In the Name of God

Sharif University of Technology - Departmenet of Computer Engineering

Artifical Intelligence - Mr. Samiei

Spring 2023

Practical Assignment 1 -

Deadline: Esfand 5th -

Cheating is Strongly Prohibited -

Please run all the cells.

Question 1 (27+5 points)

Imports

Feel free to import any library you need.

In [73]:
```
import numpy as np
import math
```

In this question, we are going to find the gold! The map_list contains a map of Mr. Samiei's personal garden from the surface to a depth of 16 meters. We know that there is a large piece of gold on this map but Mr. Samiei himself has absolutely no idea about what is going on beneath his garden yet. So we can't waste any time.

We have a guess about the location of the gold but we can't take any risks. Hence we will find our way with the Heuristic Search (informed), and the Iterative Deepening Search (uninformed). But first, let's take a quick look at the map.

Making our map

This part is for loading the map and choosing the start point. The first row is the deepest layer and the last row is the surface. As we can see, there is only one entrance at the surface because the blocks with '#' are very hard and we can't pass them. You can transform this map into any other data structure you want and use it to solve the problem.

In [100]:
```
map_list = []
map_list.append(['#', '#', '#', '#', '#', '#', '#', '#', '#'])
map_list.append(['#', '$', '#', '$', '$', '$', '$', '$', '#'])
map_list.append(['#', '$', '$', '$', '$', '$', '#', '$', '#'])
map_list.append(['#', '$', '#', 'G', '#', '$', '#', '$', '#'])
map_list.append(['#', '$', '#', '$', '#', '#', '#', '$', '#'])
map_list.append(['#', '$', '#', '#', '#', '$', '#', '$', '#'])
map_list.append(['#', '$', '#', '$', '#', '$', '#', '$', '#'])
map_list.append(['#', '$', '#', '$', '$', '$', '#', '$', '#'])
map_list.append(['#', '$', '$', '$', '#', '$', '#', '$', '#'])
map_list.append(['#', '$', '#', '$', '#', '$', '#', '$', '#'])
map_list.append(['#', '$', '#', '$', '#', '$', '$', '$', '#'])
map_list.append(['#', '$', '#', '$', '#', '$', '#', '$', '#'])
map_list.append(['#', '$', '#', '$', '#', '$', '#', '$', '#'])
map_list.append(['#', '$', '#', '$', '#', '$', '#', '$', '#'])
map_list.append(['#', '$', '#', '$', '#', '$', '#', '$', '#'])
map_list.append(['#', '#', '#', '#', '#', '#', '#', 'E', '#'])

for floor in map_list:
    print(floor)
# '#' is a hard path and you can't pass
# '$' is a soft path and you can dig it
# 'G' is our goal
# 'E' is the only entrance
```

```
['#', '#', '#', '#', '#', '#', '#', '#', '#']
['#', '$', '#', '$', '$', '$', '$', '$', '#']
['#', '$', '$', '$', '$', '$', '#', '$', '#']
['#', '$', '#', 'G', '#', '$', '#', '$', '#']
['#', '$', '#', '$', '#', '#', '#', '$', '#']
['#', '$', '#', '#', '#', '$', '#', '$', '#']
['#', '$', '#', '$', '#', '$', '#', '$', '#']
['#', '$', '#', '$', '$', '$', '#', '$', '#']
['#', '$', '$', '$', '#', '$', '#', '$', '#']
['#', '$', '#', '$', '#', '$', '#', '$', '#']
['#', '$', '#', '$', '#', '$', '$', '$', '#']
['#', '$', '#', '$', '#', '$', '#', '$', '#']
['#', '$', '#', '$', '#', '$', '#', '$', '#']
['#', '$', '#', '$', '#', '$', '#', '$', '#']
['#', '#', '#', '#', '#', '#', '#', 'E', '#']
```

In [101]:
```
# Start point of problem
initial_point = (15, 7)
map_list[15][7]
```

Out[101]:
```
'E'
```

We start with the first approach. In an uninformed search, we have to forget about our guess for the location of the gold and search until we find the 'G'.

Iterative Deepening Search (Uninformed) (+5 points)

In this section, we have two functions for you and you have to complete the second one. You have to use the DLS function in the IDS function to find the shortest path length and we have already provided the expected output at the end of this section. Remember that you don't need to return the path, we only need the length and the total steps of this algorithm.

In [102]:

```python
def DLS(problem_map, start_point, max_depth):
    if problem_map[start_point[0]][start_point[1]] == '#':
        return (False, 0)
    total_steps = 1
    if problem_map[start_point[0]][start_point[1]] == 'G':
        return (True, total_steps)
    if max_depth <= 0:
        return (False, total_steps)
    if start_point[0] < 15:
        result = DLS(problem_map, (start_point[0] + 1, start_point[1]), max_depth - 1)
        total_steps += result[1]
        if result[0]:
            return (True, total_steps)
    if start_point[0] > 0:
        result = DLS(problem_map, (start_point[0] - 1, start_point[1]), max_depth -1 )
        total_steps += result[1]
        if result[0]:
            return (True, total_steps)
    if start_point[1] > 0:
        result = DLS(problem_map, (start_point[0], start_point[1] - 1), max_depth - 1)
        total_steps += result[1]
        if result[0]:
            return (True, total_steps)
    if start_point[0] < 8:
        result = DLS(problem_map, (start_point[0], start_point[1] - 1), max_depth - 1)
        total_steps += result[1]
        if result[0]:
            return (True, total_steps)
    return (False, total_steps)
```

In [103]:
```
def IDS(problem_map, start_point, max_depth):
    # (5 points)
    # The output must be in this form: (True or False, the shortest path length, total steps until finding th
    total_steps = 0

    for i in range(max_depth):
        result = DLS(problem_map, start_point, i)
        if result[0]:
            return (True, i, result[1])

    # Your code here
    return (False, 0, total_steps)
```

In [120]:
```
answer_IDS = IDS(map_list, initial_point, 50)
if answer_IDS[0]:
    print(f'The IDS algorithm found the shortest path of length {answer_IDS[1]} in {answer_IDS[2]} steps.')
else:
    print('No way!')
```

```
The IDS algorithm found the shortest path of length 20 in 582598 steps.
```

Now we will find the shortest path length using Manhattan heuristic:

Heuristic Search (Informed) (27 points)

Like the previous section, we have to empty functions to complete and you can use the answer at the end of this section to check your outputs.

