MY ASSIGNMENT.

① * Find the derivation of $y = x^x$?

Sol:- To find the derivative of $y = x^x$?, we can use the logarithm -ic differentiation method.

Take the natural logarithm of both sides of the equation $y = x^x$:

$$\ln(y) = \ln(x^x)$$

using the properties of logarithms, we can simplify this to:

$$\ln(y) = x \ln(x)$$

Now, we can take the derivation of both sides with respect to x:

$$1/y \; dy/dx = \ln(x) + x(1/x)$$

Simplifying this expression:

$$dy/dx = y(\ln(x) + 1)$$

Substituting $y = x^x$ back in:

$$dy/dx = x^x (\ln(x) + 1)$$

Therefore, the derivation of $y = x^x$ is $dy/dx = x^x (\ln(x) + 1)$.

Python code:-

```
import sympy

x= sympy.symbol('x')

y= x**x

dy_dx = sympy.diff(y,x)

dy_dx_Simplified = sympy.simplify(dy_dx)

Print(dy_dx_Simplified).
```

O/P:-

```
x**x*(log(x)+1)
```

which is the derivation of $y = x^{\wedge}x$.

(2)

* Describe how Dijkstra's algorithm works?

Dijkstra's algorithm is a well known algorithm used in graph theory to find the shortest path between two nodes in a weighted graph. It works by maintaining a set of unvisited nodes and a set of tentative distances to those nodes. At the beginning of the algorithm, the starting node is set as the current node, and its tentative distance is set to zero.

1) Set the starting node as the current node and its tentative distance to zero.

2) For each neighbour of the current node. Calculate the tentative distance from the starting node to that neighbour. This is done by adding the weight of the edge between the current node and it's neighbour to the tentative distance of the current node.

3) If the tentative distance of a neighbour is less than it's current distance for it's the neighbour has not been visited yet. update the neighbour tentative distance and set it's previous node to be current node.

4) Mark the current node as visited and remove it from the unvisited set.

5) Choose the unvisited node with the smallest tentative distance and set it as the new current node. If there are no unvisited nodes, the algorithm is finished.

6) Repeat Step-2-5 until the target node is reached or there are no more unvisited nodes.

At the end of the algorithm, the states of shortest path from the starting node to the target node can be reconstructed by backtracking from the target node to it's previous node, then to the previous node of that node, and so, on, until the starting node is reached. The path can be obtained by following the previous node pointers and recording the nodes in the order they were visited.

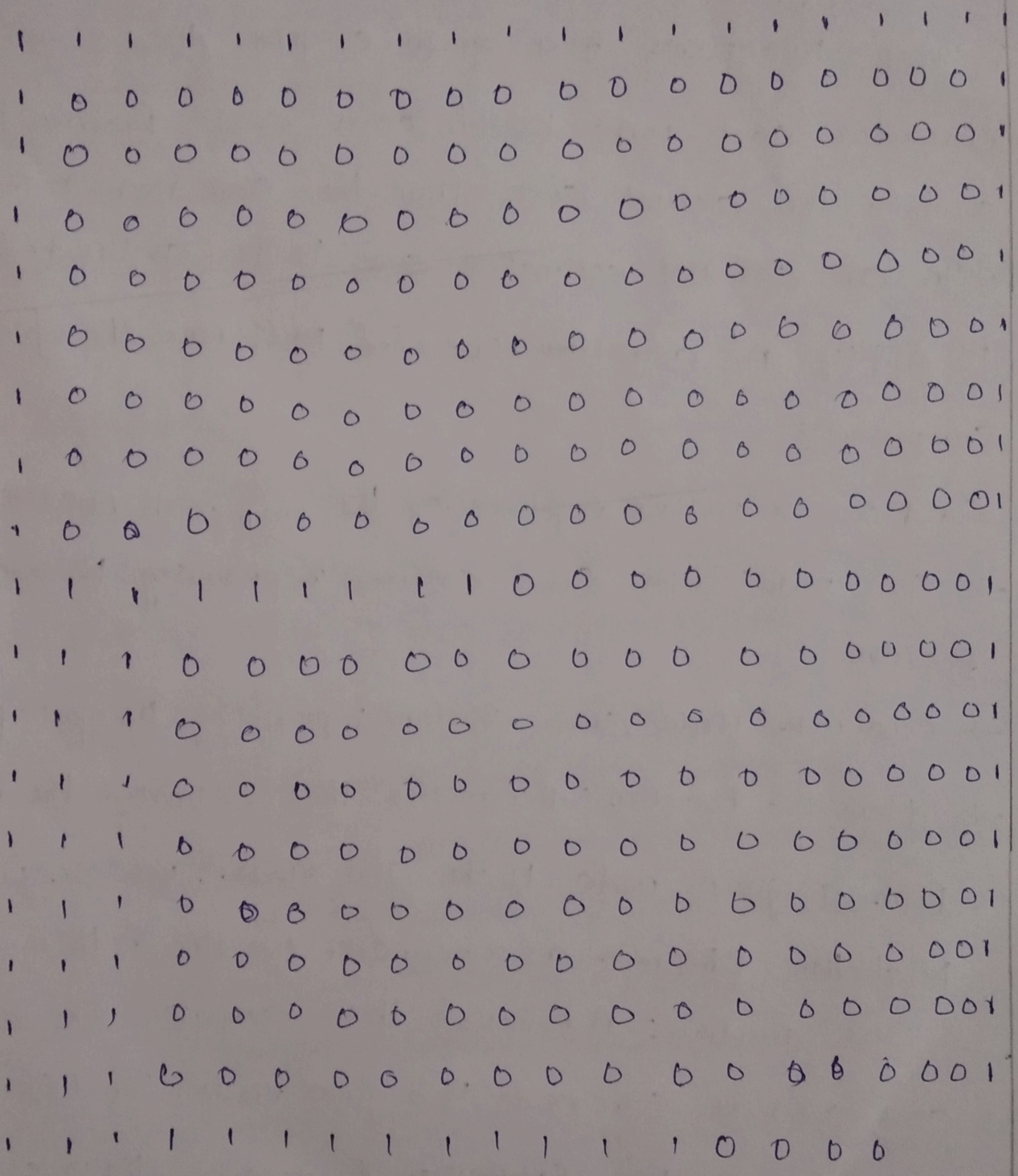
3) a) To create the 20x20 map with walls as shown in the question, you can simply initialize a 2D array in a programming language of choice and set the appropriate values to 1 to indicate the presence of obstacles, here is an example code in python:

```
"map=[[1 for i in range(20)] for j in range(20)]
for i in range(1,19):
    for j in range(1,19):
        map[i][j]=0.
"
```

This will create a 20x20 map where all the boarder elements are set to 1 and all the internal elements are set to 0.

3)b).

Here's the map with the obstacles added to it:



3) c) :- To find the shortest path between the start position and the goal position, we can use the A* Algorithm. A* is a widely used algorithm to find the shortest path between two points in a graph or grid. The algorithm evaluates each node based on a cost function that combines the actual cost from the start node to the node with an essential estimated cost to the goal node.

To apply A* algorithm, we need to define the following components:

→ Grid 20x20 matrix representing the map with obstacles.

→ Start node: tuple (row, column) representing the start position.

→ Goal node: tuple (row, column) representing the goal position.

→ Heuristic function: A function that estimates the distance from a given node to the goal node. We can use the Manhattan distance as a heuristic function in this case.

→ Cost function: A function that computes the actual cost and the start node to a given node. In this case, we can use the ~~Evaluation~~ Euclidean distance as a cost function.

Here's the implementation of A* algorithm to find the shortest path between the start and goal positions for all three sets.

Code:-

```
import math
import heapq.

# Define the grid.
grid =[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
      [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
      [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
      [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
      [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
      [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
      [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
      [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
      [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
      [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
```

3)d) Assuming that the robot initially faces down, we can define the following movement commands:

→ F: Move forward one cell in the direction that the robot is facing.

→ L: turn left 90 degrees.

→ R: turn right 90 degrees.

→ B: turn around 180 degrees.

Here's an example python code to generate the list of movement controls for the first step of start and goal points:

Code:-

```
Moves = {
    (0,1): "F",
    (0,-1): "B",
    (1,0): "R",
    (-1,0): "L"
}

start = (2,2)
goal = (2,11)

commands = [ ]
current_direction = (0,1)
for i in range (len(shortest_path) -1):
    dx = Shortest_Paths[i+1][0] - Shortest_Paths[i][0].
    dy = shortest_path[i+1][1] - shatest_Paths[i][1].
new_direction = (dx,dy)
if new_direction != current_direction:
```

3d

```
angle = get_angle (current_direction, new_direction)
    if angle == 90:
        commands.append ("L")
    elif angle == -90:
        commands.append ("R")
    elif angle == 180:
        commands.append ("B")
    current_direction = new_direction.
    commands.append ("F").

Paint (commands).
```

→ 'get_angle' function is used to calculate the angle between the current direction that the robot is facing and the new direction that it needs to face in order to move to the next cell in the path.

```
def get_angle (current_direction, new_direction):
    dot_product = current_direction[0] * new_direction[0]
                + current_direction[1] * new_direction[1].
    cross_product = current_direction[0] * new_direction[1] -
                current_direction[1] * new_direction[0].
    angle = math.atan2 (cross_product, dot_product) * 180/math.pi
    return int (angle).
```

This function uses the dot product and cross product to calculate the angle between two vectors. The angle is then converted from radians to degrees and returned as an integer.

4) Inverse Kinematics is a method of determining the joint angles required for a robotic arm to reach a desired end effector position in space. The input to the IK function is the desired end effector position and orientation, as well as the arm's physical parameters such as the lengths of the links and the initial joint angles.

The output of the IK function is the best set of joint angles required to move the arm to the desired position. The IK function typically involves solving a set of equations that relate the joint angles to the position and orientation of the end effector. There are many different approaches to solving these equations, ranging from analytical methods to numerical optimization techniques.

-g.

To write an IK function, one would need to have knowledge of the specific robotic arm being used and its physical parameters. It would also require knowledge of mathematical concepts such as vector algebra and trigonometry.