

Міністерство освіти і науки України
Національний технічний університет України «КПІ ім. Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Мультипарадигмне програмування

ЗВІТ

до лабораторної роботи № 1
«Імперативне програмування»

Виконав
студент

ІТ-04 Москалюк Данило Євгенович
(№ групи, прізвище, ім'я, по батькові)

Прийняв

ас. Очеретяний О. К.
(посада, прізвище, ім'я, по батькові)

1. Завдання лабораторної роботи

Практична робота складається із трьох завдань, які самі по собі є досить простими. Але, оскільки задача - зрозуміти, як писали код наші славні пращури у 1950-х, ми введемо кілька обмежень:

- Заборонено використовувати функції
- Заборонено використовувати цикли
- Для виконання потрібно взяти мову, що підтримує конструкцію GOTO

Завдання 1:

Обчислювальна задача тут тривіальна: для текстового файлу ми хочемо відобразити N (наприклад, 25) найчастіших слів і відповідну частоту їх повторення, упорядковано за зменшенням. Слід обов'язково нормалізувати використання великих літер і ігнорувати стоп-слова, як «the», «for» тощо. Щоб все було просто, ми не піклуємося про порядок слів з однаковою частотою повторень. Ця обчислювальна задача відома як **term frequency**.

Ось такий вигляд матимуть ввід і відповідно вивід результату програми:

Input:

```
White tigers live mostly in India
Wild lions live mostly in Africa
```

Output:

```
live - 2
mostly - 2
africa - 1
india - 1
lions - 1
tigers - 1
white - 1
wild - 1
```

Завдання 2:

Тепер, нам потрібно виконати задачу, що називається словниковим індексуванням. Для текстового файлу виведіть усі слова в алфавітному порядку разом із номерами сторінок, на яких Ці слова знаходяться. Ігноруйте всі слова, які зустрічаються більше 100 разів. Припустимо, що сторінка являє собою послідовність із 45 рядків. Наприклад, якщо взяти книгу *Pride and Prejudice*, перші кілька записів індексу будуть:

```
abatement - 89
abhorrence - 101, 145, 152, 241, 274, 281
abhorrent - 253
abide - 158, 292
```

2. Опис використаних технологій

Для виконання даної лабораторної роботи використовувалася мультипарадигменна мова C#, яка також підтримує конструкцію GOTO.

3. Опис програмного коду

1) task1.lang_with_go_to

Спочатку, задаємо масив усіх стоп-слів які ми будемо ігнорувати ('for', 'the', 'in' та інші) та оголошуємо різноманітні змінні для відслідковування слів у тексті: індекс початку та кінця слова, кількість знайдених слів, булеві змінні 'foundABeginningOfAWord', 'isAStopWord', ймовірна кількість слів у тексті та інші. Алгоритм являє собою посимвольне зчитування тексту на вхід. Починається все з пошуку початку поточного слова за допомогою ascii-значень символів – літери латинського алфавіту знаходяться у проміжку від 65 до 122, тому, значення поточного символу повинно відповідати вище згаданому проміжку. Крім того, минулий символ повинен бути у проміжку від 0 до 65 (майже усі значення, які не відповідають літерам) або у проміжку від 90 до 97. Таким чином, ми знаходимо початок слова та переходимо до пошуку кінця. Умовою кінця слова є те, щоб поточний символ знаходився у проміжку від 64 до 123, наступний символ у проміжку від 0 до 65 або від 90 до 97. Після знайдених значень початку та кінця слова починаємо посимвольно будувати знайдене слово, паралельно роблячи його lowercased (якщо значення поточного символу від 65 до 90, додаємо до нього 32), потім звіряємо знайдене слово з елементами масиву стоп-слів та додаємо його до масиву усіх слів. Таким чином, повторяємо цю операцію для усіх слів у нашому тексті. Після того, записуємо усі унікальні слова в окремий масив та паралельно звіряємо знайдені слова з тим масивом: якщо поточне слово вже є в масиві унікальних слів, то ми збільшуємо значення у відповідному масиві 'wordsFrequency'. Після того, сортуємо масив 'wordsFrequency' завдяки алгоритму 'Bubble sort' та паралельно записуємо відсортовані індекси масиву унікальних слів в окремий масив 'sortedIndices'. Потім виводимо значення елементів масиву унікальних слів за спаданням їхньої частоти появи у тексті.

