Government College of Engineering (GCOEJ), Jalgaon

(An Autonomous Institute of Government of Maharashtra)

DEPARTMENT OF COMPUTER ENGINEERING

INDUSTRIAL LECTURE REPORT ON

TIME AND SPACE COMPLEXITY ANALYSIS OF ALGORITHMS

(Academic Year 2023-24)

Submitted by:

Bhargav Shamuvel Gurav (2041009)

Government College of Engineering (GCOEJ), Jalgaon

(An Autonomous Institute of Govt. of Maharashtra)

DEPARTMENT OF COMPUTER ENGINEERING

CERTIFICATE

This is to certify that the ***Industrial Lecture*** report**, “Time and Space complexity analysis of Algorithm”**, which is being submitted here with for the award of ***LY Computer Engineering (7th Semester)*** is the result of the work completed by ***Bhargav Gurav (2041009)*** under my supervision and guidance within offline mode of classes of the institute, in the academic year 2023-24.

.

Head of Department

Dr. D. V. Chaudhari

INDEX

ABSTRACT

The analysis of time and space complexity in algorithms constitutes a pivotal cornerstone in the realm of computer science and software engineering. This report delves into the intricate study of algorithmic efficiency, illuminating the significance of evaluating computational resources—time and memory—utilized by algorithms. Exploring fundamental concepts and methodologies, it navigates through the understanding of Big O notation, elucidating how it quantifies algorithmic performance and scalability. Moreover, this report elucidates the symbiotic relationship between time and space complexities, offering insights into strategies for optimizing algorithms to achieve optimal performance and mitigate resource constraints, thereby laying a foundational understanding crucial for algorithm design and problem-solving in diverse computational domains.

INTRODUCTION

Algorithm Analysis

Analysis of efficiency of an algorithm can be performed at two different stages, before implementation and after implementation, as

1. A priori analysis − This is defined as theoretical analysis of an algorithm. Efficiency of algorithm is measured by assuming that all other factors e.g. speed of processor, are constant and have no effect on implementation.
2. A posterior analysis − This is defined as empirical analysis of an algorithm. The chosen algorithm is implemented using programming language. Next the chosen algorithm is executed on target computer machine. In this analysis, actual statistics like running time and space needed are collected.

Algorithm analysis is dealt with the execution or running time of various operations involved. Running time of an operation can be defined as number of computer instructions executed per operation.

Algorithm Complexity

Suppose X is treated as an algorithm and N is treated as the size of input data, the time and space implemented by the Algorithm X are the two main factors which determine the efficiency of X.

1. Time Factor − The time is calculated or measured by counting the number of key operations such as comparisons in sorting algorithm.
2. Space Factor − The space is calculated or measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm f(N) provides the running time and / or storage space needed by the algorithm with respect of N as the size of input data.

Space Complexity

Space complexity of an algorithm represents the amount of memory space needed the algorithm in its life cycle.

Space needed by an algorithm is equal to the sum of the following two components

A fixed part that is a space required to store certain data and variables (i.e. simple variables and constants, program size etc.), that are not dependent of the size of the problem.

A variable part is a space required by variables, whose size is totally dependent on the size of the problem. For example, recursion stack space, dynamic memory allocation etc.

Space complexity S(p) of any algorithm p is S(p) = A + Sp(I) Where A is treated as the fixed part and S(I) is treated as the variable part of the algorithm which depends on instance characteristic I. Following is a simple example that tries to explain the concept

```
Algorithm
SUM(P, Q)
Step 1 - START
Step 2 - R ← P + Q + 10
Step 3 - Stop
```

Here we have three variables P, Q and R and one constant. Hence S(p) = 1+3. Now space is dependent on data types of given constant types and variables and it will be multiplied accordingly.

Time Complexity
Time Complexity of an algorithm is the representation of the amount of time required by the algorithm to execute to completion. Time requirements can be denoted or defined as a numerical function t(N), where t(N) can be measured as the number of steps, provided each step takes constant time.
For example, in case of addition of two n-bit integers, N steps are taken. Consequently, the total computational time is t(N) = c*n, where c is the time consumed for addition of two bits. Here, we observe that t(N) grows linearly as input size increases.

