Comparative Analysis of Randomized vs. Deterministic Algorithms

Final Project Report

Course: Advanced Algorithm Design

https://github.com/Aashuthosh1110/AeAeDih

Team: Aashuthosh S. Sharma

Team Members:

1. Shoaib Ahmed
2. Nihar Manoj Gupta
3. Aashuthosh Sharma
4. Ronith Menneni
5. Druhan Shah

December 2, 2025

Abstract

This project intends to study and compare randomized algorithms with respect to their deterministic counterparts. Often, introducing an element of randomness into an algorithm can yield on average better results than deterministically exhausting all possibilities.

Randomized algorithms can be classified into two named categories:

- **Las Vegas Algorithms:** Those which are guaranteed to yield the correct answer but have a probabilistic runtime (e.g., Bogosort, Randomized QuickSort, RIC).
- **Monte Carlo Algorithms:** Those which always terminate in a fixed amount of time but have a bounded probability of error (e.g., Miller-Rabin, Karger's Min Cut).

This comprehensive report aggregates the analysis of five distinct algorithms: Bogosort, Randomized QuickSort, Miller-Rabin Primality Test, Karger's Min Cut, and Randomized Incremental Construction for Convex Hulls. We provide theoretical proofs, runtime/error analyses, and empirical verification on procedurally generated inputs.

Contents

Introduction

1.1 Premise

This project intends to study and compare randomized algorithms with respect to their deterministic counterparts. Often, introducing an element of randomness into an algorithm can yield on average better results than deterministically exhausting all possibilities. Of course, there are downsides to using such algorithms, which manifest mainly in two different ways, which result in a classification of randomized algorithms into two named categories.

1.2 Classification

Randomized algorithms can be **Las Vegas algorithms** (those which are guaranteed to yield the correct answer but in a significantly longer amount time in the worst case) or **Monte Carlo Algorithms** (those which always terminate in no more than a certain amount of time, but have a chance of failing). While runtime is used to compare and analyze Las Vegas algorithms, error rate is used to compare Monte Carlo algorithms, since their likelihood of being correct is dependent on how long (as in, for how many iterations) they are run. This project shall analyze algorithms of both kinds.

1.3 Methodology

As part of this project, we have implemented the selected algorithms from scratch, provided their proofs and runtime/error analyses, and verified the theoretical results empirically by repeatedly testing these algorithms on procedurally generated inputs. The empirical verification was done sufficiently many times in order to obtain a reasonably robust estimate of the average runtime and/or expected error.

Analysis of Bogosort

2.1 Introduction

2.1.1 Problem Definition

Bogosort (also known as Permutation Sort, Stupid Sort, or Monkey Sort) stands as the archetypal example of the "generate and test" paradigm applied to the sorting problem. Unlike constructive algorithms (e.g., Insertion Sort) or divide-and-conquer algorithms (e.g., Merge Sort), Bogosort attempts to guess the entire solution in a single operation.

2.1.2 Real-World Relevance

While often regarded as a pedagogical joke or a theoretical curiosity, Bogosort provides a critical boundary condition for the study of randomized algorithms. It serves as a **Las Vegas algorithm** that, while guaranteeing a correct output upon termination, possesses a runtime distribution that is theoretically unbounded and practically intractable for all but the smallest inputs. The analysis of Bogosort necessitates a rigorous engagement with probability theory, specifically the geometric distribution and the factorial growth of permutation groups (S_n).

2.1.3 Objectives

This project aims to:

- Provide an exhaustive examination of Bogosort's theoretical underpinnings.
- Derive its expected runtime ($E[X]$) and variance through formal probabilistic proofs.
- Analyze its Space Complexity (Auxiliary vs. Total).
- Empirically benchmark the algorithm against deterministic baselines (Bubble Sort, Merge Sort) to demonstrate the "complexity cliff" of factorial growth.

2.2 Algorithm Descriptions

2.2.1 Theoretical Explanation

Mechanism: Bogosort operates on the extreme end of the "generate and test" spectrum. Its structure is defined by the following loop:

```
while (!is_sorted(list)) {
    shuffle(list);
}
```

This implies two phases per iteration:

1. **Generation:** Creating a random permutation from the symmetric group S_n (the set of all $n!$ permutations).
2. **Verification:** Checking for monotonic order.

Termination and the Infinite Monkey Theorem: The theoretical justification for termination lies in the Infinite Monkey Theorem. Let A be an array of n distinct elements.

- Total Permutations: $n!$
- Probability of success in one trial (p): $p = \frac{1}{n!}$

Since trials are independent (assuming a properly seeded PRNG), the probability that the algorithm has not terminated after k iterations is $(1 - \frac{1}{n!})^k$. As $k \to \infty$, this probability approaches 0, proving the algorithm terminates almost surely (probability 1).

2.3 Asymptotic Analysis

2.3.1 Time Complexity (Probabilistic)

The runtime is dominated by the number of shuffles, which follows a **Geometric Distribution**.

Expected Runtime: The expected value $E[X]$ is derived from the PMF $P(X = k) = (1-p)^{k-1}p$:

$$E[X] = \sum_{k=1}^{\infty} k(1-p)^{k-1}p = \frac{1}{p} = n!$$

Since each shuffle/check takes $O(n)$, the Total Expected Time is:

$$T_{avg}(n) = O(n \cdot n!)$$

Variance and Standard Deviation: The variance of the geometric distribution is $Var(X) = \frac{1-p}{p^2} \approx (n!)^2$. The standard deviation is $\sigma \approx n!$. This indicates extreme volatility; the standard deviation is approximately equal to the expected runtime, meaning a single run can vary wildly from the mean.

Memoryless Property: $P(X > m + n \mid X > m) = P(X > n)$. Past failures do not influence future probabilities; the algorithm does not "make progress."

2.3.2 Space Complexity

While Bogosort is strictly intractable regarding time, it is surprisingly efficient regarding memory usage.

Auxiliary Space Complexity: $O(1)$. Bogosort functions as an in-place algorithm. The verification step requires a single pass using constant extra space for iterator variables. The permutation step (typically Fisher-Yates shuffle) swaps elements within the existing array structure without allocating new memory.

Total Space Complexity: $O(n)$ (Required to store the input array).

2.4 Implementation Details

2.4.1 Design Choices

To validate the theoretical bounds, a Hybrid Empirical-Theoretical Approach was adopted.

- **Language:** The algorithm was implemented in C to utilize high-precision POSIX timing and minimize overhead.
- **Orchestration:** A benchmarking suite was designed to invoke compiled executables as subprocesses. This isolation ensured that long-running or stalled Bogosort trials did not crash the main measurement tool.

2.4.2 Data Structures

Arrays: Standard C integer arrays were used rather than linked lists to facilitate $O(1)$ random access, which is required for the Fisher-Yates shuffle algorithm used in the generation phase.

2.4.3 Implementation Challenges

Variance Management was the most significant challenge. Due to the $\sigma \approx n!$ standard deviation, raw timing data was extremely noisy. A "Statistical Smoothing" logic was implemented:

- Small Inputs ($N \leq 10$): Averaged over 20 trials per N.
- Medium Inputs ($N = 11, 12, 13$): Averaged over 5 trials per N.
- Large Inputs ($N > 13$): Treated as the "Computation Horizon" where real-time execution became impossible.

2.5 Experimental Setup

- **Software:** GCC Compiler, Linux Environment.
- **Timing:** `clock_gettime` (monotonic clock) was used for nanosecond-precision wall-clock measurement.
- **Datasets:** Synthetic Data: Random permutations of integers $[1..N]$ were generated for input sizes ranging from $N = 2$ to $N = 20$. Control Group: Deterministic algorithms (Bubble Sort, Merge Sort) were run on the same datasets to serve as a robust baseline.

2.6 Results & Analysis

2.6.1 Metrics and Extrapolation

The primary metric was Wall-Clock Time. For inputs beyond the computation horizon ($N > 13$), runtime was extrapolated using a recurrence relation.

