2003
South Jersey Regional
High School Programming Contest

hosted by the
Rowan University Computer Science Department
Saturday, 12 April 2003

Contest Problem

1 Background

The Universal Product Code, which appears on most consumer goods in the USA, uses bars and spaces to represent binary digits (called 'bits'). By allowing mechanical scanners to identify products, the UPC speeds not only checkout, but also inventory and delivery tracking. The most common form of the UPC, UPC-A, represents 12 decimal digits, which are also printed along the bottom, allowing a person to type in the digits if the code does not scan correctly.

The outermost bars are two thin bars with one thin space between them, which the scanner uses for calibration. Similarly, the middle set is two thin bars with one thin space between and one thin space on each side. The bars and spaces come in four widths, with the thicker bars (spaces) being 2, 3, or 4 times as thick as the thinnest bars (spaces).

The bars are read as 1s, and the spaces are read as 0s. The thinnest bar (space) is read as a single 1 (0), and the next thicker bar (space) is read as two 1s (0s), and so on to the thickest bar (space), which is read as four 1s (0s).

For example, here is the UPC code from a box of Quaker Oatmeal Squares:

0 30000 06570 9

The scanner only actually reads a thin slice across the bars. Here is an approximation of what the scanner sees, and the translated bits:

10100011010111101000110100011010001101000110101010111001010100001001110100010011100101110100101

The 95 bits are interpreted by breaking them down into 15 groups:

101 0001101 0111101 0001101 0001101 0001101 0001101 01010 1110010 1010000 1001110 1000100 1110010 1110100 101
 1   2   3   4   5   6   7   8   9   10   11   12   13   14   15

The groups are interpreted as follows:

Group	1	2-7	8	9-14	15
Meaning	left guard bits (3 bits)	left-side digits (7 bits each)	center guard bits (5 bits)	right-side digits (7 bits each)	right guard bits (3 bits)

The left, center, and right guard bits are always the same, and are used for calibration. The other bits encode the actual digits of the bar code.

The six digits on the left-hand side are encoded using this table:

0	0001101	2	0010011	4	0100011	6	0101111	8	0110111
1	0011001	3	0111101	5	0110001	7	0111011	9	0001011

The six digits on the right-hand side are encoded using this table:

0	1110010	2	1101100	4	1011100	6	1010000	8	1001000
1	1100110	3	1000010	5	1001110	7	1000100	9	1110100

Each right-hand encoding is the complement of the left-hand encoding for the same digit; that is, all the 0s have been replaced with 1s, and vice-versa. Note that each code is unique both backwards and forwards.

Looking at just the first and last ten bits of the example UPC, we see:

$$\underbrace{101}_{\text{left guard bits}} \qquad \underbrace{0001101}_{\text{leftmost digit encoding}} \qquad \text{...} \qquad \underbrace{1110100}_{\text{rightmost digit encoding}} \qquad \underbrace{101}_{\text{right guard bits}}$$

Remembering to omit the guard bits, we see the seven-bit groupings for the leftmost and rightmost digits. From the table of left-hand encodings, we see that '0101111' is the encoding for 0; from the table of right-hand encodings, we see that '1110100' is the encoding for 9. These digits (and the others, as can be seen by examination) match those printed at the bottom of the sample UPC.

In addition to each code being unique, there are three other features designed to enable error-checking:

1. The left- and right- hand encodings have different parity.

   (a) Each left-hand encoding has *odd parity*, which means there are an odd number of 1s.
   (b) Each right-hand encoding has even parity (an even number of 1s).

   As a result of the parity settings on the left and right side, as well as each code being unique, it is easy for a scanner to tell whether a code has been read backwards. If a barcode has been scanned in the wrong way, the scanner can reverse the order of the bits before decoding.

2. The 12th digit is *modulo check digit*, computed based on the first 11 digits. The odd-numbered digits (1st, 3rd, 5th, and so on) are added together and the sum multiplied by 3. Then the even-numbered digits (2nd, 4th, and so on) are added in to that result. The final total is subtracted from the next-highest multiple of 10. For our example UPC, we get:

   $$3 \times (0 + 0 + 0 + 0 + 5 + 0) + (3 + 0 + 0 + 6 + 7)$$

   which gives 31. The next highest multiple of 10 is 40, and 40 - 31 gives 9, which is the last digit as it appears on the UPC code from the box.

