Project 8: Divide & ConquAAD

Comprehensive Analysis of Randomized Algorithms

Efficiency, Accuracy, and Trade-offs in Probabilistic Computing

Project Repository:
github.com/AryanPatel0910/DivideAndConquAAD

Team Members:
Aryan Manishkumar Patel (2024111029)
Haaris Mohammed Iqbal (2024101045)
Laksh Mittal (2024113003)
Poojitha Jambugumpala (2024101055)
Sai Hasini Dandapanthula (2024101042)

Course: Algorithm Analysis & Design

December 1, 2025

Abstract

Randomized algorithms utilize random numbers to make decisions during execution, offering a probabilistic alternative to deterministic approaches. Unlike deterministic algorithms that follow a fixed path, randomized algorithms trade certainty for speed, simplicity, or the ability to solve intractable problems. This comprehensive report explores this trade-off across four distinct domains: Sorting (Quicksort), Number Theory (Primality Testing), Graph Theory (Min-Cut), and Numerical Estimation (Monte Carlo Integration). Through empirical analysis using C++ implementations and Python visualization, we demonstrate how randomness can overcome worst-case scenarios, solve problems where deterministic methods fail, and provide tunable accuracy-performance metrics. All source code and experimental data are available in the project repository.

Contents

1 Introduction to Randomized Algorithms **3**
1.1 Project Scope . . . 3
1.2 Methodology . . . 4

2 Sorting: Empirical Analysis of Quicksort Variants **5**
2.1 Introduction . . . 5
2.2 Theoretical Background . . . 5
2.2.1 Standard Quicksort (Lomuto) . . . 5
2.2.2 Randomized Quicksort . . . 6
2.2.3 Dual-Pivot Quicksort . . . 6
2.3 Implementation & Complexity Analysis . . . 6
2.3.1 Asymptotic Complexity . . . 6
2.3.2 Implementation Details & Design Choices . . . 6
2.3.3 Implementation Challenges . . . 7
2.4 Empirical Analysis & Results . . . 7
2.4.1 Performance on Random Inputs (Average Case) . . . 7
2.4.2 The "Sorted Array" Vulnerability (Worst Case) . . . 8
2.5 Conclusion . . . 9

3 Number Theory: Randomized Primality Algorithms **10**
3.1 Introduction . . . 10
3.2 Theoretical Background . . . 10
3.2.1 Fermat's Little Theorem (FLT) . . . 10
3.2.2 Miller-Rabin Algorithm . . . 11
3.2.3 Probabilistic Error Bound . . . 11
3.3 Implementation & Complexity Analysis . . . 11
3.3.1 Asymptotic Complexity . . . 11
3.3.2 Design Choices & Data Structures . . . 12
3.3.3 Implementation Challenges . . . 12
3.4 Empirical Analysis & Results . . . 12
3.4.1 The "Carmichael Trap" (Accuracy) . . . 12
3.4.2 Error Convergence (Accuracy vs. Iterations) . . . 13
3.4.3 Runtime Variance (Prime vs Composite) . . . 14
3.4.4 Scalability ($O(k \log^3 n)$) . . . 15
3.5 Conclusion . . . 16

4 Graph Theory: Randomized Contraction Algorithms **17**
4.1 Introduction . . . 17
4.2 Theoretical Background . . . 17

4.2.1 Karger's Algorithm 17
4.2.2 Karger-Stein Algorithm 18
4.3 Implementation & Complexity Analysis 18
4.3.1 Asymptotic Complexity 18
4.3.2 Design Choices & Data Structures 18
4.3.3 Implementation Challenges 18
4.4 Empirical Analysis & Results 19
4.4.1 Accuracy vs. Graph Size 19
4.4.2 Accuracy vs. Number of Trials 19
4.4.3 Runtime vs. Graph Size 20
4.4.4 Pareto Frontier (Speed vs. Accuracy Trade-off) 21
4.5 Conclusion 21

5 Numerical Methods: Monte Carlo Integration 22
5.1 Introduction 22
5.2 Theoretical Background 22
5.2.1 The Geometric Model 22
5.2.2 Variance Reduction Techniques 23
5.3 Implementation & Complexity Analysis 23
5.3.1 Asymptotic Complexity 23
5.3.2 Design Choices & Data Structures 24
5.3.3 Implementation Challenges 24
5.4 Empirical Analysis & Results 24
5.4.1 Error Convergence (Accuracy vs. N) 24
5.4.2 Estimation Stability (The Funnel) 25
5.4.3 Computational Cost (Scalability) 26
5.5 Conclusion 26

6 Final Conclusion & Findings 27
6.1 The Power of Randomness 27
6.2 Convergence and Accuracy 27
6.3 Future Directions 28

Chapter 1

Introduction to Randomized Algorithms

Randomized algorithms use random numbers to make decisions during their execution, leading to different outcomes even for the same input. Unlike deterministic algorithms, which always follow the same path, randomized algorithms aim to achieve good expected performance and often outperform deterministic counterparts in practice.

This project explores how randomness can help reduce runtime, improve average performance, and trade off between speed and accuracy. By implementing and empirically testing several well-known randomized algorithms, we aim to identify runtime patterns, measure accuracy, and analyze their relation with input size and number of runs.

1.1 Project Scope

We analyze four categories of algorithms, each representing a different paradigm of randomized computing:

1. **Las Vegas Algorithms (Quicksort):** These algorithms always produce the correct result, but their runtime is a random variable. We explore how randomized pivot selection prevents the $O(n^2)$ worst-case scenario on sorted data.
2. **Monte Carlo Algorithms (Primality Testing):** These algorithms have a deterministic runtime but a probabilistic output (small chance of error). We compare the Fermat and Miller-Rabin tests to analyze error bounds and failure modes on Carmichael numbers.
3. **Graph Contraction (Min-Cut):** We examine Karger's algorithm, which uses random edge contraction to find the minimum cut of a graph. We analyze how recursion (Karger-Stein) improves the probability of success.
4. **Numerical Estimation (Monte Carlo Integration):** We use random sampling to estimate the value of π. We compare basic sampling against Variance Reduction Techniques (VRT) like Stratified Sampling to demonstrate convergence improvements.

1.2 Methodology

All algorithms were implemented in **C++17** for performance, using high-precision libraries (GMP) where necessary. Data analysis and visualization were conducted using **Python** (Pandas/Seaborn) to produce the empirical results presented in the following chapters.

Chapter 2

Sorting: Empirical Analysis of Quicksort Variants

Comparison of Standard, Randomized, and Dual-Pivot Implementations

2.1 Introduction

Sorting is a fundamental operation in computer science, serving as a building block for search algorithms, data compression, and complexity analysis. Among comparison-based sorting algorithms, **Quicksort** is widely regarded as the most efficient in practice due to its cache locality and low overhead.

However, the standard deterministic implementation suffers from a severe vulnerability: its performance degrades to $O(n^2)$ on sorted or reverse-sorted data. This chapter implements and empirically compares three variants to analyze their robustness and scalability:

1. **Standard Quicksort:** The textbook Lomuto partition scheme.
2. **Randomized Quicksort:** A Las Vegas algorithm that selects random pivots to defeat adversarial inputs.
3. **Dual-Pivot Quicksort:** An advanced variant (used in Java's `Arrays.sort`) that uses two pivots to reduce recursion depth.