2) task2.lang_with_go_to

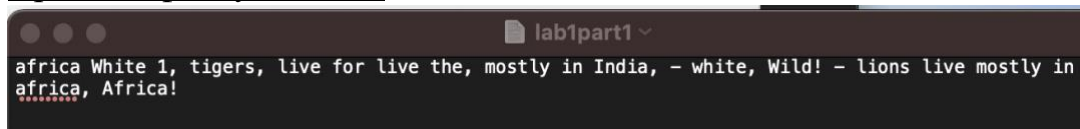
Спочатку, задаємо масив усіх стоп-слів які ми будемо ігнорувати ('for', 'the', 'in' та інші) та оголошуємо різноманітні змінні для відслідковування слів у тексті: індекс початку та кінця слова, кількість

знайдених слів, булеві змінні 'foundABeginningOfAWord', 'isAStopWord' та інші. Як і в минулому завданні, по тій же логіці, ми знаходимо усі слова в нашому тексті (не враховуючи стоп-слова), у книзі 'Pride and Prejudice' їх близько 95000. В середньому, на стандартній сторінці у 1800 символів розміщується 235-255 слів, у нашому випадку – одна сторінка рівна 245 словам. Починається все з оголошення зубчатого масиву (масиву масивів) для зберігання номерів сторінок для кожного з унікальних слів. Потім, починаємо аналіз наших знайдених слів – якщо слово зустрічається вперше – ми додаємо його до масиву унікальних слів та зубчатого масиву з номерами сторінок, як і було сказано раніше – номер сторінки вираховується наступним чином: поточний індекс слова / всього слів у сторінці (245) + 1. Якщо ж слово зустрічається не вперше, то ми додаємо номер сторінки до масиву номерів сторінок під індексом слова в масиві унікальних слів. Потім, завдяки алгоритму 'Bubble sort' сортуємо посимвольно слова та заодно масиви з номерами сторінок, щоб індекс слова в масиві унікальних слів відповідав індексу з його номерами сторінок у масиві 'pagesForWords'. Після того, будуємо строку виводу із номерами сторінок та підраховуємо їхню кількість – якщо їх більше 100 – поточне слово ігнорується та відбувається перехід до наступної ітерації.

4. Скріншоти роботи програмного застосунку

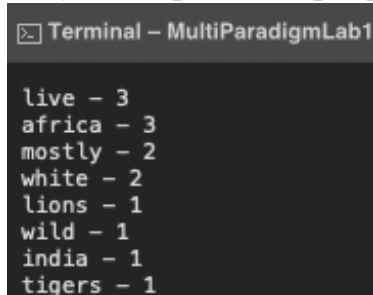
1) task1.lang_with_go_to

Приклад файлу на вхід:



```
lab1part1
africa White 1, tigers, live for live the, mostly in India, - white, Wild! - lions live mostly in
africa, Africa!
```

Результат роботи програмного застосунку:



```
Terminal - MultiParadigmLab1
live - 3
africa - 3
mostly - 2
white - 2
lions - 1
wild - 1
india - 1
tigers - 1
```

2) task2.lang_with_go_to

Приклад файлу на вхід (книга Pride and Prejudice):

```
prideAndPrejudice.txt
chapter 1
It is a truth universally acknowledged, that a single man in
possession of a good fortune, must be in want of a wife.
However little known the feelings or views of such a man
may be on his first entering a neighbourhood, this truth is
so well fixed in the minds of the surrounding families, that
he is considered the rightful property of some one or other
of their daughters.
'My dear Mr. Bennet,' said his lady to him one day, 'have
you heard that Netherfield Park is let at last?'
Mr. Bennet replied that he had not.
'But it is,' returned she; 'for Mrs. Long has just been here,
and she told me all about it.'
Mr. Bennet made no answer.
'Do you not want to know who has taken it?' cried his
wife impatiently.
'YOU want to tell me, and I have no objection to hearing it.'
This was invitation enough.
'Why, my dear, you must know, Mrs. Long says that
Netherfield is taken by a young man of large fortune from
the north of England; that he came down on Monday in a
chaise and four to see the place, and was so much delighted
with it, that he agreed with Mr. Morris immediately; that
he is to take possession before Michaelmas, and some of his
servants are to be in the house by the end of next week.'
'What is his name?'
'Bingley.'
'Is he married or single?'
'Oh! Single, my dear, to be sure! A single man of large
fortune; four or five thousand a year. What a fine thing for
```

