OPTIMIZING BUBBLE SORT

Project Calculus and Analytical geometry MT-1003

Made By

Ali Aan (23I-0708)

Owais Abdullah (23I-0808)

Abdul Saboor (23I-3039)

Section: CS-B

Table of Contents

Introduction and Objectives

Brief Introduction

Sorting algorithms are fundamental building blocks of computer science and are used to arrange data in ascending or descending order. They play an important role in a variety of applications such as data retrieval, database manipulation, and scientific computing. Among various sorting algorithms, bubble sort is a simple but inefficient algorithm that repeatedly compares adjacent elements and swaps them if they are out of order.

Objectives

This project focuses on bubble sorting algorithms and aims to investigate the application of calculus to optimize the efficiency of sorting algorithms.

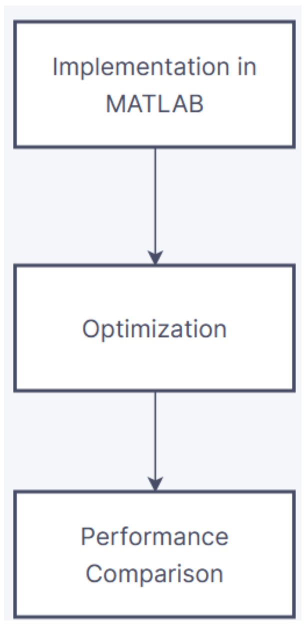
- Analyse the time complexity of the bubble sort algorithm.
- Identify and apply optimization techniques to reduce the number of comparisons or substitutions required by bubble sort algorithms.
- Compare the performance of the original bubble sort algorithm with the optimized version.

Solution Methodology

The solution involves the following steps:

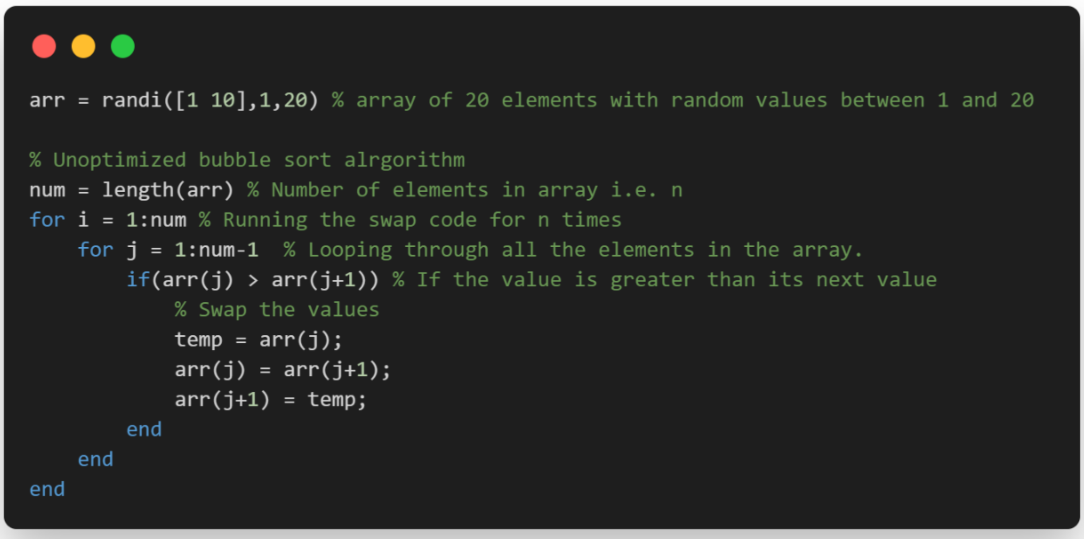
- Implementation of the bubble sort algorithm in MATLAB.
- Determining the function of comparisons required with respect to the number of elements of array.
- Optimizing the bubble sort algorithm using calculus.
- Optimizing the algorithm with other techniques.
- Performance comparison between the optimized and unoptimized algorithm.

Solution Methodology Flow Chart



Implementation of bubble sort in MATLAB

The bubble sort algorithm was implemented in MATLAB. First a test array was generated with random values. After that the array was looped n times for comparisons loop. The comparison loop was looped n-1 times through the values of the array. A comparison was made between adjacent values of the array and if the first value is greater than the second value then the values are swapped.



```
arr = randi([1 10],1,20) % array of 20 elements with random values between 1 and 20

% Unoptimized bubble sort alrgorithm
num = length(arr) % Number of elements in array i.e. n
for i = 1:num % Running the swap code for n times
    for j = 1:num-1  % Looping through all the elements in the array.
        if(arr(j) > arr(j+1)) % If the value is greater than its next value
            % Swap the values
            temp = arr(j);
            arr(j) = arr(j+1);
            arr(j+1) = temp;
        end
    end
end
```

This function sorts any array given in ascending order.

Understanding Time Complexity

Worst-case scenario: In the worst case, the algorithm has to make n passes through the array, and in each pass, it compares and swaps elements. This results in n*(n−1) comparisons and swaps. The time complexity is therefore O (n^2) because the dominant term in the expression is n^2.

Best-case scenario: The best-case scenario occurs when the array is already sorted. However, even in this case, Bubble Sort requires n−1 passes through the array to confirm that no swaps are needed. The time complexity is still O (n^2) in the best case.

Bubble Sort's time complexity is O (n^2), which means the time it takes to sort a list grows quickly as the list gets larger. If you double the size of the list, the time it takes to sort it doesn't just double but quadruples. This makes Bubble Sort not very efficient for big lists compared to some other ways of sorting.

