

CSE2003: DATA STRUCTURE AND ALGORITHMS
J COMPONENT PROJECT

SEARCH ALGORITHMS
USED BY BROWSERS
AND DATABASE.

SUBMITTED TO:
PROF. S.P MEENAKSHI

SUBMITTED BY-

RUCHIR BHASKAR	16BCE0190
PRASHANT MEHLAWAT	16BCE0910
ADITYA FIRODA	16BCE2184
(G2 SLOT)	

INTRODUCTION

IN THIS PROJECT WE ARE TRYING TO DEPICT THE ALGORITHMS USED BY BROWSERS AND MS WORD TO FIND A WORD IN A LARGE TEXT INPUT DATA. THE INPUT DATA WE ARE USING IS A NOVEL NAMED “THE KITE RUNNER” BY KHALED HOSSEINI. THERE ARE MANY TYPE OF ALGORITHMS AND DATA STRUCTURE USED FOR THIS PURPOSE ONLY. WE ARE DEPICTING INVERTED INDEX DATA STRUCTURE, BOYER MOORE SEARCH ALGORITHM AND RABIN-KARP SEARCH ALGORITHM.

MODULES COVERED IN THIS PROJECT ARE:

1. SEARCH USING INVERTED INDEX DATA STRUCTURE AND LINKED LISTS.
2. BOYER MOORE SEARCH ALGORITHM
3. RABIN-KARP SEARCH ALGORITHM

SEARCH USING INVERTED INDEX DATA STRUCTURE

An Inverted Index is a structure used by search engines and databases to make search terms to files or documents, trading the speed writing the document to the index for searching the index later on. There are two versions of an inverted index, a record-level index which tells you which documents contain the term and a fully inverted index which tells you both the document a term is contained in and where in the file it is. For example if you built a search engine to search the contents of sentences and it was fed these sentences:

{0} - "Turtles love pizza"

{1} - "I love my turtles"

{2} - "My pizza is good"

Then you would store them in a Inverted Indexes like this:

	Record Level	Fully Inverted
"turtles"	{0, 1}	{ (0, 0), (1, 3) }
"love"	{0, 1}	{ (0, 1), (1, 1) }
"pizza"	{0, 2}	{ (0, 2), (2, 1) }
"i"	{1}	{ (1, 0) }
"my"	{1, 2}	{ (1, 2), (2, 0) }
"is"	{2}	{ (2, 2) }
"good"	{2}	{ (2, 3) }

The record level sets represent just the document ids where the words are stored, and the fully inverted sets represent the document in the first number inside the parentheses and the location in the document is stored in the second number.

So now if you wanted to search all three documents for the words “my turtles” you would grab the sets (looking at record level only):

```
"turtles" {0, 1}
```

```
"my"      {1, 2}
```

Then you would intersect those sets, coming up with the only matching set being 1. Using the Fully Inverted Index would also let us know that the word “my” appeared at position 2 and the word “turtles” at position 3, assuming the word position is important your search.

There is no standard implementation for an Inverted Index as it’s more of a concept rather than an actual algorithm, this however gives you a lot of options.

For the index we can choose to use things like Hashtables, BTrees, or any other fast search data structure. **HERE WE HAVE ALSO USED LINKED LISTS.**

SOURCE CODE C/C++

```
#include <iostream>
#include <string>
#include <vector>
#include <map>
#include <fstream>
#include <sstream>
#include <cstdlib>
using namespace std;

typedef struct wordInfor      //save the word information
{
    string file;              //file name
    int line;                 //which line
    int index;                //index in the line
    wordInfor *next = NULL;   //point to next node
}wordInfor;

class InvertedIndex
```

```

{
public:
    void addFile(string fileName);
    void query(string word);
    const vector<string> &getFileList();
    ~InvertedIndex();

private:
    map<string, wordInfor*> wordDict;
    vector<string> fileList;
};

const vector<string> &InvertedIndex::getFileList()
{
    return fileList;
}

void InvertedIndex::addFile(string fileName)    //read from file, complete the wordDict
{
    string wd, line;
    ifstream in;
    in.open("xx.txt");
    if (in)
    {
        int recordConter = 0;
        while (getline(in, line))    //get every line
        {
            istringstream record(line);
            int indexConter = 0;

            while (record >> wd)    //get every word
            {
                if (wordDict.find(wd) == wordDict.end())
                {
                    wordDict[wd] = NULL;
                }
                wordInfor *n = new wordInfor();
                n->next = wordDict[wd];
                wordDict[wd] = n;
                n->file = fileName;
                n->line = recordConter;
                n->index = indexConter;

                indexConter++;
            }

            recordConter++;
        }
        fileList.push_back(fileName);
    }
}

```

```

        in.close();
    }
    else
        cout << "no such file" << endl;

}

void InvertedIndex::query(string word)
{
    wordInfor *cur = 0;

    if ( wordDict.find(word) == wordDict.end())
    {
        cout << "\"" << word << "\":" << endl;
        cout << "no such word";
        return;
    }
    else
        cur = wordDict[word];

    cout << "\"" << word << "\" occurred in:" << endl;
    while (cur)
    {
        cout << "file: " << cur->file << " line: " << cur->line << " index: " << cur->index << endl;
        cur = cur->next;
    }
}

InvertedIndex::~~InvertedIndex()
{
    for (auto it = wordDict.begin(); it != wordDict.end(); ++it)
    {
        wordInfor *q, *p = it->second;

        while (p)
        {
            q = p->next;
            delete p;
            p = q;
        }
    }
}

int main()
{
    InvertedIndex i;

    int choice;
    while (1)

