Cybersecurity & Artificial Intelligence

With

Python

By
Ahn Nuzen

Acknowledgments

The author extends his genuine thanks to the dedicated faculty of the Grossmont College Computer Science Department for their invaluable support throughout the development of this ZBook.

Their thoughtful reviews, constructive feedback, and encouragement through many rounds of revision significantly strengthened the quality and clarity of this work.

This ZBook was made possible through funding from the Zero Textbook Cost (ZTC) Acceleration II Grant. The author is thankful to the ZTC team at Grossmont College for their support, guidance, and commitment to accessible education

Ahn Nuzen
Grossmont College, Fall 2025

Cybersecurity & AI with Python Overview

This book offers a comprehensive study of fundamental data structures, including their theoretical foundations and practical implementations, with a particular focus on applications in Cybersecurity and Artificial Intelligence. Key topics include Big-O notation, algorithmic complexity analysis, linked lists, stacks, queues, trees, sets, dictionaries, and graphs.

Emphasis is placed on the critical role data structures play in efficiently organizing, processing, and protecting large-scale data. Through this exploration, learners will develop the skills needed to analyze, select, and implement appropriate data structures to address real-world computational challenges in high-demand domains. A foundational understanding of Python or an equivalent introductory programming language is assumed.

In addition to mastering core data structures, students will apply these principles to critical domains such as Cybersecurity, data science, and artificial intelligence. Through a series of hands-on projects, students will utilize Python to automate security workflows, perform threat analysis, and develop models for intrusion detection, malware classification, and log data processing. The course also includes practical engagement with open-source large language models (LLMs) and tools such as Ollama, providing students with experience in deploying advanced AI technologies to address real-world computational challenges.

By the end of this book, learners will be able to:

1. Analyze algorithmic efficiency using Big-O notation and complexity analysis to assess the performance of data-driven systems, particularly in AI and Cybersecurity contexts.
2. Apply data structures such as linked lists, stacks, queues, trees, sets, dictionaries, and or graphs to solve domain-specific problems in Cybersecurity, such as log analysis
3. Implement intelligent systems that leverage Large Language Model for real-time security analysis, summarization, and threat detection

Summary:

This book blends data structures and algorithms with practical cybersecurity applications, emphasizing Python programming and automation using AI tools. Students will explore classic computer science concepts like lists, trees, graphs, and complexity analysis, applied to real-world cybersecurity challenges such as malware detection, log processing, and network analysis. The book culminates in a capstone project involving a custom-built Python security tool with AI integration.

Module Organization Overview:

Module	Title	Key Topics
1	Course Overview & Cybersecurity Context	Objectives, expectations, cybersecurity threats, tools, and data structures' role
2	Big-O Notation & Complexity Analysis	Time/space complexity, algorithm analysis, code profiling and optimization
3	Lists, Tuples, Arrays in Log Tracking	Python collections, NumPy, network log processing
4	Dictionaries, Sets, Hash Tables in Password Management.	Hashing, credential storage, blacklists, whitelists, simple password manager
5	Stacks and Queues in Packet Inspection	Implementations, use cases, task scheduling, basic inspection queue
6	Graphs & Network Analysis	Graph algorithms, network mapping, attack path analysis, simple graph visualization
7	Linked Lists in Malware Analysis	Memory management, log buffers, malware behavior tracking
8	Trees & Tree Traversal in Attack Trees	Binary/search trees, file systems, attack trees
9	Working with LLMs & Ollama	Open-source LLMs, integrating Python with AI for threat detection
10	Automation & Virus/Vulnerability Scanning	Scripting scanners, using VirusTotal and Shodan APIs, NLP-based automation
11	Capstone Project	Development and presentation of a security-focused Python AI tool
12	Bonus Module: Career Readiness	How to use GitHub to showcase code, build a cybersecurity/AI resume, and prep for interviews.

Table - 1

Table of Contents

Module 1: Overview and Cybersecurity with AI Context

Objective

This book offers a rigorous exploration of data structure fundamentals while bridging theoretical computer science with cutting-edge applications in cybersecurity and artificial intelligence. By integrating abstract computational concepts with real-world problem-solving, it prepares learners to address modern challenges in data-driven domains through a structured, two-pronged approach.

1.1 Key Learning Areas:

Core Data Structures:

We begin with a deep dive into computational complexity analysis, including Big-O notation and space-time tradeoffs, providing a mathematical framework to evaluate data structure efficiency. We will systematically examine:

- Linear structures (arrays, linked lists, stacks, queues) with implementations optimized for memory management
- Hierarchical architecture (binary trees, heaps, tree structures) emphasizing search optimization
- Associative containers (hash tables, dictionaries) focusing on collision resolution strategies
- Graph theory applications for network modeling and relationship mapping

Each chapter pairs mathematical theories with Python implementations, demonstrating how structural choices impact performance in security-critical systems. Memory allocation patterns and data encapsulation techniques receive particular attention for their role in preventing buffer overflows and memory corruption vulnerabilities.

Practical Applications:

The second half transforms theory into practice through security-focused case studies:

- Automating threat intelligence pipelines using queue-based processing architecture
- Implementing Merkle trees for blockchain-based log integrity verification
- Developing graph neural networks for malware kinship analysis
- Designing bloom filters for efficient network intrusion detection

Practical modules guide learners through deploying machine learning AI models using open-source tools, including:

- Fine-tuning LLMs for phishing email detection via attention mechanism optimization by analyzing content and metadata of incoming emails, processing headers, subject lines, and body. The model identifies suspicious language patterns, unusual requests, or inconsistencies that may indicate phishing attempts.
- Building RAG (Retrieval-Augmented Generation) systems security policy from an internal knowledge base that contains security policies, compliance, incident records or regulations.
- Containerizing AI workflows using Ollama for reproducible threat hunting by running each threat hunting task in its own isolated environment, reducing the risk of cross-contamination or interference from other processes

It concludes with recommended capstone projects that challenge learners to architect hybrid systems combining traditional data structures with neural networks for anomaly detection in encrypted traffic. Through this applied methodology, learners gain proficiency in designing secure, efficient data pipelines while understanding the theoretical underpinnings that make modern cybersecurity and AI systems robust against evolving threats.

Prerequisites:

A foundational understanding of Python or an equivalent introductory programming language.

1.2 The role of data structures in Cybersecurity, data science, and AI.

Let us now consider the pivotal role that data structures play across the domains of cybersecurity, data science, and artificial intelligence. Data structures are not merely abstract concepts; they are the very scaffolding upon which robust, efficient, and secure computational systems are built.

Importance of Data Structures

a) Foundation for efficient data handling:

To begin with, data structures serve as the foundational framework for organizing and managing data in any computing environment. By structuring data in logical, systematic ways – be it through arrays, linked lists, or more complex arrangements—we enable faster access, manipulation, and analysis. Efficient data handling is not just about speed; it is also about accuracy and reliability, ensuring that data can be processed correctly and consistently, even under heavy loads.

b) Critical for cybersecurity:

In the realm of cybersecurity, data structures are absolutely essential. Modern security systems must store and retrieve enormous volumes of data—network logs, user credentials, system configurations, and more—often in real time. The ability to quickly search, update, and analyze this data is crucial for detecting anomalies, identifying threats, and responding to incidents. For example, hash tables may be used to store and rapidly access user authentication details, while trees and graphs can model network topologies and relationships between devices or users, making it easier to spot suspicious patterns.

c) Key in data science and AI:

Finally, data structures are indispensable in data science and artificial intelligence. Machine learning algorithms rely heavily on efficient data structures such as arrays, linked lists, trees, and graphs to manage and manipulate large datasets. These structures allow for rapid data retrieval, which is essential for pattern recognition, predictive analytics, and the training of sophisticated models. Without effective data organization, the performance of AI systems would be severely hampered, and the insights gleaned from data would be both slower and less reliable.

In summary, a strong grasp of data structures is not just beneficial—it is essential for anyone seeking to excel in cybersecurity, data science, or artificial intelligence. By mastering these concepts, you empower yourself to build systems that are not only powerful and efficient, but also secure and intelligent.

Examples

a) **Arrays and linked lists:** Used in intrusion detection systems to store and analyze network traffic data

b) **Hash tables:** Facilitate quick lookup of user credentials or threat signatures.

c) **Trees and graphs:** Represent hierarchical or relational data, such as system configurations or network topologies, aiding in vulnerability analysis

Impact of Poor Implementation

a) **Vulnerabilities:** Insecure data structures can introduce risks, such as buffer overflow attacks

b) **Performance issues:** Inefficient data structures may slow down threat detection and response.

1.3 Introduction to the Cybersecurity landscape: threats & tools

The contemporary cybersecurity landscape is defined by an ever-evolving array of threats, sophisticated adversaries, and a constantly expanding toolkit for both attackers and defenders. Cyber threats are not only more frequent but also increasingly industrialized, with adversaries leveraging automation, artificial intelligence, and advanced social engineering techniques to bypass traditional defenses

1.3.1 Cybersecurity Threats

a) Advanced Persistent Threats (APTs): Sophisticated, long-term attacks often targeting high-value data
b) Ransomware and cryptojacking: Malicious software that encrypts data or hijacks systems for cryptocurrency mining
c) IoT and mobile device vulnerabilities: Expanding attack surfaces due to the proliferation of connected devices
d) Cloud and supply chain attacks: Exploiting weaknesses in third-party services and software
e) AI-powered attacks: Cybercriminals using AI to automate and optimize attacks

1.3.2 Key AI Tools for Cybersecurity

In the contemporary cybersecurity landscape, artificial intelligence has emerged as a cornerstone for both threat detection and response. Modern AI-powered tools are transforming how organizations defend their digital assets by enabling real-time monitoring, predictive analytics, and automated incident management. These tools are not only faster and more accurate than traditional methods but are also capable of adapting to rapidly evolving threats

AI-Driven Threat Detection and Analytics

a) Behavioral Analytics: AI systems such as User and Entity Behavior Analytics (UEBA) continuously analyze user activities, identifying deviations from normal patterns that may indicate compromised accounts, insider threats, or ongoing attacks. This capability is critical for detecting subtle, advanced threats that evade conventional security measures

b) Anomaly Detection: Machine learning models process vast amounts of network and log data, flagging unusual behavior—such as suspicious login attempts or unexpected data transfers—with high precision. These models continuously learn from new data, reducing false positives and improving detection rates

AI-Powered Security Assistants and Productivity Tools

a) Security Assistants: Generative AI assistants, such as Microsoft CoPilot and AccuKnox AI CoPilot, offer recommendations, summarize threat intelligence, and support analysts in making informed decisions. These tools help security teams focus on strategic tasks by automating routine activities and filtering out noise from alerts

b) Threat Intelligence Summarization: Large language models (LLMs) condense lengthy threat reports into actionable insights, enabling rapid response to emerging risks

Automated Incident Response Systems

a) Function: AI autonomously detects threats, quarantines malware, isolates systems, and documents incidents

Module 1: Learning Materials & References:

i. Jupyter Notebooks in VS Code
ii. Understanding How Data Structures Impact Cyber Security
iii. Best AI Cybersecurity Tools
iv. The Impact Of AI On Cybersecurity
v. How AI cybersecurity tools can protect your business
vi. How is AI Changing Cybersecurity in 2025?

Assessment: Module 1: Overview and Cybersecurity with AI Context

Part A: Quiz

1.1a What is the primary objective of Module #1?

a) To provide an introductory guide to Python programming.
b) To explore theoretical data structure concepts without practical application.
c) To rigorously explore data structure fundamentals while bridging theoretical computer science with cutting-edge applications in cybersecurity and artificial intelligence.
d) To focus solely on AI applications in cybersecurity.

1.2a Which of the following is NOT listed as one of the four main categories of core data structures that are systematically examined?

a) Linear structures (arrays, linked lists, stacks, queues).
b) Hierarchical architecture (binary trees, heaps, tree structures).
c) Relational databases (SQL, NoSQL).
d) Associative containers (hash tables, dictionaries).
e) Graph theory applications (for network modeling).

1.3a Understanding memory allocation patterns and data encapsulation techniques is highlighted as particularly important for security because they help prevent which two specific vulnerabilities?

a) Phishing attacks and ransomware.
b) Buffer overflows and memory corruption vulnerabilities.
c) SQL injection and cross-site scripting.
d) Denial of service and insider threats.

1.4a Which of these is listed as a practical cybersecurity application where data structures and/or AI are applied?

a) Developing new encryption algorithms.
b) Managing physical security access control systems.
c) Designing secure network hardware components.
d) Implementing Merkle trees for blockchain-based log integrity verification.

1.5a Why are data structures considered essential in the field of cybersecurity?

a) They are only useful for storing large amounts of data, which isn't always necessary for security.
b) Modern security systems must store, retrieve, and analyze enormous volumes of data (like network logs, credentials) often in real time, and efficient data structures enable the speed required for detection and response.
c) They simplify the process of writing malicious code.
d) Data structures are primarily for data science and AI, not security.

1.6a How are Hash tables specifically used in cybersecurity contexts?

a) To represent network topologies.
b) To analyze network traffic data.
c) To model relationships between users.
d) To store and rapidly access user authentication details or facilitate quick lookup of user credentials or threat signatures.

1.7a Poor implementation of data structures in security systems can result in two negative impacts?

a) Increased system complexity and higher operational costs.
b) Difficulty in integrating with AI systems.
c) Introduction of vulnerabilities (like buffer overflows) and performance issues (slowing threat detection).
d) Reduced compatibility with different programming languages.

1.8a Which of the following is identified as a type of cybersecurity threat?

a) Natural disasters affecting data centers.
b) Hardware failures causing data loss.
c) Ransomware and cryptojacking.
d) Software licensing violations.

1.9a How has Artificial Intelligence emerged as a cornerstone in the contemporary cybersecurity landscape?

a) By replacing human analysts entirely in all security operations.
b) By enabling real-time monitoring, predictive analytics, and automated incident management, being faster, more accurate, and adaptable than traditional methods.
c) By making security systems significantly more complex and harder to manage.
d) By increasing the frequency of false positive alerts.

Part B: Short Answer

1.1b Describe how a Large Language Model (LLM) like the ones run using Ollama can assist in detecting phishing emails.
1.2b How do trees and graphs help with cybersecurity?
1.3b How do behavioral analytics use AI to assist in cybersecurity threat detection?

Part C: Python Exercises

A Refresher on Key Python Programming Principles

For the following exercises you are required to follow Python Best Practices in Appendix A, such as documenting your functions with comprehensive Docstring, applying gradual typing to functions, parameters, constants, and variables. Also, every Python script must be self-documented using docstrings, with appropriate headers.

Program Exercises:

1.1c list_is_multiple function

Write a python function **list_is_multiple** with two parameters list k, list l. The function will return True if list k is a multiple of list l, for some integers.

Examples:

```
print(list_is_multiple([1, 2, 1, 2, 1, 2], [1, 2]))    # True (repeated 3 times)
print(list_is_multiple([1, 2, 3, 1, 2, 3], [1, 2, 3])) # True (repeated 2 times)
print(list_is_multiple([1, 2, 3, 4], [1, 2])) # False (not a clean repetition)
print(list_is_multiple([1, 2], [1, 2, 3])) # False (k is shorter than l)
print(list_is_multiple([], [1, 2])) # True (empty list is 0 times any list)
print(list_is_multiple([1, 2], [])) # False (can't multiply by empty list)
```

1.2c Frequency Distribution of Words Function

Write a Python function **count_words** that counts the frequency distribution of words within a given text string.

Examples:

```
text = "hello world hello python world"
result = count_words(text)
print(result)  # {'hello': 2, 'world': 2, 'python': 1}

text = "It was the best of times, it was the worst of times,"
result = count_words(text)
print(result) # {'it': 2, 'was': 2, 'the': 2, 'best': 1, 'of': 2, 'times': 2, 'worst': 1}
```

1.3c Pseudocode Selection Sort Function

Describe, in pseudocode, a function that performs selection sort on an array of integers. Then, implement the function in Python and briefly compare the two approaches. By convention, a class name should be capitalized.

1.4c A Whale Class

```
Description: A simple Whale class
Name: Your Name
Course: CSIS 252
Professor: Ạ Nuzen
Tasks:

1. Create a class called Whale that represents a whale with the following attributes:
- name (str): The name of the whale.
- species (str): The species of whale.
- length (float): The length of the whale in meters.
- weight (float): The weight of the whale in kilograms.
You will also implement the following methods:
- __str__(): Returns a string representation of the whale in the format "name,
species=species, length=length, weight=weight".
- __lt__(other): Compares two whales based on their names.
- __eq__(other): Compares two whales based on their names.
- Create setters and getters for these attributes.


2. In a separate python script called test_whale.py
create a list of whales and output them to the console.

compare the whales using the comparison methods you implemented.
Make sure to match the sample output.

Sample output:

List of Whales:
Willy, species=Orca, length=6.0, weight=5400
Balaenoptera, species=Blue Whale, length=30.0, weight=200000
Belly, species=Blue Whale, length=30.0, weight=180000
Patches, species=Humpback Whale, length=15.0, weight=40000
Shamu, species=Orca, length=20.0, weight=6000

List of Whales sorted by Name:
Balaenoptera, species=Blue Whale, length=30.0, weight=200000
Belly, species=Blue Whale, length=30.0, weight=180000
Patches, species=Humpback Whale, length=15.0, weight=40000
Shamu, species=Orca, length=20.0, weight=6000
Willy, species=Orca, length=6.0, weight=5400

List of Whales sorted by Length:
Willy, species=Orca, length=6.0, weight=5400
Patches, species=Humpback Whale, length=15.0, weight=40000
Shamu, species=Orca, length=20.0, weight=6000
Balaenoptera, species=Blue Whale, length=30.0, weight=200000
Belly, species=Blue Whale, length=30.0, weight=180000

Is Willy less than Shamu? : False

The End …
```

Module 2: Big-O Notation & Complexity Analysis profiling optimization

Let us now turn our attention to one of the foundational concepts in computer science: Big-O notation and its critical role in analyzing and optimizing algorithms. Big-O notation provides a standardized mathematical framework for describing the efficiency of algorithms, particularly how their performance scales as the size of input data grows.

2.1 Time and space complexity concepts

Understanding and applying Big-O notation is not merely an academic exercise. It is vital for designing systems that remain efficient and scalable as data volumes grow. Whether you are developing algorithms for cybersecurity, data science, or artificial intelligence, a firm grasp of complexity analysis ensures that your solutions are robust, performant, and future-proof

2.1.2 Key Learning Areas:

Time Complexity:

Time complexity is a concept in computer science that provides a systematic way to assess and compare the efficiency of algorithms as the size of their input data grows. Specifically, time complexity quantifies the number of operations an algorithm must perform to complete a task, expressed as a function of the input size, typically denoted as $\boldsymbol{n}$.

The goal is then to measure how the runtime of an algorithm increases as the input size grows. The Big-O Notation expresses the upper bound of an algorithm's runtime in the worst-case scenario (e.g., O(n), O(n^2), O(log n)). So that we can predict the scalability and efficiency of algorithms.

Graph-1 illustrates the asymptotic growth patterns comparison between Big O functions from constant time to exponential time.

The graph plots **T(n)** (operations) against **n** (input size) using logarithmic scaling to reveal key characteristics:

- **O(1)**: Flat line demonstrating constant time complexity $T(n)=C$
- **O(log n)**: Gentle curve following $T(n)=\log_2 n$
- **O(n)**: Straight line with slope 1 $(T(n)=n)$
- **O(n log n)**: Curved line following $T(n)=n \log_2 n$
- **O(n^2)**: Parabolic curve $(T(n)=n^2)$
- **O(n^3)**: Cubic growth $(T(n)=n^3)$
- **O(2^n)**: Hyper-exponential surge $(T(n)=2^n)$

Graph – 1 Big O Growth Rate Comparison

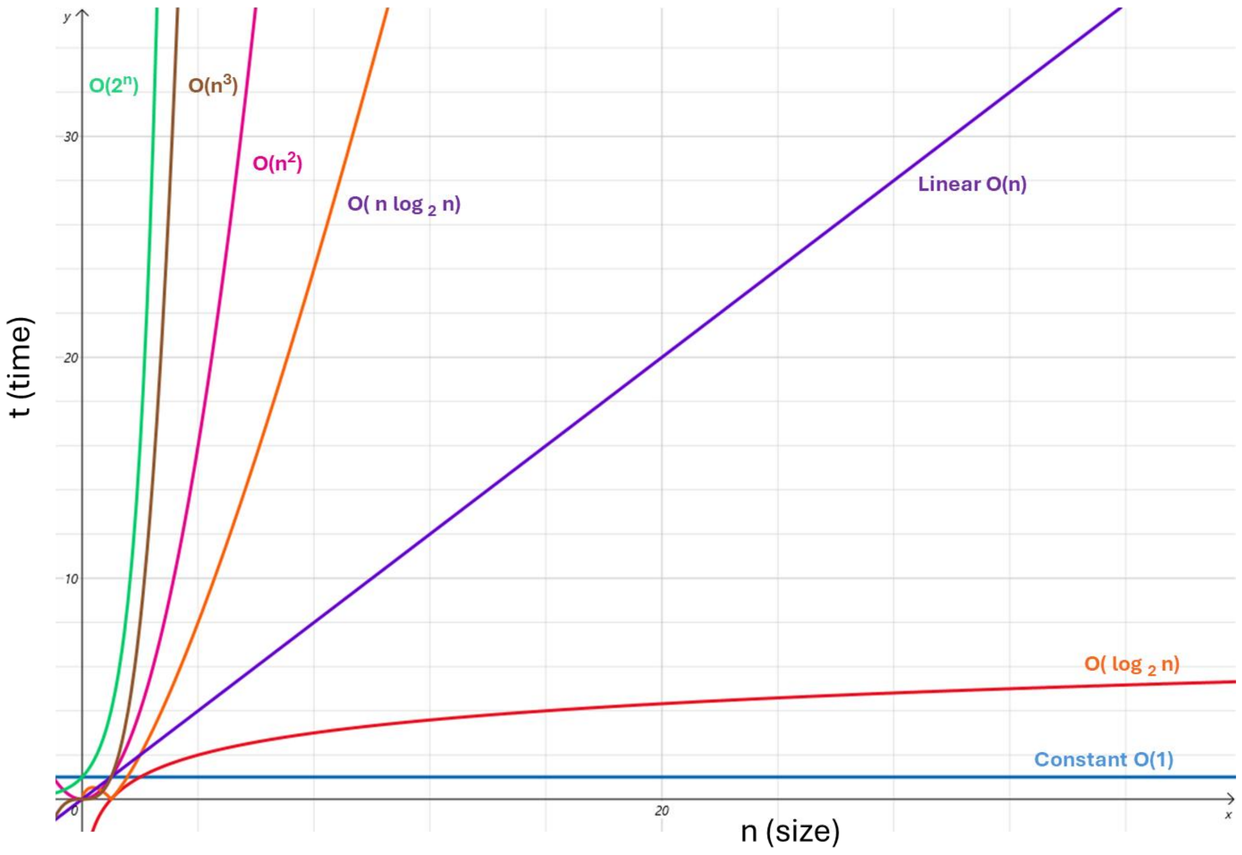


Table-2 illustrates the feasibility of various time complexities as the input size **N** increases. This table is essential for understanding which algorithms remain practical as data volumes scale, and which become prohibitively slow or resource intensive.

Notation	Shape on Graph	Example Use Case	Feasibility
O(1)	Flat line	Array index access	Excellent
O(log n)	Very slow rise	Binary search	Great
O(n)	Straight line	Linear search	Good
O(n log n)	Slightly curved upward	Merge sort, Quick sort	Acceptable
$O(n^2)$	Steep curve	Nested loops	Caution
$O(2^n)$	Very steep curve	Subset generation	Poor
O(n!)	Extremely steep curve	Permutations	Avoid if possible

Table – 2 Time complexity feasibility

Space Complexity:

Space complexity is a concept in computer science that evaluates how the memory usage of an algorithm grows as the size of the input increases. It is crucial to consider space complexity because efficient memory management is essential—not just for large datasets, but also for systems with constrained resources such as embedded devices, mobile platforms, or cloud environments with strict limits.

Purpose of Space Complexity Analysis

The primary purpose of analyzing space complexity is to ensure that algorithms are memory efficient. As datasets grow larger and systems become more complex, the amount of memory an algorithm consumes can become a critical bottleneck. By understanding and optimizing space complexity, we can prevent out-of-memory errors, reduce costs, and improve overall system performance.

Measuring Space Complexity

Space complexity is measured by the amount of additional memory an algorithm requires relative to the input size, typically denoted as **n**. This measurement considers both the input data itself, and any auxiliary space used by the algorithm, such as temporary variables, data structures, or recursive call stacks.

Big-O Notation for Space Complexity

Just as with time complexity, Big-O notation is used to express space complexity. Common examples include:

- **O(1):** The algorithm uses a fixed amount of additional memory, regardless of the input size.
- **O(n):** The algorithm's memory usage grows linearly with the input size.
- **O(n^2):** The algorithm's memory usage grows quadratically as the input size increases.

2.2 Analyzing and comparing algorithms, best, worst, average case analysis.

The process of analyzing and comparing algorithms enables us to make informed, evidence-based decisions about which algorithm is best suited for a particular task. This analysis is conducted under a variety of scenarios—most notably, the best, worst, and average cases—each of which provides unique insight into an algorithm's behavior and efficiency.

2.2.1 Key Learning Areas:

Best Case Analysis:

- This examines the minimum resource usage (time or memory) an algorithm can achieve, typically under ideal conditions. For example, in a linear search, the best case occurs when the desired element is found at the very beginning of the list, requiring only a single comparison.

Worst Case Analysis:

- This determines the maximum resource usage an algorithm might require, which is crucial for guaranteeing that an algorithm will complete within acceptable limits even under the most unfavorable conditions. In linear search, the worst case happens when the element is not present or is at the end of the list, requiring the algorithm to examine every element.

Average Case Analysis:

- This considers the expected resource usage over all possible inputs, providing a more realistic assessment of an algorithm's performance in typical use. Calculating the average case often involves probabilistic analysis and requires knowledge or assumptions about the distribution of possible inputs. For example: On average, a linear search will take O(n/2) time, which simplifies O(n).

Practical Implications:

- By understanding best, worst, and average case analysis, you will be equipped to make informed design decisions, optimize code for efficiency and reliability, and confidently implement algorithms in domains such as cybersecurity, data science, and artificial intelligence. This analysis for real-time threat detection and response. Scalability ensures systems can handle increasing data volumes without performance degradation.

2.3 Profiling and optimizing Python code

Profiling and optimization are essential practices in modern software development, especially as applications grow in scale and complexity. The objective of this section is to bridge the gap between theoretical complexity analysis—such as Big-O notation—and practical, real-world code improvement by profiling and optimizing Python programs.

By the end of this module, you will be able to apply complexity analysis directly to your code, using profiling tools to measure and identify performance bottlenecks. This process allows you to make informed, data-driven decisions about where and how to optimize, ensuring that your programs are both efficient and scalable.

2.3.1 Key Learning Areas:

Profiling is the systematic process of measuring various aspects of a program's performance, including execution time and memory usage. It provides quantitative data about how different parts of your code behave, revealing which sections consume the most resources or are responsible for slowdowns.

2.3.1a Profiling Tools:

Several robust tools are available for profiling Python code, each suited to different aspects of performance analysis:

I. cProfile:

This is a built-in Python module that provides detailed statistics about function calls, execution times, and call frequencies. It is ideal for obtaining a comprehensive overview of your program's performance and is easy to use with minimal setup

II. timeit:

Designed for measuring the execution time of small code snippets, this tool is particularly useful for comparing the efficiency of alternative implementations or micro-optimizations

III. memory_profiler:

This tool tracks memory consumption line-by-line, helping you identify memory leaks and inefficient memory usage patterns. It is invaluable for optimizing programs that handle large datasets or run in memory-constrained environments

By leveraging tools like cProfile, timeit, and memory_profiler, you can transform theoretical complexity analysis into actionable insights, resulting in programs that are both elegant and efficient.

2.3.1b Optimization Techniques:

- **Algorithm Selection:** Choose algorithms with lower time/space complexity.
- **Data Structures:** Use appropriate data structures (e.g., dictionaries for fast lookups).
- **Code Refactoring:** Simplify logic, avoid redundant calculations, and use built-in functions.
- **Parallelism:** Leverage multi-threading or multi-processing for CPU-bound tasks.
- **Caching:** Store results of expensive function calls to avoid repeated computation.

2.3.2 Best Practices:

- **Profile Before Optimizing:** Focus optimization efforts where they matter most.
- **Write Readable Code:** Balance optimization with code maintainability.
- **Test Thoroughly:** Ensure optimizations do not introduce bugs or security vulnerabilities.

2.3.4 Security Considerations:

- **Efficiency:** Fast algorithms enable timely threat detection and response.
- **Resource Management:** Optimized code reduces the risk of denial-of-service attacks by minimizing resource consumption.

Module 2: Learning Materials & References:

i. Big O Notation — with Examples in Python
ii. Top 7 Python Profiling Tools for Performance
iii. Profiling and optimizing your Python code
iv. Big O Notation and Time Complexity Guide: Intuition and Math
v. A Comprehensive Guide to Profiling in Python

Assessment: Module 2: Big-O Notation & Complexity Analysis profiling optimization

Part A: Quiz

2.1a.What is the main purpose of Big-O notation in computer science according to the sources?

A) To write elegant code
B) To predict the performance scalability and efficiency of algorithms
C) To generate permutations
D) To track memory consumption line-by-line

2.2a Time complexity measures how the runtime of an algorithm increases as which factor grows?

A) The number of functions used
B) The amount of available memory
C) The size of the input data
D) The readability of the code

2.3a Which concept evaluates how the memory usage of an algorithm grows as the size of the input increases?

A) Time Complexity
B) Profiling
C) Big-O Notation
D) Space Complexity

2.4a Based on the feasibility table (Table 2) in the sources, which time complexity is described as "Excellent"?

A) O(n!)

B) O(n^2)
C) O(1)
D) O(log n)

2.5a. Which type of algorithm analysis determines the maximum resource usage under the most unfavorable conditions?

A) Best Case Analysis
B) Average Case Analysis
C) Worst Case Analysis
D) Practical Implications Analysis

2.6a Which Python profiling tool provides detailed statistics about function calls and execution times?

A) timeit
B) cProfile
C) memory_profiler
D) Big-O

2.7a One of the optimization techniques mentioned is using appropriate:

A) Data Structures
B) Profiling Tools
C) Case Analyses
D) Big-O Notations

Part B: BIG O Ranking

2.1b Best to Worst

Most efficient to least efficient for large input sizes.

A. $5n^2 + 3n + 100$

B. $2^n + 10n + 50$

C. $10 \log n + 5$

D. $1000n + 500$

E. $8n \log n + 2n + 15$

2.2b Worst to Best

Least efficient to most efficient for large input sizes.

A. $O(n^{32})$

B. $O(\frac{n!}{2^n})$

C. $O(\sqrt{n} \times \log n)$

D. $O(\frac{n^2}{\log n})$

E. $O(2^{\sqrt{n}})$

2.3b. Python Code Complexity

```python
def linear_search(arr, target):

    """Search for target in unsorted array"""
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

def count_occurrences(arr, target):
    """Count how many times target appears"""
    count = 0
    for item in arr:
        if item == target:
            count += 1
    return count
```

```python
def binary_search(arr, target):
    """Search in sorted array"""
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1
def bubble_sort(arr):
    """Bubble sort implementation"""
    n = len(arr)

    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

2.4b What is the time complexity of function linear search for an array of n elements?
2.5b What is the time complexity of function count_occurences for an array of n elements?
2.6b What is the time complexity of function binary_search for a sorted array of n elements?
2.7b What is the time complexity of function Bubble_sort for an array of n elements?

Part C: Profiling and Optimization Python Code

2.1c Improving performance with profiling

Problem: You have a slow Python program and need to identify bottlenecks. Comment on these profiling approaches, pros and cons.

```python
import time
import cProfile
import timeit
from memory_profiler import profile

# Method 1: Basic timing
def method1_basic_timing():
    def slow_function():
        result = []
        for i in range(100000):
            result.append(i ** 2)
        return result

    start_time = time.time()
    data = slow_function()
    end_time = time.time()
    print(f"Execution time: {end_time - start_time:.4f} seconds")

# Method 2: Using timeit
def method2_timeit():
    setup_code = "result = []"
    test_code =
for i in range(100000):
    result.append(i ** 2)
"""
    execution_time = timeit.timeit(test_code, setup=setup_code, number=10)
    print(f"Average time: {execution_time/10:.4f} seconds")

# Method 3: cProfile
def method3_cprofile():
    def slow_function():
        result = []
        for i in range(100000):
            result.append(i ** 2)
        return sum(result)
      Profile.run('slow_function()')
```

```python
# Method 4: Memory profiling
@profile
def method4_memory():
    result = []
    for i in range(100000):
        result.append(i ** 2)
    return result

def slow_function():
    result = []
    for i in range(100000):
        result.append(i ** 2)
    return result
```

Module 3: Lists, Tuples, and NumPy Arrays

In cybersecurity operations, effective data management is paramount to successful threat detection and incident response. In the module, we'll examine how Python's fundamental data structures—lists, tuples, and arrays—serve as the backbone for processing and analyzing security logs, network packets, and forensic data. These structures aren't merely academic concepts; they're the workhorses that enable security analysts to transform raw data streams into actionable intelligence.

3.1 Review and advanced usage of lists and tuples

Let’s begin by revisiting the fundamental concepts of lists and tuples in Python, before moving into their advanced usage.

Both lists and tuples are sequence data types, meaning they store collections of items in a specific order. However, their properties and use cases differ in important ways.

A list is an ordered collection of items, enclosed in square brackets []. Lists are *mutable*—that is, you can add, remove, or modify elements after creation. This makes lists ideal for situations where the data set is expected to change, such as when building dynamic datasets or collections that grow or shrink during program execution.

```
colors = ["red", "green", "blue"]
colors.append("yellow")  # Adding an element
colors[1] = "emerald"    # Modifying an element
```

Lists are commonly used for homogeneous data, meaning all elements are of the same type or represent the same concept, but Python allows lists to contain mixed types as well.

Tuple Heterogeneous Data Paradigm

A tuple, on the other hand, is an ordered collection of items, enclosed in parentheses (). Tuples are *immutable*—once created, you cannot add, remove, or change elements. This immutability makes tuples suitable for storing fixed collections of items, such as coordinates, configuration settings, or any data that should not be altered after creation.

```
coordinates = (10, 20, 30)
# coordinates[1] = 25  # This would raise an error
```

Tuples are often used for heterogeneous data, where each element may represent a different attribute or type. Operationally, tuples use less space and have better time complexity than lists. A small Python code fragment below illustrates that on average the memory required for a tuple is 20% less than a list, however, the speed execution is much faster, almost 5 times faster than list.

```python
import timeit
import sys

# Timing the creation of a list and a tuple
list = [1, False, "list",3.5]
tuple = (1, False,"tuple", 3.5)

# Timing the creation of a list and a tuple
# Create list 100 million times
# Create tuple 100 million times
SIZE= 20_000_000
print(f"Creating a list and a tuple {SIZE:,d} times.")
list_time= timeit.timeit("[1, 2, 3]", number=SIZE)
print(f"Creation list time: {list_time:.2f} seconds")
tuple_time = timeit.timeit("(1, 2, 3)", number=SIZE)
print("Creation tuple time: {tuple_time:.2f} seconds")
print("How much faster is tuple compared to list? {list_time / tuple_time:.2f} times faster")
print("Size comparison:", "List size:", sys.getsizeof(list),
                              "Tuple size:", sys.getsizeof(tuple))
```

```
Creating a list and a tuple 20,000,000 times.
Creation list time: 1.08 seconds
Creation tuple time: 0.26 seconds
How much faster is tuple compared to list? 4.17 times faster
Size comparison: List size: 88 Tuple size: 72
```

Tuple Memory Efficiency

When we examine the fundamental nature of cybersecurity data, we quickly recognize that security events are inherently heterogeneous. Consider a network intrusion event: it contains a timestamp (datetime object), source IP address (string), port number (integer), protocol type (string), and threat severity (integer or enumerated value). This natural heterogeneity makes tuples an ideal choice for representing security events, as each position within the tuple carries semantic meaning corresponding to a specific attribute.

This structural approach provides what we call "positional semantics"—the first element is always the timestamp, the second always the source IP, and so forth. This predictability is crucial in security operations where data integrity and consistent interpretation can mean the difference between detecting a breach and missing it entirely.

The memory efficiency of tuples stems from their immutable nature and optimized internal structure. When Python creates a tuple, it allocates a fixed-size memory block based on the number of elements. Lists, conversely, must maintain additional overhead to support dynamic resizing operations.

Let's demonstrate this principle with a practical cybersecurity example below where we simulate a security event, with date time stamp, IP address, port, and event flag, and compare the actual memory allocation for the event using list vs tuple.

```python
import sys

# Security event as a tuple (immutable)
security_event_tuple = ('2025-06-01 14:23:15', '192.168.1.100', 22, 'SSH',
'CRITICAL')

# Same data as a list (mutable)
security_event_list = ['2025-06-01 14:23:15', '192.168.1.100', 22, 'SSH',
'CRITICAL']

print(f"Tuple memory usage: {sys.getsizeof(security_event_tuple)} bytes")
print(f"List memory usage: {sys.getsizeof(security_event_list)} bytes")
print(f"Memory savings: {(1 -
sys.getsizeof(security_event_tuple)/sys.getsizeof(security_event_list))*100:.1f}%")
```

```
Tuple memory usage: 80 bytes
List memory usage: 104 bytes
Memory savings: 23.1%
```

This 20+% memory reduction becomes significant when processing millions of security events. In a typical enterprise environment generating 100,000 security events per hour, this efficiency translates to substantial memory savings across your security infrastructure.

Tuple Speed Advantage

The dramatic performance advantage of tuples—approximately five times faster than lists for certain operations—derives from several architectural optimizations. Since tuples are immutable, Python can implement aggressive optimizations during creation, access, and iteration.

Considering this performance comparison in the context of log analysis, in the below code fragment, we simulate the creation of 1 million log events, particularly those with 'failed_login' flag.

```python
import time

# Generate sample security events
num_events = 1_000_000

# Create tuples (representing immutable security events)
start_time = time.time()
tuple_events = [
    (f'192.168.1.{i%255}', 'failed_login', i, 'ssh')
    for i in range(num_events)
]
tuple_creation_time = time.time() - start_time

# Create equivalent lists
start_time = time.time()
list_events = [
    [f'192.168.1.{i%255}', 'failed_login', i, 'ssh']
    for i in range(num_events)
]
list_creation_time = time.time() - start_time

print(f"Tuple creation: {tuple_creation_time:.4f} seconds")
print(f"List creation: {list_creation_time:.4f} seconds")
print(f"Performance ratio: {list_creation_time/tuple_creation_time:.1f}x faster")
```

```
Tuple creation: 0.5198 seconds
List creation: 1.2795 seconds
Performance ratio: 2.5x faster
```

3.2 Arrays with NumPy

Consider a typical enterprise environment: firewalls generating millions of connection records daily, intrusion detection systems producing thousands of alerts hourly, and network monitoring tools capturing terabytes of packet metadata. The conventional approach of processing this data sequentially—one record at a time—simply cannot scale to meet modern threat detection requirements.

This is where NumPy's array-based computing model revolutionized our approach to security analytics. Rather than thinking in terms of individual log entries or discrete network events, NumPy enables us to conceptualize entire datasets as mathematical objects that can be manipulated through vectorized operations.

Basic Example: Creating a NumPy Array

```python
import numpy as np

# Create a 1D array (like a list)
array1 = np.array([10, 20, 30, 40])
print("1D Array:", array1)
# 1D Array: [10 20 30 40]
# Create a 2D array (like a matrix)
array2 = np.array([[1, 2, 3], [4, 5, 6]])
print("2D Array:\n", array2)
# 2D Array:
# [[1 2 3]
#  [4 5 6]]

#array of zeros or ones
array_zeros = np.zeros((2, 3))  # 2x3 array of zeros
print("Array of Zeros:\n", array_zeros)
# Array of Zeros:
# [[0. 0. 0.]
#  [0. 0. 0.]]
array_ones = np.ones((2, 3))    # 2x3 array of ones
print("Array of Ones:\n", array_ones)
# Array of Ones:
# [[1. 1. 1.]
#  [1. 1. 1.]]
# Create an array with a range of values
array_range = np.arange(0, 10, 2)  # Start at 0, end before 10, step by 2
print("Array with Range of Values:", array_range)
# Array with Range of Values: [0 2 4 6 8]
# Create an array with random values
array_random = np.random.rand(2, 3)  # 2x3 array of random values
print("Array with Random Values:\n", array_random)
# Array with Random Values:
# [[0.5488135  0.71518937 0.60276338]
#  [0.54488318 0.4236548  0.64589411]]
# Create an identity matrix
identity_matrix = np.eye(3)  # 3x3 identity matrix
```

```
print("Identity Matrix:\n", identity_matrix)
# Identity Matrix:
# [[1. 0. 0.]
#  [0. 1. 0.]
#  [0. 0. 1.]]
# create a array of random normal distribution
array_normal = np.random.randn(2, 3)  # 2x3 array of random values from normal
distribution
print("Array with Normal Distribution:\n", array_normal)
# Array with Normal Distribution:
# [[ 0.49671415 -0.1382643   0.64768854]
#  [ 1.52302986 -0.23415337 -0.23413696]]
```

Vectorization in Security Context

Let's illustrate this paradigm shift with a concrete example. Suppose we're analyzing bandwidth utilization patterns to detect potential data exfiltration:

```
import numpy as np
import time

# Traditional approach: Sequential processing
def analyze_bandwidth_sequential(connection_logs):
    """Traditional sequential analysis of bandwidth patterns"""
    suspicious_connections = []
    for log in connection_logs:
        if log['bytes_out'] > (log['bytes_in'] * 10):  # High outbound ratio
            if log['duration'] < 60:  # Short duration
                suspicious_connections.append(log)
    return suspicious_connections

# NumPy vectorized approach
def analyze_bandwidth_vectorized(bytes_out, bytes_in, durations):
    """Vectorized analysis using NumPy arrays"""
    # All calculations performed simultaneously across entire dataset
    outbound_ratio = bytes_out / (bytes_in + 1)  # Avoid division by zero
    suspicious_mask = (outbound_ratio > 10) & (durations < 60)
    return np.where(suspicious_mask)[0]  # Return indices of suspicious connections
```

The vectorized approach doesn't just offer cleaner code—it provides orders of magnitude performance improvement by leveraging optimized C implementations and SIMD (Single Instruction, Multiple Data) operations available in modern processors.

Memory Efficiency and Data Type Optimization

NumPy's homogeneous array structure provides significant memory advantages over Python's heterogeneous data structures. In security applications, this translates to the ability to hold larger datasets in memory, enabling more comprehensive analysis:

```python
import numpy as np
# Efficient storage of network flow data
network_flows = np.dtype([
    ('timestamp', 'datetime64[s]'),
    ('src_ip', 'U15'),           # Unicode string, max 15 characters
    ('dst_ip', 'U15'),
    ('src_port', 'u2'),          # Unsigned 16-bit integer
    ('dst_port', 'u2'),
    ('protocol', 'u1'),          # Unsigned 8-bit integer
    ('bytes_transferred', 'u8'), # Unsigned 64-bit integer
    ('duration', 'f4')           # 32-bit float
])

# Create structured array for efficient storage and processing
flow_data = np.zeros(1000000, dtype=network_flows)
```

This structured approach reduces memory footprint while maintaining type safety—critical factors when processing gigabytes of security data in memory-constrained environments.

Statistical Analysis for Anomaly Detection

NumPy's statistical functions enable sophisticated anomaly detection algorithms that would be computationally prohibitive with traditional Python data structures.

Time Series Analysis for Security Monitoring

Security monitoring inherently involves time series data—connection rates, alert frequencies, bandwidth utilization over time. NumPy provides a computational foundation for sophisticated temporal analysis.

3.3 Processing network packages and log entries

Network packet analysis and log processing represent the intersection of high-frequency data ingestion and real-time analytical requirements. A single gigabit network interface can process over 1.48 million packets per second at maximum theoretical throughput. Traditional processing approaches—parsing each packet individually, storing in relational databases, then querying for analysis—introduce latencies incompatible with modern threat detection requirements.

Network packets contain structured information ideal for NumPy's array-based processing. Consider analyzing TCP connection patterns for potential port scanning detection.

Security logs from multiple sources require correlation to identify coordinated attacks or system compromises. NumPy's broadcasting capabilities enable efficient cross-correlation analysis.

Modern cybersecurity demands real-time processing capabilities. NumPy arrays can be integrated into streaming architectures for continuous analysis.

The Computational Security Paradigm

The integration of NumPy arrays into cybersecurity operations represents more than technical optimization—it embodies a fundamental shift toward computational thinking in security analysis. Rather than processing individual events sequentially, we conceptualize security data as mathematical objects amenable to vectorized operations, statistical analysis, and pattern recognition algorithms.

This paradigm enables security operations centers to achieve several critical capabilities: real-time threat detection across high-volume data streams, sophisticated statistical anomaly detection, efficient correlation of events across multiple data sources, and scalable processing architectures that grow with organizational security needs.

The performance gains—often orders of magnitude improvement over traditional approaches—translate directly into enhanced security postures: faster threat detection, more comprehensive analysis coverage, and reduced infrastructure costs through efficient resource utilization. In an era where cyber threats evolve rapidly and attack volumes continue to escalate, these computational advantages become essential components of effective cybersecurity defense strategies.

Module 3: Learning Materials & References:

i. Python Lists vs Tuples
ii. NumPy Getting Started
iii. NumPy: the absolute basics for beginners

Assessment: Module 3: Lists, Tuples, and NumPy Arrays

Part A: Quiz

3.1a Which of the following is a key difference between Python lists and tuples?

A. Lists are ordered collections enclosed in parentheses (), while tuples are ordered collections enclosed in square brackets []
B. Lists are mutable (can be changed after creation), while tuples are immutable (cannot be changed after creation).
C. Lists are primarily used for heterogeneous data, while tuples are used for homogeneous data.
D. Tuples are generally slower for creation and access compared to lists

3.2a Which data structure is recommended in the source for representing security events that contain different attributes like timestamp, IP address, and port number, and why?

A. Lists, because they are mutable and can store mixed data types.
B. Lists, because they have better time complexity and use less space.
C. Tuples, because of their immutable nature and support for heterogeneous data with positional semantics, make them ideal for fixed event structures.
D. Tuples, because they are mutable and easy to modify.

3.3a What is the primary reason why tuples are more memory efficient than lists?

A. They store fewer elements on average
B. They automatically compress the data they store.
C. They only store numerical data, which requires less memory.
D. Their immutable nature allows Python to allocate a fixed-size memory block without needing additional overhead for dynamic resizing.

3.4a In the context of using NumPy for security analytics, what is the primary benefit of "vectorization"?

A. It makes the code shorter and easier to read.
B. It allows calculations to be performed simultaneously across an entire dataset, leveraging optimized low-level operations for significant performance improvement.
C. It ensures that all data elements are of the same type.
D. It automatically detects anomalies in the data.

3.5a Approximately how much faster is tuple creation compared to list creation for a large number of items?

A. About 1.5 times faster.
B. About 2.5 times faster.
C. About 3.5 times faster.
D. About 4-5 times faster

3.6a The source describes the integration of NumPy arrays into cybersecurity operations as embodying a fundamental shift towards what kind of paradigm?

A. A relational database paradigm.
B. A computational thinking paradigm, viewing data as mathematical objects for vectorized operations and pattern recognition.
C. A sequential processing paradigm.
D. A manual analysis paradigm.

Part B: Short Answers

3.1b. Why are tuples considered ideal for representing security events like network intrusions?

3.2b. Approximately how much memory can be saved by using tuples compared to lists for representing security events?

3b. What type of advanced security analysis is enabled by NumPy's statistical functions?

Part C: Python Exercises

3.1c Security Event Analysis with Tuples

Scenario: You are a junior security analyst tasked with reviewing a simplified log of security events. Each event is recorded as a tuple containing the event ID, timestamp, event type, and status. You must identify and report on "failed" login attempts or other "fail" events.

Task:

Define Sample Data: Create a list of tuples, each representing a security event. Each event tuple should have the following structure: (event_id: int, timestamp: str, event_type: str, status: str)

Here are some sample events you can use, but feel free to add more:

```
(101, "2025-06-01 10:00:05", "Login", "success")
(102, "2025-06-01 10:00:10", "Login", "fail")
(103, "2025-06-01 10:00:15", "File Access", "success")
(104, "2025-06-01 10:00:20", "Login", "fail")
(105, "2025-06-01 10:00:25", "System Update", "success")
(106, "2025-06-01 10:00:30", "Login", "fail")
(107, "2025-06-01 10:00:35", "Logout", "success")
(108, "2025-06-01 10:00:40", "Network Connection", "fail")
```

Implement Analysis Function:

- Write a Python function named analyze_security_events That takes one argument: event_logs (a list of event tuples) and outputs the processed events.
- Expected Output Format (for each failed event):

[FAILED EVENT] Event ID: [event_id], Timestamp: [timestamp], Type: [event_type]

Sample Output (based on sample data):

```
[FAILED EVENT] Event ID: 102, Timestamp: 2025-06-01 10:00:10, Type: Login
[FAILED EVENT] Event ID: 104, Timestamp: 2025-06-01 10:00:20, Type: Login
[FAILED EVENT] Event ID: 106, Timestamp: 2025-06-01 10:00:30, Type: Login
[FAILED EVENT] Event ID: 108, Timestamp: 2025-06-01 10:00:40, Type: Network
Connection
```

3.2c Security Event Analysis with NumPy

Repeat PE3.1 with NumPy.

Module 4: Dictionaries, Sets, and Hash Tables in password management

A dictionary is a data structure that stores a collection of data as key-value pairs. Each key must be unique within the dictionary, and each key maps to a single value. The keys are typically simple data types (like integers or strings), while the values can be of any type. Dictionaries allow for efficient insertion, deletion, and lookup of values based on their keys.

Whereas a hash table is a data structure that implements the dictionary. Hash table uses a hash function to compute an index (hash code) from a key, which points to a location (bucket or slot) in an underlying array where the value is stored. This design enables average-case constant time complexity for insert, delete, and lookup operations. Table 4-0 illustrates the implementation of dictionary and set using Hash table.

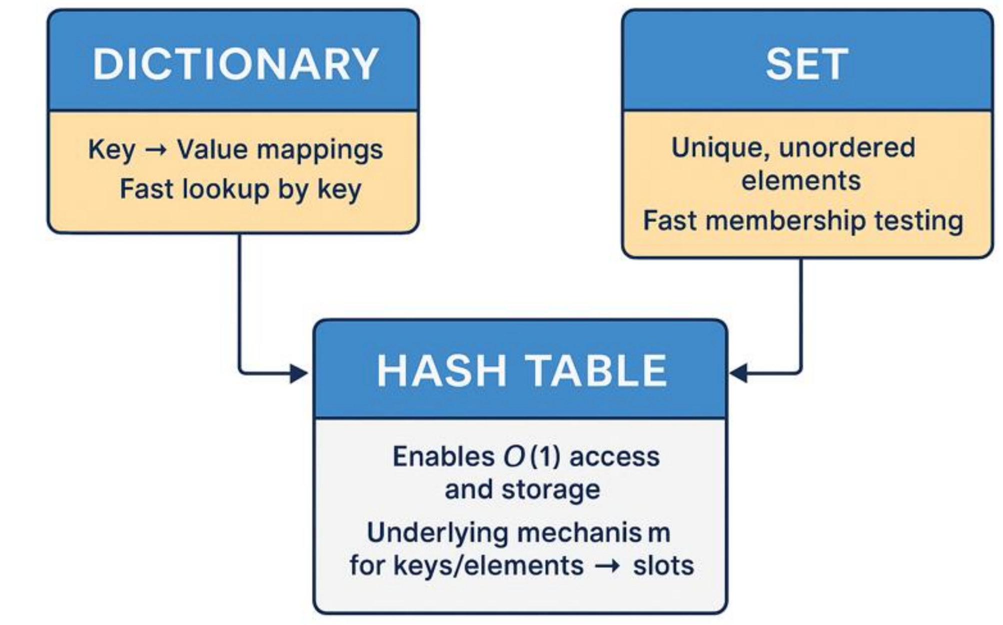

DICTIONARY
Key → Value mappings
Fast lookup by key
SET
Unique, unordered elements
Fast membership testing
HASH TABLE
Enables O(1) access and storage
Underlying mechanis m for keys/elements → slots

Table 4-0 Relationships between Dictionary, Hash Table and Set.

4.1 Efficient data lookups and storage security relevance of hashing

Dictionaries (as implemented by hash tables) and password management are related through the underlying use of hash tables for efficient lookup and storage, but the connection goes deeper when you consider how password data is stored and protected in real-world systems.

In user authentication systems, user data (username, password hash, etc.) is typically stored in a database table, which can conceptually be thought of as a key-value store—a username (the key) is associated with a password hash (the value).

Modern databases often use hash table-like indexing internally to quickly find user records by username, but the actual password storage is handled by dedicated fields in a user table

4.2 Definition

A dictionary is a data structure that stores pairs of keys and values. Each key is unique and acts as an identifier for its associated value. Dictionaries provide fast lookup, meaning you can quickly retrieve the value associated with a given key without scanning the entire collection. Internally, a dictionary typically uses a hash table , where the key is processed by a hash function to determine where its value is stored.

Dictionaries allow dynamic insertion, deletion, and updating of key-value pairs, and the order of keys is usually maintained in modern implementations (such as Python 3.7+).

Valid keys must be hashable — meaning they have a stable, immutable hash value (e.g., strings, numbers, tuples of immutable elements).

In summary, a Python dictionary is a built-in data structure that stores key-value pairs. It's similar to a real-world dictionary where you look up a word (key) to get its meaning (value).

Below is an example of dictionary, named user_profile, stores information about a user account — like what you'd find in a web app or login system.

```python
"""
example of python dictionary
"""
user_profile = {
    "username": "alice123",
    "password_hash": "a1b2c3d4...",
    "role": "admin",
    "login_attempts": 0
}
```

Key-Value Breakdown of the dictionary user_profile:

Key	Value	Type	Explanation
"username"	"alice123"	str (string)	The user's login name or ID. It identifies the user.
"password_hash"	"a1b2c3d4..."	str (string)	This is a hashed version of the user's password. It's **not** the raw password.
"role"	"admin"	str (string)	Defines the user's access level (e.g., "admin", "user", "guest").
"login_attempts"	0	int (integer)	Tracks how many times the user has tried to log in (e.g., for lockout rules).

4.2.1 Use Cases Examples

4.2.1a Check login attempts

```
# Adding a new key-value pair
user_profile["last_login"] = "2023-10-01 12:00:00"

# checking login attempts, by accessing the value of the key login_attempts
if user_profile["login_attempts"] >= 3:
    print("Account locked due to too many failed attempts.")

# Updating a value in the dictionary
# Incrementing login attempts, updating the value of login_attempts
user_profile["login_attempts"] += 1

# Changing the role of the user
user_profile["role"] = "user"
```

4.3 Why Use password_hash?

Passwords should never be stored as plain text for security reasons. Instead, we store a hash, which is a one-way encrypted version of the password. This helps protect user data even if the system is compromised.

Values, on the other hand, can be of any type and do not need to be unique.

Dictionary Key Features:

- Key-Value Pairs: Every entry consists of a unique key and an associated value.
- Uniqueness: Keys must be unique; duplicate keys overwrite existing values.

4.4 Cybersecurity Dictionary Examples

4.4.1 Credential Storage (username → password hash)

```python
import hashlib

credentials = {
    'alice': hashlib.sha256('mypassword'.encode()).hexdigest(),
    'bob': hashlib.sha256('securepass'.encode()).hexdigest()
}

# Check if a user exists and verify password
user = 'alice'
input_password = 'mypassword'
hashed_input = hashlib.sha256(input_password.encode()).hexdigest()

if user in credentials and credentials[user] == hashed_input:
    print("Access granted")
else:
    print("Access denied")
```

4.4.2. Blacklist Management (IP → block reason)

```python
blacklist = {
    '192.168.1.10': 'suspicious activity',
    '203.0.113.5': 'known malware host'
}

ip = '203.0.113.5'
if ip in blacklist:
    print(f"Blocked {ip}: {blacklist[ip]}")
else:
    print(f"{ip} is allowed")
```

4.4.3. Threat Intelligence Feed (malware hash → threat actor)

```python
threat_feed = {
    '5d41402abc4b2a76b9719d911017c592': 'APT29',
    '7d793037a0760186574b0282f2f435e7': 'Lazarus Group'
}

malware_hash = '7d793037a0760186574b0282f2f435e7'
if malware_hash in threat_feed:
    print(f"Threat identified: {threat_feed[malware_hash]}")
else:
    print("Unknown threat")
```

4.4.4. Credentials Dictionary

```
import hashlib

credentials = {
    'alice': hashlib.sha256('mypassword'.encode()).hexdigest(),
    'bob': hashlib.sha256('securepass'.encode()).hexdigest()
}

# Check if a user exists and verify password
user = 'alice'
input_password = 'mypassword'
hashed_input = hashlib.sha256(input_password.encode()).hexdigest()

if user in credentials and credentials[user] == hashed_input:
    print("Access granted")
else:
    print("Access denied")
```

4.4 Hash Tables

A hash table is a low-level data structure that underpins both dictionaries and sets.

Under the hood, Python sets are implemented using hash tables (specifically, a variant of a hash table). This is what allows for their efficient performance, especially for membership testing (in operator) and adding/removing elements.

Here's the connection:

1. **Hashing:** When you add an element to a set, Python calculates a hash value for that element. This hash value is an integer that serves as a unique identifier for the element (ideally).

2. **Storage:** This hash value is then used to determine where the element is stored within the underlying hash table.
3. **Uniqueness:** When you try to add an element that's already in the set, Python recalculates its hash. If the hash is the same and the elements are equal, Python recognizes it as a duplicate and doesn't add it again.
4. **Efficient Lookups:** When you check if an element is in a set (in operator), Python calculates the hash of the element you're looking for and directly goes to the corresponding location in the hash table. This makes membership testing very fast on average, close to constant time O(1).

4.4.1 Illustrative Analogy of Hash Table

Think of a hash table like a post office with many mailboxes.

- The **element** you want to store in the set is like a piece of mail.
- The **hash function** is like a special address-generating machine that takes the mail and produces a unique mailbox number (the hash value).
- The **hash table** is the collection of mailboxes.

When you want to check if a piece of mail (an element) is in the post office (the set), you use the address-generating machine to find its mailbox number and check if there's mail in that box. This is much faster than searching through every single mailbox.

In other words, given a Key, it is passed through a **hash function** that converts it into a number (called a **hash**). That number determines the **index** (position) in an internal array where the **value** is stored. When you access the key later, the same hash function finds the value very quickly.

Key → Hash Function → Index → Value

So, when storing something stores securely `"password"` it does something like:

```python
import hashlib
# using hashlib to hash a password

password = "securePassword123"
hashed = hashlib.sha256(password.encode()).hexdigest()
print("SHA-256 hash:", hashed)
```

Where the hashed is a digest string value of hexadecimal.

```
SHA-256 hash:
4dbd5e49147b5102ee2731ac03dd0db7decc3b8715c3df3c1f3ddc62dcbcf86d
```

The hashlib.sha256 is a function from Python's built-in hashlib module used to securely hash data using the SHA-256 algorithm (a member of the SHA-2 family).

4.4.2 SHA-256 Key Facts:

- SHA stands for Secure Hash Algorithm.
- 256 means the output is 256 bits (32 bytes).
- It's one way: you can't reverse the hash to get the original input.
- It's commonly used in password security, file integrity, and blockchain.

4.4.3 Hash Collision

Sometimes two keys might produce the same hash index. This is called a collision. To handle it, hashtables use techniques like Chaining: Store multiple items at the same index using a list. Or Open Addressing, find another empty spot in the array. Python handles this automatically for you in its dictionary.

4.5 Python Sets

A set is a collection of unique, unordered items. Sets store only unique elements, automatically eliminating duplicates. They allow fast membership testing — checking whether an element is in the set happens in constant (or near-constant) time, thanks to underlying hash table mechanisms. Sets do not support indexing or positional access because they are unordered — you cannot rely on the order in which elements appear.

Common operations on sets include union, intersection, difference, and symmetric difference, making them very useful for mathematical and logical comparisons between collections. Like dictionary keys, the elements in a set must be hashable.

Conceptually, a set is like a mathematical set: a bag of distinct items, where presence matters but order does not.

Example:

```
# Creating sets
set1 = {1, 2, 3, 4, 5}
```

```
set2 = set([3, 5, 6, 7, 7])  # Note the duplicate '7' will be
automatically removed
print(f"set1: {set1}")
print(f"set2: {set2}")

# Adding and removing elements
set1.add(6)
print(f"set1 after adding 6: {set1}")
set2.remove(3)
print(f"set2 after removing 3: {set2}")
# set2.remove(10)  # This would raise a KeyError because 10 is not
in the set
set2.discard(10) # This does nothing if the element is not present

# Set operations
set_union = set1.union(set2)
print(f"Union of set1 and set2: {set_union}")

set_intersection = set1.intersection(set2)
print(f"Intersection of set1 and set2: {set_intersection}")

set_difference = set1.difference(set2)
print(f"Difference of set1 and set2 (elements in set1 but not in
set2): {set_difference}")

# Checking for membership
print(f"Is 3 in set1? {3 in set1}")
print(f"Is 8 in set2? {8 in set2}")
```

Summary

Efficient Data Lookups and Storage	Dictionaries (hash maps) and sets provide constant-time lookups on average. Underlying hash functions, collision handling (chaining, open addressing), and load factor.- Analyze space-time trade-offs.
Use Cases: Credential Management, Blacklists, and Whitelists	Dictionaries store user credentials (e.g., username-password pairs).- Use sets for blacklists (blocked entities) and whitelists (trusted entities).- Discuss real-world scenarios in cybersecurity and access control.
Practical Applications	

Summary

AI Anomaly Detection Using Sets	Sets to track unique events or patterns.- Feed detected anomalies into an AI summarizer.- Example: detecting abnormal user logins or device connections.
Fast AI-Driven Phishing URL Detection	A hash table of known phishing URLs.- Integrate an AI classifier to flag suspicious patterns.- Combine set membership checks with AI-predicted threat scores for real-time decisions.
AI-Assisted Password Breach Checker	Hash tables to store breached password hashes.- AI models help flag weak or reused passwords.- Combine fast hash lookups with machine-learned risk assessment.
Event Frequency Analysis with AI Summarization	- Count event occurrences using dictionaries (e.g., login attempts, API calls).- Feed aggregated data into an AI summarizer to highlight trends or abnormalities. System monitoring or log analysis.

Module 4: Learning Materials & References:

i. Python Dictionaries
ii. Python Sets
iii. Hash Tables explained with PYTHON Video

Assessment: Module 4: Dictionaries, Sets, and Hash Tables in password management

Part A: Quiz

4.1a What is the primary characteristic of a Python set?

a) It stores elements in a specific order.

b) It allows duplicate elements.

c) It stores only unique elements.

d) It can only contain elements of the same data type.

4.2a Which of the following methods would you use to add a single element to a set in Python?

a) append()

b) insert()

c) add()

d) update()

4.3a What will be the output of the following Python code?

```
my_set = {1, 2, 2, 3, 3, 3}

print(len(my_set))
```

a) 6
b) 3
c) 1
d) An error

4.4a Python sets are primarily implemented using which data structure?

a) Linked lists
b) Arrays
c) Hash tables
d) Trees

4.5a What is the role of a hash function in the context of hash tables?

a) To sort the elements in the table.
b) To assign a unique index or "key" to each element for storage.
c) To compress the data stored in the table.
d) To encrypt the elements in the table.

4.6a Why does using a hash table make membership testing in sets efficient on average?

a) It iterates through all the elements to find a match.
b) It directly calculates the potential location of the element using its hash.
c) It stores elements in a sorted order, allowing for binary search.

d) It uses a tree-like structure for searching.

4.7a What happens when two different elements have the same hash value? This is known as a:
a) Hash collision
b) Hash fusion
c) Hash explosion
d) Hash diffusion

4.8a What is the purpose of the __hash__ method in a Python class?

```
class User:
    def __init__(self, username):
        self.username = username

    def __hash__(self):
        return hash(self.username)
```

A) To print the object as a string
B) To compare two objects for equality
C) To allow the object to be used as a key in dictionaries and sets
D) To convert the object into a file

4.9a What does the following code do?

```
import hashlib

text = "secure123"
hash_result = hashlib.sha256(text.encode()).hexdigest()
print(hash_result)
```

A) Encrypts the text for secure communication
B) Produces a fixed-length, irreversible hash of the string
C) Compresses the text to reduce its size
D) Converts the text to uppercase

Part B: Short Answers

4.1b How can Python sets be used in cybersecurity for identifying unique malicious IP addresses from a large log file?

4.2b In AI-driven threat intelligence, how could sets be used to compare lists of indicators of compromise (IOCs) from different sources?

4.3b Why is the underlying hash table implementation of Python sets and dictionaries beneficial for real-time cybersecurity analysis?
4.4b What is the primary role of a hash function in the context of hash tables?
4.5b What is the average time complexity for adding or removing an element from a Python set? Why

Part C: Python Exercises

4.1c Anomaly Detection using Feature Sets

Scenario:

You are developing a simple anomaly detection system for network traffic. Each network connection can be characterized by a set of features (e.g., source IP, destination port, protocol). Normal network traffic exhibits certain common sets of these features. Anomalous traffic (potential attacks) might exhibit unusual or rare combinations of features.

Tasks:

1. **Represent normal traffic patterns:** Create a Python set where each element is a tuple representing a unique combination of network traffic features observed in normal behavior. For example: ("192.168.1.10", 80, "TCP"). Populate this set with some sample normal traffic patterns.
2. **Represent incoming network connection features:** Create a Python list where each element is a tuple representing the features of a newly observed network connection. This list might contain both normal and anomalous connections.
3. **Detect anomalies:** Iterate through the list of incoming connection features. For each connection, check if its feature set (tuple) exists in the set of normal traffic patterns. If it does *not* exist, consider it a potential anomaly.
4. **Report anomalies:** Print out the feature sets of the network connections that are identified as potential anomalies.

4.2.c Building a Unique Vocabulary using Sets

Scenario:

You are working on a simple Natural Language Processing (NLP) task. You have a collection of text documents (a corpus). Your goal is to build a unique vocabulary of all the words present in this corpus. This vocabulary will be used for further text analysis.

Tasks:

1. **Represent the text corpus:** Create a Python list of strings, where each string represents a document in your corpus.
2. **Preprocess the text (basic):** For each document, convert all words to lowercase and remove basic punctuation (like periods, commas, and question marks) attached to the words. You can split each document into a list of words.
3. **Build the vocabulary:** Use a Python set to store all the unique words encountered across all the documents in the corpus. Iterate through each document, process the words, and add them to the set.
4. **Report the vocabulary:** Print the final set of unique words (the vocabulary). Also, print the total number of unique words in the vocabulary.

4.3.c Write a python function that adds a new user to a dictionary only if the username does not already exist.

```python
def add_user(users, username, password):
    # TODO: Add the user only if the username is not already in the dictionary
    pass


# Test Code
users = {}
add_user(users, "alice", "pass123")
add_user(users, "bob", "qwerty")
add_user(users, "alice", "newpass")  # should not overwrite

print(users)  # Expected: {'alice': 'pass123', 'bob': 'qwerty'}

#sampel output
# {'alice': 'pass123', 'bob': 'qwerty'}
```

4.4c Write a function called simple_hash that takes a string and returns a fake "hash" as a hexadecimal string.

Use the following algorithm:

Start with an initial value of 0.

For each character in the string:

- Convert it to its Unicode code (ord(char)),
- Multiply it by a prime number (e.g., 31),
- Add it to the hash value.

Return the final hash value as a hex string using hex().

```python
def simple_hash(text):
    # TODO: Implement the hash simulation
    pass
# Test Code
print(simple_hash("abc"))     # Sample output: '0x1f14'
print(simple_hash("password")) # Sample output: '0x7e2c'
```

4.5.c You have a class User with email and role. Write a python script that makes the objects usable as dictionary keys, so users with the same email are considered the same (i.e., only one entry per email).

```python
class User:
    def __init__(self, email, role):
        self.email = email
        self.role = role

    # TODO: Implement __hash__ and __eq__ so that users with the same email are
equal
    pass


# Test Code
u1 = User("admin@example.com", "admin")
u2 = User("admin@example.com", "superadmin")

user_roles = {u1: u1.role}
user_roles[u2] = u2.role  # should overwrite u1

print(len(user_roles))        # Expected: 1
print(user_roles[u1])         # Expected: 'superadmin'
print(user_roles[u2])         # Expected: 'superadmin'
```

Module 5: Stacks and Queues in Packet Inspection

Stacks and queues are linear data structures used to store and organize data efficiently. Think of them as containers where items are added and removed in a particular order.

5.1 Stack: Last In, First Out (LIFO)

Imagine a stack of plates: you always place the newest plate on the very top, and when you need a plate, you always take the one from the very top first. This perfectly illustrates how a stack data structure works.

The fundamental principle governing a stack is LIFO, which stands for "Last-In, First-Out." This means the last item added to the stack is always the first one to be removed.

Here are the key operations that can be performed on a stack:

- Push: This operation adds a new element to the top of the stack. Think of it as placing another plate on top of the existing stack.
- Pop: This operation removes and returns the element currently at the top of the stack. This is like taking the topmost plate away.
- Peek: This operation allows you to view the element at the top of the stack without removing it. It's like peeking at the top plate without actually taking it off.
- isEmpty: This operation checks whether the stack contains any elements. It tells you if the stack of plates is empty or not.

Because of their Last-In, First-Out (LIFO) nature, stacks are particularly effective for tasks that involve reversing order or managing backtracking processes, like an "undo" feature.

Example of Stack in Python:

```python
'''Basic Stack Operations in Python
   this script demonstrates basic stack operations using a list in Python.
   Stack operations include push, pop, peek, and checking if the stack is empty.
'''

stack:list = []     # stack starts out as an empty list
print(type(stack))  # Stack operations using a list in Python

# Push
stack.append('A')   # Push 'A' onto the stack
stack.append('B')   # Push 'B' onto the stack

# Peek not removing the top element
```

```python
print("Top of stack:", stack[-1])  # Output: B

# Pop removing the top element
print("Popped:", stack.pop())     # Output: B

# Check empty
print("Is stack empty?", len(stack) == 0)
```

5.2 Queue: First-in, First Out (FIFO)

A Queue is a linear data structure that operates on the principle of FIFO (First-In, First-Out). This means the first element added to the queue will be the first one to be removed.

Imagine a line at a checkout counter: the first person to join the line is always the first person to be served. This perfectly illustrates the FIFO principle of a queue.

Here are the key operations that can be performed on a queue:

- Enqueue: This operation adds a new element to the rear (or end) of the queue. (Think of someone joining the back of the checkout line.)
- Dequeue: This operation removes and returns the element from the front of the queue. (This is the person at the front of the line being served and leaving.)
- Peek: This operation allows you to view the element at the front of the queue without removing it. (Like seeing who is next in line without them leaving.)
- isEmpty: This operation checks whether the queue contains any elements. (Is there anyone in the line?)

Deques: (pronounced "deck") combining the best of both worlds. You can efficiently add and remove elements from *both* ends. This means a single `deque` can behave as:

- **A Stack:** By only using "add to front" (push) and "remove from front" (pop).
- **A Queue:** By using "add to rear" (enqueue) and "remove from front" (dequeue).

To ensure a strict order of processing, queues are often employed in applications like task scheduling, message queuing, and managing print jobs.

Example of Queue in Python:

```python
from collections import deque
'''Basic Queue Operations in Python
   this script demonstrates basic queue operations using deque from collections
module.
```

```
   Queue operations include enqueue, dequeue, peek, and checking if the queue is
empty.
'''
# Queue operations using deque from collections module
queue:deque = deque() # start with an empty deque
print(type(queue))  # Queue operations using deque in Python
# Enqueue
queue.append('A')   # Enqueue 'A' into the queue to the rear
queue.append('B')   # Enqueue 'B' into the queue to the rear

# Peek
print("Front of queue:", queue[0])  # Output: A

# Dequeue
print("Dequeued:", queue.popleft())  # Output: A removes the front element

# Check empty
print("Is queue empty?", len(queue) == 0)
```

Summary of key concepts and operations for Stack and Queue:

Feature	Stack	Queue
Order	LIFO	FIFO
Add Element	append()	append()
Remove Element	pop()	popleft() (from deque)
Typical Use	Undo, recursion	Task scheduling

5.3 Stacks and Queues in Cybersecurity

Stacks and queues are not merely theoretical concepts; they are fundamental data structures with critical practical applications in real-world cybersecurity systems. They are indispensable tools for efficiently managing data flow, performing log analysis, and coordinating security-related tasks.

5.3.1 Stacks in Cybersecurity: Last-In, First-Out (LIFO)

Stacks are utilized when operations need to be reversed or traced back in the exact opposite order they occurred. Their LIFO (Last-In, First-Out) nature makes them ideal for scenarios where you need to "undo" actions or follow a sequence of events backward.

5.3.1a Log File Analysis and Incident Response:

When investigating security incidents, stacks are crucial for scanning through system or intrusion logs to trace actions backward. This helps in following a chain of events that might lead to a compromise or attack. Analyzing a stack trace after a system crash or an exploit to understand the sequence of function calls. Similarly, backtracking a user's command history to identify malicious activity.

5.3.1b Configuration and Syntax Validation:

Stacks are highly effective for validating the structure and syntax of configuration files (e.g., JSON, XML, YAML) or source code. They are used to ensure balanced delimiters (like parentheses, brackets, and braces) and proper nesting.

Detecting Mismatched Brackets: Identifying improperly nested rules or unclosed tags in critical security configurations that could lead to vulnerabilities.

Analyzing Scripts for Integrity: Useful for detecting tampered scripts, injected code, or malware obfuscation techniques that might disrupt expected syntax.

Example Python Code for Bracket Validation:

```python
'''Bracket Validation in Configuration Syntax
   this script checks if a given configuration string has valid syntax using a
stack.
   It ensures that all opening brackets have corresponding closing brackets in the
correct order.
'''
def is_valid_config(text):
    stack:list = []             # Initialize an empty stack of type list
    # Dictionary to map closing brackets to their corresponding opening brackets
    pairs:dict = {')': '(', ']': '[', '}': '{'}

    # loop through each character in the text
    for char in text:
        if char in '([{':   # If it's an opening bracket
            # Push it onto the stack
            stack.append(char)
        elif char in ')]}': # If it's a closing bracket
```

```
            # Check if the stack is empty or the top of the stack doesn't match the
corresponding opening bracket
            if not stack or stack.pop() != pairs[char]:
                return False # Invalid configuration syntax
    # If the stack is empty after processing all characters, the configuration
syntax is valid
    return not stack

config = "{ 'setting': [true, false] }"
print("Valid config syntax?" , is_valid_config(config))  # True
```

5.3.1c Task Scheduling

Use cases:

a) Task scheduling for Antivirus engines or automated security tools process tasks in order: **scan → detect → report**, ie, Scheduled updates, vulnerability scans, or intrusion detection checks.
b) Message Queues for Security systems use asynchronous messaging (like RabbitMQ or Kafka) to queue logs, alerts, or audit messages. This would ensure that no log or alert is missed during high-traffic incidents.
c) Network Packet Inspection, system queuing packets for inspection before forwarding to avoid buffer overflows or DoS attacks. Ensures timely but ordered processing of network traffic.

Example Python Code:

```
from collections import deque
'''example of simple task queue using deque
   this script demonstrates a simple task queue using deque from the collections
module.
   It allows adding tasks to the queue and processing them in the order they were
added.
'''
security_tasks = deque()

# We need to add tasks to the queue (Enqueue) in the following order
security_tasks.append("Scan system")    # Enqueue a task to scan the system
security_tasks.append("Update antivirus") # Enqueue a task to update antivirus
security_tasks.append("Update firewall rules")  # Enqueue a task to update firewall
rules
```

```
security_tasks.append("Analyze intrusion logs") # Enqueue a task to analyze
intrusion logs

# Process tasks (Dequeue)
while security_tasks:
    task = security_tasks.popleft() # Dequeue the next task
    # Simulate processing the task
    print("Running task:", task)
```

5.4 Scapy Python-based network packet manipulation tools.

Scapy is a powerful Python-based network packet manipulation tool. It allows you to create, send, sniff, dissect, and forge network packets. It's widely used in cybersecurity, penetration testing, and network research because it provides fine-grained control over packet-level operations. Basic information can be found here, a video tutorial is listed in Module 5 learning materials.

Module 5: Learning Materials & References:

i. Python Stacks, Queues, and Priority Queues in Practice
ii. Deque & Stack
iii. collections — Container datatypes
iv. PyShark Network Analysis
v. Scapy Download and installation
vi. Scapy and Python

Assessment: Module 5: Stacks and Queues in Packet Inspection

Part A: Quiz

5.1a What is the order of elements in a stack?

A. First In, First Out (FIFO)
B. Last In, First Out (LIFO)
C. Random Order
D. Shortest Job First

5.2a Which Python method is used to remove the top item from a stack (implemented as a list)?

A. remove()
B. delete()
C. pop()
D. dequeue()

5.3a Which data structure is best suited for task scheduling?

A. Stack
B. Set
C. Queue
D. Dictionary

5.4a What will be the result after the following code?

```
from collections import deque
q = deque()
q.append('A')
q.append('B')
q.popleft()
print(q)
```

5.5a Which operation is NOT typically associated with a queue?

A. enqueue
B. dequeue
C. peek
D. reverse

Part C: Python Exercises

5.1c Emulate Stack

Write a Python script emulate_stack.py to emulate the Stack function in Python by using a class called Stack. The script must implement the following functions: push, pop, size, isempty, peep, and __str__.

Sample output:

```
# Example usage of the Stack class
my_stack = Stack()
my_stack.push(1)
my_stack.push(2)
my_stack.push(3)
print("Stack after pushing 1, 2, 3:", my_stack)  # Output: Stack([1, 2, 3])
print("Top element (peek):", my_stack.peek())  # Output: 3
print("Popped element:", my_stack.pop())  # Output: 3
print("Stack after popping:", my_stack)  # Output: Stack([1, 2])
print("Is stack empty?", my_stack.is_empty())  # Output: False
print("Stack size:", my_stack.size())  # Output: 2
```

5.2c Emulate Queue

Write a Python script emulate_queue.py to emulate the Queue function in Python by using a class called Queue. The script must implement, enqueue (add), dequeue (remove) size, is_empty, peek, __str__.

Sample output:

```
# Example usage of the Queue class
my_queue = Queue()
my_queue.enqueue('Task 1')
my_queue.enqueue('Task 2')
my_queue.enqueue('Task 3')
print("Queue after enqueueing ", my_queue)  # Output: Queue(['Task 1', 'Task 2',
'Task 3'])


print("Front element (peek):", my_queue.peek())  # Output: Task 1
print("Dequeued element:", my_queue.dequeue())  # Output: Task 1

print("Queue after dequeueing:", my_queue)  # Output: Queue(['Task 2', 'Task 3'])
print("Is queue empty?", my_queue.is_empty())  # Output: False
print("Queue size:", my_queue.size())  # Output: 2

# add another element to the queue
my_queue.enqueue('Task 4')
print("Queue after enqueueing Task 4:", my_queue)  # Output: Queue(['Task 2', 'Task
3', 'Task 4'])
print("Front element (peek):", my_queue.peek())  # Output: Task 2
```

5.3c Firewall Packet Queue

Write a Python script process_packets.py to stores incoming packages in a queue and process them in FIFO order. Each packet is a dictionary like:

```
# Example input
packets = [
    {"src": "10.0.0.2", "port": 80},
    {"src": "10.0.0.5", "port": 443},
    {"src": "192.168.1.1", "port": 22},
]

process_packets(packets)

# expected output:
# Processing packet from 10.0.0.2 on port 80
# Processing packet from 10.0.0.5 on port 443
# Processing packet from 192.168.1.1 on port 22
```

5.4c Capture network data using Scapy

From the module Learning Materials & References, follow the installation instructions to install scapy and dependencies, then watch the tutorial for Scapy and Python. Run Scapy from a command line and capture 20 packets from your computer, save the packets in a file called data20.pcap. Write a simple python script scapy_packets.py to output the packets, and submit data20.pcap file for credit.

Sample output:

```
###[ Ethernet ]###
 dst      = f8:63:3f:48:0d:f5
 src      = d8:47:32:85:37:04
 type     = IPv4
###[ IP ]###
   version  = 4
   ihl      = 5
   tos      = 0x0
   len      = 1452
   id       = 27026
   flags    =
   frag     = 0
   ttl      = 119
   proto     = tcp
   chksum    = 0xd33c
   src       = 142.250.176.4
   dst       = 192.168.1.214
   \options  \
###[ TCP ]###
      sport     = https
      dport     = 61669
      seq       = 3217401247
      ack       = 2302258872
      dataofs   = 5
      reserved  = 0
      flags     = A
      window    = 1126
      chksum    = 0xc338
      urgptr    = 0
      options   = []
```

5.5.c Display packets in Python with Scapy

Write a python script packets_data.py to display summary from data20.pcap. Access individual field's data and output the source ip address, source map ip address, destination ip address, ethernet or mac addresses of source and destination, for 20 tcp packages.

Sample output:

```
Source MAC: f8:63:3f:48:0d:f5
Destination MAC: d8:47:32:85:37:04
Source IP: 192.168.1.214
Destination IP: 142.250.176.4
TCP Source Port: 61669 Destination Port: 443
Packet 2:
Source MAC: d8:47:32:85:37:04
Destination MAC: f8:63:3f:48:0d:f5
Source IP: 142.250.176.4
Destination IP: 192.168.1.214
TCP Source Port: 443 Destination Port: 61669
Packet 3:
Source MAC: d8:47:32:85:37:04
```

Module 6: Graphs & Network Analysis

Graphs are fundamental structures in mathematics and computer science, consisting of vertices (also called nodes) and edges (connections between vertices). The study of graphs and their properties is known as graph theory, which has wide applications in areas such as network analysis, pathfinding, and data optimization

Representation of Graphs & Networks

Since Networks, or graphs, are collections of "nodes" (or vertices) and "edges" (or links) that connect pairs of these nodes. How we store this information computationally is what we mean by "representation." The choice of representation often depends on the type of operations you'll perform most frequently (e.g., checking if two nodes are connected, finding all neighbors of a node).

6.1 Adjacency List

This is one of the most common and preferred representation, especially for sparse graphs (graphs with relatively few edges compared to the maximum possible).

- **Concept:** For each node in the graph, we maintain a list (or set) of its direct neighbors.
- **Structure:** Typically implemented using a dictionary in Python, where:
  - **Keys:** Are the nodes themselves.
  - **Values:** Are lists or sets of nodes that are adjacent to the key node.
- **Example (Undirected Graph):** Nodes: A, B, C, D Edges: (A,B), (A,C), (B,D), (C,D)
  A: [B, C]
  B: [A, D]
  C: [A, D]
  D: [B, C]
  Example (Directed Graph): Nodes: A, B, C, D Edges: A->B, A->C, B->D, C->D
  A: [B, C]
  B: [D]
  C: [D]
  D: []
- **Pros:**
  - **Space Efficient for Sparse Graphs:** Only stores existing edges. For V vertices and E edges, it takes O(V+E) space.
  - **Efficient for Neighbor Lookup:** Quickly find all neighbors of a specific node.
- **Cons:**
  - **Checking Edge Existence:** To check if an edge (U, V) exists, you might have to iterate through the neighbors list of U, which can be O(V) in the worst case (or O(textdegreeofU)). If the value is a set, it's O(1) on average.

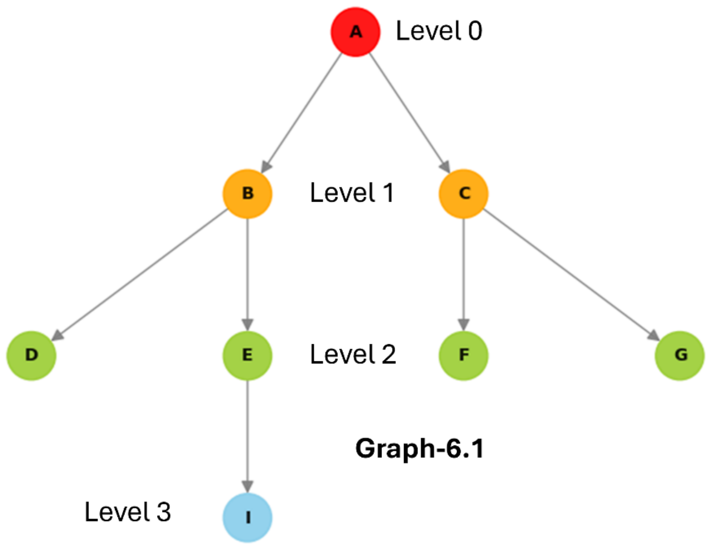
Ways to Traverse a Graph

Traversal refers to the process of visiting all the vertices in a graph, typically starting from a chosen source vertex. The goal is to systematically explore the structure and relationships encoded in the graph. There are several main techniques for graph traversal, each with its own strategies and use cases

6.2 Breadth-First Search (BFS):

BFS explores all vertices at the present depth (or "level") before moving on to vertices at the next depth level. It uses a **queue** to keep track of vertices to visit next. BFS is ideal for finding the shortest path in unweighted graphs and for exploring all reachable vertices from a source in order of distance from the source.

Below is an example of Adjacency list for Graph-6.1, a directed graph



Graph-6.1

A:[B,C] A→B, A→C
B:[D,E] B→D, B→E
C:[F,G] C→F, C→G
D:[] D isolated node
E:[I,] E→I
F:[] F isolated node
G:[] G isolated node
I:[] I isolated node

Using directed Graph-6.1 as an example, If starting from node A, BFS might visit A, then B, C, then D, E, F, G, and final I, level by level.

Key Concepts:

1. **Level by Level Exploration :** The core idea is to visit all nodes at the current "depth" (distance from the start node) before moving on to nodes at the next depth level, Horizontal before Vertical!
    - **Level 0:** The start node itself.
    - **Level 1:** All immediate neighbors of the start node.
    - **Level 2:** All unvisited neighbors of Level 1 nodes.
    - ...and so on.
2. **Queue Data Structure:** BFS uses a **queue** (First-In, First-Out or FIFO) to manage the order of node visits.
    - When you visit a node, you add its unvisited neighbors to the back of the queue.
    - You then take the next node to visit from the front of the queue.
3. **Visited Set:** To prevent infinite loops (especially in graphs with cycles) and to ensure each node is processed only once, BFS maintains a visited set (or array/list). Once a node is added to the queue, it's marked as visited.

When to Use BFS? (Applications)

BFS is a fundamental algorithm with many practical applications:

1. **Finding the Shortest Path in an Unweighted Graph:** BFS naturally finds the shortest path (in terms of the number of edges) between a starting node and all other reachable nodes in an *unweighted* graph. This is because it explores layer by layer, guaranteeing that the first time you encounter a node, you've found the shortest path to it.
2. **Web Crawlers/Indexing:** Search engines use a form of BFS to crawl the web, starting from a set of seed pages and exploring all linked pages.
3. **Social Networks:**
    - Finding the "degrees of separation" between two people.
    - Identifying all friends up to a certain level of connection.

6.3 Depth-First Search (DFS):

DFS is an algorithm for traversing or searching a graph data structure. Unlike BFS which explores level by level, DFS explores as far as possible along the vertical direction of each branch before backtracking. Imagine exploring a maze: you pick a path, follow it as far as you can, and if you hit a dead end or a visited spot, you go back to the last junction and try another unvisited path.

Using Graph-6.1 as an example, If starting from node A, DFS might visit A, then B, D. Since D is at the end of the branch we then switch E,I. Then back to C, F, and final G. The result is: A, B, D, E, I, C, F, G

Key Concepts:

1. **"Go Deep" First:** The primary characteristic of DFS is that it prioritizes exploring deeper into a branch of the graph before it "backtracks" and explores other branches at the same or higher levels.
2. **Stack Data Structure (or Recursion):**
    - **Iterative DFS:** Uses an explicit **stack** (Last-In, First-Out or LIFO) to keep track of vertices to visit. When you visit a node, you push its unvisited neighbors onto the stack. You then take the *last* added node from the stack to visit next.
    - **Recursive DFS:** More commonly, DFS is implemented recursively. The function calls themselves, and the program's implicit **call stack** serves the same purpose as an explicit stack in the iterative version. Each recursive call represents going "deeper" into the graph.
3. **Visited Set:** Just like BFS, DFS maintains a visited set (or array/list) to prevent infinite loops in graphs with cycles and to ensure each node is processed only once. Once a node is processed, it's marked as visited.

Topological Sort:

- **How it works:** This technique is used specifically for directed acyclic graphs (DAGs). It produces a linear ordering of vertices such that for every directed edge (u, v), vertex u comes before v in the ordering.
- **When to use:** Commonly used for scheduling tasks with dependencies, ensuring that all prerequisites are met before a task is started

Dijkstra's Algorithm:

- **How it works:** Dijkstra's algorithm is a variant of BFS for weighted graphs. It finds the shortest path from a source vertex to all other vertices by always expanding the least-cost path first, using a priority queue.
- **When to use:** Best for finding the shortest path in graphs with non-negative edge weights

6.4 Implementation of BFS

Here's the pseudo code for Breadth-First Search (BFS):

```
FUNCTION BFS(graph, start_node):
  CREATE a QUEUE
  CREATE a SET called VISITED

  ADD start_node to VISITED
  ENQUEUE start_node onto QUEUE

  WHILE QUEUE is NOT EMPTY:
    current_node = DEQUEUE from QUEUE

    // Process the current_node (e.g., print it, add to a traversal list)
    OUTPUT current_node

    FOR EACH neighbor OF current_node:
      IF neighbor IS NOT IN VISITED:
        ADD neighbor to VISITED
        ENQUEUE neighbor onto QUEUE
```

Explanation of the Pseudo Code:

- **FUNCTION BFS(graph, start_node):**:
  - graph: This represents the graph you want to traverse. It could be an adjacency list (most common for BFS), an adjacency matrix, etc.
  - start_node: The node from which the BFS traversal will begin.
- **CREATE a QUEUE**:
  - This initializes an empty queue data structure. A queue follows a First-In, First-Out (FIFO) principle. Elements are added to the back (enqueue) and removed from the front (dequeue).
- **CREATE a SET called VISITED**:
  - This initializes an empty set. A set is used to keep track of nodes that have already been visited during the traversal. Using a set provides very fast (average O(1)) lookup for checking if a node has been visited, which is crucial for efficiency, especially in graphs with many nodes or cycles.
- **ADD start_node to VISITED**:
  - Before processing the start_node, we immediately mark it as visited to avoid cycles and redundant processing.
- **ENQUEUE start_node onto QUEUE**:
  - The start_node is the first node to be explored, so it's added to the queue.
- **WHILE QUEUE is NOT EMPTY:**:

    - This is the main loop of the BFS algorithm. It continues as long as there are nodes in the queue waiting to be processed.
- **current_node = DEQUEUE from QUEUE**:
    - The node at the very front of the queue is removed. This is the node we are currently "visiting" or "exploring."
- **OUTPUT current_node**:
    - This step represents "processing" the node. In a real implementation, you might print the node, add it to a list representing the traversal order, check if it's a target node, or perform some other graph-specific operation.
- **FOR EACH neighbor OF current_node:**:
    - This loop iterates through all the direct neighbors (adjacent nodes) connected to the current_node.
- **IF neighbor IS NOT IN VISITED:**:
    - Before adding a neighbor to the queue, it's critical to check if it has already been visited. This prevents reprocessing nodes and avoids infinite loops in cyclic graphs.
- **ADD neighbor to VISITED**:
    - If a neighbor hasn't been visited yet, it's marked as visited *before* being enqueued. This is important to ensure that if the same neighbor is discovered again through a different path later in the current level, it won't be added to the queue redundantly.
- **ENQUEUE neighbor onto QUEUE**:
    - The unvisited neighbor is added to the back of the queue, ensuring it will be processed after all other nodes at the current level have been dequeued. This is what guarantees the "breadth-first" (level-by-level) exploration.

6.5 Implementation of DFS

DFS can be implemented either recursively (which is often more intuitive to write) or iteratively using an explicit stack. In this book, we will use recursive approach because it is the most common and often clearest way to conceptualize DFS. The function calls themselves handle "stack" management implicitly.

```
FUNCTION DFS_Recursive(graph, current_node, VISITED):
  ADD current_node to VISITED

  // Process the current_node (e.g., print it, add to a traversal list)
  OUTPUT current_node

  FOR EACH neighbor OF current_node:
    IF neighbor IS NOT IN VISITED:
        CALL DFS_Recursive(graph, neighbor, VISITED)

// To initiate the DFS for an entire graph (handling disconnected components):
FUNCTION DFS_Traversal(graph):
  CREATE a SET called VISITED
  FOR EACH node IN graph:
    IF node IS NOT IN VISITED:
      CALL DFS_Recursive(graph, node, VISITED)
```

Explanation of the Recursive Pseudo Code:

- **FUNCTION DFS_Recursive(graph, current_node, VISITED):**:
  - graph: The graph data structure.
  - current_node: The node currently being explored.
  - VISITED: A set passed around to keep track of nodes already visited across all recursive calls.
- **ADD current_node to VISITED**:
  - As soon as we enter the function for a node, we mark it as visited. This prevents reprocessing and infinite loops.
- **OUTPUT current_node**:
  - This is where you "process" the node (e.g., print it, add it to a list of traversal order, etc.). For a typical DFS, processing happens right when the node is first discovered.
- **FOR EACH neighbor OF current_node:**:
  - This loop iterates through all the direct neighbors connected to the current_node.
- **IF neighbor IS NOT IN VISITED:**:
  - If a neighbor hasn't been visited yet, it means it's a new path to explore.
- **CALL DFS_Recursive(graph, neighbor, VISITED)**:
  - This is the core of the recursion. The function calls itself on the unvisited neighbor, immediately diving deeper down that branch. The current function call will "pause" until the recursive call returns (meaning that branch has been fully explored).
- **FUNCTION DFS_Traversal(graph):**:
  - This outer function ensures that if the graph has multiple disconnected components, DFS is run on each of them to ensure all nodes are visited.

6.6 Applications: Network Mapping, Attack Path Analysis, & Social Engineering

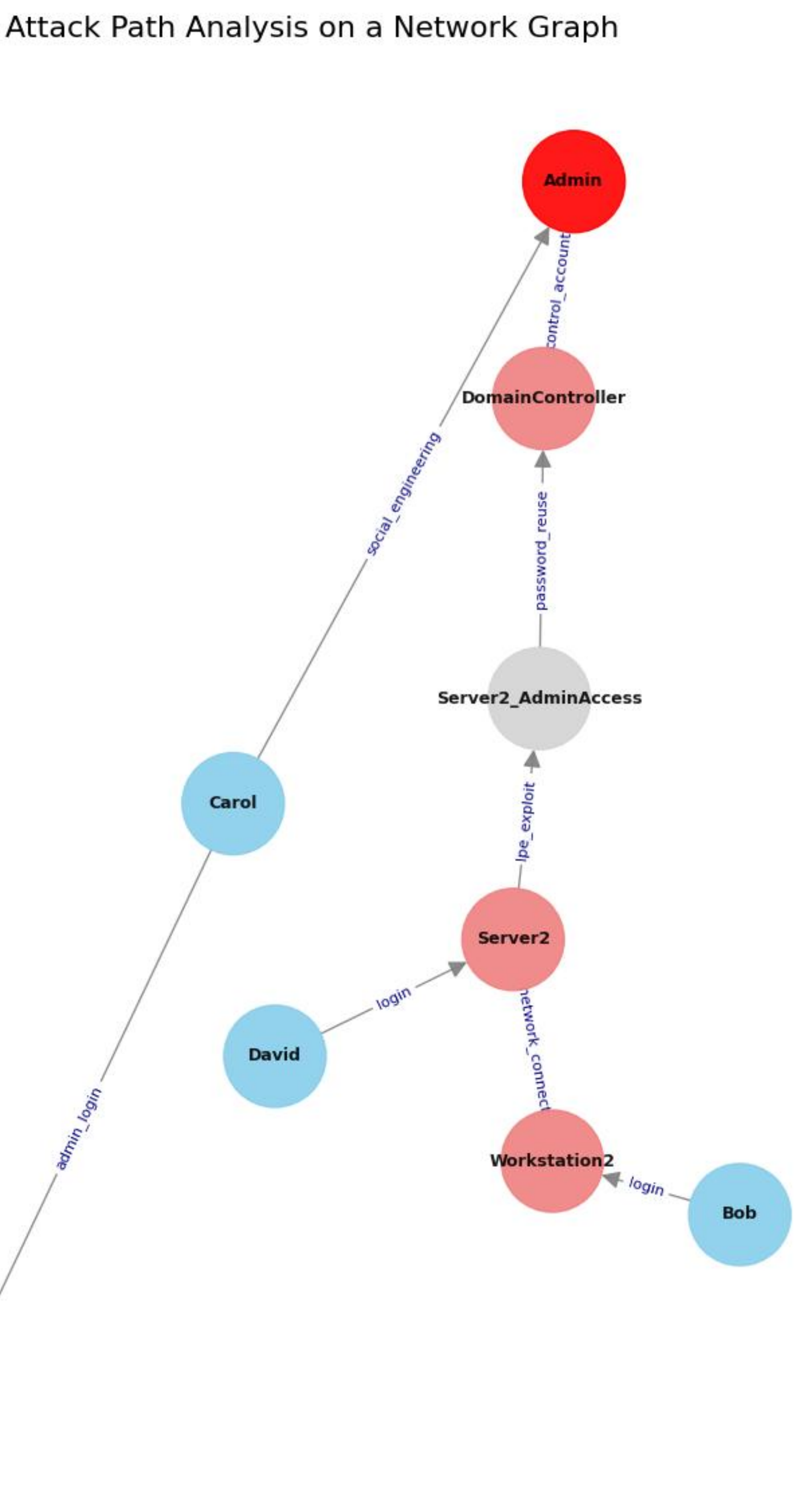
Graphs are powerful tools for mapping out network infrastructures, providing a clear visual representation of devices, connections, and access relationships within an organization. By modeling networks as graphs—where nodes represent users, devices, or accounts and edges represent access or communication paths—security teams can analyze how attackers might move laterally through the environment, such as by pivoting from one compromised host to another.

For example, graph-based analysis can reveal not only the direct connections between systems but also the underlying relationships that enable lateral movement, including privilege escalation and access to sensitive resources.

Attackers often exploit these relationships by leveraging legitimate credentials or trusted pathways, making lateral movement detection a major challenge for conventional security tools. Graph analytics, however, allows security analysts to track the sequence of access attempts, identify unusual pathways, and reconstruct an attacker's journey across the network

For instance, a user access graph might illustrate how one user's compromised credentials could be used to reach an administrator's account by traversing a series of trust relationships.

Graph6.4, utilizing Python modules **networkx** and **matplolib** to visualize user access relationships and can identify the shortest path an attacker might take to reach an administrative account. This capability is essential for both proactive defense and incident response, enabling organizations to prioritize the mitigation of high-risk pathways and strengthen critical access controls.

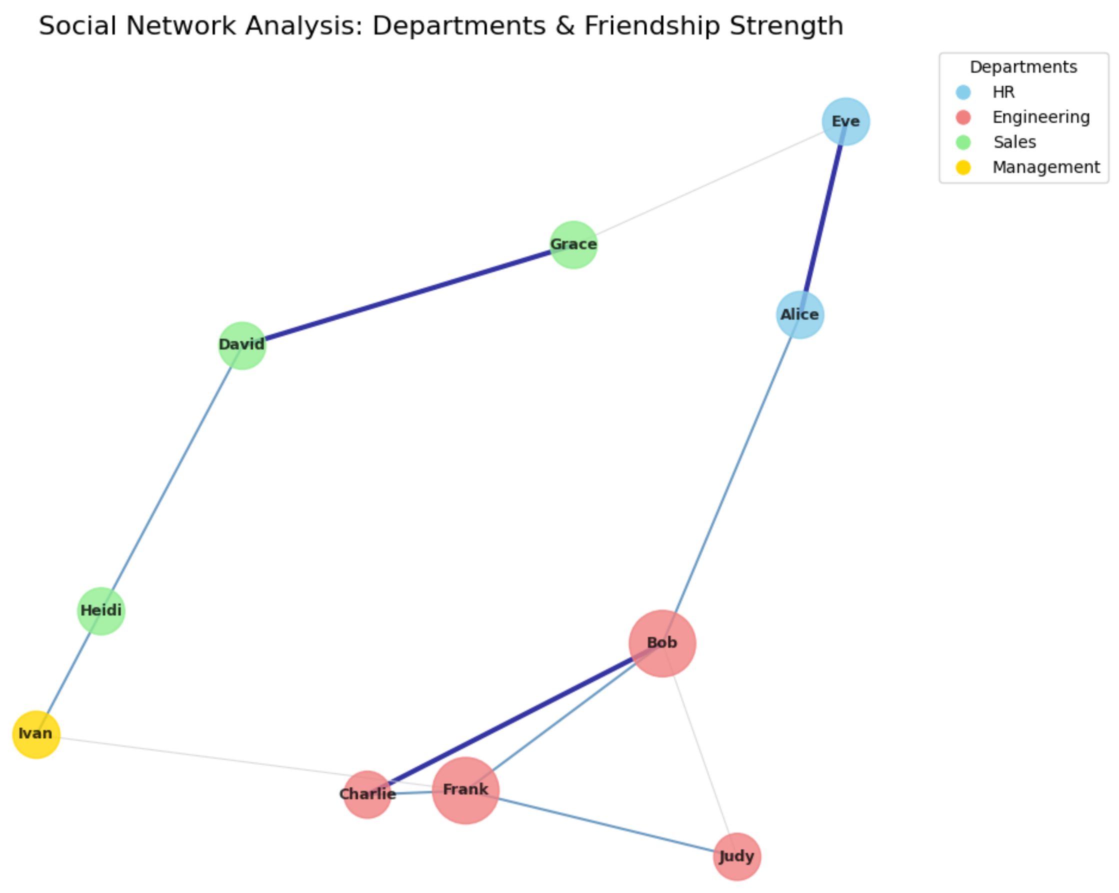

Attack Path Analysis on a Network Graph
Admin
control_account
DomainController
social_engineering
password_reuse
Server2_AdminAccess
lpe_exploit
Alice
Carol
Server2
login
David
login
network_connect
admin_login
Workstation1
Workstation2
login
Bob
network_connect
Server1

6.7 Visualizing and analyzing a simple network graph

Python networkx and matplotlib modules are excellent in visualizing and annotate graphs. As well as computing centrality, degree, and connected components. These are fundamental graph metrics that provide insights into the structure and importance of nodes within a network.

- **Degree:** The number of connections a node has. In-degree and out-degree for directed graphs.
- **Centrality:** Measures of a node's "importance" or "influence" within the network (e.g., how connected it is, how central it is to communication paths).
- **Connected Components:** Groups of nodes in an undirected graph where every node can be reached from every other node within the same group. For directed graphs, we talk about strongly and weakly connected components.

Graph 6.7 computes the critical metrics such as connectedness of employees in a hypothetical company, as well as visual graph of the relationships.



```
--- Graph Elements ---
Nodes: [('Alice', {'department': 'HR', 'age': 30}), ('Bob', {'department': 'Engineering',
'age': 28}), ('Charlie', {'department': 'Engineering', 'age': 35}), ('David', {'department':
'Sales', 'age': 40}), ('Eve', {'department': 'HR', 'age': 25}), ('Frank', {'department':
'Engineering', 'age': 32}), ('Grace', {'department': 'Sales', 'age': 29}), ('Heidi',
{'department': 'Sales', 'age': 45}), ('Ivan', {'department': 'Management', 'age': 50}),
('Judy', {'department': 'Engineering', 'age': 27})]
Edges: [('Alice', 'Bob', {'strength': 2}), ('Alice', 'Eve', {'strength': 3}), ('Bob', 'Charlie',
{'strength': 3}), ('Bob', 'Frank', {'strength': 2}), ('Bob', 'Judy', {'strength': 1}), ('Charlie',
'Frank', {'strength': 2}), ('David', 'Grace', {'strength': 3}), ('David', 'Heidi', {'strength': 2}),
('Eve', 'Grace', {'strength': 1}), ('Frank', 'Ivan', {'strength': 1}), ('Frank', 'Judy', {'strength':
2}), ('Heidi', 'Ivan', {'strength': 2})]

--- Graph Analysis ---

Node Degrees:
  Alice: 2
  Bob: 4
  Charlie: 2
  David: 2
  Eve: 2
  Frank: 4
  Grace: 2
  Heidi: 2
  Ivan: 2
  Judy: 2

Connected Components:
  Component 1: {'Judy', 'Alice', 'Charlie', 'David', 'Grace', 'Heidi', 'Eve', 'Ivan', 'Frank',
'Bob'}

Betweenness Centrality (higher = more 'broker' role):
  Bob: 0.306
  Frank: 0.306
  Alice: 0.236
  Ivan: 0.236
  Eve: 0.181
  Heidi: 0.181
  David: 0.125
  Grace: 0.125
  Charlie: 0.000
  Judy: 0.000

Closeness Centrality (higher = closer to all others):
  Bob: 0.500
  Frank: 0.500
  Alice: 0.450
  Ivan: 0.450
  Charlie: 0.409
  Eve: 0.409
  Heidi: 0.409
  Judy: 0.409
  David: 0.375
  Grace: 0.375

Eigenvector Centrality (higher = more influential connections):
  Bob: 0.540
  Frank: 0.540
  Charlie: 0.381
  Judy: 0.381
  Alice: 0.225
  Ivan: 0.225
  Eve: 0.099
  Heidi: 0.099
  David: 0.054
  Grace: 0.054
```

Module 6: Learning Materials & References:

i. Breadth First Search or BFS for a Graph
ii. https://celerdata.com/glossary/breadth-first-search-bfs
iii. Breadth-first search in 4 minutes video
iv. Depth-first search in 4 minutes video
v. What is Attack Path Analysis?
vi. Graph Algorithms for Technical Interviews
vii. Networkx Tutorials Video - Koolac
viii. Networkx tutorial Includes graph creation, traversal, algorithms, and drawing
ix. *NetworkX : Python software package for study of complex networks*
x. Networkx Gallery

Assessment: Module 6: Graphs & Network Analysis

Part A: Quiz

6.1a What is a graph in computer science?
A) A data structure that only stores numbers
B) A collection of nodes (vertices) and edges that connect pairs of nodes
C) A type of sorting algorithm
D) A way to compress files

6.2a Which traversal method explores all neighbor nodes at the present depth before moving on to nodes at the next depth level?
A) Depth-First Search (DFS)
B) Breadth-First Search (BFS)
C) Topological Sort
D) Dijkstra's Algorithm

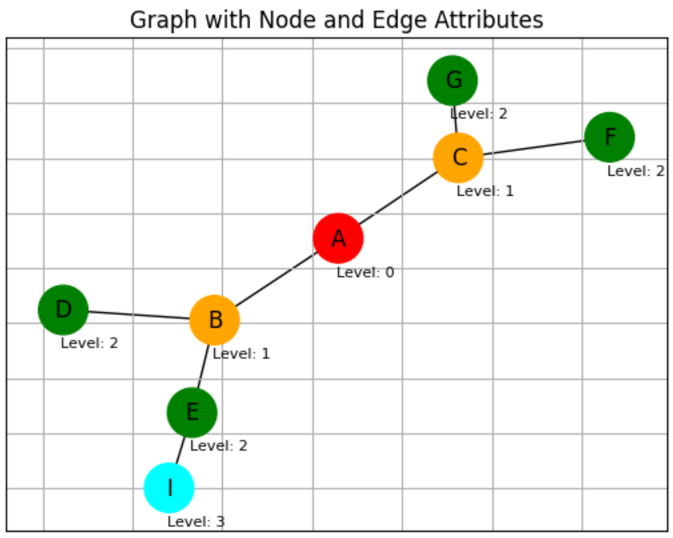
6.3a What is the main data structure used by Depth-First Search (DFS) for traversal?
A) Queue
B) Stack
C) Priority Queue
D) Linked List

6.4a Which graph traversal technique is best suited for finding the shortest path in a weighted graph with non-negative edge weights?
A) Breadth-First Search (BFS)
B) Depth-First Search (DFS)
C) Dijkstra's Algorithm
D) Topological Sort

Part C: Python Exercises

If you are not familiar with networkx and matplotlib, please read or watch the tutorial for networkx in module 6 reference by Koolac.

6.1c Using networkx and matplotlib, plot undirected graph similar to graph-6.1

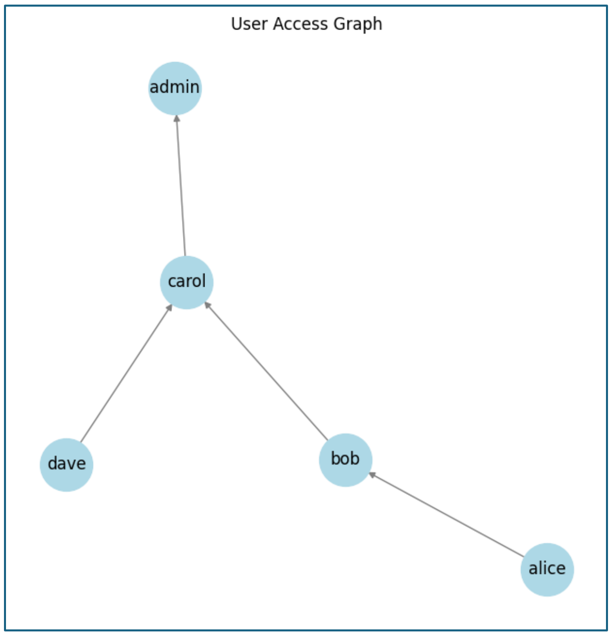


6.2c Shortest Path in a User Access Graph

Imagine you're working in a company where different employees can access each other's accounts to do certain tasks. For example:

- Alice can act on behalf of Bob
- Bob can act on behalf of Carol
- Carol can act on behalf of the Admin

You want to check if Alice can eventually reach the Admin account, by following these trust or access relationships — one person to another — like steps on a ladder.



Write a python script that build the directed graph, and find the shortest path from a given user to an admin account (if one exists)

Why This Is Useful in Cybersecurity:

This is a basic form of attack path analysis:

- If an attacker compromises Alice’s account, can they “hop” through others to reach Admin?
- By finding that path, you can understand lateral movement risks.

6.3c Detect Most Connected Node in a Phishing Network

You are analyzing a communication graph where each node is an email address and edges represent who sent messages to whom. Your goal is to:

- Build an undirected graph from the email data.
- Identify the node with the highest degree centrality (i.e., most connections).

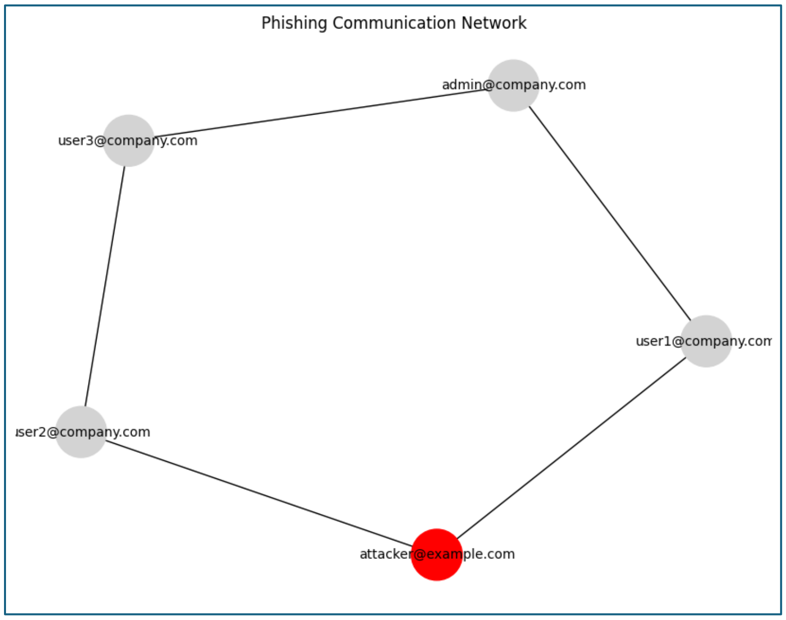
Input:

```
emails = [
    ("attacker@example.com", "user1@company.com"),
    ("attacker@example.com", "user2@company.com"),
    ("user2@company.com", "user3@company.com"),
    ("user3@company.com", "admin@company.com"),
    ("user1@company.com", "admin@company.com")
]
```

Write a python script that creates a undirected graph, compute degree of centrality using nx.degree_centrality, print the node with the highest centrality score

Sample output:

```
Most connected node: attacker@example.com
Centrality score: 0.5
```



6.4c Directed graph of social network connections

Using networkx and matplotlib create a python script that illustrates social network connections of the following individuals using directed graph. Below is the connections of the individuals from a social_following.cvs file.

```
follower,following,duration
Antoine,Dillan,20
Jim,Antoine,45
Jose,Dillan,11
Maria,Antoine,150
Mai,Antoine,17
Abdula,Antoine,100
Kathy,Dillan,5
```

Meaning of Data in social_following.csv
Each row in this file describes a directed connection from one user to another — similar to a social media "follows" relationship.

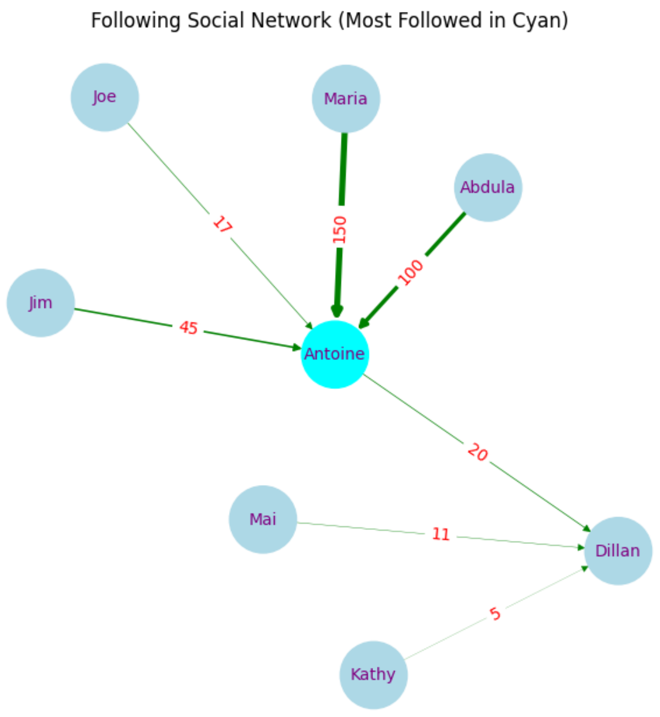
a) follower: the person who is following someone else.
b) following: the person being followed.
c) durations: how long (in hours/days/or month) the follow relationship has existed.

Tasks:

read the csv file, output the most followed users, and the directed graph

Sample output:

Most Followed User(s):
- Antoine (followed by 4 users)
Directed graph as shown on the right.



Module 7: Singly and Doubly Linked Lists

A linked list is a linear data structure where elements are not stored in contiguous memory locations. Instead, each element, or node, contains data and a pointer (or link) to the next node in the sequence.

1. Singly Linked List: Each node has two components: data and a next pointer that references the subsequent node. The list is traversed in a forward direction, starting from the head. The last node's next pointer is None, indicating the end of the list.

2. Doubly Linked List: Each node has three components: data, a next pointer, and a prev (previous) pointer. This allows for traversal in both forward and backward directions. While more flexible, doubly linked lists require slightly more memory per node to store the additional prev pointer.

Memory Management

Linked lists offer dynamic memory allocation. Unlike arrays, a linked list's size can be easily modified during runtime. Nodes are created and stored at various locations in memory, and the pointers connect them. This avoids the need for a contiguous block of memory, which can be inefficient if the size of the collection is unknown beforehand. Memory is allocated for each node as it's added and deallocated when a node is removed, providing efficient memory usage.

7.1 Singly Linked Lists

A linear collection of nodes where each node points to the **next**. The Class Node is the basic data structure where data is stored in individual elements called **nodes**. Each node contains two pieces of information: the actual data it holds and a reference (or pointer) to the very next node in the sequence.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

self.data: This stores the value of the node, such as a number, string, or any other Python object.

self.next = None: This acts as a pointer. Initially, it's set to None because when a node is first created, it doesn't point to anything. In a linked list, this next attribute will be updated to point to the next Node object in the chain. The last node in the list will have its next attribute remain None, signaling the end.

7.1.1 Operations & Performance:

The efficiency of operations is what makes linked lists useful in specific scenarios.

- **Traverse: O(n)** To find a specific node or visit every node in the list, you must start at the head (the first node) and follow each next pointer until you reach the end. The time this takes is directly proportional to the number of nodes in the list, which is denoted as **n**. Therefore, the time complexity is O(n).
- **Insert at head: O(1)** Adding a new node at the very beginning of the list is extremely fast. You simply create the new node, set its next pointer to the current head of the list, and then update the list's head to be this new node. This process takes the same amount of time regardless of the list's size. This is known as constant time, or O(1).
- **Delete after a node: O(1)** 🗑️ If you already have a reference to a node (let's call it node_A), deleting the node immediately following it (node_B) is also a constant time operation. You just change node_A.next to point to node_B.next, effectively bypassing and "unlinking" node_B from the chain.

7.1.2 Pros and Cons

- **Pros: Simple, low memory** Singly linked lists are straightforward to implement. Each node only requires extra memory for a single next pointer, making them memory-efficient compared to other data structures like doubly linked lists.
- **Cons: No backward traversal** The biggest limitation is that you can only move forward through the list. If you are at a particular node, you have no information about the node that came before it. To find the previous node, you would have to traverse the entire list again from the head.

7.2 Doubly Linked list

A **doubly linked list** is a more advanced linked list where each node is connected to both the node before it and the node after it. This structure allows for more flexible navigation compared to a singly linked list.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None
```

- **self.data:** This attribute stores the actual value or payload of the node.

- **self.prev:** This pointer holds the memory address of the **previous** node in the list. For the very first node (the head), this pointer is None.
- **self.next:** This pointer holds the memory address of the **next** node in the list. For the very last node (the tail), this pointer is None.

7.2.1 Operations & Performance:

The performance of a data structure is primarily measured by the time complexity of its operations, described using Big O notation, where **n** is the number of nodes in the list.

Traversal: O(n)

To visit every node in the list, you must start from one end (either the head or tail) and follow the next or prev pointers sequentially. The time required is directly proportional to the number of nodes.
However, a key advantage is **bidirectional traversal**. You can move both forwards and backwards, which is impossible in a singly linked list. This makes operations like finding a predecessor node trivial.

Insertion:

a) **At the Head: O(1)**

- Create a new node.
- Set its next pointer to the current head.
- Set the current head's prev pointer to the new node.
- Update the list's head to be the new node. This is a constant time operation as it doesn't depend on the list's size.

b) **At the Tail: O(1**) (assuming a tail pointer is maintained)

- Create a new node.
- Set its prev pointer to the current tail.
- Set the current tail's next pointer to the new node.
- Update the list's tail to be the new node. This is also a constant time operation.

c) **In the Middle**: **O(n)**

- The primary cost is **finding the insertion point**, which takes O(n) time for traversal.
- Once the location is found, the actual insertion is an O(1) operation involving updating the next and prev pointers of the surrounding nodes.

Deletion:

a) **At the Head**: **O(1)**
   - Update the head to point to the second node.
   - Set the new head's prev pointer to None.
   - This is a constant time operation.

b) **At the Tail**: **O(1)** (assuming a tail pointer is maintained)
   - Update the tail to point to the second-to-last node.
   - Set the new tail's next pointer to None.
   - This is also a constant time operation.

c) **In the Middle: O(1)** (if you have a direct reference to the node to be deleted)
   - This is a major performance advantage. Since each node knows its predecessor, you can instantly "bypass" it.
   - You set node.prev.next = node.next and node.next.prev = node.prev.

Note: If you only have the *data* of the node to delete, you must first traverse the list to find it, making the overall operation O(n).

7.2.2 Performance Summary Table

The table below compares the time complexity of single and doubly linked list operations.

Operation	Singly Linked List	Doubly Linked List	Explanation
Access/Search	O(n)	O(n)	To find an element, must start at the head & traverse the list
Insertion (at Head)	O(1)	O(1)	Adding a node to the beginning is a constant time operation
Insertion (at Tail)	O(1)*	O(1)*	Use the tail node pointer
Insertion (in Middle)	O(n)	O(n)	This requires first traversing to the desired position (an O(n) operation) and then performing the insertion (an O(1) operation).
Deletion (at Head)	O(1)	O(1)	Removing the first node is a constant time operation.
Deletion (at Tail)	O(1)*	O(1)*	constant time using the tail pointer .
Deletion (in Middle)	O(n)	O(n)	Similar to insertion, this requires finding the node first. context.

7.3 Use Cases

Linked lists are particularly useful in scenarios that require frequent insertions or deletions, as these operations are generally more efficient than with arrays.

7.3a Log Management

In log management systems, new log entries arrive sequentially and must be recorded in the order they occur. A linked list is an ideal structure for this.

- Linked lists efficiently store **chronological logs** without resizing memory.
- Ingest logs as a **stream**: append nodes in real-time.
- Use doubly linked list to **scroll forward/backward** in logs during investigations.

```
# Use-case: Fast log append
log_list = LinkedList()
log_list.append("10:01: Firewall breach detected")
log_list.append("10:02: IP blacklisted")
```

7.3b Malware Analysis

In malware analysis, linked lists can be used by malicious software to manage internal data structures in a way that is difficult to track.

- **API Hooking**: Malware often intercepts calls to system APIs. It might use a linked list to store information about the functions it has hooked, allowing it to easily add or remove hooks.
- **Stealth**: Because the nodes are not stored contiguously in memory, it can be harder for security software to detect the full data structure by scanning memory. The malware can store its components in scattered memory locations, linked together by pointers.

7.3c Custom Log Buffer Implementation

A **log buffer** temporarily stores log messages before they are written to a more permanent location, like a file or a database. Implementing this with a linked list can be highly effective.

A custom log buffer can be created using a doubly linked list to cap the number of stored logs (e.g., keeping only the most recent 1,000 entries).

- **Implementation**: A class can be designed to manage the log entries. This class would have a head and tail pointer, along with a max_size attribute.
- **Adding Logs**: When a new log arrives, it's added as the new head.
- **Maintaining Size**: If the list's current size exceeds max_size, the oldest log entry (at the tail) is removed. The doubly linked nature makes removing the tail a very fast O(1) operation.

This approach ensures that the memory usage of the buffer is bounded while providing efficient O(1) time complexity for both adding new logs and removing the oldest ones.

Module 7: Learning Materials & References:

i. Linked List Data Structure
ii. Singly Linked List Tutorial
iii. Doubly Linked List Tutorial

Assessment: Module 7: Singly and Doubly Linked Lists

Part A: Quiz

7.1a What is the primary structural difference between a singly linked list and a doubly linked list?

A. doubly linked list node stores an extra pointer to the previous node.
B. Singly linked lists store data, while doubly linked lists do not.
C. Singly linked list nodes are stored contiguously in memory, while doubly linked list nodes are not.
D. A doubly linked list can only be traversed backward.

7.2a In which scenario does a doubly linked list have a significant performance advantage (O(1)) over a singly linked list (O(n))?

A. Traversing the list to find a specific value.
B. Deleting a node when you have a direct pointer to that specific node.
C. Inserting a new node at the head of the list.
D. Accessing the 5th element in the list.

7.3a What is the main disadvantage of using a doubly linked list compared to a singly linked list?

A.Insertion at the beginning is slower.
B.It has a higher memory overhead.
C.It cannot be used to implement a queue.
D.Searching for an element is always slower.

7.4a Which of the following applications would be best suited for a doubly linked list over a singly linked list?

A. A real-time system that only adds new data to the end of a log file.
B. Implementing a stack data structure (LIFO).
C. Managing a simple print queue where jobs are processed in the order they are received.
D. Implementing a web browser's history for 'back' and 'forward' navigation.

7.5a Which of the following is true about a singly linked list?

A. Each node has pointers to both next and previous nodes
B. Nodes can be traversed in both directions
C. Each node has a pointer to the next node only
D. It uses more memory than a doubly linked list

7.6a . In a doubly linked list, which of the following operations is easier than in a singly linked list?

A. Inserting at the end
B. Inserting at the beginning
C. Reversing the list
D. Accessing the middle node directly

7.7a What does the last node in a singly linked list point to?

A. The first node
B. The previous node
C. Itself
D. Null

Part C: Python Exercises

7.1c Singly Linked List

Write a python class **Singlyll.py** to implement a **singly linked list**, the class must implement the following methods:

isEmpty()
getSize()
addHead()
addAfter (E target, E addItem)
	removeFirst()
	removeLast()
	__str__()

Build the Singly Linked list using the following data: Login Success, File Accessed , Firewall Triggered, Admin Login, Log Cleared, insert the data from the head.

7.1c Doubly Linked List

Write a python class **Doublell.py** to implement a doubly linked list with the following methods:

isEmpty()
getSize()
addHead()
addAfter (E target, E addItem)
	removeFirst()
	removeLast()
	__str__()

Also write a function **reverse_doubly(head)** that reverses a **doubly linked list** in-place and prints the reversed list from head to tail.

Build the Doubly Linked list using the following data: Login Success, File Accessed , Firewall Triggered, Admin Login, Log Cleared, insert the data from the head.

Module 8: Trees and Tree Traversal

This module provides a comprehensive overview of tree data structures, their traversal methods, and their applications in organizing hierarchical data and in cybersecurity.

8.1 Tree Traversal, Binary Trees, and Search Trees

In computer science, a **tree** is a hierarchical data structure that consists of nodes connected by edges. Each tree has a **root** node, and every other node has a **parent** node. Nodes can also have **child** nodes. Nodes with no children are called **leaf** nodes.

8.1.1 Binary Trees

A **binary tree** is a specific type of tree where each node has at most two children, referred to as the **left child** and the **right child**.

Binary Search Trees (BST)

A **Binary Search Tree** is a special type of binary tree with the following properties:

- The value of each node in the left subtree is less than or equal to the value of the parent node.
- The value of each node in the right subtree is greater than the value of the parent node.
- Both the left and right subtrees are also binary search trees.

This ordering makes searching for elements highly efficient.

8.1.2 Tree Traversal Methods

Tree traversal is the process of visiting each node in a tree exactly once. There are three common methods for traversing a binary tree:

8.1.2a In-order Traversal

In an in-order traversal, the left subtree is visited first, followed by the root node, and then the right subtree. The sequence is: **Left -> Root -> Right**. In a Binary Search Tree, an in-order traversal visits the nodes in ascending order.

Pseudocode

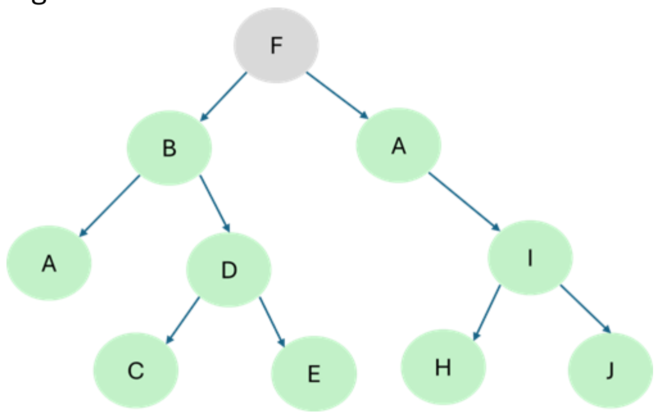
```
in_order (node)
    if node == null return
    in_order(node.left)
```

1. Traverse left
2. Visit Node
3. Traverse Right

```
    visit(node)
    in_order(node.right)
```

Example:

Fig-Tree1



For the tree in Fig-Tree1

The in-order traversal would be A, B, C, D, E, F, G, I, H, J

Time Complexity: O(n), n ⍰ number of nodes in the tree

8.1.2b Pre-order Traversal

In a pre-order traversal, the root node is visited first, followed by the left subtree, and then the right subtree. The sequence is: **Root -> Left -> Right**. This is useful for creating a copy of a tree.

Pseudocode

```
pre_order (node)
    if node == null return
    visit(node)
    pre_order(node.left)
    pre_order(node.right)
```

1. Visit Node
2. Traverse left
3. Traverse Right

Example:

For the same tree, the pre-order traversal would be: **F, B, A, D, C, E, G, I, H, J**

Time Complexity: O(n), n → number of nodes in the tree

8.1.2c Post-order Traversal

In a post-order traversal, the left subtree is visited first, then the right subtree, and finally the root node. The sequence is: **Left -> Right -> Root**. This is useful for deleting nodes from a tree.

Pseudocode

```
post_order (node)
        if node == null return
        post_order(node.left)
        post_order(node.right)
        visit(node)
```

1. Visit Node
2. Traverse left
3. Traverse Right

Example:

For the same tree, the post-order traversal would be: **A, C, E, D, B, H, J, I, G, F**

Time Complexity: O(n), n → number of nodes in the tree

Applications of Trees

- **Binary Trees:** Used in various applications such as expression parsing and data compression (Huffman coding).
- **Binary Search Trees:** Ideal for searching, inserting, and deleting data efficiently. They are used in databases, symbol tables in compilers, and in implementing sorting algorithms.

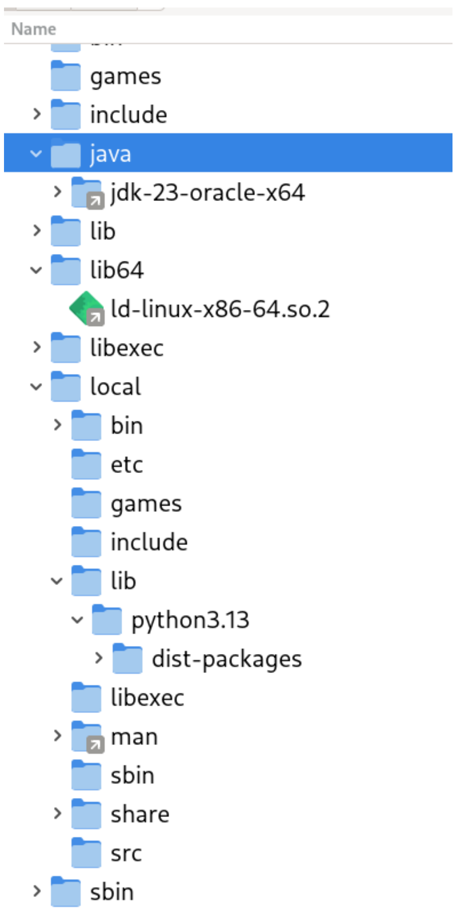
8.2 Organizing Hierarchical Data

Trees are a natural way to represent hierarchical data.

8.2a File Systems

A classic example of a hierarchical data structure is a computer file system. The root directory is the root of the tree. Directories (or folders) are nodes that can contain other directories (child nodes) and files (leaf nodes). The path to a file or directory represents a path from the root to that specific node.

Example:



8.2b Attack Trees

In cybersecurity, **attack trees** are used to model potential threats to a system. The root of the tree represents the primary goal of an attacker (e.g., "Steal User Data"). The child nodes represent the different ways an attacker could achieve that goal. These sub-goals can be broken down further into more specific actions.

Attack trees use **AND** and **OR** nodes:

- **OR nodes:** Represent alternative ways to achieve a goal. The attacker only needs to accomplish one of the child node's objectives.
- **AND nodes:** Represent a sequence of steps that must all be completed to achieve the goal.

8.3 Decision Trees for Anomaly Detection and Signature Classification

Decision Trees are a type of supervised machine learning algorithm that can be used for both classification and regression tasks.

A decision tree is a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. Think of it as a flowchart of "if-then" questions.

Example: Should I Play Golf?

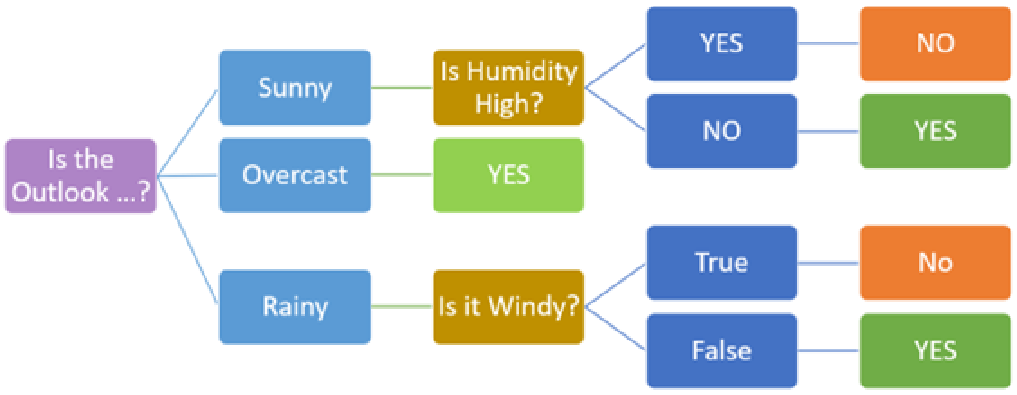
Imagine we have a dataset that records the weather conditions and whether someone went to play golf. The goal is to create a model that predicts if someone will play golf based on the weather.

Dataset:

Outlook	Temperature	Humidity	Windy	Play Golf
Sunny	Hot	High	False	No
Sunny	Hot	High	True	No
Overcast	Hot	High	False	Yes
Rainy	Mild	High	False	Yes
Rainy	Cool	Normal	False	Yes
Rainy	Cool	Normal	True	No
Overcast	Cool	Normal	True	Yes
Sunny	Mild	High	False	No
Sunny	Cool	Normal	False	Yes
Rainy	Mild	Normal	False	Yes

The Decision Tree:

Based on this data, a machine learning algorithm would learn which questions are most important to ask to decide. It would generate a tree like this:



How it Works:

The tree uses a series of questions to classify a new day.

1. **Start at the Root Node:** The most important feature the model found is **Outlook**. This is the first question it asks.
2. **Follow the Branches:** Based on the answer, you follow a path.

    a) If the outlook is **Overcast**, the tree immediately predicts **Yes**, you will play golf.
    b) If the outlook is **Sunny**, it asks a follow-up question: "Is the humidity high?" If yes, the prediction is **No**. If no, the prediction is **Yes**.
    c) If the outlook is **Rainy**, it asks another question: "Is it windy?" If true, the prediction is **No**. If false, the prediction is **Yes**.

This simple structure, with its clear "if-then" logic, makes decision trees one of the most understandable models in machine learning.

8.3b Anomaly Detection

In the context of cybersecurity, anomaly detection is the process of identifying unusual patterns that do not conform to expected behavior. Decision trees, particularly when used in an ensemble method like an **Isolation Forest**, are effective for this. An Isolation Forest is a collection of decision trees that "isolate" outliers or anomalies in a dataset. Anomalies are typically easier to separate from the rest of the data, so they are found closer to the root of the trees in the forest.

8.3c Signature Classification

Signature classification involves identifying known malicious patterns, such as virus signatures or network attack patterns. A decision tree can be trained on a dataset of labeled network traffic (i.e., "malicious" or "benign"). The tree learns a set of rules based on the features of the traffic (e.g., packet size, protocol, port number) to classify new, unseen traffic. Each path from the root to a leaf in the decision tree represents a classification rule.

8.3d Tree Traversal in Attack Trees

While the classic in-order, pre-order, and post-order traversals are primarily for binary trees, the concept of systematically visiting nodes is crucial for analyzing attack trees. Analyzing an attack tree involves traversing it to understand the different attack paths.

The goal is typically to evaluate the feasibility, cost, or likelihood of success for each path. This is often done using a **depth-first search (DFS)**, which is conceptually similar to pre-order traversal. The analysis starts at the root (the attacker's main goal) and explores each branch (sub-goal) down to the leaf nodes (the specific actions).

For example, to find the cheapest attack path, you would traverse the tree, calculating the cumulative cost of each path. A post-order-like traversal could be used to calculate the value of parent nodes based on the values of their children. For an **OR** node, the parent's value would be the minimum of its children's values (representing the easiest path). For an **AND** node, the parent's value would be the sum of its children's values (representing the total effort for all required steps).

8.4 Tree Terminology

1. Root a node without Parent (F)
2. Internal node, node with a child (F,B,D,G,I)
3. External Node or leaf, node without children (A,C,E,H,J)
4. Depth of a node, number of ancestors or edges ↑
5. Height of a tree maximum depth of any node ↓

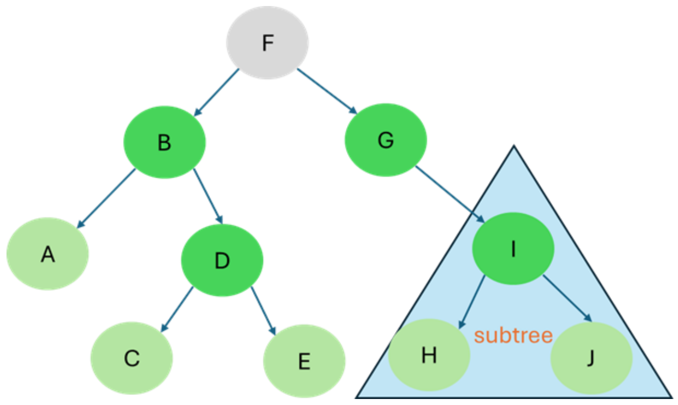


Fig-Tree2

8.5 Binary Search Tree (BST)

A Binary Search Tree, or BST, is a fundamental data structure in computer science used for organizing and storing data in a sorted manner, which allows for efficient searching, insertion, and deletion of items.

A Binary Search Tree is a type of binary tree, meaning each node has at most two children. What makes it special is a specific set of rules that it must follow for every node:

- **The Left Subtree Rule:** All values in a node's left subtree must be *less than* the node's own value.
- **The Right Subtree Rule:** All values in a node's right subtree must be *greater than* the node's own value.
- **The BST Property:** Both the left and right subtrees must also be binary search trees themselves.
- **No Duplicates:** Typically, BSTs do not allow duplicate values.

These rules create a structure where data is inherently ordered.

Consider the example of Fig-BST1:
In this example, 8 is the root node.

Notice that all numbers in its left subtree (3, 1, 6, 4, 7) are less than 8, and all numbers in its right subtree (10, 14) are greater than 8.

This rule applies recursively down the tree. For instance, for the node 3, its left child 1 is smaller, and its right child 6 is larger.

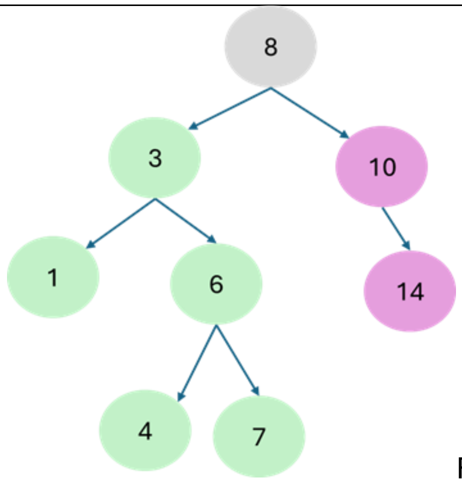


Fig-BST1

The sorted nature of a BST makes its core operations very efficient.

8.5.1a Searching (Finding a Value)

Searching for a value is the primary strength of a BST. The process is as follows:

1. Start at the root node.
2. Compare the value you are looking for with the current node's value.
3. If the values match, you've found the item.
4. If your value is **less than** the current node's value, move to the **left child**.
5. If your value is **greater than** the current node's value, move to the **right child**.
6. Repeat steps 2-5 until you find the value or you reach a dead end (a null node), which means the value is not in the tree.

Because you discard half of the remaining tree at each step, the search time is very fast.

8.5.1b Insertion (Adding a Value)

To insert a new value, you first perform a search to find where it *should* be:

1. Start at the root and search for the value you want to insert.
2. The search will eventually end at a null node (a spot where a child could be).
3. Create the new node and attach it in that empty spot, maintaining the left-less-than, right-greater-than rule.

For example, to insert the value 13 into the tree above, you would traverse from 8 to 10 to 14 and then insert 13 as the left child of 14.

8.5.1c Deletion (Removing a Value)

Deleting a node is the most complex operation because you must ensure the BST properties are maintained after the node is removed. There are three cases:

1. **Deleting a Leaf Node (No Children):** This is the simplest case. You can just remove the node from the tree by setting its parent's pointer to `null`.
2. **Deleting a Node with One Child:** You can remove the node and replace it with its single child.
3. **Deleting a Node with Two Children:** This is the tricky case. You must replace the node's value with its **in-order successor** (the smallest value in its right subtree) or its **in-order predecessor** (the largest value in its left subtree). After copying the value, you then delete the successor/predecessor node from its original position, which will be a simpler deletion case (case 1 or 2).

8.6.2 Advantages

- **Efficient Searching:** Because of the ordered structure, searching is very fast, with an average time complexity of O(logn), where 'n' is the number of nodes. This is significantly faster than the O(n) search time for an unsorted array or linked list.
- **Efficient Insertions and Deletions:** Like searching, these operations are also typically O(logn) on average.
- **Keeps Data Sorted:** A BST naturally keeps its elements in a sorted order. If you traverse the tree *in-order* (left subtree, root, right subtree), you will visit the nodes in ascending order.

8.6.3 Disadvantages

- **Unbalanced Trees:** The major drawback of a BST is that its efficiency depends on the tree being **balanced**. If you insert data that is already sorted (e.g., 1, 2, 3, 4, 5), the tree will become completely unbalanced and essentially turn into a linked list.
- **Worst-Case Performance:** In an unbalanced tree, the height of the tree can be 'n'. This degrades the performance of search, insert, and delete operations to O(n), which is the same as a simple list. More advanced trees like AVL trees or Red-Black trees were invented to solve this balancing problem.

Module 8: Learning Materials & References:

i. Python Trees
ii. Python Binary Trees
iii. Python Binary Search Trees
iv. Binary Search Trees in Python: Checking the BST Property Video

Assessment: Module 8: Trees and Tree Traversal

Part A: Quiz

8.1a Which tree traversal method, when applied to a Binary Search Tree, will always visit the nodes in ascending sorted order?

A. In-order
B. Level-order
C. Post-order
D. Pre-order

8.2a In a Binary Search Tree, where would you expect to find a node with a value smaller than the root node?

A. It could be in either the left or right subtree.
B. In the right subtree of the root.
C. Only as a direct child of the root.
D. In the left subtree of the root.

8.3a Which of the following is a common real-world example of a hierarchical data structure organized as a tree?

A. A queue of customers at a store.
B. A shopping list.
C. A deck of playing cards.
D. A computer's file system.

8.4a In the context of cybersecurity, what does an 'AND' node signify in an attack tree?

A. The attack path has failed.
B. There are multiple alternative ways to achieve the parent goal.
C. All child nodes (sub-goals) must be successfully completed to achieve the parent goal.
D. A single, specific action that must be taken.

8.5a Pre-order Traversal

```python
#Consider the following Python code for a binary tree. What is the correct pre-order traversal
sequence for the created tree?
#python
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

# Create a tree
root = Node('F')
root.left = Node('B')
root.right = Node('G')
root.left.left = Node('A')
root.left.right = Node('D')
root.right.right = Node('I')
```

A. F, B, G, A, D, I
B. F, B, A, D, G, I
C. A, B, D, F, G, I
D. A, D, B, I, G, F

8.6a Calculate the Height of a Binary Tree

```python
# Which of the following Python functions correctly calculates the height of a binary tree?
# The height is defined as the number of nodes along the longest path from the root node
# down to the farthest leaf node.

class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
```

A.

```python
def getHeight(node):
    if node is None:
        return 0
    return 1 + max(getHeight(node.left), getHeight(node.right))
```
B.
```
def getHeight(node):
    if node is None:
        return 0
    return 1 + getHeight(node.left)
```

C.
```
def getHeight(node):
    height = 0
    while node.left:
        height += 1
        node = node.left
    return height
```
D.
```
def getHeight(node):
    if node is None:
        return 0
    return getHeight(node.left) + getHeight(node.right) + 1
```

8.7a In a decision tree, what does the first node at the very top of the tree (the root node) represent?

A. A random feature from the dataset.
B. The feature with the fewest unique values.
C. The feature that provides the most information to split the data.
D. The final prediction or outcome.

8.8a What is the major advantage of using a Decision Tree model compared to more complex models like neural networks?

A .They are always more accurate.
B. They perform best on small datasets only.
C. They require no data preparation.
D. They are highly interpretable ('white-box' models).

Part C: Python Exercises

8.1c BST185 Implementation

Write a Python class BST185 to implement a Binary Tree:

The class must implement the following methods:

1/ in_order traversal

2/ pre_order traversal

3/ post_order traversal

4/ visit node method outputs the node element on the screen

5/ insert(item) according to BST, ie, left if node is less than root, right if >

6/ delete (item) if children exist promote the children after deleting the node

7/ search(item) true if found in the BST

8/ implement __str__ for node

8.2c Determine if a Binary Tree is BST

Write a Python method using 8.1c to check if a tree is BST.

Module 9: Large Language Models (LLMs)

This module provides a comprehensive exploration of leveraging Large Language Models (LLMs) to enhance cybersecurity practices. You will gain insights into the landscape of open-source LLMs, learn to integrate them with Python for automating security tasks, and master the art of prompt engineering for effective communication with these powerful models. The module culminates in practical applications, including automating threat intelligence gathering, summarizing complex log files, and generating structured security reports.

9.1 Introduction to Open-Source LLMs

This module introduces the rapidly evolving world of open-source Large Language Models and their growing significance in the field of cybersecurity. We will explore various models, their capabilities, and the platforms used to evaluate their performance.

A landscape of powerful and accessible open-source Large Language Models (LLMs) has emerged, providing alternatives to proprietary models from companies like OpenAI and Google. These open-source models are developed by a variety of organizations and research institutions and are made available to the public for use, modification, and distribution. Their popularity is fueled by the transparency, customizability, and cost-effectiveness they offer.

9.1a General-Purpose Models

These models are designed to be versatile and can handle a wide range of tasks, including text generation, summarization, translation, and question answering.

I. LLaMA 3.1: Developed by Meta AI, the Llama series of models is highly popular and widely used as a foundation for many other open-source projects. Llama 3.1, the latest iteration, offers significant improvements in reasoning and multilingual capabilities. It comes in various sizes, making it adaptable to different hardware requirements.

II. Mistral & Mixtral Models: French AI startup Mistral AI has released a series of powerful and efficient models. Mistral 7B is highly regarded for its performance despite its relatively small size. The Mixtral models, such as Mixtral 8x7B, utilize a "Mixture of Experts" (MoE) architecture, which allows them to have a large number of parameters while being computationally efficient during inference.

III. Gemma 2: Released by Google, Gemma models are built from the same research and technology used to create the Gemini models. They are available in a range of sizes and are designed for responsible AI development. Gemma 2 offers improved performance and is a strong contender in the open-source space.

9.1b Models Specialized for Coding

These models are specifically fine-tuned for software development tasks, such as code generation, completion, and debugging.

I. Code Llama: Also from Meta AI, Code Llama is a family of models built on top of Llama 2 and fine-tuned for coding tasks. It supports a wide range of programming languages and is available in various sizes to suit different needs.

II. DeepSeek Coder: Developed by DeepSeek AI, this model has gained recognition for its strong performance in code generation and understanding. It has been trained on a massive dataset of code from GitHub and other sources.

III. StarCoder 2: A collaboration between BigCode and NVIDIA, StarCoder 2 is trained on a large and diverse dataset of source code. It excels at generating code from natural language descriptions and is designed to be a powerful tool for developers.

9.1c Models for Creative Writing and Chat

While many of the general-purpose models perform well in creative writing, some models and fine-tuned versions are particularly noted for their capabilities in this area.

I. Vicuna: Developed by researchers from institutions like UC Berkeley, CMU, Stanford, and MBZUAI, Vicuna is a chat-optimized model fine-tuned from LLaMA. It is known for its impressive conversational abilities and performance that is competitive with early versions of ChatGPT.

II. Zephyr: This is a series of fine-tuned models, often based on Mistral models, that are specifically optimized for conversational AI and instruction following. They are known for providing helpful and engaging responses.

III. The open-source LLM landscape is dynamic, with new models and improved versions being released frequently. The choice of the "best" model often depends on the specific use case, hardware constraints, and desired level of customization. Platforms like Hugging Face serve as central hubs for accessing, comparing, and experimenting with these and many other open-source models.

9.1d Core Capabilities for Cyber Defense:

Learn how these models are applied to critical cybersecurity functions, including:

  - **Cyber Threat Intelligence (CTI):** Automatically gathering and analyzing threat data.
  - **Entity Extraction:** Identifying indicators of compromise (IoCs) such as IP addresses, domains, and file hashes from unstructured text.
  - **Log Summarization:** Condensing voluminous log data into concise, actionable summaries.
  - **Anomaly Detection:** Identifying unusual patterns in network traffic and system logs that may indicate a security breach.

9.3 Integrating Python with LLM APIs for Cybersecurity Tasks

This section focuses on the practical integration of LLMs into cybersecurity workflows using Python. You will learn how to interact with LLM APIs and build custom tools to automate security-related tasks.

9.3a Harnessing LLM APIs with Python:

Explore Python libraries designed for seamless interaction with LLM APIs. We will delve into libraries like **llm** and **Masked-AI**, understanding their functionalities for sending prompts, receiving responses, and managing conversations with models.

9.3b Securely Handling Sensitive Data:

A critical aspect of using public LLM APIs is ensuring the privacy and security of the data being processed. The **Masked-AI** library will be introduced as a practical solution for redacting sensitive information before it is sent to an external model, mitigating potential data leakage.

9.3c Automating Security Development Tasks:

Discovering how to leverage LLMs for:

1. **Code Generation:** Automatically generating scripts for security analysis and automation.
2. **Debugging Assistance:** Quickly identifying and resolving errors in security-focused code.
3. **Vulnerability Explanation:** Gaining a deeper understanding of security vulnerabilities through natural language explanations.

9.4 Automating Threat Intelligence Gathering with LLMs

This section explores the transformative impact of LLMs on the field of threat intelligence. You will learn how to automate the collection, analysis, and reporting of threat information.

9.4.1 Summarizing Log Files with LLM Prompts

- **Taming the Data Deluge:** Log files are a foundational element of security analysis but can be overwhelmingly large. Learn techniques like the **MapReduce** approach to break down and summarize extensive log data, making it manageable and comprehensible.
- **Crafting Effective Summarization Prompts:** The quality of a summary depends heavily on the prompt. This section will cover how to structure prompts to extract the most relevant information and generate concise, human-readable summaries of complex log entries.

9.4.2 Creating Structured Security Reports (e.g., CVE Analysis Summaries, Incident Narratives)

- **From Raw Data to Actionable Reports:** Move beyond simple summarization to generating structured and formatted security reports. Explore the use of frameworks like **LangChain** and **NVIDIA AI blueprints** to create detailed and organized outputs.
- **Automated CVE Analysis and Incident Narratives:** Learn to develop prompts that instruct an LLM to analyze a Common Vulnerabilities and Exposures (CVE) entry and produce a summary of its key aspects, potential impact, and mitigation strategies. Similarly, discover how to generate coherent and chronological incident narratives from a collection of security event logs.

9.4.3 Prompt Engineering to Extract or Translate Technical Log Output into Human-Readable Text

- **The Art and Science of Prompt Engineering:** Effective communication with LLMs is key to unlocking their full potential. This lesson will provide a structured approach to prompt engineering, focusing on clarity, context, and desired output format.
- **Best Practices for Technical Translation:** Master the techniques for transforming cryptic log data into clear and understandable text. This includes:
    - **Providing Specific Instructions:** Clearly defining the task and the expected format of the response.

- **Utilizing Few-Shot Learning:** Including examples of the desired input-to-output transformation within the prompt to guide the model.
- **Defining the Output Structure:** Explicitly stating the desired format, such as JSON, Markdown, or a simple narrative, to ensure consistent and parseable results.

Running Large Language Models (LLMs) locally on your own hardware using platforms like Ollama has become increasingly popular, offering a different set of advantages and disadvantages compared to using remote, cloud-based services. Here's a breakdown of the pros and cons of each approach.

9.5 Running Local LLMs (e.g., with Ollama)

Ollama is a tool that simplifies the process of downloading, setting up, and running open-source LLMs like Llama 3.1, Mistral, and Gemma on your personal computer.

9.5a Pros of Running a Local LLM

I. **Privacy and Data Security:** This is the most significant advantage. When you run an LLM locally, your data, prompts, and the model's outputs never leave your machine. This is crucial for handling sensitive, confidential, or proprietary information, making it ideal for both personal privacy and business security.

II. **Cost-Effectiveness at Scale:** While there's an initial hardware investment, running a local model is free of per-use charges. For heavy or continuous usage, this can be significantly cheaper than paying per API call to a remote service.

III. **Offline Accessibility:** Once the models are downloaded, you can use them without an internet connection. This provides reliability and makes it possible to work from anywhere, regardless of connectivity.

IV. **Customization and Fine-Tuning:** You have complete control over the model. You can fine-tune open-source models on your own datasets to create specialized versions tailored to specific tasks or domains. Ollama makes it easy to create and manage these custom models.

V. **No Censorship or Use-Case Restrictions:** You are not bound by the content filters or acceptable use policies imposed by remote service providers. This allows for a wider range of experimentation and application.

VI. **Speed for Smaller Tasks:** For less complex prompts, local models can provide near-instantaneous responses, as there is no network latency involved in sending data to and from a remote server.

9.5b Cons of Running a Local LLM

I. **Significant Hardware Requirements:** Running LLMs, even smaller ones, requires substantial resources. You'll need a powerful CPU, a significant amount of RAM (often 16GB at a minimum, with 32GB or more recommended), and, most importantly, a modern GPU with ample VRAM (8GB is a starting point, with 16-24GB being ideal for larger models).
II. **Performance and Model Size Limitations:** Your local hardware will limit the size and complexity of the models you can run effectively. State-of-the-art models with hundreds of billions of parameters (like OpenAI's full GPT-4) are generally too large for consumer-grade hardware. This means you may be working with less powerful models.
III. **Technical Setup and Maintenance:** While tools like Ollama have made the process much simpler, there can still be a technical learning curve. You are responsible for installation, updates, managing model files, and troubleshooting any hardware or software compatibility issues.
IV. **Slower for Complex Tasks:** For very long and complex prompts that require deep reasoning, even a powerful local setup can be slower than the massively parallelized infrastructure used by large-scale remote services.
V. **Energy Consumption:** Running high-performance hardware, especially a GPU, under a constant load can lead to a noticeable increase in your electricity bill.

9.6 Using Remote LLMs (e.g., OpenAI API, Google AI Platform, Anthropic)

This involves sending your prompts over the internet to a server owned by a company that hosts the LLM and receiving the response back via an API.

9.6a Pros of Using a Remote LLM

I. **Access to State-of-the-Art Models:** You gain access to the most powerful and advanced LLMs on the market (e.g., GPT-4o, Claude 3.5 Sonnet), which are not available to run locally. These models generally offer superior reasoning, creativity, and knowledge.

II. **No Hardware Requirements:** All the computation is done on the provider's servers. You can access these powerful models from any device with an internet connection, including lightweight laptops or smartphones.

III. **Ease of Use and Managed Infrastructure:** Remote services are designed to be "plug-and-play." You don't need to worry about setup, maintenance, updates, or managing infrastructure. The API is generally well-documented and straightforward to integrate.

IV. **Scalability:** These services are built to handle massive volumes of requests. If your application suddenly needs to scale from a few users to millions, the remote infrastructure can handle it without any changes on your part.

V. **Faster for Highly Complex Tasks:** The vast computational resources of cloud providers mean they can often process very demanding tasks faster than a local machine could.

9.6b Cons of Using a Remote LLM

I. **Privacy and Data Security Risks:** You are sending your data to a third party. While providers have security measures in place, your data may be stored on their servers and could potentially be used for model training (though many offer opt-out options). This is often a deal-breaker for sensitive applications.

II. **Ongoing Costs:** Most remote services operate on a pay-as-you-go model, typically charging per token (a unit of text). For frequent or heavy use, these costs can accumulate quickly and become very expensive.

III. **Requires Internet Connectivity:** You must have a stable internet connection to use a remote LLM. There is no offline access.

IV. **Network Latency:** There is always a delay as your prompt travels to the remote server and the response travels back. While often minimal, this latency can be noticeable and problematic for real-time applications.

V. **Censorship and Restrictions:** Remote services have content filters and usage policies that can restrict certain types of prompts or outputs. Your access could also be revoked if you violate their terms of service.

VI. **Lack of Customization:** While some services offer fine-tuning capabilities, you have far less control over the model's architecture and fundamental behavior compared to running an open-source model locally.

The choice between a local and a remote LLM depends entirely on your priorities.

I. Choose a Local LLM (Ollama) if your primary concerns are privacy, data security, cost control for high-volume use, and the ability to customize models.
II. Choose Remote LLM if you need access to the most powerful models available, have minimal hardware, require massive scalability, and prioritize ease of use over data privacy.

9.7 Summary of Local vs. Remote LLM

Feature	Local LLM (with Ollama)	Remote LLM (API Service)
Privacy	**Excellent** (Data never leaves your machine)	**Poor to Fair** (Data sent to a third party)
Cost	**High initial hardware cost, then free**	**Low/no initial cost, pay-per-use**
Performance	Limited by your hardware; faster for simple tasks	Access to SOTA models; faster for complex tasks
Hardware	**Requires powerful CPU, high RAM, and GPU**	Minimal; any device with internet works
Accessibility	**Offline capable**	Requires stable internet connection
Customization	**High** (Full control and fine-tuning)	**Low to Moderate** (Limited by API options)
Ease of Use	Moderate (Requires setup and management)	**Excellent** (Plug-and-play API)
Model Choice	Limited to open-source models that fit your hardware	Access to the largest, most powerful proprietary models
Censorship	**None**	Subject to provider's content policies

Module 9: Learning Materials & References:

I. Getting started with Local AI Video
II. Setup a Secure Local LLM for Cyber Security! (Open WebUI + Ollama) Video
III. Top 10 LLM Prompts Every Cybersecurity Professional Should Know to Boost Security
IV. https://github.com/tmylla/Awesome-LLM4Cybersecurity
V. Top Eight Large Language Models Benchmarks for Cybersecurity Practices
VI. LLM-assisted Malware Review: AI and Humans Join Forces to Combat Malware

Assessment: Module 9: Large Language Models (LLMs)

Part A: Quiz

9.1a A Cybersecurity company wants to use an LLM to analyze client infrastructure compliance checks. Which of the following factors makes running a local LLM with a tool like Ollama the most suitable choice?

a) Access to the largest, state-of-the-art models for the best possible analysis.
b) The ability to scale quickly to handle millions of documents per hour.
c) The guarantee that sensitive client data will not leave the company's internal servers.
d) The ease of use and minimal technical setup required.

9.2a A developer wants to experiment with LLMs on their personal laptop, which has limited processing power and RAM. They are concerned about high costs and complex setup. What is the primary disadvantage of choosing a local LLM in this scenario?

a) The developer will have to pay a subscription fee to use Ollama.
b) The hardware requirements for running even smaller models may exceed their laptop's capabilities.
c) The developer will not be able to use the LLM without an active internet connection.
d) The local LLM will have restrictive content filters, limiting experimentation.

9.3a When comparing the cost models of local versus remote LLMs for an application with very high and continuous usage, which statement is most accurate?

a) A remote LLM involves ongoing operational costs that can become very high, whereas a local LLM has a high initial hardware cost but no per-use fees.
b) A remote LLM is always cheaper due to the pay-per-use model.
c) A local LLM has no costs associated with it.
d) Both local and remote LLMs have identical cost structures based on tokens processed.

9.4a A startup is building a web application that needs to offer its users access to the most powerful, cutting-edge language models like OpenAI's latest GPT or Anthropic's Claude 3.5 Sonnet. Why would they choose a remote LLM API?

a) Because it allows for deep customization and fine-tuning of the model's core architecture.
b) Because it guarantees offline functionality for their web application.
c) Because it is the only way to avoid censorship and content moderation.
d) Because these specific, state-of-the-art proprietary models are not available for local download and installation.

Part B: Short Answers

9.1b Incident Summary for a Non-Technical Audience

You are a cybersecurity analyst. The following is a raw firewall log entry showing a series of blocked connection attempts from a suspicious IP address. Your manager, who is non-technical, needs to understand what happened and what the potential risk is.

Write a prompt for a Large Language Model that will translate the technical log data below into a clear, concise, and easy-to-understand summary. The summary should explain the event, identify the key pieces of information (who, what, where), and state the outcome (the connections were blocked).

Log Data:

```
TIME: 2025-07-13 18:45:12, ACTION: DENY, PROTO: TCP, SRC_IP: 185.220.101.38, DST_IP:
192.168.1.100, DST_PORT: 22 (SSH), RULE: Block_Known_Bad_Actors
TIME: 2025-07-13 18:45:13, ACTION: DENY, PROTO: TCP, SRC_IP: 185.220.101.38, DST_IP:
192.168.1.100, DST_PORT: 22 (SSH), RULE: Block_Known_Bad_Actors
TIME: 2025-07-13 18:45:15, ACTION: DENY, PROTO: TCP, SRC_IP: 185.220.101.38, DST_IP:
192.168.1.100, DST_PORT: 22 (SSH), RULE: Block_Known_Bad_Actors
```

(This question assesses your ability to use an LLM for translation and summarization, a key skill from Module 9.3.3. It requires you to engineer a prompt that specifies the desired output format and audience. **See Module 9 Reference III**)

9.2b Choosing the Right Tool for the Job (Local vs. Remote)

You have access to two types of LLMs:

1. **A local model running on Ollama:** It's private, fast for simple tasks, and based on an open-source model from early 2025.
2. **A remote API for a cutting-edge proprietary model:** It has access to the very latest information up to the current date (July 2025) but its usage is logged by a third-party company.

Create two distinct prompts, one for each LLM, based on the following request from your team lead:

"I need to draft a security advisory about the 'Flicker-Fade' vulnerability that was just disclosed two days ago. I also need to include a Python script that scans our internal network for a specific, proprietary server software we use called 'IntraServe v3.1' to see if it's vulnerable."

- **Prompt A (For the Remote LLM):** Write the prompt you would use to get information about the new vulnerability.
- **Prompt B (For the Local LLM):** Write the prompt you would use to generate the Python scanner script. Explain *why* you chose the local model for this specific task.

(This question assesses your understanding of the strategic trade-offs between local and remote LLMs. It forces you to consider the strengths of each—up-to-the-minute knowledge for the remote API and privacy/security for the local model handling internal, proprietary information.)

Part C: Python Exercises

9.1c Extracting Indicators of Compromise from LLM Output

After you query an LLM about a threat, you need to programmatically extract structured data, like Indicators of Compromise (IOCs), from its natural language response.

Write a Python function extract_iocs() that takes a simulated JSON response from an LLM API. The function should parse the JSON, access the response text, and extract all IP addresses and CVE numbers mentioned. Return them as a dictionary with two keys: ips and cves, where each key holds a list of the found indicators.

This question tests your ability to parse **JSON**, use **regular expressions** for pattern matching, and structure extracted data.

```python
import re

def extract_iocs(api_response_json):
    """
    Parses a JSON response from an LLM and extracts IP addresses and CVE numbers.
    """
    # Your code here
    # Hint: The response is a JSON string. You'll need to parse it first.
    # Regular expressions are great for finding patterns like IPs and CVEs.
    # IP pattern: \d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}
    # CVE pattern: CVE-\d{4}-\d{4,7}
    pass

# Example usage:
llm_response = """
{
  "model": "threat-intel-v2",
  "response_id": "resp_98765",
  "content": "The 'Flicker-Fade' attack is associated with CVE-2025-12345. Analysis shows
command and control traffic originating from the IP address 198.51.100.24. Another related
vulnerability is CVE-2025-67890, and we've also seen beaconing to 203.0.113.10."
}
"""

iocs = extract_iocs(llm_response)
print(iocs)

# Expected Output:
# {'ips': ['198.51.100.24', '203.0.113.10'], 'cves': ['CVE-2025-12345', 'CVE-2025-67890']}
```

9.2c Automated IP Risk Assessment with an LLM

A common security workflow involves taking a single Indicator of Compromise (IOC), like an IP address, and using an external tool or service—in this case, an LLM—to gather more information and assess its risk level.

Write a Python function is_ip_high_risk() that automates this entire process. The function should:

1. Read a single IP address from a given filename.
2. Create a specific prompt to ask an LLM about the risk level of that IP.
3. Simulate a POST request to an LLM API with the prompt.
4. Parse the JSON response from the LLM.
5. Return True if the LLM classifies the IP's risk as "high," and False otherwise.

```python
import requests
import json

# --- Setup for Simulation (You do not need to modify this) ---

# Create a dummy file containing the IP address
with open("ip_to_check.txt", "w") as f:
    f.write("185.220.101.38\n")

# Mock server class to simulate the API response
class MockResponse:
    def __init__(self, json_data, status_code):
        self.json_data = json_data
        self.status_code = status_code

    def json(self):
        return self.json_data

# This mock function simulates the API's logic
def mock_api_call(url, json):
    prompt_text = json.get("prompt", "")
    # The simulated API returns 'high' risk for this specific IP
    if "185.220.101.38" in prompt_text:
        response_data = {"ip": "185.220.101.38", "risk_level": "high", "reason":
                                                        "Associated with C2 servers."}
    else:
        response_data = {"ip": "unknown", "risk_level": "low", "reason":
                                                        "No threat data found."}
    return MockResponse(response_data, 200)

# Replace the real requests.post with our mock for this exercise
requests.post = mock_api_call

# --- Your Function to Implement ---


def is_ip_high_risk(filename, api_url):
    """
    Reads an IP from a file, queries an LLM API, and checks if the risk is high.
    """
    # 1. Read the IP address from the file. Remember to strip whitespace.
    # 2. Create a prompt for the LLM.
    # 3. Prepare the JSON payload for the API request.
    # 4. Make the POST request and get the JSON response.
    # 5. Check the 'risk_level' in the response and return True or False.
    pass

# --- Example Usage ---

api_endpoint = "https://api.threat-intel-llm.com/v1/enrich"
is_high_risk = is_ip_high_risk("ip_to_check.txt", api_endpoint)
print(f"Is the IP address a high-risk threat? {is_high_risk}")

# --- Expected Output ---
# Is the IP address a high-risk threat? True
```

Module 10: Automation & Integration of Virus and vulnerability scanning

Learning Objectives:

- Understand how to script vulnerability scans using popular tools and Python.
- Automate malware detection using VirusTotal API.
- Use Natural Language Processing and AI tools to extract and summarize vulnerability reports from platforms like Shodan.
- Generate automated alerts and formatted reports.

10.1 Scripting for Vulnerability Scanning, Reporting, and Monitoring

Topics Covered:

- Common vulnerabilities and exposures (CVEs) overview.
- Using Python with:
  - **Nmap** (network discovery and auditing).
  - **OpenVAS** or **Nikto** for scanning.
- Logging scan results and parsing outputs (XML/JSON/HTML).
- Creating continuous monitoring scripts (cron jobs or Task Scheduler).
- Email/Slack reporting or dashboard integration.

Key Tools & Libraries:

- python-nmap or libnmap
- os, subprocess, xml.etree.ElementTree
- smtplib, email, schedule

Example Project:

Task: Build a Python script that:

1. Uses Nmap to scan for open ports and services.
2. Parses the output for vulnerable services.

3. Sends an email with a vulnerability report.

```python
import nmap
import smtplib
from email.mime.text import MIMEText

def scan(host):
    nm = nmap.PortScanner()
    nm.scan(host, '20-1000')
    result = ""
    for host in nm.all_hosts():
        result += f"Host: {host}\n"
        for proto in nm[host].all_protocols():
            for port in nm[host][proto].keys():
                state = nm[host][proto][port]['state']
                result += f"  {proto.upper()} Port {port}: {state}\n"
    return result

# send result via email
```

10.2 Automating Malware Detection using Python & VirusTotal API

Topics Covered:

- VirusTotal API overview and account setup.
- Hash-based vs file-based malware scanning.
- Submitting files and URLs.
- Interpreting VirusTotal JSON responses.
- Logging and alerting malicious indicators.

Key Tools & Libraries:

- requests
- virustotal-python (optional wrapper)
- hashlib, json

Example Project:

Task: Write a Python script to:

1. Calculate the hash of a file.

2. Send it to VirusTotal.
3. Parse and display if the file is flagged malicious.

```python
import requests, hashlib

API_KEY = 'YOUR_API_KEY'
file_path = 'suspicious.exe'

def get_file_hash(file_path):
    with open(file_path, 'rb') as f:
        return hashlib.sha256(f.read()).hexdigest()

def check_virustotal(hash_value):
    url = f'https://www.virustotal.com/api/v3/files/{hash_value}'
    headers = {'x-apikey': API_KEY}
    response = requests.get(url, headers=headers)
    if response.status_code == 200:
        data = response.json()
        return data['data']['attributes']['last_analysis_stats']
    return None

hash_value = get_file_hash(file_path)
result = check_virustotal(hash_value)
print(result)
```

10.3 Automating a Vulnerability Scan and Report via NLP, AI & Shodan

Topics Covered:

- Introduction to **Shodan** as a search engine for internet-connected devices.
- Automating data gathering from Shodan using its API.
- Parsing JSON output to find open ports, device types, banners.
- Using **spaCy** or **transformers** to extract and summarize key threat information.
- Generating natural-language summaries and alerts.

Key Tools & Libraries:

- shodan (Python wrapper for Shodan API)
- spacy, transformers, nltk

- pandas, json

Example Project:

Task: Build a script to:

1. Search Shodan for exposed devices in a specific country with port 22 open.
2. Use NLP to summarize vulnerability banners.

```python
import shodan
import spacy

SHODAN_API_KEY = "YOUR_API_KEY"
nlp = spacy.load("en_core_web_sm")

def search_shodan():
    api = shodan.Shodan(SHODAN_API_KEY)
    results = api.search('port:22 country:"US"')
    for result in results['matches'][:5]:
        ip = result['ip_str']
        banner = result.get('data', 'No data')
        summary = summarize_banner(banner)
        print(f"[{ip}] - {summary}")

def summarize_banner(text):
    doc = nlp(text)
    return ' '.join([sent.text for sent in doc.sents][:2])

search_shodan()
```

Module 10: Learning Materials & References:

i. AI in DevSecOps: Automating Security Vulnerability Detection
ii. Applying Generative AI for CVE Analysis at an Enterprise Scale
iii. How to Use Shodan!
iv. Top 5 Vulnerability Scanning Tools in 2025

Assessment: Module 10: Automation & Integration of Virus and vulnerability scanning

Part A: Quiz

10.1a What is a key benefit of integrating vulnerability scanning tools with SIEM (Security Information and Event Management) systems?

A) Increases manual workload for security teams
B) Enables real-time monitoring and faster incident response
C) Prevents all zero-day attacks automatically
D) Eliminates the need for remediation steps

10.2a Which of the following is considered a best practice when automating vulnerability management?

A) Rely solely on automated tools without human oversight
B) Regularly review and update scanning policies and risk priorities
C) Ignore low-risk vulnerabilities to save time
D) Use outdated vulnerability databases to avoid false positives

10.3a Why is API-based virus scanning integration valuable for enterprise environments?

A) It limits scanning to only on-premises data
B) It allows seamless integration with custom applications and real-time reporting
C) It replaces the need for multi-layered security
D) It removes the need for machine learning in threat detection

10.4a What is the primary benefit of using NLP in vulnerability report generation?

A) Encrypting scan data for secure transmission
B) Automatically patching vulnerabilities in real time
C) Translating technical findings into human-readable summaries
D) Scanning for open ports on remote systems

C) Translating technical findings into human-readable summaries

10.5a Which of the following best describes Shodan's role in automated vulnerability scanning?

A) It patches vulnerabilities found in web applications
B) It scans networks for exposed services and ports

C) It performs static code analysis for software bugs
D) It generates AI-powered remediation recommendations

10.6a How can AI improve vulnerability prioritization in automated reports?

A) By randomly selecting vulnerabilities to address
B) By correlating vulnerabilities with threat intelligence and exploitability
C) By disabling affected systems automatically
D) By replacing all manual security tools

10.7a Which of the following is a key feature of integrating AI with vulnerability scanners?

A) Real-time firewall configuration
B) Automated generation of compliance certificates
C) Context-aware remediation suggestions
D) Manual report writing assistance only

Part C: Python Exercises

10.1.c Modify the Nmap scanner to also check OS versions.

Write a Python script detect_os.py that modifies a basic Nmap scanner to also check and display the **operating system (OS) versions** of target hosts. This exercise uses the `python-nmap` library and leverages Nmap's `-O` flag for OS detection. Note: Running OS detection requires root/administrator privileges

Instructions:

- Install the `python-nmap` library if you haven't already (`pip install python-nmap`).
- Ensure you have Nmap installed on your system.
- Run the script with appropriate privileges (e.g., using `sudo` on Linux).

Scanning scanme.nmap.org for open ports and OS information...

Sample output:

```
Host: 45.33.32.156
State: up
Protocol: tcp
Port: 22        State: open
Port: 80        State: open
Port: 9929      State: open
Port: 31337     State: open
```

```
OS Matches:
 Name: Linux 4.19 - 5.15
 Accuracy: 98%
  Type: general purpose
  Vendor: Linux
  OS Family: Linux
  OS Generation: 4.X
  Accuracy: 98%
```

10.2c Automate Weekly VirusTotal Scans on Downloads Folder

Write a Python script weekly_vt_scan.py to automate weekly VirusTotal scans on files in the Downloads folder.

This exercise uses the VirusTotal Public API v3 and assumes you have a valid API key.This exercise uses the VirusTotal Public API v3 and assumes you have a valid API key.

Goal:

1. Monitor the user's Downloads folder.
2. Automatically upload new files to VirusTotal for scanning.
3. Retrieve and log the scan results.
4. Schedule the script to run weekly (using cron, Task Scheduler, or Python's schedule module).

Requirements:

a) Python 3
b) VirusTotal API key (free or paid)
c) Install required libraries:
d) pip install requests schedule

10.3c Use Hugging Face Transformers to Generate Executive Summaries of Scan Reports

Write a Python script scan_summary_generator.py that uses Hugging Face Transformers to generate executive summaries of vulnerability or virus scan reports. This is useful for converting technical scan data into concise, human-readable summaries for decision-makers.
Goal:

a) Load a sample vulnerability or virus scan report (in plain text or JSON).

b) Use a pre-trained transformer model (e.g., `facebook/bart-large-cnn` or `google/pegasus-xsum`) to summarize the content.
c) Output a short executive summary suitable for non-technical stakeholders.

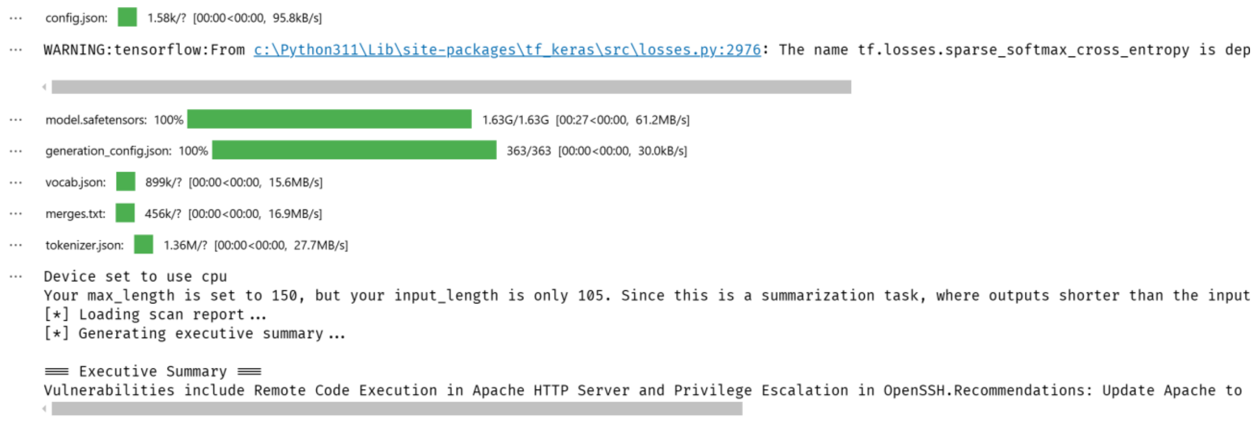
```python
#Sample Python Script: scan_summary_generator.py
from transformers import pipeline
import json
import os

# === Load a Pre-trained Summarization Model ===
summarizer = pipeline("summarization", model="facebook/bart-large-cnn")
```

Requirements:

```
Install the required libraries:
pip install transformers torch
Example Input (example_scan_report.txt):
Scan Report - July 2025
Host: 192.168.1.10
Open Ports: 22 (SSH), 80 (HTTP)
Vulnerabilities:
- CVE-2023-1234: Remote Code Execution in Apache HTTP Server
- CVE-2024-5678: Privilege Escalation in OpenSSH
Recommendations:
- Update Apache to version 2.4.57
- Apply latest OpenSSH patch
- Restrict SSH access to internal IPs only
```

Example Output:



```
config.json: 🟩 1.58k/? [00:00<00:00, 95.8kB/s]
WARNING:tensorflow:From c:\Python311\Lib\site-packages\tf_keras\src\losses.py:2976: The name tf.losses.sparse_softmax_cross_entropy is dep
model.safetensors: 100% 🟩 1.63G/1.63G [00:27<00:00, 61.2MB/s]
generation_config.json: 100% 🟩 363/363 [00:00<00:00, 30.0kB/s]
vocab.json: 🟩 899k/? [00:00<00:00, 15.6MB/s]
merges.txt: 🟩 456k/? [00:00<00:00, 16.9MB/s]
tokenizer.json: 🟩 1.36M/? [00:00<00:00, 27.7MB/s]
Device set to use cpu
Your max_length is set to 150, but your input_length is only 105. Since this is a summarization task, where outputs shorter than the input
[*] Loading scan report...
[*] Generating executive summary...

== Executive Summary ==
Vulnerabilities include Remote Code Execution in Apache HTTP Server and Privilege Escalation in OpenSSH.Recommendations: Update Apache to
```

Your max_length is set to 150, but your input_length is only 105. Since this is a summarization task, where outputs shorter than the input are typically wanted, you might consider decreasing max_length manually, e.g. summarizer('...', max_length=52)
[*] Loading scan report...

```
[*] Generating executive summary...

=== Executive Summary ===
Vulnerabilities include Remote Code Execution in Apache HTTP Server and Privilege
Escalation in OpenSSH.Recommendations: Update Apache to version 2.4.57 and apply
latest OpenSSh patch. Restrict SSH access to internal IPs only.
```

Module 11: Final Project Development

The final project is designed to guide students through the process of building, refining, and presenting a Python-based security tool or automation project using AI:

Lesson 11.1: Project Work Session – Python-Based Security Tool with AI

Students will apply their knowledge to develop a functional security tool or automation project in Python, integrating AI or machine learning components where appropriate.

Activities:

- Brainstorm and propose project ideas (e.g., automated vulnerability scanner, VirusTotal integration, Shodan-based exposure analysis, AI-powered report summarizer)
- Define project scope, goals, and deliverables.
- Begin coding: set up version control, create a project structure, and implement core features.
- Encourage documentation of code and decision-making process.

Deliverables:

- A working prototype or minimum viable product (MVP).
- A brief project plan outlining features and AI/automation components.

Lesson 11.2: Instructor Feedback and Troubleshooting

Students receive targeted feedback, resolve technical challenges, and refine their projects.

Activities:

- Instructors and teaching assistants hold code review sessions and provide feedback on architecture, code quality, and security best practices.
- Troubleshooting workshops: address common issues in Python automation, API integration (e.g., Nmap, VirusTotal, Shodan), and AI model usage.
- Peer support: students collaborate to debug and optimize each other's code.
- Emphasis on error handling, logging, and user-friendly design.

Deliverables:

- Improved project codebase with resolved issues and incorporated feedback.
- Updated documentation reflecting changes and lessons learned.

Lesson 11.3: Project Presentations and Peer Review

Students present their completed projects, demonstrate functionality, and engage in constructive peer review.

Activities:

- Each student (or team) delivers a short presentation covering:
  - Project objectives and motivation
  - Technical approach and Python/AI integration
  - Live demonstration of the tool or automation script
  - Key challenges and how they were overcome
- Peer review: classmates provide feedback on usability, innovation, and potential improvements.
- Group discussion on lessons learned, future enhancements, and real-world applications.

Deliverables:

- Final project submission (code, documentation, and presentation slides).
- Peer review forms or feedback summaries.

Assessment Criteria:

- Functionality and reliability of the tool
- Effective use of Python and AI/automation techniques
- Code quality and documentation
- Creativity and relevance to cybersecurity
- Quality of presentation and peer feedback

This structure ensures students gain hands-on experience in developing, refining, and presenting real-world security automation tools, while also benefiting from instructor guidance and peer collaboration.

Example: Capstone Project: Cyber Threat Auto-Reporter

Objective: Combine the techniques learned in previous modules to build an automation pipeline that:

- Scans a network for open services and vulnerabilities (Nmap).
- Checks files against VirusTotal for malware.
- Searchs Shodan for public exposure.
- Uses NLP to generate a PDF/HTML report and email it to a security team.

Appendix A: Python's Best Practices

Why is adherence to Python best practices essential for software development?

1. Adhering to Python best practices, such as those outlined in PEP 8 and specified in this course, is crucial for developing robust, readable, and maintainable code. Consistent compliance promotes code clarity and understanding, and noncompliance in assignments will result in a significant grade deduction (15%).
2. Documentation with docstring, using multi-line docstring is recommended for documenting Python code. This comprehensive documentation enhances code clarity and understanding.
3. Gradual Typing, this practice allows parts of your program to be statically typed while others remain dynamically typed
4. Class Variable Naming Conventions, following conventions outlined in PEP 8 is important for documenting and naming class variables
5. Enforcing OOP Encapsulation by using setters and getters with Python decorators (@property and @_attribute.setter) to enforce Object-Oriented Programming (OOP) encapsulation. This practice helps manage access visibilities to class attributes.
6. Every class should have a default __str__ method, this method provides a string representation of the object when using str() or print() on an instance, which is useful for debugging and displaying information about the object

1.0 How should class variables be named according to best practices?

According to PEP 8 conventions:

- Class variables should use lower_case_with_underscores (snake_case).
- Constant values shared by all instances should use ALL_CAPS_WITH_UNDERSCORES.
- Private variables intended for internal use should be prefixed with a single underscore (_variable_name).
- Avoid names that shadow built-in functions or types.
- Use docstring or comments to explain the purpose of variables, especially public ones.

Aspect	Convention
Naming Style	Use `lower_case_with_underscores` (snake_case) for class variables.
Constant Values	Use `ALL_CAPS_WITH_UNDERSCORES` for constants shared by all instances.
Private Variables	Prefix with a single underscore: `_variable_name`.
Avoid Confusion	Don’t use names that shadow built-ins (e.g., `list`, `str`, `id`, etc.).
Documenting	Use docstrings or comments to explain purpose, especially for public variables.

1.1 Class Naming Example:

```
from typing import Final
class Car(object):
    """
    A class representing a car.
    Attributes:
        year (int): The year of the car defaults to 2000.
        price (float): The price of the car.
        number_of_cylinders (int): Class variable shared by all instances.
        MAX_SPEED_MPH (int): Constant representing the maximum speed in mph.
        _ford_manufacturer_code (str): Private class variable for internal use only.
    """
    number_of_cylinders: int = 4  # Class variable (shared by all instances)
    MAX_SPEED_MPH: Final = 100  # Constant (conventionally treated as immutable)
    _ford_manufacturer_code: str = '1FT'  # Private class variable (internal use
only)

    def __init__(self, price: float, year: int = 2000) -> None:
        self.year: int = year  # Instance variable default 2000
        self.price: float = price  # Instance variable
```

1.2 Naming helper functions

When using recursion in a class, it's often helpful to create a private helper function. By convention in Python, the names of these internal helper functions are prefixed with a leading underscore (‘_’).

This signals that the helper function isn't intended to be called directly from outside the class. The public method serves as a clean starting point, while the helper method handles the complex recursive logic.

For example, a public method `check_bst()` would call its recursive helper, `_check_bst()`, to perform the actual validation.

2.0 What is the recommended way to document Python code?

Using multi-line docstring is the recommended method for documenting Python code. This comprehensive documentation enhances clarity and understanding for both the original author and other developers who might read the code. Tools like pdoc or pydoc can be used to generate documentation from these docstrings, with pdoc being preferred for this class.

2.1 Document Python Code with Multiline String Example

```
def div(top: int, bottom: int = 1) -> int:

    """
    integer division or floor division will return an int,
    the default value of divisor is 1 if num2 is not provided

    Parameters:
        top (int): The dividend, an integer.
        bottom (int): The divisor, an integer, default is 1.
    Raises:
        TypeError: If bottom is not an integer.
        ZeroDivisionError: If bottom is zero.

    Returns:
        int: The result of the division.
    """

    if not isinstance(bottom, int):
        raise TypeError(Divisor must be an integer')
    if bottom == 0:
        raise ZeroDivisionError('Divisor must not be zero')
    if top == 0:
        return 0

    return (top // bottom)  # integer division or floor division
```

- You can use either pydoc or pdoc, but pdoc is preferable.
- More info on pydoc https://pydoctor.readthedocs.io/en/latest/codedoc.html

- More info on pdoc https://pdoc.dev/docs/pdoc.html#what-is-pdoc

3.0 What is gradual typing in Python and why is it beneficial?

Gradual type checker is a static type of checker that checks for type errors in statically typed part of a gradually typed program. Static checkers will find bugs sooner in the statically typed part of the program. In the larger project, it becomes more difficult to debug a runtime type error. It makes the program easier to understand for a new engineer in the team as the flow of objects is hard to follow in Python

3.1 How can gradual typing be applied in Python code?

Gradual typing is applied using type hints or annotations. This involves specifying the expected types for function parameters and return values (def func(param: type) -> return_type:), as well as for variables and data structures (variable: type = value, list[type], dict[key_type, value_type], etc.). The typing module provides support for various types and concepts like Final for constants.

More info on gradual typing in python
https://typing.python.org/en/latest/spec/concepts.html

3.11 Declare Constants in Python

```
"""
Gradual Typing in Python allows parts of a program to be
dynamically typed and other parts to be statically typed, setting
the arguments type and
return type
"""
from typing import Final


PI: Final = 3.14159

class MyWebSite:
  VERSION: Final = "1.0"

  def __init__(self):
    self.HOST: Final = "localhost"
```

3.12 Functions with gradual typing

```
def say_hello(name: str) -> str:

    """
    say_hello will return a string greeting the user with their
name.
    Parameters:
        name (str): The name of the user, a string.
    Returns:
        str: A greeting message.
    """
    return f'Hello {name}'


def display_hello(name: str) -> None:
    """
    display_hello will print a greeting message to the console.
    Parameters:
        name (str): The name of the user, a string.
    Returns:
        None: This function does not return anything.
    """
    print('Hello ' + name)
```

3.13 Variables & Data Structures with Gradual Typing

```
def variables() -> None:
    # variables with type hinting/annotations
    number: int = 10
    decimal: float = 2.5
    text: str = 'Hello, World!'
    active: bool = True

    # Python data structures with type hinting/annotations

    numbers: list[int] = [11, 7, 13, 19]
    names: list[str] = ['Anton', 'Will', 'Maria']
    locations: tuple[float, float] = (10.5, 20.3)
    age: dict[str, int] = {'Anton': 25, 'Will': 30, 'Maria': 22}
    nonrepeat_numbers: set[int] = {1, 2, 3, 4, 5}
```

```python
    # Constants with type hinting/annotations
    VERSION: Final[str] = '1.0.0'
    PI: Final[float] = 3.14159

def main():
    """main function to test the above functions"""
    # Test the functions
    print(say_hello("CSIS-252"))
    display_hello('CSIS-252')

    print(div(2, 3))
    # will not throw an error

    # will raise a TypeError
    print(div(1, 'Div Error'))
```

3.14 Why is it important for every class to have a default __str__ method?

Every class should have a default __str__ method because it provides a readable string representation of an object instance. This method is automatically called when you use str() or print() on an instance of the class. It is invaluable for debugging and for easily displaying information about the object's state.

3.15 Why is the dunder main in Python considered the best practice?

Because it controls the execution context of a script, making your code more modular, reusable, and maintainable.

3.15.1 What It Does

Python assigns the special variable __name__ to each module. It behaves differently depending on how the file is run:

- If the file is run directly, __name__ == "__main__" will be True.
- If the file is imported as a module, __name__ will be set to the module's name, not "__main__".

Benefit	Explanation
Prevents unintended execution	Code under if __name__ == "__main__": won't run if the script is imported elsewhere.
Encourages modular design	You can define functions/classes for reuse in other modules without running the main logic automatically.
Easier to test	Unit tests can import your module without triggering its runtime logic.
Improves readability	Clearly separates the “definition" code from "execution" code.
Flexible execution	Allows a script to be used both as a reusable module and as a standalone program.

Example

```
if __name__ == '__main__':
    main()
```

4.0 How is OOP encapsulation enforced in Python using best practices?

Object-oriented-Programming or OOP encapsulation can be enforced in Python by using setters and getters in conjunction with decorators like @property and @_attribute.setter. This approach helps manage access visibility to class attributes, controlling how internal data is accessed and modified, even though Python's attribute access is not strictly private like in some other languages.

4.1 Getters, Setters with @property Example

```
"""
Car class with a constructor that initializes the car's make,
model, and year. It also includes a __str__ method to provide a
string representation of the car object.
with Decorators and Properties and annotations
"""


class Car(object):
    """
    A class representing a car.
    Attributes:
        _make (str): private The make of the car.
        _model (str): private The model of the car.
        _year (int): private The year of the car.
        use getter with @property decorator
        use setter with @_attribute.setter decorator

    """

    def __init__(self, make: str = "Toyota", model: str = "Corolla",
                                  year: int = 2020) -> None:
        """
        non-default constructor
        Initialize a Car object.
        """
        self._make = make
        self._model = model
        self._year = year



    @property
    def make(self) -> str:
        """
        Get the make of the car.
        """
        return self._make

    @make.setter
    def make(self, make: str) -> None:
        """
        Set the make of the car.
        """
        self._make = make

    @property
    def model(self) -> str:
        """
        Get the model of the car.
        """
        return self._model

    @model.setter
    def model(self, model: str) -> None:
        """
        Set the model of the car.
        """
        self.model = model
```

4.1 Why is it important for every class to have a default __str__ method?

Every class should have a default __str__ method because it provides a readable string representation of an object instance. This method is automatically called when you use str() or print() on an instance of the class. It is invaluable for debugging and for easily displaying information about the object's state.

4.1.1 Dunder String Method Example

```python
    def __str__(self) -> str:
        """
        Return a string representation of the car.
        """
        # This method is called when you use str(car) or print(car)
        # It returns a string that describes the car.
        return f"{self._year:<d} {self._make:<6} {self._model:<}"
```