Hands-on Activity 1.1 | Optimization and Knapsack Problem

Objective(s):

This activity aims to demonstrate how to apply greedy and brute force algorithms to solve optimization problems

Intended Learning Outcomes (ILOs):

- Demonstrate how to solve knapsacks problems using greedy algorithm
- Demonstrate how to solve knapsacks problems using brute force algorithm

Resources:

- Jupyter Notebook

Procedures:

1. Create a Food class that defines the following:

- name of the food
- value of the food
- calories of the food

2. Create the following methods inside the Food class:

- A method that returns the value of the food
- A method that returns the cost of the food
- A method that calculates the density of the food (Value / Cost)
- A method that returns a string to display the name, value and calories of the food

```
class Food(object):
    def __init__(self, n, v, w):
        self.name = n
        self.value = v
        self.calories = w
    def getValue(self):
        return self.value
    def getCost(self):
        return self.calories
    def density(self):
        return self.getValue()/self.getCost()
    def __str__(self):
        return self.name + ': <' + str(self.value)+ ', ' + str(self.calories) + '>'
```

3. Create a buildMenu method that builds the name, value and calories of the food

```
def buildMenu(names, values, calories):
    menu = []
    for i in range(len(values)):
        menu.append(Food(names[i], values[i],calories[i]))
    return menu
```

4. Create a method greedy to return total value and cost of added food based on the desired maximum cost

```
def greedy(items, maxCost, keyFunction):
    """Assumes items a list, maxCost >= 0,         keyFunction maps elements of items to numbers"""
    itemsCopy = sorted(items, key = keyFunction,
                       reverse = True)
    result = []
    totalValue, totalCost = 0.0, 0.0
    for i in range(len(itemsCopy)):
        if (totalCost+itemsCopy[i].getCost()) <= maxCost:
            result.append(itemsCopy[i])
            totalCost += itemsCopy[i].getCost()
            totalValue += itemsCopy[i].getValue()
    return (result, totalValue)
```

5. Create a testGreedy method to test the greedy method

```
def testGreedy(items, constraint, keyFunction):
    taken, val = greedy(items, constraint, keyFunction)
    print('Total value of items taken =', val)
    for item in taken:
        print('   ', item)

def testGreedys(foods, maxUnits):
    print('Use greedy by value to allocate', maxUnits,          'calories')
    testGreedy(foods, maxUnits, Food.getValue)
    print('\nUse greedy by cost to allocate', maxUnits,          'calories')
    testGreedy(foods, maxUnits, lambda x: 1/Food.getCost(x))
    print('\nUse greedy by density to allocate', maxUnits,          'calories')
    testGreedy(foods, maxUnits, Food.density)
```

6. Create arrays of food name, values and calories
7. Call the buildMenu to create menu for food
8. Use testGreedys method to pick food according to the desired calories

```
names = ['wine', 'beer', 'pizza', 'burger', 'fries','cola', 'apple', 'donut', 'cake']
values = [89,90,95,100,90,79,50,10]
calories = [123,154,258,354,365,150,95,195]
foods = buildMenu(names, values, calories)
testGreedys(foods, 2000)
```

```
     Use greedy by weight to allocate 2000 calories
     Total value of items taken = 603.0
         fries: <90, 365>
         burger: <100, 354>
         pizza: <95, 258>
         beer: <90, 154>
         wine: <89, 123>
         cola: <79, 150>
         apple: <50, 95>
         donut: <10, 195>
```

Task 1: Change the maxUnits to 100

```
testGreedys(foods, 100)
```

```
     Use greedy by value to allocate 100 calories
     Total value of items taken = 50.0
         apple: <50, 95>

     Use greedy by cost to allocate 100 calories
     Total value of items taken = 50.0
         apple: <50, 95>

     Use greedy by density to allocate 100 calories
     Total value of items taken = 50.0
         apple: <50, 95>
```

Task 2: Modify codes to add additional weight (criterion) to select food items.

```
class Food(object):
    def __init__(self, n, v, w, g):
        self.name = n
        self.value = v
        self.calories = w
        self.weight = g
    def getValue(self):
        return self.value
    def getCost(self):
        return self.calories
    def getWeight(self):
        return self.weight
    def density(self):
        return self.getValue()/self.getCost()
    def __str__(self):
        return self.name + ': <' + str(self.value)+ ', ' + str(self.calories) + '>'

def buildMenu(names, values, calories, weight):
    menu = []
    for i in range(len(values)):
        menu.append(Food(names[i], values[i], calories[i], weight[i]))
    return menu

names = ['wine', 'beer', 'pizza', 'burger', 'fries','cola', 'apple', 'donut', 'cake']
values = [89,90,95,100,90,79,50,10]
calories = [123,154,258,354,365,150,95,195]
weight = [350,400,500,530,330,220,180,380]
foods = buildMenu(names, values, calories, weight)
```

