

Master Theorem and Recursion

Motivation, formal proof, and examples
by Eric Yang Xingyu

Contents

1. Background	2
1.1. Example of Recursive Algorithms	2
1.1.1. Binary Search	2
1.1.2. Karatsuba Quick Multiplication	4
1.2. Complexity Notations Recap	5
2. Common Methods for Analysing Recursive Algorithms	5
2.1. Linear Recurrence Relations	6
2.2. Recursion Tree analysis	7
2.3. Substitution Method	9
2.4. Mathematical Induction	10
3. Master Theorem: the Silver Bullet	12
3.1. Introduction to a Base Case	12
3.2. Generalisation of the Base Case	12
3.3. Example Use Cases	12

§1. Background

Recursive algorithms are algorithms that call themselves. They are often used to solve problems that can be broken down into smaller subproblems. While it is easy to write recursive algorithms, it is often difficult to analyze their time complexity. This is usually because recursive algorithms is not like iteratively implemented algorithms where you can easily count the number of iterations, and all you need to do is to multiply the time complexity of each iteration by the number of iterations. In recursive algorithms, the number of recursive calls is implied by the input size, and the time complexity of each call is not always the same, usually dependent on the complexity of the subproblems.

Definition 1.1 (Recursive Algorithm).

A recursive algorithm is an algorithm that calls itself to solve a problem by breaking it down into smaller subproblems. The time complexity of recursive algorithms can be expressed using recurrence relations, as the time complexity of the algorithm is often dependent on the time complexity of solving the subproblem(s).

§1.1. Example of Recursive Algorithms

We first examine several examples of recursive algorithms in order to understand the nuances to iterative approaches.

§1.1.1. Binary Search

We first look at the recursive implementation of the binary search algorithm.

Example (Binary Search).

The binary search algorithm is a classic example of a recursive algorithm.

Algorithm 1: Binary Search

```

1 function Binarysearch( $A, x, \text{low}, \text{high}$ )
2   if  $\text{low} > \text{high}$ 
3     return  $-1$ 
4   else
5      $\text{mid} = (\text{low} + \text{high}) / 2$ 
6     if  $A[\text{mid}] = x$ 
7       return  $\text{mid}$ 
8     else if  $A[\text{mid}] > x$ 
9       return Binarysearch( $A, x, \text{low}, \text{mid} - 1$ )
10    else
11      return Binarysearch( $A, x, \text{mid} + 1, \text{high}$ )

```

It works by dividing the input array in half and recursively searching the left or right half of the array depending on the value of the middle element. The time complexity of binary search is $O(\log n)$, where n is the size of the input array.

Problem 1.1.1.1.

How does the iterative and recursive implementation of Binary Search differ when it comes to analyzing the time complexity?

This is a trivial problem, but it is important to understand the difference between recursive and iterative algorithms. In iterative algorithms, the number of iterations is strictly defined and visible, while in recursive algorithms, the number of recursive calls is not always visible, and the only thing we know, given that the algorithm is correct, is that the recursive algorithm terminates when it reaches the base case.

Algorithm 2: Binary Search (iterative)

```

1 function Binarysearch_iter( $A, x, \text{low}, \text{high}$ )
2   while  $\text{low} \leq \text{high}$ 
3      $\text{mid} = (\text{low} + \text{high}) / 2$ 
4     if  $A[\text{mid}] = x$ 
5       return  $\text{mid}$ 
6     else if  $A[\text{mid}] > x$ 
7        $\text{high} = \text{mid} - 1$ 
8     else
9        $\text{low} = \text{mid} + 1$ 
10    return  $-1$ 

```

From the pseudocode itself, we know that the number of iterations is $\log_2 n$, because the algorithm divides the input array in half each iteration. Some may say, this is also trivial in the recursive case, while the reason is that this algorithm is dependent on a simple recurrence relation, $T(n) = T(\frac{n}{2}) + O(1)$, where $T(n)$ is the time complexity of the algorithm, and $O(1)$ is the time complexity of the operations in the algorithm, we may also use some random letter to denote these constant time operations, so we may also write $T(n) = T(\frac{n}{2}) + c$, where c is a constant.

Remark 1.1.1.1.

