Solution: Dynamic Programming

1. **Solve Dynamic Programming Problem and Compare with Naïve approach**
   You are playing a puzzle. A random number N will be given, you have blocks of length 1 unit and 2 units. You need to arrange the blocks back to back such that you get a total length of N units. In how many distinct ways can you arrange the blocks for given N.
   a. Write a description of approach to solve it using Dynamic Programming paradigm
   b. Implement a function **blockpuzzle_dp(N)** that solves this problem using Dynamic Programming using either top-down or bottom-up approach
   c. Write pseudocode for the brute force approach
   d. Compare the time complexity of both the approaches
   e. Write the recurrence formula for the problem
   Name your file **BlockPuzzle.py**

   Example 1:

   Input: N=2, Result: 2

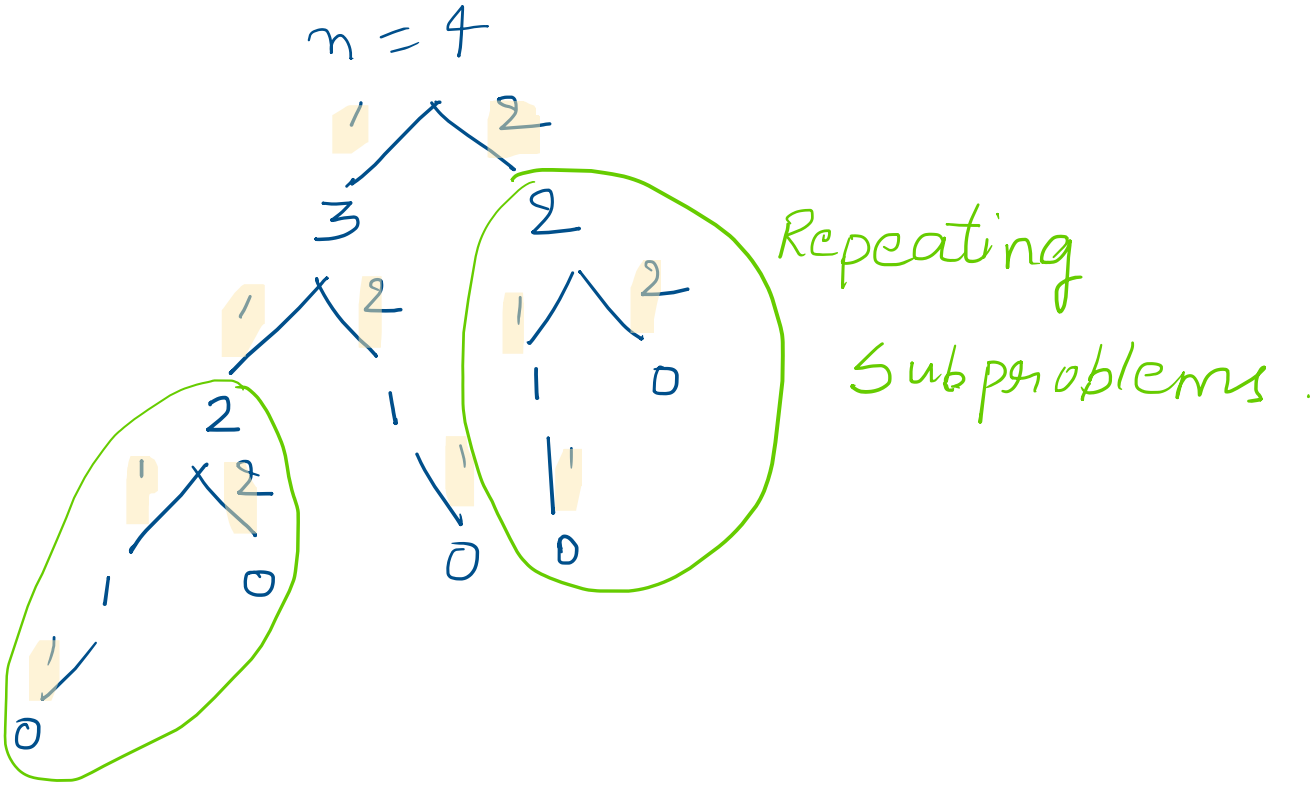
   Explanation: There are two ways. (1+1 , 2)

