ENSF 338 ASSIGNMENT REPORT

Assignment #	2
Group #	51
Lab Section	B02
Names	Sergiy Redko (30151178), Nelson Thompson (30163519)

REPOSITORY LINK

https://github.com/SergiyRedko/338Assignment2

WORK PERFORMED BY EACH MEMBER

It is difficult to clearly break down work we performed, since we adopted pair programming approach for this assignment. Below is our best shot at it:

Task	*Person Responsible*
Exercise 1	Sergiy Redko
Exercise 2	Nelson Thompson
Exercise 3	Sergiy Redko
Exercise 4	Sergiy Redko
Exercise 5	Nelson Thompson

The work distribution is 50-50 among the team members as we did the whole assignment sitting next to each other.

EXERCISE 1

1.1

Memoization is the style of writing functions in which the results of previous iteration are recorded to not repeat the same work again. So far memoization has been used in recursive functions in this course, but similar technique can be applied to other programming methods as well.

Memoization is a good technique to use where the memory loss is far outweighed by performance gain. For instance, Fibonacci sequence calculation.

1.2 & 1.3

The function recursively calculates n^{th} number in Fibonacci sequence. Notably, this function is not using memoization.

1.4

This is an example of divide-and-conquer algorithm as far as any recursive functions is. It is difficult to argue the extent of abstraction of such a limited function though.

1.5

Each time the position of Fibonacci number increments, the amount of work doubles. This gives us time complexity of $O(2^n)$.

1.6

```
# Declare dictionary to store fibonacci numbers
fib_dict = {0:0, 1:1}
def fib(n):
    if n in fib_dict:
        return fib_dict[n]
    else:
        fib_dict[n] = fib(n-1) + fib(n-2)
        return fib_dict[n]
```

1.7

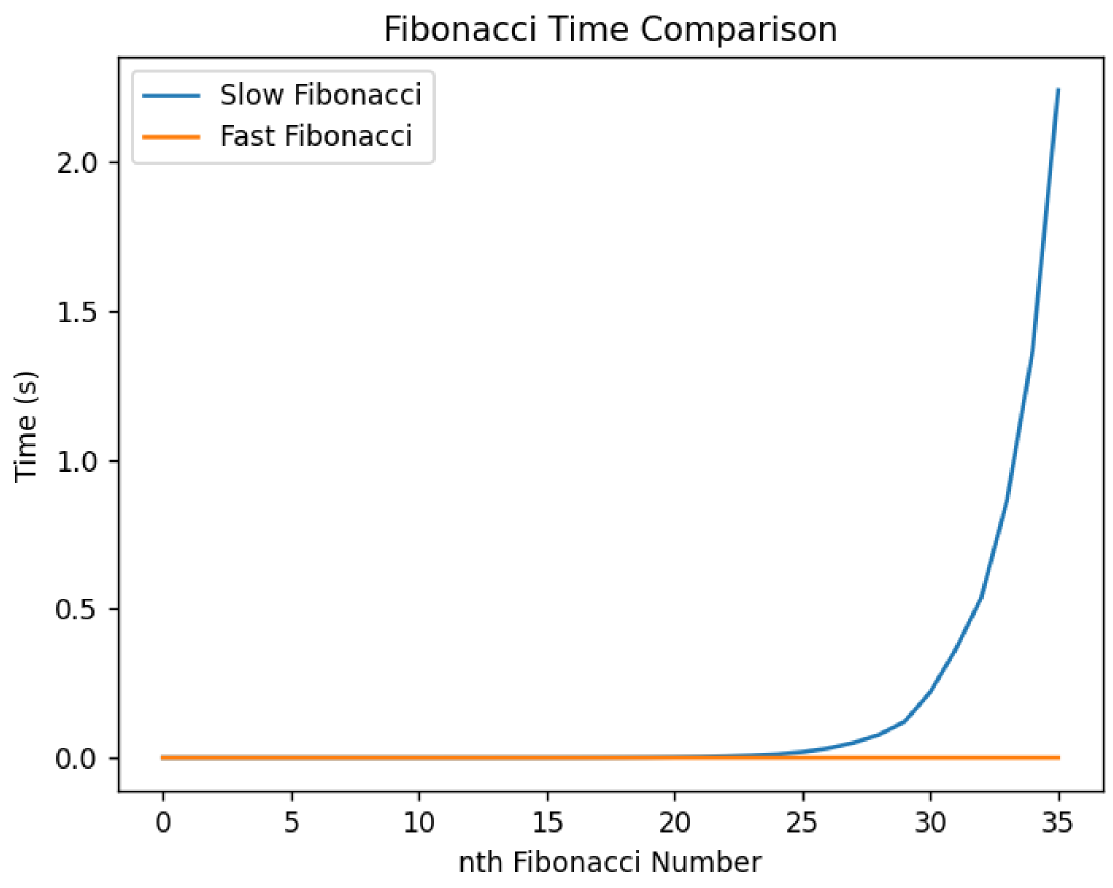
Best case scenario, we have already computed the value of fib(n), in which case its already in fib_dict. The complexity of that is $\Omega(1)$.

