String Matching algorithms

A study

Literature survey + code

PES1UG20CS260 - Naman Kashyap
PES1UG20CS249 - Mohammed Ghouse

NOTE: *To skip theory and directly go to our testcases, results and observations, go to* **page 17**

1.0. Naive:

Theory:

It is the brute force approach to string matching. It iterates through the pattern for each substring (of the same length as the pattern) in the text.

NOTE: n is the length of the text, and m is the length of the pattern. This convention is followed throughout this document.

Time complexity: **O(n^2)**
Space complexity: **O(1)**

Implementation:

This algorithm is implemented using just 2 strings, and nested for/while loops to iterate through the text and pattern respectively.

Code:

```
int Naive(string text,string pattern,int &comparisons)
{
    int n = text.length();
    int m = pattern.length();
    for(int i = 0;i <= (n - m);++i)
    {
        int j;
        for(j = 0;j < m;++j)
        {
            ++comparisons;
            if(text[i + j] != pattern[j])
            {
```

```
                break;
            }
        }
        if(j == m)
        {
            return i;
        }
    }
    return -1;
}
```

SWOT analysis:

- Strengths: Simple and easy to implement. Constant memory.
- Weaknesses: As the name suggests, it is naïve. It is slow and not smart.
- Opportunities: Does the job well enough for small text and patterns. A good algorithm to start with.
- Threats: Can lead to Time Limit Exceeded in situations where the time complexity of the algorithm/code is of importance.

1.1. Accelerated naive:

Theory:

It almost feels like a hack to able to do string matching in **O(1)** time AND space complexities! And that too in such a simple manner! But in life things that are too good to be true often are too good to be true. The catch in this algorithm is that all the characters *must be unique*.

Time complexity: **O(n)**

Space complexity: **O(1)**

Implementation:

This algorithm is implemented using a for/while loop that iterates over the text once and finds the pattern match.

Code:

```
int acceleratedNaive(string text,string pattern,int &comparisons)
{
    int n = text.length();
    int m = pattern.length();
    int i = 0;
    int j = 0;
    while(i < n)
    {
        ++comparisons;
        if(pattern[j] == text[i])
        {
            if(j == (m - 1))
            {
                return i - (m - 1);
            }
            else
            {
                j += 1;
            }
        }
        else
        {
            j = 0;
        }
        i += 1;
    }
    return -1;
}
```

SWOT analysis:

- Strengths: Simple, fast, efficient for unique text.
- Weaknesses: Does not work, gives erroneous results for non-unique texts.

- Opportunities: Almost a “hack” for unique texts. A great lesson on simplicity in solving particular problems.

- Threats: If used for non-unique texts, it could lead to disaster.

2.Boyer-Moore:

Theory:

The Boyer–Moore algorithm searches for occurrences of *P* in *T* by performing explicit character comparisons at different alignments. Instead of a brute-force search of all alignments (of which there are n - m + 1).
Boyer–Moore uses information gained by pre-processing *P* to skip as many alignments as possible.
The key insight in this algorithm is that if the end of the pattern is compared to the text, then jumps along the text can be made rather than checking every character of the text.

Implementation:

Boyer Moore is a combination of the following 2 approaches
1. Good Suffix Heuristic
2. Bad Character Heuristic
In Naive Algorithm it slides the pattern over the text one by one.
Boyer Moore does pre-processing for the same reason.
It processes the pattern and creates different arrays for each of the 2 heuristics.
It shifts the pattern as suggested by each of the 2 heuristics. It takes the maximum of the two values given by the two heuristics.