```

```

{
    cout << "enter 0 to add file" << endl;
    cout << "enter 1 to query word" << endl;
    cout << "enter 2 to quit" << endl;
    const vector<string> &fileList = i.getFileList();
    if (!fileList.empty())
    {
        cout << "added files: ";
        for (auto f : fileList)
            cout << f << " ";
        cout << endl;
    }

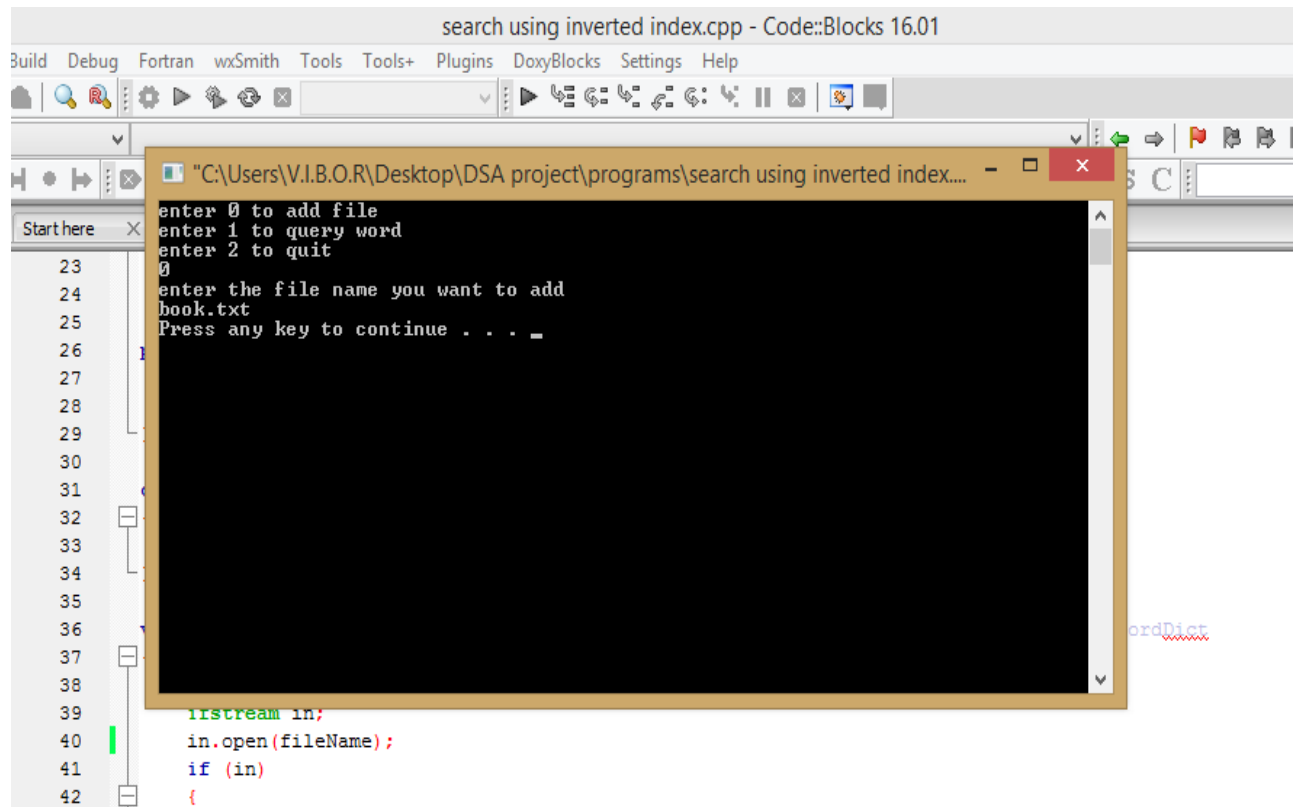
    cin >> choice;
    switch (choice)
    {
    case 0:
        {
            string file;
            cout << "enter the file name you want to add" << endl;
            cin >> file;
            i.addFile(file);
            break;
        }
    case 1:
        {
            string word;
            cout << "enter the word you want to query" << endl;
            cin >> word;
            i.query(word);
            break;
        }
    case 2:
        exit(0);
    default:
        {
            cout << "please enter 0 or 1 or 2" << endl;
        }
    }
    system("pause");
    system("CLS");
}
}

```

INPUT-

THE TEXT FILE OF BOOK “The kite runner” OF 10000 LINES OF DATA.

OUTPUT SCREENSHOTS-



```
search using inverted index.cpp - Code::Blocks 16.01
Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
"C:\Users\V.I.B.O.R\Desktop\DSA project\programs\search using inverted index...."
enter 0 to add file
enter 1 to query word
enter 2 to quit
0
enter the file name you want to add
book.txt
Press any key to continue . . . _
istream in;
in.open(fileName);
if (in)
{
```



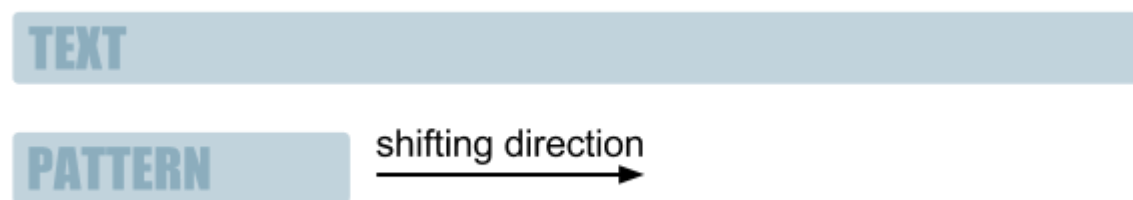
```
"C:\Users\V.I.B.O.R\Desktop\DSA project\programs\search using inverted index.... - [X] lex.cpp - Code::Blocks 10.05
enter 0 to add file
enter 1 to query word
enter 2 to quit
added files: book.txt
1
enter the word you want to query
jail
"jail":
no such wordPress any key to continue . . .
```

```
"C:\Users\V.I.B.O.R\Desktop\DSA project\programs\search using inverted index.... - [X] lex.cpp
enter 0 to add file
enter 1 to query word
enter 2 to quit
added files: book.txt
1
enter the word you want to query
runner
"runner" occurred in:
file: book.txt line: 8777 index: 11
file: book.txt line: 5053 index: 2
file: book.txt line: 1211 index: 3
file: book.txt line: 1206 index: 3
file: book.txt line: 1201 index: 11
Press any key to continue . . . _
```

BOYER MOORE SEARCH ALGORITHM

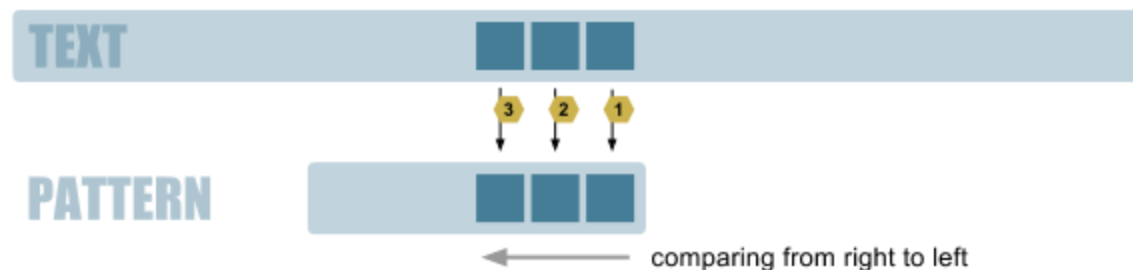
Boyer-Moore is an algorithm that improves the performance of pattern searching into a text by considering some observations.

First of all this algorithm starts comparing the pattern from the leftmost part of text and moves it to the right, as on the picture below.



In Boyer-Moore the pattern is shifted from left to right!

Unlike other string searching algorithms though, Boyer-Moore compares the pattern against a possible match from right to left as shown below.

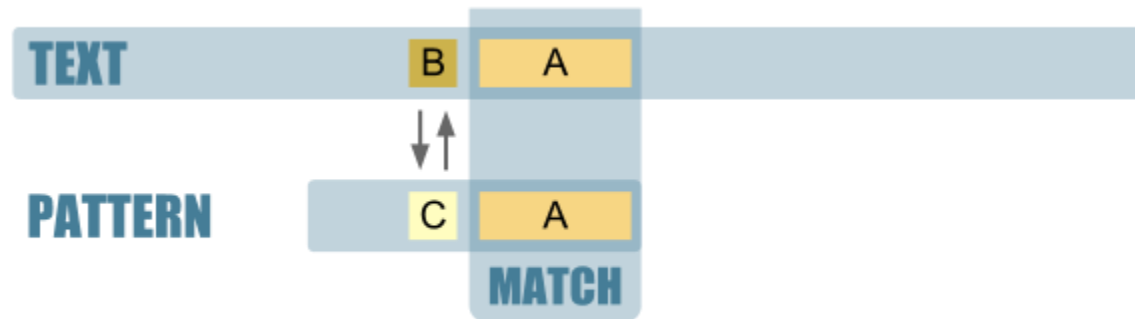


Unlike other algorithms the letters of the pattern are compared from right to left!

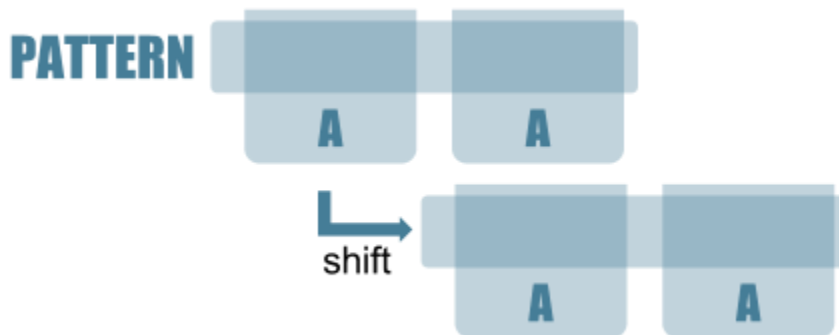
The main idea of Boyer-Moore in order to improve the performance are some observations of the pattern. In the terminology of this algorithm they are called good-suffix and bad-character shifts. Let's see by the following examples what they are standing for.

Good-suffix Shifts

We start to compare the pattern against some portion of the text where a possible match will occur. In Boyer-Moore this is done from the rightmost letter of the pattern. After some characters have matched we find a mismatch.



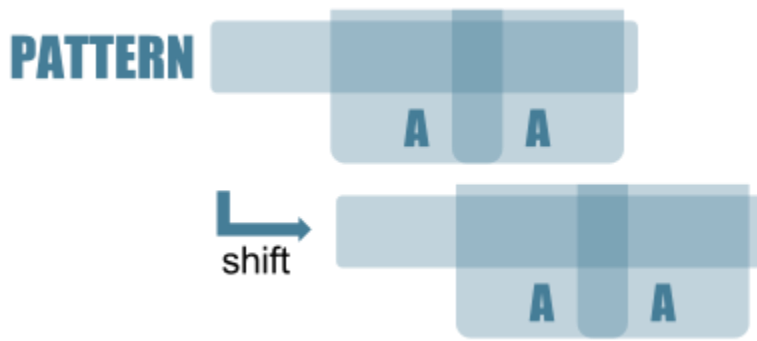
So how can we move the pattern to the right in order to skip unusual comparisons. To answer this question we need to explore the pattern. Let's say there is a portion of the pattern that is repeated inside the pattern itself, like it is shown on the picture below.



The pattern may consist of repeating portions of characters!

In this case we must move the pattern thus the repeated portion must now align with its first occurrence in the pattern.

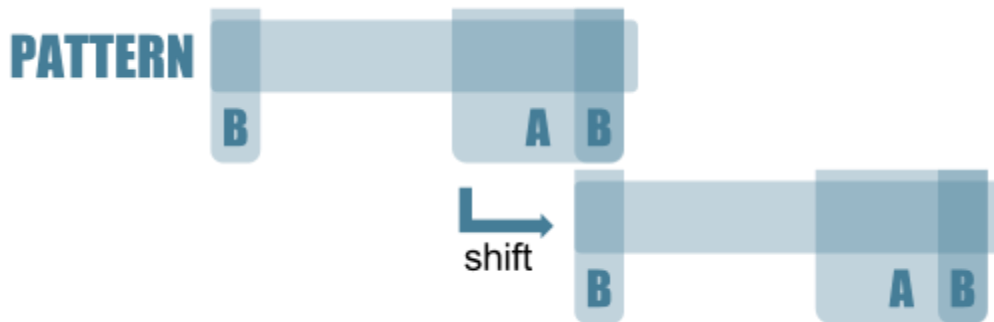
A variation of this case is when the portion from the pattern A overlaps with another portion that consists of the same characters.



Sometimes these portions may overlap!

Yet again the shift must align the second portion with its first occurrence.

