**CS5800 Project Final Report**

**Enhancing On-Campus Job Opportunities: Addressing Struggles and Empowering Students**

**Project team members**: Aravind Venkatapathy, Pramatha Bhat, Harshitha Prabhu, Bhavya Maheshwari

**Presentation Link:**

**Introduction:**

In the pursuit of higher education, on-campus jobs play a pivotal role for students, providing not only financial support but also invaluable work experience and personal growth. However, the process of finding on-campus employment can be a formidable challenge, hampering students' access to these opportunities and hindering their overall development.

One of the primary struggles that students encounter is the limited availability of on-campus jobs. With a finite number of positions, competition among students becomes intense, making it difficult for many to secure employment that suits their interests and skills. Additionally, there is often a lack of awareness among students about the various on-campus job opportunities, their application procedures, and where to access relevant information. As a result, a significant number of students remain unaware of potential job openings.

In our project, we attempt to compare the performance and efficiency of various pattern matching algorithms such as the Naïve method, Boyer Moore algorithm, Rabin Karp algorithm and Knuth–Morris–Pratt algorithm (KMP) with various test cases where in the length of the strings(Job descriptions) varies with the job search keywords that are used against it. We aim to determine which algorithm is best suited for our use case by comparing the number of comparisons made between the keywords of a job search and the job description. The number of comparisons would help determine the efficiency of each algorithm for this use case.

**View:**

**View of Aravind:**

On-campus job search has been vital for the students studying at Northeastern both in terms of monetary aspects and gaining valuable experience from the job that you are involved in. Searching for on-campus jobs can get quite taxing at times for students due to the number of jobs that they would have to sift through to find the relevant ones. On-campus job searches often involve various filters such as job type, location, industry, or work hours. Pattern searching algorithms can be extended to incorporate these filters, making the search process more personalised and tailored to the preferences of individual students. By optimising search results using pattern searching algorithms, the overall user experience of the job search platform is improved. Students can quickly find the relevant job opportunities without having to sift through a large number of unrelated listings. I believe this topic would help address the issues faced by my fellow students and hence it is relevant to me.

**View of Harshitha:**

The quest for on-campus employment is of utmost importance for Northeastern students, not only for financial gains but also for invaluable experiential learning. However, sifting through numerous job listings can become burdensome and time-consuming. Filtering through job types, locations, industries, and work hours adds complexity to the process. To alleviate these challenges, incorporating pattern searching algorithms could prove beneficial, tailoring the job search to individual preferences. By optimizing the search results, the overall user experience of the job search platform is enhanced, allowing students to swiftly identify relevant opportunities without being overwhelmed by unrelated listings. Personally, I find this topic highly relevant as it directly addresses the issues faced by my fellow students.

**View of Pramatha:**

For Northeastern students, securing on-campus employment is of utmost importance, as it not only brings financial gains but also offers invaluable experiential learning prospects. However, the process of sifting through a large number of job listings can become burdensome and time- consuming. Moreover, filtering through job types, locations, industries, and work hours adds further complexity to the job search. To mitigate these challenges, the incorporation of pattern searching algorithms presents a promising solution, enabling the tailoring of job searches to individual preferences. The optimization of search results through these algorithms enhances the overall user experience of the job search platform, allowing students to quickly find relevant opportunities while avoiding overwhelming and irrelevant listings. Personally, I am deeply invested in this topic as it directly addresses the challenges faced by my peers during their job searches.

**View of Bhavya:**

The pursuit of on-campus employment holds great significance for students at Northeastern, providing not only financial benefits but also invaluable experiential learning opportunities. Nevertheless, the task of sifting through numerous job listings can be arduous and time-

consuming. Adding to the complexity, students must filter through job types, locations, industries, and work hours. To tackle these challenges, the integration of pattern searching algorithms offers a promising solution, customizing the job search according to individual preferences. Through the optimization of search results, the overall user experience of the job search platform is elevated, enabling students to swiftly discover pertinent opportunities while avoiding overwhelming amounts of irrelevant listings. Personally, I believe this topic is highly pertinent as it directly addresses the issues encountered by my fellow students.

**Defined Question/Approach:**

This project is to provide an analysis on the various pattern matching algorithms that can be used in a job searching portal. The idea is to determine the efficiency and performance of each algorithm between job descriptions on the web and the possible keywords that students might use for a job search. We decided that number of comparisons between the job search keywords (pattern) and the job description (text) would provide an accurate estimate as to how each algorithm would perform in a real world scenario of a job portal. The goal is for the portal to minimise the number of comparisons made between a job search and a job description while also delivering accurate results with the relevant keywords.

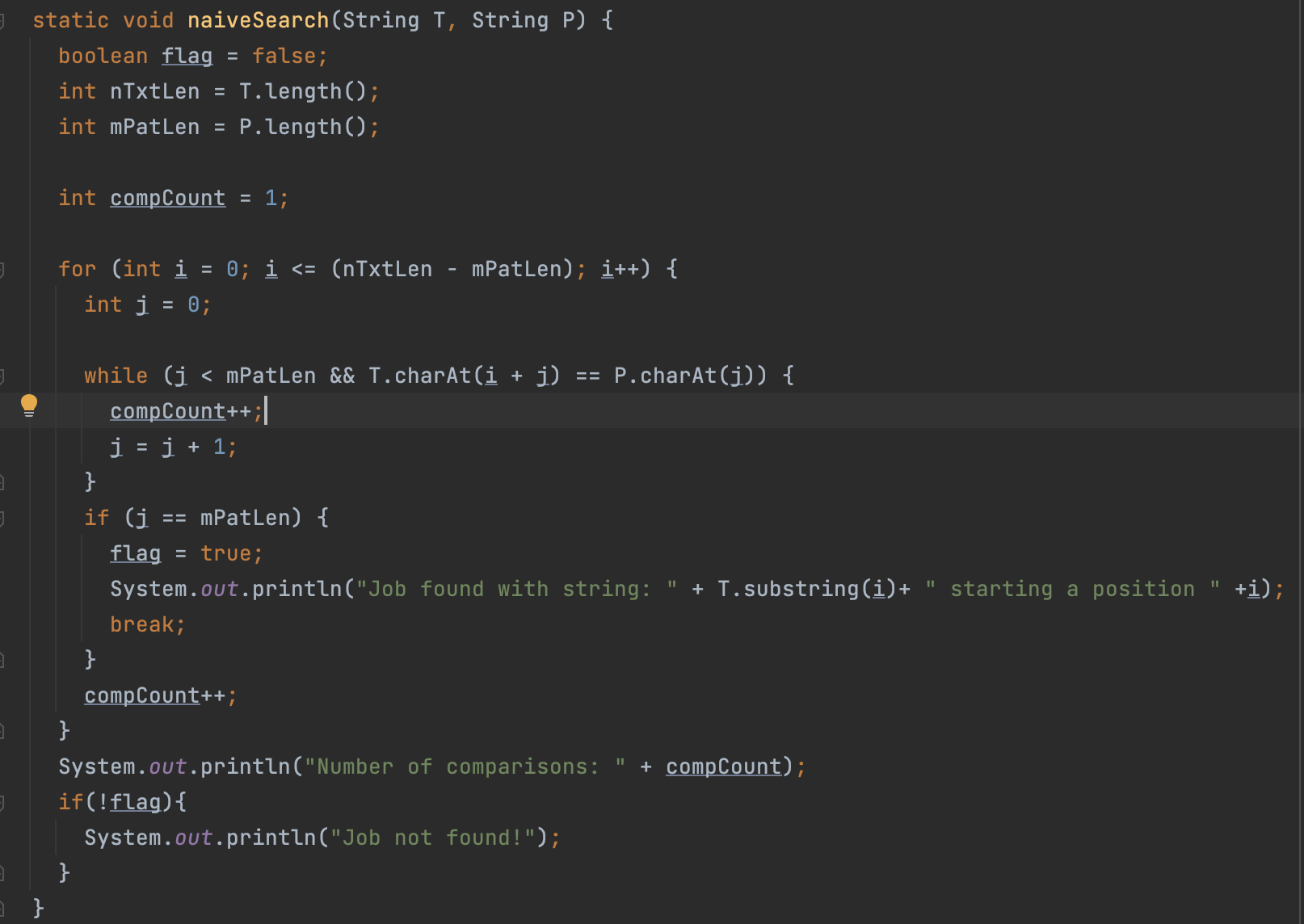
**Algorithm Analysis:**

**Naïve String Matching algorithm:**

The Naive String Matching algorithm is a simple and straightforward approach to finding occurrences of a pattern within a text. It works by comparing each character of the pattern with each character of the text, one by one. If a mismatch occurs, it moves to the next character in the text and restarts the comparison. The algorithm steps are as follows:

1. Let n be the length of the text and m be the length of the pattern.
2. Start with the first character of the text and the first character of the pattern.
3. Compare each character of the pattern with the corresponding character in the text. If they match, continue the comparison for the next characters.
4. If a mismatch occurs at any position, move to the next character in the text (i.e., shift the pattern one position to the right in the text).
5. Repeat steps 3 and 4 until either: a. The pattern is found within the text (i.e., all characters of the pattern match the corresponding characters in the text), and record the starting position of the match. b. The end of the text is reached, and no match is found.
6. If a match is found, return the starting position of the match in the text.
7. If no match is found, the pattern is not present in the text.

**The Naive String Matching algorithm has a time complexity of O((n-m+1)m).** In the worst case, it can take O(nm) time, where n is the length of the text and m is the length of the pattern. It is a simple algorithm and may not be the most efficient for large texts and patterns, but it serves as a basic foundation for more advanced string matching algorithms like Rabin-Karp, Knuth-Morris-Pratt (KMP), and Boyer-Moore.



**Rabin-Karp algorithm:**

The Rabin-Karp algorithm uses hashing to efficiently find occurrences of a pattern within a text. It computes the hash values of the pattern and substrings of the text and compares them. The algorithm steps are as follows:

1. Calculate the hash value of the pattern and the first substring of the text of the same length as the pattern using a hash function. A common hash function is to treat each character as a number and calculate the hash as a polynomial sum of these numbers.

2. Compare the hash value of the pattern with the hash value of the substring in the text. If they are equal, perform a character-by-character comparison to check for a spurious hit.

3. If the hash values are not equal, slide the pattern one position to the right and calculate the hash value of the new substring in the text.

4. Repeat steps 2 and 3 until either:

a. A match is found (hash values are equal, and the character-by-character comparison confirms the match), and record the starting position of the match in the text.

b. The end of the text is reached, and no match is found.

