UNIVERSITY OF CALGARY

ENSF 338 W23
Practical Data Structures & Algorithms

Lab Section 02
Assignment 02

Group 13

Mehvish Fatima (30161318)
Al Farhana Siddique (30157968)

February 15, 2023

https://github.com/mehvishshakeel/ENSF_338/tree/main/Assignment2

Work load distribution between the members:

Exercises	Done by:
Ex1	Mehvish
Ex2	Mehvish
Ex3	Mehvish & Al Farhana
Ex4	Al Farhana
Ex5	Al Farhana

EXERCISE 1

1. Explain, in general terms and your own words, what memoization is (0.5 pts)

Memoization is basically a method of coding a function where we store the results for the inputs and return them accordingly rather than calculating it again and again.

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)

This is a function with one argument. It has an if statement where if the argument value (n) is either 0 or 1 the function will return the argument value (1 or 0). If n is not, it will return the sum of another two functions taking the arguments as one and two less than the previous argument. This will keep on going until the function returns 1 or 0.
This is called a recursive function which takes in an argument number (n) and ultimately we will get an output of the nth Fibonacci number.

4.Is this an example of a divide-and-conquer algorithm? Explain. (0.5 pts)

No, this is not a divide-and-conquer algorithm since at no point do we break it down into sub-problems in order to solve it quicker instead, it might call itself multiple times.

5. What is its time complexity?(0.5 pts)

As the value of n increases, the time and work done to get the final output increase. When the function is called the function will call two more functions so the number of times the function gets called will increase with 2^n. Hence it has an exponential complexity of O(2^n).

6. Implement a version of the code above that use memoization to improve performance. Provide this as ex1.3.py.(2 pts)

```
feb_seq = {}

def func(n):
    if n in feb_seq:
        return feb_seq[n]
    else:
        if n == 0 or n == 1:
            feb_seq[n] = n
            return feb_seq[n]

        else:
            feb_seq[n] = func(n-1) + func(n-2)
            return feb_seq[n]
i=0
feb = []
while True:
    n = int(input("input the last number of the Fibonacci sequence:" ))
    seq = func(n)



    while i < len(feb_seq):
        feb.append(feb_seq[i])
        i += 1

    print (seq) #print the last number of the Fibonacci sequence
    print (feb) #print the Fibonacci sequence .
```

7. Perform an analysis of your optimized code: what is its computational complexity?(3pts)

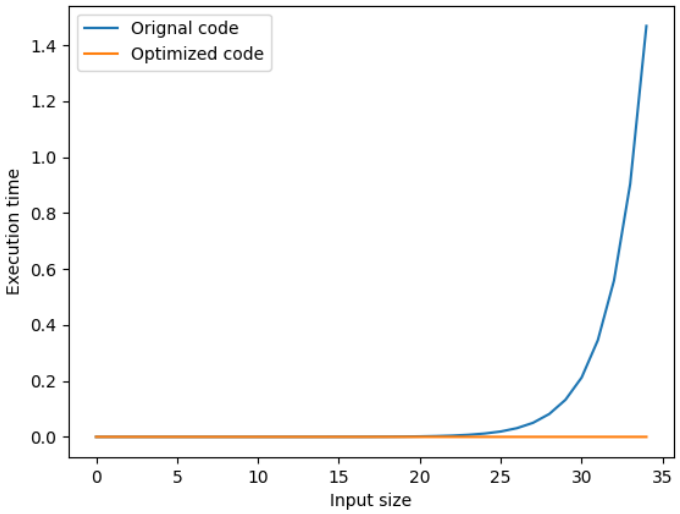
As the value of n increases, the time and work done to get the final output increases, and the value for each n input gets saved in an array(feb_seq). The use of memoization saves a lot of time as the function will not be called repeatedly for the same n value since we can call it directly from the array. So total amount of work done by the function is proportional to the unique values of n imputed. Hence the optimized code has a computational complexity of O(n).

