# Aufgabe 2: Twist

Team-ID: 00133

Team-Name: CubeFlo

Bearbeiter dieser Aufgabe:
Florian Rädiker (15 Jahre)

18. November 2018

## Inhalt

# Lösungsidee

## Twisten

Das Twisten ist sehr simpel: Hier wird die Mitte (also alles außer den ersten und letzten Buchstaben) aller Wörter im Text zufällig vermischt, da beim Twisten immer nur der erste und letzte Buchstabe stehen bleiben. Darauf achten, dass auch wirklich ein getwistetes Wort herauskommt, das sich vom ursprünglichen unterscheidet, muss man nicht, wie das gegebene Beispiel anhand des Wortes „populär“ zeigt. „populär“ ist im getwisteten Text nämlich unverändert und enthält auch keine Buchstaben doppelt, die als vertauscht angesehen werden könnten.

## Enttwisten

Das Enttwisten ist da schon wesentlich komplizierter. Grundlegend wird auch wieder der Text in seine einzelnen Worte zerlegt und jedes dieser Worte wird mit denen in der Wörterliste auf „Ähnlichkeit“ überprüft, wobei Ähnlichkeit denselben ersten und letzten Buchstaben, aber einen aus den gleichen Buchstaben bestehenden Mittelteil meint. Ein Wort muss also in eine Variante „übersetzt“ werden, bei der bei allen Wörtern, die sich ähnlich sind, dasselbe herauskommt. Durch alphabetische Sortierung des Mittelteils ohne Veränderung des ersten und letzten Buchstabens erhält man immer das gleiche Wort, egal, ob das Wort getwistet ist oder nicht. Aber einem auf diese Weise sortierten Wort können in manchen Fällen auch mehrere Wörter zugeordnet werden, wie dies zum Beispiel bei „frühen“ (Beispiel 1) bzw. „führen“ der Fall ist, da hier die Buchstaben im Mittelteil und der erste und letzte Buchstabe übereinstimmen. Deshalb muss man für ein sortiertes Wort immer mehrere mögliche enttwistete Wörter bereitstellen; dies erfolgt in Form eines Dictionaries, das einem sortierten Wort immer alle enttwisteten Möglichkeiten als Liste zuordnet.

Dieses System kann man natürlich noch ausbauen und erweitern, so beispielsweise mit zusammengesetzten Wörtern. Diese Erweiterungen sind hier aufgeführt; teilweise gibt es Überschneidungen mit der Umsetzung, um Zusammenhänge besser zu erklären. Die meisten der Erweiterungen können in bestimmten Funktionen auch abgeschaltet werden, um das Enttwisten je nach Text anzupassen. Um z.B. große Texte zu enttwisten, sollte vielleicht nicht nach zusammengesetzten Wörtern gesucht werden, da dies zeitaufwändig sein kann. Dazu siehe die Umsetzung.

### *Kleine Wörter nicht (ent-)twisten*

Wörter, die kleiner als 4 Buchstaben sind, werden weder getwistet noch enttwistet, da dies keine Änderung verursacht.

### *Enttwistete Wörter cachen*

Die Funktion, die ein Wort enttwistet, ist gecacht, sodass sie bei Gleichheit des sortierten Wortes nicht noch einmal aufgerufen werden muss.

### *Unabhängig von Groß-/Kleinschreibung*

Es gibt in der Wörterliste zum Teil Wörter, die nur in kleingeschriebener Form vorliegen, wie z.B. „englisch“. Es existiert aber auch das Wort „Englisch“, nur ist dieses nicht in der Wörterliste vorhanden. Also wird Groß- und Kleinschreibung ignoriert, indem zusätzlich zur alphabetischen Sortierung auch noch alle Großbuchstaben in Kleinbuchstaben umgewandelt werden und auch die Dictionaries entsprechend kleingeschriebene Wörter als Schlüssel haben.
Dabei wird jedoch die Groß-/Kleinschreibung des Ursprungswortes wieder für das Ergebnis

übernommen, also wird aus „Eigsnclh“ zuerst „ecgilnsh“ und dieser Sortierung zugeordnet wird das Wort „englisch“ gefunden, welches nun dieselbe Groß-/Kleinschreibung, also „Englisch“, bekommt.