Результати роботи програмного застосунку:

Початок:

```
abatement - 113,
abhorrence - 129, 188, 199, 322, 364,
abhorrent - 337,
abide - 267,
abiding - 211,
abilities - 79, 81, 123, 183, 284,
able - 18, 37, 62, 87, 93, 97, 99, 104, 111, 115, 124, 126, 127, 139, 147, 152, 153, 169, 171, 179, 184, 206, 211, 213, 219, 222, 224, 234, 246, 262, 266, 274,
276, 280, 283, 289, 295, 298, 307, 308, 316, 317, 318, 322, 327, 328, 345, 349, 361, 363, 375,
ablution - 139,
ably - 169,
```

Кінець:

```
younger - 17, 22, 42, 49, 67, 68, 85, 96, 102, 121, 123, 136, 161, 162, 163, 167, 179, 180, 194, 217, 235, 269, 302, 345,
youngest - 5, 25, 26, 29, 37, 42, 162, 209, 348,
younge - 198, 316,
yourself - 8, 20, 22, 34, 45, 58, 90, 94, 95, 97, 105, 109, 113, 118, 130, 132, 133, 138, 141, 157, 171, 188, 210, 227, 269, 275, 277, 286, 290, 291, 293, 294, 296,
297, 307, 319, 353, 354, 358, 359, 362, 366, 371, 372, 378,
yours - 16, 27, 28, 39, 43, 44, 47, 55, 84, 91, 105, 146, 161, 192, 297, 307, 321, 380, 381,
yourselves - 194,
youth - 77, 163, 192, 227, 231, 236, 275,
youths - 226,
```

5. Лістинг коду

1) task1.lang_with_go_to

```
using System;
```

```
namespace MultiParadigmLab1
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            string[] stopWords = { "for", "the", "in", "is", "a", "from" };
```

```
            int stopWordsCount = 6;
```

```
            int numberOfStopWordsFound = 0;
```

```

int numberOfWordsToEstimate = 18; // !!! must be less than or equal to the number of words in a
text
// string inputText = "africa White 1, tigers, live for live the, mostly in India, - white, Wild! - lions
live mostly in africa, Africa!";
string filepath = "lab1part1.txt";
string inputText = System.IO.File.ReadAllText(filepath);

string[] words = new string[numberOfWordsToEstimate];
int numOfWordsCounter = 0;

// variables for tracking words in input
int wordBeginsIndex = 0;
int wordEndsIndex = 0;

bool foundABeginningOfAWord = false;
bool isAStopWord = false;

int numberOfFoundWords = 0;
string wordBuilder = "";

// THE ALGORITHM ITSELF
int i = 0;
wordsHunterLoopBody:
    inputText += " ";

// trying to find a beginning of a new word
if (!foundABeginningOfAWord && wordEndsIndex != 0)
{
    int j = wordEndsIndex;
    searchingLoopBody:

        if ((inputText[j] > 64 && inputText[j] < 123) &&
            ((inputText[j - 1] > 0 && inputText[j - 1] < 65) ||
             (inputText[j - 1] > 90 && inputText[j - 1] < 97) || (inputText[j - 1] == '"') || inputText[j - 1]
             == '_'))
        {
            wordBeginsIndex = j;
            foundABeginningOfAWord = true;
            goto wordsHunterLoopBody;
        }

        j++;
        goto searchingLoopBody;
    }

// found word case
if ((inputText[i] > 64 && inputText[i] < 123) &&
    ((inputText[i + 1] > 0 && inputText[i + 1] < 65) || (inputText[i + 1] > 90 && inputText[i + 1] <
    97)
    || (inputText[i + 1] == '"') || inputText[i + 1] == '_'))
{
    wordEndsIndex = i;
    isAStopWord = false;

    // extracting the word out of array
    int extractCounter = wordBeginsIndex;
    extractLoopBody:

```

```

// making it lowercased
int unicodeValue = inputText[extractCounter];
if ((unicodeValue >= 65) && (unicodeValue <= 90))
{
    unicodeValue += 32;
}

wordBuilder += (char)unicodeValue;

if (extractCounter > wordEndsIndex)
{
    throw new ArgumentException("incorrect");
}

if (extractCounter != wordEndsIndex)
{
    extractCounter++;
    goto extractLoopBody;
}
else
{
    // if we got to the end of the word
    // checking if it is not a stop-word
    int j = 0;
checkingLoopBody:

    if (wordBuilder == stopWords[j])
    {
        isAStopWord = true;
        numberOfWordsToEstimate -= 1;
        numberOfStopWordsFound += 1;
    }

    j++;
    if (j < stopWordsCount) goto checkingLoopBody;

    if (!isAStopWord)
    {
        words[numberOfFoundWords] = wordBuilder;
        numOfWorksCounter++;
        numberOfFoundWords++;
    }

    wordBuilder = "";
    foundABeginningOfAWord = false;
}
}
i++;
if (numberOfFoundWords != numberOfWordsToEstimate)
    goto wordsHunterLoopBody;

// analyzing our words array
string[] uniqueWords = new string[numberOfWordsToEstimate];
int[] wordsFrequency = new int[numberOfWordsToEstimate];

int numberOfUniqueWords = 0;
bool isUnique = true;

```

```

    int analyzeLoopCounter = 0;
analyzeLoopLabel:
    // if we find the word not for the first time
    int internalAnalyzeLoopCounter = 0;
internalAnalyzeLoop:
    if (words[analyzeLoopCounter] == uniqueWords[internalAnalyzeLoopCounter])
    {
        wordsFrequency[internalAnalyzeLoopCounter] += 1;
        isUnique = false;
    }
    internalAnalyzeLoopCounter++;
    if (internalAnalyzeLoopCounter < numberOfUniqueWords) { goto internalAnalyzeLoop; }

    // if we find the word for the first time
    if (isUnique && words[analyzeLoopCounter] != null)
    {
        uniqueWords[numberOfUniqueWords] = words[analyzeLoopCounter];
        wordsFrequency[numberOfUniqueWords] += 1;
        numberOfUniqueWords++;
    }

    isUnique = true;
    analyzeLoopCounter++;
    if (analyzeLoopCounter < numOfWordsCounter) { goto analyzeLoopLabel; }

    // initializing our array with indices
    int[] sortedIndices = new int[numberOfUniqueWords];
    int initArrayCounter = 0;
initLoopLabel:
    sortedIndices[initArrayCounter] = initArrayCounter;
    initArrayCounter++;
    if (initArrayCounter < numberOfUniqueWords) { goto initLoopLabel; }

    // sorting indices with bubble sort
    int temp = 0;
    int temp2 = 0;
    int outerSortCounter = 1;
    int internalSortCounter = 0;
outerSortLabel:
internalSortLabel:
    if (wordsFrequency[internalSortCounter] > wordsFrequency[internalSortCounter + 1])
    {
        temp = wordsFrequency[internalSortCounter + 1];
        wordsFrequency[internalSortCounter + 1] = wordsFrequency[internalSortCounter];
        wordsFrequency[internalSortCounter] = temp;

        temp2 = sortedIndices[internalSortCounter + 1];
        sortedIndices[internalSortCounter + 1] = sortedIndices[internalSortCounter];
        sortedIndices[internalSortCounter] = temp2;
    }
    internalSortCounter++;
    if (internalSortCounter < numberOfUniqueWords - outerSortCounter) { goto internalSortLabel; }
}
    internalSortCounter = 0;
    outerSortCounter++;
    if (outerSortCounter < numberOfUniqueWords - 1) { goto outerSortLabel; }

```



```

        // outputting the result
        int outputCounter = 1;
        outputLoopLabel:
        Console.WriteLine($"{uniqueWords[sortedIndices[numberOfUniqueWords - outputCounter]]}" +
            $" - {wordsFrequency[numberOfUniqueWords - outputCounter]}");
        outputCounter++;
        if (outputCounter < numberOfUniqueWords + 1) { goto outputLoopLabel; }
    }
}
}

