UNIT 2:

Arrays, Searching and Sorting:

Arrays

Array is a data structure that is used to store variables that are of similar data types at contiguous locations. The main advantage of the array is random access and cache friendliness.

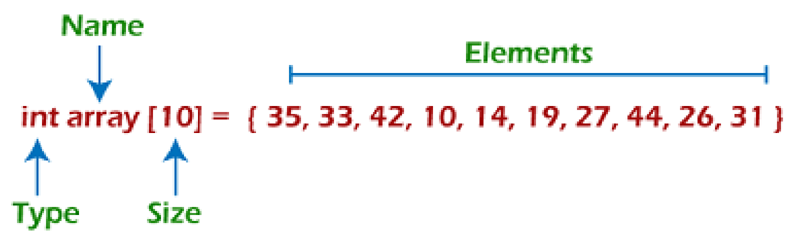
Properties of array

There are some of the properties of an array that are listed as follows -

- Each element in an array is of the same data type and carries the same size that is 4 bytes.
- Elements in the array are stored at contiguous memory locations from which the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Representation of an array

We can represent an array in various ways in different programming languages. As an illustration, let's see the declaration of array in C language -



As per the above illustration, there are some of the following important points -

- Index starts with 0.
- The array's length is 10, which means we can store 10 elements.
- Each element in the array can be accessed via its index.

Why are arrays required?

Arrays are useful because -

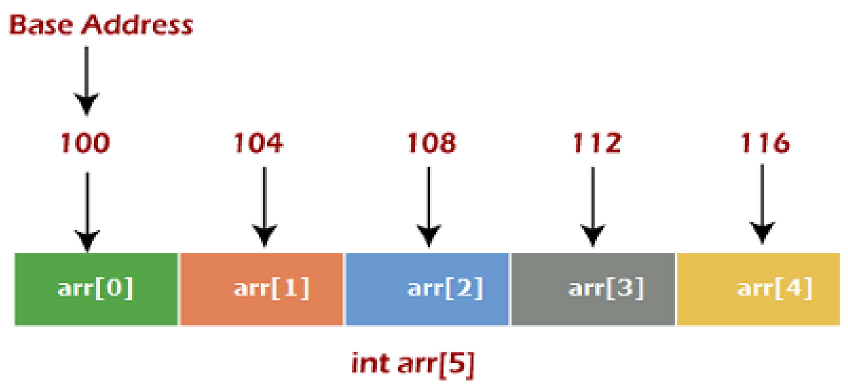
- Sorting and searching a value in an array is easier.
- Arrays are best to process multiple values quickly and easily.
- **Arrays are good for storing multiple values in a single variable -** In computer programming, most cases require storing a large number of data of a similar type. To store such an amount of data, we need to define a large number of variables. It would be very difficult to remember the names of all the variables while writing the programs. Instead of naming all the variables with a different name, it is better to define an array and store all the elements into it.

Memory allocation of an array

As stated above, all the data elements of an array are stored at contiguous locations in the main memory. The name of the array represents the base address or the address of the first element in the main memory. Each element of the array is represented by proper indexing.

We can define the indexing of an array in the below ways -

1. 0 (zero-based indexing): The first element of the array will be arr[0].
2. 1 (one-based indexing): The first element of the array will be arr[1].
3. n (n - based indexing): The first element of the array can reside at any random index number.



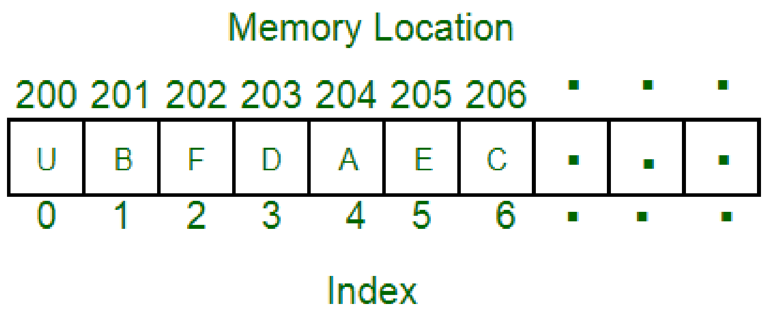
In the above image, we have shown the memory allocation of an array arr of size 5. The array follows a 0-based indexing approach. The base address of the array is 100 bytes. It is the address of arr[0]. Here, the size of the data type used is 4 bytes; therefore, each element will take 4 bytes in the memory.

There are mainly three types of the array:

- One Dimensional (1D) Array
- Two Dimension (2D) Array
- Multidimensional Array

One Dimensional Array:

- It is a list of the variable of similar data types.
- It allows random access and all the elements can be accessed with the help of their index.
- The size of the array is fixed.
- For a dynamically sized array, vector can be used in C++.
- Representation of 1D array:

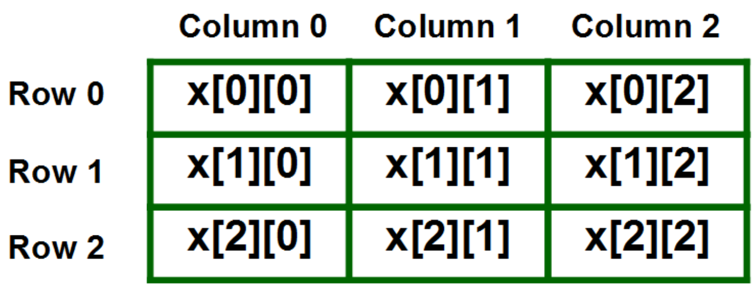


Two Dimensional Array:

It is a list of lists of the variable of the same data type.

- It also allows random access and all the elements can be accessed with the help of their index.
- It can also be seen as a collection of 1D arrays. It is also known as the Matrix.

