Solution for Question 1

Here, I have the equation provided in the question, in python, it is defined as follows:

f_best=math.exp(math.cos(53*z*))+*math.sin(43*math.exp(y))+math.exp(math.sin(29*x*))+*math.sin(67*
math.cos(math.cos(37*y*))-(2*x^2+y^2+z^2)/4

The main intent of this equation was to add random values of the variable x, y and z to find the maximum value of f(x). In order to solve this, I have took 4 approaches:

1. Random search Algorithm
2. Hill Climb Algorithm
3. Simulated annealing
4. Iterated Local Search

Random Search Algorithm

Here, the algorithm is very simple. All I have to do was just enter the random values for x,y and z, and set optimum value of f(x,y,z) first. After that, for desired number of steps, repeat the process to find the maximum value of f(x,y,z)(lets say it is f(x,y,z)'''). If this value is greater than f(x,y,z), then replace the same by the new value. **Note: the values for x,y, and z should not be decreased below -1 and increased above 1.**

The code for the same is derived below:

In [75]:
```
import random as rd
import math as m
import numpy as np
import os

#setting up the random values for x, y, and z
x=rd.uniform(-1,1)
y=rd.uniform(-1,1)
z=rd.uniform(-1,1)

#setting up the best value
f_best=m.exp(m.cos(53*z))+m.sin(43*m.exp(y))+m.exp(m.sin(29*x))+m.sin(67*m.sin(z))

#printing the values of f(x,y,z) and x, y,and z
print("The best values I have found are as follows:\n")
print("f_best=",f_best)
print("\n x=",x)
print("\n y=",y)
print("\n z=",z)
```

The best values I have found are as follows:

f_best= 1.3593732759731996

x= 0.1220732913702709

y= -0.40783911224811176

z= -0.39047808418673213

In the below code snippet, I am setting the value for f_best again to test the robustness of algorithm

In [76]:
```
#Random Search Algorithm

#setting up the random values for x, y, and z
x=rd.uniform(-1,1)
y=rd.uniform(-1,1)
z=rd.uniform(-1,1)

#setting up the best value
f_best=m.exp(m.cos(53*z))+m.sin(43*m.exp(y))+m.exp(m.sin(29*x))+m.sin(67*m.sin(z))

#printing the values of f(x,y,z) and x, y,and z
print("f_best=",f_best)
print("\n x=",x)
print("\n y=",y)
print("\n z=",z)

#setting up number of steps randomly to 10000
steps=10000

#cycle of 200 steps
for i in range(steps):
    #setting up new values of x,y and z
    xnew=rd.uniform(-1,1)
    ynew=rd.uniform(-1,1)
    znew=rd.uniform(-1,1)

    #subsituting into the equation to determine the new maxima
    f_new=m.exp(m.cos(53*znew))+m.sin(43*m.exp(ynew))+m.exp(m.sin(29*xnew))+m.sin(

    #if that new maximum value is greater than previously set random values then th
    if f_new>f_best:
        f_best_1=f_new
        x_1=xnew
        y_1=ynew
        z_1=znew

#Printing the best values that has got for 200 steps
print("\n\n f_best=",f_best_1)
print("\n x=",x_1)
print("\n y=",y_1)
print("\n z=",z_1)
```

```
f_best= 3.2087617338873136

 x= -0.20686895758409896

 y= 0.6355773255091992

 z= 0.325947721817885


 f_best= 3.693932130831758

 x= 0.7976853172283238

 y= 0.7959810092295427

 z= 0.5871343897797281
```

Hill Climb Algorithm

The procedure of the hill climb algorithm is same as the procedure of random search. But instead of putting random values for every steps, I am just considering the random values at the initial stage. Later on, for every step, I am increasing the value by random values between 0 and 0.1 which is extracted from the gaussian distribution.

In [82]:
```
#hill climb algorithm
#setting up the random values at the first stage
x=rd.uniform(-1,1)
y=rd.uniform(-1,1)
z=rd.uniform(-1,1)

#Evaluating the best value of f(x,y,z) at initial stage
f_best=m.exp(m.cos(53*z))+m.sin(43*m.exp(y))+m.exp(m.sin(29*x))+m.sin(67*m.sin(z))

#printing those values
print("f_best=",f_best)
print("\n x=",x)
print("\n y=",y)
print("\n z=",z)

#initializing the number of steps
steps=10000

#repeating hill climb procedure for the number of steps
for i in range(steps):
    #increasing the values for all variables by the numbers extracted from the gaus
    x=x+rd.gauss(0,0.1)
    y=y+rd.gauss(0,0.1)
    z=z+rd.gauss(0,0.1)

    #Substituting those values in the equation
    f_new=m.exp(m.cos(53*znew))+m.sin(43*m.exp(ynew))+m.exp(m.sin(29*xnew))+m.sin(
    if f_new>f_best:
        f_best=f_new
        x=xnew
        y=ynew
        z=znew

#printing the best values
print("\n\n\n Best values after implementing hill climb algorithm:")
print("\n fbest=",f_best)
print("\n new value for x=",x)
```

```
print("\n new value for y=",y)
print("\n new value for z=",z)
```

```
f_best= 1.2947571532145905

 x= -0.6588997529835843

 y= -0.008480999961993252

 z= -0.1753817417287693


 Best values after implementing hill climb algorithm:

 fbest= 1.2947571532145905

 new value for x= 11.334171247492863

 new value for y= 12.220454763784112

 new value for z= 7.15016198399473
```