### *Zusammengesetzte Wörter*

Bei zusammengesetzten getwisteten Wörtern ist es teilweise auch für Menschen schwierig, sie zu entziffern, und bei Wörtern, die aus mehr als zwei Wörtern zusammengesetzt sind, gibt es nur den ersten und letzten Buchstaben als Anhaltspunkt, was sehr wenig ist. Deshalb werden hier nur Zweiwortverbindungen berücksichtigt. Außerdem sind Wörter unter einer Länge von sieben Buchstaben normalerweise keine Wortverbindung, deshalb werden Wörter mit sechs Buchstaben oder weniger auch nicht berücksichtigt, um Zeit zu sparen. Es existieren zwei Dictionaries, in denen jedem ersten bzw. letzten Buchstaben ein weiteres Dictionary zugeordnet ist, das die sortierte Variante jedes Wortes ohne den letzten bzw. ersten Buchstaben allen Wörtern zuordnet, die dafür in Frage kommen. Beispielsweise wird unter dem Buchstaben „a“ im Dictionary mit den Anfangsbuchstaben „nsi“ (also „ansi“ mit dem Anfangsbuchstaben) sowohl „ANSI“ als auch „Anis“ zugeordnet.
Sollte zu einem Wort nicht auf einfache Weise wie oben beschrieben die enttwistete Variante gefunden werden, dann könnte es ein zusammengesetztes Wort sein und wird deshalb mit folgendem Prozess bearbeitet:
Zuerst wird aus dem Dictionary mit den Anfangsbuchstaben das Dictionary genommen, das die Wörter mit demselben Anfangsbuchstaben enthält wie das zu enttwistende Wort. Dasselbe geschieht mit dem letzten Buchstaben. Alle Schlüssel des Dictionarys, das die Wörter mit demselben Anfangsbuchstaben enthält, enthalten das Wort ohne den Anfangsbuchstaben. Wenn dieser Teil vollständig im Mittelteil des zu enttwistenden Wortes enthalten ist, dann könnte zumindest dieses erste Wort stimmen. Wenn man das erste mögliche Wort vom Mittelteil des zu enttwistenden Wortes „abzieht“, erhält man die Buchstaben des zweiten Wortes. Wenn sich diese Buchstaben nun im Dictionary mit den zweiten Wörtern wiederfinden, so ist dies eine mögliche Kombination. Da den Schlüsseln aber immer auch mehrere Wörter zugeordnet sein können, existieren mehrere mögliche Kombinationen aus allen Wörtern.
Ein Beispiel veranschaulicht dieses Vorgehen: Zum Wort „Patunebetrn“ („Putenbraten“, sortiert „pabeenrttun“) kann kein passendes Wort gefunden werden. Also werden mögliche Wortverbindungen gesucht. Es gibt viele Wörter, die mit „p“ anfangen, wie z.B. „PC“, „Panter“, „Peru“/ „pure“ oder „Puten“. Davon passt aber „PC“ schon mal nicht, weil im Mittelteil kein „c“ vorkommt. Zieht man „anter“ („Panter“ ohne „p“) vom Mittelteil „abeenrttu“ ab, so bleibt „tbeu“ übrig. Zu diesem Wortanfang ohne die Endung „n“ gibt es ein Wort – nämlich „Tuben“. Diese beiden Wörter zusammen würden „Pantertuben“ (mit übernommener Groß/Kleinschreibung) ergeben. Ähnlich bleibt, wenn man „uten“ (von „Puten“) von „abeenrttu“ abzieht, „baertn“ übrig. Auf „baertn“ passen zwei Wörter, nämlich „Brate(n)“ und „trabe(n)“, also gibt es noch zwei Möglichkeiten: „Putenbraten“ und „Putentraben“.
Hier ist schon zu sehen, dass zusammengesetzte Wörter in bestimmten Fällen keinen Sinn ergeben. Besser wäre es, nur spezielle Wörter, aus denen auch zusammengesetzte Wörter gebildet werden können, zu benutzen, und diese auch nach Vorkommen als erstes/zweites Wort zu ordnen. „weg“ (von z.B. „weggehen“) kommt beispielsweise nur am Anfang vor. Da dies die Wörterliste aber nicht hergibt, ist es auch nicht implementiert.
Was ist aber mit Wörtern wie „Planungssicherheit“, bei denen noch ein Fugen-s dazukommt? Im Programm wird immer, wenn kein passendes zweites Wort bzw. keine passenden zweiten Wörter gefunden wurden und der restliche Mittelteil ohne das erste Wort ein „s“ enthält, versucht, zweite

Wörter zu finden, allerdings ohne „s“. Das „s“ wird dann später wieder eingefügt, wenn ein passendes Wort bzw. passende Wörter gefunden wurden.

### *Optionen beim Enttwisten*

Dem Benutzer werden beim Enttwisten verschiedene Optionen zur Verfügung gestellt, um angepasste Ergebnisse zu erzielen. Diese sind hier aufgelistet:

- Groß-/Kleinschreibung beachten: Auswählen, ob Groß- und Kleinschreibung berücksichtigt werden soll (empfehlenswert ist, es nicht zu beachten (s.o.))
- Alle Ergebnisse anzeigen: Auswählen, ob alle gefundenen Wörter durch „/“ getrennt ausgegeben werden sollen oder nur ein Wort ausgegeben wird
- Nach zusammengesetzten Wörter suchen: Die Suche nach zusammengesetzten Wörtern kann bei großen Texten sehr zeitaufwändig sein, deshalb kann sie mit einer Mindestanzahl von Buchstaben (standardmäßig 7, siehe oben) versehen werden, sodass nur Wörter überprüft werden, die der Mindestanzahl an Buchstaben entsprechen.
- Groß-/Kleinschreibung übernehmen: Wenn ohne Berücksichtigung von Groß- und Kleinschreibung gesucht wird, kann die Groß- und Kleinschreibung vom ursprünglichen Wort auf die Ergebnisse angewandt werden.
- Warnungen anfügen: Sollte ein Wort nicht enttwistet werden können, wird „(!)“ vor das Wort gestellt, sollte es mehrere Möglichkeiten geben, die aber nicht alle angezeigt werden sollen (aufgrund der obigen Option), kann „(?)“ vorgestellt werden.
- Schnellere Variante: Alle oben genannten Optionen werden standardmäßig eingeschaltet, es existieren Funktionen, die im Prinzip genau gleich aufgebaut sind, aber auf wiederholte `if`-Bedingungen für die An- und Abschaltung der Optionen verzichten und schlichtweg alles anwenden, um schneller zu Enttwisten.

