Past-paper walk-through

Compression ▪ 1.3

eXercise

Questions in this set

- 9618/12 N25 Q6 ▪ 6 marks
- 9618/43 N25 Q2 ▪ 24 marks
- 9618/13 N24 Q6 ▪ 10 marks
- 9618/12 N24 Q7 ▪ 7 marks

Reading the mark scheme

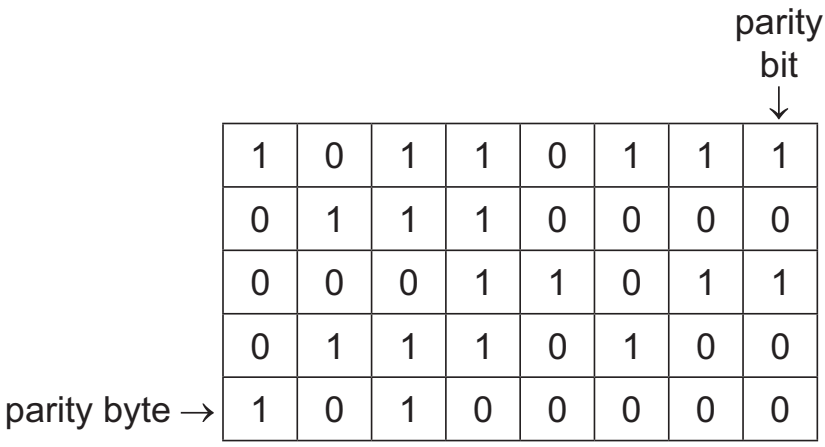
- **B1** a mark that stands on its own – the point is either made or not.
- **M1** a *method* mark: the right approach, even if the number is wrong.
- **A1** an *answer* mark, and it usually needs its M mark first.
- **C1** a working mark on the way to the answer.
- // separates answers the examiner accepts equally.
- **ecf** = error carried forward: a later part is marked on your own earlier answer.
- **owtte** = "or words to that effect" – the wording need not match.

Question 6

9618/12 N25 ▪ Q6 ▪ 6 marks

(a) A sound file is compressed by reducing the sampling rate.
State whether this is lossless or lossy compression. Justify your choice.
Justification **[1]**

Question 6


parity bit
parity byte →
1 0 1 1 0 1 1 1
0 1 1 1 0 0 0 0
0 0 0 1 1 0 1 1
0 1 1 1 0 1 0 0
1 0 1 0 0 0 0 0

Question 6

Word	Denary value
Computing	55
Science	56
Computers	57
are	58
Brilliant!	59
is	60
Fun!	61
Amazing!	62

Question 6

Binary value	00111000	00111100	00111110
Word			

Question 6

9618/12 N25 ▪ Q6 ▪ 6 marks

(b) The following table shows some words and corresponding denary values.

Word	Denary value
Computing	55
Science	56
Computers	57
are	58
Brilliant!	59
is	60
Fun!	61
Amazing!	62

Question 6

The following table shows three bytes of data that have been received.
Use the table to find the corresponding words from the binary values received.

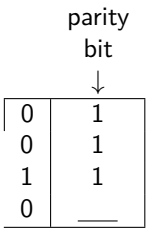
Binary value	00111000	00111100	00111110
Word			

[1]

Question 6

9618/12 N25 ▪ Q6 ▪ 6 marks

(c)(i) Complete the byte by writing the missing parity bit:



Question 6

[1]

(c)(ii) The computer also uses parity block check. The parity block check uses even parity. Computer A transmits four bytes of data to computer B, followed by a parity byte. Computer B receives the following sequence of bytes.

								parity bit ↓
	1	0	1	1	0	1	1	1
	0	1	1	1	0	0	0	0
	0	0	0	1	1	0	1	1
	0	1	1	1	0	1	0	0
parity byte →	1	0	1	0	0	0	0	0

Question 6

Following transmission, one of the four bytes of data has an error in one of the bits. Circle the bit that has been altered during the data transfer. **[1]**

Question 6

9618/12 N25 ▪ Q6 ▪ 6 marks

(d) A bitmap image has a resolution of 1000 pixels wide by 2000 pixels high. The colour depth is 16 bits.
Calculate an estimate of the file size in megabytes.
Show your working.
File size ___ megabytes **[2]**

Solution 6

(a)

Mark scheme

- 1 mark for correct answer
- Type of compression: lossy
- Justification: There will be fewer samples per second, so data will be permanently lost
- // There will be fewer samples per second, so the original sound cannot be re-created

Solution 6

(b) Mark scheme

- 1 mark for correct words in the correct order
- Binary Value
- 0011 1000
- 0011 1100
- 0011 1110
- Word
- Science is
- Amazing!

