—Twist—

TEAM-ID: 01020
GRUPPE: PLEPPOS
RAYKO EICHHORST, THEO JACOBI UND KIRU SPREU
23.11.2018

Anfänglich ist es essentiell eine Agenda zu haben, nach der man strukturiert agieren kann. Prinzipiell ist das System stets gleich.
Die ersten paar Schritte sind gewisser Maßen in den theoretischen Teil einzugliedern, worauf danach die praktische Implementierung und Exekution folgt.

1. Identifizieren des Problems
2. Formulierung des Problems
3. Entwurf eines Algorithmus
   — Falls nötig in Pseudocode
4. Implementierung des Algorithmus und
5. Anwendung des Algorithmus — der Prozess

Lösungsansatz:

FORMULIERUNG:
Diese Aufgabe besteht aus zwei Teilen. Als erstes gilt es einen Algorithmus zu schreiben, welcher einen Text “twistet”. Der zweite Teil besteht aus der Wiederrückgängigmachung eines getwisteten Textes.

ENTWURF:
Der Input wird uns anhand von Textdokumenten zur Verfügung gestellt. Dementsprechend muss man diese noch angemessen in das Programm integrieren.
Der Algorithmus des ersten Teiles sollte nur einmal durch den Input-Text iterieren und für jedes Wort (Charakter-Strings aus den deutschen Buchstaben) eine *twistify*-Methode anwenden. Diese Methode vertausche die Charakter intern eines Wortes (alle außer das Erste und Letzte). Das Resultat werde dementsprechend natürlich in dem Ausgangstext ersetzt.
Der zweite Teil sollte nach dem gleichen Prinzip fungieren, jedoch umgekehrt. Hierbei könne man nur Wörter rückgängig machen, die in der zur Verfügung gestellten Wörterliste parat sind.
Da wir Zuversicht in dieses System haben, empfinden wir den Gebrauch von Pseudocode obsolet.

KOMPLEXITÄT VON DEN ALGORITHMEN:
Die Laufzeit des gesamten Programmes sollte dabei linear sein, da jeweils nur inplace-Veränderungen vorgenommen werden. Dies bedeutet, dass die Laufzeit des Programmes eins-zu-eins mit der Inputgröße ansteigt.

Umsetzung:

Wir entschieden uns Python als Programmiersprache zu benutzen, da es eine allumfassende Sprache und für ein so primitives Programm mehr als ausreichend ist.
Zusätzlich importierten wir die Regular-Expression-, Zeit- und Random-Bibliothek.

Einspeisung von Daten

```
176  # ------------------------------------------------- #
177  timeResult=""
178  for num in range(0, 5):
179      start=time.time()
180      with open("toUntwist/untwist"+str(num+1)+".txt", "w") as f:
181          f.write(twistifySentence('toTwist/twist'+str(num+1)+'.txt'))
182      with open("twistedUntwisted/twist"+str(num+1)+"_output.txt", "w") as f:
183          f.write(untwistifySentence('toUntwist/untwist'+str(num+1)+'.txt'))
184      end=time.time()
185      timeResult+=("{} took {} amount of seconds\n".format(num+1, end-start))
186  with open("time.txt", "w") as timef:
187      timef.write(timeResult)
188
```

Funktionsweise:

- alle fünf Beispieldateien werden eingelesen und sukzessiv bearbeitet
- zusätzlich wird die benötigte Zeit gestoppt und in einem Textdokument gesichert
  - (Uns ist bewusst, dass die Messmethode relativ zu der Maschine ist, die das Programm ausführt, die Relation wird trotzdem deutlich)

- das Programm funktioniert wie folgt:
  - in einem Verzeichnis “toTwist” existieren die Inputtexte aus der BWINF-Seite
  - diese sind mit dem Wort “twist” sowie einem Index deklariert (und der txt-Extension)
  - das Ergebnis mit dem getwisteten Text wird unter “toUntwist” als “untwist” mit jeweiligen Index gespeichert
  - letztlich wird erneut auf “toUntwist” zugegriffen, die verworrenen Texte wieder normalisiert und unter “twistedUntwisted” und mit vorherigem Dateinamen gesichert
  - zusätzlich wird die evaluierte Zeit in einer “time.txt” Datei abgelegt

