The Repository

https://github.com/Drenns/ENSF338-Assignment-2

Work Performed by Each Group Member

Gio: 30123854
-Question 1
-Question 2
-Question 3

Daylen: 30160617
-Question 4
-Question 5

Question 1

1. Memoization is a technique of optimizing a search algorithm by storing previously found results from function calls so as a result any future function call that requires that value can easily access it without having to go through the same path again.
2. This part just includes the code.
3. The code calculates the Fibonacci sequence up to a specified number. For example an input of 5 would give the fifth fibonacci number (which also happens to be 5). If the input was 10, it would output 34 and so on for the other positive integers.
4. No, as a divide-and-conquer algorithm depends on a set of data. The only input is a single integer that produces a certain number after finishing. There is nothing to sort therefore it cannot implement a divide-and-conquer algorithm.
5. The time complexity of the given code is O(n^2). This is seen from the recursive calls of the function.
6. See ex1.3.py file in repository
7. The computational complexity of the updated function would be O(n) . Following the rules to calculate complexity, A) the longest path would be taken (which would also give us the worst case), so the path through the else statement where there would be no early return statements, B) the lowest order terms are discarded, so any single operations are not taken into account, and C) multiplicative constants are not to be taken into account which involve in the section of code involving two recursive function calls. Taking all of that into account, the complexity would be O(n) as the implementation of memoization cuts down on a lot of time needed as it does not need to calculate the same integers over and over again, and depends wholly on the length of time taken for the recursive calls.
8. Code and plotting shown in ex1.5.py file. Note that the fist plot (the plot of the changed function) is in thousandths of a second in comparison to the second one which is in seconds.

9. The plot for the original code is as expected, which is O(n^2) and is easily seen. The plot for the changed function is much more consistent which is to be expected with the implementation of memoization for the function, and as a result it tracks with the computational complexity of O(n).

Question 2

1. The code represents a Quick Sort implementation to sort through data, which is done through use of a divide-and-conquer method. In short, the data is sorted by having a pivot that data is checked against, and then the data gets switched to the proper side by going through the array. The average-case complexity would roughly be based on having to sort only half the data, as the worst case scenario would be if the data was completely out of order while the best case scenario would be already sorted. Taking this into account, the average-case complexity would be O(n log(n)) as the algorithm splits and handles each section of data and upon each split the data becomes half the size. The best case scenario would be when the pivots always make equal sized subarrays resulting in a tree with log(n) levels. This would also be of complexity O(n log(n)). The worst case scenario would be having the smallest or largest as the pivot for the array and subsequent subarrays and the complexity would be O(n^2).
2. See file ex2.2.py. Note: I was unable to get the values to work with arrays larger than 999. I decided on working with only 999 data inputs which may have been inaccurate but is the best to my ability when considering the sort and its functionality when handling all the data.
3. The result is relatively consistent, as the values are all in the hundredths of a second and the highest of values is still less than five hundredths of a second. The consistency can be attributed to Quick Sort, as the data sets are relatively similar in how they're ascending in order from lowest to highest.
4. The Quick Sort is fairly good already, as it properly organizes the data as needed and handles data ranging from 1000 data points to 10000. However, by using the first data point as the pivot, the worst case scenario will be when the data is already sorted or mostly sorted which is true for our data. This results in subarrays where one side is vastly larger than the other. A solution to this would be to utilize a pivot somewhere in between the first and last entry which is shown in the ex2.4.py.
5. Sorted data is good as it reduces the amount of back-and-forth the sort has to do with placing values, so there is little that needs to be changed. In the case of question 4, it can be changed to be random in order to be fully utilized with having the pivot be the first index, but with the changes in question 4 either way should yield similar results.

Question 3

1. If the data is sorted and uniformly distributed, it is faster than binary search. If the data set is smaller it is also less complex and accomplishes the same task in a small amount of time.
2. Yes the search will be affected, as it would become more difficult and time-consuming on one end compared to another. Take for example a data set based on phone books. If you were searching for an E name and had a large amount of D names and started closer to the beginning of the book, it would take you much longer than if it were uniformly distributed. It would be a similar effect if the ends had higher density, as it would take a longer time to get a name within these areas. Another key point about interpolation search is setting a “midpoint” where the value is likely to occur, and this would be impossible to do with unsorted data; without having a basis on where to roughly estimate the location in a dataset interpolation search because incredibly more unreasonable and inefficient.
3. The part of the code that would be affected would be the while loop, as it would have to be coded to take into account the different distribution. This could involve something like changing the pos variable calculation in order to find a better position to start with, as the likelihood of data being in the desired indices would be more complex to find. This could also involve changing the if and else statements underneath the pos variable as the change in distribution would potentially make it harder to accomplish the needed task if you only increment one each time.
4.
    a. Linear search is the efficient option when the data you’re given is not sorted in any way, or in a way that is inefficient for binary search or interpolation search. It’s also more useful when it comes to smaller sets of data, as the amount of work it takes to split up a small dataset can be accomplished in a similar time as with just plain linear search.
    b. As both Binary and Interpolation search require the data to be sorted in order to work properly, using divide-and-conquer would be unpredictable with data being in any position. A case where Linear search would outperform Interpolation search and Binary search would be any case where the data is unsorted beforehand, such as a list of license plates where you need to find a specific one. I don’t believe there’s any way to solve this for both Binary or Interpolation search, as they both hinge on the idea of sorted data with how they search through, and without sorted data this process would be inconsistent.

