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

In [32]:
```
class Food(object):
    def __init__(self, n, v, w, h=5):
        # Make the variables private
        self.name = n
        self.value = v
        self.calories = w
        self.health = h
    def getValue(self):
        return self.value
    def getCost(self):
        return self.calories
    def getHealth(self):
        return self.health
```

```
    def density(self):
        return self.getValue()/self.getCost()
    def healthDensity(self):
        return self.getHealth()/self.getCost()
    def weightedScore(self,value_weight = 0.5,health_weight = 0.5):
        norm_value = self.getValue() / 100
        norm_health = self.getHealth / 10
        return (value_weight * norm_value) + (health_weight * norm_health)
    def __str__(self):
        return f"{self.name}: <{self.value}, {self.calories}, {self.health}>
```

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

In [41]:
```
def buildMenu(names, values, calories, health_scores=None):
    menu = []
    for i in range(len(values)):
        if health_scores:

            menu.append(Food(names[i], values[i],calories[i],health_scores[i
        else:

            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

In [40]:
```
def greedy(items, maxCost, keyFunction):
    """Assumes items a list, maxCost >= 0,         keyFunction maps elements
    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

In [39]:
```
def testGreedy(items, constraint, keyFunction):
    taken, val = greedy(items, constraint, keyFunction)
    print('Total value of items taken =', val)
    for item in taken:
        print('   ', item)
    print()
```

In [38]:
```
def testGreedys(foods, maxUnits):
    print('=' * 60)
```

```
    print(f"Testing with maxUnits = {maxUnits}")
    print('=' * 60)
    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,          'calorie
    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

In [42]:
```
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]
health = [5, 4, 3, 2, 1, 1, 10, 1]
foods = buildMenu(names, values, calories, health)
testGreedys(foods, 100)
```

```
============================================================
Testing with maxUnits = 100
============================================================
Use greedy by value to allocate 100 calories
Total value of items taken = 50.0
    apple: <50, 95, 10>


Use greedy by cost to allocate 100 calories
Total value of items taken = 50.0
    apple: <50, 95, 10>


Use greedy by density to allocate 100 calories
Total value of items taken = 50.0
    apple: <50, 95, 10>
```

Task 1: Change the maxUnits to 100

In [ ]:
```
testGreedys(foods, 100)
```

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

In [ ]:
```
class Food(object):
    def __init__(self, n, v, w, h=5):
        # Make the variables private
        self.name = n
        self.value = v
        self.calories = w
        self.health = h
    def getValue(self):
        return self.value
```

```
    def getCost(self):
        return self.calories
    def getHealth(self):
        return self.health
    def density(self):
        return self.getValue()/self.getCost()
    def healthDensity(self):
        return self.getHealth()/self.getCost()
    def weightedScore(self,value_weight = 0.5,health_weight = 0.5):
        norm_value = self.getValue() / 100
        norm_health = self.getHealth / 10
        return (value_weight * norm_value) + (health_weight * norm_health)
    def __str__(self):
        return f"{self.name}: <{self.value}, {self.calories}, {self.health}>

def buildMenu(names, values, calories, health_scores=None):
    menu = []
    for i in range(len(values)):
        if health_scores:

            menu.append(Food(names[i], values[i],calories[i],health_scores[i
        else:

            menu.append(Food(names[i],values[i],calories[i]))
    return menu

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]
health = [5, 4, 3, 2, 1, 1, 10, 1]
foods = buildMenu(names, values, calories, health)
```

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

In [43]:

```
class Food(object):
    def __init__(self, n, v, w, h=5):
        # Make the variables private
        self.name = n
        self.value = v
        self.calories = w
        self.health = h
    def getValue(self):
        return self.value
    def getCost(self):
        return self.calories
    def getHealth(self):
        return self.health
    def density(self):
        return self.getValue()/self.getCost()
    def healthDensity(self):
        return self.getHealth()/self.getCost()
    def weightedScore(self,value_weight = 0.5,health_weight = 0.5):
        norm_value = self.getValue() / 100
        norm_health = self.getHealth / 10
```

```
        return (value_weight * norm_value) + (health_weight * norm_health)
    def __str__(self):
        return f"{self.name}: <{self.value}, {self.calories}, {self.health}>

def buildMenu(names, values, calories, health_scores=None):
    menu = []
    for i in range(len(values)):
        if health_scores:

            menu.append(Food(names[i], values[i],calories[i],health_scores[i
        else:

            menu.append(Food(names[i],values[i],calories[i]))
    return menu

def greedy(items, maxCost, keyFunction):
    """Assumes items a list, maxCost >= 0,         keyFunction maps elements
    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 items taken =', val)
    for item in taken:
        print('   ', item)
    print()

def testGreedys(foods, maxUnits):
    print('=' * 60)
    print(f"Testing with maxUnits = {maxUnits}")
    print('=' * 60)
    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,          'calorie
    testGreedy(foods, maxUnits, Food.density)

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]
health = [5, 4, 3, 2, 1, 1, 10, 1]
foods = buildMenu(names, values, calories, health)
testGreedys(foods, 100)
```

```
============================================================
Testing with maxUnits = 100
============================================================
Use greedy by value to allocate 100 calories
Total value of items taken = 50.0
    apple: <50, 95, 10>


Use greedy by cost to allocate 100 calories
Total value of items taken = 50.0
    apple: <50, 95, 10>


Use greedy by density to allocate 100 calories
Total value of items taken = 50.0
    apple: <50, 95, 10>
```

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

In [50]:
```
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
```

In [49]:
```
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)
```

In [51]:
```
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]
health = [5, 4, 3, 2, 1, 1, 10, 1]
foods = buildMenu(names, values, calories, health)
testMaxVal(foods, 100)
```

```
Use search tree to allocate 100 calories
Total costs of foods taken = 50
    apple: <50, 95, 10>
```

Supplementary Activity:

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

Real World Problem

The problem is packing an emergency bag during a disaster.

The bag has 15 kg capacity so not all items can be placed inside. Each item has a weight and an importance value. The goal is to get the most useful items without exceeding the backpack's weight limit.

Greedy Algorithm

In [2]:
```
items = [("Water", 5, 10), ("Food", 4, 8), ("FirstAid", 6, 12), ("Flashlight

capacity = 15

items.sort(key=lambda x: x[2] / x[1], reverse=True)

left = capacity
total = 0
picked = []

for i in items:
    if i[1] <= left:
        picked.append(i)
        left = left - i[1]
        total = total + i[2]

print("Greedy Result:")
print("Items picked:")

for p in picked:
    print("-", p[0], "| Weight:", p[1], "| Value", p[2])

print("Total Value:", total)
print("Total Left:", left)
```

```
Greedy Result:
Items picked:
- Powerbank | Weight: 2 | Value 5
- Water | Weight: 5 | Value 10
- Food | Weight: 4 | Value 8
- Flashlight | Weight: 3 | Value 6
Total Value: 29
Total Left: 1
```

Brute Force Algorithm

In [4]:
```python
def bruteForce(items, capacity):
    if items == [] or capacity == 0:
        return 0, []

    name, weight, value = items[0]

    if weight > capacity:
        return bruteForce(items[1:], capacity)

    val_with, items_with = bruteForce(items[1:], capacity - weight)
    val_with += value

    val_without, items_without = bruteForce(items[1:], capacity)

    if val_with > val_without:
        return val_with, items_with + [items[0]]
    else:
        return val_without, items_without
```

In [5]:
```python
value, picked = bruteForce(items, capacity)

print("Brute Force Result")
print("Items inside the backpack:")

for item in picked:
    print(item[0], "weight:", item[1], "value:", item[2])

print("Total value:", value)
```

```
Brute Force Result
Items inside the backpack:
Flashlight weight: 3 value: 6
FirstAid weight: 6 value: 12
Food weight: 4 value: 8
Powerbank weight: 2 value: 5
Total value: 31
```

Conclusion:

So I did the backpack problem for the real world problem and used greedy and brute force to solve it. The first way is the greedy way. It's easier for me. You just pick the best looking item at the time based on its worth and size. It was super fast and my code was simple. But it doesn't always give you the best total pack. That's annoying because you

can get a good answer but not the best. The second way is brute force. This one was way harder for me to get. It has to check every single choice and combination. I used recursion and it was confusing. But the good part is it always finds the perfect best answer. The problem is it's really slow. If you have a lot of items it takes forever. So I learned greedy is fast and easy but not perfect. Brute force is slow and hard to make but always right. I would use greedy for speed and brute force if I really need the right answer.