# 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
    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')
```

```
4     print('\nUse greedy by cost to allocate', maxUnits,          'calories')
5     testGreedy(foods, maxUnits, lambda x: 1/Food.getCost(x))
6     print('\nUse greedy by density to allocate', maxUnits,          'calories')
7     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

```
1 names = ['wine', 'beer', 'pizza', 'burger', 'fries','cola', 'apple', 'donut', 'cake']
2 values = [89,90,95,100,90,79,50,10]
3 calories = [123,154,258,354,365,150,95,195]
4 foods = buildMenu(names, values, calories)
5 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

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

```
1 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.

```
1 class Food(object):
2     def __init__(self, n, v, w, g):
3         self.__name = n
4         self.__value = v
5         self.__calories = w
6         self.__weight = g
7     def getValue(self):
8         return self.__value
9     def getCost(self):
10        return self.__calories
11    def getWeight(self):
12      return self.__weight
13    def density(self):
14        return self.getValue()/self.getCost()
15    def __str__(self):
16        return self.__name + ': <' + str(self.__value)+ ', ' + str(self.__calories) + '
17
18 def buildMenu(names, values, calories, weight):
19   menu = []
20   for i in range(len(values)):
21     menu.append(Food(names[i], values[i],calories[i], weight[i]))
22   return menu
23
24 def testGreedys(foods, maxUnits):
25     print('Use greedy by value to allocate', maxUnits,          'calories')
26     testGreedy(foods, maxUnits, Food.getValue)
27     print('\nUse greedy by cost to allocate', maxUnits,          'calories')
28     testGreedy(foods, maxUnits, lambda x: 1/Food.getCost(x))
29     print('\nUse greedy by density to allocate', maxUnits,          'calories')
30     testGreedy(foods, maxUnits, Food.density)
31     print('\nUse greedy by weight to allocate', maxUnits,          'calories')
32     testGreedy(foods, maxUnits, Food.getWeight)
```

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

additional weight.

```
1 names = ['wine', 'beer', 'pizza', 'burger', 'fries','cola', 'apple', 'donut', 'cake']
2 values = [89,90,95,100,90,79,50,10,80]
3 calories = [123,154,258,354,365,150,95,195,361]
4 weights = [76,56,44,71,49,52,55,69,79]
5 foods = buildMenu(names, values, calories, weights)
6 testGreedys(foods, 2000)
```

```
    Use greedy by value to allocate 2000 calories
    Total value of items taken = 673.0
        burger: <100, 354, 71>
        pizza: <95, 258, 44>
        beer: <90, 154, 56>
        fries: <90, 365, 49>
        wine: <89, 123, 76>
        cake: <80, 361, 79>
        cola: <79, 150, 52>
        apple: <50, 95, 55>

    Use greedy by cost to allocate 2000 calories
    Total value of items taken = 593.0
        apple: <50, 95, 55>
        wine: <89, 123, 76>
        cola: <79, 150, 52>
        beer: <90, 154, 56>
        donut: <10, 195, 69>
        pizza: <95, 258, 44>
        burger: <100, 354, 71>
        cake: <80, 361, 79>

    Use greedy by density to allocate 2000 calories
    Total value of items taken = 673.0
        wine: <89, 123, 76>
        beer: <90, 154, 56>
        cola: <79, 150, 52>
        apple: <50, 95, 55>
        pizza: <95, 258, 44>
        burger: <100, 354, 71>
        fries: <90, 365, 49>
        cake: <80, 361, 79>

    Use greedy by weight to allocate 2000 calories
    Total value of items taken = 588.0
        cake: <80, 361, 79>
        wine: <89, 123, 76>
        burger: <100, 354, 71>
        donut: <10, 195, 69>
        beer: <90, 154, 56>
        apple: <50, 95, 55>
        cola: <79, 150, 52>
        fries: <90, 365, 49>
```

## 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]
weights = [76,56,44,71,49,52,55,69,79]
foods = buildMenu(names, values, calories, weights)
testMaxVal(foods, 2000)
```

```
Use search tree to allocate 2000 calories
Total costs of foods taken = 603
    donut: <10, 195, 69>
    apple: <50, 95, 55>
    cola: <79, 150, 52>
    fries: <90, 365, 49>
    burger: <100, 354, 71>
    pizza: <95, 258, 44>
```

```
beer: <90, 154, 56>
wine: <89, 123, 76>
```

## Supplementary Activity:

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

```
"""A logistics company is acting as a sub-contractor for a famous online shopping site.
Lei is a part-time employee at the logistics company, employed as a delivery rider for
parcels at a warehouse. Today, there are 10 different parcels to be delivered, each with
their own individual locations, weights, and values. Before he needs to go to school,
Lei wants to determine the parcels that have the maximum price value in a given set of t

class Parcel(object):
  def __init__(self, name, t, p, w):
    self.__name = name
    self.__time = t
    self.__price = p
    self.__weight = w

  def getTime(self):
      return self.__time

  def getPrice(self):
      return self.__price

  def getWeight(self):
      return self.__weight

  def getPricePerWeight(self):
    return self.getPrice()/self.getWeight()

  def __str__(self):
    return self.__name + ': <' + str(self.__time)+ ', ' + str(self.__price)+ ', ' + str(

def createParcelList(names, times, prices, weights):
  parcelList = []
  for i in range(len(names)):
    parcelList.append(Parcel(names[i], times[i], prices[i], weights[i]))
  return parcelList

def parcelGreedy(parcels, maxTime, sortBy):
  sortedParcels = sorted(parcels, key = sortBy, reverse = True)
  result = []
  totalPrice, totalTime = 0.0, 0.0
  for i in range(len(sortedParcels)):
```