In [121]:
```
# Goal point of problem
goal = (3, 3)
map_list[3][3]
```

Out[121]:
```
'G'
```

In [122]:

```python
# Manhattan Heuristic
def h(current_point, goal_point):
    # (2 points)
    h = None
    h = int(math.fabs(current_point[0] - goal_point[0])) + int(math.fabs(current_point[1] - goal_point[1]))
    return h
```

In [123]:
```python
def Heuristic_search(problem_map, start_point, goal_point):
    # (25 point)
    # The output must be in this form: (True or False, the shortest path length, total steps until finding th
    total_steps = 0
    shortest_path = 0
    ans = []
    # Your code here
    opened_list = set()
    closed_list = set()

    opened_list.add((start_point[0], start_point[1], 0, None))
    g = {}
    g[(start_point[0], start_point[1], 0, None)] = 0
    while (len(opened_list) != 0):
        q = find_least_f(opened_list, goal_point)
        opened_list.remove(q)
        # find successors
        successors = []

        if q[0] - 1 >= 0:
            successors.append([q[0] - 1, q[1]])

        if q[1] - 1 >= 0:
            successors.append([q[0], q[1] - 1])
        if q[1] + 1 < 9:
            successors.append([q[0], q[1] + 1])


        if q[0] + 1 < 16:
            successors.append([q[0] + 1, q[1]])


        for point in successors:
            if map_list[point[0]][point[1]] == '#':
                continue
            elif map_list[point[0]][point[1]] == 'G':
                shortest_path = len_shortest_path((point[0], point[1], 0, q), 0)
                ans.append(shortest_path)
                continue
            else:
                succ_g = g.get(q) + 1
                succ_h = h(point, goal_point)
                had_lower = False
```

```
                for point2 in opened_list:
                    if point2[0] == point[0] and point2[1] == point[1] and point2[2] < succ_g + succ_h:
                        had_lower = True
                        break

                for point2 in closed_list:
                    if point2[0] == point[0] and point2[1] == point[1] and point2[2] < succ_g + succ_h:
                        had_lower = True
                        break

                if had_lower:
                    continue

                else:
                    if q[3] is not None and point[0] == q[3][0] and point[1] == q[3][1]:
                        continue
                    new_point = (point[0], point[1], succ_g + succ_h, q)
                    total_steps += 1
                    opened_list.add(new_point)
                    g[new_point] = succ_g

        closed_list.add(q)

    if len(ans) != 0:
        return (True, min(ans), total_steps)

    return (False, shortest_path, total_steps)

def len_shortest_path(point_goal, counter):

    if point_goal[3] is None:
        return counter
    else:
        return len_shortest_path(point_goal[3], counter + 1)


def find_least_f(opened_list, goal_point):
    ans = None
    least_f = math.inf
    for point in opened_list:
        heu = h(point, goal_point)
        if heu <= least_f:
            least_f = heu
```

```
            ans = point
    return point
```

In [119]:
```
answer_HS = Heuristic_search(map_list, initial_point, goal)
if answer_HS[0]:
    print(f'The Heuristic algorithm found the shortest path of length {answer_HS[1]} in {answer_HS[2]} steps.
else:
    print('No way!')
```

```
The Heuristic algorithm found the shortest path of length 20 in 169 steps.
```

Question 2 (19+23 points)

Imports

Feel free to import any library you need.

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
```

In this question, we want to solve a scheduling problem for those students who have a lot of available courses to take and don't know how to fit them into their weekly plans! In fact, we need them to provide us with a list of courses with a rating for every course which indicates the tendency of taking that course. Then we use Genetic Algorithm and Simulated Annealing to arrange their weekly plans with the most desired courses from the given least without any conflicts.

Test Data

In the cell below we have a code that generates a random acceptable course list to test your implemented code. According to the current parameters, we have 5 active days every week and every day has 6 free time slots. Also, we have three types of courses to define:

- type 1: Sessions on days 1 and 3 of every week with one timeslot per day. For example, the second timeslot of days 1 and 3.
- type 2: Sessions on days 2 and 4 of every week with one timeslot per day. For example, the third timeslot of days 1 and 3.

- type 3: Sessions on any active day of every week with two consecutive timeslots. For example, the second and third timeslots of day 3.

In [11]:
```
# Generating data
num_courses = 20
daily_timeslots = 6
active_week_days = 5
courses_rating_range = 5
courses = []
for i in range(num_courses):
    course_type = np.random.randint(3)
    if course_type == 0:
        courses.append([[1, 3], [np.random.randint(daily_timeslots) + 1], np.random.randint(courses_rating_ra
    elif course_type == 1:
        courses.append([[2, 4], [np.random.randint(daily_timeslots) + 1], np.random.randint(courses_rating_ra
    else:
        time_slot = np.random.randint(daily_timeslots - 1) + 1
        courses.append([[np.random.randint(active_week_days) + 1], [time_slot, time_slot + 1], np.random.rand
courses
```

Out[11]:
```
[[[1, 3], [4], 2],
 [[2, 4], [3], 1],
 [[2, 4], [6], 3],
 [[4], [3, 4], 3],
 [[2, 4], [5], 1],
 [[2, 4], [6], 5],
 [[2], [4, 5], 1],
 [[2, 4], [1], 2],
 [[1, 3], [5], 2],
 [[2, 4], [5], 4],
 [[1, 3], [3], 4],
 [[4], [4, 5], 1],
 [[2, 4], [5], 3],
 [[2, 4], [5], 5],
 [[2], [5, 6], 3],
 [[1, 3], [1], 5],
 [[1], [2, 3], 4],
 [[1, 3], [2], 1],
 [[2], [2, 3], 3],
 [[2, 4], [4], 5]]
```

Representation of the output is a one-dimensional list of 0s and 1s with the length of num_courses. For every taken course from the courses list, we put a 1 at the corresponding index in the output list and a 0 for the courses that are not taken.

In [12]:
```
# Sample output
sample_output = [np.random.randint(2) for _ in range(num_courses)]
sample_output
```

Out[12]:
```
[1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1]
```

Now lets start with Genetic algorithm.

Genetic algorithm (+23 points)

In every iteration of the genetic algorithm, we have a population of solutions. Then we apply selection, crossover, and mutation operations on the current population to generate the next generation in order to find better solutions for our problem. But first, we need an initial population.

Complete the below function that takes the size of the population and the number of courses in our list and returns a random population:

In [19]:
```
def initial_population(population_size, num_courses):
    # (3 points)
    # returns a list random solutions with size of num_courses
    zeroth_generation = []
    for i in range(population_size):
        zeroth_generation.append([np.random.randint(2) for _ in range(num_courses)])
    return zeroth_generation
sample = initial_population(20,20)
sample
```

Out[19]:
```
[[1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
 [1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0],
 [1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0],
 [1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0],
 [1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0],
 [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
 [1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0],
 [1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0],
 [1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0],
 [0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1],
 [0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0],
 [1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0],
 [1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1],
 [1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1],
 [1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1],
 [1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1],
 [1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1],
 [1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0],
 [1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0],
 [0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0]]
```

After generating the zeroth generation, we need a fitness function to measure how good every solution is. One good approach is to aggregate ratings of all taken courses and determine the fitness as below. You can see if we have any time conflicts in output, the fitness function returns zero. Hence, the best solution should have fitness greater than zero and it means that there are conflicts between chosen courses in our solution.

$$Fitness = \begin{cases} \sum \text{Ratings of taken courses} & \text{no conflicts in the given schedule} \\ 0 & \text{o.w.} \end{cases}$$

Now, complete the function below that takes a generation of solutions as input and returns a list of fitnesses for that generation:

In [21]:

```
def fitness(courses, generation, daily_timeslots, active_week_days):
    # (5 points)
    fitness_list = []
    # Your code here
    for gene in generation:
        schedule = []
        is_conflict = False
        for i in range(active_week_days):
            schedule.append([0 for _ in range(daily_timeslots)])
        gene_fitness = 0
        for i in range(len(gene)):
            if gene[i] == 1:
                if len(courses[i][0])==1:

                    if schedule[courses[i][0][0]][courses[i][1][0]-1]==0 and schedule[courses[i][0][0]][cours
                        schedule[courses[i][0][0]][courses[i][1][0]-1]= 1
                        schedule[courses[i][0][0]][courses[i][1][1]-1] = 1

                        gene_fitness += courses[i][2]
                    else:
                        is_conflict = True
                        break
                else:
                    if schedule[courses[i][0][0]][courses[i][1][0]-1] == 0 and schedule[courses[i][0][1]][cou
                        gene_fitness += courses[i][2]
                        schedule[courses[i][0][0]][courses[i][1][0]-1] = 1
                        schedule[courses[i][0][1]][courses[i][1][0]-1] = 1
                    else:
                        is_conflict = True
                        break

        if is_conflict:
            fitness_list.append(0)
        else:
            fitness_list.append(gene_fitness)

    return fitness_list
fitSample = fitness(courses,sample,6,5)
```

```
print(fitSample)
```

```
[19, 0, 0, 0, 0, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Here we have three functions: selection, crossover, and mutation. The selection function takes a generation and its corresponding fitness list as input and returns a population called parents which will be used in the crossover function to generate the next generation. The method of our selection is Roulette Wheel and you have to implement this method here.

In [22]:
```python
# Roulette Wheel Selection
import random
def selection(generation, fitness_list):
    # (6 points)
    parents = []
    total_fitness = sum(fitness_list)
    select=[]
    r = random.uniform(0,total_fitness)
    current = 0
    for i in range(len(generation)):
        current += fitness_list[i]
        r = random.uniform(0,total_fitness)

        if current > r:
            parents.append(generation[i])
        # Your code here

    return parents
select = selection(sample,fitSample)
select
```

Out[22]:
```
[[1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
 [1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0],
 [1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0],
 [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
 [1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0],
 [1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0],
 [1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0],
 [0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1],
 [0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0],
 [1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0],
 [1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1],
 [1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1],
 [1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1],
 [1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1],
 [1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1],
 [1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0],
 [1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0],
 [0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0]]
```

In the crossover function, we take generated parents from the selection part and do a crossover from a random index for every consecutive pair. Be aware that every parent in the parents list belongs to exactly one pair.

In [24]:
```
def crossover(parents):
    # (4 points)
    new_generation = []

    for i in range(0, len(parents) - 1, 2):
        # Your code here
        crossP = random.randint(0,20)
        parents[i][crossP:len(parents[i])] = parents[i+1][crossP:len(parents[i])]
        new_generation.append(parents[i])

    if len(new_generation) < len(parents):
        new_generation.append(parents[-1])
    return new_generation
cross = crossover(select)
cross
```

Out[24]:
```
[[1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0],
 [1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
 [1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0],
 [1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1],
 [0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0],
 [1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1],
 [1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1],
 [1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0],
 [0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0],
 [0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0]]
```

And at the final step, the mutation function takes the generation as input and for every solution in the generation, randomly chooses an index, and if the value is 0 changes it to 1 and vice versa.

In [25]:
```
def mutation(generation):
    # (3 points)
    for gene in generation:
        change = random.randint(0,19)
        if gene[change] == 1:
            gene[change] = 0
        else:
            gene[change]= 1

       # Your code here
    return generation
mutate = mutation(cross)
mutate
```

Out[25]:
```
[[1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0],
 [1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
 [1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0],
 [0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1],
 [0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0],
 [1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1],
 [1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1],
 [1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0],
 [0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
 [0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0]]
```

Now we can complete our genetic function using above functions:

In [26]:
```
def genetic(courses, daily_timeslots, active_week_days, population_size=50, max_generation=2290, mutation_pro
    # This function do seletion, crossover, and mutation for max_generation number of iterations
    # Note that in every iteration apply mutation only with probability of mutation_prob
    population = initial_population(population_size, len(courses))
    fitness_list = fitness(courses, population, daily_timeslots, active_week_days)
    best_fitness = max(fitness_list)
    best_plan = population[np.argmax(fitness_list)]
    best_fitness_list = []
    for i in range(max_generation):
        population = crossover(selection(population, fitness_list))
        p = np.random.rand()
        if p > mutation_prob:
            population = mutation(population)
        fitness_list = fitness(courses, population, daily_timeslots, active_week_days)
        generation_best_fitness = max(fitness_list)
        if best_fitness < generation_best_fitness:
            best_fitness = generation_best_fitness
            best_plan = population[np.argmax(fitness_list)]
        best_fitness_list.append(max(fitness_list))

    return best_fitness_list, best_fitness, best_plan
```