Worst case scenario, no entries exist in fib_dict other than the default ones. Each entry must be calculated once, this gives a linear relationship between n and complexity. Therefore, the time complexity is $O(n)$.

1.8

```
def func(n):
    if n == 0 or n == 1:
        return n
```

```python
    else:
        return func(n-1) + func(n-2)

# Declare dictionary to store fibonacci numbers
fib_dict = {0:0, 1:1}
def fib(n):
    if n in fib_dict:
        return fib_dict[n]
    else:
        fib_dict[n] = fib(n-1) + fib(n-2)
        return fib_dict[n]

if __name__ == "__main__":
    import timeit
    import matplotlib.pyplot as plt

    slow_fib = []

    for i in range(0, 36):
        slow_fib.append(timeit.timeit("func({})".format(i), setup="from
__main__ import func", number=1))

    fast_fib = []

    for i in range(0, 36):
        fast_fib.append(timeit.timeit("fib({})".format(i), setup="from
__main__ import fib", number=1))

    # Plot slow_fib and fast_fib in plt
    plt.plot(slow_fib, label="Slow Fibonacci")
    plt.plot(fast_fib, label="Fast Fibonacci")
    plt.legend()
    plt.title("Fibonacci Time Comparison")
    plt.xlabel("nth Fibonacci Number")
    plt.ylabel("Time (s)")
    plt.show()
```



1.9

The improved Fibonacci code remains fast regardless of the sample size, while the original implementation slows down logarithmically with the size of n. This supports our findings in parts 1.5 and 1.7 of this exercise.