- Its dimension can be increased from 2 to 3 and 4 so on.
- They all are referred to as a multi-dimension array.
- The most common multidimensional array is a 2D array.
- Representation of 2 D array:



	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Difference Table:

Basis	**One Dimension Array**	**Two Dimension Array**
Definition	Store a single list of the element of a similar data type.	Store a 'list of lists' of the element of a similar data type.
Representation	Represent multiple data items as a list.	Represent multiple data items as a table consisting of rows and columns.
Declaration	The declaration varies for different programming language: 1. For C++, ***datatype variable_name[row]*** 2. For Java, ***datatype [] variable_name= new datatype[row]***	The declaration varies for different programming language: 1. For C++, ***datatype variable_name[row][column]*** 2. For Java, ***datatype [][] variable_name= new datatype[row][column]***

Dimension	One	Two
Size(bytes)	size of(datatype of the variable of the array) * size of the array	size of(datatype of the variable of the array)* the number of rows* the number of columns.
Address calculation.	Address of a[index] is equal to (base Address+ Size of each element of array * index).	Address of a[i][j] can be calculated in two ways row-major and column-major 1. **Column Major:** Base Address + Size of each element (number of rows(j-lower bound of the column)+(i-lower bound of the rows)) 2. **Row Major:** Base Address + Size of each element (number of columns(i-lower bound of the row)+(j-lower bound of the column))

Example	int arr[5]; //an array with one row and five columns will be created.
	{a , b , c , d , e}
		int arr[2][5]; //an array with two rows and five columns will be created.
		a b c d e
		f g h i j

Multidimensional array

Multidimensional arrays are one of the most powerful features of the C programming language. They allow you to store data in a ***table-like format***, where each ***row*** and ***column*** can be accessed using an ***index***. In this blog post, we'll look at multidimensional arrays in C, including their ***syntax***, ***example usage***, and ***output***.

Syntax of Multidimensional Arrays in C

To create a ***multidimensional array*** in C, you need to specify the number of dimensions and the ***size*** of each dimension. The general syntax for declaring a multidimensional array is as follows:

```
type array_name[size1][size2]...[sizeN];
```

Here, ***type*** is the ***data type*** of the elements that will be stored in the array, ***array_name*** is the name of the array, and ***size1***, ***size2***, ..., ***sizeN*** are the sizes of each dimension of the array.

For example, the following code declares a ***2-dimensional array*** of integers with ***3 rows*** and ***4 columns*: int** my_array[3][4];

It creates an array with ***3 rows*** and ***4 columns***, for a total of ***12 elements***. Each element is of type ***int***.

Accessing Elements of Multidimensional Arrays

To access an element of a ***multidimensional array***, you need to specify the ***indices*** for each dimension. For example, to access the element in the ***second row*** and ***third column*** of ***my_array***, you would use the following syntax:

```
int element = my_array[1][2];
```

Note that the ***indices*** start at ***0***, so the first row is ***my_array[0]***, the second row is ***my_array[1]***, and so on. Similarly, the first column of each row is ***my_array[i][0]***, and so on.

Initializing Multidimensional Arrays

You can initialize a multidimensional array when you declare it by specifying the values for each element in the array. For example, the following code declares and initializes a ***2-dimensional array*** of integers with ***2 rows*** and ***3 columns*:**

```
int my_array[2][3] = {
  {1, 2, 3},
  {4, 5, 6}
};
```

It creates an array with ***2 rows*** and ***3 columns*** and initializes the elements to the specified values.

Iterating Over Multidimensional Arrays

You can iterate over the elements of a multidimensional array using ***nested loops***. For example, the following code iterates over the elements of ***my_array*** and prints their values:

```
for (int i = 0; i< 2; i++) {
  for (int j = 0; j < 3; j++) {
printf("%d ", my_array[i][j]);
  }
printf("\n");
}
```

This code loops through each row and column of ***my_array***, and prints each element with a space between them. The ***printf("\n")*** statement is used to print a ***newline*** character after each row.

Example Usage of Multidimensional Arrays in C

Let's look at a practical example of using ***multidimensional arrays*** in C. Suppose we want to create a program that stores the grades of ***5 students*** in ***4 different subjects***. We can use a ***2-dimensional array*** to store this data, where each row represents a ***student***, and each column represents a ***subject***.

Example:

Here's an example program that prompts the user to enter the grades for each ***student*** and ***subject***, and then calculates the ***average grade*** for each ***student*** and ***subject*:**

```
#include <stdio.h>

int main() {
  int grades[5][4];

  // Prompt user to enter grades
```

```c
for (int i = 0; i< 5; i++) {
printf("Enter grades for student %d:\n", i+1);
for (int j = 0; j < 4; j++) {
printf("Subject %d: ", j+1);
scanf("%d", &grades[i][j]);
}
}

// Calculate average grade for each student
printf("\nAverage grade for each student:\n");
for (int i = 0; i< 5; i++) {
float sum = 0;
for (int j = 0; j < 4; j++) {
sum += grades[i][j];
}
float avg = sum / 4;
printf("Student %d: %.2f\n", i+1, avg);
}

// Calculate average grade for each subject
printf("\nAverage grade for each subject:\n");
for (int j = 0; j < 4; j++) {
float sum = 0;
for (int i = 0; i< 5; i++) {
sum += grades[i][j];
}
float avg = sum / 5;
printf("Subject %d: %.2f\n", j+1, avg);
}

return 0;
}
```

