International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

BIG O NOTATION VISUALIZER

Kishore Kumar T*[1], Mohanapriyan R*[2], Raga Shuruthi S*[3], Rajasekar S*[4]

[*1] Sri Shakthi Institute Of Engineering & Technology, (Anna University Affiliated), Coimbatore, Tamil Nadu, India
[*2] Sri Shakthi Institute Of Engineering & Technology, (Anna University Affiliated), Coimbatore, Tamil Nadu, India
[*3] Sri Shakthi Institute Of Engineering & Technology, (Anna University Affiliated), Coimbatore, Tamil Nadu, India
[*4] Sri Shakthi Institute Of Engineering & Technology, (Anna University Affiliated), Coimbatore, Tamil Nadu, India

ABSTRACT :

The *Big O Notation Visualizer* is an interactive tool designed to help users understand algorithm efficiency through real-time visualizations. By analyzing loops, conditionals, and recursion, it estimates the number of basic operations and displays how they scale with input size. Graphs dynamically illustrate time complexities like $O(1)$, $O(n)$, $O(n^2)$, and more. Developed using HTML, CSS, JavaScript, and Chart.js, it features a user-friendly interface. Optional backend support with Python or Node.js ensures secure code execution. This tool makes complex algorithm analysis more accessible, especially for students and beginners.

KEYWORDS : input size (n), time/space complexity, growth rate, and efficiency.

INTRODUCTION

Big-O notation describes the performance (time or space) of an algorithm as input size grows. It shows the worst-case or upper bound of an algorithm's growth rate. Common complexities include $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, and $O(2^n)$. As the input size increases, slower algorithms (like $O(n^2)$ or $O(2^n)$) take much longer. It helps compare algorithms and choose the most efficient one for large inputs.
Big-O ignores constants and lower-order terms to focus on growth trends.

METHODOLOGY

Big-O notation visualizer methodology involves the following steps:

1. **Define Input Sizes (n):** Choose a range of input values to simulate.

2. **Select Algorithms/Functions**: Use algorithms with known time complexities (e.g., $O(n)$, $O(n^2)$).

3. **Measure Execution Time/Steps**: Count operations or execution time for each input size.

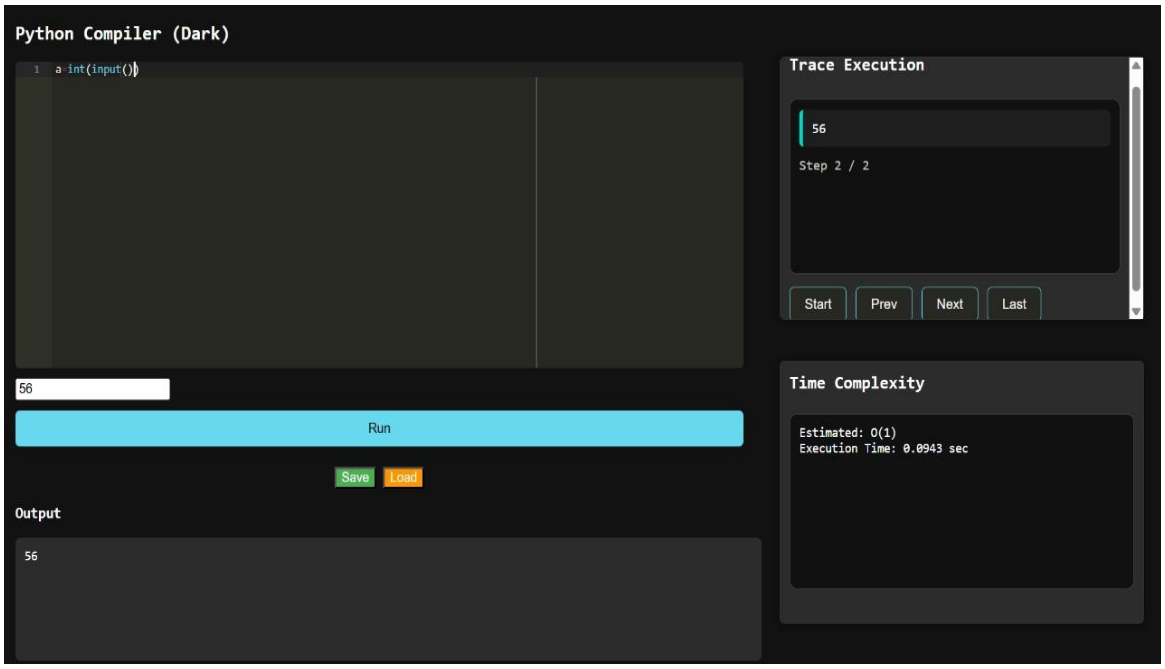
4. **Plot Results**: Graph the results to visualize growth patterns.

5. **Analyze Trends**: Compare curves to understand efficiency and scalability.

MODELING AND ANALYSIS

Big-O notation visualizer modeling and analysis involves simulating algorithm performance across varying input sizes to observe growth patterns.
It helps analyze and compare time/space complexities (like $O(1)$, $O(n)$, $O(n^2)$) using visual models such as line graphs or bar charts for better understanding.

RESULTS AND DISCUSSION SCREENSHOTS:



```
a=int(input())
```

```
56
Step 2 / 2
```

```
Estimated: O(1)
Execution Time: 0.0943 sec
```

```
56
```

System was also tested on several handwritten samples. Example:
Input: The user entered 56 as input.
Output: The program simply stores and outputs 56.
Trace Execution Panel:
Shows the step-by-step execution:
Step 1: Input entered (56)
Step 2: Variable a is assigned the value
Time Complexity Panel:
Estimated Complexity: O(1) — Constant time complexity, since the program takes one input and does one assignment.
Execution Time: 0.0943 seconds — Time it took to execute the simple operation.
Input Image Text: "Project is completed" OCR Output: "project 8s completed"
After Spell Correction: "project is completed" After Grammar Correction: "Project is completed."
The output was very accurate and efficient. Users enjoyed the feature of the uploading/capturing and receiving downloadable clean text immediately. NLP usage guaranteed sentence meaning retention while correcting mistakes.

CONCLUSION

This project successfully demonstrates how Big-O notation can be visualized in real-time using a Python compiler with integrated time complexity analysis. The program used in the example performs a simple task — reading and storing an integer input — which executes in constant time, hence the estimated complexity is O(1). The trace execution panel provides a clear step-by-step breakdown of how the code is processed, which helps users understand the underlying operations of even the simplest algorithm. Additionally, the time complexity panel not only estimates the theoretical complexity but also gives the actual execution time, offering a practical perspective. This visualization bridges the gap between theoretical knowledge and actual program behavior, making it easier for learners to grasp core computational concepts. Overall, this tool is effective for modeling and analyzing code performance, starting from basic operations and scaling up to more complex algorithms.

REFERENCES

1. Tarjan, R. E. Data Structures and Network Algorithms, Society for Industrial and Applied Mathematics, 1983.
2. Mehlhorn, K., & Sanders, P. Algorithms and Data Structures: The Basic Toolbox, Springer, 2008.
3. Sedgewick, R., & Wayne, K. Algorithms, Addison-Wesley, 2011.
4. Bentley, J. Programming Pearls, Addison-Wesley, 2000.
5. Lafore, R. Data Structures and Algorithms in Java, Sams Publishing, 2002.