ENSF 338 Assignment 2

Carson May and Caiden Affleck
(30139961, 30145805)

Github Repo

Task Distribution

Member	Tasks Completed	Total Work Done
Carson	Q2, Q3, 1/3 of Q4. Editing and debugging Q1, Q4, and Q5	50%
Caiden	Q1, Q5, and 2/3 of Q4. Editing and debugging Q2 and Q3	50%

Question 1

Questions:

1. Explain, in general terms and your own words, what memoization is (0.5 pts)
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)
4. Is this an example of a divide-and-conquer algorithm? Explain. (0.5 pts)
5. What is its time complexity? (0.5 pts)
6. Implement a version of the code above that use memoization to improve performance. Provide this as ex1.3.py. (2 pts)
7. Perform an analysis of your optimized code: what is its computational complexity? (3 pts)
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)
9. Discuss the plot and compare them to your complexity analysis. (1 pt) 1 Yep, not a typo, we are not talking about "memorization".

Part 1:

Memoization is a process where values from a computation are stored in a 'cache' and then retrived from that same 'cache' the next time the same computation is done.

Part 2/3:

The code is to calculate the fibonacci number at index 'n', this function works by recursively adding together n-1 and n-2 until the base cases are reached (0 or 1) then all of the numbers are added back together.

Part 4:

Yes, the function breaks down the number (n) until the base case (0 or 1) is reached. At this point the function will then add together all of the sub-solutions to get the fibonacci number at index 'n'.

Part 5:

Time complexity is T(2^N)

Part 6:

ex1.3.py is in the Q1 folder

Part 7:

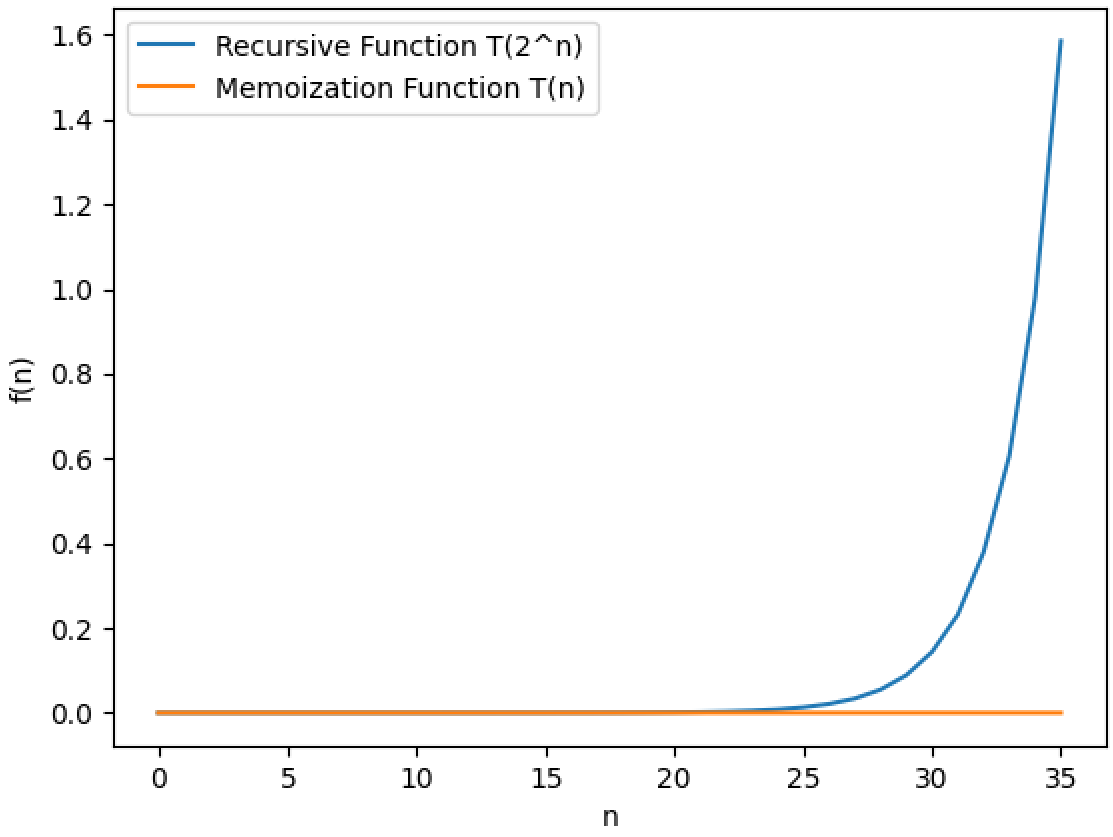
Looking at the previous computational complexity of the pure recursive function, it had O(2^n), which is due to the amount of function calls that occur.

