Projekt zaliczeniowy - dokumentacja

Jakub Dakowski

Wstęp

Prezentowany program jest algorytmem kryptograficznym i steganograficznym umożliwiającym na osadzanie informacji w błędach ortograficznych w dostarczonym do programu tekście w języku polskim. Jest to całkowicie autorski projekt napisany w języku Python. Wybór tego języka jest spowodowany prototypowością tego rozwiązania.

Przykład działania programu

Wewnątrz tekstu “Sklepów cynamonowych” osadzony zostanie tekst:

> test test

Algorytmowi podajemy klucz kryptograficzny `1234`, a wynikiem działania są poniższe akapity:

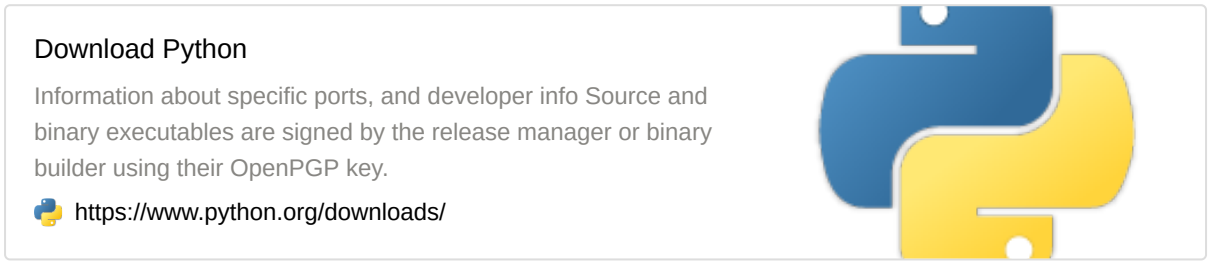
> W okresie najkrutszyh , sęnych dni zimowyh , ujętych z obó stron , od porankó i od wieczora , w futrzane krawendzie zmierzhów , gdy miasto rozgałenziało sę coraz głębiej w labirynty zimowych

> nocy , z tródem pżywoływane przez krutki siwit do opamientania , do powrotu — ojćec mój był już zatracony , zaprzedany , zapżysiężony tamtej sferze .
>
> Twarz jego i głowa zarastały wówczas bujnie i dziko świym włosem , sterczomcym nierególarnie wiechćami , szczecinami , długimi pędzlami , stżelajomcymi z brodawek , z brwi , z dziurek od nosa — co nadawało jego fizjonomii wyglomd starego , nastroszonego lisa .
>
> Węch jego i słuh zaostrzył ść niepąiernie i znać było po grze jego milczomcej i napientej twarzy , rze za posirednictwem tyh zmysłów pozostaje on w ćągłym kontakcie z niewidzialnym światem ciemnych zakamarków , dziur mysich , zmurszałych przestrzeni pustych pod podłogą i kanałów kominowych .

Tekst ten można następnie odszyfrować z pomocą innego skryptu.

Uruchamianie programu

Program wymaga posiadania zainstalowanego Pythona (pisałem go korzystając z wersji 3.10, ale każda młodsza od 3.8.0 także powinna zadziałać).



Zalecane jest także posiadanie biblioteki `tqdm` możliwej do zainstalowania komendą `python -m pip install tqdm`. Zapewnia ona jedynie ładne paski postępu. Aby uruchomić program, wpisujemy będąc w jego folderze w linii komend: `python saver [KLUCZ] [TEKST W KTÓRYM OSADZONE BĘDĄ DANE] [DANE DO OSADZENIA]`. Dostępne jest także więcej opcji:

- `-h` udostępnia pomoc dla użytkownika,

- `-o` umożliwiające podanie pliku, do którego wynik ma być zapisany,
- `--volume`, dzięki któremu program określi najpierw pojemność pliku.

Aby uruchomić skrypt do odszyfrowywania, wpisujemy `python loader [KLUCZ] [PLIK WYNIKOWY] [PLIK WYJŚCIOWY]`. Tutaj dostępna jest tylko opcja `-h`.

> 💡 W folderze `zrodla` zamieściłem zestaw przykładowych plików do testów.

