# 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
    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')
```

```
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]
weight = [10,20,30,40,50,60,70,80,90]
foods = buildMenu(names, values, calories, weight)
testGreedys(foods, 2000)
```

```
Use greedy by value to allocate 2000 calories
Total value of items taken = 603.0
    burger: <100, 354, 40>
    pizza: <95, 258, 30>
    beer: <90, 154, 20>
    fries: <90, 365, 50>
    wine: <89, 123, 10>
    cola: <79, 150, 60>
    apple: <50, 95, 70>
    donut: <10, 195, 80>

Use greedy by cost to allocate 2000 calories
Total value of items taken = 603.0
    apple: <50, 95, 70>
    wine: <89, 123, 10>
    cola: <79, 150, 60>
    beer: <90, 154, 20>
    donut: <10, 195, 80>
    pizza: <95, 258, 30>
    burger: <100, 354, 40>
    fries: <90, 365, 50>

Use greedy by density to allocate 2000 calories
Total value of items taken = 603.0
    wine: <89, 123, 10>
    beer: <90, 154, 20>
    cola: <79, 150, 60>
    apple: <50, 95, 70>
    pizza: <95, 258, 30>
    burger: <100, 354, 40>
    fries: <90, 365, 50>
    donut: <10, 195, 80>

Use greedy by weight to allocate 2000 weight
Total value of items taken = 603.0
    donut: <10, 195, 80>
    apple: <50, 95, 70>
```

```
apple: <50, 95, 70>
cola: <79, 150, 60>
fries: <90, 365, 50>
burger: <100, 354, 40>
pizza: <95, 258, 30>
beer: <90, 154, 20>
wine: <89, 123, 10>
```

Task 1: Change the maxUnits to 100

```
#type your code here

maxUnits = 100
testGreedys(foods, 100)
```

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

```
# type your code here

class Food(object):
    def __init__(self, n, v, w, wt):
        self.name = n
        self.value = v
        self.calories = w
        self.weight = wt

    def getWeight(self):
        return self.weight
    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) + ', ' + st

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

def greedy(items, maxCost, keyFunction):
    """Assumes items a list, maxCost >= 0,         keyFunction maps elements of items to
    itemsCopy = sorted(items, key = keyFunction,
                       reverse = True)
    result = []
    totalValue, totalCost = 0.0, 0.0
```

```
    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 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)

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 = [10,20,30,40,50,60,70,80,90]
foods = buildMenu(names, values, calories, weight)
testGreedys(foods, 2000)
```

```
Use greedy by value to allocate 2000 calories
Total value of items taken = 603.0
    burger: <100, 354, 40>
    pizza: <95, 258, 30>
    beer: <90, 154, 20>
    fries: <90, 365, 50>
    wine: <89, 123, 10>
    cola: <79, 150, 60>
    apple: <50, 95, 70>
    donut: <10, 195, 80>

Use greedy by cost to allocate 2000 calories
Total value of items taken = 603.0
    apple: <50, 95, 70>
    wine: <89, 123, 10>
    cola: <79, 150, 60>
    beer: <90, 154, 20>
    donut: <10, 195, 80>
    pizza: <95, 258, 30>
    burger: <100, 354, 40>
    fries: <90, 365, 50>

Use greedy by density to allocate 2000 calories
Total value of items taken = 603.0
```

```
        wine: <89, 123, 10>
        beer: <90, 154, 20>
        cola: <79, 150, 60>
        apple: <50, 95, 70>
        pizza: <95, 258, 30>
        burger: <100, 354, 40>
        fries: <90, 365, 50>
        donut: <10, 195, 80>
```

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

```
# type your code here

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,            'weight')
    testGreedy(foods, maxUnits, Food.getWeight)

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 = [10,20,30,40,50,60,70,80,90]
foods = buildMenu(names, values, calories, weight)
testGreedys(foods, 2000)
```

