Touchsides Technical Assignment

QUESTION 1

Please develop a console / GUI application that is used to determine the frequency of each word that appears in the book War and Peace by Leo Tolstoy. Your application can be written in C# or Java.

Ideally your application will have a user interface that allows the user to select/specify the input file and displays the answers to the below questions as static output.

Output:
Most frequent word: {word} occurred {x} times
Most frequent 7-character word: {word} occurred {x} times
Highest scoring word(s) (according to Scrabble): {word} with a score of {x}

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text.RegularExpressions;
using System.Threading;
using System.Windows;

namespace TechTest
{
  /// <summary>
  /// Interaction logic for MainWindow.xaml
  /// </summary>
  public partial class MainWindow : Window
  {

    private Dictionary<char, int> ScrabbleScores = new Dictionary<char, int>()
    {
      { 'a', 1 }, { 'b', 3 }, { 'c', 3 }, { 'd', 2 },
      { 'e', 1 }, { 'f', 4 }, { 'g', 2 }, { 'h', 4 },
      { 'i', 1 }, { 'j', 8 }, { 'k', 5 }, { 'l', 1 },
      { 'm', 3 }, { 'n', 1 }, { 'o', 1 }, { 'p', 3 },
      { 'q', 10 }, { 'r', 1 }, { 's', 1 }, { 't', 1 },
      { 'u', 1 }, { 'v', 4 }, { 'w', 4 }, { 'x', 8 },
      { 'y', 4 }, { 'z', 10 }, { '-', 0 }
    };
    public MainWindow()
    {
      InitializeComponent();
    }

    private void GetFile_Click(object sender, RoutedEventArgs e)
    {
      Result.Text = "Find a txt file";
      Microsoft.Win32.OpenFileDialog dlg = new Microsoft.Win32.OpenFileDialog()
      {
        // Set filter for file extension and default file extension
        DefaultExt = ".txt",
        Filter = "Text Files (*.txt)|*.txt"
      };

      // Display OpenFileDialog by calling ShowDialog method
      Nullable<bool> result = dlg.ShowDialog();

      // Get the selected file
      if (result == true)
      {
```

```
//address a different thread to update UI
ThreadPool.QueueUserWorkItem((o) =>
{
  //user feedback
  Dispatcher.Invoke((Action)(() => Result.Text = "Processing... (Please wait)"));
  Dispatcher.Invoke((Action)(() => FindButton.IsEnabled = false));

  // Open document
  string filename = dlg.FileName;
  StreamReader reader = new StreamReader(dlg.FileName);

  //setup variables
  string currentLine;
  Dictionary<string, int> words = new Dictionary<string, int>();
  int highScrabbleScore = 0;
  string highScrabbleScoreWord = string.Empty;
  DateTime wordsToHashTimeTaken = DateTime.Now;

  //loop the lines of the document
  while ((currentLine = reader.ReadLine()) != null)
  {
    var splitString = Regex.Replace(currentLine, "[^A-Za-z -]", "").ToLower().Split(' ');
    foreach (string word in splitString)
    {
      if (word == string.Empty)
      {
        continue;
      }
      if (words.ContainsKey(word))
      {
        // The word already exists
        words[word] = words[word] + 1;
      }
      else
      {
        // First occurance of the word
        words.Add(word, 1);

        //get the scrabble score
        int score = 0;
        foreach (char letter in word)
        {
          try
          {
            score += ScrabbleScores[letter];
          }
          catch { }
        }
        if (score > highScrabbleScore)
        {
          highScrabbleScore = score;
          highScrabbleScoreWord = word;
        }
        else if (score == highScrabbleScore)
        {
          highScrabbleScoreWord += ", " + word;
        }
      }
    }
  }
  TimeSpan createHashTimeSpan = DateTime.Now - wordsToHashTimeTaken;

  //use linq to get output
  DateTime mostCommonWordTimeTaken = DateTime.Now;
  KeyValuePair<string, int> mostCommonWord = words.FirstOrDefault(x => x.Value == words.Values.Max());
  TimeSpan mostCommonWordTimeSpan = DateTime.Now - mostCommonWordTimeTaken;

  DateTime mostCommonWord7TimeTaken = DateTime.Now;
  Dictionary<string, int> words7 = words.Where(c => c.Key.Length == 7).ToDictionary(p => p.Key, p => p.Value);
  KeyValuePair<string, int> mostCommonWord7 = words7.FirstOrDefault(x => x.Value == words7.Values.Max());
  TimeSpan mostCommonWord7TimeSpan = DateTime.Now - mostCommonWord7TimeTaken;
```

