Course : ENSF 338

Assignment # : 2

Student Names : Magdy Hafez, Nimna Wijedasa

UNIVERSITY OF CALGARY

Github

https://github.com/notnimna4761/ENSF_338/tree/main/Assignment%202

Exercise 1

In the lab, we have discussed how **memoization**[1] can be used to improve the performance of an algorithm.

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

   Memoization 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. 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)**

   It recursively calls itself and the function func takes an integer n as an argument and returns the n-th Fibonacci number.

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

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

   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)

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

```
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
```

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

   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 with the size of the input n. So the complexity is O(n) as only 1 function is called as opposed to the un-optimized function of complexity O(2^n) where two functions are called.

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)**

```
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

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

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

Look at the following code:

```
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
```

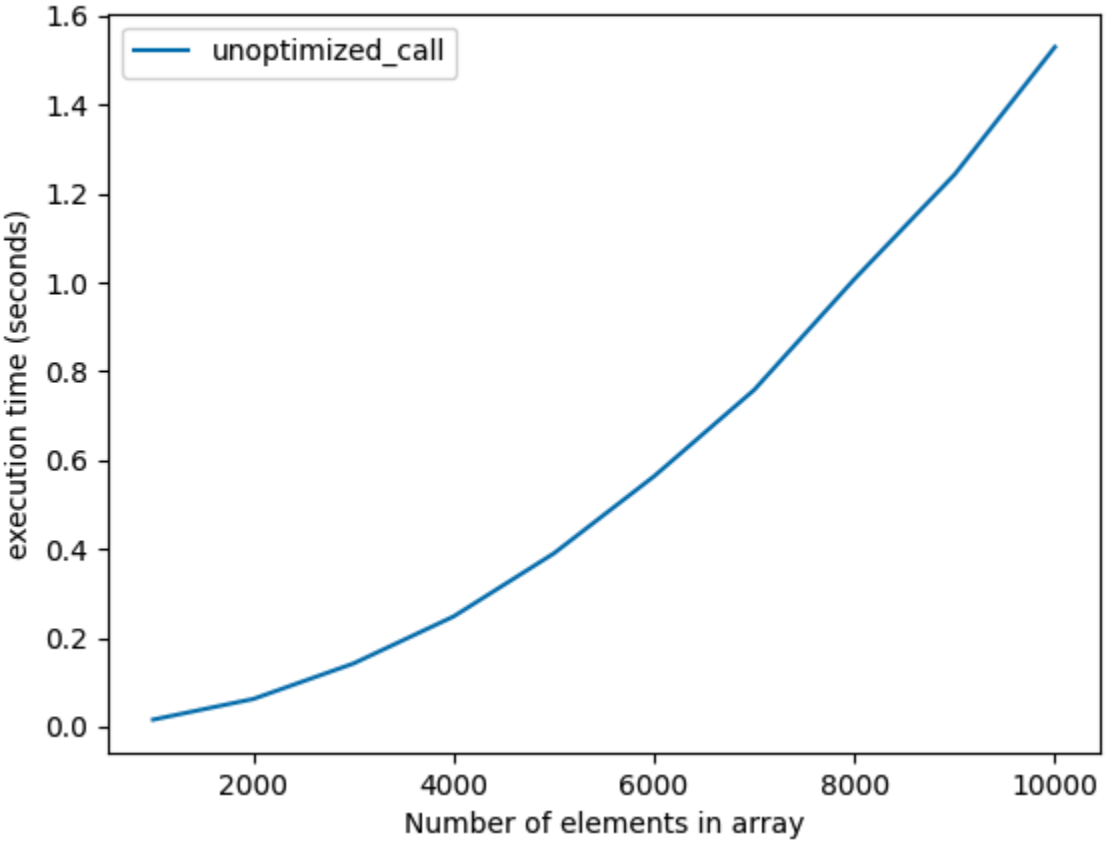
1. Explain what the code does and perform an average-case complexity analysis. Describe the process, not just the result. **(2 pts)**

   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).

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

```
import json
import matplotlib.pyplot as plt
import sys
import timeit

sys.setrecursionlimit(20000)
def un_optimized_sort(arr, low, high):
    if low < high:
        pi = unoptimized_quicksort(arr, low, high)
        un_optimized_sort(arr, low, pi-1)
        un_optimized_sort(arr, pi + 1, high)

def unoptimized_quicksort(array, start, end):
    p = array[start] #p 10
    low = start + 1 # low = 1
    high = end      # high = 5
    while True:
        while low <= high and array[high] >= p:     # 5 >= 10
            high = high - 1
        while low <= high and array[low] <= p:      #  <= 10
            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

with open("q2.json", "r") as inF:
    arr = json.load(inF)

def unoptimized_call(i):
    unoptimized = un_optimized_sort(arr[i], 0, len(arr[i])-1)

unoptimized_array_y = []
array_x = []

for i in range(len(arr)):
    unoptimized_time = timeit.timeit(lambda: unoptimized_call(i), number=1)
    unoptimized_array_y.append(unoptimized_time)
    array_x.append(len(arr[i]))

