Practical Data Structures and Algorithms – ENSF 338

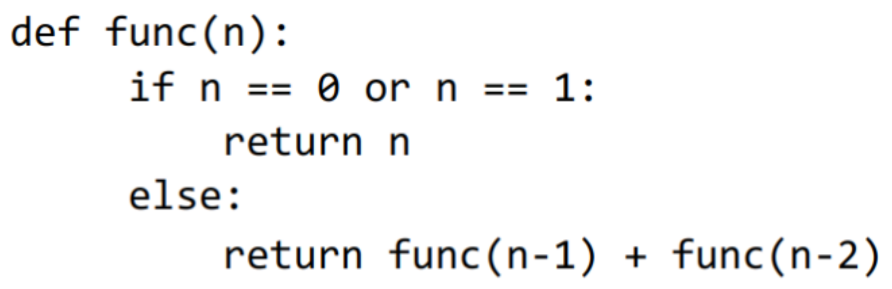
Members: Marshal Kalynchuk (30153895), Shahdeen Rahman (30141411)

Date Submitted: Feb 12, 2023

Repo: https://github.com/Marshal-Kalynchuk/ensf338_assignment_02.git

Exercise 1:

1. Memorization is an optimization technique used in programming. It makes applications more efficient and faster. Memorization works by storing computational results in a cache, and retrieving data from the cache as needed. This way, instead of computing the same data twice, it can just retrieve the data from the cache.
2. Consider the following code:

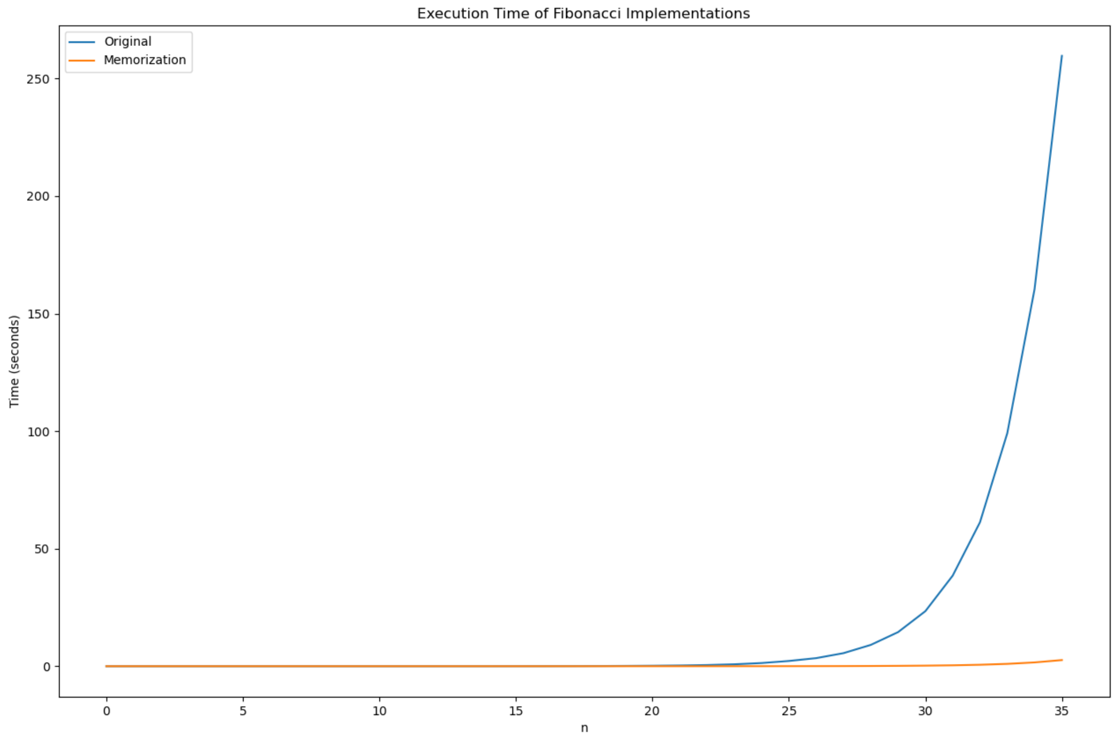


```
def func(n):
    if n == 0 or n == 1:
        return n
    else:
        return func(n-1) + func(n-2)
```

