Assignment 3 Strings

> ***Note:*** *During development, we recommend that you put your project folders under the* `source` *directory inside the assignment's root directory, so that they form a structure as described in the Submission Guidelines section. This is required for the redesigned judger to find your source files. See the Notes on Judger section for more information.*

> ***Note:*** *Each problem is shipped with 3 sample test cases which you can find under the* `data` *directory inside the assignment's root directory.*

> ***Note:*** *In each problem, your program's output may have extraneous spaces at the end of each line, and/or extraneous empty lines at the end of the output; they won't affect the correctness of your program. Be careful, however, to put the correct amount of spaces and new-lines (if any) at other positions in the output.*

Part 1 - Line Wrap

Background

In the earliest days of mainframe computing, electronic displays didn't exist. Instead, computers are equipped with *teletypewriters* (TTYs) as consoles to print messages to (and to read inputs from). A TTY would, upon receiving a character from the computer via a communication wire, print it on paper just as if one pressed a key on an ordinary typewriter.

Olivetti Teleprinter (courtesy of Wikimedia Commons)

On a typewriter, moving to a new line is a manual operation. The typewriter doesn't move to a new line automatically as the cursor reaches the end of a line; instead, the operator instructs the typewriter to do so by presses the "carriage return" and "line feed" keys manually when the cursor is near end-of-line. TTYs are similar: they require the running computer program to break up into lines whatever it wants to print out by inserting new-line characters where appropriate. This is called "line wrapping".

In today's computing, line wrapping is still an important operation:

- Your console emulator (e. g. `conhost.exe`) now does the line wrapping for you.

- In almost all graphical applications (like web browsers and word processors), line wrapping is done to fit text into fixed-width text areas.

Problem description

You're tasked to write a program that line-wraps a given piece of text to fit a specified display width W (in characters) by inserting a new-line character (`'\n'` in C++) *between words* according to the following rules:

1. At any point in time, you must attempt to fit as many words into the current line as possible without inserting a new-line.
    - A *word* is defined as a sequence of non-whitespace characters (including punctuations like `.` and `"`).
2. Insert a new-line if trying to fit the next word into the current line would make the length of the current line exceed W.
    - You must not insert new-lines inside a word.
3. If you encounter a word whose length on itself already exceed W, place it on its own line and call it a day.

> ***Note:*** *Do not insert any new-line inside the word even in this case.*

Input/output specification

The input to your program will look like

```
23

Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.

Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat.
```

The first line (`23` in this case) is the specified W. The rest of the input is the piece of text for your program to line-wrap. The output of your program to this input should be

```
Lorem ipsum dolor sit
amet, consectetur
adipiscing elit, sed do
eiusmod tempor
incididunt ut labore et
dolore magna aliqua. Ut
enim ad minim veniam,
quis nostrud
exercitation ullamco
laboris nisi ut aliquip
ex ea commodo
consequat.
```

You can see the provided test cases for more examples of valid input and output.

> ***Note:*** *The input text may contain whitespace characters other that space (e. g. new-lines or tabstops), or sequences of multiple consecutive whitespace characters.*
>
> - *You should treat any sequence of one or more whitespace characters in the input identically, the same as you treat a single space.*
> - *In your output, each pair of adjacent words must be seperated either by a single space or by a single new-line, determined solely by the rules stated in the problem description.*

> ***Hints:***
>
> - *You can repeatedly read in words from the input until the end by using the loop*
>
>   ```
>   string word;
>   while (cin >> word) {
>       // do something with word...
>   }
>   ```
>
>   *The loop will stop when there are no more words in the input. Similarly, you can read in lines repeatedly by using the loop*
>
>   ```
>   string line;
>   while (getline(cin, line)) {
>       // do something with line...
>   }
>   ```

and the loop will stop when there are no more lines in the input.

