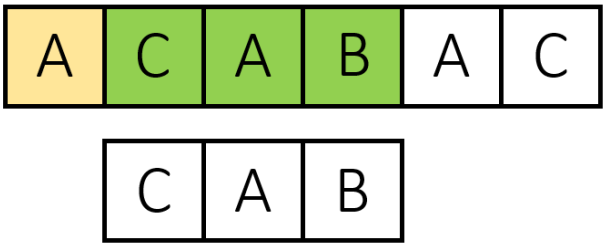


Habib University
CS 412: Algorithms

String Matching Problem

Algorithms to solve it and their analysis

Authors
Arham Ahmed
Maham Shoaib Patel
Murtaza Faisal Shafi
Khubaib Naeem Kasbati

December 7, 2020

Contents

Chapter 1

Introduction to String Matching Problem

The problem of finding occurrence(s) of a pattern string within another string or body of text. There are many different algorithms for efficient searching.[1] String Matching Problem is simply finding occurrence(s) of a `pattern` (substring) in a `text` (string).
Mathematically, the problem is finding occurrence(s) of a `pattern` x where x is a series of m characters, $x = \langle x_1, x_2, \ldots x_m \rangle$, $x_i \in \Sigma, i = 1, 2, \ldots, m$ in a `text`, y where y is a series of n characters, $y = \langle y_1, y_2, \ldots y_n \rangle, j = 1, 2, \ldots, n$.[2]
We will be focusing on finding all the occurrences of a `pattern` in `text`. This problem is relevant in the real world and some of its applications include:

1. Plagiarism detection
2. DNA sequencing to find patterns of a particular DNA sequence.
3. Text editors.
4. Searching for files.
5. Spam filters.

[1] You can find all implementations of our algorithms here.
[2] Definition inspired from CS 369 University of Auckland [1]

Chapter 2

Solutions to the Problem

In this portion, we will look at 3 algorithms to solve the string matching problem.

1. Brute Force / Naïve Algorithm
2. Rabin-Karp Algorithm
3. Knuth-Morris-Pratt Algorithm

2.1 Brute Force / Naïve Algorithm

Brute Force or the Naïve Algorithm is a relatively easier and simple algorithm to follow. However, the algorithm is slower than other solutions to the problem. It simply creates a window size of size m which is equal to the size of the *pattern* being detected in the *text*. It slides the window over every turn and compares the window to the pattern and if it matches then it simply returns a match at the starting index of the window.

Some features of the Brute Force algorithm are as follows:

- It does not require any pre-processing of the text or the pattern.
- It shifts the window by 1 after every comparison.
- It compares every element in the pattern with every element in the window with no specific order of comparison.

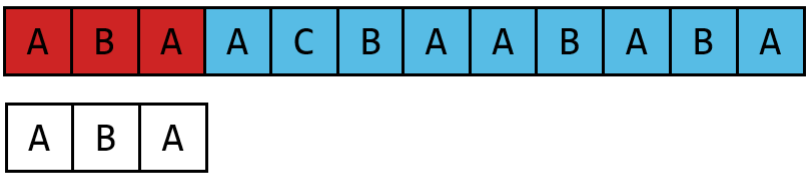
2.1.1 Algorithm

The brute force algorithm, as explained above, works through making consecutive comparisons by sliding a window of size m (which is the size of the pattern) over a text of size n.
After the window is created, we then make comparisons of every letter at index i of the window with the letter at same index i of the pattern. If every element in the window matches that of the pattern then we return the starting index of the window in the text. We can thus write the pseudo code as follows:

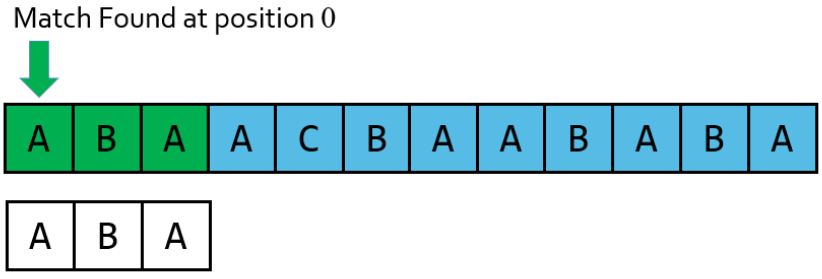
```
procedure BruteForce(pattern, text)
    for j ← 0 to len(text) − len(pattern) − 1 do
        for i ← 0 to len(pattern) − 1 do
            if text[i+j] ≠ pattern[i] then
                break
            end if
            if i = len(pattern) − 1 then
                print(i)
            end if
        end for
    end for
end procedure
```

2.1.2 Example

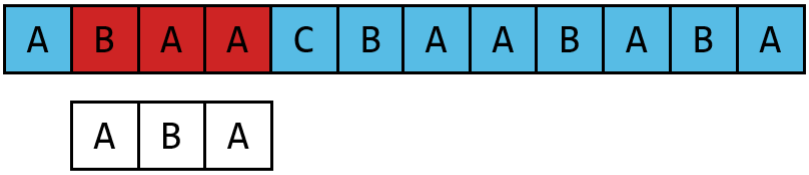
The functionality of the algorithm can be explained by taking the example of the following text: **ABAACBAABABA** where we need to search for the pattern: **ABA**. The algorithm will create a window of size m which is 3 in this case and create the window over the first three elements of the text as follows:



As all the elements of the window and the pattern match, a match is produced at the index of the first element of the window in the text as follows:



The window then slides over by one and compares all the elements of the window with those of the pattern again. If no match is produced, the window slides to the right by 1 and continues its search.



It continues to do so until it reaches the end and prints/returns the indexes of the the first element of the window after every match is produced.

2.1.3 Code

The algorithm can be coded in python as follows:

```
def NaiveAlgorithm(pattern, text):
    m = len(pattern)
    n = len(text)
    for i in range(n-m+1):
        for j in range(m):
            if text[i + j] != pattern[j]:
                break
            if j == m - 1:
            print("Pattern is found at position: " + str(i))
```

2.1.4 Complexity Analysis

The time complexity of the algorithm is $O(n \cdot m)$ as the upper bound for comparisons is $m \cdot n$.

2.2 Rabin-Karp Algorithm

Rabin-Karp is an algorithm which tries to avoid unnecessary comparison through hashing, since comparing two hashes is faster than comparing every character one by one. We use a version of hashing called **rolling hash**.

2.2.1 Rolling Hash

A rolling hash is a hash function where the input is hashed in a window which moves across the input. It uses the previously known hash value and the character that enters the window and the one that leaves it in order to compute hash value. For that initially, we need to compute the hash value, which can be done using the following algorithm.

```
function HASH(word, quotient)
    result ← 0
    m ← len(word)
    for i ← 0 to m − 1 do
        result ← (result + ord(word[i])×2^(m−1−i)) mod quotient
    end for
    return result
end function
```

The Pseudo-code for the rolling hash is as follows

```
function ROLLINGHASH(remove, add, oldvalue, windowsize, quotient)
    m ← windowsize
    return (2× oldvalue - ord(remove)×2^m + ord(b) ) mod quotient
end function
```

2.2.2 Algorithm

Let m be size of `pattern` and n be size of `text`. The algorithm is as follows:

1. First compute the hash of the first m characters in hash and hash of the *pattern*.
2. See if the hash of the first m characters matches with the hash of the *pattern*. If so, check the m characters in the window and see if they actually match with the *pattern*. If it does, that is one of our answers.
3. Move the window by 1 and calculate the new hash by using the rolling hash function and see if the new hash matches the hash of the *pattern*. If so, check the m characters in the window and see if they actually match with the *pattern*. If it does, that is one of our answers.If the window doesn't end on the last character in the *text*, then repeat this step.

