Question 1 b

i. Stack and Linked List

Think a stack like a bunch of books. They are put on top of each other. So you will take the first book to read not the last one. You will take a book that is more recent or that is on top. You would not take a book from down to read. If you have two books and you are reading the top one, and one book is left to read, you will add books above the one that is left that means last in first out. In the way of task scheduler, you do tasks from the one that are the most recent tasks. In stack is like you do the most recent task first or you can undo actions which is not good for managing tasks as scheduling tasks Operations like push, pop and peek are used in stack and also first in last is out is used. Undoing can be done easily in this part and also the cause of simplicity and efficiency.

While linked list have dynamic size for custom functionalities and are more complex to implement the undo function. The linked list is a node based structure where one node points to the next node. They can handle complex job management and can easily grow or shrink as needed. In linked list, the elements are accessed in sequence and each element points to the next one. Linked list can handle more complex navigation part and can easily grow and shrink as needed.

Question 1 c

Stack

Pros :

The stack is perfect fit for LIFO order where the most recent action is undone first and it is easy to manage. it carries the activities of pushing, popping, peeking, and checking if the stack is empty. Additional operations for instance the search of an element is not directly encouraged since it is complex to implement if needed and may thus be time consuming.

Cons :

Cons is that it is a limited use that allows for undoing the task in reverse order and it is not suitable for redo functionality. Unlike arrays or lists you can't just go straight to an element. You have to remove elements from the stack until you get to the one you want. That might not be very efficient.

linked list

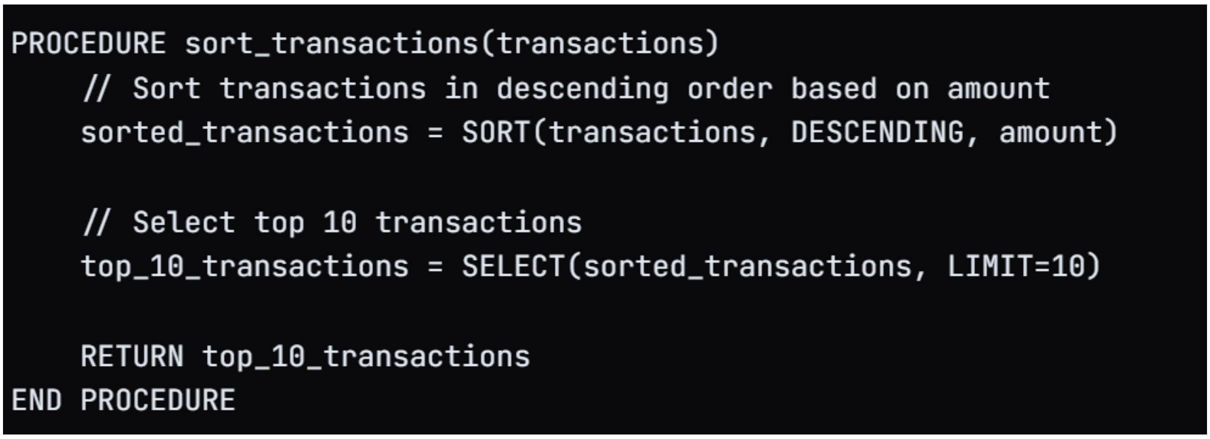
Pros :

it is flexible since it can implement complex undo functionality if extended to a doubly linked list. Linked lists do not have a fixed size. They can grow and shrink dynamically as elements are added or removed, making efficient use of memory. The size of a linked list is limited only by the system’s memory, unlike arrays, which have a fixed size determined at creation.

Cons :

It is more complex than needed for simple undo operations. To get to an element within the linked list one needs to go all the way to the head of the note for singly linked list or to the tail of the list in as case of the doubly linked list. This results in O(n) Accessing any general element in this linked list of linked lists takes an O(n) time. Implementing and debugging algorithms that use linked lists can be challenging because of how pointers/references must be managed, especially when adding or removing nodes.

Question 2 a (i)



```
PROCEDURE sort_transactions(transactions)
    // Sort transactions in descending order based on amount
    sorted_transactions = SORT(transactions, DESCENDING, amount)

    // Select top 10 transactions
    top_10_transactions = SELECT(sorted_transactions, LIMIT=10)

    RETURN top_10_transactions
END PROCEDURE
```

Question 2 (ii)

Big O (O) notation

It signifies the maximum amount of time the algorithm will take to run. It offers a time complexity worst-case scenario.