Output:

```
Enter grades for student 1:
Subject 1: 80
Subject 2: 75
Subject 3: 90
Subject 4: 85
Enter grades for student 2:
Subject 1: 70
Subject 2: 85
Subject 3: 80
Subject 4: 75
Enter grades for student 3:
Subject 1: 90
Subject 2: 80
Subject 3: 85
Subject 4: 95
Enter grades for student 4:
Subject 1: 75
Subject 2: 90
Subject 3: 75
Subject 4: 80
Enter grades for student 5:
Subject 1: 85
Subject 2: 70
Subject 3: 80
Subject 4: 90
Average grade for each student:
Student 1: 82.50
Student 2: 77.50
Student 3: 87.50
Student 4: 80.00
Student 5: 81.25
Average grade for each subject:
Subject 1: 80.00
Subject 2: 80.00
Subject 3: 82.00
Subject 4: 85.00
```

Explanation:

In this program, we first declare a ***2-dimensional array 'grades'*** with ***5 rows*** and ***4 columns***, to store the grades for each student and subject. After that, we prompt the user to enter the grades for each student and subject using ***nested loops***. The ***'printf'*** statements are used to display the prompt, and the ***'scanf'*** statement is used to read the input from the user and store it in the appropriate element of the array.

Next, we calculate the average grade for each student and subject using nested loops. The ***'sum'*** variable is used to keep track of the total grade for each student or subject, and the ***'avg'*** variable is used to calculate the average grade by dividing the sum by the number of subjects or students. Finally, we use ***'printf'*** statements to display the average grades for each student and subject. As we can see, the program successfully calculates the average grades for each student and subject based on the input provided by the user.

In addition to the example program, we discussed earlier, there are many other applications of multidimensional arrays in C. For example, you can use a ***3-dimensional*** array to store and manipulate data in a ***3-dimensional space***, such as a ***cube*** or a ***sphere***. Similarly, you can use a ***4-dimensional array*** to represent data that varies across four dimensions, such as ***time***, ***space***, ***temperature***, and ***pressure***.

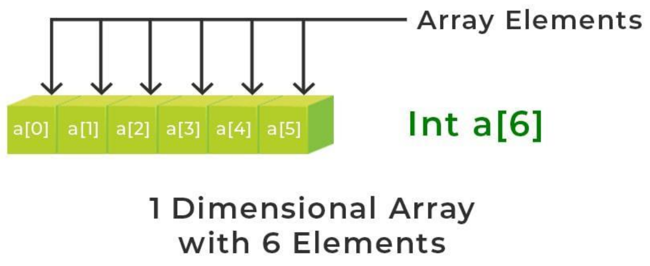
One common use of multidimensional arrays in C is for ***image processing*** and ***computer vision applications***. For example, you can use a 2***-dimensional array*** to represent an image, with each element of the array representing a pixel in the image. By manipulating the values of the elements in the array, you can perform a wide range of operations on the image, such as scaling, rotating, cropping, and filtering.

Another application of multidimensional arrays is in ***numerical analysis*** and ***scientific computing***. Many ***scientific simulations*** and ***calculations*** require the use of multidimensional arrays to represent complex data structures and perform numerical operations. By using arrays with high precision and accuracy, scientists and engineers can model and analyze complex phenomena in fields such as physics, chemistry, and biology. One important consideration when working with multidimensional arrays in C is memory management. Because multidimensional arrays can be quite large and complex, it is important to be mindful of how the data is stored in memory and how it is accessed by the program. You should be aware of the potential for memory leaks, buffer overflows, and other memory-related errors that can occur when working with large and complex data structures.

Address calculation using column and row major ordering

This article focuses on calculating the address of any element in a 1-Dimensional, 2-Dimensional, and 3-Dimensional array in Row major order and Column major order.

Calculating the address of any element In the 1-D array:
A 1-dimensional array (or single-dimension array) is a type of linear array. Accessing its elements involves a single subscript that can either represent a row or column index.
Example:

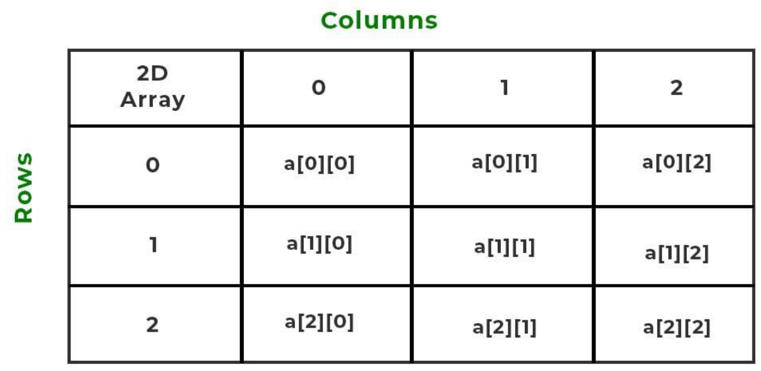


1-D array

To find the address of an element in an array the following formula is used-
Address of A[I] = B + W * (I – LB)
I = Subset of element whose address to be found,
B = Base address,
W = Storage size of one element store in any array(in byte),
LB = Lower Limit/Lower Bound of subscript(If not specified assume zero).

Example: Given the base address of an array **A[1300 ………… 1900]** as **1020** and the size of each element is 2 bytes in the memory, find the address of **A[1700].**
Solution:
Given:
Base address B = 1020
Lower Limit/Lower Bound of subscript LB = 1300
Storage size of one element store in any array W = 2 Byte
Subset of element whose address to be found I = 1700
Formula used:
*Address of A[I] = B + W * (I – LB)*
Solution:
*Address of A[1700] = 1020 + 2 * (1700 – 1300)*
*= 1020 + 2 * (400)*
= 1020 + 800
Address of A[1700] = 1820
Calculate the address of any element in the 2-D array:
The 2-dimensional array can be defined as an array of arrays. The 2-Dimensional arrays are organized as matrices which can be represented as the collection of rows and columns as array[M][N] where M is the number of rows and N is the number of columns.