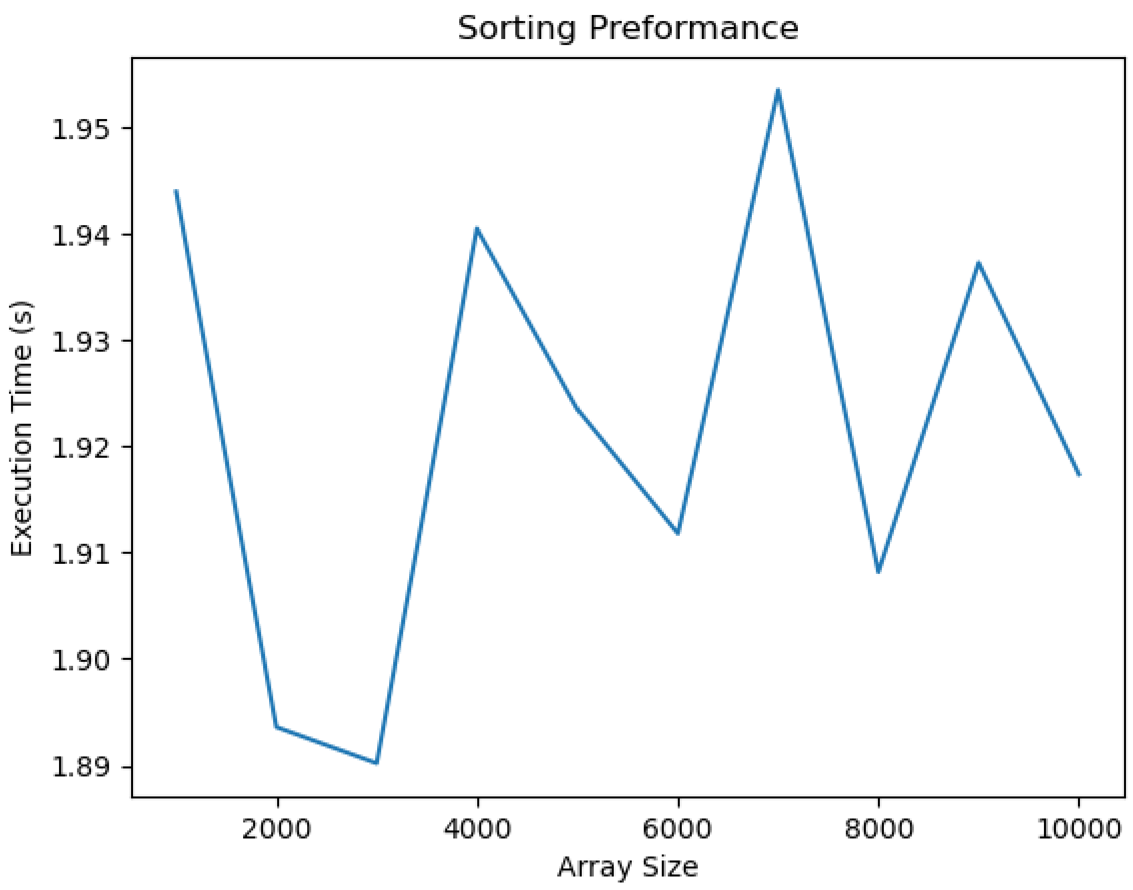
EXERCISE 2

2.1

This code is an implementation of the QuickSort algorithm. Func1 will take an array to be sorted along with both a high, and a low index which will represent the bounds of the part of the array which needs to be sorted. The function is recursive with each recursion being used to smaller sub arrays of one element each. Func2 selects a pivot element from the array and arranges elements based on whether they are larger or smaller than the pivot.

The QuickSort has a time complexity of O(n * log(n)), where "n" is the number of elements in the array. This is because portioning the array will, on average, divide the array into roughly equal heights. therefore, each level of recursion will process roughly half of the elements with log(n) levels of recursion. In each recursion the program preforms O(n) operations. This makes the overall time complexity O(n * log(n)

