UNIVERSITY OF CALGARY

ENSF 338

Assignment #2

Lab Section #02, Group #48

Tahmeed Mahmud(30158740), Daniel Mellen(30158835)

Feb. 9, 2023

Work Performed By Each Member

Q1: Tahmeed
Q2: Daniel
Q3:Tahmeed
Q4:Daniel
Q5:Daniel & Tahmeed

50% of work was done by Tahmeed and 50% by Daniel.

Question 1

1. By saving the outcomes of expensive function calls and delivering the cached results when the identical inputs are received again, the technique of memory is used to optimize dynamic programming techniques. Memoization enhances the algorithm's performance and aids in the reduction of duplicate computations.

2. The nth Fibonacci number is returned via a recursive function that is implemented in the code. Starting with 0 and 1, the Fibonacci sequence is a set of numbers where each number is the sum of the two before it.

3. In the code, the nth Fibonacci number is calculated.

4. No, this is not a divide-and-conquer algorithm in action. Divide-and-conquer algorithms split a problem into smaller sub-problems that can be resolved on their own and then combined to address the main problem. By computing the lower values iteratively, the code above finds the solution.

5. The code has an exponential time complexity of O(2n). This is due to the fact that the function receives two calls for each call it receives, resulting in a high number of function calls for big values of n.

6.

```
    def mem_func(n, memo={}):
        if n == 0 or n == 1:
            return n
        if n in memo:
            return memo[n]
        memo[n] = mem_func(n-1, memo) + mem_func(n-2, memo)
        return memo[n]
```

7. The memoized version of the code has a linear computational complexity of O(n). This is because fewer function calls are required because the result of each function call is saved in the memo dictionary and is retrieved for subsequent calls with the same input.

8. Use Python's time module to time both the original code and the updated version. This is coded as:

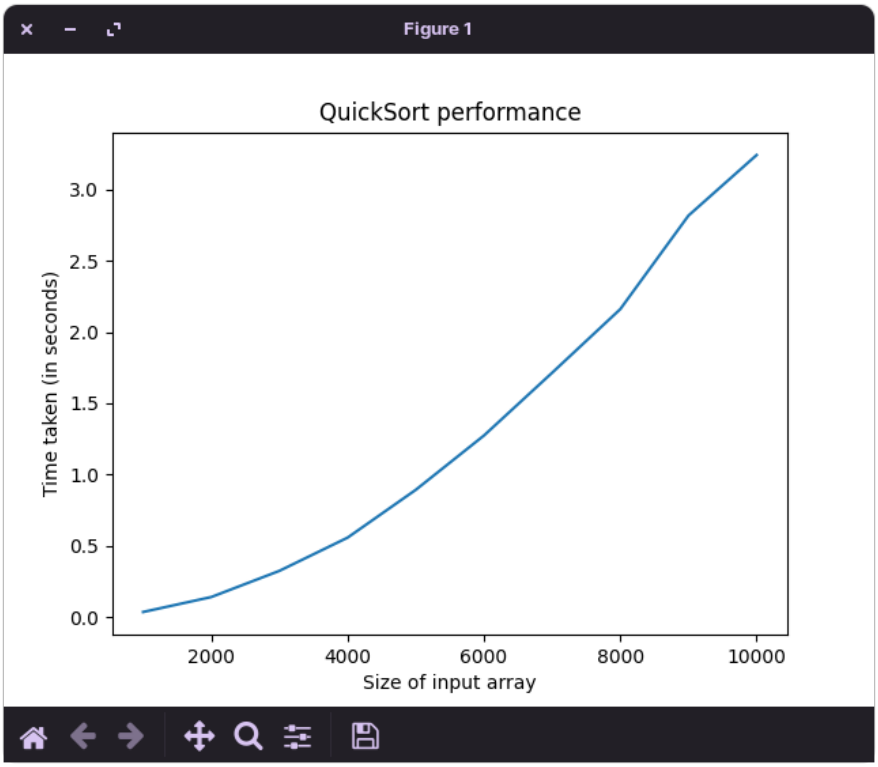
```
import time

def fib(n):
    if n == 0 or n == 1:
        return n
    return fib(n-1) + fib(n-2)

def memo_fib(n, memo={}):
    if n == 0 or n == 1:
        return n
    if n in memo:
        return memo[n]
    memo[n] = memo_fib(n-1, memo) + memo_fib(n-2, memo)
    return memo[n]

numbers = list(range(36))
fib_times = []
memo_fib_times = []
for n in numbers:
    start = time.time()
    fib(n)
    end = time.time()
    fib_times.append(end-start)
    start = time.time()
    memo_fib(n)
    end = time.time()
    memo_fib_times.append(end-start)
```

