Fall 2024 – CC-213 Data Structure and Algorithms
Week 03 – Abstract Data Types, Arrays, Polynomials and Sparse Matrix

1 Abstract Data Types (ADTs)

- ADTs are a fundamental concept in computer science and are widely used in the study of data structures and algorithms. An Abstract Data Type is a mathematical model that defines a data structure by specifying the *operations* that can be performed on it and the *behavior* of these operations, independent of their implementation. In other words, ADTs focus on *what* a data structure should do rather than *how* it should be implemented.
- **Definition**: An Abstract Data Type is a specification of a data type that describes its logical behavior, the operations that can be performed on it, and the type of values these operations produce. It does not specify how these operations are implemented.
- **Purpose**: ADTs are used to separate the concept of *what* a data structure should do from the *how* part (implementation). This abstraction allows programmers to focus on designing efficient algorithms and system behavior without being bogged down by implementation details.
- **Key Characteristics of ADTs**
    - **Abstraction of Implementation:** ADTs do not concern themselves with the underlying implementation. For instance, a Stack ADT specifies that elements can be pushed and popped, but whether it is implemented using an array or a linked list is not specified.
    - **Encapsulation:** ADTs encapsulate the data and operations together, hiding the details of implementation from the user.
    - **Modularity:** ADTs enable modular programming, making it easy to change or replace the underlying data structure without affecting the rest of the code.
- **Examples of Common ADTs:** Some of the most common ADTs are:
    - 1D Array (One-Dimensional Array)
    - 2D Array (Two-Dimensional Array)
    - ND Array (N-Dimensional Array)
    - Polynomial
    - Sparse Matrix
    - Stack
    - Queue
    - List
- **1D Array (One-Dimensional Array)**
    - **Definition**: A 1D array is a linear collection of elements of the same type, stored in contiguous memory locations. Each element can be accessed using a unique index.
    - **Operations:**
        - Access(i): Access the element at index i.
        - Insert(i, value): Insert a value at index i.
        - Delete(i): Remove the element at index i.
        - Search(value): Search for a value in the array.
    - **Use Case Example**: Suppose we have an array of temperatures for 7 days in a week: float temperatures[7] = {29.5, 30.1, 28.8, 29.0, 27.5, 28.2, 29.8}; Here, each day's temperature is stored in a specific position, and we can access any day's temperature using its index.
- **2D Array (Two-Dimensional Array)**

- **Definition**: A 2D array is a matrix-like structure where elements are stored in rows and columns. Each element is identified by two indices, one for the row and one for the column.
- **Operations:**
  - Access(i, j): Access the element at row i and column j.
  - Insert(i, j, value): Insert a value at position (i, j).
  - Delete(i, j): Delete the element at position (i, j).
  - Traverse(): Visit each element in the matrix.
- **Use Case Example**: Consider a 2D array representing a chessboard: char chessboard[8][8]; Here, the array stores the state of each square in an 8x8 grid, with chessboard[i][j] giving the piece (or empty square) at the i-th row and j-th column.

- **ND Array (Multi-Dimensional Array)**
  - **Definition**: An ND array is an extension of the 1D and 2D arrays to n dimensions, where each element is indexed by n values. It is useful for representing complex data structures such as tensors in machine learning or multi-level game boards.
  - **Operations:**
    - Access(i1, i2, ..., in): Access the element using n indices.
    - Insert(i1, i2, ..., in, value): Insert a value at the specified position.
    - Delete(i1, i2, ..., in): Delete the element at the specified position.
    - Traverse(): Visit each element in the array.
  - **Use Case Example**: Consider a 3D array representing a 3x3x3 Rubik's cube: char rubiksCube[3][3][3]; Here, rubiksCube[1][2][0] would give the value of a specific cubelet on the Rubik's cube.
- **Polynomial ADT**
  - **Definition**: A polynomial ADT represents a mathematical polynomial of the form: $P(x)=a_n x^n+a_{n-1}x^{n-1}+...+a_1 x+a_0$, where each a_i is a coefficient and n is the degree of the polynomial.
  - **Operations:**
    - Add(P1, P2): Adds two polynomials.
    - Multiply(P1, P2): Multiplies two polynomials.
    - Evaluate(P, x): Evaluates the polynomial P at a given value of x.
    - Display(P): Displays the polynomial in standard form.
  - **Use Case Example**: Consider the polynomial: $P(x)=3x^3+5x^2-2x+7$. The polynomial can be stored as a list of (coefficient, exponent) pairs: int poly[4][2] = {{3, 3}, {5, 2}, {-2, 1}, {7, 0}}; Here, poly is a two dimensional array of polynomial terms including their coefficient and power, and operations like addition or multiplication can be defined to manipulate these terms.
- **Sparse Matrix ADT**
  - **Definition**: A sparse matrix is a 2D matrix in which most of the elements are zero. It is represented in a compact form by storing only the non-zero elements along with their positions.
  - **Operations**:
    - Insert(i, j, value): Insert a non-zero value at position (i, j).
    - Delete(i, j): Delete a non-zero value from position (i, j).
    - Add(M1, M2): Adds two sparse matrices.
    - Multiply(M1, M2): Multiplies two sparse matrices.
    - Transpose(M): Transposes a sparse matrix.
  - **Use Case Example**: Suppose we have the following sparse matrix:

0	0	3	0
22	0	0	0
0	0	0	9
0	5	0	0

  - Instead of storing the entire 4x4 matrix, we can store only the non-zero elements as: row, col and value. int SparseMatrix [4][3] = {{0, 2, 3}, {1, 0, 22}, {2, 3, 9}, {3, 1, 5}}; Here, sparse is two-dimensional array of non-zero elements with their row and column indices.
- **Stack ADT**
  - **Description**: A stack is a linear data structure that follows the *Last-In-First-Out (LIFO)* principle. The main operations for a stack are:
    - push(element): Adds an element to the top of the stack.
    - pop(): Removes and returns the top element.
    - peek(): Returns the top element without removing it.
    - isEmpty(): Checks if the stack is empty.
  - **Use Case Example**: Suppose you are building a text editor with an *undo* feature. Each action (typing, deleting, etc.) is pushed onto a stack. When the user presses "Undo," the most recent action is popped from the stack, effectively reversing the last change.
- **Queue ADT**
  - **Description**: A queue is a linear data structure that follows the *First-In-First-Out (FIFO)* principle. The main operations for a queue are:
    - enqueue(element): Adds an element to the end of the queue.
    - dequeue(): Removes and returns the front element.
    - isEmpty(): Checks if the queue is empty.
  - **Use Case Example**: Consider a printer queue. Jobs sent to the printer are processed in the order they are received. The first job to be added (enqueued) is the first one to be printed (dequeued).
- **List ADT**
  - **Description**: A list is a collection of elements with the following operations:
    - insert(position, element): Inserts an element at a specified position.
    - delete(position): Deletes the element at a given position.
    - get(position): Returns the element at the specified position.
    - size(): Returns the number of elements in the list.
  - **Use Case Example**: Suppose you are developing a music player with a playlist feature. Each song is stored as an element in a list. Users can add a song at a specific position, delete a song, or retrieve a song to play.
- **Difference between ADTs and Data Structures**
  - **ADTs** define the *behavior* and *operations* of a data type. For example, a *Queue* ADT specifies operations like enqueue and dequeue but does not define whether these operations should be implemented using an array or a linked list.
  - **Data Structures**, on the other hand, are concrete implementations of these ADTs. For example, a *Queue* can be implemented using an array or a linked list. The *data structure* provides the physical representation and internal storage mechanisms.
- **Example: Stack ADT vs. Stack Data Structure**
  - **Stack ADT**: Specifies the operations like push, pop, and peek, along with the LIFO behavior. The underlying implementation is not defined.
  - **Stack Data Structure**: A *stack* can be implemented using:
    - An **array-based stack** where the elements are stored in an array.

