Problem A: Trip Routing

Your employer, the California Car Club (CCC), has decided to provide a trip routing service to its members. Your job is to write a program which reads a list of departure point-destination point pairs and calculates the shortest routes between them. For each trip, your program will print a report which itemises the names of each city passed through, with route names and leg distances.

Input to your program will be in two parts. The first part is a map in the form of a list of highway segments. Each segment is designated by a line containing four fields which are separated by commas. The first two fields are 1–20 characters each, and are the names of the cities which are at each end of the highway segment. The third field is the 1–10 character name of the route. The fourth field is the number of miles between the two endpoints, expressed as a positive integer. The highway segment list will be terminated by an empty line.

The second part of the input is a list of departure point-destination point pairs, one per line. The departure point is given first, followed by a comma and the destination point. Each of the cities is guaranteed to have appeared in the first part of the input data, and there will be a path that connects them. The list is terminated by the end of file.

The output should be a series of reports, one for each departure point-destination point pair in the input. Each report should be in exactly the same form as those in the example below. There should be two blank lines before each report.

There will be no extraneous blanks in the input. There will be no more than 100 cities in the map and no more than 200 highway segments. The total distance in each best route is guaranteed to fit within a 16-bit integer.

Sample input

```
San Luis Obispo,Bakersfield,CA-58,117
Bakersfield,Mojave,CA-58,65
Mojave,Barstow,CA-58,70
Barstow,Baker,I-15,62
Baker,Las Vegas,I-15,92
San Luis Obispo,Santa Barbara,US-101,106
San Luis Obispo,Santa Barbara,CA-1,113
Santa Barbara,Los Angeles,US-101,95
Bakersfield,Wheeler Ridge,CA-99,24
Wheeler Ridge,Los Angeles,I-5,88
Mojave,Los Angeles,CA-14,94
Los Angeles,San Bernardino,I-10,65
San Bernardino,Barstow,I-15,73
Los Angeles,San Diego,I-5,121
San Bernardino,San Diego,I-15,103

Santa Barbara,Las Vegas
San Diego,Los Angeles
San Luis Obispo,Los Angeles
```

Sample output

From	To	Route	Miles
Santa Barbara	Los Angeles	US-101	95
Los Angeles	San Bernardino	I-10	65
San Bernardino	Barstow	I-15	73
Barstow	Baker	I-15	62
Baker	Las Vegas	I-15	92
		Total	387

From	To	Route	Miles
San Diego	Los Angeles	I-5	121
		Total	121

From	To	Route	Miles
San Luis Obispo	Santa Barbara	US-101	106
Santa Barbara	Los Angeles	US-101	95
		Total	201

Problem B: Transaction Processing

You have been called upon to write a program which performs one of the initial steps in posting transactions to a general ledger. The central principle of double-entry bookkeeping is that the sum of all debits must equal the sum of all credits. This is true for each transaction. For the purposes of your program, positive numbers represent debits and negative numbers represent credits. That is, 2.00 is a two dollar debit, and -2.00 is a two dollar credit. The purpose of your program is to check that each transaction balances, and to report it if it doesn't.

Input data to your program will come in two sections. The first section is a list of up to 100 accounts in the general ledger. It consists of lines in the format:

```
nnnxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

where `nnn` is a three-digit account number and `xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx` is a 1–30 character account name string. This section is terminated by a record starting with 000, which is not used as an account number.

The second section of the input data consists of 15-character records, one per line in the format

```
sssnnnxxxxxxxxx
```

where `sss` is a three-digit sequence number, `nnn` is a three-digit account number, and `xxxxxxxxx` is a nine-digit amount in dollars and cents (without the decimal point). Each of these records is one entry of a transaction. A transaction consists of between two and ten entries with identical sequence numbers. Each transaction will be contiguous within the input data. This section of input data is terminated by a record which has a sequence number of 000.

Nothing is to be printed for transactions which balance. For transactions which do not balance, an exception report is to be printed in the form:

```
*** Transaction sss is out of balance ***
nnn xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx vvvvvvv.vv
nnn xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx vvvvvvv.vv
.
.
.
999 Out of Balance                  vvvvvvv.vv
```

where `nnn` is an account number, `xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx` is the corresponding account name, and `vvvvvvv.vv` is the amount. Print a space between the above fields. The entries should be listed in the order that they were received in the input. The last entry in the report is one you will create to make the transaction balance, using the special account number 999 (the suspense account). Print a blank line after each exception report.

Sample input