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.(2pt)

```
import matplotlib.pyplot as plt
import numpy as np
import timeit

#Optimized code
feb_seq = {}
def func(n):
    if n in feb_seq:
        return feb_seq[n]
    else:
        if n == 0 or n == 1:
            feb_seq[n] = n
            return feb_seq[n]

        else:

            feb_seq[n] = func(n-1) + func(n-2)
            return feb_seq[n]
def func2(t):
    for t in range(35):
        func(t)


#Orignal code
def func_o(n):
    if n == 0 or n == 1:
        return n
    else:
        return func_o(n-1) + func_o(n-2)

def func2_o(t):
    for t in range(35):
        func_o(t)

Etime_orig = []
Etime_opt = []
Xval = []
```

```
i=0
for i in range (35):

    elapsed_time_o = timeit.timeit(lambda:func_o(i),number=1)
    #print(f"Execution time for orignal is {elapsed_time_o} seconds")
    Etime_orig.append(elapsed_time_o)

    elapsed_time = timeit.timeit(lambda:func(i),number=1)
    #print(f"Execution time for optimized is {elapsed_time} seconds")
    Etime_opt.append(elapsed_time)

    Xval.append(i)


plt.plot(Xval,Etime_orig, label = "Orignal code" )
plt.plot(Xval, Etime_opt, label = "Optimized code")
plt.legend()

plt.xlabel('Input size')
plt.ylabel('Execution time')
plt.show()
```

9. Discuss the plot and compare them to your complexity analysis. (1 pt)



As we can see in the plot the original code shows an exponential graph corresponding to its complexity O(2^n) whereas the optimized code shows a linear graph corresponding to its complexity O(n).

EXERCISE 2

```
import sys
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
```

1. Explain what the code does and perform an average-case complexity analysis. Describe the process, not just the result. (2pts)

This code is an example of a quicksort algorithm. The func1 takes in an array (arr), the first and the last index of the array. We should start sorting from-including and between (low, high respectively). This function basically does partitioning after func2 is called.

func2 takes in the array, the first index, and the last index. “p” acts as the pivot. It then starts comparing the pivot element to the highest element, if the highest element is less than the pivot element it will get placed behind it, and if it is more than it, it will be placed in front of the pivot element. Ultimately it returns the high index.

func1 then sets this returned high value as “pi”. Then it recursively calls itself twice, once by taking in the arguments as low and p1-1 and second by taking in

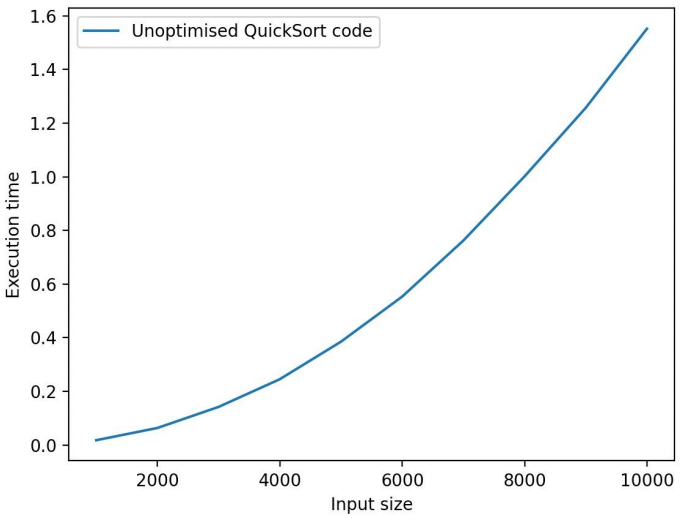
the arguments as pi+1 and high. This way it partitions the main array into two sub-arrays and continues the process of sorting.

Average-case analysis:
As this code is an example of a quick sort algorithm, it takes the array and sorts it by switching values and splitting it into sub-arrays and each subarray then goes on splitting into 2 repeatedly until there is only one element in each array. So if we take n as the number of elements in the array the average-case complexity analysis will be O(nlogn). Although there is a possibility that the pivot is the largest or the smallest element. In that case, the complexity will be closer to O(n^2).

