The Rabin-Karp algorithm for substring search uses a rolling hash function to efficiently search for occurrences of a pattern within a text. It achieves this by hashing the pattern and substrings of the text, then comparing the hashes to quickly identify potential matches. If the hashes match, it performs a full comparison to confirm the match.

```
import java.util.*;
public class RabinKarpAlgorithm {
    private static final int PRIME = 101; // Prime number for hashing
    private static final int BASE = 256;  // Base for hashing
    public static List<Integer> search(String text, String pattern) {
        List<Integer> indices = new ArrayList<>();
        if (text == null || pattern == null || text.length() == 0 || pattern.length() == 0)
            return indices;
        int patternHash = calculateHash(pattern, pattern.length());
        int textHash = calculateHash(text, pattern.length());
        for (int i = 0; i <= text.length() - pattern.length(); i++) {
            if (patternHash == textHash && checkEqual(text, i, i + pattern.length() - 1, pattern)) {
                indices.add(i);
            }
            if (i < text.length() - pattern.length()) {
                textHash = recalculateHash(text, i, i + pattern.length(), textHash, pattern.length());
            }
        }
        return indices;
    }
    private static int calculateHash(String str, int len) {
        int hash = 0;
        for (int i = 0; i < len; i++) {
            hash += (int) (str.charAt(i) * Math.pow(BASE, len - i - 1)) % PRIME;
            hash %= PRIME;
        }
        return hash;
```

```
    }
    private static int recalculateHash(String str, int oldIndex, int newIndex, int oldHash, int patternLen) {
        int newHash = oldHash - str.charAt(oldIndex);
        newHash /= BASE;
        newHash += (int) (str.charAt(newIndex) * Math.pow(BASE, patternLen - 1)) % PRIME;
        newHash %= PRIME;
        return newHash;
    }
    private static boolean checkEqual(String text, int start1, int end1, String pattern) {
        if (end1 - start1 + 1 != pattern.length())
            return false;
        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) {
        String text = "ABABDABACDABABCABAB";
        String pattern = "ABABCABAB";
        List<Integer> indices = search(text, pattern);
        if (indices.isEmpty()) {
            System.out.println("Pattern not found in the text.");
        } else {
            System.out.println("Pattern found at indices: " + indices);
        }
```

```
    }
}
```

Explanation:

calculateHash: This method calculates the hash value of a substring using the rolling hash technique. It iterates over the characters of the substring and calculates the hash using the given prime number and base.

recalculateHash: This method updates the hash value when moving the window of the rolling hash. Instead of recalculating the hash from scratch, it subtracts the contribution of the leftmost character and adds the contribution of the rightmost character.

checkEqual: This method checks if two substrings are equal character by character.

search: This method performs the actual substring search using the Rabin-Karp algorithm. It iterates over the text, calculates the hash of each substring, and compares it with the hash of the pattern. If the hashes match, it performs a full comparison to confirm the match.

Now, let's discuss the impact of hash collisions on the algorithm's performance and how to handle them:

Impact of Hash Collisions: Hash collisions occur when different inputs produce the same hash value. In the context of the Rabin-Karp algorithm, hash collisions can lead to false positives, where different substrings produce the same hash value as the pattern. This can result in incorrect matches and decrease the algorithm's accuracy.

Handling Hash Collisions: To handle hash collisions, the Rabin-Karp algorithm uses a strong hash function that minimizes the likelihood of collisions. Choosing a large prime number for hashing and a large base can help reduce the probability of collisions. Additionally, if a hash collision is detected, the algorithm can perform a full comparison to confirm the match, ensuring accuracy.

Overall, while hash collisions can impact the performance and accuracy of the Rabin-Karp algorithm, careful selection of hash parameters and additional checks can help mitigate their effects.