Significance:

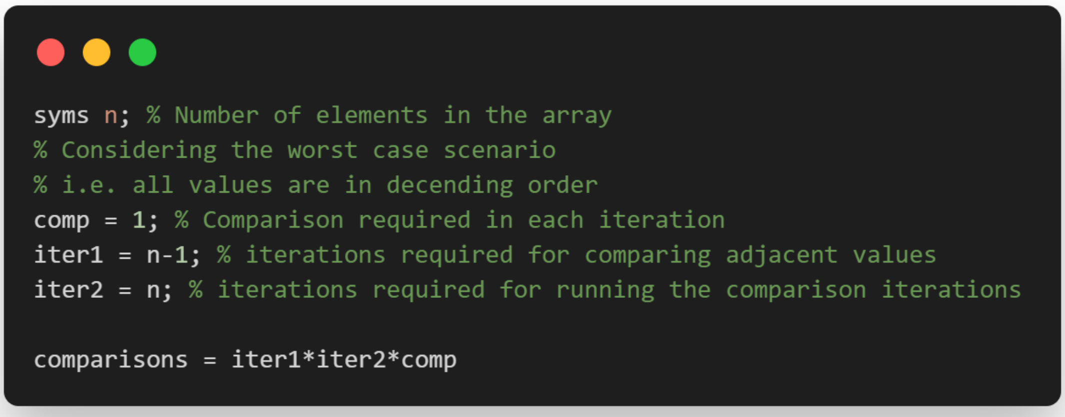
The significance of the O (n^2) time complexity in algorithmic efficiency is that Bubble Sort is not very efficient, especially for large datasets. As the size of the input grows, the number of comparisons and swaps increases quadratically. This makes Bubble Sort impractical for sorting large datasets compared to more efficient sorting algorithms with lower time complexities, such as Quick Sort (O (n log n)) or Merge Sort (O (n log n)).

Bubble Sort is often used for educational purposes because of its simplicity, but it's rarely used in practical applications where sorting efficiency is crucial. Understanding the time complexity helps in choosing the right sorting algorithm based on the size and characteristics of the input data.

Derivation and Optimization Process

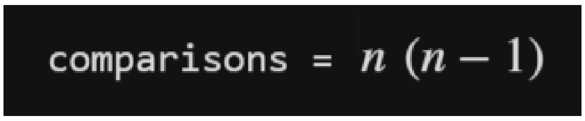
MATLAB Task

Writing total number of comparisons as the function of total number of elements ‘n’



```
syms n; % Number of elements in the array
% Considering the worst case scenario
% i.e. all values are in decending order
comp = 1; % Comparison required in each iteration
iter1 = n-1; % iterations required for comparing adjacent values
iter2 = n; % iterations required for running the comparison iterations

comparisons = iter1*iter2*comp
```

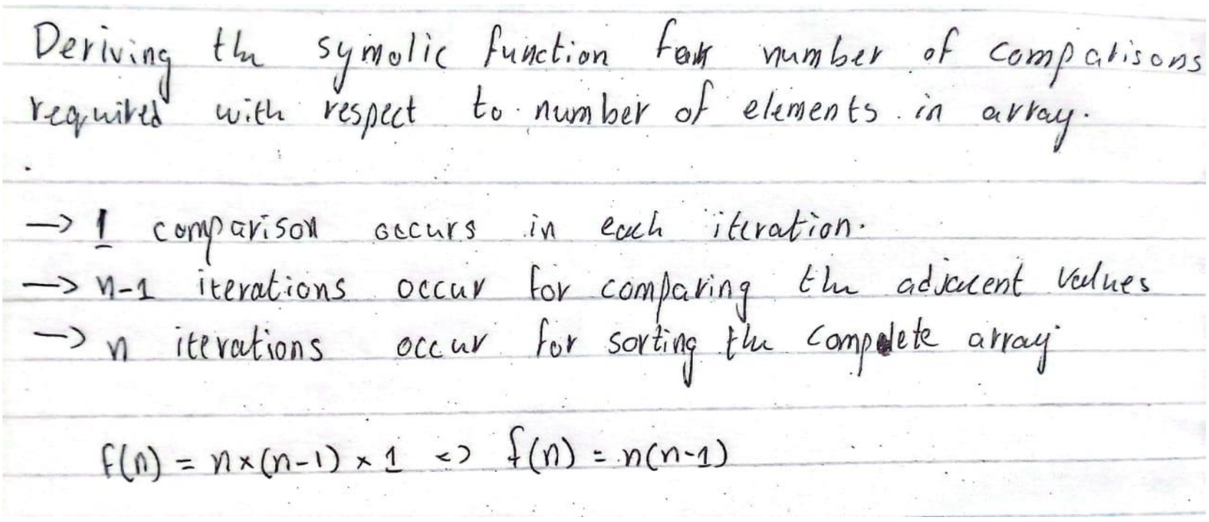
Output: 

comparisons = $n\ (n-1)$

$$f(n) = n(n-1)$$

By Hand Task

a.



