Marmara University - Faulty of Engineering

Department of Computer Engineering

CSE2246 Analysis of Algorithms (Spring 2023)

Submit Date: 17/05/2023.

String Matching Problem

Student Number (ID)	Name	Surname	Job Distribution
150120076	Furkan	Gökgöz	Input/Output – coding/debugging / Report
150120998	Abdelrahman	Zahran	Horspool's Algorithm / Report
150121043	Bora	Duman	Boyer – Moore Algorithm / Report
150121051	Arda	Öztürk	Brute Force Algorithm / Report

Sections Of the Report: -

- Section (1): Problem Definition.
- Section (2): Explanation.
- Section (3): Brute – force Algorithm.
- Section (4): Horspool's Algorithm.
- Section (5): Boyer – Moore Algorithm.
- Section (6): Implementation (Code).
- Section (7): Performance Analysis (Brute force vs. Horspool's vs. Boyer – Moore).
- Section (8): Test Cases.

Section (1): Problem Definition: -

The string-matching problem is a fundamental task in computer science that involves finding occurrences of one string, known as the pattern, within another string, known as the text. The goal is to determine whether the pattern exists in the text and, if so, locate all the positions or indices where the pattern occurs.

In more detail, the problem can be stated as follows: given a text string T of length n and a pattern string P of length m, we want to determine all the occurrences of P within T. An occurrence refers to a substring of T that is identical to P. The position or index of an occurrence is the starting position of the substring within T.

String Matching Problem Types: -

1. Exact matching vs. approximate matching: The string-matching problem can be further divided into exact matching and approximate matching. In exact matching, the goal is to find occurrences of the pattern exactly as it appears in the text. In approximate matching, the goal is to find occurrences that are similar to the pattern but may have some differences, such as substitutions, insertions, or deletions. Approximate matching is often used in applications such as spell checking or DNA sequence alignment.

2. Multiple patterns: In some cases, we may need to find occurrences of multiple patterns within the text simultaneously. This scenario is known as multiple pattern matching. It can be addressed using algorithms like Aho-Corasick, which efficiently searches for multiple patterns in linear time relative to the text length.

3. Overlapping occurrences: The string-matching problem can be further categorized based on whether overlapping occurrences are allowed or not. In non-overlapping matching, each occurrence of the pattern in the text is disjoint from the others. In overlapping matching, occurrences can overlap with each other. The choice depends on the specific requirements of the problem.

4. Online vs. offline matching: String matching can be performed in an online or offline manner. In online matching, the text is processed incrementally, and pattern occurrences are detected as the text is being read. In offline matching, the entire text is available upfront, and the pattern is matched against it in a batch processing manner. Online matching is useful in scenarios where the text is continuously streaming or arriving in chunks.

5. Applications: The string-matching problem has numerous applications across various domains. Some examples include:

   - Text searching and indexing: Search engines use string matching algorithms to efficiently retrieve relevant documents based on user queries.
   - DNA sequence analysis: String matching is used to find similarities or patterns in DNA or protein sequences, aiding in genetic research and bioinformatics.
   - Data mining and information extraction: String matching is employed to extract structured information from unstructured data, such as extracting names, dates, or email addresses from text documents.
   - Plagiarism detection: String matching algorithms can be used to identify instances of plagiarism by comparing a given document with a large database of existing documents.
   - Computer security: String matching is used in intrusion detection systems to identify known patterns or signatures of malicious code or network attacks.

These are just a few examples of the wide range of applications and variations of the string-matching problem. The specific techniques and algorithms used may vary depending on the application and its requirements.

The string-matching problem can be approached using various algorithms, each with its own advantages and efficiency characteristics. Some commonly used algorithms for string matching include:

1. Naive algorithm: This simple algorithm checks for the pattern's occurrence at every possible position in the text. It has a time complexity of O((n-m+1) * m) in the worst case.

2. Knuth-Morris-Pratt (KMP) algorithm: This algorithm utilizes the concept of a failure function to avoid unnecessary comparisons. It preprocesses the pattern to determine the maximum suffix that is also a prefix for each position. The KMP algorithm achieves a time complexity of O(n+m) in the worst case.

3. Boyer-Moore algorithm: This algorithm uses a heuristic approach to skip unnecessary comparisons by examining the characters from right to left. It preprocesses the pattern and utilizes two rules, the bad character rule and the good suffix rule, to efficiently skip irrelevant portions of the text. The Boyer-Moore algorithm typically has a time complexity of O(n/m) in the best case and O(nm) in the worst case.

4. Rabin-Karp algorithm: This algorithm employs hashing to efficiently compare the pattern with the text. It computes the hash values of the pattern and a sliding window of the text and compares them. The Rabin-

Karp algorithm achieves an average-case time complexity of O(n+m) but can have a worst-case time complexity of O((n-m+1) * m).

These are just a few examples of algorithms used to solve the string-matching problem. The choice of algorithm depends on factors such as the expected size of the text and pattern, the desired time complexity, and any specific requirements or constraints of the problem at hand.

Section (2): Explanation: -

Main Overview:

The experiment aims to compare the performance of three string matching algorithms, namely Brute-force, Horspool's, and Boyer-Moore algorithms. The algorithms will be implemented in C programming language to search for a pattern in a long HTML file (longer than 1MB). The output of the algorithms will be a highlighted HTML file with the number of occurrences of the pattern.

To compare the three string matching algorithms (brute-force, Horspool's algorithm, and Boyer-Moore algorithm), we need to design an experiment that involves searching for occurrences of a given pattern in a long HTML file and measuring the performance of each algorithm in terms of character comparisons and running time.

- Step 1: Designing the Experiment
  - Long HTML files: We will need to decide on or generate long HTML files. These files can be obtained from the internet, ensuring that they are larger than 1MB. The URLs or sources of the HTML files should be documented in the detailed report.
  - Patterns: We need to choose patterns for searching within the HTML files. For Type 1 (HTML files with English text), meaningful English words can be used as patterns. For Type 2 (HTML files with random bit-strings), randomly generated bitstrings of varying lengths can be used as patterns. At least three samples of each type should be selected.

  Describe the chosen patterns and their properties (length, repetitions, etc.). It's important to consider patterns with varying lengths, as well as patterns with different first characters and repetition patterns, to observe the effects on algorithm performance.
- Step 2: Coding and Running

  Implement all three string matching algorithms (brute-force, Horspool's algorithm, and Boyer-Moore algorithm) in C programming language. The implemented code can perform the following tasks:
  - Outputting bad symbol tables and good suffix tables: Each algorithm requires precomputing tables for efficient searching. These tables will be printed for each pattern in the detailed report.
  - Generating an output HTML file with highlighted pattern occurrences: The algorithms should search for all occurrences of the pattern in the HTML file and highlight them using the **<mark></mark>** tag. The updated HTML file should be saved. The method for generating the highlighted file will be documented in the detailed report.
  - Recording the number of character comparisons and running time: Measure the number of character comparisons and the running time (in milliseconds) for each algorithm. These metrics will be printed for each pattern in the detailed report. To obtain accurate results, it's recommended to run each algorithm multiple times and take the average.

  Additionally, running tests of the code for the provided text and pattern have been conducted to ensure correctness. The output will be provided in the detailed report.
- Step 3: Illustrating and Analyzing Results
  - Timing/complexity results: Present the timing/complexity results in plots and/or tables. Show the running time of each algorithm for different HTML files and patterns.
  - Performance comparison: Compare the performance of all three algorithms for various HTML files and patterns. Analyze the results in terms of character comparisons and running time. Comment on the findings in the detailed report.

To compare the three string matching algorithms (brute-force, Horspool's algorithm, and Boyer-Moore algorithm) for the string-matching problem, we can follow these steps:

1. Implement the algorithms: Write code to implement all three algorithms in C programming language. Each algorithm should take a text string and a pattern string as input and return the number of occurrences of the pattern in the text. The implementations are correct and adhere to the descriptions of the algorithms.

2. Generate test cases: Create a set of test cases to evaluate the performance of the algorithms. The test cases should include different text and pattern combinations, covering various scenarios such as short and long texts, short and long patterns, repetitive patterns, patterns with different starting characters, etc.

3. Measure runtime: Run each algorithm on the test cases and measure their running time. Start a timer before calling the algorithm and stop it after the algorithm completes its execution. Record the running time for each test case. Repeat the measurement multiple times to obtain more accurate results.

4. Count character comparisons: Modify the implementations of the algorithms to keep track of the number of character comparisons performed during the matching process. Update the count every time a character in the text is compared with a character in the pattern. Record the number of character comparisons for each test case.

5. Analyze the results: Compare the performance of the algorithms based on their running time and the number of character comparisons. Identify patterns and trends in the results. Consider factors such as the length of the text and pattern, the starting character of the pattern, and the presence of repetitive patterns. Look for cases where one algorithm outperforms the others and try to understand the reasons behind it.

6. Present the findings: Present the results and analysis in a clear and organized manner. Use tables or charts to display the running time and character comparison metrics for each algorithm and test case. Provide explanations and insights into the observed performance differences. Discuss any limitations or assumptions made during the experiment.

By following these steps, we have managed effectively to compare the three string matching algorithms and gain insights into their performance characteristics for different scenarios.

Section (3): Brute – force Algorithm: -

The brute force algorithm, also known as the naive algorithm, is a simple and straightforward approach for finding occurrences of a pattern string within a larger text string. It involves systematically checking for the pattern's occurrence at every possible position in the text.

The following is a step-by-step explanation of how the brute force algorithm works:

1. Let T be the text string of length n, and P be the pattern string of length m.

2. Iterate through each position i in the text string T from 0 to n - m. This represents the starting position in the text where the pattern is being compared.

3. For each position i, compare the characters of the text starting from position i with the characters of the pattern.

4. Compare the characters of the pattern with the corresponding characters in the text at each position j ($0 \leq j < m$). If at any position j, the characters do not match, break the comparison loop, and move to the next position i+1 in the text.

5. If the loop completes without any mismatches, it means the pattern has been found at position i. Store or output the index i as an occurrence of the pattern.
6. Repeat steps 2 to 5 until all positions in the text have been checked.

The brute force algorithm compares each character of the pattern with the corresponding character in the text, starting from every position in the text. If the pattern is found at a particular position, it is considered an occurrence.

The brute force algorithm has a time complexity of O((n-m+1) * m) in the worst case, where n is the length of the text and m is the length of the pattern. This worst-case scenario occurs when the pattern has to be compared with every possible substring of the text.

While the brute force algorithm is simple to implement, it may not be the most efficient solution for large texts or patterns. It becomes impractical when the text or pattern sizes are significant, as the number of comparisons increases exponentially. In such cases, more efficient algorithms like the Knuth-Morris-Pratt (KMP) algorithm, Boyer-Moore algorithm, or Rabin-Karp algorithm are preferred.

However, the brute force algorithm serves as a baseline algorithm and can be useful for small-scale problems or as a starting point for understanding more advanced string-matching algorithms.

Best Applications: -

The brute force string matching algorithm is a simple and straightforward method for finding occurrences of a pattern within a larger text. While it may not be the most efficient algorithm in terms of time complexity, there are still several situations where it can be useful:

- Text Editors: Brute force string matching can be applied in text editors to implement features like "Find" and "Find and Replace." It allows users to search for specific strings within a document or replace them with different ones.
- File Search: When searching for files on a computer or a server, the brute force string matching algorithm can be used to locate files that contain a particular keyword or phrase. This is especially handy when the file system doesn't provide advanced search capabilities.
- Data Extraction: In data extraction tasks, where specific information needs to be extracted from a large text corpus, brute force string matching can be employed to identify and extract relevant data points based on predefined patterns or keywords.
- Password Cracking: Brute force string matching algorithms can be used for password cracking, where an attacker systematically tries different combinations of characters to guess a password. This application, however, is typically used for malicious purposes and is highly discouraged.
- Text Processing: Brute force string matching can be utilized for tasks like text cleaning and normalization. It allows for the identification and removal of unwanted patterns, such as special characters or formatting, from text data.
- DNA Sequencing: In bioinformatics, brute force string matching algorithms can be employed to compare DNA sequences and identify regions of similarity or specific patterns within genetic data.
- Network Security: Brute force string matching can be used in network security applications, such as intrusion detection systems (IDS), to search for specific patterns or signatures in network traffic. This can help identify and block malicious activities or detect potential security breaches.

It's worth noting that while the brute force string matching algorithm serves as a baseline solution in many of these scenarios, there are more efficient algorithms available for more substantial text processing tasks or scenarios that require real-time processing.

The best-case, worst-case, and average-case scenarios for the brute force string matching algorithm:

1. Best-case scenario (Best case time complexity):

   - The best-case scenario for the brute force algorithm occurs when the pattern is not present in the text.
   - In this case, the algorithm will perform a single pass through the text, comparing each character of the pattern with the corresponding character in the text.
   - The algorithm will quickly determine that the pattern is absent and can terminate.
   - The best-case time complexity for the brute force algorithm is O(n), where n is the length of the text.
   - This is because the algorithm needs to examine each character in the text exactly once to conclude that the pattern is not present.

2. Worst-case scenario (Worst case time complexity):

   - The worst-case scenario for the brute force algorithm occurs when the pattern occurs at every possible position in the text or when there are multiple occurrences of the pattern overlapping with each other.
   - In this case, the algorithm needs to compare the pattern with every possible substring of the text, resulting in a large number of comparisons.
   - The worst-case time complexity for the brute force algorithm is O((n-m+1) * m), where n is the length of the text and m is the length of the pattern.
   - This worst-case time complexity arises when the pattern needs to be compared with every substring of the text, resulting in (n-m+1) substrings to check and each comparison requiring m operations.

3. Average-case scenario (Average case time complexity):

   - Determining the average-case time complexity for the brute force algorithm can be challenging, as it depends on the characteristics of the input strings and their distribution.
   - In general, the average-case time complexity is influenced by the probability of finding the pattern at different positions in the text.
   - On average, the brute force algorithm needs to check about (n-m+1)/2 positions in the text to find a single occurrence of the pattern, assuming a random distribution of pattern occurrences.
   - Therefore, the average-case time complexity for the brute force algorithm is typically considered to be O(nm/2) or simply O(nm), where n is the length of the text and m is the length of the pattern.
   - This average-case time complexity arises from the need to compare the pattern with a subset of the text's substrings, with an expected number of comparisons around (n-m+1)/2.

It's important to note that the best-case, worst-case, and average-case scenarios provide different insights into the performance of the brute force algorithm. The worst-case scenario represents the upper bound of the algorithm's performance, while the average-case scenario gives a more realistic expectation based on typical input distributions. The best-case scenario illustrates the optimal situation where the algorithm can quickly determine the absence of the pattern.

Section (4): Horspool's Algorithm: -

The Horspool's algorithm is a string-matching algorithm that efficiently searches for occurrences of a pattern within a text. It was developed by R. Nigel Horspool's in 1980 as an improvement over the brute force algorithm. The algorithm is particularly useful when the pattern is relatively long compared to the text.

Here's a step-by-step explanation of how the Horspool's algorithm works:

1. Preprocessing Step:

   - The algorithm begins with a preprocessing step where it constructs a shift table, also known as the "bad character shift" table.
   - The shift table is an array that stores the maximum shift distance for each character in the pattern based on its rightmost occurrence.
   - To construct the shift table, the algorithm iterates through each character in the pattern from left to right.
   - For each character, it determines its rightmost occurrence in the pattern and calculates the shift value as the difference between the pattern's length and the index of the rightmost occurrence.
   - If a character does not occur to the right of the current position, a default shift value is set to the length of the pattern.

2. Matching Step:

   - Once the shift table is constructed, the algorithm proceeds to the matching step where it searches for occurrences of the pattern within the text.
   - The matching process starts by aligning the right end of the pattern with the corresponding position in the text.
   - It then slides the pattern over the text from right to left, comparing characters one by one.
   - If a mismatch occurs at any position, the algorithm uses the shift table to determine the shift value based on the character that mismatched.
   - The pattern is then shifted to the right by the calculated shift value, and the process continues with the next comparison.
   - This sliding and shifting process continues until either a match is found, or the end of the text is reached.

The Horspool's algorithm achieves efficiency by utilizing the shift table. By precomputing the maximum shift distances for each character, it can skip unnecessary comparisons when a mismatch occurs. This allows the algorithm to shift the pattern farther ahead, reducing the overall number of character comparisons required.

Shift Table: -

In Horspool's algorithm, the shift table is a lookup table that determines how much the pattern should be shifted after a mismatch occurs between the pattern and a portion of the text being searched. The table is based on the last occurrence of each character in the pattern.

The shift table is created during the preprocessing stage of the algorithm and is used during the search stage. The shift table is an array with one entry for each possible character in the alphabet. The entry for each character specifies the amount by which the pattern should be shifted if that character is the first character in the pattern that does not match the corresponding character in the text being searched.

If a character does not appear in the pattern, the default shift value should be used. The default shift value is equal to the length of the pattern, which means that the pattern will be shifted by its full length if there is a mismatch with a character not present in the pattern.

To create the shift table, the algorithm scans the pattern from right to left and stores the distance between the rightmost occurrence of each character in the pattern and the end of the pattern. If a character appears multiple times in the pattern, the rightmost occurrence should be used for the shift table.

The shift table allows the algorithm to skip unnecessary comparisons and quickly determine the next possible location for a match. By using the shift table to shift the pattern after a mismatch, the algorithm avoids examining the same characters repeatedly and achieves faster string matching.

To construct the shift table in Horspool's algorithm, you need to follow these steps:

1. Initialize the shift table: Create an array with entries corresponding to each character in the alphabet. Initialize each entry with the default shift value, which is the length of the pattern.
2. Scan the pattern from left to right, except for the last character.
3. Calculate the shift distance for each character: For each character in the pattern, except the last one, calculate the shift distance by subtracting 1 from the position of that character in the pattern.
4. Update the shift table: Update the corresponding entry in the shift table for each character in the pattern with the calculated shift distance.

Here is a step-by-step example to illustrate the construction of the shift table:

Pattern: "EXAMPLEPATTERN" Alphabet: {'A', 'E', 'L', 'M', 'N', 'P', 'R', 'T', 'X'}

1. Initialize the shift table:
   - Table: [14, 14, 14, 14, 14, 14, 14, 14, 14]
2. Scan the pattern from left to right, except for the last character:
   - Character "E" is at position 1.
   - Character "X" is at position 10.
   - Character "A" is at position 0 (not considering the last character)
3. Calculate the shift distance for each character:
   - Character "E" has shift distance 0.
   - Character "X" has shift distance 9.
   - Character "A" has shift distance -1 (default shift value)
4. Update the shift table:
   - Table: [0, 14, 14, 14, 14, 14, 14, 14, 9]

The resulting shift table provides the shift distances for each character in the alphabet. During the search stage of Horspool's algorithm, this table is used to efficiently determine the amount by which the pattern should be shifted when a mismatch occurs with the text being searched.

The time complexity of the Horspool's algorithm is typically O(n + m), where n is the length of the text and m is the length of the pattern. This complexity arises from the preprocessing step of constructing the shift table (O(m)) and the matching step of sliding and shifting the pattern (O(n)). In practice, Horspool's algorithm performs well for large texts or patterns with a relatively small alphabet.

Overall, the Horspool's algorithm is a simple yet efficient approach for string matching. While it may not be as advanced as some other algorithms, it provides a significant improvement over the brute force algorithm and is widely used in various applications that require pattern searching and matching.

Best Applications: -

Horspool's algorithm, also known as the Boyer-Moore-Horspool's algorithm, is a string-matching algorithm that offers improvements over the brute force approach. It can be particularly useful in the following applications:

1. Text Search: Horspool's algorithm is commonly employed in text search applications to find occurrences of a pattern within a larger text. It is especially beneficial when searching for patterns in large text corpora or files, as it can significantly reduce the number of comparisons required compared to brute force algorithms.

2. Data Compression: Horspool's algorithm can be utilized in data compression algorithms, such as the Lempel-Ziv-Welch (LZW) algorithm. By efficiently identifying repeating patterns in the input data, Horspool's algorithm helps in achieving better compression ratios and reducing the overall size of the compressed data.

3. DNA Sequencing: In bioinformatics, Horspool's algorithm can be applied to DNA sequencing tasks. It allows for the efficient identification of genetic patterns or motifs within DNA sequences, aiding in tasks such as gene discovery, sequence alignment, and sequence comparison.

4. File Carving: Horspool's algorithm can be useful in file carving, which involves recovering files or data from fragmented or corrupted storage media. By searching for specific file signatures or headers, Horspool's algorithm helps identify and extract files from the raw data.

5. Network Intrusion Detection: Horspool's algorithm can be employed in network intrusion detection systems (IDS) or packet sniffers to search for known malicious patterns or signatures in network traffic. It helps in detecting and alerting potential security threats or attacks.

6. Compiler Design: Horspool's algorithm can be utilized in compiler design for tasks such as lexical analysis and pattern matching. It assists in efficiently identifying keywords, symbols, or patterns in the source code, enabling the compiler to generate tokens and analyze the code structure.

It's important to note that while Horspool's algorithm provides improvements over brute force algorithms, there are more advanced string-matching algorithms available, such as the full Boyer-Moore algorithm or the Knuth-Morris-Pratt algorithm, which may offer even better performance in certain scenarios.

The best-case, worst-case, and average-case scenarios for the Horspool's string-matching algorithm:

1. Best-case scenario:

   - The best-case scenario for the Horspool's algorithm occurs when the pattern is not present in the text or is found at the very beginning of the text.

- In this case, the algorithm performs a single pass through the text, comparing each character of the pattern with the corresponding character in the text until a mismatch is found at the first position.
- The best-case time complexity for the Horspool's algorithm is O(n), where n is the length of the text.
- This is because the algorithm examines each character in the text exactly once before determining the absence of the pattern.

2. Worst-case scenario:

- The worst-case scenario for the Horspool's algorithm occurs when the pattern occurs at every possible position in the text or when there are multiple occurrences of the pattern overlapping with each other.
- In this case, the algorithm needs to compare the pattern with every possible substring of the text, resulting in a large number of comparisons.
- The worst-case time complexity for the Horspool's algorithm is O(nm), where n is the length of the text and m is the length of the pattern.
- This worst-case time complexity arises when the algorithm needs to check every substring of the text for a match with the pattern.

3. Average-case scenario:

- Determining the average-case time complexity for the Horspool's algorithm can be challenging and depends on various factors such as the characteristics of the input strings and their distribution.
- In practice, the average-case time complexity of the Horspool's algorithm is often considered to be O(n + m), where n is the length of the text and m is the length of the pattern.
- The average-case time complexity accounts for the preprocessing step of constructing the shift table (O(m)) and the matching step of sliding and shifting the pattern (O(n)).
- The Horspool's algorithm typically performs well on average for various text and pattern distributions, making it a practical choice for many string-matching tasks.

It's important to note that the best-case, worst-case, and average-case scenarios provide different perspectives on the performance of the Horspool's algorithm. The worst-case scenario represents the upper bound of the algorithm's performance, while the average-case scenario gives a more realistic expectation based on typical input distributions. The best-case scenario illustrates the optimal situation where the algorithm can quickly determine the absence or presence of the pattern.

Section (5): Boyer – Moore Algorithm: -

The Boyer-Moore algorithm is a string-matching algorithm that efficiently searches for occurrences of a pattern within a text. It was developed by Robert S. Boyer and J Strother Moore in 1977 and is widely regarded as one of the most efficient string searching algorithms, especially for cases where the pattern is relatively long compared to the text.

Here's a step-by-step explanation of how the Boyer-Moore algorithm works:

1. Preprocessing Step:

- The algorithm begins with a preprocessing step where it constructs two lookup tables: the "bad character" table and the "good suffix" table.

- The bad character table is used to determine the maximum shift distance based on the mismatched character encountered during the matching process. It records the rightmost occurrence of each character in the pattern.
- The good suffix table provides information on the shift amount when a mismatch occurs, but a portion of the pattern matches the text. It determines the maximum shift that can be applied based on the matching suffix of the pattern.

2. Matching Step:

- Once the lookup tables are constructed, the algorithm proceeds to the matching step where it searches for occurrences of the pattern within the text.
- The matching process begins by aligning the right end of the pattern with the corresponding position in the text.
- The algorithm slides the pattern over the text from right to left, comparing characters from right to left as well.
- If a mismatch occurs at any position, the algorithm consults the bad character table to determine the maximum shift distance based on the mismatched character. If the mismatched character is not present in the pattern, the entire pattern can be shifted by the length of the pattern.
- Additionally, the algorithm uses the good suffix table to determine the maximum shift amount based on the longest matching suffix of the pattern.
- The algorithm then shifts the pattern to the right by the larger of the two computed shift distances and continues comparing characters.
- The process repeats until a complete match is found or the end of the text is reached.

The Boyer-Moore algorithm achieves its efficiency by utilizing the bad character and good suffix tables. These lookup tables allow the algorithm to quickly skip portions of the text that are guaranteed not to contain the pattern, reducing the number of comparisons required.

Bad – Symbol Table: -

In the Boyer-Moore algorithm, the bad symbol table, also known as the "bad character rule" or "occurrence heuristic," is a lookup table that helps determine the maximum amount by which the pattern can be shifted when a mismatch occurs between the pattern and the text being searched. The table is based on the last occurrence of each character in the pattern.

The bad symbol table is created during the preprocessing stage of the Boyer-Moore algorithm and is used during the search stage. It is an array with one entry for each possible character in the alphabet. The entry for each character specifies the maximum number of positions that the pattern can be shifted when a mismatch occurs with that character in the text.

To create the bad symbol table, the algorithm scans the pattern from right to left and stores, for each character, the distance between the rightmost occurrence of that character in the pattern and the end of the pattern. If a character does not appear in the pattern, the default shift value is equal to the length of the pattern.

During the search stage, when a mismatch occurs between the pattern and the text, the bad symbol table is consulted to determine the maximum shift distance. The algorithm looks at the character in the text that caused the mismatch and checks the corresponding entry in the bad symbol table. If the character is present in the bad symbol table, the

pattern is shifted to align that character in the text with the rightmost occurrence of that character in the pattern. If the character is not present in the bad symbol table, the pattern is shifted by the full length of the pattern.

By utilizing the bad symbol table, the Boyer-Moore algorithm can skip comparisons and efficiently shift the pattern to potential match positions. This rule is particularly effective when there are many instances of the same character in the pattern, as it allows the algorithm to skip over portions of the text where the character is known to occur, reducing the overall number of comparisons needed.

To construct the bad character table in the Boyer-Moore algorithm, you need to follow these steps:

1. Initialize the bad character table: Create an array with entries corresponding to each character in the alphabet. Initialize each entry with the default shift value, which is the length of the pattern.
2. Scan the pattern from right to left: Starting from the second-to-last position in the pattern, for each character, determine the rightmost occurrence of that character in the pattern.
3. Calculate the shift distance for each character: For each character in the alphabet, calculate the shift distance by subtracting the position of the rightmost occurrence of that character in the pattern from the length of the pattern.
4. Update the bad character table: Update the corresponding entry in the bad character table for each character in the pattern with the calculated shift distance.

Here is a step-by-step example to illustrate the construction of the bad character table:

Pattern: "EXAMPLEPATTERN" Alphabet: {'A', 'E', 'L', 'M', 'N', 'P', 'R', 'T', 'X'}

1. Initialize the bad character table:
    - Table: [14, 14, 14, 14, 14, 14, 14, 14, 14]
2. Scan the pattern from right to left:
    - Character "N" occurs at position 12.
    - Character "R" occurs at position 11.
    - Character "T" occurs at position 8.
    - Character "P" occurs at position 7.
    - Character "M" occurs at position 5
    - Character "L" occurs at position 3.
    - Character "E" occurs at position 1.
    - Character "X" does not occur in the pattern.
    - Character "A" does not occur in the pattern.
3. Calculate the shift distance for each character:
    - Character "N" has shift distance 2.
    - Character "R" has shift distance 3.
    - Character "T" has shift distance 6.
    - Character "P" has shift distance 7.

- Character "M" has shift distance 9.
- Character "L" has shift distance 11.
- Character "E" has shift distance 13.
- Character "X" has default shift distance 14.
- Character "A" has default shift distance 14.

4. Update the bad character table:

- Table: [7, 13, 11, 9, 2, 7, 3, 6, 14]

The resulting bad character table provides the maximum shift distances for each character in the alphabet. During the search stage of the Boyer-Moore algorithm, this table is used to efficiently determine the amount by which the pattern should be shifted when a mismatch occurs with the text being searched.

Good – Suffix Table: -

In the Boyer-Moore algorithm, the good suffix table, also known as the "good suffix rule," is a lookup table that helps determine the maximum amount by which the pattern can be shifted when a mismatch occurs between the pattern and the text being searched. The table is based on the suffixes of the pattern.

The good suffix table is created during the preprocessing stage of the Boyer-Moore algorithm and is used during the search stage. It is an array with entries corresponding to each position in the pattern, except for the last position. Each entry in the table stores the maximum shift distance for a specific suffix of the pattern.

To create the good suffix table, the algorithm examines each suffix of the pattern. A suffix is a substring that starts at a certain position in the pattern and extends to the end of the pattern. The algorithm determines the longest proper suffix (a proper suffix is a suffix that is not equal to the entire pattern) that matches a suffix of the pattern from the right side.

For each suffix, the algorithm calculates the shift distance by finding the position of the leftmost occurrence of the suffix in the pattern and subtracting it from the length of the pattern. If the suffix does not have any matches in the pattern, the default shift value is equal to the length of the pattern.

The good suffix table is used during the search stage when a mismatch occurs between the pattern and the text. The algorithm examines the length of the suffix that matches the text starting from the mismatch position. If there is a match, the pattern is shifted to align the matched suffix in the text with the same suffix in the pattern. If there is no match, the algorithm consults the good suffix table to determine the maximum shift distance based on the length of the suffix.

By utilizing the good suffix table, the Boyer-Moore algorithm can efficiently shift the pattern to potential match positions, taking advantage of the information about matching suffixes. This rule is particularly effective when there are repeated patterns or substrings in the pattern, as it allows the algorithm to skip over portions of the text where the matched suffix occurs, reducing the overall number of comparisons needed.

To construct the good suffix table in the Boyer-Moore algorithm, follow these steps:

1. Initialize the good suffix table: Create an array with entries corresponding to each position in the pattern.
2. Calculate the length of the longest proper suffix: Starting from the last position in the pattern, compare each suffix of the pattern with the suffixes of the entire pattern from the right side. Keep track of the length of the longest proper suffix that matches a suffix of the pattern.

3. Update the good suffix table: For each position in the pattern, calculate the shift distance by subtracting the length of the longest proper suffix from the length of the pattern. If there is no occurrence of the suffix, set the default shift value as the length of the pattern.

4. Handle the special case: If the first character of the pattern does not match with any other occurrence in the pattern, set the default shift value for the first position as the length of the pattern.

Here is a step-by-step example to illustrate the construction of the good suffix table:

Pattern: "EXAMPLEPATTERN" Length of the pattern: 14

1. Initialize the good suffix table: Table: [_, _, _, _, _, _, _, _, _, _, _, _, _, _]

2. Calculate the length of the longest proper suffix:

    - Starting from the last position, compare suffixes:

        - "N" matches with the suffix "N" => longest proper suffix length = 1
        - "RN" matches with the suffix "RN" => longest proper suffix length = 2
        - "ERN" matches with the suffix "ERN" => longest proper suffix length = 3
        - "TERN" matches with the suffix "TERN" => longest proper suffix length = 4
        - "PTERN" does not match any suffix.
        - "APTERN" does not match any suffix.
        - "MAPTERN" does not match any suffix.
        - "EMAPTERN" does not match any suffix.
        - "LEMAPTERN" does not match any suffix.
        - "XLEMAPTERN" does not match any suffix.
        - "AXLEMAPTERN" does not match any suffix.
        - "PAXLEMAPTERN" does not match any suffix.
        - "MPAXLEMAPTERN" does not match any suffix.
        - "EMPAXLEMAPTERN" does not match any suffix.