Task 3: Test your modified code to test the greedy algorithm to select food items with your additional weight.

```
def testGreedys(foods, maxUnits):
    print('Use greedy by value to allocate', maxUnits,          'calories')
    testGreedy(foods, maxUnits, Food.getValue)
    print('\nUse greedy by cost to allocate', maxUnits,          'calories')
    testGreedy(foods, maxUnits, lambda x: 1/Food.getCost(x))
    print('\nUse greedy by density to allocate', maxUnits,          'calories')
    testGreedy(foods, maxUnits, Food.density)
    print('\nUse greedy by weight to allocate', maxUnits,          'calories')
    testGreedy(foods, maxUnits, Food.getWeight)

testGreedys(foods, 100)
```

```
     Use greedy by value to allocate 100 calories
     Total value of items taken = 50.0
         apple: <50, 95>

     Use greedy by cost to allocate 100 calories
     Total value of items taken = 50.0
         apple: <50, 95>

     Use greedy by density to allocate 100 calories
     Total value of items taken = 50.0
         apple: <50, 95>

     Use greedy by weight to allocate 100 calories
     Total value of items taken = 50.0
         apple: <50, 95>
```

9. Create method to use Bruteforce algorithm instead of greedy algorithm

```
def maxVal(toConsider, avail):
    """Assumes toConsider a list of items, avail a weight
       Returns a tuple of the total value of a solution to the
         0/1 knapsack problem and the items of that solution"""
    if toConsider == [] or avail == 0:
        result = (0, ())
    elif toConsider[0].getCost() > avail:
        #Explore right branch only
        result = maxVal(toConsider[1:], avail)
    else:
        nextItem = toConsider[0]
        #Explore left branch
        withVal, withToTake = maxVal(toConsider[1:],
                                     avail - nextItem.getCost())
        withVal += nextItem.getValue()
        #Explore right branch
        withoutVal, withoutToTake = maxVal(toConsider[1:], avail)
        #Choose better branch
        if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))
        else:
            result = (withoutVal, withoutToTake)
    return result


def testMaxVal(foods, maxUnits, printItems = True):
    print('Use search tree to allocate', maxUnits,
          'calories')
    val, taken = maxVal(foods, maxUnits)
    print('Total costs of foods taken =', val)
    if printItems:
        for item in taken:
            print('   ', item)


names = ['wine', 'beer', 'pizza', 'burger', 'fries','cola', 'apple', 'donut', 'cake']
values = [89,90,95,100,90,79,50,10]
calories = [123,154,258,354,365,150,95,195]
foods = buildMenu(names, values, calories)
testMaxVal(foods, 2400)
```

```
 Use search tree to allocate 2400 calories
 Total costs of foods taken = 603
     donut: <10, 195>
     apple: <50, 95>
     cola: <79, 150>
     fries: <90, 365>
     burger: <100, 354>
     pizza: <95, 258>
     beer: <90, 154>
     wine: <89, 123>
```

Supplementary Activity:

- Choose a real-world problem that solves knapsacks problem

I choose the calorie deficit for weight loss, which solves knapsacks problem. Calorie deficit requires us to take fewer calories than what our body has to burn so we can lose weight. The computation of calories is needed so we can know if our body will take smaller calories than what our body will burn

- Use the greedy and brute force algorithm to solve knapsacks problem

```
class Diet(object):
    def __init__(self, n, w):
        self.food = n
        self.calories = w
    def getCalo(self):
        return self.calories
    def __str__(self):
        return self.food + ': ' + str(self.calories)

def buildtable(foods, calories):
    table = []
    for i in range(len(calories)):
```

```
def maxVal(toConsider, avail):
    if toConsider == [] or avail == 0:
        result = (0, ())
    elif toConsider[0].getCalo() > avail:
        result = maxVal(toConsider[1:], avail)
    else:
        nextItem = toConsider[0]
        withVal, withToTake = maxVal(toConsider[1:],
                                     avail - nextItem.getCalo())
        withVal += nextItem.getCalo()
        withoutVal, withoutToTake = maxVal(toConsider[1:], avail)
        if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))
        else:
            result = (withoutVal, withoutToTake)
    return result

def testMaxVal(diets, maxUnit, printItems = True):
    print('Use search tree to suggest', maxUnit,
          'calories meal')
    val, taken = maxVal(diets, maxUnit)
    print('Total costs of calories taken =', val)
    if printItems:
        for item in taken:
            print('   ', item)

foods = ['pocari','cobra','sandwich','footlong','fish and chips','sprite','orange','bread','pie']
calories = [123,154,258,354,365,150,95,195]
diets = buildtable(foods, calories)
testMaxVal(diets, 800)
```

```
 Use search tree to suggest 800 calories meal
 Total costs of calories taken = 798
     bread: 195
     orange: 95
     footlong: 354
     cobra: 154
```

Conclusion:

I therefore conclude that after making the codes, the greedy algorithm focuses on higher values, whereas the bruteforce algorithm utilized the maximum values in which it chooses more selections and has smaller remainder