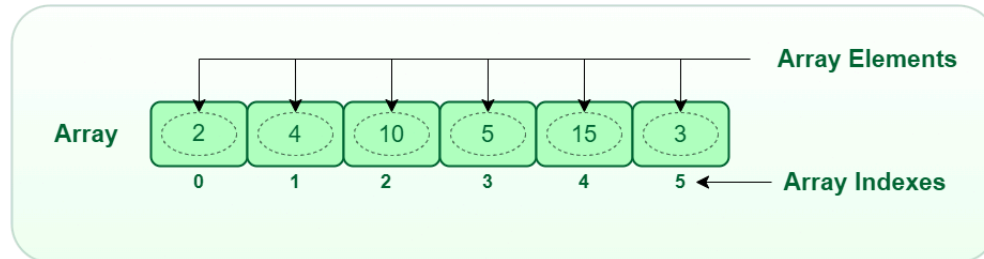


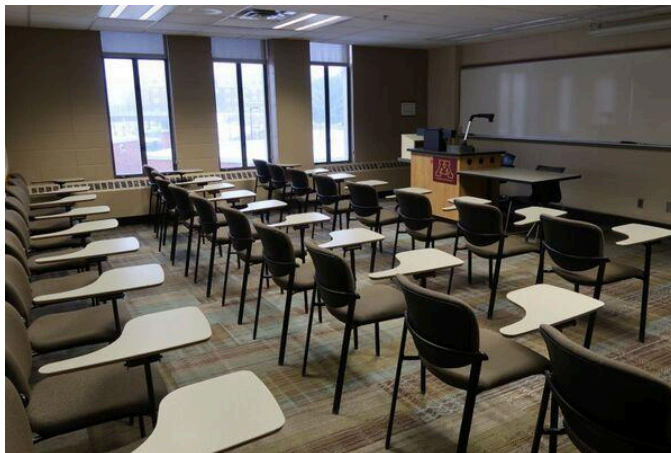
Chapter 1: Array

1.1 Array basics

1.1.1 Introduction



An array is a fixed-dimensional and indexed sequence of fix number of elements of a single type. Every aspect of this definition is important. First, an array has dimensions. So we can create 1D arrays, 2D arrays, 3D arrays, and so on. The simplest form is the one-dimensional array. Second, an array is an indexed collection. Indexed means each element has a designated position. Hence, you can retrieve an element by its position. For example, if you ask what is the value at position 2 in the above picture then the answer is 10. Third, an array has a fixed number of elements along each dimension. This count of elements along a dimension is called the length of the dimension. When people deal with a 1D array, they typically use the term length of the array to indicate the length of that sole dimension. Note that this does not mean you have to use a constant in your code as an array dimension length in your source code. It means that when you create an array in the program, you have to specify the length of each of its dimensions that you cannot change later. Finally, an array can hold elements of only a single type. That is, you cannot mix strings with numbers in an array – not even mix integer numbers with fractional numbers. All these restrictions are essential for efficient operations. In fact, arrays are by far the fastest data structures that computers can process. There is a reason for this efficiency. A computer's own memory is a 1D array of fixed-sized memory cells. Consequently, the notion of a programming language array fits directly with how computers store and access data from memory.



The sit arrangement in a classroom can be described using a 2D array. In a multidimensional array, each array element has an index for each of the array dimensions. So a 2D array of classroom seats has two indexes, a row index and a column index. The value to store in the index depends on array type. For example, you may be interested to know which seats are occupied and what are not. Then each element can be a Boolean value. However, if you want to know who sits where, then each element of the array will be a student object.

Accessing an Array Element: If you consider how the CPU of your computer reads/writes data items from the computer memory (aka the RAM) for an array, you will understand why array access is so efficient. The CPU is given the starting address of the array in memory and the index of the element program needs to access. As the size of the object an array holds can be larger than the size of a single memory cell, CPU cannot just add the index with the starting address to locate the element in the searched index. However, we know that the elements are all of the same type. So their size is fixed. Therefore, the CPU just multiply the index by the fixed object size and add that result with the starting address to determine the address of the searched element. Then it needs a single memory load/store operation to read/write the element. So accessing element by index from an array is a constant-time operation.

This also tells you why it is important to have the length of each dimension of an array immutable (means you cannot change). Because only then the language can decide how many consecutive memory cells it should reserve from the hardware memory to store the array.

1.2 Operations on Array

The most common operations that you do on an array are writing/reading a data item to/from an index. Suppose you have a 1D array named *student_names*. If you want get the 5th student from the array then you write:

```
fifth_student = student_names[5]
```

To change the value at 5th index and set it to 'John Doe', you write:

```
student_names[5] = "John Doe"
```

Typically, you do not shift the positions of elements of an array, unless you are swapping the positions of two elements (often needed for sorting the array in increasing or decreasing order of element values). Similarly inserting a new element at a particular index or removing an element from that index is also atypical as that involves shifting the positions of other elements. You avoid such operations on an array as they are costly. However, it is common to create a new array from an existing array where the new array has a filtered subset of elements from the old array or have the same elements but in a different order. There is another basic data structure that is highly efficient for that, it is called a *List*.

Determining the Length of an Array: it is a curious question, how do you know the dimension lengths of an array. Languages that are focused on the best performance say that the programmers should store the length of each dimension in some other variable. This makes sense as you provide the dimension lengths when you create the array. This means that when you write something like *array[i]* to access the *i*th element of the array, the language does not check if *i* is crossing the length of the array. In other word, there is no checking if the index you gave is valid or not. If you give an invalid index, you get invalid data. Examples of such languages are C, Fortran, and Golang.

On the other hand, some other languages, particularly object oriented languages, store the length information you supplied in the first memory cell of the array and you cannot access that location from your program. When you write `array[i]` in such a language, the generated code puts the array access inside a conditional if-else block. The if-else block checks if *i* is greater than equal to 0 and less than the length of the array. If that is not true then you get an error when your program executes the statement containing `array[i]`. Examples of such languages are Java and C#.

