PRACTICAL - 9

AIM : To perform djikstra's algorithm

PSEUDO CODE :

DijkstraShortestPath(Graph graph, Vertex source)

Step 1: Create an empty set to keep track of visited vertices and initialize the distance to the source vertex as 0 and all other vertices as infinity.
Step 2: Create an empty priority queue (min heap) to store vertices based on their tentative distance values.
Step 3: Insert the source vertex into the priority queue with distance 0.
Step 4: while the priority queue is not empty, do steps 5-9
Step 5: Extract the vertex with the minimum distance from the priority queue; let's call it currentVertex.
Step 6: Mark currentVertex as visited.
Step 7: for each neighborVertex of currentVertex, do steps 8-9
Step 8: Calculate the tentative distance to neighborVertex through currentVertex.
Step 9: If the tentative distance is less than the current distance to neighborVertex, update the distance and insert neighborVertex into the priority queue.
Step 10: Return the computed distances as the shortest path distances from the source vertex to all other vertices.

CODE :

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define V 100

void dijkstra(int graph[V][V], int source, int vertices) {
   int distance[vertices];  // To store the minimum distance from the source to each vertex
   int visited[vertices];  // To keep track of visited vertices
   int i, j, min_distance, u;

   for (i = 0; i < vertices; i++) {
      distance[i] = INT_MAX; // Set distance to infinity for all vertices
      visited[i] = 0; // Mark all vertices as not visited
   }

   distance[source] = 0; // Distance from source to itself is always 0
```

```c
    for (i = 0; i < vertices - 1; i++) {
        min_distance = INT_MAX;
        for (j = 0; j < vertices; j++) {
            if (!visited[j] && distance[j] < min_distance) {
                min_distance = distance[j];
                u = j;
            }
        }

        visited[u] = 1;

        for (j = 0; j < vertices; j++) {
            if (!visited[j] && graph[u][j] && (distance[u] + graph[u][j] < distance[j])) {
                distance[j] = distance[u] + graph[u][j];
            }
        }
    }

    printf("Shortest distances from source vertex %d:\n", source);
    for (i = 0; i < vertices; i++) {
        printf("Vertex %d: %d\n", i, distance[i]);
    }
}

int main() {
    printf("Rishita Chaubey - A2305221265");
    int graph[V][V];
    int vertices, source;

    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);

    printf("Enter the adjacency matrix of the graph (0 for no edge, positive values for edge
weights):\n");
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    printf("Enter the source vertex: ");
    scanf("%d", &source);

    dijkstra(graph, source, vertices);
    return 0;
```

}

OUTPUT :

```
Rishita Chaubey - A2305221265

Enter the number of vertices: 5
Enter the adjacency matrix of the graph (0 for no edge, positive
 values for edge weights):
0 10 0 5 0
0 0 1 2 0
0 0 0 0 4
0 3 9 0 2
7 0 6 0 0
Enter the source vertex: 0
Shortest distances from source vertex 0:
Vertex 0: 0
Vertex 1: 8
Vertex 2: 9
Vertex 3: 5
Vertex 4: 7
```

COMPLEXITY : O(V^2)

PRACTICAL - 10

AIM : To perform strassen's matrix multiplication

PSEUDO CODE :

StrassensMatrixMultiplication(Matrix A, Matrix B)

Step 1: If the size of matrices A and B is small (base case), perform traditional matrix multiplication and return the result.
Step 2: Divide matrices A and B into four equal-sized submatrices: A11, A12, A21, A22, and B11, B12, B21, B22.
Step 3: Create 10 temporary matrices: M1, M2, M3, M4, M5, M6, M7, C11, C12, C21, C22.
Step 4: Calculate M1 = (A11 + A22) * (B11 + B22).
Step 5: Calculate M2 = (A21 + A22) * B11.
Step 6: Calculate M3 = A11 * (B12 - B22).
Step 7: Calculate M4 = A22 * (B21 - B11).
Step 8: Calculate M5 = (A11 + A12) * B22.
Step 9: Calculate M6 = (A21 - A11) * (B11 + B12).
Step 10: Calculate M7 = (A12 - A22) * (B21 + B22).
Step 11: Calculate submatrices C11, C12, C21, and C22:
    C11 = M1 + M4 - M5 + M7
    C12 = M3 + M5
    C21 = M2 + M4
    C22 = M1 - M2 + M3 + M6
Step 12: Assemble the result matrix C by combining C11, C12, C21, and C22.
Step 13: Return the result matrix C.

CODE :

