Lab No: 17 **Date:** 2081/12/29

Write a program in C implementing Matrix Chain Multiplication.

Theory:

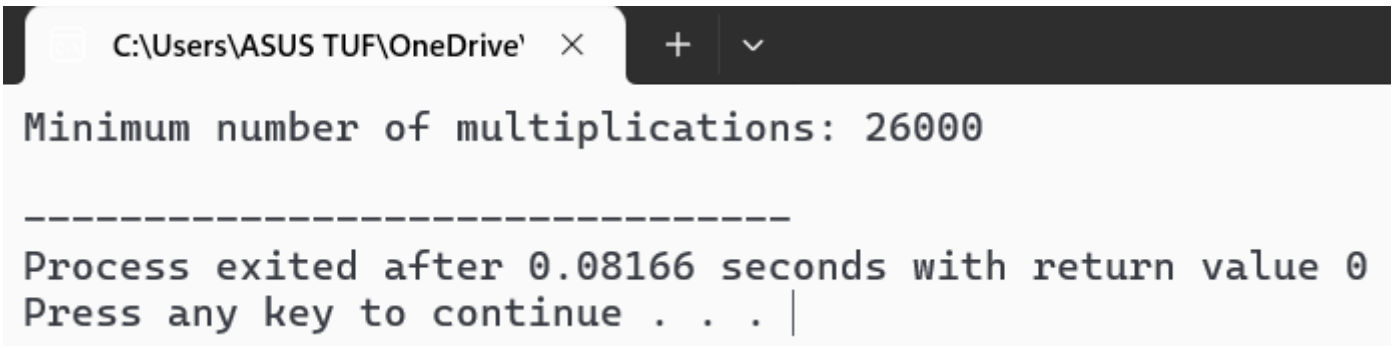
This problem focuses on finding the most efficient way to multiply a sequence of matrices. Matrix multiplication is associative, so different parenthesizations can lead to different computational costs. The goal is to determine the order of multiplication that minimizes the total number of scalar operations. It is typically solved using Dynamic Programming.

Source Code:

```
#include <stdio.h>
#include <limits.h>
void matrixChainOrder(int p[], int n) {
    int m[n][n],i,j,k,L;
    for (i = 1; i < n; i++)
        m[i][i] = 0;

    for (L = 2; L < n; L++) {
        for (i = 1; i < n - L + 1; i++) {
            int j = i + L - 1;
            m[i][j] = INT_MAX;
            for (k = i; k < j; k++) {
                int q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }
    printf("Minimum number of multiplications: %d\n", m[1][n-1]);
}
int main() {
    int arr[] = {40, 20, 30, 10, 30};
    int n = sizeof(arr)/sizeof(arr[0]);
    matrixChainOrder(arr, n);
    return 0;
}
```

OUTPUT:



Lab No: 18 **Date:** 2081/12/29

Write a program in C implementing String Edit Distance.

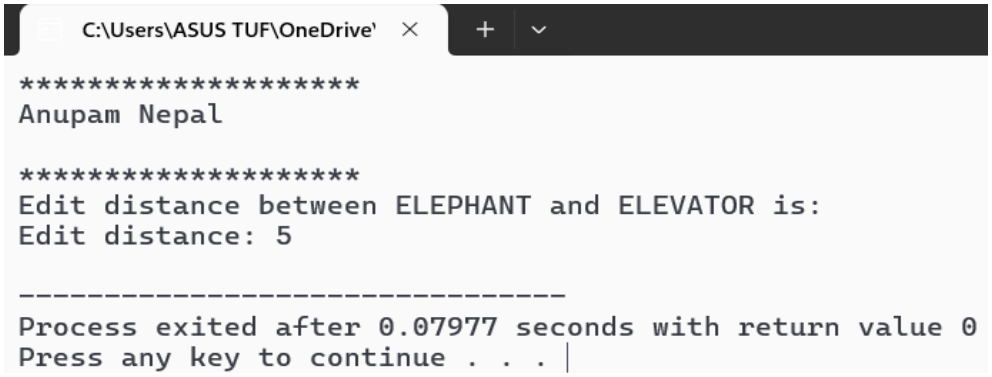
Theory:

The Edit Distance problem determines the minimum number of operations (insertions, deletions, and substitutions) required to convert one string into another.

Source Code:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int min(int a, int b, int c) {
   if (a < b && a < c) return a;
   if (b < c) return b;
   return c;
}
int editDistance(char *str1, char *str2) {
   int m = strlen(str1), n = strlen(str2), i, j;
   int dp[m+1][n+1];
   for (i = 0; i <= m; i++)
      for (j = 0; j <= n; j++) {
         if (i == 0) dp[i][j] = j;
         else if (j == 0) dp[i][j] = i;
         else if (str1[i-1] == str2[j-1])
            dp[i][j] = dp[i-1][j-1];
         else
            dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]);
      }
   return dp[m][n];
}
int main() {
   char str1[] = "ELEPHANT", str2[] = "ELEVATOR";
   printf("Edit distance between %s and %s is:\n",str1,str2);
   printf("Edit distance: %d\n", editDistance(str1, str2));
   return 0;
}
```

OUTPUT:



Madan Bhandari Memorial College

Lab No: 19 **Date:** 2082/1/03

Write a program in C implementing 0/1 Knapsack Problem.

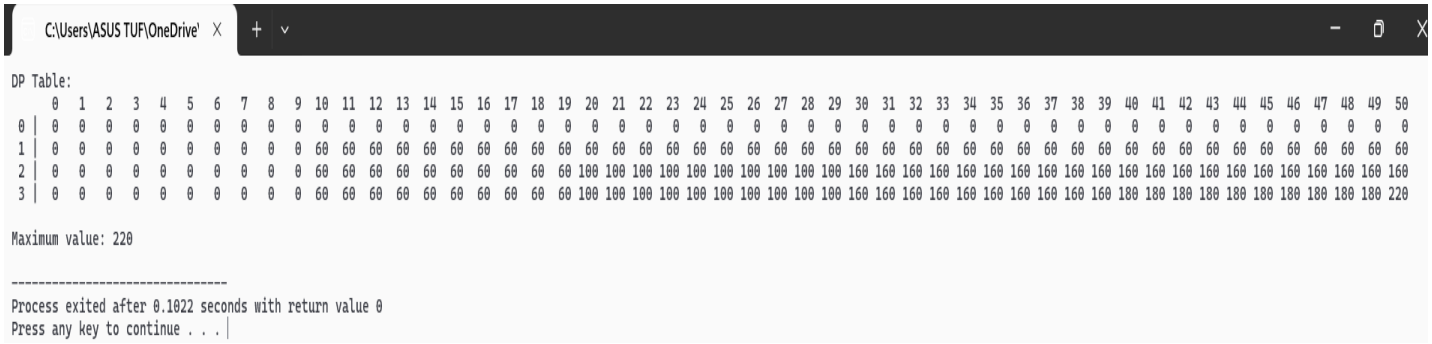
Theory:

Given a set of items with individual weights and values, the goal is to determine the maximum total value that can be placed in a knapsack of fixed capacity without splitting any item. Each item is either included or excluded (hence 0/1). This is a fundamental NP-complete problem, commonly solved using Dynamic Programming.

Source Code:

```
#include <stdio.h>
int max(int a, int b) { return (a > b) ? a : b; }
int knapsack(int W, int wt[], int val[], int n) {
   int dp[n+1][W+1], i, w;
   for (i = 0; i <= n; i++) {
      for (w = 0; w <= W; w++) {
         if (i == 0 || w == 0)
            dp[i][w] = 0;
         else if (wt[i-1] <= w)
            dp[i][w] = max(val[i-1] + dp[i-1][w - wt[i-1]], dp[i-1][w]);
         else
            dp[i][w] = dp[i-1][w];
      }
   }
   printf("\nDP Table:\n    ");
   for (w = 0; w <= W; w++)
      printf("%3d ", w);
   printf("\n");
   for (i = 0; i <= n; i++) {
      printf("%2d |", i);
      for (w = 0; w <= W; w++)
         printf("%3d ", dp[i][w]);
      printf("\n");
   }
   return dp[n][W];
}
int main() {
   int val[] = {60, 100, 120};
   int wt[] = {10, 20, 30};
   int W = 50;
   int n = sizeof(val) / sizeof(val[0]);
   int result = knapsack(W, wt, val, n);
   printf("\nMaximum value: %d\n", result);
   return 0;
}
```

OUTPUT:


C:\Users\ASUS TUF\OneDrive'
DP Table:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 | 0 0 0 0 0 0 0 0 0 0 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60
2 | 0 0 0 0 0 0 0 0 0 0 60 60 60 60 60 60 60 60 60 60 100 100 100 100 100 100 100 100 100 100 160 160 160 160 160 160 160 160 160 160 160 160 160 160 160 160 160 160 160 160 160
3 | 0 0 0 0 0 0 0 0 0 0 60 60 60 60 60 60 60 60 60 60 100 100 100 100 100 100 100 100 100 100 160 160 160 160 160 160 160 160 160 160 180 180 180 180 180 180 180 180 180 180 220
Maximum value: 220
Process exited after 0.1022 seconds with return value 0
Press any key to continue . . .

Lab No: 20 **Date:** 2082/1/03

Write a program in C implementing Sub-Set Sum Problem.

Theory:

Given a set of integers, determine whether a subset exists whose sum equals a given value. This is a fundamental NP-complete problem, often solved using recursive or Dynamic Programming approaches. It's a special case of the Knapsack problem and important in cryptography and decision making.

Source Code:

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 100

void printSubset(int subset[], int size) {
   printf("{ ");
   int i;
   for (i = 0; i < size; i++)
      printf("%d ", subset[i]);
   printf("}\n");
}

void printAllSubsets(int arr[], int i, int sum, int subset[], int subsetSize) {
   if (sum == 0) {
      printSubset(subset, subsetSize);
      return;
   }
   if (i < 0 || sum < 0)
      return;

   printAllSubsets(arr, i - 1, sum, subset, subsetSize);

   subset[subsetSize] = arr[i];
   printAllSubsets(arr, i - 1, sum - arr[i], subset, subsetSize + 1);
}

int main() {
   int arr[] = {3, 34, 4, 12, 5, 2};
   int n = sizeof(arr) / sizeof(arr[0]);
   int sum = 9, i, j;
   printf("********************\n");
   printf("\tAnupam\n");
         printf("********************\n");

   bool dp[n+1][sum+1];

   for (i = 0; i <= n; i++)
      dp[i][0] = true;
```

```
    for (j = 1; j <= sum; j++)
        dp[0][j] = false;

    for (i = 1; i <= n; i++) {
        for (j = 1; j <= sum; j++) {
            if (arr[i-1] > j)
                dp[i][j] = dp[i-1][j];
            else
                dp[i][j] = dp[i-1][j] || dp[i-1][j - arr[i-1]];
        }
    }

    printf("Input array: ");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\nTarget sum: %d\n", sum);

    if (!dp[n][sum]) {
        printf("No subset with given sum found.\n");
    } else {
        printf("Subsets with sum %d:\n", sum);
        int subset[MAX];
        printAllSubsets(arr, n - 1, sum, subset, 0);
    }
    return 0;
}
```

OUTPUT: