Recursive Algorithm Complexity Analysis

Effective method, recursion relation and more

Quang Nguyen, Cuong Dang[1]

[1]CS Department
UIT VNU-HCMC

Complexity Analysis Tutorial

Overview of Asymptotic Notation

Why Asymptotic Notation?

Asymptotic notation allows us to express the growth of algorithms in terms of input size n as it becomes large. This helps to:

- Abstract away constants and lower-order terms.
- Focus on the most significant factors influencing runtime.
- Compare algorithms based on their efficiency.

Big-O Notation

Big-O ($\mathcal{O}$) is used to describe the upper bound of an algorithm's time or space complexity. It provides a worst-case scenario. Formally,

$$f(n) = \mathcal{O}(g(n)) \text{ if there exist constants } c > 0 \text{ and } n_0 \geq 0 \text{ such that for all } n \geq n_0,$$

$$f(n) \leq c \cdot g(n).$$

Example: The time complexity of merge sort is $\mathcal{O}(n \log n)$, meaning it will never exceed this bound as input size grows.

Omega Notation

Omega (Ω) notation describes the lower bound of an algorithm's time or space complexity, representing the best-case scenario. Formally,

$$f(n) = \Omega(g(n)) \text{ if there exist constants } c > 0 \text{ and } n_0 \geq 0 \text{ such that for all } n \geq n_0,$$

$$f(n) \geq c \cdot g(n).$$

Example: The best-case time complexity of merge sort is $\Omega(n \log n)$.

Theta Notation

Theta (Θ) provides a tight bound, representing both the upper and lower bounds of an algorithm's growth rate. Formally,

$$f(n) = \Theta(g(n)) \text{ if } f(n) = \mathcal{O}(g(n)) \text{ and } f(n) = \Omega(g(n)).$$

Example: The time complexity of merge sort is $\Theta(n \log n)$, since it has both a lower and upper bound of this order.

What is a Recurrence Relation?

A recurrence relation is a formula that permits us to compute the members of a sequence one after another, starting with one or more given values.

- It expresses each term as a function of its preceding term(s).
- Often requires initial values to uniquely determine the sequence.

Example: First-order Recurrence Relation

Consider the recurrence relation:

$$x_{n+1} = c \cdot x_n \quad (n \geq 0, x_0 = 1) \tag{1.4.1}$$

This relation tells us that:

$$x_1 = c \cdot x_0, \quad x_2 = c \cdot x_1, \quad \dots, \quad x_n = c^n$$

- The sequence is $x_0 = 1, x_1 = c, x_2 = c^2, \dots$
- This is a first-order recurrence relation because each value depends only on the immediately preceding value.

Second-order Recurrence Relation

Now consider a second-order recurrence relation:

$$x_{n+1} = x_n + x_{n-1} \tag{1.4.2}$$

- This relation requires two starting values (e.g., x_0 and x_1) to generate the sequence.
- Without specifying these starting values, the sequence is not uniquely determined.

Key Observations

- In a recurrence relation, the number of previous terms used determines the order (e.g., first-order, second-order).
- Initial conditions are crucial for determining the sequence uniquely.
- Recurrence relations are often used in algorithm analysis and solving difference equations.

Problem Statement

We want to solve the first-order inhomogeneous recurrence relation:

$$x_{n+1} = b_{n+1}x_n + c_{n+1} \quad (n \geq 0, x_0 \text{ given})$$

where $b_1, b_2, \ldots$ and $c_1, c_2, \ldots$ are given sequences. Our goal is to find the sequence x_n.

Motivation and Strategy

Trying to guess the solution by writing out the first few terms is impractical:

$$x_1 = b_1 x_0 + c_1, \quad x_2 = b_2 b_1 x_0 + b_2 c_1 + c_2, \ldots$$

Instead, we use a more systematic approach:

1. Make a change of variables to reduce the complexity of the recurrence.
2. Solve the resulting simpler recurrence relation.
3. Revert to the original variable to find the solution.