Time complexity: **O(n) - average case , O(m*n) - worst case.**
Space complexity: **O(k) where k is the number of ASCII values.**

Code:

```
int boyerMoore(string text, string pattern,int &comparisons)
{
    // s is shift of the pattern with respect to text
    int s=0, j;
    int m = pattern.size();
    int n = text.size();

    int bpos[m+1], shift[m+1];

    for(int i=0;i<m+1;i++)
    {
        shift[i]=0;
    }

    //do preprocessing
    _preprocessStrongSuffix(shift, bpos, pattern, m,comparisons);
    _preprocessBadShift(shift, bpos, pattern, m);

    while(s <= n-m)
    {

        j = m-1;

        ++comparisons;
        while(j >= 0 && pattern[j] == text[s+j])
        {
            ++comparisons;
            j--;
        }

        if (j<0)
        {
            return s;
        }
        else
            s += shift[j+1];
    }
    return -1;
}
```

SWOT analysis:

- Strengths: Searching for patterns in linear time for English words.

- Weaknesses: Long patterns with repetitive letters

- Opportunities: Very useful for pattern searching because of pre-processing of patterns which increases its efficiency.

- Threats : The main drawback is pre-processing time and space required which depend on alphabet size / pattern size.

3.Rabin-Karp:

Theory:

Rabin–Karp algorithm is a string-matching algorithm created by Richard M. Karp and Michael O. Rabin (1987) that uses hashing to find an exact match of a pattern string in a text. It uses a *rolling hash* to quickly filter out positions of the text that cannot match the pattern, and then checks for a match at the remaining positions. Generalizations of the same idea can be used to find more than one match of a single pattern, or to find matches for more than one pattern.

Implementation:

It is implemented using the concept of hashing. The hash value of the pattern is calculated using an appropriate hash function. Then a window of the text (L to R) is taken and the hash value of that substring of the text is calculated. If there is a match, then each character in the pattern is matched with each character of the substring.

The "*sliding window*" approach is used, and hence the concept of "*rolling hash*" is appropriated to find new hash values efficiently as the window slides.
Time complexity: **O(m + n) - average case , O(m*n) - worst case.**
Space complexity: **O(1).**

Code:

```
int rabinKarp(string text,string pattern,int q,int &comparisons)
{
    int m = pattern.size();
    int n = text.size();
    int i, j;
    long long p = 0; // hash value for pattern
    long long t = 0; // hash value for text
    int h = 1;

    for (i = 0; i < m - 1; i++)
    {
        h = (h * d) % q;
    }

    for (i = 0; i < m; i++)
    {
        p = (d * p + pattern[i]) % q;
        t = (d * t + text[i]) % q;
    }

    for(i = 0;i <= n - m;++i)
    {
        ++comparisons;
        if(p == t)
        {
            for(j = 0;j < m;++j)
            {
                ++comparisons;
                if(text[i + j] != pattern[j])
                {
                    break;
                }
            }
            if(j == m)
            {
                return i;
            }
```

```
        }
        if(i < (n - m))
        {
            t = (d*(t - text[i] * h) + text[i + m]) % q;
            if(t < 0)
            {
                t += q;
            }
        }
    }
    return -1;
}
```

SWOT analysis:

- Strengths: Average case time complexity is O(m + n) and space complexity is O(1).
- Weaknesses: If we use a bad hash function it is not efficient and may work as naive.
- Opportunities: It is used for searching / matching patterns in a text using a hash function. It does not travel through every character in the initial phase, rather it filters the characters that do not match and performs the comparison.
- Threats: Bad hash function can make Rabin Karp work as a Naive algorithm with more mathematical operations.

4.KMP:

Theory:

The KMP algorithm searches for occurrences of a word within a main string by employing the observation that when a mismatch

occurs, the word itself embodies sufficient information to determine where the next match could begin.
The algorithm was conceived by James H Morris and independently discovered by Donald Knuth a few weeks later from automata theory.

Implementation:

This algorithm uses the degenerating property of pattern and improves the worst-case complexity.
The basic idea is whenever we detect a mismatch, we already know some of the characters in the text in the next window.
We take advantage of this information to avoid matching the characters that we know will lead to mismatch.