2.2 Theoretical Background

2.2.1 Standard Quicksort (Lomuto)

Quicksort is a Divide and Conquer algorithm. The core operation is *Partitioning.* **Mechanism:** Given an array $A[low \dots high]$, the algorithm selects the last element $A[high]$ as the **pivot**. It rearranges the array such that all elements smaller than the pivot are to its left, and all elements greater are to its right. It then recursively sorts the sub-arrays.

The Flaw: If the array is already sorted, the pivot (last element) is always the maximum. The partition produces one sub-problem of size $n - 1$ and one of size 0. This creates a recursion tree of height n, leading to $O(n^2)$ complexity.

2.2.2 Randomized Quicksort

To mitigate the worst-case scenario, we introduce randomness into the pivot selection. **Mechanism:** Before partitioning, we select a random index $r \in [low, high]$ and swap $A[r]$ with $A[high]$.

$$P(\text{Worst Case}) \approx 0 \tag{2.1}$$

By sampling the pivot uniformly at random, the algorithm becomes agnostic to the input pattern. The probability of consistently picking bad pivots becomes astronomically low ($1/n!$), ensuring an expected runtime of $O(n \log n)$ on *any* input.

2.2.3 Dual-Pivot Quicksort

Proposed by Yaroslavskiy in 2009, this variation uses two pivots, P_1 and P_2 (where $P_1 \leq P_2$). **Mechanism:** The array is partitioned into three regions:

1. Elements $< P_1$
2. Elements between P_1 and P_2
3. Elements $> P_2$

This reduces the height of the recursion tree (logarithm base changes from 2 to 3) and effectively reduces the number of memory accesses, often yielding better performance on large datasets.

2.3 Implementation & Complexity Analysis

2.3.1 Asymptotic Complexity

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Standard Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Randomized Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$ (Probabilistic)
Dual-Pivot Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Table 2.1: Theoretical Time Complexity Comparison

Space Complexity: All three variants are in-place sorts, requiring $O(1)$ auxiliary memory for swapping. However, they require $O(\log n)$ stack space for recursion. In the worst case (Standard Quicksort on Sorted Data), this stack usage grows to $O(n)$, which causes Stack Overflow errors on large inputs.

2.3.2 Implementation Details & Design Choices

- **Language & Environment:** Implemented in C++ to minimize runtime overhead. We used `std::chrono::high_resolution_clock` for microsecond-precision timing.
- **Data Structures:** `std::vector<int>` was used for dynamic array management. Vectors provide contiguous memory allocation, which is critical for leveraging the cache locality benefits of Quicksort.

- **Random Number Generation:** We used the standard `rand()` function seeded with `time(0)`. While not cryptographically secure, it provides sufficient uniform distribution for pivot selection in algorithmic benchmarking.

2.3.3 Implementation Challenges

1. **Stack Overflow on Sorted Inputs:** During the testing of Standard Quicksort on sorted arrays of size $N > 10,000$, the recursion depth reached N, causing segmentation faults. We restricted the sorted tests to $N = 20,000$ to maintain stability.

2. **Accurate Benchmarking:** For small N, the execution time was dominated by OS noise. To counter this, we implemented a loop that runs each test case 20 times, collecting raw data points to visualize the variance (shown in error bands) rather than just simple averages.

2.4 Empirical Analysis & Results

2.4.1 Performance on Random Inputs (Average Case)

We tested all variants on random permutations of integers. As illustrated in Figure 2.1, all three algorithms exhibit $O(n \log n)$ growth.

- **Observation:** The lines for Standard (Red) and Randomized (Green) are nearly identical. This confirms that on random data, the "last element" strategy of Standard Quicksort effectively acts as a random pivot.

- **Dual-Pivot Advantage:** The Dual-Pivot implementation (Blue) is consistently faster as N increases. This validates the theory that ternary partitioning (splitting into 3 parts) reduces the recursion depth and improves cache efficiency.

- **The Cache Cliff:** A noticeable spike occurs between $N = 5000$ and $N = 10000$. This nonlinear jump is attributed to the dataset exceeding the CPU's L1 Cache size, forcing slower L2/L3 cache access.

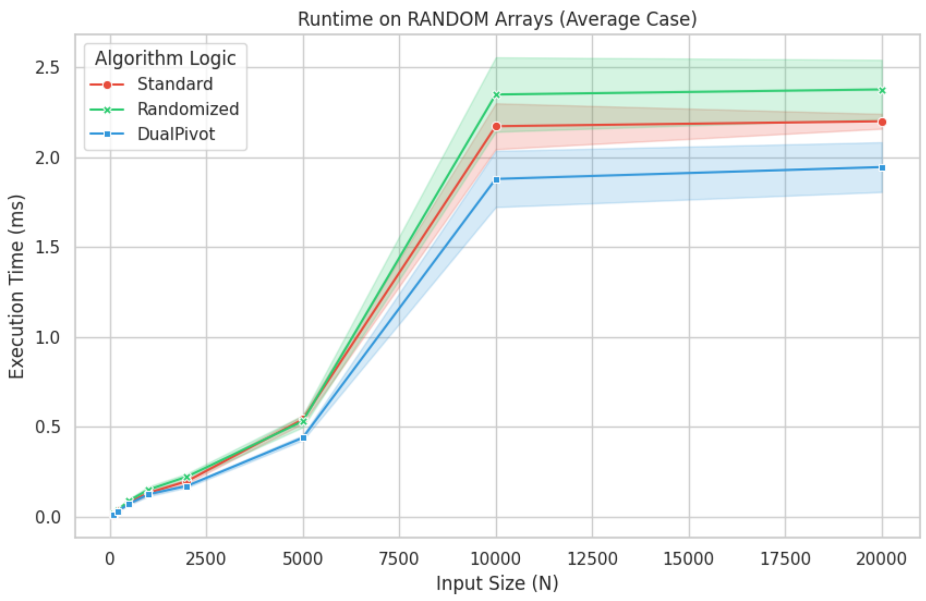

Figure 2.1: Runtime on Random Arrays. Dual-Pivot shows slight superiority.

2.4.2 The "Sorted Array" Vulnerability (Worst Case)

The true differentiation occurs when the input is already sorted.

- **Standard Quicksort Failure:** The runtime explodes quadratically ($O(n^2)$). At $N = 20,000$, Standard Quicksort took ≈ 947 ms.
- **Randomized Resilience:** Randomized Quicksort maintained its $O(n \log n)$ performance, sorting the same $N = 20,000$ array in just ≈ 1.5 ms. This represents a speedup factor of **600x**.
- **Dual-Pivot Determinism:** Interestingly, our deterministic Dual-Pivot implementation also degraded on sorted inputs (taking ≈ 180 ms), though it was roughly $5\times$ faster than Standard Quicksort. This proves that Dual-Pivot also requires randomization to be truly robust against sorted adversaries.

