Mobility and Smart Cities

TD1 Make Change

Objectives

We have a list of coin and an amount of money to give back.

A valid solution is a list of coins that sum to the amount.
To be a perfect solution we want to give back the minimum number of coins.

Example

We will base our tests with two example:

1. We have the following coins [5,2,1,0.5,0.2,0.1,0.05,0.02,0.01] and we want to give back 12.35. The best solution is [5,5,2,0.2,0.1,0.05].
2. We have the following coins [5,2,1.5] and we want to give back 8. The best solution is [5,1.5,1.5].

Greedy algorithm

Description

The greedy algorithm is to give back the biggest coin possible at each step. It will take the best solution at each step without taking into account what's next.
So the order of the coins can be important.

Implementation

```
import time
```

```
# amount to change
amount = 12.35

# available coins in the cash register (do not take into
account the number of coins available)
available_coins = [5,0.5,0.1,2,1,0.2,0.05,0.02,0.01]

# Greedy : function to make change with unsorted coins
def make_change_unsorted(amount, coins):
    i = 0
    change = []
    while amount > 0 and len(coins) > i:
        # print(str(round(amount//coins[i])) + " Coins of " +
str(coins[i]) + "€")
        for j in range(round(amount//coins[i])):
            change.append(coins[i])
```

```
            amount = round(amount%coins[i],2)
            i = i+1

    return change

# Greedy : function to make change with sorted coins
def make_change_sorted(amount, coins):
    coins.sort(reverse=True)
    i = 0
    change = []

    while amount > 0 and len(coins) > i:
        # print(str(round(amount//coins[i])) + " Coins of " +
str(coins[i]) + "€")
        for j in range(round(amount//coins[i])):
            change.append(coins[i])
        amount = round(amount%coins[i], 2)
        i = i+1
    if amount > 0:
        print(f"Cannot make exact change for {amount:.2f}€")
    return change

# Tests
print("   Exemple 1   ")
print("Available coins:", available_coins, "for", amount,
"€")

start_unsorted = time.perf_counter()
sol = make_change_unsorted(amount, available_coins)
time_unsorted = time.perf_counter() - start_unsorted

print("==== Greedy Unsorted ====")
print(f"Result: {sol}")
print(f'Sum of coins: {sum(sol)}')

print("==== Greedy Sorted ====")

start_sorted = time.perf_counter()
sol = make_change_sorted(amount, available_coins)
time_sorted = time.perf_counter() - start_sorted

print(f"Result: {sol}")
print(f'Sum of coins: {sum(sol)}')

print("")
print("   Exemple 2   ")

amount = 8
available_coins = [2,5,1.5]

print("Available coins:", available_coins, "for", amount,
"€")

print("==== Greedy Unsorted ====")
start_ex2_unsorted = time.perf_counter()
```

```
sol = make_change_unsorted(amount, available_coins)
time_ex2_unsorted = time.perf_counter() - start_ex2_unsorted

print(f"Result: {sol}")
print(f'Sum of coins: {sum(sol)}')

print("==== Greedy Sorted ====")
start_ex2_sorted = time.perf_counter()
sol = make_change_sorted(amount, available_coins)
time_ex2_sorted = time.perf_counter() - start_ex2_sorted

print(f"Result: {sol}")
print(f'Sum of coins: {sum(sol)}')

print("")
print("   Times   ")
# Print times
print(f'Time ex1 unsorted: {time_unsorted:.8f}s')
print(f'Time ex1 sorted: {time_sorted:.8f}s')
print(f'Time ex2 unsorted: {time_ex2_unsorted:.8f}s')
print(f'Time ex2 sorted: {time_ex2_sorted:.8f}s')
```

```
   Exemple 1
Available coins: [5, 0.5, 0.1, 2, 1, 0.2, 0.05, 0.02, 0.01]
for 12.35 €
==== Greedy Unsorted ====
Result: [5, 5, 0.5, 0.5, 0.5, 0.5, 0.1, 0.1, 0.1, 0.05]
Sum of coins: 12.35
==== Greedy Sorted ====
Result: [5, 5, 2, 0.2, 0.1, 0.05]
Sum of coins: 12.35

   Exemple 2
Available coins: [2, 5, 1.5] for 8 €
==== Greedy Unsorted ====
Result: [2, 2, 2, 2]
Sum of coins: 8
==== Greedy Sorted ====
Cannot make exact change for 1.00€
Result: [5, 2]
Sum of coins: 7

   Times
Time ex1 unsorted: 0.00003582s
Time ex1 sorted: 0.00002812s
Time ex2 unsorted: 0.00002229s
Time ex2 sorted: 0.00002753s
```

