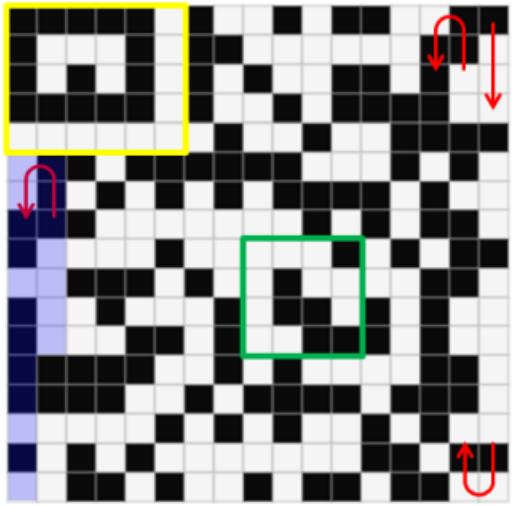
Task 3. QF Codes

Available marks: 32



Quentin Foobar has invented his own form of QR codes, called *QF codes*. Quentin stole quite a few ideas from QR, but has simplified the design. A QF code is a square bitmap with an odd side length, between 17 and 49 inclusive. It has three regions:

- an alignment box of 6 × 5 bits at the top left (see image, yellow rectangle),
- a key box of 4 × 4 bits whose top left bit is exactly in the in the centre of the QF bitmap (green), and
- encoded data filling the rest in a zig-zag formation, starting at the top right and filling down (red arrows, padding bits are blue).

Data to be encoded is printable ASCII text, but with a special mode when the entire text consists of digit characters. There is no error detection or error correction coding, which is why QF came to be known as "Quick Failure", in contrast to QR's Quick Response.

The encoding modes are

- **Text mode** (type 0): encoded data consists of the normal ASCII representation using 7 bits.
- **Decimal mode** (type 1): data is encoded in groups of 3 digits from the left. Each group is treated as a 3-digit decimal number, 14 is added and the result converted to a 10-bit binary number. If there are 2 digits left over, add 14 and convert to 7 bits; if 1 digit, add 14 and convert to 4 bits. For example, the digit string 12345 would be encoded in 17 bits as (123 + 14 = 137 = 10001001) followed by (45 + 14 = 59 = 0111011).

The *data length* (len) is the number of symbols to be encoded. The *encoded length* is the number of bits in the encoded data string.

The 16 bits in the key box comprise:

- 1 bit for the encoding mode.
- 5 bits for the mask used (see *Masking* below).
- 10 bits recording the result of the expression `3*len+3` (it expands the representation by 2 bits but improves the distribution of 1s and 0s).

The key box layout is left to right and top to bottom in a 4 × 4 pattern. Thus the mode and first 3 mask bits are on the first row.

Given the encoded length and the 30 + 16 = 46 bits reserved for the alignment and key boxes, the encoder first has to calculate the minimum size QF code that can accommodate everything. The code *capacity* is the number of data bits it can accommodate.

If necessary, the encoder pads the encoded data bit string out to the full capacity by appending alternating 7-bit A and B patterns, starting with A. A is decimal 77 (1001101) and B is decimal 42 (0101010). Truncation occurs as required, so for example if only 4 more bits are needed to reach capacity and the last pattern used was A, the first 4 bits from B are used (0101).

Masking

It's possible to assemble the whole code now, but there's a catch. By chance, the bit patterns in the encoded data could cause a scanner problems, say because they look a little like the alignment box or have large regions of the same bit value.

As you may know, using the XOR operator is a simple way to change the appearance of a bit string, and by repeating the operation you can recover the original string. The QF encoder tries all 32 possible 5-bit patterns called *masks* in turn, applying the mask to each 5-bit segment of the data to create a provisional code. Each provisional code is evaluated for safety. Whichever mask produces the safest code is used.

The safety evaluation has 3 steps, applied to the whole code structure as laid out, including alignment and key boxes. Remember to include the current mask in the key box before evaluation.