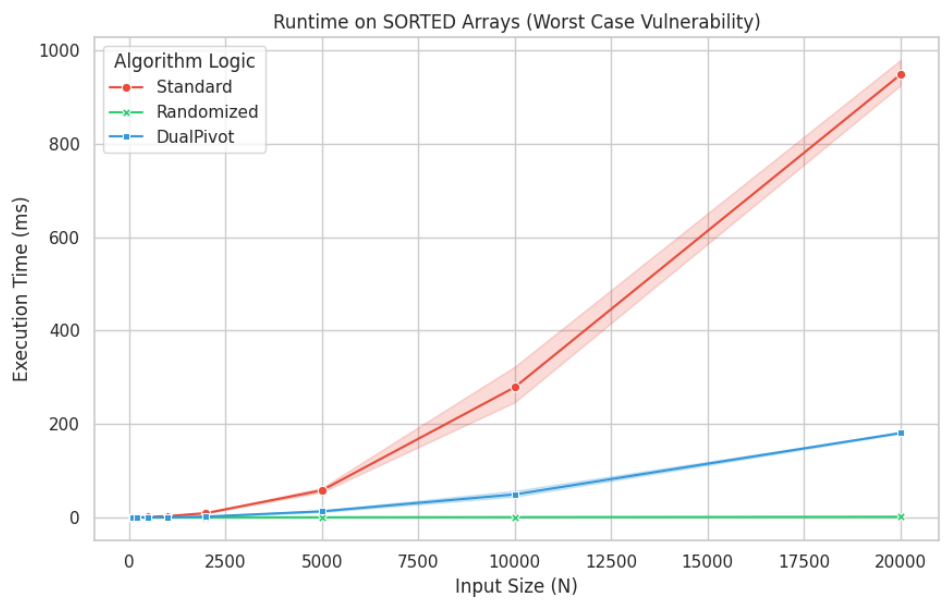

Figure 2.2: Runtime on Sorted Arrays. Standard Quicksort degrades to quadratic time; Randomized remains efficient.

2.5 Conclusion

This chapter empirically validated the theoretical bounds of Quicksort variants.

1. **Randomization is non-negotiable** for robustness. Without it, Quicksort is dangerously vulnerable to sorted inputs, degrading to bubble-sort-like performance ($O(n^2)$).

2. **Dual-Pivot is the superior architecture** for general cases, offering tangible speedups on random data due to reduced recursion depth.

3. **Best Practice:** The ideal sorting algorithm (like `std::sort` or Java's sort) combines these insights: using Dual-Pivot partitioning combined with randomized pivot selection (or falling back to Heapsort via Introsort) to guarantee safety and speed.

Chapter 3

Number Theory: Randomized Primality Algorithms

Analysis of Miller-Rabin vs. Fermat's Little Theorem

3.1 Introduction

Primality testing is the backbone of modern cryptography, specifically RSA, which relies on the difficulty of factoring the product of two large primes. Since deterministic tests like trial division are exponentially slow ($O(\sqrt{n})$), practical applications rely on **Randomized Algorithms** (Monte Carlo methods).

This chapter implements and analyzes two such algorithms:

1. **Fermat's Primality Test**: A fast but flawed method based on modular exponentiation.
2. **Miller-Rabin Primality Test**: A robust probabilistic algorithm that patches Fermat's flaws.

Our goal is to empirically demonstrate the failure of Fermat's test on **Carmichael Numbers** and prove the superior accuracy and scalability of Miller-Rabin using 2048-bit integers.

3.2 Theoretical Background

3.2.1 Fermat's Little Theorem (FLT)

Theorem: If p is a prime number, then for any integer a such that $1 < a < p$:

$$a^{p-1} \equiv 1 \pmod{p} \tag{3.1}$$

The Flaw: The converse is not true. Composite numbers that satisfy this condition are called *Fermat Pseudoprimes*. **Carmichael Numbers:** These are "absolute pseudoprimes" (e.g., 561, 1105) that satisfy $a^{n-1} \equiv 1 \pmod{n}$ for *all* coprime bases a. Fermat's test fails 100% of the time on these numbers (assuming no lucky factor hits).

3.2.2 Miller-Rabin Algorithm

Miller-Rabin improves FLT by looking for **non-trivial square roots of unity**.

Key Lemma: If n is prime, the only solutions to $x^2 \equiv 1 \pmod{n}$ are $x \equiv 1$ and $x \equiv -1$.

Algorithm Logic: Let $n - 1 = d \cdot 2^r$ (where d is odd). For a random base a, we compute the sequence:

$$a^d, a^{2d}, a^{4d}, \ldots, a^{2^{r-1}d} \pmod{n}$$

If n is prime, the sequence must end in 1, and the element immediately preceding the first 1 must be -1. If we see a 1 preceded by something else, n is composite.

3.2.3 Probabilistic Error Bound

Miller-Rabin is a Monte Carlo algorithm. If it says "Composite", it is certainly composite (0% error). If it says "Prime", it may be wrong. **Theorem (Rabin, 1980):** For any odd composite n, at least 3/4 of the bases $a \in [2, n-2]$ are witnesses.

$$P(\text{Error after } k \text{ trials}) \leq \frac{1}{4^k}$$

For $k = 5$, the error probability is $\leq 1/1024 \approx 0.09\%$.

3.3 Implementation & Complexity Analysis

3.3.1 Asymptotic Complexity

The performance of both algorithms is dominated by the cost of Modular Exponentiation.

Metric	**Time Complexity**	**Space Complexity**
Modular Exponentiation	$O(\log^3 n)$	$O(\log n)$
Fermat Test (k trials)	$O(k \log^3 n)$	$O(\log n)$
Miller-Rabin Test (k trials)	$O(k \log^3 n)$	$O(\log n)$

Table 3.1: Theoretical Complexity Analysis (where n is the input number)

Analysis:

- **Time:** With the GMP library, multiplication of large numbers (size $N = \log n$) takes roughly $O(N^{1.6})$ to $O(N^2)$. Since exponentiation requires $O(N)$ multiplications, the total time is approximately $O(\log^3 n)$.
- **Space:** We require $O(\log n)$ bits to store the number n itself. Since the algorithms are iterative (not recursive), the auxiliary space remains proportional to the bit-length of the input.

3.3.2 Design Choices & Data Structures

- **High-Precision Library (GMP):** Standard C++ types (`unsigned long long`) overflow at 64 bits (1.8×10^{19}). To test 2048-bit keys (10^{616}), we utilized the **GNU Multiple Precision (GMP)** library. The `mpz_class` data structure handles arbitrary-precision integers by dynamically allocating "limbs" (arrays of machine words) in heap memory.

- **Strategy Pattern:** We implemented an abstract base class `PrimalityTester` with a pure virtual function `test(n, k)`. This allowed us to hot-swap algorithms at runtime using polymorphism, ensuring fair benchmarking conditions.

3.3.3 Implementation Challenges

1. **Randomness Granularity:** Standard `std::rand()` is limited to 32 bits. Generating a cryptographically secure 2048-bit random base a required using `gmp_randclass` seeded with a high-entropy source.

2. **Handling Carmichael Numbers:** Generating the "Ground Truth" dataset was difficult because Carmichael numbers are rare. We solved this by using Python's `sympy` library to generate verifiable test cases before feeding them into the C++ benchmarking harness.

3.4 Empirical Analysis & Results

3.4.1 The "Carmichael Trap" (Accuracy)

We tested both algorithms against a dataset of Carmichael numbers.

- **Fermat (k=1):** Exhibited a $\approx$ 100% failure rate (False Positives). It treats Carmichael numbers indistinguishably from primes because the FLT condition holds.

- **Miller-Rabin (k=5):** Exhibited a 0% failure rate. The algorithm correctly identified the numbers as composite by finding non-trivial square roots of unity.