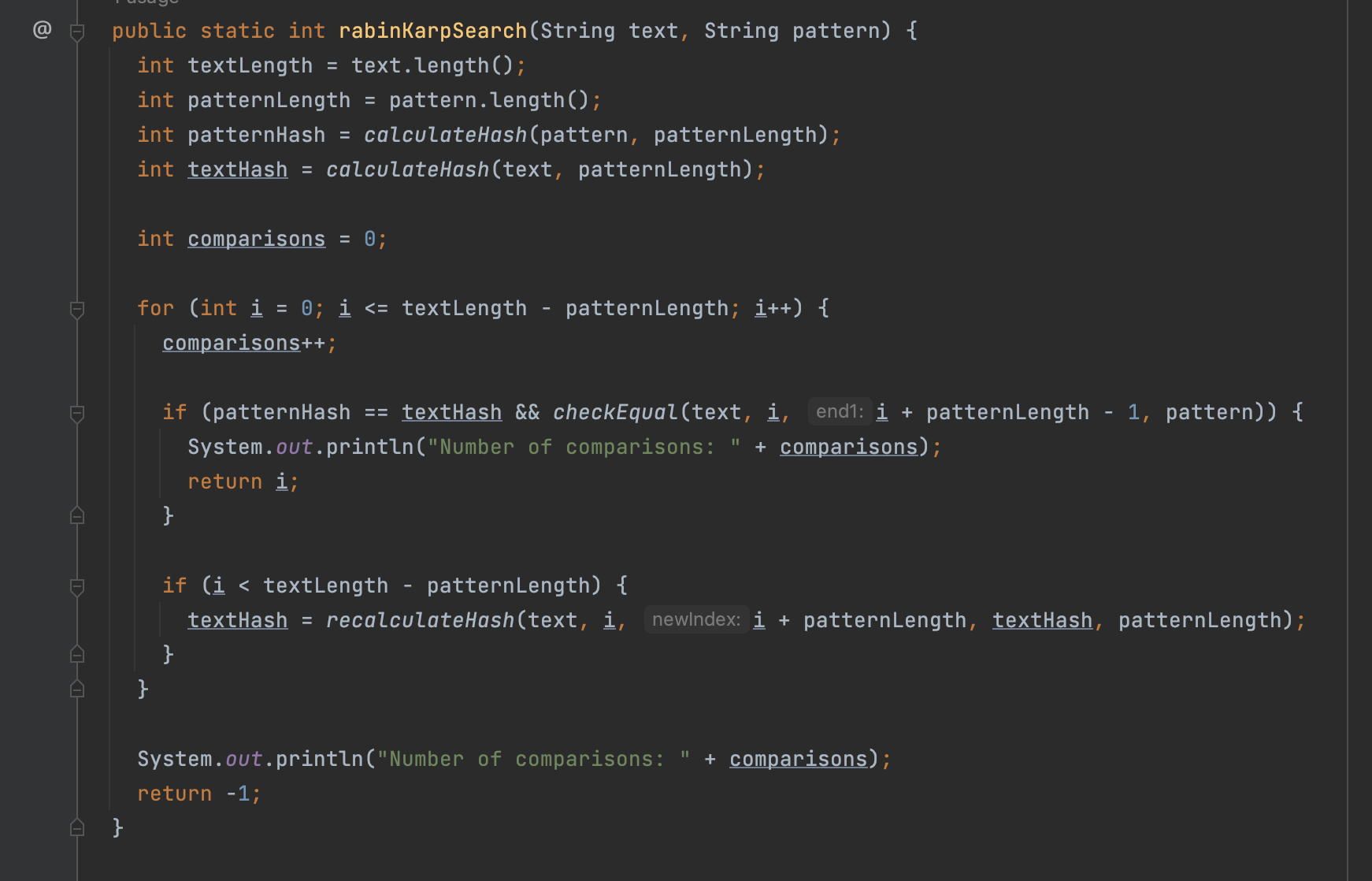
5. If a match is found, return the starting position of the match in the text.

6. If no match is found, the pattern is not present in the text.

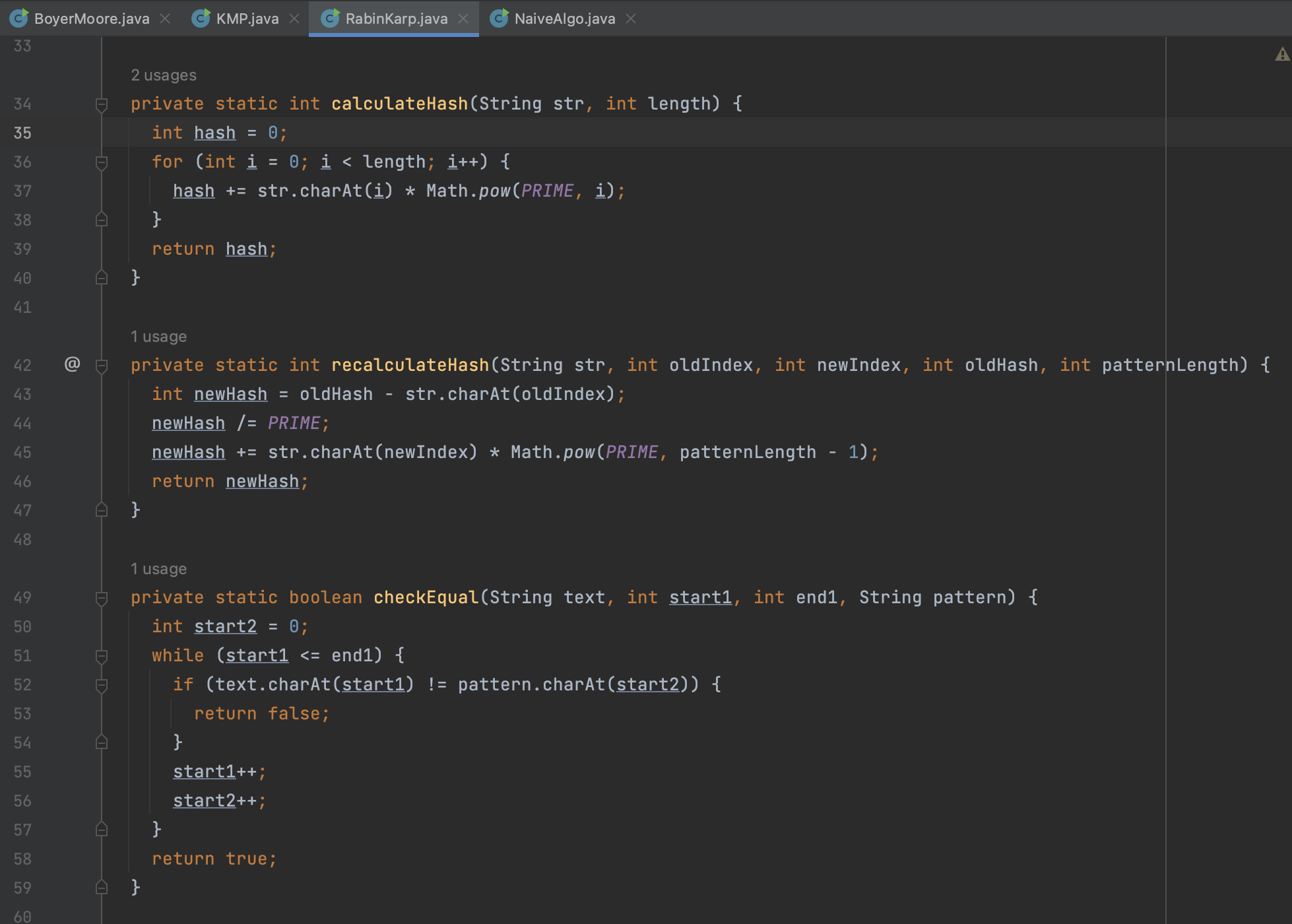
Rabin-Karp uses rolling hash to efficiently calculate the hash value of the next substring in the text by subtracting the contribution of the first character of the previous substring and adding the contribution of the new character at the end of the substring.

It's important to choose a good hash function to avoid collisions and minimize the chances of spurious hits. Moreover, to avoid false positives in the hash comparison, the algorithm must verify the match by performing a character-by-character comparison when hash values are equal. The Rabin-Karp algorithm has an average time complexity of O(n + m) in the best and average cases, where `n` is the length of the text, and `m` is the length of the pattern. However, in the worst case, it can take O(n\*m) time due to hash collisions, similar to the Naive String Matching algorithm. Still, Rabin-Karp can be more efficient in practice when the hash function and hash comparison are carefully implemented.

Rabin Karp Search function:



Other functions including hash functions and checking if the pattern and substrings of text are equal:



The calculateHash method calculates the hash value of a string, the recalculateHash method recalculates the hash after a shift, and the checkEqual method checks if two substrings are equal.

**Knuth-Morris-Pratt (KMP) algorithm:**

The Knuth-Morris-Pratt (KMP) algorithm efficiently finds occurrences of a pattern within a text without unnecessary backtracking. It preprocesses the pattern to create a partial match table (also known as the "failure function" or "prefix function") that helps in skipping unnecessary character comparisons during the search. The algorithm steps are as follows:

1. Preprocess the pattern:
   1. Initialize two pointers, i and j, both starting at 0.
   2. Create a partial match table (also called "failure function" or "prefix function") lps[] of size equal to the length of the pattern.
   3. Set lps[0] = 0.
2. While i < pattern length:
   1. If pattern[i] matches pattern[j], set lps[i] = j + 1 and increment both i and j.
   2. If pattern[i] does not match pattern[j], check if j is greater than 0:
      * If yes, set j = lps[j-1].
      * If no, set lps[i] = 0 and increment i.
3. Once the partial match table (lps[]) is constructed, start the pattern matching in the text using the following steps:
   1. Initialize two pointers, i and j, both starting at 0.
   2. While i < text length:

b.1 If pattern[j] matches text[i], increment both i and j.

b.2 If j reaches the length of the pattern, a match is found at i - j. Record the match and set j = lps[j-1].

b.3 If pattern[j] does not match text[i], check if j is greater than 0:

* + - * If yes, set j = lps[j-1].
      * If no, increment i.

