

**Mini Project**

**Course Title: Artificial Intelligence**

**Course Code:** **CSE 366**

**Semester: Fall 2021**

**Section No:** **01**

**Submitted By**

**Name: Syeda Tamanna Sheme**

**ID:2018-2-60-010**

**Submitted To**

**Md Al-Imran**

**Lecturer**

**Department of Computer Science & Engineering**

**Date of Submission:** **13 January, 2021**

# **Mini Project**

**Question-01:**

Write a program in Python to calculate income tax. Bear in mind, the program will take input from the user. The user may provide any kind of input. It is your responsibility to handle unexpected inputs with appropriate process. Please give attention to the following data-

Income Tax

First 300k Taka 0%

Next 100k Taka 5%

Next 300k Taka 10%

Next 400k Taka 15%

Next 500k Taka 20%

Rest of ALL 25%

Some special cases=> Women and Citizens with age > 65, tax for First 350K is 0%. For Disabled => First 450K carries 0% tax. Parents of disabled pay 0% tax on First 350K. And wounded freedom fighters pay 0% on first 475K Taka.

**Solution:**

**print('Type of Tax Category: ')  
print('1-General case')  
print('2-Women and Citizens whose age is greater than 65')  
print('3-Disabled')  
print('4-Parents of disabled')  
print('5-Wounded Freedom Fighter')  
  
*# console input of option*n = int(input('Enter option : '))  
*# console input of income*try:  
 income = int(input('Enter the income(k) : '))  
except ValueError:  
 print("Please Enter Appropriate Value !!")  
else:  
 print ('No exception occurred')  
  
if (n == 1):  
 if (income <= 300):  
 tax = 0  
 elif (income > 300 and income <= 400):  
 tax = (income - 300) \* 5 / 100  
 elif (income > 400 and income <= 700):  
 tax = (300 \* 0 / 100) + (100 \* 5 / 100) + (income - 400) \* 10 / 100  
 elif (income > 700 and income <= 1100):  
 tax = (300 \* 0 / 100) + (100 \* 5 / 100) + (300 \* 10 / 100) + (income - 700) \* 15 / 100  
 elif (income > 1100 and income <= 1600):  
 tax = (300 \* 0 / 100) + (100 \* 5 / 100) + (300 \* 10 / 100) + (400 \* 15 / 100) + (income - 1100) \* 20 / 100  
 else:  
 tax = (300 \* 0 / 100) + (100 \* 5 / 100) + (300 \* 10 / 100) + (400 \* 15 / 100) + (500 \* 20 / 100) + (  
 income - 1600) \* 25 / 100  
  
elif (n == 2):  
 if (income <= 350):  
 tax = 0  
 elif (income > 350 and income <= 450):  
 tax = (income - 350) \* 5 / 100  
 elif (income > 450 and income <= 750):  
 tax = (350 \* 0 / 100) + (100 \* 5 / 100) + (income - 450) \* 10 / 100  
 elif (income > 750 and income <= 1150):  
 tax = (350 \* 0 / 100) + (100 \* 5 / 100) + (300 \* 10 / 100) + (income - 750) \* 15 / 100  
 elif (income > 1150 and income <= 1650):  
 tax = (350 \* 0 / 100) + (100 \* 5 / 100) + (300 \* 10 / 100) + (400 \* 15 / 100) + (income - 1150) \* 20 / 100  
 else:  
 tax = (350 \* 0 / 100) + (100 \* 5 / 100) + (300 \* 10 / 100) + (400 \* 15 / 100) + (500 \* 20 / 100) + (  
 income - 1650) \* 25 / 100  
  