Simulated Annealing

It is also known as metropolis Algorithm. The process is same as per the above algorithms, but here, I am considering the worst case as well. Lets say even if the maximum value of the function f(x,y,z) is not greater than the best maximum value, I will consider that value keeping the probablistic value in consideration i.e.:

```
P(t,r,s) = math.exp*((f_new-f_best)/t)
```

In worst case, I am taking the random value between 0 and 1. If that value is less than P(t,r,s), then the new maximum value of f(x,y,z) will be considered.

In [78]:
```
#simulated annealing
#setting some random values in the variable
x=rd.uniform(-1,1)
y=rd.uniform(-1,1)
z=rd.uniform(-1,1)

#evaluating the best maximum value
f_best=m.exp(m.cos(53*z))+m.sin(43*m.exp(y))+m.exp(m.sin(29*x))+m.sin(67*m.sin(z))

#printing those values
print("f_best=",f_best)
print("\n x=",x)
print("\n y=",y)
print("\n z=",z)
n=rd.random()

steps=10000
#for all number of steps
for i in range(1,steps):
    #setting the value of temperature
    t=5*1-i/steps
    #setting the random values again
    x=x+rd.gauss(0,0.1)
```

```
    y=y+rd.gauss(0,0.1)
    z=z+rd.gauss(0,0.1)

    #setting the maxima
    f_new=m.exp(m.cos(53*znew))+m.sin(43*m.exp(ynew))+m.exp(m.sin(29*xnew))+m.sin(
    #checking the condition if the new maximum is greater than the old maximum
    if f_new>f_best:
            f_best=f_new
            x=xnew
            y=ynew
            z=znew

    #accepting the worse case
    elif n<m.exp((f_new-f_best)/t):
            f_best=f_new
            x_best=xnew
            y_best=ynew
            z_best=znew

#printing the best maximum values after implementation of algorithm
print("\n\n\n Best values after implementing Simulated annealing algorithm:")
print("\n fbest=",f_best)
print("\n new value for x=",x)
print("\n new value for y=",y)
print("\n new value for z=",z)
```

```
f_best= 1.3511862006780184

 x= 0.6497908009786122

 y= -0.20150905779720563

 z= -0.4312629316351999


 Best values after implementing Simulated annealing algorithm:

 fbest= 0.7343862489040611

 new value for x= 7.321586854713488

 new value for y= -15.908129061545555

 new value for z= -0.5280338205337053
```

Iterated Local search

Iterated Local Search algorithm is different than the rest of the algorithm that i have described before.Here, First of all the local maximum is considered. After getting the local maximum value, algorithm starts doing random small walks nearby in the pursuit of another local value which will be better than the present local maxima. For these random walks, hill climbing is performed. If it finds local maxima which is greater than considered at first glance, it replaces with the new one. In the below code, I took the same equation described in the question, now here there will be two loops iterated: one loop is for big steps and one loop is for small "local" steps.Their values are 10 and 200 respectively. Under the small loops, hill climb is performed and the maximum condition is checked again. Outside small steps, in big steps, same condition is checked and evaluated.

In [81]:
```
#Iterated local search
#optimal solution
xopt=rd.uniform(-1,1)
yopt=rd.uniform(-1,1)
zopt=rd.uniform(-1,1)

#optimum function
f_opt=m.exp(m.cos(53*zopt))+m.sin(43*m.exp(yopt))+m.exp(m.sin(29*xopt))+m.sin(67*m

print("\n f_best=",f_opt)
print("\n x=",xopt,
      "\n y=",yopt,
      "\n z=",zopt)

f_best=f_opt
x_best=xopt
y_best=yopt
z_best=zopt
#defining standard number of steps
big_step=500
small_steps=1000

#Iterated local search started from big steps
for i in range(big_step):
    for j in range(1,small_steps):
        xnew=xnew+rd.gauss(0,0.1)
        ynew=ynew+rd.gauss(0,0.1)
        znew=znew+rd.gauss(0,0.1)
        f_new=m.exp(m.cos(53*znew))+m.sin(43*m.exp(ynew))+m.exp(m.sin(29*xnew))+m.
        #checking the condition of local maxima
        if f_new>f_best:
            f_best= f_new
            x_best= xnew
            y_best= ynew
            z_best= znew
     #checking the condition of this local maxima is greater than the global maxim
    if f_best>f_opt:
        fopt=f_best
        xopt=x_best
        yopt=y_best
        zopt=z_best

    #increasing the value of xbest, ybest, and zbest
    xbest=x_best+rd.gauss(0,0.1)
    ybest=y_best+rd.gauss(0,0.1)
```

```
    zbest=z_best+rd.gauss(0,0.1)

#printing the local value
print ( '\n fopt=', f_opt,
        '\n xopt=' , xopt,
        '\n yopt=',yopt,
        '\n zopt=',zopt,)
```

```
f_best= 0.8047639987910946

x= -0.06932260071554874
y= 0.9068895966973785
z= -0.03681240692570964

fopt= 0.8047639987910946
xopt= -0.06932260071554874
yopt= 0.9068895966973785
zopt= -0.03681240692570964
```