The tradeoff here is that accessing an array element in Java/C# is two to three times slower than accessing an array element in C/Fortran/Golang. However, if you are a careless programmer then Java/C# gives better protection that you will not make errors. This efficiency of array access is one of the main reasons critical system software such as the operating systems and compilers are written in C.

1.3 Linear Array Properties

- I. All the elements will be of the same data type
- II. All the non-dummy values will be at the left followed by the dummy values (zero/None/null)
- III. Length of an array is fixed once declared and cannot be changed

1.3.1. Ways of Creating a Numpy Array

```
1. FUNCTION create_array(size)
2.   array = new Array(size)
3.   FOR i = 0 TO size-1
4.     array[i] = 0
5.   END FOR
6.   RETURN array
7. END FUNCTION
8. arr = create_array(5)
```

1.3.2 Linear Array Operations

- I. Iteration
- II. Resize
- III. Copy (Pass by Value)
- IV. Shift: Left and Right
- V. Rotate: Left and Right
- VI. Reverse: Out-of-place and In-place
- VII. Insert: At the end or anywhere else
- VIII. Delete: Last element or any other element

I. Iteration

Iteration refers to checking all the values index by index. The main idea is to go to that memory location and check all the values index by index. The time complexity of this is $O(n)$. The space complexity is $O(1)$.

```

FUNCTION Iterate(arr)
  FOR element IN arr
    PRINT element
  END FOR
END FUNCTION

```

II. Resize

We can not resize an array because it has a fixed memory location. However, if we ever need to resize an array, we need to create a new array with a new length and then copy the values from the original array. For example, if we have an array [10, 20, 30, 40, 50] whose length is 5 and want to resize the array with length 8. The new array will be [10, 20, 30, 40, 50, None, None, None]. Here None is representing empty values. The time complexity of this is $O(n)$. The space complexity is $O(n)$.

```

FUNCTION Resize(arr, newSize)
  CREATE resized ARRAY of size newSize
  FOR i FROM 0 TO MINIMUM OF arr.length AND newSize
    resized[i] = arr[i]
  END FOR
  RETURN resized
END FUNCTION

```

III. Copy (Pass by Value)

Copy array means you initialize a new array with the same length as the given array to copy and then copy the old array's value by value. As the variable where we store the array only stores the memory location, only copying the value is not enough for array copying. For example, if you have an array titled `arr = [1, 2, 3, 4]` and write `a2 = arr`. It does not mean you have copied `arr` to `a2`. The time complexity of this is $O(n)$. The space complexity is $O(n)$.

```

FUNCTION CopyArray(arr)
  CREATE copy ARRAY of size arr.length
  FOR i FROM 0 TO arr.length - 1
    copy[i] = arr[i]
  END FOR
  RETURN copy
END FUNCTION

```

IV. Shift: Left and Right

Shifting an Array Left: Shifting an entire array left moves each element one (or more, depending how the shift amount) position to the left. Obviously, the first element in the array will fall off at the beginning and be lost forever. The last slot of the array before the shift (ie. the slot where the last element was until the shift) is now unused (we can put a None there to signify that). For example, shifting the array [5, 3, 9, 13, 2] left by one position will result in the array [3,

9, 13, 2, None]. Note how the array[0] element with the value of 5 is now lost, and there is an empty slot at the end. The time complexity of this is $O(n)$. The space complexity is $O(1)$.

```
FUNCTION ShiftLeft(arr)
  IF arr.length == 0 THEN RETURN
  FOR i FROM 0 TO arr.length - 2
    arr[i] = arr[i + 1]
  END FOR
  arr[arr.length - 1] = 0
END FUNCTION
```

Shifting an Array Right: Shifting an entire array right moves each element one (or more, depending how the shift amount) position to the right. Obviously, the last element in the array will fall off at the end and be lost forever. The first slot of the array before the shift (ie., the slot where the first element was until the shift) is now unused (we can put a None there to signify that). The size of the array remains the same however because the assumption is that you would put something in the now-unused slot. For example, shifting the array [5, 3, 9, 13, 2] right by one position will result in the array [None, 5, 3, 9, 13]. Note how the array[4] element with the value of 2 is now lost, and there is an empty slot at the beginning. The time complexity of this is $O(n)$. The space complexity is $O(1)$.

```
FUNCTION ShiftRight(arr)
  IF arr.length == 0 THEN RETURN
  FOR i FROM arr.length - 1 DOWNTO 1
    arr[i] = arr[i - 1]
  END FOR
  arr[0] = 0
END FUNCTION
```

V. Rotate: Left and Right

Rotating an Array Left: Rotating an array left is equivalent to shifting a circular or cyclic array left where the 1st element will not be lost, but rather move to the last slot. Rotating the array [5, 3, 9, 13, 2] left by one position will result in the array [3, 9, 13, 2, 5]. The time complexity of this is $O(n)$. The space complexity is $O(1)$.

```
FUNCTION RotateLeft(arr)
  IF arr.length == 0 THEN RETURN
  SET first = arr[0]
  FOR i FROM 0 TO arr.length - 2
    arr[i] = arr[i + 1]
  END FOR
  arr[arr.length - 1] = first
END FUNCTION
```

Rotating an Array Right: Rotating an array right is equivalent to shifting a circular or cyclic array right where the last element will not be lost, but rather move to the 1st slot. Rotating the array [5, 3, 9, 13, 2] right by one position will result in the array [2, 5, 3, 9, 13]. The time complexity of this is $O(n)$. The space complexity is $O(1)$.

```
FUNCTION RotateRight(arr)
  IF arr.length == 0 THEN RETURN
  SET last = arr[arr.length - 1]
  FOR i FROM arr.length - 1 DOWNTO 1
    arr[i] = arr[i - 1]
  END FOR
  arr[0] = last
END FUNCTION
```

VI. Reverse: Out-of-place and In-place

Reversing an array can be implemented in two ways. First, we will create a new array with the same size as the original array and then copy the values in reverse order. The method is called out-of-the-place operation.