While how to express the constant component in the time complexity actually does not really matter, because time complexity is usually analysed asymptotically, and the constant component is usually ignored, and non-rigorously, we may just write $T(n) = T(\frac{n}{2}) + 1$.

And here we have some tricky problems:

Problem 1.1.1.2.

- How can we derive a closed-form solution for the recurrence relation like $T(n) = T(\frac{n}{2}) + c$?
- How can we transfer the time complexity $T(n)$ to big O notation?

These will be answered in the following sections of the article.

§1.1.2. Karatsuba Quick Multiplication

We introduce the other example to be discussed in this article, the Karatsuba multiplication algorithm. This algorithm is a fast multiplication algorithm that multiplies two numbers in $O(n^{\log_2 3})$ time complexity, which is faster than the naive multiplication algorithm that has a time complexity of $O(n^2)$.

Example (Karatsuba Multiplication).

The Karatsuba multiplication algorithm is a recursive algorithm that multiplies two numbers in $O(n^{\log_2 3})$ time complexity. The algorithm works by dividing the input numbers into two halves and recursively multiplying the subparts of the numbers. The time complexity of the Karatsuba multiplication algorithm is derived from the recurrence relation $T(n) = 3T(\frac{n}{2}) + cn$.

Algorithm 3: Karatsuba Multiplication

```

1 function Karatsuba(x, y)
2   if  $x < 10$  or  $y < 10$ 
3     return  $x \cdot y$ 
4    $n = \max(\text{length}(x), \text{length}(y))$ 
5    $m = \lceil \frac{n}{2} \rceil$ 
6    $x_h, x_l = \text{split}(x, m)$ 
7    $y_h, y_l = \text{split}(y, m)$ 
8    $a = \text{Karatsuba}(x_h, y_h)$ 
9    $b = \text{Karatsuba}(x_l, y_l)$ 
10   $c = \text{Karatsuba}(x_h + x_l, y_h + y_l) - a - b$ 
11  return  $a \times 10^{2m} + c \times 10^m + b$ 

```

In this article, we do not discuss the details of the algorithm, but only derive the recurrence relation from the recursive implementation. If you want to know more about the algorithm, you can refer to [wikipedia](#), or the jupyter notebook under “quick multiplication” in this repository.

Theorem 1.1.2.1 (Karatsuba Multiplication Recurrence Relation).

The time complexity of the Karatsuba multiplication algorithm is $T(n) = 3T(\frac{n}{2}) + cn$, where $T(n)$ is the time complexity of the algorithm, and c is a constant, and n is the number of digits of the input numbers.

Proof. The Karatsuba multiplication algorithm computes the product of two n -digit numbers, x and y , by splitting each into two parts of approximately $\frac{n}{2}$ digits: Let $x = 10^{\frac{n}{2}} \cdot a + b$ and $y = 10^{\frac{n}{2}} \cdot c + d$, where a, b, c, d are numbers with $\frac{n}{2}$ digits (assuming n is even for simplicity).

Instead of performing four multiplications (as in the naive $O(n^2)$ method), Karatsuba reduces this to three multiplications of numbers of size $\frac{n}{2}$:

1. Compute $p_1 = a \cdot c$.
2. Compute $p_2 = b \cdot d$.
3. Compute $p_3 = (a + b) \cdot (c + d)$.

The product of x and y is then given by $10^n \cdot p_1 + 10^{\frac{n}{2}} \cdot (p_3 - p_1 - p_2) + p_2$.

Now we can derive the recurrence relation of exact time complexity of the algorithm.

In each of p_1 , p_2 , and p_3 , we multiply two numbers of size $\frac{n}{2}$, so the time complexity of each of these operations is $T(\frac{n}{2})$. Combining together, we have $3T(\frac{n}{2})$.

While we also need to consider the cost of non-recursive steps, which is $O(n)$, because we need to split the input numbers into two parts, and combine the results of the subproblems, which are introduced in line 4, 5, 6, and 7 of the pseudocode. But be careful that, we need to add cn instead of just n , because the cost of these operations is not always the same, and there are multiple operations that are of $O(n)$ in the non-recursive part of the algorithm.

