VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Lab Assignment - III, Winter Semester 2023-24

Course Code : BCSE204P	**Slot** : L9+ L10
Course Name: Design and Analysis of Algorithms	**Marks** : 10
Date of Submission : 19-03-2024	

1. String Matching:

i)Naïve approach

CODE:

```
// C program for Naive Pattern Searching algorithm
#include <stdio.h>
#include <string.h>

void search(char* pat, char* txt)
{
	int M = strlen(pat);
	int N = strlen(txt);

	/* A loop to slide pat[] one by one */
	for (int i = 0; i <= N - M; i++) {
		int j;

		/* For current index i, check for pattern match */
		for (j = 0; j < M; j++)
			if (txt[i + j] != pat[j])
				break;

		if (j
			== M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
			printf("Pattern found at index %d \n", i);
	}
}

// Driver's code
int main()
{
	char txt[] = "AABAACAADAABAAABAA";
	char pat[] = "AABA";

	// Function call
	search(pat, txt);
	return 0;
```

}

CODE SCREENSHOT:

OUTPUT:

Pattern found at index 0
Pattern found at index 9
Pattern found at index 13

OUTPUT SCREENSHOT:

ii)Rabin Karp algorithm

CODE:

```
/* Following program is a C implementation of Rabin Karp
Algorithm given in the CLRS book */
#include <stdio.h>
#include <string.h>

// d is the number of characters in the input alphabet
#define d 256

/* pat -> pattern
        txt -> text
        q -> A prime number
*/
void search(char pat[], char txt[], int q)
{
```

```
int M = strlen(pat);
int N = strlen(txt);
int i, j;
int p = 0; // hash value for pattern
int t = 0; // hash value for txt
int h = 1;

// The value of h would be "pow(d, M-1)%q"
for (i = 0; i < M - 1; i++)
        h = (h * d) % q;

// Calculate the hash value of pattern and first
// window of text
for (i = 0; i < M; i++) {
        p = (d * p + pat[i]) % q;
        t = (d * t + txt[i]) % q;
}

// Slide the pattern over text one by one
for (i = 0; i <= N - M; i++) {

        // Check the hash values of current window of text
        // and pattern. If the hash values match then only
        // check for characters one by one
        if (p == t) {
                /* Check for characters one by one */
                for (j = 0; j < M; j++) {
                        if (txt[i + j] != pat[j])
                                break;
                }

                // if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
                if (j == M)
                        printf("Pattern found at index %d \n", i);
        }

        // Calculate hash value for next window of text: Remove
        // leading digit, add trailing digit
        if (i < N - M) {
                t = (d * (t - txt[i] * h) + txt[i + M]) % q;

                // We might get negative value of t, converting it
                // to positive
                if (t < 0)
                        t = (t + q);
        }
}
```

```
}

/* Driver program to test above function */
int main()
{
        char txt[] = "GEEKS FOR GEEKS";
        char pat[] = "GEEK";
        int q = 101; // A prime number
        search(pat, txt, q);
        return 0;
}
```

CODE SCREENSHOT:

OUTPUT:

Pattern found at index 0
Pattern found at index 10

OUTPUT SCREENSHOT:

iii) K M P algorithm

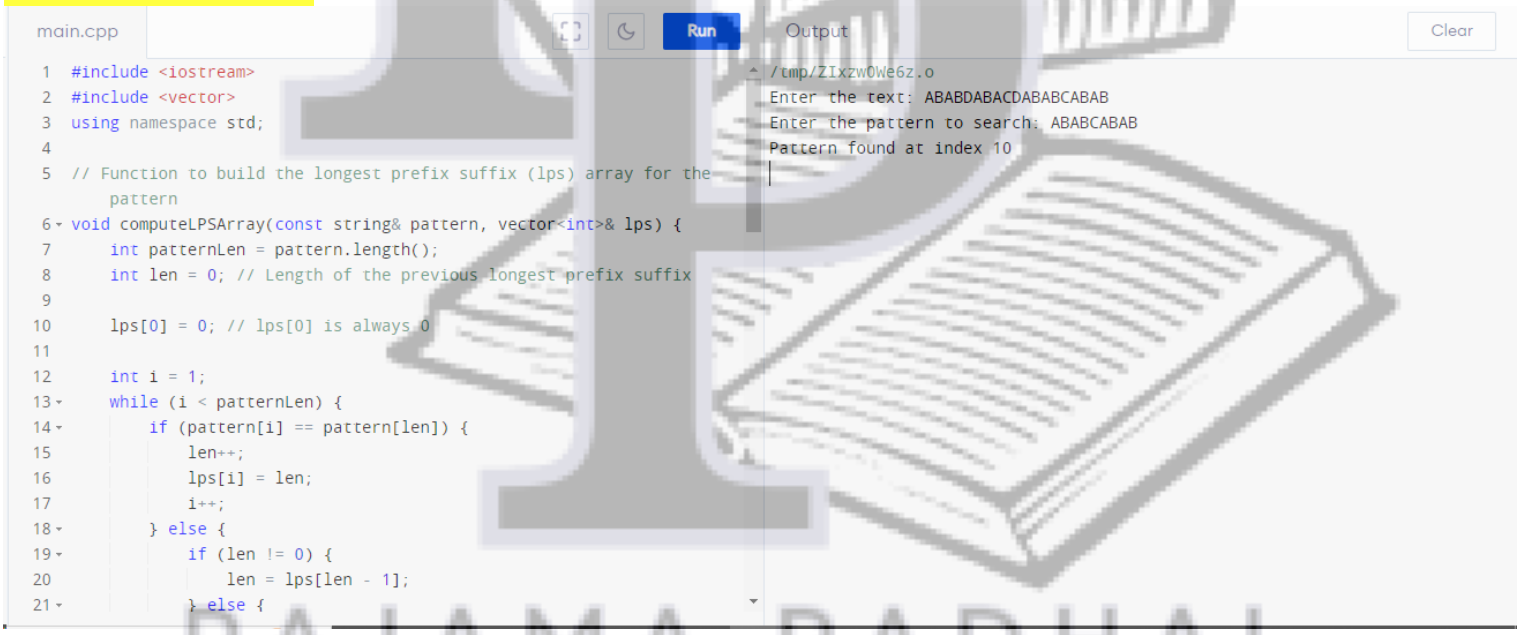
CODE:

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

```
// Function to build the longest prefix suffix (lps) array for the pattern
void computeLPSArray(const string& pattern, vector<int>& lps) {
    int patternLen = pattern.length();
    int len = 0; // Length of the previous longest prefix suffix

    lps[0] = 0; // lps[0] is always 0

    int i = 1;
    while (i < patternLen) {
        if (pattern[i] == pattern[len]) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}

// Function to perform pattern searching using the KMP algorithm
void KMPSearch(const string& text, const string& pattern) {
    int textLen = text.length();
    int patternLen = pattern.length();

    // Create lps array to store the longest prefix suffix values
    vector<int> lps(patternLen, 0);

    // Preprocess the pattern to build the lps array
    computeLPSArray(pattern, lps);

    int i = 0; // Index for text[]
    int j = 0; // Index for pattern[]

    while (i < textLen) {
        if (pattern[j] == text[i]) {
            i++;
            j++;
        }

        if (j == patternLen) {
            cout << "Pattern found at index " << i - j << endl;
            j = lps[j - 1];
```

```
    } else if (i < textLen && pattern[j] != text[i]) {
      if (j != 0)
        j = lps[j - 1];
      else
        i++;
    }
  }
}

int main() {
  string text, pattern;
  cout << "Enter the text: ";
  cin >> text;

  cout << "Enter the pattern to search: ";
  cin >> pattern;

  KMPSearch(text, pattern);

  return 0;
}
```

CODE SCREENSHOT:



OUTPUT:

Enter the text: ABABDABACDABABCABAB
Enter the pattern to search: ABABCABAB
Pattern found at index 10

OUTPUT SCREENSHOT:

```
Output                                                    Clear
/tmp/ZIxzw0We6z.o
Enter the text: ABABDABACDABABCABAB
Enter the pattern to search: ABABCABAB
Pattern found at index 10
```

