C0S4015-B Technical and Professional Skills

Coursework 1

Team 3B

Hasan Akhtar(23011124), Hamza Alhalabi(23011223), Daniel Omylak(22011154), Uzair Ali(22009327), Shabbir Ahmad(23015739)

Psuedocode:

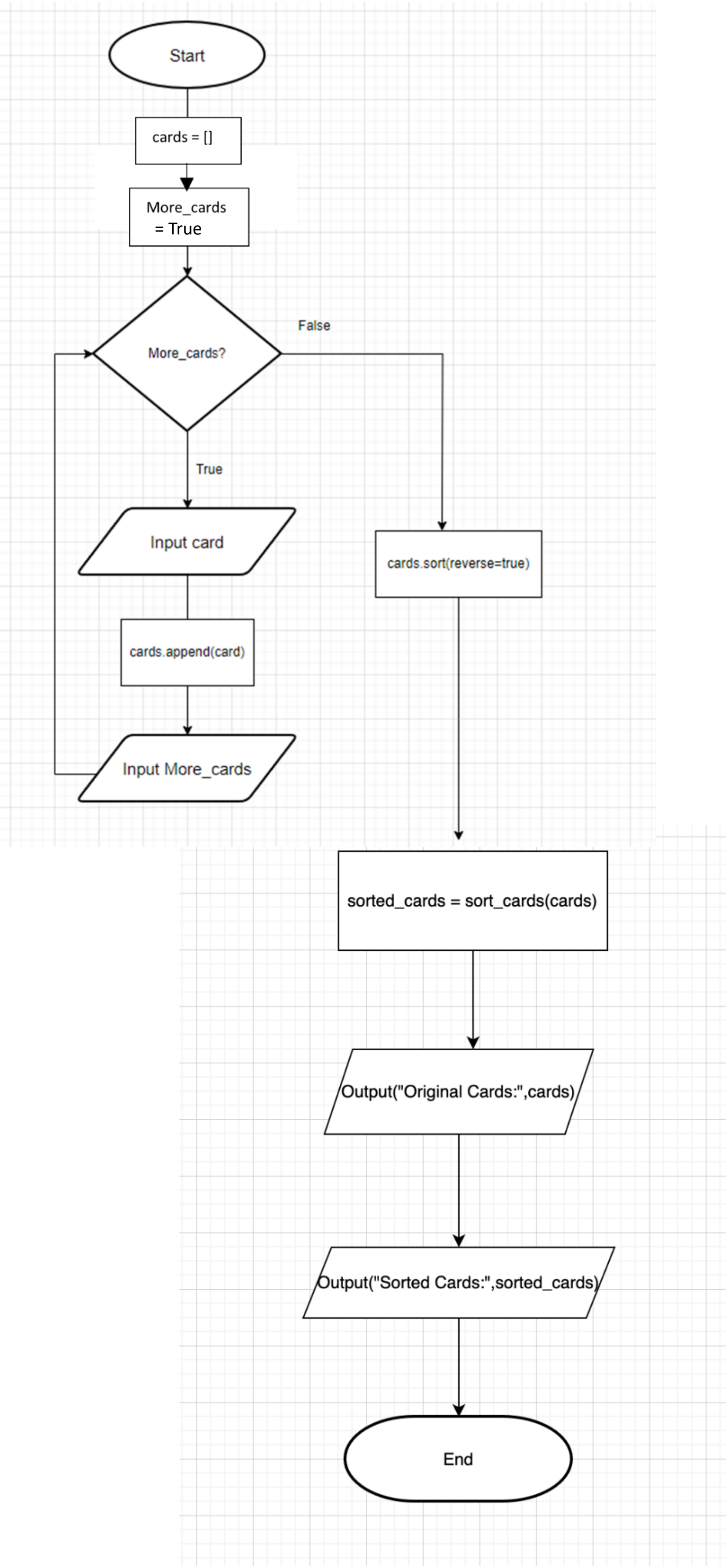
This code is using a merge sort to sort a number of cards which after the sort will merge together again. Before this, it will take user input for the cards (their integer representations) and then add them to an array and format the array to be how the problem requires it.

```
FUNCTION merge_sort(Cards)
    IF Cards.length <= 1 THEN
        RETURN Cards

    Middle = Cards.length // 2
    Left = Cards[:Middle]
    Right = Cards[Middle:]

    Left = merge_sort(Left)
    Right = merge_sort(Right)

    RETURN  merge(Left, Right)
ENDFUNCTION


FUNCTION merge(Left, Right)
    Result = []
    I = 0
    J = 0
    WHILE I < Left.length AND J < Right.length
        IF Left[I] < Right[J] THEN
            Result.APPEND(Left[I])
            I = I + 1
        ELSE
            Result.APPEND(Right[J])
            J = J + 1
        ENDIF
    ENDWHILE

    Result.EXTEND(Left[I:])
    Result.EXTEND(Right[J:])

    RETURN Result
ENDFUNCTION

FUNCTION sort_cards(Cards)
    Sorted_Cards = merge_sort(Cards)

    RETURN Sorted_Cards
ENDFUNCTION

cards = []
More_cards = True
```

