VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Name: Apurba Koirala

Reg no: 22BCE3799

Subject Code: BCSE204P

Course Title: Design and Analysis of Algorithms Lab

Lab Slot: L39 + L40

Guided by: Dr. IYAPPAN P

Lab Assessment 3

1. Design a solution to see if a content C = PGGA is plagiarized in Text T = SAQSPAPGPGGAS using the following Algorithms.
    a. KMP Algorithm
    b. Rabin-Karp Algorithm

a. KMP Algorithm

Algorithm:

```
KMP_String_Matcher(T, P):
    n = length(T)
    m = length(P)
    π = Compute_LPS(P)
    i = 0
    j = 0

    while i < n:
        if T[i] == P[j]:
            i = i + 1
            j = j + 1

            if j == m:
                print("Pattern occurs at index", i - j)
                j = π[j - 1]

        else:
            if j != 0:
                j = π[j - 1]
            else:
                i = i + 1

Compute_LPS(P):
    m = length(P)
    LPS = array of size m
    LPS[0] = 0
    len = 0
    i = 1

    while i < m:
        if P[i] == P[len]:
            len = len + 1
            LPS[i] = len
            i = i + 1
```

```
        else:
            if len != 0:
                len = LPS[len - 1]
            else:
                LPS[i] = 0
                i = i + 1

    return LPS
```

Source Code:

```cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;

void constructLps(string &pat, vector<int> &lps) {

    int len = 0;

    lps[0] = 0;

    int i = 1;
    while (i < pat.length()) {

        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        }

        else {
            if (len != 0) {

                len = lps[len - 1];
            }
            else {
                lps[i] = 0;
```

```
            i++;
        }
    }
  }
}

vector<int> search(string &pat, string &txt) {
    int n = txt.length();
    int m = pat.length();

    vector<int> lps(m);
    vector<int> res;

    constructLps(pat, lps);
    int i = 0;
    int j = 0;

    while (i < n) {

        if (txt[i] == pat[j]) {
            i++;
            j++;

            if (j == m) {
                res.push_back(i - j);

                j = lps[j - 1];
            }
        }

        else {

            if (j != 0)
                j = lps[j - 1];
            else
                i++;
        }
```

```
    }
    return res;
}

int main() {
    string txt = "SAQSPAPGPGGAS";
    string pat = "PGGA";

    vector<int> res = search(pat, txt);
    for (int i = 0; i < res.size(); i++)
        cout << res[i] << " ";

    return 0;
}
```

Input:

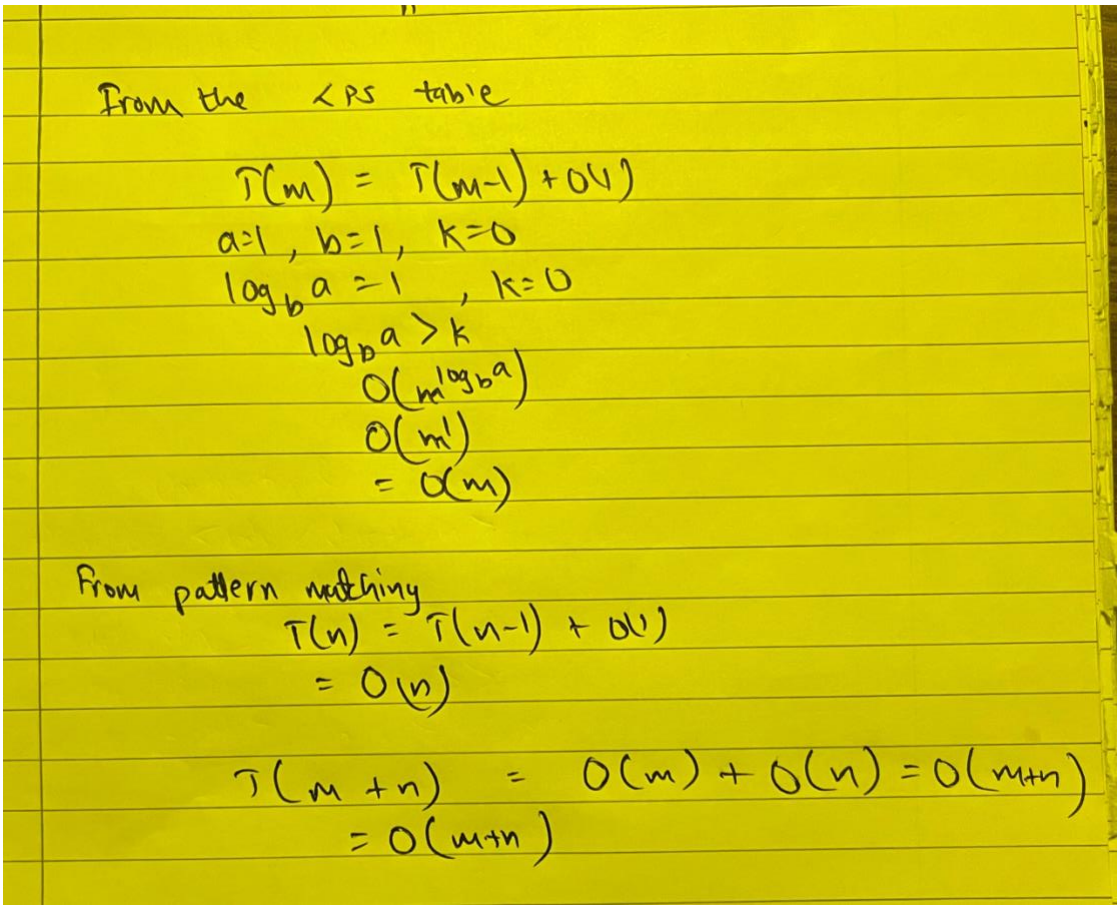
```
    string txt = "SAQSPAPGPGGAS";
    string pat = "PGGA";
```

