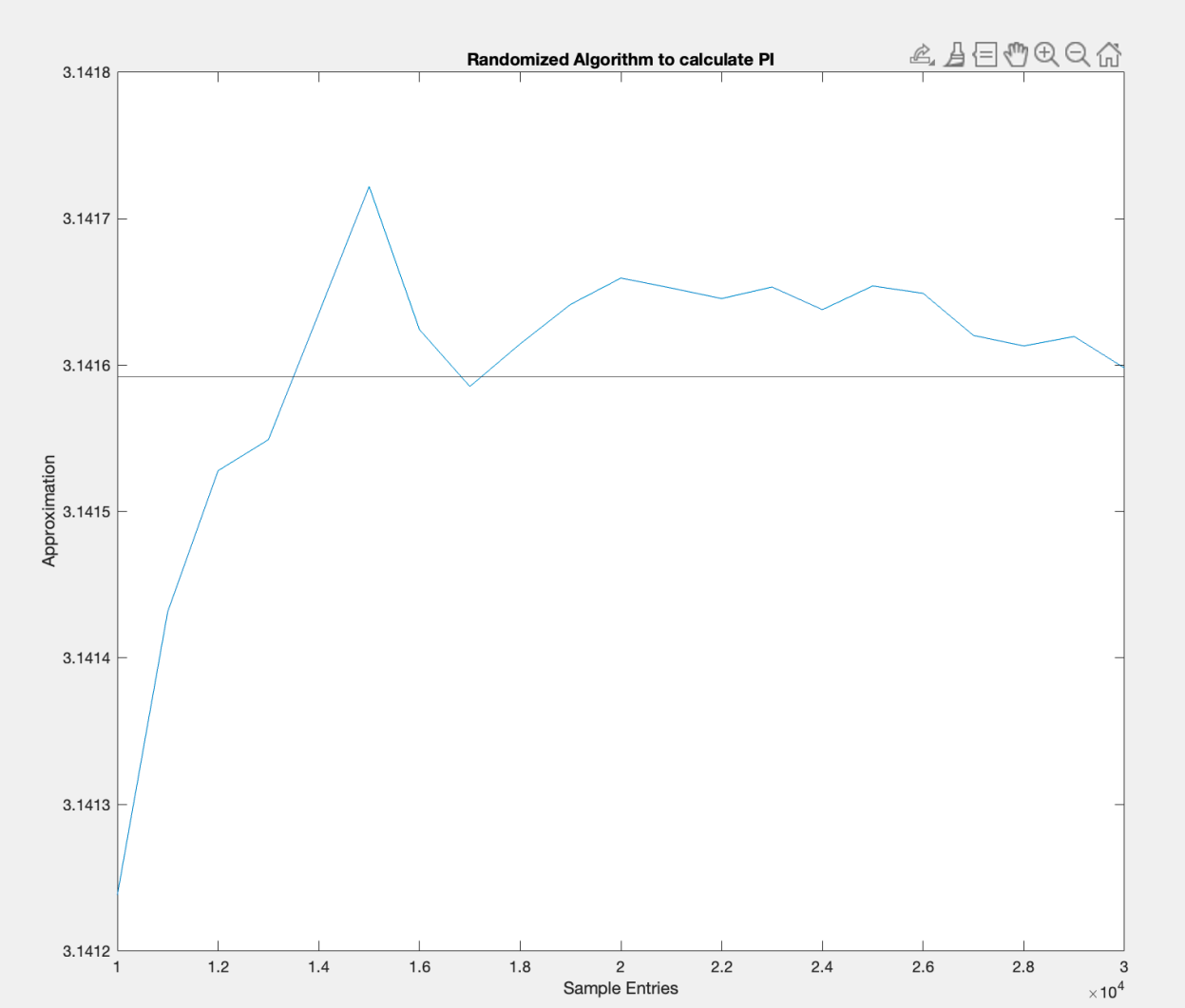
Q.06. Compute value of Π

Compute Π using randomized algorithm

% Source Code

```
sim_size=100;
xne=zeros(1,sim_size);
yne=zeros(1,sim_size+1);
base=10000;
step=1000;
k=1;
tt=0;
sum=0;
for i=base:step:(base+step*sim_size)
    xne(k)=i;
    for j=1:i
        sum=sum+pi_approx(j);
    end
    tt=tt+i;
    yne(k)=yne(k)+(sum/tt);
    k=k+1;
end

plot(xne,yne);
title("Randomized Algorithm to calculate PI")
ylabel("Approximation")
xlabel("Sample Entries")
yline(3.141592);

function pi_app=pi_approx(samples)
x=rand(1,samples);
y=rand(1,samples);
k=sum(x.^2+y.^2<=1);
pi_app=4*k/samples;
end
```

122CS0067


Randomized Algorithm to calculate PI
Approximation
Sample Entries
3.1418
3.1417
3.1416
3.1415
3.1414
3.1413
3.1412
1
1.2
1.4
1.6
1.8
2
2.2
2.4
2.6
2.8
3
×10^4

122CS0067

Q.07. NUMERICAL INTEGRATION

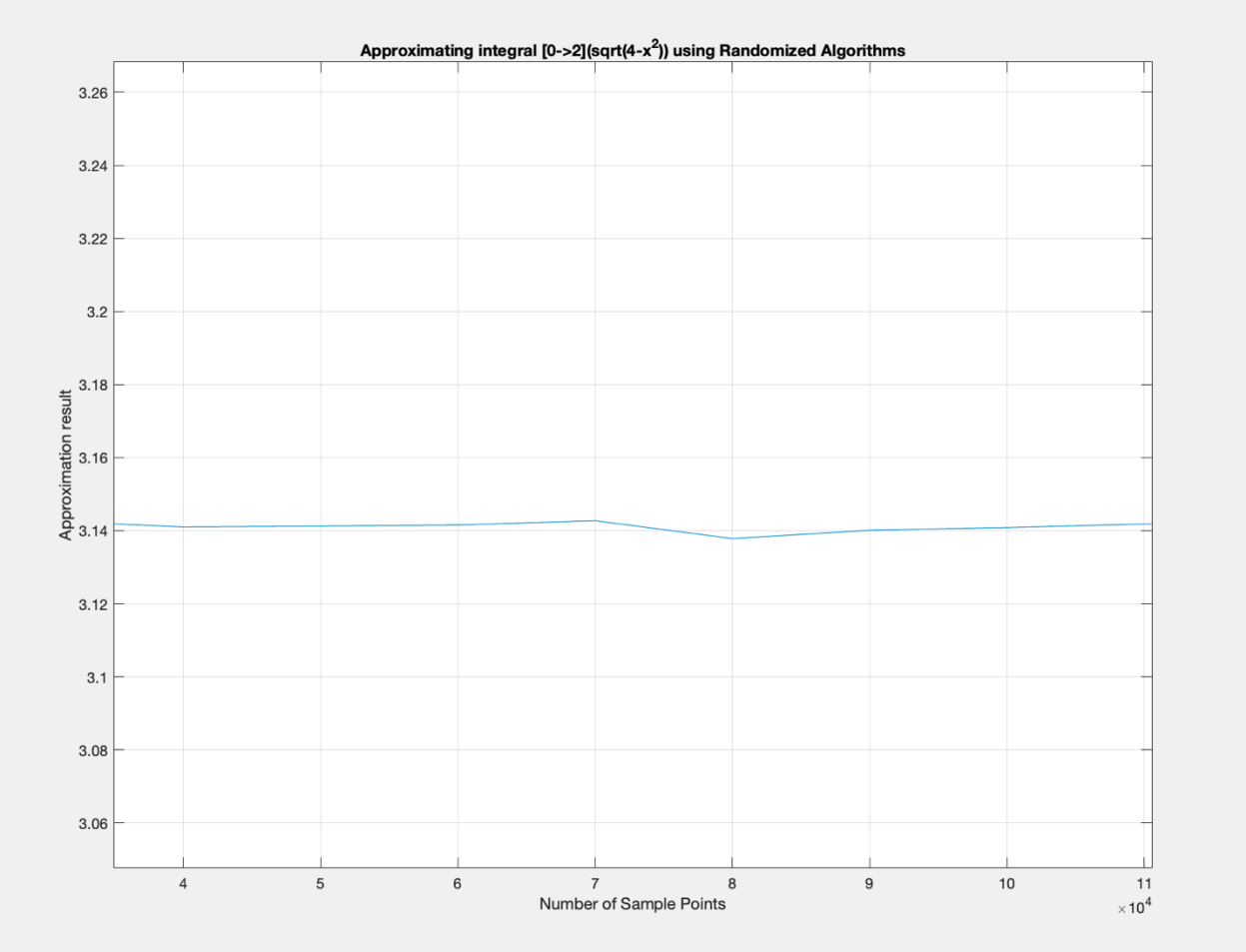
Write a program that computes the value of the following integral using randomized algorithm.