Change of Variables

Define a new unknown sequence y_n as:

$$x_n = (b_1 b_2 \dots b_n) y_n \quad (n \geq 1, x_0 = y_0)$$

Substituting into the recurrence:

$$b_1 b_2 \dots b_{n+1} y_{n+1} = b_{n+1} b_1 b_2 \dots b_n y_n + c_{n+1}$$

After dividing both sides by $b_1 b_2 \dots b_{n+1}$, we get:

$$y_{n+1} = y_n + d_{n+1} \quad \text{where} \quad d_{n+1} = \frac{c_{n+1}}{b_1 b_2 \dots b_{n+1}}$$

Solving the Simpler Recurrence

Now we solve the simplified recurrence:

$$y_{n+1} = y_n + d_{n+1}$$

The solution is:

$$y_n = y_0 + \sum_{j=1}^{n} d_j$$

Thus, we have the solution for y_n.

Final Solution

Returning to the original variable x_n, we substitute back:

$$x_n = (b_1 b_2 \dots b_n)\left(x_0 + \sum_{j=1}^{n} d_j\right)$$

where:

$$d_j = \frac{c_j}{b_1 b_2 \dots b_j}$$

This provides the general solution to the first-order inhomogeneous recurrence.

Example: Solving a Recurrence

Consider the example:

$$x_{n+1} = 3x_n + n \quad (n \geq 0, x_0 = 0)$$

Using the change of variables $x_n = 3^n y_n$, we get:

$$y_{n+1} = y_n + \frac{n}{3^{n+1}}$$

Summing gives:

$$y_n = \sum_{j=0}^{n-1} \frac{j}{3^{j+1}}$$

Finally, revert to $x_n = 3^n y_n$ for the solution.

Problem Statement

We consider linear second-order homogeneous recurrence relations of the form:

$$x_{n+1} = ax_n + bx_{n-1} \quad (n \geq 1, x_0 \text{ and } x_1 \text{ given})$$

The goal is to find the sequence x_n. The key idea is to look for solutions of the form $x_n = \alpha^n$.

Solving the Recurrence

Substitute $x_n = \alpha^n$ into the recurrence relation:

$$\alpha^{n+1} = a\alpha^n + b\alpha^{n-1}$$

Dividing by α^{n-1}, we get the **characteristic equation**:

$$\alpha^2 = a\alpha + b$$

This quadratic equation determines the nature of the solution depending on the roots of α.

Three Cases for the Roots

There are three possible cases for the characteristic equation:

1. **Two distinct real roots**: The characteristic equation has two distinct roots α_1 and α_2.
$$x_n = c_1\alpha_1^n + c_2\alpha_2^n$$
2. **Repeated root**: The characteristic equation has a double root α.
$$x_n = (c_1 + c_2 n)\alpha^n$$
3. **Complex roots**: The characteristic equation has complex roots $\alpha = re^{i\theta}$.
$$x_n = r^n(c_1\cos(n\theta) + c_2\sin(n\theta))$$

In all cases, c_1 and c_2 are constants determined by initial conditions.

Example: Fibonacci Sequence

Consider the recurrence relation for the Fibonacci numbers:

$$F_{n+1} = F_n + F_{n-1} \quad (n \geq 1, F_0 = 1, F_1 = 1)$$

The characteristic equation is:

$$\alpha^2 = \alpha + 1$$

Solving, we find the roots:

$$\alpha_1 = \frac{1+\sqrt{5}}{2}, \quad \alpha_2 = \frac{1-\sqrt{5}}{2}$$

Thus, the general solution is:

$$F_n = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^n$$

Determining the Constants

We use the initial conditions $F_0 = 1$ and $F_1 = 1$ to solve for c_1 and c_2:

$$F_0 = 1 = c_1 + c_2$$

$$F_1 = 1 = c_1\alpha_1 + c_2\alpha_2$$

Solving this system of equations yields:

$$c_1 = \frac{\alpha_1}{\sqrt{5}}, \quad c_2 = -\frac{\alpha_2}{\sqrt{5}}$$

Thus, the explicit formula for the Fibonacci numbers is:

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

Summary

To solve second-order linear homogeneous recurrence relations:

1. Write the characteristic equation: $\alpha^2 = a\alpha + b$.
2. Solve the quadratic to find the roots α_1, α_2.
3. Depending on the nature of the roots (real distinct, repeated, or complex), write the general solution.
4. Use initial conditions to determine the constants c_1 and c_2.