In [27]:
```
best_fitness_list, best_fitness, best_plan = genetic(courses, daily_timeslots, active_week_days)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[27], line 1
----> 1 best_fitness_list, best_fitness, best_plan = genetic(courses, daily_timeslots, active_week_days)

Cell In[26], line 15, in genetic(courses, daily_timeslots, active_week_days, population_size, max_generatio
n, mutation_prob)
     13     population = mutation(population)
     14 fitness_list = fitness(courses, population, daily_timeslots, active_week_days)
---> 15 generation_best_fitness = max(fitness_list)
     16 if best_fitness < generation_best_fitness:
     17     best_fitness = generation_best_fitness

ValueError: max() arg is an empty sequence
```

In [67]:
```
print(best_fitness)
print(best_plan)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[67], line 1
----> 1 print(best_fitness)
      2 print(best_plan)

NameError: name 'best_fitness' is not defined
```

In [29]:
```
# Write a code to show our best_plan in a two-dimensional list. (2 points)
# You can fill free timeslots with 0 and busy timeslots with 1.
# Your code here

schedule1 = []
is_conflict = False
for i in range(active_week_days):
    schedule1.append([0 for _ in range(daily_timeslots)])
gene_fitness = 0
for i in range(len(best_plan)):
    # Your code here
schedule1
```

```
  Cell In[29], line 12
    schedule1
    ^
IndentationError: expected an indented block after 'for' statement on line 10
```

In [75]:
```
# Plot best_fitness_list over generations.
plt.plot(list(range(1, len(best_fitness_list) + 1)), best_fitness_list)
plt.xlabel('Genration')
plt.ylabel('Best fitness')
plt.title('Best fitness over all generations plot')
plt.show()
```

Simulated Annealing (19 points)

In simulated annealing, we start from a random initial state and try to reach better states. Every state is a solution to our problem or in the other words a list of 0s and 1s that shows which courses are taken by the student. We are only allowed to move from the current state to one of the neighbor states or stay in the current state. we will explain more about how exactly we are going to take our moves, but first, we need to choose our start state.

Complete the function below that takes number of available courses and generates a random initial state:

In [45]:
```
def initial_state(num_courses):
    # (1 point)
    state = []

    for i in range(num_courses):
        s = random.randint(0,1)
        state.append(s)

    return state
state = initial_state(20)
state
```

Out[45]:
```
[0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0]
```

Now we have to find a neighbor state for the next step. We can define a neighbor state as a solution with only one difference. For example, if we have a state like [0, 1, 1, 1, 0], then [1, 1, 1, 1, 0] and [0, 1, 0, 1, 0] are two neighbor states for it.

Complete the function below that takes the current state and returns a random neighbor of it:

In [51]:
```
def get_random_neighbor(current_state):
    # (2 points)
    neighbor_state = current_state.copy()
    index = np.random.randint(len(neighbor_state))
    if neighbor_state[index] == 0:
        neighbor_state[index] = 1
    else:
        neighbor_state[index] = 0
    return neighbor_state
new = get_random_neighbor(state)
new
```

Out[51]:
```
[0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0]
```

After choosing a neighbor we have to compare the fitness of the current and neighbor state and act as below:

IF ($\Delta F = Fitness_{neighbor} - Fitness_{current} > 0$):

Go to the neighbor state

ELSE:

Go to the neighbor state with probability $e^{\frac{\Delta F}{T}}$

- T is temperature and after every step will be multiplied by a positive constant $A < 1$: $T_{i+1} = A * T_i$

We can use our fitness function from the Genetic section to calculate the fitnesses (attention: you have to pass proper inputs to the fitness function and the output will be a list, not a number). Now complete the function below that returns a boolean and tells us if we go to the neighbor state or not in case of $\Delta F < 0$:

In [52]:
```python
import math
import random
def go_neighbor_state(delta_f, temperature):
    # (3 points)
    # The out put must be boolean
    # Your code here
    if delta_f>= 0 :
        return True
    else:
        population = [0,1]
        prob = math.e ^(delta_f/temperature)
        probs = [1-prob,prob]
        result = random.choices(population,probs)
        if result == 0:
            return False
        else:
            return True
```

Now we can complete our simulated annealing function using above functions:

In [61]:
```python
def simulated_annealing(courses, daily_timeslots, active_week_days, temperature=10, temperature_limit=1e-9, t
    # (11 points)
    # temprature_limit and max_iters are two parameters for ending the algorithm
    current_state = initial_state(len(courses))
    current_fitness = fitness(courses, [current_state], daily_timeslots, active_week_days)[0]
    fitness_list = [current_fitness]
    best_state = None
    best_fitness = 0
    T = temperature
    states = []
    states.append(current_state)
    for i in range(max_iters):
        if T<temperature_limit:
            break
        newNeighbor = get_random_neighbor(current_state)
        newFitness = fitness(courses,[newNeighbor],daily_timeslots,active_week_days)[0]
        current_fitness = fitness(courses, [current_state], daily_timeslots, active_week_days)[0]
        fitness_list.append(newFitness)
        states.append(newNeighbor)
        deltaF = fitness_list[i] - current_fitness
        result = go_neighbor_state(deltaF,T)
        if result == True:
            current_state = newNeighbor
        T *= temperature_scale
        # Your code here
    best_fitness = max(fitness_list)
    home = fitness_list.index(best_fitness)
    best_state = states[home]

    return fitness_list, best_state, best_fitness
```

In [62]:
```python
fitness_list, best_state, best_state_fitness = simulated_annealing(courses, daily_timeslots, active_week_days
```