Solution 6

- 1
- October/November
- 2025

Solution 6

(c)(i) Mark scheme

- 1 mark for correct answer parity bit
- ↓
- 0
- 1
- 0
- 1
- 1
- 1

Solution 6

- 0
- 0

Solution 6

(c)(ii) Mark scheme

- 1 mark for correct answer parity bit
- ↓
- 1
- 0
- 1
- 1
- 0
- 1

Solution 6

- 1
- 1
- 0
- 1
- 1
- 1
- 0
- 0
- 0
- 0

Solution 6

- 0
- 0
- 0
- 1
- 1
- 0
- 1
- 1
- 0
- 1

Solution 6

- 1
- 1
- 0
- 1
- 0
- 0 parity byte$\longrightarrow$
- 1
- 0
- 1
- 0

Solution 6

- 0
- 0
- 0
- 0

1

Solution 6

(d) Worked steps

File size is the number of pixels times the bits per pixel, converted to the unit asked for. Set the whole calculation out on one line before evaluating it — one mark is for the working.

```
(1000 * 2000 * 16) / (8 * 1000 * 1000)
```

Solution 6

- Pixels: $1000 \times 2000 = 2\,000\,000$
- Bits: $2\,000\,000 \times 16 = 32\,000\,000$
- Bytes: $\div 8 = 4\,000\,000$
- Megabytes: $\div 10^6 = \mathbf{4}$ MB

The shortcut the mark scheme also accepts: 16 bits is 2 bytes, so $1000 \times 2000 \times 2 \div 10^6$ reaches the same answer with one division fewer.

Divide by 8 **once**, at the point where bits become bytes — folding it in early is where the factor-of-eight errors come from. And answer in the unit the question names: the working ends in bytes and megabytes were asked for.

Solution 6

The word “estimate” is in the question because a real bitmap also carries a header, so the true file is slightly larger. Nothing needs adding for it; it is the reason the answer is an estimate rather than an exact size.

Mark scheme

- 1 mark for the working
- 1 mark for the correct answer
- Working:
- (1000 * 2000 * 16) / (8 * 1000 * 1000)
- // (1000 * 2000 * 2 / (1000 * 1000)
- Answer: 4 megabytes

Question 2

9618/43 N25 ▪ Q2 ▪ 24 marks

A program reads string data from a text file into a linear queue and then compresses the data.
The queue is created using the global 1D array `Queue`. The queue can store up to 100 elements. Each element is initialised to the empty string ""
The queue has two pointers that are global to the program:

- `QueueHead` that points to the index of the first element in the queue, initialised to -1
- `QueueTail` that points to the index of the last element in the queue, initialised to -1

Question 2

The global variable `NumberItems` stores the number of elements stored in the queue, initialised to 0

(a) Write program code to declare and initialise `Queue`, `QueueHead`, `QueueTail` and `NumberItems`

Save your program as **Question2_N25**.

Copy and paste the program code into part **2(a)** in the evidence document. **[2]**

Question 2

9618/43 N25 ▪ Q2 ▪ 24 marks

A program reads string data from a text file into a linear queue and then compresses the data. The queue is created using the global 1D array `Queue`. The queue can store up to 100 elements. Each element is initialised to the empty string `""`
The queue has two pointers that are global to the program:

- `QueueHead` that points to the index of the first element in the queue, initialised to -1
- `QueueTail` that points to the index of the last element in the queue, initialised to -1

Question 2

The global variable `NumberItems` stores the number of elements stored in the queue, initialised to 0

(b) The function `Enqueue()` takes a string as a parameter. The function checks if the queue is full, and returns Boolean `FALSE` if the queue is full.

