Data Structures

UCS301

Array

Department of CSE
Thapar Institute of Engineering and Technology, Patiala

Array ADT

`float marks[10];`

- The simplest but useful data structure.
- Assign single name to a homogeneous collection of instances of one abstract data type.
  - All array elements are of same type, so that a pre-defined equal amount of memory is allocated to each one of them.
- Individual elements in the collection have an associated index value that depends on array dimension.

Contd…

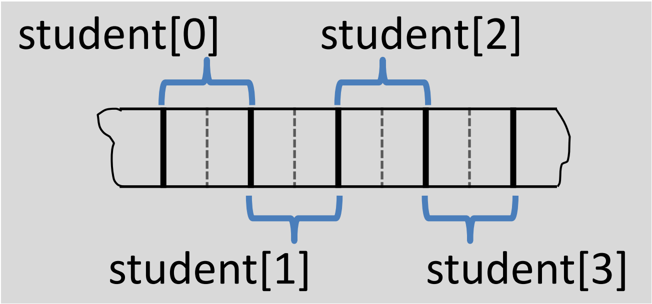
- One-dimensional and two-dimensional arrays are commonly used.
- Multi–dimensional arrays can also be defined.

- Usage:
  – Used frequently to store relatively permanent collections of data.
  – Not suitable if the size of the structure or the data in the structure are constantly changing.

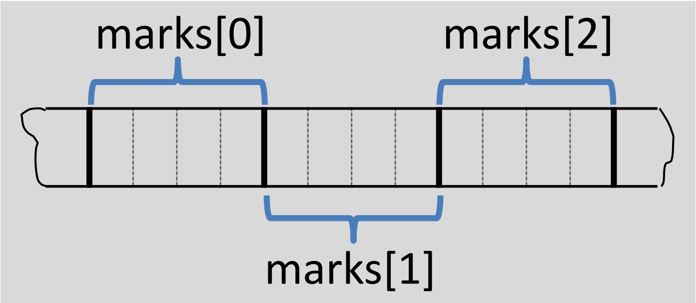
Memory Storage

Memory Storage – One Dimensional Array

```
int student[4];
```




```
float marks[3];
```



Memory Storage – Two Dimensional Array

```
int marks[3][5];
```

- Can be visualized in the form of a matrix as

	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	marks[0][0]	marks[0][1]	marks[0][2]	marks[0][3]	marks[0][4]
Row 1	marks[1][0]	marks[1][1]	marks[1][2]	marks[1][3]	marks[1][4]
Row 2	marks[2][0]	marks[2][1]	marks[2][2]	marks[2][3]	marks[2][4]

Contd…

- Row-major order

$$\underbrace{\begin{array}{|c|c|c|c|c|}\hline (0,0) & (0,1) & (0,2) & (0,3) & (0,4) \\ \hline \end{array}}_{\text{Row0}}\underbrace{\begin{array}{|c|c|c|c|c|}\hline (1,0) & (1,1) & (1,2) & (1,3) & (1,4) \\ \hline \end{array}}_{\text{Row1}}\underbrace{\begin{array}{|c|c|c|c|c|}\hline (2,0) & (2,1) & (2,2) & (2,3) & (2,4) \\ \hline \end{array}}_{\text{Row2}}$$

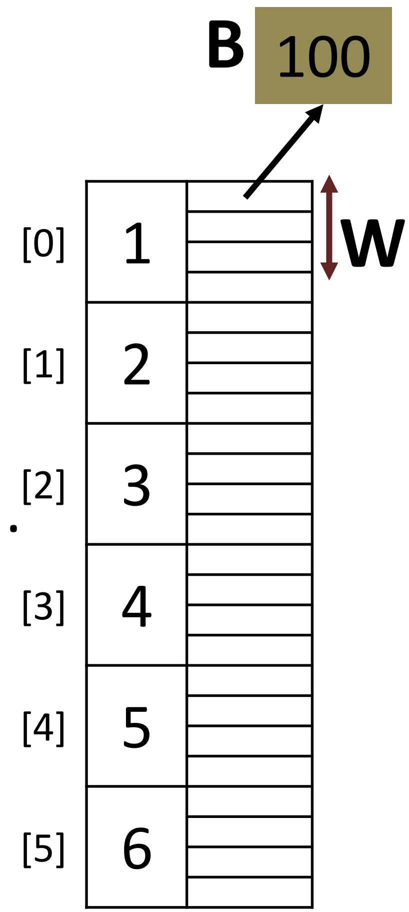
- Column-major order

$$\underbrace{\begin{array}{|c|c|c|}\hline (0,0) & (1,0) & (2,0) \\ \hline \end{array}}_{\text{Col0}}\underbrace{\begin{array}{|c|c|c|}\hline (0,1) & (1,1) & (2,1) \\ \hline \end{array}}_{\text{Col1}}\underbrace{\begin{array}{|c|c|c|}\hline (0,2) & (1,2) & (2,2) \\ \hline \end{array}}_{\text{Col2}}\underbrace{\begin{array}{|c|c|c|}\hline (0,3) & (1,3) & (2,3) \\ \hline \end{array}}_{\text{Col3}}\underbrace{\begin{array}{|c|c|c|}\hline (0,4) & (1,4) & (2,4) \\ \hline \end{array}}_{\text{Col4}}$$

Array Address Computation

1D array – address calculation

- Let A be a one dimensional array.
- Formula to compute the address of the I^{th} element of an array (A[I]) is:



Address of A[I] = B + W * (I – LB)

where,

B = Base address/address of first element, i.e. A[LB].

W = Number of bytes used to store a single array element.

I = Subscript of element whose address is to be found.

LB = Lower limit / Lower Bound of subscript, if not specified assume 0 (zero).

1D array – address calculation

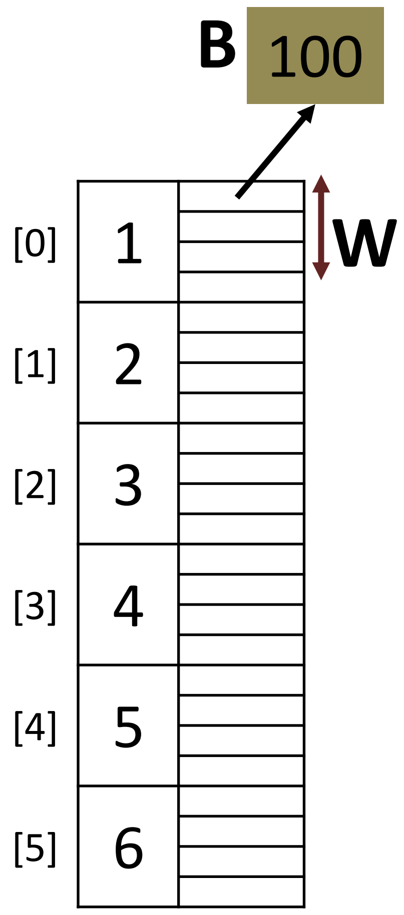
- Let A be a one dimensional array.
- Formula to compute the address of the I^{th} element of an array (A[I]) is:

Address of A[I] = B + W * (I – LB)

Given:

B = 100, W = 4, and LB = 0

A[0] = 100 + 4 * (0 – 0) = 100



1D array – address calculation

- Let A be a one dimensional array.
- Formula to compute the address of the I^{th} element of an array (A[I]) is:

Address of A[I] = B + W * (I – LB)

Given:

B = 100, W = 4, and LB = 0

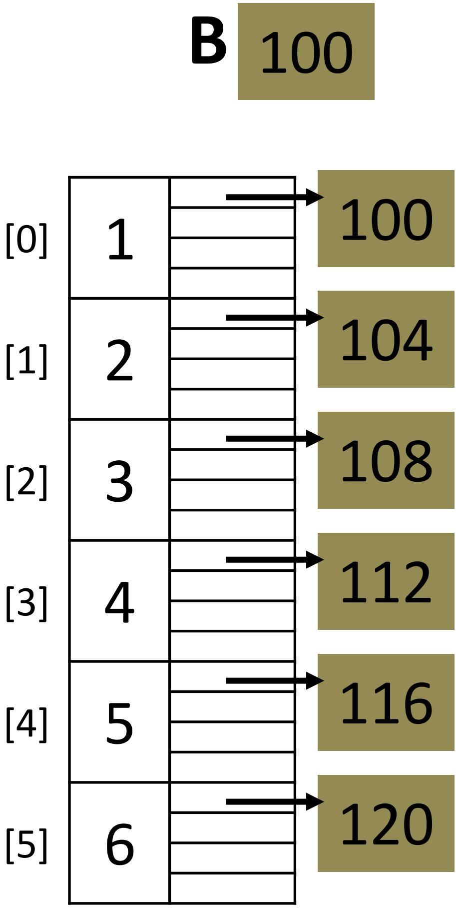
A[1] = 100 + 4 * (1 – 0) = 104

A[2] = 100 + 4 * (2 – 0) = 108

A[3] = 100 + 4 * (3 – 0) = 112

A[4] = 100 + 4 * (4 – 0) = 116

A[5] = 100 + 4 * (5 – 0) = 120



Example – 1

- Similarly, for a character array where a single character uses 1 byte of storage.
- If the base address is 1200 then,

Address of A[I] = B + W * (I – LB)

Address of A[0] = 1200 + 1 * (0 – 0) = 1200

Address of A[1] = 1200 + 1 * (1 – 0) = 1201

…

Address of A[10] = 1200 + 1 * (10 – 0) = 1210

Example – 2

- If **LB** = 5, **Loc(A[LB])** = 1200, and **W** = 4.
- Find **Loc(A[8])**.

Address of A[I] = B + W * (I – LB)

$$\begin{aligned} Loc(A[8]) &= Loc(A[5]) + 4 * (8 - 5) \\ &= 1200 + 4 * 3 \\ &= 1200 + 12 \\ &= 1212 \end{aligned}$$

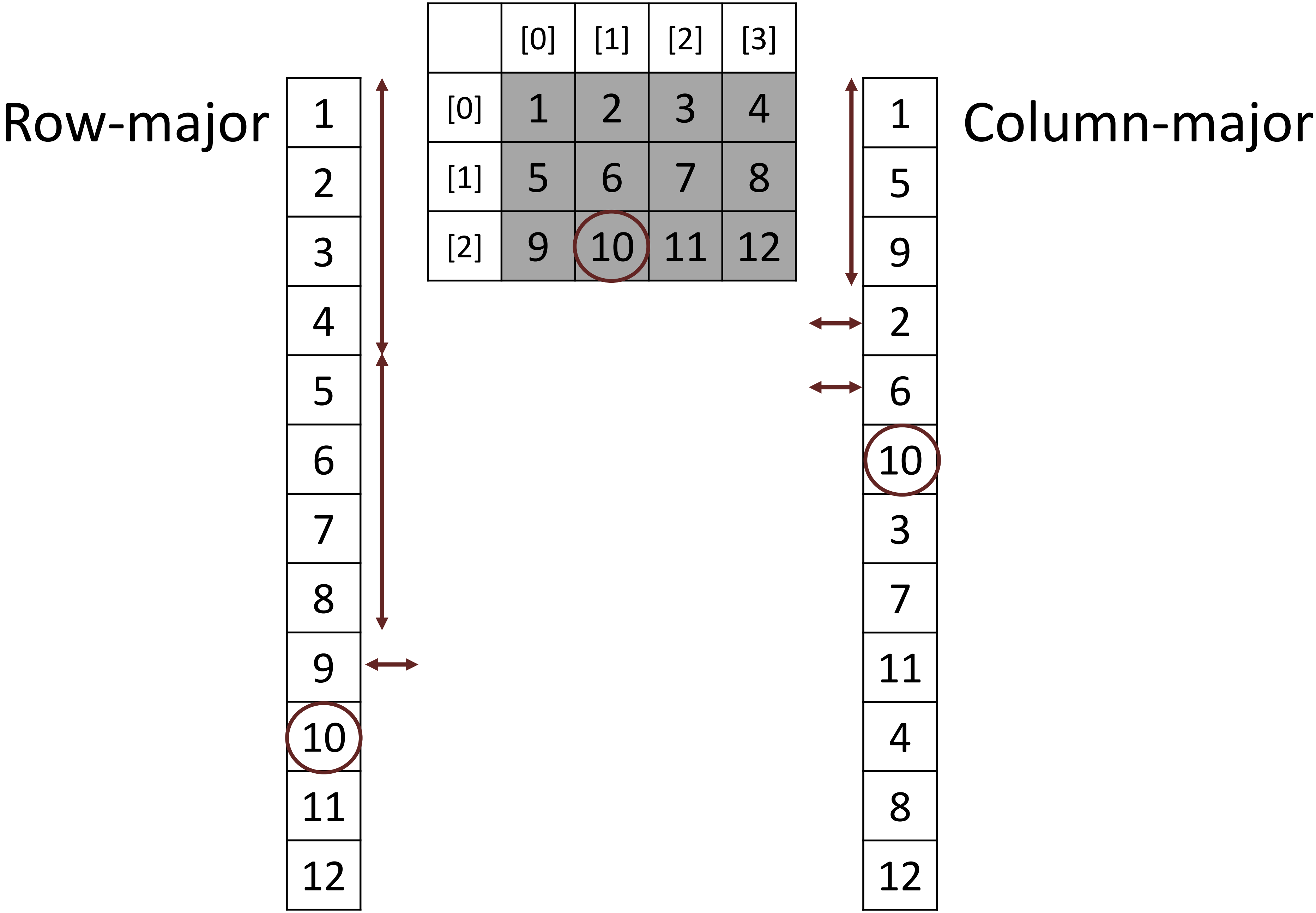
Example – 3

- Base address of an array **B[1300…..1900]** is **1020** and size of each element is 2 bytes in the memory. Find the address of **B[1700]**.

Address of A[I] = B + W * (I – LB)

- Given: **B** = 1020, **W** = 2, **I** = 1700, **LB** = 1300

Address of B[1700] = 1020 + 2 * (1700 – 1300)
= 1020 + 2 * 400
= 1020 + 800
= 1820

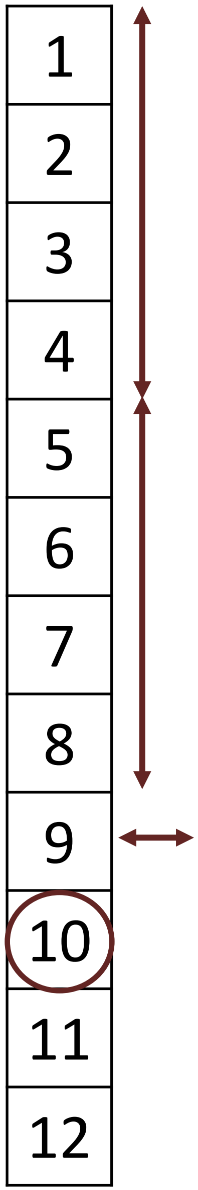

Row-major
Column-major
[0]
[1]
[2]
[3]
1 2 3 4
5 6 7 8
9 10 11 12
1 2 3 4 5 6 7 8 9 10 11 12
1 5 9 2 6 10 3 7 11 4 8 12

2D Array – Address Calculation

- If **A** be a two dimensional array with **M** rows and **N** columns. We can compute the address of an element at **I**th row and **J**th column of an array (**A[I][J]**).

  B = Base address/address of first element, i.e. A[LBR][LBC]
  I = Row subscript of element whose address is to be found
  J = Column subscript of element whose address is to be found
  W = Number of bytes used to store a single array element
  LBR = Lower limit of row/start row index of matrix, if not given assume 0
  LBC = Lower limit of column/start column index of matrix, if not given assume 0
  N = Number of column of the given matrix
  M = Number of row of the given matrix

Row-major



	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	10	11	12

M = 3

N = 4

Address of A[2][1] =

B + W * (4 * (2 – 0) + (1 – 0))

Address of A[I][J] =

B + W * (N * (I – LBR) + (J – LBC))

$M = 3$

$N = 4$

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	10	11	12

Column-major

1
5
9
2
6
10
3
7
11
4
8
12

Address of A [2][1] =

$$B + W * ((2 - 0) + 3 * (1 - 0))$$

Address of A [I][J] =

$$B + W * ((I - LBR) + M * (J - LBC))$$

Contd…

- Row Major

Address of A[I][J] = B + W * (N * (I – LBR) + (J – LBC))

- Column Major

Address of A [I][J] = B + W * ((I – LBR) + M * (J – LBC))

- Note: **A[LBR…UBR, LBC…UBC]**

M = (UBR – LBR) + 1

N = (UBC – LBC) + 1

Example – 4

- Suppose elements of array **A[5][5]** occupies **4** bytes, and the address of the first element is **49**. Find the address of the element **A[4][3]** when the storage is row major.

Address of A[I][J] = B + W * (N * (I – LBR) + (J – LBC))

- Given: **B** = 49, **W** = 4, **M** = 5, **N** = 5, **I** = 4, **J** = 3, **LBR** = 0, **LBC** = 0.

Address of A[4][3] = 49 + 4 * (5 * (4 – 0) + (3 - 0))

= 49 + 4 * (23)

= 49 + 92

= 141

Example – 5

- An array **X [-15…10, 15…40]** requires **one** byte of storage. If beginning location is **1500** determine the location of **X [0][20]** in column major.

Address of A[I][J] = B + W * [(I – LBR) + M * (J – LBC)]

- Number or rows (**M**) = **(UBR – LBR) + 1** = [10 – (- 15)] +1 = 26
- Given: **B** = 1500, **W** = 1, **I** = 0, **J** = 20, **LBR** = -15, **LBC** = 15, **M** = 26

Address of X[0][20] = 1500 + 1 * [(0 – (-15)) + 26 * (20 – 15)]

= 1500 + 1 * [15 + 26 * 5]

= 1500 + 1 * [145]

= 1645

Example – 6

- A two-dimensional array defined as **A [-4 … 6] [-2 … 12]** requires **2 bytes** of storage for each element. If the array is stored in row major order form with the address **A[4][8]** as **4142**. Compute the address of **A[0][0].**

  Address of A[I][J] = B + W (N (I – LBR) + (J – LBC))

- **Given:**

  W = 2, LBR = –4, LBC = –2

  #rows = M = 6 + 4 + 1 = 11 #columns = N = 12 + 2 + 1 = 15

  Address of A[4][8] = 4142

- **Address of A[4][8] = B + 2 (15 (4 – (–4)) + (8 – (–2)))**

  4142 = B + 2 (15 (4 + 4) + (8 + 2)) = B + 2 (15 (8) + 10) = B + 2 (120 + 10)

  4142 = B + 260

  Thus, B = 4142 – 260 = 3882

- **Now, Address of A[0][0] = 3882 + 2 (15 (0 – (–4)) + (0 – (–2)))**

  = 3882 + 2 (15(4) + 2) = 3882 + 2 (62)

  = 3882 + 124

  = 4006

Array Basic Operations

Operations on Linear Data Structures

- Traversal
- Search – Linear and Binary.
- Insertion
- Deletion
- Sorting – Different algorithms are there.
- Merging – During the discussion of Merge Sort.

TRAVERSAL

Processing each element in the array.

Example – Print all the array elements.

Algorithm arrayTraverse(A,n)

Input: An array **A** containing **n** integers.

Output: All the elements in **A** get printed.

```
for i = 0 to n-1 do
        Print A[i]
```

```
int arrayTraverse(int arr[], int n)
{
   for (int i = 0; i < n; i++)
          cout << "\n" << arr[i];
}
```

Example – Find minimum element in the array.

Algorithm arrayMinElement(A,n)

Input: An array **A** containing **n** integers.

Output: The minimum element in **A**.

```
min = 0
for i = 1 to n-1 do
    if A[min] > A[i]
        min = i
return A[min]
```

```c
int arrayMinElement(int arr[], int n)
{  int min = 0;
   for (int i = 1; i < n; i++)
   {   if (arr[i] < arr[min])
           min = i;
   }
   return arr[min];
}
```

Search

Find the location of the element with a given value.

Linear Search

- Used if the array is unsorted.
- Example:

Search 7 in the following array

i → 0 → 1 → 2 → 3 → 4 → 5 → 6

a[]										
	10	5	1	6	2	9	7	8	3	4

Found at index 6

Search 11 in the following array

i → 0 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

a[]										
	10	5	1	6	2	9	7	8	3	4

Not found

Contd…

Algorithm linearSearch(A,n,num)

Input: An array **A** containing **n** integers and number **num** to be searched.

Output: Index of **num** if found, otherwise -1.

```
for i = 0 to n-1 do
    if A[i] == num
        return i
return -1
```

```
int linearSearch(int a[], int n, int num)
{  for (int i = 0; i < n; i++)
       if (a[i] == num)
            return i;
    return -1;
}
```

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3	5	7	8	9	12	15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3	5	7	8	9	12	15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3	5	7	8	9	12	15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3	5	7	8	9	12	15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3 5 7 8 9 12 15

Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3 5 7 8 9 12 15

Pseudo Code of Binary Search

Iterative

```
BinarySearch(A[1..n], K)
l←1; r ←n
while l ≤ r do
m←(l + r)/2
if K = A[m] return m
else if K <A[m] r ←m − 1
else l←m + 1
return −1
```

Recursive

```
BinarySearch(A[1..n],l,r K)
while l ≤ r do
m←(l + r)/2
if K = A[m] return m
else if K <A[m]
    BinarySearch(A[1..n],l,m−1 K)
else
    BinarySearch(A[1..n],m+1,r K)
return −1
```

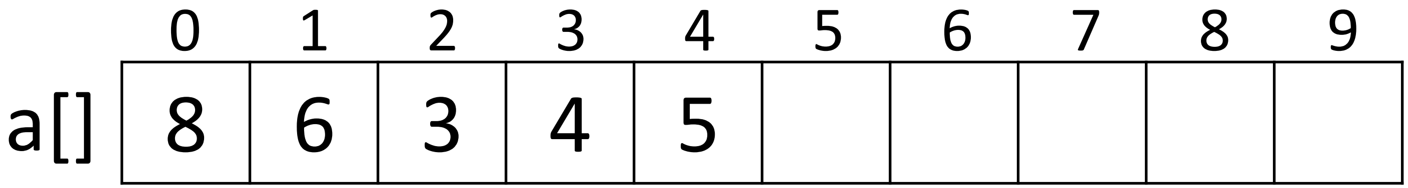
Insertion

Insert an element in the array

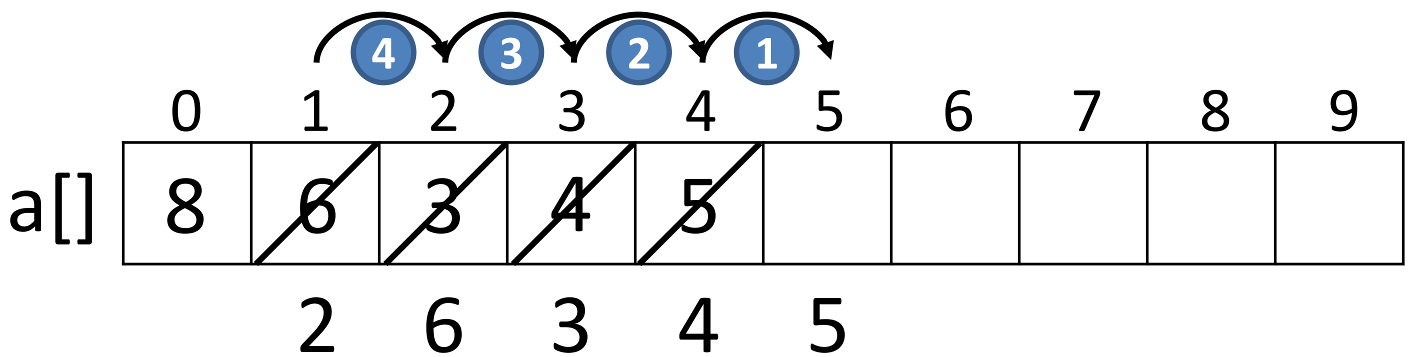
Deletion

Delete an element from the array

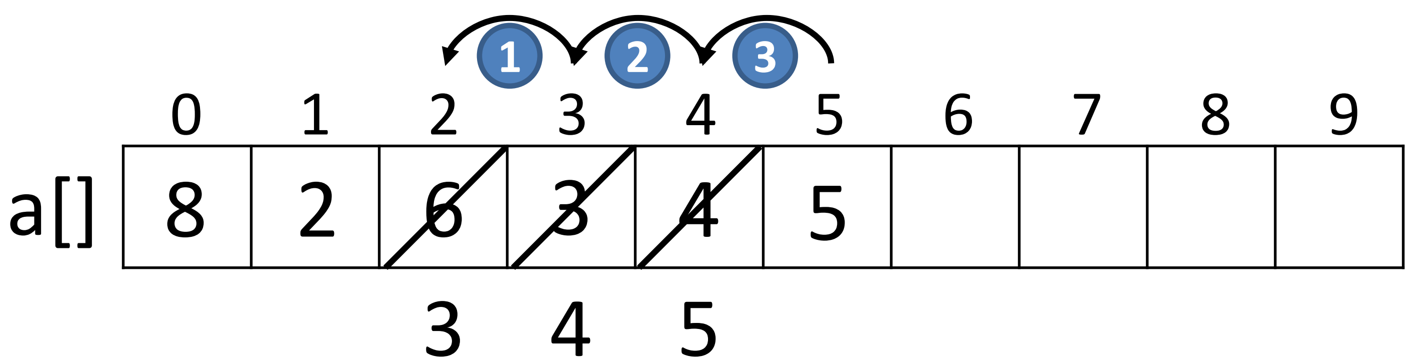
Insertion and Deletion



- Insert 2 at index 1



- Delete the value at index 2



Algorithm – Insertion

Algorithm insertElement(A,n,num,indx)

Input: An array **A** containing **n** integers and the number **num** to be inserted at index **indx**.