---

3. Update the good suffix table: Table: [4, _, _, _, 4, _, _, _, _, _, _, _, _, _]

4. Handle the special case:

    - The first character "E" does not have any other occurrence in the pattern.
    - Set the default shift value for the first position as 14 (length of the pattern).
    - Table: [14, 4, _, _, 4, _, _, _, _, _, _, _, _, _]

The resulting good suffix table provides the maximum shift distances for each suffix of the pattern. During the search stage of the Boyer-Moore algorithm, this table is used to efficiently determine the amount by which the pattern should be shifted when a mismatch occurs with the text being searched.

The time complexity of the Boyer-Moore algorithm is typically considered to be O(n + m), where n is the length of the text and m is the length of the pattern. In practice, the Boyer-Moore algorithm often outperforms other string-matching algorithms, especially when the pattern is long, or the text is large.

The Boyer-Moore algorithm is widely used in various applications that require efficient string searching, such as text editors, compilers, and data processing systems. Its ability to achieve high performance by utilizing lookup tables makes it a valuable tool in situations where fast pattern matching is crucial.

Best Applications: -

The Boyer-Moore algorithm is a highly efficient string-matching algorithm that is widely used in various applications. Some of the best applications of the Boyer-Moore algorithm include:

1. Text Search: The Boyer-Moore algorithm is extensively utilized in text search applications to find occurrences of a pattern within a larger text. It is particularly beneficial when searching for patterns in large text corpora or files, as it significantly reduces the number of comparisons required, making it faster than brute force or other string-matching algorithms.

2. Data Compression: The Boyer-Moore algorithm can be employed in data compression algorithms, such as the Burrows-Wheeler transform and the DEFLATE algorithm used in formats like ZIP and gzip. By efficiently identifying repeating patterns in the input data, the algorithm helps achieve better compression ratios and reduce the overall size of compressed data.

3. Spell Checking: Boyer-Moore algorithm can be used in spell-checking systems to identify misspelled words. It allows for efficient searching and comparison of words against a dictionary or a list of valid words, enhancing the accuracy and speed of spell-checking operations.

4. Virus Scanning: In antivirus software, the Boyer-Moore algorithm can be employed to search for known virus signatures within files or network traffic. By efficiently scanning for specific patterns or sequences associated with known malware, the algorithm helps in detecting and preventing virus infections.

5. DNA Sequencing: The Boyer-Moore algorithm can be utilized in DNA sequencing tasks to search for specific genetic patterns or motifs within DNA sequences. It aids in tasks such as gene discovery, sequence alignment, and comparison, providing valuable insights into genetic data analysis.

6. Code Search: The Boyer-Moore algorithm can be applied in code search tools or integrated development environments (IDEs) to quickly locate code snippets or functions within large codebases. It enables developers to efficiently search for specific patterns or identifiers, enhancing code navigation and productivity.

7. File Carving: The Boyer-Moore algorithm can be used in file carving techniques to recover fragmented or partially corrupted files from storage media. By searching for specific file signatures or headers, the algorithm helps in identifying and extracting files from raw data.

Overall, the Boyer-Moore algorithm's ability to leverage pre-processing and advanced pattern matching techniques makes it well-suited for applications that involve searching, matching, or identifying patterns within large datasets.

The best-case, worst-case, and average-case scenarios for the Boyer-Moore string matching algorithm:

1. Best-case scenario:

    - The best-case scenario for the Boyer-Moore algorithm occurs when the pattern is not present in the text or is found at the very beginning of the text.

- In this case, the algorithm performs a single pass through the text, comparing each character of the pattern with the corresponding character in the text until a mismatch is found at the first position.
- The best-case time complexity for the Boyer-Moore algorithm is O(n/m), where n is the length of the text and m is the length of the pattern.
- This is because the algorithm can quickly determine the absence or presence of the pattern by examining only a fraction of the characters in the text.

2. Worst-case scenario:

- The worst-case scenario for the Boyer-Moore algorithm occurs when the pattern occurs at every possible position in the text or when there are multiple occurrences of the pattern overlapping with each other.
- In this case, the algorithm needs to compare the pattern with every possible substring of the text, resulting in a large number of comparisons.
- The worst-case time complexity for the Boyer-Moore algorithm is O(nm), where n is the length of the text and m is the length of the pattern.
- This worst-case time complexity arises when the algorithm needs to check every substring of the text for a match with the pattern.

3. Average-case scenario:

- Determining the average-case time complexity for the Boyer-Moore algorithm can be challenging and depends on various factors such as the characteristics of the input strings and their distribution.
- In practice, the average-case time complexity of the Boyer-Moore algorithm is often considered to be O(n/m), where n is the length of the text and m is the length of the pattern.
- The average-case time complexity takes into account the efficient skipping and shifting mechanisms of the algorithm based on the bad character and good suffix tables.
- The Boyer-Moore algorithm typically performs well on average, especially when the pattern is longer compared to the text.

It's important to note that the best-case, worst-case, and average-case scenarios provide different perspectives on the performance of the Boyer-Moore algorithm. The worst-case scenario represents the upper bound of the algorithm's performance, while the average-case scenario gives a more realistic expectation based on typical input distributions. The best-case scenario illustrates the optimal situation where the algorithm can quickly determine the absence or presence of the pattern. Overall, the Boyer-Moore algorithm is known for its efficient performance in practice, particularly for longer patterns or texts.

Section (6): Implementation: -

In the Implementation of this project, we used C programming language and its imported libraries provided and supported by oracle.

- C File(code): -

Project.c

- Imported Libraries: -
  - C Libraries
- Input Files: -

bitInput1.html bitInput2.html bitInput3.html input1.html input2.html input3.html

- Output File: -
  - Depends on the input file!

- Functions: -

```
//-----------------------------------------------------------------------------//
// Function to find the maximum of two integers
int max(int a, int b) {
    return a > b ? a : b;  // Returns 'a' if 'a' is greater than 'b', otherwise returns 'b'
}
//-----------------------------------------------------------------------------//
```

This max() Function: -

The function is declared with the return type **int**, indicating that it will return an integer value.

- The function is named **max**, which suggests its purpose of finding the maximum value.
- The function takes two parameters, **a** and **b**, both of type **int**. These parameters represent the two integers between which we want to find the maximum value.
- The body of the function consists of a single line of code:

  return a > b ? a : b;

  This is a conditional expression, often referred to as the ternary operator. It compares **a** and **b** using the greater-than operator (**>**). If **a** is greater than **b**, the expression evaluates to **a**, indicating that **a** is the maximum value. Otherwise, if **b** is greater than or equal to **a**, the expression evaluates to **b**, indicating that **b** is the maximum value.
- The result of the conditional expression is then returned as the output of the function.

To summarize, the **max** function takes two integers as input and returns the larger of the two.

```
//-----------------------------------------------------------------------------//
// Utility function to read the contents of a file into a string
char *read_file(const char *filename) {
    FILE *file = fopen(filename, "rb");  // Opens the file in binary mode for reading
    if (file == NULL) {
        printf("Error opening file: %s\n", filename);  // Prints an error message if the file cannot be
opened
```

```
        exit(EXIT_FAILURE);  // Exits the program with a failure status
    }

    fseek(file, 0, SEEK_END);  // Moves the file pointer to the end of the file
    long length = ftell(file);  // Retrieves the current position of the file pointer, which represents
the file size
    if (length < 26) {
        printf("Error: Invalid pattern");  // Prints an error message if the file size is less than 26
        exit(EXIT_FAILURE);  // Exits the program with a failure status
    }
    fseek(file, 0, SEEK_SET);  // Moves the file pointer back to the beginning of the file

    char *content = (char *)malloc(length - 25);  // Allocates memory to store the file content, excluding
the last 25 characters
    if (content == NULL) {
        printf("Error allocating memory for file content.\n");  // Prints an error message if memory
allocation fails
        exit(EXIT_FAILURE);  // Exits the program with a failure status
    }
    int i = 0;
    int j = 0;
    length -= 14;  // Decreases the file size by 14
    for (; i < 12; i++) {
        fgetc(file);  // Reads and discards the next character from the file, 12 times
    }
    for (; i < length; i++, j++) {
        content[j] = fgetc(file);  // Reads the next character from the file and stores it in the
'content' array
    }
    fclose(file);  // Closes the file
    content[j] = '\0';  // Appends a null terminator at the end of the 'content' array to signify the end
of the string

    return content;  // Returns the pointer to the dynamically allocated 'content' array
}
//-----------------------------------------------------------------------------//
```

This readFile() Function: -

- The code defines a function named **read_file** that takes a parameter **filename**, which is a C-style string representing the name of the file to be read. The function returns a pointer to a character array (**char***), which will hold the contents of the file.
- The function begins by attempting to open the specified file using the **fopen** function with the mode **"rb"**, which stands for "read binary." If the file cannot be opened (i.e., the **file** pointer is **NULL**), an error message is printed, indicating the failure to open the file. The **exit(EXIT_FAILURE)** statement is used to terminate the program with a failure status.

- The **fseek** function is used twice. First, it seeks to the end of the file (**SEEK_END**) to determine the file size. The current position of the file pointer is obtained using **ftell**, which returns the size as a long integer. If the file size is less than 26, an error message is printed, indicating an invalid pattern. Again, the program is terminated with a failure status.
- After validating the file size, the file pointer is moved back to the beginning of the file using **fseek(file, 0, SEEK_SET)**.
- The code allocates memory to store the file content (excluding the last 25 characters). It uses the **malloc** function, casting the return value to a **char***. If the memory allocation fails (i.e., **content** is **NULL**), an error message is printed, and the program exits with a failure status.
- Two variables, **i** and **j**, are initialized to 0. The variable **length** is reduced by 14, indicating that the last 14 characters of the file will be excluded from the content.
- The code enters a loop that reads and discards the next character from the file 12 times using the **fgetc** function. This is done to skip the first 12 characters.
- Another loop is executed to read the remaining characters from the file and store them in the **content** array. The loop runs until **i** reaches the value of **length**, and the characters are stored in the **content** array at the index **j**. Both **i** and **j** are incremented after each iteration.
- Once the file reading is complete, the file is closed using the **fclose** function to free system resources.
- Finally, a null terminator (**'\0'**) is appended at the end of the **content** array to indicate the end of the string.
- The function returns the pointer to the dynamically allocated **content** array, which now holds the contents of the file.

In summary, the **read_file** function reads the contents of a file into a dynamically allocated character array and returns a pointer to that array. It performs file handling operations, memory allocation, and character-by-character reading from the file.

```c
//-------------------------------------------------------------------------//
// Utility function to save the highlighted HTML file
void save_file(const char *filename, const char *content) {
    FILE *file = fopen(filename, "a");  // Opens the file in append mode
    if (file == NULL) {
        printf("Error opening file: %s\n", filename);  // Prints an error message if the file cannot be
opened
        exit(EXIT_FAILURE);  // Exits the program with a failure status
    }

    fwrite(content, 1, strlen(content), file);  // Writes the content to the file
    fclose(file);  // Closes the file
}
//-------------------------------------------------------------------------//
```

This is save_file() Function: -

- The code defines a function named **save_file** that takes two parameters: **filename** and **content**. The **filename** parameter is a C-style string that represents the name of the file to which the **content** will be saved. The **content** parameter is also a C-style string, representing the data to be written to the file.

- The function begins by attempting to open the specified file using the **fopen** function with the mode **"a"**, which stands for "append". If the file cannot be opened (i.e., the **file** pointer is **NULL**), an error message is printed, indicating the failure to open the file. The **exit(EXIT_FAILURE)** statement is used to terminate the program with a failure status.
- Once the file is successfully opened, the **fwrite** function is used to write the content to the file. The **fwrite** function takes four arguments: the source (**content**), the size of each element (1 byte in this case), the number of elements to write (determined by **strlen(content)**), and the destination file (**file**). This line of code writes the entire **content** string to the file.
- After writing the content, the file is closed using the **fclose** function to free system resources.

In summary, the **save_file** function saves the provided **content** string to a file specified by the **filename**. It performs file handling operations, opens the file in append mode, writes the content to the file using **fwrite**, and closes the file when finished.

```
//-------------------------------------------------------------------------------------//
// addList -> generates an HTML file containing tables for a "Bad Symbol Table" and a "Good Symbol Table," along with some additional HTML code
void addList(const char *filename, const char *pattern, const int bad_char[MAX_CHAR], int good[], int patternLen) {
    int i, j;
    FILE *output = fopen(filename, "w");  // Opens the file in write mode
    fprintf(output, "<html>\n<head>\n<style>\ntable, th, td {\nborder: 1px solid black;\ntext-align: center;\n}\nbody {\n\tword-wrap: break-word;\n}\n</style>\n</head>\n<body>\n<h2>Bad Symbol Table</h2>\n<table style>\n\t<tr>\n\t\t<th>*</th>\n");

    // Printing the Bad Symbol Table
    for (i = 1; i < MAX_CHAR; i++) {
        if (bad_char[i] != -1)
            fprintf(output, "\t\t<th>%c</th>\n", i);  // Prints a table header for each character that has a valid value (-1 represents an invalid value)
    }
    fprintf(output, "\t</tr>\n\t<tr>\n\t\t<td>%d</td>\n", patternLen);  // Prints the number of characters in the pattern in a table cell
    for (i = 1; i < MAX_CHAR; i++) {
        if (bad_char[i] != -1)
            fprintf(output, "\t\t<td>%d</td>\n", bad_char[i]);  // Prints the corresponding values from the 'bad_char' array in table cells
    }
    fprintf(output, "\t</tr>\n</table>\n<h2>Good Symbol Table</h2>\n<table style>\n\t<tr>\n\t\t<th>k</th>\n\t\t<th>Pattern</th>\n\t\t<th>Shift</th>\n</tr>\n");

    // Printing the Good Symbol Table
    for (i = 0; i < patternLen; i++) {
        fprintf(output, "<tr>\n\t\t<td>%d</td>\n\t\t<td>", i + 1);  // Prints the row number in a table cell
```

```
        for (j = 0; j < patternLen; j++) {
            if (patternLen - i - 1 == j) {
                fprintf(output, "<u>");  // Adds an underline tag for the current character in the pattern
            }
            fprintf(output, "%c", pattern[j]);  // Prints the characters of the pattern
        }
        fprintf(output, "</u>");

        fprintf(output, "</td>\n\t\t<td>%d</td>\n</tr>\n", good[i]);  // Prints the corresponding shift
value in the Good Symbol Table
    }
    fprintf(output, "\t</tr>\n");
    fprintf(output, "\t</tr>\n</table>\n\n</body>\n</html>\n\n<h2>Output</h2>\n");  // Prints HTML code
for other sections of the file
    fclose(output);  // Closes the file
}
//-----------------------------------------------------------------------------------------//
```

This addList() Function: -

- The code defines a function named **addList** that takes five parameters: **filename**, **pattern**, **bad_char**, **good**, and **patternLen**. These parameters represent the file name, the pattern string, an array **bad_char** with maximum character values, an array **good** containing shift values, and the length of the pattern string.
- The function starts by opening the specified file in write mode using **fopen** and assigns the file pointer to the **output** variable.
- The function then writes HTML code to the file using **fprintf** to generate an HTML file structure and styling. It includes a CSS style for tables, table headers, and table cells.
- The function writes an HTML heading and starts a table for the "Bad Symbol Table" section. It writes a table row header with an asterisk (*****) as the first cell.
- Next, it iterates over the **bad_char** array to print table headers for each character that has a valid value (not equal to -1). The corresponding characters are printed in the table headers.
- It writes the closing tags for the current row of the table and starts a new row. It prints the pattern length in the first cell of the new row and then iterates over the **bad_char** array again. It prints the corresponding values from the **bad_char** array in the table cells.
- After completing the "Bad Symbol Table" section, it writes HTML code for the "Good Symbol Table" section. It starts a new table and writes table headers for "k", "Pattern", and "Shift".
- It enters a loop to print rows for each index in the **good** array. For each row, it prints the row number in the first cell, then iterates over the pattern characters. If the current character matches the position determined by **patternLen - i - 1**, it adds an underline tag **<u>** to the HTML output.
- After printing the pattern characters, it closes the underline tag, writes the closing tags for the current row, and moves to the next row.
- The function continues by writing HTML code for other sections of the file, including an "Output" heading.
- Finally, it closes the file using **fclose(output)**.

In summary, the **addList** function generates an HTML file that contains tables for a "Bad Symbol Table" and a "Good Symbol Table". It utilizes the provided data, such as pattern string, character arrays, and pattern length, to populate the tables with appropriate values. The function writes HTML code for styling, table structure, and content.

```
//-------------------------------------------------------------------------------------//
// Utility function to print a portion of a string to a file
void printOutputNStr(FILE *output, const char *str, const int num) {
    int i;

    // Iterate over the characters in the string up to the specified number
    for (i = 0; i < num; i++) {
        // Write each character to the output file
        fputc(str[i], output);
    }

    return;
}
//-------------------------------------------------------------------------------------//
```

This is pritnOutputNStr() Function : -

The provided code is an implementation of a utility function called **printOutputNStr**. Here's a detailed breakdown of the function:

- The function is defined with a return type of **void** and takes three parameters: **output** (a file pointer indicating the output file), **str** (a pointer to the string to be printed), and **num** (an integer representing the number of characters to be printed).
- The function declares an integer variable **i** that will be used as a loop counter.
- The function enters a **for** loop that iterates from 0 to **num - 1**.
- Inside the loop, the function uses the **fputc** function to write each character of the string **str[i]** to the output file indicated by the **output** file pointer.
- Once all the characters have been written, the loop ends.
- The function ends and returns.

The purpose of this utility function is to print a portion of a string to a file. It provides a convenient way to write a specific number of characters from a string to a specified output file.

```
//-------------------------------------------------------------------------------------//
// Utility function to insert <mark></mark> tag around the pattern
void insert_mark_tag(const char *outPutS, const char *text, const char *pattern, int *count) {
    const char *mark_open = "<mark>";
    const char *mark_close = "</mark>";
    FILE *output = fopen(outPutS,"a");
    size_t text_len = strlen(text);
    size_t pattern_len = strlen(pattern);
    size_t mark_open_len = strlen(mark_open);
```

```c
size_t mark_close_len = strlen(mark_close);

// Calculate the required memory for the highlighted text

// Set the starting position for writing the highlighted text, accounting for additional characters

const char *src = text;

int idx = 0;
int temp;
int lastIndexE = -1;
int lastIndexS = 0;

// Iterate over the source text
while (src < text + text_len) {
    // Check if the current portion of the source matches the pattern
    if (strncmp(src, pattern, pattern_len) == 0) {
        // Insert the opening <mark> tag
        fprintf(output,mark_open);

        // Copy the text between the last match and the current match
        if (idx >= lastIndexE) {
            printOutputNStr(output, src, pattern_len);
        } else {
            temp = lastIndexE - idx;
            printOutputNStr(output, src + temp, idx - lastIndexS);
        }

        // Insert the closing </mark> tag
        fprintf(output, mark_close);

        // Update the indices of the last match positions
        lastIndexE = idx + pattern_len;
        lastIndexS = idx;
    } else {
        // Copy non-matching characters
        if (idx >= lastIndexE) {
            fputc(*src, output);
        }
    }
    // Move to the next character
    *src++;
    idx++;
}
```

```
    // Append the closing </p> tag
    fprintf(output,"</p>\0");

    return;
}
//-----------------------------------------------------------------------------------------//
```

This is insert_mark_tag() Function: -

The provided code is an implementation of a utility function called **insert_mark_tag**. Here's a detailed breakdown of the function:

- The function takes three parameters: **outPutS** (the output file name), **text** (the original text), **pattern** (the pattern to be highlighted), and a pointer to an integer variable **count**.
- The function declares two constant strings **mark_open** and **mark_close**, which represent the opening and closing HTML tags **<mark>** and **</mark>**.
- The function opens the output file in append mode using **fopen** and assigns the file pointer to the variable **output**.
- The function calculates the lengths of the **text**, **pattern**, **mark_open**, and **mark_close** strings using the **strlen** function.
- The function sets the starting position for writing the highlighted text. The **src** pointer is initialized to **text**.
- The function initializes several integer variables: **idx** (used to keep track of the current position in the source text), **temp** (a temporary variable), **lastIndexE** (the index of the end position of the last match), and **lastIndexS** (the index of the start position of the last match).
- The function enters a loop that iterates over the characters in the source text. The loop continues until the **src** pointer reaches the end of the **text**.
- Within the loop, the function checks if the current portion of the source text matches the **pattern**. This is done using the **strncmp** function, which compares a substring of **src** with the **pattern** for a length of **pattern_len**.
- If a match is found, the function performs the following steps:
    - It inserts the opening **<mark>** tag into the output file using **fprintf**.
    - It copies the text between the last match and the current match position into the output file. The **printOutputNStr** function is called to handle this copying based on the last match positions and the length of the copied text.
    - It inserts the closing **</mark>** tag into the output file using **fprintf**.
    - It updates the indices **lastIndexE** and **lastIndexS** to reflect the new last match positions.
- If a match is not found, the function copies the non-matching characters from the source text to the output file using **fputc**.
- After processing all the characters in the source text, the function appends the closing **</p>** tag to the output file using **fprintf**.
- The function ends and returns.

The purpose of this function is to insert HTML **<mark>** tags around the occurrences of the pattern in the original text. It searches for matches by comparing substrings of the source text with the pattern and copies the appropriate sections of the text to the output file, while inserting the opening and closing **<mark>** tags as necessary. This function helps highlight the occurrences of the pattern in the text for visual representation.

```c
//-----------------------------------------------------------------------------//
// Brute-force string matching
int brute_force(const char *text, const char *pattern, int *comparisons) {
    int text_len = strlen(text);
    int pattern_len = strlen(pattern);
    int count = 0;

    // Iterate over the text using a sliding window approach
    for (int i = 0; i <= text_len - pattern_len; i++) {
        int j;

        // Compare characters of the pattern with the corresponding characters in the text
        for (j = 0; j < pattern_len; j++) {
            (*comparisons)++;  // Increment the comparison counter

            // If a mismatch is found, break out of the inner loop
            if (text[i + j] != pattern[j]) {
                break;
            }
        }

        // If the inner loop completed without a break, a match is found
        if (j == pattern_len) {
            count++;  // Increment the match counter
        }
    }

    // Return the total number of matches found
    return count;
}
//-----------------------------------------------------------------------------//
```

This is brute_force() Function: -

- The code defines a function named **brute_force** that performs brute-force string matching. It takes three parameters: **text**, **pattern**, and **comparisons**. The **text** parameter represents the source text where the matching will be performed, the **pattern** parameter represents the pattern to be matched, and the **comparisons** parameter is a pointer to an integer that will keep track of the number of character comparisons made.
- The function starts by calculating the lengths of the **text** and **pattern** strings using the **strlen** function.
- It initializes a variable **count** to keep track of the number of matches found.

- The function enters a loop that iterates over the **text** string using a sliding window approach. The loop variable **i** represents the starting position of the window.
- Inside the loop, it initializes a variable **j** that will be used to iterate over the characters of the **pattern** string.
- It enters another loop that compares the characters of the **pattern** with the corresponding characters in the **text**. The loop variable **j** represents the current index in the **pattern** string.
- For each comparison, it increments the **comparisons** counter using the **comparisons** pointer.
- If a mismatch is found (i.e., the characters at the corresponding positions in **text** and **pattern** are not equal), it breaks out of the inner loop.
- If the inner loop completes without a break, it means that a match is found. It increments the **count** variable to keep track of the matches.
- After processing the current window, the outer loop moves the window by incrementing the loop variable **i** by 1.
- The loop continues until the entire **text** string is covered by the sliding window, i.e., until **i <= text_len - pattern_len**.
- Finally, the function returns the total number of matches found (**count**).

In summary, the **brute_force** function implements a simple brute-force algorithm for string matching. It compares the **pattern** with substrings of the **text** by iterating over the **text** using a sliding window approach. For each position of the window, it compares the characters of the **pattern** with the corresponding characters in the **text**. It keeps track of the number of comparisons made and the number of matches found. The function returns the total count of matches.

```
//-------------------------------------------------------------------------------------//
// Print the Bad Symbol Table
void print_shift_table(const int ShiftTable[MAX_CHAR]) {
    printf("Shift Table:\n");  // Print a header indicating the name of the table
    for (int i = 1; i < MAX_CHAR; i++) {
        if (ShiftTable[i] != -1) {
            printf("%c: %d\n", (char)i, ShiftTable[i]);  // Print the character and its corresponding
distance
        }
    }
    printf("\n");  // Print a blank line for readability
}
//-------------------------------------------------------------------------------------//
```

This is print_shift_table() Function: -

The provided code is a function named **print_shift_table** that takes an array **ShiftTable** as input and prints the contents of the table. Here's a breakdown of the code:

1. The function signature indicates that the function returns **void**, meaning it does not return any value.
2. The function is defined as **void print_shift_table(const int ShiftTable[MAX_CHAR])**, which means it takes an array **ShiftTable** as a parameter. The parameter is declared as **const int**, indicating that the function will not modify the contents of the array.

3. The function starts with the statement **printf("Shift Table:\n");**, which prints a header indicating the name of the table. This will be the first line printed when the function is called.

4. The function then enters a **for** loop, starting from **i = 1** and continuing until **i** is less than **MAX_CHAR**. **MAX_CHAR** is assumed to be a predefined constant representing the maximum number of characters in the table.

5. Inside the **for** loop, the code checks if the value at index **i** of the **ShiftTable** array is not equal to **-1**. If this condition is true, it means there is a valid entry for that character in the table.

6. If the condition is true, the code executes the following statement: **printf("%c: %d\n", (char)i, ShiftTable[i]);**. This statement prints the character corresponding to the current index **i** and its corresponding distance stored in **ShiftTable[i]**. The **%c** format specifier is used to print the character, and **%d** is used to print the distance as an integer.

7. After the **for** loop, the code executes the statement **printf("\n");**, which prints a blank line for readability. This will be the last line printed when the function is called.

In summary, the **print_shift_table** function takes an array as input and prints the contents of the array, specifically the character and its corresponding distance for each valid entry in the table. The function adds a header before printing the entries and includes a blank line at the end for readability.

```c
//-------------------------------------------------------------------------------//
// Horspool's algorithm
// Function to generate the shift table for Horspool's algorithm
void shift_Table(char *pattern, int PatternLength, int ShiftTable[MAX_CHAR]) {

    // Initialize the shift table with the maximum shift value (PatternLength)
    for (int i = 0; i < MAX_CHAR; i++) {
        ShiftTable[i] = -1;
    }

    // Compute the shift value for each character in the pattern except the last one
    for (int i = 0; i < PatternLength - 1; i++) {
        ShiftTable[(int)pattern[i]] = PatternLength - i - 1;
    }
    print_shift_table(ShiftTable);
}
//-------------------------------------------------------------------------------//
```

This is shift_Table() Function: -

The provided code is a function named **shift_Table** that generates a shift table for Horspool's algorithm. Here's a breakdown of the code:

- The code includes a comment indicating that it is implementing Horspool's algorithm.
- The function signature is **void shift_Table(char *pattern, int PatternLength, int ShiftTable[MAX_CHAR])**, which means it takes three parameters:
    - **pattern**: a pointer to a character array representing the pattern to search for.

- **PatternLength**: an integer indicating the length of the pattern.
- **ShiftTable**: an integer array used to store the shift values.

- The function starts by initializing the shift table with the maximum shift value (PatternLength). It uses a **for** loop that iterates over all elements of the **ShiftTable** array and assigns the value -1 to each element. This effectively sets all entries in the shift table to -1.
- The function then enters another **for** loop that computes the shift value for each character in the pattern except the last one. The loop iterates from 0 to **PatternLength - 1**, excluding the last character.
- Inside the loop, the code accesses the **pattern** array at index **i** and converts it to an integer using **(int)pattern[i]**. This conversion is done because the shift table uses the integer representation of characters as indices.
- The code then assigns the shift value to the corresponding index in the **ShiftTable** array. The shift value is computed as **PatternLength - i - 1**, which represents the distance between the current character and the last character of the pattern.
- After computing the shift values, the code calls the function **print_shift_table(ShiftTable)** to print the contents of the shift table. This function is assumed to be defined elsewhere and is responsible for printing the table.

In summary, the **shift_Table** function generates a shift table for Horspool's algorithm. It initializes the table with the maximum shift value, then computes the shift value for each character in the pattern except the last one. Finally, it calls a separate function to print the contents of the shift table.

```
//-------------------------------------------------------------------------------------//
//-------------------------------------------------------------------------------------//
// Horspool's algorithm for string matching
int Horspool_Search(char *text, int TextLength, char *pattern, int PatternLength, int *comparisons) {
    int ShiftTable[MAX_CHAR];
    shift_Table(pattern, PatternLength, ShiftTable);  // Generate the shift table
    int shift;
    int count = 0;
    int i = 0;

    // Main loop to search for the pattern in the text
    while (i <= TextLength - PatternLength) {
        int j = PatternLength - 1;

        // Compare characters from right to left until a mismatch is found or the pattern is fully matched
        while (j >= 0 && ++(*comparisons) && (pattern[j] == text[i + j])) {
            j--;
        }
        //Shifting
        shift = ShiftTable[(int)text[i + PatternLength - 1]];
        i += shift > 0 ? shift : PatternLength;

        //Full match occurs
```

```
        if (j < 0) {
            count++;  // Increment the match counter
        }
    }

    return count;  // Return the total number of matches found
}
//-----------------------------------------------------------------------------------------//
```

This is Horspool_Search() Function: -

The provided code implements Horspool's algorithm for string matching. Here's a detailed breakdown of the algorithm:

1. The function **Horspool_Search** takes several parameters: **text** (the text to search within), **TextLength** (the length of the text), **pattern** (the pattern to find within the text), **PatternLength** (the length of the pattern), and a pointer to an integer variable **comparison** that will be used to keep track of the number of character comparisons made during the search. The function returns the count of occurrences of the pattern within the text.
2. The function declares an integer array **ShiftTable** to hold the shift values for each character in the pattern.
3. The function calls the **shift_Table** function, passing **pattern**, **PatternLength**, and **ShiftTable** as arguments, to generate the shift table. The shift table determines the amount to shift the pattern for efficient skipping of unnecessary comparisons.
4. The function initializes variables **shift**, **count**, and **i**.
    - **shift** stores the shift value based on the last character of the current pattern match.
    - **count** keeps track of the number of occurrences of the pattern found in the text.
    - **i** is an index variable used in the main loop to search for the pattern in the text.
5. The function enters a main loop that continues until **i** reaches **TextLength - PatternLength**. This ensures that there are enough characters remaining in the text to match the pattern.
6. Within the main loop, the function initializes **j** to **PatternLength - 1**, representing the index of the rightmost character in the pattern.
7. The algorithm compares characters from right to left in the pattern and text until a mismatch is found or the pattern is fully matched:
    - The condition **j >= 0** ensures that the loop continues until the leftmost character of the pattern is reached.
    - The condition **++(*comparisons)** increments the **comparisons** counter and ensures that each character comparison is counted.
    - The condition **(pattern[j] == text[i + j])** checks if the characters at the corresponding positions in the pattern and text match.
    - If a match is found, **j** is decremented to continue the comparison.
    - If a mismatch is found, the inner loop ends.

8. After the inner loop ends, the function calculates the shift value based on the last character of the current pattern match:

    - **shift** is assigned the value from the **ShiftTable** array at the index corresponding to the last character of the pattern match in the text.

9. The variable **i** is updated by adding the maximum value between **shift** and **PatternLength** to move the index to the next possible match position.

10. If **j** is less than 0, it means a full match of the pattern has occurred. The **count** variable is incremented to record the match.