If the queue is not full, the parameter is inserted into the next position in `Queue`. The function updates the appropriate pointer(s), updates `NumberItems` and then returns Boolean `TRUE`

Write program code for `Enqueue()`

Save your program.

Copy and paste the program code into part **2(b)** in the evidence document. **[5]**

Question 2

9618/43 N25 ▪ Q2 ▪ 24 marks

A program reads string data from a text file into a linear queue and then compresses the data. The queue is created using the global 1D array `Queue`. The queue can store up to 100 elements. Each element is initialised to the empty string `""`
The queue has two pointers that are global to the program:

- `QueueHead` that points to the index of the first element in the queue, initialised to -1
- `QueueTail` that points to the index of the last element in the queue, initialised to -1

Question 2

The global variable `NumberItems` stores the number of elements stored in the queue, initialised to 0

(c) The function `Dequeue()` returns the string `"False"` if the queue is empty.
If the queue is not empty, the function returns the next element in the queue. The function updates the appropriate pointer(s) and updates `NumberItems`
Write program code for `Dequeue()`
Save your program.
Copy and paste the program code into part **2(c)** in the evidence document. **[3]**

Question 2

9618/43 N25 ▪ Q2 ▪ 24 marks

A program reads string data from a text file into a linear queue and then compresses the data. The queue is created using the global 1D array `Queue`. The queue can store up to 100 elements. Each element is initialised to the empty string `""`
The queue has two pointers that are global to the program:

- `QueueHead` that points to the index of the first element in the queue, initialised to -1
- `QueueTail` that points to the index of the last element in the queue, initialised to -1

The global variable `NumberItems` stores the number of elements stored in the queue, initialised to 0

Question 2

(d) The text file `BinaryData.txt` stores individual binary digits, '1' and '0'. Each digit is on a new line. For example, the first line in the text file stores '1', the second line stores '1'.
The procedure `ReadData()` reads in each line from the text file `BinaryData.txt` and inserts it into the queue using the appropriate method.
The procedure needs to work for a text file with any number of lines up to a maximum of 100.
Write program code for `ReadData()`

Save your program.
Copy and paste the program code into part **2(d)** in the evidence document.

Question 2

[5]

Question 2

9618/43 N25 ▪ Q2 ▪ 24 marks

A program reads string data from a text file into a linear queue and then compresses the data. The queue is created using the global 1D array `Queue`. The queue can store up to 100 elements. Each element is initialised to the empty string `""`
The queue has two pointers that are global to the program:

- `QueueHead` that points to the index of the first element in the queue, initialised to -1
- `QueueTail` that points to the index of the last element in the queue, initialised to -1

The global variable `NumberItems` stores the number of elements stored in the queue, initialised to 0

Question 2

(e) The string data in the text file is compressed.
The compression algorithm counts the number of times each binary digit appears consecutively, then stores the binary digit followed by the number of times it appears.
The algorithm stores the compressed data in a single string.
For example, if the text file contains the data:
1 1 0 0 0 1 1 1
The compression algorithm will create the string "120313" because there are two '1' digits, followed by three '0' digits, followed by three '1' digits.

Question 2

The procedure `Compress()` uses `Dequeue()` to remove each element from the queue in turn. The procedure then compresses the data following the compression algorithm described. The new compressed string is stored in the global variable `NewString`
You can assume that one binary digit will never appear more than nine times consecutively in the sequence.
You can assume that there will always be at least one item in the queue.
Write program code for `Compress()`

Save your program. Copy and paste the program code into part **2(e)** in the evidence document.

Question 2

[6]

Question 2

9618/43 N25 ▪ Q2 ▪ 24 marks

A program reads string data from a text file into a linear queue and then compresses the data. The queue is created using the global 1D array `Queue`. The queue can store up to 100 elements. Each element is initialised to the empty string `""`
The queue has two pointers that are global to the program:

- `QueueHead` that points to the index of the first element in the queue, initialised to -1
- `QueueTail` that points to the index of the last element in the queue, initialised to -1

The global variable `NumberItems` stores the number of elements stored in the queue, initialised to 0

(f)(i) Write program code for the main program to:

- call `ReadData()`
- call `Compress()`

Question 2

- output the content of the compressed string.

