Rigorous Analysis of Algorithms:
A Guided Study of Knuth's Section 1.2.10

1.2.10 Analysis of an Algorithm

The rigorous analysis of an algorithm refers to the study of its behavior as a function of its input size, with particular emphasis on estimating its resource usage — typically time or space — with provable guarantees.

Motivation

Why analyze an algorithm?

- To predict performance before implementation.
- To compare competing approaches in a machine-agnostic way.
- To determine worst-case, average-case, and best-case behaviors.
- To detect whether a solution is practical or merely theoretical.

While real-world performance also depends on low-level factors (e.g., CPU cache, compiler optimizations), mathematical analysis abstracts these away to focus on growth rates and scalability.

Definition of the Problem

Knuth presents this section using a concrete example: analyzing the average number of divisions performed by Euclid's algorithm (Algorithm E), which computes the greatest common divisor (GCD) of two integers m and n.

The question posed:

> Assuming that the value of n is fixed, and m ranges over all positive integers, what is the *average number of times* step E1 (division) is performed in Algorithm E?

Let T_n denote this average.

Formal Setup

To make T_n meaningful despite m ranging over infinitely many values, Knuth observes that:

> Once the algorithm begins, m is reduced modulo n. Thus, the only relevant values of m are $1 \leq m \leq n$.

So we redefine:

$$T_n = \frac{1}{n} \sum_{m=1}^{n} (\text{number of times step E1 is executed when Algorithm E is run on input } (m, n))$$

This reframing ensures that the analysis proceeds over a finite domain.

A Glimpse at the Outcome (for CR thinkers)

Knuth later references that T_n grows approximately as:

$$T_n \sim \frac{12 \ln 2}{\pi^2} \ln n$$

but this is a result that will be approached much later. For now, we develop the analytical tools needed to understand and derive such estimates.

Discussion: Analysis as a Discipline

Knuth carefully distinguishes between three major categories:

Algorithm Analysis:
Study of the *quantitative behavior* (e.g., running time, space usage) of a specific algorithm on well-defined inputs.

Algorithm Theory:
Addresses whether a problem can be solved algorithmically, regardless of resource usage.

Implementation:
Concerns actual performance, optimizations, hardware characteristics, and software engineering.

In this book, analysis is central: mathematical techniques, especially asymptotics and recurrences, will be used to analyze algorithm behavior with formal precision.

Theoretical Formalization (AS Mode)

An algorithm can be abstractly modeled as a deterministic transition system. Suppose that a computational method is defined by a tuple:

$$(Q, I, \Omega, f)$$

Where:

- Q is the set of all computational states.
- $I \subseteq Q$ is the set of valid input states.
- $\Omega \subseteq Q$ is the set of halting (output) states.
- $f : Q \to Q$ is the transition function, with the constraint:
$$f(q) = q \quad \text{for all } q \in \Omega$$

Given an input $x \in I$, we define a computational sequence:

$$x_0 = x, \quad x_{k+1} = f(x_k)$$

This sequence halts if $\exists k$ such that $x_k \in \Omega$. If so, we say the algorithm produces output x_k on input x.

This model lets us define what it means for a method to be *finite*, *deterministic*, and *effective*, thus grounding algorithm analysis in set-theoretic terms.

Computational Methods vs. Algorithms

A **computational method** may fail to halt for some inputs. For example, a reactive system may engage in infinite interaction with its environment. An **algorithm**, however, must always terminate for all valid inputs in a finite number of steps.

Knuth's Terminology:

- **Algorithm:** A computational method that is guaranteed to halt.
- **Program:** A syntactic representation of an algorithm in a formal language (e.g., MIX, machine code, C).

Formal Requirements of an Algorithm (Restated Precisely)

An algorithm must satisfy:

(1) **Finiteness:** The algorithm terminates on all valid inputs after a finite number of steps.

(2) **Definiteness:** Each instruction is precisely defined and unambiguous.

(3) **Input:** Zero or more externally supplied quantities from specified domains.

(4) **Output:** One or more results with a specified relation to the inputs.

(5) **Effectiveness:** Every operation is sufficiently basic to be carried out exactly, in a finite amount of time, by a human (or machine) using known procedures.