Omega (Ω) notation

Represents the lower bound of the algorithm's running time. It provides a best-case scenario for the time complexity.

Theta (Θ) notation

Represents the average-case running time of the algorithm. It provides a tight bound on the time complexity when the input size grows without bound.

Best Case scenario

If everything goes perfectly, all transactions will be arranged in descending order based on the amount. The heap will initially contain the first 10 transactions, and any following transactions will always be less than the minimum in the heap, resulting in no need to add or remove items from the heap. This would lead to a time complexity of O(n) or Ω(n), where n represents the quantity of transactions.

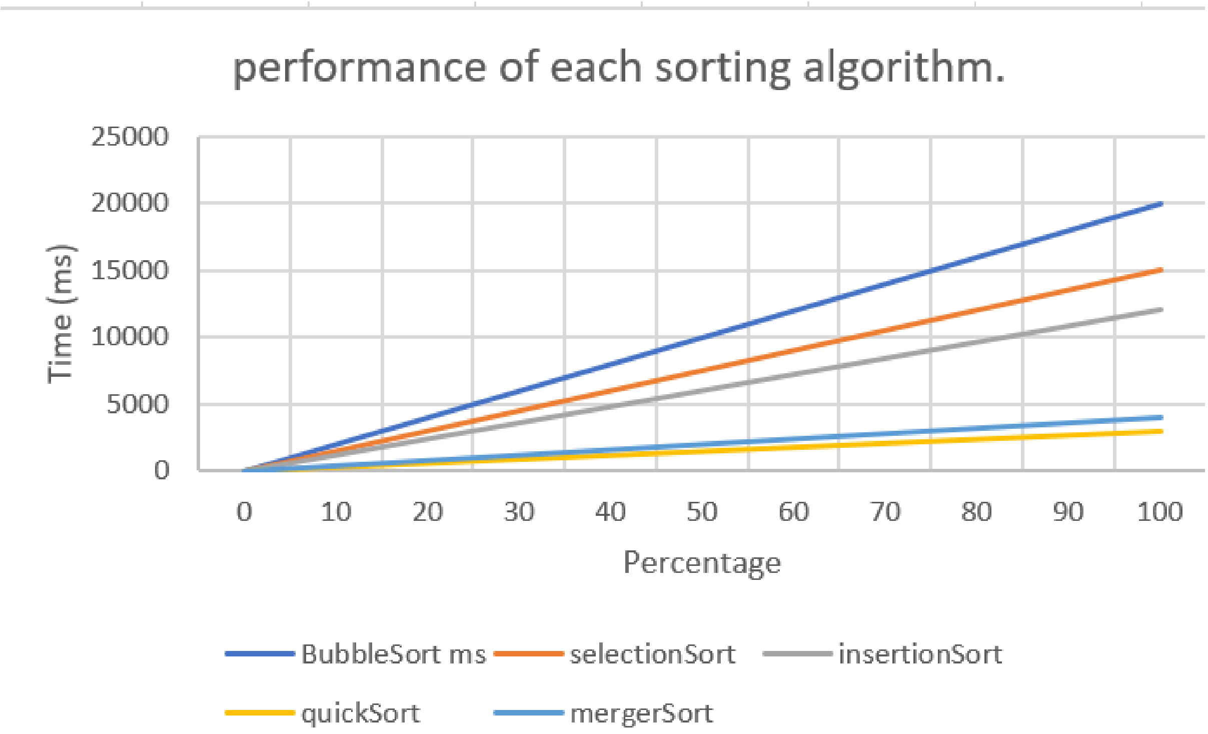
Worst Case scenario

If things go really bad, the transactions will be arranged from lowest to highest amount. After each insertion, the transaction is added to the heap and then the heap is reconstructed. This would lead to a time complexity of O(n log n).

Average Case:

The average case is more complex to analyze, but it is generally assumed to be closer to the worst-case scenario. Therefore, the average time complexity is also O(n log n).

In conclusion, the time complexity for the heap-based algorithm to find the top 10 transactions is O(n log n) on average and in the worst-case scenario, with n being the number of transactions. The upper limit of the time complexity is O(n) in the optimal scenario.


performance of each sorting algorithm.
Time (ms)
Percentage
BubbleSort ms
selectionSort
insertionSort
quickSort
mergerSort