Example:


Columns
Rows

2D Array	0	1	2
0	a[0][0]	a[0][1]	a[0][2]
1	a[1][0]	a[1][1]	a[1][2]
2	a[2][0]	a[2][1]	a[2][2]

2-D array

To find the address of any element in a **2-Dimensional** array there are the following two ways-

1. Row Major Order
2. Column Major Order

1. Row Major Order:

Row major ordering assigns successive elements, moving across the rows and then down the next row, to successive memory locations. In simple language, the elements of an array are stored in a Row-Wise fashion.

To find the address of the element using row-major order uses the following formula:

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

I = Row Subset of an element whose address to be found,
J = Column Subset of an element whose address to be found,
B = Base address,
W = Storage size of one element store in an array(in byte),
LR = Lower Limit of row/start row index of the matrix(If not given assume it as zero),
LC = Lower Limit of column/start column index of the matrix(If not given assume it as zero),
N = Number of column given in the matrix.

Example: Given an array, **arr[1………10][1………15]** with base value **100** and the size of each element is **1 Byte** in memory. Find the address of **arr[8][6]** with the help of row-major order.

Solution:

Given:

Base address B = 100
Storage size of one element store in any array W = 1 Bytes
Row Subset of an element whose address to be found I = 8
Column Subset of an element whose address to be found J = 6
Lower Limit of row/start row index of matrix LR = 1
Lower Limit of column/start column index of matrix = 1
Number of column given in the matrix N = Upper Bound – Lower Bound + 1
= 15 – 1 + 1
= 15

Formula:
*Address of A[I][J] = B + W * ((I – LR) * N + (J – LC))*
Solution:
*Address of A[8][6] = 100 + 1 * ((8 – 1) * 15 + (6 – 1))*
*= 100 + 1 * ((7) * 15 + (5))*
*= 100 + 1 * (110)*
Address of A[I][J] = 210
2. Column Major Order:
If elements of an array are stored in a column-major fashion means moving across the column and then to the next column then it's in column-major order. To find the address of the element using column-major order use the following formula:

Address of A[I][J] = B + W * ((J – LC) * M + (I – LR))
I = Row Subset of an element whose address to be found,
J = Column Subset of an element whose address to be found,
B = Base address,
W = Storage size of one element store in any array(in byte),
LR = Lower Limit of row/start row index of matrix(If not given assume it as zero),
LC = Lower Limit of column/start column index of matrix(If not given assume it as zero),
M = Number of rows given in the matrix.

Example: Given an array **arr[1………10][1………15]** with a base value of **100** and the size of each element is **1 Byte** in memory find the address of arr[8][6] with the help of column-major order.
Solution:
Given:
Base address B = 100
Storage size of one element store in any array W = 1 Bytes
Row Subset of an element whose address to be found I = 8
Column Subset of an element whose address to be found J = 6
Lower Limit of row/start row index of matrix LR = 1
Lower Limit of column/start column index of matrix = 1
Number of Rows given in the matrix M = Upper Bound – Lower Bound + 1
= 10 – 1 + 1
= 10
Formula: used
*Address of A[I][J] = B + W * ((J – LC) * M + (I – LR))*
*Address of A[8][6] = 100 + 1 * ((6 – 1) * 10 + (8 – 1))*
*= 100 + 1 * ((5) * 10 + (7))*
*= 100 + 1 * (57)*
Address of A[I][J] = 157
From the above examples, it can be observed that for the same position two different address locations are obtained that's because in row-major order movement is done across the rows and

then down to the next row, and in column-major order, first move down to the first column and then next column. So both the answers are right.

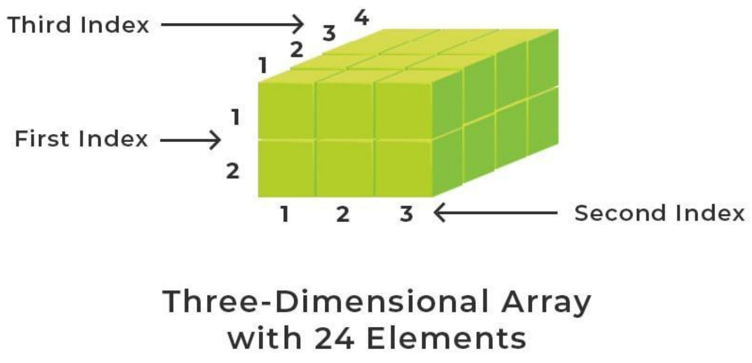
So it's all based on the position of the element whose address is to be found for some cases the same answers is also obtained with row-major order and column-major order and for some cases, different answers are obtained.

Calculate the address of any element in the 3-D Array:
A **3-Dimensional** array is a collection of 2-Dimensional arrays. It is specified by using three subscripts:

1. Block size
2. Row size
3. Column size

More dimensions in an array mean more data can be stored in that array.

Example:



3-D array

To find the address of any element in 3-Dimensional arrays there are the following two ways-

- Row Major Order
- Column Major Order

1. Row Major Order:
To find the address of the element using row-major order, use the following formula:

Address of A[i][j][k] = B + W *(M * N(i-x) + N *(j-y) + (k-z))
Here:

B = Base Address (start address)
W = Weight (storage size of one element stored in the array)
M = Row (total number of rows)
N = Column (total number of columns)
P = Width (total number of cells depth-wise)
x = Lower Bound of Row
y = Lower Bound of Column
z = Lower Bound of Width

Example: Given an array, **arr[1:9, -4:1, 5:10]** with a base value of **400** and the size of each element is **2 Bytes** in memory find the address of element **arr[5][-1][8]** with the help of row-major order?
Solution:
Given:
Row Subset of an element whose address to be found I = 5
Column Subset of an element whose address to be found J = -1
Block Subset of an element whose address to be found K = 8
Base address B = 400
Storage size of one element store in any array(in Byte) W = 2
Lower Limit of row/start row index of matrix x = 1
Lower Limit of column/start column index of matrix y = -4
Lower Limit of blocks in matrix z = 5
M(row) = Upper Bound – Lower Bound + 1 = 9 – 1 + 1 = 9
N(Column)= Upper Bound – Lower Bound + 1 = 1 – (-4) + 1 = 6

