1A) To solve this puzzle with dynamic programming we are going to need to break the problem down into smaller subproblems. In this case
the puzzle can either be assembled with pieces or 1 unit or 2 units to compile up to n. We can approach this with dynamic programming
by recognizing that the problem can be broken down into its previous problems. If you need to a puzzle of length N, then you HAVE to get
to N by adding either a 1 unit puzzle piece or a 2 unit puzzle piece. With that known, you know that the previous length of the puzzle was
either N-1 or N-2, and that the solution to N would be (N-1)+(N-2) since those would be all of the previous puzzle combinations, plus the added
1 unit and 2 unit to get to N. Once we realize this, we can essentially put into effect a simple fibonacci sequence for the previous
steps that will get us the correct answer for any number of puzzle units.

1C) Brute Force Pseudocode

```
blockpuzzle_bf(N):
    if N < 0:
        return 0
    elif N == 0:
        return 1
    else:
        return (blockpuzzle_bf(N-2) + blockpuzzle_bf(N-1))
```

1D) Time complexity of the dynamic programming solution would be theta(N) (linear) , where the brute force would be theta(2^N) (exponential).
The dynamic programming solution has a lower order of growth.

1E) Recurrance Relation

```
    t(n) = 1                    for n = 0
    t(n) = 1                    for n = 1
    t(n) = t(n-1)+t(n-2)        for n > 1
```

2c) This was a diffucylt problem to approach from both directions, and I struggled with using memoization and recursion with the top down approach,
but wanted to find a way to at least attempt it. For bottom up it was quite simple. We had 2 base cases defined such that if on your turn if N = 1 you lost
and if N = 2 you won. I put these into an array with boolean values [0,1] to start, along with N = 3 as false, since it has no options but to take 1.
Working up from there, you only need to check the next iteration in the array's factors, and if any of them have a positive boolean value at their location in the array, then it would be possible to win if you started at that iteration. So for N = 4, you had to look at its factors of 1 and 2. Since 2 is possible as established in the
base case, we know that its possible to win at 4, so array[3] gets the value 1. You continue to iterate through your array until you reach the N-th iteration. My top down
approach I attempted to implement a helper function to find if a factors of N had valid moves. It would return a boolean 1 or 0 depending on if for a given N value, there was

a move that could ensure a win. Once this was returned, it was saved in the memo array as a true or false. As the recursion kept running by deducting from N and running the
function again. If the memo ever already had a true or false value for a given number, it would stop and not run the recursion or helper function.

2d) I believe the time complexity for my solition to the top down approach was BigO N^2, but to be honest trying to do recursion and memoization made it difficult for me to track.

2e) Bottom-up approach has runs on a linear growth for both time and space. O(n).

2f) Recurrance Relation

t(n) = 1	for n >= 3
t(n) = t(n)+c	for n < 3 - for my bottom up
t(n) = 2t(n) + conquer	for n < 3 - for my top down

DEBRIEF:
1: 10 hours on the assignment, not including the discussion which I will begin after this.
2. Moderate, the questions were not explained clearly and left some thing ambigious. Should not need to check 4 Ed posts to undertand the question.
3. This weeks material, 70%, but the material from the previous weeks still onlu 30-50% but no time to go back and review.
4. Need better ways to learn how to evaluate our code or other peoples code for a recurrance relation. I have no real idea how to do it on anything.