Volume 2, Issue 12, December 2024 **ISSN (E):** 2938-3757

Web of Technology: Multidimensional Research Journal
webofjournals.com/index.php/4

EVALUATION OF EFFECTIVENESS BASED ON THE ANALYSIS OF THE STRUCTURE AND ALGORITHMS OF STRING DATA

Toirov Bobirmirzo Nodir ugli
11th Grade Student, Uzbekistan, Bukhara District, Specialized School
E-mail: toirovbobirmirzo53@gmail.com

Abstract
In data structures, a string is a sequence of characters used to represent text. Strings are commonly used in computer programs to store and manipulate textual data. They can be manipulated using various operations such as concatenation, substring extraction, and comparison.

Keywords: Encryption, hashing, pattern searching, naive string matching, KMP algorithm, Rabin-Karp algorithm, Boyer-Moore algorithm, Z-algorithm.

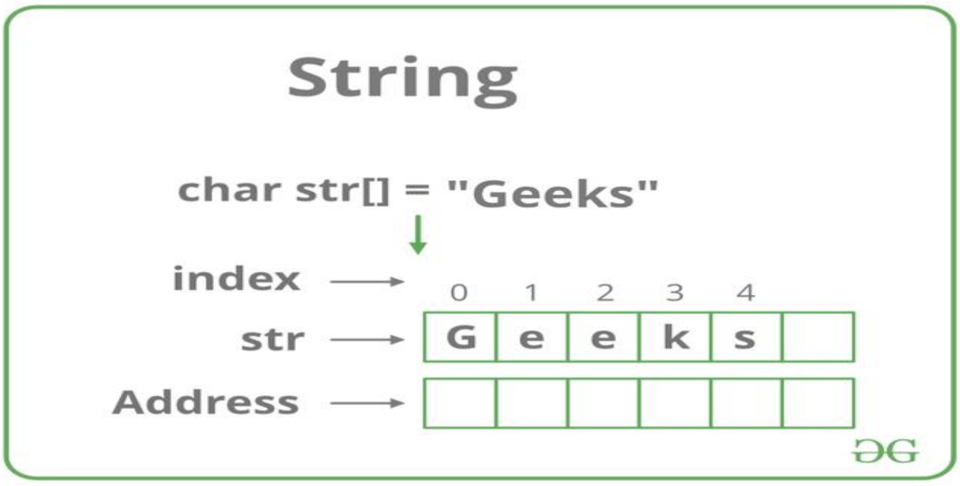
Introduction

Strings support a wide range of operations, including concatenation, substring extraction, length calculations, and more. These operations allow developers to efficiently manipulate and process string data.Text Processing: Strings are widely used for text processing tasks such as searching, manipulating, and parsing text data.

• Encryption and Hashing: Strings are commonly used in encryption and hashing algorithms to protect sensitive data and ensure data integrity.

• Database Operations: Strings are essential for working with databases, including storing and retrieving text-based data.

• Web development: Strings are used in web development to generate URLs, process form data, process input from web forms, and generate dynamic content.



Volume 2, Issue 12, December 2024 **ISSN (E):** 2938-3757

Web of Technology: Multidimensional Research Journal
webofjournals.com/index.php/4

Advantages of String:

• Text processing: Strings are used to represent text in programming languages. They can be used to manipulate and process text in various ways, such as searching, replacing, parsing, and formatting.

• Data representation: Strings can be used to represent other data types, such as numbers, dates, and times. For example, you can use a string to represent a date in the format "YYYY-MM-DD" or a time in the format "HH:MM:SS".

• Ease of use: Strings are easy to use and manipulate. They can be concatenated, truncated, and reversed, among other things. They also have a simple and intuitive syntax, making them accessible to programmers of all skill levels.

• Compatibility: Strings are widely used in programming languages, making them a universal data type. This means that strings can be easily transferred between different systems and platforms, making them a reliable and efficient way to communicate and exchange data.