```
    Use greedy by value to allocate 2000 calories
    Total value of items taken = 603.0
        burger: <100, 354, 40>
        pizza: <95, 258, 30>
        beer: <90, 154, 20>
        fries: <90, 365, 50>
        wine: <89, 123, 10>
        cola: <79, 150, 60>
        apple: <50, 95, 70>
        donut: <10, 195, 80>

    Use greedy by cost to allocate 2000 calories
    Total value of items taken = 603.0
        apple: <50, 95, 70>
        wine: <89, 123, 10>
        cola: <79, 150, 60>
        beer: <90, 154, 20>
        donut: <10, 195, 80>
        pizza: <95, 258, 30>
        burger: <100, 354, 40>
        fries: <90, 365, 50>
```

```
        fries: <90, 365, 50>

    Use greedy by density to allocate 2000 calories
    Total value of items taken = 603.0
        wine: <89, 123, 10>
        beer: <90, 154, 20>
        cola: <79, 150, 60>
        apple: <50, 95, 70>
        pizza: <95, 258, 30>
        burger: <100, 354, 40>
        fries: <90, 365, 50>
        donut: <10, 195, 80>

    Use greedy by weight to allocate 2000 weight
    Total value of items taken = 603.0
        donut: <10, 195, 80>
        apple: <50, 95, 70>
        cola: <79, 150, 60>
        fries: <90, 365, 50>
        burger: <100, 354, 40>
        pizza: <95, 258, 30>
        beer: <90, 154, 20>
        wine: <89, 123, 10>
```

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]
weight = [10,20,30,40,50,60,70,80,90]
foods = buildMenu(names, values, calories, weight)
testMaxVal(foods, 2000)
```

```
Use search tree to allocate 2000 calories
Total costs of foods taken = 603
    donut: <10, 195, 80>
    apple: <50, 95, 70>
    cola: <79, 150, 60>
    fries: <90, 365, 50>
    burger: <100, 354, 40>
    pizza: <95, 258, 30>
    beer: <90, 154, 20>
    wine: <89, 123, 10>
```

## Supplementary Activity:

- Choose a real-world problem that solves knapsacks problem
- Use the greedy and brute force algorithm to solve knapsacks problem

