Programowanie Funkcyjne

The Science of Words

Agnieszka Mach
Robert Wojtaszek

07.02.2020

Wstęp

Wiele interesujących technik analizy można zastosować do dużego zbioru słów. Niezależnie od tego, czy będzie to badanie struktury zdania, czy zawartości książki, przepisy te wprowadzą nas w przydatne narzędzia. Podczas manipulowania ciągami do analizy danych niektóre z najczęstszych funkcji należą do wyszukiwania podciągu i edycji obliczeń odległości. Ponieważ liczby często znajdują się w korpusie tekstu rozpoczniemy od pokazania, jak reprezentować liczby w dowolnej bazie jako ciąg znaków. Omówimy kilka algorytmów wyszukiwania ciągów, a następnie skupimy się na wyodrębnianiu tekstu, aby przestudiować nie tylko słowa, ale także sposób ich użycia.

Wyświetlanie liczby w innej bazie

Ciągi to naturalny sposób reprezentowania liczb w różnych bazach, ze względu na włączenie liter jako cyfr. Zaraz dowiemy się jak przekonwertować liczbę na ciąg znaków, który można pokazać, jako wynik.

Wyświetlanie liczby w innej bazie

1. Będziemy musieli zaimportować następujące dwie funkcje:

```
import Data.Char (intToDigit, chr, ord)
import Numeric (showIntAtBase)
```

Wyświetlanie liczby w innej bazie

2.Zdefiniujemy funkcję do reprezentowania liczby w określonej bazie w następujący sposób:

```
n `inBase` b = showIntAtBase b numToLetter n ""
```

Wyświetlanie liczby w innej bazie

3.Zdefiniujemy odwzorowanie liczb i liter dla cyfr większych niż dziewięć w następujący sposób:

```
numToLetter :: Int -> Char
numToLetter n
    | n < 10 = intToDigit n
    | otherwise = chr (ord 'a' n - 10)
```

Wyświetlanie liczby w innej bazie

4.Pokazujemy wynik używając następującego fragmentu kodu:

```
main :: IO ()
main = do
  putStrLn $ 8 'inBase' 12
  putStrLn $ 10 'inBase' 12
  putStrLn $ 12 'inBase' 12
  putStrLn $ 47 'inBase' 12
```

Wyświetlanie liczby w innej bazie

5.Poniżej przedstawiamy uzyskany dane podczas uruchamiania kodu:

```
§ runhaskell Main.hs


8

a

10

3b
```

Wyświetlanie liczby w innej bazie

Funkcja showIntAtBase przyjmuje bazę, żądaną liczbę i jej odwzorowanie z liczby na cyfrę do wydrukowania. Nasze cyfry przedstawiamy w następujący sposób: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f itd., Do 36 znaków. Podsumowując, otrzymujemy wygodny sposób reprezentowania liczby dziesiętnej w dowolnej bazie.

Odczytywanie numeru z innej bazy

System dziesiętny, binarny i szesnastkowy to powszechnie stosowane systemy liczbowe, które często są reprezentowane za pomocą łańcucha. Ten sposób pokaże, jak przekonwertować ciąg reprezentujący liczbę w dowolnej bazie na liczbę całkowitą dziesiętną. Używamy funkcji readInt, która jest podwójną funkcją showIntAtBase opisaną w poprzednim przepisie.

Odczytywanie numeru z innej bazy

1.Importujemy readInt i następujące funkcje do manipulowania znakami w następujący sposób:

```
import Data.Char (ord, digitToInt, isDigit)
import Numeric (readInt)
```

Odczytywanie numeru z innej bazy

2.Definiujemy funkcję do konwersji ciągu reprezentującego liczbę w określonej bazie na dziesiętną liczbę całkowitą w następujący sposób:

```
str 'base' b = readInt b isValidDigit letterToNum str
```

Odczytywanie numeru z innej bazy

3.Definiujemy odwzorowanie liter i cyfr dla większych cyfr, jak widać w poniższym fragmencie kodu:

```
letterToNum :: Char -> Int
letterToNum d
  | isDigit d = digitToInt d
  | otherwise = ord d - ord 'a' + 10

isValidDigit :: Char -> Int
isValidDigit d = letterToNum d >= 0
```