The Pseudo-code for the algorithm is as follows:

```
procedure RabinKarp(pattern, text, quotient)
    m ← len(pattern)
    n ← len(text)
    hashpattern ← HASH(pattern,quotient)
    hashtext ← HASH(text[:m],quotient)
    for j ← 0 to n − m − 1 do
        if hashtext = hashpattern and text[j:j+m] = pattern then
            print(j)
        end if
        if j+m < n then
            hashtext ← ROLLINGHASH(text[j], text[j+m], hashtext, m, quotient)
        end if
    end for
end procedure
```

2.2.3 Example

The following is the steps taken when the algorithm is given the parameters, `pattern` = "CDD", `text` = "CDDCDD", `quotient` = 524287

```
text = CDD , hash_text = 472 , hash_word = 472
Pattern is found at position: 0
text = DDC , hash_text = 475 , hash_word = 472
text = DCD , hash_text = 474 , hash_word = 472
text = CDD , hash_text = 472 , hash_word = 472
Pattern is found at position: 3
```

2.2.4 Code

```
def hash_(word, m, quotient):
    result = 0
    for i in range(m):
        result = (result + ord(word[i])*2**(m-(1+i))) % quotient
    return result

def rehash(a, b, hash_value, m, quotient):
    '''
    a : character leaving window.
    b: character entering the window
    hash_value : Previous Hash value
    m : length of window
    quotient: odd number to take mod by.
    Finds the new hash using old hash
    '''
    return (2*hash_value- ord(a)*2**(m) + ord(b))%quotient

def RabinKarp(word, text, quotient):
    '''
    Applies Rabin-Karp Algorithm to find all instances
    of the string.
    '''
```

```
m = len(word)
print("m = ", m)
n = len(text)
print("n = ", n)
hash_word = hash_(word, m, quotient)
hash_text = hash_(text, m, quotient)
for j in range(n-m+1):
    print("text =", text[j:j+m], ", hash_text =", hash_text, ",
        hash_word =", hash_word)
    if(hash_word == hash_text and text[j:j+m] == word):
        print("Pattern is found at position: " + str(j))
    if (j+m < n):
        hash_text = rehash(text[j], text[j+m], hash_text, m, quotient)
```

2.2.5 Complexity Analysis

The Complexity for this algorithm in the worst case is $O(m \cdot n)$ and a case where this can happen is when we have a poor hash function, where the hash function results in a lot of false positives and has to check every pattern like the brute force. But the best/average case is $O(n + m)$ since a lot of character checks are skipped due to hashing.

2.3 Knuth-Morris-Pratt Algorithm

KMP algorithm [2] was developed by Knuth Morris and Pratt independently and it is the first ever **linear time algorithm** for string matching. Unlike brute force, it totally avoids the re-examination of the previously matched characters.

2.3.1 Algorithm with an example

In order to understand the algorithm, we will first develop it intuitively by working on an example where `pattern` = `onions` and `text` = `onionions`
Every time brute force fails, it starts matching from the next character. In doing this we are not employing the knowledge from our last matching. We can use that knowledge and features of the *pattern* to know where the next matching should begin from or how many characters we can skip. These features of the pattern are stored in a prefix table, where for each of the possible string index $\{1, 2, \ldots m\}$,prefix$[i], i \in \{1, 2, \ldots m\}$, stores the length of the longest prefix of p which is also a **proper** suffix of the substring `pattern[1,. . . , i]`. The following is an example of the prefix table.

i	0	1	2	3	4	5
`pattern[i]`	o	n	i	o	n	s
`LPS[i]`	0	0	0	1	2	0

Table 2.1: Prefix Table

This prefix table can tell us how many characters of the pattern, we don't have to match again if there is a mismatch.
In our example the following happens:

1. The word matches till onion. Then there is a mismatch in the letter between **i** in the *text* and **s** in the *pattern*.
2. Our LPS array tells us since **on** is the common prefix and suffix, we can skip the first 2 characters of the *pattern* and try to match the 3rd character of *pattern* with **i**.
3. There is a match between 3rd character of *pattern* and **i** and as we move till the end of the string, there is a match and we find the instance of onions.

