Algorithmics 2017

The Rabin-Karp Algorithm

CSE 5211: Analysis of Algorithms

Dr. William Shoaff

Group 2

Louay Elbiche

Siddhesh Jethe

Zubin Kadva

Table of Contents

1. String searching algorithms

A string matching (or searching) algorithm is a class of string algorithms that returns a position where one or several patterns are found within a larger string or text. Thus, in the pattern matching problem, we are given a string $\boldsymbol{S}$ of length $\boldsymbol{n}$ and a pattern $\boldsymbol{P}$ of length $\boldsymbol{m}$, and want to find whether $\boldsymbol{P} \subseteq \boldsymbol{S}$. [1] That is, $\boldsymbol{P}$ is a substring of $\boldsymbol{S}$. In other words,

$$S[i] = P[0]$$

$$S[i+1] = P[1]$$

...

$$S[i+m+1] = P[m-1]$$

Let $\sum$ be an alphabet. Both patterns and searched text are elements of $\sum$. $\sum$ can be a human alphabet $\sum = \{A, \ldots, Z\}$ or binary $\sum = \{0, 1\}$ or even a DNA alphabet $\sum = \{A, C, G, T\}$

$$If \sum = \{a, b\};\ then\ \boldsymbol{abab}\ is\ a\ string\ over \sum \qquad [2]$$

eg. Given a pattern = XYXYZ

Index	0	1	2	3	4	5	6	7
Text	X	Y	X	Y	X	Y	Z	Y

Figure: String matching example

In Java, this is equivalent to the statement:

```
text.substring(2, 5);
```

The above line returns `XYXYZ`

1.1. Exact string matching algorithms

Exact string matching is used in searching any occurrence of pattern $\boldsymbol{P}$ in a string $\boldsymbol{S}$. These algorithms are applied in biology especially in DNA chaining. Much of data processing in bioinformatics involves recognizing patterns within DNA, RNA or protein sequences. [5]

The problem:

Given a string $\boldsymbol{S} = \boldsymbol{s_1}, \boldsymbol{s_2}, \ldots, \boldsymbol{s_n}$ of characters in some alphabet $\boldsymbol{\Sigma} = \{\boldsymbol{a_1}, \ldots, \boldsymbol{a_K}\}$ and a pattern or query string $\boldsymbol{P} = \{\boldsymbol{p_1}, \ldots, \boldsymbol{p_m}\}$, $\boldsymbol{m} < \boldsymbol{n}$ in the same alphabet, find all occurrences of **P** in **S**.

2. Known algorithms

Various algorithms exist for this purpose. Some are: [1]

- Naïve string searching algorithm
- Rabin-Karp string searching algorithm
- Knuth-Morris-Pratt algorithm
- Boyer-Moore string search algorithm

3. A brute force approach

Also known as the naïve algorithm. The principle of a brute force string matching is quite simple. We check for a match between the first characters of the pattern with the first character of the text. Thus, the idea of the naive solution is just to make a comparison character by character of the text

$$\boldsymbol{S[s \ldots s + m - 1]} \quad \forall \boldsymbol{s} \in \{\boldsymbol{0}, \ldots, \boldsymbol{n - m + 1}\}$$

and the pattern $\boldsymbol{P[0 \ldots m - 1]}$. The naïve algorithm takes as input strings **S** and **P** and checks if

$$\boldsymbol{s_i \ldots s_{i+m} = P} \quad \forall \boldsymbol{i = 1, \ldots, n - m + 1}$$

directly. [3]

```
function NaiveSearch(string, pattern)
      n := length(string)
      m := length(pattern)
      for i = 0 to n - m + 1
            for j = 0 to m
                  if string[i + j] != pattern[j]
                        break
            if j == m
                  return i
      return not found
```

Figure: Naïve string matching pseudocode

3.1. Example execution

Text	F	I	N	D		M	E
Pattern	M	E					

Iteration 1:

Index	0	1	2	3	4	5	6
Text	F	I	N	D		M	E
Pattern	M	E					

Iteration 2:

Index	0	1	2	3	4	5	6
Text	F	I	N	D		M	E
Pattern		M	E				

Iteration 3:

Index	0	1	2	3	4	5	6
Text	F	I	N	D		M	E
Pattern			M	E			

Iteration 4:

Index	0	1	2	3	4	5	6
Text	F	I	N	D		M	E
Pattern				M	E		

Iteration 5:

Index	0	1	2	3	4	5	6
Text	F	I	N	D		M	E
Pattern					M	E	

Iteration 6:

Index	0	1	2	3	4	5	6
Text	F	I	N	D		M	E
Pattern						M	E

Iteration 7:

Index	0	1	2	3	4	5	6
Text	F	I	N	D		M	E
Pattern						M	E

Done! Pattern found at index **5**

3.2. Implementation

```
/* Author: Zubin Kadva
 * Class: Analysis of Algorithms, Spring 2017
 * Project: Naive Search
 */
class NaiveSearch {
    // Returns first index position or -1 if not found
    public static int search(String text, String pattern) {
        int n = text.length();
        int m = pattern.length();
        for (int i = 0; i < n - m + 1; i++) {
            int j;
            for (j = 0; j < m; j++) {
                if (text.toLowerCase().charAt(i + j) !=
                        pattern.toLowerCase().charAt(j)) {
                    break;
                }
            }
            if (j == m) {
                return i;
            }
        }
        return -1;
    }
}
```

3.3. Generating random data

```
private static String generateText(int length) {
    String text = "";
    for (int i = 1; i <= length - 1; i++) {
        text += "a";
        if (i == length - 1) {
            text += "b";
        }
    }
    return text;
}

private static String generatePattern(int length) {
    String pattern = "";
    for (int i = 1; i <= length - 1; i++) {
        pattern += "a";
        if (i == length - 1) {
            pattern += "b";
        }
    }
    return pattern;
}

String text = generateText(100);
String pattern = generatePattern(50);
```