Time complexity: **O(m + n) - average case , O(m + n) - worst case.**
Space complexity: **O(m).**

Code:

```
int kmp_match(string text,string pattern,int &comparisons)
{
    int n = text.length();
    int m = pattern.length();
    vector<int> prefix = _kmpPrefixFunction(pattern,comparisons);
    int j = 0;
    text = ' ' + text;
    pattern = ' ' + pattern;
    for(int i = 1;i <= n;++i)
    {
        ++comparisons;
        while((j > 0) && (pattern[j + 1] != text[i]))
        {
            ++comparisons;
            j = prefix[j];
        }
        ++comparisons;
        if(pattern[j + 1] == text[i])
        {
            j = j + 1;
        }
```

```
        if(j == m)
        {
            return (i - m);
        }
    }
    return -1;
}
```

SWOT analysis:

- Strengths: very fast when compared to other algorithms, worst case efficiency is also linear time

- Weaknesses: does not work well when the size of the alphabet increases.
  For processing very large files it requires resources in the form of processors that could be a problem for smaller organizations.

- Opportunities: DNA Analysis, pattern matching done in long strings.

- Threats: pre-processing time is a lot. Hence will not be that good for long texts without much repetitive characters

5.Finite State Machine:

Theory:

A finite-state machine (FSM) or finite-state automaton is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some inputs; the change from one state to another is

called a transition. An FSM is defined by a list of its states, its initial state, and the inputs that trigger each transition. Finite-state machines are of two types—deterministic finite-state machines and non-deterministic finite-state machines. A deterministic finite-state machine can be constructed equivalent to any non-deterministic one.

Implementation:

An FSM must be created for the given pattern. The text is passed as input to the FSM which either accepts or rejects the text.
Time complexity: **O(m^3 x NO_OF_CHARS + n)**
Space complexity: **O(m).**

Code:

```
int fsm(string text, string pattern,int &comparisons)
{
    int M = pattern.size();
    int N = text.size();

    int TF[M+1][NO_OF_CHARS];

    _computeTF(pattern, M, TF,comparisons);

    int i, state=0;
    for (i = 0; i < N; i++)
    {
        state = TF[state][text[i]];
        if (state == M)
        {
            return (i - M + 1);
        }
    }
    return -1;
}
```

SWOT analysis:

- Strengths: Logically easy to visualize and construct.

- Weaknesses: An FSM is unique to a pattern. Pre-processing time is hence, high.

- Opportunities: Can be used to physically implement in Automata related projects.

- Threats: Not really practical in code.

6.0. Suffix Trie:

Theory:

Tries are an extremely special and useful data-structure that are based on the *prefix of a string*. They are used to represent the "Retrieval" of data and thus the name Trie.
Until now, our indexes have been based on extracting substrings from T. A very different approach is to extract suffixes from T.
Each path from root to leaf represents a suffix; each suffix is represented by some path from root to leaf

Implementation:

1. Generate all suffixes of a given text.
2. Consider all suffixes as individual words and build a trie.

Time complexity: **O(m)**
Space complexity: **O(n*n).**

Code:

```
list<int>* _SuffixTrieNode::_Search(string s)
{
    // If all characters of pattern have been processed,
    if (s.length() == 0)
        return indexes;
```

```
    // if there is an edge from the current node of suffix trie,
    // follow the edge.
    if (children[s.at(0)] != NULL)
        return (children[s.at(0)])->_Search(s.substr(1));

    // If there is no edge, pattern doesn't exist in text
    else
        return NULL;
}

int SuffixTrie::_search(string pattern)
{
    // Let us call recursive search function for root of Trie.
    // We get a list of all indexes (where pat is present in text) in
    // variable 'result'
    list<int> *result = root._Search(pattern);

    // Check if the list of indexes is empty or not
    if (result == NULL)
    {
        return -1;
    }
    else
    {
        list<int>::iterator i;
        int patLen = pattern.length();
        for (i = result->begin(); i != result->end(); ++i)
        {
            return (*i - patLen);
        }
        return -1;
    }
}
```

SWOT analysis:

- Strengths: O(m) time complexity
- Weaknesses: Not an in-place algorithm, takes a lot of memory to store all the suffixes.