```
#include <stdio.h>
#include <stdlib.h>


void matrixAddition(int n, int A[][n], int B[][n], int C[][n]) {
        for (int i = 0; i < n; i++)
                for (int j = 0; j < n; j++)
                        C[i][j] = A[i][j] + B[i][j];
}
```

```
void matrixSubtraction(int n, int A[][n], int B[][n], int C[][n]) {
	for (int i = 0; i < n; i++)
		for (int j = 0; j < n; j++)
			C[i][j] = A[i][j] - B[i][j];
}

void matrixMultiplication(int n, int A[][n], int B[][n], int C[][n]) {
	for (int i = 0; i < n; i++) {
		for (int j = 0; j < n; j++) {
			C[i][j] = 0;
			for (int k = 0; k < n; k++) {
				C[i][j] += A[i][k] * B[k][j];
			}
		}
	}
}

void strassenMatrixMultiplication(int n, int A[][n], int B[][n], int C[][n]) {
	if (n <= 64) {
		matrixMultiplication(n, A, B, C);
		Return;
	}

	int newSize = n / 2;
	int A11[newSize][newSize], A12[newSize][newSize], A21[newSize][newSize],
	A22[newSize][newSize];
	int B11[newSize][newSize], B12[newSize][newSize], B21[newSize][newSize],
	B22[newSize][newSize];
	int C11[newSize][newSize], C12[newSize][newSize], C21[newSize][newSize],
	C22[newSize][newSize];
```

```
int P1[newSize][newSize], P2[newSize][newSize], P3[newSize][newSize],
P4[newSize][newSize];
int P5[newSize][newSize], P6[newSize][newSize], P7[newSize][newSize];
int temp1[newSize][newSize], temp2[newSize][newSize];

for (int i = 0; i < newSize; i++) {
        for (int j = 0; j < newSize; j++) {
                A11[i][j] = A[i][j];
                A12[i][j] = A[i][j + newSize];
                A21[i][j] = A[i + newSize][j];
                A22[i][j] = A[i + newSize][j + newSize];

                B11[i][j] = B[i][j];
                B12[i][j] = B[i][j + newSize];
                B21[i][j] = B[i + newSize][j];
                B22[i][j] = B[i + newSize][j + newSize];
        }
}

matrixAddition(newSize, A11, A22, temp1);
matrixAddition(newSize, B11, B22, temp2);
strassenMatrixMultiplication(newSize, temp1, temp2, P1);

matrixAddition(newSize, A21, A22, temp1);
strassenMatrixMultiplication(newSize, temp1, B11, P2);

matrixSubtraction(newSize, B12, B22, temp1);
strassenMatrixMultiplication(newSize, A11, temp1, P3);

matrixSubtraction(newSize, B21, B11, temp1);
strassenMatrixMultiplication(newSize, A22, temp1, P4);
```