3.4. Analysis

Worst case time complexity

The `NaiveSearch` routine is expressed as:

$$T(n) = \sum_{i=0}^{n-m} \sum_{j=0}^{m-1} (1)$$

$$\therefore T(n) = \sum_{i=0}^{n-m} m$$

$$\therefore T(n) = m\,(n - m + 1)$$

$$\therefore T(n) = mn - m^2 + m$$

$$\therefore T(n) = O\,(mn)$$

If $m = \frac{n}{2}$, then

$$T(n) = \frac{n^2}{2} - \frac{n^2}{4} + \frac{n}{2}$$

$$\therefore T(n) = \frac{2n^2 - n^2 + 2n}{4}$$

$$\therefore T(n) = \frac{n^2 + 2n}{4}$$

$$\therefore T(n) = O\,(n^2)$$

This shows that the naïve algorithm is quadratic in terms of the input string ***S.***

Thus, the complexity is $\boldsymbol{O}\,(\boldsymbol{mn})$**.**

Best case time complexity

The `NaiveSearch` routine is expressed as:

$$T(n) = \sum_{j=0}^{m-1} (1)$$

$$\therefore T(n) = m$$

$$\therefore T(n) = \Omega\,(m)$$

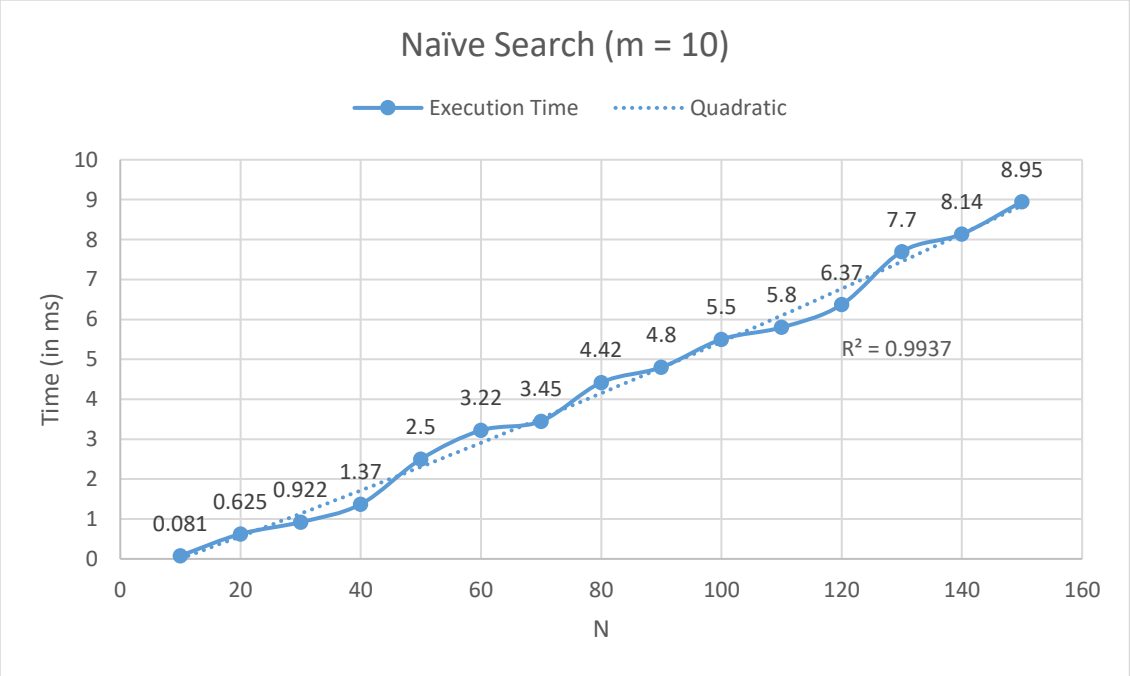
Thus, the complexity is $\boldsymbol{\Omega}\,(\boldsymbol{m})$.

3.5. Graphs

Case 1: Fixed pattern, variable text

N	Time (in ms)
10	0.081
20	0.625
30	0.922
40	1.37
50	2.5
60	3.22
70	3.45
80	4.42
90	4.8
100	5.5
110	5.8
120	6.37
130	7.7
140	8.14
150	8.95

