Master Theorem and Recurssion

Motivation, formal proof, and examples

by Eric Yang Xingyu

Contents

§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 1.1.1.1 (Binary Search).
The binary search algorithm is a classic example of a recursive algorithm.

Algorithm 1: Binary Search

```
1  function Binarysearch(A, x, low, high)
2      if low > high
3          return −1
4      else
5          mid = (low + high) / 2
6      if A[mid] = x
7          return mid
8      else if A[mid] > x
9          return Binarysearch(A, x, low, mid - 1)
10     else
11         return Binarysearch(A, x, mid + 1, 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, low, high)
2      while low <= high
3          mid = (low + high) / 2
4          if A[mid] = x
5          return mid
6          else if A[mid] > x
7              high = mid - 1
8          else
9              low = 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 1.1.2.1 (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

```
function Karatsuba(x, y)
    if x < 10 or y < 10
        return x · y
    n = max(length(x), length(y))
    m = ⌈n/2⌉
    x_h, x_l = split(x, m)
    y_h, y_l = split(y, m)
    a = Karatsuba(x_h, y_h)
    b = Karatsuba(x_l, y_l)
    c = Karatsuba(x_h + x_l, y_h + y_l) − a − b
    return a × 10^(2m) + c × 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

$$T(n) = 3T\left(\frac{n}{2}\right) + cn. \tag{1}$$

□

§1.2. Some Prereqiosites

Below are some important facts & definitions to be used for our proof and discussion.

§1.2.1. Complexity Notation

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 1.2.1.1 (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

$$\forall n \geq n_0, 0 \leq f(n) \leq c \cdot g(n). \tag{2}$$

Definition 1.2.1.2 (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

$$\forall n \geq n_0, 0 \leq c \cdot g(n) \leq f(n). \tag{3}$$

Average case complexity is not discussed in this article, as it is not commonly used in complexity analysis of recursive algorithms. But generally, we say that

$$T(f(n)) = \Theta(g(n)) \iff [T(f(n)) = O(g(n))] \wedge [T(f(n)) = \Omega(g(n))]. \tag{4}$$

Average compexity can be defined in different ways when the context varies, while in this occasion, this is not our main focus.

§1.2.2. Other Lemmas

We will also use some other lemmas to prove the time complexity of the Karatsuba algorithm. These lemmas are quite simple, and they are usually used in the analysis of recursive algorithms.

Lemma 1.2.2.1 (Geometric Series Sum).
The sum of a geometric series is given by:

$$\sum_{i=0}^{k} a^i = \frac{a^{k+1} - 1}{a - 1}. \tag{5}$$

This is a trivial fact, and could be easily obtained by induction. You may refer to other sources if you are confused.

The other lemma is about logarithm, and is a quite useful formula in the analysis of divide-and-conquer algorithms.

Lemma 1.2.2.2.
For any positive real numbers a, b, n, we have:

$$n^{\log_a b} = b^{\log_a n}. \tag{6}$$

Proof. By logarithm property, we have $n = a^{\log_a n}$ and $b = a^{\log_a b}$.

Substitute $n = a^{\log_a n}$ to the LHS, we have:

$$\text{LHS} = n^{\log_a b} = \left(a^{\log_a n}\right)^{\log_a b} = a^{\log_a n \cdot \log_a b}. \tag{7}$$

Substitute $b = a^{\log_a b}$ to the RHS, we have:

$$\text{RHS} = b^{\log_a n} = \left(a^{\log_a b}\right)^{\log_a n} = a^{\log_a b \cdot \log_a n}. \tag{8}$$

Thus LHS = RHS, which completes the proof. □

§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 arguement to prove the time complexity of the algorithm, as per the definitions**!

§2.1. First-order Linear Recurrence Relations with Constant Coefficients

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:

$$a_n = c_1 a_{n-k_1} + c_2 a_{n-k_2} + ... + c_m a_{n-k_m} + f(n) \tag{9}$$

where:

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

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

$$\begin{aligned} a_{n_0} &= b_0 \\ a_{n_0+1} &= b_1 \\ &... \\ a_{n_0+m-1} &= b_{m-1} \end{aligned} \tag{10}$$

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

$$a_n = ca_{n-1} + f(n) \tag{11}$$

is equivalent to the recursive relation of

$$T(n) = cT\left(\frac{n}{k}\right) + f(n), \tag{12}$$

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 $T(\frac{n}{k})$, 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 "Discrte 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. Recurssion Tree Method

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.

