Algorithmen und Datenstrukturen SoSe25

-Assignment 9-

Moritz Ruge
Matrikelnummer: 5600961
Lennard Wittenberg
Matrikelnummer: —

Juni 2025

1 Problem: Suchen in Zeichenketten I

Implementieren Sie den naiven Algorithmus und den Algorithmus von Rabin-Karp zur Suche in Zeichenketten. Finden Sie dann heraus, wie oft das Wort whale in Moby Dick vorkommt (ignorieren Sie dabei Groß- und Kleinschreibung). Wie schneiden Ihre Implementierungen im Vergleich ab? Hinweis: Den Roman Moby Dick finden Sie unter
http://www.gutenberg.org/files/2701/2701-0.txt.

1.1 Implementierung:

```
# task 1 isn't clear on what is expected. It states to return the count of the word "whale" but speficies substring search algorithmns to do so.
test = "The whale is a magnificent creature. Whales live in the ocean. A blue whale can grow very large. The swordwhale is not a real animal. Whalers used to hunt whales. The whalebone was valuable. Whalewhalewhale is not a word, but whale-watching is a popular activity. Whale appears in this sentence. Is whale capitalized here?"
def get_book_text(path): # helper function to access moby_dick.txt
with open(path) as f:
return f.read()

def naive_search_with_substrings(s,t): # This version of the naive search is not fit for searching an entire book due to the O(n^2) time-complexity.
text = s.lower() # convert the given string/text to lower-case
words = text.split() # split the text into list of words
l = len(t)
moby_counter = 0
for word in words:
k = len(word)
for i in range(k-l+1):  # Try each starting position
j = 0
while j < l and word[i+j] == t[j]: # start beginning of t -> j can be greater the len(t) check all positions of the given suffix from s if it contains t.
j += 1
if j == l:    # if j is grearter than len(t) than t is in s first at index i.
moby_counter += 1  # Pattern found at position i
return moby_counter

def naive_search_words_only(s,t):
text = s.lower() # convert the given string/text to lower-case
words = text.split() # split the text into list of words
count_moby = 0
for word in words:
if word == t:
count_moby += 1
return count_moby


def get_lower_case_alpha_list(): # helper function to get a list of the latin alphabet
# initialise an empty list
list = []
# filling the list with lowercase letter in alphabetical order
alpha = 'a'
for i in range(0, 26):
list.append(alpha)
alpha = chr(ord(alpha) + 1)
return list

```

```
42  def rabin_karp(s, t):
43  l = len(t)
44  alphabet = get_lower_case_alpha_list() # assigning values a:0 ... z = 25
45
46  # Base for the rolling hash
47  base = len(alphabet)  # Size of the alphabet
48  # Large prime to reduce collisions
49  prime = 101
50  text = s.lower() # convert the given string/text to lower-case
51  words = text.split() # split the text into list of words
52  count_moby = 0
53  # Precompute base^(m-1) for the rolling hash
54  h = pow(base, l-1) % prime
55  # Compute initial hash values
56  t_hash = 0
57  #s_hash = 0
58  for i in range(l):
59  t_hash = (base*t_hash + ord(t[i]))%prime # calculate t_hash for all words in s.
60  for word in words:
61  k = len(word)
62  s_hash = 0
63  # Calculate initial hash value of current word.
64  if k < l:
65  continue
66  for i in range(l):
67  s_hash = (base * s_hash + ord(word[i])) % prime
68  # Check each potential match
69  for i in range(k - l + 1):
70  # If hashes match, verify character by character
71  if s_hash == t_hash:
72  # Verify match (in case of hash collision)
73  match = True
74  for j in range(l):
75  if word[i+j] != t[j]:
76  match = False
77  break
78  if match:
79  count_moby += 1  # Pattern found at position i
80  # Compute hash for next window
81  if i < k - l:
82  # Remove leading digit, add trailing digit, multiply by base
83  s_hash = (base * (s_hash - ord(word[i]) * h) + ord(word[i+l])) % prime
84
85  # Make sure hash is positive
86  if s_hash < 0:
87  s_hash += prime
88
89  return count_moby
90
91  def main():
92  book_path = "moby_dick.txt" # I assume that the book moby dick is present as a .txt
      file in the same directory as the code.
93  moby_dick = get_book_text(book_path)
94  print(rabin_karp(test,"whale"))
95  print(naive_search_with_substrings(moby_dick,"whale"))
96  print(naive_search_words_only(moby_dick,"whale"))
97  print(rabin_karp(moby_dick,"whale"))
98
99  main()
```

1.2 Auswertung:

Beide Algorithmen finden den substring "whale" 1702 mal und 529 mal den string "whale" im Buch Moby Dick. Einen merkbaren Unterschied für die Laufzeit bzw. Berechnungszeit der Algorithmen konnten wir (auf unseren Systemen/ Rechnern) nicht feststellen.

