INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
Dundigal - 500 043, Hyderabad, Telangana

COURSE CONTENT

CYBER CRIME INVESTIGATION & DIGITAL FORENSICS LABORATORY								
VII Semester: CSE (CS)								
Course Code	**Category**	**Hours / Week**			**Credits**	**Maximum Marks**		
		L	**T**	**P**	**C**	**CIA**	**SEE**	**Total**
ACCD25	Core	0	0	2	1	40	60	100
Contact Classes: Nil	**Tutorial Classes: Nil**	**Practical Classes: 45**				**Total Classes: 45**		
Prerequisites: Nil.								

I. COURSE OVERVIEW:

This laboratory course provides practical experience in cybercrime investigation techniques, digital forensics methodologies, and the use of forensic tools for analyzing digital evidence. Students will explore cyber threats, forensic data acquisition, evidence preservation, and analysis of compromised systems while adhering to legal and ethical guidelines.

II. COURSE OBJECTIVES:

The students will try to learn:

I. Fundamental principles of cybercrime investigation and digital forensics.
II. Techniques for forensic data acquisition, analysis, and preservation
III. Hands-on experience with forensic tools such as Autopsy, FTK, En Case, and Wireshark.
IV. Investigation of cybercrime cases involving malware, fraud, identity theft, and hacking.
V. Legal and ethical considerations in digital forensics and cyber law compliance.

III. COURSE OUTCOMES:

CO1 Understand the methodologies and tools used in cybercrime investigation and digital forensics.
CO2 Perform forensic acquisition and preservation of digital evidence.
CO3 Analyze forensic artifacts from computer systems, mobile devices, and networks.
CO4 Detect and investigate various types of cybercrimes using forensic techniques.
CO5 Apply legal and ethical principles in forensic investigations and evidence handling.
CO6 Analyze and investigate cybercrime incidents by applying digital forensic tools and techniques to collect, preserve, and interpret digital evidence.

IV. COURSE CONTENT:

EXERCISES FOR CYBER CRIME INVESTIGATION & DIGITAL FORENSICS LABORATORY

Note: Students are encouraged to bring their own laptops for laboratory practice sessions. All exercises are for educational purposes only and use synthetic, non-sensitive sample data; no exercise involves accessing systems or data without authorization.

1. Getting Started Exercises

1.1 Caesar Cipher Decoder

During triage of a suspect's device, investigators frequently recover short text notes that have been obfuscated with a simple substitution cipher. Write a program that decodes a message encrypted with a Caesar cipher, given the shift value recovered separately (e.g. from a suspect's notebook).

A Caesar cipher shifts every letter forward by a fixed number of positions in the alphabet, wrapping around from 'z' back to 'a'. Non-alphabetic characters (spaces, punctuation, digits) are left unchanged, and the case of each letter is preserved.

Example: if the encrypted note is "khoor zruog" and the shift is 3, the decoded message is "hello world".

Input: message = "khoor zruog", shift = 3
Output: "hello world"

Input: message = "Uifsf't b tfdsfu", shift = 1
Output: "There's a secret"

```
public class CaesarDecoder
{
    public static String decode(String message, int shift)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

1.2 File Hash Calculator (Evidence Integrity Check)

Preserving the integrity of digital evidence is a core principle of forensic acquisition. Investigators compute a cryptographic hash of a file at the time of seizure, and re-compute it later to prove the file was not altered.

Write a function that accepts the contents of a file as a byte array and returns its SHA-256 hash as a lowercase hexadecimal string.

Note: use a standard library hashing implementation (e.g. java.security.MessageDigest) rather than writing your own hash function.

Input: data = bytes of "hello"
Output: "2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824"

Input: data = bytes of "" (empty file)
Output: "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855"

```
import java.security.MessageDigest;

public class FileHashCalculator
{
    public static String computeSHA256(byte[] data)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

1.3 File Signature (Magic Number) Identifier

Attackers and suspects often rename files to disguise their true type (e.g. renaming an image to "report.docx"). Forensic examiners instead identify file type by inspecting the first few bytes of the file — its 'magic number' — rather than trusting the extension.

Write a function that accepts the first four bytes of a file (as unsigned integers 0-255) and returns the file type, based on the signatures below.

Recognise: PNG (89 50 4E 47), JPEG (FF D8 FF E0), PDF (25 50 44 46), ZIP/DOCX/XLSX (50 4B 03 04). If the signature does not match any of these, return "UNKNOWN".

Input: bytes = [137, 80, 78, 71]
Output: "PNG"

Input: bytes = [37, 80, 68, 70]
Output: "PDF"

Input: bytes = [1, 2, 3, 4]
Output: "UNKNOWN"

```
public class FileSignatureIdentifier
{
    public static String identify(int[] header)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

1.4 IPv4 Address Validator

Network logs seized during an investigation often contain malformed or spoofed entries. Before an IP address is used as evidence, it must be validated as a properly formed IPv4 address.

Write a function that accepts a string and returns true if it is a valid IPv4 address: exactly four dot-separated numbers, each between 0 and 255 inclusive, with no leading zeros (except the number "0" itself) and no extra characters.

Input: "192.168.1.1"
Output: true

Input: "256.1.1.1"
Output: false

Input: "192.168.01.1"
Output: false

Input: "192.168.1"
Output: false

```
public class IPv4Validator
{
    public static boolean isValid(String address)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

1.5 Timeline Builder: Unix Timestamp Converter

Building an accurate event timeline is central to a forensic investigation. File-system metadata and log files usually record timestamps as a Unix epoch value (seconds since 1 January 1970, 00:00:00 UTC), which must be converted to a human-readable date-time for a report.

Write a function that accepts a Unix epoch timestamp (long) and returns the corresponding UTC date-time as a string in the format "yyyy-MM-dd HH:mm:ss".

Input: 0
Output: "1970-01-01 00:00:00"

Input: 1700000000
Output: "2023-11-14 22:13:20"

```
import java.time.Instant;
import java.time.ZoneOffset;
import java.time.format.DateTimeFormatter;

public class TimestampConverter
{
    public static String toReadableDate(long epochSeconds)
    {
        # Write code here
    }
    public static void main(String[] args)
```

```
    {
        # Write code here
    }
}
```

1.6 XOR Cipher Encoder / Decoder

A single-byte XOR cipher is a common, weak obfuscation technique found in malware configuration files and hidden data. Because XOR is its own inverse, the same function both encodes and decodes a message when given the correct key byte.

Write a function that accepts a byte array and a single key byte, and returns a new byte array where every byte has been XOR-ed with the key.

Input: data = bytes of "HI", key = 5
Output: bytes = [77, 76]

Input: data = [77, 76], key = 5
Output: bytes of "HI"

```
public class XorCipher
{
    public static byte[] apply(byte[] data, byte key)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

1.7 Password Strength Analyzer

As part of an incident report, examiners often need to summarise how weak the credentials involved in a breach were. Write a function that scores a password's strength based on the rules below and returns one of: "WEAK", "MODERATE", or "STRONG".

Award one point each for: length >= 8, contains an uppercase letter, contains a lowercase letter, contains a digit, contains a special character (any of !@#$%^&*). 0-2 points = WEAK, 3-4 points = MODERATE, 5 points = STRONG.

Input: "password"
Output: "WEAK"

Input: "Password1"
Output: "MODERATE"

Input: "P@ssw0rd!"
Output: "STRONG"

```
public class PasswordStrengthAnalyzer
{
```

```
    public static String analyze(String password)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

1.8 Base64 Evidence Decoder

Email attachments and embedded artifacts are frequently stored using Base64 encoding. Write a function that decodes a Base64-encoded string recovered from a suspect email header back into its original plain-text form.

Note: use a standard library decoder (e.g. java.util.Base64) rather than implementing the encoding table by hand.

Input: "U2VjcmV0IE1lc3NhZ2U="
Output: "Secret Message"

Input: "RXZpZGVuY2U="
Output: "Evidence"

```
import java.util.Base64;

public class Base64EvidenceDecoder
{
    public static String decode(String encoded)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

1.9 Failed Login Counter (Authentication Log Analysis)

Server authentication logs are a key artifact when investigating a suspected brute-force attack. Each log line has the format "<timestamp> <username> <status>", where status is either SUCCESS or FAILURE.

Write a function that accepts an array of log lines and a username, and returns the number of consecutive FAILURE entries for that username immediately before the most recent SUCCESS (or before the end of the log, if there is no success). This highlights how many attempts preceded a successful — potentially compromised — login.

Input: logs = ["09:00 alice FAILURE", "09:01 alice FAILURE", "09:02 alice SUCCESS"], username = "alice"
Output: 2

Input: logs = ["09:00 bob FAILURE", "09:01 bob FAILURE", "09:02 bob FAILURE"], username = "bob"

Output: 3

```
public class FailedLoginCounter
{
    public static int countPrecedingFailures(String[] logs, String username)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

1.10 MAC Address Format Normalizer

Network device identifiers (MAC addresses) appear in router logs and DHCP leases in inconsistent formats depending on the vendor — some use colons, some hyphens, some no separator at all. Before comparing addresses across evidence sources, they must be normalised.

Write a function that accepts a MAC address in any of these forms — "AA:BB:CC:DD:EE:FF", "AA-BB-CC-DD-EE-FF", or "AABBCCDDEEFF" — and returns it normalised to lowercase, colon-separated form.

Input: "AA:BB:CC:DD:EE:FF"
Output: "aa:bb:cc:dd:ee:ff"

Input: "AA-BB-CC-DD-EE-FF"
Output: "aa:bb:cc:dd:ee:ff"

Input: "AABBCCDDEEFF"
Output: "aa:bb:cc:dd:ee:ff"

```
public class MacAddressNormalizer
{
    public static String normalize(String mac)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

Try:

1. Luhn Checksum Validator: Suspect payment card numbers recovered from a device image must be validated before being logged as evidence. The Luhn algorithm is the standard checksum used by most card issuers. Your task is to complete the function isValidCardNumber() which takes a card number as a string and returns true if it passes the Luhn check, false otherwise.

Input: cardNumber = "4539578763621486"
Output: true

Input: cardNumber = "1234567890123456"
Output: false

2. Fuzzy Filename Matcher: When searching a disk image for files related to a keyword (e.g. "invoice"), exact matches often miss relevant files due to typos or partial names introduced during file recovery. Your task is to complete the function levenshteinDistance() which takes two strings, a filename and a search term, and returns the edit distance between them — the minimum number of single-character insertions, deletions or substitutions needed to turn one into the other.

Input: filename = "invoice_2024.pdf", term = "invoice"
Output: 10

Input: filename = "invioce.pdf", term = "invoice.pdf"
Output: 2

2. DIGITAL EVIDENCE AND FORENSIC INVESTIGATION PROCESS

2.1 Order of Volatility Ranker

RFC 3227 defines an 'order of volatility' that first responders must follow when collecting digital evidence — the most volatile data (likely to be lost first) must be captured before less volatile data. Write a function that accepts a list of evidence source names and returns them re-ordered from most volatile to least volatile.

Use this volatility ranking (most volatile first): "CPU registers/cache", "Routing table/ARP cache", "RAM", "Running processes", "Temporary file systems", "Disk", "Remote logging data", "Physical configuration/network topology", "Archival media/backups".

Any source not in the reference list should be placed at the end, in the order it originally appeared.

Input: sources = ["Disk", "RAM", "Archival media/backups", "CPU registers/cache"]
Output: ["CPU registers/cache", "RAM", "Disk", "Archival media/backups"]

Input: sources = ["Backups", "RAM"]
Output: ["RAM", "Backups"]

```
import java.util.*;

public class VolatilityRanker
{
    public static List<String> rankByVolatility(List<String> sources)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

2.2 Chain of Custody Entry Validator

A chain-of-custody log records every person who has handled a piece of evidence, and is only

admissible if every entry is complete. Each entry is a string of the form "<evidenceId>|<custodianName>|<timestamp>|<action>", where action is one of COLLECTED, TRANSFERRED, ANALYZED, or RETURNED.

Write a function that accepts a single log entry string and returns true only if all four fields are present and non-empty, and the action is one of the four allowed values.

Input: "EVD-001|Officer Rao|2024-03-01 10:15|COLLECTED"
Output: true

Input: "EVD-001||2024-03-01 10:15|COLLECTED"
Output: false

Input: "EVD-001|Officer Rao|2024-03-01 10:15|DELETED"
Output: false

```
public class CustodyEntryValidator
{
    public static boolean isValidEntry(String entry)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

2.3 Evidence Acquisition Hash Verifier

The forensic soundness of an acquired disk image is proven by comparing the hash taken at the moment of acquisition against the hash re-computed before analysis. If the two hashes differ, the image may have been altered and cannot be trusted as evidence.

Write a function that accepts the acquisition-time hash and the verification-time hash (both hex strings) and returns "VERIFIED" if they match exactly (case-insensitive), or "INTEGRITY FAILURE" if they do not.

Input: acquisitionHash = "3F2A9B", verificationHash = "3f2a9b"
Output: "VERIFIED"

Input: acquisitionHash = "3F2A9B", verificationHash = "3F2A9C"
Output: "INTEGRITY FAILURE"

```
public class HashVerifier
{
    public static String verify(String acquisitionHash, String verificationHash)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

2.4 Evidence Tag ID Generator

Every seized item must be tagged with a unique, well-formed evidence ID before it enters the chain of custody. The required format is "EVD-<caseNumber>-<sequenceNumber>", where caseNumber is a 4-digit zero-padded number and sequenceNumber is a 3-digit zero-padded number.

Write a function that accepts a case number and a sequence number (both integers) and returns the formatted evidence tag ID.

