

Data Structure: Arrays

Dr. Arup Kumar Pal

Department of Computer Science & Engineering

Indian Institute of Technology (ISM), Dhanbad

Jharkhand-826004

E-mail: arupkrpal@iitism.ac.in

Outline

- Introduction
- One-Dimensional Array
 - Memory Allocation for an Array
 - Operations on Arrays
- Multidimensional Arrays
- Sparse Matrices

Introduction

- An array is a finite ordered and collection of homogeneous data elements.
- Array is finite because it contains only limited number of element; and ordered, as all the elements are stored one by one in contiguous locations of computer memory in a linear ordered fashion.
- All the elements of an array are of the same data type (say, integer) only and hence it is termed as collection of homogeneous elements.

Contd...

- Array is particularly useful when we are dealing with lot of variables of the same type.
- For example, lets say I need to store the marks in math subject of 100 students.
- To solve this particular problem, either I have to create the 100 variables of int type or create an array of int type with the size 100.
- Obviously the second option is best, because keeping track of all the 100 different variables is a tedious task.
- On the other hand, dealing with array is simple and easy, all 100 values can be stored in the same array at different indexes (0 to 99).

Arrays

- An array is a collection of similar data elements. These data elements have the same data type.
- An array is conceptually defined as a collection of *<index, value> pairs*.
- Each array element is referred to by specifying the array name followed by one or more subscripts, with each subscript enclosed in square brackets.
- Each subscript must be expressed as a **nonnegative integer**.
- Thus, in the n-element array x, the array elements are x[0], x[1], x[2], ..., x[n-1].
- The number of subscripts determines the dimensionality of the array. (for example x[i] refers to an element in one-dimensional array x, and y[i][j] to an element in 2-D array y)

Declaration of Arrays

- Arrays are declared using the following syntax:

`data_type name_of_array[size_of_array]`

`float arr[10]`

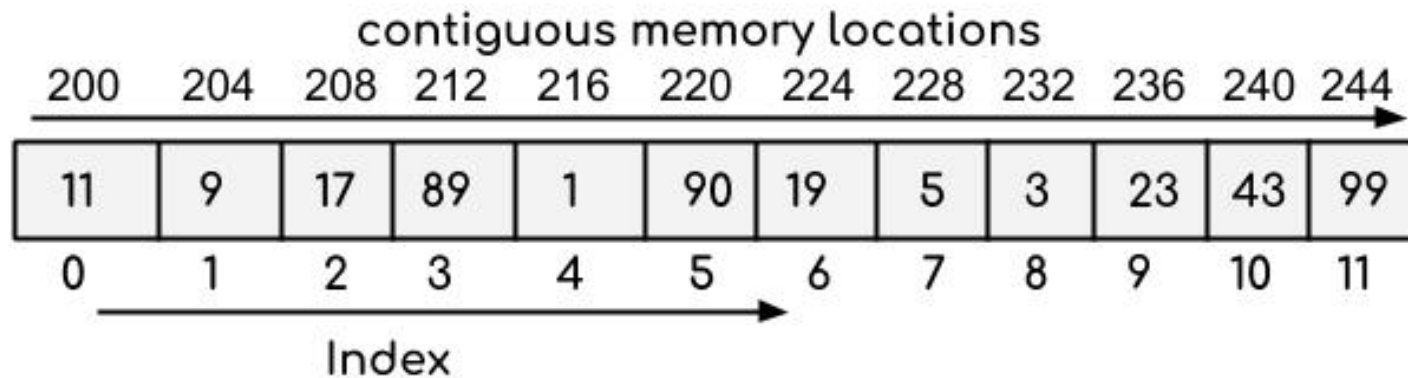
`int mark[20]`

- Sample code to read input through array

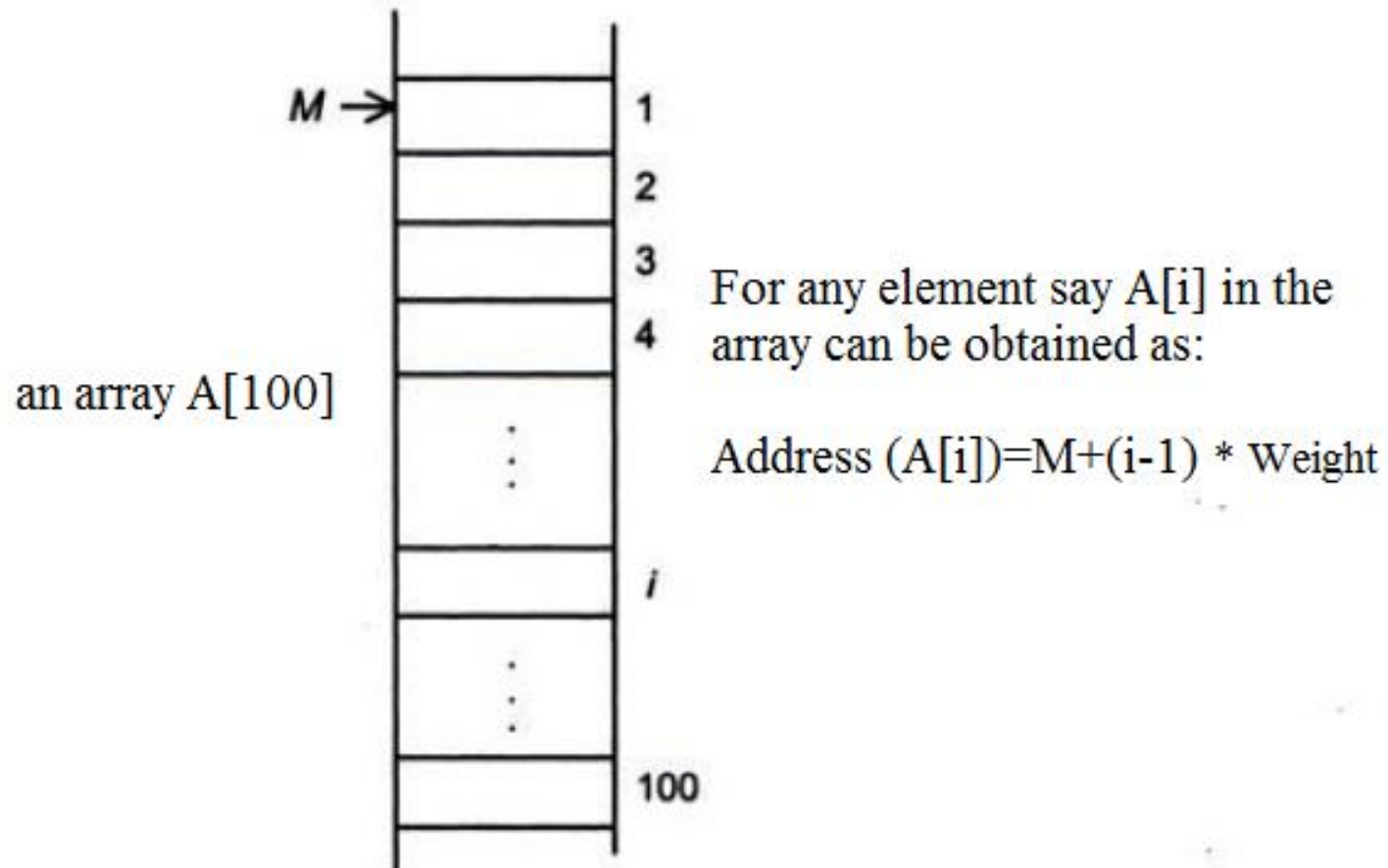
```
int j, mark[20];  
for (j=0; j<20; j++)  
scanf ("%d", &mark[j]);
```

Array Memory Representation

- The following diagram represents an integer array that has 12 elements.
- The index of the array starts with 0, so the array having 12 elements has indexes from 0 to 11.