Table: Naïve search execution time from n = 10 to 150 keeping m constant at 10

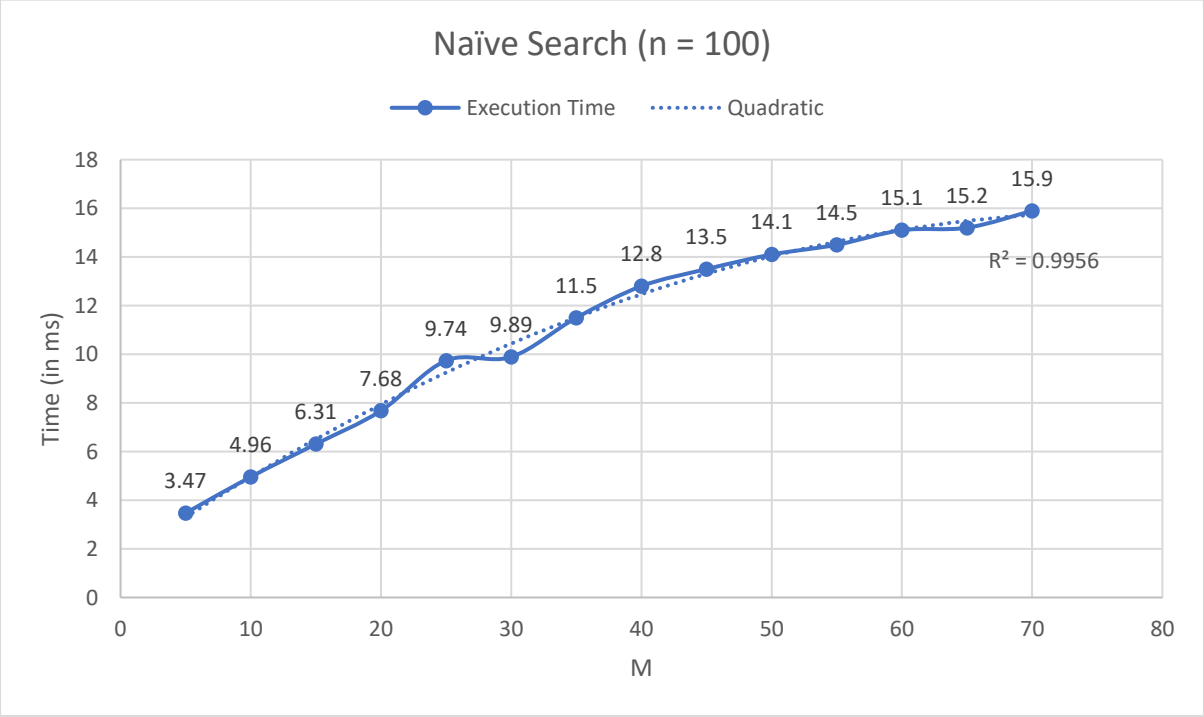


Graph: Scatter plot of the above data with a quadratic trend line

Case 2: Fixed text, variable pattern

M	Time (in ms)
5	3.47
10	4.96
15	6.31
20	7.68
25	9.74
30	9.89
35	11.5
40	12.8
45	13.5
50	14.1
55	14.5
60	15.1
65	15.2
70	15.9

Table: Naïve search execution time from m = 5 to 70 keeping n constant at 100



Graph: Scatter plot of the above data with a quadratic trend line

4. The Rabin-Karp Algorithm

In 1987, Michael O. Rabin and Richard M. Karp came up with the idea of hashing the pattern and checking it against a hashed sub-string from the text. The hash function suggested by Rabin and Karp calculates an integer value.

A hash function is a function which converts every string into a numeric value, called its hash value; for example, we might have `hash("hello") = 5`. The algorithm exploits the fact that if two strings are equal, their hash values are also equal.

Thus, string matching is reduced (almost) to computing the hash value of the search pattern and then looking for substrings of the input string with that hash value.

```
function RabinKarp(string, pattern)
    hpattern := hash(pattern);
    hstring := hash(string)
    for i = 1 to n − m + 1
        if hstring = hpattern
            if string = pattern
                return i
        hstring:= reHash(i, i + m - 1)
    return not found
```

Figure: Rabin-Karp string matching pseudocode

```
function hash(string, length)
    for i = 0 to length
        hash := hash + value(string[i]) * prime^i
    return hash
```

Figure: Generating hash function

```
function reHash(string, length, old, new, hash)
    newHash := hash – value(string[old])
    newHash := newHash / prime
    newHash := newHash + value(string[new]) * prime^(length - 1)
    return newHash
```

Figure: Computing rolling hash function

4.1. Implementation

```
/* Author: Zubin Kadva
 * Class: Analysis of Algorithms, Spring 2017
 * Project: Rabin Karp
 */

class RabinKarp {
    private static final int prime = 101;
    // Returns first index position or -1 if not found
    public static int search(String text, String pattern) {
        int n = text.length();
        int m = pattern.length();
        int pHash = hash(pattern, m - 1);
        int tHash = hash(text, m - 1);
        for (int i = 1; i <= n - m + 1; i++) {
            if (pHash == tHash) {
             if (pattern.equals(text.substring(i - 1, i + m - 1))) {
                    return i - 1;                }
            }
                tHash = reHash(text, m, i - 1, i + m - 1, tHash);
        }
        return -1;
    }
```

```
    private static int hash(String text, int length) {
        int hash = 0;
        for (int i = 0; i <= length; i++) {
            // sum (ASCII * prime ^ index)
            hash += ((int) text.charAt(i)) * Math.pow(prime, i);
        }
        return hash;
    }

    private static int reHash(String text, int length,
            int oldIndex, int newIndex, int hash) {
        // x = oldHash - ASCII
        int newHash = hash - ((int) text.charAt(oldIndex));
        // x = x / prime
        newHash = newHash / prime;
        // x + prime ^ (m-1) * ASCII
        newHash += ((int) text.charAt(newIndex)) *
                Math.pow(prime, (length - 1));
        return newHash;
    }
}
```

5. Using modular arithmetic hash

5.1. Basic plan [7]

A string of length **M** corresponds to an **M-digit base-d** number. To use a hash table of size **Q**, we need a hash function to convert an M-digit base-d number to an **int** value between **0 and Q - 1**. **Modular hashing** provides an answer: *take the remainder when dividing the number by Q*. In practice, we use a random prime Q as large a value as possible while avoiding overflow.

```
 i    0    1    2
      6    4    7      % 23 = 3
```

Figure: Q = 23; pattern = 647

5.2. Computing the hash function

The hash function is computed using **Horner's method.** For each digit in the number, we multiply by **d**, add the digit, and take the remainder when divided by **Q**. The cost for the substring search would be a multiplication, addition, and remainder calculation for each text character, for a total of **NM** operations in the worst case, no improvement over the brute-force method. It also computes the value of $\boldsymbol{d^{m-1} \bmod Q}$ in the variable **h**.