11. The main loop continues with the next possible match position.

12. Once the main loop finishes, the function returns the count of occurrences of the pattern within the text.

Horspool's algorithm improves the efficiency of string matching by utilizing a shift table to determine the optimal amount to shift the pattern for skipping unnecessary comparisons. It can be efficient for searching patterns in long texts and has a complexity of O(n/m), where n is the length of the text and m is the length of the pattern.

```c
//-------------------------------------------------------------------------------------//
// Horspool's algorithm for string matching
void Horspool_Algorithm(char *text, char *pattern) {
    int count = 0;
    int TextLength = strlen(text);
    int PatternLength = strlen(pattern);
    int comparisons = 0;

    // Start measuring execution time
    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);

    // Perform Horspool's algorithm to find matches
    count = Horspool_Search(text, TextLength, pattern, PatternLength, &comparisons);

    // Stop measuring execution time
    clock_gettime(CLOCK_MONOTONIC, &end_time);

    // Calculate the execution time in milliseconds
    double execution_time = (end_time.tv_sec - start_time.tv_sec) * 1000.0; // seconds to milliseconds
    execution_time += (end_time.tv_nsec - start_time.tv_nsec) / 1000000.0; // nanoseconds to milliseconds

    // Print the results
    printf("Occurrences: %d\n", count);
    printf("Comparisons: %d\n", comparisons);
    printf("Running time: %.4f ms\n", execution_time);
}
//-------------------------------------------------------------------------------------//
```

This is Horspool_Algorithm() Function: -

- The code defines a function named **Horspool_Algorithm** that performs Horspool's algorithm for string matching. It takes two parameters: **text** and **pattern**, which represent the text to search in and the pattern to search for, respectively.
- The function declares variables **count**, **TextLength**, **PatternLength**, and **comparisons**. **count** will store the number of occurrences of the pattern in the text, **TextLength** will store the length of the text, **PatternLength** will store the length of the pattern, and **comparisons** will keep track of the number of character comparisons made during the search process.
- It uses **strlen** to determine the lengths of the **text** and **pattern** strings and assigns the respective values to **TextLength** and **PatternLength**.
- The function declares **start_time** and **end_time** variables of type **struct timespec** to measure the execution time.
- It calls **clock_gettime** with **CLOCK_MONOTONIC** to obtain the starting time before executing the algorithm.
- The function calls **Horspool_Search** to perform the actual string-matching using Horspool's algorithm. It passes the **text**, **TextLength**, **pattern**, **PatternLength**, and the address of **comparisons** to track the number of comparisons made.
- After the search is complete, it calls **clock_gettime** again to obtain the ending time.
- The code calculates the execution time by subtracting the starting time from the ending time. It converts the time values to milliseconds and assigns the result to **execution_time**.
- The function then prints the results, including the number of occurrences of the pattern (**count**), the number of character comparisons made (**comparisons**), and the total running time in milliseconds (**execution_time**).

In summary, the **Horspool_Algorithm** function encapsulates the entire process of performing string matching using Horspool's algorithm. It measures the execution time, tracks the number of comparisons, and displays the results.

```c
//-----------------------------------------------------------------------------//
// Boyer-Moore algorithm
// preprocess_bad_character -> generate bad character table
void preprocess_bad_character(const char *pattern, int pattern_len, int bad_char[MAX_CHAR]) {
    // Initialize the bad character array with -1
    for (int i = 0; i < MAX_CHAR; i++) {
        bad_char[i] = -1;
    }

    // Compute the bad character table
    for (int i = 0; i < pattern_len - 1; i++) {
        bad_char[pattern[i]] = pattern_len - i - 1;
    }
}
//-----------------------------------------------------------------------------//
```

This is preprocess_bad_character() Function: -

- The code defines a function named **preprocess_bad_character** that is part of the Boyer-Moore algorithm. It is responsible for generating the bad character table used in the algorithm. The function takes three

parameters: **pattern**, which is the pattern to search for, **pattern_len**, which is the length of the pattern, and **bad_char**, which is an array representing the bad character table.

- The function initializes the **bad_char** array with -1 for all characters. The **bad_char** array is used to store the last occurrence position of each character in the pattern.
- It uses a **for** loop to iterate over each element of the **bad_char** array, setting the initial value of each element to -1.
- Another **for** loop is used to compute the bad character table. It iterates over each character in the pattern except for the last one.
- Inside the loop, it assigns the value **pattern_len - i - 1** to the **bad_char** array at the index corresponding to the current character. This value represents the distance between the last occurrence of the character in the pattern and the end of the pattern.
- After the loop is completed, the **bad_char** array is populated with the appropriate values representing the bad character table.

In summary, the **preprocess_bad_character** function is a utility function within the Boyer-Moore algorithm. It generates the bad character table by initializing the array with -1 and then assigning the appropriate values based on the last occurrence of each character in the pattern. The bad character table is used to determine the maximum shift distance when a mismatch occurs during the pattern matching process.

```
//-----------------------------------------------------------------------------//
// preprocess_good_suffix -> generate good suffix table
void preprocess_good_suffix(const char *pattern, int pattern_len, int good_suffix[]) {
    int i, j, k, l, m;

    // Initialize the good suffix table with default values
    for(i = 0; i < pattern_len - 1; i++){
        good_suffix[i] = pattern_len - 1;
    }
    good_suffix[pattern_len - 1] = pattern_len;

    // Calculate the values for the good suffix table
    for(int i = pattern_len - 1, l = 0; i > 0; i--, l++){
        for(k = pattern_len - 2; k >= 0; k--){
            // Check if the pattern[k - l - 1] matches the pattern[i - 1]
            if(k - l - 1 >= 0 && pattern[k - l - 1] == pattern[i - 1])
                continue;

            // Compare the characters in the suffix and the pattern
            for(j = k, m = 0; j >= k - l && j >= 0; j--, m++){
                if(pattern[j] != pattern[pattern_len - m - 1])
                    break;
            }

            // If the suffix matches the pattern, update the good suffix value
```

```
            if(j < k - l || j < 0){
                good_suffix[l] = pattern_len - k - 1;
                break;
            }
        }
    }

    // Handle the special case where the pattern ends with a good suffix
    k = 0;
    for(i = pattern_len - 1; i > 0; i--, k++){
        // Compare the characters in the prefix and the pattern
        for(j = 0; j <= k; j++){
            if(pattern[j] != pattern[i + j])
                break;
        }

        // If the prefix matches the pattern, update the good suffix value
        if(j > k){
            good_suffix[pattern_len - 1] = i;
        }
    }
}
//-----------------------------------------------------------------------------//
```

This is preprocess_good_suffix() Function: -

- The code defines a function named **preprocess_good_suffix** responsible for generating the good suffix table used in the Boyer-Moore algorithm. The function takes three parameters: **pattern**, which is the pattern to search for, **pattern_len**, which is the length of the pattern, and **good_suffix**, which is an array representing the good suffix table.
- The function starts by initializing the **good_suffix** array with default values. It assigns **pattern_len - 1** to each element in the array, except for the last element, which is set to **pattern_len**. This initialization assumes no matches are found in the pattern initially.
- Next, the function calculates the values for the good suffix table. It uses two nested loops, **for** and **for**, to iterate over the pattern.
- The outer loop starts from the second-to-last position of the pattern and iterates backward to the first position. It uses **i** as the index and **l** to keep track of the distance from the end of the pattern.
- Inside the outer loop, the inner loop starts from the second-to-last position of the pattern and iterates backward to the first position. It uses **k** as the index.
- Inside the inner loop, the code checks if the characters at **pattern[k - l - 1]** and **pattern[i - 1]** match. If they match, it continues to the next iteration of the inner loop.
- If the characters at **pattern[k - l - 1]** and **pattern[i - 1]** do not match, the code proceeds to compare the characters in the suffix (from **k - l** to **k**) and the characters in the pattern (from the end of the pattern to **pattern_len - m - 1**). It uses **j** to iterate through the suffix and **m** to iterate through the pattern.

- If a mismatch is found or the suffix is exhausted (i.e., **j < k - l** or **j < 0**), the code updates the corresponding value in the **good_suffix** table to **pattern_len - k - 1** and breaks out of the inner loop.
- After the nested loops complete, the code handles the special case where the pattern ends with a good suffix. It starts from the second-to-last position of the pattern and iterates backward to the first position.
- Inside this loop, it compares the characters in the prefix (from 0 to **k**) and the characters in the pattern (from **i** to **i + j**). It uses **j** to iterate through the prefix.
- If the prefix matches the pattern, the code updates the last element of the **good_suffix** table to **i**.

In summary, the **preprocess_good_suffix** function generates the good suffix table by initializing the array with default values and then calculating the appropriate values based on the pattern. The good suffix table is used in the Boyer-Moore algorithm to determine the maximum shift distance when a mismatch occurs during the pattern matching process.

```c
//-------------------------------------------------------------------------------------------//
// boyer_moore Algorithm
int boyer_moore(const char *text, const char *pattern, int *comparisons) {
    int text_len = strlen(text);
    int pattern_len = strlen(pattern);
    int bad_char[MAX_CHAR];
    int good_suffix[pattern_len];
    int count = 0;

    // Preprocess the bad character and good suffix tables
    preprocess_bad_character(pattern, pattern_len, bad_char);
    preprocess_good_suffix(pattern, pattern_len, good_suffix);

    int i, j, k, shift;
    for (i = pattern_len - 1; i < text_len; i += shift) {
        // Compare the pattern with the text from right to left
        for (j = i, k = 0; k < pattern_len; k++, j--) {
            ++(*comparisons);
            if (text[j] != pattern[pattern_len - k - 1]) {
                break;
            }
        }

        if (k == pattern_len) {
            // The pattern is found
            count++;
            shift = 1;
        } else if (k == 0) {
            // Mismatch at the first character of the pattern
            shift = bad_char[text[j]];
            if (shift == -1)
                shift = pattern_len;
```

```
        } else {
            // Mismatch at a character within the pattern
            shift = max(good_suffix[k - 1], max(1, bad_char[text[i]] - k + 1));
        }
    }

    return count;
}
//-----------------------------------------------------------------------------------//
```

This is boyer_moore() Function: -

The provided code implements the Boyer-Moore algorithm for string searching. Here's a detailed breakdown of the algorithm:

- The function **boyer_moore** takes two parameters: **text** (the text to search within) and **pattern** (the pattern to find within the text). It also takes a pointer to an integer variable **comparison** that will be used to keep track of the number of character comparisons made during the search. The function returns the count of occurrences of the pattern within the text.
- The function starts by initializing some variables:
  - **text_len** stores the length of the text.
  - **pattern_len** stores the length of the pattern.
  - **bad_char** is an integer array used for preprocessing the bad character table.
  - **good_suffix** is an integer array used for preprocessing the good suffix table.
  - **count** keeps track of the number of occurrences of the pattern found in the text.
- Two preprocessing functions, **preprocess_bad_character** and **preprocess_good_suffix**, are called to populate the **bad_char** and **good_suffix** arrays, respectively. These preprocessing steps help improve the efficiency of the algorithm.
- The main loop of the algorithm begins with the initialization of variables **i**, **j**, **k**, and **shift**. The loop continues until **i** reaches the end of the text.
- Within the loop, the algorithm compares the pattern with the text from right to left:
  - The inner loop compares characters of the pattern (**pattern[pattern_len - k - 1]**) with the corresponding characters in the text (**text[j]**).
  - The variable **k** keeps track of the number of characters matched so far.
  - The variable **j** starts at **i** and is decremented for each iteration of the inner loop.
- After the inner loop finishes, there are three possibilities:
  - If **k** is equal to **pattern_len**, it means that a complete match has been found. The count is incremented, and **shift** is set to 1 to move to the next possible match.

- If **k** is 0, it means there was a mismatch at the first character of the pattern. The **bad_char** array is used to determine the shift value based on the mismatched character in the text. If the mismatched character doesn't exist in the pattern, the shift is set to the length of the pattern.
- If **k** is neither 0 nor equal to **pattern_len**, it means there was a mismatch within the pattern. The **good_suffix** array and the **bad_char** array are used to determine the shift value.

- The loop continues with the next possible match position by updating **i** with **shift**.
- Once the loop finishes, the function returns the count of occurrences of the pattern within the text.

The Boyer-Moore algorithm improves the efficiency of string searching by utilizing two tables, the bad character table and the good suffix table, for efficient skipping of unnecessary comparisons. It can be particularly efficient when searching for patterns in long texts.

```
//-------------------------------------------------------------------------------------------//
// Print the Bad Symbol Table
void print_bad_symbol_table(const int bad_char[MAX_CHAR]) {
    printf("Bad Symbol Table:\n");  // Print a header indicating the name of the table
    for (int i = 1; i < MAX_CHAR; i++) {
        if (bad_char[i] != -1) {
            printf("%c: %d\n", (char)i, bad_char[i]);  // Print the character and its corresponding
distance
        }
    }
    printf("\n");  // Print a blank line for readability
}
//-------------------------------------------------------------------------------------------//
```

This is print_bad_symbol_table() Function: -

- The code defines a function named **print_bad_symbol_table** that takes an array **bad_char** as input. This array represents the bad character table generated during the preprocessing step of the Boyer-Moore algorithm.
- The function starts by printing a header line "Bad Symbol Table" to indicate the name of the table.
- It then enters a loop that iterates over the elements of the **bad_char** array. The loop variable **i** starts from 1, as the **bad_char** table typically contains information for characters with ASCII values greater than 0.
- Inside the loop, the code checks if the value at **bad_char[i]** is not equal to -1. This condition ensures that only characters present in the pattern are printed, as -1 represents characters not present in the pattern.
- If the condition is true, the code executes the **printf** statement to print the character and its corresponding distance. It uses the **(char)i** typecast to convert the integer value **i** back to its corresponding character representation.
- After printing the bad symbol table, the code prints a newline character **\n** to add a blank line for readability.

In summary, the **print_bad_symbol_table** function is responsible for printing the bad character table generated during the preprocessing step of the Boyer-Moore algorithm. It iterates over the table, excluding characters not present in the pattern, and prints each character along with its corresponding distance. The function provides a visual representation of the bad character table for analysis and debugging purposes.

```c
//-----------------------------------------------------------------------------------------//
// Print the Good Suffix Table
void print_good_suffix_table(const int good_suffix[], int pattern_len) {
    printf("Good Suffix Table:\n");  // Print a header indicating the name of the table
    int j = 1;
    for (int i = pattern_len - 1; i >= 0; i--) {
        printf("%d: %d\n", j, good_suffix[j - 1]);  // Print the index and its corresponding length
        j++;
    }
    printf("\n");  // Print a blank line for readability
}
//-----------------------------------------------------------------------------------------//
```

This is print_good_suffix_table() Function: -

- The code defines a function named **print_good_suffix_table** that takes two arguments: an array **good_suffix** and an integer **pattern_len**. The **good_suffix** array represents the good suffix table generated during the preprocessing step of the Boyer-Moore algorithm, and **pattern_len** represents the length of the pattern.
- The function starts by printing a header line "Good Suffix Table" to indicate the name of the table.
- It initializes a variable **j** with the value 1. This variable is used to keep track of the index while printing the table.
- The code enters a loop that iterates over the elements of the **good_suffix** array in reverse order, starting from **pattern_len - 1** and decrementing **i** at each iteration.
- Inside the loop, the code executes the **printf** statement to print the index **j** and its corresponding length **good_suffix[j - 1]**. The index **j** starts from 1, and **good_suffix** is accessed at index **j - 1** to match the array indexing convention.
- After printing each entry of the good suffix table, the code increments **j** to move to the next index.
- Once the loop is completed, the code prints a newline character **\n** to add a blank line for readability.

In summary, the **print_good_suffix_table** function is responsible for printing the good suffix table generated during the preprocessing step of the Boyer-Moore algorithm. It iterates over the table in reverse order, starting from the highest index, and prints each index along with its corresponding length. The function provides a visual representation of the good suffix table for analysis and debugging purposes.

```c
//-----------------------------------------------------------------------------------------//
// Main Function -> Execution starts here!
int main(int argc, char *argv[]) {
    /* Command line argument check
    if (argc != 3) {
        printf("Usage: %s <html_filename> <pattern>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    */
```

```
    struct timespec start_time, end_time;
      char html_filename[100];
    printf("Html file name (example.html):");
    scanf("%s", html_filename);

    char pattern[1000];
    printf("Pattern: ");
    scanf("%s", pattern);

    char *text = read_file(html_filename);  // Read the contents of the HTML file into a character array
    int comparisons;
    int bad_char[MAX_CHAR];
    int good_suffix[strlen(pattern)];

    // Brute-force Algorithm-----------------------------------------------------------------//

    comparisons = 0;

    clock_gettime(CLOCK_MONOTONIC, &start_time);
    int brute_force_count = brute_force(text, pattern, &comparisons);  // Call the brute force algorithm
function
    clock_gettime(CLOCK_MONOTONIC, &end_time);

    double execution_time_brute = (end_time.tv_sec - start_time.tv_sec) * 1000.0;  // Convert the time to
milliseconds
    execution_time_brute += (end_time.tv_nsec - start_time.tv_nsec) / 1000000.0;

    printf("***-------------------------------------------------------------------------------------------
***\n");
    printf("Brute-force Algorithm:\nOccurrences: %d\nComparisons: %d\nRunning time: %.4f ms\n",
brute_force_count, comparisons, execution_time_brute);
    printf("***-------------------------------------------------------------------------------------------
***\n");

    // Horspool's Algorithm------------------------------------------------------------------//

    comparisons = 0;
    printf("***-------------------------------------------------------------------------------------------
***\n");
    printf("Horspool Algorithm:\n");
    preprocess_bad_character(pattern, strlen(pattern), bad_char);  // Preprocess the bad character table
    Horspool_Algorithm(text, pattern);  // Call the Horspool's algorithm function
    printf("***-------------------------------------------------------------------------------------------
***\n");
```

```
    // Boyer-Moore Algorithm-----------------------------------------------------------------//

    printf("***-------------------------------------------------------------------------------------------
***\n");
    printf("Boyer-Moore Algorithm:\n");

    preprocess_bad_character(pattern, strlen(pattern), bad_char);  // Preprocess the bad character table
    preprocess_good_suffix(pattern, strlen(pattern), good_suffix);  // Preprocess the good suffix table

    print_bad_symbol_table(bad_char);  // Print the bad symbol table
    print_good_suffix_table(good_suffix, strlen(pattern));  // Print the good suffix table

    comparisons = 0;

    clock_gettime(CLOCK_MONOTONIC, &start_time);
    int boyer_moore_count = boyer_moore(text, pattern, &comparisons);  // Call the Boyer-Moore algorithm
function
    clock_gettime(CLOCK_MONOTONIC, &end_time);

    double execution_time_boyer_moore = (end_time.tv_sec - start_time.tv_sec) * 1000.0;  // Convert the
time to milliseconds
    execution_time_boyer_moore += (end_time.tv_nsec - start_time.tv_nsec) / 1000000.0;

    printf("Occurrences: %d\nComparisons: %d\nRunning time: %.4f ms\n\n", boyer_moore_count, comparisons,
execution_time_boyer_moore);
    printf("***-------------------------------------------------------------------------------------------
***\n");

    // Highlight occurrences in the text and save the output
    addList("output.html", pattern, bad_char, good_suffix, strlen(pattern));  // Create an HTML file with
tables for the bad symbol table and good symbol table
    insert_mark_tag("output.html",text, pattern, &brute_force_count);  // Insert HTML mark tags to
highlight occurrences of the pattern in the text


    //free(highlighted);
    free(text);

    return 0;
}
//-------------------------------------------------------------------------------------------//
```

This is main() Function: -

The provided code is the main function that executes the pattern matching algorithms, including the brute-force algorithm, Horspool's algorithm, and Boyer-Moore algorithm. Here's a breakdown of the code:

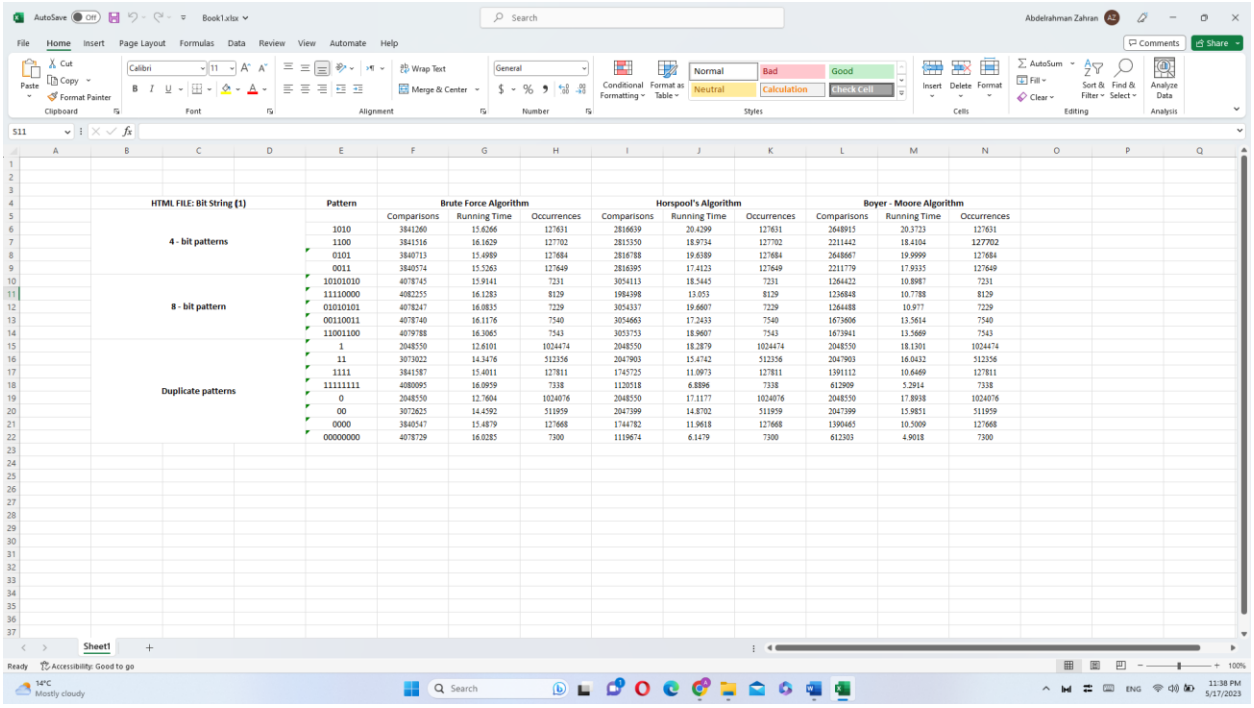
- The main function begins with commented code that checks the number of command line arguments (**argc**). If the number of arguments is not equal to 3, it prints a usage message indicating the required arguments and exits the program.
- The function declares a struct **timespec** variable named **start_time** and **end_time**. This structure is used to measure the execution time of the algorithms.
- The function declares an array **html_filename** with a size of 100 to store the name of the HTML file. It prompts the user to enter the HTML file name using **printf** and reads the input using **scanf**.
- The function declares an array **pattern** with a size of 1000 to store the pattern to search for. It prompts the user to enter the pattern using **printf** and reads the input using **scanf**.
- The function calls the **read_file** function, passing the **html_filename** array as an argument. This function reads the contents of the HTML file into a character array and returns a pointer to it. The result is assigned to the **text** variable.
- The function declares variables **comparisons**, **bad_char**, and **good_suffix**. **comparisons** is an integer used to count the number of comparisons made by the algorithms. **bad_char** and **good_suffix** are integer arrays used to store the bad character table and good suffix table for the Boyer-Moore algorithm.
- The function sets **comparisons** to 0, indicating that no comparisons have been made yet.
- The function calls the **brute_force** function, passing the **text**, **pattern**, and **comparisons** variables as arguments. This function performs the brute-force algorithm for pattern matching and returns the count of occurrences. The result is assigned to the **brute_force_count** variable.
- The function uses **clock_gettime** to measure the start time before calling the **brute_force** function and measures the end time afterward.
- The function calculates the execution time of the brute-force algorithm by subtracting the start time from the end time, converting the result to milliseconds. The result is stored in the **execution_time_brute** variable.
- The function prints information about the brute-force algorithm, including the number of occurrences, the number of comparisons made, and the execution time.
- The function calls the **preprocess_bad_character** function, passing the **pattern**, **strlen(pattern)**, and **bad_char** variables as arguments. This function preprocesses the bad character table for the Boyer-Moore algorithm.
- The function calls the **Horspool_Algorithm** function, passing the **text** and **pattern** variables as arguments. This function implements Horspool's algorithm for pattern matching.
- The function calls the **preprocess_bad_character** function again to preprocess the bad character table for the Boyer-Moore algorithm.
- The function calls the **preprocess_good_suffix** function, passing the **pattern**, **strlen(pattern)**, and **good_suffix** variables as arguments. This function preprocesses the good suffix table for the Boyer-Moore algorithm.
- The function calls the **print_bad_symbol_table** function, passing the **bad_char** array as an argument. This function prints the contents of the bad character table.

- The function calls the **print_good_suffix_table** function, passing the **good_suffix** array and **strlen(pattern)** as arguments. This function prints the contents of the good suffix table.
- The function sets **comparisons** to 0, indicating that no comparisons have been made yet.
- The function uses **clock_gettime** to measure the start time before calling the **boyer_moore** function and measures the end time afterward.
- The function calculates the execution time of the Boyer-Moore algorithm by subtracting the start time from the end time, converting the result to milliseconds. The result is stored in the **execution_time_boyer_moore** variable.
- The function prints information about the Boyer-Moore algorithm, including the number of occurrences, the number of comparisons made, and the execution time.
- The function calls the **addList** function, passing the **"output.html"**, **pattern**, **bad_char**, **good_suffix**, and **strlen(pattern)** as arguments. This function creates an HTML file with tables for the bad character table and good suffix table.
- The function calls the **insert_mark_tag** function, passing the **"output.html"**, **text**, **pattern**, and **brute_force_count** as arguments. This function inserts HTML mark tags to highlight occurrences of the pattern in the text.
- The function frees the memory allocated for the **text** array using the **free** function.
- The function returns 0, indicating successful execution of the program.

In summary, the main function prompts the user to enter the HTML file name and pattern, reads the HTML file, performs the brute-force algorithm for pattern matching, Horspool's algorithm, and Boyer-Moore algorithm. It measures and prints the execution times and other information for each algorithm. It also generates an HTML output file with highlighted occurrences of the pattern and outputs the results along with highlighted occurrences of the pattern in an HTML file named "output.html".

Section (7): Performance Analysis (Brute force vs. Horspool's vs. Boyer – Moore): -

Input (1): -



HTML FILE: Bit String (1)	Pattern	Brute Force Algorithm			Horspool's Algorithm			Boyer - Moore Algorithm		
		Comparisons	Running Time	Occurrences	Comparisons	Running Time	Occurrences	Comparisons	Running Time	Occurrences
4 - bit patterns	1010	3841260	15.6266	127631	2816639	20.4299	127631	2648915	20.3723	127631
	1100	3841516	16.1629	127702	2815350	18.9734	127702	2211442	18.4104	127702
	0101	3840713	15.4989	127684	2816788	19.6389	127684	2648667	19.9999	127684
	0011	3840574	15.5263	127649	2816395	17.4123	127649	2211779	17.9335	127649
8 - bit pattern	10101010	4078745	15.9141	7231	3054113	18.5445	7231	1264422	10.8987	7231
	11110000	4082255	16.1283	8129	1984398	13.053	8129	1236848	10.7788	8129
	01010101	4078247	16.0835	7229	3054337	19.6607	7229	1264488	10.977	7229
	00110011	4078740	16.1176	7540	3054663	17.2433	7540	1673606	13.5614	7540
	11001100	4079788	16.3065	7543	3053753	18.9607	7543	1673941	13.5669	7543
Duplicate patterns	1	2048550	12.6101	1024474	2048550	18.2879	1024474	2048550	18.1301	1024474
	11	3073022	14.3476	512356	2047903	15.4742	512356	2047903	16.0432	512356
	1111	3841587	15.4011	127811	1745725	11.0973	127811	1391112	10.6469	127811
	11111111	4080095	16.0959	7338	1120518	6.8896	7338	612909	5.2914	7338
	0	2048550	12.7604	1024076	2048550	17.1177	1024076	2048550	17.8938	1024076
	00	3072625	14.4592	511959	2047399	14.8702	511959	2047399	15.9851	511959
	0000	3840547	15.4879	127668	1744782	11.9618	127668	1390465	10.5009	127668
	00000000	4078729	16.0285	7300	1119674	6.1479	7300	612303	4.9018	7300

Here's a breakdown of the data:

4-bit patterns:

- Pattern: 1010
    - Brute Force Algorithm: Comparisons: 3841260, Running Time: 15.6266, Occurrences: 127631
    - Horspool's Algorithm: Comparisons: 2816639, Running Time: 20.4299, Occurrences: 127631
    - Boyer-Moore Algorithm: Comparisons: 2648915, Running Time: 20.3723, Occurrences: 127631
- Pattern: 1100
    - Brute Force Algorithm: Comparisons: 3841516, Running Time: 16.1629, Occurrences: 127702
    - Horspool's Algorithm: Comparisons: 2815350, Running Time: 18.9734, Occurrences: 127702
    - Boyer-Moore Algorithm: Comparisons: 2211442, Running Time: 18.4104, Occurrences: 127702
- Pattern: 0101

    - Brute Force Algorithm: Comparisons: 3840713, Running Time: 15.4989, Occurrences: 127684
    - Horspool's Algorithm: Comparisons: 2816788, Running Time: 19.6389, Occurrences: 127684
    - Boyer-Moore Algorithm: Comparisons: 2648667, Running Time: 19.9999, Occurrences: 127684
- Pattern: 0011
    - Brute Force Algorithm: Comparisons: 3840574, Running Time: 15.5263, Occurrences: 127649
    - Horspool's Algorithm: Comparisons: 2816395, Running Time: 17.4123, Occurrences: 127649
    - Boyer-Moore Algorithm: Comparisons: 2211779, Running Time: 17.9335, Occurrences: 127649

8-bit patterns:

- Pattern: 10101010
    - Brute Force Algorithm: Comparisons: 4078745, Running Time: 15.9141, Occurrences: 7231
    - Horspool's Algorithm: Comparisons: 3054113, Running Time: 18.5445, Occurrences: 7231
    - Boyer-Moore Algorithm: Comparisons: 1264422, Running Time: 10.8987, Occurrences: 7231
- Pattern: 11110000
    - Brute Force Algorithm: Comparisons: 4082255, Running Time: 16.1283, Occurrences: 8129
    - Horspool's Algorithm: Comparisons: 1984398, Running Time: 13.053, Occurrences: 8129
    - Boyer-Moore Algorithm: Comparisons: 1236848, Running Time: 10.7788, Occurrences: 8129
- Pattern: 01010101
    - Brute Force Algorithm: Comparisons: 4078247, Running Time: 16.0835, Occurrences: 7229
    - Horspool's Algorithm: Comparisons: 3054337, Running Time: 19.6607, Occurrences: 7229
    - Boyer-Moore Algorithm: Comparisons: 1264488, Running Time: 10.977, Occurrences: 7229
- Pattern: 00110011
    - Brute Force Algorithm: Comparisons: 4078740, Running Time: 16.1176, Occurrences: 7540
    - Horspool's Algorithm: Comparisons: 3054663, Running Time: 17.2433, Occurrences: 7540
    - Boyer-Moore Algorithm: Comparisons: 1673606, Running Time: 13.5614, Occurrences: 7540
- Pattern: 11001100
    - Brute Force Algorithm: Comparisons: 4079788, Running Time: 16.3065, Occurrences: 7543
    - Horspool's Algorithm: Comparisons: 3053753, Running Time: 18.9607, Occurrences: 7543
    - Boyer-Moore Algorithm: Comparisons: 1673941, Running Time: 13.5669, Occurrences: 7543

Duplicate patterns:

- Pattern: 1

    - Brute Force Algorithm: Comparisons: 2048550, Running Time: 12.6101, Occurrences: 1024474
    - Horspool's Algorithm: Comparisons: 2048550, Running Time: 18.2879, Occurrences: 1024474
    - Boyer-Moore Algorithm: Comparisons: 2048550, Running Time: 18.1301, Occurrences: 1024474
- Pattern: 11
    - Brute Force Algorithm: Comparisons: 3073022, Running Time: 14.3476, Occurrences: 512356
    - Horspool's Algorithm: Comparisons: 2047903, Running Time: 15.4742, Occurrences: 512356
    - Boyer-Moore Algorithm: Comparisons: 2047903, Running Time: 16.0432, Occurrences: 512356
- Pattern: 1111
    - Brute Force Algorithm: Comparisons: 3841587, Running Time: 15.4011, Occurrences: 127811
    - Horspool's Algorithm: Comparisons: 1745725, Running Time: 11.0973, Occurrences: 127811
    - Boyer-Moore Algorithm: Comparisons: 1391112, Running Time: 10.6469, Occurrences: 127811
- Pattern: 11111111
    - Brute Force Algorithm: Comparisons: 4080095, Running Time: 16.0959, Occurrences: 7338
    - Horspool's Algorithm: Comparisons: 1120518, Running Time: 6.8896, Occurrences: 7338
    - Boyer-Moore Algorithm: Comparisons: 612909, Running Time: 5.2914, Occurrences: 7338
- Pattern: 0
    - Brute Force Algorithm: Comparisons: 2048550, Running Time: 12.7604, Occurrences: 1024076
    - Horspool's Algorithm: Comparisons: 2048550, Running Time: 17.1177, Occurrences: 1024076
    - Boyer-Moore Algorithm: Comparisons: 2048550, Running Time: 17.8938, Occurrences: 1024076
- Pattern: 00
    - Brute Force Algorithm: Comparisons: 3072625, Running Time: 14.4592, Occurrences: 511959
    - Horspool's Algorithm: Comparisons: 2047399, Running Time: 14.8702, Occurrences: 511959
    - Boyer-Moore Algorithm: Comparisons: 2047399, Running Time: 15.9851, Occurrences: 511959
- Pattern: 0000
    - Brute Force Algorithm: Comparisons: 3840547, Running Time: 15.4879, Occurrences: 127668
    - Horspool's Algorithm: Comparisons: 1744782, Running Time: 11.9618, Occurrences: 127668
    - Boyer-Moore Algorithm: Comparisons: 1390465, Running Time: 10.5009, Occurrences: 127668
- Pattern: 00000000
    - Brute Force Algorithm: Comparisons: 4078729, Running Time: 16.0285, Occurrences: 7300
    - Horspool's Algorithm: Comparisons: 1119674, Running Time: 6.1479, Occurrences: 7300

- Boyer-Moore Algorithm: Comparisons: 612303, Running Time: 4.9018, Occurrences: 7300

Please note that the provided data is in tabular format and the information is extracted and presented in a more readable manner. we can observe the following behaviors:

- For most patterns, the Boyer-Moore algorithm consistently requires the fewest comparisons, followed by Horspool's algorithm and then the Brute Force algorithm.
- The number of comparisons generally increases with longer patterns, as more characters need to be checked for a match.
- There are some patterns where Horspool's algorithm performs better than Boyer-Moore in terms of comparisons, such as the pattern "1111" and "11111111".
- The number of occurrences of a pattern does not necessarily correspond to the number of comparisons made by the algorithms.