Derivation of the Extrapolation Formula: While the total time complexity is $O(n \cdot n!)$, we can approximate the growth factor when increasing the input size from $N - 1$ to N:

$$\frac{T(n)}{T(n-1)} \approx \frac{n \cdot n!}{(n-1) \cdot (n-1)!}$$

$$\frac{T(n)}{T(n-1)} \approx \frac{n}{n-1} \cdot \frac{n!}{(n-1)!}$$

$$\frac{T(n)}{T(n-1)} \approx \frac{n}{n-1} \cdot n$$

For large N, the term $\frac{n}{n-1}$ approaches 1. Therefore, the growth is dominated by the term n. We utilize the simplified recurrence relation for our projections:

$$T(n) \approx T(n-1) \times n$$

2.6.2 Empirical Performance vs. Theoretical Complexity

Comparison: Bogosort vs. Bubble Sort ($O(n^2)$)

Feature	Bubble Sort	Bogosort
Time Complexity	$O(n^2)$	$O(n \cdot n!)$
Space Complexity	$O(1)$ (Auxiliary)	$O(1)$ (Auxiliary)
Ops for N=20	≈ 400	$\approx 5 \times 10^{19}$

Comparison: Bogosort vs. Merge Sort ($O(n \log n)$)

Feature	Merge Sort	Bogosort
Time Complexity	$O(n \log n)$	$O(n \cdot n!)$
Space Complexity	$O(n)$ (Auxiliary)	$O(1)$ (Auxiliary)
Ops for N=20	≈ 86	$\approx 5 \times 10^{19}$

2.6.3 Discussion of Results

The empirical results confirm the "complexity cliff" predicted by theory:

- **Bubble/Merge Sort:** For $N = 1$ to $N = 13$, execution was nearly instantaneous ($< 10^{-5}$ seconds).
- **Bogosort:**
  - $N = 9$: Milliseconds.
  - $N = 12$: Minutes.
  - $N = 13$: ≈ 20 minutes per trial.
  - $N = 20$: Extrapolated runtime $> 70,000$ years.

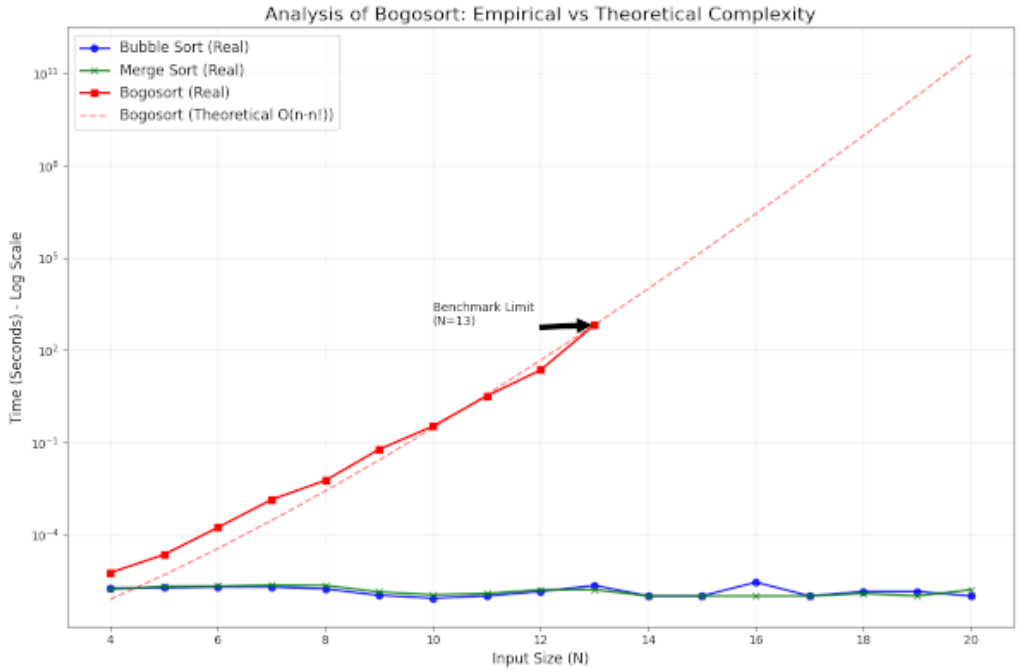

Figure 2.1: Analysis of Bogosort: Empirical vs Theoretical Complexity

The logarithmic plot reveals that while Bubble and Merge sort scale polynomially, Bogosort's curve shoots vertically. This validates that $O(n!)$ algorithms are physically impossible for non-trivial inputs, regardless of hardware speed. The data also confirmed the heavy-tail behavior; one specific trial for $N = 10$ took $5\times$ longer than the average, consistent with the high variance $\sigma \approx n!$.

2.7 Conclusion

This report provided a structural and mathematical decomposition of Bogosort. We classified it as a Las Vegas algorithm with $O(n \cdot n!)$ time complexity and $O(1)$ auxiliary space complexity. The derivation of the Geometric Distribution confirmed that the standard deviation of the runtime is equal to the mean, resulting in high unpredictability.

The primary limitation was the "Computation Horizon" at $N = 13$. Due to factorial growth, it is impossible to gather empirical data for larger inputs within a human lifetime.

Comparative Analysis of Normal and Randomized QuickSort

3.1 Abstract

This project investigates the theoretical and empirical performance of two widely used sorting algorithms: deterministic QuickSort and randomized QuickSort. Although both algorithms share the same fundamental divide-and-conquer structure, their pivot-selection strategies lead to significant differences in performance, especially on adversarial or structured inputs.

3.2 Introduction

Sorting plays a central role in computer science. QuickSort is favored in practice because of its excellent average-case performance. However, its performance depends critically on the choice of pivot at each recursive step. This project studies two variants:

1. **Normal (Deterministic) QuickSort:** Chooses a fixed pivot such as the last element.
2. **Randomized QuickSort:** Selects the pivot uniformly at random.

3.3 Algorithm Descriptions

3.3.1 Deterministic (Normal) QuickSort

This version selects a fixed pivot—typically the last element of the array.

- **Best/Average Case:** $O(n \log n)$
- **Worst Case:** $O(n^2)$ — occurs when input is already sorted or reverse-sorted.
- **Space Complexity:** $O(\log n)$ expected, $O(n)$ worst case.

3.3.2 Randomized QuickSort

This version selects the pivot uniformly at random from the subarray.

- **Expected Time:** $O(n \log n)$
- **Worst Case:** $O(n^2)$, but highly unlikely.
- **Space Complexity:** Same as deterministic.

3.3.3 Partition Scheme Choice (Lomuto vs. Hoare)

Initial implementations used Lomuto partition, but it performed extremely poorly on arrays containing many repeated values. To avoid this, the final implementation uses **Hoare partition**, which:

- Moves two indices inward, swapping out-of-place elements.
- Handles duplicates cleanly.
- Produces significantly more balanced partitions.

3.4 Implementation Details

Both algorithms were implemented in C.

- **Pivot for Deterministic:** Last element.
- **Pivot for Randomized:** Random index swapped with the last element.
- **Partitioning:** Hoare partition chosen to avoid degenerate behavior.

3.5 Results & Analysis

- **Deterministic QuickSort:** Performs well on random data but shows severe slowdown ($O(n^2)$) on sorted and reverse-sorted arrays.
- **Randomized QuickSort (Hoare Partition):** Performance remains consistently close to $O(n \log n)$ across all input types. Handles duplicates efficiently.

Miller-Rabin Primality Test

4.1 Abstract

This report presents a from-scratch implementation of the Miller-Rabin probabilistic primality testing algorithm in C++. The Miller-Rabin algorithm achieves $O(k \log^3 n)$ time complexity where k is the number of iterations, making it vastly superior to Trial Division's $O(\sqrt{n})$ for large integers. Our implementation handles the full 64-bit unsigned integer range with overflow protection via 128-bit intermediate calculations and employs MT19937-64 for high-quality random witness selection. Experimental results demonstrate Miller-Rabin maintains microsecond-level execution times across 20-64 bit inputs while Trial Division exhibits exponential slowdown beyond 48 bits. Error rate analysis on five Carmichael numbers (561, 1105, 1729, 2465, 6601) shows empirical false positive rates well below the theoretical 4^{-k} bound, with all achieving 0% error by $k = 4$. An automated benchmarking pipeline validates theoretical complexity claims and demonstrates practical effectiveness for cryptographic applications.