Hence, we have rigorously justified the recurrence relation of Karatsuba's time complexity is

$$T(n) = 3T\left(\frac{n}{2}\right) + cn.$$

□

§1.2. Complexity Notations Recap

Before we dive into complexity analysis of recursive algorithms, let's recap the complexity notations we will use in this article. Note that we will only use Big O (particularly) and Big Omega notations, as they are the most commonly used notations in complexity analysis.

Definition 1.2.1 (Big O Notation).

Big O notation is used to describe the upper bound of the growth rate of a complexity function. Formally, $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that

$$\forall n \geq n_0, 0 \leq f(n) \leq c \cdot g(n).$$

Definition 1.2.2 (Big Omega Notation).

Big Omega notation is used to describe the lower bound of the growth rate of a complexity function. Formally, $f(n) = \Omega(g(n))$ if there exist positive constants c and n_0 such that

$$\forall n \geq n_0, 0 \leq c \cdot g(n) \leq f(n).$$

Average case complexity is not discussed in this article, as it is not commonly used in complexity analysis of recursive algorithms.

§2. Common Methods for Analysing Recursive Algorithms

Now we will try to solve the last problems we mentioned in Section 1.1.1.

We just recap the definitions of the recurrence relation and the time complexity of the algorithm, and we must first point out one crucial standing point in the analysis of recursive algorithms:

Proposition 2.1.

To analyze the time complexity of a recursive algorithm, we **must** derive a non-recursive

expression for the recurrence relation that describes the time complexity of the algorithm, which is bounded only by some variables n , where n is the size of the input.

This is because the time complexity of the algorithm is usually dependent on the time complexity of solving the subproblems, and the number of subproblems is usually dependent on the size of the input. So we must derive a non-recursive solution for the recurrence relation, which is a direct function of n . If we do not do this, the recursive definition cannot be used to argue for time complexity notations, because it is not a function of n , but a transition equation that is dependent on the complexity of subproblems, despite the fact that the $T(n)$ is indeed indirectly dependent on n . However, **this is not a solid argument to prove the time complexity of the algorithm, as per the definitions!**

§2.1. Linear Recurrence Relations

To facilitate the analysis of recursive algorithms, we first introduce the formal definition of a **linear recurrence relation with constant coefficients**.

Solving recurrence relation is a vast topic in mathematical context, and there are many methods to solve recurrence relations, such as substitution method, master theorem, generating function, characteristic equation, strong/weak induction, and so on. Yet in the context of fundamental algorithm analysis, we are only engaged with **first order linear recurrence relations with constant coefficients**. The name could be a bit intimidating, but the idea is simple.

Definition 2.1.1 (First Order Linear Recurrence Relation with Constant Coefficients).

A **linear recurrence relation with constant coefficients** is a recurrence relation of the form:

$$a_n = c_1 a_{n-k_1} + c_2 a_{n-k_2} + \dots + c_m a_{n-k_m} + f(n)$$

where:

- n is a non-negative integer (or, in some contexts, an integer).
- m is a positive integer representing the **order** of the recurrence.
- c_1, c_2, \dots, c_m are constant coefficients, with $c_m \neq 0$.
- k_1, k_2, \dots, k_m are distinct positive integer constants. Without loss of generality, we can assume $0 < k_1 < k_2 < \dots < k_m$.
- $f(n)$ is a function of n .

To uniquely define the sequence a_n , we also need m initial conditions:

$$\begin{aligned} a_{n_0} &= b_0 \\ a_{n_0+1} &= b_1 \\ &\dots \\ a_{n_0+m-1} &= b_{m-1} \end{aligned}$$

where n_0 is the smallest value of n for the recurrence, where b_0, b_1, \dots, b_{m-1} are constants.

- **Homogeneous Recurrence:** If $f(n)$ is the identically zero function (i.e., $f(n) = 0$ for all n), the recurrence is called **homogeneous**.