Since we are using memoization there is going to be less function calls. The data structure that we use is a dictionary, which fills up linearly through the function (up from 2 to n). Once the dictionary is full, there is no more need for recursive function operations, and the only operations are lookups in the dictionary as well as logic. which means there is a computational complexity of O(n).

Part 8:

Added ex1.5.py to repo

Plot:



Part 9:

The plot shows two different graphs. Both graphs seem almost constant to begin with, once the graphs reach ~n=20 mark they begin to seperate. The Recursive function begins to grow exponentially (affirming T(2^n)), while the Memoization function remains almost constant (affirming T(n)). This is accurate with the analysis that was done in question 7.

Question 2

Code:

```python
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
```

Questions:

1. Explain what the code does and perform an average-case complexity analysis. Describe the process, not just the result. (2 pts)
2. Test the code on all the inputs at: https://raw.githubusercontent.com/ldklab/ensf338w23/main/assignme nts/assignment2/ex2.json Plot timing results. Provide your timing/plotting code as ex2.2.py. (2 pts)
3. Compare the timing results with the result of the complexity analysis. Is the result consistent? Why? (2 pts)
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)
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)

Answers:

1.

This code uses the quick sort method to sort arrays. Quick sort works by recursively subdividing the array and ordering each subdivision. The arrays are sorted around the pivot point, which in this case is the first element of the array. The subdivision will continue until the length of all sub-arrays have a single element.

Average Case Complexity Analysis:

Step 1 It is necessary to establish a hypothetical average partition, that is one that is not always balanced. In this case, assume that the partitioning will never be worse than a 3-to-1 split, so for every partition, one subdivision will receive n/4 elements and the other will get 3n/4.

Step 2 Now, the total partioning time for each subdivision will never exceed cn. (For example, one side will compare n/4 elements, the other will compare 3n/4 elements, resulting in n compared elements.)

Step 3 The node of the partition which receives n/4 elements each time will have log_4(n) layers. This is the shortest path to a subdivision of size one, and so every layer beyond this layer will have a partitioning time < cn.

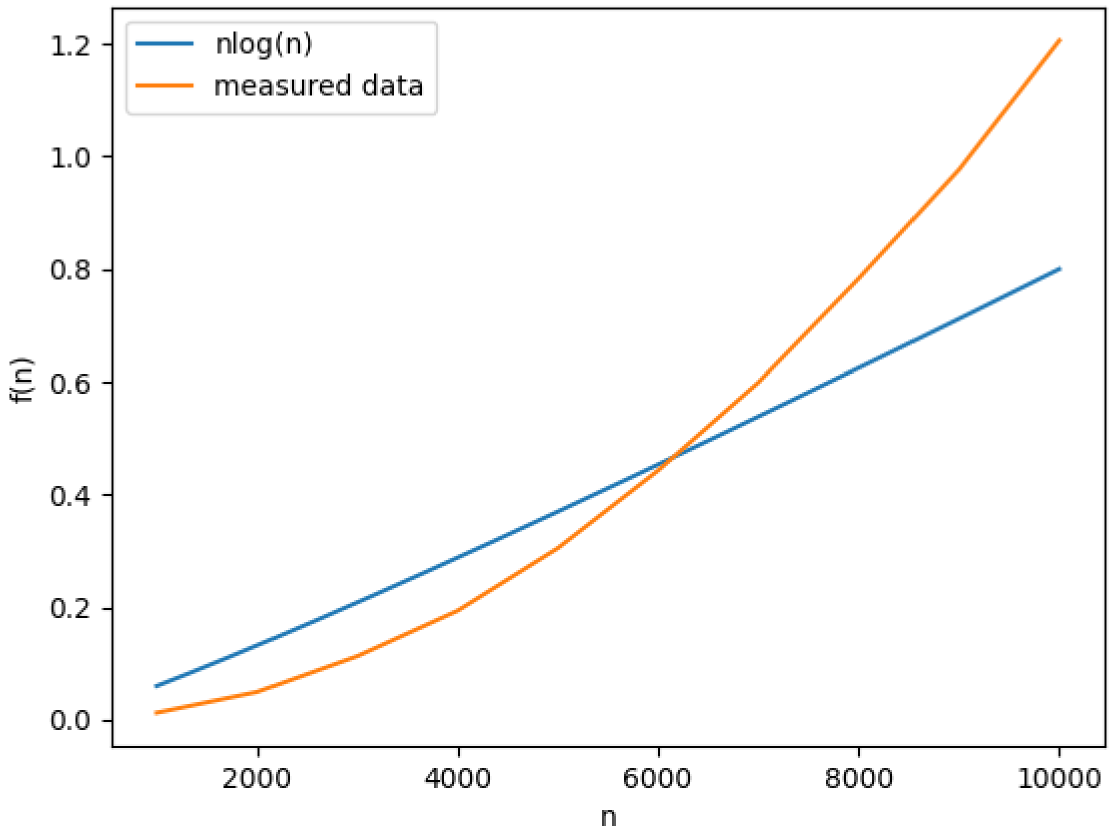
Step 4 It is now necessary to calculate the layers of the longest path to a subdivision size of one. This will be the path that receives 3n/4 elements each partition. In this case, we will reach log_4/3(n) levels.