1. Repeat step 3 until either:

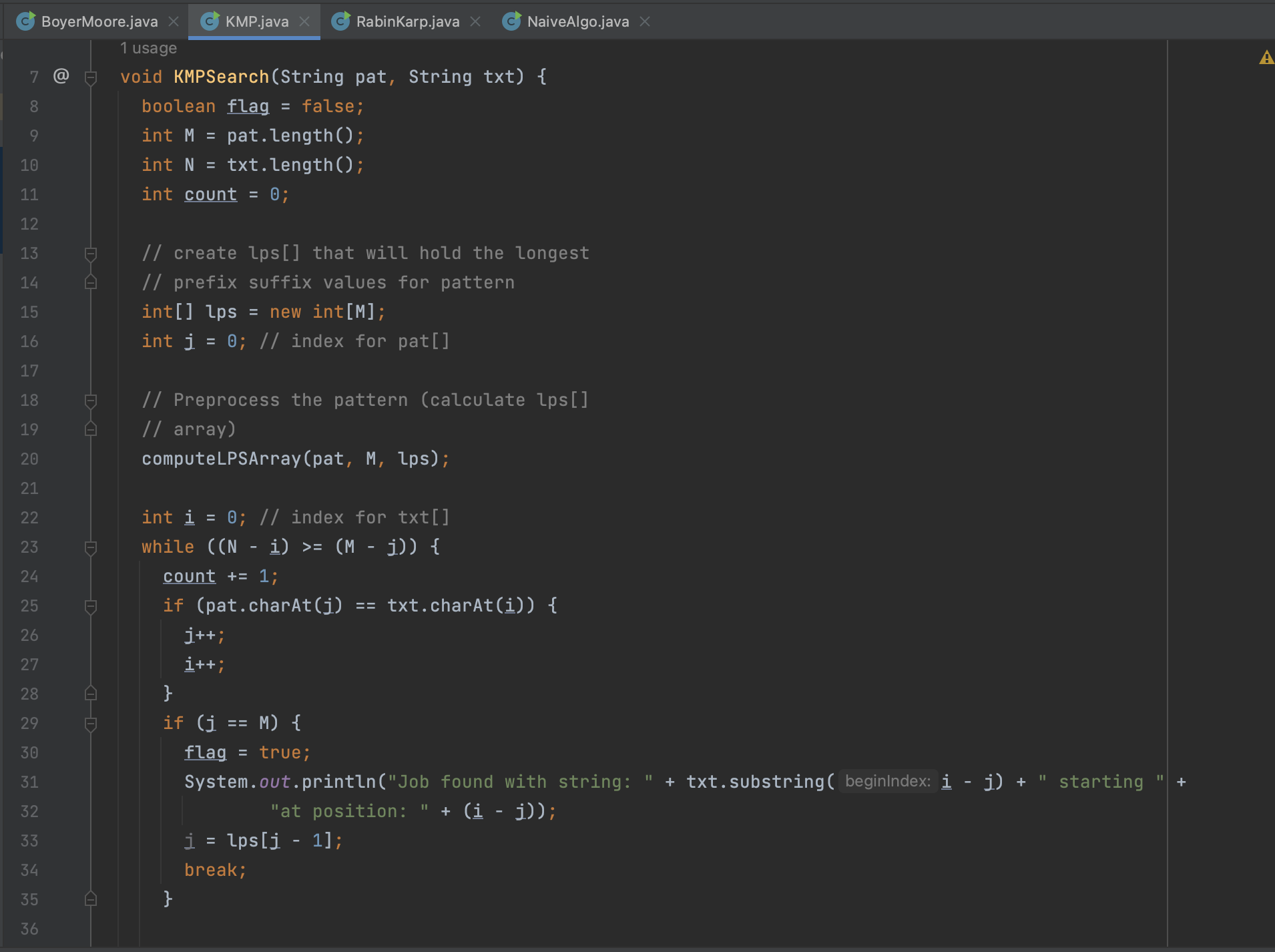
a. A match is found (pattern is found in the text), and continue the search for further matches.

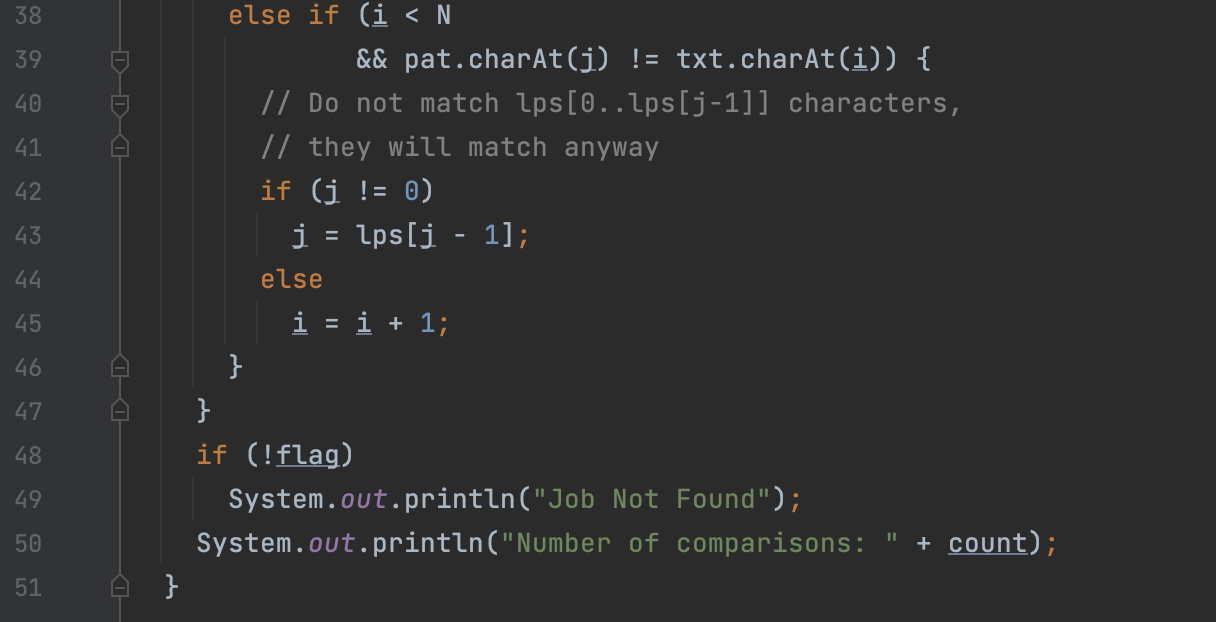
b. The end of the text is reached, and no more matches are found.

This algorithm efficiently skips unnecessary comparisons by utilizing the information stored in the lps[] array, which indicates the length of the longest proper suffix of the pattern that is also a prefix. This eliminates the need to backtrack when a mismatch occurs, making the algorithm more efficient than the Naive String Matching algorithm.

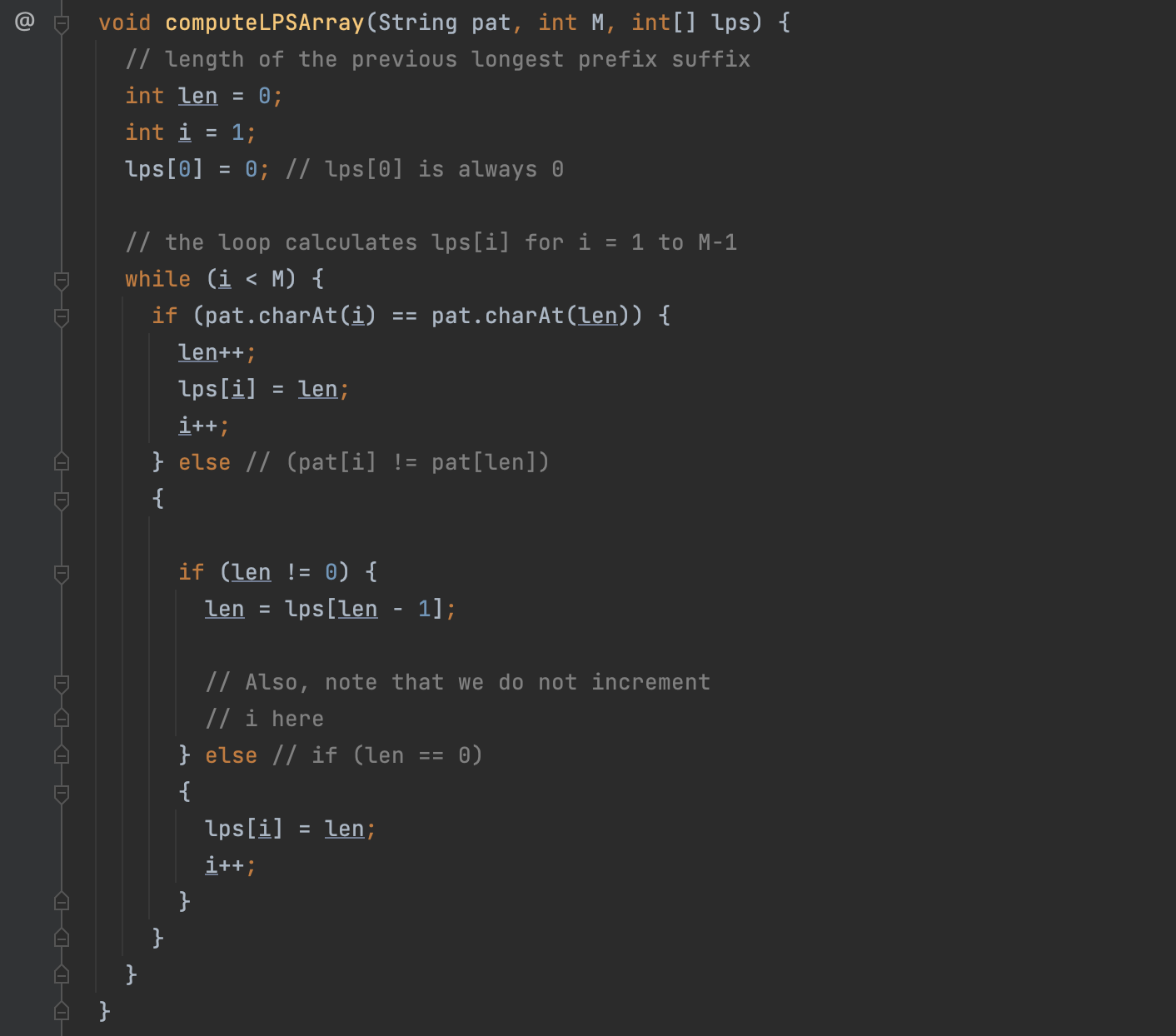
The time complexity of the Knuth-Morris-Pratt algorithm is O(n + m), where n is the length of the text, and m is the length of the pattern. The construction of the lps[] array takes O(m) time, and the pattern matching phase takes O(n) time. Thus, the overall time complexity is linear in the size of the input.

KMP search function



Top of Form

Function to compute the prefix array:



**Boyer-Moore-Horspool algorithm:**

We decided to go with the simplified version of Boyer-Moore algorithm as it is a simplified version of the Boyer-Moore algorithm using a single table (Shift Table). This algorithm preprocesses the pattern to generate a Shift Table to determine the correct mount to shift the pattern when a mismatch occurs. This shift decision is based on the input text character c’s alignment with the last character in the pattern. The algorithm steps are as follows:

shiftTable function:

1. Initialize a table of size 500 with all elements set to the length of the pattern (m).
2. Convert the pattern string (pattern) to a character array (P).
3. Iterate through each character P[j] in the pattern from index 0 to m - 2 (inclusive).
4. Set the table value at index P[j] to m - 1 - j. This step sets the rightmost occurrence of each character in the pattern in the table.

horspoolMatching function:

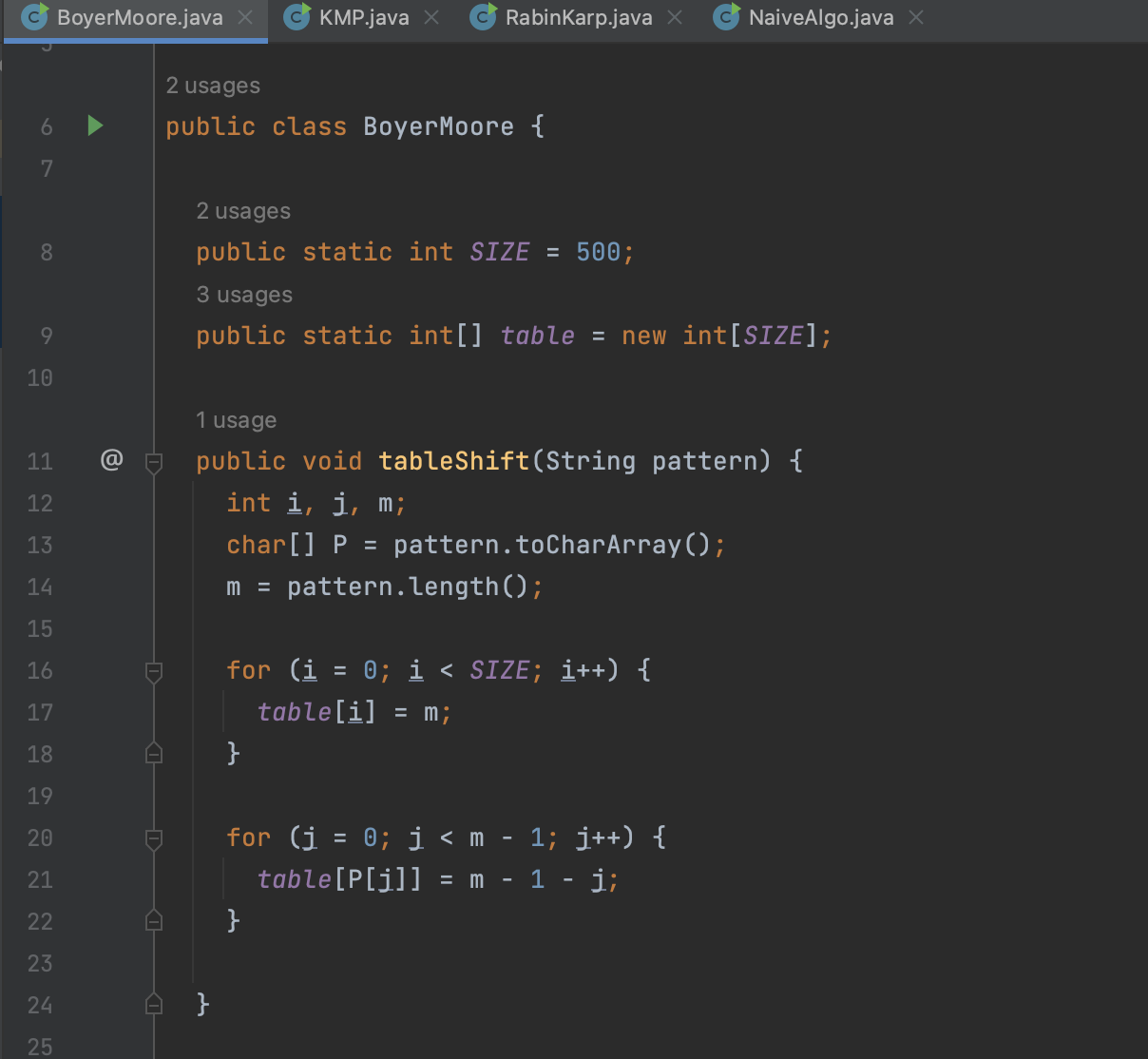
* 1. Convert the pattern string (pattern) and text string (text) to character arrays (P and T respectively).
  2. Get the lengths of the pattern and text and store them in variables m and n.
  3. Initialize a variable compCount to 0 to count the number of character comparisons made during the search.

Main Pattern Matching Loop:

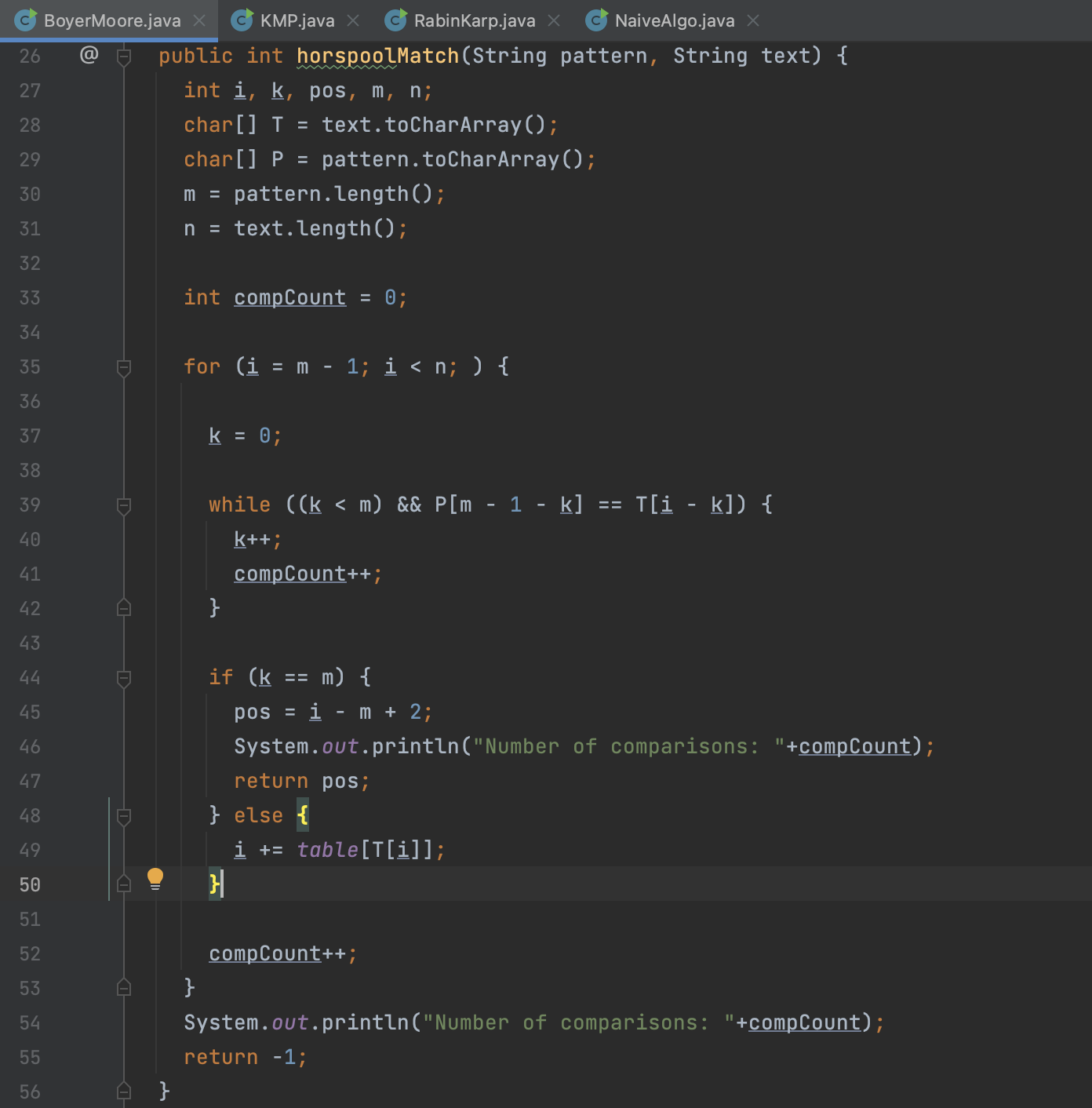
* 1. Start the loop with i set to m - 1. The variable i will be used to scan the text from right to left.
  2. Inside the loop, set k to 0, which will be used to compare characters of the pattern and the text.
  3. Compare characters of the pattern and text from right to left while k < m and P[m - 1 - k] matches T[i - k].
     + If there is a match, increment k and compCount to count the comparison.
  4. If k reaches m, it means a match is found. Calculate the position of the match (pos) as i - m + 2.
  5. Print the number of comparisons made (compCount) and return the position of the match (pos).
  6. If a match is not found, set i to i + table[T[i]], where table[T[i]] gives the shift value based on the last character in the current window of text. This allows the algorithm to efficiently skip unnecessary comparisons.
  7. Increment compCount to count the comparison for the shift operation.
  8. Repeat the loop until i reaches n (end of the text) or a match is found.
  9. If no match is found, print the number of comparisons made (compCount) and return -1.

The Boyer-Moore-Horspool algorithm uses the table calculated in the shiftTable function to perform efficient shifts and minimize the number of character comparisons, making it faster than the naive string matching algorithm for certain types of patterns and texts.

Function to generate the shift table:



Function to check the comparisons:



**Simulation and Runtime analysis:**

We had taken three test cases with varying lengths of job descriptions and job search keywords for comparisons and the results are analysed for their number of comparisons. The input of the text was done through a file where we parsed through the file to get all the strings present in the file.