elif (n == 3):  
 if (income <= 450):  
 tax = 0  
 elif (income > 450 and income <= 550):  
 tax = (income - 450) \* 5 / 100  
 elif (income > 550 and income <= 850):  
 tax = (450 \* 0 / 100) + (100 \* 5 / 100) + (income - 550) \* 10 / 100  
 elif (income > 850 and income <= 1250):  
 tax = (450 \* 0 / 100) + (100 \* 5 / 100) + (300 \* 10 / 100) + (income - 850) \* 15 / 100  
 elif (income > 1250 and income <= 1750):  
 tax = (450 \* 0 / 100) + (100 \* 5 / 100) + (300 \* 10 / 100) + (400 \* 15 / 100) + (income - 1250) \* 20 / 100  
 else:  
 tax = (450 \* 0 / 100) + (100 \* 5 / 100) + (300 \* 10 / 100) + (400 \* 15 / 100) + (500 \* 20 / 100) + (  
 income - 1750) \* 25 / 100**

**elif (n == 4):  
 if (income <= 350):  
 tax = 0  
 elif (income > 350 and income <= 450):  
 tax = (income - 350) \* 5 / 100  
 elif (income > 450 and income <= 750):  
 tax = (350 \* 0 / 100) + (100 \* 5 / 100) + (income - 450) \* 10 / 100  
 elif (income > 750 and income <= 1150):  
 tax = (350 \* 0 / 100) + (100 \* 5 / 100) + (300 \* 10 / 100) + (income - 750) \* 15 / 100  
 elif (income > 1150 and income <= 1650):  
 tax = (350 \* 0 / 100) + (100 \* 5 / 100) + (300 \* 10 / 100) + (400 \* 15 / 100) + (income - 1150) \* 20 / 100  
 else:  
 tax = (350 \* 0 / 100) + (100 \* 5 / 100) + (300 \* 10 / 100) + (400 \* 15 / 100) + (500 \* 20 / 100) + (  
 income - 1650) \* 25 / 100  
  
elif (n == 5):  
 if (income <= 475):  
 tax = 0  
 elif (income > 475 and income <= 575):  
 tax = (income - 475) \* 5 / 100  
 elif (income > 575 and income <= 875):  
 tax = (475 \* 0 / 100) + (100 \* 5 / 100) + (income - 575) \* 10 / 100  
 elif (income > 875 and income <= 1275):  
 tax = (475 \* 0 / 100) + (100 \* 5 / 100) + (300 \* 10 / 100) + (income - 875) \* 15 / 100  
 elif (income > 1275 and income <= 1775):  
 tax = (475 \* 0 / 100) + (100 \* 5 / 100) + (300 \* 10 / 100) + (400 \* 15 / 100) + (income - 1275) \* 20 / 100  
 else:  
 tax = (475 \* 0 / 100) + (100 \* 5 / 100) + (300 \* 10 / 100) + (400 \* 15 / 100) + (500 \* 20 / 100) + (  
 income - 1775) \* 25 / 100  
  
print('Total Tax to Pay ', tax, 'Taka')**

**Output:**

**Type of Tax Category:**

**1-General case**

**2-Women and Citizens whose age is greater than 65**

**3-Disabled**

**4-Parents of disabled**

**5-Wounded Freedom Fighter**

**Enter option : 1**

**Enter the income(k) : 450**

**Total Tax to Pay 10.0 Taka**

**Enter option : 2**

**Enter the income(k) : 750**

**Total Tax to Pay 35.0 Taka**

**Enter option : 3**

**Enter the income(k) : 1050**

**Total Tax to Pay 65.0 Taka**

**Enter option : 4**

**Enter the income(k) : 350**

**Total Tax to Pay 0 Taka**

**Enter option : 5**

**Enter the income(k) : 1560**

**Total Tax to Pay 152.0 Taka**

**Enter option : 1**

**Enter the income(k) : abc12**

**Please Enter Appropriate Value !!**

**NameError: name 'income' is not defined**

**Question-02:**

**Conics:** Give me the following graphs, include your code in the lab report. Describe the code also. Put the equations and name of the curve in your plot as well as report.

