**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**
**«Київський політехнічний інститут імені Ігоря Сікорського»**
**Факультет інформатики та обчислювальної техніки**
**Кафедра інформатики та програмної інженерії**

*Звіт до лабораторної роботи №1 з дисципліни*

*“Мультипарадигменне програмування”*

Прийняв:
Викладач кафедри ІПІ:
Баришич Л. М.
15 лютого 2022 року

Виконав
Студент групи ІТ-01:
Бардін В. Д.

**Київ – 2022**

TASK 1

```
namespace Lab_1;

public class Task1
{
    private const int TopN = 25;
    private const int StopWordsCount = 2;
    private const string FilePath = @"C:\Users\vbardin\Desktop\Task 1.txt";

    private static readonly string[] StopWords = {"the", "a"};

    public static unsafe void ExecuteTask1()
    {
        var textToProcess = File.ReadAllText(FilePath);
        var textToProcessLength = 0;
        var whitespaces = 0;

        countTextLength:
        {
            try
            {
                if (textToProcess[textToProcessLength] != '\0')
                {
                    textToProcessLength++;
                    goto countTextLength;
                }
            }
            catch
            {
                textToProcessLength--;
            }
        }

        // Convert text to lower case
        if (textToProcessLength == 0)
        {
            return;
        }

        fixed (char* pSource = textToProcess)
        {
            var charIdx = 0;
            // Convert letter to lower case and count whitespaces
            toLowerCycleStart:
            {
                uint letter = *(pSource + charIdx);

                if (letter - 'A' <= 'Z' - 'A')
                {
                    letter += 32;
                    *(pSource + charIdx) = (char) letter;
                }

                if (letter is ' ' or ',' or '!' or '?' or '.')
                {
                    whitespaces++;
```

```
            }
        }

        charIdx++;

        if (charIdx < textToProcessLength)
        {
            goto toLowerCycleStart;
        }
    }

    // Split words
    var indexer = 0;
    var words = new string[whitespaces + 1];

    var savedAmount = 0;

    var firstWhiteSpaceIdx = 0;
    var wordStartIndex = indexer;
    var wordEndIndex = indexer;

    findWhiteSpace:
    if (textToProcess[indexer] == ' ')
    {
        firstWhiteSpaceIdx = indexer;
        // to exit from cycle
        indexer = textToProcessLength;
    }

    indexer++;
    if (indexer < textToProcessLength)
    {
        goto findWhiteSpace;
    }

    wordEndIndex = firstWhiteSpaceIdx == wordStartIndex
        ? textToProcessLength
        : firstWhiteSpaceIdx;

    words[savedAmount] = textToProcess[wordStartIndex..wordEndIndex];
    savedAmount++;

    // cause the word starts after the whitespace character
    var currPos = firstWhiteSpaceIdx + 1;
    wordStartIndex = currPos;
    wordEndIndex = currPos;

    findWordEnd:
    if (textToProcess[currPos] == ' ' ||
        textToProcess[currPos] == ',' ||
        textToProcess[currPos] == '!' ||
        textToProcess[currPos] == '?' ||
        textToProcess[currPos] == '.')
    {
        wordEndIndex = currPos;
        words[savedAmount] = textToProcess[wordStartIndex..wordEndIndex];
        savedAmount++;
```

```
            wordStartIndex = wordEndIndex += 1;
        }

        currPos++;
        if (currPos < textToProcessLength)
        {
            goto findWordEnd;
        }

        wordEndIndex = wordStartIndex == wordEndIndex
            ? textToProcessLength
            : wordEndIndex;

        words[savedAmount] = textToProcess[wordStartIndex..wordEndIndex];

        // Remove stop words
        var stopWords = 0;

        var stwIdx = 0;
        removeStopWords:
        {
            var currStopWord = StopWords[stwIdx];