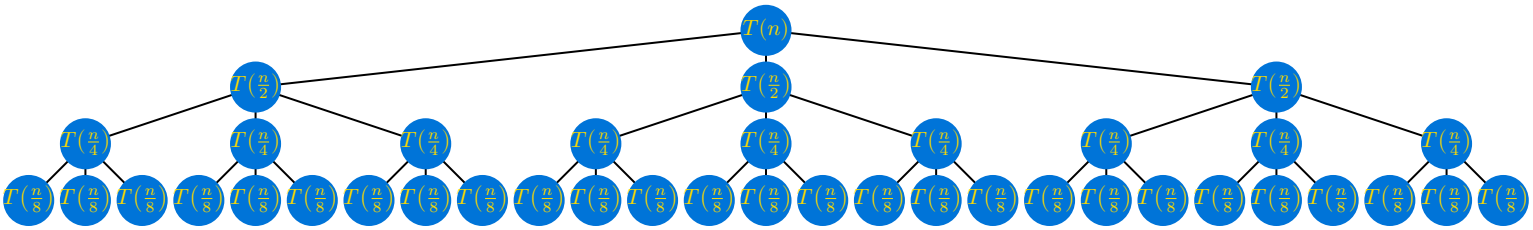

Figure 1: Recursion Tree of Karatsuba Algorithm up to 3 Levels

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 $O\big(n^{\log_2 3}\big)$.

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 $T\big(\frac{n}{2^k}\big)$. 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.

$$T(n) = 3T\left(\frac{n}{2}\right) + cn \tag{13}$$

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 $3^k \cdot c\big(\frac{n}{2^k}\big)$, 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 $3T\big(\frac{n}{2}\big)$. 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:

$$T(n) = \sum_{k=0}^{\log_2 n} 3^k \left(c\frac{n}{2^k}\right) = cn \sum_{k=0}^{\log_2 n} \left(\frac{3}{2}\right)^k \tag{14}$$

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:

$$\begin{aligned} T(n) &= cn\frac{3^{\log_2 n+1}-1}{\frac{3}{2}-1} = cn\frac{3^{\log_2 n+1}-1}{\frac{1}{2}} \\ &= 2cn\left(3^{\log_2 n+1}-1\right) = 2cn\left(3^{\log_2 n}\cdot 3-1\right) \\ &= 2cn\left(n^{\log_2 3}\cdot 3-1\right) \because n^{\log_a b} = b^{\log_a n} (\text{by base change formula}) \\ &= 6cn^{\log_2 3} - 2cn \end{aligned} \tag{15}$$

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

$$T(n) \le 6cn^{\log_2 3} - 2cn \le 6cn^{\log_2 3}, \tag{16}$$

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

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.

After reading through the proof, you can try out some more basic exercise to grasp the concept of recursion tree method. Below are some recommended exercises.

§2.2.1. Exercises

Exercise 2.2.1.1.
Use the recursion tree method to analyze the time complexity of Binary Search algorithm.

Exercise 2.2.1.2.
Derive the recurrence relation of the following recursive algorithm, and thus use the recursion tree method to analyze the time complexity of the following recursive algorithm:

```
1 function Recursive(n)
2 |  if n = 1
3 |  | return 1
4 |  else
5 |  | return Recursive(n/2) + Recursive(n/2)
```

Exercise 2.2.1.3.
Use the recursion tree method to analyze the time complexity of the following recursive algorithm:

```
1 function Recursive(n)
2 |  if n = 1
3 |  | return 1
4 |  else
5 |  | return 2 × Recursive(n/2)
```

Exercise 2.2.1.4.
Analyse the time complexity of the following recursive algorithm to generate the nth Fibonacci number.

```
1 function Fibonacci(n)
2 |  if n = 0 or n = 1
3 |  | return n
4 |  else
5 |  |  a = Fibonacci(n − 1)
6 |  |  b = Fibonacci(n − 2)
7 |  |  return a + b
```