2. Test the code on all the inputs at: https://raw.githubusercontent.com/ldklab/ensf338w23/main/assignments/assignment2/ex2.json.y Plot timing results. Provide your timing/plotting code as ex2.2.py.(2pts)

```
import sys
sys.setrecursionlimit(20000)
import json
import timeit
import matplotlib.pyplot as plt

f = open('q2.json')
dict = json.load(f)

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 f_array (i):
    low = 0
    arr = dict[i]
    high = len(arr) - 1
    func1(arr, low, high)

unoptimized_arr = []
x_axis=[]

for i in range(len(dict)):
    elapsed_time_uo = timeit.timeit(lambda:f_array(i),number=1)
    unoptimized_arr.append(elapsed_time_uo)
    x_axis.append(len(dict[i]))


plt.plot(x_axis,unoptimized_arr, label = "Unoptimised QuickSort code" )


plt.legend()

plt.xlabel('Input size')
plt.ylabel('Execution time')
plt.show()
```



3.Compare the timing results with the result of the complexity analysis. Is the result consistent? Why?(2pts)

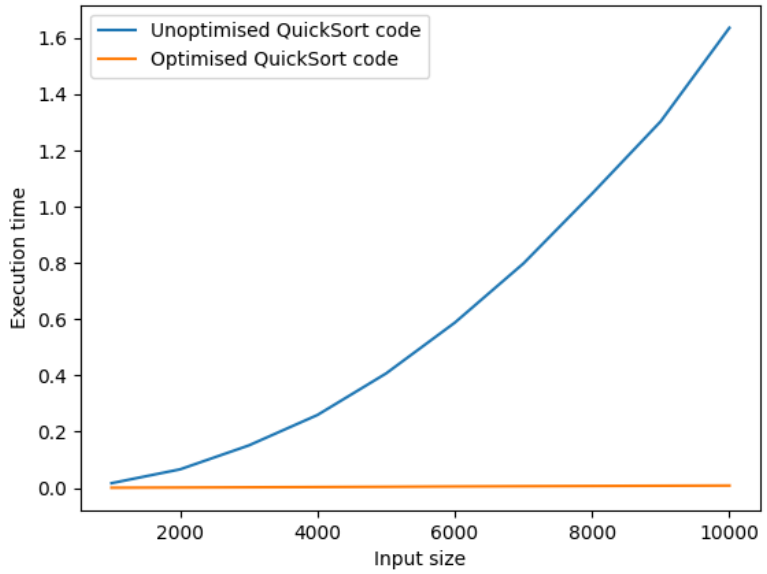
Earlier we state two situations in the complexity analysis best being O(nlogn) and the worst case being O(n^2). The plot above shows an exponential graph corresponding to O(n^2)- the worst case. This is because the chosen pivot is wrong (array[start]). If we take the pivot as the average then we will get a complexity analysis corresponding to O(nlogn), which will be consistent.

4. Change the code – if possible – to improve its performance on the input given in point 2. If possible, provide your code as ex2.4.py and plot the improved results. If not possible, explain why. (2 pts)

```
import sys
sys.setrecursionlimit(20000)
import json
import timeit
import matplotlib.pyplot as plt

f = open('q2.json')
dict = json.load(f)

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 f_array (i):
    low = 0
    arr = dict[i]
    high = len(arr) - 1
```

```
    func1(arr, low, high)

#Optimized code

def func1_opt(arr, low, high):
    if low < high:
        pi = func2_opt(arr, low, high)
        func1_opt(arr, low, pi-1)
        func1_opt(arr, pi + 1, high)

def func2_opt(array, start, end):

    p = array[(start+end)//2] #pviot is median

    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 f_array_opt (i):
    low = 0
    arr = dict[i]
    high = len(arr) - 1
    func1_opt(arr, low, high)

unoptimized_arr = []
optimized_arr = []
x_axis=[]

for i in range(len(dict)):
    elapsed_time_uo = timeit.timeit(lambda:f_array(i),number=1)

    print(f"Execution time for un optimised is {elapsed_time_uo} seconds")
    unoptimized_arr.append(elapsed_time_uo)

    elapsed_time_o = timeit.timeit(lambda:f_array_opt(i),number=1)
    print(f"Execution time for optimized is {elapsed_time_o} seconds")
    optimized_arr.append(elapsed_time_o)
```

```
    x_axis.append(len(dict[i]))

plt.plot(x_axis, unoptimized_arr, label = "Unoptimised QuickSort code" )
plt.plot(x_axis, optimized_arr, label = "Optimised QuickSort code" )

plt.legend()
plt.xlabel('Input size')
plt.ylabel('Execution time')
plt.show()
```