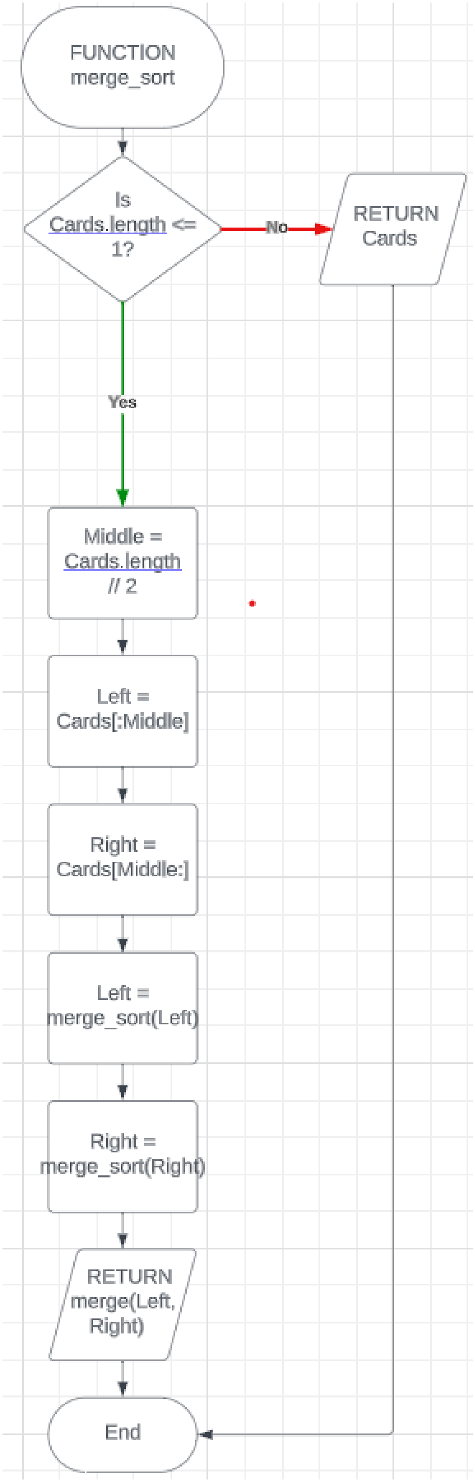
```
WHILE More_cards
        Input card
        cards.APPEND(card)
        Input More_cards
ENDWHILE
cards.sort(reverse=True)

sorted_cards = sort_cards(cards)
OUTPUT(“Original cards:”,cards)
OUTPUT(“Sorted cards:”,sorted_cards)
```

Flowcharts

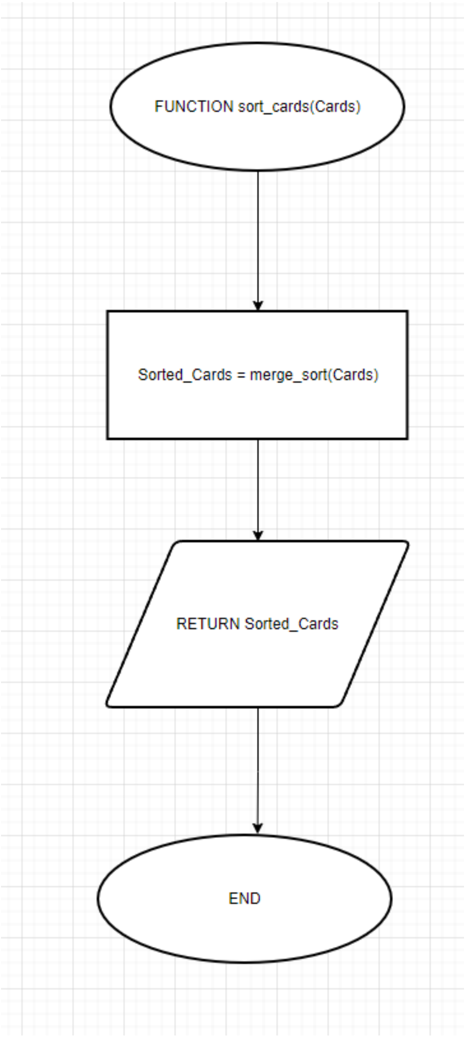
Main part:



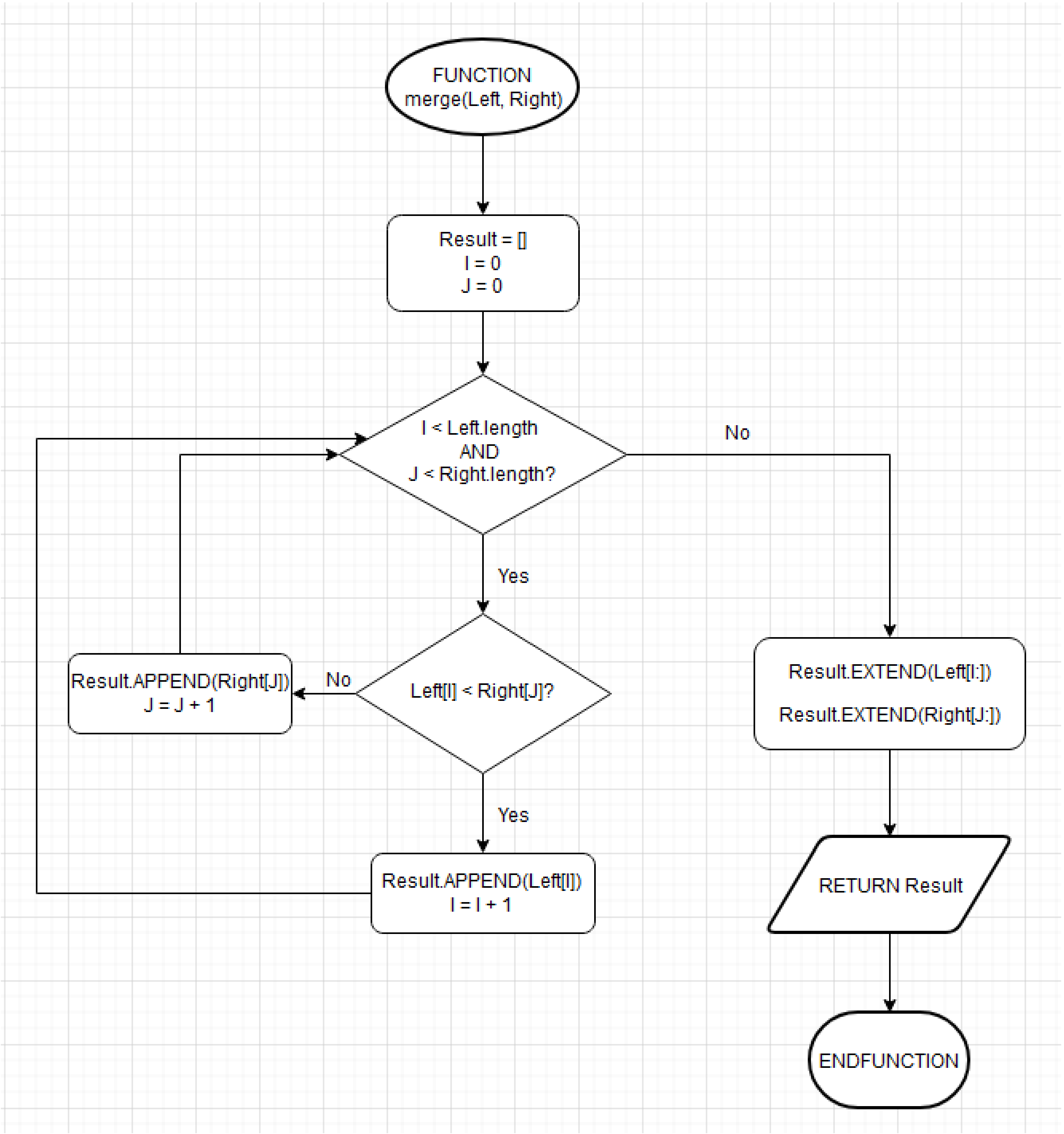
merge_sort Function:



sort_cards Function:



merge Function:



Computational Complexity

Merge sort is a sorting algorithm is a divide and conquer approach where it keeps dividing the list by 2 until it cannot be divided no more. And then it sorts them back in order and merges them back together to create a sorted array. (Time and Space Complexity Analysis of Merge Sort – GeeksforGeeks)

Divide and conquer approach is where a problem is been divided into smaller problems and conquer the subproblems by solving them and then combine the solutions of the subproblems to find a solution for the actual problem.

Time complexity of merge sort is O(n*log n) for all 3 cases (worst, average, best) as merge always uses the same approach of dividing the list and takes linear time to merge the halves together. (Merge Sort Algorithm | Studytonight)

Space complexity of merge sort is O(n) which is linear. (Time and Space Complexity Analysis of Merge Sort – GeeksforGeeks) This means that the bigger the list the more space it requires because more numbers will be required to be halved therefore more space will be required to carry out the tasks.

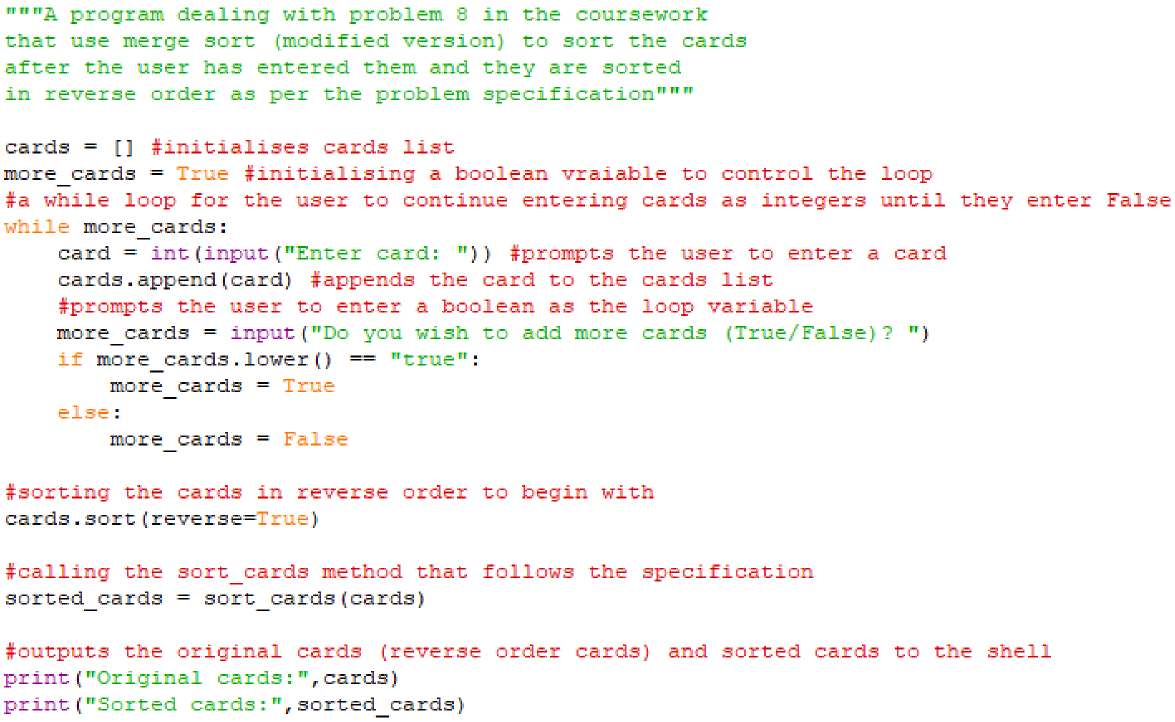
References:

https://www.studytonight.com/data-structures/merge-sort

https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-merge-sort/amp/

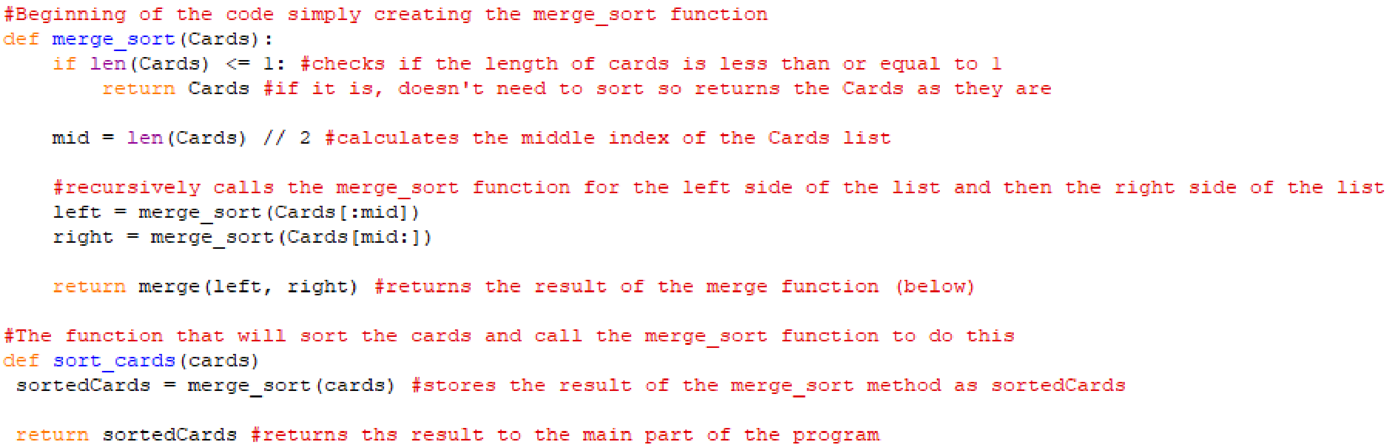
Program (our solution)

Version 1:



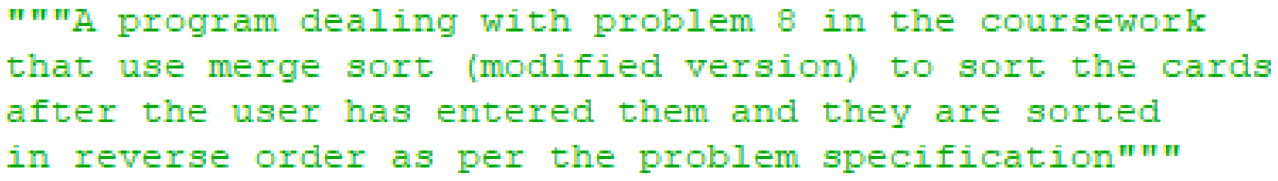
```
"""A program dealing with problem 8 in the coursework
that use merge sort (modified version) to sort the cards
after the user has entered them and they are sorted
in reverse order as per the problem specification"""

cards = [] #initialises cards list
more_cards = True #initialising a boolean vraiable to control the loop
#a while loop for the user to continue entering cards as integers until they enter False
while more_cards:
    card = int(input("Enter card: ")) #prompts the user to enter a card
    cards.append(card) #appends the card to the cards list
    #prompts the user to enter a boolean as the loop variable
    more_cards = input("Do you wish to add more cards (True/False)? ")
    if more_cards.lower() == "true":
        more_cards = True
    else:
        more_cards = False

#sorting the cards in reverse order to begin with
cards.sort(reverse=True)

#calling the sort_cards method that follows the specification
sorted_cards = sort_cards(cards)

#outputs the original cards (reverse order cards) and sorted cards to the shell
print("Original cards:",cards)
print("Sorted cards:",sorted_cards)
```

Version 2 (added code):



```
#Beginning of the code simply creating the merge_sort function
def merge_sort(Cards):
    if len(Cards) <= 1: #checks if the length of cards is less than or equal to 1
        return Cards #if it is, doesn't need to sort so returns the Cards as they are

    mid = len(Cards) // 2 #calculates the middle index of the Cards list

    #recursively calls the merge_sort function for the left side of the list and then the right side of the list
    left = merge_sort(Cards[:mid])
    right = merge_sort(Cards[mid:])

    return merge(left, right) #returns the result of the merge function (below)

#The function that will sort the cards and call the merge_sort function to do this
def sort_cards(cards)
 sortedCards = merge_sort(cards) #stores the result of the merge_sort method as sortedCards

 return sortedCards #returns ths result to the main part of the program
```