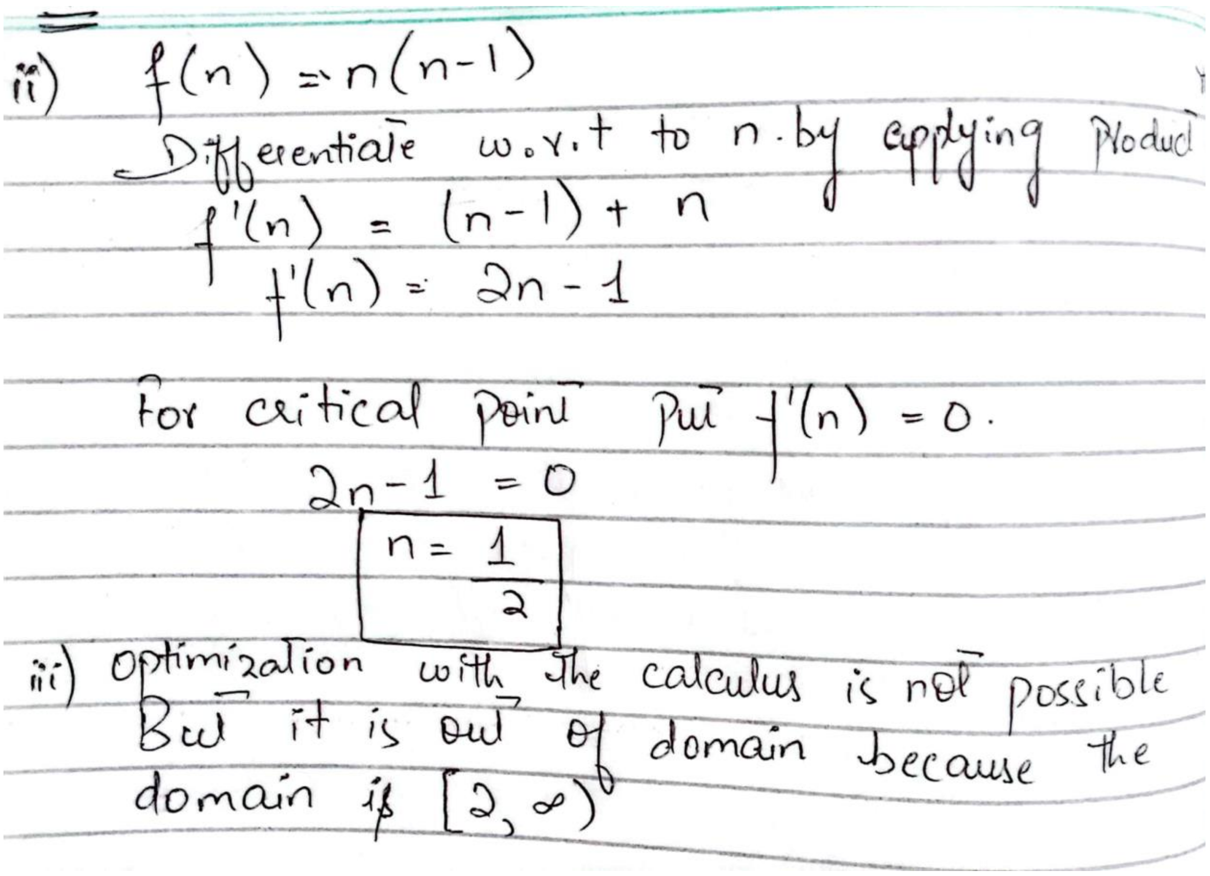
Deriving the symolic function for number of comparisons required with respect to number of elements in array.

→ 1 comparison occurs in each iteration.

→ n-1 iterations occur for comparing the adjacent values

→ n iterations occur for sorting the complete array

$f(n) = n \times (n-1) \times 1 \iff f(n) = n(n-1)$



ii) $f(n) = n(n-1)$

Differentiate w.r.t to n. by applying Produc

$f'(n) = (n-1) + n$

$f'(n) = 2n - 1$

For critical point put $f'(n) = 0$.

$2n - 1 = 0$

$$\boxed{n = \frac{1}{2}}$$

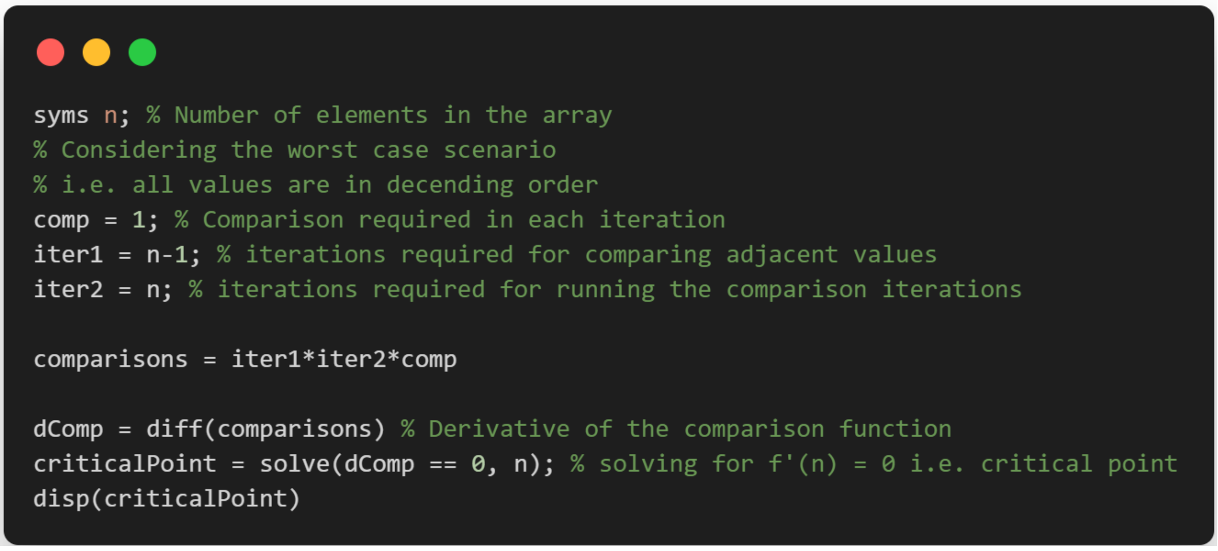
iii) Optimization with the calculus is not possible But it is out of domain because the domain is $[2, \infty)$

The solution by hand is same as the solution acquired by using the MATLAB's symbolic toolbox.

Optimization Using Calculus

MATLAB Task

Finding the derivative of the comparison function. Then solving for $f'(n) = 0$ to find the critical point.

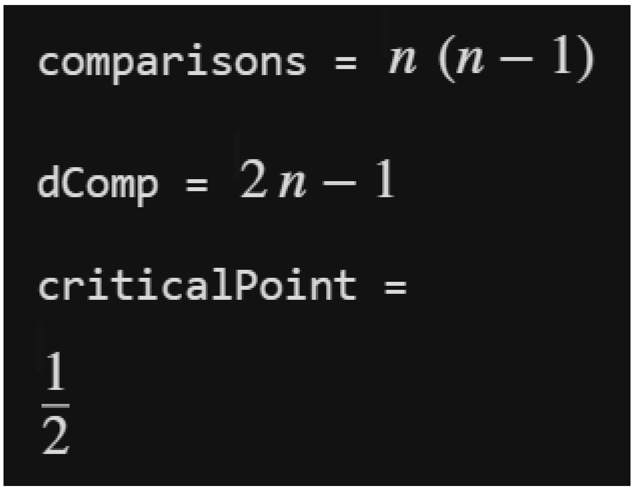


```
syms n; % Number of elements in the array
% Considering the worst case scenario
% i.e. all values are in decending order
comp = 1; % Comparison required in each iteration
iter1 = n-1; % iterations required for comparing adjacent values
iter2 = n; % iterations required for running the comparison iterations

comparisons = iter1*iter2*comp

dComp = diff(comparisons) % Derivative of the comparison function
criticalPoint = solve(dComp == 0, n); % solving for f'(n) = 0 i.e. critical point
disp(criticalPoint)
```

