UNC CHARLOTTE

ITCS 6114 Algorithms and Data Structures Project Report

Pattern Matching Algorithms

by

PRACHI MANOJKUMAR SHENDE (801306878)

YASH AGRAWAL (801316596)

April 25, 2023

FINITE AUTOMATION FOR PATTERN MATCHING

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

The string-matching automaton is a very useful tool which is used in string matching algorithm. String matching algorithms build a finite automaton scans the text string T for all occurrences of the pattern P.

- Idea of this approach is to build finite automata to scan text T for finding all occurrences of pattern P.
- This approach examines each character of text exactly once to find the pattern. Thus it takes linear time for matching but preprocessing time may be large.
- It is defined by tuple M = {Q, Σ, q, F, d} Where Q = Set of States in finite automata

Σ=Sets of input symbols
q. = Initial state
F = Final State
σ = Transition function
Time Complexity = $O(M^3|\Sigma|)$
A finite automaton M is a 5-tuple **(Q, q0,A,∑δ)**,

where,
Q is a finite set of states,
q0 ∈ Q is the start state,
A ⊆ Q is a notable set of accepting states,
∑ is a finite input alphabet,
δ is a function from Q x ∑ into Q called the transition function of M.

The finite automaton starts in state q0 and reads the characters of its input string one at a time. If the automaton is in state q and reads input character a, it moves from state q to state δ (q, a). Whenever its current state q is a member of A, the machine M has accepted the string read so far. An input that is not allowed is rejected.
A finite automaton M induces a function ∅ called the called the final-state function, from ∑* to Q such that ∅(w) is the state M ends up in after scanning the string w. Thus, M accepts a string w if and only if ∅(w) ∈ A.

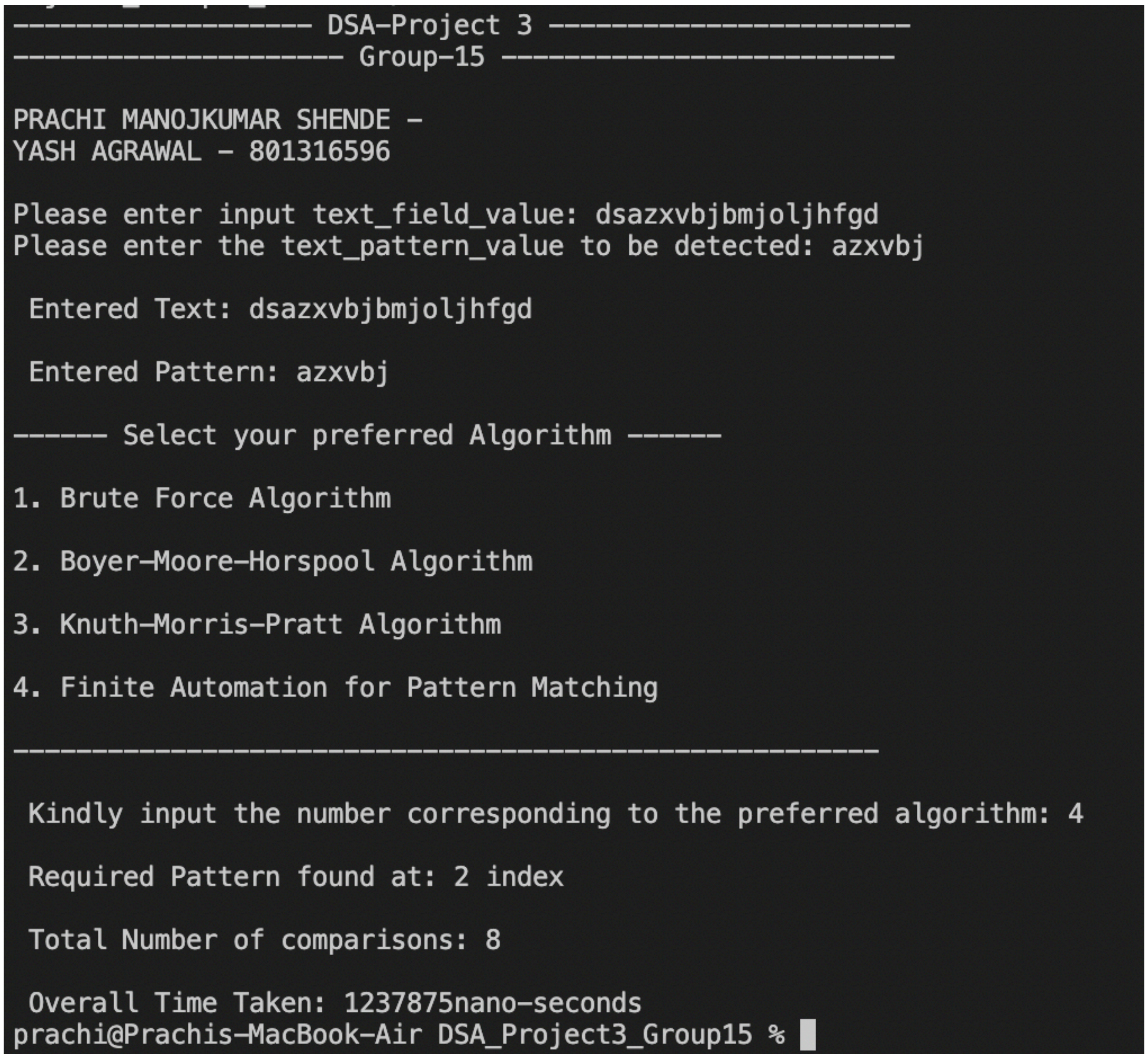
Algorithm

```
FINTE_AUTOMATA(T, P)
// T is text of length n
// P is pattern of length m

state ← 0    // Initial state is 0
for  i ← 1 to n do
    state ← δ(state, t¬i)    // Return the new state
after reading character ti
    if state == m then
        print “Match found on position”, i - m + 1
    end
end
```

