How to analyse Complexity of Recurrence Relation

The analysis of the complexity of a recurrence relation involves finding the asymptotic upper bound on the running time of a recursive algorithm. This is usually done by finding a closed-form expression for the number of operations performed by the algorithm as a function of the input size, and then determining the order of growth of the expression as the input size becomes large.

Here are the general steps to analyze the complexity of a recurrence relation:

Substitute the input size into the recurrence relation to obtain a sequence of terms.

Identify a pattern in the sequence of terms, if any, and simplify the recurrence relation to obtain a closed-form expression for the number of operations performed by the algorithm.

Determine the order of growth of the closed-form expression by using techniques such as the Master Theorem, or by finding the dominant term and ignoring lower-order terms.

Use the order of growth to determine the asymptotic upper bound on the running time of the algorithm, which can be expressed in terms of big O notation.

It's important to note that the above steps are just a general outline and that the specific

depending on the specific recurrence relation being analyzed.

In the previous post, we discussed the analysis of loops. Many algorithms are recursive. When we analyze them, we get a recurrence relation for time complexity. We get running time on an input of size n as a function of n and the running time on inputs of smaller sizes. For example in Merge Sort, to sort a given array, we divide it into two halves and recursively repeat the process for the two halves. Finally, we merge the results. Time complexity of Merge Sort can be written as T(n) = 2T(n/2) + cn. There are many other algorithms like Binary Search, Tower of Hanoi, etc.

need of solving recurrences:
The solution of recurrences is important because it provides information about the running time of a recursive algorithm. By solving a recurrence, we can determine the asymptotic upper bound on the number of operations performed by the algorithm, which is crucial for evaluating the efficiency and scalability of the algorithm.

Here are some of the reasons for solving recurrences:

Algorithm Analysis: Solving a recurrence is an important step in analyzing the time complexity of a recursive algorithm. This information can then be used to determine the best algorithm for a particular problem, or to optimize an existing algorithm.

Performance Comparison: By solving recurrences, it is possible to compare the running times of different algorithms, which can be useful in selecting the best algorithm for a particular use case.

Optimization: By understanding the running time of an algorithm, it is possible to identify bottlenecks and make optimizations to improve the performance of the algorithm.

Research: Solving recurrences is an essential part of research in computer science, as it helps to understand the behavior of algorithms and to develop new algorithms.

Overall, solving recurrences plays a crucial role in the analysis, design, and optimization of algorithms, and is an important topic in computer science.
There are mainly three ways of solving recurrences:

Substitution Method:

We make a guess for the solution and then we use mathematical induction to prove the guess is correct.

For example consider the recurrence T(n) = 2T(n/2) + n

We need to prove that T(n) <= cnLogn. We can assume that it is true for values smaller than n.

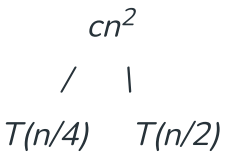
$T(n) = 2T(n/2) + n$
$<= 2cn/2Log(n/2) + n$
$= cnLogn – cnLog2 + n$
$= cnLogn – cn + n$
$<= cnLogn$

Recurrence Tree Method:

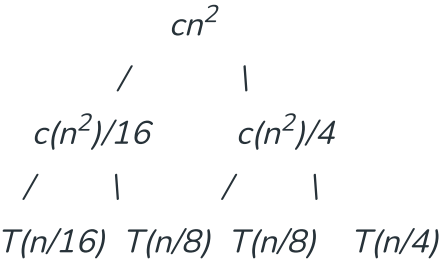
In this method, we draw a recurrence tree and calculate the time taken by every level of the tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically arithmetic or geometric series.

For example, consider the recurrence relation

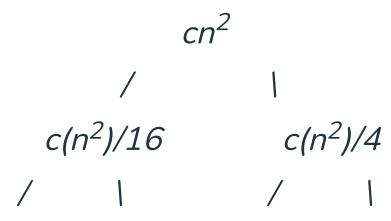
$T(n) = T(n/4) + T(n/2) + cn^2$



If we further break down the expression T(n/4) and T(n/2), we get the following recursion tree.



Breaking down further gives us following



To know the value of T(n), we need to calculate the sum of tree nodes level by level. If we sum the above tree level by level,

we get the following series T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ...
The above series is a geometrical progression with a ratio of 5/16.

To get an upper bound, we can sum the infinite series. We get the sum as $(n^2)/(1 - 5/16)$ which is $O(n^2)$

Master Method:

Master Method is a direct way to get the solution. The master method works only for the following type of recurrences or for recurrences that can be transformed into the following type.

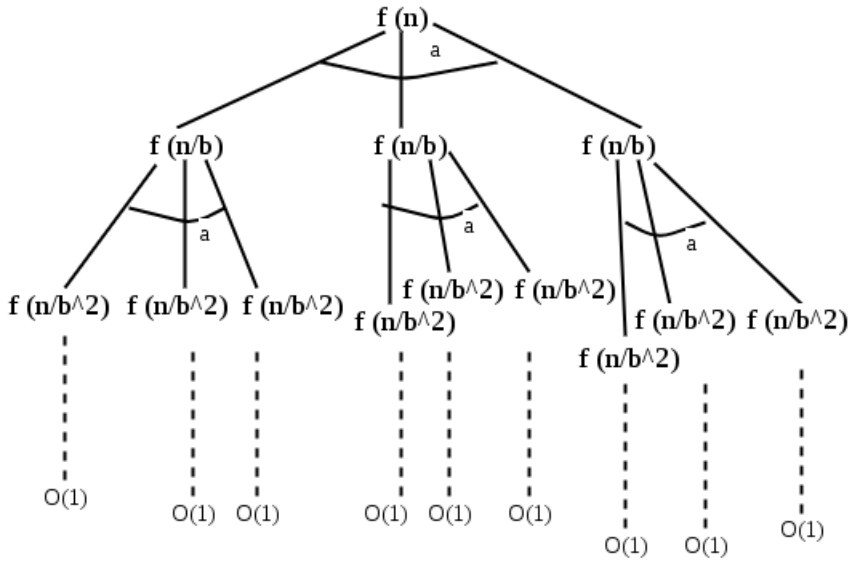
$T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$

There are the following three cases:

- If $f(n) = O(n^c)$ where $c < \text{Log}_b a$ then $T(n) = \Theta(n^{\text{Log}_b a})$
- If $f(n) = \Theta(n^c)$ where $c = \text{Log}_b a$ then $T(n) = \Theta(n^c \text{Log } n)$
- If $f(n) = \Omega(n^c)$ where $c > \text{Log}_b a$ then $T(n) = \Theta(f(n))$

How does this work?

The master method is mainly derived from the recurrence tree method. If we draw the recurrence tree of $T(n) = aT(n/b) + f(n)$, we can see that the work done at the root is $f(n)$, and work done at all leaves is $\Theta(n^c)$ where c is $\text{Log}_b a$. And the height of the recurrence tree is $\text{Log}_b n$



In the recurrence tree method, we calculate the total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically the same, then our result becomes height multiplied by work done at any level (Case 2). If work done at the root is asymptotically more, then our result becomes work done at the root (Case 3).

Examples of some standard algorithms whose time complexity can be evaluated using the Master Method

- Merge Sort: $T(n) = 2T(n/2) + \Theta(n)$. It falls in case 2 as c is 1 and $Log_b a]$ is also 1. So the solution is $\Theta(n Logn)$
- Binary Search: $T(n) = T(n/2) + \Theta(1)$. It also falls in case 2 as c is 0 and $Log_b a$ is also 0. So the solution is $\Theta(Logn)$

Notes:

- It is not necessary that a recurrence of the form $T(n) = aT(n/b) + f(n)$ can be solved using Master Theorem. The given three cases have some gaps between them. For example, the recurrence $T(n) = 2T(n/2) + n/Logn$ cannot be solved using master method.
- Case 2 can be extended for $f(n) = \Theta(n^c Log^k n)$
  If $f(n) = \Theta(n^c Log^k n)$ for some constant $k >= 0$ and $c = Log_b a$, then $T(n) = \Theta(n^c Log^{k+1} n)$

For more details, please refer: Design and Analysis of Algorithms.

Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above

Similar Reads

1. How to Analyse Loops for Complexity Analysis of Algorithms
2. Time Complexity and Space Complexity
3. Practice Set for Recurrence Relations
4. Different types of recurrence relations and their solutions
5. What does 'Space Complexity' mean?
6. A Time Complexity Question
7. Time Complexity of building a heap
8. An interesting time complexity question
9. Time Complexity of Loop with Powers
10. Time Complexity of a Loop when Loop variable “Expands or Shrinks” exponentially

Related Tutorials

1. Learn Data Structures with Javascript | DSA Tutorial
2. Introduction to Max-Heap – Data Structure and Algorithm Tutorials
3. Introduction to Set – Data Structure and Algorithm Tutorials
4. Introduction to Map – Data Structure and Algorithm Tutorials
5. What is Dijkstra’s Algorithm? | Introduction to Dijkstra's Shortest Path Algorithm

Previous Next

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Medium

Easy | Normal | Medium | Hard | Expert

Improved By : Manish Dhanuka, animagussirius7, pragatpandya, shreelakshmijoshi1, janardansthox, snehalmahasagar

Article Tags : Complexity-analysis, Algorithms, Analysis, DSA

Practice Tags : Algorithms

Improve Article | Report Issue

GeeksforGeeks
A-143, 9th Floor, Sovereign Corporate Tower, Sector-136, Noida, Uttar Pradesh - 201305

feedback@geeksforgeeks.org

C++
GoLang
SQL
R Language
Android Tutorial

Linked List
Stack
Queue
Tree
Graph

Algorithms

Sorting
Searching
Greedy
Dynamic Programming
Pattern Searching
Recursion
Backtracking

Web Development

HTML
CSS
JavaScript
Bootstrap
ReactJS
AngularJS
NodeJS

Data Science & ML

Data Science With Python
Data Science For Beginner
Machine Learning Tutorial
Maths For Machine Learning
Pandas Tutorial
NumPy Tutorial
NLP Tutorial

Interview Corner

Company Preparation
Preparation for SDE
Company Interview Corner
Experienced Interview
Internship Interview
Competitive Programming
Aptitude

Python

Python Tutorial
Python Programming Examples
Django Tutorial
Python Projects
Python Tkinter
OpenCV Python Tutorial

GfG School

CBSE Notes for Class 8
CBSE Notes for Class 9
CBSE Notes for Class 10
CBSE Notes for Class 11
CBSE Notes for Class 12
English Grammar

UPSC/SSC/BANKING

SSC CGL Syllabus
SBI PO Syllabus
IBPS PO Syllabus

Write & Earn

Write an Article
Improve an Article
Pick Topics to Write