- *When you run your program on a console, you can mark the end of your input by*
  - *on Windows: inputting* `Ctrl`+`Z` *followed by* `Enter`*, or*
  - *on macOS or Linux: pressing* `Ctrl`+`D`*.*
- *When you run your program on a console, you may observe that your program starts outputting as soon as you input a line. As a result, the input and the output are intermixed in the console window, and copy-pasting the output out becomes difficult.*

  If this happens, you should redirect your output to a file. This can be done on the Windows command line using the syntax

  ```
  :: on Windows:
  .\1_wrap.exe >output.txt
  ```

  ```
  # on macOS or Linux:
  ./1_wrap >output.txt
  ```

  or in the program by adding the statement

  ```
  freopen("output.txt", "w", stdout);
  ```

  to the start of your `main()` *function. This will allow you to open* `output.txt` *later to see your program's output.*

  - *If you use the second method, be sure to remove that statement before submitting your work.*
- *When you inspect your program's output in a text editor, you may notice that placing the cursor at the end of some lines results in the editor showing the cursor to be at column $W + 1$. This is expected: it merely states that "if you're to append any character to this line any more, that character would be at column $W + 1$", i. e. that line is currently exactly $(W + 1)$-character long.*

Part 2 - Identifierify

Background

In most programming languages, not all sequences of characters can be valid identifiers (names of variables, types, etc); in C++, for example, valid identifiers must conform to the following rules:

- The first character must be an underscore (`_`) or a letter.
- All following characters (if any) must be an underscore or be alphanumeric.

Notably, spaces, hyphens and apostrophes, all common to English phrases, must not occur in an identifier. This has the implication that, if you want to express intentions with more than one word, you'll have to adopt a convention for transforming multi-word phrases into valid identifiers. For example, to express that a variable is used for storing the "saved GPIO pull-up state", you might choose to call the variable one of:

- `SavedGPIOPullUpState`, in "PascalCase"
- `savedGpioPullUpState`, in "camelCase"
- `saved_gpio_pull_up_state`, in "snake_case"

Those are 3 popular *casing conventions* used in programming:

- PascalCase is popular among object-oriented languages (Java, Python, etc) for class names. It capitalizes each word in the phrase and joins them head-to-tail.
    - If a word in the original phrase contains an uppercase letter in non-initial position, it is left as-is in the resulting string.
- camelCase is popular among various languages (Java, JavaScript, etc) for variable and function names. It capitalizes every word in the phrase *except the first one*, and then joins them head-to-tail.
    - If a word in the original phrase contains an uppercase letter in non-initial position, it is *turned lowercase*.
- snake_case is also popular (in Python, Rust, etc) for variable and function names. It changes every letter in the phrase to lowercase, and joins them with underscores (`_`).

Problem description

You're tasked to write a program that reads in a line as a string, and outputs the string transformed into PascalCase, camelCase and snake_case.

For the purpose of this problem:

1. A *word* is defined as a sequence of alphanumeric characters.
2. A *valid identifier* is a sequence of alphanumeric characters and underscores.

> ***Note:*** *in this problem, we allow identifiers to begin with digits to simplify the problem a bit.*

Input/output specification

The input to your program will consist of a single line of text. An example is

```
saved GPIO pull-up state
```

> ***Note:*** *the input line may contain spaces at the beginning and the end, which you should ignore.*

The output of your program should consist of 3 lines in the format of

```
PascalCase: SavedGPIOPullUpState
camelCase:  savedGpioPullUpState
snake_case: saved_gpio_pull_up_state
```

with each line stating the casing convention being used, followed by the generated identifier in that casing convention.

> ***Note:*** *be careful to put the correct amount of space after each colon (`:`).*

You can see the provided test cases for more examples of valid input and output.

Part 3 - Fuzzy Search

Background

Suppose you want to search a piece of text for a word or phrase, but you don't quite know the exact spelling of it. For example, you may want to search for "separate" but accidentally enter "seperate"; a naïve search will not turn up anything except for where the original text uses the exact same misspelling.

We can use a kind of "fuzzy search" algorithm for this case. A "fuzzy search" returns not only exact matches, but any location in the text being searched that is deemed "good enough" a match -- for example, a single-letter mismatch like "analyze" vs "analyse" may still be considered "good enough" for a match. Of course, the definition of "good enough" is subject to numerous variations.

Problem description

You're tasked to develop an extremely simple fuzzy-search algorithm. This algorithm works as follows:

1. Define the *correlation* between two strings of the same length as the number of characters they share at the same places.
    - Example: the correlation between `onion` and `olive` is 2 because only their first and third characters are the same:

      ```
      onion
      +-+-- 2/5
      olive
      ```

    - Example: the correlation between `HELLO` and `hello` is 0 because none of their constituent characters are the same.
2. Define the *maximum correlation* of a string S in a piece of text T as the maximal correlation between S and all substrings of T whose length equals that of S.
    - Example: the maximum correlation of `till` in

      ```
      the ill-fated, tilted tiling machine
      ```

      is 3:

```
the ill-fated, tilted tiling machine
   -+++ 3/4
   till
```

```
the ill-fated, tilted tiling machine
               +++- 3/4
               till
```

```
the ill-fated, tilted tiling machine
                     +++- 3/4
                     till
```

- Example: the maximum correlation of `till` in

  ```
  the ill-fated, tilted distilling machine
  ```

  is 4:

  ```
  the ill-fated, tilted distilling machine
                           ++++ 4/4
                           till
  ```

3. Given a line of text T and a string S, return all *best matches* of S in T, where a *best match* is defined as a substring of T of the same length as S whose correlation with S equals the maximum correlation of S in T.

Input/output specification

The input to and output of your program should look like:

- Input:

  ```
  the ill-fated, tilted tiling machine
  till
  ```

  Output:

  ```
  the ill-fated, tilted tiling machine
     -+++ 3/4
  ```

```
   till
   ^ col 4
---
the ill-fated, tilted tiling machine
               +++- 3/4
               till
               ^ col 16
---
the ill-fated, tilted tiling machine
                     +++- 3/4
                     till
                     ^ col 23
---
Found 3 best match(es)
```

Following are precise prescriptions of the input and output formats.

The input consists of 2 lines. The first line is the text T to search in. The second line is the string S to fuzzy-search for.

> ***Note:*** *S may not be a single word; it may also contain spaces at the beginning and/or the end.*

The output should describe all best matches of S in T in the following format:

1. For each best match M whose first character is at column C of T (the leftmost character of T is considered to be at column 1), print a section in the following format:
    a. Print T verbatim as the first line.
    b. On the second line starting at column C, print a string consisting of N `+` or `-` characters, where N is the length of S. At column $C + i$, Output a `+` if M and S share the same $(i + 1)$-st character; output a `-` otherwise.
    c. Still on the second line, append a space, and then append

        ```
        {c}/{N}
        ```

        where you substitute `{c}` with the correlation between M and S, and `{N}` with N.
    d. On the third line starting at column C, print S verbatim.

e. On the fourth line starting at column C, print

```
^ col {C}
```

where you substitute C for `{C}`.

f. On the fifth line, print the string `---`.

2. After printing out all best matches, print a line in the format of

```
Found {n} best match(es)
```

where you substitute the total number of best matches for `{n}`.

You can see the provided test cases for more examples of valid input and output.

Part 4 - Syllabic Palindromes

Background

Consider the word `minami`: it's not a palindrome, however it's an example of what we shall call "syllabic palindromes" here: it consists of the syllables

```
mi na mi
```

regardless of whether you pronounce it forward or backwards. On the other hand, the word `okiko` is not a syllabic palindrome because it consists of the syllables

```
 o ki ko
```

which wouldn't be the same if reversed, even if `okiko` is a palindrome in the usual sense.

Problem description

You're tasked to write a program that determines whether a valid word is a syllabic palindrome.

For the purpose of this problem, a *valid word* is a string with following properties:

1. The string results from concatenating one or more syllables head-to-tail.
    - Examples: `a`, `aa` and `kokoro` are all valid strings; `+`, `a e` and `a_ya_ka` are not.
2. Each *syllable* has the structure of $\mathbf{V}$ or $\mathbf{CV}$, where $\mathbf{V}$ is one of `a`, `e`, `i`, `o` or `u`, and $\mathbf{C}$ is a lowercase letter that isn't any of them.
    - Examples: `a`, `ka` and `xa` are all valid syllables; `A`, `Ka`, `y`, `an` and `cha` are not.

Output `YES` if the input valid word is a syllabic palindrome, or `NO` otherwise.

Input/output specification

The input consists solely of a valid word as determined by the rules stated in the problem description.

The output consists solely of either the string `YES` or the string `NO`.

Examples:

- Input: `a`

  Output: `YES`
- Input: `minami`

  Output: `YES`
- Input: `okiko`

  Output: `NO`

You can see the provided test cases for more examples of valid input and output.

> ***Hints:*** *Before you get surprised by the output of your program and set out to debug it, be sure that you input a valid word as determined by the rules stated in the problem description.*
>
> > *The reason is because it doesn't matter at all what your program does when given invalid input. Your program may output* `YES` *against the input* `345`*, it may output* `NO` *against* `chacha`*, against* `I am AI` *it may evaluate* `uniform_int_distribution<>(0, 1)(random_device())` *and determine the output from that, or against* `evil` *it may not print anything, but delete* `C:\Windows\System32` *and then make the computer explode: we won't judge your program on any of those inputs. What it must do is consistently give correct outputs to valid inputs.*
>
> - *You can quickly determine whether a string is a valid word using the regular expression*
>
>   ```
>   ([b-df-hj-np-tv-z]?[aeiou])+
>   ```
>
>   *An example of using it in Python would be*

```
>>> import re
>>> vw = re.compile(r"([b-df-hj-np-tv-z]?[aeiou])+")
>>> vw.fullmatch("kokoro") # a re.Match object being
returned indicates successful matching
<re.Match object; span=(0, 6), match='kokoro'>
>>> vw.fullmatch("a_ya_ka") # None being returned
indicates failure to match
>>>
```

Submission Guidelines

Your submission should be a ZIP file with the following structure:

```
<your student number>.zip
|- 1_line_wrap           // Part 1
|  |- main.cpp
|
|- 2_identifierify       // Part 2
|  |- main.cpp
|
|- 3_fuzzy_search        // Part 3
|  |- main.cpp
|
|- 4_syllabic_palindrome // Part 4
   |- main.cpp
```

Notes on Judger

A number of changes have been made to the judger for this assignments:

- The command line interface has been revamped significantly. You can now judge all problems against all sample test cases simply by running the judger without any command line options, or judge `1_line_wrap` against all sample test cases by passing a single option `-1`.

  > ***Note:*** *run the judger with* `-h` *to see all accepted options.*

- A number of quality-of-life improvements have been made:
  - A help message has been added for link errors.
  - When your program produces incorrect output, the judger now prints out the difference between the correct output and your program's output in Python's `difflib` style.
  - When your program produces incorrect output, the judger now consistently puts that output inside the assignment's root directory after names like `1_line_wrap.3.out-wrong.txt`.
- Certain functionalities have been removed:
  - You can no longer specify your own input and output files by path; the judger always uses the files provided under the `data` directory.
  - You can no longer specify your own location where you put your source files; the judger only tries to find them under the `source` directory, and only if they conform to the structure described in the Submission Guidelines section.

  > ***Note:*** *As a consequence, you must leave the judger script under the assignment's root directory, so that it can find the* `data` *and* `source` *directories.*