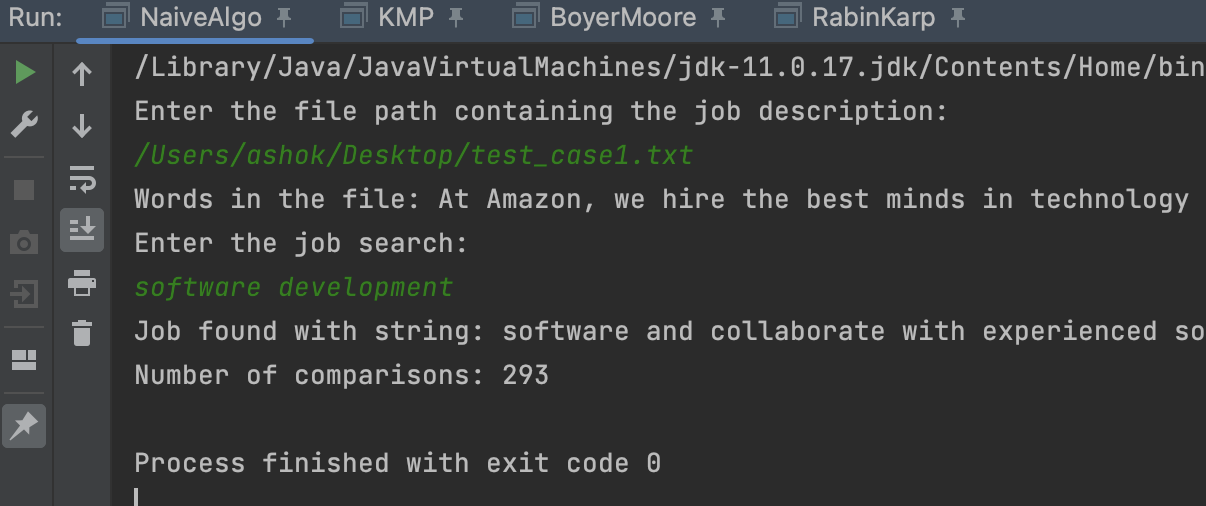
**Test case 1:**

Job Description (Text) length: 661

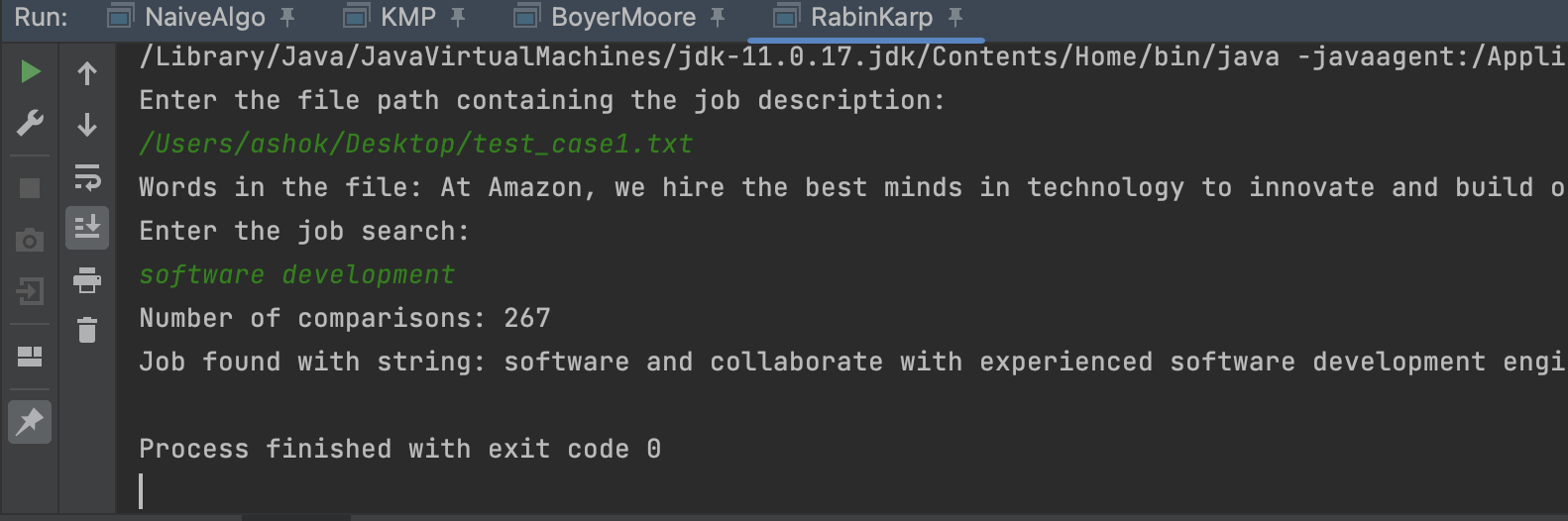
Job Search Keywords (Pattern) length: 2

Is the keyword present in the text?: Yes

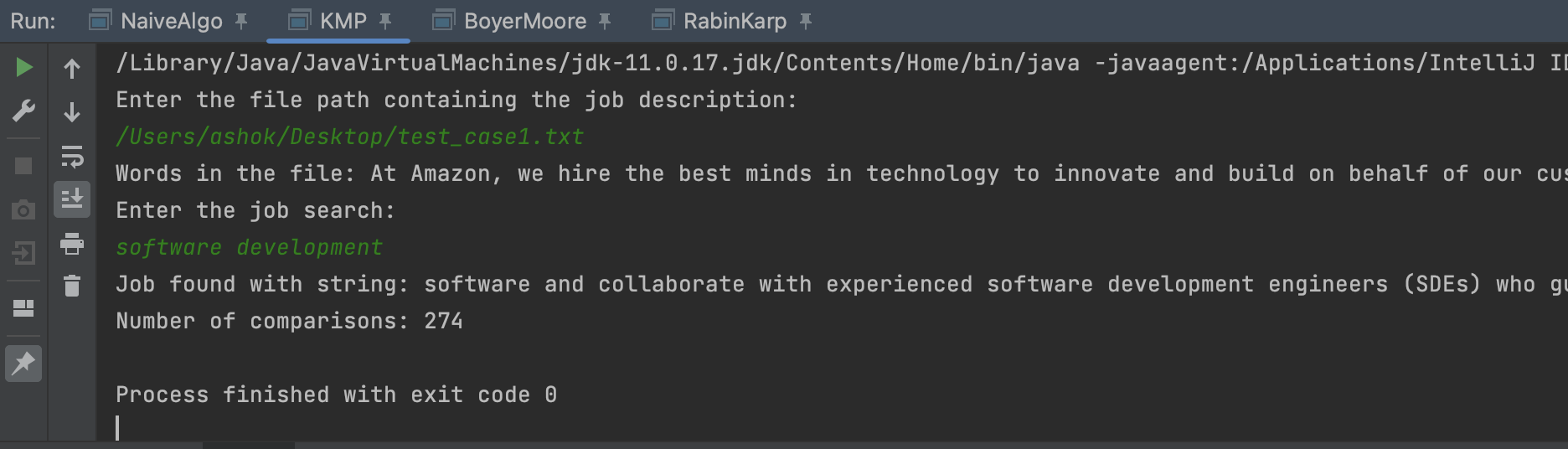
Naïve algorithm: The number of comparisons are 293



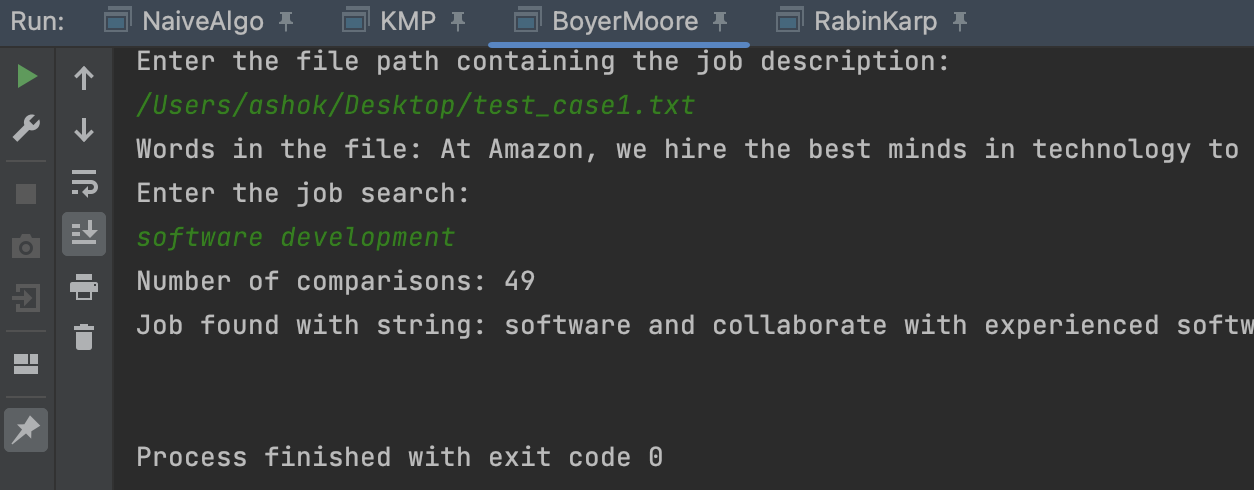
Rabin Karp algorithm: The number of comparisons are 267