3. The code shown above is a recursive Fibonacci function. It computes the nth fibonacci number. The function works by recursively calling its-self to simplify the problem (n) into trivial subproblems (n == 0 or n ==1).
4. The code above is an example of a divide-and-conquer algorithm. However, its implementation is poor. The divide-and-conquer algorithm works by dividing non-trivial computational problems into smaller, more manageable subproblems. These subproblems are straightforward to solve. Divide-and-conquer generally employs recursion or loops to break the original problem up. The code above is an example of a divide-and-conquer algorithm because it divides the problem (n) into smaller subproblems. However, it is a poor implementation of the divide-and-conquer algorithm because it splits the problem into two subproblems that are almost as big as the original. Therefore, the time complexity is almost exponential.
5. The recursive Fibonacci algorithm's time complexity is poor. Its time complexity is *T(n)* or *$O(n^2)$, m*eaning its time complexity is exponential and inefficient.
6. Implementation of the recursive fibonacci algorithm augmented with memorization (ex1.3.py):

```
fib_cache = {}
def fibonacci(n):
   global fib_cache
   if n <= 1:
       return n
   if n not in fib_cache:
       fib_cache[n] = fibonacci(n-1) + fibonacci(n-2)
   return fib_cache[n]
```

7. The time complexity of the memorization implementation is *O(n)*. Each value is only calculated once and stored in fib_cache. This is a significant improvement over the original implementation, which has a time complexity of *O(n^2)*.
8. Testing of Timeit number=100 (ex1.5.py):



9. The plot above demonstrates the poor time complexity of the original Fibonacci implementation. The original Fibonacci function's time complexity is *T(n)*, illustrated by the exponential blue line. The function fibonacci_mem implements caching and is much more efficient than the original. Its time complexity is *O(n)*. The plot clearly shows that the memorization approach significantly improves the performance of the Fibonacci algorithm. For comparison, the computation time for n=35, the original Fibonacci implementation took 259.7 seconds. And the memorization Fibonacci function took 2.6 seconds.

Exercise 2:

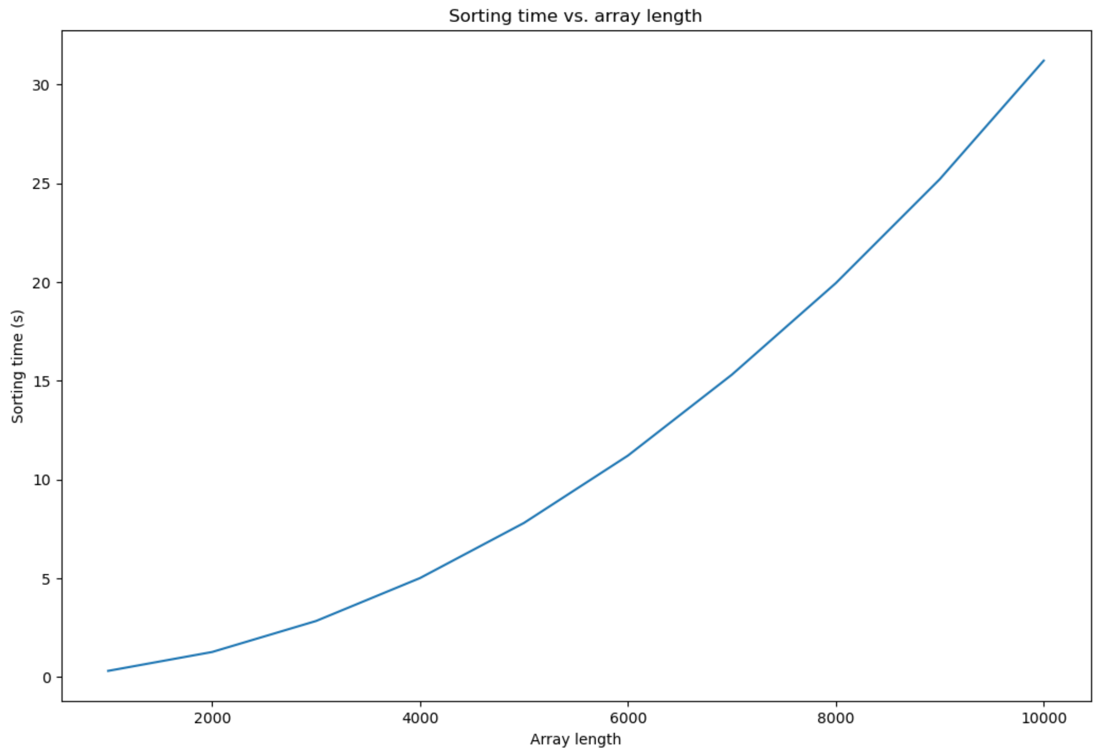
```
import sys
sys.setrecursionlimit(20000)
def func1(arr, low, high):
    if low < high:
        pi = func2(arr, low, high)
        func1(arr, low, pi-1)
        func1(arr, pi + 1, high)
def func2(array, start, end):
    p = array[start]
    low = start + 1
    high = end
    while True:
        while 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. The code above is an implementation of the QuickSort algorithm. It uses the divide-and-conquer strategy to sort an array. It works by choosing a pivot element, partitioning the array around the pivot, and then recursively sorting the sub-arrays on the right and left of the pivot.

   The ‘func2’ function performs the partitioning of the array. It starts by selecting the first element of the sub-array as the pivot and two pointers ‘low’ and ‘high’ that traverse the sub-array from either end. The while loop continuously swaps elements that are on the wrong side of the pivot until the ‘low’ pointer becomes greater than the ‘high’ pointer. The pivot is then swapped with the value pointed to by the ‘high’ pointer and its final position is returned. In the worst case, each iteration of the while loop will result in only one element being placed in its final position. Therefore, the time complexity of ‘func2’ is *O(n)*.

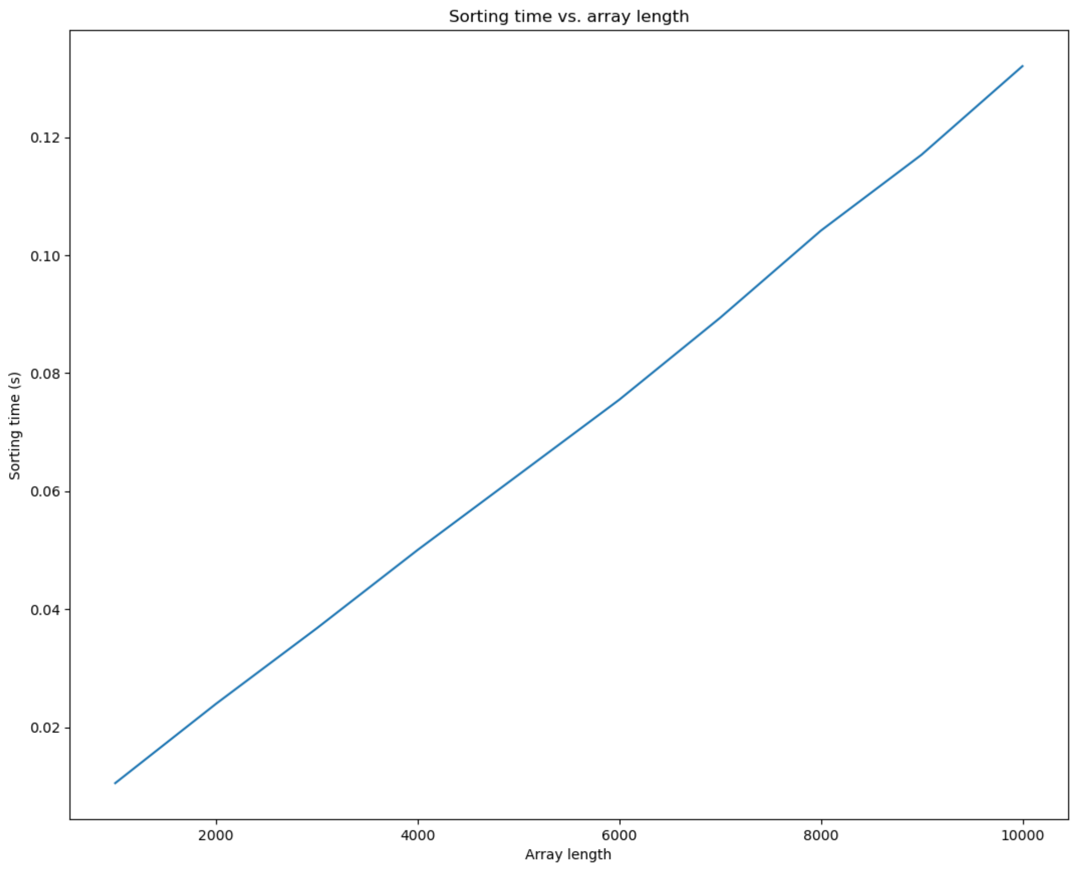
   The function ‘func1’ uses recursion to call itself twice on sub-arrays that are roughly half the size of the original sub-array. Because of this, the size of the sub-arrays will halve each time. This leads to a logarithmic number of calls to ‘func1’. In each call to ‘func1’, ‘func2’ is called once. As outlined above, ‘func2’ has a time complexity of *O(n)*. Therefore, the average time complexity of ‘func1’ is *O(n log n)*.

2. Testing of Timeit number=10 (ex2.2.py):

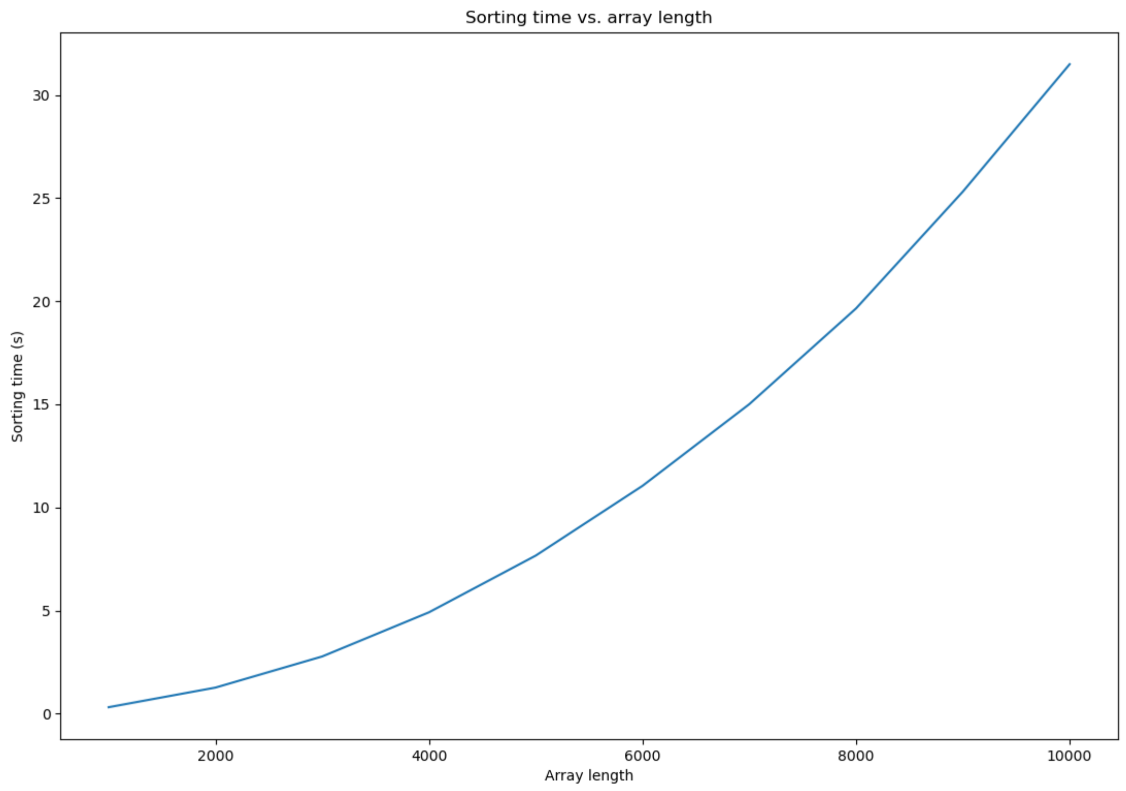


3. No the results are not consistent with the calculated time complexity in part 1. In part 1, the average time complexity was calculated to be O(n log n). However, the measured results are worse. The discrepancy between the calculated time complexity and the measured results may be due to a number of reasons. One possibility is that the data used for the measurement resulted in the worst-case scenario for the algorithm. Another possibility is that the algorithm is poorly implemented, which could lead to slower performance and a higher time complexity than expected.

4. Testing of Timeit number=10 (ex2.4.py):



5. Testing of Timeit number=10 (ex2.5) (randomized dataset):



Note: there is around one second of improvement overall.

Exercise 3:

1. Interpolation search can provide more accurate results by considering the value of the key being searched for. However in binary search, it assumes that all elements are equally likely to be searched for. Secondly, interpolation search can be faster than binary search in cases where the data is large and uniformly distributed.
2. The performance of Interpolation search can be affected if the data follows a different distribution, such as normal. If the data is not uniformly distributed, the calculation of the "pos" value in the interpolation search algorithm may not be as accurate, leading to a greater number of comparisons and a slower search.
3. To modify the Interpolation Search to follow a different distribution, the part of the code that would be affected is the calculation of the "pos" value. This calculation is based on the assumption of uniform distribution and would need to be modified to account for the different distribution of the data.
4. Linear search is your only option for searching your data if you do not have any information about the distribution of the data, or if the data is very small. In these cases, the overhead of more advanced algorithms like binary and interpolation search might outweigh the benefits.
    a. Linear search will outperform both Interpolation and Binary search if the data is very small or if the data is not sorted. In these cases, the overhead of sorting the data and the additional calculations required by Binary and Interpolation search would make linear search the faster option. To resolve this issue, Binary and Interpolation search could be combined with other techniques, such as hashing, to improve their performance on small and unsorted data.

Exercise 4:

Arrays and linked lists are two common data structures used in computer programming.

	Arrays	Linked lists
Advantages	● Efficient random access: accessing an element in an array can be done in O(1) time by using its index. ● Cache friendly: Elements in an array are stored in contiguous memory, making it cache friendly and resulting in better performance when reading data. ● Good for static data: If the data is static and not expected to change often, arrays can be a good choice.	● Dynamic size: Linked lists can be resized dynamically, making it a good choice for data that changes frequently. ● Efficient for insertions and deletions: Insertions and deletions in linked lists can be done in O(1) time.
Disadvantages	● Fixed size: Arrays have a fixed size and cannot be resized dynamically. ● Waste of memory: If the data stored in an array is sparse, the space occupied by unused elements will be wasted.	● Inefficient random access: Accessing an element in a linked list requires traversing the list, making it O(n) time. ● Cache unfriendly: Elements in a linked list are not stored in contiguous memory, making it cache unfriendly.

