122CS0067: S1

Q.1. Performance analysis of Bubble Sort

Write the program to analyse the performance of two different versions of bubble sort for randomized data sequence of integers. (i) BUBBLE SORT that terminates if the array is sorted before n-1th Pass. (ii) BUBBLE SORT that always completes the n-1th Pass.

Q.2. Performance analysis of Bubble Sort

Write the program to compare the performance of insertion sort and bubble sort for randomized data sequence of integers. Analyse their time and space complexities theoretically and then validate these complexities by running experiments with different sizes of input data.

Q.3. Average case analysis for Sorting Algorithms

For each of the data formats: random, reverse ordered, and nearly sorted, run your program say **SORT_TEST** for all combinations of sorting algorithms and data sizes and complete each of the following tables. When you have completed the tables, analyse your data and determine the asymptotic behaviour of each of the sorting algorithms for each of the data types *(i)* ***Random data, (ii) Reverse Ordered Data, (iii) Almost Sorted Data*** and **(iv)** ***Highly Repetitive Data.*** Select the suitable no of elements for the analysis that supports your program.

Q.4. Graph Representation Conversion

Write MATLAB functions to convert between different graph representations, such as adjacency matrices, adjacency lists, and edge lists.

Q.5. Coin Tossing

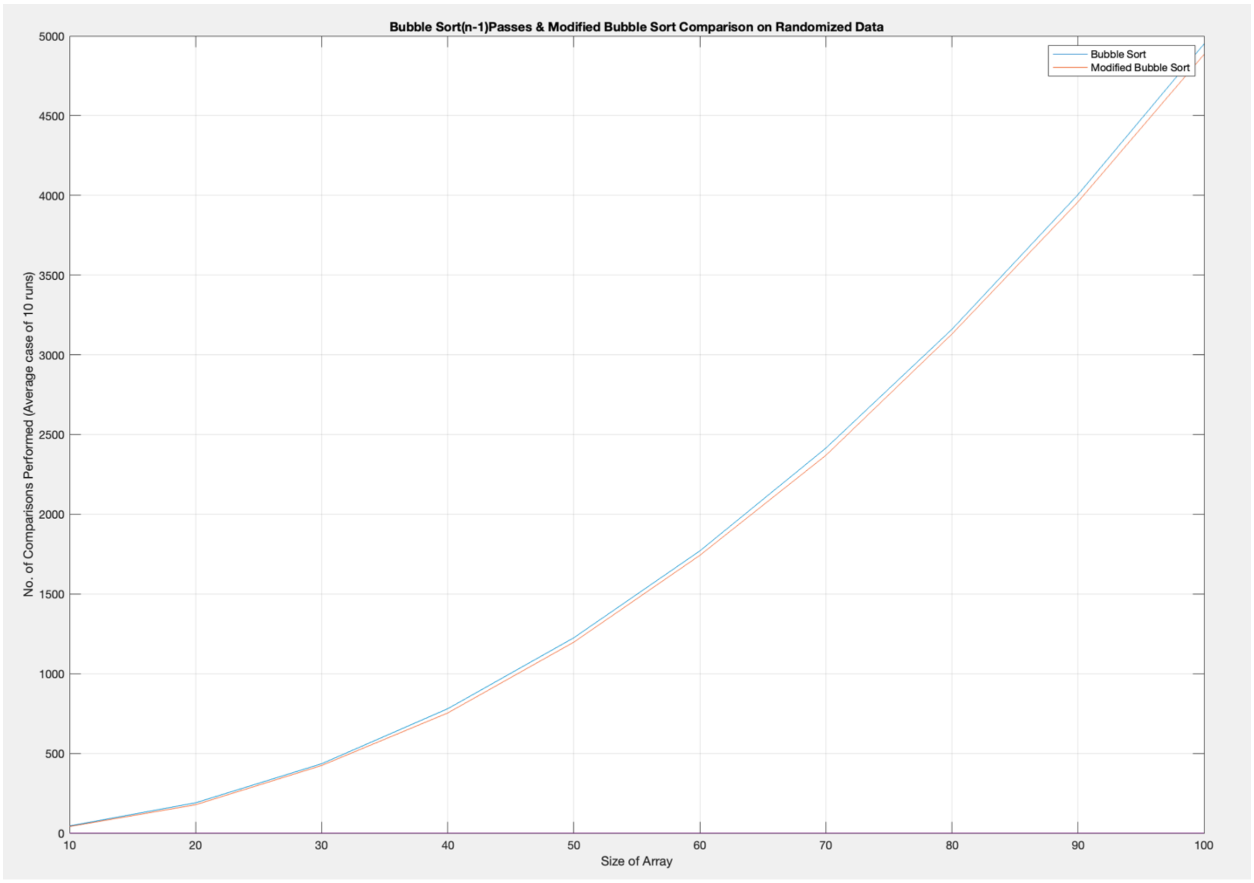
Through the simulation, show that probability of getting HEAD by tossing a fair coin is about 0.5. Write your observation from the simulation run.

