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

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

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

Task 1: Change the maxUnits to 100

```
names = ['wine', 'beer', 'pizza', 'burger', 'fries','cola', 'apple', 'donut']
values = [89,90,95,100,90,79,50,10]
calories = [123,154,258,354,365,150,95,195]
food = buildMenu(names, values, calories)
testGreedys(food, 1500)
```

```
Use greedy by value to allocate 1500 calories
Total value of items taken = 593.0
    burger: <100, 354>
    pizza: <95, 258>
    beer: <90, 154>
    fries: <90, 365>
    wine: <89, 123>
    cola: <79, 150>
    apple: <50, 95>

Use greedy by cost to allocate 1500 calories
Total value of items taken = 513.0
    apple: <50, 95>
    wine: <89, 123>
    cola: <79, 150>
    beer: <90, 154>
    donut: <10, 195>
    pizza: <95, 258>
    burger: <100, 354>

Use greedy by density to allocate 1500 calories
Total value of items taken = 593.0
    wine: <89, 123>
    beer: <90, 154>
    cola: <79, 150>
    apple: <50, 95>
    pizza: <95, 258>
    burger: <100, 354>
    fries: <90, 365>
```

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

```
def testGreedy2(items, constraint, keyFunction):
    taken, val = greedy(items, constraint, keyFunction)
    print('Total value of items taken =', val)
    for item in taken:
        print('   ', item)

def testGreedys2(foods, maxUnits, addcalories):
    newtotalcalories = maxUnits + addcalories
    print('Use greedy by value to allocate', newtotalcalories,          'calories')
    testGreedy2(foods, newtotalcalories, Food.getValue)
    print('\nUse greedy by cost to allocate', newtotalcalories,          'calories')
    testGreedy2(foods, newtotalcalories, lambda x: 1/Food.getCost(x))
    print('\nUse greedy by density to allocate', newtotalcalories,          'calories')
    testGreedy2(foods, newtotalcalories, Food.density)
```

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

```
print("After a few hours, you got even more hungry, and you feel like  you can eat 500 calories more than your previous one")
print("You go back to the store and buy foods\n")
names = ['wine', 'beer', 'pizza', 'burger', 'fries','cola', 'apple', 'donut']
values = [89,90,95,100,90,79,50,10]
calories = [123,154,258,354,365,150,95,195]
food = buildMenu(names, values, calories)
testGreedys2(food, 1500, 200)
```

```
After a few hours, you got even more hungry, and you feel like  you can eat 500 calories more than your previous one
You go back to the store and buy foods

Use greedy by value to allocate 1700 calories
Total value of items taken = 603.0
    burger: <100, 354>
    pizza: <95, 258>
    beer: <90, 154>
    fries: <90, 365>
    wine: <89, 123>
    cola: <79, 150>
    apple: <50, 95>
    donut: <10, 195>
```

```
Use greedy by cost to allocate 1700 calories
Total value of items taken = 603.0
    apple: <50, 95>
    wine: <89, 123>
    cola: <79, 150>
    beer: <90, 154>
    donut: <10, 195>
    pizza: <95, 258>
    burger: <100, 354>
    fries: <90, 365>

Use greedy by density to allocate 1700 calories
Total value of items taken = 603.0
    wine: <89, 123>
    beer: <90, 154>
    cola: <79, 150>
    apple: <50, 95>
    pizza: <95, 258>
    burger: <100, 354>
    fries: <90, 365>
    donut: <10, 195>
```

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

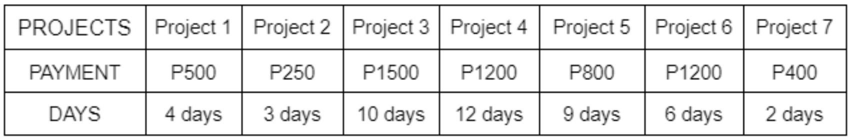
```
def maxVal(toConsider, avail):
    """Assumes toConsider a list of items, avail a weight
       Returns a tuple of the total value of a solution to the
         0/1 knapsack problem and the items of that solution"""
    if toConsider == [] or avail == 0:
        result = (0, ())
    elif toConsider[0].getCost() > 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
- Use the greedy and brute force algorithm to solve knapsacks problem

Below is a list of project available, depending on your vacant days, you want to earn as much as possible without exceeding to your vacant days



PROJECTS	Project 1	Project 2	Project 3	Project 4	Project 5	Project 6	Project 7
PAYMENT	P500	P250	P1500	P1200	P800	P1200	P400
DAYS	4 days	3 days	10 days	12 days	9 days	6 days	2 days

GREEDY METHOD ALGORITHM