```
111Cash
121Accounts Receivable
211Accounts Payable
241Sales Tax Payable
401Sales
555Office Supplies
000No such account
100111    11795
100121   -11795
101121      105
101241       -7
101401     -100
102211   -70000
102555    40000
103111   -40000
103555    40000
000000        0
```

Sample output

```
*** Transaction 101 is out of balance ***
121 Accounts Receivable                1.05
241 Sales Tax Payable                 -0.07
401 Sales                             -1.00
999 Out of Balance                     0.02

*** Transaction 102 is out of balance ***
211 Accounts Payable                -700.00
555 Office Supplies                  400.00
999 Out of Balance                   300.00
```

Problem C: Perfect Hash

Perfect Software, Inc. has obtained a government contract to examine text flowing through a high-speed network for the occurrence of certain words. Your boss, Wally Perfect, has designed a parallel processing system which checks each word against a group of small perfect hash tables.

A perfect hash function maps its input directly to a fully occupied table. Your job is to construct the perfect hash functions from the lists of words in each table. The hash function is of the form $\lfloor C/w \rfloor \bmod n$, where C is a positive integer you are to discover, w is an integer representation of an input word, and n is the length of the table. C must be as small as possible. Note that $\lfloor\ \rfloor$ is the floor function and that $\lfloor R \rfloor$ for some real number R is the largest integer that is $\leq R$.

Here are Wally's notes on the subject:

Let $W = \{w_1, w_2, \ldots, w_n\}$ consist of positive integers $w_1 < w_2 < \ldots < w_n$. The problem is to find the smallest positive integer C such that

$$\left\lfloor \frac{C}{w_i} \right\rfloor \bmod n \neq \left\lfloor \frac{C}{w_j} \right\rfloor \bmod n \qquad \text{for all } 1 \leq i < j \leq n.$$

C must be a multiple of at least one element of W.

If some

$$\left\lfloor \frac{C}{w_i} \right\rfloor \bmod n = \left\lfloor \frac{C}{w_j} \right\rfloor \bmod n \qquad \text{for all } i \neq j,$$

then the next largest C that could resolve the conflict is at least

$$\min\left(\left(\left\lfloor \frac{C}{w_i} \right\rfloor + 1\right) \cdot w_i, \left(\left\lfloor \frac{C}{w_j} \right\rfloor + 1\right) \cdot w_j\right)$$

Since all such conflicts must be resolved, it is advantageous to choose the largest candidate from among the conflicts as the next C to test.

You are to convert each word to a number by processing each letter from left to right. Consider 'a' to be 1, 'b' to be 2, ..., 'z' to be 26. Use 5 bits for each letter (shift left by 5 or multiply by 32). Thus 'a' = 1, 'bz' = $(2 \cdot 32) + 26 = 90$.

Input to your program will be a series of word lists, one per line, terminated by the end-of-file. Each line consists of between two and thirteen words of at most five lower case letters each, separated from each other by at least one blank. There will always be at least one one-letter word.

For each list, you are to print the input line. On the next line, print the C for the hash function determined by the list. Print a blank line after each C.

C will always fit in a 32-bit integer.

Sample input

```
this is a test of some words to try out
a bee see dee
the of and to a in that is i it with for as
```

Sample output

