CSE 5301-009
Project - String Matching Algorithms

Ravi Rajpurohit - 1002079916
Vedanti Ambulkar - 1001829121

Objective

We are trying to implement the String Matching Algorithms and compare their performances in order to determine which algorithm is better in terms of time taken for computation. String matching or Pattern Searching is when we have to find a substring in the main string.
Pattern searching is one of the important problems in computer science. Pattern Searching algorithms are used when we search for a string in a text file or browser or database.
For example - The main string can be "*ABCDEFGHIJKLMNOP*" and we have to see if this string contains a "*GHIJK*" substring.

We will be using the following String Matching algorithms -

- Knuth-Morris-Pratt Algorithm
- Rabin-Karp Algorithm
- Boyer-Moore Algorithm
- Naive Algorithm

Algorithms

Naive Algorithm for string matching is a character-by-character comparison of two strings. Many early computer programs that implemented basic file-searching features used this unsophisticated technique. The strings are compared character by character, and the method ends when a mismatch is discovered. Due to its slowness and memory consumption, this method of string matching is not appropriate. This is particularly inefficient because there are a huge number of strings in a text but relatively few characters in the search query.

Pseudocode -

```
SEARCH (Text, Pattern)
 1. n ← length [Text]
 2. m ← length [Pattern]
 3. for s ← 0 to n -m
 4. do if Pattern [1.....m] == Text [s + 1....s + m]
 5. print "Pattern occurs with shift" s
```

Time Complexity - O(N^2)

Knuth-Morris-Pratt Algorithm is an algorithm for string matching with linear time complexity. It was published by Donald Knuth, James Morris, and Vaughan Pratt in 1977. We compare the pattern with the main string character by character. Unlike the naive approach, we do not backtrack the main string if there is a character mismatch. Only the pattern string is backtracked. This reduces the time complexity of the algorithm.

To make this happen, we create the longest prefix suffix array for the pattern. We assign numbers to the repetition in the pattern string so that if there is a character mismatch while matching in the middle of the string, we just go back to its assigned number index.

Pseudocode -

```
LPS (Pattern)
 1. m ←length [Pattern]
 2. arr [1] ← 0
 3. k ← 0
 4. for q ← 2 to m
 5. do while k > 0 and Pattern [k + 1] ≠ Pattern [q]
 6. do k ← arr [k]
 7. If Pattern [k + 1] = Pattern [q]
 8. then k← k + 1
 9. arr [q] ← k
 10. Return arr
```

```
SEARCH (Text, Pattern)
 1. n ← length [Text]
 2. m ← length [Pattern]
 3. arr← LPS (Pattern)
 4. q ← 0                  // numbers of characters matched
 5. for i ← 1 to n         // scan S from left to right
 6. do while q > 0 and Pattern [q + 1] ≠ Text [i]
 7. do q ← arr [q]         // next character does not match
 8. If Pattern [q + 1] = Text [i]
 9. then q ← q + 1         // next character matches
 10. If q = m              // is all of p matched?
 11. then print "Pattern occurs with shift" i - m
 12. q ← arr [q]           // look for the next match
```

Time Complexity - O(m+n) {m is the size of the pattern, n is the size of the main string}

Rabin-Karp Algorithm is an algorithm for string matching algorithm created by Richard M. Karp and Michael O. Rabin in 1987 that uses hashing to find a match of a pattern in a text. Similar to the Naive Algorithm, the Rabin-Karp algorithm too slides the pattern one by one, but matches the hash value of the pattern with the hash value of the current substring from the main string. If the hash matches, it starts to match individual characters.
So, the hash needs to be calculated for Pattern as well as the substrings from the main string in Rabin Karp's algorithm.