Output:

```
Analysis\ of\ Algorithms\ Iyappan/Lab/Assessment3/KMP ; exit;
8
```

= 8 is the index

Time Complexity Analysis:



b. Rabin-Karp Algorithm

Algorithm:

```
Rabin_Karp_Matcher(T, P, d, q):
    n = length(T)
    m = length(P)
    h = d^(m-1) mod q
    p = 0
    t = 0

    for i = 0 to m-1:
        p = (d * p + P[i]) mod q
```

```
        t = (d * t + T[i]) mod q

    for s = 0 to n-m:
        if p == t:
            if P[0..m-1] == T[s..s+m-1]:
                print "Pattern occurs at index", s

        if s < n-m:
            t = (d * (t - T[s] * h) + T[s + m]) mod q
            if t < 0:
                t = t + q
```

Source Code:

```
#include <iostream>
#include <string>
using namespace std;

void search(string pat, string txt, int q)
{
    int M = pat.size();
    int N = txt.size();
    int i, j;
    int p = 0;
    int t = 0;
    int h = 1;
    int d = 256;
    for (i = 0; i < M - 1; i++)
        h = (h * d) % q;

    for (i = 0; i < M; i++) {
        p = (d * p + pat[i]) % q;
        t = (d * t + txt[i]) % q;
    }
    for (i = 0; i <= N - M; i++) {
        if (p == t) {
            for (j = 0; j < M; j++) {
                if (txt[i + j] != pat[j]) {
                    break;
                }
```

```
        }

        if (j == M)
            cout << "Pattern found at index " << i
                 << endl;
      }
      if (i < N - M) {
        t = (d * (t - txt[i] * h) + txt[i + M]) % q;
        if (t < 0)
            t = (t + q);
      }
   }
}

int main()
{
   string txt = "SAQSPAPGPGGAS";
   string pat = "PGGA";

   int q = INT_MAX;
   search(pat, txt, q);
   return 0;
}
```

Input:

```
string txt = "SAQSPAPGPGGAS";
string pat = "PGGA";
```

Output:

```
\ and\ Analysis\ of\ Algorithms\
Pattern found at index 8
```

At index 8 again

Time Complexity Analysis:

Rolling hash function updates in $O(1)$ in each shift.