However, an efficient approach might be to reverse the array in the original array. By this, we will not need to allocate extra spaces. This is known as an in-place operation. To do so we need to start swapping values from the beginning position to the end position. The idea is to swap starting value with the end value, then the second value with the second last value, and so on. The time complexity of this is $O(n)$. The space complexity is $O(1)$.

```
FUNCTION ReverseOutOfPlace(arr)
  CREATE reversed ARRAY of size arr.length
  FOR i FROM 0 TO arr.length - 1
    reversed[i] = arr[arr.length - 1 - i]
  END FOR
  RETURN reversed
END FUNCTION
```

The time complexity of this is $O(n)$. The space complexity is $O(1)$.

```
FUNCTION ReverseInPlace(arr)
  SET left = 0, right = arr.length - 1
  WHILE left < right
    SWAP arr[left] WITH arr[right]
    INCREMENT left
    DECREMENT right
  END WHILE
```

END FUNCTION

Introducing New Linear Array Property: Size

- Size indicates the number of non-dummy values in an array.
- Size of an array can never exceed the length of that array. Which means size > length is not possible.
- Size cannot be negative.
- Size is used in array insertion or deletion, or any task that involves working with the non-dummy values only.
- If size < length, insertion in an array is possible, otherwise must resize the array.
- If size > 0, at least one element from the array can be deleted, otherwise, there is nothing to delete.

VII. Insert: At the end or anywhere else

- At first, check whether insertion is possible or not. If not possible, resize the array
- Valid places of inserting: index 0 to size
- If you are inserting at the end of the array, no shifting is needed
- If you are inserting anywhere else but at the end of an array, then the subsequent elements must be right shifted before insertion
- Increase size after insertion

The time complexity of this is $O(n)$. The space complexity is $O(1)$ or $O(n)$ (If resize requires)

```
FUNCTION Insert(arr, index, value, currentSize)
  IF index < 0 OR index > currentSize THEN THROW Exception
  IF currentSize == arr.length THEN
    arr = Resize(arr, arr.length * 2)
  END IF
  FOR i FROM currentSize DOWNT0 index + 1
    arr[i] = arr[i - 1]
  END FOR
  arr[index] = value
  RETURN arr
END FUNCTION
```

VIII. Remove: Last element or any other element

- At first, check whether deletion is possible or not
- Valid places of deletion: index 0 to size-1
- If you are deleting the last element of the array, no shifting is needed

- If you are deleting any other element except for the last element, then the subsequent elements must be left shifted after deletion
- Decrease size after deletion

```
FUNCTION Delete(arr, index, currentSize)
  IF index < 0 OR index >= currentSize THEN THROW Exception
  FOR i FROM index TO currentSize - 2
    arr[i] = arr[i + 1]
  END FOR
  arr[currentSize - 1] = 0
END FUNCTION
```

The time complexity of this is $O(n)$. The space complexity is $O(1)$.

Practice Problem 1: Check Palindrome

Look up what a palindrome is. Write a function that checks whether an array is a plaindrome or not.

```
def palindrome(arr, size):
    pass
```

Practice Problem 2: Reverse Print

Write a function Rev_Print that Reverse iterates and prints all the non-dummy values of an array.

Function Call:

```
arr= np.array([1 3 2 5 0 0])
print(Rev_Print(arr, 4)) #Array, size
Output: 5 2 3 1
```

Practice Problem 3: Merge Two Arrays

Write a function Merge_Arrays that takes two arrays and the sizes of those two arrays (total 4 parameters) and mergers the two arrays. There will be no dummy value in the merged array.

Function Call:

```
arr1= np.array([1 5 0 0])
arr2= np.array([3 5 2 0])
print(Merge_Arrays(arr1, 2, arr2, 3)) #Array1, Array1 size, Array2, Array2 size
Output: [1 5 3 5 2]
```


1.4 Multidimensional Array

1.4.1 Introduction

You know the basics of arrays and understand why they are important. However, we focused mostly on linear or 1D array in our earlier discussion. Let us focus on multidimensional arrays now. Multidimensional arrays give us a natural way to deal with real-world objects that are multidimensional, and such objects are quite frequent. Consider an image rendering operation to display a picture on the screen. It is convenient to use a 2D array of pixels to represent the picture to decide how to magnify and align the pixels of the image with the points of the 2D screen. For another example, if you do computation related to heat propagation on a 3D object as part of a scientific study of solid materials' heat conductivity; it is natural to use 3D arrays to represent the solids. Take this second example. Suppose you want to read the current temperature at the $\langle x, y, z \rangle$ point of a 3D plate variable representing a solid, you can simply do this as follows:

$$cell_temperature = plate[x][y][z]$$

You see how convenient it is to write computations using multidimensional arrays! Fortunately, an element access from multidimensional arrays is as efficient as it is convenient. To understand the efficiency, we need to see how languages allocate multidimensional arrays and locate the memory address for a particular multidimensional index.

1.4.2 Multidimensional Arrays in Memory

Interestingly – but not surprisingly – programming languages store multidimensional arrays as linear arrays in the RAM. This makes sense, right? As we know the RAM is a linear array of memory cells. Let us see how it works. Remember that an array contains elements of a single type and its number of dimensions and dimension lengths are given when you create it. Therefore, a language can follow the simple technique of placing all elements for increasing values of the index along a single dimension in consecutive locations in the memory while keeping the indexes along all other dimensions fixed. After it is done traversing a dimension, then it increases the value of the index along another dimension and restart from the zero index of the previous dimension. Following this process until the indexes along all dimensions become the maximum possible, translates the whole multidimensional array into a linear array. There is just one restriction; a language must follow a fixed dimension ordering rule to store all multidimensional arrays. There are two standards here:

1. Increasing the final dimension's index first and progressively move to earlier dimensions, this is called the Row-Major Ordering. For example, Java and C take this approach.
2. Increasing the starting dimension's index first and progressively move to later dimensions, this is called the Column-Major Ordering. For example, FORTRAN takes this approach.