Upload your report [Source Codes + Simulation results +Observations] as a single pdf file with name as Roll No A1 to the specific assignment in MS Team

122CS0067: S1

Q.1. Performance analysis of Bubble Sort

Write the program to analyse the performance of two different versions of bubble sort for randomized data sequence of integers. (i) BUBBLE SORT that terminates if the array is sorted before n-1th Pass. (ii) BUBBLE SORT that always completes the n-1th Pass.



<u>Analysis:</u> *On random data, it is found that on average the bubble sort that terminates if the array is sorted before the n-1th always performs better, though marginally. The margin very slightly increases on increase of data size, but it is still a very inefficient algorithm compared to other sorting algorithms we shall use in the subsequent questions.*

122CS0067: S1

Q.2. Performance analysis of Bubble Sort

Write the program to compare the performance of insertion sort and bubble sort for randomized data sequence of integers. Analyse their time and space complexities theoretically and then validate these complexities by running experiments with different sizes of input data.

Source Code:

%Function Bubble Sort Unmodified

```
function cp = bs1(a,size)
cp=0;
for i=1:size-1
    for j=1:size-i
        cp=cp+1;
        if a(j)>a(j+1)
            temp=a(j);
            a(j)=a(j+1);
            a(j+1)=temp;
        end
    end
end
```

%Function Bubble Sort Modified

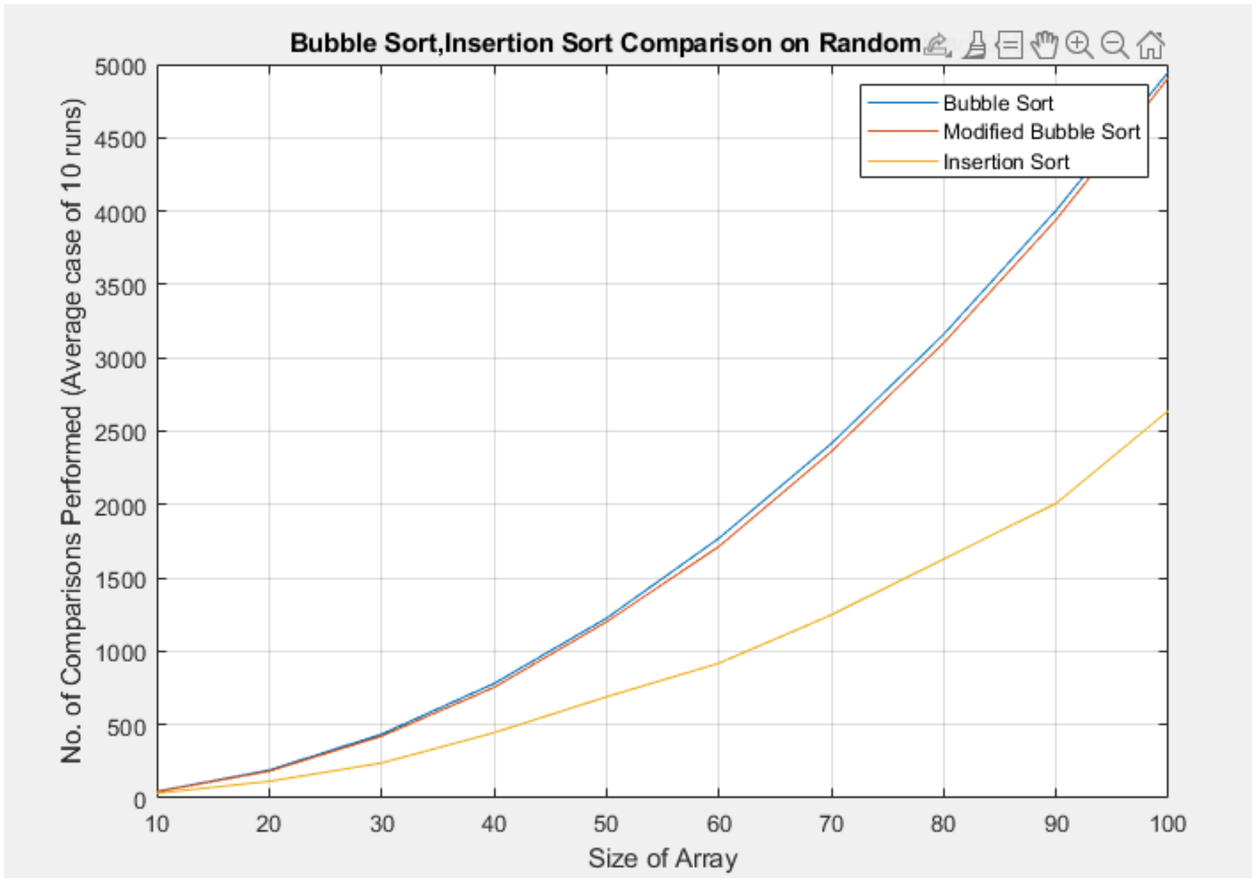
```
function cp = bs2(a,size)
cp=0;
for i=1:size-1
    flag=0;
    for j=1:size-i
        cp=cp+1;
        if a(j)>a(j+1)
            flag=1;
            temp=a(j);
            a(j)=a(j+1);
            a(j+1)=temp;
        end
    end
    if flag~=1
        break;
    end
end
```