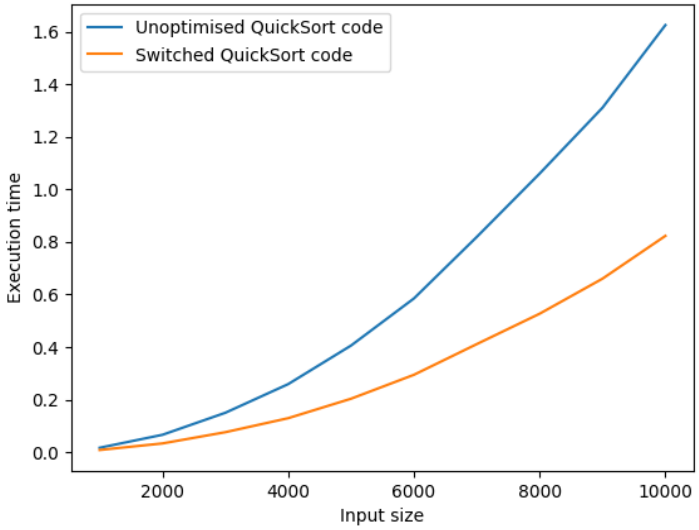


5. Alter the inputs given in point 2 – if possible - to improve the performance of the code given in the text of the question. The new inputs should contain all the elements of the old inputs, and nothing more. Plot the results and provide the new inputs as ex2.5.json). If not possible, explain why. (2 pts)

```
import sys
sys.setrecursionlimit(20000)
import json
import timeit
import matplotlib.pyplot as plt

f = open('q2.json')
dict = json.load(f)

f_switched = open('swiched_data.json')
dict_switched= json.load(f_switched)
```

```
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 f_array (i):
    low = 0
    arr = dict[i]
    high = len(arr) - 1
    func1(arr, low, high)

# Switched code

def func1_switched(arr, low, high):
    if low < high:
        pi = func2_switched(arr, low, high)
        func1_switched(arr, low, pi-1)
        func1_switched(arr, pi + 1, high)


def func2_switched(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 f_array_switched (i):
    low = 0
    arr = dict_switched[i]
    high = len(arr) - 1
    func1_switched(arr, low, high)

unoptimized_arr = []
switched_arr = []
x_axis=[]

for i in range(len(dict)):
    elapsed_time_uo = timeit.timeit(lambda:f_array(i),number=1)


    #print(f"Execution time for un optimised is {elapsed_time_uo} seconds")
    unoptimized_arr.append(elapsed_time_uo)

    elapsed_time_switched = timeit.timeit(lambda:f_array_switched(i),number=1)
    #print(f"Execution time for optimized is {elapsed_time_o} seconds")
    switched_arr.append(elapsed_time_switched)

    x_axis.append(len(dict[i]))

plt.plot(x_axis, unoptimized_arr, label = "Unoptimised QuickSort code" )
plt.plot(x_axis, switched_arr, label = "Switched QuickSort code" )

plt.legend()
plt.xlabel('Input size')
plt.ylabel('Execution time')
plt.show()
```



