1 Comparison

1.1 Hamming distance

The **Hamming distance** between **two strings of equal length** is the **number of positions at which the corresponding symbols are different**.

- "ka*rol*in" and "ka*thr*in" → 3
- 10*1*1*1*01 and 10*0*1*0*01 → 2

With Hamming distance we can formalize *substitutions* in biological sequences - or simply sequencing errors in which the wrong base pair is identified.

1.2 Edit distance

The **edit distance** is a way of **quantifying how dissimilar** two strings (e.g., words) are to one another by counting the **minimum number of operations required to transform on string into the other**.

In the *Levenshtein distance* (the most common), edit operations are: **removal**, **insertion**, and **substitution**.

The edit distance between "kitten" and "sitting" is 3. A minimal edit script that transforms the former into the latter is:

- **k**itten → **s**itten (substitute s for k)
- sitt**e**n → sitt**i**n (substitute i for e)
- sittin → sittin**g** (insert g at the end)

The number of solutions (sequences of operations) is infinite.

1.2.1 Computing the edit distance

We denote as $E(S, T)$ the edit distance between two strings S and T.

For every string S, $E(S, \varepsilon) = |S|$, that is:

- the edit distance between a string S and the empty string ε equals the length of S.
- also, $E(\varepsilon, \varepsilon) = 0$: the edit distance between two empty strings is zero.

Let $S = s_1 s_2 ... s_n$ and $T = t_1 t_2 ... t_m$ be two strings:

- we denote as $S(i)$ the prefix of length i of $S \rightarrow S(i) = s_1 s_2 ... s_i$
- likewise, as $T(j)$ the prefix of length j of $T \rightarrow T(j) = t_1 t_2 ... t_j$

Now, to compute the edit distance $E(S,T)$ between two strings S and T, with $|S| = n$ characters, and $|T| = m$ characters, a technique of dynamic programming is used.

We can easily calculate the edit distance of two substrings of S and T, with length i and j, if we have the edit distance of the two "previous" substrings, that is, the substrings of length $i-1$ and $j-1$.

To do so we prepare a matrix with $|S|+1$ rows and $|T|+1$ columns, numbered starting from 0.

The cell(i, j) will contain the value of the edit distance between prefixes S(i) and T(j).

We can fill the first row and first column right away since we know that $\forall i, E(S(i), \varepsilon) = i$ and that $\forall j, E(T(j), \varepsilon) = j$.

	-	**H**	**O**	**M**	**E**
-	0	1	2	3	4
H	1				
O	2				
U	3				
S	4				
E	5				

Now, the value of each cell can be calculated with this formula.

$$E(S(i),T(j)) = \min \begin{cases} E(S(i),T(j-1))+1 \\ E(S(i-1),T(j))+1 \\ E(S(i-1),T(j-1)) \quad \text{if} \quad s_i = t_j \\ E(S(i-1),T(j-1))+1 \quad \text{if} \quad s_i \neq t_j \end{cases}$$

That is, **the edit distance between S(i) and T(j) depends on the edit distances computed for their prefixes**.

	-	**H**	**O**	**M**	**E**
-	0	1	2	3	4
H	1	0	1	2	3
O	2	1	0	1	2
U	3	2	1	1	2
S	4	3	2	2	2
E	5	4	3	3	2

Every cell M(i, j) contains the edit distance between prefixes S(i) and T(j). **Cell M(n, m) contains the edit distance between S(n) and T(m), that is E(S, T)**.

1.2.2 Computational complexity of edit distance

The table has $(n+1)x(m+1)$ cells to be filled, hence

$$\textbf{space complexity} = \boldsymbol{\Theta}(\mathbf{nm})$$

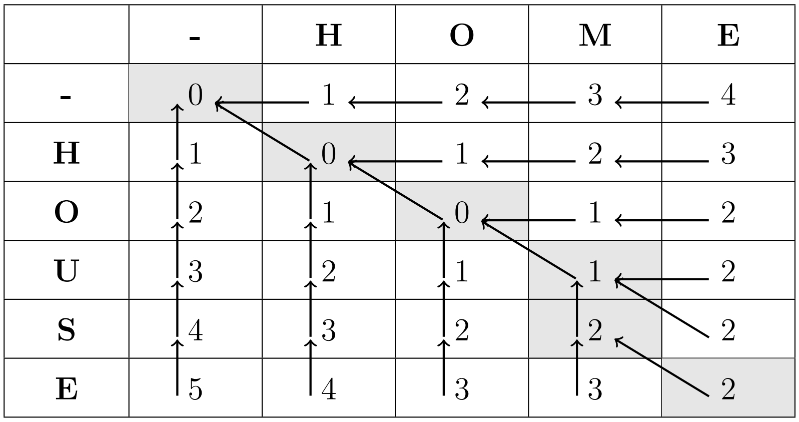
For every cell, the time required for the computation is constant (i.e. does not depend on the size of input strings), hence, the algorithm performs $(n+1) \times (m+1)$ steps of constant time. Thus

$$\textbf{time complexity} = \boldsymbol{\Theta}(\mathbf{nm})$$

1.2.3 Traceback

Once the table has been built, and the edit distance between the two strings has been computed, one may want to know the exact sequence of operations (insertions, deletions and substitutions) that transform the first string into the second.

To allow this, during the creation of the matrix, every time each cell is filled we **not only save the score value, but also which of the 3 previous cells had the minimum value**.



	-	**H**	**O**	**M**	**E**
-	0	1	2	3	4
H	1	0	1	2	3
O	2	1	0	1	2
U	3	2	1	1	2
S	4	3	2	2	2
E	5	4	3	3	2

Now, if we **start from the final cell**, we can follow the pointers until we reach the top-left corner and deduce the sequence of edit operations from the type of moves that form the path.

- **Diagonal move**: when the path makes a diagonal move there are two possibilities, depending on whether the characters that identify the start cell match or not:
  - if they are **the same**, **no operation** is applied.
  - if they **don't match**, a **substitution** takes place.
- **Vertical move**: during a vertical move an **insertion** takes place: the current character of the second string is inserted into the first string.
- **Horizontal move**: when a horizontal move occurs a character is **deleted** from the first string.

In the case of the example above the sequence of moves and the corresponding operations are:

1. **E, E → M, S:** diagonal move, E is equal to E so **no operation**.

2. **M, S → M, U:** vertical move, M and S don't match so **S is inserted in the first string**.
3. **M, U → O, O:** diagonal move, M and U don't match so **M is substituted with U**.
4. **O, O → H, H:** diagonal move, O is equal to O, so **no operation**.
5. **H, H → -, -:** diagonal move, H is equal to H, so **no operation**.

We can shorten this representation like so:

```
S - - U N D A Y
S A T U R D A Y
```

1.2.4 Computational complexity of backtracking

The number of "backtracking steps" will equal the number of columns of the final alignment. **Each step takes constant time** (just follow a pointer and read the letters needed), so the worst-case number of steps corresponds to the maximum number of columns that one alignment of strings of length $|S| = n$ and $|T| = m$ can have. This means:

$$\textbf{space complexity} = \boldsymbol{\Theta}(\mathbf{n}+\mathbf{m})$$

$$\textbf{time complexity} = \boldsymbol{\Theta}(\mathbf{n}+\mathbf{m})$$

1.2.5 Weighted edit distance

The edit distance algorithm can be generalized by giving different "weights" to the various operations. By doing so a more general formula is obtained:

$$E(S(i),T(j)) = \min \left\{ \begin{array}{rl} E(S(i),T(j-1)) + \mathbf{w_{id}} & \\ E(S(i-1),T(j)) + \mathbf{w_{id}} & \\ E(S(i-1),T(j-1)) & \text{if} \quad s_i = t_j \\ E(S(i-1),T(j-1)) + \mathbf{w}(\mathbf{s_i},\mathbf{t_j}) & \text{if} \quad s_i \neq t_j \end{array} \right.$$

- $\mathbf{w_{id}}$ represents the weight of an insertion or deletion of a character.
- $\mathbf{w}(\mathbf{s_i},\mathbf{t_j})$ represents the weight of substitution of the character s_i with t_j. This means that we can prepare a "weight matrix" with the weight for every pair of characters of the alphabet Σ we use.

1.3 Longest Common Subsequence

We have seen the edit distance as a way to measure the dissimilarity of two strings S and T, now we want to evaluate their similarity. One solution is to find the **longest common subsequence (LCS)** between S and T. The longest this common subsequence is, the more similar the two strings are.

The algorithm to solve this problem, similarly to the edit distance, makes use of dynamic programming principles.

1.3.1 Computing the LCS

We denote *LCS(S, T)* as the length of the longest common subsequence between two strings S and T, having $|S| = n$ and $|T| = m$, and *S(i)* the prefix of length i of S and *T(j)* the prefix of length j of T.

Also, given the empty string ε, for every string S, we have $\mathbf{LCS(\boldsymbol{\varepsilon}, S) = 0}$.

We can now build a similar table to that in the edit distance algorithm, but changing the formula to calculate the values in each cell.

$$E(S(i), T(j)) = \mathbf{max} \begin{cases} E(S(i), T(j-1)) + \mathbf{0} \\ E(S(i-1), T(j)) + \mathbf{0} \\ E(S(i-1), T(j-1)) + \mathbf{1} & \text{if} \quad s_i = t_j \\ E(S(i-1), T(j-1)) + \mathbf{0} & \text{if} \quad s_i \neq t_j \end{cases}$$

Note that "min" has become "max": we are looking for the LCS of **maximum** length, instead of the **minimum** set of edit operations.

As before, we also keep track of the path as the table is built.

	-	**H**	**O**	**M**	**E**
-	0	0	0	0	0
H	0	1	1	1	1
O	0	1	2	2	2
U	0	1	2	2	2
S	0	1	2	2	2
E	0	1	2	2	3

In order to reconstruct the actual LCS we again start from the bottom right and move back following the pointers.

We add a letter to the LCS only when a **diagonal move** is made **and the letters match**. In the above example:

- **E, E → M, S:** diagonal move, letters match so LCS is now "E".
- **M, S → M, U:** vertical move, do nothing.
- **M, U → O, O:** diagonal move, letters don't match, do nothing.
- **O, O → H, H:** diagonal move, letters match, LCS becomes "OE".
- **H, H → -, -:** diagonal move, letters match, LCS is now "HOE".