Practical 5 A
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***

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

#ifndef DSA_LAB_UTILITY_H
#define DSA_LAB_UTILITY_H

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

int max(int a, int b) { return (a > b)? a : b; }
int min(int a, int b) { return (a < b)? a : b; }

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));
```

```
29          if (*(array + i) < min_max[0])
30              min_max[0] = *(array + i);
31          if (*(array + i) > min_max[1])
32              min_max[1] = *(array + i);
33      }
34      return min_max;
35  }
36
37  int display_array(int *array, int no_of_elements){
38      // display given array of given size(no. of elements require because sizeof()
        ↪  returns max bound value)
39      write_log(": ");
40      for(int i=0; i<no_of_elements; i++){
41          write_log( "%d ", *(array + i));
42      }
43      return 0;
44  }
45
46  int show_2d_array(int **array, int no_of_elements){
47      // display given array of given size(no. of elements require because sizeof()
        ↪  returns max bound value)
48      write_log(": ");
49      for(int i=0; i<no_of_elements; i++){
50          printf("a[%d][i]: ", i);
51          for(int j=0; j<no_of_elements; j++) {
52  //            printf("array[%d][%d]: %d ", i, j, array[i][j]);
53              printf("%d\t", array[i][j]);
54          }
55          printf("\twhere 0<=i<=%d\n", no_of_elements-1);
56      }
57      return 0;
58  }
59
60  int display_2d_array(int **array, int no_of_elements){
61      // display given array of given size(no. of elements require because sizeof()
        ↪  returns max bound value)
62      write_log(": ");
63      for(int i=0; i<no_of_elements; i++){
64          printf("a[%d][]: ", i);
65          for(int j=0; j<no_of_elements; j++) {
66  //            printf("array[%d][%d]: %d ", i, j, array[i][j]);
67              printf("%d ", array[i][j]);
68          }
69          printf("\n");
70      }
71      return 0;
72  }
73
```

```

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 - ***strassen_s_matrix_multiplication.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];
```

```
33          }
34      }
35      write_log("Matrix Subtraction completed\n");
36  }
37
38  void multiply(int **c, int **d, int size, int size2, int **new){
39      write_log("Multiplying Matrix...");
40      if (size == 1) {
41          new[0][0] = c[0][0] *d[0][0];
42      }
43      else {
44          int i,j;
45          int new_matrix_size = size/2;
46          int **c11 = malloc(new_matrix_size * sizeof(int *));
47          int **c12 = malloc(new_matrix_size * sizeof(int *));
48          int **c21 = malloc(new_matrix_size * sizeof(int *));
49          int **c22 = malloc(new_matrix_size * sizeof(int *));
50          int **d11 = malloc(new_matrix_size * sizeof(int *));
51          int **d12 = malloc(new_matrix_size * sizeof(int *));
52          int **d21 = malloc(new_matrix_size * sizeof(int *));
53          int **d22 = malloc(new_matrix_size * sizeof(int *));
54          int **m1 = malloc(new_matrix_size * sizeof(int *));
55          int **m2 = malloc(new_matrix_size * sizeof(int *));
56          int **m3 = malloc(new_matrix_size * sizeof(int *));
57          int **m4 = malloc(new_matrix_size * sizeof(int *));
58          int **m5 = malloc(new_matrix_size * sizeof(int *));
59          int **m6 = malloc(new_matrix_size * sizeof(int *));
60          int **m7 = malloc(new_matrix_size * sizeof(int *));
61          int **temp1 = malloc(new_matrix_size * sizeof(int *));
62          int **temp2 = malloc(new_matrix_size * sizeof(int *));
63          int **temp3 = malloc(new_matrix_size * sizeof(int *));
64          int **temp4 = malloc(new_matrix_size * sizeof(int *));
65          int **temp5 = malloc(new_matrix_size * sizeof(int *));
66          int **temp6 = malloc(new_matrix_size * sizeof(int *));
67          int **temp7 = malloc(new_matrix_size * sizeof(int *));
68          int **temp8 = malloc(new_matrix_size * sizeof(int *));
69          int **temp9 = malloc(new_matrix_size * sizeof(int *));
70          int **temp10 = malloc(new_matrix_size * sizeof(int *));
71          int **te1 = malloc(new_matrix_size * sizeof(int *));
72          int **te2 = malloc(new_matrix_size * sizeof(int *));
73          int **te3 = malloc(new_matrix_size * sizeof(int *));
74          int **te4 = malloc(new_matrix_size * sizeof(int *));
75          int **te5 = malloc(new_matrix_size * sizeof(int *));
76          int **te6 = malloc(new_matrix_size * sizeof(int *));
77          int **te7 = malloc(new_matrix_size * sizeof(int *));
78          int **te8 = malloc(new_matrix_size * sizeof(int *));
79          for(i=0; i < new_matrix_size; i++) {
80              c11[i]= malloc(new_matrix_size * sizeof(int));
```