Problem 1: Factorial

Description: The factorial of a number n is defined as the product of all positive integers up to n.

Recurrence Relation:

$$T(n) = T(n-1) + O(1)$$

Explanation:

- The problem size reduces by 1 in each recursive call.
- $O(1)$ time is needed to perform the multiplication at each step.

Solution: This recurrence resolves to $T(n) = O(n)$ because we have n recursive calls, each taking constant time.

Problem 2: Sum of the First n Numbers

Description: Compute the sum of the first n natural numbers using recursion.
Recurrence Relation:

$$T(n) = T(n-1) + O(1)$$

Explanation:

- The problem size decreases by 1 in each recursive call.
- $O(1)$ time is spent to add n to the result of $T(n-1)$.

Solution: This recurrence resolves to $T(n) = O(n)$ since each recursive call contributes a constant amount of work, and there are n recursive calls.

Problem 3: Counting Paths in a Grid

Description: Given a grid of size $m \times n$, count how many ways there are to move from the top-left corner to the bottom-right corner, using only moves to the right or down.
Recurrence Relation:

$$T(m,n) = T(m-1,n) + T(m,n-1)$$

Explanation:

- Each recursive call represents moving either down $(T(m-1,n))$ or right $(T(m,n-1))$.
- Base cases: $T(1,n) = 1$ (only one way to go right across the top row), and similarly for $T(m,1)$.

Solution: The complexity is $O\left(\binom{m+n-2}{m-1}\right)$, which is the number of unique paths, derived combinatorially.

Problem 4: Tower of Hanoi (Generalized Recurrence Analysis)

Description: In the Tower of Hanoi problem, the objective is to move n disks from one rod to another using a third rod.
Recurrence Relation:

$$T(n) = 2T(n-1) + O(1)$$

Explanation:

- The problem is reduced to solving two subproblems of moving $n-1$ disks.
- $O(1)$ time is needed to move the largest disk between the rods.

Solution: This recurrence expands to $T(n) = O(2^n)$ since the recurrence doubles at each level.

Problem 5: Minimum Cost to Reach the End of an Array

Description: Given an array of costs where each element represents the cost to step on that position, find the minimum cost to reach the end starting from the first position. You can move to the next or next-to-next element.
Recurrence Relation:

$$T(n) = \min(T(n-1), T(n-2)) + cost[n]$$

Explanation:

- At each step, you decide to either move 1 or 2 positions back.
- The time complexity depends on the minimum of the two previous positions and adds the current position's cost.

Solution: Solving this recurrence using dynamic programming gives $T(n) = O(n)$ as each step requires a constant time decision.

Problem 6: Catalan Numbers

Description: The Catalan numbers are a sequence of natural numbers that have applications in combinatorial mathematics, such as counting valid expressions of parentheses.

Recurrence Relation:

$$T(n) = \sum_{i=0}^{n-1} T(i) \cdot T(n-1-i)$$

Explanation:

- Each valid sequence of parentheses can be decomposed into two smaller subsequences.
- The problem size reduces based on combinations of smaller subproblems.

Solution: Solving this recurrence yields the closed-form $T(n) = \frac{1}{n+1}\binom{2n}{n}$, and the time complexity is $O(n^2)$ for the recursive solution.

Conclusion

Recurrence relations allow us to break down complex recursive problems into smaller, solvable subproblems. In some cases, we can directly expand or solve the recurrence without needing advanced tools like the Master Theorem.
Key Takeaway: Problems such as factorials, summations, counting paths, and others involve simple recursive structures, which lead to straightforward solutions.