• Memory efficiency: Strings are usually stored in a contiguous block of memory, which makes them efficient for allocation and allocation. This means that they can be used to represent large amounts of data without taking up too much memory.

Disadvantages of Strings:

• Memory consumption: Strings can consume a lot of memory, especially when working with large strings or many strings. This can be a problem in memory-constrained environments such as embedded systems or mobile devices.

• Immutability: In most programming languages, strings are immutable, meaning they cannot be modified after they are created. This can be a disadvantage when working with large or complex strings that require frequent modification, as it can lead to inefficiency and memory overhead.

• Performance overhead: String operations can be slower than operations on other data types, especially when working with large or complex strings. This is because string operations often involve copying and reallocating memory, which can be time-consuming**.** Encoding and decoding overhead: Strings can have different character encodings, which can lead to overhead when converting between them. This can be a problem when working with data from different sources or when communicating with systems that use different encodings.

• Security vulnerabilities: Strings can be vulnerable to security vulnerabilities such as buffer overflows or injection attacks if not handled properly. This is because strings can be manipulated by attackers to execute arbitrary code or access sensitive data.

Pattern searching

In data structures, pattern searching refers to the process of finding a specific pattern or substring in a larger data set. This can be done using a variety of algorithms and techniques to efficiently search for and locate the desired pattern within the data. Pattern searching is a common problem and is used in applications such as text processing, data mining, and data mining. Some of the popular algorithms for pattern searching include brute force method, Knut-Morris-Pratt (KMP) algorithm, Boyer-Moore algorithm, and Rabin-Karp algorithm. These algorithms help in efficiently searching for patterns in data structures such as arrays, strings, and binary trees.

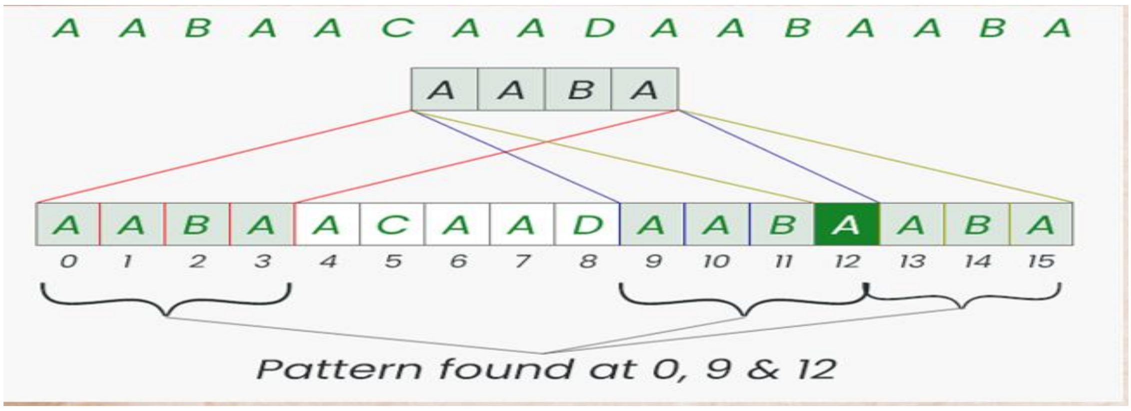
We use certain algorithms to perform the search process. The complexity of pattern searching varies from algorithm to algorithm. They are very useful in performing search in data structures. Pattern Searching algorithm is useful for finding patterns in substrings of larger strings. This process can be done using different algorithms that we will discuss in this blog.
Features of the pattern searching algorithm:

• Pattern searching algorithms quickly and accurately identify familiar patterns.
• Recognize and classify unfamiliar patterns.
• Identify patterns even if they are partially hidden.
• Quickly recognize patterns easily and automatically.