Memory Allocation of an Array

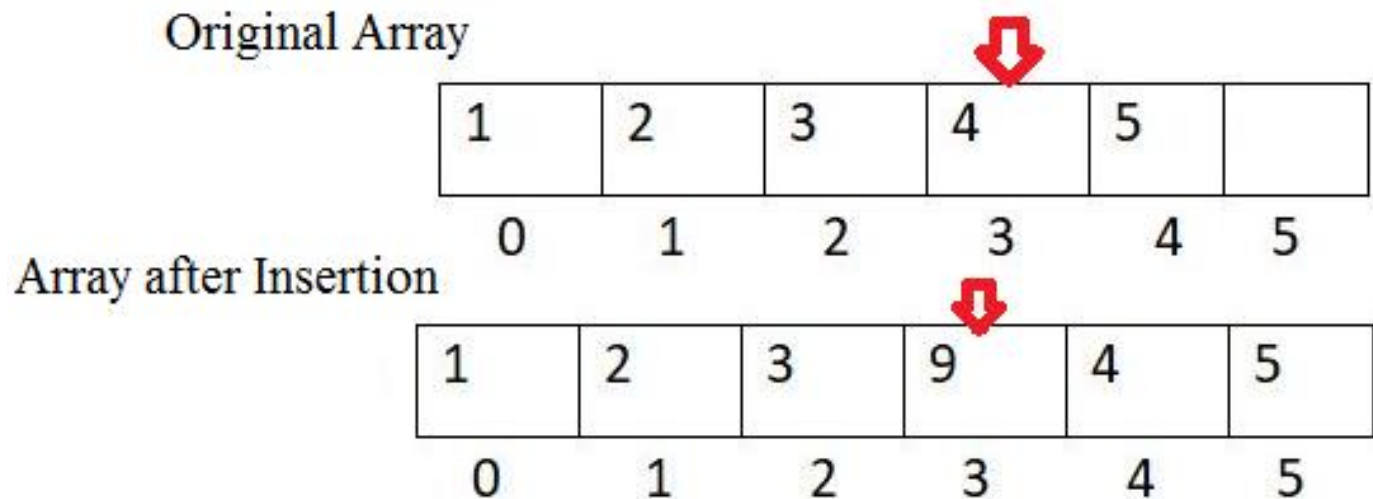


Operations on an Array

- **Retrieval of an element:** Given an array index, retrieve the corresponding value.
 - This can be accomplished in $O(1)$ time. This is an important advantage of the array relative to other structures.
- **Search:** Given an element value, determine whether it is present in the array.
 - If the array is unsorted, there is no good alternative to a linear search that iterates through all of the elements in the array and stops when the desired element is found.
 - In the worst case, this requires $O(n)$ time.
 - If the array is sorted, Binary search can be used. The Binary search only requires $O(\log n)$ time.

Contd...


- **Insertion at required position:** for inserting the element at required position, all of the existing elements in the array from the desired location must be shifted one place to the right.
 - This requires $O(n)$ time in worst case.



Contd...


- **Deletion at desired position:** leaves a “vacant” location.
 - Actually, this location can never be vacant because it refers to a word in memory which must contain some value.
 - Thus, if the program accesses a “vacant” location, it doesn’t have any way to know that the location is vacant.
 - This requires $O(n)$ time in worst case.

Original Array



1	2	3	9	4	5
0	1	2	3	4	5

Array after Deletion



1	2	3	4	5	
0	1	2	3	4	5

Sorted Arrays

- There are several benefits to using sorted arrays, namely: searching is faster, computing order statistics (the i^{th} *smallest element*) is $O(1)$, etc.
- The sorting or sorted Arrays may be considered as of pre-processing steps on data to make subsequent queries efficient.
- *The* idea is that we are often willing to invest some time at the beginning in setting up a data structure so that subsequent operations on it become faster.

Contd...

- Some sorting algorithms such as Heap sort and Merge sort require $O(n \log n)$ time in the worst case
- Whereas other simpler sorting algorithms such as insertion sort, bubble sort and selection sort require $O(n^2)$ time in the worst case.
- Quick sort have a worst case time of $O(n^2)$, but require $O(n \log n)$ on the average.

Polynomial Representation

- The general form of a polynomial is:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_2 x^2 + a_1 x + a_0$$

Where:

- $P(x)$ is the polynomial function.
- x is the variable.
- $a_n, a_{n-1}, \dots, a_2, a_1, a_0$ are coefficients (constants).
- n is a non-negative integer and represents the highest degree of the polynomial.

<div>-4x⁰ + 7x¹ + 6x²</div>			
	0	1	2
Poly	-4	7	6
int poly [3];			

Addition

```
function add_polynomials(poly1, degree1, poly2, degree2):  
    max_degree = max(degree1, degree2)  
    result = array of size (max_degree + 1)  
  
    for i from 0 to max_degree:  
        result[i] = 0  
        if i <= degree1:  
            result[i] += poly1[i]  
        if i <= degree2:  
            result[i] += poly2[i]  
  
    return result
```

Multiplication

```
function multiply_polynomials(poly1, degree1, poly2, degree2):  
    result = array of size (degree1 + degree2 + 1) filled with zeros  
  
    for i from 0 to degree1:  
        for j from 0 to degree2:  
            result[i + j] += poly1[i] * poly2[j]  
  
    return result
```


Another way Polynomial Representation

The array representation of the polynomial is give below:

2	2	0	5	1	1	1	0	2
---	---	---	---	---	---	---	---	---

Index 0 stores the coefficient of the first term, index 1 stores the exponent of the variable x and index 2 stores the exponent of the variable y in the first term. The process is repeated for the remaining terms in the polynomial.

Contd...