Complexity Analysis

- Construction of finite automata takes O(m^3 |Σ|) time in worst case, where m is the length of pattern and |S| is number of input symbols.
- This approach examines each character of text exactly once to find the pattern. So it takes linear time for matching. But pre-processing time may be large. Constructing automata is the difficult part for pattern matching.
- It runs fairly in O(n) time without considering time spend to build transition function.

Input/Output



```
------------------- DSA-Project 3 -----------------------
--------------------- Group-15 -------------------------

PRACHI MANOJKUMAR SHENDE -
YASH AGRAWAL - 801316596

Please enter input text_field_value: dsazxvbjbmjoljhfgd
Please enter the text_pattern_value to be detected: azxvbj

 Entered Text: dsazxvbjbmjoljhfgd

 Entered Pattern: azxvbj

------ Select your preferred Algorithm ------

1. Brute Force Algorithm

2. Boyer-Moore-Horspool Algorithm

3. Knuth-Morris-Pratt Algorithm

4. Finite Automation for Pattern Matching

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

 Kindly input the number corresponding to the preferred algorithm: 4

 Required Pattern found at: 2 index

 Total Number of comparisons: 8

 Overall Time Taken: 1237875nano-seconds
prachi@Prachis-MacBook-Air DSA_Project3_Group15 %
```

BRUTE-FORCE ALGORITHM

It compares first character of pattern with searchable text. If match is found, pointers in both strings are advanced. If match not found, pointer of text is incremented and pointer of pattern is reset. This process is repeated until the end of the text.

It does not require any pre-processing. It directly starts comparing both strings character by character.

Time Complexity = **O(m* (n-m))**

To implement the pattern matching algorithm using the Finite Automaton approach, the user inputs both the text and pattern as strings. The algorithm requires an extra array to store the finite automaton's states, which the algorithm uses to match the pattern. A brute-force algorithm is a simple and straightforward algorithm that is often used for pattern matching. The basic idea of this algorithm is to compare each character of the pattern with each character of the text, starting from the leftmost position.

The brute force algorithm searches all the positions in the text between 0 and n-m, whether the occurrence of the pattern starts there or not. After each attempt, it shifts the pattern to the right by exactly 1 position. The time complexity of this algorithm is O(m*n). If we are searching for n characters in a string of m characters, then it will take n*m tries.

