**Assignment – 3**

1. The 0-1 knapsack problem is a classic optimization problem in computer science and combinatorial optimization. It belongs to a class of problems known as NP-hard, meaning there is no known polynomial-time solution to solve it optimally in the general case.

In this problem, we are given a set of items, each with a weight and a value, and a knapsack with a maximum weight capacity. The goal is to determine the most valuable combination of items to include in the knapsack without exceeding its capacity. The term "0-1" indicates that for each item, we can either include it in the knapsack (by setting its binary decision variable to 1) or exclude it (by setting its binary decision variable to 0), but we cannot include a fraction of an item.

The goal is to find the combination of x\_i values that maximizes the total value of the items in the knapsack while ensuring that the total weight does not exceed the capacity of the knapsack.

Solving the 0-1 knapsack problem efficiently typically involves dynamic programming techniques, such as memoization or tabulation, to avoid redundant calculations and optimize the solution. Additionally, approximation algorithms or heuristic methods may be used to find near-optimal solutions in polynomial time for large problem instances.

# include <stdio.h>

# define max\_choice 5

# define max\_capacity 24

int diamond[][2] = {{2,3},{3,6},{5,7},{6,12},{8,17}};

int memo[max\_choice+1][max\_capacity+1];

int max(int,int);

int knapsack(int,int);

int main()

{

    int total\_choice=5;

    int capacity;

    printf("\nEnter the total capacity : ");

    scanf("%d",&capacity);

    for(int i=0; i<=max\_choice; i++)

    {

        for(int j=0; j<=max\_capacity; j++)

        {

            memo[i][j] = -1;

        }

    }

    int dia = knapsack(total\_choice,capacity);

    printf("Total prize : %d\n",dia);

    return 0;

}

int max(int n, int m)

{

    if(n>m)

    {

        return n;

    }

    else

    {

        return m;

    }

}

int knapsack(int i,int j)

{

    if(memo[i][j]!=-1)

    {

        return memo[i][j];

    }

    if(j==0 || i==0)

    {

        return 0;

    }

    else if(diamond[i-1][0]<=j)

    {

        int mx = max(knapsack(i-1,j-diamond[i-1][0])+diamond[i-1][1],knapsack(i-1,j));

        memo[i][j] = mx;

        return mx;

    }

    else

    {

        int mx = knapsack(i-1,j);

        memo[i][j] = mx;

        return mx;

    }

}

Output :-

PS D:\DAA> gcc Diamond.cpp

PS D:\DAA> ./a.exe

Enter the total capacity : 9

Total prize : 18

PS D:\DAA> ./a.exe

Enter the total capacity : 4

Total prize : 6

PS D:\DAA> ./a.exe

Enter the total capacity : 5

Total prize : 9

PS D:\DAA>

2. The change-making problem is a classic problem in computer science and dynamic programming that involves finding the minimum number of coins (or bills) required to make change for a given amount of money. It's also known as the "coin change problem."

The problem statement is as follows: Given a set of denominations of coins (or bills) and a target amount of money, determine the minimum number of coins needed to make change for the target amount. The denominations are assumed to be available in unlimited quantities.

For example, consider a set of coin denominations {1, 5, 10, 25} and a target amount of 30 cents. The minimum number of coins needed to make change for 30 cents would be two 10-cent coins and one 10-cent coin, totaling three coins.

The goal is to minimize the total number of coins used while ensuring that their total value equals the target amount A .

Dynamic programming is a commonly used technique to solve the change-making problem efficiently. The problem exhibits optimal substructure and overlapping subproblems, making it well-suited for dynamic programming.

The dynamic programming approach typically involves constructing a table to store the minimum number of coins required to make change for each possible target amount from 0 to the target amount A. The table is filled iteratively, with each entry representing the minimum number of coins needed to make change for the corresponding amount. The final entry of the table corresponds to the minimum number of coins needed to make change for the target amount A.

By solving smaller subproblems and using their solutions to build up to the solution of the larger problem, dynamic programming efficiently finds the optimal solution to the change-making problem.