Divide-and-Conquer Approach

Divide-and-Conquer is an algorithmic paradigm that divides a problem into smaller subproblems, solves the subproblems recursively, and combines their solutions to solve the original problem.
Typical steps:

- **Divide**: Break the problem into smaller, independent subproblems.
- **Conquer**: Solve each subproblem recursively.
- **Combine**: Merge the solutions of the subproblems to get the solution to the original problem.

Recurrence Relation

In Divide-and-Conquer algorithms, the time complexity can often be expressed using a recurrence relation. A typical form is:

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

where:

- a is the number of subproblems,
- n/b is the size of each subproblem,
- $f(n)$ is the cost of dividing the problem and combining the solutions.

Master Theorem

The Master Theorem provides an easy way to analyze the time complexity of Divide-and-Conquer algorithms. Given the recurrence relation

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

the solution can be determined based on the asymptotic behavior of $f(n)$:

- If $f(n) = \mathcal{O}(n^d)$ and $a > b^d$, then $T(n) = \mathcal{O}(n^{\log_b a})$.
- If $f(n) = \mathcal{O}(n^d)$ and $a = b^d$, then $T(n) = \mathcal{O}(n^d \log n)$.
- If $f(n) = \mathcal{O}(n^d)$ and $a < b^d$, then $T(n) = \mathcal{O}(n^d)$.

This allows us to quickly determine the time complexity based on a, b, and $f(n)$.

Example: Merge Sort

Merge sort is a classic example of a Divide-and-Conquer algorithm. The steps are:

- **Divide**: Split the array into two halves.
- **Conquer**: Recursively sort each half.
- **Combine**: Merge the two sorted halves to produce the sorted array.

The recurrence relation for Merge Sort is:

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

Applying the Master Theorem:

- Here, $a = 2$, $b = 2$, and $f(n) = \mathcal{O}(n)$.
- Since $a = b^1$, the second case of the Master Theorem applies.
- Thus, the time complexity of Merge Sort is $T(n) = \mathcal{O}(n \log n)$.

Example 1: Binary Search

Binary Search is a Divide-and-Conquer algorithm used to find the position of a target element in a sorted array.

Steps:

- **Divide**: Check the middle element of the array.
- **Conquer**: Recursively search in the left or right half, depending on the target.
- **Combine**: The solution is found when the target is found or the search space is exhausted.

Recurrence Relation:

$$T(n) = T\left(\frac{n}{2}\right) + \mathcal{O}(1)$$

Applying the Master Theorem:

- Here, $a = 1$, $b = 2$, and $f(n) = \mathcal{O}(1)$.
- Since $a < b^0$, the third case of the Master Theorem applies.
- Thus, the time complexity of Binary Search is $T(n) = \mathcal{O}(\log n)$.

Example 2: Strassen's Matrix Multiplication

Strassen's algorithm for matrix multiplication uses a Divide-and-Conquer approach to multiply two matrices faster than the standard algorithm.

Steps:

- **Divide**: Split each matrix into four submatrices of size $n/2 \times n/2$.
- **Conquer**: Recursively compute seven matrix products.
- **Combine**: Add or subtract the products to get the final matrix.

Recurrence Relation:

$$T(n) = 7T\left(\frac{n}{2}\right) + \mathcal{O}(n^2)$$

Applying the Master Theorem:

- Here, $a = 7$, $b = 2$, and $f(n) = \mathcal{O}(n^2)$.
- Since $a > b^2$, the first case of the Master Theorem applies.
- Thus, the time complexity of Strassen's algorithm is $T(n) = \mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2.81})$.

Example 3: Karatsuba's Algorithm (Multiplying Large Integers)

Karatsuba's algorithm is a Divide-and-Conquer algorithm used for multiplying large integers more efficiently than the standard method.

Steps:

- **Divide**: Split two n-digit numbers into two parts, each with $n/2$ digits.
- **Conquer**: Recursively compute three multiplications of smaller numbers.
- **Combine**: Use the results of the smaller multiplications to compute the product of the original numbers.

Recurrence Relation:

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

Applying the Master Theorem:

- Here, $a = 3$, $b = 2$, and $f(n) = \mathcal{O}(n)$.
- Since $a = b^1$, the second case of the Master Theorem applies.
- Thus, the time complexity of Karatsuba's algorithm is $T(n) = \mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1.585})$.

Example 4: QuickSort

QuickSort is a Divide-and-Conquer sorting algorithm. The steps are:

- **Divide**: Select a pivot element and partition the array around the pivot.
- **Conquer**: Recursively sort the two partitions.
- **Combine**: The array is sorted when both partitions are sorted.

Recurrence Relation (Average Case):

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

Applying the Master Theorem:

- Here, $a = 2$, $b = 2$, and $f(n) = \mathcal{O}(n)$.
- Since $a = b^1$, the second case of the Master Theorem applies.
- Thus, the average-case time complexity of QuickSort is $T(n) = \mathcal{O}(n \log n)$.

Advanced Example

Consider the recurrence:

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2 \tag{1}$$

In this case:

- $a = 4$: The problem is divided into 4 subproblems.
- $b = 2$: The size of each subproblem is $\frac{n}{2}$.
- $d = 2$: The cost of combining the subproblems is $O(n^2)$.

We can apply the Master Theorem by comparing a with b^d:

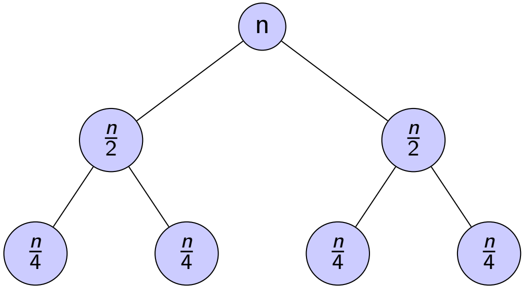
$$b^d = 2^2 = 4 \tag{2}$$

Since $a = b^d$, we fall under Case 2 of the Master Theorem, where:

$$T(n) = O(n^d \log n) = O(n^2 \log n) \tag{3}$$

Thus, the time complexity is $O(n^2 \log n)$.

Visual Representation of the Recurrence



Recursive subdivision of the problem into subproblems

Conclusion

In this example, we applied the Master Theorem to a recurrence with four subproblems, each of size $\frac{n}{2}$, and a quadratic cost for merging. By comparing a with b^d, we determined that the time complexity is $O(n^2 \log n)$.

Key Takeaway: The Master Theorem provides a quick and efficient way to analyze the time complexity of divide-and-conquer algorithms without solving the recurrence in full.

What is an Operator?

Definition: An operator T is a mapping from a vector space X to itself, i.e.,

$$T : X \to X$$

Example:

- For $X = \mathbb{R}^n$, the operator $T : \mathbb{R}^n \to \mathbb{R}^n$ could be a matrix multiplication $T(x) = Ax$ for some matrix $A \in \mathbb{R}^{n \times n}$.
- A differential operator, e.g., $T(f) = \frac{df}{dx}$ where f is a function.

Nonexpansive and Contractive Operators

Nonexpansive Operator: An operator T is called nonexpansive if for all $x, y \in X$,

$$\|T(x) - T(y)\| \leq \|x - y\|$$

That is, T does not "expand" the distance between any two points.
Contractive Operator: An operator T is contractive if there exists a constant $\alpha \in [0,1)$ such that for all $x, y \in X$,

$$\|T(x) - T(y)\| \leq \alpha \|x - y\|$$

Contractive operators "shrink" the distance between any two points.
Example:

- For $T(x) = \frac{1}{2}x$, the operator is contractive with $\alpha = \frac{1}{2}$.

Resolvent Operator

Definition: The resolvent of an operator T, denoted J_T, is defined as

$$J_T = (I + T)^{-1}$$

where I is the identity operator.
Interpretation: The resolvent is closely related to regularization, and it "smooths" out the application of the operator T.
Example: If $T(x) = x^2$, then $J_T = (I + T)^{-1}$ can be seen as solving $x + T(x) = y$ for some y.

Cayley Operator

Definition: The Cayley operator of T, denoted C_T, is given by

$$C_T = (I - T)(I + T)^{-1}$$

Properties:

- The Cayley operator can be used to map a nonexpansive operator to a different form.
- It is closely related to the resolvent operator.

Example: For a linear operator T such as $T(x) = \lambda x$, the Cayley transform becomes $C_T = \frac{1-\lambda}{1+\lambda} x$.

Picard Iteration

Definition: Picard Iteration is an iterative method for solving fixed-point equations of the form $T(x) = x$.
Algorithm: Given an initial guess x_0, iterate as follows:

$$x_{n+1} = T(x_n)$$

until convergence.
Convergence:

- If T is contractive, then Picard iteration converges to the unique fixed point of T.
- The rate of convergence is linear when T is contractive.

Example: For $T(x) = \frac{1}{2}(x+1)$, the Picard iteration would start with an initial guess and converge to the fixed point $x^* = 1$.

Applications of Picard Iteration

- Solving linear and nonlinear equations iteratively.
- Approximating fixed points in Banach spaces.
- Applications in differential equations, where Picard iteration can be used to find approximate solutions.

Conclusion

- Operators play a key role in various fields, including functional analysis and optimization.
- Nonexpansive and contractive operators define how distances between points are preserved or shrunk.
- The resolvent and Cayley operators provide tools for manipulating operators.
- Picard iteration is a powerful technique for solving fixed-point problems iteratively.

Key theorem for Fixed Iteration

Theorem

Assume $T:\mathbb{R}^n \to \mathbb{R}^n$ is θ-averaged with $\theta \in (0,1)$ and $\operatorname{Fix} T \neq \emptyset$. Then $x^{k+1} = Tx^k$ with any starting point $x^0 \in \mathbb{R}^n$ converges to one fixed point, i.e.,

$$x^k \to x^\star$$

for some $x^\star \in \operatorname{Fix} T$. The quantities $\operatorname{dist}(x^k, \operatorname{Fix} T)$, $\|x^{k+1} - x^k\|$, and $\|x^k - x^\star\|$ for any $x^\star \in \operatorname{Fix} T$ are monotonically nonincreasing with k. Finally, we have

$$\operatorname{dist}(x^k, \operatorname{Fix} T) \to 0$$

and

$$\|x^{k+1} - x^k\|^2 \leq \frac{\theta}{(k+1)(1-\theta)} \operatorname{dist}^2(x^0, \operatorname{Fix} T).$$

Gradient descent

Consider the problem:

$$\min_{x\in\mathbb{R}^n} f(x).$$

Assume f is CCP and differentiable. Then x is a solution if and only if

$$0 = \nabla f(x) \quad \Longleftrightarrow \quad x = (I - \alpha\nabla f)(x)$$

for any nonzero $\alpha \in \mathbb{R}$. In other words, x is a solution if and only if it is a fixed point of the operator $I - \alpha\nabla f$.
The FPI for this setup is:

$$x^{k+1} = x^k - \alpha\nabla f(x^k).$$

This algorithm is called the **gradient method** or **gradient descent**, and α is called the **stepsize**.

L-smoothness and convergence

Now assume f is **L-smooth**. By the cocoercivity inequality:

$$\|(I-(2/L)\nabla f)x-(I-(2/L)\nabla f)y\|^2 = \|x-y\|^2-\frac{4}{L}\langle x-y,\nabla f(x)-\nabla f(y)\rangle-\frac{1}{L}\|\nabla f(x)-\nabla f(y)\|^2 \leq \|x-$$

Therefore, $I-\alpha\nabla f$ is averaged for $\alpha \in (0,2/L)$ since:

$$I-\alpha\nabla f = (1-\theta)I+\theta(I-(2/L)\nabla f),$$

where $\theta = \alpha L/2 < 1$. Consequently, $x^k \to x^\star$ for some solution $x^\star$, if one exists, with rate:

$$\|\nabla f(x^k)\|^2 = O(1/k),$$

for any $\alpha \in (0,2/L)$.