Analysis

As we can see, the greedy is sometimes not able to reach a valid solution. It can get stuck close to the answer but not find it depending on the ordering of the coins.
In the last example, we can find a valid solution: 8€ to change with 5, 2 and 1.5€ coins. But the greedy algorithm will give back 1 coins of 5€ and 1 coin of 2€. It will not find the solution as it does not explore the next step. Despite this behavior it manages to find the perfect solution in some cases, like in the first example when the coins are ordered from the biggest to the smallest.

Generation of all the possible solutions

Description

To find the perfect solution, we can try to generate all the possible solutions and keep the best one. This way, we are sure to find the perfect solution.

Iterative implementation

```
# Function to calculate all combinations of coins to make a
specific amount
def calculate_change_combinations(coins, amount):
    # Convert to integer by removing the decimal point
    amount_cents = int(amount * 100)
    coin_values_cents = [int(coin * 100) for coin in coins]

    # Initialize a list to store combinations and their
counts
    combinations = []
    list = [(0, [], 0)]  # (current amount in cents, current
combination, current coin index)

    while list:
        current_amount, current_combination, current_coin_i =
list.pop()

        # If the current combination sums up to the target
amount, add it to the list
        if current_amount == amount_cents:
            combinations.append(current_combination)

        # If the amount is less than the target and that we
have coins left to explore
        elif current_amount < amount_cents and current_coin_i
< len(coin_values_cents):
            coin = coin_values_cents[current_coin_i]
            max_count = (amount_cents - current_amount) //
coin  # Maximum count of the current coin

            # Try adding different counts of the current coin
to explore possibilities
            for count in range(max_count + 1):
                new_amount = current_amount + count * coin
                new_combination = current_combination +
```

```
[coins[current_coin_i]] * count
                # Push the new state onto the stack for
further exploration
                list.append((new_amount, new_combination,
current_coin_i + 1))

    return combinations

# Tests

print("  Exemple 1   ")

coin_list = [5, 2, 1, 0.5, 0.2, 0.1, 0.05]
print("Available coins:", coin_list)
change_amount = 12.35
print("Available coins:", available_coins, "for", amount,
"€")

# Call the function to calculate and display the combinations
of coins for the given amount
start_ex1= time.perf_counter()
sol = calculate_change_combinations(coin_list, change_amount)
time_ex1 = start_ex1 - time.perf_counter()

# Find the minimum number of coins required to make the
change
min_coins = min([len(combination) for combination in sol])
# Print the combinations with the minimum number of coins
print(f"Minimum number of coins required: {min_coins}")
print("Combinations:")
for combination in sol:
    if len(combination) == min_coins:
        print(combination)

print("")
print("  Exemple 2   ")

coin_list = [5, 2, 1.5]
change_amount = 8
print("Available coins:", available_coins, "for", amount,
"€")

# Call the function to calculate and display the combinations
of coins for the given amount
start_ex2 = time.perf_counter()
sol = calculate_change_combinations(coin_list, change_amount)
time_ex2 = start_ex1 - time.perf_counter()

# Find the minimum number of coins required to make the
change
min_coins = min([len(combination) for combination in sol])
# Print the combinations with the minimum number of coins
print(f"Minimum number of coins required: {min_coins}")
print("Combinations:")
for combination in sol:
```

```
    if len(combination) == min_coins:
        print(combination)

print("")
print("   Times   ")
# Print times
print(f'Time ex1: {time_ex1:.8f}s')
print(f'Time ex2: {time_ex2:.8f}s')
```

```
  Exemple 1
Available coins: [5, 2, 1, 0.5, 0.2, 0.1, 0.05]
Available coins: [5, 2, 1.5] for 8 €
Minimum number of coins required: 6
Combinations:
[5, 5, 2, 0.2, 0.1, 0.05]

  Exemple 2
Available coins: [5, 2, 1.5] for 8 €
Minimum number of coins required: 3
Combinations:
[5, 1.5, 1.5]

   Times
Time ex1: -6.72253151s
Time ex2: -6.80518212s
```

