HW5

Keerthana Golla

September 2023

1 Problem-1

This problem can be solved similar way to the 0/1 knapsack problem(which is solved using a dynamic programming).

Algorithm 1 Function `canDivideEqually`

```
function canDivideEqually(P)
                                          ▷ Calculate the total sum of the array elements
    totalSum ← sum(P)
                                                  ▷ Check if the total sum is even
    if totalSum is not even then
        return False
    end if
                                                  ▷ Get the length of the array
    n ← length(P)
                                          ▷ Calculate the target sum for each subset
    targetSum ← totalSum/2
                                                  ▷ Initialize the W table
    W 2D array of size (n + 1)(targetSum + 1)
                                                  ▷ Base cases
    for i from 0 to n do
        W[i][0] ← True
    end for
                                                  ▷ Dynamic Programming
    for i from 1 to n do
        for j from 1 to targetSum do
            W[i][j] ← W[i − 1][j]
                                  ▷ Check if the current element can contribute to the subset sum
            if j ≥ P[i − 1] then
                W[i][j] ← W[i][j] ∨ W[i − 1][j − P[i − 1]]
            end if
        end for
    end for
                                                  ▷ Return the result
    return W[n][targetSum]
end function
```

Initialization:

- Initialize a 2D array W of size $(n+1) \times (targetSum+1)$, where n is the length of the array P.
- Set $W[i][0]$ to `True` for all i (from 0 to n). This is the base case representing an empty subset with a sum of 0.

Dynamic Programming Iteration: For each element in the array P (indexed by i) and for each possible sum (j from 1 to $targetSum$), the algorithm updates the W array.

- **Including the Current Element ($P[i-1]$):**
  - If the current sum j is greater than or equal to the current element $P[i-1]$, the algorithm checks whether including the current element in the subset contributes to achieving the desired sum.
  - If yes, the algorithm sets $W[i][j]$ to `True`. This means that the sum j can be achieved by including the current element.
- **Excluding the Current Element:**
  - Regardless of whether the current element is included, the algorithm sets $W[i][j]$ to the value of $W[i-1][j]$. This represents the case where the current element is not included in the subset.

Building up Solutions: As the iteration progresses, the W array is filled. For each pair (i, j), the value of $W[i][j]$ indicates whether the sum j can be achieved using a subset of the first i elements of the array P.

Final Result: The final result is obtained from $W[n][targetSum]$. If $W[n][targetSum]$ is `True`, it means that there exists a subset of the array P whose sum is equal to the target sum.

Running Time: Clearly due to two for loops in the algorithm the time complexity will be O(n * $targetSum$).

Python Code in LaTeX

```
def can_divide_equally(P):
    totalSum = sum(P)

    % Check if the total sum is even
    if totalSum \% 2 != 0:
        return False

    n = len(P)
    targetSum = totalSum // 2

    % Initialize the W table
    W = [[False] * (targetSum + 1) for _ in range(n + 1)]

    % Base cases
    for i in range(n + 1):
        W[i][0] = True

    % Dynamic Programming
    for i in range(1, n + 1):
        for j in range(1, targetSum + 1):
            W[i][j] = W[i-1][j]
            if j >= P[i-1]:
                W[i][j] |= W[i-1][j - P[i-1]]


    % Returning the result
    return W[n][targetSum]

% Example for testing:
P = [1,3,4]
result = can_divide_equally(P)
print(result)
```

2 Problem-2

This problem can be solved using Unbounded knapsack problem(which is solved using a dynamic programming).

Algorithm 2 Count Ways to Spend

```
1: function CountWaysToSpend(A, C)
2:     W ← array of size (A + 1) initialized with zeros
3:     W[0] ← 1                                              ▷ Base case
4:     for i ← 1 to A + 1 do
5:         for j ← 0 to length(C) − 1 do
6:             if i − C[j] ≥ 0 then
7:                 W[i] ← W[i] + W[i − C[j]]                 ▷ Update ways to make change for amount i
8:             end if
9:         end for
10:    end for
11:    return W[A]                                           ▷ Final result
12: end function
```

Function Definition: The algorithm defines a function `CountWaysToSpend` that takes two parameters: `A` (the target amount to make change for) and `C` (a set of coin denominations).