    - A **linked list-based stack** where nodes point to the next node.
- **Benefits of Using ADTs**
    - **Abstraction**: ADTs provide a level of abstraction that hides implementation details, allowing programmers to focus on problem-solving.
    - **Modularity**: Different implementations can be swapped without altering the overall system, leading to easier maintenance and upgrades.
    - **Reusability**: ADTs can be reused across different programs since they define a clear interface and behavior.
- **Conclusion:** Understanding Abstract Data Types (ADTs) is crucial when designing algorithms and software systems. They provide a way to conceptualize data and operations while allowing flexibility in choosing appropriate implementations. By focusing on the *what* rather than the *how*, ADTs enable efficient problem-solving and code design, forming a bridge between theory and practical programming.

2 Naive Implementations of 1D, 2D, and ND Dynamic Arrays in C++ with Examples

- Dynamic arrays in C++ are a versatile and powerful tool for managing collections of data that can change in size during the execution of a program. Unlike static arrays, which require the size to be known at compile time, dynamic arrays allow for flexible memory management and are commonly implemented using pointers and dynamic memory allocation. This flexibility is achieved using functions like new and delete, which provide control over the allocation and deallocation of memory. Below is a detailed explanation of implementing dynamic arrays in C++ for 1D, 2D, and ND (multi-dimensional) arrays.
- **1D Dynamic Array Implementation**
    - A 1D dynamic array is essentially a linear collection of elements. You can implement a 1D dynamic array using pointers and dynamic memory allocation using the new operator in C++.
    - **Example Implementation:**

```
#include <iostream>
using namespace std;

int main() {
   int size;
   cout << "Enter the size of the array: ";
   cin >> size;

   // Step 1: Allocate memory for the array dynamically
   int *arr = new int[size];

   // Step 2: Input elements into the array
   cout << "Enter " << size << " elements:\n";
   for (int i = 0; i < size; i++) {
      cin >> arr[i];
   }

   // Step 3: Display the elements of the array
   cout << "The array elements are:\n";
   for (int i = 0; i < size; i++) {
      cout << arr[i] << " ";
```

```
    }
    cout << endl;

    // Step 4: Deallocate the memory after usage
    delete[] arr;

    return 0;
}
```

- **Explanation:**
  - int *arr = new int[size]; dynamically allocates memory for an array of integers.
  - delete[] arr; frees the allocated memory after use, preventing memory leaks.
  - This implementation allows the array size to be set at runtime and makes it easy to resize the array if needed.

- **2D Dynamic Array Implementation**
  - A 2D dynamic array is a matrix-like structure where elements are stored in rows and columns. You can implement it by using an array of pointers, each pointing to a dynamically allocated row.
  - **Example Implementation:**

```
#include <iostream>
using namespace std;

int main() {
    int rows, cols;
    cout << "Enter the number of rows: ";
    cin >> rows;
    cout << "Enter the number of columns: ";
    cin >> cols;

    // Step 1: Allocate memory for an array of pointers
    int **arr = new int *[rows];

    // Step 2: Allocate memory for each row
    for (int i = 0; i < rows; i++) {
        arr[i] = new int[cols];
    }

    // Step 3: Input elements into the 2D array
    cout << "Enter the elements of the array:\n";
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cin >> arr[i][j];
        }
    }

    // Step 4: Display the 2D array
    cout << "The 2D array elements are:\n";
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
```

```
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }

    // Step 5: Deallocate the memory for each row
    for (int i = 0; i < rows; i++) {
        delete[] arr[i];
    }

    // Step 6: Deallocate the memory for the array of pointers
    delete[] arr;

    return 0;
}
```

- **Explanation:**
  - int **arr = new int *[rows]; creates an array of pointers, each pointing to a row of the matrix.
  - Each row is then allocated using arr[i] = new int[cols];.
  - This structure allows for dynamic allocation of rows and columns, providing flexibility in size and shape.
  - Memory is freed in two steps: first for each row and then for the array of pointers.

- **ND Dynamic Array Implementation**
  - An ND (multi-dimensional) dynamic array is an extension of the 1D and 2D arrays to n dimensions. Implementing a dynamic ND array involves creating a hierarchical structure where each element points to a lower-level array until the base dimension is reached.
  - **3D Array Example Implementation:**

```
#include <iostream>
using namespace std;

int main() {
    int x, y, z;
    cout << "Enter dimensions for the 3D array (x, y, z): ";
    cin >> x >> y >> z;

    // Step 1: Allocate memory for a 3D array (an array of pointers to 2D arrays)
    int ***arr = new int **[x];

    // Step 2: Allocate memory for each 2D array (array of pointers to 1D arrays)
    for (int i = 0; i < x; i++) {
        arr[i] = new int *[y];
        for (int j = 0; j < y; j++) {
            // Step 3: Allocate memory for each 1D array
            arr[i][j] = new int[z];
        }
    }
```

```
    // Step 4: Input elements into the 3D array
    cout << "Enter the elements of the 3D array:\n";
    for (int i = 0; i < x; i++) {
        for (int j = 0; j < y; j++) {
            for (int k = 0; k < z; k++) {
                cin >> arr[i][j][k];
            }
        }
    }

    // Step 5: Display the 3D array elements
    cout << "The 3D array elements are:\n";
    for (int i = 0; i < x; i++) {
        for (int j = 0; j < y; j++) {
            for (int k = 0; k < z; k++) {
                cout << arr[i][j][k] << " ";
            }
            cout << endl;
        }
        cout << endl;
    }

    // Step 6: Deallocate the memory for each 1D array
    for (int i = 0; i < x; i++) {
        for (int j = 0; j < y; j++) {
            delete[] arr[i][j];
        }
    }

    // Step 7: Deallocate the memory for each 2D array
    for (int i = 0; i < x; i++) {
        delete[] arr[i];
    }

    // Step 8: Deallocate the memory for the 3D array of pointers
    delete[] arr;

    return 0;
}
```

- **Explanation:**
  - int ***arr = new int **[x]; creates a pointer to a 2D array.
  - Each 2D array is then created by arr[i] = new int *[y];, and each 1D array within it is created by arr[i][j] = new int[z];.
  - This approach allows for the creation of dynamic arrays for any number of dimensions.
  - Memory is deallocated in reverse order, starting from the innermost array.
- **Summary of Dynamic Array Implementations in C++:**

Array Type	Memory Allocation	Access Pattern	Memory Deallocation
1D Array	int *arr = new int[size];	arr[i]	delete[] arr;
2D Array	int **arr = new int *[rows];	arr[i][j]	delete[] arr[i]; delete[] arr;
ND Array	int ***arr = new int **[x];	arr[i][j][k]	delete[] arr[i][j]; ...