# include <stdio.h>

# define total\_coin 6

int coin[] = {1,2,5,10,20,25};

int changeProblem(int,int,int [][total\_coin+1]);

int min(int,int);

int main()

{

    int change;

    printf("\nEnter the change : ");

    scanf("%d",&change);

    int memo[change+1][total\_coin+1];

    for(int i=0; i<=change; i++)

    {

        for(int j=0; j<=total\_coin; j++)

        {

            memo[i][j] = -1;

        }

    }

    int min\_coin = changeProblem(total\_coin,change,memo);

    printf("\nMinimum coin : %d",min\_coin);

    return 0;

}

int min(int n,int m)

{

    if(n<m)

    {

        return n;

    }

    else

    {

        return m;

    }

}

int changeProblem(int i,int j,int memo[][total\_coin+1])

{

    if(memo[j][i]!=-1)

    {

        return memo[j][i];

    }

    if(j==0)

    {

        return 0;

    }

    else if(i==0)

    {

        return coin[5]+1;

    }

    else if(coin[i-1]<=j)

    {

        int mn = min(changeProblem(i-1,j,memo),changeProblem(i,j-coin[i-1],memo)+1);

        memo[j][i] = mn;

        return mn;

    }

    else

    {

        int mn = changeProblem(i-1,j,memo);

        memo[j][i] = mn;

        return mn;

    }

}

Output :-

PS D:\DAA> gcc changeProblem.cpp

PS D:\DAA> ./a.exe

Enter the change : 29

Minimum coin : 3

PS D:\DAA> ./a.exe

Enter the change : 47

Minimum coin : 3

PS D:\DAA> ./a.exe

Enter the change : 23

Minimum coin : 3

PS D:\DAA> ./a.exe

Enter the change : 98

Minimum coin : 6

PS D:\DAA>

3. Chain matrix multiplication, also known as matrix chain multiplication, is a classic problem in computer science and dynamic programming that involves finding the most efficient way to multiply a sequence of matrices. The goal is to determine the order of multiplication that minimizes the total number of scalar multiplications required to compute the product.

Given a sequence of n matrices to multiply, A\_1, A\_2, ..., A\_n , where the dimensions of matrix A\_i are d\_{i-1} \* d\_i for i = 1, 2, ..., n , the problem is to find the optimal order of multiplication to minimize the total number of scalar multiplications needed.

For example, consider three matrices with dimensions A\_1: 10 \* 20 , A\_2: 20 \* 30 , and A\_3: 30 \* 40 . Multiplying them in the order (A\_1 \* A\_2) \* A\_3 requires (10 \* 20 \* 30) + (10 \* 30 \* 40) = 18000 + 12000 = 30000 scalar multiplications. However, if we multiply them in the order A\_1 \* (A\_2 \* A\_3) , the number of scalar multiplications reduces to (20 \* 30 \* 40) + (10 \* 20 \* 40) = 24000 + 8000 = 32000 .

The key insight in solving the chain matrix multiplication problem is that the order of multiplication significantly affects the total number of scalar multiplications required. Additionally, the problem exhibits optimal substructure, meaning that the optimal solution to a problem can be constructed from the optimal solutions of its subproblems.

Dynamic programming provides an efficient solution approach to the chain matrix multiplication problem by exploiting its optimal substructure and overlapping subproblems properties. The dynamic programming algorithm typically involves constructing a table to store the minimum number of scalar multiplications needed to compute the product of matrix chains of different lengths. The table is filled iteratively, with each entry representing the minimum number of scalar multiplications required for multiplying a matrix chain of a certain length.

By solving smaller subproblems and using their solutions to build up to the solution of the larger problem, dynamic programming efficiently finds the optimal order of multiplication that minimizes the total number of scalar multiplications needed to compute the product of the matrix chain. This approach ensures that unnecessary duplicate computations are avoided, leading to an optimal solution in polynomial time.

# include <stdio.h>

# define m 50

int min(int,int);

int matrixMultiplication(int,int,int [],int [m][m]);

int main()

{

    int n;

    printf("\nEnter the number of matrix : ");

    scanf("%d",&n);

    int arr[n+1];

    int memo[m][m];

    for(int i=0; i<=n+1; i++)

    {

        for(int j=0; j<=n+1; j++)

        {

            memo[i][j] = -1;

        }

    }

    printf("Enter the order...\n\n");

    for(int i=0; i<=n; i++)

    {

        scanf("%d",&arr[i]);

    }

    int min = matrixMultiplication(1,n+1,arr,memo);

    printf("\nMin : %d",min);

    return 0;

}

int min(int l,int k)

{

    if(l<k)

    {

        return l;

    }

    else

    {

        return k;

    }

}

int matrixMultiplication(int i,int j,int arr[],int memo[m][m])

{

    if(memo[i][j]==-1)

    {

        if(i==j-1)

        {

            memo[i][j] = 0;

        }

        else

        {

            int mn=10000;

            for(int k=i+1; k<j; k++)

            {

                mn = min(mn,matrixMultiplication(i,k,arr,memo)+matrixMultiplication(k,j,arr,memo)+(arr[i-1]\*arr[j-1]\*arr[k-1]));

            }

            memo[i][j] = mn;

        }

        return memo[i][j];

    }

   return memo[i][j];

}

Output :-

PS D:\DAA> gcc chainMatrix.cpp

PS D:\DAA> ./a.exe

Enter the number of matrix : 5

Enter the order...

