MARMARA UNIVERSITY – FACULTY OF ENGINEERING

CSE 2046/2246 HOMEWORK #1 REPORT

Name, Surname: Hasan Şenyurt – Mert Akbal – Hakan Kenar – Ahsen Yağmur Kahyaoğlu

Student ID: 150120531 – 150119703 – 150119675 – 150119788

Department: Computer Engineering

Input Files

There are 7 input files in total as HTML file. 3 of them are generated with random Latin words such as Lorem ipsum etc. These files are larger than 1MB. Other 3 of them are generated with random binary values. These files are larger than 1MB also. 6 files that are mentioned above are different from each other. Last input file is given by lecturer which includes ‘AT_THAT’ pattern inside.

The reason that we generated Latin words in input files is they are meaningless and they are not in order. It means algorithms will not be able to find patterns easy. Lengths of them are very long and it is hard to find the pattern with human eye. Also, binary inputs are very complex for algorithms because there are just two numbers inside of very large text. So, running time will increase because there will be overlaps, and shift values will be small. There will be lots of comparisons. Comparison count and running time will increase.

Example of Latin input HTML file (it continues, it is a section):

```
<HTML> <BODY>
Nulla tincidunt hendrerit enim, non fringilla metus iaculis id. Fusce finibus, arcu quis mattis eleifend, augue odio lobortis tellus, eget sodale

Fusce non sem ante. Praesent aliquam nunc sit amet nisl condimentum tempus. Maecenas tincidunt volutpat orci sed ultricies. Proin elementum place

Etiam laoreet mattis tincidunt. Donec accumsan metus ac eleifend imperdiet. Sed laoreet turpis id tincidunt ultricies. In porttitor sollicitudin

Morbi vitae augue ultrices, tempus dui vitae, rutrum nibh. Fusce commodo faucibus mattis. Morbi fermentum congue erat, ac posuere diam. Sed sit a

Vivamus orci dui, rutrum ac hendrerit sit amet, iaculis vel enim. Fusce venenatis tempor mattis. Ut aliquam consectetur nibh id lacinia. Aliquam

Proin quis felis scelerisque, ultrices erat nec, elementum metus. Curabitur metus nisi, viverra vehicula fringilla nec, tincidunt eget massa. Mae

Praesent pretium lacus mollis malesuada fringilla. Nam sed ex id neque vehicula lobortis quis consequat mi. Fusce magna sapien, scelerisque id fa

Integer pulvinar turpis ut laoreet dapibus. Etiam id dapibus leo. Proin pretium lectus a nisl iaculis semper. Vestibulum porttitor non augue in t
```

Figure-1: HTML Latin Input File Section

Example of Binary input file it continues, it is a section):

```
<HTML> <BODY>
110000100010011111101011111110001001010000100010010010111011001101011101100010101011100011100101 1010
010000001000110101111101101011101110111010011001100010110101100010111011000010000010110010001011 10111
011010000001101100100010000101001100011011001010001010100100101110101001111111101111110100111111 11101
101111000101001110101001010000000101000010000110110110001011011111000000010100110001100000110011 011101
100001111110101010100101101000100111111100101000111000000111001111110110001001111110101010001011 0000
000100001101101100110100010100001001110110110010101111111000001110011111001110111100100010000000 100001
101100011011000010010011001010111110000000101110111101101010000100011111101100111000101111011111 1010
101001000010111101001101011011010111110001110010010001010001101000011100110010110010110111100110 1100
100000011000100100010110001100001001100010101100100100111011011000001110011011000011000000000101 00110
001100110001100000000000011111100000001010110001010100111011110101111000111010011101111001011111 110011
110111101011100001011111111101000011101111001101001100110111100101011111100000100000011001010010 0110
000111101001111100001110001001100111111011100110011100010111111000001100111000000000110010001010 011001
100000000010010101000001000111001001000000010010110111001001001010111000111011110011100110110000 1010
101101111010010110001001110101110011001100111010000000100011101110011011100010100010001001110101 00101
101100100111011111101000000100110000111110011100011101101101100100011110111001101101011010101111 011100
```

Figure-2: HTML Binary Input File Section

Code

Brute-force Algorithm

```
def brute_force_search(text, pattern):
    #To measure runtime
    start_time = time.time()
    n = len(text)
    m = len(pattern)
    comparisons = 0
    matches = []
    found_index_start = []
    found_index_end = []
    match_count = 0

    #Check every letter of the pattern starting from 0th index and shift by one.
    for i in range(n - m + 1):
        j = 0

        while j < m and text[i + j] == pattern[j]:
            j += 1
            comparisons += 1

        #if comparison count in the while loop equals to pattern lenght increase match count by one.
        if j == m:
            matches.append(i)
            found_index_start.append(i)
            found_index_end.append(i+len(pattern)-1)
            comparisons += 1
            match_count += 1
        comparisons += 1
    brute_force_time = (time.time() - start_time) * 1000
    markText(found_index_start,found_index_end,text,match_count,comparisons,"bruteforce")
    return brute_force_time, comparisons
```

Figure-3: Code of Brute-Force Algorithm

In brute force algorithm we start checking starting indexes of pattern and text and if they are equal algorithm checks that if they are equal after the starting index until keys do not match or match occurs, then increasing the text index by one and doing the same operations until text index reaches end of the text. By calling the function markText with predetermined boundaries of the text and name of the algorithm we do mark the corresponding indexes of the text. Returning runtime of brute force algorithm and comparison count.

Boyer-Moore Algorithm

```python
def boyerMoore(pattern, text, badChar, goodSuffix,patternIndex, textIndex):

    start_time = time.time()
    i = patternIndex -1 #i = lenght of pattern - 1
    never_match = True #At the beginning there is not one match. If there is one we will use this later.
    found_index_start = []
    found_index_end = []
    comparison_count = 0
    match_count = 0

    #While index of text is less than or equal to lenght of text.
    while i <= textIndex -1:
        temp_i = i
        j = len(pattern) - 1
        any_match = False

        shift = 0

        #While chars of pattern and text is equal and their indexes are more than zero.
        while pattern[j] == text[i] and j >= 0 and i >= 0:

            any_match = True
            i -= 1
            j -= 1


        #If there is complete match of pattern to text.
        if(j==-1):

            found_index_start.append(temp_i +1- len(pattern)) #Noting start of the match index to use later.
            found_index_end.append(temp_i) #Noting end of the match index to use later.
            never_match = False #In this if condition there is complete match so never_match becomes false.
            shift = 1 #To check overlapping matches we shift text index by 1.
            i = temp_i + shift
            comparison_count += len(pattern) #Since there is complete match our comparison_count will be equal to lenght of pattern.
            match_count += 1
```

Figure-4: Code of Boyer-Moore Algorithm

```python
            #When there is partial match we do shift by the first match letter's value in bad symbol table.
            elif(any_match):

                bad_char_shift = badChar[ord(text[i])] - (temp_i - i)
                good_suffix_shift = goodSuffix[temp_i-i]
                shift = 0
                #Compare the shift values of bad symbol table and good suffix table get the greater one.
                if bad_char_shift >= good_suffix_shift:
                    shift = bad_char_shift
                else:
                    shift = good_suffix_shift

                comparison_count += temp_i - i
                i = temp_i + shift

            #When there is not any match we do shift by the lenght of the pattern due to bad symbol table.
            else:
                #shift = length of pattern
                shift = badChar[ord(text[temp_i])]
                i = temp_i + shift
                comparison_count += 1

    #When pattern is not found in text.
    if(never_match == True):
        print("Pattern not found in text!")
    boyer_moore_time = (time.time() - start_time) * 1000
    markText(found_index_start,found_index_end,text,match_count,comparison_count,"boyermoore")
    return boyer_moore_time, comparison_count
```

Figure-5: Code of Boyer-Moore Algorithm

In Boyer Moore algorithm we start checking the chars from right to left by decreasing index by one until a mismatch occurs or match occurs. In case of mismatch shift index of text by length of the pattern. In case of match shift index of text by one due to possibility of overlapping. In case of partial match shift index of text by maximum of bad symbol table and good suffix table at the index of text which mismatch occurs. By calling the function markText with predetermined boundaries of the text and name of the algorithm we do mark the corresponding indexes of the text. Returning runtime of Boyer Moore algorithm and comparison count.

Horspool Algorithm

```python
def horspool(pattern, text, badChar, patternIndex, textIndex):
    start_time = time.time()
    i = patternIndex -1 #i = lenght of pattern - 1
    never_match = True  #At the beginning there is not one match. If there is one we will use this later.
    found_index_start = []
    found_index_end = []
    comparison_count = 0
    match_count = 0

    #While index of text is less than or equal to lenght of text.
    while i <= textIndex -1:
        temp_i = i
        j = len(pattern) - 1
        any_match = False

        shift = 0

        #While chars of pattern and text is equal and their indexes are more than zero.
        while pattern[j] == text[i] and j >= 0 and i >= 0:

            any_match = True
            i -= 1
            j -= 1

        #If there is complete match of pattern to text.
        if(j==-1):

            found_index_start.append(temp_i +1- len(pattern)) #Noting start of the match index to use later.
            found_index_end.append(temp_i) #Noting end of the match index to use later.
            never_match = False #In this if condition there is complete match so never_match becomes false.
            shift = 1   #To check overlapping matches we shift text index by 1.
            i = temp_i + shift
            comparison_count += len(pattern) #Since there is complete match our comparison_count will be equal to lenght of pattern.
            match_count += 1
```

Figure-6: Code of Horspool Algorithm

```python
        #When there is partial match we do shift by the first match letter's value in bad symbol table.
        elif(any_match):
            #shift = first matches' bad symbol value
            shift = badChar[ord(text[temp_i])]
            comparison_count += temp_i - i
            i = temp_i + shift


        #When there is not any match we do shift by the lenght of the pattern due to bad symbol table.
        else:
            #shift = length of pattern
            shift = badChar[ord(text[temp_i])]
            i = temp_i + shift
            comparison_count += 1


    #When pattern is not found in text.
    if(never_match == True):
        print("Pattern not found in text!")
    horspool_time = (time.time() - start_time) * 1000
    markText(found_index_start,found_index_end,text,match_count,comparison_count,"horspool")

    return horspool_time, comparison_count
```

Figure-7: Code of Horspool Algorithm

In Horspool algorithm we start checking the chars from right to left. Increasing the text index by one until text index is equals to text length. While key of pattern equals to key of text decrease both indexes by one if a complete match occurs shift text index by one and keep doing the same. If there is partial match of some keys take the starting index of that iterations text index and find its bad symbol value and shift by that value. If there is not any partial match, then shift by length of the pattern. By calling the function markText with predetermined boundaries of the text and name of the algorithm we do mark the corresponding indexes of the text. Returning runtime of Horspool algorithm and comparison count.

Results of these three algorithms can be seen in ‘brute_force_result.html’, ‘horspool_result.html’ and ‘boyermoore_result.html’ files.

Analysis of Experiment

<u>First experiment</u> is searching '1000110' pattern in 'Binary.html'. Results of three algorithms are below. Bad Symbol Table, good suffix table, and detailed results of three algorithms are printed.

```
Bad Symbol Table: {'\x00': 7, '\x01': 7, '\x02': 7, '\x03': 7, '\x04': 7, '\x05': 7, '\x06': 7, '\x07': 7, '\x08': 7, '\t': 7, '\n': 7, '\x0b': 7, '\x0c': 7,
'\r': 7, '\x0e': 7, '\x0f': 7, '\x10': 7, '\x11': 7, '\x12': 7, '\x13': 7, '\x14': 7, '\x15': 7, '\x16': 7, '\x17': 7, '\x18': 7, '\x19': 7, '\x1a': 7, '\x1b'
: 7, '\x1c': 7, '\x1d': 7, '\x1e': 7, '\x1f': 7, ' ': 7, '!': 7, '"': 7, '#': 7, '$': 7, '%': 7, '&': 7, "'": 7, '(': 7, ')': 7, '*': 7, '+': 7, ',': 7, '-':
7, '.': 7, '/': 7, '0': 3, '1': 1, '2': 7, '3': 7, '4': 7, '5': 7, '6': 7, '7': 7, '8': 7, '9': 7, ':': 7, ';': 7, '<': 7, '=': 7, '>': 7, '?': 7, '@': 7, 'A'
: 7, 'B': 7, 'C': 7, 'D': 7, 'E': 7, 'F': 7, 'G': 7, 'H': 7, 'I': 7, 'J': 7, 'K': 7, 'L': 7, 'M': 7, 'N': 7, 'O': 7, 'P': 7, 'Q': 7, 'R': 7, 'S': 7, 'T': 7, '
U': 7, 'V': 7, 'W': 7, 'X': 7, 'Y': 7, 'Z': 7, '[': 7, '\\': 7, ']': 7, '^': 7, '_': 7, '`': 7, 'a': 7, 'b': 7, 'c': 7, 'd': 7, 'e': 7, 'f': 7, 'g': 7, 'h': 7
, 'i': 7, 'j': 7, 'k': 7, 'l': 7, 'm': 7, 'n': 7, 'o': 7, 'p': 7, 'q': 7, 'r': 7, 's': 7, 't': 7, 'u': 7, 'v': 7, 'w': 7, 'x': 7, 'y': 7, 'z': 7, '{': 7, '|':
 7, '}': 7, '~': 7, '\x7f': 7, '\x80': 7, '\x81': 7, '\x82': 7, '\x83': 7, '\x84': 7, '\x85': 7, '\x86': 7, '\x87': 7, '\x88': 7, '\x89': 7, '\x8a': 7, '\x8b'
: 7, '\x8c': 7, '\x8d': 7, '\x8e': 7, '\x8f': 7, '\x90': 7, '\x91': 7, '\x92': 7, '\x93': 7, '\x94': 7, '\x95': 7, '\x96': 7, '\x97': 7, '\x98': 7, '\x99': 7,
 '\x9a': 7, '\x9b': 7, '\x9c': 7, '\x9d': 7, '\x9e': 7, '\x9f': 7, '\xa0': 7, '¡': 7, '¢': 7, '£': 7, '¤': 7, '¥': 7, '¦': 7, '§': 7, '¨': 7, '©': 7, 'ª': 7,
'«': 7, '¬': 7, '\xad': 7, '®': 7, '¯': 7, '°': 7, '±': 7, '²': 7, '³': 7, '´': 7, 'µ': 7, '¶': 7, '·': 7, '¸': 7, '¹': 7, 'º': 7, '»': 7, '¼': 7, '½': 7, '¾'
: 7, '¿': 7, 'À': 7, 'Á': 7, 'Â': 7, 'Ã': 7, 'Ä': 7, 'Å': 7, 'Æ': 7, 'Ç': 7, 'È': 7, 'É': 7, 'Ê': 7, 'Ë': 7, 'Ì': 7, 'Í': 7, 'Î': 7, 'Ï': 7, 'Ð': 7, 'Ñ': 7, '
Ò': 7, 'Ó': 7, 'Ô': 7, 'Õ': 7, 'Ö': 7, '×': 7, 'Ø': 7, 'Ù': 7, 'Ú': 7, 'Û': 7, 'Ü': 7, 'Ý': 7, 'Þ': 7, 'ß': 7, 'à': 7, 'á': 7, 'â': 7, 'ã': 7, 'ä': 7, 'å': 7,
 'æ': 7, 'ç': 7, 'è': 7, 'é': 7, 'ê': 7, 'ë': 7, 'ì': 7, 'í': 7, 'î': 7, 'ï': 7, 'ð': 7, 'ñ': 7, 'ò': 7, 'ó': 7, 'ô': 7, 'õ': 7, 'ö': 7, '÷': 7, 'ø': 7, 'ù':
7, 'ú': 7, 'û': 7, 'ü': 7, 'ý': 7, 'þ': 7, 'ÿ': 7}

Good Suffix Table: {'0': 3, '10': 5, '110': 5, '0110': 5, '00110': 5, '000110': 5, '1000110': 5}

----------BRUTE-FORCE ALGORITHM:-----------
Pattern: 1000110
Match Count: 8171
Comparison Count: 2211094
Brute Force Running Time: 330.2652835845947 miliseconds.

----------HORSPOOL ALGORITHM:-----------
Pattern: 1000110
Match Count: 8171
Comparison Count: 1000960
Horspool Running Time: 240.0050163269043 miliseconds.

------------------------------------------------------------

----------BOYER MOORE ALGORITHM:-----------
Pattern: 1000110
Match Count: 8171
Comparison Count: 747040
Boyer Moore Running Time: 227.996826171875 miliseconds.
```

Figure-8: Result of First Experiment

We can see that most efficient algorithm is Boyer Moore by looking comparison count and running time. Shifting algorithm of Boyer Moore is more useful than others because it computes both bad symbol and good suffix heuristics and takes best one of them. After Boyer-Moore, Horspool algorithm is second best. It is an algorithm which uses only bad symbol heuristic that simpler than Boyer-Moore. Brute-force is worst one because it searches all the text shifting pattern one by one. Comparison count and running time are largest because of that.

Three algorithm found same match count which is 8171, so we can say that they worked properly.

Running time and comparison count can be seen better in plots below.

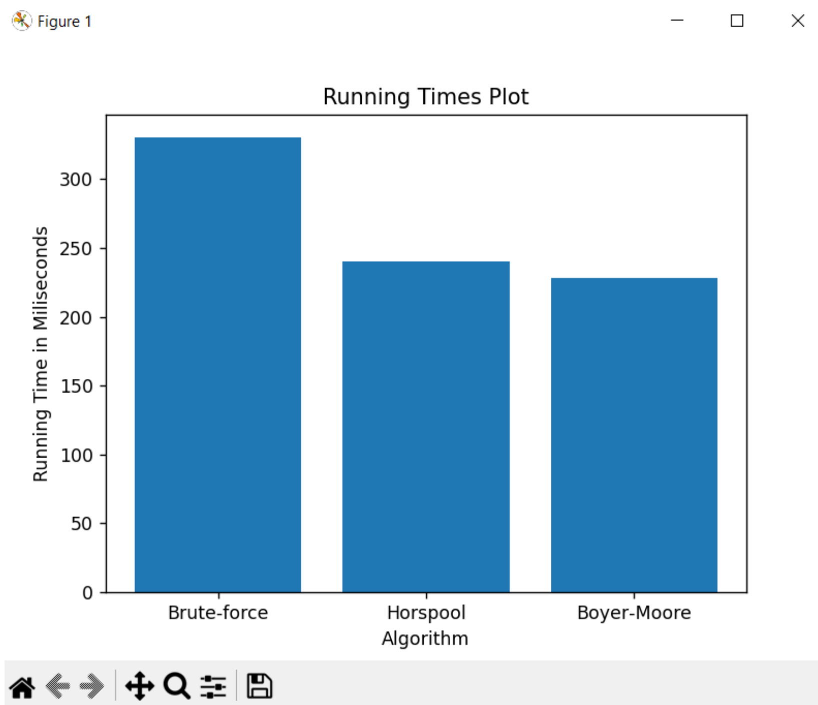

Figure-9: Running Time Plot of First Experiment

Brute-force is an algorithm that has worst efficiency. In this experiment, using Boyer-Moore algorithm is best choice to search binary pattern in binary text.

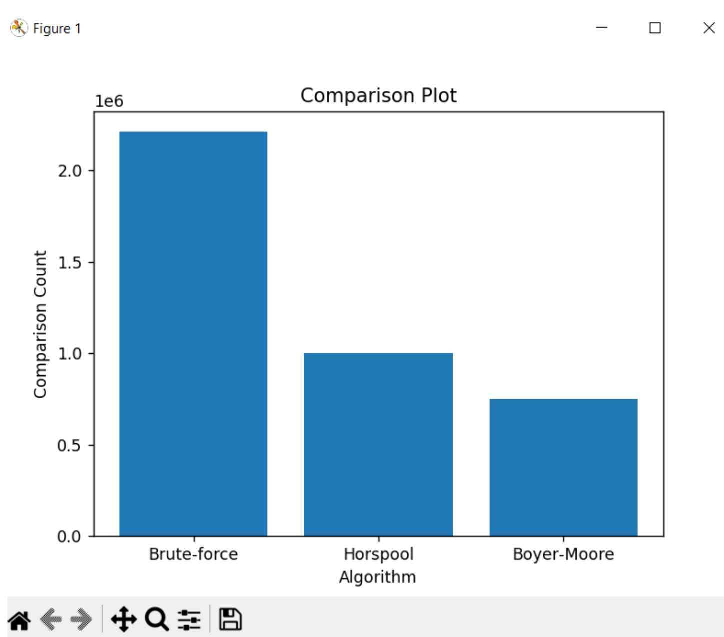

Figure-10: Comparison Count of First Experiment

Also, we can see that Boyer-Moore has the smallest comparison count. There is a huge difference between brute-force and others because of shifting algorithms of them. Horspool can be used because sometimes it is hard to implement Boyer-Moore.

Figure-11: Section of Output HTML File

Patterns that are found in HTML file is marked and highlighted. Section of it can be seen above. Also, overlap example in this text can be seen below.

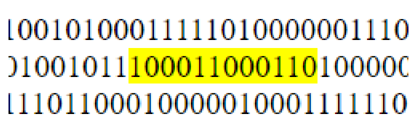

Figure-12: Section of Output HTML File (Overlap)

This highlighted text above includes two patterns that are overlapped. They are marked correctly and printed in output file. Needed controls of overlap situation is done in the code.

Second Experiment is searching ‘quis’ pattern in ‘Lorem.html’. Results of three algorithms are below. Bad Symbol Table, good suffix table, and detailed results of three algorithms are printed.

```
Bad Symbol Table: {'\x00': 4, '\x01': 4, '\x02': 4, '\x03': 4, '\x04': 4, '\x05': 4, '\x06': 4, '\x07': 4, '\x08': 4, '\t': 4, '\n': 4, '\x0b': 4, '\x0c': 4,
 '\r': 4, '\x0e': 4, '\x0f': 4, '\x10': 4, '\x11': 4, '\x12': 4, '\x13': 4, '\x14': 4, '\x15': 4, '\x16': 4, '\x17': 4, '\x18': 4, '\x19': 4, '\x1a': 4, '\x1b'
: 4, '\x1c': 4, '\x1d': 4, '\x1e': 4, '\x1f': 4, ' ': 4, '!': 4, '"': 4, '#': 4, '$': 4, '%': 4, '&': 4, "'": 4, '(': 4, ')': 4, '*': 4, '+': 4, ',': 4, '-':
4, '.': 4, '/': 4, '0': 4, '1': 4, '2': 4, '3': 4, '4': 4, '5': 4, '6': 4, '7': 4, '8': 4, '9': 4, ':': 4, ';': 4, '<': 4, '=': 4, '>': 4, '?': 4, '@': 4, 'A'
: 4, 'B': 4, 'C': 4, 'D': 4, 'E': 4, 'F': 4, 'G': 4, 'H': 4, 'I': 4, 'J': 4, 'K': 4, 'L': 4, 'M': 4, 'N': 4, 'O': 4, 'P': 4, 'Q': 4, 'R': 4, 'S': 4, 'T': 4, '
U': 4, 'V': 4, 'W': 4, 'X': 4, 'Y': 4, 'Z': 4, '[': 4, '\\': 4, ']': 4, '^': 4, '_': 4, '`': 4, 'a': 4, 'b': 4, 'c': 4, 'd': 4, 'e': 4, 'f': 4, 'g': 4, 'h': 4
, 'i': 1, 'j': 4, 'k': 4, 'l': 4, 'm': 4, 'n': 4, 'o': 4, 'p': 4, 'q': 3, 'r': 4, 's': 4, 't': 4, 'u': 2, 'v': 4, 'w': 4, 'x': 4, 'y': 4, 'z': 4, '{': 4, '|':
 4, '}': 4, '~': 4, '\x7f': 4, '\x80': 4, '\x81': 4, '\x82': 4, '\x83': 4, '\x84': 4, '\x85': 4, '\x86': 4, '\x87': 4, '\x88': 4, '\x89': 4, '\x8a': 4, '\x8b'
: 4, '\x8c': 4, '\x8d': 4, '\x8e': 4, '\x8f': 4, '\x90': 4, '\x91': 4, '\x92': 4, '\x93': 4, '\x94': 4, '\x95': 4, '\x96': 4, '\x97': 4, '\x98': 4, '\x99': 4,
 '\x9a': 4, '\x9b': 4, '\x9c': 4, '\x9d': 4, '\x9e': 4, '\x9f': 4, '\xa0': 4, '¡': 4, '¢': 4, '£': 4, '¤': 4, '¥': 4, '¦': 4, '§': 4, '¨': 4, '©': 4, 'ª': 4,
 '«': 4, '¬': 4, '\xad': 4, '®': 4, '¯': 4, '°': 4, '±': 4, '²': 4, '³': 4, '´': 4, 'µ': 4, '¶': 4, '·': 4, '¸': 4, '¹': 4, 'º': 4, '»': 4, '¼': 4, '½': 4, '¾'
: 4, '¿': 4, 'À': 4, 'Á': 4, 'Â': 4, 'Ã': 4, 'Ä': 4, 'Å': 4, 'Æ': 4, 'Ç': 4, 'È': 4, 'É': 4, 'Ê': 4, 'Ë': 4, 'Ì': 4, 'Í': 4, 'Î': 4, 'Ï': 4, 'Ð': 4, 'Ñ': 4, '
Ò': 4, 'Ó': 4, 'Ô': 4, 'Õ': 4, 'Ö': 4, '×': 4, 'Ø': 4, 'Ù': 4, 'Ú': 4, 'Û': 4, 'Ü': 4, 'Ý': 4, 'Þ': 4, 'ß': 4, 'à': 4, 'á': 4, 'â': 4, 'ã': 4, 'ä': 4, 'å': 4,
 'æ': 4, 'ç': 4, 'è': 4, 'é': 4, 'ê': 4, 'ë': 4, 'ì': 4, 'í': 4, 'î': 4, 'ï': 4, 'ð': 4, 'ñ': 4, 'ò': 4, 'ó': 4, 'ô': 4, 'õ': 4, 'ö': 4, '÷': 4, 'ø': 4, 'ù':
4, 'ú': 4, 'û': 4, 'ü': 4, 'ý': 4, 'þ': 4, 'ÿ': 4}

Good Suffix Table: {'s': 4, 'is': 4, 'uis': 4, 'quis': 4}

----------BRUTE-FORCE ALGORITHM:-----------
Pattern: quis
Match Count: 2340
Comparison Count: 1239808
Brute Force Running Time: 163.06233406066895 miliseconds.

----------HORSPOOL ALGORITHM:-----------
Pattern: quis
Match Count: 2340
Comparison Count: 355713
Horspool Running Time: 117.01297760009766 miliseconds.

------------------------------------------------------------

----------BOYER MOORE ALGORITHM:-----------
Pattern: quis
Match Count: 2340
Comparison Count: 355713
Boyer Moore Running Time: 105.99946975708008 miliseconds.
```

Figure-13: Result of Second Experiment

We can see that most efficient algorithm is Boyer Moore by looking comparison count and running time in this experiment also. After Boyer-Moore, Horspool algorithm comes as second. Brute-force is worst one because it searches all the text shifting pattern one by one. Comparison count and running time are largest because of that. In binary and normal inputs, Boyer-Moore is the best string search algorithm so far.

Three algorithm found same match count which is 2340, so we can say that they worked properly. We can observe that running time and comparison count of all of them is smaller than first experiment because ‘Lorem.html’ input has more than two inputs. ‘Binary.html’ input has two inputs. So, second experiment does not take much time as first experiment because comparison of three algorithms is small. Content of input effects these experiments in this way. Good suffix table, and bad symbol table is designed better in second experiment because there are more than ‘1’ and ‘0’, which are Latin letters.

Running time and comparison count can be seen better in plots below.

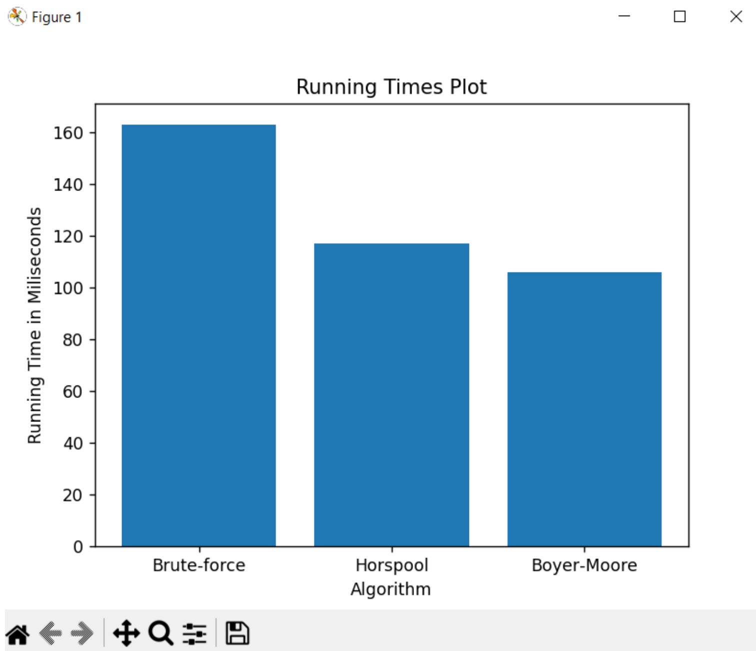

Figure-14: Running Time Plot of Second Experiment

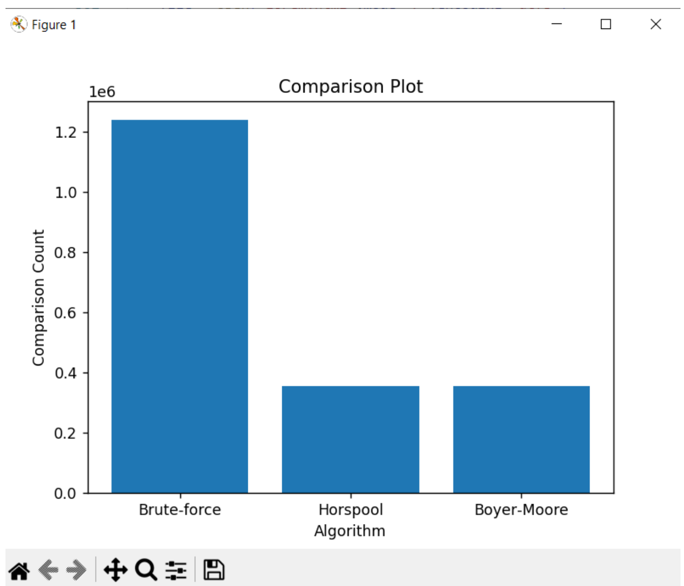

Figure-15: Comparison Count of Second Experiment

We can see the difference between first and second experiment better by looking plots of running time and comparison count. Also, we can observe that efficiency difference between three algorithms.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus laoreet enim non est rhoncus hendrerit. Suspendisse nec consectetur tortor. Vivamus nunc nisl, vestibulum a eros eu, posuere pretium est. Donec efficitur dui a ipsum tincidunt maximus. Mauris non tellus ac nibh congue rutrum. Fusce vitae pellentesque sapien. Sed elementum non felis a semper. Aenean vitae purus sit amet nisl porttitor hendrerit. In dapibus, ligula ut sagittis elementum, massa lectus tempor lorem, eu mattis nibh augue quis sapien. Suspendisse potenti. Integer tristique fermentum orci. Maecenas bibendum tellus purus, ut efficitur urna pellentesque nec. Praesent sollicitudin purus ligula, vel imperdiet velit fringilla consequat. Nam fringilla massa nec velit pellentesque placerat. Ut convallis aliquet orci ac dictum. Nulla iaculis augue nec arcu ultricies finibus. Phasellus fringilla bibendum justo id varius. Praesent fringilla mollis nisi. Sed bibendum sem at dignissim bibendum. Suspendisse et neque elementum, ultricies lorem quis, rutrum sem. Vivamus pulvinar, enim vel porttitor ornare, neque diam rutrum felis, ac commodo nisl lacus at est. Maecenas auctor quis dolor ac auctor. In at augue in tellus porttitor rhoncus ac at lorem. Sed eu urna velit. Sed posuere ligula et orci consequat, et faucibus tellus semper. Fusce et dapibus leo, vel volutpat velit. Aenean aliquet metus lacus, sed bibendum diam pulvinar quis. Integer justo neque, laoreet quis enim at, commodo lacinia justo. Fusce at ligula consectetur, varius turpis nec, tincidunt orci. Nulla in vehicula nisl. Phasellus mi risus, bibendum id dictum id, consectetur sodales nisl. Sed erat massa, tincidunt eget orci a, ultrices finibus erat. Nulla aliquet nulla et efficitur tempus. Praesent sit amet scelerisque sapien. Phasellus vitae vehicula velit. Curabitur ac libero ut velit feugiat maximus. Sed ultricies maximus urna, sed maximus massa dignissim quis. Ut at cursus ligula. Sed pretium, nulla vitae efficitur hendrerit, mauris ante condimentum risus, sed tristique sem sem quis nunc. Donec

Figure-16: Section of Output HTML File

Pattern 'quis' is found in the text. They are highlighted with yellow in output file.

Third Experiment is searching 'lorem' pattern in 'Lorem2.html'. Results of three algorithms are below. Bad Symbol Table, good suffix table, and detailed results of three algorithms are printed.

```
Bad Symbol Table: {'\x00': 5, '\x01': 5, '\x02': 5, '\x03': 5, '\x04': 5, '\x05': 5, '\x06': 5, '\x07': 5, '\x08': 5, '\t': 5, '\n': 5, '\x0b': 5, '\x0c': 5,
'\r': 5, '\x0e': 5, '\x0f': 5, '\x10': 5, '\x11': 5, '\x12': 5, '\x13': 5, '\x14': 5, '\x15': 5, '\x16': 5, '\x17': 5, '\x18': 5, '\x19': 5, '\x1a': 5, '\x1b'
: 5, '\x1c': 5, '\x1d': 5, '\x1e': 5, '\x1f': 5, ' ': 5, '!': 5, '"': 5, '#': 5, '$': 5, '%': 5, '&': 5, "'": 5, '(': 5, ')': 5, '*': 5, '+': 5, ',': 5, '-':
5, '.': 5, '/': 5, '0': 5, '1': 5, '2': 5, '3': 5, '4': 5, '5': 5, '6': 5, '7': 5, '8': 5, '9': 5, ':': 5, ';': 5, '<': 5, '=': 5, '>': 5, '?': 5, '@': 5, 'A'
: 5, 'B': 5, 'C': 5, 'D': 5, 'E': 5, 'F': 5, 'G': 5, 'H': 5, 'I': 5, 'J': 5, 'K': 5, 'L': 5, 'M': 5, 'N': 5, 'O': 5, 'P': 5, 'Q': 5, 'R': 5, 'S': 5, 'T': 5, '
U': 5, 'V': 5, 'W': 5, 'X': 5, 'Y': 5, 'Z': 5, '[': 5, '\\': 5, ']': 5, '^': 5, '_': 5, '`': 5, 'a': 5, 'b': 5, 'c': 5, 'd': 5, 'e': 1, 'f': 5, 'g': 5, 'h': 5
, 'i': 5, 'j': 5, 'k': 5, 'l': 4, 'm': 5, 'n': 5, 'o': 3, 'p': 5, 'q': 5, 'r': 2, 's': 5, 't': 5, 'u': 5, 'v': 5, 'w': 5, 'x': 5, 'y': 5, 'z': 5, '{': 5, '|':
 5, '}': 5, '~': 5, '\x7f': 5, '\x80': 5, '\x81': 5, '\x82': 5, '\x83': 5, '\x84': 5, '\x85': 5, '\x86': 5, '\x87': 5, '\x88': 5, '\x89': 5, '\x8a': 5, '\x8b'
: 5, '\x8c': 5, '\x8d': 5, '\x8e': 5, '\x8f': 5, '\x90': 5, '\x91': 5, '\x92': 5, '\x93': 5, '\x94': 5, '\x95': 5, '\x96': 5, '\x97': 5, '\x98': 5, '\x99': 5,
 '\x9a': 5, '\x9b': 5, '\x9c': 5, '\x9d': 5, '\x9e': 5, '\x9f': 5, '\xa0': 5, '¡': 5, '¢': 5, '£': 5, '¤': 5, '¥': 5, '¦': 5, '§': 5, '¨': 5, '©': 5, 'ª': 5,
'«': 5, '¬': 5, '\xad': 5, '®': 5, '¯': 5, '°': 5, '±': 5, '²': 5, '³': 5, '´': 5, 'µ': 5, '¶': 5, '·': 5, '¸': 5, '¹': 5, 'º': 5, '»': 5, '¼': 5, '½': 5, '¾'
: 5, '¿': 5, 'À': 5, 'Á': 5, 'Â': 5, 'Ã': 5, 'Ä': 5, 'Å': 5, 'Æ': 5, 'Ç': 5, 'È': 5, 'É': 5, 'Ê': 5, 'Ë': 5, 'Ì': 5, 'Í': 5, 'Î': 5, 'Ï': 5, 'Ð': 5, 'Ñ': 5, '
Ò': 5, 'Ó': 5, 'Ô': 5, 'Õ': 5, 'Ö': 5, '×': 5, 'Ø': 5, 'Ù': 5, 'Ú': 5, 'Û': 5, 'Ü': 5, 'Ý': 5, 'Þ': 5, 'ß': 5, 'à': 5, 'á': 5, 'â': 5, 'ã': 5, 'ä': 5, 'å': 5,
 'æ': 5, 'ç': 5, 'è': 5, 'é': 5, 'ê': 5, 'ë': 5, 'ì': 5, 'í': 5, 'î': 5, 'ï': 5, 'ð': 5, 'ñ': 5, 'ò': 5, 'ó': 5, 'ô': 5, 'õ': 5, 'ö': 5, '÷': 5, 'ø': 5, 'ù':
5, 'ú': 5, 'û': 5, 'ü': 5, 'ý': 5, 'þ': 5, 'ÿ': 5}

Good Suffix Table: {'m': 5, 'em': 5, 'rem': 5, 'orem': 5, 'lorem': 5}

----------BRUTE-FORCE ALGORITHM:-----------
Pattern: lorem
Match Count: 1496
Comparison Count: 1805613
Brute Force Running Time: 288.99598121643066 miliseconds.

----------HORSPOOL ALGORITHM:-----------
Pattern: lorem
Match Count: 1496
Comparison Count: 401987
Horspool Running Time: 107.06067085266113 miliseconds.

------------------------------------------------------------

----------BOYER MOORE ALGORITHM:-----------
Pattern: lorem
Match Count: 1496
Comparison Count: 401987
Boyer Moore Running Time: 123.05784225463867 miliseconds.
```

Figure-17: Result of the Third Experiment

We can see that Horspool Algorithm is better in terms of running time in this experiment. Sometimes results can change this way according to pattern and text. Shifting algorithm of Horspool worked better in this experiment. Brute-force is the worst one again in terms of comparison count and running time.

Fourth Experiment is searching ‘maximus’ pattern in ‘Lorem3.html’. Results of three algorithms are below. Bad Symbol Table, good suffix table, and detailed results of three algorithms are printed.

```
Bad Symbol Table: {'\x00': 7, '\x01': 7, '\x02': 7, '\x03': 7, '\x04': 7, '\x05': 7, '\x06': 7, '\x07': 7, '\x08': 7, '\t': 7, '\n': 7, '\x0b': 7, '\x0c': 7,
 '\r': 7, '\x0e': 7, '\x0f': 7, '\x10': 7, '\x11': 7, '\x12': 7, '\x13': 7, '\x14': 7, '\x15': 7, '\x16': 7, '\x17': 7, '\x18': 7, '\x19': 7, '\x1a': 7, '\x1b'
: 7, '\x1c': 7, '\x1d': 7, '\x1e': 7, '\x1f': 7, ' ': 7, '!': 7, '"': 7, '#': 7, '$': 7, '%': 7, '&': 7, "'": 7, '(': 7, ')': 7, '*': 7, '+': 7, ',': 7, '-':
7, '.': 7, '/': 7, '0': 7, '1': 7, '2': 7, '3': 7, '4': 7, '5': 7, '6': 7, '7': 7, '8': 7, '9': 7, ':': 7, ';': 7, '<': 7, '=': 7, '>': 7, '?': 7, '@': 7, 'A'
: 7, 'B': 7, 'C': 7, 'D': 7, 'E': 7, 'F': 7, 'G': 7, 'H': 7, 'I': 7, 'J': 7, 'K': 7, 'L': 7, 'M': 7, 'N': 7, 'O': 7, 'P': 7, 'Q': 7, 'R': 7, 'S': 7, 'T': 7, '
U': 7, 'V': 7, 'W': 7, 'X': 7, 'Y': 7, 'Z': 7, '[': 7, '\\': 7, ']': 7, '^': 7, '_': 7, '`': 7, 'a': 5, 'b': 7, 'c': 7, 'd': 7, 'e': 7, 'f': 7, 'g': 7, 'h': 7
, 'i': 3, 'j': 7, 'k': 7, 'l': 7, 'm': 2, 'n': 7, 'o': 7, 'p': 7, 'q': 7, 'r': 7, 's': 7, 't': 7, 'u': 1, 'v': 7, 'w': 7, 'x': 4, 'y': 7, 'z': 7, '{': 7, '|':
 7, '}': 7, '~': 7, '\x7f': 7, '\x80': 7, '\x81': 7, '\x82': 7, '\x83': 7, '\x84': 7, '\x85': 7, '\x86': 7, '\x87': 7, '\x88': 7, '\x89': 7, '\x8a': 7, '\x8b'
: 7, '\x8c': 7, '\x8d': 7, '\x8e': 7, '\x8f': 7, '\x90': 7, '\x91': 7, '\x92': 7, '\x93': 7, '\x94': 7, '\x95': 7, '\x96': 7, '\x97': 7, '\x98': 7, '\x99': 7,
 '\x9a': 7, '\x9b': 7, '\x9c': 7, '\x9d': 7, '\x9e': 7, '\x9f': 7, '\xa0': 7, '¡': 7, '¢': 7, '£': 7, '¤': 7, '¥': 7, '¦': 7, '§': 7, '¨': 7, '©': 7, 'ª': 7,
'«': 7, '¬': 7, '\xad': 7, '®': 7, '¯': 7, '°': 7, '±': 7, '²': 7, '³': 7, '´': 7, 'µ': 7, '¶': 7, '·': 7, '¸': 7, '¹': 7, 'º': 7, '»': 7, '¼': 7, '½': 7, '¾'
: 7, '¿': 7, 'À': 7, 'Á': 7, 'Â': 7, 'Ã': 7, 'Ä': 7, 'Å': 7, 'Æ': 7, 'Ç': 7, 'È': 7, 'É': 7, 'Ê': 7, 'Ë': 7, 'Ì': 7, 'Í': 7, 'Î': 7, 'Ï': 7, 'Ð': 7, 'Ñ': 7, '
Ò': 7, 'Ó': 7, 'Ô': 7, 'Õ': 7, 'Ö': 7, '×': 7, 'Ø': 7, 'Ù': 7, 'Ú': 7, 'Û': 7, 'Ü': 7, 'Ý': 7, 'Þ': 7, 'ß': 7, 'à': 7, 'á': 7, 'â': 7, 'ã': 7, 'ä': 7, 'å': 7,
 'æ': 7, 'ç': 7, 'è': 7, 'é': 7, 'ê': 7, 'ë': 7, 'ì': 7, 'í': 7, 'î': 7, 'ï': 7, 'ð': 7, 'ñ': 7, 'ò': 7, 'ó': 7, 'ô': 7, 'õ': 7, 'ö': 7, '÷': 7, 'ø': 7, 'ù':
7, 'ú': 7, 'û': 7, 'ü': 7, 'ý': 7, 'þ': 7, 'ÿ': 7}

Good Suffix Table: {'s': 7, 'us': 7, 'mus': 7, 'imus': 7, 'ximus': 7, 'aximus': 7, 'maximus': 7}

----------BRUTE-FORCE ALGORITHM:-----------
Pattern: maximus
Match Count: 858
Comparison Count: 1370794
Brute Force Running Time: 189.99910354614258 miliseconds.

----------HORSPOOL ALGORITHM:-----------
Pattern: maximus
Match Count: 858
Comparison Count: 233666
Horspool Running Time: 86.98916435241699 miliseconds.

------------------------------------------------------------

----------BOYER MOORE ALGORITHM:-----------
Pattern: maximus
Match Count: 858
Comparison Count: 233666
Boyer Moore Running Time: 65.05942344665527 miliseconds.
```

Figure-18: Result of the Fourth Experiment

Boyer-Moore gives the best result for this experiment. It is the most efficient way to search ‘maximus’ in the text of ‘Lorem3.html’. They found 858 ‘maximus’ pattern in the input file. Brute-force is the worst one for this algorithm. Difference in comparison time is huge, and also running time is worst among three of them.

<u>Fifth Experiment</u> is searching ‘100111100011011’ pattern in ‘Binary2.html’. Results of three algorithms are below. Bad Symbol Table, good suffix table, and detailed results of three algorithms are printed.

```
Bad Symbol Table: {'\x00': 15, '\x01': 15, '\x02': 15, '\x03': 15, '\x04': 15, '\x05': 15, '\x06': 15, '\x07': 15, '\x08': 15, '\t': 15, '\n': 15, '\x0b': 15,
 '\x0c': 15, '\r': 15, '\x0e': 15, '\x0f': 15, '\x10': 15, '\x11': 15, '\x12': 15, '\x13': 15, '\x14': 15, '\x15': 15, '\x16': 15, '\x17': 15, '\x18': 15, '\x
19': 15, '\x1a': 15, '\x1b': 15, '\x1c': 15, '\x1d': 15, '\x1e': 15, '\x1f': 15, ' ': 15, '!': 15, '"': 15, '#': 15, '$': 15, '%': 15, '&': 15, "'": 15, '(':
15, ')': 15, '*': 15, '+': 15, ',': 15, '-': 15, '.': 15, '/': 15, '0': 2, '1': 1, '2': 15, '3': 15, '4': 15, '5': 15, '6': 15, '7': 15, '8': 15, '9': 15, ':'
: 15, ';': 15, '<': 15, '=': 15, '>': 15, '?': 15, '@': 15, 'A': 15, 'B': 15, 'C': 15, 'D': 15, 'E': 15, 'F': 15, 'G': 15, 'H': 15, 'I': 15, 'J': 15, 'K': 15,
 'L': 15, 'M': 15, 'N': 15, 'O': 15, 'P': 15, 'Q': 15, 'R': 15, 'S': 15, 'T': 15, 'U': 15, 'V': 15, 'W': 15, 'X': 15, 'Y': 15, 'Z': 15, '[': 15, '\\': 15, ']'
: 15, '^': 15, '_': 15, '`': 15, 'a': 15, 'b': 15, 'c': 15, 'd': 15, 'e': 15, 'f': 15, 'g': 15, 'h': 15, 'i': 15, 'j': 15, 'k': 15, 'l': 15, 'm': 15, 'n': 15,
 'o': 15, 'p': 15, 'q': 15, 'r': 15, 's': 15, 't': 15, 'u': 15, 'v': 15, 'w': 15, 'x': 15, 'y': 15, 'z': 15, '{': 15, '|': 15, '}': 15, '~': 15, '\x7f': 15, '
\x80': 15, '\x81': 15, '\x82': 15, '\x83': 15, '\x84': 15, '\x85': 15, '\x86': 15, '\x87': 15, '\x88': 15, '\x89': 15, '\x8a': 15, '\x8b': 15, '\x8c': 15, '\x
8d': 15, '\x8e': 15, '\x8f': 15, '\x90': 15, '\x91': 15, '\x92': 15, '\x93': 15, '\x94': 15, '\x95': 15, '\x96': 15, '\x97': 15, '\x98': 15, '\x99': 15, '\x9a
': 15, '\x9b': 15, '\x9c': 15, '\x9d': 15, '\x9e': 15, '\x9f': 15, '\xa0': 15, '¡': 15, '¢': 15, '£': 15, '¤': 15, '¥': 15, '¦': 15, '§': 15, '¨': 15, '©': 15
, 'ª': 15, '«': 15, '¬': 15, '\xad': 15, '®': 15, '¯': 15, '°': 15, '±': 15, '²': 15, '³': 15, '´': 15, 'µ': 15, '¶': 15, '·': 15, '¸': 15, '¹': 15, 'º': 15,
'»': 15, '¼': 15, '½': 15, '¾': 15, '¿': 15, 'À': 15, 'Á': 15, 'Â': 15, 'Ã': 15, 'Ä': 15, 'Å': 15, 'Æ': 15, 'Ç': 15, 'È': 15, 'É': 15, 'Ê': 15, 'Ë': 15, 'Ì':
15, 'Í': 15, 'Î': 15, 'Ï': 15, 'Ð': 15, 'Ñ': 15, 'Ò': 15, 'Ó': 15, 'Ô': 15, 'Õ': 15, 'Ö': 15, '×': 15, 'Ø': 15, 'Ù': 15, 'Ú': 15, 'Û': 15, 'Ü': 15, 'Ý': 15, '
Þ': 15, 'ß': 15, 'à': 15, 'á': 15, 'â': 15, 'ã': 15, 'ä': 15, 'å': 15, 'æ': 15, 'ç': 15, 'è': 15, 'é': 15, 'ê': 15, 'ë': 15, 'ì': 15, 'í': 15, 'î': 15, 'ï': 1
5, 'ð': 15, 'ñ': 15, 'ò': 15, 'ó': 15, 'ô': 15, 'õ': 15, 'ö': 15, '÷': 15, 'ø': 15, 'ù': 15, 'ú': 15, 'û': 15, 'ü': 15, 'ý': 15, 'þ': 15, 'ÿ': 15}

Good Suffix Table: {'1': 1, '11': 8, '011': 3, '1011': 14, '11011': 14, '011011': 14, '0011011': 14, '00011011': 14, '100011011': 14, '1100011011': 14, '11100
011011': 14, '111100011011': 14, '0111100011011': 14, '00111100011011': 14, '100111100011011': 14}

----------BRUTE-FORCE ALGORITHM:-----------
Pattern: 100111100011011
Match Count: 30
Comparison Count: 2209803
Brute Force Running Time: 294.994592666626 miliseconds.

----------HORSPOOL ALGORITHM:-----------
Pattern: 100111100011011
Match Count: 30
Comparison Count: 1125616
Horspool Running Time: 284.03449058532715 miliseconds.

------------------------------------------------------------

----------BOYER MOORE ALGORITHM:-----------
Pattern: 100111100011011
Match Count: 30
Comparison Count: 518244
Boyer Moore Running Time: 138.99707794189453 miliseconds.
```

Figure-19: Result of the Fifth Experiment

We can see that Boyer-Moore is much better than other two algorithms in terms of running time. Horspool algorithm is weak for this experiment because of large pattern and text. Good suffix table makes Boyer-Moore algorithm advantageous for fifth experiment. So, we can say that pattern, bad symbol table, good suffix table and text are very important to select which algorithm to use.

<u>Sixth Experiment</u> is searching ‘000111000111’ pattern in ‘Binary3.html’. Results of three algorithms are below. Bad Symbol Table, good suffix table, and detailed results of three algorithms are printed.

```
Bad Symbol Table: {'\x00': 12, '\x01': 12, '\x02': 12, '\x03': 12, '\x04': 12, '\x05': 12, '\x06': 12, '\x07': 12, '\x08': 12, '\t': 12, '\n': 12, '\x0b': 12,
 '\x0c': 12, '\r': 12, '\x0e': 12, '\x0f': 12, '\x10': 12, '\x11': 12, '\x12': 12, '\x13': 12, '\x14': 12, '\x15': 12, '\x16': 12, '\x17': 12, '\x18': 12, '\x
19': 12, '\x1a': 12, '\x1b': 12, '\x1c': 12, '\x1d': 12, '\x1e': 12, '\x1f': 12, ' ': 12, '!': 12, '"': 12, '#': 12, '$': 12, '%': 12, '&': 12, "'": 12, '(':
12, ')': 12, '*': 12, '+': 12, ',': 12, '-': 12, '.': 12, '/': 12, '0': 3, '1': 1, '2': 12, '3': 12, '4': 12, '5': 12, '6': 12, '7': 12, '8': 12, '9': 12, ':'
: 12, ';': 12, '<': 12, '=': 12, '>': 12, '?': 12, '@': 12, 'A': 12, 'B': 12, 'C': 12, 'D': 12, 'E': 12, 'F': 12, 'G': 12, 'H': 12, 'I': 12, 'J': 12, 'K': 12,
 'L': 12, 'M': 12, 'N': 12, 'O': 12, 'P': 12, 'Q': 12, 'R': 12, 'S': 12, 'T': 12, 'U': 12, 'V': 12, 'W': 12, 'X': 12, 'Y': 12, 'Z': 12, '[': 12, '\\': 12, ']'
: 12, '^': 12, '_': 12, '`': 12, 'a': 12, 'b': 12, 'c': 12, 'd': 12, 'e': 12, 'f': 12, 'g': 12, 'h': 12, 'i': 12, 'j': 12, 'k': 12, 'l': 12, 'm': 12, 'n': 12,
 'o': 12, 'p': 12, 'q': 12, 'r': 12, 's': 12, 't': 12, 'u': 12, 'v': 12, 'w': 12, 'x': 12, 'y': 12, 'z': 12, '{': 12, '|': 12, '}': 12, '~': 12, '\x7f': 12, '
\x80': 12, '\x81': 12, '\x82': 12, '\x83': 12, '\x84': 12, '\x85': 12, '\x86': 12, '\x87': 12, '\x88': 12, '\x89': 12, '\x8a': 12, '\x8b': 12, '\x8c': 12, '\x
8d': 12, '\x8e': 12, '\x8f': 12, '\x90': 12, '\x91': 12, '\x92': 12, '\x93': 12, '\x94': 12, '\x95': 12, '\x96': 12, '\x97': 12, '\x98': 12, '\x99': 12, '\x9a
': 12, '\x9b': 12, '\x9c': 12, '\x9d': 12, '\x9e': 12, '\x9f': 12, '\xa0': 12, '¡': 12, '¢': 12, '£': 12, '¤': 12, '¥': 12, '¦': 12, '§': 12, '¨': 12, '©': 12
, 'ª': 12, '«': 12, '¬': 12, '\xad': 12, '®': 12, '¯': 12, '°': 12, '±': 12, '²': 12, '³': 12, '´': 12, 'µ': 12, '¶': 12, '·': 12, '¸': 12, '¹': 12, 'º': 12,
'»': 12, '¼': 12, '½': 12, '¾': 12, '¿': 12, 'À': 12, 'Á': 12, 'Â': 12, 'Ã': 12, 'Ä': 12, 'Å': 12, 'Æ': 12, 'Ç': 12, 'È': 12, 'É': 12, 'Ê': 12, 'Ë': 12, 'Ì':
12, 'Í': 12, 'Î': 12, 'Ï': 12, 'Ð': 12, 'Ñ': 12, 'Ò': 12, 'Ó': 12, 'Ô': 12, 'Õ': 12, 'Ö': 12, '×': 12, 'Ø': 12, 'Ù': 12, 'Ú': 12, 'Û': 12, 'Ü': 12, 'Ý': 12, '
Þ': 12, 'ß': 12, 'à': 12, 'á': 12, 'â': 12, 'ã': 12, 'ä': 12, 'å': 12, 'æ': 12, 'ç': 12, 'è': 12, 'é': 12, 'ê': 12, 'ë': 12, 'ì': 12, 'í': 12, 'î': 12, 'ï': 1
2, 'ð': 12, 'ñ': 12, 'ò': 12, 'ó': 12, 'ô': 12, 'õ': 12, 'ö': 12, '÷': 12, 'ø': 12, 'ù': 12, 'ú': 12, 'û': 12, 'ü': 12, 'ý': 12, 'þ': 12, 'ÿ': 12}

Good Suffix Table: {'1': 2, '11': 1, '111': 12, '0111': 12, '00111': 12, '000111': 6, '1000111': 6, '11000111': 6, '111000111': 6, '0111000111': 6, '001110001
11': 6, '000111000111': 6}

----------BRUTE-FORCE ALGORITHM:-----------
Pattern: 000111000111
Match Count: 234
Comparison Count: 2211881
Brute Force Running Time: 300.9977340698242 miliseconds.

----------HORSPOOL ALGORITHM:-----------
Pattern: 000111000111
Match Count: 234
Comparison Count: 960381
Horspool Running Time: 244.05860900878906 miliseconds.

------------------------------------------------------------

----------BOYER MOORE ALGORITHM:-----------
Pattern: 000111000111
Match Count: 234
Comparison Count: 482408
Boyer Moore Running Time: 155.98821640014648 miliseconds.
```

Figure-20: Result of the Sixth Experiment

In all of binary experiments, we can observe that Boyer-Moore is far better than other algorithms because it uses two heuristics that are called bad symbol and good suffix. So, if there is a text that includes few symbols, Boyer-Moore must be used. Binary inputs show this critical information.

Seventh Experiment is searching 'AT_THAT' pattern in 'at_that_example.html'. Results of three algorithms are below. Bad Symbol Table, good suffix table, and detailed results of three algorithms are printed.

```
Bad Symbol Table: {'\x00': 7, '\x01': 7, '\x02': 7, '\x03': 7, '\x04': 7, '\x05': 7, '\x06': 7, '\x07': 7, '\x08': 7, '\t': 7, '\n': 7, '\x0b': 7, '\x0c': 7,
'\r': 7, '\x0e': 7, '\x0f': 7, '\x10': 7, '\x11': 7, '\x12': 7, '\x13': 7, '\x14': 7, '\x15': 7, '\x16': 7, '\x17': 7, '\x18': 7, '\x19': 7, '\x1a': 7, '\x1b'
: 7, '\x1c': 7, '\x1d': 7, '\x1e': 7, '\x1f': 7, ' ': 7, '!': 7, '"': 7, '#': 7, '$': 7, '%': 7, '&': 7, "'": 7, '(': 7, ')': 7, '*': 7, '+': 7, ',': 7, '-':
7, '.': 7, '/': 7, '0': 7, '1': 7, '2': 7, '3': 7, '4': 7, '5': 7, '6': 7, '7': 7, '8': 7, '9': 7, ':': 7, ';': 7, '<': 7, '=': 7, '>': 7, '?': 7, '@': 7, 'A'
: 1, 'B': 7, 'C': 7, 'D': 7, 'E': 7, 'F': 7, 'G': 7, 'H': 2, 'I': 7, 'J': 7, 'K': 7, 'L': 7, 'M': 7, 'N': 7, 'O': 7, 'P': 7, 'Q': 7, 'R': 7, 'S': 7, 'T': 3, '
U': 7, 'V': 7, 'W': 7, 'X': 7, 'Y': 7, 'Z': 7, '[': 7, '\\': 7, ']': 7, '^': 7, '_': 4, '`': 7, 'a': 7, 'b': 7, 'c': 7, 'd': 7, 'e': 7, 'f': 7, 'g': 7, 'h': 7
, 'i': 7, 'j': 7, 'k': 7, 'l': 7, 'm': 7, 'n': 7, 'o': 7, 'p': 7, 'q': 7, 'r': 7, 's': 7, 't': 7, 'u': 7, 'v': 7, 'w': 7, 'x': 7, 'y': 7, 'z': 7, '{': 7, '|':
 7, '}': 7, '~': 7, '\x7f': 7, '\x80': 7, '\x81': 7, '\x82': 7, '\x83': 7, '\x84': 7, '\x85': 7, '\x86': 7, '\x87': 7, '\x88': 7, '\x89': 7, '\x8a': 7, '\x8b'
: 7, '\x8c': 7, '\x8d': 7, '\x8e': 7, '\x8f': 7, '\x90': 7, '\x91': 7, '\x92': 7, '\x93': 7, '\x94': 7, '\x95': 7, '\x96': 7, '\x97': 7, '\x98': 7, '\x99': 7,
 '\x9a': 7, '\x9b': 7, '\x9c': 7, '\x9d': 7, '\x9e': 7, '\x9f': 7, '\xa0': 7, '¡': 7, '¢': 7, '£': 7, '¤': 7, '¥': 7, '¦': 7, '§': 7, '¨': 7, '©': 7, 'ª': 7,
'«': 7, '¬': 7, '\xad': 7, '®': 7, '¯': 7, '°': 7, '±': 7, '²': 7, '³': 7, '´': 7, 'µ': 7, '¶': 7, '·': 7, '¸': 7, '¹': 7, 'º': 7, '»': 7, '¼': 7, '½': 7, '¾'
: 7, '¿': 7, 'À': 7, 'Á': 7, 'Â': 7, 'Ã': 7, 'Ä': 7, 'Å': 7, 'Æ': 7, 'Ç': 7, 'È': 7, 'É': 7, 'Ê': 7, 'Ë': 7, 'Ì': 7, 'Í': 7, 'Î': 7, 'Ï': 7, 'Ð': 7, 'Ñ': 7, '
Ò': 7, 'Ó': 7, 'Ô': 7, 'Õ': 7, 'Ö': 7, '×': 7, 'Ø': 7, 'Ù': 7, 'Ú': 7, 'Û': 7, 'Ü': 7, 'Ý': 7, 'Þ': 7, 'ß': 7, 'à': 7, 'á': 7, 'â': 7, 'ã': 7, 'ä': 7, 'å': 7,
 'æ': 7, 'ç': 7, 'è': 7, 'é': 7, 'ê': 7, 'ë': 7, 'ì': 7, 'í': 7, 'î': 7, 'ï': 7, 'ð': 7, 'ñ': 7, 'ò': 7, 'ó': 7, 'ô': 7, 'õ': 7, 'ö': 7, '÷': 7, 'ø': 7, 'ù':
7, 'ú': 7, 'û': 7, 'ü': 7, 'ý': 7, 'þ': 7, 'ÿ': 7}

Good Suffix Table: {'T': 3, 'AT': 5, 'HAT': 5, 'THAT': 5, '_THAT': 5, 'T_THAT': 5, 'AT_THAT': 5}

----------BRUTE-FORCE ALGORITHM:-----------
Pattern: AT_THAT
Match Count: 1
Comparison Count: 69
Brute Force Running Time: 0.0 miliseconds.

----------HORSPOOL ALGORITHM:-----------
Pattern: AT_THAT
Match Count: 1
Comparison Count: 17
Horspool Running Time: 0.0 miliseconds.

------------------------------------------------------------

----------BOYER MOORE ALGORITHM:-----------
Pattern: AT_THAT
Match Count: 1
Comparison Count: 17
Boyer Moore Running Time: 0.0 miliseconds.
```

Figure-21: Results of Seventh Experiment

In this experiment, Horspool and Boyer-Moore results are nearly same because length of text is very small. So, we can look at comparison counts of them. Brute-force algorithm has 69 comparison count, but others are smaller than brute-force result. All of them found 1 occurrence of ‘AT_THAT’ pattern.

Division of Labor

Hasan Şenyurt – 150120531: Boyer-Moore Algorithm, Report (Analysis, General Design)

Mert Akbal – 150119703: Horspool Algorithm, Report (Code)

Hakan Kenar – 150119675: Brute Force, Report (Input File, Code)

Ahsen Yağmur Kahyaoğlu – 150119788: Brute Force, Report (Input File, Analysis