  - **Conclusion:** Dynamic arrays are powerful data structures that can be implemented for different dimensions using pointer-based memory management in C++. Implementing 1D, 2D, and ND dynamic arrays requires a good understanding of pointers, memory allocation, and deallocation to avoid memory leaks. The flexibility of dynamic arrays makes them ideal for situations where the size or shape of the array is not known at compile time.
- **Time and Space Complexity Analysis of 1D, 2D, and ND Dynamic Arrays in C++**
  - The performance of dynamic arrays in terms of time and space complexity depends on their implementation and operations performed on them. Below, we analyze and compare the time and space complexities of 1D, 2D, and ND dynamic arrays based on the following factors:
    - Memory Allocation and Deallocation
    - Access Time for Elements
    - Insertion and Deletion of Elements
    - Space Complexity
- **1D Dynamic Array**
  - **1D** Dynamic **Array Implementation:**
    int *arr = new int[size];
  - **Time Complexity Analysis**
    - **Memory Allocation:** The new operator allocates contiguous memory for size elements. The allocation time complexity is O(1) since it allocates a block of memory in a single step.
    - **Accessing Elements: Accessing** an element in a 1D array by index is O(1) because the memory is contiguous, and element arr[i] can be accessed directly using pointer arithmetic.
    - **Insertion and Deletion:**
      - Inserting an element at the end: O(1).
      - Inserting or deleting an element at a specific index: O(n) due to shifting of elements.
  - **Space Complexity Analysis**
    - **Space Complexity:** The space required is proportional to the size of the array: O(n), where n is the number of elements.
- **2D Dynamic Array**
  - **2D Dynamic Array Implementation:**
    ```
    int **arr = new int *[rows];
    for (int i = 0; i < rows; i++) {
      arr[i] = new int[cols];
    }
    ```
  - **Time Complexity Analysis**
    - **Memory Allocation:**
      - Allocating the array of pointers (int **arr) for rows: O(1).
      - Allocating each row individually using arr[i] = new int[cols]: O(rows).

        - Total time complexity for memory allocation is O(rows).
    - **Accessing Elements:**
        - Accessing an element arr[i][j] requires two levels of pointer dereferencing: O(1).
    - **Insertion and Deletion:**
        - Inserting or deleting an entire row or column involves copying pointers and data: O(n), where n is the number of elements in the array.
        - Inserting or deleting a single element in a row/column: O(cols).
- **Space Complexity Analysis**
    - **Space Complexity:** The space required is the sum of space for pointers and the space for elements:
        - Pointers: O(rows).
        - Elements: O(rows×cols).
        - Total space complexity: O(rows×cols).

- **ND Dynamic Array**
    - **ND Dynamic Array Implementation (Example for 3D Array):**

```
int ***arr = new int **[x];
for (int i = 0; i < x; i++) {
  arr[i] = new int *[y];
  for (int j = 0; j < y; j++) {
    arr[i][j] = new int[z];
  }
}
```

    - **Time Complexity Analysis**
        - **Memory Allocation:**
            - Allocating pointers for 3D array int ***arr: O(1).
            - Allocating each 2D array for x layers: O(x).
            - Allocating each 1D array for y rows: O(x×y).
            - Total time complexity: O(x×y).
        - **Accessing Elements:**
            - Accessing an element arr[i][j][k] involves three pointer dereferencing steps: O(1).
        - **Insertion and Deletion:**
            - Inserting or deleting an entire layer/row/column involves significant memory shifts: O(n), where n is the number of elements.
    - **Space Complexity Analysis**
        - **Space Complexity:**
            - For a 3D array with **dimensions** x, y, and z:
            - Pointers for x 2D arrays: O(x).
            - Pointers for x times y rows: O(x×y).
            - Space for x times y times z elements: O(x×y×z).
            - Total space complexity: O(x×y×z).
    - **Summary and Comparison of Complexities**

Array Type	Time Complexity for Memory Allocation	Time Complexity for Accessing Elements	Space Complexity
1D Dynamic Array	O(1)	O(1)	O(n)
2D Dynamic Array	O(rows)	O(1)	O(rows×cols)

Array Type	Time Complexity for Memory Allocation	Time Complexity for Accessing Elements	Space Complexity
ND Dynamic Array	O(x×y)	O(1)	O(x×y×z)

- **Example Analysis:** Let's take an example to illustrate the time and space complexities for different array sizes:
  - **1D Array:** If size = 1000, the space complexity is O(1000).
  - **2D Array:** If rows = 10 and cols = 100, the space complexity is O(1000).
  - **3D Array:** If x = 10, y = 10, and z = 10, the space complexity is O(1000).
- In all these cases, the space complexity is linear with respect to the total number of elements. However, the time complexity for operations like insertion and deletion varies based on the dimensionality and the required shifts in memory.

- **Conclusion**
  - Dynamic arrays offer flexibility but come with different performance characteristics depending on their dimensionality. While 1D arrays provide the fastest access and simplest memory management, multi-dimensional arrays require more complex handling and more memory for storing pointers, leading to higher time and space complexities. Efficient management of these resources is essential, especially for large-scale data structures.

- **Space Complexity Analysis: User Data vs. Control Data**
  - When analyzing the space complexity of dynamic arrays, it's crucial to differentiate between **user data** (actual values that the array is storing) and **control data** (pointers and auxiliary variables required to manage these values). This distinction helps in understanding the memory overhead associated with different dimensionalities and gives a clearer picture of how memory is utilized. Here's how the space complexity breaks down into user and control data for 1D, 2D, and ND arrays in C++:
  - **1D Dynamic Array**
    - **Implementation:**
      - `int *arr = new int[size];`
    - **User Data:**
      - The array stores size integer values.
      - **Memory for User Data:** size * sizeof(int)
        - For example, if size = 1000 and sizeof(int) = 4 bytes, then:
        - Total memory for user data = 1000×4 = **4000 bytes**.
    - **Control Data:**
      - A single pointer (int *arr) to manage the base address of the array.
      - **Memory for Control Data:** sizeof(int*)
        - If sizeof(int*) = 8 bytes on a 64-bit machine, then:
        - Total memory for control data = **8 bytes**.
    - **Total Space Complexity:**
      - User Data: O(n) bytes
      - Control Data: O(1) bytes
  - **2D Dynamic Array**
    - **Implementation:**

```
int **arr = new int *[rows];
for (int i = 0; i < rows; i++) {
  arr[i] = new int[cols];
```

```
}
```