So

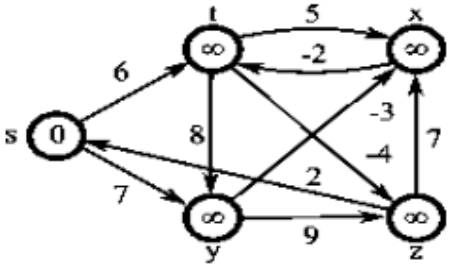
Average case = $\Theta(n-m+1) = \Theta(n)$

If all hash values collide, a full character comparison is done at each position.

$O(m(n-m+1))$

$= O(mn)$

2. Consider the given directed graph G={V,E}, Determine the shortest path from Source vertex to all the remaining vertices of the Graph by using appropriate algorithm and analyze its time complexity.



Algorithm:

```
createGraph(V, E):
    Initialize Graph with V vertices and E edges
    Return Graph

displayDistances(distances):
    Print "Vertex   Distance from Source"
    For each (vertex, distance) in distances:
        Print vertex, distance

bellmanFord(graph, source):
    Initialize distances for all vertices as ∞, except source = 0
    Repeat (V-1) times:
        For each edge (u, v, w) in graph:
            If distances[u] + w < distances[v]:
                Update distances[v] = distances[u] + w

    For each edge (u, v, w) in graph:
        If distances[u] + w < distances[v]:
            Print "Negative cycle detected"
            Return

    displayDistances(distances)
```

Source Code:

```
#include <iostream>
#include <cstdlib>
#include <climits>
#include <map>
using namespace std;

struct Edge {
    char startVertex, endVertex;
    int edgeWeight;
};

struct Graph {
    int numVertices, numEdges;
    struct Edge* edges;
};

struct Graph* initializeGraph(int vertices, int edges) {
```

```
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = vertices;
    graph->numEdges = edges;
    graph->edges = (struct Edge*)malloc(graph->numEdges * sizeof(struct Edge));
    return graph;
}

void displayFinalDistances(map<char, int>& distanceMap) {
    cout << "\nVertex\tMinimum Distance from Source" << endl;
    for (auto& entry : distanceMap) {
        cout << entry.first << "\t\t" << entry.second << endl;
    }
}

void bellmanFord(struct Graph* graph, char sourceVertex) {
    int vertices = graph->numVertices;
    int edges = graph->numEdges;
    map<char, int> shortestDistances;

    for (int i = 0; i < edges; i++) {
        shortestDistances[graph->edges[i].startVertex] = INT_MAX;
        shortestDistances[graph->edges[i].endVertex] = INT_MAX;
    }

    shortestDistances[sourceVertex] = 0;

    for (int i = 1; i <= vertices - 1; i++) {
        for (int j = 0; j < edges; j++) {
            char u = graph->edges[j].startVertex;
            char v = graph->edges[j].endVertex;
            int weight = graph->edges[j].edgeWeight;

            if (shortestDistances[u] != INT_MAX && shortestDistances[u] + weight < shortestDistances[v]) {
                shortestDistances[v] = shortestDistances[u] + weight;
            }
        }
    }
```

```
    for (int i = 0; i < edges; i++) {
        char u = graph->edges[i].startVertex;
        char v = graph->edges[i].endVertex;
        int weight = graph->edges[i].edgeWeight;

        if (shortestDistances[u] != INT_MAX && shortestDistances[u] + weight < shortestDistances[v]) {
            cout << "\nWarning: The graph contains a negative weight cycle!" << endl;
            return;
        }
    }

    displayFinalDistances(shortestDistances);
}

int main() {
    int vertices, edges;
    char source;

    cout << "Enter the number of vertices: ";
    cin >> vertices;
    cout << "Enter the number of edges: ";
    cin >> edges;
    cout << "Enter the source vertex: ";
    cin >> source;

    struct Graph* graph = initializeGraph(vertices, edges);

    for (int i = 0; i < edges; i++) {
        cout << "\nEnter properties for edge " << i + 1 << " (Start Vertex, End Vertex, Weight): ";
        cin >> graph->edges[i].startVertex >> graph->edges[i].endVertex >> graph->edges[i].edgeWeight;
    }

    bellmanFord(graph, source);

    return 0;
}
```