122CS0067: S1

```
% Insertion Sort Routine
function cp = is1(a,size)
cp=0;
for i=2:size
    j=i-1;
    cp=cp+1;
    while(j>=1 && a(j)>a(j+1))
        cp=cp+1;
        temp=a(j);
        a(j)=a(j+1);
        a(j+1)=temp;
        j=j-1;
    end
end
```

%Main Program

```
clear all;
%Bubble sort comparison
sim_size=100;
step=10;
runs=10;
xne=zeros(1,sim_size/step);
ync=zeros(1,sim_size/step);
ync2=zeros(1,sim_size/step);
ync3=zeros(1,sim_size/step);
for n=1:runs
    k=1;
for i=10:step:sim_size
    arr=round(rand(1,i)*100);
    arr2=arr;
    arr3=arr;
    xne(k)=i;
    ync(k)=ync(k)+bs1(arr,i);
    ync2(k)=ync2(k)+bs2(arr2,i);
    ync3(k)=ync3(k)+is1(arr3,i);
    k=k+1;
end
end

plot(xne,ync/runs);
hold on;
plot(xne,ync2/runs);
plot(xne,ync3/runs);
title("Bubble Sort,Insertion Sort Comparison on Randomized Data")
xlabel("Size of Array")
ylabel("No. of Comparisons Performed (Average case of 10 runs)")
legend("Bubble Sort","Modified Bubble Sort","Insertion Sort")
grid on
```



<u>Analysis:</u> *On random data, insertion sort significantly outperforms bubble sort, despite having the same O(n^2) time complexity. This just shows that insertion sort algorithm is more superior and efficient than the bubble sort algorithm.*

Q.3. Average case analysis for Sorting Algorithms

For each of the data formats: random, reverse ordered, and nearly sorted, run your program say **SORT_TEST** for all combinations of sorting algorithms and data sizes and complete each of the following tables. When you have completed the tables, analyse your data and determine the asymptotic behaviour of each of the sorting algorithms for each of the data types *(i)* ***Random data, (ii) Reverse Ordered Data, (iii) Almost Sorted Data*** and **(iv)** ***Highly Repetitive Data.*** Select the suitable no of elements for the analysis that supports your program.

(i). Random Data

*We used the following algorithms and compared them: * Bubble sort unmodified, *Bubble sort modified, *Insertion Sort, *Merge sort, *Quicksort*

The main program routine is same as in the previous instance, the Merge sort and Quicksort routines are described below:

Source Code:

%Merge Sort Function

```
function [cp,sorted] = merge_sort(a,low,high,cp)
%  Merge_sort routine
cp=cp+1;
if (high==low)
     sorted=[a(low)];
     return;
end

middle=floor((high+low)/2);
[cp,left]=merge_sort(a,low,middle,cp);
[cp,right]=merge_sort(a,middle+1,high,cp);
[cp,sorted]=merge_routine(left,right,cp);
cp=cp+3;
return;
end
```