- **User Data:**
  - The array stores rows × cols integer values.
  - **Memory for User Data:** rows × cols × sizeof(int)
    - For example, if rows = 10 and cols = 100, and sizeof(int) = 4 bytes, then:
    - Total memory for user data = 10×100×4 = **4000 bytes**.
- **Control Data:**
  - An array of rows pointers (int **arr) pointing to each row.
  - Each row pointer is sizeof(int*).
  - Additionally, for each row, there is a pointer to manage cols integers.
- **Memory for Control Data:**
  - Base pointer (int **arr): sizeof(int*)
  - Array of pointers for rows: rows × sizeof(int*)
  - If sizeof(int*) = 8 bytes and rows = 10, then:
  - Base pointer = 8 bytes.
  - Array of row pointers = 10×8bytes.
  - Total control data = 8+80=88 bytes.
- **Total Space Complexity:**
  - User Data: O(rows × cols) bytes
  - Control Data: O(rows) bytes

- **ND Dynamic Array (Example for 3D Array)**
  - **Implementation (3D Array):**

```
int ***arr = new int **[x];
for (int i = 0; i < x; i++) {
  arr[i] = new int *[y];
  for (int j = 0; j < y; j++) {
    arr[i][j] = new int[z];
  }
}
```

  - **User Data:**
    - The array stores x × y × z integer values.
    - **Memory for User Data:** x × y × z × sizeof(int)
      - For example, if x = 10, y = 10, z = 10, and sizeof(int) = 4 bytes, then:
      - Total memory for user data = 10×10×10×4 = **4000 bytes**.
  - **Control Data:**
    - A 3D array requires:
    - A base pointer (int ***arr).
    - An array of x pointers (int **), each pointing to a 2D array.
    - Each 2D array has x pointers, each pointing to y arrays (int *).
    - **Memory for Control Data:**
      - Base pointer: sizeof(int***)
      - Array of x pointers: x × sizeof(int**)
      - Array of x × y pointers for rows: x × y × sizeof(int*)
      - If sizeof(int*) = 8 bytes and x = 10, y = 10, then:
      - Base pointer: 8 bytes.
      - Array of x pointers: 10×8=8010 \times 8 = 8010×8=80 bytes.

        - Array of x × y pointers: 10×10×8=800 bytes.
        - Total control data = 8+80+800=888 bytes.
    - **Total Space Complexity:**
        - User Data: O(x × y × z) bytes
        - Control Data: O(x × y) bytes
- **Summary Table for User Data vs. Control Data**

Array Type	Dimensions	User Data	Control Data
1D Dynamic Array	size	size × sizeof(int)	sizeof(int*)
2D Dynamic Array	rows × cols	rows × cols × sizeof(int)	sizeof(int**) + rows × sizeof(int*)
3D Dynamic Array	x × y × z	x × y × z × sizeof(int)	sizeof(int***) + x × sizeof(int**) + x × y × sizeof(int*)

    - **Example Comparison**
        - Let's summarize with concrete values for different array sizes using sizeof(int) = 4 bytes and sizeof(int*) = 8 bytes:
    - **1D Array with size = 1000:**
        - User Data: 1000 × 4 = 4000 bytes
        - Control Data: 8 bytes
    - **2D Array with rows = 10 and cols = 100:**
        - User Data: 10 × 100 × 4 = 4000 bytes
        - Control Data: 8 + (10 × 8) = 88 bytes
    - **3D Array with x = 10, y = 10, z = 10:**
        - User Data: 10 × 10 × 10 × 4 = 4000 bytes
        - Control Data: 8 + (10 × 8) + (10 × 10 × 8) = 888 bytes
- **Conclusion**
    - **User Data:** Increases linearly with the total number of elements in the array.
    - **Control Data:** Increases with the number of pointers required to manage the array structure, growing exponentially as the dimensions increase (e.g., 1D, 2D, 3D).
    - Understanding this differentiation helps optimize both space and time when dealing with complex data structures, especially for large-scale applications.

3 Memory Representation of 2D, 3D, and ND Arrays: Row Major and Column Major Orders

- When working with multi-dimensional arrays in C++, understanding memory representation is essential for efficient data access and manipulation. Multi-dimensional arrays are typically stored in contiguous memory, either in **Row Major Order** or **Column Major Order**, which determines the arrangement of the array elements in linear memory. Let's explore this concept in depth, along with the mapping formulas to access elements for 2D, 3D, and ND arrays.
- **Row Major Order vs. Column Major Order**
    - **Row Major Order**:
        - Stores elements of a row in contiguous memory locations.
        - The array is stored row-by-row, meaning that all elements of the first row are stored first, followed by all elements of the second row, and so on.
        - Commonly used in C/C++.
    - **Column Major Order**:
        - Stores elements of a column in contiguous memory locations.

        - The array is stored column-by-column, meaning that all elements of the first column are stored first, followed by all elements of the second column, and so on.
        - Used by languages like Fortran and MATLAB.
- **Memory Representation of a 2D Array**
    - Let's take a 2D array with dimensions rows = m and cols = n. The element at position arr[i][j] needs to be mapped into a 1D representation.
        - **Row Major Order Mapping Formula**:
            - Address(i,j)=Base Address+(i×n+j)×Size of Data Type
            - Where: i = Row index, j = Column index, n = Number of columns
        - **Column Major Order Mapping Formula**:
            - Address(i,j)=Base Address+(j×m+i)×Size of Data Type
            - Where: i = Row index, j = Column index, m = Number of rows
    - **Example:** Consider a 2D array:

```
int arr[3][4] = {
  {1, 2, 3, 4},
  {5, 6, 7, 8},
  {9, 10, 11, 12}
};
```