Twisten eines Dokumentes

```
59  def twistifySentence(path):
60      result=[]
61      startWord = False
62      startSChar = False
63      with open(path) as f:
64          data = f.read()
65      words = re.findall(r"[a-zA-ZöÖüÜäÄß]+", data)
66      specialChars = re.findall(r"[^a-zA-ZöÖüÜäÄß]+", data)
67      if len(words)>0:
68          startWord = bool(re.match(words[0], data))
69      if len(specialChars)>0:
70          startSChar = bool(re.match(specialChars[0], data))
71      if  startWord:
72          for iter in range(0, len(specialChars)):
73              result.append(twistify(words.pop(0))+specialChars.pop(0))
74      if startSChar:
75          for iter in range(0, len(words)):
76              result.append(specialChars.pop(0)+twistify(words.pop(0)))
77      if len(words)>0:
78          result.append(twistify(words.pop(0)))
79      elif len(specialChars)>0:
80          result.append(specialChars.pop(0))
81      return ''.join(result)
```

Funktionsweise:

- Separieren von Wörtern aus den deutschen Buchstaben… — Zeile 65
  - die Einzigen die twisted werden sollen
- …und restlichen Charakterzusammensetzungen — Zeile 66
  - beispielsweise: Nummern, Satzzeichen etc.
- zunächst wird evaluiert, ob die Datei mit einem Wort oder einem nicht-Wort beginnt — Zeile 67
  - dies ist für die Wiederherstellung der Datei nötig
- insofern der Inputstring mit einem Wort beginnt, heißt das, dass die Länge der nich-Wort-Gruppe im Vergleich zu den Wörtern nur kleiner-gleich sein kann und umgekehrt —> setzte den (potentiell) kleineren Betrag als Grenze — Zeile 71
- wendet die *twistify*-Methode an den Wörtern an (mehr dazu unten)
- bindet das Resultat und die besonderen Zeichen in einem Array
- falls noch Reste in einer der beiden Arrays bestehen werden diese noch angefügt — Zeile 77
- letztlich wird die Liste in einen String konvertiert und ausgegeben

Twisten eines Wortes

In der letzten Methode wurde auf diese Funktion verwiesen, welche für den eigentlichen Vertauschprozess verantwortlich ist.

Funktionsweise:

- in der Reichweite von dem 1. bis zum vorletzten Buchstaben werden diese intern arbiträr mithilfe der *swap*-Methode vertauscht — Zeile 45
  - hierbei ist es möglich, dass die zufälligen Integerwerte die Gleichen sind, was keinerlei Veränderung am Wort zur Folge hätte
- die *swap*-Funktion nimmt ein Wort und zwei Indizes — Zeile 49
- Charakter an den jeweiligen Stellen werden nun getauscht

—> Hiermit ist der erste Teil der Aufgabe absolviert. Nun galt es nur noch diesen Prozess rückgängig zu machen.

```
43
44  def twistify(word):
45      for i in range(1, len(word)-1):
46          word=swap(r.randint(1, len(word)-2), r.randint(1, len(word)-2), word)
47      return word
48
49  def swap(i,j,input):
50      word=list(input)
51      word[i],word[j]=word[j],word[i]
52      return ''.join(word)
53
```

Untwisten eines Dokumentes

Diese Methode handelt nach dem selben Prinzip wie das Twisten eines Dokuments. Der einzige Unterschied ist, dass man nun *untwistify* an Stelle von *twistify* auf alle Wörter anwendet.

```
122 def untwistifySentence(path):
123     result=[]
124     startWord = False
125     startSChar = False
126     with open(path) as f:
127         data = f.read()
128     words = re.findall(r"[a-zA-ZöÖüÜäÄß]+", data)
129     specialChars = re.findall(r"[^a-zA-ZöÖüÜäÄß]+", data)
130     if len(words)>0:
131         startWord = bool(re.match(words[0], data))
132     if len(specialChars)>0:
133         startSChar = bool(re.match(specialChars[0], data))
134     if  startWord:
135         for iter in range(0, len(specialChars)):
136             result.append(str(untwistify(words.pop(0)))+str(specialChars.pop(0)))
137     elif startSChar:
138         for iter in range(0, len(words)):
139             result.append(specialChars.pop(0)+untwistify(words.pop(0)))
140     if len(words)>0:
141         result.append(untwistify(words.pop(0)))
142     if len(specialChars)>0:
143         result.append(specialChars.pop(0))
144     return ''.join(result)
145
```

Untwisten eines Wortes

Die *untwistify*-Funktion determiniert, ob das eingegebene, verworrene Wort in der Wörterliste vorhanden ist. Wenn dies der Fall ist wird das Wort wiederhergestellt.

```
93  def untwistify(twisted):
94      firstUpper = twisted[0].isupper()
95      start, end = getStartEnd(chr(ord(twisted[0].lower())))
96      possibleWords=[]
97      for dictWord in range(start, end):
98          match = re.findall(\
99          r'\b'+twisted[0]+r'[a-zA-ZöÖüÜäÄß]{'+str(len(twisted)-2)+r'}'+twisted[len(twisted)-1]+r'\b'
100         , wordlist[dictWord], re.IGNORECASE)
101         if len(match)>0:
102             possibleWords.append(match[0])
103     if firstUpper:
104         start, end = getStartEnd(twisted[0])
105         for dictWord in range(start, end):
106             match = re.findall(\
107             r'\b'+twisted[0]+r'[a-zA-ZöÖüÜäÄß]{'+str(len(twisted)-2)+r'}'+twisted[len(twisted)-1]+r'\b'
108             , wordlist[dictWord])
109             if len(match)>0:
110                 possibleWords.append(match[0])
111     for word in possibleWords:
112         correctWord=True
113         for i in range(1, len(twisted)-1):
114             if  len(re.findall(twisted[i], twisted[1:len(twisted)-1])) is not len(re.findall(twisted[i], word[1:len(twisted)-1])):
115                 correctWord=False
116                 break
117         if correctWord:
118             if firstUpper:
119                 word = word[0].upper()+word[1:]
120             return word
121     return twisted
```

Funktionsweise:

- anfangs wird festgestellt, ob der erste Buchstabe groß ist — Zeile 94
  —> entscheidend, wenn ein Wort nominalisiert wurde — ist nicht in der Wörterliste vorhanden
- nun werden Start- und Endindizes vom Anfangsbuchstaben in der Wörterliste festgestellt, also die Reichweite indem er sich befindet — Zeile 95
  - hierzu ist es nötig die Liste im voraus zu sortieren — die vorgegebene Liste nämlich ist nur scheinsortiert. Hin und wieder sind manche Wörter nicht an der richtigen Stelle

```
8   wordlist.sort()
```

- der eingebaute Sortieralgorithmus von Python benutzt *Timsort* (https://en.wikipedia.org/wiki/Timsort), welcher im Worst-Case O(n*log(n)) ist (best-case Ω(n))
  - da n aber konstant ist — |*wordlist*| — ändert sich dieser Wert nicht. Es kostet nur eben etwas Zeit

```
getStartEnd(chr(ord(twisted[0].lower())))
```

- die doppelnde Konvertierung von dem Buchstaben ist nötig, da der Buchstabe anscheinend noch kein “reiner” UTF-8 Charakter ist
  —> bei Übergabe als Parameter erkennt der Algorithmus den Buchstaben vorerst nicht. Erst wenn man ihn in einen numerischen Wert und wieder zurück konvertiert, funktioniert alles richtig
- die *getStartEnd*-Methode ist später noch ausführlich erklärt

- zunächst iterieren von start bis end und suchen nach möglichen Wörtern um die Suche einzugrenzen (mithilfe von RegularExpressions)— Zeile 97

```
100    r'\b'+twisted[0]+r'[a-zA-ZöÖüÜäÄß]{'+str(len(twisted)-2)+r'}'+twisted[len(twisted)-1]+r'\b'
```