```
matrixAddition(newSize, A11, A12, temp1);
strassenMatrixMultiplication(newSize, temp1, B22, P5);

matrixSubtraction(newSize, A21, A11, temp1);
matrixAddition(newSize, B11, B12, temp2);
strassenMatrixMultiplication(newSize, temp1, temp2, P6);

matrixSubtraction(newSize, A12, A22, temp1);
matrixAddition(newSize, B21, B22, temp2);
strassenMatrixMultiplication(newSize, temp1, temp2, P7);

matrixAddition(newSize, P1, P4, temp1);
matrixSubtraction(newSize, temp1, P5, temp2);
matrixAddition(newSize, temp2, P7, C11);

matrixAddition(newSize, P3, P5, C12);

matrixAddition(newSize, P2, P4, C21);

matrixAddition(newSize, P1, P3, temp1);
matrixSubtraction(newSize, temp1, P2, temp2);
matrixAddition(newSize, temp2, P6, C22);

for (int i = 0; i < newSize; i++) {
    for (int j = 0; j < newSize; j++) {
        C[i][j] = C11[i][j];
        C[i][j + newSize] = C12[i][j];
        C[i + newSize][j] = C21[i][j];
        C[i + newSize][j + newSize] = C22[i][j];
    }
}
```

```
}

int main() {
	printf(Rishita Chaubey - A2305221265\n");
	int n;
	printf("Enter the size of the square matrices: ");
	scanf("%d", &n);
	int A[n][n], B[n][n], C[n][n];
	printf("Enter elements of matrix A:\n");
	for (int i = 0; i < n; i++) {
		for (int j = 0; j < n; j++) {
			scanf("%d", &A[i][j]);
		}
	}
	printf("Enter elements of matrix B:\n");
	for (int i = 0; i < n; i++) {
		for (int j = 0; j < n; j++) {
			scanf("%d", &B[i][j]);
		}
	}
	strassenMatrixMultiplication(n, A, B, C);
	printf("Resultant matrix C:\n");
	for (int i = 0; i < n; i++) {
		for (int j = 0; j < n; j++) {
			printf("%d ", C[i][j]);
		}
		printf("\n");
	}

	return 0;
}
```

OUTPUT :

```
Rishita Chaubey - A2305221265

Enter the size of the square matrices: 2
Enter elements of matrix A:
2 3
4 1
Enter elements of matrix B:
1 5
6 7
Resultant matrix C:
20 31
10 27
```

COMPLEXITY : O(n^2.81)

PRACTICAL - 11

AIM : To perform LCS solution using dynamic programming approach

PSEUDO CODE :

LCS(string X, string Y)

Step 1: Initialize a 2D array dp of size (m+1) x (n+1), where m is the length of string X and n is the length of string Y.
Step 2: Initialize the first row and first column of dp with 0s.
Step 3: for i from 1 to m, do steps 4-8
Step 4:   for j from 1 to n, do steps 5-8
Step 5:   if X[i-1] equals Y[j-1], then
Step 6:   Set dp[i][j] = dp[i-1][j-1] + 1.
Step 7:   else,
Step 8:   Set dp[i][j] = max(dp[i-1][j], dp[i][j-1]).
Step 9: Traverse the dp array to reconstruct the LCS.
Step 10: Start from the bottom-right corner (dp[m][n]).
Step 11: While not at the top-left corner (dp[0][0]), do steps 12-15
Step 12:   if X[i-1] equals Y[j-1], then
Step 13:   Append X[i-1] to the LCS.
Step 14:   Move to the diagonal element (dp[i-1][j-1]).
Step 15:   else,
Step 16:   if dp[i-1][j] is greater than dp[i][j-1], move up to the cell above (dp[i-1][j]).
Step 17:   else, move left to the cell to the left (dp[i][j-1]).
Step 18: Reverse the LCS to get the final result.
Step 19: Return the LCS.

CODE :

```
#include <stdio.h>
#include <string.h>


int max(int a, int b) {
    return (a > b) ? a : b;
}
```

```
void lcs(char *X, char *Y, int m, int n) {
    int dp[m + 1][n + 1];
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                dp[i][j] = 0;
            else if (X[i - 1] == Y[j - 1])
                dp[i][j] = dp[i - 1][j - 1] + 1;
            else
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }
    }
    int lcsLength = dp[m][n];
    char lcsSequence[lcsLength + 1];
    lcsSequence[lcsLength] = '\0'; // Null-terminate the string

    int i = m, j = n;
    while (i > 0 && j > 0) {
        if (X[i - 1] == Y[j - 1]) {
            lcsSequence[--lcsLength] = X[i - 1];
            i--;
            j--;
        } else if (dp[i - 1][j] > dp[i][j - 1])
            i--;
        else
            j--;
    }
    printf("Longest Common Subsequence: %s\n", lcsSequence);
}
```