$$\int_0^2 \sqrt{4 - x^2}\, dx$$

%Source Code

```
sim_size=100;
xne=zeros(1,sim_size);
yne=zeros(1,sim_size+1);
base=10000;
step=10000;
k=1;
for j=1:5
    k=1;
for i=base:stepbase+step*sim_size)
    xne(k)=i;
    yne(k)=yne(k)+monte_integ(i);
    k=k+1;
end
end
plot(xne,yne/5);
grid on;
xlabel('Number of Sample Points');
ylabel('Approximation result');
title('Approximating integral [0->2](sqrt(4-x^2)) using Randomized
Algorithms');

function approx=monte_integ(samples)
%integral 0->2(sqrt(4-x^2))
%f calculates the function value at x;
f=@(x)sqrt(4-x.^2);
%matrix x of size<1,samples> stores random values between 0 to 2 for our
%problem
x=2*rand(1,samples);
%sample stores the curve values for these individual points
sample=f(x);
%our approximation is nothing but the average area under the curve
approx=2*mean(sample);
end
```

122CS0067



Q. 08. PRIMALITY TESTING

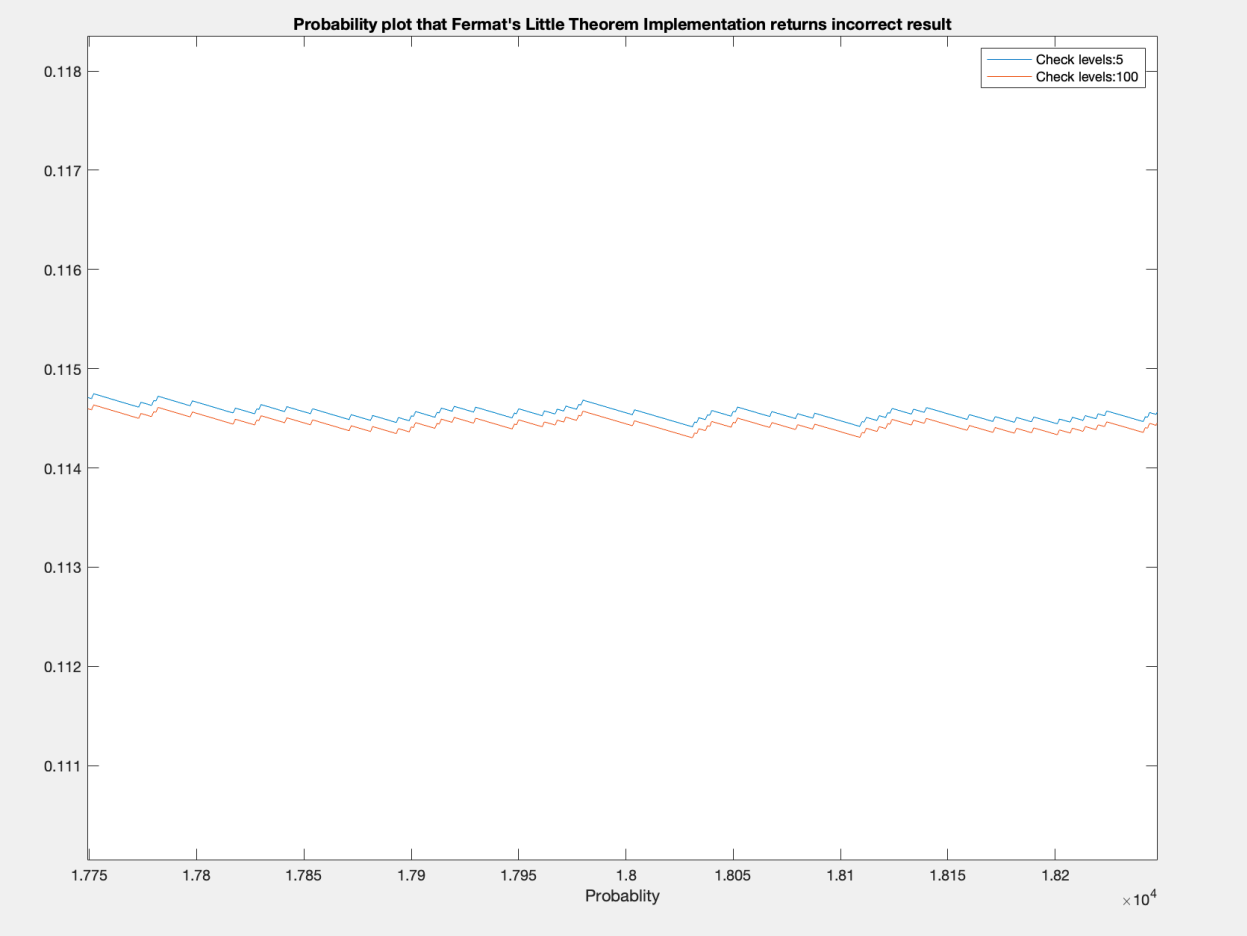
Write a program that test a number to be prime or not. Perform an analysis to compute the correctness?

```
clear all;
% Checking correctness of our primality testing implementation
k=1;
count=0;count3=0;
tt=0;
p=zeros(1,100);

p3=zeros(1,100);
xne=zeros(1,100);
for j=10:30000
    count=count+(fermat_tet(j,5)~=brutecheckprime(j));
```