It's important to note that the running time and occurrences are not represented in the graph, as the focus is specifically on the number of comparisons made by each algorithm for different patterns.

Input (2), (3): -

As the Horspool's algorithm is more efficient in smaller patterns and less complex texts to search them on, it performed better on smaller patterns as expected. When it comes to longer than two bits of patterns though, the Boyer-Moore algorithm performed the best as it is more efficient on longer patterns.

Now, here below is the table for the performance analysis for the rest of the 5 inputs, with the corresponding patterns:

HTML FILE	PATTERN	EXECUTION TIME (ms) - Number of Occurences		
		Brute Force	**Horsepool's**	**Boyer Moore**
bitInput2.html	"10101010"	"26.36 - 7592"	"27.26 - 7592"	"13.46 - 7592"
	"11110000"	"26.07 - 7413"	"18.76 - 7413"	"13.97 - 7413"
	"01010101"	"26.71 - 7484"	"28.04 - 7484"	"14.12 - 7484"
	"00110011"	"27.13 - 7314"	"27.07 - 7314"	"18.06 - 7314"
	"11001100"	"26.34 - 7423"	"27.85 - 7423"	"17.19 - 7423"
	"11111111"	"25.86 - 3774"	"8.45 - 3774"	"6.49 - 3774"
	"00000000"	"26.95 - 11109"	"10.29 - 11109"	"7.61 - 11109"
bitInput3.html	"10101010"	"27.27 - 6871"	"29.39 - 6871"	"14.79 - 6871"
	"11110000"	"28.79 - 6918"	"15.56 - 6918"	"12.38 - 6918"
	"01010101"	"25.78 - 6820"	"26.67 - 6820"	"13.07 - 6820"
	"00110011"	"25.69 - 6892"	"28.46 - 6892"	"16.29 - 6892"
	"11001100"	"28.05 - 6934"	"26.39 - 6934"	"18.65 - 6934"
	"11111111"	"28.90 - 32978"	"12.60 - 32978"	"9.17 - 32978"
	"00000000"	"24.88 - 1466"	"7.41 - 1466"	"5.91 - 1466"
input1.html	cat	"9.27 - 463"	"5.89 - 463"	"7.45 - 463"
	duck	"9.35 - 1"	"3.59 - 1"	"4.61 - 1"
input2.html	cat	"7.87 - 421"	"5.06 - 421"	"6.44 - 421"
	duck	"7.96 - 2"	"3.00 - 2"	"3.89 - 2"
input3.html	cat	"6.83 - 233"	"4.33 - 233"	"5.51 - 233"
	duck	"7.27 - 9"	"2.07 - 9"	"3.49 - 9"

As can be seen from the table above, the Boyer-Moore algorithm has proved itself as the fastest (or in other words, the most efficient) in all of the bit-string HTML input files' scenarios. After that, the Horspool's algorithm came as the second fastest, and after it came the Brute Force.

The same can't be said for the "English word" HTML input files, in which Horspool's proved itself as the fastest, after which Boyer-Moore came as the second fastest.

These results were all expected, as the Boyer-Moore algorithm is more efficient in more simplistic text inputs like the bit-string ones. Comparably, the Horspool's algorithm is more efficient in longer and more complex text inputs like the "English word" ones. The Brute Force, as it's literally in its name, is a brute-force method of searching so it had always been expected to be the last when it comes to efficiency.

Section (8): Test Cases: -

HTML Type (1): HTML files with English text.

Input (1):

```
Html file name (example.html): input1.html
Pattern: cat
***-------------------------------------------------------------------------------------------***
Brute-force Algorithm:
Occurrences: 463
Comparisons: 2084332
Running time: 9.2370 ms
***-------------------------------------------------------------------------------------------***
***-------------------------------------------------------------------------------------------***
Horspool Algorithm:
Shift Table:
a: 1
c: 2
Occurrences: 463
Comparisons: 770031
Running time: 5.8960 ms
***-------------------------------------------------------------------------------------------***
***-------------------------------------------------------------------------------------------***
Boyer-Moore Algorithm:
Bad Symbol Table:
a: 1
c: 2
Good Suffix Table:
1: 2
2: 2
3: 3
Occurrences: 463
Comparisons: 792182
Running time: 7.4570 ms
***-------------------------------------------------------------------------------------------***
```

Bad Symbol Table

*	a	c
3	1	2

Good Symbol Table

k	Pattern	Shift
1	cat	2
2	cat	2
3	cat	3

Output

eyond $14.95 OGR Publisher Rich Melvin is your host in this second look at the Lionel Trainmaster Command System- . Trainmaster has started a revolution in the O gauge hobby , and there are several other manufacturers building TMCC-compatible products . You 'll see new products from IC Controls , Train America Studios and the new Lionel ZW Transformer , plus a couple of other special features . Rich will also show you a web site where you can " test drive " Trainmaster on-line to see how it works . Its all here in this fast-paced look at the Lionel Trainmasterand Beyond ! Millenials and Model Trains Found an interesting article regarding millenials and how they are increasing activity in trade type jobs . So maybe the narrative that they do n't like to use their hands and therefore would have no interest in model trains is not so accurate . I see this all around me , and believe that many will very much be interested in and carry on this hobby and others like it . Like anything I think it 's better to try and understand this generation and what their likes and desires are than to write them off . I believe this hobby has a lot of potential with this younger generation . I agree on trying to understand this generation since our two daughters and two sons-in-law are millennials . The four of them average 38 years younger than me , so there could be quite a ' generation gap ' , the term used when I was their age . Actually the four of them are all doers and I learn a lot form them . Now if I could get only a passing interest to keep me happy ! Would suggest Lionel already has a strategy to tap Millennial interest with LionChief and smartphone apps . Have a ton of free trains Millennial son declined in favor of spending his own money on the latest and greatest offerings . That is very interesting and a faint flicker of hope , for the " classic " tactile hobbies . But for model railroading specifically , it is only one piece of a bigger puzzle . Some train buddies and I had a nice conversation about the fate of the hobby this past Sunday morning . One of the elements that I like to cite is the state of real railroading today , and how times have changed in our lifetimes , and more specifically since 9/11 . Most of us around the table had stories about being approached by railroad security . With all of the other time and money sucking activities of the modern world , it is very hard to plant that seed and have it grow into the " trainsanity " that many of a bit going forward , we are still looking at a lost generation , maybe two . Not trying to be a pessimist here , just a realist . On a related note , one that has also come up repeatedly here on the forum , there was talk of the owner of the largest hobby shop in the area possibly wanting to retire . Just another sign of the times . A great deal of bandwidth has been expended to considering the young people in the hobby . The real question that must be asked (but I doubt can be answered) is this alone : What 's the percentages of the population that is into trains (both real and model) in any given generation , and has that percentage risen or fallen over time ? If you can answer that , then alone will you be able to answer the overall question . I think it 's silly to think anyone between kid and middle age is going to be into

Html file name (example.html): input1.html
Pattern: duck

Brute-force Algorithm:
Occurrences: 1
Comparisons: 2089305
Running time: 9.3530 ms

Horspool Algorithm:
Shift Table:
c: 1
d: 3
u: 2

Occurrences: 1
Comparisons: 533718
Running time: 3.5590 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
c: 1
d: 3
u: 2

Good Suffix Table:
1: 3
2: 3

3: 3
4: 4

Occurrences: 1
Comparisons: 534946
Running time: 4.6110 ms

Bad Symbol Table

*	c	d	u
4	1	3	2

Good Symbol Table

k	Pattern	Shift
1	duck	3
2	duck	3
3	duck	3
4	duck	4

Output

eyond $14.95 OGR Publisher Rich Melvin is your host in this second look at the Lionel Trainmaster Command System- . Trainmaster has started a revolution in the O gauge hobby , and there are several other manufacturers building TMCC-compatible products . You 'll see new products from IC Controls , Train America Studios and the new Lionel ZW Transformer , plus a couple of other special features . Rich will also show you a web site where you can " test drive " Trainmaster on-line to see how it works . Its all here in this fast-paced look at the Lionel Trainmasterand Beyond ! Millenials and Model Trains Found an interesting article regarding millenials and how they are increasing activity in trade type jobs . So maybe the narrative that they do n't like to use their hands and therefore would have no interest in model trains is not so accurate . I see this all around me , and believe that many will very much be interested in and carry on this hobby and others like it . Like anything I think it 's better to try and understand this generation and what their likes and desires are than to write them off . I believe this hobby has a lot of potential with this younger generation . I agree on trying to understand this generation since our two daughters and two sons-in-law are millennials . The four of them average 38 years younger than me , so there could be quite a ' generation gap ' , the term used when I was their age . Actually the four of them are all doers and I learn a lot form them . Now if I could get only a passing interest to keep me happy ! Would suggest Lionel already has a strategy to tap Millennial interest with LionChief and smartphone apps . Have a ton of free trains Millennial son declined in favor of spending his own money on the latest and greatest offerings . That is very interesting and a faint flicker of hope , for the " classic " tactile hobbies . But for model railroading specifically , it is only one piece of a bigger puzzle . Some train buddies and I had a nice conversation about the fate of the hobby this past Sunday morning . One of the elements that I like to cite is the state of real railroading today , and how times have changed in our lifetimes , and more specifically since 9/11 . Most of us around the table had stories about being approached by railroad security . With all of the other time and money sucking activities of the modern world , it is very hard to plant that seed and have it grow into the " trainsanity " that many of a bit going forward , we are still looking at a lost generation , maybe two . Not trying to be a pessimist here , just a realist . On a related note , one that has also come up repeatedly here on the forum , there was talk of the owner of the largest hobby shop in the area possibly wanting to retire . Just another sign of the times . A great deal of bandwidth has been expended to considering the young people in the hobby . The real question that must be asked (but I doubt can be answered) is this along : What 's the percentages of the population that is into trains (both real and model) in any given generation , and has that percentage risen or fallen

Input (2):

Html file name (example.html): input2.html
Pattern: cat

Brute-force Algorithm:
Occurrences: 421
Comparisons: 1787879
Running time: 7.8710 ms

Horspool Algorithm:
Shift Table:
a: 1
c: 2

Occurrences: 421
Comparisons: 664167
Running time: 5.0620 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
a: 1
c: 2

Good Suffix Table:
1: 2
2: 2
3: 3

Occurrences: 421
Comparisons: 684335
Running time: 6.4430 ms

Bad Symbol Table

*	a	c
3	1	2

Good Symbol Table

k	Pattern	Shift
1	cat	2
2	cat	2
3	cat	3

Output

Insights on the Gold Market Tue 6 Jun , 2017 David Erfle Is This Breakout In Gold Stocks For Real A very nice move in gold and the underlying stocks . The Junior Miner Junky , David Erfle , takes a look at the breakout that we are seeing in the sector . David shares a stock that he is using as a barometer for the juniors as well as outline all the news events that have the potential to drive markets over the next to weeks . These news events include Tony just the long-awaited capital raise from RyePatch that Inkakola has been droning on and on about for months . Now we can focus on how they do ramping up into full production during the balance of the year . Not a huge dilution for expansion purposes . I would have been concerned if the dilution wiped out current shareholders or was for sustaining current state of operation . Almost $20 million with this bought deal , sale of 3000ounces of gold and the royalty . Hopefully management is smart with expects nothing less . On June 6 , 2017 at 7:57 pm , Excelsiorsays : Agreed Imtiaz Rahmat . As mentioned on the weekend show , the NSR sale for $5 million wasnt going to be enough , but most have discounted the actual Gold & Silver sales that RPM will making as they continue to ramp up into commercial production (and those will be substantial and growing) . Bill and the team have been very good stewards of company funds for the last few years , and have repeatedly stated in interviews and releases that they have worked hard not to be overly dilutive . I see no reason to expect any change in their savvy history of making solid decisions to advance the company . " Novo Resources Corp . $NVO $NSRPF is pleased to announce it has appointed Robert Humphryson as the Company 's Chief Executive Officer . Dr. Quinton Hennigh will become Novo 's first Chairman and retain his position as President . " Richard Postma - The Doctor Is In Mon 1 May , 2017 A Silver Target and The Precious Metals Over Summer Doc is with us today to update his lower silver targets and a look at the gold charts . There are some moving averages that have turned against the metals and will take time to recover . Summer is typically a rough time for the metals but we could see a the turn by the end of the year . Pardu , I believe the VIX is pretty much at a bottomI think we could " muddle " around here for a couple more days and then I would n't be surprised if we attempt a move higher . On May 2 , 2017 at 1:13 am , Excelsiorsays : Doc were you microwaving food during your interview ? = On May 2 , 2017 at 1:23 am , Excelsiorsays : BTW Were in agreement that it is nice building ones own portfolio of Smaller Producers , Developers , and Explorers and rebalancing as one sees fit . I 'm looking forward on Summer Clearance Sale . Ever Upward ! On May 1 , 2017 at 11:41 am , OOTB Jerrysays : Thanks for sharing DOC On May 1 , 2017 at 12:05 pm , Matthewsays : It looks to me like the miners have probably put in a tradeable low . For those who can view it , here 's the 2 hour chart for SIL : Very sound thoughts today Cory And Doc . Looks like the safe havens and conventional markets are about to get routed . Trump says he s going to break up the big banks and bring back a 21st century

Html file name (example.html): input2.html
Pattern: duck

Brute-force Algorithm:
Occurrences: 2
Comparisons: 1793555
Running time: 7.9680 ms

Horspool Algorithm:
Shift Table:
c: 1

d: 3
u: 2

Occurrences: 2
Comparisons: 456620
Running time: 3.0020 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
c: 1
d: 3
u: 2

Good Suffix Table:
1: 3
2: 3
3: 3
4: 4

Occurrences: 2
Comparisons: 457545
Running time: 3.8950 ms

Bad Symbol Table

*	c	d	u
4	1	3	2

Good Symbol Table

k	Pattern	Shift
1	duck	3
2	duck	3
3	duck	3
4	duck	4

Output

Insights on the Gold Market Tue 6 Jun , 2017 David Erfle Is This Breakout In Gold Stocks For Real A very nice move in gold and the underlying stocks . The Junior Miner Junky , David Erfle , takes a look at the breakout that we are seeing in the sector . David shares a stock that he is using as a barometer for the juniors as well as outline all the news events that have the potential to drive markets over the next to weeks . These news events include Tony just the long-awaited capital raise from RyePatch that Inkakola has been droning on and on about for months . Now we can focus on how they do ramping up into full production during the balance of the year . Not a huge dilution for expansion purposes . I would have been concerned if the dilution wiped out current shareholders or was for sustaining current state of operation . Almost $20 million with this bought deal , sale of 3000ounces of gold and the royalty . Hopefully management is smart with expects nothing less . On June 6 , 2017 at 7:57 pm , Excelsiorsays : Agreed Imtiaz Rahmat . As mentioned on the weekend show , the NSR sale for $5 million wasnt going to be enough , but most have discounted the actual Gold & Silver sales that RPM will making as they continue to ramp up into commercial production (and those will be substantial and growing) . Bill and the team have been very good stewards of company funds for the last few years , and have repeatedly stated in interviews and releases that they have worked hard not to be overly dilutive . I see no reason to expect any change in their savvy history of making solid decisions to advance the company . " Novo Resources Corp . $NVO $NSRPF is pleased to announce it has appointed Robert Humphryson as the Company 's Chief Executive Officer . Dr. Quinton Hennigh will become Novo 's first Chairman and retain his position as President . " Richard Postma - The Doctor Is In Mon 1 May , 2017 A Silver Target and The Precious Metals Over Summer Doc is with us today to update his lower silver targets and a look at the gold charts . There are some moving averages that have turned against the metals and will take time to recover . Summer is typically a rough time for the metals but we could see a the turn by the end of the year . Pardu , I believe the VIX is pretty much at a bottomI think we could " muddle " around here for a couple more days and then I would n't be surprised if we attempt a move higher . On May 2 , 2017 at 1:13 am , Excelsiorsays : Doc were you microwaving food during your interview ? = On May 2 , 2017 at 1:23 am , Excelsiorsays : BTW Were in agreement that it is nice building ones own portfolio of Smaller Producers , Developers , and Explorers and rebalancing as one sees fit . I 'm looking forward on Summer Clearance Sale . Ever Upward ! On May 1 , 2017 at 11:41 am , OOTB Jerrysays : Thanks for sharing DOC On May 1 , 2017 at 12:05 pm , Matthewsays : It looks to me like the miners have probably put in a tradeable low . For those who can view it , here 's the 2 hour chart for SIL . Very sound thoughts today

Input (3):

Html file name (example.html): input3.html

Pattern: cat

Brute-force Algorithm:
Occurrences: 233
Comparisons: 1573390
Running time: 6.8380 ms

Horspool Algorithm:
Shift Table:
a: 1
c: 2

Occurrences: 233
Comparisons: 581681
Running time: 4.3350 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
a: 1
c: 2

Good Suffix Table:
1: 2
2: 2
3: 3

Occurrences: 233
Comparisons: 595849
Running time: 5.5190 ms

Bad Symbol Table

*	a	c
3	1	2

Good Symbol Table

k	Pattern	Shift
1	cat	2
2	cat	2
3	cat	3

Output

surprised by the instruction that she did as he said . Then she was really glad when he pulled her close , wrapped his arms around her and kissed her . The warm , insistent pressure against her lips made her cling to him . Heat flared , melting away all her concerns about him being upset or angry or anything the least bit negative . His mouth moved slowly , as if giving her time to get used to what he was doing . If shed been able to communicate without her mouth , she would have told him that she did n't mind the kissing at all . In fact , she really , really liked it . He could do more . Really . Instead she placed her hands on his shoulders and let her body lean into his . She tilted her head and parted her lips in invitation . He reacted with a sharp intake of air and a gentle sweep of his tongue against hers . The kiss was as spectacular her insides quiver , her knees shake and her thighs tremble . Heat poured through her , making her want and need with an intensity that stunned her . One of his hands tangled in her hair , the other traced a line down her back until it settled on her rear . When he squeezed the curves , she arched forward , bringing her belly in contact with him . Now it was her turn to gasp as she felt his arousal . He wanted her . After everything that had happened and what her brother had said , Jefri wanted her . Delight blended with desire and she nearly laughed . He broke the kiss . " What do you find so amusing ? " he asked , his breath mingling with hers . " Just all of this . " " That I want to kiss you ? " " It is a bit of a surprise . " He cupped her chin and stared into her eyes . " Why ? You 're a beautiful woman . Unique man alive who would n't sell his soul for one night with you . " She blinked at him . Whoatalk about a great line . Right now she did n't even care if he meant it . " I , um , thanks . " " Youre welcome . " He smiled as he rubbed his thumb against her lower lip . " I would very much like to have dinner with you tonight . " At this point in time , she would have followed him to the moon . " Okay . I mean , that would be great . " " Good . I 'll pick you up at your room at seven . Is that convenient ? " As shed planned to spend the rest of the afternoon soaking and grooming , seven sounded good . " I 'll be ready . " " Well be going out , " he said . " There are several excellent restaurants in the city . Will you allow me to choose ? " " Of course . " " Then kiss on her lips , then stepped back . " Leave the dog at home . " Chapter 6 Billie stood in front of her closet and reviewed her array of clothes . She believed in shopping , both as a sport and a way to relax , so there were plenty of outfits to choose from . She already knew she wanted something both sexy and sophisticated , with a touch of elegance thrown in . Which pretty much left out anything with feathers , rhinestones or too much beading . She tilted her head , then straightened quickly as the curlers started to loosen . Tonight required perfect hair , which meant an hour spent in electric curlers and a double dose of hair spray . " Everyone looks good in black , " she murmured as she pulled out a clinging black dress with a deep vee and see-through sleeves . " But its so predictable . " Maybe she should try a color . Not red . Somehow that just made her look trashy . " Maybe blue dress that had

Html file name (example.html): input3.html
Pattern: duck

Brute-force Algorithm:
Occurrences: 9
Comparisons: 1599948
Running time: 7.2720 ms

Horspool Algorithm:
Shift Table:
c: 1
d: 3
u: 2

Occurrences: 9
Comparisons: 403765
Running time: 2.7070 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
c: 1
d: 3
u: 2

Good Suffix Table:
1: 3

2: 3
3: 3
4: 4

Occurrences: 9
Comparisons: 404773
Running time: 3.4900 ms

Bad Symbol Table

*	c	d	u
4	1	3	2

Good Symbol Table

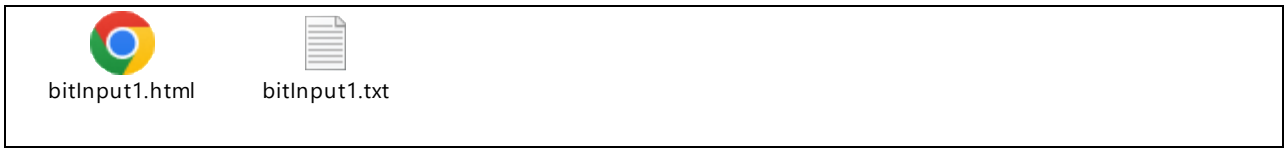
k	Pattern	Shift
1	duck	3
2	duck	3
3	duck	3
4	duck	4

Output

surprised by the instruction that she did as he said . Then she was really glad when he pulled her close , wrapped his arms around her and kissed her . The warm , insistent pressure against her lips made her cling to him . Heat flared , melting away all her concerns about him being upset or angry or anything the least bit negative . His mouth moved slowly , as if giving her time to get used to what he was doing . If shed been able to communicate without her mouth , she would have told him that she did n't mind the kissing at all . In fact , she really , really liked it . He could do more . Really . Instead she placed her hands on his shoulders and let her body lean into his . She tilted her head and parted her lips in invitation . He reacted with a sharp intake of air and a gentle sweep of his tongue against hers . The kiss was as spectacular her insides quiver , her knees shake and her thighs tremble . Heat poured through her , making her want and need with an intensity that stunned her . One of his hands tangled in her hair , the other traced a line down her back until it settled on her rear . When he squeezed the curves , she arched forward , bringing her belly in contact with him . Now it was her turn to gasp as she felt his arousal . He wanted her . After everything that had happened and what her brother had said , Jefri wanted her . Delight blended with desire and she nearly laughed . He broke the kiss . " What do you find so amusing ? " he asked , his breath mingling with hers . " Just all of this . " " That I want to kiss you ? " " It is a bit of a surprise . " He cupped her chin and stared into her eyes . " Why ? You 're a beautiful woman . Unique man alive who would n't sell his soul for one night with you . " She blinked at him . Whoatalk about a great line . Right now she did n't even care if he meant it . " I , um , thanks . " " Youre welcome . " He smiled as he rubbed his thumb against her lower lip . " I would very much like to have dinner with you tonight . " At this point in time , she would have followed him to the moon . " Okay . I mean , that would be great . " " Good . I 'll pick you up at your room at seven . Is that convenient ? " As shed planned to spend the rest of the afternoon soaking and grooming , seven sounded good . " I 'll be ready . " " Well be going out , " he said . " There are several excellent restaurants in the city . Will you allow me to choose ? " " Of course . " " Then kiss on her lips , then stepped back . " Leave the dog at home . " Chapter 6 Billie stood in front of her closet and reviewed her array of clothes . She believed in shopping , both as a sport and a way to relax , so there were plenty of outfits to choose from . She already knew she wanted something both sexy and sophisticated , with a touch of elegance thrown in . Which pretty much left out anything with feathers , rhinestones or too much beading . She tilted her head , then straightened quickly as the curlers started to loosen . Tonight required perfect hair , which meant an hour spent in electric curlers and a double dose of hair spray . " Everyone looks good in black . " she murmured as she pulled out a clinging black dress with a

HTML Type (2): HTML files that include random bit-strings.

Input (1): bitInput1.html



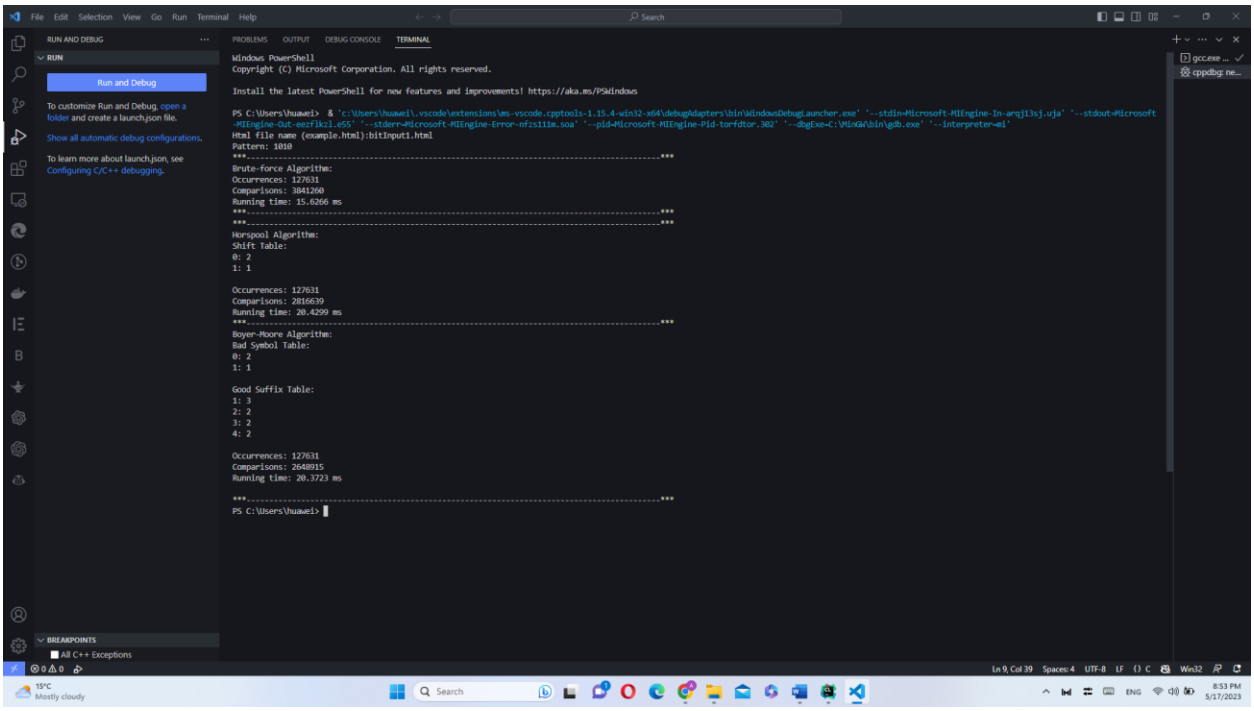
- Randomized pattern:
  - 1010

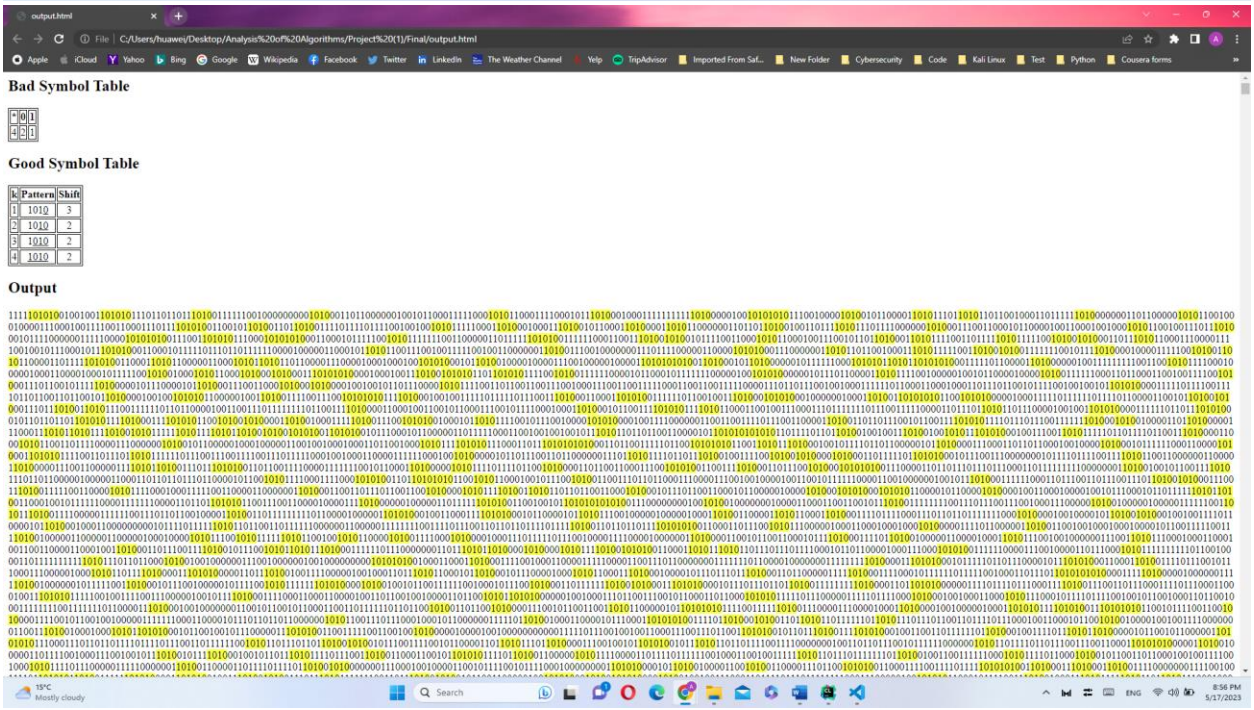
Html file name (example.html): bitInput1.html
Pattern: 1010