Pseudocode -
SEARCH (Text, Pattern, d, q)
1. n ← length [T]
2. m ← length [P]
3. h ← dm-1 mod q
4. p ← 0
5. t0 ← 0
6. for i ← 1 to m
7. do p ← (dp + P[i]) mod q
8. t0 ← (dt0+T [i]) mod q
9. for s ← 0 to n-m
10. do if p = ts
11. then if P [1.....m] = T [s+1.....s + m]
12. then "Pattern occurs with shift" s
13. If s < n-m
14. then ts+1 ← (d (ts-T [s+1]h)+T [s+m+1])mod q

Time Complexity - O(mn) + O(m) (preprocessing time)
{m is the size of the pattern, n is the size of the main string}

Boyer-Moore Algorithm is an algorithm for string matching created by Robert S. Boyer and J Strother Moore in 1977. Like the KMP algorithm, the Boyer Moore algorithm also preprocesses the pattern for the same reason. Preprocessing is done by two heuristics - Bad Character Heuristic and Good Suffix Heuristic. For this project, we have used the Bad Character Heuristic for preprocessing. The pattern is preprocessed and at every step, the pattern is slid by the number of slides suggested by the Bad Character Heuristic. Unlike other approaches, the Boyer Moore algorithm starts matching from the last character of the pattern.
The notion of the bad character heuristic is that the character of the text that does not match with the current character of the pattern is called the Bad Character. The pattern is shifted on mismatch until the mismatch becomes a match or the Pattern moves past the mismatched character.

Pseudocode -
bad_character_heuristic(Pattern, bad_character_array):
Input − Pattern, that needs to be searched
Output - bad character array for use in future

```
m = length(Pattern)
for all entries of bad_character_array, do
   set all entries to -1
done

for all characters of the Pattern, do
   set last position of each character in bad_character_array.
```

searchPattern(Pattern, text):
Input − Pattern, main string
Output − the index where the Pattern is found

```
patLen = length of Pattern
strLen = length of text
call bad_character_heuristic(Pattern, bad_character_array)
shift = 0

while shift <= (strLen - patLen), do
  j = patLen -1
  while j >= 0 and Pattern[j] = text[shift + j]
    decrease j by 1
  done
  if j < 0, then
    print the shift as, there is a match
    if shift + patLen < strLen, then
      shift= shift + patLen – bad_character_array[text[shift + patLen]]
    else
      increment shift by 1
  else
    shift = shift + max(1, j-bad_character_array[text[shift+j]])
```

Time Complexity -
O(m) preprocessing + O(mn) matching
{m is the size of the pattern, n is the size of the main string}

Time Complexity Comparison
Here is a comparison of theoretical time complexity for the above-discussed algorithms.
{m is the size of the pattern, n is the size of the main string}

Name	**Naive**	**Knuth-Morris-Pratt**	**Rabin-Karp**	**Boyer-Moore**
Time Complexity	*O(n^2)*	*O(m+n)*	*O(mn) + O(m)*	*O(mn) + O(m)*

The theory suggests that the Knuth-Morris-Pratt algorithm is the best for string matching among the other mentioned algorithms. Let us find out real-world result in our experiments.

Working

One of the ways to determine if an algorithm is better than other is the time taken for execution. *The less time, the better the algorithm.*

We generated synthetic datasets for the performance analysis of these algorithms. There are two kinds of datasets that we worked with.

Experiment 1 - Normal case data

Normal dataset with random string and a randomly sliced substring (pattern) out of the main string.