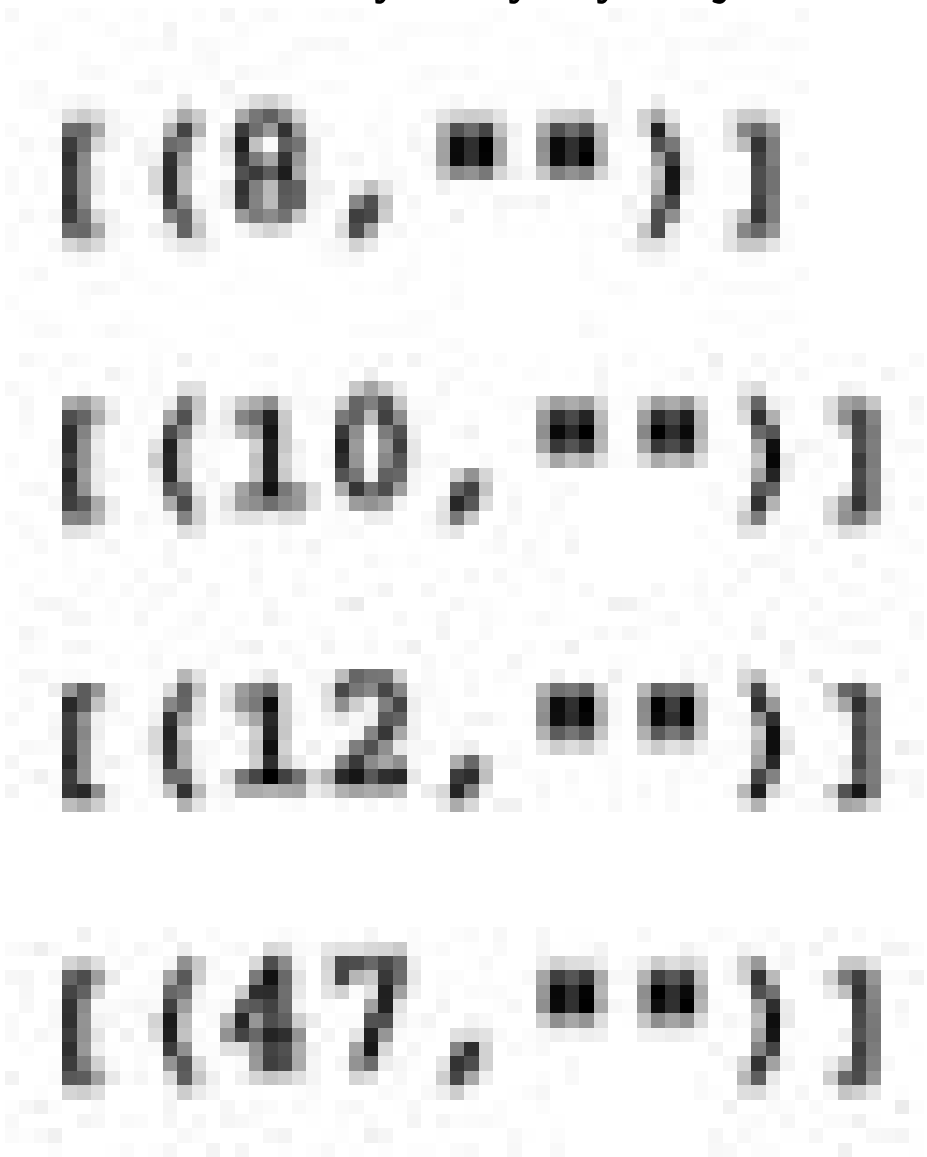
Odczytywanie numeru z innej bazy

4.Pokazujemy wynik, używając następującego wiersza kodów:

```
main :: IO ()
main = do
  print $ "8" 'base' 12
  print $ "a" 'base' 12
  print $ "10" 'base' 12
  print $ "3b" 'base' 12
```

Odczytywanie numeru z innej bazy

5.Otrzymany wynik jest następujący:



Odczytywanie numeru z innej bazy

Funkcja readInt odczytuje niepodpisaną wartość całkową i konwertuje ją na określoną bazę.

Wyszukiwanie podciągów za pomocą Data.ByteString

Istnieje wiele algorytmów wyszukiwania ciągu w innym ciągu. Sposób, który zaraz przedstawimy użyje istniejącej funkcji breakSubstring w bibliotece Data.ByteString.

Wyszukiwanie podciągów za pomocą Data.ByteString

1.Importujemy funkcję breakSubstring oraz pakiet Data.ByteString.Char8 w następujący sposób:

```
import Data.ByteString (breakSubstring)
import qualified Data.ByteString.Char8 as C
```

Wyszukiwanie podciągów za pomocą Data.ByteString

2.Pakujemy ciągi jako ByteString i wprowadamy je do breakSubstring, który ma następujący typ:
ByteString -> ByteString -> (ByteString, ByteString)