It can also be represented as a 2-dimensional array as follows:

	y^0	y^1	y^2
x^0	0	0	1
x^1	0	5	0
x^2	2	0	0

Pseudocode for Adding Two Polynomials Using an Array of Structures

```
AddPolynomials(poly1, n1, poly2, n2, result, n3)
```

```
{
    i ← 0 // Pointer for poly1
    j ← 0 // Pointer for poly2
    k ← 0 // Pointer for result

    While (i < n1) AND (j < n2)
    {
        If poly1[i].exp == poly2[j].exp
            result[k].coeff = poly1[i].coeff + poly2[j].coeff
            result[k].exp = poly1[i].exp
            i ← i + 1
            j ← j + 1
        Else If poly1[i].exp > poly2[j].exp
            result[k].coeff = poly1[i].coeff
            result[k].exp = poly1[i].exp
            i ← i + 1
        Else:
            result[k].coeff = poly2[j].coeff
            result[k].exp = poly2[j].exp
            j ← j + 1
        End If
        k ← k + 1
    }
}
```

```
struct Term
{
    int coeff; // Coefficient of the term
    int exp;   // Exponent of the term
}Poly;
Poly poly1[n1], poly2[n2], result[n3];
```

Contd...

```
// Copy remaining terms from poly1, if any
While (i < n1)
{
    result[k].coeff = poly1[i].coeff
    result[k].exp = poly1[i].exp
    i = i + 1
    k = k + 1
}
// Copy remaining terms from poly2, if any
While (j < n2)
{
    result[k].coeff = poly2[j].coeff
    result[k].exp = poly2[j].exp
    j = j + 1
    k = k + 1
}
n3 = k // Total terms in result
Return n3
}
```

Advantages and disadvantages of Arrays

Advantages

- Reading an array element is simple and efficient. The time complexity is $O(1)$ in both best and worst cases.
- This is because any element can be instantly read using indexes (base address calculation behind the scene) without traversing the whole array.

Disadvantages

- While using array, we must need to make the decision of the size of the array in the beginning, so if we are not aware how many elements we are going to store in array, it would make the task difficult.
- The size of the array is fixed so if at later point, if we need to store more elements in it then it can't be done. On the other hand, if we store less number of elements than the declared size, the remaining allocated memory is wasted.

Multidimensional Arrays

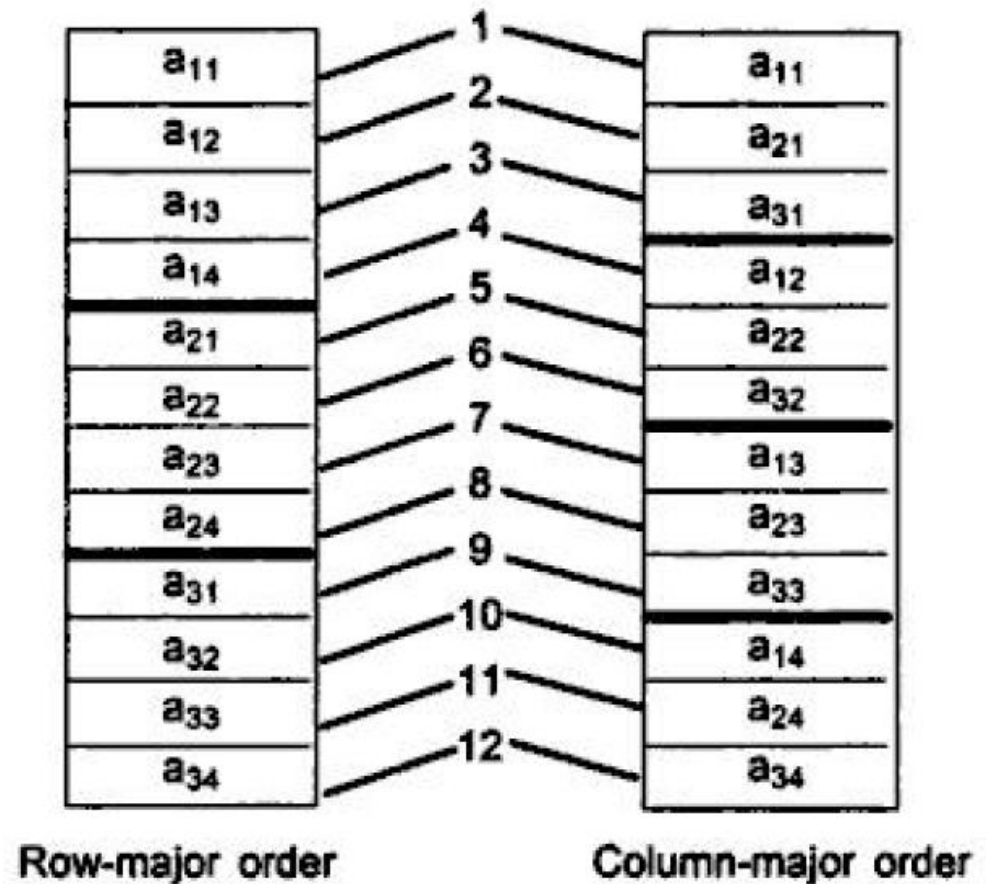
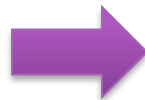
	Column 0	Column 1	Column 2
Row 0	<code>x[0][0]</code>	<code>x[0][1]</code>	<code>x[0][2]</code>
Row 1	<code>x[1][0]</code>	<code>x[1][1]</code>	<code>x[1][2]</code>
Row 2	<code>x[2][0]</code>	<code>x[2][1]</code>	<code>x[2][2]</code>