Save your program.
Copy and paste the program code into part **2(f)(i)** in the evidence document. **[2]**
(f)(ii) Test your program.
Take a screenshot of the output(s).
Save your program.
Copy and paste the screenshot(s) into part **2(f)(ii)** in the evidence document. **[1]**

Solution 2

(a) Worked steps

Four things are declared, and each has a starting value the rest of the question depends on.

```
DECLARE Queue : ARRAY[0:99] OF STRING
DECLARE QueueHead, QueueTail, NumberItems : INTEGER

FOR Index <- 0 TO 99
    Queue[Index] <- ""
NEXT Index

QueueHead <- -1
QueueTail <- -1
NumberItems <- 0
```

Solution 2

- The array is **global**, 100 elements, of type STRING, and every element starts as the empty string — that is one mark, and the loop is what earns it.
- The pointers start at **−1**, not 0, because 0 is a valid index: -1 is the value that means *this queue is empty*. `NumberItems` starts at 0.

Solution 2

If your language indexes from 0, declare 100 elements as 0 to 99; if from 1, use 1 to 100 and keep the rest of the question consistent with that choice. **Mark scheme**

- 1 mark each
- ▪ Queue declared as a (global) 1D array of 100 string elements all initialised to ""
- ▪ QueueHead and QueueTail initialised with –1, NumberItems initialised to 0
- Example program code
- Java public static String[] Queue = new String[100];
- public static Integer QueueHead;
- public static Integer QueueTail;
- public static Integer NumberItems;
- for(int x = 0; x < 100; x++){
- Queue[x] = "";
- }
- QueueHead = -1;
- QueueTail = -1;

Solution 2

(b) Worked steps

Enqueue has three jobs in a fixed order: refuse if full, store, then update the bookkeeping.

```
FUNCTION Enqueue(TheData : STRING) RETURNS BOOLEAN
    IF QueueTail >= 99 THEN
        RETURN FALSE
    ENDIF
    Queue[QueueTail + 1] <- TheData
    QueueTail <- QueueTail + 1
    NumberItems <- NumberItems + 1
    IF QueueHead = -1 THEN
        QueueHead <- 0
    ENDIF
    RETURN TRUE
ENDFUNCTION
```

Solution 2

The five marks map to five separate things, so check for each one:

1. Function header with one string parameter, the full test, and `RETURN FALSE`.
2. Storing the parameter at `QueueTail + 1`.
3. Incrementing **both** `QueueTail` and `NumberItems`.
4. The **first-element case**: when the queue was empty, `QueueHead` is still -1 and must become 0, or the first item can never be dequeued.
5. `RETURN TRUE` on every path where the item was inserted.

Solution 2

That fourth mark is the one most often lost. It is the only place where the head pointer moves during an insertion, and it is easy to overlook because the queue works for the second item onward without it. **Mark scheme**

- 1 mark each
- ▪ Function header (and end where appropriate) taking one (string) parameter and checking full and returning FALSE
- ▪ (otherwise) Storing parameter in QueueTail + 1
- ▪ Incrementing QueueTail and NumberItems
- ▪ (Dealing with first element) If QueueHead = –1 store 0 in QueueHead / increment QueueHead
- ▪ Return TRUE in all cases when inserted
- Example program code.
- Java public static Boolean Enqueue(String TheData){ if(QueueHead == -1){
- Queue[0] = TheData;
- QueueHead = 0;
- QueueTail = 0;
- NumberItems = 1;

Solution 2

(c) Worked steps

Dequeue is the mirror image, and the empty test is what makes it safe.

```
FUNCTION Dequeue() RETURNS STRING
    DECLARE ReturnValue : STRING
    IF NumberItems = 0 THEN
        RETURN "False"
    ENDIF
    ReturnValue <- Queue[QueueHead]
    QueueHead <- QueueHead + 1
    NumberItems <- NumberItems - 1
    RETURN ReturnValue
ENDFUNCTION
```

Solution 2

- The empty test may be written as `NumberItems = 0` or as `QueueHead > QueueTail`; both are accepted, and both are true of an empty queue.
- **Read the element before moving the pointer.** Incrementing `QueueHead` first returns the wrong item, and the mark for returning `Queue[QueueHead]` goes with it.
- `QueueHead` goes **up** and `NumberItems` goes **down** — the two move in opposite directions, which is worth checking before you leave the question.