        - The base address is assumed to be 1000 and sizeof(int) = 4 bytes.
        - **Row Major Order**: arr[2][3] (element = 12)
            - Address(2,3)=1000+(2×4+3)×4=1044
        - **Column Major Order**: arr[2][3] (element = 12)
            - Address(2,3)=1000+(3×3+2)×4=1044
- **Memory Representation of a 3D Array**
    - For a 3D array with dimensions x = p, y = q, and z = r, the element at position arr[i][j][k] needs to be mapped into a 1D representation.
    - **Row Major Order Mapping Formula**:
        - Address(i,j,k)=Base Address+(i×(q×r)+j×r+k)×Size of Data Type
        - Where: i = 1st Dimension index, j = 2nd Dimension index, k = 3rd Dimension index, q = Number of rows in the 2D matrix (y-axis size), r = Number of columns in the 2D matrix (z-axis size)
    - **Column Major Order Mapping Formula**:
        - Address(i,j,k)=Base Address+(k×(p×q)+j×p+i)×Size of Data Type
- **Memory Representation of an ND Array**
    - For an ND array with dimensions D1, D2, D3, ..., Dn, the element at position $arr[i_1][i_2][i_3]\ldots[i_n]$ needs to be mapped into a 1D representation.
    - **Row Major Order Mapping Formula**:
        - $Address(i_1,i_2,i_3,\ldots,i_n)=Base\ Address+(\sum k=1\ to\ n\ (i_k \times \prod j=k+1\ to\ n\ D_j))\times Size\ of\ Data\ Type$
        - Where: $i_1, i_2, i_3, \ldots, i_n$ = Indices along each dimension and $D_1, D_2, \ldots, D_n$ = Size of each dimension respectively
    - **Column Major Order Mapping Formula**:
        - $Address(i_1,i_2,i_3,\ldots,i_n)=Base\ Address+(\sum k=1\ to\ n\ (i_k \times \prod j=1\ to\ k-1\ D_j))\times Size\ of\ Data\ Type$
- **Graphical Representation and Examples**
    - **2D Array in Row Major Order**:
      [1, 2, 3, 4]

[5, 6, 7, 8]
[9, 10, 11, 12]
Linear Memory Layout: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
- **2D Array in Column Major Order**:
Linear Memory Layout: 1, 5, 9, 2, 6, 10, 3, 7, 11, 4, 8, 12

- These formulas are fundamental for understanding how multi-dimensional arrays are stored and accessed in memory, which has direct implications for optimizing algorithms and understanding cache behavior.

4 Space Complexity Analysis of Control Data for Multi-dimensional Arrays using Naive Implementations and Row and Column major orders

- When dealing with multi-dimensional arrays, the space complexity of an array isn't just about the user data stored in the array elements. It also includes **control data**, which are pointers and other variables used to manage and access the elements efficiently. This is particularly important when we are dynamically allocating memory in C/C++ for 2D, 3D, or ND arrays. Let's analyze the space complexity for different array types using naïve implementation, row-major and column-major memory representations, considering both **user data** and **control data**.
- Control **Data in 1D Arrays**
  - A 1D array is straightforward in terms of memory management. The space complexity depends almost entirely on the size of the array, as there are no additional pointers for sub-arrays.
  - **User Data Space Complexity**: If an array has n elements, and each element requires d bytes, the total space is: $S_{user}=n\times d$
  - **Control Data Space Complexity**: For a 1D array, the only control data is a pointer to the first element: $S_{control}$=Size of Pointer. Thus, for most systems, this is typically 4 bytes (32-bit) or 8 bytes (64-bit).
- Control **Data in 2D Arrays**
  - 2D arrays can be stored in **contiguous** or **non-contiguous** memory. When implementing a **dynamically** allocated 2D array, control data plays a critical role in efficiently accessing each row. Let's consider two common methods of implementing a 2D array:
  - **a. Single Pointer (Contiguous Allocation)**
    - The entire 2D array is stored as a contiguous block in memory.
    - **Control Data**: Just one pointer pointing to the start of the entire 2D array. Scontrol=Size of PointerS_{control} = \text{Size of Pointer}Scontrol =Size of Pointer
    - **Example**: int *array = new int[m * n]; // 2D array with m rows and n columns
  - **b. Array of Pointers (Non-Contiguous Allocation)**
    - Each row of the 2D array is stored separately, and an array of pointers is used to point to each row.
  - **Control Data**:
    - m pointers (one for each row). $S_{control}=m\times$ Size of Pointer. Additionally, there is a single pointer to the array of pointers.
    - $S_{control\_total}=m\times$ Size of Pointer + Size of Pointer
    - **Example**:
      int **array = new int*[m]; // Create an array of pointers
      for (int i = 0; i < m; ++i)
      array[i] = new int[n]; // Allocate each row
- **Control Data in 3D** Arrays

- For a 3D array with dimensions p, q, and r, representing depth, rows, and columns respectively, the control data becomes more complex due to the need to manage layers, rows within layers, and elements within rows.
- **Single Pointer (Contiguous Allocation)**: Entire 3D array is stored in a contiguous block of memory.
  - **Control Data**: A single pointer: $S_{control}$=Size of Pointer
- **Array of Pointers for Each 2D Layer (Non-Contiguous Allocation)**:
  - p pointers for each layer (array of 2D arrays).
  - Each 2D array is represented by q pointers (for each row in a layer).
- **Total Control Data**: $S_{control\_total}$=p×q×Size of Pointer+p×Size of Pointer+Size of Pointer

- **Control** Data **in ND Arrays**
  - For an N-dimensional array with dimensions D_1, D_2, ..., D_n, the control data depends on the implementation method and how pointers are used to reference each dimension:
  - **General Formula**:
    - **User Data Space Complexity**:
      - S_{user}=∏i=1 to n (D_i×d)
    - **Control Data Space Complexity**: The control data grows exponentially with the number of dimensions, especially in non-contiguous allocations:
      - $S_{control}$=∑i=1 to n (D_i ×Size of Pointer)
    - **Example for a 4D Array:** Let's say a 4D array has dimensions D_1 = p, D_2 = q, D_3 = r, and D_4 = s. In a non-contiguous dynamic allocation:
      - p pointers point to 3D arrays.
      - Each 3D array has q pointers for 2D arrays.
      - Each 2D array has r pointers for 1D arrays.
    - **Total Control Data Space Complexity**:
      $S_{control\_total}$=(p×q×r×Size of Pointer)+(p×q×Size of Pointer)+(p×Size of Pointer)+Size of Pointer

- **5. Memory Efficiency Considerations**
  - When implementing multi-dimensional arrays, the following considerations help optimize memory usage:
  - **Contiguous vs. Non-Contiguous Allocation**:
    - Contiguous allocations have less control data but may cause issues with memory fragmentation.
    - Non-contiguous allocations require more control pointers, but each dimension can be managed independently.
  - **Access Time**:
    - Contiguous memory provides faster access due to cache locality, while non-contiguous memory may incur higher access time.
  - **Memory Overhead**:
    - Pointers can consume significant memory, especially for higher dimensions or large array sizes.
- **Space Complexity Summary of User and Control Data for Array Implementations**

Array Type	**User Data Space (d is size of data type)**	**Control Data Space (Total)**
1D Array	n×d	Size of Pointer
2D Array (Single Pointer)	m×n×d	Size of Pointer

Array Type	User Data Space (d is size of data type)	Control Data Space (Total)
2D Array (Array of Pointers)	m×n×d	m×Size of Pointer+Size of Pointer
3D Array (Single Pointer)	p×q×r×d	Size of Pointer
3D Array (Arrays of Pointers)	p×q×r×d	p×q×Size of Pointer+p×Size of Pointer +Size of Pointer
ND Array (Single Pointer)	$\prod i=1$ to n (D_i×d)	Size of Pointer
ND Array (Arrays of Pointers)	$\prod i=1$ to n (D_i×d)	$\sum i=1$ to n (D_i×Size of Pointer)

- Understanding and balancing between user data and control data space is crucial for creating memory-efficient multi-dimensional array implementations, especially in large-scale systems.

