ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 6| June 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

IMPLEMENTATION OF find_string_size() FUNCTION TO ESTIMATE THE SIZE OF THE STRING AND COMPARING WITH BUILT-IN FUNCTION len (), FURTHER CHECKING THE EFFICIENCY OF find_string_size() METHODOLOGY USING ASYMPTOTIC NOTATIONS - A CASE STUDY

A. Krishaang[1], G.V Karthihesh
[1]6784, [2]6789
Class- XII 2026-27, Sainik School Amaravathinagar
Post: Amaravathinagar, Udumalpet Taluka, Tirupur Dt, Tamilnadu State

ABSTRACT

In the field of computer science, strings are one of the most commonly used data structures for storing and processing textual information. Determining the size or length of a string is a fundamental operation in programming and is frequently used in searching, sorting, validation, parsing, and data-processing applications. The efficiency of a program often depends on how effectively such basic operations are performed.

Most programming languages provide built-in functions to determine the length of a string. In Python, the built-in function len() is used to calculate the number of characters present in a string. Although this function is highly optimized, understanding the methodology behind string-length estimation is important for learners and researchers in computer science.

*This manuscript implements the function find_string_size(), which estimates the size of a string by traversing each character individually. Further, the execution of the methodology is examined and compared with the built-in len() function. The study also analyzes the time complexity and space complexity of the proposed methodology using asymptotic notations and evaluates its efficiency. *

In addition to theoretical analysis, the efficiency of the proposed methodology is experimentally evaluated using Python's time module. The execution time of find_string_size() is measured and compared with the built-in len() function for various input sizes. The observed results are further analyzed using asymptotic notations to determine the computational efficiency of both approaches.

KEYWORDS: *String, find_string_size(), len(), Time Complexity, Space Complexity, Asymptotic Notations, Character Count.*

1. INTRODUCTION

String is a collection of characters used to represent text data within computers. Programmers use strings to store values like names, messages, passwords, file paths, and more. Many algorithms rely on knowing how many characters are in the string before processing it.

Python's built-in len() function returns the number of characters in a string. This is possibly one of the most frequently used functions in programming, and it has been well-optimized by the Python interpreter.

By learning how the standard library calculates string length, students gain a better understanding of iteration, counting mechanisms, and algorithmic analysis. In this paper, we will implement our own function called find_string_size() to calculate the size of a string without using the built-in len() function.We will also analyze and compare the performance of this new method against the built-in one. Additionally, we will use asymptotic notation to analyze the efficiency of the function and examine its associated time and space complexity.

To determine whether this method is valid, it is important to evaluate both its validity and how well it functions. One aspect that helps measure the performance of the above method against the standard len() method is using the Python's time module to compare execution times of find_string_size() and the built-in len() methods. By comparing execution times between two approaches, we obtain first-hand information on how efficient each method is.

2. RELATED WORK

Related work in string processing techniques include string processing methods focused on modifying characters, searching for patterns in strings, searching through strings, compare strings, and estimating the size of strings; many techniques exist to perform this task efficiently in real time or when processing large amounts of data.

The built-in functions for string-processing operations in many of the modern programming languages have been developed to provide designers with both maximum performance and efficiency of memory. Studying how other implementations are performed also provides an opportunity for learners to learn more about how these operations work internally. Finally, research on algorithmic analysis is also being performed using complexity of time , complexity of space, and

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 6| June 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

asymptotic notations to assess the relative performance of the various types of operations that could result from these algorithms when operating under different conditions.

Researchers have maintained that measuring algorithm performance in an experimental environment as well as measuring algorithm performance in a theoretical environment are important aspects of evaluating the overall complexity of algorithms. Runtime benchmarking techniques are used to compare the performance of algorithms with varying sizes of data sets. The primary measurement tool for evaluating the execution time of algorithms and/or built-in functions in Python is the time module, which measures the execution time of Python code.

3. METHODOLOGY

The find_string_size() methodology is a character-counting approach used to determine the length of a given string. The find_string_size() function counts how many characters are in a given string. Its implementation differs from the standard len() because it counts one character at a time rather than returning the length as a result of more efficient internal processes.