2. All pair shortest path Algorithm
Floyd Warshall Algorithm

CODE:

```
// Floyd-Warshall Algorithm in C

#include <stdio.h>

// defining the number of vertices
#define nV 4

#define INF 999

void printMatrix(int matrix[][nV]);

// Implementing floyd warshall algorithm
void floydWarshall(int graph[][nV]) {
  int matrix[nV][nV], i, j, k;

  for (i = 0; i < nV; i++)
    for (j = 0; j < nV; j++)
      matrix[i][j] = graph[i][j];

  // Adding vertices individually
  for (k = 0; k < nV; k++) {
    for (i = 0; i < nV; i++) {
      for (j = 0; j < nV; j++) {
        if (matrix[i][k] + matrix[k][j] < matrix[i][j])
          matrix[i][j] = matrix[i][k] + matrix[k][j];
      }
    }
  }
  printMatrix(matrix);
}

void printMatrix(int matrix[][nV]) {
  for (int i = 0; i < nV; i++) {
    for (int j = 0; j < nV; j++) {
```

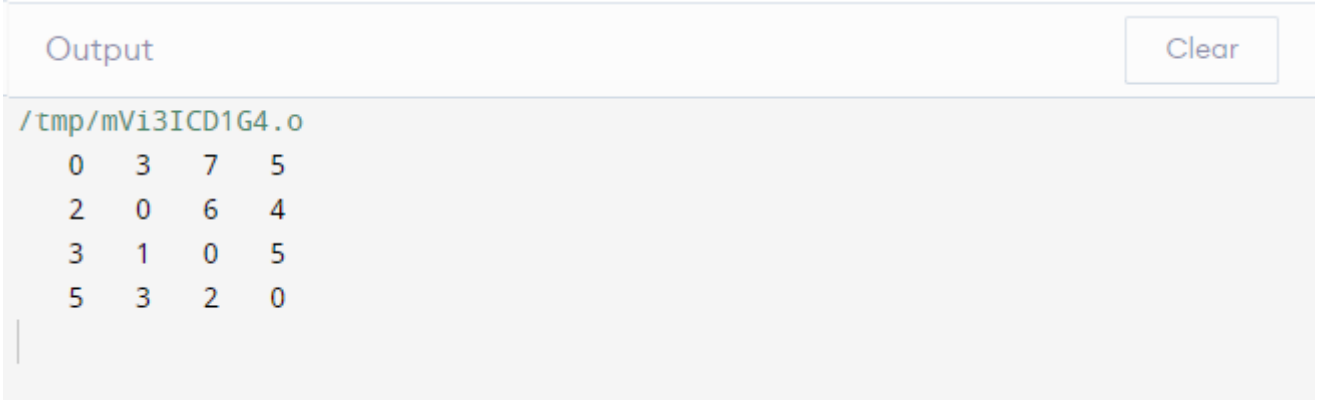
```
      if (matrix[i][j] == INF)
        printf("%4s", "INF");
      else
        printf("%4d", matrix[i][j]);
    }
    printf("\n");
  }
}

int main() {
  int graph[nV][nV] = {{0, 3, INF, 5},
          {2, 0, INF, 4},
          {INF, 1, 0, INF},
          {INF, INF, 2, 0}};
  floydWarshall(graph);
}
```

CODE SCREENSHOT:

OUTPUT:

```
0  3  7  5
2  0  6  4
3  1  0  5
5  3  2  0
```

OUTPUT SCREENSHOT:



3. Maximum Flows Algorithm

i) Ford Fulkerson

CODE:

```
// Ford - Fulkerson algorithm in C

#include <stdio.h>

#define A 0
#define B 1
#define C 2
#define MAX_NODES 1000
#define O 1000000000

int n;
int e;
int capacity[MAX_NODES][MAX_NODES];
int flow[MAX_NODES][MAX_NODES];
int color[MAX_NODES];
int pred[MAX_NODES];

int min(int x, int y) {
  return x < y ? x : y;
}

int head, tail;
int q[MAX_NODES + 2];

void enqueue(int x) {
  q[tail] = x;
  tail++;
  color[x] = B;
}

int dequeue() {
  int x = q[head];
  head++;
  color[x] = C;
  return x;
}

// Using BFS as a searching algorithm
int bfs(int start, int target) {
  int u, v;
  for (u = 0; u < n; u++) {
    color[u] = A;
  }
```

```
  head = tail = 0;
  enqueue(start);
  pred[start] = -1;
  while (head != tail) {
    u = dequeue();
    for (v = 0; v < n; v++) {
      if (color[v] == A && capacity[u][v] - flow[u][v] > 0) {
        enqueue(v);
        pred[v] = u;
      }
    }
  }
  return color[target] == C;
}

// Applying fordfulkerson algorithm
int fordFulkerson(int source, int sink) {
  int i, j, u;
  int max_flow = 0;
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      flow[i][j] = 0;
    }
  }

  // Updating the residual values of edges
  while (bfs(source, sink)) {
    int increment = O;
    for (u = n - 1; pred[u] >= 0; u = pred[u]) {
      increment = min(increment, capacity[pred[u]][u] - flow[pred[u]][u]);
    }
    for (u = n - 1; pred[u] >= 0; u = pred[u]) {
      flow[pred[u]][u] += increment;
      flow[u][pred[u]] -= increment;
    }
    // Adding the path flows
    max_flow += increment;
  }
  return max_flow;
}

int main() {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      capacity[i][j] = 0;
    }
  }
```

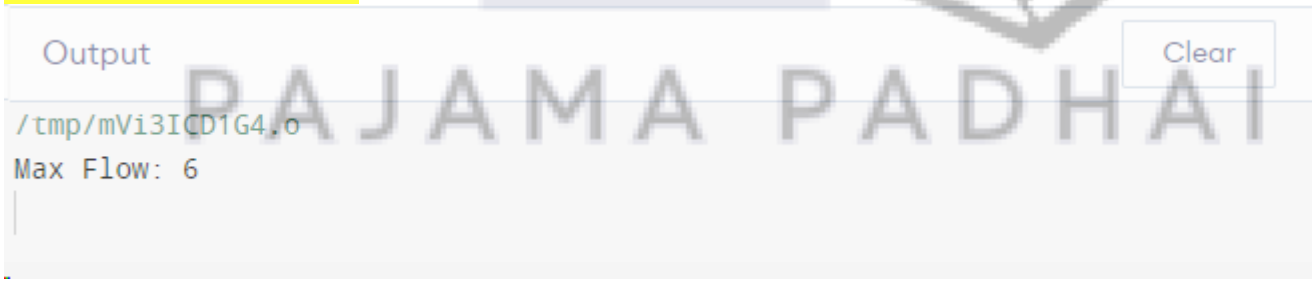
```
  n = 6;
  e = 7;

  capacity[0][1] = 8;
  capacity[0][4] = 3;
  capacity[1][2] = 9;
  capacity[2][4] = 7;
  capacity[2][5] = 2;
  capacity[3][5] = 5;
  capacity[4][2] = 7;
  capacity[4][3] = 4;

  int s = 0, t = 5;
  printf("Max Flow: %d\n", fordFulkerson(s, t));
}
```

CODE SCREENSHOT:

OUTPUT:

Max Flow: 6

OUTPUT SCREENSHOT:



ii) Push Relabel Algorithm

CODE:

```
// C# program to implement push-relabel algorithm for
// getting maximum flow of graph
using System;
```

```
using System.Collections.Generic;
using System.Collections;
using System.Linq;

class Edge
{
	public int flow;
	public int capacity;
	public int u;
	public int v;

	public Edge(int flow,int capacity,int u,int v)
	{
		this.flow = flow;
		this.capacity = capacity;
		this.u = u;
		this.v = v;
	}
}

// Represent a Vertex
class Vertex
{
	public int h;
	public int e_flow;
	public Vertex(int h,int e_flow)
	{
		this.h = h;
		this.e_flow = e_flow;
	}
}

// To represent a flow network
class Graph
{

	public int V; // No. of vertices
	public List<Vertex> ver;
	public List<Edge> edge;
	public Graph(int V)
	{
		this.V = V;
		ver = new List<Vertex>();
		edge = new List<Edge>();
		// all vertices are initialized with 0 height
		// and 0 excess flow
		for (int i = 0; i < V; i++)
```

```
            ver.Add(new Vertex(0, 0));
}

public void addEdge(int u,int v,int capacity)
{
        // flow is initialized with 0 for all edge
        edge.Add(new Edge(0, capacity, u, v));
}


public void preflow(int s)
{
        // Making h of source Vertex equal to no. of vertices
        // Height of other vertices is 0.
        ver[s].h = ver.Count;

        //
        for (int i = 0; i < edge.Count; i++)
        {
                // If current edge goes from source
                if (edge[i].u == s)
                {
                        // Flow is equal to capacity
                        edge[i].flow = edge[i].capacity;

                        // Initialize excess flow for adjacent v
                        ver[edge[i].v].e_flow += edge[i].flow;

                        // Add an edge from v to s in residual graph with
                        // capacity equal to 0
                        edge.Add(new Edge(-edge[i].flow, 0, edge[i].v, s));
                }
        }
}

// returns index of overflowing Vertex
public int overFlowVertex()
{
        for (int i = 1; i < ver.Count - 1; i++)
        if (ver[i].e_flow > 0)
                        return i;

        // -1 if no overflowing Vertex
        return -1;
}
```

