AMITY SCHOOL OF ENGINEERING & TECHNOLOGY

AMITY UNIVERSITY CAMPUS, SECTOR-125, NOIDA-201303

Soft Computing Lab

PRACTICAL FILE

COURSE CODE: CSE320

Submitted to:
Dr Bedatri Moulik

Submitted by:
Boddu Asmitha Bhavya
A2305221386
7CSE-6X

INDEX

S.No.	Experiment	Date	Signature
1.	Implement Union, Intersection, Complement and Difference operations on fuzzy sets		
2.	Implement AND, OR, NOR, NAND, XOR, and XNOR using ANN.		
3.	Predict the house price.		
4.	Implement a SINGLE POINT , MULTI POINT AND UNIFORM crossover operator in python.		
5.	Implement the Knapsack problem.		

Experiment – 1

AIM:

Implement Union, Intersection, Complement and Difference operations on fuzzy sets. Also create fuzzy relation by Cartesian product of any two fuzzy sets and perform max-min composition on any two fuzzy relations.

Theory:

Fuzzy sets are an extension of classical sets where elements have degrees of membership rather than binary membership (i.e., either in or out of the set). Each element in a fuzzy set is associated with a membership value between 0 and 1, representing the degree of membership in the set. Let's denote two fuzzy sets AAA and BBB with a universe of discourse UUU.

1. **Union of Fuzzy Sets**

The union of two fuzzy sets A and B is denoted as $A \cup B$ and defined as:

$$\mu_{A \cup B}(x) = \max(\mu_A(x), \mu_B(x))$$

This operation selects the maximum membership value between A and B for each element in the universe U.

2. Intersection of Fuzzy Sets

The intersection of two fuzzy sets A and B is denoted as $A \cap B$ and defined as:

$$\mu_{A \cap B}(x) = \min(\mu_A(x), \mu_B(x))$$

This operation selects the minimum membership value between A and B for each element in the universe U.

3. Complement of a Fuzzy Set

The complement of a fuzzy set A is denoted as A^c and defined as:

$$\mu_{A^c}(x) = 1 - \mu_A(x)$$

This operation subtracts the membership value of each element in A from 1.

4. Difference of Fuzzy Sets

The difference between two fuzzy sets A and B is denoted as $A - B$ and is defined as:

$$\mu_{A-B}(x) = \min(\mu_A(x), 1 - \mu_B(x))$$

This operation is similar to set subtraction in classical sets but is performed with fuzzy membership degrees.

5. Cartesian Product of Fuzzy Sets

The Cartesian product of two fuzzy sets A and B forms a fuzzy relation R between the elements of the two sets. The membership function of the Cartesian product is given by:

$$\mu_R(x,y) = \min(\mu_A(x), \mu_B(y))$$

Here, the fuzzy relation R represents pairs of elements from A and B, with their membership determined by the minimum of the membership values of $x \in A$ and $y \in B$.

6. Max-Min Composition of Fuzzy Relations

Given two fuzzy relations $R \subseteq X \times Y$ and $S \subseteq Y \times Z$, the max-min composition $T = R \circ S$ between them is defined as:

$$\mu_T(x,z) = \max_{y \in Y} \left(\min(\mu_R(x,y), \mu_S(y,z))\right)$$

This composition rule computes the degree of relation between $x \in X$ and $z \in Z$ by considering all intermediate elements $y \in Y$ and taking the maximum of the minimum values.

Input:

```
#Union
def union(A,B):
 A=dict()
 B=dict()
 Y=dict()

 A = {"a": 0.2, "b": 0.3, "c": 0.6, "d": 0.6}
 B = {"a": 0.9, "b": 0.9, "c": 0.4, "d": 0.5}
 print("UNION")
 print('The First Fuzzy set is:', A)
 print('The Second Fuzzy set is:', B)
 for A_key, B_key in zip(A,B):
  A_value = A[A_key]
  B_value = B[B_key]

  if A_value > B_value:
   Y[A_key]=A_value
  else:
   Y[B_key]=B_value
 print('Fuzzy Set Union is:', Y)
```

```
#Intersection
def intersection(A,B):
 A=dict()
 B=dict()
 Y=dict()
 A = {"a": 0.2, "b": 0.3, "c": 0.6, "d": 0.6}
 B = {"a": 0.9, "b": 0.9, "c": 0.4, "d": 0.5}
 print("INTERSECTION")
 print('The First Fuzzy set is:', A)
 print('The Second Fuzzy set is:', B)
 for A_key, B_key in zip(A,B):
  A_value = A[A_key]
  B_value = B[B_key]

  if A_value < B_value:
   Y[A_key]=A_value
  else:
   Y[B_key]=B_value
  print('Fuzzy Set Intersection:', Y)
```

```
#Complement
def complement(A, B):
  A=dict()
  Y=dict()
  print('COMPLEMENT')
  A = {"a": 0.2, "b": 0.3, "c": 0.6, "d": 0.6}
  print('The Fuzzy Set is:',A)
  for A_Key in A:
    Y[A_Key]=1-A[A_Key]
    print('Fuzzy set Complement is:', Y)

#Difference
def difference(A,B):
  A=dict()
  B=dict()
  Y=dict()
  A = {"a": 0.2, "b": 0.3, "c": 0.6, "d": 0.6}
  B = {"a": 0.9, "b": 0.9, "c": 0.4, "d": 0.5}
  print("DIFFERENCE")
  print('The First Fuzzy set:', A)
  print('The Second Fuzzy se:',B)
  for A_Key, B_Key in zip(A,B):
    A_value=A[A_Key]
    B_value=B[B_Key]
    B_value=1-B_value

  if A_value<B_value:
    Y[A_Key]=A_value
  else:
    Y[B_Key]=B_value
  print('Fuzzy Set Difference is:',Y)

#Cartesian Product
def cartesian():
  n=int(input("Enter number of elements in first set(A):"))
  A=[]
  B=[]
  print("CARTESIAN PRODUCT")
  print('Enter elements for A:')

  for i in range(0,n):
    ele=float(input())
    A.append(ele)
  m=int(input("\n Enter number of elements in second set(B):"))
  print("Enter elements for B:")
  for i in range(0,m):
    ele=float(input())
    B.append(ele)
  print("A={"+str(A)[1:-1]+"}")
  print("B={"+str(B)[1:-1]+"}")

  cart_prod=[]
  cart_prod=[[0 for j in range(m)]for i in range(n)]
  for i in range(n):
    for j in range(m):
      cart_prod[i][j]=min(A[i], B[j])
  print("A x B =")
  for i in range(n):
    for j in range(m):
      print(cart_prod[i][j], end=" ")
    print("\n")
  return

def main():
  while True:
    print("1.Union")
    print("2.Intersection")
    print("3.Complement")
```

```
    print("4.Difference")
    print("5.Cartesian Product")
    print("6.EXIT")
    if choice==1:
      union(A,B)
    elif choice==2:
      intersection(A,B)
    elif choice==3:
      complement(A,B)
    elif choice==4:
      difference(A,B)
    elif choice==5:
      cartesian()
    elif choice==6:
      break
    else:
      print("Wrong choice")
```

Output:

```
The First Fuzzy set is: {'a': 0.2, 'b': 0.3, 'c': 0.6, 'd': 0.6}
The Second Fuzzy set is: {'a': 0.9, 'b': 0.9, 'c': 0.4, 'd': 0.5}
Fuzzy Set Union is: {'a': 0.9, 'b': 0.9, 'c': 0.6, 'd': 0.6}

The First Fuzzy set is: {'a': 0.2, 'b': 0.3, 'c': 0.6, 'd': 0.6}
The Second Fuzzy set is: {'a': 0.9, 'b': 0.9, 'c': 0.4, 'd': 0.5}
Fuzzy Set Intersection is: {'a': 0.2, 'b': 0.3, 'c': 0.4, 'd': 0.5}

Enter number of elements in first set(A):3
CARTESIAN PRODUCT
Enter elements for A:
2
3
4

 Enter number of elements in second set(B):4
Enter elements for B:
1
4
5
2
A={2.0, 3.0, 4.0}
B={1.0, 4.0, 5.0, 2.0}
A x B =
1.0 2.0 2.0 2.0

1.0 3.0 3.0 2.0

1.0 4.0 4.0 2.0
```

Experiment-2

Aim:

Implement AND, OR, NOR, NAND, XOR, and XNOR using ANN.

Theory:

Logic gates like AND, OR, NAND, NOR, XOR, and XNOR can be modeled using Artificial Neural Networks (ANNs). A neural network for a logic gate consists of input neurons representing the binary inputs of the gate and output neurons representing the result of the logical operation. For simple gates, a single-layer perceptron (SLP) or a multi-layer perceptron (MLP) can be used.

Key Concepts:

- **Input**: Binary inputs for logic gates, typically two inputs (e.g., x1x_1x1, x2x_2x2).
- **Weights**: The parameters that determine the influence of each input on the output.
- **Activation Function**: A function used to introduce non-linearity, e.g., step function for simple gates and sigmoid for multi-layer networks.
- **Bias**: A constant value added to the weighted inputs to help the network model the gate's decision boundary.

Input:

```
import numpy as np

def unitStep(v):

  if v>=0:

    return 1

  else:

    return 0

def perceptronModel(x,w,b):

  v=np.dot(w,x)+b

  y=unitStep(v)

  return y

def AND_logicFunction(x):

  w=np.array([1,1])

  b=-1.5

  return perceptronModel(x,w,b)

#AND GATE

test1=np.array([0,1])

test2=np.array([1,1])

test3=np.array([0,0])

test4=np.array([1,0])


print("AND({},{})={}".format(0,1,AND_logicFunction(test1)))

print("AND({},{})={}".format(1,1,AND_logicFunction(test2)))

print("AND({},{})={}".format(0,0,AND_logicFunction(test3)))

print("AND({},{})={}".format(1,0,AND_logicFunction(test4)))


#OR GATE

def OR_logicFunction(x):

  w=np.array([1,1])

  b=-0.5

  return perceptronModel(x,w,b)

print("\n")

print("OR({},{})={}".format(0,1,OR_logicFunction(test1)))

print("OR({},{})={}".format(1,1,OR_logicFunction(test2)))

print("OR({},{})={}".format(0,0,OR_logicFunction(test3)))

print("OR({},{})={}".format(1,0,OR_logicFunction(test4)))


#NOT GATE

def NOT_logicFunction(x):

  wNOT = -1
```

```python
  bNOT = 0.5
  return perceptronModel(x, wNOT, bNOT)
  return NOT_logicFunction
test5 = np.array(1)
test6 = np.array(0)
print("\n")
print("NOT({})={}".format(1,
NOT_logicFunction(test5)))
print("NOT({})={}".format(0,
NOT_logicFunction(test6)))


#NOR GATE
def NOR_logicFunction(x):
  output_OR = OR_logicFunction(x)
  output_NOT = NOT_logicFunction(output_OR)
  return output_NOT
print("\n")
print("NOR({}, {}) = {}".format(0, 1,
NOR_logicFunction(test1)))
print("NOR({}, {}) = {}".format(1, 1,
NOR_logicFunction(test2)))
print("NOR({}, {}) = {}".format(0, 0,
NOR_logicFunction(test3)))
print("NOR({}, {}) = {}".format(1, 0,
NOR_logicFunction(test4)))


#NAND GATE
def NAND_logicFunction(x):
  output_AND = AND_logicFunction(x)
  output_NOT = NOT_logicFunction(output_AND)
  return output_NOT
print("\n")
print("NAND({}, {}) = {}".format(0, 1,
NAND_logicFunction(test1)))
print("NAND({}, {}) = {}".format(1, 1,
NAND_logicFunction(test2)))
print("NAND({}, {}) = {}".format(0, 0,
NAND_logicFunction(test3)))
print("NAND({}, {}) = {}".format(1, 0,
NAND_logicFunction(test4)))

#XOR GATE
def XOR_logicFunction(x):
  a,b=x
  y1 = NAND_logicFunction([a,a])
  y2 = NAND_logicFunction([b,b])
  y3 = NAND_logicFunction([a,y2])
  y4 = NAND_logicFunction([y1,b])
  finalOutput = NAND_logicFunction([y3,y4])
  return finalOutput
print("\n")
print("XOR({}, {}) = {}".format(0, 1,
XOR_logicFunction(test1)))
print("XOR({}, {}) = {}".format(1, 1,
XOR_logicFunction(test2)))
print("XOR({}, {}) = {}".format(0, 0,
XOR_logicFunction(test3)))
print("XOR({}, {}) = {}".format(1, 0,
XOR_logicFunction(test4)))


#XNOR GATE
def XNOR_logicFunction(x):
  y1 = OR_logicFunction(x)
  y2 = AND_logicFunction(x)
  y3 = NOT_logicFunction(y1)
  final_x = np.array([y2, y3])
  finalOutput = OR_logicFunction(final_x)
  return finalOutput
print("\n")
print("XNOR({}, {}) = {}".format(0, 1,
XNOR_logicFunction(test1)))
print("XNOR({}, {}) = {}".format(1, 1,
XNOR_logicFunction(test2)))
print("XNOR({}, {}) = {}".format(0, 0,
XNOR_logicFunction(test3)))
print("XNOR({}, {}) = {}".format(1, 0,
XNOR_logicFunction(test4)))
```

Output:

```
AND(0,1)=0
AND(1,1)=1
AND(0,0)=0
AND(1,0)=0

OR(0,1)=1
OR(1,1)=1
OR(0,0)=0
OR(1,0)=1

NOT(1)=0
NOT(0)=1

NOR(0, 1) = 0
NOR(1, 1) = 0
NOR(0, 0) = 1
NOR(1, 0) = 0
```

```
NAND(0, 1) = 1
NAND(1, 1) = 0
NAND(0, 0) = 1
NAND(1, 0) = 1

XOR(0, 1) = 1
XOR(1, 1) = 0
XOR(0, 0) = 0
XOR(1, 0) = 1

XNOR(0, 1) = 0
XNOR(1, 1) = 1
XNOR(0, 0) = 1
XNOR(1, 0) = 0
```

Experiment – 3

Aim:

A house with 1000 square feet(sqft) sold for $300,000 and a house with 2000 square feet sold for $500,000. These two points will constitute our *data or training set*. In this lab, the units of size are 1000 sqft and the units of price are 1000s of dollars. Predict the house price.

Size (1000 sqft)	Price (1000s of dollars)
1	300
2	500

Theory:

To predict housing prices based on house size (in square feet) using a linear regression model.

1. Dataset Overview

We will use a minimal dataset consisting of two data points:

- A house with 1000 sqft sold for $300,000.
- A house with 2000 sqft sold for $500,000.

House Size (sqft)	Price (in 1000s of dollars)
1000	300
2000	500

2. Linear Regression Model

We aim to find a linear relationship represented by the equation:

$$\text{Price} = m \times \text{Size} + b$$

Where:

- Price is the predicted price in thousands of dollars.
- Size is the size of the house in thousands of square feet.
- mmm is the slope (price increase per additional square foot).
- b is the y-intercept (predicted price for a size of 0).

3. Calculation of Parameters

Using the least squares method, we calculate slope and intercept.

4. Making Predictions

With the calculated values of mmm and bbb, we can predict housing prices for new sizes using the linear equation.

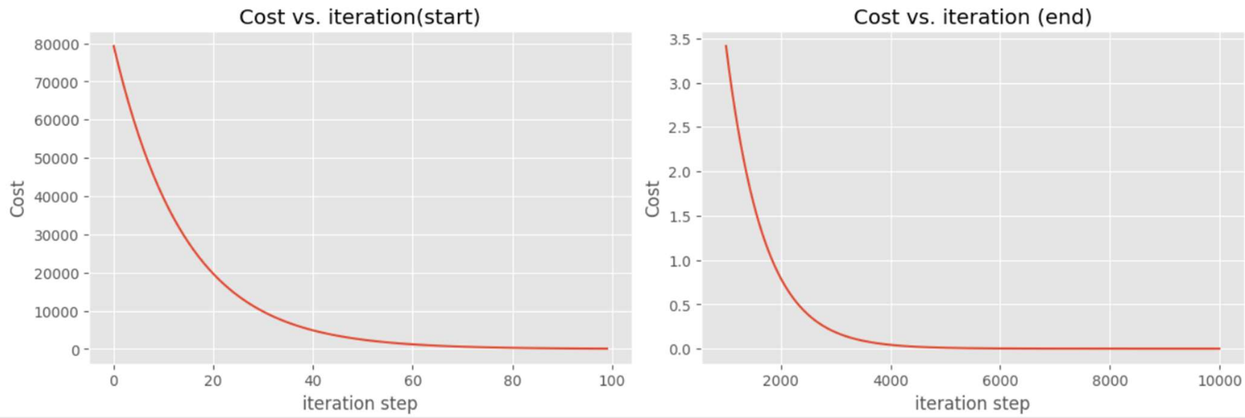
Input:

```
import math, copy
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('ggplot')
# from lab_utils_uni import plt_house_x, plt_contour_wgrad, plt_divergence, plt_gradients
```

```
x_train =np.array([1.0,2.0])
y_train = np.array([300.0,500.0])

def compute_cost(x,y,w,b):
    m = x.shape[0]
    cost = 0
    for i in range(m):
        f_wb = w * x[i] + b
        cost = cost + (f_wb - y[i])**2
    total_cost = 1 / (2 * m) * cost
    return total_cost

def compute_gradient(x,y,w,b):
  m=x.shape[0]
  dj_dw=0
  dj_db=0
  for i in range(m):
    f_wb=w* x[i] + b
    dj_dw_i = (f_wb - y[i]) * x[i]
    dj_db_i = f_wb - y[i]
    dj_dw += dj_dw_i
    dj_db += dj_db_i
  dj_dw = dj_dw / m
  dj_db = dj_db / m
  return dj_dw,dj_db

#plt_gradients(x_train,y_train,compute_cost,compute_gradient)
#plt.show()

def gradient_descent(x,y,w_in,b_in,alpha,num_iters,cost_function,gradient_function):
  J_history=[]
  p_history=[]
  b=b_in
  w=w_in
  for i in range(num_iters):
    dj_dw,dj_db=gradient_function(x,y,w,b)
    b = b - alpha * dj_db
    w = w - alpha * dj_dw

    if i<100000:
      J_history.append(cost_function(x,y,w,b))
      p_history.append([w,b])
    if i%math.ceil(num_iters/10)==0:
      print(f"Iteration {i:4}: Cost {J_history[-1]:0.2e} ",
            f"dj_dw: {dj_dw: 0.3e}, dj_db: {dj_db: 0.3e} ",
            f"w: {w: 0.3e}, b:{b: 0.5e}")
  return w,b,J_history,p_history

w_init =0
b_init =0
iterations = 10000
tmp_alpha = 1.0e-2
w_final, b_final, J_hist, p_hist = gradient_descent(x_train,y_train,w_init,b_init,tmp_alpha,iterations,compute_cost,compute_gradient)
print(f"(w,b) found by gradient descent: ({w_final:8.4f},{b_final:8.4f})")
fig, (ax1, ax2) = plt.subplots(1, 2, constrained_layout=True, figsize=(12, 4))
ax1.plot(J_hist[:100])
ax2.plot(1000 + np.arange(len(J_hist[1000:])), J_hist[1000:])
ax1.set_title("Cost vs. iteration(start)"); ax2.set_title("Cost vs. iteration (end)")
ax1.set_ylabel('Cost') ; ax2.set_ylabel('Cost')
ax1.set_xlabel('iteration step') ; ax2.set_xlabel('iteration step')
plt.show()
```

Output:

```
Iteration 2000: Cost 7.93e-01  dj_dw: -1.789e-01, dj_db:  2.895e-01  w:  1.975e+02, b: 1.03966e+02
Iteration 3000: Cost 1.84e-01  dj_dw: -8.625e-02, dj_db:  1.396e-01  w:  1.988e+02, b: 1.01912e+02
Iteration 4000: Cost 4.28e-02  dj_dw: -4.158e-02, dj_db:  6.727e-02  w:  1.994e+02, b: 1.00922e+02
Iteration 5000: Cost 9.95e-03  dj_dw: -2.004e-02, dj_db:  3.243e-02  w:  1.997e+02, b: 1.00444e+02
Iteration 6000: Cost 2.31e-03  dj_dw: -9.660e-03, dj_db:  1.563e-02  w:  1.999e+02, b: 1.00214e+02
Iteration 7000: Cost 5.37e-04  dj_dw: -4.657e-03, dj_db:  7.535e-03  w:  1.999e+02, b: 1.00103e+02
Iteration 8000: Cost 1.25e-04  dj_dw: -2.245e-03, dj_db:  3.632e-03  w:  2.000e+02, b: 1.00050e+02
Iteration 9000: Cost 2.90e-05  dj_dw: -1.082e-03, dj_db:  1.751e-03  w:  2.000e+02, b: 1.00024e+02
(w,b) found by gradient descent: (199.9929,100.0116)
```



Experiment – 4

Aim:

Implement a SINGLE POINT , MULTI POINT AND UNIFORM crossover operator in python

Theory:

Crossover combines genetic information from two parent solutions to produce offspring in genetic algorithms.

Types of Crossover Operators

1. **Single Point Crossover**:
   - **Process**: Select one random crossover point to split the parents' chromosomes and exchange segments.
   - **Example**:
     - Parents: [1, 0, 1, 1] and [0, 1, 0, 0]
     - Crossover Point: 2
     - Offspring: [1, 0, 0, 0] and [0, 1, 1, 1]
2. **Multi-Point Crossover**:
   - **Process**: Select multiple crossover points and exchange segments between parents at those points.
   - **Example**:
     - Parents: [1, 0, 1, 1] and [0, 1, 0, 0]
     - Crossover Points: 1 and 3
     - Offspring: Varies based on points.
3. **Uniform Crossover**:
   - **Process**: For each gene, randomly choose from one of the parents.
   - **Example**:
     - Parents: [1, 0, 1, 1] and [0, 1, 0, 0]
     - Offspring: Randomly selected genes result in [1, 1, 1, 0] and [0, 0, 0, 1].

Input:

```
import numpy as np

def single_point_crossover(A, B, x):

    A_new = np.append(A[:x], B[x:])

    B_new = np.append(B[:x], A[x:])

    return A_new, B_new

A = np.array([4, 8, 6, 5, 9, 2, 6, 9, 2, 3])

B = np.array([9, 8, 7, 4, 5, 2, 3, 5, 8, 7])

x = 2

A_new, B_new = single_point_crossover(A, B, x)

print("Single Point Crossover")

print("A_new:", A_new)
```

```
print("B_new:", B_new)
#MULTI POINT CROSSOVER
def multi_point_crossover(A, B, y):
    for i in y:
        A, B = single_point_crossover(A, B, i)
    return A, B
y = np.array([2, 5])  # List of crossover points
m_new, n_new = multi_point_crossover(A, B, y)
print("Multie Point Crossover")
print("M_new:", m_new)
print("N_new:", n_new)
#UNIFORM CROSSOVER
def uniform_crossover(A, B, P):
    for i in range(len(P)):
        if P[i] < 0.5:
            temp = A[i]
            A[i] = B[i]
            B[i] = temp
    return A, B
P = np.random.rand(10)  # Probability array of length 10
U_new, V_new = uniform_crossover(A, B, P)
print("Uniform Crossover")
print("U_new:", U_new)
print("V_new:", V_new)
```

Output:

```
Single Point Crossover
A_new: [4 8 7 4 5 2 3 5 8 7]
B_new: [9 8 6 5 9 2 6 9 2 3]
Multie Point Crossover
M_new: [4 8 7 4 5 2 6 9 2 3]
N_new: [9 8 6 5 9 2 3 5 8 7]
Uniform Crossover
U_new: [4 8 6 5 9 2 6 5 2 3]
V_new: [9 8 7 4 5 2 3 9 8 7]
```

Experiment – 5

Aim:

Implement the Knapsack problem.

Theory:
The Knapsack Problem is a classic optimization problem in combinatorial optimization. It involves selecting a subset of items to maximize total value without exceeding a given weight limit.

Problem Statement

- **Given**:
    - A set of items, each with a specific weight and value.
    - A maximum weight capacity (the "knapsack").
- **Objective**:
    - Determine the optimal combination of items that maximizes total value while keeping the total weight within the knapsack's capacity.

Types of Knapsack Problems

1. **0/1 Knapsack Problem**:
    - Each item can either be included (1) or excluded (0) from the knapsack.
    - Example: If you have items with weights [2, 3, 4] and values [3, 4, 5], and a capacity of 5, the optimal selection is the item with weight 2 and value 3 and the item with weight 3 and value 4, yielding a total value of 7.
2. **Fractional Knapsack Problem**:
    - Items can be broken into smaller pieces, allowing fractional inclusion.
    - Example: If an item weighs 10 kg and has a value of $100, but only 5 kg can fit in the knapsack, you can take half of the item for a value of $50.
3. **Bounded Knapsack Problem**:
    - There are limits on how many of each item can be included.
4. **Unbounded Knapsack Problem**:
    - You can take unlimited quantities of each item.

Input:

```
def knapsack(max_capacity, weights, values, n):
    # Initialize the 2D array with zeros
    K = [[0 for x in range(max_capacity + 1)] for x in range(n + 1)]
    # Build the 2D array in bottom-up manner
    for i in range(n + 1):
        for w in range(max_capacity + 1):
            if i == 0 or w == 0:
                K[i][w] = 0
            elif weights[i-1] <= w:
```

```
            K[i][w] = max(values[i-1] + K[i-1][w-weights[i-1]], K[i-1][w])
        else:
            K[i][w] = K[i-1][w]
    return K[n][max_capacity]
# Driver code
max_capacity = 10
values = [50, 40, 80, 10]
weights = [3, 4, 6, 2]
n = len(values)
print("Maximum value that can be obtained:", knapsack(max_capacity, weights, values, n))
```

Output:

```
Maximum value that can be obtained: 130
```