2.3.2 Prefix Table

we start by taking two pointers, *len* and i and set them to 0 and 1 respectively and take an array of the size of the pattern and name it `LPS` for ease. If there is a match at index *len* and index i, we will store *len* + 1 at i'th index of `LPS` array.

If there is a mismatch and the *len* pointer has not incremented, we will just increment i and continue with our search.

If *len* pointer has incremented and there is a mismatch then we will simply check the value at the index before *len* pointer in the `LPS` array (`LPS[len-1]`) and assign that value to *len*.

We will continue this until the i pointer has reached the end.

Here is an example, Consider a string:
ABCDABEABF *len* and i are 0. As we can see in the string, character 'A' is occurring at $i = 4$, we put $len + 1$ at `LPS[i]`, we increment *len* and i then we see 'B' after it at $i = 5$, which is matching with the second character of the string so we put 2 ($len + 1 = 2$) at `LPS[i]`. we can observe that 'AB' is a suffix which is occurring in the string hence we have accounted that.

i	0	1	2	3	4	5	6	7	8	9
`pattern[i]`	**A**	**B**	C	D	**A**	**B**	E	A	B	F
`LPS[i]`	0	0	0	0	1	2				

Table 2.2: Prefix Table

We can see that the character after 'B' is not matching with first and second character and we see that *len* pointer has incremented and there is a mismatch, so we will simply check the value at the index before *len* pointer in the `LPS` array `(LPS[len-1])` and assign that value to *len*. and continue this strategy until we reach the end.

i	0	1	2	3	4	5	6	7	8	9
`pattern[i]`	A	B	C	D	A	B	E	A	B	F
`LPS[i]`	0	0	0	0	1	2	0	1	2	0

Table 2.3: Prefix Table

2.3.3 Pseudocode

The Pseudocode for Knuth Morris Algorithm is as follows:

```
procedure KMPSearch(pattern,text)
    LPS ← LPSARRAY(pattern)
    i ← 0
    j ← 0
    while i < len(text) do
        if text[i] = pattern[j] then
            i ← i + 1
            j ← j + 1
        else
            if j ≠ 0 then
                j ← LPS[j − 1]
            else
                i ← i + 1
            end if
        end if
        if j = len(pattern) then
            print(i − j)
            j ← LPS[j − 1]
        end if
    end while
end procedure
```

The algorithm to compute the LPS array is as follows:

```
function LPSArray(pattern)
    prefixlen ← 0
    i ← 1
    prefixlen ← 0
    LPS [ len(pattern) ]
    LPS ← [0,...,0]
    while i < len(pattern) do
        if pattern[i] = pattern[prefixlen] then
            LPS[i] ← prefixlen + 1
            prefixlen ← prefixlen + 1
            i ← i + 1
        else
            if prefixlen ≠ 0 then
                prefixlen ← LPS[prefixlen − 1]
            else
                LPS[i] ← 0
                i ← i + 1
            end if
        end if
    end while
    return LPS
end function
```

2.3.4 Code

```
def get_LPS_array(pattern):
    '''
    Computes the LPS Array using the method
    explained in the report
    '''
    length = 0
    i = 1
    lps = [0] * len(pattern)
    while i < len(pattern):
        if pattern[i] == pattern[length]:
            lps[i] = length+1
            length += 1
            i += 1
        else:
            if length != 0:
                length = lps[length-1]
            else:
                lps[i] = 0
                i += 1
    return lps


def KMP_search(pattern, text):
    '''
```

```
    Applies KMP Algorithm to find all instances
    of the string.
    '''
    lps = get_LPS_array(pattern)
    i, j = 0, 0
    while i < len(text):
        if text[i] == pattern[j]:
            i += 1
            j += 1
        else:
            if j != 0:
                j = lps[j-1]
            else:
                i += 1
        if j == len(pattern):
            print(i-j)
            j = lps[j-1]
```