122CS0067

```
    count3=count3+(fermat_tet(j,100)~=brutecheckprime(j));
    tt=tt+1;
    p(k)=count/tt;

    p3(k)=count3/tt;
    xne(k)=tt;
    k=k+1;
end
%Plot returns probablity that our implementation returns incorrect output
%versus sample size

plot(xne,p)
hold on
title("Probability plot that Fermat's Little Theorem Implementation returns
incorrect result")
xlabel("Probablity")

plot(xne,p3)
legend("Check levels:5","Check levels:100")


function prime=fermat_tet(n,runs)
for i=1:runs
a=randi([1,n-1]);
f=modpower(a,n-1,n);
prime=1;
if f~=1
    prime=0;
    return;
end
end
end
function result = modpower(base, exponent, modulus)
    % Efficient modular exponentiation using square-and-multiply algorithm
    result = 1;
    base = mod(base, modulus);

    while exponent > 0
        if mod(exponent, 2) == 1
            result = mod(result * base, modulus);
        end
        exponent = floor(exponent / 2);
        base = mod(base^2, modulus);
    end
end
function res=brutecheckprime(n)
     res=1;
     for i=1:floor(sqrt(n))
         if mod(n,i)==0
             res=0;
             return;
         end
     end
end
```

122CS0067



Q. 09. MAJORITY ELEMENT

Write a program that FINDS majority element from a linear array using randomized algorithm. Show that probability of missing majority element is 0.00097.

Q. 10. Randomized Quick Sort

Compare the performance of randomized Quicksort with conventional quick sort for random input data stream.

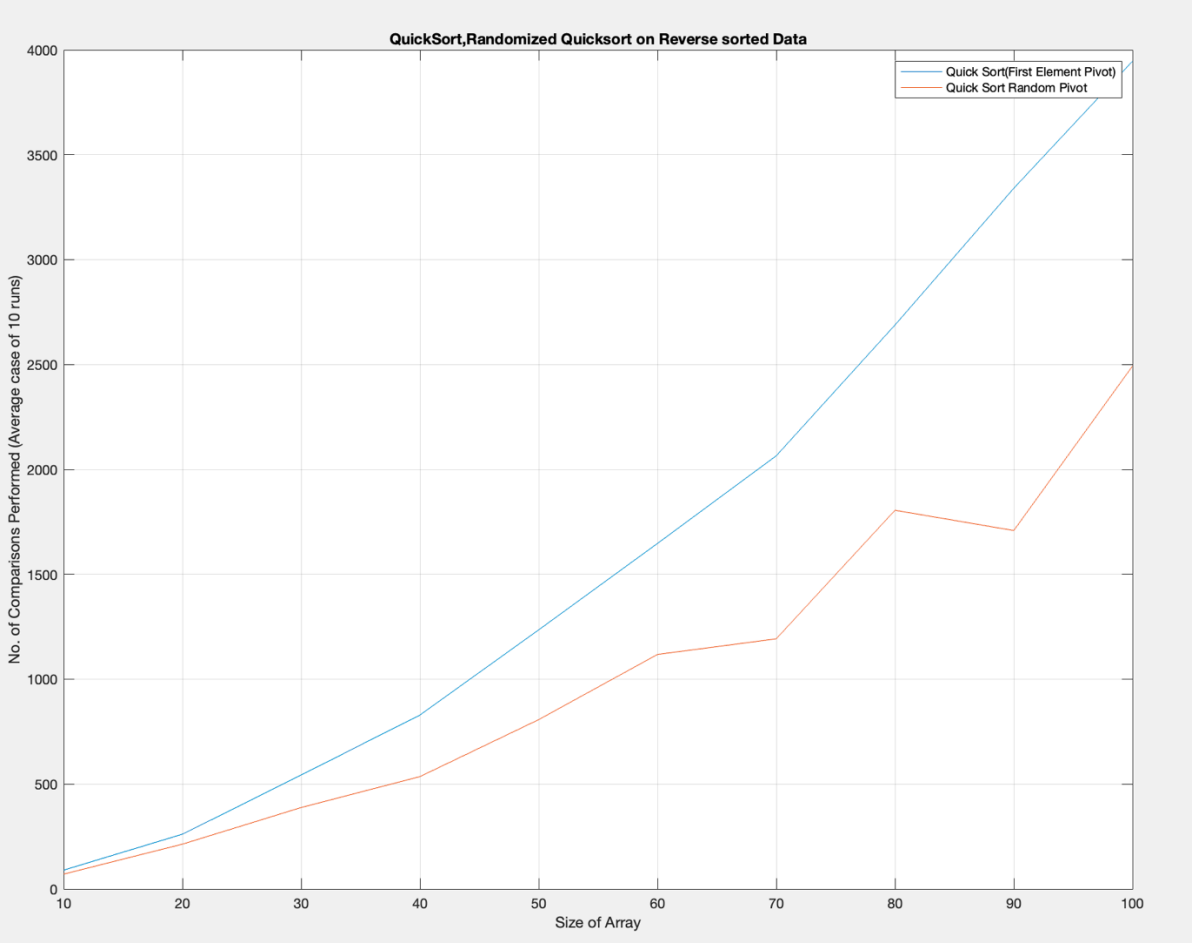
%Source Code

```
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);

for n=1:runs
    k=1;
    for i=10:step:sim_size
        arr=sort(round(rand(1,i)*100),'descend');
        %arr=almost_sorted(i,3);
        arr2=arr;
        xne(k)=i;
        ync(k)=ync(k)+qs(arr,1,i,0);
        ync2(k)=ync2(k)+qs_rand(arr2,1,i,0);
        k=k+1;
    end
end
plot(xne,ync/runs);
hold on;
plot(xne,ync2/runs);
title("QuickSort,Randomized Quicksort on Reverse sorted Data")
xlabel("Size of Array")
ylabel("No. of Comparisons Performed (Average case of 10 runs)")
legend("Quick Sort(First Element Pivot)","Quick Sort Random Pivot")
grid on
%--------------------------------
%Regular Quick Sort Implementation
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;
end
function cp= qs_rand(a, lb, ub, cp)
if lb < ub
    cp = cp + 1;
    [a, loc, cp] = partition_rand(a, lb, ub, cp);
```

122CS0067

```
    cp = qs(a, lb, loc-1, cp);
    cp = qs(a, loc+1, ub, cp);
end
cp = cp + 1;
end
%----------------------------------------
%Regular partition which selects leftmost element as pivot
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);
a(last) = pivot;
d = last;
end
%----------------------------------
%Randomized Partition Function
function [a, d, cp] = partition_rand(a, lb, ub, cp)
pivot = a(randi([lb,ub]));

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
```