Laufzeitkomplexität naiver Algorithmus: $O(kl * |mb|) \rightarrow O(kl)$ ist der Average-case des naiven Algorithmus, wobei k die Länge des aktuellen Strings ist und l die Länge des gesuchten Substrings. $|mb|$ ist die Länge des Buches Moby Dick. Laufzeitkomplexität Rabin-Karp Algorithmus: $O(|mb| * (k+l)) \rightarrow O(k+l)$ ist der Average-case des Rabin-Karp Algorithmus, wobei k die Länge des aktuellen Strings ist und l die Länge des gesuchten Substrings. $|mb|$ ist die Länge des Buches Moby Dick. Obwohl ein theoretischer Unterschied besteht, ist das Buch Moby Dick nicht lang genug, um diesen deutlich zumachen. Im Fall von Moby Dick bewegen sich die Unterschiede im nano- bis Millisekunden Bereich (auf unseren Systemen).

2 Problem: Suchen in Zeichenketten II

2.1 Rabin-Karp mit mehreren Suchmustern

Der Algorithmus von Rabin-Karp lässt sich leicht auf mehrere Suchmuster verallgemeinern. Gegeben eine Zeichenkette s und Suchmuster $t1, \ldots, tk$, bestimme die erste Stelle in s, an der eines der Muster $t1, \ldots, tk$ vorkommt. Beschreiben Sie, wie man den Algorithmus von Rabin-Karp für diese Situation anpassen kann. Was ist die heuristische Laufzeit Ihres Algorithmus (unterder Annahme, dass Kollisionen selten sind)?

2.1.1 Probelmstellung:

Gegeben:

- Eine Zeichenkette s (Text) der Länge n
- Ein Suchmuster k mit $t_1, t_2, \ldots, t_k$ der gleichen Länge m

2.1.2 Gesucht:

- Ein Algorithmus: der die erste Position in s (Text), an der irgendeins der Muster $t_1, \ldots, t_k$ vorkommt.
- Die Laufzeit des Algorithmus (unter der Annahme, dass Kollisionen selten sind)

2.1.3 Lösung:

Rabin-Karp vorgehen:

- Wir berechnen den Hashwert des Musters t
- Wir Iterieren über den Text s mit einem Fenster/Bereich der Länge m
- Berechne den Hashwert des aktuellen Fensters $s[i \ldots i + m - 1]$
- Wenn die Hashwerte übereinstimmen, vergleichen wir den Text-ausschnitt und Muster direkt, um Kollisionen zu umgehen

Rabin-Karb für mehrere Muster: [1]

- Wir berechnen den Hashwert für alle Suchmuster $t_1, t_2, \ldots, t_k$
  - Diese Hashwerte speichern wir in einer Datenstruktur (HashSets)
  - Dadurch ist die Überprüfung, ob ein Hashwert zu einem Muster gehört, in $O(1)$ möglich
- Wiederholen des normalen Algorithmus, Iterieren über alle Teilstrings der Länge m im Text s
- Berechnen des aktuellen Hashwertes vom Fenster
- Vergleichen, ob dieser Hashwert in der Menge der HashSets zu finden ist
- Wenn die Hashwerte übereinstimmen, verlgeichen wir wieder direkt

Eigenschaften eines HashSets in Scala: Eine HashSet-Struktur ist eine Datenstruktur, die eine Menge von eindeutigen Werten speichert und sehr schnelle Einfüge-, Such- und Löschoperationen erlaubt - im Schnitt in konstanter Zeit $o(1)$

- HashSets haben die Eigentschaft keine Duplikate zu erlauben, d.h. jeder Wert wird nur einmal gespeichert, doppelte werden ignoriert
- Schnelle Suche von Werten (sofern keine Kollision)
- Ein HashSet verwendet intern eine Hashfunktion um die Werte zu speichern und zu finden

Um HashSets zu benutzten importieren wir folgende Bibliothek:

```
import scala.collection.mutable.HashSet
```

Pseudocode könnte wie folgt aussehen:

```
import scala.collection.mutable.HashSet

// Hashfunktion mit Rolling Hash
def hash(s: String): Int = ...

val musterHashes = HashSet[Int]() // Initialisiere HashSet
val muster = List("bob", "tim", "leo") // Suchmuster s[i ... i+m-1]
val m = muster.length // m = Laenge des Musters bei unterschiedlicher musterlaenge
    sollte m die wenigsten charaktere haben

// Rabin-Karb vorgehen fuer mehrere Muster:
// 1. Wir berechnen den Hashwert fuer alle Suchmuster t1,t2,...,tk
for muster <- muster do
  musterHashes.add(hash(muster)) //speicher die Hashwerte im Set

// wir Iterieren ueber den Text s mit der Fenstergroesse von m
for i <- 0 to s.lenght - m do
  val fenster = s.fenster(i, i + m)
  cal fensterHash = hash(fenster)

  if musterHashes.contains(fensterHash) then
    // Direkte kontrolle ob die muster(string) und Text uebereinstimmen
    if muster.contains(fenster) then
      return i
```