Input: caseNumber = 7, sequenceNumber = 3
Output: "EVD-0007-003"

Input: caseNumber = 152, sequenceNumber = 21
Output: "EVD-0152-021"

```
public class EvidenceTagGenerator
{
    public static String generateTag(int caseNumber, int sequenceNumber)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

2.5 Investigation Phase Sequencer

The standard digital forensic investigation process consists of six phases performed in a fixed order: Identification, Preservation, Collection, Examination, Analysis, and Presentation.

Write a function that accepts a list of phase names in arbitrary order and returns true if they can be arranged into a valid subsequence of the standard order (i.e. each phase in the input appears later than the one before it in the standard order, with no repeats), or false otherwise.

Input: phases = ["Identification", "Collection", "Analysis"]
Output: true

Input: phases = ["Collection", "Identification"]
Output: false

Input: phases = ["Preservation", "Preservation"]
Output: false

```
import java.util.*;

public class PhaseSequencer
{
    public static boolean isValidSequence(List<String> phases)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

2.6 Evidence Bag Manifest Checksum

When multiple exhibits are packed into a single evidence container, an investigator generates one combined checksum for the manifest so that any change to the exhibit list (addition, removal, or reordering) can be detected later.

Write a function that accepts an array of individual exhibit hash strings and returns a single combined checksum, computed as the SHA-256 hash of all the individual hashes concatenated in the order given.

Input: hashes = ["a1b2", "c3d4"]
Output: SHA-256("a1b2c3d4") as a lowercase hex string

Input: hashes = ["a1b2"]
Output: SHA-256("a1b2") as a lowercase hex string

```
import java.security.MessageDigest;

public class ManifestChecksum
{
    public static String computeManifestChecksum(String[] hashes)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

2.7 Custody Transfer Gap Detector

An unexplained gap between one custodian handing off evidence and the next custodian receiving it can be challenged in court as a break in the chain of custody. Write a function that accepts a list of custody transfer timestamps (in chronological order, as epoch minutes) and a maximum allowed gap in minutes, and returns the number of gaps that exceed the allowed threshold.

Input: timestamps = [0, 30, 200, 210], maxGapMinutes = 60
Output: 1

Input: timestamps = [0, 10, 20, 30], maxGapMinutes = 60
Output: 0

```
public class CustodyGapDetector
{
    public static int countExcessiveGaps(int[] timestamps, int maxGapMinutes)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

2.8 Duplicate Evidence Detector

The same file is sometimes seized twice from different sources (e.g. a laptop and a cloud backup). Investigators identify duplicate exhibits by comparing hash values rather than file names, since names are easy to change.

Write a function that accepts a list of exhibit hash values and returns a list of the hashes that appear more than once, in the order they were first repeated.

Input: hashes = ["h1", "h2", "h1", "h3", "h2"]
Output: ["h1", "h2"]

Input: hashes = ["h1", "h2", "h3"]
Output: []

```
import java.util.*;

public class DuplicateEvidenceDetector
{
    public static List<String> findDuplicates(List<String> hashes)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

2.9 Evidence Retention Period Checker

Digital evidence must be retained for a minimum period that depends on the case category, after which it becomes eligible for disposal review. Write a function that accepts the case category and the number of years already elapsed since case closure, and returns "RETAIN" or "ELIGIBLE FOR REVIEW".

Use these minimum retention periods: "Felony" = 10 years, "Misdemeanor" = 5 years, "Civil" = 7 years. If elapsed years is less than the minimum for the category, return "RETAIN"; otherwise return "ELIGIBLE FOR REVIEW".

Input: category = "Misdemeanor", elapsedYears = 6
Output: "ELIGIBLE FOR REVIEW"

Input: category = "Felony", elapsedYears = 6
Output: "RETAIN"

```
public class RetentionPeriodChecker
{
    public static String checkRetention(String category, int elapsedYears)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

2.10 Evidence Access Log Anomaly Finder

Only the custodians named in the chain-of-custody log should ever access an evidence file. An access log line has the format "<accessorName> <timestamp>". Write a function that accepts the list of authorised custodian names and the list of access log lines, and returns the list of accessor names that appear in the access log but are NOT in the authorised custodian list — a sign of unauthorised evidence access.

Input: custodians = ["Officer Rao", "Officer Iyer"], accessLog = ["Officer Rao 10:00", "Officer Mehta 10:05"]
Output: ["Officer Mehta"]

Input: custodians = ["Officer Rao"], accessLog = ["Officer Rao 10:00"]
Output: []

```
import java.util.*;

public class AccessAnomalyFinder
{
    public static List<String> findUnauthorizedAccess(List<String> custodians, List<String>
accessLog)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

Try:

1. Custody Chain Completeness Checker: Given the full sequence of chain-of-custody entries for one evidence item, an investigation is only sound if the action sequence follows a valid pattern — it must start with COLLECTED, and every TRANSFERRED must be followed eventually by another custody action, ending in either ANALYZED or RETURNED. Your task is to complete the function isChainComplete() which takes an array of action strings (in the order they occurred) and returns true if the chain is complete and valid, false otherwise.

Input: actions = ["COLLECTED", "TRANSFERRED", "ANALYZED"]
Output: true

Input: actions = ["TRANSFERRED", "ANALYZED"]
Output: false

2. Evidence Integrity Report Generator: After analysis, investigators compare the hash of every exhibit recorded at acquisition time against its hash at analysis time, to produce a report of which exhibits are intact and which have failed integrity checks. Your task is to complete the function generateIntegrityReport() which takes two parallel arrays — acquisitionHashes[] and analysisHashes[] — of equal length, and returns a list of the indices where the two hashes do not match.

Input: acquisitionHashes = ["a1", "b2", "c3"], analysisHashes = ["a1", "b9", "c3"]
Output: [1]

Input: acquisitionHashes = ["a1", "b2"], analysisHashes = ["a1", "b2"]
Output: []

3. DISK IMAGING AND DATA ACQUISITION

3.1 Sector-to-Byte Converter

Forensic imaging tools work in units of sectors, but investigators need to reason about evidence size in bytes. Write a function that accepts the number of sectors on a disk and the sector size in bytes, and returns the total disk size in bytes.

Also write the inverse: given a total size in bytes and a sector size, return the number of whole sectors it contains (ignore any partial trailing sector).

Input: sectors = 1000000, sectorSize = 512
Output: 512000000

Input: totalBytes = 512000300, sectorSize = 512
Output: 1000000

```
public class SectorByteConverter
{
    public static long sectorsToBytes(long sectors, int sectorSize)
    {
        # Write code here
    }
    public static long bytesToWholeSectors(long totalBytes, int sectorSize)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

3.2 Disk Image Segment Splitter

Legacy file systems and evidence storage rules often require a disk image to be split into fixed-size segments (e.g. 2 GB chunks) rather than one large file. Write a function that accepts the total disk size in bytes and a segment size in bytes, and returns the number of segments needed and the size of the final (possibly partial) segment.

Return the result as two values: total segment count, and the size in bytes of the last segment.

Input: totalBytes = 5000, segmentSize = 2000
Output: segments = 3, lastSegmentSize = 1000

Input: totalBytes = 4000, segmentSize = 2000
Output: segments = 2, lastSegmentSize = 2000

```
public class ImageSegmentSplitter
{
    public static int[] computeSegments(long totalBytes, long segmentSize)
    {
        # Write code here – return new int[] { segmentCount, lastSegmentSize }
    }
    public static void main(String[] args)
```

```
    {
        # Write code here
    }
}
```

3.3 Acquisition Time Estimator

Before starting a full disk acquisition, an examiner needs to estimate how long imaging will take so the process can be scheduled appropriately. Write a function that accepts the disk size in megabytes and the sustained transfer rate in megabytes per second, and returns the estimated acquisition time in whole minutes, rounded up.

Input: diskSizeMB = 500000, transferRateMBps = 200
Output: 42

Input: diskSizeMB = 6000, transferRateMBps = 100
Output: 1

```
public class AcquisitionTimeEstimator
{
    public static int estimateMinutes(long diskSizeMB, long transferRateMBps)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

3.4 Bit-Stream Copy Verifier

A forensic image is only sound if it is a true bit-for-bit copy of the source. Write a function that accepts the source sector count, the destination (image) sector count, the source hash, and the destination hash, and returns "MATCH" only if both the sector counts and the hashes are identical, or "MISMATCH" otherwise.

Input: sourceSectors = 1000, destSectors = 1000, sourceHash = "ab12", destHash = "ab12"
Output: "MATCH"

Input: sourceSectors = 1000, destSectors = 998, sourceHash = "ab12", destHash = "ab12"
Output: "MISMATCH"

```
public class BitStreamVerifier
{
    public static String verifyCopy(long sourceSectors, long destSectors, String sourceHash,
String destHash)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
```

```
        # Write code here
    }
}
```

3.5 Forensic Image Format Identifier

Different acquisition tools produce image files with different extensions, and analysis tools need to know the format before they can open the file. Write a function that accepts a file name and returns the image format based on its extension.

Recognise: ".dd" or ".img" → "RAW", ".e01" → "EnCase (E01)", ".aff" → "Advanced Forensic Format (AFF)", ".vmdk" → "Virtual Disk (VMDK)". The match should be case-insensitive. Any other extension should return "UNKNOWN".

Input: "case101_disk1.E01"
Output: "EnCase (E01)"

Input: "case101_disk1.dd"
Output: "RAW"

Input: "case101_disk1.qcow2"
Output: "UNKNOWN"

```
public class ImageFormatIdentifier
{
    public static String identifyFormat(String fileName)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

3.6 Write-Blocker Compliance Checker

A hardware or software write-blocker must guarantee that no write operation ever reaches the source evidence drive during acquisition. Given a log of operations performed during an imaging session, each formatted as "<operation> <target>" where operation is READ or WRITE, and target is SOURCE or DESTINATION, write a function that returns true only if there is no WRITE operation targeting SOURCE anywhere in the log.

Input: log = ["READ SOURCE", "WRITE DESTINATION", "READ SOURCE"]
Output: true

Input: log = ["READ SOURCE", "WRITE SOURCE"]
Output: false

```
public class WriteBlockerChecker
{
    public static boolean isCompliant(String[] log)
    {
```

```
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

3.7 Disk Geometry Sector Counter

Older disk geometry is described in terms of Cylinders, Heads, and Sectors-per-track (CHS). Before an image can be validated, an examiner sometimes needs to compute the total sector count that a given CHS geometry implies, to cross-check it against the value reported by the imaging tool.

Write a function that accepts the number of cylinders, heads, and sectors-per-track, and returns the total number of addressable sectors (cylinders x heads x sectorsPerTrack).

Input: cylinders = 16383, heads = 16, sectorsPerTrack = 63
Output: 16514064

Input: cylinders = 100, heads = 4, sectorsPerTrack = 32
Output: 12800

```
public class DiskGeometrySectorCounter
{
    public static long totalSectors(int cylinders, int heads, int sectorsPerTrack)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

3.8 Used / Free Space Calculator

When deciding whether to perform a full physical image or a targeted logical acquisition, an examiner first checks how much of a disk is actually in use. Write a function that accepts the total disk size and the used space (both in bytes) and returns the free space in bytes and the percentage of the disk that is free, rounded to one decimal place.

Input: totalBytes = 1000000000, usedBytes = 400000000
Output: freeBytes = 600000000, freePercent = 60.0

Input: totalBytes = 250000000, usedBytes = 250000000
Output: freeBytes = 0, freePercent = 0.0

```
public class DiskSpaceCalculator
{
    public static double[] computeFreeSpace(long totalBytes, long usedBytes)
    {
        # Write code here – return new double[] { freeBytes, freePercent }
    }
```

```
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

3.9 Live vs Dead Acquisition Selector

Choosing between a live acquisition (imaging a running system) and a dead acquisition (imaging powered-off media) depends on the situation. Write a function that accepts two booleans — isSystemPoweredOn and isFullDiskEncrypted — and returns the recommended acquisition method as a string.

Rules: if the system is powered on AND the disk is encrypted, return "LIVE ACQUISITION REQUIRED" (the encryption key would be lost on shutdown). If the system is powered on and not encrypted, return "LIVE ACQUISITION RECOMMENDED". If the system is powered off, return "DEAD ACQUISITION" regardless of encryption.

Input: isSystemPoweredOn = true, isFullDiskEncrypted = true
Output: "LIVE ACQUISITION REQUIRED"

Input: isSystemPoweredOn = true, isFullDiskEncrypted = false
Output: "LIVE ACQUISITION RECOMMENDED"

Input: isSystemPoweredOn = false, isFullDiskEncrypted = true
Output: "DEAD ACQUISITION"

```
public class AcquisitionMethodSelector
{
    public static String recommendMethod(boolean isSystemPoweredOn, boolean
isFullDiskEncrypted)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

3.10 Image Segment Hash Chain Verifier

When a disk image is split into multiple segments, each segment is individually hashed at acquisition time. Before analysis, every segment must be re-hashed and compared to its recorded value — if even one segment fails, the whole image is considered untrustworthy.

Write a function that accepts two equal-length arrays — the recorded segment hashes and the freshly recomputed segment hashes — and returns a list of the (0-based) indices of any segments whose hashes do not match.