## Umsetzung

Das Programm ist in Python 3 geschrieben.
Am wichtigsten ist die Klasse `Twister` in der Datei `Aufgabe2/twister.py`, die Funktionen bereitstellt, um Texte oder Wörter zu twisten und enttwisten.
Sowohl zum Twisten als auch zum Enttwisten müssen alle Wörter aus einem als Parameter übergebenen Text gefiltert werden, um dann einzeln ge- bzw. enttwistet zu werden. Dafür werden reguläre Ausdrücke mit dem Modul `re` verwendet. Der reguläre Ausdruck zum Filtern der Wörter sieht so aus: `[a-zA-ZäöüÄÖÜßß]{4,}` (siehe `Twister.WORD_PATTERN` im Quellcode). Innerhalb der eckigen Klammern befinden sich alle Zeichen, aus denen ein Wort im Deutschen (normalerweise) besteht, also den Buchstaben von A bis Z Groß und Klein inklusive der Umlaute und dem großen und kleinen Eszett. `{4,}` besagt, dass das Wort mindestens 4 Buchstaben besitzen muss, ansonsten würde weder Twisten noch Enttwisten etwas verändern. Mittels der Funktion `re.sub` wird jedes Vorkommen dieses regulären Ausdrucks entweder von der Twist- oder der Enttwist-Funktion bearbeitet und verändert.

### Twisten

Zum Twisten benutzt die Funktion `Twister.twist_match` das Modul `random`, um per `random.shuffle` den Mittelteil zu mischen und anschließend mit dem ersten und letzten Buchstaben angefügt wieder zurückzugeben. Die Funktion bekommt als Parameter direkt ein `match`-Objekt der `re.sub`-Funktion, sodass dies nicht noch vorher in einen String umgewandelt werden muss und somit

schneller ist. Eine Funktion zum Twisten von Wörtern existiert aber auch und heißt `Twister.twist_word`. Das Twisten eines Textes übernimmt die Funktion `Twister.twist_text`, die mit dem regulären Ausdruck sucht.

### Enttwisten

Das Enttwisten eines Textes wird von der Funktion `Twister.untwist_text` übernommen, die auch mit dem regulären Ausdruck sucht und mit dem Ergebnis der Funktion `Twister.untwist_word` ersetzt. Diese Funktionen stellen alle unter „Lösungsidee > Optionen Beim Enttwisten“ genannten Optionen (außer der letzten) in Form von Parametern bereit, während die Funktionen `Twister.untwist_text_fast` und `Twister.untwist_sorted_word_fast` nach der letzten Option handeln und keine Optionen bieten, dafür aber auch schneller sind. Weil Funktionen mit zusätzlichen Optionen nicht gefordert sind und leichter zu erklären sind, beziehe ich mich im Folgenden nur auf diese einfacheren Funktionen. `Twister.untwist_word_fast` bekommt als Parameter nur das unbearbeitete Wort. Die Funktion ist trotzdem gecacht, sodass sie bei Wörtern mit exakt gleicher Schreibweise nicht noch einmal aufgerufen wird. Diese Funktion übernimmt auch die Groß- und Kleinschreibung des getwisteten Wortes auf das Ergebnis und ruft zum Enttwisten die ebenfalls gecachte Funktion `Twister.untwist_sorted_word_to_list` auf. Diese erhält als Parameter das sortierte, kleingeschriebene Wort und ist somit effizienter gecacht. Sie gibt entweder eine Liste mit den Ergebnissen oder das sortierte Wort, das als Parameter übergeben wurde, zurück. Durch den Cache müssen für ähnliche Wörter (die in kleingeschriebener Sortierung also gleich sind) nicht noch einmal alle Schritte durchgeführt werden.
Um zusammengesetzte Wörter zu enttwisten gibt es noch eine Funktion, `Twister.untwist_compound_word_fast`, die nach dem unter „Lösungsidee“ beschriebenen Verfahren arbeitet.

#### *Optimierung der Wörterliste*

Die Verwaltung der Wörterliste übernimmt die Klasse `TwistingDict` im Modul `Aufgabe2/twisting_dict.py`.

Es müssen vier Wörterlisten gegeben sein, um alle Optionen zu benutzen. Dies sind:

- Ein `dict`, das die sortierte Variante mit richtiger Groß- und Kleinschreibung den möglichen Wörtern zuordnet (`TwistingDict.words`) (wird hier nicht verwendet)
- Ein `dict`, das die sortierte, kleingeschriebene Variante den möglichen Wörtern (in richtiger Schreibweise) zuordnet (`TwistingDict.words_lower`)
- Zwei Dictionaries, die je die Wörter nach erstem und letztem Buchstaben sortieren und die sortierten Teile ohne ersten bzw. letzten Buchstaben den möglichen Wörtern zuordnen (siehe das Verfahren zu zusammengesetzten Wörtern unter „Lösungsidee“) (`TwistingDict.words_start` bzw. `TwistingDict.words_end`)

Alle Dictionaries nur aus der Wörterliste zu laden würde sehr viel Zeit beanspruchen, und deshalb wurden schon vorab Dateien erstellt, die die Dictionaries in aufbereiteter Form speichern. Dies ist nicht zwingend notwendig, trägt aber zu einem schnelleren Einlesen bei. Die Erstellung der Dateien wurde vom Skript `Aufgabe2/create_wordlists.py` übernommen, deren Verfahren hier nicht weiter beschrieben wird, da dies nicht direkt zur Lösung gehört.
Die beiden erstgenannten Dictionaries sind beide in der Datei `Aufgabe2/words` gespeichert. Dort sind pro Zeile einer sortierten Variante durch ein `#` abgetrennt alle darauf passenden Wörter zugeordnet.