Metoda

Tworząc ten algorytm operowałem się w znacznym stopniu zestawem obserwacji matematycznych dotyczących języka. Te okażą się bardzo istotnymi elementami do zrozumienia wysokopoziomowo sposobu działania algorytmu, dlatego także zostaną dołączone. W tym samym fragmencie opisana zostanie implementacja tych elementów.

Kodowanie błędów w błędach ortograficznych

Rozpocznijmy od określenia, czym dokładnie jest błąd ortograficzny. Tutaj błędem ortograficznym jest podstawienie fragmentów ciągu zgodnie z regułami:

$$
\begin{gathered}
ó \leftrightarrow u \\
ż \leftrightarrow rz \\
ą \leftrightarrow om \\
ś \leftrightarrow si \\
ę \leftrightarrow en \\
ć \leftrightarrow ci \\
h \leftrightarrow ch
\end{gathered}
$$

Można teraz wyobrazić sobie funkcję, która pobiera słowo i ciąg wartości binarnych o długości możliwych podstawień, a zwraca słowo w którym popełniono tylko podstawienia przy określonych okazjach. Dla przykładu:

$$
\begin{gathered}
f(\text{"}brzęczy\text{"}, 00_2) = \text{"}brzęczy\text{"} \\
f(\text{"}brzęczy\text{"}, 10_2) = \text{"}bżęczy\text{"} \\
f(\text{"}brzęczy\text{"}, 01_2) = \text{"}brzenczy\text{"} \\
f(\text{"}brzęczy\text{"}, 11_2) = \text{"}bżenczy\text{"}
\end{gathered}
$$

Rolę tej funkcji spełnia `encode_text(text: str, bytes_: str) -> str`, która jednak jest tak naprawdę “obudówką” dla funkcji `encode(word: str, byt: list[bool]) -> tuple[str, list[bool]]`. Przeprowadza ona konwersje, obsługuje zakończenie programu i zapisywanie długiego tekstu z zachowaniem linii i obsługi interpunkcji. `encode`, poza wyrazem, zwraca także bity, których nie udało się zmieścić w wyrazie. Jej działanie jest całkiem proste. Iteruje ona po znalezionych dopasowaniach do wyrażenia regularnego, aby zakodować informację tam, gdzie jest to możliwe.

Użycie wyrażenia regularnego

Odnajdywanie formy bezbłędnej z użyciem klas abstrakcji

Warto jednak zauważyć, że kodowanie powinno zawsze odbywać się na podstawie wyrazu przenoszącego same zera. Trzeba więc wyznaczać jego wersję bezbłędną. Funkcja `encode_text` używa do tego procedury `find_zero(word) -> tuple[str, bool]`. Opiera ona swoje działanie na klasach abstrakcji. Koncept ten zostanie teraz przedstawiony.

Zdefiniujmy relację dwuargumentową R na zbiorze wszystkich ciągów liter alfabetu polskiego U. Relacja ta zachodzić będzie między dwoma ciągami, jeżeli możliwe jest uzyskanie jednego ciągu z drugiego przez popełnienie serii takich podstawień.

Można z łatwością udowodnić, że będzie to relacja równoważności:

- jest przechodnia z definicji (dwie serie błędów można połączyć tworząc jedną dłuższą),
- jest symetryczna, gdyż możliwe jest odwrócenie każdego błędu przez zastosowanie odwrotnej jego wersji,
- jest zwrotna, gdyż stosując serię podstawień $(x \to x', x' \to x)$ wracamy do oryginalnej formy wyrazu.

Na podstawie tego używać można konceptu klas abstrakcji. Takie klasy abstrakcji zwraca dla otrzymanego wyrazu funkcja `equivalent(word: str) -> list[str]`.

