<u>Assignment-4</u>

Name – Ujjwal Lehri
Roll No. – 60216403224
Course – B.Tech CSE 4th sem

---

1. **Implement matrix chain order, print optimal parameters, recursive and memoised matrix chains.**
2. **Write a note on the features of dynamic programming.**

---

1) matrix chain multiplication:

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

#define MIN(a,b)((a)<(b)? (a):(b))
int name = 65; //ascii of A

int MatrixChainOrder_recursive(int *matrices, int i, int j, int **s);
int MatrixChainOrder_DP(int *matrices, int i, int j, int **dp, int **s);
int MatrixChainOrder_iterative(int *matrices, int size, int **s);
void printOptimalParens(int **s, int i, int j);

int main(){
    LARGE_INTEGER frequency, start, end;
    QueryPerformanceFrequency(&frequency);
    int x;
    double time_taken;

    //20 matrices
    int Matrices[21] = {12, 20, 18, 24, 30, 28, 35, 40, 38, 42, 50, 48, 56, 60, 55, 65, 70, 68, 75, 80, 82};
    int n = sizeof(Matrices) / sizeof(Matrices[0]);

    int **dp = (int **)calloc(n, sizeof(int *));
    int **s = (int **)calloc(n, sizeof(int *));
    for (int i = 0; i < n; i++) {
        s[i] = (int *)calloc(n, sizeof(int));
        dp[i] = (int *)calloc(n, sizeof(int));
        for(int j = 0; j<n; j++) dp[i][j] =-1;
    }
```

```c
    /*----------------------------------MatrixChainOrder_recursive----------------------------------*/

    QueryPerformanceCounter(&start);
      x =  MatrixChainOrder_recursive(Matrices, 1, n-1, s);
    QueryPerformanceCounter(&end);

    time_taken = (double)(end.QuadPart- start.QuadPart) * 1e9 / frequency.QuadPart;

    printf("\nMinimum number of multiplications using recursion: %d\n", x);
    printf("optimal Parentheses: ");
    name = 65;
    printOptimalParens(s, 1, n-1);
    printf("\ntime taken: %20.2lf ns\n\n", time_taken);

    /*----------------------------------MatrixChainOrder_DP----------------------------------*/

    QueryPerformanceCounter(&start);
      x =  MatrixChainOrder_DP(Matrices, 1, n-1, dp, s);
    QueryPerformanceCounter(&end);
    time_taken = (double)(end.QuadPart- start.QuadPart) * 1e9 / frequency.QuadPart;

    printf("Minimum number of multiplications using DP (recursive): %d\n", x);
    printf("optimal Parentheses: ");
    name = 65;
    printOptimalParens(s, 1, n-1);
    printf("\ntime taken: %20.2lf ns\n\n", time_taken);

    /*----------------------------------MatrixChainOrder_iterative----------------------------------*/

    QueryPerformanceCounter(&start);
      x =  MatrixChainOrder_iterative(Matrices, n, s);
    QueryPerformanceCounter(&end);
    time_taken = (double)(end.QuadPart- start.QuadPart) * 1e9 / frequency.QuadPart;

    printf("Minimum number of multiplications using DP (iterative): %d\n", x);
    printf("optimal Parentheses: ");
    name = 65;
    printOptimalParens(s, 1, n-1);
    printf("\ntime taken: %20.2lf ns\n\n", time_taken);

    return 0;
}
```

```
/**********************************************************************/
/*                          recursive method                          */
/**********************************************************************/

int MatrixChainOrder_recursive(int *matrices, int i, int j, int **s){
   if(i < 1 || i>j) return-1;

   if(i == j) return 0;

   int cost = INT_MAX;
   for(int k = i; k<j; k++){
      int Newcost = MatrixChainOrder_recursive(matrices, i, k, s) +
MatrixChainOrder_recursive(matrices, k+1, j, s) + matrices[i-1]*matrices[k]*matrices[j];

      if(Newcost < cost){
          cost = Newcost;
          s[i][j] = k;
      }
   }