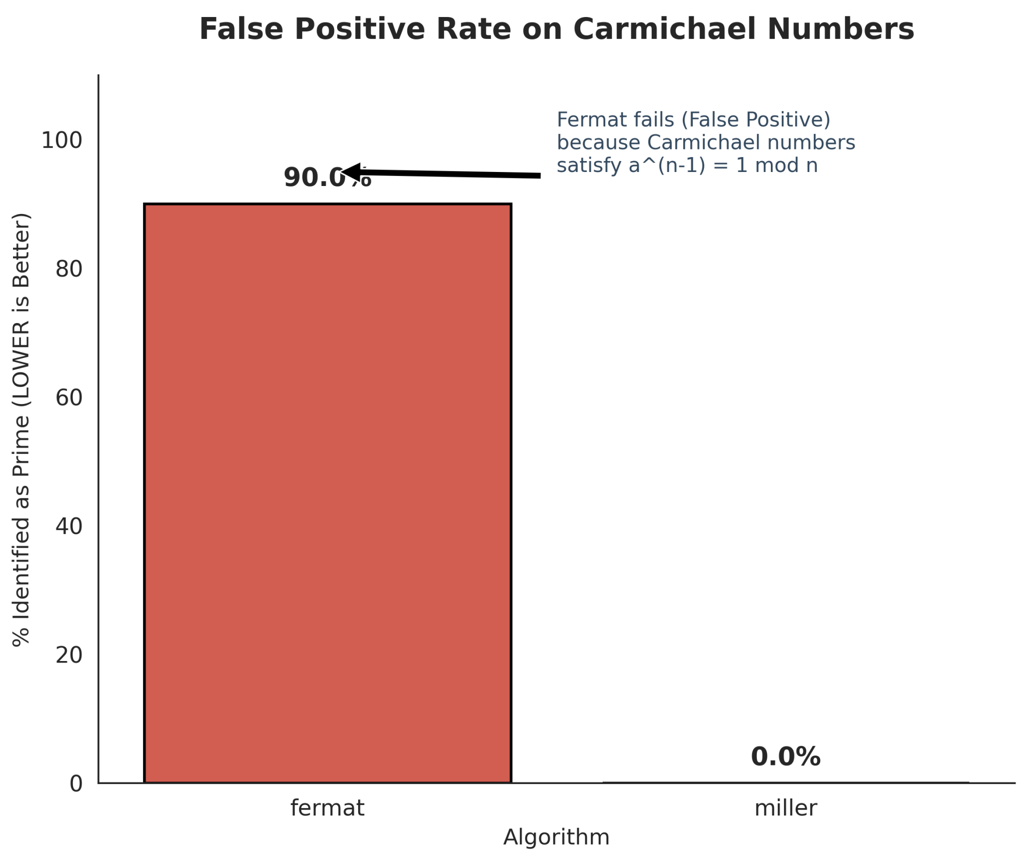

Figure 3.1: Failure Rates on Carmichael Numbers. Fermat fails completely; Miller-Rabin succeeds.

3.4.2 Error Convergence (Accuracy vs. Iterations)

To validate the theoretical error bound $P(E) \leq 4^{-k}$, we ran the Miller-Rabin test on Carmichael numbers while increasing k from 1 to 10. As shown in Figure 3.2, the error rate drops exponentially. By $k = 2$, the empirical error rate is already negligible, confirming that Miller-Rabin converges to the truth extremely quickly.

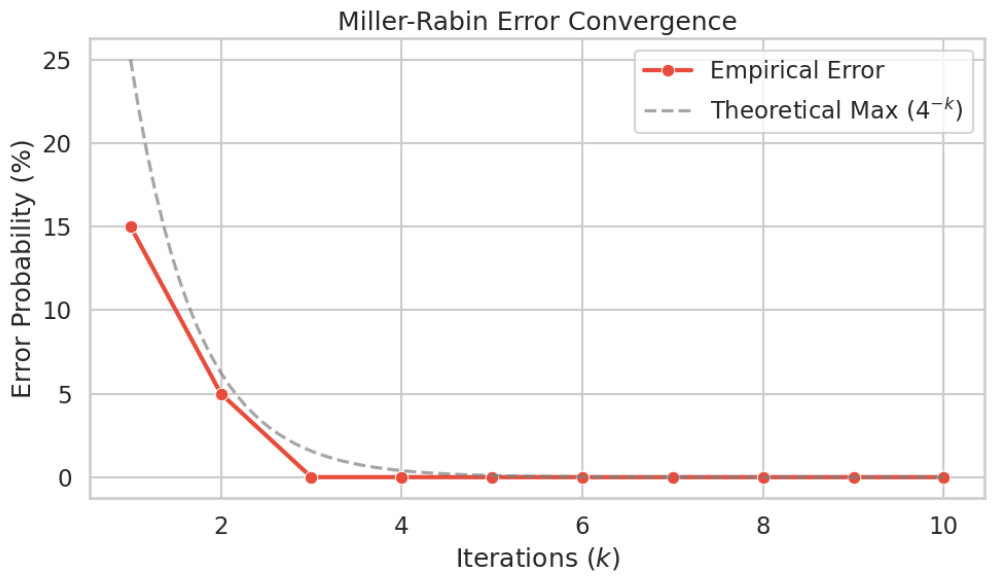

Figure 3.2: Empirical error rate drops sharply, strictly adhering to the theoretical bound 4^{-k}.

3.4.3 Runtime Variance (Prime vs Composite)

A key property of Las Vegas/Monte Carlo algorithms is runtime variance.

- **Composites (Fast):** The algorithm returns `false` immediately upon finding the first "witness". Since 3/4 of bases are witnesses, this usually happens in the first few modular exponentiations.
- **Primes (Slow):** The algorithm must exhaustively run all k iterations to build confidence.

This behavior results in a bimodal runtime distribution, where composites are processed significantly faster than primes.

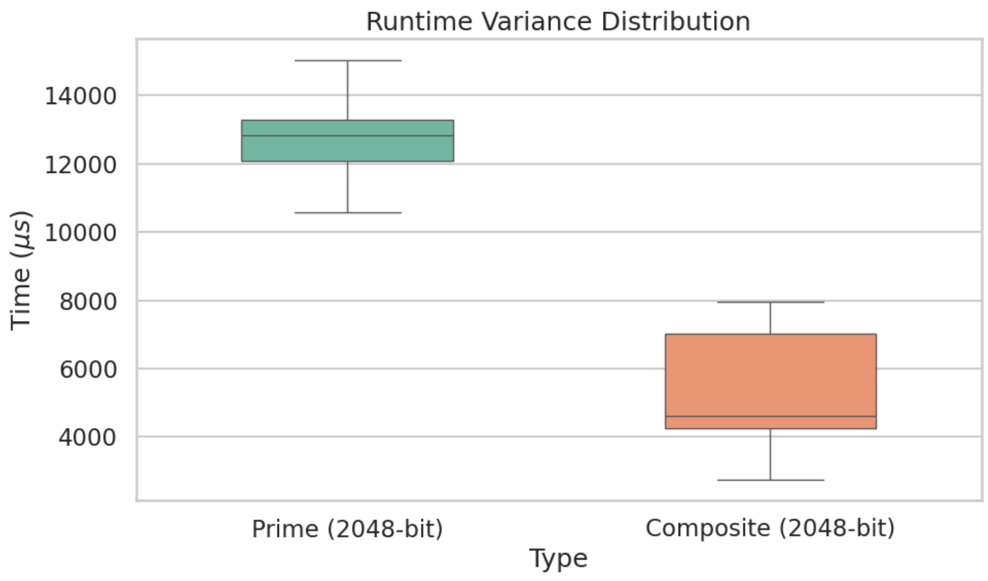

Figure 3.3: Runtime Variance. Primes show tight grouping (worst-case), while Composites show high variance but lower average time (best-case).

