FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION

OF HIGHER EDUCATION

ITMO UNIVERSITY

Report

on the practical task No. 8

“Practical analysis of advanced algorithms”

Performed by

*Mariia Koroleva*

*J42322c*

Accepted by

Dr Petr Chunaev

St. Petersburg

2022

**Goal**

*Practical analysis of advanced algorithms*

**Formulation of the problem**

***Book:*** *Thomas H. Cormen Charles E. Leiserson Ronald L. Rivest Clifford Stein Introduction to Algorithms Third Edition, 2009 (or other editions).*

*Sections:*

*I Foundations*

*4 Divide-and-Conquer*

*5 Probabilistic Analysis and Randomized Algorithms*

*VI Graph Algorithms*

*23 Minimum Spanning Trees*

*25 All-Pairs Shortest Paths*

*26 Maximum Flow*

*IV Advanced Design and Analysis Techniques*

*15 Dynamic Programming*

*16 Greedy Algorithms*

*VII Selected Topics*

***I.*** *Choose* ***two*** *algorithms (interesting to you and not considered in the course) from the above-mentioned book sections.*

***II.*** *Analyse the chosen algorithms in terms of time and space complexity, design technique used, etc. Implement the algorithms and produce several experiments. Analyse the results.*

**Brief theoretical part**

For the analysis, it was decided to choose the *Naïve string matcher* and *Rabin-Karp* algorithms. Both algorithms are designed to solve the so-called *string-matching problem*.

Task ormalization. It is assumed that given:

* *Text* - can be represented as an array T[1:n+1], where n is the length of the text;
* *Sample* (template) in the form of an array P[1:m+1], where m is the length of the template (m<=n);
* The final *alphabet* ∑ (it can be a list of letters, numbers or other characters), where d is the number of characters in the alphabet (|∑|).

A pattern P is said to occur in a text with the *shift* s if 0<=s<=n-m and T[s+1:s+m+1] = P[1:m+1]. In other words, fragment P occurs in text T starting from position s+1.

The *task* is to *find this shift s*. Or, in case if the pattern P is not contained in T, return a result stating that the fragment was not found.

The *Naïve string matcher* algorithm runs with a loop that checks for each possible *shift value s* that P = T[s+1:s+m+1].

*Rabin-Karp algorithm*. It works like the naïve string matcher algorithm, however, at each iteration, not substrings are compared, but their hashes - all characters are considered as numbers in the base d number system - the length of the dictionary. Thus, the algorithm consists of two parts: preprocessing and comparison.

The pseudocode looks like this:

Изображение выглядит как текст

Автоматически созданное описание

The third line calculates h - a fixed value, which should be equal to P[0]. The value of q is chosen such that the equality is observed:

h = d\*\*(m-1) % q

The value t\_0 is a digital representation of the fragment T[0:m+1] at shift s=0, t\_s – at shift s=s. At the same time, if we know t\_s, we can always calculate the next t\_s+1 using the formula (line 14).

In line 11, after comparing hashes, we see a repeated comparison of strings in their initial string format. That was done to avoid collisions can occur, where fragments with different content can have the same hashes.

**Results**

***Naïve string matcher algorithm analysis***.

*Time complexity*: o((n-m+1)m).

n-m+1 is a number of iterations of s. m is the length of P (fragment), so the comparison complexity is equal to m.

*Space complexity*. We are dealing with string, the largest of which is the text itself. As we can see from time-complexity, the execution time depends on its size and the size of the substring being searched for.

*Design technique used*. A loop with a comparison procedure on each iteration.

***Rabin-Karp algorithm analysis.***

*Time complexity*: o(m) + o((n-m+1)m), where v is a number of possible shifts.

O(m) time is spent on preprocessing (hashing) and o((n-m)m) is spent on the loop with comparisons.

*Space complexity*: besides containing the strings, we should preserve the dictionary of possible symbols, where every symbol is allocated by a number.

*Design technique used*. The algorithm consists of two parts: preprocessing and comparison. The comparison part is designed as in the naïve string matcher algorithm, however instead of string comparison, hash-number comparison is used.

**Conclusions**

There is a huge number of different algorithms of text search more effective than considered here. However, these two are the ones where everything was beginning.

Rabin and Karp introduced the idea that is used in all modern search machines.