Finally only a portion of A, let's say "B", can happen to occur in the very beginning of the pattern, as on the diagram below.



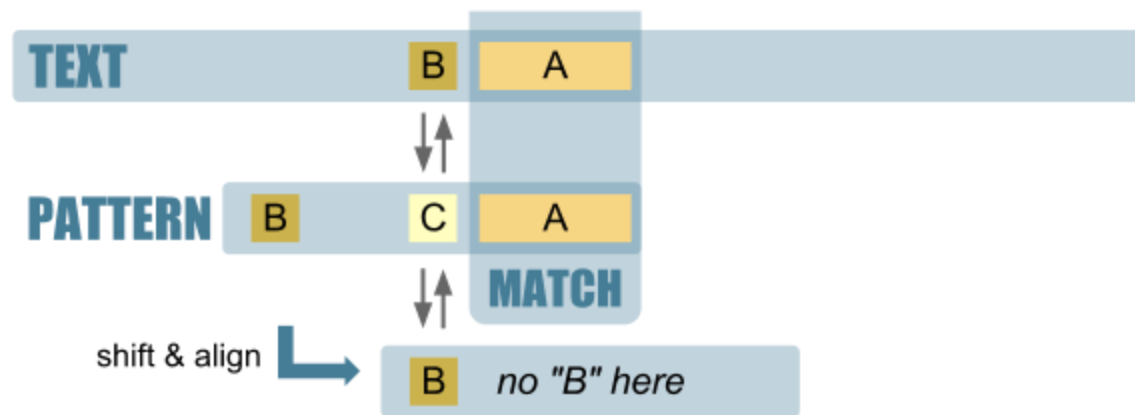
Only

a sub-string of the pattern may re-occur at its front!

Now we must align the left end of the pattern with the rightmost occurrence of "B".

Bad Character Shifts

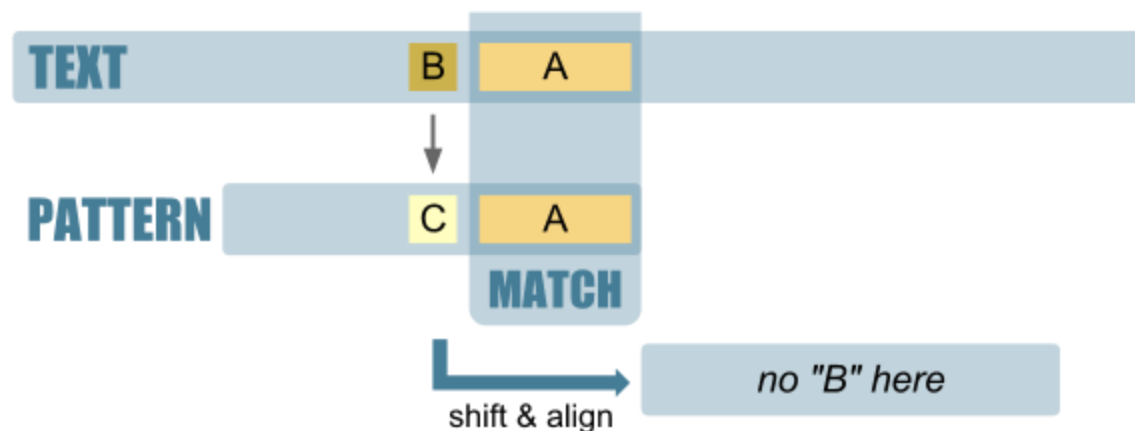
Beside the good-suffix shifts the Boyer-Moore algorithm make use of the so called bad-character shifts. In case of a mismatch we can skip comparisons in case the character in the text doesn't happen to appear in the pattern. To become clearer let's see the following examples.



If

the mismatched letter of the text appears in the pattern only in its front we can align it easily!

In the picture above we see that the mismatched character "B" from the text appears only in the beginning of the pattern. Thus we can simply shift the pattern to the right and align both characters B, skipping comparisons. An even better case is described by the following diagram where the mismatched letter isn't contained into the pattern at all. Then we can shift forward the whole pattern.



In

case the mismatched letter isn't contained into the pattern we move forward the pattern!

Maximum of Good-suffix and Bad-Character shifts

Boyer-Moore needs both good-suffix and bad-character shifts in order to speed up searching performance. After a mismatch the maximum of both is considered in order to move the pattern to the right.

ALGORITHM-

Input: Text with n characters and Pattern with m characters

Output: Index of the first substring of T matching P

1. Compute function last
2. $i \leftarrow m-1$
3. $j \leftarrow m-1$
4. Repeat
5. If $P[j] = T[i]$ then
6. if $j=0$ then
7. return i // we have a match
8. else
9. $i \leftarrow i - 1$
10. $j \leftarrow j - 1$
11. else
12. $i \leftarrow i + m - \text{Min}(j, 1 + \text{last}[T[i]])$
13. $j \leftarrow m - 1$
14. until $i > n - 1$
15. Return "no match"

SOURCE CODE C/C++

```
/* Program for Bad Character Heuristic of Boyer Moore String Matching Algorithm */
#include <limits.h>
#include <string.h>
#include <stdio.h>
#define NO_OF_CHARS 256
// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b) ? a : b;
}
// The preprocessing function for Boyer Moore's bad character heuristic
void badCharHeuristic(char *str, int size, int badchar[NO_OF_CHARS])
{
    int i;
    // Initialize all occurrences as -1
    for (i = 0; i < NO_OF_CHARS; i++)
        badchar[i] = -1;
    // Fill the actual value of last occurrence of a character
```

```

    for (i = 0; i < size; i++)
        badchar[(int) str[i]] = i;
}
void search(char *txt, char *pat)
{
    int c=0;
    int m = strlen(pat);
    int n = strlen(txt);
    int badchar[NO_OF_CHARS];
    badCharHeuristic(pat, m, badchar);
    int s = 0; // s is shift of the pattern with respect to text
    while (s <= (n - m))
    {
        int j = m - 1;
        while (j >= 0 && pat[j] == txt[s + j])
            j--;
        if (j < 0)
        {
            c=1;
            printf("\n pattern occurs at shift = %d", s);
            s += (s + m < n) ? m - badchar[txt[s + m]] : 1;
        }
        else
            s += max(1, j - badchar[txt[s + j]]);
    }
    if(c==0)
        printf("\nWord not found");
}