122CS0067: S1

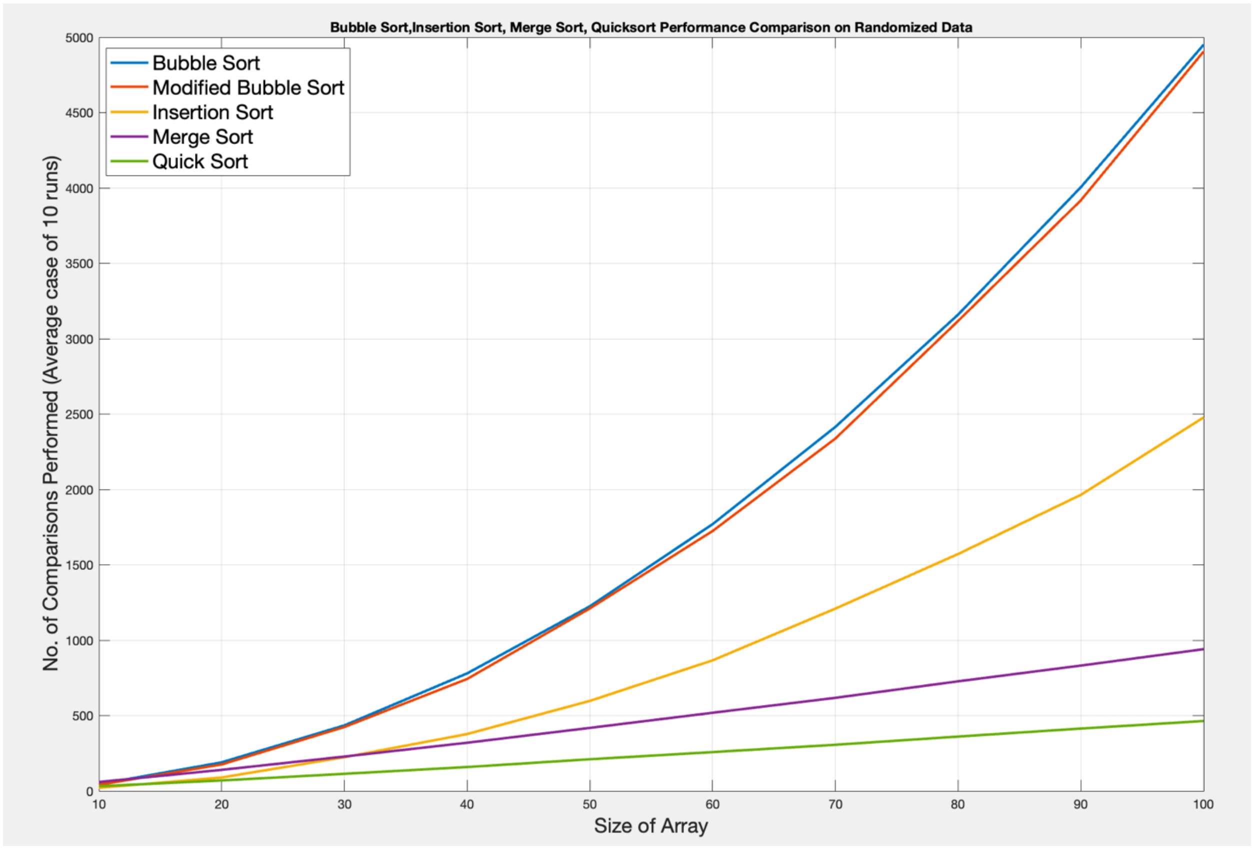
```
function [cp,sorted_arr] = merge_routine(a,b,cp)
%The Merge Function called within Merge Sort
l1=length(a);
l2=length(b);

i=1;j=1;
sorted_arr=[];
cp=cp+1;
while(i<=l1 & j<=l2)
    cp=cp+2;
    if(a(i)<b(j))
        sorted_arr=[sorted_arr,a(i)];
        i=i+1;
    else
        sorted_arr=[sorted_arr,b(j)];
        j=j+1;
    end
end
sorted_arr=[sorted_arr,a(i:end),b(j:end)];
return;
end
```

%Quicksort Routine

```
function  cp= qs(a, lb, ub, cp)
    if lb < ub
        cp = cp + 1;
        [a, loc, cp] = partition(a, lb, ub, cp);
        cp = qs(a, lb, loc-1, cp);
        cp = qs(a, loc+1, ub, cp);

    end
    cp = cp + 1;
    c = a;
end

function [a, d, cp] = partition(a, lb, ub, cp)
    pivot = a(lb);
    start = lb + 1;
    last = ub;

    while start <= last
        while start <= ub && a(start) <= pivot
            start = start + 1;
            cp = cp + 1;
        end
        cp = cp + 1;
        while last >= lb + 1 && a(last) > pivot
            last = last - 1;
            cp = cp + 1;
        end
        cp = cp + 1;
        if start < last
            temp = a(start);
            a(start) = a(last);
            a(last) = temp;

        end

    end
    a(lb) = a(last);
```

122CS0067: S1

```
    a(last) = pivot;
    d = last;
end
```



Analysis:

*As expected, the algorithms with average case complexities of O(*n^2*) {Bubble, Insertion Sort) have inferior performance against the O(*nlogn*) {Merge, Quick Sort} sorting algorithms.*

For this particular implementation, it is found that insertion sort performs somewhat better than merge sort when working on smaller arrays. The average crossover point is about array size (32).