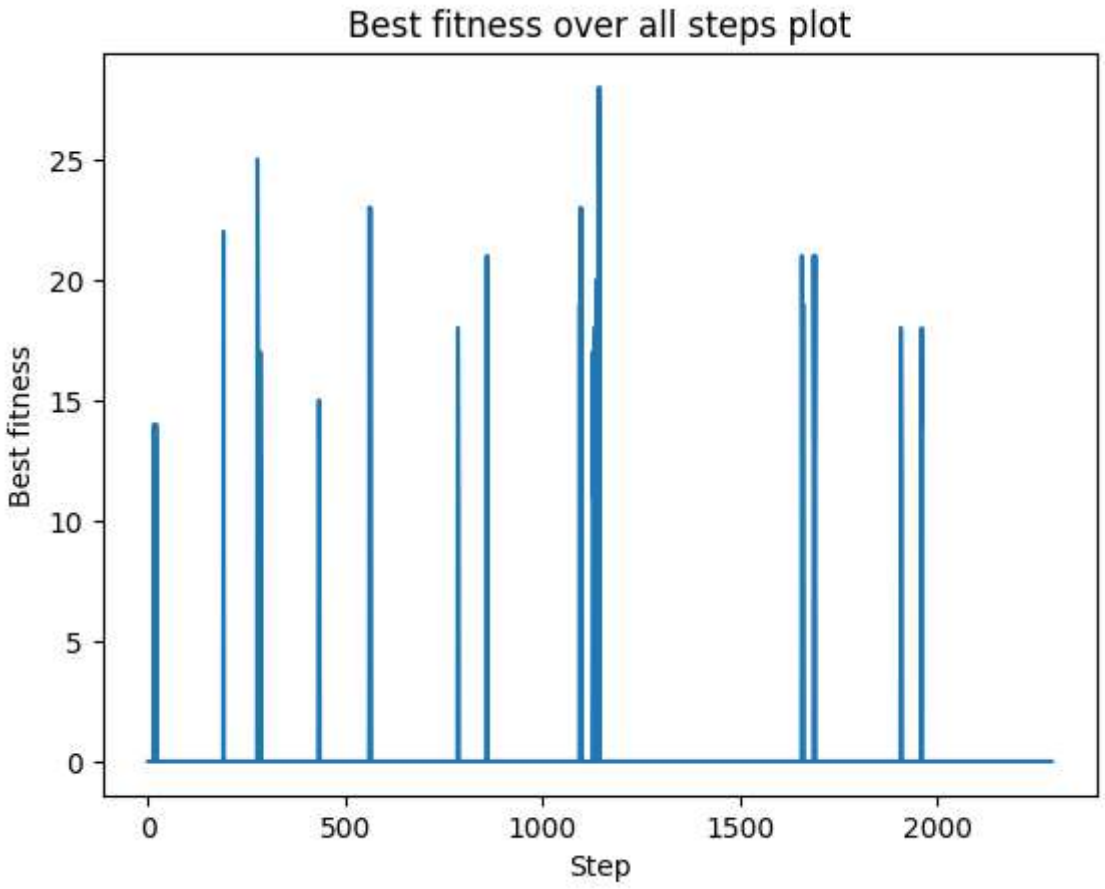
In [63]:
```python
print(best_state_fitness)
print(best_state)
```

```
28
[1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1]
```

In [64]:
```
# Use your code from genetic section to show our best_state in a two-dimensional list. (2 points)
schedule2 = []
is_conflict = False
for i in range(active_week_days):
    schedule2.append([0 for _ in range(daily_timeslots)])
gene_fitness = 0
for i in range(len(best_state)):
    # Your code here
schedule2
```

```
  Cell In[64], line 9
    schedule2
    ^
IndentationError: expected an indented block after 'for' statement on line 7
```

In [65]:
```
# Plot fitness_list over all steps.
plt.plot(list(range(1, len(fitness_list) + 1)), fitness_list)
plt.xlabel('Step')
plt.ylabel('Best fitness')
plt.title('Best fitness over all steps plot')
plt.show()
```



Question 3 (37 points)

In this question we are going to implement gradient descent algorithm, test it on one-variable and two-variable functions and then visualize the path that algorithm takes to reach the global minimum.

Imports

Feel free to import any library you need. You can use any library you want to complete this question.

In [10]:
```
import numpy as np
import matplotlib.pyplot as plt
```

One-variable functions

Fill these empty blocks according to the given functions:

Function 1:

$$f_1(x) = \frac{x^2}{1 - sin(\frac{x}{80})} \quad and \quad x \in [80, 150]$$

In [3]:
```
import math
def f1(x):
    # (1 point)
    # Your code here
    need1 = x*x
    need2 = 1 - np.sin(x/80)
    f = need1*need2
    return f
```

Function 2:

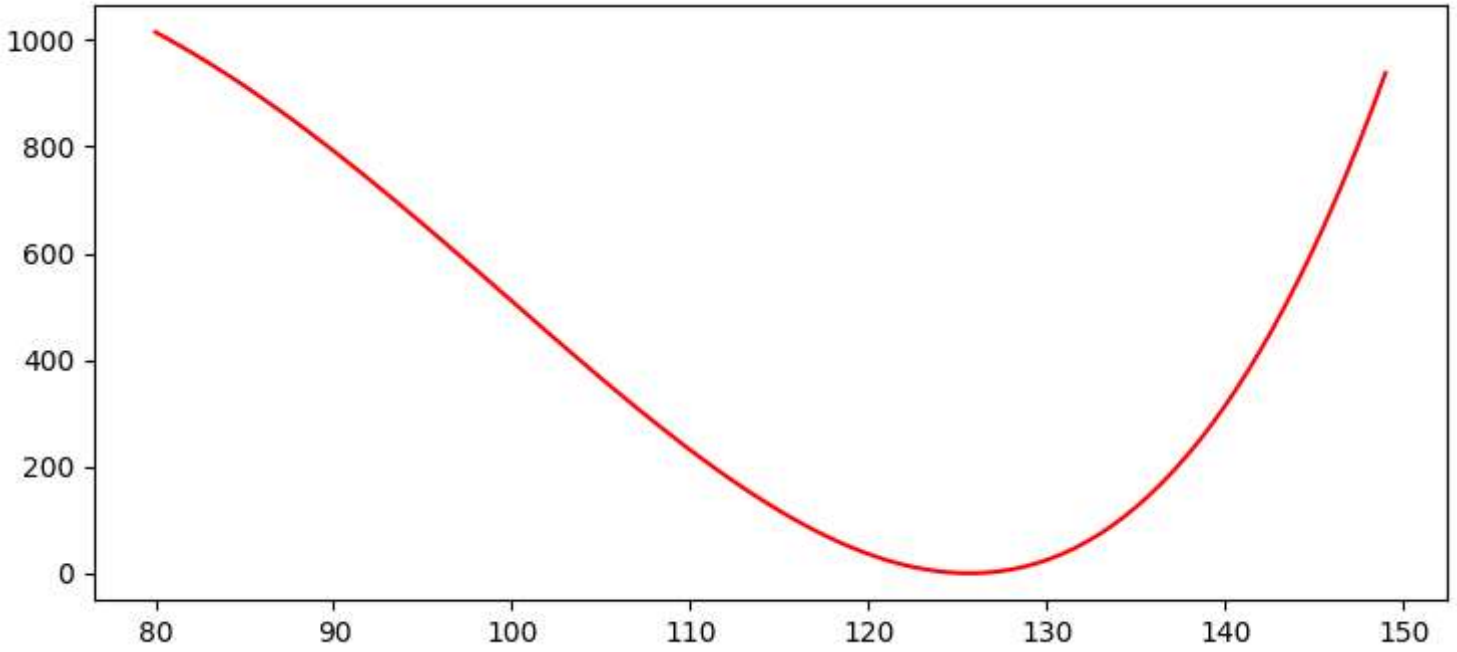
$$f_2(x) = \frac{sin(\frac{x}{10})}{x} \quad and \quad x \in [80, 250]$$