Row- or Column- Major Representation

- Earlier representations of multidimensional arrays mapped the location of each element of the multidimensional array into a **location of a one-dimensional array**.
- Consider a two dimensional array with r rows and c columns. The number of elements in the array $n = rc$.
- The element in location $[i][j]$, $0 \leq i < r$ and $0 \leq j < c$, will be mapped onto an integer in the range $[0, n-1]$.
- If this is done in **row-major order** — the elements of row 0 are listed in order from left to right followed by the elements of row 1, then row 2, etc. — the mapping function is **$ic + j$** .
- If elements are listed in **column-major order**, the mapping function is **$jr + i$** .

Contd...

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix}_{3 \times 4}$$



Memory Allocation in 2D-arrays (A mxn)

- Logically, a matrix appears as two-dimensional but physically it is stored in a linear fashion.
- In order to map from logical view to physical structure, we need indexing formula.

Row-major order. Assume that the base address is the first location of the memory, that is,
1. So, the address of a_{ij} will be obtained as

$$\begin{aligned}\text{Address } (a_{ij}) &= \text{Storing all the elements in first } (i - 1)\text{-th rows} \\ &\quad + \text{The number of elements in } i\text{-th row up to } j\text{-th column.} \\ &= (i - 1) \times n + j\end{aligned}$$

So, for the matrix $A_{3 \times 4}$, the location of a_{32} will be calculated as 10. Instead of considering the base address to be 1, if it is at M , then the above formula can be easily modified as:

$$\text{Address } (a_{ij}) = M + (i - 1) \times n + j - 1$$

Contd...

Column-major order. Let us first consider the starting location of matrix is at memory location 1. Then

$$\begin{aligned}\text{Address } (a_{ij}) &= \text{Storing all the elements in first } (j - 1)\text{-th columns} \\ &\quad + \text{The number of elements in } j\text{-th column up to } i\text{-th rows.} \\ &= (j - 1) \times m + i\end{aligned}$$

And considering the base address at M instead of 1, the above formula will stand as

$$\text{Address } (a_{ij}) = M + (j - 1) \times m + i - 1$$

Sparse Matrices

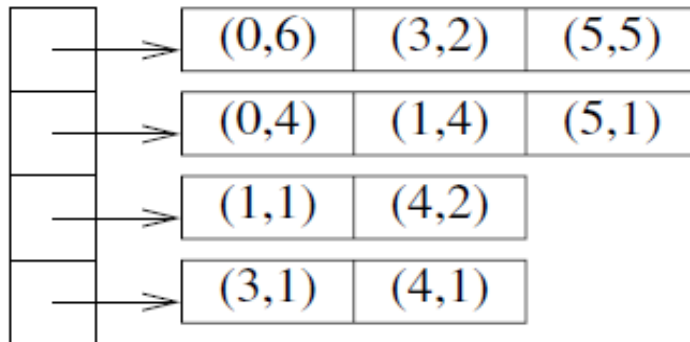
- A matrix is sparse if a large number of its elements are zeros.
- Storing of null elements of a sparse matrix is nothing but wastage of memory
- Rather than store such a matrix as a two-dimensional array with lots of zeroes, a common strategy is to save space by explicitly storing only the non-zero elements.
- There are several ways to represent this matrix as a one-dimensional array.

Contd...

- Other sparse matrices may have an irregular or unstructured pattern.

$$\begin{bmatrix} 6 & 0 & 0 & 2 & 0 & 5 \\ 4 & 4 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

(a)



(c)

row	0	0	0	1	1	1	2	2	3	3
col	0	3	5	0	1	5	1	4	3	4
val	6	2	5	4	4	1	1	2	1	1

(b)

FIGURE: Unstructured matrices.

