Session 6: Algorithmic Thinking

1. Steps for Solving a Difficult Programming Problem

I. Describe in English the task in precise language.

II. Decompose the description into well-defined components. For each component, give a step by step recipe of the logic, so that a computer can follow.

III. Translate the description of each component into runnable code, and test each component.

IV. Combine the code together into one coherent program and test the entire program.

1.1 Example: Case 7b (Optimal Pricing)

I. Describe

Given a list of prices, calculate the profit for each price and store the results in a dictionary. Also, find the price that gives the best profit.

II. Decompose

A. Loop through the list of prices.

B. Calculate the profit for each price: this is equal to the price multiplied by the demand. The demand can be found using case 7a).

C. Store the result in a dictionary: the key is the price and the value is the associated profit.

D. Find the best price: define a variable to keep track of the best price found so far and another variable for the best profit. When looping through the prices, update the variables appropriately.

III. Translate

A. Loop through . . .

[1]:
```
priceList=[0,5,10,15,20,25,30,35]
for price in priceList:
    print(price,end=' ')
```

```
0 5 10 15 20 25 30 35
```

B. Calculate the profit . . .

[2]:
```
# Solution of Case 7a
def demand(price,values):
    count=0
    for value in values:
        if value>=price:
            count+=1
    return count
```

[3]:
```
price=20
values=[32,10,15,18,25,40,50,43]
profit=price*demand(price,values)
print(profit)
```

```
100
```

C. Store the result...

```
[4]: result={}
     price=20
     profit=100
     result[price]=profit
     print(result)
```

```
{20: 100}
```

D. Find the best price...

```
[5]: bestPrice=15
     bestProfit=75
     curPrice=20
     curProfit=100
     if curProfit>bestProfit:
         bestProfit=curProfit
         bestPrice=curPrice
     print(bestPrice,bestProfit)
```

```
20 100
```

IV. Combine

First code directly in a notebook cell and print intermediate results for ease of debugging.

```
[6]: priceList=[0,5,10,15,20,25,30,35]
     values=[32,10,15,18,25,40,50,43]
     bestPrice=0
     bestProfit=0
     result={}
     for price in priceList:
         print('Price: ',price,end=' ')
         profit=price*demand(price,values)
         print('Profit', profit,end=' ')
         result[price]=profit
         print('Dictionary',result)
         if profit>bestProfit:
             bestProfit=profit
             bestPrice=price
             print('Updated bestPrice:',bestPrice,'bestProfit:',bestProfit)
     print('Final bestPrice:',bestPrice,'bestProfit',bestProfit)
```

```
Price:  0 Profit 0 Dictionary {0: 0}
Price:  5 Profit 40 Dictionary {0: 0, 5: 40}
Updated bestPrice: 5 bestProfit: 40
Price:  10 Profit 80 Dictionary {0: 0, 5: 40, 10: 80}
Updated bestPrice: 10 bestProfit: 80
Price:  15 Profit 105 Dictionary {0: 0, 5: 40, 10: 80, 15: 105}
Updated bestPrice: 15 bestProfit: 105
```

```
Price:  20 Profit 100 Dictionary {0: 0, 5: 40, 10: 80, 15: 105, 20: 100}
Price:  25 Profit 125 Dictionary {0: 0, 5: 40, 10: 80, 15: 105, 20: 100, 25: 125}
Updated bestPrice: 25 bestProfit: 125
Price:  30 Profit 120 Dictionary {0: 0, 5: 40, 10: 80, 15: 105, 20: 100, 25: 125, 30: 120}
Price:  35 Profit 105 Dictionary {0: 0, 5: 40, 10: 80, 15: 105, 20: 100, 25: 125, 30: 120, 35
Final bestPrice: 25 bestProfit 125
```

Final Solution

[7]:
```python
def optPrice(priceList,values):
    bestProfit=0
    bestPrice=0
    result={}
    for price in priceList:
        profit=demand(price,values)*price
        result[price]=profit
        if profit>bestProfit:
            bestProfit=profit
            bestPrice=price
    return bestPrice, result
```

[8]:
```python
priceList=[0,5,10,15,20,25,30,35]
values=[32,10,15,18,25,40,50,43]
bestPrice,result=optPrice(priceList,values)
print('Best price:',bestPrice)
print('Profit for each price:',result)
```

```
Best price: 25
Profit for each price: {0: 0, 5: 40, 10: 80, 15: 105, 20: 100, 25: 125, 30: 120, 35: 105}
```

2. Practice Problems

Case 8. Optimal Hourly Contract

Write a function `optimalContract` with two input arguments:

- `hours`: the number of hours you would like to work.
- `contracts`: a dictionary mapping the name of a contract to a list of two numbers. The first number is the hourly rate for the first 40 hours. The second number is the bonus for overtime hours, as a proportion of the hourly rate.

