EXERCISE SET 2

PROGRAMMING WORKSHOP I, ACADEMIC YEAR 2019-2020

Deadline: 8/12/2019, 23:59

Read before you start

Read the ENTIRE pronunciation carefully and "draw" your program on paper.

Decide what variables you will need, what names you will give them, whether constants are needed and if so for what quantities, what control structures to use for each function, and what additional functions are needed. Each time you complete a stage, confirm that your program is working properly before moving on.

If you need clarification or have problems, send a message to the e-class discussion area. Warning: posting code to eclass is not allowed.

This task can be done in groups of up to 2 people. You don't have to be in a group with the same person you are in the lab with or who did the previous assignment. You can discuss the exercises with your fellow students but you are not allowed to exchange code in any way.

If you work in a group, then before starting work log into Autolab, select **hw1** and build a group via **Group Options.** Don't leave this process for the last minute!

Start early! Planning is always MUCH more time consuming than you expect.

Late submissions of exercises are not accepted.

Your exercises will be graded on the following (in no particular order):

• Correctness of calculations
• Correct use of pointers, functions and arrays •
Effective use of appropriate structures, variables, constants, etc. • General
program formatting (alignment, variable and constant names, etc.) • Compliance with
specifications • Effective comments

The use of goto, the use of gets and the use of global variables are strictly prohibited.

You can assume that keyboard input will always be given in the correct format.

Reading text, storing it in memory and processing it In this task you will write a program that reads a text word by word and:

1. Stores each word of the text in a two-dimensional array of characters. If a word appears more than once in the text, it is stored once in this table.

2. Stores the order in which the words appear in the text in a separate two-dimensional index array.

In summary, the functions specified by this work are the following:

1. Print the text, using specific rules as to its ending formatting.

2. Create a histogram. For each word, the number of occurrences in text.

3. Calculation of the space occupied in the memory in the following two alternative cases of storing the information:

   - Each word of the text is stored only once in the character table. At the same time, we maintain a table of indexes that determines the order of words within the text.

   - The text is stored in its entirety word by word in a character array

4. Replacing some of the words in the text with their synonyms. Word-synonym matching is provided by a two-dimensional array of characters that is given to you ready-made.

Code provided ready-made You are provided with the static library **libhw2.a,** the file **hw2.h,** which you are not allowed to change, and the skeleton file **hw2.c** in which you are asked to complete your code. The code you write should be contained within the **hw2_main function.** The **main** function is given to you ready-made and should not be changed.

The library **libhw2.a** contains the implementations of the functions:

- **init_stack_size:** Configures the stack size so that the task works for large text files. The function is called inside the **main function.**

- **read_synonyms:** Reads word and synonym matches from a file. More information about the function can be found in step 4.

To compile your program, make sure the file **libhw2.a** is in the same directory as **hw2.c, hw2.h** and write:

```
gcc -Wall -g hw2.c -o hw2 -lhw2 -L.
```

Stage 0 - Reading execution arguments/options from the command line

The program you will write accepts from the command line the arguments with which it will be executed and from the established input (with redirection to a file) the text to be edited. Valid options that can be given from the command line are:

- **-p:** Prints the final text.
- **-h:** Computes the histogram of the text words and prints it in standard exit.
- **-r <filename>:** Finds each word in the text contained in the synonym table and maintains a pointer to its synonym. It requires as an additional parameter the filename **<filename>** from which the synonyms will be read. <filename> **must** follow the **-r** parameter .

The options may be given in any order, but the order in which the corresponding functions are performed (regardless of the order in which they are given by the user on the command line) is as follows: **-h ÿ -r ÿ -p**

For example, if the user gives the **-p -h** options it , then the function that will be executed first matches the **-h** option and then the function that matches the **-p option.**

One or more of the above command line options should be provided when running the program. If no option is provided or even one invalid option is provided, the message **"Incorrect command-line arguments!"** is printed. followed by a newline character and the program terminates.