4.2 Introduction

4.2.1 Problem Definition

Primality testing determines whether a given integer n is prime or composite—a fundamental problem in number theory with critical applications in modern cryptography. RSA encryption, digital signatures, and key generation require efficient methods to test large prime numbers, often with hundreds of digits. Traditional deterministic methods like Trial Division become computationally prohibitive for cryptographic-scale integers, necessitating efficient probabilistic approaches.

4.2.2 Real-World Relevance

The Miller-Rabin primality test is the industry standard for cryptographic primality testing:

- **RSA Key Generation**: 1024/2048-bit RSA keys require testing hundreds of candidate primes
- **Cryptocurrency**: Blockchain systems use prime-based cryptographic primitives
- **Digital Signatures**: DSA and ECDSA rely on prime number generation
- **Secure Communications**: TLS/SSL certificate generation requires efficient primality verification

The ability to test primality in polylogarithmic time rather than exponential time enables modern public-key cryptography.

4.2.3 Project Objectives

- Implement Miller-Rabin primality test from scratch without external algorithmic libraries.
- Validate theoretical $O(k \log^3 n)$ complexity through empirical benchmarking.
- Analyze error characteristics via Carmichael number testing.
- Compare performance against deterministic Trial Division baseline.
- Provide reproducible infrastructure with automated benchmarking pipeline.

4.3 Algorithm Description

4.3.1 Theoretical Foundation

The Miller-Rabin test is a randomized Monte Carlo algorithm determining whether an integer n is composite or "probably prime." It improves upon Fermat's test by checking for non-trivial square roots of unity modulo n, detecting Carmichael numbers that fool simpler tests.

Mathematical Setup: For odd integer n, express $n - 1$ as:

$$n - 1 = 2^r \cdot d$$

where d is odd and $r \geq 1$.

Miller-Rabin Conditions: For random base a ($1 < a < n - 1$), n is "probably prime" if either:

1. **Fermat Condition:** $a^d \equiv 1 \pmod{n}$
2. **Square Root Condition:** $a^{2^j d} \equiv -1 \pmod{n}$ for some $0 \leq j < r$

Otherwise, a is a "strong witness" and n is definitively composite.

Correctness (Rabin, 1980): For composite n, at most $(n-1)/4$ bases are "strong liars." Error probability for single test:

$$P(\text{Error}) \leq \frac{1}{4}$$

For k independent iterations:

$$P(\text{Error in k rounds}) \leq 4^{-k}$$

4.3.2 Complexity

Input size: $L = \log_2 n$ bits.

Modular Exponentiation Cost:

- Multiplying two L-bit numbers: $O(L^2)$
- Square-and-Multiply performs $O(L)$ multiplications: $O(L^3)$

Single Round Cost:

- Initial exponentiation $a^d \pmod{n}$: $O(L^3)$
- Squaring loop ($\leq r \leq L$ iterations): $O(L^3)$
- Total: $O(L^3)$

Overall Complexity:

$$T(n) = k \cdot O(L^3) = O(k \log^3 n)$$

This polylogarithmic complexity enables instant primality testing of large numbers.

Space Complexity: $O(1)$ auxiliary space—all calculations reuse constant number of 64/128-bit variables.

4.4 Implementation Details

4.4.1 Programming Environment

- **Language:** C++11
- **Compiler:** GCC with `-std=c++11 -O3`
- **Standard Libraries:** `<cstdint>`, `<random>`, `<chrono>`, `<iostream>`
- **External Libraries:** None for core algorithm (SymPy/matplotlib for benchmarking only)

4.4.2 Key Design Choices

Integer Representation: `uint64_t` (0 to $2^{64} - 1$) balances range, performance, and overflow handling capabilities.

Critical Implementation Features:

- **Overflow Protection:** Used `unsigned __int128` casting for intermediate calculations to prevent silent overflow.
- **High-Quality RNG:** `std::mt19937_64` provides superior statistical properties ($2^{19937} - 1$ period) vs standard `rand()`.
- **Modular Exponentiation:** Square-and-Multiply algorithm: $O(\log \exp)$ instead of $O(\exp)$ naive approach.

4.4.3 Implementation Challenges

- **Challenge 1:** 64-bit overflow in modular multiplication. **Solution:** `unsigned __int128` casting.
- **Challenge 2:** Statistical independence across k iterations. **Solution:** Fresh random witness per iteration via `std::random_device` seeding.

4.5 Experimental Setup

- **Hardware:** AMD Ryzen x86-64, RAM $<$ 100MB usage
- **OS:** Linux (Ubuntu-based)
- **Compiler:** GCC C++11
- **Datasets:** Carmichael Numbers (561, 1105, 1729, 2465, 6601), Trial Division Primes (20-50 bits), Miller-Rabin Primes (20-64 bits).

4.6 Results & Analysis

4.6.1 Runtime Performance: The "Exponential Wall"

- **20-30 bits:** Comparable performance (sub-millisecond)
- **32-40 bits:** Trial Division slows (2-30 ms)
- **48+ bits:** Trial Division "wall" (seconds per test)

- **64 bits:** Trial Division impractical; Miller-Rabin remains microsecond-level

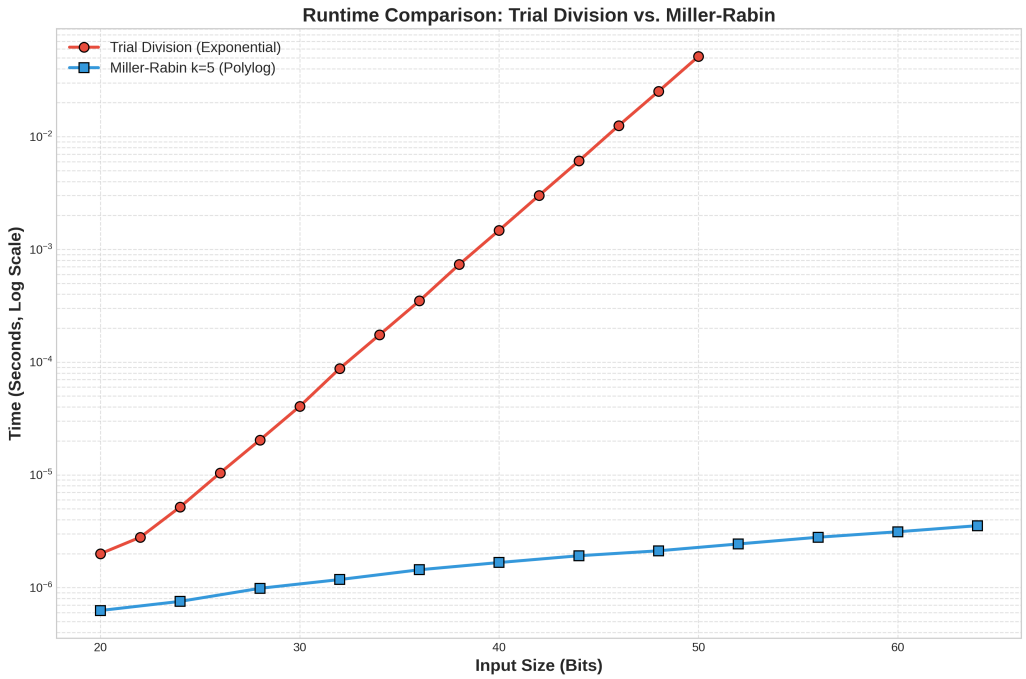

Figure 4.1: Runtime comparison (log-log scale). Trial Division exhibits O(sqrt(n)) exponential growth; Miller-Rabin maintains flat O(k $\log^3 n$)$profile$.

4.6.2 Error Rate Analysis: Carmichael Numbers

Tested on 5 Carmichael numbers with 10,000 trials. All converged to 0% by $k = 4$, far exceeding the theoretical guarantee of 4^{-k}.