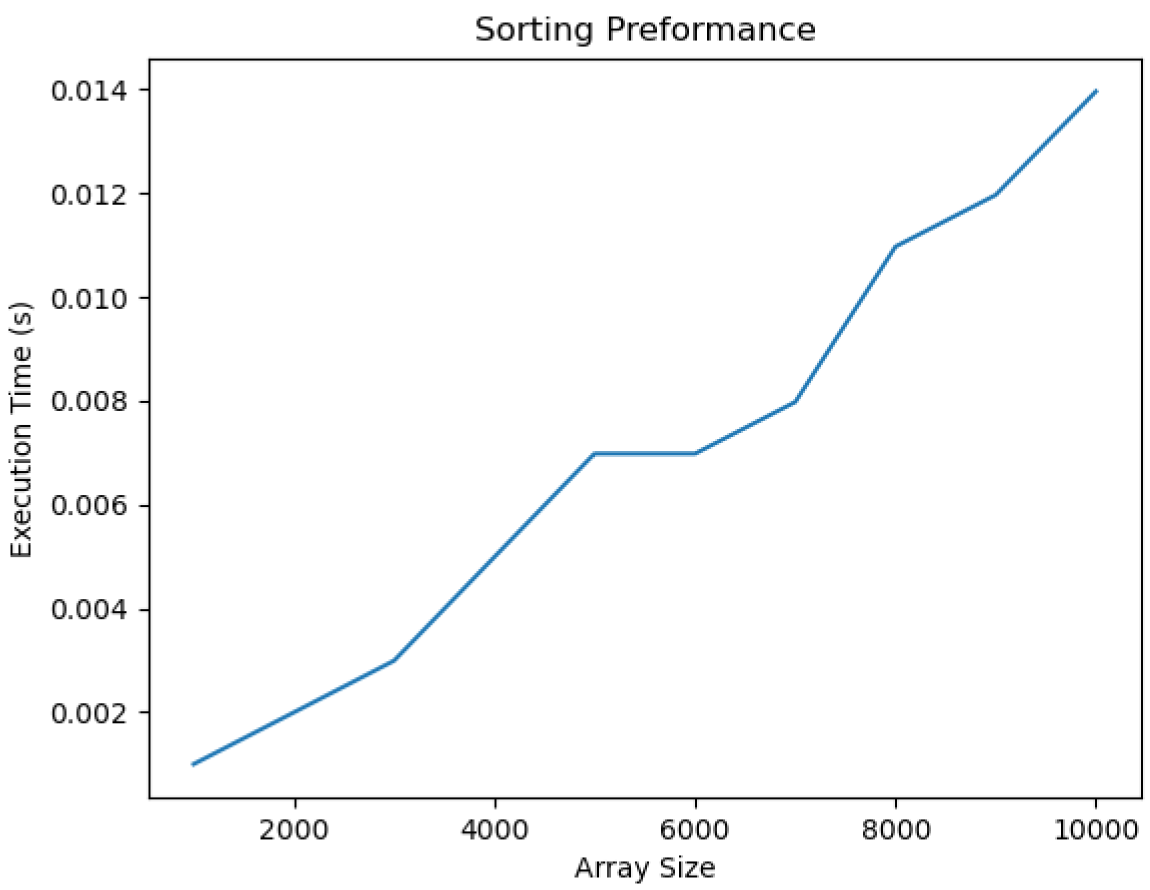
2.2



2.3

All the tests are quite close in timing and do not much change in timings as the array size increases. There is a small noticeable increase in the time taken as the array size increases but it is too small of a change to determine if this result is consistent with time complexity of O(n*log(n)). However we can predict the time complexity based on the organization of the input arrays and the program. Because the program always picks the first element in the array as the pivot, the worst case scenario for the program will occur when the array is close to fully sorted as each recursion will only sort a few items in the array. This worst-case scenario will result in a time complexity of O(n^2). The arrays provided are nearly-sorted so the time complexity would be near this worst-case scenario with these inputs.