**Solution:**

The figure given above is of a Hyperbola:

**Hyperbola:**

Everything has a curve that belongs to the curves of conic sections. There are four types of conic sections – circles, parabola, ellipse, and hyperbola. A hyperbola is defined as a curve that is made of all the sets of points the difference of whose distances from the two fixed points in the plane is constant. The figure is the vertical hyperbola its equation has the form:

**-**

**Description:**

I first import numpy and matplotlib before plotting the hyperbola. After that, I ran from -200 to 200 to make the grid with x and y values. Then I use meshgrid to get the x,y values. The graph is then plotted on the X-axis and Y-axis.

Then I calculated a and b values using the hyperbolic equation. As per my plotting equation, a=49 and b=32. Finally, I use plt.contour(x, y,(y\*\*2/a\*\*2 - x\*\*2/b\*\*2),[1]) to display a hyperbolic equation and give it a title. The plot is then displayed using plt.show ().

The equation I plot is:

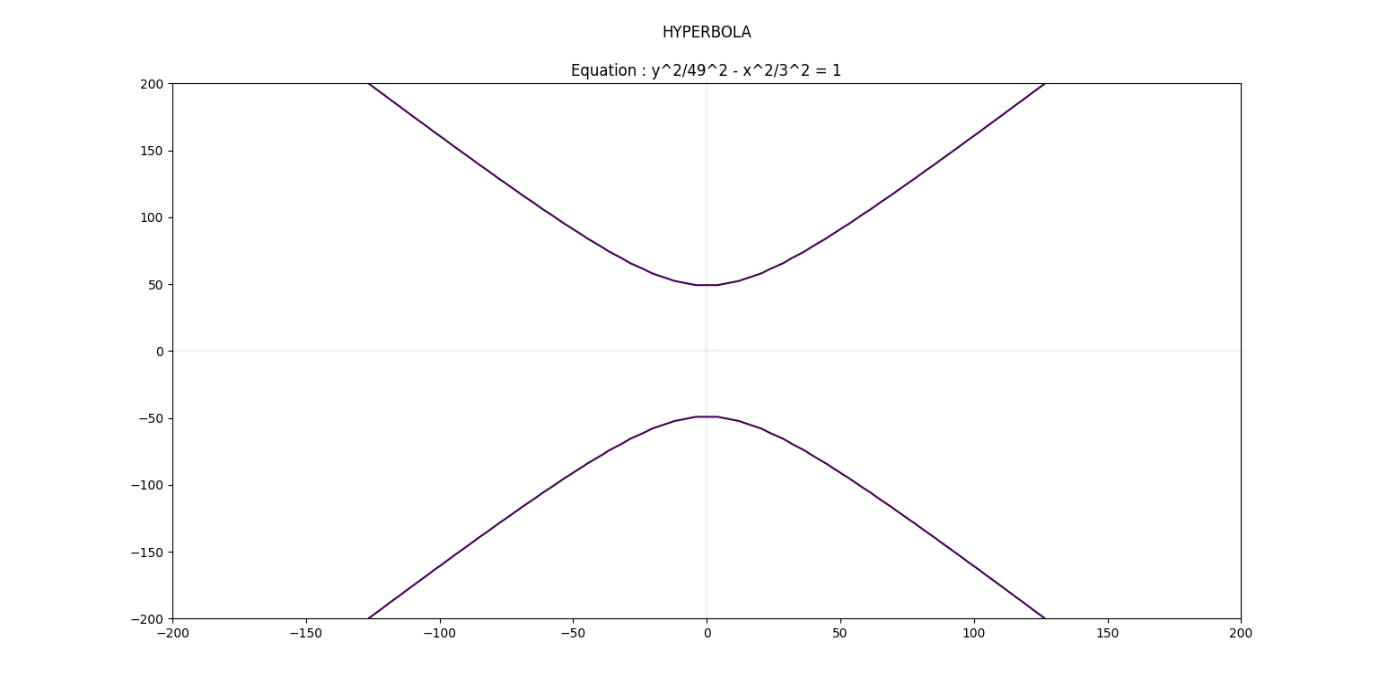
**-**

**Code:**

**import numpy as np  
import matplotlib as mpl  
import matplotlib.pyplot as plt**

***#first creating the grid with x values and y values running from -200 to 200*x = np.linspace(-200, 200)  
y = np.linspace(-200, 200)  
*#get x,y values from the grid*x, y = np.meshgrid(x, y)  
  
*#plot the x and Y axis*plt.axhline(0, alpha=.1)  
plt.axvline(0, alpha=.1)  
  
*#set a and b value according to the hyperbolic equation*a =49  
b =32  
*#plot hyperbolic equation*plt.contour(x, y,(y\*\*2/a\*\*2 - x\*\*2/b\*\*2) ,[1])  
*#add title*plt.title("HYPERBOLA\n\nEquation : y^2/49^2 - x^2/3^2 = 1")  
*#display the plot*plt.show()**

**Output:**



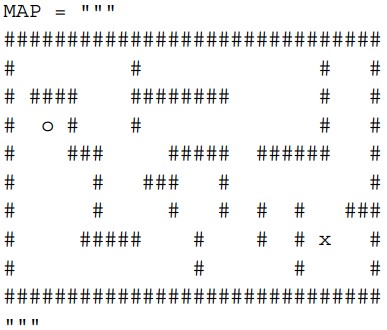
**Figure: Hyperbola**

**Question-03:**

As an AI engineer you are asked to design a robot which will be able to solve any kind of maze. Now build the maze solver code and simulate the code on your preferred IDE for Python. In this regard, you are also asked to solve it using both informed search and uninformed search techniques. Also provide a comparative code analysis showing the differences among the techniques.

**Instructions: You must have to apply the search code you have written in the lab classes. Do not apply library functions for the search algorithms. Online solution is not accepted as plagiarism is strictly forbidden. But you are encouraged to study from quality materials. You may also follow any acceptable mechanism to build the following maze and to find the output.**

**Hints: MAP should be a variable as follows.**



**Where moving diagonal path cost = 1.7 and moving regular path cost = 1.0**

**Consider Euclidean distance for computing the heuristic values.**

**Solution:**

**Uninformed Search:**

**import time  
class Maze(object):  
  
 def \_\_init\_\_(self, grid, location):  
 *"""Instances differ by their current agent locations."""* self.grid = grid  
 self.location = location  
   
 def display(self):  
 *"""Print the maze, marking the current agent location."""* for r in range(len(self.grid)):  
 for c in range(len(self.grid[r])):  
 if (r, c) == self.location:  
 print ('#')  
 else:  
 print (self.grid[r][c])  
 print  
 print  
  
 def moves(self):  
  
def neighbor(self, move):  
class Agent(object):  
  
 def bfs(self, maze, goal):def main():  
  
 grid = """  
 ##############################  
 # # # #  
 # #### ######## # #  
 # o # # # #  
 # ### ##### ###### #  
 # # ### # #  
 # # # # # # ###  
 # ##### # # # x #  
 # # # #  
 ##############################  
 """  
  
 maze = Maze(grid, (1,1))  
 maze.display()  
  
 agent = Agent()  
 goal = Maze(grid, (19,18))  
 path = agent.bfs(maze, goal)  
  
 while path:  
 move = path.pop(0)  
 maze = maze.neighbor(move)  
 time.sleep(0.25)  
 maze.display()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()**

