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` and `string`

Exercise at a glance

Implement a class `Levenshtein` to calculate the Levenshtein distance between two, i.e. sequence of characters.

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 one. The Levenshtein distance between two word-sequences (a sentence) is the minimum number of single-word edits (insertions, deletions or substitutions) required to change one word-sequence into the other one.

We will use it here for words.

Let's assume we have the two different words "highlight" and "heightlite".

The Levenshtein distance between them is:

- Word sequence 1: `h i g h t l i g h t`
- Word sequence 2: `h e i g h t l i t e`

The character `e` in the beginning and at end is inserted in sequence 2 and the characters `g` and `h` at the end are deleted in sequence 2. Therefore, we say that we have a Levenshtein distance of 4.

We also see, that here the needed operations to transform one word into the other are not unique

- Word sequence 1: `h i g h t l i g h t`
- Word sequence 2: `h e i g h t l i t e`

With insertion of 'e', substitution of "g" by "t" and "h" by "e" and deletion of "e" we also get four operations, i.e. a Levenshtein distance of 4.

The next example shows the Levenshtein distance between two word sequences. 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" is 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, ryan/sky and air/travel plus the two deletions of the words five and seven. This example again 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` or https://en.wikipedia.org/wiki/Levenshtein_distance.[1]

Task 1.1

Implement now a class `Levenshtein` with the following interface:[2]

```
class Levenshtein
{
public:
   // astr1
   Levenshtein(
      const std::vector<char>& aw1,  // first word aw1 compared to aw2
      const std::vector<char>& aw2)  // i.e. LD between aw1 and aw2 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<char> mw1;
   std::vector<char> mw2;
   int mi_zCnt_m1;   // size of mstr1
   int mi_spCnt_n2;  // size of mstr2
   // contains the matrix, used for LD calculation as a vector
```

[1] At least these links were correct in 2025-09-05.

[2] Of course, other implementations 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. And the forth reason, it eases my task, which is the main reason.

```
   int mpi_mat[1000]; // later we create a dynamic matrix, currently sizes
                      // of mw1 and mw2 are limited to 30 charcters
   /* maybe other members are needed */
};
```
[3]

a)

You need to implement the method `CalcLevenshteinDistance();` You will find implementations at the internet.
Do not forget to implement tests for this functionality[4].

Here you find an example of a test:

```
  // We calculate the Levenshtein distance of Tier and Tor which should be 2
   TEST(LevenshteinTest, LevenshteinDistTierTor)
   {
      LOG_METHOD(g_DEBUGns, g_DEBUGcl, "LevenshteinDistTierTor", "");

      vector<char> s1{ 'T', 'i', 'e', 'r' };
      vector<char> s2{ 'T', 'o', 'r' };
      Levenshtein dist(s1, s2);
      EXPECT_EQ(2, dist.CalcLevenshteinDistance());
   }
```

Implement other tests, which also show that you can handle cases, when the array `mpi_mat` is too big enough or when one or both strings are empty or when the Levenshtein distance is zero or …. Or …. Just test so that you are sure, that I am 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:

```
   TEST(LevenshteinTest, LevenshteinDistTierTiere)
   {
      LOG_METHOD(g_DEBUGns, g_DEBUGcl, "LevenshteinDistTierTiere", "");

      vector<char> s1{ 'T', 'i', 'e', 'r' };
      vector<char> s2{ 'T', 'i', 'e', 'r', 'e' };
      Levenshtein dist2(s2, s1);
      EXPECT_EQ(1, dist2.CalcLevenshteinDistance());
      TRACE("Levenshtein Matrix of Tiere/Tier\n" << dist2.GetMatrixAsString());
      TRACE("Needed steps: " << dist2.backtrace());
      EXPECT_EQ(0, dist2.GetSubstCnt());
      EXPECT_EQ(1, dist2.GetDelCnt());
      EXPECT_EQ(0, dist2.GetInsCnt());

      Levenshtein dist(s1, s2);
      EXPECT_EQ(1, dist.CalcLevenshteinDistance());
      TRACE("Levenshtein Matrix of Tier/Tiere\n" << dist.GetMatrixAsString());
      TRACE("Needed steps: " << dist.backtrace());
```

[3] It might be easier to implement it as a vector<vector<int>>, but for learning progress and future exercises, we just use this approach.
[4] Functionality without (**automatic**) tests does not exist. Consequently code without tests is not considered for evaluation by me.

```
    EXPECT_EQ(0, dist.GetSubstCnt());
    EXPECT_EQ(0, dist.GetDelCnt());
    // one Insertion into Tier results in Tiere
    EXPECT_EQ(1, dist.GetInsCnt());
}
```

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

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

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

Therefore, please add the following lines of code before all your test code (once is enough).[5]

```
inline std::ostream& operator<<(
       std::ostream& str, const std::vector<std::char& 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()
{
    vector<char> s1{ "T", "i", "e", "r", "e" };
    vector<char> 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
```

Do not forget to implement different tests, which automatically check whether your implementation of backtrace might be correct.

The above code is not enough. It does not automatically test the 8 steps.

[5] Details to understand this code, you will get later.

c)

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

The output to `cout` of

```
TEST(LevenshteinTest, TestPrPrint)
{
   vector<char> s1{ 'Z', 'o', 'o', 't', 'i', 'e', 'r', 'e' };
   vector<char> s2{ 'Z', 'i', 'e', 'g', 'e', 't', 'i', 'e', 'r' };

   Levenshtein dist(s1, s2);
   auto ld = dist.CalcLevenshteinDistance();
   cout << dist.GetPrettyPrint(ld, "s1", "and s2");
}
```

should be:

```
s1     : Z o o     t i e r e

and s2 : Z i e g e t i e r    // LD: 5, subs: 2 / ins : 2 / del : 1 gold words: 8
```

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(
      mw1, mw2, 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 << mw1 << "\n";
   }
   ostr << astr_recognText << setw(blankRecogn) << " : ";
   // checking if prettyPrinting has worked else
   // we proceed with normal printing
```

```
        if (prettyPrintCheck)
        {
            ostr << prettyRecogn;
        }
        else
        {
            ostr << mw2;
        }
        ostr <<  " // LD: " << ai_ld
            << ", subs: " << GetSubstCnt()
            << " / ins : " << GetInsCnt() << " / del : " << GetDelCnt()
            << " gold words: " << mw1.size() << "\n";
        return ostr.str();
    }
```

The called method `GoldAndRecogAsPrettyString` could have the following implementation.

```
    bool GoldAndRecogAsPrettyString
    (
        const vector<T_BaseType>& goldVec,
        const vector<T_BaseType>& 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)
                {
                    T_BaseType gold = goldVec[goldCounter];
                    T_BaseType recogn = recognVec[recognCounter];

                    int goldLen = 1;
                    int recognLen = 1;
                    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)
           {
              T_BaseType gold = goldVec[goldCounter];
              int goldLen = 1;
              prettyGold = prettyGold + gold + " ";
              prettyRecogn = prettyRecogn + std::string(goldLen + 1, ' ');
              ++goldCounter;
           }

           else
              return false;

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

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

           else
              return false;

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

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

           else
              return false;

        } // else
     }

     return true;
  }
```

Do not forget to implement different tests which automatically check whether your implementation of `GetPrettyPrint` might be correct.