```
int main() {
    printf(Rishita Chaubey - A2305221265\n");
    char X[100], Y[100];
    printf("Enter the first string: ");
    scanf("%s", X);
    printf("Enter the second string: ");
    scanf("%s", Y);
    int m = strlen(X);
    int n = strlen(Y);
    lcs(X, Y, m, n);
    return 0;
}
```

OUTPUT :

```
Rishita Chaubey - A2305221265

Enter the first string: ABCBDAB
Enter the second string: BDCAB
Longest Common Subsequence: BDAB
```

COMPLEXITY :

O(m * n), where m and n are the lengths of the input strings.

PRACTICAL - 12

AIM : To perform knapsack 0/1 using dynamic programming approach

PSEUDO CODE :

KnapsackDP(int capacity, int weights[], int values[], int n)

Step 1: Create a 2D array dp of size (n+1) x (capacity+1).
Step 2: Initialize the first row and the first column of dp with 0s.
Step 3: for i from 1 to n, do steps 4-8
Step 4:   for w from 0 to capacity, do steps 5-8
Step 5:   if weights[i-1] <= w, then
Step 6:   Set dp[i][w] to the maximum of (values[i-1] + dp[i-1][w - weights[i-1]]) and dp[i-1][w].
Step 7:   else,
Step 8:   Set dp[i][w] to dp[i-1][w].
Step 9: Initialize variables maxProfit to 0 and remainingCapacity to capacity.
Step 10: Starting from dp[n][capacity], traverse the dp array to find the selected items:
a. If dp[i][remainingCapacity] is not equal to dp[i-1][remainingCapacity], include item i and subtract its weight from remainingCapacity. Add item i's profit to maxProfit.
b. Move to dp[i-1][remainingCapacity] if dp[i][remainingCapacity] is equal to dp[i-1][remainingCapacity].
Step 11: Return maxProfit and the list of selected items.

CODE :

```
#include <stdio.h>


int max(int a, int b) {
   return (a > b) ? a : b;
}


int knapsack(int capacity, int weights[], int values[], int n) {
   int dp[n + 1][capacity + 1];
   for (int i = 0; i <= n; i++) {
      for (int w = 0; w <= capacity; w++) {
```

```
            if (i == 0 || w == 0)
                dp[i][w] = 0;
            else if (weights[i - 1] <= w)
                dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
            else
                dp[i][w] = dp[i - 1][w];
        }
    }

    return dp[n][capacity];
}

int main() {
printf(Rishita Chaubey - A2305221265\n");

    int capacity, n;
    printf("Enter the knapsack capacity: ");
    scanf("%d", &capacity);
    printf("Enter the number of items: ");
    scanf("%d", &n);
    int weights[n], values[n];
    for (int i = 0; i < n; i++) {
        printf("Enter weight and value for item %d: ", i + 1);
        scanf("%d %d", &weights[i], &values[i]);
    }

    int maxProfit = knapsack(capacity, weights, values, n);
    printf("Maximum profit that can be obtained: %d\n", maxProfit);

    return 0;
}
```

OUTPUT :

```
Rishita Chaubey - A2305221265

Enter the knapsack capacity: 10
Enter the number of items: 4
Enter weight and value for item 1: 2 4
Enter weight and value for item 2: 3 5
Enter weight and value for item 3: 4 7
Enter weight and value for item 4: 5 10
Maximum profit that can be obtained: 19
Items selected: Item 4 Item 3 Item 2 Item 1
```

COMPLEXITY :

O(n * W), where n is the number of items and W is the maximum knapsack capacity