Formula used:
*Address of[I][J][K] =B + W (M * N(i-x) + N *(j-y) + (k-z))*
Solution:
*Address of arr[5][-1][8] = 400 + 2 * {[9 * 6 * (5 – 1)] + 6 * [(-1 + 4)]} + [8 – 5]*
*= 400 + 2 * (9*6*4)+(6*3)+3*
*= 400 + 2 * (237)*
= 874
2. Column Major Order:
To find the address of the element using column-major order, use the following formula:1

Address of A[i][j][k]= B + W(M * N(i – x) + M *(k – z) + (j – y))
Here:

B = Base Address (start address)
W = Weight (storage size of one element stored in the array)
M = Row (total number of rows)
N = Column (total number of columns)
P = Width (total number of cells depth-wise)
x = Lower Bound of Row
y = Lower Bound of Column
z = Lower Bound of Width

Example: Given an array **arr[1:8, -5:5, -10:5]** with a base value of **400** and the size of each element is **4 Bytes** in memory find the address of element **arr[3][3][3]** with the help of column-major order?
Solution:

Given:
Row Subset of an element whose address to be found I = 3
Column Subset of an element whose address to be found J = 3
Block Subset of an element whose address to be found K = 3
Base address B = 400

Storage size of one element store in any array(in Byte) W = 4
Lower Limit of row/start row index of matrix x = 1
Lower Limit of column/start column index of matrix y = -5
Lower Limit of blocks in matrix z = -10
M (row)= Upper Bound – Lower Bound + 1 = 8-1+1 = 8
N (column)= Upper Bound – Lower Bound + 1 = 5 +5 + 1 = 11
Formula used:
*Address of[i][j][k] = B + W(M * N(i – x) + M * (j-y) + (k – z))*
Solution:
*Address of arr[3][3][3] = 400 + 4 * ((8*11*(3-1)+8*(3-(-5)+(3-(-10)))*
*= 400 + 4 * ((88*2 + 8*8+13)*
*= 400 + 4 * (253)*
= 400 + 1012
= 1412

Basic Operations in the Arrays

The basic operations in the Arrays are insertion, deletion, searching, display, traverse, and update. These operations are usually performed to either modify the data in the array or to report the status of the array.

Following are the basic operations supported by an array.

- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.
- **Display** − Displays the contents of the array.

Applications of Arrays:

Arrays are fundamental data structures that store a collection of elements of the same data type in contiguous memory locations. They have numerous applications across various fields and programming tasks. Some common applications of arrays include:

1. Data Storage and Manipulation: Arrays are commonly used to store and manage large sets of data, such as lists of numbers, characters, or objects. They provide easy access to individual elements through indexing and facilitate efficient data retrieval and manipulation.
2. Algorithms: Arrays play a crucial role in many algorithms, such as searching, sorting, and dynamic programming. For example, sorting algorithms like quicksort and mergesort heavily use arrays to rearrange elements efficiently.
3. Matrices and Multidimensional Arrays: Arrays can be used to represent matrices and multidimensional data structures. Applications include image processing, graphics rendering, simulations, and scientific computations.

4. Dynamic Data Structures: Arrays serve as the basis for implementing more complex data structures like stacks, queues, linked lists, and trees. For example, stacks and queues can be implemented using arrays to store elements in a specific order.
5. Buffer Management: Arrays are widely used in computer science and networking for buffer management, where they hold a temporary set of data before it is processed.
6. Caches: Arrays are used in caching mechanisms to store frequently accessed data items, improving performance by reducing access times to slower storage locations.
7. Lookup Tables: Arrays are used to create lookup tables, which associate input values with corresponding output values. This approach speeds up computations by avoiding repetitive calculations.
8. Hashing: Arrays play a significant role in hash tables, used for fast data retrieval in key-value storage systems and databases.
9. Dynamic Programming: In dynamic programming, arrays are used to store intermediate results and reduce redundant computations in recursive algorithms.
10. Graph Algorithms: Arrays are employed in graph representations, such as adjacency matrices or adjacency lists, to efficiently solve graph-related problems like finding the shortest path, traversals, and network analysis.
11. Image Processing: In image processing, arrays are utilized to represent images as grids of pixels, enabling various transformations, filtering, and feature extraction.
12. Audio Processing: Arrays are used in digital signal processing to store and process audio signals for tasks like noise reduction, filtering, and compression.

Overall, arrays are essential tools in programming and data processing due to their simplicity, efficiency, and versatility. Understanding their applications allows developers to choose the appropriate data structures for specific tasks, leading to optimized and well-organized programs.

Sparse Polynomial representation and addition:

Polynomial is an expression which is composed of terms, wherein terms are composed of coefficient and exponent. An example of a polynomial is: $4x^3+5x^2+6x+9$. This polynomial is composed of four terms with the following sets of coefficient and exponent – {(4,3), (5,2), (6,1), (9,0)}. Thus representation of a polynomial using arrays is straightforward. The subscripts of the array may be considered as exponents and the coefficients may be stored at an appropriate place referred to by the subscript. Array representation for the above example is:

arr:	9	6	5	4	coefficients

subscripts: 0 1 2 3 exponents

The above representation works fine for the above example, but for polynomials such as 5×6+7, array representation is considered feasible on account of memory usage, since it results in a sparse arrangement as follows:

arr	7	0	0	0	0	0	5
	0	1	2	3	4	5	6 (Subscripts)

Sparse matrix representation may be considered for the above matrix as given below:

Rows	**Cols**	**Value**	
1	7	2	(Total)
0	0	7	
0	6	5	

For addition of two polynomials using arrays, subscript-wise elements may be added to result in the polynomial containing result of addition of two polynomials.