Output:



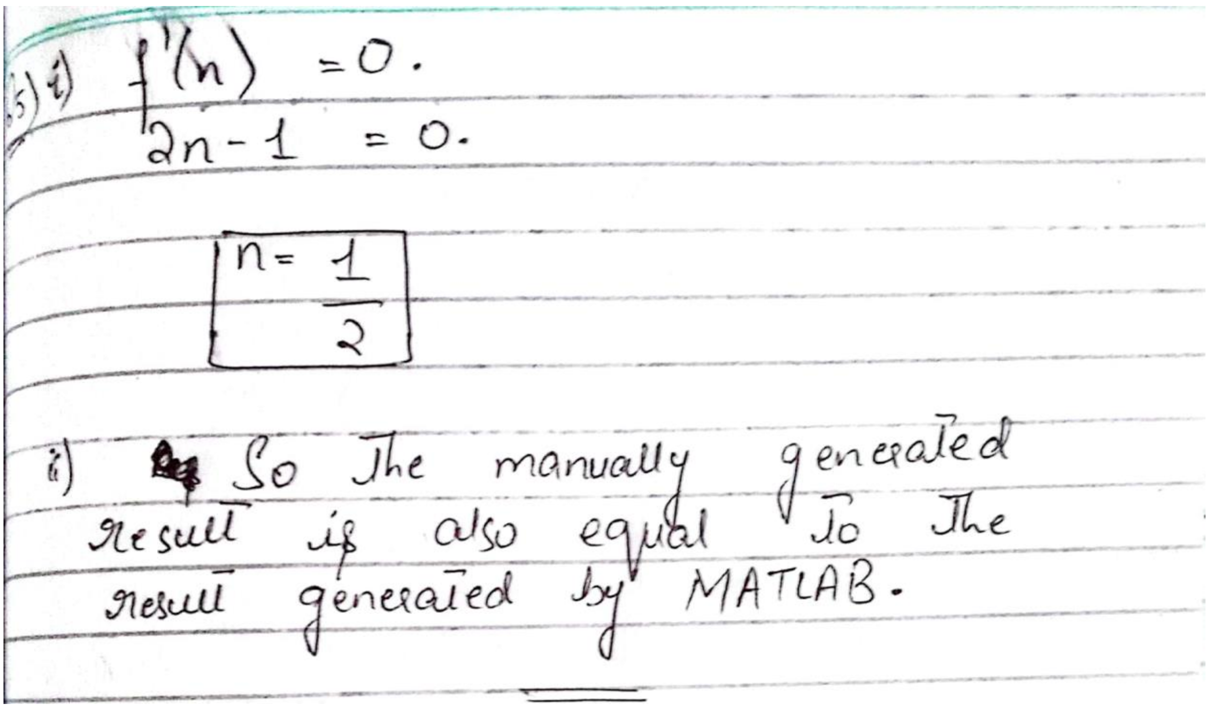
```
comparisons = n (n − 1)

dComp = 2n − 1

criticalPoint =

1
─
2
```

By Hand Task



5) i) $f'(n) = 0.$

$2n - 1 = 0.$

$\boxed{n = \frac{1}{2}}$

ii) So The manually generated result is also equal To The result generated by MATLAB.

The critical point comes out to be 0.5, which is not a part of the domain. Domain is $[2, \infty)$ since the array must have at least 2 values for sorting. Thus, optimizing the function using calculus is difficult, however, certain algorithmic techniques can be used for further optimization of the function.

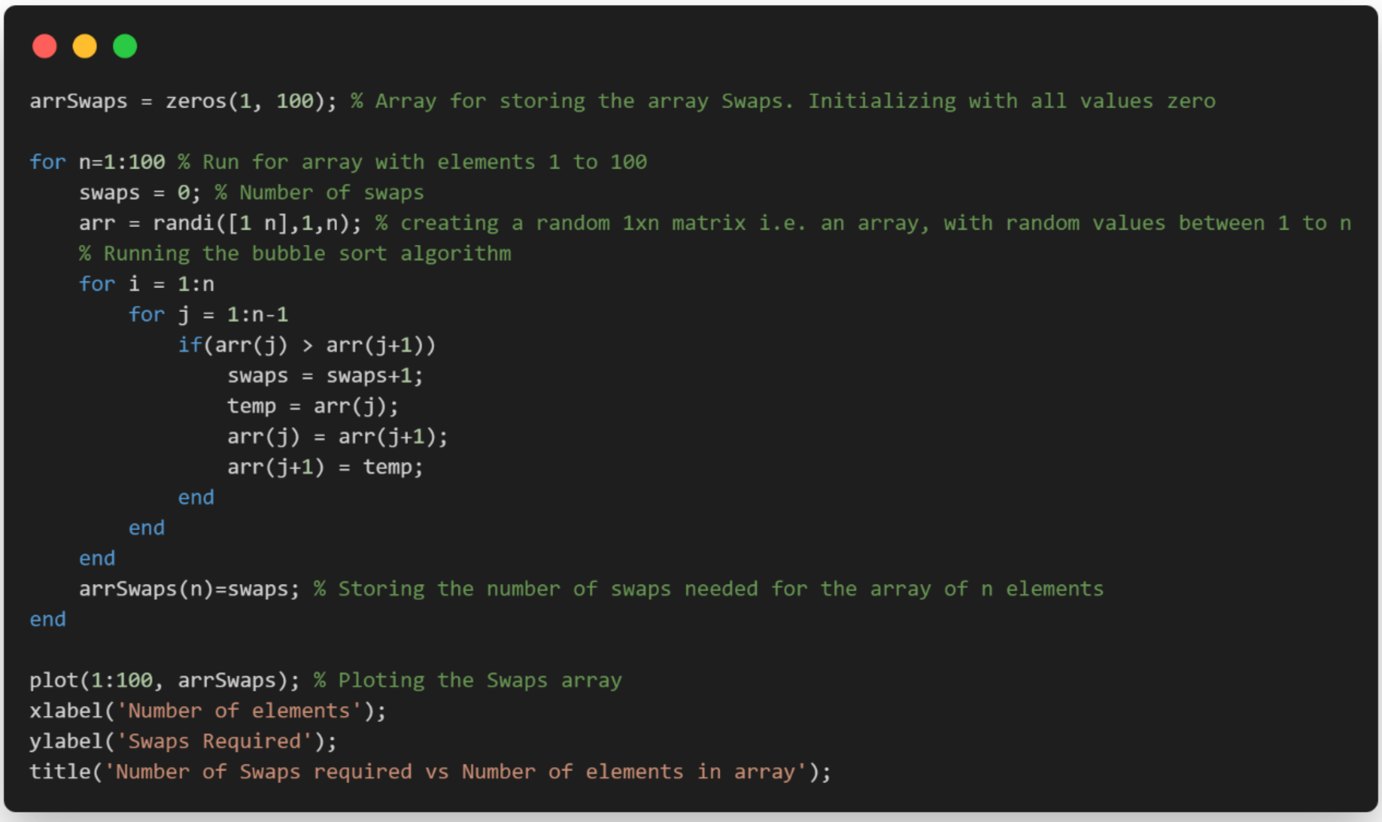
In the second iteration loop, values of the array can be looped to $n - i$, where i is the number of iterations occurred. This can lead to half the number of comparisons needed for the unoptimized bubble sort. Thus, the function of number of swaps becomes