8 7 4 2 4 3

Min : 240

Enter the number of matrix : 4

Enter the order...

7 9 3 11 6

Min : 513

2. The Longest Common Subsequence (LCS) problem is a classic computer science problem that involves finding the longest subsequence common to two sequences (strings) given as input. A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

For example, consider two sequences:

Sequence 1: ABCBDAB

Sequence 2: BDCAB

The longest common subsequence between these two sequences is "BCAB," with a length of 4.

The LCS problem is commonly solved using dynamic programming. The key idea behind dynamic programming for this problem is to build a table (often called an LCS table) where each cell represents the length of the longest common subsequence between substrings of the input sequences.

The LCS table is typically filled iteratively, starting from the smallest subproblems (i.e., substrings of length 1) and building up to the entire sequences. At each step, the table is updated based on the previously computed values.

The LCS table can be represented as a two-dimensional array. The rows correspond to characters of the first sequence, and the columns correspond to characters of the second sequence. Each cell (i, j) in the table represents the length of the LCS between the substrings ending at positions i and j in the two sequences.

Once the LCS table is filled, the length of the longest common subsequence is found in the bottom-right cell of the table. Additionally, the actual longest common subsequence itself can be reconstructed by backtracking through the table from the bottom-right cell to the top-left cell.

The LCS problem has various applications in bioinformatics, text comparison, version control systems, and many other areas where identifying similarities between sequences is important. Its efficient solution using dynamic programming makes it a fundamental problem in algorithm design and computer science.

# include <stdio.h>

# include <string.h>

# define x 50

# define y 50

int lcs(int,int,char [],char [],int[x][y]);

void printLCS(int,int,char [],char [],int [x][y]);

int max(int,int);

int temp = 0;

int main()

{

    char a[50],b[50];

    printf("\n1st String : ");

    gets(a);

    printf("\n2nd String : ");

    gets(b);

    int memo[x][y];

    for(int i=0; i<=strlen(a); i++)

    {

        for(int j=0; j<=strlen(b); j++)

        {

            memo[i][j] = -1;

        }

    }

    int max = lcs(strlen(a),strlen(b),a,b,memo);

    printf("Max : %d",max);

    printf("\n");

    printLCS(strlen(a),strlen(b),a,b,memo);

    return 0;

}

//  Line glass cutting , assembly line shceduling

int max(int a,int b)

{

    int mx = (a>b) ? a : b;

    return mx;

}

int lcs(int i,int j,char a[],char b[],int memo[x][y])

{

    if(memo[i][j]!=-1)

    {

        return memo[i][j];

    }

    if(i==0 || j==0)

    {

        memo[i][j]=0;

    }

    else if(a[i-1]==b[j-1])

    {

        memo[i][j] = lcs(i-1,j-1,a,b,memo)+1;

    }

    else

    {

        memo[i][j] = max(lcs(i-1,j,a,b,memo),lcs(i,j-1,a,b,memo));

    }

    return memo[i][j];

}

void printLCS(int i,int j,char a[],char b[],int memo[x][y])

{

    if(i==0 || j==0)

    {

        return;

    }

    if(a[i-1]==b[j-1])

    {

        printLCS(i-1,j-1,a,b,memo);

        printf("%c",a[i-1]);

    }

    else if(memo[i-1][j] > memo[i][j-1])

    {

        printLCS(i-1,j,a,b,memo);

    }

    else

    {

        printLCS(i,j-1,a,b,memo);

    }

}

Output :-

PS D:\DAA> gcc LongestCommonSub.cpp

PS D:\DAA> ./a.exe

1st String : hellomsu

2nd String : easyforu

Max : 3

esu

PS D:\DAA> ./a.exe

1st String : khushi

2nd String : shikhu

Max : 3

shi

PS D:\DAA> ./a.exe

1st String : have a nice day

2nd String : a day is nice

Max : 8

a a nice

PS D:\DAA>