Stage 1 - Reading the text from the established input

In the **hw2_main** function define the following tables:

1. A two-dimensional array of characters with **MAX_WORDS** number of rows and number of columns **MAX_WORD_LEN.** Initialize the array to contain the '\0' character in all its positions.

2. A two-dimensional array of pointers to character with **MAX_WORDS** number of rows and number of columns 2. Initialize the array to contain NULL in all its positions.

Write a function that takes as parameters a two-dimensional array of characters with column number **MAX_WORD_LEN** and a two-dimensional array of pointers to character with number of columns 2 and has a return type of **void.**

The function reads a text word by word from the specified input until either the array is full or the input is finished. Each word of the text is separated from the previous and the next by white characters.

For each word read, the function does the following:

1. Converts all uppercase characters in the word to lowercase. For character conversions you can use appropriate functions from ctype.h

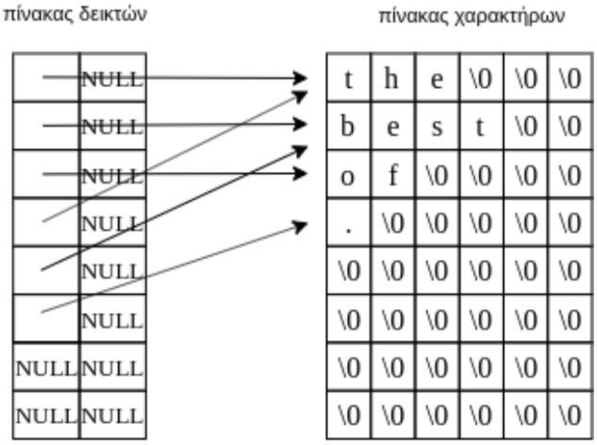
2. Checks if the word exists in the character array. One word per line (regardless of its length) is always entered in the character table. Assume that the table has enough columns to store any word of the text.

- If the word exists in the character table (already read once), it finds its address in the table.
- If the word does not exist in the character array, adds it to the first empty line of the array.

3. Adds a new entry to the index table so that the last non-empty position of the first column of the index table points to the last word read from the text and stored in the character table (either a new entry or an existing one) .

Note #1: To simplify the editing required, punctuation marks **(. , ; ?)** appear as separate words within the text (space between a punctuation mark and the preceding and following words).

For example, for the text **"The best of the best ."** the character array and pointer array are configured as in the following figure.



Write code in **hw2_main** that reads a text verbatim from standard input, using the function you just wrote. The function takes as parameters when it is called the 2D array of characters and the 2D array of pointers you defined at this stage.

In this phase the second column of the index table is not used, the elements of which simply remain initialized to NULL.

Note #2

When you are typing the words of text (and not giving them from a file via conventional input redirection) then you must type control-D to signal the end of the text (input).

Stage 2 - Print the text

Write a function that takes as parameters a two-dimensional array of pointers to character with number of columns 2 and an integer **linelen** and has a return type of void. The function should print to standard output the text referenced by the array of pointers, as follows:

- Prints a newline character before the first word of the text.
- The first word of the text and every word after a period, exclamation mark or (English) question mark begins with a capital letter.

- Words and punctuation marks are separated by a space character.
- As an exception to the above rule, if the next word to be added leads to a line longer than **linelen** characters, a newline character is printed instead of a blank character, followed by the next word (on a new line). You can assume that **linelen** will always be greater than or equal to **MAX_WORD_LEN.**

The process is repeated for the entire text referenced by the table, so that all lines have the maximum possible length of words (without breaking them), but no line exceeds linelen **characters .** A newline character is printed at the end.

Write code in **hw2_main** which, if the user gives the **-p parameter on the command line,** calls the above function with parameters the array of pointers defined / populated in the previous step and the **MAX_LINE_LEN** integer specified in the file **hw2.h .**

Stage 3 - Calculation of the histogram

In the function **hw2_main** define an array of integers of size **MAX_WORDS** and initialize it to contain in all its positions the value 0. Each position of the array corresponds to the number of occurrences of the word which is in the corresponding position of the character array.

