Comprehensive Guide to Arrays and Strings in Data Structures

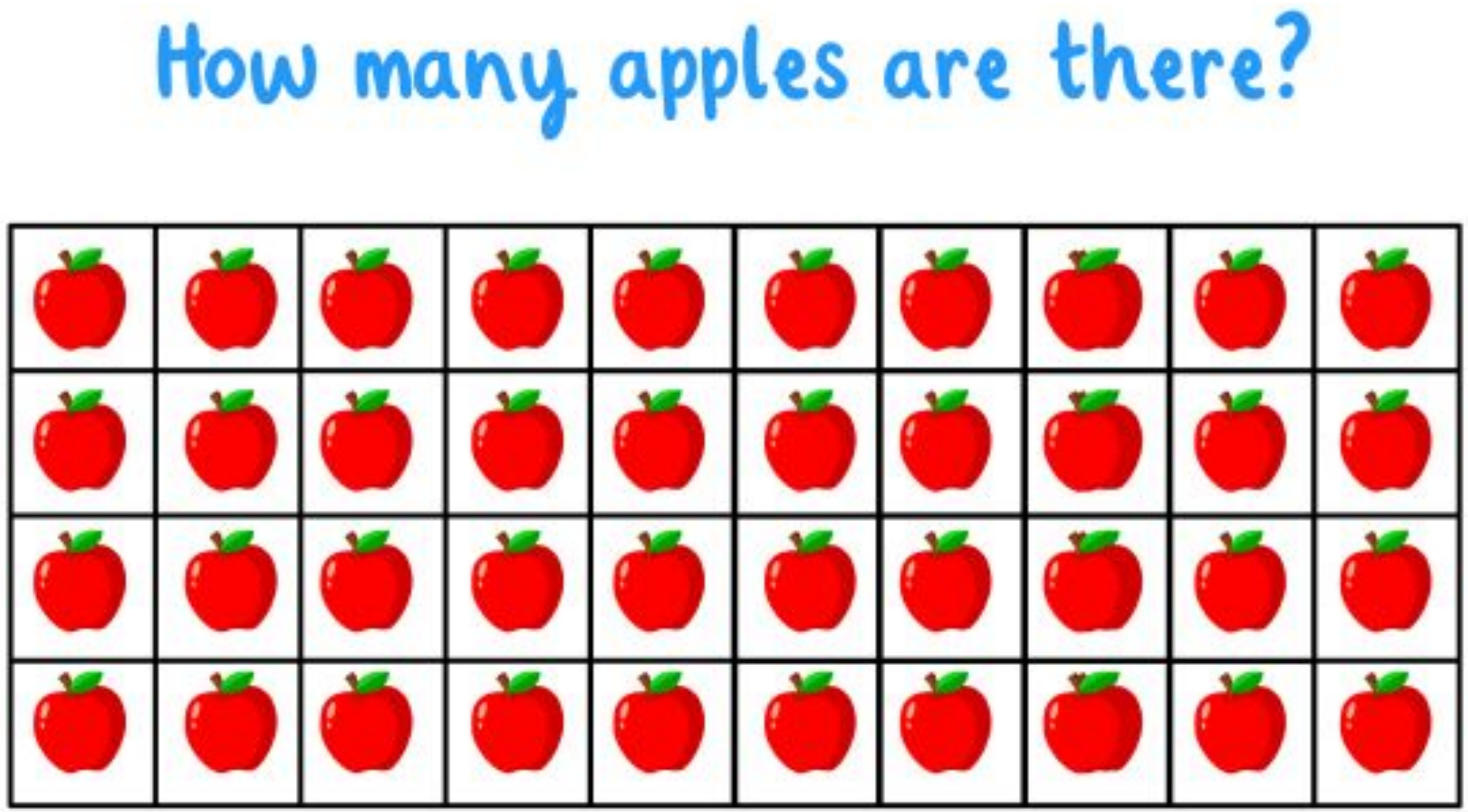
Dr. Machbah Uddin

Associate Professor

Dept. of Computer Science & Math

Bangladesh Agricultural University

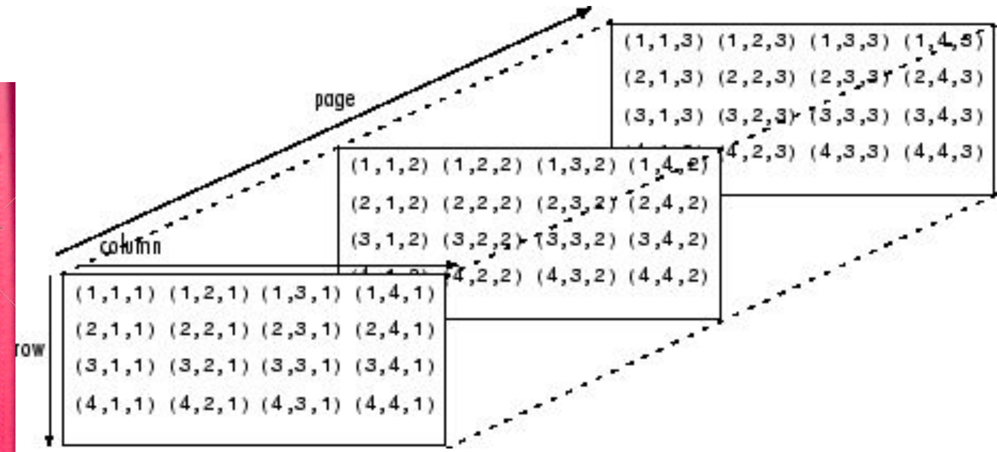
Arrays



Dr. Machbah Uddin, Associate Professor, BAU, machbah.csm@bau.edu.bd

Array

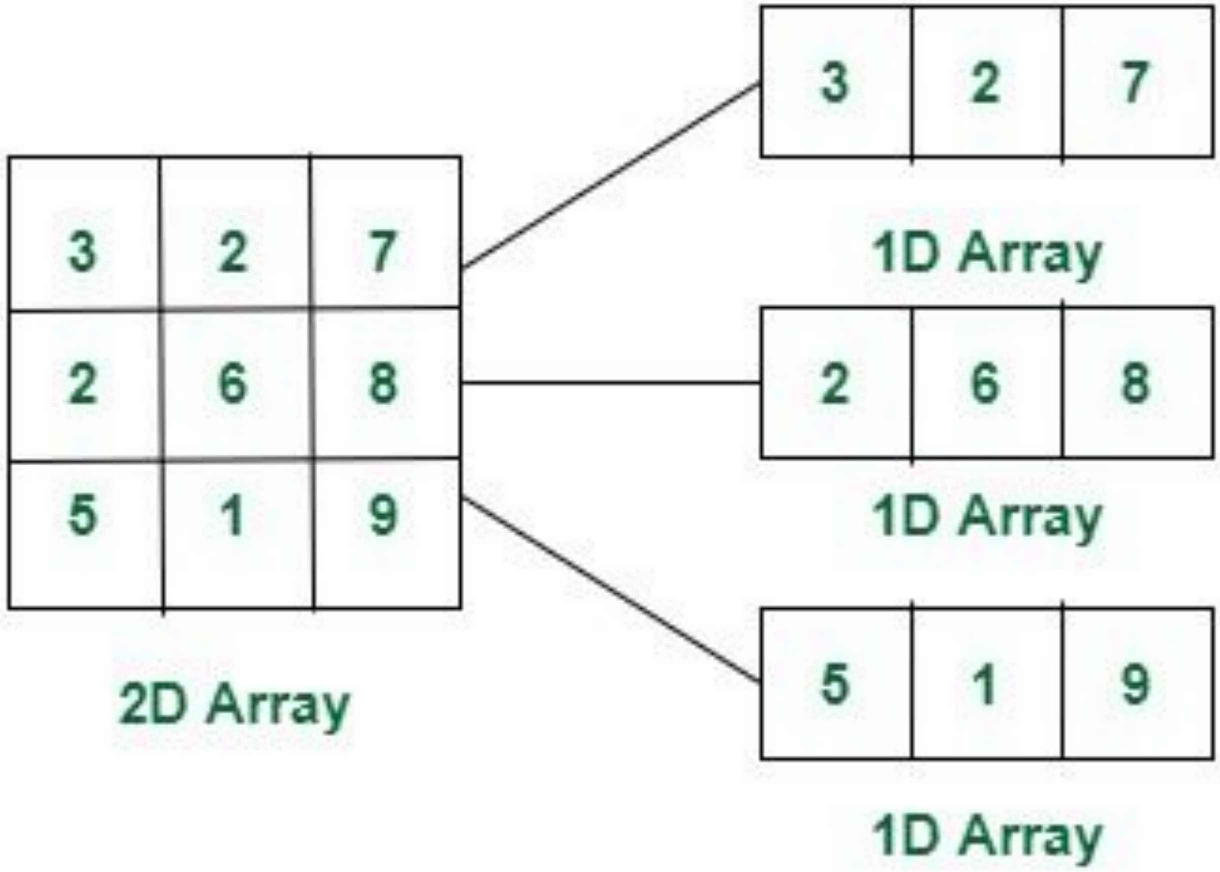

BIGSTOCK


page
column
row
(1,1,3) (1,2,3) (1,3,3) (1,4,3)
(2,1,3) (2,2,3) (2,3,3) (2,4,3)
(3,1,3) (3,2,3) (3,3,3) (3,4,3)
(4,2,3) (4,3,3) (4,4,3)
(1,1,2) (1,2,2) (1,3,2) (1,4,2)
(2,1,2) (2,2,2) (2,3,2) (2,4,2)
(3,1,2) (3,2,2) (3,3,2) (3,4,2)
(4,2,2) (4,3,2) (4,4,2)
(1,1,1) (1,2,1) (1,3,1) (1,4,1)
(2,1,1) (2,2,1) (2,3,1) (2,4,1)
(3,1,1) (3,2,1) (3,3,1) (3,4,1)
(4,1,1) (4,2,1) (4,3,1) (4,4,1)