Recursive Implementation

```
def generates_all_combinations(amount, available_coins,
max_coins):
    # Convert the amount to cents (an integer)
    amount_cents = int(amount * 100)

    # Initialize a list to store all combinations
    all_combinations = []

    coins_list = []
    for i in range(len(available_coins)):  # Iterate through
available coin types
        for j in range(max_coins[i]):  # Repeat each coin
type based on max allowed
            coins_list.append(int(available_coins[i] * 100))
# Convert coin values to cents

    # Generate all combinations
    for r in range(1, amount_cents + 1):
        combinations_r = generate_combinations(coins_list, r,
amount_cents)
        if combinations_r:
            # Convert combinations back to euros and cents
```

```
            combinations_r_euros = [c / 100 for c in
combinations_r[0]]
            return combinations_r_euros

    return all_combinations

# Helper
def generate_combinations(input_list, r, target_amount,
current_combination=[]):
    if r == 0:
        if sum(current_combination) == target_amount:
            return [current_combination]  # Return the valid
combination
        else:
            return []  # Return an empty list for invalid
combinations

    if not input_list:
        return []  # Base case: Return an empty list if
input_list is empty

    first, rest = input_list[0], input_list[1:]

    # Generate combinations with the first element included
    with_first = generate_combinations(rest, r - 1,
target_amount, current_combination + [first])

    # Generate combinations without the first element
    without_first = generate_combinations(rest, r,
target_amount, current_combination)

    return with_first + without_first

# Tests

print("  Exemple 1   ")

available_coins = [5, 0.5, 0.1, 2, 1, 0.2, 0.05, 0.02, 0.01]
max_coins = [2, 3, 4, 2, 3, 4, 2, 3, 4]
amount = 12.35

start_with_cut = time.perf_counter()
combinations = generates_all_combinations(amount,
available_coins, max_coins)
time_with_cut = time.perf_counter() - start_with_cut

print("Available coins:", available_coins, "with max coins:",
max_coins, "for", amount, "€")
# Print the first valid combination found
if combinations:
    print("Valid combination:", combinations)
else:
    print("No valid combination found.")

print("  Exemple 2   ")
```

```
available_coins = [5, 2, 1.5]
max_coins = [10, 10, 10]
amount = 8

print("Available coins:", available_coins, "with max coins:",
max_coins, "for", amount, "€")

start_ex2_with_cut = time.perf_counter()
combinations = generates_all_combinations(amount,
available_coins, max_coins)
time_ex2_with_cut = time.perf_counter() - start_ex2_with_cut

# Print the first valid combination found
if combinations:
    print("Valid combination:", combinations)
else:
    print("No valid combination found.")

# Print times
print("")
print(f'Time ex1: {time_with_cut:.8f}s')
print(f'Time ex2: {time_ex2_with_cut:.8f}s')
```

```
  Exemple 1
Available coins: [5, 0.5, 0.1, 2, 1, 0.2, 0.05, 0.02, 0.01]
with max coins: [2, 3, 4, 2, 3, 4, 2, 3, 4] for 12.35 €
Valid combination: [5.0, 5.0, 0.1, 2.0, 0.2, 0.05]
  Exemple 2
Available coins: [5, 2, 1.5] with max coins: [10, 10, 10] for
8 €
Valid combination: [5.0, 1.5, 1.5]

Time ex1: 0.16850012s
Time ex2: 0.00169333s
```

Analysis

We manage to obtain a the perfect solution in both our implementation. As we have to generate all the combinations this takes a lot of time compare to the greedy algorithm. The time needed to compute the solutions will increase with the number of coins and the amount to change, therefore it is not a good solution for a real world problem as it will not scale.

Dynamic Programming

Description

Dynamic programming is a common approach where we find combinations of smaller values to reach a target value. In this case, we're trying to make change using different types of coins. We consider all possible combinations of coins to reach the desired amount. This approach takes O(nW) steps, where n is the number of coin types.

It uses a matrix to store solutions to sub-problems and returns the minimum number of coins needed to make change. If it's not possible to make change with the given coins, it returns "Infinity." Another matrix can be used to find the specific coins for the optimal solution.

Wikipedia

```
def _get_change_making_matrix(coins, target_amount):
    # Initialize the matrix
    num_coins = len(coins)
    dp_matrix = [[0 for _ in range(target_amount + 1)] for _
in range(num_coins + 1)]

    for i in range(target_amount + 1):
        dp_matrix[0][i] = float('inf')  # By default, there
is no way of making change

    return dp_matrix

def find_min_coins(coins, target_amount):
    # to int :
    coins = [int(coin * 100) for coin in coins]
    target_amount = int(target_amount * 100)

    dp_matrix = _get_change_making_matrix(coins,
target_amount)

    for c in range(1, len(coins) + 1):
        for amount in range(1, target_amount + 1):
            coin_value = coins[c - 1]

            if coin_value == amount:
                dp_matrix[c][amount] = 1
            elif coin_value > amount:
                dp_matrix[c][amount] = dp_matrix[c - 1]
[amount]
            else:
                without_this_coin = dp_matrix[c - 1][amount]
                with_this_coin = 1 + dp_matrix[c][amount -
coin_value]
                if with_this_coin < without_this_coin:
                    dp_matrix[c][amount] = with_this_coin
                else:
                    dp_matrix[c][amount] = without_this_coin

    # Initialize a list to store the coin combinations
    coin_combinations = []
```

```
    # Backtrack to find the coin combinations
    c, r = len(coins), target_amount
    while c > 0 and r > 0:
        if dp_matrix[c][r] == dp_matrix[c - 1][r]:
            c -= 1
        else:
            coin_combinations.append(coins[c - 1])
            r -= coins[c - 1]

    # Divide the coin values by 100 to convert back to euros
    coin_combinations = [coin / 100 for coin in
coin_combinations]
    return coin_combinations


# Tests

print("  Exemple 1   ")

available_coins = [5, 0.5, 0.1, 2, 1, 0.2, 0.05, 0.02, 0.01]
amount = 12.35

print("Available coins:", available_coins, "for", amount,
"€")

start_ex1 = time.perf_counter()
result = find_min_coins(available_coins, amount)
time_ex1 = time.perf_counter() - start_ex1

if result is not None:
    print(f"Minimum number of coins needed: {len(result)}")
    print(f"Coin combinations: {result}")
else:
    print("It's not possible to make change for the given
amount.")


print("  Exemple 2   ")

available_coins = [5, 2, 1.5]
amount = 8

print("Available coins:", available_coins, "for", amount,
"€")

start_ex2 = time.perf_counter()
result = find_min_coins(available_coins, amount)
time_ex2 = time.perf_counter() - start_ex2


if result is not None:
    print(f"Minimum number of coins needed: {len(result)}")
    print(f"Coin combinations: {result}")
else:
    print("It's not possible to make change for the given
```

```
amount.")

# Times

print("")
print(f'Time ex1: {time_ex1:.8f}s')
print(f'Time ex2: {time_ex2:.8f}s')
```

```
  Exemple 1
Available coins: [5, 0.5, 0.1, 2, 1, 0.2, 0.05, 0.02, 0.01]
for 12.35 €
Minimum number of coins needed: 6
Coin combinations: [0.05, 0.2, 2.0, 0.1, 5.0, 5.0]
  Exemple 2
Available coins: [5, 2, 1.5] for 8 €
Minimum number of coins needed: 3
Coin combinations: [1.5, 1.5, 5.0]

Time ex1: 0.00202374s
Time ex2: 0.00043158s
```

Analysis

Using this method, we are able to generate a perfect solution in a reasonable amount of time. It's faster than generating all the possible solutions.