Problem Set 7 — Module 7

Shenyu Li

October 30th 2024

1 6.1. contiguous subsequence

The maximum subarray sum problem involves finding the contiguous subarray within a one-dimensional array of numbers which has the largest sum.

1.1 MY Process

To solve this problem, utilize a dynamic programming approach. define a variable $s[j]$ as the maximum subarray sum ending at index j. The idea is to keep track of the current maximum sum as iterate through the array.

For each element $a[j]$: - two choices: 1. Extend the previous subarray by including $a[j]$. 2. Start a new subarray with $a[j]$ as the only element.

Thus, the recurrence relation can be defined as:

$$s[j] = \max(s[j-1] + a[j], a[j])$$

where $s[0]$ is initialized to $a[0]$.

The overall maximum subarray sum is given by:

$$\text{max\_sum} = \max_{1 \le j \le n} (s[j])$$

1.2 Algorithm Steps

The algorithm can be implemented in Python as follows:

```
def max_subarray_sum(arr):
    if not arr:
        return 0

    max_sum = float('-inf')
    current_sum = 0

    for number in arr:
        current_sum = max(current_sum + number, number)
        max_sum = max(max_sum, current_sum)
    return max_sum
```

1.3 Dry Run Example

Let's consider the array:

$$\text{arr} = [5, 15, -30, 10, -5, 40, 10]$$

1.4 Dry Run Steps

- Initialization: - max_sum = $-\infty$ - current_sum = 0

- Iteration: 1. For number = 5: - current_sum = $\max(0 + 5, 5) = 5$ - max_sum = $\max(-\infty, 5) = 5$

2. For number = 15: - current_sum = $\max(5 + 15, 15) = 20$ - max_sum = $\max(5, 20) = 20$

3. For number = -30: - current_sum = $\max(20 - 30, -30) = -10$ - max_sum = $\max(20, -10) = 20$

4. For number = 10: - current_sum = $\max(-10 + 10, 10) = 10$ - max_sum = $\max(20, 10) = 20$

5. For number = -5: - current_sum = $\max(10 - 5, -5) = 5$ - max_sum = $\max(20, 5) = 20$

6. For number = 40: - current_sum = $\max(5 + 40, 40) = 45$ - max_sum = $\max(20, 45) = 45$

7. For number = 10: - current_sum = $\max(45 + 10, 10) = 55$ - max_sum = $\max(45, 55) = 55$

- Final Result: The maximum subarray sum is 55, corresponding to the subarray $[10, -5, 40, 10]$.

2 6.5 Pebbling a checkerboard

Given a checkerboard with 4 rows and n columns, where each square contains an integer, place some or all of the $2n$ pebbles such that each pebble can be placed on exactly one square. The objective is to maximize the sum of the integers in the squares that are covered by pebbles, under the constraint that no two pebbles can be placed on horizontally or vertically adjacent squares.

2.1 Thought Process

To tackle this problem, break it down into two main parts:

2.2 Part (a)

1. Define State: - Let n_i be the number of patterns of pebbles that can be placed in the first i rows of a column. - Initialize: $n_0 = 1$ and $n_1 = 2$.

2. Recurrence Relation: - For any position i, there are two scenarios for placing pebbles: 1. Not placing a pebble at the i-th position, which allows for n_{i-1} patterns. 2. Placing a pebble at the i-th position, which requires that the

previous pebble is placed at $i-2$, giving n_{i-2} patterns. - Therefore, we derive the relation:

$$n_i = n_{i-1} + n_{i-2}$$

- This is a Fibonacci sequence, and we can compute the number of patterns for any n.

2.3 Part (b)

1. Define State: - Let $s(i,t)$ represent the maximum value obtainable by placing pebbles in the first i columns with the last column type being t.