Example:

Polynomial 1: $4x^3 + 5x^2 + 6x + 7$

Polynomial 2: $3x^3 + 4x^2 + 2x + 2$

$7x^3 + 9x^2 + 8x + 9$

Array Representation:

Subscripts: 0 1 2 3

Polynomial 1:	7	6	5	4
Polynomial 2:	2	2	4	3
Result of sum:	9	8	9	7

Matrix multiplication

In this section we will see how to multiply two matrices. The matrix multiplication can only be performed, if it satisfies this condition. Suppose two matrices are A and B, and their dimensions are A (m x n) and B (p x q) the resultant matrix can be found if and only if n = p. Then the order of the resultant matrix C will be (m x q).

Algorithm

```
matrixMultiply(A, B):
Assume dimension of A is (m x n), dimension of B is (p x q)
Begin
  if n is not same as p, then exit
  otherwise define C matrix as (m x q)
  for i in range 0 to m - 1, do
    for j in range 0 to q – 1, do
      for k in range 0 to p, do
        C[i, j] = C[i, j] + (A[i, k] * A[k, j])
      done
    done
  done
End
```

Example

```
#include<iostream>
using namespace std;
int main() {
  int product[10][10], r1=3, c1=3, r2=3, c2=3, i, j, k;
  int a[3][3] = {
    {2, 4, 1},
    {2, 3, 9},
    {3, 1, 8}
  };
  int b[3][3] = {
    {1, 2, 3},
    {3, 6, 1},
    {2, 4, 7}
  };
  if (c1 != r2) {
    cout<<"Column of first matrix should be equal to row of second matrix";
  } else {
    cout<<"The first matrix is:"<<endl;
    for(i=0; i<r1; ++i) {
      for(j=0; j<c1; ++j)
        cout<<a[i][j]<<" ";
      cout<<endl;
    }
    cout<<endl;
    cout<<"The second matrix is:"<<endl;
    for(i=0; i<r2; ++i) {
      for(j=0; j<c2; ++j)
        cout<<b[i][j]<<" ";
      cout<<endl;
    }
    cout<<endl;
    for(i=0; i<r1; ++i)
      for(j=0; j<c2; ++j) {
        product[i][j] = 0;
      }
    for(i=0; i<r1; ++i)
      for(j=0; j<c2; ++j)
        for(k=0; k<c1; ++k) {
          product[i][j]+=a[i][k]*b[k][j];
        }
    cout<<"Product of the two matrices is:"<<endl;
    for(i=0; i<r1; ++i) {
      for(j=0; j<c2; ++j)
```

```
            cout<<product[i][j]<<" ";
          cout<<endl;
       }
    }
    return 0;
}
```

Output

```
The first matrix is:
2 4 1
2 3 9
3 1 8
The second matrix is:
1 2 3
3 6 1
2 4 7
Product of the two matrices is:
16 32 17
29 58 72
22 44 66
```

Searching

Searching is the process of looking through the data contained in a data structure and determining if a specific value is present. (And potentially returning it.)
The contains methods of the ArrayList, for example, is a method that searches the list for a given value and returns true of false.

Linear Search

The most basic search algorithm is a *linear search*. This is just a fancy name for “start at the first element and go through the list until you find what you are looking for.” It is the simplest search. Here is a simple implementation for an array:

```
public static int linearSearch(int[] a, int value) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        if (a[i] == value)
            return i;
    }
    return -1;
}
```

If you look carefully at it, you'll realize that the efficiency is $O(N)$�(�). Is it possible to do better than this, though?

Binary Search

Binary search is a searching algorithm that is faster than $O(N)$�(�). It has a catch, though: It requires the array you are searching to already be sorted. If you have a sorted array, then you can search it as follows:

1. Imagine you are searching for element *v*.
2. Start at the middle of the array. Is *v* smaller or larger than the middle element? If smaller, then do the algorithm again on the left half of the array. If larger, then do the algorithm again on the right half of the array.

Basically, with every step, we cut the number of elements we are searching for in half. Here is a simple implementation:

```
public static int binarySearch(int[] a, int value) {
    int lowIndex = 0;
    int highIndex = a.length - 1;
    while (lowIndex <= highIndex) {
        int midIndex = (lowIndex + highIndex) / 2;
        if (value < a[midIndex])
            highIndex = midIndex - 1;
        else if (value > a[midIndex])
            lowIndex = midIndex + 1;
        else // found it!
            return midIndex;
    }
    return -1; // The value doesn't exist
}
```

The efficiency of this is more complicated than the things we've been analyzing so far. We aren't doing N� work in the worst case, because even in the worst case we are cutting the array in half each step. When an algorithm cuts the data in half at each step, that efficiency is *logarithmic*. In this case, $O(\log_2 N)$�(log2 �). In efficiency, we frequently drop the base 2 from logarithmic efficiency, so we can just write this as $O(\log N)$�(log �). This is significantly better than an $O(N)$�(�) algorithm.

Sorting Algorithms

Binary search is great, but it requires your array be already sorted. So, how do we sort an array? This is a big question, and there has been a lot of work in the field of Computer Science to figure out the best ways to sort. Let's go over some common algorithms along with their efficiencies.

Please note that there is an amazing website called visualgo that has nice illustrations of most of these algorithms. As you go through these notes, visit it to watch the algorithms in action and improve your understanding. The notes here are not sufficient to properly understand the algorithms, you should use the visualization as well.