- darauf folgt das gleiche Prozedere, für den Fall, dass das Wort groß geschrieben ist — Zeile103
- letztlich durchlaufen wir alle gefundenen Möglichkeiten und geben das richtige Wort wieder — Zeile 111
  - falls keins gefunden wurde, geben wir das initiale Wort aus
- wir können Wörter ausschließen, indem wir überprüfen, ob die Anzahl an mittleren Buchstaben in dem möglichen Wort und dem initialen Wort übereinstimmt — Zeile114
  - wenn dies nicht der Fall, gehen wir über zum nächsten Eintrag
- letztlich wird überprüft, ob das Wort anfänglich in Großschrift war (falls es nominalisiert war, o. Ä.) — Zeile 118

Notiz:

Anfangs hatten wir die Eingrenzung der Suche noch nicht eingebaut. Wir durchsuchten jedesmal die gesamte Wörterliste nach möglichen Wörtern.
Die verbesserte Suche nach Wörtern in dem Wörterbuch fungiert mit Suchen nach Indizes. Diese dauert mit Binary Search jeweils O(log(n)). Pro Buchstabe wird diese Funktion zwei mal ausgeführt. (Zusätzlich kommt die Sortierung der Wörterliste am Anfang noch hinzu), jedoch verringert es die Laufzeit des gesamten Programmes immens, deshalb sind es diese Reglementierungen wert.

Reichweite der Wörterliste

Mit Hilfe von Binary-Search evaluiert man die Reichweite von dem jeweiligen Input-Charakter.

```
39  def getStartEnd(letter):
40      start = binarySearchStart(letter, 0, len(wordlist)-1)
41      end = binarySearchEnd(letter, start, len(wordlist)-1)
42      return (start, end)
43
```

Funktionsweise:

- in der *binarySearchStart*-Funktion wird mit dem Prinzip von Binary Search der Index der unteren Grenze eines Buchstabens wiedergegeben

```
10  def binarySearchStart(char, s, v):
11      index = s+((v-s)//2)
12      if index==0 or (char is wordlist[index][0] and wordlist[index][0] is not wordlist[index-1][0]):
13          return index
14      if (char>wordlist[index][0]):
15          return binarySearchStart(char, index, v)
16      elif (char<=wordlist[index][0]):
17          return binarySearchStart(char, s, index)
18
```

- … und bei der *binarySearchEnd*-Funktion das obere Ende (+1)

```
24  def binarySearchEnd(char, s, v):
25      index = s+((1+v-s)//2)
26      if index >= len(wordlist)-1 or (wordlist[index][0] is not char and wordlist[index-1][0] is char):
27          return index
28      if (char>=wordlist[index][0]):
29          return binarySearchEnd(char, index, v)
30      elif (char<wordlist[index][0]):
31          return binarySearchEnd(char, s, index)
```

Beispiele

1.

INPUT:

Der Twist
(Englisch twist = Drehung, Verdrehung)
war ein Modetanz im 4/4-Takt,
der in den frühen 1960er Jahren populär
wurde und zu
Rock'n'Roll, Rhythm and Blues oder spezieller
Twist-Musik getanzt wird.

TWISTED:

Der Twsit
(Elsngcih twsit = Deurhng, Vendreurhg)
war ein Meatdnoz im 4/4-Takt,
der in den fherün 1960er Jeharn poälupr
wrude und zu
Rcok'n'Rlol, Rthyhm and Blues oedr sllizeeepr
Tiwst-Msuik gznaett wrid.

UNTWISTED:

Der Twsit
(Englisch twsit = Drehung, Verdrehung)
war ein Meatdnoz im 4/4-Takt,
der in den frühen 1960er Jahren populär
wurde und zu
Rock'n'Rlol, Rthyhm and Blues oder spezieller
Tiwst-Musik getanzt wird.

2.

INPUT:

Hat der alte Hexenmeister sich doch einmal wegbegeben! Und nun sollen seine Geister auch nach meinem Willen leben. Seine Wort und Werke merkt ich und den Brauch, und mit Geistesstärke tu ich Wunder auch.

TWISTED:

Hat der atle Heietmexensr sich dcoh einaml wbegebeegn! Und nun seolln senie Gietesr acuh nach meinem Wleiln leben. Sinee Wort und Wrkee mekrt ich und den Bcruah, und mit Getssästeikre tu ich Wendur acuh.