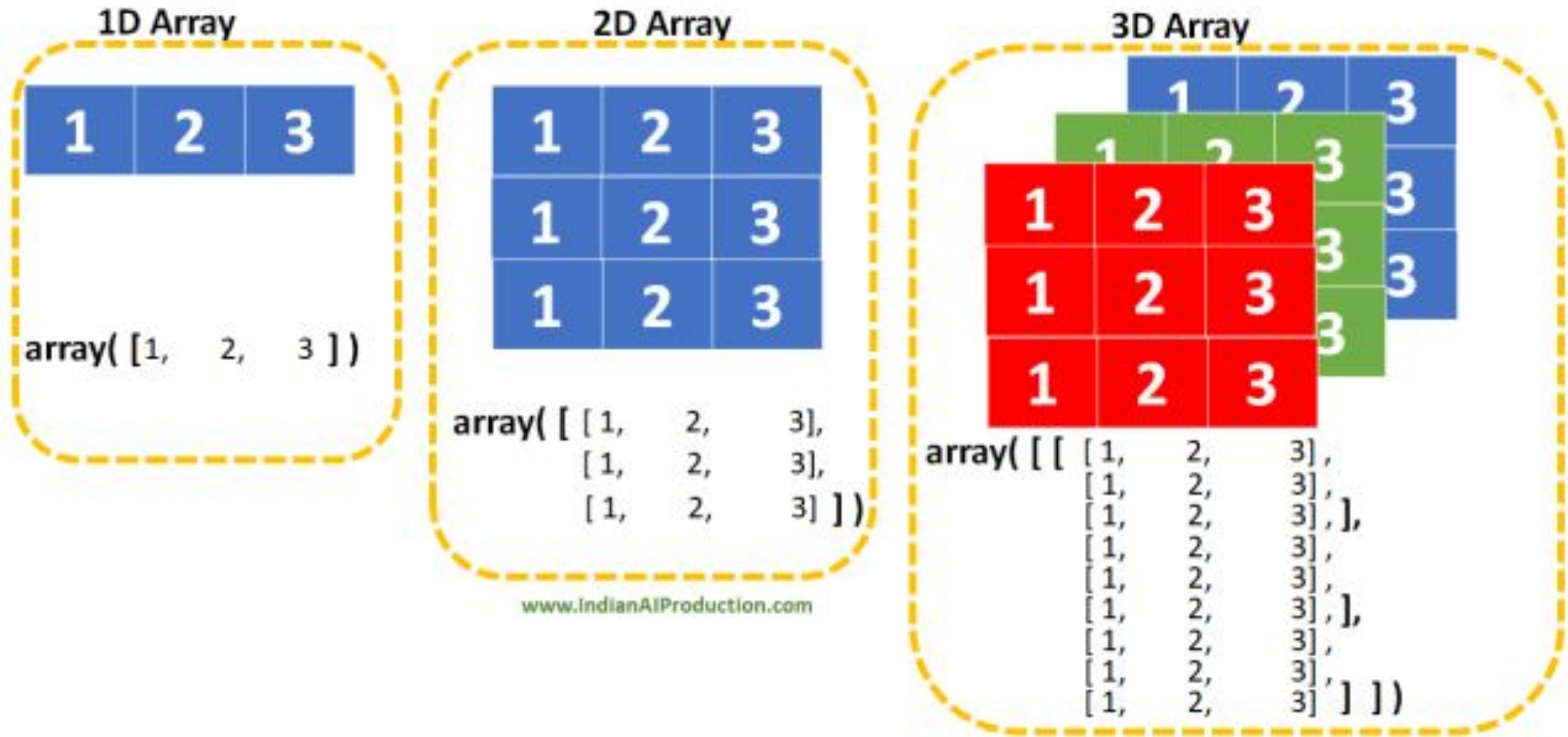
What is an Array?

- An array is a data structure that stores elements of the same type in contiguous memory.
- It allows random access to elements using an index.
- Types: One-dimensional (1D) and Multidimensional Arrays.

1D and 2D array

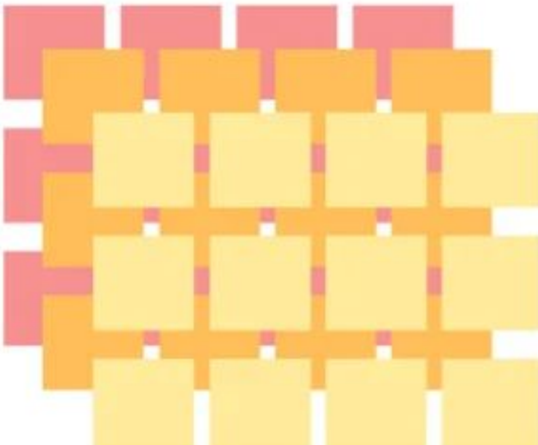


3D arrays

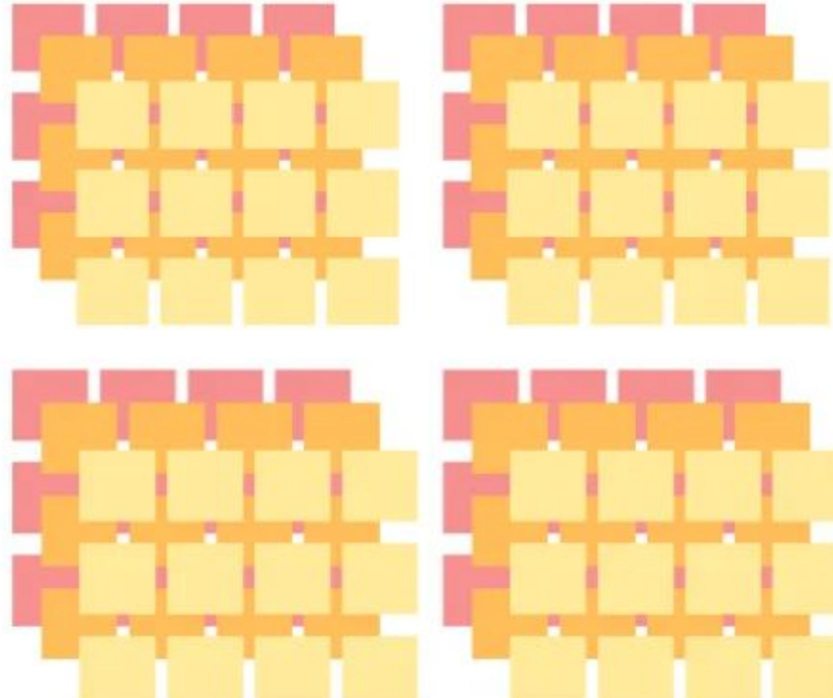


1D Array
1 2 3
array([1, 2, 3])
2D Array
1 2 3
1 2 3
1 2 3
array([[1, 2, 3],
[1, 2, 3],
[1, 2, 3]])
www.IndianAIProduction.com
3D Array
1 2 3
1 2 3
1 2 3
array([[[1, 2, 3] ,
[1, 2, 3] ,
[1, 2, 3] ,],
[1, 2, 3] ,
[1, 2, 3] ,
[1, 2, 3] ,],
[1, 2, 3] ,
[1, 2, 3] ,
[1, 2, 3]]])

4D arrays



3D array: An array of 2D arrays



4D array: An array of 3D arrays

Why Use Arrays?