BASICS OF ALGORITHM ANALYSIS

Why Analysis of Algorithms is important?
To predict the behavior of an algorithm without implementing it on a specific computer.
It is much more convenient to have simple measures for the efficiency of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system changes. It is impossible to predict the exact behavior of an algorithm. There are too many influencing factors. The analysis is thus only an approximation; it is not perfect. More importantly, by analyzing different algorithms, we can compare them to determine the best one for our purpose.
Types of Algorithm Analysis:

1. Best case
2. Worst case
3. Average case

Best case: Define the input for which algorithm takes less time or minimum time. In the best case calculate the lower bound of an algorithm. Example: In the linear search when search data is present at the first location of large data then the best case occurs.

Worst Case: Define the input for which algorithm takes a long time or maximum time. In the worst calculate the upper bound of an algorithm. Example: In the linear search when search data is not present at all then the worst case occurs.

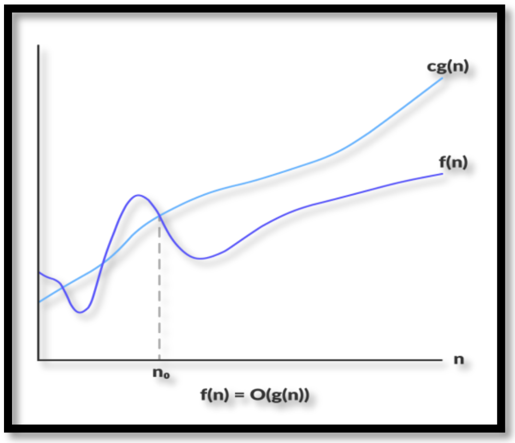
Average case: In the average case take all random inputs and calculate the computation time for all inputs. And then we divide it by the total number of inputs.
Average case = all random case time / total no of case

ASYMPTOTIC ANALYSIS: BIG-O NOTATION AND MORE

The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations.
An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.
The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

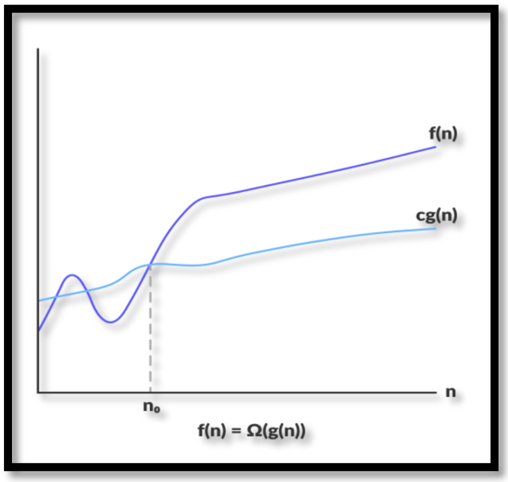
Asymptotic Notations
Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.
For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.
But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.
When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.
There are mainly three asymptotic notations:

1. Big-O notation
2. Omega notation
3. Theta notation



Big-O Notation (O-notation)
Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.

Big-O gives the upper bound of a function
O(g(n)) = { f(n): there exist positive constants c and n0
such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n0$ }
The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.
For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)).
Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

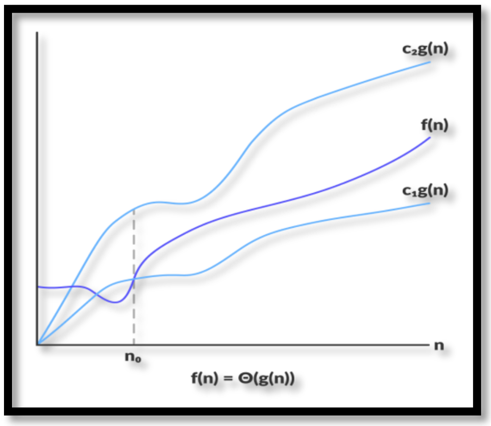


Omega Notation (Ω-notation)
Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.
Omega gives the lower bound of a function
Ω(g(n)) = { f(n): there exist positive constants c and n0
such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n0$ }
The above expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c such that it lies above cg(n), for sufficiently large n.

For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n)).

Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.



Theta bounds the function within constants factors

For a function g(n), Θ(g(n)) is given by the relation:

Θ(g(n)) = { f(n): there exist positive constants c1, c2 and n0
such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0 }

The above expression can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched between c1g(n) and c2g(n), for sufficiently large n.

If a function f(n) lies anywhere in between c1g(n) and c2g(n) for all n ≥ n0, then f(n) is said to be asymptotically tight bound.

PRACTICAL APPLICATIONS AND EXAMPLES

Real-World Applications:

1. Search Engines:

Case Study: Google's PageRank algorithm is a prime example of an efficient algorithm. By analyzing the link structure of the web, PageRank determines the relevance of web pages, enabling Google to provide efficient search results. Its time complexity is optimized to swiftly navigate through vast amounts of data, ranking pages based on their importance.

2. Sorting Algorithms in Databases:

Case Study: Database management systems utilize sorting algorithms like Quicksort and Mergesort to efficiently arrange data. These algorithms offer optimal time complexities, allowing for faster retrieval and manipulation of data, thus enhancing the overall performance of the database.

3. Compression Algorithms:

Case Study: ZIP compression uses algorithms like Huffman coding or Lempel-Ziv-Welch (LZW) to efficiently compress files. These algorithms optimize space complexity, reducing file sizes while maintaining or even improving the quality of the data. This enables quicker file transfers and saves storage space.

4. Network Routing:

Case Study: Routing algorithms in networking, such as Dijkstra's algorithm, help find the shortest path between nodes in a network efficiently. By analyzing network topology and optimizing time complexities, these algorithms enable routers to efficiently transmit data packets, reducing latency and congestion.

5. Image Processing:
Case Study: Convolutional Neural Networks (CNNs) employ efficient algorithms to process images. Algorithms like the Fast Fourier Transform (FFT) optimize time complexities, allowing CNNs to efficiently recognize patterns and features within images, enabling applications in facial recognition, object detection, and medical imaging.
Efficient algorithms underpin various technologies and systems, enhancing their speed, reliability, and scalability, thereby demonstrating the crucial role of complexity analysis in optimizing computational tasks across diverse real-world applications.

OPTIMIZATION STRATEGIES

Here are several optimization strategies used in algorithm design:
1. Algorithmic Selection:
Choose the Right Algorithm: Different algorithms solve the same problem with varying efficiencies. Select the most suitable algorithm based on the problem's characteristics and constraints.

2. Data Structures:
Use Efficient Data Structures: Utilize appropriate data structures like arrays, hash tables, heaps, and trees that suit the problem requirements. Optimal data structures can significantly enhance algorithm performance.

3. Divide and Conquer:
Divide Large Problems: Break down complex problems into smaller, more manageable subproblems using techniques like divide and conquer (e.g., Merge Sort, Quick Sort). This reduces the overall computational load.

4. Dynamic Programming:
Memoization and Tabulation: Employ dynamic programming techniques to store and reuse intermediate results, avoiding redundant calculations. Memoization (top-down) and tabulation (bottom-up) are effective in optimizing recursive algorithms.

5. Greedy Algorithms:
Greedy Strategy: Implement greedy algorithms when applicable. Greedy algorithms make locally optimal choices at each step, aiming to reach a global optimum. They often have lower time complexities.

6. Pruning Techniques:
Algorithmic Pruning: In tree-based algorithms or search spaces, pruning techniques like alpha-beta pruning in minimax algorithms can eliminate unnecessary branches, reducing time complexities.

7. Approximation Algorithms:
Approximate Solutions: For NP-hard problems, employ approximation algorithms to find solutions that are close to optimal within a reasonable time frame. These algorithms sacrifice accuracy for efficiency.

8. Parallelism and Concurrency:
Parallel Computing: Utilize parallel algorithms and concurrency to leverage multiple processing units simultaneously, thereby reducing execution time for certain types of problems.

9. Preprocessing and Caching:
Precompute and Cache Results: Identify parts of the algorithm where results can be precomputed and stored for reuse. Caching frequently used data can significantly improve performance.

10. Space-Time Trade-offs:
Trade-off Analysis: Evaluate trade-offs between time and space complexities. Sometimes, optimizing one may increase the other. Finding a balance suitable for the problem context is crucial.
By employing these strategies in algorithm design and implementation, developers can optimize algorithms to be more efficient in terms of time and space complexities, leading to faster execution and better utilization of computational resources.

CONCLUSION

In conclusion, the exploration into the realm of time and space complexity analysis unveils the critical essence of efficiency in algorithmic design. Through the lenses of Big O notation, this report illuminated the means to gauge and quantify the performance of algorithms, facilitating a nuanced understanding of their scalability and resource utilization. The interplay between time and space complexities emerged as a pivotal consideration, guiding strategies for optimizing algorithms to strike a balance between computational resources, ultimately fostering the creation of more efficient and scalable solutions.

REFERENCES

1. ChatGPT
2. GeeksForGeeks
3. JavatPoint