UNTWISTED:

Hat der alte Hexenmeister sich doch einmal wbegebeegn! Und nun sollen seine Geister auch nach meinem Willen leben. Seine Wort und Werke merkt ich und den Brauch, und mit Geistesstärke tu ich Wunder auch.

3.

INPUT:

Ein Restaurant, welches a la carte arbeitet, bietet sein Angebot ohne eine vorher festgelegte Menüreihenfolge an. Dadurch haben die Gäste zwar mehr Spielraum bei der Wahl ihrer Speisen, für das Restaurant entstehen jedoch zusätzlicher Aufwand, da weniger Planungssicherheit vorhanden ist.

TWISTED:

Ein Rturasaent, weelhcs a la crate ateriebt, biteet sien Angebot ohne enie vorher felteggstee Mlhüfgeeiroenne an. Dacrduh heban die Gstäe zwar mehr Sluapriem bei der Whal ihrer Sspieen, für das Rsrnuaetat enhsteten jedcoh zhcsuztielär Aafnuwd, da wgneier Pnschageliiurnhest vaodrnhen ist.

UNTWISTED:

Ein Restaurant, welches a la ctrae abrietet, bietet sein Angebot ohne eine vorher festgelegte Mnegrnüeoeflhie an. Dadurch haben die Gäste zwar mehr Spielraum bei der Wahl ihrer Speisen, für das Restaurant entstehen jedoch zusätzlicher Aufwand, da weniger Pegilrnnsachsuieht vorhanden ist.

4.

INPUT:

Augusta Ada Byron King, Countess of Lovelace, war eine britische Adelige und Mathematikerin, die als die erste Programmiererin überhaupt gilt. Bereits 100 Jahre vor dem Aufkommen der ersten Programmiersprachen ersann sie eine Rechen-Mechanik, der einige Konzepte moderner Programmiersprachen vorwegnahm.

TWISTED:

Auugsta Ada Boryn Knig, Cuetnoss of Loaevlce, war eine bcshiitre Aeilgde und Mkitehamieratn, die als die ertse Paigrrrmroeemin üauerhbpt glit. Beretis 100 Jhare vor dem Afokmmuen der ersten Phasregmaecroipmrrn easnrn sie eine Rechen-Mcihaenk, der eingie Kzeptone moenedrr Parsohmriegcprmaern vnwaeoghrm.

UNTWISTED:

Auugsta Ada Boryn Knig, Cuetnoss of Loaevlce, war eine britische Adelige und Mathematikerin, die als die erste Programmiererin überhaupt gilt. Bereits 100 Jahre vor dem Aufkommen der ersten Programmiersprachen ersann sie eine Rechen-Mechanik, der einige Konzepte moderner Programmiersprachen vorwegnahm.

5.

INPUT:

Alice fing an sich zu langweilen; sie saß schon lange bei ihrer Schwester am Ufer und hatte nichts zu thun. Das Buch, das ihre Schwester las, gefiel ihr nicht; denn es waren weder Bilder noch Gespräche darin. „Und was nützen Bücher," dachte Alice, „ohne Bilder und Gespräche?"

Sie überlegte sich eben, (so gut es ging, denn sie war schläfrig und dumm von der Hitze,) ob es der Mühe werth sei aufzustehen und Gänseblümchen zu pflücken, um eine Kette damit zu machen, als plötzlich ein weißes Kaninchen mit rothen Augen dicht an ihr vorbeirannte.

[…]

Es war hohe Zeit sich fortzumachen; denn der Pfuhl begann von allerlei Vögeln und Gethier zu wimmeln, die hinein gefallen waren: da war eine Ente und ein Dodo, ein rother Papagei und ein junger Adler, und mehre andere merkwürdige Geschöpfe. Alice führte sie an, und die ganze Gesellschaft schwamm an's Ufer.

TWISTED:

Aclie fnig an scih zu lewgnlaien; sie saß scohn lgane bei ihrer Ssewecthr am Ufer und httae nichts zu thun. Das Buch, das irhe Sesthcewr las, gefeil ihr nihct; denn es wearn weder Bldeir noch Gäphecrse diarn. „Und was netzün Bücehr," dthcae Alcie, „ohne Biedlr und Grhesäpce?"