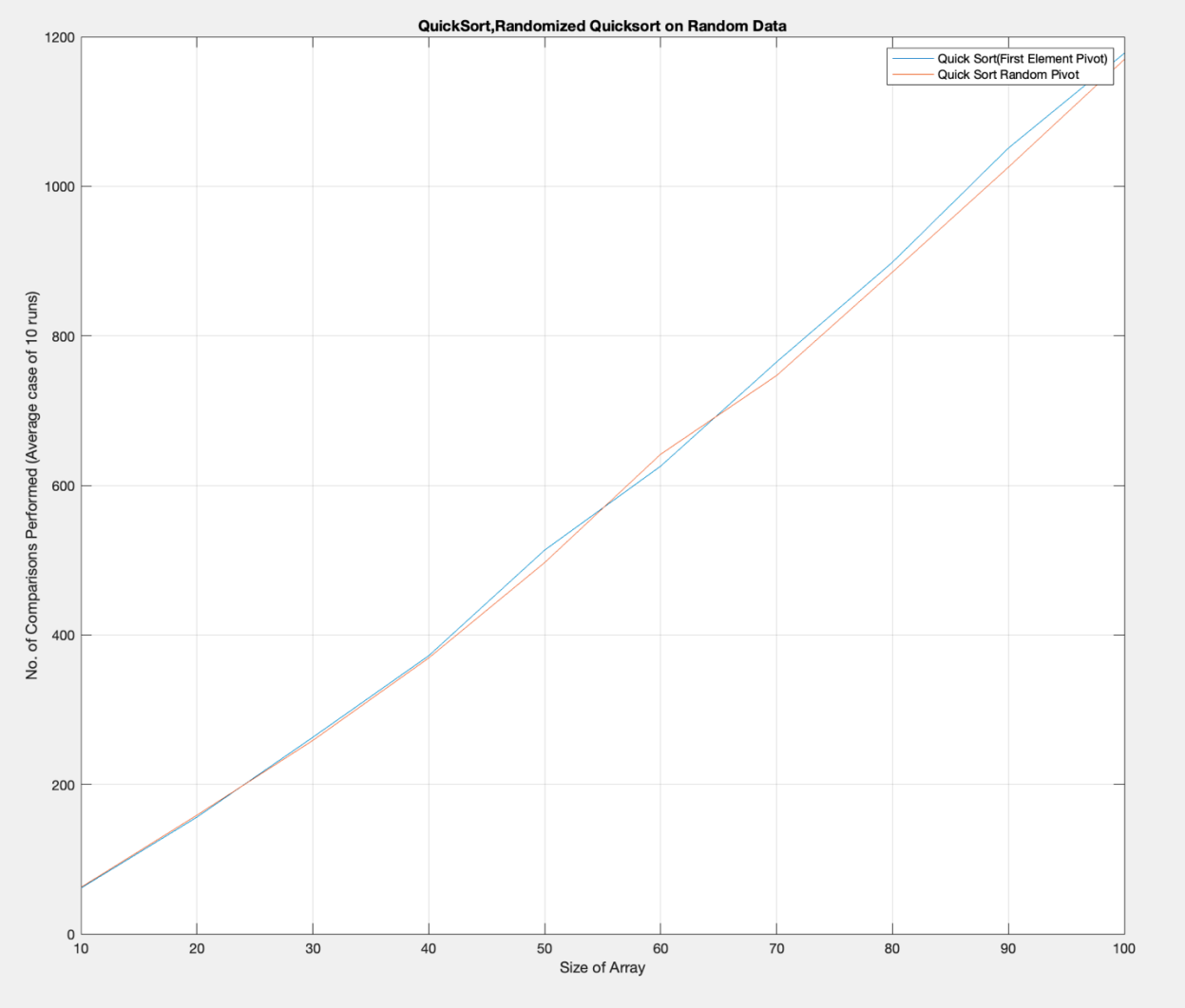
```
122CS0067
a(lb) = a(last);
a(last) = pivot;
d = last;
end
%-----------
function almost_sorted = almost_sorted(n,perturb)
%Generates almost sorted 1Xn arrays
mat1=sort(round(rand(1,n)*10));
mat2=randi([-perturb +perturb],1,n);
almost_sorted=mat1+mat2;
end
```



Analysis:

On the special case that the input to the quicksort algorithm is reverse sorted data, the algorithm encounters its worst-case scenario. Clearly, the randomized quicksort algorithm outshines the one with the static pivot selection in this case.

122CS0067



On regular random input data, both have approximately the same performance as is expected. However, the randomized algorithm does perform better in the worst case scenario that the array is reverse sorted as we have seen.