EXERCISE 3

In the Lecture, we have discussed multiple search algorithms, building up to what is known as the Interpolation Search. Interpolation search is a variant/enhancement on the Binary search for multiple reasons, and is implemented usually as:

1. What are some of the key aspects that makes Interpolation search better than Binary search (mention at least 2) (2 pts)

Interpolation search uses a mathematical formula to make a better prediction as to where the value might be located. It makes better use of the information provided about the spacing between every element and the first and last values. Binary search on the other hand would take more time to find the value since it will keep on breaking the array into half repeatedly until it finds the value, not making better use of the data provided,

2. An underlying assumption of the interpolation search, is that sorted data are uniformly distributed. What happens if the data follows a different distribution (something like normal)? Will the performance be affected? Discuss why (whether yes or no) (3 pts)

The formula used in interpolation makes the estimation of the value assuming the array is sorted and uniformly distributed. If it was not, the assumption would be inaccurate and further away from the actual value. If we were to take the normal distribution as an example, there could be a number of the same values together and the outlier values would throw off the estimation, which will result in the assumption being unreliable and it would take more time for the search algorithm to find the values

Hence, the performance of the interpolation search can definitely be affected based on the type of array imputed.

3. If we want to modify the Interpolation Search to follow a different distribution, which part of the code will be affected? (2 pts)

The line of code that calculates the variable "pos" would need to be modified to use the new distribution. Specifically line 5 which is

```
pos = low + int(((float (high - low) / (arr[high] - arr [low])) * (x - arr [low])))
```

4. When comparing: linear, Binary, and Interpolation Sort

a. When is Linear Search your only option for searching your data as Binary and Interpolation might fail? (1 pt)

If the data is not sorted, the binary and interpolation search methods assume that it is sorted. As a result, these methods are ineffective and inefficient when dealing with unsorted data.

b. What is a case that Linear search will outperform both Interpolation and Binary search, and Why? Is there a way to improve Binary and Interpolation Sort to resolve this issue? (2 pts)

In situations, where data exhibits an irregular distribution or contains a significant number of duplicates, using linear search can outperform both Interpolation and Binary search. This is because linear search evaluates each element in the dataset sequentially, making it ideal for unordered and unstructured data. However, using a hybrid algorithm that combines sorting and searching techniques can potentially improve the performance of binary and interpolation searches when dealing with such datasets.

EXERCISE 4

In the lecture recordings, we discussed some of the main differences between arrays (or lists in python) and Linked Lists.

1. Compare advantages and disadvantages of arrays vs linked list (complexity of task completion) (2 pts)

Advantages of arrays include easy usage, quick retrievals and replacements if the index is known, and efficient access and use of memory space compared to linked lists. On the other hand, arrays have disadvantages such as complications with deletion, insertion, and resizing, and a lack of resizability compared to linked lists.

Linked lists, on the other hand, are dynamically allocated and can easily delete and insert values, with reduced memory wastage due to their contiguous nature. However, linked lists are not indexable, so accessing a specific position requires

traversing the entire list, making them slower and less efficient in terms of access compared to arrays.

2. For arrays, we are interested in implementing a replace function that acts as a deletion followed by insertion. How can this function be implemented to minimise the impact of each of the standalone tasks? (3 pts)

To minimize the impact of the standalone tasks in a replace function for arrays, the function can be implemented by adjusting the size of the array, shifting elements as necessary, inserting the new element, and updating the size of the array to reflect the change. This approach optimizes performance by only dynamically changing the size of the array when it is necessary.

3. Assuming you are tasked to implement a Singly Linked List with a sort function, given the list of sort functions below, state the feasibility of using each one of them and elaborate why is it possible or not to use them. Also show the expected complexity for each and how it differs from applying it to a regular array: (1 pt each)

a. Selection sort

Selection sort is feasible to use for sorting a singly linked list, but it is not the most efficient sorting algorithm for this data structure. it searches the list entirely to search for a single change at the presence of the first item, then starting again from the beginning, and selection sort requires multiple passes through the entire list, making it inefficient for larger lists.