- Efficient way to store multiple values.
- Fast access time using indices.
- Saves memory by avoiding overhead of pointers.

One-Dimensional Array (1D)

- A linear data structure where elements are stored in a single row.
- Example: int arr[5] = {1, 2, 3, 4, 5};

Multidimensional Arrays

- Arrays with more than one index (e.g., 2D, 3D).
- Example (2D Array): int matrix[3][3] = {{1,2,3},{4,5,6},{7,8,9}};

Array Operations Overview

- Insertion
- Traversal
- Deletion
- Searching

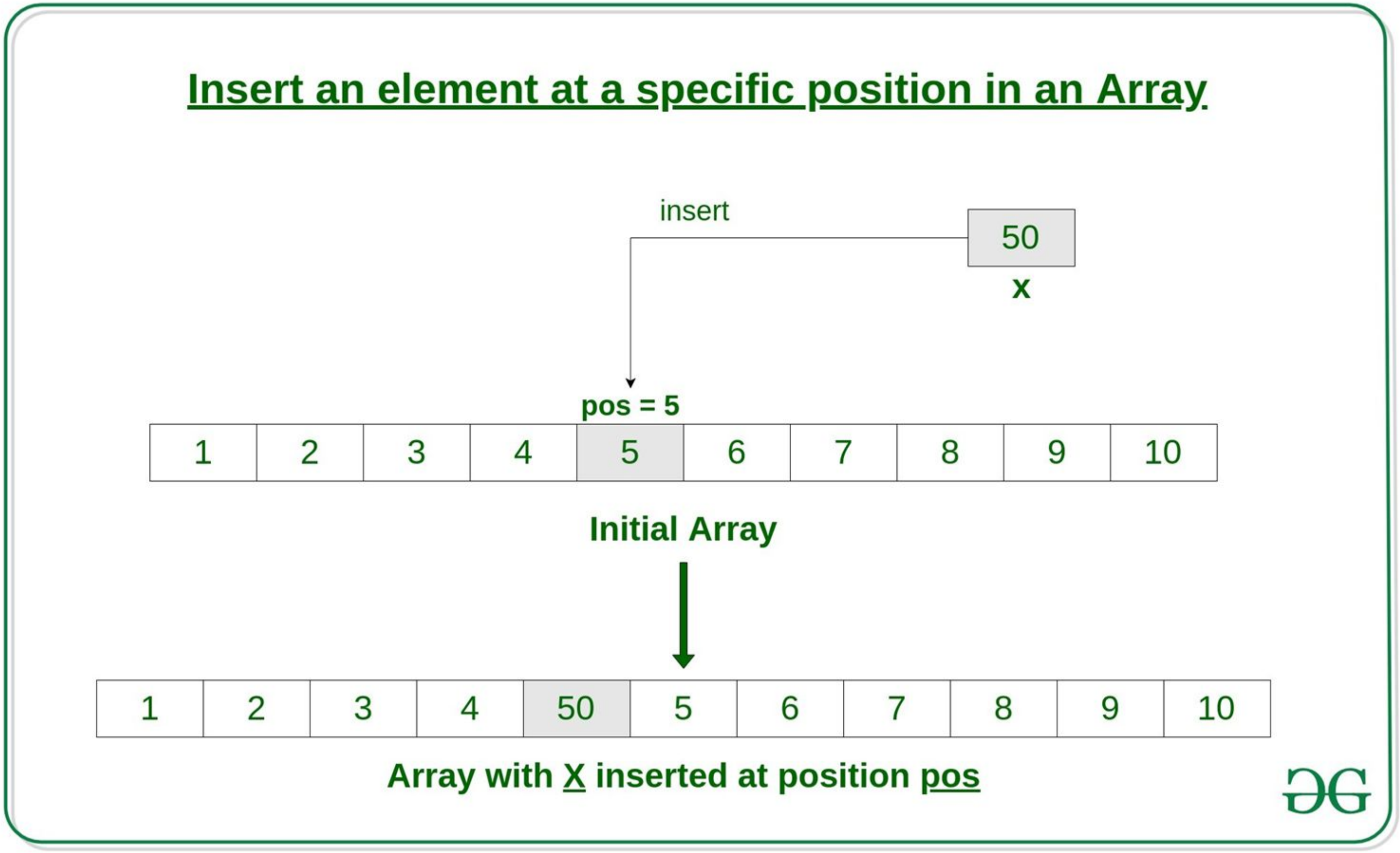
Insertion in Arrays

- Adding an element at a specific position.
- If inserting in the middle, shift elements to the right.

Example: Insertion

- Given array: [10, 20, 30, 40]
- Insert 25 at index 2:
- [10, 20, 25, 30, 40]

Array insert



Traversing an Array

- Accessing each element one by one.
- Can be done using loops.
- Example: for i in range(len(arr)): print(arr[i])