The Rabin-Karp method is based on efficiently computing the hash function for position **i + 1** in the text, given its value for position **i**. Using the notation $\boldsymbol{t_i}$ for `txt.charAt(i)`, the number corresponding to the **M-character** substring of text that starts at position **i** is:

$$x_i = t_i\, d^{m-1} + t_{i+1}\, d^{m-2} + \cdots + t_{i+m-1}\, d^0$$

and we can assume that we know the value of $\boldsymbol{h(x_i) = x_i \bmod Q}$. Shifting one position right in the text corresponds to replacing x_i by:

$$x_{i+1} = (x_i - t_i\, d^{m-1})\, d + t_{i+m}.$$

We subtract off the leading digit, multiply by **d**, then add the trailing digit.

i	**0**	**1**	**2**	**3**	**4**	**…**
current value	6	3	2	1	7	
new value		3	2	1	7	
		3	2	1	*current value*	
	-	3	0	0		
			2	1	*subtract leading digit*	
	*		1	0	*multiply by radix*	
		2	1	0		
	+			7	*add trailing digit*	
		2	1	7	*new value*	

Figure: Key computation; radix = 10

5.3. A modified algorithm [8]

```
function rabin-karp-matcher(T, P, d, q)
    n := T.length
    m := P.length
    h := d^(m−1) mod q
    p := 0
    t_0 := 0
    for i = 1 to m
        p := (d * p + P[i]) mod q
        t_0 := (d * t_0 + T[i]) mod q
    for s = 0 to n - m
        if p == t_s
            if P[1 .. m] == T[s + 1 .. s + m]
                return s
        if s < n - m
            t_{s+1} = (d ∗ (t_s − T[s + 1] ∗ h) + T[s + m + 1]) mod q
    return not found
```

Figure: Rabin Karp using modular hash function

5.4. Monte Carlo vs Las Vegas correctness

After finding a hash value for an **M-character** substring of the text that matches the pattern hash value, we compare those characters with the pattern to ensure that we have a true match, not just a hash collision. We do not do that test because using it requires backup in the text string. Instead, we make the hash table "size" **Q** as large as we wish, since we are not actually building a hash table, just testing for a collision with one key, our pattern. We will use a long value greater than $\mathbf{10^{20}}$, making the probability that a random key hashes to the same value as our pattern less than $\mathbf{10^{-20}}$, an exceedingly small value. If that value is not small enough for you, you could run the algorithms again to get a probability of failure of less than $\mathbf{10^{-40}}$. This algorithm is an early and famous example of a **Monte Carlo** algorithm that has a guaranteed completion time but fails to output a correct answer with a small probability. The alternative method of checking for a match could be slow but is guaranteed to be correct. Such an algorithm is known as a **Las Vegas** algorithm.

5.5. A complete example

Given

text = 31246849621378;

pattern = 68496;

q = 23;

d = 10;

Calculations

m = `length(pattern)` = 5;

h = $d^{m-1} \ mod \ q = 10^{5-1} \ mod \ 23 = 18$;

`hash(pattern)` = 2

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
	3	1	2	4	6	8	4	9	6	2	1	3	7	8
0	3	% 23 = 3												
1	3	1	% 23 = (3 * 10 + 1) % 23 = 8											
2	3	1	2	% 23 = (8 * 10 + 2) % 23 = 13										
3	3	1	2	4	% 23 = (13 * 10 + 4) % 23 = 19									
4	3	1	2	4	6	% 23 = (19 * 10 + 6) % 23= 12								
5		1	2	4	6	8	% 23 = ((12 + 3 * (23 - 18)) * 10 + 8) % 23 = 2							
6			2	4	6	8	4	% 23 = ((2 + 1 * (23 - 18)) * 10 + 4) % 23 = 5						
7				4	6	8	4	9	% 23 = ((5 + 2 * (23 - 18)) * 10 + 9) % 23 = 21					
8					6	8	4	9	6	% 23 = ((21 + 4 * (23 - 18)) * 10 + 6) % 23 = 2				

Figure: Rabin Karp search example

return i – m + 1 = 8 – 5 + 1 = **4**

Done! The pattern is found at position **4.**

6. Implementation

```
/* Author: Zubin Kadva, Siddhesh Jethe
 * Class: Analysis of Algorithms, Spring 2017
 * Project: Rabin Karp modular hashing
 * Reference:
http://algs4.cs.princeton.edu/53substring/RabinKarp.java.html
 */


import java.math.BigInteger;
import java.security.SecureRandom;


public class RabinKarpMatcher {
    static long q = BigInteger.probablePrime(31, new
SecureRandom()).longValue();
    static int d = 256;
    private static long hash(String key, int m) {
        long h = 0;
        for (int i = 0; i < m; i++) {
            h = (d * h + key.charAt(i)) % q;
        }
        return h;
    }
    // Las Vegas version: does pat[] match txt[i..i - m + 1] ?
    private static boolean check(String txt, int i, int m, String
pat) {
        for (int j = 0; j < m; j++) {
            if (pat.charAt(j) != txt.charAt(i + j)) {
                return false;
            }
        }
        return true;
    }
```

```
    // Returns first index position or -1 if not found
    public static int search(String text, String pattern) {
        int n = text.length();
        int m = pattern.length();
        // h = d ^ (m-1) mod q
        long h = BigInteger.valueOf(d).modPow(BigInteger.
                    valueOf(m-1), BigInteger.valueOf(q)).longValue();
        long p = hash(pattern, m);
        long t = hash(text, m);

        // check for match at offset 0
        if ((p == t) && check(text, 0, m, pattern))) {
            return 0;
        }
        for (int i = m; i < n; i++) {
       // Remove leading digit, add trailing digit, check for match.
            t = (t + q - h * text.charAt(i - m) % q) % q;
            t = (t * d + text.charAt(i)) % q;
            // Match
            int offset = i - m + 1;
            if ((p == t) && check(text, offset, m, pattern)) {
                return offset;
            }
        }
        return -1;
    }
}
```

7. Analysis

Worst case time complexity

The `search` routine is expressed as:

$$T(n) = \sum_{i=0}^{m-1}(1) + \sum_{i=0}^{n-m-1}\sum_{j=0}^{m-1}(1)$$

$$\therefore T(n) = m + \sum_{i=0}^{n-m-1} m$$

$$\therefore T(n) = m + m\,(n-m)$$

$$\therefore T(n) = m + mn - m^2$$

$$\therefore T(n) = O\,(mn)$$

Thus, the complexity is $\boldsymbol{O\,(mn)}$.