2.3.5 Complexity Analysis

Let length of `pattern` $= m$ and length of `text` $= n$.The creation of the `LPS` array takes $O(m)$ time, since i needs to be incremented $O(m)$ times, and the prefixlen change command decreases prefixlen by at least 1, and can be maximum incremented $O(m)$ times in the algorithm hence that statement runs max $O(m)$ times giving complexity $O(m)$
The main KMP Algorithm i pointer is incremented $O(n)$ times and j pointer can be rolled back when there are mismatches. Since j can only roll back as much as it has progressed upto mismatch, each character is examined atmost twice, hence there are at most $O(n)$ comparison leading to the algorithm complexity as $O(n + m)$

Chapter 3

Empirical Analysis

Complexities gives us a fair idea of which algorithm are better in theory. In reality there are other costs as well that hidden in complexity. In complexity analysis, we usually see the worst case, but those are just upper bounds and in reality may not be the same. It also helps compare algorithms with the same complexity, for example Rabin-Karp and Brute Force who have the same worst case analysis.

3.1 Review of Complexities

The below are the theoretical complexities that we knew. We can use them to see whether our results match what we would expect or not.

Algorithm	**Complexity**
Brute Force Algorithm	Worst case: $O(nm)$
Rabin-Karp Algorithm	Worst case: $O(nm)$
	Average/Best Case: $O(n+m)$
KMP Algorithm	Worst case: $O(n+m)$

Table 3.1: Theoretical Complexities

3.2 How Analysis was done

The analysis was done on the same computer and in the same language (Python) in order to ensure fair comparison.

1. To see the impact of text size, text size was varied from 5000 to 1,000,000 in steps of 5,000
2. To see the impact of pattern size, pattern size was varied from 100 to 10000 in steps of 50.
3. In the testing stage, the algorithm was run thrice for each pair of `(len(text)`, `len(pattern))` done and the average time was taken. This is to account for different CPU loads and to even out the issues.