Input: recorded = ["h1", "h2", "h3"], recomputed = ["h1", "h9", "h3"]
Output: [1]

Input: recorded = ["h1", "h2"], recomputed = ["h1", "h2"]
Output: []

```
import java.util.*;

public class SegmentHashChainVerifier
{
    public static List<Integer> findFailedSegments(String[] recorded, String[] recomputed)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

Try:

1. Segment Reassembly Validator: Split image segments are named sequentially, e.g. "case1.001", "case1.002", "case1.003". Before reassembling the full image, an examiner must confirm no segment is missing. Your task is to complete the function isSegmentSetComplete() which takes an array of segment file names (in any order, all sharing the same base name) and returns true if the numeric suffixes form an unbroken sequence starting at 001, false if any segment is missing.

Input: segments = ["case1.002", "case1.001", "case1.003"]
Output: true

Input: segments = ["case1.001", "case1.003"]
Output: false

2. Compression Ratio Validator: Some acquisition tools store images in a compressed, sparse format to save storage. As a sanity check, examiners compute the compression ratio and flag any image whose ratio falls outside a plausible range, since an unusually low or high ratio can indicate a corrupted or incomplete image. Your task is to complete the function validateCompressionRatio() which takes the raw disk size and the compressed image size (both in bytes) and returns true only if the compression ratio (raw / compressed) is between 1.0 and 20.0 inclusive.

Input: rawBytes = 10000000, compressedBytes = 2000000
Output: true

Input: rawBytes = 10000000, compressedBytes = 100000
Output: false

4. FILE SYSTEM FORENSICS AND DATA RECOVERY

4.1 File Slack Space Calculator

File systems allocate storage in fixed-size clusters, so a file almost never fills its last cluster exactly — the unused remainder is called 'slack space' and can contain fragments of previously deleted files, making it forensically significant.

Write a function that accepts a file size in bytes and the cluster size in bytes, and returns the slack space in bytes: the difference between the total space allocated to the file (rounded up to a whole number of clusters) and the actual file size.

Input: fileSize = 1000, clusterSize = 4096
Output: 3096

Input: fileSize = 8192, clusterSize = 4096
Output: 0

```
public class SlackSpaceCalculator
{
    public static long computeSlackSpace(long fileSize, long clusterSize)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

4.2 Cluster Chain Fragmentation Checker

A file's data is stored as a chain of clusters. When free space becomes scattered, a file's clusters may no longer be contiguous, which affects how a recovery tool must reconstruct the file.

Write a function that accepts an array of cluster numbers, in the order they belong to a file, and returns true if the clusters are contiguous (each cluster number is exactly one more than the previous), or false if the file is fragmented.

Input: clusters = [100, 101, 102, 103]
Output: true

Input: clusters = [100, 101, 105, 106]
Output: false

```
public class FragmentationChecker
{
    public static boolean isContiguous(int[] clusters)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
```

```
}
```

4.3 FAT Deleted Entry Scanner

In a FAT file system, a directory entry whose first byte has been overwritten with 0xE5 marks the entry as deleted — the rest of the entry (including the starting cluster) is often left intact, which is what makes undelete tools possible. A first byte of 0x00 marks the end of the directory listing, so no entries after it are valid.

Write a function that accepts an array of integers representing the first byte of each successive directory entry, and returns the count of entries marked as deleted (first byte equal to 0xE5, i.e. 229). Stop scanning as soon as a 0x00 (0) entry is reached.

Input: firstBytes = [65, 229, 66, 0, 229]
Output: 1

Input: firstBytes = [229, 229, 229]
Output: 3

```
public class FatDeletedEntryScanner
{
    public static int countDeletedEntries(int[] firstBytes)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

4.4 File Carver (Header/Footer Signature Scan)

When file system metadata is damaged or missing, examiners fall back on 'file carving' — scanning raw, unstructured bytes for known file header and footer signatures to recover embedded files.

Write a function that accepts a byte stream (as an array of integers 0-255), a header signature, and a footer signature, and returns the inclusive [startIndex, endIndex] range of the first embedded file found — the position of the header through the last byte of the following footer. If no complete header-footer pair is found, return "NOT FOUND".

Input: data = [10, 20, 255, 216, 30, 40, 255, 217, 50], header = [255, 216], footer = [255, 217]
Output: [2, 7]

Input: data = [10, 20, 30], header = [255, 216], footer = [255, 217]
Output: "NOT FOUND"

```
public class FileCarver
{
    public static int[] carve(int[] data, int[] header, int[] footer)
    {
        # Write code here – return null (treat as NOT FOUND) if no match
    }
```

```
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

4.5 MAC Timestamp Anomaly (Timestomping) Detector

'Timestomping' is an anti-forensic technique where an attacker deliberately alters a file's Modified, Accessed, and Created (MAC) timestamps to hide when it was really placed on the system. One common giveaway is a Created time that is later than the Modified time, which is not possible under normal file system behaviour.

Write a function that accepts a file's Created and Modified timestamps as Unix epoch values and returns "SUSPICIOUS - CREATED AFTER MODIFIED" if the created time is after the modified time, or "NORMAL" otherwise.

Input: created = 1700000500, modified = 1700000000
Output: "SUSPICIOUS - CREATED AFTER MODIFIED"

Input: created = 1700000000, modified = 1700000500
Output: "NORMAL"

```
public class TimestompDetector
{
    public static String analyze(long created, long modified)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

4.6 NTFS Alternate Data Stream Detector

NTFS allows a file to carry hidden 'alternate data streams' (ADS), referenced with the syntax "filename:streamname". Malware and suspects sometimes use ADS to hide data behind an otherwise innocuous-looking file.

Write a function that accepts a file name string (with no drive letter or path) and returns true if it contains an alternate data stream (i.e. a colon appears after the first character), along with the stream name; return false and an empty stream name if there is no colon.

Input: "document.txt:secret"
Output: hasStream = true, streamName = "secret"

Input: "document.txt"
Output: hasStream = false, streamName = ""

```
public class AdsDetector
{
    public static String[] detectStream(String fileName)
```

```
    {
        # Write code here – return new String[] { "true"/"false", streamName }
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

4.7 Recycle Bin Metadata Parser

When a file is deleted through the operating system's recycle bin, metadata about the original file is preserved separately so it can be restored or examined later. Assume this metadata is provided as a single string of the form "<originalPath>|<deletionTimestamp>|<originalSizeBytes>".

Write a function that accepts this metadata string and returns its three components: the original file path, the deletion timestamp (as a long), and the original file size in bytes (as a long).

Input: "C:\\Users\\bob\\secret.docx|1700000000|20480"
Output: originalPath = "C:\\Users\\bob\\secret.docx", deletionTime = 1700000000, sizeBytes = 20480

Input: "C:\\report.pdf|1699999000|4096"
Output: originalPath = "C:\\report.pdf", deletionTime = 1699999000, sizeBytes = 4096

```
public class RecycleBinMetadataParser
{
    public static String[] parse(String metadata)
    {
        # Write code here – return new String[] { originalPath, deletionTime, sizeBytes }
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

4.8 File Type Mismatch Detector

Suspects often rename a file to disguise its true content (e.g. renaming a hidden ZIP archive to "vacation.jpg"). Comparing the file's extension against its actual magic number reveals this kind of deception.

Write a function that accepts a file name and the first four bytes of its content, identifies the actual file type from the signature (PNG = 89 50 4E 47, JPEG = FF D8 FF E0, PDF = 25 50 44 46, ZIP = 50 4B 03 04), and returns "MATCH" if the extension agrees with the actual type, or "MISMATCH - Actual type <type>" if it does not. Extensions to check: .png, .jpg/.jpeg, .pdf, .zip.

Input: fileName = "photo.png", header = [137, 80, 78, 71]
Output: "MATCH"

Input: fileName = "photo.png", header = [37, 80, 68, 70]
Output: "MISMATCH - Actual type PDF"

```
public class FileTypeMismatchDetector
{
    public static String check(String fileName, int[] header)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

4.9 Cluster-to-Disk-Offset Calculator

To manually extract a specific cluster from a raw disk image (for example, to recover a single deleted file whose starting cluster is known), an examiner must convert a cluster number into a byte offset within the image file.

Write a function that accepts a cluster number, the number of the first cluster in the data area, the cluster size in bytes, and the byte offset where the volume begins on the disk, and returns the byte offset of the start of that cluster within the disk image.

Input: clusterNumber = 10, dataAreaStartCluster = 2, clusterSize = 4096, volumeStartOffset = 1048576
Output: 1081344

Input: clusterNumber = 2, dataAreaStartCluster = 2, clusterSize = 512, volumeStartOffset = 0
Output: 0

```
public class ClusterOffsetCalculator
{
    public static long computeOffset(long clusterNumber, long dataAreaStartCluster, long clusterSize, long volumeStartOffset)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

4.10 Unallocated Space Calculator

Deleted-but-not-yet-overwritten files typically live in a volume's unallocated space, which is why examiners estimate how much unallocated space exists before deciding whether a deep carving pass is worthwhile.

Write a function that accepts a cluster allocation bitmap as a boolean array (true = allocated, false = free) and returns the number of unallocated clusters and the percentage of the volume that is unallocated, rounded to one decimal place.

Input: bitmap = [true, false, false, true]
Output: unallocatedClusters = 2, unallocatedPercent = 50.0

Input: bitmap = [true, true, true, true]
Output: unallocatedClusters = 0, unallocatedPercent = 0.0

```
public class UnallocatedSpaceCalculator
{
    public static double[] computeUnallocated(boolean[] bitmap)
    {
        # Write code here – return new double[] { unallocatedClusters, unallocatedPercent }
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

Try:

1. Journal Transaction Recovery Counter: Journaling file systems record pending changes in a journal before committing them, so that a crash mid-write can be recovered cleanly. Only transactions marked COMMITTED can be safely replayed during recovery; UNCOMMITTED transactions must be discarded. Your task is to complete the function countRecoverableTransactions() which takes an array of transaction status strings and returns the count of COMMITTED entries.

Input: entries = ["COMMITTED", "UNCOMMITTED", "COMMITTED"]
Output: 2

Input: entries = ["UNCOMMITTED", "UNCOMMITTED"]
Output: 0

2. FAT Short Filename Reconstructor: A FAT directory entry stores the short file name as a fixed 11-character field — the first 8 characters are the name (space-padded on the right) and the last 3 are the extension (also space-padded). Your task is to complete the function reconstructFileName() which takes the raw 11-character field and returns it reformatted as "NAME.EXT", with padding trimmed and the dot omitted entirely if there is no extension.

Input: "REPORT TXT"
Output: "REPORT.TXT"

Input: "FOLDER   "
Output: "FOLDER"

5. MEMORY AND VOLATILE DATA FORENSICS

5.1 Process List Parser & Orphan Detector

A raw memory dump can be parsed to reconstruct the list of processes that were running at the moment of capture, even after the system is powered off. Each reconstructed entry has the form "<PID>:<PPID>:<ImageName>". A process whose parent PID (PPID) does not correspond to any PID in the list — and is not the root value 0 — is 'orphaned', which can indicate its true parent already exited or was hidden.

Write a function that accepts an array of process entry strings and returns the list of PIDs (as integers) whose PPID does not match any PID present in the list, ignoring PPID 0.

Input: entries = ["4:0:System", "100:4:explorer.exe", "250:9999:svchost.exe"]
Output: [250]

Input: entries = ["4:0:System", "100:4:explorer.exe"]
Output: []

```
import java.util.*;

public class ProcessListParser
{
    public static List<Integer> findOrphanedProcesses(String[] entries)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

5.2 Hidden Process (Rootkit) Detector

Rootkits often hide a malicious process from the operating system's own process listing (e.g. Task Manager) while the process still physically exists in RAM. Comparing an OS-level process list against a raw memory-scan process list reveals any process the OS is hiding.

Write a function that accepts the list of PIDs reported by the OS and the list of PIDs found by scanning raw memory, and returns the PIDs that appear in the memory scan but not in the OS-level list.

Input: osListPids = [4, 100, 250], memoryListPids = [4, 100, 250, 666]
Output: [666]

Input: osListPids = [4, 100], memoryListPids = [4, 100]
Output: []

```
import java.util.*;

public class HiddenProcessDetector
{
    public static List<Integer> findHiddenProcesses(int[] osListPids, int[] memoryListPids)
    {
        # Write code here
```

```
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

5.3 Suspicious Parent-Child Process Validator

Certain Windows processes are only ever legitimately spawned by one specific parent — for example, svchost.exe should be a child of services.exe, and lsass.exe should be a child of wininit.exe. Malware frequently disguises itself using a legitimate process name but attaches it to the wrong parent.

Write a function that accepts an array of "<childProcess>:<actualParent>" strings and returns the list of child process names whose actual parent does not match the expected parent, using this reference table: svchost.exe -> services.exe, lsass.exe -> wininit.exe, csrss.exe -> smss.exe. Process names not in the reference table are not flagged.

Input: pairs = ["svchost.exe:explorer.exe", "lsass.exe:wininit.exe"]
Output: ["svchost.exe"]

Input: pairs = ["svchost.exe:services.exe", "notepad.exe:explorer.exe"]
Output: []

```
import java.util.*;

public class ParentChildValidator
{
    public static List<String> findSuspiciousPairs(String[] pairs)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

5.4 Established Network Connection Extractor

Open network connections are highly volatile and are usually only recoverable from a live RAM capture. Each recovered connection line has the format "<localIP>:<localPort> -> <remoteIP>:<remotePort> <STATE>".

Write a function that accepts an array of these connection lines and returns only the connections whose state is ESTABLISHED, with the state removed from the returned string.

Input: lines = ["192.168.1.5:49213 -> 8.8.8.8:443 ESTABLISHED", "192.168.1.5:49214 -> 10.0.0.5:80 CLOSED"]
Output: ["192.168.1.5:49213 -> 8.8.8.8:443"]

Input: lines = ["10.0.0.2:80 -> 10.0.0.9:5000 CLOSED"]
Output: []

```
import java.util.*;

public class ConnectionExtractor
{
    public static List<String> extractEstablished(String[] lines)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

5.5 Live Capture Method Selector

The correct way to preserve volatile data depends on the state the investigator finds the system in. Write a function that accepts two booleans — isSystemCrashed and isSystemPoweredOn — and returns the recommended approach as a string.

