APMA 1930X Homework 2

Milan Capoor

1. Given a list of coin denominations and a target sum, what is the minimum number of coins needed to make the target sum?

Let $\mathcal{D} = \{d_i\}_{i=1}^n$ be the set of coin denominations, and let S be the target sum.

We define a function $V_i(x)$ as the minimum number of coins needed to make the sum x using the first i coin denominations.

Correspondingly, define $A_i(x)$ which returns an ordered list of the number of coins of each denomination required to make the sum.

In this formulation, we are looking for $V_n(S)$ and $A_n(S)$.

Let's start with an easier problem: suppose that we only have one denomination. By assumption, the smallest denomination will always be 1, so clearly,

$$V_1(S) = S$$

and $A_1(S) = \{S\}$ where the number of 1s is S.

Now suppose that we have the optimal number of coins needed to make the sum x using the first $i < n$ denominations (and the associated set). We want to find the optimal number of coins needed to make the sum x using the first $i + 1$ denominations.

If $d_{i+1} > x$, then we cannot use the $i + 1$th denomination to make the sum x. Thus,

$$V_{i+1}(x) = V_i(x), \quad A_{i+1}(x) = A_i(x)$$

Otherwise, we have two options: either we use the $i + 1$th denomination or we don't.

If we use the $i + 1$ denomination, then

$$V_{i+1}(x) = 1 + V_i(x - d_{i+1})$$

where $V_i(x - d_{i+1})$ is the optimal number of coins needed to make the rest of the sum.

If we don't use the $i + 1$ denomination, then

$$V_{i+1}(x) = V_i(x)$$

because we assume that $V_i(x)$ is already optimal.

Since we want the minimum number of coins,

$$V_{i+1}(x) = \min\{1 + V_i(x - d_{i+1}), V_i(x)\}$$

and

$$A_{i+1}(x) = \begin{cases} A_i(x - d_{i+1}) \cup \{d_{i+1}\} & \text{if } V_{i+1}(x) = 1 + V_i(x - d_{i+1}) \\ A_i(x) & \text{if } V_{i+1}(x) = V_i(x) \end{cases}$$

by the same logic.

This is a recursive definition and we already know $V_1(x) = x$ so we can use dynamic programming to find $V_n(S)$ and $A_n(S)$.

We build an optimal value table

	1	2	$\cdots$	S
d_1	$V_1(1)$	$V_1(2)$	$\cdots$	$V_1(S)$
d_2	$V_2(1)$	$\ddots$		
$\vdots$	$\vdots$		$\ddots$	
d_n	$V_n(1)$			$V_n(S)$

(and a optimal elements table defined similarly) according to the rules above and we can find $V_n(S)$ and $A_n(S)$ in $O(nS)$ time.

Using this method,

Q1. Denominations [1, 6, 10] and target sum 13 returns a minimal count of 3 corresponding to [1, 2, 0] (one 1, two 6s, and zero 10s)

Q2. Denominations [1, 1930, 2023] and target sum 10^6 returns a minimal count of 505 corresponding to [10, 15, 480] (ten 1s, fifteen 1930s, and four-hundred eighty 2023s)

2. Given two sequeunces of digits [0, 9], what is the length of the longest common subsequence?

WLOG, let the shorter sequence be $\{a_i\}_1^n$ and the longer sequence be $\{b_i\}_1^m$.

Let $V_i(j)$ be the length of the longest common subsequence of the first i elements of the shorter sequence and the first j elements of the longer sequence.

We want to find $V_n(m)$.

Consider $V_1(x)$ for $x = 1, \ldots, m$. If a_1 is not in the first x elements of the longer sequence, then $V_1(x) = 0$. Otherwise, $V_1(x) = 1$.

Similarly, $V_i(1) = 1$ if $a_i = b_1$ for some i and 0 otherwise.

Now suppose that we have the optimal length of the longest common subsequence of the first i elements of the shorter sequence and the first $j-1$ elements of the longer sequence.

First we check if a_i appears in $b_j, b_{j+1}, \ldots, b_m$.

If $a_i = b_j$, then it could be in the longest common subsequence.

If

$$V_{i-1}(j) + 1 > V_i(j-1)$$

then b_j will contribute to a common sequence that is longer than any other using the first $j-1$ elements of the longer sequence. Otherwise, take the last optimal sequence $V_i(j-1)$, i.e.

$$V_i(j) = \max\{V_{i-1}(j) + 1, V_i(j-1)\}$$

If $a_i \neq b_j$, then we check if

$$V_{i-1}(j) > V_i(j-1)$$

so the best sequence not using a_i is better than the best sequence using a_i. Then

$$V_i(j) = \max\{V_{i-1}(j), V_i(j-1)\}$$

This check corrects for a common element which appears earlier in the longer sequence than any other but would ultimately result in a shorter common subsequence.

We can build a dynamic programming table

		Longer			
		b_1	b_2	$\cdots$	b_m
Shorter	a_1	$V_1(1)$	$V_1(2)$	$\ldots$	$V_1(m)$
	a_2	$V_2(1)$	$\ddots$		
	$\vdots$	$\vdots$		$\ddots$	
	a_n	$V_n(1)$			$V_n(m)$

and fill it in left-to-right, top-to-bottom.

Then $V_{i-1}(j)$ is the value in the same column but the row above, and $V_i(j-1)$ is the value in the same row but the column to the left. We choose the maximum of these two values according to the rules above and thus are always increasing as we move diagonally down and to the right. The longest common subsequence is the value in the bottom right corner of the table.

Using this method,

Q1. Sequences $[1, 3, 9, 6, 7, 2, 1, 8, 5, 4]$ and $[3, 0, 4, 3, 5, 9, 2, 6, 1, 1, 8, 0]$ have max length 5 corresponding to $[3, 9, 2, 1, 8]$

Code implementation (in Python):

```
# ------------------------ PROBLEM 1 ------------------------
# PROBLEM 1 STATEMENT

# Input:
# - denominations: List[Int], List of distinct positive integers
#   whose first element is always 1
# - sum: Int, positive integer

# Output:
# - count: Int, smallest number of elements from a_n that sum to x
# - elements: List[Int], list of number of each denomination used to sum to x

# Given tests:
p1_test1_input = {'denominations': [1, 7, 10], 'sum': 14}
p1_test1_output = {'count': 2, 'elements': [0, 2, 0]}

p1_test2_input = {'denominations': [1, 5, 10, 25], 'sum': 51}
p1_test2_output = {'count': 3, 'elements': [1, 0, 0, 2]}

p1_test3_input = {'denominations': [1, 7, 10], 'sum': 14}
p1_test3_output = {'count': 2, 'elements': [0, 2, 0]}

p1_test4_input = {'denominations': [1, 7, 10], 'sum': 25}
p1_test4_output = {'count': 4, 'elements': [1, 2, 1]}

# Evaluation questions:
p1_question1_input = {'denominations': [1, 6, 10], 'sum': 13}

p1_question2_input = {'denominations': [1, 1930, 2023], 'sum': 10**6}

# PROBLEM 1 SOLUTION

def problem_1(test: dict) -> dict:
    denominations = test['denominations']
    target_sum = test['sum']

    # Initialize (denominations x target_sum) array
    optimal_value_table = [[None for _ in range(target_sum + 1)]
        for _ in range(len(denominations))]

    # Base case: only allowed coins with value 1
    optimal_value_table[0] = [{'count': target,
                               'elements': [target] + [0 for _ in range(1, len(denominations))]
                    for target in range(target_sum + 1)]

    # Inductive step
    for index in range(1, len(denominations)):
        for target in range(target_sum + 1):
            coin_value = denominations[index]
            previous_state = optimal_value_table[index - 1][target]
```

```
            if coin_value > target:
                optimal_value_table[index][target] = previous_state
            else:
                value_with_new_coin = optimal_value_table[index][target - coin_value]

                if value_with_new_coin['count'] + 1 < previous_state['count']:
                    coins_list = value_with_new_coin['elements']
                    updated_coins_list = [coin + 1
                        if i == index
                        else coin
                        for i, coin in enumerate(coins_list)]

                    optimal_value_table[index][target] = {
                        'count': value_with_new_coin['count'] + 1,
                        'elements': updated_coins_list}
                else:
                    optimal_value_table[index][target] = previous_state

    potential_combinations = [row[-1] for row in optimal_value_table]
    induction_results = min(potential_combinations, key=lambda x: x['count'])

    return {'count': induction_results['count'],
            'elements': induction_results['elements']}

def run_p1_tests(problem_1) -> None:
    if problem_1(p1_test1_input) == p1_test1_output:
        print("Test 1 passed")
    else:
        print("Test 1 failed")
        print(f"Expected: {p1_test1_output}, Returned: {problem_1(p1_test1_input)}\n")

    if problem_1(p1_test2_input) == p1_test2_output:
        print("Test 2 passed")
    else:
        print("Test 2 failed")
        print(f"Expected: {p1_test2_output}, Returned: {problem_1(p1_test2_input)}\n")

    if problem_1(p1_test3_input) == p1_test3_output:
        print("Test 3 passed")
    else:
        print("Test 3 failed")