To implement a replace function that acts as a deletion followed by insertion in an array, one approach could be to implement the deletion by shifting all elements after the deleted element to fill the gap. This way, the insertion task can be completed in O(1) time since there will always be a gap in the array. The time complexity of this operation will be O(n) where n is the number of elements in the array.

Sort	Feasibility	Explanation
Selection sort	Not feasible	Selection sort has a time complexity of O(n^2) and it is not suitable for linked lists since linked lists have slow random access.
Insertion sort	Feasible	Insertion sort is feasible for linked lists because it works well with small data sets and it has a time complexity of O(n^2). The time complexity of insertion sort on a linked list is the same as that on an array.
Merge sort	Feasible	Merge sort can be used for linked lists because it can handle large data sets and has a time complexity of O(n log n). However, implementing merge sort on a linked list requires recursively splitting the list into halves and recombining them, which is not as straightforward as working with arrays.
Bubble sort	Not feasible	Bubble sort has a time complexity of O(n^2) and it is not suitable for linked lists since linked lists have slow random access.
Quick sort	Feasible	Quick sort can be used for linked lists because it can handle large data sets and has a time complexity of *O(n log n)*. However,

		implementing quicksort on a linked list requires choosing a pivot and partitioning the list around it, which is not as straightforward as working with arrays.

Exercise 5:

Part 1:

a. Stacks implement the Last-In-First-Out (LIFO) principle. Adding data to the head of the stack ensures that the most recently added item is at the top of the stack and can be easily removed first.
b. Theoretically, you can add a node to the end of a linked list. To insert an item at the end of a linked list, you would start at the head of the list and traverse the links until the end is reached. Then, you would create a new node with the desired value and set its next reference to null. Finally, you would update the next reference of the previous last node to point to the newly added node. However, this would violate the LIFO principle.
c. Pushing or popping data from the head of a stack has a time complexity of *O(1)*. However, inserting data to the end of a stack has a time complexity of *O(n)*. This is because it must traverse n nodes before the new data can be inserted.