The user enters a string when this function is executed. A counter is created, initialized to zero, and then each character in the provided string is reviewed. For each character that is discovered, the counter is incremented by 1.When all of the characters have been reviewed, the counter will hold the size of the string. To verify the accuracy of this method, the size reported from running find_string_size() against that returned from using the standard len() function will be compared. The method will then be analyzed using asymptotic notation to determine its efficiency.

In order to assess how efficient our proposed approach is, we will perform execution time measurements using the time module in Python. The start and end times for executing both functions will be recorded and the difference between the two values will be compared. The same process will also be performed for the built-in function len(), so we can compare the two different approaches' performance with each other.

4. STRUCTURE OF THE CHARACTER COUNT TABLE

Input String: `COMPUTER`

Character	Position	Count
C	0	1
O	1	2
M	2	3
P	3	4
U	4	5
T	5	6
E	6	7
R	7	8

5. FUNCTIONING OF CHARCTER COUNT TABLE

When the user submits a string to the system, the find_string_size() function will iterate through each of the characters in that string and accumulate both the location of each character and keep a count of how many characters have been processed so far.

For example, we start with the first character and count it. Each time we encounter a new character, we increment the count by one, continuing this process until we have processed the final character in the string.

The value that is contained in the counter variable after we have processed the entire string is equal to the size of the string. To ensure that the value we obtained matches the results from Python's default len() function, we compare the two results.

After the size of the string is determined, the execution time required by the methodology is measured using the Python time module. Similar measurements are carried out for the built-in len() function. The collected execution times are tabulated and graphically represented to analyze the efficiency of both approaches for different input sizes. The experimental observations help in understanding how the execution time changes as the size of the input string increases. The comparison also demonstrates the advantage of using optimized built-in functions for practical applications involving large amounts of textual data.

6. ALGORITHM: find_string_size()

```
STEP 01: START
STEP 02: ACCEPT A STRING FROM THE USER.
STEP 03: INITIALIZE A VARIABLE COUNT = 0.
STEP 04: RECORD THE START TIME FOR THE
find_string_size() FUNCTION.
STEP 05: TRAVERSE EACH CHARACTER PRESENT
IN THE STRING.
STEP 06: FOR EACH CHARACTER ENCOUNTERED,
INCREMENT COUNT BY 1.
STEP 07: CONTINUE UNTIL ALL CHARACTERS IN
THE STRING ARE PROCESSED.
STEP 08: RECORD THE END TIME FOR THE
find_string_size() FUNCTION.
STEP 09: CALCULATE THE EXECUTION TIME OF
find_string_size().
STEP 10: RECORD THE START TIME FOR THE
BUILT-IN len() FUNCTION.
STEP 11: EXECUTE THE len() FUNCTION ON THE
SAME STRING.
STEP 12: RECORD THE END TIME FOR THE len()
FUNCTION.
STEP 13: CALCULATE THE EXECUTION TIME OF
len().
STEP 14: COMPARE THE RESULTS OBTAINED FROM
BOTH METHODS.
STEP 15: DISPLAY THE STRING SIZE AND
EXECUTION TIMES.
STEP 16: STOP.
```

7. PROGRAM: find_string_size():

```
import time
# find_string_size() function definition
```

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 6| June 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

```
def find_string_size(text):

    count = 0

    for ch in text:
        count += 1

    return count

# main() function definition

def main():

    text = input("Enter a String : ")

    start_time = time.time()

    custom_size = find_string_size(text)

    end_time = time.time()

    custom_execution_time = end_time -
start_time

    start_time = time.time()

    builtin_size = len(text)

    end_time = time.time()

    builtin_execution_time = end_time -
start_time

    print("\nSize using
find_string_size() :",
          custom_size)

    print("Execution Time :",
          custom_execution_time)

    print("\nSize using len() :",
          builtin_size)

    print("Execution Time :",
          builtin_execution_time)

# Calling Function

if __name__ == "__main__":
    main()
```

8. FUNCTIONING OF CODE

The code implements the **find_string_size()** methodology and compares its execution time with Python's built-in **len()** function. The code implements a new method of measuring the size of a given string (find_string_size()) and times how long it takes to run against Python's built-in len() function. The user is prompted for a string to be processed using the new method. The find_string_size() function traverses each character in the input string and increments a counter variable (i.e., for every character). When all characters have been processed, the counter's final value is equal to the size of the input string.