```
    if (totalTime+sortedParcels[i].getTime()) <= maxTime:
      result.append(sortedParcels[i])
      totalTime += sortedParcels[i].getTime()
      totalPrice += sortedParcels[i].getPrice()
  return (result, totalPrice)

def printGreedy(parcels, maxTime, sortBy):
  parcels, priceValue = parcelGreedy(parcels, maxTime, sortBy)
  print('Total price value of parcels chosen =', priceValue)
  for parcel in parcels:
    print('   ', parcel)

def printGreedys(parcels, maxTime):
  print('Use greedy by price to allocate', maxTime, 'minutes')
  printGreedy(parcels, maxTime, Parcel.getPrice)
  print('\nUse greedy by time to allocate', maxTime, 'minutes')
  printGreedy(parcels, maxTime, lambda x: 1/Parcel.getTime(x))
  print('\nUse greedy by price per weight to allocate', maxTime, 'minutes')
  printGreedy(parcels, maxTime, Parcel.getPricePerWeight)
  print('\nUse greedy by weight to allocate', maxTime, 'minutes')
  printGreedy(parcels, maxTime, Parcel.getWeight)

def bruteForce(parcels, maxTime):
  if parcels == [] or maxTime == 0:
    result = (0, ())
  elif parcels[0].getTime() > maxTime:
    result = bruteForce(parcels[1:], maxTime)
  else:
    nextParcel = parcels[0]

    withPrice, withTime = bruteForce(parcels[1:], maxTime - nextParcel.getTime())
    withPrice += nextParcel.getPrice()

    withoutPrice, withoutTime = bruteForce(parcels[1:], maxTime)

    if withPrice > withoutPrice:
      result = (withPrice, withTime + (nextParcel,))
    else:
      result = (withoutPrice, withoutTime)
  return result

def printBruteForce(parcels, maxTime, printItems = True):
    print('\nUse search tree to allocate', maxTime,
          'minutes')
    price, time = bruteForce(parcels, maxTime)
    print('Total costs of foods taken =', price)
    if printItems:
        for item in time:
            print('   ', item)

names = ["Phone Case", "Arduino Kit", "Handheld Fan", "Smart Watch", "Portable Aircon Ur
```

```
91          "Bluetooth Speaker", "Dining Set", "Denim Jacket", "Ceramic Mug", "Laser Printe
92 prices = [100, 1200, 200, 900, 2300, 650, 1950, 525, 225, 3500]
93 time = [45, 30, 60, 60, 75, 120, 90, 30, 105, 20]
94 weights = [10, 500, 400, 20, 2500, 200, 2000, 100, 30, 4000]
95 parcels = createParcelList(names, time, prices, weights)
96 printGreedys(parcels, 240)
97 printBruteForce(parcels, 240)
```

```
    Use greedy by price to allocate 240 minutes
    Total price value of parcels chosen = 8950.0
        Laser Printer: <20, 3500, 4000>
        Portable Aircon Unit: <75, 2300, 2500>
        Dining Set: <90, 1950, 2000>
        Arduino Kit: <30, 1200, 500>

    Use greedy by time to allocate 240 minutes
    Total price value of parcels chosen = 5525.0
        Laser Printer: <20, 3500, 4000>
        Arduino Kit: <30, 1200, 500>
        Denim Jacket: <30, 525, 100>
        Phone Case: <45, 100, 10>
        Handheld Fan: <60, 200, 400>

    Use greedy by price per weight to allocate 240 minutes
    Total price value of parcels chosen = 1750.0
        Smart Watch: <60, 900, 20>
        Phone Case: <45, 100, 10>
        Ceramic Mug: <105, 225, 30>
        Denim Jacket: <30, 525, 100>

    Use greedy by weight to allocate 240 minutes
    Total price value of parcels chosen = 8950.0
        Laser Printer: <20, 3500, 4000>
        Portable Aircon Unit: <75, 2300, 2500>
        Dining Set: <90, 1950, 2000>
        Arduino Kit: <30, 1200, 500>

    Use search tree to allocate 240 minutes
    Total costs of foods taken = 8950
        Laser Printer: <20, 3500, 4000>
        Dining Set: <90, 1950, 2000>
        Portable Aircon Unit: <75, 2300, 2500>
        Arduino Kit: <30, 1200, 500>
```

## Conclusion:

```
#type your conclusion here:
---
The Knapsack problem represents the need for
and real life. Being able to efficiently opti
```

type your conclusion here:

```
and real life. Being able to efficiently opt
Knapsack problem context is important since i
the Greedy Solution and the Brute Forcing sol
are not the most optimal solutions as they or
knowing the entire data set. Its solution is
rather it is the most optimal at the current
```

---

The Knapsack problem represents the need for optimization both in programming and real life. Being able to efficiently optimize how we consider tasks in the Knapsack problem context is important since it shows a great difference. While the Greedy Solution and the Brute Forcing solution are possible, these methods are not the most optimal solutions as they only solve them as they are without knowing the entire data set. Its solution is not the most optimal solution but rather it is the most optimal at the current state of the problem.