Sollte es mehrere Möglichkeiten geben, sind die Wörter durch Kommata voneinander getrennt, also sieht eine Zeile zum Beispiel so aus: `Achse#Achse,Asche`. Aus dieser Datei liest `TwistingDict.init_words` alle Wörter ein und fügt sie in der Variante, wie sie dort stehen, `TwistingDict.words` hinzu. Die Schlüssel von `TwistingDict.words_lower` werden zusätzlich zum Sortieren, was bereits in der Datei steht, in Kleinbuchstaben umgewandelt.
Die anderen beiden Dictionaries sind in viele Dateien aufgeteilt, die sich je nach dem ersten bzw. letzten Buchstaben benennen. Die Dateien `Aufgabe2/dict/start/a`, `/start/b`, `/start/…` sind für `TwistingDict.words_start`, die Dateien unter `Aufgabe2/dict/end/` sind für `TwistingDict.words_end`. Jede Datei enthält wieder pro Zeile ein sortiertes, aber kleingeschriebenes Wort (zusätzlich ohne ersten bzw. letzten Buchstaben), und dazu passende Wörter, die gegebenenfalls durch Kommata voneinander getrennt sind. Unter `dict/start/a` ist zum Beispiel die Zeile `nsi#ANSI,Anis` zu finden oder unter `dict/end/ü` `hauptmen#Hauptmenü`.

### *HashableCounter – Rechnen mit Buchstaben*

Um die verschiedenen Wortteile wie in der Lösungsidee für zusammengesetzte Wörter beschrieben voneinander subtrahieren zu können, werden diese Wortteile (also auch die Schlüssel der entsprechenden zwei Dictionaries) nicht als sortiertes Wort in Form eines Strings, sondern als `HashableCounter`, einer Klasse, die von `Counter` aus dem Modul `collections` erbt, gespeichert. `Counter`-Objekte zählen die Vorkommnisse von Buchstaben in einem Wort und stellen sie als `dict` dar. Zum Beispiel „Programmierer“ wird zu `Counter({'r': 4, 'm': 2, 'e': 2, 'P': 1, 'o': 1, 'g': 1, 'a': 1, 'i': 1})`.
Die `HashableCounter`-Klasse aus dem Modul `Aufgabe2/hashable_counter.py` ist `hashable`, um als Schlüssel für Dictionaries verwendet zu werden, und kann ein anderes `HashableCounter`-Objekt subtrahieren, solange der Minuend mehr Buchstaben enthält als der Subtrahend. Also ist `HashableCounter("hashablecounter") - HashableCounter("achterbahn")` das gleiche wie `HashableCounter({'s': 1, 'l': 1, 'e': 1, 'o': 1, 'u': 1})`, wohingegen `HashableCounter("Florian") - HashableCounter("florian")` einen `ArithmeticError` produziert, da nicht alle Buchstaben vom Subtrahend auch im Minuend vorkommen; der `HashableCounter` ist natürlich case-sensitive und daher liegen alle Buchstaben auch nur in Kleinbuchstaben vor. Wie genau das funktioniert ist im Quellcode nachzulesen.

## Beispiele

Im Folgenden werden alle Beispieldaten der BwInf-Webseite grundlegend besprochen, um einen Überblick darüber zu geben, was das Programm alles kann und nicht kann.

**twist1.txt**

*Der (!)Tiswt*
*(Englisch (!)tiswt = Drehung, Verdrehung)*
*war ein Modetanz/Monatdez im 4/4-Takt,*
*der in den frühen/führen 1960er Jahren populär*
*wurde und zu*
*Rock'n'(!)Rlol, (!)Rhhtym and Blues oder spezieller*
*(!)Tiswt-Musik getanzt wird.*

Alle Wörter außer „Twist“, „Roll“ und „Rhythm“ werden richtig erkannt, was auch dementsprechend gekennzeichnet ist. Diese Wörter entstammen aber auch keinem gewöhnlichen deutschen

Wortschatz und sind auch nicht in der Liste vorhanden. Einziges zusammengesetztes Wort ist „Modetanz“, wo auch die Variante „Monatdez“ gefunden wird, da keine Informationen über die Eignung von Wörtern für Wortverbindungen vorliegt.

### twist2.txt

*Hat der alte Hexenmeister sich doch einmal wegbegeben/weggebeben!*
*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.*

Es gibt zwei Möglichkeiten, „wegbegeben“ zu interpretieren (von denen die eine keinen Sinn ergibt). Hier ist auch gut die Verwendung des Substantivs „Weg“ fälschlicherweise als Präfix „weg-“ zu erkennen. Die Wortverbindungen „Geistesstärke“ und „Hexenmeister“ sind nicht weiter interessant, da sie sich schon in der Wörterliste befinden.

