Practical 5
Implementation of Strassen's Fast Multiplication of Matrices Algorithm

GAHAN M. SARAIYA, 18MCEC10

18mcec10@nirmauni.ac.in

I. INTRODUCTION

Aim of this practical is to implement C program to calculate exponential value for number using divide and conquer.

II. IMPLEMENTATION

I. Utility *utility.h*

```c
//
// Created by jarvis on 17/8/18.
//

#ifndef DSA_LAB_UTILITY_H
#define DSA_LAB_UTILITY_H

#include <string.h>
#include <stdarg.h>

int write_log(const char *format, ...) {
    if(DEBUG) {
        printf("\n[DEBUG_LOG]> ");
        va_list args;
        va_start (args, format);
        vprintf(format, args);
        va_end (args);
    }
}

int *get_min_max(int *array, int no_of_elements, int min_max[]){
    // get minimum and maximum of array
//    printf("elements of array: ");
    for(int i=0; i<no_of_elements; i++){
//        printf("%d ", *(array + i));
        if (*(array + i) < min_max[0])
            min_max[0] = *(array + i);
        if (*(array + i) > min_max[1])
```

```c
            min_max[1] = *(array + i);
    }
    return min_max;
}

int display_array(int *array, int no_of_elements){
    // display given array of given size(no. of elements require because sizeof() returns max bound value)
    write_log(": ");
    for(int i=0; i<no_of_elements; i++){
        write_log( "%d ", *(array + i));
    }
    return 0;
}

int show_2d_array(int **array, int no_of_elements){
    // display given array of given size(no. of elements require because sizeof() returns max bound value)
    write_log(": ");
    for(int i=0; i<no_of_elements; i++){
        printf("a[%d][i]: ", i);
        for(int j=0; j<no_of_elements; j++) {
//            printf("array[%d][%d]: %d ", i, j, array[i][j]);
            printf("%d\t", array[i][j]);
        }
        printf("\twhere 0<=i<=%d\n", no_of_elements-1);
    }
    return 0;
}

int display_2d_array(int **array, int no_of_elements){
    // display given array of given size(no. of elements require because sizeof() returns max bound value)
    write_log(": ");
    for(int i=0; i<no_of_elements; i++){
        printf("a[%d][]: ", i);
        for(int j=0; j<no_of_elements; j++) {
//            printf("array[%d][%d]: %d ", i, j, array[i][j]);
            printf("%d ", array[i][j]);
        }
        printf("\n");
    }
    return 0;
}


void swap(int *one, int *two){
    // swap function to swap elements by location/address
    int temp = *one;
    *one = *two;
    *two = temp;
}

#endif //DSA_LAB_UTILITY_H
```