1. Naive String Matching

The simple string matching algorithm is the simplest pattern matching algorithm. It compares a pattern with all possible substrings of a text. The time complexity of this algorithm is O(mn), where m is the length of the pattern and n is the length of the text.
Input: text = "AABAACAADAABAABA", pattern = "AABA"

Result: Pattern found at index 0, pattern found at index 9, pattern found at index 12 will be of the form:


A A B A A C A A D A A B A A B A
A A B A
A A B A A C A A D A A B A A B A
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Pattern found at 0, 9 & 12

```
def search(pat, txt):
        M = len(pat)
        N = len(txt)
        for i in range(N-M+1):
                for j in range(M):
                        k = j+1
                        if(txt[i+j] != pat[j]):
                                break
                if(k == M):
                        print("Pattern found at index ", i)
txt = "AABAACAADAABAAABAA"
pat = "AABA"
search(pat, txt)
```

Here, a function named search is defined, which takes two parameters named pat and txt. These strings store the lengths of pat and text txt in variables M and N, respectively.

Outer loop:
This loop iterates over the variable i from 0 to N - M + 1. This is necessary to correctly insert the length of the pattern into the text.
Inner loop:
Here, j is iterated over the values from 0 to M. If the character at index i + j of the text does not match the character at index j of the pattern, the loop is stopped (break).
If the inner loop never breaks (i.e., all characters match), the else part is executed and the index i where the pattern is found is printed.

Usage:
Text (txt) - "AABAACAADAABAAABAA"
Pattern (pat) - "AABA"
The program calls the search(pat, txt) function and displays the locations where the pattern was found as follows:

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

So, the pattern "AABA" was found in the text at indexes 0, 9, and 13
This is the Python code, and below we will look at the C++ code.

```
#include <bits/stdc++.h>
using namespace std;
void search(char* pat, char* txt)
{
        int M = strlen(pat);
        int N = strlen(txt);
        for (int i = 0; i <= N - M; i++) {
                int j;
                for (j = 0; j < M; j++)
                        if (txt[i + j] != pat[j])
                                break;
              if (j == M)
                      cout << "Pattern found at index " << i << endl;
        }
}
int main()
{
        char txt[] = "AABAACAADAABAAABAA";
        char pat[] = "AABA";
        search(pat, txt);
        return 0;
}
```

2. KMP algoritmi:

The Knuth-Morris-Pratt (KMP) algorithm is a linear-time algorithm for string searching. It is more efficient than the simple pattern search algorithm, especially for large amounts of text, because it avoids unnecessary character comparisons. The main idea of KMP is to precompute a partial correspondence table, which helps determine the maximum length of valid prefixes and allows you to efficiently skip unnecessary comparisons.

Time complexity: O(n + m)

3.Rabin-karp algoritmi

The Rabin-Karp algorithm is a heuristic algorithm that uses hashing to match a pattern to a text. It computes a hash value for each substring of the pattern and the text. If the hash values match, it performs a character-by-character comparison to confirm the match. The time complexity of the Rabin-Karp algorithm is O (m + n), where m is the length of the pattern and n is the length of the text. Unlike a simple string matching algorithm, it does not traverse each character in the initial step, but rather filters out characters that do not match and then performs the comparison. Rabin-Karp compares the hash values of the string, not the strings themselves. For efficiency, the hash value of the next position in the text is easily computed from the hash value of the current position.

4. Z algoritmi:

The Z algorithm is another linear time series matching algorithm that efficiently finds occurrences of a pattern in text. It is based on the concept of Z-boxes, which represent the longest substring starting at a given position, which is also the prefix of the string. The Z algorithm preprocesses the pattern and text to create a Z-array, which is then used to find occurrences of the pattern in the text.

Time complexity: O(n+m)

5. Boyer Mur algoritmi:

The Boyer-Moore algorithm is a powerful and efficient string search algorithm that is widely used in practice. It is known for its ability to skip large sections of text when a mismatch occurs, making it particularly effective for searching large texts. The algorithm uses two heuristics: the bad character rule and the good addition rule.