$$f(n) = n\frac{(n-1)}{2}$$

The function can be optimized more by checking if the array is sorted, if zero swaps are made in the swaps code, then the sorting loop can be broken. However, this is not a general case and depends on worst-case or best-case scenario, so the algorithm is not further optimized.

Visualization and Analysis

100 random arrays were sorted, and the swaps required for those arrays were plotted against the number of elements in each array.



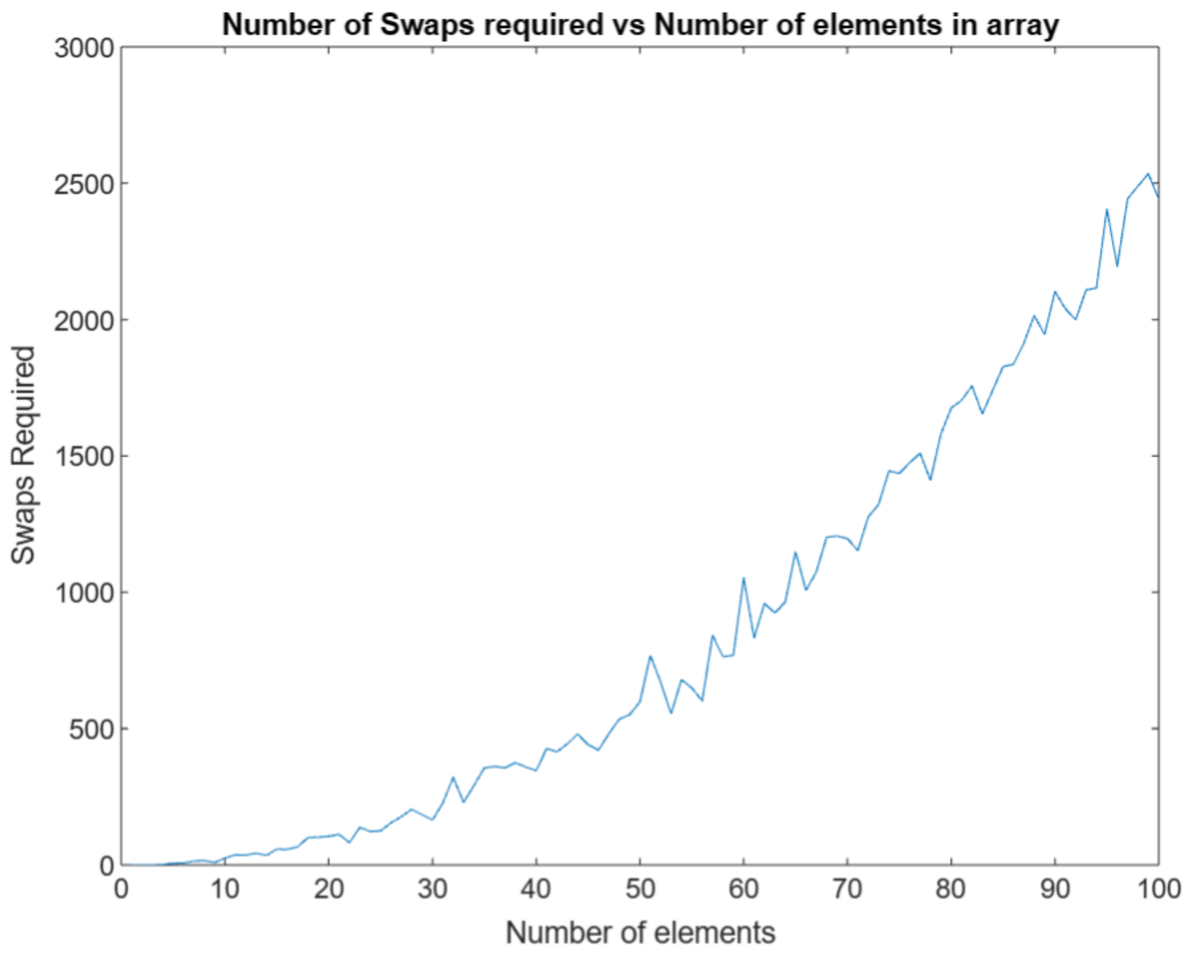
```
arrSwaps = zeros(1, 100); % Array for storing the array Swaps. Initializing with all values zero

for n=1:100 % Run for array with elements 1 to 100
    swaps = 0; % Number of swaps
    arr = randi([1 n],1,n); % creating a random 1xn matrix i.e. an array, with random values between 1 to n
    % Running the bubble sort algorithm
    for i = 1:n
        for j = 1:n-1
            if(arr(j) > arr(j+1))
                swaps = swaps+1;
                temp = arr(j);
                arr(j) = arr(j+1);
                arr(j+1) = temp;
            end
        end
    end
    arrSwaps(n)=swaps; % Storing the number of swaps needed for the array of n elements
end

plot(1:100, arrSwaps); % Ploting the Swaps array
xlabel('Number of elements');
ylabel('Swaps Required');
title('Number of Swaps required vs Number of elements in array');
```

First, the swaps array was initialized. Then 100 arrays of elements 1 to 100 were generated and sorted. The swaps for each array were noted and then plotted against the number of elements in each array.