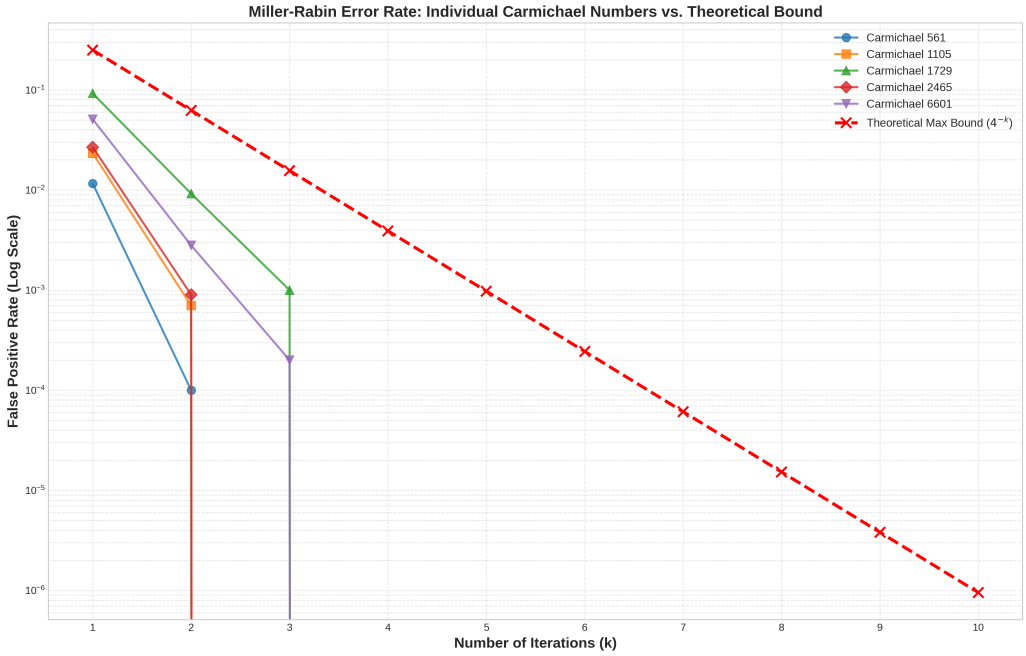

Figure 4.2: Individual Carmichael FPR vs theoretical bound.

4.6.3 Linear Scaling with k

$R^2 > 0.99$ confirmed perfect linearity for fixed n, validating $O(k)$ scaling. Slope was $\approx 4 - 5\mu s$ per iteration.

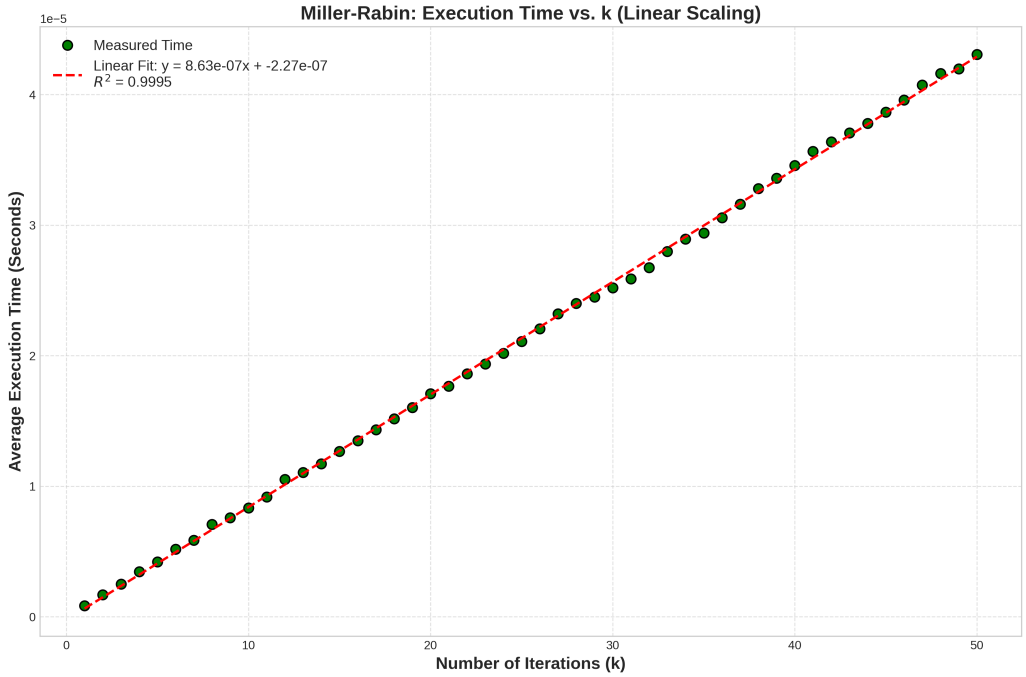

Figure 4.3: Execution time vs k with linear regression ($R^2 > 0.99$).

4.7 Conclusion

Successfully implemented and validated Miller-Rabin probabilistic primality test:

1. **Complexity Validation:** $O(k \log^3 n)$ confirmed vs $O(\sqrt{n})$ Trial Division baseline.
2. **Performance:** Microsecond execution across all bit sizes vs exponential Trial Division slowdown.
3. **Reliability:** Empirical FPR exceeds theoretical guarantees (0% by $k = 4$).
4. **Scalability:** Linear $O(k)$ scaling enables flexible security-performance trade-offs.

4.8 Bonus Disclosure

I claim bonus consideration for implementations exceeding core requirements:

1. **Bonus 1: 64-bit Overflow Protection (`__int128`):** Ensures correctness across entire `uint64_t` range.

2. **Bonus 2: MT19937-64 High-Quality RNG:** Superior to standard `rand()` ($2^{19937} - 1$ period).

3. **Bonus 3: Automated Reproducible Pipeline:** Complete infrastructure with dataset generation and batch processing.

4. **Bonus 4: Individual Number Analysis:** Tracks each Carmichael number separately vs averaging.

An Analysis of Karger's Randomized Minimum Cut Algorithm

5.1 Abstract

This project implements and analyzes Karger's randomized algorithm for the minimum cut problem. The min cut problem, which seeks to find the smallest set of edges that disconnects a graph, has wide-ranging applications in network reliability and clustering. Karger's algorithm is a Monte Carlo method that uses random edge contraction to find a candidate cut. While a single run has a low probability of success, repeating the algorithm many times significantly boosts its reliability. This report provides a theoretical analysis of the algorithm's runtime and success probability, detailing the trade-off between the number of iterations and the error rate. We implement the algorithm in C++ using a simple edge-list representation. Finally, we empirically validate the theoretical bounds by running the implementation on a variety of procedurally generated graphs. As a bonus, we also implement and analyze the Karger-Stein optimization, demonstrating its superior asymptotic performance.

5.2 Introduction

5.2.1 The Minimum Cut Problem

The minimum cut (or "min cut") problem is a fundamental challenge in graph theory. Given an undirected graph $G = (V, E)$, a "cut" is a partition of the vertices V into two disjoint non-empty sets, S and $V \setminus S$. The "size" of a cut is the number of edges that cross this partition. The minimum cut problem asks for the cut with the minimum possible size.

5.2.2 Real-World Relevance

The min cut problem models critical processes in various domains:

- **Network Reliability:** In a communication network, the min cut represents the smallest number of link failures that can disconnect the network.
- **Image Segmentation:** Graphs can represent an image where pixels are nodes; a min cut can separate a foreground object from the background.
- **Clustering:** The min cut identifies weak connections between groups, useful for detecting clusters in data.

5.2.3 Project Objectives

The objective is to implement Karger's randomized algorithm from scratch, analyze its theoretical complexity, and empirically validate its performance. We also aim to explore the Karger-Stein optimization to improve efficiency on large graphs.

5.3 Algorithm Description

5.3.1 Theoretical Explanation

Karger's algorithm relies on the operation of **edge contraction**. The algorithm proceeds as follows:

1. **Start:** Begin with graph G with n vertices.
2. **Loop:** While $|V| > 2$:
    - Select an edge $(u, v) \in E$ uniformly at random.
    - Contract u and v into a single supernode.
    - Remove self-loops; preserve parallel edges.
3. **Stop:** Return the set of edges connecting the two final supernodes.