Example: Traversal

- Array: [5, 10, 15, 20]
- Output: 5 10 15 20

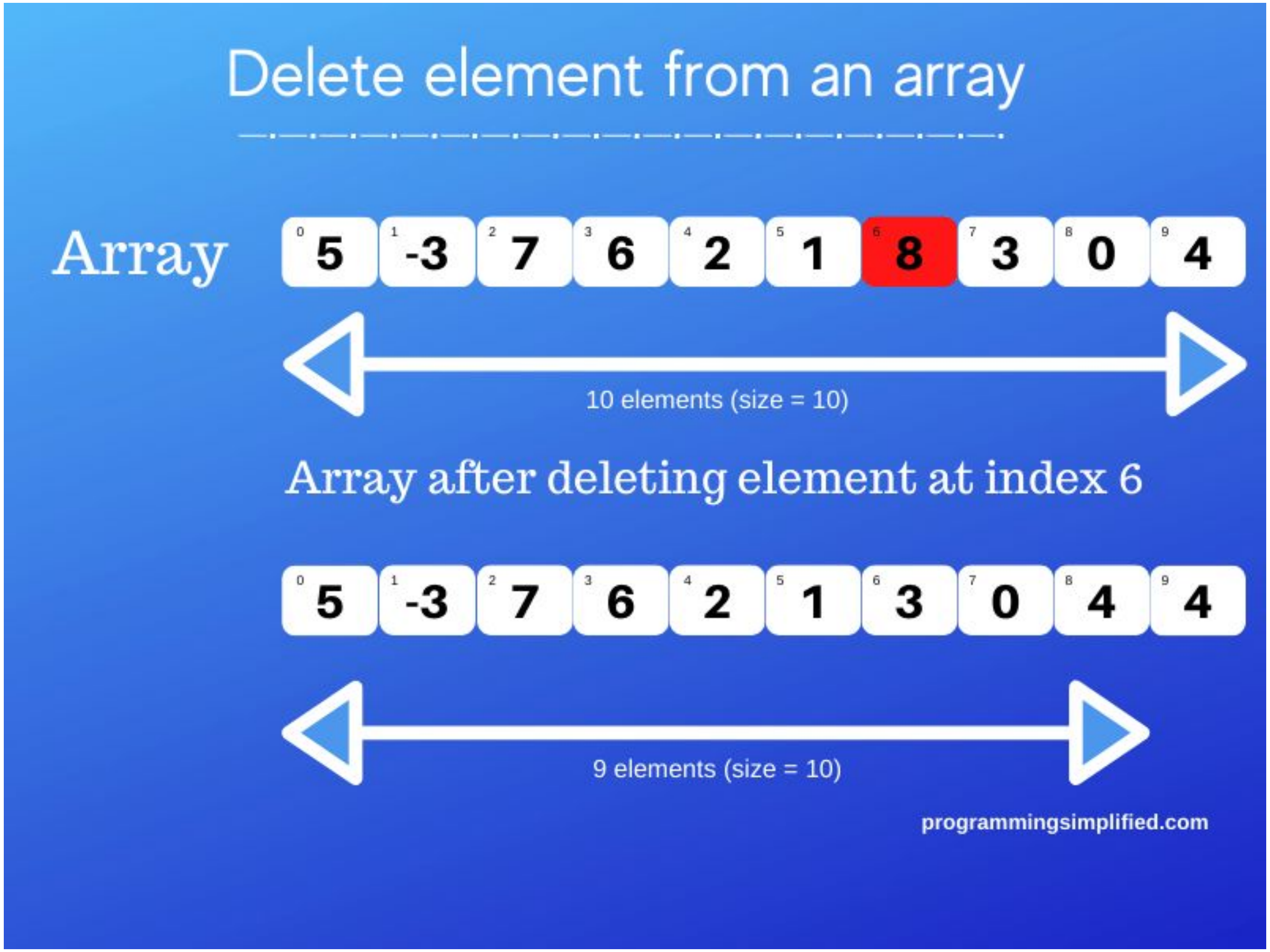
Deletion in Arrays

- • Removing an element by shifting elements to the left.
- • Last element becomes empty.

Example: Deletion

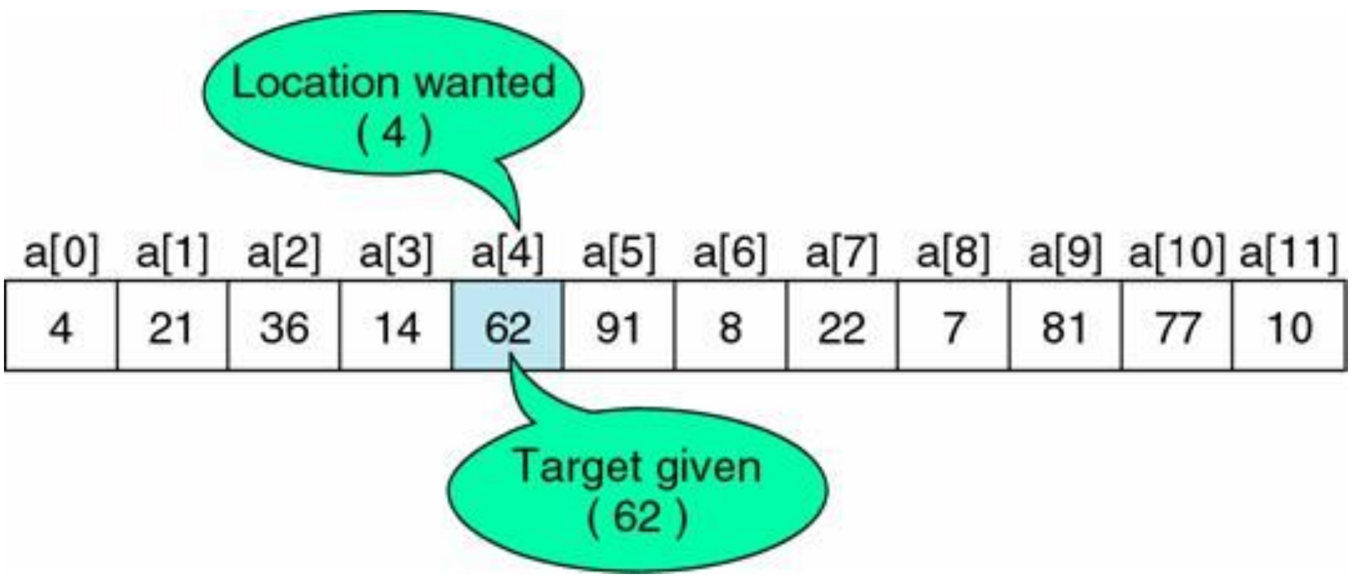
- Given array: [10, 20, 30, 40]
- Delete element at index 2:
- [10, 20, 40]

Array Element Delete



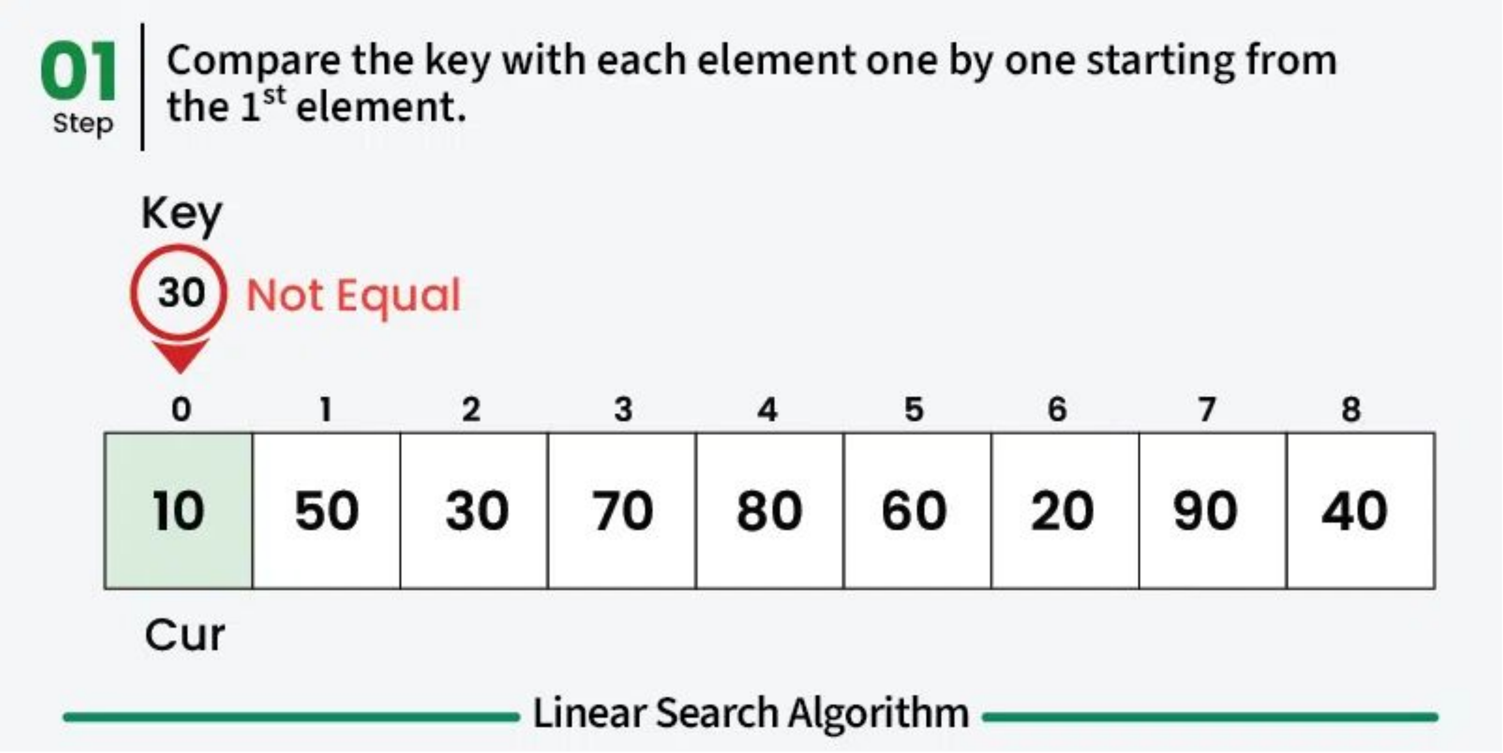
Search Operation in Arrays

- • Linear Search: Check elements one by one.
- • Binary Search: Works on sorted arrays, divides array into halves.



Example: Linear Search

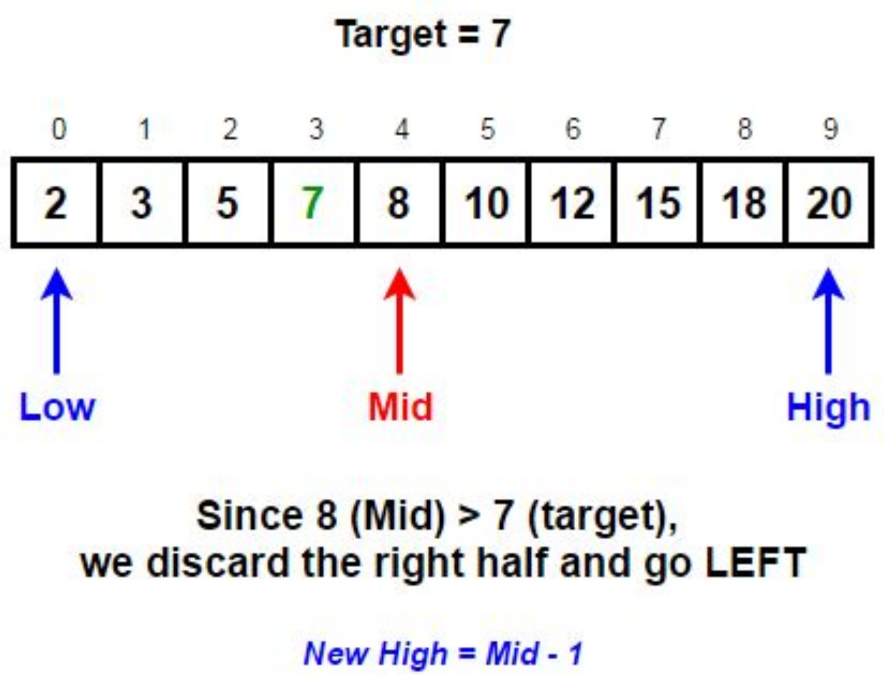
- Array: [5, 10, 15, 20]
- Find 15 ⟶ Found at index 2.



Dr. Machbah Uddin, Associate Professor, BAU, machbah.csm@bau.edu.bd

Example: Binary Search

- Sorted Array: [5, 10, 15, 20]
- Find 15 → Mid = 10 → Search right → Found.



2D Arrays: Traversal

- • Uses nested loops.
- Example:
- for i in range(rows):
- for j in range(cols):
- print(matrix[i][j])

2D Arrays: Insertion

- • Specify row and column index.
- Example:
- matrix[1][2] = 9

2D Arrays: Deletion

- • Remove an element by shifting elements in a row/column.
- Example:
- Delete row 2 → Shift remaining rows up.

2D Arrays: Searching

- • Linear search checks each element.
- • Binary search only works on sorted 2D arrays.

Example: 2D Array Search

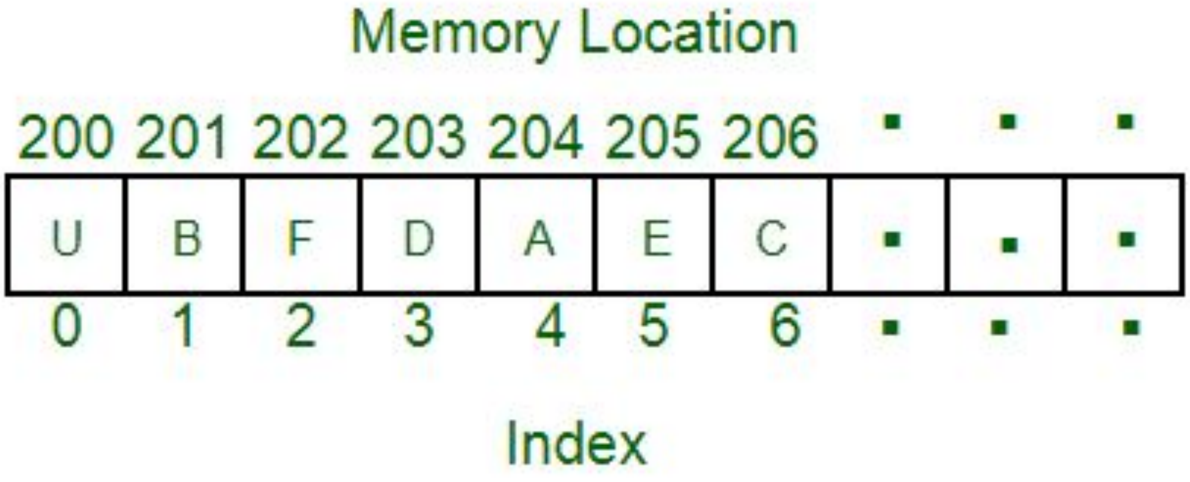
- Given matrix:
- [[1, 2], [3, 4]]
- Find 3 $\longrightarrow$ Found at (1,0).

Memory Representation of 1D Arrays

- • Elements stored in contiguous memory.
- • Address calculated as: Base Address + (Index * Size).

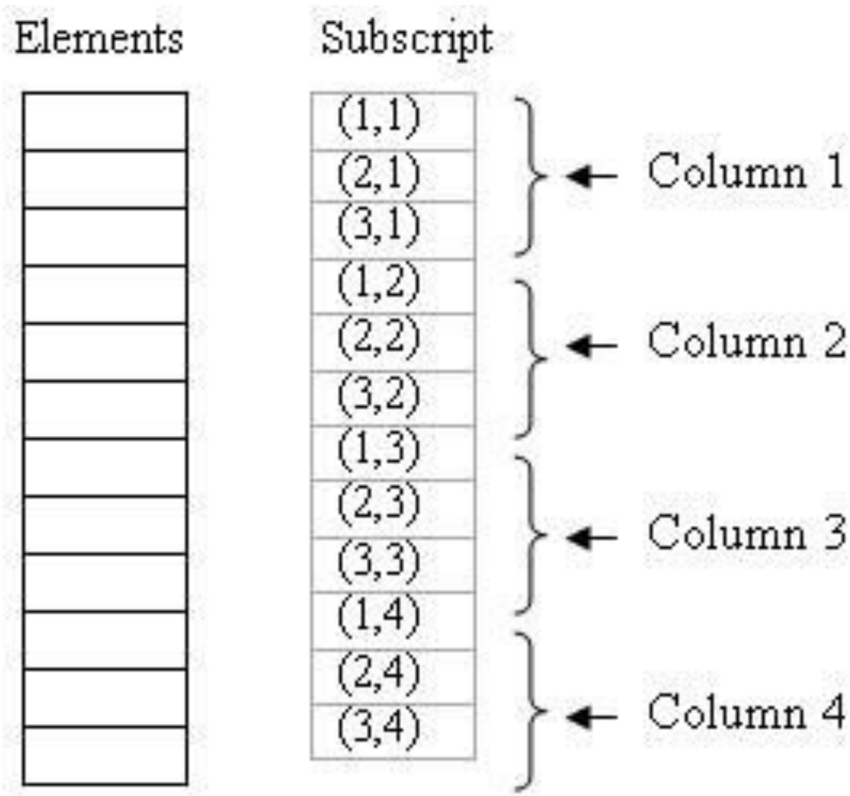
Example: Memory in 1D Arrays

- Array: [10, 20, 30]
- Base Address = 1000, Size = 4 bytes.
- Addresses: 1000, 1004, 1008.



Memory Representation of 2D Arrays

- • Stored in Row-Major or Column-Major Order.
- • Address = Base + [(Row * ColCount) + Col] * ElementSize.



