Calculation of Levenshtein Distance

Learning Objectives

- File I/O
- Using your IDE (e.g. Visual Studio)
- Implementation of a (not too simple) algorithm in C++
- Testing (Test-First, i.e. Think, Red-Bar, Green-Bar, Refactor
- Using C++-classes vector, string

Exercise at a glance

Implement a class `Levenshtein` to calculate the Levenshtein distance between two-word sequences.

Detailed Exercise Descriptions

The Levenshtein distance is a string metric for measuring the difference between two different word sequences or between two different words. The Levenshtein distance between two words is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other. We will use it here for word sequences.

Let's assume we have the two different word sequences "swiss one nine two mike" and "swiss one two nine mike".

The Levenshtein distance between them is two:

- Word sequence 1: `swiss one  nine two mike`
- Word sequence 2: `swiss one two nine  mike`

The word `two` in the beginning is inserted in sequence 2 and the word `two` at the end is deleted in sequence 2. Therefore, we say that we have a Levenshtein distance of two.

The Levenshtein distance between "sky travel five seven juliett dobry den praha radar radar contact descend one hundred" and "ryan air juliett dobry den praha radar radar contact descend one hundred" in 4:

```
-   sky travel five seven juliett dobry den praha radar radar contact descend one hundred
-   ryan air             juliett dobry den praha radar radar contact descend one hundred
```

We have two substitutions and two deletions. This example also shows that the number of operations to transform one word sequence into the other is not unique, but the distance (metric value) itself is. First two deletions and then two substitutions would also transform the first into the second:

```
-   sky travel five seven juliett dobry den praha radar radar contact descend one hundred
-             ryan air   juliett dobry den praha radar radar contact descend one hundred
```

Or first a deletion and then two substitutions and again a deletion would also solve the transformation task.

```
-   sky travel five seven juliett dobry den praha radar radar contact descend one hundred
-       ryan   air        juliett dobry den praha radar radar contact descend one hundred
```

Google the algorithm and implement it. You find it e.g. in Wikipedia at de.wikipedia.org/wiki/Levenshtein-Distanz and a C++-implementation you find even at en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Levenshtein_distance

Task 1.1

Implement now a class `Levenshtein` with the following interface:[1]

```
class Levenshtein
{
public:
   // astr1
   Levenshtein(
      const std::vector<std::string>& astr1,  // first string compared to astr2
      const std::vector<std::string>& astr2)  // i.e. LD between
                                              // astr1 and astr2 is calculated

   //! returning the Levenshtein distance between astr1 and astr2
   int CalcLevenshteinDistance();
   std::string backtrace() const;
   std::string GetPrettyPrint(int ai_ld, std::string astr_goldText="",
      std::string astr_recognText="") const;
   /* maybe other methods */

private:
      // first index runs over size of mstr1
   int Get(int st1Ind, int st2Ind) const {
      return (mpi_mat[st1Ind * mi_spCnt_n2 + st2Ind]);
   }
   std::vector<std::string> mstr1;
   std::vector<std::string> mstr2;
   int mi_zCnt_m1;   // size of mstr1
   int mi_spCnt_n2;  // size of mstr2
   // contains the matrix, used for LD calculation as a vector
   int mpi_mat[1000]; // later we create a dynamic matrix, currently sizes
                      // of astr1 and astr2 are limited
   /* maybe other members */
};
```[2]

a)

You need to implement the method `CalcLevenshteinDistance();` You will find implementations at the internet.

---

[1] Of course, other implementation exist, but use this interface. If every group has the same interface, we can also exchange code and you can integrate better code from me. The third reason is that you should not just copy from the internet, but should adapt at least at little bit.

[2] It might be easier to implement it as a vector<vector<int>>, but for learning progress and future exercise, we just use this approach.

Do not forget to implement tests for this functionality.