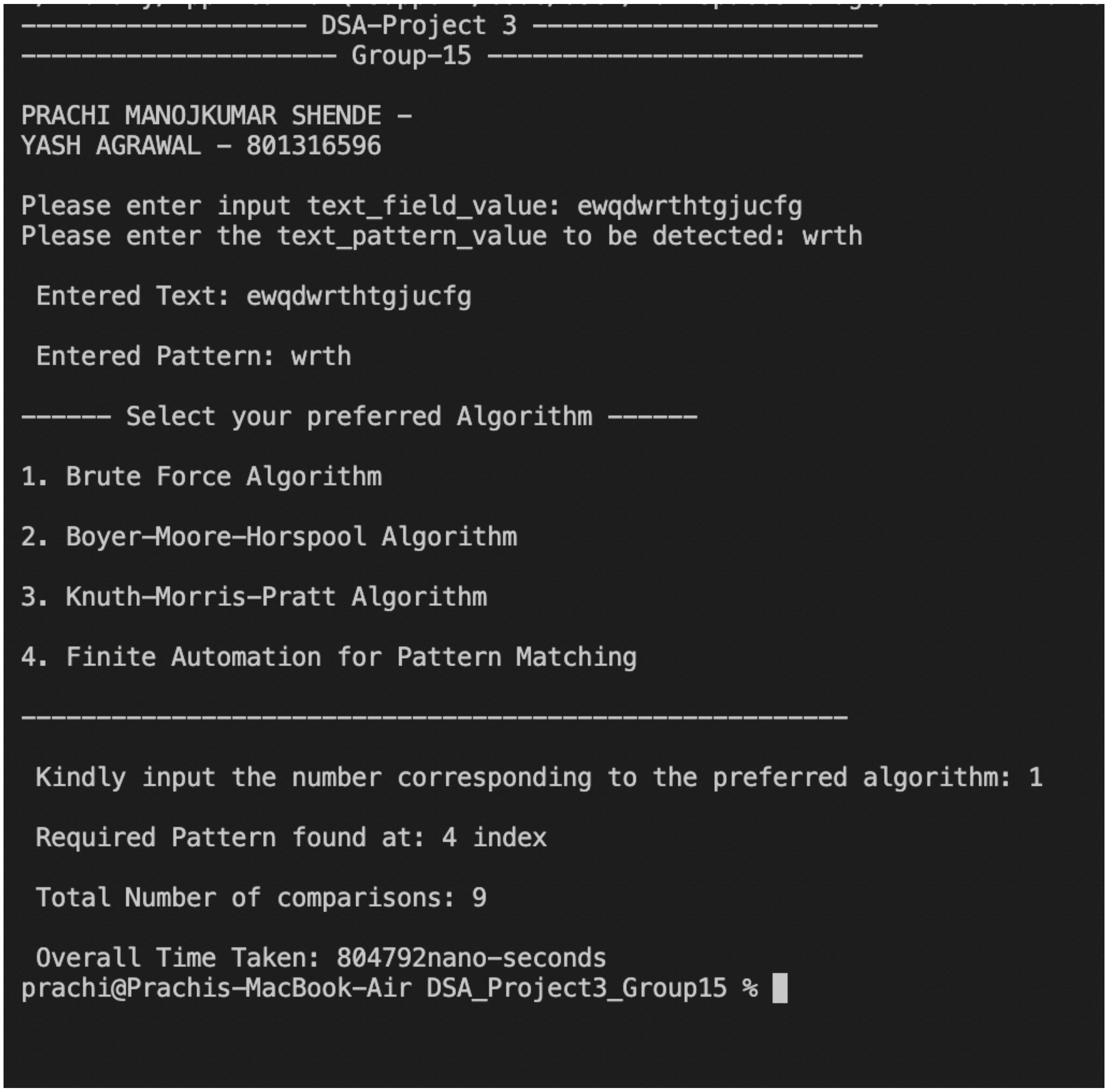
Algorithm

```
Searching for a pattern, P[0...m-1], in
text, T[0...n-1]

BruteForceStringMatch(T[0...n-1], P[0...m-1])
for i ← 0 to n-m
        do
                j ← 0
while j < m and P[j] = T[i+j]
        do
                j++
if j = m then
        return i
return -1
```

Complexity Analysis

The time complexity of brute force is **O(mn)**, which is sometimes written as **O(n*m)** . So, if we were to search for a string of "n" characters in a string of "m" characters using brute force, it would take us n * m tries.