Output:



The graph is not smooth because the arrays were randomly generated, and the function is counting the number of swaps made. The comparison function graph will be smooth because it is more general and is same for all types of arrays.

By Hand Task

Q_{6a}

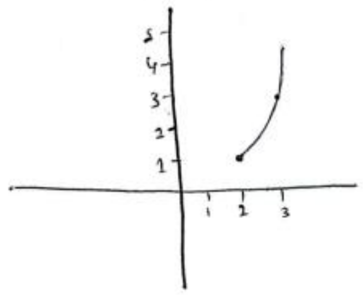
→ Finding the intersepts :

→ $n(n-1) = 0$

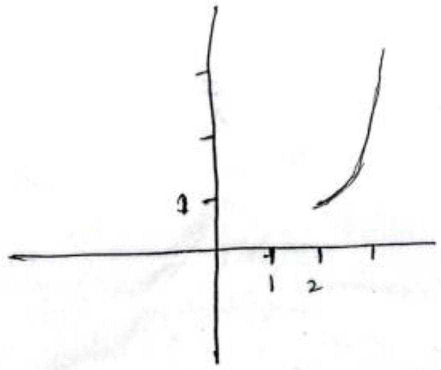
~~[illegible]~~ $n = 1$, $n = 0$ Domain is $[2, \propto)$

→ and the vertex are $\left(\frac{1}{2}, -\frac{1}{4}\right)$ are not part of domain.

The graph for $n(n-1)$ is:



→ The graph of $\frac{n(n-1)}{2}$ will be a little toward y-axis as it is divided by 2:

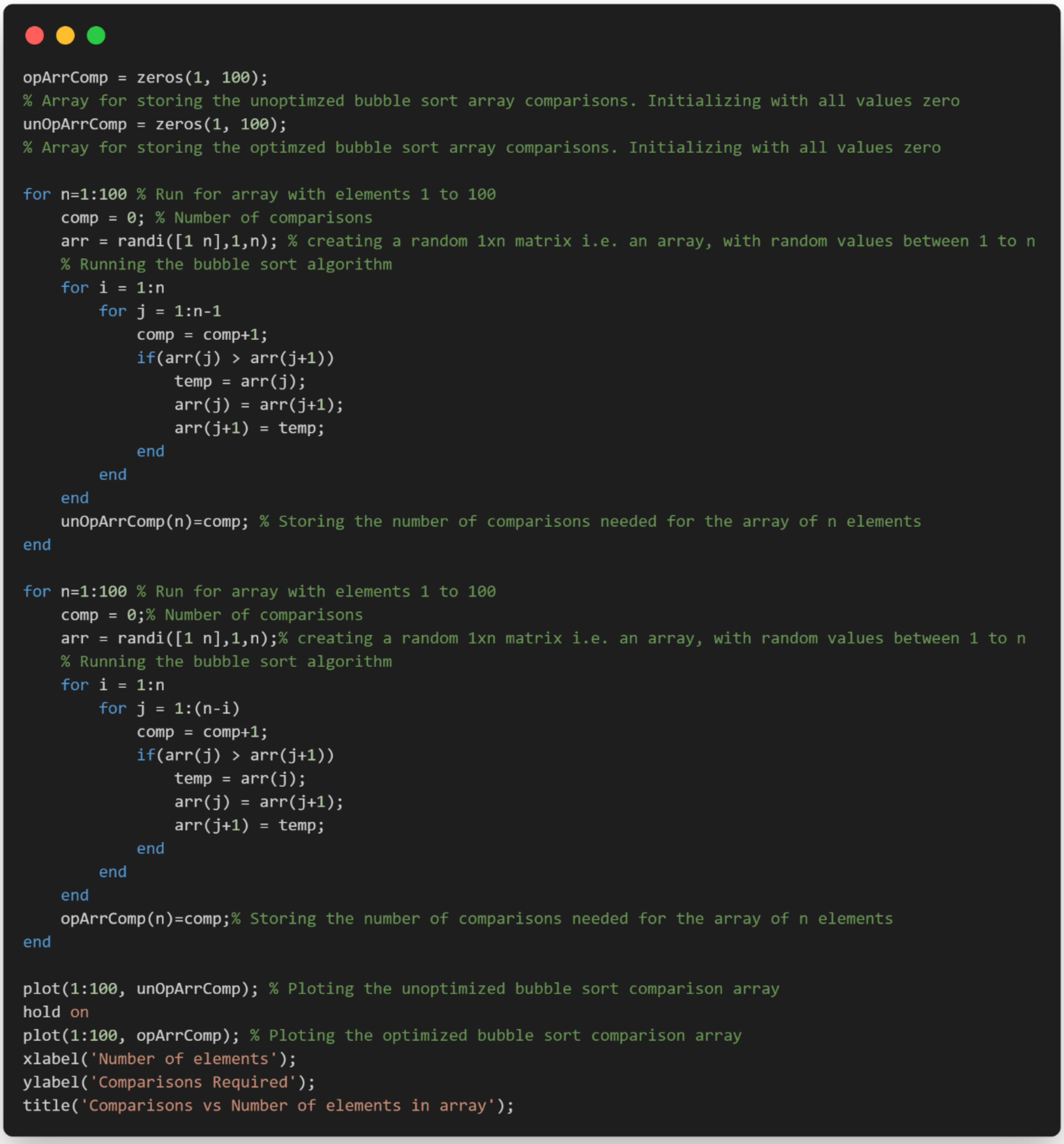


→ ~~[illegible]~~ Although both of the graphs has same y-value or starting values but the graph of $\frac{n(n-1)}{2}$ is a little tilted towards y-axis.