**Appendix**

**# Naive String Matcher algorithm**

**def** NSM(T, P):

  '''

  Finds a fragment P in a text T by use of Naive string matcher algorithm

  Params:

  T - str, a text in which the search will be implemented

  P - str, a substring to be found in T

  Output:

  s - array of integers, the shifts (the indices where P is found in T)

  '''

  shifts = []

  n, m = len(T), len(P)

  for s in range(n-m + 1):

        if P == T[s+1 : s+m+1]:

            shifts.append(s)

  return shifts

t = 'naive string matcher alg'

p = 'al'

NSM(t, p)

p = 'dlw'

NSM(t, p)

p = 'a'

NSM(t, p)

**# Rabin-Karp algorithm**

3 versions of python implementation if RK algorithm are considered here:

1. Digit string case

2. Strings with any symbols

3. Algorithm with a use of a ready hash function

*# Digit case*

**def** RabKarp(T, P, d=10):

    '''

    Finds a fragment P in a text T by use of Rabin-Karp algorithm.

    Only digital text is considered here.

    Params:

    T - str, a text in which the search will be implemented

    P - str, a substring to be found in T

    d - int, the length of the dictionary of symbols used;

    Output:

    s - array of integers, the shifts (the indices where P is found in T)

    '''

    shifts = []

    n, m = len(T), len(P)

*# Selecting the q*

    for q in range(2, n\*\*3):

        h = d\*\*(m-1) % q

        if h == int(P[0]):

            break

    p, t = 0, 0

*# Finding the coded form of P and T[0:m+1]*

    for i in range(m):

        p = (d\*p + int(P[i])) % q

        t = (d\*t + int(T[i])) % q

*# Comparing with different slices of T*

    for s in range(n-m+1):

        if p == t:

            if int(P) == int(T[s : s+m]):

                shifts.append(s)

        if s < n - m:

            t = (d\*(t - int(T[s])\*h) + int(T[s+m]))%q

    return shifts

T = '49503829587592023829'

P = '950'

d = 10

RabKarp(T, P, 10)

P = '95'

RabKarp(T, P, 10)

P = '9'

RabKarp(T, P, 10)

*# Adapting to the case with symbols*

**def** RabKarp\_str(T, P):

    '''

    Finds a fragment P in a text T by use of Rabin-Karp algorithm.

    Params:

    T - str, a text in which the search will be implemented

    P - str, a substring to be found in T

    Output:

    s - array of integers, the shifts (the indices where P is found in T)

    '''

    shifts = []

    n, m = len(T), len(P)

*# Creating a dictionary from used symbols with a length d*

    alphabet = set(T)

    d = len(alphabet)

    h\_dict = {l:i for i, l in enumerate(alphabet)}

*# Selecting the q*

    for q in range(2, n\*\*3):

        h = d\*\*(m-1) % q

        if h == h\_dict[P[0]]:

            break

    p, t = 0, 0

    for i in range(m):

        p = (d\*p + h\_dict[P[i]]) % q

        t = (d\*t + h\_dict[T[i]]) % q

    for s in range(n-m+1):

        if p == t:

            if P == T[s : s+m]:

                shifts.append(s)

        if s < n - m:

            t = (d\*(t - h\_dict[T[s]]\*h) + h\_dict[T[s+m]])%q

    return shifts

T='dkdl dld lleed'

P = 'dld'

RabKarp\_str(T, P)

T = 'djkss;; w[[[w jjedw dj'

P = 'dj'

RabKarp\_str(T, P)

*# String case using hash function*

**def** RabinKarp\_hash(T, P):

    '''

    Finds a fragment P in a text T by use of Rabin-Karp algorithm.

    Params:

    T - str, a text in which the search will be implemented

    P - str, a substring to be found in T

    Output:

    s - array of integers, the shifts (the indices where P is found in T)

    '''

    n, m = len(T), len(P)

    p = hash(P);

    shifts = []

    for s in range(n-m+1):

        t = hash(T[s:s+m])

        if t == p:

            if T[s:s+m] == P:

                shifts.append(s)

    return shifts

T = 'Empty spaces, what are we living for... Abandoned places...'

P = 'living'

RabinKarp\_hash(T, P)

P = 'mine'

RabinKarp\_hash(T, P)

P = 'e'

RabinKarp\_hash(T, P)