**Output:**

**##############################**

**# # # #**

**# #### ######## # #**

**# o # # # #**

**# ### ##### ###### #**

**# # ### # #**

**# # # # # # ###**

**# ##### # # # x #**

**# # # #**

**##############################**

**##############################**

**# # # #**

**# #### ######## # #**

**# o # # # #**

**# ·### ##### ###### #**

**# · # ### # ···· #**

**# · # # ··# ·# #· ###**

**# ·##### ·# ·· # # x #**

**# ········ # # #**

**##############################**

**Informed Search Method:**

**import math  
*# Class containing the methods to solve the maze*class MazeSolver(SearchProblem):  
 *# Initialize the class* def \_\_init\_\_(self, board):  
 self.board = board  
 self.goal = (0, 0)  
  
 for y in range(len(self.board)):  
 for x in range(len(self.board[y])):  
 if self.board[y][x].lower() == "o":  
 self.initial = (x, y)  
 elif self.board[y][x].lower() == "x":  
 self.goal = (x, y)  
  
 super(MazeSolver, self).\_\_init\_\_(initial\_state=self.initial)  
  
 *# Define the method that takes actions  
 # to arrive at the solution* def actions(self, state):  
 actions = []  
 for action in COSTS.keys():  
 newx, newy = self.result(state, action)  
 if self.board[newy][newx] != "#":  
 actions.append(action)  
  
 return actions  
  
 *# Update the state based on the action* def result(self, state, action):  
 x, y = state  
  
 if action.count("up"):  
 y -= 1  
 if action.count("down"):  
 y += 1  
 if action.count("left"):  
 x -= 1  
 if action.count("right"):  
 x += 1  
  
 new\_state = (x, y)  
  
 return new\_state**

***# Check if we have reached the goal* def is\_goal(self, state):  
 return state == self.goal  
  
 *# Compute the cost of taking an action* def cost(self, state, action, state2):  
 return COSTS[action]  
  
 *# Heuristic that we use to arrive at the solution* def heuristic(self, state):  
 x, y = state  
 gx, gy = self.goal  
  
 return math.sqrt((x - gx) \*\* 2 + (y - gy) \*\* 2)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 *# Define the map* MAP = """  
 ##############################  
 # # # #  
 # #### ######## # #  
 # o # # # #  
 # ### ##### ###### #  
 # # ### # #  
 # # # # # # ###  
 # ##### # # # x #  
 # # # #  
 ##############################  
 """  
  
 *# Convert map to a list* print(MAP)  
 MAP = [list(x) for x in MAP.split("\n") if x]  
  
 *# Define cost of moving around the map* cost\_regular = 1.0  
 cost\_diagonal = 1.7  
  
 *# Create the cost dictionary* COSTS = {  
 "up": cost\_regular,  
 "down": cost\_regular,  
 "left": cost\_regular,  
 "right": cost\_regular,  
 "up left": cost\_diagonal,  
 "up right": cost\_diagonal,  
 "down left": cost\_diagonal,  
 "down right": cost\_diagonal,  
 }  
  
 *# Create maze solver object* problem = MazeSolver(MAP)  
  
 *# Run the solver* result = astar(problem, graph\_search=True)  
  
 *# Extract the path* path = [x[1] for x in result.path()]  
  
 *# Print the result* print()  
 for y in range(len(MAP)):  
 for x in range(len(MAP[y])):  
 if (x, y) == problem.initial:  
 print('o', end='')  
 elif (x, y) == problem.goal:  
 print('x', end='')  
 elif (x, y) in path:  
 print('·', end='')  
 else:  
 print(MAP[y][x], end='')  
  
 print()**

**Output:**

**##############################**

**# # # #**

**# #### ######## # #**

**# o # # # #**

**# ### ##### ###### #**

**# # ### # #**

**# # # # # # ###**

**# ##### # # # x #**

**# # # #**

**##############################**

**##############################**

**# # # #**

**# #### ######## # #**

**# o # # # #**

**# ·### ##### ###### #**

**# · # ### # ···· #**

**# · # # ··# ·# #· ###**

**# ·##### ·# ·· # # x #**

**# ········ # # #**

**##############################**

**An analysis of the differences between the techniques using comparison code:**

 Maze Runner game is a game that requires a pathfinding algorithm to get to the destination with the shortest path. This algorithm is used in an  Non Player Character that will move from the start node to the destination node. However, the use of incorrect algorithms can affect the length of the computing process to find the shortest path. The longer the computing process, the longer the players have to wait. This study compared pathfinding algorithms A \*, and Breadth-First Search (BFS) in the Maze Runner game.

Uniform Cost Searches are a subset of Breadth First Searches (UCS). A weighted graph search is what UCS is. Cost storage is used to decide the order in which nodes are visited. The purpose of BFS is to get from the source vertex to the goal vertex. Starting with the source vertex, some Vertex in the visited set visits the entire layer of unvisited vertexes, and recently visited vertexes are added to the visited set. By viewing the vertices as labyrinth cells, BFS can be compared to the Micro mouse maze. The BFS method in Micro mouse labels cells by searching from the start cell to all nearby neighbours. "Zero" is written in the first cell (0). The program keeps track of which cells are directly adjacent to the start cell. BFS is capable of determining the shortest route. The search will continue until the target is found. he advantage of BFS is that it will stop searching as soon as it finds the shortest path, with the drawback being that memory accesses are more scattered and play less well with caches.

We utilize the Data Structure Priority Queue to implement the A\* algorithm since we need to find the cell with the lowest cost. Unlike a FIFO (First In First Out) queue, the elements in a Priority Queue are removed according to their priority. The element's value may be given precedence (highest or lowest). The Priority Queue is included in Python's Queue module, and the priority is the lowest number, making it ideal for implementing A\*. The fact that it's beating A\* is likely due to the overhead associated with A\* not being worth the savings in spaces explored.

In Maze Runner Game, A\* and Breadth First Search can be utilized to determine the shortest path. A\* is the best pathfinding algorithm, especially for Maze game puzzles. This is backed up by the fact that only a small amount of computational power is required and that the search time is quite short. In terms of computing process, memory utilization, and computing time, the appropriate method can improve the game.

**Question-04:**

In the logic program, we specify the puzzle as follows:

* Steve has a blue car.
* The person who owns a cat live in Canada. Matthew lives in the USA.
* The person with a black car lives in Australia.
* Jack has a cat.
* Alfred lives in Australia.
* The person who has a dog live in France.
* Who has a rabbit?

The goal is the find the person who has a rabbit. Here are the full details about the four people as puzzle solver input data:



**Solution:**

**from kanren import \***

**from kanren.core import lall**

***# Declare the variable***

**people = var()**

***# Define the rules***

**rules = lall(**

***# There are 4 people***

**(eq, (var(), var(), var(), var()), people),**

***# Steve's car is blue***

**(membero, ('Steve', var(), 'blue', var()), people),**

***# Person who has a cat lives in Canada***

**(membero, (var(), 'cat', var(), 'Canada'), people),**

***# Matthew lives in USA***

**(membero, ('Matthew', var(), var(), 'USA'), people),**

***# The person who has a black car lives in Australia***

**(membero, (var(), var(), 'black', 'Australia'), people),**

***# Jack has a cat***

**(membero, ('Jack', 'cat', var(), var()), people),**

***# Alfred lives in Australia***

**(membero, ('Alfred', var(), var(), 'Australia'), people),**

***# Person who owns the dog lives in France***

**(membero, (var(), 'dog', var(), 'France'), people),**

***# Who has a rabbit?***

**(membero, (var(), 'rabbit', var(), var()), people)**

**)**

***# Run the solver***

**solutions = run(0, people, rules)**

***# Extract the output***

**output = [house for house in solutions[0] if 'rabbit' in house][0][0]**

***# Print the output***

**print('\n' + output + ' is the owner of the rabbit')**

**print('\nHere are all the details:')**

**attribs = ['Name', 'Pet', 'Color', 'Country']**

**print('\n' + '\t\t'.join(attribs))**

**print('=' \* 57)**

**for item in solutions[0]:**

**print('')**

**print('\t\t'.join([str(x) for x in item]))**

**Output:**