Time complexity: O(n*m)

Use cases for pattern search:

Pattern search algorithms are important for finding occurrences of a specific pattern in a larger text or data set. The following are some use cases for pattern search algorithms:

• Text processing and editing: Search and replace functions in text editors or word processors often use pattern search algorithms to identify and replace specific patterns or strings.

• Information retrieval: Search engines use pattern search algorithms to efficiently find relevant documents or web pages based on user queries.

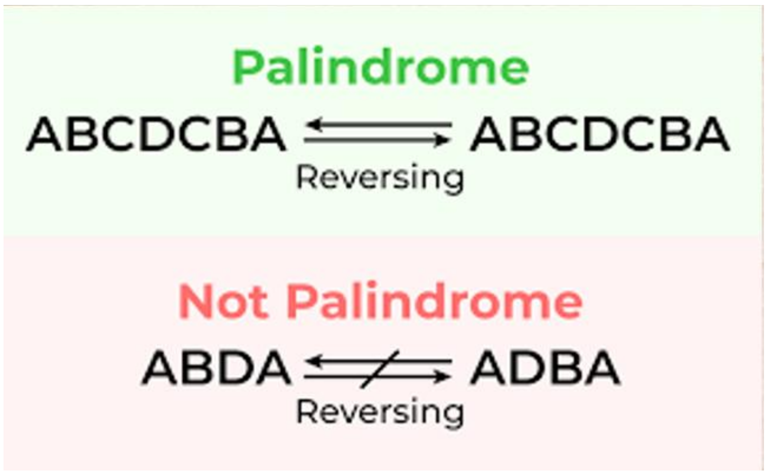
• Data mining: Identifying patterns in large data sets, such as finding sequences of events or behaviors, is a common application. Pattern searching helps extract meaningful information from data.

Volume 2, Issue 12, December 2024 ISSN (E): 2938-3757

• Bioinformatics: Analysis of DNA and protein sequences involves searching for specific patterns or motifs. Pattern searching algorithms are essential for tasks such as gene identification and sequence alignment in bioinformatics.
• Image processing: In image recognition, pattern searching algorithms can be used to identify specific shapes, structures, or textures within an image.
• Network security: Intrusion detection systems use pattern searching algorithms to identify patterns in network traffic that indicate malicious activity or security threats.
• Database systems: Database queries often require specific patterns or values within tables.
In a variety of applications, indexing and pattern matching algorithms optimize these searches for performance.
• Signal processing: In signal processing, pattern search algorithms can be used to detect specific patterns or trends in signals, which can be helpful in tasks such as speech recognition or audio processing.
• Robotics and computer vision: Pattern search is essential in robotics for object detection and navigation. Computer vision systems use these algorithms to detect and track objects in the visual field.

Palindrome string
A palindrome string is a sequence of characters that reads the same forwards and backwards. In terms of data structures, you can implement a function or algorithm to check whether a given string is a palindrome by comparing the characters at the beginning and end of the string. This can be achieved by using data structures such as arrays, strings, or linked lists to store and manipulate the characters of the input string. A string is considered a palindrome if the characters match at corresponding positions when passing through both ends.
If the reverse of a string is the same as the string, the string is called a palindrome. For example, "abba" is a palindrome because the reverse of "abba" is equal to "abba", so both of these strings are equal and are called palindromes, but "abbc" is not a palindrome.
A palindrome string has some of the following properties:
• A palindrome string has a symmetric structure, meaning that the characters in the first half of the string are the same as those in the second half, but in reverse order.
• Any string of length 1 is always a palindrome.
How can a palindrome be detected?
• Depending on case sensitivity, all characters should be converted to lowercase or uppercase, or left as is.
• Then the first and last characters are compared. If they are the same, the second and the rest are compared.
• If all pairs of characters match, it is a palindrome, otherwise it is not.