Example of Algorithm Analysis: Euclid's Algorithm (Revisited)

Let us restate the key question:

> For a fixed n, how many times is step E1 (division) executed on average across all $m \in \{1, 2, \ldots, n\}$?

The observation:

$$m \mod n \in \{0, 1, \ldots, n-1\}$$

So the first step reduces m to $r = m \bmod n$, making m effectively drawn from $\{1, \ldots, n\}$.

Thus:

$$T_n = \frac{1}{n} \sum_{m=1}^{n} C(m, n)$$

where $C(m, n)$ counts the number of divisions performed when computing $\gcd(m, n)$ via Algorithm E.

Preview of Later Derivation (CR Prompt)

Knuth mentions (without proof here) that for large n:

$$T_n \sim \frac{12 \ln 2}{\pi^2} \ln n$$

This stunning result reflects deep number-theoretic properties of the Euclidean algorithm, including its relation to continued fractions and Farey sequences.

CR Prompt: Try writing a program to compute T_n for n up to 1000 and plot it against $\ln n$. Do you see the logarithmic growth?

AS Prompt: Proving Correctness of GCD Algorithm

Exercise: Prove that Algorithm E indeed computes the $\gcd(m, n)$.

Hint: Show that $\gcd(m, n) = \gcd(n, r)$ where $r = m \bmod n$. Then argue by induction that the algorithm terminates with $\gcd(m, n)$ in variable n.

Transition to 1.2.11: Asymptotic Representations

The next section formalizes the tools needed for this type of analysis. Instead of focusing on exact counts, we turn to:

- Big-O: upper bounds on growth
- Theta: tight bounds
- Omega: lower bounds
- Euler's formula and integral approximations

These let us convert summations, recurrences, and discrete behavior into tractable asymptotic expressions — the language of algorithmic scalability.

1.2.11 Asymptotic Representations

In analyzing algorithms, exact expressions for resource usage (e.g., running time, memory) are often complex or intractable. Fortunately, many applications do not require precise values; we typically care more about how performance scales with input size.

This leads us to **asymptotic analysis** — the study of a function's growth rate as its input approaches infinity.

The most common notations used are:

- $\mathcal{O}$ (Big-O)
- Θ (Theta)
- Ω (Big-Omega)
- o (Little-o)
- ω (Little-omega)

These are formal tools that suppress constant factors and lower-order terms in favor of dominant trends.

1.2.11.1 The O-notation (Big-O)

The Big-O notation is a formalization of an upper bound on a function's growth rate.

Definition

Let $f(n)$ and $g(n)$ be real-valued functions defined for all sufficiently large n. We say:

$$f(n) = \mathcal{O}(g(n)) \quad \text{if and only if there exist constants } c > 0,\ n_0 \in \mathbb{N} \text{ such that } |f(n)| \leq c{\cdot}|g(n)| \text{ for all } n \geq n_0.$$

In other words, $f(n)$ does not grow faster than a constant multiple of $g(n)$, beyond some cutoff point n_0.

Interpretation

This notation suppresses:

- Constant factors: $\mathcal{O}(5n^2) = \mathcal{O}(n^2)$.
- Lower-order terms: $\mathcal{O}(n^2 + n + 7) = \mathcal{O}(n^2)$.

CR Prompt: Graph n, $n \log n$, n^2, and 2^n on the same plot. Try to estimate when one function overtakes another. Use this visual intuition to understand what $\mathcal{O}$ ignores.

Examples

- $7n^2 + 3n + 2 = \mathcal{O}(n^2)$
- $\log_2 n = \mathcal{O}(\log_{10} n) = \mathcal{O}(\ln n)$ (all logarithms differ only by constant factors)
- $n = \mathcal{O}(n^2)$ (but not vice versa!)

Important: $\mathcal{O}(g(n))$ denotes a *set* of functions. So we should write:

$$f(n) \in \mathcal{O}(g(n))$$

even though it is customary in CS literature to write $f(n) = \mathcal{O}(g(n))$.