The following diagram shows the difference between these two approaches for a 2D matrix of 3 × 3 dimensions. (By the way, row is the vertical and column is the horizontal axis in 2D. You should already know that. Here Dimension 0 represents the row and Dimension 1 represents the column.)

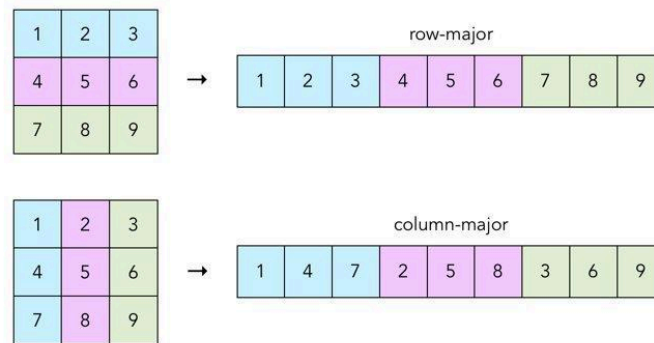


Figure 1: Two Different Ways of Storing Multidimensional Arrays in a Linear Format

There is a great impact of different languages choosing different ordering of dimensions for storing multidimensional arrays. The impact is related to what is efficient in computer hardware and what is not. The way modern computers are built, it is several times more efficient to access data to/from consecutive memory locations than from locations that are distant from one another. As a result, if you iterate the elements of a multidimensional array in an order that is different from how they are stored in memory then your program can be several times slower. Now given the index ordering chosen by Java is radically opposite from how FORTRAN does it, if you simply translate an efficient Java code into an equivalent FORTRAN code then the performance will tank.

The important lesson here is that efficient programming is often language dependent.

The following diagram shows how the row-major approach works for a 3D $3 \times 3 \times 3$ cube. From this example, you understand the higher dimension count of the original multidimensional array is not a problem.

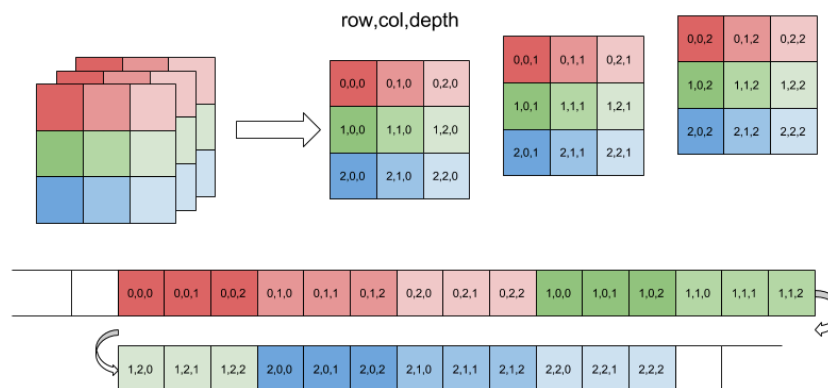


Figure 2: Row Major Ordering of Indexes of a 3D Array

Let us stick to the row-major ordering, as that is more common in application programming. You should be interested to know how to determine the translated linear index of a multidimensional index from the original array. For that, imagine you have a 4D array with dimensions $M \times N \times O \times P$, named *box*, and the element at the index you are interested in is the *box*[*w*][*x*][*y*][*z*]. Then the index of that element in the linear array is:

$$w \times (N \times O \times P) + x \times (O \times P) + y \times P + z$$

You understand the general rule, right? Basically multiply the upper dimension indexes by the multiplication of the lengths of lower dimensions then add them all. So easy, isn't it?

Do a practice of the reverse operation. Suppose a linear array of length 128 actually stores a 3D array of dimensions $4 \times 4 \times 8$. What are the multidimensional indexes of the element stored at location 111? Remember that indexing starts from 0 in each dimension.

Since it is so simple to represent multidimensional arrays as linear arrays in memory, some languages do not support multidimensional arrays as building block data structures at all. They only support linear arrays. The idea is, since it is such a simple computation to go back and forth between multidimensional and linear arrays, let programmers handle the logic of index conversions and keep the language simple. C is a classic example of such a language that does not allow creation of multidimensional arrays dynamically (which means during program runtime).

You have to admit that this is a valid argument.

Solution of the Example Problem:

The dimension of the given array is $[4][4][8]$ and length is 128. We need to map the dimension of linear index 111.

The equation we get is:

$$X * (4*8) + Y*8 + Z = 111 \text{ where } 0 \leq X \leq 3, 0 \leq Y \leq 3 \text{ and } 0 \leq Z \leq 7.$$

Now to find the value of X, Y and Z we need to repeatedly divide the remainder of the linear index with the multiplication of the lengths of lower dimensions till you reach the last dimension. So first $111/(4*8)$ then the quotient is x and then you use the remainder R to do $R/8$ to get y index. The remainder of that is the z index

Following the process we get,

$$X = 111 // (4*8) = 3 \text{ and } 111 \% (4*8) = 15$$

$$Y = 15 // 8 = 1 \text{ and } 15 \% 8 = 7$$

$$Z = 7$$

So, the dimension of linear index 111 is $[3][1][7]$.

Now practice finding the dimension of 96, 107, and 60 of the linear index.

1.4.3 Initialization of a Multidimensional Array

For **Python**, we can use the [NumPy](#) library for initializing a multidimensional array. On the other hand, for Java we can directly create a multidimensional array.

Note: To use the [NumPy](#) library we have to import the numpy library first.

For Python, to create an empty array, we can use the **zeros()** function of **NumPy** and for Java it can be created by using extra square brackets (`[]`) when declaring the array .