To evaluate the methodology's efficiency, the built-in Python time module is used. Specifically, the current system time is recorded prior to executing find_string_size() and again after execution is finished. The elapsed time is computed as the difference between the two recorded values, and represents the amount of time that it took to run the find_string_size() function.

The same process is followed using the built-in len() function. The amount of time it took to execute len() is recorded, and used to compare it with the time it took to run find_string_size(). The size of each string and the respective execution times are displayed.

The main() function handles user input, execution of both methods and the calculation of the execution times and the display of results. Finally, the conditional statement if name == "main" ensures that when the file is executed on its own, the start of its execution is triggered by calling the main() method.

This methodology helps in studying the correctness of the proposed function and provides practical evidence for comparing its efficiency with the built-in len() function.

9. COMPLEXITY OF ALGORITHM

In computer science, algorithm analysis plays an important role in determining the efficiency of a program. There are many different algorithms that can solve the same problem, but they will each utilize varying degrees of time and memory to execute the solution.

The overall goal of analyzing an algorithm's performance is to find out which one(s) will be performed most efficiently in solving the specific problem you are trying to solve.Two important factors considered during analysis are time complexity and space complexity. Time complexity measures the amount of time required to execute an algorithm, while space complexity measures the amount of memory consumed during execution.

The proposed **find_string_size()** methodology is analyzed using asymptotic notations and compared with the built-in **len()** function. This comparison helps in understanding the computational efficiency of both approaches.

10. SPACE COMPLEXITY

The space complexity of an algorithm refers to the amount of memory required during execution as a function of input size.

Method find_string_size() uses just one counter for counting processed characters in a string. There are no other auxiliary data structures created when executing this method (i.e., no arrays, lists, hashmaps, etc.). Hence, there will be no extra memory consumed based on the length of the string.

Equally, the built in len() function will have little or nothing additional required to execute. Hence, the space complexity of the proposed methodology is:

Space Complexity = O(1)

Since the auxiliary memory does not increase with input size, the methodology exhibits constant space complexity.

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 6| June 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

11. TIME COMPLEXITY

The time complexity describes the time taken by an algorithm as a function of the size of the input given to the algorithm, and it indicates how much of time is devoted to executing that algorithm.

The find_string_size() method processes every character in the string a single time. Therefore the time taken will grow at a linear rate based on the size of the string supplied to find_string_size().

The asymptotic notation used for analyzing algorithms is as follows:

Big O (O): This is used to indicate the worst-case scenario.

Big Theta (Θ): This is used to indicate the average case scenario.

Big Omega (Ω): This is used to indicate the best-case scenario.

The find_string_size() algorithm uses an approach to traverse all characters of the string to compute the size of the string, and it takes proportionally as much time as supplying a string to the method increases in size.

Thus:

find_string_size() = O(n)

The built-in len() function retrieves an internal length measurement maintained by Python, so it has an almost constant time to compute, depending on the size of the string passed to it.

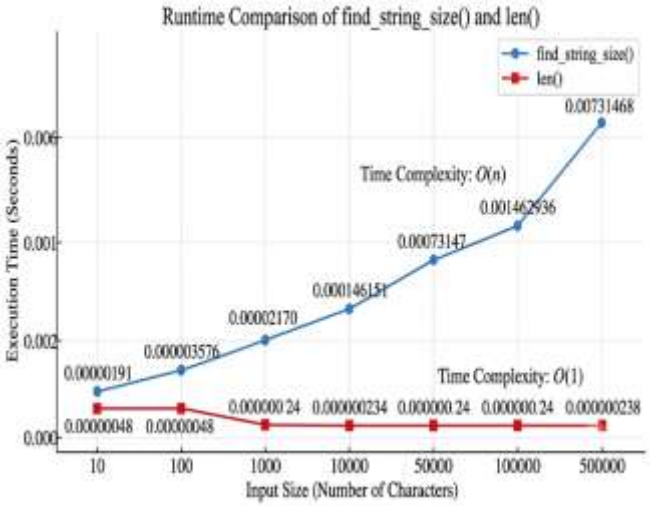
Thus: len() = O(1)