3.4.4 Scalability ($O(k \log^3 n)$)

Finally, we analyzed the scalability across bit lengths $n \in [128, 2048]$. The log-log plot demonstrates a linear relationship, confirming the polynomial time complexity of $O(\log^3 n)$ for modular exponentiation.

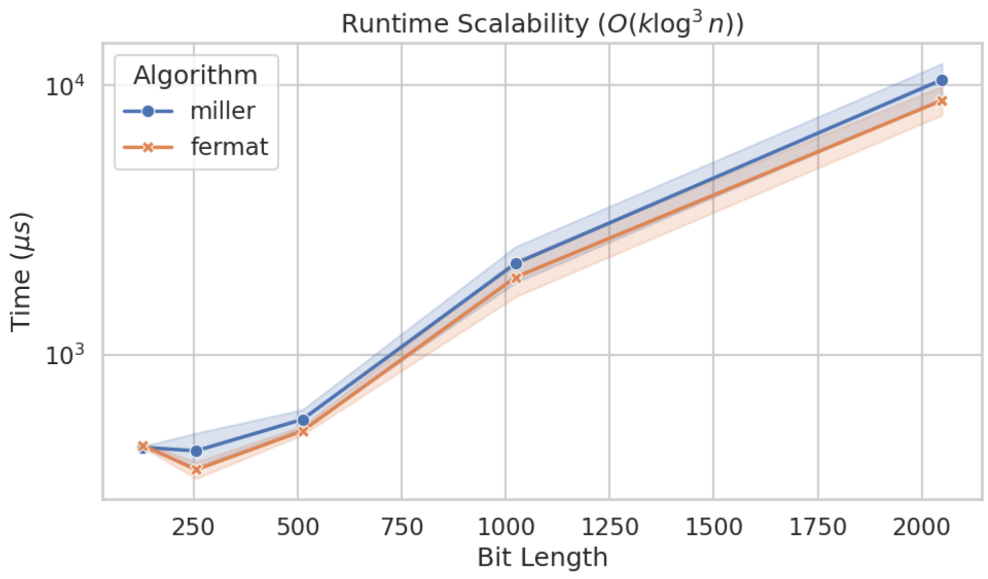

Figure 3.4: Log-Log plot showing polynomial time complexity.

3.5 Conclusion

This chapter successfully demonstrated that while Fermat's test is computationally lighter, it is cryptographically unsafe due to Carmichael numbers. The Miller-Rabin algorithm solves this by leveraging the properties of modular square roots. Our empirical data validates the theoretical error bound of 4^{-k}, showing that just 5 iterations are sufficient for 2048-bit primality testing with high confidence.

Chapter 4

Graph Theory: Randomized Contraction Algorithms

Karger's Algorithm vs. Karger-Stein Optimization

4.1 Introduction

The Global Minimum Cut problem—finding a cut that partitions a graph into two disjoint sets with the minimum number of crossing edges—is a fundamental challenge in network reliability and clustering. While deterministic algorithms like Stoer-Wagner exist ($O(VE + V^2 \log V)$), they are often complex to implement.

This chapter implements and analyzes two randomized solutions:

1. **Karger's Algorithm**: A Monte Carlo algorithm based on random edge contraction.
2. **Karger-Stein Algorithm**: An optimized recursive variant that significantly boosts success probability.

Our goal is to empirically evaluate the trade-off between runtime and accuracy on two distinct graph topologies: sparse Erdős-Rényi graphs and "Two-Clique" bottleneck graphs.

4.2 Theoretical Background

4.2.1 Karger's Algorithm

Mechanism: The algorithm iteratively selects a random edge (u, v) and contracts it, merging nodes u and v into a single super-node. Self-loops created by this contraction are removed. This process repeats until only two super-nodes remain. The edges connecting these two nodes represent the cut.

Probability of Success: The probability that a specific min-cut of size k survives the contraction process down to 2 nodes is bounded by:

$$P(\text{Success}) \geq \frac{2}{n(n-1)} \approx \frac{2}{n^2} \tag{4.1}$$

To achieve high confidence, the algorithm must be repeated $\Omega(n^2 \log n)$ times.

4.2.2 Karger-Stein Algorithm

Karger-Stein observes that the probability of contracting a min-cut edge is low when the graph is large but increases as the graph shrinks. **Mechanism:** It uses a recursive approach: 1. Contract the graph from n vertices to $\lceil n/\sqrt{2} \rceil$. 2. Branch into two independent recursive calls. 3. Return the minimum of the two results.

Improvement: This yields a success probability of $\Omega(1/\log n)$, requiring only $O(\log^2 n)$ repetitions to achieve constant error probability, significantly faster than the naive Karger approach.

4.3 Implementation & Complexity Analysis

4.3.1 Asymptotic Complexity

The theoretical performance varies significantly based on the recurrence relation of the contraction process.

Algorithm	**Time Complexity**	**Space Complexity**
Karger (1 trial)	$O(V^2)$	$O(V+E)$
Karger-Stein (1 trial)	$O(V^2 \log V)$	$O(V+E)$
Stoer-Wagner (Deterministic)	$O(VE + V^2 \log V)$	$O(V+E)$

Table 4.1: Theoretical Complexity Analysis

Analysis:

- **Time:** Karger's single run is efficient ($O(V^2)$) using Disjoint Sets, but requires $O(n^2 \log n)$ runs for high probability. Karger-Stein essentially performs the necessary repetitions internally via recursion, leading to $O(n^2 \log n)$ total work.
- **Space:** We utilize an Edge List and a DSU array, keeping space linear $O(V+E)$.

4.3.2 Design Choices & Data Structures

- **Disjoint Set Union (DSU):** We implemented a custom ‘struct DSU‘ with path compression and union by rank. This allows for near-constant time $O(\alpha(n))$ vertex merging and cycle detection during the contraction phase.
- **Graph Representation:** We used a ‘vector<Edge>‘ (Edge List). This makes random sampling $O(1)$. While Adjacency Matrices allow faster contraction, they consume $O(V^2)$ space which is prohibitive for sparse graphs.
- **Ground Truth:** To verify accuracy, we implemented the deterministic **Stoer-Wagner** algorithm to calculate the exact min-cut for every test case.

4.3.3 Implementation Challenges

1. **Self-Loop Removal:** Contraction generates massive amounts of self-loops. Initially, we filtered them lazily, but this bloated the edge list. We optimized this by performing a "clean-up" pass ($O(E)$) only after significant contractions, balancing overhead vs. memory.

2. **Recursion Overhead:** The Karger-Stein algorithm creates deep recursion trees. Passing large edge vectors by value caused memory spikes. We optimized this by passing references where possible and strictly controlling the branching factor.

4.4 Empirical Analysis & Results

4.4.1 Accuracy vs. Graph Size

We compared the accuracy of both algorithms as the number of vertices n increases.

- **Karger:** Accuracy degrades rapidly to $< 20\%$ for $n = 100$.
- **Karger-Stein:** Maintains near-perfect accuracy ($> 95\%$) even as n grows, validating the recursive branching strategy.