Python	Java
<pre>import numpy as np m = np.zeros((2,3), dtype=int)</pre>	<pre>int [][] arr = new int[2][3];</pre>
Here, (2,3) indicates the dimension [namely 2 rows and 3 columns] and dtype indicates datatype. If dtype is not given then the default datatype of a numpy array is float.	Here, [2][3] indicates the dimension [namely 2 rows and 3 columns respectively]

For Python, we can also create a multidimensional array by using the **array()** function of **NumPy** and for Java we can initialize it as Array of Arrays.

Python	Java
<pre>matrix = np.array([[4,3,8], [2,5,1]])</pre>	<pre>int [][] matrix = { {4,3,8}, {2,5,1} };</pre>
The resulting array, matrix will be a 2x3 matrix as shown below: 4 3 8 2 5 1	

For simplicity's sake, we shall only be dealing with 2D matrices in this lecture.

1.4.4 Creating a 2D Matrix

```
FUNCTION create2DArray(rows, cols, initialValue):
    DECLARE matrix[rows][cols]
    FOR i FROM 0 TO rows - 1:
        FOR j FROM 0 TO cols - 1:
            matrix[i][j] = initialValue
    RETURN matrix
```

1.4.5 Iteration of a 2D Matrix

1. Row Wise Iteration

```
FUNCTION iterateRows(matrix):
    FOR EACH row IN matrix:
        FOR EACH value IN row:
            PRINT value
        PRINT newline
```

The time complexity of this is $O(m*n)$. The space complexity is $O(1)$. Here row = m, column = n

2. Column Wise Iteration:

```
FUNCTION iterateColumns(matrix):
    rows = LENGTH(matrix)
    cols = LENGTH(matrix[0])
    FOR j FROM 0 TO cols - 1:
        FOR i FROM 0 TO rows - 1:
            PRINT matrix[i][j]
        PRINT newline
```

The time complexity of this is $O(m*n)$. The space complexity is $O(1)$. Here row = m, column = n

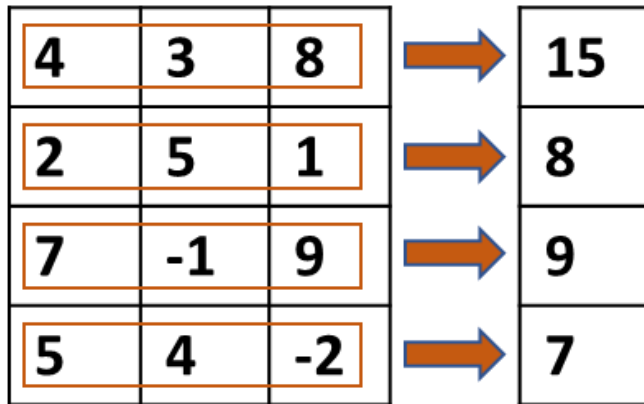
1.4.6 Summation

1. Sum of all elements in a 2D matrix:

```
FUNCTION sumAllValues(matrix):
    sum = 0
    FOR EACH row IN matrix:
        FOR EACH value IN row:
            sum += value
    RETURN sum
```

The time complexity of this is $O(m*n)$. The space complexity is $O(1)$. Here row = m, column = n

2. Sum of every row in a given matrix



Thus, the resulting matrix will always have 1 column to store row wise sum and equal number of rows of the main matrix.

```
FUNCTION sumRows(matrix):  
    DECLARE rowSums[LENGTH(matrix)]  
    FOR i FROM 0 TO LENGTH(matrix) - 1:  
        sum = 0  
        FOR EACH value IN matrix[i]:  
            sum += value  
        rowSums[i] = sum  
    RETURN rowSums
```

The time complexity of this is $O(m*n)$. The space complexity is $O(m)$. Here row = m, column = n

3. Sum of every column in a given matrix

4	3	8
2	5	1
7	-1	9
5	4	-2

18	11	16
----	----	----

Thus, the resulting matrix will always have 1 row to store column wise sum and equal number of columns of the main matrix.

```
FUNCTION sumColumns(matrix):  
    rows = LENGTH(matrix)  
    cols = LENGTH(matrix[0])  
    DECLARE colSums[cols]  
    FOR j FROM 0 TO cols - 1:  
        sum = 0  
        FOR i FROM 0 TO rows - 1:  
            sum += matrix[i][j]  
        colSums[j] = sum  
    RETURN colSums
```

The time complexity of this is $O(m*n)$. The space complexity is $O(n)$. Here row = m, column = n