Brute-force Algorithm:
Occurrences: 127631
Comparisons: 3841260
Running time: 15.6266 ms

Horspool Algorithm:
Shift Table:
0: 2
1: 1
Occurrences: 127631
Comparisons: 2816639
Running time: 20.4299 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
0: 2
1: 1
Good Suffix Table:
1: 3
2: 2
3: 2
4: 2
Occurrences: 127631
Comparisons: 2648915
Running time: 20.3723 ms





- 1100

Html file name (example.html): bitInput1.html
Pattern: 1100

Brute-force Algorithm:
Occurrences: 127702
Comparisons: 3841516
Running time: 16.1629 ms

_____

_____

Horspool Algorithm:

Shift Table:

0: 1

1: 2

Occurrences: 127702

Comparisons: 2815350

Running time: 18.9734 ms

_____

_____

Boyer-Moore Algorithm:

Bad Symbol Table:

0: 1

1: 2

Good Suffix Table:

1: 1

2: 3

3: 3

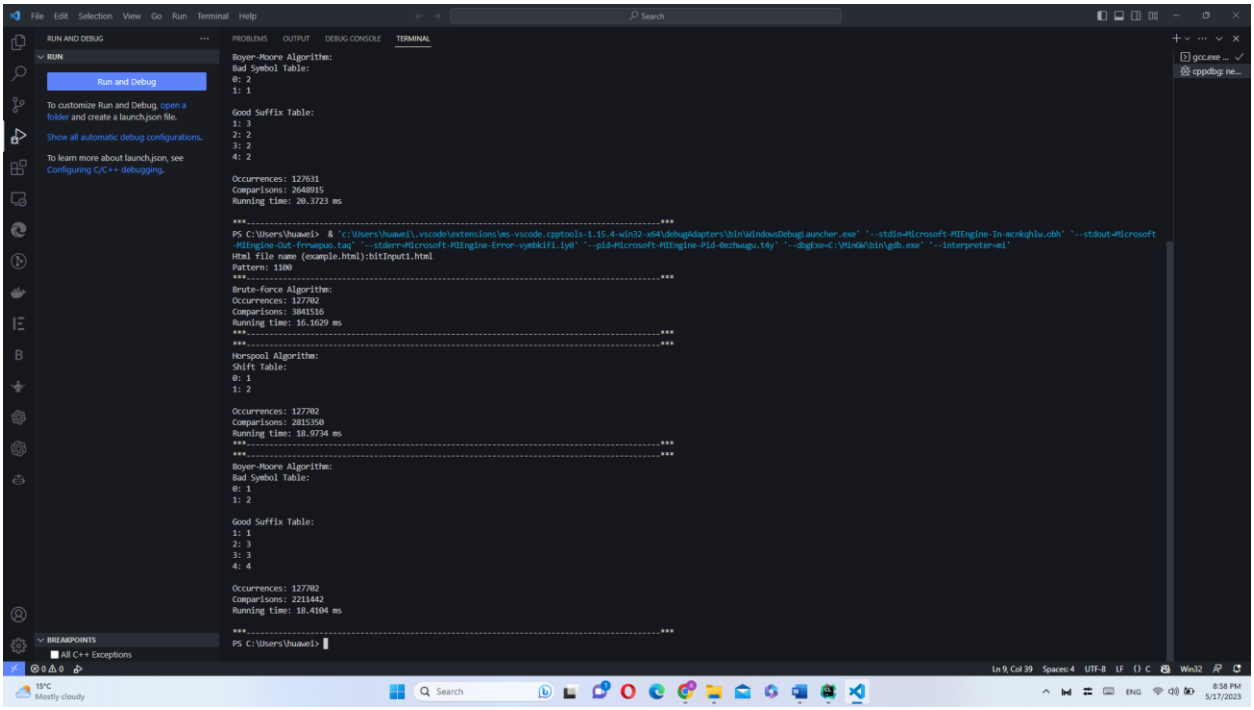
4: 4

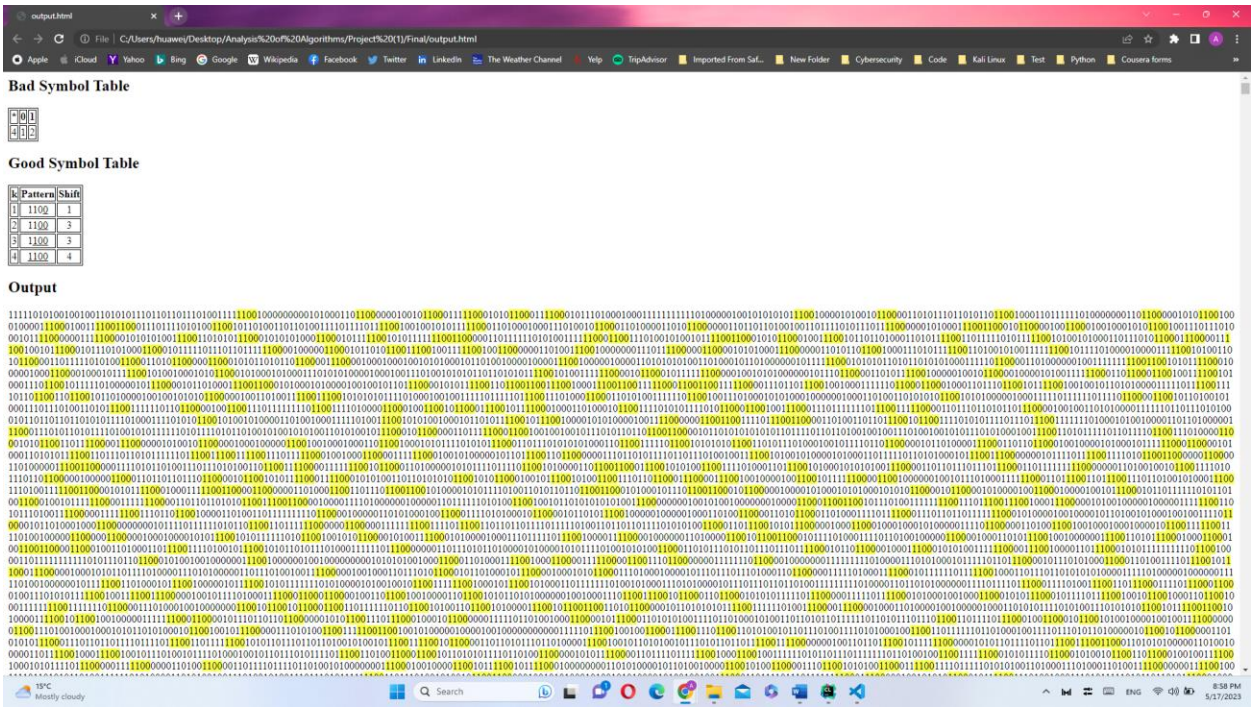
Occurrences: 127702

Comparisons: 2211442

Running time: 18.4104 ms

_____





- 0101

Html file name (example.html): bitInput1.html
Pattern: 0101

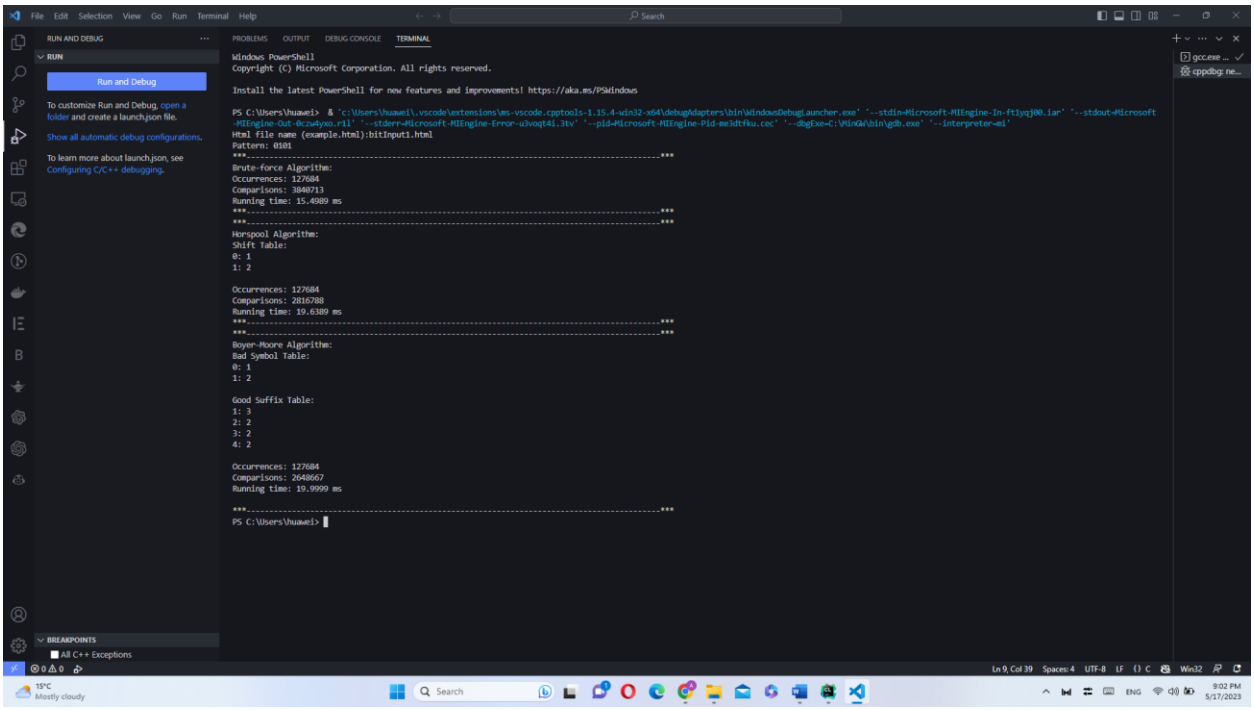
Brute-force Algorithm:
Occurrences: 127684
Comparisons: 3840713
Running time: 15.4989 ms

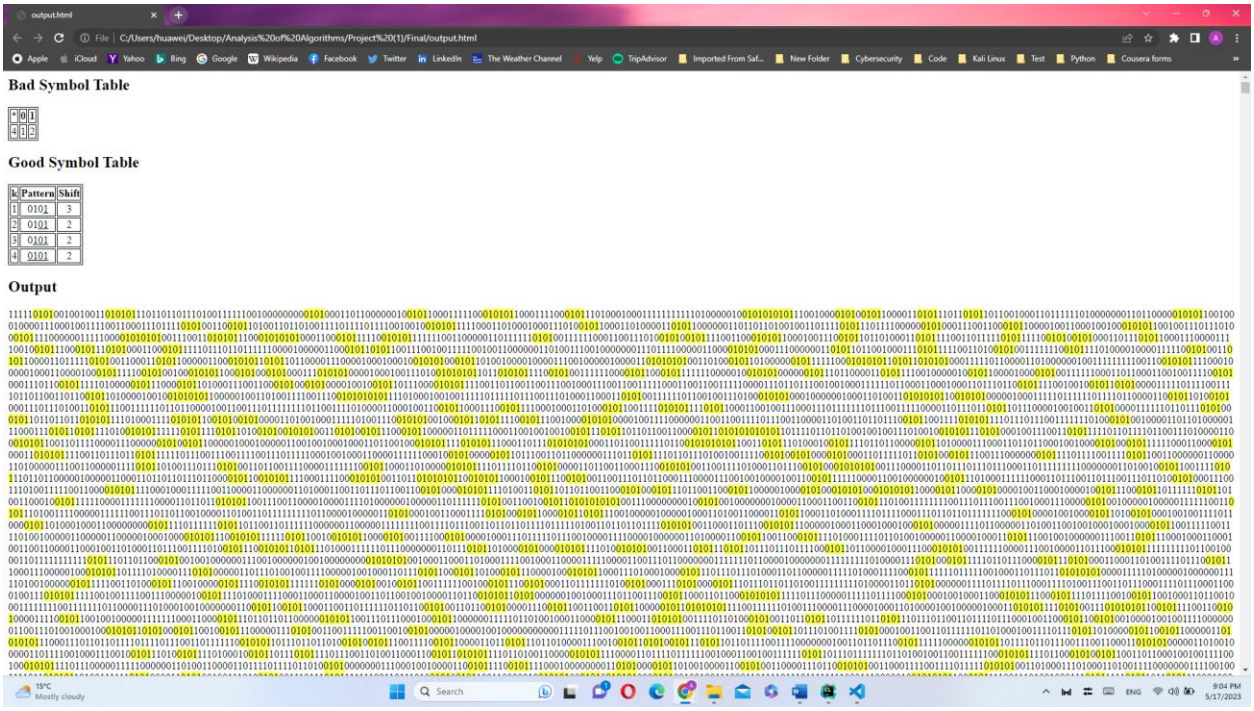
Horspool Algorithm:
Shift Table:
0: 1
1: 2
Occurrences: 127684
Comparisons: 2816788
Running time: 19.6389 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
0: 1
1: 2
Good Suffix Table:
1: 3
2: 2
3: 2
4: 2
Occurrences: 127684
Comparisons: 2648667

Running time: 19.9999 ms

_____





- 0011

Html file name (example.html): bitInput1.html

Pattern: 0011

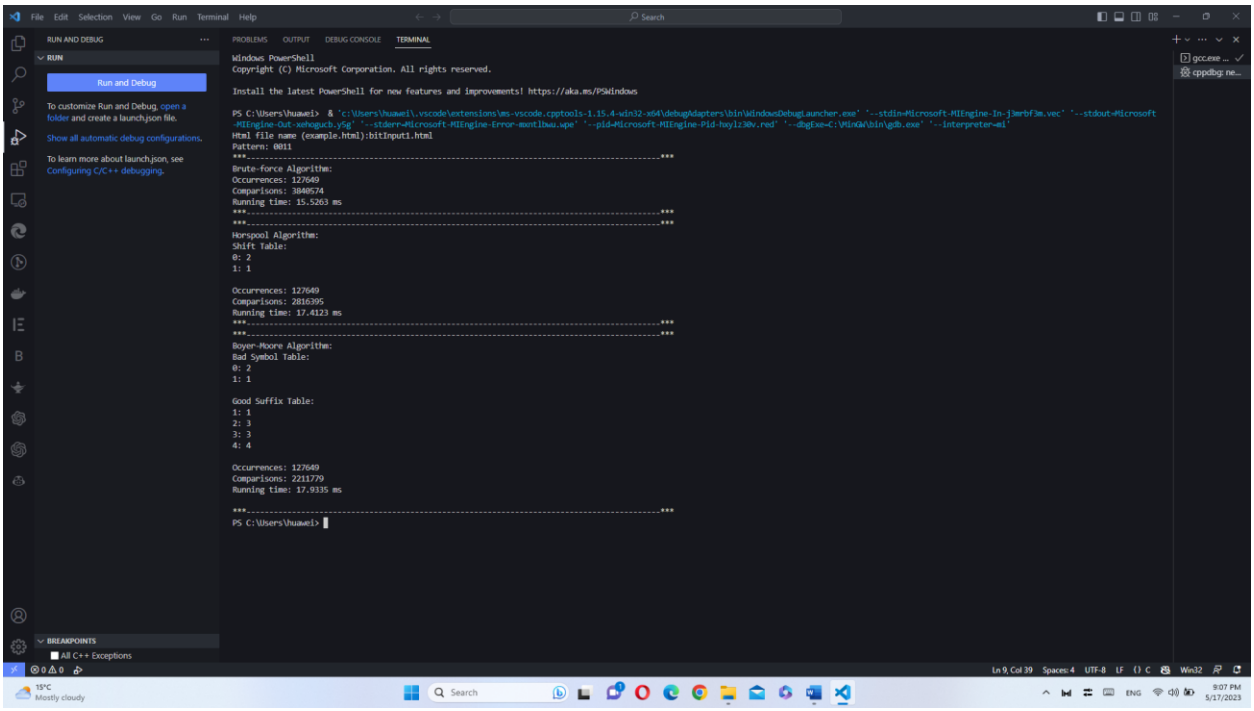
_____

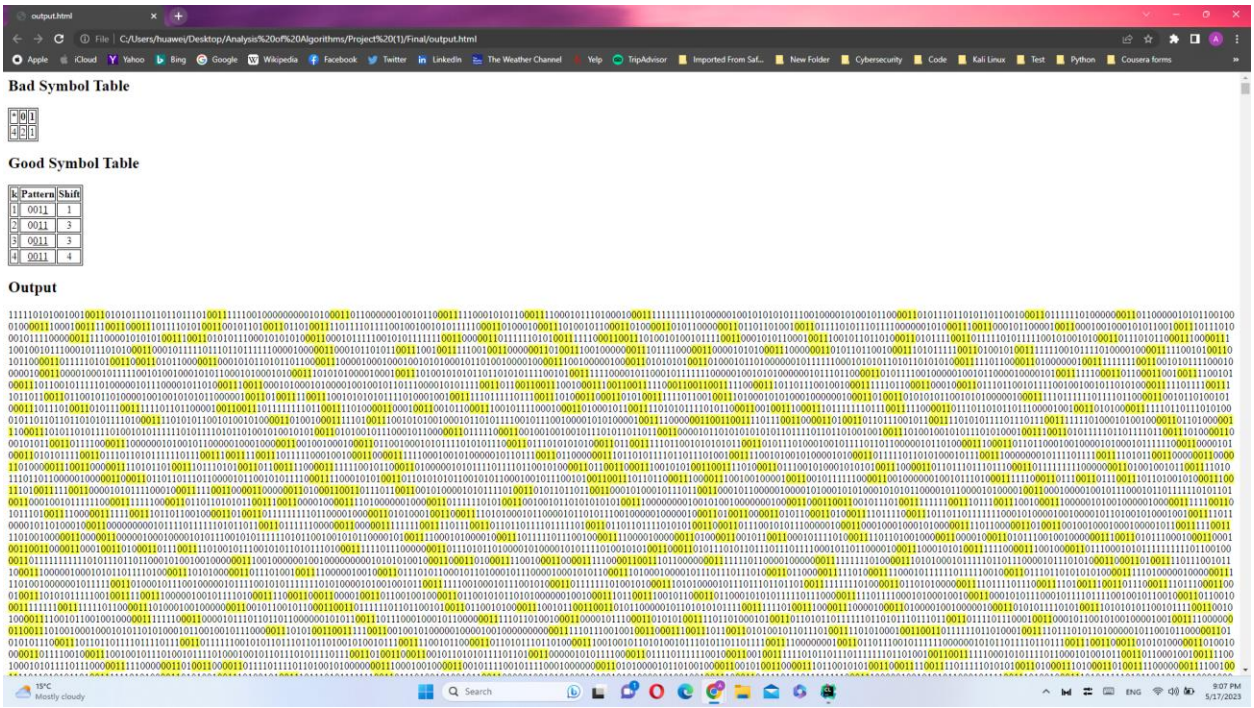
Brute-force Algorithm:

Occurrences: 127649
Comparisons: 3840574
Running time: 15.5263 ms

Horspool Algorithm:
Shift Table:
0: 2
1: 1
Occurrences: 127649
Comparisons: 2816395
Running time: 17.4123 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
0: 2
1: 1
Good Suffix Table:
1: 1
2: 3
3: 3
4: 4
Occurrences: 127649
Comparisons: 2211779
Running time: 17.9335 ms





- 10101010

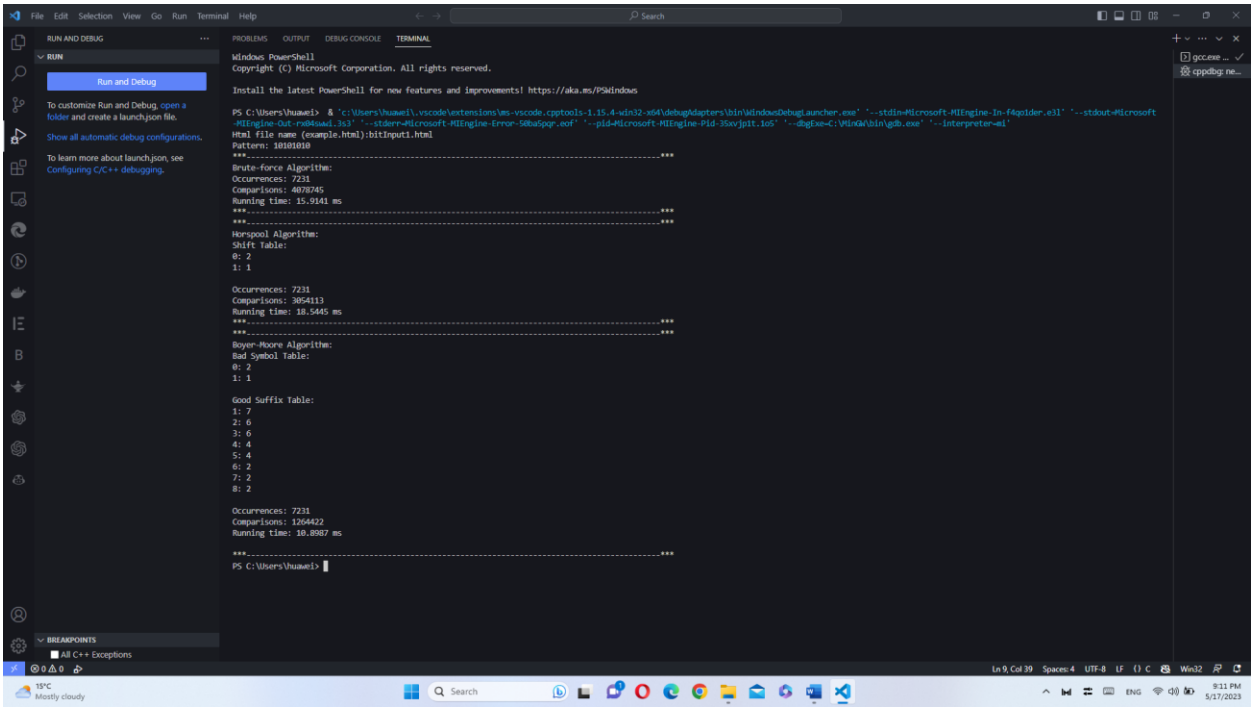
Html file name (example.html): bitInput1.html
Pattern: 10101010

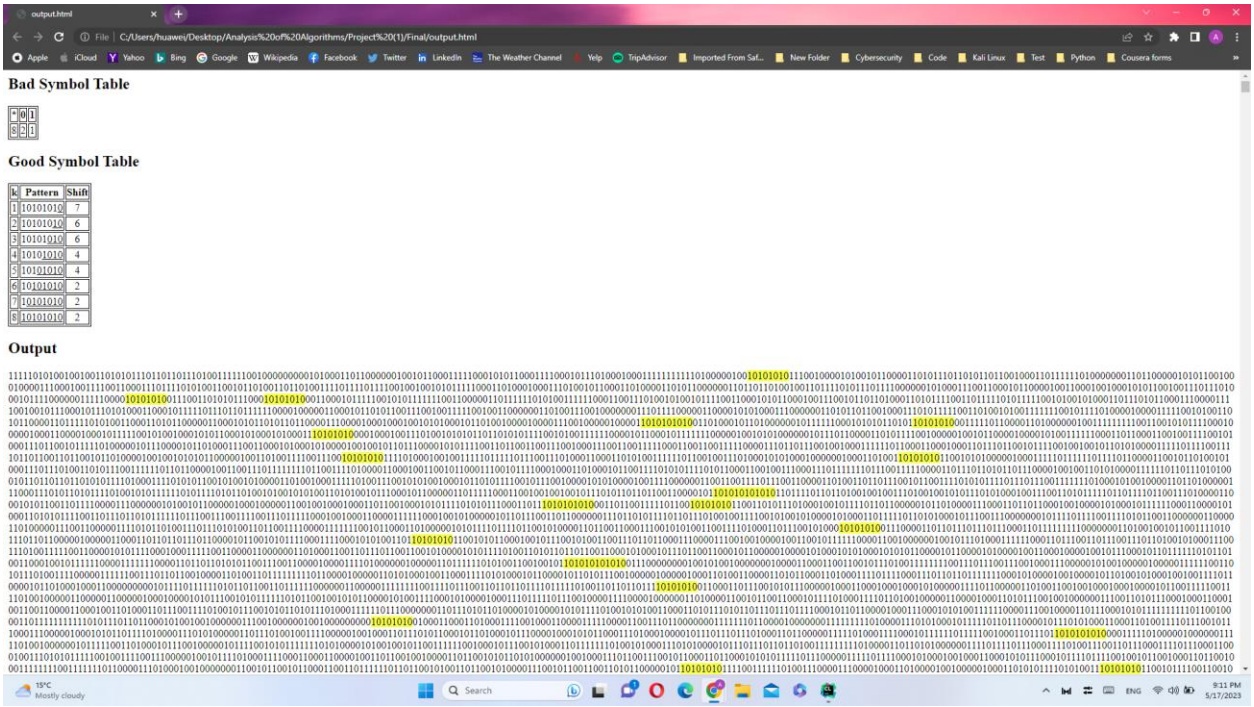
Brute-force Algorithm:
Occurrences: 7231
Comparisons: 4078745
Running time: 15.9141 ms

Horspool Algorithm:
Shift Table:
0: 2
1: 1
Occurrences: 7231
Comparisons: 3054113
Running time: 18.5445 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
0: 2
1: 1
Good Suffix Table:
1: 7
2: 6
3: 6
4: 4
5: 4
6: 2

7: 2
8: 2
Occurrences: 7231
Comparisons: 1264422
Running time: 10.8987 ms
_____





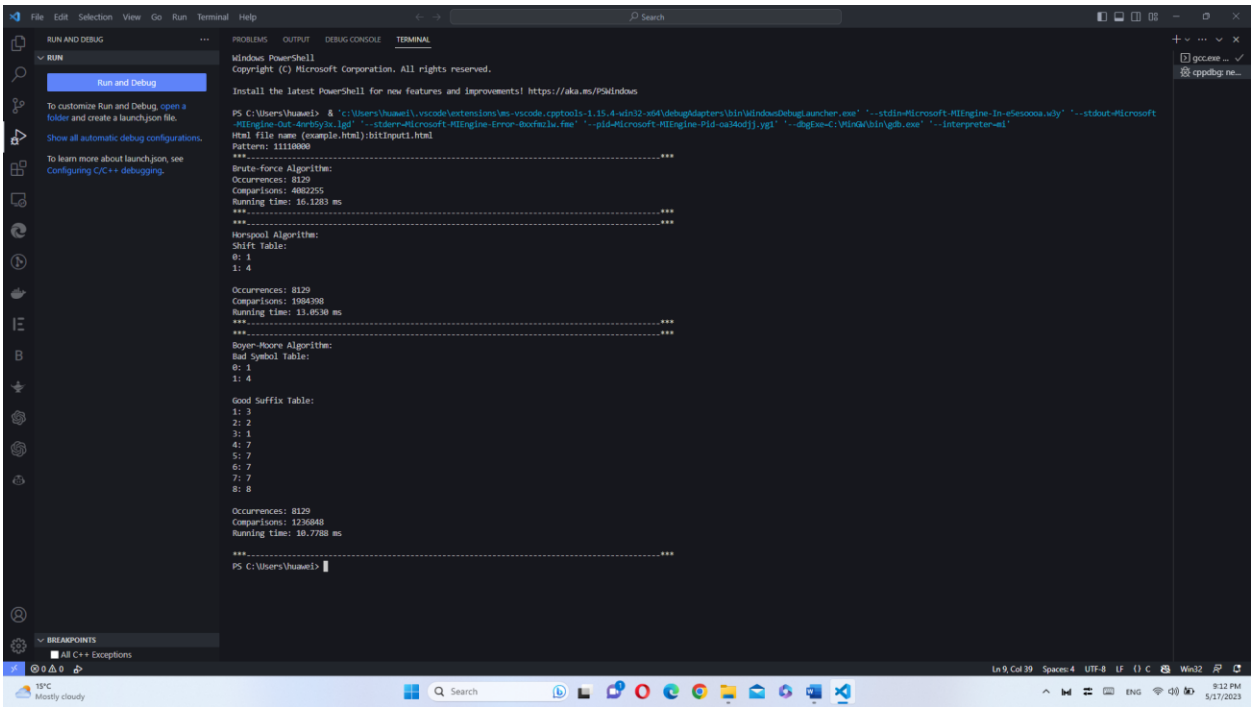
- 11110000

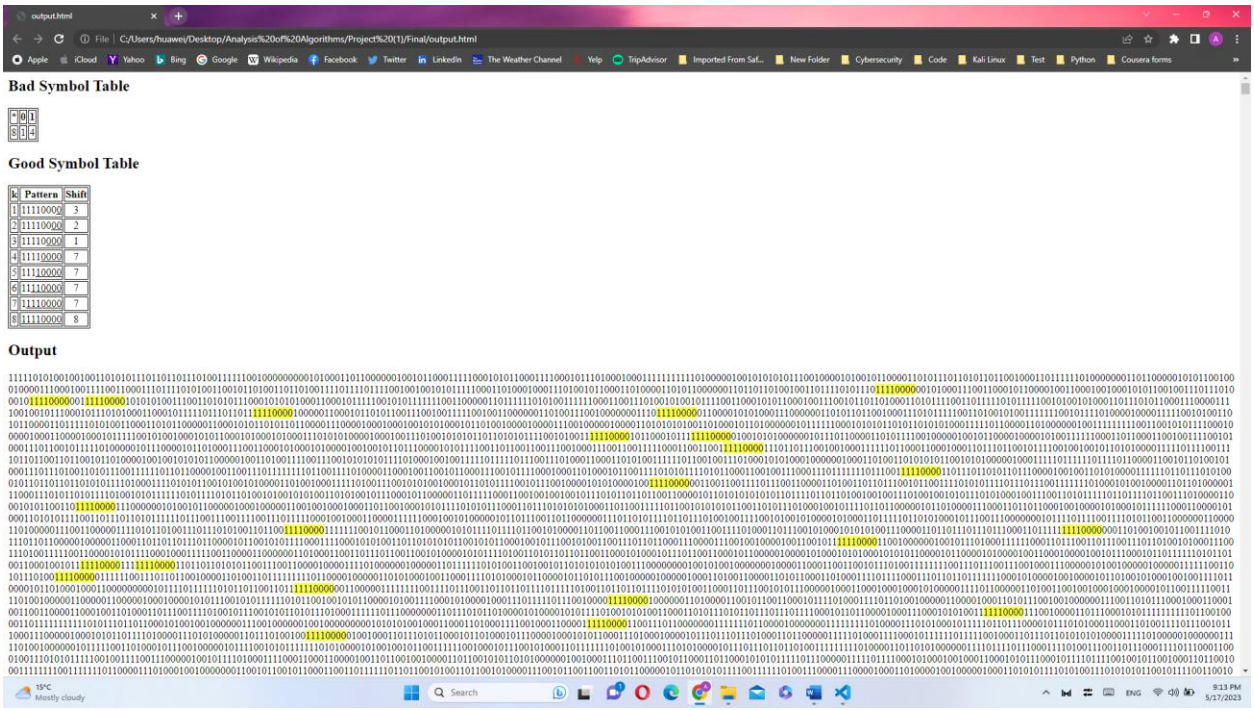
Html file name (example.html): bitInput1.html
Pattern: 11110000

Brute-force Algorithm:
Occurrences: 8129
Comparisons: 4082255
Running time: 16.1283 ms