```
this is a test of some words to try out
17247663
```

```
a bee see dee
4427

the of and to a in that is i it with for as
667241
```

Problem D: Pascal Program Lengths

Your local computer user's group publishes a quarterly newsletter, and in each issue there is a small Turbo Pascal programming problem to be solved by the membership. Members submit their solutions to the problem to the newsletter editor, and the member submitting the shortest solution to the problem receives a prize.

The length of a program is measured in units. The unit count is determined by counting all occurrences of reserved words, identifiers, constants, left parentheses, left brackets, and the following operators: +, −, *, /, =, <, >, <=, >=, <>, @, ^, and :=. Comments are ignored, as are all other symbols not falling into one of the categories mentioned above. The program with the lowest unit count is declared the winner. Two or more programs with equal unit counts split the prize for the quarter.

In an effort to speed the judging of the contest, your team has been asked to write a program that will determine the length of a series of Pascal programs and print the number of units in each.

Input to your program will be a series of Turbo Pascal programs. Each program will be terminated by a line containing tilde characters in the first two columns, followed by the name of the submitting member. Each of these programs will be syntactically correct and use the standard symbols for comments (braces) and subscripts (square brackets).

For each program, you are print a separate line containing the name of the submitting member and the unit count of the program. Use a format identical to that of the sample below.

In the Southern California Regional finals, all teams were using Turbo Pascal. Here are some additional notes on Turbo Pascal for those not familiar with the language:

- Identifiers start with an underscore (_) or a letter (upper or lower case) which is followed by zero or more characters that are underscores, letters or digits.
- The delimiter for the beginning and ending of a string constant is the single forward quote (’). Each string is entirely on a single source line (that is a string constant cannot begin on one line and continue on the next). If ’’ appears within a string then it represents a single ’ character that is part of the string. A string constant consisting of a single ’ character is, therefore, represented by ’’’’ in a Turbo Pascal program. The empty string is allowed.
- The most general form of a numeric constant is illustrated by the constant `10.56E-15`. The `10` is the integral part (1 or more digits) and is always present. The `.56` is the decimal part and is optional. The `E-15` is the exponent and it is also optional. It begins with an upper or lower case `E`, which is followed by a sign (`+` or `-`). The sign is optional.
- Turbo Pascal supports hexadecimal integer constants which consist of a $ followed by one or more hex digits (‘0’ to ‘9’, ‘a’ to ‘f’, ‘A’ to ‘F’). For example, $a9F is a legal integer constant in Turbo Pascal.
- The only comment delimiters that you should recognise are {}, and not `(**)`. Comments do not nest.
- ‘+’ and ‘−’ should be considered as operators wherever possible. For example in `x := -3` the ‘−’ and the ‘3’ are separate tokens.

- Subranges of ordinal types can be expressed as `lower..upper`. For example, `1..10` is a subrange involving the integers from 1 to 10.
- All tokens not mentioned anywhere above consist of a single character.

Sample input

```
PROGRAM SAMPLEINPUT;

VAR
  TEMP : RECORD
    FIRST, SECOND : REAL;
    END;

BEGIN {Ignore this }
TEMP.FIRST := 5.0E-2;
READLN (TEMP.SECOND);
WRITELN ('THE ANSWER IS', TEMP.FIRST * TEMP.SECOND : 7 : 3)
END.
~~A. N. Onymous
```

Sample output