Write a function that takes the following two parameters: a) a two-dimensional array of pointers to character with column number 2 and b) a one-dimensional array of integers. The function has a **void return type.**

The function is called with parameters the array of pointers of stage 1 and the array of integers you defined in the current stage and stores in the array of integers the number of occurrences of each word in the text.

Write another function that takes as parameters a two-dimensional character array with **MAX_WORD_LEN** columns and an array of integers. The function is called with arguments the array of characters defined in stage 1 and the array of integers you defined in the current stage and returns nothing. The function outputs the histogram as follows:

Prints a newline character.

It loops through the character array and for each word prints the word to occupy **MAX_WORD_LEN** positions and be right-aligned, colon, space and its number of occurrences as follows:

- Prints as many '$' characters as there are thousands of occurrences of the word in the text.
- As long as we subtract the thousands printed above, it prints that many '#' characters, as many hundreds of times the word appears in the text.
- After subtracting the thousands and hundreds printed above, it prints as many '@' characters as there are dozens of times the word appears in the text.
- After subtracting the thousands, hundreds, and tens printed above, it prints as many '*' characters as the word appears in the text.

This "encoded" printout of the number of occurrences of each word is followed by a newline character.

For example if the word "the" occurs 2146 times it will print:

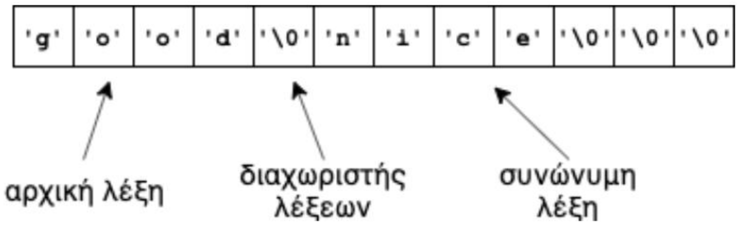
the: $$#@@@@****

After implementing the above, fill in the part of your code that checks if the -h parameter was given from the command line and takes the corresponding actions.

Stage 4 - Finding synonyms

At this stage you will use the ready-made function **read_synonyms** with the first argument the name of a file and the second argument a two-dimensional array of characters in which to store synonym information. On success, the function returns 1, otherwise (eg if the file name is incorrect) it returns 0.

The function assumes that the file contains on each line two words with a space between them, the second of which is a synonym of the first. The function reads the contents of the file and stores in each line of the table pairs of words and their synonyms, as shown in the figure below. In each entry, the initial word is separated from its synonym by a '\0' character which serves at the same time as the terminal of the initial word, while the synonymous word follows in the next positions of the table and is also terminated by a '\0' character. Any lines at the end of the table that do not contain information contain the character '\0' in all their positions.

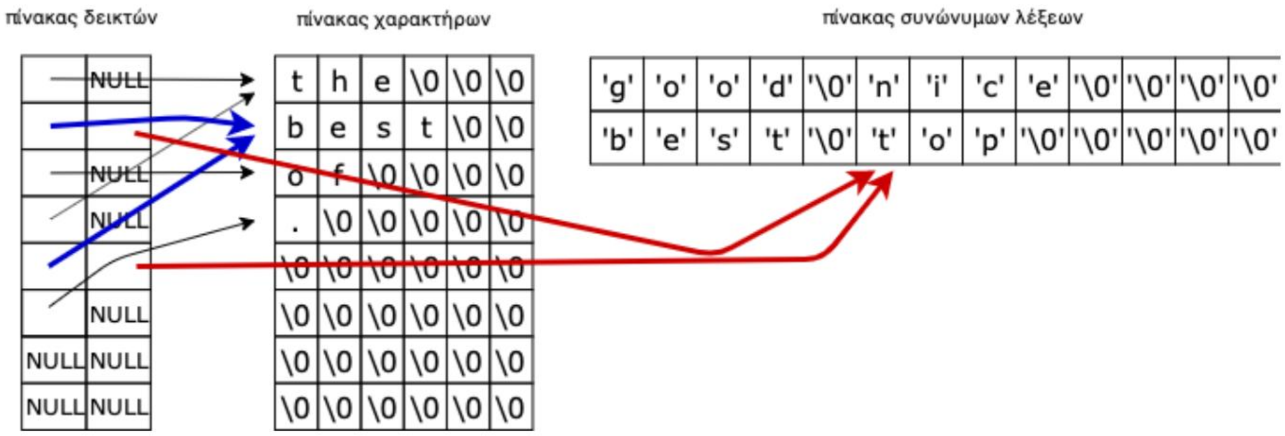


'g'	'o'	'o'	'd'	'\0'	'n'	'i'	'c'	'e'	'\0'	'\0'	'\0'

