

# **Practical 5**

## **Implementation of Strassen's Fast Multiplication of Matrices Algorithm**

GAHAN M. SARAIYA, 18MCEC10

[18mcec10@nirmauni.ac.in](mailto: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*

---

```
1 //
2 // Created by jarvis on 17/8/18.
3 //
4
5 #ifndef DSA_LAB_UTILITY_H
6 #define DSA_LAB_UTILITY_H
7
8 #include <string.h>
9 #include <stdarg.h>
10
11 int write_log(const char *format, ...) {
12     if(DEBUG) {
13         printf("\n[DEBUG_LOG]> ");
14         va_list args;
15         va_start (args, format);
16         vprintf(format, args);
17         va_end (args);
18     }
19 }
20
21 int *get_min_max(int *array, int no_of_elements, int min_max[]){
22     // get minimum and maximum of array
23     // printf("elements of array: ");
24     for(int i=0; i<no_of_elements; i++){
25         // printf("%d ", *(array + i));
26         if (*(array + i) < min_max[0])
27             min_max[0] = *(array + i);
28         if (*(array + i) > min_max[1])
```

```
29         min_max[1] = *(array + i);
30     }
31     return min_max;
32 }
33
34 int display_array(int *array, int no_of_elements){
35     // display given array of given size(no. of elements require because sizeof()
36     // returns max bound value)
37     write_log(": ");
38     for(int i=0; i<no_of_elements; i++){
39         write_log(" %d ", *(array + i));
40     }
41     return 0;
42 }
43
44 int show_2d_array(int **array, int no_of_elements){
45     // display given array of given size(no. of elements require because sizeof()
46     // returns max bound value)
47     write_log(": ");
48     for(int i=0; i<no_of_elements; i++){
49         printf("a[%d][i]: ", i);
50         for(int j=0; j<no_of_elements; j++) {
51             printf("array[%d][%d]: %d ", i, j, array[i][j]);
52             printf("%d\t", array[i][j]);
53         }
54         printf("\twhere 0<=i<=%d\n", no_of_elements-1);
55     }
56     return 0;
57 }
58
59 int display_2d_array(int **array, int no_of_elements){
60     // display given array of given size(no. of elements require because sizeof()
61     // returns max bound value)
62     write_log(": ");
63     for(int i=0; i<no_of_elements; i++){
64         printf("a[%d][i]: ", 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
74 void swap(int *one, int *two){
75     // swap function to swap elements by location/address
```

```

74     int temp = *one;
75     *one = *two;
76     *two = temp;
77 }
78
79 #endif //DSA_LAB_UTILITY_H

```

---

## II. Main Program - *recursive\_exponential.c*

---

```

1  //
   ↳ -----
2  // Author: Gahan Saraiya
3  // GiT: http://github.com/gahan9/
4  // StackOverflow: https://stackoverflow.com/users/story/7664524
5  // Website: http://gahan9.github.io/
6  //
   ↳ -----
7  // Implementing Strassen's's Matrix Multiplication Algorithm with Divide and
   ↳ Conquer
8
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <math.h>
12 #include "../utils/constant.h"
13 #include "../utils/utility.h"
14 #define MAX_SIZE 32
15
16 void add(int **a, int **b, int size, int **c) {
17     write_log("Adding matrix\n");
18     int i, j;
19     for (i = 0; i < size; i++) {
20         for (j = 0; j < size; j++) {
21             c[i][j] = a[i][j] + b[i][j];
22         }
23     }
24     write_log("Matrix Addition completed\n");
25 }
26
27 void sub(int **a, int **b, int size, int **c) {
28     write_log("subtracting matrix\n");
29     int i, j;
30     for (i = 0; i < size; i++) {
31         for (j = 0; j < size; j++) {
32             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;
177                 new[i][j] = te4[i][a];
178             }
179         }
180     }
```

```

179         if(j>=0 && j<new_matrix_size && i>= new_matrix_size && i <
            ↳ nsize2){
180             c=i-new_matrix_size;
181             new[i][j] = te5[c][j];
182         }
183         if(j>=new_matrix_size && j< nsize2 && i>= new_matrix_size && i<
            ↳ nsize2 ){
184             d = i-new_matrix_size;
185             e = j-new_matrix_size;
186             new[i][j] = te8[d][e];
187         }
188     }
189 }
190 }
191 }
192
193 void main(){
194     int size, p, itr, itr1, i, j;
195     printf("Enter Size of square matrix: \n");
196     scanf("%d", &size);
197     printf("Size of square matrix is : %d (%d x %d)\n", size, size, size);
198     int tempS = size;
199     if(size & size-1 != 0){
200         p = log(size)/log(2);
201         size = pow(2, p+1);
202     }
203     int **a = malloc(size * sizeof(int *));
204     for (i = 0; i < size; i++) {
205         a[i] = malloc(size * sizeof(int));
206     }
207     int **b = malloc(size * sizeof(int *));
208     for (i = 0; i < size; i++) {
209         b[i] = malloc(size * sizeof(int));
210     }
211     printf("Enter elements of 1st matrix\n");
212     for (itr = 0; itr < size; itr++) {
213         for (itr1 = 0; itr1 < size; itr1++) {
214             if (itr >= tempS || itr1 >= tempS)
215                 a[itr][itr1] = 0;
216             else {
217                 printf("\na[%d][%d]: ", itr, itr1);
218                 scanf("%d", &a[itr][itr1]);
219             }
220         }
221     }
222     printf("Enter 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("Multiplying matrix 1-----\n");
238 show_2d_array(a, size);
239 printf("with 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("Answer:---\n");
246 show_2d_array(new, size);
247 }
```

---

## II.1 Output

```
1 Enter Size of square matrix: 4
2 Size of square matrix is : 4 (4 x 4)
3
4 Enter elements of 1st matrix
5
6 a[0][0]: 1
7 a[0][1]: 1
8 a[0][2]: 1
9 a[0][3]: 1
10 a[1][0]: 1
11 a[1][1]: 1
12 a[1][2]: 1
13 a[1][3]: 1
14 a[2][0]: 1
15 a[2][1]: 1
16 a[2][2]: 1
17 a[2][3]: 1
18 a[3][0]: 1
```



```

19 a[3][1]: 1
20 a[3][2]: 1
21 a[3][3]: 1
22
23 Enter elements of 2nd matrix
24
25 a[0][0]: 1
26 a[0][1]: 1
27 a[0][2]: 1
28 a[0][3]: 1
29 a[1][0]: 1
30 a[1][1]: 1
31 a[1][2]: 1
32 a[1][3]: 1
33 a[2][0]: 1
34 a[2][1]: 1
35 a[2][2]: 1
36 a[2][3]: 1
37 a[3][0]: 1
38 a[3][1]: 1
39 a[3][2]: 1
40 a[3][3]: Multiplying matrix 1-----
41 a[0][i]: 1      1      1      1      where 0<=i<=3
42 a[1][i]: 1      1      1      1      where 0<=i<=3
43 a[2][i]: 1      1      1      1      where 0<=i<=3
44 a[3][i]: 1      1      1      1      where 0<=i<=3
45 with matrix 2-----
46 a[0][i]: 1      1      1      1      where 0<=i<=3
47 a[1][i]: 1      1      1      1      where 0<=i<=3
48 a[2][i]: 1      1      1      1      where 0<=i<=3
49 a[3][i]: 1      1      1      1      where 0<=i<=3
50 Answer:---
51 a[0][i]: 4      4      4      4      where 0<=i<=3
52 a[1][i]: 4      4      4      4      where 0<=i<=3
53 a[2][i]: 4      4      4      4      where 0<=i<=3
54 a[3][i]: 4      4      4      4      where 0<=i<=3

```