        print(f"Expected: {p1_test3_output}, Returned: {problem_1(p1_test3_input)}\n")

    if problem_1(p1_test4_input) == p1_test4_output:
        print("Test 4 passed")
    else:
        print("Test 4 failed")
        print(f"Expected: {p1_test4_output}, Returned: {problem_1(p1_test4_input)}\n")

def run_p1_questions(problem_1) -> None:
```

```
    print("Evaluation questions:")
    print(f"Q1 Input: {p1_question1_input}, Returned: {problem_1(p1_question1_input)}")
    print(f"Q2 Input: {p1_question2_input}, Returned: {problem_1(p1_question2_input)}")

# PROBLEM 1 OUTPUT

run_p1_tests(problem_1)
run_p1_questions(problem_1)

# ------------------------ PROBLEM 2 ------------------------
# PROBLEM 2 STATEMENT

# Input:
# - subsequence_1: List[Int], List of digits [0-9]
# - subsequence_2: List[Int], List of digits [0-9]

# Output:
# - max_length: Int, length of the longest common subsequence
#    (not necessarily consecutive) between subsequence_1 and subsequence_2
# - longest_subsequence: List[Int], longest common subsequence between subsequence_1
#   and subsequence_2

# Given tests:
p2_test1_input = {'subsequence_1': [1, 3, 9, 6, 7, 2, 1, 8, 5, 4],
                'subsequence_2': [3, 0, 8, 9, 2]}
p2_test1_output = {'max_length': 3, 'longest_subsequence': [3, 9, 2]}

# Evaluation questions:
p2_question1_input = {'subsequence_1': [1, 3, 9, 6, 7, 2, 1, 8, 5, 4],
                    'subsequence_2': [3, 0, 4, 3, 5, 9, 2, 6, 1, 1, 8, 0]}

# PROBLEM 2 SOLUTION

def problem_2(test: dict) -> dict:
    shorter_sequence = test['subsequence_1']
        if len(test['subsequence_1']) < len(test['subsequence_2'])
        else test['subsequence_2']
    longer_sequence = test['subsequence_1']
        if len(test['subsequence_1']) >= len(test['subsequence_2'])
        else test['subsequence_2']

    # Initialize (shorter_sequence x longer_sequence) array
    optimal_value_table = [[0 for _ in range(len(longer_sequence))]
        for _ in range(len(shorter_sequence))]

    longest_list_table = [[[] for _ in range(len(longer_sequence))]
        for _ in range(len(shorter_sequence))]

    #Base case
    for j in range(len(longer_sequence)):
        if shorter_sequence[0] in longer_sequence[:j+1]:
```

```
            optimal_value_table[0][j] = 1
            longest_list_table[0][j] = [longer_sequence[j]]
        else:
            optimal_value_table[0][j] = 0
            longest_list_table[0][j] = []

    for i in range(len(shorter_sequence)):
        if longer_sequence[0] in shorter_sequence[:i+1]:
            optimal_value_table[i][0] = 1
            longest_list_table[i][0] = [longer_sequence[0]]

    # Inductive step
    for i in range(1, len(shorter_sequence)):
        for j in range(1, len(longer_sequence)):
            if shorter_sequence[i] == longer_sequence[j]:
                if optimal_value_table[i - 1][j] + 1 > optimal_value_table[i][j-1]:
                    optimal_value_table[i][j] = optimal_value_table[i-1][j] + 1
                    longest_list_table[i][j] = longest_list_table[i-1][j]
                        + [longer_sequence[j]]
                else:
                    optimal_value_table[i][j] = optimal_value_table[i][j-1]
                    longest_list_table[i][j] = longest_list_table[i][j-1]
            else:
                if optimal_value_table[i - 1][j] > optimal_value_table[i][j-1]:
                    optimal_value_table[i][j] = optimal_value_table[i - 1][j]
                    longest_list_table[i][j] = longest_list_table[i-1][j]
                else:
                    optimal_value_table[i][j] = optimal_value_table[i][j-1]
                    longest_list_table[i][j] = longest_list_table[i][j-1]

    return {'max_length': optimal_value_table[-1][-1],
            'longest_subsequence': longest_list_table[-1][-1]}

# PROBLEM 2 OUTPUT

def run_p2_tests(problem_2) -> None:
    if problem_2(p2_test1_input) == p2_test1_output:
        print("Test 1 passed")
    else:
        print("Test 1 failed")
        print(f"Expected: {p2_test1_output}, Returned: {problem_2(p2_test1_input)}\n")

def run_p2_questions(problem_2) -> None:
    print("Evaluation questions:")
    print(f"Q1 Input: {p2_question1_input}, Returned: {problem_2(p2_question1_input)}")

run_p2_tests(problem_2)
run_p2_questions(problem_2)
```