Comparative Analysis

MATLAB Task

The comparisons made by the optimized and unoptimized function were compared.



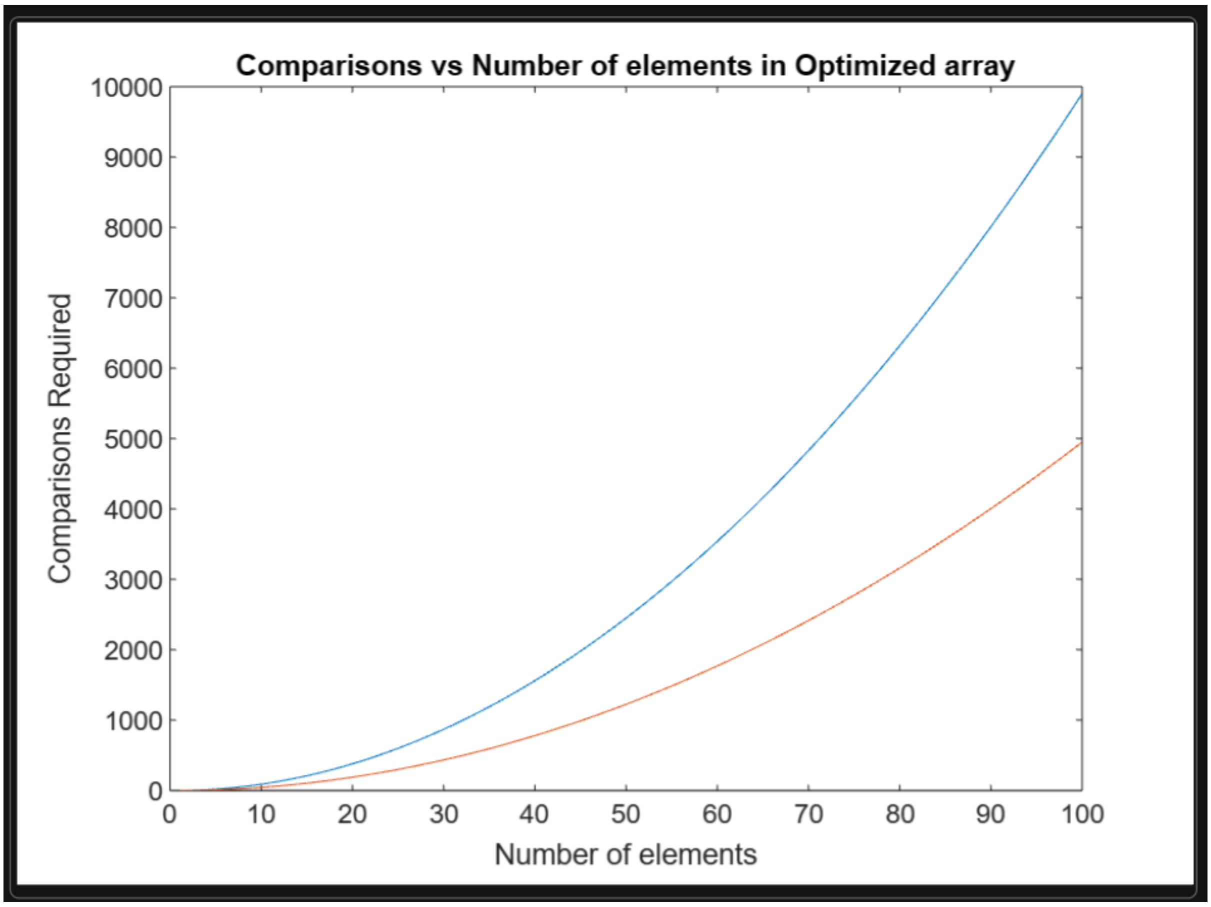
```matlab
opArrComp = zeros(1, 100);
% Array for storing the unoptimzed bubble sort array comparisons. Initializing with all values zero
unOpArrComp = zeros(1, 100);
% Array for storing the optimzed bubble sort array comparisons. Initializing with all values zero

for n=1:100 % Run for array with elements 1 to 100
    comp = 0; % Number of comparisons
    arr = randi([1 n],1,n); % creating a random 1xn matrix i.e. an array, with random values between 1 to n
    % Running the bubble sort algorithm
    for i = 1:n
        for j = 1:n-1
            comp = comp+1;
            if(arr(j) > arr(j+1))
                temp = arr(j);
                arr(j) = arr(j+1);
                arr(j+1) = temp;
            end
        end
    end
    unOpArrComp(n)=comp; % Storing the number of comparisons needed for the array of n elements
end

for n=1:100 % Run for array with elements 1 to 100
    comp = 0;% Number of comparisons
    arr = randi([1 n],1,n);% creating a random 1xn matrix i.e. an array, with random values between 1 to n
    % Running the bubble sort algorithm
    for i = 1:n
        for j = 1:(n-i)
            comp = comp+1;
            if(arr(j) > arr(j+1))
                temp = arr(j);
                arr(j) = arr(j+1);
                arr(j+1) = temp;
            end
        end
    end
    opArrComp(n)=comp;% Storing the number of comparisons needed for the array of n elements
end

plot(1:100, unOpArrComp); % Ploting the unoptimized bubble sort comparison array
hold on
plot(1:100, opArrComp); % Ploting the optimized bubble sort comparison array
xlabel('Number of elements');
ylabel('Comparisons Required');
title('Comparisons vs Number of elements in array');
```