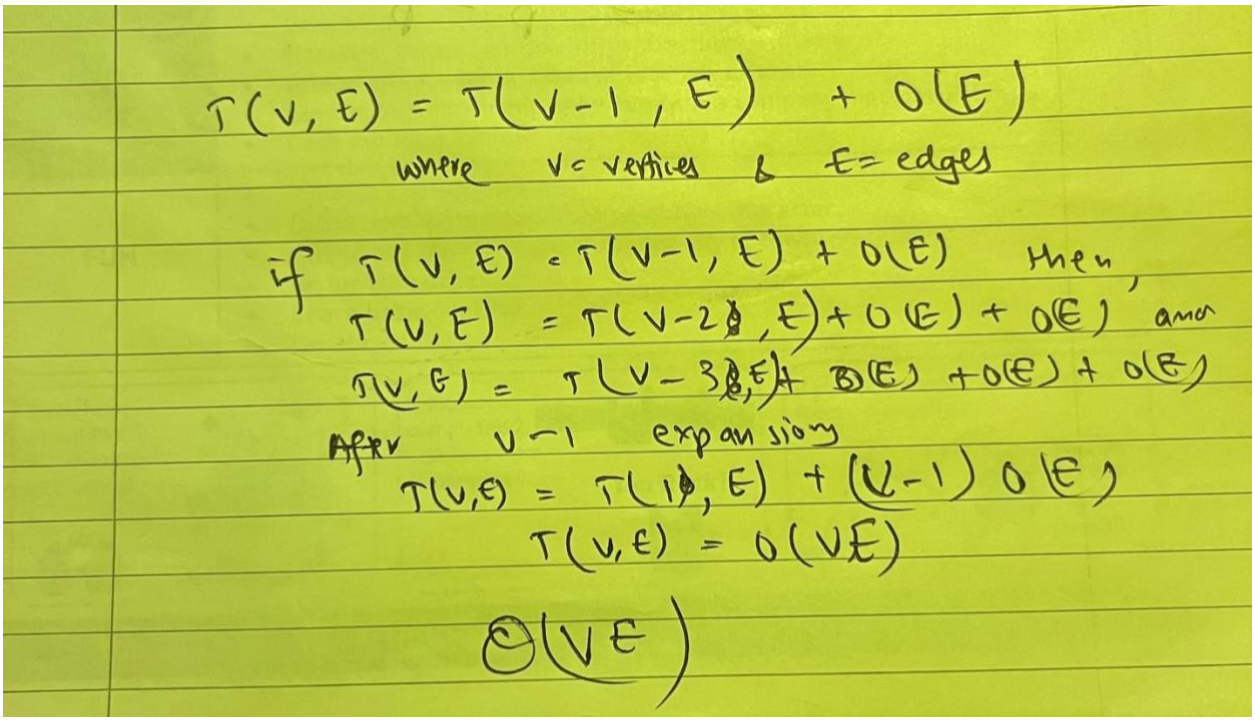
Input:

```
Enter the number of vertices: 5
Enter the number of edges: 10
Enter the source vertex: s

Enter properties for edge 1 (Start Vertex, End Vertex, Weight): s t 6

Enter properties for edge 2 (Start Vertex, End Vertex, Weight): s y 7

Enter properties for edge 3 (Start Vertex, End Vertex, Weight): t x 5

Enter properties for edge 4 (Start Vertex, End Vertex, Weight): t y 8

Enter properties for edge 5 (Start Vertex, End Vertex, Weight): t z -4

Enter properties for edge 6 (Start Vertex, End Vertex, Weight): y x -3

Enter properties for edge 7 (Start Vertex, End Vertex, Weight): y z 9

Enter properties for edge 8 (Start Vertex, End Vertex, Weight): z x 7

Enter properties for edge 9 (Start Vertex, End Vertex, Weight): z s 2

Enter properties for edge 10 (Start Vertex, End Vertex, Weight): x t -2
```

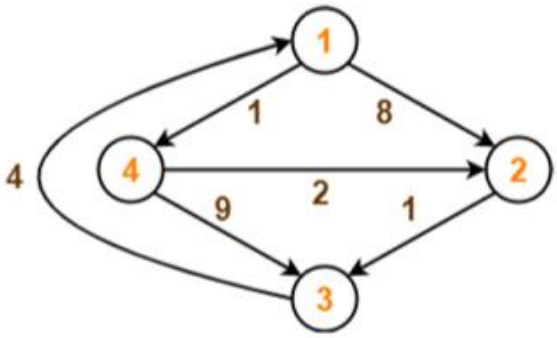
Output:

```
Vertex  Minimum Distance from Source
s               0
t               2
x               4
y               7
z               -2

Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[Process completed]
```

Time Complexity Analysis:



3. Determine step-by-step process of finding the shortest path distance between every pair of vertices using Floyd warshall algorithm, from the directed weighted graph.



Algorithm

```
FloydWarshall(graph, vertices):
    numVertices = size of vertices

    for k from 0 to numVertices - 1:
        for i from 0 to numVertices - 1:
            for j from 0 to numVertices - 1:
                start = vertices[i]
                end = vertices[j]
                intermediate = vertices[k]

                if graph[start][intermediate] is not INF and graph[intermediate][end] is not INF:
                    if graph[start][end] is INF or graph[start][end] > graph[start][intermediate] +
graph[intermediate][end]:
                        graph[start][end] = graph[start][intermediate] + graph[intermediate][end]
```

Source Code

```
#include <iostream>
#include <vector>
#include <map>
#include <climits>
```

```
using namespace std;

void floydWarshall(map<char, map<char, int> > &graph, vector<char> &vertices) {
    int numVertices = vertices.size();

    for (int k = 0; k < numVertices; k++) {
        for (int i = 0; i < numVertices; i++) {
            for (int j = 0; j < numVertices; j++) {
                char start = vertices[i], end = vertices[j], intermediate = vertices[k];

                if (graph[start][intermediate] != INT_MAX && graph[intermediate][end] != INT_MAX &&
                    (graph[start][end] == INT_MAX || graph[start][end] > graph[start][intermediate] +
graph[intermediate][end])) {
                    graph[start][end] = graph[start][intermediate] + graph[intermediate][end];
                }
            }
        }
    }
}

int main() {
    int numVertices, numEdges;

    cout << "Enter the number of vertices: ";
    cin >> numVertices;
    cout << "Enter the number of edges: ";
    cin >> numEdges;

    vector<char> vertices(numVertices);
    map<char, map<char, int> > graph;

    cout << "Enter the vertex labels: ";
    for (int i = 0; i < numVertices; i++) {
        cin >> vertices[i];
        for (int j = 0; j < numVertices; j++) {
            graph[vertices[i]][vertices[j]] = (i == j) ? 0 : INT_MAX;
        }
```

```
    }

    cout << "Enter edges in format (source destination weight):" << endl;
    for (int i = 0; i < numEdges; i++) {
        char source, destination;
        int weight;
        cin >> source >> destination >> weight;
        graph[source][destination] = weight;
    }

    floydWarshall(graph, vertices);

    cout << "\nAll-Pairs Shortest Paths:\n   ";
    for (char vertex : vertices) {
        cout << vertex << "\t";
    }
    cout << "\n------------------------------------------------\n";

    for (char start : vertices) {
        cout << start << " | ";
        for (char end : vertices) {
            if (graph[start][end] == INT_MAX)
                cout << "INF\t";
            else
                cout << graph[start][end] << "\t";
        }
        cout << endl;
    }

    return 0;
}
```

Input

```
orithms\ Iyappan/Lab/Assessment3/FM ; exit;
Enter the number of vertices: 4
Enter the number of edges: 6
Enter the vertex labels: 1
2
3
4
Enter edges in format (source destination weight):
1 4 1
1 2 8
4 2 2
4 3 9
2 3 1
3 1 4
```

Output

```
All-Pairs Shortest Paths:
   1    2       3       4
-----------------------------------------------
1 | 0   2       0       0
2 | 5   0       1       0
3 | 4   2       0       0
4 | 7   2       3       0
```

Time Complexity Analysis

$T(V) = V * T(V-1) + O(1)$

Since each iteration needs $O(V^3)$ for a vertex

$$T(V) = V^3 + (V-1)^3 + (V-2)^3 + \ldots + (V-(V-1))^3$$

$$= \left(\frac{V^2(V+1)}{2}\right)^2$$

$$= \frac{V^2(V^2+1)^2}{4}$$

$= O\left(\frac{V^4}{4}\right)$ which is simplified to

$O(V^3)$ //