Find the number of recursive calls made by the algorithm for a given input n (the number of nodes in the recursive tree), and prove that

$$T(n) = \Theta(\phi^n), \text{where } \phi = \frac{1+\sqrt{5}}{2}. \tag{17}$$

§2.3. Telescoping Method

The other method to solve the time complexity of recursive algorithms is the telescoping method. The method is a more mechanical method, because it has almost the same treatment to all cases. It is 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:

$$\begin{aligned} T(n) &= 3T(n/2) + cn \\ &= 3[3T(n/4) + c(n/2)] + cn \\ &= 9T(n/4) + (3/2)cn + cn \\ &= 9[3T(n/8) + c(n/4)] + (3/2)cn + cn \\ &= 27T(n/8) + (9/4)cn + (3/2)cn + cn \\ &= \cdots \end{aligned} \tag{18}$$

(19)

and we can generalise the pattern that:

$$T(n) = 3^k T\left(\frac{n}{2^k}\right) + cn \sum_{i=0}^{k-1} \left(\frac{3}{2}\right)^i, \tag{20}$$

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

$$T(n) = 3^{\log_2(n)} \cdot T(1) + cn \cdot [1 + (3/2) + (3/2)^2 + ... + (3/2)^{\log_2(n)-1}]. \tag{21}$$

The constant part of the expression does not depend on n, so when $n \to \infty$, 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 $T(n) = O\left(n^{\log_2 3}\right)$ is the same as the recursion tree method, so we will not repeat it here.

□

§2.3.1. Exercises

Below are some recommended exercises to practice the telescoping method. But you may also try to solve the problems using recursion tree method, or whatever else, and compare the results.

Exercise 2.3.1.1.
Use the telescoping method to analyze the time complexity of the following recursive algorithm:

```
1 function Recursive(n)
2 |  if n = 1
3 |  |  return 1
4 |  else
5 |  |  return 5 × Recursive(n/2) + n
```

Exercise 2.3.1.2.
Telescoping a closed-form expression for the following recursively defined time complexity:

$$T(n) = \begin{cases} T\left(\frac{n}{3}\right) + T\left(2\frac{n}{3}\right) + n^2 \text{ , if } n > 1 \\ 1 \text{ , if } n = 1 \end{cases} \tag{22}$$

and thus prove the worst case time complexity of the algorithm is $O(n \log n)$

§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 $T(n) = O\left(n^{\log_2 3}\right)$. 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 $T(n) = O\left(n^{\log_2 3}\right)$ for all n of the form $n = 2^m$, where m is a positive integer, consistent with our earlier assumption. The recurrence relation is $T(n) = 3T\left(\frac{n}{2}\right) + cn$, 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 $T(1) = O\left(1^{\log_2 3}\right)$. 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:

$$T(n) = 3T\left(\frac{n}{2}\right) + cn, \tag{23}$$

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:

$$T\left(\frac{n}{2}\right) \leq c_0 \left(\frac{n}{2}\right)^{\log_2 3}. \tag{24}$$

Substitute this into the recurrence:

$$T(n) \leq 3 \cdot c_0 \left(\frac{n}{2}\right)^{\log_2 3} + cn. \tag{25}$$

Simplify the expression:

$$\left(\frac{n}{2}\right)^{\log_2 3} = n^{\log_2 3} \cdot \left(\frac{1}{2}\right)^{\log_2 3} = n^{\log_2 3} \cdot 2^{-\log_2 3}. \tag{26}$$

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

$$T(n) \leq 3 \cdot c_0 n^{\log_2 3} \cdot \left(\frac{1}{3}\right) + cn = c_0 n^{\log_2 3} + cn. \tag{27}$$

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

$$c_0 n^{\log_2 3} + cn \leq c_0 n^{\log_2 3} \tag{28}$$

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 $\log_2 3 \approx 1.585 > 1$, $cn = O\big(n^{\log_2 3}\big)$, and for large n, $cn \leq c_1 n^{\log_2 3}$ for some constant c_1. Thus:

$$T(n) \leq c_0 n^{\log_2 3} + cn \leq c_0 n^{\log_2 3} + c_1 n^{\log_2 3} = (c_0 + c_1) n^{\log_2 3}. \tag{29}$$

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 $T(n) \leq a n^{\log_2 3} - bn$, 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:

$$T(n) \leq 3\left[6c\left(\frac{n}{2}\right)^{\log_2 3} - 2c\left(\frac{n}{2}\right)\right] + cn = 3\left[6cn^{\log_2 3} \cdot 2^{-\log_2 3} - cn\right] + cn = 6cn^{\log_2 3} - 3cn + cn = 6cn^{\log_2 3} - 2cn, \tag{30}$$

  which matches exactly.