Heuristische Laufzeit: TODO

2.2 Implimentierung des Algorithmus

Implementieren Sie Ihren Algorithmus aus 2.1. Beantworten Sie sodann folgende Frage: Was kommt öfter in dem Roman Sense & Sensibility vor: sense oder sensibility/sensible?
Hinweis: Siehe http://www.gutenberg.org/files/161/161-0.txt.

3 Problem: Suche in Zeichenketten III

Sei $\sum = C, G, T, A$. Sei $s = CTTGGATTA$ und $t = TTA$.

3.1 Naiver Algorithmus

Verwenden Sie den naiven Algorithmus, um festzustellen, ob/wo das Muster t in der Zeichenkette s vorkommt. Zeigen Sie die einzelnen Schritte.

```
1. i = 0, j = 0 --> s[0], t[0]: C != T
2. i = 1, j = 0 --> s[1], t[0]: T == T
2.2. i = 2, j = 1 --> s[2], t[1]: T == T
2.3. i = 3, j = 2 --> s[3], t[2]: G != T
3. i = 4, j = 0 --> s[4], t[0]: G != T
4. i = 5, j = 0 --> s[5], t[0]: A != T
5. i = 6, j = 0 --> s[6], t[0]: T == T
5.2. i = 7, j = 1 --> s[7], t[1]: T == T
5.3. i = 8, j = 2 --> s[8], t[2]: A == A ==> t found.
Der String t kommt am Index 6/ 7. Position in s vor.

A : 0, T : 1, G : 2, C : 3 und die Primzahl 5
```

3.2 Rabin-Karp Algorithmus

Verwenden Sie den Algorithmus von Rabin-Karp, um festzustellen, ob/wo das Muster t in der Zeichenkette s vorkommt. Verwenden Sie A : 0, T : 1, G : 2, C : 3 und die Primzahl 5 als Modulus für die Hashfunktion. Zeigen Sie die einzelnen Schritte.

```
s = CTTGGATTA und t = TTA
h(t) = h(TTA) = 4^2+4^1+0 = 4^2+4 = 20 mod 5 = 0
1. h(CTT) = 3*4^2+4^1+4^0 = 56 mod 5 = 1 --> 0 != 1 -> not found
2. h(TTG) = 4^2+4+2 = 22 mod 5 = 2 --> 0 != 2 -> not found
3. h(TGG) = 4^2+2*4+2 = 26 mod 5 = 1 --> 0 != 1 -> not found
4. h(GGA) = 2*4^2+2*4 = 40 mod 5 = 0 --> 0 == 0 -> found at position 4 / index 3
Charaktervergleich nach dem naiven Algorithmus:
1. i = 0, j = 0 --> s'[0], t[0]: G != T -> not found t ist nicht an der Position 4 n s.
5. h(GAT) = 2*16+1 = 17 mod 5 = 2 --> 0 != 2 -> not found
6. h(ATT) = 4+1 = 5 mod 5 = 0 --> 0 == 0 -> found at position 6 / index 5
Charaktervergleich nach dem naiven Algorithmus:
1. i = 0, j = 0 --> s'[0], t[0]: A != T -> not found t ist nicht an der Position 6 n s.
7. h(TTA) = 16+4 = 20 mod 5 = 0 --> 0 == 0 found at position 6/ Index 7
Charaktervergleich nach dem naiven Algorithmus:
1. i = 0, j = 0 --> s'[0], t[0]: T == T
2. i = 1, j = 1 --> s'[1], t[1]: T == T
3. i = 2, j = 2 --> s'[2], t[2]: A == A
s' == t
Der String t kommt am Index 6/ 7. Position in s vor.
```

3.3 Knuth-Morris-Pratt Algorithmus

Verwenden Sie den Algorithmus von Knuth-Morris-Pratt, um festzustellen, ob/wo das Muster t in der Zeichenkette s vorkommt. Zeigen Sie die einzelnen Schritte.

TODO

References

[1] Nick Dandoulakis-User. *Using Rabin-Karp to search for multiple patterns in a string.* URL: https://stackoverflow.com/questions/1318126/using-rabin-karp-to-search-for-multiple-patterns-in-a-string. (accessed: 25.06.2025).