Wyszukiwanie podciągów za pomocą Data.ByteString

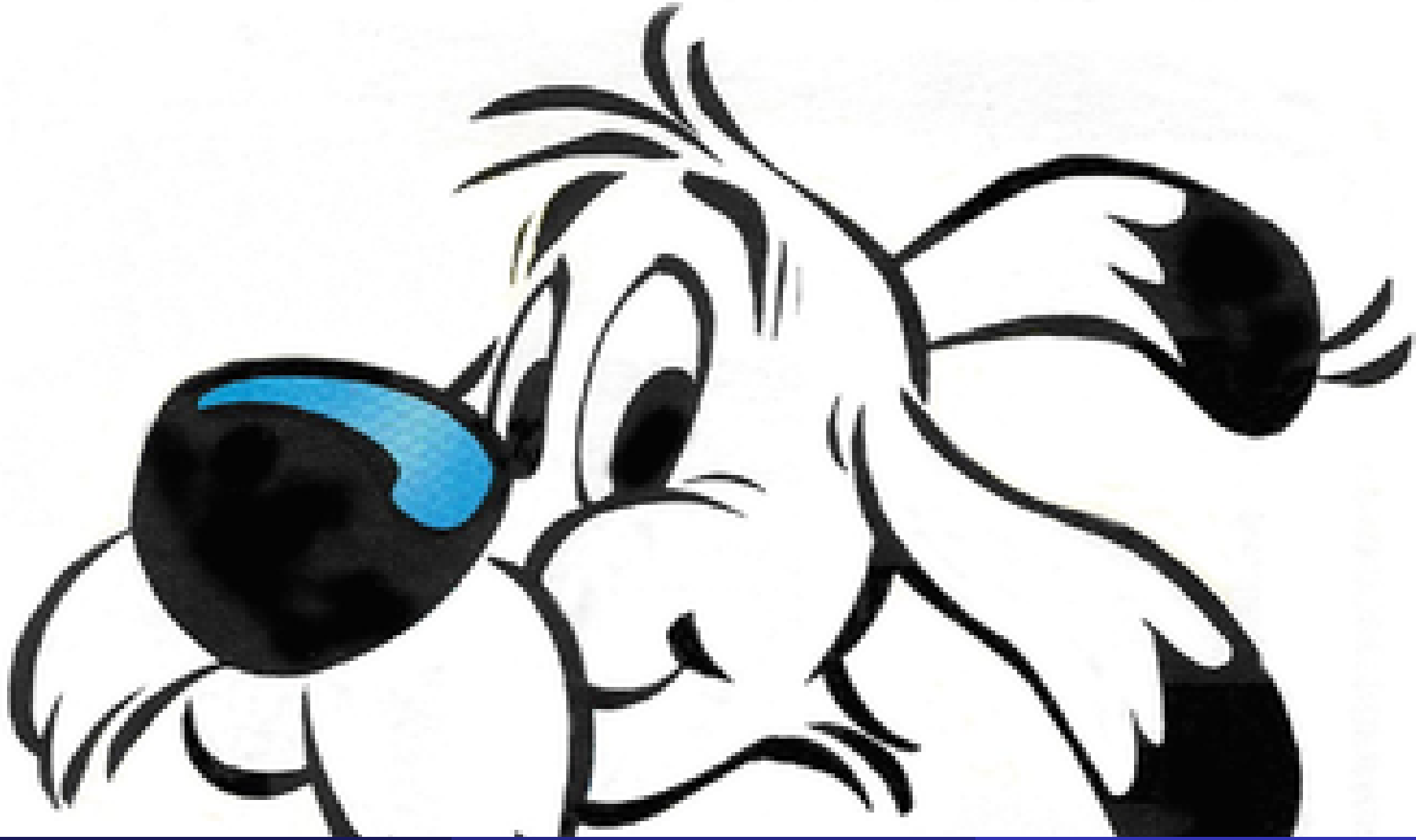
Następnie określamy czy ciąg został znaleziony:

```
substringFound :: String -> String -> Bool

substringFound query str =
  (not . C.null . snd) $
  breakSubstring (C.pack query) (C.pack str)
```

Wyszukiwanie podciągów za pomocą Data.ByteString

Funkcja breakSubstring sprawdza rekurencyjnie, czy wzorzec jest prefiksem ciągu.



Wyszukiwanie ciągu przy użyciu algorytmu Boyer – Moore – Horspool

Algorytm wyszukiwania łańcucha Horspool zaimplementowany w tym sposobie działa dobrze dla prawie wszystkich długości wzorów i rozmiarów alfabetu. Dzięki wstępnemu przetwarzaniu zapytania algorytm może skutecznie pominąć zbędne porównania. Za moment pokażemy sposób implementacji uproszczonej wersji zwany algorytmem Horspoola, który osiąga ten sam średni najlepszy przypadek co algorytm Boyer-Moore. Algorytmy Boyera-Moore'a powinny być stosowane tylko wtedy, gdy dodatkowy wymagany czas i przestrzeń do przygotowania są dopuszczalne.

Wyszukiwanie ciągu przy użyciu algorytmu Boyer – Moore – Horspool

1.Będziemy używać kilku funkcji Data.Map w następujący sposób:

```
import Data.Map (fromList, (!), findWithDefault)
```

Wyszukiwanie ciągu przy użyciu algorytmu Boyer – Moore – Horspool

2.Dla wygody zdefiniujemy krótki reprezentujący indeks znaków:

```
indexMap xs = fromList $ zip [0..] xs

revIndexMap xs = fromList $ zip (reverse xs) [0..]
```

Wyszukiwanie ciągu przy użyciu algorytmu Boyer – Moore – Horspool

3.Zdefiniujemy algorytm wyszukiwania, aby używał funkcji rekurencyjnej bmh w następujący sposób:

```
bmh :: Ord a => [a] -> [a] -> Maybe Int

bmh pat xs = bmh' (length pat - 1) (reverse pat) xs pat
```

Wyszukiwanie ciągu przy użyciu algorytmu Boyer – Moore – Horspool

4.Rekurencyjnie znajdujemy wzorzec w bieżącym indeksie, dopóki indeks nie przekroczy długości łańcucha, jak zauważyć można w poniższym fragmencie kodu:

```
bmh' :: Ord a => Int -> [a] -> [a] -> [a] -> Maybe Int

bmh' n [] xs pat = Just (n + 1)
bmh' n (p:ps) xs pat
  | n >= length xs    = Nothing
  | p == (indexMap xs) ! n = bmh' (n - 1) ps xs pat
  | otherwise              = bmh' (n + findWithDefault
```

Wyszukiwanie ciągu przy użyciu algorytmu Boyer – Moore – Horspool

```
                                        (length pat) (sMap ! n) pMap)
                                        (reverse pat) xs pat
where sMap = indexMap xs
      pMap = revIndexMap pat
```

Wyszukiwanie ciągu przy użyciu algorytmu Boyer – Moore – Horspool

5. Testujemy funkcję:

```
main :: IO ()
main = print $ bmh "Wor" "Hello World"
```

Wyszukiwanie ciągu przy użyciu algorytmu Boyer – Moore – Horspool

6.Poniższe dane wyświetlają pierwszy indeks:
Just 6

Wyszukiwanie ciągu przy użyciu algorytmu Boyer – Moore – Horspool

Ten algorytm porównuje pożądany wzór z ruchomym oknem w tekście. Wydajność wynika z tego, jak szybko ruchome okno przesuwa się od lewej do prawej przez ten tekst. W algorytmie Horspool zapytanie jest porównywane z bieżącym oknem znak po znaku od prawej do lewej, a okno przesuwa się o rozmiar zapytania w najlepszym przypadku.