12. RUNTIME COMPLEXITY OF find_string_size() and len()

Input Size	find_string_size() (seconds)
10	1.9073486328125e-06
100	3.576278686`5234375e-06
1000	2.1696090698242188e-05
10000	0.0001461505889892578
50000	0.0007314682006835938
100000	0.0014629364013671875
500000	0.007314682006835937 5

Input Size	len() (seconds)
10	4.76837158203125e-07
100	4.76837158203125e-07
1000	2.384185791015625e-07
10000	2.384185791015625e-07
50000	2.384185791015625e-07
100000	2.384185791015625e-07
500000	2.384185791015625e-07

13. GRAPHICAL REPRESENTATION



14. THE RUNTIME COMPLEXITY OF find_string_size():

For Loop Function:

```
for ch in text:
    count += 1
```

For the above code, the loop executes once for every character present in the string.

Therefore, the time complexity is:**O(n)**

where **n** represents the number of characters present in the string.

Built-in len() Function:

```
len(text)
```

The built-in function retrieves the string length from internally stored metadata.

Therefore, the time complexity is:**O(1)**

Total Time Complexity:

find_string_size()

- Character Traversal : O(n)
- Counting Operation : O(n)

Overall Time Complexity:

Best Case : O(n)

Average Case : Θ(n)

Worst Case : O(n)

Final Time Complexity:

1. **O(n)**
2. **len()**

Length Retrieval : O(1)

Final Time Complexity:**O(1)**

15. CONCLUSION

In programming, it is possible to use multiple methods to measure the length of a string. The method described below for calculating the length of a string uses a variable that is incremented during each traversal of the string.

The results of the timing experiments described below, measured using Python's built-in time module, indicate that the output of the proposed method for finding the length of a string is the same as the output from calling Python's built-in len() function. However, as the length of the string increases, so does

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 6| June 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

the time it takes to run the proposed method, because each character must be examined individually.

In terms of formulas, the method for calculating the length of the given string has an asymptotic complexity of O(n) — the length of the string is proportional to the number of characters in the string. By contrast, because Python keeps a count of the total number of characters in its internal representation of a string, calling Python's built-in len() function is an O(1) operation.

For this reason, while the proposed method for calculating the length of a string is helpful for understanding how strings are internally represented in a computer, the built-in len() function is a more efficient method for measuring the length of a string for all programming purposes.

16. ACKNOWLEDGEMENT

Apart from our efforts, the success of any work or project depends largely on the encouragement and guidelines of many others. I take this opportunity to express my gratitude to the people who have been instrumental in the successful completion of this research paper.

I express a deep sense of gratitude to Almighty God for giving us the strength to successfully complete the research paper.

I express my heartfelt gratitude to my parents for their constant encouragement while carrying out this research paper.

I express my deep sense of gratitude to the luminary, **The Principal Captain (Indian Navy) Rahul Thiyyat, Sainik School Amaravathinagar,** who has been continuously motivating and extending their helping hand to us.

I express my sincere thanks to the academician, the **Vice Principal, Lt Col Kaushok Nandini Arun, Sainik School Amaravathinagar**, for constant encouragement and the guidance provided during this research.

My sincere thanks to **Mr.Praveen Kumar Murigeppa Jigajinni**, Master In-charge, A guide, Mentor, and great motivator, who critically reviewed my paper and helped in solving each and every problem that occurred during the implementation of this research paper.

17. REFERENCES

1. *https://www.freecodecamp.org/news/time-complexity-of-algorithms/*
2. *https://www.educative.io/edpresso/time-complexity-vs-space-complexity*
3. *https://en.wikipedia.org/wiki/Space_complexity*