UNIVERSITY OF WITWATERSRAND

ELEN4020

DATA INTENSIVE COMPUTING IN DATA SCIENCE

Laboratory Exercise 1

Authors:
KISHAN NAROTAM - 717 931
JESAL CHANA - 603 177

Authors:
SYED HUSSAIN - 600 524
YUSUF ALLY - 604 973

23rd February 2018

I. Problem Description

A K-dimensional integer array, represented as $A[N_0][N_1]...[N_{K-1}]$, must be created for different values of K. The integer array is bounded by $N_0, N_1, ..., N_{K-1}$, and once created, the array must undergo three procedures which are as follows:

1) Initialize all elements in the array to be zeroes
2) Set 10% of all elements in the array uniformly to 1's
3) In a random fashion across all dimensions in the array:
    a) Choose 5% of the elements
    b) Print the coordinate indices of the elements
    c) Print the value of the element at the chosen cell.

Four integer arrays must be created, each with various dimension size as well as array size. For each of these arrays, the three procedures must be called, however it must be completed as if the number of dimensions in the array are not static.

II. Background

An array is defined as a data structure that contains a group of elements, in the case of this problem the elements are integers [1]. An array can have multiple dimensions, for example an array with one dimension is regarded as a row, whereas an array with two dimensions is regarded as a matrix with a row and a column. However as the dimensions of the matrix reach three or more, the array has a new coordinate property known as depth/page [2], [3]. Figure 1 in Appendix A is a representation of a 3-dimensional array with the coordinate properties. As the dimensions increase it becomes more difficult to represent these arrays in a graphical representation.

A. Memory Allocation

The memory and layout of a K-dimensional array could be allocated in multiple ways. Two of the most common ways of this memory layout are row-major order or column-major order.

In a 2D array, the row-major order layout would put the first row in contiguous memory, the second row would follow, and so on. Similarly, for column-major order layout, the first column would be in contiguous memory, the second column would follow and so on [2].

In a 3D array, using the dimensions of row, column and depth/page, Figure 2 in Appendix A shows how a 3-dimensional array memory is allocated and how the layout is presented in row-major order [2].

In order to calculate the memory location of an element from its respective index in the array, the following formulae can be used for row-major order and column-major order respectively for a K-dimensional array.

$$\begin{aligned} offset &= n_K + N_K \cdot (n_{K-1} + N_{K-1} \cdot (n_{K-2} + N_{K-2} \cdot \\ &\quad (... + N_2 n_1)...)) \\ &= \sum_{i=1}^{K} \left(\prod_{j=i+1}^{K} N_j \right) n_i \end{aligned} \tag{1}$$

$$\begin{aligned} offset &= n_1 + N_1 \cdot (n_2 + N_2 \cdot (n_3 + N_3 \cdot (... + N_{K-1} n_K)...)) \\ &= \sum_{i=1}^{K} \left(\prod_{j=1}^{i-1} N_j \right) n_i \end{aligned} \tag{2}$$

III. Function Descriptions

A. `main`

This is the function that is responsible for calling the subsequent functions. The user is given two options, whether to enter an array of any dimension and any size and to perform the three procedures or to perform the three procedures of the predefined arrays presented in Section I.

B. `allocateArray`

This function has two main responsibilities : to dynamically allocate the desired array, and to then call the three procedures that execute on the array. This function takes as inputs the initializer array (`iniArray[]`), and the number of dimensions.

1) Dynamic memory allocation: The array is generated using `malloc`, which allocates a contiguous memory block. The size of the block is determined by the number of elements in the array and the number of bytes per element.

2) Calling procedures: Once the array has been generated, the three procedures are called in succession.

C. `procedureOne`

This procedure sets every element in the array to zero. It takes as inputs the array, and the number of elements in the array (in order to know how many elements it should step through).

D. `procedureTwo`

This procedure sets the first 10% of elements of the array to one. Like `procedureOne` it takes as inputs the array and the number of elements in the array. Variable `tenPercentElements` is the value of 10% of elements of the array, and is used to determine whether an element should be set to one.

E. `procedureThree`

This procedure randomly chooses 5% of elements in the array and prints the corresponding value and coordinate indices of these elements.

