Problem #1
STRING PROCESSING
Novice

Source file: problem1.extension

One day you wake up to find that you need to write a simple string processor. This simple string processor reads in a single line of text followed by a line of commands that will modify the string. Your program should output the line of processed text.

The commands are as follows:

- 0 - move cursor to the start of the line
- $ - move cursor to the position after the last character (end of line).
- x - delete the character at the cursor position if not at end of line.
- s - swap the character at the cursor with that to the right of it, as long as the cursor is not at or just before the end of the line.
- ix - insert the character 'x' at the cursor position, and move the cursor along one space.
- u - make the character go into upper case if a letter, and move the cursor along one space.
- + - move the cursor right one space.
- - - move the cursor left one space.

The commands are input as a single line following the input of the text line. Commands are not separated by spaces. The cursor starts at the first character in the line for each input line. The input is terminated by an input text line that contains only a single character, '#'.

INPUT: Your program should first accept, at a suitable prompt, a line of text of not more than 80 characters. Then your program should accept a command line of not more than 30 characters. Your program should continue to accept lines of text for process until the string ‘END’ is entered.

OUTPUT: Your program should output the text line after it has been processed.

EXAMPLE: Enter text line: **Hello, I am a frog.**
Enter command line**: $-----xxxxipieirisioin**
Result: Hello, I am a person.

Enter text line: **Needle nardle noo.**
Enter command line: **+++xizuu+++xxips**
Result: NeezLE napel noo.

Enter text line: **END**

Problem #2
POWERS AND PRIMES
Novice

Source file: problem2.extension

The French mathematician A. de Polignac made a remarkable observation. It seems that every odd number larger than one can be written as a sum of a power of two and a prime. For example,

$$53 = 2^4 + 37,$$
$$241 = 2^7 + 113, \text{ and}$$
$$999999 = 2^{16} + 934463$$

Unfortunately, this observation is not true for one odd integer $n < 3{,}000{,}000$.

The purpose of this problem is two-fold. First, find the odd integer *n* that does not satisfy dePolignac's observation. Second, given any odd integer *n* greater than 1, but less than 3,000,000, find the power of 2 and the prime when summed gives *n*.

INPUT: Your program should accept an integer value for *n* using a suitable prompt. Your program should continue to prompt for an integer and output the results until an entry of zero (0) terminates the program.

OUTPUT: First output the odd integer *n* that does not satisfy dePolignac's observation. Then for each integer *n* inputted, output the power of two and the prime that satisfies dePolignac's observation.

EXAMPLE: n = 779 does not satisfy observation. **** This is NOT the correct answer

Enter an integer: **53**
Power of two = 4, Prime = 37

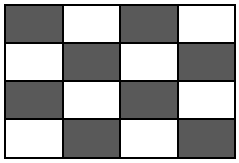
Enter an integer: **999999**
Power of two = 16, Prime = 934463

Enter an integer: **0**

Problem #3
SUM THE SQUARE
Novice

Source file: problem3.extension

Let n be an even positive integer. Fill in the cells of an nxn matrix with the numbers 1,2, …, n^2 row by row, from left to right. We will then assume the grid is colored in a checkerboard pattern, e.g.:



Checkerboard

You will compute the sum of the integers in the black squares and the sum of the integers in the white squares. Then you will take the difference between these two values (sum of black squares – sum of white squares). You will then output these values.

INPUT: Your program should accept an even integer value for n ($2 \le n \le 20$) using a suitable prompt. Your program should continue to prompt for an integer and output the results until an entry of zero (0) terminates the program.

OUTPUT: Your program should output the data in the following form:

Black Sum = XX
White Sum = XX
Difference = XX

Problem #4
TRIANGULAR NUMBERS
Novice

Source File: problem4.extension

The n^{th} triangular number is the number of symbols needed to construct a triangle with *n* rows. The 3rd triangular number is 6 because we can construct a triangle of 3 rows with 6 symbols (we are using asterisks) and would look like this:

```
  *
 * *
* * *
```

The 5th triangular number is 15 because we can construct a triangle of 5 rows with 15 asterisks and would look like this:

```
    *
   * *
  * * *
 * * * *
* * * * *
```

For this problem, you will read in a positive number, call it *n*. Then calculate the n^{th} triangular number and output that number to the screen.

INPUT: Your program should accept an integer value between 1 and 100, inclusive, using a suitable prompt. Your program should continue to prompt for an integer value and output the results until an entry of zero (0) terminates the program.

OUTPUT: Your output should be the n^{th} triangular number.

EXAMPLE:
```
Enter a number:  5
Triangular number is 15

Enter a number:  1
Triangular number is 1

Enter a number: 0
```

Problem #5
BALL BOUNCE
Novice

Source File: problem5.extension

When a ball is dropped from a height to a floor, it bounces back up a certain percentage of the height it was originally dropped from. And it continues bouncing back that percentage until is stops bouncing. The purpose of this problem is to determine the height that a ball achieves after its n^{th} bounce given the starting height and its percentage recovery. The following chart illustrates the height the ball achieves after each of 8 bounces when dropped from a height of 64 feet with a percent recovery of 50%:

Bounce	Height
1st	32ft.
2nd	16ft.
3rd	8ft.
4th	4ft.
5th	2ft.
6th	1ft.
7th	.5ft.
8th	.25 ft.
etc.	etc.

A ball with a percent recovery of 75%, when dropped from a height of 64 feet, will bounce to height of 48 feet after one bounce. We could even have superballs with a percent recovery greater that 100%.

INPUT: Your program should accept the height in feet, a percent recovery, and a bounce number in this order, using suitable prompts. Your program should continue until a value of zero (0) is entered for the height.

OUTPUT: Your program should output the height the ball achieves after the given number of bounces rounded to 2 decimal places

EXAMPLES: Enter starting height: **64**
Enter percent recovery: **50**
Enter bounces: **4**

On bounce 4, the ball will be at 4.00 feet.

Enter starting height: **125**
Enter percent recovery: **75**
Enter bounces: **5**

On bounce 5, the ball will be at 29.66 feet.

Enter starting height: **0**

Problem #6
STRING COMPRESSION
Novice

Source file: problem6.extension

Consider the string 'AAAABCCCCCDDDD' consisting of alphabetic characters only. This string is of length 14. Since the string consists of alphabetic characters only, duplicate characters can be removed and replaced with a duplication factor n. With this technique the string can be compressed and represented by '4AB5C4D', which means four 'A's, followed by a 'B', followed by 5 'C's, then 4 'D's. The compressed string is of length 7.

The purpose of this problem is to take a string in compressed form and recreate the original uncompressed string. The compressed string will be of the format 'nA...' where n, the duplication factor, is an integer between 2 and 99 and A is an uppercase alphabetic character. Single characters will not be prefixed with a duplication factor. For instance, the string 'AABCDE' would be compressed to '2ABCDE' (length 6), instead of '2A1B1C1D1E' (length 10).

INPUT: Your program should accept, at a suitable prompt, a compressed string of not more than 60 characters. Your program should continue to accept strings until the string 'END' is entered.

OUTPUT: Your program should output the uncompressed string, 40 characters per line (it may be necessary to break an uncompressed string over multiple lines).

EXAMPLE: Enter string: **3A4B7D**
AAABBBBDDDDDDD

Enter string: **22D7AC18FGD**
DDDDDDDDDDDDDDDDDDDDDDAAAAAAACFFFFFFFFFF
FFFFFFFFGD

Enter string: **END**

Problem #7
CROSS SUM
Novice

Source File: problem7.extension

A cross sum is formed from a set of 5 numbers. The following is the cross sum formed from the set {1,2,3,4,5}:

```
  4
5 1 2
  3
```

Notice that the sum of the numbers across equals the sum of the number down, in this case 8. Therefore, a cross sum can be represented by two subsets of numbers where each number is represented in one of the two subsets, but only one number is in both subsets, and the sum of each subset is equal. The two subsets for the above cross sum would be {1,3,4} and {1,2,5}.

The purpose of this problem is to compute the numbers in each subset that makes up a cross sum when given the set of 5 numbers and the value that each subset must sum.

INPUT: Your program should accept 5 numbers and a value at suitable prompts.

OUTPUT: Your program should output the two subsets of numbers that make up the cross sum in the form:

The two sets are {X, Y, Z} and {V, W, X}

The order of the numbers in each set is not important.

EXAMPLES:

Enter 5 numbers: **1 2 3 4 5**
Enter a sum value: **8**

The two sets are {4, 1, 3) and {5, 2, 1}

Enter 5 numbers: **18 16 19 8 26**
Enter a sum value: **53**

The two sets are {19, 8, 26) and {18, 19, 16}

Problem #8
POLAR EQUATIONS
Novice

Source File: problem8.extension

The purpose of this program is to plot a graph for the pair of polar equations:

$$x = r \cdot \cos\theta \text{ and } y = r \cdot \sin\theta ,$$

where $r = 10 \cdot (1 + 0.5 \cdot \cos\theta)$

for $0 \le \theta \le 2\pi$

We will plot points for θ values in this range, but only in increments of $2\pi/36$. Use π=3.14159.

The range for x and y are, respectively: $-15 \le x \le 15$ and $-12 \le y \le 12$. You should round the x and y values to the nearest integer. x is the horizontal axis and y is the vertical axis. Coordinates should be plotted on your graph as '*' (asterisks). You should plot a '+' for the origin, (0,0). If the origin is on the graph, an '*' would replace the '+'.

A sample plot may appear as:

```
          *
      *       *
  *           +           *
      *       *
          ***
          *
        *   *
```