Horspool Algorithm:
Shift Table:
0: 1
1: 4
Occurrences: 8129
Comparisons: 1984398
Running time: 13.0530 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
0: 1
1: 4
Good Suffix Table:
1: 3
2: 2
3: 1
4: 7
5: 7
6: 7
7: 7
8: 8
Occurrences: 8129
Comparisons: 1236848
Running time: 10.7788 ms



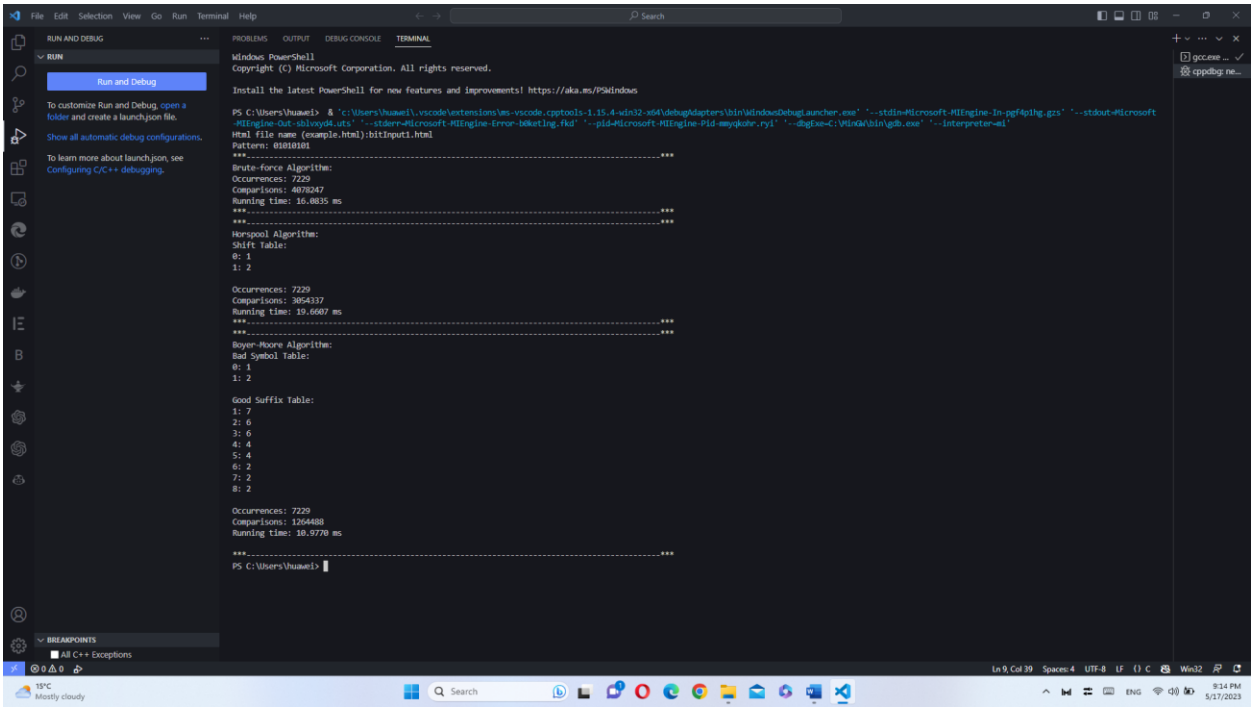


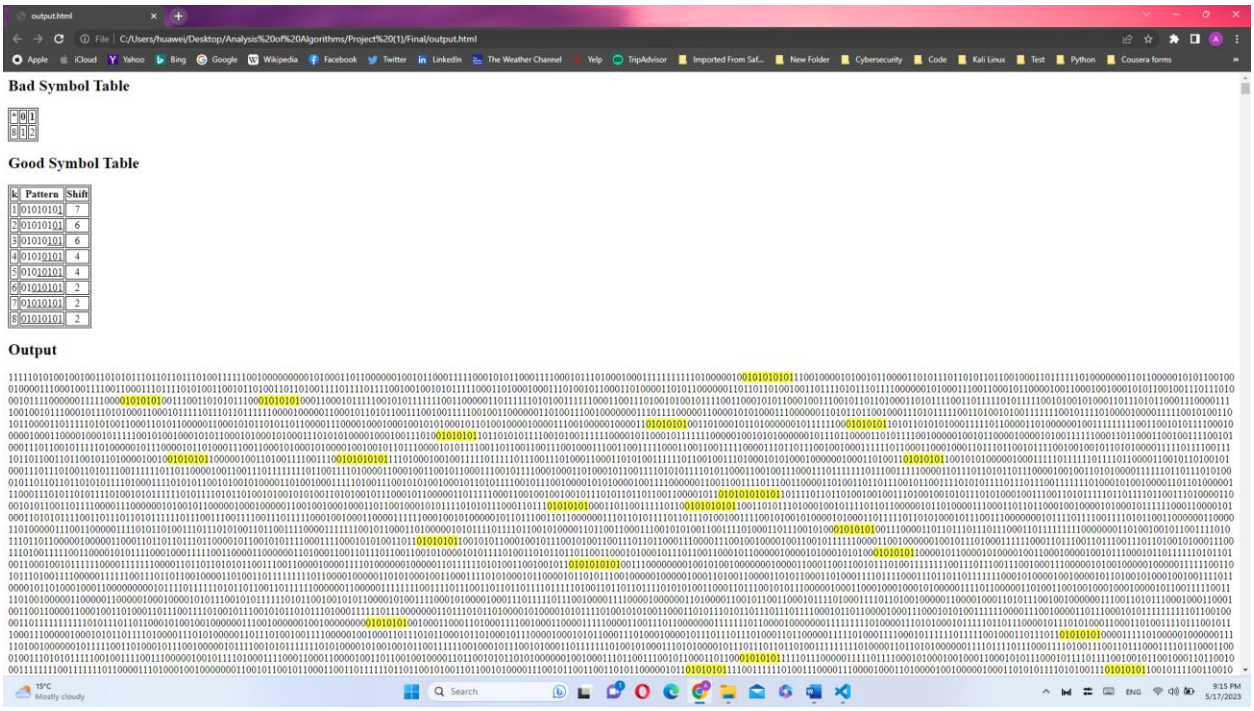
- 01010101

Html file name (example.html): bitInput1.html
Pattern: 01010101

Brute-force Algorithm:
Occurrences: 7229
Comparisons: 4078247
Running time: 16.0835 ms

_____
_____
Horspool Algorithm:
Shift Table:
0: 1
1: 2
Occurrences: 7229
Comparisons: 3054337
Running time: 19.6607 ms
_____
_____
Boyer-Moore Algorithm:
Bad Symbol Table:
0: 1
1: 2
Good Suffix Table:
1: 7
2: 6
3: 6
4: 4
5: 4
6: 2
7: 2
8: 2
Occurrences: 7229
Comparisons: 1264488
Running time: 10.9770 ms
_____





- 00110011

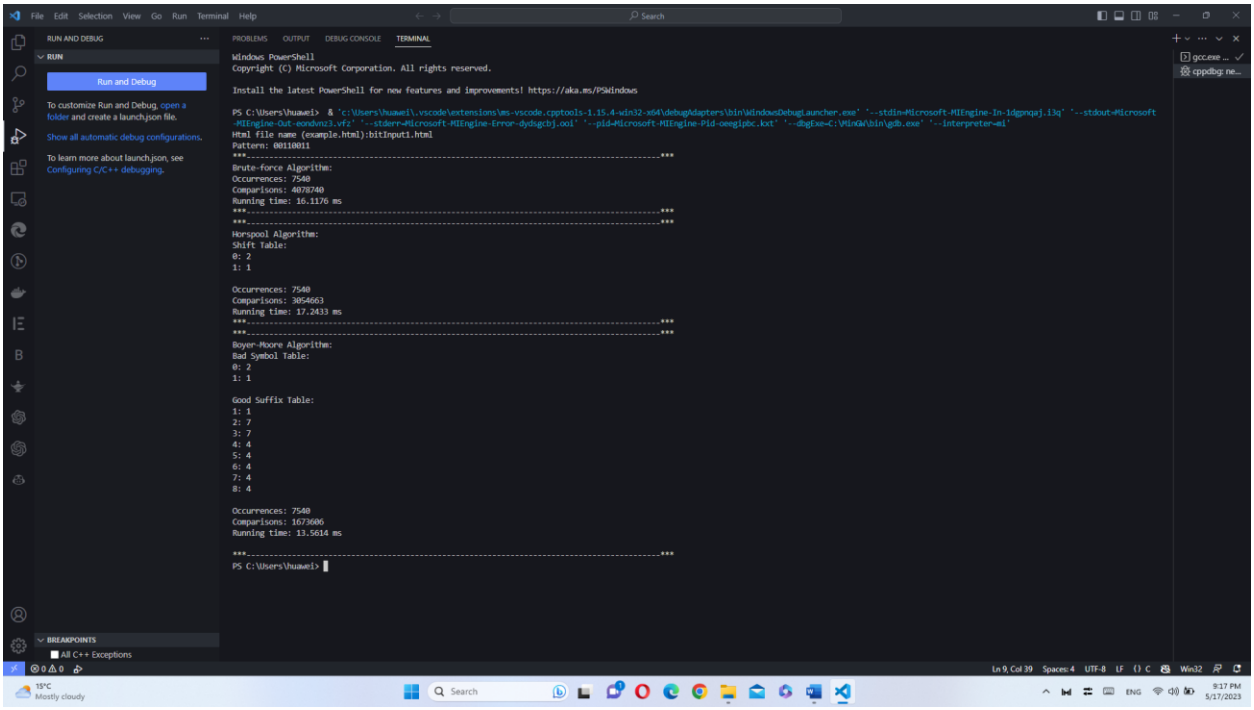
Html file name (example.html): bitInput1.html
Pattern: 00110011

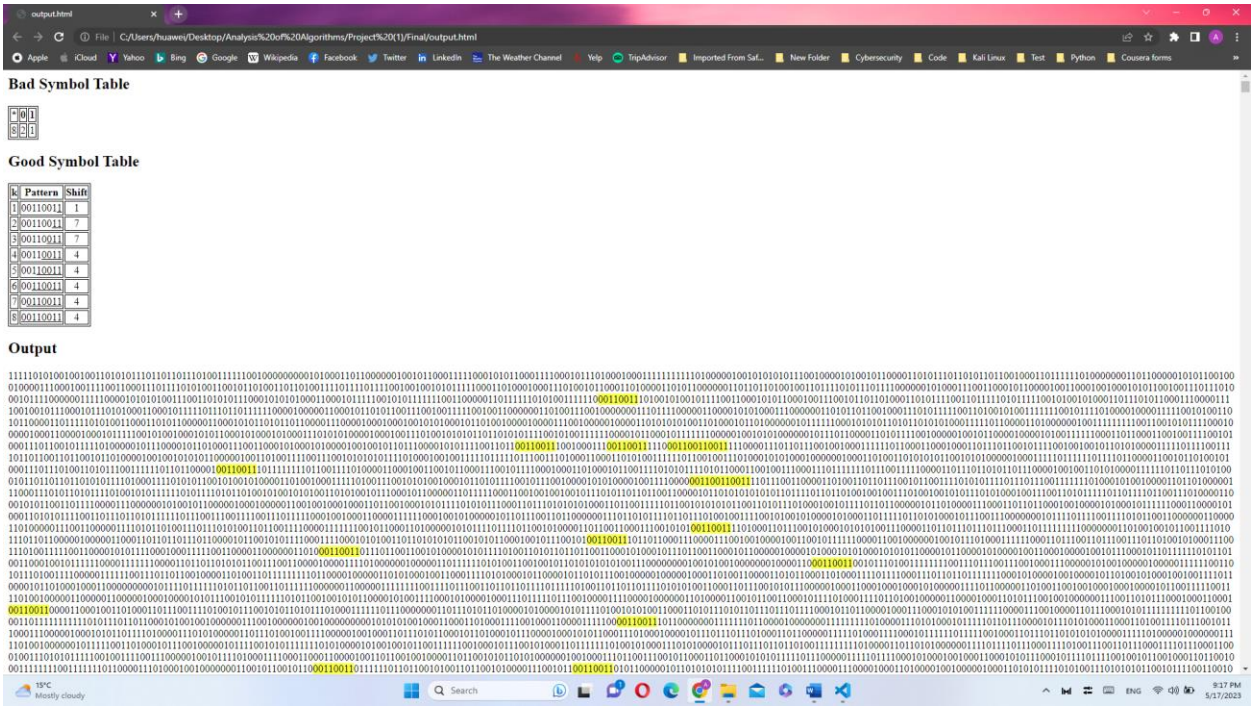
Brute-force Algorithm:
Occurrences: 7540
Comparisons: 4078740
Running time: 16.1176 ms

Horspool Algorithm:
Shift Table:
0: 2
1: 1
Occurrences: 7540
Comparisons: 3054663
Running time: 17.2433 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
0: 2
1: 1
Good Suffix Table:
1: 1
2: 7
3: 7
4: 4
5: 4
6: 4

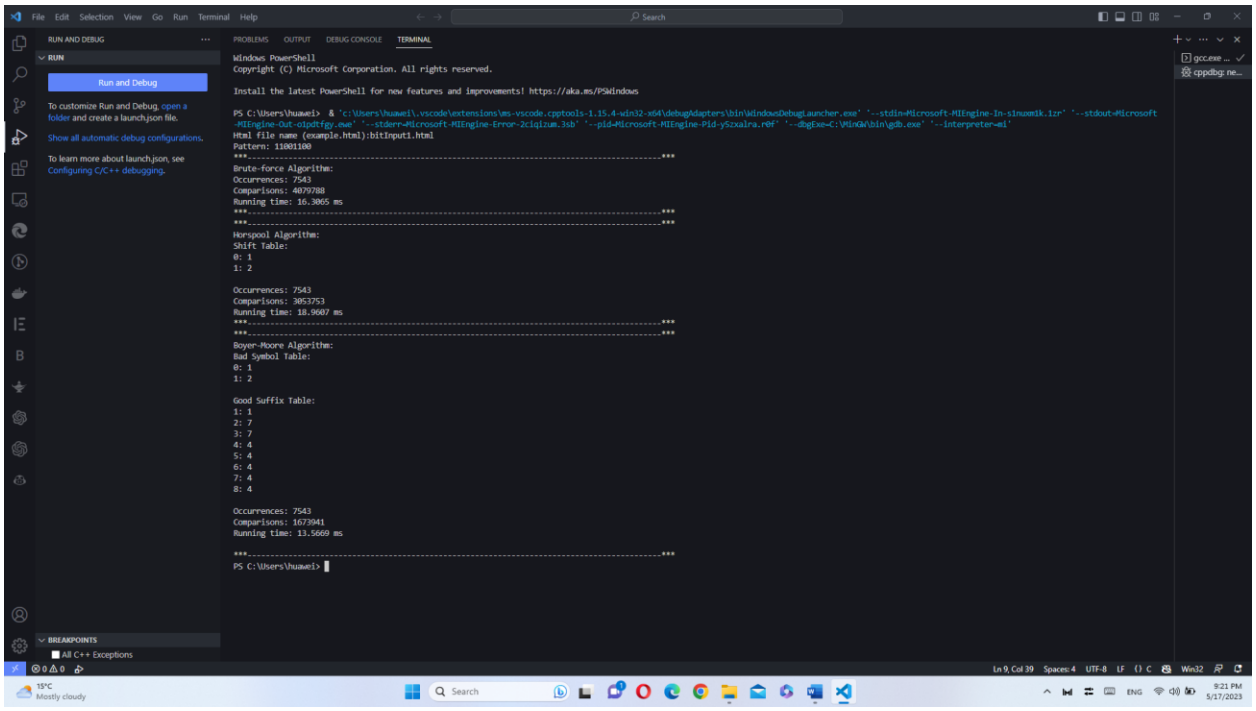
7: 4
8: 4
Occurrences: 7540
Comparisons: 1673606
Running time: 13.5614 ms

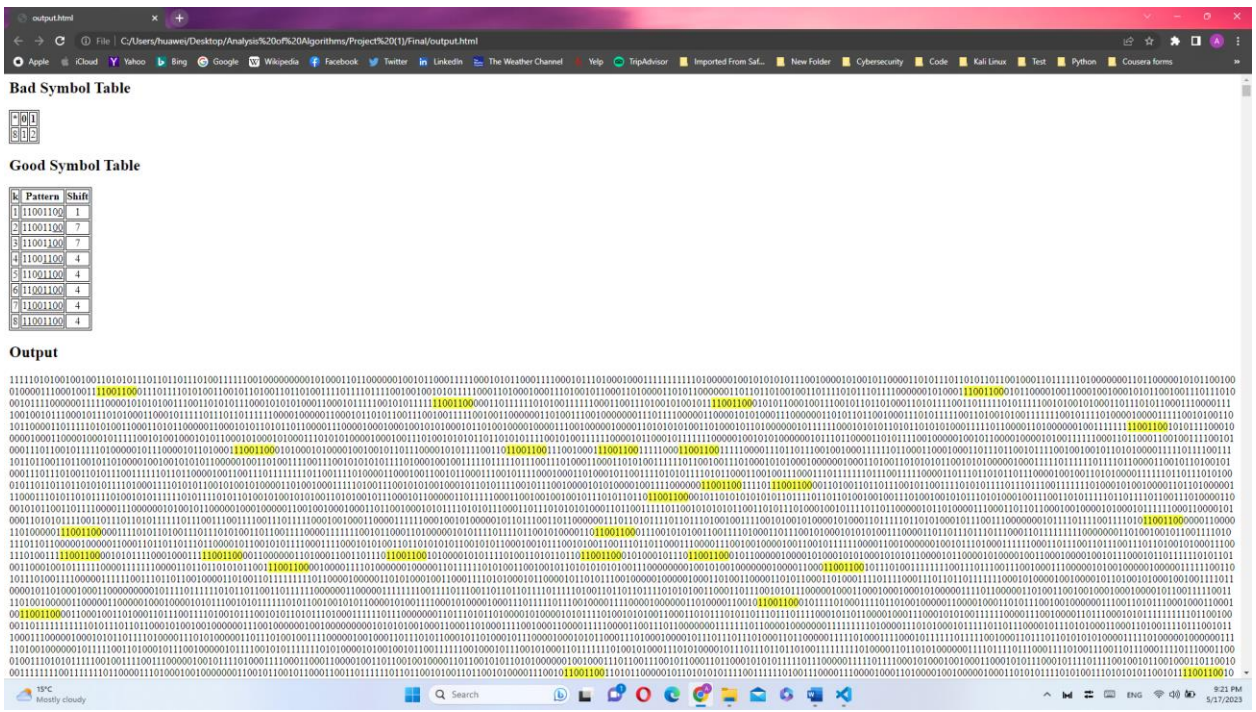




- 11001100

```
Html file name (example.html): bitInput1.html
Pattern: 11001100
***-----------------------------------------------------------------------------------------------***
Brute-force Algorithm:
Occurrences: 7543
Comparisons: 4079788
Running time: 16.3065 ms
***-----------------------------------------------------------------------------------------------***
***-----------------------------------------------------------------------------------------------***
Horspool Algorithm:
Shift Table:
0: 1
1: 2
Occurrences: 7543
Comparisons: 3053753
Running time: 18.9607 ms
***-----------------------------------------------------------------------------------------------***
***-----------------------------------------------------------------------------------------------***
Boyer-Moore Algorithm:
Bad Symbol Table:
0: 1
1: 2
Good Suffix Table:
1: 1
2: 7
3: 7
4: 4
5: 4
6: 4
7: 4
8: 4
Occurrences: 7543
Comparisons: 1673941
Running time: 13.5669 ms
***-----------------------------------------------------------------------------------------------***
```





- Pattern (all ones): 1/11/1111/11111111
  - 1

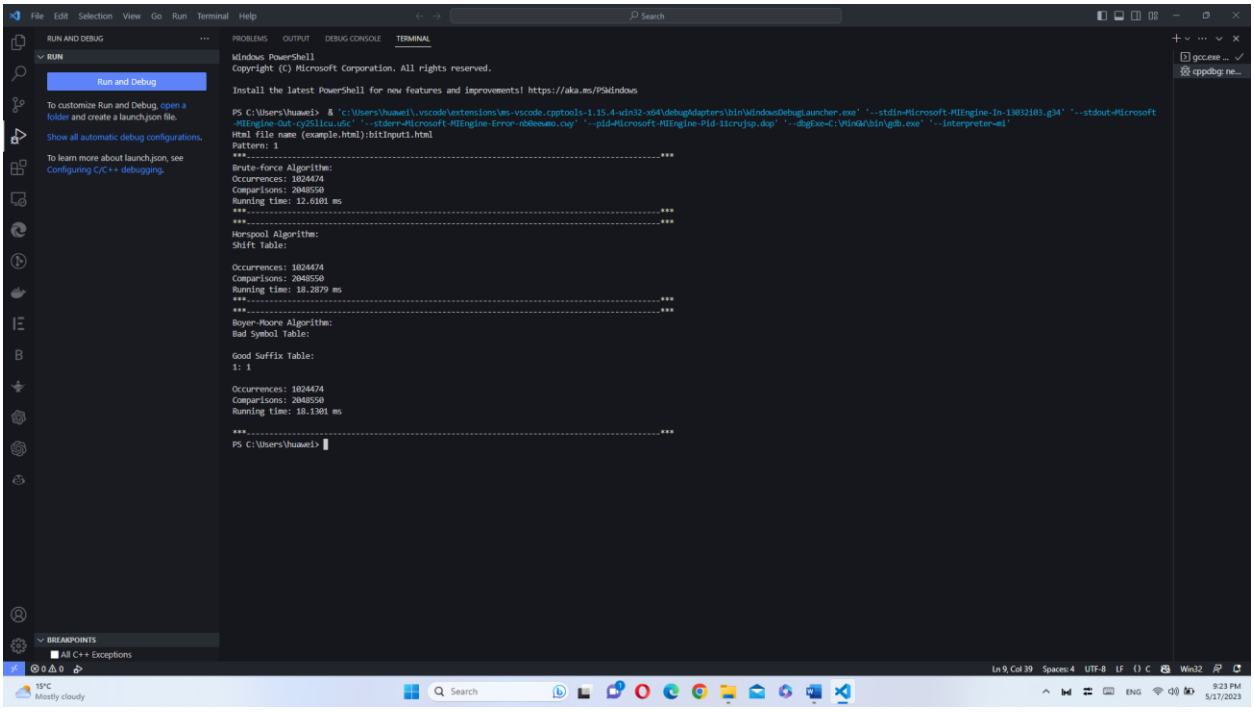
  Html file name (example.html): bitInput1.html
  Pattern: 1

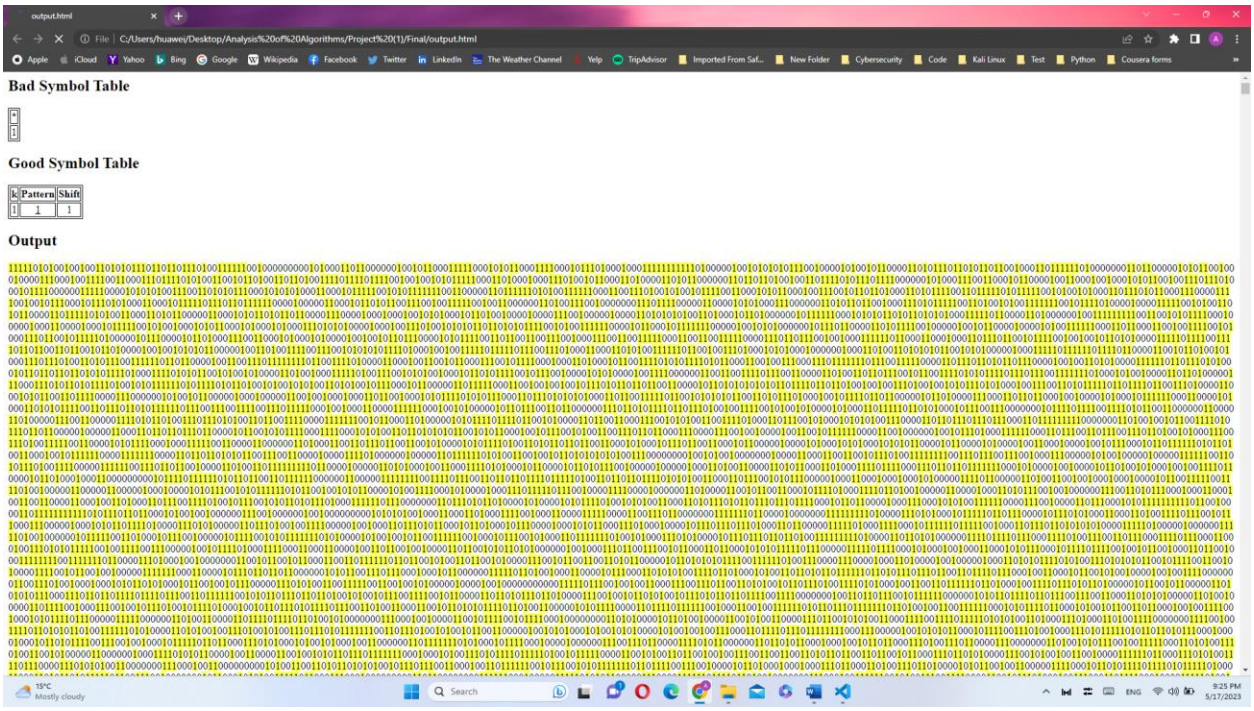
  Brute-force Algorithm:
  Occurrences: 1024474

Comparisons: 2048550
Running time: 12.6101 ms

Horspool Algorithm:
Shift Table:
Occurrences: 1024474
Comparisons: 2048550
Running time: 18.2879 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
Good Suffix Table:
1: 1
Occurrences: 1024474
Comparisons: 2048550
Running time: 18.1301 ms





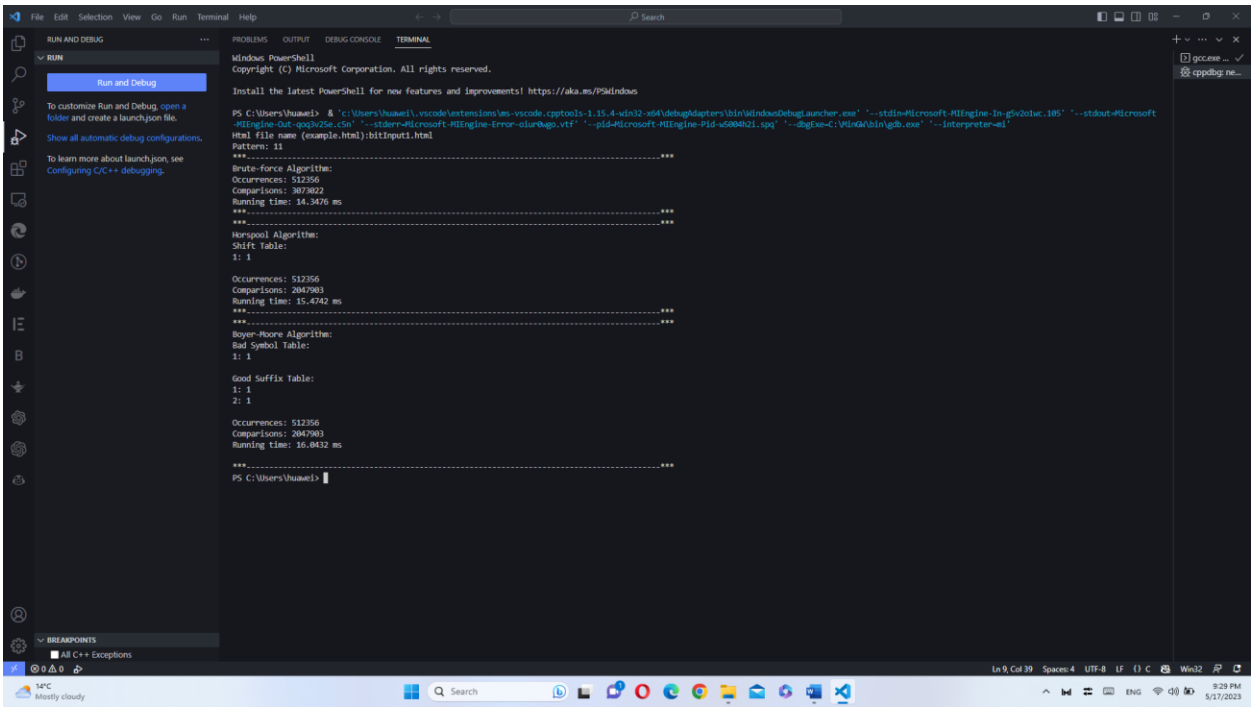
- 11

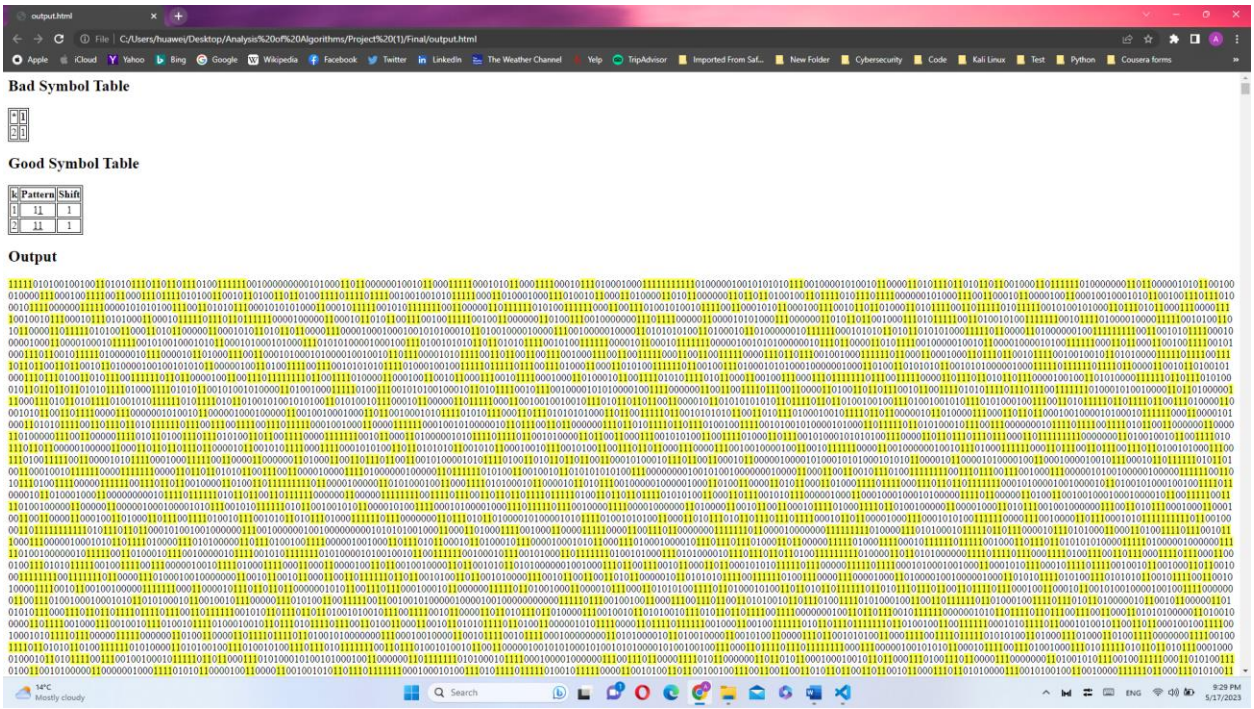
Html file name (example.html): bitInput1.html
Pattern: 11

Brute-force Algorithm:
Occurrences: 512356
Comparisons: 3073022
Running time: 14.3476 ms

Horspool Algorithm:
Shift Table:
1: 1
Occurrences: 512356
Comparisons: 2047903
Running time: 15.4742 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
1: 1
Good Suffix Table:
1: 1
2: 1
Occurrences: 512356
Comparisons: 2047903
Running time: 16.0432 ms