1) Random coordinate generation: The initialized array is in the form `int iniArray[] =` $\{N_0, N_1, ..., N_{K-1}\}$, where K is the array dimension and $N_0, N_1, ...$ are the sizes of each dimension. For example, `int iniArray[] =` $\{3, 3\}$ creates a 2D $(3X3)$ array.

A random coordinate for all K indexes is generated using $rand()\%N_K$, where $\%N_K$ bounds each coordinate index. This bounding ensures a valid coordinate is created every time. Each coordinate is then stored in `coordinate_array`.

2) Coordinate mapping: The randomly generated coordinate is now mapped to its 1D linear array index. This is done using the row major order indexing equation (lines 150-156). The corresponding value is obtained using this index. The coordinate and value is then printed, for 5% of elements in the array.

IV. Pseudo Code

The pseudo code for the entire program is seen in Appendix B. Although written in one file, each function is split up into their own algorithms to better understand the pseudo code of each. Algorithm 1 in Appendix B is the pseudo code for the `main` function and Algorithm 2 in Appendix B is the pseudo code for the `allocateArray` void function. Algorithms 3, 4, 5 in Appendix B, are the pseudo codes for the first, second and third procedures respectively.

V. Final Code and Output

The final program was coded in C on an Ubuntu system and compiled using a custom makefile. The program was tested using *K*-dimensional arrays of different size to ensure it works as desired. These outputs can be seen in Figures 3, 4, 5, and 6 in Appendix C. The four given arrays mentioned in section I are tested in the program and yield the relevant results.

VI. Conclusion

A method was created to step through the elements of multiple arrays which varied in size and dimensions. The program yielded relevant results and was done so in a general way, i.e. the number of dimensions in the array are dynamic, as opposed to a nested loop. The coding was done in C on an Ubuntu operating system and was executed using a makefile in the command line prompt.

References

[1] Christensson, P; *Array Definition*; https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays; Last Accessed: 21/02/2018

[2] Bendersky, E; *Memory layout of multi-dimensional arrays - Eli Bendersky's website*; https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays; Last Accessed: 21/02/2018

[3] Mathworks; *Multidimensional Arrays - MATLAB & Simulink*; https://www.mathworks.com/help/matlab/math/multidimensional-arrays.html; Last Accessed: 21/02/2018