Common Growth Classes (AS Summary Table)

Class	Name / Interpretation
$\mathcal{O}(1)$	Constant time
$\mathcal{O}(\log n)$	Logarithmic time (e.g., binary search)
$\mathcal{O}(n)$	Linear time (e.g., single pass through array)
$\mathcal{O}(n \log n)$	Linearithmic time (e.g., mergesort)
$\mathcal{O}(n^2)$	Quadratic time (e.g., naive matrix multiplication)
$\mathcal{O}(2^n)$	Exponential time
$\mathcal{O}(n!)$	Factorial time (e.g., brute-force TSP)

Mathematical Properties of Big-O (AS)

Let $f(n)$, $g(n)$, and $h(n)$ be asymptotically positive functions. Then:

- **Transitivity:** If $f(n) = \mathcal{O}(g(n))$ and $g(n) = \mathcal{O}(h(n))$, then $f(n) = \mathcal{O}(h(n))$.
- **Additivity:** $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max(f(n), g(n)))$.
- **Multiplicative Constants:** For any constant $c > 0$, if $f(n) = \mathcal{O}(g(n))$ then $c \cdot f(n) = \mathcal{O}(g(n))$.
- **Non-Negativity:** $f(n) = \mathcal{O}(g(n))$ does not imply $f(n) \geq 0$, but it is often assumed in complexity contexts.

AS Prompt: Prove that $f(n) = \mathcal{O}(n \log n)$ and $g(n) = \mathcal{O}(n)$ implies $f(n) + g(n) = \mathcal{O}(n \log n)$.
CR Prompt: Think about this: Is $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$ always true? Why or why not?

A Visual Metaphor for CR Thinkers

Imagine $g(n)$ is a fast-growing tide. The function $f(n)$ is a boat that floats somewhere on or below that tide after some point n_0.

The Big-O claim is:

> Beyond some n_0, $f(n)$ is always under control — the tide (i.e., $c \cdot g(n)$) never fails to carry the boat above it.

Formal Proof: $f(n) = 3n^2 + 2n + 7 \in \mathcal{O}(n^2)$

Goal: Find constants $c > 0$ and n_0 such that:

$$3n^2 + 2n + 7 \leq c \cdot n^2 \quad \text{for all } n \geq n_0$$

Proof:
We start by bounding the lower-order terms:

$$2n \leq 2n^2 \quad \text{for all } n \geq 1$$

$$7 \leq 7n^2 \quad \text{for all } n \geq 1$$

Adding:

$$3n^2 + 2n + 7 \leq 3n^2 + 2n^2 + 7n^2 = 12n^2$$

Hence:

$$3n^2 + 2n + 7 \leq 12n^2 \quad \text{for all } n \geq 1 \Rightarrow f(n) \in \mathcal{O}(n^2)$$

With $c = 12$, $n_0 = 1$, the condition is satisfied. ■

1.2.11.1 Concludes with a Subtle Warning

Knuth emphasizes: It is mathematically *incorrect* to write $f(n) = \mathcal{O}(g(n))$ as an equation. Big-O is a *set*, and $f(n)$ is a *member*.

Correct logic:

$$f(n) \in \mathcal{O}(g(n))$$

But common CS usage often simplifies this for notational convenience.

Tight Bounds: Θ Notation (AS Clarity)

Definition: $f(n) \in \Theta(g(n))$ if there exist constants $c_1, c_2 > 0$ and n_0 such that:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \text{for all } n \geq n_0$$

This defines an **asymptotic equivalence class** — both upper and lower bounded by $g(n)$.

Example:

$$f(n) = 4n^2 + 2n + 1 \in \Theta(n^2)$$

Because:

$$4n^2 + 2n + 1 \leq 7n^2 \quad (\text{for } n \geq 1)$$
$$4n^2 + 2n + 1 \geq 4n^2 \quad (\text{for } n \geq 1)$$

So $f(n)$ is squeezed between $4n^2$ and $7n^2$ for all $n \geq 1$.

Lower Bound: Ω Notation

Definition: $f(n) \in \Omega(g(n))$ if there exist $c > 0$ and n_0 such that:

$$f(n) \geq c \cdot g(n) \quad \text{for all } n \geq n_0$$

This is the dual of Big-O: it says $f(n)$ grows at least as fast as $g(n)$ asymptotically.

Little-o and Little-ω (Sharp Bounds)

Definition (Little-o):

$$f(n) \in o(g(n)) \iff \forall c > 0, \exists n_0 \text{ such that } |f(n)| \leq c \cdot |g(n)| \text{ for all } n \geq n_0$$

This is a *strict upper bound*: $f(n)$ grows strictly slower than $g(n)$.