# plotting the points
plt.plot(array_x, unoptimized_array_y, label = "unoptimized_call")
```

```
# naming the x axis
plt.xlabel('Number of elements in array')
# naming the y axis
plt.ylabel('execution time (seconds)')
# adding a legend
plt.legend()

# function to show the plot
plt.show()
```



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

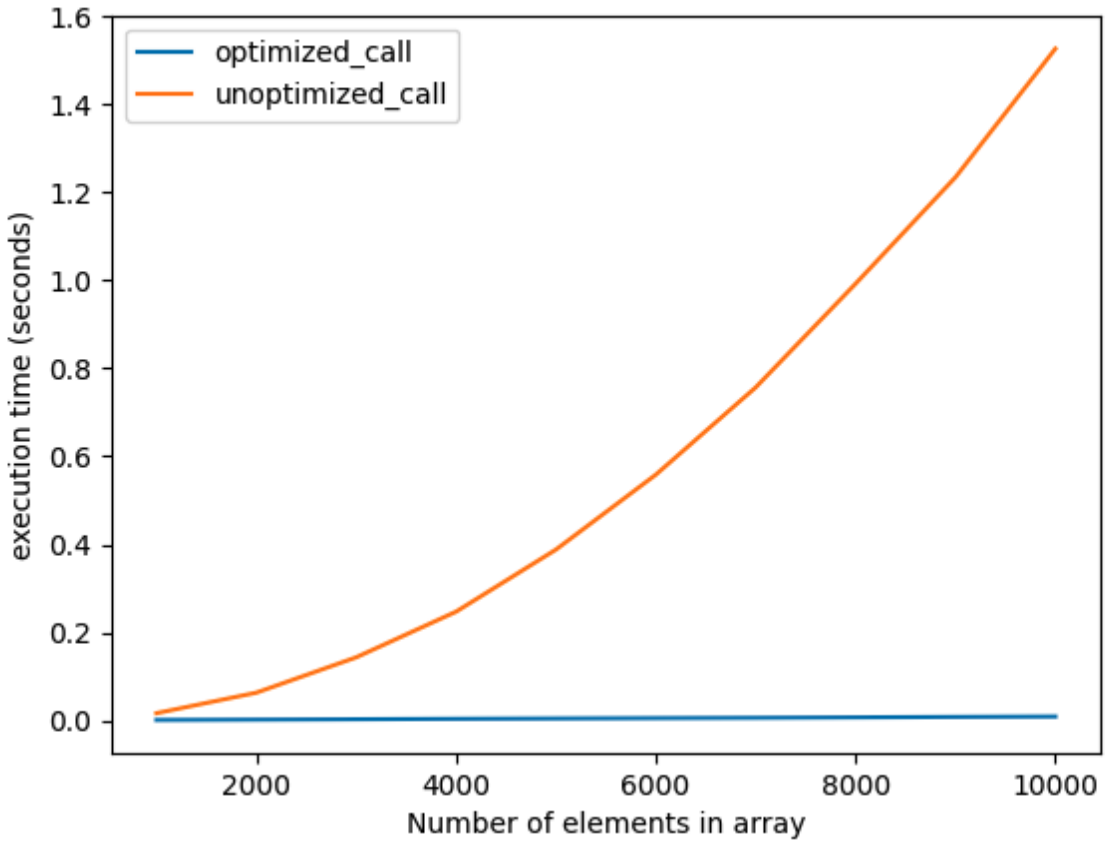
The average time complexity of the function as described above is O(n log n) but when plotted has a graph of n^2 this is the worst-case scenario. This is a consequence of the wrong pivot being selected namely the array [start]. That is why the plot is not consistent with the complexity analysis. We can achieve a close plot to O (n log n) if we select array [middle] as the pivot instead.

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 json
import matplotlib.pyplot as plt
import sys
import timeit

sys.setrecursionlimit(20000)
def optimized_sort(arr, low, high):
    if low < high:
        pi = optimized_quicksort(arr, low, high)
        optimized_sort(arr, low, pi-1)
        optimized_sort(arr, pi + 1, high)
    return

def optimized_quicksort(array, start, end):
    p = array[(start + end)//2]
    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 + low ) // 2)

def un_optimized_sort(arr, low, high):
    if low < high:
        pi = unoptimized_quicksort(arr, low, high)
        un_optimized_sort(arr, low, pi-1)
        un_optimized_sort(arr, pi + 1, high)

def unoptimized_quicksort(array, start, end):
    p = array[start] #p 10
    low = start + 1 # low = 1
    high = end      # high = 5
    while True:
        while low <= high and array[high] >= p:     # 5 >= 10
            high = high - 1
        while low <= high and array[low] <= p:      #  <= 10
            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

with open("q2.json", "r") as inF:
    arr = json.load(inF)

def optimized_call(i):
    optimized = optimized_sort(arr[i], 0, len(arr[i])-1)

unoptimized_array_y = []
optimized_array_y = []
array_x = []

for i in range(len(arr)):
    optimized_time = timeit.timeit(lambda: optimized_call(i), number=1)
    optimized_array_y.append(optimized_time)
    array_x.append(len(arr[i]))

# plotting the points
plt.plot(array_x, optimized_array_y, label = "optimized_call")



with open("q2.json", "r") as inF:
    arr = json.load(inF)

def unoptimized_call(i):
    unoptimized = un_optimized_sort(arr[i], 0, len(arr[i])-1)

unoptimized_array_y = []
array_x = []

for i in range(len(arr)):
    unoptimized_time = timeit.timeit(lambda: unoptimized_call(i), number=1)
    unoptimized_array_y.append(unoptimized_time)
    array_x.append(len(arr[i]))

# plotting the points
plt.plot(array_x, unoptimized_array_y, label = "unoptimized_call")

# naming the x axis
plt.xlabel('Number of elements in array')
# naming the y axis
plt.ylabel('execution time (seconds)')
# adding a legend
plt.legend()

# function to show the plot
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)**



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:

```
def interpolation_search(arr, x):
    low = 0
    high = len(arr) - 1
    while low <= high and x >= arr[low] and x <= arr[high]:
        pos = low + int(((float(high - low) / ( arr[high] - arr[low])) * ( x - arr[low])))
        if arr[pos] == x:
            return pos
        if arr[pos] < x:
            low = pos + 1
        else:
            high = pos - 1
    return -1
```

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

   Interpolation search works better than binary search for sorted and uniformly distributed arrays.
   Interpolation search estimates the position of the target element based on the values of the elements surrounding it, while binary search always starts by checking the middle element of the list, which makes interpolation more efficient.

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)**

Yes, since the interpolation is based on estimation and it might not be accurate if the data were not uniformly distributed, which will make the search take more time and steps to find the element.Thus, The performance will be degraded and it would be wize to use another sorting algorithm than interpolation.

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 the calculation of the "mid" index gets effected as the distribution changes. The exact formula to calculate the mid will vary according to the distribution.

4. When comparing: linear, Binary, and Interpolation Sort
    1. When is Linear Search your only option for searching your data as Binary and Interpolation might fail? **(1 pt)**

When the entire dataset is unorganized and is not uniformly distributed. Both of which are vital for binary and quick search to perform optimally.

    2. 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)**

When the distribution is highly irregular but ordered either in an ascending order or descending order. This is because when the binary search and interpolation search split the irregularly distributed data the assumed value is off from the correct one.

We can improve binary and interpolation sort by using them both in tandem. Adaptively switching from binary to interpolation when the estimated guess is far off from the actual value we can speed up the sort.

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:

   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.

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 minimize the impact of each of the standalone tasks? **(3 pts)**

```
def shift(array,number_to_be_replaced,number_to_replace,index):
    n = len(array)
    if number_to_be_replaced < number_to_replace:
        while(index != (n-1) and (array[index] > array[index+1])):
                    array[index], array[index+1] = array[index+1], array[index]
                    index += 1