```
vacant = int(input("Enter how many days you are available: "))
```

```
Enter how many days you are available: 6
```

```
class Project(object):
    def __init__(self, p, pay, d):
        self.project = p
        self.payment = pay
        self.days = d

    def getValue(self):
        return self.payment

    def getCost(self):
        return self.days
    def __str__(self):
        return self.project + ': P' + str(self.payment)+ ', ' + str(self.days) + ' days'

def projectList(projs, payment, days):
    projlist = []
    for i in range(len(payment)):
        projlist.append(Project(projs[i], payment[i],days[i]))
    return projlist

def greedy(items, maxCost, keyFunction):
    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)

def testGreedy(items, constraint, keyFunction):
    taken, val = greedy(items, constraint, keyFunction)
    print('Total value of payment received =', val, "Pesos")
    for item in taken:
        print('->', item)

def testGreedys(foods, maxUnits):
    print('Maximizing', maxUnits,          'days')
    testGreedy(foods, maxUnits, Project.getValue)


projects = ["Project 1", "Project 2", "Project 3", "Project 4", "Project 5", "Project 6", "Project 7"]
pay = [500, 250, 1500, 1200, 800, 1200, 400]
days = [4, 3, 10, 12, 9, 6, 2]
foods = projectList(projects, pay, days)
testGreedys(foods, vacant)
```

```
Maximizing 6 days
Total value of payment received = 2900.0 Pesos
```

```
-> Project 1: P2500, 4 days
-> Project 7: P400, 2 days
```

BRUTE FORCE ALGORITHM

```
class Project(object):
    def __init__(self, p, pay, d):
        self.project = p
        self.payment = pay
        self.days = d

    def getValue(self):
        return self.payment

    def getCost(self):
        return self.days
    def __str__(self):
        return self.project + ': P' + str(self.payment)+ ', ' + str(self.days) + ' days'

def projectList(projs, payment, days):
    projlist = []
    for i in range(len(payment)):
        projlist.append(Project(projs[i], payment[i],days[i]))
    return projlist

def maxVal(toConsider, avail):
    if toConsider == [] or avail == 0:
        result = (0, ())
    elif toConsider[0].getCost() > avail:
        result = maxVal(toConsider[1:], avail)
    else:
        nextItem = toConsider[0]

        withVal, withToTake = maxVal(toConsider[1:],
                                     avail - nextItem.getCost())
        withVal += nextItem.getValue()
        withoutVal, withoutToTake = maxVal(toConsider[1:], avail)

        if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))
        else:
            result = (withoutVal, withoutToTake)
    return result

def testMaxVal(foods, maxUnits, printItems = True):
    print('Maximizing', maxUnits, 'days')
    val, taken = maxVal(foods, maxUnits)
    print('Total value of payments received =', val)
    if printItems:
        for item in taken:
            print('->', item)

projects = ["Project 1", "Project 2", "Project 3", "Project 4", "Project 5", "Project 6", "Project 7"]
pay = [500, 250, 1500, 1200, 800, 1200, 400]
days = [4, 3, 10, 12, 9, 6, 2]
foods = projectList(projects, pay, days)
testMaxVal(foods, vacant)
```

```
Maximizing 6 days
Total value of payments received = 2900
-> Project 7: P400, 2 days
-> Project 1: P2500, 4 days
```

Conclusion:

In this activity, I have learned that the Brute Force Algorithm could have a different result with the Greedy Method Algorithm. In Brute Forcing, we are looking for all the possible combinations of the item, maximizing the value without exceeding the limit before adding it to the dataset. However in the Greedy Method Algorithm, if the calorie of a certain doesn't exceed the limit, they will add it to the data set, if the next calorie of the certain item does exceed the limit, they will not add it. However, if it doesn't it will add it to the data set