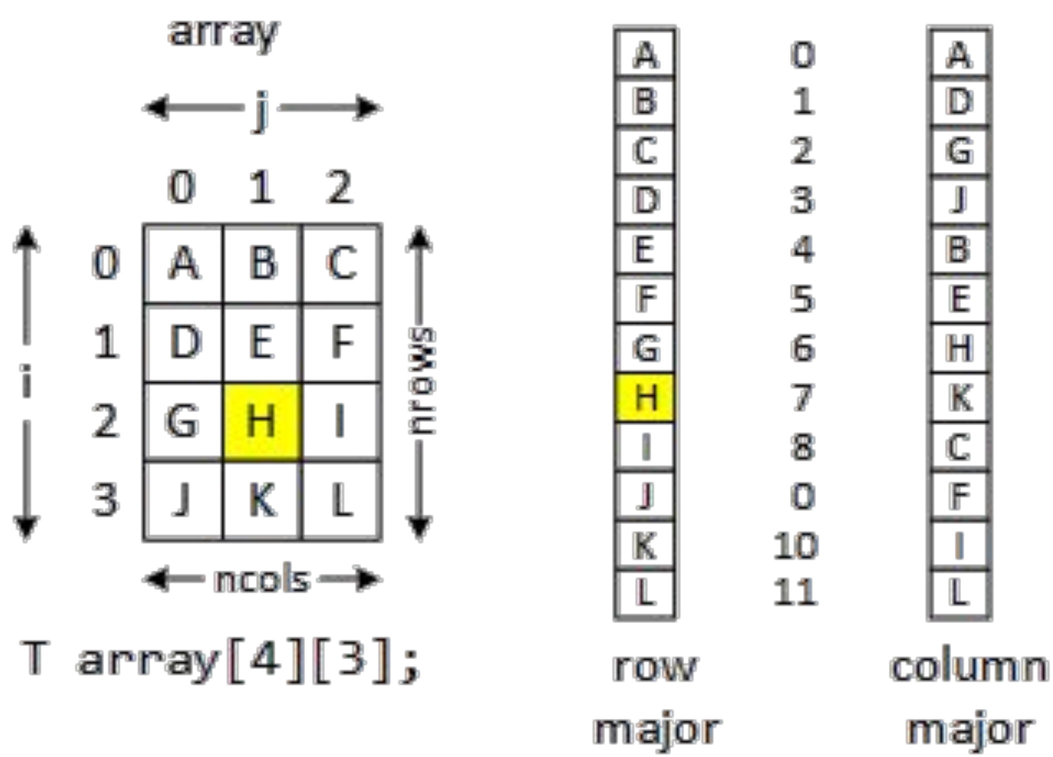
Elements
Subscript
(1,1)
(2,1)
(3,1)
(1,2)
(2,2)
(3,2)
(1,3)
(2,3)
(3,3)
(1,4)
(2,4)
(3,4)
Column 1
Column 2
Column 3
Column 4

Example: Memory in 2D Arrays

- Matrix: [[1, 2], [3, 4]]
- Base Address = 1000, Size = 4 bytes.
- Row-Major: 1000, 1004, 1008, 1012.

Comparing Row-Major vs Column-Major

- Row-Major: Elements of each row stored together.
- Column-Major: Elements of each column stored together.

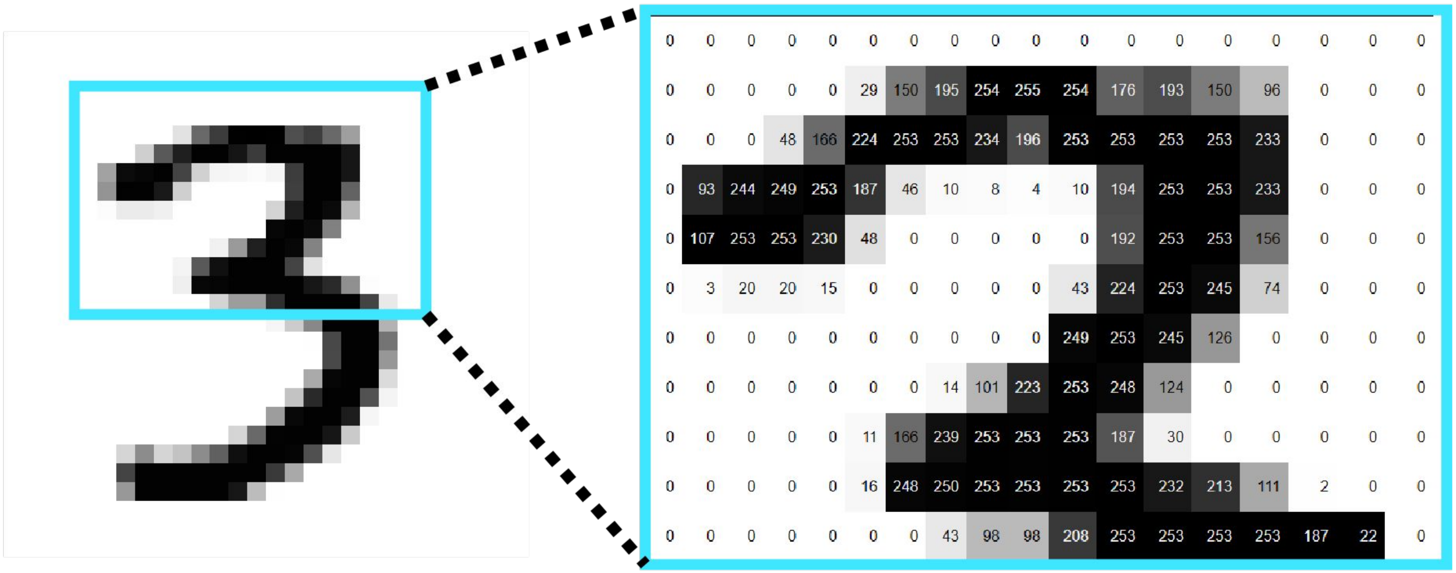


Applications of Arrays

- • Used in databases, image processing, and matrices.
- • Data storage in programming languages.

Real-World Example: Image Processing

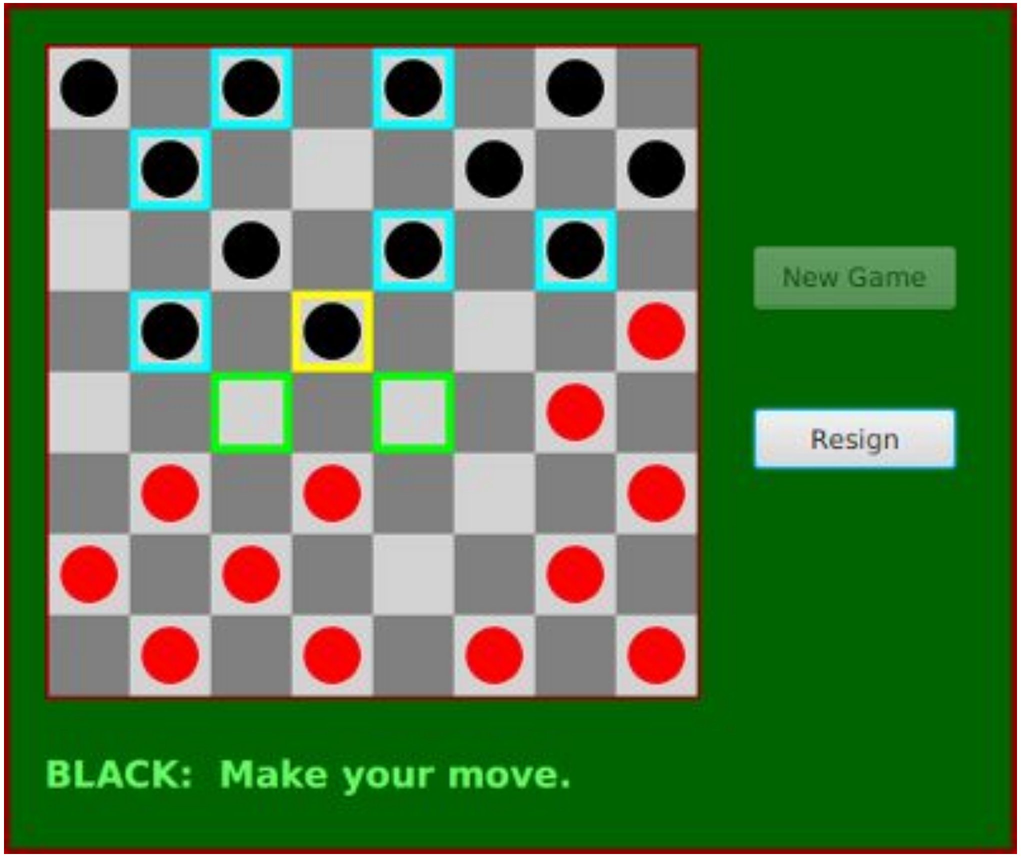
- Images stored as 2D arrays.
- Pixel values accessed using array indices.