3. The left, center, and right guard bits are always exactly the same.

These properties of bar codes greatly increase their reliability. It is impossible for a single bit to be read incorrectly and still produce a valid code: either a set of guard bits will be wrong, or one of the 12 digits will have the wrong parity. Based on this, it should be obvious from a moment's thought that no odd number of incorrect bit readings can produce a valid result.

It is also impossible for any two bits to be read incorrectly and still produce a valid code: if they aren't in the same digit encoding, then there will be some combination of incorrect guard bits and incorrect parity readings. If two wrong bits do appear in the same digit encoding, it will either be a coding not in either of the tables, in which case the barcode won't scan, or it will be a valid coding but the changed digit will cause the check digit to be incorrect.

As no odd number of wrong bits can produce a valid code, and as it's not possible for two bits to go wrong and have a valid result, for a code to read incorrectly and still be accepted there must be at least four bits affected, and those four must be in one of a relatively few arrangements. Based on the specifications for UPC-A, the probability that four random bit changes will produce a scannable result is 0.00002073[1].

The programming problem for this contest is to interpret a 95-character string of 0s and 1s as if it were a UPC code, printing out the digits represented and reporting on any errors that may be present.

---

[1]Thanks to Dex Whittinghill of the Rowan Math Department for his assistance figuring this out.

2 Input/Output Specification

2.1 Input

For text input, your program should accept input in the following format:

1. A single integer, $\mathcal{N}$, laying out the number of UPC encodings to be tested.
2. $\mathcal{N}$ lines, each with exactly 95 characters, either 1 or 0.

You do not need to do error-checking on the input. There will be nothing on the first line except a single integer. There will be no characters except for 1 or 0 on the remaining lines. Each line of data will have exactly 95 characters. There will be no blank lines.

You may choose to have your program read the input directly from the keyboard, or ask the user for a filename and then read the file. Alternatively, users of GUI-based programming environments may prefer to have a single text box into which a bitstring is entered, and also have a button which causes the program to report on that bitstring. Any of these variations is acceptable.

2.2 Output

Your program must generate output according to the following description:

1. The text **Item** *N***:** , indicating which UPC code is being reported on.
2. 12 digits in the format *n nnnn nnnn n*, indicating the numbers represented by that UPC code. If a digit is invalid, the program should print a lower-case letter instead of a digit, indicating the problem:

   (a) In case of a parity error print a **p**.

   (b) In case of an unknown code which is not in the table, print a **u**.

   If a code is both unknown *and* incorrect parity for the position it is in, the parity error takes priority, so you would print a **p**. Digits which *can* be identified from the input should be displayed, even if other digits are garbled.

3. If the code was reversed, your program should recognize this and correct for it, printing the 12 digits it represents as described above and adding the notation '**(reversed)**' at the end of the line. A backwards code is not itself an error.

   Note that some codes may have more parity errors in one direction than the other; in that case, you should interpret the code in the way that produces the fewest errors. If the code has the same number of parity errors both ways, treat it as if the original scan was correct.

4. If no errors were found, print **No Errors Found** at the end of the line.

5. If an error in one or more of the digits was found, a line describing them should be added. It should begin with the message **Digit Errors:** and then say how many had wrong parity and how many were unknown.

6. If the check digit is incorrect, a line should be added that begins **Check Digit Error:** and indicates what the check digit should be.

   Note that your program cannot test the check digit if there was a parity error or an unknown code, since either (a) it can't identify the 11 digits necessary to compute the modulo check digit, or (b) it can't identify the check digit itself.

7. If a set of guard bits was incorrect, add a line which begins **Guard Bit Errors:**, and indicates which set(s) of guard bits was wrong with one or more of the words **left**, **center**, **right**.

Your output does **not** have to conform exactly to the sample output as regards spacing or use of upper/lower case. The samples were formatted to be easy to read, but duplicating that format is **not** a requirement for your program.