Wyszukiwanie ciągu przy użyciu algorytmu Rabin-Karp

Algorytm Rabina-Karpa znajduje wzór w treści tekstu dopasowując unikalną reprezentację wzoru do ruchomego okna. Unikalna reprezentacja lub skrót jest obliczana przez uwzględnienie ciągu jako liczby zapisanej w dowolnej bazie 26 lub większej.

Wyszukiwanie ciągu przy użyciu algorytmu Rabin-Karp

1.Używamy rozszerzenia języka OverloadedStrings, aby ułatwić manipulacje ByteString w naszym kodzie w następujący sposób. Zasadniczo umożliwia to zachowanie polimorficzne dla łańcuchów, dzięki czemu kompilator GHC może wnioskować o nim jako o typie ByteString, gdy jest to konieczne:
- LANGUAGE OverloadedStrings -

Wyszukiwanie ciągu przy użyciu algorytmu Rabin-Karp

2.Importujemy algorytmy Rabina-Karpa:

```
import Data.ByteString.Search.KarpRabin (indicesOfAny)
import qualified Data.ByteString as BS
```

Wyszukiwanie ciągu przy użyciu algorytmu Rabin-Karp

3.Zdefiniujemy kilka wzorców, aby znaleźć i uzyskać podstawę z pliku big.txt:

```
main = do
  let needles = [ "preparing to go away"
                , "is some letter of recommendation"]
  haystack <- BS.readFile "big.txt"
```

Wyszukiwanie ciągu przy użyciu algorytmu Rabin-Karp

4.Uruchamiamy algorytm Rabina-Karpa dla wszystkich wzorców wyszukiwania:

```
print $ indicesOfAny needles haystack
```

Wyszukiwanie ciągu przy użyciu algorytmu Rabin-Karp

5.Kod wyświetla wszystkie indeksy znalezione dla każdej igły jako listę krotek:

```
$ runhaskell Main.hs

[(3738968,[1]),(5632846,[0]),(5714386,[0])]
```

Wyszukiwanie ciągu przy użyciu algorytmu Rabin-Karp

W algorytmie Rabina-Karpa stałe okno przesuwa się od lewej do prawej, porównując unikalne wartości skrótu w celu uzyskania skutecznych porównań. Funkcja skrótu konwertuje ciąg znaków na jego reprezentację numeryczną.

Wyszukiwanie ciągu przy użyciu algorytmu Rabin-Karp

Oto przykład konwersji łańcucha na bazę b równą 256:
„hello” = h '* b4 + e' * b3 + l '* b2 + l' * b1 + o '* b0
(co daje wynik 448378203247), gdzie każda litera h '= ord h (co daje 104)
i tak dalej, i tak dalej.

Dzielenie łańcucha na linie, słowa lub dowolne tokeny

Przydatne dane są często rozproszone między ogranicznikami, takimi jak przecinki lub spacje, co sprawia, że podział ciągów jest niezbędny w przypadku większości zadań.

Dzielenie łańcucha na linie, słowa lub dowolne tokeny

1.Jedyną potrzebną funkcją jest splitOn, który importujemy w następujący sposób:
import Data.List.Split (splitOn)

Dzielenie łańcucha na linie, słowa lub dowolne tokeny

2.Najpierw podzielimy ciąg na linie:

```
main = do
  input <- readFile "input.txt"
  let ls = lines input
  print $ ls
```

Dzielenie łańcucha na linie, słowa lub dowolne tokeny

3.Linie zostają wyświetlane na liście:

```
[ "first line","second line"
, "words are split by space"
, "comma,separated,values"
, "or any delimiter you want"]
```

Dzielenie łańcucha na linie, słowa lub dowolne tokeny

4.Następnie rozdzielamy ciąg znaków na spacje:

```
let ws = words $ ls !! 2
print ws
```

Dzielenie łańcucha na linie, słowa lub dowolne tokeny

5.Słowa są wyświetlane w liście:
["words",ąre",śplit","by",śpace"]

Dzielenie łańcucha na linie, słowa lub dowolne tokeny

6.Następnie pokażemy, jak podzielić ciąg na dowolną wartość za pomocą następujących wierszy kodu:

```
let cs = splitOn "," $ ls !! 3
print cs
```

Dzielenie łańcucha na linie, słowa lub dowolne tokeny

7.Wartości są rozdzielane przecinkami w następujący sposób:
[ćomma",śeparated","values"]

Dzielenie łańcucha na linie, słowa lub dowolne tokeny

8.Na koniec pokazujemy podział na wiele liter:

```
let ds = splitOn "an" $ ls !! 4
print ds
```

Dzielenie łańcucha na linie, słowa lub dowolne tokeny

9.Dane wyjściowe są następujące:
[ór any d",łimit",ż you want"]