```

/* Driver program to test above function */

```
int main()
```

```
{
```

```
    char pat[100];
```

char txt[]="In molecular genetics, the three prime untranslated region (3'-UTR) is the section of messenger RNA (mRNA) that immediately follows the translation termination codon. An mRNA molecule is transcribed from the DNA sequence and is later translated into protein. Several regions of the mRNA molecule are not translated into protein including the 5' cap, 5' untranslated region, 3' untranslated region, and the poly(A) tail. The 3'-UTR often contains regulatory regions that post-transcriptionally influence gene expression. mRNA structure, approximately to scale for a human mRNA, where the median length of 3'UTR is 700 nucleotides Regulatory regions within the 3'-untranslated region can influence polyadenylation, translation efficiency, localization, and stability of the mRNA.[1][2] The 3'-UTR contains both binding sites for regulatory proteins as well as microRNAs (miRNAs). By binding to specific sites within the 3'-UTR, miRNAs can decrease gene expression of various mRNAs by either inhibiting translation or directly causing degradation of the transcript. The 3'-UTR also has silencer regions which bind to repressor proteins and will inhibit the expression of the mRNA. Many 3'-UTRs also contain AU-rich elements (AREs). Proteins bind AREs to affect the stability or decay rate of transcripts in a localized manner or affect translation initiation. Furthermore, the 3'-UTR contains the sequence AAUAAA that directs addition of several hundred adenine residues called the poly(A) tail to the end of the mRNA transcript. Poly(A) binding protein (PABP) binds to this tail, contributing to regulation of mRNA translation, stability, and export. For

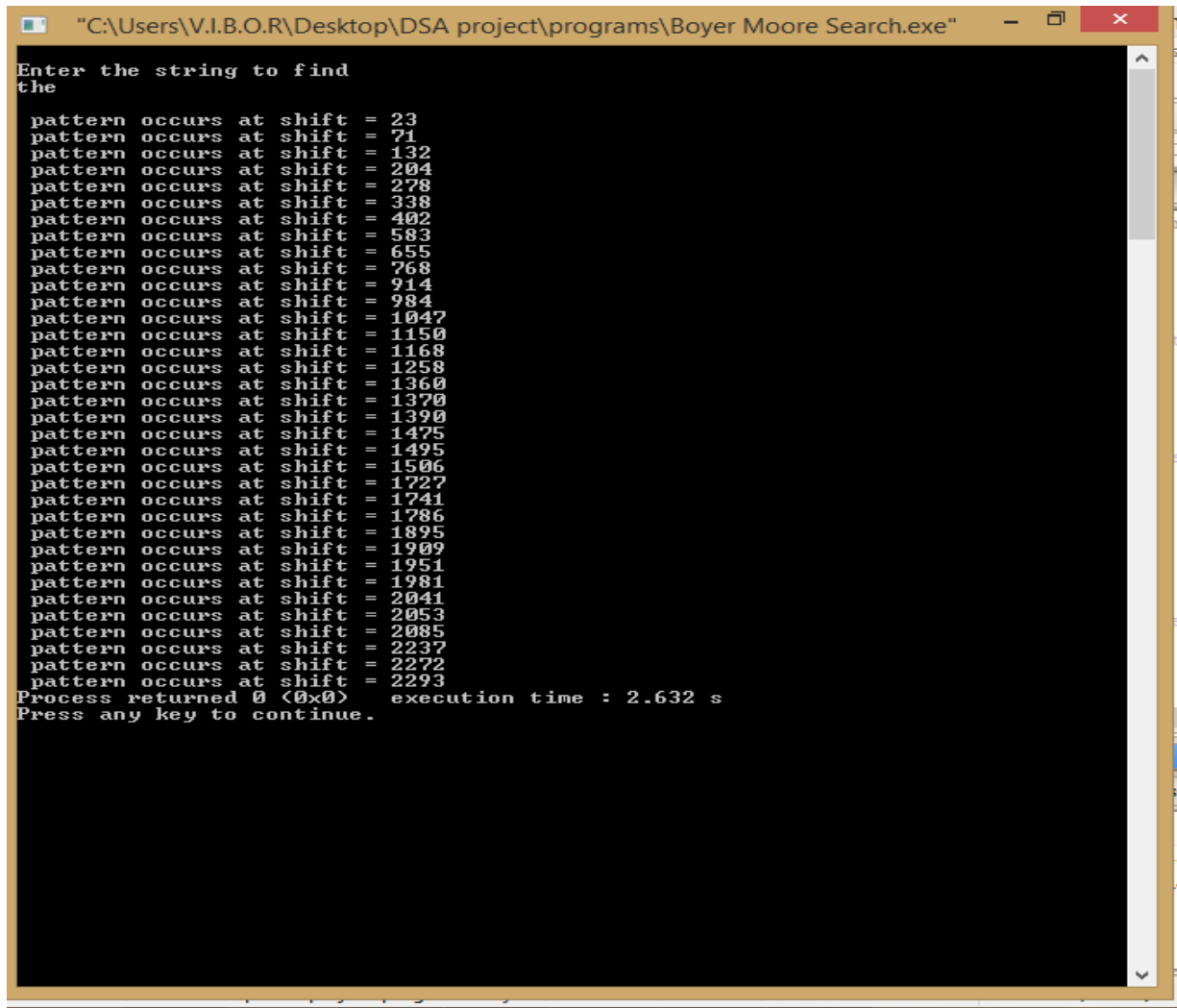
example, poly (A) tail bound PABP interacts with proteins associated with the 5' end of the transcript, causing a circularization of the mRNA that promotes translation. The 3'-UTR can also contain sequences that attract proteins to associate the mRNA with the cytoskeleton, transport it to or from the cell nucleus, or perform other types of localization. In addition to sequences within the 3'-UTR, the physical characteristics of the region, including its length and secondary structure, contribute to translation regulation. These diverse mechanisms of gene regulation ensure that the correct genes are expressed in the correct cells at the appropriate times.";

```
printf("\nEnter the string to find\n");
scanf("%s",&pat);
search(txt, pat);
return 0;
}
```