- Opportunities : string matching algorithms, find out length of longest substring that appears at least twice in text, Implementing Dictionary.

- Threats: Not an in-place algorithm. Good only for searching multiple patterns on same text.

6.1. Suffix Tree:

Theory:

A suffix tree is a compressed trie containing all the suffixes of a given text as the keys and positions in the text as their values.
It allows fast implementation of many important string algorithms.
The construction of such a tree for the string S takes time and space linear in the length of S. Once constructed several operations can be performed very quickly for instance locating a substring in S, locating matches for a regular expression, pattern etc.

Implementation:

1. Generate all suffixes of a given text.
2. Consider all suffixes as individual words and build a compressed trie.

Time complexity: **O(m) (average case and worst case).**
Space complexity: **O(m).**

SWOT analysis:

- Strengths: Pattern matching in O(m) time, space complexity is O(m).
- Weaknesses: Not an in-place algorithm
- Opportunities: Computational Biology, free text search, text editing etc.
- Threats: Not an in-place algorithm, Complex in terms of implementation. Good only for searching multiple patterns on same text.

Testcases:

General format:

- Every input file is a large file with every 2 lines as:
  text
  pattern
- Every file is 7,550 lines long i.e., 3,775 testcases in total in every category, which progress from *best case* to *worst case* for each text.

We had 4 categories of testcase files:

1. English words: *A 7,550 line file with 50-100 words in each text and patterns chosen as 1st to last words in the text followed by a "NOT FOUND" case.*

Ex:

Text : count blood heat consider play

Pattern : count
Pattern : blood
.
.
.
Pattern : play
Pattern : pattern *(NOT FOUND case)*

2. Binary strings: *A 7,550 line file with 250-500 character long binary strings in each text and patterns ranging from 5 to 20 character long patterns (taken as a sliding window of text) followed by a "NOT FOUND" case.*

Ex:

Text : 00100110111011011

Pattern : 0010011011 *(pattern of length 5-20)*
Pattern : 11011 *(shift by 5 every time)*
.
.
.
Pattern : 0000O *(NOT FOUND case)*

3. DNA sequences: *A 7,550 line file with 250-500 character long DNA sequences in each text and patterns ranging from 5 to 20 character long*

patterns (taken as a sliding window of text) followed by a "NOT FOUND" case.

Ex:

Text : AGCTGACTAGCTTGACCTA

Pattern : AGCTGACT *(pattern of length 5-20)*
Pattern : ACTAG *(shift by 5 every time)*
.
.
.
Pattern : ACGTB *(NOT FOUND case)*

4. Random characters: *A 7,550 line file with 250-500 character long random ASCII character sequences in each text and patterns ranging from 5 to 20 character long patterns (taken as a sliding window of text) followed by a "NOT FOUND" case.*

Ex:

Text : "Cy4(P@39PhG]Y1[q5B(55fNU

Pattern : "Cy4(P@39Ph *(pattern of length 5-20)*
Pattern : P@39P *(shift by 5 every time)*
.
.
.
Pattern : ***** *(NOT FOUND case)*

Observations:

Our program runs the following algorithms:

- Naïve
- Boyer Moore
- Rabin Karp
- Finite State Machine
- Knuth-Morris-Pratt

On every category's testcase file and times how *long* each algorithm takes to execute, and the number of key comparisons it makes. This entire process is repeated 10 times to get an average runtime for *each algorithm* - for *each category*, and to minimize the effect of variable runtimes.
So that is 3,775 testcases in each category. (7,550) lines of input.
4 categories.
10 iterations.

We have used 3 + 1 files:

1. HEADER.h
2. server.cpp
3. driver.cpp

\+

makefile.mk to compile and run all of the above.