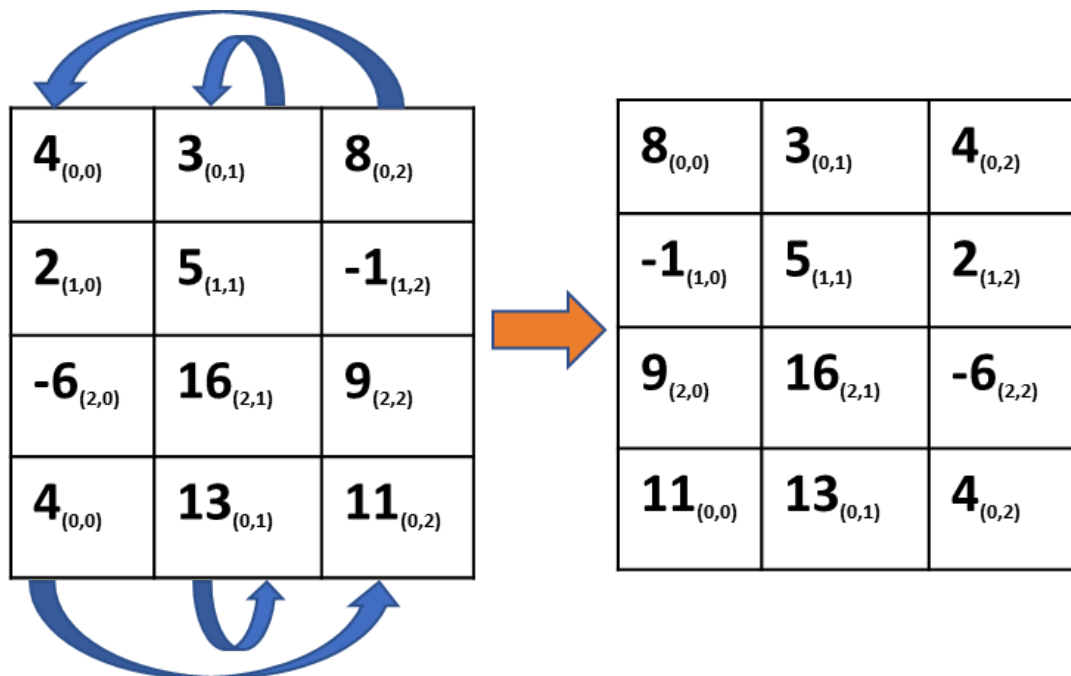
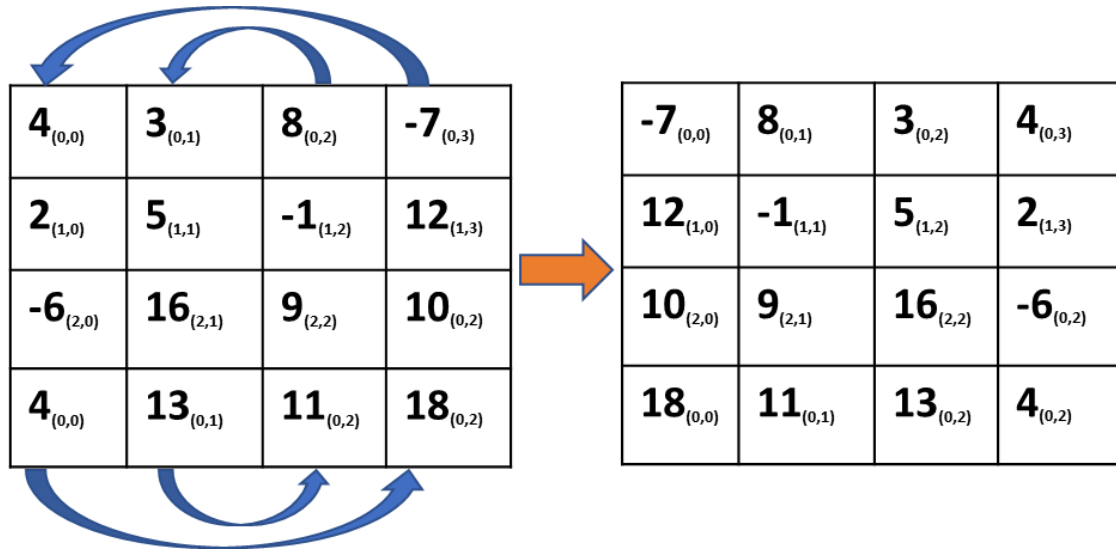
1.4.7 Swap the columns of a m x n matrix

0^{th} column \rightleftharpoons $(n-1)^{\text{th}}$ column

1st column \Rightarrow (n-2)th column

2nd column \Rightarrow (n-3)th column

and so on and so forth



```
FUNCTION swapColumns(matrix):
    rows = LENGTH(matrix)
    cols = LENGTH(matrix[0])
```



```

mid = cols // 2
FOR j FROM 0 TO mid - 1:
    col1 = j
    col2 = cols - 1 - j
    FOR i FROM 0 TO rows - 1:
        temp = matrix[i][col1]
        matrix[i][col1] = matrix[i][col2]
        matrix[i][col2] = temp

```

The time complexity of this is $O(m*n)$. The space complexity is $O(1)$. Here row = m, column = n

1.4.8 Addition:

1. Add the elements of the primary diagonal in a square matrix

4 _(0,0)	3 _(0,1)	8 _(0,2)
2 _(1,0)	5 _(1,1)	1 _(1,2)
7 _(2,0)	6 _(2,1)	9 _(2,2)

The main diagonal of a square matrix are the elements whose row number and column number are equal, a_{ii}

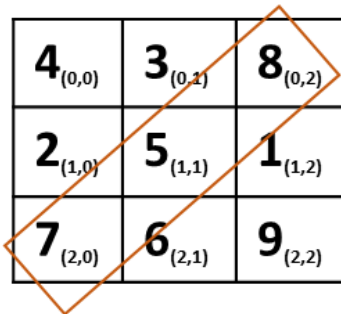
```

FUNCTION sumPrimaryDiagonal(matrix):
    sum = 0
    FOR i FROM 0 TO LENGTH(matrix) - 1:
        sum += matrix[i][i]
    RETURN sum

```

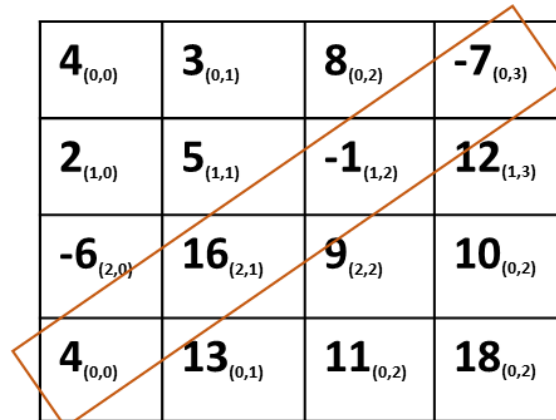
The time complexity of this is $O(m)$. The space complexity is $O(1)$. Here row = m, column = n

2. Add the elements of the secondary diagonal in a square matrix



4 _(0,0)	3 _(0,1)	8 _(0,2)
2 _(1,0)	5 _(1,1)	1 _(1,2)
7 _(2,0)	6 _(2,1)	9 _(2,2)

Secondary diagonal of a 3x3 Matrix



4 _(0,0)	3 _(0,1)	8 _(0,2)	-7 _(0,3)
2 _(1,0)	5 _(1,1)	-1 _(1,2)	12 _(1,3)
-6 _(2,0)	16 _(2,1)	9 _(2,2)	10 _(2,3)
4 _(3,0)	13 _(3,1)	11 _(3,2)	18 _(3,3)