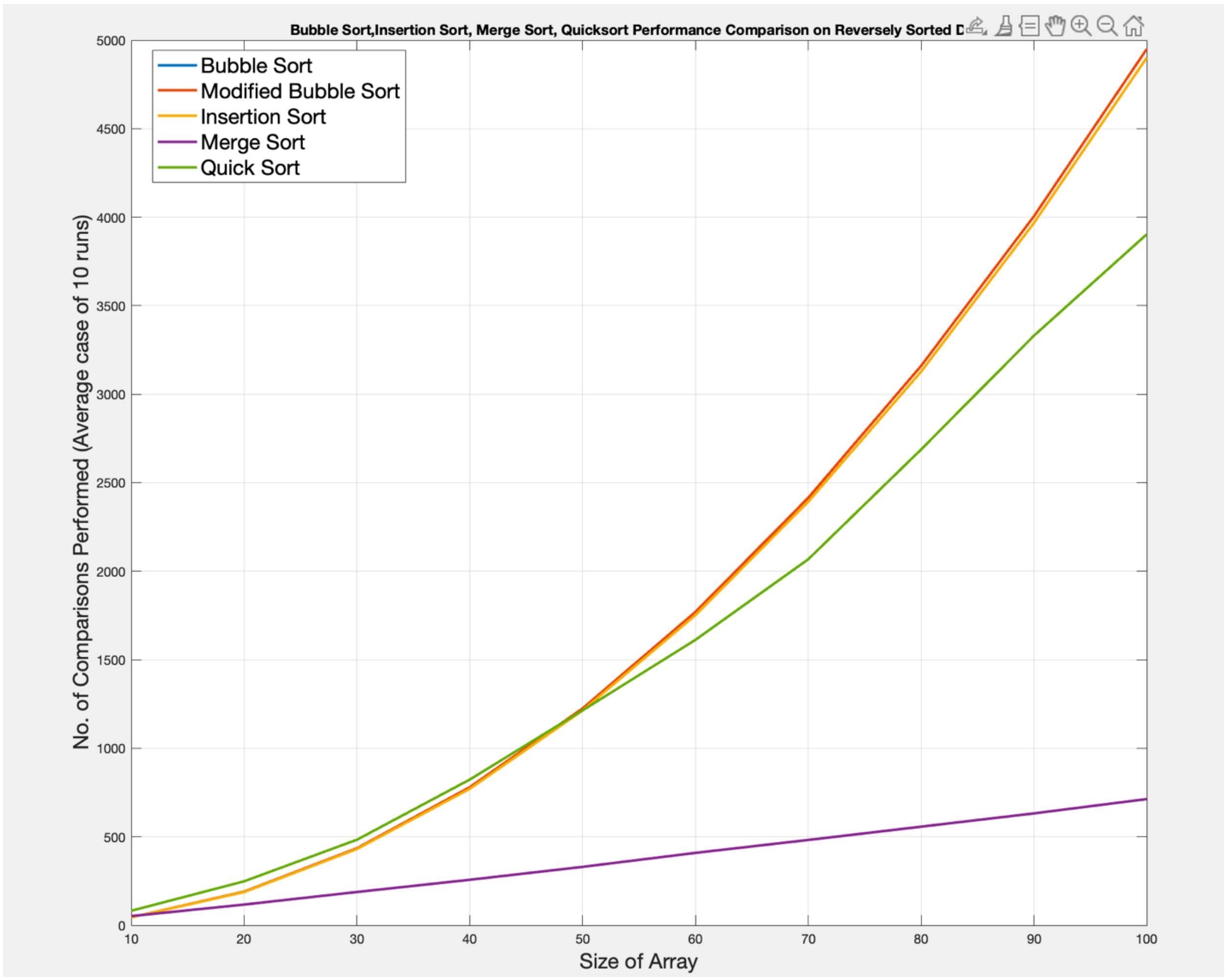
122CS0067: S1

(ii) Reverse ordered data

To generate reverse ordered data, we generated pseudo random arrays and sorted them in descending order using MATLAB's inbuilt sorting function.