The function should return two objects. The first is the best possible pay under the specified number of hours worked, and the second is a list of the names of all contracts resulting in the best pay. (If one contract is better than all the rest, then the list has one element. If two or more contracts are tied for the best pay, then the list contains all of the names of the optimal contracts.)

```python
contracts={'A':[10,.8],'B':[12,0],'C':[12,.1]}
optimalContract(38,contracts)
```

```
(456, ['B', 'C'])
```

```
optimalContract(42,contracts)
```

```
(506.4, ['C'])
```

```
optimalContract(60,contracts)
```

```
(760.0, ['A'])
```

I. Describe in English the task in precise language.
Go through the given contracts, calculate the pay under each contract for the specified number of hours worked, and return the highest pay and the names of the contracts yielding the highest pay.
II. Decompose the description into well-defined components. For each component, give a step by step recipe that a computer can follow.
A. Loop through the given contracts.
B. Calculate the pay under each contract. The logic is the same as the `calculateWage` function from session 3.
C. Keep track of the set of contracts with the best pay: define a variable to record the best pay found so far, and a list recording all of the contracts found so far with the best pay. When processing a new contract, if the contract has worse pay, then do nothing. If it has equal pay as the best so far, then add it to the list. If it has striclty better pay, then update the best pay so far and make this contract the only element in the list.
III. Translate the description of each component into runnable code, and test each component.

```
[9]: # A. Loop through...
     contracts={'A':[10,.5],'B':[12,0],'C':[12,.1]}

     for name in contracts:
         base,bonus=contracts[name]
         print(name,'Base:',base,'Bonus:',bonus)
```

```
A Base: 10 Bonus: 0.5
B Base: 12 Bonus: 0
C Base: 12 Bonus: 0.1
```

```
[10]: # B. Calculate the pay ...
      from session3 import calculateWage
      hours=43
      base=10
      bonus=.5
      pay=calculateWage(hours,base,bonus)
      pay
```

```
445.0
```

```
[11]: # C. Keep track ...
      bestPay=400
      bestContracts=['E','F']
      name='A'
      pay=445
      if pay>bestPay:
```

```
            bestPay=pay
            bestContracts=[name]
        elif pay==bestPay:
            bestContracts.append(name)
        print(bestPay,bestContracts)
```

```
445 ['A']
```

IV. Combine the code together into one coherent program and test the entire program. (First code directly in a notebook cell and print intermediate results for ease of debugging.)

```
[12]: from session3 import calculateWage

      hours=38
      contracts={'A':[10,.5],'B':[12,0],'C':[12,.1]}

      bestPay=0
      bestContracts=[]
      for name in contracts:
          base,bonus=contracts[name]
          print('Processing contract',name,'Base:',base,'Bonus:',bonus,end='  ')
          pay=calculateWage(hours,base,bonus)
          print('Pay:',pay)
          if pay>bestPay:
              bestPay=pay
              bestContracts=[name]
          elif pay==bestPay:
              bestContracts.append(name)
          print('\tbestPay:',bestPay,'bestContracts:',bestContracts)
```

```
Processing contract A Base: 10 Bonus: 0.5 Pay: 380
        bestPay: 380 bestContracts: ['A']
Processing contract B Base: 12 Bonus: 0 Pay: 456
        bestPay: 456 bestContracts: ['B']
Processing contract C Base: 12 Bonus: 0.1 Pay: 456
        bestPay: 456 bestContracts: ['B', 'C']
```

Final Solution:

```
[13]: from session3 import calculateWage

      def optimalContract(hours,contracts):
          bestPay=0
          bestContracts=[]
          for name in contracts:
              base,bonus=contracts[name]
              pay=calculateWage(hours,base,bonus)
              if pay>bestPay:
                  bestPay=pay
                  bestContracts=[name]
              elif pay==bestPay:
```

```
                bestContracts.append(name)
        return bestPay,bestContracts
```

[14]:
```
contracts={'A':[10,.8],'B':[12,0],'C':[12,.1]}
optimalContract(38,contracts)
```

```
(456, ['B', 'C'])
```

[15]:
```
optimalContract(42,contracts)
```

```
(506.4, ['C'])
```

[16]:
```
optimalContract(60,contracts)
```

```
(760.0, ['A'])
```

Case 9: Estimating Demand for Substitutable Products

This problem generalizes case 7a) to selling two products. Write a function named `demand` with two input arguments:

- `priceVector`: a list of length 2 containing two positive numbers, corresponding to the proposed prices for the two products.
- `values`: a list in which each element is a list of length 2, corresponding to the valuation of a customer for the two products.

Assume that each customer would only purchase one of the two products: if the customer's valuations for both products are greater than or equal to the corresponding prices, then the customer will purchase the product in which his/her valuation minus the price is the largest. If there is a tie, then the customer will purchase the first product. For example, if the valuation of a customer is `[9,8]` then

- If `priceVector=[6,4]`, then the customer will purchase the second product because $8 - 4 > 9 - 6$.
- If `priceVector=[5,4]`, then the customer will purchase the first product because $9 - 5 \geq 8 - 4$.
- If `priceVector=[10,8]`, then the constumer will purchase the second product.
- If `priceVector=[10,10]`, then the customer will purchase neither products.