2. Recurrence Relation: - The value can be expressed as:

$$s(i,t) = \max_{c(t,s)=true} (s(i-1) + v(i,t))$$

- Where $v(i,t)$ is the value obtained from placing pebbles, and $c(i,j)$ determines if types i and j are compatible. - Finally, we need to compute:

$$\max_{0 \le t < 8} (s(n,t))$$

2.4 Algorithm Steps for Part (a)

Here is a Python implementation of the algorithm for part (a):

```
def count_patterns(n):
    if n == 0:
        return 1
    elif n == 1:
        return 2

    patterns = [0] * (n + 1)
    patterns[0] = 1
    patterns[1] = 2

    for i in range(2, n + 1):
        patterns[i] = patterns[i - 1] + patterns[i - 2]

    return patterns[n]
```

2.5 Dry Run for Part (a)

Let's consider a dry run with $n = 4$:

- Initialization: - $n_0 = 1$ - $n_1 = 2$

- Iteration: 1. For $n = 2$: - $n_2 = n_1 + n_0 = 2 + 1 = 3$ 2. For $n = 3$: - $n_3 = n_2 + n_1 = 3 + 2 = 5$ 3. For $n = 4$: - $n_4 = n_3 + n_2 = 5 + 3 = 8$

- Result: The number of patterns for $n = 4$ is 8.

2.6 Algorithm Steps for Part (b)

Here is a Python implementation of the algorithm for part (b):

```
def max_pebble_value(n, values):
    dp = [[0] * 8 for _ in range(n + 1)]

    for i in range(1, n + 1):
        for t in range(8):
            max_value = 0
            for j in range(8):
                if compatible(t, j):  # Assume compatible() is defined
                    max_value = max(max_value, dp[i - 1][j])
            dp[i][t] = max_value + values[i - 1][t]

    return max(dp[n])
```

2.7 Dry Run for Part (b)

Let's consider a dry run with $n = 4$ and the following values:
- Given values:

$$\text{values} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \end{bmatrix}$$

- Initialization: dp array is set to zero.
- Iteration: - For $i = 1$: - For $t = 0$: - Update $dp[1][0] = v(1, 0)$ - Continue for all types t from 0 to 7. - Repeat for $i = 2, 3, \ldots, n$.
- The final result is obtained from the last row of dp: - Return $\max(dp[n])$.

3 6.8 longest common substring

Given two strings $x = x_1 x_2 \ldots x_n$ and $y = y_1 y_2 \ldots y_m$, find the length of their longest common substring, that is, the largest k for which there are indices i and j such that

$$x_i x_{i+1} \ldots x_{i+k-1} = y_j y_{j+1} \ldots y_{j+k-1}.$$

aim to do this in time $O(mn)$.

3.1 my Process

To solve the problem of finding the longest common substring, use dynamic programming. define $L(i, j)$ as the length of the longest common substring that ends at $x[i]$ and $y[j]$.

The recursive relation is defined as follows:

$$L(i,j) = \begin{cases} 0 & \text{if } x[i] \neq y[j] \\ L(i-1,j-1)+1 & \text{if } x[i] = y[j] \end{cases}$$

This means that if the characters $x[i]$ and $y[j]$ match, extend the length of the common substring found so far. If they do not match, the length is reset to zero.

To find the length of the longest common substring, maintain a variable to track the maximum length encountered during the computation.

3.2 Algorithm Steps

Python implementation of the algorithm:

```
def longest_common_substring(x, y):
    n = len(x)
    m = len(y)

    L = [[0] * (m + 1) for _ in range(n + 1)]

    max_length = 0  substring

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if x[i - 1] == y[j - 1]:
                L[i][j] = L[i - 1][j - 1] + 1
                max_length = max(max_length, L[i][j])
            else:
                L[i][j] = 0

    return max_length
```

3.3 Dry Run Example

consider an example with strings $x =$ "$abcde$" and $y =$ "$abfce$".