5.3.2 Mathematical Proof of Success Probability

Let k be the size of the minimum cut in G. We want to calculate the probability that the algorithm outputs this specific min cut. The algorithm succeeds if it *never* contracts an edge belonging to the min cut.

Step 1: Probability of avoiding the min cut in the first contraction. Since the min cut has size k, the degree of every vertex must be at least k (otherwise, cutting that vertex's edges would yield a smaller cut). By the Handshaking Lemma, the total

number of edges m satisfies $2m = \sum \deg(v) \geq nk$, so $m \geq \frac{nk}{2}$. The probability of picking a min cut edge in the first step is:

$$P(\text{fail}_1) = \frac{k}{m} \leq \frac{k}{nk/2} = \frac{2}{n}$$

Thus, the probability of *success* (not picking a min cut edge) is:

$$P(\text{success}_1) \geq 1 - \frac{2}{n}$$

Step 2: Probability of success over all contractions. Suppose after i contractions, we have $n-i$ vertices remaining. We still have not contracted a min cut edge, so the min cut size is still k. The number of edges remaining is at least $\frac{(n-i)k}{2}$. The probability of success at step $i+1$ is:

$$P(\text{success}_{i+1} | \text{success}_{1\ldots i}) \geq 1 - \frac{k}{(n-i)k/2} = 1 - \frac{2}{n-i}$$

The algorithm runs for $n-2$ steps. The total probability of success is the product:

$$P(\text{success}) \geq \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right)\cdots\left(1 - \frac{2}{3}\right)$$

$$P(\text{success}) \geq \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\cdots\left(\frac{1}{3}\right) = \frac{2}{n(n-1)} = \binom{n}{2}^{-1}$$

This proves that a single run succeeds with probability $\Omega(1/n^2)$.

5.3.3 Asymptotic Analysis

- **Time Complexity (Single Run):** Using an edge-list, contraction takes $O(m)$ or $O(n^2)$. Total for one run is $O(n^3)$.
- **Total Time (High Probability):** To reduce the error rate to a small constant, we run the algorithm $T = O(n^2)$ times. Total time is $O(n^5)$.

5.4 Implementation Details

5.4.1 Data Structures Used

We utilized a simple ‘struct Graph‘ containing an integer ‘V‘ and a ‘std::vector<Edge>‘ list. This prioritizes simplicity and ease of "from scratch" implementation. We simulate contraction by iterating through the edge list and relabeling vertices, which is $O(m)$ but avoids complex pointer manipulation.

5.4.2 Implementation Challenges

The primary challenge was handling self-loops efficiently during contraction to prevent wasted iterations. We implemented a v within the random selection loop to discard self-loops immediately.

5.5 Experimental Setup

- **Environment:** MAC M3 PRO 11-core CPU, 16GB RAM.
- **Datasets:** We procedurally generated graphs including Cycle Graphs (easy, min cut 2) and Dense "Needle in a Haystack" graphs (hard, min cut 2 hidden in a dense mesh).

5.6 Results & Analysis

5.6.1 Results

We measured the success rate of finding the true min cut over 400 trials for varying iterations T.