Hier you find an example:

```
   // We calculate the Levenshtein distance of Tier and Tor which should be 2
bool LevenshteinDistTierTor()
   {
      vector<string> s1{ "T", "i", "e", "r" };
      vector<string> s2{ "T", "o", "r" };
      Levenshtein dist(s1, s2);
      return 2 == dist.CalcLevenshteinDistance();
   }
```

Implement other tests, which also show that you can handle cases, when the array `mpi_mat` is not big enough or when one or both strings are empty or when the Levenshtein distance is zero. Just test so that you are sure, that I not able to crash your implementation or get wrong results.

b)

Implement the method backtrace, which shows the steps to transform mstr1 into mstr2. We explain best by an example, i.e. a test:

```
   bool LevenshteinDistRenTierTiere()
   {
      vector<string> s1{ "R", "e", "n", "T", "i", "e", "r" };
      vector<string> s2{ "T", "i", "e", "r", "e" };
      Levenshtein dist(s1, s2);
      cout << "Needed steps: " << dist.backtrace());
      return (4 == dist.CalcLevenshteinDistance());
   }
```

The expected output of the cout command to the screen should be:

```
   Needed steps: Del Del Del Equ Equ Equ Equ Ins
```

We first have three deletions to transform “Rentier” to “Tiere”. The we have four equal strings and then we have a insertion.

You might get problems to output a type vector<string>, i.e. a compiler error.

Therefore, please add the following lines of code before all your test code (once is enough).[3]

```
   inline std::ostream& operator<<(
          std::ostream& str, const std::vector<std::string>& v)
   {
      std::string blank = "";
      for (auto iter : v)
      {
         str << blank << iter;
         blank = " ";
      }
      return str;
   }
```

Calling with “Tiere” and “Rentier” as shown in the following test in:

```
bool LevenshteinDistTiereRenTier()
```

[3] Details of understanding you will get later.

```
{
   vector<string> s1{ "T", "i", "e", "r", "e" };
   vector<string> s2{ "R", "e", "n", "T", "i", "e", "r" };
   Levenshtein dist(s1, s2);
   cout << "Needed steps: " << dist.backtrace());
   return (4 == dist.CalcLevenshteinDistance());
}
```

should result in

```
Needed steps: Ins Ins Ins Equ Equ Equ Equ Del
```

It should also be possible for complete words (are normal use case):

```
bool LevenshteinDistWords()
{

   vector<string> s1{ "Tango", "ind", "echo", "romeo" };
   vector<string> s2{ "Ind", "echo", "romeo", "abba"};
   Levenshtein dist(s1, s2);
   cout << "Needed steps: " << dist.backtrace());
   return (3 == dist.CalcLevenshteinDistance());
}
```

The expected output is:

```
Needed steps: Subs Del Equ Equ Ins
```

And again, we emphasize that the needed steps are not unique. Th following steps are also possible with the same Levenshtein distance:

```
Needed steps: Del Subs Equ Equ Ins
```

Do not forget enough tests, i.e. you need to test the expected output of backtrace against the output you get.

c)

Implement now together with using the (tested) method backtrace the method GetPrettyPrint, which creates a better output, easy to recognize differences between to strings.

The output to cout of

```
bool LevenshteinDistWords()
{
   vector<string> s1{ "Tango", "ind", "echo", "romeo" };
   vector<string> s2{ "Ind", "echo", "romeo", "abba"};
   Levenshtein dist(s1, s2);
   auto ld = dist.CalcLevenshteinDistance();
   cout << dist.GetPrettyPrint(ld, "s1", "and s2") << "\n";
   return 3 == ld;
}
```

should be:

```
s1      : Tango ind echo romeo

and s2 : Ind       echo romeo abba  // LD: 3, gold words: 4
```