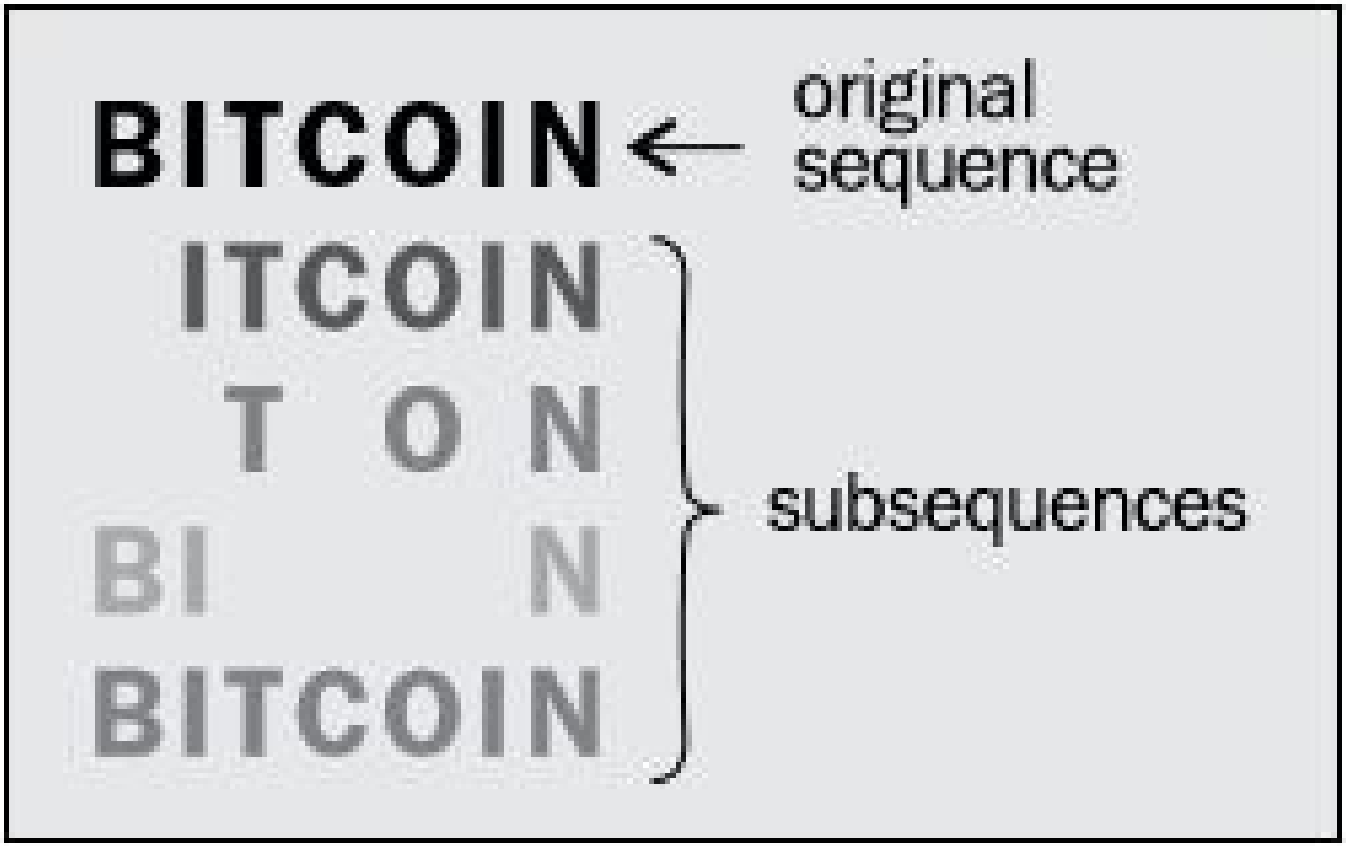
Znajdowanie najdłuższej wspólnej podsekwencji

Jednym ze sposobów porównania podobieństwa ciągów jest znalezienie ich najdłuższej wspólnej podsekwencji. Jest to przydatne w znajdywaniu różnic pomiędzy mutacjami danych, takich jak kod źródłowy lub sekwencje genomu.

Znajdowanie najdłuższej wspólnej podsekwencji

Podsekwencja ciągu jest tym samym ciągiem, z którego usunięto zero lub więcej indeksów.

Znajdowanie najdłuższej wspólnej podsekwencji



Znajdowanie najdłuższej wspólnej podsekwencji

1. Jedyny import, którego potrzebujemy, to ten przydatny pakiet, aby łatwo obsługiwać zapamiętywanie:
import qualified Data.MemoCombinators as Memo

Znajdowanie najdłuższej wspólnej podsekwencji

2.W celu zapamiętywania funkcji tworzymy funkcję, które pobierają dwa argumenty łańcuchowe:

```
memoize :: (String -> String -> r) -> String -> String -> r
memoize = Memo.memo2
    (Memo.list Memo.char) (Memo.list Memo.char)
```

Znajdowanie najdłuższej wspólnej podsekwencji

3. Definiujemy największą wspólną funkcję podsekwencji:

```
lcs :: String -> String -> String

lcs = memoize lcs'
  where lcs' xs'@(x:xs) ys'@(y:ys)
    | x == y = x : lcs xs ys
    | otherwise = longer (lcs xs' ys) (lcs xs ys')
    lcs' _ _ = []
```