Code that generated the input arrays:

```
arr=sort(round(rand(1,i)*100),'descend');
```



Analysis:

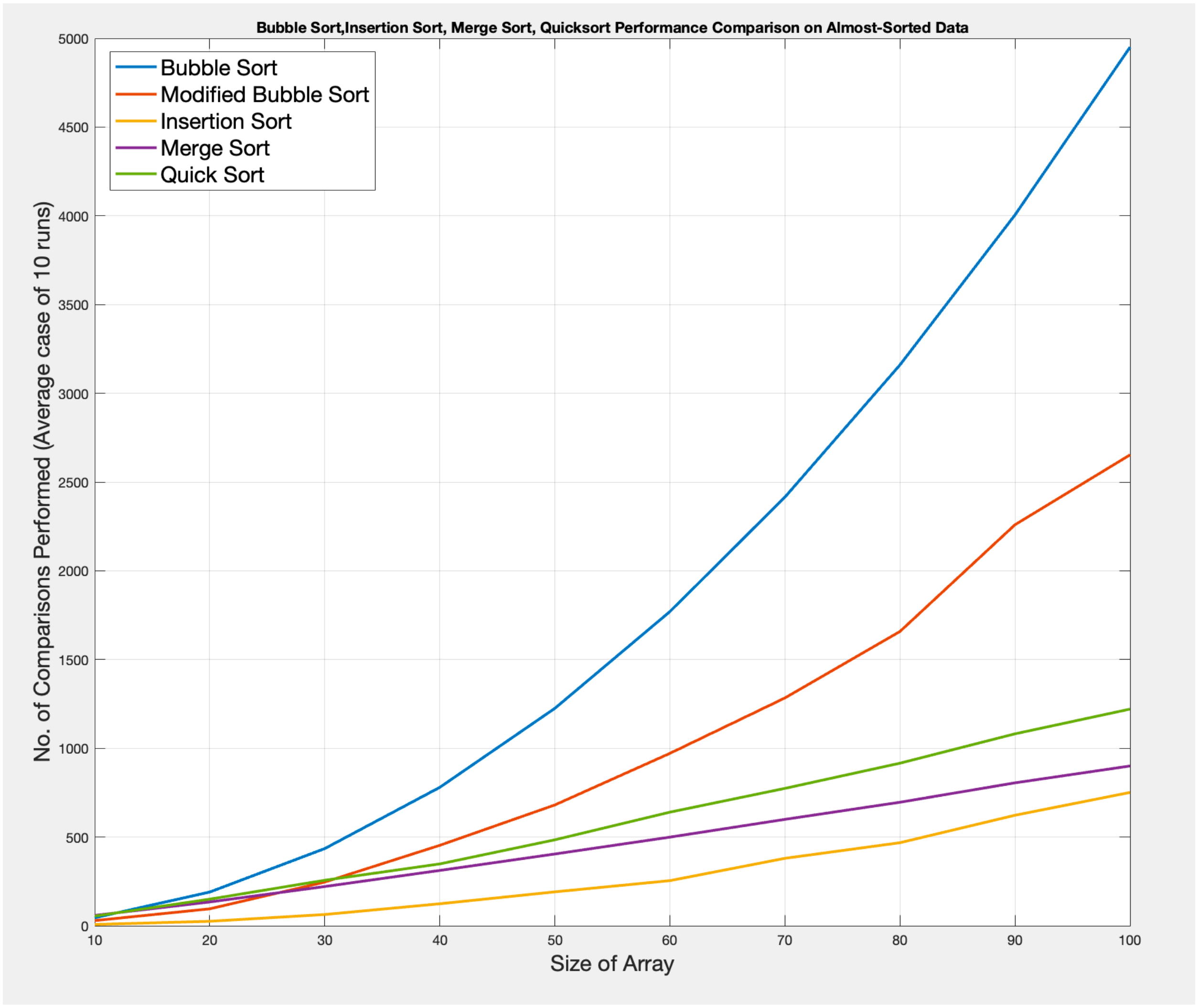
On reversely sorted data, bubble, insertion sort both encounter their worst-case scenario and have performance of o(n^2). For this particular implementation of quicksort, which always selects the leftmost element in the subarray as the pivot, it is also the worst-case scenario. It too has a performance of O(n^2). This can be taken care of to improve performance in this case by randomizing the selection of the pivot element. Merge sort has a performance of the order O(nlogn).

(iii)Almost Sorted Data

To generate almost sorted data, we generated pseudo random arrays, sorted them and then introduced minor perturbations in them using some of MATLAB's inbuilt functions.