II. Main Program - ***recursive_exponential.c***

```c
// ------------------------------------------------------------------------------------------
// Author: Gahan Saraiya
// GiT: http://github.com/gahan9/
// StackOverflow: https://stackoverflow.com/users/story/7664524
// Website: http://gahan9.github.io/
// ------------------------------------------------------------------------------------------
// Implementing Strassen's's Matrix Multiplication Algorithm with Divide and Conquer

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "../utils/constant.h"
#include "../utils/utility.h"
#define MAX_SIZE 32

void add(int **a, int **b, int size, int **c) {
    write_log("Adding matrix\n");
    int i, j;
    for (i = 0; i < size; i++) {
        for (j = 0; j < size; j++) {
            c[i][j] = a[i][j] + b[i][j];
        }
    }
    write_log("Matrix Addition completed\n");
}

void sub(int **a, int **b, int size, int **c) {
    write_log("subtracting matrix\n");
    int i, j;
    for (i = 0; i < size; i++) {
        for (j = 0; j < size; j++) {
            c[i][j] = a[i][j] - b[i][j];
        }
    }
    write_log("Matrix Subtraction completed\n");
```

```
}

void multiply(int **c, int **d, int size, int size2, int **new){
    write_log("Multiplying Matrix...");
    if (size == 1) {
        new[0][0] = c[0][0] *d[0][0];
    }
    else {
        int i,j;
        int new_matrix_size = size/2;
        int **c11 = malloc(new_matrix_size * sizeof(int *));
        int **c12 = malloc(new_matrix_size * sizeof(int *));
        int **c21 = malloc(new_matrix_size * sizeof(int *));
        int **c22 = malloc(new_matrix_size * sizeof(int *));
        int **d11 = malloc(new_matrix_size * sizeof(int *));
        int **d12 = malloc(new_matrix_size * sizeof(int *));
        int **d21 = malloc(new_matrix_size * sizeof(int *));
        int **d22 = malloc(new_matrix_size * sizeof(int *));
        int **m1 = malloc(new_matrix_size * sizeof(int *));
        int **m2 = malloc(new_matrix_size * sizeof(int *));
        int **m3 = malloc(new_matrix_size * sizeof(int *));
        int **m4 = malloc(new_matrix_size * sizeof(int *));
        int **m5 = malloc(new_matrix_size * sizeof(int *));
        int **m6 = malloc(new_matrix_size * sizeof(int *));
        int **m7 = malloc(new_matrix_size * sizeof(int *));
        int **temp1 = malloc(new_matrix_size * sizeof(int *));
        int **temp2 = malloc(new_matrix_size * sizeof(int *));
        int **temp3 = malloc(new_matrix_size * sizeof(int *));
        int **temp4 = malloc(new_matrix_size * sizeof(int *));
        int **temp5 = malloc(new_matrix_size * sizeof(int *));
        int **temp6 = malloc(new_matrix_size * sizeof(int *));
        int **temp7 = malloc(new_matrix_size * sizeof(int *));
        int **temp8 = malloc(new_matrix_size * sizeof(int *));
        int **temp9 = malloc(new_matrix_size * sizeof(int *));
        int **temp10 = malloc(new_matrix_size * sizeof(int *));
        int **te1 = malloc(new_matrix_size * sizeof(int *));
        int **te2 = malloc(new_matrix_size * sizeof(int *));
        int **te3 = malloc(new_matrix_size * sizeof(int *));
        int **te4 = malloc(new_matrix_size * sizeof(int *));
        int **te5 = malloc(new_matrix_size * sizeof(int *));
        int **te6 = malloc(new_matrix_size * sizeof(int *));
        int **te7 = malloc(new_matrix_size * sizeof(int *));
        int **te8 = malloc(new_matrix_size * sizeof(int *));
        for(i=0; i < new_matrix_size; i++) {
            c11[i]= malloc(new_matrix_size * sizeof(int));
            c12[i]= malloc(new_matrix_size * sizeof(int));
            c21[i]= malloc(new_matrix_size * sizeof(int));
            c22[i]= malloc(new_matrix_size * sizeof(int));
            d11[i]= malloc(new_matrix_size * sizeof(int));
            d12[i]= malloc(new_matrix_size * sizeof(int));
            d21[i]= malloc(new_matrix_size * sizeof(int));
            d22[i]= malloc(new_matrix_size * sizeof(int));
            m1[i]= malloc(new_matrix_size * sizeof(int));
            m2[i]= malloc(new_matrix_size * sizeof(int));
            m3[i]= malloc(new_matrix_size * sizeof(int));
            m4[i]= malloc(new_matrix_size * sizeof(int));
            m5[i]= malloc(new_matrix_size * sizeof(int));
            m6[i]= malloc(new_matrix_size * sizeof(int));
            m7[i]= malloc(new_matrix_size * sizeof(int));
            temp1[i]= malloc(new_matrix_size * sizeof(int));
            temp2[i]= malloc(new_matrix_size * sizeof(int));
            temp3[i]= malloc(new_matrix_size * sizeof(int));
            temp4[i]= malloc(new_matrix_size * sizeof(int));
            temp5[i]= malloc(new_matrix_size * sizeof(int));
            temp6[i]= malloc(new_matrix_size * sizeof(int));
            temp7[i]= malloc(new_matrix_size * sizeof(int));
            temp8[i]= malloc(new_matrix_size * sizeof(int));
            temp9[i]= malloc(new_matrix_size * sizeof(int));
            temp10[i]= malloc(new_matrix_size * sizeof(int));
