ASSIGNMENT-3

1. Problem Statement

Given an undirected, connected graph, we want to traverse the graph randomly, ensuring that all nodes are eventually visited. Instead of following a predefined order, we select the next node randomly from the current node's neighbors.

Objective: Implement a randomized depth-first search (DFS) to traverse the graph and compare the performance with DFS and BFS.

CODE:

```
% Randomized DFS algorithm
function gpath = dfs(graph, start, target, visited, path)
  % If target is found, return the path
  if (start == target)
    gpath = [path, start];
    return;
  end

  % If already visited, return empty
  if visited(start)
    gpath = [];
    return;
  end
```

```
    visited(start) = true;
    path = [path, start];

    neighbours = find(graph(start, :));

    % Shuffle the neighbours randomly
    neighbours = neighbours(randperm(length(neighbours)));

    % Perform DFS on each neighbor
    for i = 1:length(neighbours)
        node = neighbours(i);
        if ~visited(node)
            result = dfs(graph, node, target, visited, path);
            if ~isempty(result)
                gpath = result;
                return;
            end
        end
    end

    gpath = [];
end
% Traditional DFS Algorithm
function gpath = standard_dfs(graph, start, target, visited, path)
    % If target is found, return the path
    if (start == target)
        gpath = [path, start];
        return;
    end
```

```
    % If already visited, return empty
    if visited(start)
        gpath = [];
        return;
    end

    visited(start) = true;
    path = [path, start];

    neighbours = find(graph(start, :));



    neighbours = neighbours(randperm(length(neighbours)));

    % Perform DFS on each neighbor
    for i = 1:length(neighbours)
        node = neighbours(i);
        if ~visited(node)
            result = standard_dfs(graph, node, target, visited, path);
            if ~isempty(result)
                gpath = result;
                return;
            end
        end
    end

    gpath = [];
end
```

```
% Creating a sample graph to check the performance of the algorithm
numNodes = 10;
s = [1 1 2 3 4 5 6 3 4];
t = [2 3 4 5 7 2 4 1 3];

G = digraph(s, t);

A = adjacency(G);

visited = false(numNodes, 1);
path = [];

% Call Randomized DFS function
newpath = dfs(A, 1, 7, visited, path);
disp('Randomized DFS Path:');
disp(newpath);
time_randomized = toc;

visited = false(numNodes, 1);
path = [];
% Call Standard DFS function
disp('Standard DFS Path: ');
standard_path = standard_dfs(A, 1, 7, visited, path);
disp(standard_path);
time_standard = toc;

% Comparing the performances
disp('Permance Comaprison: ');
fprintf('Randomized DFS: Time = %.6f seconds, Path Length = %d\n', time_randomized,
length(newpath));
```

```
fprintf('Standard DFS Path: Time = %.6f seconds, Path Length = %d\n', time_standard, length(standard_path));
```

SIMULATION OUPUT:

Randomized DFS Path:

1 3 5 2 4 7

Standard DFS Path:

1 3 5 2 4 7

Performance Comparison:

Randomized DFS: Time = 3272.038840 seconds, Path Length = 6

Standard DFS Path: Time = 3272.041995 seconds, Path Length = 6

OBSERVATIONS:

From the results of the simulation, we can infer that although both randomized DFS and the traditional DFS algorithm work on similar principles, the randomized DFS algorithm can avoid worst-case scenarios with the help of randomness while the same cannot be said for the traditional DFS algorithm.

2. Problem Statement

Given a large dataset (millions of records), sorting needs to be performed efficiently while avoiding worst-case scenarios. Challenges:

1. Deterministic QuickSort suffers from O(n2) worst-case complexity if the pivot selection is poor.

2. Large datasets may cause performance bottlenecks in memory usage and computation time.

3. Parallel execution requires adaptable sorting algorithms.

Objective: Implement Randomized QuickSort, analyze its performance, and compare it with traditional sorting algorithms (Quick Sort & Merge Sort).

CODE:

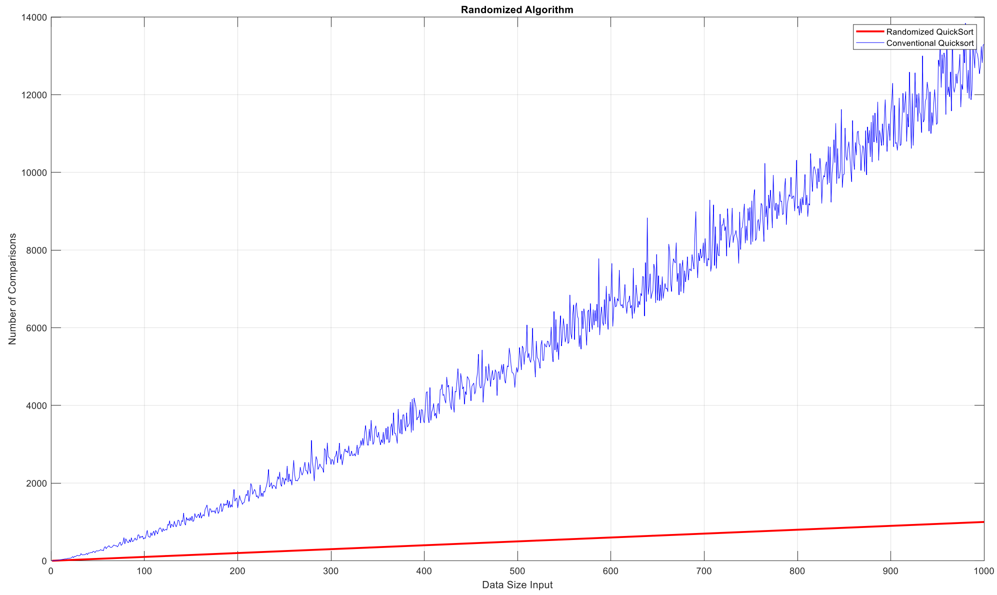
```
% Randomized Algorithm
max_size = 1000;
x_axis = 1:max_size;
y_axis = zeros(1, max_size);
% Functions to highlight the implementation of traditional QuickSort
% algorithm
function [arr, pidx, num] = partition(arr, low, high)
    num = 0;
    pivot = arr(high);
    i = low-1;

    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
% Implementation of Randomized QuickSort algorithm on different arrays
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

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:

Most of the traditional algorithms like Quick sort, Merge sort etc, tend to perform much worse compared to Randomized Quick sort as they tend to go into worst-case scenarios. In terms of time complexity, traditional sorting algorithms go upto O(N^2) and O(N) for randomized sorting algorithms.