Observations and conclusions

Important Notations

f(x,y,z) : The best value of function set on some particular values

f(x,y,z)""" : The best value of function set after running the algorithm for number of steps

f(x,y,z)"""- f(x,y,z) : Difference between those values

Algorithm name	f(x,y,z)	number of steps	f(x,y,z)"""	f(x,y,z)"""-f(x,y,z)
Random Search	3.20	10000	3.69	0.49
Hill Climb	1.294	10000	1.294	0
Simulated Annealing	1.37	10000	0.73	-0.64
Iterated Local Search	0.80	10000	0.80	0

By Keeping Positive difference in consideration, I preferred **Random Search Algorithm**

Solution for Question 2

Here, the excel sheet for the co ordinates has been provided. Along with that, the distance matrix has been provided.First, I had given the thought of using the brute force algorithm. But as the number of Stores were too much, I have dropped the Idea of the brute force Algorithm.

The main reason behind dropping the Idea was the algorithm searches for every possibilities and provides every permutations. By taking the number of stores in consideration, it could have became time consuming job.

To cope up with this problem, It came under observation that the co-ordinates for every store has been provided. These co-ordinates can be utilized efficiently If I could use Genetic algorithm. Hence I went with Genetic algorithm.

I used Genetic Algorithm for two purposes:

1. To find the efficient distance and the Sequences by which the stores should be prioritize to provide the supply
2. To prefer the mode of transport for every store.

1. Genetic Algorithm to find the efficient route and distance : In the first one, I have used genetic algorithm to find the route and the distances, here I am using the co ordinates data that has been provided under the excel sheet.