APPENDIX A
DIAGRAMS

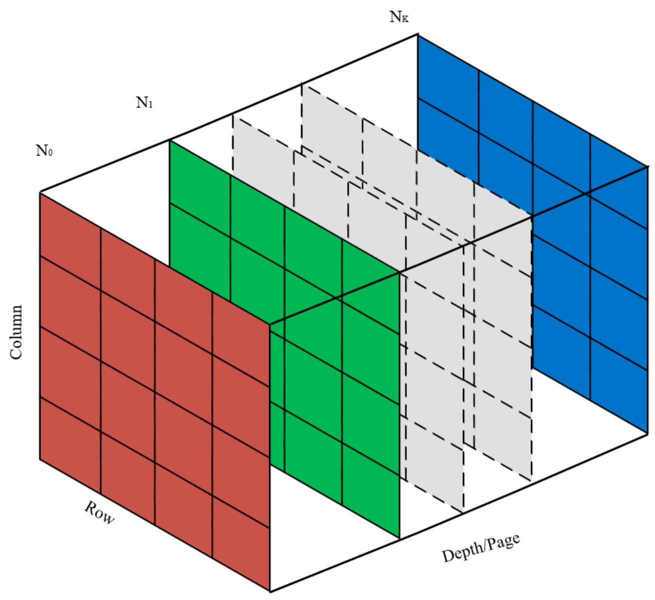


Fig. 1. Representation of a *3*-dimensional array

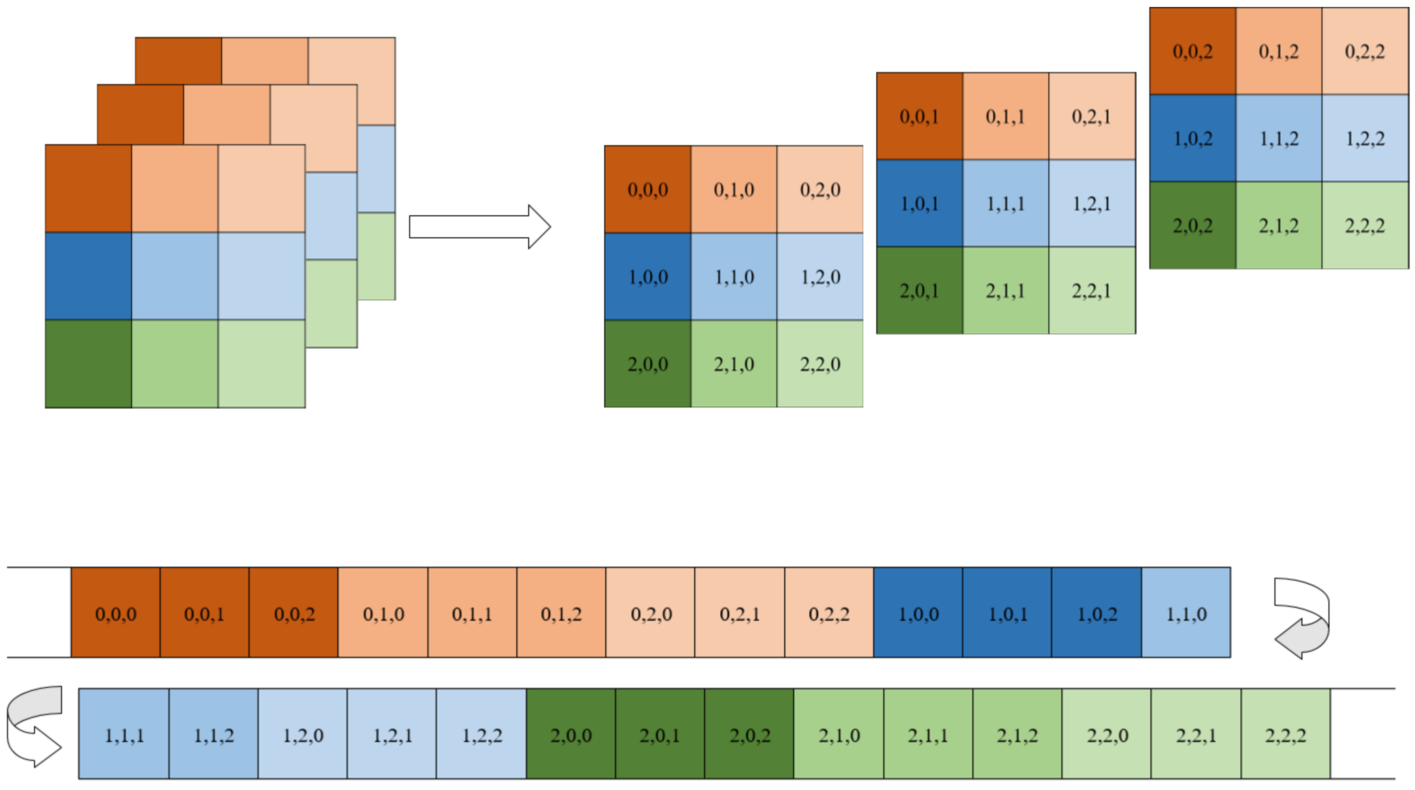


Fig. 2. Memory allocation of a 3-dimensional in row-major order [2]

APPENDIX B
PSEUDO CODE

Algorithm 1 `main` Function

```
$srand(time(NULL))$
$Dimension$
$mode$

Display “Press 1 to run or 2 to input an array”
$mode \leftarrow$ User input
**if** $mode = 2$ **then**
  Display “Enter array dimension: ”
  $Dimension \leftarrow$ User input
  $userArray[Dimension]$
  Display “Enter the size of each dimension: ”
  **for** $i = 0$ to $Dimensions$ **do**
    $userArray[i] \leftarrow$ User input
  **end for**
  $allocateArray(userArray, Dimension)$
**end if**

**if** $mode = 1$ **then**
  $arrayInitializer1[\,] \leftarrow (100, 100)$
  $Dimension \leftarrow sizeof(arrayInitializer1)/sizeof(int)$
  $allocateArray(arrayInitializer1, Dimension)$

  $arrayInitializer2[\,] \leftarrow (100, 100, 100)$
  $Dimension \leftarrow sizeof(arrayInitializer2)/sizeof(int)$
  $allocateArray(arrayInitializer2, Dimension)$

  $arrayInitializer3[\,] \leftarrow (50, 50, 50, 50)$
  $Dimension \leftarrow sizeof(arrayInitializer3)/sizeof(int)$
  $allocateArray(arrayInitializer3, Dimension)$

  $arrayInitializer4[\,] \leftarrow (20, 20, 20, 20, 20)$
  $Dimension \leftarrow sizeof(arrayInitializer4)/sizeof(int)$
  $allocateArray(arrayInitializer4, Dimension)$
**end if**
```