Contd...

- Some well known sparse matrices which are symmetric in form:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 & 0 \\ 4 & 1 & 2 & 0 & 0 \\ 3 & 3 & 2 & 1 & 0 \\ 2 & 4 & 1 & 3 & 3 \end{bmatrix}$$

Lower Triangular Matrix

(a)

$$\begin{bmatrix} 1 & 3 & 1 & 5 & 8 \\ 0 & 1 & 3 & 2 & 4 \\ 0 & 0 & 2 & 3 & 4 \\ 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 4 \end{bmatrix}$$

Upper Triangular Matrix

(b)

$$\begin{bmatrix} 2 & 1 & 0 & 0 & 0 \\ 1 & 3 & 4 & 0 & 0 \\ 0 & 1 & 1 & 2 & 0 \\ 0 & 0 & 4 & 7 & 4 \\ 0 & 0 & 0 & 3 & 5 \end{bmatrix}$$

Tridiagonal Matrix

(c)

Memory Representation

Memory representation of lower-triangular matrix

Consider the following lower-triangular matrix:

$$\begin{bmatrix} a_{11} & & & & \\ a_{21} & a_{22} & & & \\ a_{31} & a_{32} & a_{33} & & \\ \vdots & \vdots & \vdots & & \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix}_{n \times n}$$

Contd...

Row-major order. According to row-major order, the address of any element a_{ij} , $1 \leq i$, $j \leq n$ can be obtained as

$$\begin{aligned}\text{Address } (a_{ij}) &= \text{Number of elements up to } a_{ij} \text{ element} \\ &= \text{Total number of elements in first } i - 1 \text{ rows} \\ &\quad + \text{Number of elements up to } j\text{-th column in the } i\text{-th row} \\ &= 1 + 2 + 3 + \dots + (i - 1) + j \\ &= \frac{i(i - 1)}{2} + j\end{aligned}$$

If the starting location of the first element, that is, of a_{11} is M , then the address of a_{ij} , $1 \leq i$, $j \leq n$ will be

$$\text{Address } (a_{ij}) = M + \frac{i(i - 1)}{2} + j - 1$$

Contd...

Column-major order. According to column-major order, the address of any element a_{ij} , $1 \leq i, j \leq n$ can be obtained as

$$\begin{aligned}\text{Address } (a_{ij}) &= \text{Number of elements up to } a_{ij} \text{ element} \\ &= \text{Total number of elements in first } j-1 \text{ columns} \\ &\quad + \text{Number of elements up to } i\text{-th row in the } j\text{-th column} \\ &= [n + (n-1) + (n-2) + \dots + (n-j+2)] + (i-j+1) \\ &= \{n \times (j-1) - [1 + 2 + 3 + \dots + (j-2)] + (j-1)\} + i \\ &= n \times (j-1) - \frac{j(j-1)}{2} + i \\ &= (j-1) \times \left(n - \frac{j}{2}\right) + i\end{aligned}$$

If the starting location of the first element (that is, of a_{11}) is M then the address of a_{ij} , $1 \leq i, j \leq n$ will be

$$\text{Address } (a_{ij}) = M + (j-1) \times \left(n - \frac{j}{2}\right) + i - 1$$

Contd...

Memory representation of upper-triangular matrix

As an another example, consider the following form of a upper-triangular matrix:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ & a_{22} & a_{23} & \dots & a_{2n} \\ & & a_{33} & \dots & a_{3n} \\ & & & \ddots & \\ & & & & a_{nn} \end{bmatrix}_{n \times n}$$

Contd...

Row-major order. According to row-major order, the address of any element a_{ij} , $1 \leq i, j \leq n$ can be obtained as

$$\begin{aligned}\text{Address } (a_{ij}) &= \text{Number of elements up to } a_{ij} \text{ element} \\ &= \text{Total number of elements in first } (i - 1) \text{ rows} \\ &\quad + \text{Number of elements up to } j\text{-th column in the } i\text{-th row} \\ &= n + (n - 1) + (n - 2) + \cdots + (n - i + 2) + (j - i + 1) \\ &= n \times (i - 1) - [1 + 2 + 3 + \cdots + (i - 2) + (i - 1)] + j \\ &= n \times (i - 1) - \frac{i(i - 1)}{2} + i \\ &= (i - 1) \times \left(n - \frac{i}{2} \right) + j\end{aligned}$$

If the starting location of the first element, i.e. of a_{11} is M , then the address of a_{ij} , $1 \leq i, j \leq n$ will be

$$\text{Address } (a_{ij}) = M + (i - 1) \times \left(n - \frac{i}{2} \right) + j - 1$$

Contd...