Code that generated the input arrays:

```
function almost_sorted = almost_sorted(n,perturb)
%Generates almost sorted 1Xn arrays

mat1=sort(round(rand(1,n)*10));
mat2=randi([-perturb +perturb],1,n);
almost_sorted=mat1+mat2;
end
```



122CS0067: S1

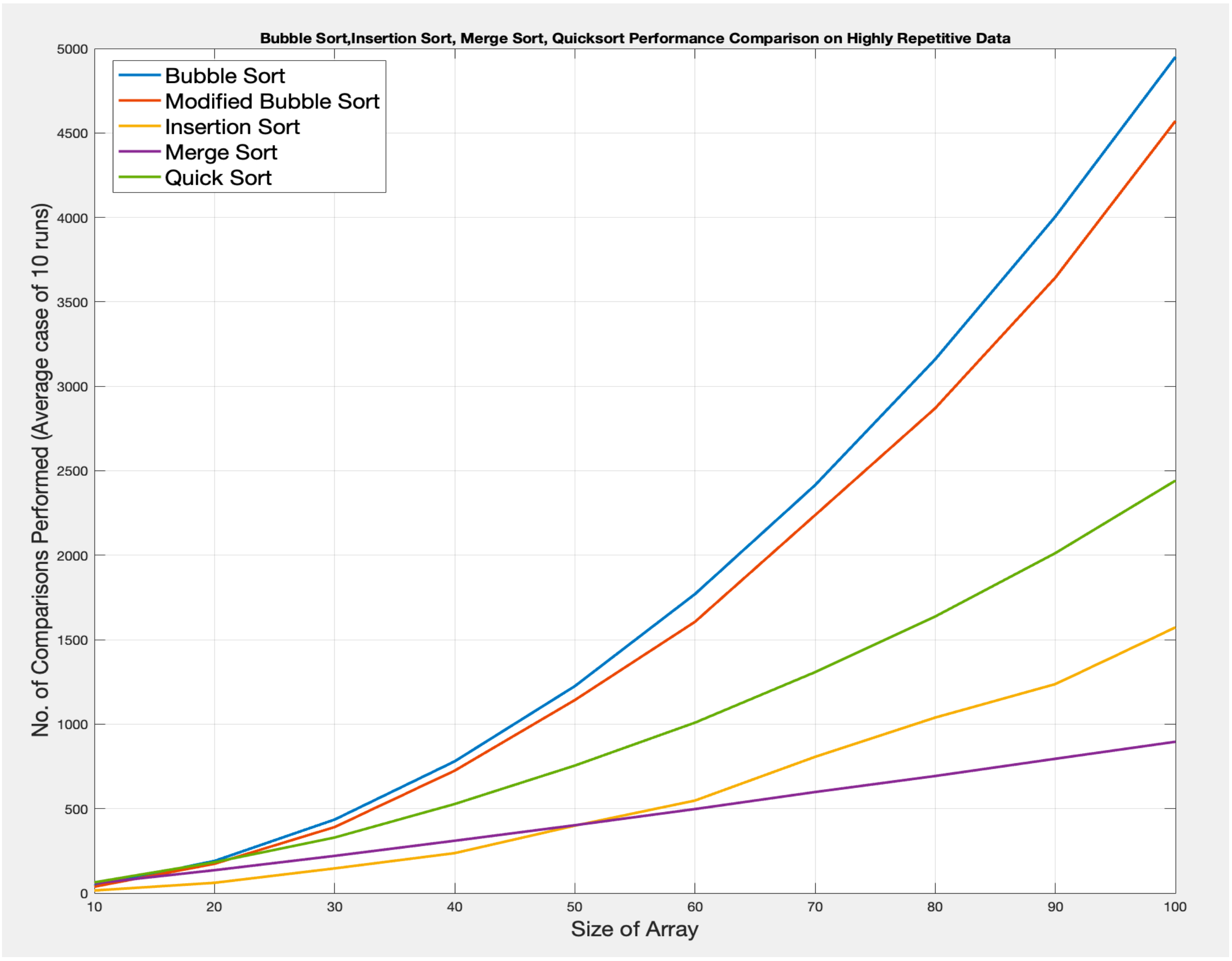
Analysis:

Insertion Sort outshines all other sorting algorithms on receiving almost sorted data. The modified bubble sort performs better than its average case performance. The other algorithms almost follow their average case performance.