```
81              c12[i]= malloc(new_matrix_size * sizeof(int));
82              c21[i]= malloc(new_matrix_size * sizeof(int));
83              c22[i]= malloc(new_matrix_size * sizeof(int));
84              d11[i]= malloc(new_matrix_size * sizeof(int));
85              d12[i]= malloc(new_matrix_size * sizeof(int));
86              d21[i]= malloc(new_matrix_size * sizeof(int));
87              d22[i]= malloc(new_matrix_size * sizeof(int));
88              m1[i]= malloc(new_matrix_size * sizeof(int));
89              m2[i]= malloc(new_matrix_size * sizeof(int));
90              m3[i]= malloc(new_matrix_size * sizeof(int));
91              m4[i]= malloc(new_matrix_size * sizeof(int));
92              m5[i]= malloc(new_matrix_size * sizeof(int));
93              m6[i]= malloc(new_matrix_size * sizeof(int));
94              m7[i]= malloc(new_matrix_size * sizeof(int));
95              temp1[i]= malloc(new_matrix_size * sizeof(int));
96              temp2[i]= malloc(new_matrix_size * sizeof(int));
97              temp3[i]= malloc(new_matrix_size * sizeof(int));
98              temp4[i]= malloc(new_matrix_size * sizeof(int));
99              temp5[i]= malloc(new_matrix_size * sizeof(int));
100             temp6[i]= malloc(new_matrix_size * sizeof(int));
101             temp7[i]= malloc(new_matrix_size * sizeof(int));
102             temp8[i]= malloc(new_matrix_size * sizeof(int));
103             temp9[i]= malloc(new_matrix_size * sizeof(int));
104             temp10[i]= malloc(new_matrix_size * sizeof(int));
105             te1[i]= malloc(new_matrix_size * sizeof(int));
106             te2[i]= malloc(new_matrix_size * sizeof(int));
107             te3[i]= malloc(new_matrix_size * sizeof(int));
108             te4[i]= malloc(new_matrix_size * sizeof(int));
109             te5[i]= malloc(new_matrix_size * sizeof(int));
110             te6[i]= malloc(new_matrix_size * sizeof(int));
111             te7[i]= malloc(new_matrix_size * sizeof(int));
112             te8[i]= malloc(new_matrix_size * sizeof(int));
113         }
114         for(i=0; i < new_matrix_size; i++){
115             for(j=0; j < new_matrix_size; j++){
116                 c11[i][j] = c[i][j];
117                 c12[i][j] = c[i][j+new_matrix_size];
118                 c21[i][j] = c[i+new_matrix_size][j];
119                 c22[i][j] = c[i+new_matrix_size][j+new_matrix_size];
120                 d11[i][j] = d[i][j];
121                 d12[i][j] = d[i][j+new_matrix_size];
122                 d21[i][j] = d[i+new_matrix_size][j];
123                 d22[i][j] = d[i+new_matrix_size][j+new_matrix_size];
124             }
125         }
126
127         add(c11, c22, new_matrix_size, temp1);
128         add(d11, d22, new_matrix_size, temp2);
```

```
129         multiply(temp1, temp2, new_matrix_size, size, m1);
130
131         add(c21, c22, new_matrix_size, temp3);
132         multiply(temp3, d11, new_matrix_size, size, m2);
133
134
135         sub(d12, d22, new_matrix_size, temp4);
136         multiply(c11, temp4, new_matrix_size, size, m3);
137
138         sub(d21, d11, new_matrix_size, temp5);
139         multiply(c22, temp5, new_matrix_size, size, m4);
140
141         add(c11, c12, new_matrix_size, temp6);
142         multiply(temp6, d22, new_matrix_size, size, m5);
143
144         sub(c21, c11, new_matrix_size, temp7);
145         add(d11, d12, new_matrix_size, temp8);
146         multiply(temp7, temp8, new_matrix_size, size, m6);
147
148         sub(c12, c22, new_matrix_size, temp9);
149         add(d21, d22, new_matrix_size, temp10);
150         multiply(temp9, temp10, new_matrix_size, size, m7);
151
152         add(m1, m7, new_matrix_size, te1);
153         sub(m4, m5, new_matrix_size, te2);
154         add(te1, te2, new_matrix_size, te3);    //c11
155
156         add(m3, m5, new_matrix_size, te4);     //c12
157         add(m2, m4, new_matrix_size, te5);     //c21
158
159         add(m3, m6, new_matrix_size, te6);
160         sub(m1, m2, new_matrix_size, te7);
161
162         add(te6, te7, new_matrix_size, te8);   //c22
163
164         int a=0;
165         int b=0;
166         int c=0;
167         int d=0;
168         int e=0;
169         int nsize2 = 2*new_matrix_size;
170         for(i=0; i < nsize2; i++){
171             for(j=0; j < nsize2; j++){
172                 if(j>=0 && j<new_matrix_size && i>=0 && i<new_matrix_size){
173                     new[i][j] = te3[i][j];
174                 }
175                 if(j>=new_matrix_size && j<nsize2 && i>=0 && i<new_matrix_size){
176                     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("\nEnter 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("\nEnter elements of 2nd matrix\n");
```

```
223     for (itr = 0; itr < size; itr++) {
224         for (itr1 = 0; itr1 < size; itr1++) {
225             if (itr >= tempS || itr1 >= tempS)
226                 a[itr][itr1] = 0;
227             else {
228                 printf("\na[%d][%d]: ", itr, itr1);
229                 scanf("%d", &b[itr][itr1]);
230             }
231         }
232     }
233     int **new = malloc(size * sizeof(int *));
234     for (i = 0; i < size; i++) {
235         new[i] = malloc(size * sizeof(int));
236     }
237     printf("\nMultiplying matrix 1--------\n");
238     show_2d_array(a, size);
239     printf("\nwith matrix 2--------\n");
240     show_2d_array(b, size);
241     multiply(a, b, size, size, new);
242 
243     if (tempS < size)
244         size = tempS;
245     printf("\nAnswer:---\n");
246     show_2d_array(new, size);
247 }
```

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]: 1
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
```