Definition (Little-ω):

$$f(n) \in \omega(g(n)) \iff \forall c > 0, \exists n_0 \text{ such that } |f(n)| \geq c \cdot |g(n)| \text{ for all } n \geq n_0$$

This is a *strict lower bound*: $f(n)$ grows strictly faster than $g(n)$.

Examples:

$$\log n \in o(n^\varepsilon) \quad \text{for all } \varepsilon > 0$$
$$n^2 \in \omega(n \log n)$$

AS Proof: $\log n \in o(n^\varepsilon)$ for all $\varepsilon > 0$

Proof Sketch:

Use the fact that:

$$\lim_{n\to\infty} \frac{\log n}{n^\varepsilon} = 0 \quad \text{for any } \varepsilon > 0$$

This follows from L'Hôpital's Rule (or more elementary: exponential growth dominates logarithmic).

Therefore:

$$\forall c > 0,\ \exists n_0 \text{ such that } \frac{\log n}{n^\varepsilon} < c \Rightarrow \log n < cn^\varepsilon$$

Hence, $\log n \in o(n^\varepsilon)$. ■

Ordering Growth Rates (AS Table)

Let $f(n)$ and $g(n)$ be typical functions. Then:

$$\log\log n \ll \log n \ll n^{\varepsilon} \ll n \ll n\log n \ll n^2 \ll 2^n \ll n! \ll n^n$$

Where $\ll$ means: $f(n) \in o(g(n))$.

CR Prompt: Find a real-world algorithm or data structure corresponding to each growth rate. (E.g., hash table = $\mathcal{O}(1)$ average-case, binary search = $\mathcal{O}(\log n)$, etc.)

Next: Euler's Summation Formula

Having established asymptotic notation, Knuth proceeds to deeper methods for evaluating and approximating sums, especially when closed forms are unavailable.

Coming up next:

- Transition from discrete sums to integrals
- Use of Euler's summation formula for precise asymptotics
- Examples with harmonic numbers and power sums

This will deepen our capacity to derive bounds for algorithms involving loops, recursive calls, and nested summations.

1.2.11.2 Euler's Summation Formula

Motivation

Many algorithms contain loops or recursive calls that can be modeled as discrete sums:

$$S(n) = \sum_{k=1}^{n} f(k)$$

For instance, the total number of comparisons in insertion sort is $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$. But what if $f(k)$ has no closed form?

To analyze such sums rigorously, we need tools to:

- Approximate the sum using integrals.
- Bound the error of such approximation.
- Estimate growth using asymptotics.

One such tool is **Euler's Summation Formula**, a discrete-to-continuous bridge.

The Formula

Let $f(x)$ be a real-valued function defined on $[a, b]$ and possessing a continuous derivative. Then:

$$\sum_{k=a}^{b} f(k) = \int_a^b f(x)\,dx + \frac{f(a)+f(b)}{2} + \sum_{r=1}^{m} \frac{B_{2r}}{(2r)!}\left(f^{(2r-1)}(b) - f^{(2r-1)}(a)\right) + R_m$$

Where:

- B_{2r} are the even-indexed **Bernoulli numbers**
- $f^{(k)}(x)$ is the k-th derivative of f
- R_m is the remainder term (often negligible for large m)

Special case (first-order):

$$\sum_{k=a}^{b} f(k) = \int_a^b f(x)\,dx + \frac{f(a)+f(b)}{2} + \epsilon$$

for some small error ϵ when f is smooth and monotonic.

Derivation: Euler-Maclaurin Formula (AS)

This formula is a result of approximating a sum by the trapezoidal rule and correcting for curvature using higher derivatives:

$$\sum_{k=a}^{b} f(k) \approx \int_a^b f(x)\,dx + \frac{f(a)+f(b)}{2}$$

is just the trapezoidal rule. The Euler correction uses higher-order Taylor expansions and Bernoulli polynomials.