Output: Successful insertion of **num** at **indx**.

1. for i = n – 1 to indx do
2.   A[i + 1] = A[i]
3. A[indx] = num
4. n = n + 1

```
void insert(int a[], int num, int pos)
{   for(int i = n-1; i >= pos; i--)
          a[i+1] = a[i];
      a[pos] = num;
       n++;
}
```

Algorithm – Deletion

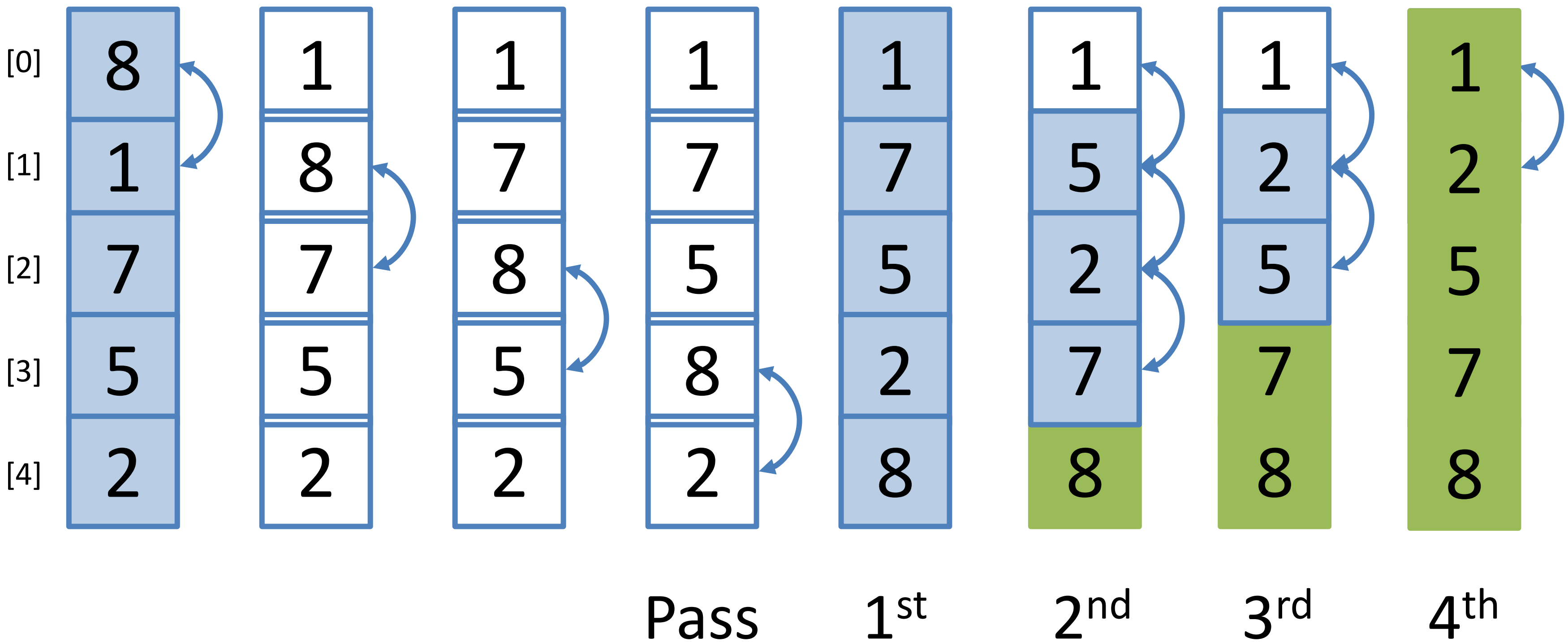
Algorithm deleteElement(A,n,indx)
Input: An array **A** containing **n** integers and the index **indx** whose value is to be deleted.
Output: Deleted value stored initially at **indx**.

1. temp = A[indx]
2. for i = indx to n – 2 do
3. A[i] = A[i + 1]
4. n = n – 1
5. return temp

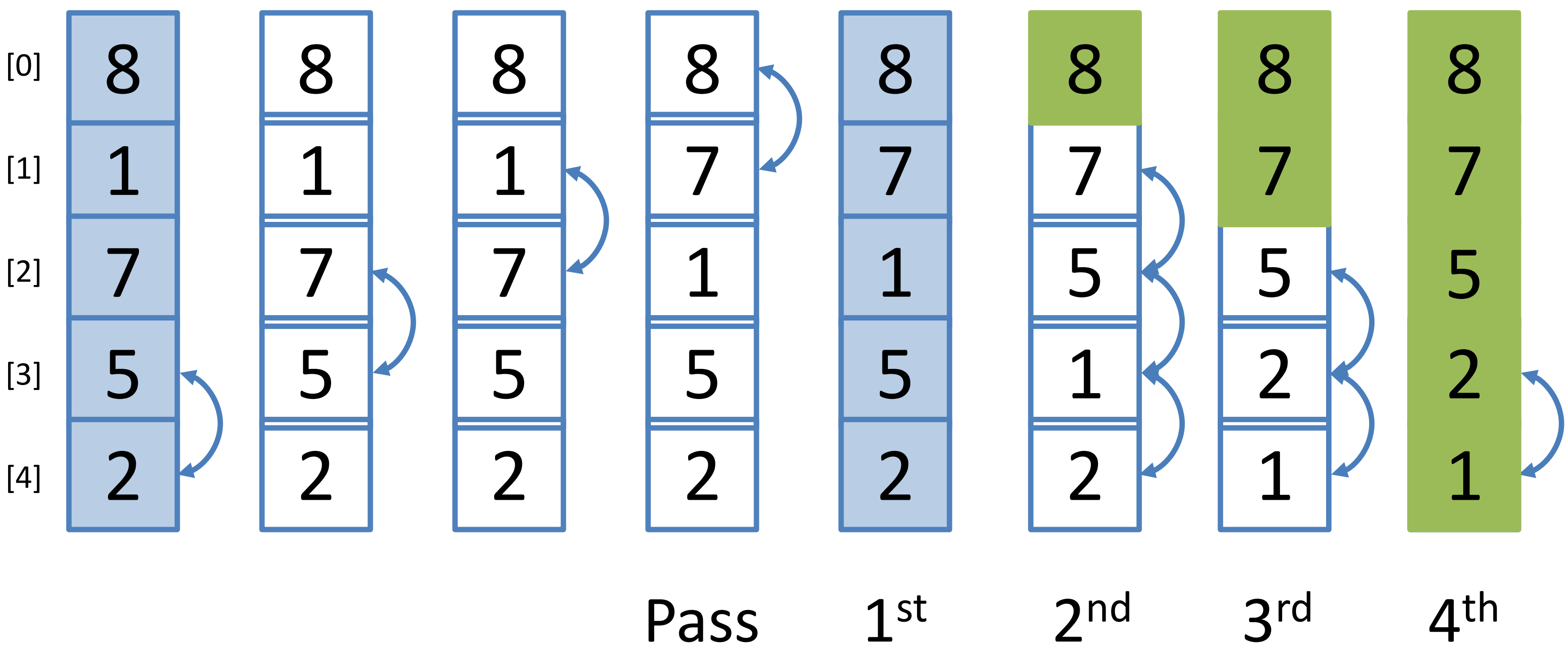
```
int deleteElement(int a[], int pos)
{   int temp = a[pos];
     for(int i = pos; i <= n-2; i++)
          a[i] = a[i+1];
     n--;
     return temp;
}
```

Bubble Sort

Bubble Sort – Ascending



Bubble Sort – Descending



Algorithm – Bubble Sort

Algorithm bubbleSort(A,n)

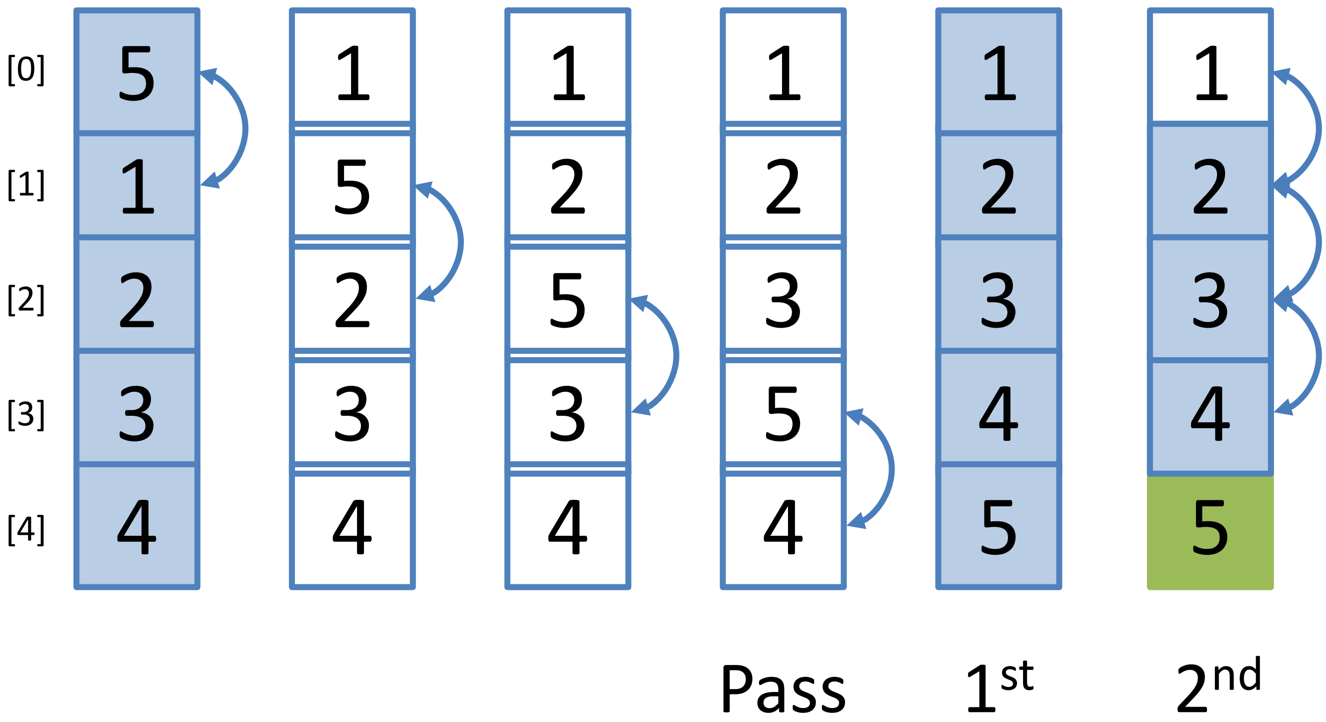
Input: An array **A** containing **n** integers.

Output: The elements of **A** get sorted in increasing order.

1. **for** i = 1 to n – 1 **do**
2. **for** j = 0 to n – i – 1 **do**
3. **if** A[j] > A[j + 1]
4. Exchange A[j] with A[j+1]

In all the cases, complexity is of the order of n^2.

Optimized Bubble Sort?



Algorithm – Optimized Bubble Sort

Algorithm bubbleSortOpt(A,n)
Input: An array **A** containing **n** integers.
Output: The elements of **A** get sorted in increasing order.

```
for  i = 1  to  n – 1
   flag = true
   for j = 0  to  n – i – 1  do
        if A[j] > A[j + 1]
           flag = false
           Exchange A[j] with A[j+1]
   if  flag == true
        break;
```

The best case complexity reduces to the order of n, but the worst and average is still n^2. So, overall the complexity is of the order of n^2 again.