If you do not want to implement by yourself, use the following code:

```
    string Levenshtein::GetPrettyPrint
    (
       int ai_ld,
       string astr_goldText,
       string astr_recognText
    ) const
    {
       std::string prettyGold;
       std::string prettyRecogn;
       vector<string> opsVect = C_String(backtrace()).split_V(" ", true);
       int lenDiff = static_cast<int>(
             astr_recognText.length() - astr_goldText.length());
       int blankGold   = lenDiff < 0 ? 3 :  lenDiff + 3;
       int blankRecogn = lenDiff > 0 ? 3 : -lenDiff + 3;

       bool prettyPrintCheck = GoldAndRecogAsPrettyString(
          mstr1, mstr2, opsVect, prettyGold, prettyRecogn);
       ostringstream ostr;
       ostr << astr_goldText << setw(blankGold) << " : ";
       // checking if prettyPrinting has worked else we proceed with normal printing
       if (prettyPrintCheck)
       {
          ostr << prettyGold << "\n";
       }
       else
       {
          ostr << mstr1 << "\n";
       }
       ostr << astr_recognText << setw(blankRecogn) << " : ";
       // checking if prettyPrinting has worked else we proceed with normal printing
       if (prettyPrintCheck)
       {
          ostr << prettyRecogn;
       }
       else
       {
          ostr << mstr2;
       }
       ostr <<  " // LD: " << ai_ld
          << " gold words: " << mstr1.size() << "\n";
       return ostr.str();
    }
```

```
    bool GoldAndRecogAsPrettyString
    (
       const vector<std::string>& goldVec,
       const vector<std::string>& recognVec,
       const vector<std::string>& opsVec,
       std::string& prettyGold,
       std::string& prettyRecogn
```

```
)
{
   int goldCounter = 0;
   int recognCounter = 0;
   int goldSize = static_cast<int>(goldVec.size());
   int recognSize = static_cast<int>(recognVec.size());
   for (auto iter : opsVec)
   {
      if (iter == "Subs")
      {
         if (goldCounter < goldSize && recognCounter < recognSize)
         {
            std::string gold = goldVec[goldCounter];
            std::string recogn = recognVec[recognCounter];

            int goldLen = (int)gold.length();
            int recognLen = (int)recogn.length();
            if (goldLen < recognLen)
            {
               prettyRecogn = prettyRecogn + recogn + " ";
               prettyGold = prettyGold + gold +
                   std::string(recognLen - goldLen + 1, ' ');
            }
            else
            {
               prettyGold = prettyGold + gold + " ";
               prettyRecogn = prettyRecogn + recogn +
                   std::string(goldLen - recognLen + 1, ' ');
            }
            ++goldCounter;
            ++recognCounter;
         } // if (goldCounter < goldSize && recognCounter < recognSize)

         else
            return false;

      } // if (iter == "Subs")

      else if (iter == "Del")
      {
         if (goldCounter < goldSize)
         {
            std::string gold = goldVec[goldCounter];
            int goldLen = (int)gold.length();
            prettyGold = prettyGold + gold + " ";
            prettyRecogn = prettyRecogn + std::string(goldLen + 1, ' ');
            ++goldCounter;
         }

         else
            return false;

      } // else if (iter == "Del")

      else if (iter == "Ins")
      {
         if (recognCounter < recognSize)
         {
            std::string recogn = recognVec[recognCounter];
            int recognLen = (int)recogn.length();
            prettyRecogn = prettyRecogn + recogn + " ";
            prettyGold = prettyGold + std::string(recognLen + 1, ' ');
            ++recognCounter;
         }
```

```
            else
               return false;

         } // else if (iter == "Ins")

         else
         {
            if (goldCounter < goldSize && recognCounter < recognSize)
            {
               std::string gold = goldVec[goldCounter];
               std::string recogn = recognVec[recognCounter];
               prettyGold = prettyGold + gold + " ";
               prettyRecogn = prettyRecogn + recogn + " ";
               ++goldCounter;
               ++recognCounter;
            }

            else
               return false;

         } // else
      }

      return true;
   }
```