```
Program by A. N. Onymous contains 29 units.
```

Problem E: Circle Through Three Points

Your team is to write a program that, given the Cartesian coordinates of three points on a plane, will find the equation of the circle through them all. The three points will not be on a straight line.

The solution is to be printed as an equation of the form

$$(x - h)^2 + (y - k)^2 = r^2 \tag{1}$$

and an equation of the form

$$x^2 + y^2 + cx + dy - e = 0 \tag{2}$$

Each line of input to your program will contain the x and y coordinates of three points, in the order A_x, A_y, B_x, B_y, C_x, C_y. These coordinates will be real numbers separated from each other by one or more spaces.

Your program must print the required equations on two lines using the format given in the sample below. Your computed values for h, k, r, c, d, and e in Equations 1 and 2 above are to be printed with three digits after the decimal point. Plus and minus signs in the equations should be changed as needed to avoid multiple signs before a number. Plus, minus, and equal signs must be separated from the adjacent characters by a single space on each side. No other spaces are to appear in the equations. Print a single blank line after each equation pair.

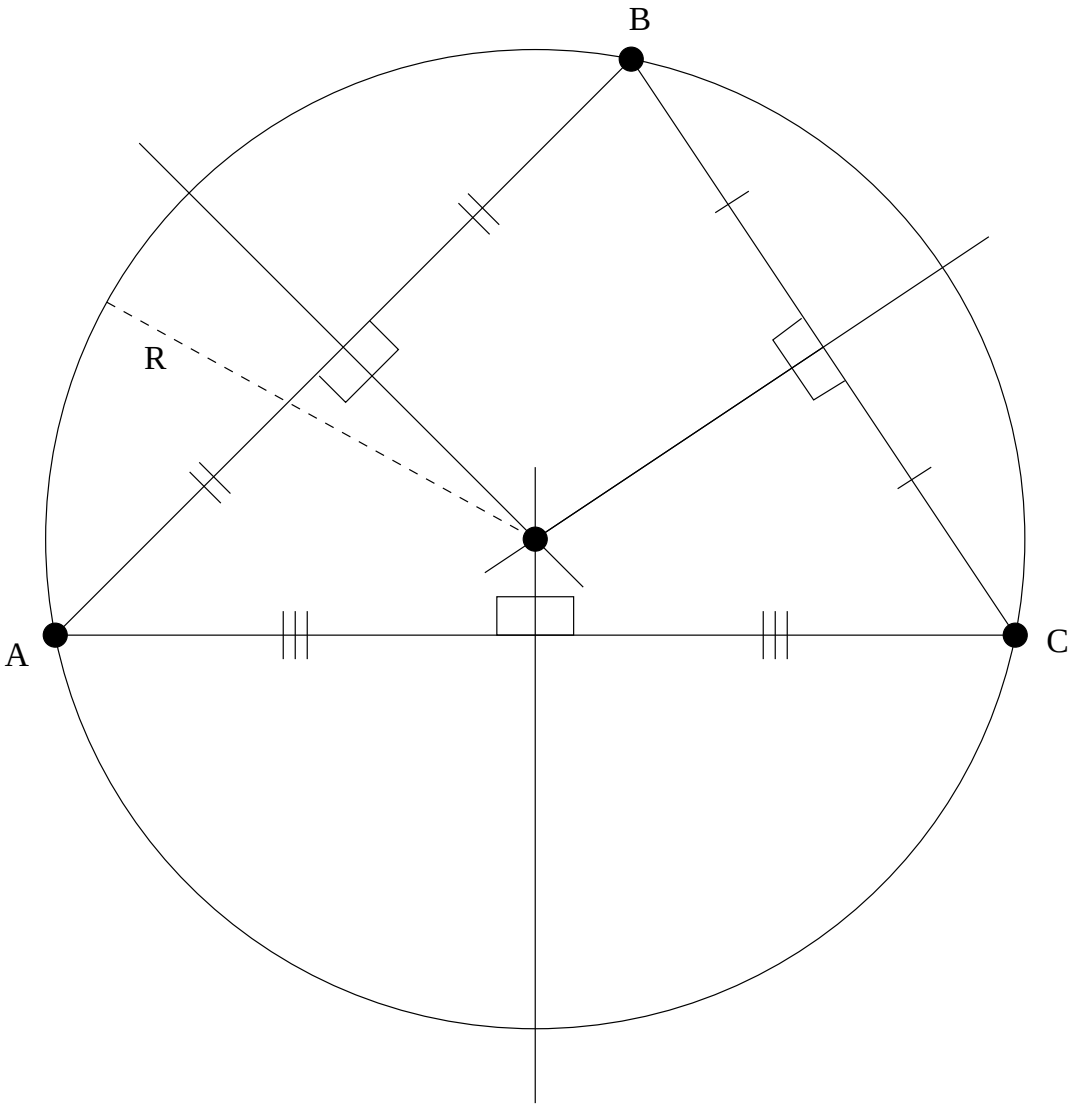
Sample input

```
7.0 -5.0 -1.0 1.0 0.0 -6.0
1.0 7.0 8.0 6.0 7.0 -2.0
```

Sample output

```
(x - 3.000)^2 + (y + 2.000)^2 = 5.000^2
x^2 + y^2 - 6.000x + 4.000y - 12.000 = 0