INPUT PARAGRAPH-

In molecular genetics, the three prime untranslated region (3'-UTR) is the section of messenger RNA (mRNA) that immediately follows the translation termination codon. An mRNA molecule is transcribed from the DNA sequence and is later translated into protein. Several regions of the mRNA molecule are not translated into protein including the 5' cap, 5' untranslated region, 3' untranslated region, and the poly(A) tail. The 3'-UTR often contains regulatory regions that post-transcriptionally influence gene expression. mRNA structure, approximately to scale for a human mRNA, where the median length of 3'UTR is 700 nucleotides Regulatory regions within the 3'-untranslated region can influence polyadenylation, translation efficiency, localization, and stability of the mRNA.[1][2] The 3'-UTR contains both binding sites for regulatory proteins as well as microRNAs (miRNAs). By binding to specific sites within the 3'-UTR, miRNAs can decrease gene expression of various mRNAs by either inhibiting translation or directly causing degradation of the transcript. The 3'-UTR also has silencer regions which bind to repressor proteins and will inhibit the expression of the mRNA. Many 3'-UTRs also contain AU-rich elements (AREs). Proteins bind AREs to affect the stability or decay rate of transcripts in a localized manner or affect translation initiation. Furthermore, the 3'-UTR contains the sequence AAUAAA that directs addition of several hundred adenine residues called the poly(A) tail to the end of the mRNA transcript. Poly(A) binding protein (PABP) binds to this tail, contributing to regulation of mRNA translation, stability, and export. For example, poly (A) tail bound PABP interacts with proteins associated with the 5' end of the transcript, causing a circularization of the mRNA that promotes translation. The 3'-UTR can also contain sequences that attract proteins to associate the mRNA with the cytoskeleton, transport it to or from the cell nucleus, or perform other types of localization. In addition to sequences within the 3'-UTR, the physical characteristics of the region, including its length and secondary structure, contribute to translation regulation. These diverse mechanisms of gene regulation ensure that the correct genes are expressed in the correct cells at the appropriate times

OUTPUT SCREENSHOTS-



```
"C:\Users\V.I.B.O.R\Desktop\DSA project\programs\Boyer Moore Search.exe"
Enter the string to find
the
pattern occurs at shift = 23
pattern occurs at shift = 71
pattern occurs at shift = 132
pattern occurs at shift = 204
pattern occurs at shift = 278
pattern occurs at shift = 338
pattern occurs at shift = 402
pattern occurs at shift = 583
pattern occurs at shift = 655
pattern occurs at shift = 768
pattern occurs at shift = 914
pattern occurs at shift = 984
pattern occurs at shift = 1047
pattern occurs at shift = 1150
pattern occurs at shift = 1168
pattern occurs at shift = 1258
pattern occurs at shift = 1360
pattern occurs at shift = 1370
pattern occurs at shift = 1390
pattern occurs at shift = 1475
pattern occurs at shift = 1495
pattern occurs at shift = 1506
pattern occurs at shift = 1727
pattern occurs at shift = 1741
pattern occurs at shift = 1786
pattern occurs at shift = 1895
pattern occurs at shift = 1909
pattern occurs at shift = 1951
pattern occurs at shift = 1981
pattern occurs at shift = 2041
pattern occurs at shift = 2053
pattern occurs at shift = 2085
pattern occurs at shift = 2237
pattern occurs at shift = 2272
pattern occurs at shift = 2293
Process returned 0 (0x0)   execution time : 2.632 s
Press any key to continue.
```



```
"C:\Users\V.I.B.O.R\Desktop\DSA project\programs\Boyer Moore Search.exe"
Enter the string to find
xxy
Word not found
Process returned 0 (0x0)   execution time : 4.585 s
Press any key to continue.
```

TIME COMPLEXITY-

- Complexity is $O(n)$. The execution time can actually be *sub-linear*: it doesn't need to actually check every character of the string to be searched but rather skips over some of them (check *right-most* character of the block of *m* *first*, if not found in pattern can skip entire rest of block).
- Best-case performance is $O(n/m)$. In the best case, only one in *m* characters needs to be checked.
- Worst case complexity is $O(m+n)$

ADVANTAGES-

- Boyer-Moore algorithm is extremely fast on large alphabet (relative to the length of the pattern).
- The payoff is not as for binary strings or for very short patterns.
- For the very shortest patterns, the naïve algorithm may be better.

IT IS FASTER THAN RABIN-KARP ALGORITHM

DISADVANTAGES-

- For binary strings Boyer-Moore algorithm is not recommended.

RABIN-KARP SEARCH ALGORITHM

Rabin–Karp algorithm seeks to speed up the testing of equality of the pattern to the substrings in the text by using a hash function. A hash function is a function which converts every string into a numeric value, called its *hash value*.

The key to the Rabin–Karp algorithm's performance is the efficient computation of hash values of the successive substrings of the text. The Rabin fingerprint is a popular and effective rolling hash function. The Rabin fingerprint treats every substring as a number in some base, the base being usually a large prime. For example, if the substring is "hi" and the base is 101, the hash value would be $104 \times 101^1 + 105 \times 101^0 = 10609$ (ASCII of 'h' is 104 and of 'i' is 105).

Technically, this algorithm is only similar to the true number in a non-decimal system representation, since for example we could have the "base" less than one of the "digits"

The essential benefit achieved by using a rolling hash such as the Rabin fingerprint is that it is possible to compute the hash value of the next substring from the previous one by doing only a constant number of operations, independent of the substrings' lengths.

For example, if we have text "abracadabra" and we are searching for a pattern of length 3, the hash of the first substring, "abr", using 101 as base is:

```
// ASCII a = 97, b = 98, r = 114.  
hash("abr") = (97 × 1012) + (98 × 1011) + (114 × 1010) = 999,509
```

We can then compute the hash of the next substring, "bra", from the hash of "abr" by subtracting the number added for the first 'a' of "abr", i.e. 97×101^2 , multiplying by the base and adding for the last a of "bra", i.e. 97×101^0 . Like so:

```
//           base   old hash   old 'a'           new 'a'  
hash("bra") = [1011 × (999,509 - (97 × 1012))] + (97 × 1010) =  
1,011,309
```

If the substrings in question are long, this algorithm achieves great savings compared with many other hashing schemes.