Sparse Matrix

Sparse Matrix

- A matrix is sparse if many of its elements are zero.
- A matrix that is not sparse is dense.
- Two possible representations
  - Array (also known as triplet)
  - Linked list

$$\begin{bmatrix} 0 & 0 & 0 & 2 \\ 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 9 \\ 0 & 5 & 4 & 0 \end{bmatrix}$$

Array representation

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	15	0	0	22	0	-15
[1]	0	11	3	0	0	0
[2]	0	0	0	-6	0	0
[3]	0	0	0	0	0	0
[4]	91	0	0	0	0	0
[5]	0	0	28	0	0	0

Row	Col	Value
6	6	8
0	0	15
0	3	22
0	5	-15
1	1	11
1	2	3
2	3	-6
4	0	91
5	2	28

Operations

- Transpose
- Addition
- Multiplication

Transpose

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	15	0	0	22	0	-15
[1]	0	11	3	0	0	0
[2]	0	0	0	-6	0	0
[3]	0	0	0	0	0	0
[4]	91	0	0	0	0	0
[5]	0	0	28	0	0	0

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	15	0	0	0	91	0
[1]	0	11	0	0	0	0
[2]	0	3	0	0	0	28
[3]	22	0	-6	0	0	0
[4]	0	0	0	0	0	0
[5]	-15	0	0	0	0	0

Row	Col	Value
6	6	8
0	0	15
0	3	22
0	5	-15
1	1	11
1	2	3
2	3	-6
4	0	91
5	2	28

Original

Row	Col	Value
6	6	8
0	0	15
3	0	22
5	0	-15
1	1	11
2	1	3
3	2	-6
0	4	91
2	5	28

Column Major

Row	Col	Value
6	6	8
0	0	15
0	4	91
1	1	11
2	1	3
2	5	28
3	0	22
3	2	-6
5	0	-15

Row Major

Addition

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	15	0	0	22	0	-15
[1]	0	11	3	0	0	0
[2]	0	0	0	-6	0	0
[3]	0	0	0	0	0	0
[4]	91	0	0	0	0	0
[5]	0	0	28	0	0	0

+

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	15	0	0	0	91	0
[1]	0	11	0	0	0	0
[2]	0	3	0	0	0	28
[3]	22	0	-6	0	0	0
[4]	0	0	0	0	0	0
[5]	-15	0	0	0	0	0

=

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	30	0	0	22	91	-15
[1]	0	22	3	0	0	0
[2]	0	3	0	-6	0	28
[3]	22	0	-6	0	0	0
[4]	91	0	0	0	0	0
[5]	-15	0	28	0	0	0

Addition

Counter = 14

Row	Col	Value
6	6	8
0	0	15
0	3	22
0	5	-15
1	1	11
1	2	3
2	3	-6
4	0	91
5	2	28

Row	Col	Value
6	6	8
0	0	15
0	4	91
1	1	11
2	1	3
2	5	28
3	0	22
3	2	-6
5	0	-15

Row	Col	Value
6	6	14
0	0	30
0	3	22
0	4	91
0	5	-15
1	1	22
1	2	3
2	1	3
2	3	-6
2	5	28
3	0	22
3	2	-6
4	0	91
5	0	-15
5	2	28

Multiplication

- Compute A x B
- First take transpose of B.
- Multiply only if the corresponding elements are present and add them for each position in the resultant matrix.

Multiplication

	[0]	[1]	[2]	[3]
[0]	0	10	0	12
[1]	0	0	0	0
[2]	0	0	5	0
[3]	15	12	0	0

×

	[0]	[1]	[2]	[3]
[0]	0	0	8	0
[1]	0	0	0	23
[2]	0	0	9	0
[3]	20	25	0	0

=

	[0]	[1]	[2]	[3]
[0]				
[1]				
[2]				
[3]				

Multiplication

	[0]	[1]	[2]	[3]
[0]	0	10	0	12
[1]	0	0	0	0
[2]	0	0	5	0
[3]	15	12	0	0

×

	[0]	[1]	[2]	[3]
[0]	0	0	8	0
[1]	0	0	0	23
[2]	0	0	9	0
[3]	20	25	0	0

=

	[0]	[1]	[2]	[3]
[0]	240			
[1]				
[2]				
[3]				

Multiplication

	[0]	[1]	[2]	[3]
[0]	0	10	0	12
[1]	0	0	0	0
[2]	0	0	5	0
[3]	15	12	0	0

×

	[0]	[1]	[2]	[3]
[0]	0	0	8	0
[1]	0	0	0	23
[2]	0	0	9	0
[3]	20	25	0	0

=

	[0]	[1]	[2]	[3]
[0]	240	300		
[1]				
[2]				
[3]				

Multiplication

	[0]	[1]	[2]	[3]
[0]	0	10	0	12
[1]	0	0	0	0
[2]	0	0	5	0
[3]	15	12	0	0

×

	[0]	[1]	[2]	[3]
[0]	0	0	8	0
[1]	0	0	0	23
[2]	0	0	9	0
[3]	20	25	0	0

=

	[0]	[1]	[2]	[3]
[0]	240	300	0	
[1]				
[2]				
[3]				

Multiplication

	[0]	[1]	[2]	[3]
[0]	0	10	0	12
[1]	0	0	0	0
[2]	0	0	5	0
[3]	15	12	0	0

×

	[0]	[1]	[2]	[3]
[0]	0	0	8	0
[1]	0	0	0	23
[2]	0	0	9	0
[3]	20	25	0	0

=

	[0]	[1]	[2]	[3]
[0]	240	300	0	230
[1]				
[2]				
[3]				

Multiplication

	[0]	[1]	[2]	[3]
[0]	0	10	0	12
[1]	0	0	0	0
[2]	0	0	5	0
[3]	15	12	0	0

×

	[0]	[1]	[2]	[3]
[0]	0	0	8	0
[1]	0	0	0	23
[2]	0	0	9	0
[3]	20	25	0	0

=

	[0]	[1]	[2]	[3]
[0]	240	300	0	230
[1]	0			
[2]				
[3]				

Multiplication

	[0]	[1]	[2]	[3]
[0]	0	10	0	12
[1]	0	0	0	0
[2]	0	0	5	0
[3]	15	12	0	0

×

	[0]	[1]	[2]	[3]
[0]	0	0	8	0
[1]	0	0	0	23
[2]	0	0	9	0
[3]	20	25	0	0

=

	[0]	[1]	[2]	[3]
[0]	240	300	0	230
[1]	0	0	0	0
[2]	0	0	45	0
[3]	0	0	120	276

	[0]	[1]	[2]	[3]
[0]	0	10	0	12
[1]	0	0	0	0
[2]	0	0	5	0
[3]	15	12	0	0

×

	[0]	[1]	[2]	[3]
[0]	0	0	0	20
[1]	0	0	0	25
[2]	8	0	9	0
[3]	0	23	0	0

=

	[0]	[1]	[2]	[3]
[0]				
[1]				
[2]				
[3]				