Create a two-dimensional character array with **MAX_SYNONYMS** rows and **MAX_WORD_LEN*2** columns that will be used to store word synonym information.

Write a function that takes as parameters a two-dimensional array of pointers to character with column number 2 and a two-dimensional array of characters containing the synonyms and has a return type of **void.** The function is called with parameters the 2D array of pointers you defined in step 1 and the character array containing the synonyms you defined above.

The function works like this: By traversing the first column of the index table, it looks to see if each word that the index pointers point to is in the initial words of the synonym table. If such a word is found, the address of the synonymous word is stored in the second column of the current row of the index table, otherwise the original value is kept
NULL. The figure below shows how the previous example is formed if for the word **best** the synonym is **top.**



Write code that, when the user specifies the **-r** option on the command line followed by the name of a file, does the following:

1. Calls the **read_synonyms** function giving the file name as the first parameter and as the second parameter the table of synonyms.
2. If the call is successful, it calls the function you defined in the current step to update the index table with the synonyms that appear in the text.

Update the step 2 print function so that if a word has a synonym in the second column of the index table, then after the word is printed, its synonym is printed between parentheses and following the rules described in step 2 for spacing between words , for line length and for capitalization.

Stage 5 (Optional) - Calculate storage costs

We want to calculate the total cost of storing the information (of the original text without taking into account the synonym information) for the following two alternative storage methods:

A. By the way we store the information as described above, that is, each word appears only once in the two-dimensional character array. For each word we keep one or more entries in the index table, depending on the number of occurrences of that word in the text. In this case we add the lengths of the words stored in the character array and the number of entries in the pointer array times the size of a pointer.

B. If we hypothetically stored each word in the text on a separate line of the two-dimensional character array, and each word appeared in it as many times as it occurs in the text. In this case, the storage cost is obtained if we sum the lengths of the words stored in the two-dimensional array.

Write a function that takes as parameters a two-dimensional array of characters with column number **MAX_WORD_LEN** and a two-dimensional array of pointers to character with number of columns 2 and has a return type of **long int.** The function returns the size that the text occupies if stored with the hypothetical storage method A.

Similarly, write another function that takes the same parameters and has the same return type with the previous function and returns the size that the text occupies if stored with the B storage method.

Write a third function that takes the same parameters as the previous two functions and has a return type of **double.** The function is called with parameters the array of characters and the array of pointers defined in step 1 and prints the ratio of the size of memory occupied if we choose method A as the storage method compared to method B.

Once you have implemented the above functions, appropriately pad the part of your code that checks the program's arguments to support the additional **-s option.** In the case that this option is given, a newline character is printed as the last action of the program, the above ratio with a precision of 3 decimal places and a newline character.

Think: Which of the two storage methods takes up the least space? Does the size and/or type of text affect the result? Can you explain why this is?

Guidelines for submitting work to Autolab

1. Comment out the full names and AEMs of the team members at the beginning of the **hw2.c** file. Please write comments in English characters ONLY.
2. Make a directory named **hw2submit** and copy **hw2.c** into it
3. Make a text file named **team.txt** and add to it the full names and AEMs of the team members, even if the team consists of one person.
4. Right-click the **hw2submit** directory and select **Compress here as tar.gz.** I will a file named **hw2submit.tar.gz will be created.**
5. Login to Autolab and select **hw2.**

   ÿ If you are a group of two and have not already done so, create a group via **Group Options.**

6. **Submit** hw2submit.tar.gz . _