### twist3.txt

*Ein Restaurant, welches a la (!)carte abrietet/arbeitet, bietet sein Angebot ohne eine vorher festgelegte Menüreihenfolge/Monierengefühle 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.*

Die Fremdwörter „a“ und „la“ sind zu kurz, um eine Rolle zu spielen, während „carte“ natürlich nicht richtig erkannt wird, weil es sich nicht in der Wörterliste befindet. Dieser Text kann vergleichsweise sehr schnell enttwistet werden (bei mir etwa 0,2 Sekunden), weil zwar 2 Wortverbindungen vorkommen, die zeitaufwändig sind, dafür aber nur ein Wort vorkommt, das nicht richtig erkannt wird. Auch die geringe Länge des Wortes „carte“ spart Zeit, weil es nicht auf passende Wortverbindungen überprüft wird.
Interessant ist noch das Wort „Planungssicherheit“, das einen Fugenlaut enthält, aber trotzdem richtig interpretiert wird. Sobald das Programm als mögliches erstes Wort „Planung“ prüft und deshalb „lanung“ von „laceeghhinnrssui“, dem Mittelteil, abgezogen wird, bleibt „sceehhirsi“ zurück, und weil dazu kein passendes zweites Wort gefunden wird, dieser Teil aber ein „s“ enthält, wird nach „sceehhiri“, also ohne ein „s“ gesucht und „Sicherhei(t)“ gefunden. Nach dem Übernehmen der Groß- und Kleinschreibung des getwisteten Wortes erhält man aus „PlanungsSicherheit“ „Planungssicherheit“.

### twist4.txt

*(!)Autugsa Ada (!)Broyn (!)King, Csuetons/Csutones of (!)Lvolacee, war eine britische Adelige und Mathematikerin, die als die erste Programmiererin überhaupt gilt. Bieters/Bereits/Breites 100 Jahre vor dem Aufkommen der ersten Programmiersprachen ersann sie eine Rechen-Mechanik, der einige Konzepte moderner Programmiersprachen vorwegnahm.*

Hier braucht das Programm etwa doppelt so lang wie bei `twist3.txt`, was den vielen unbekannten Wörtern geschuldet ist, die viel Zeit verbrauchen. Zwar werden nur Wörter, die mehr als 7 Buchstaben haben, für die Suche nach passenden Wortverbindungen genutzt, aber „Augusta“, „Countess“ und „Lovelace“ gehören dazu, was die einzigen (vermeintlichen) Wortverbindungen hier sind.

**twist5.txt**

Hier sind nur die interessantesten Teile des Textes, um Platz zu sparen:

*[…] Sie überlegte sich eben, (so gut es ging, denn sie war schläfrig und dumm von der Hitze,) ob es der Mühe (!)wtreh sei aufzustehen und Gänseblümchen zu pflücken, um eine Kette damit zu machen, als plötzlich ein weißes Kaninchen mit (!)rthoen Augen dicht an ihr vortriebanne/vanbornierte/vorabinterne/vorbeirannte/vornantriebe/vornanbriete/vornantreibe.*

*Dies war garde/grade nicht sehr merkwürdig; Alice fand es auch nicht sehr außerordentlich, daß sie das Kaninchen sagen hörte: „O weh, o weh! Ich werde zu spät kommen!“ […] Aber als das Kaninchen seine Uhr aus der Westentasche zog, nach der Zeit sah und eilig filetorf/feiltorf/fieltorf/filetorf/feiltorf/fieltorf/fortlief, sprang Alice auf. […]*

Und hier noch die wichtigsten Wörter (hauptsächlich Wortverbindungen):

- Dachspitze: *Dpazischte/Dachspitze*
- Breitegrade: *Badregierte/Bardegierte/Bdeartigere/Beitragerde/Beitragrede/Bergdatiere/Bertaderige/Betraderige/Bertaerdige/Brateerdige/Bietergarde/Bietergrade/Bereitgarde/Bereitgrade/Berietgarde/Berietgrade/Breitegarde/Breitegrade/Britedragee/Britegerade/Breitdragee/Breitgerade/Brietdragee/Brietgerade/Bargetreide/Beidergrate/Beiderragte/Beidertrage/Biedergrate/Biederragte/Biedertrage*
- (Neu-)Seeland: *(Neu-)Saelend/Seeland*
- Dinah: *(wird nicht erkannt)*
- Schlüsselchen: *(wird nicht erkannt)*

Dieser Text enthält viele unbekannt Wörter und braucht entsprechend lange. Es gibt viele Wörter, die nicht mehr aktuell geschrieben sind, zusätzliche „h“s enthalten („werth“, „rothen“). Diese Wörter kommen immer mal wieder vor und verlangsamen das Programm. Auch Doppelbedeutungen wie bei „garde/grade“ zeigen, dass sich das Verfahren nicht eignet, um Texte umkehrbar zu verschlüsseln. Wörter wie „Dpazischte“ würde es nicht geben, wenn die Wörterliste auch informationen über die Verwendbarkeit in Wortverbindungen enthielte und Eigennamen wie „Dinah“ oder auch Verniedlichungen wie „Schlüsselchen“ werden sowieso nicht erkannt.
Es gibt bestimmt bessere Texte, um einen solchen Enttwister effizient auszuprobieren.