Input/Output



```
------------------- DSA-Project 3 -----------------------
-------------------- Group-15 -------------------------

PRACHI MANOJKUMAR SHENDE -
YASH AGRAWAL - 801316596

Please enter input text_field_value: ewqdwrthtgjucfg
Please enter the text_pattern_value to be detected: wrth

 Entered Text: ewqdwrthtgjucfg

 Entered Pattern: wrth

------ Select your preferred Algorithm ------

1. Brute Force Algorithm

2. Boyer-Moore-Horspool Algorithm

3. Knuth-Morris-Pratt Algorithm

4. Finite Automation for Pattern Matching

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

 Kindly input the number corresponding to the preferred algorithm: 1

 Required Pattern found at: 4 index

 Total Number of comparisons: 9

 Overall Time Taken: 804792nano-seconds
prachi@Prachis-MacBook-Air DSA_Project3_Group15 %
```

KNUTH - MORRIS - PRATT ALGORITHM

The KMP matching algorithm uses degenerating property (pattern having the same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst-case complexity to O(n).
The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match.

```
# Initialising variables:
i = 0
j = 0
m = len(W)
n = len(S)

# Start search:
while (i < m && j < n){
   # Last character matches -> Substring found:
   if (W[i] == S[j] && i == m - 1):
      return true

   # Character matches:
   else if (W[i] == S[j]):
      i++
      j++

   # Character didn't match -> Backtrack:
   else:
      i = arr[i - 1]
      if i == 0:
         j++
}

# Substring not found:
return false
```

- KMP algorithm preprocesses pat[] and constructs an auxiliary lps[] of size m (same as the size of the pattern) which is used to skip characters while matching.

- Name lps indicates the longest proper prefix which is also a suffix. A proper prefix is a prefix with a whole string not allowed. For example, prefixes of "ABC" are "", "A", "AB" and "ABC". Proper prefixes are "", "A" and "AB". Suffixes of the string are "", "C", "BC", and "ABC".

- We search for lps in sub-patterns. More clearly we focus on sub-strings of patterns that are both prefix and suffix.

- For each sub-pattern pat[0..i] where i = 0 to m-1, lps[i] stores the length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].

Unlike the Naive algorithm, where we slide the pattern by one and compare all characters at each shift, we use a value from lps[] to decide the next characters to be matched. The idea is to not match a character that we know will anyway match.

How to use lps to decide the next positions

- We start the comparison of pat[j] with j = 0 with characters of the current window of text.
- We keep matching characters txt[i] and pat[j] and keep incrementing i and j while pat[j] and txt[i] keep **matching**.
- When we see a **mismatch**
- We know that characters pat[0..j-1] match with txt[i-j…i-1] (Note that j starts with 0 and increments it only when there is a match).
- We also know (from the above definition) that lps[j-1] is the count of characters of pat[0…j-1] that are both proper prefix and suffix.
- From the above two points, we can conclude that we do not need to match these lps[j-1] characters with txt[i-j…i-1] because we know that these characters will anyway match.

Complexity Analysis

Time Complexity: O(m + n)
Space Complexity: O(m + n)

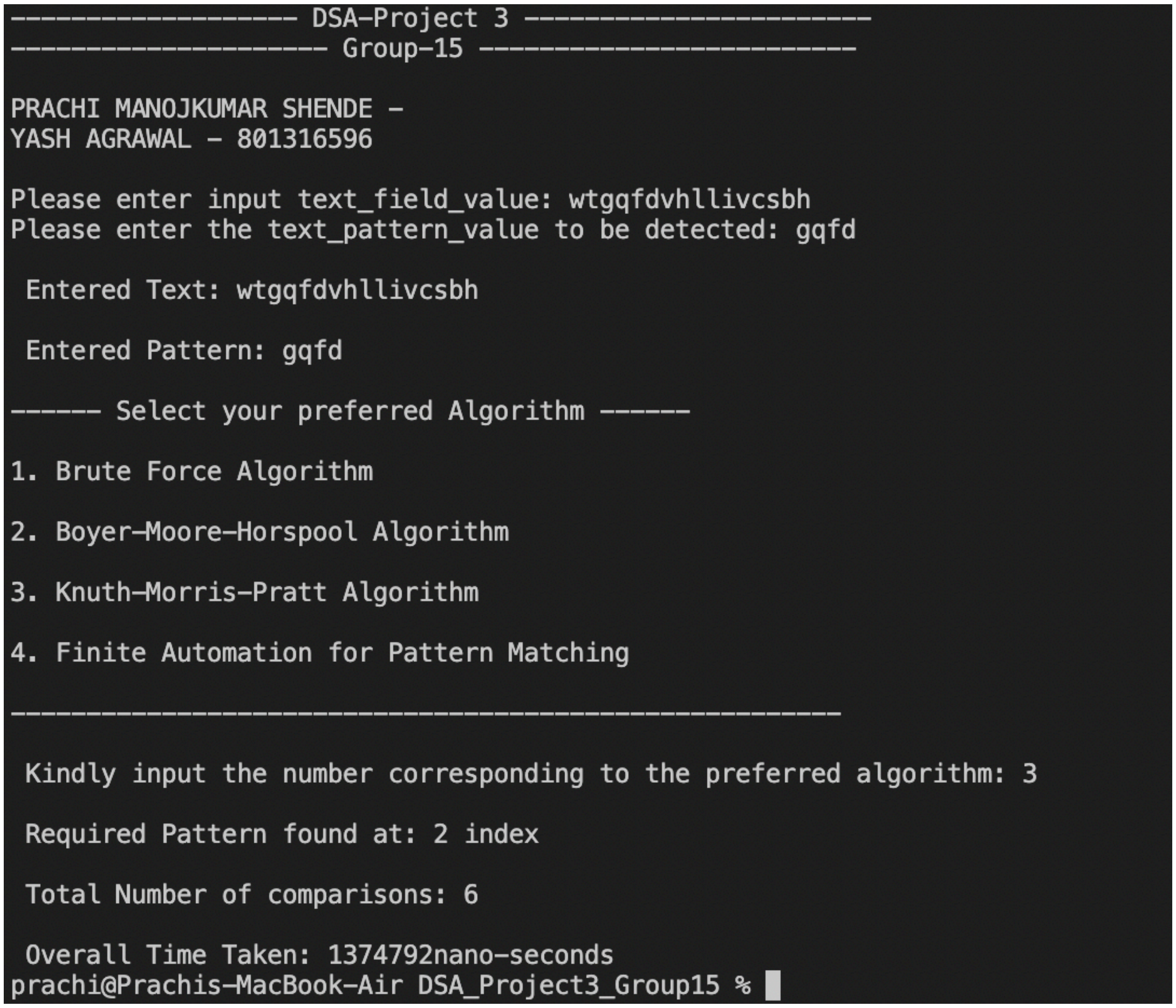
Algorithm

HorspoolMatching(T, P) ---------- --> O(nm) = Worst Case Ex. P = ba^{m-1} and T = a^n, O(n/m) = Best Case Ex. P = b^m and T = a^n. However for random texts it is O(n).

Input: Pattern P and text T
Output: Index the left end of the first matching substring or -1 if no such substring exists

```
ShiftTable(P) // Generate Shift Table --------- --> O( |Σ| + m ) = O( S + m ) //
 S = Size of Alphabet, m = Size of Pattern.
i = m -1 // Position of the Pattern's right end
**while** i <= n-1 **do** ---------- --> O(n)
        k = 0 // Number of matched Characters
        **while** k <= m-1 and P[m-1-k] = T[i-k] **do**
                k=k+1
        **if** k = m

        return i – m + 1
**else** i = i + table[T[i]]
**return -1**
```

Input/Output



```
-------------------- DSA-Project 3 -----------------------
---------------------- Group-15 -------------------------

PRACHI MANOJKUMAR SHENDE -
YASH AGRAWAL - 801316596

Please enter input text_field_value: wtgqfdvhllivcsbh
Please enter the text_pattern_value to be detected: gqfd

 Entered Text: wtgqfdvhllivcsbh

 Entered Pattern: gqfd

------ Select your preferred Algorithm ------

1. Brute Force Algorithm

2. Boyer-Moore-Horspool Algorithm

3. Knuth-Morris-Pratt Algorithm

4. Finite Automation for Pattern Matching

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

 Kindly input the number corresponding to the preferred algorithm: 3

 Required Pattern found at: 2 index

 Total Number of comparisons: 6

 Overall Time Taken: 1374792nano-seconds
prachi@Prachis-MacBook-Air DSA_Project3_Group15 % 
```

BOYER MOORE HORPOOL ALGORITHM

Like KMP and Finite Automata algorithms, Boyer Moore algorithm also preprocesses the pattern. Boyer Moore is a combination of the following two approaches.

- Bad Character Heuristic
- Good Suffix Heuristic

Both of the above heuristics can also be used independently to search a pattern in a text. Let us first understand how two independent approaches work together in the Boyer Moore algorithm. If we take a look at the Naive algorithm, it slides the pattern over the text one by one. KMP algorithm does preprocessing over the pattern so that the pattern can be shifted by more than one. The Boyer Moore algorithm does preprocessing for the same reason. It processes the pattern and creates different arrays for each of the two heuristics. At every step, it slides the pattern by the max of the slides suggested by each of the two heuristics. So it uses greatest offset suggested by the two heuristics at every step.
Unlike the previous pattern searching algorithms, the Boyer Moore algorithm starts matching from the last character of the pattern.