Rules: if the system has crashed (a crash/blue-screen dump exists), return "ANALYZE HIBERNATION/CRASH DUMP FILE" regardless of power state. Otherwise, if the system is powered on, return "LIVE RAM CAPTURE (software tool)". Otherwise, return "RAM CAPTURE NOT POSSIBLE - SYSTEM OFF, VOLATILE DATA LOST".

Input: isSystemCrashed = false, isSystemPoweredOn = true
Output: "LIVE RAM CAPTURE (software tool)"

Input: isSystemCrashed = false, isSystemPoweredOn = false
Output: "RAM CAPTURE NOT POSSIBLE - SYSTEM OFF, VOLATILE DATA LOST"

Input: isSystemCrashed = true, isSystemPoweredOn = true
Output: "ANALYZE HIBERNATION/CRASH DUMP FILE"

```
public class CaptureMethodSelector
{
    public static String recommendMethod(boolean isSystemCrashed, boolean isSystemPoweredOn)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

5.6 Printable String Carver

The classic first step in analysing a raw memory dump is extracting runs of printable ASCII text, since usernames, URLs, chat fragments, and passwords are often recoverable this way even without understanding the surrounding structure.

Write a function that accepts a memory dump as an array of byte values (integers 0-255) and a minimum length, and returns the list of maximal runs of printable ASCII characters (byte values 32 to 126 inclusive) that are at least the minimum length. A run ends at any byte outside that range.

Input: bytes = [97, 98, 99, 0, 100, 101, 102, 103, 1, 104, 105], minLength = 4
Output: ["defg"]

Input: bytes = [72, 101, 108, 108, 111, 0, 33], minLength = 4
Output: ["Hello"]

```
import java.util.*;

public class StringCarver
{
    public static List<String> carveStrings(int[] bytes, int minLength)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

5.7 Suspicious DLL Path Detector

Process injection often works by loading a malicious DLL into a legitimate process's memory space. Because legitimate system DLLs almost always load from C:\Windows\System32 or C:\Windows\SysWOW64, a module loaded from any other path is worth flagging for review.

Write a function that accepts an array of "<processName>:<modulePath>" strings and returns the entries whose module path is not located under C:\Windows\System32 or C:\Windows\SysWOW64.

Input: modules = ["svchost.exe:C:\\Windows\\System32\\ntdll.dll", "svchost.exe:C:\\Users\\Public\\evil.dll"]
Output: ["svchost.exe:C:\\Users\\Public\\evil.dll"]

Input: modules = ["lsass.exe:C:\\Windows\\SysWOW64\\sspicli.dll"]
Output: []

```
import java.util.*;

public class SuspiciousDllDetector
{
    public static List<String> findSuspiciousModules(String[] modules)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

5.8 Windows FILETIME to Unix Epoch Converter

Windows kernel structures recovered from memory (such as process creation times) store timestamps as FILETIME values — the number of 100-nanosecond intervals since 1 January 1601 (UTC). Investigators must convert these to Unix epoch seconds to build a unified timeline alongside other evidence.

Write a function that accepts a FILETIME value (as a long) and returns the equivalent Unix epoch time in seconds. The offset between the two epochs is 116444736000000000 (in 100-ns units), and there are 10,000,000 such units per second.

Input: filetime = 116444736000000000
Output: 0

Input: filetime = 132223104000000000
Output: 1577836800

```
public class FiletimeConverter
{
    public static long toUnixEpochSeconds(long filetime)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

5.9 Volatile vs Non-Volatile Data Classifier

Before planning acquisition, an investigator must know which sources of evidence will disappear when the system is powered off, and which will not. Write a function that accepts a list of data source names and classifies each one as "VOLATILE" or "NON_VOLATILE".

Classify as VOLATILE: "RAM contents", "Running processes", "Open network connections", "ARP cache". Classify as NON_VOLATILE: "Hard disk contents", "Backup media", "Archived logs". Return the results as "<source>:<classification>" strings, in the same order as the input.

Input: sources = ["RAM contents", "Hard disk contents"]
Output: ["RAM contents:VOLATILE", "Hard disk contents:NON_VOLATILE"]

Input: sources = ["ARP cache", "Backup media"]
Output: ["ARP cache:VOLATILE", "Backup media:NON_VOLATILE"]

```
import java.util.*;

public class VolatileDataClassifier
{
    public static List<String> classify(List<String> sources)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
```

```
    }
}
```

5.10 Suspicious Memory Region Permission Scanner

Memory regions are normally either writable or executable, but rarely both — a region that is simultaneously Readable, Writable, and Executable (RWX) is a classic indicator of injected shellcode, since legitimate code does not need to write to the same memory it executes.

Write a function that accepts an array of "<address>:<protection>" strings and returns the list of addresses whose protection is exactly "RWX".

Input: regions = ["0x1000:RWX", "0x2000:RW", "0x3000:R"]
Output: ["0x1000"]

Input: regions = ["0x5000:RW", "0x6000:R"]
Output: []

```
import java.util.*;

public class MemoryPermissionScanner
{
    public static List<String> findRwxRegions(String[] regions)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

Try:

1. Process Injection Risk Scorer: A single suspicious indicator is rarely conclusive on its own, but a process that trips several indicators at once is a strong candidate for deeper analysis. Your task is to complete the function findHighRiskProcesses() which takes three equal-length arrays for a set of processes — processNames[], isHidden[] (boolean), hasSuspiciousParent[] (boolean), and hasRwxRegion[] (boolean) — and returns the names of processes with two or more true flags.

Input: processNames = ["svc1.exe", "svc2.exe"], isHidden = [true, false], hasSuspiciousParent = [true, false], hasRwxRegion = [false, false]
Output: ["svc1.exe"]

Input: processNames = ["svc1.exe"], isHidden = [true], hasSuspiciousParent = [false], hasRwxRegion = [false]
Output: []

2. Session Token Extractor: Authentication tokens are often still resident in RAM long after a user's session ends, and recovering one from a memory dump can prove a user was logged into a specific service. Your task is to complete the function extractTokens() which takes an array of raw strings recovered from memory and returns every substring that matches the pattern "token=" immediately

followed by exactly 32 hexadecimal characters, returning only the 32-character hex value (not the "token=" prefix).

Input: strings = ["auth token=3f2a9bcd1234567890abcdef12345678 end", "noise"]
Output: ["3f2a9bcd1234567890abcdef12345678"]

Input: strings = ["no tokens here"]
Output: []

NETWORK FORENSICS AND PACKET ANALYSIS

6.1 TTL-Based Hop Count Estimator

Operating systems set a packet's Time-To-Live (TTL) to a well-known starting value, which then decreases by one at every router hop. Estimating the original starting value from a captured packet's TTL lets an investigator estimate how many network hops the packet travelled — useful for spotting spoofed source addresses.

Write a function that accepts the TTL value observed in a captured packet and returns the estimated number of hops travelled, assuming the original TTL was the smallest of these common starting values that is still >= the observed TTL: 64, 128, 255.

Input: observedTTL = 54
Output: 10

Input: observedTTL = 118
Output: 10

Input: observedTTL = 250
Output: 5

```
public class TtlHopEstimator
{
    public static int estimateHops(int observedTTL)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

6.2 Port Scan Detector

A port scan shows up in captured traffic as a single source address making connection attempts to an unusually large number of distinct destination ports in a short window. Each connection attempt is given as "<sourceIP>:<targetPort>".

Write a function that accepts an array of connection attempt strings and a threshold, and returns the list of source IPs that attempted connections to at least `threshold` distinct ports.

Input: attempts = ["10.0.0.5:22", "10.0.0.5:23", "10.0.0.5:80", "10.0.0.9:443"], threshold = 3
Output: ["10.0.0.5"]

Input: attempts = ["10.0.0.9:443", "10.0.0.9:80"], threshold = 3
Output: []

```
import java.util.*;

public class PortScanDetector
{
    public static List<String> detectScanners(String[] attempts, int threshold)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

6.3 Well-Known Port/Protocol Identifier

Quickly identifying what protocol a captured connection belongs to helps an examiner triage a large packet capture. Write a function that accepts a port number and returns the associated protocol name.

Use this mapping: 20 or 21 -> "FTP", 22 -> "SSH", 23 -> "Telnet", 25 -> "SMTP", 53 -> "DNS", 80 -> "HTTP", 110 -> "POP3", 143 -> "IMAP", 443 -> "HTTPS", 3389 -> "RDP". Any other port should return "UNKNOWN".

Input: 443
Output: "HTTPS"

Input: 22
Output: "SSH"

Input: 8080
Output: "UNKNOWN"

```
public class ProtocolIdentifier
{
    public static String identify(int port)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

6.4 DNS Tunneling Indicator Scanner

DNS tunneling — smuggling data out of a network encoded inside DNS queries — typically produces queried domain names with an unusually long subdomain label, since the encoded data has to fit somewhere in the query.

Write a function that accepts a list of queried domain names and returns the ones whose first label (the part before the first dot) is longer than 50 characters, as a simple heuristic for possible tunneling

activity.

Input: domains = ["aGVsbG8gd29ybGQgdGhpcyBpcyBhIHZlcnkgbG9uZyBlbmNvZGVkIHN0cmluZw.example.com", "google.com"]
Output: ["aGVsbG8gd29ybGQgdGhpcyBpcyBhIHZlcnkgbG9uZyBlbmNvZGVkIHN0cmluZw.example.com"]

Input: domains = ["mail.google.com", "www.example.com"]
Output: []

```
import java.util.*;

public class DnsTunnelScanner
{
    public static List<String> flagLongLabels(List<String> domains)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

6.5 HTTP Authentication Failure Counter

A web server access log line has the format "<IP> - - [<time>] \"<METHOD> <path>\" <status>". Repeated 401 (Unauthorized) or 403 (Forbidden) responses from the same source IP are a strong sign of a brute-force login attempt against the site.

Write a function that accepts an array of log lines and a threshold, and returns the list of source IPs that produced at least `threshold` responses with status 401 or 403.

Input: lines = ["10.0.0.1 - - [t] \"GET /login\" 401", "10.0.0.1 - - [t] \"GET /login\" 401", "10.0.0.1 - - [t] \"GET /login\" 401", "10.0.0.2 - - [t] \"GET /\" 200"], threshold = 3
Output: ["10.0.0.1"]

Input: lines = ["10.0.0.2 - - [t] \"GET /\" 200"], threshold = 3
Output: []

```
import java.util.*;

public class HttpAuthFailureCounter
{
    public static List<String> findBruteForceSources(String[] lines, int threshold)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

6.6 ARP Spoofing Detector

In a normal network, one IP address should resolve to exactly one MAC address at any given time. ARP spoofing (a common man-in-the-middle technique) shows up in captured traffic as the same IP address being associated with more than one MAC address.

Write a function that accepts an array of "<IP>|<MAC>" mapping strings observed over the capture period, and returns the list of IP addresses that were seen mapped to more than one distinct MAC address.

Input: mappings = ["192.168.1.1|AA:BB:CC:00:11:22", "192.168.1.1|11:22:33:44:55:66", "192.168.1.2|AA:BB:CC:00:11:23"]
Output: ["192.168.1.1"]

Input: mappings = ["192.168.1.2|AA:BB:CC:00:11:23", "192.168.1.2|AA:BB:CC:00:11:23"]
Output: []

```
import java.util.*;

public class ArpSpoofDetector
{
    public static List<String> findSpoofedIPs(String[] mappings)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

6.7 TCP Flag Combination Classifier

Certain unusual TCP flag combinations are strongly associated with reconnaissance scanning techniques rather than normal traffic — for example, a packet with no flags set at all (a 'NULL scan') or one with FIN, PSH, and URG all set (a 'XMAS scan').

Write a function that accepts a comma-separated string of the flags set on a packet and classifies it. Use these rules: "" (empty) -> "NULL SCAN"; "FIN,PSH,URG" -> "XMAS SCAN"; "SYN" -> "SYN SCAN / CONNECTION ATTEMPT"; "SYN,ACK" -> "HANDSHAKE STEP 2 (NORMAL)"; "ACK" -> "NORMAL ACK"; "RST" -> "CONNECTION RESET"; anything else -> "UNKNOWN FLAG COMBINATION".

Input: "FIN,PSH,URG"
Output: "XMAS SCAN"

Input: ""
Output: "NULL SCAN"

Input: "SYN,ACK"
Output: "HANDSHAKE STEP 2 (NORMAL)"

```
public class TcpFlagClassifier
{
    public static String classify(String flags)
```

```
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

6.8 Packet Capture Size Estimator

Before starting a long packet capture, an examiner needs to estimate the storage the resulting .pcap file will require. Write a function that accepts the average packets captured per second, the average packet size in bytes, and the intended capture duration in seconds, and returns the estimated total capture size in megabytes (1 MB = 1,000,000 bytes), rounded to one decimal place.

Input: packetsPerSecond = 500, avgPacketSizeBytes = 800, durationSeconds = 3600
Output: 1440.0

Input: packetsPerSecond = 100, avgPacketSizeBytes = 1500, durationSeconds = 60
Output: 9.0

```
public class CaptureSizeEstimator
{
    public static double estimateSizeMB(long packetsPerSecond, long avgPacketSizeBytes, long
durationSeconds)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

6.9 Beaconing Interval Detector

Malware command-and-control (C2) traffic often 'phones home' at a regular interval, producing a highly consistent time gap between successive outbound connections to the same destination — a pattern known as beaconing, and a strong indicator when found in a capture.

Write a function that accepts a sorted array of connection timestamps (in seconds) to the same destination and a tolerance in seconds, and returns true if every gap between consecutive timestamps is within the tolerance of the first gap (indicating regular beaconing), or false otherwise.

Input: timestamps = [1000, 1060, 1121, 1180], toleranceSeconds = 5
Output: true

Input: timestamps = [1000, 1060, 1300, 1305], toleranceSeconds = 5
Output: false

```
public class BeaconingDetector
{
    public static boolean isBeaconing(int[] timestamps, int toleranceSeconds)
```

```
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

6.10 Protocol Bandwidth Breakdown Calculator

Summarising which protocols consumed the most bandwidth in a capture is one of the first steps in traffic triage. Write a function that accepts a parallel list of protocol names and their byte counts, and returns each protocol's share of the total traffic as a percentage, formatted as "<protocol>:<percent>%" with the percentage rounded to one decimal place, in the same order as the input.