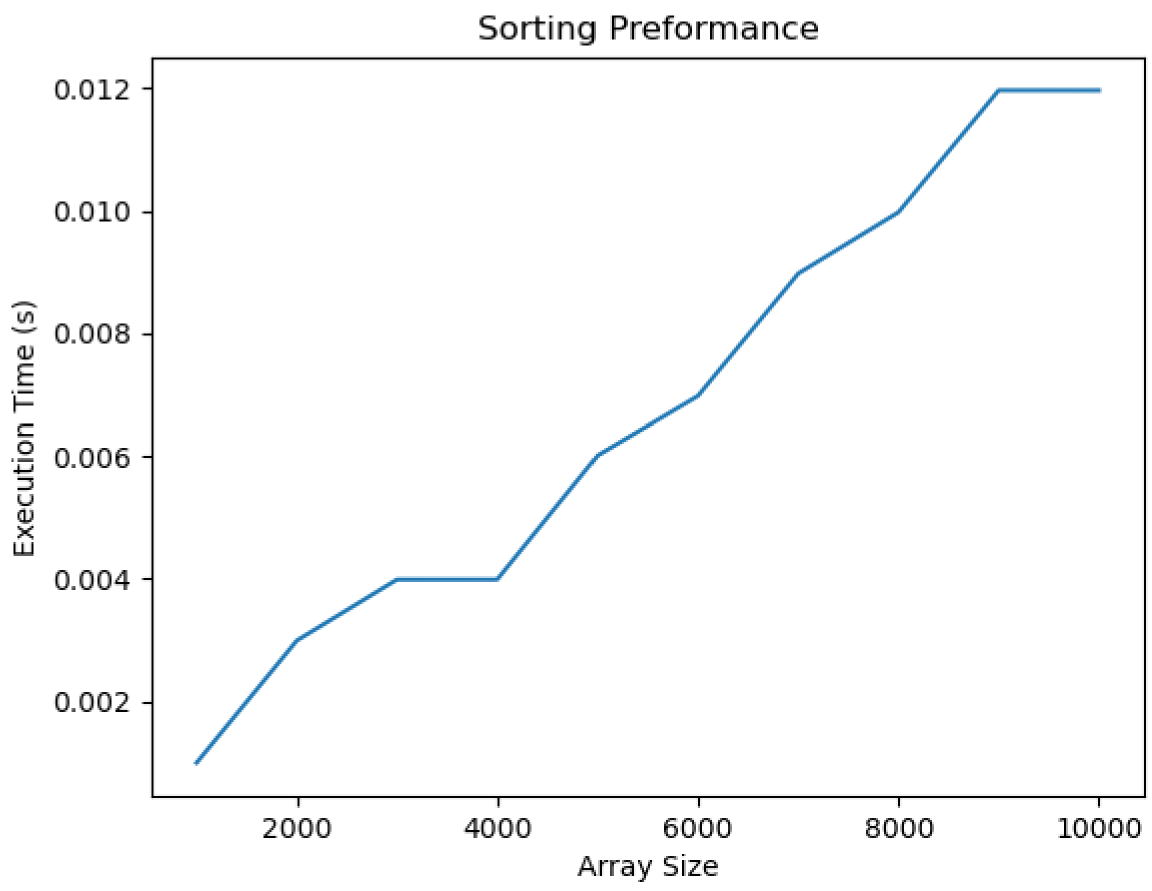
2.4



This plot matches the time complexity of O(n * log(n)) much closer than the plot produced by the given function. It can also be observed that the slowest execution time using the method is still much lower than even the fastest execution time of the previous method. This is because the new functions implement three key improvements:

1. Pivot is chosen randomly. By choosing the pivot randomly, the problem presented by having nearly sorted arrays is alleviated, now potentially splitting the entire array in halves at each recursion.
2. Early recursion termination. To avoid the taxing overhead that can occur when sub-arrays become small, the program now switches to an insertion sort once the subarrays become smaller than 11 elements.
3. Tail call optimization. In the original code, will always preform two recursive calls. This will lead to a large call stack which is quite inefficient when it comes to memory usage. To avoid this, the second recursive call can be made before returning from the first call. This process is known as Tail call optimization.

2.5



This plot matches the time complexity of O(n * log(n)) much closer than the plot produced by the given function. It can also be observed that the slowest execution time using the method is still much lower than even the fastest execution time of the assignment provided method. By giving the function arrays that are less sorted than previously, each recursion is more likely to split the array into two smaller arrays making the number of required recursion lessened.

EXERCISE 3

3.1

Interpolation search is not necessarily better than binary search. It is, however, better for uniformly distributed data (or any other type of known “interpolatable”, sorted data) since it jumps to where it expects data to be, rather than to the middle of remaining zone. The benefits of using interpolation search for uniformly distributed data:

- Less time complexity (works faster)
- Allows us to make an intelligent guess about data.

3.2

Yes, performance would be affected, usually for the worst. The closer the data is to uniform distribution the less steps would interpolation search have to perform to get to the result. Take 2 arrays for instance:

1. abcdefghijklmnopqrstuvwxyz
2. aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbccccccddddeeeeeeffff fffgggggghhhhhiiiiijjjjjkkkkklmnopqrstuvwxyz

Let us assume we are using our knowledge about English alphabet to find element b in this list. In both cases we would go close to the beginning of the array (and beginning of each following iteration), since that is where we expect to find the element. In case 1 it would take us just a few (likely 2) iterations to get to b. In case 2 it would take us around a dozen iterations.

For case 2 binary search would clearly be a better fit.

3.3.

We would change the way pos is calculated each iteration, since it is the “brains” of the search algorithm. Binary search is a good default (pos = (high + low)/2) and should only be changed if we are confident that data would be presented in a specific case.

3.4

A

Any case of non-sorted data. Binary search (and interpolation search, which is a type of binary search) are reliant on data being sorted. If they are run on unsorted data, the result would be unpredictable and likely wrong.

B

First, if the data is unsorted, linear search would perform better than sorting data and then searching it.