Column-major order. According to column-major order, the address of any element a_{ij} , $1 \leq i, j \leq n$ can be obtained as

$$\begin{aligned}\text{Address } (a_{ij}) &= \text{Number of elements up to } a_{ij} \text{ element} \\ &= \text{Total number of elements in first } (j - 1) \text{ columns} \\ &\quad + \text{Number of elements up to } i\text{-th row in the } j\text{-th column} \\ &= [1 + 2 + 3 + \dots + (j - 1)] + i \\ &= \frac{j(j - 1)}{2} + i\end{aligned}$$

If the starting location of the first element, i.e. of a_{11} is M , then the address of a_{ij} , $1 \leq i, j \leq n$ will be

$$\text{Address } (a_{ij}) = M + \frac{j(j - 1)}{2} + i - 1$$

Contd...

Memory representation of tridiagonal matrix

Let us consider the following tridiagonal matrix:

$$\begin{bmatrix} a_{11} & a_{12} & & & & \\ a_{21} & a_{22} & a_{23} & & & \\ & a_{32} & a_{33} & a_{34} & & \\ & & a_{43} & a_{44} & a_{45} & \\ & & & \vdots & & \\ & & & & \vdots & \\ & & & & & a_{(n-1)(n-2)} & a_{(n-1)(n-1)} & a_{(n-1)n} \\ & & & & & & a_{n(n-1)} & a_{nn} \end{bmatrix}$$

Contd...

Row-major order. According to row-major order, the address of any element a_{ij} , $1 \leq i, j \leq n$ can be obtained as

$$\begin{aligned}\text{Address } (a_{ij}) &= \text{Number of elements up to } a_{ij} \text{ element} \\ &= \text{Total number of elements in first } (i - 1) \text{ rows} \\ &\quad + \text{Number of elements up to } j\text{-th column in the } i\text{-th row} \\ &= \{2 + [3 + 3 + \dots + \text{up to } (i - 2) \text{ terms}]\} + (j - i + 2) \\ &= 2 + (i - 2) \times 3 + j - (i - 2) \\ &= 2 + 2 \times (i - 2) + j\end{aligned}$$

If the starting location of the first element, i.e. of a_{11} is M then the address of a_{ij} , $1 \leq i, j \leq n$ will be

$$\text{Address } (a_{ij}) = M + 2 \times (i - 2) + j + 1$$

Contd...

Column-major order. According to column-major order, the address of any element a_{ij} , $1 \leq i, j \leq n$ can be obtained as

$$\begin{aligned}\text{Address } (a_{ij}) &= \text{Number of elements up to } a_{ij} \text{ element} \\ &= \text{Total number of elements in first } (j - 1) \text{ columns} \\ &\quad + \text{Number of elements up to } i\text{-th row in the } j\text{-th column} \\ &= \{2 + [3 + 3 + \dots + \text{up to } (j - 2) \text{ terms}]\} + (i - j + 2) \\ &= 2 + (j - 2) \times 3 + i - (j - 2) \\ &= 2 + 2 \times (j - 2) + i\end{aligned}$$

If the starting location of the first element, i.e. of a_{11} is M then the address of a_{ij} , $1 \leq i, j \leq n$ will be

$$\text{Address } (a_{ij}) = M + 2 \times (j - 2) + i + 1$$

Three Dimensional (3-D) Array

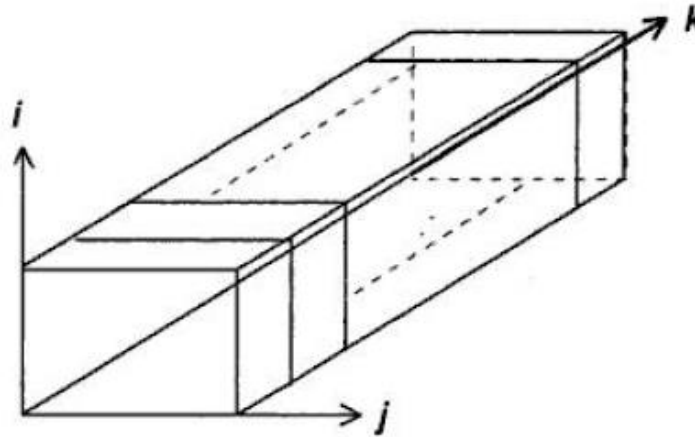


Fig. 2.12 A three-dimensional array.