Algorithm 2 `allocateArray` Function

```
Input: $iniArray[\,]$ and $dimension$

$num\_elements \leftarrow 1$
for $i = 0$ to $dimension$ do
    $num\_elements \leftarrow num\_elements * iniArray[i]$
end for
refToInteger $array$
$array \leftarrow$ refToInteger $malloc(sizeof($refToInteger $num\_elements))$
Display “Dimensions $dimensions$, Elements $num\_elements$”

if $array \leftarrow NULL$ then
    Display “malloc failed”
    $exit(1)$
end if

for $i = 0$ to $num\_elements$ do
    $array[i] \leftarrow i$
end for
$procedureOne(array, num\_elements)$
$procedureTwo(array, num\_elements)$
$procedureThree(array, iniArray, num\_elements, dimension)$
```

Algorithm 3 `procedureOne` Function

```
Input: $arr[\,]$ and $numElements$

Display “Procedure 1: ”
for $i = 0$ to $numElements$ do
    $arr[i] \leftarrow 0$
end for
Display “...Executed procedure 1...”
```

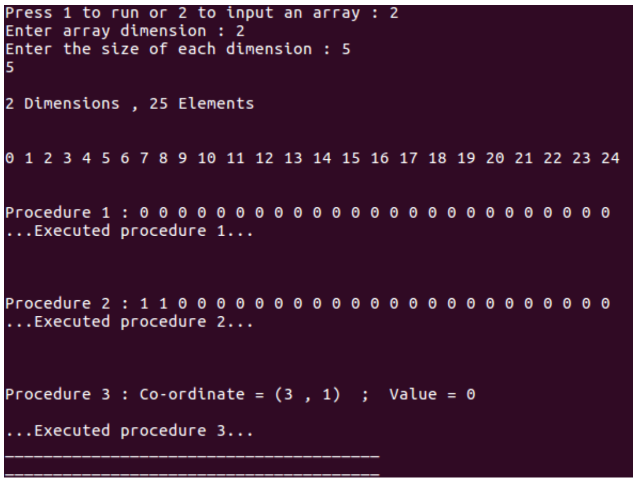
Algorithm 4 `procedureTwo` Function

```
Input: $arr[\,]$ and $numElements$

Display “Procedure 2: ”
$tenPercentElements \leftarrow numElements/10$
for $i = 0$ to $numElements$ do
    if $i < tenPercentElements$ then
        $arr[i] \leftarrow 1$
    end if
end for
Display “...Executed procedure 2...”
```