9. The running times of the original code and the revised version for each value of n are displayed on the results plot. The plot will demonstrate that, especially for higher values of n, the enhanced version runs substantially more quickly than the original version. The plot and complexity analysis can be compared to see how well the actual running time matches the anticipated time complexity.

Question 2

1. The quicksort method is being used by the program to sort an array. This program is broken down into two functions, func1 and func2. Func1 is the main recursive function that implements the quicksort method of data organization by dividing the array into two subarrays and recursively sorting them. Func2 contains "low" and "high" pointers that partition the array around the pivot element chosen by func1. QuickSort's average-case time complexity is O(n log n) for n array elements. Each func1 recursive call processes half of the elements. The time complexity of func2 is O(n), because the while loop must process each element once.

2. Plot graph:




```
import sys
import json
import matplotlib.pyplot as plt
import time
import threading
threading.stack_size(33554432)


sys.setrecursionlimit(20000)

def func1(arr, low, high):
    if low < high:
        pi = func2(arr, low, high)
        func1(arr, low, pi-1)
        func1(arr, pi + 1, high)

def func2(array, start, end):
    p = array[start]
    low = start + 1
    high = end
    while True:
```

```
        while low <= high and array[high] >= p:
            high = high - 1
        while low <= high and array[low] <= p:
            low = low + 1
        if low <= high:
            array[low], array[high] = array[high], array[low]
        else:
            break
    array[start], array[high] = array[high], array[start]
    return high

def main():
    filename = "338A2Q2.json"
    try:
        with open(filename, 'r') as file:
            try:
                data = json.load(file)
            except json.decoder.JSONDecodeError:
                print(f"Error: {filename} does not contain valid JSON
data.")
                return

            times = []
            sizes = []
            for arr in data:
                start_time = time.time()
                func1(arr, 0, len(arr) - 1)
                end_time = time.time()
                times.append(end_time - start_time)
                sizes.append(len(arr))

            plt.plot(sizes, times)
            plt.xlabel("Size of input array")
            plt.ylabel("Time taken (in seconds)")
            plt.title("QuickSort performance")
            plt.show()
    except FileNotFoundError:
        print(f"Error: {filename} not found.")

if __name__ == "__main__":
```

```
main()
```

3) The timing results look similar to what O(n * log n) would look in a graph. The result is fairly consistent but there are slight fluctuations in the curve, especially towards the end. There is no “model” provided to measure consistency of the result but the time taken to perform the operation varies more greatly towards the end (larger sizes of input arrays). However, after running the program multiple times I have received plots with more and less variation.

4) I am not sure how to optimize it further for the following reasons:

-The input size is very large and QuickSort is the fastest sorting method I know of.
- I could change the pivot element to be the median of the array instead of the very first element but I would have to find the median which would take additional time and computational power.
-I could rewrite the entire program as using an in-place sorting algorithm but not enough instructions are provided on the context of the func1 and func2 and the program overall. (would the size of the JSON input data vary?, would the datatype always be a list of integer lists?, etc)

Question 3

1) Interpolation search is better than binary search when the values are uniformly distributed, as it narrows down the search space more efficiently. Likewise it can be slower with smaller non-uniform datasets. However it can handle non uniformly distributed data, (which binary search can’t). Another benefit of interpolation is that it provides an informed estimate as to where the required value could be by utilizing the distribution of values in the array.

2)The performance of interpolation search CAN be affected if the data follows a different distribution. If the data follows a different distribution, such as a normal distribution, the accuracy of the estimated position may be affected. This can result in a larger search space and slower performance. Binary search would be faster in that scenario because binary search does not rely on any assumptions about the data distribution.