Step 5 Now, combining the partitioning time of each level with the total number of levels, we have O(nlog_3/4(n)). We can use the fact that log_4/3(n) = log_10(n) / log_10(4/3). Furthermore, since log_10(4/3) is a constant, we can disregard it, so we are left with O(nlog(n)) as an average case complexity.

2.

ex2.2.py is located within the github repo

Plot:



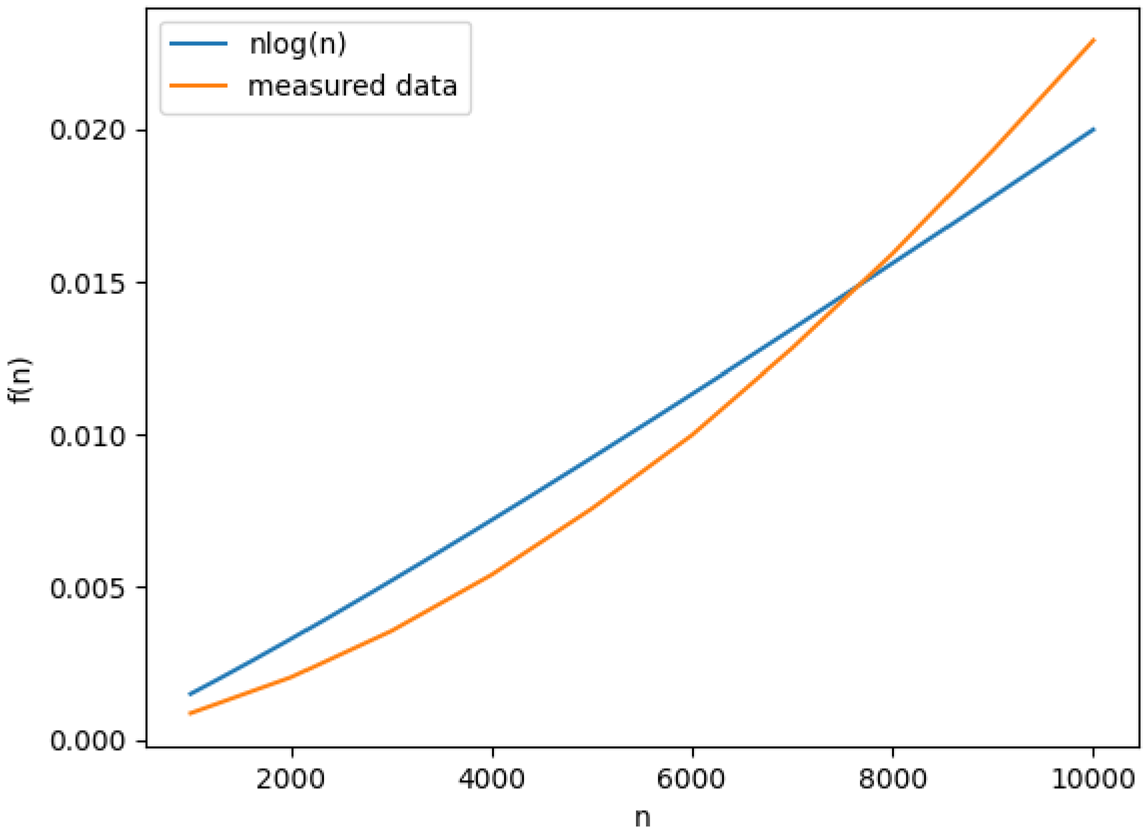
3.