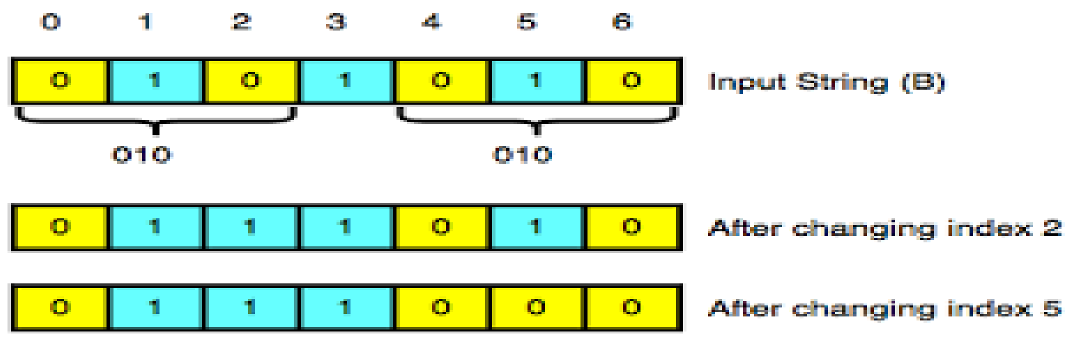
Now let's check whether the expression is a palindrome or not:
```
def isPalindrome(string):
        l = 0
        h = len(string) - 1
        while h > l:
                l+= 1
                h-= 1
                if string[l-1] != string[h + 1]:
                        return False
        return True
def isRotationOfPalindrome(string):
        if isPalindrome(string):
                return True
        n = len(string)
        for i in range(n-1):
                string1 = string[i + 1:n]
                string2 = string[0:i + 1]
                string1+=(string2)
                if isPalindrome(string1):
                        return True
        return False
print ("1" if isRotationOfPalindrome("aab") == True else "0")
print ("1" if isRotationOfPalindrome("abcde") == True else "0")
print ("1" if isRotationOfPalindrome("aaaad") == True else "0")
```



Palindrome check function (is Palindrome). Then it starts from the left and right corners. In subsequent lines, the characters are checked until they match
return True This function checks whether the given string is a palindrome. Starting from the beginning (l) and the end (h) of the string, each character is compared. If no character’s match, the function returns False. If all characters match, it returns True.
return False This function checks whether the given string is a palindrome cycle:
First, it checks whether the string itself is a palindrome using the isPalindrome function.
If the string is not a palindrome, it checks all possible cycles.
It checks each cycle using the isPalindrome function. If any cycle is a palindrome, it returns True.

If any cycle is not a palindrome, it returns False.
If the result is True, "1" is printed.
If the result is False, "0" is printed.
Examples:
The string "aab" is a rotation of a palindrome (because the rotation of "aba" is a palindrome).
The string "abcde" is not a rotation of a palindrome.
The string "aaaad" is not a rotation of a palindrome.
The code checks these examples using the isRotationOfPalindrome function and prints the results.

Binary String
A binary string is a string consisting of only two characters, usually the digits 0 and 1, that represents a sequence of binary numbers.
Binary String Variables:
In computer programming, binary string variables are used to store binary data, which is represented in binary (base-2) format rather than text (base-10) format. The three most common types of binary string variables used in databases are "BINARY", "VARBINARY", and "BLOB". Here is a brief overview of each:
BINARY: The BINARY data type is used to store fixed-length binary data. The data that is placed in a column must always be the same size, and the size of the column must be specified when the table is created. For example, a BINARY column can only store binary strings that are 10 bytes long if its definition is BINARY(10).
VARBINARY: The VARBINARY data type is similar to BINARY, but it allows for variable-length binary data. As a result, the data stored in a column can be any size, and the column size does not need to be specified. For example, a VARBINARY column can store binary strings of any size from 0 to 65,535 bytes.
BLOB: The BLOB (Binary Large Object) data type is used to store large binary data objects, such as images, audio files, or video files. BLOB columns are typically used when the size of the data being stored exceeds the maximum size allowed by the BINARY or VARBINARY data types. BLOB columns are often used to store very large files that cannot fit directly into a table, as they can store binary data of any size.



Properties of Binary String:
The unique characters 0 and 1 are usually used to create binary strings. The following are the important characteristics of binary strings:

Web of Technology: Multidimensional Research Journal
webofjournals.com/index.php/4

• Length: The number of bits in a binary string determines its length.
• Concatenation: Concatenation can be achieved by placing two or more binary strings one after the other.
• Substring: Binary strings can be split or split into binary strings for each substring.
• Prefix and Suffix: A prefix is a substring that starts a binary string at the beginning. A suffix of a binary string is a substring that is added to the end of the string.
• Hamming Distance: The number of points at which corresponding characters diverge in two binary strings of the same length is called the Hamming Distance.
• Regular Language: The set of all binary strings is a regular language, i.e. a finite state machine or regular expression can understand it.
• Binary arithmetic: In binary arithmetic, each bit corresponds to a power of 2, and binary strings can be used to represent integers. Given a string, we can check whether it is a binary string in Python:

```
def check(string):
    p = set(string)
    s = {'0', '1'}
    if s == p or p == {'0'} or p == {'1'}:
        print("Yes")
    else:
        print("No")
 if __name__ == "__main__":
     string = "101010000111"
     check(string)
```

Now let's look at the code:
def check(string) This line declares a function called check and accepts a parameter called string.
p = set(string) This line stores all the unique characters of the string argument as a set and assigns it to the variable p. The set function automatically removes duplicate elements from the string.
s = {'0', '1'} This line declares a set called s and adds the characters '0' and '1' to it. This set represents the elements of binary numbers.
if s == p or p == {'0'} or p == {'1'} This line tests a conditional expression:
If the set p is equal to the set s (meaning that p contains only the characters '0' and '1'), or if the set p contains only the characters '0', or if the set p contains only the characters '1', then the following block is executed and returns "Yes" if the condition is true, and "No" otherwise.
The last part of the code defines the entry point of the Python program. The if name == "main": statement: If the script is run directly (not imported), then the following block is executed: The string variable is set to 101010000111. check(string) is called, operating on the string argument.

CONCLUSION

So, the string data structure plays an important role in many areas of software. Strings are the primary means of representing and manipulating textual data. Here is a brief summary of the important topics based on the string data structure:

Binary strings consist of only the digits 0 and 1 and are widely used in computer science. They are one of the basic formats for working with computer language and hardware. Algorithms for parsing and manipulating binary strings usually need to be efficient and simple, as they can contain large amounts of data.

Pattern searching is used to find a specific pattern or substring within strings. This process is very important in computer science, as it is widely used in text analysis, search engines, bioinformatics, and many other fields. There are several popular pattern searching algorithms, including:

Naive Search Algorithm: The simplest algorithm, it checks every possibility.

Knuth-Morris-Pratt (KMP) Algorithm: Speeds up the search process by using prior reports.

Boyer-Moore Algorithm: Improves efficiency by searching backwards and forwards.

Palindrome strings are strings that are the same when read backwards. Detecting and manipulating palindromes is used in solving many algorithmic problems. Although detecting palindromes is simple, its various versions, such as searching for palindromes among substrings, can be complex.

String data structures are of great importance in software engineering and computer science. Their analysis and processing algorithms must be efficient and reliable, as they contain large amounts of textual data. Binary strings, pattern search, and palindrome strings are important parts of string data structures, which play an important role in solving various algorithmic problems.

REFERENCES

[1]. Introduction to Algorithms" - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (2009)

[2]. Kent D. Lee. Steve Hubbard Data Structures and Algorithms with Python

[3]. Mark Allen Weiss "Data in C++

[4]. https://www.geeksforgeeks.org/python-programming-language-tutorial/ Algorithm-book-by-Karumanchi

[5]. Data structures, algorithms and applications in C++ by Sartaj Sahni.