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	29	150	195	254	255	254	176	193	150	96	0	0	0
0	0	0	48	166	224	253	253	234	196	253	253	253	253	233	0	0	0
0	93	244	249	253	187	46	10	8	4	10	194	253	253	233	0	0	0
0	107	253	253	230	48	0	0	0	0	0	192	253	253	156	0	0	0
0	3	20	20	15	0	0	0	0	0	43	224	253	245	74	0	0	0
0	0	0	0	0	0	0	0	0	0	249	253	245	126	0	0	0	0
0	0	0	0	0	0	0	14	101	223	253	248	124	0	0	0	0	0
0	0	0	0	0	11	166	239	253	253	253	187	30	0	0	0	0	0
0	0	0	0	0	16	248	250	253	253	253	253	232	213	111	2	0	0
0	0	0	0	0	0	0	43	98	98	208	253	253	253	253	187	22	0

Real-World Example: Game Development

- Grid-based games use 2D arrays.
- Example: Chess board representation.



Challenges with Arrays

- Fixed size (static arrays).
- Costly insertions/deletions in the middle.

Conclusion

- Arrays provide fast access to elements.
- Different operations allow manipulation.
- Used extensively in programming and applications.

Weather Monitoring

- **Problem:** A meteorological department collects hourly temperature readings over a month. Design a system to store and analyze this data (e.g., find the highest, lowest, and average temperatures).
- ✅ **Why Use an Array?**
- The number of readings per day is fixed (24 readings per day × 30 days = 720 readings).
- Arrays allow fast indexing to access any hour’s data efficiently.

Implementation

```
temperatures = [30, 32, 28, 27, 31, 33, 29]  # Example temperatures
highest = max(temperatures)
lowest = min(temperatures)
average = sum(temperatures) / len(temperatures)
print(f"Highest: {highest}, Lowest: {lowest}, Average: {average:.2f}")
```

Stock Price Prediction

- **Problem:** A stock analyst wants to **calculate a 5-day moving average** of stock prices to identify trends.
- ✅ **Why Use an Array?**
- The order of stock prices is crucial (sequential data).
- Arrays provide **efficient traversal** for calculating moving averages.

Implementation

- stock_prices = [100, 102, 101, 99, 98, 97, 103]
- window_size = 5
- moving_averages = [sum(stock_prices[i:i+window_size])/window_size for i in range(len(stock_prices)-window_size+1)]
- print(moving_averages)

Traffic Management System (Car Plate Number Logs)

- **Problem:** A smart traffic system stores license plate numbers of 10,000 cars daily to analyze **rush hours** and **detect duplicate entries**.
- ✅ **Why Use an Array?**
- The data is **large but structured** (each car plate is stored at a fixed index).
- Arrays allow **quick searching** and **pattern analysis**.

Implementation

- car_plates = ["ABC123", "XYZ789", "LMN456", "XYZ789"] # Duplicate detected
- duplicates = set([plate for plate in car_plates if car_plates.count(plate) > 1])
- print(f"Duplicate Plates: {duplicates}")

DNA Sequence Matching

- **Problem:** Bioinformatics researchers store long **DNA sequences** as character arrays and need to find whether a smaller DNA sequence exists in a larger one.
- ✅ **Why Use an Array?**
- DNA sequences are naturally stored as **arrays of characters**.
- Efficient searching algorithms (like **KMP or Rabin-Karp**) require array-based processing.

Implementation

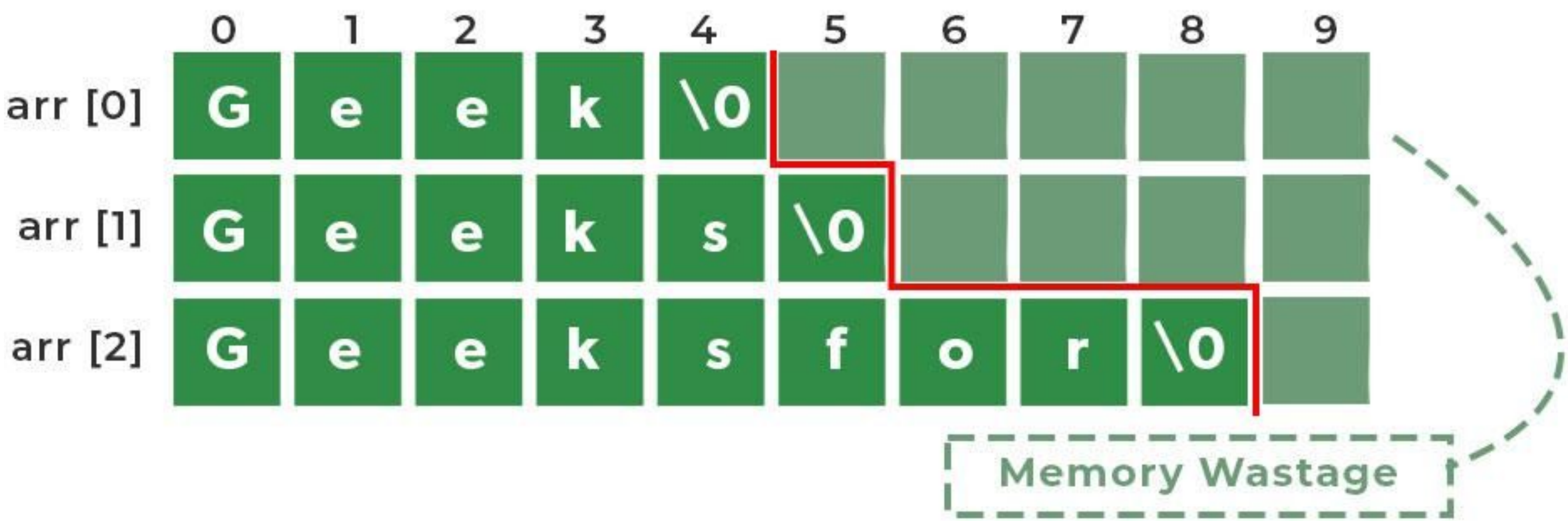
- def is_subsequence(dna, sub):

  return sub in dna
- dna_sequence = "AGCTTAGGCTA"
- sub_sequence = "GGCT"
- print("Found!" if is_subsequence(dna_sequence, sub_sequence) else "Not Found!")

Strings

- Definition: A string is a sequence of characters.
- Examples: "Hello, World!", "DNA Sequence: ATGC", "Binary String: 101010".
- Importance in computer science: Text processing, data storage, and communication.
- Applications: Search engines, bioinformatics, cryptography, and more.