In [58]:
```
# PART 1 = Creates the Distance Matrix
# PART 2 = Plots the optimial route on a map.
# PART 3 = Calculates the distance of a given path
# PART 4 = Main GA parameters (Probability of creating each type of child)
# PART 5 = Carries out a Crossover
# PART 6 = Carries out a Mutation
# PART 7 = Selects the fittest parents
# PART 8 = Creates the new population for the next generation
# PART 9 = number of generations
# PART 10 =  Input the data for the problem
# PART 11 = Run the main GA (including population size)


import sys, math, random, heapq
import matplotlib.pyplot as plt
from itertools import chain

if sys.version_info < (3, 0):
   sys.exit("""Sorry, requires Python 3.x, not Python 2.x.""")

class Graph:

    def __init__(self, vertices):
        self.vertices = vertices
        self.n = len(vertices)

    def x(self, v):
        return self.vertices[v][0]

    def y(self, v):
        return self.vertices[v][1]

    # Lookup table for distances
    _d_lookup = {}

################################################################################
##### PART 1: Converts grid co-ordinates of the nodes into a distance matrix

    def d(self, u, v):
        # Check if the distance was computed before
        if (u, v) in self._d_lookup:
            return self._d_lookup[(u, v)]
        # Otherwise compute euclidean distance
        _distance = math.sqrt((u[0] - v[0])**2 + (u[1] - v[1])**2)
        # Add to dictionary
        self._d_lookup[(u, v)], self._d_lookup[(v, u)] = _distance, _distance
```

```
        return _distance

################################################################################
##### PART 2: Plots the final optimal route

    def plot(self, tour=None):
        """Plots the cities and superimposes given tour"""
        if tour is None:
            tour = Tour(self, [])

        _vertices = [self.vertices[0]]

        for i in tour.vertices:
            _vertices.append(self.vertices[i])

        _vertices.append(self.vertices[0])

        plt.title("Cost = " + str(tour.cost()))
        plt.plot(*zip(*_vertices), '-r')
        plt.scatter(*zip(*self.vertices), c="b", s=10, marker="s")
        plt.show()

################################################################################
##### PART 3: Calculates the distance of a given (full) path
class Tour:

    def __init__(self, g, vertices = None):
        """Generate random tour in given graph g"""
        self.g = g
        if vertices is None:
            self.vertices = list(range(1, g.n))
            random.shuffle(self.vertices)
        else:
            self.vertices = vertices
        self.__cost = None

    def cost(self):
        """Return total edge-cost of tour"""
        if self.__cost is None:
            self.__cost = 0
            for i, j in zip([0] + self.vertices, self.vertices + [0]):
                self.__cost += self.g.d(self.g.vertices[i], self.g.vertices[j])
        return self.__cost

################################################################################
##### PART 4: IMPORTANT Main GA parameters (Probability of creating each type of ch

class GeneticAlgorithm:

    def __init__(self, g, population_size, k=10, elite_mating_rate=0.2,
                 mutation_rate=0.2, mutation_swap_rate=0.2):   #Probability of eac
        """Initialises algorithm parameters"""

        self.g = g

        self.population = []
        for _ in range(population_size):
            self.population.append(Tour(g))

        self.population_size = population_size
        self.k = k
        self.elite_mating_rate = elite_mating_rate
        self.mutation_rate = mutation_rate
        self.mutation_swap_rate = mutation_swap_rate
```

```
    ##########################################################################
    ##### PART 5 = Crossover - as described in Teaching Material (Week 11 = AI)

        def crossover(self, mum, dad):
            """Implements ordered crossover"""

            size = len(mum.vertices)

            # Choose random start/end position for crossover
            alice, bob = [-1] * size, [-1] * size
            start, end = sorted([random.randrange(size) for _ in range(2)])

            # Replicate mum's sequence for alice, dad's sequence for bob
            for i in range(start, end + 1):
                alice[i] = mum.vertices[i]
                bob[i] = dad.vertices[i]

            # Fill the remaining position with the other parents' entries
            current_dad_position, current_mum_position = 0, 0

            for i in chain(range(start), range(end + 1, size)):

                while dad.vertices[current_dad_position] in alice:
                    current_dad_position += 1

                while mum.vertices[current_mum_position] in bob:
                    current_mum_position += 1

                alice[i] = dad.vertices[current_dad_position]
                bob[i] = mum.vertices[current_mum_position]

            # Return twins
            return Tour(self.g, alice), Tour(self.g, bob)

    ##########################################################################
    ##### PART 6 = Mutation - swaps pairs, as in TSP_HillClimb.py and described in Week
        def mutate(self, tour):
            """Randomly swaps pairs of cities in a given tour according to mutation rat

            # Decide whether to mutate
            if random.random() < self.mutation_rate:

                # For each vertex
                for i in range(len(tour.vertices)):

                    # Randomly decide whether to swap node i
                    if random.random() < self.mutation_swap_rate:

                        # Randomly choose other node to swap j
                        j = random.randrange(len(tour.vertices))

                        # Swap i and j
                        tour.vertices[i], tour.vertices[j] = tour.vertices[j], tour.ver

    ##########################################################################
    ##### PART 7 = Selects the fittest parents = takes a random sample of solutions and
        def select_parent(self, k):
            """Implements k-tournament selection to choose parents"""
            tournament = random.sample(self.population, k)
            return max(tournament, key=lambda t: t.cost())


    ##########################################################################
```

```
##### PART 8 = Creates the new population for the next generation

    def evolve(self):
        """Executes one iteration of the genetic algorithm to obtain a new generat

        new_population = []

        for _ in range(self.population_size):

            # K-tournament for parents
            mum, dad = self.select_parent(self.k), self.select_parent(self.k)
            alice, bob = self.crossover(mum, dad)

            # keep children if better than a parent
            if random.random() < self.elite_mating_rate:
                if alice.cost() < mum.cost() or alice.cost() < dad.cost():
                    new_population.append(alice)
                if bob.cost() < mum.cost() or bob.cost() < dad.cost():
                    new_population.append(bob)

            else:
                self.mutate(alice)
                self.mutate(bob)
                new_population += [alice, bob]

        # Add new population to old
        self.population += new_population

        # Take the fittest individuals in population, including Elitism
        self.population = heapq.nsmallest(self.population_size, self.population, k

################################################################################
##### PART 9 = IMPORTANT ##### number of generations = (similar to number of steps

    def run(self, iterations=50):  #Number of Generations: more = better, but takes
        for _ in range(iterations):
            self.evolve()

    def best(self):
        return max(self.population, key=lambda t: t.cost())

################################################################################
##### PART 10 =  IMPORTANT ## Input the data for the problem

#Practical 10, Question 1
g = Graph([
(54.1391723668124,10.1630144708565),
(5.16311785475916,18.3713689888664),
(58.1077341232713,97.325872003479),
(16.6066387616466,73.6752906752222),
(10.9215147296463,26.9026260431418),
(40.2830425950062,49.0059611185157),
(95.9067950103118,31.2537921390593),
(38.8068853736467,61.8496341187066),
(50.7736281603872,41.7122988413166),
(16.1210451723109,4.17415211185698),
(30.5725260368337,25.6641613944403),
(88.6541848293903,28.6034058450262),
(76.1583329812348,52.0601020081577),
(57.9907902884404,58.2901693224847),
(80.1180100459334,80.7424210060046),
(90.879027610417,90.9769056531731),
(2.58720752188314,45.8084939684228),
(74.0498005304951,2.98613115772289),
```

```
(71.7091166549543,37.2581705831028),
(22.3719171429327,94.5307472331531),
(24.1011220595671,11.0665812260669),
(42.2780005223912,9.14186038530076),
(58.3280585995255,76.1590760811723),
(71.2535984441914,18.2126848196072),
(57.2421947321968,82.6687611686287)])

################################################################################
##### PART 11 = IMPORTANT ## Runs the main GA (including population size)

for _ in range(5):  #Runs the GA 5 times
    ga = GeneticAlgorithm(g, 50)    #Inputs (graph co-ordinates, POPULATION SIZE)
    ga.run()


    ########## Output
    best_tour = ga.best()
    #g.plot(best_tour)   #Plots the path on a graph

    best_path = best_tour.vertices
    size = len (g.vertices)
    for i in range(size-1):
        best_path[i]=best_path[i]+1   #Must +1 to nodes to chnage numbering from (0
    best_path.insert(0,1)   #Adds start node to the list
    print(best_path,best_tour._Tour__cost)   #Prints the best path and distance
```

```
[1, 22, 21, 10, 2, 11, 6, 25, 8, 16, 15, 23, 20, 3, 4, 17, 5, 19, 13, 12, 18, 14,
9, 24, 7] 763.2145488170531
[1, 17, 9, 11, 14, 25, 3, 4, 20, 16, 15, 23, 8, 6, 22, 7, 12, 19, 13, 18, 24, 5,
2, 10, 21] 765.2849403636998
[1, 22, 18, 3, 23, 25, 16, 15, 13, 19, 24, 12, 14, 21, 10, 5, 11, 2, 17, 9, 8, 20,
4, 6, 7] 778.6622447038028
[1, 22, 10, 6, 20, 25, 14, 23, 3, 15, 16, 7, 19, 21, 2, 5, 17, 12, 18, 24, 13, 4,
8, 9, 11] 804.3848091868077
[1, 21, 11, 10, 2, 17, 5, 6, 12, 13, 15, 14, 9, 19, 24, 7, 16, 25, 23, 20, 4, 3,
8, 22, 18] 758.5000640613383
```