5 Polynomial Abstract Data Type (ADT)

- A **Polynomial** is a **mathematical** expression consisting of variables (also known as indeterminates), coefficients, and exponents, combined using addition, subtraction, and multiplication operations. The general form of a polynomial is given by:
  $P(x)=a_n x^n+a_{n-1} x^{n-1}+...+a_1 x^1+a_0$ Where:
  - $a_n, a_{n-1}, ..., a_1, a_0$ are the **coefficients** (real or integer values).
  - x is the **variable**.
  - $n, n-1, ..., 1, 0$ are the **exponents** or **degrees**.
- The **Polynomial ADT (Abstract Data Type)** is used to represent and manipulate polynomial expressions efficiently. It allows operations such as addition, subtraction, multiplication, differentiation, and evaluation of polynomials at a particular value of the variable.
- **Key Characteristics of Polynomial ADT**
  - **Coefficients and Exponents:**
    - Each term in the polynomial has a coefficient and an exponent.
    - Example: In the polynomial $4x^3+2x^2-5x+7$ the terms are $4x^3, 2x^2, -5x$, and 7.
    - The coefficients are 4, 2, -5, and 7.
    - The exponents are 3, 2, 1, and 0, respectively.
  - **Ordered by Exponents:**
    - Polynomials are usually stored in a manner where terms are arranged in descending order of exponents.
    - For instance, $4x^3+2x^2-5x+7$ is arranged in the order of powers of x.
  - **Sparse Representation:**
    - When the polynomial has many missing degrees, a sparse representation is used, where only non-zero coefficients and their corresponding exponents are stored.
    - Example: $6x^{100}+4x^5+2x^2$ can be stored as {(100,6),(5,4),(2,2)}, where each pair is in the form (exponent, coefficient).
- **Polynomial ADT Operations**
  - **Addition:**
    - Add **corresponding** coefficients of terms with the same exponents. Example: $(4x^3+2x^2) + (3x^3+5x) = 7x^3+2x^2+5x$.
  - **Subtraction:**
    - Subtract corresponding coefficients of terms with the same exponents. Example: $(4x^3+2x^2) - (3x^3+5x) = x^3+2x^2-5x$.
  - **Multiplication:**

        - Multiply each term of one polynomial by every term of the other, combining like terms. Example: $(x+2)(x^2-x+3) = x^3-x^2+3x+2x^2-2x+6 = x^3+x^2+x+6$.
    - **Evaluation:**
        - Substitute a particular value for the variable x and compute the value of the polynomial. Example: If $P(x)=3x^2+2x-5$, then $P(2)=3\times(2)^2+2\times2-5 = 3\times4+4-5 = 12+4-5 = 11$.
    - **Differentiation:**
        - Differentiate the polynomial to find its derivative. Example: $P(x)=3x^2+2x-5$ gives $P'(x)=6x+2$.
- **Implementing Polynomial ADT in C++**
    - Polynomials can be implemented using arrays, linked lists, or sparse matrices, depending on the nature of the polynomial and required operations. Let's explore two common methods:
    - An array is used where the index represents the exponent and the value at that index represents the coefficient. This approach is effective for **dense polynomials** with few missing terms.
    - **Example:** $3x^4+5x^3-6x+2$ can be stored as: $[2,0,-6,5,3]$ where index 0 corresponds to x^0, index 1 to x^1, index 2 to x^2, and so on.
    - **Drawback**: Wastes space if there are large gaps between degrees (e.g., $3x^{100}+2x^2$).

```
#include <iostream>
using namespace std;

// A simple polynomial class using an array representation
class Polynomial {
private:
  int* coefficients;
  int max_degree;

public:
  // Constructor
  Polynomial(int degree) {
    max_degree = degree;
    coefficients = new int[max_degree + 1] {0}; // Initialize all coefficients to 0
  }

  // Function to set the coefficient for a given degree
  void setCoefficient(int degree, int coefficient) {
    coefficients[degree] = coefficient;
  }

  // Display the polynomial
  void display() {
    for (int i = max_degree; i >= 0; i--) {
      if (coefficients[i] != 0) {
        cout << coefficients[i] << "x^" << i;
        if (i != 0) cout << " + ";
      }
    }
```

```
        cout << endl;
    }

    ~Polynomial() {
        delete[] coefficients;
    }
};

int main() {
    Polynomial poly(4); // Create a polynomial of degree 4
    poly.setCoefficient(4, 3); // Set coefficient of x^4 as 3
    poly.setCoefficient(3, 5); // Set coefficient of x^3 as 5
    poly.setCoefficient(1, -6); // Set coefficient of x as -6
    poly.setCoefficient(0, 2); // Set constant term as 2

    cout << "Polynomial: ";
    poly.display(); // Output: 3x^4 + 5x^3 - 6x^1 + 2x^0

    return 0;
}
```

- **Sparse Polynomial Representation**
  - In sparse polynomials, storing all coefficients in an array wastes space. Instead, only non-zero coefficients are stored along with their corresponding exponents, reducing memory usage significantly.
  - Example: Given $4x^{50}+3x^2-2$, store as in two dimensional array: {(50,4),(2,3),(0,−2)}
- **Applications of Polynomial ADT**
  - **Physics and Engineering Calculations**: Modeling curves, fitting data.
  - **Computer Graphics**: Representation of curves using Bézier polynomials.
  - **Machine Learning**: Polynomial regression for predicting values.
- Understanding the Polynomial ADT and its various implementations helps in selecting the right data structure for efficient computation, depending on the polynomial's sparsity and the nature of operations required.

6 Sparse Matrix Abstract Data Type (ADT)

- A **Sparse Matrix** is a matrix in which the majority of the elements are zero, typically having only a few non-zero elements. Mathematically, a matrix is considered sparse if more than half of its elements are zeros. The **Sparse Matrix ADT** is designed to store and efficiently manipulate sparse matrices by using compact data structures that only store the non-zero values and their positions. This reduces both memory usage and computational cost.
- For example, consider a 4x4 matrix:

  0 0 0 3
  0 2 0 0
  0 0 0 0
  0 0 4 0

  This matrix has 16 elements, but only 3 are non-zero. Using traditional storage methods would require storing all 16 elements, which is inefficient. With a sparse matrix representation, only the non-zero values and their positions are stored, reducing the space requirement.
- **Key Characteristics of Sparse Matrix ADT**