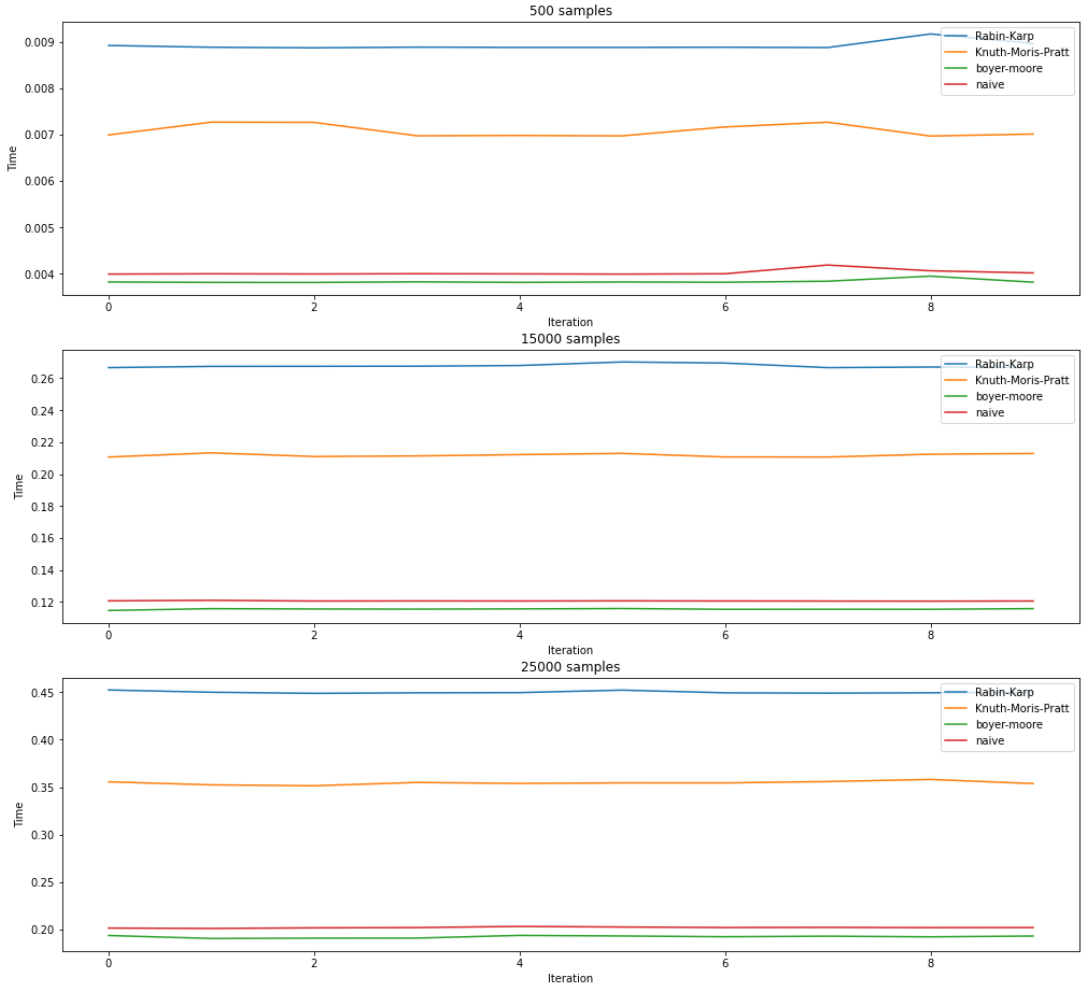
for example -

Main string - ABCDEFGHIJKLMNOP

Pattern string - JKL

We generated 500, 15000, and 25000 samples as datasets and ran all the algorithms 10 times on each dataset to get a good idea of average running time.