Question 4

Q4
a)

Arrays:

+ Advantages +

-Simple to implement in most programming languages.

-Cache-Friendly, as they are stored in contingent memory thus resulting in a faster access time.

-index-based accessing, makes it simple to read and accessed by just using the indices of the array.

Accessing an element takes O(1) time to complete.

- Disadvantages -

-Fixed size once it is created. leading to memory wastage in some cases. Not properly modified code will lead to checking every index in the array.

-Poor dynamic memory allocation, adding or removing elements requires shifting which is O(n) complexity for n elements.

-traversing

Linked Lists:

+ Advantages +

-Dynamic size, linked lists can increase and shrink when needed. More efficient than arrays in this regard.

-Insertion or deletion of elements in a linked list can be performed in O(1) complexity. This does not require any shifting or reallocation of memory.

- Disadvantages -

-Searching for an element in a linked list takes O(n) time.

b)

We can create a single replace function that inserts the value into the index requested. This act of insertion deletes the element at that index so we don't have to shift the array at all. We check to see if the index is less than the length of the list, if it is we execute what was mentioned above, if it is not then the index is not in bounds
of the array and can't be inserted.

This is a O(1) space and time complexity

c)

Selection Sort:

-Feasible-

traverses the linked list and finds the smallest element. This node has the head pointed to it and is deleted from its original place.

complexity is O(n^2)

does not differ between array and linked list as they both have to traverse the entire list.

Insertion Sort:

-Feasible-

Inserts elements into their correct position by updating pointers.

complexity is O(n^2)

does not differ, both is O(n^2)

differs when implemented, inserting into sorted sublist takes O(n) time while a linked lists pointers are O(1) to update.

Merge Sort:

-Feasible-

Divide and conquer sort that divides the list into smaller sublists recursively. Use a fast and slow pointer to find the mid point.

complexity is O(n * log n)

does not differ from an array

Bubble Sort:

-FEASIBLE but not efficient-

Swapping adjacent entries to order them and goes through all n elements in a linked list

complexity is O(n^2)

does not differ from an array

Quick Sort:

-FEASIBLE-

Divide and conquer technique, picking a pivot element and splitting the linked list into two parts. Traverse linked list and place elements

that are less than the partition to the left. After recursively sort these two lists.

complexity is O(nlogn) when the partition is not the highest or lowest value in the list. if this is the case

the complexity becomes O(n^2)

does not differ in time complexity of an array

Question 5

1.

a)

Because the stack is a last in first out (LIFO) structure.

By inserting items at the top of the stack, it can be ensured that the newest items will be at the top,
and removed first.

b)
Yes data can be added to the end of the stack this is done by traversing the whole linked list and updating the pointers.

c)
pushing (insertion) and popping are O(1)

2.
a)
This is to add elements to the end as it is a FIFO structure. The tail now acts as the 'head' as new data comes here, while old data is discarded. FIFO (grocery store line)

b)
yes, but it is inefficient as you have to traverse the entire linked list to find the end in order to insert.
which is O(n)

c)
without tail:
not possible because we add data at the end of the memory data.
but we could go through the entire list and find the end and add.
enqueuing is O(n) compared to O(1) with tail.
dequeuing is the same as it still has a head pointer. O(1) complexity.
d)
When adding elements to the front of a linked list (enqueueing) and removing elements from the end of the list (dequeuing), it's not efficient to use the head and tail pointers respectively. This is because when we remove an element from the end of the list, we need to change the next pointer of the previous element to point to nothing. This requires iterating through the entire list, making the code more complex and less efficient
3.
a1)
LIFO structure for a stack and data is added at the head as the head keeps track of the Last in First out element.

b1)
insertion of data at the end of the circular list can happen because access of the last node is available.

c1)

insertion and popping would be O(1) as we have head and tail pointers to each element. If we knew the element (first or last) we can simply update pointers.

a2)

data is enqueued at the tail of a FIFO structure

b2)

yes, we can access the last node because we have a pointer from the end to start. This makes it circular.

c2)

It is the same complexity as enqueuing and dequeuing.

d2)

it would be possible and a preferred method as we can access previous and next nodes with ease.