Multiplication

	[0]	[1]	[2]	[3]
[0]	0	10	0	12
[1]	0	0	0	0
[2]	0	0	5	0
[3]	15	12	0	0

×

	[0]	[1]	[2]	[3]
[0]	0	0	8	0
[1]	0	0	0	23
[2]	0	0	9	0
[3]	20	25	0	0

=

	[0]	[1]	[2]	[3]
[0]	240	300	0	230
[1]	0	0	0	0
[2]	0	0	45	0
[3]	0	0	120	276

	[0]	[1]	[2]	[3]
[0]	0	10	0	12
[1]	0	0	0	0
[2]	0	0	5	0
[3]	15	12	0	0

×

	[0]	[1]	[2]	[3]
[0]	0	0	0	20
[1]	0	0	0	25
[2]	8	0	9	0
[3]	0	23	0	0

=

	[0]	[1]	[2]	[3]
[0]	240			
[1]				
[2]				
[3]				

Multiplication

	[0]	[1]	[2]	[3]
[0]	0	10	0	12
[1]	0	0	0	0
[2]	0	0	5	0
[3]	15	12	0	0

×

	[0]	[1]	[2]	[3]
[0]	0	0	8	0
[1]	0	0	0	23
[2]	0	0	9	0
[3]	20	25	0	0

=

	[0]	[1]	[2]	[3]
[0]	240	300	0	230
[1]	0	0	0	0
[2]	0	0	45	0
[3]	0	0	120	276

	[0]	[1]	[2]	[3]
[0]	0	10	0	12
[1]	0	0	0	0
[2]	0	0	5	0
[3]	15	12	0	0

×

	[0]	[1]	[2]	[3]
[0]	0	0	0	20
[1]	0	0	0	25
[2]	8	0	9	0
[3]	0	23	0	0

=

	[0]	[1]	[2]	[3]
[0]	240	300		
[1]				
[2]				
[3]				

Multiplication

	[0]	[1]	[2]	[3]
[0]	0	10	0	12
[1]	0	0	0	0
[2]	0	0	5	0
[3]	15	12	0	0

×

	[0]	[1]	[2]	[3]
[0]	0	0	8	0
[1]	0	0	0	23
[2]	0	0	9	0
[3]	20	25	0	0

=

	[0]	[1]	[2]	[3]
[0]	240	300	0	230
[1]	0	0	0	0
[2]	0	0	45	0
[3]	0	0	120	276

	[0]	[1]	[2]	[3]
[0]	0	10	0	12
[1]	0	0	0	0
[2]	0	0	5	0
[3]	15	12	0	0

×

	[0]	[1]	[2]	[3]
[0]	0	0	0	20
[1]	0	0	0	25
[2]	8	0	9	0
[3]	0	23	0	0

=

	[0]	[1]	[2]	[3]
[0]	240	300	0	
[1]				
[2]				
[3]				

Multiplication

	[0]	[1]	[2]	[3]
[0]	0	10	0	12
[1]	0	0	0	0
[2]	0	0	5	0
[3]	15	12	0	0

×

	[0]	[1]	[2]	[3]
[0]	0	0	8	0
[1]	0	0	0	23
[2]	0	0	9	0
[3]	20	25	0	0

=

	[0]	[1]	[2]	[3]
[0]	240	300	0	230
[1]	0	0	0	0
[2]	0	0	45	0
[3]	0	0	120	276

	[0]	[1]	[2]	[3]
[0]	0	10	0	12
[1]	0	0	0	0
[2]	0	0	5	0
[3]	15	12	0	0

×

	[0]	[1]	[2]	[3]
[0]	0	0	0	20
[1]	0	0	0	25
[2]	8	0	9	0
[3]	0	23	0	0

=

	[0]	[1]	[2]	[3]
[0]	240	300	0	230
[1]				
[2]				
[3]				

Multiplication

	[0]	[1]	[2]	[3]
[0]	0	10	0	12
[1]	0	0	0	0
[2]	0	0	5	0
[3]	15	12	0	0

×

	[0]	[1]	[2]	[3]
[0]	0	0	8	0
[1]	0	0	0	23
[2]	0	0	9	0
[3]	20	25	0	0

=

	[0]	[1]	[2]	[3]
[0]	240	300	0	230
[1]	0	0	0	0
[2]	0	0	45	0
[3]	0	0	120	276

	[0]	[1]	[2]	[3]
[0]	0	10	0	12
[1]	0	0	0	0
[2]	0	0	5	0
[3]	15	12	0	0

×

	[0]	[1]	[2]	[3]
[0]	0	0	0	20
[1]	0	0	0	25
[2]	8	0	9	0
[3]	0	23	0	0

=

	[0]	[1]	[2]	[3]
[0]	240	300	0	230
[1]	0			
[2]				
[3]				

Multiplication

	[0]	[1]	[2]	[3]
[0]	0	10	0	12
[1]	0	0	0	0
[2]	0	0	5	0
[3]	15	12	0	0

×

	[0]	[1]	[2]	[3]
[0]	0	0	8	0
[1]	0	0	0	23
[2]	0	0	9	0
[3]	20	25	0	0

=

	[0]	[1]	[2]	[3]
[0]	240	300	0	230
[1]	0	0	0	0
[2]	0	0	45	0
[3]	0	0	120	276

	[0]	[1]	[2]	[3]
[0]	0	10	0	12
[1]	0	0	0	0
[2]	0	0	5	0
[3]	15	12	0	0

×

	[0]	[1]	[2]	[3]
[0]	0	0	0	20
[1]	0	0	0	25
[2]	8	0	9	0
[3]	0	23	0	0

=

	[0]	[1]	[2]	[3]
[0]	240	300	0	230
[1]	0	0	0	0
[2]	0	0	45	0
[3]	0	0	120	276

Multiplication

Counter = 0

Row	Col	Value
4	4	5
0	1	10
0	3	12
2	2	5
3	0	15
3	1	12

Row	Col	Value
4	4	5
0	3	20
1	3	25
2	0	8
2	2	9
3	1	23

Row	Col	Value
4	4	5
0	2	8
1	3	23
2	2	9
3	0	20
3	1	25

Row	Col	Value
4	4	

Multiplication

Counter = 6

Row	Col	Value
4	4	5
0	1	10
0	3	12
2	2	5
3	0	15
3	1	12

Row	Col	Value
4	4	5
0	3	20
1	3	25
2	0	8
2	2	9
3	1	23

Row	Col	Value
4	4	6
0	0	240
0	1	300
0	3	230
2	2	45
3	2	120
3	3	276

	[0]	[1]	[2]	[3]
[0]	240	300	0	230
[1]	0	0	0	0
[2]	0	0	45	0
[3]	0	0	120	276