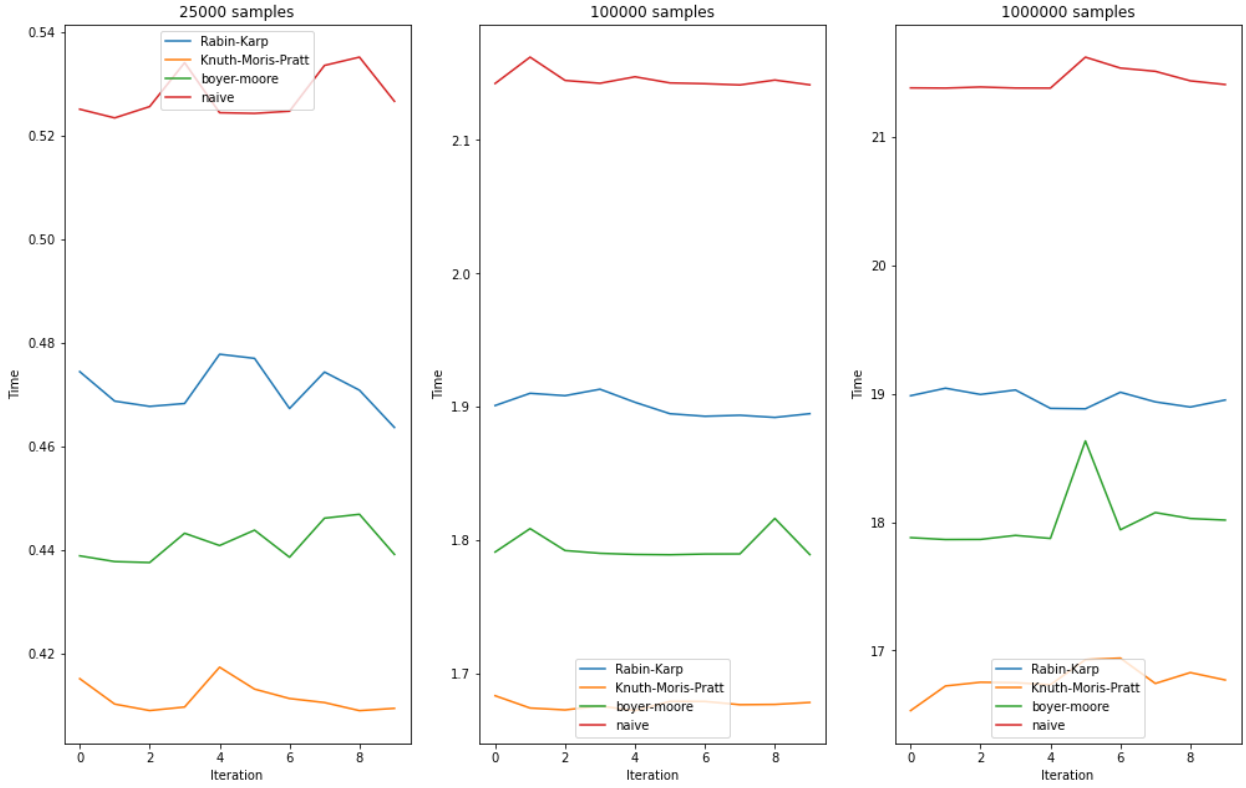
This experiment showed that the naive algorithm is really good (same as Boyer-Moore). It turned out to be better than the theoretically best Knuth-Morris-Pratt algorithm. The following image shows the same information in the graphic.



Experiment 2 - Worst Case data

Upon analysis, we found out that sophisticated string matching algorithms like the KMP algorithm provides an advantage when there is repetition in data.
For example, when data is like -
Main string - AAAAAAAAAAAAAAAAB
Pattern string - AAB

So, we generated such random data of 25000, 100000, and 1000000 samples. We increased the number of samples to get a significant number while comparing the time of execution.
This experiment showed that the KMP algorithm is the best (takes the least time) among others as the theory suggested.
The following graph reflects the same information.
(y-axis is time in seconds, the x-axis is the number of iterations on the same dataset)



Conclusion

By the experiments that we ran and the theory we understood with the progress of this project, we concluded

- A naive algorithm is a good approach if the main string contains randomly arranged characters
- The Boyer-Moore algorithm performs well in both experiments
- The Rabin-Karp method is only better than the naive approach in experiment 2, when the main string contains a repetitive arrangement of characters
- The Knuth-Morris-Pratt algorithm is the best algorithm considering experiment 2. It provides the advantage when the main string has a repetitive arrangement of characters.

Individual Contribution

- Ravi Rajpurohit contributed to coding the KMP algorithm, the naive algorithm, and defining the dataset (worst-case dataset) for which the advantage of the KMP algorithm would be reflected
- Vedanti Ambulkar contributed to coding the Rabin-Karp algorithm, and Boyer-Moore algorithm and generating the datasets
- Both of us contributed to the performance comparison of these algorithms and wrote the final report