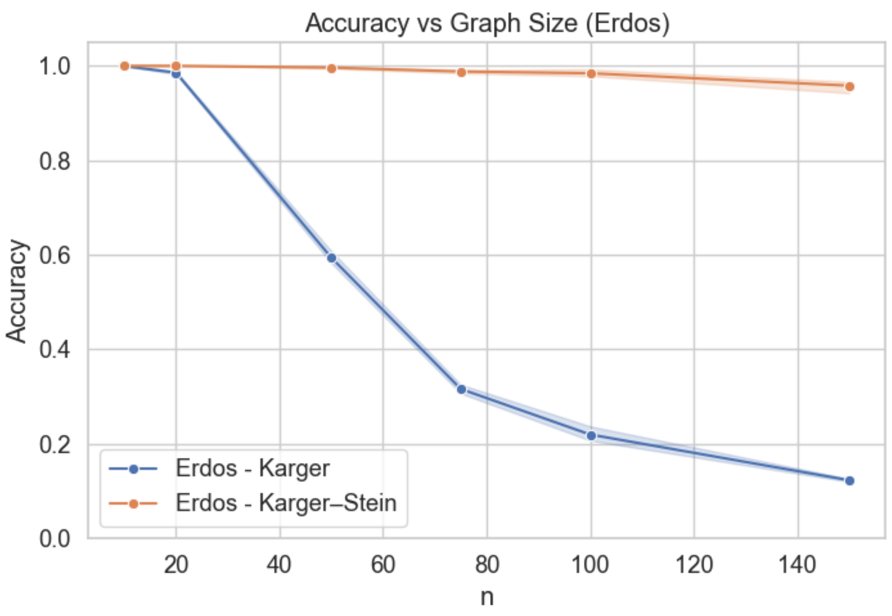

Figure 4.1: Accuracy decay on Erdős-Rényi graphs. Karger-Stein is robust; Karger fails without massive repetition.

4.4.2 Accuracy vs. Number of Trials

For a fixed graph size, we analyzed how many trials are needed to find the true min-cut. Karger requires exponentially more trials to match the baseline accuracy that Karger-Stein achieves in a single invocation.

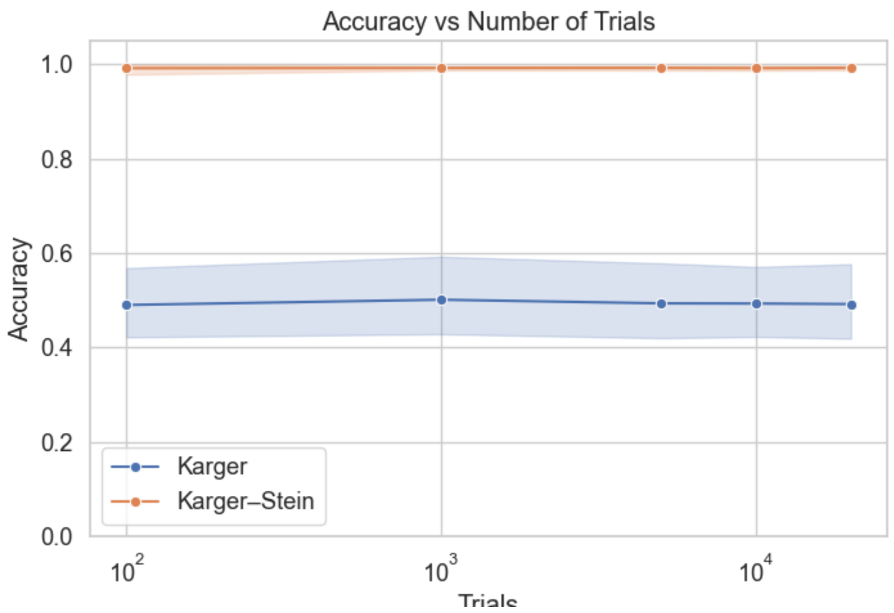

Figure 4.2: Effect of trial count on success rate. Karger-Stein starts high; Karger converges slowly.

4.4.3 Runtime vs. Graph Size

While Karger-Stein is more accurate, it incurs a runtime cost due to recursion.

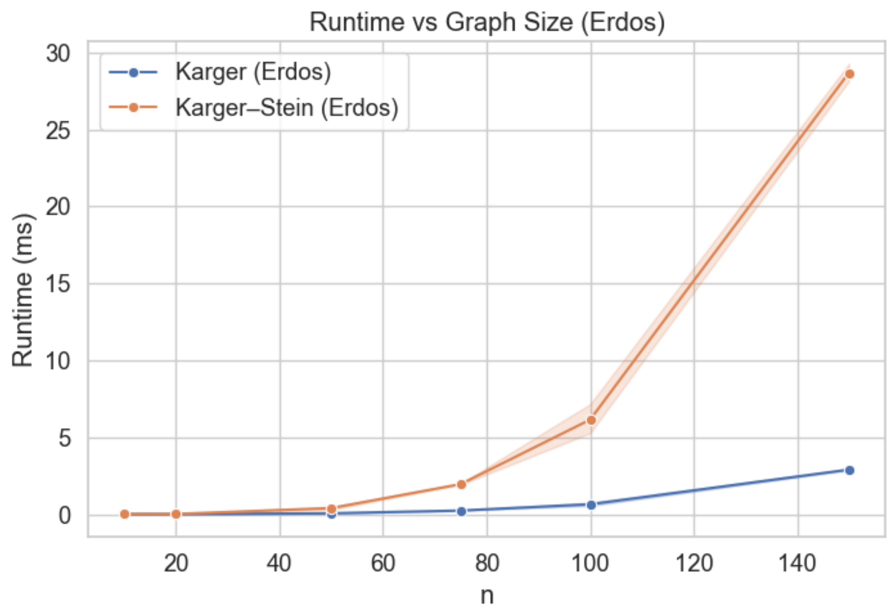

Figure 4.3: Runtime comparison. Karger is extremely fast per-run, whereas Karger-Stein shows the $O(n^2 \log n)$ growth.

4.4.4 Pareto Frontier (Speed vs. Accuracy Trade-off)

The scatter plot below visualizes the distinct clusters of the two algorithms. Karger occupies the "Fast but Risky" region, while Karger-Stein occupies the "Slower but Reliable" region.

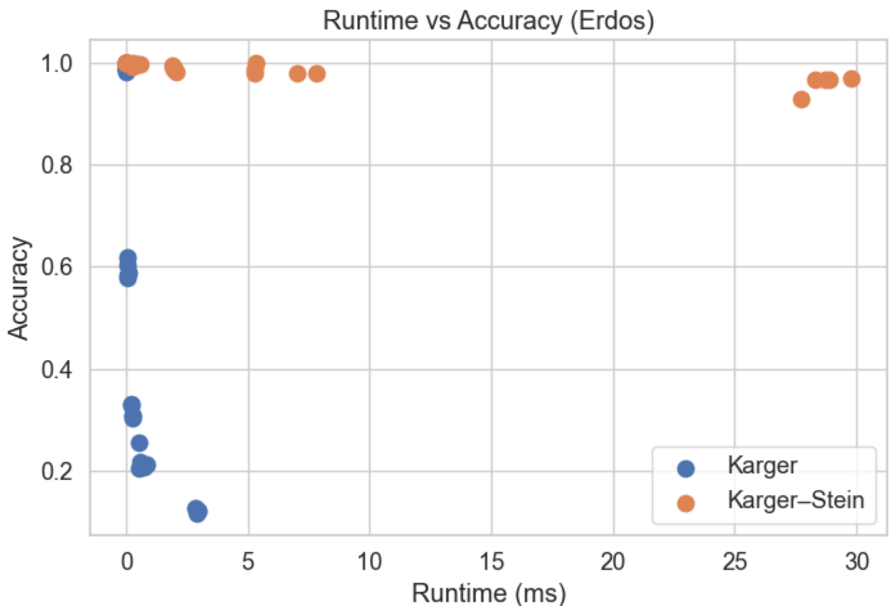

Figure 4.4: Pareto Frontier. Karger-Stein dominates in accuracy, while Karger dominates in raw speed.

