Name: Grishma Maheshbhai Thumar

Student ID: 922950012

Course: CSC 510, Spring 2025

Project: Homework 3

1. The idea is to “fill” the 2×n grid incrementally (similar to the domino tiling example from the slides). The strategy is to fill the board cell by cell. We choose the first (i.e. leftmost/topmost) empty cell and then try to place each piece in every allowed orientation such that the piece covers that cell and all cells of the piece lie within the grid and on empty squares. After placing a piece, we recursively attempt to tile the remaining board. When the board is completely filled, we have found one valid tiling.

```
TETRIS(n):
   // Create a 2 x n board
   board := 2×n grid, initially all empty
   return PlacePieces(board)

PlacePieces(board):
   if every cell in board is filled:
      return 1

   (r, c) := coordinates of the first empty cell (row–major order)
   count := 0

   for each piece in {J, L, O}:
      for each allowed rotation R of piece
         positions := list of coordinates [(r + dr, c + dc)] for each offset in the rotation
           if all positions are within board bounds and are empty then
              // Place the piece: mark all these positions as filled
              for each (i, j) in positions
                  board[i][j] := TRUE
            // Recurse on the updated board
              count := count + PlacePieces(board)
            // Backtrack: remove the piece
              for each (i, j) in positions
                  board[i][j] := FALSE

  return count
```

2. The text splitting problem asks whether a string S can be segmented into a sequence of valid words. We modify this idea to count the number of ways to split S. Starting from an index *i* in the string, we try every possible substring S[i…j] (for j from i up to the end) and check if it is a valid word (using a dictionary lookup). If it is, we recursively count the number of valid segmentations for the remaining substring

S[j+1…end]. When we pass the end of the string, we count that as one complete valid segmentation.

```
COUNTSPLIT(S[1, … , n], i):
  if i = n + 1:
    return 1              // Reached the end
  count := 0
  for j = i , …, n
    if S[i...j] is a word:
      count := count + COUNTSPLIT(S, j + 1)
  return count
```

3. At Blossom Bake Shop, cupcakes come in sets of 6, 9, or 20. The problem is to decide whether it is possible to buy exactly *n* cupcakes by choosing some combination of these sets. This is similar to a subset-sum (or coin change decision) problem where you subtract the set sizes from *n*. The recursive (backtracking) solution subtracts each available set size from *n* and recurses. The base cases are: if $n = 0$ then you have an exact match (return TRUE), and if $n < 0$ then the current combination overshoots (return FALSE).

```
CUPCAKE(n):
  if n == 0:
    return TRUE           // Exactly matched the requirement
  if n < 0:
    return FALSE
  // Try each available set
  if CUPCAKE(n - 6)
     return TRUE
  if CUPCAKE(n - 9)
     return TRUE
  if CUPCAKE(n - 20)
     return TRUE
  return FALSE
```

4. Inspired by the wood-cutting and domino tiling examples from the slides, we start with a rectangular piece of wood of size $m \times n$. The direct sale price is given by *P[m][n]*. However, you can try making axis-aligned cuts (either vertical or horizontal) that split the rectangle into two smaller rectangles. For each rectangle of size m × n, the optimal revenue is the maximum of:

   Selling the whole rectangle for P[m][n].

   For every vertical cut (i from 1 to n − 1), the sum of the best revenues for rectangles m × i and m × (n − i).

   For every horizontal cut (j from 1 to m − 1), the sum of the best revenues for rectangles j × n and (m − j) × n.

```
RECT(P[1, … , M][1, … , N], m, n):
    best := P[m][n]      // revenue from selling the whole piece
    // Try all vertical cuts.
    for i = 1 , … , n - 1:
        revenue := RECT(P, m, i) + RECT(P, m, n - i)
         best := max(best, revenue)
    // Try all horizontal cuts.
    for j = 1 , …. , m - 1:
        revenue := RECT(P, j, n) + RECT(P, m - j, n)
         best := max(best, revenue)

    return best
```