KMP algorithm: The number of comparisons are 274



Boyer-Moore-Horspool algorithm: The number of comparisons are 49



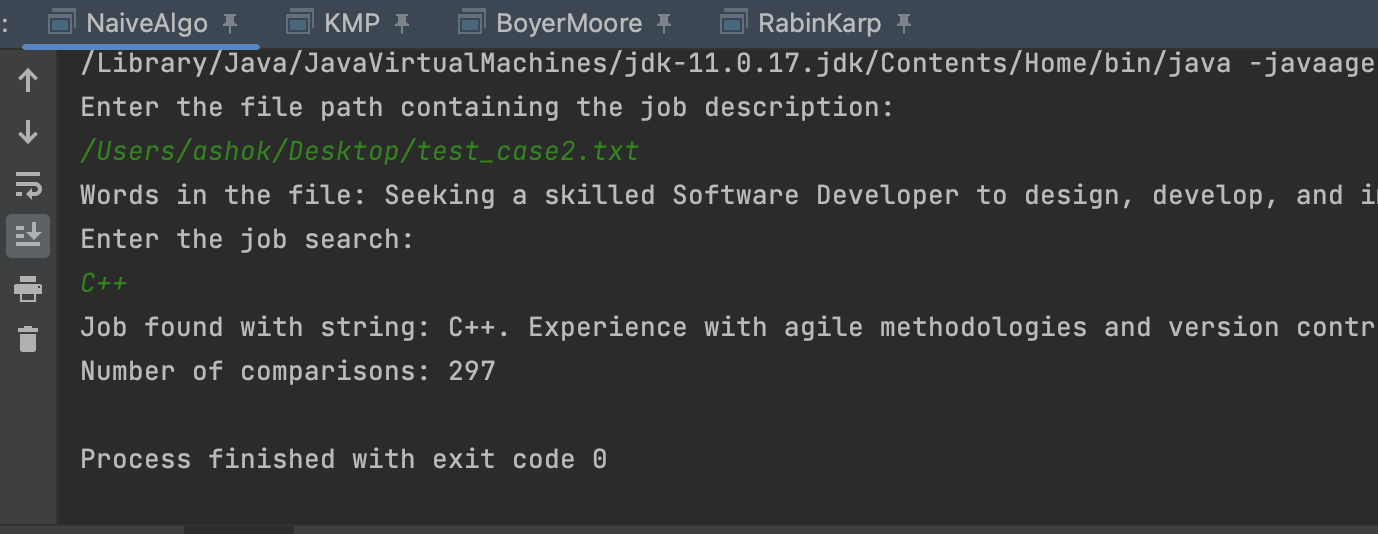
**Test case 2:**

Job Description (Text) length: 51

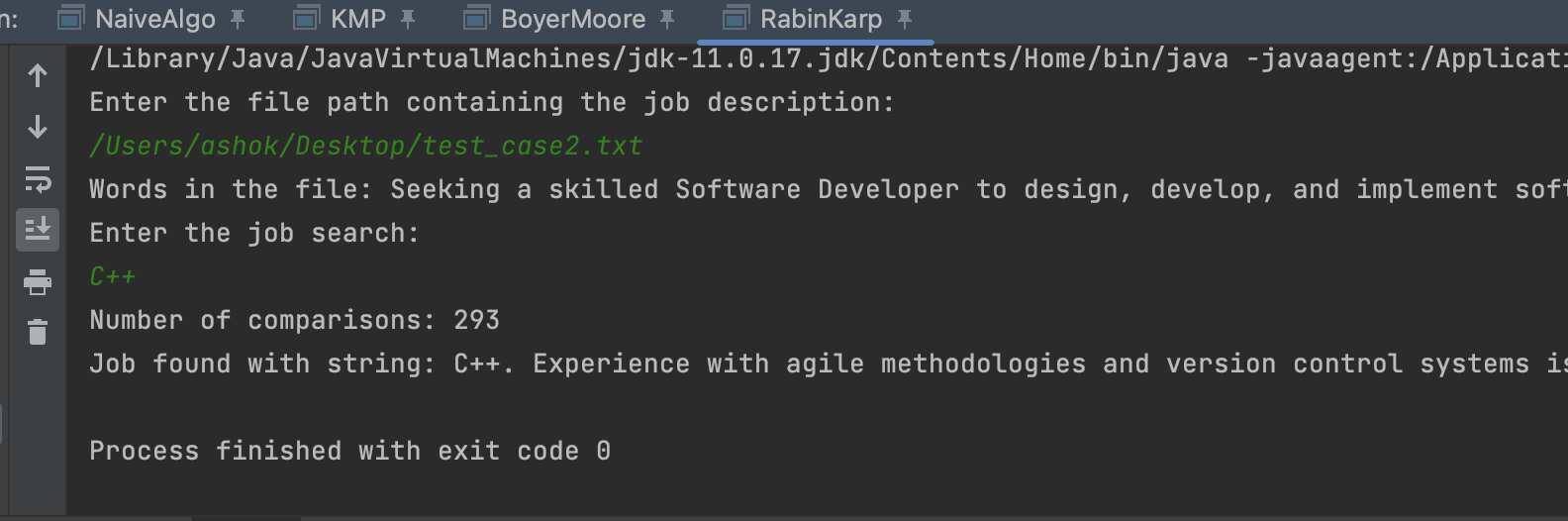
Job Search Keywords (Pattern) length: 1

Is the keyword present in the text?: Yes

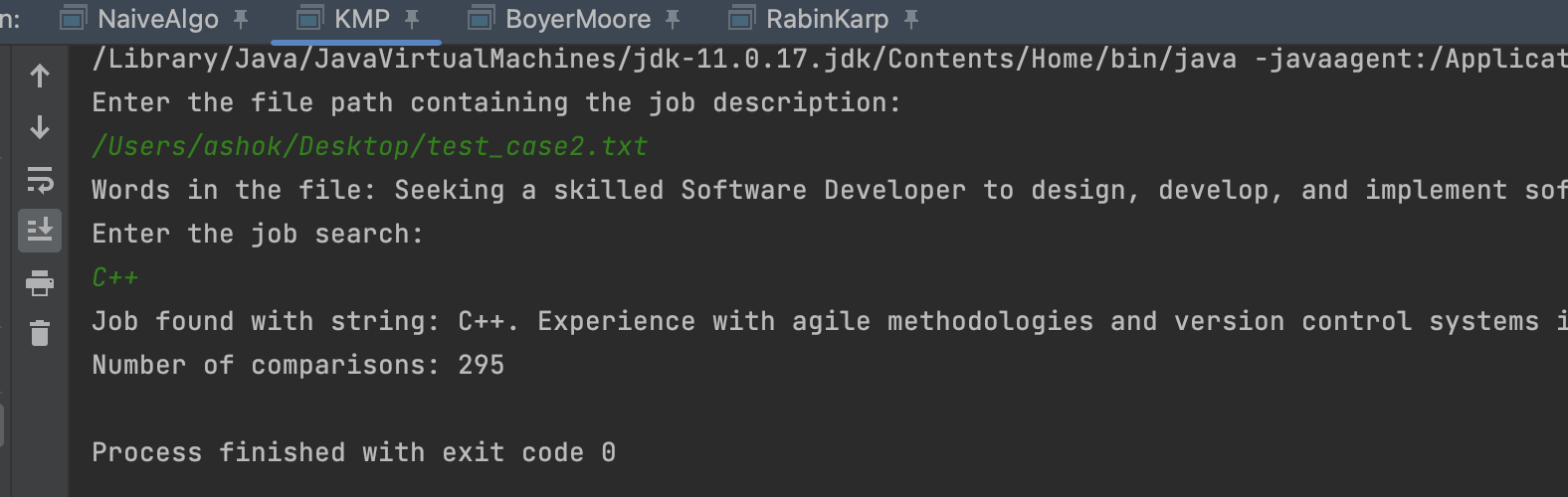
Naïve algorithm: The number of comparisons are 297



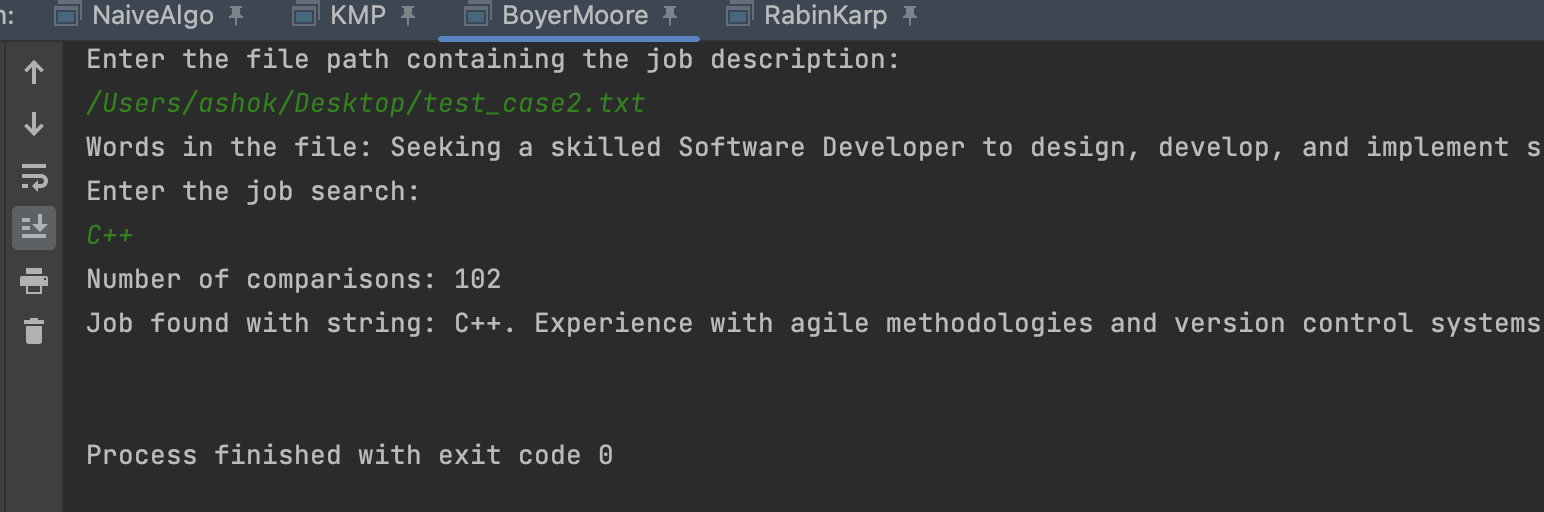
Rabin Karp algorithm: The number of comparisons are 293