Best case time complexity

The `search` routine is expressed as:

$$T(n) = \sum_{i=0}^{m-1}(1) + \sum_{i=0}^{n-1}(1)$$

$$\therefore T(n) = m + n$$

$$\therefore T(n) = \Omega\ (m+n)$$

Thus, the complexity is $\boldsymbol{\Omega\,(m+n)}$.

The average time complexity of the algorithm is also $\boldsymbol{m+n}$ and follows a similar proof. Therefore, the average case time complexity is $\boldsymbol{\theta\,(m+n)}$.

Space complexity

The hash function is computed until the length of the pattern. This is given as:

$$length(pattern) = m$$

Thus, the complexity is $\boldsymbol{O(m)}$.

In summary,

Complexities ↓ / Algorithm →		Naïve Search	Rabin Karp
Performance	Best	$\Omega\ (m)$	$\Omega\ (m+n)$
	Average	$\theta\ (m+n)$	$\theta\ (m+n)$
	Worst	$O\ (mn)$	$O\ (mn)$
Space		$O\ (1)$	$O\ (m)$

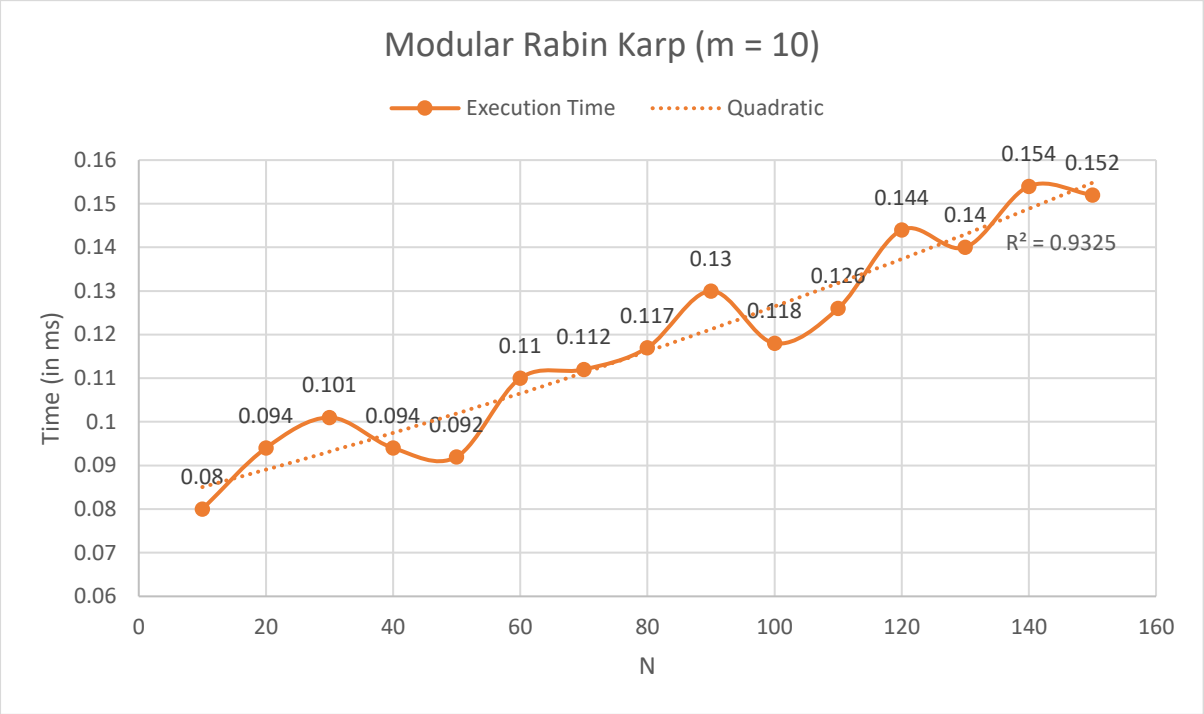
8. Graphs

8.1. Execution Time

Case 1: Fixed pattern, variable text

N	Time (in ms)
10	0.08
20	0.094
30	0.101
40	0.094
50	0.092
60	0.11
70	0.112
80	0.117
90	0.13
100	0.118
110	0.126
120	0.144
130	0.14
140	0.154
150	0.152

Table: Rabin Karp (modular) search execution time from n = 10 to 150 keeping m constant at 10