## Bewertung der Enttwist-Qualität

Die Qualität des Enttwistens ist natürlich beschränkt, aber nichtsdestotrotz werden alle normalen Wörter, die im deutschen Wortschatz (bzw. in der Wörterliste) vorhanden sind, richtig enttwistet bzw. mehrere Möglichkeiten ausgegeben. Einzig Wörter, die aus mehr als zweien zusammengesetzt sind oder nicht in der Wörterliste stehen, werden nicht bzw. nicht richtig erkannt. Auch bei zusammengesetzten Wörtern kann man nicht übermäßig viel erwarten. Dass das richtige Wort dabei sein wird, ist wahrscheinlich, aber trotzdem werden natürlich viele Möglichkeiten gefunden, die es so überhaupt nicht gäbe, weil die Wörterliste ja keine Informationen darüber enthält, in welcher Weise sich ein Wort zum Zusammensetzen eignet und auch Präfixe nur zufällig enthalten sind, so wie „der Weg“ plötzlich als Präfix zu „wegbegeben“ wird.
Ich denke aber, dass die Enttwist-Qualität der eines Menschen durchaus gleicht: Bei besonders langen oder kompliziert zusammengesetzten Wörtern hat auch der Mensch Schwierigkeiten. Was

allerdings nicht geht, ist, Wörter aus dem Zusammenhang zu erschließen. Auch mit falsch geschriebenen Wörtern oder einfach Wörtern, die es so nicht mehr gibt, wie dies z.B. in `twist5.txt` der Fall ist, tut sich das Programm schwer. Aber hier ist es ähnlich wie beim Menschen: Was der Mensch nicht kennt, kann er sich auch nur höchstens erschließen; und ich nehme mal an, dass vielen zu „Mdaotenz“ auch nicht gleich „Modetanz“ einfällt (aber auf keinen Fall „Monatdez“ :-)).

In Bezug auf die Aufgabe ist die Enttwist-Qualität durchaus ausreichend, um zumindest Teile zu verstehen. Wenn einzelne, für das Verständnis wichtige Schlagwörter enttwistet werden können, sollte das schon ausreichen, um einen Text zu verarbeiten. Ich persönlich würde eher nach einer guten Verschlüsselung suchen.

## Quellcode

### Bedeutung der Skripte

```
python3 aufgabe2-beispieldaten.py
```

Twistet und enttwistet die Beispieldaten und gibt die Ergebnisse inklusive der benötigten Zeit aus.

```
python3 aufgabe2-cmd.py
```

Ein Kommandozeilentool, das Texte oder Dateien twistet oder enttwistet. Für mehr Informationen über verfügbare Befehle ist der Befehl `help` zuständig.

```
twister.py
```

Enthält die `Twister`-Klasse.

```
twisting_dict.py
```

Enthält die `TwistingDict`-Klasse.

```
hashable_counter.py
```

Enthält die `HashableCounter`-Klasse.

```
wordoperations.py
```

Enthält Funktionen, um Wörter zu sortieren und zu bearbeiten.

```
create_wordlists.py
```

Übernimmt die Aufbereitung der Wörterliste für ein schnelleres Einlesen (hier nicht weiter beschrieben).

### class Twister

```
WORD_PATTERN = r"[a-zA-ZäöüÄÖÜßß]{4,}"
```

Klassenattribut, das den regulären Ausdruck für Wörter speichert.

```
CACHE_SIZE = 512
```

Klassenattribut mit der maximalen Cache-Größe, die zum Enttwisten verwendet wird.

### `self.dict`

Jede Twister-Klasse besitzt ein Objekt der Klasse `TwistingDict` (siehe dort), das alle benötigten Dictionaries speichert.

### `twist_text(text)`

Twistet einen Text `text` (als `str`), indem jedes Wort, das auf `Twister.WORD_PATTERN` passt, durch den Aufruf von `Twister.twist_match` ersetzt wird.

```
def twist_text(self, text):
    return re.sub(self.WORD_PATTERN, self.twist_match, text)
```

`re.sub` bekommt drei Parameter: Der erste ist der reguläre Ausdruck, mit dem im dritten Parameter nach Vorkommnissen gesucht wird. Die Vorkommnisse werden als `match`-Objekt der Funktion im zweiten Parameter (siehe dort) nacheinander übergeben und diese Funktion muss einen String zurückgeben, der statt des originalen Textes eingefügt wird.

### `twist_match(match)`

Funktioniert wie `Twister.twist_word`, nur dass als Parameter ein `match`-Objekt übergeben wird, das mittels `match.group(0)` in den gefundenen String umgewandelt wird. Die Funktion besitzt bis auf diese Zeile den gleichen Inhalt, aber ruft mit dem String nicht noch eine zusätzliche Funktion auf, um Zeit (vor allem bei größeren Texten) zu sparen.