            te1[i]= malloc(new_matrix_size * sizeof(int));
            te2[i]= malloc(new_matrix_size * sizeof(int));
            te3[i]= malloc(new_matrix_size * sizeof(int));
            te4[i]= malloc(new_matrix_size * sizeof(int));
            te5[i]= malloc(new_matrix_size * sizeof(int));
            te6[i]= malloc(new_matrix_size * sizeof(int));
            te7[i]= malloc(new_matrix_size * sizeof(int));
            te8[i]= malloc(new_matrix_size * sizeof(int));
        }
        for(i=0; i < new_matrix_size; i++){
            for(j=0; j < new_matrix_size; j++){
                c11[i][j] = c[i][j];
                c12[i][j] = c[i][j+new_matrix_size];
                c21[i][j] = c[i+new_matrix_size][j];
                c22[i][j] = c[i+new_matrix_size][j+new_matrix_size];
                d11[i][j] = d[i][j];
                d12[i][j] = d[i][j+new_matrix_size];
                d21[i][j] = d[i+new_matrix_size][j];
                d22[i][j] = d[i+new_matrix_size][j+new_matrix_size];
            }
        }

        add(c11, c22, new_matrix_size, temp1);
        add(d11, d22, new_matrix_size, temp2);
        multiply(temp1, temp2, new_matrix_size, size, m1);

        add(c21, c22, new_matrix_size, temp3);
        multiply(temp3, d11, new_matrix_size, size, m2);


        sub(d12, d22, new_matrix_size, temp4);
        multiply(c11, temp4, new_matrix_size, size, m3);

        sub(d21, d11, new_matrix_size, temp5);
        multiply(c22, temp5, new_matrix_size, size, m4);

        add(c11, c12, new_matrix_size, temp6);
        multiply(temp6, d22, new_matrix_size, size, m5);

        sub(c21, c11, new_matrix_size, temp7);
        add(d11, d12, new_matrix_size, temp8);
        multiply(temp7, temp8, new_matrix_size, size, m6);

        sub(c12, c22, new_matrix_size, temp9);
        add(d21, d22, new_matrix_size, temp10);
        multiply(temp9, temp10, new_matrix_size, size, m7);

        add(m1, m7, new_matrix_size, te1);
        sub(m4, m5, new_matrix_size, te2);
        add(te1, te2, new_matrix_size, te3);    //c11

        add(m3, m5, new_matrix_size, te4);     //c12
        add(m2, m4, new_matrix_size, te5);     //c21

        add(m3, m6, new_matrix_size, te6);
        sub(m1, m2, new_matrix_size, te7);

        add(te6, te7, new_matrix_size, te8);   //c22

        int a=0;
        int b=0;
        int c=0;
        int d=0;
        int e=0;
        int nsize2 = 2*new_matrix_size;
        for(i=0; i < nsize2; i++){
            for(j=0; j < nsize2; j++){
                if(j>=0 && j<new_matrix_size && i>=0 && i<new_matrix_size){
                    new[i][j] = te3[i][j];
                }
                if(j>=new_matrix_size && j<nsize2 && i>=0 && i<new_matrix_size){
                    a=j-new_matrix_size;
                    new[i][j] = te4[i][a];
                }
                if(j>=0 && j<new_matrix_size && i>= new_matrix_size && i <
                 ↪  nsize2){
                    c=i-new_matrix_size;
                    new[i][j] = te5[c][j];
                }
                if(j>=new_matrix_size && j< nsize2 && i>= new_matrix_size && i<
                 ↪  nsize2 ){
                    d = i-new_matrix_size;
                    e = j-new_matrix_size;
                    new[i][j] = te8[d][e];
                }
            }
        }
    }
}

void main(){
    int size, p, itr, itr1, i, j;
    printf("Enter Size of square matrix: \n");
    scanf("%d", &size);
    printf("Size of square matrix is : %d (%d x %d)\n", size, size, size);
    int tempS = size;
    if(size & size-1 != 0){
        p = log(size)/log(2);
        size = pow(2, p+1);
    }
    int **a = malloc(size * sizeof(int *));
    for (i = 0; i < size; i++) {
        a[i] = malloc(size * sizeof(int));
    }
    int **b = malloc(size * sizeof(int *));
    for (i = 0; i < size; i++) {
        b[i] = malloc(size * sizeof(int));
    }
    printf("Enter elements of 1st matrix\n");
    for (itr = 0; itr < size; itr++) {
        for (itr1 = 0; itr1 < size; itr1++) {
            if (itr >= tempS || itr1 >= tempS)
                a[itr][itr1] = 0;
            else {
                printf("\na[%d][%d]: ", itr, itr1);
                scanf("%d", &a[itr][itr1]);
            }
        }
    }
    printf("Enter elements of 2nd matrix\n");
    for (itr = 0; itr < size; itr++) {
        for (itr1 = 0; itr1 < size; itr1++) {
            if (itr >= tempS || itr1 >= tempS)
                a[itr][itr1] = 0;
            else {
                printf("\na[%d][%d]: ", itr, itr1);
                scanf("%d", &b[itr][itr1]);
            }
        }
    }
    int **new = malloc(size * sizeof(int *));
    for (i = 0; i < size; i++) {
        new[i] = malloc(size * sizeof(int));
    }
    printf("Multiplying matrix 1--------\n");
    show_2d_array(a, size);
    printf("with matrix 2--------\n");
    show_2d_array(b, size);
    multiply(a, b, size, size, new);

    if (tempS < size)
        size = tempS;
    printf("Answer:---\n");
    show_2d_array(new, size);
}
```

