# Exercise 1

1. In general terms, memorization is a technique which speeds up the execution time of a function. It does this by storing the results of the function call in the cache when a function is called, so that when the same function is called again, the cached results are returned. This speeds up computing time, which in return improves performance, especially for programs which call the same function multiple times.
2. # Fibonacci function  
   *def* func(*n*):  
    if *n* == 0 or *n* == 1:  
    return *n* else:  
    return func(*n* - 1) + func(*n* - 2)
3. This function is an example of a tail optimized recursion Fibonacci Sequence. The function takes a value n, and if n is either 0 or 1, it returns n, otherwise the function returns the sum of the previous two Fibonacci numbers.
4. Yes, the Fibonacci Sequence code provided is an example of a Divide-and-Conquer Algorithm. This is because the function takes the nth Fibonacci number and divides it into two subproblems if it is not equal to 1 or 0. The subproblems it divides into is the (n – 1) and (n – 2) Fibonacci numbers. These get solved recursively until the Fibonacci number, n, is equal to either 1 or 0. When the function reaches 1 or 0, the result of the function is returned. The subproblems (n – 1) and (n – 2) are then added together to find the nth Fibonacci number, which the function is looking for.
5. If n is either 1 or 0, then the time complexity is O(1) which is of constant time complexity and the most efficient, because it returns a constant immediately. But, if n is greater than 1, then it returns func(n – 1) + func(n – 2) which is less efficient because the function calls itself twice. When a function is called it either returns 0 or 1, or goes through another iteration of returning func(n – 1) + func(n – 2). So the time required to find the nth Fibonacci number, is g(n) = g(n – 1) + g(n – 2) + O(1). Both g(n – 1) and g(n – 2) are recursive, so they each split into their own branches, and repeat, increasing exponentially each split. Therefore, the time complexity when n is greater than 1 is O(2n), which is of exponential time complexity.
6. *def* fibmemo(*n*, *cache*={}):  
    if *n* == 0 or *n* == 1:  
    return *n* else:  
    if *n* in *cache*:  
    return *cache*[*n*]  
    else:  
    *cache*[*n*] = fibmemo(*n*-1) + fibmemo(*n*-2)  
    return *cache*[*n*]
7. Analysis of Optimized code:  
    - The function fibmemo has arguments it uses to find the Fibonacci Sequence for the nth Fibonacci number, and an empty dictionary to store any previous results in the function computes.  
    - if n is 0 or 1, the function returns n which is of time complexity O(1).  
    - if n is greater than 1, it looks in the dictionary cache for the nth value, if it is in cache then it is returned.  
    - if the value is not in the cache, the function computes the nth Fibonacci number and stores it in the cache, and returns the value.  
    - Since each Fibonacci number is computed once and then stored in the cache, there is no reason for the function to continually calculate the next Fibonacci number through recursion. This means the function can go directly to the cache for the next Fibonacci number and return the desired value. So the time complexity is O(n) and space complexity is O(n) so the computational complexity is O(n).

import timeit

import matplotlib.pyplot as plt

import numpy as np

# Memoization

*def* fibmemo(*n*, *cache*={}):

if *n* == 0 or *n* == 1:

return *n*

else:

if *n* in *cache*:

return *cache*[*n*]

else:

*cache*[*n*] = fibmemo(*n*-1) + fibmemo(*n*-2)

return *cache*[*n*]

# Recursive

*def* func(*n*):

if *n* == 0 or *n* == 1:

return *n*

else:

return func(*n* - 1) + func(*n* - 2)

if \_\_name\_\_ == "\_\_main\_\_":

memoTime = [timeit.timeit(*lambda* : fibmemo(i), *number*=1) for i in range(36)]

recTime = [timeit.timeit(*lambda* : func(i), *number*=1) for i in range(36)]

print(sum(memoTime))

print(len(memoTime))

memoTimeSum = np.cumsum(memoTime)

recTimeSum = np.cumsum(recTime)

plt.plot(memoTimeSum \* 100000, *label*="Memoization (x100000)")

plt.plot(recTimeSum, *label*="Recursive")

plt.title('Time taken for Fibonacci Sequence for Two Different Function Types:\nMemoization and Recursive')

plt.xlabel("Fibonacci Number")

plt.ylabel("Time (s)")

plt.legend(*loc*='upper left')

plt.show()

plt.close()

1. Chart, line chart

   Description automatically generated  
   The plot clearly shows that the memorization function has complexity of O(n), linear complexity, while the recursive function has complexity of O(2n), exponential complexity. This agrees with the complexity analysis determined in the previous questions.