From the plot, it is evident that the results are worse than the average-case complexity for large n. This is likely due to an ineffecient pivot point based on how the input data is formatted. The input data is already almost sorted, and therefore having a pivot point of the first element in the array will always result in a lopsided partition.

4.

ex2.4.py is located within the github repo The code was changed to set the pivot point to the halfway point of the array, as the given input data already seemed roughly sorted, and the quicksort algorithm is most efficient when there is an even partition between the subarrays, therefore a halfway pivot point would produce a closer-to-even partition as opposed to a start pivot point.

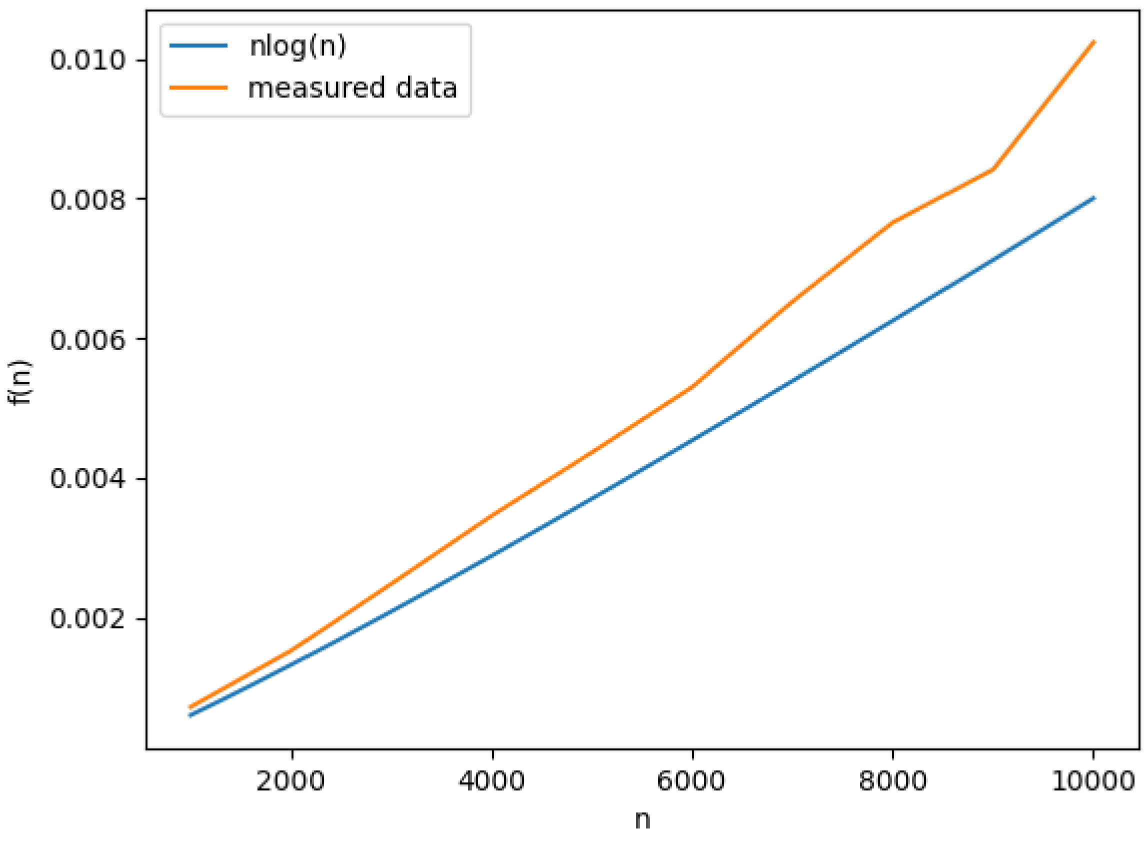
Plot:



5.

ex2.5.json is located within the repo. The data was altered by shuffling all of the elements in the input data. This results in a completely unordered set of data, as opposed to an almost ordered set. This unordered data will work more efficiently with the chosen pivot point.