We now give a rigorous derivation of the first two terms:

Theorem (First-Order Euler Summation):

Let $f(x)$ be continuously differentiable on $[a, b]$. Then:

$$\sum_{k=a}^{b} f(k) = \int_a^b f(x)\,dx + \frac{f(a)+f(b)}{2} + \int_a^b \left(x - \lfloor x \rfloor - \frac{1}{2}\right) f'(x)\,dx$$

Proof Sketch:
Let $\phi(x) = x - \lfloor x \rfloor - \frac{1}{2}$. Then $\phi(x)$ is a periodic function with zero average on each unit interval.
We write:

$$f(k) = \int_{k-1}^{k} f'(x)\,dx + f(k-1)$$

and sum over k. Rewriting the result leads to a representation involving $\phi(x)$, which integrates against $f'(x)$ to yield the correction term.
■

AS Examples: Applying Euler's Formula

Example 1: Harmonic numbers
Define:

$$H_n = \sum_{k=1}^{n} \frac{1}{k}$$

Apply Euler's formula with $f(k) = 1/k$:

$$H_n = \int_1^n \frac{1}{x} dx + \frac{1}{2}\left(\frac{1}{1} + \frac{1}{n}\right) + R = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \cdots$$

Where γ is the Euler–Mascheroni constant (≈ 0.5772). The full expansion is:

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \cdots$$

CR Prompt: Plot $H_n - \ln n$ for large n. Can you visually see the convergence toward γ?
AS Prompt: Use Euler's formula to approximate $\sum_{k=1}^{n} \sqrt{k}$.

Example 2: Power sums

Let:

$$S_p(n) = \sum_{k=1}^{n} k^p$$

Euler's formula gives an asymptotic expansion:

$$S_p(n) = \frac{n^{p+1}}{p+1} + \frac{n^p}{2} + \frac{pn^{p-1}}{12} - \frac{p(p-1)(p-2)n^{p-3}}{720} + \cdots$$

Examples:

$$S_1(n) = \sum_{k=1}^{n} k = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$$
$$S_2(n) = \sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$$

These expansions allow precise asymptotics when exact formulas are messy or unavailable.

Bernoulli Numbers (Required for Higher-Order Terms)

Euler's formula involves Bernoulli numbers B_k, defined via the generating function:

$$\frac{t}{e^t - 1} = \sum_{k=0}^{\infty} \frac{B_k t^k}{k!}$$

First few:

$$B_0 = 1, \quad B_1 = -\frac{1}{2}, \quad B_2 = \frac{1}{6}, \quad B_4 = -\frac{1}{30}, \quad B_6 = \frac{1}{42}, \quad \text{(odd-indexed } B_{2k+1} = 0 \text{ for } k \geq 1\text{)}$$

AS Prompt: Derive the asymptotic formula for $\sum_{k=1}^{n} \log k$ using Euler's formula. (Hint: Stirling's approximation.)

CR Prompt: Visual and Conceptual Intuition

Imagine: A sum is a staircase. Euler's summation formula lays a smooth curve (integral) over that staircase and adjusts the curve with bumps (Bernoulli corrections) to match the steps precisely.

This intuition helps explain why the approximation is so accurate, even with just a few terms.

Next: Some Asymptotic Calculations

Now that we've developed a powerful approximation technique, we move to specific worked examples involving:

- Harmonic number asymptotics
- Logarithmic integrals
- Generalized p-series
- Rigorously bounding algorithm costs

This next section bridges Euler's tools and the actual task of analyzing algorithms' time complexity.

1.2.11.3 Some Asymptotic Calculations

Having developed a robust toolset (asymptotic notation and Euler's summation formula), we now turn to concrete examples of asymptotic calculations that appear frequently in algorithm analysis.

Knuth emphasizes how these results help us estimate the running time of loops and recursive calls by understanding the growth of common sums.

Example 1: Harmonic Numbers

Recall the definition of the n-th harmonic number:

$$H_n = \sum_{k=1}^{n} \frac{1}{k}$$

Asymptotic Expansion:

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \cdots$$

Proof Sketch (via Euler's Summation):
Apply the formula to $f(x) = 1/x$:

$$\sum_{k=1}^{n} \frac{1}{k} \approx \int_1^n \frac{1}{x}\,dx + \frac{1}{2}\left(\frac{1}{1} + \frac{1}{n}\right) = \ln n + \gamma + \cdots$$

The corrections beyond $\ln n$ come from higher derivatives of $1/x$ and Bernoulli numbers, giving the exact expansion.

Applications in Algorithm Analysis

The harmonic number arises in:

- **Average-case analysis of quicksort**
- **Expected height of binary search trees**
- **Loop analysis where iteration range halves each time**

Since $H_n = \ln n + \gamma + o(1)$, we often write:

$$H_n = \Theta(\log n)$$

in asymptotic terms.

