1(a)

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city.

1(b)

If there are n cities, the time complexity of iterating all possible routes will be O(n!), which is impossible to calculate when n is large.

1(c)

A postman needs to deliver packages to n families. TSP can help him know the optimal route to visit every families exactly once with minimal total distance.

2(ab)

The teacher had said in the class that we should use the distance that measure in 12:00 am, but I can't find the time parameter in googlemap api. So, I didn't specify the time of the requests in my program.

My code:

```
import googlemaps
import pandas as pd

address = []
df = pd.read_csv('Final_address.csv')
for i in range(31):
  address.append(df['Store Address'][i])

distance_mat = [[0 for x in range(31)] for y in range(31)]

gmaps = googlemaps.Client(key='My api')

for i in range(31):
  result = gmaps.distance_matrix(address[i], [address[j] for j in range(15)], mode = 'driving')
  for j in range(15):
    distance_mat[i][j] = result["rows"][0]["elements"][j]["distance"]["value"]
```

```
  result = gmaps.distance_matrix(address[i], [address[j] for j in
 range(15, 31)], mode = 'driving')
  for j in range(16):
    distance_mat[i][j+15] = result["rows"][0]["elements"][j]["dis
tance"]["value"]
print(distance_mat)
```

Result:

```
distance_mat = [[0, 697, 529, 3160, 4534, 3837, 3378, 4760, 6296,
 5351, 4929, 3592, 9090, 8884, 5444, 8116, 2992, 3681, 3791, 1033
0, 7554, 7161, 5275, 6986, 5001, 9606, 9385, 7941, 7934, 7499, 75
23],
        [891, 0, 469, 2457, 3318, 3133, 2675, 4056, 5945, 4648, 4
226, 3942, 8387, 8181, 4303, 7412, 2289, 2977, 3088, 7919, 6809,
5945, 3889, 5733, 3820, 8861, 8641, 7197, 7190, 6755, 6779],
        [787, 410, 0, 2872, 4246, 3548, 3090, 4471, 6007, 5063, 4
641, 3303, 8801, 8596, 5156, 7827, 2704, 3392, 3503, 10042, 7265,
 6873, 4986, 6698, 4713, 9317, 9097, 7653, 7645, 7211, 7235],
        [3131, 2609, 2953, 0, 1395, 1210, 858, 3168, 4709, 3759,
2556, 3525, 7933, 7728, 3415, 6959, 1871, 1938, 1165, 5996, 4327,
 4023, 1966, 3810, 1897, 6831, 6611, 5167, 4222, 4272, 4290],
        [4838, 4316, 4660, 1709, 0, 899, 1344, 4035, 5334, 4357,
3908, 4810, 8800, 8595, 4282, 7826, 3647, 3611, 778, 4612, 3220,
2638, 2051, 2982, 1098, 5921, 5040, 5579, 4307, 4676, 3246],
        [3919, 3397, 3741, 788, 1011, 0, 423, 3113, 4943, 3705, 3
212, 3889, 7879, 7673, 3360, 6905, 2772, 2726, 778, 7473, 3994, 3
638, 2018, 3478, 1099, 7116, 6895, 5451, 4274, 4325, 3962],
        [3497, 2975, 3318, 365, 1314, 1060, 0, 3416, 5246, 4008,
3515, 4191, 8182, 7976, 3663, 7207, 2237, 2303, 1081, 5612, 3995,
 3638, 2331, 3478, 1401, 7418, 7198, 5754, 4587, 4638, 3959],
        [6340, 5257, 6162, 3908, 4164, 4266, 4197, 0, 1830, 2634,
 3377, 1505, 4769, 4564, 611, 3795, 3571, 2932, 3934, 9476, 7681,
 7562, 5401, 7165, 5390, 9300, 9501, 7884, 7657, 7708, 7647],
        [8039, 6956, 7861, 5607, 4557, 4828, 5896, 2065, 0, 965,
1970, 2619, 3315, 3109, 1602, 3212, 5270, 4631, 4052, 8323, 6528,
 6408, 3329, 6011, 4237, 11540, 8347, 6615, 7038, 5504, 6494],
```

```
        [5864, 5073, 5686, 3522, 3325, 3596, 3811, 2010, 965, 0,
1005, 2564, 4154, 3949, 1769, 4051, 3179, 2807, 2820, 7090, 4702,
 5176, 2395, 4107, 3004, 7554, 6439, 4477, 4519, 4570, 4587],
        [5612, 4832, 5433, 3217, 3013, 3284, 3506, 3787, 2534, 14
32, 0, 4124, 4723, 4518, 3329, 4620, 2874, 3312, 2507, 6778, 3703
, 4864, 1397, 3109, 2692, 5552, 5441, 3570, 3487, 3572, 3590],
        [6195, 2856, 3457, 5267, 5522, 5625, 5556, 2246, 3196, 29
72, 3966, 0, 5574, 5369, 2344, 4600, 2561, 3250, 5292, 9247, 7452
, 7333, 5172, 6936, 5161, 10659, 9272, 9235, 7428, 7479, 7418],
        [10595, 8510, 10417, 8163, 8083, 8521, 8452, 5150, 3140,
3959, 4556, 5475, 0, 238, 4530, 1112, 7826, 7187, 7577, 11848, 80
70, 9934, 5563, 9537, 7762, 8475, 9015, 6626, 6789, 6954, 8227],
        [10602, 8517, 10423, 8170, 8051, 8322, 8459, 5156, 3108,
3927, 4524, 5481, 275, 0, 4499, 1119, 7833, 7194, 7545, 11816, 82
80, 9902, 5532, 9505, 7730, 8686, 9225, 6836, 7000, 7165, 8437],
        [6611, 4263, 6433, 4180, 4281, 4552, 4469, 271, 1581, 177
1, 2766, 1308, 5009, 4804, 0, 4035, 3843, 2486, 3776, 8046, 6252,
 6132, 3972, 5735, 3960, 11388, 8071, 7620, 6228, 6279, 6217],
        [9645, 7560, 9466, 7213, 7468, 7571, 7502, 4199, 2982, 38
01, 4399, 4524, 978, 772, 4298, 0, 6876, 6237, 7238, 12233, 8231,
 10318, 5406, 9921, 8147, 8637, 9176, 6787, 6951, 7116, 8388],
        [2998, 2219, 2820, 1649, 2460, 2744, 2519, 2068, 3981, 26
60, 2018, 1954, 6422, 6216, 2315, 5448, 0, 770, 2309, 7307, 5513,
 5393, 3200, 4996, 3221, 7676, 7455, 6012, 6004, 5570, 5478],
        [4060, 3281, 3882, 2352, 2672, 2957, 2641, 1935, 3492, 25
27, 2057, 2037, 7810, 7605, 2182, 6836, 1453, 0, 3075, 7346, 5551
, 5432, 3454, 5035, 3260, 7817, 7596, 6153, 5527, 5578, 5517],
        [3991, 3469, 3812, 876, 453, 738, 1160, 3279, 3800, 2824,
 2375, 4054, 8044, 7839, 3526, 7070, 2731, 2797, 0, 5013, 3203, 3
080, 1232, 2687, 320, 6522, 5022, 4858, 3488, 3539, 3169],
        [10883, 7951, 8849, 5019, 4130, 4231, 4654, 7321, 6997, 6
021, 5572, 7874, 10550, 10344, 7080, 10447, 6265, 6712, 4110, 0,
2086, 2002, 3559, 1631, 3862, 4762, 3881, 4625, 3962, 3485, 2242]
,
        [9405, 8910, 9227, 4559, 3491, 3771, 4193, 8557, 8233, 43
39, 6807, 9110, 8746, 8541, 8316, 8087, 5374, 5781, 3385, 2093, 0
, 1806, 2625, 1435, 3179, 3035, 2153, 2898, 2235, 1757, 786],
```

```
        [10609, 7677, 8575, 4218, 3750, 3430, 3853, 7047, 6724, 5
748, 5298, 7601, 10276, 10071, 6806, 10173, 5991, 6438, 3309, 224
4, 2093, 0, 3553, 1965, 3482, 4769, 3888, 4632, 3969, 3492, 2532]
,
        [4968, 4188, 4789, 3021, 1974, 2233, 2655, 4210, 3887, 22
49, 2461, 4764, 7439, 7234, 3970, 7773, 2825, 3233, 1847, 4643, 2
736, 2669, 0, 2219, 1641, 4661, 4551, 3587, 2631, 2682, 2700],
        [6825, 6063, 6647, 3812, 2745, 3024, 3447, 5994, 5671, 46
95, 4245, 6548, 9223, 9018, 5753, 9929, 4700, 5108, 2639, 1787, 1
628, 811, 2104, 0, 2432, 4329, 3447, 4710, 3424, 3067, 1636],
        [4511, 3989, 4333, 1380, 621, 592, 1015, 3705, 4271, 3294
, 2703, 4480, 8471, 8265, 3952, 7497, 3364, 3317, 470, 5233, 4105
, 3259, 1744, 3588, 0, 6936, 6715, 5271, 4000, 4050, 3639],
        [9718, 9222, 9539, 8051, 7526, 6381, 8407, 8506, 8497, 75
12, 7062, 9601, 7356, 8056, 8579, 7602, 7589, 7996, 7102, 4655, 2
946, 4632, 4739, 4261, 5789, 0, 1002, 2739, 2689, 2671, 3103],
        [9377, 8882, 9199, 6167, 5099, 5379, 5801, 14127, 13803,
6335, 6722, 14680, 9912, 9707, 13886, 9253, 7249, 7656, 4993, 365
3, 1944, 3630, 4385, 3259, 4787, 882, 0, 2399, 2335, 2317, 2100],
        [7207, 6712, 7029, 5541, 5016, 5300, 5897, 5995, 4777, 36
93, 4552, 7091, 7343, 7137, 6068, 6684, 5078, 5486, 4591, 4695, 2
721, 4407, 2997, 3915, 4164, 3135, 2937, 0, 1441, 1606, 2878],
        [8521, 8025, 8343, 4336, 4134, 4419, 4841, 6337, 5428, 43
48, 4588, 6891, 7837, 7631, 6097, 7178, 5147, 5555, 3875, 3547, 1
573, 3259, 2370, 2888, 3671, 2805, 2516, 2946, 0, 825, 1730],
        [7835, 7339, 7656, 6168, 5643, 5928, 6524, 6623, 5718, 56
29, 5179, 7339, 7012, 6806, 6544, 6353, 5706, 6113, 5219, 4315, 2
342, 4028, 2725, 3657, 5346, 2790, 2569, 1140, 675, 0, 2499],
        [8619, 8124, 8441, 4229, 3162, 3441, 3864, 6406, 5478, 43
98, 4657, 6960, 7834, 7628, 6165, 8281, 5117, 5524, 3055, 2768, 5
51, 2481, 2478, 2600, 2849, 3252, 2370, 3045, 1730, 1781, 0]]
```

3(a)

The answer will be $\log_{30}(500000000) = 5.889137385669104$

So, the computer can handle 5 nodes at most in the path.

3(b)

```python
order = [i for i in range(1,31)]
random.shuffle(order)
print(order)
```

Result:

```
[7, 15, 2, 19, 11, 25, 21, 30, 14, 10, 26, 1, 27, 20, 9, 5, 23,
18, 24, 12, 29, 17, 8, 4, 6, 28, 3, 13, 16, 22]
```

3(c)

```python
cityNum = 31

def randomSolution():
    cities = list(range(1, cityNum))
    random.shuffle(cities)
    cities.insert(0, 0)
    cities.append(0)
    return cities

def routeLength(sub_tour):
    routeLength = 0
    for i in range(len(sub_tour) - 1):
        routeLength += distance_mat[sub_tour[i]][sub_tour[i + 1]]
    return routeLength

sum = 0
for i in range(1000):
  sub_tour = randomSolution()
  sum += routeLength(sub_tour)
threshold = sum / 1000
print('Average distances: ', threshold)
```

Result:

```
Average distances:  155112.664
```

3(d)

```python
initial_tour_list = []
while len(initial_tour_list) < 1000:
  sub_tour = randomSolution()
```

```
if routeLength(sub_tour) < threshold:
  initial_tour_list.append(sub_tour)
else:
  u = random.random()
  if u < (threshold / routeLength(sub_tour)):
    initial_tour_list.append(sub_tour)
```

4(a)

GA:

Particle definition: a route starts from 0, pass through 1-30 with random order, and then back to 0.
Objective function: sum of the distances of the route.
Goal: minimize the objective function, i.e. minimize the total distance of the route.
Constrain: no constrain
Selection: I use roulette selection, which the weight of particles is the reciprocal of their objective function values.
Crossover: I will randomly mask half (in expectation) of the cities (except start and end city). Those cities that are masked will not be exchanged, while the cities that are not masked will exchange between father and mother. By the way, there is a crossover probability which is set at 0.8, since always doing crossover will lead to bad result so I set a crossover probability meaning the possibility to crossover.
Mutation: Randomly exchange two cities in the route with mutation probability.

SA:

Particle definition: a route starts from 0, pass through 1-30 with random order, and then back to 0.
Objective function: sum of the distances of the route.
Goal: minimize the objective function, i.e. minimize the total distance of the route.
Constrain: no constrain
Boltzmann rate: 1.0
Initial temperate: 1000
Operation: single swap, swap random two cities in the route.
Temperature schedule: Polynomial scheduling

4(b)

GA:

INPUT n: number of iterations

INPUT population: a list of routes

INPUT rouletteSelection: selection mechanism function

INPUT crossover: crossover mechanism function

INPUT mutation: mutation mechanism function

DATA t : current time index

DATA child1 : first child generate by crossover

DATA child2 : second child generate by crossover

OUTPUT best_tour: best route the GA found

OUTPUT best_points: the total distance of best route, i.e min distance

```
ALGORITHM
t = 0
For t < n:
    population = rouletteSelection(population)
    child1, child2 = crossover(population)
    population = mutation(population)
    best_points = min(population)
    best_route = population.index(min(population))
    t += 1
return best_tour, best_points
```

SA:

INPUT n: number of iterations

INPUT routeLength: objective function, total distances of route

INPUT single_swap: operation, swap two cities in route

INPUT getTemp: temperature scheduling function

INPUT K: Boltzmann rate

DATA t : current time index

DATA cur_tour : the point currently investigated

DATA new_tour : the newly generated individual

DATA cur_dist : the distance of currently point

DATA new_dist : the distance of newly generated individual

OUTPUT best_tour: best route the GA found

OUTPUT best_points: the total distance of best route, i.e min distance

ALGORITHM

```
t = 0
while t < n:
    new_tour = single_swap(cur_tour)
    new_dist = routeLength(new_tour)

    if new_dist < cur_dist:
      cur_tour = new_tour
      cur_dist = new_dist
      if cur_dist < best_dist:
          best_tour = cur_tour
          best_dist = cur_dist
    else:
        if rand(0.0, 1.0) < exp( - (new_dist - cur_dist) / (K * getTemp(t)):
            cur_tour = new_tour
            cur_dist = new_dist

    t += 1
  return best_tour, best_dist
```

4(c)

population and initial_tour_list : list of routes (2d python list), can use 3(cd) code to generate

distance_mat: distance matrix (2d python list)

iter: number of iterations

GA:

Python code:

```
def GA(population, distance_mat, iter=1000):
  cross_prob = 0.5
  mutate_prob = 0.01
  population_size = 1000
  city_num = 31
  mps = 1000 # mating pool size

  def fitness_val(sub_tour):
```

```
    routeLength = 0
    for i in range(len(sub_tour) - 1):
        routeLength += distance_mat[sub_tour[i]][sub_tour[i + 1]]
    return routeLength

  def roulette_select(population):
    fitness_list = []
    for tour in population:
      fitness_list.append(1 / fitness_val(tour)) # inverse the fitness value since our target is minimization

    population = random.choices(population, k=mps, weights=fitness_list)
    return population

  def crossover(tour1, tour2):
    tour1_child = tour1.copy()
    tour2_child = tour2.copy()
    if random.random() < cross_prob:
      remain1 = tour1.copy()
      remain2 = tour2.copy()
      mask = list(np.random.randint(2, size=city_num-1))
      mask.insert(0, 1)
      for m in range(city_num):
        if mask[m] == 1 :
          remain2.remove(tour1_child[m])
          remain1.remove(tour2_child[m])

      t = 0
      for m in range(city_num):
        if mask[m] == 0 :
          tour1_child[m] = remain2[t]
          tour2_child[m] = remain1[t]
          t += 1

    return tour1_child, tour2_child

  def mutation(sub_tour):
```

```
    if random.random() < mutate_prob:
      pos1, pos2 = random.sample([i for i in range(1, 30)], k=2)
      sub_tour[pos1], sub_tour[pos2] = sub_tour[pos2], sub_tour[p
os1]
    return sub_tour

  # initial population
  population_fit = [fitness_val(tour) for tour in population]
  best_points = max(population_fit)
  best_tour = population[population_fit.index(max(population_fit)
)]

  for _ in range(iter):
    # select
    population = roulette_select(population)
    # crossover
    mating_pool = list(range(0, mps))
    random.shuffle(mating_pool)
    for i in range(int(mps/2)):
      child1, child2 = crossover(population[mating_pool[i*2]], po
pulation[mating_pool[i*2+1]])
      population.append(child1)
      population.append(child2)
    # mutation
    for i in range(len(population)):
      population[i] = mutation(population[i])

    # evaluation
    population_fit = [fitness_val(tour) for tour in population]
    best_points = min(population_fit)
    best_tour = population[population_fit.index(min(population_fi
t))]

  return best_tour, best_points
```

SA:

Python code:

```
def SA_multiparticles(initial_tour_list, distance_mat, particles_
num = 1000):
  def SA(initial_tour, distance_mat, iter = 1000):
    city_num = 31
    K  = 1.0  # Boltzmann rate
    T_start = 1000  # initial temperate

    def single_swap(sub_tour):
      temtour = sub_tour.copy()
      ran = random.sample(range(1, city_num), k=2)
      temtour[ran[0]], temtour[ran[1]] = temtour[ran[1]], temtour
[ran[0]]
      return temtour

    def routeLength(sub_tour):
      routeLength = 0
      for i in range(len(sub_tour) - 1):
          routeLength += distance_mat[sub_tour[i]][sub_tour[i + 1
]]
      return routeLength

    def getTemp(t):
      return (1 - t/iter) * T_start

    cur_tour = initial_tour
    cur_dist = routeLength(cur_tour)
    best_tour = cur_tour.copy()
    best_dist = cur_dist

    t = 0;
    while t < iter:
      new_tour = single_swap(cur_tour)
      new_dist = routeLength(new_tour)

      if new_dist < cur_dist:     # keep new_tour if energy is re
duced
        cur_tour = new_tour
        cur_dist = new_dist
```

```
        if cur_dist < best_dist:
            best_tour = cur_tour.copy()
            best_dist = cur_dist
      else:
          if random.uniform(0.0, 1.0) < math.exp( - (new_dist - c
ur_dist) / (K * getTemp(t)) ):
              cur_tour = new_tour.copy()
              cur_dist = new_dist

      t += 1
    return best_tour, best_dist

  best_dist = -1
  for i in range(particles_num):
    cur_tour, cur_dist = SA(initial_tour=initial_tour_list[i], di
stance_mat=distance_mat, iter=1000)
    if cur_dist < best_dist or best_dist == -1:
      best_dist = cur_dist
      best_tour = cur_tour
  return best_tour, best_dist
```

4(d)

GA:

Optimal sequence: `[0, 2, 1, 17, 14, 7, 8, 9, 10, 22, 29, 27, 28, 25, 26, 30, 20, 19, 23, 21, 18, 4, 24, 5, 6, 3, 11, 15, 13, 12, 16, 0]`

Optimal distance: `54058`

Computing time: `34.3478524684906 seconds`

SA:

Optimal sequence: `[0, 16, 17, 7, 14, 11, 9, 8, 13, 12, 15, 10, 22, 25, 26, 28, 29, 27, 30, 20, 19, 21, 23, 4, 18, 24, 5, 6, 3, 1, 2, 0]`

Optimal distance: `50385`

Computing time: `9.625982999801636 seconds`

5(a)

SA outperform GA in TSP. It needs shorter time to calculate can usually get better route than GA. SA is more stable than GA, too. It can always get quite good results (best distance always around 50000). GA is unstable. This also means it can sometimes get better route than SA. But, GA may also can get route that distance is around 70000. Overall, I would say that if you won't run for many times, SA will be a better choice.

5(b)

GA:

Pros: Since unstable results, I can seldom get really good result (45000 distance route)
Cons: large computing time, unstable results

SA:

Pros: small computing time, always get good route
Cons: I can't find any cons

5(c)

Best method here is SA. We can iterate more in a temperature. In my code, I only do one operation in one temperature. We could do more operations in one temperature. The reason is to traverse its neighborhood as much as we can, try to find better neighbor in this temperature. E.g. iterate 200 times in 300c instead of 1 time.

5(d)

The threshold of rejection sampling can change. We can first do SA one time with random initial route. The best distance of this SA will be the new threshold. By this mean, the initial sequences can better than original method, and the computing time of one SA is neglectable (the 9.625 seconds is 1000 times of SA, meaning doing SA one time is only 9.625ms)