ANSWERS

A7.9 Compression, encryption and hashing

Data representation · A level · OCR H446 1.3.1, AQA 7517 4.5.6.9 · about 40 min

What this lesson is about

Run length and dictionary coding, the Caesar and Vernam ciphers, symmetric and asymmetric encryption, and hashing.

- Lossy and lossless
- Run length encoding
- Dictionary coding
- Breaking the Caesar cipher
- The Vernam cipher
- Symmetric and asymmetric encryption
- Hashing

Questions

6 marks in all

1. A program's source code is compressed to send it by email. Which kind of compression must be used, and why? [1 mark]

- ☐ **A** Lossless, because the exact original must be rebuilt
- ☐ **B** Lossy, because it gives smaller files
- ☐ **C** Lossy, because nobody reads every character
- ☐ **D** Either, because code is text

> **Answer:** A. Losing even one character could change what the program does, so only lossless compression is acceptable.

2. What does this program print? [1 mark]

```
row = "AAAABBBCCD"
out = ""
i = 0
while i < len(row):
    j = i
    while j < len(row) and row[j] == row[i]:
        j = j + 1
    out = out + str(j - i) + row[i]
    i = j
print(out)
```

> **Answer:**
>
> ```
> 4A3B2C1D
> ```
>
> Run length encoding replaces each run with its length and the repeated value.

3. Which conditions must a Vernam cipher key meet for perfect security? [1 mark]

Tick every answer that is true.

- ☐ **A** It is truly random
- ☐ **B** It is at least as long as the plaintext
- ☐ **C** It is used only once
- ☐ **D** It is a prime number

> **Answer:** A, B, C. It must also be kept secret. There is no requirement for the key to be prime.

4. What does it mean for a cipher to be computationally secure? [1 mark]

- ☐ **A** It could be broken in theory, but not in a useful amount of time with current computers
- ☐ **B** It can never be broken, whatever the computing power
- ☐ **C** It uses a computer to generate a random key
- ☐ **D** It is secure only when run on a computer

> **Answer:** A. Only the one-time pad is perfectly secure. Other ciphers rely on breaking them taking far too long.

5. Why are passwords stored as hashes rather than encrypted? [1 mark]

- ☐ **A** A hash is one-way, so a stolen table does not reveal the passwords
- ☐ **B** A hash is shorter to type
- ☐ **C** A hash can be decrypted by the server when needed
- ☐ **D** A hash never has collisions

> **Answer:** A. The server hashes what the user types and compares hashes. Encrypted passwords could be decrypted by anyone who found the key.

6. Alice wants to send Bob a message using asymmetric encryption. Which key does she encrypt it with? [1 mark]

- ☐ **A** Bob's public key
- ☐ **B** Bob's private key
- ☐ **C** Alice's private key
- ☐ **D** A key they have both kept secret

> **Answer:** A. Only Bob's private key can decrypt what his public key encrypted, and he never shares that key.

The task: a compressed route

`route` is a string of words separated by single spaces: `FWD <cm>` and `LEFT <degrees>` commands. 1. Write `encode(text)`. The parameter `text` is a string of words separated by single spaces. It returns a pair: a **list** of the different words in the order they first appear (the dictionary), and a **string** of each word's index in that list, separated by single spaces. 2. Write `decode(dictionary, encoded)`, which takes those two values and returns the original string. 3. Print `dictionary: <the words separated by spaces>`, then `encoded: <the encoded string>`, then `original: <n> characters, encoded: <m> characters` using `len` of `route` and of the encoded string, then `decoded matches: <True or False>` comparing the decoded string with `route`. 4. Drive the **decoded** route: `FWD n` is `forward(50, distance=n)` and `LEFT n` is `turn_left(angle=n)`, where `n` is the number after the word. Build the encoded string with your function; do not type it in.

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

route = "FWD 20 LEFT 90 FWD 20 LEFT 90 FWD 20 LEFT 90 FWD 20 LEFT 90"

def encode(text):
    dictionary = []
    codes = []
    return dictionary, " ".join(codes)

def decode(dictionary, encoded):
    return ""
```

The hint students can ask for: Walk the words in order. The first time you meet a word, add it to the end of the dictionary; every time, write down its position in the dictionary. Decoding looks each number up again. Drive from the decoded words, not the original string.

A solution

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

route = "FWD 20 LEFT 90 FWD 20 LEFT 90 FWD 20 LEFT 90 FWD 20 LEFT 90"

def encode(text):
    dictionary = []
    codes = []
    for word in text.split(" "):
        if word not in dictionary:
            dictionary.append(word)
        codes.append(str(dictionary.index(word)))
    return dictionary, " ".join(codes)

def decode(dictionary, encoded):
    return " ".join(dictionary[int(code)] for code in encoded.split(" "))

dictionary, encoded = encode(route)
print("dictionary:", " ".join(dictionary))
print("encoded:", encoded)
print("original:", len(route), "characters, encoded:", len(encoded), "characters")
plain = decode(dictionary, encoded)
```

```
print("decoded matches:", plain == route)

words = plain.split(" ")
for i in range(0, len(words), 2):
    if words[i] == "FWD":
        forward(50, distance=int(words[i + 1]))
    elif words[i] == "LEFT":
        turn_left(angle=int(words[i + 1]))
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.