    elif number_to_be_replaced  > number_to_replace:
        while(index != 0 and (array[index] < array[index-1])):
                array[index-1], array[index] = array[index], array[index-1]
                index -= 1
    return array
def replace_func(array,number_to_replace,number_to_be_replaced):
    j = 0
    for i in array:
        if int(i) == number_to_be_replaced :
            array[j] = number_to_replace
            index_num = j
            break
        j += 1
    return shift(array,number_to_be_replaced,number_to_replace,index_num)
```

The question asked for the implementation of a replace function that acts as a deletion followed by insertion, so I made this code with two functions: replace_func and bubble_sort. To minimize the impact of each of the standalone tasks, this replace function only needs to replace the number with the number to be replaced. Which is more efficient than inserting then deleting since I only need to iterate through the loop once. Also, since the array is already sorted, we don't have to sort the array again but the other part of the array, for instance if i have array [1,2,5,8,15,20,22] and i want to replace the 8 with 30, then i don't need to sort whats before 8 since its already sorted, i did this implementation by adding conditional statement (if number to replace (which is 30) > number to be replaced (which is 8)) which will check if the number that i am replacing is bigger than the number that was already there. Thus, if it's bigger, then bubble sort need to be applied on the bigger values since the array was sorted when it was given, and if it's smaller , then the bubble sort will be applied to the smaller

values. Then it will return back to the replace func and the sorted array with the new value.

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)**
    1. Selection sort
    2. Insertion Sort
    3. Merge Sort
    4. Bubble sort
    5. Quick Sort

a)Selection sort:

It is complex to use Selection sort algorithm for linked list since it needs to swap elements and linked list's data are stored in fragments where pointers point to these fragments not contiguous memory which makes it more complex to just swap two values. Expected complexity is O(n^2). On the other hand, if it was an array, not a linked list, using the Selection sorting algorithm makes more sense and easier to implement since the arrays are stored in a contiguous memory and it takes constant time to swap two elements. So Selection sort is not recommended for linked list and the algorithm is not feasible for a very large number of data set

b) insertion:

Insertion sorting algorithm can be used in linked lists as it is not hard to implement. The insert algorithm only needs to make sure to insert it in the appropriate position for each element and it will sort the linked list. Expected Complexity is O(n^2). It is easy to implement insertion sort algorithms in arrays as well since it makes sure to insert it in the right index. The only difference between linked list and arrays using insertion algorithm is that linked list uses pointers to sort while inserting and inserting the right data in the right areas for arrays. Algorithm is not feasible for very large number of data sets

c) Merge Sort:

Merge sort algorithm is possible to implement for linked list since its based on divide and conquer principle which isn't too difficult to implement since it can divide the list to half and merge the sorted half. Expected complexity is O(n log n). Moreover, regarding arrays, it is also easy to apply it since it only divides and sorts the array, the only difference between both arrays and linked list is that linked list need to change and rearrange pointers and arrays only need to swap

elements together.This algorithm is feasible for a very large number of data set and this is likely due to its time complexity.

d) Bubble sort:

Bubble sort is easy to implement for linked lists since data doesn't have to be contagious in order to use this algorithm. It works by a continuous comparison between the neighbor elements and swapping the pointers together if they are in the wrong order. Expected complexity is O(n^2). It is also easy to implement for arrays since it only needs to swap values. The algorithm is not feasible for a very large number of data sets.

e) Quick Sort:

Quick sort might not work well with linked lists since quick lists need to swap elements and it is difficult to implement in linked lists since the linked lists are only fragments of memory that are pointed at using pointers. Expected Complexity O(n log n), it is easier to use a quick sort algorithm on arrays instead since it is based on contiguous memory that can be swapped which is more consistent than a linked list. Algorithms are not feasible for a very large number of data sets.

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
    1. why? **(0.5 pts)**

        Well, stack is a type of linked list where the nodes are pushed at the top since it follows the last in first out principle . Elements are pushed at the head of the list to ensure that the list follows the stack principle, not to mention, it will make it easier to pop elements since you can pop the one from the head straightaway.

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

        yes, but it will be more complicated to make it work as a stack and pushing and popping from the bottom since it will be less convenient and more unnecessary operations, which can affect the time complexity and the efficiency of the code. That can be easily avoided by pushing data from the top and popping from the top.

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

        if the stack gets pushed and popped at the head of the stack it will be O(1), but if it's at the tail it will be O(n).

2. In Queues, we added a new pointer that points to the tail of the linked list
    1. why? **(0.5 pts)**

        Since the queue follows the first in first out principle, it is required to have a tail that points at the end of the linked list, which enqueues the new data into the linked list while the header dequeues the data.

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

Yes, the queue can be implemented without the tail pointer. However, it is not recommended since we will have to go through the list from the head until we reach to the end of the list everytime we need to add data to the list

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

   Dequeuing would take O(1) but enqueueing would take O(n) instead of O(1)

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

   yes but it's not a good idea, since it will make insertion from constant time operation to linear time instead, since removing from the tail demands a pointer to the predecessor to the current tail which takes as many times as the number of elements in the list.

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

a) Why?

   The nodes are pushed at the top since it follows the last in first out principle . Elements are pushed at the head of the list to ensure that the list follows the stack principle, not to mention, it will make it easier to pop elements since you can pop the one from the head straightaway.

b) Can we insert data at the end of the linked list?

   yes, but it will be more complicated to make it work as a stack and pushing and popping from the bottom since it will be less convenient and more unnecessary operations, which can affect the time complexity and the efficiency of the code. That can be easily avoided by pushing data from the top and popping from the top.

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

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

a) Why?

Since the queue follows the first in first out principle, it is required to have a tail that points at the end of the linked list, which enqueues the new data into the linked list while the header dequeues the data.

b) Can we implement the Queue without the tail?

   Yes, the queue can be implemented without the tail pointer. Since we can go back and forth through the doubly linked list.

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

   there will be no difference in terms of time complexity.

d) Can we change the behavior of the enqueue and dequeue where we enqueue at head and dequeue at tail? Do you think it is a good idea?

   we can do that and it will not make a difference since we can go back and forth through the queue but it’s better to follow the convention of enqueuing at the tail and dequeuing at the head.