KMP algorithm: The number of comparisons are 295



Boyer-Moore-Horspool algorithm: The number of comparisons are 102



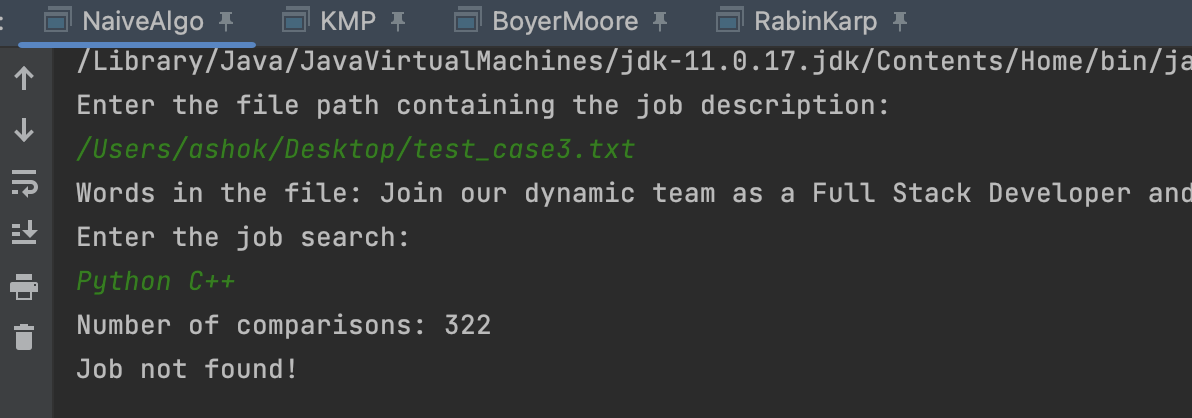
**Test case 3:**

Job Description (Text) length: 46

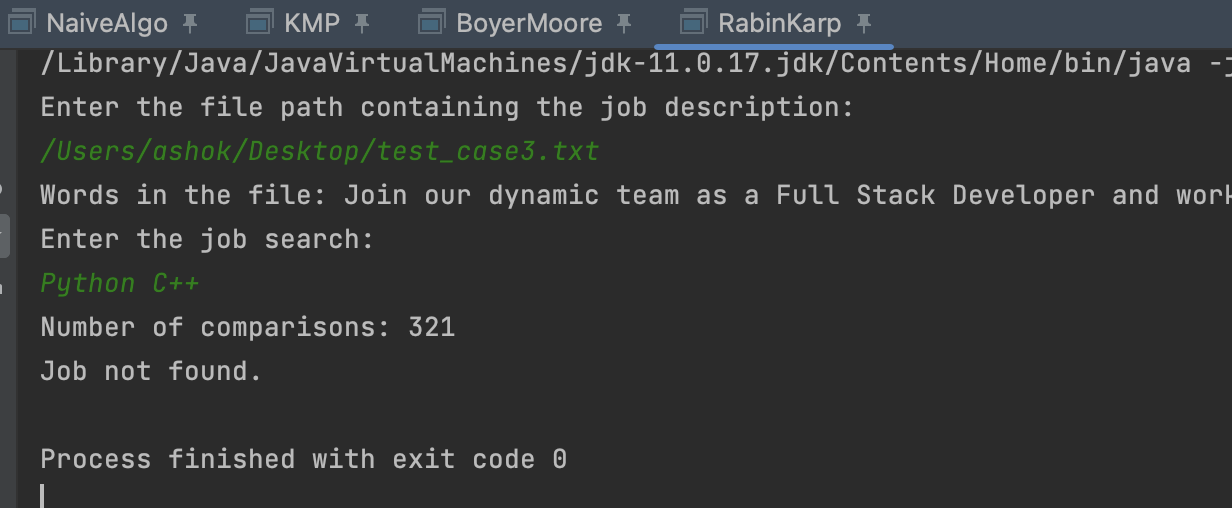
Job Search Keywords (Pattern) length: 2

Is the keyword present in the text?: No

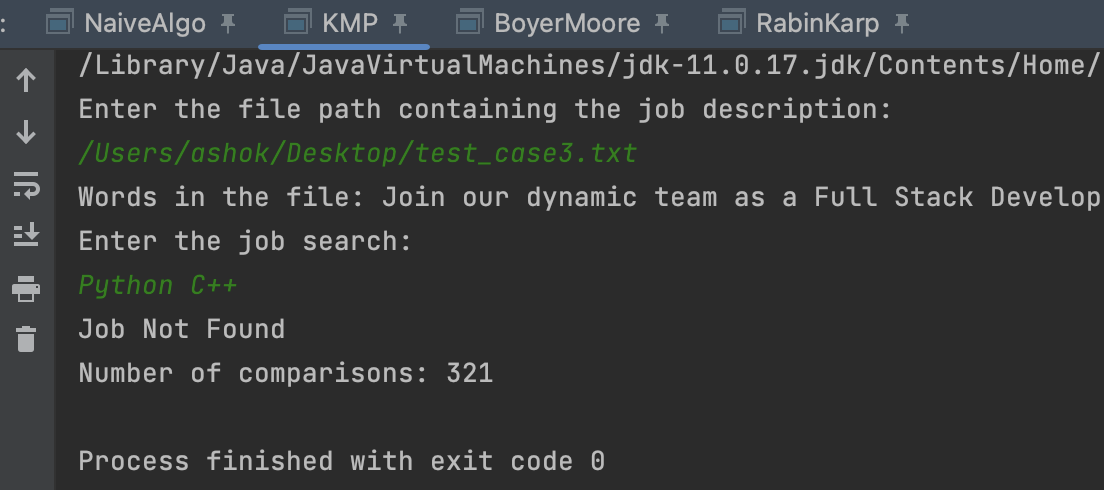
Naïve algorithm: The number of comparisons are 322



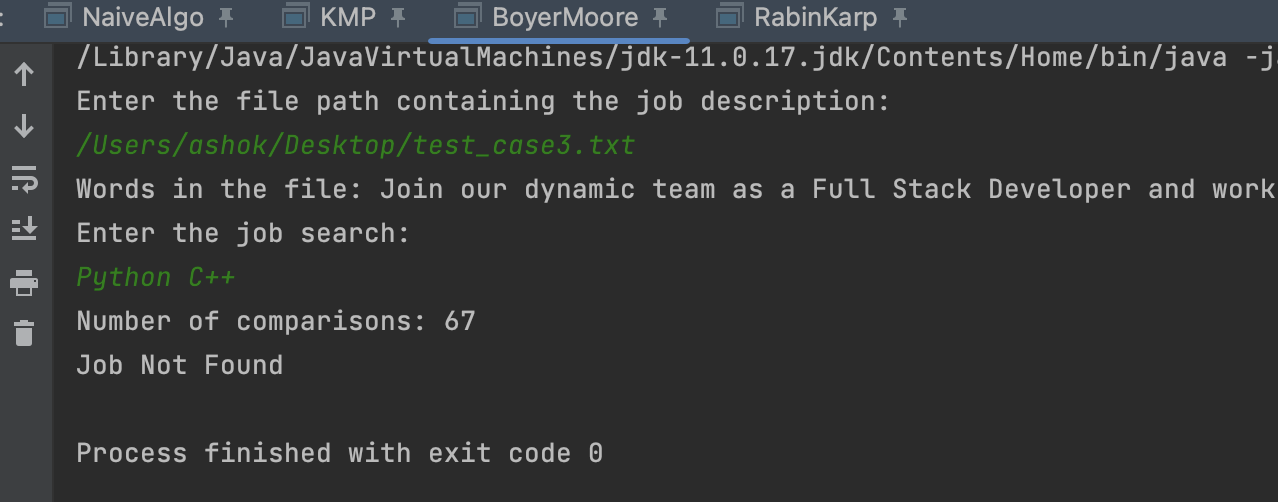
Rabin Karp algorithm: The number of comparisons are 321



KMP algorithm: The number of comparisons are 321



Boyer-Moore-Horspool algorithm: The number of comparisons are 67



Number of comparisons table:

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm/Test Case | 1 | 2 | 3 |
| Naive | 293 | 297 | 322 |
| Rabin-Karp | 267 | 293 | 321 |
| KMP | 274 | 295 | 321 |
| Boyer-Moore-Horspool | 49 | 102 | 67 |

**Conclusion:**

Based on our test runs, the observations made are:

1. Boyer-Moore-Horspool algorithm tends to generate the least number of comparisons whilst generating the same keyword in the result amongst all the algorithms for all the test cases. Also, the same algorithm generates the least number of comparisons even when the keyword is not present in the text.
2. The lengths of the text and the pattern matter as the order of the number of comparisons change for the other algorithms apart from Boyer-Moore-Horspool as seen in test cases 2 and 3.
3. By the better performance of Boyer-Moore-Horspool, we can conclude that it is the most efficient algorithm for our use case of job search which involves a sizable content to compare the pattern with as it performs character comparisons between a character in the text and a character in the pattern database from right to left.
4. **In the best and average cases, Boyer-Moore-Horspool has linear time complexity, making it particularly fast for patterns with unique characters or patterns that do not have many occurrences of the same character.**
5. It is also to be noted that Boyer-Moore-Horspool is determined to be fastest for this use case while an another algorithm could be more efficient for other use cases as it depends on various factors like longer patterns, large alphabets and average-case performance.
6. We believe with more time, we would have been able to test with more permutations of the text and pattern. Also, we would have been able to collect more datapoints perhaps from a dataset and created a front end application as well for better visualization.

In conclusion, we are happy with the effort of our team members and the result of our project.

**Takeaway:**

**Aravind:** I am really happy that I got to work on this project as I always wanted to do something out of the box and hence proceeded to collaborate with my team on researching more about the various pattern matching algorithms and their applications. Adding to that, I am delighted that I got the chance to work on this project with my team members who were instrumental in the successful completion of the project. This project helped me understand the importance of all the pattern/string matching algorithms and their advantages and disadvantages. Furthermore, I was happy that I was able to code in Java for this project which is my primary language. Each one of my team members had their own perspective about this course and the project topic and this helped solve each other’s problems that he/she faced during this course and the project. I was able to play around with algorithms that are used in the applications used by each student in Northeastern daily, and that gave me a sense of satisfaction.