Second, if the item we are searching for has a tendency of being close to the beginning of the end of data (it would have to be one of these exclusively, and programmer would have to know which it is) linear search would have to go through fewer iterations to get there. Also, an iteration of linear search is faster than that of binary or interpolation search since there is less “thinking” involved.

Third, like second, if the data is heavily skewed to beginning or the end, and binary/interpolation search are not well-suited to traverse it.

Interpolation search might be improved, by modifying positioning algorithm to better reflect the distribution of data. It would become rather situational in this case though.

EXERCISE 4

4.1

Advantages and disadvantages of arrays and linked lists are dependent on situation. Below we discuss some advantages and disadvantages.

LOOKUP BY INDEX

Lookup by index has complexity O(1) in an array and O(n) in a linked list. This favors arrays in applications with frequent data lookup of specific members.

GROWING/SHRINKING

Growing/shrinking is O(n) for both linked list and array.

NOTE: insertion/deletion of element is O(1) for linked list if the position is known, but traversing to find that position is O(n).

Arrays, however, are disadvantaged for this operation. Insertion or deletion of an element must be followed by shifting the data for all pos ≥ n. This can be resource expensive, when compared to linked-list. Especially for operations at the beginning of array/list.

On top of that, array is not a mutable object. This means that in case array has run out of space a new array would have to be created and the old one would have to be copied into the new one in its entirety. That is another O(n) operation.

SEARCH

Search complexity depends on implementation and appearance of data. If data is unsorted or linear search is implemented complexity is O(n) for both array and linked list.

If data is sorted, or a sort more intelligent than linear sort is implemented array would typically have a better performance, since it is easier to traverse and index.

SORT

Will be compared in detail in part 3.

4.2

Both insertion and deletion are O(n) operations in array.

In order to optimize the process, both operations may be lumped together to prevent shifting of data. Easiest way to implement it:

Array[atThisLocation] = data_we_are_deletin_and_replacing_with_new

During lab 3 Maan Khedr mentioned that the data in the array is to be treated as if it is sorted. This changes the situation a little. Still, to optimize performance, both operations should be lumped together to minimize data shifting. In this case algorithm looks like the following:

1. Find location where we need to delete the data. Let us call this location *del*.
2. Find location where we need to insert the data. Let us call this location *ins*.

3. Shift all the data between *ins* and *del* (inclusive of *ins* location, but exclusive of *del* location; *del* location is to be overridden, but not shifted) towards *del* location.
4. Array[*ins*] = data_to_insert

This way we only need to shift |*del* – *ins*| number of elements, rather than 2 · num_elements – *ins* – *del*.

Though, the complexity (strictly) remains O(n).

When lumping these operations together, or even just performing delete before insert we always prevent the necessity for growing the array.

4.3

Each of these sorts can be implemented for both array and linked list. Some performance penalties may apply.

SELECTION SORT

In both array and linked list, it has O(n^2) complexity. The algorithm would be the same for both, but implementation differs.

In case of linked list there would be a lot of memorization involved for keeping track of the pointers to:

- current element,
- element preceding current element,
- element to swap the current element with,
- element preceding the element to sap the current element with.

Overall, implementation of this sort in singly linked list would have slightly lower performance.

INSERTION SORT

Here is how I would implement insertion sort in linked list:

1. First, it doesn't need to be an in-place sort. Create a temporary list.
2. Take out elements from the beginning of the original list and insertion-sort them (via injection) into the temporary list.
3. Copy the head from temporary list to head of original list.

Since there are up to n comparisons for up to n elements the complexity is still O(n^2); much like the array implementation.

Unlike in the array implementation, there is no need to shift elements for every insertion, which saves resources for every iteration. Insertion sort in linked list would be much faster than one in array.

MERGE SORT

Merge sort can be implemented in linked lists with the same complexity as in arrays. Fast pointer would have to be used to find middle. Once middle is found, list can be unlinked and sorted by recursive call to merge sort method.

The complexity is O(n·log(n)) for both array and list implementations.

BUBBLE SORT

Bubble sort is similar in both array and linked list implementation both are O(n^2), both would take the same number of iterations to sort identical data.

It is a terrible sort in both arrays and lists.

Traversing the linked list would not come with a performance penalty since the direction of movement is always ascending.

QUICK SORT

Similar to merge sort, quick sort can be implemented in linked list using slow and fast pointers. It can be split in two similarly to merge sort as well, and then call itself recursively on the partitions.

Much like the quick sort in array it has O(n^2) complexity in worst case scenario.

EXERCISE 5

1 Stacks

1a

Insertion (push), in stacks, adds the inserted data at the head due to following the Last-In-First-Out (LIFO) principle. In more detail this means that the most recently added element will be the first one remover or "popped" from the stack.

It is easy to manage popping the element at the head of a linked list, but not at the end of a singly linked list, since that would involve traversing towards the end (which is resource expensive).

1b

Yes, it is possible to insert data at the end of a linked list. It can be done by either traversing towards the end or keeping track of the last element.

1c

Inserting data at the head of a linked list ensures a constant time complexity of O(1) for both push and pop operations. This is a benefit of the Last-In-First-Out principle.

If data is inserted at the end of a linked-list, push and pop operations will have a linear complexity of O(n) since the program would have to traverse to the end of the linked list to perform the operation.

2 Queues

2a

This new pointer is used to allow efficient enqueueing of the data. Queues follow the First-In-First-Out (FIFO) principle which says that the first item added to the queue is also the first one to be removed. The new tail pointer is used to allow the program to reach the tail element in O(1) complexity.

2b

It is possible to implement a queue without a tail. One such way to do this would be through keeping count of the length of the queue and navigating to the end of the queue each time a new element is added. This method would have enqueuing complexity of O(n), which is a much worse method that that described in 2a.

2c

When a tail is used, both the enqueue and the dequeue will have an O(1) time complexity. When it is not used, the enqueue operation will have an O(n) time complexity since adding to the queue would require traversing to the end of the linked list. The dequeue operation will still have a O(1) complexity since the head node still points to the element to be removed. This showcases how the FIFO principle helps ensure efficient operation.

2d
Yes, we can. It is not a good idea though, since dequeue operations would have O(n) complexity (since we would have to traverse to n-1 element to unlink it from nth element regardless of whether tail is implemented.).

3 Stacks
Circular doubly linked list is a symmetric entity. As such the functionality of head and tail are interchangeable. Especially if the list class has pointers to both head and tail.

a
The head and tail may exchange their functionality with no performance penalty since head can access tail element as easily as the head one.

As such, there is no difference whether we enqueue/dequeue at head or tail.

b
In a circular doubly linked list, it is possible to insert data at the end of the list. Unlike in a singly-linked list, the tail is already connected to head element, thus complexity of adding the element at the tail is the same as that of adding an element at the head (which is O(1)).

c
Inserting data at the head of a linked list ensures a constant time complexity of O(1) for both push and pop operations. Using the circular nature of the list, even when inserting data at the end of the linked list, the push and pop operations will have time complexities of O(1) since it would now become possible to reach the end of the list in constant time.

3 Queues
Circular doubly linked list is a symmetric entity. As such the functionality of head and tail are interchangeable.

a
The tail pointer already exists in the head element we can implement another tail pointer to save a single traversing operation.

In a circular doubly linked list head and tail elements can be accessed equally easily (O(1) complexity) as such it doesn't matter whether we insert elements at the head, or tail. The choice of our enqueuing location would still dictate the popping location (which is opposite of enqueuing locaiton).

b
Well, sort of yes, sort of no. It is a requirement for a circularly linked list to have a pointer to tail from head. As such, we would always have an O(1) complexity access to tail. So, it is sort of impossible to implement queue without tail, since there always will be one in head element.

c
There is no difference for reasons described above.

d

Yes, we can. It will also yield no performance penalty.

It is not necessarily a bad idea, but not a good one either. If someone else looks at our code they would expect a queue to be implemented in a certain (common) way, where elements are enqueued at tail and dequeued at head. Implementing circular doubly liked list just to not comply with standard practices is… unwise.

That being said, if a particular application requires that this is done for some reason, it is going to be a viable option.