Solution 2

The string `"False"` here is a sentinel value, not a Boolean: the function's return type is STRING, so it must return one. Returning the Boolean `FALSE` is a type error in most of the languages this paper allows. **Mark scheme**

- 1 mark each
- ▪ Function header (and end), checking if Queue is empty (NumberItems = 0 or QueueHead > QueueTail) and returning "False"
- ▪ (otherwise) returning Queue[QueueHead]
- ▪ Incrementing QueueHead, decrementing NumberItems
- Example program code
- Java public static String Dequeue(){
- String ReturnValue;
- if(NumberItems == 0){ return "False";
- }else{
- ReturnValue = Queue[QueueHead];
- QueueHead++;
- NumberItems--;

Solution 2

(d) Worked steps

`ReadData()` is a procedure — it returns nothing, it fills the queue as a side effect. Five marks, five separate things, so build it in five visible pieces.

```
PROCEDURE ReadData()
    DECLARE Line : STRING
    DECLARE Ok   : BOOLEAN
    TRY
        OPENFILE "BinaryData.txt" FOR READ
        WHILE NOT EOF("BinaryData.txt")
            READFILE "BinaryData.txt", Line
            Ok <- Enqueue(Line)
        ENDWHILE
        CLOSEFILE "BinaryData.txt"
    EXCEPT
        OUTPUT "The file could not be read"
    ENDTRY
ENDPROCEDURE
```

Solution 2

1. Procedure header and end.
2. The file **opened and closed**, with the close in the right place — after the loop, inside the successful path.
3. A loop **until end of file**, one line per pass.
4. Each value **read into a variable** inside that loop.
5. `Enqueue()` called with the value, **and its return value stored or used**.

Solution 2

Two details carry the marks people miss. `Enqueue()` is a **function**: it returns FALSE when the queue is full, so assigning its result — and ideally stopping when it is FALSE — is what the fifth bullet asks for. And the exception handling must wrap **all** the file access, not just the `OPENFILE`; a file that vanishes mid-read fails on the `READFILE`.

Mark scheme

- 1 mark each to max 5
- ▪ Procedure header (and end where appropriate)
- ▪ Opening the file and closing the file (in appropriate place)
- ▪ Looping until EOF // looping through each line …
- ▪ … read in each value …
- ▪ … calling Enqueue() with read in value and store/use return value
- ▪ Exception handling with try except and appropriate output with all file access within try
- Example program code
- Java public static void ReadData(){
- Boolean ReturnValue;
- Boolean FinishLoop = false;
- try{

Solution 2

(e) Worked steps

Run-length encoding: walk the queue, count how many times each value repeats, and append the value followed by its count.

```
PROCEDURE Compress()
    DECLARE Current, NextValue : STRING
    DECLARE Count : INTEGER
    NewString <- ""
    Current <- Dequeue()
    WHILE Current <> "False"
        Count <- 1
        NextValue <- Dequeue()
        WHILE NextValue = Current
            Count <- Count + 1
            NextValue <- Dequeue()
        ENDWHILE
        NewString <- NewString & Current & NUM_TO_STR(Count)
        Current <- NextValue
    ENDWHILE
ENDPROCEDURE
```

Solution 2

Six marks, and they map to the six things this has to do:

1. Procedure header and end, with the finished string kept in the **global** variable the next part will output.
2. `Dequeue()` called and its return value **stored**.
3. Called **repeatedly until it returns `"False"`** — that sentinel is how the empty queue announces itself, which is why `Dequeue` returns a STRING rather than a Boolean.
4. Each new value **compared with the previous** one.
5. A **counter** of consecutive matches, reset to 1 at the start of each run.
6. The digit and its count **appended** to the string — `NewString <- NewString & …`, never `NewString <- …`, or every run overwrites the last.