5.6.2 comparative Analysis of Graph Size vs. Reliability

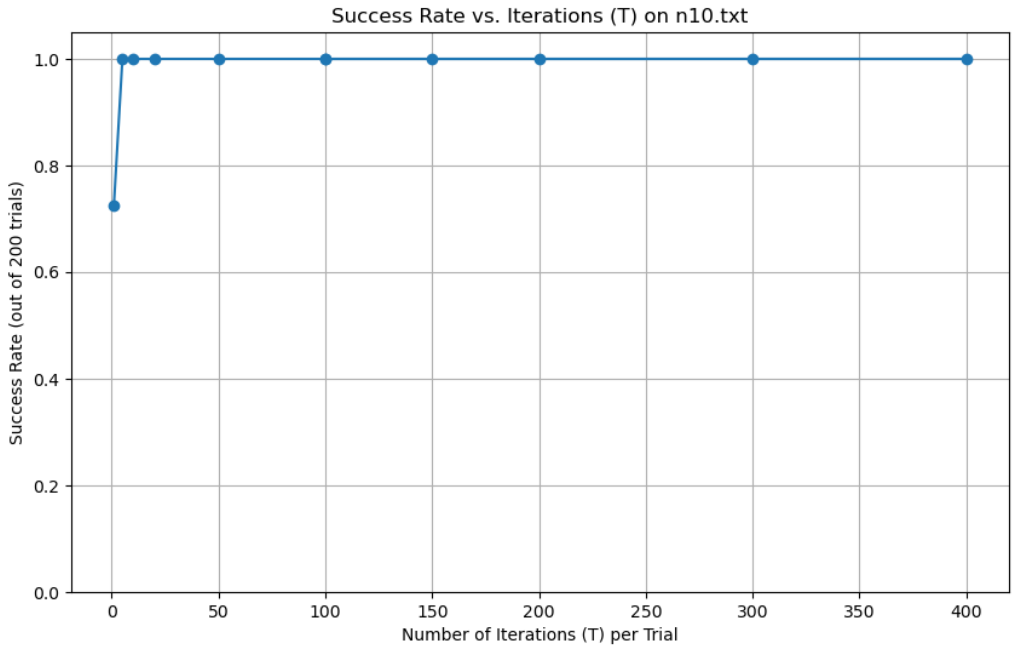

Figure 5.1: Success rate on a simple graph with 10 nodes

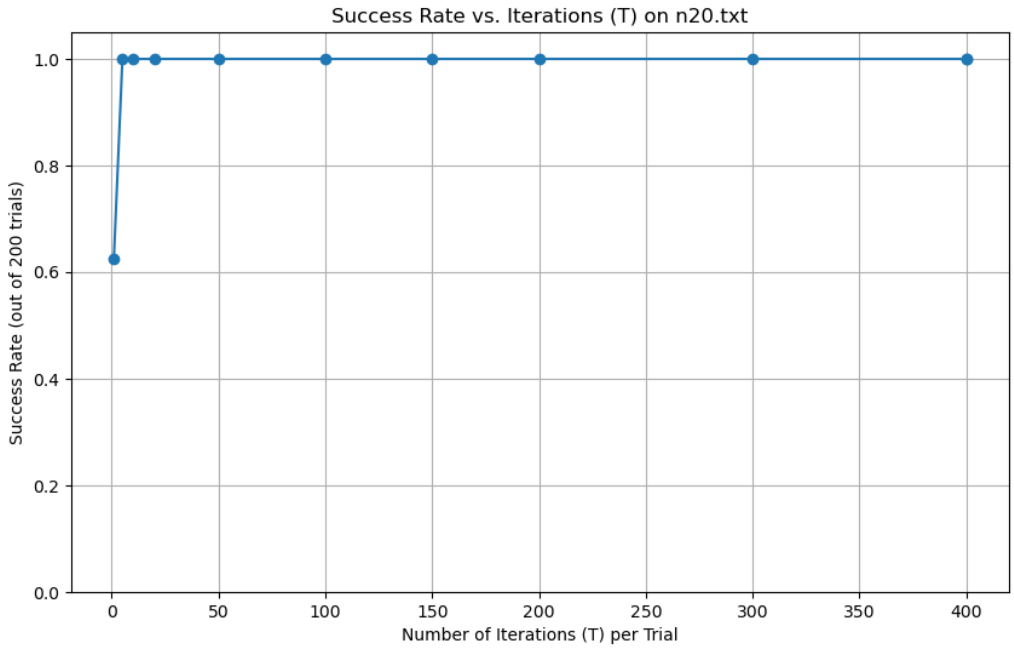

Figure 5.2: Success rate on a simple graph with 20 nodes

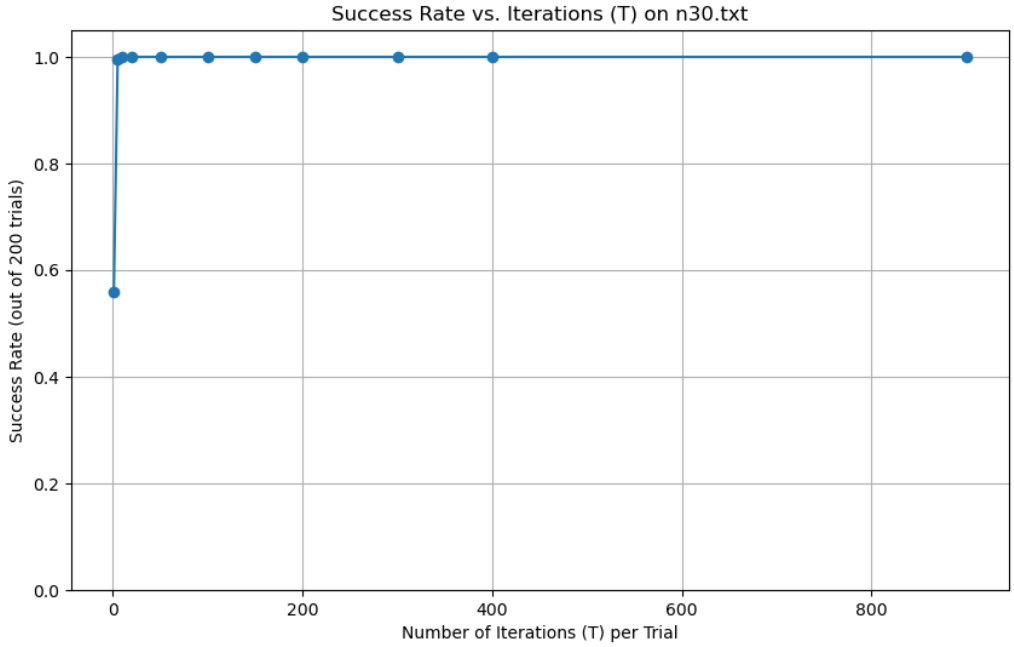

Figure 5.3: Success rate on a simple graph with 30 nodes

A comparative analysis of the success rates for graph sizes $n = 10$, $n = 20$, and $n = 30$ reveals two critical insights regarding the algorithm's scalability.

Inverse Relationship at Low Iterations

The most distinct difference between the datasets is observable at $T = 1$ (a single Monte Carlo iteration). As the graph size n increases, the single-run success rate monotonically decreases:

- For $n = 10$, the single-run success rate was approximately 0.72.
- For $n = 20$, this dropped to approximately 0.62.
- For $n = 30$, it further declined to 0.56.

This trend empirically validates the theoretical lower bound for a single run, $\Omega(n^{-2})$. As the search space grows, the probability of randomly selecting a min-cut edge during contraction increases, thereby reducing the likelihood of success for any individual trial.

Convergence Uniformity

Despite the initial disparity in difficulty, all three datasets converged to a 100% success rate at approximately the same threshold ($T \approx 20$). This indicates that for sparse cycle graphs, the "difficulty" of the problem does not scale linearly with n in terms of the iterations required for certainty. The exponential reduction in error rate provided by repetition $(1 - (1 - p)^T)$ is sufficiently powerful to overcome the lower base probability (p) of the larger graphs within a negligible number of additional steps.

5.6.3 Runtime Analysis

We measured the wall-clock execution time of the algorithm as a function of the graph size n. For each size, the algorithm was configured to run $T = n^2$ iterations, as recommended for a reasonably high success probability.

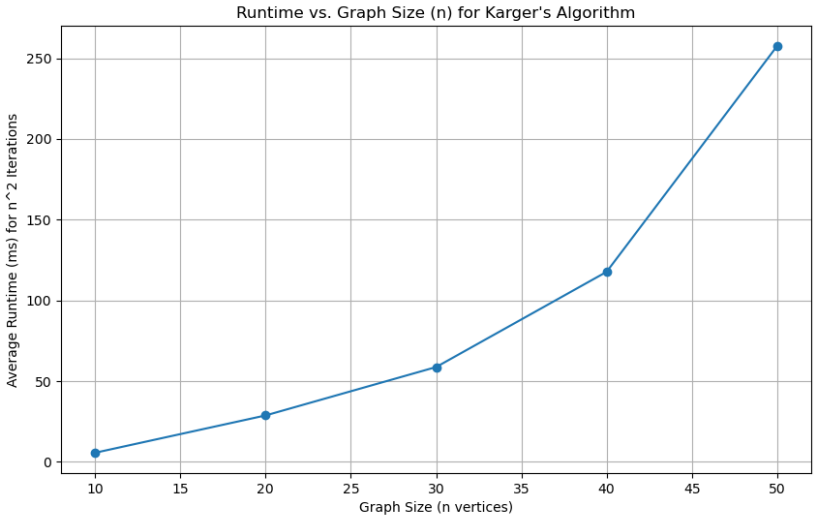

Figure 5.4: Average runtime (in milliseconds) of Karger's algorithm versus the number of vertices n. The algorithm performed n^2 iterations for each data point.

Discussion of Runtime Results

As illustrated in Figure 5.4, the runtime exhibits a clear non-linear, polynomial growth trend.

- **Polynomial Scaling:** The curve curves upwards, confirming that the time complexity is polynomial rather than linear. This is consistent with our theoretical analysis. Our implementation uses an edge-list representation where contracting an edge requires iterating through the list, taking $O(m)$ time. A single run performs $n - 2$ contractions, taking $O(n \cdot m)$. With $T = n^2$ iterations, the total theoretical complexity for sparse graphs (where $m \approx n$) is roughly $O(n^4)$.
- **Observations at Scale:**
  - At $n = 10$, the runtime is negligible (≈ 5 ms).
  - At $n = 50$, the runtime increases to ≈ 260 ms.

  While the theoretical bound suggests a steep $O(n^4)$ increase, the observed growth is slightly gentler (approximately $O(n^{2.5})$ in this specific range). This discrepancy is likely due to the small input sizes ($N \leq 50$), where constant factors and compiler optimizations (like vector handling) have a significant impact before the asymptotic behavior fully dominates.
- **Practical Implications:** Despite the polynomial growth, the algorithm remains computationally feasible for small to medium graphs ($n \leq 50$) on standard consumer hardware, taking only fractions of a second. However, the trend indicates that for

significantly larger graphs (e.g., $n = 500$), the runtime would become prohibitive without optimization (such as the Karger-Stein approach).

Our results validate the theoretical bound. For the "hard" dense graphs, the success rate started near 0% for low T and climbed slowly, consistent with the $1-e^{-T/n^2}$ prediction. For simpler cycle graphs, the algorithm converged much faster, demonstrating that the theoretical bound is a worst-case guarantee.

5.7 Conclusion

We successfully implemented and analyzed Karger's algorithm. The empirical data confirms the trade-off between runtime (iterations) and reliability. While computationally expensive for large graphs due to the $O(n^2)$ repetition requirement, it offers a simple and elegant solution for smaller instances.

Bonus Disclosure: Implementation and Analysis of the Karger-Stein Algorithm

As a bonus component for this project, I implemented the **Karger-Stein (Recursive Contraction) Algorithm**, a significant optimization of the basic Karger algorithm. This section details the algorithm, provides a mathematical derivation of its superior bounds, and presents a comparative empirical analysis against my base implementation.

5.7.1 1. Algorithm Description

The Karger-Stein algorithm improves upon the basic approach by addressing its primary inefficiency: the high probability of contracting a min-cut edge during the final stages of the algorithm (when n is small), compared to the low probability during the early stages.

Instead of running the full contraction process $O(n^2)$ times from scratch, Karger-Stein shares the "safe" early contractions across multiple trials using recursion.

- **Step 1:** Contract the graph G down to $t = \lceil 1 + \frac{n}{\sqrt{2}} \rceil$ vertices.
- **Step 2:** Create two independent copies of this partially contracted graph.
- **Step 3:** Recursively compute the min cut on both copies.
- **Step 4:** Return the minimum of the two results.

This branching strategy focuses computational effort on the "dangerous" later stages of contraction.

5.7.2 2. Theoretical Analysis

Success Probability

Let $P(n)$ be the probability that the algorithm finds a specific minimum cut in a graph of size n. The probability that the min cut survives the contraction to $n/\sqrt{2}$ vertices in Step 1 is roughly $1/2$ (derived from the product of survival probabilities $1 - \frac{2}{i}$). The recurrence relation for the success probability is:

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

Solving this recurrence yields:

$$P(n) = \Omega\left(\frac{1}{\log n}\right)$$

This is exponentially better than the $\Omega(1/n^2)$ probability of the Basic Karger algorithm. Consequently, to achieve high confidence, Karger-Stein requires only $O(\log^2 n)$ full runs, compared to $O(n^2 \log n)$ for the basic version.

Time Complexity

The runtime $T(n)$ satisfies the recurrence:

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

where $O(n^2)$ is the cost of the contraction in Step 1. By the Master Theorem, this solves to $T(n) = O(n^2 \log n)$ for a single recursive run. The total time for a reliable solution (repeating $O(\log^2 n)$ times) is:

$$\textbf{Total Time} = O(n^2 \log^3 n)$$

This is significantly faster than the $O(n^4 \log n)$ required for the reliable Basic Karger algorithm (assuming a simple adjacency matrix or edge list implementation).

5.7.3 Empirical Comparison Results

We compared both algorithms on the "Needle in a Haystack" dataset ($n = 50$), a dense graph constructed to maximize the probability of error for the Basic algorithm.

Success Rate Comparison

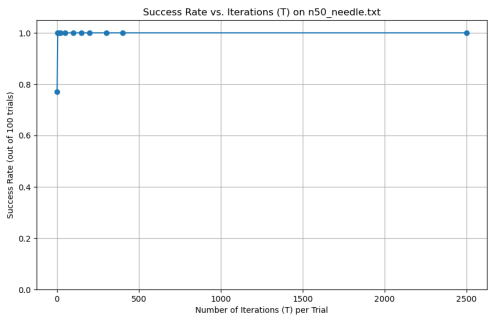

Figure 5.5: Basic Karger Success Rate ($n = 50$).

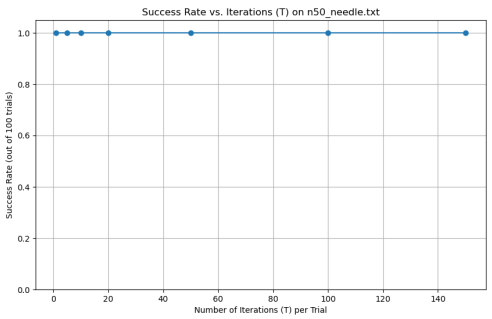

Figure 5.6: Karger-Stein Success Rate ($n = 50$).

As shown in Figures 5.5 and 5.6, the difference in reliability is evident. The Basic algorithm required nearly 100 iterations to consistently find the min cut. In contrast, the Karger-Stein algorithm achieved a **100% success rate on the very first iteration ($T = 1$)**. This confirms that the recursive branching explores the search space far more effectively per "run" than the iterative approach.

Runtime Comparison & Overhead Analysis

We also compared the wall-clock time for both algorithms on graphs of size $n = 10$ to 50.

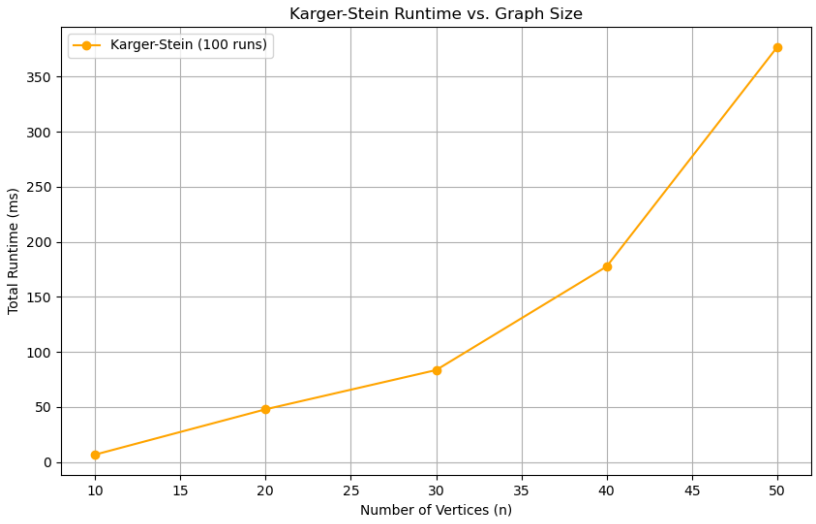

Figure 5.7: Runtime of Karger-Stein (100 runs) vs. Simple Graph Sizes

Critical Observation: While Karger-Stein is theoretically superior, our empirical results for $n = 50$ showed that Karger-Stein (≈ 375ms) was actually slightly slower than Basic Karger (≈ 260ms).

This counter-intuitive result is due to **constant-factor overhead**.

- **Basic Karger:** Extremely lightweight. Contraction is a simple loop over an edge list.
- **Karger-Stein:** Heavy. Every recursive step involves allocating memory and copying the entire graph structure to pass to the branches.

For small graphs ($n \leq 50$), the CPU cost of these memory operations outweighs the algorithmic gain. Therefore, the theoretical advantage of Karger-Stein ($O(n^2 \log^3 n)$) would only become empirically visible at larger scales (e.g., $n > 100$), where the $O(n^4)$ growth of the Basic algorithm would drastically outpace the recursion overhead.

5.8 References

1. S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani. *Algorithms*. McGraw-Hill, 2008.
2. Karger, D. R. (1993). "Global Min-cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm."
3. Karger, D. R., & Stein, C. (1996). "A new approach to the minimum cut problem." *Journal of the ACM*.

Randomized Incremental Construction of Convex Hulls

6.1 Abstract

This report investigates the efficiency and behavior of randomized algorithms in computational geometry, specificallt of the Randomized Incremental Construction (RIC) for Convex Hulls. We contrast this randomized approach with the deterministic Graham Scan algorithm to evaluate the practical implications of randomization. While Graham Scan offers a guaranteed $O(n \log n)$ time complexity, RIC leverages random input permutation to achieve a similar expected time complexity of $O(n \log n)$. Our empirical analysis demonstrates that while RIC performs exceptionally well on uniform random distributions, often surpassing the deterministic baseline, it remains vulnerable to specific worst-case structures, degrading to $O(n^2)$.

6.2 Introduction

Randomized algorithms often provide simpler and more efficient solutions to geometric problems compared to their deterministic counterparts by avoiding worst-case input configurations through random permutations. This is demonstrated perfectly by the Randomized Incremental Construction (RIC) algorithm for the 2D Convex Hull problem. The Convex Hull of a set of points is the smallest convex polygon enclosing the set.

The primary objective is to study the performance characteristics of the RIC algorithm. By comparing it against the standard deterministic Graham Scan algorithm, we aim to highlight the trade-offs between theoretical worst-case guarantees and expected behavior on average inputs.

6.3 Theoretical Analysis

6.3.1 Graham Scan

The execution of the Graham Scan is bipartite, consisting of a preprocessing phase and a traversal phase. We first select an anchor point p_0 (typically the point with the minimum y-coordinate) and sort the remaining $n1$ points angularly around p_0. The identification of p_0 requires $O(n)$ time. The angular sorting, utilizing any optimal comparison-based sorting algorithm, requires $O(n \log n)$ time. This establishes a lower bound of $\Omega(n \log n)$ for the preprocessing phase.

The traversal phase constructs the hull by iterating through the sorted sequence and maintaining a candidate hull on a stack S. For each point p_i in the sorted sequence, the algorithm enforces convexity by repeatedly removing the point at the top of the stack if the turn formed by the second-to-top point, the top point, and p_i is not a left turn (counter-clockwise). This local geometric predicate is computed in constant time.

To prove the linearity of the traversal phase, we employ amortized analysis. Let us account for the stack operations. Each point from the input set P is pushed onto the stack exactly once. A point is popped from the stack at most once; once removed, it is permanently discarded and never re-evaluated. Consequently, the total number of stack operations is bounded by $2n$. The total time complexity $T(n)$ is the sum of the sorting cost and the scanning cost, given by $T(n) = O(n \log n) + O(n) = O(n \log n)$.

6.3.2 Randomized Incremental Construction

We analyze the algorithm using the conflict graph framework. We maintain a bipartite graph G_r between the edges of the current hull $CH(Sr)$ (where S_r is the set of the first r inserted points) and the set of uninserted points PS_r. An edge $(e, q) existsin \mathrm{G}_r$ if the point q is visible from the hull edge e. The algorithm inserts p_{r+1} by locating it via G_r; if p_{r+1} has no conflicts, it is internal. If it has conflicts, the visible edges of $CH(S_r)$ are removed, and new edges incident to p_{r+1} are created. The cost of step r is proportional to the number of conflict pointers that must be updated, which occurs only for points conflicting with the newly created edges.

We employ backward analysis to bound the expected work. Consider the state after step r, where the hull $CH(S_r)$ is fixed. We view the transition from $r-1$ to r as the random insertion of p_r. Because the permutation is random, p_r is equally likely to be any of the points in S_r. A structural change occurs only if p_r is a vertex of $CH(S_r)$. The number of edges created at step r is equal to the degree of p_r in $CH(S_r)$. The expected number of structural changes is bounded by the average degree of a vertex in a planar triangulation, which is constant.

The dominant cost is the maintenance of the conflict graph. A point $q \in P$
S_r updates its conflict pointer at step r only if the hull edge it conflicted with at step $r1$ was removed (or rather, "created" in the forward view) by the insertion of p_r. For a fixed uninserted point q, let k be the number of edges of $CH(S_r)$ visible to q. These edges are defined by a chain of vertices in S_r. The conflict pointer for q changes only if one of the defining vertices of its conflict edge is p_r. The probability of this occurring is $\frac{2}{r}$.

Thus, the expected number of updates for a single point q is $\sum_{r=1}^{n} \frac{2}{r} = 2H_n$, where H_h is the n-th Harmonic number. Since $H_n \approx \ln n$, the expected cost per point is $O(\log n)$. Therefore, the total expected running time is $O(n \log n)$.

6.4 Implementation and Methodology

The algorithms were implemented in C++ to maximize computational efficiency, utilizing double-precision arithmetic for coordinate representation. A key challenge in geometric algorithms is handling floating-point precision; we addressed this by employing an epsilon threshold for all geometric predicates.

For the RIC implementation, a doubly linked list was chosen to represent the hull vertices. This structure supports $O(1)$ insertion and deletion operations, which are frequent during the update phase. The implementation performs a linear traversal of the hull to determine point visibility. While this simplifies the data structure requirements, it introduces a dependency on the instantaneous size of the hull. The Graham Scan implementation utilizes a standard vector and the standard library sorting algorithm to order points angularly, followed by a linear scan using a stack-based approach.

The experimental evaluation was conducted on a Linux environment using the GCC compiler with optimization level 3, as well as the Python 3.14 runtime, with plots generated using MatPlotLib. We evaluated performance on two distinct datasets: uniformly distributed points in a square, representing the average case, and points arranged on a circle, representing the worst-case scenario for RIC where every point belongs to the hull.

6.5 Results and Discussion

The empirical results reveal a stark contrast between the two algorithms depending on the input distribution. On uniformly distributed data, while both algorithms were demonstrably close to linear (theoretically log-linear, but the test sizes don't make it very visible), the RIC implementation demonstrated superior performance, executing in under 0.002 seconds for large inputs. This efficiency arises because the convex hull of uniform random points has a small expected size (logarithmic relative to n), making the linear visibility check negligible. The Graham Scan, while fast, incurs a fixed cost due to sorting, consistently taking approximately 0.003 seconds for similar input sizes.

However, the limitations of the naive RIC implementation becomes evident with the circular dataset. Since every point on a circle belongs to the convex hull, the hull size grows linearly with the number of inserted points. Consequently, the linear visibility check forces the algorithm into a quadratic runtime behavior, taking nearly 1.4 seconds for the largest inputs. In comparison, the Graham Scan maintained its stability, processing the worst-case input in approximately 0.004 seconds, consistent with its $O(n \log n)$ guarantee.

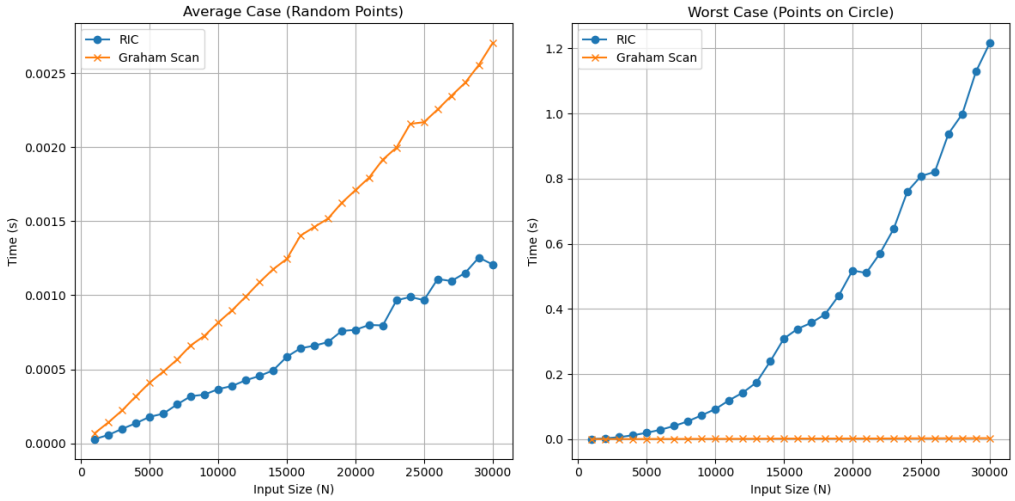

Figure 6.1: Runtime comparison: RIC vs Graham Scan on Uniform and Circular distributions.

6.6 Conclusion

The study confirms that randomized algorithms like RIC can offer excellent performance for typical inputs, often outperforming deterministic methods due to smaller constant factors and the low probability of expensive structural updates. In fact, theoretical results can be extended to show an even greater improvement in higher dimensions. However, the reliance on randomness does not eliminate the possibility of worst-case behavior if the underlying implementation does not account for specific degenerate cases. For general-purpose applications where input distribution is unpredictable but asymptotic time complexity is paramount (unlikely, but who knows), deterministic algorithms like Graham Scan provide a safer, more consistent performance guarantee.

Overall Conclusion

This comparative analysis demonstrates that while randomized algorithms inherently compromise on absolute certainty—either by gambling with time (Las Vegas) or accuracy (Monte Carlo)—they often provide solutions that are far more practical than their deterministic counterparts.

1. Simplicity vs. Complexity

Algorithms like Randomized QuickSort and Karger's Min Cut are conceptually simpler and easier to implement than their deterministic rivals (e.g., Median-of-Medians pivot selection or max-flow based min-cut algorithms). This simplicity translates to fewer lines of code and lower constant overhead, leading to faster execution in practice.

2. Avoiding Worst Cases

As seen with QuickSort and RIC, deterministic algorithms often have specific "Achilles' heels" (sorted arrays, circular point sets). Randomization effectively destroys these structures. By randomly permuting the input, we transform "worst-case inputs" into "worst-case random events," the probability of which becomes astronomically low.

3. Scalability in the Real World

In domains like cryptography, deterministic solutions are often theoretically possible but practically useless. As shown in our Miller-Rabin analysis, deterministic Trial Division hit a wall at 48 bits, while the randomized approach scaled effortlessly to 64 bits and beyond. Here, the trade-off of a 4^{-k} error rate is statistically negligible compared to the benefit of actually being able to compute the result. Additionally, computational geometry methods are commonplace in most modern computing applications, and the speed benefit provided by the Randomized Incremental Construction algorithm is a godsend.

Final Thought

Ultimately, this project highlights that "correctness" and "speed" are not binary absolutes. In modern computing, a solution that is correct with probability $1 - 10^{-20}$ and runs in milliseconds is often infinitely more valuable than a solution that is guaranteed correct but requires the lifespan of the universe to compute. Randomization is the tool that allows us to bridge this gap.