(iv) Highly Repetitive Data

To generate highly repetitive data we used some of MATLAB's inbuilt functions. The code is given below:

```
function arr=highly_rep(n)
arraySize = n;
baseData = round(rand()*100); % Base random value
arr = repmat(baseData, 1, arraySize);
% Adding random noise
arr = arr+round(rand(1,n))-round(rand(1,n));
end
```



Analysis:

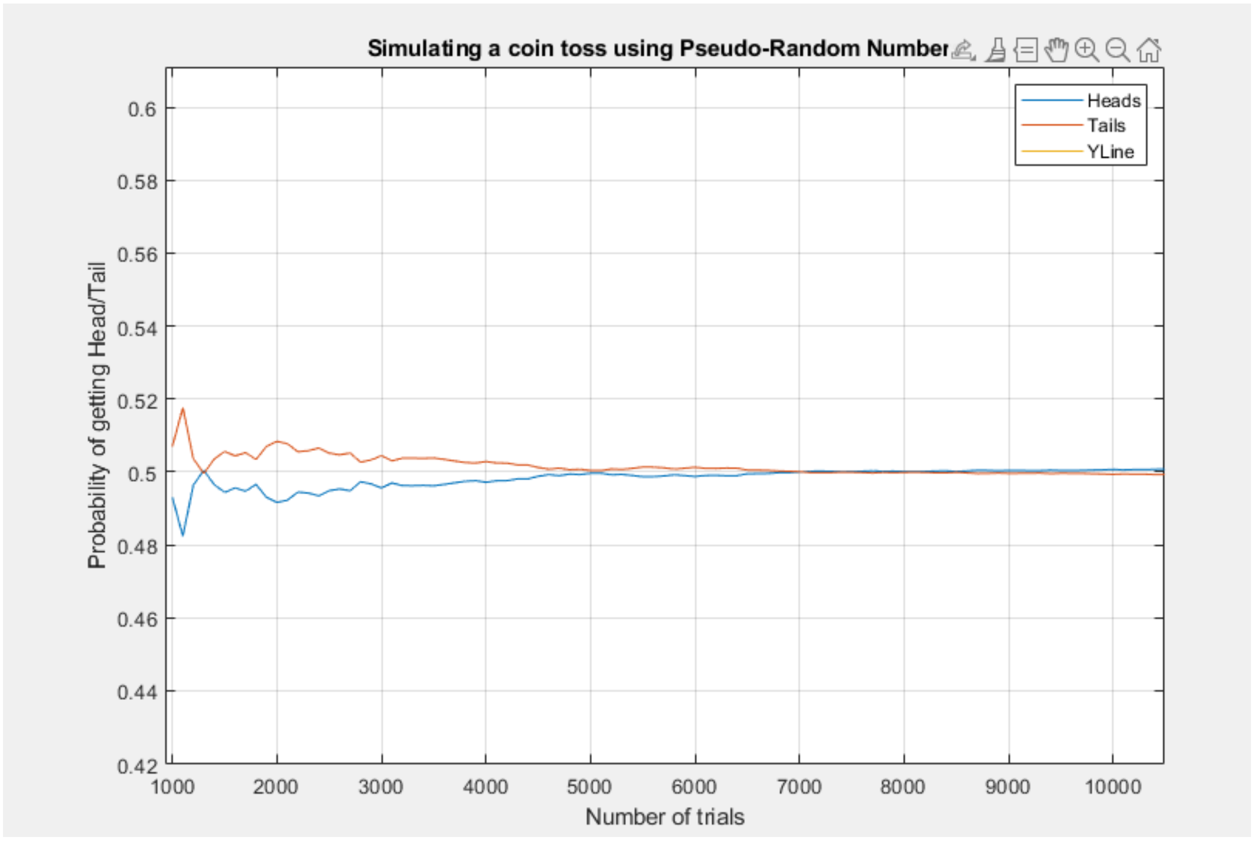
Insertion sort does somewhat better than its average case performance, quick sort does somewhat worse, and the other algorithms perform in the same order of their average case complexities for highly repetitive data as per this simulation.

Q.5. Coin Tossing

Through the simulation, show that probability of getting HEAD by tossing a fair coin is about 0.5. Write your observation from the simulation run.

```
clear  all;
%Coin toss using rand function
% No. of Coin Tosses
n=30000;
prob_h=zeros(1,10);
prob_t=zeros(1,10);
trials=zeros(1,10);
tt=0;
res=0;
%randi([0,1])

k=1;
for i=1000:100:n
    for j=1:i
        res=res+round(rand(1,1));
    end
    tt=tt+i;
    prob_h(k)=(res/tt);
    prob_t(k)=(1-(res/tt));

    trails(k)=i;
    k=k+1;
end

plot(trails,prob_h)
hold on
plot(trails,prob_t)
xlabel("Number of trials")
ylabel("Probability of getting Head/Tail")
title("Simulating a coin toss using Pseudo-Random Numbers");
plot(0.5,zeros(1,10))
grid on;
legend("Heads","Tails","YLine");
```

122CS0067: S1


Simulating a coin toss using Pseudo-Random Number
Heads
Tails
YLine
Probability of getting Head/Tail
Number of trials