Sie übetrelge sich eben, (so gut es gnig, denn sie war srilhäcfg und dmum von der Hztie,) ob es der Mühe werth sei asfhzeetuun und Gbeeähnüclmsn zu pflküecn, um enie Ktete dmait zu mceahn, als pcltözilh ein weßies Kacninhen mit rehton Augen dihct an ihr vrnnrtabieoe.

[…]

Es war hohe Zeit scih fetumorahzcn; dnen der Pfhul baengn von allrelei Velgön und Githeer zu wilmemn, die hienin gllfeaen wraen: da war eine Etne und ein Dodo, ein rohetr Paeapgi und ein jguenr Adler, und mhere aendre mrrkdiügewe Göshcefpe. Ailce füthre sie an, und die gazne Ghflcessleat smhwcam an's Uefr.

UNTWISTED:

Alice fing an sich zu langweilen; sie saß schon lange bei ihrer Schwester am Ufer und hatte nichts zu thun. Das Buch, das ihre Schwester las, gefiel ihr nicht; denn es waren weder Bilder noch Gespräche darin. „Und was nützen Bücher," dachte Alice, „ohne Bilder und Gespräche?"

Sie überlegte sich eben, (so gut es ging, denn sie war schläfrig und dumm von der Hitze,) ob es der Mühe werth sei aufzustehen und Gänseblümchen zu pflücken, um eine Kette damit zu machen, als plötzlich ein weißes Kaninchen mit rehton Augen dicht an ihr vrnnrtabieoe.

[…]

Es war hohe Zeit sich fetumorahzcn; denn der Pfuhl bangen von allerlei Vögeln und Githeer zu wimmeln, die hinein gefallen waren: da war eine Ente und ein Dodo, ein rohetr Papagei und ein junger Adler, und mehre andere merkwürdige Geschöpfe. Alice führte sie an, und die ganze Gesellschaft schwamm an's Ufer.

6.

INPUT:

leeres Dokument

TWISTED:

leeres Dokument

UNTWISTED:

leeres Dokument

7.

INPUT:

1337

TWISTED:

1337

UNTWISTED:

1337

Quellcode:

```
import random as r
import re
import time

with open('wordlist.txt') as f:
    wordlist = f.readlines()
## THE SORTING OF wordlist IS MANDATORY BECAUSE OF THE BINARY-SEARCH PROCESS
wordlist.sort() # nlogn

## EVALUATE THE INDEX WHERE THE CHARACTER IN THE wordlist IS ON THE VERGE TO ITS
PREDECESSOR
def binarySearchStart(char, s, v):
    ## THE INDEX BETWEEN S AND V
    index = s+((v-s)//2)
    ## CONDITION MENTIONED ABOVE IS ACHIEVED OR IS AT THE VERY BEGINNING OF
wordlist
    if index==0 or (char is wordlist[index][0] and wordlist[index][0] is not
wordlist[index-1][0]):
        return index
    ## BASIC BINARY-SEARCH:
    if (char>wordlist[index][0]):
        return binarySearchStart(char, index, v)
    # NOTE: char = wordlist[index][0] BECAUSE IF char IS IN THE CORRECT RANGE
ALEADY IT STILL IS SUPPOSED TO JUMP TO THE BEGINNING LEDGE
    elif (char<=wordlist[index][0]):
        return binarySearchStart(char, s, index)

def binarySearchEnd(char, s, v):
    index = s+((1+v-s)//2)
    if index >= len(wordlist)-1 or (wordlist[index][0] is not char and
wordlist[index-1][0] is char):
        return index
    if (char>=wordlist[index][0]):
        return binarySearchEnd(char, index, v)
    elif (char<wordlist[index][0]):
        return binarySearchEnd(char, s, index)

## BASICALLY THE SAME ALGORITHM AS ABOVE, BUT EACH TIME A WORD GETS TWISTED
ABOVE, IT GETS UNTWISTED HERE
def getStartEnd(letter):
    start = binarySearchStart(letter, 0, len(wordlist)-1)
    end = binarySearchEnd(letter, start, len(wordlist)-1)
    return (start, end)
```

```
## TWISTS AN INPUT WORD ARBITRARILY -- ONLY THE FIRST AND LAST CHARACTER STAY
UNTOUCHED
def twistify(word):
    ## RANGING FROM THE FIRST TO SECOND-LAST AND MIXING THE LETTERS
    # NOTE: IF THE ARBITRARY INTEGERS ARE THE SAME NOTHING CHANGES
    ## -- WORDS MAY ACCIDENTALLY STAY THE SAME -- I DIDNOT SEE SPECIFICATION FOR
THIS CASE SO I WILL IGNORE IT
    for i in range(1, len(word)-1):
        word=swap(r.randint(1, len(word)-2), r.randint(1, len(word)-2), word)
    return word

## WITH TWO INDICES AND AN INITIAL WORD -- THE ELEMENTS AT THE INDICES
RESPECTIVELY SWAP
def swap(i,j,input):
    word=list(input)
    word[i],word[j]=word[j],word[i]
    return ''.join(word)
```

```
## INTAKING A PATH TO A TXT-FILE -- APPLY THE twistify-METHOD TO EACH WORD
def twistifySentence(path):
    result=[]
    startWord = False
    startSChar = False
    with open(path) as f:
        data = f.read()
    ## SEPARATING WORDS IN GERMAN.. (ARE THE ONLY ONES BEING TWISTED)
    words = re.findall(r"[a-zA-ZöÖüÜäÄß]+", data)
    ## ..FROM THE NON-READABLE CHARACTERS (I.E. NUMBERS, PUNCTUATION. ETC.)
    specialChars = re.findall(r"[^a-zA-ZöÖüÜäÄß]+", data)
    ## IF A WORD IS COMMENCING THE STRING THIS IS TRUE
    if len(words)>0:
        startWord = bool(re.match(words[0], data))
    ## VICE VERSA WITH specialChars
    if len(specialChars)>0:
        startSChar = bool(re.match(specialChars[0], data))
    ##  IF THE STRING BEGINS WITH A WORD
    ##  IT MEANS THAT THE LENGTH OF specialChars CAN ONLY SMALLER-OR-EQUAL TO
THE LENGTH OF words
    if  startWord:
        for iter in range(0, len(specialChars)):
            result.append(twistify(words.pop(0))+specialChars.pop(0))
    ## VICE VERSA
    if startSChar:
        for iter in range(0, len(words)):
            result.append(specialChars.pop(0)+twistify(words.pop(0)))
    ## IF THERE ARE LEFT-OVER ELEMENTS THEY GET INSERTED HERE
    if len(words)>0:
        result.append(twistify(words.pop(0)))
    elif len(specialChars)>0:
        result.append(specialChars.pop(0))
    ## RETURN RESULT
    return ''.join(result)
```

```
## DETERMINES WETHER A TWISTED WORD IS IN THE wordlist DICTIONARY(IF FOUND =>
RETURNING IT)
def untwistify(twisted):
    ## ESTABLISHES IF THE FIRST LETTER IS UPPER-CASE (THIS WORD MAY NOT EXACTLY
BE IN THE DICTIONARY IF IT IS 'SUBSTANTIVIERT')
    firstUpper = twisted[0].isupper()
    ## EVALUATE THE INTERVAL WHERE ALL THE WORDS WITH THE FIRST LETTER(IN LOWER
