Exercise 1

1. Memorization is the technique of storing a very computationally demanding algorithm’s output data and using that when the function is called with different data values to speed up the calculations.
2. No ans required.
3. It recursively calls itself and the function func takes an integer n as an argument and returns the n-th Fibonacci number.
4. It is not a divide and conquer as it does not break the problem into subproblems that are easier to solve computationally rather it calls itself multiple times creating 2 instances of the same problem.
5. Time complexity is the time taken for a function to solve a certain problem usually expressed using Big O notation which is the average time taken to solve the problem regardless of the type of hardware. As we have 2 recursive function calls the complexity increases as a magnitude of 2 therefore the time complexity is O(2^n)

def func(n, memo={}):

if n == 0 or n == 1:

return n

elif n in memo:

return memo[n]

else:

result = func(n-1) + func(n-2)

memo[n] = result

return result

1. Computational complexity is the number of calculations, memory and recourses required to solve the problem using our algorithm. As the optimized code stores the Fibonacci numbers in a list instead of calculating them every time thereby in the worst case, the number of function calls grows exponentially with the size of the input n. So the complexity is O(n)

import timeit

import matplotlib.pyplot as plt

def unoptimized\_fib(n):

if n == 0 or n == 1:

return n

else:

result = unoptimized\_fib(n-1) + unoptimized\_fib(n-2)

return result

def optimized\_fib(n, memo={}):

if n == 0 or n == 1:

return n

elif n in memo:

return memo[n]

else:

result = optimized\_fib(n-1, memo) + optimized\_fib(n-2, memo)

memo[n] = result

return result

def unoptimized\_fib\_call(n):

unoptimized\_fib(n)

def optimized\_fib\_call(n):

optimized\_fib(n)

unoptimized\_array\_y = []

optimized\_array\_y = []

array\_x = []

for i in range(35):

unoptimized\_time = timeit.timeit(lambda: unoptimized\_fib\_call(i), number=1)

unoptimized\_array\_y.append(unoptimized\_time)

optimized\_time = timeit.timeit(lambda: optimized\_fib\_call(i), number=1)

optimized\_array\_y.append(optimized\_time)

array\_x.append(i)

# plotting the points

plt.plot(array\_x, unoptimized\_array\_y, label = "unoptimized\_fib\_call")

plt.plot(array\_x, optimized\_array\_y, label = "optimized\_fib\_call")

# naming the x axis

plt.xlabel(' execution time (seconds)')

# naming the y axis

plt.ylabel('nth input number')

# adding a legend

plt.legend()

# function to show the plot

plt.show()

 unoptimized\_fib\_call Executed in 1.9363947500023642

optimized\_fib\_call Executed in 1.4874996850267053e-05

A picture containing graphical user interface

Description automatically generated

The plot describes the complexity analysis results as the graph of the un-optimized code is a function of O(2^n) (shown in blue) while the optimized is a function of O(n) (shown in orange) as described by the graph

Exercise 2

the code takes in an array (arr),  a low and a high integer which corresponds to the indexes. It first uses a pivot and switches the values to the left if they are greater than the pivot and returns the high index. This high index is stored in the variable pi

Then it recursively calls itself with the same arguments but this time instead of high it uses a value of pi-1 so the high gradually decreases and likewise on the second recursive call has the low replaces with pi+1 to make its way upwards. These functions appropriately sort the given array ”arr”

On average this algorithm divides the array into 2 smaller halves using the divide and conquer algorithm which need to be sorted equally. This makes the complexity O (n log n). however on the worst case where the divided array is completely full on one side and empty on the other the worst case complexity is O (n^2).

Exercise 3

Exercise 4:

Question 1) In this question, it asks to compare the pros and cons of using arrays vs linked list,

Advantages of Arrays:

Memory allocation: Arrays are stored in contiguous blocks of memory,which helps in allocating memory

Simplicity: it is simple to use often a built in type

Retrievals: it is way easier to retrieve data from memory using indexing, which makes random accessing much easier.

Disadvantages of Arrays:

Size is fixed: it is awkward to add/remove elements, once it is created, it can’t change dynamically.

Not memory efficient: it can waste a lot of space if some spaces in the memory has not been used.

Advantages of linked list:

Memory allocation: it can grow and shrink dynamically, since each node requires a pointer that is stored in the memory that points at another pointer.

flexibility : it is more flexible to add and remove elements, since it is based on pointers more than storing the actual data

Disadvantages of linked lists:

Memory allocation: slow to access elements at specific index, since it has to go from the beginning of the linked list all the way until it find the specific index or data. It might even be in the last node and that is not efficient at all compared to arrays

Memory: it takes more memory than arrays since it needs to store every pointer with the data node.

In conclusion, linked list is a better option in terms of inserting and removing data elements , on the other hand, picking arrays to acquire random access to all the elements is a wise decision.

Exercise 5