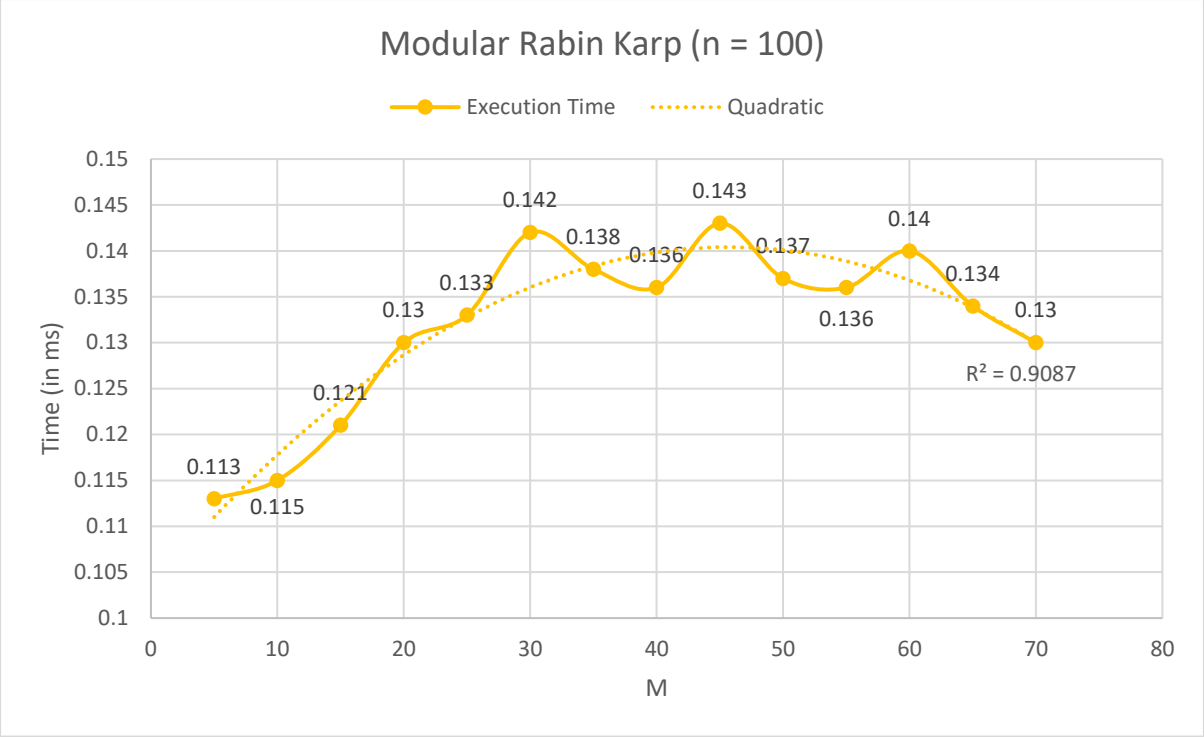


Graph: Scatter plot of the above data with a quadratic trend line

Case 2: Fixed text, variable pattern

M	Time (in ms)
5	0.767
10	0.908
15	1
20	1.1
25	1.3
30	1.5
35	1.3
40	1.72
45	2
50	2.11
55	2.29
60	2.1
65	2.1
70	2

Table: Rabin Karp (modular) search execution time from m = 5 to 70 keeping n constant at 100

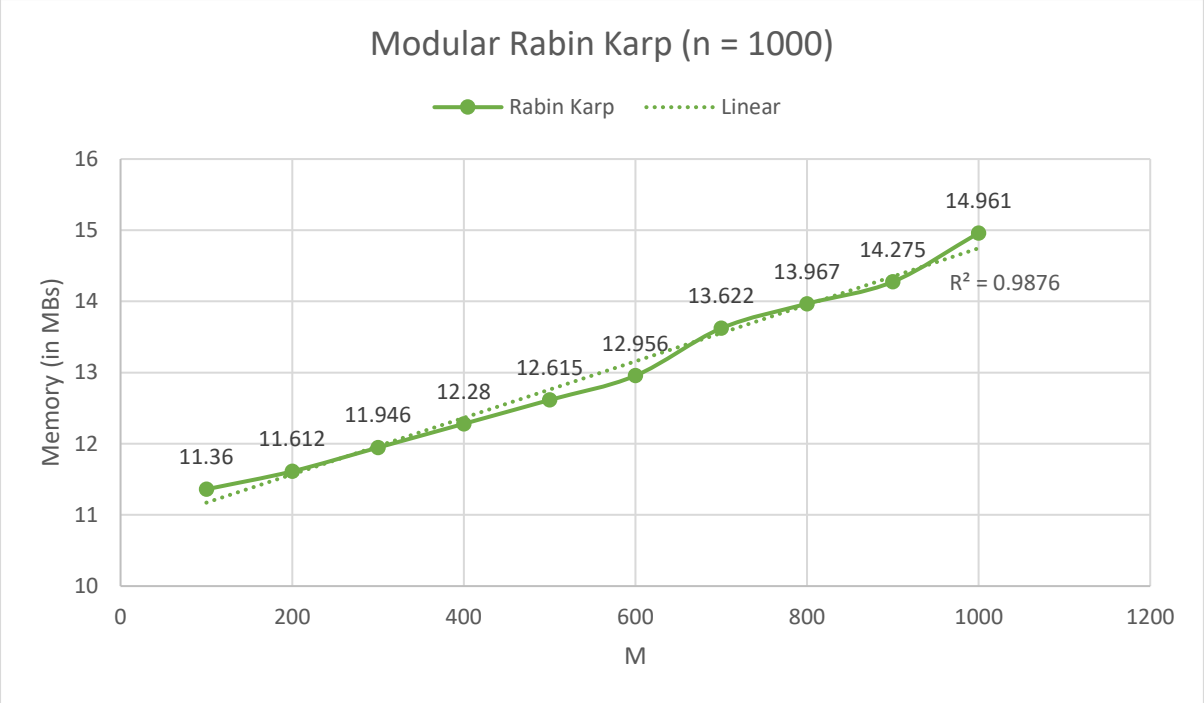


Graph: Scatter plot of the above data with a quadratic trend line

8.2. Memory consumption

M	Memory Consumption
100	11.36
200	11.612
300	11.946
400	12.28
500	12.615
600	12.956
700	13.622
800	13.967
900	14.275
1000	14.961