Solution 2

The subtle part is the value that ends a run: it has already been dequeued, so it must become the `Current` of the next run rather than being read again. Dequeuing it twice loses one item from every run boundary. **Mark scheme**

- 1 mark each
- ▪ Procedure header (and end where appropriate), storing final string compressed data in global variable
- ▪ Call Dequeue() and storing return value ...
- ▪ ... repeatedly until the return value == "False"
- ▪ Comparing new value with previous ...
- ▪ ... maintaining counter of number of occurrences ...
- ▪ ... appending digit and number of occurrences to a string without overriding
- Example program code
- Java public static void Compress(){
- String First = Dequeue();
- String NewLine = "";
- Integer Count;

Solution 2

(f)(i) Worked steps

The main program wires the two procedures together in order.

```
NewString <- ""
// initialise Queue, QueueHead, QueueTail and NumberItems as in part (a)
CALL ReadData()
CALL Compress()
OUTPUT NewString
```

Solution 2

Two marks: **calling `ReadData()` and then `Compress()`**, and **outputting the compressed string**.

Solution 2

The order is not arbitrary — `Compress()` empties the queue that `ReadData()` fills, so reversing them compresses nothing and outputs an empty string. And the globals must be initialised before either runs, or `QueueHead` still holds whatever the last run left in it. **Mark scheme**

- 1 mark each
- ▪ Calling ReadData() then Compress()
- ▪ Outputting compressed string
- Example program code
- Java public static void main(String args[]){
- NewString = "";
- for(int x = 0; x < 100; x++){
- Queue[x] = "";
- }
- QueueHead = -1;
- QueueTail = -1;
- NumberItems = 0;

Solution 2

(f)(ii)

Mark scheme

- 1 mark for screenshot showing output
- Example

Question 6

9618/13 N24 ▪ Q6 ▪ 10 marks

6 A computer system stores text, images and sound.

(a) A character set is used to represent characters in a computer.

Identify **and** describe **one** character set.

Question 6

Question 6

Question 6

[2]

(b) The colour of each pixel in a bitmapped image is represented by 8 bits.

(i) State the largest number of different colours that can be represented by 8 bits.

Question 6

(ii) State **one** drawback of increasing the number of bits that represents each pixel in the bitmap image.

Question 6

(iii) A bitmap image can be compressed using lossy compression.

Explain the reasons why lossy compression is often suitable for a bitmap image.

Question 6

(c) **(i)** Explain how an analogue sound wave is converted into digital data.

Question 6

(ii) Describe **one** method of compressing a sound file using lossy compression.

Question 6

Solution 6

(a) Worked steps

Two marks: one for **naming** a character set, one for a description that **matches** the one you named. So decide first, then describe.

- **ASCII** — 7 or 8 bits per character, representing 128 or 256 characters, enough for the Latin alphabet, digits and punctuation.
- **Unicode** — 8, 16 or 32 bits per character, representing 256 to over 65 536 characters, enough for the characters of every language.

Solution 6

Pick one and stay with it. Naming ASCII and then describing 65 536 characters scores the identification mark only, and that pairing is exactly what "matching description" is testing.

The description has to be **quantitative**: give the bits per character or the number of characters. "It represents characters as binary" is true of both and distinguishes neither. **Mark scheme**

- 1 mark for identification 1 mark for matching description
- e.g.
- ▪
- ASCII

Solution 6

- ▪
- 7/8 bits per character // represents 128/256 characters // represents all
- characters from Latin alphabet
- ▪
- UNICODE
- ▪
- 8/16/32 bits per character // represents 256/65536+ characters //
- represents all characters in all languages
- 2

Solution 6

(b)(i)

Worked steps

Colour depth is the number of bits per pixel, and the number of colours is how many different values those bits can take.

- 8 bits per pixel, so $2^8 = \mathbf{256}$ colours.

The rule generalises: n bits give 2^n colours, so 1 bit is 2 colours (black and white), and 24-bit "true colour" is 2^{24}, about 16.7 million.

Mark scheme

- 1 mark for:
- 256 // 28
- 1

Solution 6

(b)(ii)

Mark scheme

- 1 mark for:
- Increased file size
- 1

Solution 6

(b)(iii) Worked steps

Two marks from:

- The change is often **not noticeable** — the data removed is usually beyond what the eye detects, for example small differences in shade or fine detail.
- Lossy produces a **much larger reduction** in file size than lossless does.