2. Genetic Algorithm to prefer the mode of transport for every store : After getting Efficient distance using genetic algorithm, now it is time to determine whether to use van or lorry. for this I am about to implement abother genetic algortihm that will determin when to use van and when to use Lorry.

The general assumption I am considering over here is : Both Warehouses W1 and W2 are Sharing 3 vans and 1 lorry.

As we know that the problem of prioritizing vehicles for every shop can be resolved by many algorithms. But here, I have preferred Genetic Algorithm to solve this issue as it will come up with efficient solution.

I the below code snippet, I am just assigning the values of routes and efficient distance I have got from the previous code snippet

In [59]:
```
a=best_tour._Tour__cost
b=best_path
```

Genetic Algorithm Involves some basic steps

: **1. Creation of genomes :** In order to create the bunch of solutions or population, one must have skeletal structure of the solution(or here I am calling it genomes). For this, I am

just passing the length of variable "b" which has been assigned from the above code snippet. The function gen_genome() will create the binary sequences of length of variable "b" which is nothing but the best route I got by running the above genetic algorithm.

I have created named Tuple for vans and lorry where each one has the attribute "Name" which is nothing but the name of the vehicles, "costpermil", which is nothing but the cost per mile, for van it is 1 pounds and for lorry it is 2 pounds. And the Most important attribute which i am using for fitness score determination that will be explained is : "no_node" which is nothing but the maximium number of nodes or stores that particular vehicle can visit. For Vans, it is 12 as I am assuming both warehouses are sharing 3 vans. Hence 3 Vans multiplied by the maximum number of stores each van can visit i.e. 4, which is equal to 12. And as I am using single lorry, the number of Stores it can visit will remain 16.

Now I took the binary Sequence over here because there were two options : Go for the Vans or single lorry. Here "1" will be for the Vans and "0" will be for the lorry.

2. Creation of the population : After creating specimens(or genomes) it is time to create the population. The function "generate_population()" will create the bunch of solutions which will be passed for the fitness score assessment.

3. Fitness Score determination : After creating population(or bunch of solutions). It is time to assess fitness score for each genome in the population. For this I am passing the binary sequence. For every sequence, I am checking the number of 1's and 0's. I am maintaining the number of counts for number of 1's and 0's. If number of 1's or 0's are greater than the maximum number of nodes I described on the point number 1, the score 0 will be assesed for that particular solution.