3   Sample Data

3.1   Sample Input

```
10
101010111101111010101111000101100100110001101010101110100110110010111001101100100010011011001 01
101010111101111010101111000101100100110001101010101110100110110010111001101100100010011010001 01
101100111101111010101111000101100100110001101010101110100110110010111001101100100010011011001 01
101010111101111010101111000101100100110001101010101110100110110010111001101100100010010111001 01
001010111101111010101111000101100100110001101010101110100110110010111001101100100010011011001 01
001010111101111010101111000101100100110001101011101110100110110010111001101100100010010111001 01
101001101100100010011011001110100110110010111010101011000110010010110100011101010111101111010 101
111001101100100010011011001110100110110010111011101011000110010010110100011101010111101111010 111
111001110100100010011011001110100110110010111011101011000110010010110100011101010111101111010 111
111001110100100010111011101110100110110010111011101011000110010010110100011101010111011111010 111
```

(Your floppy disk has a copy of this data set in the file named **sample.txt**.)

3.2   Sample Output

```
Item  1: 6 36920 92427 2  No Errors Found

Item  2: 6 36920 92427 p
         Digit Errors: 1 wrong parity.

Item  3: u 36920 92427 2
         Digit Errors: 1 unknown.

Item  4: 6 36920 92427 4
         Check Digit Error: should be 2.

Item  5: 6 36920 92427 2
         Guard Bit Errors: left.

Item  6: 6 36920 92427 4
         Check Digit Error: should be 2.
         Guard Bit Errors: left center.

Item  7: 6 36920 92427 2  (reversed)  No Errors Found

Item  8: 6 36920 92427 2  (reversed)
         Guard Bit Errors: left center right.

Item  9: 6 36920 92427 4  (reversed)
         Check Digit Error: should be 2.
         Guard Bit Errors: left center right.

Item 10: 6 u6920 92pp7 4  (reversed)
         Digit Errors: 2 wrong parity.  1 unknown.
         Guard Bit Errors: left center right.
```

4   Test Data

Run your program on this input and make enough screenshots to show the output:

```
25
10100011010111101000110100011010001101000110101010111001010100001001110100010011100101110100101
10100011010111101010110100011010001101000110101010111001010100001001110100010011100101110100101
10100011010111101000101100011010001101000110101010111001010100001001110100010011100101110100101
10100011010111101000110100011010001101000110100010111001010100001001110100010011100101110100101
10100011010111101000110100011010001101000110101010111001010100001001110100010011100101110010101
00101011110111101010111100010110010011000110101110111010011011001011100110110010001001011100101
10100110110010000100110110011101001101100101110101010110001100100110100011110101011110111010101
11100110110010000100110110011101001101100101110111010110001100100110100011110101011110111010111
11100111010010000100110110011101001101100101110111010110001100100110100011110101011110111010111
11100111010010001011101110111010011011001011101110101100011001001101000111101010111011111010111
10100011010001011011101101101110110001011000101010111001011001101110010110011011011001011100101
10100011010001011011101101101110110001010001101010111001011001101110010110011011011001011100101
10100011010001011011101101101110110001011000101000111001011001101110010110011011011001011100101
10100111010011011011001101001101100110100111010101000110100011011101101101110110100010110000001
11100011010001011011101101101110110001011000101110111001011001101110010110011011011001011100111
10100011010100011001001101101111011011101000110101011100101011100110011011011001111001010011010101
1010010101010001100100110110111101101110100011010101110010101110011001101101100111001010011010101
10101110110110001001001100011010110111000110101010110011011001101100110110011011001101000010101
10101000100110011011001101100110110011011001101010101100011011010110001100100100011011011110101
10100011010110001001100100011010001101000110101010101110010110011011011001001110110011010000100101
10101110110100011001001101110110010011011000101010110011010011101001000101000010000101011100101
10100111010100001000010100010010111001011001101010100011011001001011110110010011010101011011110101
10100011010100011011110100110010101111011011101010111010011001011100101110010100111010010001010101
10100010010111001010011101001110100111001011101010111011011111010100110010111101101010101011000101
10100010010111001010011101001110100111001011101010111011011111010100110010111101101010101011000101
```

(Your floppy disk has a copy of this data set in the file named **testdata.txt**.)

Your program will be tested on additional data known only to the judges.