  - **Efficient Memory Usage:** Sparse matrices store only non-zero elements, significantly reducing the memory required. For large matrices with very few non-zero elements, this leads to substantial savings in storage.
  - **Faster Computation:** Sparse matrices skip operations involving zero values, speeding up computations.
  - **Storage Schemes:** Sparse matrices use special storage schemes such as, **Coordinate List (COO)**, **Compressed Sparse Row (CSR)**, and **Compressed Sparse Column (CSC)**.
- **Representations of Sparse Matrices**
  - Sparse matrices are usually represented using one of the following formats:
  - **Coordinate List (COO) Format:** Each non-zero element is stored along with its row and column index. Example: Row: [0, 1, 3], Col: [3, 1, 2] and Value: [3, 2, 4]. Here, the non-zero values are stored along with their corresponding row and column indices.
  - **Compressed Sparse Row (CSR) Format:** The matrix is stored in three arrays:
    - Values: Stores non-zero elements.
    - Column Index: Stores the column indices of non-zero elements.
    - Row Pointer: Stores the cumulative count of non-zero elements up to the beginning of each row.
  - **Example for the above matrix:**
    - Values: [3, 2, 4],
    - Column Index: [3, 1, 2], and
    - Row Pointer: [0, 1, 2, 2, 3]. Here, Row Pointer indicates the cumulative count of non-zero elements up to the beginning of each row.
- **Operations on Sparse Matrix ADT**
  - **Addition:** Add corresponding non-zero elements from two matrices. If a position has non-zero elements in both matrices, their sum is stored in the resultant matrix.
  - **Multiplication:** Multiply corresponding non-zero elements from the matrices. Only non-zero rows and columns are multiplied, skipping zero elements.
  - **Transpose:** Swap the row and column indices for each non-zero element. This is particularly useful for certain matrix algorithms.
  - **Scalar Multiplication:** Multiply each non-zero element by a scalar value.
- **Applications of Sparse Matrix ADT**
  - **Graph Representations:** Sparse matrices are used to represent graphs with a large number of vertices but few edges. For a graph with 1000 vertices and only 10 connections, using a dense adjacency matrix would waste a lot of memory.
  - **Image Processing:** Sparse matrices represent large images where most of the pixels have the same background value. This is common in scenarios like medical imaging or geographic data.
  - **Machine Learning:** Sparse matrices are used in datasets like text analysis (e.g., document-term matrices) and recommender systems. In natural language processing, a **bag-of-words** model or **TF-IDF** representation results in large matrices with many zero entries.
  - **Finite Element Method:** In engineering and physics simulations, sparse matrices are used to represent systems of equations resulting from discretization of partial differential equations (PDEs).
  - **Optimization Problems:** Sparse matrices are employed in linear programming and optimization problems where the constraint matrices are often sparse.

- **Scientific Computing:** Many problems in computational science result in sparse matrices, especially those involving large-scale simulations of physical systems (e.g., electrical networks, fluid dynamics).

- **Sparse Matrix Example in C++**
  - Consider the following C++ code to represent a sparse matrix using the **Coordinate List (COO) Format**:

```
#include <iostream>
#include <vector>
using namespace std;

// Class to represent a sparse matrix using Coordinate List (COO) format
class SparseMatrix {
private:
  int rows, cols;  // Number of rows and columns in the matrix
  vector<int> rowIndex;  // List of row indices for non-zero elements
  vector<int> colIndex;  // List of column indices for non-zero elements
  vector<int> value;     // List of non-zero values

public:
  // Constructor to initialize matrix size
  SparseMatrix(int r, int c) : rows(r), cols(c) {}

  // Function to add a non-zero element to the sparse matrix
  void addElement(int r, int c, int val) {
    if (val != 0) {
      rowIndex.push_back(r);
      colIndex.push_back(c);
      value.push_back(val);
    }
  }

  // Function to display the sparse matrix
  void display() {
    cout << "Row Index: ";
    for (int r : rowIndex)
      cout << r << " ";
    cout << endl;

    cout << "Column Index: ";
    for (int c : colIndex)
      cout << c << " ";
    cout << endl;

    cout << "Values: ";
    for (int v : value)
      cout << v << " ";
    cout << endl;
  }
```

```
};

int main() {
  // Create a sparse matrix of size 4x4
  SparseMatrix sparse(4, 4);

  // Add some non-zero elements to the sparse matrix
  sparse.addElement(0, 3, 3);
  sparse.addElement(1, 1, 2);
  sparse.addElement(3, 2, 4);

  // Display the sparse matrix
  cout << "Sparse Matrix Representation (COO Format): " << endl;
  sparse.display();

  return 0;
}
```

- **Advantages of Sparse Matrices**
  - **Efficient Memory Usage:** Store only non-zero elements, reducing memory consumption significantly.
  - **Faster Operations:** Skipping zero elements speeds up matrix operations like multiplication.
  - **Scalability:** Efficient for very large matrices with a sparse structure.
- **Disadvantages of Sparse Matrices**
  - **Complexity in Operations:** Operations such as addition and multiplication can be more complex compared to dense matrices.
  - **Overhead:** Requires additional storage for row and column indices, leading to increased overhead for very dense matrices.
- Understanding Sparse Matrix ADT is crucial in applications where memory optimization and computational efficiency are necessary, making it a valuable tool in data structures and algorithms.
- **Compressed Sparse Row (CSR) Format for Sparse Matrix.**
  - Compressed **Sparse Row (CSR) Format** is a popular data structure used to store sparse matrices efficiently. It is particularly suitable when dealing with large matrices with a significant number of zero elements. The CSR format represents a matrix using three separate arrays:
    - **Values**: Stores all the non-zero elements of the matrix.
    - **Column Indices**: Stores the column indices corresponding to each non-zero element.
    - **Row Pointers**: Stores the cumulative count of non-zero elements up to the beginning of each row.
  - **Understanding CSR with an Example:**
    - Let's consider a simple 4x4 matrix:

      0 0 3 0
      4 0 0 6
      0 0 0 0
      7 0 8 0
    - We will construct the three arrays step-by-step.