ALGORITHM-

1. $n = \text{Length}[T], m = \text{Length}[P]$
2. $h = d^{m-1} \bmod q$
3. $p = 0, to = 0$
4. for $i = 0$ to m
5. do $p = (d * p + P[i]) \bmod q$
6. to $= (d * to + T[i]) \bmod q$
7. For $s = 0$ to $n - m$
8. do if $p = to$
9. then if $P[1..m] == T[s+1..s+m]$
10. then "Pattern founds at shift " s
11. if $s < n - m$
12. then $ts+1 = (d(ts - T[s+1])h + T[s+m+1])$

SOURCE CODE C/C++

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <cstdlib>
using namespace std;
#define d 256
void search(char *pat, char *txt, int q)
{
    int c=0;
    int M = strlen(pat);
    int N = strlen(txt);
    int i, j;
    int p = 0;
    int t = 0;
    int h = 1;
    for (i = 0; i < M - 1; i++)
        h = (h * d) % q;
    for (i = 0; i < M; i++)
    {
        p = (d * p + pat[i]) % q;
        t = (d * t + txt[i]) % q;
    }
    for (i = 0; i <= N - M; i++)
    {
        if (p == t)
```

```

    {
        for (j = 0; j < M; j++)
        {
            if (txt[i + j] != pat[j])
                break;
        }
        if (j == M)
        {
            c=1;
            cout<<"Word found at index: "<<i<<endl;
        }
    }
    if (i < N - M)
    {
        t = (d * (t - txt[i] * h) + txt[i + M]) % q;
        if (t < 0)
            t = (t + q);
    }
}
if(c==0)
    cout<<"\nWord not found\n";
}
int main()
{

```

char txt[]="In molecular genetics, the three prime untranslated region (3'-UTR) is the section of messenger RNA (mRNA) that immediately follows the translation termination codon. An mRNA molecule is transcribed from the DNA sequence and is later translated into protein. Several regions of the mRNA molecule are not translated into protein including the 5' cap, 5' untranslated region, 3' untranslated region, and the poly(A) tail. The 3'-UTR often contains regulatory regions that post-transcriptionally influence gene expression. mRNA structure, approximately to scale for a human mRNA, where the median length of 3'UTR is 700 nucleotides Regulatory regions within the 3'-untranslated region can influence polyadenylation, translation efficiency, localization, and stability of the mRNA.[1][2] The 3'-UTR contains both binding sites for regulatory proteins as well as microRNAs (miRNAs). By binding to specific sites within the 3'-UTR, miRNAs can decrease gene expression of various mRNAs by either inhibiting translation or directly causing degradation of the transcript. The 3'-UTR also has silencer regions which bind to repressor proteins and will inhibit the expression of the mRNA. Many 3'-UTRs also contain AU-rich elements (AREs). Proteins bind AREs to affect the stability or decay rate of transcripts in a localized manner or affect translation initiation. Furthermore, the 3'-UTR contains the sequence AAUAAA that directs addition of several hundred adenine residues called the poly(A) tail to the end of the mRNA transcript. Poly(A) binding protein (PABP) binds to this tail, contributing to regulation of mRNA translation, stability, and export. For example, poly (A) tail bound PABP interacts with proteins associated with the 5' end of the transcript, causing a circularization of the mRNA that promotes translation. The 3'-UTR can also contain sequences that attract proteins to associate the mRNA with the cytoskeleton, transport it to or from the cell nucleus, or perform other types of localization. In addition to sequences within the 3'-UTR, the physical characteristics of the region, including its length and secondary structure, contribute to translation regulation. These diverse mechanisms of gene regulation ensure that the correct genes are expressed in the correct cells at the appropriate times.";

```

    char pat[100];