**Pramatha:**

**Harshitha:**

**Bhavya:**

**Appendix:**

**Naïve Method:**

import java.io.BufferedReader;  
import java.io.FileReader;  
import java.io.IOException;  
import java.util.Scanner;  
  
public class NaiveAlgo {  
  
 static void naiveSearch(String T, String P) {  
 boolean flag = false;  
 int nTxtLen = T.length();  
 int mPatLen = P.length();  
  
 int compCount = 1;  
  
 for (int i = 0; i <= (nTxtLen - mPatLen); i++) {  
 int j = 0;  
  
 while (j < mPatLen && T.charAt(i + j) == P.charAt(j)) {  
 compCount++;  
 j = j + 1;  
 }  
 if (j == mPatLen) {  
 flag = true;  
 System.*out*.println("Job found with string: " + T.substring(i)+ " starting a position " +i);  
 break;  
 }  
 compCount++;  
 }  
 System.*out*.println("Number of comparisons: " + compCount);  
 if(!flag){  
 System.*out*.println("Job not found!");  
 }  
 }  
 public static void main(String[] args) {  
  
 Scanner sc = new Scanner(System.*in*);  
 System.*out*.println("Enter the file path containing the job description:");  
 String filePath = sc.next();  
 String T = "";  
 try {  
 String fileContent = *readAndStoreWords*(filePath);  
 System.*out*.println("Words in the file: " + fileContent);  
 T = fileContent;  
 } catch (IOException e) {  
 System.*err*.println("Error reading the file: " + e.getMessage());  
 }  
 System.*out*.println("Enter the job search:");  
 String P = sc.next();  
  
 *naiveSearch*(T, P);  
  
 }  
 public static String readAndStoreWords(String filePath) throws IOException {  
 StringBuilder fileContent = new StringBuilder();  
  
 try (FileReader fileReader = new FileReader(filePath);  
 BufferedReader bufferedReader = new BufferedReader(fileReader)) {  
  
 String line;  
 while ((line = bufferedReader.readLine()) != null) {  
 String[] words = line.split("\\s+");  
 for (String word : words) {  
 fileContent.append(word).append(" ");  
 }  
 }  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
 return fileContent.toString();  
 }  
}

**Rabin-Karp algorithm:**

import java.io.BufferedReader;  
import java.io.FileReader;  
import java.io.IOException;  
import java.util.Scanner;  
  
public class RabinKarp {  
 private static final int *PRIME* = 101;  
  
 public static int rabinKarpSearch(String text, String pattern) {  
 int textLength = text.length();  
 int patternLength = pattern.length();  
 int patternHash = *calculateHash*(pattern, patternLength);  
 int textHash = *calculateHash*(text, patternLength);  
  
 int comparisons = 0;  
  
 for (int i = 0; i <= textLength - patternLength; i++) {  
 comparisons++;  
  
 if (patternHash == textHash && *checkEqual*(text, i, i + patternLength - 1, pattern)) {  
 System.*out*.println("Number of comparisons: " + comparisons);  
 return i;  
 }  
  
 if (i < textLength - patternLength) {  
 textHash = *recalculateHash*(text, i, i + patternLength, textHash, patternLength);  
 }  
 }  
  
 System.*out*.println("Number of comparisons: " + comparisons);  
 return -1;  
 }  
  
 private static int calculateHash(String str, int length) {  
 int hash = 0;  
 for (int i = 0; i < length; i++) {  
 hash += str.charAt(i) \* Math.*pow*(*PRIME*, i);  
 }  
 return hash;  
 }  
  
 private static int recalculateHash(String str, int oldIndex, int newIndex, int oldHash, int patternLength) {  
 int newHash = oldHash - str.charAt(oldIndex);  
 newHash /= *PRIME*;  
 newHash += str.charAt(newIndex) \* Math.*pow*(*PRIME*, patternLength - 1);  
 return newHash;  
 }  
  
 private static boolean checkEqual(String text, int start1, int end1, String pattern) {  
 int start2 = 0;  
 while (start1 <= end1) {  
 if (text.charAt(start1) != pattern.charAt(start2)) {  
 return false;  
 }  
 start1++;  
 start2++;  
 }  
 return true;  
 }  
  
 public static void main(String[] args) {  
 Scanner sc = new Scanner(System.*in*);  
 System.*out*.println("Enter the file path containing the job description:");  
 String filePath = sc.next();  
 String text = "";  
 try {  
 String fileContent = *readAndStoreWords*(filePath);  
 System.*out*.println("Words in the file: " + fileContent);  
 text = fileContent;  
 } catch (IOException e) {  
 System.*err*.println("Error reading the file: " + e.getMessage());  
 }  
 System.*out*.println("Enter the job search:");  
 String pattern = sc.next();  
  
 int index = *rabinKarpSearch*(text, pattern);  
 if (index != -1) {  
 System.*out*.println("Job found with string: " + text.substring(index)+ " starting a position " +index);  
 } else {  
 System.*out*.println("Job not found.");  
 }  
 }  
 public static String readAndStoreWords(String filePath) throws IOException {  
 StringBuilder fileContent = new StringBuilder();  
  
 try (FileReader fileReader = new FileReader(filePath);  
 BufferedReader bufferedReader = new BufferedReader(fileReader)) {  
  
 String line;  
 while ((line = bufferedReader.readLine()) != null) {  
 String[] words = line.split("\\s+");  
 for (String word : words) {  
 fileContent.append(word).append(" ");  
 }  
 }  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
 return fileContent.toString();  
 }  
}

**KMP algorithm:**

import java.io.BufferedReader;  
import java.io.FileReader;  
import java.io.IOException;  
import java.util.Scanner;  
  
public class KMP {  
 void KMPSearch(String pat, String txt) {  
 boolean flag = false;  
 int M = pat.length();  
 int N = txt.length();  
 int count = 0;  
  
 // create lps[] that will hold the longest  
 // prefix suffix values for pattern  
 int[] lps = new int[M];  
 int j = 0; // index for pat[]  
  
 // Preprocess the pattern (calculate lps[]  
 // array)  
 computeLPSArray(pat, M, lps);  
  
 int i = 0; // index for txt[]  
 while ((N - i) >= (M - j)) {  
 count += 1;  
 if (pat.charAt(j) == txt.charAt(i)) {  
 j++;  
 i++;  
 }  
 if (j == M) {  
 flag = true;  
 System.*out*.println("Job found with string: " + txt.substring(i - j) + " starting " +  
 "at position: " + (i - j));  
 j = lps[j - 1];  
 break;  
 }  
  
 // mismatch after j matches  
 else if (i < N  
 && pat.charAt(j) != txt.charAt(i)) {  
 // Do not match lps[0..lps[j-1]] characters,  
 // they will match anyway  
 if (j != 0)  
 j = lps[j - 1];  
 else  
 i = i + 1;  
 }  
 }  
 if (!flag)  
 System.*out*.println("Job Not Found");  
 System.*out*.println("Number of comparisons: " + count);  
 }  
  
 void computeLPSArray(String pat, int M, int[] lps) {  
 // length of the previous longest prefix suffix  
 int len = 0;  
 int i = 1;  
 lps[0] = 0; // lps[0] is always 0  
  
 // the loop calculates lps[i] for i = 1 to M-1  
 while (i < M) {  
 if (pat.charAt(i) == pat.charAt(len)) {  
 len++;  
 lps[i] = len;  
 i++;  
 } else // (pat[i] != pat[len])  
 {  
  
 if (len != 0) {  
 len = lps[len - 1];  
  
 // Also, note that we do not increment  
 // i here  
 } else // if (len == 0)  
 {  
 lps[i] = len;  
 i++;  
 }  
 }  
 }  
 }  
  
 // Driver code  
 public static void main(String[] args) {  
 Scanner sc = new Scanner(System.*in*);  
 System.*out*.println("Enter the file path containing the job description:");  
 String filePath = sc.next();  
 String text = "";  
 try {  
 String fileContent = *readAndStoreWords*(filePath);  
 System.*out*.println("Words in the file: " + fileContent);  
 text = fileContent;  
 } catch (IOException e) {  
 System.*err*.println("Error reading the file: " + e.getMessage());  
 }  
 System.*out*.println("Enter the job search:");  
 String pattern = sc.next();  
 new KMP().KMPSearch(pattern, text);  
 }  
  
 public static String readAndStoreWords(String filePath) throws IOException {  
 StringBuilder fileContent = new StringBuilder();  
  
 try (FileReader fileReader = new FileReader(filePath);  
 BufferedReader bufferedReader = new BufferedReader(fileReader)) {  
  
 String line;  
 while ((line = bufferedReader.readLine()) != null) {  
 String[] words = line.split("\\s+");  
 for (String word : words) {  
 fileContent.append(word).append(" ");  
 }  
 }  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
 return fileContent.toString();  
 }  
}

**Boyer-Moore-Horspool algorithm:**

import java.io.BufferedReader;  
import java.io.FileReader;  
import java.io.IOException;  
import java.util.Scanner;  
  
public class BoyerMoore {  
  
 public static int *SIZE* = 500;  
 public static int[] *table* = new int[*SIZE*];  
  
 public void tableShift(String pattern) {  
 int i, j, m;  
 char[] P = pattern.toCharArray();  
 m = pattern.length();  
  
 for (i = 0; i < *SIZE*; i++) {  
 *table*[i] = m;  
 }  
  
 for (j = 0; j < m - 1; j++) {  
 *table*[P[j]] = m - 1 - j;  
 }  
  
 }  
  
 public int horspoolMatch(String pattern, String text) {  
 int i, k, pos, m, n;  
 char[] T = text.toCharArray();  
 char[] P = pattern.toCharArray();  
 m = pattern.length();  
 n = text.length();  
  
 int compCount = 0;  
  
 for (i = m - 1; i < n; ) {  
  
 k = 0;  
  
 while ((k < m) && P[m - 1 - k] == T[i - k]) {  
 k++;  
 compCount++;  
 }  
  
 if (k == m) {  
 pos = i - m + 2;  
 System.*out*.println("Number of comparisons: "+compCount);  
 return pos;  
 } else {  
 i += *table*[T[i]];  
 }  
  
 compCount++;  
 }  
 System.*out*.println("Number of comparisons: "+compCount);  
 return -1;  
 }  
  
 public static void main(String[] args) {  
 int pos;  
 Scanner sc = new Scanner(System.*in*);  
 System.*out*.println("Enter the file path containing the job description:");  
 String filePath = sc.next();  
 String text = "";  
 try {  
 String fileContent = *readAndStoreWords*(filePath);  
 System.*out*.println("Words in the file: " + fileContent);  
 text = fileContent;  
 } catch (IOException e) {  
 System.*err*.println("Error reading the file: " + e.getMessage());  
 }  
 System.*out*.println("Enter the job search:");  
 String pattern = sc.next();  
 BoyerMoore sT = new BoyerMoore();  
  
 sT.tableShift(pattern);  
 pos = sT.horspoolMatch(pattern, text);  
  
 if (pos == -1) {  
 System.*out*.println("Job Not Found");  
 } else {  
 pos--;  
 System.*out*.println("Job found with string: " + text.substring(pos) + " starting at position: " +pos + "\n");  
 }  
 }  
  
 public static String readAndStoreWords(String filePath) throws IOException {  
 StringBuilder fileContent = new StringBuilder();  
  
 try (FileReader fileReader = new FileReader(filePath);  
 BufferedReader bufferedReader = new BufferedReader(fileReader)) {  
  
 String line;  
 while ((line = bufferedReader.readLine()) != null) {  
 String[] words = line.split("\\s+");  
 for (String word : words) {  
 fileContent.append(word).append(" ");  
 }  
 }  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
  
 return fileContent.toString();  
 }

Top of Form

Bottom of Form

Bottom of Form