Secondary diagonal of a 4x4 Matrix

For an element a_{ij} in the secondary diagonal, can you find out the j for a particular i ?
Hint: try $i+j$ for a_{ij} in the secondary diagonal and find out the relation.

```
FUNCTION sumSecondaryDiagonal(matrix):
    sum = 0
    n = LENGTH(matrix)
    FOR i FROM 0 TO n - 1:
        sum += matrix[i][n - 1 - i]
    RETURN sum
```

3. Add two matrices of same dimension

```
FUNCTION addMatrices(matrix1, matrix2):
    rows = LENGTH(matrix1)
    cols = LENGTH(matrix1[0])
    DECLARE result[rows][cols]
    FOR i FROM 0 TO rows - 1:
        FOR j FROM 0 TO cols - 1:
            result[i][j] = matrix1[i][j] + matrix2[i][j]
    RETURN result
```

Both the time complexity and space complexity of this is $O(m*n)$. Here row = m , column = n .

1.4.9 Multiply two matrices

```
FUNCTION multiplyMatrices(matrix1, matrix2):
    rows1 = LENGTH(matrix1)
    cols1 = LENGTH(matrix1[0])
    cols2 = LENGTH(matrix2[0])
    DECLARE result[rows1][cols2]
    FOR i FROM 0 TO rows1 - 1:
        FOR j FROM 0 TO cols2 - 1:
            result[i][j] = 0
            FOR k FROM 0 TO cols1 - 1:
                result[i][j] += matrix1[i][k] * matrix2[k][j]
    RETURN result
```

Suppose the dimension of matrix1 is $(m \times n)$ and dimension of matrix2 is $(n \times p)$. Then the time complexity of the multiplication is $O(m \times n \times p)$. The space complexity is $O(m \times p)$.

Exercises

1.1: Given two arrays `a[]` and `b[]` of size `n` and `m` respectively. The task is to find union between these two arrays.

Union of the two arrays can be defined as the set containing distinct elements from both the arrays. If there are repetitions, then only one occurrence of element should be printed in the union.

Example 1:

Input:

5 3

1 2 3 4 5

1 2 3

Output:

5

Explanation:

1, 2, 3, 4 and 5 are the elements which comes in the union set of both arrays. So the count is 5.

Example 2:

Input:

6 2

85 25 1 32 54 6

85 2

Output:

7

Explanation:

85, 25, 1, 32, 54, 6, and 2 are the elements which comes in the union set of both arrays. So the count is 7.

1.2: Given an unsorted array `arr[]` of size `N` having both negative and positive integers. The task is place all negative element at the end of array without changing the order of positive element and negative element.

Example 1:

Input :

`N = 8`

`arr = [1, -1, 3, 2, -7, -5, 11, 6]`

Output :

`1 3 2 11 6 -1 -7 -5`

Example 2:

Input :

`N=8`

`arr = [-5, 7, -3, -4, 9, 10, -1, 11]`

Output :

`7 9 10 11 -5 -3 -4 -1`

1.3: Given an unsorted array `A` of size `N` that contains only non-negative integers, find a continuous sub-array which adds to a given number `S`.

In case of multiple subarrays, return the subarray which comes first on moving from left to right.

Example 1:

Input:

N = 5, S = 12

A = [1,2,3,7,5]

Output: 2 4

Explanation: The sum of elements from 2nd position to 4th position is 12.

Example 2:

Input:

N = 10, S = 15

A = [1,2,3,4,5,6,7,8,9,10]

Output: 1 5

Explanation: The sum of elements from 1st position to 5th position is 15.

1.4: Given an array of N positive integers and an integer X. The task is to find the frequency of X in the array.

Example 1:

Input:

N = 5

arr = [1, 1, 1, 1, 1]

X = 1

Output:

5

Explanation: The frequency of 1 is 5.

1.5: Given an array arr and an integer K where K is smaller than size of array, the task is to find the Kth smallest element in the given array. It is given that all array elements are distinct.

Example 1:

Input:

N = 6

arr = [7, 10, 4, 3, 20, 15]

K = 3

Output : 7

Explanation : 3rd smallest element in the given array is 7.

Example 2:

Input:

N = 5

arr = [7, 10, 4, 20, 15]

K = 4

Output : 15

Explanation : 4th smallest element in the given array is 15.

1.6: Given an array of size N containing only 0s, 1s, and 2s; sort the array in ascending order.

Example 1:

Input:

N = 5

arr = [0, 2, 1, 2, 0]

Output:

0 0 1 2 2

Explanation: 0s 1s and 2s are segregated into ascending order.

Example 2:

Input:

N = 3

arr = [0, 1, 0]

Output:

0 0 1

Explanation: 0s 1s and 2s are segregated into ascending order.

1.7: Given an array arr of N non-negative integers representing the height of blocks. If the width of each block is 1, compute how much water can be trapped between the blocks during the rainy season.

Example 1:

Input:

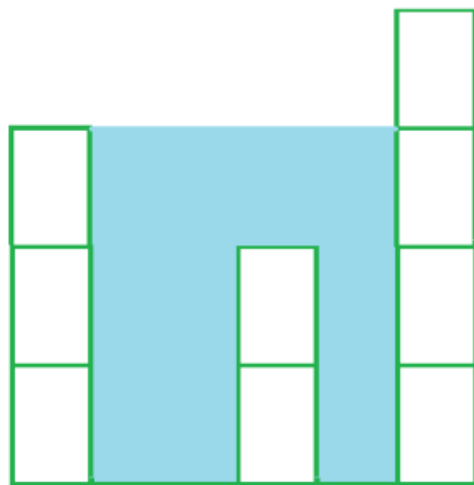
N = 6

arr = [3,0,0,2,0,4]

Output:

10

Explanation:



Bars for input {3, 0, 0, 2, 0, 4}

Total trapped water = 3 + 3 + 1 + 3 = 10

Example 2:

Input:

N = 4

arr = [7,4,0,9]

Output:

10

Explanation: Water trapped by above block of height 4 is 3 units and above block of height 0 is 7 units. So, the total unit of water trapped is 10 units.