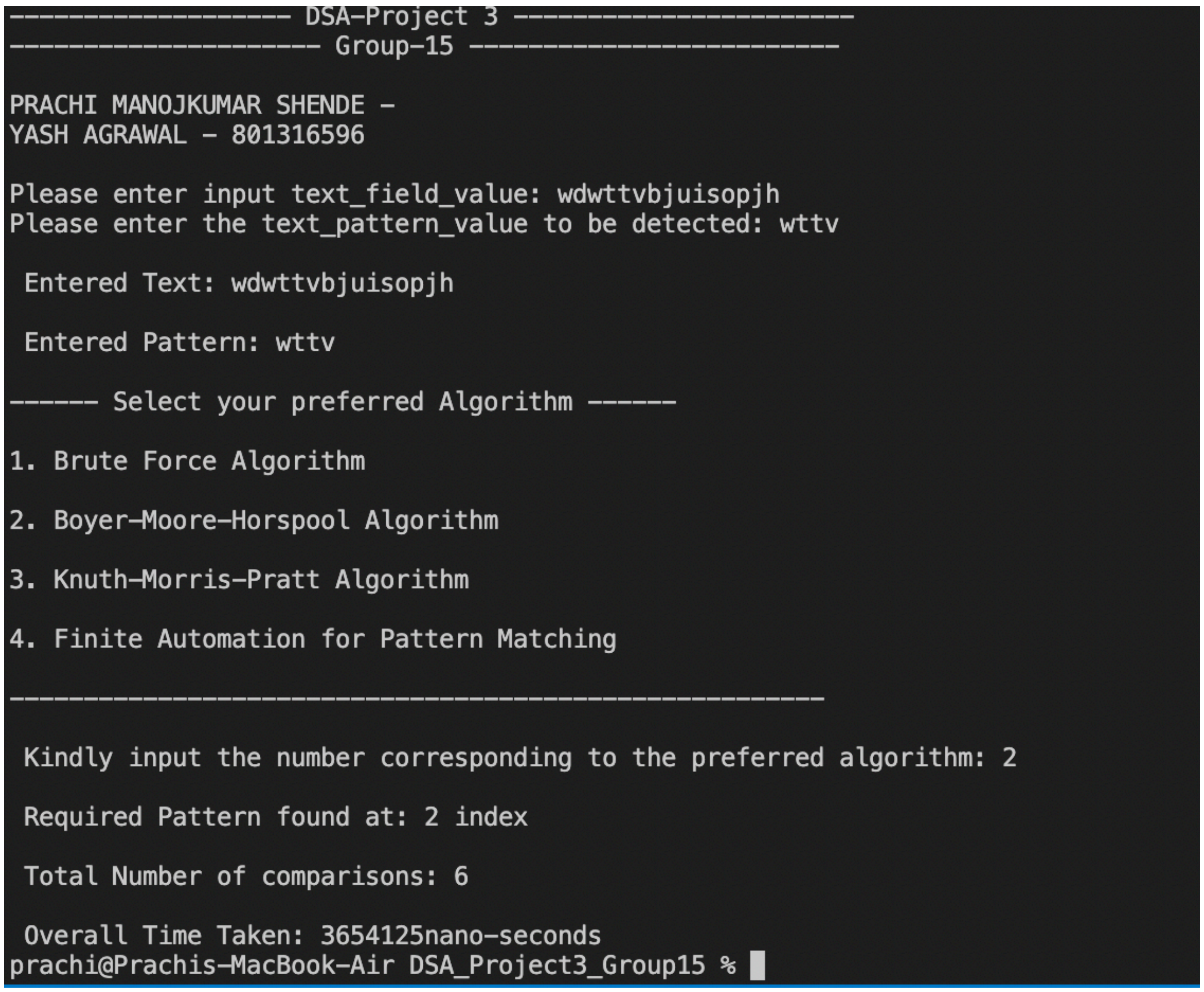
Algorithm

```
HorspoolMatching(T, P) ---------- --> O(nm) = Worst Case Ex. P = ba^{m-1} and T = a^n,
O(n/m) = Best Case Ex. P = b^m and T = a^n.

However for random texts it is O(n). Input: Pattern P and text T
Output: Index the left end of the first matching substring or -1 if no such substring exists

ShiftTable(P) // Generate Shift Table --------- --> O( |Σ| + m ) = O( S + m ) //

S = Size of Alphabet, m = Size of Pattern.
i = m -1 // Position of the Pattern's right end
        while i <= n-1 do ---------- --> O(n)
                k = 0 // Number of matched Characters
        while k <= m-1 and P[m-1-k] = T[i-k]

                do k=k+1
        if k = m
                return i – m + 1 else i = i + table[T[i]]

return -1
```

Complexity Analysis

- In the worst-case the performance of the Boyer-Moore-Horspool algorithm is O(mn), where m is the length of the substring and n is the length of the string.
- The average time is O(n). In the best case, the performance is sub-linear, and is, in fact, identical to Boyer-Moore original implementation.
- Regardless, the Boyer-Moore-Horspool is quicker and the internal loop is simpler than Boyer-Moore.

Input/Output



```
------------------- DSA-Project 3 -----------------------
--------------------- Group-15 -------------------------

PRACHI MANOJKUMAR SHENDE -
YASH AGRAWAL - 801316596

Please enter input text_field_value: wdwttvbjuisopjh
Please enter the text_pattern_value to be detected: wttv

 Entered Text: wdwttvbjuisopjh

 Entered Pattern: wttv

------ Select your preferred Algorithm ------

1. Brute Force Algorithm

2. Boyer-Moore-Horspool Algorithm

3. Knuth-Morris-Pratt Algorithm

4. Finite Automation for Pattern Matching

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

 Kindly input the number corresponding to the preferred algorithm: 2

 Required Pattern found at: 2 index

 Total Number of comparisons: 6

 Overall Time Taken: 3654125nano-seconds
prachi@Prachis-MacBook-Air DSA_Project3_Group15 %
```

TEST INPUT

Sr. No	Sample Test Input
1	QWERTYUIOPTRYWUIWROEIOFBFOOO /QWERTY
2	ZXCVBNMLKJHGFDSAPOI /BNMLKJH
3	ASDFGHJVBNZMCNHFYR /FGHJVBNZ
4	FGRTUQIEONSVCKFJHSGYUEO /QIEONSVCK