Assignment 1

07.11.23

Maria Bruno (3636989), Francesca Carlon (3664579), Linnet Moxon (3387157)

Task 1

a)

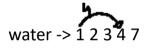
water -> 1 2 3 4 7

nice -> 1 5 6

sun -> 5 6 7 8 9

is -> 6

b)



nice -> 156



is -> 6

<u>intersection of water and nice without skip pointer:</u>

start with *nice*, because the postings-list is shorter. Compare 1 from *nice* and 1 from *water*. There is a match. Increase the number in both lists. Compare 5 from *nice* and 2 from *water*. No match. Compare 5 from *nice* and 3 from *water*. No match. Compare 5 from *nice* and 4 from *water*. No match. 4 in *water* is the lower number, increase it. Compare 7 from *water* with 5 from *nice*. No match. Increase *nice*. Compare 6 from *nice* to 7 from *water*. No match.

Total: 1 Match

Intersection of water and nice with skip pointer:

If we go from 1 to 4 in *water*, we skip the comparison of 5 from *nice* to 2 and 3 in *water* and therefore save some steps.

Skip pointer in sun:

In theory you could add a skip pointer here, although it's not necessary, because except from *sun* no postings-list goes beyond 7 and therefore the intersection ends after 7 anyways.

Task 2

The function takes "O'Really isn't in S." as input string. For every character in the string, that is not a letter or a number (here: apostrophe and dot) it adds a white space around that character. All the characters – punctuation padded with white space – are then added to a new string. This new string is split into a list of tokens at white space and saved as list_of_tokens. This list is returned in the end.

If you give the function the sentence "O'Really isn't in S." as input string, it will return

```
["O", "'", "Really", "isn", "'", "t", "in", "S", "."]
```

The algorithm has a linear runtime in the number of input characters n(O(n)) because, in the for-loop, it goes through each element of the string just once.

Task 3

			v1 (j)	v2	v3	v4	v5
			M	0	U	S	E
		0	1	2	3	4	5
u1 (i)	М	1	<mark>0</mark>	<mark>1</mark>	<mark>2</mark>	3	4
u2	U	2	<mark>1</mark>	1	1	2	3
u3	0	3	<mark>2</mark>	1	<mark>1*</mark>	2	3
u4	S	4	3	2	2	1	3
u5	E	5	4	3	3	3	1

^{*}because we look above two levels -> u1v1 = 0 => u3v3 = 0+1 = 1

When filling the matrix according to the Damerau-Levenshtein algorithm rules the cell u3v3 fits the condition $u_i = v_{j-1}$ and $u_{i-1} = v_j$. Therefore to fill it, we look at all cells two levels above, two levels to the left, and two levels diagonally. This means, for u3v3 we consider u1v1, u1v2, u1v3, u2v1, u3v1, according to D_{i-2} , j-2. We add +1 for the operation "transpose" to each one of the cells. The minimum value is 1.

All the other cells are filled according to the other rules of the Damerau-Levenshtein algorithm (insert, delete, replace).

Programming Task 1

```
mation
                    tokenizes and normalizes the text of the
def prepare file(filename):
    tweets = []
    with open(filename, encoding="utf8") as f:
        for lines in f:
            line = lines.split('\t')
            line = [line[1], line[4]]
            tweets.append(line)
    for 1st in tweets:
        new content.append(lst[0])
        new content.append(tokenize and normalize(lst[1]))
    return new content
def tokenize and normalize(text):
   normalized token = []
```

```
tokens = text.split()
    for word in tokens:
        normalized token.append(word.lower())
    return normalized token
appears in
    :return dictionary of the form {'term': number of docu-
def create dict(filename):
    content = prepare file(filename)
    term dict = {}
    for item in content:
        for word in item:
            if word not in term dict.keys():
                term dict.setdefault(word)
    return term dict
malized tokens], ...]
```