Name: Grishma Maheshbhai Thumar

Student ID: 922950012

Course: CSC 510, Spring 2025

Project: Homework 4

Highlighted the recurrence or backtracking algorithm from the previous homework in yellow.

1. Tiling a 2 * n Grid with J, L, and O Pieces

Algorithm from HW 3

```
TETRIS(n):
  // Create a 2×n board (all cells empty)
  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 cells covered by placing piece R at (r,c)
      if positions are all in bounds and empty:
        mark those positions as filled
        //recursion
        count := count + PlacePieces(board)
        // unmark those positions (backtrack)
        for each (i, j) in positions
              board[i][j] := FALSE

  return count
```

Dynamic Programming

```
function TETRISDP(n):

   // dp[i][mask]: # ways to tile columns 0..i-1 with column i having configuration
//"mask"
   // There are 4 possible masks: 0, 1, 2, 3 (using 2-bit representation)

   let dp[0..n][0..3] be a 2D array initialized to 0
   dp[0][0] = 1   // Start with column 0 fully empty

   for i = 0 to n-1:
      for mask in {0,1,2,3}:
         ways = dp[i][mask]
         if ways == 0:
            continue

         // Enumerate all valid placements P that can be applied starting at column i
         // given that column i is partially filled as indicated by 'mask'.
         // Each placement P:
         //    - covers some cells in column i (and possibly column i+1),
         //    - uses one of the pieces (J, L, or O in an allowed rotation),
         //    - is valid only if all required cells are free.
         // Let P produce a transition: it covers k columns and leaves a new mask,
//new_mask

         for each valid placement P given state (i, mask):
            let (k, new_mask) = Transition(P, mask)
            if i + k <= n:
               dp[i+k][new_mask] = dp[i+k][new_mask] + ways

   return dp[n][0]    // Return count when all columns are processed and no spillover.
```

Data Structure: A 2D table dp[0..n][0..3] is used, where the first index indicates the column number and the second index is a 2-bit mask.
Fill Order: We fill in increasing column order from 0 up to n, ensuring that when we compute transitions from column i, we already know the number of tilings up to that point.
Runtime Analysis:
There are O(n) columns and 4 masks per column, so O(4n) states. For each state, a constant number of placements is checked. Hence, the algorithm runs in O(n) time multiplied by a constant factor that depends on the number of valid placements.

2. Counting the Number of Ways to Split a String into Words

Algorithm from HW 3

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

Dynamic Programming

```
function COUNTSPLITDP(S):
  n = length(S)
  let dp[0..n] be an array of integers, initially all 0
  dp[n] = 1   // Base case: one way to split an empty string

  for i = n-1 down to 0:
    dp[i] = 0
    for j = i to n-1:
      if S[i...j] is a valid word:
        dp[i] = dp[i] + dp[j+1]
  return dp[0]
```

Data Structure: A one-dimensional array dp[0..n] stores the number of ways to split the substring starting at each index.
Fill Order: We work backwards from the end of the string (i.e. from index n−1n-1n−1 to 0), ensuring that when computing dp[i], the subproblems dp[j+1] (for j≥i) have already been computed.
Runtime Analysis:
The outer loop runs O(n) times, and for each i the inner loop runs at most O(n) times, so the overall runtime is $O(n^2)$ (plus the cost of checking whether a substring is a valid word).

3. Determining if it is possible to buy exactly n cupcakes

Algorithm from HW 3

```
CUPCAKE(n):
  if n == 0:
      return TRUE
  if n < 0:
      return FALSE
  if CUPCAKE(n-6)
      return TRUE
  if CUPCAKE(n-9)
      return TRUE
  if CUPCAKE(n-20)
      return TRUE
  return FALSE
```

Dynamic Programming

```
function CUPCAKEDP(n):
  let dp[0..n] be an array of booleans, all initialized to FALSE
  dp[0] = TRUE  // 0 cupcakes can always be "bought" (buy nothing)

  for i = 1 to n:
    if i - 6 >= 0 and dp[i-6] == TRUE:
      dp[i] = TRUE
    if i - 9 >= 0 and dp[i-9] == TRUE:
      dp[i] = TRUE
    if i - 20 >= 0 and dp[i-20] == TRUE:
```

```
        dp[i] = TRUE
    return dp[n]
```

Data Structure: A simple one-dimensional Boolean array dp[0..n] is used, where each entry answers whether the corresponding cupcake total is achievable.
Fill Order: We iterate from 1 up to n so that when considering dp[i], the values for dp[i−6], dp[i−9], and dp[i−20] have already been computed.
Runtime Analysis:
The loop runs once for each number from 1 to n, and each iteration does a constant amount of work. Thus, the runtime is O(n).

4. Cutting a Piece of Wood of Size m * n

Algorithm from HW 3

```
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
```

Dynamic Programming

```
function RECTDP(P, M, N):
    // Create a 2D array dp[1..M][1..N]
    for m = 1 to M:
        for n = 1 to N:
            dp[m][n] = P[m][n]  // Initialize with the sale price for the whole piece
```

```
        // Try all vertical cuts
        for i = 1 to n - 1:
            dp[m][n] = max(dp[m][n], dp[m][i] + dp[m][n-i])

        // Try all horizontal cuts
        for j = 1 to m - 1:
            dp[m][n] = max(dp[m][n], dp[j][n] + dp[m-j][n])
    return dp[M][N]
```

Data Structure: A two-dimensional array dp[1..M][1..N], holds the best revenue for every subrectangle.

Fill Order: We loop over mmm from 1 to M and over n from 1 to N. When computing dp[m][n], all smaller subproblems (resulting from any allowed vertical or horizontal cut) have already been solved.

Runtime Analysis:
The two nested loops over m and n require O(M·N) iterations. Inside each cell, two inner loops run up to n and m iterations, respectively. Hence, the overall time complexity is O(M·N·(M + N)).