Input: protocols = ["HTTP", "DNS", "SSH"], byteCounts = [7000, 2000, 1000]
Output: ["HTTP:70.0%", "DNS:20.0%", "SSH:10.0%"]

Input: protocols = ["HTTPS"], byteCounts = [5000]
Output: ["HTTPS:100.0%"]

```
import java.util.*;

public class BandwidthBreakdownCalculator
{
    public static List<String> computeBreakdown(String[] protocols, long[] byteCounts)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

Try:

1. Fragmented Packet Reassembly Validator: Large packets are split into IP fragments before transmission, and each fragment carries its own byte offset and length. A capture only contains a fully reassemblable packet if the fragments cover the total payload size exactly, with no gaps and no overlaps. Your task is to complete the function isReassemblyComplete() which takes an array of [offset, length] fragment pairs and the total expected payload size, and returns true if the fragments — once sorted by offset — cover the range [0, totalSize) exactly once each, or false otherwise.

Input: fragments = [[0, 500], [500, 600]], totalSize = 1100
Output: true

Input: fragments = [[0, 500], [600, 500]], totalSize = 1100
Output: false

2. Outbound Data Exfiltration Detector: A sudden spike in outbound traffic volume, well above a host's normal baseline, can indicate a large file or database being exfiltrated. Your task is to complete the function findExfiltrationHours() which takes an array of hourly outbound byte totals, a baseline byte count, and a multiplier, and returns the list of hour indices (0-based) where the outbound byte total exceeds baseline * multiplier.

Input: hourlyBytes = [1000, 1200, 900, 15000], baseline = 1000, multiplier = 3
Output: [3]

Input: hourlyBytes = [1000, 1200, 900], baseline = 1000, multiplier = 3
Output: []

7. EMAIL AND SOCIAL MEDIA FORENSICS

7.1 Email Header Field Parser

Every email carries a block of header lines before the message body, and pulling out the key fields is the first step in any email examination. Write a function that accepts the raw header as an array of lines (each in the form "<FieldName>: <value>") and returns the values of the From, To, Subject, and Date fields, in that order. If a field is not present, return an empty string for it.

Input: lines = ["From: alice@example.com", "To: bob@example.com", "Subject: Meeting", "Date: Mon, 1 Jan 2024 10:00:00 +0000"]
Output: ["alice@example.com", "bob@example.com", "Meeting", "Mon, 1 Jan 2024 10:00:00 +0000"]

Input: lines = ["From: carol@example.com"]
Output: ["carol@example.com", "", "", ""]

```
public class EmailHeaderParser
{
    public static String[] parseHeaders(String[] lines)
    {
        # Write code here – return new String[] { from, to, subject, date }
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

7.2 Spoofed Sender Detector

A common phishing technique sets the visible 'From' address to look legitimate while quietly routing replies to an attacker-controlled domain via the 'Reply-To' header. Write a function that accepts the From address and the Reply-To address, and returns "SUSPICIOUS - DOMAIN MISMATCH" if their domains differ, or "NORMAL" if the domains match (case-insensitive).

Input: from = "alice@bank.com", replyTo = "alice@bank-support.ru"
Output: "SUSPICIOUS - DOMAIN MISMATCH"

Input: from = "alice@bank.com", replyTo = "support@bank.com"
Output: "NORMAL"

```
public class SpoofedSenderDetector
{
    public static String check(String from, String replyTo)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

7.3 Email Hop Timeline Validator

Each mail server that relays a message adds its own 'Received' header to the top of the header block, so a genuine email's Received timestamps should strictly decrease as you read down the header (newest hop first, oldest last). A forged or manually inserted Received line breaks this ordering.

Write a function that accepts an array of hop timestamps in the order they appear in the header (top to bottom) and returns true if the timestamps are strictly decreasing, or false if any hop is out of order.

Input: timestamps = [1700003000, 1700002000, 1700001000]
Output: true

Input: timestamps = [1700003000, 1700004000, 1700001000]
Output: false

```
public class HopTimelineValidator
{
    public static boolean isValidOrder(long[] timestamps)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

7.4 Double-Extension Attachment Detector

A classic malicious-attachment trick is naming a file so it appears to be a harmless document while actually being executable — for example "invoice.pdf.exe", which many mail clients still display with the .pdf icon truncated.

Write a function that accepts a file name and returns true if it ends in a known executable extension (.exe, .scr, .bat, or .js) AND has another extension-like segment earlier in the name (i.e. the name contains more than one dot), which together indicate a disguised executable.

Input: "invoice.pdf.exe"
Output: true

Input: "photo.jpg"
Output: false

Input: "setup.exe"
Output: false

```
public class DoubleExtensionDetector
{
    public static boolean isDisguisedExecutable(String fileName)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

7.5 Phishing Link Mismatch Detector

Phishing emails frequently display link text that looks like a trustworthy URL while the actual hyperlink target points somewhere else entirely. Write a function that accepts the domain shown as the link's display text and the domain the link actually points to, and returns true if they do not match (case-insensitive), or false if they do.

Input: displayedDomain = "mybank.com", actualDomain = "mybank-secure-login.ru"
Output: true

Input: displayedDomain = "mybank.com", actualDomain = "mybank.com"
Output: false

```
public class PhishingLinkDetector
{
    public static boolean isMismatched(String displayedDomain, String actualDomain)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

7.6 Message-ID Duplicate Detector

Every legitimate email should carry a unique Message-ID. Finding the same Message-ID more than once in a mailbox export can indicate a re-injected, forwarded-and-altered, or spoofed message.

Write a function that accepts an array of Message-ID strings and returns the list of IDs that appear more than once, in the order they were first repeated.

Input: ids = ["<abc123@mail.com>", "<def456@mail.com>", "<abc123@mail.com>"]
Output: ["<abc123@mail.com>"]

Input: ids = ["<a@mail.com>", "<b@mail.com>"]
Output: []

```
import java.util.*;

public class MessageIdDuplicateDetector
{
    public static List<String> findDuplicates(String[] ids)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

7.7 Social Media Snowflake ID Timestamp Decoder

Many social media platforms (e.g. Twitter/X, Discord) generate post and message IDs using a 'snowflake' scheme, where the creation timestamp is embedded directly in the ID's upper bits. This lets an examiner recover a precise creation time even if the platform's displayed timestamp has been edited or removed from a screenshot.

Write a function that accepts a snowflake ID (as a long) and returns its creation time as Unix epoch seconds. Use the Twitter snowflake formula: milliseconds since the platform epoch = (id >> 22); platform epoch = 1288834974657 ms since the Unix epoch. Convert the resulting millisecond timestamp to whole Unix epoch seconds (integer division).

Input: snowflakeId = 0
Output: 1288834974

Input: snowflakeId = 4194304
Output: 1288834974

```
public class SnowflakeDecoder
{
    public static long decodeToUnixSeconds(long snowflakeId)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

7.8 Social Media Username Normalizer

The same person's handle is often referenced inconsistently across a case file — with or without a leading '@', in different letter casing, or with stray whitespace. Normalising every handle to one canonical form makes it possible to reliably cross-reference the same account across evidence sources.

Write a function that accepts a raw username string and returns it normalised: leading/trailing whitespace trimmed, any leading '@' removed, and the result converted to lowercase.

Input: " @Jane_Smith"
Output: "jane_smith"

Input: "JohnDoe123"
Output: "johndoe123"

```
public class UsernameNormalizer
{
    public static String normalize(String rawUsername)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

7.9 Recipient Count Classifier

The number of recipients on a message is a useful first signal when triaging whether an email was a targeted, personal communication or part of a mass-mailing (spam or phishing) campaign. Write a function that accepts the number of To, Cc, and Bcc recipients and classifies the message.

Rules, based on the total recipient count (To + Cc + Bcc): exactly 1 -> "INDIVIDUAL"; 2 to 10 inclusive -> "GROUP"; more than 10 -> "MASS MAILING".

Input: toCount = 1, ccCount = 0, bccCount = 0
Output: "INDIVIDUAL"

Input: toCount = 5, ccCount = 3, bccCount = 0
Output: "GROUP"

Input: toCount = 1, ccCount = 0, bccCount = 50
Output: "MASS MAILING"

```
public class RecipientCountClassifier
{
    public static String classify(int toCount, int ccCount, int bccCount)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

7.10 Deleted Chat Message Fragment Carver

Deleted chat messages often remain physically present in a raw database file export until the storage is overwritten, bounded by structural markers the app's storage format uses internally. Write a function that accepts a raw recovered string and returns every message found between the markers "MSG_START:" and ":MSG_END", in the order they appear.

Input: "junkMSG_START:hello there:MSG_ENDmorejunkMSG_START:bye:MSG_ENDend"
Output: ["hello there", "bye"]

Input: "nothing here"
Output: []

```
import java.util.*;

public class ChatMessageCarver
{
    public static List<String> carveMessages(String raw)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

Try:

1. Received Chain Continuity Checker: Each mail relay hop should hand the message from the server named in its own "by" field to the server that appears as the next hop's "from" field. Any break in that chain suggests a header was forged or removed. Your task is to complete the function countBrokenHops() which takes an array of "<from>:<by>" strings, in header order, and returns the number of places where one hop's "by" value does not match the next hop's "from" value.

Input: hops = ["a:b", "b:c", "x:d"]
Output: 1

Input: hops = ["a:b", "b:c"]
Output: 0

2. Social Media Post Deletion Gap Detector: On platforms where post IDs are issued roughly sequentially over time, an unusually large jump between two consecutive post IDs from the same account can indicate that posts in between were deleted. Your task is to complete the function countSuspiciousGaps() which takes a sorted array of post IDs and a gap threshold, and returns the number of consecutive pairs whose ID difference exceeds the threshold.

Input: postIds = [1001, 1002, 1005, 1006], gapThreshold = 2
Output: 1

Input: postIds = [1001, 1002, 1003], gapThreshold = 2
Output: 0

9. Malware Forensics and Reverse Engineering

These exercises cover static-analysis techniques used to triage a suspicious binary before it is ever executed: recognising file structure, spotting signs of packing or obfuscation, and recovering simple obfuscated data. No exercise involves running, transmitting, or writing self-propagating or damaging code — every input below is a small synthetic byte array or string standing in for a fragment of a real binary.

9.1 PE Header Signature Validator

Before any deeper analysis, an examiner confirms that a file recovered from disk is actually a well-formed Windows executable. A valid PE (Portable Executable) file begins with the two-byte "MZ" DOS signature and contains, at the offset stored in its DOS header, the four-byte "PE\0\0" signature marking the start of the PE header proper.

Write a function that accepts a byte array representing the start of a file and the offset (an int, read from bytes 60–63 of the DOS header in a real file, but supplied directly here) where the PE header should begin, and returns true only if the file starts with "MZ" (0x4D, 0x5A) and the four bytes at the given offset are "PE\0\0" (0x50, 0x45, 0x00, 0x00).

Input: data = [0x4D, 0x5A, 0x00, 0x00, 0x50, 0x45, 0x00, 0x00], peOffset = 4
Output: true

Input: data = [0x25, 0x50, 0x44, 0x46], peOffset = 4
Output: false

```
public class PeHeaderValidator
{
    public static boolean isValidPe(int[] data, int peOffset)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

9.2 Shannon Entropy Calculator

Packed or encrypted malware sections have byte values distributed almost uniformly at random, which shows up as high Shannon entropy (close to the theoretical maximum of 8.0 for a byte stream); ordinary code and text sit noticeably lower. Entropy is one of the first numbers a reverse engineer checks per section.

Write a function that accepts a byte array and returns its Shannon entropy in bits per byte, rounded to two decimal places, using the standard formula $-\Sigma\, p(x) \log_2 p(x)$ over the observed byte-value frequencies.

Input: data = bytes of "AAAAAAAA"
Output: 0.0

Input: data = [0, 1, 2, 3, 4, 5, 6, 7] (eight distinct byte values, each once)

Output: 3.0

```
public class EntropyCalculator
{
    public static double shannonEntropy(int[] data)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

9.3 Suspicious API Import Scanner

A PE file's import table lists every external function it calls. Certain Windows API functions are individually harmless but, in combination, are strongly associated with process injection and other malicious behaviour — a quick static-analysis win before any dynamic analysis is attempted.

Write a function that accepts an array of imported function names and returns, in their original order, only the names that appear in this watch-list: "VirtualAllocEx", "WriteProcessMemory", "CreateRemoteThread", "SetWindowsHookEx", "NtUnmapViewOfSection". The comparison should be exact (case-sensitive).

Input: ["GetProcAddress", "VirtualAllocEx", "CreateRemoteThread", "printf"]
Output: ["VirtualAllocEx", "CreateRemoteThread"]

Input: ["ReadFile", "WriteFile", "CloseHandle"]
Output: []

```
public class SuspiciousImportScanner
{
    public static String[] scan(String[] imports)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

9.4 Printable String Extractor

The Unix "strings" utility — one of the most-used tools in malware triage — scans a binary for runs of printable characters, which often reveal hard-coded URLs, file paths, or error messages even inside a stripped or partially obfuscated sample.

