Problem A - EPIC

Professor Plum likes the idea of visiting EPIC for MICS 2014. He wants you to write a program to generate ASCII art printing "EPIC" vertically for a sign to tape to the back of the van on the trip. Since he does not know the dimensions of the sign, he wants your program to take as input a positive integer scaling factor.

Scaling Factor	Letter Dimension (# characters × # characters)	Line Width of Letters (# characters)	Blank Lines Between Letters
1	5 × 5	1	1
2	10 × 10	2	2
3	15 × 15	3	3
10	50 × 50	10	10

A scaling factor of 1 would produce:

```
EEEEE
E
EEEE
E
EEEEE

PPPPP
P   P
PPPPP
P
P

IIIII
  I
  I
  I
IIIII

CCCCC
C
C
C
CCCCC
```

A scaling factor of 2 would produce:

```
EEEEEEEEEE
EEEEEEEEEE
EE
EE
EEEEEEEEEE
EEEEEEEEEE
EE
EE
EEEEEEEEEE
EEEEEEEEEE


PPPPPPPPPP
PPPPPPPPPP
PP      PP
PP      PP
PPPPPPPPPP
PPPPPPPPPP
PP
PP
PP
PP


IIIIIIIIII
IIIIIIIIII
    II
    II
    II
    II
IIIIIIIIII
IIIIIIIIII


CCCCCCCCCC
CCCCCCCCCC
CC
CC
CC
CC
CC
CC
CCCCCCCCCC
CCCCCCCCCC
```

INPUT SPECIFICATION.
The input contains a single line with a positive integer scaling factor.

OUTPUT SPECIFICATION.
The output should contain the ASCII art for the sign corresponding to the scaling factor specified by the input. **NOTE**: Except for the blank line(s) separating EPIC letters, each line should be the same length by padding shorter lines with blanks.

Problem B - Even-Odd Sort

Even Professor Plum knows that he is a little odd. Thus, on a recent test he had his students write the following "even-odd sort" to arrange positive integers such that:

- smallest even integer is at index 0,
- smallest odd integer is at index 1,
- second smallest even integer is at index 2,
- second smallest odd integer is at index 3,
- etc.
- pattern continues until either the even or odd numbers run out, then the remain values are in ascending order.

INPUT SPECIFICATION.
The input consists of no more than 1000 positive integers with each integer on a line by itself. The last line in the file contains the integer 0 which is NOT to be included in the sort.

OUTPUT SPECIFICATION.
The output consists of the positive input values rearranged in "even-odd" sorted order as defined above. Each integer should be on a line by itself.

Sample Input

```
8
2
11
6
3
6
5
13
4
7
1
9
0
```

Output for the Sample Input

```
2
1
4
3
6
5
6
7
8
9
11
13
```

Problem C - Word-Find Solver

The Pi-rats Restaurant is Professor Plums favorite restaurant because the placemats always have a word-find puzzle to entertain his kids while he reads the newspaper. Occasionally, his kids ask for help which cuts into his reading time, so he has asked the Pi-rat Restaurant to post the word-find puzzle on their web-site. Now, he wants a program to find the words in the puzzle.

INPUT SPECIFICATION.
The first line of input will specify the length (in characters) of the sides of the letter matrix (the matrix of letters will be square). The length, *l*, will be in the range 1 <= *l* <= 100. The next *l* lines of input will be the matrix itself, each line will contain *l* uppercase letters.

A list of words will follow. Each word will be on a line by itself; there will be 100 or fewer words. Each word will be 100 or fewer characters long, and will only contain uppercase letters.
The final line of input will contain a single zero character.

OUTPUT SPECIFICATION.
Your program should attempt to find each word from the word list in the puzzle. A word is "found" if all the characters in the word can be traced in a single (unidirectional) horizontal, vertical, or diagonal line in the letter matrix. Words may not "wrap around" the edges of the puzzle, but horizontal, vertical and diagonal words may be in reverse order ("backwards"), i.e., all eight directions from beginning to end like a “normal” word-find puzzle.

Each word should produce one line of output. For each word that is found, your program should print the coordinates of its first and last letters in the matrix on a single line, separated by a single space. Coordinates are pairs of comma-separated integers (**indexed from 1**), where the first integer specifies the row number and the second integer specifies the column number. If a word is not found, the string "`Not found`" should be output instead of a pair of coordinates. Each word from the input will be in the puzzle at most once.

Sample Input

```
5
ECEEE
ESEEE
DISKE
EDEEE
EEEEE
DISC
DISK
DICE
0
```

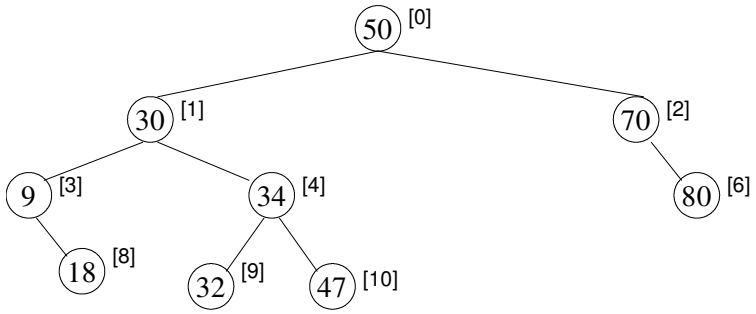
Output for the Sample Input

```
DISC: (4,2) to (1,2)
DISK: (3,1) to (3,4)
DICE: Not found
```

Problem D — BST Maximum Index

Trees are particularly annoying to Professor Plum. He likes to fly kites in the center of campus, but the wind keeps blowing his kites into the trees. Professor Plum has spent enough time observing trees to notice that some are more even than others. Some have branches evenly spread throughout the tree; others, though, seem weighted down on one side with a disproportionate amount of branches.