   return cost;
}

/**********************************************************************/
/*                 recursive method with Memoization                  */
/**********************************************************************/

int MatrixChainOrder_DP(int *matrices, int i, int j, int **dp, int **s){
   if(i < 1 || i>j) return-1;

   if(i == j) return 0;
   if(dp[i][j] !=-1){
      return dp[i][j];
   }
   int cost = INT_MAX;
   for(int k = i; k<j; k++){
      int Newcost = MatrixChainOrder_DP(matrices, i, k, dp, s) + MatrixChainOrder_DP(matrices, k+1, j,
dp, s) + matrices[i-1]*matrices[k]*matrices[j];

      if(Newcost < cost){
          cost = Newcost;
          dp[i][j] = cost;
          s[i][j] = k;
      }
   }
   return cost;
}
```

```
/*************************************************************************/
/*                          iterative method                             */
/*************************************************************************/

int MatrixChainOrder_iterative(int *matrices, int size, int **s){

   int dp[size][size];

   for(int i = 1; i<size;i++){
      dp[i][i] = 0;
   }

   // chain length L2 = A x B, L3 = A x B x C, etc
   for(int L = 2; L<size; L++){ // minimum chain length can be 2: multiplication of 2 matrix

      // starting point for combinations
      for(int i = 1; i < size-L+1; i++){
         int j = i+L-1;  // end index

         dp[i][j] = INT_MAX;
         for(int k=i; k<j; k++){
            int cost = dp[i][k] + dp[k+1][j] + matrices[i-1] * matrices[k] * matrices[j];
            if(cost < dp[i][j]){
               dp[i][j] = cost;
               s[i][j] = k;
            }
         }
      }
   }
   return dp[1][size-1];
}

/*************************************************************************/
/*                    printing Optimal Parentheses                       */
/*************************************************************************/

void printOptimalParens(int **s, int i, int j){
   if (i==j) {
      printf("%c", (char) name);
      name++;
   }
   else{
      printf("(");
      printOptimalParens(s, i, s[i][j]);
      printOptimalParens(s, s[i][j]+1, j);
      printf(")");
   }
}
```

Output:

```
PS C:\Users\Ujjwal\Desktop\C\DAA_assignment\MatrixChain> cd "c:\Users\Ujjwal\Desktop\

Minimum number of multiplications using recursion: 626892
optimal Parentheses: ((((((((((((((((((AB)C)D)E)F)G)H)I)J)K)L)M)N)O)P)Q)R)S)T)
time taken:       13509514900.00 ns

Minimum number of multiplications using DP (recursive): 626892
optimal Parentheses: ((((((((((((((((((AB)C)D)E)F)G)H)I)J)K)L)M)N)O)P)Q)R)S)T)
time taken:             48600.00 ns

Minimum number of multiplications using DP (iterative): 626892
optimal Parentheses: ((((((((((((((((((AB)C)D)E)F)G)H)I)J)K)L)M)N)O)P)Q)R)S)T)
time taken:             26300.00 ns
```

2) Write a note on the features of dynamic programming.

Dynamic Programming (DP) is an algorithmic strategy used to solve optimization problems that have multiple possible solutions. Each solution is associated with a value, and the goal is to find a solution with the optimal (minimum or maximum) value. DP achieves this by breaking the problem into overlapping subproblems, solving each subproblem once, and storing the results to avoid redundant computations.

- **Optimal Substructure:** A problem can be solved optimally by combining optimal solutions of subproblems.
- **Overlapping Subproblems:** Subproblems overlap, so the results can be reused and used again without the need of recalculating the result.
- **Memoization (Top-Down):** A top-down technique in which the results of function calls are stored to prevent repeated computations.
- **Tabulation (Bottom-up):** Bottom-up approach that fills in a table iteratively without recursion.
- **Time Optimization:** Reduces exponential time complexity to polynomial or linear time.
- **Deterministic Results:** For the same input, dynamic programming always yields the same result.