 0.126
Blue: 0.1864
Red: 0.40665
```

Question 2

In [2]:
```
import numpy as np

#iterations and daily prob
iterations = 17000
prob_more_than_4 = 0.4

#over 4 bikes sold prob
num_bikes_probabilities = [0.4, 0.35, 0.2, 0.05]
num_bikes_sold = [5, 6, 7, 8]

#model prob
model_probabilities = [0.45, 0.35, 0.15, 0.05]
model_bonuses = [10, 15, 25, 30]

#simulation
total_bonus = 0

for _ in range(iterations):
    if np.random.rand() < prob_more_than_4:
        # Determine the number of bikes sold
        num_bikes = np.random.choice(num_bikes_sold, p=num_bikes_probabilities)

        # Calculate the bonus for each bike sold
        for _ in range(num_bikes):
            model = np.random.choice(range(4), p=model_probabilities)
            total_bonus += model_bonuses[model]

#solve bonus
expected_bonus = total_bonus / iterations
print(f"Expected Daily Bonus: ${expected_bonus:.2f}")
```

```
Expected Daily Bonus: $34.88
```

Question 3

In [3]:
```
import numpy as np

#parameters
iterations = 17000
```

```
start_age = 24
end_age = 60
years = end_age - start_age

initial_salary = 55000
contribution_rate = 0.09  #both

#salary
avg_salary_raise = 0.0283
std_salary_raise = 0.0072

#investments
investment_allocation = {
    "A": 0.50,
    "B": 0.25,
    "C": 0.25
}
investment_returns = {
    "A": {"mean": 0.0691, "std": 0.1289},
    "B": {"mean": 0.0894, "std": 0.1521},
    "C": {"mean": 0.0988, "std": 0.1714}
}

#simulation
final_balances = []

for _ in range(iterations):
    salary = initial_salary
    balance = 0

    for year in range(years):
        annual_contribution = salary * contribution_rate

        #return
        for investment, allocation in investment_allocation.items():
            contribution = annual_contribution * allocation
            annual_return = np.random.normal(investment_returns[investment]["mean"], investment_returns[investment]["std"])
            balance += contribution * (1 + annual_return)

        salary_raise = np.random.normal(avg_salary_raise, std_salary_raise)
        salary *= (1 + salary_raise)

    final_balances.append(balance)

#solve
expected_final_balance = np.mean(final_balances)
print(f"Expected 401(k) Balance at Age 60: ${expected_final_balance:.2f}")
```

```
Expected 401(k) Balance at Age 60: $327585.15
```

Question 4

In [4]:

```
#without greedy hueristic
import random

#parameters
values = [12, 16, 22, 8]
weights = [4, 5, 7, 3]
capacity = 140
max_quantity = 10
num_items = len(values)

#greedy hueristic
def greedy_heuristic(values, weights, capacity, max_quantity):
    #value/weight
    value_weight_ratio = [(values[i] / weights[i], i) for i in range(len(values))]
    #value/weight sort
    value_weight_ratio.sort(reverse=True, key=lambda x: x[0])

    total_value = 0
    total_weight = 0
    solution = [0] * num_items

    for ratio, i in value_weight_ratio:
        if total_weight < capacity:
            max_add = min(max_quantity, (capacity - total_weight) // weights[i])
            solution[i] = max_add
            total_value += max_add * values[i]
            total_weight += max_add * weights[i]

    return total_value, solution

#metahueristic
def random_restart_heuristic(values, weights, capacity, max_quantity, n_iterations):
    best_value = 0
    best_solution = None

    for _ in range(n_iterations):
        random_values = values[:]
        random.shuffle(random_values)
        total_value, solution = greedy_heuristic(random_values, weights, capacity, max_quantity)
        if total_value > best_value:
            best_value = total_value
            best_solution = solution

    return best_value, best_solution

#solve
n_iterations = 17000
best_value, best_solution = random_restart_heuristic(values, weights, capacity, max_quantity, n_iterations)
print(f"Best Value: {best_value}")
print(f"Best Solution: {best_solution}")
```

```
Best Value: 516
Best Solution: [10, 10, 2, 10]
```

In [5]:

```
import random

#parameters
values = [12, 16, 22, 8]
weights = [4, 5, 7, 3]
capacity = 140
```

```python
max_quantity = 10
num_items = len(values)

#metaheuristic
def random_restart_heuristic(values, weights, capacity, max_quantity, n_iterations):
    best_value = 0
    best_solution = None

    for _ in range(n_iterations):
        #random solution
        solution = [random.randint(0, max_quantity) for _ in range(num_items)]

        #calculate total weight
        total_weight = sum(solution[i] * weights[i] for i in range(num_items))
        total_value = sum(solution[i] * values[i] for i in range(num_items))

        if total_weight <= capacity and total_value > best_value:
            best_value = total_value
            best_solution = solution

    return best_value, best_solution

#solve
n_iterations = 17000
best_value, best_solution = random_restart_heuristic(values, weights, capacity, max_quantity, n_iterations)
print(f"Best Value: {best_value}")
print(f"Best Solution: {best_solution}")
```

```
Best Value: 438
Best Solution: [6, 10, 9, 1]
```

Question 5

In [7]:

```python
import numpy as np

# Define the objective function and the constraint
def objective_function(x):
    return 12 * x[0] + 16 * x[1] + 22 * x[2] + 8 * x[3]

def constraint(x):
    return 4 * x[0] + 5 * x[1] + 7 * x[2] + 3 * x[3]

#parameters
initial_temperature = 1000
cooling_rate = 0.95
iterations = 1000
min_temperature = 1e-5

#initial
x_current = np.random.randint(0, 11, 4)
best_solution = x_current.copy()
best_value = objective_function(x_current)

#simulated
temperature = initial_temperature
for i in range(iterations):
    # Generate a neighbor solution by modifying one of the variables
    x_new = x_current.copy()
    index = np.random.randint(0, 4)
    x_new[index] = np.clip(x_new[index] + np.random.choice([-1, 1]), 0, 10)

    # Ensure the new solution satisfies the constraint
    if constraint(x_new) <= 140:
        new_value = objective_function(x_new)

        # Calculate the acceptance probability
        if new_value > best_value:
            best_solution = x_new.copy()
            best_value = new_value
            x_current = x_new.copy()
        else:
            delta = new_value - best_value
            acceptance_probability = np.exp(delta / temperature)
            if np.random.rand() < acceptance_probability:
                x_current = x_new.copy()

    temperature = max(temperature * cooling_rate, min_temperature)

    #print
    if i % 100 == 0:
        print(f"Iteration {i}: Best Value = {best_value}, Best Solution = {best_solution}")

#solve
print(f"\nFinal Solution: {best_solution}")
print(f"Final Objective Value: {best_value}")
```

```
Iteration 0: Best Value = 156, Best Solution = [1 7 0 4]
Iteration 100: Best Value = 418, Best Solution = [10 10  3  9]
Iteration 200: Best Value = 418, Best Solution = [10 10  3  9]
Iteration 300: Best Value = 418, Best Solution = [10 10  3  9]
Iteration 400: Best Value = 418, Best Solution = [10 10  3  9]
Iteration 500: Best Value = 418, Best Solution = [10 10  3  9]
Iteration 600: Best Value = 418, Best Solution = [10 10  3  9]
Iteration 700: Best Value = 418, Best Solution = [10 10  3  9]
Iteration 800: Best Value = 418, Best Solution = [10 10  3  9]
Iteration 900: Best Value = 418, Best Solution = [10 10  3  9]

Final Solution: [10 10  3  9]
Final Objective Value: 418
```