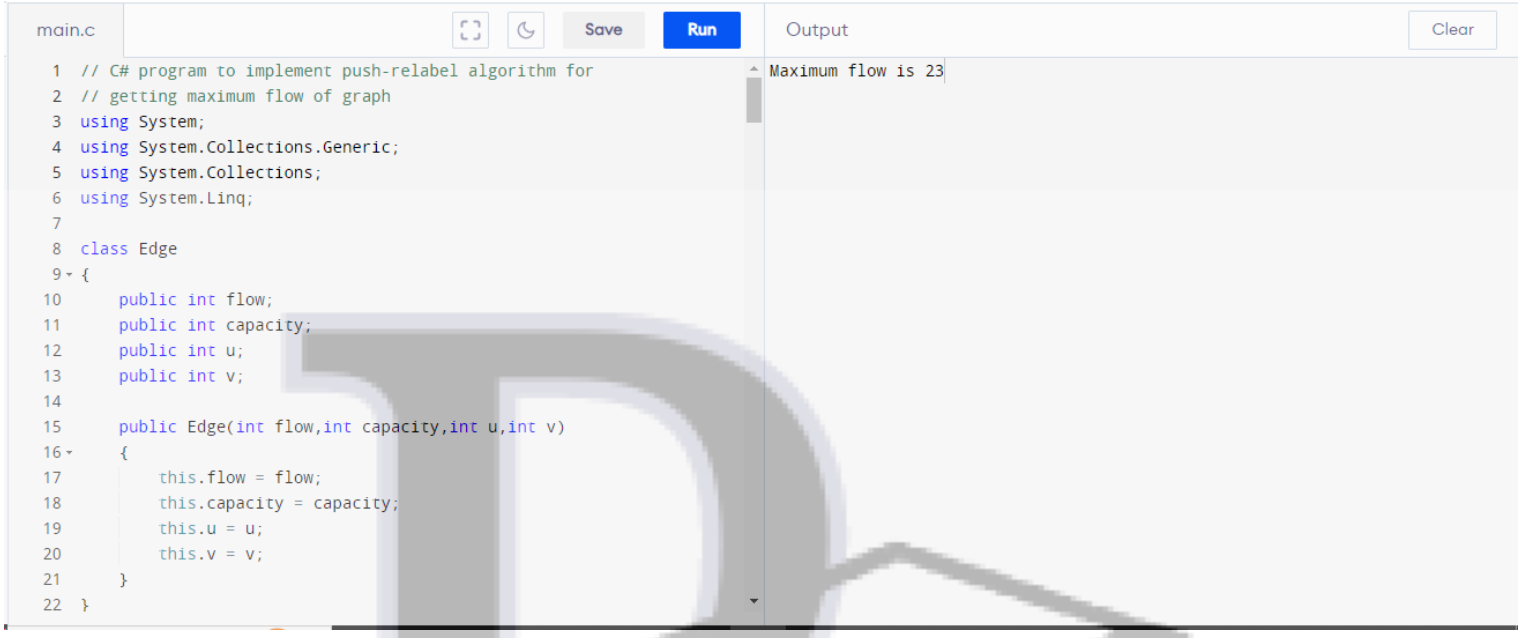
```
// Update reverse flow for flow added on ith Edge
public void updateReverseEdgeFlow(int i,int flow)
{
        int u = edge[i].v;
        int v = edge[i].u;

        for (int j = 0; j < edge.Count; j++)
        {
                if (edge[j].v == v && edge[j].u == u)
                {
                        edge[j].flow -= flow;
                        return;
                }
        }

        // adding reverse Edge in residual graph
        Edge e = new Edge(0, flow, u, v);
        edge.Add(e);
}

// To push flow from overflowing vertex u
public bool push(int u)
{
        // Traverse through all edges to find an adjacent (of u)
        // to which flow can be pushed
        for (int i = 0; i < edge.Count; i++)
        {
                // Checks u of current edge is same as given
                // overflowing vertex
                if (edge[i].u == u)
                {
                        // if flow is equal to capacity then no push
                        // is possible
                        if (edge[i].flow == edge[i].capacity)
                                continue;

                        // Push is only possible if height of adjacent
                        // is smaller than height of overflowing vertex
                        if (ver[u].h > ver[edge[i].v].h)
                        {
                                // Flow to be pushed is equal to minimum of
                                // remaining flow on edge and excess flow.
                                int flow = Math.Min(edge[i].capacity - edge[i].flow,
                                                ver[u].e_flow);

                                // Reduce excess flow for overflowing vertex
                                ver[u].e_flow -= flow;
```

```
                        // Increase excess flow for adjacent
                        ver[edge[i].v].e_flow += flow;

                        // Add residual flow (With capacity 0 and negative
                        // flow)
                        edge[i].flow += flow;

                        updateReverseEdgeFlow(i, flow);

                        return true;
                    }
                }
            }
            return false;
        }


        // function to relabel vertex u
        public void relabel(int u)
        {
            // Initialize minimum height of an adjacent
            int mh = 2100000;

            // Find the adjacent with minimum height
            for (int i = 0; i < edge.Count; i++)
            {
                if (edge[i].u == u)
                {
                    // if flow is equal to capacity then no
                    // relabeling
                    if (edge[i].flow == edge[i].capacity)
                        continue;

                    // Update minimum height
                    if (ver[edge[i].v].h < mh)
                    {
                        mh = ver[edge[i].v].h;

                        // updating height of u
                        ver[u].h = mh + 1;
                    }
                }
            }
        }

        // main function for printing maximum flow of graph
```

```
    public int getMaxFlow(int s,int t)
    {
        preflow(s);

        // loop until none of the Vertex is in overflow
        while (overFlowVertex() != -1)
        {
            int u = overFlowVertex();
            if (!push(u))
                relabel(u);
        }

        // ver.back() returns last Vertex, whose
        // e_flow will be final maximum flow
        return ver[ver.Count-1].e_flow;
    }
}



class HelloWorld {


    static void Main() {

        // Driver program to test above functions
        int V = 6;
        Graph g = new Graph(V);

        // Creating above shown flow network
        g.addEdge(0, 1, 16);
        g.addEdge(0, 2, 13);
        g.addEdge(1, 2, 10);
        g.addEdge(2, 1, 4);
        g.addEdge(1, 3, 12);
        g.addEdge(2, 4, 14);
        g.addEdge(3, 2, 9);
        g.addEdge(3, 5, 20);
        g.addEdge(4, 3, 7);
        g.addEdge(4, 5, 4);

        // Initialize source and sink
        int s = 0, t = 5;

        Console.WriteLine("Maximum flow is " + g.getMaxFlow(s, t));
    }
```

}

CODE SCREENSHOT:




```
// C# program to implement push-relabel algorithm for
// getting maximum flow of graph
using System;
using System.Collections.Generic;
using System.Collections;
using System.Linq;

class Edge
{
    public int flow;
    public int capacity;
    public int u;
    public int v;

    public Edge(int flow,int capacity,int u,int v)
    {
        this.flow = flow;
        this.capacity = capacity;
        this.u = u;
        this.v = v;
    }
}
```

OUTPUT:

Maximum flow is 23

OUTPUT SCREENSHOT: