Part 2: Complexity Analysis

Question 1

Complexity

Time Complexity : *O(2^{n+m})*
Space Complexity : *O($max(m, n)$)*

Explanation

This method uses recursion to find the number of distinct paths from the top-left corner of a grid to the bottom-right corner.
For each step, the method makes a recursive call to move down and another to move right hence a factor of 2.

Question 2

Complexity

Time complexity: *O(n*\m)*
Space complexity: *O(n)*

Explanation

From the ArrayList "pieces" containing 'm' objects, this function finds all possible combinations of objects
that can form the target value of length 'n'. In the worst cases,

- The first loop run $n + 1$ times.
- The inner loop which iterates on the number of objects in 'pieces', runs m times.

The operation comparison of if a string of length 'piece.length()' matches the current piece takes O(piece.length()) time.

In the worst case, this comarison is done for each piece in the inner loop hence additional O(m * piece.length()) time complexity
during the construction of new combinations, lists copied have sizes proportional to the target hence and overal complexity of
O(n * m).

Question 3

Complexity

Time complexity: *O(n*\m)*
Space complexity: O(n)

Explanation

This function returns true if the target of lenght 'n' can be obtained by combinations of elements in the
"options" array of size 'm' and false otherwise. Using dynamic programming, the main loop iterate over
each index from 0 untill the target's value and each iteration, the inner loop iterates over the options array hence making it a nested loop with time complexity O(n*m).

Question 4

What will be the time complexity using recursion instead of dynamic programming ?

- For MistFonction2, the time complexity of the recursive function is *O(m^n)*
  because for each of the 'n' characters in 'target' it makes n calls for each of the 'm' elements in 'pieces'.

- For MistFonction3, the time complexity of the recursive function is *O(*options^n*)* because it makes n calls for each element in the 'options' array.

Part 3: Stacks

Problem 1

Algorithm:

Time Complexity: *O(*n^3*)*

Space Complexity: *O(n)*

Problem 2

Time Complexity:

- pairDestroyer *O(*n^3*)*
- destroyer *O(*n^2*)*
- emptyTemp *O(*n*)*

Space Complexity: *O(n)*