4. Printing was disabled so only the time for computation is taken into account.

3.3 Impact of size of `text`

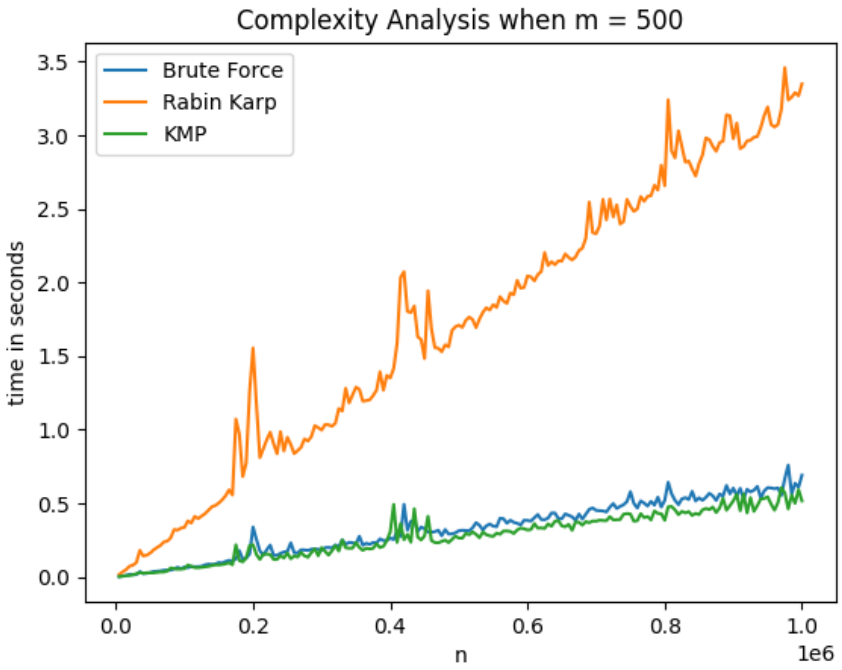


Figure 3.1: Analysis increasing size of n

As size of text increase, time taken increases at a linear rate which is what we expect from complexities since m is fixed.
Knuth Morris Pratt Algorithm is taking the least amount of time as we would expect since its complexity is linear and comparisons of a character happen at most 2 times.
There is surprising result between Rabin-Karp and Brute Force. Brute Force is much more efficient than Rabin-Karp. The reason for this can be the high cost of hashing which the brute force avoids, there might also be some collisions which result in Rabin-Karp doing more computation in the best case.

3.4 Impact of size of pattern

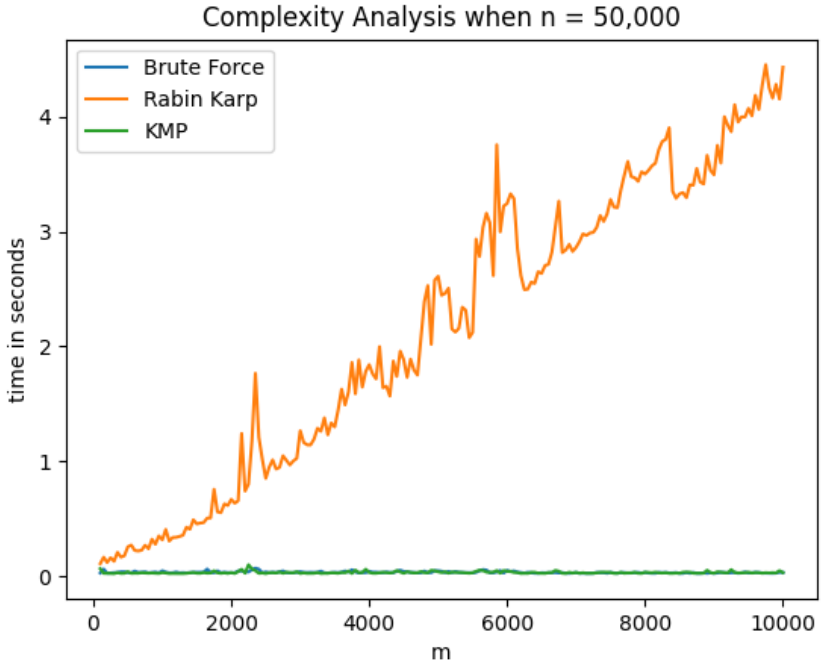


Figure 3.2: Analysis increasing size of m

Knuth Morris Pratt Algorithm takes almost constant as we would expect since $n > m$. Rabin-Karp increases way faster than others. This is because as pattern size increases probability of collisions leading to false positives and hence lot of checking by brute force coupled with cost of hashing, leading to huge growth. Brute force case is strange. This can be because the strings are randomly generated so we don't have to scan entire string to know it is bad. So worse case isn't hit.

3.5 Conclusions

KMP Algorithm seems to have the best performance with Brute Force close behind and maybe good for small strings. Rabin-Karp is the slowest but it still is useful, since we can use it to pattern match for multiple patterns at once saving time. The code for evaluation can be found in the repository.

Bibliography

[1] Georgy Gimel'farb. *String Matching Algorithms*. Tech. rep. URL: http://www-igm.univ-mlv.fr/%7B~%7Dlecroq/string/index.html (page 1).

[2] Donald E Knuthf, James H Morris, and Vaughan R Pratt. *FAST PATTERN MATCHING IN STRINGS**. Tech. rep. 2. 1977 (page 8).