

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:

```

1 # task 1 isn't clear on what is expected. It states to return the count of the word "
  whale" but specifies substring search algorithms to do so.
2 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?"
3 def get_book_text(path): # helper function to access moby_dick.txt
4     with open(path) as f:
5         return f.read()
6
7 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.
8     text = s.lower() # convert the given string/text to lower-case
9     words = text.split() # split the text into list of words
10    l = len(t)
11    moby_counter = 0
12    for word in words:
13        k = len(word)
14        for i in range(k-1+1): # Try each starting position
15            j = 0
16            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
17                j += 1
18            if j == l: # if j is greater than len(t) then t is in s first at
  index i.
19                moby_counter += 1 # Pattern found at position i
20    return moby_counter
21
22 def naive_search_words_only(s,t):
23     text = s.lower() # convert the given string/text to lower-case
24     words = text.split() # split the text into list of words
25     count_moby = 0
26     for word in words:
27         if word == t:
28             count_moby += 1
29     return count_moby
30
31
32 def get_lower_case_alpha_list(): # helper function to get a list of the latin alphabet
33     # initialise an empty list
34     list = []
35     # filling the list with lowercase letter in alphabetical order
36     alpha = 'a'
37     for i in range(0, 26):
38         list.append(alpha)
39         alpha = chr(ord(alpha) + 1)
40     return list

```

```

41
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
61     for word in words:
62         k = len(word)
63         s_hash = 0
64         # Calculate initial hash value of current word.
65         if k < l:
66             continue
67         for i in range(l):
68             s_hash = (base * s_hash + ord(word[i])) % prime
69         # Check each potential match
70         for i in range(k - l + 1):
71             # If hashes match, verify character by character
72             if s_hash == t_hash:
73                 # Verify match (in case of hash collision)
74                 match = True
75                 for j in range(l):
76                     if word[i+j] != t[j]:
77                         match = False
78                         break
79                 if match:
80                     count_moby += 1 # Pattern found at position i
81             # Compute hash for next window
82             if i < k - l:
83                 # Remove leading digit, add trailing digit, multiply by base
84                 s_hash = (base * (s_hash - ord(word[i]) * h) + ord(word[i+l])) % prime
85
86             # Make sure hash is positive
87             if s_hash < 0:
88                 s_hash += prime
89
90     return count_moby
91
92 def main():
93     book_path = "moby_dick.txt" # I assume that the book moby dick is present as a .
94     txt file in the same directory as the code.
95     moby_dick = get_book_text(book_path)
96     print(rabin_karp(test, "whale"))
97     print(naive_search_with_substrings(moby_dick, "whale"))
98     print(naive_search_words_only(moby_dick, "whale"))
99     print(rabin_karp(moby_dick, "whale"))
100
101 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 t_1, \dots, t_k , bestimme die erste Stelle in s , an der eines der Muster t_1, \dots, t_k vorkommt. Beschreiben Sie, wie man den Algorithmus von Rabin-Karp für diese Situation anpassen kann. Was ist die heuristische Laufzeit Ihres Algorithmus (unter der Annahme, dass Kollisionen selten sind)?

2.1.1 Problemstellung:

Gegeben:

- Eine Zeichenkette s (Text) der Länge n
- Ein Suchmuster k mit t_1, t_2, \dots, 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, \dots, 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 $s[i \dots i + m - 1]$
- Wenn die Hashwerte übereinstimmen, vergleichen wir den Text-ausschnitt und Muster direkt, um Kollisionen zu umgehen

Rabin-Karp für mehrere Muster: [1]

- Wir berechnen den Hashwert für alle Suchmuster t_1, t_2, \dots, 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, vergleichen 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 Löschooperationen erlaubt - im Schnitt in konstanter Zeit $O(1)$

- HashSets haben die Eigenschaft 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 benutzen importieren wir folgende Bibliothek:

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

Pseudocode könnte wie folgt aussehen:

```
1 import scala.collection.mutable.HashSet
2
3 // Hashfunktion mit Rolling Hash
4 def hash(s: String): Int = ...
5
6 val musterHashes = HashSet[Int]() // Initialisiere HashSet
7 val muster = List("bob", "tim", "leo") // Suchmuster s[i ... i+m-1]
8 val m = muster.length // m = Laenge des Musters bei unterschiedlicher musterlaenge
9 // sollte m die wenigsten caractere haben
10
11 // Rabin-Karb vorgehen fuer mehrere Muster:
12 // 1. Wir berechnen den Hashwert fuer alle Suchmuster t1,t2,...,tk
13 for muster <- muster do
14   musterHashes.add(hash(muster)) //speicher die Hashwerte im Set
15
16 // wir iterieren ueber den Text s mit der Fenstergroesse von m
17 for i <- 0 to s.length - m do
18   val fenster = s.substring(i, i + m)
19   val fensterHash = hash(fenster)
20
21   if musterHashes.contains(fensterHash) then
22     // Direkte kontrolle ob die muster(string) und Text uebereinstimmen
23     if muster.contains(fenster) then
24       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 $\Sigma = C, G, T, A$. 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).