Algorithm 5 `procedureThree` Function

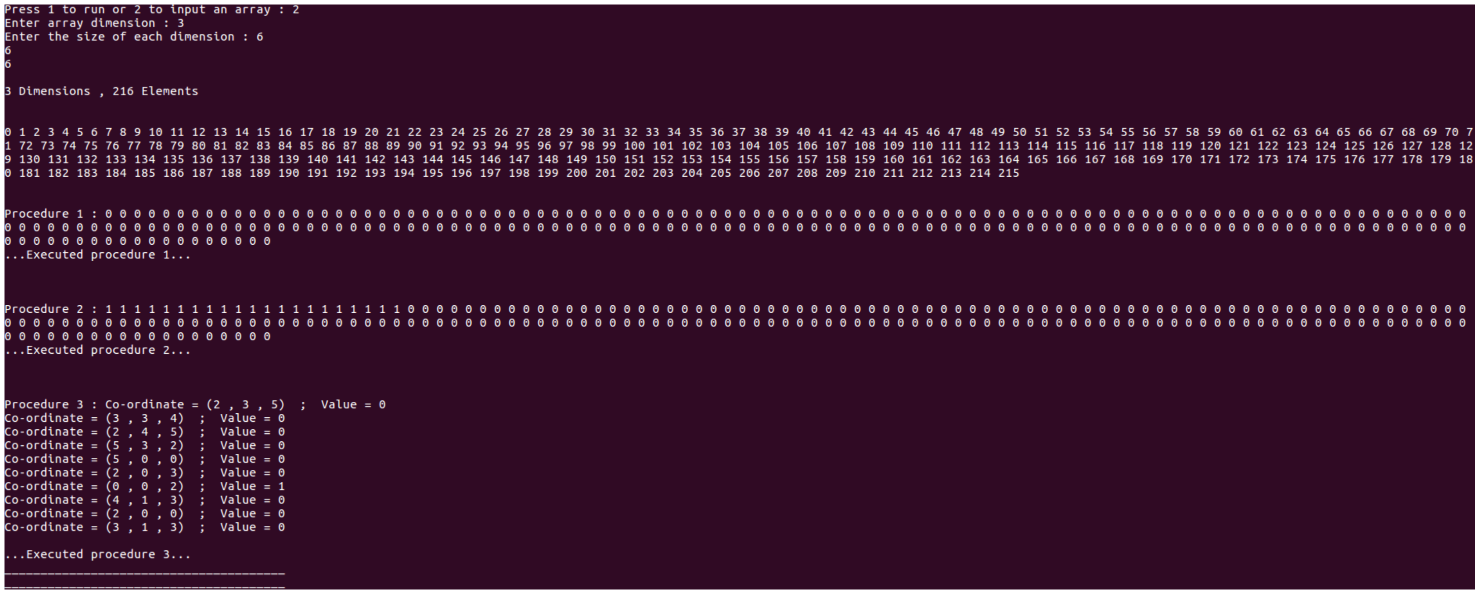
```
Input: $arr[\ ]$, $arr2[\ ]$, $numberElements$, $dimensions$

Display "Procedure 3: "
$fivePercentElements \leftarrow numberElements * 5/100$
for $a = 0$ to $fivePercenElements$ do
    refToInteger $coordinate\_array$
    $array \leftarrow$ refToInteger $malloc(sizeof($refToInteger $dimension))$
    Display "Coordinate = ("
    for $i = 0$ to $dimension$ do
        $coordinate\_array[i] \leftarrow rand()\%arr2[i]$
        Display $coordinat\_array[i]$
        if $i < dimension - 1$ then
            Display ","
        end if
    end for
    Display ")"
    $product \leftarrow 1$
    $index \leftarrow 0$
    $product2 \leftarrow 1$
    for $i = 0$ to $dimension - 1$ do
        $product \leftarrow 1$
        for $j = i + 1$ to including $dimension - 1$ do
            $product \leftarrow product * arr2[j]$
        end for
        $product2 \leftarrow product * coordinate\_array[i]$
        $index \leftarrow index + product2$
    end for
    Display " ; Value = " $arr[index]$
end for
Display "...Executed procedure 3..."
```