Znajdowanie najdłuższej wspólnej podsekwencji

4.Wewnętrznie definiujemy funkcję, która zwraca łańcuch o większej długości:

```
longer as bs
  | length as > length bs = as
  | otherwise = bs
```

Znajdowanie najdłuższej wspólnej podsekwencji

5.Uruchamiamy funkcję na dwóch ciągach:

```
main :: IO ()
main = do
  let xs = "find the lights"
  let ys = "there are four lights"
  print $ lcs xs ys
```

Znajdowanie najdłuższej wspólnej podsekwencji

6.Otrzymujemy najdłuższy wspólny podciąg pomiędzy dwoma łańcuchami:
"the lights"

Obliczanie kodu fonetycznego

Jeśli mamy do czynienia z nurtem angielskich słów, możemy podzielić je na kody fonetyczne, aby zobaczyć, jak brzmią podobnie. Kody fonetyczne działają na dowolne ciągi alfabetyczne, a nie tylko na słowa.

Obliczanie kodu fonetycznego

1.Importujemy funkcję kodu fonetycznego:

```
import Text.PhoneticCode.Soundex (soundexNARA,
  soundexSimple)
import Text.PhoneticCode.Phonix (phonix)
```

Obliczanie kodu fonetycznego

2.Definiujemy listę podobnie brzmiących słów:
ws = ["haskell", "hackle", "haggle", "hassle"]

Obliczanie kodu fonetycznego

3. Testujemy kody fonetyczne tych słów:

```
main :: IO ()
main = do
  print $ map soundexNARA ws
  print $ map soundexSimple ws
  print $ map phonix ws
```

Obliczanie kodu fonetycznego

4.Nasze dane zostają wyświetlone w następujący sposób:

```
$ runhaskell Main.hs

["H240","H240","H240","H240"]

["H240","H240","H240","H240"]

["H82","H2","H2","H8"]
```

Obliczanie odległości edycji między dwoma ciągami

Odległością edycji lub odległością Levenshteina nazywamy minimalną liczbę prostych operacji na łańcuchach wymaganych do konwersji jednego łańcucha na inny. Za moment obliczymy odległość edycji na podstawie tylko wstawek, usunięć i podstawień znaków.

Obliczanie odległości edycji między dwoma ciągami

1.Jedyny import, jakiego potrzebujemy, to możliwość łatwego zapamiętywania funkcji za pomocą następującego wiersza kodu:
import qualified Data.MemoCombinators as Memo

Obliczanie odległości edycji między dwoma ciągami

2.Definiujemy funkcję odległości Levenshteina:

```
lev :: Eq a => [a] -> [a] -> Int
lev a b = levM (length a) (length b)
  where levM = memoize lev'
    lev' 1 j
      | min i j == 0 = max i j
      | otherwise    = minimum
```

Obliczanie odległości edycji między dwoma ciągami

```
[ ( 1 + levM (i-1) j )
, ( 1 + levM i (j-1) )
, ( ind i j + levM (i-1) (j-1) ) ]
```

Obliczanie odległości edycji między dwoma ciągami

3.Definiujemy funkcję wskaźnika, która zwraca 1, jeśli znaki nie pasują:

```
ind i j
  | a !! (i-1) == b !! (j-1) = 0
  | otherwise = 1
```

Obliczanie odległości edycji między dwoma ciągami

4. Tworzymy funkcję w celu umożliwienia zapamiętywanie funkcji, które przyjmują dwa argumenty łańcuchowe:
memoize = Memo.memo2 (Memo.integral) (Memo.integral)

Obliczanie odległości edycji między dwoma ciągami

5.Wyświetlamy odległość edycji między dwoma ciągami:

```
main = print $ lev "mercury" "sylvester"
```

Obliczanie odległości edycji między dwoma ciągami

6.W następstwie dostajemy co następuje:

```
$ runhaskell Main.hs

8
```

Obliczanie odległości Jaro – Winklera między dwoma stringami

Odległość Jaro-Winklera mierzy podobieństwo ciągu reprezentowane, jako liczba rzeczywista między 0 a 1. Wartość 0 nie odpowiada podobieństwu, a 1 odpowiada identycznemu dopasowaniu.

Obliczanie odległości Jaro – Winklera między dwoma strunami

1.Będziemy potrzebować dostępu do funkcji elemIndices, która jest importowana w następujący sposób:
import Data.List (elemIndices)

Obliczanie odległości Jaro – Winklera między dwoma strunami

2.Definiujemy funkcję Jaro-Winklera w oparciu o następującą formułę:

```
jaro :: Eq a => [a] -> [a] -> Double

jaro s1 s2
  | m == 0    = 0.0
  | otherwise = (1/3) * (m/ls1 + m/ls2 + (m-t)/m)
```

Obliczanie odległości Jaro – Winklera między dwoma strunami

3.Definiujemy używane zmienne:

```
where ls1 = toDouble $ length s1

      ls2 = toDouble $ length s2

       m' = matching s1 s2 d

       d = fromIntegral $
       max (length s1) (length s2) 'div' 2 - 1

       m = toDouble m'

       t = toDouble $ (m' - matching s1 s2 0) 'div' 2
```

Obliczanie odległości Jaro – Winklera między dwoma strunami

4.Definiujemy funkcję pomocniczą w celu przekonwertowania liczby całkowitej na typ Double:

```
toDouble :: Integral a => a -> Double

toDouble n = (fromIntegral n) :: Double
```