In [2]:
```python
import math
def f2(x):
    # (1 point)
    # Your code here
    need1 = np.sin(x/10)
    f = need1/x
    return f
```

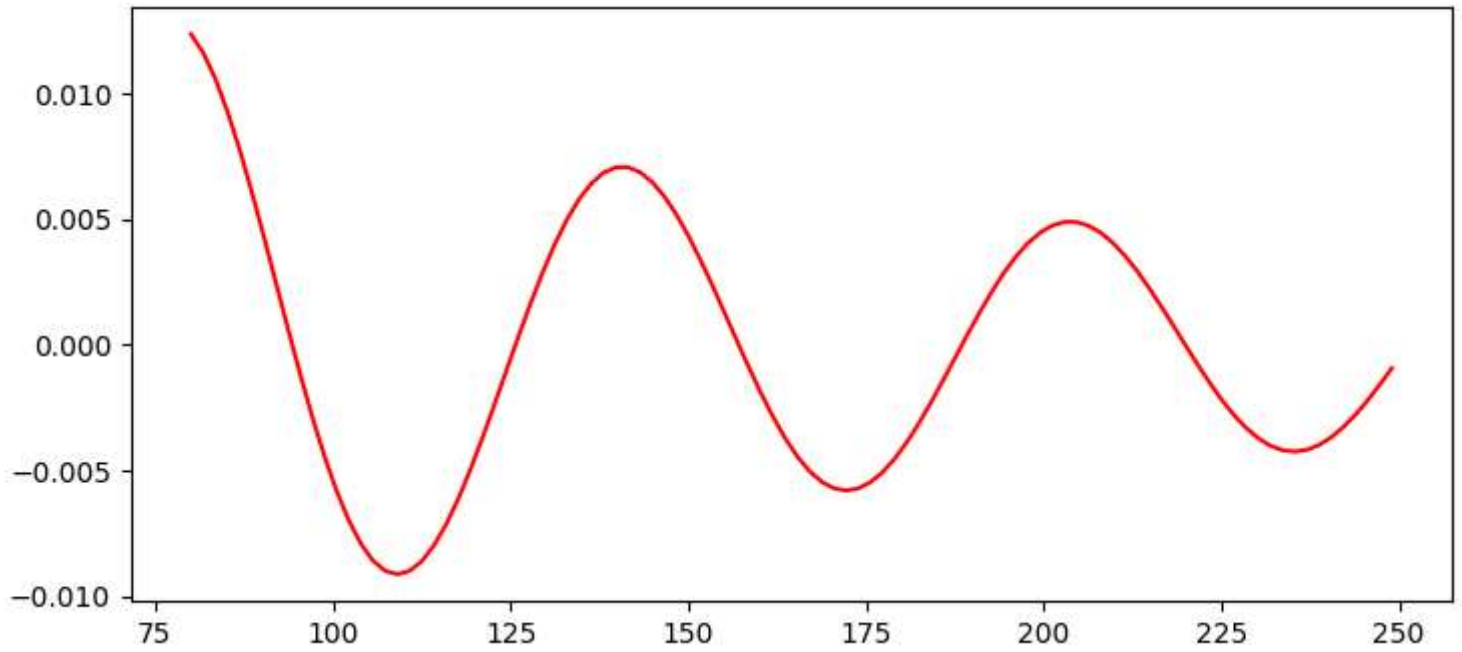
Complete the following function to draw a one-variable function and run the cells below it:

In [4]:
```python
import numpy as np
from matplotlib import pyplot as plt
def draw_one_var(func, x_range):
    # (3 points)
    # Your code here
    plt.rcParams["figure.figsize"] = [7.50, 3.50]
    plt.rcParams["figure.autolayout"] = True
    x = np.linspace(x_range[0], x_range[-1], 100)
    plt.plot(x, func(x), color='red')
    plt.show()
```

In [179]:
```
draw_one_var(f1, range(80, 150))
```



In [256]:
```
draw_one_var(f2, range(80, 250))
```



Now we want to choose a proper function and perform gradient descent on it and find the minimum. Which funtion is a good choice for this? Why? (3 point)

Your answer: function 1 is good choice because in gradient descent algorithm we are trying to find local min of a function but function 2 has more than 1 local min so it's not good.

Complete the function below and use it to find the minimum of your chosen function:

In [15]:
```python
def one_var_gradient_descent(function, initial_point, learning_rate = 0.1, max_iters = 1000):
    # (6 points)
    x = initial_point
    for i in range(max_iters):
        h = 10**(-6)
        grad = (function(x+h)-function(x))/h
        x = x - learning_rate * grad
    # Your code here
    y = function(x)
    return y, x
```

In [16]:
```python
# Use one_var_gradient_descent function and find the minimum point: (2 points)
# Your code here
print(one_var_gradient_descent(f1,130))
print(one_var_gradient_descent(f1,120))
print(one_var_gradient_descent(f1,110))
```

```
(0.0, 125.66370588706319)
(0.0, 125.66370534598069)
(0.0, 125.66370541793096)
```

Does it correspond to the plot of function? (1 point)

Your answer: Yes we can see that by checking the value of function with the plot that near point x = 125 is minimum of function

Two-variable functions

Now we want to do the same process for two-variable functions. Consider the following function and implement it in the cell below it:

Function 3:

$$f_3(x) = 5x^2 + 2y^2 - xy \quad and \quad x \in [-1, 1] \quad , y \in [-1, 1]$$

In [2]:
```python
def f3(x, y):
    # (1 point)
    # Your code here
    need1 = 5*x*x
    need2 = 2*y*y
    need3 = x*y
    f = need1 + need2 - need3
    return f
```

Complete the function below that performs two-dimensional gradient descent and returns the x and y sequence of the path from the initial point to the minimum:

In [15]:
```python
def two_var_gradiant_descent(function, initial_point, learning_rate=0.01, max_iterations=1000):
    # (12 points)
    x_sequence = [initial_point[0]]
    y_sequence = [initial_point[1]]
    x = initial_point[0]
    y = initial_point[1]
    for i in range(max_iterations):
        epsilon = 10**-6
        # Your code here
        gradx = (function(x+epsilon,y)-function(x,y))/epsilon
        x = x - learning_rate * gradx
        grady = (function(x,y+epsilon)-function(x,y))/epsilon
        y = y - learning_rate * grady
        x_sequence.append(x)
        y_sequence.append(y)


    return x_sequence, y_sequence
```

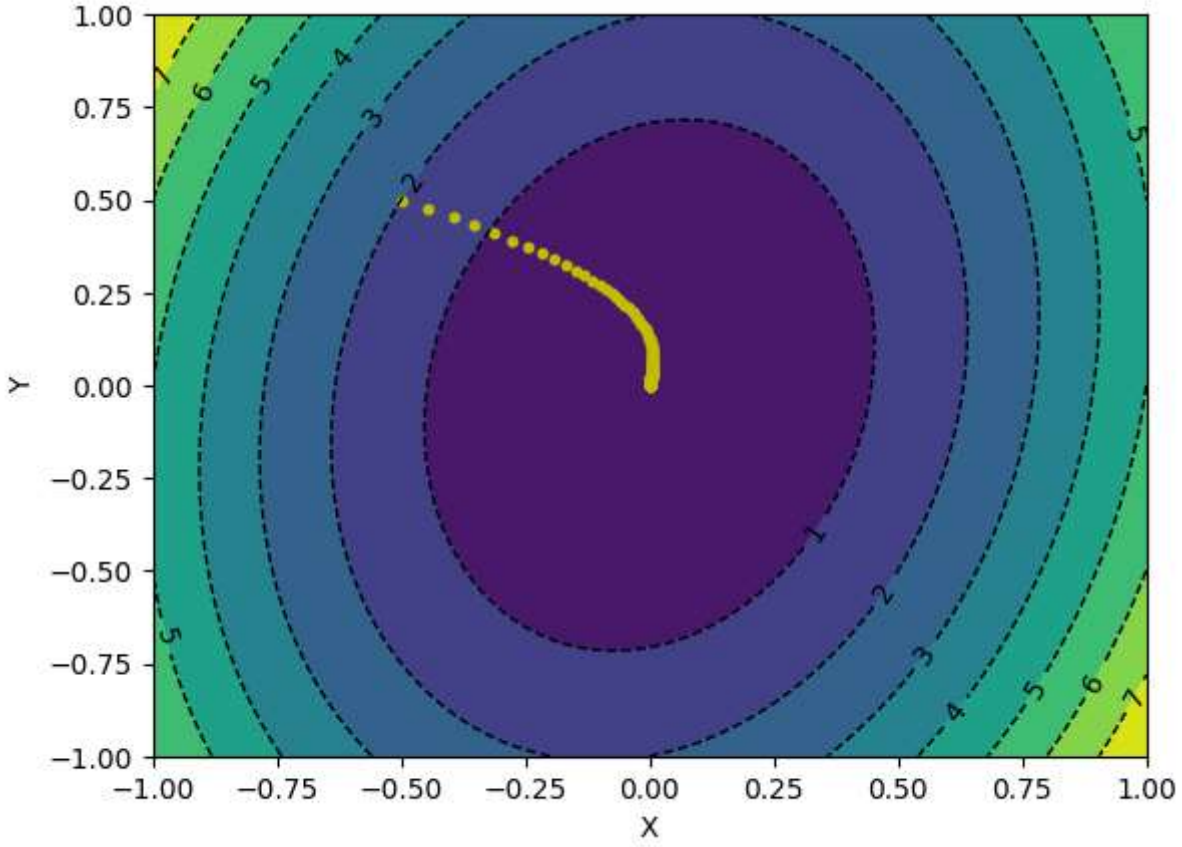
Now we use the below function to visualize the calculated path of two_var_gradiant_descent:

In [16]:
```python
def draw_points_sequence(func, x_sequence, y_sequence):
    X, Y = np.meshgrid(np.linspace(-1.0, 1.0, 100), np.linspace(-1.0, 1.0, 100))
    Z = func(X, Y)
    cp = plt.contour(X, Y, Z, colors='black', linestyles='dashed', linewidths=1)
    plt.clabel(cp, inline=1, fontsize=10)
    cp = plt.contourf(X, Y, Z, )
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.scatter(x_sequence, y_sequence, s=10, c="y")
    plt.show()
```

Find the minimum of the function 3 using two_var_gradiant_descent and then visualize the path with draw_points_sequence for the given parameters:

- initial_point = (-0.5, 0.5)
- learning_rate = 0.01
- max_iterations = 1000

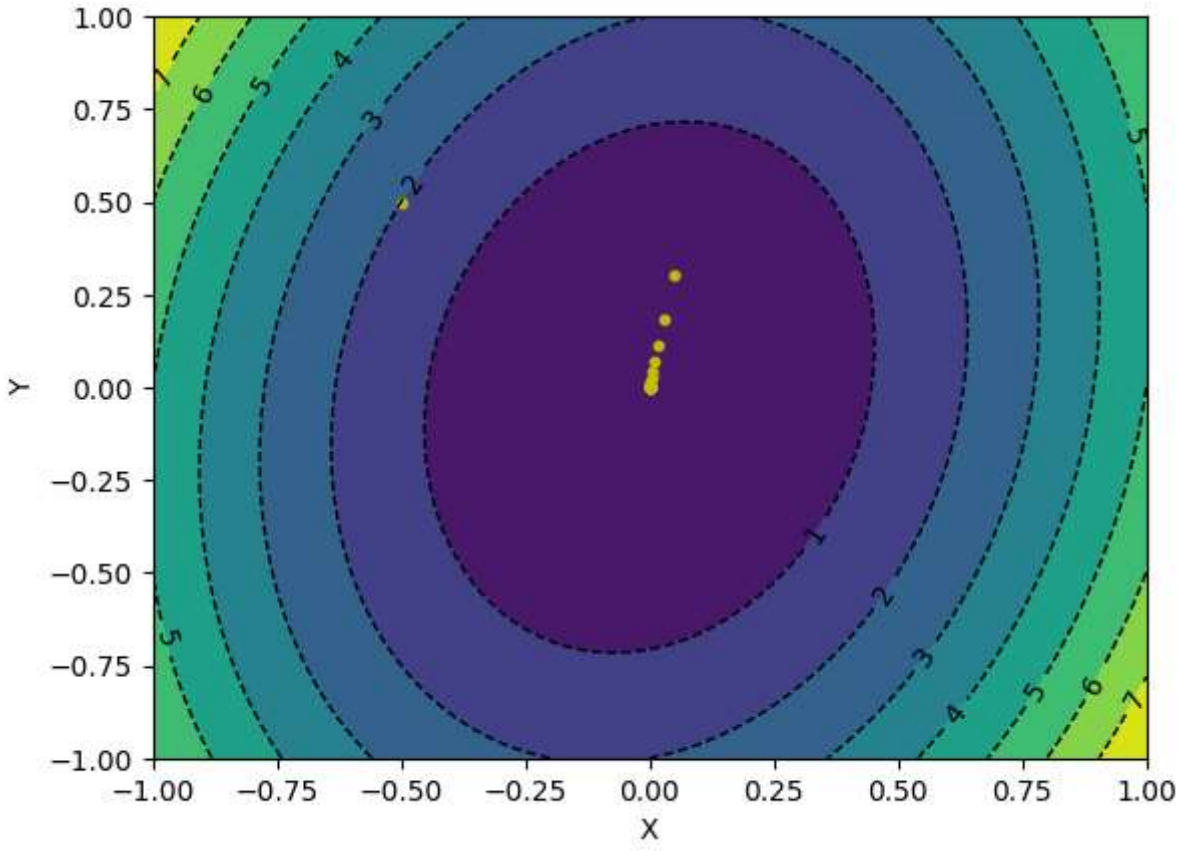
In [17]:
```
# Your code here (2 points)
seqx, seqy = two_var_gradiant_descent(f3,(-0.5,0.5))
draw_points_sequence(f3,seqx,seqy)
```



Find the minimum of the function 3 using two_var_gradiant_descent and then visualize the path with draw_points_sequence for the given parameters:

- initial_point = (-0.5, 0.5)
- learning_rate = 0.1
- max_iterations = 1000

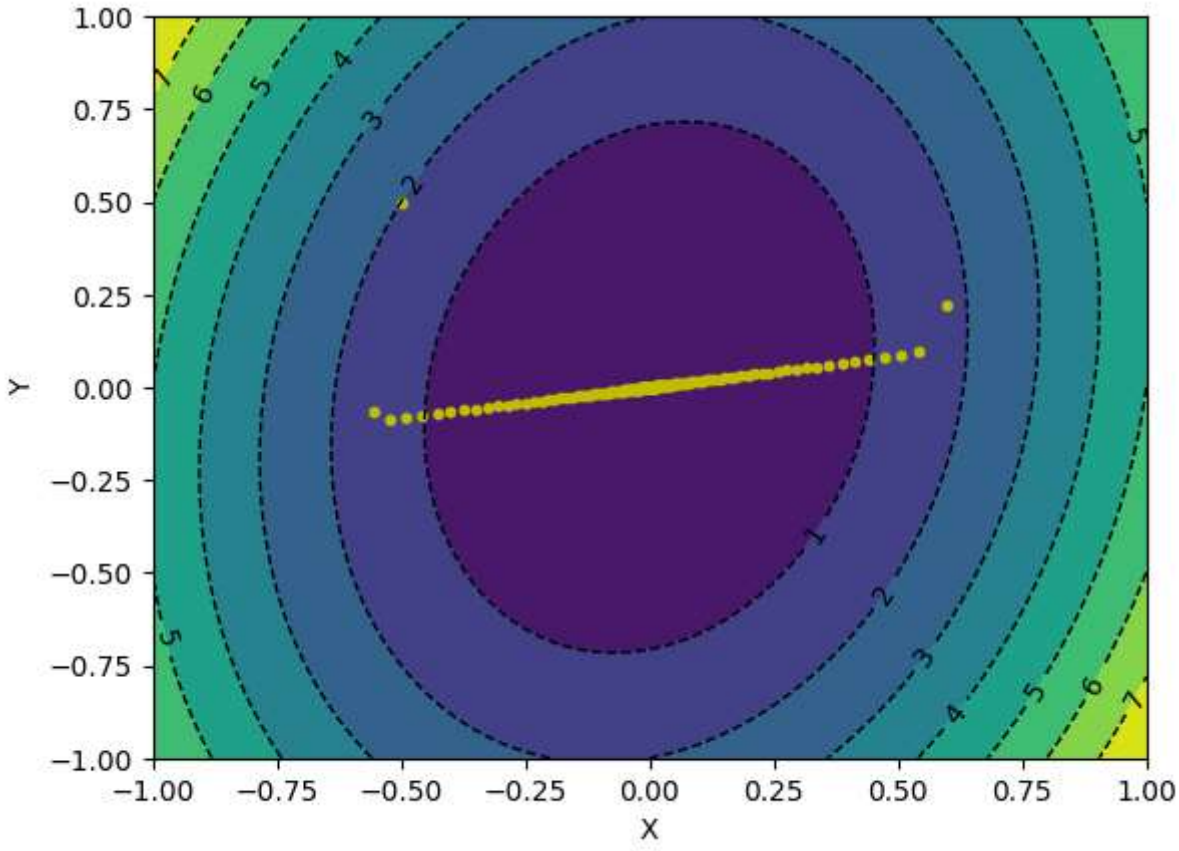
In [18]:
```
# Your code here (2 points)
seqx, seqy = two_var_gradiant_descent(f3,(-0.5,0.5),0.1)
draw_points_sequence(f3,seqx,seqy)
```



Find the minimum of the function 3 using two_var_gradiant_descent and then visualize the path with draw_points_sequence for the given parameters:

- initial_point = (-0.5, 0.5)
- learning_rate = 0.2
- max_iterations = 1000

In [19]:
```
# Your code here (2 points)
seqx, seqy = two_var_gradiant_descent(f3,(-0.5,0.5),0.2)
draw_points_sequence(f3,seqx,seqy)
```



Explain your observation from these three plots. What can we say about the learning rate? (5 points)

Your answer: We can say that by increasing the learning rate from 0.01 to 0.1 we are increasing the rate of reaching the minimum point but after change of 0.1 to 0.2, we are playing back and fourth within the minimum point until we reach it