# Question 1

## Questions:

1. Explain, in general terms and your own words, what memoization is (0.5 pts)
2. Consider the following code:

```
def func(n):
  if n == 0 or n == 1:
  return n
  else:
  return func(n-1) + func(n-2)
```

3. What does it do? (0.5 pts)
4. Is this an example of a divide-and-conquer algorithm? Explain. (0.5 pts)
5. What is its time complexity? (0.5 pts)
6. Implement a version of the code above that use memoization to improve performance. Provide this as ex1.3.py. (2 pts)
7. Perform an analysis of your optimized code: what is its computational complexity? (3 pts)
8. Time the original code and your improved version, for all integers between 0 and 35, and plot the results. Provide the code you used for this as ex1.5.py. (2 pt)
9. Discuss the plot and compare them to your complexity analysis. (1 pt) 1 Yep, not a typo, we are not talking about “memorization”.

## Part 1:

Memoization is a process where values from a computation are stored in a 'cache' and then retrived from that same 'cache' the next time the same computation is done.

## Part 2/3:

The code is to calculate the fibonacci number at index 'n', this function works by recursively adding together n-1 and n-2 until the base cases are reached (0 or 1) then all of the numbers are added back together.

## Part 4:

Yes, the function breaks down the number (n) until the base case (0 or 1) is reached. At this point the function will then add together all of the sub-solutions to get the fibonacci number at index 'n'.

## Part 5:

Time complexity is T(2^N)

## Part 6:

ex1.3.py is in the Q1 folder

## Part 7:

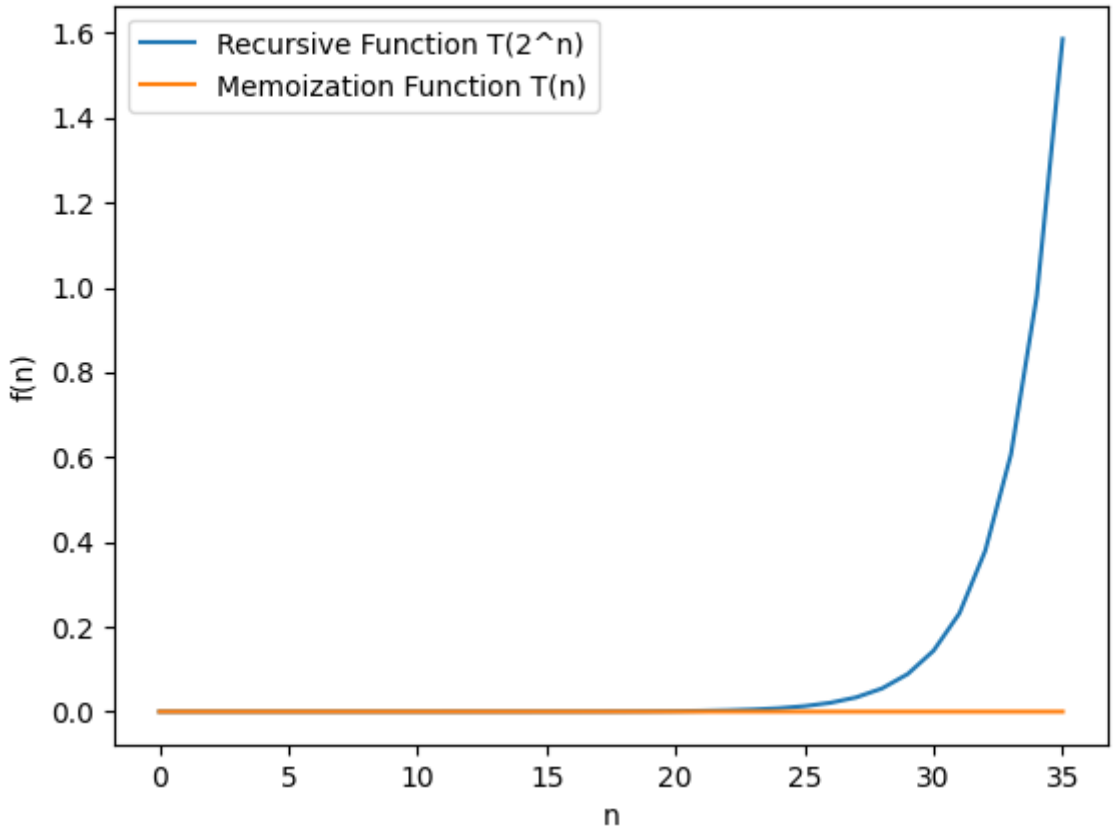
Looking at the previous computational complexity of the pure recursive function, it had O(2^n), which is due to the amount of function calls that occur.

Since we are using memoization there is going to be less function calls. The data structure that we use is a dictionary, which fills up linearly through the function (up from 2 to n). Once the dictionary is full, there is no more need for recursive function operations, and the only operations are lookups in the dictionary as well as logic. which means there is a computational complexity of O(n).

## Part 8:

Added ex1.5.py to repo

**Plot:**




## Part 9:

The plot shows two different graphs. Both graphs seem almost constant to begin with, once the graphs reach ~n=20 mark they begin to seperate. The Recursive function begins to grow exponentially (affirming T(2^n)), while the Memoization function remains almost constant (affirming T(n)). This is accurate with the analysis that was done in question 7.