- 1111

Html file name (example.html): bitInput1.html
Pattern: 1111
_____

Brute-force Algorithm:
Occurrences: 127811
Comparisons: 3841587
Running time: 15.4011 ms

_____

_____

Horspool Algorithm:

Shift Table:

1: 1

Occurrences: 127811

Comparisons: 1745725

Running time: 11.0973 ms

_____

_____

Boyer-Moore Algorithm:

Bad Symbol Table:

1: 1

Good Suffix Table:

1: 3

2: 2

3: 1

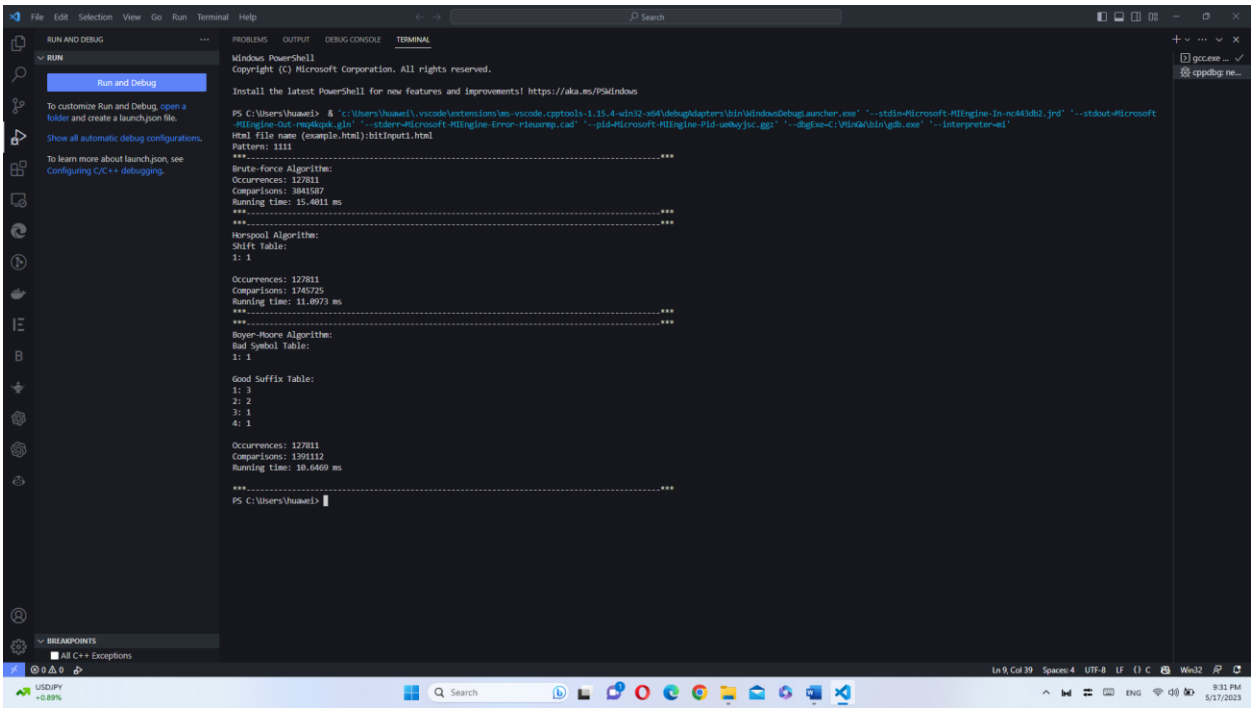
4: 1

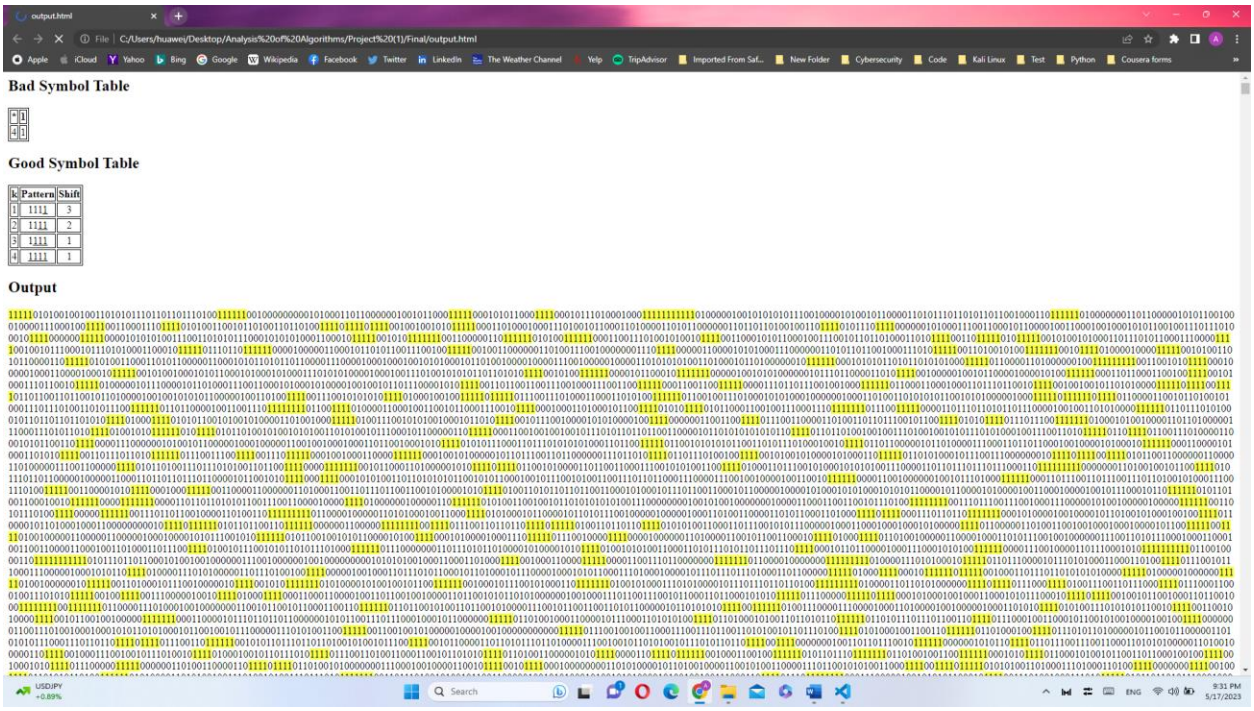
Occurrences: 127811

Comparisons: 1391112

Running time: 10.6469 ms

_____





- 11111111

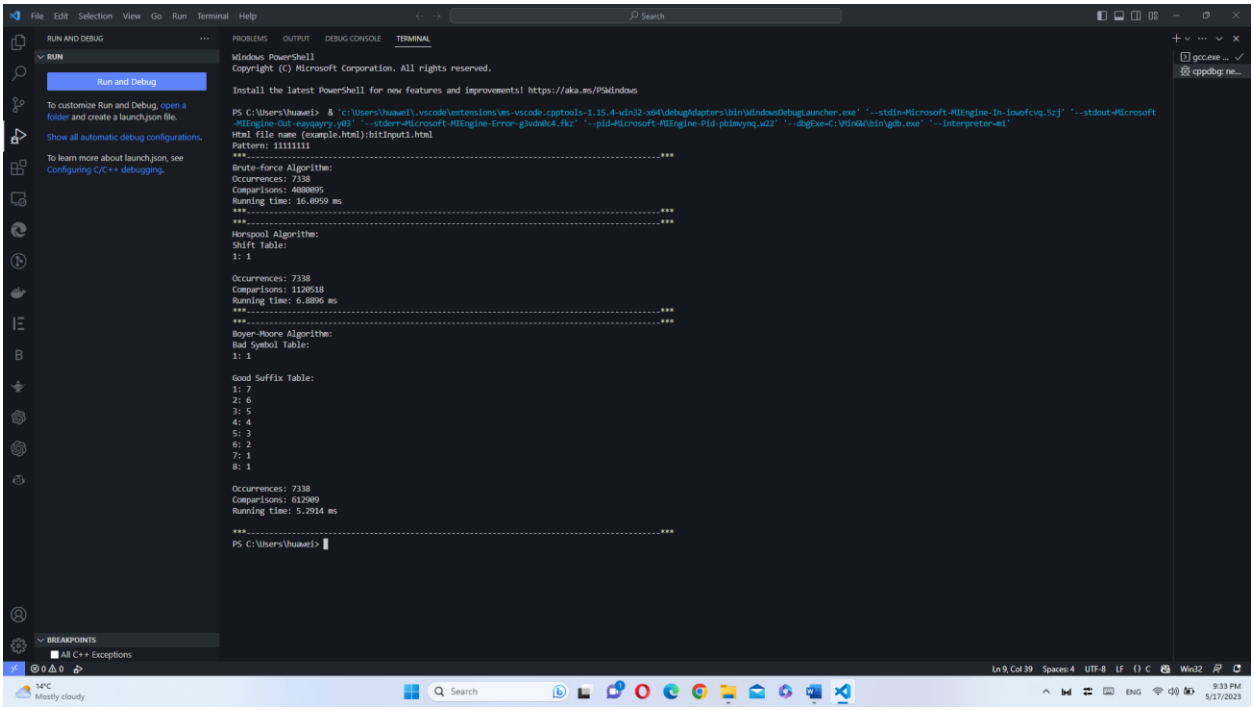
Html file name (example.html): bitInput1.html
Pattern: 11111111

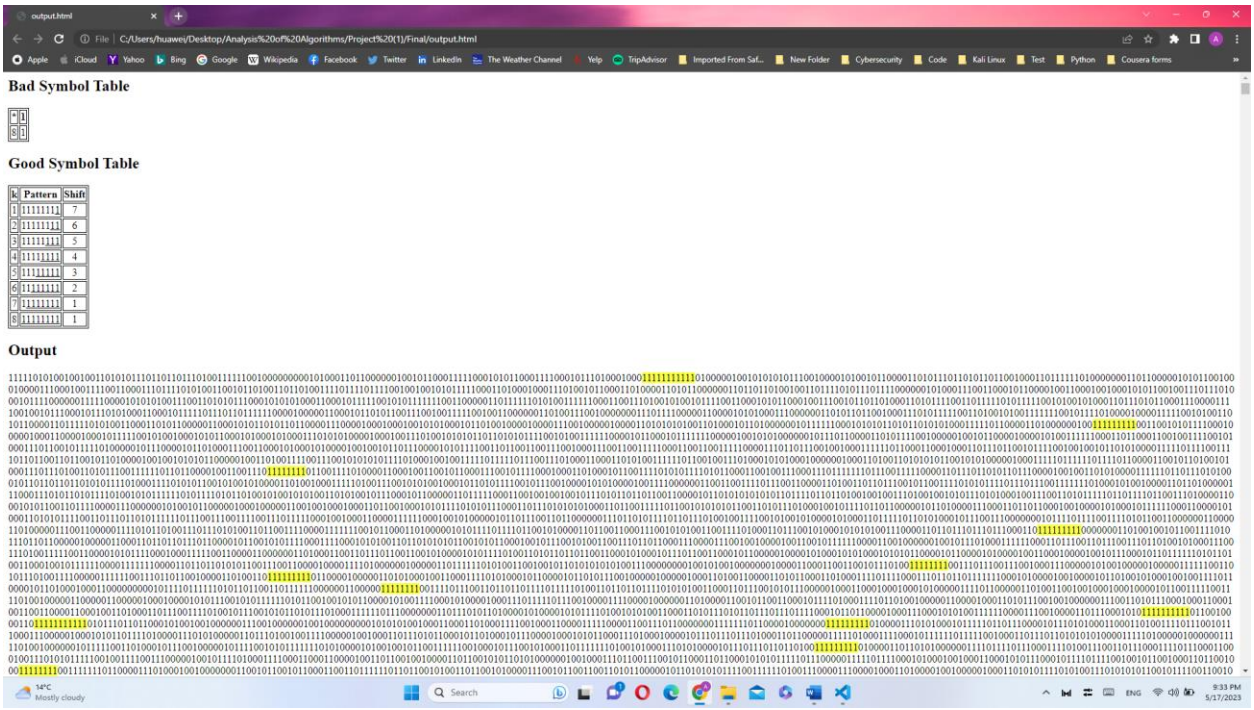
Brute-force Algorithm:
Occurrences: 7338
Comparisons: 4080095
Running time: 16.0959 ms

Horspool Algorithm:
Shift Table:
1: 1
Occurrences: 7338
Comparisons: 1120518
Running time: 6.8896 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
1: 1
Good Suffix Table:
1: 7
2: 6
3: 5
4: 4
5: 3
6: 2
7: 1
8: 1

Occurrences: 7338
Comparisons: 612909
Running time: 5.2914 ms
_____





- Pattern (all zeros): 0/00/0000/00000000
  - 0

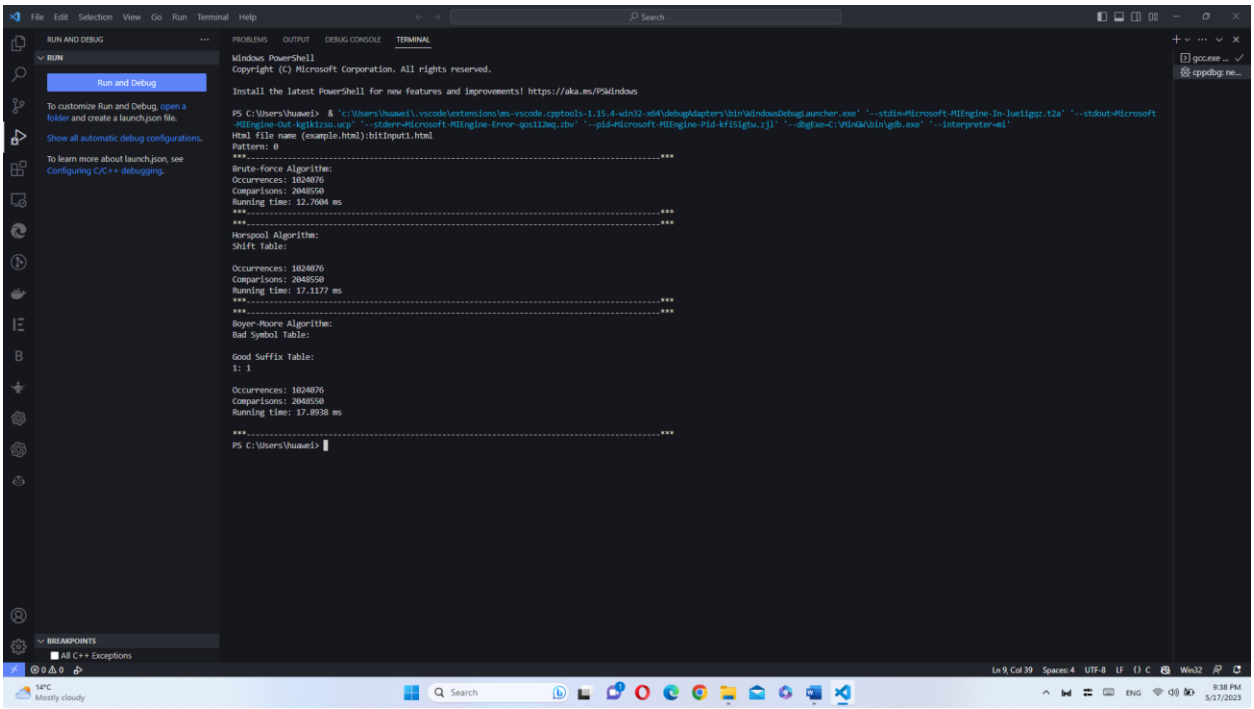
Html file name (example.html): bitInput1.html

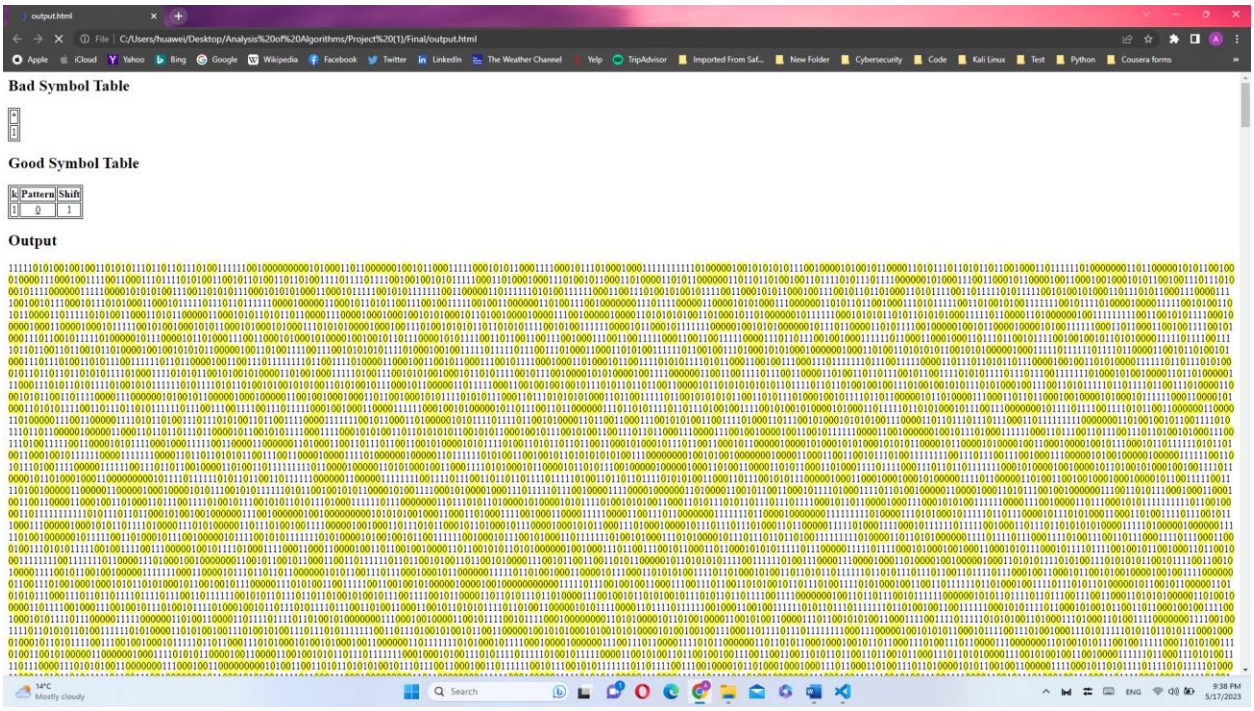
Pattern: 0

Brute-force Algorithm:
Occurrences: 1024076
Comparisons: 2048550
Running time: 12.7604 ms

Horspool Algorithm:
Shift Table:
Occurrences: 1024076
Comparisons: 2048550
Running time: 17.1177 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
Good Suffix Table:
1: 1
Occurrences: 1024076
Comparisons: 2048550
Running time: 17.8938 ms





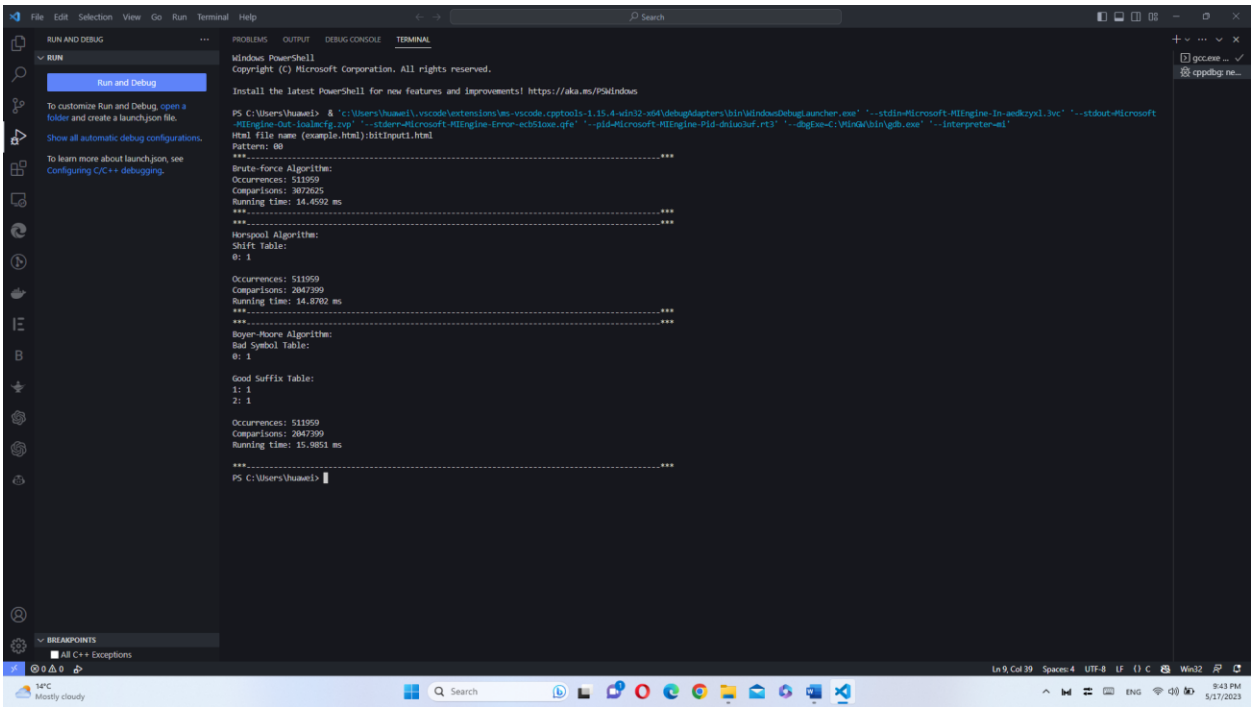
- 00

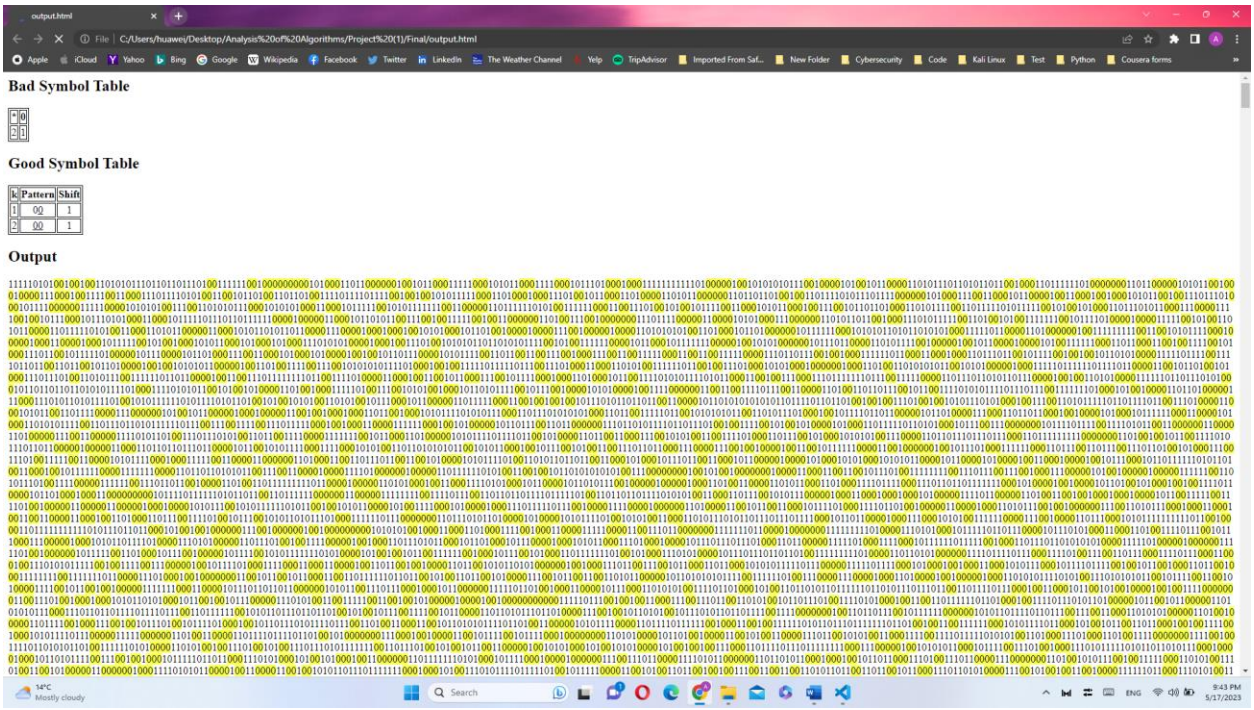
Html file name (example.html): bitInput1.html
Pattern: 00

Brute-force Algorithm:
Occurrences: 511959
Comparisons: 3072625
Running time: 14.4592 ms

Horspool Algorithm:
Shift Table:
0: 1
Occurrences: 511959
Comparisons: 2047399
Running time: 14.8702 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
0: 1
Good Suffix Table:
1: 1
2: 1
Occurrences: 511959
Comparisons: 2047399
Running time: 15.9851 ms



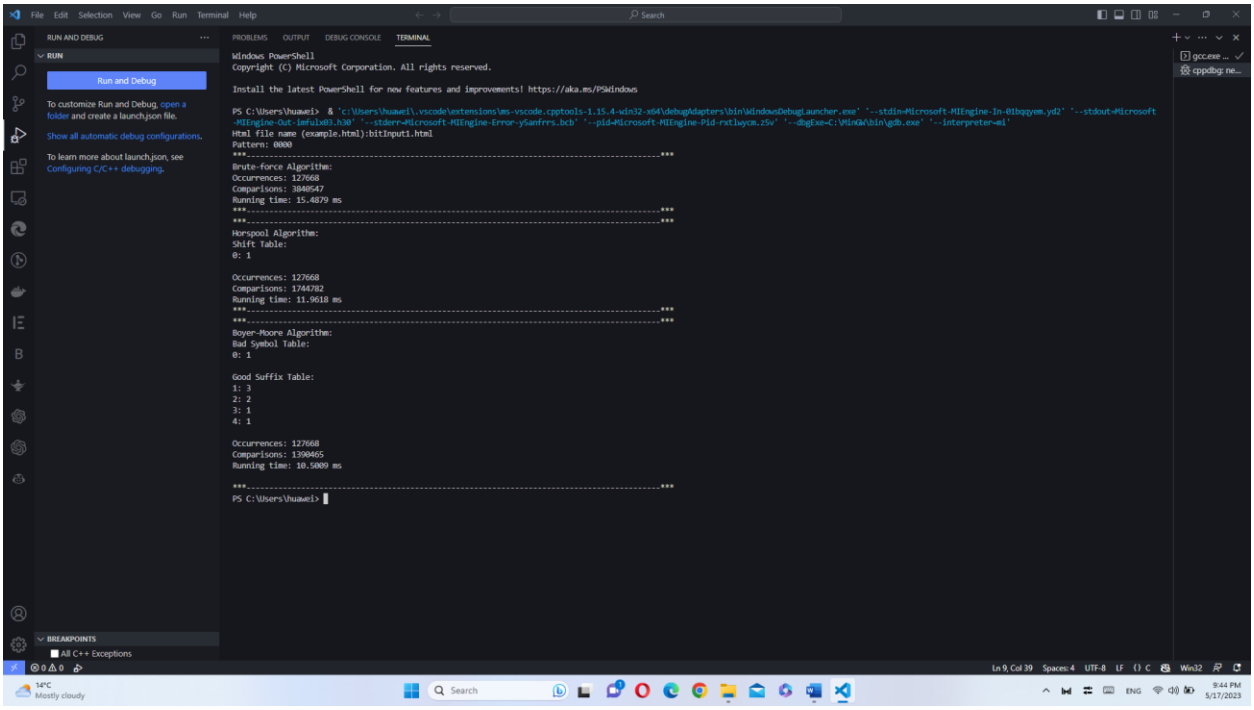


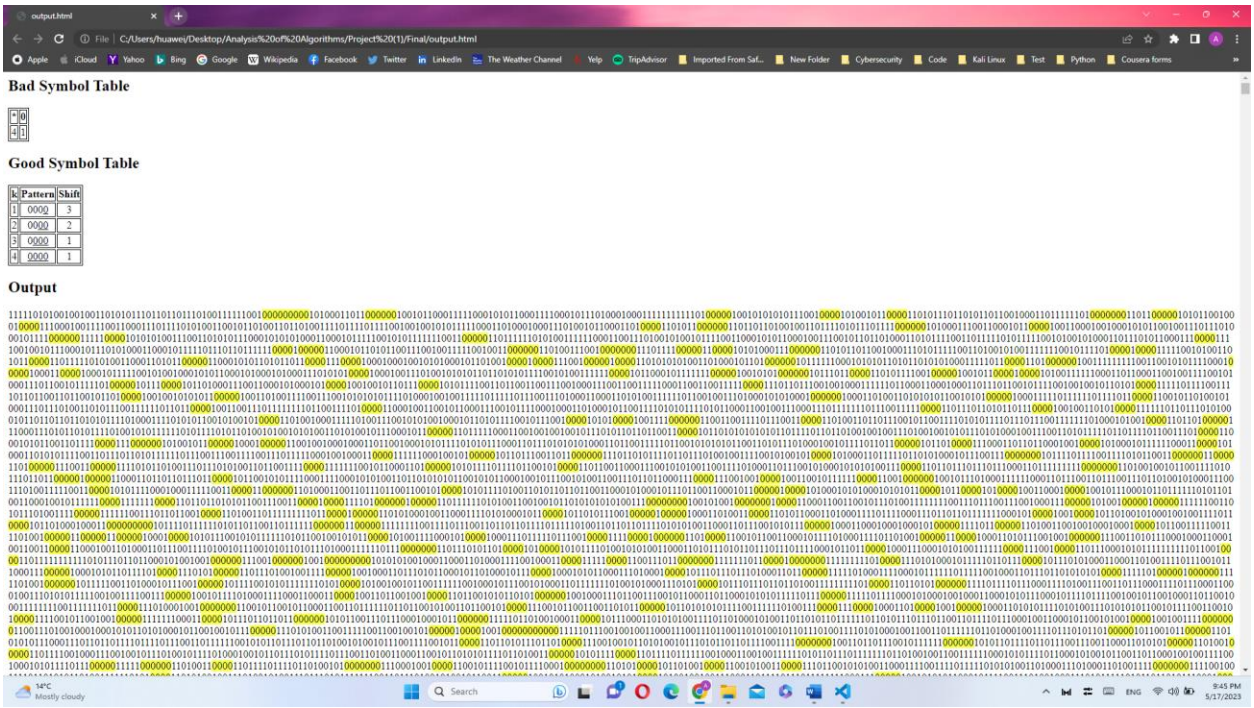
- 0000

Html file name (example.html): bitInput1.html
Pattern: 0000
_____

Brute-force Algorithm:
Occurrences: 127668
Comparisons: 3840547
Running time: 15.4879 ms

_____
_____
Horspool Algorithm:
Shift Table:
0: 1
Occurrences: 127668
Comparisons: 1744782
Running time: 11.9618 ms
_____
_____
Boyer-Moore Algorithm:
Bad Symbol Table:
0: 1
Good Suffix Table:
1: 3
2: 2
3: 1
4: 1
Occurrences: 127668
Comparisons: 1390465
Running time: 10.5009 ms
_____





- 00000000

Html file name (example.html): bitInput1.html
Pattern: 00000000

Brute-force Algorithm:
Occurrences: 7300
Comparisons: 4078729
Running time: 16.0285 ms