(x - 3.921)^2 + (y - 2.447)^2 = 5.409^2
x^2 + y^2 - 7.842x - 4.895y - 7.895 = 0
```


B
R
A
C

Problem F: BitBlt

Home Garage Enterprises has selected your organisation to write the graphics software for their projected FGA (Future Graphics Adapter) for the IBM PC and compatibles. Your team has been assigned the task of writing the BitBlt routine, which will need to interface with routines written elsewhere in your organisation.

The FGA is a memory mapped, bit plane oriented colour frame buffer. It supports 256 by 256 pixels in up to three bit planes, for eight colours. The pixels are packed 16 to a 32-bit integer, with the rightmost pixel in the low-order bit of each integer. (Your job is to write software for this thing, not to tell Home Garage Enterprises how bad their market research was. After all, they did pay in advance.)

The BitBlt routine is designed to handle rectangular areas on a graphics display or on a display image in memory. It allows you to specify a source and destination rectangle of the same size and a logical operation to perform in moving the pixels of the source rectangle into the destination rectangle. The BitBlt routine will align the source rectangle over the destination rectangle and will perform the logical operation between corresponding pixels in each rectangle, replacing the pixels in the destination rectangle with the result.

Your BitBlt routine is designed to handle a single bit plane. It will be called multiple times by the other software if the FGA is set to one of its multiple bit plane modes. It will be used both for reading and writing display memory, and for performing the same operations on non-display memory which can be used as scratchpads for other drawing operations.

The BitBlt routine will be implemented as a separately compiled file of Pascal or C functions. Suitable header files and skeleton files have been provided for both Pascal and C. The discussion below is in terms of Pascal, with any significant differences between the Pascal and C cases noted (the precise case of C identifiers can be seen in the C header file). The BitBlt procedure is defined as follows:

```
TYPE
  BltOp= (BltClear, BltNotSAndNotD, BltSAndNotD, BltInvert,
          BltNotSAndD, BltInvertCopy, BltSXorD, BltNotSOrNotD,
          BltSAndD, BltSEquivD, BltCopy, BltSOrNotD,
          BltNoop, BltNotSOrD, BltSOrD, BltSet);

PROCEDURE BitBlt(src:BltBitmapPtr; SX,SY:INTEGER;
                 dest:BltBitmapPtr; DX,DY:INTEGER;
                 Width, Height:INTEGER; Op:BltOp);
```

The header files and the procedure definitions in the skeleton files contain official definitions and must not be modified.

The arguments to procedure BitBlt are a source bitmap with the coordinates of the upper left corner of the source rectangle, a destination bitmap with the coordinates of the upper left corner of the destination rectangle, the width and height of the rectangle, and the operation to be performed. The source and destination rectangles are combined using the given operation to modify the destination bitmap.

Bitmaps are rectangular arrays of bits. They are defined in the Pascal and C header files which are described later. The coordinates of the bit in the upper left corner is (0, 0) and the coordinates of the bit in the lower right corner is (Bitmap^.Width – 1, Bitmap^.Height – 1). BitBlt operations are clipped by the boundaries of the source and destination bitmaps. Thus the BitBlt call:

```
BitBlt(S,S^.Width-20,S^.Height-20,D,-10,-10,30,30,Op);
```

will affect only the rectangle between (0, 0) and (9, 9) in D.

There is nothing to prevent the same bitmap from being used as both source and destination. If this occurs, and the source and destination rectangles overlap, care must be taken to ensure that the bits of the source rectangle are not being modified before they are used. Note that the routines provided for bit map manipulation guarantee a unique correspondence between a bitmap record and its bit array, so that a comparison of the bitmap record pointers is sufficient to tell that the same bitmap is being used.

While the names and order of the BitBlt operations may seem arbitrary, there is actually both a rhyme and a reason for them. With the source bit being either 0 or 1, and the destination bit being either 0 or 1, four truth table entries must be defined in order to determine the outcome of combining one bit from the source and one bit from the destination. Four entries, each of which could be either 0 or 1 means that there are sixteen unique truth tables which define all possible logical operations on the two bits. The generalised truth table (Table 1) shows the logical term which would evaluate as 1 given the input values associated with that location in the table and 0 given the input values anywhere else.

Table 2 arranges those terms in a row and shows how each BltOp defines the corresponding entries in its truth table. The order of the operations was determined by treating the bits of the operation's truth table as a binary integer and sorting.

To understand the choice of names, you must realize that you can generate the logical formula of a BltOp by taking the logical or of all the terms which head the columns whose value is 1 for that BltOp. Thus, BltSXorD has 1s in the second and third columns giving a logical formula of (not S and D) or (S and not D), which is the definition of exclusive or. Similarly, the names of the other BltOps have been chosen by transforming the resulting logical formula into a form which is more intuitive than the (equivalent) bare recitation of the combination of terms.

Note that BltClear, BltInvert, BltNoop and BltSet do not depend on the source bitmap. Nevertheless a source and a destination bitmap will always be provided. The results of passing nil as the pointer to either the source or destination bitmap when calling BitBlt is undefined and if it happens, BitBlt is permitted to walk all over memory, trash the disk, or anything else that will relieve its frustration at not being able to perform its appointed task.

The Pascal and C header files define the bitmap constants and types and declare the routines for allocating and disposing of bitmaps. The allocation and disposal routines were developed by another team in your organisation, and are provided in a compiled form.

	S 1	S 0
D 1	SD	$\overline{S}D$
D 0	$S\overline{D}$	$\overline{SD}$

$\overline{S}$ means "not S"
SD means "S and D"
Table 1

	SD	$\overline{S}D$	$S\overline{D}$	$\overline{SD}$
BltClear	0	0	0	0
BltNotSAndNotD	0	0	0	1
BltSAndNotD	0	0	1	0
BltInvert	0	0	1	1
BltNotSAndD	0	1	0	0
BltInvertCopy	0	1	0	1
BltSXorD	0	1	1	0
BltNotSOrNotD	0	1	1	1
BltSAndD	1	0	0	0
BltSEquivD	1	0	0	1
BltCopy	1	0	1	0
BltSOrNotD	1	0	1	1
BltNoop	1	1	0	0
BltNotSOrD	1	1	0	1
BltSOrD	1	1	1	0
BltSet	1	1	1	1

Table 2

Header file declarations

CONSTANTS

`BitsPerWord` The number of bits stored in an integer (16). This constant is defined to give a meaningful name to the value, and its value will never be changed.

`FGAWidth` Width in pixels of FGA display.

`FGAHeight` Height in pixels of FGA display.

`MaxBitmapSize` Maximum number of integers which could be allocated to the actual bit array in the bitmap. The bit array may be smaller.

TYPES