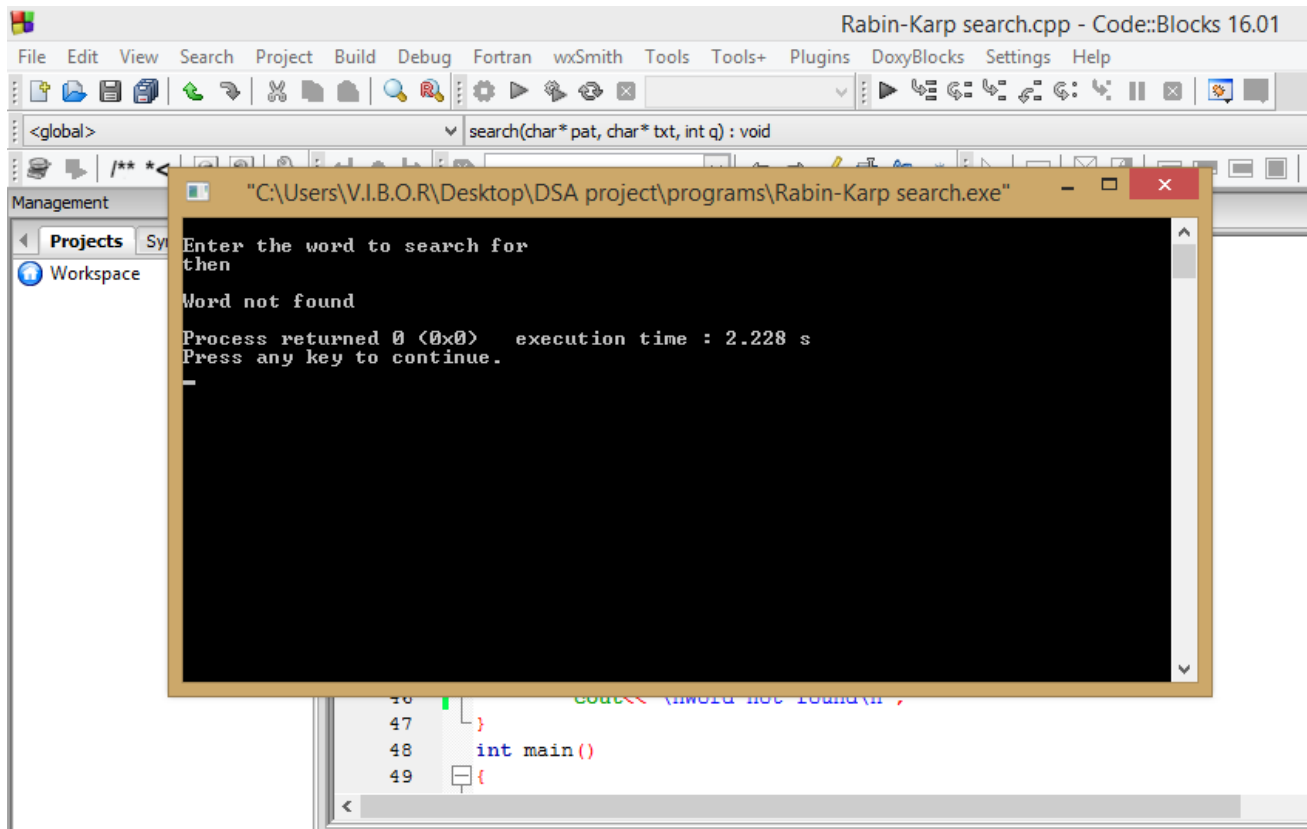
```

```
cout<<"\nEnter the word to search for\n";
cin>>pat;
int q = 100000;
search(pat, txt, q);
return 0;
}
```

INPUT PARAGRAPH-

In molecular genetics, the three prime untranslated region (3'-UTR) is the section of messenger RNA (mRNA) that immediately follows the translation termination codon. An mRNA molecule is transcribed from the DNA sequence and is later translated into protein. Several regions of the mRNA molecule are not translated into protein including the 5' cap, 5' untranslated region, 3' untranslated region, and the poly(A) tail. The 3'-UTR often contains regulatory regions that post-transcriptionally influence gene expression. mRNA structure, approximately to scale for a human mRNA, where the median length of 3'UTR is 700 nucleotides. Regulatory regions within the 3'-untranslated region can influence polyadenylation, translation efficiency, localization, and stability of the mRNA.[1][2] The 3'-UTR contains both binding sites for regulatory proteins as well as microRNAs (miRNAs). By binding to specific sites within the 3'-UTR, miRNAs can decrease gene expression of various mRNAs by either inhibiting translation or directly causing degradation of the transcript. The 3'-UTR also has silencer regions which bind to repressor proteins and will inhibit the expression of the mRNA. Many 3'-UTRs also contain AU-rich elements (AREs). Proteins bind AREs to affect the stability or decay rate of transcripts in a localized manner or affect translation initiation. Furthermore, the 3'-UTR contains the sequence AAUAAA that directs addition of several hundred adenine residues called the poly(A) tail to the end of the mRNA transcript. Poly(A) binding protein (PABP) binds to this tail, contributing to regulation of mRNA translation, stability, and export. For example, poly (A) tail bound PABP interacts with proteins associated with the 5' end of the transcript, causing a circularization of the mRNA that promotes translation. The 3'-UTR can also contain sequences that attract proteins to associate the mRNA with the cytoskeleton, transport it to or from the cell nucleus, or perform other types of localization. In addition to sequences within the 3'-UTR, the physical characteristics of the region, including its length and secondary structure, contribute to translation regulation. These diverse mechanisms of gene regulation ensure that the correct genes are expressed in the correct cells at the appropriate times.

OUTPUT SCREENSHOTS-



The screenshot shows the Code::Blocks IDE with the file "Rabin-Karp search.cpp" open. The console window is active, displaying the following output:

```
Enter the word to search for
then
Word not found
Process returned 0 (0x0)   execution time : 2.228 s
Press any key to continue.
```

The background shows the IDE's interface with the "Projects" and "Workspace" panels on the left and the source code editor at the bottom. The source code visible in the editor includes:

```
46     cout << "Word not found\n";
47 }
48 int main()
49 {
```

```
"C:\Users\V.I.B.O.R\Desktop\DSA project\programs\Rabin-Karp search.exe" - [X] [Help]
Enter the word to search for
the
Word found at index: 23
Word found at index: 71
Word found at index: 132
Word found at index: 204
Word found at index: 278
Word found at index: 338
Word found at index: 402
Word found at index: 583
Word found at index: 655
Word found at index: 768
Word found at index: 914
Word found at index: 984
Word found at index: 1047
Word found at index: 1150
Word found at index: 1168
Word found at index: 1258
Word found at index: 1360
Word found at index: 1370
Word found at index: 1390
Word found at index: 1475
Word found at index: 1495
Word found at index: 1506
Word found at index: 1727
Word found at index: 1741
Word found at index: 1786
Word found at index: 1895
Word found at index: 1909
Word found at index: 1951
Word found at index: 1981
Word found at index: 2041
Word found at index: 2053
Word found at index: 2085
Word found at index: 2237
Word found at index: 2272
Word found at index: 2293

Process returned 0 (0x0)   execution time : 5.135 s
Press any key to continue.
```


TIME COMPLEXITY-

- The running time of the Rabin-Karp algorithm in the worst-case scenario is $O((n-m+1)m)$ but it has a good average-case running time.
- If the expected number of valid shifts is small $O(1)$ and the prime q is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time $O(n+m)$ plus the time to required to process spurious hits.

DISADVANTAGES-

- Slow worst case behaviour.

APPLICATION-

- Bioinformatics- Used in looking for similarities of two or more proteins; i.e. high sequence similarity usually implies significant structural or functional similarity.
- Text processing.
- Compression.

CONCLUSION

WE HAVE SUCCESSFULLY IMPLEMENTED THE DIFFERENT TYPES OF SEARCH ALGORITHMS USED BY BROWSERS AND CERTAIN DATABASES LIKE GOOGLE.

WE ALSO HAVE IMPLEMENTED A NEW DATA STRUCTURE INVERTED INDEX AND HAVE COMPLETED THE FOLLOWING ALGORITHMS/ PROGRAMS AND HAVE COMPARED THEIR COMPLEXITY.

1. SEARCH USING INVERTED INDEX DATA STRUCTURE AND LINKED LISTS REPRESENTATION.
2. BOYER MOORE SEARCH ALGORITHM
3. RABIN-KARP SEARCH ALGORITHM