We stored the output of the driver in driverOutput.txt, which has outputs of all 10 iterations.

Given below is the output of 1 such iteration:

Iteration 1:
English:

Naive	: 37656 ns	881216 comparisons
Boyer Moore	: 30589 ns	764562 comparisons
Rabin Karp	: 29282 ns	853593 comparisons
FSM	: 664137 ns	20433078 comparisons
KMP	: 19216 ns	1753930 comparisons

Binary:

Naive	: 28027 ns	1199237 comparisons
Boyer Moore	: 15623 ns	460882 comparisons
Rabin Karp	: 31247 ns	639970 comparisons

FSM : 2166165 ns 100839425 comparisons
KMP : 31247 ns 1600768 comparisons

DNA:
Naive : 31247 ns 1001637 comparisons
Boyer Moore : 15623 ns 552740 comparisons
Rabin Karp : 15623 ns 771021 comparisons
FSM : 2221174 ns 102458635 comparisons
KMP : 43048 ns 1796766 comparisons

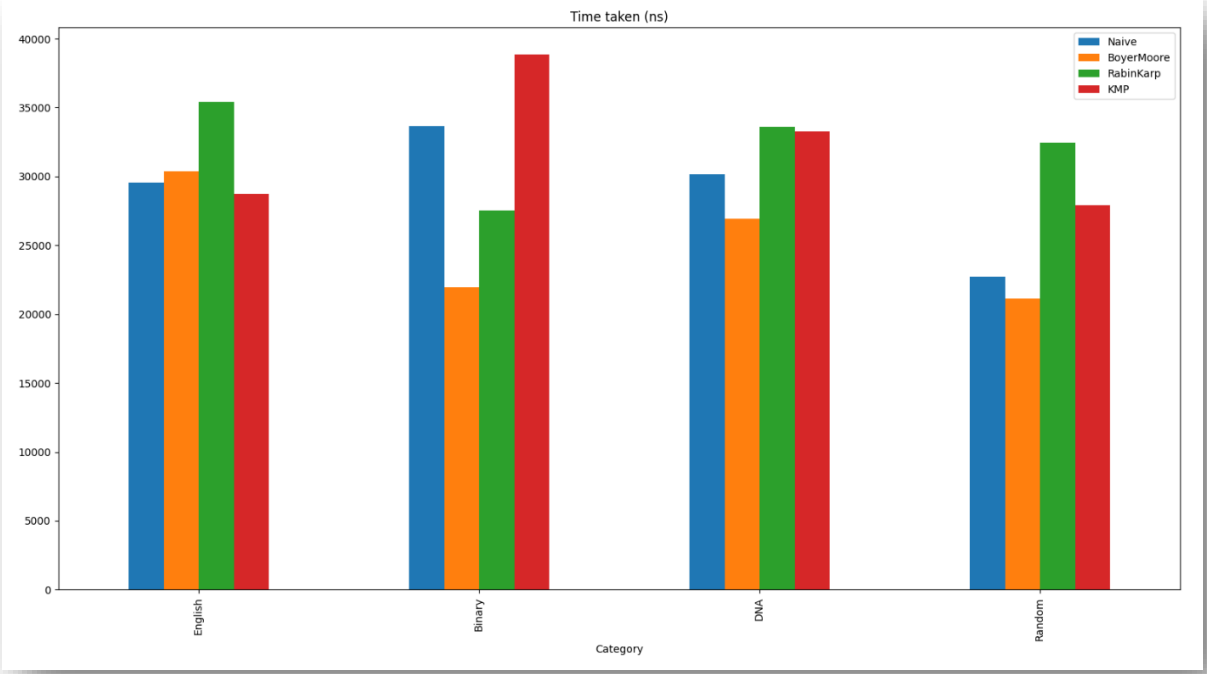
Random:
Naive : 13209 ns 788812 comparisons
Boyer Moore : 15726 ns 760919 comparisons
Rabin Karp : 31513 ns 781854 comparisons
FSM : 2203068 ns 101639004 comparisons
KMP : 15624 ns 1649294 comparisons

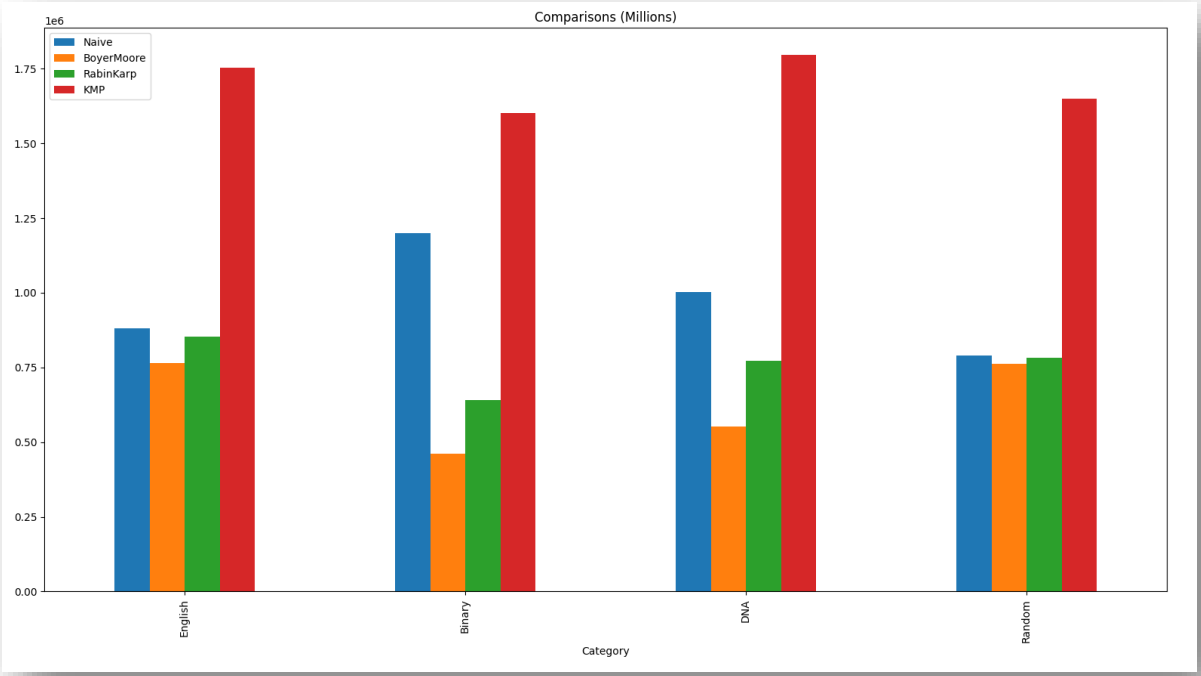
To better understand and visualize the performances of these algorithms we used matplotlib to construct grouped bar graphs of the average runtimes.

Here are the outputs:

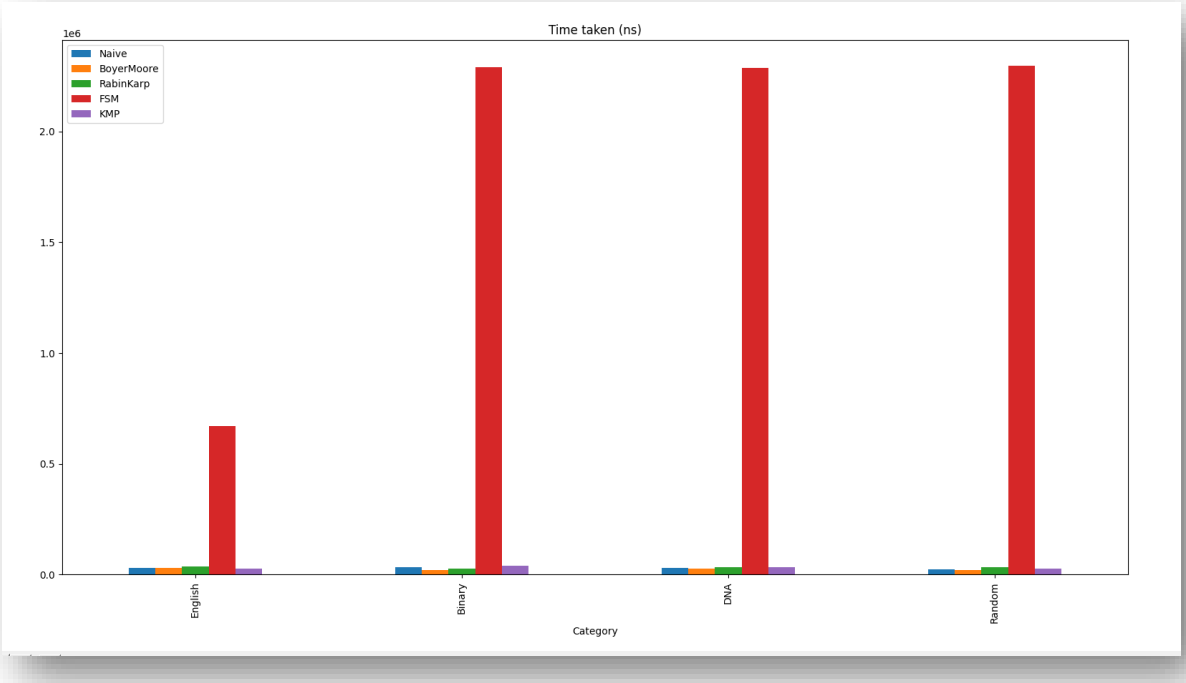
```
PS D:\NK\PES CS Engineering\Sem V\AA\Project\Code> python .\graphicalRepresentation.py
Time taken:
  Category     Naive  BoyerMoore  RabinKarp       KMP
0  English  29576.55    30358.60   35386.05  28719.35
1   Binary  33667.30    21978.20   27518.80  38862.80
2      DNA  30182.30    26951.25   33592.90  33256.85
3   Random  22730.80    21122.70   32459.90  27925.70

Comparisons:
  Category    Naive  BoyerMoore  RabinKarp      KMP
0  English   881216      764562     853593  1753930
1   Binary  1199237      460882     639970  1600768
2      DNA  1001637      552740     771021  1796766
3   Random   788812      760919     781854  1649294
```


Time taken (ns)
Naive
BoyerMoore
RabinKarp
KMP
40000
35000
30000
25000
20000
15000
10000
5000
0
English
Binary
DNA
Random
Category


Comparisons (Millions)
1e6
Naive
BoyerMoore
RabinKarp
KMP
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
English
Binary
DNA
Random
Category

NOTE: *FSM was removed as it was not comparable to the rest as we can clearly see in the image below.*



Result:

- As we can see in the above graphs, naïve, though simple, is not the most efficient at doing the job.
- Because our pattern length was not very long, naïve still did a decent job. But performances of better algorithms such as KMP, Rabin Karp was not too reflective of how much better they are.
- Even though our testcases were broad, we can still see that Boyer-Moore, Rabin-Karp and KMP have performed well.
- Boyer-Moore seems to find the right balance between preprocessing time and simplicity as it has the overall best performance.
- The performance of FSM was extremely bad, thus proving that it is a good theoretical concept, but not very useful in real-life scenarios.
- FSM string matching, in fact, held us back from increasing the pattern length to better test the other algorithms. The pre-processing time for FSM proved to be just too much for it to be practical.
- We did not test accelerated naïve and suffix tries and trees since they were not at all practical for our test cases.

Scope for future work:

More rigorous testing of Boyer-Moore, Rabin Karp and Knuth-Morris-Pratt with longer patterns and bigger testcases with more categories.