Both halves are needed for a full answer: lossy is worth using because it saves a great deal *and* because what it discards is not missed. Either point alone is one mark. **Mark scheme**

- 1 mark for each bullet point (max 2)

Solution 6

- e.g.
- ▪
- The change may not be noticeable // Data removed is usually not noticed
- by the human eye
- ▪
- ... for example, changes in shade/detail
- ▪
- It produces a larger decrease in file size compared to lossless // Lossy
- decreases file size considerably
- 2

Solution 6

(c)(i) Worked steps

Describe **sampling**, in two steps.

- The **magnitude of the analogue sound wave is measured** a set number of times each second, at regular intervals.
- Each measurement is **given a binary value and stored in sequence**.

The two marks correspond to the two things a sampling rate and a sampling resolution control: how often you measure, and how finely each measurement is recorded. An answer that says only "the sound is converted to binary" describes the outcome without the method. **Mark scheme**

Solution 6

- 1 mark for each bullet point (max 2)
- ▪
- Value/magnitude/size of the analogue sound wave is measured a set
- number of times each second/time / at set intervals
- ▪
- Each sample/reading/measurement is given the binary number and
- stored in sequence
- 2
- 9618/13
- October/November

Solution 6

- 2024

Solution 6

(c)(ii) Worked steps

Each mark is a **method plus its explanation**, so give both halves of each point.

- **Decrease the sample rate** — fewer measurements are stored per second, so fewer bits per second of sound.
- **Decrease the sample resolution** — each measurement is stored in fewer bits.
- **Remove sounds outside the range of human hearing** — frequencies nobody can hear need not be stored at all.

Solution 6

A bare list of three methods earns three marks at most out of six. The pattern the mark scheme uses throughout this paper is *point, then expansion*, and the expansion is where the understanding shows.

Note the trade-off worth mentioning: the first two reduce quality — a lower rate loses high frequencies, a lower resolution adds quantisation noise — while the third is lossy in name only, since what it discards was never audible. **Mark scheme**

- 1 mark for each correct point and 1 mark for matching expansion
- e.g.
- ▪
- Decrease sample rate

Solution 6

- ▪
- ... fewer samples/readings/measurements stored per second // fewer bits
- per second stored
- ▪
- Decrease sample resolution
- ▪
- ... fewer bits per sample/reading/measurement // each sample has
- fewer bits
- ▪
- Sound outside of set/human hearing range is removed

Solution 6

- ▪
- ... fewer measurements are stored / decreases the number of possible
- binary values so fewer bits are stored
- 2

Question 7

9618/12 N24 ▪ Q7 ▪ 7 marks

A student takes a photograph of a science experiment.

(a)(i) Define the following bitmap terms. **[2]**

(a)(ii) Explain why changing the image resolution will affect the image quality and file size. **[2]**

(a)(iii) Identify **one** lossless method of compressing an image. **[1]**

(b) The student draws a picture on paper that is scanned into the computer and saved as a vector graphic.

Define the vector graphic terms property and drawing list. **[2]**

Solution 7

(a)(i)

Mark scheme

- 1 mark for each correct definition
- Colour depth:
- ▪ the number of bits used to represent a colour // the number of colours that can be represented in an image
- File header:
- ▪ stores data about the image file / metadata

Solution 7

(a)(ii) Mark scheme

- 1 mark for each correct explanation
- Image quality:
- ▪ Decreasing resolution means details within the image are lost because there are fewer pixels // Increasing resolution means the image is more detailed because there are more pixels
- File size:
- ▪ Decreasing the resolution will decrease the file size because there are fewer pixels therefore less data // Increasing the resolution will increase the file size because there are more pixels therefore more data

Solution 7

(a)(iii)

Mark scheme

- 1 mark for correct method
- For example:
- ▪ Run-Length Encoding

Solution 7

(b) Mark scheme

- 1 mark for each correct definition
- Property:
- ▪ an attribute of a drawing object // data about a shape // defines one aspect of the appearance of a drawing object
- Drawing list:
- ▪ all the drawing objects/shapes in an image // stores the commands/descriptions / mathematical equations required to draw each object
- 2

Solution 7

- October/November
- 2024