Horspool Algorithm:
Shift Table:
0: 1

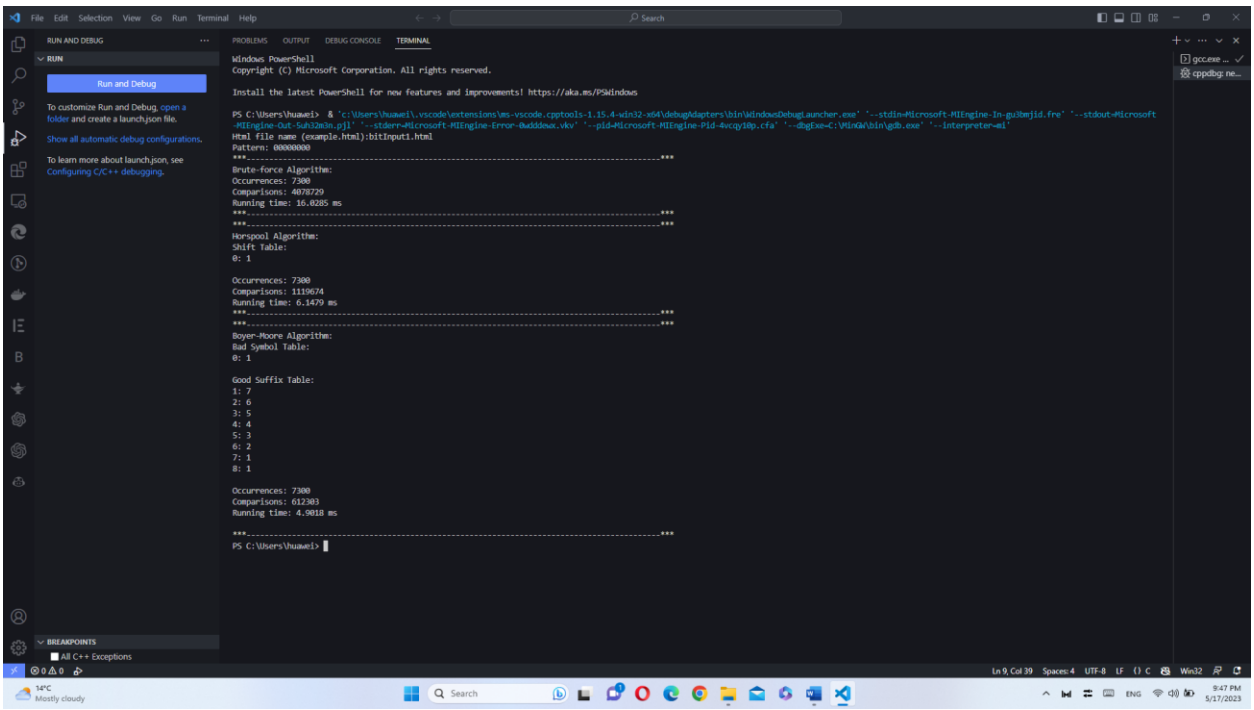
Occurrences: 7300
Comparisons: 1119674
Running time: 6.1479 ms

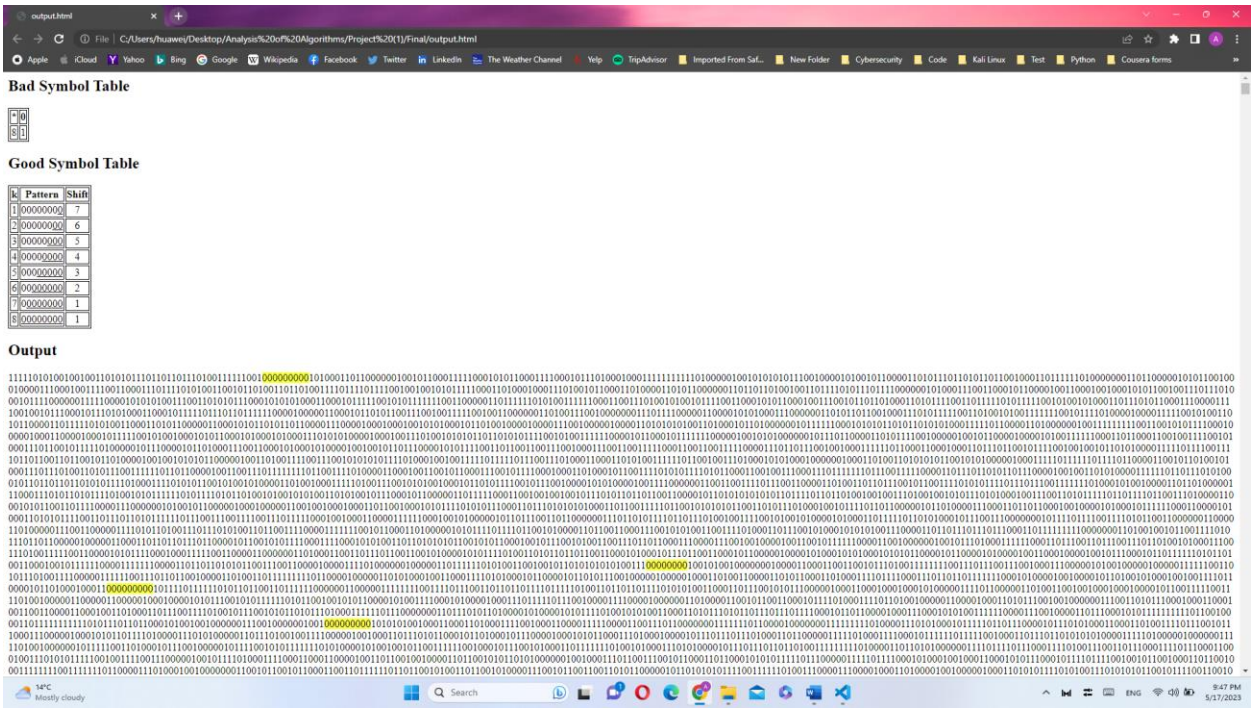
Boyer-Moore Algorithm:
Bad Symbol Table:
0: 1

Good Suffix Table:
1: 7
2: 6
3: 5
4: 4
5: 3
6: 2

7: 1
8: 1

Occurrences: 7300
Comparisons: 612303
Running time: 4.9018 ms





Input (2): bitInput2.html

bitInput2.html bitInput2.txt

- Randomized pattern:
  - 10101010

Brute-force Algorithm:
Occurrences: 7592
Comparisons: 3761445
Running time: 26.3860 ms

Horspool Algorithm:
Shift Table:
0: 2
1: 1
Occurrences: 7592
Comparisons: 2898825
Running time: 27.2680 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
0: 2
1: 1
Good Suffix Table:
1: 7
2: 6
3: 6
4: 4
5: 4
6: 2
7: 2
8: 2
Occurrences: 7592
Comparisons: 1177406
Running time: 13.4650 ms

Bad Symbol Table

Screenshot 2023-05-1

*	0	1
8	2	1

Good Symbol Table

k	Pattern	Shift
1	10101010	7
2	10101010	6
3	10101010	6
4	10101010	4
5	10101010	4
6	10101010	2
7	10101010	2
8	10101010	2

Output


```
0000100111100000000101101101011111000101010110001110001001110110100000101010001001100000100101111010000001010101111001111111011101100000100110
1111101011101011111101011111110010010100101011111000101011000111010000100111110010101110101000011010100000011000100111101110101110000000011001001
1010000011000110101000000001101011001101100001001011110111000010100011000011111101011001101101011110111000111101011110000000111110010010101111110
0001111000010011111110100100001101100000001011000101100000001111100101000010111101001100010011111101011010010011011110000011111110111100000010101
1101110000111111011100000011101101011111011101100100100011000010110111101101101111000110000010001000110000100101001001001001110111100100100111
0101101101111101111101011000111011001001100011101101001100000111001110100010111111001010100100101001100001011011000110101001101000100100011 0
0001001010101010001100100110011100000000100110010011011111011001101000001001001101011100110010101110111101011100110000000100001100110101001000 10
1100110001010010000010101001001010101010100000001101110110101010000110111010110000010000000111100000100110101110010110011011010000110011001111100
0101110110110010000010000100111011011001111000001010000111100110010110011011010011100101110010001110100111011100010100010011011001101011100111
1001101100010001100011110001100110001001110001100000101100010001001000010001010010000011011111001000110100001101000011010111110100000000001001
1110111011110101001000001011101011100001000011011101000011001010011101110100111010000000110001101000001010001110011101001000001001110010101111100
0100101101000011111000101110011010101000010010110101101100100011001000111011100100011001110100001011000000101010001100111100101001 0011
```

- 11110000

****----------***

Brute-force Algorithm:
Occurrences: 7413
Comparisons: 3624844
Running time: 26.0750 ms

Horspool Algorithm:
Shift Table:
0: 1
1: 4

Occurrences: 7413
Comparisons: 2058513
Running time: 18.7640 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
0: 1
1: 4

Good Suffix Table:
1: 3
2: 2
3: 1
4: 7

5: 7
6: 7
7: 7
8: 8

Occurrences: 7413
Comparisons: 1220951
Running time: 13.9750 ms
****----------***

Bad Symbol Table

*	0	1
8	1	2

Good Symbol Table

k	Pattern	Shift
1	01010101	7
2	01010101	6
3	01010101	6
4	01010101	4
5	01010101	4
6	01010101	2
7	01010101	2
8	01010101	2

Output

0000100111100000000101101101011111100010101011000111000100111011010000010101000100110000010010111101000000101010111100111111101110110000001001101
1111101011101011111010111111100100101001010111100010101100011101000010011111001010111010100001101010000011000100111101110101110000000011001001
1010000011000110101000000001101011001101100001001011110111000010100011000011111101011001101101011101110001111010111100000001111100100101011111 0
0001111000010011111110100100001101100000001011000101100000001111100101000010111101001100010011111101011010010011011100000111111011110000001010
1101110000111111011100000001110110101111101111011100100100011000010110111101101101111000110000010001000110000100101001001001001110111100100100111
0101101110111110111110101100011101100100110001110110100110000011100111010001011111100101010010010101001100001011011000110101001101000100100011 0
0001010101010001100100110011100000000100110010011011111011001101000000100100110101110011001010110111110110101110011000000010000110011010100010
1100110001010010000010101001001010101001000000011011110110101010000011011101011000000100000000111100000010011010111001011001101101000011001100111100
0101110110110010000010000100111011011011001111000001010000111100110011010011011010011100101110010001110100111011100010100010011011001101011100111
1001101100010001100011110001100110001001110001100000101100010001001000010001010010000011011111001000110100001101000011010111110100000000001001
1110111011110101001000001011101011100001000011011101000011001010011101110100111010000000110001101000001010001110011101001000001001110010101111100
0100101101000011111000101110011010101000010010110101101100100011001000111011100100011001110110001110100000101100000010101000110011110010010011

- 01010101

Brute-force Algorithm:
Occurrences: 7484
Comparisons: 3923854
Running time: 26.7190 ms

Horspool Algorithm:
Shift Table:
0: 1
1: 2

Occurrences: 7484
Comparisons: 2858430
Running time: 28.0400 ms

Boyer-Moore Algorithm:
Bad Symbol Table:

0: 1
1: 2

Good Suffix Table:
1: 7
2: 6
3: 6
4: 4
5: 4
6: 2
7: 2
8: 2

Occurrences: 7484
Comparisons: 1220159
Running time: 14.1270 ms

Bad Symbol Table

*	0	1
8	1	4

Good Symbol Table

k	Pattern	Shift
1	11110000	3
2	11110000	2
3	11110000	1
4	11110000	7
5	11110000	7
6	11110000	7
7	11110000	7
8	11110000	8

Output

0000100111100000001011011010111110001010101100011100010011101101000001010100010011000001001011110100000010101011110011111110111011000001001101
1111101011101011111010111111100100101001010111100010101100011101000010011111001010111010100001101010000001100010011110111010111000000001100100
1010000011000110101000000001101011001101100001001011110111000010100011000011111101011001101101011110111000111101011110000000111111001001010111110
0001111000010011111110100100001101100000001011000101100000001111100101000010111101001100010111111010110100100110111000001111111011110000001010
1101110000111111011110000001110110101111101110011001001000110000101101111101101111100011000001000100011000010010100100100100111011110010010011
0101101110111101111101011000111011001001100011101101001100000111001110100010111111001010100100101010011000010110110001101010011010001001000110
0001001010101000110010011100111000000010011001001101111101100110100000100100110101110011001010111011110110101110011000000010000110011010100010
1100110001010010000010101001001010101001000000110111011010101000011011010110000010000000111100000010011010111001011001101010000110011001111100
0101110110110100000010000100111011011001111000010101000011110011001101001101101001110010111001000111010011101110001010001001101100110101110011100
1001101100010001100011110001100110001001110001100000101100010001001000010001010010000011011111001000110100001101000011010111111010000000001001

- 00110011

Brute-force Algorithm:
Occurrences: 7314
Comparisons: 3985410
Running time: 27.1330 ms

Horspool Algorithm:
Shift Table:
0: 2

1: 1

Occurrences: 7314
Comparisons: 2708224
Running time: 27.0770 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
0: 2
1: 1

Good Suffix Table:
1: 1
2: 7
3: 7
4: 4
5: 4
6: 4
7: 4
8: 4

Occurrences: 7314
Comparisons: 1591050
Running time: 18.0630 ms

Bad Symbol Table

*	0	1
8	2	1

Good Symbol Table

k	Pattern	Shift
1	00110011	1
2	00110011	7
3	00110011	7
4	00110011	4
5	00110011	4
6	00110011	4
7	00110011	4
8	00110011	4

Output

0000100111100000001011011010111110001010101100011100010011101101000001010100010011000001001011110100000010101011110011111101110110000010011011111101011011011

- 11001100

Brute-force Algorithm:
Occurrences: 7423
Comparisons: 3693881
Running time: 26.3420 ms

Horspool Algorithm:
Shift Table:
0: 1
1: 2

Occurrences: 7423
Comparisons: 3041315
Running time: 27.8520 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
0: 1
1: 2

Good Suffix Table:
1: 1
2: 7
3: 7
4: 4
5: 4
6: 4
7: 4
8: 4

Occurrences: 7423
Comparisons: 1564443
Running time: 17.1960 ms

Bad Symbol Table

*	0	1
8	1	2

Good Symbol Table

k	Pattern	Shift
1	11001100	1
2	11001100	7
3	11001100	7
4	11001100	4
5	11001100	4
6	11001100	4
7	11001100	4
8	11001100	4

Output


```
0000100111100000000101101101011111000101010110001110001001110110100000101010001001100000100101111010000001010101111001111111011101100000100110
1111101011101011111101011111110010010100101011111000101011000111010000100111110010101110101000011010100000011000100111101110101110000000011001001
1010000011000110101000000001101011001101100001001011110111000010100011000011111101011001101101011101110001110101111000000011110010010101111110
0001111000010011111110100100001101100000001011000101100000001111100101000010111101001100010011111101011010010010110110000011111101111000000101
1101110000111110111000000111011010111110111011100100100011000010110111101101111000110000010001000110000100101001001001001110111100100100111
0101101110111101111101011000111011001001100011101101001100000011001110100010111111001010100100101010011000010110110001101010011010001001000110
0001001010101000110010011100111000000001001100100110111110110011010000010010011010111001100110101011101111011010111001100000010000110011010100010
110011000101010000010101001001010100100000011011101101010000110111010110000010000000111100000100110101110010110011011010000110011001111000
0101110110110010000010000010011101101100111100000101000011100110010011101001101101001110010011101001110110001010001001101100110101011100111
1001101100010001100011110001100110010010011100011000001011000100010010000100010100100000110111110010001101000011010000110101111110100000000001001
1110111011101010010000010111010111000010000110111010000110010100111011101001110100000001100011010000010100011100111010010000010011100101111100
0100101101000011111000101110011101010100001001011010110110010001100100011101110010001100111011000110100000101100000010101000110011110010010011
```


- Pattern (all ones): 11111111

  - 11111111

Brute-force Algorithm:
Occurrences: 3774
Comparisons: 3604582
Running time: 25.8620 ms

Horspool Algorithm:
Shift Table:
1: 1

Occurrences: 3774
Comparisons: 892582
Running time: 8.4380 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
1: 1

Good Suffix Table:
1: 7
2: 6
3: 5
4: 4
5: 3

6: 2
7: 1
8: 1

Occurrences: 3774
Comparisons: 510453
Running time: 6.4900 ms

Bad Symbol Table

*	1
8	1

Good Symbol Table

k	Pattern	Shift
1	1111111<u>1</u>	7
2	111111<u>11</u>	6
3	11111<u>111</u>	5
4	1111<u>1111</u>	4
5	111<u>11111</u>	3
6	11<u>111111</u>	2
7	1<u>1111111</u>	1
8	<u>11111111</u>	1

Output

0000100111100000001011011010111110001010101100011100010011101101000001010100010011000001001011110100000010101011110011111110111011000001001101
1111101011101011111010111111100100101001010111110001010110001110100001001111100101011101010000110101000000110001001111011101011100000000011001001
1010000011000110101000000001101011001101100001001011110111000010100011000011111101011001101101011101110001111010111100000001111100100101011111 0
0001111000010011111110100100001101100000001011000101100000001111100101000010111101001100010011111101011010010011011100000111111011110000001010
1101110000111111011100000011101101011111011101100100100011000010110111101101111000110000010001000110000100101001001001001110111100100100111
0101101111011111101011000111011001001100011101101001100000111001110100010111111001010100100101001100001011011000110101001101000100100011 0
0001001010101000110010011100111000000010011001001101111101100110100000100100110101110011001010111011110110101110011000000010000110011010100010
1100110001010010000010101001001010101001000000110111011010101000011011101011000001000000001111000001001101011100101100110110100001100110011110 0
0101110110110010000010000100111011011001111000001010000111100110010110110100110010111001000111010011101110001010001001101100110101110011 1
1001101100010001100011110001100110001001110001100000101100010001001000010001010010000011011111001000110100001101000011010111110100000000001001
1110111011110100100000010111010111000010000110111010000110010100111011101001110100000000110001101000001010001110011101001000001001110010101111100
0100101101000011111000101110011101010100001001011010110110010001100100011101110010001100111011000111010000010110000001010100011001111001001001 1

- Pattern (all zeros): 00000000
  - 00000000

Brute-force Algorithm:
Occurrences: 11109
Comparisons: 4074118
Running time: 26.9590 ms

Horspool Algorithm:
Shift Table:
0: 1
Occurrences: 11109
Comparisons: 1175429
Running time: 10.2940 ms

Boyer-Moore Algorithm:

Bad Symbol Table:
0: 1
Good Suffix Table:
1: 7
2: 6
3: 5
4: 4
5: 3
6: 2
7: 1
8: 1
Occurrences: 11109
Comparisons: 631364
Running time: 7.6160 ms

Input (3): bitInput3.html

bitInput3.html bitInput3.txt

- Randomized pattern:
  - 10101010

Brute-force Algorithm:
Occurrences: 6871
Comparisons: 4294862
Running time: 27.2770 ms

Horspool Algorithm:
Shift Table:
0: 2
1: 1

Occurrences: 6871
Comparisons: 2945703
Running time: 29.3890 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
0: 2
1: 1

Good Suffix Table:
1: 7

2: 6
3: 6
4: 4
5: 4
6: 2
7: 2
8: 2

Occurrences: 6871
Comparisons: 1333688
Running time: 14.7900 ms

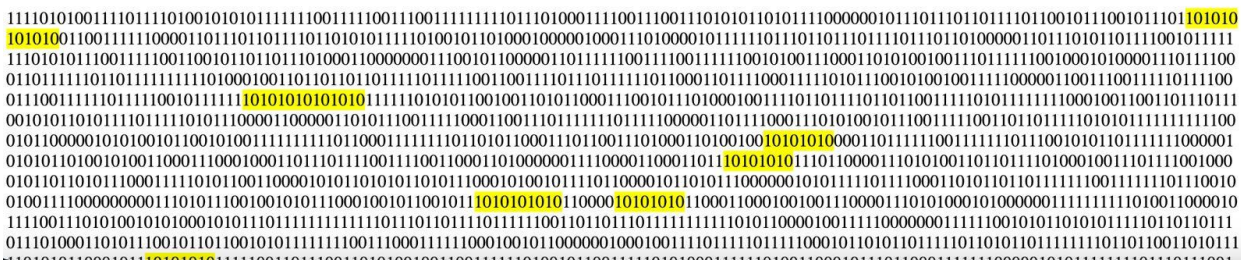
Bad Symbol Table

*	0	1
8	2	1

Good Symbol Table

k	Pattern	Shift
1	10101010	7
2	10101010	6
3	10101010	6
4	10101010	4
5	10101010	4
6	10101010	2
7	10101010	2
8	10101010	2

Output



- 11110000

Brute-force Algorithm:
Occurrences: 6918
Comparisons: 4865613
Running time: 28.7990 ms

Horspool Algorithm:
Shift Table:
0: 1
1: 4

Occurrences: 6918

Comparisons: 1440414
Running time: 15.5600 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
0: 1
1: 4

Good Suffix Table:
1: 3
2: 2
3: 1
4: 7
5: 7
6: 7
7: 7
8: 8

Occurrences: 6918
Comparisons: 1032473
Running time: 12.3820 ms

Bad Symbol Table

*	0	1
8	1	4

Good Symbol Table

k	Pattern	Shift
1	11110000	3
2	11110000	2
3	11110000	1
4	11110000	7
5	11110000	7
6	11110000	7
7	11110000	7
8	11110000	8

Output

11110101001111011110100101010111111100111110011100111111111101110100011110011100111010101101011110000001011101110110111101100101110010111001101101010
10101001100111110000110111011011110110101011111010010110100010000010001110100001011111101110110110111101110110100000110111010110111100101111
11101010111001111100110010110110111010001100000001110010110000011011111100111100111111001010011100011010100100111011111100100010100001110111100
01101111110110111111111101000100110110110110111110111110011001111011101111110110001101111000111110101110010100100111110000011001110011111011100
01110011111101111100101111111010101010101011111110101011001001101011000111001011101000100111101011110110111011001111101011111111000100110011011101110111
00101011010111101111101011100001100000110101110011111000110011101111111011110000011011110001110101001011100111110011011011111101010111111111100
01011000001010100101100101001111111111101100011111111011010110001110110011101000110100100101010100001101111110011111110111001010110111111000001
01010110100101001100011100010001101110111001110011001100011010000001110000110001101110101010101110110000111010100110110111101000100110111001000
01011011010111000111110101100110000101101010110110110000101001011110110000101101011100000010101111101110001101011011011111110011111110100110
01001110000000000011101011100100101011100010010110010111010101010110000010101010110001100010010011100001110101000101000000111111111101001100001
11110011101010010101000101011101111111111111101110110111101111110011011011101111111110101100001001110000000011111100101011010101111101101101
01110100011010111001011011001010111111110011100011111100010010110000001000100111101111110111110001011010110111111101101101111111101101100110101011

- 01010101

Brute-force Algorithm:

Occurrences: 6820
Comparisons: 3770175
Running time: 25.7860 ms

Horspool Algorithm:
Shift Table:
0: 1
1: 2

Occurrences: 6820
Comparisons: 3070793
Running time: 26.6780 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
0: 1
1: 2

Good Suffix Table:
1: 7
2: 6
3: 6
4: 4
5: 4
6: 2
7: 2
8: 2

Occurrences: 6820
Comparisons: 1182800
Running time: 13.0700 ms

Bad Symbol Table

*	0	1
8	1	2

Good Symbol Table

k	Pattern	Shift
1	01010101	7
2	01010101	6
3	01010101	6
4	01010101	4
5	01010101	4
6	01010101	2
7	01010101	2
8	01010101	2

Output

11110101001111011110100101010111111100111110011100111111111011101000111100111001110101011010111100000010111011101101111011001011100101110110101010101010011001111111000011011101101111011011011011111011101101101111011011011011111011101101000000110111010110111100101111111101011101110011111110110111011011011111011101101000000110111010110111100101111111101011101110011111110110111011011011111011101101000000110111010110111100101111111101011101110011111110110111

5: 4
6: 4
7: 4
8: 4

Occurrences: 6892
Comparisons: 1593436
Running time: 16.2930 ms

Bad Symbol Table

*	0	1
8	2	1

Good Symbol Table

k	Pattern	Shift
1	00110011	1
2	00110011	7
3	00110011	7
4	00110011	4
5	00110011	4
6	00110011	4
7	00110011	4
8	00110011	4

Output

11110101001111011110100101010111111001111100111001111111110111010001111001110011101010110101111000000101110111011011110110010111001011101101010
10101001100110111110000110111011011110110101011111010010110100010000010001110100001011111101110110111011101110110100000110111010110111100101111
11101010111001111100110010110110111010001100000001110010110000011101111110011110011111100101001110001101010010011101111110010000101000011101110
01101111110110111111111110100010011011011011011111101111100110011110110111111101100011011110001111101011100101001001111110000011001110011110111100
01110011111101111100101111111010101010101011111101010110010011010110001110010111010001001111011011110110110011111010111111110001001100110111011
00101011010111101111101011100001100000110101110011111000110011101111111011111000001101111000111010100101110011111001101101111101010111111111100
01011000001010100101100101001111111111011000111111110110101100011101100111010001101001001010101000011011111001111111011100101011011111111000001
01010110100101001100011100010001101110111100111100110001101000000111100001100011011101010101110110000111010100110111011110100010011101111001000
01011011010111000111110101100110000101011010101101011100010100101111011000010110101110000001010111110111100011010110110111111100111111101110010
01001111000000000011101011100100101011100010010110010111010101010110000101010101100011000100100111000011101010001010000001111111110100110000010
11110011101010010101000101011101111111111111101110110111011111100110110111011111111110101100001001111100000001111110010101101010111110110110111
01110100011010111001011011001010111111100111000111111000100101100000010001001111011111011111000101101010110111110110101101111111101101100110101011

- 11001100

Brute-force Algorithm:
Occurrences: 6934
Comparisons: 4547718
Running time: 28.0560 ms

Horspool Algorithm:
Shift Table:
0: 1
1: 2
Occurrences: 6934
Comparisons: 2525525
Running time: 26.3900 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
0: 1
1: 2

Good Suffix Table:
1: 1
2: 7
3: 7
4: 4
5: 4
6: 4
7: 4
8: 4

Occurrences: 6934
Comparisons: 1677563
Running time: 18.6640 ms

Bad Symbol Table

*	0	1
8	1	2

Good Symbol Table

k	Pattern	Shift
1	11001100	1
2	11001100	7
3	11001100	7
4	11001100	4
5	11001100	4
6	11001100	4
7	11001100	4
8	11001100	4

Output

11110101001110111101001010101111111001111100111001111111110111010001111001110011101010110101111000000101110111011011110110010111001011101101010
10101001100111111000011011101101111101101010111110100101101000100000100011101000010111111011101101110111011110110100000110111010110111100101111
11101010111001111100110010110110111010001100000001110010110000011011111100111001111100101001110001101010010011101111110010001010000111011100
01101111110101111111111010001001101101101101111101111100110011111101101110111110100011011110001111101011001010010011111000001100111001111101111100
01110011111101111100101111111101010101010101011111101010110010011010110001110010111010001001111011011110110110011111010111111110001001100110111011
00101011010111101111101011100001100000110101110011111000110011101111111011111000001101111000111010100101110011111001101101111101010111111111100
01011000001010100101100101001111111110110001111111110110101100011101100111010001101001001010101000011011111001111111011100101011011111110000001
01010110100101001100011100010001101111011110011100110010011101000000111100001100011011101010101110110000111010100110110111101000100111011110010001
01011011010110001111110101100110001010110101101011011000101001011110110000101101011100000010101111101111000110101101101111111001111111101110010
01001111000000000011101011100100101011100010010110010111010101010110000101010101100011000100100111000011101010001010000001111111110100110000010
11110011101010010101000101011101111111111111101110110111101111110011011011110111111111110101100001001111100000001111110010101101010111110110110111
01110100011010111001011011001010111111110011100011111100010010110000001000100111101111011111000101101011011111101101011011111111101101100110101011

- Pattern (all ones): 11111111
  - 11111111

Brute-force Algorithm:
Occurrences: 32978
Comparisons: 5005570

Running time: 28.9000 ms

Horspool Algorithm:
Shift Table:
1: 1

Occurrences: 32978
Comparisons: 1713023
Running time: 12.6020 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
1: 1

Good Suffix Table:
1: 7
2: 6
3: 5
4: 4
5: 3
6: 2
7: 1
8: 1

Occurrences: 32978
Comparisons: 913300
Running time: 9.1700 ms

Bad Symbol Table

*	1
8	1

Good Symbol Table

k	Pattern	Shift
1	11111111	7
2	11111111	6
3	11111111	5
4	11111111	4
5	11111111	3
6	11111111	2
7	11111111	1
8	11111111	1

Output

11110101001111011110100101010111111001111100111001111111110111010001111001110011101010110101111000000101110111011011110110010111001011101101010
10101001100111111000011011101101111011010101111101001011010001000001000111010000101111110111101101110111011101101010000011011101011011110010111111
1110101011100111110011001011101101110100011000000011100101100000110111111001111001111110010100111000110101001001110111111001000101000011101111100
01101111110110111111111010001001101101101101111101111100110011101110111111101100011011110001111101011100101001001111100000011001110011111011111100
01110011111101111100101111111101010101010101111110101011001001101011000111001011101000100111101101111011011001111101011111111000100110011011101110111
00101011010111101111101011100001100000110101110011111000110011101111111011111000001101111000111010100101110011111001101101111101010111111111100
01011000001010100101100101001111111111011000111111110110101100011101100111010001101001001010101000011011111100111111101110010101101111111000001
01010110100101001100011100010001101110111001111001100011010000001111000011000110111010101011101100001110101001101101111010001001110111110010000
01011101101011100011111010110011000010101101010110101110001010011011110110000101101011100000010101111101111000110101101101111111001111111101110010
01001111000000000011101011100100101011100010010110010111010101010110000101010101100011000100100111000011101010001010000001111111111010011000010
11110011101010010101000101011101111111111111011101101111011111100110110111011111000000001111110010101101010111110110110111
01110100011010111001011011001010111111110011100011111100010010110000001000100111101111101111100010110101101111101101011011111110110110011010111

- Pattern (all zeros): 00000000
  - 00000000

Brute-force Algorithm:
Occurrences: 1466
Comparisons: 3426330
Running time: 24.8830 ms

Horspool Algorithm:
Shift Table:
0: 1

Occurrences: 1466
Comparisons: 767668
Running time: 7.4110 ms

Boyer-Moore Algorithm:
Bad Symbol Table:
0: 1

Good Suffix Table:
1: 7
2: 6
3: 5
4: 4
5: 3

6: 2
7: 1
8: 1

Occurrences: 1466
Comparisons: 473424
Running time: 5.9160 ms

Bad Symbol Table

*	0
8	1

Good Symbol Table

k	Pattern	Shift
1	00000000	7
2	00000000	6
3	00000000	5
4	00000000	4
5	00000000	3
6	00000000	2
7	00000000	1
8	00000000	1

Output

```
1111010100111101111010010101011111110011111001110011111111101110100011110011100111010101101011110000001011101110110111101100101110010111011010101
1010100110011111100001101110110111101101010111110100101101000100000100011101000010111111011101101110111101110110100000110111010110111100101111
1110101011100111110011001011011011101000110000000111001011000001101111110011110011111100101001110001101010010011101111110010001010000111011110
0110111111011011111111110100010011011011011011111011111001100111101110111111011000110111100011111010111001010010011111000001100111001111101111100
0111001111110111110010111111101010101010101111110101011001001101011000111001011101000100111101101111011011001111101011111111000100110011011101110111
0010101101011110111110101110000110000011010111001111000110011101111110111110000011011110001110101001011100111110011011011111010101111111111100
0101100000010101001011001010011111111110110001111111101101011000111011001110100011010010010101010000110111111001111111011100101011011111111000001
0101011010010100110001100010001101110111100111100110001101000000111100001100011011101010101110110000111010100110110111101000100111011111001000
0101101101011100011111010110011000010101101010110101110001010010111101100001011010111000000101011111011110001101011011011111110011111110111001
0100111100000000011101011100100101011100010010110010111010101010110000101010101100011000100100111000011101010010100000011111111110100110000010
1111001110101001010100010101110111111111111110111011011110111111100110110111011111111110101100001001111100000001111110010101101010111110110110111
0111010001101011100101101100101011111111001110001111110001001011000000100010011110111101111100010110101101111101101011011111111011011001101011
```