```
BltBitmap = RECORD
  Width : INTEGER;          (* Width of bitmap in pixels. *)
  Height : INTEGER;         (* Height of bitmap in pixels.*)
  Bitmap : ^BitArray;       (* Pointer to the bitmap array. The *)
  END;                      (* bits of the bitmap are packed into *)
                            (* words in the bit array. In the C version *)
                            (* bitmap is simply declared as int *. *)

BltBitmapPtr = ^BltBitmap;
                            (* Pointer to a bitmap record. This is the
                               normal way of referencing a bitmap. *)
```

```
BitArray = ARRAY [0..MaxBitmapSize-1] OF INTEGER;
             (* The array type used to access the actual bit
                array of the bitmap. It is defined only to
                give Pascal access to the bits. Be warned that
                the actual size of the array is determined from
                the width and height of the bitmap, and higher
                indexed elements of this array will reference
                beyond the end of the memory allocated to it.
                Each row begins on a word boundary, so the
                actual number of words allocated to the array is
                ((Width+BitsPerWord-1) DIV BitsPerWord) * Height.
                This type isn't defined in the C header because
                bitmap is declared int *. *)
```

ROUTINES

```
FUNCTION NewBltBitmap(Width, Height : INTEGER):BltBitmapPtr
          (* Given the width and height in pixels of a bitmap,
             allocates a bit array big enough to hold it,
             and then allocates and initialises a bitmap
             record to represent it. The bit array itself
             remains uninitialised. A pointer to the new
             bitmap record is returned.

             Once the first FGA arrives, a similar function
             will be provided which will return a pointer
             to a predefined bitmap record whose bit array
             pointer points into FGA memory, so that FGA
             based bitmaps can be defined. *)

PROCEDURE DisposeBltBitmap(VAR BltBitmap:BltBitmapPtr);
          (* Given a pointer to a bitmap, free the bit array
             associated with it, and then free the bitmap.
             When FGA memory can be referenced, this routine
             will be smart enough to recognise FGA based
             bitmaps and will not to attempt to free them.
             C programmers should note that BltBitmap is a VAR
             parameter, so when DisposeBltBitmap is called from
             C the address of a variable of type BltBitmapPtr
             should be passed *)

FUNCTION WordWidth(PixWidth : INTEGER) : INTEGER;
          (* Given the width of a bitmap in pixels, returns
             the number of integers required to hold a row.
             The function assumes that the leftmost pixel of
             a row is the most significant of the 16 bits stored
```

```
in an integer. *)
```