            var currTtpWordIdx = 0;
            internalCycle:
            {
                if (words[currTtpWordIdx] == currStopWord)
                {
                    words[currTtpWordIdx] = "";
                    stopWords++;
                }

                currTtpWordIdx++;
            }
            if (currTtpWordIdx < whitespaces + 1)
            {
                goto internalCycle;
            }

            currTtpWordIdx = 0;
            stwIdx++;
        }

        if (stwIdx < StopWordsCount)
        {
            goto removeStopWords;
        }

        var tfDescriptors = new TermFrequencyDescriptor[whitespaces + 1 -
stopWords];
        var descriptorsExists = 0;

        var currWordIdx = 0;
        createTermFrequencyDescriptors:
        {
            var descriptorFound = false;
```

```
            if (words[currWordIdx] != "")
            {
                var tfdIdx = 0;
                findTfxDescriptor:
                {
                    if (tfdIdx < descriptorsExists &&
                        tfDescriptors[tfdIdx].Term == words[currWordIdx])
                    {
                        tfDescriptors[tfdIdx].Frequency++;
                        descriptorFound = true;
                    }

                    tfdIdx++;
                }
                if (tfdIdx < descriptorsExists)
                {
                    goto findTfxDescriptor;
                }

                if (!descriptorFound)
                {
                    var tfd = new TermFrequencyDescriptor(words[currWordIdx],
1);
                    tfDescriptors[descriptorsExists] = tfd;
                    descriptorsExists++;
                }
            }

            currWordIdx++;
        }
        if (currWordIdx < whitespaces + 1)
        {
            goto createTermFrequencyDescriptors;
        }

        // Create terms descriptors and order them
        var currTfdIdx = 0;
        order:
        {
            var idx = 0;
            innerLoop:
            {
                if (tfDescriptors[idx].Frequency < tfDescriptors[idx +
1].Frequency)
                {
                    (tfDescriptors[idx], tfDescriptors[idx + 1]) =
                        (tfDescriptors[idx + 1], tfDescriptors[idx]);
                }

                idx++;
            }
            if (idx < whitespaces - stopWords - currTfdIdx)
            {
                goto innerLoop;
            }
```

```
            currTfdIdx++;
        }
        if (currTfdIdx < whitespaces - stopWords)
        {
            goto order;
        }

        // take top N

        var mostlyUsed = new TermFrequencyDescriptor[TopN];
        var descriptorsSelected = 0;
        currTfdIdx = 0;
        takeTopN:
        {
            if (tfDescriptors[currTfdIdx].Term != "")
            {
                mostlyUsed[descriptorsSelected] = tfDescriptors[currTfdIdx];
                descriptorsSelected++;
            }
            else
            {
                goto printRes;
            }

            currTfdIdx++;
        }
        if (currTfdIdx < descriptorsExists &&
            currTfdIdx < TopN)
        {
            goto takeTopN;
        }

        printRes:
        {
            var pIdx = 0;
            printCycle:
            {
                Console.WriteLine(mostlyUsed[pIdx]);
                pIdx++;
            }
            if (pIdx < descriptorsSelected)
            {
                goto printCycle;
            }
        }
    }
}

struct TermFrequencyDescriptor
{
    public string Term { get; set; }
    public int Frequency { get; set; }

    public TermFrequencyDescriptor(string term, int frequency)
    {
        Term = term;
        Frequency = frequency;
```

```
    }

    public override string ToString()
    {
        return $"{Term} - {Frequency}";
    }
}
```