- **Non-homogeneous Recurrence:** If $f(n)$ is not the identically zero function, the recurrence is called **non-homogeneous**.
- **First-Order Linear Recurrence:** A **first-order** linear recurrence is a special case where $m = 1$ and $k_1 = 1$. It has the form: $a_n = ca_{n-1} + f(n)$ with **one** initial condition $a_{n_0} = b_0$

It's important to note that the choice of using a_n to represent the sequence, as opposed to $T(n)$ or any other notation (e.g., x_n , y_n), is purely a **matter of convention and symbolic representation**. The underlying mathematical structure and properties of the recurrence relation remain the same regardless of the specific symbol used to denote the sequence. The choice of notation often depends on context or personal preference. To avoid confusion over the first order linear recurrence relation, we specifically state that the recursive relation of

$$a_n = ca_{n-1} + f(n)$$

is equivalent to the recursive relation of

$$T(n) = cT\left(\frac{n}{k}\right) + f(n),$$

where $k \in \mathbb{N}^+$. It is also **important** that, in the sequence expression a_n , n is only an index to the sequence, and it is not necessarily the size of the input of the algorithm. In $T(n)$, however, n is the size of the input of the algorithm, that's why we have something like $T\left(\frac{n}{k}\right)$, but not $T(n-1)$, which is not meaningful in the context of algorithm analysis.

In the scope of FIT2004, we are only interested in the first-order linear recurrence relation with constant coefficients, which is a simple form of recurrence relation that can be solved using some normal methods, without introducing too many extra mathematical concepts.

Remark 2.1.1.

Recurrence Relation is an **INMENSE** topic in mathematics, and it is related to many other topics, such as generating functions, characteristic equations, differential equations, system of equations, eigenvalue & eigenvector, graph theory, dynamic system, vector space, polynomial rings, or even advanced topics in abstract algebra, such as group theory, ring theory, field theory! It is highly recommended to take it as a important topic in mathematics, and I really like Chapter 8 of the book “[Discrete Mathematics and its Applications](#)” by Kenneth H. Rosen, which is a good introduction to the topic. It will be an excellent start to figure out the connection between linear recurrence relations and ordinary differential equations.

With the notion of recurrence relation explained, we can now move on to the methods of solving recurrence relations, which are crucial in the analysis of recursive algorithms. But do note that, the method we discuss here is only applicable to the first-order linear recurrence relation with constant coefficients, and it is not applicable to higher-order linear recurrence relations, non-linear recurrence relations, which are more complex and require more advanced mathematical tools to solve, such as generative function.

§2.2. Recursion Tree analysis

One of the most intuitive methods to analyze the time complexity of basic recursive algorithms is by investigating the **recursion tree**. The recursion tree is a visual representation of the recursive calls made by the algorithm, which helps us understand the number of recursive calls and the time complexity of each call. In this method, we will try to find the time complexity of each call with respect to the input size, and then sum up the time complexity of all calls by traversing all levels of the recursion tree.

In the context of Karatsuba's algorithm, we can represent the recursive calls as a tree, where each level of the tree corresponds to a recursive call, and the branching factor is the number of subproblems created at each level. In this case, we have a single root $T(n)$ and three children $T(\frac{n}{2})$ at each level, so on and so forth. So this is actually a ternary tree.

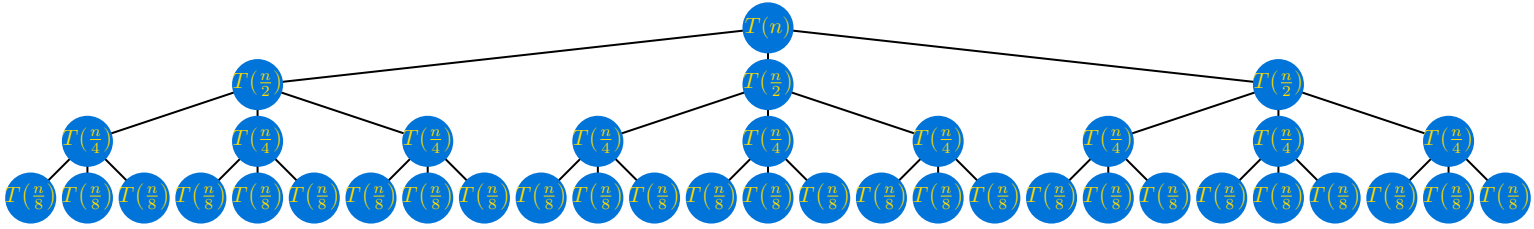


Figure 1: Recursion Tree of Karatsuba Algorithm up to 3 Levels

Above is the recursion tree of the Karatsuba multiplication algorithm up to 3 levels. Theoretically, we can continue to draw the recursion tree to the leaf nodes, where base cases are reached. Before we proceed to prove the worst-case time complexity of the Karatsuba algorithm, first recap on several facts on trees, that for a binary tree, we have 2^k nodes at level k , and the number of levels is $(\log_2 n) + 1$, where n is the number of nodes. We can extend this to a ternary tree, where we have 3^k nodes at level k , and the number of levels is $\log_2 n$.

Problem 2.2.1.

Prove that the worst-case time complexity of the Karatsuba algorithm is $O(n^{\log_2 3})$.

Remark 2.2.1.

For clarity and conciseness, we **assume that n is always even** and of the form 2^k , where k is a positive integer, else, we need to apply ceiling function to some variables, for example $\lceil \log_2 n \rceil$ and $\lceil \frac{n}{2^k} \rceil$. From a pragmatic perspective, and computer science practice, this is 100% acceptable. However, I have to mention that, from a mathematical perspective, this is not rigorous enough, and we need to consider the general case, where n is not necessarily a power of 2, and we need to consider the ceiling function!

Proof. As per what have been mentioned, for input size n , we have 3^k nodes at level k , and the number of levels is $\log_2 n$. Additionally, we know that the subproblem at level k is of size $\frac{n}{2^k}$, so the time complexity of one subproblem at level k is $T(\frac{n}{2^k})$. Hence, we have obtained all the information we need to figure out the total time complexity of the algorithm in a summation form instead of recursive form.

$$T(n) = 3T\left(\frac{n}{2}\right) + cn$$

and each subproblems have its recursive cost and non-recursive cost as well.

At level k , we have 3^k problems of size $\frac{n}{2^k}$ with total non-recursive cost $3^k \cdot c\left(\frac{n}{2^k}\right)$, where c is some constant.

For the recursive cost, it's a quite different case, because we know that the base case is $T(1) = 1$, and each problem invokes three subproblems of size $\frac{n}{2}$, so we have $3T\left(\frac{n}{2}\right)$. In the end, we will have a total recursive cost of $3^k T(1)$.

This is quite anti-intuitive, but it is true, because the base case is a constant time operation, and 3^k is the number of subproblems at level k , which is a constant number! Because in this case n , the input size, is approaching infinity, so however large 3^k is, we always have $k < \log_2 n < n$, so 3^k is regarded as a constant number.

Thus, we conclude that we only need to consider the sum of non-recursive costs in all levels to get $T(n)$

So we can sum up all costs by traversing all levels of the recursion tree, and we have:

$$T(n) = \sum_{k=0}^{(\log_2 n)} 3^k \left(c \frac{n}{2^k}\right) = cn \sum_{k=0}^{\log_2 n} \left(\frac{3}{2}\right)^k$$

Remark 2.2.2.

We are counting from 0 to $\log_2 n$, because the root node is at level 0, and the leaf nodes are at level $\log_2 n$, so the total number of levels is $\log_2 n + 1$.

This is a geometric series, and we know that the sum of a geometric series is $\sum_{i=0}^k a^i = \frac{a^{k+1}-1}{a-1}$, so we have:

$$\begin{aligned} T(n) &= cn \frac{3^{\log_2 n+1} - 1}{\frac{3}{2} - 1} = cn \frac{3^{\log_2 n+1} - 1}{\frac{1}{2}} \\ &= 2cn(3^{\log_2 n+1} - 1) = 2cn(3^{\log_2 n} \cdot 3 - 1) \\ &= 2cn(n^{\log_2 3} \cdot 3 - 1) \because n^{\log_a b} = b^{\log_a n} \text{ (by base change formula)} \\ &= 6cn^{\log_2 3} - 2cn \end{aligned}$$

Now we use the formal definition of Big O notation, to show that $T(n) = O(n^{\log_2 3})$:

$$T(n) \leq 6cn^{\log_2 3} - 2cn \leq 6cn^{\log_2 3},$$

as $n \in \mathbb{N}^+$, and $\exists c_0 \in \mathbb{R}^+$, $\exists n_0 \in \mathbb{N}^+$, $\forall n \geq n_0$, $0 \leq 6cn^{\log_2 3} - 2cn \leq 6cn^{\log_2 3}$, where $c_0 = 6c$, $n_0 = 1$, so we have proven that $T(n) = O(n^{\log_2 3})$. \square

As you see, the recursion tree method is quite intuitive, and we do not completely apply the properties of the recursive relation except for its recursive definition. Yet it is not always the case that we can solve the time complexity of the algorithm by recursion tree method, because the recursion tree method is only applicable to some simple recursive algorithms, where the number of subproblems is a constant number, and the size of the subproblems is a constant fraction of the input size. This method is usually more time-consuming than substitution method, which is more mechanical and less intuitive.

§2.3. Substitution Method

The other method to solve the time complexity of recursive algorithms is the substitution method. The substitution method is a more mechanical method, because it has almost the same treatment to all cases. It is also called “telescoping” method, because it is like telescoping a series, where we try to find a pattern in the time complexity of the algorithm, and then we use the base case and undetermined coefficients to derive an exact expression for the recurrence relation and thus prove the time complexity of the algorithm.

But in a nutshell, this method is quite similar to mathematical induction. You need to find a pattern and take an educated guess, but you donot need hypothesis, because you already have the base case, which can be used to derive the exact expression of the recurrence relation.

Proof. We iteratively enumerate the first few terms of the recurrence relation:

$$\begin{aligned}
 T(n) &= 3T(n/2) + cn \\
 &= 3[3T(n/4) + c(n/2)] + cn \\
 &= 9T(n/4) + (3/2)cn + cn \\
 &= 9[3T(n/8) + c(n/4)] + (3/2)cn + cn \\
 &= 27T(n/8) + (9/4)cn + (3/2)cn + cn \\
 &= \dots
 \end{aligned}$$

and we can generalise the pattern that:

$$T(n) = 3^k T\left(\frac{n}{2^k}\right) + cn \sum_{i=0}^{k-1} \left(\frac{3}{2}\right)^i,$$

which is quite similar to what we have derived in the recursion tree method.

Note that, the base case is $T(1) = O(1)$, which implies that $\frac{n}{2^k} = 1$, so $k = \log_2 n$, and we can substitute this to the expression above

$$T(n) = 3^{\log_2(n)} \cdot T(1) + cn \cdot [1 + (3/2) + (3/2)^2 + \dots + (3/2)^{\log_2(n)-1}].$$

The constant part of the expression does not depend on n , so when $n \rightarrow \infty$, the time complexity is dominated by the term $3^{\log_2 n} = n^{\log_2 3}$, which is the approximately $n^{1.585}$.

The last step of using the formal definition of Big O notation to prove that $T(n) = O(n^{\log_2 3})$ is the same as the recursion tree method, so we will not repeat it here.

□

§2.4. Mathematical Induction

Another method to establish the time complexity of the Karatsuba algorithm is mathematical induction (MI). While this approach is not typically the most efficient for *deriving* the time complexity of recursive algorithms from scratch—since it requires knowing the target complexity beforehand—it is an excellent tool for *verifying* a conjectured time complexity, such as $T(n) = O(n^{\log_2 3})$. In this case, we already suspect the complexity from the recursion tree and substitution methods, and induction provides a rigorous proof of correctness, especially when the target is known.

Proof. We use mathematical induction to prove that the worst-case time complexity of the Karatsuba algorithm satisfies $T(n) = O(n^{\log_2 3})$ for all n of the form $n = 2^m$, where m is a positive integer, consistent with our earlier assumption. The recurrence relation is $T(n) = 3T(\frac{n}{2}) + cn$, with the base case $T(1) = O(1)$.

Consider $n = 1$ (i.e., $m = 0$):

- $T(1)$ is a constant-time operation (e.g., multiplying two single-digit numbers), so $T(1) = d$ for some constant $d > 0$.
- We need to show $T(1) = O(1^{\log_2 3})$. Since $1^{\log_2 3} = 1$, and $T(1) = d \leq d \cdot 1 = d \cdot 1^{\log_2 3}$, the base case holds with a constant $c_0 \geq d$.

Assume the statement holds for all $n = 2^k$ where $k < m$, i.e., there exists a constant $c_0 > 0$ such that $T(n) \leq c_0 n^{\log_2 3}$ for all such n . (We'll determine c_0 later to ensure the inequality holds.)

We need to prove the statement for $n = 2^m$, i.e., $T(n) \leq c_0 n^{\log_2 3}$. Using the recurrence relation:

$$T(n) = 3T\left(\frac{n}{2}\right) + cn,$$

where $\frac{n}{2} = 2^{m-1}$. Since $\frac{n}{2} < n$ and $\frac{n}{2} = 2^{m-1}$ is a power of 2, we can apply the inductive hypothesis:

$$T\left(\frac{n}{2}\right) \leq c_0 \left(\frac{n}{2}\right)^{\log_2 3}.$$

Substitute this into the recurrence:

$$T(n) \leq 3 \cdot c_0 \left(\frac{n}{2}\right)^{\log_2 3} + cn.$$

Simplify the expression:

$$\left(\frac{n}{2}\right)^{\log_2 3} = n^{\log_2 3} \cdot \left(\frac{1}{2}\right)^{\log_2 3} = n^{\log_2 3} \cdot 2^{-\log_2 3}.$$

Since $2^{\log_2 3} = 3$, we have $2^{-\log_2 3} = \frac{1}{3}$. Thus:

$$T(n) \leq 3 \cdot c_0 n^{\log_2 3} \cdot \left(\frac{1}{3}\right) + cn = c_0 n^{\log_2 3} + cn.$$

We need $T(n) \leq c_0 n^{\log_2 3}$, so:

$$c_0 n^{\log_2 3} + cn \leq c_0 n^{\log_2 3}$$

must hold. This inequality is false unless the additional term cn is accounted for. Instead, we adjust our hypothesis to include a lower-order term or choose c_0 sufficiently large. Since $\log_2 3 \approx 1.585 > 1$, $cn = O(n^{\log_2 3})$, and for large n , $cn \leq c_1 n^{\log_2 3}$ for some constant c_1 . Thus:

$$T(n) \leq c_0 n^{\log_2 3} + cn \leq c_0 n^{\log_2 3} + c_1 n^{\log_2 3} = (c_0 + c_1) n^{\log_2 3}.$$

Choose c_0 large enough in the hypothesis (e.g., $c_0 \geq d + c$) to absorb lower-order terms across all n . To be precise, let's hypothesize a form that matches our earlier closed-form solution, such as $T(n) \leq 6cn^{\log_2 3} - 2cn$, and test it (as derived from the recursion tree: $T(n) = 6cn^{\log_2 3} - 2cn$):

- Try $T(n) \leq 6cn^{\log_2 3} - 2cn$.
- Base case: $T(1) = d \leq 6c \cdot 1 - 2c = 4c$, true if $c \geq \frac{d}{4}$.
- Inductive step:

$$T(n) \leq 3 \left[6c \left(\frac{n}{2}\right)^{\log_2 3} - 2c \left(\frac{n}{2}\right) \right] + cn = 3[6cn^{\log_2 3} \cdot 2^{-\log_2 3} - cn] + cn = 6cn^{\log_2 3} - 3cn + cn = 6cn^{\log_2 3} - 2cn,$$

which matches exactly.

Thus, $T(n) \leq 6cn^{\log_2 3} - 2cn \leq 6cn^{\log_2 3}$, and by the definition of Big O, $T(n) = O(n^{\log_2 3})$.

The induction confirms that $T(n) = O(n^{\log_2 3})$, consistent with the recursion tree and substitution methods. This approach leverages the recurrence directly and verifies our earlier result rigorously. \square

§3. Master Theorem: the Silver Bullet

Congratulations! If you have made it this far, after understanding the above-mentioned methods, you are now ready to learn the Master Theorem! The Master Theorem is a powerful tool for solving recurrence relations of a specific form, and it allows us to get the time complexity of an algorithm in a simple and straightforward way, without the need for complex derivations or proofs or calculations like what we have done in the previous sections. However, the most important part is not how the theorem is like and how we can use it, but why it works. We will first discuss the intuition behind the Master Theorem from a basic case, and then generalise it to the Master Theorem.

§3.1. Introduction to a Base Case

§3.2. Generalisation of the Base Case

§3.3. Example Use Cases