Łatwo zauważyć, że niektóre klasy abstrakcji będą na siebie nachodzić (przykładowo *Bug* oraz *Bóg*). W związku z tym jednym z problemów do rozwiązania dla algorytmu będzie rozstrzyganie takich konfliktów. Podobny problem można zauważyć w sytuacji, gdzie niemożliwe jest wskazanie poprawnego zapisu. Dlatego `find_zero` po wyznaczeniu zawartości danej klasy abstrakcji sprawdza, które z nich występują w zbiorze sjp.pl:

Lista słów z odmianami

https://sjp.pl/slownik/odmiany/

Jeżeli żadna z form nie występuje w zbiorze, procedura jako poprawną formę wyznacza odnaleziony zapis i z pomocą zwracanej wartości logicznej informuje resztę programu, aby nie kodować w tym słowie informacji. Umożliwi to dekoderowi przyjęcie tej samej formy wyrazu za poprawną.

Jeżeli jakieś formy wyrazu wystąpią jednak w zbiorze, program dokonuje losowego wyboru słowa. Jako ziarno losowe ustalany jest na początku działania podany klucz symetryczny.

Program dokonuje tej procedury tylko raz na danym wyrazie i w następnych iteracjach dla wszystkich elementów z klasy abstrakcji zwraca tą samą wartość z pamięci.

Program iteruje więc po kolejnych wyrazach znajdując formę poprawną (kodującą same wartości 0) i przepisuje je w taki sposób, aby kodowały preferowaną informację. Gdy komunikat do przesłania się skończy, bądź miejsce się wyczerpie, program przerywa działanie.

Deszyfrowanie informacji

Procedura deszyfrowania jest znacznie łatwiejsza. Po odpowiednim oczyszczeniu słowa procedura `decode_text(text: str) -> list[bytes]` odnajduje jego formę poprawną z pomocą opisanej już funkcji `find_zero`. Następnie przywołuje `decode(zero, word) -> list[bool]`, aby ta, z pomocą wspomnianego już wyrażenia regularnego, porównała formę poprawną i wykorzystaną i zwróciła ciąg binarny informujący, gdzie popełniono błąd. Tylko miejsca, gdzie **można** popełnić błąd są porównywane.

Podsumowanie

Zdaje się, że program podana implementacja radzi sobie z całkiem obszernym zestawem przykładów. Niestety jednak pojemność większości dostępnych dzieł literackich nie jest wystarczająca, aby przesyłać większe pliki. Oto kilka przykładowych pojemności:

Tekst (pobrany z wolnelektury.pl)	Pojemność (bajty)
Henryk Sienkiewicz *W pustyni i w puszczy*	8234

Tekst (pobrany z wolnelektury.pl)	Pojemność (bajty)
Adam Mickiewicz *Dziady część III*	1546
Bruno Schulz *Sklepy cynamonowe*	297
Antoine de Saint-Exupéry *Mały książę*	969

Warto też zauważyć, że dość łatwo zauważyć nietypowość tych błędów. Aby temu zapobiec możnaby przykładowo zapisywać każdy wyraz tylko w jednej błędnej formie oraz zmniejszyć gęstość błędów. Program nie gwarantuje także w aktualnym stanie zaszyfrowania danych. Jest ono bowiem używane tylko, gdy napotkane słowa występują w słowniku.

Niewykluczone, że pomogłoby także użycie lepszego szyfru oraz wcześniejszej kompresji, lub szyfrowania tekstu. Możliwe jest jednak, że potrzebne byłyby całe sagi do niewidocznego kodowania dłuższych komunikatów. Na szczęście, w aktualnych czasach generowanie długich tekstów nie wydaje się niemożliwe. Szczególnie, że dokładnie to osiąga wciskanie losowych przycisków w słowniku wewnątrz smartfona.

Ciekawym wątkiem jest rozważenie możliwości poszerzenia stosowanego szyfru. Próbowałem dodać do możliwych podstawień także “ź” → “zi” oraz “ń” → “ni”, ale niszczyło to program. Wydaje się, że dodanie ich powoduje kolizję z “ż” → “rz” oraz “ę” → “en”, gdyż wymiana tych elementów szyfru zapewnia działanie programu.

Uzyskany algorytm wydaje się mieć całkiem duży potencjał do rozwoju. Wydaje się jednak, że przez swoją słabą pojemność zostanie on co najwyżej ciekawostką i narzędziem artystycznym, o ile pobierze go ktokolwiek poza autorem i sprawdzającym.