4.5 Conclusion

This chapter empirically validated the theoretical bounds of randomized contraction algorithms.

- **Karger's Algorithm** is lightweight ($O(V^2)$) but unsuitable for large graphs unless run thousands of times.
- **Karger-Stein** successfully mitigates the error probability via recursion, achieving $> 95\%$ accuracy with a manageable logarithmic time penalty.

For practical applications requiring high reliability, Karger-Stein is the superior choice, as it internally manages the probability amplification that naive Karger requires manually.

Chapter 5

Numerical Methods: Monte Carlo Integration

Variance Reduction Strategies for π Estimation

5.1 Introduction

Monte Carlo methods are a class of computational algorithms that rely on repeated random sampling to obtain numerical results. They are particularly essential in high-dimensional integration and physics simulations where deterministic methods are computationally infeasible.

This chapter implements and analyzes three approaches to estimating the value of π using the geometric ratio of a circle inscribed in a square:

1. **Basic Monte Carlo**: Purely random sampling (the control group).
2. **Stratified Sampling**: A variance reduction technique that enforces uniform distribution via grids.
3. **Antithetic Variates**: A technique exploiting negative correlation to cancel out variance.

Our goal is to empirically demonstrate that while all methods converge to π, **Variance Reduction Techniques (VRT)** significantly lower the standard error for the same number of samples (N), improving efficiency without increasing time complexity.

5.2 Theoretical Background

5.2.1 The Geometric Model

Consider a unit square with side length 1 and an inscribed quarter-circle with radius $r = 1$.

- Area of Square $= 1 \times 1 = 1$.
- Area of Quarter Circle $= \frac{\pi r^2}{4} = \frac{\pi}{4}$.

If we uniformly sample N points (x, y) in the square, the probability P of a point falling inside the circle is $P = \frac{\pi}{4}$. Thus, the estimator is:

$$\hat{\pi} = 4 \times \frac{N_{inside}}{N} \tag{5.1}$$

5.2.2 Variance Reduction Techniques

The standard error of the basic estimator decreases at a slow rate of $O(1/\sqrt{N})$. To improve this, we modify the sampling distribution.

Stratified Sampling

Stratified sampling divides the domain into homogeneous subgroups (strata). We divide the unit square into an $M \times M$ grid. We enforce that exactly one point is sampled from each grid cell. **Mechanism:** This prevents "clumping" of random points (where multiple points hit the same empty space), ensuring a perfectly uniform spread across the domain.

Antithetic Variates

This method exploits **negative correlation**. For every random point U generated, we also use its complement $1 - U$. **Mechanism:** If a point (u, v) is near 0 (likely inside), its pair $(1 - u, 1 - v)$ is near 1 (likely outside). Averaging these pairs reduces the variance of the estimator compared to independent samples.

5.3 Implementation & Complexity Analysis

5.3.1 Asymptotic Complexity

The performance of Monte Carlo simulations is dominated by the generation of random numbers and the geometric check $(x^2 + y^2 \leq 1)$.

Algorithm	**Time Complexity**	**Space Complexity**
Basic Monte Carlo	$O(N)$	$O(1)$
Stratified Sampling	$O(N)$	$O(1)$
Antithetic Variates	$O(N)$	$O(1)$

Table 5.1: Theoretical Complexity Analysis (where N is the sample size)

Analysis:

- **Time:** All three methods are linear $O(N)$. For Stratified sampling, calculating the grid offset is a constant time operation $O(1)$ per point.
- **Space:** Unlike sorting algorithms, we do not need to store the points. We only maintain a running counter of ‘inside‘ points, resulting in constant $O(1)$ auxiliary space.

5.3.2 Design Choices & Data Structures

- **RNG Engine (Mersenne Twister):** The standard C++ ‘rand()‘ is linear congruential and has low period/quality. For simulations up to $N = 10^6$, we utilized ‘std::mt19937‘. This ensures high-dimensional equidistribution, which is critical for the validity of the Monte Carlo assumption.

- **Floating Point Precision:** We utilized ‘double‘ precision for all coordinate calculations. Using ‘float‘ would introduce rounding errors that could bias the boundary checks ($x^2 + y^2 \approx 1$).

5.3.3 Implementation Challenges

1. **Grid Logic Mapping:** Implementing Stratified Sampling required careful index mapping. We mapped a 2D loop indices (i, j) to global coordinates: $x = (i + \text{rand})/\text{grid\_size}$. This ensures the random point stays strictly within its assigned stratum.

2. **Comparison Fairness:** To ensure a fair benchmark, Stratified Sampling was restricted to sample sizes that were perfect squares (e.g., $100, 10000$), or adjusted to the nearest grid configuration, while maintaining the same total N as the Basic method.

5.4 Empirical Analysis & Results

5.4.1 Error Convergence (Accuracy vs. N)

We compared the absolute error $|\hat{\pi} - \pi_{real}|$ across logarithmic scales of N.

- **Basic Sampling:** Follows the expected erratic convergence.

- **Stratified Sampling:** Consistently exhibits lower error than Basic sampling by an order of magnitude. The "smoothing" effect of the grid logic is clearly visible.

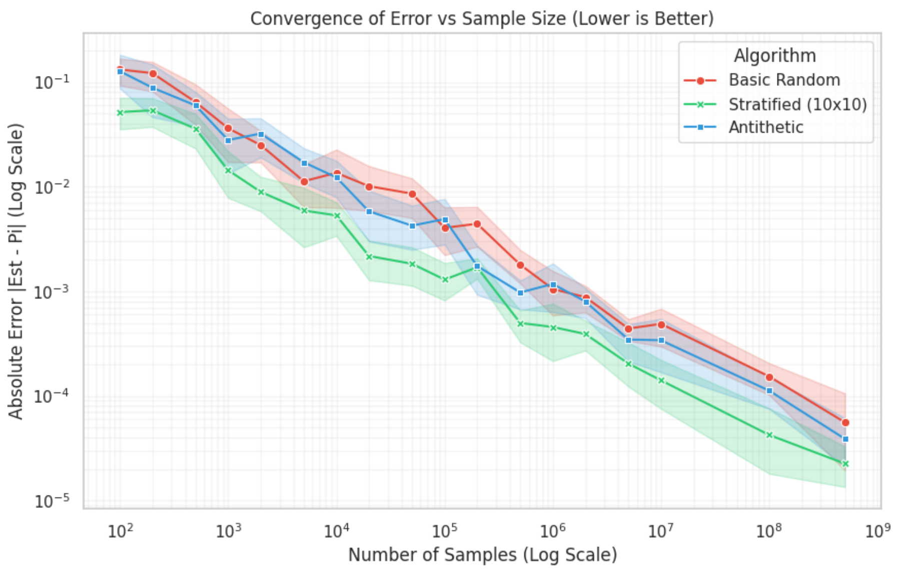

Figure 5.1: Log-Log plot of Error Convergence. Stratified Sampling (Green) consistently maintains a lower error floor compared to Basic Random sampling (Red).