Write a function that accepts a byte array and a minimum length, and returns an array of every maximal run of printable ASCII bytes (0x20–0x7E inclusive) that is at least the minimum length, as strings, in the order they occur. A run ends at the first non-printable byte or the end of the array.

Input: data = bytes of "AB\u00003http://evil.example\u0000ok", minLength = 4

Output: ["http://evil.example"]

Input: data = bytes of "hi\u0000there", minLength = 5
Output: ["there"]

```
public class StringExtractor
{
    public static String[] extractStrings(int[] data, int minLength)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

9.5 PE Section Name Anomaly Detector

Compilers and linkers produce a predictable, small set of section names (".text", ".data", ".rdata", ".rsrc", ".reloc"). Packers and custom loaders frequently rename or add sections, so an unrecognised section name is a lightweight but useful indicator worth flagging for closer review.

Write a function that accepts an array of section names recovered from a PE file and returns the subset that are not in the standard set { ".text", ".data", ".rdata", ".rsrc", ".reloc", ".idata", ".pdata" }, preserving their original order.

Input: [".text", ".data", ".UPX0", ".UPX1", ".rsrc"]
Output: [".UPX0", ".UPX1"]

Input: [".text", ".data", ".rdata"]
Output: []

```
public class SectionNameAnomalyDetector
{
    public static String[] findAnomalousSections(String[] sectionNames)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

9.6 Single-Byte XOR Key Recovery

Malware configuration blocks and dropped strings are frequently obfuscated with a single-byte XOR key rather than real encryption. Because English text has a very high frequency of the space character (0x20), the correct key is almost always the one that produces the most space bytes when applied to the ciphertext — a classic, fast recovery technique.

Write a function that accepts a byte array of suspected single-byte-XOR-obfuscated data and returns the single-byte key (0–255) that, when XOR-ed against every byte, produces the greatest number of resulting 0x20 (space) bytes. If there is a tie, return the smallest such key.

Input: data = bytes of "hello" each XOR-ed with 0x05
Output: 5

```
public class XorKeyRecovery
{
    public static int recoverKey(int[] data)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

9.7 Import Hash (Imphash) Calculator

An "imphash" fingerprints a binary by its imported functions rather than its file bytes, so two builds of the same malware family that differ only in packing or a recompiled resource section still produce the same imphash — a powerful way to cluster related samples.

Write a function that accepts an array of (dllName, functionName) import pairs in the order they appear in the import table, and returns an MD5 hash of the string formed by lower-casing each "dllname.functionname" pair, joining them with commas, and hashing the result (see Exercise 1.2 for computing an MD5-style digest with a standard library, substituting MessageDigest.getInstance("MD5")).

Input: imports = [("KERNEL32.dll", "VirtualAlloc"), ("USER32.dll", "MessageBoxA")]
Output: "<32-character lowercase hex MD5 digest of \"kernel32.dll.virtualalloc,user32.dll.messageboxa\">"

```
import java.security.MessageDigest;
```

```
public class ImphashCalculator
{
    public static String computeImphash(String[][] imports)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

9.8 YARA-Style Rule Matcher

YARA is the industry-standard tool for writing pattern-based malware detection rules: a rule lists several string patterns and a condition for how many must be present. This exercise implements a simplified single-rule, "match all" version of that idea.

Write a function that accepts the full text of a file (as a string) and an array of required substrings, and returns true only if every required substring is present somewhere in the file text — mirroring a YARA rule whose condition is "all of them".

Input: `fileText = "this dropper calls VirtualAllocEx then WriteProcessMemory before exiting", patterns = ["VirtualAllocEx", "WriteProcessMemory"]`
Output: `true`

Input: `fileText = "this dropper calls VirtualAllocEx before exiting", patterns = ["VirtualAllocEx", "WriteProcessMemory"]`
Output: `false`

```
public class YaraStyleMatcher
{
    public static boolean matchesAll(String fileText, String[] patterns)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

Try:

1. **Packed Executable Score Aggregator:** No single indicator proves a file is packed, but several weak indicators together are convincing. Your task is to complete the function packingScore(), which takes the average section entropy (double), the number of anomalous section names (int, from Exercise 9.5), and the number of suspicious imports found (int, from Exercise 9.3), and returns an overall risk score out of 100: entropy contributes up to 60 points scaled linearly from 0.0–8.0, anomalous sections contribute 10 points each (capped at 20), and suspicious imports contribute 5 points each (capped at 20). Round the total to the nearest whole number and cap it at 100.

Input: `avgEntropy = 7.9, anomalousSections = 2, suspiciousImports = 3`
Output: `94`

2. **Mini Opcode Disassembler:** A full disassembler decodes machine code into human-readable assembly mnemonics. Your task is to complete the function disassemble(), which takes an array of raw opcode bytes and returns an array of mnemonics using this reduced instruction set (single-byte opcodes only): 0x90 → "NOP", 0xC3 → "RET", 0xCC → "INT3", 0x50 → "PUSH EAX", 0x58 → "POP EAX". Any byte not in this table should map to "UNKNOWN".

Input: `opcodes = [0x90, 0x50, 0xCC, 0x58, 0xC3, 0xF4]`
Output: `["NOP", "PUSH EAX", "INT3", "POP EAX", "RET", "UNKNOWN"]`

10. Web and Dark Web Forensics

These exercises cover artifacts an examiner recovers from a suspect's browser — history, cache, cookies, and downloads — and the address and pattern conventions used when tracing activity involving anonymised (Tor/.onion) services and cryptocurrency payments. All URLs, addresses, and records below are synthetic and used for practice only; no exercise involves accessing any live site, network, or service.

10.1 URL Component Parser

Browser history entries are raw URLs. Before they can be summarised or clustered in a report, an examiner needs to split each one into its component parts.

Write a function that accepts a URL string and returns its scheme, host, and path as an array of three strings, in that order. Assume the URL is well-formed and always includes a scheme (e.g. "http" or "https") and a host; if there is no path, return "/" for the path component and ignore any query string or fragment.

Input: "https://mail.example.com/inbox?tab=unread"
Output: ["https", "mail.example.com", "/inbox"]

Input: "http://example.com"
Output: ["http", "example.com", "/"]

```
public class UrlComponentParser
{
    public static String[] parse(String url)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

10.2 .onion Address Format Validator

Tor hidden-service addresses recovered from a browser's history or bookmarks follow a strict format. Version 3 addresses consist of exactly 56 lowercase base32 characters (a–z, 2–7) followed by the suffix ".onion"; anything else is not a valid v3 address.

Write a function that accepts a string and returns true if it is a syntactically valid Tor v3 .onion address: exactly 56 characters from the set [a-z2-7] followed by ".onion", and nothing else.

Input: "abcdefghijklmnopqrstuvwxyz234567abcdefghijklmnopqrstuvwx.onion"
Output: true

Input: "example.onion"
Output: false

Input: "abcdefghijklmnopqrstuvwxyz234567abcdefghijklmnopqrstuvwx.com"
Output: false

```
public class OnionAddressValidator
{
    public static boolean isValidV3(String address)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

10.3 WebKit Timestamp Converter

Chrome and other WebKit-based browsers store history and cookie timestamps not as a Unix epoch, but as microseconds since 1 January 1601, 00:00:00 UTC — a detail examiners must account for or a timeline reconstruction will be off by centuries.

Write a function that accepts a WebKit timestamp (microseconds since 1601-01-01 UTC) and returns the corresponding UTC date-time as a string in the format "yyyy-MM-dd HH:mm:ss". You may find it useful that there are 11,644,473,600 seconds between 1601-01-01 and the Unix epoch (1970-01-01).

Input: 13303564800000000
Output: "2022-07-29 10:40:00"

Input: 11644473600000000
Output: "1970-01-01 00:00:00"

```
import java.time.Instant;
import java.time.ZoneOffset;
import java.time.format.DateTimeFormatter;
```

```
public class WebKitTimestampConverter
{
    public static String toReadableDate(long webkitMicros)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

10.4 Cookie Persistence Classifier

Recovered cookies fall into two categories that matter to an examiner differently: session cookies (deleted when the browser closes, so their presence on disk suggests a crash or improper shutdown) and persistent cookies (which survive and can indicate long-term tracking or a remembered login).

Write a function that accepts a cookie's expiry value, given as a Unix epoch timestamp, or -1 if the cookie has no expiry attribute at all, and returns "SESSION" if the value is -1, or "PERSISTENT" otherwise.

Input: expiry = -1

Output: "SESSION"

Input: expiry = 1893456000
Output: "PERSISTENT"

```
public class CookiePersistenceClassifier
{
    public static String classify(long expiry)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

10.5 Download Record Extension Mismatch Detector

A browser's download history records both the saved file name and, separately, the MIME type reported by the server. A mismatch between the two — a file saved as ".jpg" but reported as "application/x-msdownload" — is a strong signal that a download was disguised.

Write a function that accepts a saved file name and its reported MIME type, and returns true if they are inconsistent. Use this mapping of expected extensions per MIME type: "image/jpeg" → .jpg/.jpeg, "application/pdf" → .pdf, "application/zip" → .zip. A MIME type not in this list is never flagged as a mismatch.

Input: fileName = "invoice.jpg", mimeType = "application/x-msdownload"
Output: false

Input: fileName = "photo.jpg", mimeType = "application/zip"
Output: true

Input: fileName = "report.pdf", mimeType = "application/pdf"
Output: false

```
public class DownloadExtensionMismatchDetector
{
    public static boolean isMismatch(String fileName, String mimeType)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

10.6 Search Term Extractor

Search engine queries embedded in browser history URLs are often one of the most revealing artifacts in an investigation — they show intent, not just activity.

Write a function that accepts a search-results URL and returns the decoded search term, taken from its "q" query parameter. Replace any "+" characters in the parameter value with spaces. If there is no "q" parameter, return an empty string.

Input: "https://search.example.com/results?q=how+to+wipe+a+hard+drive&hl=en"
Output: "how to wipe a hard drive"

Input: "https://search.example.com/results?hl=en"
Output: ""

```
public class SearchTermExtractor
{
    public static String extractQuery(String url)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

10.7 Most-Visited Domain Analyzer

Summarising thousands of raw history entries down to “which sites did this person actually use” is a routine first step before an examiner reads any individual URL in detail.

Write a function that accepts an array of full URLs from a browser history export and returns the single host name that appears most often. Extract the host the same way as Exercise 10.1. If there is a tie, return whichever host occurs first among the tied hosts, in order of first appearance.

Input: ["https://mail.example.com/a", "https://mail.example.com/b", "https://news.example.com"]
Output: "mail.example.com"

```
public class MostVisitedDomainAnalyzer
{
    public static String mostVisitedHost(String[] urls)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

10.8 Cryptocurrency Address Classifier

Payments on dark web marketplaces are typically made in cryptocurrency, so financial tracing starts with recognising what kind of address a recovered string actually is, before any lookup against a blockchain explorer takes place.

Write a function that accepts a string and classifies it using these simplified format rules, checked in order: starts with "1" or "3" and is 26–35 characters → "BITCOIN"; starts with "bc1" → "BITCOIN"; starts with "0x" and is 42 characters total → "ETHEREUM"; otherwise → "UNKNOWN".

Input: "1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa"
Output: "BITCOIN"

Input: "0x71C7656EC7ab88b098defB751B7401B5f6d8976F"
Output: "ETHEREUM"

Input: "not-an-address"
Output: "UNKNOWN"

```
public class CryptoAddressClassifier
{
    public static String classify(String address)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

Try:

1. **Browsing Session Reconstructor:** Raw browser history is just a flat list of (url, timestamp) visits, but investigators reason about sessions — continuous stretches of activity. Your task is to complete the function groupIntoSessions(), which takes a chronologically sorted array of visit timestamps (seconds) and returns a list of sessions, where a new session begins whenever the gap since the previous visit exceeds 1800 seconds (30 minutes). Each session should be represented as the array of indices belonging to it.

Input: timestamps = [100, 250, 400, 5000, 5100]
Output: [[0, 1, 2], [3, 4]]

2. **Referrer Chain Builder:** Phishing kits and drive-by-download pages often reveal themselves through an unusually long chain of redirects. Your task is to complete the function buildReferrerChain(), which takes a map of pageUrl → referrerUrl (or null if there is no referrer) and a starting page, and returns the full chain of pages from the starting page back to the original page with no referrer, in order from the starting page to the origin. If a cycle is detected (a page refers back to a page already in the chain), stop and return the chain built so far.

Input: referrers = {"C": "B", "B": "A", "A": null}, start = "C"
Output: ["C", "B", "A"]

11. Insider Threat Investigation and Cyber Fraud Detection

These exercises focus on threats that originate from within an organisation — employees or contractors misusing legitimate access — and on detecting financial fraud patterns in transaction data. As with earlier exercises, all employees, accounts, and records below are synthetic and used for practice only.

11.1 After-Hours Access Flagger

Insider threat programs often start with a simple baseline: an employee accessing systems well outside their normal working hours is not proof of wrongdoing, but it is worth a second look, especially in the days before a resignation.

Write a function that accepts an access timestamp's hour (0–23, in the employee's local time) and returns true if the hour falls outside standard business hours, defined as 07:00–19:00 inclusive of 07 and exclusive of 19.

Input: hour = 2
Output: true

Input: hour = 9
Output: false

Input: hour = 19
Output: true

```
public class AfterHoursFlagger
{
    public static boolean isAfterHours(int hour)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

11.2 Departing Employee Data Movement Detector

A well-documented insider threat pattern is a spike in data downloads or transfers in the final days before an employee's last day — often described as a “flight risk” indicator by insider threat analysts.

Write a function that accepts an array of daily download volumes in megabytes for the 30 days before an employee's departure, and returns true if the volume on any of the final 5 days is more than 5 times the average of the preceding 25 days.

