ELEN4020A: Data Intensive Computing Lab 1

Sasha Berkowitz - 818737 Nick Raal - 793528 Jared Gautier - 820687 Arunima Pathania - 1117426

I. Introduction

In the lab it was required to write code to perform different procedures on N-dimentional, or multi-dimensional, arrays.

In order to make these procedures operational for any value N, the array is converted into a longer one-dimensional array representing the values of the multi-dimensional one, as described in Section 2. Each function then operates on a 1-D array, where each elements index corresponds to its position in the N-dimensional array.

The indexing runs from the first to last array dimensions, for example in a multi-dimensional array of [15][15][15], the value at position (1,0,0) will have position (1) in the final array, at position (0,1,0) will have position (16) and so on. Functions to convert between an array of co-ordinates and its corresponding final index value were implemented and used as described in Section 4.

These procedures take as input a value k, being the number of dimensions, and an array of the dimension values. They are individually discussed in sections 2 through 4.

II. Procedure 1

In the first procedure a one-dimensional array is created. The length of this array is equal to the product of the dimensions of the multidimensional arrays size. Each element in the array is then initialised to zero using a for-loop as shown in Algorithm 1.

Algorithm 1 Procedure 1

Require: Returns *finalArr* with all values set to 0

Take $values$ as input to $initialiseArray()$

$*finalArr \leftarrow malloc(values * sizeof(int))$
for $a \leftarrow 0$ to $values$ **do**
  $finalArr[a] \leftarrow 0$
end for
return $finalArr$

III. Procedure 2

The second procedure uses a for loop, iterates over every 10th element in the array and sets the value to 1. This ensures that 10% of the values are uniformely set to 1, as is shown in the pseudo code in Algorithm 2.

IV. Procedure 3

The third procedure uses a for loop and randomly prints the values of 5% of the total elements in the array. This can be shown in the pseudo code in Algorithm 3.

Additionally, it prints the co-ordinates of the array element. The procedure that converts the index of the final array to the K-dimensinal co-ordinates is shown in Algorithm 4.

Algorithm 2 Procedure 2

Require: Returns *finalArr* with 10% of values set to 1

Take $values$ and $finalArr[]$ as input to $populateOnes()$

for $a \leftarrow 0$ to $values$ **do**
  $finalArr[a] \leftarrow 1$
  $a \leftarrow a + 10$
end for
return $finalArr$

Algorithm 3 Procedure 3

Require: Uniformaly in a random fashion chooses 5% of the elements and prints $coordArr$ and $index$ of the $finalArr$

Takes $values$, $finalArr[]$, $dimArr[]$ and k as input to $printVals()$

$index \leftarrow 0$
for $a \leftarrow 0$ to $values * 0.05$ **do**
  $index \leftarrow rand()\%values$
  $*coordArr \leftarrow indexToCoord(k, dimArr, index)$
  printf $\leftarrow index$
  printf $\leftarrow$ Coordinates
  for $b \leftarrow 0$ to k **do**
    printf $\leftarrow coordArr[b]$
  end for
  printf $\leftarrow$ Value
  printf $\leftarrow finalArr[index]$
end for

Algorithm 4 indexToCoord

Require: Returns $fetchedCoordArr$, the index value to an array of coordinates

Takes k , $dimArr$ and $index$

$*fetchedCoordArr \leftarrow malloc(k * sizeOf(int))$
for $a \leftarrow 0$ to k **do**
  $fancy \leftarrow 1$
  for $b \leftarrow 0$ to $k - a - 1$ **do**
    $fancy \leftarrow fancy * dimArr[b]$
  end for
  $fetchedCoordArr[a] \leftarrow index/fancy$
  $index \leftarrow index\%fancy$
end for
return $fetchedCoordArr$