   Example 2:

   Input: N=3, Result: 3

   Explanation: There are three ways (1+1+1, 1+2, 2+1)

a. We have repeating subproblems as shown in the figure below. We can cache these use the dynamic programming technique to find the result.



b. Following is code for bottom-up approach

```
def blockpuzzle_dp(n):
    if n==1: return 1

    memo = [0]*(n+1)
    memo[1] = 1
    memo[2] = 2
    for i in range(3, n+1):
        memo[i] = memo[i-1] + memo[i-2]

    return memo[n]
```

c. brute force approach

```
def blockpuzzle_bruteforce(n):
    if (n <= 0):
        return 0
    if (n == 1):
        return 1
    if(n ==2):
        return 2
    return blockpuzzle_bruteforce(n-1) + blockpuzzle_bruteforce(n-2)
```

d. Brute force approach has time complexity of theta(2^n) (If we draw recursion tree: it will have n levels and at level k 2^k amount of work will be done)

The dynamic programming approach has time complexity of theta(n).

The dynamic programming approach solves this problem extremely faster compared to the brute force approach.

e. Recurrence Formula:

F(n) = 0 for n = 0

F(n) = 1 for n = 1

F(n) = 2 for n = 2

F(n) = F(n-1) + F(n-2) for all values of n

2. **Solve a problem using top-down and bottom-up approaches of Dynamic Programming technique**

Two players A & B are playing a game. The rules of the game are:

At the start one number N will be given. The player who starts would have to pick a number i such that 0<i<N, the condition is that N%i == 0. The second player would pick a number j from N-i, satisfying the condition 0<j<(N-i). And the game goes on until there is no

more possibility of making any selection. Each player would play in turns, and A always starts the game. Assume both players play optimally.

Given a number N return if A would win the game or not.

Example 1:

Input: N=2, Result: True

Explanation: A choses 1, and B has no more numbers to chose

Example 2:

Input: N=3, Result: False

Explanation: A choses 1, B choses 1, and A has no more numbers to chose

a. Implement a solution to this problem using Top-down Approach of Dynamic Programming, name your function **game_topdown(N)**

```
def game_topdown(n):
    memo = [None]*(n+1)
    return game_topdown_helper(n, memo)

def game_topdown_helper(n, memo):
    print('solving for: ', n)
    if n<=1:
        return False

    if memo[n] is not None:
        return memo[n]

    for x in range(1, n):
        if(n%x==0):
            if not game_topdown_helper(n-x, memo):
                memo[n] = True
                return True
    memo[n] = False

    return False
```

b. Implement a solution to this problem using Bottom-up Approach of Dynamic Programming, name your function **game_bottomup(N)**
Memo[i] holds the value of the game result for A for n=i. 'A' will check for all factors and see if there is any factor that will make 'B' lose.

```
def game_bottomup(n):
    memo = [False]*(n+1)

    memo[0] = False
    memo[1] = False

    for i in range(2, n+1):
        for j in range(1,i):
```

```
            if(i%j == 0 and memo[i-j] == False):
                    memo[i] = True
                    break

    print(memo)
    return memo[n]
```

c. Explain your approach to solve this problem. How is your top-down approach different from the bottom-up approach?

The bottom-up approach solves the problem from i=2 onwards until the value of n. game_topdown_helper() recursively solves the problem for n until base case is reached following top down approach.

d. What is the time complexity and Space complexity using Top-down Approach

Time: $O(n^2)$ [x goes from 1 to n and for each x helper function is called at zmost n times]

Space: theta(n)

e. What is the time complexity and Space complexity using Bottom-up Approach

Time: $O(n^2)$

Space: theta(n)

f. Write the subproblem and recurrence formula for your approach

Subproblem: f(i) can be given as: will A win the game for a value of N=i.

Recurrence Formula:

F(n) = False for n = 0 or 1

F(n) = !F(n-x) for, 0<x<N such that n%x = 0 for n>1