You're given a predetermined financial limitation and a desire to maximize desired retail acquisitions within a confined geographic area, this problem explores the development of a strategic approach for navigating and prioritizing places. The model will consider factors such as individual rating, cost, perceived distance and route that maximizes satisfaction within budgetary constraints. The problem oject seeks to address the common challenge faced by travellers of efficiently allocating resources in a setting brimming with enticing options and limited financial means. Ultimately, the resulting framework aims to empower travellers to make informed decisions and enhance their overall shopping experience.

```
class Gala(object):
    def __init__(self, n, c, r, d):
        self.name = n
        self.cost = c
        self.rating = r
```

```
        self.distance = d

    def getDistance(self):
        return self.distance
    def getCost(self):
        return self.cost
    def getRating(self):
        return self.rating
    def Expenses(self):
        return self.getCost()/self.getDistance()
    def __str__(self):
        return self.name + ': <' + str(self.cost)+ ', ' + str(self.rating) + ', ' + str(s

def buildMenu(names, cost, rating, distance):
    menu = []
    for i in range(len(values)):
        menu.append(Gala(names[i], cost[i],rating[i], distance[i]))
    return menu

def greedy(items, maxCost, keyFunction):
    """Assumes items a list, maxCost >= 0,         keyFunction maps elements of items to
    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].getDistance()
    return (result, totalValue)

def testGreedy(items, constraint, keyFunction):
    taken, val = greedy(items, constraint, keyFunction)
    print('Total Expenses for the transportation of the places selected =', val)
    for item in taken:
        print('   ', item)

def testGreedys(foods, maxUnits):
    print('Use greedy by cost to allocate', maxUnits,                 'budget')
    testGreedy(foods, maxUnits, Gala.getCost)
    print('\nUse greedy by distance to allocate', maxUnits,           'budget')
    testGreedy(foods, maxUnits, lambda x: 1/Gala.getDistance(x))
    print('\nUse greedy by expenses to allocate', maxUnits,           'budget')
    testGreedy(foods, maxUnits, Gala.Expenses)
    print('\nUse greedy by rating to allocate', maxUnits,             'budget')
    testGreedy(foods, maxUnits, Gala.getRating)

names = ['SM Mall of Asia', 'SM Marikina', 'SM North Edsa', 'Sta. Lucia', 'Ayala Feliz','
#cost = [130,30,100,20,20,150,100,25,115]
```

```
#cost = [130,30,100,20,20,150,100,25,115]
cost = [130,30,100,20,20,150,100,25,115]
rating = [9,7,8,5,10,7,4,8]
distance = [23,5,6,8,6,30,2,7,19]
places = buildMenu(names, cost, rating, distance)
testGreedys(places, 500)
```

```
Use greedy by cost to allocate 500 budget
Total Expenses for the transportation of the places selected = 69.0
    Okada: <150, 7, 30>
    SM Mall of Asia: <130, 9, 23>
    SM North Edsa: <100, 8, 6>
    Gateway: <100, 4, 2>
    Sta. Lucia: <20, 5, 8>

Use greedy by distance to allocate 500 budget
Total Expenses for the transportation of the places selected = 57.0
    Gateway: <100, 4, 2>
    SM Marikina: <30, 7, 5>
    SM North Edsa: <100, 8, 6>
    Ayala Feliz: <20, 10, 6>
    Ayala Marikina Heights: <25, 8, 7>
    Sta. Lucia: <20, 5, 8>
    SM Mall of Asia: <130, 9, 23>

Use greedy by expenses to allocate 500 budget
Total Expenses for the transportation of the places selected = 57.0
    Gateway: <100, 4, 2>
    SM North Edsa: <100, 8, 6>
    SM Marikina: <30, 7, 5>
    SM Mall of Asia: <130, 9, 23>
    Ayala Marikina Heights: <25, 8, 7>
    Ayala Feliz: <20, 10, 6>
    Sta. Lucia: <20, 5, 8>

Use greedy by rating to allocate 500 budget
Total Expenses for the transportation of the places selected = 85.0
    Ayala Feliz: <20, 10, 6>
    SM Mall of Asia: <130, 9, 23>
    SM North Edsa: <100, 8, 6>
    Ayala Marikina Heights: <25, 8, 7>
    SM Marikina: <30, 7, 5>
    Okada: <150, 7, 30>
    Sta. Lucia: <20, 5, 8>
```

## Brute Force

```
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.getDistance()
        #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 = ['SM Mall of Asia', 'SM Marikina', 'SM North Edsa', 'Sta. Lucia', 'Ayala Feliz','Ol
#cost = [130,30,100,20,20,150,100,25,115]
cost = [130,30,100,20,20,150,100,25,115]
rating = [9,7,8,5,10,7,4,8]
distance = [23,5,6,8,6,30,2,7,19]
places = buildMenu(names, cost, rating, distance)
testMaxVal(places, 500)
```

```
 Use search tree to allocate 500 calories
 Total costs of foods taken = 85
     Ayala Marikina Heights: <25, 8, 7>
     Okada: <150, 7, 30>
     Ayala Feliz: <20, 10, 6>
     Sta. Lucia: <20, 5, 8>
     SM North Edsa: <100, 8, 6>
     SM Marikina: <30, 7, 5>
     SM Mall of Asia: <130, 9, 23>
```

## Conclusion:

In the knapsack problem, the decision between greedy and brute-force comes down to speed vs.

perfection. Greedy algorithms prioritize efficiency over value, quickly snagging close to ideal answers. Brute force promises the best possible outcome, but it becomes lost in an ocean of options that keeps becoming bigger, rendering it useless for more complex situations. The best course of action ultimately depends on the complexity of the situation and the urgency of locating the "golden prize."