3.4 Initialization

- Lengths: - $n = 5$ (length of x) - $m = 5$ (length of y) - Initialize a 2D array L with dimensions $(n+1) \times (m+1)$:

$$L = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

- max_length = 0

3.5 Iteration

- Outer Loop: For $i = 1$ to 5: - Inner Loop: For $j = 1$ to 5: 1. $i = 1, j = 1$: $x[0] = a, y[0] = a$ (Match) - $L[1][1] = L[0][0] + 1 = 1$ - max_length = 1

2. $i = 1, j = 2$: $x[0] = a, y[1] = b$ (No Match) - $L[1][2] = 0$

3. $i = 1, j = 3$: $x[0] = a, y[2] = f$ (No Match) - $L[1][3] = 0$

4. $i = 1, j = 4$: $x[0] = a, y[3] = c$ (No Match) - $L[1][4] = 0$

5. $i = 1, j = 5$: $x[0] = a, y[4] = e$ (No Match) - $L[1][5] = 0$

6. Continue iterating through the remaining indices. The final values for the table will be:

$$L = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

- Final Result: The longest common substring length is max_length = 2.

3.6 6.17/8 coin question

Given denominations $x_1, x_2, \ldots, x_n$, want to make change for a value v, but you are allowed to use each denomination at most once. Determine if you can make change for v using each denomination at most once.

3.7 my Process

To solve this variation of the change-making problem, use dynamic programming. define a boolean array $b(i, j)$ where: - $b(i, j)$ indicates whether it is possible to make change for j using the first i denominations.

State Transition 1. If choose to use the i-th denomination: - If j is greater than or equal to x_i:

$$b(i, j) = b(i - 1, j - x_i)$$

2. If do not use the i-th denomination:

$$b(i, j) = b(i - 1, j)$$

Combining these gives:

$$b(i, j) = b(i - 1, j - x_i) \vee b(i - 1, j)$$

Where $\vee$ represents a logical OR.

Complexity The algorithm operates with a time complexity of $O(nv)$, where v is the target value and n is the number of denominations.

4 Algorithm Steps

Python implementation of the algorithm:

```
def can_make_change(denominations, v):
    n = len(denominations)

    b = [[False] * (v + 1) for _ in range(n + 1)]

    for i in range(n + 1):
        b[i][0] = True

    for i in range(1, n + 1):
        for j in range(1, v + 1):
            # If we can make change without using the current denomination
            b[i][j] = b[i-1][j]
            # If we can make change by using the current denomination
            if j >= denominations[i-1]:
                b[i][j] = b[i][j] or b[i-1][j - denominations[i-1]]

    return b[n][v]
```

5 Dry Run Example

Let's consider an example with denominations $1, 5, 10, 20$ and a target value $v = 31$.

5.1 Initialization

- Denominations: $[1, 5, 10, 20]$ - Target value: $v = 31$ - Initialize a 2D array b of size $(n + 1) \times (v + 1)$:

$$b = \begin{bmatrix} \text{True} & \text{False} & \text{False} & \dots & \text{False} \\ \text{True} & \text{False} & \text{False} & \dots & \text{False} \\ \text{True} & \text{False} & \text{False} & \dots & \text{False} \\ \text{True} & \text{False} & \text{False} & \dots & \text{False} \\ \text{True} & \text{False} & \text{False} & \dots & \text{False} \end{bmatrix}$$

5.2 Filling the DP Table

- Outer Loop: For $i = 1$ to 4 (number of denominations): - Inner Loop: For $j = 1$ to 31: - When $i = 1$ (denomination = 1)**: - Fill the first row: - $b[1][1] = b[0][0] \rightarrow$ True - $b[1][2] = b[0][1] \rightarrow$ False - Continue filling, resulting in:

$$b = \begin{bmatrix} \text{True} & \text{True} & \text{True} & \dots & \text{True} \\ \text{True} & \text{True} & \text{False} & \dots & \text{False} \\ \dots & & & & \end{bmatrix}$$

- Continue for $i = 2, 3, 4$**: - Fill in values based on the previous results and the conditions outlined.

- After processing all denominations, check $b[n][v]$.

Final Result - After filling the table, $b[4][31]$ will indicate whether we can make change for 31 using the given denominations.