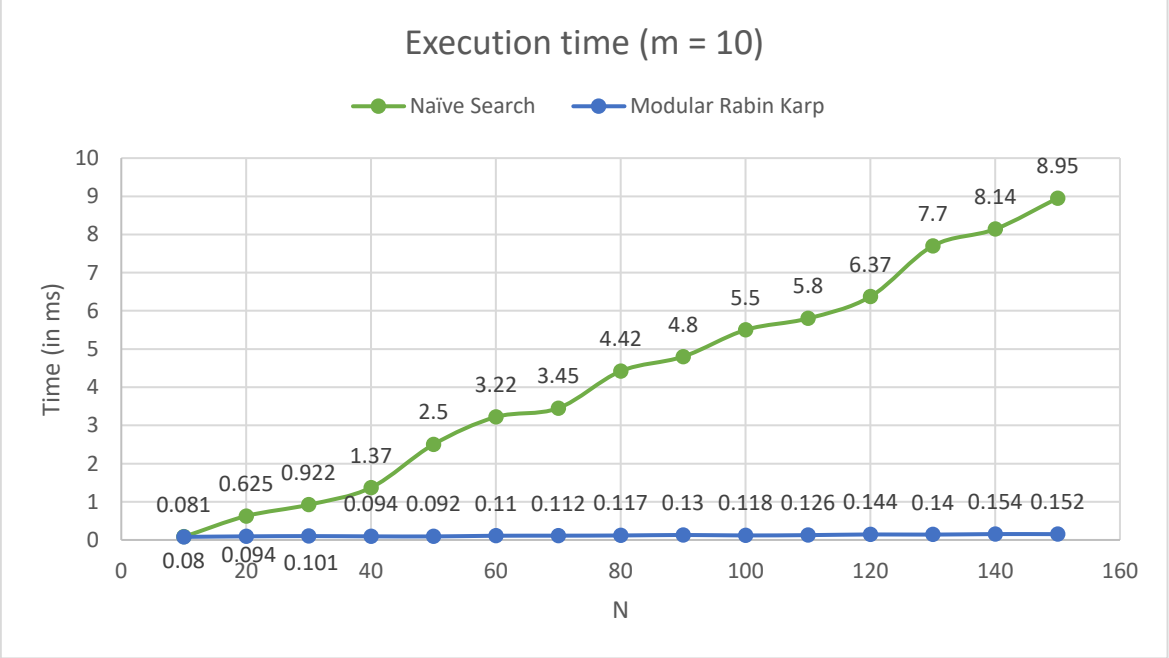
Table: Rabin Karp (modular) search memory consumption from m = 100 to 1000 keeping n constant at 1000



Graph: Scatter plot of the above data with a linear trend line

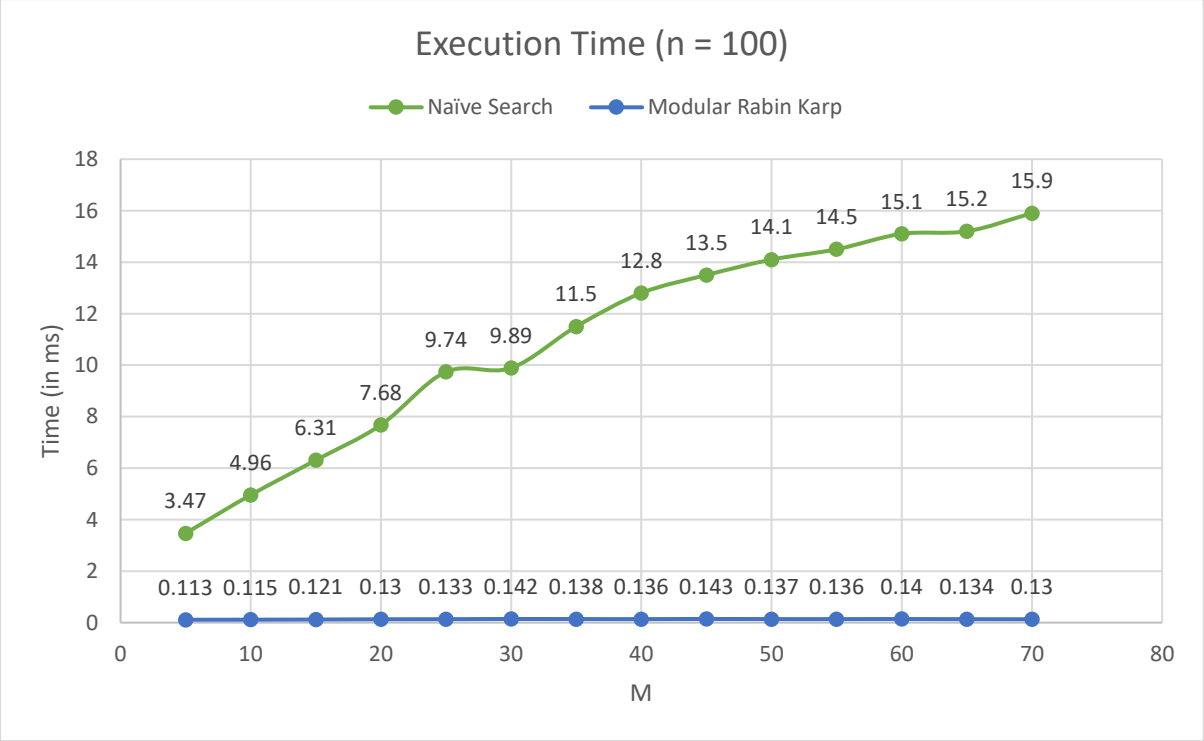
9. Comparisons

Case 1: Fixed pattern, variable text



Graph: Scatter comparing Naïve Search and Rabin Karp (modular) Search

Case 2: Fixed text, variable pattern



Graph: Scatter comparing Naïve Search and Rabin Karp (modular) Search

10. Bioinformatics

10.1. Background [5]

A DNA string consists of nucleotides. Each nucleotide is an alphabet $\boldsymbol{\Sigma} = \{\mathbf{A}, \mathbf{C}, \mathbf{G}, \mathbf{T}\}$. Certain nucleotides or amino acid sequences have properties that are known to biologists. Eg. The sequence **ATG** must be present at the beginning of a gene. A **conserved DNA** is a sequence of nucleotides that is found in the DNA of multiple species. A **primer** is a conserved DNA sequence used in **Polymerase Chain Reaction (PCR)** to identify the location of the DNA sequence that will be amplified.

eg. Consider the following sequence of nucleotides:

G C T A C T A T T T T T C A T

When read in forward frame 0, this sequence encodes the following sequence of amino acids:

A T I F H

Because of redundancy in genetic code, the following sequence of nucleotides will produce the same sequence of amino acids, where lower case letters show the differences with the original string:

GCT ACT ATT TTT CAT

GCc ACc ATT TTa CAc

GCc ACc ATc TTg CAT

GCT ACT ATc TTa CAc

GCg ACc ATa TTc CAc

A T I F H

10.2. Example

Consider an alphabet $\boldsymbol{\Sigma} = \{\mathbf{A}, \mathbf{C}, \mathbf{G}, \mathbf{T}\}$. We map characters in base 4 system as:

Character	Digit
A	0
C	1
G	2
T	3

Figure: Nucleotide alphabet mapping

Consider a string S = "AGGCTATTA" and a pattern "TATT". The substrings are:

A	G	G	C	T	A	T	T	A
A	G	G	C	T	A	T	T	A
A	G	G	C	T	A	T	T	A
A	G	G	C	T	A	T	T	A
A	G	G	C	T	A	T	T	A
A	G	G	C	T	A	T	T	A

Figure: Substrings in given DNA sequence

Given a pattern $P = p_1 \dots p_m$, its hash value p in base K can be computed as:

$$p = p_m + K(p_{m-1} + K(p_{m-2} + \cdots + K(p_2 + Kp_1)) \dots)$$

Given a string $S = s_1 \dots s_n$ where $n > m$, the numeric value of $S[1, m]$, t_0 can be computed as:

$$t_0 = s_m + K(s_{m-1} + K(s_{m-2} + \cdots + K(s_2 + Ks_1)) \dots)$$

The hash can be rolled over as:

$$t_i = K(t_{i-1} - K^{m-1}s_i) + s_{i+m}$$

Thus, anytime $p = t_i$ for some i, then $i + 1$ is the position of such a match.

We can pre compute the value of K^{m-1}as this number is a constant.

We compute hash of the pattern as:

TATT = $p = 3 + 4(3 + 4(0 + 4 * 3)) = 207$

We compute hash of the sub strings as:

A	G	G	C	$t_0 = 1 + 4(2 + 4(2 + 4 * 0)) = 41$
G	G	C	T	$t_1 = 4(41 - 4^3 * 0) + 3 = 167$
G	C	T	A	$t_2 = 4(167 - 4^3 * 2) + 0 = 156$
C	T	A	T	$t_3 = 4(156 - 4^3 * 2) + 3 = 115$
T	A	T	T	$t_4 = 4(115 - 4^3 * 1) + 3 = 207$
A	T	T	A	$t_5 = 4(207 - 4^3 * 3) + 0 = 60$

Figure: Hash mappings for given DNA sequence

$p = t_4$. Thus, the algorithm returns $4 + 1 =$ **5** as the position of the match.

10.3. DNA codons

Amino acids found in proteins are denoted by a single-letter code. These are stored in protein databases. The DNA codons representing each amino acid are also listed. All 64 possible 3-letter combinations of the nucleotides A, C, G, T are used either to encode one of these amino acids or as one of the three stop codons that signals the end of a sequence. Since most amino acids have multiple codons, some DNA sequences might denote the same protein sequence.

Amino Acid	SLC	DNA codons
Isoleucine	I	ATT, ATC, ATA
Leucine	L	CTT, CTC, CTA, CTG, TTA, TTG
Valine	V	GTT, GTC, GTA, GTG
Phenylalanine	F	TTT, TTC
Methionine	M	ATG
Cysteine	C	TGT, TGC
Alanine	A	GCT, GCC, GCA, GCG
Glycine	G	GGT, GGC, GGA, GGG
Proline	P	CCT, CCC, CCA, CCG
Threonine	T	ACT, ACC, ACA, ACG
Serine	S	TCT, TCC, TCA, TCG, AGT, AGC
Tyrosine	Y	TAT, TAC
Tryptophan	W	TGG
Glutamine	Q	CAA, CAG
Asparagine	N	AAT, AAC
Histidine	H	CAT, CAC
Glutamic acid	E	GAA, GAG
Aspartic acid	D	GAT, GAC
Lysine	K	AAA, AAG
Arginine	R	CGT, CGC, CGA, CGG, AGA, AGG
Stop codons	Stop	TAA, TAG, TGA

Figure: 20 amino acids, their single-letter database codes (SLC), and their corresponding DNA codons [9]

11. Choosing a prime

The best value for $\boldsymbol{q}$ is a prime number that is smaller than $2^B - 1$ where B is the number of bits in a machine word on a given processor architecture.

For **16-bit** words, we can take

$$q < 2^{15} - 1$$

$$\therefore q < 32767$$

For **32-bit** words, we can take

$$q < 2^{31} - 1$$

$$\therefore q < 2147483647$$

12. References and tools

[1] String searching algorithms from https://en.wikipedia.org/wiki/String_searching_algorithm

[2] Akhtar Rasool, Amrita Tiwari, Gunjan Singla, Nilay Khare, *"String Matching Methodologies: A Comparative Analysis"*, (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 3 (2), 2012

[3] Marc GOU, *"Algorithms for String matching"*, July 30, 2014

[4] Ashish Prosad Gope, Rabi Narayanan Behra, *"A Novel Pattern Matching Algorithm in Genome Sequence Analysis"*, (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 5 (4), 2014

[5] Alexander Dekhtyar, "*String Matching Problems and Bioinformatics"*, Bioinformatics Algorithms

[6] Rabin-Karp algorithm from https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm

[7] Robert Sedgewick, Kevin Wayne-Algorithms, "*Essential Information about Algorithms and Data Structures 4th Edition*", Addison-Wesley, 2011

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "*Introduction to Algorithms Third Edition"*, The MIT Press, 2009

[9] DNA codons from http://www.cbs.dtu.dk/courses/27619/codon.html

The NetBeans profiler for profiling the performance of the algorithms.

Microsoft Excel for plotting graphs of the gathered data.