3) If we want to modify the Interpolation Search to follow a different distribution, the if and else statements could be changed to work with the new data distribution more efficiently. If we have a formula for the calculation of the estimated position of the target item, that too will be modified to reflect the new data distribution.

4a) If the data is unsorted, binary search and interpolation search cannot be used because they both assume the data is sorted. Therefore Linear search is the correct option in such scenarios.

4b) Linear search can outperform both interpolation and binary search in certain cases, such as when the data set is extremely small or when the resource cost to compare between elements is very high. If the data set is very small, the overhead of performing either Interpolation or Binary search may be larger than the time it takes to simply iterate through all elements in the data set using Linear search. One way to improve Binary and Interpolation search for unordered data sets is sorting the data set before searching, using a separate function.

QUESTION 4

1. **G**

Arrays benefit from random access to elements, which means that any element can be retrieved in O(1) time. However, inserting or deleting elements from an array necessitates shifting the elements after the insertion or deletion point, which takes O(n) time where n is the array's size. Because you only need to update the next pointers of the surrounding elements, inserting or deleting elements anywhere in the list takes O(1) time. The disadvantage of linked lists is that you cannot access elements in O(1) time by traversing the list from the head to the desired element, which takes O(n) time.

2. **G**

To reduce the impact of the replace function in arrays, make sure that it always replaces an element at the end of the array and pops the last element. You don't have to shift any elements this way, and the operation takes O(1) time.

3. **G**

a. A singly linked list cannot perform selection sort because the sort requires swapping elements, which is not possible. As with a regular array, the expected complexity would be O(n2).

b. Insertion sort is feasible for a singly linked list because it is an efficient sorting algorithm for small data sets or partially sorted data sets. The time complexity is expected to be O(n2), which is the same as for a regular array.

c. Merge sort is feasible for a singly linked list because it can be used to sort a linked list by dividing it recursively into smaller sub-lists and then merging the sub-lists back together. The time complexity is expected to be O(n log n), which is the same as for a regular array.
d. For a singly linked list, bubble sort is feasible, and the expected time complexity is O(n2), which is the same as for a regular array.
e. Quick sort is not possible for a singly linked list because the sort requires swapping elements, which a singly linked list does not allow. As with a regular array, the expected time complexity is O(n2).

QUESTION 5

1.

a. Because stacks are based on the last-in, first-out (LIFO) principle, insertion (push) adds the newly inserted data at the head. This means that the most recently added item to the stack will be the first to be removed. By inserting new data at the top, the most recently added item will always be the first to be removed when the stack is popped.

b. No, we cannot insert data at the end of a linked list in a stack because the first item added would be the first one removed, violating the LIFO principle.

c. Inserting data at the end of the linked list would take the same amount of time as inserting at the beginning (O(1)). However, because we would need to traverse the entire linked list to reach the item at the head, popping would have a longer operation time of O(n).

2. G

a. To improve the efficiency of enqueue operations, we added a new pointer in Queues that points to the tail of the linked list. Because enqueue operations add data to the

linked list's end, having a direct reference to the tail allows us to insert new data in O(1) time rather than O(n) time if we had to traverse the entire linked list to find the tail.

b. We can implement a Queue without the tail pointer, but it will make enqueue operations less efficient because we will have to traverse the entire linked list to find the end each time we insert new data.

c. Without the tail pointer, enqueue operations would have a time complexity of O(n), whereas dequeue operations would still have a time complexity of O. (1). Enqueue and dequeue operations with the tail pointer have a time complexity of O. (1).

3. **g**

a. In stacks, insertion (push) still adds newly inserted data to the top, according to the LIFO principle.

b. In stacks, we cannot insert data at the end of a linked list.

c. With a circular doubly linked list, the operation time for pushing and popping data from the stack would still be O(1).

To efficiently enqueue data in Queues, the tail pointer would still be required.

d. No, changing the enqueue and dequeue behavior would still result in a stack, not a queue. The circular doubly linked list would not alter the queue data structure's underlying principles.