CASE) LIE
    # BUG: chr(ord(...)) IS NECESARRY
    start, end = getStartEnd(chr(ord(twisted[0].lower())))
    ## ALL RESULTS WHERE THE FIRST- AND LAST CHARACTER ARE THE SAME AND LENGTHS
CONCUR
    # NOTE: re.IGNORECASE -> BECAUSE THE OF THE SAME REASON FROM firstUpper
    possibleWords=[]
    for dictWord in range(start, end):
        match = re.findall(\
        r'\b'+twisted[0]+r'[a-zA-ZöÖüÜäÄß]
{'+str(len(twisted)-2)+r'}'+twisted[len(twisted)-1]+r'\b'
        , wordlist[dictWord], re.IGNORECASE)
        if len(match)>0:
            possibleWords.append(match[0])
    ## SAME PROCESS AS ABOVE JUST IN CASE THE FIRST LETTER IS UPPER-CASE
    if firstUpper:
        start, end = getStartEnd(twisted[0])
        for dictWord in range(start, end):
            match = re.findall(\
            r'\b'+twisted[0]+r'[a-zA-ZöÖüÜäÄß]
{'+str(len(twisted)-2)+r'}'+twisted[len(twisted)-1]+r'\b'
            , wordlist[dictWord])
            if len(match)>0:
                possibleWords.append(match[0])
    ## FOR EVERY word OUT OF THE possibleWords IT CHECKS WETHER THE FOLLOWING
CONSTRAINTS APPLY
    for word in possibleWords:
        correctWord=True
        ## IN AN INTERVAL FROM THE SECOND TO SECOND-LAST CHARACTER
        for i in range(1, len(twisted)-1):
   ## IF A CHARACTER APPEARANCE DIFFERS IN EITEHR WORDS IT GETS EXCLUDED
            if  len(re.findall(twisted[i], twisted[1:len(twisted)-1])) is not
len(re.findall(twisted[i], word[1:len(twisted)-1])):
                correctWord=False
                break
        ## IF IT DIDNOT BREAK IT CONNOTES THE word BEING CORRECT
        if correctWord:
            ## RAISES THE FIRST CHARACTER TO UPPER-CASE IF TEH INITIAL WORD WAS
SO
            ## ALTHOUGH IF IT IS A NOUN AND ALREADY UPPER-CASE IT STILL ASURES
FOR THE INFAMOUS 'SUBSTANTIVIER'-CASE
            if firstUpper:
                word = word[0].upper()+word[1:]
            ## RETURNS FOUND WORD
```

```
            return word
    ## THE WORD WAS NOT IN THE DICTIONARY AND GETS RETURNED UNALTERED
    return twisted
```

```
def untwistifySentence(path):
    result=[]
    startWord = False
    startSChar = False
    with open(path) as f:
        data = f.read()
    ## SEPARATING WORDS CONSISTING OF GERMAN CHARACTERS.. (ARE THE ONLY ONES
BEING UNTWISTED)
    words = re.findall(r"[a-zA-ZöÖüÜäÄß]+", data)
    ## ..FROM THE NON-READABLE CHARACTERS (I.E. NUMBERS, PUNCTUATION. ETC.)
    specialChars = re.findall(r"[^a-zA-ZöÖüÜäÄß]+", data)
    ## IF A WORD IS COMMENCING THE STRING THIS IS TRUE
    if len(words)>0:
        startWord = bool(re.match(words[0], data))
    ## VICE VERSA WITH specialChars
    if len(specialChars)>0:
        startSChar = bool(re.match(specialChars[0], data))
    ##  IF THE STRING BEGINS WITH A WORD
    ##  IT MEANS THAT THE LENGTH OF specialChars CAN ONLY SMALLER-OR-EQUAL TO
THE LENGTH OF words
    if  startWord:
        for iter in range(0, len(specialChars)):
            result.append(str(untwistify(words.pop(0)))
+str(specialChars.pop(0)))
    ## VICE VERSA
    elif startSChar:
        for iter in range(0, len(words)):
            result.append(specialChars.pop(0)+untwistify(words.pop(0)))
    ## IF THERE ARE LEFT-OVER ELEMENTS THEY GET INSERTED HERE
    if len(words)>0:
        result.append(untwistify(words.pop(0)))
    elif len(specialChars)>0:
        result.append(specialChars.pop(0))
    ## RETURN RESULT
    return ''.join(result)

# ------------------------------------------------- #
timeResult=""
for num in range(1, 8):
    start=time.time()
    with open("toUntwist/untwist"+str(num)+".txt", "w") as f:
        f.write(twistifySentence('toTwist/twist'+str(num)+'.txt'))
    with open("twistedUntwisted/twist"+str(num)+"_output.txt", "w") as f:
        f.write(untwistifySentence('toUntwist/untwist'+str(num)+'.txt'))
    end=time.time()
    timeResult+=("{} took {} amount of seconds\n".format(num, end-start))
with open("time.txt", "w") as timef:
    timef.write(timeResult)
```