Number of rows = x (number of elements in a column)

Number of columns = y (number of elements in a row) and

Number of pages = z

$$\begin{aligned}\text{Address } (a_{ijk}) &= \text{Number of elements in first } (k - 1) \text{ pages} \\ &\quad + \text{Number of elements in } k\text{-th page up to } (i - 1) \text{ rows} \\ &\quad + \text{Number of elements in } k\text{-th page, in } i\text{-th row up to } j\text{-th column} \\ &= xy(k - 1) + (i - 1)y + j\end{aligned}$$

Assignment-1

- An unsorted array of size n contains distinct integers between 1 and $n+1$ with one element missing. Give an $O(n)$ algorithm to find the missing integer without using any extra space.
 - Algorithm:
 1. Sort the array in $O(n \log n)$ time.
 2. Traverse the array to find the first gap where $arr[i] \neq i + 1$.
 - Time Complexity: $O(n \log n)$ due to sorting.

Assignment-1

- An unsorted array of size n contains distinct integers between 1 and $n+1$ with one element missing. Give an $O(n)$ algorithm to find the missing integer without using any extra space.

Algorithm:

1. Compute the expected sum of integers from 1 to $n + 1$ using the formula:

$$\text{Expected Sum} = \frac{(n + 1)(n + 2)}{2}$$

2. Compute the actual sum of the array elements.
3. The missing number is the difference between the expected sum and the actual sum.

Assignment-2

- Print Trinomial Triangle

```
      1
    1 1 1
  1 2 3 2 1
1 3 6 7 6 3 1
1 4 10 16 19 16 10 4 1
```

Contd...

```
PrintTrinomialTriangle(n)
{
    Initialize T as a 2D array with size [n][2n-1] filled with 0
    T[0][n-1] = 1 // Set the middle element of the first row to 1
    // Step 2: Generate the trinomial triangle
    For i = 1 to n-1
        For j = 0 to 2n-2
            T[i][j] ← GetValue(T, i-1, j-1) + GetValue(T, i-1, j) + GetValue(T, i-1, j+1)
        End For
    End For
    // Step 3: Print the triangle
    For i = 0 to n-1
        For j = 0 to 2n-2
            If T[i][j] == 0
                Print T[i][j]= " "
            Print T[i][j]
        End For    Print new line
    End For
}

Function GetValue(T, i, j)
{
    If i < 0 OR j < 0 OR j >= 2n-1
        Return 0
    Return T[i][j]
}
```

Assignment-3

- Given an unordered array X of n integers, create a new array M of n elements, where each element $M[i]$ represents the product of all elements in X except $X[i]$. Division operations are not allowed, but you may use additional memory.

Contd...

```
productExceptSelf(int arr[], int n, int prod[])
{
    for (int i = 0; i < n; i++)
        prod[i] = 1;

    // Compute product of all elements except arr[i]
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i != j) {
                prod[i] *= arr[j];
            }
        }
    }
}
```

Complexity: $O(n^2)$

Using Prefix and Suffix Array – $O(n)$ Time and $O(n)$ Space

Let us understand this with an example [2, 3, 1, 4]

1. Create two arrays `left[]` and `right[]` of size `n` each
2. Create a product array `prod[]` to store result
3. The left array is going to store product of prefix elements such that `left[i]` does not have `arr[i]` in product. The values of `left` is going to be [1, 2, 6, 6]
4. Similarly compute `right[]` to store suffix products except self. `right[] = [12, 4, 4, 1]`
5. Now compute product array. `prod[i] = left[i] * right[i]`. `prod[] = [12, 8, 24, 6]`

Contd...

```
void productExceptSelf(int arr[], int n, int prod[])
{ // If only one element, return an array with 1
  if (n == 1) {
    prod[0] = 1;
    return;
  }

  int left[n], right[n];

  // Construct the left array
  left[0] = 1;
  for (int i = 1; i < n; i++)
    left[i] = arr[i - 1] * left[i - 1];

  // Construct the right array
  right[n - 1] = 1;
  for (int j = n - 2; j >= 0; j--)
    right[j] = arr[j + 1] * right[j + 1];

  // Construct the product array using left[] and right[]
  for (int i = 0; i < n; i++)
    prod[i] = left[i] * right[i];
}
```

Time Complexity: $O(n)$.

The array needs to be traversed three times, so the time complexity is $O(n)$.

Auxiliary Space: $O(n)$. Two extra arrays

Reference Book

- Classic Data Structures by D. Samanta

Thank You