Bubble Sort

Bubble sort is an easy algorithm based on going through the array and swapping pairs of numbers. The algorithm works as follows:

Algorithm

- Starting with the first element, compare it to the second element. If they are out of order, swap them.
- Compare the 2nd and 3rd element in the same way. Then the 3rd and 4th, etc. (So, go through the array once doing pair-wise swaps.)

- Repeat this N times, where N is the number of elements in the array.

There are a few observations you can make that can speed this up a bit:

1. After the first pass, the largest element is at the end. After the second pass, the second largest element is second to the end, etc. (This means each pass you make doesn't actually need to go through the entire array, later passes can stop earlier.)
2. If you make a pass and don't perform any swaps, then the array is sorted.

Sample Implementation

```
public void bubbleSort(int[] a) {
   int n = a.length;
   boolean didSwap = true;

   // Keep going until we didn't do any swaps during a pass
```

```
    while (didSwap) {
        didSwap = false;
        // Make a pass, swapping pairs if needed.
        for (int i = 0; i < n - 1; i++) {
            if (a[i] < a[i + 1]) {
                didSwap = true;
                swap(a, i, i + 1);
            }
        }
        // The previous loop puts the largest value at the end,
        // so we don't need to check that one again.
        n--;
    }
}
```

Efficiency

In the worst case (even doing the optimizations mentioned above) the first pass does (N-1) compares and swaps. The second pass does (N-2) compares and swaps, etc. This happens N times. This makes the efficiency $O(N^2)$�(�2).

Insertion Sort

Insertion sort is the type of sorting most people do if you give them a set of 7 playing cards, out of order, and ask them to sort them in their hand.

Algorithm

In each iteration, i, take the element in the ith position, compare it to the one before until you find the place where it belongs. (In other words, while it is less than the one behind it, keep moving it backwards.)

Sample Implementation

```
public static void insertionSort(int[] a) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        int index = i;
        int valueToInsert = a[index];
        while (index > 0 && valueToInsert < a[index - 1]) {
            a[index] = a[index - 1];
            index--;
        }
        a[index] = valueToInsert;
    }
}
```

Efficiency

The overall algorithm “inserts” all N items individually, and each insertion takes N compares in the worst case. This makes the efficiency $O(N^2)$�(�2). However, in practice, it is typically faster than bubble sort despite having the same big-o efficiency.

Selection Sort

Selection sort if based on the idea of “selecting” the elements in sorted order. You search the array for the smallest element and move it to the front. Then, you find the next smallest element, and put it next. Etc.

Algorithm

In each iteration, i, selection the i^{th} smallest element and store it in the i^{th} position. After completing N iterations, the array is sorted.

Sample Implementation

```
public static void selectionSort(int[] a) {
    int n = a.length;
    for (int i = 0; i < n - 1; i++) {
        int indexOfSmallest = i;
        for (int j = i + 1; j < n; j++)
            if (a[j] < a[indexOfSmallest])
                indexOfSmallest = j;
        swap(a, i, indexOfSmallest);
    }
}
```

Efficiency

There are N iterations of the algorithm, and each iteration needs to find the smallest element. Finding the smallest element is $O(N)$�(�), and doing an $O(N)$�(�) operating N times results in an effiency of $O(N^2)$�(�2).

3.4. Merge Sort

Merge sort is a recursive algorithm that involves splitting and merging the array.

Algorithm

The algorithm works as follows:

1. Divide the array in half.
2. Recursively sort both halves.

3. Merge the halves back together.

Sample Implementation

```
public static void mergeSort(int[] a) {
    if (a.length > 1) {
        int mid = a.length / 2; // first index of right half
        int[] left = new int[mid];
        for (int i = 0; i < left.length; i++) // create left subarray
            left[i] = a[i];
        int[] right = new int[a.length - mid];
        for (int i = 0; i < right.length; i++) // create right subarray
            right[i] = a[mid + i];
        mergeSort(left); // recursively sort the left half
        mergeSort(right); // recursively sort the right half
        merge(left, right, a); // merge the left and right parts back into a whole
    }
}

public static void merge(int[] left, int[] right, int[] arr) {
    int leftIndex = 0;
    int rightIndex = 0;
    for (int i = 0; i < arr.length; i++) {
        if (rightIndex == right.length || (leftIndex < left.length && left[leftIndex] < right[rightIndex])) {
            arr[i] = left[leftIndex];
            leftIndex++;
        } else {
            arr[i] = right[rightIndex];
            rightIndex++;
        }
    }
}
```

Efficiency

The process of merging two sorted lists together is $O(N)$�(�). Merge sort, effectively, does this $O(\log N)$�(log �) times. (Because it splits the list in half at every step.) So, that makes the overall efficiency $O(N \log N)$�(� log �).

For a more detailed understanding of where the $O(\log N)$�(log �) comes from, check out this detailed explanation from Khan Academy.