TASK 2

```
namespace Lab_1;

public class Task2
{
    private const string FilePath = @"C:\Users\vbardin\Desktop\Task 1.txt";
    public const int MaxWordsPerLine = 7;

    public static void ExecuteTask()
    {
        var fileContent = File.ReadAllLines(FilePath);

        var addedLines = 0;
        var normalizedLines = new string[fileContent.Length *
MaxWordsPerLine][];

        for (var lIndex = 0; lIndex < fileContent.Length; lIndex++)
        {
            var line = ToLower(fileContent[lIndex]);
            var lWords = line.Split(new[] {' ', ',', '.'},
StringSplitOptions.RemoveEmptyEntries);

            var chunkedLines = new string[(int) Math.Ceiling(lWords.Length /
(double) MaxWordsPerLine)][];
            if (lWords.Length >= MaxWordsPerLine)
            {
                chunkedLines = lWords.Chunk(MaxWordsPerLine).ToArray();
            }

            for (var i = 0; i < chunkedLines.Length; i++)
            {
                // Extend normalized lines arr
                var reqSize = addedLines + chunkedLines.Length;
                if (reqSize >= normalizedLines.Length)
                {
                    var timesToExtend = (int) Math.Ceiling(reqSize / (double)
normalizedLines.Length);
                    var newSize = normalizedLines.Length * timesToExtend;

                    var temp = new string[newSize][];
                    for (var j = 0; j < normalizedLines.Length; j++)
                    {
                        temp[j] = normalizedLines[j];
                    }

                    normalizedLines = temp;
                }
```

```
                normalizedLines[addedLines] = chunkedLines[i];
                addedLines++;
            }
        }

        var shrunkLines = new string[addedLines][];
        for (var i = 0; i < shrunkLines.Length; i++)
        {
            shrunkLines[i] = normalizedLines[i];
        }

        const int linesPerPage = 45;
        var pagesAmount = (int) Math.Ceiling(shrunkLines.Length / (double)
linesPerPage);

        var pages = new string[pagesAmount][][];
        var linesProcessed = 0;

        for (var i = 0; i < pagesAmount; i++)
        {
            pages[i] = new string[linesPerPage][];
            for (var lIndex = 0; lIndex < linesPerPage && lIndex +
linesProcessed < shrunkLines.Length; lIndex++)
            {
                pages[i][lIndex] = shrunkLines[lIndex + linesProcessed];
            }

            linesProcessed += linesPerPage;
        }

        // There is no sense to shrunk all pages cause only the last one can
be not full
        var realLastPageSize = shrunkLines.Length - linesPerPage *
            (int) Math.Floor(shrunkLines.Length / (double) linesPerPage);

        var shrunkLastPage = new string[realLastPageSize][];

        for (var lIndex = 0; lIndex < realLastPageSize; lIndex++)
        {
            shrunkLastPage[lIndex] = pages[pagesAmount - 1][lIndex];
        }

        pages[pagesAmount - 1] = shrunkLastPage;

        var termsAdded = 0;
        var termDescriptors = new TermDescriptor[0];

        var pIndex = 0;
        pp_1:
        {
            var page = pages[pIndex];
            var descriptorsToSave = ProcessPage(pIndex, page,
termDescriptors);

            for (var i = 0; i < descriptorsToSave.Length; i++)
            {
```

```
                var reqLenght = termsAdded + descriptorsToSave.Length;
                // Extend term descriptors array
                if (reqLenght >= termDescriptors.Length)
                {
                    var newSize = reqLenght;
                    if (termDescriptors.Length != 0)
                    {
                        var timesToExtend = (int) Math.Ceiling(reqLenght /
(double) termDescriptors.Length);
                        newSize = termDescriptors.Length * timesToExtend;
                    }

                    var temp = new TermDescriptor[newSize];
                    for (var j = 0; j < termDescriptors.Length; j++)
                    {
                        temp[j] = termDescriptors[j];
                    }

                    termDescriptors = temp;
                }

                termDescriptors[termsAdded] = descriptorsToSave[i];
                termsAdded++;
            }

            pIndex++;
        }
        if (pIndex < pages.Length)
        {
            goto pp_1;
        }

        var descriptors = new TermDescriptor[termsAdded];
        for (var i = 0; i < descriptors.Length; i++)
        {
            descriptors[i] = termDescriptors[i];
        }

        var et1 = 0;
        et_1:
        {
            var termDescriptor = descriptors[et1];
            Console.WriteLine($"{termDescriptor.Term} - {string.Join(',',
termDescriptor.Pages.Where(x => x != 0))}");

            et1++;
        }
        if (et1 < descriptors.Length)
        {
            goto et_1;
        }
    }

    private static TermDescriptor[] ProcessPage(int pIndex, string[][] lines,
TermDescriptor[] existingDescriptors)
    {
        const int avgWordsPerPage = 250;
```

```
        var descriptorsAdded = 0;
        var descriptors = new TermDescriptor[avgWordsPerPage];

        var lIndex = 0;
        processPage_Line:
        {
            var line = lines[lIndex];

            var wIndex = 0;
            processPage_Word:
            {
                var word = line[wIndex];
                var isDescriptorExists = false;
                TermDescriptor termDescriptor = null!;

                var descriptorsAmount = existingDescriptors.Length == 0
                    ? descriptorsAdded
                    : existingDescriptors.Length;

                var dIndex = 0;
                processPage_DescriptorExists:
                {
                    termDescriptor = existingDescriptors.Length != 0
                        ? existingDescriptors[dIndex]
                        : descriptors[dIndex];

                    if (termDescriptor is null)
                    {
                        goto processPage_DescriptorExists_Found;
                    }

                    if (termDescriptor.Term != word)
                    {
                        goto processPage_DescriptorExists_SectionEnd;
                    }

                    isDescriptorExists = true;
                    goto processPage_DescriptorExists_Found;

                    processPage_DescriptorExists_SectionEnd:
                    dIndex++;
                }
                if (dIndex < descriptorsAmount)
                {
                    goto processPage_DescriptorExists;
                }

                processPage_DescriptorExists_Found:
                if (termDescriptor != null && isDescriptorExists)
                {
                    var pAddedIndex = 0;
                    processPage_DescriptorExists_Found_IsPageAdded:
                    {
                        var addedPage = termDescriptor.Pages[pAddedIndex];
                        if (addedPage == pIndex + 1)
                        {
                            goto processPage_Word_SectionEnd;
```

```
                        }

                        pAddedIndex++;
                    }
                    if (pAddedIndex < termDescriptor.Pages.Length)
                    {
                        goto processPage_DescriptorExists_Found_IsPageAdded;
                    }

                    // Descriptor's pages array is full we need to extend it
                    if (termDescriptor.Pages.Length ==
termDescriptor.PagesSaved)
                    {
                        var temp = new int[termDescriptor.Pages.Length * 2];

                        var pArrayExtendIndex = 0;
                        processPage_DescriptorExists_Found_ExtendPage:
                        {
                            temp[pArrayExtendIndex] =
termDescriptor.Pages[pArrayExtendIndex];
                            pArrayExtendIndex++;
                        }
                        if (pArrayExtendIndex < termDescriptor.Pages.Length)
                        {
                            goto
processPage_DescriptorExists_Found_ExtendPage;
                        }

                        termDescriptor.Pages = temp;
                    }

                    termDescriptor.Pages[termDescriptor.PagesSaved] = pIndex
+ 1;
                    termDescriptor.PagesSaved++;
                }
                else
                {
                    var newDescriptor = new TermDescriptor
                    {
                        Term = word,
                        Pages =
                        {
                            [0] = pIndex + 1
                        },
                        PagesSaved = 1
                    };

                    var reqLength = descriptorsAdded + 1;
                    if (reqLength >= descriptors.Length)
                    {
                        var timesToExtend = (int) Math.Ceiling(reqLength /
(double) descriptors.Length);
                        var temp = new TermDescriptor[descriptors.Length *
timesToExtend];

                        var dExtendIndex = 0;
                        processPage_DescriptorExists_Extend:
```

```
                        {
                            temp[dExtendIndex] = descriptors[dExtendIndex];
                            dExtendIndex++;
                        }
                        if (dExtendIndex < descriptors.Length)
                        {
                            goto processPage_DescriptorExists_Extend;
                        }

                        descriptors = temp;
                    }

                    descriptors[descriptorsAdded] = newDescriptor;
                    descriptorsAdded++;
                }

processPage_Word_SectionEnd:
                wIndex++;
            }
            if (wIndex < line.Length)
            {
                goto processPage_Word;
            }

            lIndex++;
        }
        if (lIndex < lines.Length)
        {
            goto processPage_Line;
        }

        var shrunkDescriptors = new TermDescriptor[descriptorsAdded];

        for (var i = 0; i < shrunkDescriptors.Length; i++)
        {
            shrunkDescriptors[i] = descriptors[i];
        }

        return shrunkDescriptors;
    }

    private static string ToLower(string str)
    {
        unsafe
        {
            if (str.Length == 0)
            {
                return "";
            }

            fixed (char* pSource = str)
            {
                var charIdx = 0;
                // Convert letter to lower case and count whitespaces
                toLowerCycleStart:
                {
                    uint letter = *(pSource + charIdx);
```

```
                    if (letter - 'A' <= 'Z' - 'A')
                    {
                        letter += 32;
                        *(pSource + charIdx) = (char) letter;
                    }
                }

                charIdx++;

                if (charIdx < str.Length)
                {
                    goto toLowerCycleStart;
                }
            }
        }

        return str;
    }
}

internal class TermDescriptor
{
    private const int DefaultPagesAmount = 4;

    public string Term { get; set; }
    public int PagesSaved { get; set; }
    public int[] Pages { get; set; }

    public TermDescriptor()
    {
        Pages = new int[DefaultPagesAmount];
    }
}
```