Memory Representation of an Array of Strings

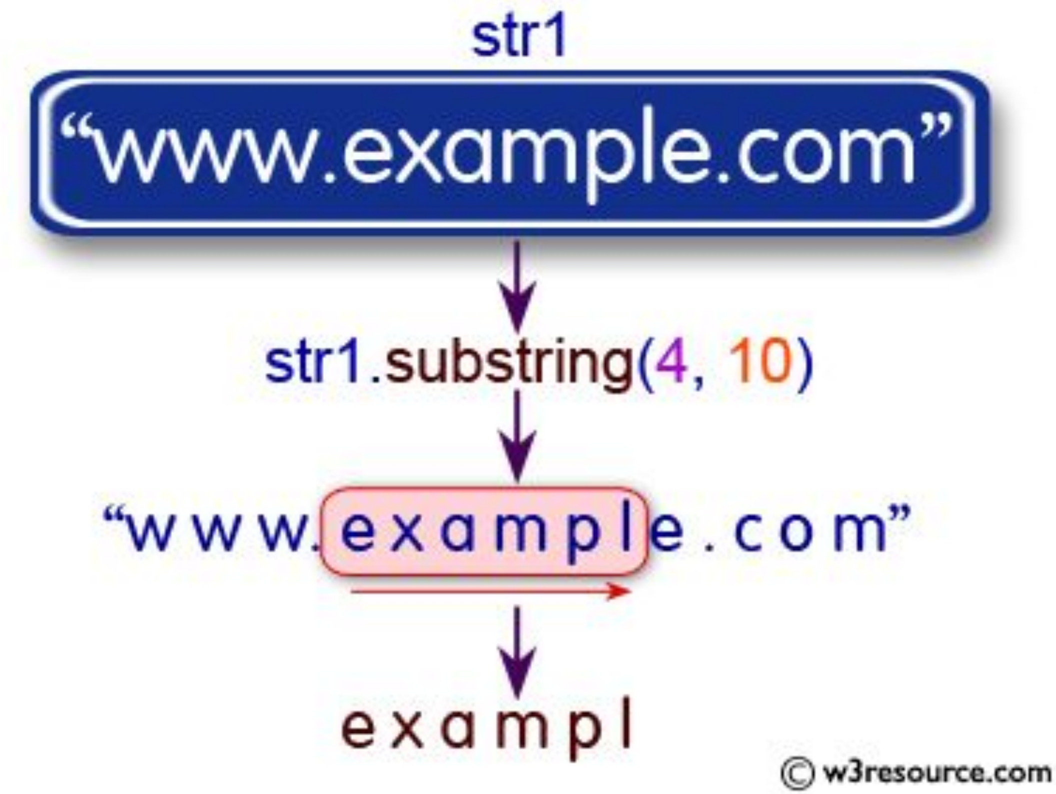


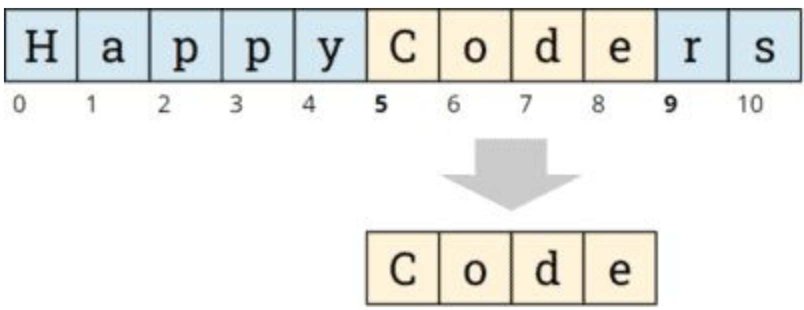
Dr. Machbah Uddin, Associate Professor, BAU, machbah.csm@bau.edu.bd

String Operations

- **Basic Operations:**
  - Concatenation: "Hello" + "World" = "HelloWorld".
  - Substring extraction: "HelloWorld".substring(0, 5) = "Hello".
  - Length calculation: len("Hello") = 5.
- **Advanced Operations:**
  - Searching for patterns.
  - Replacing substrings.
  - Splitting and joining strings.

Substring

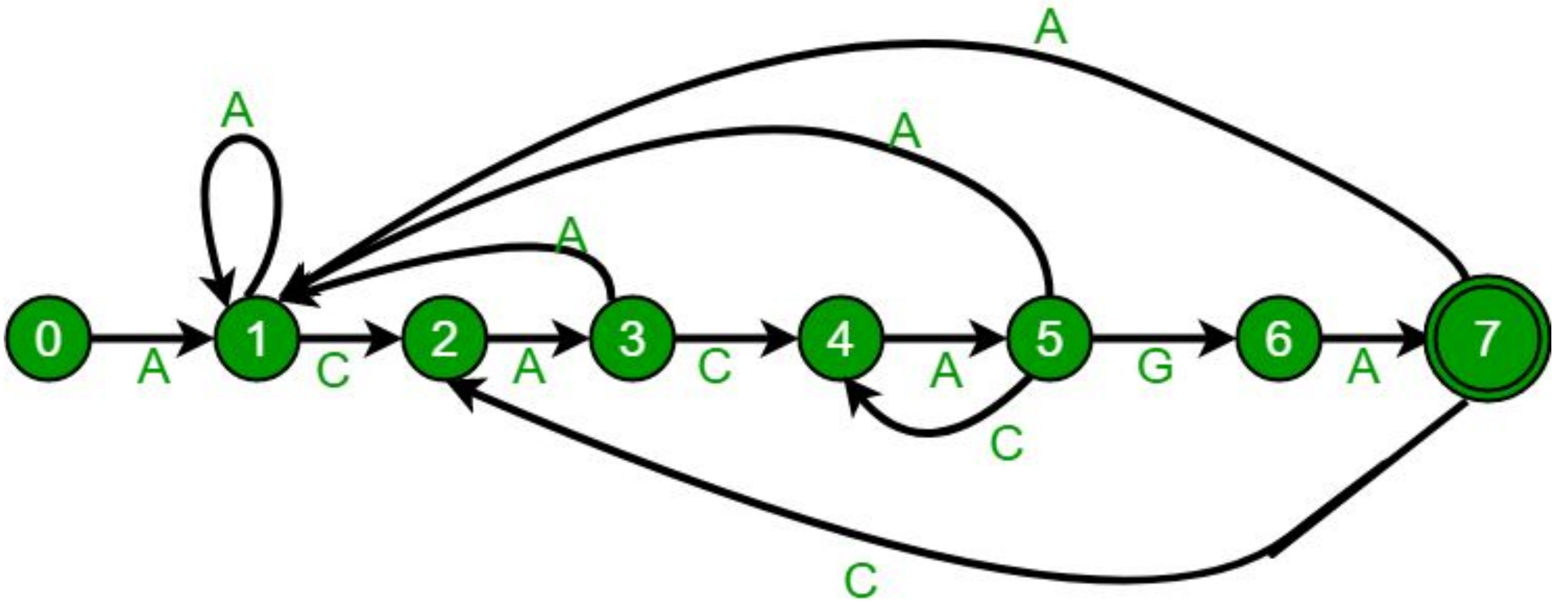

str1
"www.example.com"
str1.substring(4, 10)
"www.example.com"
exampl
© w3resource.com


H a p p y C o d e r s
0 1 2 3 4 5 6 7 8 9 10
C o d e

String searching

- **Naive (Brute-Force) Search:**
  - Checks every possible substring.
  - Time complexity: O(n*m), where n = text length, m = pattern length.
- **Knuth-Morris-Pratt (KMP) Algorithm:**
  - Uses a prefix table to skip unnecessary comparisons.
  - Time complexity: O(n + m).
- **Boyer-Moore Algorithm:**
  - Skips characters based on bad character and good suffix rules.
  - Time complexity: O(n/m) in best case.
- **Rabin-Karp Algorithm:**
  - Uses hashing to compare patterns.
  - Time complexity: O(n + m) on average.

Pattern Search



Pattern Search

Text: A A B A A C A A D A A B A A B A

Pattern: A A B A

Pattern found at index 0, 9 and 12

Dr. Machbah Uddin, Associate Professor, BAU, machbah.csm@bau.edu.bd

DNA Sequence Matching

- **Problem:** Given a DNA sequence and a short query sequence, find all occurrences of the query in the DNA sequence.
- **Why mandatory?** DNA sequences are represented as **strings** (A, T, G, C). Using arrays wouldn't work efficiently for pattern matching.

DNA Sequence Matching

- def find_dna_sequence(dna, query):
-     positions = []
-     for i in range(len(dna) - len(query) + 1):
-         if dna[i:i+len(query)] == query:
-             positions.append(i)
-     return positions

- dna_sequence = "ATGCGATGACCTGAGGCTAGCATGAC"
- query_sequence = "ATG"
- print(find_dna_sequence(dna_sequence, query_sequence)) # Output: [0, 6, 20]

Password Strength Checker

- **Problem:** Verify if a password meets security standards (length, uppercase, special character, etc.).
- 🔹 **Why mandatory?** Passwords are **always** stored and validated as **strings**.

Password Strength Checker

- import re

- def check_password_strength(password):
-     if (len(password) >= 8 and
-         re.search(r'[A-Z]', password) and
-         re.search(r'[a-z]', password) and
-         re.search(r'[0-9]', password) and
-         re.search(r'[@#$%^&+=]', password)):
-         return "Strong Password"
-     return "Weak Password"

- print(check_password_strength("Pass@123")) # Output: Strong Password
- print(check_password_strength("pass123"))   # Output: Weak Password