1. Score 2 penalty points for every run of 4 equal bits in either cardinal direction (left-right or up-down). Add 1 penalty point each for additional adjacent bit of the same value in the same direction. For example, a run of 7 equal bits would attract 2+3 = 5 points.
2. Score 3 penalty points for each block of 2×2 equal bits, counting overlapping blocks separately. For example, a 4×3 region of uniform colour has 6 overlapping blocks and thus 18 PPs.

3. Score *n* penalty points for uneven bit distribution, where *n* is the absolute value of the difference between the total number of 1s and 0s in the entire code. For example, A 19×19 QF code has 361 bits. If 192 are 1s and 169 are 0, score 23 PPs.

Each step accumulates penalty points, and the safest solution has the fewest total penalty points. If that's not unique, select the mask with the highest numerical value among the equal best.

Your Task

This task has two components, the QF Encoder and the QF Decoder. They may be completed by the same or different team members, and be one or two programs. Code can be shared.

For the **Encoder**, write a program that applies the above rules exactly to generate the QF code corresponding to each of several inputs, one per line (although QF codes *could* record multi-line data, only printable text is in scope for this task). The output format uses `#` for 1 and `.` for 0, and double-spaces the characters on each line of the code. The QF code pictured above is generated from the string "ProgComp Grand Final, 08/10/2021", and its textual representation is

```
# # # # # . # . . # . # # . . # #
# . . . # . # # . . . . . # # .
# . # . # . # . # . . # # . # . .
# # # # # . # . . # . # # # # . .
. . . . . . . # . . # . . # # # #
. # # . # # # # # # . . . # . # .
. # . # . # . # . # # # # . # . .
# # # . . . . . . . # . # . # # .
# . . . . # . . . . . # . # . # #
. . # # # . # . . # . . . . # # .
# . . # . . . # . # # . # . # . .
# . . . # # . # . . # # # . # . .
# # # # # . . # . # . . . . # # .
# # # # . . # . # # # # . # # # .
. . . . . # . # . # . . # . # . .
# . # . # . . . . . . . # # . # #
. . # # . # . . # . # # . # # . .
mask 5 score 143 (86, 54, 3) checksum 79
```

Following the QF code, display the selected mask in decimal and the corresponding penalty scores, then a checksum to assist the judges in confirming the correctness of the result. Calculate the checksum for each row by adding each 8-bit chunk, working from right to left. Add all the checksums and display the total modulo 256. The mask for this code is `00101`, penalty score components are 86 for runs, 54 for blocks and 3 for uneven bit counts. The first row sum is 179+244+1 = 424 and the final checksum is 4175 mod 256 = 79.

The message has 32 characters, so that's 224 data bits and 17×17 − 46 − 224 = 19 padding bits Padding bits are shown as the blue overlay in the image. They're masked of course.

Assessment, Encoder

There are two test files in the data pack, **encode1.dat** and **encode2.dat**. The judges will ask you to show your results for one or both of them. Encoding is worth 19 of the 32 marks. For full marks your implementation must handle both encoding methods, determine the correct QF size, encode the data correctly, choose the most suitable mask, position the alignment and key boxes accurately and fill in the encoded data in the right locations and order. Part marks are available, but the results must be recognisably QF codes at a minimum.

For the **Decoder**, write a program that reads a single QF code bitmap in the above format (without the mask, scores and checksum) and displays the original data. Test files are named **decode*.qf** . If it's working the output should be meaningful (look for a recognisable pattern in the digits if it's decimal mode).

Note however that the code may have been either rotated by a multiple of 90 degrees, or reflected about the horizontal or vertical axis (the alignment box is asymmetric so that helps). Although the code will be structurally and semantically correct, the mask used may not be the optimal one and the padding bits may be changed. Accordingly the image may have misleading artefacts, though it won't be completely ambiguous. Only files of the form **decode3*.qf** are modified in this way.

Assessment, Decoder

There are many test files in the data pack. The judges will ask you to show your results for a few of them. Decoding is worth 13 of the 32 marks. For full marks your implementation must handle the geometric transformations, determine the mode and size, unmask and decode the data correctly, without any spurious data at the end. Part marks are available if some QF codes are accurately decoded.

XOR

XOR is exactly the same as "not equal to". 0 XOR 1 = 1, 1 XOR 0 = 1, 0 XOR 0 = 0, 1 XOR 1 = 0.