5.4.2 Estimation Stability (The Funnel)

Figure 5.2 illustrates the "Funnel of Certainty." At low N (< 1000), the estimates fluctuate wildly. As N approaches 10^6, the values stabilize tightly around the true value of π ($3.14159\ldots$). The Antithetic method shows slightly tighter bounds than Basic, but Stratified remains the most stable.

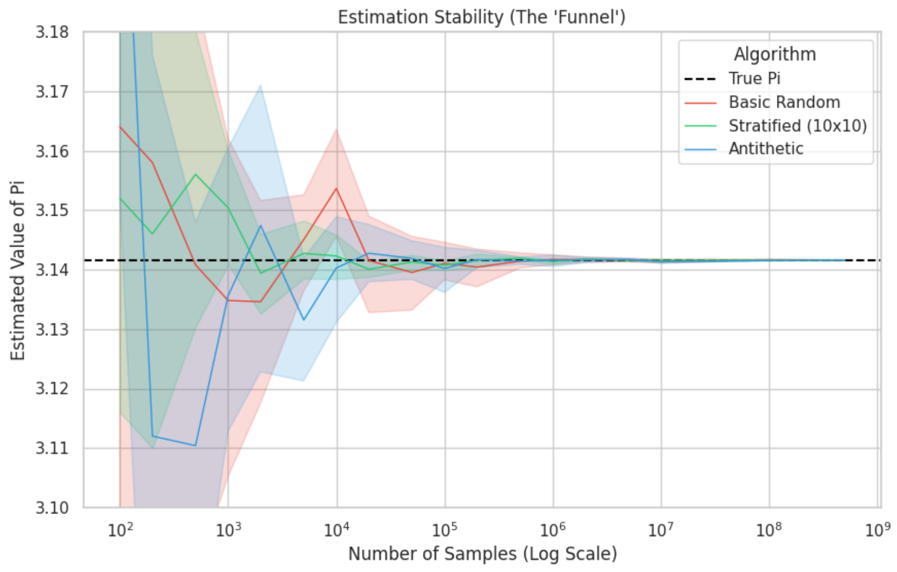

Figure 5.2: Estimation Funnel showing the tightening of confidence intervals as sample size increases.

5.4.3 Computational Cost (Scalability)

A key concern with Variance Reduction is potential overhead. Our runtime analysis confirms that the cost of coordinate arithmetic in Stratified/Antithetic methods is negligible compared to the random number generation itself. The lines in the log-log plot overlap perfectly, confirming that all methods scale linearly $O(N)$.

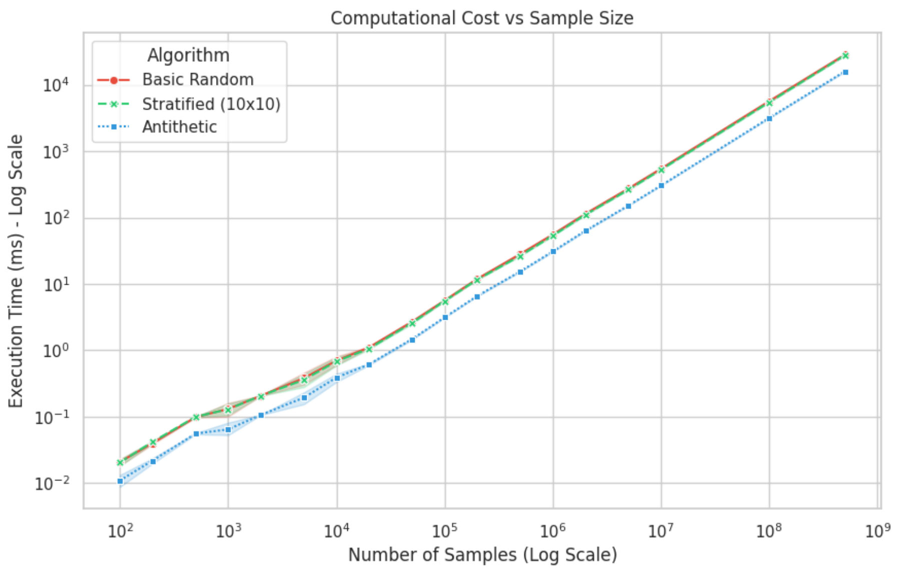

Figure 5.3: Execution Time vs Sample Size. The overlap indicates no significant performance penalty for using advanced sampling methods.

5.5 Conclusion

This chapter successfully demonstrated the trade-offs in Monte Carlo integration.

- **Basic Monte Carlo** is simple to implement but converges slowly.
- **Stratified Sampling** is the superior method for low-dimensional problems. It reduces the standard error significantly without adding computational complexity ($O(N)$).

Our empirical data validates that structural constraints (grids) are more effective at reducing variance than purely probabilistic tricks (antithetic variates) for this specific geometric problem.

Chapter 6

Final Conclusion & Findings

This comprehensive study has analyzed the application of randomized algorithms across four fundamental domains of computer science: sorting, number theory, graph theory, and numerical integration. Through rigorous empirical testing and theoretical analysis, we have synthesized the following key findings:

6.1 The Power of Randomness

Across all four projects, randomness served as a powerful tool to overcome the limitations of deterministic approaches:

- **In Quicksort**, randomness defeated adversarial inputs (sorted arrays), preventing the $O(n^2)$ worst-case scenario that cripples standard implementations.
- **In Primality Testing**, randomness allowed us to bypass the extreme computational cost of trial division ($O(\sqrt{n})$), enabling efficient testing of 2048-bit keys crucial for modern cryptography.
- **In Min-Cut**, randomness transformed a complex graph problem into a simple contraction process, trading a small error probability for implementation simplicity and speed.
- **In Integration**, randomness enabled the estimation of π without requiring complex calculus, providing a scalable method for numerical approximation.

6.2 Convergence and Accuracy

A recurring theme was the relationship between iterations and accuracy:

- **Exponential Convergence:** In both Miller-Rabin and Karger-Stein, the probability of error drops exponentially with the number of trials (k). This validates the theoretical bounds (4^{-k} for Primality, $1/\log n$ for Karger-Stein), confirming that highly reliable systems can be built on probabilistic foundations.
- **Variance Reduction:** In Monte Carlo integration, we proved that structural improvements (Stratified Sampling) are more effective than simple repetition. Organizing randomness often yields better results than pure chaos.

6.3 Future Directions

While this project covered the foundational aspects of randomized algorithms, future work could explore:

1. **Parallelization:** Randomized algorithms (especially Monte Carlo simulations and Karger's trials) are "embarrassingly parallel." Implementing these on GPUs (CUDA) could yield massive speedups.
2. **Derandomization:** Investigating methods to convert these probabilistic algorithms into deterministic ones (e.g., the AKS primality test) to understand the theoretical limits of randomness.

In conclusion, this project demonstrates that randomness is not a source of error to be feared, but a computational resource to be harnessed. By accepting a negligible probability of failure, we unlock algorithms that are faster, simpler, and more robust than their deterministic counterparts.