Obliczanie odległości Jaro – Winklera między dwoma strunami

5.Definiujemy funkcję pomocniczą, aby znaleźć liczbę pasujących znaków w określonej odległości:

```
matching :: Eq a => [a] -> [a] -> Int -> Int

matching s1 s2 d = length $ filter
  (\(c,i) -> not (null (matches s2 c i d)))
  (zip s1 [0..])
```

Obliczanie odległości Jaro – Winklera między dwoma strunami

6. Definiujemy funkcję pomocniczą, aby znaleźć ilość pasujących znaków z określonego znaku o określonym indeksie:

```
matches :: Eq a => [a] -> a -> Int -> Int -> [Int]

matches str c i d = filter (<= d) $
  map (dist i) (elemIndices c str)
  where dist a b = abs $ a - b
```

Obliczanie odległości Jaro – Winklera między dwoma strunami

7. Testujemy algorytm wyświetlając kilka przykładów:

```
main = do
  print $ jaro "marisa" "magical"
  print $ jaro "haskell" "hackage"
```

Obliczanie odległości Jaro – Winklera między dwoma strunami

8. Podobieństwa są wyświetlane, jako takie, które sugerują, że „marisa” jest bliższa „magicznemu” niźli „haskell” do „hackowaniu”.

```
$ runhaskell Main.hs

0.7460317460317460

0.7142857142857142
```

Naprawianie błędów ortograficznych za pomocą odległości edycji

Sposób ten pokaże, jak znaleźć ciągi, które są w odległości jednej edycji od określonego ciągu. Funkcji tej możemy użyć do poprawienia pisowni.

Naprawianie błędów ortograficznych za pomocą odległości edycji

1.Importujemy kilka funkcji znakowych i listowych:

```
import Data.Char (toLower)
import Data.List (group, sort)
```

Naprawianie błędów ortograficznych za pomocą odległości edycji

2.Definiujemy funkcję zwracającą ciągi, które znajdują się w odległości jednej edycji:

```
edits1 :: String -> [String]

edits1 word = unique $
              deletes ++ transposes ++ replaces ++ inserts
  where splits   = [ (take i word', drop i word') |
    i <- [0..length word']]
```

Naprawianie błędów ortograficznych za pomocą odległości edycji

3. Tworzymy listę ciągów z usuniętym jednym znakiem:

```
deletes = [ a ++ (tail b) |
  (a,b) <- splits, (not.null) b]
```

Naprawianie błędów ortograficznych za pomocą odległości edycji

4. Tworzymy listę ciągów z zamienionymi dwoma znakami:

```
transposes = [a ++ [b!!1] ++ [head b] ++ (drop 2 b) |
  (a,b) <- splits, length b > 1 ]
```

Naprawianie błędów ortograficznych za pomocą odległości edycji

5. Tworzymy listę ciągów, w których jeden ze znaków zostanie zastąpiony inną literą w alfabecie:

```
replaces   = [ a ++ [c] ++ (drop 1 b)
   | (a,b) <- splits
   , c <- alphabet
   , (not.null) b ]
```

Naprawianie błędów ortograficznych za pomocą odległości edycji

6. Tworzymy listę ciągów znaków z jednym znakiem wstawionym w dowolnym miejscu:

```
inserts = [a ++ [c] ++ b
| (a,b) <- splits
, c <- alphabet ]
```

Naprawianie błędów ortograficznych za pomocą odległości edycji

7.Definiujemy alfabet i funkcję pomocnika, aby przekonwertować ciąg na małe litery:

```
alphabet = ['a'..'z']
word' = map toLower word
```

Naprawianie błędów ortograficznych za pomocą odległości edycji

8.Definiujemy funkcję pomocniczą, aby uzyskać unikalne elementy z listy:

```
unique :: [String] -> [String]
unique = map head.group.sort
```

Naprawianie błędów ortograficznych za pomocą odległości edycji

9.Wyświetlamy wszystkie możliwe ciągi, które są w odległości jednej edycji od następującego ciągu:

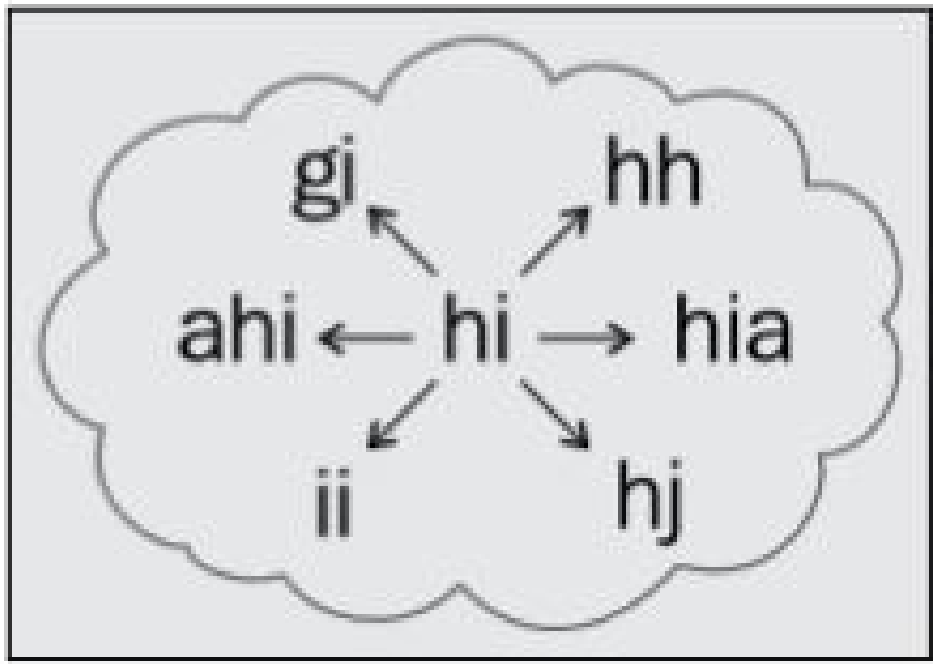
```
main = print $ editsi "hi"
```

Naprawianie błędów ortograficznych za pomocą odległości edycji

```
["ahi","ai","bhi","bi","chi","ci","dhi","di","ehi","ei","fhi","fi","ghi",
"gi","h","ha","hai","hb","hbi","hc","hci","hd","hdi","he","hei","hf","hfi
","hg","hgi","hh","hhi","hi","hia","hib","hic","hid","hie","hif","hig","h
ih","hii","hij","hik","hil","him","hin","hio","hip","hiq","hir","his","hi
t","hiu","hiv","hiw","hix","hiy","hiz","hj","hji","hk","hki","hl","hli","
hm","hmi","hn","hni","ho","hoi","hp","hpi","hq","hqi","hr","hri","hs","hs
i","ht","hti","hu","hui","hv","hvi","hw","hwi","hx","hxi","hy","hyi","hz"
,"hzi","i","ih","ihi","ii","jhi","ji","khi","ki","lhi","li","mhi","mi","n
hi","ni","ohi","oi","phi","pi","qhi","qi","rhi","ri","shi","si","thi","ti
","uhi","ui","vhi","vi","whi","wi","xhi","xi","yhi","yi","zhi","zi"]
```

Naprawianie błędów ortograficznych za pomocą odległości edycji

Bardzo intuicyjnie stworzyliśmy sąsiedztwo słów, które różnią się tylko 1 wstawieniem, usunięciem lub zastąpieniem bądź transpozycją.



Naprawianie błędów ortograficznych

Podczas gromadzenia danych dostarczonych przez ludzi mogą wkraść się błędy ortograficzne. Sposób ten poprawi błędnie napisane słowo za pomocą prostego narzędzia sprawdzania pisowni.

Naprawianie błędów ortograficznych

1.Importujemy funkcje:

```
import Data.Char (isAlpha, isSpace, toLower)
import Data.List (group, sort, maximumBy)
import Data.Ord (comparing)
import Data.Map (fromListWith, Map, member, (!))
```

Naprawianie błędów ortograficznych

2.Definiujemy funkcję, która automatycznie poprawi pisownię każdego słowa w zdaniu:

```
autofix :: Map String Int -> String -> String

autofix m sentence = unwords $
                     map (correct m) (words sentence)
```

Naprawianie błędów ortograficznych

3.Pobieramy słowa z całego tekstu:

```
getWords :: String -> [String]

getWords str = words $
               filter (\x -> isAlpha x || isSpace x) lower

  where lower = map toLower str
```

Naprawianie błędów ortograficznych

4.Obliczamy mapę częstotliwości podanych występowania słów:

```
train :: [String] -> Map String Int

train = fromListWith (+) . ('zip' repeat 1)
```

Naprawianie błędów ortograficznych

5.Znajdujemy ciągi w odległości jednej edycji:

```
edits 1 :: String -> [String]

edits1 word = unique $
              deletes ++ transposes ++ replaces ++ inserts

  where splits = [ (take i word', drop i word')
                 | i <- [0..length word']]

  deletes = [ a ++ (tail b)
                 | (a,b) <- splits
                 , (not.null) b ]

  transposes = [ a ++ [b !! 1] ++ [head b] ++ (drop 2 b)
```

Naprawianie błędów ortograficznych

```
                | (a,b) <- splits, length b > 1 ]

replaces = [ a ++ [c] ++ (drop 1 b)
                | (a,b) <- splits, c <- alphabet
                , (not.null) b ]

inserts = [a ++ [c] ++ b |
                  (a,b) <- splits, c <- alphabet ]

alphabet = ['a'..'z']

word' = map toLower word
```

Naprawianie błędów ortograficznych

6.Znajdźmy słowa, które dzieli od siebie odległość edytowania dwóch:

```
knownEdits2 :: String -> Map String a -> [String]

knownEdits2 word m = unique $ [ e2
                              | e1 <- edits1 word
                              , e2 <- edits1 e1
                              , e2 `member` m]
```

Naprawianie błędów ortograficznych

7.Definiujemy funkcję pomocniczą, aby uzyskać unikalne elementy z listy:

```
unique :: [String] -> [String]

unique = map head.group.sort
```

Naprawianie błędów ortograficznych

8.Znajdź znane słowa z listy:

```
known :: [String] -> Map String a -> [String]

known ws m = filter ('member' m) ws
```

Naprawianie błędów ortograficznych

9.Poprawiamy błąd w pisowni poprzez zwracanie najczęstszego kandydata:

```
correct :: Map String Int -> String -> String

correct m word = maximumBy (comparing (m!)) candidates
  where candidates = head $ filter (not.null)
                          [ known [word] m
                          , known (edits1 word) m
                          , knownEdits2 word m
                          , [word] ]
```

Koniec

Prezentację przygotowali:
Agnieszka Mach
Robert Wojtaszek