```

2) task2.lang_with_go_to

```

using System;

namespace MultiParadigmLab1._2
{
    class Program
    {
        static void Main(string[] args)
        {
            string bookText = System.IO.File.ReadAllText($"prideAndPrejudice.txt");
            int characterCount = 706573; //num of symbols in a given text

            string[] stopWords = { "for", "that", "the", "into", "of", "with", "an", "at",
                "in", "is", "so", "if", "as", "on", "a", "from", "and", "to", "by" };
            int stopWordsCount = 19;
            int numberOfStopWordsFound = 0;

            string[] words = new string[95000];

            // for tracking words
            int wordBeginsIndex = 0;
            int wordEndsIndex = 0;
            string wordBuilder = "";
            int numberOfFoundWords = 0;

            bool foundABeginningOfAWord = false;
            bool isAStopWord = false;

            int i = 0;
        wordsHunterLoopBody:
            // trying to find a beginning of a new word
            if (!foundABeginningOfAWord && wordEndsIndex != 0)
            {
                int j = wordEndsIndex;
            searchingLoopBody:

                if ((bookText[j] > 64 && bookText[j] < 123) &&
                    ((bookText[j - 1] > 0 && bookText[j - 1] < 65) ||
                    (bookText[j - 1] > 90 && bookText[j - 1] < 97) || (bookText[j - 1] == '"') || bookText[j - 1] ==
                    '—'))
                {
                    wordBeginsIndex = j;
                    foundABeginningOfAWord = true;

                    goto wordsHunterLoopBody;
                }
            }
        }
    }
}

```

```

        j++;
        goto searchingLoopBody;
    }

    // found word case
    if ((bookText[i] > 64 && bookText[i] < 123) &&
        ((bookText[i + 1] > 0 && bookText[i + 1] < 65) || (bookText[i + 1] > 90 && bookText[i + 1] <
97)
        || (bookText[i + 1] == '"') || bookText[i + 1] == '—'))
    {
        wordEndsIndex = i;
        isAStopWord = false;

        // extracting the word out of array
        int extractCounter = wordBeginsIndex;
extractLoopBody:
        if (bookText[extractCounter] == ' '
            || bookText[extractCounter] == '\n')
        {
            extractCounter++;
            wordBuilder = "";
        }
        // making it lowercased
        int asciiValue = bookText[extractCounter];
        if ((asciiValue > 64) && (asciiValue < 91))
        {
            asciiValue += 32;
        }

        wordBuilder += (char)asciiValue;

        if (extractCounter != wordEndsIndex)
        {
            extractCounter++;
            goto extractLoopBody;
        }
        else
        {
            // if we got to the end of the word
            // checking if it is not a stop-word
            int j = 0;
checkingLoopBody:

            if (wordBuilder == stopWords[j])
            {
                isAStopWord = true;
                numberOfStopWordsFound += 1;
            }
            j++;
            if (j < stopWordsCount) goto checkingLoopBody;

            if (!isAStopWord)
            {
                words[numberOfFoundWords] = wordBuilder;
                numberOfFoundWords++;
            }
        }
    }

```

```

        wordBuilder = "";
        foundABeginningOfAWord = false;
    }
}
i++;
if (i < characterCount)
    goto wordsHunterLoopBody;

// analyzing our words array
int possibleNumberOfUniqueWords = 6500;
string[] uniqueWords = new string[possibleNumberOfUniqueWords];
int numberOfUniqueWords = 0;
bool isUnique = true;

int currentPage = 1;
int wordsPerPage = 245;

// allocating memory for subarrays to track pages
// index in this arr corresponds to index in the uniqueWords arr
int[][] pagesForWords = new int[possibleNumberOfUniqueWords][];
int allocationCounter = 0;
allocationLoopLabel:
    pagesForWords[allocationCounter] = new int[500];
    allocationCounter++;
    if (allocationCounter < possibleNumberOfUniqueWords) { goto allocationLoopLabel; }

// tracking pages (1 page == 245 words)
int analyzeLoopCounter = 0;
analyzeLoopLabel:
    int internalAnalyzeLoopCounter = 0;
internalAnalyzeLoop:
    // if we find the word not for the first time
    if (words[analyzeLoopCounter] == uniqueWords[internalAnalyzeLoopCounter])
    {
        // here goes the code if we have already found a word
        currentPage = analyzeLoopCounter / wordsPerPage + 1;

        // looking for an empty space in an array
        int emptySpaceCounter = 0;
emptySpaceSearchLoop:
        if (pagesForWords[internalAnalyzeLoopCounter][emptySpaceCounter] == 0)
        {
            pagesForWords[internalAnalyzeLoopCounter][emptySpaceCounter] = currentPage;
            goto endArray;
        }
        emptySpaceCounter++;
        if (emptySpaceCounter < 500) { goto emptySpaceSearchLoop; }

        endArray:
        isUnique = false;
    }
    internalAnalyzeLoopCounter++;
    if (internalAnalyzeLoopCounter < numberOfUniqueWords) { goto internalAnalyzeLoop; }

// if we find the word for the first time
if (isUnique && words[analyzeLoopCounter] != null)
{
    // checking word's correctness