        - **Values**: Only the non-zero elements of the matrix are stored sequentially. In the above matrix, the non-zero elements are 3, 4, 6, 7, 8. **Values Array**: [3, 4, 6, 7, 8]
        - **Column Indices**: This array stores the column index for each non-zero element in the Values array. In the matrix, the non-zero elements are located at the following positions:
            1. 3 is in the 1st row, 3rd column → index 2
            2. 4 is in the 2nd row, 1st column → index 0
            3. 6 is in the 2nd row, 4th column → index 3
            4. 7 is in **the** 4th row, 1st column → index 0
            5. 8 is in the 4th row, 3rd column → index 2
            6. **Column Indices Array**: [2, 0, 3, 0, 2]
    - **Row Pointers**: This array holds the cumulative count of non-zero elements at the start of each row. It helps us identify the start and end of each row's elements in the Values array. The Row Pointers array has a length of number of rows + 1.
    - To construct this array, we count the number of non-zero elements in each row:
        - Row 1: 1 non-zero element
        - Row 2: 2 non-zero elements
        - Row 3: 0 non-zero elements
        - Row 4: 2 non-zero elements
    - The cumulative sum of these counts gives us the Row Pointers array:
        - Row 1 starts at index 0
        - Row 2 starts at index 1
        - Row 3 starts at index 3
        - Row 4 starts at index 3
        - End of matrix is at index 5
    - **Row Pointers Array**: [0, 1, 3, 3, 5]
    - **Final CSR Representation:**
        - For the above matrix, the CSR format can be summarized as:
        - **Values**: [3, 4, 6, 7, 8]
        - **Column Indices**: [2, 0, 3, 0, 2]
        - **Row Pointers**: [0, 1, 3, 3, 5]
    - **How to Interpret:**
        - To find the non-zero elements in each row:
        - **Row 1**: From index 0 to 1 (exclusive) in the Values array → [3] at column 2
        - **Row 2**: From index 1 to 3 (exclusive) → [4, 6] at columns 0, 3
        - **Row 3**: From index 3 to 3 (empty row) → []
        - **Row 4**: From index 3 to 5 → [7, 8] at columns 0, 2
    - This representation is compact and efficient, as it only stores non-zero elements, making it ideal for large sparse matrices.
    - **Time Complexity Compressed Sparse Row (CSR) format:**
        - In the **Compressed Sparse Row (CSR)** format, retrieving the value at a specific row and column index involves the following steps:
        - Identify **the Row**: Use the Row Pointers array to find the starting and ending positions of the non-zero elements for the given row.
        - **Search Within the Row**: Check the Column Indices array between these positions to see if the target column index is present.
        - **Get the Value**: If the column index is found, use this position to get the corresponding value from the Values array.

    - **Time Complexity Analysis:**
        - Let's break down the operations:
        - **Identify the Row**: Accessing the Row Pointers array to find the start and end index of a given row takes constant time **O(1)**.
        - **Search Within the Row**: The search within the Column Indices array is performed between the two indices obtained from the Row Pointers array. In the worst case, this search could involve traversing all non-zero elements of the row, making it a **linear search**. If the number of non-zero elements in the row is k, this operation has a **time complexity of O(k)**.
        - **Get the Value**: If the column index is found, accessing the corresponding value in the Values array also takes constant time **O(1)**.
    - **Overall Time Complexity:**
        - The overall time complexity to get a value at a specific row and column index is dominated by the **linear search** within the row's non-zero elements, resulting in a worst-case complexity of: O(k)
        - where k is the number of non-zero elements in the target row. If k is small, the search is efficient; if k is large (e.g., the row is dense), the search time increases accordingly.
- **Time Complexity of a dense matrix representation**
    - In a simple matrix representation (also called a dense matrix representation), matrices are stored in a 2D array or a **flat** 1D array. The complexities of common matrix operations using this representation depend on the size of the matrix, typically defined as m x n, where:
        - m = number of rows
        - n = number of columns
    - Below are **the** time complexities for some common matrix operations:
- **1. Accessing an Element:**
    - **Operation**: Retrieve the element at a specific row and column (e.g., matrix[i][j]).
    - **Complexity**: **O(1)**.
    - **Explanation**: Direct access to any element takes constant time.
- **2. Traversing the Entire Matrix:**
    - **Operation**: Visiting every element in the matrix (e.g., printing all elements, summing all elements).
    - **Complexity**: **O(m × n)**.
    - **Explanation**: The entire matrix has m * n elements, and each element needs to be processed once.
- **3. Matrix Addition/Subtraction:**
    - **Operation**: Add or subtract two matrices of size m x n.
    - **Complexity**: **O(m × n)**.
    - **Explanation**: Every element in the two matrices must be added/subtracted.
- **4. Matrix Multiplication:**
    - **Operation**: Multiply two matrices. Suppose A is of size m × n and B is of size n × p.
    - **Complexity**: **O(m × n × p)**.
    - **Explanation**: Each element in the resulting matrix is computed using a dot product of rows of A and columns of B, which involves O(n) multiplications and additions.
- **5. Transpose of a Matrix:**
    - **Operation**: Transpose a matrix of size m x n (converting rows into columns).
    - **Complexity**: **O(m × n)**.

    - **Explanation**: Each element needs to be moved to its transposed position, requiring a single traversal of all elements.
- **6. Matrix-Vector Multiplication:**
    - **Operation**: Multiply a matrix of size $m \times n$ by a vector of size $n \times 1$.
    - **Complexity**: **$O(m \times n)$**.
    - **Explanation**: Each of the m rows of the matrix is multiplied by the n-element vector.
- **7. Determinant Calculation:**
    - **Operation**: Calculate the determinant of a square matrix of size $n \times n$.
    - **Complexity**: **$O(n!)$** for a naive approach, but using efficient algorithms like **LU decomposition**, it can be done in **$O(n^3)$**.
    - **Explanation**: Determinants are typically computed using Gaussian elimination or cofactor expansion, both of which have a cubic time complexity for large matrices.
- **8. Inverse of a Matrix:**
    - **Operation**: Compute the inverse of a square matrix of size n x n.
    - **Complexity**: **$O(n^3)$** using Gaussian elimination or matrix decomposition methods.
    - **Explanation**: Finding the inverse involves solving a set of linear equations, which can be done in cubic time.
- **9. Matrix Exponentiation:**
    - **Operation**: Raise a square matrix of size $n \times n$ to a power k.
    - **Complexity**: **$O(n^3 \times \log(k))$**.
    - **Explanation**: Using repeated squaring (exponentiation by squaring), each squaring operation takes $O(n^3)$ time, and we need log(k) such operations.
- **10. Row/Column Sum:**
    - **Operation**: Calculate the sum of a particular row or column.
    - **Complexity**: **$O(n)$** for row sum and **$O(m)$** for column sum.
    - **Explanation**: For a row, traverse all elements in that row (total n elements); for a column, traverse all elements in that column (total m elements).
- **11. Finding Maximum/Minimum Element:**
    - **Operation**: Find the maximum or minimum value in a matrix.
    - **Complexity**: **$O(m \times n)$**.
    - **Explanation**: Every element must be compared, requiring a full traversal of the matrix.
- **12. Checking if a Matrix is Symmetric:**
    - **Operation**: Check if a square matrix is symmetric (i.e., A[i][j] == A[j][i] for all i and j).
    - **Complexity**: **$O(n^2)$** for an $n \times n$ square matrix.
    - **Explanation**: Only the upper triangle of the matrix needs to be compared with the lower triangle.
- **Summary:**
    - **Accessing an element**: $O(1)$
    - **Traversing the matrix**: $O(m \times n)$
    - **Addition/Subtraction**: $O(m \times n)$
    - **Multiplication**: $O(m \times n \times p)$
    - **Transpose**: $O(m \times n)$
    - **Determinant**: $O(n^3)$
    - **Inverse**: $O(n^3)$
    - **Matrix Exponentiation**: $O(n^3 \times \log(k))$
- For dense matrices, many operations can become computationally intensive as matrix sizes increase, making more efficient representations and algorithms necessary in practical applications.