**ASSIGNMENT-2**

Q.6. Compute value of π

Compute π using randomized algorithm.

**Code:**

% Computing Pi using Monte Carlo simulation

max\_size = 100;

y\_axis = zeros(1,max\_size);

x\_axis = 1:max\_size;

k = 0;

for i=1:max\_size

x = rand;

y = rand;

if (x^2 + y^2 <= 1)

k = k+1;

end

y\_axis(i) = (k/i) \* 4;

end

% Plot

figure;

plot(x\_axis, y\_axis, LineWidth=1.5, Color='b');

title("Computing the value of PI");

grid on;

**Simulation:**

****

**Observations:**

The graph that has been simulated using the Monte Carlo simulation shows all the possible outcomes but does not accurately show the correct value of π (which is theoretically 3.14).

Q. 8. PRIMALITY TESTING

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

**Code:**

% Correctness of the primality testing

max\_size = 100;

x\_axis = 1:max\_size;

y\_axis = zeros(1, max\_size);

for n=1:max\_size

if n < 2

y\_axis(n) = 0;

continue;

end

flag = 0;

num = 0;

for i=2:sqrt(n)

if rem(n, i) == 0

flag = 1;

break;

end

num=num+1;% Record the conditional checks

end

if flag == 0

y\_axis(n) = num;

end

y\_axis(n) = num;

end

% Plotting the correctness of the primality correctness

figure;

plot(x\_axis, y\_axis, LineStyle="-", Color='g');

title("Correctness of Primality Correctness");

xlabel("Numbers");

ylabel("No. of Comparisons");

grid on;

**Simulation:**



**Observations:**

From the graph, we can observe that the number of comparisons increases as the number increases. The composite numbers are below the peaks hence being the local minima of the curve while the prime numbers are the local maxima of the curve.

Q. 10. Randomized Quick Sort

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

**Code:**

% Comparing the randomized quicksort with conventional quicksort

max\_size = 1000;

x\_axis = 1:max\_size;

y\_axis = zeros(1, max\_size);

% functions to execute the conventional quick sort

function [arr, pidx, num] = partition(arr, low, high)

num = 0;

pivot = arr(high);

i = low-1;

for j=low:high-1

num = num+1;

if arr(j) > pivot

i=i+1;

temp =arr(i);

arr(i) = arr(j);

arr(j) = temp;

end

end

i = i+1;

temp = arr(i);

arr(i) = arr(high);

arr(high) = temp;

pidx=i;

end

function [arr, totalcomp] = quicksort(arr, low, high)

totalcomp = 0;

if low < high

[arr, pidx, num] = partition(arr, low, high);

totalcomp = totalcomp + num;

[arr, leftcomp] = quicksort(arr, low, pidx-1);

totalcomp = totalcomp + leftcomp;

[arr, rightcomp] = quicksort(arr, pidx+1, high);

totalcomp = totalcomp + rightcomp;

end

end

% Randomized algorithm

for n=1:max\_size

arr = round(rand(1, n)\*100);

arr1 = zeros(1, n);

num=0;

for i=1:n-1

j = randi(1, n);

if arr(j) ~= 0

break;

end

arr1(j) = arr(i);

num=num+1;

end

for a=1:n-2

if arr1(a+1) < arr1(a)

break;

end

num=num+1;

end

y\_axis(n) = num;

end

% Conventional Quicksort

y\_axis2 = zeros(1, max\_size);

for t=1:max\_size

arr2 = round(rand(1, t)\*100);

[~, num1] = quicksort(arr2, 1, t);

y\_axis2(t) = num1;

end

figure;

plot(x\_axis, y\_axis, LineWidth=2, Color='r');

title("Randomized Algorithm");

xlabel('Data Size Input');

ylabel("Number of Comparisons");

grid on;

hold on;

plot(x\_axis, y\_axis2, LineStyle="-", Color='b');

legend("Randomized QuickSort", "Conventional Quicksort");

**Simulation:**

****

**Observations:**

Curves that occupy more area in the graph show worse performance. So hence we see that the randomized quicksort algorithm shows a much better performance(shows almost linear behavior while taking linear extra space) than the conventional quicksort algorithm.