```

```

int correctnessCount = 0;
correctnessCheckLabel:

    if (words[analyzeLoopCounter][correctnessCount] == ""
        || words[analyzeLoopCounter][correctnessCount] == '-'
        || words[analyzeLoopCounter][correctnessCount] == "'")
    {
        goto notCorrectWordCase;
    }

correctnessCount++;
if (correctnessCount < words[analyzeLoopCounter].Length) { goto correctnessCheckLabel; }
// here goes the code if we've not met the word before and the word is correct
uniqueWords[numberOfUniqueWords] = words[analyzeLoopCounter];

currentPage = analyzeLoopCounter / wordsPerPage + 1;

// looking for an empty space in an array
int searchEmptyCounter = 0;
emptySearchLabel:
    if (pagesForWords[numberOfUniqueWords][searchEmptyCounter] == 0)
    {
        pagesForWords[numberOfUniqueWords][searchEmptyCounter] = currentPage;
        goto endArray2;
    }
    searchEmptyCounter++;
    if (searchEmptyCounter < 500) { goto emptySearchLabel; }
endArray2:
    numberOfUniqueWords++;
}
notCorrectWordCase:
    isUnique = true;
    analyzeLoopCounter++;
    if (analyzeLoopCounter < numberOfFoundWords) { goto analyzeLoopLabel; }

// sorting it alphabetically
string tempString;
int[] tempIntArray;
int sortLoopLimit = 1;
bool isCurrentBigger = false;

int outerSortLoopCounter = 0;
outerSortLoopLabel:
    int internalSortLoopCounter = 0;
internalSortLoopLabel:
    int compCount = 0;
compCountLabel:
    sortLoopLimit = uniqueWords[internalSortLoopCounter].Length >
uniqueWords[internalSortLoopCounter + 1].Length ?
    uniqueWords[internalSortLoopCounter + 1].Length :
uniqueWords[internalSortLoopCounter].Length;

    if (uniqueWords[internalSortLoopCounter][compCount] >
uniqueWords[internalSortLoopCounter + 1][compCount])
    {
        isCurrentBigger = true;
        goto swapLabel;
    }

```

```

        else if (uniqueWords[internalSortLoopCounter][compCount] <
uniqueWords[internalSortLoopCounter + 1][compCount]) { goto swapLabel; }
        compCount++;
        if (compCount < sortLoopLimit) { goto compCountLabel; }

swapLabel:
        if (isCurrentBigger)
        {
            tempString = uniqueWords[internalSortLoopCounter + 1];
            uniqueWords[internalSortLoopCounter + 1] = uniqueWords[internalSortLoopCounter];
            uniqueWords[internalSortLoopCounter] = tempString;

            tempIntArray = pagesForWords[internalSortLoopCounter + 1];
            pagesForWords[internalSortLoopCounter + 1] =
pagesForWords[internalSortLoopCounter];
            pagesForWords[internalSortLoopCounter] = tempIntArray;

            isCurrentBigger = false;
        }
        internalSortLoopCounter++;
        if (internalSortLoopCounter < numberOfUniqueWords - 1) { goto internalSortLoopLabel; }
        outerSortLoopCounter++;
        if (outerSortLoopCounter < numberOfUniqueWords - 1) { goto outerSortLoopLabel; }

// outputting the result
string outputPagesBuilder = "";
int outputCounter = 0;
outputLoopLabel:

    int outputPagesCounter = 0;
    outputPagesLoop:
        // ignoring repeating page numbers (if word occurs >2 times on the same page)
        if ( pagesForWords[outputCounter][outputPagesCounter]
            != pagesForWords[outputCounter][outputPagesCounter + 1] )
        {
            outputPagesBuilder += $"{pagesForWords[outputCounter][outputPagesCounter]}, ";
        }

        outputPagesCounter++;
        if ((pagesForWords[outputCounter][outputPagesCounter + 1] != 0) && (outputPagesCounter <
498)) { goto outputPagesLoop; }

        // ignoring words that occurs over 100 times
        if (outputPagesCounter < 100)
        {
            Console.WriteLine($"{uniqueWords[outputCounter]} - {outputPagesBuilder}");
        }

        outputCounter++;
        outputPagesBuilder = "";
        if (outputCounter < numberOfUniqueWords) { goto outputLoopLabel; }
    }
}
}

```