Example 2: Logarithmic Integral Approximations

A common estimation in analysis:

$$\sum_{k=1}^{n} \log k$$

We know:

$$\sum_{k=1}^{n} \log k = \log(n!) \approx n \log n - n + \frac{1}{2}\log n + \frac{1}{2}\log(2\pi) + \frac{1}{12n} - \cdots$$

This is Stirling's approximation, derivable from Euler's formula applied to $f(x) = \log x$.
Applications:

- Space complexity of storing permutations.
- Counting comparisons in heap construction.
- Bounding entropy in data compression.

Example 3: Generalized Harmonic Sums

Let:

$$H_n^{(p)} = \sum_{k=1}^{n} \frac{1}{k^p}$$

Then:

- If $p = 1$: $H_n^{(1)} = H_n = \ln n + \gamma + \cdots$
- If $p > 1$: $H_n^{(p)} \to \zeta(p)$ as $n \to \infty$ (Riemann zeta function)

Example values:

$$\zeta(2) = \sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}, \quad \zeta(3) = \sum_{k=1}^{\infty} \frac{1}{k^3} \approx 1.202$$

CR Prompt: Try computing $\sum_{k=1}^{1000} \frac{1}{k^2}$ and compare to $\pi^2/6$. How fast does convergence occur?

Example 4: Growth of Factorials

Let $f(n) = \log(n!)$

$$\log(n!) = \sum_{k=1}^{n} \log k \approx n \log n - n + \frac{1}{2} \log n + \frac{1}{2} \log(2\pi) + \cdots$$

Exponentiating both sides gives:

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

This is Stirling's formula, vital in algorithm analysis when:

- Estimating the size of $n!$ (e.g., permutations)
- Bounding binomial coefficients
- Analyzing average-case costs (e.g., expected number of comparisons in random sampling)

Example 5: Sum of Floor Functions

Let $S(n) = \sum_{k=1}^{n} \left\lfloor \frac{n}{k} \right\rfloor$

Asymptotic Result:

$$S(n) = n \ln n + (2\gamma - 1)n + \mathcal{O}(\sqrt{n})$$

This arises in:

- Number-theoretic algorithms
- Sieve of Eratosthenes complexity
- Divisor functions and gcd-based operations

AS Prompt: Derive this approximation using integration and properties of $\lfloor n/k \rfloor$.

Example 6: Total Number of Bits to Represent Integers 1 to n

$$\sum_{k=1}^{n} \lfloor \log_2 k \rfloor \approx n \log_2 n - n + \Theta(\log n)$$

Application: Optimal coding lengths, binary encoding cost, radix sort behavior.

Final Synthesis

The examples above represent "typical" summations encountered in:

- Loop iteration cost analysis
- Divide-and-conquer recurrence solutions
- Tree traversal complexities
- Hashing, sorting, and probabilistic structures

Each asymptotic form — logarithmic, linear, polynomial, or exponential — becomes a building block in our analysis toolkit.

AS Final Prompt:

Let $T(n) = \sum_{k=1}^{n} \frac{\log k}{k}$

Estimate $T(n)$ asymptotically. Use substitution or Euler's formula.

Hint: This converges slowly to $\frac{1}{2}(\log n)^2 + \gamma_1 + o(1)$, where γ_1 is the first Stieltjes constant.

CR Final Prompt:

Invent your own algorithm and count something: comparisons, swaps, divisions, etc. Try to model the count as a sum. Then attempt to:

1. Estimate it with an integral.
2. Check if Euler's formula helps.
3. Classify it with Big-O notation.

Conclusion of Sections 1.2.10 – 1.2.11.3

We have now:

- Defined rigorous asymptotic notation.
- Derived and proved bounds for common algorithmic quantities.
- Applied Euler's summation formula to extract precise approximations.
- Classified major classes of growth and their algorithmic implications.

This forms the mathematical backbone of algorithm analysis — a fusion of:

- Summations and integrals,
- Limit behavior,
- Recurrence solving,
- Bounding via asymptotic envelopes.

These tools will be used repeatedly across TAOCP in analyzing sorts, searches, tree structures, and number-theoretic computations.