The time complexity of the selection sort for a singly linked list is O(n^2)

b. Insertion Sort

The insertion sort algorithm is feasible as it can be used to sort a linked list by rearranging the links between the node which is done by iterating through the linked list and compare each node with the previous nodes and rearranging the linked list as needed.

The time complexity here is also O(n^2).

c. Merge Sort

Merge sort is feasible as it divides the list into two halves, sort each half, and then merge the two sorted halves back together.

It has a time complexity of O(n log n) for both arrays and singly linked lists.

d. Bubble sort

Bubble sort is feasible but not an efficient choice for sorting a singly linked list, bubble sorting is repeatedly swap adjacent elements that are in the wrong order

until the list is sorted. The time complexity of bubble sort being O(n^2) makes it inefficient for larger lists

e. Quick Sort

Bubble sort is feasible but not an efficient choice for sorting a singly linked list, bubble sorting is repeatedly swap adjacent elements that are in the wrong order until the list is sorted. The time complexity of bubble sort being O(n^2) makes it inefficient for larger lists.

EXERCISE 5

Stacks and Queues are a special form of linked lists with some modifications that makes operation better. For parts 1 and 2 assume we are using a singly linked list

1. In stacks, insertion (push) adds the newly inserted data at head

a. why? (0.5 pts)

Stacks work on a "last in, first out" principle, which means the last item added to the stack should be the first one removed. By adding new items to the beginning of the linked list (i.e., at the "head"), we can quickly add and remove items from the stack without having to search through the entire list each time.

b. Can we insert data at the end of the linked list? (0.5 pts)

Yes, we can insert data at the end of the linked list, but it would not be efficient for a stack since we need to access the top of the stack, which is the head. Inserting at the end would require traversing the entire linked list, which would result in slower operation time.

c. If yes, then what is the difference in operation time (if any) for pushing and popping data from the stack? (1 pt)

In a stack that uses a singly linked list, adding and removing data have a constant time complexity of O(1), which implies that both operations execute at the same speed.

2. In Queues, we added a new pointer that points to the tail of the linked list

a. why? (0.5 pts)

Queues follow a “first in, first out” principle which is having new pointer that points to the tail of the linked list. which makes the enqueuing operation more efficient since it allows for efficient insertion of new nodes at the tail end of the queue, and removal of nodes from the head end of the queue.

b. Can we implement the Queue without the tail? (0.5 pts)

Yes we can implement a queue without a tail but doing so would make the enqueuing operation less efficient and slow. One would have to traverse through the linked list from the head to the tail every time we need to enqueue a new node.

c. If yes, then what is the difference in operation time (if any) for enqueuing and dequeuing data from the stack? (1 pt)

There would be a difference in operation time. O(n) (n being the number of nodes) is the operation for implementing the queue without a tail pointer, whereas it is O(1) operation for dequeuing data

d. Can we change the behaviour of the enqueue and dequeue where we enqueue at head and dequeue at tai? Do you think it is a good idea? (1 pt)

It is possible to do the said behaviour. However, it is not recommended as it violates the “first in, first out’ principle of the queue which comes with confusion and errors.

3. Revisit your answers for part 1 and 2 but now with the assumption that we are using circular doubly linked list (5 pts)

In a circular doubly linked list, the head and tail pointers would point to each other, and each node would have a next and previous pointer.

Stacks: In order to implement the last in first out principle of a stack with a circular doubly linked list, new data is always added as the head node. This ensures that the head node always points to the latest data added. It is not possible to add data at the end of the list in a stack, as this would go against the intended last in first out principle.

Queue: To follow the first in first out principle in queues using a circular doubly linked list, we put a new pointer to point to the tail of the linked list. However, if we don't have a tail pointer, we can still implement the queue by using the head and reference pointers to move through the circular doubly linked list. Both the enqueue and dequeue operations have a time complexity of O(1). It's not recommended to change the behaviour to a stack-like last in first out principle, as this goes against the intended behaviour of a queue.