Input: dailyVolumes = [50, 48, 52, 49, 51, ... (25 days near 50), 40, 45, 900, 50, 48]
Output: true

Input: dailyVolumes = [50, 48, 52, 49, 51, ... (25 days near 50), 55, 47, 60, 52, 49]

Output: false

```
public class DepartureDataMovementDetector
{
    public static boolean isSuspiciousSpike(int[] dailyVolumes)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

11.3 Privilege Creep Auditor

"Privilege creep" occurs when an employee accumulates access rights across several role changes without old, no-longer-needed permissions ever being revoked — a common root cause of insider misuse.

Write a function that accepts two arrays of permission strings — the permissions required by an employee's current role, and the permissions the employee currently holds — and returns an array of the excess permissions the employee holds but does not need, sorted alphabetically.

Input: required = ["READ_HR", "READ_PAYROLL"], held = ["READ_HR", "READ_PAYROLL", "WRITE_FINANCE", "ADMIN_VPN"]
Output: ["ADMIN_VPN", "WRITE_FINANCE"]

Input: required = ["READ_HR"], held = ["READ_HR"]
Output: []

```
public class PrivilegeCreepAuditor
{
    public static String[] findExcessPermissions(String[] required, String[] held)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

11.4 Benford's Law Fraud Screener

Fraud examiners use Benford's Law — the observation that in most naturally occurring sets of numbers, the leading digit 1 appears far more often than 9 — as a first-pass screen for fabricated financial figures, which tend to have an unnaturally even spread of leading digits.

Write a function that accepts an array of positive transaction amounts and returns the leading digit (1–9) that occurs most frequently across the set. If there is a tie, return the smallest such digit.

Input: amounts = [123.45, 156.00, 199.99, 245.10, 999.00]
Output: 1

Input: amounts = [500.00, 512.00, 590.00, 250.00]
Output: 5

```
public class BenfordScreener
{
    public static int mostCommonLeadingDigit(double[] amounts)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

11.5 Duplicate Payment Detector

Duplicate or near-duplicate payments to the same vendor — the same amount, paid twice within a short window — are one of the most common accounts-payable fraud and error patterns an investigator screens for.

Write a function that accepts an array of payments, each described as (vendorId, amount, dayNumber), and returns the indices of any payment pair to the same vendor, for the same amount, made within 3 days of each other.

Input: payments = [("V1", 4200.00, 10), ("V2", 500.00, 11), ("V1", 4200.00, 12)]
Output: [0, 2]

Input: payments = [("V1", 4200.00, 10), ("V1", 4200.00, 40)]
Output: []

```
public class DuplicatePaymentDetector
{
    public static int[] findDuplicatePair(Object[][] payments)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

11.6 Shared Credential Usage Detector

Account sharing undermines accountability during an investigation: if two logins under the same username occur from physically distant locations too close together in time, the account is almost certainly shared or compromised (an “impossible travel” indicator).

Write a function that accepts two consecutive login events for the same username — each as (timestamp in seconds, distanceFromPreviousKm) — and returns true if the implied speed between them exceeds 900 km/h (faster than commercial air travel).

Input: secondsBetween = 1800, distanceKm = 800
Output: true

Input: secondsBetween = 36000, distanceKm = 800
Output: false

```
public class SharedCredentialDetector
{
    public static boolean isImpossibleTravel(long secondsBetween, double distanceKm)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

11.7 Segregation of Duties (SoD) Checker

Segregation of Duties is a core financial control: the person who creates a vendor record should not also be the person who approves payments to that vendor, since together those two permissions allow a single employee to create a fake vendor and pay themself.

Write a function that accepts a list of conflicting permission pairs and a set of permissions held by one employee, and returns true if the employee holds both permissions from any conflicting pair — an SoD violation.

Input: conflictingPairs = [("CREATE_VENDOR", "APPROVE_PAYMENT")], held = ["CREATE_VENDOR", "APPROVE_PAYMENT", "READ_HR"]
Output: true

Input: conflictingPairs = [("CREATE_VENDOR", "APPROVE_PAYMENT")], held = ["CREATE_VENDOR", "READ_HR"]
Output: false

```
public class SegregationOfDutiesChecker
{
    public static boolean hasViolation(String[][] conflictingPairs, String[] held)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

11.8 Insider Risk Score Aggregator

Insider threat programs typically combine several weak individual signals — none of which alone is proof of anything — into a single risk score so investigators can prioritise which employees to review first.

Write a function that accepts five booleans for a single employee — afterHoursAccess, dataMovementSpike, excessPrivileges, sodViolation, and recentPerformanceFlag (e.g. a poor review or disciplinary note) — and returns the number of true signals as the employee's risk score (0–5).

Input: `afterHoursAccess = true, dataMovementSpike = true, excessPrivileges = false, sodViolation = false, recentPerformanceFlag = true`
Output: `3`

Input: `afterHoursAccess = false, dataMovementSpike = false, excessPrivileges = false, sodViolation = false, recentPerformanceFlag = false`
Output: `0`

```
public class InsiderRiskScorer
{
    public static int score(boolean afterHoursAccess, boolean dataMovementSpike, boolean
excessPrivileges, boolean sodViolation, boolean recentPerformanceFlag)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

Try:

1. **Round-Tripping Detector:** “Round-tripping” is a fraud technique where money is moved through a chain of accounts and returns close to its starting point, disguising its origin. Your task is to complete the function findRoundTrip(), which takes an array of transactions, each as (fromAccount, toAccount, amount, day), and returns true if there exists a chain of transactions starting and ending at the same account, each transaction occurring on a later day than the one before it, and each amount within 2% of the original amount.

Input: `transactions = [("A","B",10000,1), ("B","C",9950,3), ("C","A",9900,6)]`
Output: `true`

2. **Collusion Ring Finder:** Two or more employees who approve each other's expense claims — rather than each having an independent approver — can defeat a single-approver control. Your task is to complete the function findCollusionPairs(), which takes an array of (submitterId, approverId) approval records and returns every pair of employees who have each approved at least one of the other's claims (a mutual approval relationship), with each pair reported once.

Input: `approvals = [("E1","E2"), ("E2","E1"), ("E3","E4")]`
Output: `[("E1", "E2")]`

12. Intrusion Detection and Prevention Systems (IDS/IPS)

These exercises model the core logic behind Intrusion Detection and Prevention Systems: matching traffic against known attack signatures, spotting anomalous behaviour, and evaluating rule sets and detector performance. As with earlier exercises, all packets, addresses, and rules below are synthetic and used for practice only.

12.1 Signature-Based Pattern Matcher

The simplest form of intrusion detection compares each packet's payload against a database of known attack signatures — fixed byte or text patterns seen in previous exploits.

Write a function that accepts a packet payload string and an array of signature strings, and returns the first signature found as a substring of the payload, or null if none match. Check signatures in the order given.

Input: payload = "GET /cgi-bin/../../etc/passwd HTTP/1.1", signatures = ["../../", "DROP TABLE"]
Output: "../../"

Input: payload = "GET /index.html HTTP/1.1", signatures = ["../../", "DROP TABLE"]
Output: null

```
public class SignatureMatcher
{
    public static String findSignature(String payload, String[] signatures)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

12.2 SYN Flood Detector

A SYN flood is a denial-of-service technique in which an attacker sends many TCP connection requests (SYN packets) without ever completing the handshake, exhausting the target's connection table. An IPS watches for a source that opens far more connections than it completes.

Write a function that accepts the number of SYN packets and the number of completed handshakes (SYN-ACK-ACK) observed from a single source IP within a monitoring window, and returns true if the source should be flagged as a SYN flood: at least 20 SYN packets were seen, and fewer than 10% of them were completed.

Input: synCount = 500, completedCount = 30
Output: true

Input: synCount = 15, completedCount = 0
Output: false

```
public class SynFloodDetector
{
    public static boolean isSynFlood(int synCount, int completedCount)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

12.3 Protocol-Port Anomaly Detector

Attackers sometimes tunnel one protocol over the port conventionally reserved for another (e.g. running a shell over port 80) to slip past simple firewalls. A protocol-aware IDS checks that traffic on a well-known port actually matches the protocol expected there.

Write a function that accepts a destination port and a declared application-layer protocol, and returns true if the pairing is anomalous. Expected pairings are: 80 → "HTTP", 443 → "TLS", 22 → "SSH", 53 → "DNS". A port from this list paired with any other protocol is anomalous; a port not in this list is never anomalous.

Input: port = 80, protocol = "SSH"
Output: true

Input: port = 443, protocol = "TLS"
Output: false

Input: port = 8080, protocol = "SSH"
Output: false

```
public class ProtocolPortAnomalyDetector
{
    public static boolean isAnomalous(int port, String protocol)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

12.4 Payload Length Anomaly Detector

Exploits that rely on buffer overflows often send payloads far longer than a protocol would ever legitimately require. A simple anomaly check flags any payload whose length is a statistical outlier relative to typical traffic for that protocol.

Write a function that accepts an array of recent payload lengths (bytes) for a given protocol and a new payload length, and returns true if the new length exceeds the mean of the recent lengths by more than three times their standard deviation.

Input: recentLengths = [200, 210, 190, 205, 195], newLength = 4096
Output: true

Input: recentLengths = [200, 210, 190, 205, 195], newLength = 220
Output: false

```
public class PayloadLengthAnomalyDetector
{
    public static boolean isOutlier(int[] recentLengths, int newLength)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

12.5 Firewall Rule Engine (First-Match)

IPS devices enforce an ordered list of rules against each packet and apply the action of the first rule that matches, falling back to a default action if nothing matches — the same first-match semantics used by real firewalls such as iptables.

Write a function that accepts an array of rules, each formatted as "ACTION srcIp dstPort" (where either field may be "*" to match anything), plus a packet's source IP and destination port, and returns the action ("ALLOW" or "DENY") of the first matching rule. If no rule matches, return "DENY" (default-deny).

Input: rules = ["ALLOW 10.0.0.0 22", "DENY * 22", "ALLOW * *"], srcIp = "192.168.1.5", dstPort = 22
Output: "DENY"

Input: rules = ["ALLOW 10.0.0.0 22", "DENY * 22", "ALLOW * *"], srcIp = "10.0.0.0", dstPort = 22
Output: "ALLOW"

```
public class FirewallRuleEngine
{
    public static String evaluate(String[] rules, String srcIp, int dstPort)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

12.6 SQL Injection Pattern Detector

Web application firewalls (a form of IPS) inspect form input for classic SQL injection markers before it ever reaches the database layer.

Write a function that accepts a user-submitted input string and returns true if it contains any of the following case-insensitive patterns commonly seen in SQL injection attempts: "' or '1'='1", "union select", "drop table", "--" (a SQL comment marker). Otherwise return false.

Input: "admin' OR '1'='1"
Output: true

Input: "1; DROP TABLE users;--"
Output: true

Input: "ordinary search term"
Output: false

```
public class SqlInjectionDetector
{
    public static boolean containsSqlInjection(String input)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

12.7 Fragmentation Overlap Detector

Fragmentation-based evasion techniques (such as "teardrop"-style attacks) send IP fragments with deliberately overlapping byte ranges so that the IDS and the target host reassemble the packet differently, letting malicious content slip past inspection.

Write a function that accepts an array of fragments, each given as (offset, length) in bytes, and returns true if any two fragments overlap — that is, their byte ranges [offset, offset + length) intersect.

Input: fragments = [(0, 100), (100, 100), (250, 50)]
Output: false

Input: fragments = [(0, 100), (80, 50)]
Output: true

```
public class FragmentationOverlapDetector
{
    public static boolean hasOverlap(int[][] fragments)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

12.8 Detector Accuracy Evaluator

An IDS is only as useful as its accuracy: too many false positives overwhelm analysts, while missed detections leave real intrusions unnoticed. Evaluating a detector against a labelled test set is a routine tuning step before deployment.

Write a function that accepts the counts of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) from a test run, and returns an array of two values: the precision (TP / (TP + FP)) and the recall (TP / (TP + FN)), each rounded to two decimal places. If a denominator is zero, return 0.0 for that value.

Input: tp = 80, fp = 20, tn = 850, fn = 50
Output: [0.80, 0.62]

Input: tp = 0, fp = 0, tn = 100, fn = 10
Output: [0.0, 0.0]

```
public class DetectorAccuracyEvaluator
{
    public static double[] evaluate(int tp, int fp, int tn, int fn)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

Try:

1. **Stateful Connection Tracker:** A stateful firewall allows inbound return traffic for a connection the internal network itself initiated, without needing an explicit rule for it. Your task is to complete the function isAllowed(), which processes a chronological list of packets, each described as (direction, srcIp, srcPort, dstIp, dstPort). For an OUTBOUND packet, record its (srcIp, srcPort, dstIp, dstPort) tuple as an established connection. For an INBOUND packet, allow it only if it is the reverse of a previously established outbound connection (i.e. its own srcIp/srcPort/dstIp/dstPort match dstIp/dstPort/srcIp/srcPort of a prior OUTBOUND packet); otherwise deny it. Return an array of "ALLOW"/"DENY" decisions, one per packet, in order.

Input: packets = [("OUTBOUND","10.0.0.5",5000,"93.184.216.34",443),
("INBOUND","93.184.216.34",443,"10.0.0.5",5000),
("INBOUND","198.51.100.9",80,"10.0.0.5",6000)]
Output: ["ALLOW", "ALLOW", "DENY"]

2. **Mini Signature-Rule Matcher:** Real IDS engines (such as Snort) match packets against rules that constrain several fields at once. Your task is to complete the function matchRule(), which takes a packet (protocol, dstPort, payload) and an array of rules formatted as "protocol dstPort contains:pattern", and returns the index of the first rule that matches all three conditions (protocol equality, port equality, and the payload containing the given pattern), or -1 if no rule matches.

Input: packet = ("TCP", 80, "GET /../../etc/passwd"), rules = ["UDP 53 contains:AAAA", "TCP 80 contains:../.."]
Output: 1

13. Incident Response and Threat Intelligence

These exercises simulate tasks performed by a Security Operations Centre (SOC) analyst or incident responder: triaging alerts, matching indicators against threat intelligence feeds, and measuring how quickly an organisation detects and responds to an incident. As with earlier exercises, all indicators, addresses, and logs below are synthetic and used for practice only.

13.1 Indicator of Compromise (IOC) Matcher

Threat intelligence feeds publish lists of Indicators of Compromise (IOCs) — IP addresses, file hashes, or domains known to be associated with malicious activity. When triaging a batch of extracted values from a log, an analyst needs to know instantly which ones already appear on a known-bad list.

Write a function that accepts an array of extracted indicators and an array of known-bad indicators, and returns a new array containing only the extracted indicators that also appear in the known-bad list, in their original order. The comparison should be case-insensitive.

Input: `extracted = ["203.0.113.9", "8.8.8.8", "45.33.32.156"], knownBad = ["45.33.32.156", "198.51.100.23"]`
Output: `["45.33.32.156"]`

Input: `extracted = ["EVIL.COM", "example.com"], knownBad = ["evil.com"]`
Output: `["EVIL.COM"]`

```
public class IocMatcher
{
    public static String[] findMatches(String[] extracted, String[] knownBad)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

13.2 Alert Severity Scorer

Not every alert deserves the same attention. Analysts triage incoming alerts by scoring them against a small set of risk factors so the highest-risk items surface first.

Write a function that accepts three booleans — externalSource (the alert originated outside the corporate network), knownIocMatch (the alert matched a threat-intel indicator), and sensitiveAsset (the target is a sensitive asset, e.g. a domain controller) — and returns a severity of "LOW", "MEDIUM", "HIGH", or "CRITICAL". Award one point for each true value: 0 points = LOW, 1 point = MEDIUM, 2 points = HIGH, 3 points = CRITICAL.

Input: `externalSource = false, knownIocMatch = false, sensitiveAsset = false`
Output: `"LOW"`

Input: `externalSource = true, knownIocMatch = true, sensitiveAsset = true`
Output: `"CRITICAL"`

```
public class AlertSeverityScorer
{
    public static String score(boolean externalSource, boolean knownIocMatch, boolean
sensitiveAsset)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

13.3 CIDR Range Matcher

Firewall and allow-list rules are usually written as CIDR blocks (e.g. "10.0.0.0/24") rather than individual addresses. Before an analyst can say whether a suspicious connection came from inside a trusted range, they need to test an IP address against a CIDR block.

Write a function that accepts an IPv4 address and a CIDR block (both as strings) and returns true if the address falls within that block, false otherwise. You may assume both inputs are well-formed IPv4 values (see Exercise 1.4).

Input: ip = "192.168.1.57", cidr = "192.168.1.0/24"
Output: true

Input: ip = "10.0.5.20", cidr = "10.0.0.0/16"
Output: true

Input: ip = "172.16.5.1", cidr = "172.16.0.0/24"
Output: false

```
public class CidrMatcher
{
    public static boolean isInRange(String ip, String cidr)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

13.4 Typosquat Domain Detector

Phishing campaigns frequently register domains that closely resemble a trusted brand (e.g. "paypa1.com" instead of "paypal.com") to trick users. Threat intel triage flags any observed domain that is suspiciously close — but not identical — to a protected brand domain.

Write a function that accepts an observed domain and a protected brand domain, and returns true if the observed domain is a likely typosquat: its edit distance (see Exercise “Try” 2, Fuzzy Filename Matcher)

from the brand domain is between 1 and 2 inclusive. Return false if the domains are identical (distance 0) or too different (distance > 2).

Input: observed = "paypa1.com", brand = "paypal.com"
Output: true

Input: observed = "paypal.com", brand = "paypal.com"
Output: false

Input: observed = "wellsfargo-support.net", brand = "wellsfargo.com"
Output: false

```
public class TyposquatDetector
{
    public static boolean isLikelyTyposquat(String observed, String brand)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

13.5 Alert Burst (Port-Scan) Detector

A single source triggering many alerts in a short window is a classic sign of automated scanning rather than isolated, incidental traffic. Analysts set a threshold: a source is flagged if it produces at least N alerts within any T-second sliding window.

Write a function that accepts a chronologically sorted array of alert timestamps (in seconds, all from the same source), a window length T, and a threshold N, and returns true if any window of T seconds contains at least N timestamps.

Input: timestamps = [1, 2, 3, 4, 20], windowSeconds = 5, threshold = 4
Output: true

Input: timestamps = [1, 10, 20, 30], windowSeconds = 5, threshold = 2
Output: false

```
public class PortScanDetector
{
    public static boolean isBurst(int[] timestamps, int windowSeconds, int threshold)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

13.6 Threat Feed Deduplicator

Analysts often subscribe to several threat intelligence feeds at once, which frequently republish the same indicator with different capitalisation or surrounding whitespace. Before the indicators can be loaded into a blocklist, the combined feed must be cleaned and deduplicated.

Write a function that accepts an array of raw indicator strings drawn from multiple feeds and returns a new array with each indicator trimmed of leading/trailing whitespace, lower-cased, and de-duplicated, preserving the order in which each distinct indicator first appeared.

Input: [" Evil.com", "evil.com ", "GOOD.org", "good.org"]
Output: ["evil.com", "good.org"]

Input: ["1.2.3.4", "1.2.3.4", "1.2.3.5"]
Output: ["1.2.3.4", "1.2.3.5"]

```
import java.util.LinkedHashSet;
```

```
public class ThreatFeedDeduplicator
{
    public static String[] deduplicate(String[] rawIndicators)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

13.7 Mean Time to Detect (MTTD) Calculator

MTTD (Mean Time to Detect) is a standard SOC metric: the average delay, across a set of incidents, between when an event actually occurred and when it was first detected. Lowering MTTD is a primary goal of incident response tuning.

Write a function that accepts two equal-length arrays of Unix epoch timestamps — eventTimes and detectionTimes, where detectionTimes[i] is always greater than or equal to eventTimes[i] — and returns the mean detection delay in seconds, rounded down to the nearest whole second.

Input: eventTimes = [1000, 2000, 3000], detectionTimes = [1030, 2090, 3060]
Output: 60

Input: eventTimes = [500], detectionTimes = [500]
Output: 0

```
public class MttdCalculator
{
    public static long meanDetectionSeconds(long[] eventTimes, long[] detectionTimes)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
```

```
}
```

13.8 Kill Chain Stage Classifier

Mapping an alert to a stage of the Cyber Kill Chain helps an analyst judge how far an intrusion has progressed. This exercise uses a simplified, four-stage version of the model.

Write a function that accepts an alert type string and returns its Kill Chain stage: "PHISHING_EMAIL" or "MALICIOUS_ATTACHMENT" map to "DELIVERY"; "MALWARE_DROPPED" or "PRIVILEGE_ESCALATION" map to "INSTALLATION"; "C2_BEACON" or "LATERAL_MOVEMENT" map to "COMMAND_AND_CONTROL"; "DATA_EXFIL" maps to "ACTIONS_ON_OBJECTIVES". Any other alert type should return "UNKNOWN".

Input: "C2_BEACON"
Output: "COMMAND_AND_CONTROL"

Input: "DATA_EXFIL"
Output: "ACTIONS_ON_OBJECTIVES"

Input: "USB_INSERTED"
Output: "UNKNOWN"

```
public class KillChainClassifier
{
    public static String classify(String alertType)
    {
        # Write code here
    }
    public static void main(String[] args)
    {
        # Write code here
    }
}
```

Try:

1. **Incident Correlation Engine:** A SOC receives a flat stream of alerts, each with a source IP and a timestamp. Alerts should be grouped into the same incident if they share a source IP and occur within 300 seconds of another alert already in that group (chained, not just from the first alert). Your task is to complete the function groupIntoIncidents(), which takes an array of alerts (sourceIp, timestamp) and returns a list of incident groups, each containing the indices of the alerts that belong together.

Input: alerts = [("10.0.0.5", 100), ("10.0.0.5", 250), ("10.0.0.5", 900), ("10.0.0.9", 105)]
Output: [[0, 1], [2], [3]]

2. **Weighted Threat Score Aggregator:** Different threat intel feeds have different reliability. Your task is to complete the function aggregateScore(), which takes an array of (indicatorScore, feedConfidence) pairs — where indicatorScore is 0–100 and feedConfidence is 0.0–1.0 — for the same indicator reported by multiple feeds, and returns a single overall risk score as the confidence-weighted average, rounded to the nearest whole number.

Input: reports = [(90, 1.0), (40, 0.5)]
Output: 73

Final Investigation Project & Report Submission

This capstone exercise brings together the tools and techniques built across Exercises 1–13. Working individually or in pairs, students are given a small synthetic "case package" (fabricated log files, a couple of recovered files, and a suspect notebook excerpt — all synthetic, non-sensitive sample data) and must reuse or extend their earlier programs to reconstruct what happened, then present their findings as a formal written forensic report. There is no single "correct" output here — you are graded on the soundness of your method, the correctness of your code, and the clarity of your report.

13.9 Case Scenario

A workstation at a small company ("SynthCorp") is suspected of being used to exfiltrate a customer file shortly before an employee's exit interview. You have been given a disk image extract containing: a Windows authentication log, a suspect's personal notebook (partially enciphered), a handful of recovered files with altered extensions, and a hash manifest taken at the time of seizure. Your task is to determine whether unauthorized access occurred, build a timeline of the relevant events, and identify what — if anything — left the building.

13.10 Investigation Tasks

Using the tools you built in Exercises 1–10 (or new code where needed), work through the following. Document your commands, inputs, and outputs as you go — this becomes your evidence log.

1. **Verify evidence integrity.** Recompute the SHA-256 hash of every file in the case package and compare it against the supplied hash manifest. Flag any mismatch before proceeding.

2. **Identify true file types.** Inspect the file signature (magic number) of each recovered file and note any file whose extension does not match its actual type.

3. **Decode the notebook.** The suspect's notebook excerpt is obfuscated with a Caesar shift. Recover the shift value by inspection and decode the note in full.

4. **Analyse the authentication log.** Determine how many failed login attempts preceded each successful login in the log, and flag any account with an unusually high count before success.

5. **Build the timeline.** Convert every Unix timestamp found across the log and file metadata into readable UTC date-times and lay them out in chronological order.

6. **Validate network artefacts.** Check any IP addresses and MAC addresses recovered from the log for well-formedness, and normalise all MAC addresses to a single format for comparison.

7. **Assess recovered credentials.** Score the strength of any plaintext or Base64-encoded credentials found in the package, and note whether weak credentials could explain the access.

13.11 Build the Investigation Toolkit

Consolidate the functions you have already written into a single toolkit class so the whole investigation can be re-run from one place. Wire each method to the corresponding exercise you completed earlier; you do not need to rewrite the underlying logic, only integrate it.

```
public class InvestigationToolkit
{
    public static String decodeNote(String message, int shift)
    {
        # Write code here – reuse Exercise 1.1
    }
```

```
    public static String verifyIntegrity(byte[] data, String expectedHash)
    {
        # Write code here – reuse Exercise 1.2
    }
    public static String identifyFileType(int[] header)
    {
        # Write code here – reuse Exercise 1.3
    }
    public static String buildTimelineEntry(long epochSeconds)
    {
        # Write code here – reuse Exercise 1.5
    }
    public static int countPrecedingFailures(String[] logs, String username)
    {
        # Write code here – reuse Exercise 1.9
    }
    public static void main(String[] args)
    {
        # Write code here – run the full case package through the steps above
    }
}
```

13.12 Final Report Requirements

Submit a single written report (PDF or Word, 4–8 pages) alongside your code. The report must contain the following sections:

1. **Executive Summary.** One paragraph stating what you were asked to investigate and your overall conclusion.

2. **Methodology.** Which tools you used at each step and why, referencing the exercises they came from.

3. **Evidence Integrity Log.** The hash verification results for every file in the case package.

4. **Timeline of Events.** A single chronological table built from every timestamp you recovered.

5. **Findings.** What you concluded happened, with each claim tied back to a specific piece of evidence.

6. **Recommendations.** Practical steps SynthCorp could take to prevent a similar incident.

7. **Appendix.** Your source code and the raw command output you generated while working through 14.2.

13.13 Submission Checklist & Grading Rubric

Submit your InvestigationToolkit source file and your report through the course portal by the posted deadline.

Component	Weight	What is assessed
Evidence integrity & file identification	15%	Correct hashing and file-signature results
Decoding & log analysis	20%	Correct Caesar decode and failed-login analysis
Timeline accuracy	15%	All timestamps converted and correctly ordered
Toolkit code quality	20%	Integration, correctness, and clarity of InvestigationToolkit
Report clarity & evidence linkage	20%	Findings are traceable to specific evidence
Recommendations	10%	Practical, specific, and justified by the findings

V. REFERENCE BOOKS:

1. Bill Nelson, Amelia Phillips, Christopher Steuart, **Guide to Computer Forensics and Investigations**, 7th Edition, Cengage Learning, 2022.
2. Jason T. Luttgens, Matthew Pepe, Kevin Mandia, **Incident Response & Computer Forensics**, 4th Edition, McGraw-Hill Education, 2014.
3. Eoghan Casey, Digital Evidence and Computer Crime: Forensic Science, Computers, and the Internet, 3rd Edition, Academic Press, 2011.

VI. MATERIALS ONLINE

1. Course Outline Description
2. Lab Manul