Initialization: An array `W` of size $(A + 1)$ is initialized with zeros. This array will be used to store the number of ways to make change for each amount from 0 to A. The base case is set: `W[0] = 1`, indicating that there is exactly one way to make change for an amount of 0 (using no coins).

Dynamic Programming Loop: The algorithm then enters a nested loop:

- The outer loop iterates over each amount from 1 to A.
- The inner loop iterates over each coin denomination in the set `C`.
- For each combination of amount i and coin denomination $C[j]$, the algorithm checks if the current coin denomination can be used to make change for the current amount (`i - C[j] >= 0`).

Update ways to wake change: If the condition is true, it means the current coin denomination can be used to make change for the current amount. The number of ways to make change for amount i is updated by adding the number of ways for (`i - C[j]`). This step follows the principle of dynamic programming by building up solutions for smaller sub problems.

Result: The final result is the value stored in `W[A]`, which represents the number of ways to make change for the given amount A using the provided coin denominations `C`.

Time Complexity: Similar to problem 1, Clearly due to two *for* loops in the algorithm the time complexity will be O(n * A).

Python Code in LaTeX

```
def count_ways_to_spend(A, C):
    W = [0] * (A + 1)
    W[0] = 1  # Base case

    for i in range(1, A + 1):
        for j in range(len(C)):
            if i - C[j] >= 0:
                print(W[i], i, C[j])
                W[i] += W[i - C[j]]
                print(W[i])

    return W[A]

# Example
A = 6
C = [3, 5]
result = count_ways_to_spend(A, C)
print(f"Number of ways to spend exactly {A} dollars: {result}")
```

3 Problem-3

Lets use indicator random variables and linearity of expectation to analyze the expected number of local spikes in the given array A.

Let X_i be the indicator random variable for the event that the i-th element is a local spike. It takes the value 1 if $A[i]$ is a local spike and 0 otherwise.

$$X_i = \begin{cases} 1 & \text{if } A[i] > \max(A[1], A[2], \ldots, A[i-1]) \\ 0 & \text{otherwise} \end{cases}$$

Now, the total number of local spikes, X, is the sum of these indicator variables:

$$X = X_1 + X_2 + \ldots + X_n$$

The expected value of X is given by the linearity of expectation:

$$E[X] = E[X_1] + E[X_2] + \ldots + E[X_n]$$

Now, let's calculate the expected value of each X_i:

$$E[X_i] = P(\text{the } i\text{-th element is a local spike})$$

For the i-th element to be a local spike, it must be greater than all of its preceding elements. Since the array consists of distinct integers, the probability that $A[i]$ is greater than all preceding elements is $1/i$.

$$E[X_i] = \frac{1}{i}$$

Now, we can express the expected number of local spikes:

$$E[X] = E[X_1] + E[X_2] + \ldots + E[X_n] = \frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{n}$$

This is the harmonic series, and its growth is logarithmic. Therefore, the expected number of local spikes is $O(\log n)$.

Steps for detailed explanation terms-wise:

Indicator Random Variables: X_i is defined as the indicator random variable for the event that the i-th element is a local spike. It takes the value 1 if $A[i]$ is greater than all preceding elements; otherwise, it takes the value 0.

Total Number of Local Spikes: X is defined as the total number of local spikes, expressed as the sum of the indicator variables: $X = X_1 + X_2 + \ldots + X_n$.

Linearity of Expectation: The expected value of the total number of local spikes ($E[X]$) is computed using the linearity of expectation: $E[X] = E[X_1] + E[X_2] + \ldots + E[X_n]$.

Probability Calculation: The probability $E[X_i]$ that the i-th element is a local spike is calculated as $1/i$. This is based on the assumption that the array consists of distinct integers.

Expression of Expected Value: The expected value $E[X]$ is expressed as the sum of the harmonic series: $E[X] = \frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{n}$.

Harmonic Series: It is stated that the growth of the harmonic series is logarithmic: $H_n = \frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{n} = \log(n) + O(1)$.

Conclusion: Therefore, $E[X] = O(\log n)$, indicating that the expected number of local spikes is logarithmic in the size of the array.

The correctness of the proof relies on the proper definition of indicator variables, the use of probability calculations based on the distinctness of integers, and the application of the harmonic series properties. If these assumptions hold, then the proof is valid, and the expected number of local spikes is indeed $O(\log n)$.