Example 3:

Input:

N = 3

arr = [6,9,9]

Output:

0

Explanation: No water will be trapped.

1.8: Given a sorted array arr of distinct integers. Sort the array into a wave-like array(In Place). In other words, arrange the elements into a sequence such that $arr[1] \geq arr[2] \leq arr[3] \geq arr[4] \leq arr[5]$ If there are multiple solutions, find the lexicographically smallest one.

Example 1:

Input:

n = 5

arr = [1,2,3,4,5]

Output: 2 1 4 3 5

Explanation: Array elements after sorting it in wave form are 2 1 4 3 5.

Example 2:

Input:

n = 6

arr = [2,4,7,8,9,10]

Output: 4 2 8 7 10 9

Explanation: Array elements after sorting it in wave form are 4 2 8 7 10 9.

1.9: Given an array A of N elements. Find the majority element in the array. A majority element in an array A of size N is an element that appears more than $N/2$ times in the array.

Example 1:

Input:

N = 3

A = [1,2,3]

Output:

-1

Explanation: Since, each element in [1,2,3] appears only once so there is no majority element.

Example 2:

Input:

N = 5

A = [3,1,3,3,2]

Output:

3

Explanation: Since, 3 is present more than $N/2$ times, so it is the majority element.

1.10: Given an array of N integers arr where each element represents the max length of the jump that can be made forward from that element. Find the minimum number of jumps to reach

the end of the array (starting from the first element). If an element is 0, then you cannot move through that element.

Note: Return -1 if you can't reach the end of the array.

Example 1:

Input:

N = 11

arr = [1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9]

Output: 3

Explanation: First jump from 1st element to 2nd element with value 3. Now, from here we jump to 5th element with value 9, and from here we will jump to the last.

Example 2:

Input :

N = 6

arr = [1, 4, 3, 2, 6, 7]

Output: 2

Explanation: First we jump from the 1st to 2nd element and then jump to the last element.

1.11: Given an array arr denoting heights of N towers and a positive integer K. For each tower, you must perform exactly one of the following operations exactly once.

- Increase the height of the tower by K
- Decrease the height of the tower by K

Find out the minimum possible difference between the height of the shortest and tallest towers after you have modified each tower.

Note: It is compulsory to increase or decrease the height by K for each tower. After the operation, the resultant array should not contain any negative integers.

Example 1:

Input:

K = 2, N = 4

Arr = [1, 5, 8, 10]

Output:

5

Explanation: The array can be modified as [3, 3, 6, 8]. The difference between the largest and the smallest is $8-3 = 5$.

Example 2:

Input:

K = 3, N = 5

Arr = [3, 9, 12, 16, 20]

Output:

11

Explanation: The array can be modified as [6, 12, 9, 13, 17]. The difference between the largest and the smallest is $17-6 = 11$.

1.12: Given arrival and departure times of all trains that reach a railway station. Find the minimum number of platforms required for the railway station so that no train is kept waiting.

Consider that all the trains arrive on the same day and leave on the same day. Arrival and departure time can never be the same for a train but we can have arrival time of one train equal to departure time of the other. At any given instance of time, same platform can not be used for

both departure of a train and arrival of another train. In such cases, we need different platforms.
Note: Time intervals are in the 24-hour format(HHMM) , where the first two characters represent hour (between 00 to 23) and the last two characters represent minutes (this may be > 59).

Example 1:

Input: n = 6

arr = [0900, 0940, 0950, 1100, 1500, 1800]

dep = [0910, 1200, 1120, 1130, 1900, 2000]

Output: 3

Explanation: Minimum 3 platforms are required to safely arrive and depart all trains.

Example 2:

Input: n = 3

arr = [0900, 1100, 1235]

dep = [1000, 1200, 1240]

Output: 1

Explanation: Only 1 platform is required to safely manage the arrival and departure of all trains.

1.13: Given an array A of positive integers of size N, where each value represents the number of chocolates in a packet. Each packet can have a variable number of chocolates. There are M students, the task is to distribute chocolate packets among M students such that :

1. Each student gets exactly one packet.
2. The difference between maximum number of chocolates given to a student and minimum number of chocolates given to a student is minimum.

Example 1:

Input:

N = 8, M = 5

A = [3, 4, 1, 9, 56, 7, 9, 12]

Output: 6

Explanation: The minimum difference between maximum chocolates and minimum chocolates is $9 - 3 = 6$ by choosing following M packets : [3, 4, 9, 7, 9].

Example 2:

Input:

N = 7, M = 3

A = [7, 3, 2, 4, 9, 12, 56]

Output: 2

Explanation: The minimum difference between maximum chocolates and minimum chocolates is $4 - 2 = 2$ by choosing following M packets : [3, 2, 4].

1.14: Given an array of positive integers. Find the length of the longest sub-sequence such that elements in the subsequence are consecutive integers, the consecutive numbers can be in any order.

Example 1:

Input:

N = 7

a = [2,6,1,9,4,5,3]

Output:

6

Explanation: The consecutive numbers here are 1, 2, 3, 4, 5, 6. These 6 numbers form the longest consecutive subsequence.

Example 2:

Input:

N = 7

a = [1,9,3,10,4,20,2]

Output:

4

Explanation: 1, 2, 3, 4 is the longest consecutive subsequence.

1.15: Given two sorted arrays nums1 and nums2 of size m and n respectively, print the median of the two sorted arrays.

Example 1:

Input: nums1 = [1,3], nums2 = [2]

Output: 2.00000

Explanation: merged array = [1,2,3] and median is 2.

Example 2:

Input: nums1 = [1,2], nums2 = [3,4]

Output: 2.50000

Explanation: merged array = [1,2,3,4] and median is $(2 + 3) / 2 = 2.5$