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

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

Again, induction is quite handy for proving the time complexity of recursive algorithms. But only when it is possible to make conjecture on the time complexity of the algorithm, and the conjecture is correct, or in the context of exam, you are given the conjecture, and you need to prove it. It is not a good method to derive the time complexity of the algorithm from scratch, because it requires some intuition and educated guess.

§2.4.1. Exercises

Exercise 2.4.1.1.
Discuss the worst time complexity of the following recursive algorithm, and prove it using the induction method:

```
1 function Recursive(n)
2 |  if n = 1
3 |  | return 1
4 |  else
5 |  | return Recursive(n/2) + Recursive(n/3) + Recursive(n/6)
```

Hint: this is a different case where each subproblem has a different size, and therefore, we will have multiple summations in the final expression of the time complexity, which could not be combined. Take an educated guess, or the computation will be quite complex.

Exercise 2.4.1.2.
The merge sort algorithm has a time complexity of

$$\begin{cases} 2T(\frac{n}{2}) + n \text{ , if } n > 1 \\ 1 \text{ , if } n = 1 \end{cases} \tag{31}$$

Use the induction method to prove that the worst-case time complexity of the merge sort algorithm is $O(n \log n)$.

§2.5. Substitution Method

The last recommended method to solve the time complexity of recursive algorithms is the substitution method. It is somewaht similar to induction, but it is more mechanical and less intuitive. The method is based on the idea of making an educated guess of the time complexity of the algorithm, and then proving it by the definition of complexity notations by finding appropriate constants that satisfy the definitions.

Now we show how to prove the worst case time complexity of the Karatsuba algorithm using the substitution method.

Proof. We start by making an educated guess that the time complexity of the Karatsuba algorithm is $T(n) = O\big(n^{\log_2 3}\big)$. We then prove this by substitution.

We assume that $T(n) \leq cn^{\log_2 3}$ for some constant $c > 0$ and all n of the form $n = 2^m$, where m is a positive integer. This is our hypothesis.

We need to prove that $T(n) = O\big(n^{\log_2 3}\big)$ for all n of the form $n = 2^m$. We use the recurrence relation $T(n) = 3T(\frac{n}{2}) + cn$ and the base case $T(1) = O(1)$.

Substitute the hypothesis into the recurrence relation:

$$T(n) \leq 3c\left(\frac{n}{2}\right)^{\log_2 3} + cn = 3cn^{\log_2 3 - 1} + cn = cn^{\log_2 3}\left(\frac{3}{2}\right) + cn. \tag{32}$$

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

$$cn^{\log_2 3}\left(\frac{3}{2}\right) + cn \leq cn^{\log_2 3} \tag{33}$$

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 sufficiently large. Since $\log_2 3 \approx 1.585 > 1$, $cn = O\big(n^{\log_2 3}\big)$, and for large n, $cn \leq c_1 n^{\log_2 3}$ for some constant c_1. Thus:

$$T(n) \leq cn^{\log_2 3} + cn \leq cn^{\log_2 3} + c_1 n^{\log_2 3} = (c + c_1)n^{\log_2 3}. \tag{34}$$

Choose c large enough in the hypothesis (e.g., $c \geq c_1$) to absorb lower-order terms across all n. To be precise, let's hypothesize a form that matches □

Remark 2.5.1.
The technique we play here is a bit abstruce, but we are basically playing with inequality properties. By proper assumption and manipulation, we construct a structure that fits into the definition of Big O notation, and then we prove that the structure is correct by the definition of Big O notation.

§2.5.1. Exercises

Below are some recommended exercises to practice the substitution method.

Exercise 2.5.1.1.
Use the substitution method to analyze the time complexity of the following recursive algorithm:

```
1 function Recursive(n)
2 |  if n = 1
3 |  |  return 1
4 |  else
5 |  |  return 2 × Recursive(n/2) + n^2
```

Exercise 2.5.1.2.
It is known that an algorithm has a time complexity of $T(n) = T(n-2) + 3$, and $T(1) = T(2) = 1$. Use the substitution method to find the worst-case time complexity of the algorithm.

Hint: this is a very special first-order linear recurrence relation, and we need two base cases to solve it. But our method is still applicable, as long as it is a first-order linear recurrence relation.

§3. Master Theorem: the Silver Bullet

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.

To make the following contents more readable, we introduce *devide and counquer recurrence relation*. Some of the contents in this part are from Discrete Math and Its Applications by Kenneth H. Rosen [1]. We introduce one terminology from the book here, and we will use it in the following sections.

Definition 3.1 (Divide and Conquer Recurrence Relation).
A **divide and conquer recurrence relation** is a recurrence relation of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{35}$$

where:

- n is a non-negative integer (or, in some contexts, an integer).
- a and b are positive integers.
- $f(n)$ is a function of n, the size of the input.
- The recurrence relation describes the time complexity of a divide-and-conquer algorithm that divides the input of size n into a subproblems of size $\frac{n}{b}$ each, and combines the results of the subproblems in $f(n)$ time.
- In the context of algorithm analysis, we can assume that $n \mid b$, i.e., n is divisible by b, and thus $\frac{n}{b} \in \mathbb{N}^+$.

This is something we are very familiar with, because most of the recursive algorithms we have discussed in the previous sections are divide-and-conquer algorithms, such as the Karatsuba algorithm, the merge sort algorithm, the quicksort algorithm, and so on. The Master Theorem is a generalisation of the divide-and-conquer recurrence relation, and it is a powerful tool to solve the time complexity of these algorithms.

While in some cases, the problem is not evenly divided by the algorithm, so we have multiple $T(n)$ terms in the recurrence relation, to which the Master Theorem is not applicable. Our following discussion will be based on the assumption that all problems are evenly divided by the algorithm. Analising the time complexity of algorithms that do not evenly divide the problem is not something can be achived by the Master Theorem, and it requires more advanced mathematical tools, such as generating function.

§3.1. Introduction to a Base Case

From previous examples and exercises, we find that, for all evenly divided algorithms, we have the compexity of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{36}$$

where a and b are positive integers, and $f(n)$ is a function of n. We can assume that n is divisible by b, i.e., $\frac{n}{b} \in \mathbb{N}^+$. In most cases, we can assume that $f(n)$ is a polynomial function of n, and we can write it as $f(n) = n^c$, where c is a constant. Practically, f(n) has only one term, because in the context of algorithm analysis, we are only interested in the dominant term of the time complexity, and we can ignore the lower-order terms. So rigorously, n^c is the most significant term in $f(n)$.

It is also important to distinguish that, $aT\left(\frac{n}{b}\right)$'s cost is caused by recursive calls themselves, while $f(n)$ are incurred out size of the recurssie calls. In this section, we call it **non-recursive cost**. But note that they are also calculated recursively despite the name, because they are a part of $T(n)$, and $T(n)$ is recursively defined!

The core of **Master Theorem** is discuss the contribution of recursive cost and non-recursive cost to the time complexity of the algorithm. In this section, we only consider the case when $f(n)$ is constant, meaning that $f(n) = c$, where c is some constant.

Theorem 3.1.1.
Basic Master Theorem: let an increasing function $f(n)$ be given, and let a and b be positive integers, and they satisfy

$$f(n) = af\left(\frac{n}{b}\right) + c \tag{37}$$

n is divisible by b, and $a, b \in \mathbb{N}^+$, and c is a constant(positive real number). then

$$f(n) = \begin{cases} O(n^{\log_b a}) \text{ if } a > 1 \\ O(\log n) \text{ if } a = 1 \end{cases}. \tag{38}$$

Additionally, when $n = b^k$ and $a \neq 1$, where $k \in \mathbb{N}^+$,

$$f(n) = C_1 n^{\log_b a} + C_2, \tag{39}$$

where $C_1 = f(1) + \frac{c}{a-1}, C_2 = -\frac{c}{a-1}$.

Proof. Frist, recall that in the examples of the Karatsuba algorithm, we conclude that the recursive part of the total cost can be expressed as the complexity of the base case times by the number of recursive calls, which in this case is a^k, where a is the number of subproblems to be solved by one recursive call, and k is the number of levels of the recursion tree. This is derived by applyting the recursive definition until we reach the base case. As for the non-recursive part, similarly, can be written as the sum of a constant series multiplied by the number of levels of the recursion tree, which is $c\sum_{i=0}^{k-1} a^i$. With this motivation, consider let $n = b^k$, where $k \in \mathbb{N}^+$, and k is the number of levels of the recursion tree. We can write the total cost of the algorithm as:

$$f(n) = a^k f(1) + \sum_{i=0}^{k-1} a^i c = a^k f(1) + c\sum_{i=0}^{k-1} a^i. \tag{40}$$

This makes thing very simple, since we only need to discuss the case of $a = 1$ and $a > 1$ separately.

when $a = 1$: $\sum_{i=0}^{k-1} a^i = \sum_{i=0}^{k-1} 1 = k$, and thus we have

$$f(n) = f(1) + ck. \tag{41}$$

As per stated in the theorem, we have $n = b^k \Longrightarrow k = \log_b n$, therefore

$$f(n) = f(1) + c\log_b n. \tag{42}$$

Now we can discuss the worst-case time complexity of the algorithm when $a = 1$.

Since our base case is constant, we can conclude that $f(n) = O(\log n)$ when $a = 1$. Rigorously,

$$\forall n \in \mathbb{N}^+, \exists c_0 \in \mathbb{R}^+, c_0 > c, \text{such that } f(n) \leq c_0 \log n. \tag{43}$$

Hence, $T(f(n)) = O(\log n)$ when $a = 1$.

Remark 3.1.1.
(Optional) Just remark for discussing the case when n is not a power of b. In this case, we must have $b^k < n < b^{k+1}$, where k is a positive integer. We have assumed that $f(n)$ is a increasing function, so we have

$$f(n) \leq f\left(b^{k+1}\right) = f(1) + c(k+1) = f(1) + c + ck \leq f(1) + c + c\log_b n. \tag{44}$$

This also helps to conclude that $f(n) = O(\log n)$ when $a = 1$. Rigorously,

$$\forall n \in \mathbb{N}^+, \exists c_0 \in \mathbb{R}^+, c_0 > c, \text{such that } f(n) \leq c_0 \log n. \tag{45}$$

Hence, $T(f(n)) = O(\log n)$ when $a = 1$.

when a > 1: we still assume that $n = b^k$, where $k \in \mathbb{N}^+$, and k is the number of levels of the recursion tree. Recall that $f(n) = a^k f(1) + c\sum_{i=0}^{k-1} a^i$, and we have $\sum_{i=0}^{k-1} a^i = \frac{a^k-1}{a-1}$ because it's a geometric series of common ratio a. Therefore, we have

$$\begin{aligned}
f(n) &= a^k f(1) + c\frac{a^k - 1}{a-1} \\
&= a^k f(1) + \frac{ca^k}{a-1} - \frac{c}{a-1} \\
&= a^k\left(f(1) + \frac{c}{a-1}\right) - \frac{c}{a-1} \\
&= a^{\log_b n}\left(f(1) + \frac{c}{a-1}\right) - \frac{c}{a-1} \\
&= n^{\log_b a}\left(f(1) + \frac{c}{a-1}\right) - \frac{c}{a-1} \\
&= C_1 n^{\log_b a} + C_2
\end{aligned} \tag{46}$$

where $C_1 = f(1) + \frac{c}{a-1}$, $C_2 = -\frac{c}{a-1}$. We can easily derive the worst case time complexity of the algorithm when $a > 1$:

$$\forall n \in \mathbb{N}^+, \exists C_0 \in \mathbb{R}^+, C_0 > C_1, \text{such that } f(n) \leq C_0 n^{\log_b a}, \tag{47}$$

and hence, $T(f(n)) = O\left(n^{\log_b a}\right)$ when $a > 1$.

Remark 3.1.2.
(Optional) Again we just want to show the treatment of the case when n is not a power of b, for those who are interested. In this case, we must have $b^k < n < b^{k+1}$, where k is a positive integer. By our assumption that $f(n)$ is an increasing function, we have

$$\begin{aligned}
f(n) &\leq f\left(b^{k+1}\right) = C_1 a^{k+1} + C_2 \\
&\leq (C_1 a) a^{\log_b n} + C_2 \\
&= (C_1 a) n^{\log_b a} + C_2
\end{aligned} \tag{48}$$

where $k \leq \log_b n < k + 1$.

Again, by the formal definition of Big O notation

$$\forall n \in \mathbb{N}^+, \exists C_0 \in \mathbb{R}^+, C_0 > C_1, \text{such that } f(n) \leq C_0 n^{\log_b a}. \tag{49}$$

Hence, $T(f(n)) = O\left(n^{\log_b a}\right)$ when $a > 1$.

To conclude, we have proven all the statements in the theorem. □

Below is a optional exercise to check your understanding.

Problem 3.1.1.
Prove the same result using mathematical induction.

Now we can reflect on the proof of this basic case of the Master Theorem. The proof is quite simple, and it is based on the intuition that the time complexity of the algorithm is recursively defined and we can find an closed-form, exact expression for the divide-and-conquer recurrence relation.

§3.2. Generalisation of the Theorem

Now we can try to take the previous results to a broader context. Previously, we considered only the case when the non-recursive component is only a constant function of n. But in practice, the non-recursive component can be a polynomial function of n, and in the previous definition of divide-and-conquer recurrence relation that:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n), \tag{50}$$

we can generalise to $f(n) = cn^d$, where $c, d \in \mathbb{R}^+$ and are constants, Thus, the previous case is a special case of this general case, where $d = 0$, so that we have a constant function of $f(n) = c$.

Theorem 3.2.1.
Master Theorem: Let an increasing function $f(n)$ be given, and let a and b be positive integers, and they satisfy

$$f(n) = af\left(\frac{n}{b}\right) + cn^d \tag{51}$$

n is divisible by b, and $a, b \in \mathbb{N}^+$, and $c, d \in \mathbb{R}^+$, and c, d are constants. Then

$$f(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d. \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases} \tag{52}$$

Proof. First recall the treatment of $T(n)$ in the previous sections, where we applied recursion tree method. We can apply the same treatment to the general case of the recursive complexity applicable in the Master Theorem.

We can derive the exact form of the recurrence by telescoping the recursive relation:

$$f(n) = af\left(\frac{n}{b}\right) + cn^d. \tag{53}$$

When $n := \frac{n}{b}$,

$$f\left(\frac{n}{b}\right) = af\left(\frac{n}{b^2}\right) + c\left(\frac{n}{b}\right)^d. \tag{54}$$

Substitute Eq. 54 to Eq. 53, we have:

$$f(n) = a\left[af\left(\frac{n}{b^2}\right) + c\left(\frac{n}{b}\right)^d\right] + cn^d = a^2 f\left(\frac{n}{b^2}\right) + c\frac{n^d}{b^d} + cn^d. \tag{55}$$

Continue this pattern until we are at the k th level of the recursion tree, we have $k = \log_b n$ so $\frac{n}{b^k} = 1$ (actually, but I prefer to keep this form anyway), and the base case is reached, we have:

$$f(n) = a^k f\left(\frac{n}{b^k}\right) + c\sum_{i=0}^{k-1} n^d \left(\frac{a}{b^d}\right)^i. \tag{56}$$

Note that b^d is also a constant as $b, d \in \mathbb{R}^+$, so we can split it from the summation and get a sum of geometric series, and we have:

$$f(n) = a^k f\left(\frac{n}{b^k}\right) + cn^d \sum_{i=0}^{k-1} \left(\frac{a}{b^d}\right)^i. \tag{57}$$

The geometric series is quite noticeable here, as it is a ratio of a and b^d, which are exactly the variables we discussed by cases in the theorem.

We first discuss the special case when $\boldsymbol{a = b^d}$. In this case, the geometric series is a constant, and we have:

$$f(n) = b^{dk} f(1) + cn^d \sum_{i=0}^{k-1} 1 = b^{dk} f(1) + cn^d k. \tag{58}$$

Since $k = \log_b n$, we have $b^{dk} = b^{\log_b n^d} = n^d$, hence

$$f(n) = n^d f(1) + n^d c \log_b n = n^d (f(1) + c \log_b n). \tag{59}$$

Since $f(1)$, the cost of base case, is a constant,

$$\forall n \in \mathbb{N}^+, \exists c_0 \in \mathbb{R}^+, c_0 > cn^d, \text{such that } f(n) \leq c_0 n^d \log n. \tag{60}$$

Hence conclude that $f(n) = O(n^d \log n)$ when $a = b^d$.

In the rest of the cases, the geometric series still plays an important rule in the inference, as it either converge to a constant or diverge to infinity, depending on the ratio of a and b^d.

Next, we will discuss the case when $\boldsymbol{a < b^d}$. In this case, the geometric series converges to a constant,

> **Remark 3.2.1.**
> For those who have diffuculty in understanding the convergence of the sum of the series, here is a unrigorous explaination. When $a < b^d$, as i goes to larger, $\frac{a}{b^d}$ goes smaller, and therefore, when we sum up the series, it tends to converge to a constant. This is a property of geometric series, and it is a well-known fact in mathematics.

so we can actually combine the summation $\sum_{i=0}^{k-1} \left(\frac{a}{b^d}\right)^i$ directly to the constant c, and we have:

$$f(n) = a^k f(1) + cn^d. \tag{61}$$

Again, we substitute $k = \log_b n$:

$$f(n) = a^{\log_b n} f(1) + cn^d = n^{\log_b a} f(1) + cn^d. \tag{62}$$

Since n is the only variable in the expression, let constants $C_1 = f(1)$, $C_2 = \log_b a$,

$$f(n) = C_1 n^{C_2} + cn^d = n^d\big(C_1 n^{C_2 - d} + c\big). \tag{63}$$

Hence,

$$\forall n \in \mathbb{N}^+, \exists C_0 \in \mathbb{R}^+, C_0 > C_1 n^{C_2 - d} + c, \text{such that } f(n) \le C_0 n^d. \tag{64}$$

Therefore, $f(n) = O\big(n^d\big)$ when $a < b^d$.

Finally, we discuss the case when $\boldsymbol{a > b^d}$. In this case, the geometric series diverges to infinity, because $\frac{a}{b^d} > 1$, and we have:

$$f(n) = a^k f(1) + cn^d \sum_{i=0}^{k-1} \left(\frac{a}{b^d}\right)^i. \tag{65}$$

We may apply Lemma 1.2.2.1 to the series to get the closed-form expression of the complexity function, and we have:

$$\sum_{i=0}^{k-1} \left(\frac{a}{b^d}\right)^i = \frac{\left(\frac{a}{b^d}\right)^k - 1}{\left(\frac{a}{b^d}\right) - 1}. \tag{66}$$

We know $k = \log_b n$, so:

$$\sum_{i=0}^{k-1} \left(\frac{a}{b^d}\right)^i = \frac{\left(\frac{a}{b^d}\right)^{\log_b n} - 1}{\frac{a}{b^d} - 1}. \tag{67}$$

Now we can substitute Eq. 67 to Eq. 65, and we have:

$$f(n) = a^k f(1) + \frac{1}{\frac{a}{b^d} - 1} cn^d \left(\left(\frac{a}{b^d}\right)^{\log_b n} - 1\right). \tag{68}$$

Note that $\big(b^d\big)^{\log_b n} = n^d$, and $a^{\log_b n} = n^{\log_b a}$, so we have:

$$\begin{aligned} f(n) &= n^{\log_b a} f(1) + \frac{cn^d}{\frac{a}{b^d} - 1}\left(\frac{n^{\log_b a}}{n^d} - 1\right) \\ &= n^{\log_b a} f(1) + \frac{cn^d}{\frac{a}{b^d} - 1}\big(n^{\log_b a - d} - 1\big). \end{aligned} \tag{69}$$

Again, let constants $C_1 = f(1)$, $C_2 = \frac{cn^d}{\frac{a}{b^d} - 1}$,

$$\begin{aligned} f(n) &= C_1 n^{\log_b a} + C_2\big(n^{\log_b a - d}\big) - C_2 \\ &= C_1 n^{\log_b a} + C_2 n^{\log_b a} - C_2 n^d - C_2 \\ &= (C_1 + C_2) n^{\log_b a} - C_2\big(n^d + 1\big) \end{aligned} \tag{70}$$

Hence we have

$$f(n) = (C_1 + C_2)n^{\log_b a} - C_2\big(n^d + 1\big) \leq (C_1 + C_2)n^{\log_b a}. \tag{71}$$

Therefore,

$$\forall n \in \mathbb{N}^+, \exists C_0 \in \mathbb{R}^+, C_0 > C_1 + C_2, \text{such that } f(n) \leq C_0 n^{\log_b a}. \tag{72}$$

Thus, we have $f(n) = O\big(n^{\log_b a}\big)$ when $a > b^d$. □

§3.3. Example Use Cases

§3.3.1. Binary Search

§3.3.2. Karatsuba Algorithm

Bibliography

[1] K. Rosen, *Discrete Mathematics and Its Applications*, 8th ed. New York (N.Y.): McGraw-Hill Education, 2018.