### `twist_word(word)`

Twistet das als String übergebene `word`. Wörter kleiner als 4 Buchstaben werden ignoriert, da diese vom Twisten nicht betroffen sind. Die Mitte des Wortes wird in eine Liste umgewandelt, um mit `random.shuffle` geshuffelt werden zu können. `random.shuffle` vermischt die Elemente einer Liste zufällig in-place. Um die Liste wieder in einen String zu verwandeln, wird die Funktion `str.join` benutzt, die die Elemente einer Liste mit einem String (in diesem Fall einem leeren String) verbindet.

```
def twist_word(self, word):
    if len(word) < 4:  # kleine Wörter müssen nicht getwistet werden
        return word
    # nur die Mitte des Wortes shuffeln
    middle = list(word[1:-1])
    random.shuffle(middle)
    return word[0] + "".join(middle) + word[-1]
```

### `untwist_text_fast(text)`

Enttwistet den Text `text`. Mit der gleichen Technik wie in `Twister.twist_text` wird für jedes Wort `Twister.untwist_sorted_word` aufgerufen, nur dass hier mit `lambda match: self.untwist_word_fast(match.group(0))` die Funktion direkt mit dem Wort aufgerufen wird, damit der Cache auf der Funktion besser funktioniert (siehe dort).

### `@lru_cache(CACHE_SIZE)`
### `untwist_word_fast(word)`

Enttwistet ein Wort `word` und gibt das Ergebnis als String zurück. Es werden alle Optionen benutzt, um das Wort zu enttwisten. Um mit dem Wort zu suchen, wird es zunächst mit `wordoperations.sort_word` sortiert, um dann mit `Twister.untwist_sorted_word_to_list_fast` eine Liste mit Möglichkeiten oder `None` bzw. eine leere Liste zu erhalten, wenn keine Ergebnisse gefunden wurden (siehe dort). Dementsprechend werden entweder alle Ergebnisse durch „/“ getrennt und mit übernommener Groß- und Kleinschreibung durch `wordoperations.apply_case` zurückgegeben oder das ursprüngliche Wort mit „(!)“ vorangestellt zurückgegeben.

Dass die Funktion gecacht ist, bringt natürlich nichts, wenn die Wörter in unterschiedlicher Groß- bzw. Kleinschreibung vorliegen oder unterschiedlich getwistet sind, aber da sich die `match`-Objekte immer unterscheiden würden, ist es sinnvoller, wenigstens direkt das Wort als Parameter zu nehmen. `lru_cache` stammt aus dem Modul `functools` und verhindert, dass die Funktion bei gleichen Parametern alle Berechnungen von vorne anstellen muss, weil zu den Parametern immer gleich der Rückgabewert gespeichert wird. Dieser kann dann stattdessen zurückgegeben werden, um Zeit zu sparen.

```
@lru_cache(CACHE_SIZE)
def untwist_word_fast(self, word):
    sorted_word = wordoperations.sort_word(word.lower())
    res = self.untwist_sorted_word_to_list_fast(sorted_word)
    if not res:  # Es wurde kein Ergebnis gefunden
        return "(!)" + word
    for i in range(len(res)):
        # Für jedes Ergebnis die Groß-/Kleinschreibung übernehmen
        res[i] = wordoperations.apply_case(word, res[i])
    return "/".join(res)
```

```
@lru_cache(CACHE_SIZE)
untwist_sorted_word_to_list_fast(sorted_word)
```

`sorted_word` ist ein bereits sortiertes, kleingeschriebenes Wort, um den Cache effizienter zu gestalten, da er nun auch bei ähnlichen Wörtern funktioniert (im Gegensatz zu `untwist_word_fast`). Zunächst wird in `self.dict.words_lower` nach passenden Wörtern gesucht, und wenn keine gefunden werden (es gibt also eine `KeyError`-Exception), wird nach zusammengesetzten Wörtern mittels `Twister.untwist_compound_word_fast` gesucht und das Ergebnis zurückgegeben. Das Ergebnis ist entweder `None` bzw. eine leere Liste, wenn keine Wörter gefunden wurden, oder eine Liste aus möglichen Wörtern. Eine leere Liste könnte von `untwist_compound_word_fast` zurückgegeben werden, wenn ein Wort zwar eine ausreichende Länge hat, aber dazu keine passenden Wortverbindungen gefunden werden.

```
@lru_cache(CACHE_SIZE)
def untwist_sorted_word_to_list_fast(self, sorted_word):
    try:
        return self.dict.words_lower[sorted_word]
    except KeyError:
        return self.untwist_compound_word_fast(sorted_word)
```

```
untwist_compound_word_fast(sorted_word)
```

Sucht nach passenden zusammengesetzten Wörtern zu `sorted_word`, einem bereits sortierten, kleingeschriebenem Wort, nach dem unter „Lösungsidee > Enttwisten > Zusammengesetzte Wörter“ beschriebenen Verfahren.

Wenn eine mögliche Kombination von zwei Wörtern gefunden wurde, kann es unter Umständen mehrere Möglichkeiten geben, so wie „gerade“ und „Garde“ sich ähnlich sind. Der Funktion `itertools.product` werden deshalb die zwei Listen übergeben und die Funktion gibt einen Iterator für alle Kombinationen, je ein Wort aus der ersten und ein Wort aus der zweiten Liste, zurück.

```
def untwist_compound_word_fast(self, sorted_word):
  # Wörter unter 7 Buchstaben sind meistens keine Verbindungen und werden
  # nicht geprüft
  if len(sorted_word) < 7:
      return None
  middle = sorted_word[1:-1]  # der Mittelteil des Wortes
```

```
    middle_hash = HashableCounter(middle)  # Zum Rechnen
    # alle Wörter, die aufgrund des ersten Buchstabens in Frage kommen
    first_words = self.dict.words_start[sorted_word[0]]
    # alle Wörter, die aufgrund des letzten Buchstabens in Frage kommen
    second_words = self.dict.words_end[sorted_word[-1]]
    results = []  # alle Ergebnisse

    # alle möglichen ersten Wörter durchgehen und einzeln prüfen
    # first_hash ist ein HashableCounter,
    # first_word_list ist eine Liste aller Wörter, die auf den Hash passen
    for first_hash, first_word_list in first_words.items():
        try:
            # zu dieser Differenz sollte es zweite Wörter geben, wenn dieses
            # erste Wort genommen wird
            possible_second_hash = middle_hash - first_hash
        except ArithmeticError:
            # Funktioniert nicht, also nächstes Wort probieren
            continue
        try:
            # second_word_list sind alle möglichen Wörter für das zweite
            # Wort, die aufgrund des ersten möglichen Wortes in Frage kommen
            second_word_list = second_words[possible_second_hash]
        except KeyError:
            # Es gibt keine passenden zweiten Wörter
            if possible_second_hash["s"] > 0:
                # ABER der restliche Teil für das zweite Wort enthält
                # mindestens ein 's', das ein Fugen-s sein könnte,
                # also das ganze nochmal, aber diesmal mit einm 's' weniger
                possible_second_hash["s"] -= 1
                try:
                    second_word_list = second_words[possible_second_hash]
                except KeyError:
                    # auch hier wird nichts gefunden, also das nächste erste
                    # mögliche Wort probieren
                    continue
                # es gibt Wörter, die mit Fugen-s zusammenpassen
                # alle Kombinationen werden durch itertools.product mit dem
                # Fugen-s angehängt
                results += (first + "s" + second for first, second in
                        itertools.product(first_word_list, second_word_list))
                continue
            else:
                # Es gibt auch kein mögliches Fugen-s im restlichen Teil,
                # also das nächste Wort probieren
                continue
        # es wurden passende Wörter in diesem Durchlauf gefunden
        # alle möglichen Kombinationen werden durch itertools.product
        # angehängt
        results += (first+second for first, second in
                    itertools.product(first_word_list, second_word_list))
    return results
```

### Ergänzende Funktionen

Es gibt noch weitere Funktionen zum Enttwisten, die mehr Optionen bieten. Da dies aber nicht zur Aufgabe gehört, sind sie hier nicht aufgeführt. Die Funktionen sind trotzdem ganz praktisch, wenn ein Text mit individuellen Einstellungen enttwistet werden soll, um ein für den Text angemessenes Ergebnis zu produzieren. Hauptsächlich gibt es die Funktion `Twister.untwist_text`; alle diese

Funktionen arbeiten aber nach demselben Prinzip, nur haben sie noch ein paar zusätzliche `if`-Bedingungen, um bestimmte Optionen ein- bzw. auszuschalten.

### class TwistingDict

Diese Klasse stellt alle Dictionaries als Attribute zur Verfügung, die zum Enttwisten benötigt werden, und mit dem unter „Umsetzung > Enttwisten > Optimierung der Wörterliste“ Verfahren eingelesen werden.

`self.words`
Aufbau: $\{< sortiertes\_wort >: [< passendes\_wort1 >, < passendes\_wort2 >, \dots]\}$

Es sind jeweils die sortierten Worte mit richtiger Groß- und Kleinschreibung allen möglichen enttwisteten Wörtern zugeordnet, die darauf passen. Dieses `dict` wird von den beschriebenen Standardfunktionen nicht verwendet, kann aber benutzt werden, um Groß- und Kleinschreibung zu berücksichtigen.

`self.words_lower`
Aufbau: $\{< sortiertes\_kleines\_wort >: [< passendes\_wort1 >, < passendes\_wort2 >, \dots]\}$

Wie `self.words`, nur dass die sortierten Wörter kleingeschrieben sind.

`self.words_start`
Aufbau: $\{< Anfangsbuchstabe >: \{< HashableCounter\ ohne\ Anfangsbuchstaben >: [< passendes\_wort1 >, \dots]\}, \dots\}$

Ordnet jedem Anfangsbuchstaben ein `dict` zu, welches wiederum `HashableCounter`-Objekten, die ein Wort ohne den jeweiligen Anfangsbuchstaben enthalten, darauf passende Wörter zuordnet.

`self.words_end`
Aufbau: $\{< Endbuchstabe >: \{< HashableCounter\ ohne\ Endbuchstaben >: [< passendes\_wort1 >, \dots]\}, \dots\}$

Ordnet jedem Endbuchstaben ein `dict` zu, welches wiederum `HashableCounter`-Objekten, die ein Wort ohne den jeweiligen Endbuchstaben enthalten, darauf passende Wörter zuordnet.

### class HashableCounter

Die Funktionsweise der `HashableCounter`-Klasse ist bereits unter „Umsetzung > Umsetzung > HashableCounter – Rechnen mit Buchstaben“ beschrieben worden.
Die Klasse implementiert die `__sub__`-Methode, um zwei Objekte subtrahieren zu können. `__sub__` geht alle Elemente des Subtrahenden durch und zieht diese – wenn auch im Minuenden in ausreichender Zahl vorhanden – von diesem ab. Sollte dies nicht möglich sein, wird eine `ArithmeticError`-Exception geworfen.