4. Selection Function: The Function selection_pair() selects the pair of solutions(or genomes) depending on the fitness score. Here I am keeping two by default solutions for the next generation.

5. Crossover function: The function single_point_crossover() performs crossover function once. lets assume two genomes: genome1 and genome2. Under this function some part of genome1 is assigned to genome2 and some part of genome2 is assigned to the genome1. Although one can perform multiple crossover function, but here I have assumed to go with single one by keeping complexity in consideration.

6. Mutation Function : The mutation() function changes the small part of the solution or genome, creating the completely new geneartion of solution.

After this, I am running one function for the Evolution and it took 100 generations to get the perfect solutions for me. The elpased time was 1.9 milliseconds.

In [70]:
```
#this is the genetic algorithm coding to determine what kind of vehicle can be prej
#Creation of Genomes
#Creation of Population
#Determine the fitness score of every individual
#Selection of those individuals
#Crossover of mates
#Mutating selected individuals
#Creating the new generation
```

```
#####Before that, I have to import some of the import libraries
from collections import namedtuple
from typing import List, Callable, Tuple

###########Creating Genomes: a genetic representation of solution
import random as rd
from functools import partial

Genome=List[int] #Accepting the genome sequences
Population=List[Genome] #List/"population" of Genomes
Thing=namedtuple('Vehicle',['name','costpermil','no_node']) #preference of the mod
FitnessFunc=Callable[[Genome],int] #fitness function determination
PopulationFunc=Callable[[],Population]
SelectionFunc=Callable[[Population,FitnessFunc],Tuple[Genome,Genome]] #Selection f
CrossoverFunc=Callable[[Genome,Genome],Tuple[Genome,Genome]] #Crossover function
MutationFunc=Callable[[Genome],Genome] #Mutation function

vehicle=[]
#here I am assuming that both ware houses are sharing 3 vans and one lorry, hence t
car=Thing('Vans',1,12)
truck=Thing('lorry',2,16)

############################################## Creation of Genomes : Creating the sh
def gen_genome(length):
    for i in range(length):
        #generation of random values of the length b which is nothing but the lengt
        return rd.choices([0,1],k=length)


####################Creation of population : the function of generating new solutio
def generate_population(size,genome_length):
    #declaring global value to maintain the track of the length of best route
    global length1
    length1=genome_length
    return [gen_genome(genome_length) for _ in range(size)]

####################Determine the fitness score of every individual:the fitness fu
def fitness_score(genome:Genome):
    #checking the value of the length of genome with the best route
    if len(genome)!=length1:
        raise ValueError("Invalid Genome length")

    node1=0
    value=0
    node2=0
    for i in range(len(genome)):
        if genome[i]==1:
            #maintaining the count for the number of 1's which will be the preferen
            node1=node1+1
        elif genome[i]==0:
            #maintaining the count for the number of 0's which will be the preferen
            node2=node2+1

    #if the number of counts of 1's or 0's is exceeding the maximium number of stor
    if node1>car.no_node or node2>truck.no_node:
        return 0
    else:
        value=node1+node2
        return value

################################Selection of those individuals : Selection functio
def selection_pair(population, fitness_func):
    #solution for the highest fitness function should be selected
    return rd.choices(population,weights=[fitness_score(genome) for genome in popul
```

```
##################################Crossover of mates : here the single point crossove

def single_point_crossover(Genome1,Genome2):
    #checking the Length of pair of genomes
    if(len(Genome1)!=len(Genome2)):
        raise ValueError("The Length of both the genomes should be same")

    length2=len(Genome1)
    #taking the random number for the assignment
    p=rd.randint(1,length2-1)

    #Returning modified genes
    return Genome1[0:p]+Genome2[p:],Genome2[0:p]+Genome1[p:]

##################################Mutation:Mutating selected individuals###########

def mutation(genome,num,probability=0.5):
    for _ in range(num):
        #performing slight modification
        index=rd.randrange(len(genome))
        genome[index]=genome[index] if rd.random()>probability else abs[genome[ind

    return genome

##################################Running the evolution#############################
def run_evolution(Populate_func:PopulationFunc,fitness_lim,fitness_func:FitnessFun
    population=Populate_func()

    #generating the population or bunch of solutions
    for i in range(gen_lim):
        population=sorted(population,key=lambda genome:fitness_func(genome),revers

        #fitness score assessment
        if fitness_func(population[0])>=fitness_lim:
            break
    #carrying over first pair of solutions
    next_gen=population[0:2]

    #performing selection, crossover and mutation function to create next generati
    for j in range(int((len(population)/2)-1)):
        parents=selection_func(population,fitness_func)
        offsrping_a,offspring_b=crossover_func(parents[0],parents[1])
        next_gen+=[offsrping_a,offspring_b]

    #sorting the next generation in order to keep the best one at the top with high
    population=sorted(population,key=lambda genome:fitness_func(genome),reverse=Tru

    return population,i+1

#####################################################################Run the Entire Genet

import time
#keeping the track of time
start_time=time.time()

#running the function of evolution
populations,generations=run_evolution(Populate_func=partial(generate_population,si
                                      fitness_func=partial(fitness_score),
                                      fitness_lim=740,
                                      gen_lim=100)
end_time=time.time()

#labelling the name according to the binary notifications : 1 for vans and 0 for lo
```

```
def genome_to_things(Genome):
    result=[]
    for i in range(len(Genome)):
        if Genome[i]==1:
            result+=[car.name]
        else:
            result+=[truck.name]

    return result

print(f"number of generations: {generations}")
print(f"Elapsed time : {end_time-start_time}")

result1=genome_to_things(populations[0])
```

```
number of generations: 100
Elapsed time : 0.01900315284729004
```