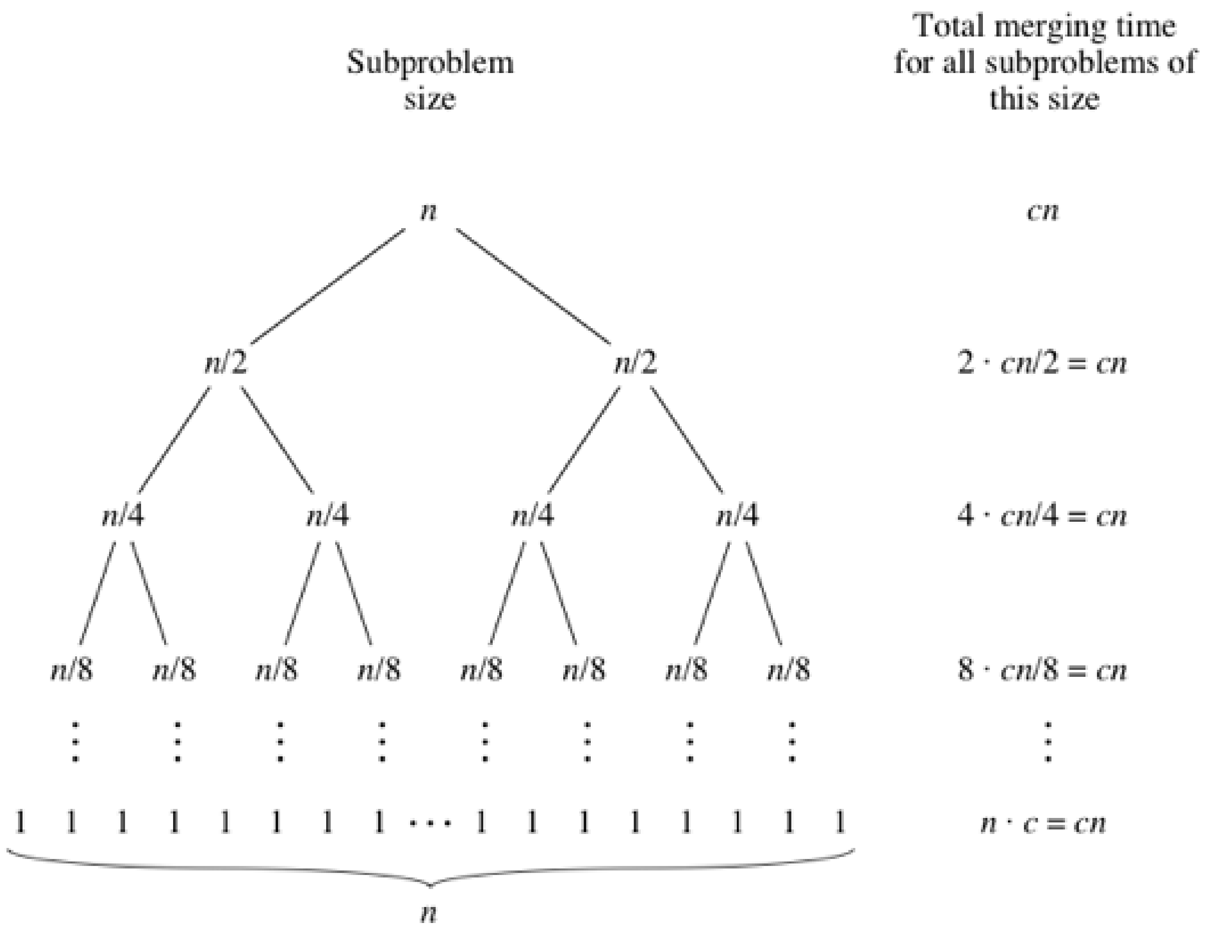


Image license: CC-BY-NC-SA. Courtesy of [Khan Academy]

Quick Sort

Quick sort is an interesting algorithm because while its worst case is technically $O(N^2)$�(�2), in practice it is almost always $O(N \log N)$�(� log �).

Quick sort is a recursive algorithm based around the idea of choosing a pivot item and sorting around it.

Algorithm

1. “Randomly” choose an element from the array as your pivot.

2. Partition the array around your pivot, making sure that items less than the pivot are to the left of it and all items greater than or equal to the pivot are to the right of it.
3. Recursively sort both parts.

Sample Implementation

```java
public static void quickSort(int[] a) {
    quickSort(a, 0, a.length - 1);
}

public static void quickSort(int[] a, int low, int high) {
    if (low < high) {
        int pivotIndex = partition(a, low, high); // value at pivotIndex will be in correct spot
        quickSort(a, low, pivotIndex - 1); // recursively sort the items to the left (< pivot)
        quickSort(a, pivotIndex + 1, high); // recursively sort the items to the right (>= pivot)
    }
}

public int partition(int[] arr, int low, int high) {
    int pivot = arr[low]; // select the first value as the pivot value (there are better ways to do this!)
    int boundaryIndex = low + 1; // the index of the first place (left-most) to put a value < pivot
    for (int i = low; i < high; i++) {
        if (arr[i] < pivot) {
            if (i != boundaryIndex) { // if it is ==, no swap needed since it's on correct side of partition
                swap(arr, i, boundaryIndex);
            }
            boundaryIndex++; // must always bump boundary if a smaller element found at arr[i]
        }
    }
    /*
     * boundaryIndex is the first (left-most) index of values >= pivot, because all
     * elements to the left of boundaryIndex are < pivot so we'll swap the pivot (at
     * index low) with the value at boundaryIndex -1, so pivot is placed correctly
     */
    swap(arr, low, boundaryIndex - 1);
    return boundaryIndex - 1;
}
```

Efficiency

Analyzing the efficiency here is somewhat complicated by the choice of pivot. If the pivot is perfectly chosen each time, then the array is effectively split in half at each step. That would make the efficiency $O(N \log N)$�(� log �) for the same reason that merge sort is. However, in the worst case the pivot isn't in the middle, instead it is at one of the ends. That would make the efficiency $O(N^2)$�(�2).

So, which is it? Technically, the worst case complexity is $O(N^2)$�(�2). However, in practice with a randomly chosen pivot each time, it is usually $O(N \log N)$�(� log �). It is also usually faster than merge sort, because the partitioning process is faster than merging.

Definition: Stable Sorts

A stable sort is one where, after sorting, the relative position of equal elements remains the same. Why is this useful? Let’s imagine you are sorting students and you want a list of students sorted by gender, but within gender you want them sorted by name. If you are using a stable sort, then you first sort the entire list by name and then sort the entire list by gender. The final list will have the ordering you want.

Bubble, insertion and merge sort are stable sorts. Selection and quick sort are not stable sorts.