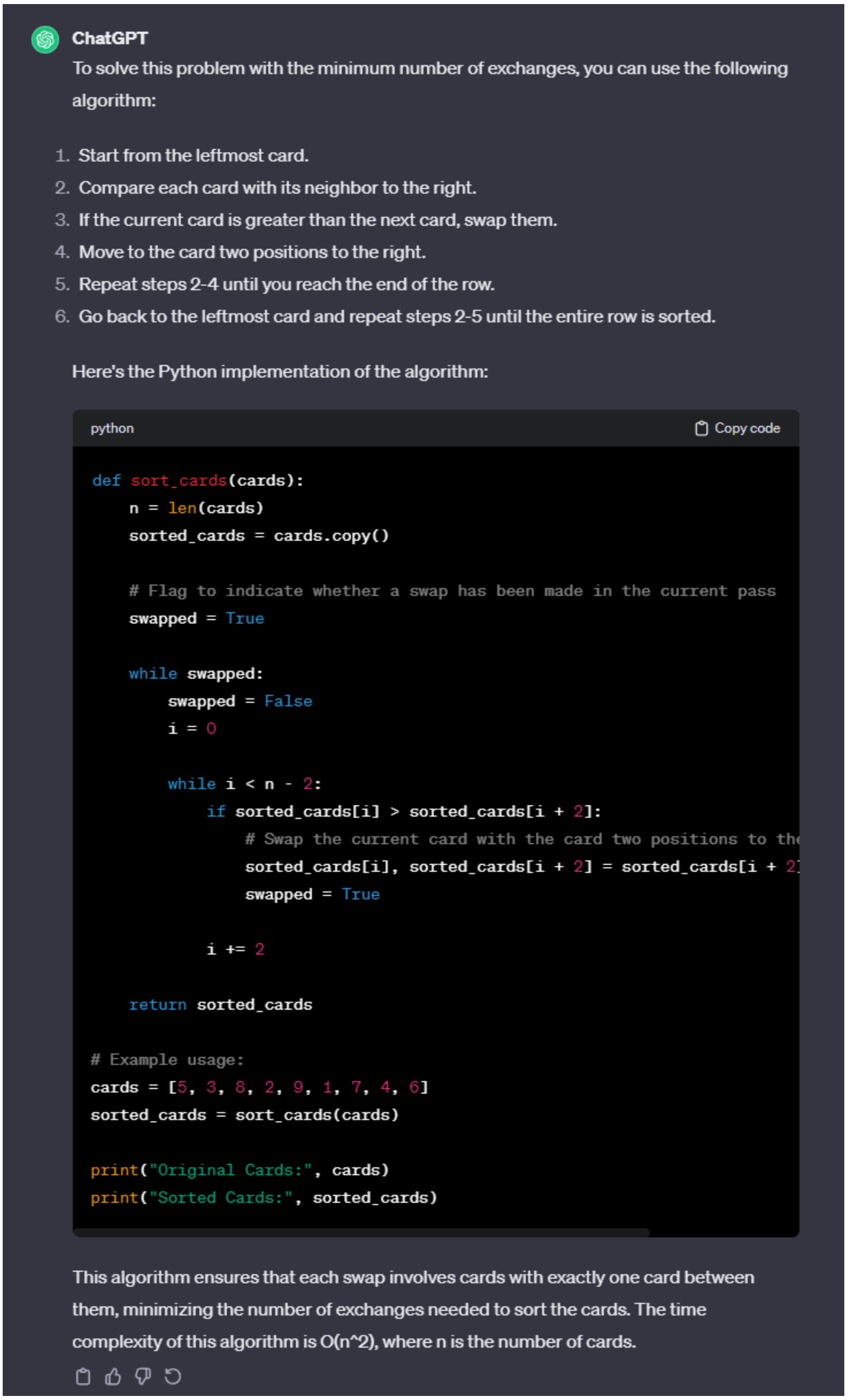
Version 3 (final version):



```
"""A program dealing with problem 8 in the coursework
that use merge sort (modified version) to sort the cards
after the user has entered them and they are sorted
in reverse order as per the problem specification"""
```

```
#Beginning of the code simply creating the merge_sort function
def merge_sort(cards):
    if len(cards) <= 1: #checks if the length of cards is less than or equal to 1
        return cards #if it is, doesn't need to sort so returns the Cards as they are

    mid = len(cards) // 2 #calculates the middle index of the Cards list

    #recursively calls the merge_sort function for the left side of the list and then the right side of the list
    left = merge_sort(cards[:mid])
    right = merge_sort(cards[mid:])

    return merge(left, right) #returns the result of the merge function (below)

#The merge function which will linearly merge the elements of the list together in sorted order
def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])

    return result

#The function that will sort the cards and call the merge_sort function to do this
def sort_cards(cards):
 sortedCards = merge_sort(cards) #stores the result of the merge_sort method as sortedCards

 return sortedCards #returns ths result to the main part of the program

cards = [] #initialises cards list
more_cards = True #initialising a boolean vraiable to control the loop
#a while loop for the user to continue entering cards as integers until they enter False
while more_cards:
    card = int(input("Enter card: ")) #prompts the user to enter a card
    cards.append(card) #appends the card to the cards list
    #prompts the user to enter a boolean as the loop variable
    more_cards = input("Do you wish to add more cards (True/False)? ")
    if more_cards.lower() == "true":
        more_cards = True
    else:
        more_cards = False

#sorting the cards in reverse order to begin with
cards.sort(reverse=True)

#calling the sort_cards method that follows the specification
sorted_cards = sort_cards(cards)

#outputs the original cards (reverse order cards) and sorted cards to the shell
print("Original cards:",cards)
print("Sorted cards:",sorted_cards)
```

