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 2

1. Implement any two of the following using Dynamic programming technique. Give working demonstration with an example.
   a. Assembly Line Scheduling
   b. Matrix Chain Multiplication
   c. Longest Common Subsequence
   d. 0/1 Knapsack

Solution:

b. Matrix Chain Multiplication

Algorithm:

1. Input n: Take input for the number of matrices.
2. Input n+1 dimensions: Store dimensions in an array arr.
3. Initialize DP table: Create a 2D array dp of size n × n with all values set to 0.
4. Base case: Set dp[i][i] = 0 since a single matrix requires no multiplication.
5. Iterate over chain length len: Loop from 2 to n to consider different sub-chain lengths.
6. Loop over starting index i: Iterate from 0 to n - len to define matrix ranges.
7. Compute end index j: Set j = i + len for the ending matrix in the current range.
8. Initialize dp[i][j] = INT_MAX to store the minimum multiplication cost.
9. Try all possible partitions k: Loop k from i+1 to j-1 to find the optimal split.
10. Compute multiplication cost using the formula:

    dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j] + arr[i] * arr[k] * arr[j])
11. Update dp[i][j] with the minimum cost found for multiplying matrices from i to j.
12. Return dp[0][n-1] as the final result, which contains the minimum multiplication cost.

Code:

```
#include <iostream>
#include <vector>

using namespace std;

int matrixMultiplication(vector<int> &arr) {
    int n = arr.size();
    vector<vector<int>> dp(n, vector<int>(n, 0));

    for (int len = 2; len < n; len++) {
        for (int i = 0; i < n - len; i++) {
            int j = i + len;
            dp[i][j] = INT_MAX;

            for (int k = i + 1; k < j; k++) {
                int cost = dp[i][k] + dp[k][j] + arr[i] * arr[k] * arr[j];
                dp[i][j] = min(dp[i][j], cost);
            }
        }
    }

    return dp[0][n - 1];
}

int main() {
    int n;
    cout << "Enter the number of matrices: ";
    cin >> n;

    vector<int> arr(n + 1);
    cout << "Enter the " << n + 1 << " dimensions of matrices: ";

    for (int i = 0; i <= n; i++) {
        cin >> arr[i];
```

```
    }

    cout << "Minimum cost of matrix multiplication: " << matrixMultiplication(arr) << endl;
    return 0;
}
```

Input:
(Same problem as class)

```
Enter the number of matrices: 4
Enter the 5 dimensions of matrices: 13
5
89
3
34
```

Output:

```
Minimum cost of matrix multiplication: 2856
```

Time Complexity Analysis:

Time complexity analysis of matrix chain multiplication:

$\frac{n(n+1)}{2}$ elements inserted

$\leq n^2$

calculating all ~~but~~, meaning each element is to be calculated, taking at most n multiplications

Meaning,

$n^2 \times n$

$= \Theta(n^3)$

c. Longest Common Subsequence

Solution:

Algorithm

1. Take input strings s1 and s2.

2. Create a 2D vector called memo with dimensions (length of s1 + 1) by (length of s2 + 1) and initialize all values to -1.

3. Define a recursive function lcs that takes s1, s2, their lengths, and the memo table as input.

4. If either string has length 0, return 0.

5. If the value at memo[m][n] is not -1, return that value to avoid redundant calculations.

6. If the last characters of both strings match, add 1 to the result of lcs called on the remaining substrings.

7. If the last characters do not match, return the maximum value between lcs called on (m, n-1) and (m-1, n).

8. Store the computed value in memo[m][n] before returning it.

9. Print the result of calling lcs on the input strings.

Code:

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

int lcs(string &s1, string &s2, int m, int n, vector<vector<int>> &memo) {


    if (m == 0 || n == 0)
        return 0;

    if (memo[m][n] != -1)
        return memo[m][n];
```

```
    if (s1[m - 1] == s2[n - 1])
        return memo[m][n] = 1 + lcs(s1, s2, m - 1, n - 1, memo);

    return memo[m][n] = max(lcs(s1, s2, m, n - 1, memo), lcs(s1, s2, m - 1, n, memo));
}

int main() {
    string s1;
    string s2;

    cout<< "Enter first string: ";
    cin>>s1;
    cout<< "\nEnter second string: ";
    cin>>s2;
    cout<<"\n";
    int m = s1.length();
    int n = s2.length();
    vector<vector<int>> memo(m + 1, vector<int>(n + 1, -1));
    cout << lcs(s1, s2, m, n, memo) << endl;
    return 0;
}
```

Input:
(Same as class)

```
Enter first string: abcb

Enter second string: bdcab
```

Output:

```
3
```

Time complexity analysis:

Time complexity analysis of Longest Common Subsequence

By algorithm,

$$T(m,n) = T(m-1,n) + T(m,n-1) - T(m-1,n-1) + O(1)$$

Using ~~bottom-up~~ top-down computation,

$$T(m,n) = O(1)$$

the table of $(m \times n)$ is formed,

meaning $m \times n$ subproblems,

so, time complexity is $O(m \times n)$

2. Implement N-Queens problem and analyse its time complexity using backtracking.

Solution:

Algorithm

1. Start placing queens row by row.
2. Check if the column and diagonals are safe.
3. Place a queen and mark the column and diagonals.
4. Move to the next row recursively.
5. If all queens are placed, return the position.
6. If placement fails, backtrack and try another column.
7. If no solution is found, return -1.

Code:

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

int placeQueens(int i, vector<int> &cols, vector<int> &leftDiagonal,
                vector<int> &rightDiagonal, vector<int> &cur) {
    int n = cols.size();

    if(i == n) return 1;

    for(int j = 0; j < n; j++){

        if(cols[j] || rightDiagonal[i + j] ||
                leftDiagonal[i - j + n - 1])
            continue;

        cols[j] = 1;
        rightDiagonal[i+j] = 1;
        leftDiagonal[i - j + n - 1] = 1;
        cur.push_back(j+1);

        if(placeQueens(i + 1, cols, leftDiagonal, rightDiagonal, cur))
            return 1;

        cur.pop_back();
        cols[j] = 0;
        rightDiagonal[i+j] = 0;
        leftDiagonal[i - j + n - 1] = 0;
    }
    return 0;
}

vector<int> nQueen(int n) {
```

```
    vector<int> cols(n, 0);
    vector<int> leftDiagonal(n*2, 0);
    vector<int> rightDiagonal(n*2, 0);
    vector<int> cur;

    if(placeQueens(0, cols, leftDiagonal, rightDiagonal, cur))
        return cur;

    else return {-1};
}

int main() {
    int n;
    cout<<"Enter Number of Queens: ";
    cin>>n;
    cout<<"\n";
    vector<int> ans = nQueen(n);
    for(auto i: ans){
        cout << i << " ";
    }
    return 0;
}
```

Input:

```
Enter Number of Queens: 8
```

Output:

```
1 5 8 6 3 7 2 4 %
```

Time Complexity Analysis:

Time complexity Analysis for N-Queens Problem

The backtracking approach explores each row & tries placing a queen in all valid columns.

$$T(n) = nT(n-1) + O(n)$$

using Master's method,

$a = n$
$b = 1$
$k = 1$

here,

$\log_b a = \log_1 n = n$

$k = 1$

$\log_b a > k$

CASE-I

$n^{\log_1 n} = n^n$

$= n!$

$$O(n!)$$

3. Implement Graph Coloring Problem and analyse its time complexity using Backtracking.

Solution:

Algorithm:

1. Take number of vertices, adjacency matrix, and number of colors as input.
2. Initialize a color array with 0.
3. Define a function to check if a color assignment is valid.

4. Use recursion to assign colors to vertices.
5. If all vertices are colored, print the solution.
6. If no valid coloring is found, return failure.

Code:

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

bool isSafe(int v, vector<vector<int>> &graph, vector<int> &color, int c, int V) {
    for (int i = 0; i < V; i++) {
        if (graph[v][i] && color[i] == c)
            return false;
    }
    return true;
}

bool graphColoringUtil(vector<vector<int>> &graph, int m, vector<int> &color, int v, int V) {
    if (v == V)
        return true;

    for (int c = 1; c <= m; c++) {
        if (isSafe(v, graph, color, c, V)) {
            color[v] = c;
            if (graphColoringUtil(graph, m, color, v + 1, V))
                return true;
            color[v] = 0;
        }
    }
    return false;
}

bool graphColoring(vector<vector<int>> &graph, int m, int V) {
    vector<int> color(V, 0);
    if (!graphColoringUtil(graph, m, color, 0, V)) {
```

```
        cout << "Solution does not exist" << endl;
        return false;
    }

    cout << "Solution Exists: Assigned Colors: ";
    for (int i = 0; i < V; i++)
        cout << color[i] << " ";
    cout << endl;
    return true;
}

int main() {
    int V, m;
    cout << "Enter the number of vertices: ";
    cin >> V;

    vector<vector<int>> graph(V, vector<int>(V, 0));

    cout << "Enter the adjacency matrix (" << V * V << " values row-wise):" << endl;
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            cin >> graph[i][j];

    cout << "Enter the number of colors: ";
    cin >> m;

    graphColoring(graph, m, V);
    return 0;
}
```

Input:

```
Enter the number of vertices: 3
Enter the adjacency matrix (9 values row-wise):
0
1
1
1
0
1
1
1
0
Enter the number of colors: 3
```

Output:

```
Solution Exists: Assigned Colors: 1 2 3
```

Time Complexity Analysis:

Time Complexity Analysis for Graph Coloring

$$T(V) = m \cdot T(V-1) + O(V)$$

m = number of colors
V = number of vertices
$O(V)$ = time taken to check if color assignment is valid.

$a = m$
$b = 1$
$f(V) = O(V)$
$d = 1$

$$T(V) = mT(V-1) + OV$$
$$T\,V\,(mT(V-2) + O(V-1)) + O(V)$$
$$= m^2T(V-2) + mO(V-1) + O(V)$$

till $V = 1$

$$T(V) = m^{V-1}T(1) + O(Vm^{V-1})$$
$$T(V) = O(m^V)$$
$$= O(m^V)$$