```
        string resultText = "Most frequent word: \"" + mostCommonWord.Key + "\" occurred " +
mostCommonWord.Value.ToString() + " times";
        resultText += "\n\rMost frequent 7-character word: \"" + mostCommonWord7.Key + "\" occurred " +
mostCommonWord7.Value.ToString() + " times";
        resultText += "\n\rHighest scoring word(s) (according to Scrabble): \"" + highScrabbleScoreWord + "\" with a score of " +
highScrabbleScore.ToString();

        string processingTime = "Create hash map and find high scrabble score: " +
createHashTimeSpan.TotalMilliseconds.ToString("0.000", System.Globalization.CultureInfo.InvariantCulture) + "ms";
        processingTime += "\n\rFind most frequent word: " + mostCommonWordTimeSpan.TotalMilliseconds.ToString("0.000",
System.Globalization.CultureInfo.InvariantCulture) + "ms";
        processingTime += "\n\rFind most frequent 7 letter word: " +
mostCommonWord7TimeSpan.TotalMilliseconds.ToString("0.000", System.Globalization.CultureInfo.InvariantCulture) + "ms";

        //output to user
        Dispatcher.Invoke((Action)(() => Result.Text = resultText));
        Dispatcher.Invoke((Action)(() => FindButton.IsEnabled = true));


        Dispatcher.Invoke((Action)(() => Computations.Text = processingTime));

      });

    }
  }

 }
}
```

QUESTION 2

Please assess the performance of your application using the usual measures. Please write a paragraph or two describing the performance achieved and possible ways you could improve it.

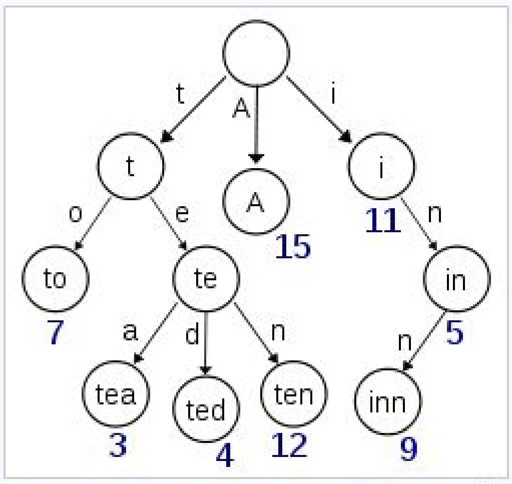
The performance of the application runs with an average time of 600ms on a Intel Core i7 2.20GHz Processor with a file that has 66 056 lines and a length of 576 831 words. That equates to just under 1 million words a second. Considering that a novel is categorised as anything over 40 000 words for the Nebula awards this program should manage almost any book in under a second.

The txt file is processed line by line with a stream reader. A regex is then used on the line to remove any odd characters and then converted to lowercase to make it easier to work with. The line is then split by a space character and this array is looped through. If the word exists in the dictionary, where the word is the key and the value is the word count, then the count is incremented otherwise the word is added to the dictionary and count is set to one. If it is the first time the word has occurred then the scrabble score is also calculated.

This line by line and word processing has an **O(n)** complexity while being added to the dictionary. The complex part comes when the scrabble score is calculated. Assuming worst case scenario each word is different which means the complexity becomes **O(n)⨉O(p)** where **O(n)** is the number of words and **O(p)** is the average word length. In this case 576 831 words and 5.82 letters per word.

All the linq statements seem to use at worst an **O(n log n)** complexity whereas something like a “where” linq statement uses **O(n)** as it iterates through the collection. Extension methods that use indexed access, such as ElementAt, Skip, Last or LastOrDefault, will check to see whether or not the underlying type implements IList<T>, so that you get **O(1)** access instead of **O(N)**.

A way of improving this solution would be to find a better way to calculate the highest scrabble score. This can be achieved by seeing if the cumulative count would not get high enough and using a break to get out of the loop. A small optimization is already used due to the fact only new words are used to calculate high scores.
Another interesting solution to the problem would be to use a trie solution which can be seen below.



A trie for keys "A","to", "tea", "ted", "ten", "i", "in", and "inn".

In computer science, a trie, also called digital tree and sometimes radix tree or prefix tree (as they can be searched by prefixes), is a kind of search tree—an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are not necessarily associated with every node. Rather, values tend only to be associated with leaves, and with some inner nodes that correspond to keys of interest.