Observations:

In the last code statement I am multiplying the distance in terms of number of miles with total cost which is equal to total cost of vans and total cost of lorry.

Below Code Snippet is the output of entire solution

In [73]:
```
################################################## Showing All the outputs ########
print(f"Stores to which each warehouse should provide the supply :\n {b}")
print(f"\n\nMinimum distance each vehicle should take if that vehicle is visiting e

print("\n\nPreferences of the vehicles Per store\n")
for i in range(len(b)):
    print(f"{result1[i]} can be utilized for Store {b[i]}")

print(f"\n\nThe total cost should be :{a*5} pounds")#as I am taking 3 vans each va
```

```
Stores to which each warehouse should provide the supply :
 [1, 21, 11, 10, 2, 17, 5, 6, 12, 13, 15, 14, 9, 19, 24, 7, 16, 25, 23, 20, 4, 3,
8, 22, 18]


Minimum distance each vehicle should take if that vehicle is visiting every Store
:758.5000640613383


Preferences of the vehicles Per store

lorry can be utilized for Store 1
Vans can be utilized for Store 21
lorry can be utilized for Store 11
lorry can be utilized for Store 10
Vans can be utilized for Store 2
lorry can be utilized for Store 17
lorry can be utilized for Store 5
Vans can be utilized for Store 6
lorry can be utilized for Store 12
Vans can be utilized for Store 13
Vans can be utilized for Store 15
Vans can be utilized for Store 14
lorry can be utilized for Store 9
lorry can be utilized for Store 19
lorry can be utilized for Store 24
lorry can be utilized for Store 7
Vans can be utilized for Store 16
Vans can be utilized for Store 25
lorry can be utilized for Store 23
Vans can be utilized for Store 20
Vans can be utilized for Store 4
lorry can be utilized for Store 3
Vans can be utilized for Store 8
lorry can be utilized for Store 22
lorry can be utilized for Store 18


The total cost should be :3792.5003203066917 pounds
```

Solution for Question 3

Background: The Scenario is there are 5 friends living in 5 different cities from Edinburgh and they goes as per the Sequence : 1. Edinburgh, 2. Aberdeen, 3. Glasgow, 4. Dundee, 5. Inverness and 6.Stirling. I have to go and meet each and every single friend but being an international student, I dont know from where should I start. As Trains and buses on strike, either I have to go by taxi or by my own car. As public transports are on strike, the rates of taxis were all time high. Hence, i preferred my own car as fuel were costing 2 pounds per mile.

Aim: The main aim over here is to find the sequence of towns going through which the distance will be minimum and hence, the cost.

Model: This problem is related to travelling sales person problem.I have the data of distance in terms of the miles from Edinburgh to all cities. The Distance matrix from Edinburgh to rest of the cities is as shown as follows(**Note:The upper triangle of matrix is kept empty on purpose as the distance are similiar and the figures are rounded off**) :

	1	2	3	4	5	6
1	0					
2	127	0				
3	47	146	0			
4	62	66	81	0		
5	156	103	168	137	0	
6	38	120	30	56	143	0

To solve this problem, I have the following data:

Mode of transport	Cost per litre(pounds/litre)	Average(miles/litre)
Car	2	10

As the number of cities(or vertices) is low, so the number of possible combinations are 720. Now the thing is for every combination, I might have to take out the product of distances and cost per mile

To find the efficient path and minimum distance, I am using the brute force algorithm and then I am calculating the cost as per the below formulaes

The formulaes are as follows:

Expected ammount of petrol to be filled(litres)=best distance/Average

Total cost for the petrol= Expected ammount of petrol to be filled(litres)*2

These distances we might get from the possible combinations

Optimization method: Here, the brute force algorithm is used. In the Brute force all the combinations and their respective distances are calculated. Out of those combinations, the most efficient combination is preferred. In the below code snippet, I am performing the same method. I am passing the distance matrix as 'dist'.

And here, I am keeping the track of time so that I will come to know the time it took to run the code.