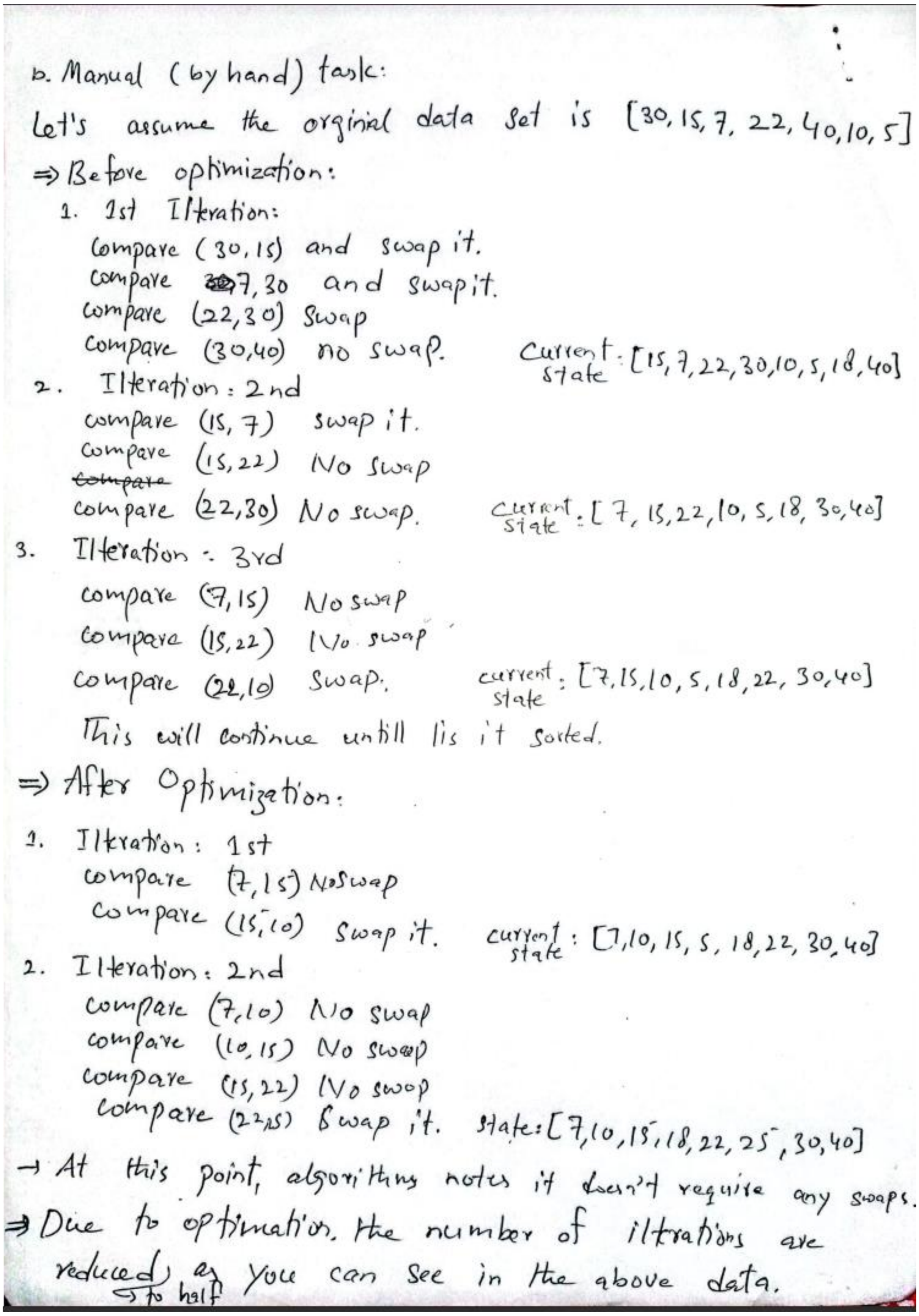
100 arrays of elements 1 to 100 with random values were generated and sorted using the unoptimized bubble sort. Similarly, 100 more arrays were generated and sorted using the optimized bubble sort. The number of comparisons were noted and plotted against the number of elements.



🟦 Unoptimized array 🟧 Optimized array

It can be inferred from the graph that the algorithm function requires half the number of comparisons after optimizing.

By Hand Task



b. Manual (by hand) task:

Let's assume the orginial data set is [30, 15, 7, 22, 40, 10, 5]

⇒ Before optimization:

1. 1st Iteration:
   compare (30, 15) and swap it.
   compare ~~30~~7, 30 and swap it.
   compare (22, 30) Swap
   compare (30, 40) no swap. Current state: [15, 7, 22, 30, 10, 5, 18, 40]
2. Iteration: 2nd
   compare (15, 7) swap it.
   compare (15, 22) No Swap
   ~~compare~~
   compare (22, 30) No swap. Current state: [7, 15, 22, 10, 5, 18, 30, 40]
3. Iteration: 3rd
   compare (7, 15) No swap
   compare (15, 22) No swap
   compare (22, 10) Swap. current state: [7, 15, 10, 5, 18, 22, 30, 40]

   This will continue untill lis it sorted.

⇒ After Optimization:

1. Iteration: 1st
   compare (7, 15) No Swap
   compare (15, 10) Swap it. current state: [7, 10, 15, 5, 18, 22, 30, 40]
2. Iteration: 2nd
   compare (7, 10) No swap
   compare (10, 15) No swap
   compare (15, 22) No swop
   compare (22, 25) Swap it. state: [7, 10, 15, 18, 22, 25, 30, 40]

→ At this point, algorithms notes it doesn't require any swaps.

⇒ Due to optimation, the number of iltrations are reduced → to half as you can see in the above data.

The results found by using MATLAB and doing it by hand are same

Conclusion

To sum up, the optimization of the Bubble Sort algorithm was accomplished effectively by means of MATLAB, calculus-based optimizations, and different algorithmic techniques. The calculus-based approach seemed to be ineffective in terms of optimization as the critical points were out of domain. The algorithmic techniques made the number of comparisons half with respect to the unoptimized algorithm. Finally, the optimized and unoptimized algorithms were compared for testing purposes.

Group Member Contributions

Ali Aan (23I-0708): Wrote the MATLAB codes. Did Q2 the MATLAB task part of Q4, Q5, Q6, Q7.

Owais Abdullah (23I-0808): Wrote the Introduction and Objectives part. Did by hand task of Q4, Q5.

Abdul Saboor (23I-3039): Wrote Understanding Time Complexity part, and conclusion. Did by hand task of Q6, Q7.

We had difficulties in understanding the MATLAB syntax, however, they were resolved after going through the MATLAB reference material provided. We also had difficulties in deriving the comparison function and understanding the time complexity. However, they were resolved by clearly understanding the MATLAB implementation of the algorithm.