The function should return a list of two numbers, representing the number of customers purchasing each product.

```
values=[[25,15],[18,18],[30,20],[30,30]]
priceVector=[25,20]
demand(priceVector,values)
```

```
[2, 1]
```

I. Describe in English the task in precise language.

Go through the customers. For each customer, figure out which of the products will he/she purchase, if any. Keep track of the total number of customers purchasing each product.

II. Decompose the description into well-defined components. For each component, give a step by step recipe that a computer can follow.

A. Loop through the customers.

B. Figure out which product the customer will purchase. If the customer values both products less than their prices, then he/she will purchase nothing. If the customer only values the first product above its price, then he/she will purchase the first product. If the customer only values the second product above its price, then he/she will purchase the second product. If the customer values both products above their prices, then he/she will purchase the product for which the difference between the valuation and the price is the largest, breaking ties in favor of the first product.

Alternative: Calculate the difference between the customer's valuation and the price for each product. If the difference is negative for both products, then the customer will purchase nothing. Otherwise, the customer will purchase the product with the greater difference, breaking ties in favor of the first product.

C. Keep track of the total number of customers purchasing each product: define a variable for each product tracking the number of customers purchasing that product so far, and incrementing this by one when needed.

III. Translate the description of each component into runnable code, and test each component.

[17]:
```
# A. Loop through...

values=[[25,15],[18,18],[30,20],[30,30]]
for curVal in values:
    v1=curVal[0]
    v2=curVal[1]
    print(v1,v2)
```

```
25 15
18 18
30 20
30 30
```

[18]:
```
# B. Figure out...

curVal=[25,15]
priceVector=[25,10]

if curVal[0]<priceVector[0] and curVal[1]<priceVector[1]:
    print('Purchase nothing')
elif curVal[0]>=priceVector[0] and curVal[1]<priceVector[1]:
    print('Purchase first product')
elif curVal[0]<priceVector[0] and curVal[1]>=priceVector[1]:
    print('Purchase second product')
else:
    diff0=curVal[0]-priceVector[0]
    diff1=curVal[1]-priceVector[1]
    if diff0>=diff1:
        print('Purchase first product')
    else:
        print('Purchase second product')
```

```
Purchase second product
```

```
[19]: # Alternative logic for B.
      curVal=[25,15]
      priceVector=[25,10]
      diff0=curVal[0]-priceVector[0]
      diff1=curVal[1]-priceVector[1]
      if diff0<0 and diff1<0:
          print('Purchase nothing')
      elif diff0>=diff1:
          print('Purchase first product')
      else:
          print('Purchase second product')
```

```
Purchase second product
```

```
[20]: # C. Keep track...
      count=[0,0]
      count[0]+=1
      count[0]+=1
      count[1]+=1
      print(count)
```

```
[2, 1]
```

IV. Combine the code together into one coherent program and test the entire program. (First code directly in a notebook cell and print intermediate results for ease of debugging.)

```
[21]: values=[[25,15],[18,18],[30,20],[30,30]]
      priceVector=[25,20]
      count=[0,0]
      for curVal in values:
          print('Current value vector:',curVal)
          diff0=curVal[0]-priceVector[0]
          diff1=curVal[1]-priceVector[1]
          print('\t Difference of valuation and prices',diff0,diff1)
          if diff0<0 and diff1<0:
              print('\tPurchase nothing')
              continue
          elif diff0>=diff1:
              print('\tPurchase first product')
              count[0]+=1
          else:
              print('\tPurchase second product')
              count[1]+=1
          print('\tCount:',count)
```

```
Current value vector: [25, 15]
         Difference of valuation and prices 0 -5
        Purchase first product
        Count: [1, 0]
Current value vector: [18, 18]
         Difference of valuation and prices -7 -2
```

```
        Purchase nothing
Current value vector: [30, 20]
         Difference of valuation and prices 5 0
        Purchase first product
        Count: [2, 0]
Current value vector: [30, 30]
         Difference of valuation and prices 5 10
        Purchase second product
        Count: [2, 1]
```

Final Solution:

[22]:
```
def demand(priceVector, values):
    count=[0,0]
    for curVal in values:
        diff0=curVal[0]-priceVector[0]
        diff1=curVal[1]-priceVector[1]
        if diff0<0 and diff1<0:
            continue
        elif diff0>=diff1:
            count[0]+=1
        else:
            count[1]+=1
    return count

values=[[25,15],[18,18],[30,20],[30,30]]
priceVector=[25,20]
demand(priceVector,values)
```

```
[2, 1]
```

(Optional Challenge Problem 1) Generalize Case 9 to arbitrarily many products: the input list `priceVector` may have n elements, for an arbitrary positive integer n. Each element of the list `values` is a list of length n, corresponding to a customer's valuation for each of the n products. Each customer maximizes the difference between valuation and the price, breaking ties for products of lower index. The customer purchases nothing if he/she values none of the products above the corresponding price. The function return a list of length n representing the number of customers who purchases each product.

(Optional Challenge Problem 2) Generalize Case 7b) appropriately to find optimal pricing with multiple products: the input list `priceList` is now a list of price vectors, each of which is a list of length n. The input list `values` is as in the problem above. The function `optPrice` should return the optimal price vector found.