After that, I have found all possible permutations of paths of the required length and all those paths are stored in variable.

The best distance has been initiallized to the highest value.

Then for every path, distance from end point to start point, and then, rest of intermediate distances are calculated and eveything is summed up. Meanwhile the route combinations are maintained.

If the new distance is less than the initiallized distance, then that distance is considered. Along with that, the respective route is also considered.

After that I have written a short logic where the ammount of required petrol is calculated by taking factor of best distance and Average. Then, the product of amount of petrol and the

rate is calculated.

In [34]:
```
import numpy as np
from itertools import permutations
import time

#starting the time for the sake of keeping the track
start=time.time()

#Distance Matrix
dist=np.array([[0,127,47,62,156,38],
               [127,0,146,66,103,120],
               [47,146,0,81,168,30],
               [62,66,81,0,137,56],
               [156,103,168,137,0,143],
               [38,120,30,56,143,0]])
#################finding all the permutations of paths of required number of lengths

########## Finds all possible permutations of paths of the required length

size = len(dist[1,:])  #Gives the number of nodes
path = []     #Start with an empty vector
for i in range(size):
    path.append(i+1)   #Buids up a vector of 1,2,3,4...,n
all_paths = permutations (path)#uses "permutations" command to get all paths

########## Calculates the distance of each path, one at a time
best_distance =999   #set an very high initial best distance (as minimising)
mini_cost=10000
pet_lit=0    #initializing Expected ammount of petrol to be filled(litres)
Average=10   #Taking the average value
total_cost=0       #initializing the cost
for route in all_paths:
#     print(route)
    new_distance = dist[route[size-1]-1][route[0]-1] #  back to the start inclued j
    for i in range(size-1):
        new_distance = new_distance + dist[route[i]-1][route[i+1]-1]# now all the

#     print(new_distance)

    ##### Check if new path is shorter than best path
    if new_distance < best_distance:
        best_distance = new_distance
        best_path = route

#code to calculate the minimum cost of the petrol
pet_lit=best_distance/Average

total_cost=pet_lit*2 #As per the description, the total cost of petrol is number o

#Dictionaries of cities in Sequence as per the Question:
Cities_scot={1:'Edinburgh',2:'Aberdeen',3:'Glasgow',4:'Dundee',5:'Inverness',6:'St

# print answers
print("Best route all over the cities in scotland(number wise) : ", best_path,
      "\n \n \n Best Distance traced: ", best_distance,
      "\n \n Minimum cost: ", total_cost)

print("\n \n Best route all over the cities in scotland:")
for i in best_path:
    print("\n",Cities_scot[i])
```

```
end = time.time()
print('\n RunTime = ',end-start)   #prints out time taken for code to run
```

```
Best route all over the cities in scotland(number wise) :  (1, 3, 6, 5, 2, 4)


 Best Distance traced:  451

 Minimum cost:  90.2


 Best route all over the cities in scotland:

 Edinburgh

 Glasgow

 Stirling

 Inverness

 Aberdeen

 Dundee

 RunTime =  0.013007402420043945
```

Observations After using the brute force approach to determine the efficient path, the following sequence will be:

Edinburgh>Glasgow>Stirling>Inverness>Aberdeen>Dundee>Edinburgh

For this, I have only one single optimum path by which I got the minimal distance. Brute force algorithm over here tried all of these possibilities and checked their respective optimal distances, firstly, it considered one optimal path and minimal distance, then it tried all of those routes and their minimal distance is checked. If that distance is less than the previous considered distance,then the former route is discarded and the new route is considered.

The minimal distance I have got after tracing is **451 miles**.

After this, I implemented the calculation of total quantity of petrol required and calculated the minimal total cost which is equal to quantity multiplied by the petrol rate per litre.

That is **90 pounds and 20 pence**

The time elapsed to find the optimal solution was **1.3 milliseconds**.

Conclusions It is hard being an international student to find efficient distances and routes manually, especially if you have friends living in the different corners of the nation. For such problems brute force becomes very handy as the number of nodes are low. In the above problem there were only 6 cities and there are 720(i.e. 6!) number of solutions available. The time elapsed for solution finding was nearly one milliseconds. The merit of the algorithm is it will give an accurate solution. The main disadvantage of the brute force algorithm is it will take a lot of time to process as the number of nodes increase. If we take the number of possibilities,nodes i.e. cities, and elapsed time in consideration, all of them are proportionally related. As the number of nodes increased, the number of solutions increased

which soars the processing time. For instance if I take more than 20 nodes i.e. cities in consideration, there will be 2432902008176640000 numbers of possibilities. By the processor I am using, If I take the elapsed time for 6 cities in consideration and implement the same solution for 20 cities, it will take 4.388x10^15 milliseconds which is roughly a hundred thousand years.

Under such cases, Algorithms like Greedy algorithm, Genetic algorithm becomes quite handy.

In [ ]: