1.

Fertilizer	Production cost	Material cost	Sum of cost	Selling price	Earns per ton
A	100	255	355	350	-5
B	150	320	470	550	80
C	200	410	610	450	-160
D	250	465	715	700	-15

In the above table, we can clearly see that only B is profitable. So, after supply 5000 tons of A and 4000 tons of D, we should put all of our money to produce B to maximize our profit.

Available tons	Nitrates	Phosphates	Potash	Chalk
Before	1000	2000	1500	unlimited
After supply A&D	150	1300	650	unlimited

After supply 5000 A and 4000 D, we can only supply at most 3000 B, since the bottleneck is on Nitrates.

The quantity of each fertilizer to be produced:

	A	B	C	D
Quantity	5000	3000	0	4000

Profit: 5000*(-5) + 4000*(-15) + 3000*80 = 155000

I further use Gurobi (C++ linear problem solver) to verify my thoughts.

Code:

```
#include <iostream>
#include <fstream>
#include <string>
#include "gurobi_c++.h"
#include <sstream>

using namespace std;

/*read data*/
int main()
{
/*build environment*/
	try {
		GRBEnv env = GRBEnv();
		GRBModel model = GRBModel(env);

		/* variable setting*/
		//
		GRBVar A;
		A = model.addVar(5000, GRB_INFINITY, 0, GRB_CONTINUOUS);
```

```
GRBVar B;
B = model.addVar(0, GRB_INFINITY, 0, GRB_CONTINUOUS);
GRBVar C;
C = model.addVar(0, GRB_INFINITY, 0, GRB_CONTINUOUS);
GRBVar D;
D = model.addVar(4000, GRB_INFINITY, 0, GRB_CONTINUOUS);

model.update();

/*Objectives*/
GRBLinExpr sum = 0,sum1=0,sum2=0,sum3=0;
sum = (350-255)*A + (550-320)*B + (450-356)*C + (700-465)*D;
sum1 = 100*A + 150*B + 200*C + 250*D;
model.setObjective(sum-sum1, GRB_MAXIMIZE);
/*Constraints*/
//1
sum1 = 0.05*A + 0.05*B + 0.1 *C + 0.15 *D;
model.addConstr( sum1 <= 1000);
//2
sum2 = 0.1*A + 0.15*B + 0.2 *C + 0.05 *D;
model.addConstr( sum2 <= 2000);
//3
sum3 = 0.05*A + 0.1 *B + 0.1 *C + 0.15 *D;
model.addConstr( sum3 <= 1500);

/*Solve the problem*/
model.update();
model.optimize();

/*Output the result*/
cout << "\nObj: " << model.get(GRB_DoubleAttr_ObjVal) << endl;

cout<<"\nA:" <<endl;
cout<<A.get(GRB_DoubleAttr_X)<<endl;
cout<<"\nB:" <<endl;
cout<<B.get(GRB_DoubleAttr_X)<<endl;
cout<<"\nC:" <<endl;
cout<<C.get(GRB_DoubleAttr_X)<<endl;
cout<<"\nD:" <<endl;
cout<<D.get(GRB_DoubleAttr_X)<<endl;
```

```
	}
	catch (GRBException message) {
		cout << "Error code = " << message.getErrorCode() << endl;
		cout << message.getMessage() << endl;
	}
	catch (...) {
		cout << "Exception during optimization" << endl;
	}
	system("pause");
	return 0;
}
```

Result:

```
Set parameter Username
Academic license - for non-commercial use only - expires 2023-09-24
Gurobi Optimizer version 9.5.2 build v9.5.2rc0 (mac64[arm])
Thread count: 8 physical cores, 8 logical processors, using up to 8
    threads
Optimize a model with 3 rows, 4 columns and 12 nonzeros
Model fingerprint: 0x44936d93
Coefficient statistics:
  Matrix range     [5e-02, 2e-01]
  Objective range  [5e+00, 1e+02]
  Bounds range     [4e+03, 5e+03]
  RHS range        [1e+03, 2e+03]
Presolve removed 3 rows and 4 columns
Presolve time: 0.00s
Presolve: All rows and columns removed
Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    1.5500000e+05   0.000000e+00   0.000000e+00      0s

Solved in 0 iterations and 0.00 seconds (0.00 work units)
Optimal objective  1.550000000e+05

Obj: 155000

A:
5000

B:
3000

C:
0

D:
4000
sh: pause: command not found
Program ended with exit code: 0
```

2.(a)

let X[i] be item arr start at Shadow Daggers

S[i] points 〃 〃

w[i] weights 〃 〃

Objective: $\sum_{i=1}^{15} S[i]X[i] + \text{additional}$

Constrains:

(i) $\sum_{i=1}^{15} W[i]X[i] \leq 529$

(ii)

((X[0] | X[1] | X[2]) & (X[3] | X[4] | X[5]) & (X[12] | X[13] | X[14])) == 1

Additions:

(b) if x[0] & x[5]:
additional += 5

(c) if x[3] & (x[9] | x[8]):
additional += 15

(d) if (x[7] | x[10]) & x[5] & x[14]:
additional += 25

(e) if x[12] & x[13] & x[14]:
additional += 70

2(b)

```
item_size = 15
```

```python
weight_list = [3.3, 3.4, 6.0, 26.1, 37.6, 62.5, 100.2, 141.1, 119.2,
 122.4, 247.6, 352.0, 24.2, 32.1, 42.5]
points_list = [7, 8, 13, 29, 48, 99, 177, 213, 202, 210, 380, 485, 9
, 12, 15]

def fitness_val(bag):
    weight = 0
    for i in range(item_size):
      if bag[i]:
        weight += weight_list[i]
    if weight > 529:
      return 0
    # a
    a_cond = (bag[0] or bag[1] or bag[2]) and (bag[3] or bag[4] or b
ag[5]) and (bag[12] or bag[13] or bag[14])
    if not a_cond:
      return 0
    points = 0
    for i in range(item_size):
      if bag[i]:
        points += points_list[i]
    # b
    if bag[0] and bag[5]:
      points += 5
    # c
    if bag[3] and (bag[9] or bag[8]):
      points += 15
    # d
    if (bag[7] or bag[10]) and bag[5] and bag[14]:
      points += 25
    # e
    if bag[12] and bag[13] and bag[14]:
      points += 70

    return points

count = 0
for i in range(2**15):
  temp = i
  bag = []
  for _ in range(15):
    bag.append(temp%2)
    temp = temp >> 1
  if fitness_val(bag) != 0:
    count += 1
```

```
print(count)
```
Result:

6455

2(cd)

```
import random
import math
import numpy as np
from matplotlib import pyplot as plt
def GA(iter=20):
  cross_prob = 0.1
  mutate_prob = 0.07
  population_size = 10
  item_size = 15
  mps = 6 # mating pool size
  weight_list = [3.3, 3.4, 6.0, 26.1, 37.6, 62.5, 100.2, 141.1, 119.
2, 122.4, 247.6, 352.0, 24.2, 32.1, 42.5]
  points_list = [7, 8, 13, 29, 48, 99, 177, 213, 202, 210, 380, 485,
 9, 12, 15]

  def fitness_val(bag):
    weight = 0
    for i in range(item_size):
      if bag[i]:
        weight += weight_list[i]
    if weight > 529:
      return 0
    # a
    a_cond = (bag[0] or bag[1] or bag[2]) and (bag[3] or bag[4] or b
ag[5]) and (bag[12] or bag[13] or bag[14])
    if not a_cond:
      return 0
    points = 0
    for i in range(item_size):
      if bag[i]:
        points += points_list[i]
    # b
    if bag[0] and bag[5]:
      points += 5
    # c
    if bag[3] and (bag[9] or bag[8]):
      points += 15
    # d
    if (bag[7] or bag[10]) and bag[5] and bag[14]:
      points += 25
    # e
```

```
    if bag[12] and bag[13] and bag[14]:
      points += 70

    return points

  def roulette_select(population):
    fitness_list = []
    for bag in population:
      fitness_list.append(fitness_val(bag))

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

  def crossover(bag1, bag2):
    bag1_child = bag1.copy()
    bag2_child = bag2.copy()
    for i in range(item_size):
      if random.uniform(0.0, 1.0) <= cross_prob:
        bag1_child[i], bag2_child[i] = bag2_child[i], bag1_child[i]
    return bag1_child, bag2_child

  def mutation(bag):
    if random.uniform(0.0, 1.0) <= mutate_prob:
      bag[0], bag[1], bag[2] = 1^bag[0], 1^bag[1], 1^bag[2]
    if random.uniform(0.0, 1.0) <= mutate_prob:
      bag[3], bag[4], bag[5] = 1^bag[3], 1^bag[4], 1^bag[5]
    if random.uniform(0.0, 1.0) <= mutate_prob:
      bag[6], bag[7], bag[8], bag[9], bag[10], bag[11] = 1^bag[6], 1
^bag[7], 1^bag[8], 1^bag[9], 1^bag[10], 1^bag[11]
    if random.uniform(0.0, 1.0) <= mutate_prob:
      bag[12], bag[13], bag[14] = 1^bag[12], 1^bag[13], 1^bag[14]
    return bag

  # random generate 10 population
  population = []
  for i in range(population_size):
    population.append([])
    for _ in range(item_size):
      population[i].append(random.randint(0, 1))
  population_fit = [fitness_val(bag) for bag in population]
  best_points = max(population_fit)
  best_bag = 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]], popul
ation[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(bag) for bag in population]
    best_points = max(population_fit)
    best_bag = population[population_fit.index(max(population_fit))]

  return best_bag, best_points

best_bag, best_points = GA(iter=20)
print(best_bag)
print(best_points)
```

Result:

```
[0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1]
781
```

2(e)

Hill Climbing:

```
def hillClimbing(iter=200):
  item_size = 15
  weight_list = [3.3, 3.4, 6.0, 26.1, 37.6, 62.5, 100.2, 141.1, 119.
2, 122.4, 247.6, 352.0, 24.2, 32.1, 42.5]
  points_list = [7, 8, 13, 29, 48, 99, 177, 213, 202, 210, 380, 485,
 9, 12, 15]

  def fitness_val(bag):
    weight = 0
    for i in range(item_size):
      if bag[i]:
        weight += weight_list[i]
    if weight > 529:
      return 0
    # a
```

```
    a_cond = (bag[0] or bag[1] or bag[2]) and (bag[3] or bag[4] or b
ag[5]) and (bag[12] or bag[13] or bag[14])
    if not a_cond:
      return 0
    points = 0
    for i in range(item_size):
      if bag[i]:
        points += points_list[i]
    # b
    if bag[0] and bag[5]:
      points += 5
    # c
    if bag[3] and (bag[9] or bag[8]):
      points += 15
    # d
    if (bag[7] or bag[10]) and bag[5] and bag[14]:
      points += 25
    # e
    if bag[12] and bag[13] and bag[14]:
      points += 70

    return points

  def single_swap(bag):
    tempbag = bag.copy()
    ran = random.sample(range(1, item_size), k=2)
    tempbag[ran[0]], tempbag[ran[1]] = tempbag[ran[1]], tempbag[ran[
0]]
    return tempbag

  # random bag
  bag = []
  for _ in range(item_size):
    bag.append(random.randint(0, 1))
  best_points = fitness_val(bag)
  best_bag = bag

  for i in range(iter):
    new_bag = single_swap(bag)
    if best_points < fitness_val(new_bag):
      best_bag = new_bag
      best_points = fitness_val(new_bag)
      bag = new_bag

  return best_bag, best_points
```

```
best_bag, best_points = hillClimbing(iter=200)
print(best_bag)
print(best_points)
```

Result:

```
[0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0]
756
```

Random Walk:

```
def randomWalk(iter=200):
  item_size = 15
  weight_list = [3.3, 3.4, 6.0, 26.1, 37.6, 62.5, 100.2, 141.1, 119.
2, 122.4, 247.6, 352.0, 24.2, 32.1, 42.5]
  points_list = [7, 8, 13, 29, 48, 99, 177, 213, 202, 210, 380, 485,
 9, 12, 15]

  def fitness_val(bag):
    weight = 0
    for i in range(item_size):
      if bag[i]:
        weight += weight_list[i]
    if weight > 529:
      return 0
    # a
    a_cond = (bag[0] or bag[1] or bag[2]) and (bag[3] or bag[4] or b
ag[5]) and (bag[12] or bag[13] or bag[14])
    if not a_cond:
      return 0
    points = 0
    for i in range(item_size):
      if bag[i]:
        points += points_list[i]
    # b
    if bag[0] and bag[5]:
      points += 5
    # c
    if bag[3] and (bag[9] or bag[8]):
      points += 15
    # d
    if (bag[7] or bag[10]) and bag[5] and bag[14]:
      points += 25
    # e
    if bag[12] and bag[13] and bag[14]:
      points += 70
```

```
    return points

  def single_swap(bag):
    tempbag = bag.copy()
    ran = random.sample(range(1, item_size), k=2)
    tempbag[ran[0]], tempbag[ran[1]] = tempbag[ran[1]], tempbag[ran[
0]]
    return tempbag

  # random bag
  bag = []
  for _ in range(item_size):
    bag.append(random.randint(0, 1))
  best_points = fitness_val(bag)
  best_bag = bag

  for i in range(iter):
    new_bag = single_swap(bag)
    if best_points < fitness_val(new_bag):
      best_bag = new_bag
      best_points = fitness_val(new_bag)
    bag = new_bag

  return best_bag, best_points

best_bag, best_points = randomWalk(iter=200)
print(best_bag)
print(best_points)
```

Result:

```
[1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0]
814
```

2(f)

I test two operators, swapping and single insertion. The performance of these two operators are quite same. They two seems really random. The fitting value will drop to zero even with high iterations.

Progress diagram:

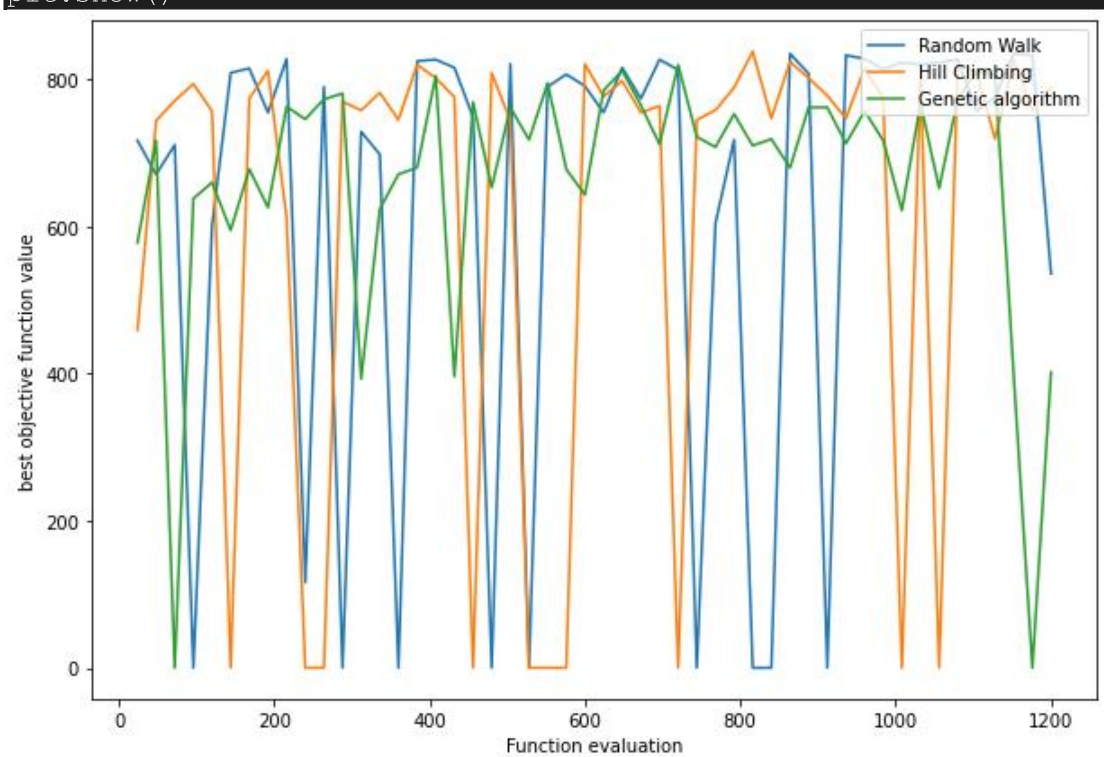
Random walk and hill climbing need 1 function evaluation per iteration. But, GA need 24 function evaluation per iteration (except first iteration). I let the x axis be # function evaluation in multiple of 24 to compare there performances.

```
iteration = list(range(24, 1201, 24))
randomWalk_plot = []
hillClimbing_plot = []
GA_plot = []
for iter in iteration:
```

```
  _, best_dist = randomWalk(iter=iter)
  randomWalk_plot.append(best_dist)
  _, best_dist = hillClimbing(iter=iter)
  hillClimbing_plot.append(best_dist)
  _, best_dist = GA(iter=int(iter/24))
  GA_plot.append(best_dist)

# plot
plt.figure(figsize=(10, 7))
plt.plot(iteration, randomWalk_plot, label='Random Walk')
plt.plot(iteration, hillClimbing_plot, label='Hill Climbing')
plt.plot(iteration, GA_plot, label='Genetic algorithm')
plt.legend(loc='upper right')
plt.ylabel("best objective function value") # y label
plt.xlabel("# Function evaluation") # x label

plt.show()
```



3(a)

```
# Incheon, Seoul, Busan, Daegu, Daejeon, Gwangju, Suwon-si, Ulsan, J
eonju, Cheongju-si, Changwon, Jeju-si, Chuncheon, Hongsung, Muan
distance_mat =  [[0, 27, 335, 244, 141, 257, 33, 316, 186, 115, 304,
 439, 102, 95, 275],
          [27, 0, 330, 237, 144, 268, 31, 307, 195, 113, 301, 453, 7
5, 111, 290],
```

```
            [335, 330, 0, 95, 199, 193, 304, 54, 189, 221, 35, 291, 33
0, 271, 233],
            [244, 237, 95, 0, 117, 171, 212, 75, 130, 130, 72, 324, 23
6, 191, 215],
            [141, 144, 199, 117, 0, 137, 114, 192, 61, 36, 167, 323, 1
75, 74, 171],
            [257, 268, 193, 171, 137, 0, 238, 222, 77, 173, 161, 186,
311, 162, 44],
            [33, 31, 304, 212, 114, 238, 0, 284, 164, 84, 274, 423, 91
, 83, 260],
            [316, 307, 54, 75, 192, 222, 284, 0, 198, 205, 67, 341, 29
6, 266, 265],
            [186, 195, 189, 130, 61, 77, 164, 198, 0, 96, 154, 263, 23
4, 97, 111],
            [115, 113, 221, 130, 36, 173, 84, 205, 96, 0, 190, 359, 13
9, 74, 205],
            [304, 301, 35, 72, 167, 161, 274, 67, 154, 190, 0, 275, 30
6, 237, 202],
            [439, 453, 291, 324, 323, 186, 423, 341, 263, 359, 275, 0,
 498, 344, 165],
            [102, 75, 330, 236, 175, 311, 91, 296, 234, 139, 306, 498,
 0, 170, 340],
            [95, 111, 271, 191, 74, 162, 83, 266, 97, 74, 237, 344, 17
0, 0, 180],
            [275, 290, 233, 215, 171, 44, 260, 265, 111, 205, 202, 165
, 340, 180, 0]]
```

3(b)

One need 14! = 87178291200

3(c)

```
def randomWalk(iter=100):
  city_num = 15
  best_tour = []
  best_dist = 0

  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 randomSolution():
```

```
    cities = list(range(1, city_num))
    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

  sub_tour = randomSolution()
  best_tour = sub_tour.copy()
  best_dist = routeLength(best_tour)

  for i in range(iter):
    new_tour = single_swap(sub_tour)
    if routeLength(new_tour) < best_dist:
      best_tour = new_tour
      best_dist = routeLength(new_tour)
    sub_tour = new_tour

  return best_tour, best_dist

best_tour, best_dist = randomWalk(iter=100)
print(best_tour)
print(best_dist)
```

Result:

```
[0, 8, 1, 12, 9, 11, 14, 5, 10, 2, 13, 4, 7, 3, 6, 0]
2216
```

3(d)

```
def hillClimbing(iter=100):
  city_num = 15
  best_tour = []
  best_dist = 0

  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 randomSolution():
    cities = list(range(1, city_num))
    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

  sub_tour = randomSolution()
  best_tour = sub_tour.copy()
  best_dist = routeLength(best_tour)

  for i in range(iter):
    new_tour = single_swap(sub_tour)
    if routeLength(new_tour) < best_dist:
      best_tour = new_tour
      best_dist = routeLength(new_tour)
      sub_tour = new_tour

  return best_tour, best_dist

best_tour, best_dist = hillClimbing(iter=100)
print(best_tour)
print(best_dist)
```

Result:

```
[0, 1, 12, 4, 13, 11, 7, 2, 10, 3, 9, 8, 14, 5, 6, 0]
1849
```

3(e)

```
def tabuSearch(iter=100):
  city_num = 15
  best_tour = []
  best_dist = 0
  tabu_list = []
  tabu_tenure = 10

  def single_swap(sub_tour):
    temtour = sub_tour.copy()
    ran = random.sample(range(1, city_num), k=2)
    move = (ran[0], ran[1]) if ran[0] < ran[1] else (ran[1], ran[0])
```

```
    temtour[ran[0]], temtour[ran[1]] = temtour[ran[1]], temtour[ran[
0]]
    return temtour, move

  def randomSolution():
    cities = list(range(1, city_num))
    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

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

  for _ in range(iter):
    new_tour = []
    new_dist = -1
    for i in range(1, city_num):
      for j in range(i + 1, city_num):
        test_tour, move = single_swap(cur_tour)
        test_dist = routeLength(test_tour)
        cond = ((move in tabu_list) and ((test_dist < new_dist) or (
new_dist == -1))) or (test_dist < best_dist)
        if cond:
          new_tour = test_tour
          new_dist = test_dist
          moveb = move
    cur_tour = new_tour
    cur_dist = new_dist
    if cur_tour:
      if cur_dist < best_dist:
        best_tour = cur_tour
        best_dist = cur_dist
      tabu_list.append(tuple(reversed(moveb)))
      if len(tabu_list) > tabu_tenure:
        del tabu_list[0]
```

```
    if not cur_tour:
      break
  return best_tour, best_dist

best_tour, best_dist = tabuSearch(iter=100)
print(best_tour)
print(best_dist)
```

Result:

```
[0, 12, 1, 6, 13, 9, 4, 10, 2, 7, 3, 11, 14, 5, 8, 0]
1528
```

3(f)

```
def SA(iter=400):
  city_num = 15
  K  = 1.0  # Boltzmann rate
  T_start = 400 # 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 randomSolution():
    cities = list(range(1, city_num))
    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

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

  cur_tour = randomSolution()
  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 reduced
      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 - cur_di
st) / (K * getTemp(t)) ):
            cur_tour = new_tour.copy()
            cur_dist = new_dist

    t += 1
  return best_tour, best_dist

best_tour, best_dist = SA(iter=400)
print(best_tour)
print(best_dist)
```

Result:

```
[0, 13, 8, 4, 3, 7, 10, 2, 5, 11, 14, 9, 6, 12, 1, 0]
1573
```

3(g)

```
def antColony(iter=200):
  city_num = 15
  ant_num = 30
  decay_rate = 0.8
  scaling = 2

  pheromone_mat = np.full((city_num, city_num), 1, dtype=np.float64)

  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 next_step(ant):
    start = ant[-1]
    candidate = []
```

```
    for i in range(1, city_num):
      if not i in ant:
        candidate.append(i)
    weights = []
    for item in candidate:
      weights.append(pheromone_mat[start][item])
    [next] = random.choices(candidate, weights=weights)
    return next

  for _ in range(iter):
    ants = np.full((ant_num, 1), [0], dtype=int)
    ants = ants.tolist()
    for i in range(ant_num):
      for _ in range(city_num - 1):
        ants[i].append(next_step(ants[i]))
      ants[i].append(0)
    ants_dist = [routeLength(route) for route in ants]
    best_dist = min(ants_dist)
    worse_dist = max(ants_dist)
    best_ants = []
    for i in range(len(ants_dist)):
      if ants_dist[i] == best_dist:
        best_ants.append(ants[i])

    # pheromone decay
    pheromone_mat *= decay_rate
    delta = scaling * best_dist / worse_dist
    for best in best_ants:
      for i in range(len(best) - 1):
        pheromone_mat[best[i]][best[i+1]] += delta
        pheromone_mat[best[i+1]][best[i]] += delta
  return best_ants[0], best_dist

best_tour, best_dist = antColony(iter=200)
print(best_tour)
print(best_dist)
```

Result:

```
[0, 1, 12, 9, 4, 8, 3, 2, 7, 10, 5, 11, 14, 13, 6, 0]
1492
```

3(h)

Hill climbing:

Strength: Can always find at least local minimum with enough iteration.

Weakness: Has high probability to trap in local minimum.

Random walk:

Strength: won't trap in local minimum.
Weakness: its randomness is like random sample. So, it may cost lots of time to find local or global minimum

Simulated annealing:
Strength: Its simple and easy to use, since its base on exhausted search.
Weakness: It use possibility to determine whether to go to the next point, which may cause leaving the global or local minimum.

Tabu search:
Strength: It traverse all neighborhoods without turning back, which can possibly help to get better minimum than hill climbing, since hill climbing doesn't traverse all neiborhoods.
Weakness: It traverse almost all of it neighborhoods, so it may still have high probability to trap in local minimum.

Ant colony:
Strength: With enough ants, this strategy can find really great solution to minimum.
Weakness: Its running time may be very long to this question (The final compared grapth runs 1min30sec because of ant colony) and also the consume a lot of memory when the amounts of ants grows.

```
iteration = list(range(1, 201))
randomWalk_plot = []
hillClimbing_plot = []
tabu_plot = []
SA_plot = []
antColony_plot = []
for iter in iteration:
  _, best_dist = randomWalk(iter=iter)
  randomWalk_plot.append(best_dist)
  _, best_dist = hillClimbing(iter=iter)
  hillClimbing_plot.append(best_dist)
  _, best_dist = tabuSearch(iter=iter)
  tabu_plot.append(best_dist)
  _, best_dist = SA(iter=iter)
  SA_plot.append(best_dist)
  _, best_dist = antColony(iter=iter)
  antColony_plot.append(best_dist)
# plot
plt.figure(figsize=(10, 7))
plt.plot(iteration, randomWalk_plot, label='Random Walk')
plt.plot(iteration, hillClimbing_plot, label='Hill Climbing')
plt.plot(iteration, tabu_plot, label='Tabu Search')
plt.plot(iteration, SA_plot, label='Simulated Annealing')
plt.plot(iteration, antColony_plot, label='Ant Colony')
plt.legend(loc='upper right')
plt.ylabel("best distance") # y label
```

```
plt.xlabel("Iterations") # x label

plt.show()
```