Program (ChatGPT solution)



To solve this problem with the minimum number of exchanges, you can use the following algorithm:

1. Start from the leftmost card.
2. Compare each card with its neighbor to the right.
3. If the current card is greater than the next card, swap them.
4. Move to the card two positions to the right.
5. Repeat steps 2-4 until you reach the end of the row.
6. Go back to the leftmost card and repeat steps 2-5 until the entire row is sorted.

Here's the Python implementation of the algorithm:

```python
def sort_cards(cards):
    n = len(cards)
    sorted_cards = cards.copy()

    # Flag to indicate whether a swap has been made in the current pass
    swapped = True

    while swapped:
        swapped = False
        i = 0

        while i < n - 2:
            if sorted_cards[i] > sorted_cards[i + 2]:
                # Swap the current card with the card two positions to the
                sorted_cards[i], sorted_cards[i + 2] = sorted_cards[i + 2
                swapped = True

            i += 2

    return sorted_cards

# Example usage:
cards = [5, 3, 8, 2, 9, 1, 7, 4, 6]
sorted_cards = sort_cards(cards)

print("Original Cards:", cards)
print("Sorted Cards:", sorted_cards)
```

This algorithm ensures that each swap involves cards with exactly one card between them, minimizing the number of exchanges needed to sort the cards. The time complexity of this algorithm is O(n^2), where n is the number of cards.

The complexity of the ChatGPT algorithm is O(n^2) with n being the number of cards whilst our algorithm has a complexity of O(nlogn) with n also being the number of cards. Depending on the number of cards each algorithm has advantages. O(n^2) would be faster for a smaller number of cards whilst O(nlogn) would be faster for a larger number of cards and for the worst case scenario it would be considerably faster.

Team Evaluation

Hasan:

Hasan was elected to be the team leader at the beginning of this project. He has completed his part of the flowchart (Merge function part), pseudocode (merge function and everything outside of an function), converted his pseudocode to python code (merge function) and making it look more presentable (added comments, organised it, etc). Has attended every meet up session that was organised and was able to do so punctually. Was able to direct and give everyone in the group an equal amount of work to test their skills.

Uzair:

Uzair, independently used his own initiative to find the chatGPT solution above. He then wrote the comparison to chatGPT, mostly comparing the complexity of our algorithm to the complexity of the algorithm chatGPT wrote. He also, did the Main part of the flowchart (which was the bit where the user inputs numbers for the cards, the original cards are outputted, the sort_cards function is called and the sorted cards are outputted) and started writing the python code.

Shabbir:

Shabbir produced the pseudocode for the sort_cards function and also the flowchart for it.

Dan:

Dan was responsible for the merge_sort function of the pseudocode, flowchart and python code. He performed his tasks admirably and also put in some extra effort. One of the extra things he did was writing a short explanation for the pseudocode.

Hamza:

Hamza researched about the complexity of merge sort and wrote the full computational complexity section with references. He also pasted the psuedocode into the python file, ready for formatting into python code.