Part 2:

a. Queues follow a First-In-First-Out (FIFO) implementation. A pointer is added to the end of a linked list in queues to improve the time complexity of adding data to the end of the list. Normally, the time complexity of adding an element to the end of a singly linked list is *O(n)*. However, if a pointer is added to the end, then the time complexity of adding an element to the end is *O(1)*. Adding a pointer to the end of the linked list allows FIFO to work.
b. Yes, it is possible to implement a queue without a tail pointer. This can be done using a linked list to keep track of both the head and the tail of the queue as you insert and remove items.
c. The implementation of a tailless queue is less efficient than a queue with a tail pointer. The tailless queue is less efficient because finding the last item in the queue has a time complexity of *O(n)*. Whereas finding the last item in a queue with a tail pointer has a time complexity of *O(1)*. In the context of enqueuing and dequeuing, a tailless queue has a time complexity of O(n) and a tailed queue has a time complexity of O(1).
d. Enqueuing at the head and dequeuing at the end defeat the purpose of a FILO data structure. This is not recommended because it makes the queue behave like a Last-In-First-Out (LIFO) data structure, which is not the intended behavior of a queue. In addition, this implementation is not efficient and would require traversing the entire linked list for each dequeue operation, which has a time complexity of O(n), where n is the number of items in the queue. This can lead to slow performance and decreased efficiency for large queues.
e.

Part 3:

Part 1:

a. A stack would behave the same way because it follows the LIFO data structure.
b. As previously stated, it is possible to insert an element at the end of a linked list. This violates the LIFO principle.
c. Pushing or popping data from the head of a stack has a time complexity of *O(1)*. However, In the case that the linked list is a circular doubly linked list, the time complexity for inserting an item at the end would be *O(1)*. This is because the end is linked to the head.

Part 2:

a. In the case that the linked list is a circular doubly linked list, a queue would still follow a FIFO approach.
b. As stated above, it is possible to implement a queue without a tail pointer.
c. The implementation of a tailless queue would not change the time complexity in the case that the linked list is circular. It would have a time complexity of *O(1)*. This is because the head is linked to the tail.
d. Making the linked list circular and doubly linked would not significantly change the response in part 2 d.