Plot:


nlog(n)
measured data
0.010
0.008
0.006
0.004
0.002
f(n)
2000
4000
6000
8000
10000
n

Question 3: Interpolation Search

Questions:

What are some of the key aspects that makes Interpolation search better than Binary search?

```
Interpolation search is quicker than binary search, as binary search will always
search a set of data by accessing the midpoint of it, while interpolation search
is dynamic and can change this access point based on the value being searched.
Interpolation also has a better average time complexity of O(log(log(n))),
compared to binary search with an average time complexity of O(log(n))
```

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)

```
If a dataset's distribution differs from a uniform distribution, the performance
of interpolation search will be negatively affected. A non-uniform distribution
may result in inaccurate estimates of the position of target values, and
consequently, will require more steps to find the values. With a normal
distribution, the extent to which performance is degraded will likely depend on
the skew of the data. If there's no large skew, the performance may be hindered
slightly, but it will still perform well. In cases of highly-skewed data, the
impact on performance will be much greater as the estimates of the position of
target values will be much less precise.
```

If we want to modify the Interpolation Search to follow a different distribution, Which part of the code will be affected?

```
In the given implementation of interpolation search, the calculation of pos would
be affected. Ideally, one would take into account the cumulative distribution
function (cdf) of a distribution and use that information to find a more precise
range where the target value would lie.
```

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?

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?

```
a) This case occurs when the data given is not sorted in any meaningful way and
does not follow any distribution. In this case, interpolation and binary sort will
not be able to predict the position of target values; therefore, they cannot be
used.

b) A case where this situation may occur is in small sets of data. In these cases,
binary and interpolation search are computationally taxing, as their complexity is
not necessary for such small data sets. The simplicity of linear search is
favourable. The only way to improve the performance of interpolation and binary
search in these cases is to reduce the number of operations done by them. In
already optimized binary and interpolation search algorithms, reducing the number
of operations done is likely not possible while maintaining the functionality of
the algorithms.
```

Question 4: Linked lists & Arrays

Questions:

Compare advantages and disadvantages of arrays vs linked list (complexity of task completion)

```
A large advantage of linked lists is resizability. In many languages, the sizes of
arrays cannot change; however, this is not true in python. Arrays in python are
dynamic, but are inefficient when memory relocation is necessary to increase the
size. Linked lists do not hold this burden; consequently, they are optimal when
the desired size of an array is unknown. Insertion and deletion are also very easy
when dealing with linked lists, with time complexities of O(1), as only the
pointers of the nodes need to be adjusted. Conversely, insertion and deletion in
arrays is computationally expensive, as all of the elements have to be shifted,
resulting in an O(n) complexity.
Arrays are advantageous when the desired size of of an array is known and provide
advantages such as quick access with indexing. Accessing an element in an array
will have a time complexity of O(1), while linked lists will have to sift through
each node, resulting in an O(n) complexity.
```

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

```
The replace function (assuming a value to find and replace, and the value to
replace with) could do both the deletion and the insertion in the same line of
code. This is done by looping through each element of the array until we find
the desired value. Once the value is found we can use the iterator variable to
hold our index, then in one line of code we can set the value of the index at the
array equal to the value to replace with. Since python has a garbage collector,
the value that we replaced with will get deleted. This minimizes the impact of
each standalone task efficiently, as the most taxing component of each task is
shifting each element of the array following the insertion/deletion. By simply
changing the value of the array at the target index, this shifting won't occur.
```

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. Selection Sort:

```
1. It is possible to use a selection sort in a singly linked list, with a
complexity of O(n^2). The difference between applying it to a regular array is
that with a regular array the space complexity is O(n) due to an extra copy of the
array being made during the sort (if the array is not dynamic). While a selection
sort with a linked list has O(1) space complexity, due to the sorting being done
in place, and only needing a constant amount of memory.
```

2. Insertion Sort:

```
2. It is possible to use an insertion sort in a singly linked list, with a
complexity of O(n^2). The complexities are the same as a python array, but a
static array would result in a space complexity of O(n) (whereas the linked list
is O(1)).
```

3. Bubble Sort:

```
3. It is possible to use an bubble sort for a singly linked list, with a
complexity of O(n^2). The complexities are the same as a python array, but a
static array would result in a space complexity of O(n).
```