APPENDIX C
OUTPUT



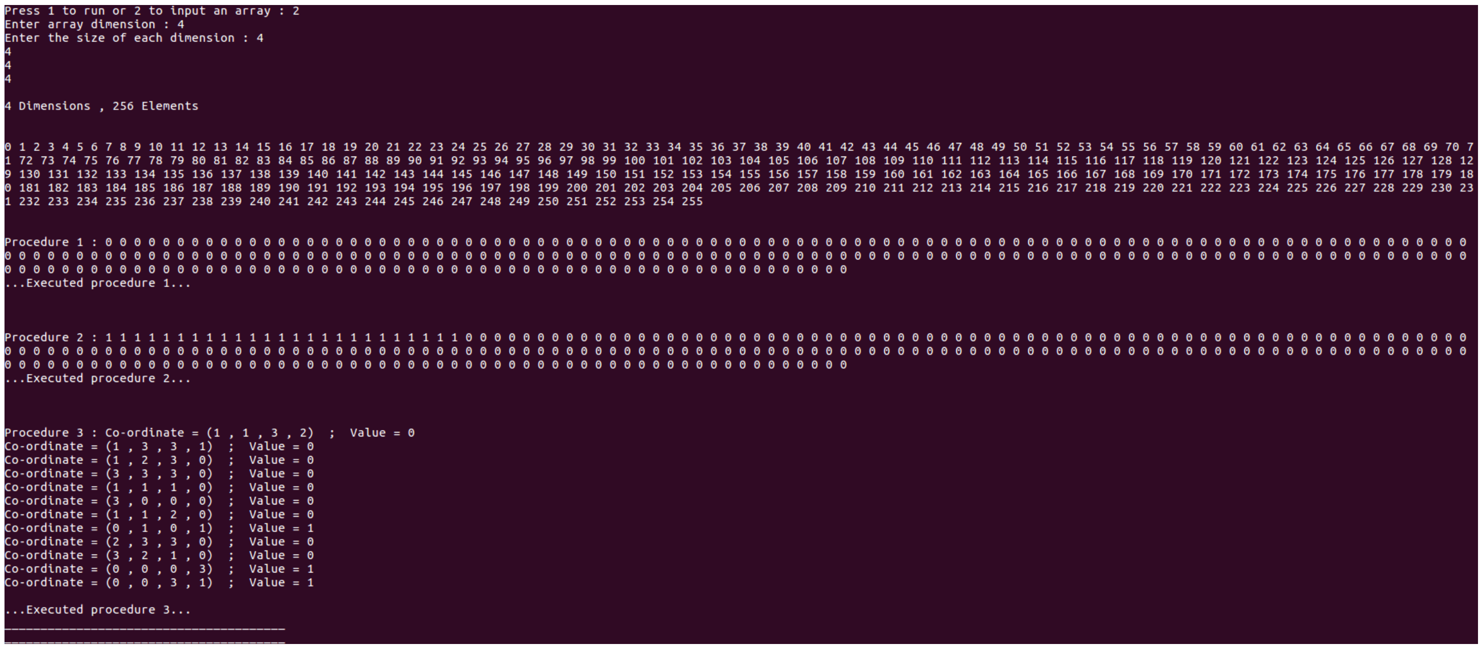
```
Press 1 to run or 2 to input an array : 2
Enter array dimension : 2
Enter the size of each dimension : 5
5

2 Dimensions , 25 Elements


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


Procedure 1 : 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
...Executed procedure 1...



Procedure 2 : 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
...Executed procedure 2...



Procedure 3 : Co-ordinate = (3 , 1)  ;  Value = 0

...Executed procedure 3...
_______________________________________
_______________________________________
```