II.1 Output

```
Enter Size of square matrix: 4
Size of square matrix is : 4 (4 x 4)

Enter elements of 1st matrix

a[0][0]: 1
a[0][1]: 1
a[0][2]: 1
a[0][3]: 1
a[1][0]: 1
a[1][1]: 1
a[1][2]: 1
a[1][3]: 1
a[2][0]: 1
a[2][1]: 1
a[2][2]: 1
a[2][3]: 1
a[3][0]: 1
```

```
a[3][1]: 1
a[3][2]: 1
a[3][3]: 1

Enter elements of 2nd matrix

a[0][0]: 1
a[0][1]: 1
a[0][2]: 1
a[0][3]: 1
a[1][0]: 1
a[1][1]: 1
a[1][2]: 1
a[1][3]: 1
a[2][0]: 1
a[2][1]: 1
a[2][2]: 1
a[2][3]: 1
a[3][0]: 1
a[3][1]: 1
a[3][2]: 1
a[3][3]: Multiplying matrix 1--------
a[0][i]: 1        1        1        1                where 0<=i<=3
a[1][i]: 1        1        1        1                where 0<=i<=3
a[2][i]: 1        1        1        1                where 0<=i<=3
a[3][i]: 1        1        1        1                where 0<=i<=3
with matrix 2--------
a[0][i]: 1        1        1        1                where 0<=i<=3
a[1][i]: 1        1        1        1                where 0<=i<=3
a[2][i]: 1        1        1        1                where 0<=i<=3
a[3][i]: 1        1        1        1                where 0<=i<=3
Answer:---
a[0][i]: 4        4        4        4                where 0<=i<=3
a[1][i]: 4        4        4        4                where 0<=i<=3
a[2][i]: 4        4        4        4                where 0<=i<=3
a[3][i]: 4        4        4        4                where 0<=i<=3
```