Professor Plum often teaches Data Structures so he knows that the same thing can happen in binary search trees (BSTs). Recall that for each node in a Binary Search Tree (BST) all values in the left-subtree are < the node and all values in the right-subtree are > the node. The shape of a BST depends on the order in which values are added. For example, if we start with an empty BST and insert the sequence of values: 50, 70, 30, 80, 34, 32, 9, 47, 18, then we get the BST:



Typically, Professor Plum uses nodes with left and right "pointers/references" to implement a BST. However, we could store a BST in an array such that:

- the root is at index 0
- a node at index *i* would have its left child at index 2 * *i* + 1, and
- a node at index *i* would have its right child at index 2 * *i* + 2.

Notice that the above tree is annotated with the indices in square-brackets. e.g., 34 at index "[4]".

The above BST stored in an array (leaving blank unused slots) would be:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
50	30	70	9	34		80		18	32	47											. . .

Professor Plum likes the idea of using an array to store a BST instead of a linked implementation, but he is concerned about the array's size. He wants you to write a program to determine the largest index value needed for a given sequence of BST values. For the above input sequence, the program should report a maximun index of 10.

INPUT SPECIFICATION.
Each line of data consists of a sequence of positive integers to be stored in a BST with each sequence being terminated by the value 0. A line with a single 0 value indicates that there is no more data.

OUTPUT SPECIFICATION.
The output for each BST should appear in the same order as their respective input cases. The output should be in the format "`BST` *c* `has a maximum index of` *i*", where *c* is the case number starting at 1 and *i* is the maximum index of the BST stored in an array.

Sample Input

```
50 70 30 80 34 32 9 47 18 5 0
3 20 10 5 8 30 0
0
```

Output for the Sample Input

```
BST 1 has a maximum index of 10
BST 2 has a maximum index of 24
```

Problem E — Hardest Hangman

Professor Plums does not like to fly, so family vacations are always by car. To entertain his kids they often play the game *hangman* as they drive. You are probably familiar with the game *Hangman*, but the "standard" rules are as follows:

1. One player (Professor Plum) chooses a secret word, then writes out a number of dashes equal to the word length.
2. The other player (one of his kids) begins guessing letters. Whenever she guesses a letter contained in the hidden word, the first player reveals all instances of that letter in the word. Otherwise, the guess is wrong.
3. The game ends when either all the letters in the word have been revealed, or when the guesser has run out of guesses.

As his kids have gotten older Professor Plum needs to select more challenging words, but does not want to haul a thick dictionary on vacation. Thus, he wants you to write a program to determine the "hardest" words in the dictionary **for each length word** in the dictionary. The "hardest" word of a given length is defined as follows:

- analysis all the words in the dictionary to determine the overall frequencies for all 26 letters.
- the *difficulty* of a word is the sum of frequencies for each of it's letters. Suppose for example that an analysis of all dictionary words found frequencies of a's to be 8.167%, h's to be 6.094%, and t's to be 9.056%, then the word "that" would have a sum of frequencies of 32.373%. The "hardest" word is defined to be the word with the smallest sum of letter frequencies. If two words have equal "*difficulty*," then the word that's later alphabetically is harder.

WARNING: To avoid floating-point round off errors (and incorrect results), DON'T actually compute the frequency of each letter by (count of the letter *x* in all dictionary words) / (count of all letters in all dictionary words). Since the (count of all letters in all dictionary words) is the same for each letter, just use the (count of the letter *x* in all dictionary words) as an exact indication of letter *x*'s overall frequency.

INPUT SPECIFICATION.
The input consists of dictionary words with each word on a line by itself. The last line in the file contains a single asterisk "*". No word will exceed 35 letters in length and there will be less than 150,000 dictionary words.

OUTPUT SPECIFICATION.
One line of output should appear for word lengths of 1 up to the length of the longest dictionary word. Each line of output should start with "`Hardest word of length c:`", where *c* is a number. The rest of the line should be a single-space separator followed by either: the lower-case version of the hardest word, or the string "`No word of that length is in the dictionary!`".

Sample Input

```
a
bat
eel
i
tall
teal
you
zip
*
```

Output for the Sample Input

```
Hardest word of length 1: i
Hardest word of length 2: No word of that length is in the dictionary!
Hardest word of length 3: you
Hardest word of length 4: teal
```

Problem F — Highly Recursive Definition

Professor Plums likes recursion, but his students typically are confused by it. During a recent faculty meeting his mind wandered, and he invented the following recursive mathematical function, G(n):

$$G(n) = n \text{ for all value of } n \leq 0$$

$$G(n) = G(n-6) + G(n-4) + G(n-2) \text{ for all values of } n > 0.$$

He wants you to write a program to compute values of the function G(n).

INPUT SPECIFICATION.
The input consists of integer values of *n* with each integer on a line by itself. The last line in the file contains an integer 0 which is NOT to be included in the output eventhough G(0) is 0. All the integer values of n and G(n) will fit in a 64-bit integer variable.

OUTPUT SPECIFICATION.
Each line of input (except the 0) produces a line of output formatted as "`G(`*n*`) = `*x*", where *n* is the input number and *x* is the G(n) function value. (NOTE: there is one space on each side of the equal sign.)

Sample Input

```
10
-5
100
20
125
0
```

Output for the Sample Input

```
G(10) = -50
G(-5) = -5
G(100) = -40699755505816
G(20) = -1056
G(125) = -1816041553102722 59
```