Fig. 3. Output result of a 2-Dimensional Array of size 5, i.e. `Array[5][5]`



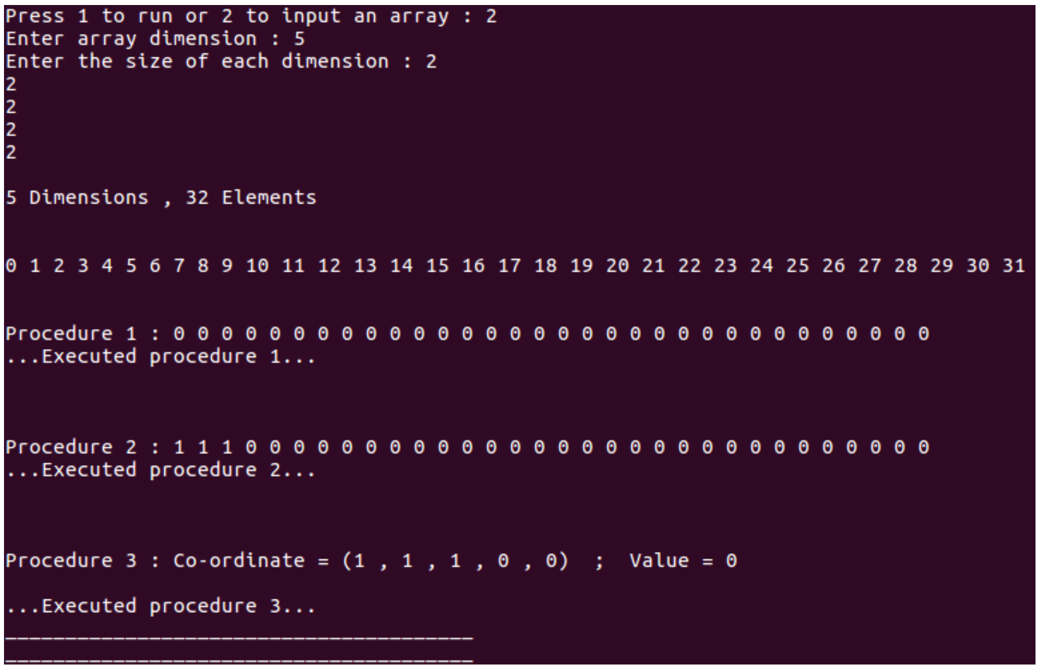
```
Press 1 to run or 2 to input an array : 2
Enter array dimension : 3
Enter the size of each dimension : 6
6
6

3 Dimensions , 216 Elements


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 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 7
1 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 12
9 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 18
0 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215


Procedure 1 : 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 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 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 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 0 0 0 0 0 0 0 0
...Executed procedure 1...



Procedure 2 : 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 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 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 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 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
...Executed procedure 2...



Procedure 3 : Co-ordinate = (2 , 3 , 5)  ;  Value = 0
Co-ordinate = (3 , 3 , 4)  ;  Value = 0
Co-ordinate = (2 , 4 , 5)  ;  Value = 0
Co-ordinate = (5 , 3 , 2)  ;  Value = 0
Co-ordinate = (5 , 0 , 0)  ;  Value = 0
Co-ordinate = (2 , 0 , 3)  ;  Value = 0
Co-ordinate = (0 , 0 , 2)  ;  Value = 1
Co-ordinate = (4 , 1 , 3)  ;  Value = 0
Co-ordinate = (2 , 0 , 0)  ;  Value = 0
Co-ordinate = (3 , 1 , 3)  ;  Value = 0

...Executed procedure 3...
_______________________________________
_______________________________________
```

Fig. 4. Output result of a 3-Dimensional Array of size 6, i.e. `Array[6][6][6]`



```
Press 1 to run or 2 to input an array : 2
Enter array dimension : 4
Enter the size of each dimension : 4
4
4
4

4 Dimensions , 256 Elements


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 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 7
1 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 12
9 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 18
0 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 23
1 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255


Procedure 1 : 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 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 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 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 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
...Executed procedure 1...



Procedure 2 : 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 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 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 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 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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
...Executed procedure 2...



Procedure 3 : Co-ordinate = (1 , 1 , 3 , 2)  ;  Value = 0
Co-ordinate = (1 , 3 , 3 , 1)  ;  Value = 0
Co-ordinate = (1 , 2 , 3 , 0)  ;  Value = 0
Co-ordinate = (3 , 3 , 3 , 0)  ;  Value = 0
Co-ordinate = (1 , 1 , 1 , 0)  ;  Value = 0
Co-ordinate = (3 , 0 , 0 , 0)  ;  Value = 0
Co-ordinate = (1 , 1 , 2 , 0)  ;  Value = 0
Co-ordinate = (0 , 1 , 0 , 1)  ;  Value = 1
Co-ordinate = (2 , 3 , 3 , 0)  ;  Value = 0
Co-ordinate = (3 , 2 , 1 , 0)  ;  Value = 0
Co-ordinate = (0 , 0 , 0 , 3)  ;  Value = 1
Co-ordinate = (0 , 0 , 3 , 1)  ;  Value = 1

...Executed procedure 3...
_______________________________________
_______________________________________
```

Fig. 5. Output result of a 4-Dimensional Array of size 4, i.e. `Array[4][4][4][4]`



```
Press 1 to run or 2 to input an array : 2
Enter array dimension : 5
Enter the size of each dimension : 2
2
2
2
2

5 Dimensions , 32 Elements


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


Procedure 1 : 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
...Executed procedure 1...



Procedure 2 : 1 1 1 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
...Executed procedure 2...



Procedure 3 : Co-ordinate = (1 , 1 , 1 , 0 , 0)  ;  Value = 0

...Executed procedure 3...
_______________________________________
_______________________________________
```

Fig. 6. Output result of a 5-Dimensional Array of size 2, i.e. `Array[2][2][2][2][2]`