NOTE:

```
The previous three sorts are feasible with small linked lists, but if the linked
list is with a lot of data it is more feasible to use a sort of O(nlog(n))
complexity (merge and quick sorts).
```

4. Merge Sort:

```
4. It is possible to use a merge sort on a singly linked list, with a time
complexity of O(nlog(n)) and a space complexity of O(n). This sort is feasible to
use for a larger linked list, as it will perform more effiecent than any of the
O(n^2) sorting algorithims. A python array will have the same space complexity and
time complexity as a singly linked list.
```

5. Quick Sort:

```
5. It is possible to use a quick sort on a singly linked list, although it will
look different than if it was applied to an array. This is due to a singly linked
list lacking the power to find a value at a given index, and instead must traverse
```

```
through itself to find the value. The pivot element must be also be chosen
differently, as it cannot be selected by simply accessing an index in the linked
list (as previously mentioned). The complexity is the same as an array, where
O(n^2) for worst case complexity and O(nlog(n)) on average. This sort is feasible
for larger linked lists as it will hold an O(nlog(n)) time complexity, performing
better than O(n^2) sorting algorithims.
```

Question 5

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) b. Can we insert data at the end of the linked list? (0.5 pts) c. If yes, then what is the difference in operation time (if any) for pushing and popping data from the stack? (1 pt)
2. In Queues, we added a new pointer that points to the tail of the linked list a. why? (0.5 pts) b. Can we implement the Queue without the tail? (0.5 pts) c. If yes, then what is the difference in operation time (if any) for enqueuing and dequeuing data from the stack? (1 pt) 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? (1 pt)
3. Revisit your answers for part 1 and 2 but now with the assumption that we are using circular doubly linked list (5 pts)

1.

1. To save time and code. Since the 'head' node is already accessibly it is much easier to insert data at the head rather than traverse through the whole list to insert data at the end. Stacks also inherit "last-in first-out" for applications of them, so being pushed in at the head makes this true.
2. Yes, to insert data at the end of a (singly) linked list we would need to traverse to the final node in the list and then link the new data.
3. It would be longer to push and pop data from the end of a stack, since traversing the list is needed O(n). The pointer to the start of the list already exists and makes it much quicker to push and pop data from the list O(1), whereas the end of the list requires a traversal.

2.

1. A pointer to the end of the linked list exists so that the end of the linked list can easily be found, not requiring a traversal of the list. It is not needed, but without it the list would have to be iterated through till the end was found everytime a new item was added to the queue.
2. Yes, but the queue will take longer to enqueue data, as a full iteration of the queue is needed to get to where new data is to be enqueued.
3. Since operation with the tail pointer is one operation to find the end of the queue, the complexity is O(1). We would go from O(1) to traversing the queue until we find the tail item, which means n amount of iterations resulting in O(n) complexity.
4. Yes, it is possible to do so; however, it would be inefficient. Deqeueing at the tail would make it necessary to first traverse the linked list to find the element that is linked to the tail, and then relink the tail to this node. This almost makes the tail obsolete. Enqeueing at the tail is much more efficient, as no traversal is required. Ultimately, reversing the roles of head and tail is not a good idea.

3.

1. (circular doubly linked list)

1. To keep the "last-in first-out" nature of stacks true.

2. Yes, since the list is circular, the head and tail are connected resulting in a very quick way to find the end of the list.
3. Since we no longer have to traverse the list and just follow the head's pointer to the tail node, the time is now constant, which means there is no difference in time. The issue with doing this is not keeping the nature of a stack being "last-in first-out".

2. (circular doubly linked list)

1. By definition, the pointer already exists in a circular doubly linked list.
2. A circular doubly linked list already references its tail from its head, so if you remove this reference it is no longer circular. This is possible but would result in inefficent time and no longer a circular doubly linked list.
3. O(n) is going to be the complexity due to a traversal of the list being required without the reference to the tail, this is because enqueing data requires the tail of the list, and now since there isnt a reference we must traverse each element of the list till we reach the tail O(n).
4. Yes, it is possible to do so. Since we are dealing with a circular doubly linked list, the tasks involved with deqeueing and enqeueing would not change, they would just happen at different places within the list. Consequently, reversal of head and tail would not effect the functionality of the queue and would not necessarily be a bad idea. However, for the sake of nomenclature, it is best to keep the roles as is.